2OS

more programming from the machine up

Philip Machanick

Last typeset February 2, 2016

Contents

List of Figures

List of Tables

Definitions

A

absolute path – A path from the root of the file system, in Unix designated by starting with "/". See also *system path*, *relative path*, *path*.

abstraction – Hiding inessential details so the user or programmer need not worry about how something is implemented and can focus on how to use it.

access control list – An *ACL* indicates specific *permissions* for a list of users for a specific file or directory.

ACL – See *access control list*.

address space – Range of addresses as seen by a *process*. See also *virtual memory*, *virtual address space*. *physical address space*.

alias – Mac OS X equivalent of a *soft link*.

API – An *application programming interface* is a set of libraries providing user-level services.

application programming interface – See *API*.

B

backing store – Also called *swap* or *swap space*: slower device than RAM used as the lowest level of the *virtual memory* hierarchy.

bandwidth – Average work completed per unit time, usually measured in data transfers. See also *latency*, *throughput*.

barrier – A *synchronization* primitive that forces tasks to wait until a given number reach that point.

bit mask – A bit pattern used in bitwise logical operations to extract specific bits; often shortened to *mask*.

block – Minimum unit of allocation in a device or in a *cache*.

block-oriented – An IO device designed to transfer data in fixed-sized units.

buffer – Memory faster than a target or source device, used to bridge speed gaps between fast and slow parts of a system. Differs from a cache in being longer-term storage more under program control. See also *spooling*, *cache*.

C

cache – A fast memory that is used to fake the effect of the entire memory being faster than a reasonably affordable memory technology. Decisions as to what is in a faster layer are made in hardware. The fastest cache is integrated into the CPU in recent designs, and is the *highest-level* or *level 1* (also: *L1* cache). There can be 1 or more lower levels of cache, usually in current designs integrated into the CPU chip, numbered L2, ... See also *buffer*, *spooling*, *cache miss*, *cache hit*.

cache hit – Item accessed is in a given level of *cache*. See also cache miss.

cache miss – Item accessed is not in a given level of *cache*. See also cache hit.

central processing unit – See *CPU*, *core*.

cleaning – In *VM*, doing a *write back* so the *dirty* state of a *page* can be cleared.

cloud – Services available over the Internet with some variability as to whether they are *networked services* or *distributed systems*.

copy on write – A page table entry marked as *COW* indicates there is more than one virtual page referring to the same physical page, which must be copied to a new physical page once modified.

cooperative scheduler – A scheduler in which a process only gives up the CPU when it makes a call to an OS service. See also *preemptive scheduler*.

core – In designs with multiple *CPUs* on a chip (*multicore*), each CPU is called a core. Cores often share the lowest-level on-chip *cache*.

CPU – The *central processing unit* that contains the instruction-processing logic. See also *core*.

COW – See *copy on write*.

critical section – A point in code where there would be a *race condition* if access were not sequentialised using *synchronization*.

cylinder – A collection of *tracks* all the same radius from the centre of a disk on a disk with multiple platters. See also *seek*.

D

deadlock – Two or more tasks are waiting for each other and cannot progress. See also *livelock*.

determinism – In a scheduler, ability to predict an upper bound on an event. See also *realtime*.

device driver – Low-level software that controls an individual device such as a disk and presents a standard interface to the operating system.

directory – A logical unit of a *file system* that contains any combination of *files* and directories. See also *folder*.

dirty – State of any unit of the memory hierarchy where it has been modified relative to one or more lower levels. See also *cleaning*.

distributed system – A system where services have location-independent names and whether a service is local or remote is an implementation detail, not implicitly or explicitly named as remote or local. See also *networked service*, *cloud*.

DRAM – See *dynamic random access memory*.

dynamic linking – Linking that is delayed until a program runs. See also *linker*, *library*, *static linking*, *executable file*, *object file*.

dynamic random access memory (DRAM) – RAM usually implemented with a capacitor storing a bit that needs to be refreshed periodically to maintain its value: relatively inexpensive, but not as fast as *SRAM*.

E

embedded system – A computer that is part of another machine or device.

exception – See *interrupt*.

`exec` – In Unix-derived systems, replace the current *address space* by that of a given *executable file* and start running that executable. See also `fork`.

executable file – A file that can be run directly. See also *linker*, *object file*.

external fragmentation – Reduction in usable memory resulting from scattered freeing of used space. See also *internal fragmentation*.

F

FAT – See *file allocation table*.

fairness – In a scheduler: the property that all processes receive a share of the CPU. See also *starvation*.

FCFS – See *first come first served*.

fault tolerance – Probability that a system keeps running, irrespective of whether a fault occurs. See also *reliability*.

file – Logical unit of a *file system* that a user sees as a single entity containing information.

file allocation table – *FAT*: a way of keeping track of disk blocks making up a file as a linked list.

file system – Logical organisation of one or more devices (or one or more *partitions*) containing *files* and *directories*.

first come first served – In scheduling *FCFS* means processes are processed in the order they arrive.

flash – Type of *RAM* that is *non-volatile* and hence can be used to implement storage. See also *wear levelling*, *solid-state drive*.

folder – A *directory* as viewed in a graphical view of a *file system*.

`fork` – In Unix-derived systems, *spawn* a new instance of a *process*. See also `exec`.

forward page table – A *page table* that uses a part of the *virtual address* as an index to find the corresponding physical address translation. See also *inverted page table*, *multilevel page table*, *TLB*.

fragmentation – Reduced usability of a memory system arising from scattered usage or wasteful allocation. See also *internal fragmentation*, *external fragmentation*.

free software – Software that may be distributed free of charge and for which source code is also freely available. See also *open source*.

Free Software Foundation – (FSF) Promotes *free software*. See also *GNU*.

FSF – See *Free Software Foundation*.

function-like macro – See *preprocessor macro*.

G

global replacement policy – *Replacement policy* that considers all processes. See also *local replacement*.

GNU – GNU is Not Unix: software label of *Free Software Foundation*.

H

hard link – An alternative name for a file; in Unix-type systems usually implemented by more than one directory entry pointing to the same *inode*. See also *soft link*.

hard realtime – A realtime requirement that if not met means system failure. See also *realtime*, *soft realtime*.

header file – A file merged into another file by the *preprocessor*, before it is seen by the compiler.

heuristic – An informed guess: used when an exact algorithmic approach is either not possible or too inefficient to be practical.

hit – At any level of the memory hierarchy: the item being accessed is present at that level. See also *miss*.

hot swapping – Ability to replace a part without shutting down or data loss. See also *RAID*.

I

index node – See *inode*.

inode – Unix-style *index node* used to store file attributes and pointers to file blocks.

internal fragmentation – Loss of memory arising from allocation in fixed-sized units, resulting in more allocated than needed. See also *external fragmentation*.

interprocess communication – *IPC* is any mechanism that allows passing of information from one process to another.

interrupt – Event that breaks the sequence of execution, often resulting in use of a jump table to find an interrupt handler. See also *interrupt handler*, *jump table*, *trap*, *exception*.

interrupt handler – Code invoked to handle an interrupt. Generally must be short to minimise backing up other interrupts. See also *interrupt*.

inverted page table – A *page table* that uses a part of the *virtual address* as a hash index to find the corresponding physical address translation, with one entry per physical page. See also *forward page table*, *multilevel page table*.

IPC – See *interprocess communication*.

J

journal – A log used to recover a file system after an unclean shutdown. See also *journalling file system*.

journalling file system – A file system that logs transaction on disk in a *journal* so those not completed can be redone after an unclean shutdown. See also *logical journal*, *physical journal*.

jump table – Table of jump instructions that can be used to transfer control code based on an index. See also *interrupt*.

K

kernel – Part of operating system that has full access to all hardware. See also *microkernel*, *monolithic kernel*.

kernel mode – See *system mode*.

L

latency – Time to complete on operation. See also *bandwidth*, *throughput*.

library – Precompiled code available to link into programs. See also *linker*, *dynamic linking*, *static linking*.

linker – A program that combines separately compiler files. See also *object file*, *library*.

livelock – Two or more tasks cannot progress because of a mutual dependency that does not cause them to wait. See also *deadlock*.

local replacement policy – *Replacement policy* that considers only the current processes. See also *global replacement*.

locality – The principle that a program uses a small subset of memory at a time. See also *spatial locality*, *temporal locality*.

lock – A *synchronization* primitive that ensures *mutual exclusion*. See also *spinlock*, *mutex*.

logical journal – A log in a *journalling file system* that records metadata only. See also *physical journal*.

M

macro – See *preprocessor macro*.

mask – See *bit mask*.

memory protection – OS support usually in *virtual memory* for protecting processes against incorrect access by each other, or to memory regions that should not be accessible to that process.

microkernel – Minimal *kernel*, with features in it that cannot be implemented outside the kernel. See also *monolithic kernel*.

mirroring – Organisation of multiple disks that replicates data across each disk to improve speed, fault tolerance or both. See also *striping*, *RAID*, *hot swapping*.

miss – At any level of the memory hierarchy: the item being accessed is not present at that level. See also *hit*.

monolithic kernel – A *kernel* that implements all core functions within the kernel. See also *microkernel*.

multicore – See *core*.

multilevel feedback queue – In a scheduler, a version of a *multilevel queue* in which processes migrate between priority levels according to their behaviour.

multilevel page table – A *page table* that is split into levels to minimise the need to store translations for parts of the address space that are not used. Usually a *forward page table*. See also *inverted page table*.

multilevel queue – In a scheduler, the ready queue is divided into levels according to priority. See also *multilevel feedback queue*.

multitasking – Ability to run more than one process or thread simultaneously, switching use of the CPU between them.

mutex – Type of *lock* that puts waiting tasks to sleep and queues them to ensure *fairness*. See also *spinlock*, *synchronization*.

N

networked service – A service that is explicitly named as existing via a network. See also *distributed system*, *cloud*.

non-volatile – Of memory: does not require power to maintain its contents.

null pointer – A pointer value that represents no memory location, usually stored as a zero. See also *pointer*.

O

object file – A compiled portion of a program that must be combined with other files to make an executable file. See also *linker*.

open source – Alternative name for *open source*, favoured by those making economic rather than moral arguments.

P

page – Fixed-size logic unit of memory that is the smallest unit managed by most *virtual memory* systems. See also *segment*, *page fault*.

page fault – Occurs when a *page* is not in physical memory, and the OS must intervene.

page frame – A page as represented in *physical* memory. See also *virtual memory*.

page table – Data structure used to represent virtual to physical page translations. See also *forward page table*, *inverted page table*, *multilevel page table*, *page frame*.

path – Sequence of directory names, in Unix separated by "/". See also *system path*, *relative path*, *absolute path*.

PCB – See *process control block*.

permissions – Access rights to a specific file or directory. See also *access control list*.

physical address space – *Address space* as seen by the hardware: actual RAM addresses. See also *virtual memory*, *virtual address space*, *page frame*.

physical journal – A log in a *journalling file system* that records metadata and changed file contents. See also *logical journal*.

platter – Recordable surface of a disk.

portable – Designed for ease of implementation on multiple platforms.

Portable Operating System Interface – See *POSIX*.

POSIX – *Portable Operating System Interface*: libraries designed to standardise interfaces of Unix-style services for portability across Unix variants and other operating systems.

preemptive scheduler – A scheduler that can interrupt a process to pass control to another. See also *cooperative scheduler*.

preprocessor – A text processor that transforms source files before they are seen by the compiler. See also *preprocessor macro*, *header file*.

preprocessor macro – A symbol created using `#define` is replaced by the *preprocessor* wherever it occurs, by the rest of the text on the same line, before the compiler sees the code; a *function-like macro* has parameters.

priority – Any measure of adjustment of the order of processing that takes into account information about a given unit of work.

process – A program while it is running. See also *thread*, *task*.

process control block – A *PCB* is a data structure that keeps track of process state such as saved registers and open files.

processor – Logic unit that interprets instructions and includes the fastest layers of memory, registers and caches. Also called *central processing unit* (*CPU*). See also *core*.

Q

quantum – See *time quantum*.

R

race condition – An update of a shared variable depends on the order two or more *tasks* reach that point in the code. See also *critical section*.

RAM – See *random access memory*.

RAID – Redundant Array of Independent (formerly Inexpensive) Disks. Depending on RAID level, uses variants on *mirroring* and *striping*.

random access memory – *RAM* is any memory that has an addressing scheme that equally allows any item to be accesses without e.g., a delay to make that region accessible.

read – Access contents of memory or device. See also *write*.

realtime – A requirement that a task be done by a time deadline. See also *hard realtime*, *soft realtime*, *determinism*.

relative path – Path in Unix starting with anything but “`/`”, relative to the current working directory. See also *system path*, *path*, *absolute path*, *working directory*.

reliability – Probability that as system does not develop a fault. See also *fault tolerance*.

remote procedure call – *RPC* provides a wrapper around invoking remote services that looks like a function or method call.

replacement – At any level of the memory hierarchy, evicting a given *victim* unit to make space for a required unit not already present at that level.

replacement policy – Decision algorithm for choosing a *victim page* to evict from main memory in *VM*. See also *global, local replacement*.

RPC – See *remote procedure call*.

S

scheduler – Part of OS that determines which task to run next; can also refer to a disk scheduler.

seek – Movement of the disk head to the right *track* or *cylinder*.

segment – Logical unit of address space. See also *page*.

semaphore – A *synchronization* primitive that has a counter and a queue; decreasing the count blocks the task if the count ≤ 0.

shell – In Unix-like systems, the environment where you run programs including a scripting language.

soft link – A pointer to a file or directory; if the original moves, depending how it is implemented, the link may break. See also *hard link*, *alias*.

soft realtime – A realtime requirement that if not met can be handled by a fallback option like a drop in quality. See also *realtime*, *hard realtime*.

solid-state drive – An *SSD* is a disk equivalent made of electronic components, usually *flash*, without moving parts.

spatial locality – The principle that a program is likely to use memory close to a location that has been accessed some time soon. See also *temporal locality*, *locality*.

spawn – Create a new instance of a *process*. See also *fork*.

speedup – After a change, $\frac{t_{before}}{t_{after}}$. See also *Amdahl's Law*.

spinlock – Type of *lock* that spins on a shared variable. See also *mutex*.

spooling – Dumping of output to an intermediate device or buffer to ensure outputs to that device are not interleaved: mostly associated with printers. A whole word now but originally an acronym SPOOL for **s**imultaneous **p**eripheral **o**peration **o**ff-**l**ine. See also *buffer*, *cache*.

SRAM – See *static random access memory*.

SSD – See *solid-state drive*, *flash*.

starvation – In a scheduler: when a processes does not receive a share of the CPU. See also *fairness*.

static linking – Linking that is done when creating an executable file. See also *linker*, *library*, *dynamic linking*, *executable file*, *object file*.

static random access memory (SRAM) – RAM usually implemented with a transistor storing a bit that does not need to be refreshed periodically to maintain its value: relatively expensive, and as faster than *DRAM*. Also requires more components than DRAM per bit, and hence not as dense, which is why it is more expensive. Generally used for *caches*.

striping – Organisation of multiple disks so accesses use each disk in turn to gain speed. See also *mirroring*, *RAID*.

swap – Also called *swap space*. See *backing store*.

synchronization – Maintaining mutual exclusion across a *critical section*. See also *lock*, *mutex*, *barrier*, *semaphore*.

system mode – CPU state for kernel-level code with unlimited access to hardware. Also called *kernel mode*, *supervisor mode*, See also *user mode*.

system path – Sequence of path names, in Unix separated by "`:`" used to find executables run with no path name. See also *path*, *relative path*, *absolute path*.

T

task – A *thread* or a *process*.

temporal locality – The principle that a program is likely to use the same memory again some time soon. See also *spatial locality*, *locality*.

throughput – Average work completed per unit time, usually measured in progress of a process. *latency*, *bandwidth*.

thread – Separately scheduled component of a *process* in the same address space as the parent process. See also *task*.

time quantum – The *time size* that a *preemptive scheduler* allows a process before interrupting it.

time slice – See *quantum*.

TLB – See *translation lookaside buffer*.

track – A logical unit of a disk organised in a circle at a particular radius. If a disk has multiple platters, all the tracks at the same radius collectively form a *cylinder*. See also *seek*.

translation lookaside buffer – A *TLB* is a small subset of an active *page table* that can be looked up very fast.

trap – A kind of *interrupt* signalled explicitly by a program. See also *exception*.

U

Unix – Operating system designed in the 1970s and now the basis for various free variants including Linux; a play on the name Multics, a large complex operating system that preceded it.

user mode – CPU state for user-level code with limited access to hardware. See also *system mode*.

V

victim – Any unit of the memory hierarchy that is selected for *replacement*.

virtual address space – *Address space* as seen by a process in *virtual memory*. See also *physical address space*.

virtual file system – (VFS) A software layer that hides differences in file systems e.g. making remote file systems appear to be local.

virtual memory – (VM) OS support for address translation from a virtual to a physical address space. See also *locality*, *pages*, *memory protection*, *backing store*.

volume – Logical organisation of a part of a file system that can be part of a device or span multiple devices. See also *partition*.

VM – See *virtual memory*.

VFS – See *virtual file system*.

W

wear levelling – In *flash* memory devices: moving around contents that is detected to be frequently modified to avoid wearing out flash (since each location has a limited number of write cycles).

working directory – Directory relative to which paths are defined. See also *path*, *relative path*, *absolute path*.

working set – Minimum set of pages needed in main memory to avoid unnecessary page faults. See also *resident set*.

write – Modify memory or device contents. See also *read*.

write back – Clear the *dirty* state of any unit of the memory hierarchy by writing the modifications to a lower layer. See also *write through*.

write through – Avoid the *dirty* state of in the memory hierarchy by writing modifications to a lower layer immediately. See also *write back*.

X

XNU – Mac OS X *kernel*: stands for X is not Unix.

…To the Operating System

1 Introduction

An operating system is about as close to the machine as any code gets. Since the main function of the operating system (OS) is to hide the hardware from the user for a variety of reasons, you seldom get closer to the hardware in other kinds of program, particularly in an OS that protects the hardware from direct access.

Background *This material follows from* MIPS2C *and may later be combined to create a complete book,* MIPS2OS. *If you did not attend a prior course using the* MIPS2C *part, you can find it at* `http://homes.cs.ru.ac.za/philip/Courses/CS2-arch-C/`

In this book I approach the problem of understanding an OS from the point of view of a C programmer who needs to understand enough of how an OS works to program efficiently and avoid traps and pitfalls arising from not understanding what is happening underneath you. If you have a deep understanding of the memory system, you will not program in a style that loses significant performance by breaking the assumptions of the OS designer. If you have an understanding of how IO works, you can make good use of OS services. As you work through this book you will see other examples.

Key Abstractions

How does an OS hide hardware – and other details that are difficult for the user? It defines *abstractions* that create the appearance of a machine that is easier to use than the real machine. An abstraction is anything that hides *inessential detail*: things you do not need to know to make something work for you. In object-oriented programming, abstractions are implemented by hiding private details of a class, so you cannot rely on them to use the class. In an OS, abstraction is about hiding how the OS actually works so you can focus on using it without worrying about what is behind it. Abstraction allows you to focus on what is important to you and also is an important design tool, allowing the OS designer to change the inner workings as needed without breaking user-level programs.

For example, a user has the appearance of *files* and *directories* (often graphically presented as *folders*) that hide the raw disk – or other underlying device. A *file system* provides operations that hide the low-level structure of the device. Abstractions like this are not just a user-level convenience. If a newer better idea of how to implement files is discovered, an OS designer can implement this new design without breaking user-level code or the user experience because a properly implemented abstraction hides this sort of detail.

At a lower level, input and output (IO) are hidden by an abstraction layer that provides basic operations like opening a file, reading and writing, that can hide the fact that very different devices are involved. In a Unix-type system, the kind we mostly study in this book, IO abstraction allows devices like the screen and keyboard to be treated like files with particular limitations (e.g., you cannot write to the keyboard – not with useful effect, anyway). IO abstraction includes hiding the details of a network so it can be accessed at a relatively high level without having to know much about the underlying technology.

At the level of individual devices, a *device driver* is an extension of the core operating system functionality that allows a class of devices to be handled in a uniform way (e.g., all disks work pretty much the same way, if the device driver is done right and hides all the device-specific detail).

Another key abstraction is the memory model. To a user-level program, address space in most systems (at the low level: your programming language usually hides this in another layer of abstraction) is simply a sequence of bytes numbered from zero to the largest number that can be addressed by the hardware. The operating system (usually) converts this user-level view to a mix of different programs sharing memory, each with the illusion that it has the whole system to itself. User-level programs need not know how big the physical memory is: the OS takes care of managing the illusion that each program has a whole machine to itself that may or may not have more memory required than the physical main memory. The OS also ensures that a program cannot access memory outside its own space. This abstraction is called *virtual memory*. Aside from VM, there is also a hardware-implemented memory hierarchy from the fastest on-chip cache (made of static RAM, SRAM) to the main memory (made of dynamic RAM, DRAM). The speed gap between the fastest and slowest layers can be a factor of over a million. When we study memory hierarchy, it will be clearer how this can work without a huge slowdown relative to the fastest memory layer.

When a program runs, that program usually shares the machine's resources with other programs – not just memory, but also IO devices and the processor. The processor is also called the *central processing unit* (*CPU*). If a chip has more than one CPU, it is called a *multicore* system, and each CPU is called a *core*.

Once a program is running, it becomes a *process*. Sharing the CPU or CPUs is necessary for a number of reasons. First, there is a mix of requirements: most users have an interactive process that currently has their attention and others they need to keep running but do not immediately need to access. For example, if you are editing a text document, your mail program is still running and can report new mail, allowing you to change your focus. Another reason to run a mix of programs is *latency hiding*. Latency is the time to complete one operation. Some operations like disk access are very slow compared to instruction execution time, so a lot of CPU time would be wasted if every disk operation caused the CPU to stall. An illusion of progress can be maintained across very slow operations as long as there is other useful work to do. Latency is often in competition with the averaged rate of work completion (*throughput* when measuring progress of processes; *bandwidth* when measuring data transfers). You can minimise latency for a task of interest if you give it total use of the CPU, but you lose throughput because the CPU would be idle when that task had to wait for IO.

Related to managing processes is the decision as to which process runs next: *scheduling*. There are many approaches to scheduling, with varying strategies for balancing throughput and latency. Some IO devices also need a scheduler. A disk is slow to access and ordering accesses so those close together on the disk are handled together can give significant speed gains, for example. To the user, whatever scheduler is used, the effect should be as if each process was running to completion, each with its own machine – except where information may be shared.

The overall effect of these abstractions includes another really useful abstraction: *portability*, the ability to move code to a very different system ideally with at most a recompile.

As a consequence of all the levels of abstraction is that the computer at the programmer level looks a lot simpler than if you had to control the hardware directly, manage memory explicitly and control the way different processes shared the machine. Some of this helps at the level of ordinary users, who can use files, launch programs, etc., without having to know how any of this works – much as a driver of a car needs to know nothing of the underlying engineering or manufacturing that goes into designing and building a car. On the other hand, a driver needs a better understanding of how a car works than a passenger does. So there is a hierarchy of skill levels – just as with a car, a Computer Science graduate needs a better understanding of what is under the hood than a "passenger": someone who only uses a computer at a shallow level.

The take home message? *An OS hides hardware and low-level machine-specific software from the user to make it easier to use the system in ordinary user-level code, to protect against errors and to make it easier to write portable code.*

Origins of C and Unix

Early operating systems were completely written in assembly language. That made it easy to address hardware directly, but coding of such a large system in assembly language is difficult and error-prone. It also limits an OS to specific hardware, contrary to the goal of portability.

Very early computers had small memories and processors very slow by today's standards, so an OS could not be very large. But as speeds and memory sizes grew, the potential for a larger, more complex OS grew. To give you some idea, in the 1970s, a machine with a megabyte of RAM was a large system, far bigger than, for example, a machine owned by a research group doing their own projects. When Unix was first designed, it had to fit on a machine with a few thousand bytes of main memory.

The Unix system was named as play on the name of a bigger more complex system, Multics, that preceded it. A small group at Bell Labs[1], the research lab owned by the then telephone monopoly in the US, American Telephone and Telegraph (AT&T), had a smallish system on which they wanted an OS convenient for research. They started with an assembly-language implementation, but decided to design their own language that could access hardware, with minimal assembly language. This language, C, was based on other earlier languages. Its oldest predecessor, CPL, was a large, complex language that was never fully

[1]Despite being owned by a commercial entity, Bell Labs employed first-class academics, the kind who win Nobel Prizes, who had a lot of freedom to work on open-ended problems. Their most famous invention is the transistor. The name Unix is sometimes spelt in all capitals but it is not actually an acronym, since the letters do not stand for anything. It had an earlier name, UNICS (**UN**iplexed **I**nformation and **C**omputing **S**ervice) – noting that Multics stood for **Mult**iplexed **I**nformation and **C**omputer **S**ervices – but "Unix" has become a word in its own right.

implemented. A simplified version, designed for systems implementations, Basic CPL (BCPL) followed [Richards 1969]. C was inspired by BCPL, but added back features left out like floating-point numbers [Ritchie et al. 1978; Kernighan and Ritchie 1988].

Two things ensured the popularity of Unix. Bell Labs made source code available at low cost to academic projects, and its relatively straightforward implementation in C made it relatively portable as well as relatively easy to build on in research projects. It was only later, in the 1980s, that the free software movement, largely inspired by Richard Stallman [Stallman 2002], led to widely available completely free source code, but "inexpensive" works as well as "free" when all the competition is either expensive or not released as source.

A note on naming: Stallman prefers "free software" ("free" as in "liberty", not as in "free beer") rather than "open source". The open source movement emphasises the utility of making source code available – multiple eyes checking code for example – whereas the free software movement is motivated more by the philosophy of sharing than the putative efficiency of making source code available. "Free" does not necessarily mean free of cost: you can charge for "free" software or services related to it, as long as you make the source code available and do not stop others giving it away.

The free software movement uses GNU as a label for its software projects. Aside being the English name for a wildebeest, GNU stands for **G**NU is **N**ot **U**nix – a recursive wordplay. You study computer science to appreciate things like that.

The take home message? *C is closely related to Unix as its original high-level implementation language, and has remained the language of choice for implementing Unix-derived operating systems, even those written from scratch, such as the Linux and Free BSD kernels, which are usually packaged with GNU software.*

Types of Operating System

Not so long ago, there were two kinds of computer well known to the average user: desktops (remodelled sometimes into portable versions – called laptops and notebooks – really the same thing, if with a design focus more on low power use than speed to allow for working off a battery) and servers. Mobile devices – smart phones and tablets – are a specialization of portable computers, designed to work in smaller memories and to run longer off batteries, with addition of phone functionality. They nonetheless (mostly[2]) draw on older consumer operating systems. The two most popular, iOS and Android, are based on Unix underpinnings. Two other whole categories of OS, *embedded* and *realtime* systems, are less well known, but in very wide use.

Realtime systems are in two flavours: *hard realtime* and *soft realtime*. A hard realtime system has to respond to a requirement in a specified time, otherwise it is broken. An example is a computer controlling a car's anti-lock braking system. If it does not respond in time, the car could crash. A soft realtime system must also respond within a time deadline but is allowed to fail – possibly with some permitted performance degradation. An example is a music or video player. Up to a limit where it may irritate the listener or viewer, some glitches are acceptable. Soft realtime is not too hard to add to a conventional OS as we will see when we look at scheduling, but hard realtime is difficult to add after the event because guaranteeing response time is difficult if a given level of throughput of regular applications is expected, and a realtime system does not work well on top of a hierarchical memory system with huge variations in memory latency.

An embedded system is one where a computer is part of another machine, e.g., a computer that controls an appliance like a washing machine, a computer built into a car to manage the engine or anti-lock braking or a computer built into a factory. Embedded systems often are not upgraded their entire lifetime, and therefore need not have much flexibility in design, but need to be able to run for a long time without maintenance. Depending on the application, an embedded system can have a cut-down version of a conventional OS (such as a Linux variant – even versions of Windows are common in things like ATMs), a realtime OS or a special-purpose highly simplified OS.

In the consumer space, we tend to think of operating systems as being from a rather small selection. The Windows family is in one corner and, in the other, a variety of systems with roots in Unix. These include Linux, FreeBSD, Mac OS X, iOS and Android. In fact there are many embedded and realtime systems and some ship in an enormous number of devices but you do not see the OS because the software is completely hidden from the user. Many modern appliances and cars contain computers – sometimes multiple computers. Because these are built into the machine and have no software visible to the user, they are in effect invisible. Since they are part of a particular machine, they have a limited range of functions and so can be of specialised design, and to not need to support a huge number of consumer apps. A popular mobile device or desktop computer, on the other hand,

[2]Blackberry 10 is an exception: it is based on QNX, a realtime operating system.

has to be general-purpose and support programming a wide variety of user-level software.

The take home message? *There are many operating systems in devices you wouldn't think of as computers – embedded systems. Realtime operating systems are specialised and there are many of them. The desktop, server and mobile computing worlds are a tiny fraction of the total both in variety and number of devices. The number of operating system variants is much smaller where the main point is user-installable software because issues like compatibility limit market share.*

What I Cover

Operating systems is a vast subject to cover. Here, I look at enough to give a taste of the subject, and to relate it back to programming in C and how C relates to the machine code level. I look at how an operating system is divided between a kernel and user space, scheduling of processes and threads, how IO and files are organised, memory management, and examine how process and threads are implemented and used at user level. As an extension of processes and threads I also briefly outline another mode of parallelism: distributed computing and how it relates to the cloud.

The goal is to give you a general understanding of operating systems with examples mostly from the Unix family, with occasional comparisons with other types of system. I do not go deeply into internals: the idea is that you should become more proficient in using and working with an OS rather then learn how to implement one. If you want to implement an OS, this book will give you a start but you will need to learn a lot more.

Exercises

1. There are many types of free software licences.
    (a) Look up variants on the Free Software Foundation's (FSF) General Public Licence (GPL), generally used with GNU software, as well as the MIT and BSD licenses.
    (b) What are the practical differences?
    (c) How does open source differ from free software? Are the differences significant?

   (d) Can you think of a practical reason for making source code freely available, besides having more people available to debug code?

   (e) Why is software not always published with a free license?

   (f) If you want to publish software you have created, explain how you would go about choosing a license model.

2. An operating system is about abstractions. Discuss each of the following, and how they make an OS more convenient to the user (or programmer).

   (a) Virtual memory.

   (b) Processes.

   (c) File Systems.

   (d) Input and output.

   (e) Device drivers.

   (f) Scheduler.

