Textbook Notes

Saturday, June 29, 2019 5:43 PM

Chapter 2. Intro to OS

Von Neumann's model of computing: Billions of instructions every second get executed. The processor **fetches** an instruction from memory, **decodes** it and **executes** it. It then moves on to the next instuction, and so on, until the program is complete.

Operating System (OS): a body of software that is responsible for making it easy to run programs, allowing programs to share memory, enable programs to interact with devices, etc.

2.1 Virtualizing the CPU
Virtualization: the process of the OS taking a physical resource (such a processor, memory, or disk) and transforming it into a more general, powerful, and easy-to-use virtual form of itself. *That is why we sometimes call the OS a virtual machine.*

Because virtualization allows many programs to run, to concurrently access their own instructions and data, and many programs to access devices, the OS is also known as a **resource manager.**

The **policy** of the OS decides between two programs who should run.

2.2 Virtualizing Memory
Physical memory: an array of bytes. To **read** memory, one must specify an **address** to be able to access the data stored there; to **write** (or **update**) memory, one must also specify the data to be written to the given address.

2.3 Concurrency
Programs taking place at the same time (multi-threading)
Atomically: taking place all at once

2.4 Persistence
Storing data *persistently* since DRAM is volatile and if system crashes, any data in it is lost.
We use some kind of **input/out** or I/O device, e.g. a hard drive or solid-state drive (SSD)
Steps for the OS to write to disk: first figure out where on disk this new data will reside, and then keeping track of it in various structures the file system maintains

2.5 Design Goals
We want to provide high performance and to minimze the overheads of the OS. E.g. of overheads: extra time (more instructions) and extra space (in memory in on disk)

Protection: Having several programs running at the same time, we want to make sure that malicious or accidental bad behavior harming one program doesn't harm others (and certainly not the OS because that affects ALL programs). That is why we need **isolation,** the key to protection and what underlies much of what an OS must do.

Other goals: **energy-efficiency, security** (an extension of protection), **mobility**

Chapter 4. The Abstraction: The Process

Process: informally, a running program.
Program itself is useless, but OS takes it and transforms it into something useful

virtualization: the illusion of the OS providing a nearly-endless supply of CPUs
time sharing: allows users to run as many concurrent processes as they would like, with performance being the cost.
mechanisms: low-level machinery; mechanisms are low-level methods or protocols that implement a needed piece of functionality.
context switch: the ability of the OS to stop running one program and start running another on a given CPU.
policies: the intelligence of the OS, policies are algorithms for making some kind of decision within the OS (e.g. given a number of possible programs to run on a CPU, which program should the OS run?)
scheduling policy: answers the above question. It uses historical information, workload knowledge, and performance metrics.

process: the abstraction provided by the OS of a running program
machine state: what a program can read or update when it is running i.e. at any time, what parts of the machine are important to execution of this program.
The most obvious component of machine state is the **memory/address state** of the instructions. Next, there are the **registers**. Some special registers are the **program counter (PC)** (sometimes called the **instruction pointer** or **IP**) telling us which instruction of the program will execute next. Similarly a **stack pointer** and associated **frame pointer** are used to manage the stack for function parameters, local variables, and return addresses. Finally, programs also use storage devicers like I/O information (e.g. including a list of the files the process currently has open.)

Process API
all the following are available on any modern OS:

- **Create:** creating a new process must be included. Examples are typing a command into the shell, or double-clicking on an app icon. The OS is invoked to create a new process to run the program you have indicated.
- **Destroy:** there also needs to be an interface to destroy processes forcefully. Many processes run and end on their own, but when they don't, you need to be able to do it manually.
- **Wait:** sometimes it's useful to wait for a process to stop running.
- **Miscallaneous Control:** other controls are possible. An example is a method that suspends a process and then resumes it later.
- **Status:** this is to get status info about a process, such as long it has to run for, or what state it is in.

Process Creation
first thing the OS must do is to **load** its code and any static data into memory, into the address space of the process. Programs initially reside on **disk** or **flash-based SSDs** in some kind of **executable format**. The OS reads those bytes and places them into memory somewhere.
Old OS's perform this process **eagerly** (all at once before running the program); modern OS's perform the process **lazily** (by loading pieces of code or data only as they are needed during program execution.
OS also allocates memory for the program's **run-time stack** (or just **stack**). C programs use the stack for local variables, function parameters, and return addresses. The OS will initialize the stack with parameters to main() function, i.e. argc and *argv[ ].
Some memory is also allocated for the **heap**. In C programs, the program's **heap** is used for explicitly requested dynamically-allocated data; programs request such space by calling the malloc() and free it explicitly by calling free().

I/O
In UNIX systems, each process by default has three open **file descriptors**: standard input, output, and error.

Process States
A process can have a number of states.

- **running:** process is running on a processor. This means it is executing instructions.
- **ready:** process is ready to run but for some reason the OS has chosen not to run it at this given moment.
- **blocked**: process has performed some kind of operation that makes it not ready to run until some other event takes place.

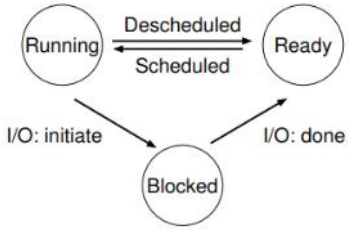


Figure 4.2: **Process: State Transitions**

Moving from ready to running means the process has been **scheduled**; being moved from running to ready means the process has been **descheduled**. Once a program has been blocked, it stays blocked until some event occurs to make it ready.

The OS tracks important things. 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 registers (i.e. placing their values back into the actual physical registers), the OS can resume running the process.

ASIDE: KEY PROCESS TERMS

- The **process** is the major OS abstraction of a running program. At any point in time, the process can be described by its state: the contents of memory in its **address space**, the contents of CPU registers (including the **program counter** and **stack pointer**, among others), and information about I/O (such as open files which can be read or written).
- The **process API** consists of calls programs can make related to processes. Typically, this includes creation, destruction, and other useful calls.
- Processes exist in one of many different **process states**, including running, ready to run, and blocked. Different events (e.g., getting scheduled or descheduled, or waiting for an I/O to complete) transition a process from one of these states to the other.
- A **process list** contains information about all processes in the system. Each entry is found in what is sometimes called a **process control block** (**PCB**), which is really just a structure that contains information about a specific process.

Chapter 5. Interlude: Process API

This chapter is about process creation in UNIX systems. In UNIX, we use system calls **fork()** and **exec()** to create a new process. You can also use **wait()** to wait for a process to complete.

Fork()

process identifier (PID): used to name the process if one wants to do something with the process (e.g. stop it from running.)
fork(): used to create a new process. It makes an (almost) exact copy of the calling process. To the OS, it will look like there are two copies of the process running, both of which return from the fork() system call.
the **child** isn't an exact copy of the **parent.** Specifically, although it has its own copy of the address space (own private memory), its own registers, its own PC, and so forth, the value it returns to the caller of the **fork()** is different. Parent receives the PID of the child, while the child receives a return code of zero. This is useful as it allows use to write the code that handles the two different cases.
The **forking()** process is **non-determinstic** (i.e., the **scheduler** is complex and we cannot make strong assumptions about which process (the child or the parent) it will run first), which leads to problems with **multi-threaded program** (**concurrency**).

The child and parent have different data segments

copy-on-write: if either the parent or child writes to the data, make a copy and let the process write the copy. The other process keeps the original

http://man7.org/linux/man-pages/man2/fork.2.html

Wait()

a parent process can **wait()** to delay its execution until the child finishes executing. Adding **wait()** makes the output deterministic as we can trace the order of the processes executions.

Exec()

used when we want to run a program that is different from the calling program. For example, calling **fork()** is only useful when you want to keep running copies of the same program. However, if you want to run a *different* program, you use **exec()**

See the following example:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {          // fork failed; exit
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) { // child (new process)
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: "wc" (word count)
    myargs[1] = strdup("p3.c"); // argument: file to count
    myargs[2] = NULL;           // marks end of array
    execvp(myargs[0], myargs);  // runs word count
    printf("this shouldn't print out");
  } else {                  // parent goes down this path (main)
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

Figure 5.3: **Calling `fork()`, `wait()`, And `exec()` (`p3.c`)**

And its output:

```
prompt> ./p3
hello world (pid:29383)
hello, I am child (pid:29384)
      29     107    1030 p3.c
hello, I am parent of 29384 (rc_wait:29384) (pid:29383)
prompt>
```

After executing the program, the shell figures out where in the file system the exectuable 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.

*^ Study them **WELL.***

Essentially, what the **fork()** does:
Given the name of an executable (e.g. wc), and some arguments (e.g. p3.c), 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 argc of that process. Thus, it does not create a new process; rather, it transforms the currently running

program (formerly p3) into a different running program (wc). After the exec() in the child, it is almost as if p3.c never ran; a successful call to exec() never returns.

Redirection

prompt> wc p3.c > newfile.txt

In the above example, the output is redirected into the output file. The shell accomplishes this task by closing standard output and opening the file newfile.txt. From here on, it sends any output of the program to newfile.txt instead of the screen.

Specifically, UNIX systems start looking for free file descriptors at zero.

Pipes

The output of one process is connected to an in-kernel **pipe** (i.e. a queue) and the input of another process is connected to that same pipe; thus, the output of one process seamlessly is used as input to the next (as long as the chains of commands can be strung together.)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>

int main(int argc, char *argv[]) {
  int rc = fork();
  if (rc < 0) {
    // fork failed
    fprintf(stderr, "fork failed\n");
    exit(1);
  } else if (rc == 0) {
    // child: redirect standard output to a file
    close(STDOUT_FILENO);
    open("./p4.output", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);

