CS 111: Operating Systems
Lecture Notes

Henry Genus

Fall 2018

Contents

1 Introduction to Systems

```
$ ls -l big
-rw-r--r-- 1 eggert faculty 9000000000000000000 Sep 21 11:31 big
$ grep x big
$ time grep x big
real 0m0.009s
```

The grep command here analyzes roughly 1021 bytes/second... which is unrealistically fast! How is this possible?..... big is a sparse file! (generated from $ truncate ... big) grep "cheated", in a sense, by knowing the file was empty.

However...

```
$ grep -r
```

does not skip sparse files, so it can be stalled... and it is used by the NSA!

Clearly, Operating System choices are important.

How do we define an Operating System?

- American Heritage; 4th Edition (2000) := Software designed to control the hardware of a specific data processing system input and output to allow users and applications to make use. Note that this definition wouldn't include Linux, since it works on most hardware!
- Encarta (2007) := the master control program in a computer
- Wikipedia v. 917297650 := system software that manages computer hardware and software resources and provides common services for computer programs

Thus we can see that the definition has moved from control to user-environment interaction facilitation We can get our goals from this information!

What are the goals of an Operating System?

- External Goals:
  - protection (from hackers and bugs)
  - robustness (in unforeseen circumstances)
  - utilization (time wise; always working)
  - performance (time/memory/energy)
- Internal Goals:
  - simplicity (complexity = cost)
  - flexibility (easy to mutate)

BUT systems have their downsides...

1. TRADEOFFS
   - *Waterbed Effect* — fixing a problem causes another unrelated one
   - *Binary Classification* — when we must classify binary items by proxy. Less errors of one type means more of another!
   - Say we want to sort 6 million records of 1024 bytes. 6 million records take 6GB! We need to be more space efficient! SO we use a pointer array! Copying is now 2^7 times faster, but we sacrificed simplicity! (we must edit our software to use pointers)

- The moral of the story is NOTHING IS FREE

2. INCOMMENSURATE SCALING
   - Not all parts of a system grow at the same rate.
   - There are two common types:
     - *Economies of Scale*: Big = Fast (think: factory)
     - *Diseconomies of Scale*: Small = Fast (think: STAR network)
3. EMERGENT PROPERTIES
   - Systems sometimes have properties that are not present in any individual members
   - ex) Napster and Torrents: UCLA received a network speed boost, but this resulted in an increase in music torrenting!
4. PROPAGATION OF EFFECTS
   - ex) Shift-JIS — str: ab練c = a | b | / | ... | c
     Japanese characters had to be modeled as two byte characters, but some filenames failed since the arbitrary bits could look like a slash! We cannot always predict the consequences of a chance in all areas of the system.

More broadly, systems introduce a lot of complexity. Complexity is hard to define, so we look for benchmarks:

(i) a large number of components

(ii) a large number of interconnections

(iii) a large number of irregularities

(iv) a team of designers, implementers, or maintainers

(v) a long *Kolmogorov Complexity*
Kolmogorov Complexity := the length of the shortest description.

Not all signs are necessary for a system to be complex!

We can generalize our problems to a few major Operating System goals:

- virtualization
- concurrency
- persistence

(and if it were up to Eggert)

- evolution
- flexibility

Notice that these are contradictory... especially eggert's!

But why even use an OS? Well...let's try to make an application without one!

2 Bare-Metal Application

A paranoid professor wants to develop a program that displays the word count of typed input.

SPECS

- desktop (non-network)
- word = [A-Za-z]+
- output = [0-9]+
- program runs on boot

How does booting work?

x86 Boot Procedure

CPU RAM is cleared on reset... how can we load the program?

We generally use EEPROM, but EEPROM is expensive and static, so we can't hold our program or kernel there. Additionally in desktops, it is read-only and hold both firmware and the location of the software to bootstrap.

We use *GUID (Globally Unique Identifier)* to identify disk partitions

- 128-bit quantity
- without these IDs, firmware won't know if it changed or not
- held in GUID partition table (GPT)

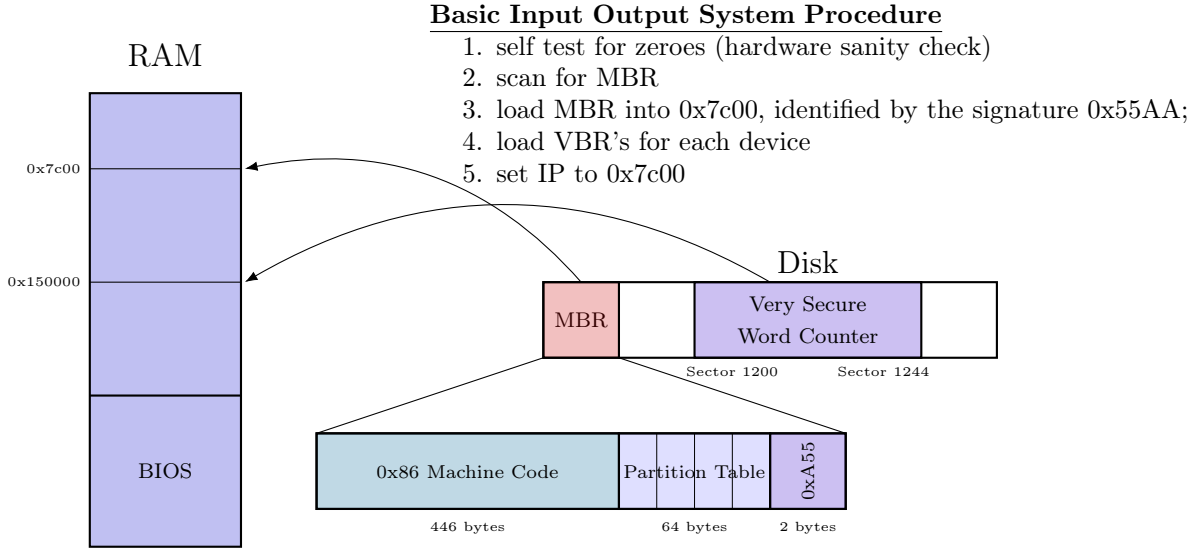
The basic bootstrap loading process is:

1. all registers start at 0 (which means kernel mode)
2. BIOS (firmware) sets up from the EEPROM and jumps to kernel
3. the kernel sets up its own stack and preloads some text to certain domains

The overall load order is thus

$$\text{firmware} \rightarrow \text{MBR (OS-agnostic)} \rightarrow \text{VBR (OS-specific)} \rightarrow \text{kernel} \rightarrow \text{app}$$

We can visualize it like so:



With this knowledge, we can write our boot software:

Word Count App

```
// this is not called; it runs on boot
void main(void) {
// we read 80 sectors of 512 bytes — 40 KiB
for (int i = 0; i < 80, i++)
  read_ide_sector(i+100, 0x2000 + 512*i);

// jump to the first instruction
goto *0x2000;
}
```

We use the following input/output primitives:

```
#include <sys/io.h>
// CPU send signal to disk controller via bus
// disk controller sends back data from disk
// retrieve address 'a' (bus address) from disk
// this instruction is slow because signals travel on bus
unsigned char inb(unsigned short int port) { asm("...") };
// get data from port with address "port"
void outb(unsigned char value, unsigned short int port);
// write a byte of data "value" to port "port" (hardware specific
void insl(unsigned short int port, void *addr, unsigned long int count);
// read "count" bytes from "port" to "addr"
```

The layout of the drive to be loaded is as follows:

status/cmd 1f7	sector # 1f6 - 1f3	status/cmd 1f2	status/cmd 1f1 - 1f0

We will load according to the following protocol:

1. inb from 0x1F7 (status register) to check if controller is ready
2. outb to 0x1F2 (parameter 1 register) to give number of sectors to be read
3. outb to 0x1F3-0x1F6 (parameter 2 register) to give 32 bit sector offset...4 writes
4. outb to 0x1F7 (status register) a bit pattern telling controller we want to READ
5. inb from 0x1F7 (status register) to check if data is ready for copying
6. insl from 0x1F0 (device cache) 128 bytes of data

```
void read_ide_sector(int sector, int address) {
  // poll for readiness (1)
  while ((inb(0x1F7) & 0xC0) != 40) continue;

  // tell the controller we want 1 sector (2)
  outb(0x1F2, 1);

  // tell the controller where sector is (3)
  for (int i = 0; i < 4; i++) outb(0x1F3+i, sector>>(8*i) & 0xFF);

  // tell the controller we want to read (0x20=READ) (4)
  outb(0x1F7, 0x20);

  // wait for data to be ready for copying (5)
  while ((inb(0x1F7) & 0xC0) != 40) continue;

  // copy 128 bytes of data to addr from cache
  insl(0x1F0, address, 128);
}
```

Now that we can do the computation, we need a function to display the result.

- the screen pointer is represented by (base) + (row) + (column): [80]x[25]
- this utilizes memory mapped IO, no programmed IO, so it is not a bottleneck
- 16 bit quantity with low order as ASCII character and high order as appearance

Here is our code:

```
void display(long long nwords) {
  short *screen = (short*) 0xb8000 + 80*25/2 - 80/2);
  do {
    screen[0] = (nwords % 10) + '0';
    screen[1] = 7; // gray on black
    screen -= 2;
  } while ((nwords/=10) != 10);
}
```

Now that we have built our utilities, we need the outer layer function to tie it together:

```
// at addr 0x2000 jumped to by boot protocol
void main(void) {
  // 1TB > 2^31, so we use a 64 bit digit
  long long int nwords = 0;
  // bool for starting in the middle of a word on line
  int len, s = 50000;
  do {
    char buf[513];
    buf[512] = 0;
    len = strlen(buf);
    read_ide_sector(s++, (int) buf);
    nwords += cws(buf, len, &inword);
  } while (len == 512);
  display(nwords);
}
```

Now we only need the actual word count...

```
int cws(char *buf, int bufsize, bool *inword) {
  int w = 0;
  for (int i = 0; i < bufsize, i++) {
    bool alpha = isalpha((unsigned char)buf[i])'
    w += alpha & !*inword;
    *inword = isalpha(buf[i]);
  }
  return w;
}
```

...and we are done! our problem has a few issues, however...

1. Duplication of Code
   - BIOS must already do RAM reading, but we re-wrote it by hand
   - Code is not easily reusable
2. VERY special purpose—not generalizable
   - what if we wanted to boot with UEFI instead?
3. Inefficient
   - We spend a long time waiting. We could use yield() instead
   - insl() chews up the bus

- copy disk to CPU to RAM

4. Faults crash the entire CPU

We can fairly easily increase efficiency using *double buffering*.
:= we load the next output data while the first is being printed

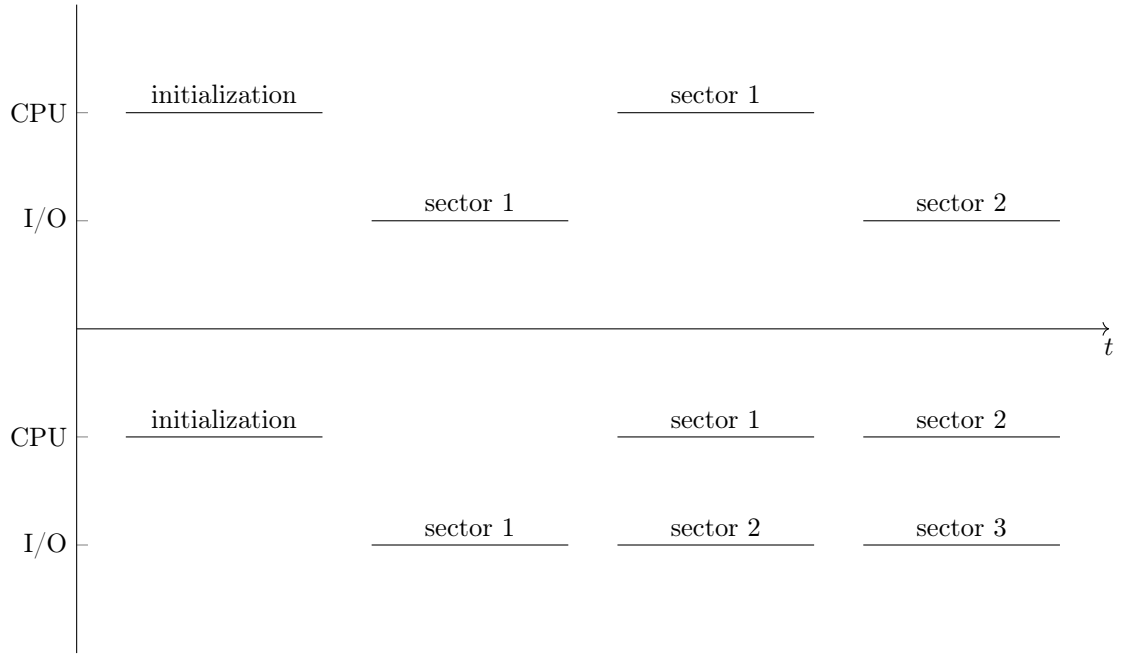


Figure 1: single (top) vs double (bottom) buffer speed

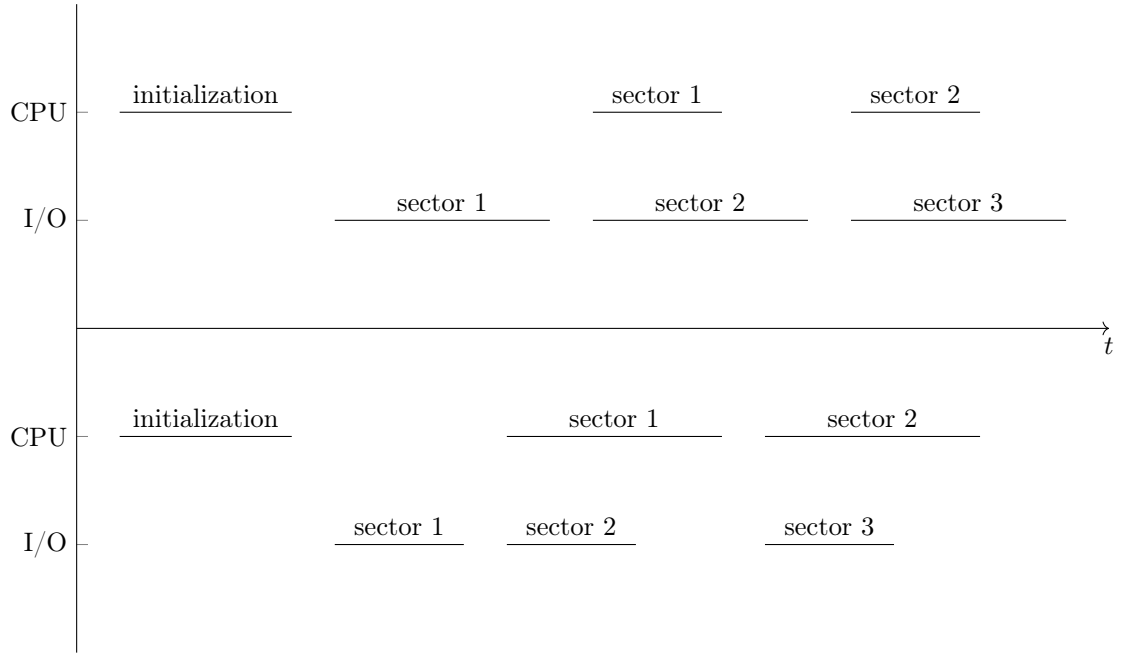


Figure 2: fast CPU (top) vs fast I/O (bottom) double buffer speed

We can thus observe that in certain situations, double buffering nearly doubles the program speed!

The one-piece solution clearly has many faults; we don't utilize some of our best tools:

1. Modularity

- a divide-and-conquer approach
- A system is divided into interacting subsystems called modules; attempts to subvert modular borders can cause effects to propagate
- these modules communicate through interfaces

2. Abstraction
   - the ability to treat others entirely based on external specs
   - is based on of the quality of the interface
3. Layering
   - a system which has layers of modules which can only interact with modules of a distance <= 1 layer from itself
4. Hierarchy
   - a system which has subsystems assembled upon self-contained subsystems
   - Contains the number of interactions between N elements to N*(N-1)
5. Iteration
   - starting with a simple system which fulfills some of the spec, then adding more.
   - Makes it easier to catch bugs and make adjustments

We can defeat the repeating of code using Modularity!