3. Having seen a little assembly language, would you like to write a whole operating system that way? Does C seem more suited to the task? Explain.

4. How many examples of embedded systems can you think of? Look around your home: what could contain a computer?

5. Is an MP3 player an example of a hard realtime system or a soft realtime system? Explain.

6. Is the anti-lock braking system of a car an example of a hard realtime system or a soft realtime system? Explain.

7. The principle of *locality* is that a program uses a small subset of memory at a time. Explain how locality could make VM viable, even though the fastest layer of the memory hierarchy is a million or more times faster than the slowest layer.

8. A disk is millions of times slower than a CPU. What tricks can we use to disguise this fact, so disk access does not slow the system to a crawl?

9. You are implementing an ATM system for a bank from scratch. Give points for an against each of the following approaches to choosing an OS:

   (a) Write your own OS from scratch.
   (b) Use a version of Linux.
   (c) Use Micrsoft Windows.
   (d) Use Mac OS X.
   (e) Use iOS.
   (f) Use Android.
   (g) Investigate embedded OS options for one suited to the task.

10. Is portability a more important issue for consumer apps than for embedded devices? Explain.

2 The Kernel

MMOST OPERATING SYSTEMS are structured as a layer that has total control over all hardware, the *kernel*, and at least one other layer, the *user level*, that is restricted in what it can access. Where systems differ is in how big and complex the kernel is. A *monolithic kernel* contains everything that needs direct hardware access including *device drivers* and can include quite large components like the graphics subsystem. At the other end of the scale, a *microkernel* does the absolute minimum that has to have total access to all hardware, and even low-level components that need direct access to hardware can be outside the kernel, provided their access to hardware can be limited to a very specific part of the system.

2.1 System Calls and IPC

Since a kernel provides access to parts of the hardware that are not accessible to ordinary programs, there has to be a mechanism for asking the kernel to provide a service not possible to implement at user level. As we will see later, memory management in most systems limits direct passing of information, so switching in and out of the kernel is necessary to pass information around.

Inter-process communication (IPC) is a general term for communicating between processes. However IPC is implemented, some kernel intervention is required unless the processes concerned are specifically set up to have a shared resource they can use, like a shared region of memory. In general terms we will assume IPC has to go via the kernel. In order to understand how IPC can work, we need first to understand what a system call is and how it works, since this is a mechanism to switch control to the kernel.

Hardware Support for Modes

Since some things are only allowed in kernel mode, to enforce this, hardware support is necessary. There is a lot of difference between different CPU architectures; we will look at a generic case, rather than go into detail of any specific machine.

A common approach is to have a *status register* that keeps track of the mode the system is in. In addition to keeping track of the mode the system is in, the interrupt handler needs information about what caused the interrupt. A register that contains a value identifying the cause is one approach to this. The MIPS processor has several registers in "coprocessor 0" that are used to keep track of details of interrupts, including the machine address where a bad memory access occurred.

Heads up: *For simplicity, we talk loosely of a "status register" but in a real machine, there may be more to machine state than that, as illustrated with the MIPS example.*

The status register can generally only be set in user mode by an interrupt. When an interrupt occurs, control switches to a predefined point in the operating system (depending on the type of interrupt) and the status register is switched to kernel mode.

To exit kernel mode, the kernel must first restore any registers and any other machine state specific to the interrupted process, then an instruction for returning from an interrupt sets the status register back to user mode, and jumps to the location containing the next instruction in the user process that should run to restart the interrupted process. That "next" instruction can be the one after the one causing the interrupt if that instruction completed (e.g. if it was a trap, such as a `syscall` – see below), or it could be the instruction that caused the interrupt if it did not complete. An example of an instruction that should be restarted after an interrupt completes is one that required some intervention from the memory manager before it could execute to completion.

The instruction to exit kernel mode cannot be executed in user mode: any attempt to do so should trap to the kernel and the operating system will usually treat this as an error and kill the process. One exception: if you are running another operating system either in an emulation or virtual mode, the kernel may treat such an instruction as part of emulating another kernel, and handle it more benignly.

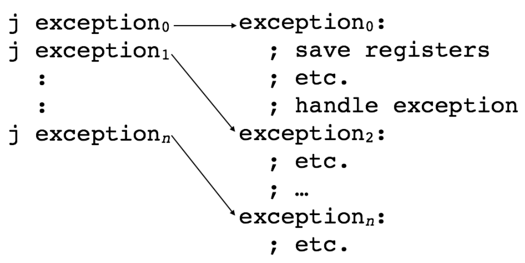
The take home message? *Hardware support for modes prevents a user-level process from taking actions that could violate security.*

System Calls and Interrupts

To enter kernel mode, it is necessary to signal to the CPU that a change in state from user mode to kernel mode is required, and the method for entry to the kernel has to be controlled so that it does not expose a security hole.

On the MIPS processor, a system call is invoked via the `syscall` instruction, which requires a value in the register `$v0` to indicate which system call is invoked. You can also pass a value in if required by the system call through the usual register conventions for parameter (argument) passing. In the SPIM simulator, we have a very approximate implementation of `syscall` to enable simple functions like printing. On a real machine, the `syscall` instruction breaks execution of the user-level program and passes control to the operating system, which has to interpret the `syscall` instruction and register values to work out what to do next.

A `syscall` is a special case of in *interrupt* (called an *exception* in the MIPS architecture) – an event that breaks control from going to the next instruction (or may even stop an instruction from completing), and takes you to a specific location in memory belonging to the kernel. An interrupt that is generated specifically and directly by an instruction is called a *trap*. Other interrupts can be caused by external events, like a timer going off, or by a failure of an instruction to complete (an error, or something the operating system needs to handle). The result is the same in all cases: the machine switches to kernel (sometimes called *supervisor*) mode and control transfers to a location specific to that type of interrupt. In the simplest model, there are sequential locations, one for each interrupt type, and each location contains a jump instruction to the place the interrupt is actually handled. Such a scheme is called an *exception vector*. MIPS is a little more complex: each exception location is 256 bytes long (space for 64 instructions), but the same basic idea applies – if an exception needs nontrivial code, its handler must branch out of the location where it starts. Allowing space for 64 instructions means simpler cases can be handled quickly; wasting most of 256 bytes if a jump instruction is all that is used is a small overhead compared with the flexibility to handle simple cases quickly.



```
j exception0 ──→ exception0:
j exception1        ; save registers
  :                 ; etc.
  :                 ; handle exception
j exceptionn      exception2:
                    ; etc.
                    ; …
                  exceptionn:
                    ; etc.
```

Figure 2.1: Basic Interrupt Vector. Some schemes as in MIPS put more space between locations but the idea is the same: an interrupt takes you to a specific memory location in kernel mode.

Heads up: *A system call has to generate an interrupt because that is the only way to get from user to kernel mode. A user-level process has to give up control completely before the kernel takes charge otherwise it could violate security.*

To return control to the user-level program, the OS needs to know where it was when interrupted, and to be able to restore register values. Restoring register values is not a very different problem to what you do when you return from a function call at machine code level except that for an interrupt, the program does not necessarily know it has been interrupted, so all register saves and restores must be done by the kernel. Also, there is no guarantee that the interrupted process will be the next to start, so registers will need to be stored in a data structure specific to the process. When a process is restarted, the kernel has to restore registers from this data structure. This data structure, commonly called a *process control block* (*PCB*), must contain enough information to restart a process. A PCB generally contains the information needed to control the lifetime of a process. That information at minimum includes all registers (excluding any only accessible by the kernel) and is likely also to include information about or at least a pointer to the information about resources owned by the process including open files.

Heads up: *When an interrupt occurs, a process generally cannot know that it needs to save and restore registers so it is the kernel's responsibility to return things to the state when the interrupt occurred. This is different from a function call, which is a more symmetrical contract between caller and callee.*

Two MIPS machine registers, `$k0` and `$k1`, a reserved for kernel use. There is nothing to stop you using these in ordinary code (they are just general-purpose registers) but because they could change any time out of control of your code using them is not a great idea. The MIPS convention of reserving these two registers for the kernel allows an interrupt that can be handled quickly in simple code that needs only two registers to get away without saving registers. This situation obviously only applies to interrupts that pass control straight back to the interrupted process.

In very simple cases, an interrupt handler may be able to hand control back to the interrupted process, in which case these reserved registers come in useful. Otherwise, the whole set of registers has to be dumped to the process's PCB so they can be recovered when the process is restarted.

The take home message? *An interrupt is necessary to transfer to the kernel so a user-level process cannot access any part of the machine that it is not entitled to see or modify.*

System Mode

Given all this machinery, there is one other essential requirement: a way of keeping track of whether the CPU is in user or system mode. In user mode, the operating system limits access to hardware. An ordinary user-level program cannot access all memory or directly access an IO device on most systems. Without hardware support to prevent a user-level program from accessing all parts of the system, out of control code could crash the whole system, or security holes could be exposed. Most CPUs have two privilege levels, system and user, and a status register that keeps track of which mode the system is in. A user-level process can only enter system mode by some kind of interrupt, but the kernel (or any code in system mode) can transfer control to user-level code.

For ordinary user-level processes, the term *user mode* is in common use. There is a bit more variation in *system mode*. System mode is also called *kernel mode* or *supervisor mode*.

IPC

On now to basics of IPC. We will see more detail later when we study processes (chapter 6), where we look more specifically at mechanisms in Unix-like operating systems. For now, I provide a brief overview.

The simplest model of IPC is placing information in a shared region of memory. If one process modifies a region of memory visible to another and the other periodically checks for updates, all that is required to implement this form of IPC is the ability to share memory across two or more processes.

Another model is passing a *message* through the kernel. That means copying data from the one process to the other, with a transfer into kernel mode to initiate the transfer. Message-passing IPC can be made very efficient. The L4 microkernel, initially a research project but now deployed on over 1.5-billion devices [Open Kernel Labs 2012], illustrates this by implementing IPC that is 20 times faster than that of an earlier attempt at a microkernel, the Mach project [Liedtke 1993].

Copying from one process to another, as we will see when we study memory in more detail, is complicated by the fact that memory protection prevents processes from seeing each other's address space.

The take home message? *Speed of IPC is a critical factor in the viability of microkernels, less so for performance of monolithic kernels, because a microkernel achieves protection by placing devices and services in separate address spaces.*

2.2 Kernel or User Space?

So what should be in the kernel and what should be in user space?

An absolutely minimal kernel is designed on the principle that something should only be in the kernel if that is the only way to make that feature work. Take for example a device driver, the software that controls a device like a disk or a network interface. Should that be in the kernel? Some aspect of it must be controlled by the kernel because it may need to have access to a particular region of memory not accessible to normal user processes, or be required to respond to a particular kind of interrupt. In a microkernel, some aspects of setting up a device driver and passing it an interrupt would be in the kernel but the main body of the code would be another user-level process. In a microkernel world, efficient IPC is really important otherwise talking to device drivers would be prohibitively slow.

I already mentioned the L4 microkernel, which is one of the smallest microkernels. Slightly bigger is the MINIX 3 kernel, used in operating systems education [Tanenbaum and Woodhull 2006] and research. An earlier version of MINIX inspired the development of Linux. Linux is most decidedly not a microkernel: device drivers, for example are part of the kernel.

MINIX 3 has slightly different design goals than L4. While L4 was designed

Table 2.1: Sizes of kernels

Kernel	size
Linux 3.16	6.4MB
L4	100kB
Minux 3.3	187kB

with performance and correctness in mind, MINIX 3 is designed for *fault tolerance*. Because major components of MINIX are outside the kernel, if they crash they can theoretically be restarted [Herder et al. 2006]. Since MINIX 3 is also designed to aid teaching about OS, it is coded relatively simply, while L4 is coded for performance, including highly machine-specific machine code to implement its most performance-critical features.

Table 2.1 shows kernel sizes on Intel x86 architecture for a few examples; the size on a RISC architecture would be bigger since RISC processors like MIPS has less dense code. These sizes are the file size of the compiled kernel. Actual run time memory footprint is considerably larger as the kernel has data structures that can be quite large.

To illustrate how complex the kernel can be in a system that has evolved over time, the Mac OS X kernel has multiple layers:

- *Mach* – manages IPC, process scheduling, memory and other low-level functions
- *BSD*[1] – networking and Unix-like system calls
- *networking* – builds on the BSD layer
- *file systems* – supports multiple types of file organization including Apple's own HFS+ and several Unix-derived variants
- *I/O kit* – infrastructure for building device drivers
- *kernel extensions* – components to add functionality such as new network capabilities that can be loaded without rebuilding the kernel

[1]Berkeley Software Distribution: a variant of Unix developed at University of California, Berkeley. Noted for the quality of its network implementation.

The Mac OS X kernel is called XNU, for “X is not Unix”. Since the “X” is a roman numeral ten, it is not clear how XNU should be pronounced.

To answer the question posed by the section title: it very much depends. In an OS like Linux, OS X or Windows, the kernel includes a lot of functionality including device drivers. In an OS like Minix 3 or L4, the kernel is a very small piece of code.

The trade-offs are:

- *speed* – a bigger kernel means fewer trips in and out of the kernel and hence less overhead
- *footprint* – a smaller kernel uses up fewer system resources such as faster levels of memory (caches); this can compensate for more trips in and out of the kernel
- *reliability* – if the kernel is smaller, it is easier to be sure it is correct; the L4 kernel has been proved mathematically to be correct [Klein et al. 2009], something you cannot do with a piece of code as big as the Linux, Windows or Mac OS X kernel
- *fault tolerance* – if services and drivers are outside the kernel as in L4 and Minix 3, the system can in principle be designed to recover from faults in this kind of code whereas any crash in the kernel is hard to recover

The Mach project was one of the earlier microkernels and because it delivered poor performance, the view arose that microkernels were inherently slow. Later projects like L4 and Minix 3 have to some extent addressed that issue. L4 is able to run the Linux kernel as a user-level task by handling interrupts in L4 and passing back anything that Linux has to handle with a performance penalty of about 2% [Härtig and Roitzsch 2006]. Linux in this case is being run in a kind of emulation mode to support backwards compatibility and a 2% performance hit is not bad in that scenario.

The take home message? *A microkernel is inherently slower than a monolithic kernel because it requires more switches in and out of kernel mode. That performance cost can to some extent be overcome by very efficient IPC and by the kernel itself having a smaller resource footprint than a monolithic kernel. As system requirements evolve, microkernels may become more popular.*

2.3 What the Kernel Does

Now we have some idea of the design choices, what are the essentials of what a kernel does?

First, it must manage processes. A process is the embodiment of a program when it runs. It has memory, a share of the CPU and may also get to control specific files or other machine resources. The main process-related kernel functions we will examine are *scheduling* – deciding which process runs when – and IPC. We in addition look at how processes can cooperate using IPC and synchronisation primitives, and how multithreading differs from coordination of multiple processes sharing a specific workload.

Second, it must manage memory. Memory management cannot be seen in isolation from managing processes because launching a new process requires an efficient strategy for allocating its memory and some types of IPC work through shared memory.

Third, it must manage input and output. IO involves devices that vary a lot in how they are controlled, and outer layers of the operating system (not the kernel) try to make them look more uniform. The kernel handlers the lower-level details, particularly responding to interrupts, and managing the way memory interacts with IO.

The take home message? *The kernel has certain core functionality – managing processes, memory and IO. The extent to which the detail of this is in the kernel or outside defines the difference between a monolithic kernel and a microkernel.*

Exercises

1. The original Mac OS and MS DOS, the text-only predecessor of Microsoft Windows, ran all code in one mode (no separate user and kernel modes).

   (a) Both were originally designed to run only one program at a time. Discuss whether in such a system a separate kernel mode is useful.

   (b) Early versions of Windows and the Mac OS evolved to run multiple programs simultaneously still without hardware separation of user and kernel mode. Discuss why the OS designers may have made this choice and positive and negative consequences.

2. The Mac OS X kernel is a composite of the Mach kernel and several layers. Discuss why it was designed this way, rather than using Mach as a microkernel.
3. Explain why efficient IPC is such a big factor in the viability of a microkernel.
4. Explain why the MIPS processor allows 256 bytes for each location in the interrupt vector when an instruction is only 32 bits (4 bytes).
5. A typical current generation CPU has about 64KiB of first-level (L1), i.e., fastest, cache. Discuss how a very small microkernel may have less impact on overall system performance than much bigger kernel, taking into account memory hierarchy.
6. The L4 and Minix 3 kernels place device drivers in user space. Discuss how such a device driver can work when some hardware it has to access can only be reached in kernel mode.
7. If a device driver crashes, the system has to be rebooted. True or false? Discuss.
8. A system with protection from badly behaved user code needs hardware support for kernel mode. Explain what has to be done to enter or leave kernel mode to avoid security violations.
9. Explain what has to go into a PCB to make it possible to restart a process. Some detail we need is missing; focus on what we have already covered.
10. Did the Mach project advance the popularity of microkernels? Explain.

3 Schedulers

DECIDING WHICH PROCESS or – sometimes which thread – to run next is a core function of most operating systems. Only very simple devices that run the same process all the time do not need a capability of switching between processes. Deciding which process or thread should go next is called *scheduling*. Scheduling is important because a good approach maintains a balance between allowing processes that are computation-heavy to make good progress, while allowing those that are IO heavy to make up for lost time.

To keep things general, I adopt the Linux terminology of *tasks*, which can be either processes or threads where I do not need to distinguish the two.

Managing the balance between requirements for different kinds of tasks is a moving target. Early operating systems, designed for computers with much smaller memories than we generally have today, had an easier task as a limited number of processes could be active at any time. With memories in multiple gigabytes, a hundred or more processes can be active at once and a poor approach to scheduling could make for a system that is unresponsive for interactive use or that does not allow fair progress of longer-running, less interactive tasks.

Schedulers split into two broad categories: general-purpose and realtime schedulers, with other subdivisions as we will see shortly.

A general-purpose scheduler aims to handle a range of workloads; a realtime scheduler aims to provide time guarantees for processes that have a realtime requirement. Realtime further divides into two categories: hard realtime requires that a given transaction complete within a given deadline otherwise the system is broken. Soft realtime softens this requirement to allow an approximate version of the transaction if time runs out.

An example of a hard realtime requirement is an anti-lock braking system on a car. If the calculation is not done in time, the braking system does not work as designed and is flawed. An example of soft realtime is video streaming. If the sound breaks up a bit or the picture pixellates, within some tolerance of the user, the system is not broken.

Soft realtime can be implemented in a general-purpose scheduler by giving realtime processes a higher priority than normal so they have a high probability of running when they have to and completing a task in time, provided they are programmed sufficiently efficiently. Hard realtime cannot be implemented on top of a scheduler not designed for guaranteed response times.

Hard realtime is a specialised subject and designs can be very specific to the particular application domain. I mainly described general-purpose systems here. To start, I present some theoretical variations, then make them more practical. I then describe how some of these general concepts apply in particular systems.

Another variant on scheduling is a *long-term* scheduler, which decides whether to admit a process to the system, and a *short-term* scheduler, which decides which of the currently active processes should run next. The long-term scheduler is usually very simple and lets any new process move to the ready queue immediately, unless the system would be overcommitted for memory or the number of allowed active processes.

A process can become inactive if waiting for a resource, e.g., an IO event. In its simplest form, a scheduler has three queues:

- *admission* – processes that have not yet been started
- *ready* – processes that can run but are not active
- *waiting* – processes that cannot run until a resource constraint like waiting for an IO operation is lifted

Corresponding to these queues, a process can be in one of four states, as illustrated in figure 3.1:

- *not admitted* – in the admission queue
- *ready* – as above
- *waiting* – as above
- *active* – the active process on a given CPU

On systems with more than one CPU (multicore systems, for example), there may be more than on active process. Such systems may also have separate queues for each CPU to avoid bottlenecks in accessing a single central queue of each type.

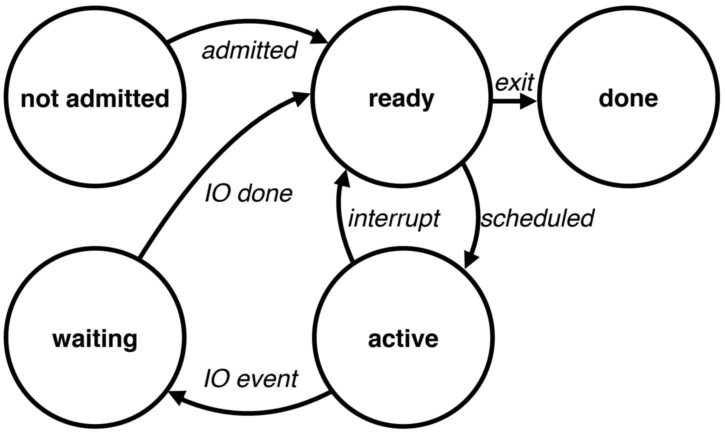


Figure 3.1: Process states. Circles are states and arrows are labelled with reasons for transitions between states. An interrupt that does not cause an IO wait puts an active process back into the ready queue.

Another less common distinction in schedulers is *preemptive* versus *cooperative*. A preemptive scheduler interrupts a task either when it runs out of allocated time, the task itself causes an interrupt (e.g., a system call, or something the memory system must pay attention to), or some external event interrupts execution. Another task can be scheduled at that point, and the interrupted task resumed later. A cooperative scheduler relies on the user-level code to invoke an operating system service that signifies it can give up control at that point. Early versions of the Mac OS and Windows used cooperative scheduling. In general, a separate kernel makes it possible to do preemption; any OS where user-level code can directly access OS data structures cannot safely implement preemption, as an interrupt may leave an OS data structure in an inconsistent state.

3.1 Theoretical Approaches

Most schedulers in common use are *preemptive*, i.e., a process does not run as long as it likes but is forced to give up the CPU after a time interval if it does not do so itself. To simplify initial presentation, I describe approaches first without preemption.

Non-preemptive scheduling

First, what do we look for in a scheduler? The most significant properties are:

- *fairness* – no process should experience *starvation*, where it gets no share of the CPU and ideally each process should receive an equitable share of the CPU

- *responsiveness* – processes should not wait an unacceptable time to make progress, especially those that interact directly with the user

- *minimal wait time* – a process would spend as little time as possible in queues, when it could be ready to run

Related to fairness is the expectation that a process will make progress, so a process that waits a lot of the time for IO, for example, should get enough use of the CPU to compensate as far as possible for time lost to waiting.

The absolutely simplest scheduler is *first come first served* (*FCFS*). An FCFS scheduler runs each process to completion in the order it arrives. FCFS has two advantages: simplicity and fairness. While it may not be the most efficient at minimising wait time and responsiveness, it is very simple to implement. A queue in the order of arrivals allows the scheduler to pick off the job that has been in the system longest and schedule it.

In practice, FCFS is unlikely to be used for anything but the long-term scheduler, since real schedulers use preemption, and waiting for IO events soon makes order of initial entry to the system irrelevant as IO-bound processes drop behind and small CPU-intensive processes complete.

Another theoretical approach is *shortest job first*. The value of *SJF* is that it minimises wait time because a long-running process waiting for all the smaller processes only sees their run time as wait time, whereas anything waiting for a long-running process sees its lengthy run time as wait time. SJF can also make for a responsive system, since user interactions may be short-lived processes compared with e.g. running a large weather model. SJF has two problems: it is not possible to implement, because it requires future knowledge of how long a process will run, and it can result in starvation. SJF is nonetheless useful in simulation studies to compare against other approaches to measure the trade-off in minimising wait time with SJF versus a more practical algorithm.

The take home message? *SJF minimises wait time, but is not a practical strategy as it requires prediction of future run time of each process.*

Preemptive scheduling

A real process very seldom starts from the beginning and runs to completion because almost all processes do some IO, and IO is so slaw that a process waiting for IO should give up the CPU to another while it waits. A scheduler could let each process run until it has to wait for IO but that would be unfair on processes that do a lot of IO. They would spend a lot of time waiting not only for their IO to complete, but for any long-running compute-intensive process that grabbed the CPU while they were waiting for an IO event.

Early versions of the Mac OS used a form of non-preemptive (cooperative) scheduling where a process could lose control of the CPU when it invoked one of a list of operating system services (these were not systems calls, because the OS did not have a separate kernel). The theory was that developers would frequently use system services related to the user interface in programs that were designed to be interactive. A program that did not behave cooperatively or that ran into a bug like an infinite loop could hog the CPU in this model of scheduling.

A *preemptive scheduler* grabs control of the CPU away from a process without waiting for it to give up control. A process can lose the CPU as a result of any type of *interrupt*. In general, an interrupt is any event that breaks the flow of execution. Most interrupts are caused by events outside the currently running code but a *trap* is an interrupt that is specifically signalled by the code, most commonly in a modern OS to cause a system call. Other kinds of interrupt include:

- *timer* – a specific time interval has elapsed and the program is interrupted: this can be because the program itself has set a timer, or because the scheduler has set a timer
- *page fault* – an attempt at a memory access fails in a virtual memory system (see Chapter 5)
- *IO event* – an IO event has started and needs attention, e.g., a network packet has arrived
- *IO completion* – an IO event has finished, and has been set up to interrupt when it completes
- *protection fault* – attempt to access or modify hardware state not allowed at the current protection level

- *floating point exception* – obtaining a result or using value that is invalid e.g., dividing by zero

In some cases, interrupts (like a protection fault) are errors and your program will be terminated by the OS. Recoverable kinds of interrupt like page faults, timers and IO-related interrupts result in the process being suspended. Depending on the reason for suspending the process, the process's state changes from running to either ready or waiting, depending on whether the interrupt causes an IO event for the current process.

A preemptive scheduler will generally set a timer for a particular *time quantum* or *time slice*, the maximum time that a process can run. If it does not complete or there is no other interrupt but the time the timer goes off, the process loses the CPU and the scheduler picks the next ready process to run.

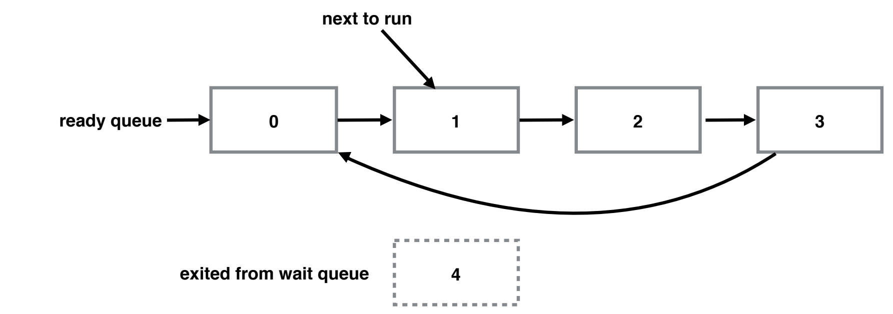
One of the simplest models of preemptive scheduling is a *round robin* scheduler[1]. A round robin scheduler organizes active processes in a circular queue (a data structure in which the last entry points back to the start) and as each process gives up the CPU without losing its ready status, the next ready process becomes the active process. Any process that rejoins the ready queue can be inserted at the "head" of the queue (the next location to be processed), giving any process that has had to wait a burst of CPU time to make up for lost time, as illustrated in figure 3.2 (ignore the bird[2]).

A round-robin scheduler ensures fairness but it does not assist IO-bound processes with making progress as the only way they catch up is by being put at the head of the queue when their waiting ends. If the process has a burst of CPU activity that exceeds the time quantum, it in effect goes to the back of the ready queue, so this is not really a significant advantage.

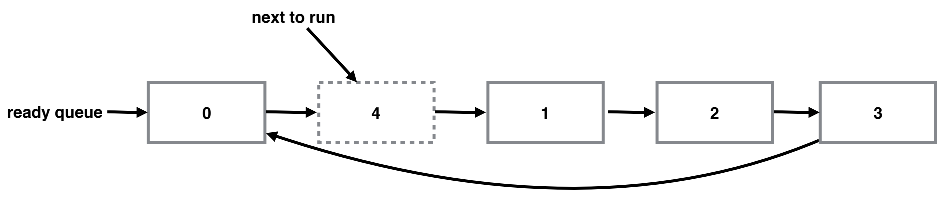
The take home message? *The round robin approach provides a basic model that is a starting point for a practical scheduler but it does not have the flexibility to deal with a diverse workload.*

[1] Aficionados of wildlife will be wondering what a fat bird has to do with scheduling. The term "round robin" originally applied to a petition written on a circular ribbon, so ringleaders could not be identified and punished by a vengeful monarch. The "robin" part is a corruption of the French for ribbon, "ruban". What does this all have to do with schedulers? Nothing, except the origin of the name, which has also come to mean everyone taking their turn one after another.

[2] Source: Pixabay, `http://pixabay.com/en/red-robin-bird-animal-winter-157576/`.



(a) About to add previously waiting process



(b) Peviously waiting process added

(c) A fat bird

Figure 3.2: Round-robin scheduler. When a process ("4" here) exits the waiting queue, it becomes the next process to run as illustrated. Otherwise each process runs in turn, with a circular queue formed by linking the last to the first entry.

3.2 More Practical Approaches

More practical approaches to scheduling have to take into account the mix of a typical workload, which can include IO-bound and CPU-bound processes, as well as processes where the IO component is mostly user interaction. If IO is mostly user interaction, a particular concern is making it appear responsive to the user.

As speed and size of computers vary, the workload mix also varies. Since rapid advances in cost lowering make bigger and bigger configurations more commonplace, adjustments to design trade-offs can be necessary. Here, I examine general principles and look in the next section at specific examples to show how rapidly designs can evolve.

The most important addition to the basic round robin scheme is *priority*. Priority is any scheme that adjusts the order of processing to take into account differences in units of work. A process that is IO bound should in principle have higher priority than a CPU-bound process, while an interactive process should have higher priority still. Soft realtime can be accommodated by giving realtime processes higher priority than others; hard realtime is more difficult.

There are various schemes for taking priority into account. The simplest is to have different ready queues for different priorities: this approach is called a *multilevel queue*. The next process scheduled is the next one in the highest-priority ready queue that is not empty. In this scheme, starvation is a risk because processes in lower priority queues will never be scheduled if there are always processes in higher priority queues. Starvation is addressed by a *multilevel feedback queue*, in which processes can migrate between priorities. A common approach is for a process to drop a level (go to lower priority) each time it finishes a time quantum without being interrupted and to go up a level (higher priority) each time it re-enters the ready queue after an interrupt is processed. In such schemes, it is also common to limit the number of levels a given process can migrate up or down in priority, and adjusting the place a process starts and how far it can migrate are tuning parameters. For example, a soft realtime process may start at the highest priority, and be limited in the number of levels of priority it can drop to avoid failing to respond in time. Figure 3.3 illustrates the basic idea, without complications like limiting the range of variation of priority of a given process.

