Sistemas Operativos

Process

The Abstraction: A Process

The definition of a **process**, informally, is quite simple: it is a running program.

The OS creates this illusion by **virtualizing** the CPU. By running one process, then stopping it and running another, and so forth. This basic technique, known as **time sharing** of the CPU.

The natural counterpart of time sharing is **space sharing**, where a resource is divided (in space) among those who wish to use it.

We call the low-level machinery **mechanisms**; mechanisms are low-level methods or protocols that implement a needed piece of functionality.

Machine state: what a program can read or update when it is running. Thus the memory that the process can address (called its **address space**) is part of the process. There are some particularly special **registers** that form part of this machine state (*program counter – PC, stack pointer and frame pointer*). Such I/O information might include a list of the **files** the process currently has open.

Process API

- **Create**
- **Destroy**
- **Wait**
- **Miscellaneous Control**
- **Status**

Process Creation

The first thing that the OS must do to run a program is to **load** its code and any static data into the address space of the process from **disk**.

In early operating systems, the loading process is done **eagerly**; modern OSes perform the process **lazily**.

Some memory must be allocated for the program's run-time **stack** (**local variables, function parameters, and return addresses**).

The OS may also allocate some memory for the program's **heap**, used for explicitly requested dynamically-allocated data.

The OS has three open **file descriptors**, for **standard input, output,** and **error**.

Process States

- **Running**: This means it is executing instructions.

- **Ready**: The OS has chosen not to run it at this given moment.
- **Blocked**: A process has performed some kind of operation that makes it not ready to run until some other event takes place.

Being moved from ready to running means the process has been **scheduled**; being moved from running to ready means the process has been **descheduled**.

These types of decisions are made by the OS **scheduler**.

<u>Data **Structures**</u>

The OS likely will keep some kind of **process list** for all processes that are ready, as well as some additional information to track which process is currently running. The OS must also track, in some way, blocked processes. Individual structure that stores information about a process as a **Process Control Block (PCB).**

The **register context** will hold, for a stopped process, the contents of its registers. When a process is stopped, its registers will be saved to this memory location; by restoring these, the OS can resume running the process. **Context switch**.

Sometimes a system will have an **initial** state that the process is in when it is being created.

Also, a process could be placed in a **final** state where it has exited but has not yet been cleaned up (in UNIX-based systems, this is called the **zombie** state).

Process API

<u>The fork() System Call</u>

The **fork()** system call is used to create a new process.

The process that is created is an (almost) exact copy of the calling process. In UNIX systems, the **process identifier (PID)** is used to name the process if one wants to do something with the process. The newly-created process (called the **child**, in contrast to the creating **parent**) just comes into life as if it had called fork() itself. While the parent receives the PID of the newly-created child, the child receives a return code of zero.

The output is not **deterministic**. The CPU **scheduler** determines which process runs at a given moment in time.

<u>The wait() System Call</u>

This system call won't return until the child has run and exited.

<u>The exec() System Call</u>

Given the name of an executable, and some arguments, it **loads** code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized. Then the OS simply runs that program, passing in any arguments as the argv of that process. It transforms the currently running program into a different running program.

Motivating The API

The **shell** is just a user program . It shows you a prompt and then waits for you to type something into it. You then type a command into it; the shell then figures out where in the file system the executable resides, calls **fork()** to create a new child process to run the command, calls some variant of **exec()** to run the command, and then waits for the command to complete by calling wait(). When the child completes, the shell returns from **wait()** and prints out a prompt again, ready for your next command.

The output of the program is redirected into another output file: the shell closes **standard output** and opens the file. Specifically, UNIX systems start looking for free **file descriptors** at zero.

The **pipe()** system call. In this case, the output of one process is connected to an **inkernel pipe** (i.e., **queue**), and the input of another process is connected to that same pipe.

The **kill()** system call is used to send **signals** to a process, including directives to go to sleep, die, and others.

Mechanism: Limited Direct Execution

Basic Technique: Limited Direct Execution

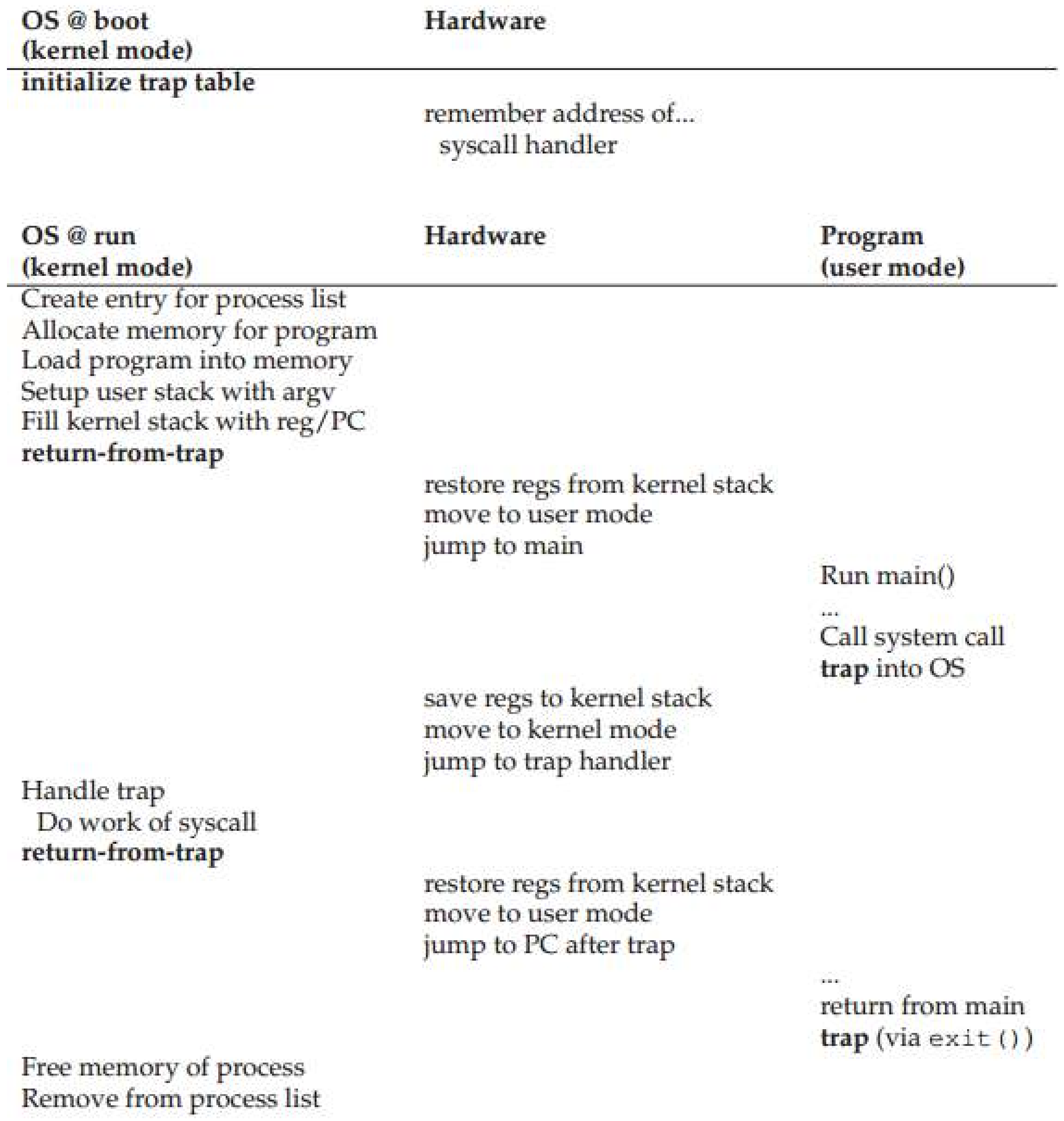
Limited direct execution: run the program directly on the CPU. When the OS wishes to start a program running, it creates a process entry for it in a process list, allocates some memory for it, loads the program code into memory (from disk), locates its entry, jumps to it, and starts running the user's code.

Restricted Operations

In **user mode**, applications do not have full access to hardware resources. In **kernel mode**, the OS has access to the full resources of the machine. Special instructions (**system calls**) to **trap** into the kernel and **return-from-trap** back to user-mode programs are also provided, as well as instructions that allow the OS to tell the hardware where the **trap table** resides in memory.

On x86, for example, the processor will push the program counter, flags, and a few other registers onto a per-process **kernel stack**; the return-from-trap will pop these values off the stack and resume execution of the usermode program.

The kernel sets up a **trap table** at boot time. When the machine boots up, it does so in privileged (kernel) mode. One of the first things the OS thus does is to tell the hardware what code to run when certain exceptional events occur. The OS informs the hardware of the locations of these **trap handlers**, usually with some kind of special instruction. To specify the exact system call, a **system-call** number is usually assigned to each system call. This level of indirection serves as a form of protection. It is a privileged operation.



OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember address of... syscall handler

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
Create entry for process list Allocate memory for program Load program into memory Setup user stack with argv Fill kernel stack with reg/PC **return-from-trap**		
	restore regs from kernel stack move to user mode jump to main	
		Run main() ... Call system call **trap** into OS
	save regs to kernel stack move to kernel mode jump to trap handler	
Handle trap Do work of syscall **return-from-trap**		
	restore regs from kernel stack move to user mode jump to PC after trap	
		... return from main **trap** (via `exit()`)
Free memory of process Remove from process list		

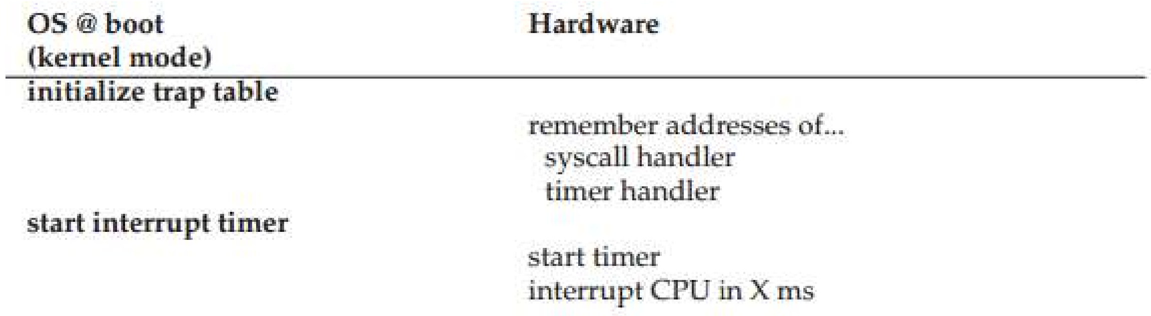
Switching Between Processes

How can the operating system **regain control** of the CPU so that it can switch between processes?

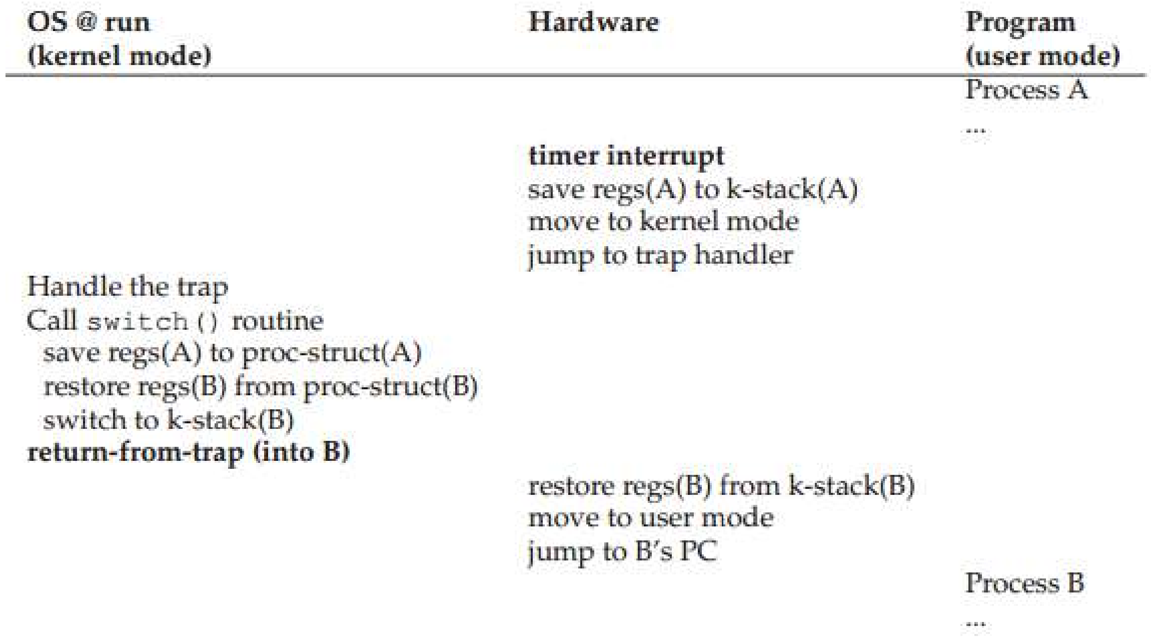
- *A Cooperative Approach: Wait For System Calls*: Systems like this often include an explicit **yield** system call, which does nothing except to transfer control to the OS so it can run other processes. Applications also transfer control to the OS when they do something **illegal.**
- *A Non-Cooperative Approach: The OS Takes Control*: The addition of **a timer interrupt** gives the OS the ability to run again on a CPU even if processes act in a non-cooperative fashion.

Saving and Restoring Context

A **context switch** is conceptually simple: all the OS has to do is save a few register values for the currently-executing process (onto its *kernel stack*, for example) and restore a few for the soon-to-be-executing process (from its kernel stack).



OS @ boot (kernel mode)	**Hardware**
initialize trap table	
	remember addresses of... syscall handler timer handler
start interrupt timer	
	start timer interrupt CPU in X ms



OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
		Process A ...
	timer interrupt save regs(A) to k-stack(A) move to kernel mode jump to trap handler	
Handle the trap Call `switch()` routine save regs(A) to proc-struct(A) restore regs(B) from proc-struct(B) switch to k-stack(B) **return-from-trap (into B)**		
	restore regs(B) from k-stack(B) move to user mode jump to B's PC	
		Process B ...

Switches contexts, specifically by changing the **stack pointer** to use B's kernel stack (and not A's). The kernel registers are explicitly saved by the software (i.e., the OS), but this time into memory in the process structure of the process.

Scheduling

Workload Assumptions

Processes running in the system, sometimes collectively called the **workload.**

1. Each job runs for the same amount of time.
2. All jobs arrive at the same time.
3. Once started, each job runs to completion.
4. All jobs only use the CPU.
5. The run-time of each job is known.