But what are the benefits of this? Well...Let's say the number of bugs in a program $\propto$ N and that the cost to fix a bug $\propto$ N , where N is the number of lines in a program. The time to debug is thus $O(N^2)$, but breaking it into K modules makes the time $O(N^2/K)$

NOTE: In reality, this is a simplification, since it assumes all bugs are 100% local, but the generalization still applies as we approach perfect modularity

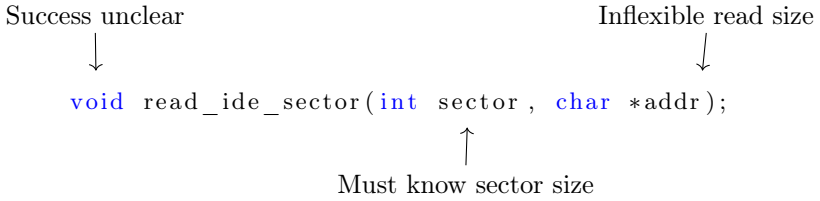
We need some metrics with which to judge the quality of modularity

3 Modularity

We need metrics for gauging the quality of API in terms of modularity:

- performance (interfaces and modularity hurt performance)
- robustness (how well are errors and faults dealt with)
- simplicity (easy to use/learn/maintain)
- neutrality/flexibility/portability (API must make few assumptions to run on all systems)
- evolvability (needs to be able to improve; design for iteration)

Let's evaluate our interface for reading in the bare-metal application:



Thus it is clear that this interface has many problems; Linux has the improved interface:

```
char *readline(int fd);
```

This is still a bad design for an operating system in many ways; we can evaluate it according to our criteria to see

- Modularity:
  - assumes the system does memory allocation
- Performance:
  - unbounded work — can be very inefficient on long lines
  - un-batched — large overhead on short lines
- Robustness:
  - apps can crash on long lines
- Neutrality:
  - forces line ending convention
- Simplicity
  - quite simple

Thus it is clear that we need a more flexible and powerful API which will allow us to directly access low-level hardware. Linux includes the following:

```
// lseek shifts the read and write point, white read loads the data
off_t lseek(int fd, off_t offset, int flag);
ssize_t read(int fd, char*addr, size_t bufsize);

// these have now been implemented as one primitive:
char *pread(int fd, char *buf, size_t bufsize);
```

Ultimately, we can see that the main difference between the two API's comes down to memory allocation — choice of API is incredibly important! Our procedure call API is not the only place where modularity is important, however; suppose we have the following source code:

```
int factorial(int n) { return (n == 0 ? 1 : n * factorial(n-1)); }
```

and we run the following commands:

```
$ gcc -S -O1 fact.c
$ cat fact.s
```

which results in the following output assembly code:

```
fact:
  movl $1, %eax
  testl %edi, %edi
  ne .L8
  ret

.L8:
  pushq %rbx
  movl %edi, %ebx
  leal -1(%edi), %edi
  call fact
  imult %ebx, %eax
  popq %rbx
  ret
```

What could go wrong? Consider a malignant actor runs the following assembly code:

```
badfact:
  movq $0x39c54e, (%rsp)
  ret
```

We could prevent this by implementing *Hard Modularity*

- no trust is necessary (nor often exists)
- one (or both) modules are protected
- avoids fate

as opposed to the currently used *soft modularity*

- the caller and callee are part of "one big happy family"
- a cheap but unsafe approach
- operates according to the caller/callee contract:

Contract	**Violation Consequence**
Callee only modifies its own variables and variables it shares with the caller; it does not modify the stack pointer and leaves the stack the same as it was called.	Callee corrupts the caller's stack and the caller may use incorrect values in later computations.
Callee returns to the caller.	Callee gets unexpected values of loses control of the program.
Return values are stored in %eax.	Callee will receive whichever value is in %eax, which may be incorrect.
Caller saves values in temporary registers to stack before calling callee.	Callee may change values that caller needs and caller will receive an incorrect result.
Callee will not have a disaster that affects caller (ex: early termination).	*Fate Sharing*: caller will also terminate.

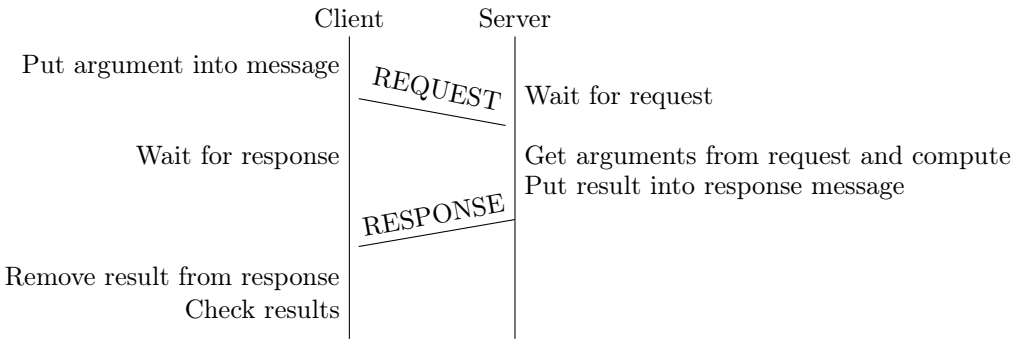
We can apply modularity to all 3 of the fundamental abstractions:

1. Interpreters/function calls (soft)
   - run untrusted module on a software "machine"
   - have 3 components:
     (a) instruction reference: location of next instruction
     (b) repertoire: set of actions and instructions
     (c) environmental reference: tells where to find environment
2. Processors:
   - a general purpose implementation of an interpreter
   - Instruction reference is a program counter
   - Repertoire includes ADD, SUB, CMP, and JMP; LOAD not READ
   - Have a stack and wired environmental reference
3. Java Virtual Machine (utilized by SEASNET)
   - Unprotected:
     (a) communicates via shared memory → SLOW
     (b) run out of stack space
     (c) functions can step on each other's memory
     (d) infinite loops
4. Virtualization (hard)
   - we simulate the interface of a physical object by:
     (a) creating many virtual objects by multiplexing one physical one
     (b) creating one virtual objects by aggregating multiple physical ones
     (c) implementing a virtual object from a physical one by emulation

- some hardware and assembly is involved
- limited to a preset service

5. Client/Server (hard)
   - Details:
     - run each module on its own machine
     - communicate via network messages
     - no reliance on shared state; global data is safe
   - Transaction is arms' length, so errors do not propagate
     - error propagation is called fate sharing
     - since there is none, all of our errors are controlled
   - Client can place limit on services
   - Encourages explicit, well defined interfaces

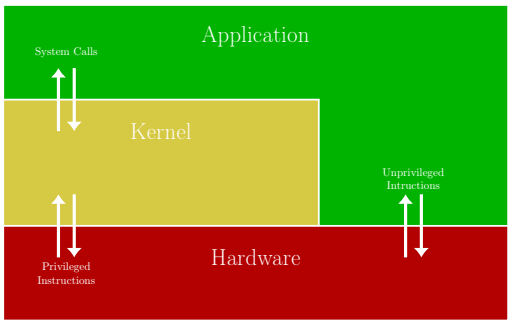
We can visualize the client/server interaction as follows:



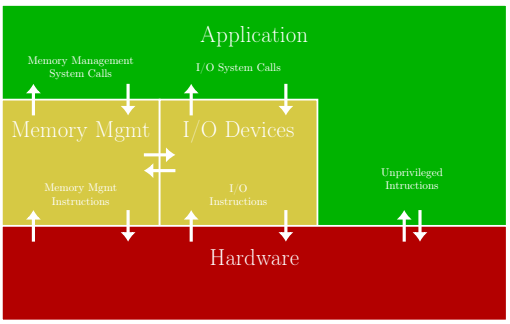
We focus on number 2, and number 3 is covered in CS 118.

4 Virtualization

The standard design for an operating system can be viewed as follows:



(a) Course granularity view



(b) Fine granularity view

We can see that our model has three general levels of instruction privilege:

- risky code with direct access to resources (OS only code) such as inb, outb
- code that accesses resources directly some of the time (iffy code) such as movl (movl is privileged if memory section to be accessed is in kernel)

- code with no access to code (unprivileged)

Untrusted modules can use code which does not have direct access; i.e. unprivileged code and (depending on context) iffy code.

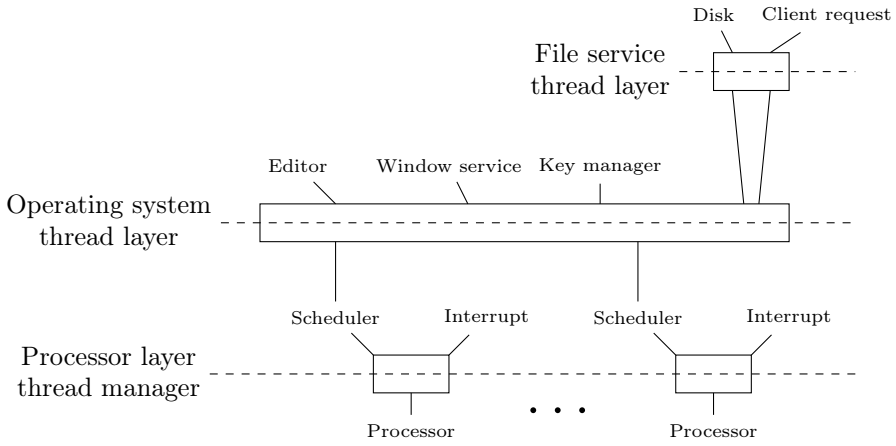
To allow for these restrictions on an untrusted module, the system boots in privileged mode, sets the environment for the untrusted code, and jumps to the untrusted code in unprivileged mode. However, if the untrusted application desires access to system resources, we send an *interrupt* with a *supervisor call* instruction.

If the error code matches the error code corresponding to a given operation, the privileged instruction is run. The *trap* signal jumps to the kernel and raises the permission. The *return from trap* signal jumps to the user and lowers the permission. Trap locations are defined by *trap handlers*. These are held in a *trap table* in the kernel stack. The key for this hash table is the *system call number*, commonly known as the *interrupt number*.

On the hardware side, when a trap occurs:

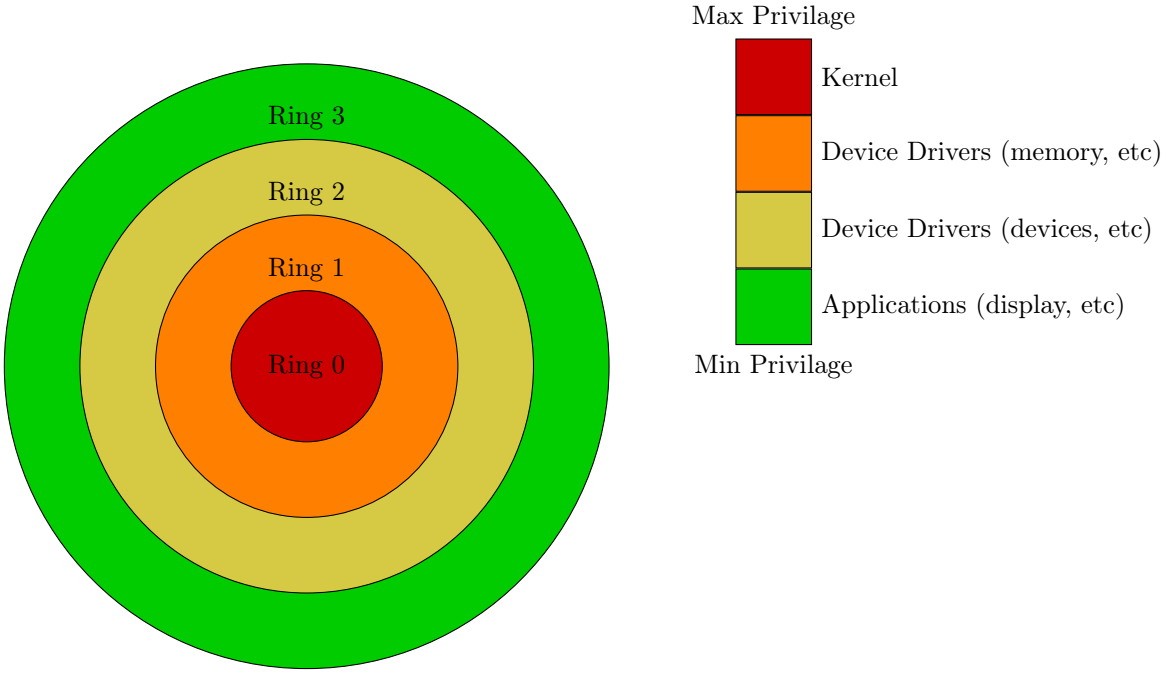
1. Running program information will be pushed onto the stack.
2. **eip** will be set to interrupt service routine.
3. Stack info will be popped from the stack when **reti** is called.

For interrupts, we can think of a processor as a hard-wired thread manager with two threads, the processor thread that runs the scheduler and the interrupt thread running in kernel mode. Threads with decreased privelage can be built on top of each other like so:



We define functions that give access to privileged instructions as *system calls*. This is as opposed to a *procedure call*, which is a function implemented on the stack that can call other functions. x86-64 Linux system calls use the convention of %rax for the interrupt number and %rdi, %rsi, %rdx, %r10, %r8, %r9, in that order, for arguments.

System calls are an example of the *protected transfer of control*, wherein the instruction pointer is set to a location known to be safe. On x86-64, there are 2 privilege bits corresponding to 4 levels of privilege that require different amounts of protection for control transfer. These can be visualized as follows:



But memory is only one thing to be protected; we must also control time access so as to avoid stalling!

5 Parallelism

The basic model of time control is the idea of virtualizing the processor. This is the idea of using the hardware to build more virtual versions of itself, and has the advantage of time sharing. We can use it to convince a program it is running with control of the entire computer, whilst also keeping certain procedures protected.

We do these by modeling a program running on a virtual processor as a *process*. These have their own ALU, registers, and (virtual) memory.

We prevent hackers from exploiting these by limiting access to system calls. Whilst it is slow, the time cost is not on the same order of I/O and is thus irrelevant.

How do we run a program?

1. load its data and code to memory/address space in executable format.
   This can be done *eagerly* or *lazily*
   - eagerly := load all data before the program runs
   - lazily := load data only as needed
2. allocate stack and heap
3. initialize stdin, stdout, stderr file descriptors
4. jump to main and transfer the control to the process to be run

C provides an api for this:

```
pid_t fork(void);
// creates a child process and returns a PID (0 in child, child PID in parent)
// the child runs the same code as the parent starting from after the fork
int execvp(const char *file, char *const argv[]);
// run a process
```

Process	Thread
fork()	pthread_create()
exec()	pthread_join()
waitpid()	pthread_join()

```
// argv is an array of strings that represent the argv for the call
// all data is copied except for the new data from the earlier section
```

The protocol for running a child program is:

- fork() — new virtual machine, same program
- exec() — same virtual machine, new program

This can lead to errors, so the parent is responsible for "baby proofing" the program. A bootstrap booter, on the other hand, calls exec() but not fork(), since we do not want to be returned to.

This can introduce a few error:

```
// we can have only the child exit with:
if (!fork()) ... exit(0);
// but what if the parent never calls wait? or worse:
if (fork()) ... exit(0);
// now the child is left as an orphan!
while(true) fork();
// a fork bomb takes up ALL available memory with a "dud" processor!
```

The data copied by fork and replaced by exec can be large and includes all program data except:

- PID and parent PID
- accumulated execution time
- file locks

Another abstraction we use is called a *thread*. Within a process, we may have multiple threads. A thread has its own program counter and registers. A thread switch does not require a change in address space; a thread only has its own stack/heap, referred to as thread-local storage, which includes unique copies of things like errno. Processes have an ID, SP, PC, and page-map-address (PMAR); a thread has virtual versions of all of these.

But why even use threads?

1. exploiting parallelism
2. overlap to avoid blocking inefficiencies (within the same program)
3. latency control
4. exploit the multiprocessor

Processes, on the other hand, would make it hard to share data!

How do we use threads?

1. load program text and data
2. allocate thread and run

Both of these are done by the *thread manager*.

We identify threads similarly to how we identify processes: Both types give a *handle*:

- Processes give an identifier (pid_t) in the form of an integer. This is *opaque*, and therefore restricts direct access to the processor that granted the ID.

- Threads give an identifier (pthread_t) in the form of a pointer. This is *transparent*, and therefore allows direct access and sharing of data structures.

Contrary to what its name would have you believe, kill() is not the actual process killer, since a parent still holds on to the process. exit() does not kill either, as an exiting child can be ignored and left as a zombie. Therefore it must be waitpid()!

We actually have a few additional ways to interrupt processes and threads:

```
pid_t waitpid(pid_t pid, int *status, int options);
// recombine the threads upon completion
// note a call to waitpid with options=-1 is equivelant to wait()
// Upon which the parent thread catches all the completed threads
void exit(int status);
// pass the return value
// uses jmp, so it never actually returns
void _noreturn _exit(int status);
// does not clear bufferx etc -> faster
// raise a signal on another thread
int kill(pid_t pid, int sig);
// if sig < 0, send to all decendents as well
// if pid = getpid(), we can use:
int raise(int sig);
// these only work if the sender and receiver share a user
unsigned int alarm(unsigned int seconds);
// a timer to guarantee the child does not run forever
// but the child could end the alarm with alarm(0)!
```

That being said, threads may still turn into orphans or zombies, so a process with index 1 takes in all orphaned threads by periodically calling

```
int waitpid(-1, status, UNOHAND);
// does not wait for children to complete, but does catch all completed children
```

We can actually simplify this process and reduce overhead by combining the fork() and exec() system calls, but it often ends up being more work than it is worth. That being said, C provides a procedure call for this:

```
int posix_spawn(pid_t *pid, const char *path,
                      const posix_spawn_file_actions_t *file_actions,
                      const posix_spawnattr_t *attrp,
                      char *const argv[], char *const envp[]);
// pid: pointer to process
// file: executable
// actions: fds for the child
// attrp: other attributes for the child
// envp: environment pointer
// "restrict" means that there exist no other pointers pointing to these objects
```

5.1 Inter-Process Communication

UNIX has a command called 'date' of the form:

Wed Jan 14 13:34:03 PST 2015

We attempt to emulate it in C with a status of the form:

```
bool printdate(void) {
  if ((pid_t p = fork()) < 0) return false;
  // have a child do the call!
  if (p == 0) {
```

```
    execvp("usr/bin/date", (char*[]){"date", "-u", 0});
    // this is only reachable on exec failure
    exit(127);
  }
  // if we get here, we are in the parent
  int status;
  if(waitpid(p, &status, 0) < 0) return false;
  // WIFEXITED tells us if the child exited (as opposed to signaled)
  // WEXITSTATUS gives us the exit value
  return (WIFEXITED(status) && WEXITSTATUS(status) == 0);
}
```

Unsurprisingly, there are other ways to transfer data between isolated processes. Processes (mostly) run in isolation for:

- ease of debugging
- security

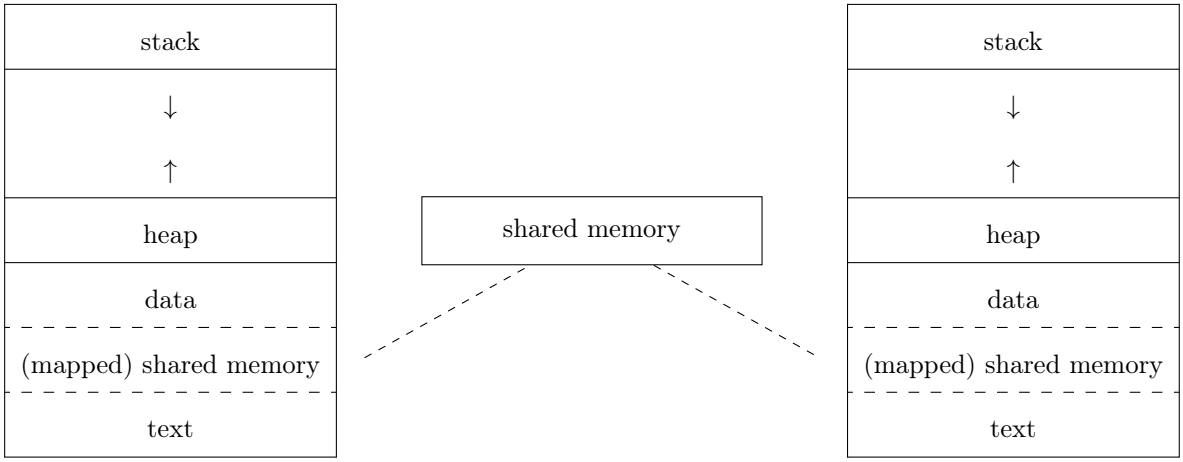
But there are a few notable exceptions:

- fork() conveys the child pid to the parent
- wait() conveys exit info to the parent
- kill() conveys info to everyone

Isolation is a good thing, but it makes it hard to communicate big data; how can we?