Another approach to varying a simple round-robin scheme is taking into account each task's share of elapsed CPU time as a measure of how much progress each task has made relative to others. The idea starts from the notion that if a CPU could somehow magically allow all ready processes to run at once, dividing CPU time equally between them, you would have a fair scheduler. Since this is not possible, a measure of progress based on dividing actual elapsed CPU time by number of ready processes, called *virtual CPU time*, is used to estimate elapsed time across all processes. An approach based on virtual CPU time can replace priorities by ensuring that the process with the shortest elapsed virtual CPU time goes next. For this to make sense, any new process arriving in the system needs to have its elapsed time initialised based on the shortest elapsed virtual CPU time of any task already in the system, otherwise it could hog the CPU until it caught up. This scheme can give tasks weights corresponding to priorities that make lower-priority tasks exhaust their allocated run time faster [Pabla 2009].

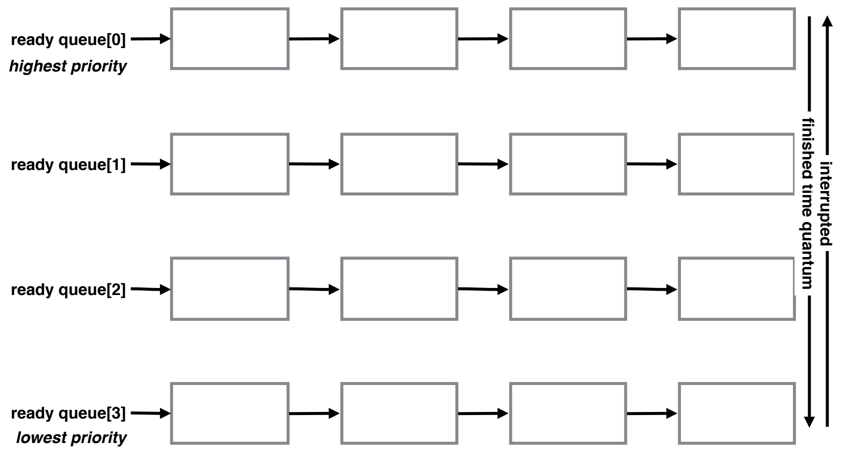


Figure 3.3: Multilevel Feedback Queues. If a task uses up its time quantum it goes to a lower-priority queue. If interrupted, it resumes in a higher-priority queue.

Another issue with real schedulers is dealing with competing users. If one user e.g. schedules 100 tasks and another schedules 10 tasks, should the first user get 10 times the CPU time of the second user? A related issue is where the OS supports threads. If a group of threads is scheduled completely independently, they may get held up if a result needed by one thread is not ready when the others need it. Treating all tasks equally without taking cooperative and competitive effects into account may not always give the best result.

Gang scheduling attempts to schedule a group of related tasks that have dependences between each other as a group. On a multiprocessor system, that implies scheduling them at the same time so any bottlenecks arising from one task waiting for a result from another are eliminated. Although originally developed for large-scale multiprocessor supercomputers [Feitelson and Jettee 1997], gang scheduling has in recent times also been explored as an option for cloud computing [Moschakis and Karatza 2012].

Implementing hard realtime is very different. Each task has to have a time deadline, and has to be scheduled taking into account its predicted run time and time deadline. Dynamic priority changes in a realtime scheduler break *determinism*, predictability of time to complete [Schaffer and Reid 2011]. Systems differ: the time deadline may simply be a property of determinism and the design of the particular process, i.e., if it has time *N* to complete a task, it is coded to take less than that time and relies on the scheduler to prioritise it over non-realtime tasks, and to schedule it without delay if an external event triggers it.

Getting hard realtime right goes beyond scheduling: it creates big challenges for the memory hierarchy as a factor of a million or more difference in timing would really matter for hard realtime, so a page fault in a hard realtime process would almost certainly result in missing its deadline; a delay of a few tens or hundreds of clock cycles to handle a cache miss is less problematic but makes coding for the general case difficult. An *embedded system*, which contains a computer as part of a machine, appliance, etc. is often custom-designed with a few additional requirements as possible to avoid problems with determinism.

The take home message? *Priority can be implemented directly with multilevel queues and indirectly with virtual CPU time. Hard realtime presents another whole set of problems and cannot easily be handled in the same framework as regular scheduling.*

3.3 Examples

Evolution of the Linux scheduler is an interesting case study because there have been major changes in strategy. Windows provides a contrast of a relatively stable design. I contrast two Linux schedulers, the $O(1)$ scheduler and the *completely fair scheduler* (*CFS*), to illustrate how fast the free software world can move.

To understand the design issues, you need to be clear on notation used in algorithm analysis for predicting scalability. I present a very approximate reminder here presuming you will know this from prior study of algorithms; any good algorithms book should explain the notation properly [Cormen et al. 2009]. We care about how fast the function describing execution time grows (and sometimes memory use, if that grows significantly more than the original data). We do not care as much about constant factors or lower-order terms. A function like $12N^2 + 2N + 4$ will be bigger than $1000N\log N + 10^6$ if N is big enough. So we focus on the largest term in the function using the "big-O" notation. The first example is $O(N^2)$ and the second is $O(N\log N)$. We also do not concern ourselves with the base of a log, because you can covert between log bases by multiplying by a constant. The best we can do in general is constant time, i.e., time that has a fixed upper bound, which we write as $O(1)$. A big-O function is used to characterise *time complexity* – in essence a prediction of how fast time grows as N grows. An $O(N^2)$ algorithm, for example, takes 4 times as long (to a good approximation, since we dropped lower-order terms) every time we double N because $(2N)^2 = 4N^2$.

Heads up: *Algorithm analysis teach us that, unless we are dealing with very small N, there is little point trying to improve an algorithm whose time complexity predicts it scales poorly. Rather, we need to find a better algorithm with lower time complexity.*

Prior to the 2.6 version of the Linux kernel, Linux used an $O(N)$ scheduler [Aas 2005, p 15], i.e., one in which at least some operations scaled linearly in time required with the number of active processes. A number of factors make it useful to have a scheduler (and in general, other kernel operations) scale better than $O(N)$. With growing memory sizes and CPU speeds typical users even with a desktop system have a growing number of active processes. With support in the kernel for threads, the number of tasks (threads or processes) to be managed grows even more. In a big server, even more active tasks may exist.

In kernel version 2.6.23, the $O(1)$ scheduler was replaced by the *completely fair scheduler* (*CFS*). While CFS takes time $O(\log N)$, this function grows so slowly that even for very large values of N efficiencies introduced by the CFS make the tiny extra overhead worthwhile.

Linux $O(1)$ scheduler

In algorithm analysis, $O(1)$ time means the upper bound on execution time is a constant, and does not vary as problem size N varies.

Here are a few details of the $O(1)$ scheduler, considered so good at the time that there was little room for improvement [Aas 2005]:

- there were 140 priority levels, each represented by a run queue
- only tasks in the highest-priority queue with runnable tasks are scheduled
- when a task uses up its time slice, it is moved to an expired priority list and at the same time, its time slice is recalculated
- when the ready list is empty, a pointer swap exchanges the roles of the ready and expired lists for that priority
- tasks have a *static priority* assigned at launch (called a *nice* value in the Unix world, with a range of -20..19, and a default value of 0)
- as a task runs its *dynamic priority* is adjusted with a range of ± 5 from the static priority (smaller values are higher priority)

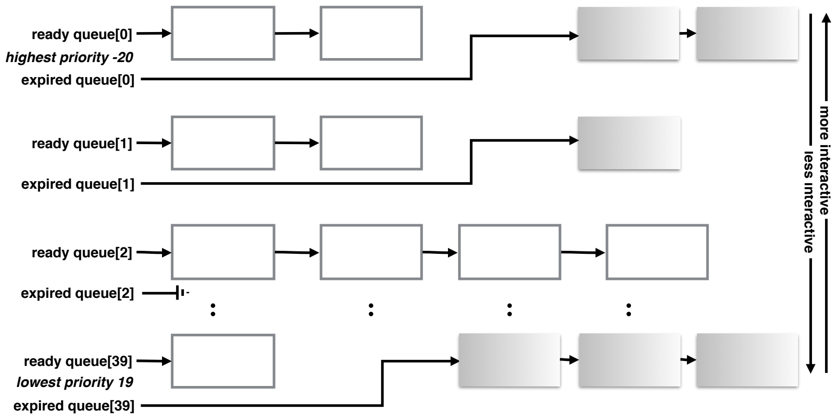


Figure 3.4: Linux $O(1)$ Queues. When a ready queue empties, the ready and expired pointers are swapped. Heuristics adjust priorities up and down based on "interactivity" of tasks.

All these operations are possible to do in a bounded amount of time. Finding the highest priority non-empty queue is done by keeping a bitmap in which a 1 at bit b means queue b has at least one entry. Finding the position in a bitmap with the lowest-numbered set bit, if there is an upper bound on the number of bits (here, 140 – rounded up to a whole number of machine words), this operation can be found in $O(1)$ time. Moving a task to an the expired list and recalculating its time slice can also be done in a constant amount of time: a pointer swap to can be done in $O(1)$ time. Figure 3.4 summarises the major features.

This then is an efficient implementation of a multilevel feedback queue; the complication comes in the feedback part, where priorities can be adjusted according to how interactive a process is. The degree of interactivity is estimated from how much sleep time a task racks up. A highly interactive process should in principle spend a lot of its time sleeping: waiting for IO, particularly from really sluggish devices like humans. Since this is not something that can be calculated accurately, the $O(1)$ scheduler uses complex *heuristics* to work out how interactive a given task is, and uses this calculation to adjust its priority up or down. A heuristic is an informed guess – in general, an approximation technique you use when an accurate answer is either not possible or too inefficient to calculate.

The $O(1)$ scheduler includes other details like preventing a user from creating multiple new tasks that take advantage of apparently "interactive" behaviour to gain an unfair share of the CPU.

The take home message? *The Linux $O(1)$ scheduler was much more efficient than its predecessor and needed constant time as opposed to $O(N)$ time to choose the next task to run at the expense of complex heuristics to adjust priorities.*

Linux CFS scheduler

In practice, though the $O(1)$ scheduler worked well and was more efficient than its predecessor, finding correct formulations for heuristics turned out to be difficult, and unwanted behaviour could result. The heuristics were also complex to calculate, to some extent negating the value of the other efficiencies of the scheduler. For these reasons, CFS was created [Molnar 2007], and first appeared in the 2.6.23 version of the Linux kernel.

The CFS is based on attempting to simulate the effect of an idealised multitasking CPU, where all tasks ready to run at the same time run simultaneously, hence having a fair share of the CPU. A real CPU cannot run a multitasking workload that way, so CFS attempts to fake this by keeping track of the amount of time each task has previously run, and allowing the task with least elapsed run time to go next to catch up. This run time is measured as *virtual runtime*, actual time scaled by the number of active tasks, to simulate the effect of each task getting a fair share of the CPU. A task is allowed to run until its virtual runtime puts it ahead of the next-worst-off task, when it goes back into the queue. The actual amount it is allowed to run is a little past the point where it has had more of the CPU than the next task to schedule, so each task gets a reasonable stretch of runtime before losing the CPU.

The trick used to make all this possible without unacceptable overheads to find the next task is each task is put into a *red-black tree*, a binary search tree ordered the usual way, but kept balanced as nodes are added or removed. Keeping the tree balanced ensures that all operations are $O(\log N)$.

Whenever a task becomes active, its virtual runtime is reset to a new value based on the smallest in the tree (with a slight adjustment so it and the current most deserving task can run for a reasonable stretch).

CFS implements priorities by weighting each task, based on its static priority. Lower-priority tasks in effect run a faster virtual clock, so their realtime clock times out faster.

Current versions of the Linux scheduler implement a mix of schemes including CFS. What is interesting is the relatively rapid evolution: in 2005, the $O(1)$

scheduler was well-established as efficient and hard to improve. In April 2007 Ingo Molnár[3] announced the first release of CFS.

The take home message? *The Linux CFS scheduler is marginally less efficient than its predecessor at choosing the next task – $O(N \log N)$ time – but does away with the need to calculate complex heuristics to adjust priorities.*

Windows scheduler

The Windows scheduler by contrast is relatively stable. The original Windows scheduler, reflecting the transition from MS DOS and the relatively primitive CPUs available at the time for personal computers, had two layers of scheduler. It had a preemptive scheduler that could switch CPU use between a DOS and a Windows virtual machine, and a cooperative scheduler to manage Windows applications [Pietrek 1992].

The Windows NT scheduler [Russinovich 1997] introduced a modern preemptive scheme with support for threads. To be different from the Unix tradition, a higher priority is a higher number. NT assigns priorities at thread granularity, with priority 0 reserved for a system idle thread, which executes when nothing else is able to run. Priorities 16 to 31 are for realtime tasks, and administrator privileges are required to access these priorities. As with Linux implementations of a multilevel feedback queue, NT tasks priorities vary within a range. On NT's case, that range is defined by a *priority class*, which can be one of realtime, high, normal or idle. When a time quantum expires, a task is interrupted or another task's state changes from waiting to ready, the scheduler decides which task to run next. If a task has switched from waiting to ready, it runs next if it is the highest-priority ready task, otherwise the next ready task with highest priority runs next.

Dynamic priorities adjust within the allowed range (e.g., a high priority thread cannot decay to normal or rise to realtime): if a thread finishes a time quantum, it goes down by 1. If an external event corresponding to user interaction like a mouse click completes, the receiving task receives a big boost in priority. A tasks can increase its priority repeatedly through a series of such events until it reaches the maximum for its class.

NT also has an anti-starvation mechanism. The OS periodically scans the

[3]`http://lwn.net/Articles/230501/`

ready queues for tasks that have not run for at least 3 seconds. Any such task is temporarily given the highest priority of its class and is put on a queue of tasks to be scheduled; it runs just that time with double the usual time.quantum.

Later Windows schedulers have not changed much in basic approach, though details such as handling of multiprocessor scheduling have been refined [Russinovich et al. 2012, Chapter 5]. Microsoft in 1999 decided to drop the NT name, since it required payment of royalties to Northern Telecom [Smith 2010], but current Microsoft kernels are based on the original NT design.

The take home message? *The Windows scheduler is classic example of multilevel feedback queues. The basic design has been stable over a long time, with fine-tuning of details as requirements have evolved.*

open versus closed

Aside from the fact that Linux kernel development is open to everyone because of free availability of source code, another key philosophy difference between the two kernels is the way they cater for different market segments. From the start, Microsoft differentiated the server and desktop versions of NT, and more recently Microsoft further subdivides the space into server, desktop and mobile versions. While the Linux scheduler architecture is designed to be modular, allowing the possibility of changing the scheduler for different niches, there is no marketing-driven pressure to enforce artificial differences. For this reason, the default Linux scheduler needs to work well for scenarios like a computer that doubles for desktop and server use. Microsoft, on the other hand, is able to tune the scheduler differently (e.g., adjusting the time quantum) for different markets, even if the underlying code does not change.

The take home message? *Linux and Windows kernels have both been very successful and operate on scales from small custom devices to large servers. The free software philosophy and lack of market pressures to constrain design have resulted in greater flexibility in development of the Linux scheduler, while Microsoft has relied on their ability to tune the scheduler to particular market niches to adapt to change.*

Exercises

1. Explain why shortest job first is impractical, even if approximated.

2. In a real system, is first come first served practical? With preemption, what is the nearest equivalent?
3. The early Mac OS used *cooperative scheduling*, where a program could only give up the CPU if it called one of a number of OS services. Discuss strengths and weaknesses of this approach.
4. Aside from accessing a file in user-level code, what else could cause an IO interrupt? Would any of these causes of an IO interrupt result in any significant difference in any OS activity other than handling the interrupt and moving the interrupted processes from ready to waiting, then back to ready?
5. Explain why a round robin scheduler is not a good solution to scheduling a mix of IO-intensive and CPU-intensive tasks.
6. Explain why implementing hard realtime is difficult in a general-purpose operating system.
7. Explain how virtual CPU time takes into account long-running processes when sheduling a new task.
8. A process is split into 4 threads cooperative threads that exchange information on a regular basis. On a quad-core machine (4 cores), explain how gang scheduling could be useful in this scenario.
9. Explain the key differences between the Linux $O(1)$ scheduler and the Linux CFS scheduler.
10. A new OS is being designed and has to support hard realtime processes as well as a Linux-compatible application layer. Comment on the strengths and weaknesses of each of the following approaches:
    (a) Add hard realtime as an extra scheduling option on a Linux kernel
    (b) Start from a barebones kernel like L4, add a hard realtime scheduler with application support and run Linux as in emulation mode for the Linux application layer
    (c) Design a realtime kernel from scratch, port the Linux application layer and system calls, and include lower-priority scheduling for non-realtime processes

4 IO and Files

INPUT AND OUTPUT is an area critical to the performance of a computer system. A disk can be a million or more times slower than the average time to process an instruction; network latency is of a similar order. Devices using faster storage like flash reduce the scale of the problem, but IO remains a bottleneck. The operating system has to address this bottleneck by a variety of tricks and techniques all of which add up to *latency hiding* – the basic time for a single IO operation is not improved by these things, but is hidden so it is either less obvious or obscured entirely.

How can this be done?

First, most operating systems do not run a single program at a time. By using *multitasking*, an operating system can ensure that there is some work available to do rather than stalling for an IO operation.

Second, if the operating system is aware of the device characteristics, it can use tricks like clustering multiple nearby accesses to reduce the penalty of setting up an access. This trick works because many devices take a lot more time to set up an access than to transfer data but accesses to similarly located regions can be faster if they are grouped together.

Third, tricks specific to an operation can speed it up. For example, there is no need for a process to wait for a *write* to a device to complete because the process doesn't need the result, so the write can be dumped to faster memory and completed in the background while the process continues. This technique is called *buffering*. A buffer can also be used for *reads* – but for a different purpose. By exploiting the fact that it is quicker to read a big chunk of data in one go than with a sequence of small operations, buffering for reads works by reading more than the single request into memory, in the expectation that more of the neighbouring contents will be needed soon. This is an example of the *locality* principle. A buffer is usually in main memory (RAM).

A variant on the buffer idea is *spooling*. Originally an acronym (SPOOL,

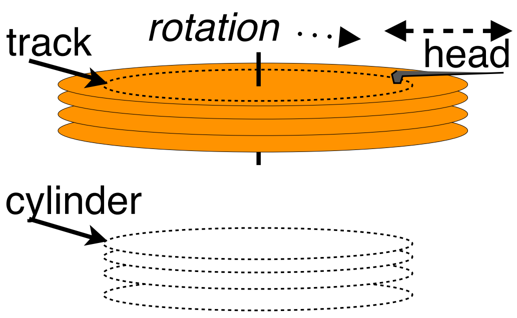


Figure 4.1: Conceptual layout of a disk. Tracks and cylinders are marked using magnetic storage and are not actually visible. An optical drive stores information by changing the reflectivity of a spot on the disk. A cylinder is a logical grouping of tracks through vertically stacked platters.

for simultaneous peripheral operations on-line), spooling is a kind of buffering that applies to devices like printers that cannot interleave operations. If you print a document, you expect the entire document to go through the printer without being mixed with other print operations – even if someone else (or even you if you are impatient) sends something to the printer before your document is done. Waiting for a printer is much the same issue as waiting for a write to a disk – there is no need for the printing process to stall until printing finishes. The only real difference is that where spooling is involved, the OS has to take a full print job as an entity, which requires a bit more management than ordinary buffering. Spooling can also use a slower device like a disk, since the delay for a printer is long by standards of computer speeds.

We look at some if this separately when considering processes and scheduling; the main focus here is the lower level, up to the file system, which provides a common abstraction for multiple device types.

4.1 Device Interface

At the lowest level, each hardware device could work differently. A disk requires its head to move to or from the centre of the disk in a *seek* to find the right *track*. Most disks have multiple *platters*, so tracks are grouped logically together vertically as a *cylinder*. Once at the right track or cylinder, the disk has to rotate to the right position for the required access. Flash is completely different: it is a special kind of RAM that doesn't lose its connects when power goes off. Unlike main memory, flash is much slower for writes than for reads, and can wear out if the same location is frequently modified.

For a disk, seek time is a major part of the time for any transaction, and rotational delay is not far behind. Unless data is transferred in a really big quantity at once, transfer time is a smaller component of total time. Reads and writes take much the same time, unlike flash, where there is a big penalty for writes.

These and other differences have to be hidden from the programmer, including programmers who are coding higher-level parts in the OS, otherwise every new device type would require extensive recording. Also, the device-level operations are not convenient for programmers, who want to see a device in terms of logical units, not low-level locations. At the lowest level of the programming, those logical units can be blocks on disk (seen as an array indexed from zero, much as bytes are addressed in RAM). Mostly, programmers do not need to go to such a low level, and see a disk organised as a *file system*, in which the logical units are *files* and *directories* (*folders*, in a graphical viewer). The file system can also be organised into *volumes*, a kind of virtual disk, that can be part of a disk, or even span multiple disks.

The very low level device-specific detail is hidden by a *device driver*, code that provides standardised operations to the operating system so similar types of device can be handled in similar ways. Above the device driver level, the operating system still has to know what type of device it is so it can offer appropriate operations, but the device driver takes care of details like how to find a given byte on the device. Despite the abstractions provided by device drivers, the operating system still needs to know if the device can support operations like random access, read and write. Devices also may be character-oriented (like a keyboard) or block-oriented (like a disk). Here is a summary of some of the variations:

- *read* – ability to retrieve data on a device; most devices are readable but there are exceptions, like printers and the screen
- *write* – ability to modify data on a device; most devices are writeable but there are exceptions, like a CD ROM or a keyboard
- *block-oriented* – ability to transfer data in big chunks; a disk is a good example
- *character-oriented* – ability to transfer data a character at a time, such as a keyboard

- *sequential* – access limited to going from one end to the other, like a tape
- *random access* – access can be to any location without a time penalty (though in practice, some devices like disks do score if you access regions close together, to reduce seek and rotational delay)

And here are some of the performance parameters for different some common device types:

- *disk*
  - seek time – time for the head to move in or out to the right track; usually measured in milliseconds (ms)
  - rotational delay – time for the disk to move the right area to the head as it spins; on average, half a rotation
  - transfer time – number of bytes transferred divided by transfer speed
- *flash*
  - read delay – measured in microseconds (μs)
  - write delay – measured in milliseconds but faster than disk; slower than reads

Transfer time for flash and disk vary a lot more than the setup time to access a specific location, since that depends on the type of interface.

To the user, a file is a single entity that contains a specific kind of information like data or code; that it is implemented using *blocks* that may be randomly scattered over one or more physical devices is hidden by lower-level abstractions.

Some systems implement a *virtual file system* (VFS) that can make remote and local devices or even file systems spanning multiple devices look the same to the user as a single local device. Sun Microsystems (now part of Oracle) implemented an early version of a VFS for their Network File System (NFS), now common on other Unix-style platforms, to make remote disks look like local disks as far as possible at the programmer level [Sandberg et al. 1985].

There is significant overlap between the issues concerning file media and networks, including tricks for hiding latency. Since networks are a large subject on their own, I leave them for others to cover in detail. When reading about disks and flash devices, bare in mind that similar issues apply to networks.

4.2 Files and Devices

The Unix operating system (and descendants including Linux and Mac OS X) abstract the properties of a device by treating it as a file wherever possible. A keyboard or screen, for example, is a special kind of device. A keyboard can be read from in and is character-oriented. A screen in its simplest form can only be written to as a character-oriented device (graphics is handled differently: in a modern system what used to be a screen is now often a terminal program).

Files nonetheless are a higher-level abstraction than a logical way of handling devices consistently. The file system as a whole has a structure, usually hierarchical, and allows you to have logical names for files without having to know how they are represented and even on what media they are stored. The storage medium only matters in terms of details specific to it. Some variations include:

- *removable* – can be logically and probably physically ejected
- *read-only* – such as prerecorded CDs or DVDs (whether with media or regular data content)
- *write-once* – such as some kinds of writeable CD

These variations, like other properties of a device, should not effect the way you treat a file if it doesn't run into a medium-specific limitation.

The hierarchical structure of a file system divides into directories, and can also subdivide a disk or collection of disks into logical *volumes*. A volume is a logical unit that you may think of as a single disk even if it is actually part of a disk or may span multiple disks. In the Microsoft world, volumes are distinguished by a single-letter label; most other file systems allow more generous and flexible naming.

One of the most important aspects of a file system is the mapping from the logical file to the physical device. There are two methods in common use: a *file allocation table* or *FAT* and variants on the Unix inode (index node) structure. An inode can point to the start of a file or a directory, and a directory in a FAT system is represented in a special file. FAT was used in older Microsoft systems and is still in wide use in portable devices.

In recent years Microsoft has migrated from FAT file systems to NTFS (New Technology File System) but maintains compatibility with FAT devices because they are in such wide use. Portable flash drives or USB sticks commonly usually use FAT because they are relatively small and portability across different systems is an important consideration in this case.

FAT has two major drawbacks. Because the FAT is stored in memory, it is a big overhead for very large devices. Random access for large files is inefficient, since the entry for each block must be checked to find the *n*th block.

If for example a block size is 8KB, a 1TB disk would require a FAT with 125-million entries. If each entry was 4 bytes, that would be 500MB of RAM just for the FAT. While RAM is increasingly inexpensive, this would be a huge overhead – on a machine with 8GiB of RAM, the FAT would take up about 6% of total RAM.

FAT and a basic inode system both have the drawback that the system can become damaged by an out of control reboot or power outage.

What I aim to do here is provide a sense of how you weigh up alternative approaches by contrasting the main ideas behind FAT file systems and inodes. I do not cover full details of implementation; there are many variations in real systems accommodating growth in device sizes since each approach was originally designed.

FAT

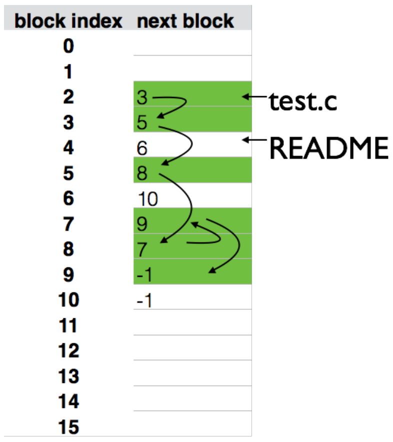
The essential idea of FAT is to keep a linked list in RAM representing the blocks (called *clusters* in the DOS and Windows world, because each logical block is a group of lower-level device-specific blocks) on the device. The linked list is stored as an array indexed by block number so if you can find each block quickly in the table if you know its block number.

This table is a copy of the same structure on disk, where it remains available across reboots or power failures, so changes must be copied back to disk.

To find a file, you need to know its starting block. From there, each successive block is found by a pointer into the table (stored as a block number rather than a pointer, so the table need not be modified each time it is stored in a different region of memory).

Figure 4.2 illustrates a simplified FAT representing two files, `test.c` and `README`. In this simplified FAT, there are 16 blocks, and a invalid block number, -1, is used to mark the last block of a file. Each entry in the table contains a number indicating the next block.

In a real FAT, the lowest-numbered blocks may be reserved for system use, and the directory structure has to be represented in the file system so individual files can be found.



block index	next block
0	
1	
2	3
3	5
4	6
5	8
6	10
7	9
8	7
9	-1
10	-1
11	
12	
13	
14	
15	

Figure 4.2: Conceptual File Allocation Table (FAT). The file `test.c` is shown with its blocks coloured and arrows showing the order of sequential access. File `README` is shown uncoloured, with no arrows. A next block "pointer" of -1 indicates there is no following block. Empty spaces indicate unused blocks.

For purpose of creating simple examples, let's assume we have the following functions:

- `int start(char*)` – given a file name will return the block number where the file starts, or a value less than zero if it doesn't exist
- `char *readblock(int)` – given a block number, allocates a `char` array buffer big enough for a block, reads the block into the buffer and returns a pointer to it; the caller must deallocate the newly allocated memory

Assume also:

- the FAT is in an global `int` array called `FAT`
- block size is defined in a preprocessor symbol `BLOCKSIZE`

Here is an outline of code to access each block number (not the contents of the block) of a specific file, if we know its starting block.

```
block = start (filename);
while (block != -1) {
  // do something with current block
  block = FAT[block];
}
```

How about accusing a specific byte in a file? We can do it with the given functions. We need to determine which block the byte falls in, read in that block and extract the relevant byte. Let us create a function to do this:

```
char byteAt (char * filename, int whichByte) {
  int target block = whichByte / BLOCKSIZE,
      offset = whichByte % BLOCKSIZE,
      startblock = start(filename);
  char * buffer = NULL,
      returnval = '\0';
  for (int i = 0; i <= targetblock) {
    block = FAT[block];
  }
  buffer = readblock(block);
  returnVal = buffer[offset];
  free (buffer);
  return returnVal;
}
```

Random access, then, requires that we process each block sequentially until we find the one we need, because of the linked-list structure. This is not quite as inefficient as it sounds. If the block size is large enough, though time to do a random access is linear, the actual time is a small fraction of the time to do linear search a byte at a time. Also, accessing the FAT in RAM is so fast compared with a disk access that we can get away with thousands of accesses of the FAT table without taking time significant compared with the disk operation.

This function is only a rough outline. It should also include error checking such as attempting to read a file that does not exist, or reading past end of file. In a real system, such errors would be caught and either result in a termination of the read operation with an error code, or a program crash.

Check through the code to make sure you understand what it does. Why for example do I calculate the value `offset`? Relate the code to figure 4.2.

The take home message? *A FAT file system is reasonably efficient for small devices. Though random access requires a linear search for the right disk block, accessing a FAT table in RAM is so much faster than disk accesses that linear search is acceptable in most cases.*

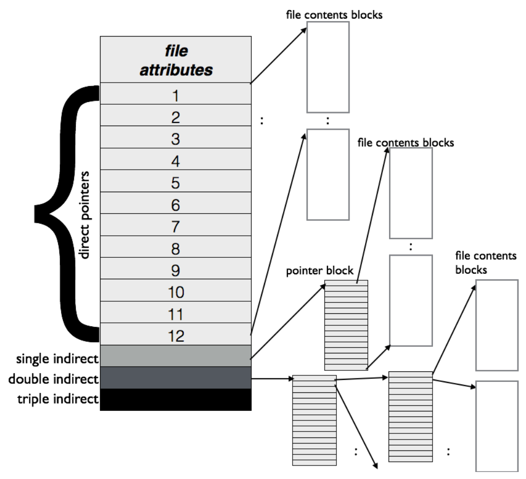


Figure 4.3: Conceptual index node (inode). The top-level block contains file attributes, 12 direct pointers, an direct pointer, a double-indirect pointer and a triple-indirect pointer. Each pointer block is the size of a disk block and contains as many pointers as fit into that size.

inodes

Take a look at figure 4.3, which illustrates the inode concept, omitting most of the double-indirect pointer blocks and not even more of the triple-indirect blocks. All files are represented by at least one inode block, containing file attributes (such as access permissions) and a block of 15 pointers. The first 12 of these pointers point directly to blocks on disk. The next pointer is a *single-indirect* block. It points to a disk block that contains pointers to disk blocks. A *double-indirect* pointer is next. It points to a disk block containing pointers do disk blocks, each of which is also pointers to disk blocks. Finally, there is a *triple-indirect* pointer, which points to two layers of blocks of pointers before you reach the actual file.