    // now exec "wc"...
    char *myargs[3];
    myargs[0] = strdup("wc");   // program: wc (word count)
    myargs[1] = strdup("p4.c"); // arg: file to count
    myargs[2] = NULL;           // mark end of array
    execvp(myargs[0], myargs);  // runs word count
  } else {
    // parent goes down this path (main)
    int rc_wait = wait(NULL);
  }
  return 0;
}
```

Figure 5.4: **All Of The Above With Redirection (p4 . c)**

Process Groups

a process should use the signal() system call to "catch" various signals; doing so ensures that when a particular signal is delivered to a process, it will suspend its normal execution and run a particular piece of code in response to the signal.

User

The user logs in using a password to access system resources. Users can typically only control their own processes and it is the job of the OS to parcel out system resources to each user.

ASIDE: KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID (PID)**.
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent**; the newly created process is called the **child**. As sometimes occurs in real life [J16], the child process is a nearly identical copy of the parent.
- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses `fork()`, `wait()`, and `exec()` to launch user commands; the separation of fork and exec enables features like **input/output redirection**, **pipes**, and other cool features, all without changing anything about the programs being run.
- Process control is available in the form of **signals**, which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user**; the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.

brenwood enterprise
310 477 7209

Chapter 6. Mechanism: Limited Direct Execution

Performance: how can we implement virtualization without adding excessive overhead to the system?
control: how can we run processes efficiently while retaining control over the CPU?

Limited direct execution

"direct execution" just 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 point (i.e. the main() routing or something similar), jumps to it, and starts running the user's code.

OS	Program
Create entry for process list	
Allocate memory for program	
Load program into memory	
Set up stack with argc/argv	
Clear registers	
Execute **call** main()	
	Run main()
	Execute **return** from main
Free memory of process	
Remove from process list	

Figure 6.1: **Direct Execution Protocol (Without Limits)**

user mode: restricted in what it can do.
kernel mode: the OS runs this. The code that runs can do what it likes, including privileged operations such as issuing I/O requests and executing all types of restricted instructions.
system calls: allow the kernel to carefully expose certain key pieces of functionality to user programs, such as accessing the file system, creating and destroying processes, communicating with other processes, and allocation more memory.

To execute a system call, a program executes a special **trap** instruction that simultaneously jumps into the kernel and raises the privilege to kernel mode. Now that we are in the kernel, we can perform whatever privileged operations (if allowed) and do the required work for the calling process. When finished, the OS calls a special **return-from-trap** instruction which returns the user program and reduces the privilege level back to user mode.

For the trap to know which code to run inside the OS, the kernel sets up a **trap table** at boot time. The OS informs the hardware of the locations of **trap handlers**.

system-call number: assigned to each system call. The OS when handling system calls inside the trap handler examines this number, ensures it is valid, and executes the corresponding code. This indirection is for **protection**; the user cannot specify an exact address to jump to, but rather must request a particular service via number.

Switching Between Processes

if a process is running on the CPU, this by definition means the OS is *not* running.

Cooperative approach

The OS trusts the processes of the system to behave reasonably. Processes that run for too long are assumed to periodically give up the CPU so that the OS can decide to run some other task.
This works because most processes transfer control of the CPU to the OS quite frequently by making system calls.

Non-Cooperative approach

A **timer interrupt**, a device that can be programmed to raise an interrupt every a number of miliseconds. When the interrupt is raised, the currently running process is halted, and the pre-configured **interrupt handler** in the OS runs.

Saving and restoring context

A **context switch** is a low-level piece of code where the OS saves a few register values for the currently-executing process (e.g. onto its kernel stack) and restore a few for the soon-to-be-executing process (from its kernel stack).
The OS switches stacks and code from one process and returns another.

During timer interrupts, the user registers of the running process are implicitly saved by the hardware, using the kernel stack of that process.
Switching from process A to B, 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.

ASIDE: KEY CPU VIRTUALIZATION TERMS (MECHANISMS)

- The CPU should support at least two modes of execution: a restricted **user mode** and a privileged (non-restricted) **kernel mode**.
- Typical user applications run in user mode, and use a **system call** to **trap** into the kernel to request operating system services.
- The trap instruction saves register state carefully, changes the hardware status to kernel mode, and jumps into the OS to a pre-specified destination: the **trap table**.
- When the OS finishes servicing a system call, it returns to the user program via another special **return-from-trap** instruction, which reduces privilege and returns control to the instruction after the trap that jumped into the OS.
- The trap tables must be set up by the OS at boot time, and make sure that they cannot be readily modified by user programs. All of this is part of the **limited direct execution** protocol which runs programs efficiently but without loss of OS control.
- Once a program is running, the OS must use hardware mechanisms to ensure the user program does not run forever, namely the **timer interrupt**. This approach is a **non-cooperative** approach to CPU scheduling.
- Sometimes the OS, during a timer interrupt or system call, might wish to switch from running the current process to a different one, a low-level technique known as a **context switch**.

Chapter 7. Scheduling: Introduction

Assumptions about the processes, or **jobs**, that are running on the system:

i. each job runs for the same amount of time
ii. all jobs arrive at the same time
iii. once started, each job runs to completion
iv. all jobs only use the CPU (i.e. they perform no I/O)
v. the run time of each job is known

scheduling metric
a metric is just something that we use to measure something.
turnaround time metric: the time at which the job completes minus the time at which the job arrived in the system.

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

For now, since we assumed that all jobs arrive at the same time,
$T_{arrival} = 0$
Hence, $T_{turnaround} = T_{completion}$

FIFO (First **I**n **F**irst **O**ut)
It works for the above situation. If you have 3 processes that all reach the CPU at approx the same time that all take the same amount of time (all 10 seconds), it works. The average turnaround time is $\frac{10+20+30}{3} = 20$

However,
if their durations isn't the same, it won't work because of the following.

Convoy Effect
A number of relatively-short potential consumers of a resource get queued behind a heavyweight resource consumer.
The approach that fixes the problem is **Shortest Job First (SJF)**

Let's relax another assumption: jobs can now arrive at any time instead of all at once.
We actually run into a roadblock, so we proceed to relax a third assumption: jobs now must **not necessarily** run to completion. They can be paused.

Shortest Time-to-Completion First (STCF)

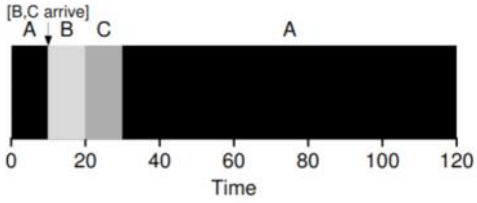


Figure 7.5: **STCF Simple Example**

Even though A reached before B and C, it is paused since it has a larger **Shortest Time-to-Completion First** than B and C, so we preempt A and begin B and C. After we are done with B and C, we continue A.

*also called **Preemptive Shortest Job First (PSJF) scheduler**

Any time a new job enters the system, the STCF scheduler determines which of the remaining jobs (including the new job) has the least time left, and schedules that one.

A New Metric: Response Time

Defined to be the time from when the job arrives in a system to the first time it is scheduled.

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

Round Robin

the basic idea is simple: 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.

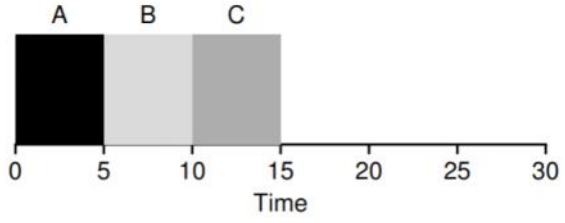


Figure 7.6: **SJF Again (Bad for Response Time)**

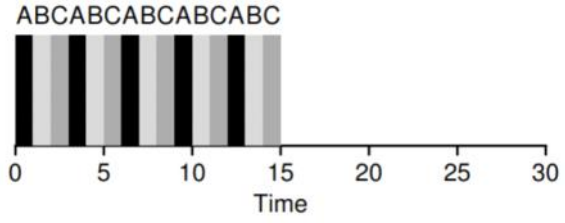


Figure 7.7: **Round Robin (Good For Response Time)**

Note that the length of a time slice must be a multiple of the timer-interrupted period; thus if the timer interrupts every 10 milliseconds, the time slice could be 10, 20, or any other multiple of 10 ms.

The shorter the length of the time slice, the better the performance of RR under the response-time metric. However, making the time slice too

short is problematic: suddenly the cost of context switching will dominate overall performance (aka overhead). It needs to be long enough to **amortize** the cost of switching without making it so long that the system is no longer responsive.

RR is an excellent scheduler if response time is our only metric.
It is also one of the *worst* policies if turnaround time is our metric.
More generally, 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.

We now relax assumption 4. Now, we can perform **I/O**.
When a running process requests I/O, the CPU would be sitting idle and is blocked. Therefore, it is a good idea to make the CPU complete another process while the I/O takes place. Once the I/O request is complete, an interrupt is raised, and the CPU can now decide which process to continue.

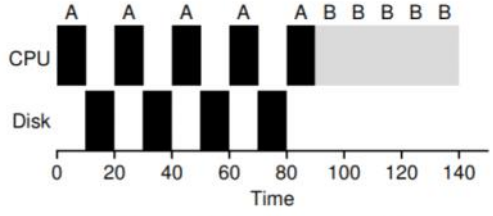


Figure 7.8: **Poor Use Of Resources**

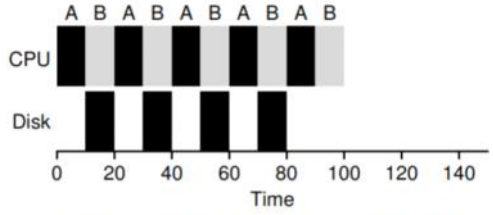


Figure 7.9: **Overlap Allows Better Use Of Resources**

Chapter 8. Scheduling: The Multi-Level Feedback Queue

MLFQ has a number of distinct **queues**, each assigned a different **priority level**.
At any given time, a job that is ready to run is on a single queue.
MLFQ uses priorities to decide which job should run at a given time: a job with higher priority (i.e. a job on a higher queue) is chosen to run.
If two jobs on a given queue have the same priority, we will just use RR scheduling among those jobs.
First two basic rules for MLFQ:

- **Rule 1:** $If\ Priority(A) > Priority(B), A\ runs, B\ doesn't.$
- **Rule 2:** $If\ Priority(A) = Priority(B), A\ \&\ B\ run\ in\ RR.$

Rather than giving a fixed priority to each job, MLFQ *varies* the priority of a job based on its *observed behavior*.
If, for example, a job repeatedly relinquishes the CPU while waiting for input from the keyboard, MLFQ will keep its priority high, as this is how an interactive process might behave. If, instead, a job uses the CPU intensively for long periods of time, MLFQ will reduce its priority.
In this way, MLFQ will try to learn about processes as they run, and thus use the history of the job to predict its future behavior

How To Change Priority

- **Rule 3:** When a job enters the system, it is placed at the highest priority (the topmost queue)
- **Rule 4a:** If a job uses up an entire time slice while running, its priority is *reduced* (i.e. it moves down one queue)
- **Rule 4b:** If a job gives up the CPU before the time slice is up, it stays at the *same* priority level.

Example 2: Along Came a Short Job
Because the algorithm doesn't know whether a job will be a short job or a long-running job, it first assumes it might be a short job, thus giving the job high priority. If it actually is a short job, it will run quickly and complete; if it is not a short job, it will slowly move down the queues, and thus soon prove itself to be a long-running more batch-like process. In this manner, MLFQ approximates SJF.

I/O
According to Rule 4b, if a process gives up the processor before using up its time slice, we keep it at the same level priority. The point of this is if an interactive job, is doing a lot of I/O, for example, it will yield the CPU before its time slice is complete; in such case, we don't wish to penalize the job and thus simply keep it at the same level.

Problems
If there are too many interactive jobs in the system, they will combine to consume all CPU time, and thus long-running jobs will never receive any CPU time. This is called **starving**.
Gaming the scheduler: if a user sneakily writes a program that does BS I/O to stay in higher priority and monopolize the CPU.

To avoid starvation, do periodic **boosts** on all jobs in the system by placing them all in the topmost queue.

- **Rule 5:** After some time period *S*, move all the jobs in the system to the topmost queue.

This solves starvation as now all process are guaranteed to not starve. Also, if a CPU-bound job has become interactive, the scheduler treats it properly once it has received the priority boost.

Now to deal with the user gaming with the scheduler. It is obviously related to rules 4a and 4b.

- **Rule 4:** Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e. moves down one queue).

Tuning MLFQ
How many queues should there be? How big should the time slice be per queue? How often should priority be boosted in order to avoid starvation and account for changes in behavior?
Read page 9 in chapter 8 for ways people parameterize schedulers.

- Rule 1: If Priority(A) > Priority(B), A runs (B doesn't).
- Rule 2: If Priority(A) = Priority(B), A & B run in round-robin fashion using the time slice (quantum length) of the given queue.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S, move all the jobs in the system to the topmost queue.

Chapter 13. The Abtraction: Address Spaces

address space - the running program's view of memory in the system.

It contains all of the memory state of the running program
the **code,** the **stack,** the **heap,** the **registers**
The **code** of the program (the instruction) have to live in memory somewhere, and thus they are in the address space. The program, while it is running, uses a **stack** to keep track of where it is in the function call chain as well as to allocate local variables and pass parameters and return values to and from routines. Finally, the **heap** is used for dynamically-allocated, user-managed memory, such that you might receive froma call to malloc() in C or new in an object-oriented language such as C++ or Java.

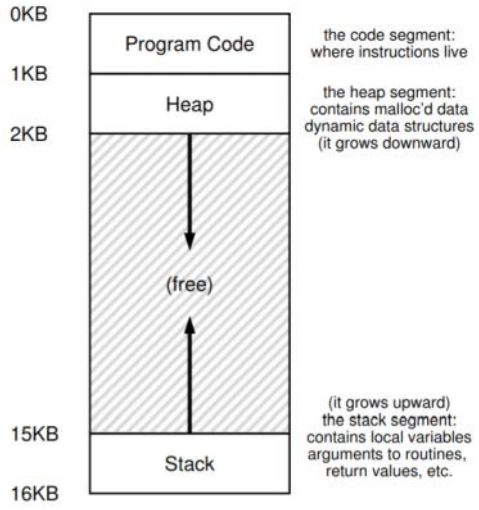


Figure 13.3: **An Example Address Space**

When we describe the address space, what we are describing is the **abstraction** that the OS is providing to the running program. The program isn't really in memory at the physical addresses 0 through 16 KB; rather it is loaded at some arbitrary physical addresses.

This is called **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, when the reality is quite **different**.

Goals of the OS:

- **Transparency** - the program shouldn't be aware of the fact that memory is virtualized; rather, it should behave as if it has its own private physical memory
- **Efficiency** - both time and space efficient
- **Protection** - protect process from one another as well as the OS itself from processes.
    - **Isolation** - each process should be running its own isolated cocoon, safe from the ravages of other faulty or even malicious processes

Chapter 14. Interlude: Memory API

Stack: where allocations and deallocations of it are managed **implicitly** by the compiler for you.
Heap: where allocations and deallocations are **explicitly** handled by you, the programmer.
**the rest of this chapter mainly discusses proper use of malloc() and free(). Not midterm important.*

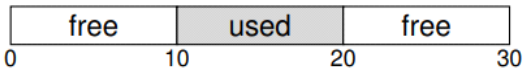
Chapter 17. Free-Space Management

http://pages.cs.wisc.edu/~remzi/OSTEP/vm-freespace.pdf

Malloc() takes a single parameter, size, which is the number of bytes requested by the application; it hands back a pointer (of no particular type, or a **void pointer** in C lingo) to a region of that size (or greater).
free() takes a pointer and frees the corresponding chunk

The space that this library manages is known as the **heap**, and the generic data structure used to manage free space in the heap is some kind of **free list**. The structure contains references to all f the free chunks of space in the managed region in memory.

external fragmentation
the free space gets chopped up into different regions that are not continuous.

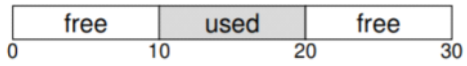


In this diagram, a request for 15 bytes will fail even though there are 20 bytes free.

internal fragmentation
the allocator hands out chunks of memory bigger than that requested, so the memory gets wasted as some of it remains unused by the process.

assume no compaction
once memory is handed out to a client, it cannot be relocated to another location in memory.

coalescing
initial problem: if you have this:



and you call free(10), you still *can't* use any memory space that is bigger than 10 bytes.
solution: coalescing!
the allocator coalesces free space when a chunk of memory is freed. The idea is simple: "When returning a free chunk in memory, look carefully at the addresses of the chunk you are returning as well as the nearby chunks of free space; if the newly freed space sits right next to one (or two) existing free chunks, merge them into a single larger free chunk.

Tracking the size of allocated regions

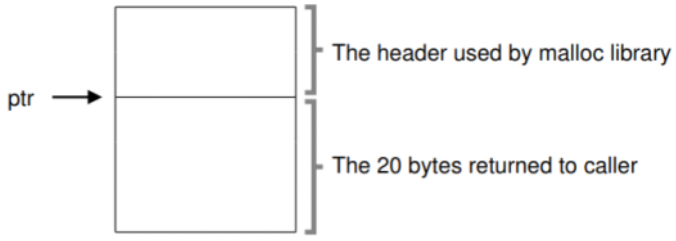


Figure 17.1: **An Allocated Region Plus Header**

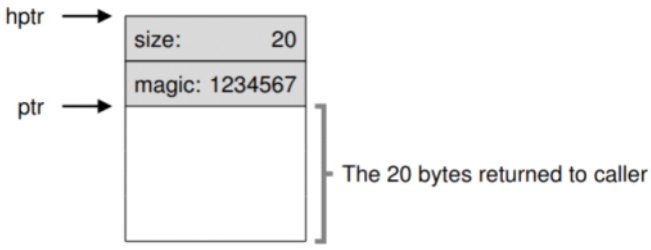


Figure 17.2: **Specific Contents Of The Header**

the free(void *ptr) does not take a size parameter, so most allocators store a little bit of extra information in the **header** block which is kept in memory, usually before the handed-out chunk of memory.

Note: the size of the free region is the size of the header plus the size of the space allocated to the user. Thus, when a user requests *N* bytes of memory, the library does not search for a free chunk of size *N*; rather it searches for a free chunk of size *N* plus the size of the header.

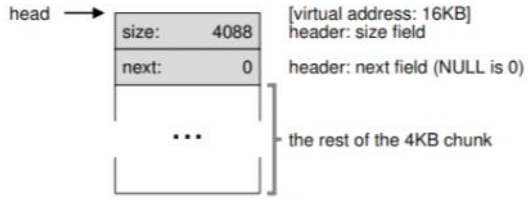


Figure 17.3: **A Heap With One Free Chunk**

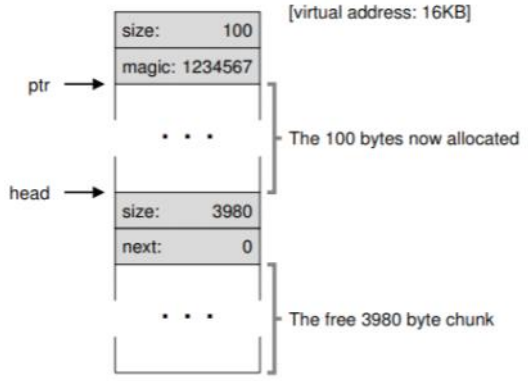


Figure 17.4: **A Heap: After One Allocation**

The status of the list is that it has a single entry of size 4088. The **head** pointer contains the beginning address of this range; let's assume it is 16KB. See figure 17.3.
Now, let's imagine that a chunk of memory is requested of size 100 bytes. To service this request, the library will first find a chunk that is large enough to accommodate the request; because there is only one free chunk, this chunk will be chosen.
Then, the chunk will **split** into two: one chunk big enough to service the request, and the remaining free chunk. Assuming an 8-byte header, the space in the heap is now 3980.

$$(4088 - 100 - 8 = 3980)$$

Basic Strategies

See lec. 5 slides for visual representations and advantages/disadvtanges of each these strategies

Best fit

- search through the entire free list
- find chunks of memory that are bigger than or equal to the requested size
- return the one that is the smallest in that group

Worst fit

- search through the entire free list
- find chunks of memory that are bigger than or equal to the requested size
- return the one that is biggest in that group

Exact opposite of Best fit

First fit

Simply finds the first block that is big enough and returns the requested amount to the user. This approach is faster since doesn't traverse the entire list before returning a chunk.

Next fit

Instead of always beginning the first-fit search at the beginning of the list, this algorithm keeps an extra pointer to the location within the list where one was looking last. The point of this is to soread the searches for free space throughout the list more uniformly. The **First fit** algorithm sometimes pollute the beginning of the free list so **Next fit** fixes that issue.

Other Approaches

Segregated lists

If a particular application has one (or a few) popular-sized requests that it makes, keep a

separate list just to manage objects of that size; keep all other requests in a more general memory allocator.

By having a chunk of memory that only contains one particular size requests, there is much less fragmentation. Also, the search is done faster as you search smaller lists

For more info, see chapter 17 pages 13-15.

Chapter 18. Paging: Introduction

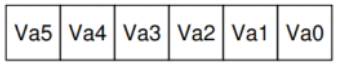
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-paging.pdf

Paging: chopping up space into fixed-sized pieces
Instead of splitting up a process's address space into some number of variable-sized logical segments (e.g. code, heap, stack), we divide it into fixed-ed units, each of which we call a **page**.
We view physical memory as an array of fixed-sized slots called **page frames**; each of these frames can contain a single virtual-memory page.

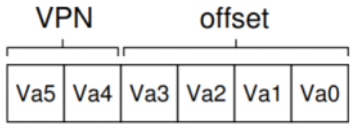
The OS keeps a **free list** of all free pages and just grabs the free X free off the list

To record where each virtual page of the address space is placed in physical memory, the OS usually keeps *a per*-process data structure known as **a page table**. The major role of the page table is to store **address translations** for each of the virtual pages of the address space, thus letting us know where in physical memory each page of the address space, thus letting us know where in physical memory each page resides.

To translate a 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.



becomes:



physical frame number?

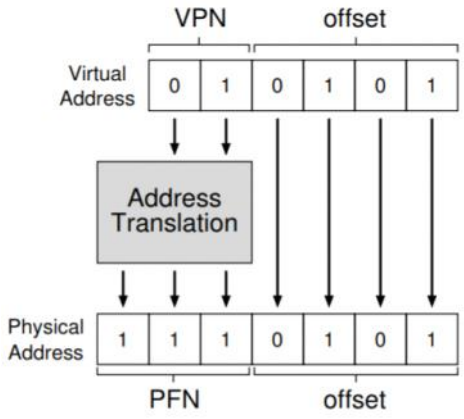


Figure 18.3: **The Address Translation Process**

The offset stays the same, because the offset just tells us which byte within the page we want.

The simplest form of what's inside the page table is called a **linear page table**, which is just an array. The OS indexes the array by the VPN (virtual page number), and looks up the page-table entry (PFN).

As for the contents of each PTE, we have a number of different bits in there worth understanding at some level. A **valid bit** is common to indicate whether the particular translation is **valid**; for example, when a program starts running, it will have code and heap at one end of its address space, and the stack at the other. All the unused space in-between will be marked **invalid**, and if the process tries to access such memory, it will generate a trap to the OS which will likely terminate the process.

Some bits:

- **protection bits:** indicates where the page could be read from, written to, or executed from.
- **present bit:** indicates whether this page is in physical memory or on disk (it has been **swapped**).
- **dirty bit**: indicating whether the page has been modifed since it was brought into memory.

Chapter 19. Paging: Faster Translations (TLBs)

http://pages.cs.wisc.edu/~remzi/OSTEP/vm-tlbs.pdf

Translation-lookaside buffer (TLB)
Part of the chip's **memory-management unit (MMU)**, and is simply a hardware **cache** popular virtual-to-physical address translations (aka **address-translation cache**).

Steps for the TLB

1. Get VPN and check if TLB holds the translation
    a. If yes, we have **TLB hit.** Get the PGN and offset stays the same. (~<1 cycle)
    b. If no, we have **a TLB miss**. More work to do:
        i. Go to the page table in **RAM** and find the needed VPN reference.
            - If you find it in **RAM**, evict the **farthest-in-the-past** entry in the TLB and replace it with the needed VPN. (~20-1000 cycles)
            - If it's not in **RAM**, it's on **DISK**, and it'll take extremely long to retrieve it and replace it with the **farthest-in-the-past** entry in the TLB. (~80M cycles)

The TLB improves performance due to **spatial locality**

- the elements of the array are packed tightly into pages (are close to one another in **space**)

If the page size is bigger, the array access would have fewer misses.

If the program completes, accesses the array again, we're likely to see an even better result: no misses! This is because of **temporal locality**

- the quick re-referencing of memory items in **time.**

Like any cache, TLBs rely upon both spatial and temporal locality for success
Idea behind temporal locality: an instruction or data item that has been recently accessed will likely be re-accessed soon in the future.
Idea behind spatial locality: if a program accesses memory at address x, it will likely soon access memory near x.

Who Handles The TLB Miss?
The **hardware**.
- Has to know where the page tables are located in memory as well as their format
- On a miss, the hardware would "walk" the page table, find the correct page-table entry and extract the desired translation, update the TLB with the translation, and retry the instruction.

software-managed TLB.
- On a TLB miss, the hardware simply raises an exception, which pauses the current instruction stream, raises the privilege level to kernel mode, and jumps to a trap handler.
- Main advantage of this:
  - *flexibility*: the OS can use any data structure it wants to implement the page table, without necessitating hardware change.
  - *simplicity*: as seen in the TLB control flow

The hardware doesn't do much on a miss but raise an exception and the let the OS TLB miss handler do the rest.

TLB Contents: What's In There?
fully associative: any given translation can be anywhere in the TLB, and that the hardware will search the entire TLB in parallel to find the desired translation.

Other common contents:
- **valid bit**: says whether the entry has a valid translation or not
- **protection bit:** determines how a page can be accessed
- **etc.**

Chapter 21. Beyond Physical Memory: Mechanisms

http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys.pdf

swap space
a reserved space on the disk for moving pages back and forth (from memory).
The OS can read from and write to the swap space, in paged-sized units. To do that, the OS will need to remember the **disk address** of a given page.

The hardware can tell whether the piece of information in each page-table entry is there or not is through the **present bit**. The act of accessing a page that is not in physical memory is commonly referred to as a **page fault.**

The **page fault** is handled by the **page-fault handler** that is part of software.
The OS uses the bits in the PTE normally used for data such as the PFN of the page for a disk address. When the OS receives a page fault for a page, it looks in the PTE to find the address, and issues the request to disk to fetch the page into memory.

When the disk I/O completes, the OS willl 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. This next attempt may generate a TLB miss, which would then be serviced and update the TLB with the translation (one could alternately update the TLB when servicing the page fault to avoid this step). Finally, a last restart would find the translation in the TLB and thus proceed to fetch the desired data or instruction from memory at the translated physical address.
During all this, the process is **blocked** and the OS will be free to run other ready processes.

What If Memory Is Full?
The process of picking a page to kick out, or **replace** is known as the **page-replacement policy.**

The OS first must find a physical frame for the soon-to-be-faulted-in page to reside within; if there is no such page, we'll have to wait for the replacement algorithm to run and kick some pages out of memory, thus freeing them for use here.
With a physical frame in hand, the handler then issues the I/O request to read in the page from swap space. Finally, when that slow operation completes, the OS updates the page table and retries the instruction. The retry will result in a TLB miss, and then, upon another retry, a TLB hit, at which point the hardware will be able to access the desired item.

The OS actually keeps a small portion of memory free more proactively.
To do that, it has 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 that is responsible for freeing memory runs. The reason evicts pages until there are HW pages available. The background thread is called the **swap daemen** or **page daemon**.

Other techniques include **clustering** or **grouping** a number of pages and writing them out at once to the swap partition, thus increasing the efficiency of the disk.

Summary
In this brief chapter, we have introduced the notion of accessing more memory than is physically present within a system. To do so requires more complexity in page-table structures, as a present bit (of some kind) must be included to tell us whether the page is present in memory or not. When not, the operating system page-fault handler runs to service the page fault, and thus arranges for the transfer of the desired page from disk to memory, perhaps first replacing some pages in memory to make room for those soon to be swapped in.

Chapter 22. Beyond Physical Memory: Policies

http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys-policy.pdf

Given that main memory holds some subset of all the pages in the system, it can rightly be viewed as a **cache** for virtual memory pages in the system.
Goal: to minimze the number of **cache misses** i.e. minimize the number of times that have to fetch a page from disk.

average memory access time (AMAT)
$AMAT = T_M + (P_{miss} * T_D)$

T_M represents the cost of accessing memory
T_D represents the cost of accessing disk
P_{miss} represents the probability of not finding the data in the cache (a miss); it varies from 0.0 to 1.0

The Optimal Replacement Policy
Furthest in the future algorithm

Based on **principle of locality** using
- *frequency*

- *recency*

This leads to the following two algorithms:

- **Least-Frequently-Used (LFU)** policy replaces the least-frequently-used page when an eviction must take place.
- **Least-Recently-Used (LRU)** policy replaces the least-recently-used page

These both achieve better hit rates than the previous algorithms (FIFO, Random) and are based on the *history*.

Workloads Instead of Small Traces

First workload is 100% random . It doesn't matter what policy you use here because there's no trend or correlation to follow. It only does as well as the Optimal algorithm (OPT) when the page is big enough to fit all the data.

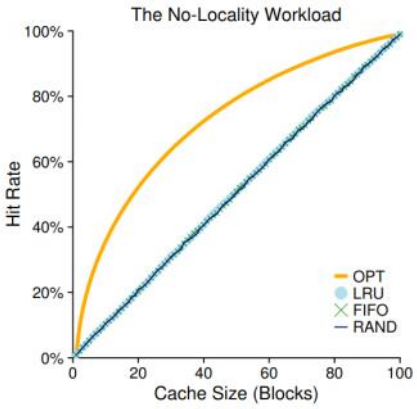


Figure 22.6: **The No-Locality Workload**

Next, the **"80-20" Workload**

80% of the references are made to 20% of the pages (the "hot" pages); the remaining 20% of the references are made to the remaining 80% of the ppages (the "cold" pages). "Hot" pages are referred to most of the time, and "cold" pages the remainder.

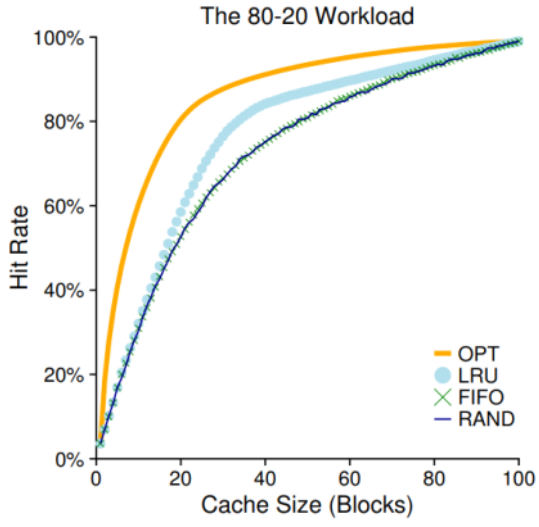


Figure 22.7: **The 80-20 Workload**

The final workload is the **Looping Sequential Workload** as in it we refer to 50 pages in sequence, starting at 0, then 1, …, up to 49, and then we loop, repeating those accesses, for a total of 10,000 access to 50 unique pages.

This is the worst scenario for LRU and FIFO as they kick out older pages. Random does better but not as well as the optimal.

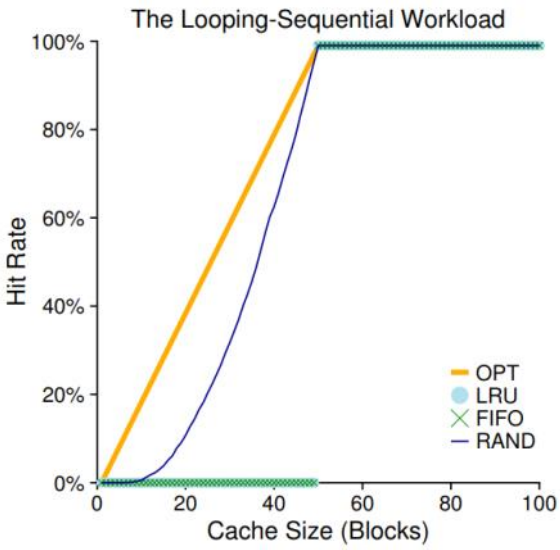


Figure 22.8: **The Looping Workload**

Implementing Historical Algorithms

For the LRU, upon each *page access* (i.e. each memory access, whether an instruction fetch or a load or store), we must update some data structure to move this page to the front of the list (i.e. the MRU side). This is the opposite of FIFO where the list of pages is only accessed when a page is evicted or when a new page is added to the list.

To keep track of which pages have been least- and most-recently used, the system has to do some accounting work *on every memory reference*. Without great care, such accounting could greatly reduce performance.

Approximating LRU

Using hardware support in the form of **use bit** per page of the system where the use bits live in memory somewhere. Whenever a page is referenced (i.e. read or written), the use bit is set by hardware to 1.

With the **clock algorithm**: imagine all the pages of the system are arranged in a circular list.

A **clock hand** points to some particular page to begin with (doesn't matter which). When a replacement must occur, the OS checks if the curently-pointed to page P has a use bit of 1 or 0.

- If 1, this implies that page P was recently used and thus is not a good candidate for replacement. Thus, the use bit for P is set to 0 and the clock hand is incremented to the next page (P+1).
- The algorithm continues it finds a use bit that is 0, implying this page has not been recently used

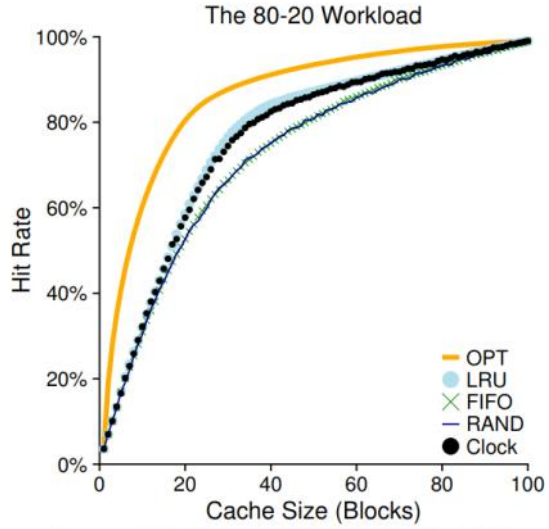


Figure 22.9: **The 80-20 Workload With Clock**

Considering Dirty Pages
If a page has been **modified** and is thus **dirty**, it must be written back to disk to evict it, which is expensive. If it has not been modified (and is thus clean), the eviction is free; the physical frame can simply be reused for other purposes without additional I/O. Thus, some VM systems prefer to evict clean pages over dirty pages. To support this behavior, the hardware includes a **modified bit** (aka **dirty bit**) that is set any time a page is written.

Other VM Policies
demand paging: the OS brings the page into memory when it is accessed "on demand".
prefetching: the OS could guess that a page is about to be used, and thus bring it in ahead of time.
clustering/grouping: writing to disk is expensive. Instead of writing out one at a time, the system can collect a number of pending writes together in memory and write them to disk in one write.

Thrashing
When memory is simply oversubscribed, and the memory demands of the set of running processes simply exceeds the available physical memory. The OS will be constantly paging, we call it **thrashing**.
Approaches to combat this is **admission control**, where the OS could decide to not run a subset of processes with the hope of the **working sets** (the pages that they are using actively) fit in memory and make progress.
Another approach is using an **out-of-memory killer** that chooses a memory-intensive process and kills it.

Chapter 26. Concurrency: An Introduction
http://pages.cs.wisc.edu/~remzi/OSTEP/threads-intro.pdf

A **multi-threaded** program has more than one point of execution (i.e. multiple PCs, each of which is being fetched and executed from).
Threads share the same address space and thus can access the same data.

If there are two threads that are running on a single processor, when switching from running one (T1) to running the other (T2), a **context switch** must take place.
Very similar to process context switching.

- with processes, we saved the state to a **process control block (PCB)**
- with threads, we'll do the same to a **thread control block (TCB)**

Stack is different for multi-threaded processes

- For a **single-threaded** process, there is a single stack residing at the bottom of the address space
- However, in a **multi-threaded** process, each thread runs independently and may call various routines to do whatever work it is doing.
- There will be one stack per frame.
- Any stack-allocated variables, parameters, return values, and other things that we put on the stack will be placed in what is called **thread-local** storage i.e. the stack of the relevant thread.

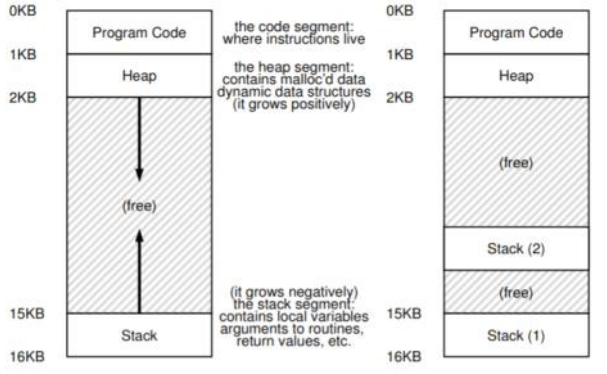


Figure 26.1: **Single-Threaded And Multi-Threaded Address Spaces**

Why use threads?
Two reasons.

1. **Parallelism**
    - you might be running a program that works with very large arrays
    - you can speed things up by using the processors to each perform a portion of the work
2. **To avoid blocking program progress due to slow I/O**
    - if you're writing a program that does a lot of I/O (either sending or receiving messages, page faults, etc.)
    - instead of waiting, you may want your program to do something else
    - while one thread waits, the CPU scheduler can switch to other threads

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

Threads may run in any order and that's non-deterministic. There is no reason to assume that a thread that is created first will run first. What runs next is determined by the OS **scheduler,** and sometimes it may be hard to know what will run at any given moment of time.

Race Conditions
the results depend on the timing execution of the code. With bad luck, the results become **non-deterministic** or **indeterminate**.

Because multiple threads executing this code result in a race condition, we call this code a **critical section**. This is a piece of code that accesses a shared variable and must not be concurrently executed by more than one thread.

What we really want for this code is called **mutual exclusion**. This means that if one thread is executing within the critical section, other threads will be prevented from doing so.

We Want Atomicity
We want instructions that would not be interrupted mid-instruction, and the hardware can guarantee that. Atomically means "as a unit", or "all or none"

we ask the hardware for a few useful instructions to build the general set of **synchronization primitives**. With hardware support and help from the OS, we will be able to build multi-threaded code that accesses critical sections in a synchronized and controlled manner.

ASIDE: KEY CONCURRENCY TERMS
CRITICAL SECTION, RACE CONDITION,
INDETERMINATE, MUTUAL EXCLUSION

These four terms are so central to concurrent code that we thought it worth while to call them out explicitly. See some of Dijkstra's early work [D65,D68] for more details.

- A **critical section** is a piece of code that accesses a *shared* resource, usually a variable or data structure.
- A **race condition** (or **data race** [NM92]) arises if multiple threads of execution enter the critical section at roughly the same time; both attempt to update the shared data structure, leading to a surprising (and perhaps undesirable) outcome.
- An **indeterminate** program consists of one or more race conditions; the output of the program varies from run to run, depending on which threads ran when. The outcome is thus not **deterministic**, something we usually expect from computer systems.
- To avoid these problems, threads should use some kind of **mutual exclusion** primitives; doing so guarantees that only a single thread ever enters a critical section, thus avoiding races, and resulting in deterministic program outputs.

Chapter 27. Interlude: Thread API

http://pages.cs.wisc.edu/~remzi/OSTEP/threads-api.pdf

**detailed instructions on thread creation are included in this chapter. I may not have taken those notes down as I focused on content that will be most relevant to the midterm. Nonetheless, the information in this chapter will prove to be quite useful for the multi-threading lab.*

Locks

when you have a region of code that is a **critical section**, and thus needs to be protected to ensure correct operation, locks are quite useful.

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

- If no other other holds the lock when pthread_mutex_lock() is called, the thread will acquire the lock and enter the critical section.
- If another thread does indeed hold the lock, the thread trying to grab the lock will not return from the call until it has acquired the lock (implying that the thread holding the lock has released it via the unlock call).

The above code is broken in two important ways.

1. **Lack of proper initialization**. all locks must be properly initialized in order to guarantee that they have been the correct values to begin with thus and work as desired when lock and unlock are called.
2. **Fails to check error codes when calling lock and unlock**. if your code doesn't properly check error codes, the failure will happen silently, which in this case could allow multiple threads into a critical section.

condition variables

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.

ASIDE: THREAD API GUIDELINES

There are a number of small but important things to remember when you use the POSIX thread library (or really, any thread library) to build a multi-threaded program. They are:

- **Keep it simple.** Above all else, any code to lock or signal between threads should be as simple as possible. Tricky thread interactions lead to bugs.
- **Minimize thread interactions.** Try to keep the number of ways in which threads interact to a minimum. Each interaction should be carefully thought out and constructed with tried and true approaches (many of which we will learn about in the coming chapters).
- **Initialize locks and condition variables.** Failure to do so will lead to code that sometimes works and sometimes fails in very strange ways.
- **Check your return codes.** Of course, in any C and UNIX programming you do, you should be checking each and every return code, and it's true here as well. Failure to do so will lead to bizarre and hard to understand behavior, making you likely to (a) scream, (b) pull some of your hair out, or (c) both.
- **Be careful with how you pass arguments to, and return values from, threads.** In particular, any time you are passing a reference to a variable allocated on the stack, you are probably doing something wrong.
- **Each thread has its own stack.** As related to the point above, please remember that each thread has its own stack. Thus, if you have a locally-allocated variable inside of some function a thread is executing, it is essentially *private* to that thread; no other thread can (easily) access it. To share data between threads, the values must be in the **heap** or otherwise some locale that is globally accessible.
- **Always use condition variables to signal between threads.** While it is often tempting to use a simple flag, don't do it.
- **Use the manual pages.** On Linux, in particular, the pthread man pages are highly informative and discuss much of the nuances presented here, often in even more detail. Read them carefully!

Chapter 28. Locks

A **lock** is a variable that needs to be declared. It holds the state of the lock at any instant of time. It is either **available** (thus no thread holds the lock) or **acquired** (thus exactly one thread holds the lock) presumably in a **critical section**.

One process tries to acquire the lock by calling lock(). If no other thread is holding it, it becomes the lock **owner**. Now, if another thread calls lock(), it wont return anything so it is prevented from entering that section.