1. Storing in a file
   - First program must finish before the seconds can access it
   - I/O is SLOW
2. Message Passing
   - We send parts of a large structure in messages
   - This also involves slow copying
3. Shared Memory
   - give up isolation in pursuit of efficiency

If we require isolation, we must still use the first two methods.



6 Files

Input/output have a few important properties:

- I/O is very slow compared to CPU operations, so adding CPU overhead is fine
- robustness and security are the focuses
- API must work for many types of devices

We are temped to make a complex API to handle all this, but that would make usage difficult!

WE WANT: a simple, portable API that can work with the two major I/O device types:

Storage (flash, disk, etc.)	Stream(display, keyboard, etc.)
Request/Response	Spontaneous Generation
Random Access	Most Recent Data
Finite Size	Theoretically Infinite

So can we write an API that can handle both? Of course! Linux did it! How? By treating everything as a file!

But we have an issue: random access; so in addition to the standard OPEN/READ/WRITE/-CLOSE, we use

```
off_t lseek(int fd, off_t offset, int whence);
// where whence = SEEK_START || SEEK_END || SEEK_CURR}
// returns an error if used on a stream
```

The introduces the idea of *API orthogonality*. We want features to be "at right angles" to each other, ie the function should be independent of the function call. For example, read() has it, but not lseek()!

Here is a non-orthogonal API for reference:

```
int creat(const char *path, mode_t mode);
// using an open functionality requires a different function call to create a file!
```

Solution: introduce an O_CREAT flag to open()...and we dont need creat() any more!

Similar issues arise with concurrent lseek() and read()/write(), so we have two special commands

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);
ssize_t pwrite(int fd, const void *buf, size_t count, off_t offset);
```

BUT ORTHOGONALITY SOMETIMES FIGHTS CORRECTNESS

```
int unlink(const char *pathname);
int rename(const char *old, const char *new);
// these two function should be independent, but can we collide the axes?
```

```
// Process 1:
open("f", O_RDONLY);
// ...
read("f"...);
```

```
// Process 2:
// ...
unlink("f");
```

The read does not fail; Linux allows this by waiting to delete the file until no one has it open.

Here is an example of when a lack of modularity opens security flaws:

```
# our commands:
$ touch secret
$ ls -l secret
-rw-r--r-- 0 ... secret
$ chmod 600 secret
$ ls -l secret
-rw------- 0 ... secret
$ echo $password > secret
# this is secure, right?...of course not!
# bad process:
$(sleep 100; cat) < secret
# a bad process could open it before you restrict the permission and wait!
```

As a general rule, concurrent access to a file must be managed very carefully

```
$ (cat a & cat b) > file
# file becomes an aggregation of a and b
$ (cat > a & cat > b) < file
# file is split amongst a and b
```

The gap between file descriptors and files can create many such errors, including

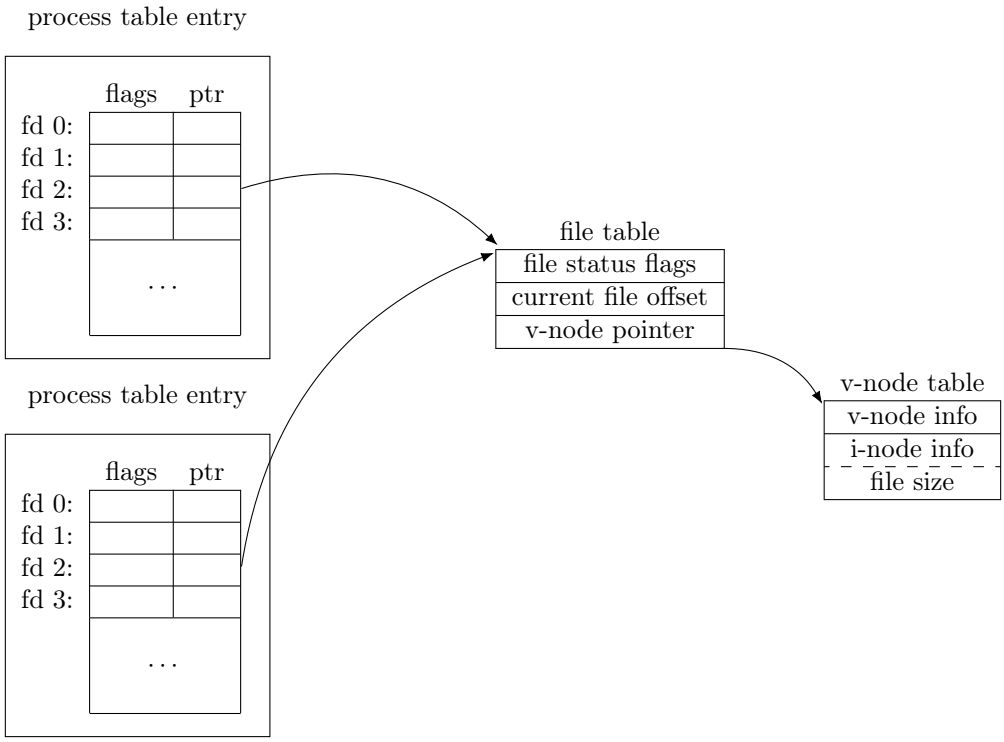
- races
- open but unnamed files (EMFILE = too many files)
- file descriptor leaks
- file descriptor not open (EBADF)
- I/O error (EIO — this one sucks to debug because it is so vague)
- EOF (return 0)
- device no longer there
- empty stream

In the case of the empty stream, we have two choices:

1. hang and wait for data
2. throw EAGAIN

But, unsurprisingly, people will be unhappy either way.

6.1 File Descriptors



- File descriptors are indices in a pointer table; vnode info includes "pipe/non-pipe" value
- On fork(), a copy of the parent's fd table is made
- On dup(), the child's pointer is removed, but the file remains for the parent
- Any number of the fd table entries may be present

How do these handle a complex case like $ cat file > output?

```
read(fd...);
write(1...);

// the shell must manipulate file descriptors to get the output! (like so:)
pid_t p = fork();
if (p==0) {
  int fd = open("output", O_WRONLY | O_CREAT | O_TRUNC, 0666);
  dup2(fd, 1);
  close(fd); // so we do not write to the wrong file
  execlp("/usr/bin/cat", (char* []){"cat", "file", 0});
}
```

But this type of error can happen on accident as well

```
int fd = open("/tmp/foo", O_RDWR | O_CREAT, 0600);
if (fd < 0)
if (unlink("/tmp/foo") != 0) abort();
// this thread does not let two concurrent threads run it!
```

Linux complicates the API to provide a solution to the earlier naming problem; it provides the flag O_EXCL that throws an error if the file already exists. But we don't want failure... we don't care what the name is!

SOLUTION:

```
do {
  char buf[100];
  sprintf(buf, "tmp/foo%ld", random());
  fd = open(buf, O_RDWR | O_CREAT, 0600);
} while (fd < 0 && errno == EEXIST);
```

```
// but the temp directory could fill, and we would lose our (unlinked) file!
```

Here are a few more weird cases:

```
$ cat f > f
# f is left empty!
$ cat < f > f
# cat is emptied
$ cat < f >> f
# infinite loop! (so long as f wasn't empty)
```

To handle these cases, we need to introduce another structure called a "Pipe"

7 Pipes

Pipes have two main benefits:

- they control same inter-process communication
- they are transparent (the program doesn't know they exist

How are they implemented?

```
int pipe(int pipefd[2]);
pipefd[0] = "read"
pipefd[1] = "write"
// these can be thought of like 0 = stdin and 1 = stdout
```

This builds a circular bounded buffer that can be used to communicate between processes BUT that introduces the case of a full buffer; what do we do?

- write what you can and return the size
- wait
- throw error

Pipes therefore have a third benefit—flow control!

```
$ du | sort -n
```

How do we use pipes?

```
int pipefd[2];
pid_t cpid;
char buf;
if (argc != 2) exit(EXIT_FAILURE);
if (pipe(pipefd) == -1) {perror("pipe"); exit(1);}
if ((cpuid = fork()) == 0) {
  close(pipefd[1];
  while(read(pipefd[0], &buf, 1) > 0) write(STDOUT_FILENO, &buf, 1);
  write(STDOUT_FILENO, "\n", 1);
  close(pipefd[0];
  _exit(0);
}
else { // parent
  close(pipefd[0]);
  write(pipefd[1], argv[1], strlen(argv[1]));
  close(pipefd[1]);
  wait(NULL);
```

```
    exit(0);
}
// note: the order of execution is not guaranteed
```

Pipes, as mentioned above, are not perfect; we have 4 major categories of pipe failure:

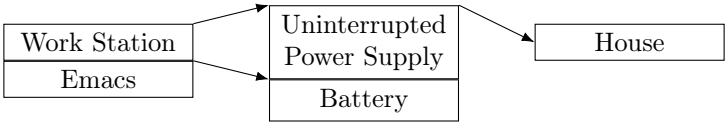
1. Reading from an empty pipe where the writer never writes $\implies$ hang indefinitely
2. Reading from an empty pipe with no writers $\implies$ return 0
3. Writing to a full pipe where the reader never reads $\implies$ suspend/hang indefinitely
4. Writing to a full pipe where the reader never reads
   - SIGPIPE since returning 0 would probably cause many people to retry
   - if SIGPIPE is ignored, write fails with ESPIPE

Note that the big mistakes and performance errors occur when we do not close pipes. For example, a named pipe can cause an fd leak, since it could be referenced by any program

A common issue:

```
while (condition) printf("%d\n", number);
// since this would run forever on write failure,
// we have an asynchronous signal SIGPIPE
// which kills a program if no readers remain
```

BUT sometimes there is a reader, who's not listening (plus closing has a delay). Take the setup:



What do we do if the power in the house goes out?

1. do nothing and lose work
2. OS saves a stack of running processes and state in the stack
3. End-to-End (processes are notified and can figure it out for themselves)

Since #2 can make clock-based apps go haywire, we consider #3; we have 3 ways of notifying processes that we have lost power

1. poll /dev/power
   this is annoying, since the burden is on the programmer
2. /dev/powering_off
   the kernel blocks this unless we have lost power (too complicated)
3. SIGNALS
   grab the attention of a program not designed to receive it

8 Signals

Why do we use signals?

- asynchronous I/O with aread()

  - returns right away and kernel keeps going with handling the data
  - get a SIGIO signal later
- error in your code (SIGFPE)
  - divide by zero
  - floating point overflow
  - invalid instruction
- impatient user of infinite loop (SIGINT)
  - ^C to end program
- impending power outage (SIGPWR)
  - So we can do any saving before shutdown
- to check for dying children (SIGCHLD)
  - p = waitpid(-1, &status, WNOHANG)
  - now we don't have to call this method every 100 milliseconds
- user went away (SIGHUP)
- alarms
  - alarms are not inherited by fork() but by execvp()
- suspending a process
  - $ kill -STOP 29; kill -CONT 29
- kill the program SIGKILL
  - cannot be caught or ignored

We can kill a process with:

```
while (fork()) continue;
$ kill -KILL 29316        # does not kill children
# however this does not kill the shell bomb
$ kill -KILL -29316       # kills all children as well
```

Code must often be developed specifically to be able to handle signals; take the code

```
fd = open("foo", O_RDONLY);
fo = open("foo.gz", O_WRONLY);
while (compress(fd, fo)) continue;
close(fo);
unlink("foo");
// THIS CODE IS NOT ATOMIC AND CAN BE INTERRUPTED BY A SIGNAL
```

We can attempt to avoid these errors by implementing a signal handler

```
static void cleanup (int sig) {
  unlink("foo.gz");
  _exit(1);
}
int main(){...
  fd = open("foo", O_RDONLY);
  signal(SIGINT, cleanup);*
  fo = open("foo.gz", O_WRONLY);
  while (compress(fd, fo)) continue;
  close(fo);
  signal(SIGINT, SIG_DFL);*
  unlink("foo");
...}
```

but this still leaves race conditions .

In our current implementation, all threads are affected by the sign. Should all threads handle the signal? Would all threads handle it the same? NO; threads have their own signal mask to ignore signals. This is why pthread_sigmask() affects only current thread!

So how do we handle them? By default threads have their signals blocked. We use pthread_sigmask() to unblock the signal if we want a thread to handle it. Linux picks one random thread to deliver the signal. We make a mask with

```
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oldset);
// how = SIG_BLOCK, SIG_SCIMAKS, SIG_UNBLOCK
```

this allows the signal to arrive even before function returns. With this, we can build critical sections such as

```
sigset_t ss;
sigemptyset(&ss);
sigaddset(&ss, SIGINT);
pthread_sigmask(SIG_BLOCK, &ss, 0);
// critical section here .......
pthread_sigmask(SIG_UNBLOCK, &ss, 0);
```

But how can a signal handler manage memory access?

```
void handle_interrupt(int sig) {
  fprintf(stderr, "Interrupted\n");
  unlink(...);
}
fprintf(...) { malloc(...); } // interrupt
malloc(...) { // operating on heap }
// if we interrupt malloc and fprintf will call malloc again
// the second malloc may corrupt the heap, thus the first malloc call
```

Only some system calls can safely be used in handlers! We can call most system calls, such as:

- _exit()
- write()

But there are exceptions:

- exit() (calls malloc, flushes I/O buffer)
- fprintf()
- malloc()

We can perform all system calls in a single handler with:

```
void handle_interrupt(int sig) {
if (pthread_self() == stgmgr) really_handle_interrupt();
else pthread_kill(SIGINT, stgmgr); // forward signal to stage manager
}

# a more conservative approach is to set the variable and handle outside
sig_atomic_t volatile globv;
void handle_interrupt(int sig) {
  global = 1;
}
// always memory access, no cache!
```

But even with our scrupulous effort, interrupts can still cause difficulty:

```
read("/dev/tty", buf, 100);
// SIGHUP signal arrives
// run SIGHUP handler
// returns and continue reading
```

This means we have to complicate our code:

```
while (read("/dev/tty", buf, 100) == -1 && errno == EINTR) continue;
```

These types of errors are common with long system calls; clearly scheduling concurrent threads properly is important.

9 Scheduling

Schedulers can work on multiple levels:

1. long-term scheduler
   which processes should be admitted to OS?
   - look at return value of fork()
   - if fork() < 0, denied by OS
2. medium-term scheduler
   which processes reside in RAM?
3. short-term scheduler
   which threads get to run on the limited number of CPUs

A process can be in one of three states:

1. *Running* := executing instructions on a processor
2. *Ready* := ready to run but waiting on OS
3. *Blocked* := waiting on another event

In addition to these three, there are the edge cases of:

1. *Initial* := just created, environment not set up
2. *Final/Zombie* := process is complete but hasn't been cleaned up yet

Moving from ready to running is called being *scheduled.*Moving from running to ready is called being *de-scheduled.* The act of scheduling is assigning CPUs to threads. Each instruction pointer requires a CPU to run. This requires a harmony of hardware and software

Determining how to schedule is easy when we have enough CPUs, but what do we do if there are more threads than CPUs?

We need a few things to answer this

- Theory: scheduling policy
- Practice: Scheduling and Dispatch Mechanisms

Tiny gaps where neither thread is executing are called *context switches*, and are done by the OS. These can be timed in two broad ways:

1. Cooperative Scheduling :=a thread "volunteers" to give up its CPU with syscalls
2. Preemptive Scheduling := the OS preempts threads every time slice
    - this is cheap and easier on the OS
    - this is common in IOT/embedded systems
    - short slice = less efficient
    - long slice = long wait

How do threads "volunteer"?