Note that “pointer” here means a block number on disk, not a memory address.

This organization allows small files to be represented efficiently, and scale to very large files.

An example illustrates how this scheme works. We can also use this as a design case study to work out how big we need to make our pointers in order to make full use of the capabilities of an inode. One inode only represents the contents of a single file, but we also need to think through how big a pointer needs to be to be able to number every block on a disk. The inode structure does not define the size of a pointer or the size of a disk block: these are numbers we must set to allow a particular maximum file size as well as to set the maximum number of disk blocks we can reference.

So size here depends on two things: how big a disk block is and how big a pointer to a disk block is. First, I focus on how big a file we can represent, then look at how big a disk can be represented with the chosen parameters. Let us work the numbers based on an 8kB disk block and a 32-bit pointer (32b=4B). These numbers mean a pointer block can contain $\frac{8000}{4} = 2000$ entries. We can construct a formula for maximum file size, based on generalizing from the description of an inode. The top-level block stores 12 pointers to blocks. Let us number levels from the top down as levels 0, 1, ... This way the single-indirect pointer is L_1, the double-indirect level is L_2 and the triple-indirect level is L_3.

Let us call block size in bytes s and pointer size (also in bytes) p. Then we can store $\frac{s}{p}$ pointers in a block, and the disk space a block points to is $\frac{s}{p} \times s$, since each pointer points to one block of size s. I now generalise this observation to all levels of inode pointer.

The size of file that can be represented purely by using L_0 is $12 \times s$. For L_1, we need to know the number of pointers in the block (2000 in our example, but let us keep it general for now). The size of file we can represent using only L_1 is $\frac{s}{p} \times s$ bytes. To that we need to add L_0 since we only use L_1 if L_0 is full, but let us work out each layer separately then add them all. L_2 adds $\frac{s}{p}^2 \times s$ bytes and L_3 adds $\frac{s}{p}^3 \times s$ bytes. So putting this all together, this is the maximum file size s_{max} in bytes that can be represented using inodes with up to triple-indirect pointers:

$$\begin{aligned} s_{max} &= L_{0_{max}} + L_{1_{max}} + L_{2_{max}} + L_{3_{max}} \\ &= 12 \times s + \frac{s}{p} \times s + \left(\frac{s}{p}\right)^2 \times s + \left(\frac{s}{p}\right)^3 \times s \\ &= 12s + \frac{s^2}{p} + \frac{s^3}{p^2} + \frac{s^4}{p^3} \end{aligned} \tag{4.1}$$

Back to our example. What is the biggest file we can represent with $s = 8000$ and $p = 4$? Plug into equation 4.1. First, note that $\frac{s}{p} = \frac{8000}{4} = 2000$ and so each time we go up by a factor of $\frac{s}{p}$, the next term increases by 2000 times:

$$\begin{aligned} s_{max} &= \left(12s + \frac{s^2}{p} + \frac{s^3}{p^2} + \frac{s^4}{p^3}\right)\text{B} \\ &= (12 \times 8\text{K} + 16\text{M} + 32\text{G} + 64\text{T})\text{B} \\ &\approx 64\text{TB} \end{aligned}$$

This is a rather large number: 64 terabytes is quite large for one file – and we are assuming we can access the entire file using 32-bit block numbers. Luckily our file pointers in inodes only need to be able to be able to address at block level, so the biggest number we actually need to represent in an inode pointer is about 8-billion (because block size s is 8000). We need more than 32 bits to represent this range of block numbers. 2^{32} is a bit over 4-billion, and allowing that we multiply each pointer by block size, we can address 34TB with unsigned 32-bit inode pointers. Assume for now this is enough for the size of one file.

An actual location within a file is a byte offset from the start; to find that byte, we need to know which block it is in, then recalibrate the byte offset to be from the start of the block. The actual location within the file cannot be a 32-bit number if we can have a file bigger than 4GiB, even if we can make do with a 32-bit block number. So in our code, if we want huge files, we will need to use something like a 64-bit number (the next size up from 32 bits).

Heads up: *We need more bits to represent a position within a file than we need to represent a block pointer because each block is a lot bigger than one byte.*

It is a slight inconvenience that our inodes store 32-bit numbers internally but we will need bigger numbers to represent an exact location on the disk. More of an issue though with using 32-bit numbers for block numbers (which we call pointers here) is that this restricts us to a maximum disk size (or logical volume size) of 34TB with 8kB blocks. Given that in the consumer space disks of a few terabytes are commonplace, this limit is starting to look low even for the consumer space, let alone large-scale servers.

Can we fix this by upping the block size? Doing so is wasteful if we have small files: the bigger the block size, the more likely we are to waste a significant amount of space for small files that are not an exact multiple of a block size. For a very large file, wasting a few kB or even tens of kB is a minor overhead but if a file system contains a lot of small files, the wastage if the block size is very large can be a significant fraction of the used disk space.

What if I change the pointer size to 64 bits, keeping block size s at 8kB? $2^{64} \times 8000B = \approx 1.8 \times 10^{19} \times 8000B \approx 1.5 \times 10^{23}B$ or 150 zettabytes (ZB). This will probably be good for the lifetime of any computer we buy today. Keeping the block size at 8kB, this means we have now 1000 pointers per inode block, and

$p = 8$:

$$\begin{aligned} s_{max} &= \left(12s + \frac{s^2}{p} + \frac{s^3}{p^2} + \frac{s^4}{p^3}\right)\mathrm{B} \\ &= (12 \times 8\mathrm{K} + 8\mathrm{M} + 8\mathrm{G} + 8\mathrm{T})\mathrm{B} \\ &\approx 8\mathrm{TB} \end{aligned}$$

For most purposes, a limit on file size of 8TB should be more than sufficient, and this can easily be changed on moving to a bigger disk where files bigger than this are needed by changing the block size. Although we have limited the number of blocks that can be in *one* file, the disk can be as big as is reachable by 64 bits (the size of each stored file pointer, representing a physical or device block number), which is a reasonable compromise.

Heads up: *Making the block size bigger means the pointer reach of the inode scheme is bigger but a bigger block size is wasteful for small files as a larger amount of wastage as a fraction of useful disk allocation occurs as the block size increases.*

From here on, for simplicity, I introduce P_b for the number of pointers per block, so:

$$P_b = \frac{s}{p} \tag{4.2}$$

This out of the way, how do we access a particular byte at a known offset from the start of a file? Remember with a FAT file system, we had to do linear search through the FAT, though the time penalty is somewhat justified by the fact that the FAT is stored in RAM and therefore quick to access, and a big enough block size to limit the number of searches makes the linear search overhead tolerable for files that are not very large.

For an inode-based system, you can calculate based on the position in the file which block you want. Then, if it is in the first 12 logical blocks, you can look up the device block directly, otherwise you need one of the various levels of indirect pointer.

I sketch out the code here for finding the device block number an inode-based file of a given size, given the relative block number in the file. Appending to a file follows similar logic, except you may need to allocate the final block, so I do not present that separately.

Assume there is a preprocessor symbol `BLOCKSIZE` corresponding to the value s. We need to divide the location in the file by `BLOCKSIZE` to find which logical block our desired location is in. If the block is one of the first 12, we can look it up in the top-level (direct) pointer table in the inode itself, load that block into RAM and access the required byte[1]. Assume also we have a directory data structure that includes in it the identity of the underlying device. We need the following:

- some types:
  - `typedef unsigned int fileptr_size_t` – so we can easily change the size of our pointers
  - `typedef unsigned int file_size_t` – so we can easily change the representation of the size of a file
  - `typedef unsigned int error_t` – to represent error values; I make each error type a power of 2 so I can combine multiple error codes in one value
- some functions; in all cases involving file access, if something fails, set the error code by calling `error` then return 0:
  - `fileptr_size_t blockFromOffset (file_size_t offset)` – given a location (byte offset) within a file, return the logical block containing that byte
  - `void error (error_t type)` – set up an error code to be handled elsewhere
  - `fileptr_size_t getindirect (inode_t *inode, fileptr_size_t logicalblock, unsigned level)` – for indirect blocks in an `inode`, with level 0 indicating indirect, 1 double-indirect and 2 triple-indirect, find the device block number in the `inode`
  - `fileptr_size_t getDiskBlock (fileptr_size_t logicalblock, inode_t *inode)` – for a given logical (relative) block number, b_R, find the corresponding device block number, r_D

Whether we are appending to a file or finding a specific location, we need to translate a byte offset within the file to a specific block and offset within that block.

To find a block number given an offset within the file is easy: we just do an integer divide. So the hard part is, given a block number find the actual block by navigating through the `inode` structure. Let us focus on that.

[1]For simplicity I assume all reads are a whole block.

What we are doing is translating the relative block number within a file (how far into the file we are, counting in units of blocks), b_R into an absolute block number b_D, counted over all blocks on the device.

The basic idea is that you start with the relative block number within a file, and as you eliminate part of the inode structure as being too small to contain that block number, you reduce relative the block number by the number of blocks in the earlier part of the inode structure. Here are the steps in outline, for translating relative block number b_R to device block number b_D (assuming b_R is a valid block number, i.e., not off the end of the file). At each step, if we are able to retrieve b_D, we stop. Whenever we need a pointer block, we need to retrieve it (it may be on disk or in memory, but we do not deal with that detail here). The steps in outline – stopping as soon as you find R_D – are:

1. $b_R < 12$? If so, is use b_R as an index into the direct blocks and retrieve b_D
2. replace b_R by $b_R - 12$. Is the new $b_R < P_b$? If so, use b_R as an index into the indirect pointer block to retrieve r_D.
3. replace b_R by $b_R - P_b$. Is the new $b_R < P_b^2$? If so, find out which second-level block b_R as in and look up b_D in that indirect pointer block.
4. replace b_R by $b_R - P_b^2$. Is the new $b_R < P_b^3$? If so, find out which second-level block b_R as in, then which third-level block b_R is in and look up b_D in that indirect pointer block.

What follows is very conceptual. Much detail is left out and the division between what should be in the header file and what should be in the compiled file is left open, so you can see everything in one place. Note when defining a preprocessor macro for a calculation, it is safest to enclose it in parentheses so it will be calculated as a unit wherever the macro is expanded.

The data type for an `inode` is left undefined except for the part that identifies blocks and the size of the represented file to keep things as simple as possible.

The most tricky part of finding the right disk block is navigating triple-indirect blocks. Each pointer in the first block refers to another block containing pointers. Each pointer in the first block takes you to a block of pointers, each of which takes you to another block of pointers. So each position in the first block takes you ultimately to one of $\left(\frac{s}{p}\right)^2$ positions. This to find the right index in the first-level block, we need to divide by this number. That gives us an index in the

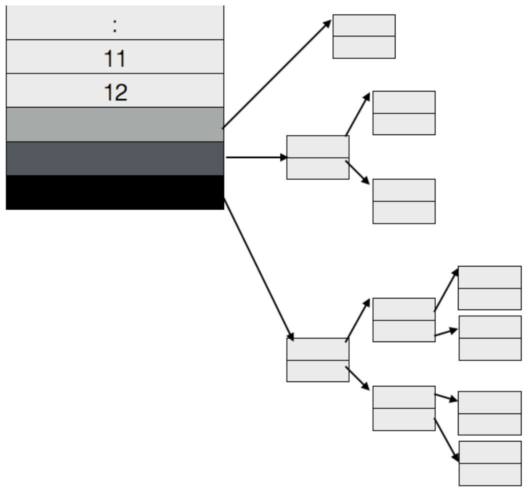


Figure 4.4: Minimal inode pointers: A full complement of indirect blocks is illustrated with the artificially small example of only two file pointers per block.

first-level block that takes us to a second-level block. The offset in this block is the remainder from the previous division.

Figure 4.4 illustrates detail of a a full set of the indirect pointer blocks, cut down to only two file pointers per block, so we can see a whole example in a small picture. It is easier to visualize the whole scheme like this than with a realistic example with thousands of pointer blocks.

Here now is our algorithm sketch converted to C code capturing the major details.

```
typedef unsigned int fileptr_size_t;
typedef unsigned long int file_size_t;
typedef unsigned int error_t;
typedef struct INODE inode_t;

// error codes + each a power of 2 so we can add them
#define GOOD       0
#define NOFILE_ERR 1
#define EOF_ERR    2

#define BLOCKSIZE 8000
// number of direct pointers
#define MAXDIRECT 12
```

```
// pointers per block
#define POINTERSPERBLOCK (BLOCKSIZE / sizeof(fileptr_size_t))

struct INODE {
// file attributes go here
file_size_t allocated; // number of bytes allocated
fileptr_size_t direct[MAXDIRECT];
fileptr_size_t * indirect,
    * doubleindirect,
    * tripleindirect;
};

// rescale an offset in bytes to an offset in blocks
fileptr_size_t blockFromOffset (file_size_t offset) {
    return (fileptr_size_t) (offset / BLOCKSIZE);
}

// set up an error condition that will be handled elsewhere
void error (error_t type);

// get the logical block in an inode indirect structure, given
// level depth, with 0 for single-indirect; the given logical
// block numbered from the start of this section of the inode
fileptr_size_t getindirect (inode_t *inode, fileptr_size_t
                            logicalblock, unsigned level);

// in our scheme disk block of 0 signifies an error
fileptr_size_t getDiskBlock (fileptr_size_t logicalblock,
                             inode_t *inode) {
   if (logicalblock > blockFromOffset(inode->allocated)) {
     error (EOF_ERR);
     return 0;  // a disk block of 0 is never valid
   }
   if (logicalblock < MAXDIRECT)
     return inode->direct[logicalblock];
   // rescale the problem to start from the first indirect block
   logicalblock -= MAXDIRECT;
   if (logicalblock < POINTERSPERBLOCK)
     return getindirect (inode, logicalblock, 0);
   // rescale problem to start from the first double indirect block
   logicalblock -= POINTERSPERBLOCK;
   if (logicalblock < POINTERSPERBLOCK*POINTERSPERBLOCK)
     return getindirect (inode, logicalblock, 1);
   // rescale problem to start from the first triple indirect block
   logicalblock -= POINTERSPERBLOCK*POINTERSPERBLOCK;
   // no need for if here because we checked we aren't past EOF
   return getindirect (inode, logicalblock, 2);
}
```

The hard part we have now isolated down to function `getindirect`. To implement it you need to do some arithmetic to divide the residual logical block number (relative block b_R after subtracting the block numbers representing the previous parts of the inode) into pieces. If the block you want is found via double-indirect blocks, you will need to do this twice, once to find the right entry in the top-level (L1) table of block pointers and a second time to find the right entry in the second-level (L2) table. For triple-indirect pointers, you need another layer of arithmetic to get to the right L3 pointer block, then into the right entry in that block.

Heads up: *The hard part of inode block look-up is the fact that each of the pointer blocks has a different number of layers of indirection but it is worth working out a general solution so any future change in design can easily be accommodated.*

All of this is a great programming exercise, so completing the example is left to you.

What I have left out so far is where all this is stored. Obviously so files can be found between shutdowns, the inodes need to be stored on disk. For fast random access, should the inodes for a particular file currently being accessed be stored in RAM? Let's work through this as another design exercise.

The fact that pointers are allocated as a whole disk block means that you incur a penalty of a disk access only once for each pointer block (in our examples, containing 1000 or 2000 pointers). For a single-indirect block, that means you incur this penalty once, then you can access each block pointed to by that pointer block without having to go to disk again. If you work with the figure of 1000 pointers per block, even going to triple-indirect pointers adds at worst an overhead of 3 extra disk accesses per thousand for sequential access, an acceptable overhead.

Only strictly random access then is a problem in terms of extra overhead if inodes are not cached in RAM (the way FAT is), and that problem is in any case addressed to some extent by file buffering (or cacheing). File buffering can also apply to storing inode pointer blocks, since they are also disk-based data. Handling file buffering as a separate concern helps to reduce complexity of inode lookup code.

A major detail left out is how all this is stored on disk: that varies from system to system, and can become quite complex. An example of a design trade-off: are inodes and pointer blocks stored in a predefined region to make lookup faster, at the cost of limiting the total number of files, or is a more flexible scheme used that allows unlimited growth in number of files, at the expense of complexity?

The take home message? *The inode scheme allows for much bigger files than FAT at the expense of making random access more complex. The benefit of FAT in keeping file lookup information in RAM is relatively minor as the overheads of looking up inode pointers on disk are a small fraction of total disk accesses and can be offset by cacheing, which reduces overall disk accesses.*

journalling

Both inode and FAT schemes have the problem that an unclean shutdown can leave the disk-based version of their data structures in an inconsistent state.

Generally, updating a file system for a particular operation involves more than one step. For example, in an inode-based system, changes may require alterations to the directory entry as well as the inode pointer structure. If the system shuts down partway through a multi-step transaction or even partway through one transaction, the file system could be inconsistent. Examples of how that inconsistency could manifest include a directory entry to a file that is meant to be deallocated, an inode pointer that no longer points to valid data, or valid data that does not have an inode pointer that points to it.

One way of dealing with this problem is to traverse all the data structures in the file system to check for inconsistencies. In the Unix world, the program that does this is called `fsck`[2] (for file system check). For smallish file systems, having

[2]Pronunciation is left to the imagination.

to wait after an unclean restart for `fsck` to run was annoying; with disks in the terabytes containing millions of files, the wait would be worse than annoying.

A *journalling file system* solves the problem by keeping a *journal*: a log of file transactions. Each transaction requires a single write to the log to record, making it unlikely to be interrupted partway through. If it is, the logged transaction hasn't happened anyway, since the order of operation is log first, then do transaction.

After an unclean restart, the operating system detects that there has been a problem and replays any entries in the log that have not correctly completed.

The common approach to journalling, which keeps a *logical journal*, only records metadata (such as updating the directory entry or the inode structure), not the actual content of update. After a crash, the user should check whether whatever file was being worked on at the time of the crash contains garbage.

Where more robust fault recovery is required, a *physical journal*, which records contents as well, can be used, at the cost of doubling the number of disk accesses.

Most file systems in common use at time of writing use a logical journal.

The take home message? *Journalling allows a file system to be kept consistent without having to check it in detail across unclean reboots. A logical journal only records file system structure; a physical journal records file contents as well.*

file organization

In most file systems, files are organised into *directories*, visualised as *folders* in a file viewer. Directories provide a hierarchy to make large numbers of files manageable, and also provide a larger unit of granularity than a single file for managing access rights.

In some systems, a directory is a special case of a file; in others it is a completely separate structure. Making a directory a special case of a file makes it easier to treat the two similarly when similar abstractions apply. For example, Unix-based systems, which do not treat directories was a special case of a file, have *hard links*, which are in effect another name for a file. A hard link is represented by more than one directory entry pointing to the same inode. If a directory was just a special case of a file, a hard link to a directory could be handled the same way as a hard link to a file. In practice, though, you cannot make a hard link to a directory in Unix-style systems because they implement directories differently to files.

A Unix-style hard link only works in the same file system or volume, since an inode may not be meaningful outside the context where it is defined so Unix-style systems also have a *soft link*, which is a representation of the location where a file is stored. The hard link mechanism relies on the fact that essential metadata like ownership and permissions is stored in the inode so more than one reference to the same inode will not result in possible inconsistencies in metadata. Whenever a new hard link is created, a count is updated in the inode so that "deleting" the file has the effect of removing one of its directory entries until the link count reaches zero, when the file is really deleted.

If a file is moved or deleted, a hard link is more robust than a soft link. If the file is renamed, moved or deleted from one directory location, any hard link to it elsewhere can still access it. A soft link, on the other hand is specific to the original name and location.

Other file systems have similar concepts. On Mac OS X, for example, an *alias* is a slightly smarter version of a soft link that follows the original file if it is renamed or removed, but still points at the original location if the original file is overwritten by a new one of the same name. Windows has a similar concept called a *short cut*.

All of these variations on the soft link concept are implemented as a small file containing some kind of representation of what the link refers to.

On Unix-style systems, if you do an $ls-l$ command (long form of directory listing), between the permissions string and the user name, there is usually a "1" indicating that there is only one name for the file. Each time you create a hard link, that number increases. Remove one instance of the hard link, and the number goes down. For example:

```
$ ls -l test.txt
-rw-r--r--    1 philip  501         208 Oct  3  2013 test.txt
$ ln test.txt test2.text
$ ls -l test.txt
-rw-r--r--  2 philip  501  208 Oct  3  2013 test.txt
```

To get rid of this file, you need to remove it in *both* places where it now exists – remember, a hard link is another name for the *same* file.

To create a hard link on a Unix-style command line, you use the `ln` command. The first item is an existing file, the second its alternative name. Either name may include a path. A soft link is created by adding the `-s` option. A soft link can point to a directory. Here is an example:

```
$ ln -s /tmp/ test
ls -l test
lrwxr-xr-x  1 philip  501  13 Feb 11 21:44 test -> /tmp/
```

Note the extra "`l`" (lowercase "L" not a one) at the start of the permissions string, indicating that this is a soft link.

> **The take home message?** *A file system's user-level abstractions are files, directories and various kinds links. In graphical interfaces some of these concepts may have different names like folders, aliases and short cuts.*

4.3 Performance

Performance falls into two broad categories: speed and error-free operation. These categories further subdivide. Speed can be see in terms of *latency* (time to complete one operation) or *bandwidth* (also called *throughput*), average amount of work done over time. Error-free operation can arise either from two properties. The first, *reliability*, is the probability of running without a fault. The second, *fault tolerance*, is the probability of keeping running even if there is a fault.

These concepts apply across the system, but are most important for IO since IO is the slowest part of the system and also the part most likely to fail. These competing measures are often in conflict. It is easier to achieve low latency if bandwidth can be sacrificed and fault tolerance can be achieved by adding extra redundant parts, at the expense of increasing the probability that a part will fail.

An example of how overall throughput can be sacrificed for latency is in the design of a phone, which is designed to respond instantly user requests like picking up a phone call. This low latency is achieved by making the phone waste CPU cycles waiting for user actions. We will see how reliability and fault tolerance interact when we look at RAID disk systems.

> **Heads up:** *Be clear on the difference between latency and bandwidth and why it is possible (sometimes desirable) to trade the one for the other.*

4.3.1 Speed

disk timing

Let us look at the general issue of performance for a disk – for flash the issues are similar but the latencies are caused by very different physics and are much lower.

The other main differences between disk and flash is that flash writes are much slower than reads and flash wears out if written often. Disks can also become damaged with use, but the cause of the damage is less predictable. I return to these issues further when looking at reliability.

To access as specific block on the disk (or *sector* when we are looking at low-level disk units), there are three components of the total time:

- *rotational delay* – on average, half a rotation[3]
- *seek time* – the time for the head to move to the correct track (or more correctly, with a multi-platter disk, cylinder); this time is usually advertised as an average for a given disk
- *transfer time* – this is the time for the given number of bytes to move (on or off the disk), usually much less than the other components of the total time

To calculate these numbers is reasonably straightforward. Seek time is usually given by the manufacturer (though it should be measured for a given system, since files organization may change the average). Rotational delay can be calculated from the disk rotation rate, usually given in revolutions per minute. Transfer time can be calculated by dividing the number of bytes to transfer by the transfer rate.

An example illustrates how to calculate the total time (as an average). Here are some numbers for a specific disk:

- rotation speed: 14400rpm
- platters: 6
- track size: 500,000B
- average seek time: 10ms

First, let us calculate rotational delay. A rotation speed of 14400rpm means $\frac{1}{60}$th of that number of rotations per second. So that means the disk does 240 rotations per second. One rotation then takes $\frac{1}{240}$ or about 4.2×10^{-3}s. Half a rotation then takes 2.1×10^{-3}s (or 2.1ms).

What about transfer time? That is limited by the interface and controller, but we can work out the maximum rate the disk could move the desired data by using

[3]It is possible to use *rotational position sensing* to reorder requests to minimize rotational delay, but we ignore this possibility since it is not a feature of consumer-technology disks.

the track size, number of platters and rotation speed. In this case, because a track is 500,000B and there are 6 platters, we can move $6 \times 500,0000$B in one rotation, or in 4.2×10^{-3}s. That means the maximum rate data can move from the disk is:

$$\frac{5 \times 10^5 \times 6}{4.2 \times 10^{-3}}\text{B/s} \quad \approx \quad 700\text{MB/s}$$

If we translate that to time per byte, it is $\frac{1}{7 \times 10^8} \approx 1.4$ns – or, to put on the same scale as our other times in ms, 1.4×10^{-6}ms.

This is a hard limit on how fast data can move on or off this disk purely based on device speed. Short bursts of faster access may be possible if the disk has a fast RAM buffer on board. Buffering writes to disk is easy, provided you do not run out of memory. The writer dumps to the buffer and goes away. A read buffer needs smarter organization, as it needs to anticipate future reads.

Let's put all these numbers into a formula:

$$t_{total}(\text{bytes}) \quad = \quad t_{seek} + t_{rotation} + t_{transfer} \times \text{bytes} \tag{4.3}$$

Time units depend on the numbers we plug in. Since the biggest is in ms, we can standardise on that unit; anything very much smaller will be close to zero and disappear as roundoff error.

How long, given these numbers, does it take to move various sizes of data off the disk if it is contiguous on the disk, so you only incur rotational delay and seek time once? Let us look at 1 byte, 1000 bytes and 1,000,000 bytes. Here is each plugged into equation 4.3.

$$\begin{aligned} t_{total}(1) &= (10 + 2.1 + 1.4 \times 10^{-6} \times 1)\text{ms} \\ &= 12.1000014\text{ms} \\ &\approx 12.1\text{ms} \end{aligned}$$

So if we transfer 1 byte, the biggest single factor is seek time, then rotational delay. Transfer time is insignificant.

For 1000 bytes:

$$\begin{aligned} t_{total}(1000) &= (10 + 2.1 + 1.4 \times 10^{-6} \times 1000)\text{ms} \\ &= 12.1014\text{ms} \\ &\approx 12.1\text{ms} \end{aligned}$$

the transfer time is still so small as to be insignificant. Finally, for 1,000,000 bytes:

$$\begin{aligned} t_{total}(1 \times 10^6) &= (10 + 2.1 + 1.4 \times 10^{-6} \times 1 \times 10^6)\text{ms} \\ &= 13.5\text{ms} \end{aligned}$$

Even in this case, disk transfer time is only about 10% of the total elapsed time. Can you see why, even if you logically want one byte, it makes more sense to move a big chunk into RAM on the off chance that you will want the rest of it?

When we measure IO performance, we split our measure between

- *latency* – time to complete a whole transaction
- *bandwidth* or *throughput* – average time per unit of work

If we are accessing a single byte off a disk, latency is about as good as it gets. We retrieve it in a about 12ms. Throughput in this case is $\frac{1\text{B}}{12.1\times10^{-3}\text{s}}$, or about 83byes/s. If we retrieve 1000 bytes, latency is only a tiny amount worse. But bandwidth is $\frac{1000\text{B}}{12.1\times10^{-3}\text{s}}$, or about 83kbyes/s. What if we retrieve 1MB? Latency is now a little worse: 13.5ms. But bandwidth is now $\frac{1\times10^{6}\text{B}}{13.5\times10^{-3}\text{s}}$, or about 74Mbyes/s.

What this example illustrates is that there is often a trade-off between latency and bandwidth. If the goal is to make each individual transaction as fast as possible, the average rate of work per unit time may drop.

Our ideal case is to create the appearance of low latency because that is what the user sees – click the mouse, and you don't expect to wait – while maximising bandwidth, because that is a measure of how efficiently the system is being used. When hardware is very cheap, it becomes possible to favour latency over bandwidth. A cell phone, for example, dedicates most of its computing power to instant responses rather than ensuring the hardware is kept busy.

We now turn to various methods of hiding latency; scheduling another task (chapter 3) is another strategy for latency hiding.

The take home message? *In a disk, latency is dominated by seek time and other overheads to get started unless very big amounts of data are moved once the access is set up.*

buffering

A *buffer* is a region of fast memory between devices of different speeds that allows the faster device to carry on after depositing data in the buffer. A buffer can also hold data placed there from a slower device in anticipation of demand from the faster device. These two modes are respectively *write* and *read buffering*[4].

Write buffering is easier than read buffering. As long as the buffer is not full, a write can dump to the buffer and the faster device can continue without waiting.

[4]Usually, reading and writing are from the perspective of the CPU, the fastest part of the system.

In the case where the faster device is the CPU and the slower device is a disk, the speed gap may be factors of millions so no buffer will be big enough if there are continuous writes without a break. Fortunately most workloads are not that unforgiving: bursts of writes are followed by bursts of computation.

For read buffering to be effective, it is necessary to anticipate future reads. A simple way to do this is to read a larger amount than a given request, in anticipation that the surrounding content will be needed. When we study memory, we will see that it is quite common that memory accesses are followed by others in the same region. This is called *spatial locality*. The same principle applies to a lot of IO. It is very seldom that data is accessed in truly random order from a file.

Return to our example of total transaction time for a disk. Since the time difference between transferring a few kilobytes is virtually indistinguishable from the time to transfer one byte, most disk IO is actually results in a transfer of at least a few kilobytes even if only one byte is wanted. There is a huge payoff for this "wasteful" disk access if any of the extra content is needed: the cost of a modest amount of extra memory for the buffer is easily worth the time saving of avoiding another disk access.

The take home message? *Write buffering is straightforward: writes are stored in RAM until it is worth writing to a slower device. Read buffering requires anticipation of future demand by reading more than is requested.*

spooling

Spooling applies when a device has to have all its output in a single unit. The usual application is a printer. A print job has to be handled as a complete unit, but a printer is so much slower than most of the rest of the system[5] that it is impractical to manage a print job from the process that created it. Instead, the data and instructions on how to process it are dumped into a file and queued for later processing.