```
1  lock_t mutex; // some globally-allocated lock 'mutex'
2  ...
3  lock(&mutex);
4  balance = balance + 1;
5  unlock(&mutex);
```

once the owner of the lock calls **unlock()**, other threads can now lock it for their own jobs. If there are no jobs waiting, the lock just sits free. If there are jobs waiting, one of them will notice it and acquire it.

without locks, the OS schedules the threads any way it sees fit. Locks give some control back to the programmer

Mutex

the name the POSIX library gives to a lock because it achieves **Mut**ual **Ex**clusion.

Disabling Interrupts to achieve Mutual Exclusion

by turning off interrupts before a critical section, we ensure the process running doesn't get preempted by another process and that is executes **atomically**.

Negatives:

- have to allow the process to do a privileged instruction and trust it wont abuse it
- doesn't work on multi-processors
- interrupts being disabled for a long time means some interrupts get lost
- very poor interactivity and I/O

Why using flags is bad
the first thread that enters the critical section will call **lock()** which tests whether the flag is equal to 1 and then sets the flag to 1 to indicate that the thread now holds the lock. When finished, it calls **unlock()** and sets the flag to 0. If another thread happens to call lock() while the first thread is in the critical section, it will **spin wait** in the while loop for that thread to unlock.

This doesn't work because if the threads are timed unfortunately, they can both set the flag to 1 and enter the critical section. It is also bad for performance because that process waiting in the spin-lock might wait for a long time until the other process releases the lock. If that thread that owns the lock gets stuck in an infinite loop, the thread waiting will never get to run as it'll never get access to that critical section.

test-and-set instruction
returns the old value pointed to by the old_ptr and simultaneously updates said value to new.
The key is that it all happens **atomically**. This instruction is enough to build a simple **spin-lock**.

By making both the **test** (of the old lock value) and **set** (of the new value) a single atomic instruction, we ensure only one thread acquires the lock. In the **spin-lock**, we need a **preemptive scheduler** (i.e. one that will interrupt a thread via a timer, in order to run a different thread, from time to time)

Evaluating spin-locks
It is **correct** as it provides **mutual exclusion**, but **it is not fair.** It doesn't provide any guarantee that a thread waiting will ever get to the critical section.

For a single-processor, spin locks are very bad as preempting a process that's within the critical section means the other ones waiting will spin until their time slice is over. We then after maybe many time slices of nothing getting done return to the one whose acquired the lock for it to continue

With multi-processor, though, it works okay. Imagine Thread A on CPU 1 and Thread B on CPU 2, both contending for a lock. If Thread A on CPU 1 grabs the lock, then Thread B tries to, it will run on B. Presuming the critical section is short, B then acquires it.
Spinning to wait for a lock held on another processor doesn't waste many cycles and thus can be effective.

Compare-And-Swap
basic idea is tot est whether the value at the address specified by **ptr** is equal to **expected**. If so, update the memory location pointed to by **ptr** with the new value. If not, do nothing.

This is very similar to **test-and-set**.

Load-Linked and Store-Conditional
The **load-linked** operates very similar to a typical load instruction, by simply fetching a value from memory and placing it in a register. The key difference comes with the **store-conditional**, which only succeeds (and updates the value stored at the address just load-linked from) if no intervening store to the address has taken place.
In the case of success, the store-conditional returns 1 and updates the value at ptr to value; if it fails, the value at ptr is not updated and 0 is returned.

```
1  int LoadLinked(int *ptr) {
2      return *ptr;
3  }
4
5  int StoreConditional(int *ptr, int value) {
6      if (no update to *ptr since LoadLinked to this address) {
7          *ptr = value;
8          return 1; // success!
9      } else {
10         return 0; // failed to update
11     }
12 }
```

Figure 28.5: **Load-linked And Store-conditional**

Fetch-And-Add
this instruction atomically increments a value while returing the old value at a particular address.

```
1  int FetchAndAdd(int *ptr) {
2      int old = *ptr;
3      *ptr = old + 1;
4      return old;
5  }
```

ticket lock
instead of a single value, this solution uses a ticket and turn variable in a combination to build a lock.
Basic operation:

- 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 in the thread's "**turn**"
- then the instruction lock->turn is then used to determine which thread's turn it is
- when (myturn == turn) for a given thread, it is that thread's turn to enter the critical section
- unlock is accomplished simply by incrementing the turn such that the next waiting thread (if there is one) can now enter the critical section

The important difference with this solution versus our previous attemps is that it ensures progress for all threads. Once a thread is assigned its ticket value, it will be scheduled at some point in the future
Previously, on the **test-and-set** for example, a thread could spin forever even as other threads acquire and release the lock.

Problem with all of these methods
imagine T1 acquires a lock but while running in the critical section, gets stuck somewhere. T2 tries to access the lock but cant because of mutual exclusion. T1 is stuck in there until its timeslice is up in which case T2 gets in the critical section and runs. But again, it's T1's turn to go into the critical section, but it's still stuck in the same place as before. So there is an entire time slice wasted while T2 spins and waits for T1 that's trying the same thing but can never succeed. For N threads, that's N-1 time slices wasted.

Yielding
When a thread is going to spin, just give up the CPU to another thread with the yield() system call.
A thread can have 3 states: **running**, **ready**, or **blocked**. When you yield, you simply change its state from **running** to **ready**. Thus, the yielding process essentially deschedules itself.

Problem:
Imagine you have 100 threads contending for a lock repeatedly. If one thread acquires the lock and is preemted before releasing it, the other 99 will each call lock() and try to get into the critical section, but since it's already locked, they'll all start spinning and consequently all call yield(), which is a huge waste of CPU.
Further, a thread can **starve** if it gets stuck in an endless yield loop while other threads repeatedly enter and exit the critical section.

Sleeping instead of Spinning
we use a **queue** to keep track of which threads are waiting to acquire the lock.
Also, **park()** to put a calling thread to sleep, and **unpark(threadID)** to wake a particular thread as designated by threadID. These two calls put a caller to sleep if it tries tp acquire a held lock and wakes it up when the lock is free. When a thread is woken up, the flag stays at 1 even if the lock is about to be free. It is simply passed from the thread done with the lock to the one whose turn it is (from the **queue**) with the flag **not** being set to 0 in between.

wakeup/waiting race
a park by a thread that sleeps forever after a switch at the wrong time
^to fix that, **setpark()**

Two-phase lock
this kind of lock realizes that spinning can be useful, particularly if the lock is about to be released. So in the first phase, the lock spins for a while, hoping that it can acquire the block.
However, if a lock is not acquired during the first spin phase, a second phase is entered, where the caller is put to sleep, and only woken up when the lock becomes free later.

Chapter 30 (through 30.1). Condition Variables
There are many cases where a thread wishes to check whether a **condition** is true before continuing

To wait for a condition to become true, a thread can make sure of what is known as a **condition variable**: an explicit queue that threads can put themselves on when some state of execution (i.e. some **condition**) is not as desired (by **waiting** on the condition); some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue.

In other words, if the condition is not true, the threads put themselves on the queue. Another thread, when the condition becomes true, can then wake one or more of those waiting threads and allow them to continue.

Chapter 32. Common Concurrency Problems

Bugs are divided into **deadlock** and **non-deadlock** problems. The previous chapters focused deadlocks, but this chapter focuses on non-deadlocks.

Two major types of non-deadlock bugs:

1. **atomicity violation** bugs
2. **order violation** bugs

Atomicity Violation

```
1  Thread 1::
2  if (thd->proc_info) {
3    fputs(thd->proc_info, ...);
4  }
5
6  Thread 2::
7  thd->proc_info = NULL;
```

Figure 32.2: **Atomicity Violation (`atomicity.c`)**

In this example, if there is a context switch exactly before line 3 executes, thread 2 will set the value of proc_info to NULL and so fputs will fail as it will try to dereference NULL. We can fix it by setting locks.