Scheduling Metrics

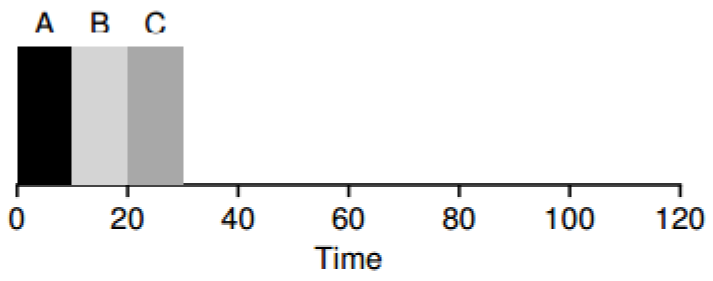
Turnaround time:

$$T_{turnaround} = T_{completion} - T_{arrival}$$

Two other relevant metrics are **performance** and **fairness**.

First In, First Out (FIFO)

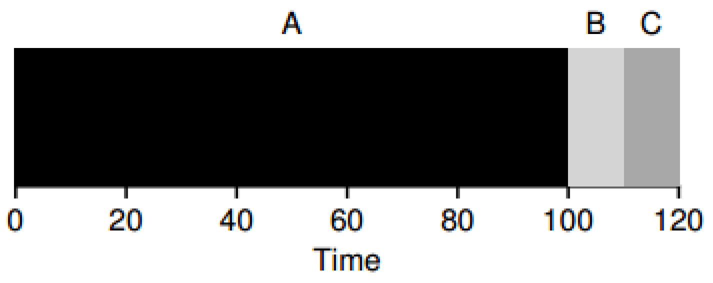
Three jobs arrive in the system, A, B, and C, at roughly the same time. A arrived just a hair before B which arrived just a hair before C. Assume also that each job runs for 10 second:



The average turnaround time for the three jobs is simply:

$$\frac{10+20+30}{3} = 20$$

A runs for 100 seconds while B and C run for 10 each:

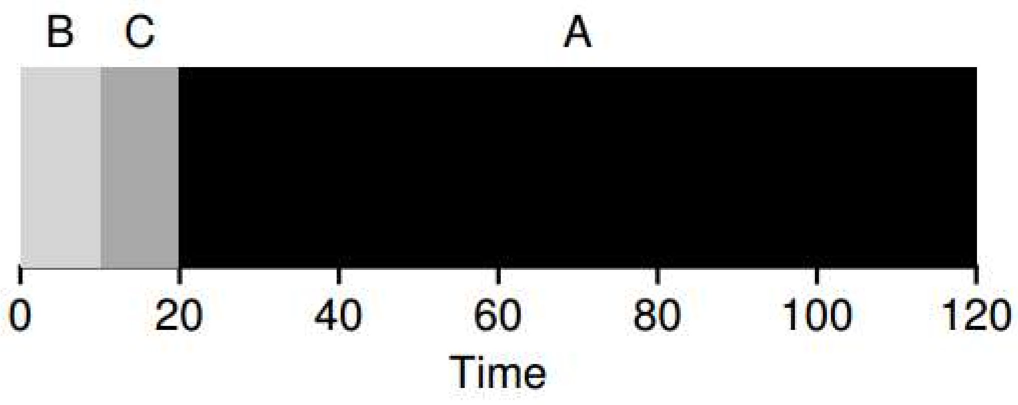


The average turnaround time:

$$\frac{100+110+120}{3} = 110$$

Convoy effect.

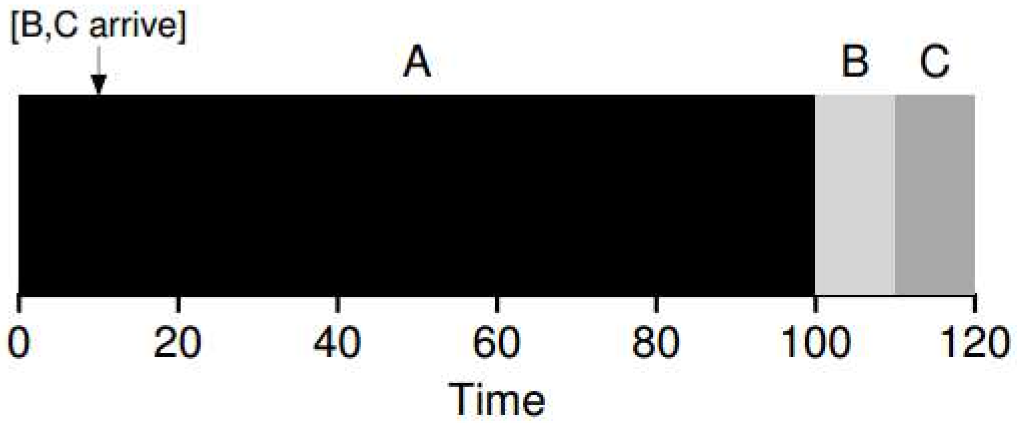
Shortest Job First (SJF)



Average turnaround:

$$\frac{10+20+120}{3} = 50$$

A arrives at t = 0 and needs to run for 100 seconds, whereas B and C arrive at t = 10 and each need to run for 10 seconds.

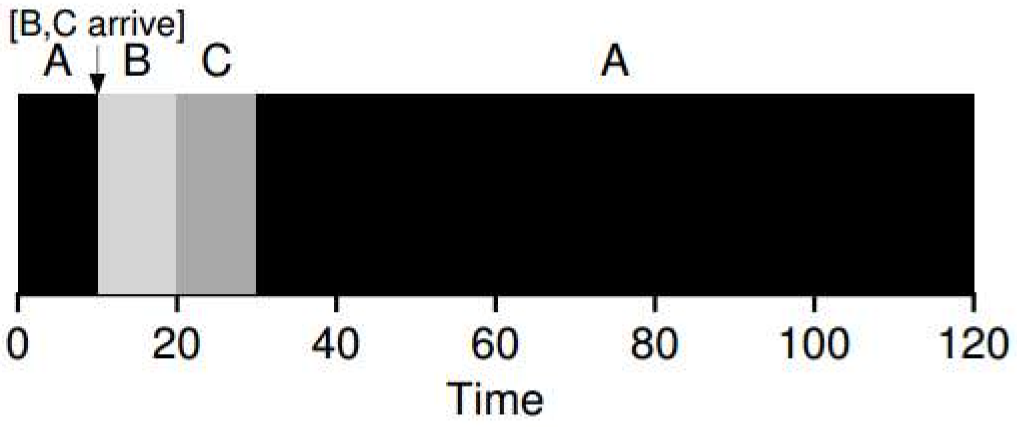


Average turnaround time: 103.33

$$\frac{100+(110-10)+(120-10)}{3}$$

Shortest Time-to-Completion First (STCF)

Non-preemptive schedulers run each job to completion before considering whether to run a new job. Virtually all modern schedulers are **preemptive**, the scheduler can perform a context switch, stopping one running process temporarily and resuming (or starting) another.



Average turnaround time: 50

$$\frac{(120-0)+(20-10)+(30-10)}{3}$$

Response Time

$$T_{response} = T_{firstrun} - T_{arrival}$$

Round Robin

Instead of running jobs to completion, RR runs a job for a **time slice** (sometimes called a **scheduling quantum**) and then switches to the next job in the run queue. It repeatedly does so until the jobs are finished. For this reason, RR is sometimes called **time-slicing**. Note that the length of a time slice must be a multiple of the timer-interrupt period.

Assume three jobs A, B, and C arrive at the same time in the system, and that they each wish to run for 5 seconds. RR with a time-slice of 1 second would cycle through the jobs quickly.

Average response time:

$$\frac{0+1+2}{3} = 1;$$

Deciding on the length of the time slice presents a trade-off to a system designer, making it long enough to **amortize** the cost of switching without making it so long that the system is no longer responsive.

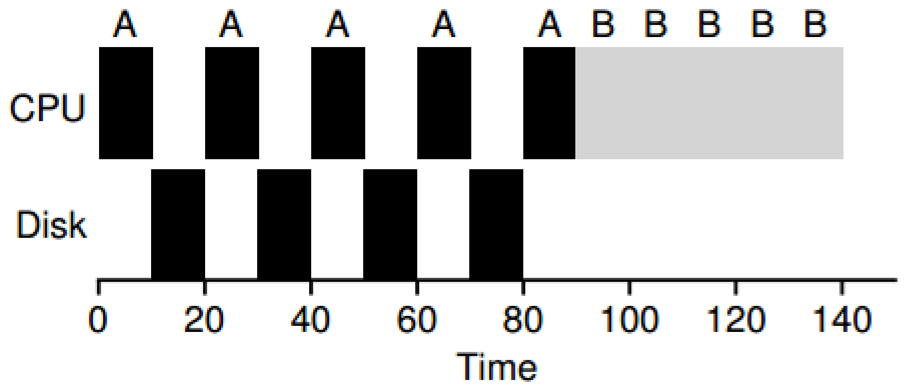
A finishes at 13, B at 14, and C at 15, for an average turnaround time of 14.

Any policy (such as RR) that is **fair**, i.e., that evenly divides the CPU among active processes on a small time scale, will perform poorly on metrics such as turnaround time.

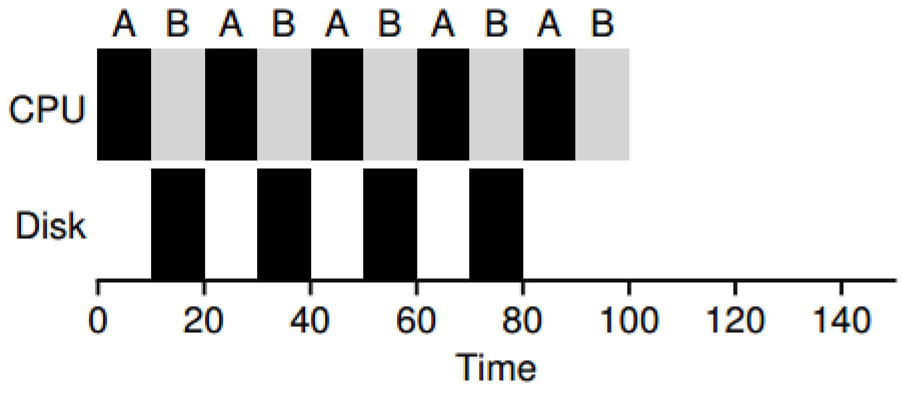
Incorporating I/O

A scheduler clearly has a decision to make when a job initiates an I/O request, because the currently-running job won't be using the CPU during the I/O; it is **blocked** waiting for I/O completion. The scheduler also has to make a decision when the I/O completes.

A and B, which each need 50 ms of CPU time. A runs for 10 ms and then issues an I/O request (assume here that I/Os each take 10 ms), whereas B simply uses the CPU for 50 ms and performs no I/O. The scheduler runs A first, then B after.



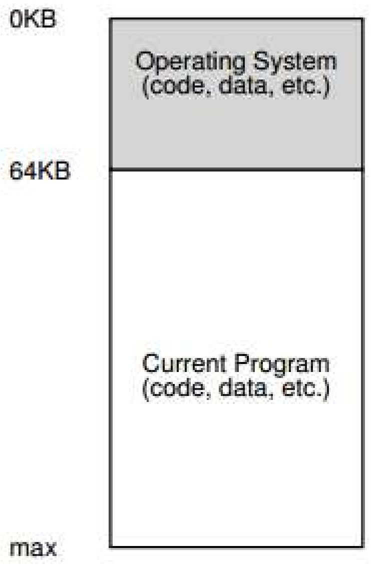
A common approach is to treat each 10-ms sub-job of A as an independent job. With STCF, the choice is clear: choose the shorter one, in this case A. Then, when the first sub-job of A has completed, only B is left, and it begins running. Then a new sub-job of A is submitted, and it preempts B and runs for 10 ms. Doing so allows for **overlap**, with the CPU being used by one process while waiting for the I/O of another process to complete.



Address Spaces

Early Systems

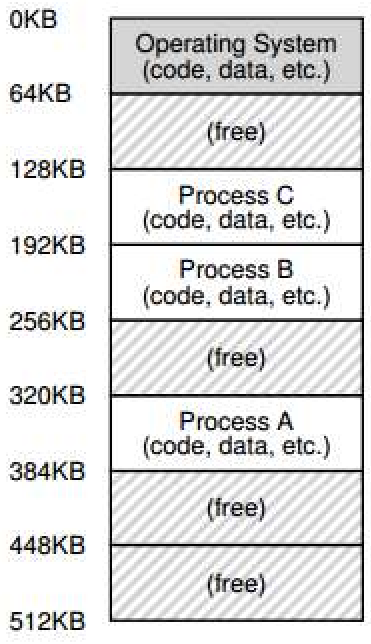
The OS was a set of routines (a library, really) that sat in memory, and there would be one running program (a process) that currently sat in **physical memory** and used the rest of memory.



Multiprogramming and Time Sharing

Multiple processes were ready to run at a given time, and the OS would **switch** between them. Doing so increased the **effective utilization** of the CPU. **Time sharing.** The notion of **interactivity** became important.

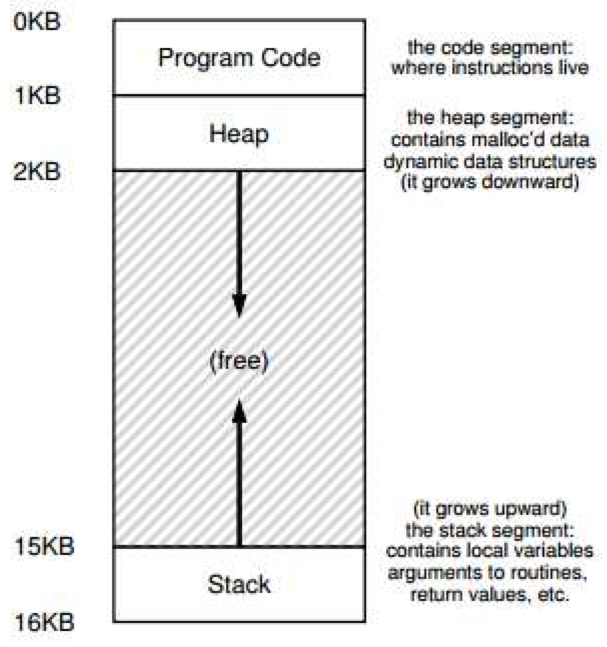
Leave processes in memory while switching between them, allowing the OS to implement time sharing efficiently.



The Address Space

Easy to use abstraction of physical memory. **Address space**, and it is the running program's view of memory in the system.

The address space of a process contains all of the memory state of the running program: the code, the stack and the heap.


0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.

The **abstraction** that the OS is providing to the running program. The OS is **virtualizing memory**, because the running program thinks it is loaded into memory at a particular address (say 0) and has a potentially very large address space.

By using memory **isolation**, the OS further ensures that running programs cannot affect the operation of the underlying OS.

Goals

- **Transparency**: the OS should implement virtual memory in a way that is invisible to the running program.
- **Efficiency**: the OS should strive to make the virtualization as efficient as possible, both in terms of time and space.
- **Protection**: the OS should make sure to protect processes from one another as well as the OS itself from processes).