Spooling differs from buffering in that buffering does not force a particular request to run to completion – a disk, for example, is capable of accepting requests in pretty much any order, as long as they do not alter the intended content.

The take home message? *Spooling applies to output, usually printing, that has to be handled as a complete unit once the device is ready.*

[5]A human may be slower but not much else is.

cacheing

Cacheing is storing content in a faster memory than that in which it is normally stored to speed up access. Cacheing differs from buffering in the sense that the lifetime of the contents is not clearly defined. Whereas a buffer empties as its contents is used up, a cache keeps a copy of its contents unless it has to be evicted to make room for new content.

In some systems, a disk buffer is called a file cache. A disk buffer can operate a bit like a cache in the sense that its contents may not be immediately flushed after the last request completes in case the contents may be needed again. Whether the term cache or buffer applies is not always clear.

The term *cache*[6] is most clearly applicable at the hardware level, where the memory hierarchy starts with registers and first-level cache. Caches at the hardware level are mostly invisible to software other than as a performance enhancement. File caches, on the other hand, are managed in software, if transparently, as part of the operating system.

The take home message? *Anything that redirects access from a slow kind of memory to a faster one is a win. Cacheing differs from buffering in* intent*: a cached item stays there as long as possible whereas a buffer is more transient. This can be a subtle difference, so there is overlap in the concepts, hence the fact that file buffers are sometimes called a file cache.*

disk scheduling

Since seek time is one of the bigger components of disk access time, minimizing seek time is a useful goal. If the disk is relatively idle, it has only one access at a time; we are interested in the scenario where accesses are queued, so the OS has the option to choose which to schedule next.

The simplest approach to disk scheduling is to take access requests in the order they arrive. A more sophisticated approach is to order the requests, and take all the requests that can be serviced while moving the head in a particular direction. Once the head reaches the inner or outer edge of the disk, it takes all requests in the opposite order. These basic schemes can have variations:

- *first come first served* or *FCFS* – take requests in the order they arrive

[6]Pronounced like “cash”: it is after all one of the more expensive kinds of memory.

- *elevator algorithm* – like an elevator (lift), process requests in the same direction up to the innermost or outermost track, then start over; variations:
  - *one-way* versus *two-way* – either only process requests in one direction (one-way algorithm) and skip back to the start, or process requests in both direction; a two-way algorithm is sometimes called *circular*
  - *stop at last request* versus *move head right to edge* – when moving the head towards the centre or towards the outermost track, either stop as soon as there are no more requests in that direction, or carry on moving the head all the way before stopping

These different approaches have advantages and disadvantage. For a flash device where seek time is not a big issue (there is some latency to set up a new request, but there is only really a saving if adjacent accesses are for a part of flash very close to the previous access), FCFS is acceptable. For a disk with significant seek time, some variant on an elevator algorithm is likely to give better performance.

Flash gains from requests relatively close together but that gain is mostly made by choosing a suitable block size; there is no equivalent to seek time that is much higher than any other component of access time. While writes to flash are relatively slow, that latency can be masked by buffering.

> **The take home message?** *For a disk, seek time is the biggest component of latency so ordering requests to minimise seeks is worthwhile. For flash, there is less incentive to reorder requests, since latencies of shifting to another part of flash are pretty much invariant.*

4.3.2 Reliability and Fault Tolerance

I describe reliability and fault tolerance and use RAID as a case study. Reliability and fault tolerance, like latency and bandwidth, are concepts that can be traded for each other, depending on requirements. RAID provides an example of how these trade-offs can apply.

First, let us consider some causes of unreliability.

A disk is a rapidly rotating piece of machinery with heads that move in and out across its surfaces very fast. Mechanical failure can damage a disk, often but not always beyond repair – particularly with low-cost drives where repairs would cost more than they are worth. Repairing – or recovering data from – a physically damaged drive is usually only worth the effort if data of value has not been backed up. The recordable surface can also become unreliable, in which case blocks (or sectors, if we are thinking at a lower level) can get mapped out. Usually the drive itself manages this level of error handling. A block that is hard to read or write may result in multiple retries before there is a good result. In such a case, the drive's low-level controller (often part of the drive itself) will mark the black as bad, and not make it available (ideally after copying the contents elsewhere). This level of error handling may be invisible to the user or even the operating system.

Flash has no physical moving parts, but has a limited number of write cycles so frequently written parts wear out faster than less frequently written regions. A flash device that is meant to replace a disk in regular use (as opposed to a simpler one for a purpose like moving data around on physical media) often has *wear levelling*: the device's internal controller detects when a block is written frequently, and copies that block to a less frequently modified region. Wear levelling can also be implemented by the OS. In either case, the result is longer lifetime of the flash device at the expense of overheads of moving frequently modified data – which should ideally be done when the system is relatively idle, but may have a noticeable performance impact if that is not possible.

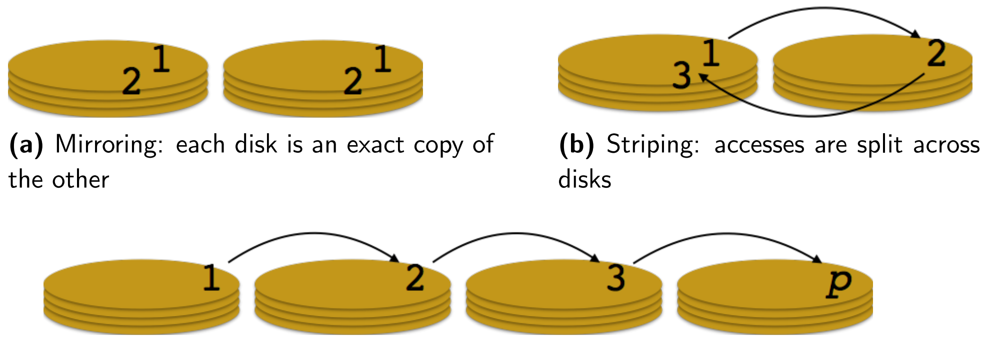
Since flash is a rapidly-evolving technology it is easiest on the whole to build wear levelling into the device's own controller than for the OS to do wear levelling, even if this carries a risk of a particular device having a sub-optimal strategy. The best wear levelling strategy depends on how the device is used, so a flash device intended for relatively rare writes (e.g., a portable backup device) may not have the best strategy for a dissimilar purpose (e.g., using it as a high-traffic database).

There are various techniques for keeping track of whether a block whether on disk or flash is good, including a checksum (calculated over the contents: if any contents change, there is a very low probability that the checksum will still be correct) and more advanced error checking and correcting codes.

The take home message? *A disk can break because it has rapidly-moving parts; flash can wear out from excessive writes to the same bits.*

RAID

RAID originally stood for redundant array of inexpensive disks [Patterson et al. 1988]; disk manufacturers, possibly with an eye to selling RAID drives at a higher price point, changed the acronym to stand for redundant array of *independent* disks. RAID is based on two ideas:



(a) Mirroring: each disk is an exact copy of the other

(b) Striping: accesses are split across disks

(c) Striping with parity

Figure 4.5: Mirroring versus striping. The numbers show order of accessing a disk for a request. An access labelled "p" is a parity block. Schemes with parity blocks generally need more disks in parallel to allow for this extra redundancy.

- smaller mass-market disks that are less expensive than bigger server-oriented disks can be scaled up by operating them in parallel
- multiple smaller disks are more likely to fail than one bigger disk; fault tolerance can be added with error checking and correcting information

RAID can dramatically improve bandwidth over a single drive but cannot improve latency for small accesses because the individual drives must still do seeks and incur rotational delay.

RAID comes in many variations from a limited number of drives in parallel with no extra ability to correct failures, to multiple drives in parallel with error correction capability.

> **Heads up:** *Increasing the number of parts to implement something makes it inherently less* reliable. *The more parts there are, the more likely something is to break. Adding in error checking and correction adds* fault tolerance, *the ability to recover from failure. It is important to distinguish these two concepts. A RAID device is* not *more reliable than a single disk even if it is sometimes incorrectly marketed as such. It is, however, if it implements error checking and correction, more* fault tolerant.

I present here a brief summary of RAID variants. The most elementary versions of RAID do either *striping* or *mirroring*, as illustrated in figure 4.5.

In mirroring, disk contents are replicated. The goal is fault tolerance not performance (figure 4.5a). Read speed can be improved by mirroring since the same block is in more than on device, but write performance is inherently slower because each write must be duplicated and is held up by the slowest drive.

In striping, disk accesses are split across two or more drives. Striping can be at various levels of granularity. If at block level, sequential block accesses each are from a different disk. Figure 4.5b shows accessing three blocks, numbered 1, 2 and 3 on a block-level striped disk. If organized right, the accesses can be overlapped, speeding up overall transfer rate. Figure 4.5c shows an additional disk added with parity check information. A parity check, depending how much information is stored, allows testing for errors and may also allow correcting errors. RAID drives can have a single disk dedicated to parity, or distribute parity blocks over all the drives..

Distributed parity is more complicated to implement than a dedicated parity drive but has the advantage of relatively even use of all the drives, whereas a single parity drive is modified every time a block is modified no matter which of the other drives it is on. In either case, the goal is to allow failure of a disk (or more than one disk, depending on the scheme) to be tolerated.

Some RAID systems support *hot swapping*, the ability to remove and replace a damaged drive without shutting the system down or losing information

RAID levels are numbered from zero (simple striping) through six, with increasing ability to recover from errors. Remembering what each RAID level signifies is not of vital importance since this is a shifting terrain. What you do need to know when specifying RAID is:

- *is read performance more important than write speed?* In this case, a RAID scheme with extensive parity checking can be win even though creating parity data is an overhead for writes

- *is fault tolerance more important than speed?* If so, maximising parity information is important

- *is speed more important than fault tolerance?* In this case, striping only may be acceptable provided data is recoverable or backed up periodically

A simple mirroring scheme can be relatively fast since all the fault tolerance is from simple replication, at the cost of doubling the needed storage.

The take home message? *There are many variants on RAID, with different balances between fault tolerance, speed and cost.*

4.4 Protection and Security

Protection and security are huge subjects; I only touch on them briefly here as they are implemented in file systems.

The Unix model is to do protection at the file and directory level using permissions: in the simplest system, the world is divided into the user (owner), a group (which can have multiple users) and others. Each file or directory has 9 bits of permissions, 3 for each category, representing read, write and execute. For a directory, execute permission means you can make the directory your working directory. If you do `ls -l` on the command line (long form of directory or file listing), the first thing shown on each line is the permissions, in order user, group, others, for example:

```
-rwxr-xr-x
```

In this case, the file listed is an executable: all users can execute it ("`x`" permissions are on for all three categories of user). A directory that all users make their working directory (using `cd` on the command line) has "`x`" permissions set as well. The "`r`" for a directory singifies being able to list its contents; a "`w`" signifies being able to alter its contents (e.g., create a new file or remove a file). Here are permissions for a directory that everyone can view or make their working directory but only the owner can modify:

```
drwxr-xr-x
```

Note the "`d`" at the start of the permissions, indicating this is a directory.

The Unix permissions model is limited in flexibility. If you have a group of people with different roles depending on context, the user-group-others model is too inflexible. A more recent model, an *access control list* (*ACL*), allows finer-grained control over permissions. Each file or directory can have a list of users and their rights.

ACLs are available as an additional option in recent versions of Linux as well as in Microsoft Windows and Mac OS X.

As distributed computing (§6.5) becomes more commonplace more sophisticated modes of access control that were perviously used in research may be revived. One example is a *capability*, a data structure representing a particular set of permissions for a particular object [Jatho III 2014].

The take home message? *Unix-style permissions and ACLs are the two most common modes of permissions and file system security.*

4.5 Other Device Types

Aside from disks and flash drives, there are many other types of device than can attach to a computer. Some are so slow, the computer is better off doing other work rather than wait for them. Others, though slow, need to be responsive.

A computer screen, though an output device, operates at the low level through the memory system and a graphics processing unit (GPU). The key attribute of graphics processing is a combination of large data flows with (by computer standards) long breaks between and specialist computation. A screen that redraws 100 times per second needs to be able to move the necessary data in a small fraction of $\frac{1}{100}s$ to avoid the appearance of flicker. A reasonably large screen with a resolution like 2560×1440 pixels that needs to move 4B per pixel through the memory system in less than $\frac{1}{100}s$ requires a data rate of around 1.5GB/s. Luckily much of the actual data transfer happens between the GPU and a dedicated memory, so a graphics device really looks more like a specialist CPU than an IO device to the rest of the system.

At the other end of the scale, a keyboard needs very little processing power to handle since people are very slow by computing standards. Such advice also works best with some dedicated hardware, a buffer to dump keystrokes to, that can be processed when the CPU gets around to it. In practice, since we expect a keyboard to be responsive, handling keystrokes needs a reasonable level of software intervention. However, that intervention is usually in the application layer so the programmer in effect gets to decide whether to make keyboard operation feel interactive or not.

Networks open up another whole range of issues so complex that they are generally handled as a separate subject. Networks have similar latency-bandwidth trade-offs to disks: a single transaction takes a long time to set up compared with the average time per byte for a transfer. However, reliability dictates that network data be sent in relatively small units, up to a few kilobytes, even if it would be faster on paper to send larger units.

The take home message? *There is a huge range of device types with speed variation from slowest to fastest of a factor of a million or more, and big variations in reliability and modes of access.*

Exercises

1. Look up specifications of real disks and SSDs, and rework the numbers in the example on page 58.

2. Here are numbers for another device:

   - rotation speed: 7200rpm
   - platters: 4
   - track size: 500,000B
   - average seek time: 12ms

   Calculate:

   (a) Time to transfer 1 MB (only transfer time)

   (b) Time to transfer 1 byte (only transfer time)

   (c) Average rotational delay

   (d) Total time to transfer 1 MB (including seek time and rotational delay)

   (e) Total time to transfer 4KB (including seek time and rotational delay)

   (f) Total time to transfer 1 B (including seek time and rotational delay)

3. Comment in the light of the answer to question 2 on why operating systems attempt to transfer relatively large units at a time.

4. Add error checking to the FAT random access read code (function `byteAt` on page 44). Your revised function should set a global variable `errno` to a value indicating the type of error, and return a null `char` (the value `'\0'`). You should also check if the value returned by `readblock` is not a `NULL` pointer (what should you do if it is `NULL`?).

5. Relate the code of the FAT random access read code (function `byteAt` on page 44) to figure 4.2. Assume block size is 8KB and that you want the byte at position 25,000. Work through the code and check that you get to the right block, and see how to use the offset value.

6. How many machine-code instructions in the MIPS instruction set is the FAT random access read code (function `byteAt` on page 44) per loop iteration? If you had to find block number 10,000 and each MIPS instruction takes 1ns

(10^{-9}s), how long does it take to find block number 10,000 as compared with a disk read taking about 10ms (10^{-2}s)? Considering that most CPUs execute instructions at a rate faster than one per nanosecond, comment on whether the linear search of FAT is a real problem.

7. Calculate how big a file can be stored using inodes with triple-indirect pointers if a disk block is 8KiB in size and each pointer is 40 bits.

8. Complete the code for inode allocation (page 51) for:

   (a) Double-indirect pointers

   (b) Triple-indirect pointers

9. A general formula for the number of disk accesses needed to fetch every pointer block if the maximum-sized file is created using all levels of indirection in the inode scheme described in this chapter would be useful.

   (a) Derive this formula.

   (b) Use this formula to calculate the number of disk accesses for pointer blocks if every block in the maximum-sized file is accessed once.

   (c) Calculate overhead as a fraction: extra accesses $\div$ data accesses.

10. Extend the given inode access code to implement appending *N* bytes to a file. Your code should handle the following cases

    - the extra bytes fits into the last existing block
    - the extra bytes require one or more additional blocks at the same level of indirection of pointer blocks
    - the extra bytes require another level of indirection of pointer blocks

11. You are designing a large web site that is updated relatively infrequently, has a very high rate of visits and needs high read performance. Discuss what variant on RAID would suit the problem.

12. Look up what *role-based access control* is and contrast it with Unix-style permissions and ACLs.

5 Memory

MANAGING MEMORY IS a core operating systems function. Only very early or very primitive OS designs do not have OS-managed memory, and most rely on hardware support for memory management. Managing memory is important for two reasons: memory access is a critical performance bottleneck that has to be managed right to avoid performance problems, and protecting against invalid memory accesses is important for both correctness and security.

In this chapter I examine core ideas of memory management and some variations in implementation strategy.

First, some history and a rationale for memory management.

5.1 History and Rationale

In the very early days of computers, memories were too small for much of an operating system but the smallness of memories also created an incentive to manage memory efficiently. The idea of a *memory hierarchy* goes back a long way and early systems required the programmer to manage what was in each level.

Virtual memory (VM) is the concept of an automatic mapping between programmer and hardware addressing. A *virtual* address, as seen by a program, appears to define an offset from the start of memory. The VM system hides the fact that the actual location in memory is different. VM allows programs to run as if they have the entire address space to themselves, even though multiple other programs are running with the same illusion.

VM provides four major benefits, which I expand on later:

- it protects programs from each other

- it allows each program to be written without knowledge of what else is using memory
- it allows the address space to be extended to a backing store (in recent times, a disk or flash solid-state drive), also known as *swap* or *swap space*
- it allows a program to be loaded into memory without requiring a contiguous region of memory to fit its entire requirement

The Ferranti Atlas [Lavington 1978] was one of the first computers to feature VM. Designed in the late 1950s and launched about 1962, it used fixed-size *pages* as the unit of transfer between fast and slow memory. Unlike a modern disk, its secondary storage was a magnetic drum, a cylinder with read-write heads on the outside. Pages were 512 words; a word was 48 bits or 6 bytes, making a page 3KiB in modern terms.

Despite this early start, VM took a while to be universally accepted. Some large-scale computers 10 years later still required the programmer to manage occupancy of main memory and early PCs also did not have virtual memory. Part of the reason for the slow acceptance of VM is that it is difficult to implement properly and hardware support must be designed with software in mind. The Atlas project was an example of close collaboration between hardware and software designers. Without this collaboration, it is difficult to make VM efficient.

Also, early PCs were designed more for the hobbyist than the professional user, so reliability was a low priority as compared with cost and simplicity. The original IBM PC, running MS DOS, was only intended to run one program at a time, though workarounds for that were soon found. Once Microsoft introduced Windows, running multiple programs was more commonplace but even so it was only Windows 3.0 in 1990 that first supported virtual memory. Apple too was slow – the early Mac OS ran all programs in the same address space. Even when VM was introduced by Apple in 1991, it just extended the address space that all programs ran in with swap space on disk to support using more memory than the physical RAM, rather than giving each program a separate address space.

Microsoft and Apple eventually rewrote their systems substantially to create a true VM with a separate address space for each process. In Apple's case, that involved rebuilding the entire OS on a new kernel; Microsoft gradually rewrote their OS so it had a proper kernel with separate address spaces rather than with everything running in one address space.

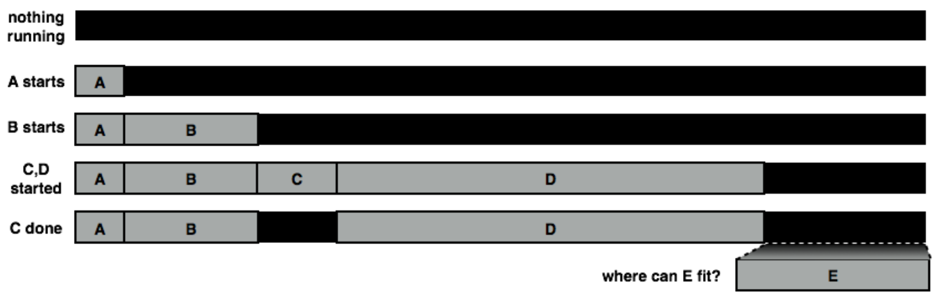


Figure 5.1: Memory external fragmentation. After the small program C finishes, there is enough space to launch E but not all in one place.

5.2 Key Concepts of VM

non-contiguous loading

One of the most important benefits of VM is *non-contiguous* loading of programs. If the entire address space of a program had to be loaded into an unbroken sequence of memory, as time went on and programs started and finished, memory would increasingly be broken up into small unusable fragments. Figure 5.1 illustrates *external fragmentation*, the situation where usable free memory is scattered outside areas in use. Though the total free space is more than enough to launch a new program E in the illustrated example, there is not enough in any one place for that new demand. Virtual memory fixes external fragmentation by allowing a program to have its memory use *non-contiguous*, i.e., while the program sees its address space as continuous without gaps, its actual real memory usage can be relatively scattered.

Figure 5.2 shows how in a scheme where memory is allocated in fixed-sized *pages*, which need not be contiguously allocated, as programs leave the system if they do not leave a big enough contiguous free space for a new program, it doesn't matter. The pages for the new program can be scattered around, as shown for program E. Notice the white spaces at the end of the address space of programs that do not use up their last page. This is *internal fragmentation*, memory wasted because it is not fully used by the program that owns it. On average, a program wastes half a page out of its total address space (or, more accurately, half a page for any subdivision of address space, e.g. the stack). Mostly, this is an acceptable price to pay for avoiding external fragmentation. For this reason, most VM systems allocate memory in fixed-size pages.

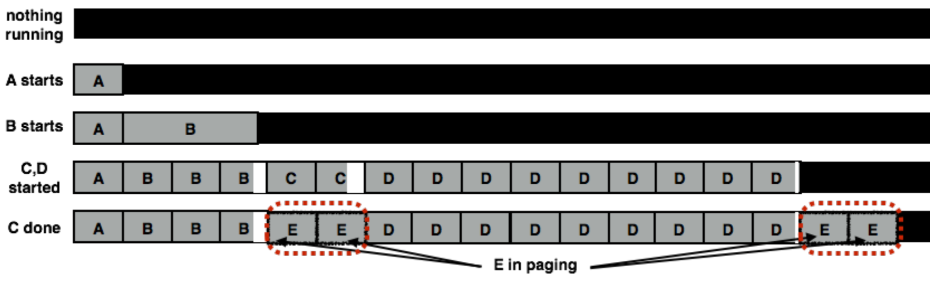


Figure 5.2: Paging and internal fragmentation. Now after the small program C finishes, there is enough space to launch E because it does not all have to be in one place. The price of fixed-size pages is a fraction of a page lost at the end of the address space when it is not an exact multiple of page size. This is called *internal fragmentation*.

Internal fragmentation is an issue for very large memory systems where very big page sizes may be desirable. In systems designed to run very large programs, a solution to this problem is to allow more than one page size.

The take home message? *Fixed-size pages solve the external fragmentation problem at the expense of internal fragmentation. Usually internal fragmentation is a relatively small fraction of total memory used – on average, half a page per subdivision of the address space – and is an acceptable price.*

address translation

A core component of any VM system is translation from the *virtual address space* (as seen by the program) to the *physical address space* (as seen by the hardware). This translation is not that different in concept from mapping logical block numbers in a file to physical block numbers on a device. However, a big practical difference is the fact that address translation is in the critical path of execution, i.e., if it is slow, it slows down the fastest part of the system, the CPU. Finding the right physical or device block in IO need not be super-fast because it is part of IO, which is already very slow.

There are various ways of representing the translation from virtual to physical addresses. Almost all systems today use fixed-sized pages, so I focus on that. In the past, some systems organised memory into variable-sized *segments*. A segment usually corresponded to a logical software component such as an

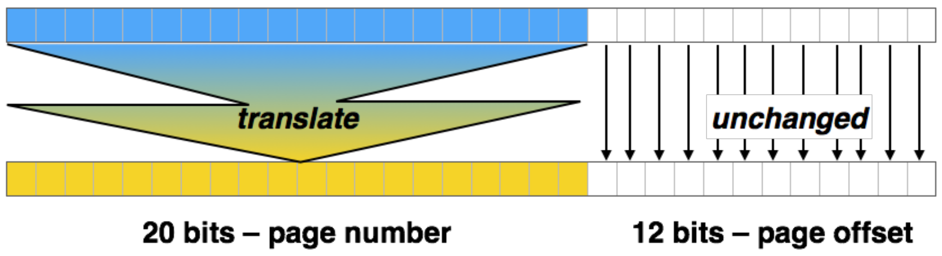


Figure 5.3: Page translation. The virtual address is translated to a physical address by looking up the physical page number that corresponds to the virtual page number. The offset within the page remains unchanged.

independent piece of code (such as a function) or a data structure. External fragmentation as well as the relative complexity of keeping track of segments has made this approach unpopular, though at least one family of machines had a successful VM system based on segments [Mayer 1982] and the Intel 286 architecture (the last before the 386 introduced 32-bit addressing) had hardware support for segments.

Another benefit of paging over segments is it is easy to do page address translation. If each page is sized a power of 2 bytes, then a fixed number of bits represents the *page number*, and remaining bits the offset within the page. Translation is then a matter of looking up the physical translation of the virtual page number; the offset within a page is unchanged. Figure 5.3 illustrates the general scheme for a 32-bit address, pages of 4KiB size. A 4KiB address range requires 12 bits so the offset is 12 bits and the page number is 20 bits.

Page sizes can vary a lot but to keep things simple, we will work with only one size, 4KiB. Page sizes are generally a power of 2 because this way, the page number and offset can easily be split within an address, based on the number of bits needed to represent that particular power of 2.

A page in physical memory is often called a *page frame*. I prefer to refer to physical and virtual pages to keep the distinction explicit. However since others use this terminology, I echo it when discussing real systems.

Heads up: *Actual page sizes can vary: 4KiB is a convenient size for illustrating examples, and is a size used on many real systems.*

In a paging scheme, a *page table* keeps track of page translations, usually for each separate address space (corresponding to a process). The two common schemes are a *forward* page table, usually implemented as a *multilevel* page table, and an *inverted page table*. I focus mainly here on forward page tables since they are the more common scheme.

An inverted page table is a hash table with one entry per physical page number. Virtual page numbers are used as a hash index and if there is no hash collision, the corresponding physical page belongs to that virtual page number. An inverted page table can be a lot smaller than a forward page table; it is possible to implement a single page table for the entire system. However in practice it turns out to very complex to implemement.

A forward page table is an array indexed by virtual page number, containing entries with a physical page number and status bits for each page. It may also include information on where the page is on the backing store (disk or SSD).

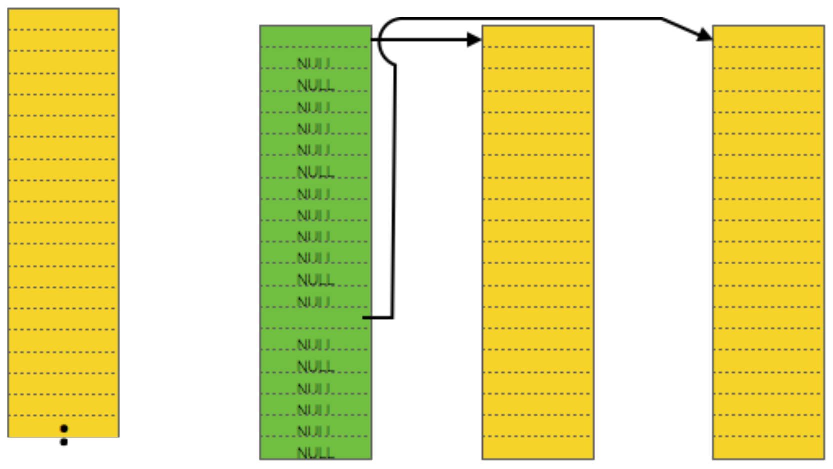
For a reasonably large address space with significant gaps a page table representing the entire address space will have a high number of unused entries. A multilevel page table solves this problem by splitting the address bits used to index the table. A second-level (or lower) table is split into chunks that need not be allocated if the part of address space covered by that chunk is not used.

To see how a forward page table works, assume a 32-bit address space. The size of the page table depends on the number of page numbers. In the example illustrated in figure 5.3, the page number requires 20 bits, meaning the table has to have $2^{20} \approx$ 1-million entries[1]. How big must each entry be? It must contain the 20 bits that replace the virtual page number, and status bits. At minimum, each entry needs to be 24 bits[2]; 32 bits is more convenient since 24 bits on some machines is not an efficient unit to fetch from memory. That means the page table would need 4MiB of memory. Every separate virtual address space would have to have such a table. If 100 processes were active at once, the page tables alone would use up 400MiB, a sizeable chunk even out of a few GiB of RAM. To make things worse, with 64-bit addressing, the page table would be enormous. With the same-sized pages, a 64-bit machine would need a page table with $2^{64-12} = 2^{52} \approx 4.5 \times 10^{15}$ entries. We need not even multiply this number by the number of bytes per entry to know it is impractically large for an in-memory table.

Figure 5.4 contrasts a single-level and two-level page table. If the single-level table was illustrated completely it would cover the entire address space. The two-level table shown mostly has `NULL` pointers in its L1 table and therefore only needs

[1]$2^{20} = 1,048,576$

[2]The absolute minimum status bits would be one for showing the page is valid and another showing whether it is modified relative to other levels of memory, but it is impractical to use data structures that are not allocated in whole bytes.



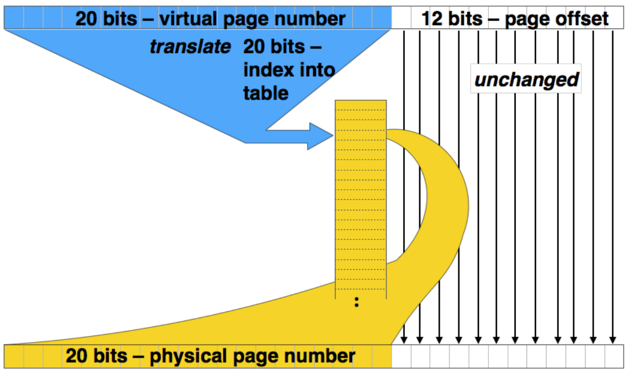
(a) Single-level **(b)** Two-level

Figure 5.4: Two forward page table schemes. In a two-level page table the first-level (L1) table contains pointers to multiple second-level (L2) tables, each of a subset of a single-level table. The L1 table contains `NULL` pointers to unused (hence unallocated) L2 tables. The single level page table if drawn completely would cover the same address space as *all* L2 tables combined with no gaps.