```
1  pthread_mutex_t proc_info_lock = PTHREAD_MUTEX_INITIALIZER;
2
3  Thread 1::
4  pthread_mutex_lock(&proc_info_lock);
5  if (thd->proc_info) {
6    fputs(thd->proc_info, ...);
7  }
8  pthread_mutex_unlock(&proc_info_lock);
9
10 Thread 2::
11 pthread_mutex_lock(&proc_info_lock);
12 thd->proc_info = NULL;
13 pthread_mutex_unlock(&proc_info_lock);
```

Figure 32.3: **Atomicity Violation Fixed (`atomicity_fixed.c`)**

Order-Violation

Order matters.

```
1  Thread 1::
2  void init() {
3     mThread = PR_CreateThread(mMain, ...);
4  }
5
6  Thread 2::
7  void mMain(...) {
8      mState = mThread->State;
9  }
```

Figure 32.4: **Ordering Bug (`ordering.c`)**

As mMain(…) runs, it assumes it's been initialized, but with bad ordering, thread 2 will crash with a NULL-pointer dereference.

"The desired order between two (groups of) memory accesses is flipped (i.e. A should always be executed before B, but the order is not enforced during execution)"

We use **condition variables** to add synchronization.

```
1  pthread_mutex_t mtLock = PTHREAD_MUTEX_INITIALIZER;
2  pthread_cond_t  mtCond = PTHREAD_COND_INITIALIZER;
3  int mtInit             = 0;
4
5  Thread 1::
6  void init() {
7     ...
8     mThread = PR_CreateThread(mMain, ...);
9
10    // signal that the thread has been created...
11    pthread_mutex_lock(&mtLock);
12    mtInit = 1;
13    pthread_cond_signal(&mtCond);
14    pthread_mutex_unlock(&mtLock);
15    ...
16 }
17
18 Thread 2::
19 void mMain(...) {
20     ...
21     // wait for the thread to be initialized...
22     pthread_mutex_lock(&mtLock);
23     while (mtInit == 0)
24         pthread_cond_wait(&mtCond, &mtLock);
25     pthread_mutex_unlock(&mtLock);
26
27     mState = mThread->State;
28     ...
29 }
```

Figure 32.5: **Fixing The Ordering Violation (`ordering_fixed.c`)**

Deadlocks

```
Thread 1:                    Thread 2:
pthread_mutex_lock(L1);      pthread_mutex_lock(L2);
pthread_mutex_lock(L2);      pthread_mutex_lock(L1);
```

Figure 32.6: **Simple Deadlock (`deadlock.c`)**

This scenario doesn't necessarily mean a deadlock will occur, rather it is where a deadlock *can* occur. If after pthread_mutex_lock(L1) there is a context switch, and after pthread_mutex_lock(L2) there is another context switch, we end up with a deadlock. Both threads are waiting for the other and neither can run.
The presence of a **cycle** is indicative of a deadlock.

Reasons Deadlocks Occur

- Large code bases have complex dependencies between components.
- Because of **encapsulation**.
    - While trying to hide details of implementations and make software easier to build in a modular way, we are **more prone** to deadlocks.

Conditions for Deadlocks

Four conditions must hold for a deadlock to occur.

1. **Mutual Exclusion**: threads are able to claim exclusive control of resources that they require.
2. **Hold-and-wait**: threads hold resources allocated to them (e.g. locks) while waiting for additional resources (e.g. locks they wish to acquire)
3. **No preemption**: resources (e.g. locks) cannot be forcible remove from threads that are holding them
4. **Circular wait**: there exits a circular chain of threads such that each thread holds one or more resources (e.g. locks) that are being requsted by the next thread in the chain.

ALL FOUR CONDITIONS MUST BE MET FOR DEADLOCK TO OCCUR.

Prevention

Circular wait:

Most practicular prevention technique is to avoid circular waiting by providing **total ordering** on lock acquisition. For example, if there are only two locks in the system (L1 and L2), you can prevent deadlocks by always acquiring L1 before L2. This strict ordering ensures no cyclical waiting; hence, no deadlock.

For more than two locks, you use **partial ordering**. For example, "*i_mutex* before *i_mmap_rwsem*" or "*i_mmap_rwsm* before *private_lock* before *swap_lock* before *i_pages* lock".

Hold-and-wait
This bug can be avoided by acquiring all locks at once, atomically.

```
1    pthread_mutex_lock(prevention);   // begin acquisition
2    pthread_mutex_lock(L1);
3    pthread_mutex_lock(L2);
4    ...
5    pthread_mutex_unlock(prevention); // end
```

When holding the prevention lock, the thread doing so can grab the locks in any order it wants to because it is holding the prevention lock while doing so.

The problem with this technique is that when calling a routine, this approach requires us to know exactly which locks must be helt and to acquire them ahead of time. This works against us when **encapsulating**.

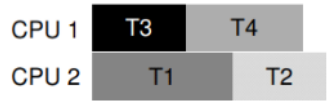
No Preemption
Because we generally view locks as held until unlock is called, multiple lock acquisition often gets us into trouble because when waiting for one lock we are holding another.
We use *pthread_mutex_trylock(),* which either grabs the lock (if it's available) and returns success or returns an error code indicating the lock is held.

```
1  top:
2    pthread_mutex_lock(L1);
3    if (pthread_mutex_trylock(L2) != 0) {
4      pthread_mutex_unlock(L1);
5      goto top;
6    }
```

New problem with this: **livelock**. Two threads could both be repeatedly attempting this sequence and repeatedly failing to acquire both locks. There is no deadlock but no progress is being made.

Mutual Exclusion
This final prevention technique focuses on avodiing the need for mutual exclusion altogether. You need powerful hardware instructions to build data structures in a manner that does not require explicit locking.

Deadlock Avoidance via Scheduling
avoidance might be preferable to prevention.
A smart scheduler can schedule tasks in a way that threads that acquire the same locks are never run together.



It is okay for T1 and T3 to run together, and T2 and T4 to run together, but not T1 and T2 as that can lead to a deadlock.

Chapter 33. Event-based Concurrency (Advanced)

The approach to event-based concurrency is the following:
You simply wait for something (i.e. an "event") to occur; when it does, you check what type of event it is and do the small amount of work it requires (which may include issuing I/O requests, or scheduling other events for future handling, etc).

Event loop

```
while(1) {
    events = getEvents();
    for (e in events)
        processEvent( e);
}
```

The code that processes each event is known as the **event handler**. Importantly, when a handler processes an event, it is the only activity taking place in the system.

Select() (or poll())
These APIs check whether there is any incoming I/O that should be attended to.

select() examines the I/O descriptor sets whose addresses are passed in readfds, writefds, and errorfds to see if some of their descriptors are ready for reading, writing , or have exceptional condition pending, respectively. On return, select() replaces the given descriptor sets with subsets consisting of those descriptors that are ready for the requested operation. select() returns the total number of ready descriptors in all the sets.

Select() lets you check whether descriptors can be read from as well as written to; the former lets a server determine that a new packet has arrived and is in need of processing, whereas the latter lets the service know when it is OK to reply.

Using this approach means there is only one event being handled at a time, and so there is no need to acquire or release locks; the event-based server cannot be interrupted by another thread because it is decidedly single threaded.

Blocking system calls
With multi-threaded programs, when we issue I/O calls, the other threads can run and do their jobs. But with the event-based approach, if an event handler issues a call that blocks, the **entire** server will block until the call completes. The system now sits idle and resources are wasted. Therefore, **no blocking calls are allowed**.

Asynchronous I/O
These interfaces enable an application to issue an I/O request and return control immediately to the caller, before the I/O has completed; additional interfaces enable an application to determine whether various I/Os have completed.

Chapter 36. I/O Devices

"Classical" diagram of a typical system

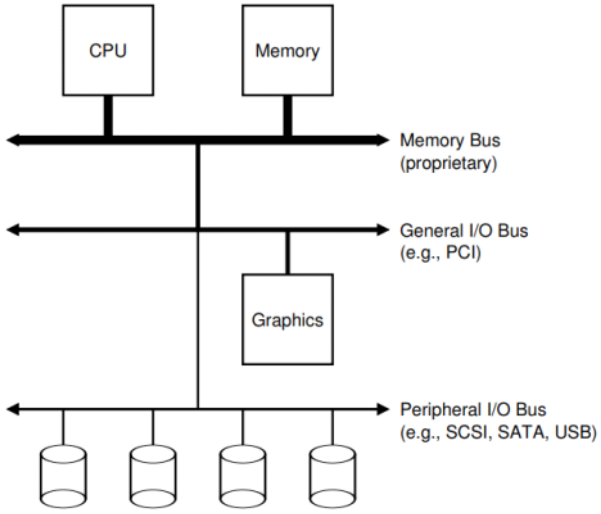


Figure 36.1: **Prototypical System Architecture**

More advanced/modern systems
The CPU connects to an I/O chip via Intel's proprietary **DMI (Direct Media Interface)**, and the rest of the devices connect to this chip via a numer of different interconnects. On the right, one or more hard drives connect to the system via the **eSATA** interface; **ATA** (the **AT Attachment**), then **SATA** (for **Serial ATA**) and now eSATA (**external SATA).**
Below the I/O chips are a number of **USB (Universal Serial Bus)** connections, which enable a keyboard and mouse to be attached to the computer.

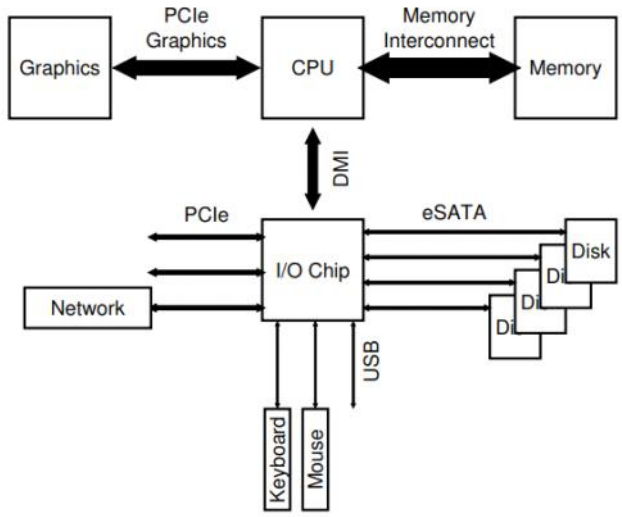


Figure 36.2: **Modern System Architecture**

Canonical Device

Two important components:

1. the first hardware **interface** it presents to the rest of the system
2. its **internal structure**. this part of the device is implementation specific and is responsible for implementing the abstraction the device presents to the system

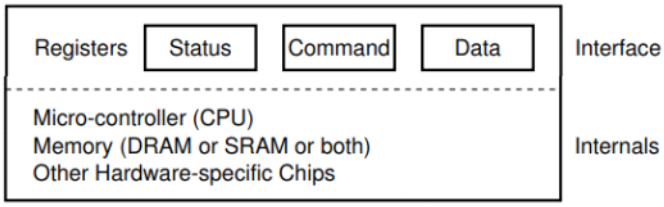


Figure 36.3: **A Canonical Device**

The above simplified device interface has three registers

1. **status** register - which can be read to see the current status of the device
2. **command** register - to tell the device to perform a certain task
3. **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
    (starts the device and executes the command)
While (STATUS == BUSY)
    ; // wait until device is done with your request
```

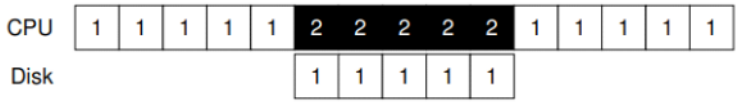
^In the first protocol, the OS waits until the device is ready to receive a command by repeatedly reading the status register, called **polling** the device. Second, the OS sends some data down to the data register; one can imagine that if this were a disk, for example, that multiple writes would need to take place to transfer a disk block to the device. When the main CPU is involved with the data movement, we refer to it as **programmed I/O (PIO)**. Third, the OS writes a command to the command register; doing so implicitly lets the device know the both the data is present and that it should begin working on the command. Finally, the OS waits for the device to finish by again polling it in a loop, waiting to see if it is finished (it may theng et an error code to indicate success or failure).

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 predetermined **interrupt service routing (ISR)** or more simply an **interrupt handler**.

Interrupts allow for **overlap** of computation and I/O, which is key for improved utilization.

E.g. of overlap:



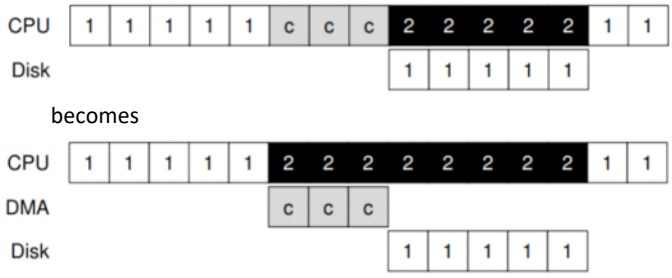
Using interrupts is **not always** the best solution. For example, if a device performs its tasks very quickly: the first poll usually finds the device to be done with task. Using an interrupt in this case will actually **slow down** the system: switching to another process, hadnling the interrupt, and switching back to the issuing process is expensive.
If a device is fast, it may be best to poll; if it is low, interrupts, which allow overlap, are best.
If speed is unknown, use a **hybrid** that polls for a little while and then if device is not yet finished uses interrupts, called the **two-phase** approach.

Tip: although interrupts allow for overlap of computation and I/O, they only really make sense for slow devices. Otherwise, the cost of interurpt handling and context switching may outweigh the benefits of interrupts.olli

Interrupts in Networks
When a huge stream of incoming packets each generate an interrupt, it is impossible for the OS to **livestock**, i.e. find itself only processing interrupts and never allowing a user-level process to run and actually service the requests.
In this case, it may be better to occasionally use **polling** to better control what is happening in the system and allow the web server to service some requests before going back to the device to check for more packet arrivals.

Coalescing
In this setup, 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.

Direct Memory Access (DMA)
A **DMA** engine is essentially a very specific device within a system that can orchestrate transfers between devices and main memory without much CPU intervention.
e.g. 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. The OS is now done. When the DMA is done, it raises an interrupt and the OS knows the transfer is complete.



Methods of Device Interaction
1. explicit **I/O instructions** - specify a way for the OS to send data to specific device registers. Instructions are **privileged** since the OS is the only entity allowed to communicate with devices.
2. **memory-mapped I/O** - hardware makes device registers available as if they were memory locations. To access a particular register, the OS issues load (to read) or store (to write) the address; the hardware then routes the load/store to the device instead of main memory.

How to fit the devices into the OS
It uses **abstraction** and a piece of software at the lowest level of the OS called a **device driver**, which encapsulates any specifics of device interactions.
Negative: if a device has many special capabilities, but has to present a generic interface to the rest of the kernel, those special capabilities will go unused.

IDE Disk Driver
Presents a simple interface to the system, consisting of four types of registers
1. control
2. command block
3. status
4. error

The basic protocol to interact with a device is as follows:

- **wait for drive to be ready** - read status register until drive is READY and not BUSY
- **write parameters to command registers** - write the sector count, local block address (LBA) of the sectors to be accessed, and driv enumber to command registers
- **start the I/O** - by issuing read/write to command register. Write READ-WRITE command to command register
- **data transfer (for writes)** - wait until drive status is READY and DRQ (drive request for data); write data to data report
- **handle interrupts** - in the simplest case, handle an interrupt for each sector transferred; more complex approaches allow batching and thus one final interrupt when the entire transfer is complete
- **error handling** - after each operation, read the status register. If the ERROR bit is on, read the error register for details.

Chapter 37. Hard Disk Drives

the interface

the drive consists of a large number of sectors (512-byte blocks), which of which can be read or written. We can view the disk as an array of sectors; 0 to n-1 is the **address space** of the drive.

the 512-byte write is **atomic** (i.e. it will either complete in its entirety or it won't complete at all).

two blocks near one-another within the drive's address space will be faster than two blocks that are far apart. Accessing blocks in a contiguous chunk is the fastest access mode, and usually much faster than any more random access pattern.

basic geometry

platter - a circular hard surface on which data is store persistently by inducing magnetic changes to it. Has two sides, each of which are called a **surface**.

The platters are bound together around the **spindle**, which is connected to a motor that spins the platters around at a constant fixed rate. The rate of rotation is measured in **rotations per minute (RPM)**

Data is encoded on each surface in concentric circles of sectors, each of which is called a **track**. A single surface contains many thousands and thousands of tracks, tightly packed together, which hundreds of tracks fitting into the width of a human hair.

To read and write from the surface, we need a mechanism that allows us to either sense (i.e. read) the magnetic patterns on the disk or to induce a change in (i.e. write) them. This process of reading and writing is accomplished by the **disk head**; there is one usch 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.

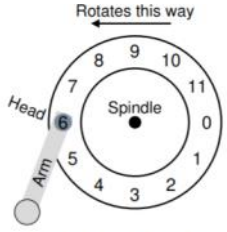


Figure 37.2: **A Single Track Plus A Head**

Single-track Latency: The Rotational Delay

Imagine we receive a request to read block 0.

The **disk** must wait for the desired **sector** to rotate under the **disk head** (this happens often enough in modern drives), and this I/O service time is called **rotational delay**.

Multiple Tracks: Seek Time

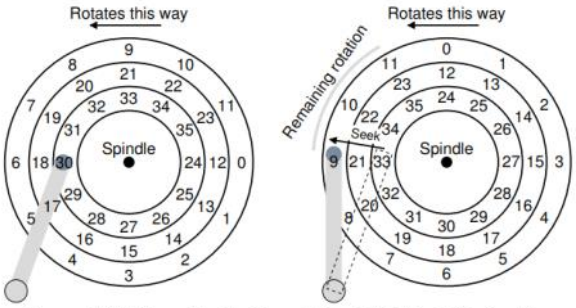


Figure 37.3: **Three Tracks Plus A Head (Right: With Seek)**

Let's study how we might service a request made to a distant sector e.g. 11.
The drive has to first move the disk arm to the correct track (in this case the outermost) in a process known as **seek**.
Phases of **seek**:

- acceleration - the disk arm gets moving
- coasting - the disk are moves at full speed
- deceleration - the disk arm slows down
- settling - the head is carefully positioned over the correct track

The **settling time** is quite significant (~0.5 to 2 ms)
When sector 11 passes under the disk head, the final phase of I/O takes place: the **transfer**, where data is either read from or written to the surface.

Other track details
many drives employ a **track skew** to make sure that sequential reads can be properly services even when crossing track boundaries. Sectors are often skewed like this because when switching from one track to another, the disk needs time to reposition the head (even to neighboring tracks). Without the skew, the head would be moved to the next track but the desired next block would have already rotated under the head, and thus the drive would have to wait almost an entire rotational delay to access the next block.