```
#include <sched.h>
int sched_yield(void);
```

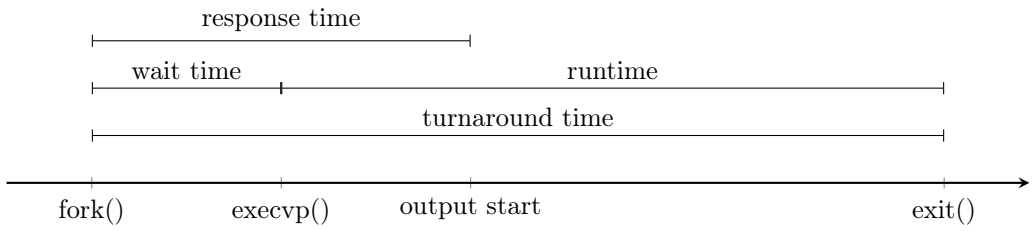
We can use this to make waiting on a device more efficient:

```
// we can bust wait
while(isbusy(device)) continue;
// we can poll, which is still ineffecient (but better)
while(isbusy(device)) sched_yield()
// or we can block, telling kernel to wake it when the condition is met
while(isbusy(device)) wait_until_ready()
```

The Linux scheduler runs the following algorithm:

```
for (;;) {
  choose an unblocked thread
  load it into a CPU
  run it
  yield
  store state
  for each thread that has become ready
    unblock
}
```

This scheduler is completely terrible! People have measured! But what did they measure?



In addition, SEASNET screws us students over with a priority queue with priorities:

1. root
2. operations staff
3. students/faculty

And each of these contains its own sub-scheduler and algorithm

Usually, priority 1 is the highest priority, but SEASNET uses the ides of *niceness*. If p1 has niceness x and p2 has niceness y > x, p2 will defer. users can raise a program's niceness, but cannot lower it.

We can't be too harsh though... there is a lot of complexity involved in *real-time systems*.

9.1 Real-Time Scheduling

This contains two types of deadlines:

- HARD:
  - deadlines CANNOT be missed, $\implies$ performance = correctness
  - predictability > performance, $\implies$ caches are the enemy
  - use polling instead of interrupts, since polling controls test duration
- SOFT:
  - a missed deadline is not necessarily a failure
  - 2 scheduling options:
    1. rate-monotonic scheduling $\coloneqq$ give a % usage to job data streams
    2. earliest deadline first $\coloneqq$ can drop late requests when inundated. This allows one stream to monopolize the CPU.
       ex) Video Playback: Each frame is treated as its own request. When the connection is slow, the scheduler periodically drops frames; this is often imperceptible to humans

Most real-life systems have both hard and soft real-time scheduling; this introduces a lot of complexity.

10 Scheduling Policies

10. Scheduling Policies

We use the following processes to compare scheduling policies:

Jobs	Arrival Time	Work
A	0	3
B	1	5
C	3	2

We denote the cost of a context switch as "δ".

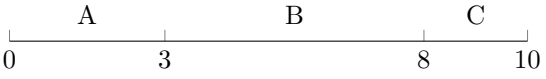
FIRST-COME, FIRST-SERVED (FLFS)

The process:

1. hold a queue of threads waiting to run
2. run threads to completion

This is good for batch application with pre-known tasks (payroll, auditing, etc.)

Jobs	Wait Time	Turnaround Time
A	0	5
B	$4+\delta$	$6+\delta$
C	$5+2\delta$	$14+2\delta$
D	$13+3\delta$	$17+3\delta$
AVG	$5.5+1.5\delta$	$10.5+1.5\delta$



Properties:
+ fair
+ utilization
- long wait time
- convoy effect

Total time of execution = $20 + 3\delta$

* the cost of a context switch is cheap here, since no save state is necessary

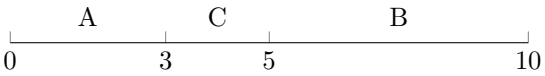
SHORTEST JOB FIRST (SJF)

The process:

1. hold a min-heap of waiting threads
2. run threads to completion

This algorithm makes the assumption that runtimes are (roughly) known.

Jobs	Wait Time	Turnaround Time
A	0	5
B	$4 + \delta$	$6 + \delta$
C	$9 + 3\delta$	$18 + 3\delta$
D	$4 + 2\delta$	$8 + 3\delta$
AVG	$4.25 + 1.5\delta$	$9.25 + 1.5\delta$



Properties:
- \+ improved avg weight and turnaround time
- \+ high utilization
- \- unfair: starvation

Total time of execution = $20 + 3\delta$

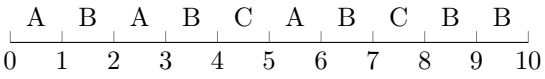
* the cost of a context switch is cheap here, since no save state is necessary

ROUND-ROBIN (RR)

The process: FCFS, but each process is only run for a set time slice

This algorithm necessitates preemption.

Jobs	Wait Time	Turnaround Time
A	0	$15 + 14\delta$
B	δ	$5 + 5\delta$
C	2δ	$18 + 15\delta$
D	3δ	$11 + 13\delta$
AVG	$4.25 + 1.5\delta$	$12.25 + 11.25\delta$



Properties
- \+ fair, if new jobs are placed at the end of the queue (pictured algorithm does not do that)
- \+ shorter wait time
- \- lower utilization

Total time of execution = $20 + 15\delta$

* the total time to run all of the tasks has gone up

The above scheduling policies have been static, some scheduling policies are dynamic, and depend on the state of the environment of the machine.

PRIORITY SCHEDULING

The process: Jobs are given priority, and the highest runs first

Linux uses an inverted version with "niceness", where nice tasks will defer to others. Students can increase niceness, but not lower it

```
nice.c = {
  read args
  set priority
  execvp("gcc", (char* []){"gcc", "foo.c");
}
```

Specifically, this is an example of dynamically assigned priority

There is an even more complicated version of this called a:

MULTILEVEL FEEDBACK QUEUE (MLFQ)

Goals:

1. optimize turnaround time
2. responsive to interactive users
3. optimize response time

Maintain many queues with distinct priority levels.

The process: Round Robin within a queue according to the rules:

(i) if P(A) > P(B), run A
(ii) If P(A) = P(B), Round Robin
(iii) new jobs enter at top queue
(iv) jobs move down a queue after using time allotment
(v) boost all jobs to the top queue after a set time interval

Parametrizing the time interval is tough; it is a voodoo constant. Some find this value using *decay-usage algorithms*. Others let users manipulate it with hints, called advice.

11 Concurrency

We like loading into RAM; it is fast BUT this is the recipe for races.

Say I want to keep track of my money:

```
bool deposit(int amt) {
  if (amt < 0 || INT_MAX - amt > balance) return false;
  balance += amt;
  return true;
}
bool withdraw(int amt) {
  if (amt < 0 || balance - amt > 0) return false;
  balance += amt;
  return true;
}
```

This is a TERRIBLE implementation

- if two threads try to deposit at the same time, one may be ignored
- an interrupt during the += can cause an ignore

This is a mistake in synchronization, which can occur in:

- a single CPU with preemptive scheduling
- a single thread with interrupt handling
- a multiple CPU system

We call these errors caused by concurrent access *race conditions* := pieces of code which can cause race conditions are called *critical sections*. If we can make critical sections atomic at the point of observability, we can say that the actions are serializable, or the same result could be found by running the operations sequentially.

We have atomicity iff we have:

(i) Mutual Exclusion ≔ one thread in the critical section excludes all others

(ii) Bounded Wait ≔ eventually, waiting threads will be able to enter the critical section

There are a few major complications that we must guarantee atomicity around:

(i) Preemptive Scheduling
we may be interrupted mid critical section and leave in an invalid state

(ii) Threads
a thread might be kicked out while others are running and leave the others to utilize an invalid state

The *Goldilocks principle* for critical sections: If everyone only reads, then everyone is safe; only writes cause problems. Therefore when searching for critical sections, we repeatedly

1. look for shared writes (each of the shared writes should be in a critical sectIon)
2. expand to include dependent reads and intervening computation

```
// new_balance dependent on balance
long long new_balance = balance + amt;
balance = new_balance;
```

Guaranteeing these rules is hard... we can avoid it with a few tricks:

1. Single-Threaded Code
2. Event-Driven Programming
   - useful for when we have 1 CPU and many threads
   - of the form:
     for(;;) {
     wait for an event;
     act on that event;
     }
   - acting on E must be fast enough as to avoid any waiting
   - common in IOT appliances
   - Downsides:
     - code is restricted
     - no true parallelism
     - too easy to be interesting
3. Synchronization via Load & Store
   the size of objects is restricted:
   - no large objects! (copying takes multiple cycles and is not atomic)
   - no small objects! (writing less than a byte still copies the whole byte first)

   Therefore we can use only objects of 1 byte, since that is the granularity of x86-64 copy.

We can also have synchronization errors without any critical sections whatsoever:

```
// thread 1
for (long i = 0; i < n; i++) continue
// thread 2
...
n = 0;
// i is only copied once! s trying to force thread 1 to stop waiting fails
// we must use the keyword 'volatile' to require copying from memory each time
```

To guarantee atomicity, we must introduce the concept of locks!

12 Locks

Let's implement our own version of a pipe

```
#define PIPE_SIZE 1<<12
struct pipe {
  char buf[PIPE_SIZE];
  unsigned r, w;
  lock_t *alock;
}
void lock(lock_t* alock);
void unlock(lock_t* alock);
// we ignore the implementation of lock and unlock for now
char readc(struct pipe *p) {
  lock(&l);
  // we cannot hold the lock while the pipe is full; we wait
  while (p->r == p->w) {
    unlock(&l);
    lock(&l);
  }
  char c = p->buf[p->r++ % PIPE_SIZE];
  unlock(&l);
  return c
}
// we do a writec the same way
```

This is a fine grain lock; we could use a global lock, but that would be course grain and slow

How do we implement a lock?

We can use the following instruction

```
int xchgl(int* p, int val) {
  int old = *p;
  *p = val;
  return old;
}
```

and thus write the following cosine function:

```
void cosine(struct s *p) {
  do {
    double d1 = p->d;
    double d2 = cos(d1);
  } while(xchgl(&p->d, d1, d2) != old);
}
```

but the value can change before we run xchgl and cause us to loop infinitely!

We need to use cas!

```
bool cas(int *p, double old, double new) {
  if (*p==old) {*p=new; return 1;}
  else return 0;
}
```

We can instead use the command to build a portable

```
void cosine(struct s *p) {
  do {
    double d1 = p->d;
    double d2 = cos(d1);
  } while(!cas(&p->d, d1, d2);
}
```

We can utilize the above commands to abstract a lock API of the following sort:

```
struct s {
  double d;
  lock_t l;
}
void cosine(struct s *p) {
  lock(&s->l);
  p->d = cos(p->d);
  unlock(&p->l);
}
```

Maximum performance is found by minimizing the critical section, but we have reached the atomicity of the next layer down, so this is as well as we can do!

This style of lock is still pretty slow; a mutex is more efficient than a spin lock because we sleep while waiting

Let's try to make one!

```
//Mutex
typedef struct s{
  bool acquired;
  thread_descriptor_t *blocked;
  lock_t lock;
} bmutex_t;
// blocked forms linked list of waiting threads

void acquire(bmutex_t *b) {
  again:
  lock(&b->lock);
  if(!b->acquired) {
    b->acquired = true;
    unlock(&b->lock);
  }
  else {
    self->blocked->blocked = true;
    add_self_to_blocked_queue();
    unlock(&b->lock);
    yield();
    goto again;
  }
}
void release(bmutex *b) {
  b->locked = 0;
  unlock(&b->lock);
}
```

We can generalize a mutex to for a number of concurrent threads with a semaphore.

Semaphore

The basics:

- locked when ctr <= 0
- ctr = numbers of resources left
- N threads waiting if ctr = N
- to lock, increase ctr and place self on queue

Eggert is not a huge fan of semaphores, so they were not discussed much. They are effectively a primitive for locks and condition variables. We can, however, use these to prevent thrashing!

We can also use this to solve an earlier problem; if the pipe is empty, neither reading nor writing can be done, since it appears full. This is called the *producer/consumer problem.*

SO we abstract from a semaphore to solve it with a condition variable

Condition Variable

A condition variable contains 2 parts:

1. bool isempty
2. blocking mutex (binary semaphore)

The API is as follows:

```
int pthread_cond_wait(condvar_t *c, bmutex_t *b);
// wait until the condition becomes true
int pthread_cond_signal(condvar_t *c);
// notify the first waiting thread that c is true
int pthread_cond_broadcast(condvar_t *c);
// notify all waiting threads that c is true
// Used for when there are too many variations for separate conditions
```

A wake does not guarantee a condition is met; the thread must check again.

We implement one like so

```
struct pipe {
  char buf[BUFSIZ];
  bmutex_t b;
  int r, w;
  condvar_t nonempty;
}
int pipe_read(struct pipe *p) {
  again:
  acquire(&p->b);
  if (p->w == p->r) {
    pthread_cond_signal(&p->nonempty, &p->b);
    goto again;
  }
  char c = p->buf[p->r++%BUF_SIZE];
  release(&p->b);
  notify(&p->nonempty);
  return c;
}
```

We can use these ideas to create a few thread-safe data structures:

1. Concurrent Counters:
   - Place a lock around access and incrementation
   - Not perfectly scalable, since multiprocessing greatly increases time cost
2. Scalable Counter:
   - Each thread gets its own counter.
     - Threads share one global counter, local counters are flushed after set time
     - Value is ± n_threads*flushTime
     - Time to flush locks: high = inaccurate, low = not scalable
3. Concurrent Linked Lists:
   - Locking Options:

(a) Hand-over-hand/lock coupling
  - Each node has its own lock
  - This is slow, so we usually do not use it

(b) List Locking
  - One lock for head and one for tail
  - Wait if queue is full or empty

4. Concurrent Hash Table:
   - One lock per bucket
   - Still constant time!

NOTE: With these, we still want to avoid premature optimization

There is still an optimization we can perform for small critical sections for a speedup; we cheat our way to improving locks with machine code

```
lock:
  movl $1, %eax

unlock:
  xrelease movl $0, mutex

try:
  xacquire lock xchgl %eax, mutex
  cmp $0, %eax
  jne try
  ret
# store edits into a cache & write into memory only on success
```

A few observations:

- This relies on caching for speed, so critical sections must be sufficiently small
- Note that if we did not use test and sleep, we could get a race condition
- This makes a single-threaded program slower due to overhead

BUT WAIT, we can still get bugs based on usage

```
bool copyc(struct pipe *p, struct pipe *q){
  bool ok = 0;
  acquire(&p->b);
  acquire(&q->b);
  if(p->w - p->r == 0 && q->w - q->r != 1024){
    q->buff[q->w++%1024] = p->buf[p->r++%1024];
    ok = 1;
  }
  release(&p->b);
  release(&q->b);
  return ok;
}
// multiple threads could end up waiting on each other, and the program halts
```

This situation is called a

12.1 Deadlock

There are four condition for deadlock:

1. *circular wait* := there is a cycle of dependencies

2. *mutual exclusion* ≔ threads claim exclusive control of required resources
3. *no preemption of locks* ≔ resources cannot be forcefully taken from threads
4. *hold & wait* ≔ threads hold resources while waiting for more

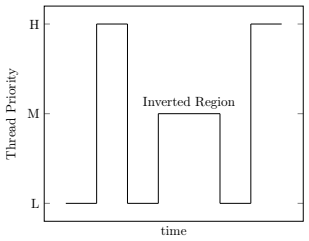
We have a few ways to avoid deadlocks (besides just using lock-free and wait-free data structures)

1. Dynamic Deadlock Detection
   - The OS checks for possible deadlock in any thread on request & EDEADLOCK on fail
   - This pushes deadlock handling onto the app; if there is a deadlock, the OS just gives up
2. Lock Ordering
   - assign a priority to each lock (often by address)
   - always lock objects in order of priority
   - wait on first lock failure but release all and try again if a later one fails
3. Intervention
   - kill deadlocked threads
   - is used much too often

There are some types of deadlock that don't fit neatly into any solution

1. Parent/Child Communication
   If the two threads communicate via two pipes and both write a lot and don't read, both may wait on a full pipe!
2. Priority Inversion
   - High priority thread needs a lock
   - Low priority thread has that lock
   - Low priority thread is waiting on mid priority

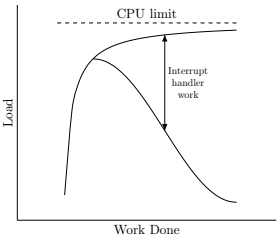
   Then the high priority is waiting on the mid and low!

   SOLUTION:
   threads gain the priority of the highest priority dependent thread

We can also be stopped by *live-lock* ≔ when the thread is trying to run but failing.

For example, an inundated router may spend most of its time accepting requests, since they come via interrupt, and never actually fulfill any requests. We have two possible solutions:
1. if router fills, toss requests until the router hits a low threshold
2. if requests are above a threshold, block interrupts



We discuss two of the most common non-deadlock bugs:

1. *atomicity violation* ≔ code has an implied atomicity which is violated. This involves an *atomicity assumption* of a non-atomic action. This can be solved by locks!

```
// Thread 1::
if (thd->proc_info) {
  fputs(thd->proc_info, ...);
}

// Thread 2::
thd->proc_info = NULL;
```

2. *order violation* := the desired order of two memory accesses is flipped. This usually involves trying to act on a variable that is inialized in another thread. This can be solved by semaphores.

```
// Thread 1::
void init() {
  nThread = PR_CreateThread(nMain, ...);
}

// Thread 2::
void nMain(...) {
  nState = nThread->State;
}
```

13 File Systems

Take the ORNL Cray Cluster Star

- Storage: 1 EB (10^18 bytes $\rightarrow$ 1 million TB)
- Speed: 10 TB/s output
- Size: 40 cabinets with 19 inch racks
- Cost: $50 million
- Utilizes flash and disk
- 2 systems:
  - Lustre for distributed
  - ZFS for local

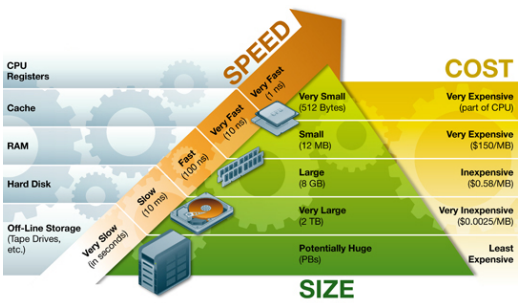
	Flash	Disk
Cost	$110/TB	$14/TB
Reliability		✓
Durability		✓
Speed	✓	

How do we judge performance?

(a) throughput := total requests or bytes per second (read is advertised, since it's fast)
(b) latency := delay between request and response
(c) utilization := fraction of capacity doing useful work

But (a) and (b) are competing values!

What strategies do we have?



We exploit locality of reference by caching data we expect to need into higher memory:

- spacial locality := accessing address i means accessing address $i \pm 1$ is likely
- temporal locality := accessing address i means accessing i again is likely

The file system takes advantage of this locality in 3 main ways:

- Prefetching := the OS caches memory it expects to need ahead of time
- Batching := the OS reads all around a piece of data on read (includes prefetching)
- Dallying := the OS caches writes with the hopes that it can perform adjacent writes

But what if we need the data to be written NOW?