Hardware-based address translation/address translation: the hardware transforms each memory access, changing the **virtual address** provided by the instruction to a **physical address** where the desired information is actually located.

It must thus **manage memory**, keeping track of which locations are free and which are in use.

In virtualizing memory, the hardware will **interpose** on each memory access, and translate each virtual address issued by the process to a physical address where the desired information is actually stored.

Assumptions

- The user's address space must be placed contiguously in physical memory.
- The address space is less than the size of physical memory.
- Address space is exactly the same size.

Dynamic (Hardware-based) Relocation

Two hardware registers within each CPU: one is called the **base** register, and the other the **bounds**.

Memory reference is generated by the process, it is **translated** by the processor:

```
physical address = virtual address + base
```

Address translation: the hardware takes a virtual address the process thinks it is referencing and transforms it into a physical address which is where the data actually resides. This relocation of the address happens at runtime (**dynamic relocation**).

The processor will first check that the memory reference is within **bounds** to make sure it is legal.

The part of the processor that helps with address translation the **memory management unit (MMU).**

Bound registers: 1) it holds the size of the address space 2) it holds the physical address of the end of the address space.

The hardware should provide special instructions to modify the base and bounds registers, allowing the OS to change them when different processes run.

The CPU must be able to generate **exceptions** in situations where a user program tries to access memory illegally.

Operating System Issues

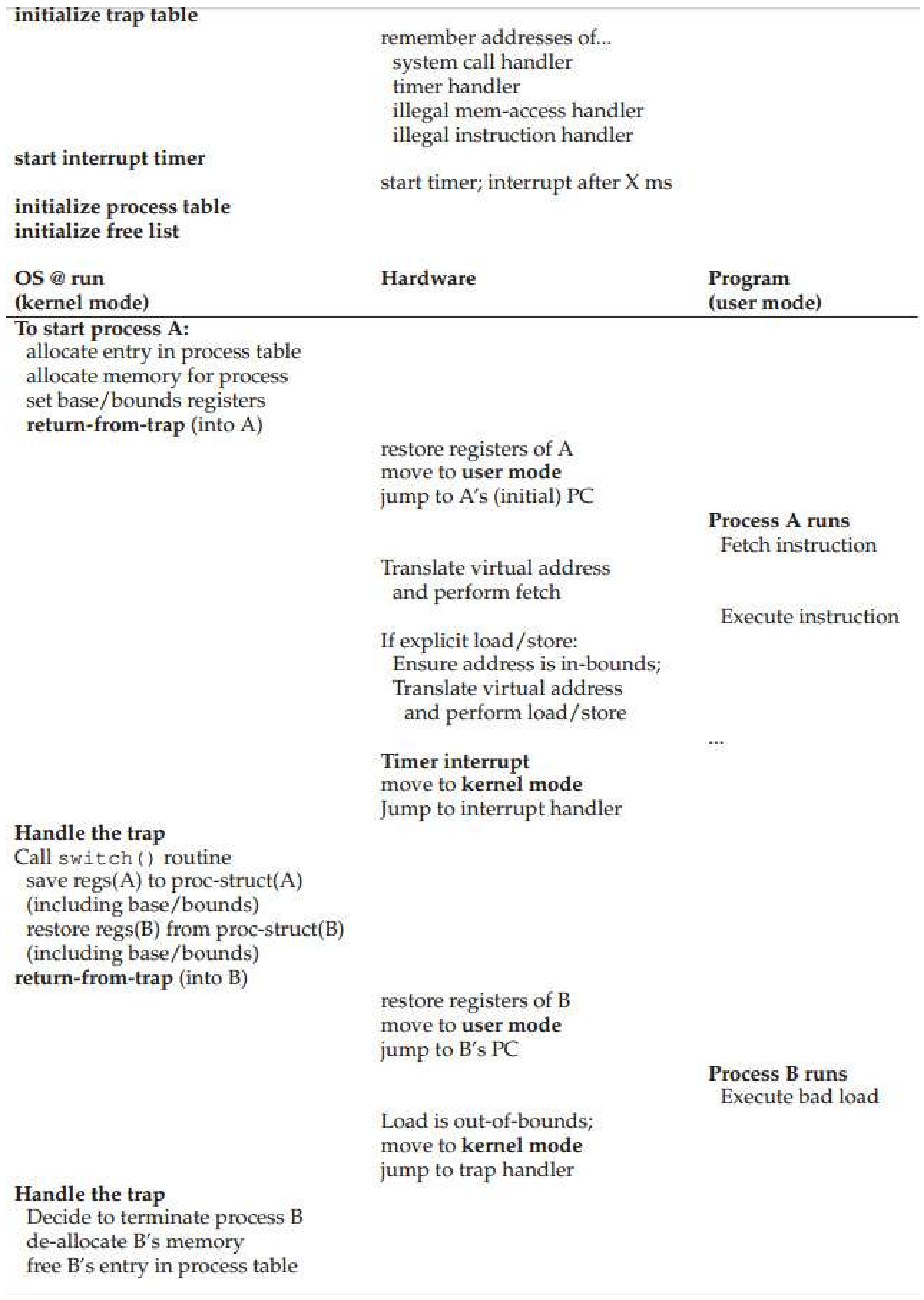
When a new process is created, the OS will have to search a data structure (often called a **free list**) to find room for the new address space and then mark it used.

Upon termination of a process, the OS thus puts its memory back on the free list, and cleans up any associated data structures as need be.

The OS decides to stop running a process, it must save the values of the base and bounds registers to memory, in some per-process structure such as the **process structure** or **process control block (PCB).**

To move a process's address space, the OS first deschedules the process; then, the OS copies the address space from the current location to the new location; finally, the OS updates the saved base register (in the process structure) to point to the new location.

The OS must provide **exception handlers**, or functions to be called, as discussed above; the OS installs these handlers at boot time (via privileged instructions).



initialize trap table		
	remember addresses of...	
	system call handler	
	timer handler	
	illegal mem-access handler	
	illegal instruction handler	
start interrupt timer		
	start timer; interrupt after X ms	
initialize process table		
initialize free list		

OS @ run (kernel mode)	**Hardware**	**Program (user mode)**
To start process A:		
allocate entry in process table		
allocate memory for process		
set base/bounds registers		
return-from-trap (into A)		
	restore registers of A	
	move to **user mode**	
	jump to A's (initial) PC	
		Process A runs
		Fetch instruction
	Translate virtual address and perform fetch	
		Execute instruction
	If explicit load/store:	
	Ensure address is in-bounds;	
	Translate virtual address and perform load/store	
		...
	Timer interrupt	
	move to **kernel mode**	
	Jump to interrupt handler	
Handle the trap		
Call `switch()` routine		
save regs(A) to proc-struct(A) (including base/bounds)		
restore regs(B) from proc-struct(B) (including base/bounds)		
return-from-trap (into B)		
	restore registers of B	
	move to **user mode**	
	jump to B's PC	
		Process B runs
		Execute bad load
	Load is out-of-bounds;	
	move to **kernel mode**	
	jump to trap handler	
Handle the trap		
Decide to terminate process B		
de-allocate B's memory		
free B's entry in process table		

Segmentation

Segmentation: Generalized Base/Bounds

Having a base and bounds pair per logical segment of the address space. A **segment** is just a contiguous portion of the address space of a particular length: code, stack, and heap.

MMU required to support segmentation: a set of three base and bounds register pairs.

The hardware detects that the address is out of bounds, traps into the OS, likely leading to the termination of the offending process (**segmentation violation** or **segmentation fault**).

Which Segment Are We Referring To?

Explicit approach: chop up the address space into segments based on the top few bits of the virtual address.

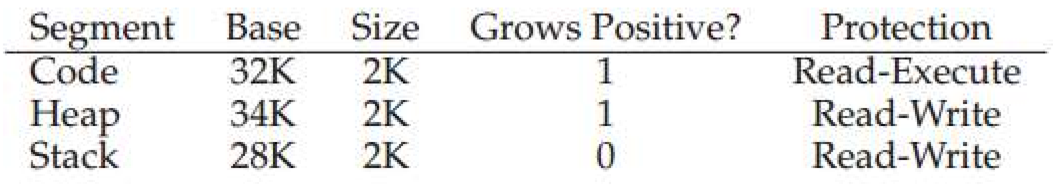
Implicit approach: the hardware determines the segment by noticing how the address was formed.

What About The Stack?

Instead of just base and bounds values, the hardware also needs to know which way the segment grows (a bit, for example, that is set to 1 when the segment grows in the positive direction, and 0 for negative).

Support for Sharing

It is useful to share certain memory segments between address spaces. **Code sharing:** we need **protection bits**. Basic support adds a few bits per segment, indicating whether or not a program can read or write a segment, or perhaps execute code that lies within the segment.



Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

Fine-grained vs. Coarse-grained Segmentation

Coarse-grained: it chops up the address space into relatively large, coarse chunks.

Fine-grained segmentation: allowed for address spaces to consist of a large number of smaller segments. Supporting many segments requires a **segment table** stored in memory.

OS Support

External fragmentation: physical memory quickly becomes full of little holes of free space, making it difficult to allocate new segments, or to grow existing ones.

One solution to this problem would be to **compact** physical memory by rearranging the existing segments.

A simpler approach is to use a free-list management algorithm that tries to keep large extents of memory available for allocation.

- **Best fit**: first, search through the free list and find chunks of free memory that are as big or bigger than the requested size. Then, return the one that is the smallest in that group of candidates.
- **Worst fit**: find the largest chunk and return the requested amount.
- **First fit**: finds the first block that is big enough and returns the requested amount to the user.
- **Next fit:** keeps an extra pointer to the location within the list where one was looking last.

Segregated lists: if a particular application has a few popular-sized request that it makes, keep a separate list just to manage objects of that size.

Binary buddy allocator: When a request for memory is made, the search for free space recursively divides free space by two until a block that is big enough to accommodate the request is found. When returning a block to the free list, the allocator checks whether the "buddy" is free; if so, it coalesces the two blocks into a double block. This recursive **coalescing process** continues up the tree, either restoring the entire free space or stopping when a buddy is found to be in use.

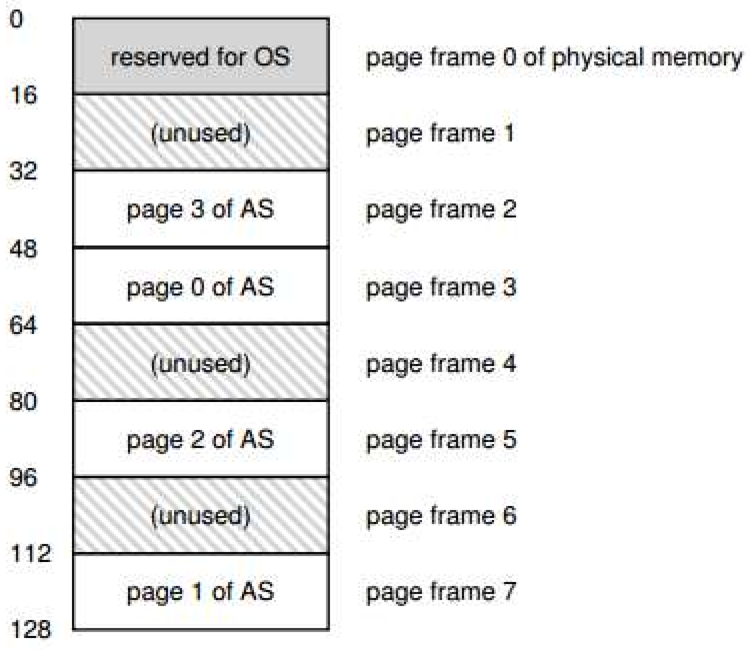
Paging

The Abstraction: A Process

Paging: to chop up space into fixed-sized pieces (**page**). Correspondingly, we view physical memory as an array of fixed-sized slots called **page frames**.

Advantages:

- **Flexibility**: the system will be able to support the abstraction of an address space effectively, regardless of how a process uses the address space.
- **Simplicity**: simple free-space management (**free list** of pages).



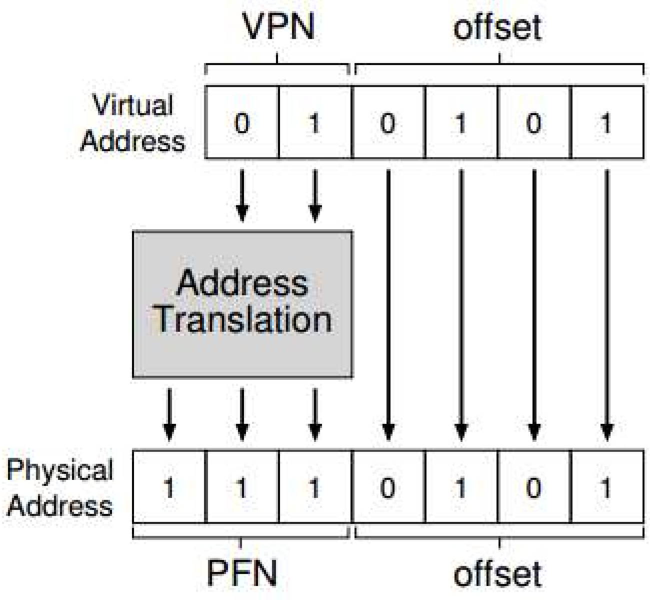
The major role of the **page table (per-process** data structure) is to store **address translations** for each of the virtual pages of the address space.

To **translate** the virtual address that the process generated, we have to first split it into two components: the **virtual page number (VPN)**, and the **offset** within the page.

Virtual address space of the process is 64 bytes, we need 6 bits total for our virtual address.

The page size is 16 bytes in a 64 -byte address space; thus we need to be able to select 4 pages, and the top 2 bits of the address do just that. Thus, we have a 2-bit **virtual page number (VPN)**. The remaining bits tell us which byte of the page we are interested in, 4 bits in this case; we call this the **offset**.

With our **virtual page number**, we can now index our page table and find which **physical frame virtual page** resides within: the **physical frame number (PFN)** (also sometimes called the **physical page number** or **PPN**). Thus, we can translate this virtual address by replacing the VPN with the PFN and then issue the load to physical memory.

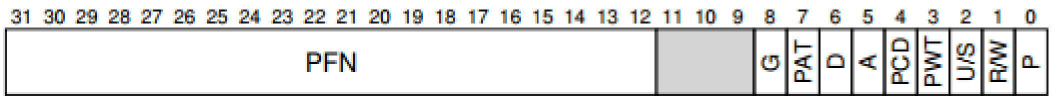


We store the page table for each process in **memory**.

The **page table** is just a data structure that is used to map virtual addresses (or really, **virtual page numbers**) to physical addresses (**physical frame numbers**). The simplest form is called a **linear page table**, which is just an array. The OS indexes the array by the **virtual page number (VPN)**, and looks up the **page-table entry (PTE)** at that index in order to find the desired **physical frame number (PFN)**.