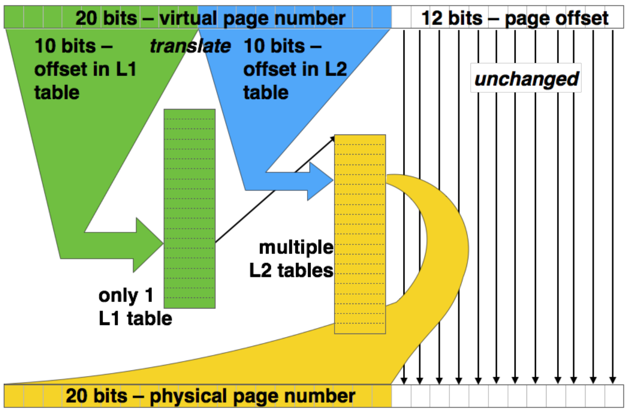
two L2 tables, a small fraction of the total space the L1 table needs. In a more realistic situation, there would be more L2 tables than this, but the total size of the table would in most cases be much smaller than a single-level table.

Figure 5.5 illustrates the difference in the lookup methods of a single-level and a two-level table. In the single-level table, the virtual page number can be used directly, at the cost of the table being very large. In the two-level table, the virtual page number is split in two. The high-order half is an index into the L1 table. Provided it does not index an entry with a `NULL` pointer, the pointer in the L1 table points to the correct L2 table and the low-order bits of the virtual page number can be used to index into the L2 table. Figure 5.5b only shows one L2 table to avoid clutter but in general there will be multiple L2 tables.

Why is the high-order part of the virtual page number used to index L1 and the low-order part to index L2? Doing it this way around, if a large stretch of the address space is not used (e.g., if the stack is put at high memory, leaving a big gap between it and the rest of the address space), any `NULL` pointers in the L1 table will represent a big range of addresses. Reversing the order of indexing will not have this useful feature of making it possible to skip large parts of the address space easily. What does it mean if an entry in the L1 table is a `NULL` pointer?


20 bits – virtual page number
12 bits – page offset
translate
20 bits – index into table
unchanged
:
20 bits – physical page number

(a) Single-level


20 bits – virtual page number
12 bits – page offset
10 bits – offset in L1 table
translate
10 bits – offset in L2 table
unchanged
multiple L2 tables
only 1 L1 table
20 bits – physical page number

(b) Two-level

Figure 5.5: Page translation. In the two-level scheme translation is in two steps. First, the correct entry in the L1 table is found and used to find the right L2 table. Then the translation is looked up as an offset into the L2 table,

Depending on the implementation of the page table this could indicate an invalid memory reference, or a need to allocate a new L2 table.

The two-level scheme generalises to more levels but two levels are sufficient for a 32-bit address space.

Heads up: *Only the lowest-level page tables contain page table entries. The higher-level (counting from level one, L1, as the top level) tables contain pointers to lower-level tables. The lowest-level tables each contain a fragment of what would be in a single-level table.*

To implement a page table, in addition to the page number translation, the page table needs *status bits* to represent the state of the page. There is a lot of variability in the status bits used. Here are some examples:

- *present* – the page exists in memory
- *dirty* – the page has been modified since being copied to the current level of the memory hierarchy (could also be called the *modified* bit)
- *no-execute* – this page may not be executed, i.e., it is data
- *accessed* – periodically unset and set only when the page is accessed, to keep track of pages not used recently
- *read-write* – indicates whether the page may be modified

Each bit can be on (set to 1) or off (set to 0).

The present bit is used to check if the page can be accessed. If the bit is off, that means the page is not in memory and has to be fetched from backing store. The dirty bit is set whenever a page is modified. If the page loses its memory occupancy, it has to be copied to backing store so changes aren't lost. It is not practical to do this every time the page is modified (this is called *write through*) because backing storage is so slow compared with RAM, let alone the CPU. Copying a page to backing store can be done periodically when the IO system is not busy to reduce the need to do a *write back* when a page is evicted from memory: this is called *cleaning* the page. The no-execute and read-write status bits have some overlap – but for finer-grained security policies it is useful to distinguish the cases. We will see more on how the accessed bit is used when looking at *replacement policy*.

How could we code page translation and checking or modifying status bits in C? We need bit-level logic operations. Let us look first at separating out the page number and offset. What we need to do is to split a 32-bit number into the lower 12 bits and the upper 20 bits. We also need to adjust the lower 20 bits so they are in the low 20 bits of a word, so they represent a number in the range $0\ldots2^{20}$ to use as an index into a single-level page table. Splitting the virtual page number again to form an L1 and an L2 index is easy once we know how to dp this.

First, eliminating unwanted bits can be done using a *bit mask* (often shortened to *mask*). The idea derives from the logic identities $A \wedge 1 = A$ and $A \wedge 0 = 0$. We do a bitwise **and** of a word with a word-sized mask containing 1s in the positions we want to keep. This results in zeroes everywhere except where there is a 1 in the mask.

operator	logical effect	example
`&`	bitwise **and**	`A & 0x7` extracts low 3 bits of `A`
`|`	bitwise **or**	`A | 0x1` == `A` except lowest bit is always set
`~`	bitwise **negate**	`~A` inverts bits of `A` (1's complement of `A`)
`^`	bitwise **exclusive or**	`A ^ B` contains 1s only where `A` and `B` differ
`<<`	bitwise **left shift**	`A << 7` shifts `A` 7 places to the left (== $\texttt{A}\times 2^7$)
`>>`	bitwise **left right**	`A >> 7` shifts `A` 7 places to the right ($\approx \texttt{A}\div 2^7$)

Table 5.1: Bitwise operations useful for masking. Most C compilers will use a logical right shift for unsigned types and an arithmetic right shift for unsigned types.

In C, we write a bitwise **and** using the `&` operator – not to be confused with the memory reference (pointer creation) operator spelt the same way. That is distinct from logical **and**: `&&`. See table 5.1 for C bitwise operators.

For example, if we want the low 4-bits of a byte containing `01101110`, and want the rest to be zeroed, our bitwise **and** looks like this:

```
01101110 &
00001111
00001110
```

In C code, we can initialise the mask using hex notation (I use `unsigned char` because that is C's 8-bit unsigned `int` type) :

```
unsigned char mask = 0xF;   // binary: 11110000
unsigned char value = 0x6E; // binary: 01101110
```

That technique, adjusted to 32 bits, tells us how to extract either the low 12 or the 20 high bits. We need to construct a mask containing 32 bits with the relevant bits set to 1. The C type `unsigned` is a synonym for `unsigned int` since that is the most common unsigned type used. We can defined the masks as follows, and also the number of the bits in the offset, which we need soon:

```
unsigned offsetmask = 0x00000FFF;
unsigned pagemask   = 0xFFFFF000;
unsigned offsetbits = 12;
```

Putting the leading zeros in `offsetmask` is not necessary but makes it easier to see how the two masks differ.

It can also be convenient to use a preprocessor symbol to define a mask. This makes it a constant value seen by the compiler but not allocated memory unnecessarily. Doing this is convenient if you want to put a mask in a header file and do not want it to be created as a new variable every time the header is included (which requires working around the fact that the linker will see it as a duplicate):

```
#define OFFSETMASK 0x00000FFF
#define PAGEMASK   0xFFFFF000
#define OFFSETBITS 12
```

I prefer to spell preprocessor symbols in all capitals to distinguish them from variables, functions and types. That is just a convention; the preprocessor does not care how they are spelt. For the human reader, this is a useful convention, and is widely used by C programmers. Preprocessor symbols are expanded before the compiler sees the code; think of them as if you used search and replace in a text editor to replace the symbol by what it stands for. From here on, in examples I use these symbols instead of the variables I defined previously.

With either of these approaches, finding the page offset is easy; I will use the preprocessor symbols from here on:

```
   // address known here and of type unsigned
   unsigned pageoffset = address & OFFSETMASK;
```

How do we find the page number? We can use our other mask, then shift the resulting number right. But we do not really need to do the first step because shifting the number right will also make the low 12 bits disappear. So here it is in one step:

```
   unsigned pagenumber = address >> OFFSETBITS;
```

This works on most C compilers because a right shift of an unsigned value is interpreted as a *logical right shift* if the value is unsigned. To be safe, using both steps in the *opposite* order to that you may feel is natural ensures that even if the compiler generates code for an arithmetic right shift[3] you get the correct result. For this to work, you need a version of `PAGEMASK` shifted to the low end of the word:

```
#define PAGEMASKSHIFTED   0x000FFFFF
```

[3]An arithmetic right shift shifts copies the sign bit to the right, rather than filling with zeroes from the left, on a machine that uses two's complement for negatives.

Finally here is the correct (safer) code:

```
   unsigned pagenumber = (address >> OFFSETBITS) &
                         PAGEMASKSHIFTED;
```

Preprocessor symbols can also define *macros*, a piece of text with parameters that you fill in when you use them. This is different from a function in that macro expansion happens in the preprocessor stage *before the compiler sees your code*. So you can "pass in" arbitrary chunks of text, including type names. For example:

```
#define RIGHTSHIFT(A,N) A >> N
```

To use the correct terminology, a plain preprocessor symbol is a `macro` and this new notation is a `function-like macro`. However, I find it tedious to use such a long name and so I prefer to use *preprocessor symbol* for a regular macro, and *macro* for a function-like macro.

Note that it is *essential* not to have any space before the open "(" of the parameter list, otherwise the parameter list will be treated as part of the text the symbol stands for.

If you use `RIGHTSHIFT` anywhere in your code, whatever appears in parentheses after the name gets substituted in. It is common practice to put parentheses around the right hand side so the macro expansion can be done without worrying about operator precedence:

```
#define RIGHTSHIFT(A,N) (A >> N)
```

With this defined (usually in a header though that is not essential since a macro generates no runtime resources except by virtue of being expanded), we can rewrite our example as:

```
   unsigned pagenumber = RIGHTSHIFT(address,OFFSETBITS) &
                         PAGEMASKSHIFTED;
```

Macros can be a big convenience but can also lead to code that is hard to understand so I use them sparingly. Do not expect to find a lot of them in my examples.

Heads up: *Function-like macros can be a very useful feature to avoid writing repetitious code where you cannot write a function. But they can lead to unreadable, unmaintainable code so beware of using them extensively.*

I am assuming here that I know that a machine address is 32 bits and the page offset needs 12 bits. Can we generate these masks in a more general way, rather than hard-coding them? In principle, yes, because we can use `sizeof` to determine the size of any pointer type, which tells us how many bytes a memory address is. We can then use bitwise operations to construct masks with the requisite number of 1s. Assume we have a data type `mask_t` that has enough bits to contain the mask. This is how we can go about constructing a mask with *N* ones in the low end of a variable of type `mask_t`:

```
mask_t mask = 0;
for (int i = 0; i < N; i++) {
   mask <<= 1; // shift left 1 position
   mask  |= 1; // make the low digit 1
}
```

To create a mask at the high end of the word, the body of the loop changes to this:

```
   highmask >>= 1;          // shift right 1 position
   highmask  |= 0x80000000; // make the high digit 1
```

If we need a mask that does not start at the high or low bit, we can create it at either end of the word using one of these techniques then shift it to where we want it.

On the whole though it is easier to hard-code masks in the place where we set the size of pointers in global system data structures that relate to paging, since this is not something that changes often.

What about extracting status bits? That is a very similar concept. If we want to test if a bit in a given position is 1, we make a mask that only has that bit set. Then a bitwise **and** will reveal whether that particular bit is set in the status.

If we have a pointer to page table entry, `PTentry`, how to we use this to calculate the physical address? First, we check the status bits. If the page is present, we are good to go, otherwise we signal a *page fault*, signifying that the OS must handle the case where the page is not in memory. If the page is in memory and we are doing a write (the data reference of a store instruction in a RISC architecture), we turn on the dirty bit. Finally, we can translate the virtual page number to physical and combine it with the offset to create the real address.

Here is an outline of code to do this, assuming we have defined preprocessor symbols for the various status bits:

```
address_t translate (PTable * table,
                     address_t v_address, bool write) {
   address_t virtual_page = get_page(v_address);
   address_t *entry = get_entry (table, virtual_page);
   if (!entry) {
      // SIGNAL MEMORY FAULT
   }
   if (*entry & PRESENT) {
      if (write) { // set dirty bit
         *entry |= DIRTY;
      }
      address_t physical_page = (*entry >> OFFSETBITS) &
                                PAGEMASKSHIFTED;
      address_t offset = v_address & OFFSETMASK;
      return (physical_page << OFFSETBITS) | offset;
   } else {
      // SIGNAL PAGE FAULT
   }
}
```

What happens if a page fault occurs? Since accessing backing store is so much slower than CPU operations or accessing RAM, most operating systems suspend the current process while waiting for the missing page to be fetched. Another complication is that if physical memory is almost full, another page has to be chosen as a *victim* for *replacement*. Replacement is complicated, so I handle that separately in §5.3 More Advanced Concepts. In a real OS, page replacement happens long before real memory is really used up to avoid a situation where handling page replacement itself runs out of memory.

The take home message? *A page table provides a mechanism for translating from virtual to physical page numbers. C provides simple mechanisms for bitwise operations so where manipulation of table entries must be done in software, C works well. I cover single-level and two-level page tables in some detail here. A real page table will contain other details I have omitted for clarity.*

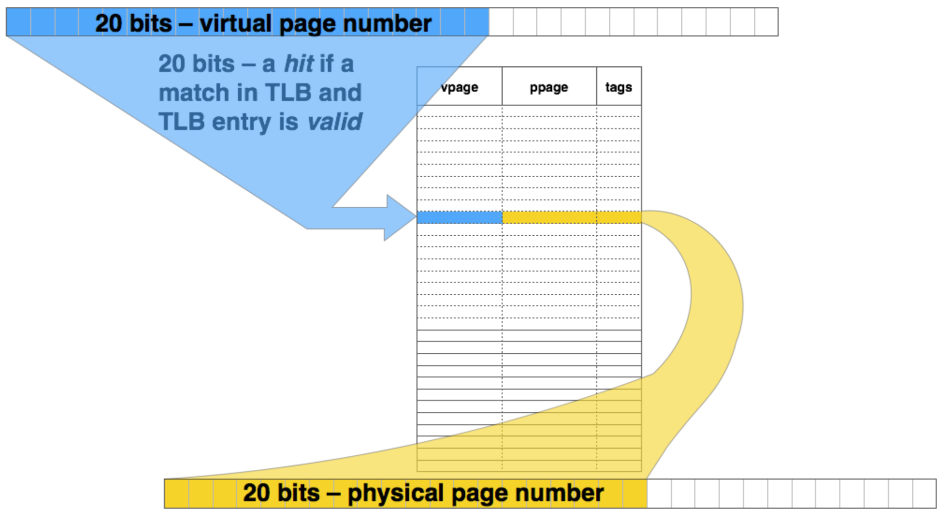

20 bits – virtual page number
20 bits – a *hit* if a match in TLB and TLB entry is *valid*
vpage
ppage
tags
20 bits – physical page number

Figure 5.6: A TLB hit. If the TLB finds the desired virtual page number in its subset of the page table, it provides the physical page translation. A TLB entry can be marked *invalid* so a match is only reported if the valid bit in the tag is also set.

hardware support

For all this to work, hardware support is essential. Every time an instruction is fetched, its address needs to be translated, as do branch and jump targets and addresses used to reference data (loads and stores on a RISC architecture; other types of instruction set may access memory in other ways – and that makes things even more complicated). If page table lookups were implemented purely in software, a VM machine would run many times slower than a machine that only accessed memory physically.

A common solution is to keep a small fraction of the current page table in a very fast, specialized cache called a *translation lookaside buffer* or *TLB*. The TLB can be looked up fast enough that it does not slow down processing in the case where the required translation is in the TLB. A TLB generally is relatively small so this speed is possible. Sizes typically range from 32 to 128 entries, and it is also common to have a separate TLB for data references (DTLB) and instruction references (ITLB) for three reasons:

- instructions and data are often separated out into distinct parts of the address space and there may be protection against executing data as code
- data and instruction referencing often follow very different patterns – e.g., a

tight loop may access a very narrow range of code addresses while stepping through a large data structure.

- the first-level (L1) cache is often split into instruction and data caches that are separately accessed and since the TLB is the critical path for L1 cache access (i.e., it limits how fast the cache can be accessed), it makes for a simpler design if the TLB is also split in the same way

When an address appears in the CPU, the first step is to check if that address is found in the TLB. The virtual page number is compared in hardware with virtual page numbers stored in every location in the TLB where that virtual page number could be stored. If it is found, the page translation continues in a hardware implementation of the software algorithm I describe. If it is not found (a TLB *miss*), the correct page table entry must be found, a location in the TLB selected for this translation and the page table entry is installed in the TLB. If the page table entry has to evict another entry, the victim entry's status has to be copied back to the page table in case it has modified any status bits. On some systems, there is hardware support for looking up the page table to handle a TLB miss. Even with this hardware support, a TLB miss can cause significant slowdown, so minimising TLB misses is essential for good VM performance.

A TLB may contain tag bits in addition to those in the page table, including a *valid* bit to signify that the location in the TLB contains a valid page translation. When a different process is given control of the CPU, a simple strategy is to *invalidate* the entire TLB, which can be done by setting each valid bit to zero. With this strategy, it is not necessary to keep track of which process or page table is represented in the TLB.

Heads up: *The TLB relies on locality at page granularity. A program that access a lot of pages, even if the total memory it accesses is small and mostly fits in the caches, can have very poor performance arising from a high number of TLB misses.*

The TLB does not eliminate the need to access the page table in software. Even on systems where looking up the page table to handle a TLB miss is done in hardware, page table contents must be initialized in software, and handling a page fault is usually done in software. So the techniques given for C coding using bit operations do have real use, even if the common case is that the TLB handles the translation in hardware, and finding a page table entry that is not in the TLB is also often handled in hardware.

Because hardware support is so critical to achieving acceptable performance, page table design is relatively inflexible. An OS pretty much has to use whatever options are on offer from the hardware design. At best, the OS designer can make creative use of the provided design, at risk of making design choices that do not work with a hardware upgrade or bug fix.

The take home message? *The TLB makes page table access affordable. If the common case is a TLB hit and no page fault, VM works as designed, with very little performance penalty over accessing real memory directly. The importance of hardware support for acceptable performance means the OS designer cannot choose a page table organization not anticipated by the hardware designer.*

replacement policy and locality

Choosing which page to *replace* (evict from memory) is an important design consideration in VM.

Locality is an important principle at all levels of the memory hierarchy. As with caches, VM relies on a combination of two types of locality:

- *temporal locality* – any item referenced is likely to be referenced again soon; this implies a page, once resident, should be kept in main memory as fast as possible and that a TLB entry should be kept in the TLB as long as possible
- *spatial locality* – when an item is referenced, items close to it are likely to be referenced; this implies that pages should be big enough that neighbouring items that will be needed soon are likely to be in main memory after a page is fetched from backing store

A “memory reference” means any of an instruction fetch or either kind of data operation: a read or a write.

What makes VM different from caches is the huge penalty for accessing the slowest level. For this reason, pages are a lot bigger than the units used in caches (called blocks or cache lines), typically 4KiB or bigger. Also, because of the high cost of a mistake, it is worth managing what is in main memory in software.

How to decide which page to evict when memory runs out is a critical aspect of the performance of VM as evicting the wrong page, one that is needed soon, has expensive consequences. For this reason, a clear understanding of locality is important if page replacement is to be implemented efficiently.

A key insight into how to manage main memory occupancy is the *working set*, originally defined as follows [Denning 1968]:

a working set of pages is the minimum collection of pages that must be loaded in main memory for a process to operate efficiently, without "unnecessary" page faults

An operational definition of a working set is the set of pages accessed over a defined time period. The exact time period chosen is tuned to take into account the relative cost of accessing backing store, as well as typical characteristics of programs. In general, the aim is to maximising the number of processes that can make progress by giving each the biggest possible amount of physical memory, within the constraints of available memory.

If the working set is bigger than it needs to be, some processes will be allowed more memory than they absolutely need to make progress, forcing other processes to incur more page faults than necessary.

Accurately calibrating the working set is useful for a *global page replacement policy*: the next page selected for eviction should be one that is no longer in a process's working set. That can be determined by keeping track of how recently each page has been accessed, as well as how many pages each process has in main memory. In practice, a policy exactly like this is impractical to implement as it would require time-stamping all memory references (at least the most recent to each page), and an efficient way of finding the least recently referenced page.

The absolutely ideal replacement policy is the one that evicts the page next used furthest in the future. That is even less practical as it requires foreknowledge of the exact order of memory accesses into the future. Here are policies in order of increasing simplicity to implement:

- *optimal* – not practical to implement but can be simulated to compare against realistic policies: the optimal strategy replaces the page used furthest in the future

- *least recently used* – or LRU: based on locality principles, the page last used longest ago is likely not to be needed soon

- *first in first out* – or FIFO: easy to implement as pages can be kept in a simple list and the page at the older end of the list is evicted first

FIFO is not a great approximation to optimal because the oldest page may also be one that is referenced often, e.g., if it contains a critical data structure. It is possible to construct an order of references to pages that makes a FIFO scheme have *more* page faults if memory size increases. This effect is called *Belady's Anomaly*, after the person who discovered it. LRU is impractical to implement exactly but can be approximated using a *clock algorithm* [Carr and Hennessy 1981]. The clock name derives from visualising all physical page frames arranged in a circle with a pointer like a clock hand moving around the circle. When a page has to be replaced, if the pointer is pointing at a clean page (dirty bit not set) with the accessed bit not set, that page becomes a candidate for replacement. If not, the pointer moves on. Each time it encounters a page with the accessed bit set, it unsets the accessed bit. Each time it encounters a dirty page, it schedules it for cleaning (writing back to backing store). Eventually it will encounter a clean page with the accessed bit off, and that page becomes a candidate for replacement.

The clock algorithm works on the assumption that pages that have their accessed bit set are not likely to be least recently used, if the accessed bit is set on every reference. While a dirty page may also have not been accessed recently, it is more efficient to schedule it for cleaning so it will be ready next time the clock hand reaches it than to write it back at the same time as another page has to be read from backing store.

A further refinement on approximating LRU is to maintain a *page standby list*, a list of recently selected victim pages, and only evict the oldest page on that list. That way, any page erroneously selected as a candidate for eviction can be rescued if it is needed soon after being selected as a victim.

Real systems try to balance the requirements of each process against global free memory and approximate working set by calculating a *resident set* – pages used over a predefined time interval. A process in general should not drop below its resident set in main memory but can be expected to give up pages in excess of its resident set. A global strategy for page replacement can balance requirements across processes, using this principle.

The take home message? *Page replacement policy is a trade-off between approximating the optimal strategy and ease of implementation, with approximations to LRU a good compromise.*

protection

If each process has its own address space, the OS can ensure that no process, no matter what bugs there are in its code, can access another process's address space. In practice, this level of protection can be thwarted by exploiting security holes in system calls, which run in kernel space and hence are not protected, except in a true microkernel OS.

Before VM with separate addresses spaces was commonplace, a bug that took down a whole system was commonplace. Early versions of the Mac and Windows suffered from this.

The take home message? *Without hardware protection, memory addressing, particularly with languages like C that do not manage memory automatically, makes for unreliable systems. Even if memory is sufficient not to need swap space, VM is worthwhile for protection.*

5.3 More Advanced Concepts

efficient process launching

As we see where we study processes, when a process is launched, the fundamental operation is copying an existing process's address space. While more efficient approaches to launching a new process have been devised since early days of Unix, it is still sometimes necessary to copy the entire address space when launching a process that duplicates the code of an existing process (e.g., to split a workload between similar processes running independently).

Copying the entire address space is inefficient particularly as each process will start out the same then start modifying memory where they differ in their behaviour. One approach to this problem is *copy on write* (*COW*): instead of copying the address space, the page table is copied and each entry is marked as a *COW* page (a COW status bit is needed for this). When a page is modified, it is copied and the COW status cleared in all page tables pointing to it.

COW has another use: zeroing memory before launching a process. If all pages initially point to a COW page that only contains zeroes, the entire address space appears to the code as if it is zeroes. As soon as anything is modified, the zeroed COW page is copied, the page table entry for the page that was modified changes so it is no longer a COW page and points to a new physical page frame that has to be allocated by the OS. In this way a very large address space can be zeroed without actually having to allocate the pages or execute the code that fills them with zeroes, until a particular page is modified at which time it has to be allocated a new physical page frame and copied.

The take home message? *Copy on write (COW) allows for efficient initialisation of an address space to zeroes as well as efficient copying of the address space of a process, which is needed when a process is a duplicate of another, usually to split a large workload.*

sharing

Sometimes forcing each process into a separate address space is too restrictive and sharing information between processes is required. One mechanism for this is a *shared segment*, a region of the address space that is available to more than one process. In the Unix world, there is a standard for creating a region of shared memory and attaching it to another process. The approach used relies on the same permissions system as is used for files so that it should not be possible for someone without the correct permissions to see or modify a shared segment.

Here are some system calls used for shared segments.

- `shmget` – obtain a shared-memory identifier to use with other system calls; can also create the shared segment
- `shmctl` – adjust permissions of the shared segment; can also remove a shared segment
- `shmat` – map (“attach”) a shared segment into the current address space – usually you supply a `NULL` pointer to the system call and let it decide where to place the shared segment
- `shmdt` – unmap a shared segment from the current address space

There is a fair degree of complexity in setting up these calls so I skip the detail here: this is a good example for an exercise in reading documentation.

Another approach to sharing is to *memory map* a file to the virtual address space. The effect of memory mapping is as if the file contents were in memory and sharing is an option, with a result similar to a shared segment. The main difference is that since the sharing happens though a file, the memory mapped file can be reloaded if the process dies or quits and then restarts. While it is possible to memory map a file to a specified address, implementation of this varies so it is better to leave the placement in the address space to the `mmap` system call.

Here are some system calls used for memory mapping:

- `mmap (void *addr, size_t len, int prot, int flags, int fd, off_t offset)`: for this one example I supply some detail to give a taste of how such system calls are set up:
  - `addr` – start address in the address space for mapping: easiest to set as 0 (`NULL`) so the system can choose where to place the memory mapped content
  - `len` – how many bytes to copy from the file to the memory-mapped region (may be rounded up to a whole number of pages)
  - `prot` – set the protections that apply (access, read, write, execute)
  - `flags` – controls extent of sharing or not with other processes
  - `fd` – file descriptor: a number associated with an open file (usually; their are some other options)
  - `offset` – how far into the file to start mapping
- `munmap` – given a start address and length, removes a memory mapping from the address space

As with shared segments, the system calls are quite complex but I supply more detail in this case to illustrate the general idea. As with the shared segment calls, you should look up documentation to understand them.

Whether with a shared segment or a memory-mapped file, sharing should generally not use pointers to data in the shared memory that assume a fixed location in the address space because there is no guarantee that any other process will see the shared data as being in the same part of its address space.

The take home message? *Shared memory or a shared memory-mapped file provide ways for processes to exchange information. See more ways for processes or other types of tasks to share information in Chapter 6.*

Performance: OS Design and User-Level Programming

Mostly, when we program, we do not need to think about memory hierarchy because locality (temporal and spatial) takes care of things for us, but we can write code that trashes these assumptions. For example, the following

```
for (int i = 0; i < 1024*1024; i+=1024)
   a[i] = b[i] + i;
```

makes poor use of the TLB. Why? Assume an `int` is 4 bytes. Then each memory access is 4KiB apart. On a machine with a 4KiB page size, each reference to `a[i]` and `b[i]` will be on a different page. The code therefore will reference 2048 different pages (1024 pages for each of `a[i]` and `b[i]`). Since each iteration uses 2 new pages, the TLB will exhibit no locality and every loop iteration will experience two TLB misses. If the arrays have already been accessed and are in caches and main memory, this code could run anywhere from 10 to 100 times slower than a loop with much better locality like the following (which does the same work but on adjacent data items):

```
for (int i = 0; i < 1024; i++)
   a [i] = b [i] + i;
```

In this particular example, the first version of the loop will also have poor locality for all levels of the memory hierarchy since it could incur a page fault and cache miss for every memory reference. However, if another loop repeats the same memory accesses soon after, the second version will not have page faults and not many cache misses, but will still incur about as many TLB misses, since the TLB is unlikely to be big enough to hold 2048 page translations.

It is also useful to understand the limits of virtual memory. If you have a very large data set that is too big for main memory, VM will cope with this but not necessarily optimally. For example, if you have to sort data that is too big to fit in RAM, you could just use the standard quicksort library algorithm and hope that VM will save you. In practice, you may achieve better performance with a disk-based merge sort, in which you explicitly move data between disk and RAM. Why? Because you can do your data transfers in much bigger units than one page, because you know the order your data is going to be accessed.

The take home message? *Even with the TLB where performance penalties are nowhere near as high as unnecessary page faults, a program can suffer significant slowdown if you do not understand how VM works when coding. The memory hierarchy is designed to work well in the common case, but you need to understand where the design assumptions can go wrong to avoid serious performance pitfalls.*

5.4 Examples

An inverted page table is used in IBM's POWER architecture and the less expensive PowerPC derived from it.

It is common for processors to have hardware support for page-table walking so that time for TLB misses can be minimized. The MIPS family of processors, back in the day when they were used for high-end systems, had software management to TLB misses. The design trade-off: hardware TLB miss management is much faster, but allows no flexibility to adjust policy on what should be in the TLB, or page table design [Nagle et al. 1993].

Intel's x86 range uses a hardware-managed TLB that in earlier designs assumed the TLB represented a single address space and had to be flushed whenever there was a switch to a new address space. To make virtualising more efficient, more recent Intel designs have added tags to each TLB entry to identify which process an entry belongs to so more than one address space can be represented at once in the TLB [Neiger et al. 2006]. In the simplest case, the design defines a two-level page table and TLB entries are assumed to relate to exactly one page table so changing to another process requires flushing the TLB.

Since performance efficiency is tied to using the available hardware support, a portable OS like Linux has to have completely different page table code for different architectures. The inverted page table organization on a PowerPC, for example, is so different from a two-level page table on an Intel machine that there is very little shared code. While one of the key strategies for achieving portability is to aim for hardware-independent design wherever possible, this is one area where that goal is frustrated by reality.

Linux page replacement has gone through several iterations [Van Riel 2001]. It is difficult to describe the Linux approach in simple terms because it combines page management with managing disk buffering (file cacheing) and memory-mapped files. Linux page replacement uses the *page frame reclaiming algorithm* (*PFRA*). PFRA starts by selecting page frames not owned by any process, such as disused memory mapped files [Bovet and Cesati 2005, Chapter 17].

The take home message? *There are many variations on page table management and replacement strategy but a practical OS must base its strategy on the underlying hardware. Even a portable OS like Linux has significantly different code for each different CPU architecture it runs on.*

Exercises

1. A program accesses about 1MiB of data, in steps of 4KiB. This data fits the L2 cache and the penalty for accessing the L2 cache is not high and does

not cause a significant speed loss. The data is all in main memory so there are no page faults. Explain what could cause the program to run about 10 times slower than with an ideal memory hierarchy.