Multi-zoned disk drives: outer tracks have more sectors than inner tracks. Disk is organized into multiple zones, and where a zone is consecutive set of tracks on a surface.

Finally, an important part of any modern disk drive is its **cache**, called a **track buffer**. This is some small amount of memory (8 - 16MB) which the drive uses to hold data read from or written to the disk.
For example, when reading a sector from the disk, the drive might decide to read in all of the sectors on that track and cache them into memory; doing so allows the drive to quickly respond to any subsequent requests on the same track.
On writes, the drive has to choose

- acknowledge the write has been completed when it has put the data in its memory (called **write back** caching or **immediate reporting**)
- or after the write has actually been written to disk (called **write through**)

Write back caching can make the drive appear "faster" but can be dangerous if the file system requires data to be written to the disk in a certain order for correctness.

I/O time
$T_{I/O} = T_{seek} + T_{rotation} + T_{transfer}$

random workload - issues small (e.g. 4KB) reads to random locations on the disk. Common in database management systems.
sequential workload - reads a large number of sectors consecutively from the disk

Disk Scheduling
SSTF: Shortest Seek Time First
SSTF orders the queue of I/O requests by track, picking requests on the nearest track to complete first. For example, assuming the current position of the head is over the inner track, and we have requests for sectors 21 (middle track) and 2 (outer track), we would issue request to 21 first, wait for it complete, and then issue the request to 2.

Problem with SSTF is the **drive geometry** is not available to the host OS; rather, it sees an array of blocks. This problem is fixed by implementing **nearest-block-first (NBF)**, which schedules the request with the nearest block address next.

Second issue: **starvation**.
if there were a steady stream of requests to the inner track, where the head currently is positioned. Requests to any other tracks would then be ignored completely by the pure SSTF approach.

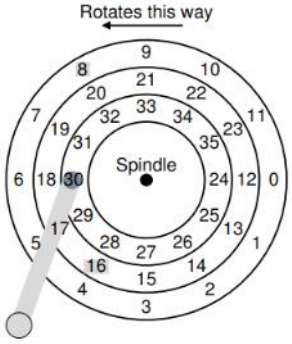
Solution: **Elevator (aka SCAN or C-SCAN)**
the algorithm simply moves back and forth across the disk servicing in order across the tracks. A single pass across the disk (from outer to inner, or inner to outer) a **sweep**. Thus, if a request comes for a block on a track that has already been serviced on this sweep of the disk, it is not handled immediately, but rather queued until the next sweep (coming in the other direction).

F-SCAN freezes the queue to be serviced when it is doing a sweep. This places requests that come in during the sweep into a queue to be serviced later. Doing so avoids **starvation** of far-away requests, by delaying the servicing of late-arriving (but nearer by) requests.

C-SCAN
instead of sweeping in both directions across the disk, the algorithm only sweeps from outer to inner, and then resets the outer track to begin again.

Unfortunately, SCAN is not good enough.

SPTF: Shortest Positioning Time First
If seek time is much higher than rotational delay, then SSTF is fine. However, imagine if seek is quite a bit faster than rotation. It would here make more sense to **seek further** instead of waiting for the closer one to rotate.
Example:



We start at position 30. We have requests for both 8 and 16. If seek time is much faster than rotation time, waiting for 16 to rotate all the way doesn't make sense. Instead, we go to the further request by seeking.

I/O merging
imagine a series of requests to read blocks 33, 8, then 34. The scheduler should **merge** the requests for blocks 33 and 34 into a single two-block request; any re-ordering that the scheduler does is performed upon the merged requests.
This lowers overhead.

Should the OS wait before issuing an I/O to disk?
One might naively think that the disk once it has received a single I/O should immediately issue the request to the drive; called **work-conserving**
anticipatory disk scheduling/non-work-conserving waits for a new and "better" request to arrive and thus increase the overall efficiency.

Chapter 38. Redundant Arrays of Inexpensive Disks (RAIDs)

RAID is a technique to use multiple disks in concert to build a faster, bigger, more reliable system.
Externally, a RAID looks like a disk: a group of blocks one can read or write.
Internally, a RAID consists of multiple disks, memory (both volatile and non-), and one or more processors to manage the system. A hardware RAID is very much like a computer system, specialized for the task of managing a

group of disks.

Advantages:

- **performance** - using multiple disks in parallel can greatly speed up I/O times
- **capacity** - large data sets demand large disks
- **reliablity** - without RAID, spreading data across multiple disks makes data vulnerable to the loss of a single disk. With some form of **redundency**, RAIDs can tolerate the loss of a disk and keep operating as if nothing were wrong.

RAIDs provide these advantages **transparently** to systems that use them, i.e. a RAID looks just like a big disk to the host system. This means that a single disk can be replaced by a RAID without changing a single line of software code. This greatly improves **deployability** of RAID, enabling users and admins to put a RAID to use without worries of software compatibility.

Interface and RAID Internals
When a file system issues a *logical I/O* request to the RIAD, the RAID internally must calculate which disk (or disks) to access in order to complete the request, and then issue one or more *physical I/Os* to do so.

A RAID system is often built as a separate hardware box, with a standard connection to a host. Internally, however, RAIDs are fairly complex, consisting of a microcontroller that runs firmware to direct the operation of the RAID, volatile memory such as DRAM to buffer data blocks as they are read and written, and in some cases, non-volatile memory to buffer writes safely and perhaps even specialized logic to perform parity calculations. At a high level, a RAID is very much a specialized computer system: it has a processor, memory, and disks; however, instead of running applications, it runs specialized software designed to operate the RAID.

Fault Model
RAIDs are designed to detect and recover form certain kinds of disk faults; thus, knowing exacly which faults to expect is critical in arriving upon a working design.
fail-stop fault model - disk is in exactly one of two states: working or failed. With a working disk, all blocks can bread or written. In contrast, when a disk is failed, we assume it is permanently lost.

Raid Level 0: Striping
This isn't actually not a RAID level at all; there is no redundancy. **Striping** serves as an excellent upper-bound on performance and capacity and thus is worth understanding.

The simplest form of striping will **stripe** blocks across the disks of the system as follows:

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

Figure 38.1: **RAID-0: Simple Striping**

^spread the blocks of the array across the disks in RR fashion. This approach is designed to extract the most parallelism from the array when requests are made for contiguous chunks of the array (e.g. in a large sequential read). We call blocks in the same row a **stripe,** thus blocks 0, 1, 2, and 3 are in the same stripe above.
In the example above the chunk size is 1. You can also do with other chunk sizes.

Disk 0	Disk 1	Disk 2	Disk 3	
0	2	4	6	chunk size:
1	3	5	7	2 blocks
8	10	12	14	
9	11	13	15	

Figure 38.2: **Striping With A Bigger Chunk Size**

Each chunk is 4KB, so our **chunk size** is 8KB, and a stripe consists of 4 chunks or 32KB of data.

Chunk Sizes
chunk sizes mostly affect performance of the array.

For example, a small chunk size implies that many files will get striped across many disks, thus increasing the parallism of reads and writes to a single file; however, the positioning time to access blocks across multiples disks increases, because the positioning time for the entire request is determined by the maximum of the positioning times of the requests across all drives.

A big chunk size, on the other hand, reduces such intra-file parallism, and thus relies on multiple concurrent

requests to achieve high throughput. However, large chunk sizes reduce positioning time; if, for example, a single file fits within a chunk and is thus is placed on a single disk, the positioning time incurred while accessing it will just be positioning time of a single disk.

Analysis of RAID-0

- *capacity* - it is perfect: given N disks each of size B blocks, striping delivers N.B blocks of useful capacity.
- *reliability* - very bad: any disk failure will lead to data loss.
- *performance* - excellent: all disks are utilized, often in parallel, to service user I/O requests

Evaluating RAID performance

There are two performance metrics:

- *single-request latency* - understanding the latency of a single I/O request to a RAID is useful as it reveals how much parallelism can exist during a single logical I/O operation.
- *steady-state throughput* - the total bandwidth of many concurrent requests.

Two types of workloads

- **sequential** - we assume that requests to the array come in large contiguous chunks (e.g. request starting at block x and ending at block $x + 1$).
- **random** - we assume that each request is rather small, and that each request is to a different random location on disk (e.g. a random stream of requests may first access 4KB at logical address 10, then at logically address 550,000, then at 20,100, and so on).

With sequential access, a disk is at its most efficient, spending little time seeking and waiting for rotation and most of its time transferring data.
With random access, most time is spent seeking and waiting for rotation and relatively little time is spent transferring data (since the requests are rather small).

RAID Level 1: Mirroring

we simply make more than one copy of each block in the system, with each copy placed on a separate disk. By doing so, we can tolerate disk failures.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

Figure 38.3: **Simple RAID-1: Mirroring**

^this is called **RAID-10** which contains two large striping (RAID-0) arrays, and then mirrors (RAID-1) on top of them.
When reading from a block from a mirrored array, the RAID has a choice: it can read either copy. But when writing a block, it **must** update *both* copies of the data to preserve reliability.

RAID-1 Analysis

- *capacity* - expensive: with a mirroring level=2, we only obtain half of our peak useful capacity. With N disks of B blocks, RAID-1 useful capacity is (N.B)/2
- *reliability* - really good: it can tolerate the failure of any one disk (because there is an identical copy of it). This approach is good for handling only a single failure. If both copies fail, the data is lost.
- *performance* -
    - from the perspective of the latency of a single read request, we can see it is the same as the latency on a single disk; all the RAID-1 does is direct the read to one of its copies. The writes is roughly the same since despite having to write to two blocks, the writes take place in **parallel**. However, since the logical write must wait for both physical writes to complete, it suffers the worst-case seek and rotational delay of the two requests, and thus will be slightly higher than a write to a single disk.

steady-state throughput analysis

when writing out to disk sequentially, each logical write must result in two physical writes. Max bandwidth obtained during sequential writing to a mirrored array is $\frac{N}{2} * S$, or half the peak bandwidth.
unfortunately, we obtain the exact same performance during a sequential read. Thus, the sequential read will only obtain a bandwidth of $(\frac{N}{2} * S)$ MB/s.

Random reads are the best case for a mirrored RAID since we can distribute the reads across all the disks, and thus obtain the full possible bandwidth. Thus, for random reads, RAID-1 delivers N*R MB/s

Random writes: $\frac{N}{2} * S$ MB/s. Each logical write must turn into two physical writes, and thus while all of the disks

will be in use, the client will only perceive this as half the available bandwidth.
Even though a write to logical block x turns into two parallel writes to two different physical disks, the bandwidth of many small requests only achieves half of what we saw with striping

RAID Level 4: Saving Space with Parity
Parity-based approaches attempt to use less capacity and thus overcome the huge space penalty paid by mirrored systems. The cost: performance.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.4: **RAID-4 With Parity**

^for each stripe of data, we have added a single **parity** block that stores the redundant information for that stripe of blocks.
E.g. parity block P1 has redundant information that is calculated from blocks 4,5,6, & 7.

For a given set of bits, we **XOR** of all of those bits returned **0** if there is an **even** number of 1's in the bits, and a **1** if there is **od** number of 1's.

C0	C1	C2	C3	P
0	0	1	1	XOR(0,0,1,1) = 0
0	1	0	0	XOR(0,1,0,0) = 1

To remember it in an easy way: *the number of 1s in any row, including the parity bit, must be an **even** number*.
That is the **invariant** that the RAID must maintain in order for the parity to be correct.

Recovering from a failure with the parity bit
To figure out what values have been in the column, we simply have to red in all the other values in that row (including the XOR'd parity bit) and **reconstruct** the right answer (to get an even number of 1s).

What about blocks of bigger sizes?
E.g. if we had blocks of size 4 bits, it will look like this:

Block0	Block1	Block2	Block3	Parity
00	10	11	10	11
10	01	00	01	10

RAID-4 Analysis

- *capacity* - RAID-4 uses 1 disk for parity information for every group of disks it is protecting. Thus, our useful capacity for a RAID group is $(N-1)*B$.
- *reliability* - RAID-4 tolerates 1 disk failure and no more. If more than one disk is lost, there is simply no way to reconstruct the lost data.
- *performance* -
  steady-state throughput
  Sequential reads/writes
  sequential reads can utilize all of the disks except for the parity disk, and thus deliver a peak effective bandwidth of $(N-1)*S$ MB/s

  Sequential writes: when writing a big chunk of data to disk, RAID-4 can perform a simple optimization called **full-stripe write**.

  E.g. imagine the case where the blocks 0, 1, 2, and 3 have been sent to the RAID as part of a write request. In this case, the RAID can simply calculate the new value of P0 (by performing an XOR across the blocks 0, 1, 2, and 3) and then write all of the blocks to the five disks above in parallel. Full-stripe writes are the most efficient way for RAID-4 to write to disk.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

Figure 38.5: **Full-stripe Writes In RAID-4**

The effective bandwidth is (N-1)*S MB/s

For *sequential* writes on RAID-4, the effective bandwidth is the same as above: (N-1)*S MB/s
(same as sequential reads).

Random reads/writes
For *random* reads, a set of 1-block random reads is spread across data disk of the system but not the parity disk. Effective bandwidth is (N-1)*R MB/s.

For *random* writes, when overwriting one bit in a block, the parity is no longer correct, so we gotta update that too. Two ways to this:

additive parity: read in all of the other data blocks in the stripe in parallel and XOR those with the new block to compute the parity block. To complete the write, you then write the new data and new parity to their respective disks, also in parallel.
Problem with this method: it scales the number of disks, so in much larger RAIDs requires a high number of reads to compute the parity.

subtractive parity: 3 steps
a. read in the old data at C2 ($C2_{old} = 1$) and the old parity ($P_{old} = 0$)
b. compare the old data with the new data
    i. if they are the same ($C2_{new} = C2_{old}$), the parity block will stay the same
    ii. if they are different, we flip the parity block

$$P_{new} = (C_{old}\ XOR\ C_{new})\ XOR\ P_{old}$$

We perform the XOR-ing over all the bits in the block (e.g. 4096 bytes in each block multiplied by 8 bits per byte).

When to use additive parity or subtractive parity?
Let's assume we are using the subtractive method. For each write, the RAID has to perform 4 physical I/Os (two reads and two writes). Now imagine there are lots of writes submitted to the RAID; how many can it perform in parallel?

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
*4	5	6	7	+P1
8	9	10	11	P2
12	*13	14	15	+P3

imagine there are two writes to 4 and 13 in disks 0 and 1. The RAID has no problem performing these in parallel. **However**, it can't read/write the parity block in parallel. That's the **performance bottleneck** called **small-write problem** for parity-based RAIDs.
Even though the data disks can be accessed in parallel, the parity bit prevents the parallelism

We achieve (R/2) MB/s, so the RAID-4 *throughput* under random small writes is **terrible**.

Now, *latency*.
A single read is just mapped to a single disk, and so its latency is the same as the latency of a single disk request.
The latency of a single write requires two reads and then two writes; the reads and writes can happen in parallel, and so the total latency is above twice that of a single disk.

^this was all RAID-4 performance analysis

RAID Level 5: Rotating Parity
This is almost idential to RAID-4, except that is **rotates** the parity block across the drives.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

Figure 38.7: **RAID-5 With Rotated Parity**

RAID-5 Analysis
The effective capacity and failure tolerance is the same as RAID-4. So are sequential read and write performance.
The latency of a single request (whether a read or write) is also the same.

Random reads is a little better because we can now utilize all disks.
Random writes improve a lot over RAID-4, as it allows for parallelism across requests.
E.g. imagine a write to block 1 and a write to block 10.
This turns into requests to

- disk 1 and disk 4 (for block 1 and its parity)
- disk 0 and disk 2 (for block 10 and its parity)

Then there is parallelism.

Bandwidth for small writes will be $\frac{N}{4} \cdot R\ MB/S$

RAID-5 has almost completely replaced RAID-4, except for systems that can only do RAID-4 since RAID-4 is easier to build.

Summary of bandwidths

	RAID-0	RAID-1	RAID-4	RAID-5
Capacity	$N \cdot B$	$(N \cdot B)/2$	$(N-1) \cdot B$	$(N-1) \cdot B$
Reliability	0	1 (for sure) $\frac{N}{2}$ (if lucky)	1	1
Throughput				
Sequential Read	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Sequential Write	$N \cdot S$	$(N/2) \cdot S$	$(N-1) \cdot S$	$(N-1) \cdot S$
Random Read	$N \cdot R$	$N \cdot R$	$(N-1) \cdot R$	$N \cdot R$
Random Write	$N \cdot R$	$(N/2) \cdot R$	$\frac{1}{2} \cdot R$	$\frac{N}{4} R$
Latency				
Read	T	T	T	T
Write	T	T	$2T$	$2T$

Figure 38.8: **RAID Capacity, Reliability, and Performance**

Chapter 39. Interlude: Files and Directories
http://pages.cs.wisc.edu/~remzi/OSTEP/file-intro.pdf

Persistent storage device, such as a classic **hard disk drive** or a more modern **solid-state storage device**, stores information permanently.

Files and Directories
a **file** is 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; the user is not aware of this name.

The low-level name of a file is called its **inode number**

The OS does not need to know much about the structure of the file; rather, the responsibility of the file system is simply to store such data persistently on disk and make sure that when you request the data again, you get what you put there in the first place.

a **directory** also has a low-level name (i.e. an inode number), but it contains a list of pairs (user-readable name, low-level name).

E.g. let's say there is a file with the low-level name "10" and referred to by a user-readable name "foo". The directory that "foo" resides in would have an entry ("foo", "10") that maps the user-readable name to the low-level name.

Directory tree: placing directories within other directories, under which all files and directories are stored.

The directory hierarchy starts at a **root directory** and uses a **separator** to name subsequent **sub-directories** until the desired file or directory is named.

E.g. of **absolute pathname**: /foo/bar.txt

The two parts in a file name separated by a period:
the first part is an arbitrary name, whereas the second is the file **type** (e.g. .jpg)

Creating Files

This can be accomplished with the **open** system call and passing **O_CREAT** flag.
e.g.

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

One important aspect of **open()** is what is returns: a **file descriptor**.
A file descriptor is just an integer that is private per process. It is used in UNIX systems to access files.
Once a file is opened, you use the file descriptor to read or write to the file.
It's a **capability** to perform certain operations.

The file descriptors are like pointers kept in a simple structure (e.g. an array) that point to a **struct file**, which will be used to track information about the file being read or written.

Reading and Writing Files

strace: traces every system call made by a program while it runs, and dumps the trace to the screen for you to see.
Using strace:

```
prompt> echo hello > foo
prompt> cat foo
hello
prompt>
```

We wanna see how "cat" accessed foo to echo its output.

```
prompt> strace cat foo
...
open("foo", O_RDONLY|O_LARGEFILE)       = 3
read(3, "hello\n", 4096)                = 6
write(1, "hello\n", 6)                  = 6
hello
read(3, "", 4096)                       = 0
close(3)                                = 0
...
prompt>
```

open() system call

The call to open returns 3 because each running process already has three files open:

- **standard input** (which the process can read to receive input)
- **standard output** (which the process can write to in order to dump info on the screen)
- **standard error** (which the process can write error messages to)

These are represented by 0, 1, and 2, respectively.

read() system call

- first argument: the file descriptor
- second argument: points to a buffer where the result of the read() will be placed
- third argument: size of the buffer

returns the number of bytes read ("hello" has 6 characters)

write() system call

if highly optimized, calls write() directly
else, uses **printf()** internally.

close() system call
The cat tries to read more bytes in the file, but since there are none left, returns 0, and the program knows it has read the entire file. Thus, the program calls **close()** to indicate that it is done with the file.