Contents:

- A **valid bit** is to indicate whether the particular translation is valid.
- **Protection bits**, indicating whether the page could be read from, written to, or executed from.
- A **present bit** indicates whether this page is in physical memory or on disk (i.e., it has been **swapped out**).
- A **dirty bit**, indicating whether the page has been modified since it was brought into memory.
- A **reference bit** (a.k.a. **accessed bit**) is sometimes used to track whether a page has been accessed.



Page-table base register contains the physical address of the starting location of the page table.

```
// Extract the VPN from the virtual address
VPN = (VirtualAddress & VPN_MASK) >> SHIFT

// Form the address of the page-table entry (PTE)
PTEAddr = PTBR + (VPN * sizeof(PTE))

// Fetch the PTE
PTE = AccessMemory(PTEAddr)

// Check if process can access the page
if (PTE.Valid == False)
    RaiseException(SEGMENTATION_FAULT)
else if (CanAccess(PTE.ProtectBits) == False)
    RaiseException(PROTECTION_FAULT)
else
    // Access is OK: form physical address and fetch it
    offset   = VirtualAddress & OFFSET_MASK
    PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
    Register = AccessMemory(PhysAddr)
```

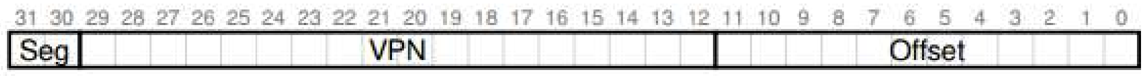
Bigger Pages

Big pages lead to waste within each page, a problem known as **internal fragmentation**.

Hybrid Approach: Paging and Segments

We use the **base** not to point to the segment itself but rather to hold the physical address of the page table of that segment. The **bounds** register is used to indicate the end of the page table.

In the hardware, assume that there are thus three base/bounds pairs, one each for **code**, **heap**, and **stack**.



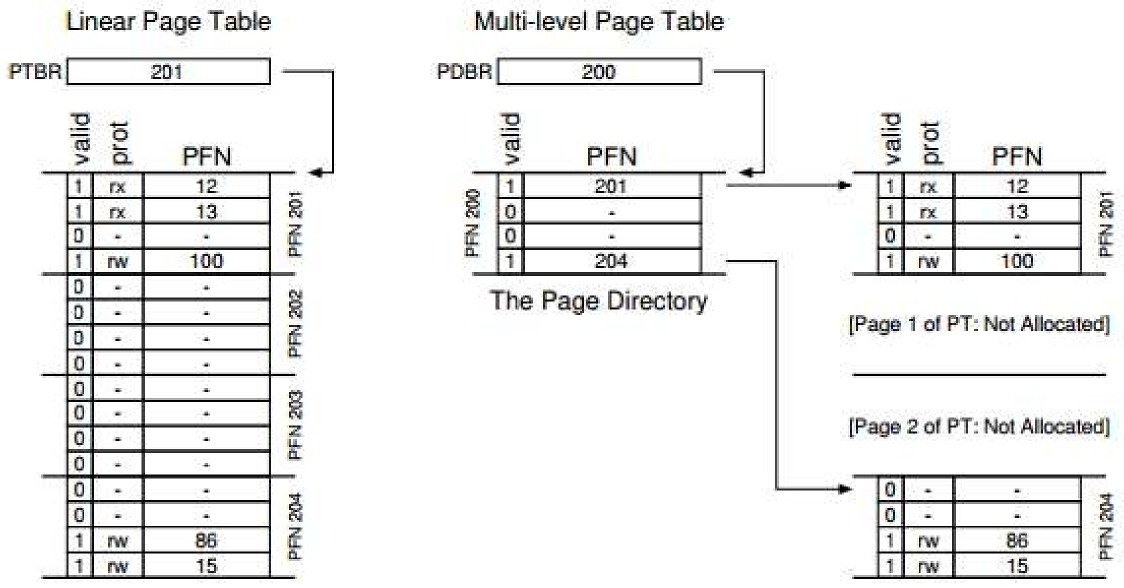
```
SN           = (VirtualAddress & SEG_MASK) >> SN_SHIFT
VPN          = (VirtualAddress & VPN_MASK) >> VPN_SHIFT
AddressOfPTE = Base[SN] + (VPN * sizeof(PTE))
```

Segmentation is not quite as **flexible** as we would like, as it assumes a certain usage pattern of the address space.

This hybrid causes **external fragmentation** to arise again.

Multi-level Page Tables

First, chop up the page table into page-sized units; then, if an entire page of page-table entries (PTEs) is invalid, don't allocate that page of the page table at all. To track whether a page of the page table is valid (and if valid, where it is in memory), use a new structure, called the **page directory**.



The page directory consists of a number of **page directory entries (PDE)**. A PDE has a **valid bit** and a **page frame number (PFN)**. If the PDE entry is valid, it means that at least one of the pages of the page table that the entry points to (via the PFN) is valid, i.e., in at least one PTE on that page pointed to by this PDE, the valid bit in that PTE is set to one. If the PDE entry is not valid (i.e., equal to zero), the rest of the PDE is not defined.

The multi-level table only allocates page-table space in proportion to the amount of address space you are using. We add a level of **indirection** through use of the page directory, which points to pieces of the page table; that indirection allows us to place page-table pages wherever we would like in physical memory.

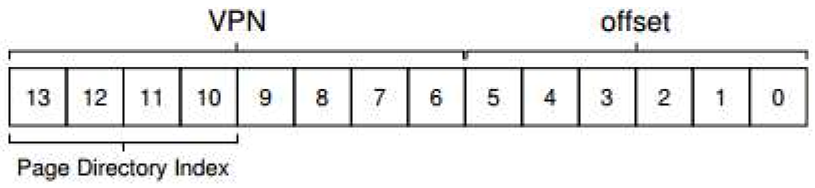
On a TLB miss, two loads from memory will be required to get the right translation. Thus, the multi-level table is a small example of a **time-space trade-off**. Another obvious negative is **complexity.**

Example – you can skip it

Imagine a small **address space** of size **16KB**, with **64-byte pages**. Thus, we have a **14-bit virtual address space**, with **8 bits** for the **VPN** and **6 bits** for the **offset**. A **linear page table** would have **2^8 (256) entries**, even if only a small portion of the address space is in use.

To build a **two-level page table** for this address space, we start with our full linear page table and break it up into page-sized units. Assume each **PTE** is **4 bytes** in size. Thus, our **page table** is **1KB (256 × 4 bytes)** in size. Given that we have **64-byte pages**, the **1KB page table** can be divided into **16 64-byte pages**; each **page** can hold **16 PTEs**.

The **page directory** needs one entry per page of the page table; thus, it has **16 entries**.

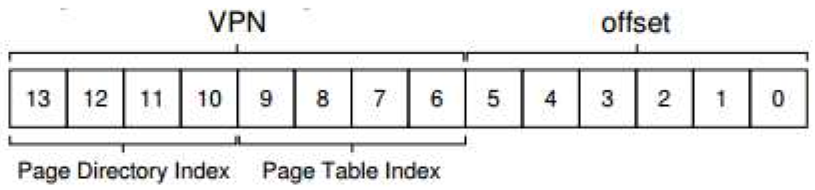


We extract the **page-directory index (PDInde**x) from the VPN. We can use it to find the address of the page-directory entry (PDE) with a simple calculation:

PDEAddr = PageDirBase + (PDIndex * sizeof(PDE)).

If the page-directory entry is marked invalid, we know that the access is **invalid**, and thus raise an **exception**.

If, however, the PDE is valid, we now have to fetch the pagetable entry (PTE) from the page of the page table pointed to by this pagedirectory entry.



This **page-table index (PTIndex)** can then be used to index into the page table itself, giving us the address of our PTE:

PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))

Page Directory		Page of PT (@PFN:100)			Page of PT (@PFN:101)		
PFN	valid?	PFN	valid	prot	PFN	valid	prot
100	1	10	1	r-x	—	0	—
—	0	23	1	r-x	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	80	1	rw-	—	0	—
—	0	59	1	rw-	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	—	0	—
—	0	—	0	—	55	1	rw-
101	1	—	0	—	45	1	rw-

In **physical page 100** (the **physical frame number** of the **0th page** of the **page table**), **VPNs 0** and **1** are valid (the **code segment**), as are **4** and **5** (the **heap**). **PFN 101**, **VPNs 254** and **255** (the **stack**) have valid mappings.

Here is an address that refers to the **0th byte of VPN 254: 0x3F80**, or **11 1111 1000 0000** in binary.

The **top 4 bits** of the **VPN** to **index** into the **page directory**. Thus, **1111** will choose the **last** (**15th**, if you start at the 0th) entry of the **page directory.**

This points us to a **valid page** of the **page table** located at **address 101.** We then use the **next 4 bits** of the **VPN (1110)** to index into that page of the **page table** and find the desired **PTE. 1110** is the **next-to-last (14th) entry** on the page, and tells us that **page 254** of our **virtual address space** is **mapped** at **physical page 55**.

By **concatenating PFN=55 (or hex 0x37)** with **offset=000000**, we can thus form our desired **physical address** and issue the request to the **memory** system:

PhysAddr = (PTE.PFN << SHIFT) + offset = 00 1101 1100 0000 = 0x0DC0

More Than Two Levels

Splitting the **page directory** itself into multiple pages, and then adding another page directory on top of that, to point to the pages of the page directory.

The Translation Process

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)   // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                  // TLB Miss
    // first, get page directory entry
    PDIndex = (VPN & PD_MASK) >> PD_SHIFT
    PDEAddr = PDBR + (PDIndex * sizeof(PDE))
    PDE     = AccessMemory(PDEAddr)
    if (PDE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        // PDE is valid: now fetch PTE from page table
        PTIndex = (VPN & PT_MASK) >> PT_SHIFT
        PTEAddr = (PDE.PFN << SHIFT) + (PTIndex * sizeof(PTE))
        PTE     = AccessMemory(PTEAddr)
        if (PTE.Valid == False)
            RaiseException(SEGMENTATION_FAULT)
        else if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
```

Inverted Page Tables

We keep a single page table that has an entry for each physical page of the system. The entry tells us which process is using this page, and which virtual page of that process maps to this **physical page**. Finding the correct entry is now a matter of **searching through** this data structure.

Swap Space

Swap space: we swap pages out of memory to disk and swap pages into memory from it. The OS will need to remember the **disk address** of a given page.

The Present Bit

In the **PTE**, If the **present bit** is set to one, it means the page is present in physical memory; if it is set to zero, the page is on disk. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault**.

The Page Fault

Page-fault handler: If a page is not present and has been swapped to disk, the OS will need to swap the page into memory in order to service the page fault. When the OS receives a page fault for a page, it looks in the **PTE** to find the **disk address**, and issues the request to disk to **fetch** the page into memory. When the **disk I/O** completes, the OS will then update the page table to mark the page as present, update the PFN field of the page-table entry (PTE) to record the in-memory location of the newly-fetched page, and retry the instruction.

The process will be in the **blocked** state, this allows the **overlap** of processes.

What If Memory Is Full?

Page-replacement policy: **page out** one or more pages to make room to **page in** a page from swap space.

Page Fault Control Flow

```
VPN = (VirtualAddress & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success == True)    // TLB Hit
    if (CanAccess(TlbEntry.ProtectBits) == True)
        Offset   = VirtualAddress & OFFSET_MASK
        PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
        Register = AccessMemory(PhysAddr)
    else
        RaiseException(PROTECTION_FAULT)
else                    // TLB Miss
    PTEAddr = PTBR + (VPN * sizeof(PTE))
    PTE = AccessMemory(PTEAddr)
    if (PTE.Valid == False)
        RaiseException(SEGMENTATION_FAULT)
    else
        if (CanAccess(PTE.ProtectBits) == False)
            RaiseException(PROTECTION_FAULT)
        else if (PTE.Present == True)
            // assuming hardware-managed TLB
            TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
            RetryInstruction()
        else if (PTE.Present == False)
            RaiseException(PAGE_FAULT)
```

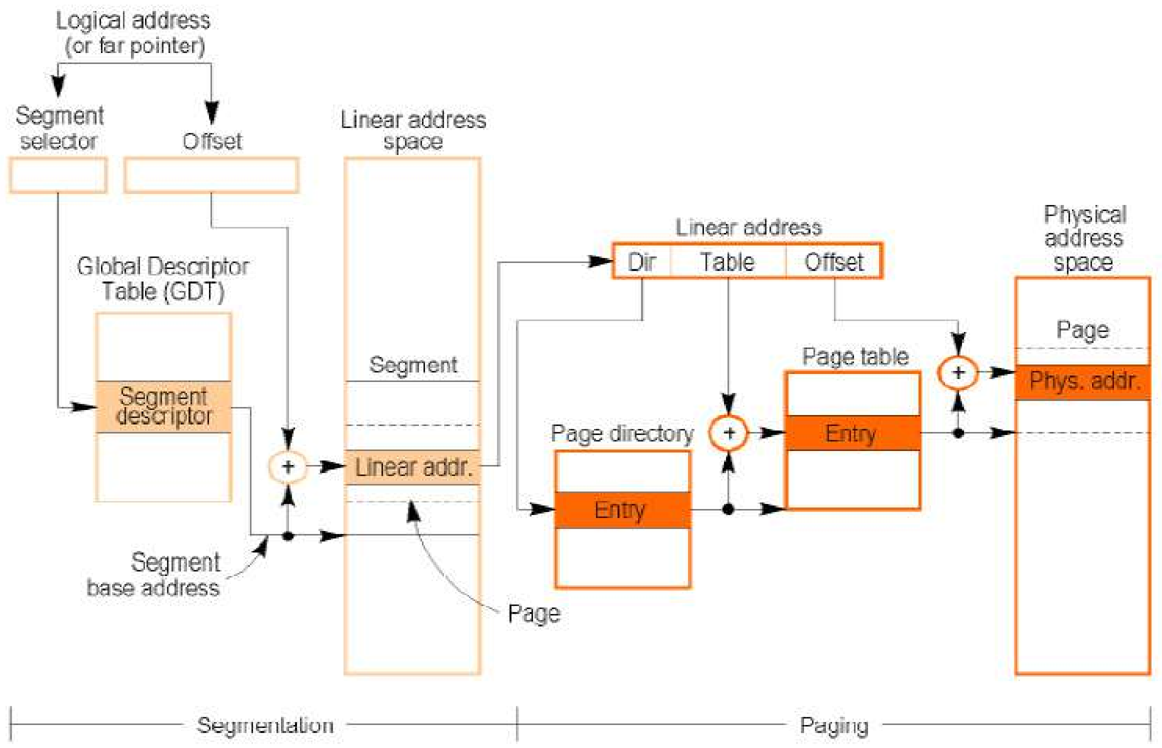
When Replacements Really Occur

To keep a small amount of memory free, most operating systems thus have some kind of **high watermark (HW)** and **low watermark (LW)** to help decide when to start evicting pages from memory.

When the OS notices that there are fewer than LW pages available, a background thread **(swap daemon** or **page daemon**) that is responsible for freeing memory runs. The thread evicts pages until there are HW pages available and goes to sleep.