```
sync();
// flush all buffers to permanent storage
// performed system wide, so it is often a waste of effort
fsync(int fd);
// flush all buffers for a given file descriptor and wait for completion
fdatasync(int fd);
// flush all data buffers for a given file descriptor and wait for completion
// fast, since metadata is the slow part (it is stored in memory)
```

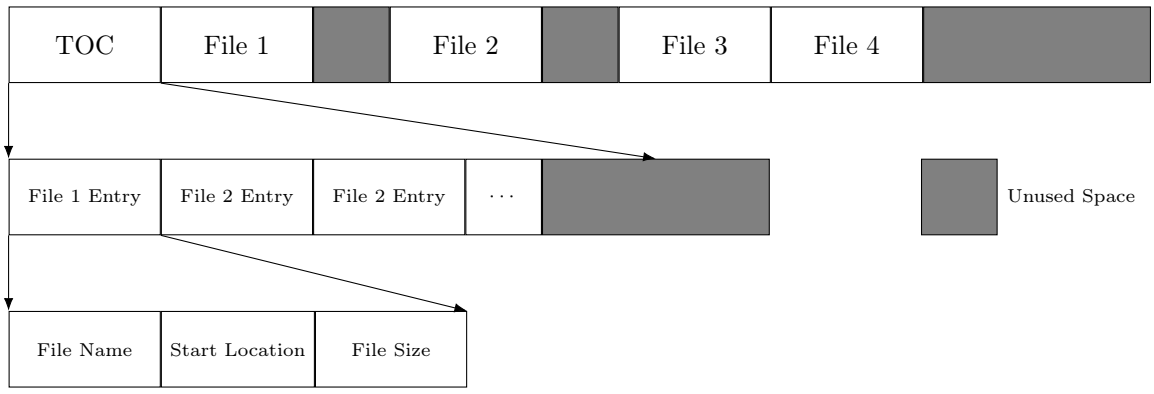
The IBM General Parallel File System uses a few other tricks for optimization:

- Striping
- Distributed Metadata := Copies of file metadata exist to alleviate IO bottleneck
- Distributed Locking := Systems allow users to lock sections of file, and not the whole file
- Efficient Directory Indexing := GPFS uses a fancier data structure to represent files.
- File System Stays Live During Maintenance

13.1 File Systems

:= a data structure for primary and secondary storage with support for searching.

13.1.1 Very Simple File System (RT-11)



RULES:

- 2mb of memory broken into 512 byte sectors
- files start on sector boundaries
- files are continuous (like in RT-11)
- sectors will only be partially used
- first 10 sectors form a table of contents

- file size is statically decided on creation
- first byte indicates if the directory is full
- files can only span a single region of free space

PROS:

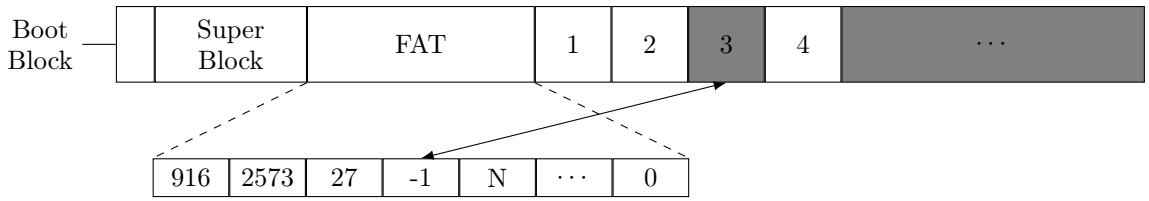
- predictable
- easy truncation
- good sequential access

CONS

- file number limit (2^6)
- difficult to grow files
- only one directory — no user ones
- no file permissions fragmentation
  - internal := wasted data within allocated space
  - external := scattered and therefore unusable free space

13.1.2 FAT File System

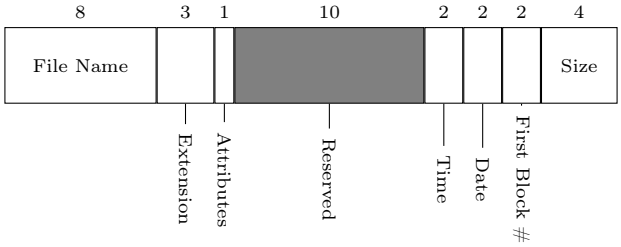
The VSFS was the primary file system until Bill Gates came along in the 70s and created...



The goal: turn files into linked lists of blocks to avoid external fragmentation.

RULES:

1. Blocks contain discrete size blocks of file data
2. Boot block contains setup code
3. A *super block* is fixed size and contains the file system's
   - Size
   - Version number
   - Number of blocks in use
   - Root directory block number
4. FAT contains address of the block which comes next in the file
   - -1 $\iff$ Free block in file system
   - 0 $\iff$ EOF (last block in current file)
   - N $\iff$ Next block in this file is N
5. Directories form a tree rooted at the root directory
6. Directories contain data blocks just like files, but the form is different, as follows:



PROS:

- no external fragmentation
- no preallocation
- large file count limit

CONS:

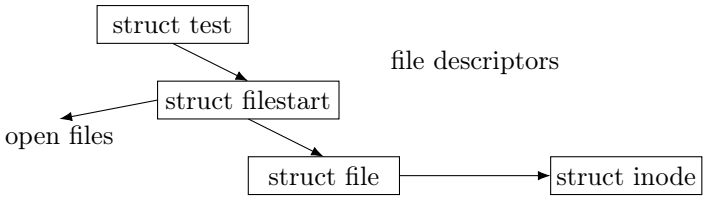
- internal fragmentation is not fixed
- can be addressed by periodic defragmentation of files; this is both costly and risky
- moving a file to another directory can lose both copies, so this is forbidden
- no random access — we would need to walk through the whole list to find a byte offset

Random access was important to Unix, so Linux made this easy

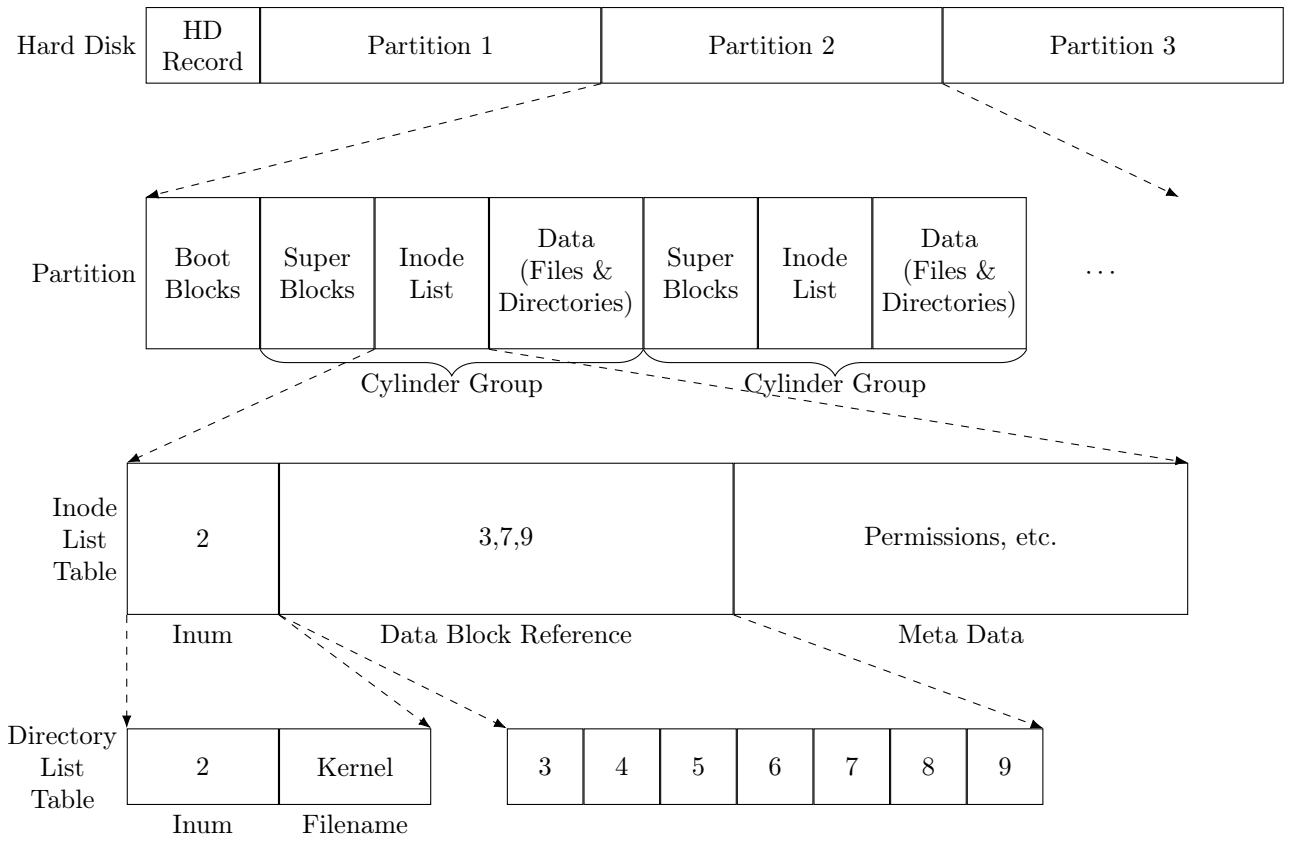
14 The Unix File System

We seek to create a hybrid system that utilizes the best parts of both systems; we want to utilize both the RAM and disk space.

We build our file system based on the following abstraction.



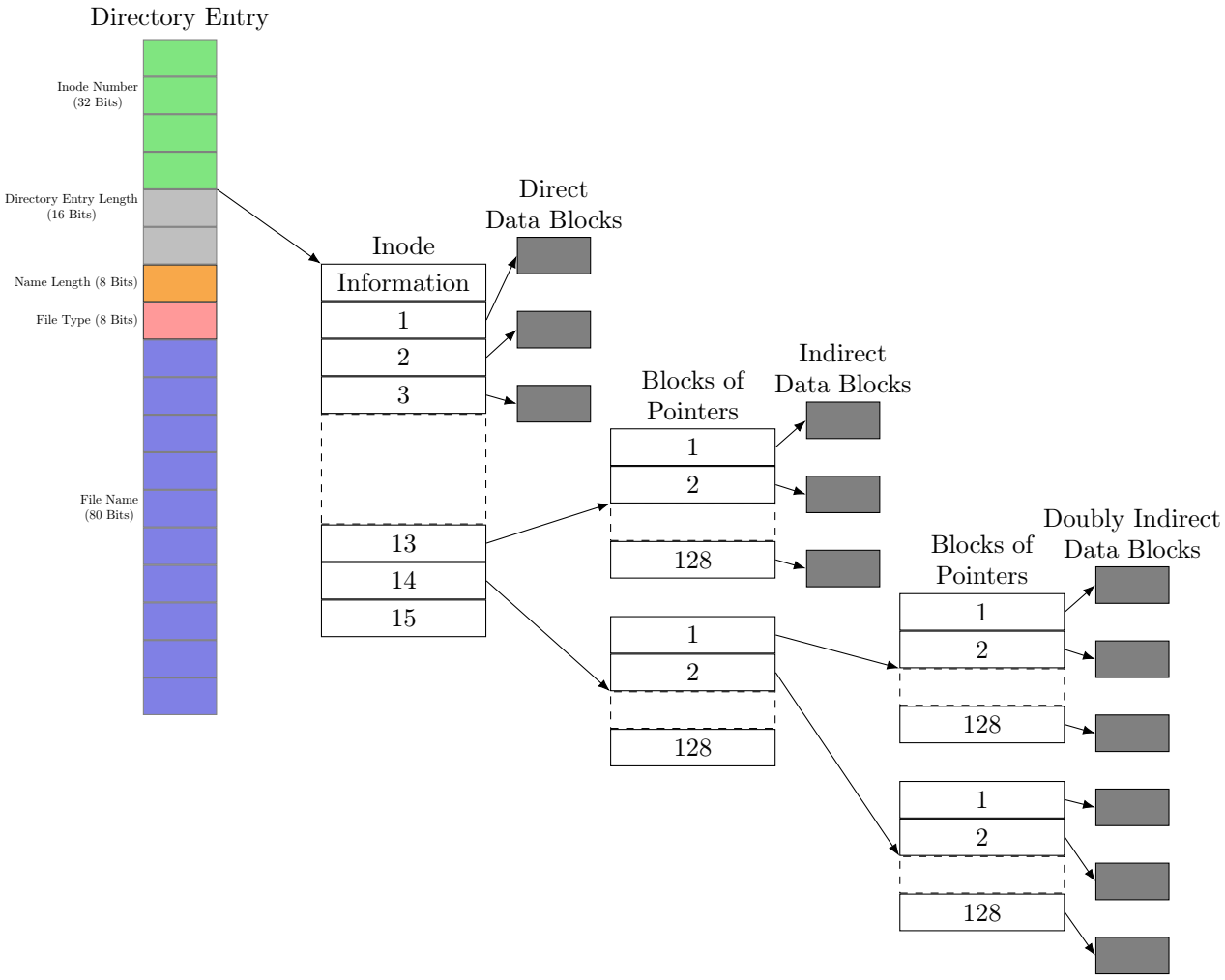
We can then, on the virtual file system layer, implement:



SECTIONS

- Super Block := file system metadata including the root block number
  - Block Bitmap maps index to FREE (0) or ALLOCATED (1) for all blocks
  - (in BFFS) Inode Bitmap maps index to FREE (0) or ALLOCATED (1) for all inodes
- Inode List/Table table of fixed size inode entries that store file metadata and ptr to data
- Data groups of blocks are divided into sectors
  - these are actually tracks, since they can be read quickly
  - these cylinder groups form the partitions of the file system
  - the first directory is left empty since unlink operates on the previous
  - a file referenced by a directory name is a hard link

14.1 Inodes



The data block array stores the block numbers of partitions of data

- 0 means the section is empty
- a file with many 0's is referred to as holey

Inode Metadata (in order):

- owner (32-bit)
- timestamp (last modified time)
- access time (mtime, sysclock)
- inode change (ctime)
- permissions
- file type (directory/regular file/...) (cached, since this one doesn't change)

There are a bunch of new problems though:

1. slow copying (fixed with binary trees)
2. file data is in a linked list, so parsing is slow
    - thankfully, lseek() is O(1)

- Berkeley Fast File System addresses this with an inode bitmap held in RAM

3. There is an arbitrary limit on the size of our file!
   - a 10 block array can store 10 blocks $\times$ 8192 bytes/block = 82 kB
   - We solve this with an indirect block which points to a block of block pointers. This adds 1023 blocks $\times$ 8192 bytes/block + 82 kB = 8 mB
   - This is still not good enough, so we introduce a doubly indirect block. This adds 1023^2 blocks $\times$ 8192 bytes/block = 8 GB
   - With a triply indirect block, we meet a decent file size (2^{35} blocks = 4TB)
   - If use a 4×10^{12} byte file as a database:
     - we can initialize to zero and avoid any writing
     - space will exhaust; 4×10^{12} may not represent the available space
     - triply indirect block data takes three block accesses

   BUT The above inode approach causes Internal Fragmentation!

How bad is this fragmentation?

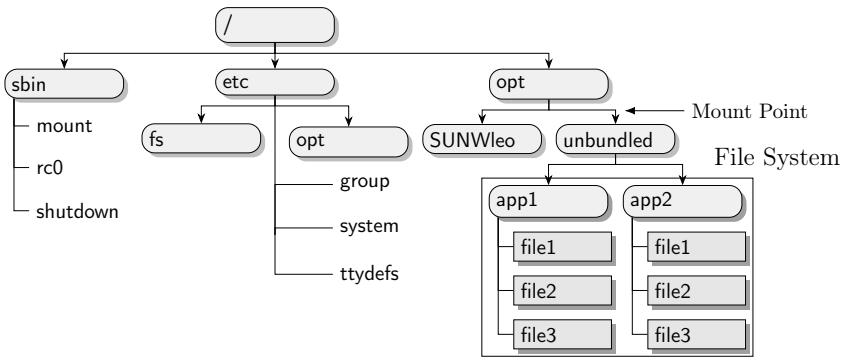
- file size = 0 blocks — 48 bytes wasted for the inode
- file size = 1 blocks — 44 bytes + 8191 bytes = 8235 bytes wasted

If we make a file in the following way:

```
int fd = open("foo", O_CREAT | O_WRONLY | O_TRUNC, 0666);
if (lseek(fd, 1000000000, SEEK_SET)) abort();
else write(fd, "Hello", 5);
```

```
$ ls -l big
-rw-rw-rw- 5 ...
# the file size is still listed as 5!
# wasted 44 (inode) + 8188 (indirect) + 8198 (double) + 8187 (triple) bytes!
```

14.2 Mounting



How could we keep track of multiple file systems? We pass keeping track to the user by *mounting*

- choose one of the devices to be the root
- mount the other onto the root

The Mount Table maps a device name to its path; a device name therefore acts like a symbolic link

We identify files by inode number, but these are not shared among devices! We need a (device number, inode number) pair! namei() can resolve a name into one of these pairs, recursively, but this introduces a scalability issue!

To solve this, we COULD store pairs in the parent directory, but we want to be able to unmount and mount anywhere. The hosting filesystem should have no reference to the hosted and (more importantly) hard links cant cross devices, since devices can have different types of file system

Similarly, but not as mounting, we can put a file in "chroot jail"

```
$ chroot("subdir")
# now the file can only see files below it
# to get out, we can run:

ln /usr/bin/emacs subdir/bin/emacs
ln /dev/tty subdir/dev/tty
ln /dev/dsk/00 subdir/dev/dsk/00
open /dev/dsk/00 // for rw- permissions
ln {some other file to accessible location}
```

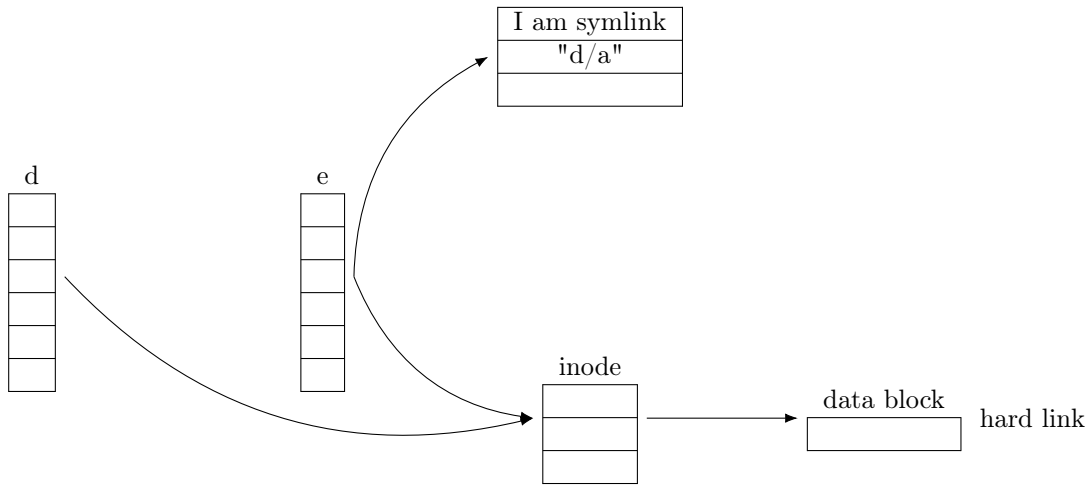
14.3 File Types

UNIX supports the following file types:

File Type	Symbol	Created by	Removed by
Regular File	-	Editors, cp, etc.	rm
Directory	d	mkdir	rmdir, rm -f
Character device file	c	mknod	rm
Block device file	b	mknod	rm
UNIX domain socket	s	socket(2)	rm
Named pipe	p	mknod	rm
Symbolic link	l	lm -s	rm

Symbolic Links (l)

:= a file whose contents contain another file's name. This differs from a hard link, which maps to the inode number.



File names are resolved recursively by namei(). As such, symlinks can introduce problems

```
# ex 1)
$ ln -s . loop
$ ls -l loop
/loop/loop/loop/......