Writing to a file is similar.
First, a file is **opened** for writing
Then, the **write()** system call is called, perhaps repeatedly for larger files
finally, **close()**

Reading and Writing, But Not Sequentially
Sometimes, you want to read or write to a specific offset within a file.

lseek() system call

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

- 1st argument: file descriptor
- 2nd argument: the offset, which positions the **file offset** to a particular location within the file
- 3rd argument: **whence** determines exactly how the seek is performed

from the man page of 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.
```

For each file a process opens, the OS tracks a "current" offset, which determines where the next read or write will begin reading from or writing to within the file. Thus, part of the abstraction of an open file is that it has a current offset, which is updated in one of two ways.

- the first is when a read or write of N bytes takes place, N is added to the current offset
- the second is *explicitly* with **lseek**, which changes the offset as specified above

The file structures represent all of the currently opened files in the system; together, they are sometimes referred to as an **open file table**.

Shared File Table Entries: *fork()* And *dup()*
Normally, each logical reading or writing of a file is independent, and each has its own current offset while it accesses the given file.

However, not always.

fork()
using such a system call, we have the child adjust the current offset and the parent offset checking and printing out its value. We make use of a **reference count**.
When a file table entry is shared, its reference count is incremented; only when both processes close the file (or exit) will the entry be removed.

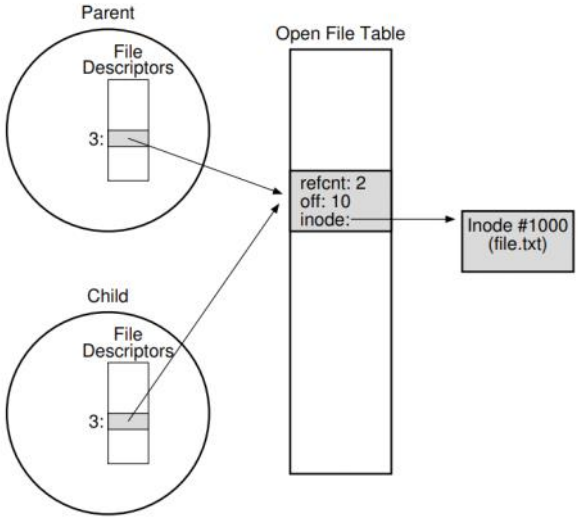


Figure 39.3: **Processes Sharing An Open File Table Entry**

Another way of sharing is with **dup()** system call.
The **dup()** call allows a process to create a new file descriptor that refers to the same underlying open file as an existing descriptor.

```
int main(int argc, char *argv[]) {
    int fd = open("README", O_RDONLY);
    assert(fd >= 0);
    int fd2 = dup(fd);
    // now fd and fd2 can be used interchangeably
    return 0;
}
```

Figure 39.4: **Shared File Table Entry With dup() (dup.c)**

dup() is useful when writing a UNIX shell and performing operations like output redirection.

Writing Immediately with fsync()
Most of the time, when we call write, we tell a file system to write this data to persistent storage.
The file system, for performance reasons, will **buffer** such writes in memory for some time; at that later point in time, the write(s) will actually be issued to the storage device. Often, data will not be lost.

However, some apps require more.

fsync (int fd)
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.

Renaming Files
mv renames files

```
prompt> mv foo bar
```

Using strace, you can see that it uses system call **rename(char *old, char *new)**
One interesting guarantee by rename() call is that it is implemented as an **atomic** call with respect to system crashes; if the system crashes during the renaming, the file will either be named the old name or the new name.

Getting Information About Files
metadata: information about each file it is storing.
Each file system usually keeps this information in a structure called an **inode**.
All inodes reside on disk; a copy of active ones are usually cached in memory to speed up access.

Removing files
Using **rm**, we see that it calls **unlink()**.

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

Making Directories

You can only update a directory indirectly (e.g. by creating files, directories, or other object types within it). The file system considers itself responsible for the intergrity of the directory data.

mkdir()

```
prompt> strace mkdir foo
...
mkdir("foo", 0777)                          = 0
...
prompt>
```

When such a directory is created, it is considered "empty".
However, an empty directory has two entries:

- one entry that refers to itself (.)
- one entry that refers to its parent (..)

Reading Directories

ls does that
E.g. other way to read directory:

```
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;
}
```

Deleting Directories

rmdir does so but requires the directory to be empty (i.e. only have "." and ".." entries).

Hard Links

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.

```
prompt> echo hello > file
prompt> cat file
hello
prompt> ln file file2
prompt> cat file2
hello
```

^we created a file with the word "hello" in it. We then create a hard link to that file using the **ln** program. After this, we can examine the file by either opening **file** or **file2**.

link() simply creates another name in the directory you are creating the link to, and refers to the **same inode** number of the original file. The file is not copied in any way; rather, you now just have two human-readable names both referring to the **same file**.

why we use unlink()

When you create a file, you are really doing two things

- first, you are making a structure (the inode) that will track virtually all relevent information about the file, including its size, where its blocks are on disk, etc.
- second, you are **linking** a human-readable name to that file, and putting that link into a directory.

```
prompt> rm file
removed 'file'
prompt> cat file2
hello
```

^the reason this works is because when a file system inlinks a file, it checks a **reference count** within the inode number. This ref count allows the file system to track how many different file names have been linekd to this particular inode.

Symbolic Links
a.k.a **soft link**.

You can't create a hard link to a directory; can't hard link to file in other disk partitions (bc inode numbers are only unique within a particular file system.

to create a soft link: **ln -s**

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
```

Looks similar to hard link, but it's different.

The first difference is that a symbolic link is actually a file itself but of a different type.
Running **ls** you can see that the first character in the leftmost column is "-" for regular files, "d" for directories, and **"l"** for soft links

```
prompt> ls -al
drwxr-x---  2 remzi remzi   29 May  3 19:10 ./
drwxr-x--- 27 remzi remzi 4096 May  3 15:14 ../
-rw-r-----  1 remzi remzi    6 May  3 19:10 file
lrwxrwxrwx  1 remzi remzi    4 May  3 19:10 file2 -> file
```

The reason file2 is 4 bytes is because the way a symbolic link is formed is by holding the pathname of the linked-to file as the data of the link file.

Finally, because of the way symbolic links are created, they leave a possibility for a **dangling reference**.

```
prompt> echo hello > file
prompt> ln -s file file2
prompt> cat file2
hello
prompt> rm file
prompt> cat file2
cat: file2: No such file or directory
```

Permission Bits and Access Control Lists
The abstraction of files/directories is different from that of the CPU and memory, in that files are commonly **shared** among different users and processes and not always private.

permission bits

```
prompt> ls -l foo.txt
-rw-r--r--  1 remzi wheel  0 Aug 24 16:29 foo.txt
```

the bits "-rw-r--r--" determine for each regular file, directory, and other entities, exactly who can access it and how.
The permissions consist of three groupings:

- what the **owner** of the file can do to it
- what someone in a **group** can do to the file
- what **anyone** (or **other**) can do

In the example above, the first 3 characters say the owner and read and write (rw-), and only readable by members of the group **wheel** and also by anyone in the system (r--r--).

chmod
the way the owner of the file can change these permissions

access control list (ACL)
more sophisticated permission controls.
Access control lists are a more general and powerful way to represent exactly who can access a given resource.

Making and Mounting A File System
To make a file system, most file systems provide a tool, called **mkfs**.
The idea is as follows:

- give the tool, as input, a device (such as a file systems e.g. /dev/sda1) and a file system type (e.g. ext3), and it simply writes an empty file system, starting with a root directory, onto that disk partition

However, once the file system is created, it needs to be made accessible within the uniform file-system tree.
Mount does that.

ASIDE: KEY FILE SYSTEM TERMS

- A **file** is an array of bytes which can be created, read, written, and deleted. It has a low-level name (i.e., a number) that refers to it uniquely. The low-level name is often called an **i-number**.
- A **directory** is a collection of tuples, each of which contains a human-readable name and low-level name to which it maps. Each entry refers either to another directory or to a file. Each directory also has a low-level name (i-number) itself. A directory always has two special entries: the `.` entry, which refers to itself, and the `..` entry, which refers to its parent.
- A **directory tree** or **directory hierarchy** organizes all files and directories into a large tree, starting at the **root**.
- To access a file, a process must use a system call (usually, `open()`) to request permission from the operating system. If permission is granted, the OS returns a **file descriptor**, which can then be used for read or write access, as permissions and intent allow.
- Each file descriptor is a private, per-process entity, which refers to an entry in the **open file table**. The entry therein tracks which file this access refers to, the **current offset** of the file (i.e., which part of the file the next read or write will access), and other relevant information.
- Calls to `read()` and `write()` naturally update the current offset; otherwise, processes can use `lseek()` to change its value, enabling random access to different parts of the file.
- To force updates to persistent media, a process must use `fsync()` or related calls. However, doing so correctly while maintaining high performance is challenging [P+14], so think carefully when doing so.
- To have multiple human-readable names in the file system refer to the same underlying file, use **hard links** or **symbolic links**. Each is useful in different circumstances, so consider their strengths and weaknesses before usage. And remember, deleting a file is just performing that one last `unlink()` of it from the directory hierarchy.
- Most file systems have mechanisms to enable and disable sharing. A rudimentary form of such controls are provided by **permissions bits**; more sophisticated **access control lists** allow for more precise control over exactly who can access and manipulate information.

Chapter 40. File System Implementation
http://pages.cs.wisc.edu/~remzi/OSTEP/file-implementation.pdf
Very Simple File System (vsfs)

40.1 The Way To Think

- **data structures**: what type of on-disk structures are utilized by the file system to organize its data and metadata?
- **access methods**: how does it map the calls made by a process, such as open(), read(), write(), etc., onto its structures?

40.2 Overall Organization
the disk is divided into **blocks**; simple file systems use just one block size, e.g. 4 KB.
They are addressed from 0 to N-1.

Most of the disk is user data in a region called **data region**.
A file system has to also track information about each file. This information is the **metadata**, and tracks things like which blocks (in the data region) comprise a file, the size of the file, its owners and access rights, access and modify times, and other similar kinds of information. This is tored using a structure called an **inode**.

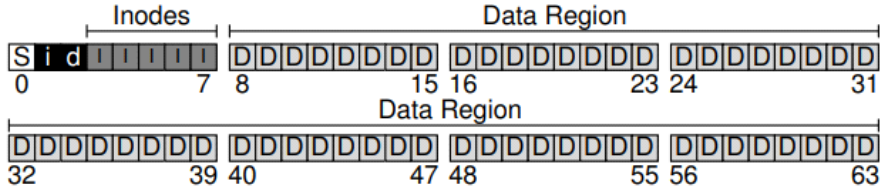
To accommodate the inodes, we'll need to reserve some space on the disk called **inode table**, which simply holds an array of on-disk inodes.

Another primary component is called **allocation structures**, which track whether inodes or data blocks are free or allocated.

We choose a simple structure known as a **bitmap**, one for the data region (the **data bitmap**), and one for the inode table (the **inode bitmap**).
In a bitmap, each bit is used to indicate whether the corresponding object/block is free (0) or in-use (1).

The last block is reserved for the **superblock**, which 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.

The final visual representation of the file system:



S: superblock
i: inode bitmap
d: data bitmap
I: inode table
D: data region

40.3 File Organization: The Inode
Each inode is referred to by a number (called the **i-number**), which we earlier called **low-level name** of the file.

Inside each node is virtually all of the information you need about a file:
- **type** (e.g. regular file, directory, etc.)
- **size**
- number of **blocks** allocated to it
- **protection information** (such as who owns the file, who can access it)
- some **time** information (including when the file was created, modified, or last accessed
- information about where its data blocks reside on disk (e.g. pointers of some kind)

This is all collectively called the file **metadata**.

One of the most important decisions in the design of the inode is how it refers to where data blocks are. One simple approach is to have one or more **direct pointers** (disk addresses) inside the inode; each pointer refers to one disk block that belongs to the file. However, this approach does not work with really big (e.g. bigger than the block size multiplied by the number of direct pointers in the inode) files.

Multi-Level Index
indirect pointer - 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. Thus, an inode may have some fixed number of direct pointers (e.g. 12) and a single indirect pointer.

For even larger files, just add another pointer to the inode: the **double indirect pointer**. This pointer refers to a block that contains pointers to indirect blocks, each of which contain pointers to data blocks.

40.4 Directory Organization

In vsfs (as in many file systems), directories have a simple organization; a directly basically just contains a list of pairs (entry name, inode numbers). For each file or directory, there is a string and a number in the data block(s) of the directory. For each string, there may also be a length (assuming variable-sized names).

Often, file systems treat directories as a special type of file. Thus, a directory has an inode, somewhere in the inode table (with the type field of the inode marked as "directory" instead of "regular file").

40.5 Free Space Management

A file system must track which inodes and data blocks are free, and which are not, so that when a new file or directory is allocated, it can find space for it. This is **free space management**.

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

40.6 Access Paths: Reading and Writing

Reading A File From Disk example

You want to simply open a file, read it, and then close it.

- when you issue an open("/foo/bar", O_RDNLY) call, the file system first need to find the inode for the file **bar** to obtain some basic info about the file (permission info, file size, etc.).
- the file must **traverse** the pathname and thus locate the desired inode.
- since all traversals begin at the **root directory** (called **/**), the first the FS will read from disk is the inode of the root directory.
- To find the inode, we must know its **i-number**, and we find the i-number of a file or directory in its parent directory.
- Once the inode is read in, the FS can look inside of it to find pointer of data blocks, which contain the contents of the root directory.
- The FS will thus use these on-disk pointers to read through the directory, looking for an entry for **foo**.
- After reading in one or however many directory entry blocks, it will find the entry for foo; and in turn, the FS will have found the inode number of foo.

- the next step is to recursively traverse the pathname until the desired inode is found
- the final step of **open()** is to read bar's inode into memory; the FS then does a final permissions check, allocates a file descriptor for this process in the per-process open-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 is deallocated.

The amount of I/O generated by the open is proportional to the length of the pathname.

Writing A File To Disk example

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

- unlike reading, writing to a file must also **allocate** a block.
- when writing out a new file, each write not only has to write data but has to first decide which block to allocate to the file and thus update other structures of the disk accordingly (e.g. the data bitmap and inode).
- thus, each write to a file logically generates five I/Os:
  1. one to read the data bitmap (which is then updated to mark the newly-allocated block as used)
  2. one to write the bitmap (to reflect its new state to disk)

3. two more to read and then write the inode (which is updated with the new block's location)
4. one to write the actual block itself.

the amount of write traffic is even worse when one considers a simple and common operation such as file creation.

- to create a file, the file system must not only allocate an inode, but also allocate space within the directory containing the new file.
- the total amount of I/O traffic to do so is quite high.
    1. one read to the inode bitmap (to find a free inode)
    2. one write to the inode bitmap (to mark it allocated)
    3. one write to the new inode itself (to initialize it)
    4. one to the data of the directory (to link the high-level name of the file to its inode number)
    5. one to read and write to the directory inode to update it\

40.7 Caching and Buffering

Since such simple operations like reading, writing, and creating a file make many I/O calls, we use memory (DRAM) to cache important blocks.

static partitioning

Early file system introduced **fixed-size cache** to hold popular blocks. Strategies such as **LRU** would decide which blocks to keep in cache. This fixed-size cache would usually be allocated at boot time to be roughly 10% of total memory.

This can be wasteful; what if the file system doesn't need 10% of memory at a given point in time?

dynamic partitioning

Many modern OS's integrate VM pages and file system pages into a **unified page cache**. In this way, memory can be allocated more flexibly across VM and file system, depending on which needs more memory at a given time.

file open example with caching

the first open may generate a lot of I/O traffic to read in directory inode and data, but subsequent file opens of that same file will mostly hit in the cache and thus no I/O is needed.

effect of caching on writes

whereas read I/O can be avoided altogether with a sufficiently large cache, write traffic has to go to disk in order to become persistent.

write buffering has a number of performance benefits

- first, by delaying writes, the file system can **batch** some updates into a smaller set of I/Os
    - e.g. if an inode bitmap is updated when one file is created and then updated moments later as another file is created, the file system saves an I/O by delaying the write after the first update.
- second, by buffering a number of writes in memory, the system can then **schedule** the subsequent I/Os and thus increase performance.
- finally, some writes are avoided altogether by delaying them
    - e.g. if an app creates a file and then deletes it, delaying the writes to reflect the file creation to disk **avoids** them entirely.

Modern OS's write in memory every 5-30 seconds. The trade-off: if the system crashes before the updates have been propagated to disk, the updates are lost; howeverm by keeping writes in memory longer, performance can be improved by batching, scheduling, and even avoiding writes.

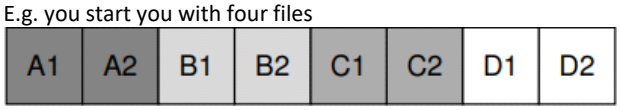
Chapter 41. Locality and The Fast File System

http://pages.cs.wisc.edu/~remzi/OSTEP/file-ffs.pdf
(week 8 lecture 2)

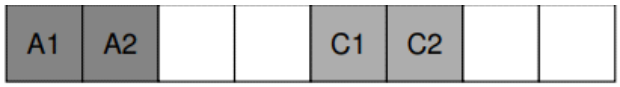
41.1 Poor Performance

The problem with the original file system was **poor performance**. The old UNIX file system treated the disk like it was a RAM; data was spread all over the place without regard to the fact that the medium holding the data was a disk, and thus had real and expensive positioning costs.

It also would end up getting **fragmented**, as the free space was not carefully managed.
E.g. you start you with four files



then B and D are deleted



then E is added

E gets spread across the disk and when accessing E, you don't get peak (sequential) performance.

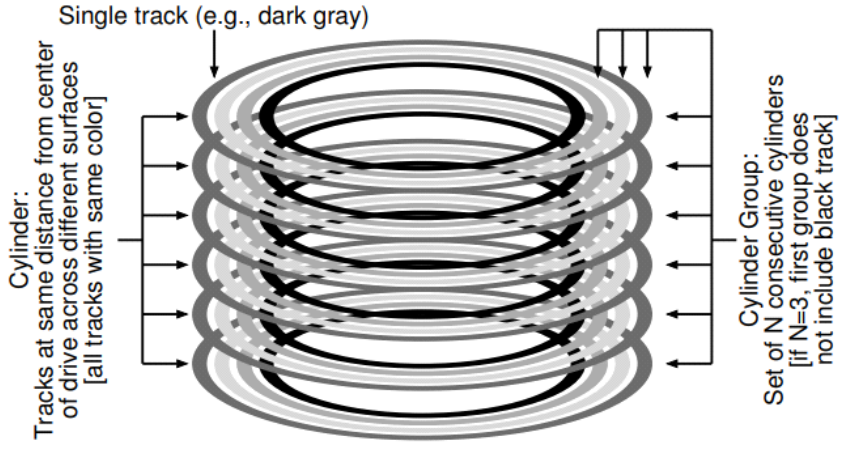
Disk **defragmentation** tools reorganize on-disk data to place files contiguously and make free space for one or a few contiguous regions, moving data around and then rewriting inodes and such to reflect the changes.

Smaller blocks were good because they minimized **internal fragmentation** but bad for transfer as each block might require a positioning overhead to reach it.

41.2 FFS: Disk Awareness is the Solution
The idea behind the **Fast File System (FFS)** was to design the file system structures and allocation policies to be "disk aware" and thus improve performance.

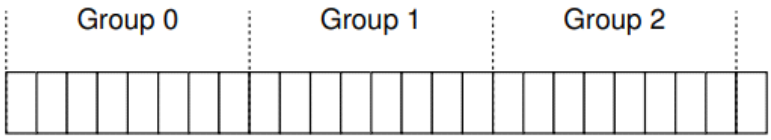
They kept the same APIs (open(), read(), write(), etc.), but changed the **internal implementation**.