2. Explain the trade-off between internal and external fragmentation.

3. As relates to extracting and processing components of a page table entry:

   (a) Write C code to calculate the page number mask and number of offset bits given the page size in bytes and the fact that `sizeof` can calculate the size of a pointer type.

   (b) Assume a page table entry is stored as an `unsigned int` and the high 20 bits are the page number and the next 4 bits are status bits, representing present (P), dirty, (D), accessed (A) and no-execute (NX).

      i. write out masks in both `unsigned int` initializations and preprocessor macro format needed to extract each of the page number, the P, D, A and NX bits.

      ii. write code to use a mask (either format) to extract a page number from a page table entry and right-shift it to the low end of a word

      iii. write out a mask in both formats again to extract the page offset from an address

      iv. write out code to replace the virtual page number in an address by the physical page number, given the address and the page table entry containing the page translation

      v. write out code to check a given bit in the status bits, given the mask and a page table entry

      vi. write out code to set a given bit in the status bits, given the mask and a page table entry (your code should not change anything else when setting the bit)

      vii. rewrite the examples in the chapter that use left and rights shifts using function-like macros and comment on whether this aids readability (base your macros on `RIGHTSHIFT` on page 82)

4. Explain advantages and disadvantages of each of the following page replacement strategies:

   (a) optimal

   (b) FIFO
   (c) LRU
   (d) clock

5. When an entry has to be replaced in the TLB, explain steps necessary to ensure the page table is correct before that entry can be overwritten. Take into account possible values of the tag bits.

6. Two processes, `reader` and `writer` share information between them through shared memory. Give the system calls first for a shared segment then for a memory-mapped file to create such a shared-memory region in the `writer` process, make it available to the `reader` process then remove it from the system in the `writer` process. You need not worry about actually using it or ensuring that the other process has finished before removing it.

7. A cache usually contains tags based on the physical address to identify which memory locations a particularly cache block contains (a *physically-addressed cache*). In a *virtually-addressed cache*, the tags are based on the virtual address. A virtually-addressed cache reduces the importance of a TLB in achieving performance at the expense of making aliases (different virtual addresses that refer to the same physical address) harder to handle. Discuss advantages and disadvantages of a virtually-addressed cache taking into account these points; include examples of how aliases could occur.

8. Discuss why the page table strategy required for a PowerPC processor cannot simply be used on an Intel CPU.

9. A page table entry is 32 bits and a pointer is 32 bits. If an address space consists of the following range of virtual page numbers, calculate how big a single-level and a two-level page table are, and comment on the difference.
   - *global variables* – 1–256
   - *constant pool* – 4,096–8,191
   - *heap* – 32,768–131,071
   - *code segment* – 262,144–327,679
   - *stack* – 523,778–524,287

10. What data structures would you need, in addition to a page table for each process, to implement a global clock algorithm?

11. Sketch out system calls to do each of the following (see page 91 for shared memory system calls; look up the details or read the `man pages`):
    - Create a key for a shared segment
    - Use the key to create a new shared segment
    - Attach the shared segment to the process that created it
    - Attach the shared segment to another process
    - Put a data item into the shared segment in the creating process
    - Extract a value from the shared data item in the shared segment in the a process that did not create the data item
    - Remove the shared segment.
12. Write code to (see the system calls for memory mapped files on p 91):
    - Map a newly created file to memory
    - Map it to the memory space of another process
    - Put a data item into the memory mapped region in the creating process
    - Extract a value from the memory mapped region in the a process that did not create the data item
    - Unmap the file.
13. Which is easier to use: a shared segment or a memory-mapped file? Why?

6 Parallel Programming

PROCESSES ARE THE FUNDAMENTAL UNIT of work allocation in an OS. A process is the representation of a program when it is running. Each process in a modern operating systems has a separate address space meaning that launching a process requires a new page table and a new allocation of physical memory, corresponding to the process's minimum requirements to run. A thread is a lighter-weight concept: a separate thread of execution that can be scheduled separately without a new address space.

I cover memory issues in Chapter 5; the important details we need here are that in most operating systems, a process has a separate address space represented by a *page table* that translates a *virtual address* (in the address space as seen by the process) into a *physical address* (in the address space of the actual real hardware).

In Chapter 3, I cover how processes and threads are scheduled. Here I focus on what processes and threads are, how they are launched, how they share information and how that information sharing is managed. The focus in this chapter is on user-level programming, not on how threads and processes are implemented in the OS.

Processes and threads are not the only way to achieve parallel execution. Another approach is a *distributed system*, in which processing can be spread out over a network of computers. This is a large complex subject, so I only introduce the basics and relate the concept to the growing trend of cloud-based computing.

To start, I outline the major concepts then go on to expand on each of thread and process launching, sharing and communication and synchronization, mostly using examples from Unix-style systems. I end with a brief overview of distributed systems and how they relate to the cloud.

6.1 Concepts

A process is the embodiment of a running program, in its simplest form. However a process can *spawn* another instance of itself (or in the Unix world, *fork*). More accurately a process is defined as a separately scheduled unit of execution in its own address space. Ordinarily, that is exactly what a program is embodied as when it runs. It has an address space that is distinct from any other instance of the same program that could be running simultaneously, and is scheduled as an entity. The ability to create more than one instance out of an existing process is useful for spreading out a large workload.

Why would you want to split a workload into separately scheduled components?

First, you may have more than one CPU available (almost always the case, since multicore systems become commonplace) and splitting the workload means each separate component can work in parallel. Second, a single thread of execution could be blocked, e.g., waiting for an IO event, and splitting the workload means other parts can continue without being stalled.

Why do we distinguish processes and threads?

In earlier designs, the kernel could only schedule processes. A process is a relatively heavyweight unit to manage, with a page table corresponding to its address space and data representing resources it controls like open files. Switching between processes could involve expensive operations like flushing the TLB (resulting in TLB misses until the TLB refilled). The overheads of switching between processes discourage writing multi-process code except in situations involving a relatively large workload. For this reason, threads were invented.

Early version of threads were implemented strictly at user level, though they used system calls for purposes like setting a timer so a thread could be preempted. In recent systems, thread scheduling is implemented in the kernel. While managing threads in the kernel increases overheads in the OS compared with user-level thread management, the OS is able to use tricks like scheduling related threads simultaneously (see *gang scheduling*, page 29), which it cannot do if it is not aware of threads.

There are various different approaches to implementing threads. The approach most widely supported across different platforms is *Pthreads*, based on the *POSIX* standard (Portable Operating System Interface), an attempt at providing platform-independent abstractions based on Unix-style functionality. Pthreads is one of the earliest POSIX standards [Mueller 1993] and is widely implemented across

Unix variants including Linux and Mac OS X, as well as Windows (though not supported by Microsoft, who have their own threads implementation).

In some systems, Pthreads are layered on top of a lower-level thread implementation; for our purposes, we will use Pthreads as if it is a native API, since this chapter is about using threads not how they are implemented.

There is more variation across operating systems in how processes are launched. In Unix-type systems, the standard is a system call `fork`, which copies the entire address space of a process and launches a new copy with the newly copied address space. When the new process being launched is a new program, the `fork` system call is immediately followed by an `exec` system call, which replaces the address space by that needed to launch a new executable file.

In Windows, more complex system calls are used to create a process. The Unix philosophy of separating out `fork` and `exec` is designed to allow arbitrary code to run between the two calls so neither needs to have a complex interface. The consequence of this simplicity is that the `fork` system call possibly needlessly results in an entire address space being copied only to be wiped out by `exec`. For this reason, some systems implement an alternative, `vfork`, which does not copy page tables, with the expectation that a variant on `exec`, `execve`, will be called to create the new process's address space. We will not explore all the complexities of variants on `fork` and only look at the simple case of creating a duplicate of the parent process.

Why would anyone want to call `fork` and `exec`, only to waste a copy of the parent process's page table? This sequence is common in a *shell*, the simple command interpreter that runs in a terminal session. If you type the name of an executable file on the command line, the shell calls `fork` creating a new copy of itself, and that new copy calls `exec` to launch the new program. Since the shell is a very small simple process, it does not have a very large page table, so copying it is not a very big operation.

Aside from these differences in setup, how else do threads and processes differ?

Because a thread is not launched in a new address space, it can share information with its parent process and other threads in the same process through ordinary variables. Cooperating processes cannot do so because they do not see each other's address space and must instead use one of the following mechanisms:

- *shared segment* – as outlined in Chapter 5, processes can create a shared segment that can be made known and accessible to other processes

- *memory-mapped file* – also in Chapter 5, one or more processes can memory map a file and share information through memory that way
- *pipes* – a pipe on the command line (using the "`|` symbol) can also be implemented in code: "file" output can be sent to one end of a pipe, while "file" input can be received from the other end of a pipe
- *sockets* – a generalisation of pipes that allows the source and destination to be different machines on a network

You can think of a pipe as a form of buffered IO similar to file IO but with no file in-between; sockets are more like network communication. In either case, the sender does not have to wait for the recipient, but the recipient blocks if there is nothing to receive. However, because of its similarity to networking, socket communication also allows *unreliable* communication like a datagram on a network [Sechrest 1986].

All of these communication mechanisms are variations of *interprocess communication* (*IPC*). This is not an exhaustive list; microkernels for example sometimes implement IPC with short messages that are passed through the kernel to other processes.

A final consideration with tasks is *synchronisation*. If two or more parts of a program are running simultaneously (either really, on more than one core or CPU, or conceptually, through scheduling each in turn), it is possible that inconsistencies in accessing shared data occur. Think how a variable is implemented in machine code. It may be held for some time in a register and only later written back to memory. If a variable is updated in one task, then inspected or updated by another, if it has not been written back to memory after a change in the first task, the version the second task sees is not up to date and inconsistencies can ensue. For example, if process *A* updates a shared variable `counter` by adding 1, and process *B* also updates it by adding 1, the following sequence can occur, assuming `count = 0` at the start:

Process *A*		**Process *B***		`count`
action	*reg*	*action*	*reg*	in mem
`reg=count`	0	`reg=count`	0	0
`reg++`	1	`reg++`	1	0
`mem=reg`	1	`mem=reg`	1	1

What has happened? Shouldn't the final value of `count` be 2 if both processes incremented it? Writing to memory has to happen sequentially – one process gets in before the other. Since both set `count` to `1`, whichever got in last set the value in memory. Imagine now that Process *A* for some reason runs faster than Process *B* and increments `count` 4 times between where Process *B* copies it to the register and where Process *B* writes it back to memory. What will happen now is that Process *B* will have updated count to `4` the Process *B*, instead of updating it to 5 as you would expect, sets it back to `1`. Here is such a sequence of events (in which "`reg`" is the register used to hold `count`, and "`me`" is where "`count`" is held in memory):

Process *A*		**Process *B***		`count`
action	*reg*	*action*	*reg*	in mem
`reg=count`	`0`	`reg=count`	`0`	`0`
`reg++`	`1`	`reg++`	`1`	`0`
	`1`	`reg++`	`2`	`0`
	`1`	`reg++`	`3`	`0`
	`1`	`reg++`	`4`	`0`
`mem=reg`	`1`	`mem=reg`	`4`	`1`

I show the last step as simultaneous but even if this is true, only one process can update the memory at once, so the last step is really one of:

	`1`	`mem=reg`	`4`	`4`
`mem=reg`	`1`		`4`	`1`

Or:

`mem=reg`	`1`		`4`	`1`
	`1`	`mem=reg`	`4`	`4`

This could arise from a loop like this, where Process *A* has $N = 1$ and Process *B* has $N = 4$, and the compiler keeps `count` in a register for the duration of the loop:

```
count = 0;
// something here forcing count to spill to memory
for (int i = 0; i < N; i++) {
   count ++;  // count in register here
   // do some useful work
}
// compiler spills count to memory only here
```

The two processes either run on a separate CPU so "`reg`" for each is a different piece of hardware, or they are separately scheduled on the same CPU, in which case the kernel has to save and restore the registers at each context switch, so the practical effect is as if “`reg`” is a different physical piece of hardware for each process. At the end, the value in memory could be either `1` or `4`, depending which process completed its memory update last, as illustrated.

A situation like this where the result depends on which process or thread gets there first is called a *race condition* and is usually a programming error. Any piece of code that contains an update to a shared data structure is a *critical section* and updates in a critical section should be protected by one of several forms of *synchronisation*, mechanisms to enforce sequential updates.

Threads and processes assume all the computation is on one machine. Network-based computing allows parts of a workload to be on another computer. Distributed systems go a step further and abstract away the network: is a distributed system uses a resource on a remote machine, that is a performance detail, rather than integral to the design of the system. As far as the user is concerned, it is a single system. Cloud-based services contain aspects of distributed computing as well as aspects of network-based computing.

Following sections expand on these concepts.

The take home message? *A task (thread or process) can get launched in various ways. A Unix-style processes launch combines* `fork` *and* `exec`*. Communicating threads can use their shared address space. Cooperating processes need to be able to communicate outside their own address space. Synchronisation is necessary to protect critical sections to ensure consistency in updating shared data structures. Distributed systems abstract away the network so the physical location of a service becomes an implementation detail rather than being integral to the design.*

6.2 Launching

I start with launching a process the traditional Unix way, since that is relatively simple. You only need understand the `fork` system call. Complications arise with sharing (§6.3) and synchronization (§6.4).

`fork` to launch child processes

The `fork` system call looks very simple. It takes no parameters, and can return one of three kinds of value:

- *success, still in parent process* – returns a positive integer, the process id of the new child process
- *fail, still in parent process* – returns -1
- *success, in child process* – returns 0

Since the child and parent process are (almost) exact copies of each other, you have to use the value returned by `fork` to work out if you are in the parent or child process (after checking it is not negative, indicating failure). It is easy to remember that the child process sees a value of zero as its returned value, as a process ID will in general be a positive integer and it makes sense that the parent knows the identity of any process it creates so it can control it if necessary, whereas a child process should in some cases not be able to do that. For example, if a child process is created by a command shell, it would be strange if the new process could terminate the process that created it.

What could make `fork` fail? There are three possible causes. Creating another process could exceed a system-wide limit on active processes, it could exceed the limit for the current user or the system could be out of swap space (VM backing store). Older Unix versions only had a system-wide limit on active processes but that meant a single user could make the system unusable by writing a problem that carried on calling `fork` in a loop. Any attempt at shutting the program down would fail because launching a new process to kill the program doing all the `fork` calls would fail because running the `kill` command would require starting a new process. Forcing a logout of the user is possible if the user exceeds their limit on active processes but does not exceed the system-wide limit: another user with administration privileges could log in and put a stop to the errant log in session.

Here is a simple example of forking processes:

```
// test effect of fork
// picks up N from command line
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_PROCS     5

void perform_work (int passed_in_value) {
  printf("In process %d\n", passed_in_value);
}
```

```
void print_usage (char *name) {
fprintf(stderr,
    "Usage: %s N\n"
    "where N is number of processes to launch\n"
    "default: %d\n",
name, NUM_PROCS);
exit (1);
}

int main (int argc, char *argv[]) {
  int result_code;
  int N = NUM_PROCS;
  pid_t wpid;
  int status = 0;

  if (argc == 2) {
    char *end;
   N = (int)strtol (argv[1], &end, 10);
   if (*argv[1] == '\0' || *end != '\0') {
     print_usage (argv[0]);
   }
  } else if (argc != 1)
     print_usage (argv[0]);

  // create all processes one by one
  for (int i = 0; i < N; i++) {
    printf("In main: creating process %d\n", i);
    pid_t pid = fork ();
    assert(pid >= 0);
    // child process? do work and get out of loop
    if (!pid) {
      perform_work (i);
      return i; // use exit(i) if not in main
    } else {
      fprintf(stderr,
         "successfully launched process %d, pid=%u\n",
         i, pid);
    }
  }

  // wait for each child to complete
  while ((wpid = wait(&status)) > 0) {
      if (WIFEXITED(status)) {
```

```
      status = WEXITSTATUS(status);
      printf("Exit status of %d was %d (%s)\n",
         (int)wpid, status,
         (status % 2) ? "odd" : "even");
    }
  }
  printf("All child processes completed successfully\n");
  exit(EXIT_SUCCESS);
}
```

Note a couple of tricks in the code. First, the "usage" message is split over several lines for readability. In C, ending a string constant and immediately after starting another treats the two constants as if there was no break. So:

```
"Usage: %s N\n"
"where N is no. processes to launch\n"
"default: %d\n"
```

has the same effect as writing:

```
"Usage: %s N\nwhere N is no. processes to launch\ndefault: %d\n"
```

but the former is easier to read. The line breaks *outside* the double-quotes have no effect – they are just to aid the reader of the code. Without the `'\n'` characters in the string, it would display without line breaks.

Another detail: using `assert`. If the condition passed to `assert` is **true**, it does nothing. If **false**, it terminates the program with a message showing the line and source file of the assertion. You can turn off assertions with the compile option `-DNDEBUG`.

Finally, waiting for each process to terminate is a little complicated. Calling `wait` suspends the process if there are any child processes and returns if any of them exits. On exit, `wait` returns the child process ID, and sets the value pointed at by its parameter, packing two 16-bit numbers into an integer, representing the returned status (as returned by a value passed in to `exit` or a value returned, if the child process completed by a `return` statement in the main program) and the cause, if not a clean exit. Preprocessor macro `WIFEXITED` extracts **true** if it was a clean exit and the macro `WEXITSTATUS` extracts the returned `exit` or `return` value (which only makes sense to do if it was a clean exit). The loop checking for child processes terminates when `wait` returns a process ID of 0, indicating there are no waiting child processes.

Here is an example of running the program on the command line:

```
$ ./forktest 3
In main: creating process 0
successfully launched process 0, pid=50141
In main: creating process 1
successfully launched process 1, pid=50142
In main: creating process 2
In process 0
In process 1
successfully launched process 2, pid=50143
Exit status of 50141 was 0 (even)
Exit status of 50142 was 1 (odd)
In process 2
Exit status of 50143 was 2 (even)
All child processes completed successfully
```

launching a thread

Launching a thread, like `fork`, splits a program into two separately scheduled units of code. For consistency with the way we think of cooperating processes, it is useful to think of the main program as a thread though it is actually launched as a process. The difference between the main program and any other thread is that if the main program quits, it ends the process, so it would also end any threads it launched, even if it returned or did an `exit` system call without explicitly terminating any threads it had launched.

Launching a thread is different from `fork`: it is more like calling a function, though each new thread gets a new stack so it can call and return from functions (or methods in an object-oriented language) independently. I focus here on the Pthreads approach. To launch a thread, you call `pthread_create` and pass in

- *a pointer to store the thread id* – of type `pthread_t*` (an internally-defined type that identifies a thread)
- *thread attributes* – `NULL` if you want the default attributes
- *pointer to start routine* – a function called to launch the thread; if it returns, it has the same effect as calling `pthread_exit`
- *argument to pass in to the start routine* – of type `void*` to allow a pointer to any type of value

If a call to `pthread_create` succeeds, it return 0, otherwise the return value is an error code. To ensure that a thread has completed, the launching thread should usually call `pthread_join`. Calling `pthread_join` results in waiting for the given thread to terminate. A call of `pthread_join` requires:

- `pthread_t*` *value* – to identify the thread being waited for
- `void**` *a pointer to return a value from the thread* – (`NULL` if no value is required)

The type for returning a value is `void**` because it is a pointer to a location that could contain a pointer, which allows an arbitrary type of value to be sent back.

Al these `void` pointers and pointers to pointers require type casts to extract the actual value pointed to, and hence require very careful programming to avoid mistakes – a hazard arising from the fact that C does not have checkable mechanisms for implementing generality as are found in object-oriented languages (classes with inheritance for example).

Here is a small example illustrating how to launch threads, pass in values and wait for threads to complete. If each thread takes about the same amount of time, they will usually run in approximately the same order, so outputs in this case will look as if the threads ran sequentiall.

```
// http://en.wikipedia.org/wiki/POSIX_Threads
// adapted to pick up N from command line
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <assert.h>

#define NUM_THREADS     5

void *perform_work (void *argument) {
  int passed_in_value;

  passed_in_value = *((int *) argument);
  printf("In thread %d!\n", passed_in_value);

  /* optionally: insert more useful stuff here */

  return NULL;
}

void print_usage (char *name) {
fprintf(stderr,
    "Usage: %s N\n"
    "where N is number of threads to launch\n"
    "default: %d\n",
name, NUM_THREADS);
exit (1);
}

int main (int argc, char *argv[]) {
  pthread_t *threads;
  int *thread_args;
  int result_code, index;
  int N = NUM_THREADS;

  if (argc == 2) {
    char *end;
   N = (int)strtol (argv[1], &end, 10);
   if (*argv[1] == '\0' || *end != '\0') {
     print_usage (argv[0]);
   }
  } else if (argc != 1)
     print_usage (argv[0]);
  threads = malloc(sizeof (pthread_t)*N);
  thread_args = malloc(sizeof (int)*N);

  // create all threads one by one
  for (index = 0; index < N; ++index) {
    thread_args[index] = index;
    printf("In main: creating thread %d\n", index);
    result_code = pthread_create(&threads[index], NULL,
           perform_work, (void *) &thread_args[index]);
    assert(0 == result_code);
  }

  // wait for each thread to complete
  for (index = 0; index < N; ++index) {
    // block until thread 'index' completes
    result_code = pthread_join(threads[index], NULL);
    printf("In main: thread %d has completed\n", index);
    assert(0 == result_code);
  }

  printf("In main: All threads completed successfully\n");
  exit(EXIT_SUCCESS);
}
```

Here is an example of running the program on the command line:

```
$ ./pthreads 3
In main: creating thread 0
In main: creating thread 1
In main: creating thread 2
In thread 0!
In thread 1!
In thread 2!
In main: thread 0 has completed
In main: thread 1 has completed
In main: thread 2 has completed
In main: All threads completed successfully
```

6.3 Sharing and Communication

Chapter 5 outlines two approaches to sharing in a processes: shared memory and memory-mapped files (see page 91). I revisit these concepts briefly here to illustrate how they can be used between cooperating processes. I also briefly examine other interprocess communication mechanisms. IPC in general can be a complicated subject; I focus here on the use of pipes, since that is a mechanism you should be familiar with from using the UNIX command line.

Threads do not need these concepts since threads in the same process share an address space and therefore can use global variables or pointers to shared data with no complication – except preventing race conditions, as explored in §6.4.

shared memory

Creating a shared segment takes some setting up, and it is also necessary to understand how it is named so it can be found by any child process that needs it. The simplest approach is to do all the set up before forking child processes: that way they will have all the necessary information in their new copy of the parent's address space. Here is a very simple example that creates a shared segment and if a word is supplied on the command line, writes it to the segment then detaches from it. If the word on the command line is `"delete"`, the program removes the shared segment. If the program runs without a word on the command line, it prints whatever is in the shared segment.

```
// adapted from http://stackoverflow.com/questions/5656530/
//                       how-to-use-shared-memory-with-linux-in-c
// adaptations:
//    use the executable name from command line to make the key
//    if "delete" is all that's in shared memory, remove the segment
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */
#define USAGE    "usage: %s [data]\n" \
                 "       data given: write to shared segment\n" \
                 "       data not given: write out shared segment\n" \
                 "       data=delete removes shared segment\n"


int main (int argc, char *argv[]) {
    key_t key;
    int shmid;
    char *data;
    int mode;

    if (argc > 2) {
        fprintf(stderr, USAGE, argv[0]);
        exit(1);
    }
    // make the key:
    // file must exist: safe to use executable from command line
    if ((key = ftok(argv[0], 'R')) == -1) {
        perror("ftok");
        exit(1);
    }
    //  create the segment:
    if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(1);
    }
    // attach to the segment to get a pointer to it:
    data = shmat(shmid, (void *)0, 0);
    // returns -1 for error: assumes this is never a valid pointer
    // so we can cast away the pointer and treat bit pattern as int
    if (data == (char *)(-1)) {
        perror("shmat");
        exit(1);
    }
    // read or modify the segment, based on the command line:
    if (argc == 2) {
        printf("writing to segment: \"%s\"\n", argv[1]);
        strncpy(data, argv[1], SHM_SIZE);
    } else
        printf("segment contains: \"%s\"\n", data);

    // if the entire contents is "delete", remove the segment
    if (strcmp (data, "delete") == 0) {
       if (shmctl (shmid, IPC_RMID, NULL) == -1)
           perror("shmctl");
           exit(1);
    } else // detach from the segment:
    if (shmdt(data) == -1) {
        perror("shmdt");
        exit(1);
    }
    return 0;
}
```

A few things to note about the example. First, to create a multiline string represented by a preprocessor symbol (USAGE), though C permits splitting a string into segments surrounded by double-quotes that can be separated by any whitespace including a line break, a preprocessor symbol has to be defined all on one line. Putting a "\" character just before the line break makes the preprocessor treat the line break as part of the text the symbol represents, rather than ending the definition at the line break.

In all the system calls, a return value of -1 signals an error. If the value returned is supposed to be pointed, the presumption is that the integer representation of -1 will never be a valid pointer, and the pointer can be cast to an int type to check if it matches the bit pattern for -1 (or the value -1 can be cast to a pointer type to compare with the returned value[1].

Next, look at ftok. This uses a file name that has to exist to create a unique **key** that can be used to identify a shared segment. In the example

```
ftok(argv[0], 'R')
```

[1]In a 2's complement world this is not such a bad assumption. The machine representation of -1 in 2's complement is a word with all 1s. Even in a machine that allowed the full range of machine addresses, none of these calls can reasonably be expected to return a pointer to the very last byte in the address space.

the first parameter is the file name. It can be any file that exists on the system, but it is a safe bet that the executable name as typed on the command line exists, which is why I use `argv[0]`. If you really are being cautious about not recycling the same file name in more than one `ftok` call, using `argv[0]` is not completely safe as you could run the same executable name from a different directory. The `'R'` is an extra identifier (it can be any `int` value) that can be used to alter the key if the same file name is used again.

Having obtained a key, we can use it to create (or do nothing except check our permissions to access it, if it already exists) a shared segment using `shmget`. In the call, we use the key, specify how many bytes we want and create flags combining values using a bitwise **or**. The first part, `0644`, specifies permissions in octal (a number starting with a zero is reads as octal, or base 8, in C). Why octal? Three bits are used to specify permissions: read, write, execute. A `6` has the read and write bits set (`110` in base 2), whereas a `4` only has read permission set (`100` in base 2). So this permissions string allows the current user (the first three bits) read and write permission, and the rest of the user's group (next three bits) and others (last three bits) only read permission. The result of the system call is an id that can be used to attach to the shared segment, the final step before we can use it.

The `shmat` call puts the shared segment into the current address space and returns a pointer to it so we can access it. Calling `shmat` is much like calling `malloc`: you get a pointer to a chunk of data. There are a few other details to how you can use `shmat`; this should be enough to give you a sense how things work.

By this stage we have a shared segment, but we are still in the same process. If you called `fork` at this point, you would have two processes that each could talk to each other through the same shared segment. The sample program keeps things simple. It either puts something in the shared segment or reports what was there before, based on the command line. If the command line contained the word `"delete"`, the program calls `shmctl`, a complicated system call that can do many different things but in this example deletes the shared segment. If you do not choose to remove the shared segment, the program detaches from it. Once the program exits, the shared segment is still there and can be found on the next run if you calculate the key correctly and use the same ID.

memory-mapped file

Here is a simple example of using a memory-mapped file, illustrating the essentials of the `mmap` system call. First, the parameters passed into `mmap` are:

- `void *addr` – start address of the new memory map; complicated to use and easiest left at 0 (`NULL`)
- `size_t len` – how big the region should be (rounded up to a whole number of pages; any extra length will be zero-filled, if mapped from a file)
- `int prot` – specified by the bitwise **or** of any of:
  - `PROT_NONE` – no access permitted
  - `PROT_READ` – read access permitted
  - `PROT_WRITE` – write access permitted
  - `PROT_EXEC` – execution permitted
- `int flags` – various including `MAP_ANONYMOUS` meaning no file associated with the mapping; the offset is ignored in that case
- `int fd` – file descriptor of an open file or `-1` if not based on a file
- `off_t offset` – useful if the memory map is based on actual file: the offset from the start of the file (should be a whole multiple of page size)

It returns a value of type `void *`, the start address of the newly mapped region.

Heads up: `mmap`*, because it interacts with both the VM system and file buffering, can have significant variations across systems. Check your system documentation starting from the* `man` *page for details that may differ from this description.*

Now the example:

```
// simple demo of memory mapping
// stackoverflow.com/questions/
//     13274786/how-to-share-memory-between-process-fork
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>

static int *glob_var;
```

```
int main(void)
{
    glob_var = mmap(NULL, sizeof *glob_var, PROT_READ | PROT_WRITE,
                    MAP_SHARED | MAP_ANON, -1, 0);

    *glob_var = 1;

    if (fork() == 0) {
        *glob_var = 5;
        exit(EXIT_SUCCESS);
    } else {
        wait(NULL);
        printf("%d\n", *glob_var);
        munmap(glob_var, sizeof *glob_var);
    }
    return 0;
}
```

A few things to take note of: the example uses an anonymous map, i.e., there is no file specified. So after `munmap`, there is no version of the data saved in a file. Note that flags are specified as "`MAP_SHARED|MAP_ANON`"[2]. An alternative to a shared map (changes in any process are reflected in any other sharing the map) is a private map, which uses copy on write.

There are many variations on how to set up a memory map; this example is sufficient to do the equivalent of a shared segment. The setup is a little simpler, and `mmap` has the advantage of allowing significantly more variations, such as attaching the map to a file so it still exists when the process terminates. If you use a significantly more complex call to `mmap`, you run into potential for system-dependencies. That can be seen as a problem, or an advantage, depending how you look at it. More complex users of `mmap` carry the risk of not being portable, but there is the potential to use large amounts of memory efficiently in a way sympathetic to the overall VM and file buffering system.