# ex 2) (assume . is 'd')
$ ln -s .. loop
$ ls -l loop
/loop/d/loop/......
```

How do we resolve this?

- find, grep, and similar operations do not follow symlinks
- namei() has a depth limit on recursions

```
$ ln -s linkloop linkloop
$ cat linkloop
# sets errno and returns -1!
```

Device Drivers (b||c)

both character and block devices are windows into a drive

- character — stream-ish
- block — storage-ish (read & write are limited to a fixed size)

For example: the serial port driver sends and receives bytes by wire.

```
$ mknod /dev/ser1 c 59 23
# mknod —— make node
# /dev/ser1 —— path to device
# c —— character
# 59 23 —— device ID
```

These form the basis of file systems! They are antiquated, but common in IOT devices, which dont use an OS.

These drivers suggest users can access data outside of establishes guidelines. In fact, root can access memory and read and write by path!

Knowing this, how can we protect our data?

```
# we want to protect our proposal
$ rm prop.txt
# the file is gone right? NOPE: if there were links, it doesn't get deleted
$ (rm prop.txt; ckst) < prop.txt
# (ckst is self defined to check link count)
# what if someone was already reading it?
$ chmod 700 .
# they could still get into the directory by stealing our disk!
$ shred prop.txt
# overwrite our data 2-3 times with random data
# BUT the OS may have copied the data elsewhere
$ shred /dev/ds2b
# we shred our entire file system!
# but what if another device copied it?
```

MELT THE DRIVE (seriously... even physical shredding leaves readable magnetic residue)

Since apps generally use block writing/perform writes on the block level, the block driver sees a large set of requests; how does it schedule these? We have a few algorithm choices:

1. SHORTEST SEEK TIME FIRST
   (a) perform the shortest seek first
   (b) maximal throughput
   (c) very unfair — allows starvation
2. FIRST COME, FIRST SERVED
   (a) perform the oldest seek first
   (b) low throughput
   (c) very fair
3. ELEVATOR
   (a) a hybrid approach
   (b) perform the closest request in a set direction
   (c) good throughput
   (d) prioritizes locations near the center

   we can use a non-snaking path to avoid this

BUT these algorithms are designed for disks! What can we do for blocks? (which don't have the same seek time?

(4) Anticipatory Scheduling

- is done at the OS or controller level
  - naively:
    $$\{P1, 1000, P2\} \implies P1, 1000, P2$$
  - anticipatory:
    $$\{P1, 1000\} \to \{1000\} \to \text{ dally } \to \{P2, 1000\} \to \{1000\} \implies P1, P2, 1000$$

  A write cannot be switched with a read/write to the same location.

This introduces a new set of risks.

15 Robustness

To discuss risks like this, we need to define a few key terms;

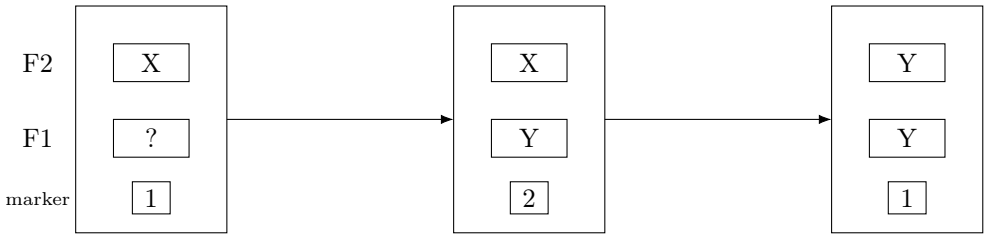
- error ≔ a mistake by the designer or user
- fault ≔ a latent problem in design
- failure ≔ the problem caused when the fault trap springs
- durability ≔ the system's ability for data to survive limited failure
- atomicity
  1. *All-or-Nothing Atomicity*
     - From the point of view of a function's invoker, the sequence either:
       (a) Completes

(b) Aborts such that it appears the action was never started (back out)

- How can we give the READ function AoN atomicity?
  (a) *Blocking Read*: wait and sets return address before READ
  (b) *Non-Blocking Read*: kernel returns if stream is empty
  (c) Non-Atomic: READ waits and blocks until char is delivered

2. *Before-or-After Atomicity*
   - From the view of a function's invokers, the result is the same as if the actions occurred completely before or completely after one another
   - If two actions have before or after atomicity, they are *serializable* := there exists some serial order of those concurrent actions that would, if followed, lead to the same ending state
3. *Sequential/external time consistency*
   - If it appears to the outside that the events occurred in a certain order, the correct result is as if they were executed in this orders.
   - This is only required in some cases.

So take a test editor, say EMACS, and assume it is writing to blocks; what happens if the power goes out while we are overwriting our data? This leads us to our Golden Rule of Atomicity: never delete your only copy.



Our first attempt at atomicity: we create a new file and write the new data there; on completion we switch which is the active file.

- But what if we lost power mid-swap? We wouldn't know which is the active data?
  We can solve this in a probabilistic sense with checksums post fail
- But what if it a block write isn't atomic? Write from A to B is 3 stages: A $\rightarrow$? $\rightarrow$ B
  We can make an atomic write using 3 blocks!

File Data Contents

F1	A	?	B	B	B	B	B
F2	A	A	A	?	B	B	B
F3	A	A	A	A	A	?	B

time $\rightarrow$

Algorithm:
1. write 3 in series
2. use best 2/3 to choose
3. choose block 0 if they all differ

Since we have atomic block write, we can discuss file system robustness on a larger scale:

We use a Lampion-Sturgis failure model:

1. storage writes may fail
2. storage writes may corrupt other blocks
3. storage blocks may decay spontaneously
4. a read can detect a bad block (using checksums)
5. errors are rare
6. reports can be done in time

We also need to establish a few file system robustness invariants:

1. every block serves at most one purpose — failure allows a program to overwrite another program's data
2. all referenced blocks are properly initialized — failure allows an uninitialized block to look like a pointer to a random block
3. all referenced blocks are marked as "used" – failure allows data to be overwritten
4. all non-referenced blocks are marked as "free" — failure leaves an unused block marked as used → data leak

The failure of most of these would mean the loss of data—except number 4, so our failure hinges on number 4.

Take a risky operation. . . say rename("d/a", "e/b")

- fsck() will catch our erros, but it is insanely slow!
- It prioritizes the inode data and moves unreferenced inodes to a lost & found
- File permissions are set to kernel only

This could be done in one of a few ways:

BAD Algorithm:
1. block a → RAM
2. block b → RAM
3. update blocks
4. block a → flash
5. block b → flash

GOOD Algorithm:
1. read block a into RAM
2. read inode into RAM
3. read block b into RAM
4. update blocks and increment link count
5. write inode to flash
6. write b to flash
7. write a to flash
8. link count -= 1 and write inode to flash

BAD:

- Failure between 4 & 5 would lose data!

GOOD:

- Instruction (5-6): only old link exists but lc = 2
- Instruction (6-7): the old and new copy exist and lc = 2
- Instruction (7-8): only new link exists but lc = 2

The good one is better because we would prefer to lose space over data!

We can now look at abstracting a single block write into multiple blocks:

15.0.1 COMMIT RECORDS

- document writes such that writes are atomic
- put commit records and all writes into a well recognized location

15.0.2 JOURNALING

Journal

...	A'	B'	CR	DR

Cell Storage

			B
	A		

Algorithm:

1. Write A' to journal
2. Write B' to journal
3. Write CR to journal (BEGIN)
4. Copy A' to cell storage (CHANGEA)
5. Copy B' to cell storage (CHANGEB)
6. Write DR to journal (OUTCOME)

We then reorganize, and we're done.

Consequences of Failure:
Pre BEGIN: No effect.
Post BEGIN: Never initiated.
Post OUTCOME: Write Complete.

We keep cells in RAM and only copy to disk on write

- \- Wastes storage and will eventually run off disks
- \+ Solves many inconsistency problems
- \+ If mostly writing, avoids seeks

Since we can fail during reboot, our recovery strategy must also be *idempotent*, := execution one time is the same as executing any number of times.

We have to write each entry twice... what if we do a large write? Failure seems likely.

$\Longrightarrow$ this is true, but most apps think between writes, so long writes are rare.

What do we do if a directory is replaced with a file & there is a crash?

- Linux uses the idea of "revoke records" to record a negative change in extv4
- Linux also writes

Is this viable for flash?

No — the performance benefit of this is that we ignore seek costs for write, but we know where we are going to write here.

We have two major journaling options: (the example uses write-ahead)

Write Ahead (ext4 option)

1. log changes in data
2. write changes in data
3. write the commit record
4. write the done record

RECOVERY:

- replay commit records

DOWNSIDE:

- the start of our replay can be hard to find

BENEFITS:

- more likely to save last action
- do not need to keep multiple versions

Write Behind

1. log old data values
2. write the commit record
3. write changes in data
4. write the done record

RECOVERY:

- undo the changes

DOWNSIDE:

- we must copy all of our data on each write

BENEFITS:

- old data cached $\implies$ small pre-write benefit
- more conservative $\implies$ more recovery
- no data searching $\implies$ faster recovery

We need to be able to handle the failure of low level operations. We describe two interactions:

1. cascading aborts
   - if a low level operation fails, the high level one fails
   - very automate-able
   - Example: Write-Ahead Protocol Cascading Abort:
     (a) While logging planned writes, error occurs
     (b) Abort record
     (c) Send cascading aborts to higher level functions.
2. compensating actions
   - if a low level operation fails, the higher one makes up for it
   - very flexible
   - Example: Write-Ahead Protocol Compensating Action:
     (a) Log planned writes
     (b) Commit record
     (c) While writing to disk, error occurs. After reboot compensating actions continue writing to cell memory using data written in the log.

There are many different types of corruption which can occur in a file system:

1. gamma rays can flip a bit
   - this happens roughly once a week
   - ECC memory with a parity bit can fix single flips and catch doubles
2. Drive Failure
   - this includes physical damage to the drive
   - companies give an annualized fail rate to give % fail chance for a year of operation
   - we can use SMART to check condition metrics; f we find a bad sector, we have replacement ones
3. User error
   - ex) rm * .o (remove all files) instead of rm *.o (remove all object files)

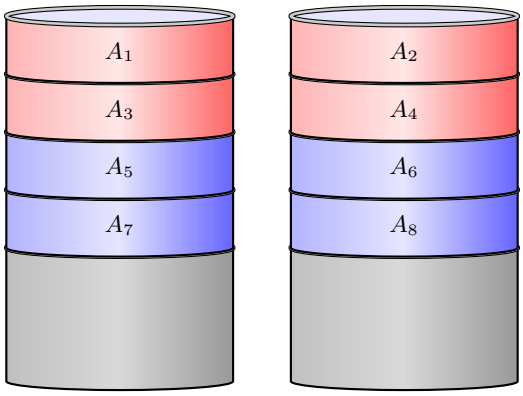
4. OER Errors
    - include configuration errors, which are the majority of errors
    - application and OS errors also quality

We can catch and fix power failure with our old method. We can catch drive failure with a log structured file system, but to fix it we need to make a copy.

15.1 Redundant Array of Independent Disks (RAID)

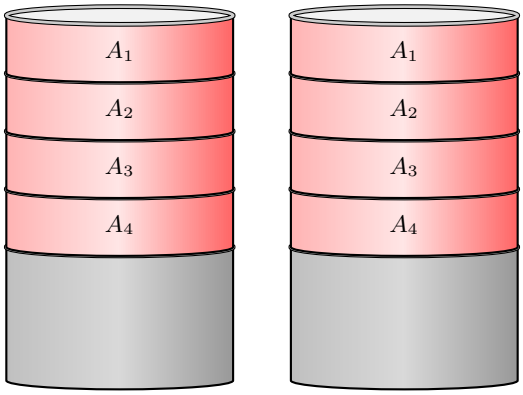
RAID was originally developed to save money by aggregating many small drives into a large one It is now one of our most useful memory tools!

RAID-0



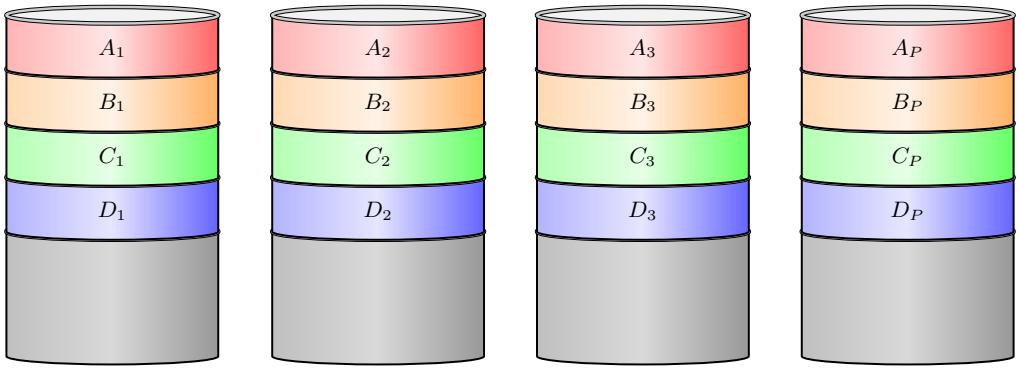
- We concatenate physical disks to form a large virtual one.
- We stripe the data to increase concurrency for read and write.

RAID-1



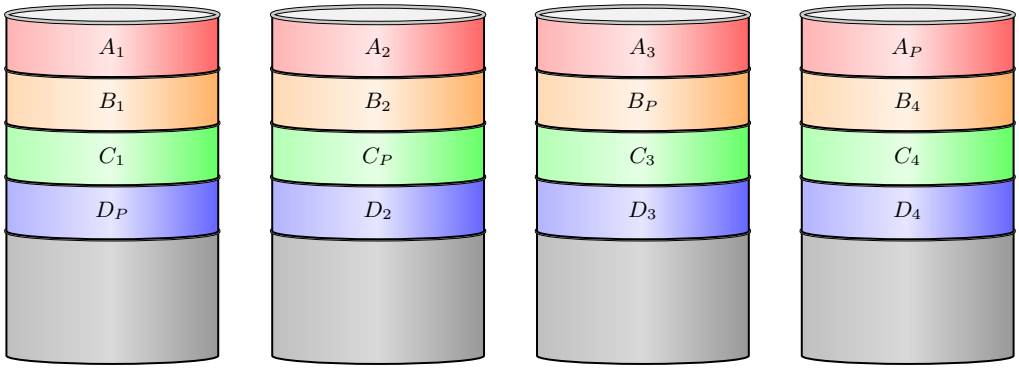
- We mirror half of our disks.
- We double write speed and half storage.
- We can now survive with half our data lost.

RAID-4



- We reserve one of our disks for *parity bits*. Parity bits are calculated with exclusive-or ($\oplus$)
- When a drive fails, we go into degraded mode;
  - We especially need to guarantee robustness here.
  - We can use a hot spare to replace a corrupt drive.
  - We can now add drives easily.
  - We can now recover from a drive loss.

RAID-5



- We stripe within a RAID 4 system.
- This is faster but more difficult to add drives.
- The book says these have almost entirely replaced RAID 4, but Eggert now seems to disagree.

All RAID uses a full-stop model; on detection of an error the operation stops. It can use checksums that include the data location

16 Virtual Memory

What methods of protecting memory from corruption by bad processes do we have?

1. Hire better programmers
2. Enable subscript/NULL checking

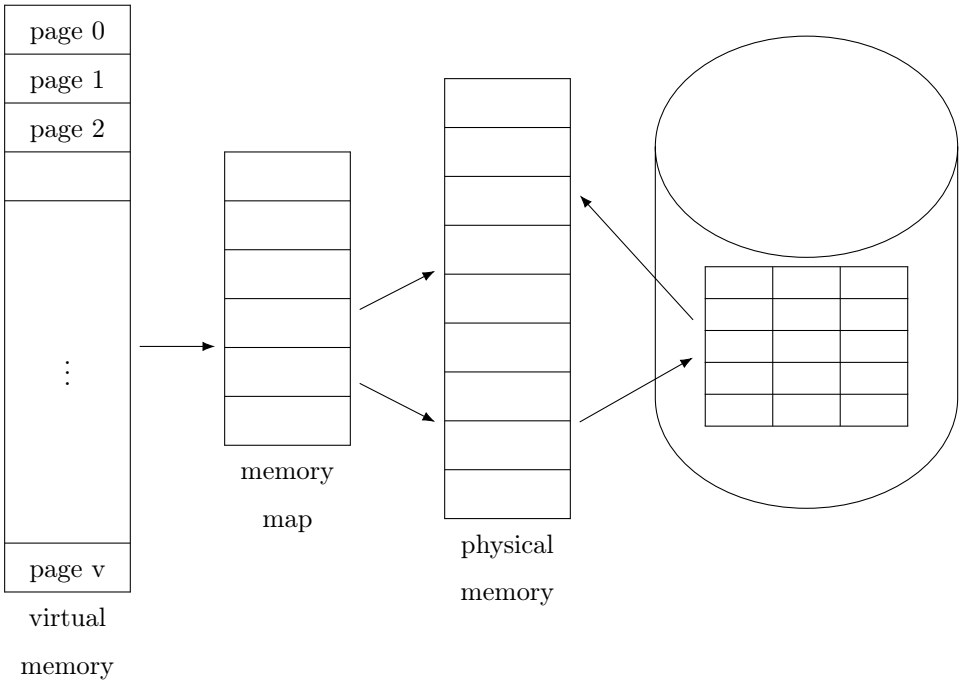
- requires a language that stores array sizes
- conditional branches and checking make this very slow

3. Base & Bound Registers
   - simple: only requires 2 registers
   - good for batch environments, bad for dynamic
   - requires contiguous memory per-process and thus can cause external & internal fragmentation
   - not conducive to sharing; requires repetition of code with copying for each process
4. Segmented Memory
   - a hardware table mapping indices to a (array, index)/(segment#, offset) pair
   - does not require adjustment of locations
   - still has external fragmentation

All of these have downsides—a better and the most common one is virtual memory. This is for a few reasons:

- frees programmers from worrying about physical locations and fragmentation
- prevents processes from illegal memory access
- lets processes and threads share memory
- lets VM be larger than actual memory
  while this is technically true, it can cause rapid switching of pages (*thrashing*)

How does it work?



- Each thread has a register %cr3 that points to the page table/memory map.
- The page table maps virtual addresses to physical addresses.

Note that Intel keeps the specific content format of the page table secret, but this abstraction is enough to work with.

Pages are supported by the hardware — x86-64 has a CTRL register that stores page sizes. How do we choose the size of virtual pages?

- large pages → small table (saves time & space)
- small pages → large table (less fragmentation)

Are loops allowed? (virtual → physical → virtual)

- This would allow us to look at the table, so to prevent memory cheating, Linux doesn't even let us look at the table.

How do we edit the page table then?