41.3 Organizing Structure: The Cylinder Group
FFS divides the disk into a number of **cylinder groups**.
A single **cylinder** is a set of tracks on different surfaces of a hard drive that are the same distance from the center of the drive. FFs aggregates N consecutive cylinders into a group, and thus the entire disk can thus be viewed as a collection of cylinder groups.



^4 outermost tracks of a drive with 6 platters, and a cylinder group that consists of 3 cylinders.

Disks export a logical address space of blocks and hide details of their geometry from clients.
Thus, modern file systems organize the drive into **block groups**, each of which is just a consecutive portion of the disk's address space.



^every 8 blcoks are organized into a different block group. Critically, by placing two files within the same group, FFS can ensure that accessing one after the other will not result in long seeks across the disk.

To use these groups to store files and directories, FFS needs to have the ability to place files and directories into a group, and track all necessary information about them therein. To do so, FFS includes all the structures you might expect a file system to have within each group, e.g. space for inodes, data blocks, and some structures to track whether each of those are allocated or free.

- **super block (S)**: kept for reliability reasons; by keeping multiple copies, if one copy becomes corrupt, you can still mount the file system using a working replica.
- **inode bitmap (ib)** & **data bitmap (db)**: track whether the inodes and data blocks of the group are allocated.
- **inode and data block**: just like those in very-simply file system (VSFS)

41.4 Policies: How to Allocate Files and Directories

the basic mantra: *keep related stuff together, and keep unrelated stuff apart*.

The first step is the placement of directories: find the cylinder group with a low number of allocated directories and a high number of free inodes, and put the directory data and inode in that group.

For files, FFS does two things:

- first, it makes sure to allocate the data blocks of a file in the same group as its inode, thus preventing long seeks between inode and data
- second, it places all files that are in the same directory in the cylinder group of the directory they are in.

e.g. if a user creates four files, /a/b, /a/c, /a/d, and b/f, FFS would try to place the first three near one another (same group) and the fourth far away (in some other group).

41.6 The Large-File Exception

Large files placed within a block would entirely fill the block group and prevent "related" files from being placed within this block group, and thus may hurt file-access locality.

Therefore, for large files, FFS does the following:

- after some number of blocks are allocated into the first block group, FFS places the next "large" chunk of the file in another block group.
- Then, the next chunk of the file is placed in yet another different block group, and so on.

Instead of doing this:

```
group inodes     data
    0 /a-------- /aaaaaaaaa aaaaaaaaaa aaaaaaaaaa a---------
    1 ---------- ---------- ---------- ---------- ----------
    2 ---------- ---------- ---------- ---------- ----------
```

You do this:

```
group inodes     data
    0 /a-------- /aaaaa---- ---------- ---------- ----------
    1 ---------- aaaaa----- ---------- ---------- ----------
    2 ---------- aaaaa----- ---------- ---------- ----------
    3 ---------- aaaaa----- ---------- ---------- ----------
    4 ---------- aaaaa----- ---------- ---------- ----------
    5 ---------- aaaaa----- ---------- ---------- ----------
    6 ---------- ---------- ---------- ---------- ----------
```

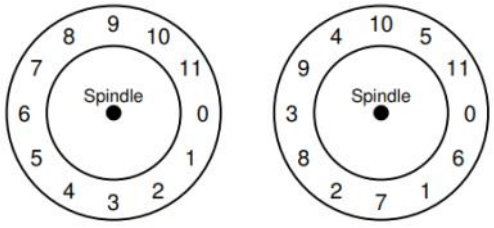
Spreading blocks of a file across the disk will hurt performance, particularly in the relatively common case of sequential file access. But with a large enough chunk size, the file system will spend most of its time transferring data from disk and just a little time seeking between chunks of the block.

This process of reducing an overhead by doing more work per overhead paid is called **ammortization**.

41.7 A Few Other Things About FFS

Small files back then were 2 KB and using 4 KB blocks meant about 50% **internal fragmentation**. The solution was to introduce **sub-blocks** which were 512-byte little blocks that the file system could allocate to files.

Another problem arose during sequential reads. FFS would first issue a read to block 0; ; by the time the read was complete, and FFS issued a read to block 1, it was too late: block 1 had rotated under the head and now the read to block 1 would incur a full rotation.



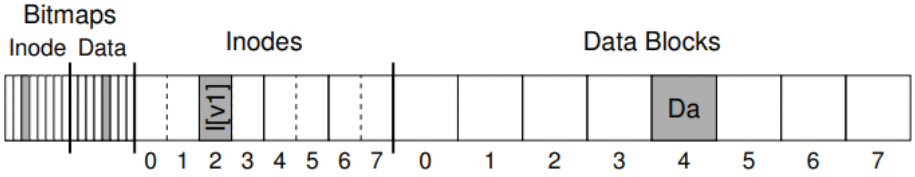
By skipping over every other block (in the example), FFS has enough time to request the next block before it went past the disk head. In fact, FFS was smart enough to figure out for a particular disk how many blocks it should skip in doing layout in order to avoid the extra rotations; this technique was called **parameterization**, as FFS would figure out the specific performance parameters of the disk and use those to decide on the exact staggered layout scheme.

Chapter 42. Crash Consistency: FSCL and Journaling

http://pages.cs.wisc.edu/~remzi/OSTEP/file-journaling.pdf
(week 8 lecture 2)

42.1 A Detailed Example

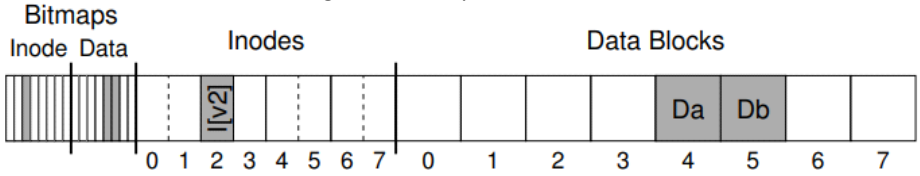
Appending to a file that already exists.



When we append to the file, we are adding a new data block to it, and thus must update three on-disk structures:

- the inode (which must point to the new block and record the new larger size due to the append)
- the new data block Db
- and a new version of the data bitmap to indicate that the new data block has been allocated

What we want is the final-disk image of the file system to look like this:



For this, we must perform three separate writes to the disk:

- one each for the inode (I[v2])

- bitmap (B[v2])
- data block (Db)

They don't happen immediately but rather stay in main memory (in the **page cache**) for some time first (e.g 5-30s). But then, the system crashes.

Crash Scenarios

Imagine only a single write succeeds; there are three possible outcomes:

- **Just the data block (Db) is written to disk**. In this case, the data is on disk, but there is no inode that points to it and no bitmap that even says the block is allocated. Thus, it is as if the write never occurred. This is not a problem for the file-system crash consistency, but it is a problem for the user (who just lost some data).
- **Just the updated inode (I[v2]) is written to disk**. In this case, the inode points to the disk address (5) where Db was about to be written, but Db has not yet been written there. Thus, if we trust that pointer, we will read **garbage** data from the disk (the old contents of disk address 5).
    - Further, we have a new problem, which we call a **file-system inconsistency**. The on-disk bitmap is telling us that data block 5 has not been allocated, but the inode is saying that it has. The disagreement between the bitmap and the inode is an inconsistency in the data structures of the file system; to use the file system, we must somehow resolve this problem (more on that below).
- **Just the updated bitmap (B[v2]) is written to disk**. In this case, the bitmap indicates that block 5 is allocated, but there is no inode that points to it. Thus the file system is inconsistent again; if left unresolved, this write would result in a **space leak**, as block 5 would never be used by the file system.

There are also three more crash scenarios in this attempt to write three blocks to disk. In these cases, two writes suceed and the last one fails:

- **The inode (I[v2]) and bitmap (B[v2]) are written to disk, but not data (Db)**. In this case, the file system metadata is completely consistent: the inode has a pointer to block 5, the bitmap indicates that 5 is in use, and thus everything looks OK from the perspective of the file system's metadata. But there is one problem: 5 has garbage in it again.
- **The inode (I[v2]) and the data block (Db) are written, but not the bitmap (B[v2]).** In this case, we have the inode pointing to the correct data on disk, but again have an inconsistency between the inode and the old version of the bitmap (B1). Thus, we once again need to resolve the problem before using the file system.
- **The bitmap (B[v2]) and data block (Db) are written, but not the inode (I[v2])**. In this case, we again have an inconsistency between the inode and the data bitmap. However, even though the block was written and the bitmap indicates its usage, we have no idea which file it belongs to, as no inode points to the file.

The Crash Consistency Problem describes the many problems that can occur to our on-disk file system image because of crashes: we can have

- inconsistency in file system data structures
- space leaks
- return garbage data to a user

42.2 Solution #1: The File System Checker

fsck (fixing the inconsistencies later when rebooting)

This approach fails for the case where there is no inconsistency but the inode points to garbage data. The only real goal is to make sure the file system metadata is internally consistent.

Summary of what fsck does:

- **superblock**: fsck first checks if the superblock looks reasonable, mostly doing sanity checks such as making sure the file system size is greater than the number of blocks that have been allocated. Usually the goal of these sanity checks is to find a suspect (corrupt) superblock; in this case, the system (or admin) may decide to use an alternate copy of the superblock.
- **free blocks**: Next, fsck scans the inodes, indirect blocks, double indirect blocks, etc., to understand which blocks are currently allocated within the file system. It uses this knowledge to produce a correct vision of the allocation bitmaps; so if there is any inconsistency between bitmaps and inodes, it is resolved by trusting the information within the inodes.
- **inode state**: each inode is checked for corruption or other problems. If there are problems with the inode fields that are not easily fixed, the inode is considered suspect and cleared by fsck; the inode bitmap is correspondingly updated.
- **inode links**: fsck also verifies the link count of each allocated inode (the link count indicates the number of different directories that contain a reference to this particular file). To verify the link count, fsck scans through the entire directory tree, starting at the root, and builds its own link counts

for every file and directory in the file system. If there is a mismatch, action is taken to correct the problem.
- **duplicates**: fsck also checks for duplicate pointers (i.e. cases where two different inodes refer to the same block).
- **bad blocks**: it also checks for bad block pointers while scanning through the list of all pointers.
- **directory checks**: fsck performs additional integrity checks on the contents of each directory, making sure "." and ".." are the first entries, that each inode referred to ina directory entry is allocated, and ensuring that no directory is linked to more than once in the entire hierarchy.

42.3 Solution #2: Journaling (or Write-Ahead Logging)

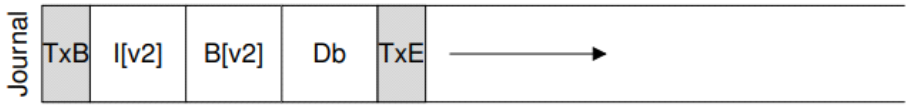
Basic idea: when updating the disk, before overwriting the structures in place, first write down a little node (somewhere else on disk) describing what you are about to do.

Linux ext3 journaling file system

Data Journaling

Example: we wish to write the inode (I[v2]), the bitmap (B[(v2]), and data block (Db) to disk again.

This is what the journal looks like:



The transaction begin (TxB) tells us about this update, including information about the pending update to the file system, andsome kind of **transaction identifier (TID)**. The middle blocks contain the exact content we are updating called **physical logging**. The final block (TxE) marks the end of this transaction and will also contain the TID.

Once the transaction is safely on disk, we are ready to overwrite the old structures in the file system in a process called **checkpointing**.

Trying to write the set of blocks in the transaction (e.g., TxB, I[v2], B[v2], Db, TxE) to disk, what if system crashes?
- issue each one at a time, waiting for each to complete, and then issuing the next
  - **slow**
- issue all five block writes at once, as this would turn five writes into a single sequential write and be faster
  - **unsafe**: given such a big write, the disk internally may perform scheduling and complete small pieces of the big write in any order.

to avoid this problem, the file system issues the transactional write in two steps:
- first, it writes all blocks except the TxE block to the journal **all at once**.
- when those writes complete, the file system issues the write of the TxE block.

Thus, the protocol to update the file system becomes:
1. **Journal write**: write the content sof the transaction (including TxB, metadata, and data) to the log, wait for these writes to complete.
2. **Journal commit**: write the transaction commit block (containing TxE) to the log, wait for the write to complete, transaction is said to be **commited**.
3. **Checkpoint**: write the contents of the update (metadata and data) to their final on-disk locations.

Recovery

If the crash happens after the transaction has committed to the log, but before the checkpoint is complete, the file system can **recover**.
- when the system boot, the FS recovery process will scan the log and look for transactions that have committed to the disk
- these transactions are **replayed** with the FS again attempting to write out the blocks in the

transaction to their final on-disk locations
^this is called **redo logging**.

Batching Log Updates
With journaling, we logically commit all of this information to the journal for each of our two file creations; because the files are in the same directory, and assuming they even have inodes within the same inode block, this means that if we're not careful, we'll end up writing these same blocks over and over.

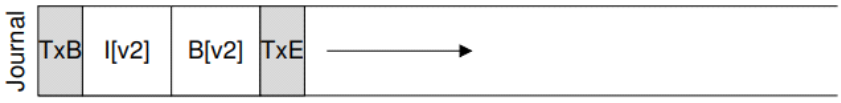
To help, some FS do not commit each update to disk one at a time; rather, one can buffer all updates into a global transaction.

Making the Log Finite
When the log keeps getting larger, it reaches a point where no further transactions can be commited to the disk. To address this, journaling FS treat the log as a **circular data structure**, re-using it over and over.

Journaling becomes:
1. **Journal write**: write the content sof the transaction (including TxB, metadata, and data) to the log, wait for these writes to complete.
2. **Journal commit**: write the transaction commit block (containing TxE) to the log, wait for the write to complete, transaction is said to be **commited**.
3. **Checkpoint**: write the contents of the update (metadata and data) to their final on-disk locations.
4. **Free**: some time later, mark the transaction free in the journal by updating the journal superblock.

We are still writing each data block to the disk twice, which is expensive.

Metadata Journaling
Same as above (**data journaling**), but the user is not writing to the journal.



The data block Db would instead be written to the file system, avoiding the extra write. Given that most I/O traffic to the disk is data, not writing data twice substantially reduces the I/O load of journaling.

Writing the Db to the block after the logging/checkpointing can result in issues. If system crashes after I[v2] and B[v2] are checkpointed but Db doesn't get written, I[v2] points to garbage.

Thus, FS write data blocks Db to the disk **first** before related metadata is written to disk.

1. **Data write**: write the content sof the transaction (including TxB, metadata, and data) to the log, wait for these writes to complete.
2. **Journal metadata write**: write the begin block and metadata to the log; wait for writes to complete.
3. **Journal commit**: write the transaction commit block (containing TxE) to the log, wait for the write to complete, transaction is said to be **commited**.
4. **Checkpoint metadata**: write the contents of the update (metadata and data) to their final on-disk locations.
5. **Free**: some time later, mark the transaction free in the journal by updating the journal superblock.

Tricky Case: Block Reuse
tltltltltltltltltltltltltltltlt

Chapter 43. Log-structured File Systems
http://pages.cs.wisc.edu/~remzi/OSTEP/file-lfs.pdf
(week 8 lecture 2)

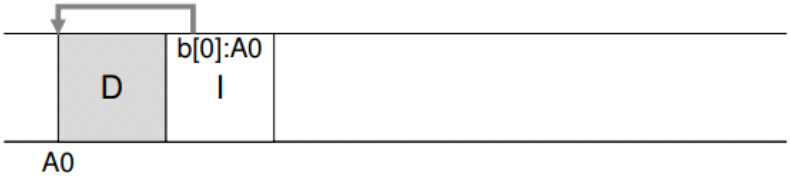
Motivation for this FS was the following:
- **System memories are growing**: as memory gets bigger, more data can be cached in memory, as more data is cached, disk traffic increasingly consists of writes, as reads are serviced by the cache. Thus, FS performance is largely determined by its write performance.
- **There is a large gap between random I/O performance and sequential I/O performance**: Hard-drive transfer bandwidth has increased a great deal over the

years [P98]; as more bits are packed into the surface of a drive, the bandwidth when accessing said bits increases. Seek and rotational delay costs, however, have decreased slowly; it is challenging to make cheap and small motors spin the platters faster or move the disk arm more quickly. Thus, if you are able to use disks in a sequential manner, you gain a sizeable performance advantage over approaches that cause seeks and rotations.

- **Existing file systems perform poorly on many common workloads**: e.g. FFS would perform a large number of writes to create a new file of size one block. Thus, although FFS places all of these blocks within the same block group, FFS incurs many short seeks and subsequent rotational delays and thus performance falls far short of peak sequential bandwidth.
- **File systems are not RAID-aware**: e.g. both RAID-4 and RAID-5 have the **small-write problem** where a logical write to a single block causes 4 physical I/Os to take place. Existing file systems do not try to avoid this worst-case RAID writing behavior.

Log-structured File System (LFS):
When writing to disk, LFS first buffers all updates (including metadata) in an in-memory **segment**; when the segment is full, it is written to disk in one long, sequential transfer to an unused part of the disk. LFS never overwrites exisiting data, but rather always writes segments to free locations.

41.1 Writing to Disk Sequentially
Basic idea:
Imagine we are writing a data block D to a file. However, when a user writes a data block, it is not only data that gets written to disk, but also **metadata** (**inode I**)



^it look sumtin lak dis

43.2 Writing Sequentially And Effectively
Simply writing to disk in sequential order is not enough to achieve peak performance. You must issue a large number of **contiguous** writes (or one large write) to the drive in order to achieve good write performance.

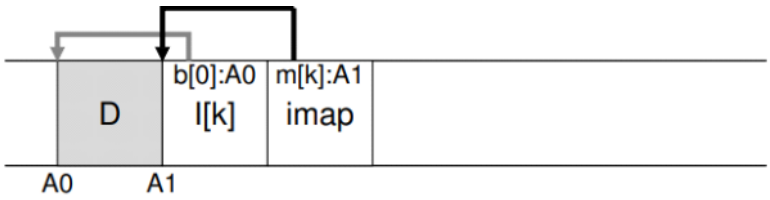
To achieve this, LFS uses **write buffering**
Before writing to disk, LFS keeps track of updates in memory; when it has received a sufficient number of updates, it writes them disk all at once, thus ensuring efficient use of the disk.

The large chunk of updates LFS writes at one time is referred to by a **segment**. As long as the segment is large enough, these writes will be efficient.

43.3 How Much To buffer?
How many updates should LFS buffer before writing to disk?
Every time you write, you pay a fixed overhead of the positioning cost. Thus, how much do you have to write to **amortize** that cost? The more you write, the better, and the closer you get to achieving peak bandwidth.

43.5 Solution Through Indirection: The Inode Map
inode map (imap)
The imap is a structure that takes an inode number as input and produces the disk address of the most recent version of the inode. Thus, you can imagine it be implemented as a simple *array*. Any time an inode is written to disk, the imap is updated with its new location.

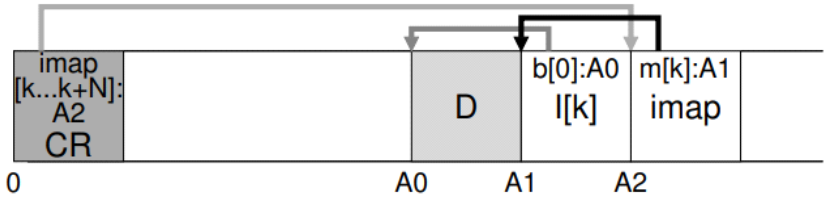
LFS places chunks of the inode map right next to where it is writing all of the other new information. Thus, when appending a data block to a file k, LFS actually writes the new data block, its inode, and a piece of the inode map all together onto the disk.



43.6 Completing the Solution: The Checkpoint Region

How do we find the inode map, now that pieces of it are also now spread across the disk?

LFS has a fixed place on disk for this, known as the **checkpoint region (CR)**. The CR contains pointers to the latest pieces of the inode map, and thus the inode map pieces can be found by reading the CR first.