The take home message? *In its most basic form, memory mapping provides exactly the same service as a shared segment with simpler set up. Memory mapping can do a lot more, at the expense of variations across systems. A shared segment is works consistently on all Unix-derived systems. However if you keep use of* `mmap` *simple and avoid non portable details, many prefer it as the more modern approach.*

[2]The proper spelling of the second flag is `MAP_ANONYMOUS` but Mac OS X only supports the shorter spelling and Linux accepts both.

pipes

Communicating with pipes is very like reading and writing files. You open a `pipe` using a 2-element `int` array to represent file descriptors for respectively the input and output ends of the pipe. In a simple example, you do this before calling `fork`, which means the two sets of file descriptors are exact copies of each other. After calling `fork`, you close the file descriptor that does not apply (either the read or write end of the pipe, depending whether the parent or child process is doing the reading or writing). Then the writing process can write whatever it likes using any file output operation to the output "file". The other process reads until it detects end of file. The following example is straight from the Linux `man` page for `pipe`:

```
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;
    if (argc != 2) {
    fprintf(stderr, "Usage: %s <string>\n", argv[0]);
    exit(EXIT_FAILURE);
    }
    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */
        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);
    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

A detail to note in this example: using `_exit` rather than the usual `exit` system call is a little safer in a child process: this new variant on `exit` omits a few IO-related details like flushing buffers, which may interfere with the parent process.

Use of pipes is relatively simple in the case where you are launching the child process after creating the pipe. It is also possible to create a pipe using a file-like construct called a *FIFO* (for first-in-first-out), also called a *named pipe*. A FIFO is created using the `mkfifo` system call, and is given a name within the file system. It looks like a regular file, but has to be opened for both reading and writing before any IO operations take effect. If a process opens a FIFO for reading or writing and attempts either operation, the process blocks until another process opens the other end of the FIFO for the opposite operation.

6.4 Synchronization

All synchronisation starts from the problem of a race condition. Any solution must solve the problem of inconsistent updates in its own implementation, otherwise we are back where we started. I outline the simplest approach to start, then show how it can become part of more sophisticated approaches. The Pthreads library calls can be set up to work across cooperating processes as well as threads. If these operations are to work across separate processes, the data structures have to be in shared memory, and the constructs may need to be initialised with a specific status bit to indicate they are in shared memory between separate processes.

The simplest way to ensure consistency is to allow only one process or thread at a time to enter a critical section. The simplest mechanism for ensuring this *mutual exclusivity* is a *lock* and the simplest kind of lock is a *spinlock*. Any lock requires an identity (otherwise you would be restricted to a single lock for the entire set of cooperating threads or processes) and a way of recording whether it is *set* or *unset* – the *state* of the lock. The simplest way of encoding both the lock's identity and its state is to use a single variable that can be `1` for set or `0` for unset.

The lock's identity then is the specific variable and the state is its value.

Next simplest is a *mutex*, which also only allows one task into a critical section, but uses a queue to order requests to prevent starvation (ensure fairness) and puts waiting tasks to sleep so they do not waste CPU time or cause unnecessary contention for memory when a lock releases.

A *semaphore* generalizes a mutex to allow up to some number N tasks past a certain point. A semaphore has a counter and a queue. If the count is above zero, a task can proceed and lower the count. If not, it is queued. A semaphore with $N = 1$ is much like a mutex except a mutex can *only* be unlocked but the task that locked it, whereas any task can increase the count on a semaphore. Semaphores can be tricky to program, and are not widely used in practical code.

Finally, a *barrier* forces a fixed number (call it N again) of tasks to wait when they reach the barrier until the last of that number of tasks reaches the barrier, then they are all woken up. A barrier also is usually implemented with a queue. A barrier is a useful construct for workloads where each task can work independently then must pause until all other tasks reach a particular stage of computation, usually to swap data and go on to the next parallel stage of computation.

In what follows, I use task and thread interchangeably, since most Pthreads primitives can be configured to work on processes as well as on threads.

spinlock detail

A lock has three basic operations (with more complications available in some implementations, like testing the lock without stalling if it is set):

- *initialise* – usually to unset (`0`)
- *lock* – set the lock (to `1`), and wait for if it was already set; also called *acquiring* the lock
- *unlock* – unset the lock (to `0`); also called *freeing* the lock

Setting (acquiring) the lock requires a way to test if it is already set and only actually complete the setting operation if this task is the one that actually acquired the lock. The basic mechanism of a spinlock is:

```
while (lock == 1) ; // spin
lock = 1;
```

and unlocking is dead simple:

```
lock = 0;
```

The empty loop keeps checking the lock variable until it becomes `0` then grabs it by setting it to 1. This has an appealing simplicity: we can represent the lock as an `int` variable, and the code is easy to understand. However, there are a few problems with this as it stands. First, if a compiler puts the value of `lock` in a register, the loop will never terminate because the code generated will keep checking the copy in a register, not the value in memory that another task could update. We can fix that in C by making the variable *`volatile`*:

```
volatile int lock = 0;
```

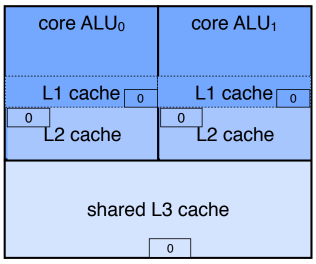
That however does not solve the race condition between exiting the loop and setting the lock variable. For that we need an *atomic memory operation*, something that lets us exit the loop only if we simultaneously manage to set the lock variable. There are various primitives that can be used to implement this:

- *test and set* – write to memory and return the previous value in one step
- *load-link store conditional* – (`ll-sc`) a pair of instructions: `load-link` (`ll`) reads a memory location and `store-conditional` `sc` only succeeds if the memory location has not been written since the previous `load-link`.
- *compare and swap* – swap a register value with memory contents only if the two values are the same and report success if the swap happened

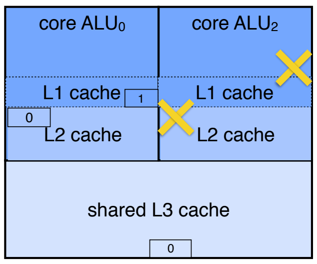
Of these, test and set is the least general because it requires that all contending lock setters use the same value for set and unset; using `ll-sc`, it is possible to use a different value identifying each holder of the lock if necessary. We will not consider complications arising out of these variations and just assume we can do a machine code equivalent of a spinlock loop that works properly. In the MIPS instruction set, if an `ll` instruction followed by a `sc` succeeds, the register used to store a value to memory is overwritten with , otherwise `0`. A failure in MIPS can occur if there is an interrupt between the pair of instructions; success if only guaranteed if they follow one after the other. From here on, I use two primitives:

- *`lock(var)`* – use `var` to represent the lock value and spin until it can be set
- *`unlock(var)`* – unset the lock held via variable `var`

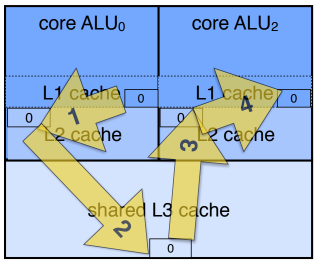
Initialising the lock is trivial: `var` is set to `0`. So now we can protect a critical section, provided all tasks competing for it can access the same lock variable:



(a) Unlocked



(b) Locked in ALU$_0$



(c) After unlock

Figure 6.1: Spinlock and caches. Whenever a lock value is updated, it must be *invalidated* from other caches. When a CPU that no longer has a valid copy reads its value, it has to be updated through the cache hierarchy.

```
lock(var);
  // update shared data
unlock(var);
```

That looks pretty simple, but a spinlock has a number of disadvantages, so other synchronisation mechanisms are more common in real code.

First, a spinlock does not guarantee *fairness*: since every acquisition of the lock is a race it is possible, if there is heavy contention for a lock, that one task gets starved.

Second, a spinlock is a huge bottleneck on the memory system. In a multiprocessor system, caches have to be kept consistent, so no cache may modify a value that is shared in another cache. If two or more CPUs share access to the same memory location (as with a shared segment, threads in one address space or a shared mapped region), they can only have a duplicate if no CPU is trying to modify it. Any memory write has to have exclusive access: the cache block containing the write location has to be *invalidated* from all other caches. Any further attempt at accessing that data will incur a *cache miss* in the non-writing CPUs. The CPU that did the write has to *write back* the modified data. In multicore systems, it is common for all cores to share a cache and any write backs may have to go through that (usually lowest-level) shared cache. If there is no shared cache, write backs have to go through RAM, which is even slower.

Figure 6.1 illustrates how locking an unlocking a spinlock generates memory traffic. In 6.1a, both CPUs (labelled as ALUs here because the L1 and L2 caches are part of the CPU chip) see the value in its unlocked state and neither is trying to modify it. In 6.1b, ALU_0 has just locked the spinlock and that forces an *invalidation*: it is flushed out of any other cache so no other CPU sees the value in an inconsistent state. In 6.1c, ALU_0 has changed the value back to `0` and ALU_1 has simultaneously tried to read it. ALU_1 has a cache miss and the cache mechanism forces ALU_0 to write it back to the shared L3 cache via its own L2 cache so ALU_1 can handle the cache miss correctly. Finally, ALU_1 has the updated value in its own L1 cache via its L2 cache. In a situation of multiple tasks on multiple CPUs all contending for the same lock, a flurry of invalidations and misses will ensue.

Pthreads includes a spinlock (invoked by `pthread_spin_lock`, using a data structure of type `pthread_spinlock_t*` to hold the lock state); we will not explore its use in depth. A spinlock, while inefficient, can be used as a building block for better primitives, because it is reasonable to use for a lock that is held for a very short time, minimising the chances of contention for the lock. Other primitives that are in general more efficient have more overhead to set up.

The take home message? *Spinlocks are quick and efficient if held for a short time when overheads of a more sophisticated primitive are not worthwhile, and are also used as building blocks for other more complex primitives. The do not ensure fairness and the memory system can become a major bottleneck if multiple tasks contend for the same lock.*

mutexes

A *mutex* (for *mutual exclusion*) is a more sophisticated form of lock. Like a spinlock, it only allows one contending task in at a time. Unlike a spinlock, it puts waiting tasks on a queue, with two benefits:

- *fairness* – if waiting tasks are processed in the order they are enqueued, no task will wait indefinitely
- *efficiency* – unlike a spinlock, a waiting task is put to sleep, meaning its CPU can be used for other things, and the unlock operation wakes up the

next task on the queue, avoiding a flurry of contention for the lock variable

Drawbacks include:

- *overheads* – setting up the queue data structure and enqueueing are more work than testing a spinlock a few times
- *overheads of sleep and wake* – a task put to sleep may lose its working set in the caches or worse still in main memory so waking from sleep can be expensive even where all tasks on the same CPU are threads in the same process, avoiding the need for a heavyweight context switch

Overall, unless you are sure a lock will be held a very short time, a mutex is better than a spinlock. Since updating the data structures needed for a mutex require a lock to be held for a very short time, it is reasonable to use a spinlock to protect updating the internal data structures of a mutex.

Pthreads implements mutexes. It has a data type to store state of a particular lock, `pthread_mutex_init`, usually used through a pointer. The major calls are:

- *`pthread_mutex_init`* – pass in a `pthread_mutex_init*` pointer as well as attributes which can be `NULL`
- *`pthread_mutex_lock`* – pass in a `pthread_mutex_init*` pointer: suspend the thread if the lock is already held, otherwise acquire the lock and continue
- *`pthread_mutex_trylock`* – like `pthread_mutex_lock` except the returned value indicates whether the lock was successfully acquired or not, without suspending the thread
- *`pthread_mutex_unlock`* – unlock the mutex, allowing the next queued suspended thread (if any) to continue

The take home message? *A mutex is the best approach to implementing a simple critical section in the general case, though a spinlock may be better if the lock is held a very short time. Spinlocks are used to protect the basic operations used to implement a mutex, which is acceptable since each basic operation is very quick, usually less than 10 machine instructions.*

barriers

The typical use of a barrier is to intersperse periods of parallel computation with pauses to exchange data. Often, barriers are paired, so the data exchange phase can be completed before the next parallel computation phase. Here is a typical outline of a parallel algorithm that uses barriers (assume each task looks like this):

```
  while (MoreWork) {
    // parallel computation not accessing shared data
    Barrier(P); // wait for all N threads
    // update shared data
    Barrier(S); // wait for all N threads
}
```

The shared data-updating phase may use mutexes to ensure consistency. Code written in this style will usually avoid doing synchronization in the parallel phase.

Heads up: *Barriers are an optional part of the POSIX standard so be prepared for the possibility that you may have to implement your own, or find an implementation that is not part of your existing POSIX library.*

Another thing that may be done between two barriers is rebalancing workload. If some tasks finish a lot faster than others, a bigger share of work could be allocated to them. Load balancing in general is tricky and could be the subject of a whole book on its own.

Pthreads also provides a barrier construct. The major calls are:

- *pthread_barrier_init* – pass in a `pthread_barrier_t*` pointer as well as attributes which can be `NULL`, and the number of tasks (*N*) the barrier needs to see before it allows all to restart
- *pthread_barrier_wait* – pass in a `pthread_barrier_t*` pointer: suspend the task unless it is the *N*th to reach the barrier, otherwise restart all tasks waiting on the barrier

The take home message? *A barrier is a relatively easy synchronisation primitive to use. As long as you get the basic concept and can configure your parallel code to split into purely parallel sequences split by pauses to exchange data, you can simplify algorithm design over most other approaches.*

language-based versus library-based synchronization

Some programming languages (e.g., Java) have synchronization primitives built into the language. In C and many other languages, synchronization is provided by library calls. There are advantages and disadvantages of both approaches.

For features built into the language, like `synchronized` methods in Java, advantages are:

- *automatic locking and unlocking* – the programmer, once a method or block of code is declared as `synchronized`, need not remember to lock and unlock
- *improvements affect all code* – any improvements, particularly if they are in the runtime environment or dynamically linked libraries and do not need a rebuild, improve all code without programmer effort
- *standard approach* – programmers can see where synchronization is used by looking for standard language constructs
- *efficiency* – since the approach is built into the language, significant effort can be invested in the compiler to make it work well with other aspects of the code like memory allocation that can affect performance

Downsides of features built into the language include:

- *inflexibility* – if a fundamentally new approach to synchronization is developed, it cannot be used unless the language-based approach fits it or is abandoned
- *over-use* – a method or code block declared `synchronized` (or equivalent in languages other than Java) still requires the overheads of synchronisation even when used in a context where this is not needed

For a library-based approach, some advantages are:

- *flexibility* – fundamentally new approaches can easily be adopted
- *focused use* – synchronisation need not be used where it is not required
- *language portability* – widely-used standards like Pthreads can apply across languages (e.g., C and C++), making it relatively easy to change the implementation language

Drawbacks of a library-based approach include:

- *programmer error* – for example, forgetting to unlock a lock is a more likely error with a library-based approach than a feature built into a language
- *possible interactions with other language features* – e.g., the way memory is used can make a big difference to performance and if the compiler is unaware that data is used to implement a synchronization primitive, inefficiencies can result
- *rebuild to see improvements* – although improvements in a dynamic library can be seen without a rebuild, other changes may require a recompile to see an improvement

In C++, a neat approach to the problem of forgetting to unlock is to use class constructors and destructors. If you have a `Lock` class that represents the fact that you want a lock, the trick is to put the lock operation into the constructor for `Lock` and the unlock operation into the destructor. You need an object representing whether the lock is held or not. For purpose of example, assume that class is called `Lock_data`. Then you can code a critical section as follows, if variable `lockstate` of class `Lockdata` represents the lock you want to hold here:

```
{  // open a block for local variables
   Lock lockvar(lockstate);
   // do stuff that needs to be protected
}  // destructor unlocks here
```

This approach relies on the fact that in C++, when an object is defined, its constructor is invoked before any following code and, when it is about to go out of scope, its destructor is invoked. Placing "`{ }`" around the critical section containing the `Lock` variable ensures that the variable's destructor is invoked at the closing "`}`". Any other piece of code that does the same thing using the same `Lock_data` variable will be controlled by the same lock[3].

The take home message? *Language-based primitives are less common than library-based approaches, in part because there is no consensus on the best approach to parallel coding. Even if you have language-based primitives, it is useful to understand what is happening underneath them.*

[3]In general, you can create constructs in C++ that use a constructor-destructor pair like this. This coding idiom is called **r**esource **a**cquisition is **i**nitialization (RAII). RAII works correctly if you `break` or `return` from the code block.

6.5 Distributed Systems and the Cloud

The *cloud* has in recent years become the big new thing. But is it so new? I start by explaining the general concept of distributed computing then relate it to what is now called the cloud.

Networked services are characterised by being named by their location. The name `www.google.com` identifies a particular web site at a particular location. By magic of the domain name system, Google (and other big service providers) convert this single address to multiple servers to spread the load. But you are nonetheless presented with the service as if you have to know where it is located.

Distributed computing abstracts away location. A service is named so it can be identified but whether it is on your computer, implemented on a server in your building, a local cluster of computers or spread out over multiple remote computers is not explicit in the name. So *distributed systems* are distinguished from *networked services* by *location-independent naming* and the fact that whether a service is local or remote is an implementation detail, not implicit or explicit in the name of the service.

In networked computing, there are abstractions that simplify communication over a network. *Remote procedure* call (*RPC*) for example wraps an API targeting remote services in something that looks like a function or method call. Java includes support for RPC in the form of Java Remote Method Invocation (RMI) [Waldo 1998]. There are newer approaches like REST (REpresentational State Transfer) [Fielding 2000, Chapter 5], which is suited to web services, and ways of invoking remote functionality will continue to evolve.

Distributed computing attempts to disguise the remote nature of services and resources as far as possible. Some mechanisms to support distributed computing:

- *location-independent naming* – names of services are not related to where they are physically located so whether they are local, on a local network or remote is an implementation detail that can change

- *disconnected operation* – when the network breaks a distributed service may have a fallback option to keep working without access to networked resources

- *replication* – to promote scalabillity, devices or services may be replicated, ideally transparently to the user, who accesses them via a single name

Are cloud-based services network services or distributed computing? Most have attributes of both.

Consider Google Drive, for example. You can access it through a web interface, which makes it look like a networked service. You can mount a Google Drive on a regular computer system and it functions just like part of the file system, except it is accessible in more than one place. It interacts with Google apps in a way that does not always make clear whether it is transporting data over the network or not. Underlying all Google services is a distributed file system [Ghemawat et al. 2003]. Yet many of those services look to the user like networked services – accessed through a web page.

In the mobile app space, these things look more like true distributed systems in that a specialist app often does away (at least at the user interface level with the appearance of domain names, URLs, and the like.

Peer-to-peer (P2P) apps in some ways have attributes of distributed systems in that they have little or no central control, though they often retain network-based interfaces and users are very much aware of what is happening over the network and what is not. P2P traffic introduces new challenges for network operators in that much of it is local and does not reach the higher-volume parts of the network where the big operators do traffic monitoring and modelling [Otto et al. 2011].

The interaction of cloud and P2P promises interesting new developments in systems design. BitTorent Sync for example has an API to build services on top of P2P file sharing, which could solve one of the harder problems of large-scale distributed systems: scalability [Farina et al. 2014; Machanick and Hunt 2014].

The take home message? *In a true distributed system a logical name is location-independent and whether an object, service, etc. is local or remote is a implementation detail. True distributed systems have not found their way out of the research lab; P2P and cloud systems have some of the benefits and properties of a distributed system, even if they are compromises on the pure idea.*

6.6 Parallel Programming Hazards

In general parallel programming is harder than sequential programming. In a edition to the usual kinds of bugs, you have to worry about race conditions, correct use of synchronisation primitives and the challenge of repeatable testing when the exact ordering of events between tasks can vary, since the scheduler is not guaranteed to schedule each task in exactly the same order with exactly the same timing between interrupts on different runs.

One of the issues that can arise is a *deadlock*. A deadlock occurs when two or more tasks block each other on synchronisation. The following is one of the simpler ways this could happen:

Task 0	**Task 1**
`lock A`	`lock B`
`lock B`	`lock A`
`wait for B`	`Wait for A`

As shown, the locks are simultaneous but this is not necessary: as long as both tasks hit the second lock after the other task has locked it, neither can continue.

Avoiding deadlocks is a problem in parallel code design. Deadlocks seldom arise in a situation as simple as this where it is obvious that there is a problem. If Task 0 managed to lock both locks before Task 1 tried to lock the first time, there would be no problem. A program could run many times before the deadlock manifested. Minimising the use of locks is a good design principle, particularly also avoiding the need to hold more than one lock at a time. Using higher-level constructs like barriers can also help. In general, a defensive approach of avoiding the kind of situation that can lead to hard-to-discover bugs is the best strategy.

There is a significant amount of theory about deadlock avoidance and detection; in practice, implementing algorithms that do any of this as part of the operating system is impractical because there are so many different ways deadlocks can arise – not only as a result of locks, but also holding resources that only one task can hold, like open files. Deadlock avoidance is usually seen as a programming problem rather than as something to design into the OS. When you write your own parallel code, keeping things simple is a good strategy, as deadlocks cannot occur if no process ever holds more than one exclusive resource (lock, open file, etc.). The closer you get to this ideal, the less likely you are to create a deadlock in your code.

In distributed systems, deadlock can be difficult to handle and there, some of the theory of deadlock detection such as constructing wait-for graphs, is useful [Mitchell and Merritt 1984; Raynal 2013].

A more subtle problem is called *livelock*: two or more tasks, while changing state rather than waiting, cannot make progress because they need something from each other. This can arise if there is some attempt at breaking deadlocks, e.g., releasing a lock if there is no progress and trying again. Another example is if each task has to tell the other it has passed a certain point in a computation, and it cannot do so unless the other has passed that point. Each task tells the other: "sorry, not there yet" and makes no progress.

Parallel programming is a large area, with new ideas arising out making best use of warehouse-scale computing [Barroso et al. 2013], used to implement massively scalable services like Google search, Facebook and Amazon's Elastic Cloud Computing. This is an exciting subject and well worth studying further in your future work.

The take home message? *Parallel programming in general is hard – deadlocks are just one problem you can run into. Using high-level abstractions and well-tested methods is a good starting point.*

Exercises

1. For the thread launch code on page 108, work out how to add a delay that is larger, the lower the number passed into the thread as a parameter. See if you can make the threads
   (a) Produce output in the thread in a different order
   (b) Finish in a different order.

   If you fail in either case, explain why.

2. In the barrier outline of page 123, why do I use two different barriers? What could go wrong if I used the same barrier variable in both places?

3. Look up the detail of each of the following Pthreads primitives and expand on the basic calls I outline. Explain what else you can do with each primitive.
   (a) spinlock
   (b) mutex
   (c) sempahore
   (d) barrier

4. Expand figure 6.1 to 4 CPUs, each contending for the same lock when it is released. Draw the sequence of cache operations as each processor tries to modify the lock variable. Each processor will try to get exclusive access,

invalidating it from other caches. Whichever wins will modify the variable then have it invalidated from its local cache. Use this example to discuss why a spinlock can be inefficient.

5. Explain the advantages and disadvantages of memory mapping versus shared segments. In a simple case, which would you use?

6. Explain advantages and disadvantages of using `fork` versus threads, and give examples where each is the better approach.

7. Explain how COW aids in implementing Unix-style process launching.

8. Another atomic memory operation is *atomic swap*: contents of a register are swapped with memory contents in a single indivisible operation. Sketch out a spinlock implementation using atomic swap.

9. Create a pipe for each of the following scenarios:

   (a) you launch a child process that processes outputs from a parent process

   (b) you launch a new process that has to write values that a separately launched process will read

10. Two processes reach a barrier. Each process can only continue to the next step if the other reports it has completed a specific computation. If the two processes both have not done the required computation at the barrier, could this lead to livelock or deadlock? Explain.

11. Is email implemented as a distributed system or a networked service? Explain.

12. Is Dropbox a distributed system or a networked service? Explain.

References

Aas, J. (2005). Understanding the Linux 2.6. 8.1 CPU scheduler. Technical report, Silicon Graphics, Inc. `http://joshaas.net/linux/linux_cpu_scheduler.pdf`.

Barroso, L. A., Clidaras, J., and Hölzle, U. (2013). *The datacenter as a computer: An introduction to the design of warehouse-scale machines*. Synthesis lectures on computer architecture. Morgan & Claypool Publishers, 2nd edition. `http://www.morganclaypool.com/doi/abs/10.2200/S00516ED2V01Y201306CAC024`.

Bovet, D. P. and Cesati, M. (2005). *Understanding the Linux Kernel*. O'Reilly, Sebastopol, CA, 3rd edition.

Carr, R. W. and Hennessy, J. L. (1981). WSCLOCK – a simple and effective algorithm for virtual memory management. In *Proc, Eighth ACM Symp. on Operating Systems Principles*, SOSP '81, pages 87–95, New York, NY, USA. ACM.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to algorithms*. MIT Press, Cambridge, MA, 3rd edition.

Denning, P. J. (1968). The working set model for program behavior. *Commun. ACM*, 11(5):323–333.

Farina, J., Scanlon, M., and Kechadi, M.-T. (2014). Bittorrent sync: First impressions and digital forensic implications. *Digital Investigation*, 11:S77–S86.

Feitelson, D. G. and Jettee, M. A. (1997). Improved utilization and responsiveness with gang scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 238–261. Springer.

Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*. PhD thesis, University of California, Irvine.

Ghemawat, S., Gobioff, H., and Leung, S.-T. (2003). The Google file system. In *Proc. 19th ACM Symp. on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA. ACM.

Härtig, H. and Roitzsch, M. (2006). Ten years of research on L4-based real-time systems. In *Proc. 8th Real-Time Linux Workshop*.

Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. (2006). MINIX 3: A highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev.*, 40(3):80–89.

Jatho III, E. W. (2014). *A survey of distributed capability file systems and their application to cloud environments*. PhD thesis, Naval Postgraduate School, Monterey, CA.

Kernighan, B. W. and Ritchie, D. M. (1988). *The C programming language*. Prentice Hall, Englewood Cliffs, NJ.

Klein, G., Derrin, P., and Elphinstone, K. (2009). Experience report: SeL4: Formally verifying a high-performance microkernel. In *Proc. 14th ACM SIGPLAN Int. Conf. on Functional Programming*, ICFP '09, pages 91–96.

Lavington, S. H. (1978). The Manchester Mark I and Atlas: a historical perspective. *Commun. ACM*, 21(1):4–12.

Liedtke, J. (1993). Improving IPC by kernel design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188.

Machanick, P. and Hunt, K. (2014). Preliminary thoughts on services without servers. In *Proc. SATNAC 2014*, pages 469–470, Port Elizabeth.

Mayer, A. J. W. (1982). The architecture of the Burroughs B5000: 20 years later and still ahead of the times? *SIGARCH Comput. Archit. News*, 10(4):3–10.

Mitchell, D. P. and Merritt, M. J. (1984). A distributed algorithm for deadlock detection and resolution. In *Proc. 3rd annual ACM Symp. on Principles of distributed computing*, pages 282–284. ACM.

Molnar, I. (2007). Modular scheduler core and completely fair scheduler [CFS]. `http://lwn.net/Articles/230501/`. Linux-Kernel mailing list; Accessed: 5 June 2015.

Moschakis, I. A. and Karatza, H. D. (2012). Evaluation of gang scheduling performance and cost in a cloud computing system. *The J. of Supercomputing*, 59(2):975–992.

Mueller, F. (1993). A library implementation of POSIX Threads under UNIX. In *Winter USENIX*, pages 29–42.

Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T., and Brown, R. (1993). Design tradeoffs for software-managed TLBs. In *Proc. 20th Int. Symp. on Computer Architecture (ISCA '93)*, pages 27–38, San Diego, CA.

Neiger, G., Santoni, A., Leung, F., Rodgers, D., and Uhlig, R. (2006). Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology J.*, 10(3).

Open Kernel Labs (2012). Open Kernel Labs software surpasses milestone of 1.5 billion mobile device shipments. `http://www.ok-labs.com/releases/release/ok-labs-software-surpasses-milestone-of-1.5-billion-mobile-device-shipments`.

Otto, J. S., Sánchez, M. A., Choffnes, D. R., Bustamante, F. E., and Siganos, G. (2011). On blind mice and the elephant: Understanding the network impact of a large distributed system. In *Proc ACM SIGCOMM 2011 Conf.*, pages 110–121.

Pabla, C. S. (2009). Completely fair scheduler. *Linux J.*, 2009(184):4. `http://www.linuxjournal.com/article/10267`.

Patterson, D. A., Gibson, G., and Katz, R. H. (1988). A case for redundant arrays of inexpensive disks (RAID). In *Proc. 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116.

Pietrek, M. (1992). Inside the Windows scheduler. *Dr. Dobb's J.*, 17(8):64–71.

Raynal, M. (2013). Distributed deadlock detection. In *Distributed Algorithms for Message-Passing Systems*, pages 401–423. Springer.

Richards, M. (1969). BCPL: A tool for compiler writing and system programming. In *Proc. Spring Joint Computer Conf.*, pages 557–566.

Ritchie, D. M., Johnson, S., Lesk, M., and Kernighan, B. (1978). The C programming language. *Bell Sys. Tech. J*, 57:1991–2019. `http://www3.alcatel-lucent.com/bstj/vol57-1978/articles/bstj57-6-1991.pdf`.

Russinovich, M. (1997). Inside the Windows NT scheduler, part 1. *Windows IT Pro Magazine*. `http://windowsitpro.com/systems-management/inside-windows-nt-scheduler-part-1`.

Russinovich, M., Solomon, D., and Ionescu, A. (2012). *Windows internals*. Microsoft Press, Redmond, WA, 6th edition.

Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., and Lyon, B. (1985). Design and implementation of the Sun network filesystem. In *Proc. Summer USENIX Conf.*, pages 119–130.

Schaffer, J. and Reid, S. (2011). The joy of scheduling. QNX White Paper, `http://qnx.symmetry.com.au/resources/whitepapers/qnx_joy_of_scheduling.pdf`.

Sechrest, S. (1986). An introductory 4.4 BSD interprocess communication tutorial. In *Unix Programmer's Supplementary Documents*, volume 1 (PS1). 4.3 Berkeley Software Distribution, Department of Electrical Engineering and Computer Science, University of California, Berkeley.

Smith, M. (2010). Windows IT Pro: A 15-year perspective. *Windows IT Pro Magazine*. `http://windowsitpro.com/windows-server/windows-it-pro-15-year-perspective`.

Stallman, R. (2002). Chapter 1 the GNU project. In *Free software, free society: Selected essays of Richard M. Stallman*, pages 13–25. GNU Press.

Tanenbaum, A. S. and Woodhull, A. S. (2006). *Operating Systems Design and Implementation*. Prentice Hall, Upper Saddle River, NJ, 3rd edition.

Van Riel, R. (2001). Page replacement in Linux 2.4 memory management. In *USENIX Annual Tech. Conf., FREENIX Track*, pages 165–172.

Waldo, J. (1998). Remote procedure calls and Java Remote Method Invocation. *IEEE Concurrency*, 6(3):5–7.