```
int mmap(void *vir_addr, size_t len, int prot, int flags, int fd, off_t offset);
// returns the virtual address of the new page (not necessarily the one you give)
// flags
//  fixed — do not give me an address other than what I asked for
//  private — other processes are not allowed to see our memory
```

What if the length is not a multiple of the page size?

- this is not supported; use int getpagesize()
- this limits pages to INT_MAX, so we use long sysconf() to use larger pages

Are all protections reasonable?

rw-: ordinary data

r--: read only data such as variables marked const

r-x: code (read is given because they are open source)

rwx: dynamically generated close

- this allows buffer overflow attacks
- the stack is often still rwx because of legacy

---: useful for catching addressing errors

- can be used for guard pages on either side of the array
- can be used to guard a recursive frame from modifying the base pointer in stack
- IS used on 0 to protect from NULL accesses

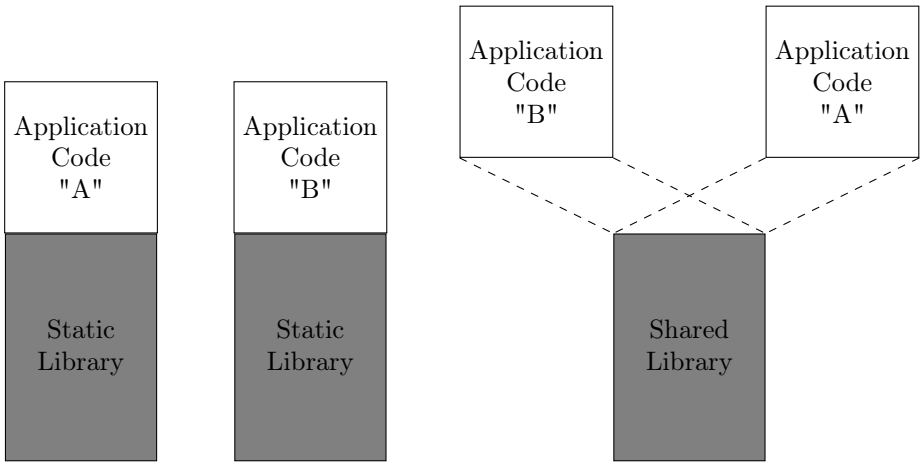
Note that 'x' is expensive, so low end chips with no security risk often combine 'x' with 'r'.

mmap trivia:

- The most common file to map is /dev/zero. This functions as an input only stream of zeroes
  - read — succeed and fill page with zeroes
  - write — succeed without doing anything
- Two processes cannot map to the same address at the same
- If a file has been mapped, it will not be deleted while still mapped

- A mapped page is not intrinsically tied to any file
- It allows us to use dynamic linking of libraries:

```
$ gcc main.c -o executable -lm;
# -lm links to math library
```



Processes share only an image of the code they share, so the virtual addresses do not match, so how do we map virtual addresses to physical addresses?

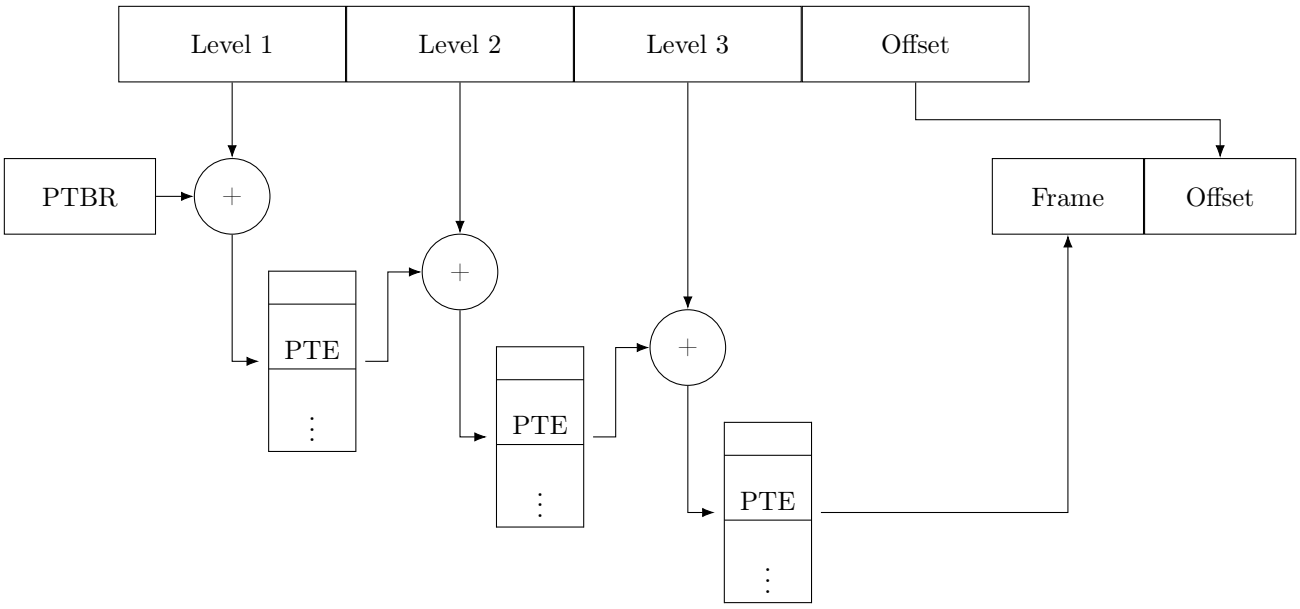
- We use the *Global Offset Table* := rwx self-modifying code that gives the offset for machine language system calls.

17 Memory

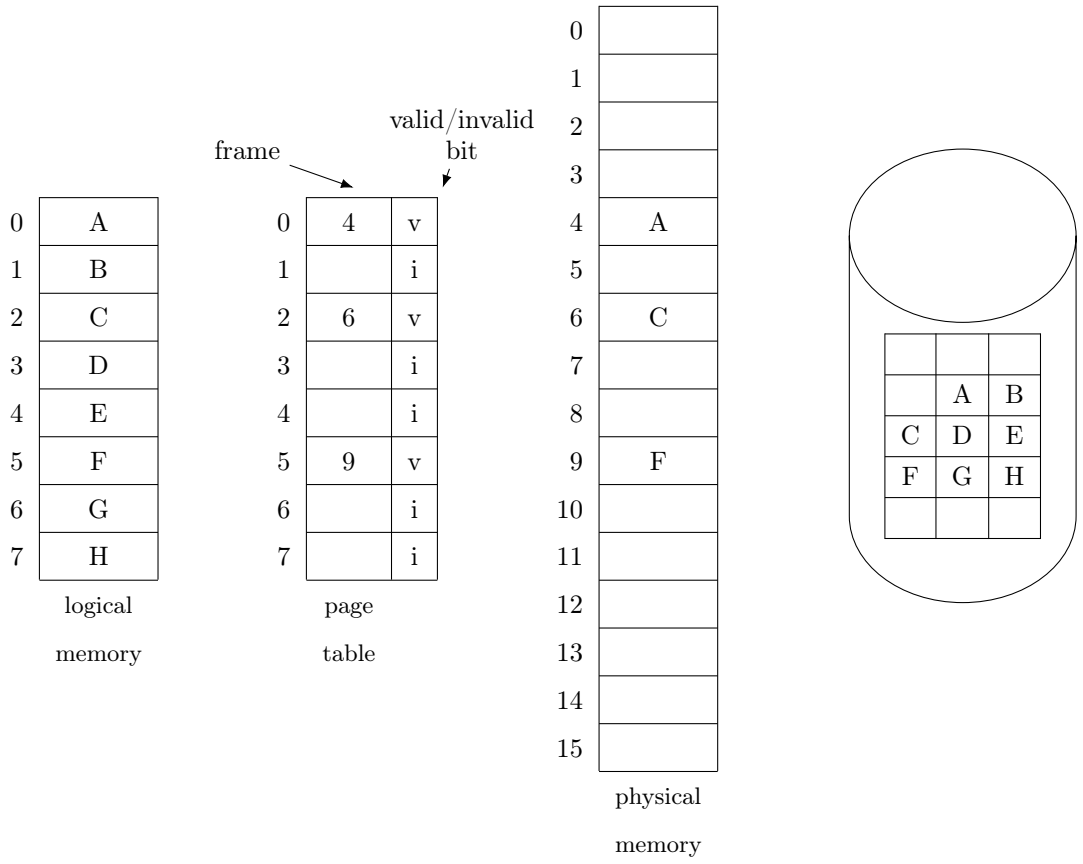
When utilizing virtual memory, we often have to deal with *page faults* := the failure of the hardware's rule for page lookup because:

(a) the user lacks the permissions to access the page
(b) the page is invalid
(c) the page does not exist

The hardware attempts to look the page up in the page table using the index of the form



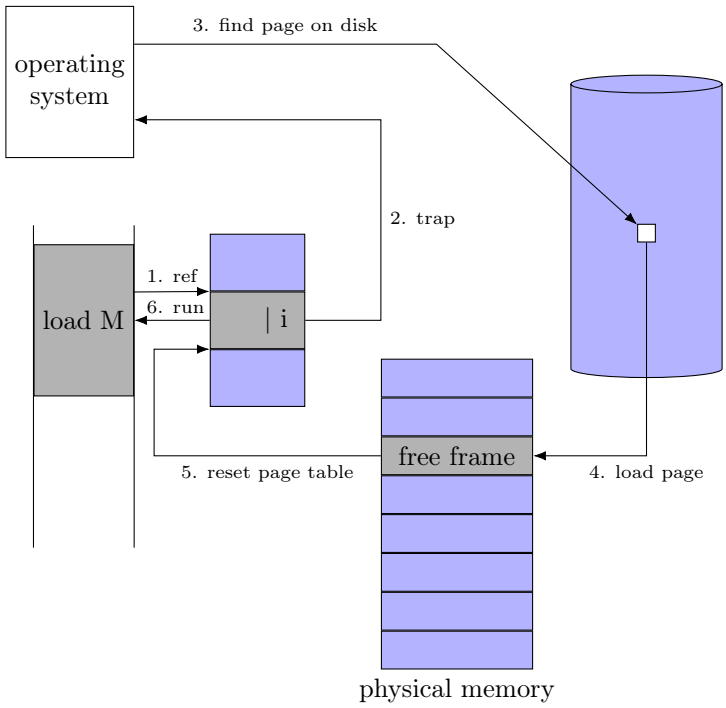
Which then finds the page like so



But if the page is not in memory, we have a page fault! The page fault is treated like any other fault:

- the OS looks in the interrupt source vector (aka trap table) and jumps to indicated page; note that we could end up trapped if this page faults as well
- The kernel has a few options on a page fault

1. the kernel level fix: assume the program is buggy and kill it
2. the program level fix: arrange the stack as if there were no fault and send signal
3. the data level fix: change the page table entry and jump before the fault (the slowest)



The third option is called *paging*; we use it to simulate a big machine on a small one (for instance).

The kernel memory where the physical pages are cached from the disk is the *swap space*.

17.1 Page Replacement

```
// we assume that we have implemented the following functions:
int swapmap(int process, int virtual_address);
// this returns the disk address in the swap space or FAIL
tuple (process, va) removal_policy()
// this returns the process and virtual address of the next victim page
int pmap(int va);
// refers to the physical address mapped to by a given virtual