43.7 Reading A File From Disk: A Recap

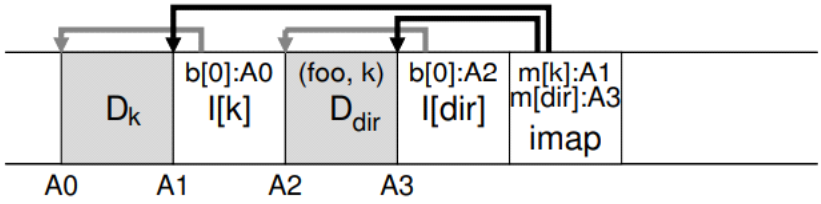
Assume you have nothing in memory to begin.

The first on-disk data structure we must read is the checkpoint region. The checkpoint region contains pointers to the entire inode map, and thus LFS then reads in the entire inode map and caches it in memory. After this point, when given an inode number of a file, LFS simply looks up the inode-number to inode-disk-address mapping in the imap, and read sin the most recent version of the inode. To read a block from the file, at this point, LFS proceeds exactly as a typical UNIX file system, by using direct pointers or indirect pointers or doubly-indirectly pointers as need be.

43.8 What About Directories?

Fortunately, directory structure is basically identical to classic UNIX file systems, in that a directory is just a collection of (name, inode number) mappings.

For example, when creating a file on disk, LFS must both write the new inode, some data, as well as the directory data and its inode that refer to this file. Remember that LFS will do so sequentially on the disk. Thus, creating a file **foo** in a direcoty would lead to following structures on disk:



The piece of the inode map contains the info for the location of both the directory file **dir** as well as the newly-created file **f**. Thus, when accessing file **foo** (with inode number k), you would first look in the inode map (usually cached in memory) to find the location of the inode of direcotry **dir.** You would then read the directory inode, which gives you the location of the directory data; reading this data block gives you the name-to-inode number mapping of (foo, k). You then consult the inode map again to find the location of the inode number k, and finally read the desired data block at address.

**idea how important the following is tbh:*

Another serious problem with LFS that inode map solves is the **recursive update problem**. The problem arises in any file system that never updates in place, but rather moves updates to new locations on the disk.

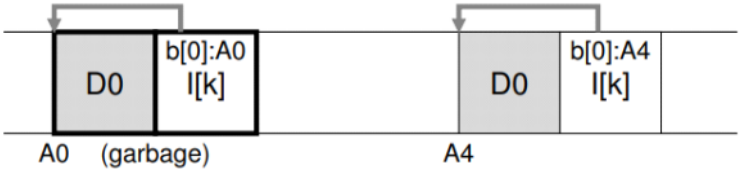
Specifically, whenever an inode is updated, its location on disk changes. If we hadn't been careful, this would have also entailed an update to the directory that points to this file,

which then would have mandated a change to the parent of that directory, and so on, all the way up the file system tree. LFS cleverly avoids this problem with the inode map. Even though the location of an inode may change, the change is never reflected in the directory itself; rather, the imap structure is updated while the directory holds the same name-to-inode-number mapping. Thus, through indirection, LFS avoids the recursive update problem

43.9 Garbage Collection

LFS repeatedly writes the latest version of a file (including its inode and data) to new locations on disk. This implies that LFS leaves old versions of file structures scattered throughout the disk: **garbage**.

Imagine there is an exisiting file referred to by inode number k, which points to a single data block D0. We now update the block, generating both a new inode and a new data block.



^both the inode and data blocks have two versions on disk, one old (left) and one current (**live**) (right). By the simple act of logically updating a data block, a number of new structures must be persisted by LFS, thus leaving old versions of said blocks on disk.

Another example, imagine we instead append a block to that original file k. A new version of the inode is generated, but the old data block is still pointed to by the inode. Thus, it still still live and part of the current file system.

To address this, LFS keeps only the latest live version of a file; thus, LFS must periodically find these old dead versions of file data, inodes, and other structures, and **clean** them. This is called **garbage collection**.

LFS works on a segment-by-segment basis of doing its garbage collection by keeping only the latest live version of a file.

It would leave free **holes** mixed between allocated space on disk. Write performance would drop because LFS would not be able to find large contiguous region to write to disk sequentially with high performance.

Instead, LFS works on a segment-by-segment basic, thus clearing up large chunks of space for subsequent writing. Specifically, the clean reads in M exisiting segments, **compacts** their contents into N new segments (N<M) and then writes the N segments to disk in new locations. The old M segments are then free and can be used by the FS for subsequent writes.

43.10 Determining Block Liveness

To determine whethere a data block D is live, LFS adds more info to each segment that describes each block.

It includes its inode number (which file is belongs to) and its offset (which block of the file this is). This is recorded in the structure at the head of the segment called **segment summary block**. With this, you can determine whether a block is live or dead.

Pseudocode:

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

43.11 A Policy Question: Which Blocks To Clean, And When?

Determining when to clean is easier; either periodically (during idle time) or when you have to because the disk is full.

In determining which blocks to clean, we use **hot** and **cold** segmnets.

- A **hot** segment is one in which the contents are being frequently over-written
  - for this segment, it is better to wait a long time before cleaning it
- A **cold** segment may have a few dead blocks but the rest of its contents are relatively stable
  - one should clean cold segments sooner than hot segments later.

43.12 Crash Recovery And The Log

Chapter 44. Flash-Based SSDs

http://pages.cs.wisc.edu/~remzi/OSTEP/file-ssd.pdf
(week 8 lecture 2)

Solid-state storage has no mechanical or moving parts like hard drives; rather, they are simply build out of transistors, much like memory and processors. However, unlike RAM, SSDs retain information despite power loss.

44.1 Storing a Single Bit

In a **single-level cell (SLC)** flash, only a single bit is stored within a transistor; with **multi-level cell (MLC)** flash, two bits are encoded into different levels of charge (e.g. 00, 01, 10, 11) and are low, somewhat low, somewhat high, high levels.

44.2 From Bits to Banks/Planes

Banks

A bank is access in two differnet sized units:

- **blocks**, which are typically of size 128 KB or 256 KB
- **pages**, which are a few KB (e.g. 4 KB)

Within each bank there is a large number of blocks; within each block, there are a large number of pages.

44.3 Basic Flash Operations

- **Read (a page)**: a client can read any page by specifying the read command and appropriate page number of the device. Being able to access any location uniformly quickly means the device is a **random access** device.
- **Erase (a block)**: before writing to a page within a flash, you must first **erase** the entire **block** the **page** lies in. Once finished, the entire block is reset and each page is ready to be programmed.
- **Program (a page)**: once a block has been erased, the program command can be used to change some of the 1s within a page to 0s and write the desired contents of a page to the flash.

Each page has a **state** associated with it.

- Pages start in an **INVALID** state.
- by erasing the block that page resides within, you set the page to **ERASED**, meaning you can now program the page.
- when you program a page, its state changes to **VALID**, meaning its contents have been set and can be read.

A Detailed Example

You have 4 8-bit pages in a 4-page block.
Initially, all pages in the block have already been programmed.

Page 0	Page 1	Page 2	Page 3
00011000	11001110	00000001	00111111
VALID	VALID	VALID	VALID

You then want to write to Page 0, so you have to ERASE the pages in the block

Page 0	Page 1	Page 2	Page 3
11111111	11111111	11111111	11111111
ERASED	ERASED	ERASED	ERASED

You can now program Page 0 as need be. **Note**: Data in pages 1, 2, and 3 also got ERASED, so it is important to copy them elsewhere before you ERASE the block.

Page 0	Page 1	Page 2	Page 3
00000011	11111111	11111111	11111111
VALID	ERASED	ERASED	ERASED

44.4 Flash Performance and Reliability

One reliability problem within flash chips is called **disturbance**, when accessing a particular page within a flash, it is possible that some bits get flipped in neighboring pages.

44.5 From Raw Flash to Flash-Based SSDs

Internally, an SSD consists of some number of raw flash chips. It also contains some amount of volatile memory; such memory is useful for caching and buffering of data as well as for mapping tables. Finally, an SSD contains control logic to orchestrate device operation.

The **flash translation layer (FTL)** takes read and write requests of **logical blocks** (that comprise the device interface) and turns them into low-level read, erase, and program commands on the underlying **physical blocks** and **physical pages** (that comprise the actual flash device).

To get excellent performance, we use multiple flash chips in **parallel**, reduce **write amplification**, and **wear level**

44.6 Bad Approach

Direct-mapping is very costly for performance (for every write: read, erase, program) and not reliable as popular blocks will quickly wear out.

44.7 Log-Structured FTL

Logging: upon a write to logical block N, the device appends the write to the next free spot in the currently-being-written-to block.
To allow for subsequent reads of block N, the device keeps a **mapping table** which stores the physical address of each logical block in the system.

Example:
the client is reading or writing 4-KB sized chunks, and that the SSD contains some large number of 16-KB sized blocks, each divided into four 4-KB pages.
The client issues the following:

- Write(100) with contents a1
- Write(101) with contents a2
- Write(2000) with contents b1
- Write(2001) with contents b2

The **logical block addresses** (e.g. 100, 101) are used by the client of the SSD (e.g. a file system) to remember where info is located.

Internally, the device must transform these block writes into the erase and program operations supported by the raw hardware, and somehow record, for each logical block address, which **physical page** of the SSD stores its data.

Initially, the state of our SSD is INVALID, so to write to it, the block (to be written in) first needs to be ERASED.

What if the client wants to **read** logical block 100?
The SSD must transform a read issued to logical blocks 100 inti physical page 0.

The SSD finds a location for the write, usually just picking the next free page; it then programs that page with the block's contents, and records the logical-to-physical mapping in its mapping table. Subsequent reads simply use the table to **translate** the logical block address presented by the client into the physical page numebr required to read the data.

44.8 Garbage Collection

Imagine the same example as above, but now blocks 100 and 101 are written to again.

The process of finding garbage blocks and reclaiming them for future use is called **garbage collection**.

- find the block that contains one or more garbage pages
- read in the live (non-garbage) pages from that block
- write out those live pages to the log
- reclaim the entire block for use in writing

There must be enough information within each block to enable the SSD to determine whether each page is live or dead. One way to do this is to store, at some location within each block, information about which logical blocks are stored within each page. The device can then use the mapping table to determine whether each page within the block holds live data or not.

44.9 Mapping Table Size

Block-Based Mapping

One approach to reduce the costs of mapping is to only keep a pointer per **block** of the device, instead of per **page**, reducing the amount of mapping info.

This doesn't work well for performance reasons. Namely, with **small-writes**.

With block-level mapping, the FTL only needs to record a single address translation for all of this data. The address mapping is chopped into chunks that are the size of the physical blocks within the flash. Thus, the logical block address consists of two portions: a **chunk number** and an **offset**. Because we are assuming four logical blocks fit within each physical block, the offset portion of the logical addresses requires 2 bits; the remaining (most significant) bits form the chunk number.

Reading with FTL

- first, the FTL extracts the chunk number from the logical block address presented by the client, by taking the topmost bits out of the address
- then, the FTL looks up the chunk-number to physical-page mapping in the table
- finally, FTL computes the address of the desired flash page by adding the offset from the logical address to the physical address of the block.

While block level mappings greatly reduce the amount of memory needed for translations, they cause significant performance problems when writes are smaller than the physical block size of the device; as real physical blocks can be 256KB or larger, such writes are likely to happen quite often.

Hybrid Mapping

With this approach, the FTL keeps a few blocks erased and directs all writes to them, called **log blocks**. Because the FTL wants to be able to write any page to any location within the log block without all the copying required by a pure block-based mapping, it keeps **per-page** mappings for these log blocks.

The FTL has two types of mapping tables in its memory:

- a small set of per-map mappings (called **log table**)
- a larger set of per-block mapping in the **data table**

When looking for a particular logical block:

- the FTL will first consult the **log table**
    - if the logical block's location is not found there, the FTL will then consult the **data table** to find its location and then access the requested data.

The log blocks need to be small. To do so, the FTL has to periodically examine log blocks (which have a pointer per page) and **switch** them into blocks that can be pointed to by only a single block pointer.

Read pages 14-16 to see the example *if you have time*

Page Mapping Plus Caching

A simple way is to just cache only the active parts of the FTL in memory, thus reducing the amount of memory needed.

If a workload only accesses a small set of pages, the translations of those pages will be stored in the in-memory FTL, and performance will be excellent without high memory cost. However, if memory cannot contain the **working set** of necessary translations, each access will minimally require an extra flash read to first bring in the missing mapping before being able to access the data itself.

44.10 Wear Leveling

Because multiple erase/program cycles will wear out a flash block, the FTL should try its best to spread that work across all the blocks of the device evenly. In this way, all blocks will wear out at roughly the same time.

44.11 SSD Performance And Cost

Performance

The biggest difference in performance, as compared to disk drives, is realized when performing random reads and writes.

ASIDE: KEY SSD TERMS

- A **flash chip** consists of many banks, each of which is organized into **erase blocks** (sometimes just called **blocks**). Each block is further subdivided into some number of **pages**.
- Blocks are large (128KB–2MB) and contain many pages, which are relatively small (1KB–8KB).
- To read from flash, issue a read command with an address and length; this allows a client to read one or more pages.
- Writing flash is more complex. First, the client must **erase** the entire block (which deletes all information within the block). Then, the client can **program** each page exactly once, thus completing the write.
- A new **trim** operation is useful to tell the device when a particular block (or range of blocks) is no longer needed.
- Flash reliability is mostly determined by **wear out**; if a block is erased and programmed too often, it will become unusable.
- A flash-based **solid-state storage device** (**SSD**) behaves as if it were a normal block-based read/write disk; by using a **flash translation layer** (**FTL**), it transforms reads and writes from a client into reads, erases, and programs to underlying flash chips.
- Most FTLs are **log-structured**, which reduces the cost of writing by minimizing erase/program cycles. An in-memory translation layer tracks where logical writes were located within the physical medium.
- One key problem with log-structured FTLs is the cost of **garbage collection**, which leads to **write amplification**.
- Another problem is the size of the mapping table, which can become quite large. Using a **hybrid mapping** or just **caching** hot pieces of the FTL are possible remedies.
- One last problem is **wear leveling**; the FTL must occasionally migrate data from blocks that are mostly read in order to ensure said blocks also receive their share of the erase/program load.

Chapter 45. Data Integrity and Protection

http://pages.cs.wisc.edu/~remzi/OSTEP/file-integrity.pdf
(week 8 lecture 2)

45.1 Disk Failure Modes

Two types of single-block failures are common:

- **latent-sector errors (LSEs)**
- **block corruption**

LSEs happen when a disk sector (or group of sectors) has been damaged in some way (e.g. disk head touches the surface i.e. **head crash**). In-disk **error correcting codes (ECC)** are used by the driver to determine them; if they are not good, and the drive does not have enough info to fix the errors, the disk will return an error when a request is ussed to read them.

Corruption can occur (e.g. buggy disk firmware may write a block to the wrong location). In such a case, the disk ECC indicates the block contents are fine, but from the client's perspective the wrong block is returned when subsequently accessed. This is a **silent fault**; the disk gives no

indication of the problem when returning the faulty data.

Some additional findings about LSEs are:
- Costly drives with more than one LSE are as likely to develop additional errors as cheaper drives
- For most drives, annual error rate increases in year two
- The number of LSEs increase with disk size
- Most disks with LSEs have less than 50
- Disks with LSEs are more likely to develop additional LSEs
- There exists a significant amount of spatial and temporal locality
- Disk scrubbing is useful (most LSEs were found this way)

Some findings about corruption:
- Chance of corruption varies greatly across different drive models within the same drive class
- Age effects are different across models
- Workload and disk size have little impact on corruption
- Most disks with corruption only have a few corruptions
- Corruption is not independent within a disk or across disks in RAID
- There exists spatial locality, and some temporal locality
- There is a weak correlation with LSEs

45.2 Handling Latent Sector Errors

When a storage system tries to access a block, and the disk returns an error, the storage system should simply use whatever redundancy mechanism it has to return the correct data.

E.g.
- in mirrored RAID, the system should access the alternate copy.
- in a RAID-4 or RAID-5 system based on parity, the system should reconstruct the block from the other blocks in the parity group.

45.3 Detecting Corruption: The Checksum

Detection of corruption is the key problem here.

Checksum

A checksum is simply the result of a function that takes a chunk of data as input and computes a function over said data, producing a small summary of the contents of the data. The summary is the checksum.

Common Checksum Functions

- **XOR**
  XORs each chunk of the data block being checksummed, producing a single value that represents the XOR of the entire block.

  Limitation: e.g. if two bits in the same position within each checksummed unit change, the checksum will not detect the corruption.

- **Addition**
  fast; uses 2's complement addition over each chunk of the data, ignoring overflow.

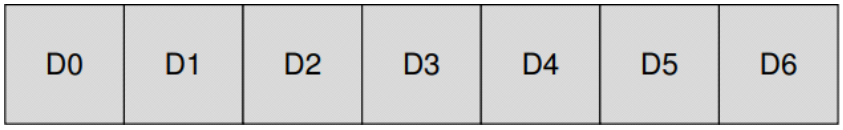
- **Fletcher Checksum**
  two check bytes, s1 and s2. block D consists of bytes d1-dn; $s1 = (s1 + d_i)\%255$ and $s2 = (s2 + s1)\%255$

- **Cyclic Redundancy Check (CRC)**
  Over a data block D, treat D as if it is a large binary number and divide it by an agreed upon value (k). The remainder of this division is the value of the CRC.

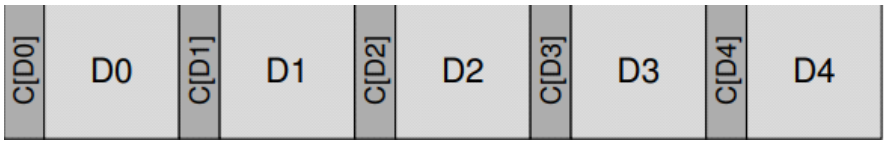
Checksum Layout

How should the checksum be stored on disk?

Without checksums:



With checksums:



For disks that have the ability, drive manufacturers formatted the drives with 520-byte secotrs (instead of 512); an extra 8 bytes per sector to store the checksum.

For disks that can't be resized like above:

In this scheme, the n checksums are stored together in a sector, followed by n data blocks, followed by another checksum sector fot he next n blocls, and so forth.

45.4 Using Checksums

When reading a block D, the client (i.e. file system) also reads its checksum from disk Cs(D), which is called the **stored checksum**. The client then computes the checksum over the retrieved block D, which is called **computed checksum** Cc(D).

At this point the client compares the stored and computed checksums.

- if they are equal (i.e. Cs(D) == Cc(D)) the data has likely not been corrupted
- if they don't match, the data has been changed since the time it was stored.

45.5 A New Problem: Misdirected Writes

Misdirected writes arise in disks and RAID controllers which write the data to disk correctly, except in the **wrong** location. In a single-disk system, this means that the disk wrote block Dx not to address x (as desired) but rather to address y (thus "corrupting" Dy); in addition, within a multi-disk system, the controller may also write Di,x not to address x of disk i but rather to some other disk j.

We fix this by adding more info to each checksum: a **physical identifier (physical ID)**

44.6 One Last Problem: Lost Writes

lost writes occur when the device informs the upper layer that a write has completed but in fact it never is persisted. So what remains is the old contents of the blcok rather than the updated new contents.

Solutions: **write verify** or **read-after-write**; by immediately reading back the data after a write, a system can ensure that the data indeed reached the disk surface.

44.7 Scrubbing

by periodically reading through every block of the system, and checking whether checksums are still valid, the disk system can reduce the chances that all copies of a certain data item become corrupted.

44.8 Overheads of Checksumming

Two kinds: space and time.