```
PFN = FindFreePhysicalPage()
if (PFN == -1)              // no free page found
    PFN = EvictPage()       // run replacement algorithm
DiskRead(PTE.DiskAddr, pfn) // sleep (waiting for I/O)
PTE.present = True          // update page table with present
PTE.PFN     = PFN           // bit and translation (PFN)
RetryInstruction()          // retry instruction
```

<u>X86 Memory Management</u>

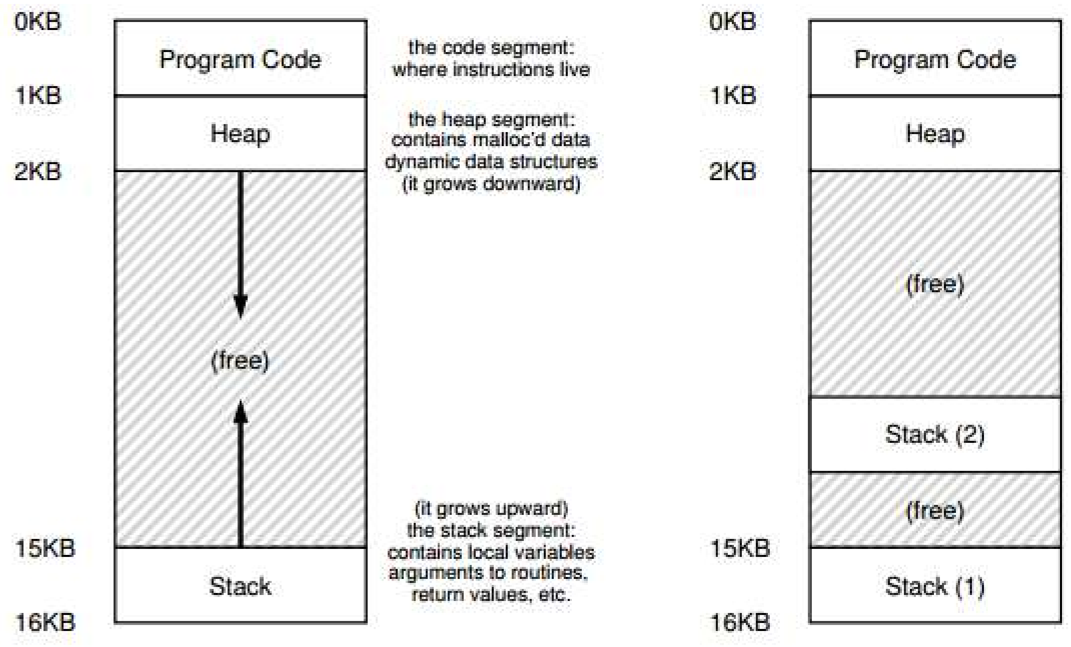


Concurrency

Multi-threaded program has more than one point of execution (i.e., multiple PCs, each of which is being fetched and executed from).

The state of a single **thread:** it has a program counter (PC), its own private set of registers. The **context switch** between, as the register state of T1 must be saved and the register state of T2 restored before running T2. We'll need one or more **thread control blocks (TCBs)** to store the state of each thread of a process. The address space remains the same (i.e., there is no need to switch which page table we are using).

Instead of a single **stack** in the address space, there will be one per thread.



0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack
the code segment: where instructions live
the heap segment: contains malloc'd data dynamic data structures (it grows downward)
(it grows upward) the stack segment: contains local variables arguments to routines, return values, etc.
0KB
1KB
2KB
15KB
16KB
Program Code
Heap
(free)
Stack (2)
(free)
Stack (1)

Thread-local storage: the stack of the relevant thread.

Why Use Threads?

Transforming your standard **single-threaded** program into a program using a thread per CPU is called **parallelization.**

Threading enables **overlap** of I/O with other activities within a single program.

Shared Data: Uncontrolled Scheduling

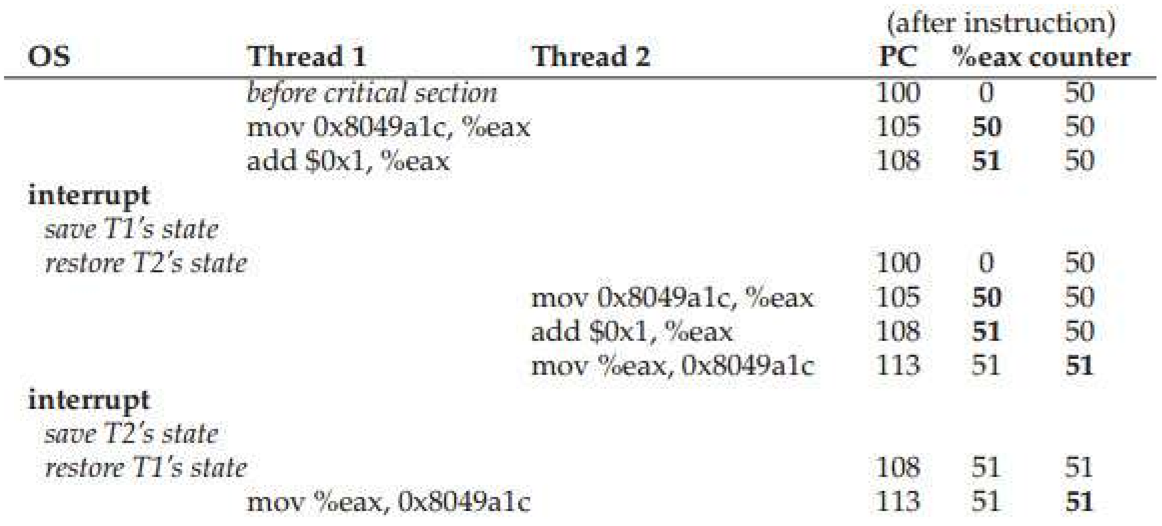
```c
#include <stdio.h>
#include <pthread.h>
#include "mythreads.h"

static volatile int counter = 0;

//
// mythread()
//
// Simply adds 1 to counter repeatedly, in a loop
// No, this is not how you would add 10,000,000 to
// a counter, but it shows the problem nicely.
//
void *
mythread(void *arg)
{
    printf("%s: begin\n", (char *) arg);
    int i;
    for (i = 0; i < 1e7; i++) {
        counter = counter + 1;
    }
    printf("%s: done\n", (char *) arg);
    return NULL;
}

//
// main()
//
// Just launches two threads (pthread_create)
// and then waits for them (pthread_join)
//
int
main(int argc, char *argv[])
{
    pthread_t p1, p2;
    printf("main: begin (counter = %d)\n", counter);
    Pthread_create(&p1, NULL, mythread, "A");
    Pthread_create(&p2, NULL, mythread, "B");

    // join waits for the threads to finish
    Pthread_join(p1, NULL);
    Pthread_join(p2, NULL);
    printf("main: done with both (counter = %d)\n", counter);
    return 0;
}
```



			(after instruction)		
OS	**Thread 1**	**Thread 2**	**PC**	**%eax**	**counter**
	before critical section		100	0	50
	mov 0x8049a1c, %eax		105	**50**	50
	add $0x1, %eax		108	**51**	50
interrupt					
save T1's state					
restore T2's state			100	0	50
		mov 0x8049a1c, %eax	105	**50**	50
		add $0x1, %eax	108	**51**	50
		mov %eax, 0x8049a1c	113	51	**51**
interrupt					
save T2's state					
restore T1's state			108	51	51
	mov %eax, 0x8049a1c		113	51	**51**

Race condition: the results depend on the timing execution of the code. Instead of a **deterministic** computation, we call this result **indeterminate**, where it is not known what the output will be.

Critical section: piece of code that accesses a shared variable (or more generally, a shared resource) and must not be concurrently executed by more than one thread.

Mutual exclusion: if one thread is executing within the critical section, the others will be prevented from doing so.

Atomicity

Atomically, means "as a unit", which sometimes we take as "all or none."

Thus, what we will instead do is ask the hardware for a few useful instructions upon which we can build a general set of what we call **synchronization primitives**.

Thread API

Thread Creation

```
#include <pthread.h>
int
pthread_create(      pthread_t *        thread,
               const pthread_attr_t *  attr,
                     void *             (*start_routine)(void*),
                     void *             arg);
```

Thread Completion

```
int pthread_join(pthread_t thread, void **value_ptr);
```

This routine takes two arguments. The first is of type pthread t, and is used to specify which thread to wait for. The second argument is a pointer to the return value you expect to get back.

Never return a pointer which refers to something allocated on the thread's call stack.

Locks

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Locks: provides mutual exclusion to a critical section.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
```

Or

```
int rc = pthread_mutex_init(&lock, NULL);
assert(rc == 0); // always check success!
```

(Note that a corresponding call to **pthread_mutex_destroy()** should also be made)

Then:

```
pthread_mutex_t lock;
pthread_mutex_lock(&lock);
x = x + 1; // or whatever your critical section is
pthread_mutex_unlock(&lock);
```

```
void Pthread_mutex_lock(pthread_mutex_t *mutex) {
  int rc = pthread_mutex_lock(mutex);
  assert(rc == 0);
}
```

Condition Variables

Condition variables: they are useful when some kind of signaling must take place between threads, if one thread is waiting for another to do something before it can continue.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

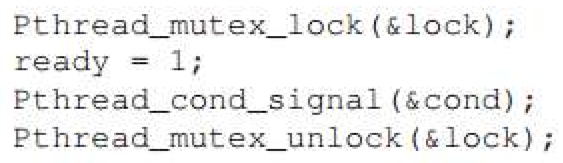
To use a condition variable, one has to in addition have a lock that is associated with this condition. When calling either of the above routines, this lock should be held.

pthread_cond_wait(): puts the calling thread to sleep and releases the lock, and thus waits for some other thread to signal it, usually when something in the program has changed that the now-sleeping thread might care about. Before returning after being woken, re-acquires the lock

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  cond = PTHREAD_COND_INITIALIZER;

Pthread_mutex_lock(&lock);
while (ready == 0)
    Pthread_cond_wait(&cond, &lock);
Pthread_mutex_unlock(&lock);
```

When signaling (as well as when modifying the global variable ready), we always make sure to have the lock held.



```
Pthread_mutex_lock(&lock);
ready = 1;
Pthread_cond_signal(&cond);
Pthread_mutex_unlock(&lock);
```

Locks

Locks: The Basic Idea

A **lock** variable is either **available** and thus no thread holds the lock, or **acquired**, and thus exactly one thread holds the lock and presumably is in a **critical section**.

Calling the routine **lock()** tries to acquire the lock; if it is free, the thread will acquire the lock and enter the critical section (owner of the lock). If another thread then calls **lock()** on that same lock variable, it will not return while the lock is held by another thread. Once the owner of the lock calls **unlock()**, the lock is now available again.

Mutex: POSIX lock is used to provide **mutual exclusion** between threads in a critical section.

Evaluating Locks

Basic criteria:

- **Mutual exclusión**
- **Fairness**
- **Performance**

Lock types:

1. **Controlling Interrupts**: disable interrupts for critical sections.
   Problems: allows any calling thread to perform a **privileged** operation; it does not work on multiprocessors; turning off interrupts for extended periods of time can lead to interrupts becoming lost, which can lead to serious systems problems; it can be inefficient.
2. **Just Using Loads/Stores**:

```
typedef struct __lock_t { int flag; } lock_t;

void init(lock_t *mutex) {
    // 0 -> lock is available, 1 -> held
    mutex->flag = 0;
}

void lock(lock_t *mutex) {
    while (mutex->flag == 1)  // TEST the flag
        ; // spin-wait (do nothing)
    mutex->flag = 1;          // now SET it!
}

void unlock(lock_t *mutex) {
    mutex->flag = 0;
}
```

Thread 1	Thread 2
call `lock()`	
while (flag == 1)	
interrupt: switch to Thread 2	
	call `lock()`
	while (flag == 1)
	flag = 1;
	interrupt: switch to Thread 1
flag = 1; // set flag to 1 (too!)	

Problems: correctness; performance: spin-waiting wastes time waiting for another thread to release a lock.

3. **Test-and-set/atomic exchange:** It returns the old value pointed to by the ptr, and simultaneously updates said value to new. The key, of course, is that this sequence of operations is performed **atomically**.

```
int TestAndSet(int *old_ptr, int new) {
    int old = *old_ptr; // fetch old value at old_ptr
    *old_ptr = new;     // store 'new' into old_ptr
    return old;         // return the old value
}
```

Enough to build a simple **spin lock.**

```
typedef struct __lock_t {
    int flag;
} lock_t;

void init(lock_t *lock) {
    // 0 indicates that lock is available, 1 that it is held
    lock->flag = 0;
}

void lock(lock_t *lock) {
    while (TestAndSet(&lock->flag, 1) == 1)
        ; // spin-wait (do nothing)
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

A thread calls **lock()** and it is **free** (**flag = 0**). The thread calls **TestAndSet(flag, 1)**, the routine will **return** the old value of flag, which is **0**; thus, the calling thread will not get caught spinning in the while loop and will acquire the lock.

A thread already has the lock held (**flag = 1**). The thread will call lock() and then call **TestAndSet(flag, 1)** as well. It will **return** the old value at flag (**1**), while simultaneously **setting** it to **1** again. This thread will spin and spin until the lock is finally released.

It requires a **preemptive scheduler** (i.e., one that will interrupt a thread via a timer).

Problems: spin locks don't provide any **fairness** guarantees; in the single CPU systems, performance **overheads** can be quite painful; on multiple CPUs, spin locks work reasonably well (if the number of threads roughly equals the number of CPUs).

4. **Compare-And-Swap:**

```
int CompareAndSwap(int *ptr, int expected, int new) {
    int actual = *ptr;
    if (actual == expected)
        *ptr = new;
    return actual;
}
```

```
void lock(lock_t *lock) {
    while (CompareAndSwap(&lock->flag, 0, 1) == 1)
        ; // spin
}
```

5. **Load-Linked and Store-Conditional**:

```
int LoadLinked(int *ptr) {
    return *ptr;
}

int StoreConditional(int *ptr, int value) {
    if (no one has updated *ptr since the LoadLinked to this address) {
        *ptr = value;
        return 1; // success!
    } else {
        return 0; // failed to update
    }
}
```

```
void lock(lock_t *lock) {
    while (1) {
        while (LoadLinked(&lock->flag) == 1)
            ; // spin until it's zero
        if (StoreConditional(&lock->flag, 1) == 1)
            return; // if set-it-to-1 was a success: all done
                    // otherwise: try it all over again
    }
}

void unlock(lock_t *lock) {
    lock->flag = 0;
}
```

One thread calls **lock()** and executes the **load-linked**, **returning 0** as the lock is not held. It is interrupted and another thread enters the lock code. The **second thread** to attempt the **store-conditional** will **fail** (because the other thread updated the value of flag between its load-linked and storeconditional).

6. **Fetch-And-Add:**

```
int FetchAndAdd(int *ptr) {
    int old = *ptr;
    *ptr = old + 1;
    return old;
}
```

```
typedef struct __lock_t {
    int ticket;
    int turn;
} lock_t;

void lock_init(lock_t *lock) {
    lock->ticket = 0;
    lock->turn   = 0;
}

void lock(lock_t *lock) {
    int myturn = FetchAndAdd(&lock->ticket);
    while (lock->turn != myturn)
        ; // spin
 }

void unlock(lock_t *lock) {
    lock->turn = lock->turn + 1;
}
```

When a thread wishes to acquire a **lock**, it first does an atomic **fetch-and-add** on the **ticket** value; that value is now considered this thread's "**turn**". When **myturn equals turn** for a given thread, it is that thread's turn to **enter** the critical section. **Unlock increments** the turn such that the next waiting thread can now enter the critical section.

Once a thread is assigned its ticket value, it will be scheduled at some point in the future.

7. **Yielding approach:**

```
void init() {
    flag = 0;
}

void lock() {
    while (TestAndSet(&flag, 1) == 1)
        yield(); // give up the CPU
}

void unlock() {
    flag = 0;
}
```

Problem: performance.

8. **Sleeping Instead Of Spinning**:

```
typedef struct __lock_t {
    int flag;
    int guard;
    queue_t *q;
} lock_t;

void lock_init(lock_t *m) {
    m->flag  = 0;
    m->guard = 0;
    queue_init(m->q);
}

void lock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (m->flag == 0) {
        m->flag = 1; // lock is acquired
        m->guard = 0;
    } else {
        queue_add(m->q, gettid());
        m->guard = 0;
        park();
    }
}

void unlock(lock_t *m) {
    while (TestAndSet(&m->guard, 1) == 1)
        ; //acquire guard lock by spinning
    if (queue_empty(m->q))
        m->flag = 0; // let go of lock; no one wants it
    else
        unpark(queue_remove(m->q)); // hold lock (for next thread!)
    m->guard = 0;
}
```

Park() to put a calling thread to sleep, and **unpark(threadID)** to wake a particular thread as designated by threadID. **Guard** is used, basically as a spin-lock around the flag and queue manipulations the lock is using.

When a thread can not acquire the lock we add ourselves to a queue, set guard to 0, and yield the CPU.

Wakeup/waiting race: a thread will be about to park, assuming that it should sleep until the lock is no longer held. If a switch happens at that time to a thread holding the lock and that thread then released the lock, the first thread would then sleep forever (potentially).

```
        queue_add(m->q, gettid());
        setpark(); // new code
        m->guard = 0;
```

Setpark(): a thread can indicate it is about to park.

9. **Two-Phase Locks**: in the **first phase**, the lock **spins** for a while, hoping that it can acquire the lock. **Second phase** is entered, where the caller is put to **sleep**, and only woken up when the lock becomes free later.

   Futex:

```
void mutex_lock (int *mutex) {
  int v;
  /* Bit 31 was clear, we got the mutex (this is the fastpath)  */
  if (atomic_bit_test_set (mutex, 31) == 0)
    return;
  atomic_increment (mutex);
  while (1) {
      if (atomic_bit_test_set (mutex, 31) == 0) {
          atomic_decrement (mutex);
          return;
      }
      /* We have to wait now. First make sure the futex value
         we are monitoring is truly negative (i.e. locked). */
      v = *mutex;
      if (v >= 0)
        continue;
      futex_wait (mutex, v);
  }
}

void mutex_unlock (int *mutex) {
  /* Adding 0x80000000 to the counter results in 0 if and only if
    there are not other interested threads */
  if (atomic_add_zero (mutex, 0x80000000))
    return;

  /* There are other threads waiting for this mutex,
     wake one of them up.  */
  futex_wake (mutex);
}
```

Condition Variables

<u>Parent Waiting For Child: Spin-based Approach</u>

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Definition and Routines

A **condition variable** is an explicit queue that threads can put themselves on when some state of execution (i.e., some condition) is not as desired; some other thread, when it changes said state, can then wake one (or more) of those **waiting** threads and thus allow them to continue (by **signaling** on the condition).

```
int done  = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

The responsibility of **wait()** is to release the lock and put the calling thread to sleep (atomically). When the thread wakes up (after some other thread has signaled it), it must re-acquire the lock before returning to the caller.

It is likely simplest and best to hold the lock while signaling when using condition variables.

The Producer/Consumer (Bounded Buffer) Problem

Producers generate data items and place them in a buffer; **consumers** grab said items from the buffer and consume them in some way.

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

```
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    for (i = 0; i < loops; i++) {
        put(i);
    }
}

void *consumer(void *arg) {
    int i;
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
}
```

Bounded buffer is a **shared resource.**

```
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // p1
        if (count == 1)                       // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                               // p4
        Pthread_cond_signal(&cond);           // p5
        Pthread_mutex_unlock(&mutex);         // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);           // c1
        if (count == 0)                       // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                      // c4
        Pthread_cond_signal(&cond);           // c5
        Pthread_mutex_unlock(&mutex);         // c6
        printf("%d\n", tmp);
    }
}
```

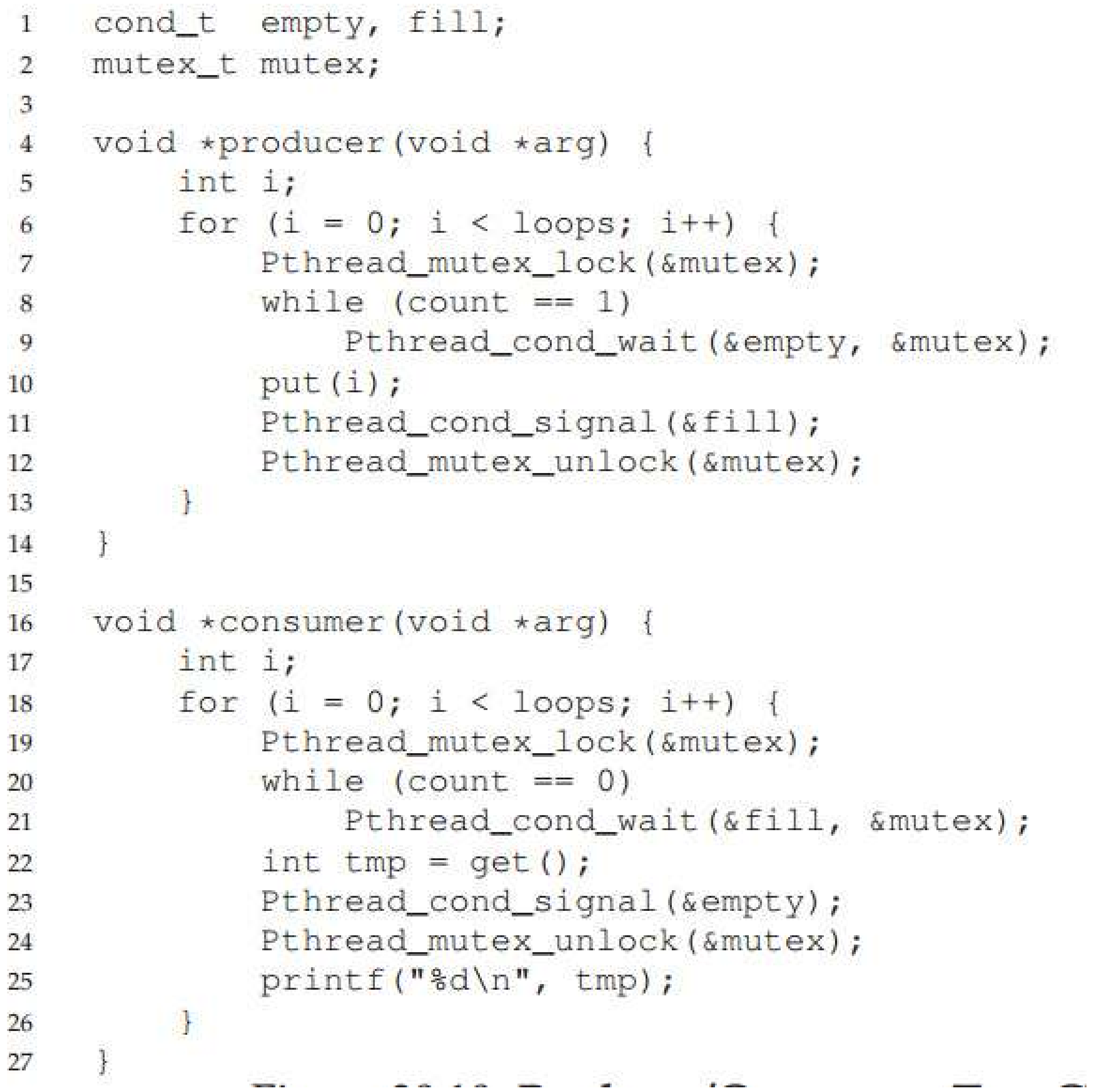
T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

```
cond_t  cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // p1
        while (count == 1)                      // p2
            Pthread_cond_wait(&cond, &mutex); // p3
        put(i);                                 // p4
        Pthread_cond_signal(&cond);             // p5
        Pthread_mutex_unlock(&mutex);           // p6
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);             // c1
        while (count == 0)                      // c2
            Pthread_cond_wait(&cond, &mutex); // c3
        int tmp = get();                        // c4
        Pthread_cond_signal(&cond);             // c5
        Pthread_mutex_unlock(&mutex);           // c6
        printf("%d\n", tmp);
    }
```

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...



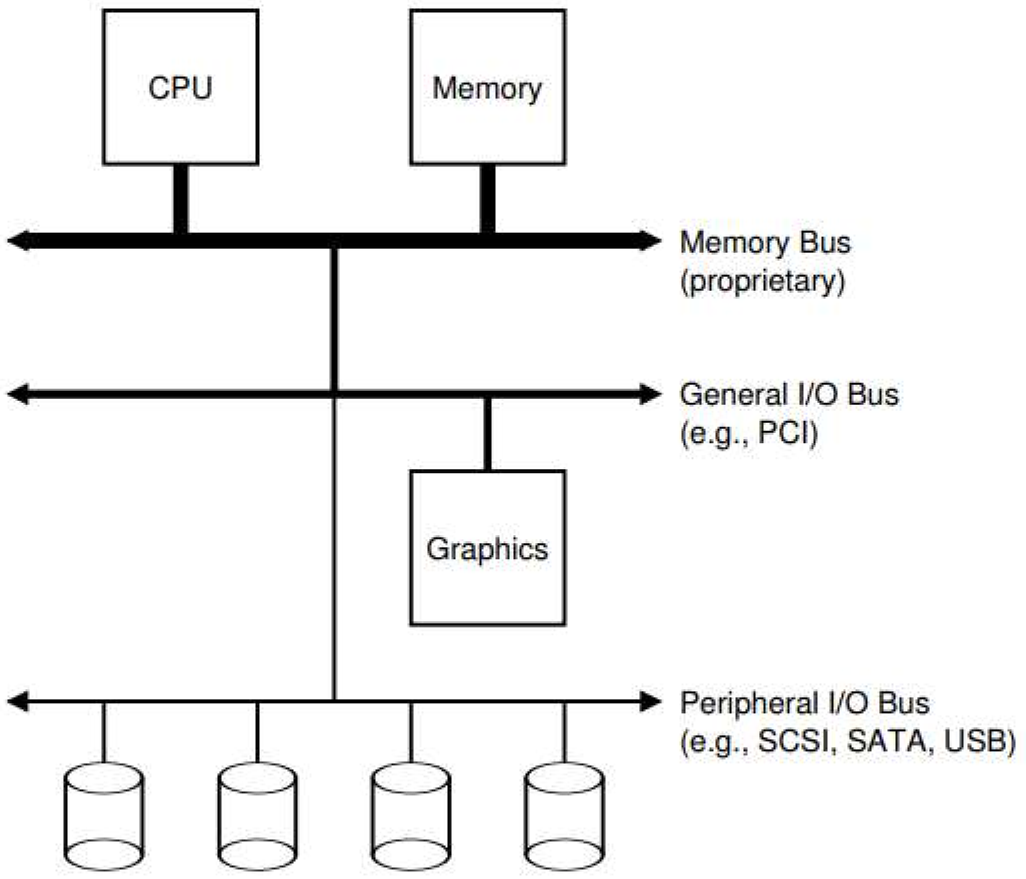
```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

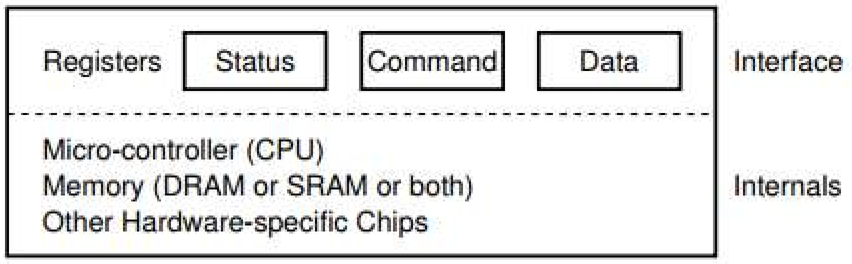
I/O Device

System Architecture



A Canonical Device

A device has two important components. The first is the **hardware interface** it presents to the rest of the system. That allows the system software to control its operation. Thus, all devices have some specified interface and protocol for typical interaction. The second part of any device is its **internal structure**. This part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system.



The Canonical Protocol

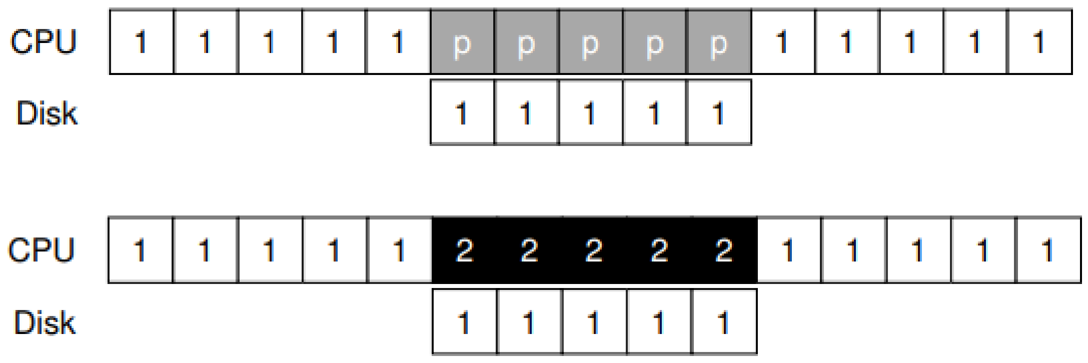
The (simplified) device interface is comprised of three registers: a status register, which can be read to see the current **status** of the device; a **command** register, to tell the device to perform a certain task; and a **data** register to pass data to the device, or get data from the device.

```
While (STATUS == BUSY)
    ; // wait until device is not busy
Write data to DATA register
Write command to COMMAND register
    (Doing so starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

The OS waits until the device is ready to receive a command by repeatedly reading the status register; we call this **polling** the device. The OS sends some data down to the data register, when the main CPU is involved with the data movement, we refer to it as **programmed I/O (PIO)**. The OS writes a command to the command register. Finally, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished.

Lowering CPU Overhead With Interrupts

Instead of polling the device repeatedly, the OS can issue a request, put the calling process to sleep, and context switch to another task. When the device is finally finished with the operation, it will raise a **hardware interrupt**, causing the CPU to jump into the OS at a pre-determined **interrupt service routine (ISR)** or more simply an **interrupt handler**.



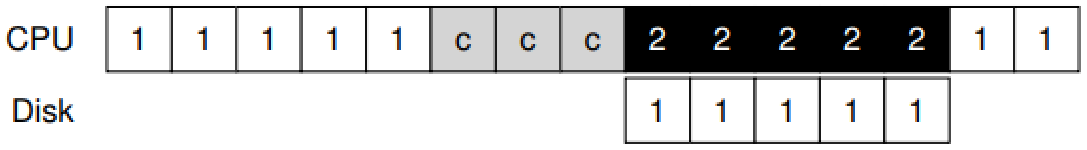
If the speed of the device is not known, or sometimes fast and sometimes slow, it may be best to use a **hybrid** that polls for a little while and then, if the device is not yet finished, uses interrupts.

Another reason not to use interrupts arises in networks. When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to **livelock**, that is, find itself only processing interrupts and never allowing a user-level process to run and actually service the requests.

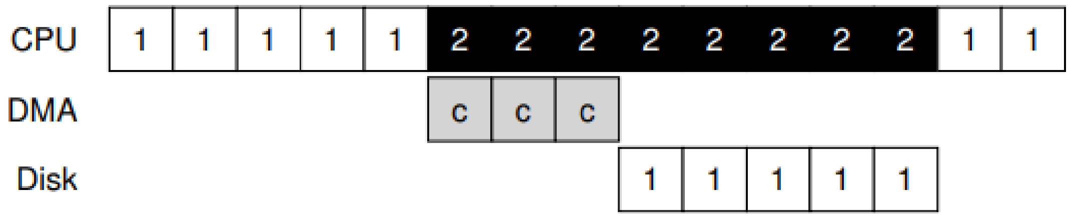
Coalescing: a device which needs to raise an interrupt first waits for a bit before delivering the interrupt to the CPU. While waiting, other requests may soon complete, and thus multiple interrupts can be coalesced into a single interrupt delivery, thus lowering the overhead of interrupt processing.

More Efficient Data Movement With DMA

During the I/O, the CPU must copy the data from memory to the device explicitly, one word at a time.



Direct Memory Access (DMA): To transfer data to the device, the OS would program the DMA engine by telling it where the data lives in memory, how much data to copy, and which device to send it to. At that point, the OS is done with the transfer and can proceed with other work. When the DMA is complete, the DMA controller raises an interrupt, and the OS thus knows the transfer is complete.



Methods Of Device Interaction

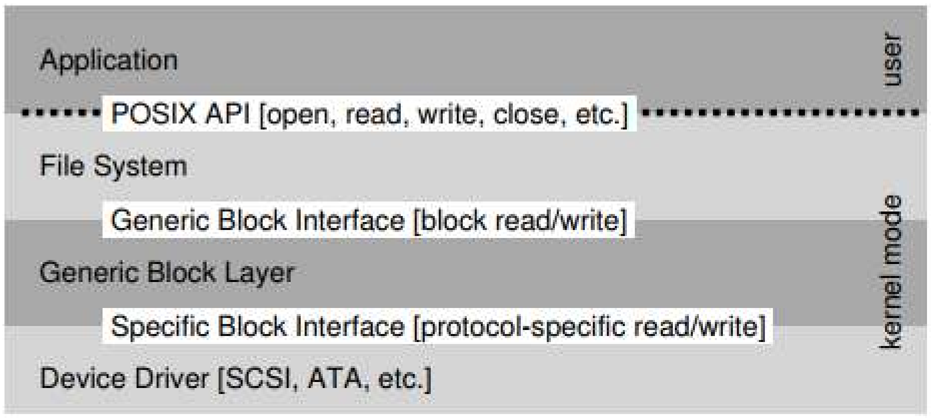
The first, oldest method (used by IBM mainframes for many years) is to have explicit I/O instructions (specify a way for the OS to send data to specific device registers). Such instructions are usually privileged.

The second method to interact with devices is known as **memory-mapped I/O**. With this approach, the hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues a load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

Fitting Into The OS: The Device Driver

How to fit devices, each of which have very specific interfaces, into the OS, which we would like to keep as general as possible.

The problem is solved through abstraction. At the lowest level, a piece of software in the OS must know in detail how a device works. We call this piece of software a **device driver**, and any specifics of device interaction are encapsulated within.


Application
user
POSIX API [open, read, write, close, etc.]
File System
Generic Block Interface [block read/write]
Generic Block Layer
kernel mode
Specific Block Interface [protocol-specific read/write]
Device Driver [SCSI, ATA, etc.]

If there is a device that has many special capabilities, but has to present a generic interface to the rest of the kernel, those special capabilities will go unused.

Hard Disk Drives

The Interface

We can view the disk as an array of sectors; 0 to n − 1 is thus the **address space** of the drive.

A single 512-byte write is atomic; thus, only a portion of a larger write may complete (**torn write**). Accessing blocks in a contiguous chunk (i.e., a sequential read or write) is the fastest access mode, and usually much faster than any more random access pattern.

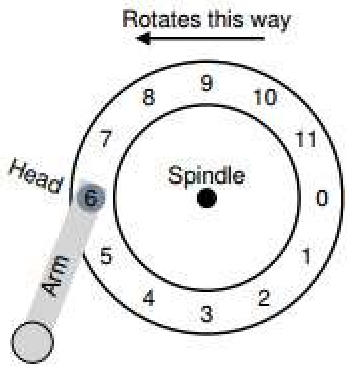
Platter, a circular hard surface on which data is stored persistently by inducing magnetic changes to it. A disk may have one or more platters; each platter has 2 sides, each of which is called a **surface.**

These platters are usually made of some hard material (such as aluminum), and then coated with a thin magnetic layer that enables the drive to persistently store bits even when the drive is powered off.

The platters are all bound together around the **spindle**, which is connected to a motor that spins the platters around at a constant (fixed) rate.

Data is encoded on each surface in concentric circles of sectors; we call one such concentric circle a **track**.

The process of reading and writing is accomplished by the **disk head**; there is one such head per surface of the drive. The disk head is attached to a single **disk arm**, which moves across the surface to position the head over the desired track.

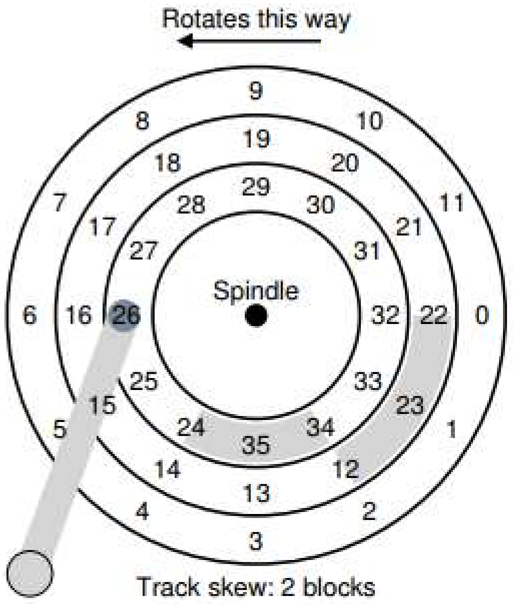


The **seek** has many phases: first an **acceleration** phase as the disk arm gets moving; **then coasting** as the arm is moving at full speed, then **deceleration** as the arm slows down; finally settling as the head is carefully positioned over the correct track.

The Rotational Delay: Time to wait for the desired sector to rotate under the disk head.

First a seek, then waiting for the rotational delay, and finally the transfer.

Many drives employ some kind of track skew to make sure that sequential reads can be properly serviced even when crossing track boundaries.



Multi-zoned disk drives, where the disk is organized into multiple zones, and where a zone is consecutive set of tracks on a surface. Each zone has the same number of sectors per track, and outer zones have more sectors than inner zones.

The **cache (track buffer**) is just some small amount of memory (usually around 8 or 16 MB) which the drive can use to hold data read from or written to the disk.

Should it acknowledge the write has completed when it has put the data in its memory, or after the write has actually been written to disk? The former is called **write back** caching (or sometimes **immediate reporting**), and the latter **write through**.

Files and Directories

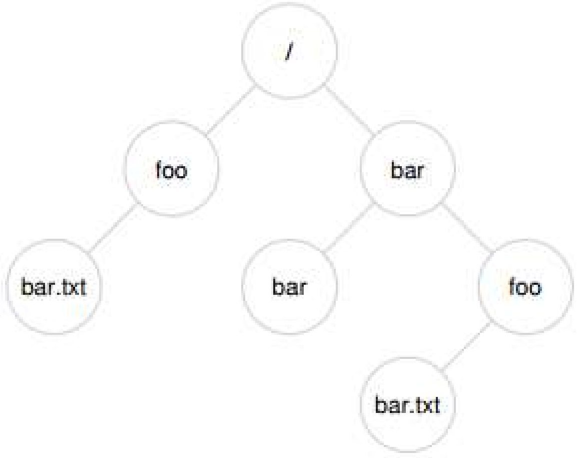
Abstraction: **persistent storage**. A persistent-storage device, such as a classic hard disk drive or a more modern solid-state storage device, stores information permanently (or at least, for a long time)

Files and Directories

Abstraction: A **file** is simply a linear array of bytes, each of which you can read or write. Each file has some kind of **low-level name**, usually a number of some kind; often, the user is not aware of this name (**inode number**).

Abstraction: A **directory,** like a file, also has a low-level name (i.e., an inode number), but its contents are quite specific: it contains a list of (user-readable name, low-level name) pairs.

By placing directories within other directories, users are able to build an arbitrary **directory tree** (or **directory hierarchy**), under which all files and directories are stored.



The directory hierarchy starts at **a root directory (**in UNIX-based systems, the root directory is simply referred to as /) and uses some kind of separator to name subsequent sub-directories until the desired file or directory is named. For example, if a user created a directory foo in the root directory /, and then created a file bar.txt in the directory foo, we could refer to the file by its **absolute pathname**, which in this case would be /foo/bar.txt.

The second part of the file name is usually used to indicate the **type** of the file (**convention**).

The File System Interface

Creating Files

```
int fd = open("foo", O_CREAT|O_WRONLY|O_TRUNC, S_IRUSR|S_IWUSR);
```

One important aspect of open() is what it returns: a **file descriptor**. A file descriptor is just an integer, private per process, and is used in UNIX systems to access files; thus, once a file is opened, you use the file descriptor to read or write the file, assuming you have permission to do so. A file descriptor is a **capability**, i.e., an opaque handle that gives you the power to perform certain operations

Reading and Writing Files

The first argument to **read()** is the file descriptor, thus telling the file system which file to read. The second argument points to a buffer where the result of the read() will be placed. The third argument is the size of the buffer. If the call to read() returns successfully: the number of bytes it read is returned. Otherwise, zero. Writing a file is accomplished via a similar set of steps with the **write()** system call.

Reading from some random offsets within the document:

```
    off_t lseek(int fildes, off_t offset, int whence);

If whence is SEEK_SET, the offset is set to offset bytes.
If whence is SEEK_CUR, the offset is set to its current
location plus offset bytes.
If whence is SEEK_END, the offset is set to the size of
the file plus offset bytes.
```

Part of the abstraction of an open file is that it has a current offset, which is updated in one of two ways: a read or write of N bytes takes place, N is added to the current offset; the second is explicitly with **lseek**.

When a process calls **fsync()** for a particular file descriptor, the file system responds by forcing all dirty (i.e., not yet written) data to disk, for the file referred to by the specified file descriptor. The fsync() routine returns once all of these writes are complete.

Renaming Files

rename() call is implemented as an **atomic** call with respect to system crashes.

```
int fd = open("foo.txt.tmp", O_WRONLY|O_CREAT|O_TRUNC,
              S_IRUSR|S_IWUSR);
write(fd, buffer, size); // write out new version of file
fsync(fd);
close(fd);
rename("foo.txt.tmp", "foo.txt");
```

Getting Information About Files

To see the metadata for a certain file, we can use the **stat()** or **fstat()** system calls.

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* protection */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for filesystem I/O */
    blkcnt_t  st_blocks;  /* number of blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

As it turns out, each file system usually keeps this type of information in a structure called an inode

Removing Files

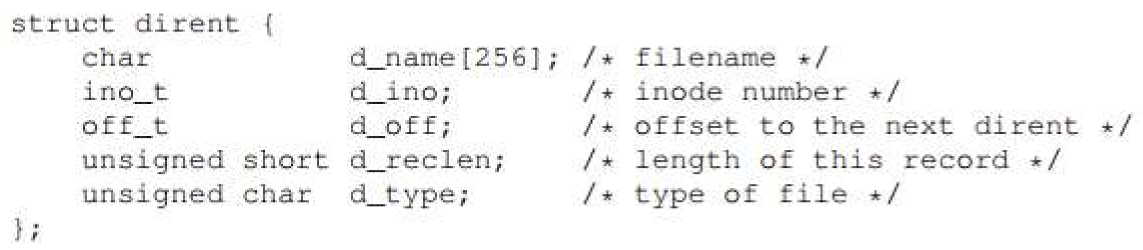
unlink() just takes the name of the file to be removed, and returns zero upon success.

Making Directories

To create a directory, a single system call, **mkdir()**, is available. An empty directory has two entries: one entry that refers to itself, and one entry that refers to its parent. The former is referred to as the **"."** (dot) directory, and the latter as **".."** (dot-dot).

Reading Directories

```
int main(int argc, char *argv[]) {
    DIR *dp = opendir(".");
    assert(dp != NULL);
    struct dirent *d;
    while ((d = readdir(dp)) != NULL) {
        printf("%lu %s\n", (unsigned long) d->d_ino, d->d_name);
    }
    closedir(dp);
    return 0;
}
```



```
struct dirent {
    char           d_name[256]; /* filename */
    ino_t          d_ino;       /* inode number */
    off_t          d_off;       /* offset to the next dirent */
    unsigned short d_reclen;    /* length of this record */
    unsigned char  d_type;      /* type of file */
};
```

Deleting Directories

rmdir() has the requirement that the directory be empty before it is deleted.

Hard Links

The **link()** system call takes two arguments, an old pathname and a new one; when you "link" a new file name to an old one, you essentially create another way to refer to the same file. The way link works is that it simply creates another name in the directory you are creating the link to, and refers it to the same inode number of the original file.

When you create a file, you are really doing two things. First, you are making a structure (the **inode**) that will track virtually all relevant information about the file, including its size, where its blocks are on disk, and so forth. Second, you are linking a human-readable name to that file, and putting that link into a directory.

The **reference count** (sometimes called the **link count**) allows the file system to track how many different file names have been linked to this particular inode.

Symbolic Links

Hard links are somewhat limited: you can't create one to a directory (for fear that you will create a cycle in the directory tree); you can't hard link to files in other disk partitions (because inode numbers are only unique within a particular file system, not across file systems); etc. Thus, a new type of link called the **symbolic link** was created.

A symbolic link is actually a file itself, of a different type. The way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file.

Dangling reference: removing the original file named file causes the link to point to a pathname that no longer exists.

Making and Mounting a File System

To make a file system, **mkfs**. The idea is as follows: give the tool, as input, a device a file system type (e.g., ext3), and it simply writes an empty file system, starting with a root directory, onto that disk partition.

It needs to be made accessible within the uniform file-system tree: **mount()**. Takes an existing directory as a target **mount point** and essentially paste a new file system onto the directory tree at that point.

File System Implementation

Vsfs (the Very Simple File System).

Aspects of a file system

The first is the **data structures** of the file system. The second aspect of a file system is its **access methods**.

Overall Organization

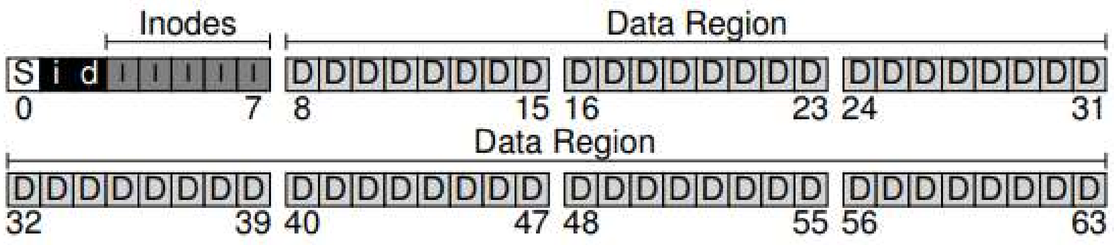
The first thing we'll need to do is divide the disk into blocks. Most of the space in any file system is user data (**data region**).

The file system has to track information about each file (**metadata**). To store this information, file systems usually have a structure called an **inode**.

Inode table: simply holds an array of on-disk inodes.

One primary component that is needed is some way to track whether inodes or data blocks are free or allocated (**allocation structures**): **bitmap**, one for the data region (the **data bitmap**), and one for the inode table (the **inode bitmap**). Each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1).

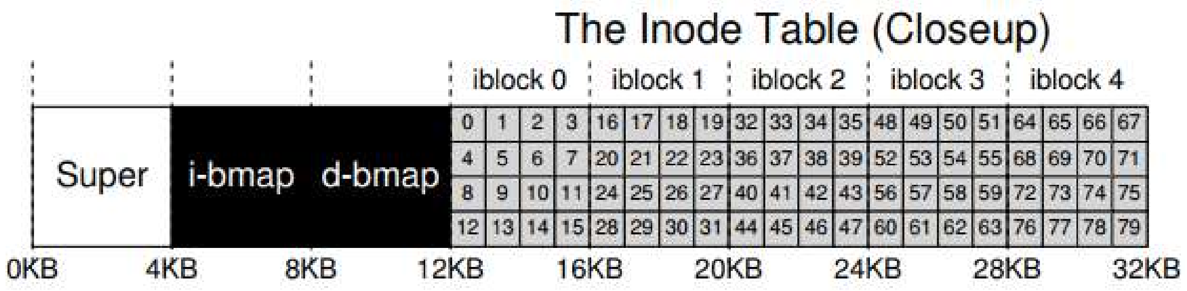
The **superblock** contains information about this particular file system, including, for example, how many inodes and data blocks are in the file system, where the inode table begins, and so forth. When mounting a file system, the operating system will read the superblock first, to initialize various parameters, and then attach the volume to the file-system tree.



File Organization: The Inode

The name inode is short for **index node**, used because these nodes were originally arranged in an array, and the array indexed into when accessing a particular inode.

Each inode is implicitly referred to by a number (called the **inumber)**. Given an i-number, you should directly be able to calculate where on the disk the corresponding inode is located.



```
blk    = (inumber * sizeof(inode_t)) / blockSize;
sector = ((blk * blockSize) + inodeStartAddr) / sectorSize;
```

Size	Name	What is this inode field for?
2	mode	can this file be read/written/executed?
2	uid	who owns this file?
4	size	how many bytes are in this file?
4	time	what time was this file last accessed?
4	ctime	what time was this file created?
4	mtime	what time was this file last modified?
4	dtime	what time was this inode deleted?
2	gid	which group does this file belong to?
2	links_count	how many hard links are there to this file?
4	blocks	how many blocks have been allocated to this file?
4	flags	how should ext2 use this inode?
4	osd1	an OS-dependent field
60	block	a set of disk pointers (15 total)
4	generation	file version (used by NFS)
4	file_acl	a new permissions model beyond mode bits
4	dir_acl	called access control lists

One of the most important decisions in the design of the inode is how it refers to where data blocks are. One simple approach would be to have one or more direct pointers (disk addresses) inside the inode.

The Multi-Level Index

To support bigger files, **indirect pointers**: Instead of pointing to a block that contains user data, it points to a block that contains more pointers, each of which point to user data.
An **extent** is simply a disk pointer plus a length (in blocks).
Double indirect pointer: this pointer refers to a block that contains pointers to indirect blocks, each of which contain pointers to data blocks.
Multi-level index approach to pointing to file blocks.

File allocation table, or **FAT** file system: to make **linked allocation** work better, some systems will keep an in-memory table of link information, instead of storing the next pointers with the data blocks themselves.

Directory Organization

A directory basically just contains a list of (entry name, inode number) pairs. Each entry has an inode number, record length (the total bytes for the name plus any left over space), string length (the actual length of the name), and finally the name of the entry.

A directory has an inode, somewhere in the inode table (with the type field of the inode marked as "directory" instead of "regular file").

Free Space Management

Some early file systems used **free lists**, where a single pointer in the super block was kept to point to the first free block. Some form of a **B-tree** to compactly represent which chunks of the disk are free.

The file system will thus search through the **bitmap** for an inode that is free, and allocate it to the file; the file system will have to mark the inode as used (with a 1) and eventually update the on-disk bitmap with the correct information.

Pre-allocation policies such as looking for contiguous blocks of free memory for improving performance.

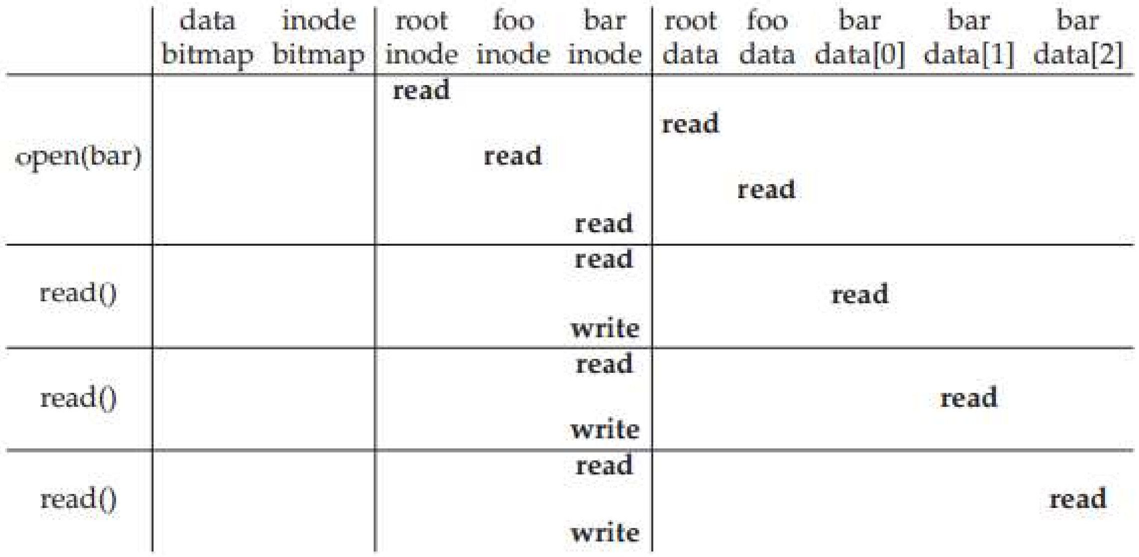
Access Paths: Reading and Writing

Reading A File From Disk:

To do so, the file system must be able to find the inode, but all it has right now is the full pathname. The file system must traverse the pathname and thus locate the desired inode. All traversals begin at the root of the file system, in the root directory which is simply called /. Thus, the first thing the FS will read from disk is the inode of the root directory. But where is this inode? To find an inode, we must know its i-number. The root inode number must be "well known" (2 in UNIX).

The next step is to recursively traverse the pathname until the desired inode is found. The FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-file table, and returns it to the user. Once open, the program can then issue a read() system call to read from the file. At some point, the file will be closed: the file descriptor should be deallocated.

Also note that the amount of I/O generated by the open is proportional to the length of the pathname.

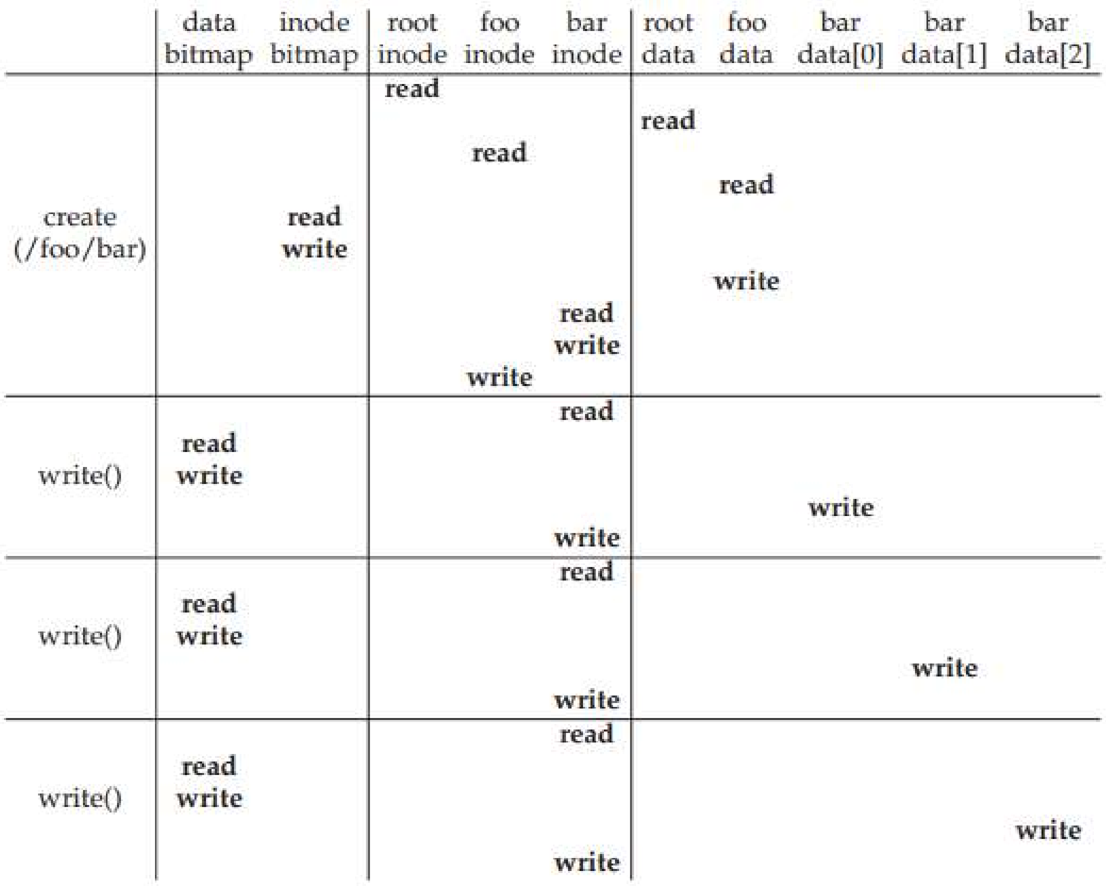


	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
			read							
						read				
open(bar)				**read**						
							read			
					read					
					read					
read()								**read**		
					write					
					read					
read()									**read**	
					write					
					read					
read()										**read**
					write					

Writing to Disk

First, the file must be opened. Then, the application can issue write() calls to update the file with new contents. Finally, the file is closed.

Unlike reading, writing to the file may also **allocate** a block. Each write to a file logically generates five I/Os: one to read the data bitmap (which is then updated to mark the newly-allocated block as used), one to write the bitmap (to reflect its new state to disk), two more to read and then write the inode (which is updated with the new block's location), and finally one to write the actual block itself.
To create a file: one read to the inode bitmap (to find a free inode), one write to the inode bitmap (to mark it allocated), one write to the new inode itself (to initialize it), one to the data of the directory (to link the high-level name of the file to its inode number), and one read and write to the directory inode to update it.



	data bitmap	inode bitmap	root inode	foo inode	bar inode	root data	foo data	bar data[0]	bar data[1]	bar data[2]
create (/foo/bar)			read							
						read				
				read						
							read			
		read								
		write								
							write			
					read					
					write					
				write						
write()					read					
	read									
	write									
								write		
					write					
write()					read					
	read									
	write									
									write	
					write					
write()					read					
	read									
	write									
										write
					write					

Caching and Buffering

Early file systems thus introduced a **fixed-size cache** to hold popular blocks. This **static partitioning** of memory, however, can be wasteful.

Modern systems, in contrast, employ a **dynamic partitioning** approach. Specifically, many modern operating systems integrate virtual memory pages and file system pages into a **unified page cache**.

Write buffering (as it is sometimes called) certainly has a number of performance benefits. First, by delaying writes, the file system can **batch** some updates into a smaller set of I/Os. Second, the system can then **schedule** the subsequent I/Os and thus increase performance. Finally, some writes are **avoided** altogether by delaying them.

Some applications (such as databases), to avoid unexpected data loss due to write buffering, they simply force writes to disk, by calling fsync(), by using **direct I/O** interfaces that work around the cache, or by using the **raw disk** interface and avoiding the file system altogether