void pfault(int va, int proc, int access_type...) {
  if (swapmap(proc, va) == FALL) kill(proc);
  else {
    (vic_proc, vic_va) = removal_policy();
    int vic_pa = vic_proc->pmap(vic_va);
    vic_proc->pmap(vic_va) = FAIL;
    /// write vic_pa to flash at location swapmap(vic_proc, vic_va)
    // read vic_pa from flash at location swapmap(proc, va)
        proc->pmap(va) = pa;
    }
  }
```

How do we decide the removal policy? We need some heuristic for when to swap a page:

(a) Nobody needs the page (and there is no dynamic linkage to it)
(b) Page has not changed since load (and therefore RAM data is valid)
(c) Page is not needed for a while (based on an approximation

Here follows a few simple page replacement policies:

ORACLE

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	2	2	2	2	2	2	2	2	2	2	7	7	
	0	0	0	0	0	0	4	4	4	0	0	0	0	0	0	0	0	0	
		1	1	1	3	3	3	3	3	3	3	3	1	1	1	1	1	1	

page frames

Figure 5: 9 replacements

FIRST IN, FIRST OUT

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	0	0	0	7	7	7	
	0	0	0	0	3	3	3	2	2	2	2	2	1	1	1	1	0	0	
		1	1	1	1	0	0	0	3	3	3	3	3	2	2	2	2	1	

page frames

Figure 6: 15 replacements

Note that with FIFO, increasing the number of pages increases the number of replacements; this is called *Beladay's Anomaly*, and it occurs because FIFO is not a *stack algorithm* (ie the state of a smaller cache at any point is not a subset of the larger cache),

FIFO is easy to implement, since the table is in the kernel and the kernel retrieves pages; we just keep a table of when each page was brought into RAM.

LEAST RECENTLY USED

reference string

7	0	1	2	0	3	0	4	2	3	0	3	2	1	2	0	1	7	0	1
7	7	7	2	2	2	2	4	4	4	0	0	0	1	1	1	1	1	1	
	0	0	0	0	0	0	0	0	3	3	3	3	3	3	0	0	0	0	
		1	1	1	3	3	3	2	2	2	2	2	2	2	2	2	7	7	

page frames

Figure 7: 12 replacements

LRU is more difficult to implement, since the kernel does not take over on each access and therefore has no chance to edit the last accessed time. We can address this in one of a few ways:

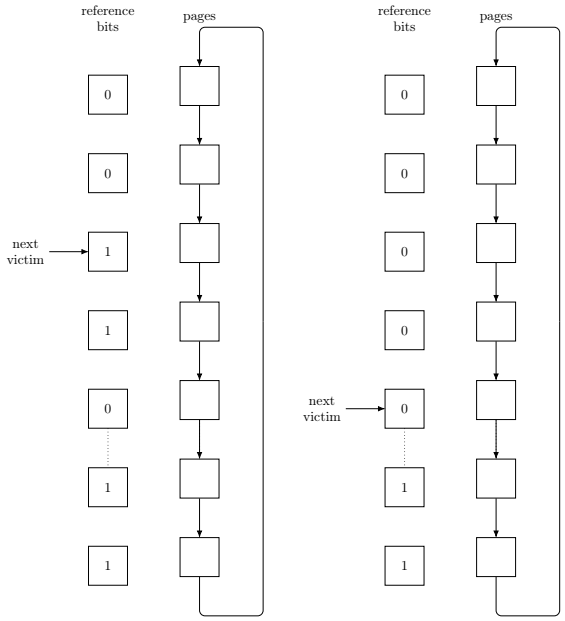
1. get hardware support
   - this is slow and rarely used
2. periodic invalidation
   - Procedure:
     (a) the kernel periodically invokes a timer fault to mark all pages as invalid

(b) any succeeding access will then cause a fault which allows the kernel to update times

- for clock precision, this is done in hardware after about a second passes. The time choice is important:
  - a long gap means less interruption
  - a short gap means better approximation

Possible Optimizations:

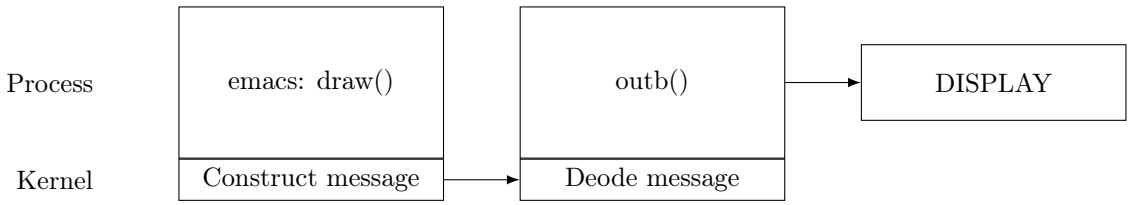
1. Demand Paging
   - Procedure:
     1. start with no pages in RAM
     2. c startup routine (cstrt()) causes initial fault
     3. call to main causes a second fault
   - This strives to lessen wait time by skipping unnecessary copies
     - best case: we only need the two pages
     - worst case: we need all the pages. This performs worse than the standard approach since it prevents us from batch copying.
2. Copy on Write
   - only give processes read permission and do not copy entries write is attempted
   - this is done by vfork() to get the benefit of threading without race conditions
   - vfork() is equivalent to fork, except:
     (i) Parent and child can share memory.
     (ii) Parent is frozen and cannot run until child either exits or execs
3. Dirty Bit
   - a special bit in a page table entry indicates whether the page has been modified.
   - The bit is set to 1 if it has been modified since last load, 0 if not
   - we only need write a victim to memory if the dirty bit is 1
   - often implemented by making pages O_RDONLY so that the first access is a fault
   - sometimes implemented using the clock algorithm, which works like the elevator



18 Remote Procedure Calls

We start by observing a virtual usage of the client/server approach

X PROTOCOL



The X server handles the keyboard, mouse, and display of the system

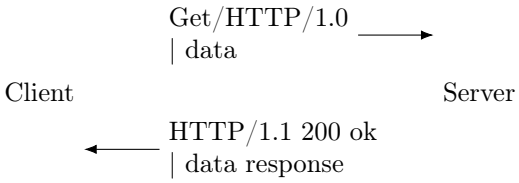
This is an example of a *distributed system* in which the server effectively acts as the system kernel. and uses *remote procedure calls*. Our goal in these systems is modularity rather than virtualization.

This has a few distinction from standard procedure calls

1. the caller and callee do not need to share an address space
2. passing by reference and pointers are not allowed
3. the client and server need not be on the same architecture (i.e. x86 vs x86-64 or x86-64 vs SPARC64)

But the last of these distinctions introduces long size and endian-ness problems. We address these by *marshaling* commands. This adds a layer of abstraction on the commands.

Messages can be passed in multiple bit pattern forms such as XML (slow), JSON, and IEE 754. One such usage is HTTP:



What could go wrong?

- messages can get lost/duplicated/corrupted
- the network/server can go down (or be slow)

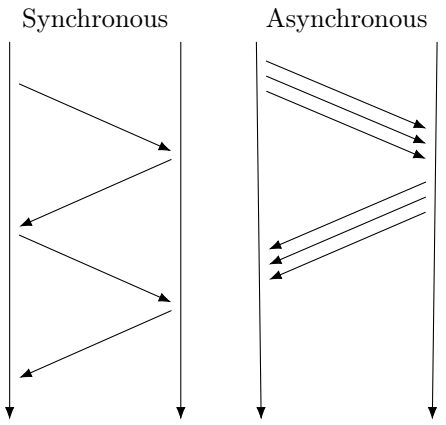
& these appear identical to the client!

We deal with these issues as two categories:

1. Corruption
   (a) better checksums in messages
   (b) end-to-end (rather than link) encryption
      - *link encryption* := devices decrypt and re-encrypt
      - *end-to-end encryption* := data stays encrypted for the entire journey
   (c) resend messages on detection

2. Network Issues
   (a) At-Least-Once RPC (keep trying) — good for idempotent information
   (b) At-Most-Once RPC (log on error) — good for transactions
   (c) Exactly-Once RPC (don't make mistakes) — very (some would say too) difficult to implement

Another downside is INEFFICIENCY. We have 3 major approaches to combatting this:



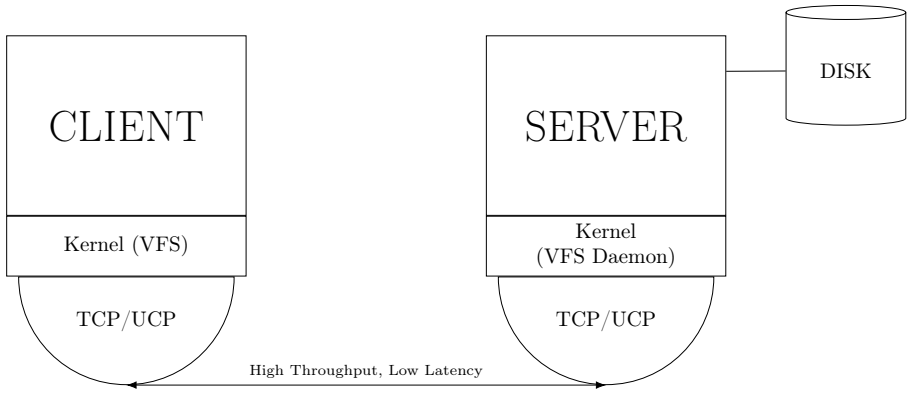
1. Asynchronous System Calls (Pipelining)
   - allows for more useful work time
   - complicated partial failure
   - out-of-order responses
   - requests must be independent
2. Cache-on-the-Client
   - cache responses to likely requests
   - common with web browsers
3. Pre-Fetching
   - guess a response prior to receiving methods
   - hogs resources

We can use these ideas to implement

18.1 Networked File Systems

We analyze *NFS models* := a Linux-type system which sends requests (like open, read, lseek...) to a network. Two commonly used versions are

v3: useful in machine rooms, since it uses *UDP* := establishes a connection before sending packets
v4: useful for the internet, since it uses *TCP* := sends packets without establishing a connection



The kernel uses a VFS (such as btts, ext4, etc) to

- transparently marshal procedure calls
- send and receive unencrypted packets (encryption started with VFSv4)

Basically, we emulate a local file system with a network! This is great for modularity (we don't even need the same underlying architecture, remember?) but now we are relying on a network; what if it goes down?

For example, note that read is short enough to avoid failure by interrupt; but what if the network goes down?

- Option A: read() hangs — this exposes us to infinite hangs!
- Option B: send a ^C to interrupt — apps which assume a return from read get complicated!

The system administrator has to choose, but we're wrong either way.

So what do we do?

- In short, don't let the network go down, but we design our protocol carefully just in case

Even without a crash, close() is costly, as the client waits for all outstanding requests

NFS Primitives:

```
# all elements are fixed-size
# fh and attr are concatenated and cast to integers
MKDIR(dirfh, name, attr) -> fh & sttr
REMOVE(dirfh, name)-> status
READ(fh, offset, size) -> data
LOOKUP(dirfh, name) -> fh & attr
# example message:
LOOKUP("/usr/local", "/bin") -> 4728 + {output of ls}
```

An nfs *file handle* uniquely identifies a file on a server

- it survives renames and reboots
- on Unux, it is a concatenation of device number, inode number, and serial number

They cause the server to be stateless, as they do not rely on the state of the client So will a concurrent WRITE and RENAME cause problems?

- NO; the file handle does not change

Will a concurrent REMOVE and WRITE?

- YES, which violates the Linux standard! (would wait to delete, but NFSv3 is stateless)

There are two cases:

1. REMOVE & WRITE are from the same client
   - save a dummy request to a temp file names ".NFS#"
   - the client is responsible for the cleanup
2. REMOVE & WRITE from different clients
   - set errno = ESTALE

A related error occurs with concurrent REMOVE & WRITE or WRITE & WRITE. With the system as we currently understand it, the NFS would not recognize the stale request and would write to the wrong file. Therefore we need a serial number to represent what version of this file handle a file is.

19 Security

The original NFS assumed both the client and server kernels are trustworthy and that the server UID's match with the client UID's.

How does NFS address the superuser? (obviously every root can't have root permissions)

- we make user 0 "nobody" and give them minimal privileges

This is simplistic: the naming scheme in reality is complicated and involves strings.

Soon after, we developed authentication (via Kerberos, etc) to encrypt packets. We need this authentication to prevent attacks against:

1. Privacy (unauthorized release of information)
2. Integrity (tempering with unauthorized data)
3. Service (Denial of Service)

Our goals are therefore:

- Allow authorized access (+)
- Disallow unauthorized access (-)
- Good performance (+)

Positive goals are easy because they get reported, but negative goals are more difficult

When attempting to build a secure system, we perform *Threat Modeling*; the threats to the security of a system include:

- insiders
- social engineering (generally non-coding)
- network attacks (virus, DOS)
- device attacks (USB) attacks

The last is so common that it is recommended people use a "usb condom".

Our security within a system is trivial; we simply chroot untrusted processes so they think they have access to the root. Even this is not perfect, however: for instance, processes P and P' can communicate with gettimeofday(), where P chews up computer time and P' checks the clock, transmitting data with only clock change. In this way, P can send messages using morse code. This is called a *covert channel*, and they are effectively unavoidable; we instead only focus on critical attacks

To prevent these, our system requires 4 major properties

1. Authentication (password, credentials)
2. Integrity (checksum)
3. Authorization (access control list)
4. Auditing (log)

and obviously correctness and performance.

19.1 Authentication

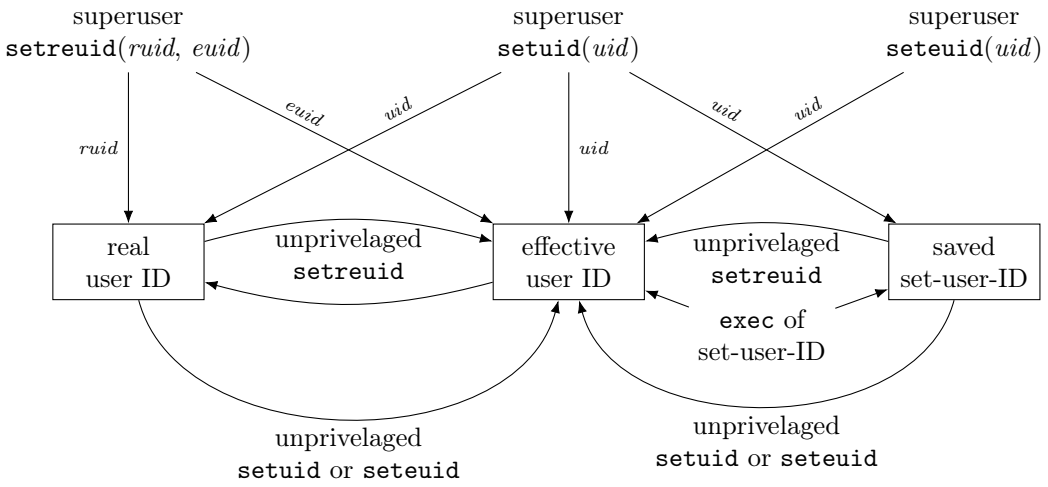
We utilize *authentication* to prevent masquerading. Authentication is based on 3 major properties:

1. who the principal is (ex. retinal scan)
2. something the principal has (ex. smart card)
3. something the principal knows (ex. password)

but these are often equivalent, as we can bootstrap from one to the other. For example, SEASNET is based on (1) but uses (3) for verification.

There are two phases of authentication:

1. *external authentication* := protection from the outside world.
   - the system verifies that the user is a part of the system
   - examples include a trusted login agent (OS), key, password, biometrics, MFA
   - attack types
     - theft of hardware tokens
     - fraudulent servers
     - chain breaking
     - buggy login agents
     - well known default passwords
     - SIN attacks (SIM card backup)
     - password recovery attacks
     - bad routers
   - From an OS standpoint, many of these are beyond our purview.
2. *internal authentification* := protection from inside the system
   - the system verifies the user is a specific member of a system
   - by design, internal authentification is much more efficient
   - common type of attack is a PATH injection:
     - Internal authentication in Linux comes in the form of UIDs
     - The UID is the current user, and the EUID is the program's creator
     - There are select programs (called setuid programs) which this is not true for; these are marked with a UID execution bit of "s"
     - These programs are automatically trusted by the root (ex. /usr/bin/passwd)
     - a PATH injection is a bad user setting the path such that a system() call behaves poorly
     - For that reason, we avoid usages such as system("ls")



Shells are at risk for PATH injection, since they implicitly call system(), but our c programs must be reliable (especially in terms of buffer overflow). Sometimes, however, there isn't much we can do (take setuid()). At least setuid() requires root access

Once we verify identity, we move on to:

19.2 Authorization

There are two schemes for access to resources:

1. *Direct Access*

   ```
   char *p = mmap(fd, offset, etc...);
   ```

   The system checks authority once and maps a pointer onto the process address space. Thus is done with page table manipulation. Properties:

   + high performance
   - easy resource corruption (by bad actors and race conditions)
   - necessitates hardware support

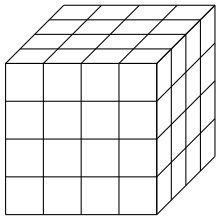
2. *Indirect Access*

   ```
   int fd = open(file, ...);
   read(fd, ...);
   ```

   The system issues a request to handler or system call to check on every attempted resource access.

   - low performance
   + avoid corruption
   + easier to revoke permission

We can think of an operating system as a set of principals and resources:



- We require 3 parameters to check whether a given user can access a given resource.
- This array seems massive, but the data is not random, so there must be a more efficient way to represent the information.

It turns out we can represent it in two dimensions!

We do this with an *Access Control List*

```
$ getfacl .
user: rwx
group: r-x
other: r-x
$ setfacl -m g:tas:rwx .
$ getfacl .
user: rwx
group: r-x
other: r-x
tas: rwx
# we have appended the group "tas" to the end of the directory's list
```

Associated with each file is a list of access rights under the control of the file's owner. Using this, we can form our own ad-hoc groups outside of the built-in ones.

This is not the only way to represent authorization, and isn't perfect for every purpose; we may need to capturing rights accurately or letting ordinary users specify rights. ACLs in general are not flexible enough for large organizations. Instead we can use role-based access control (RBAC); this model more complex but secure than ACLs.

The model: keep a distinction between users and their roles.

For example, the user "eggert" could have multiple roles assigned to him: ie eggert as cs-faculty and as sys-admin

- cs-faculty allows him to change grades.
- sys-admin allows him to make changes to other users' permissions.

These permissions are usually mutually exclusive, and roles can have sub-roles, which may share some permissions.

In both models, each file lists permissions for each user; these are *resource-centric*. We can instead use a principal-centric model called a *Capability Model*, where users hold an object with an unforgeable unique ID representing a resource.

Linux utilizes both approaches

- Each descriptor has its own separate permissions from the file (capability)
- Each file is resource-centric and holds its own permissions

```
fd = open ("abc", O_RDONLY)              <--- code
r--                rwx                   <--- permissions
read-only          read, write, execute  <--- description of permissions
// the file "abc" has the permissions rwx
// the fd is the capability with permissions r--
// any action done with fd can only read, even though the file is rwx
```

Note that the flags could include O_PATH, which gives no permissions, only showing the existence of the file. We could then duplicate the file descriptor using dup2, but then you'd have 2 useless fd's.

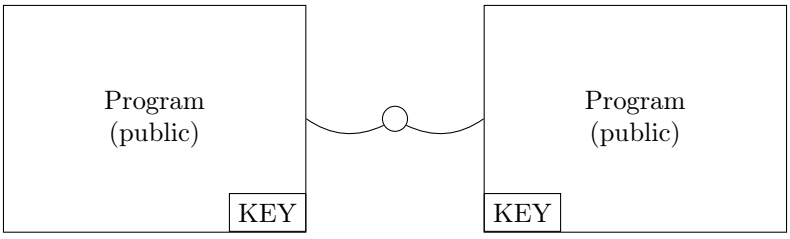
We can therefore run:

```
int stat(const char *path, struct stat *buf);
// return information about the file that fd points to
```

We have addressed *auditing* with logs and *integrity* with checksums already.

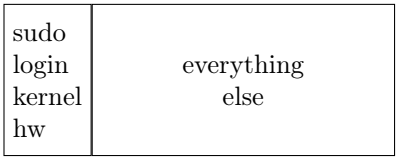
20 Encryption

When we send data over an untrusted network, we need to encrypt the data! We do this according to *Kerchoff's Design Principal* := minimize what needs to be kept secret.



The question of security is a question of trust. Most programs run by the OS are not trusted (cat, grep, sh, gcc, make...) but a few are (login, sudo, su...), and can therefore call setuid()

If we are worried about security, the latter are the risk: how do we choose who to trust?



We call the set of all trusted programs the *trusted computing base*. We seek to minimize this.

BUT can we really trust anything? Take Ken Thompson, creator of Unix. He once wrote a Turing Award winning paper called "Reflections on Trusting Trust". The paper described a bug he built into Unix.

```
// in the login program, Ken added the following code
if(strcmp(user,"ken")==0) uid = 0;
// however, this would be easily found out, so in gcc, he adds:
if(compiling(login.c)) generate strcmp
// now no one looking at the login source code can see the bug
// he also adds in gcc
if(compiling(gcc.c)) generate generate strcmp
// he compiles gcc and edits the code back — the code is now gone!
```

Couldn't this be found in the assembly? NO:

- he can modify the code for objdmp
- he can even make it so that using other code injects bugs!

The moral of the story is, "You need to trust your tools" (as an aside, it also shows gcc really should be in the TCB).

With this principal in mind, we model secure message passing:

$$A \to B \text{“I'm Alice”}^K$$

This is BAD; it is subject to replay attacks. Thus we need a *nonce* := a nonsense string used to verify possession of a key:

$$A \to B \text{ “I'm Alice”}$$
$$B \to A \text{ “<nonce>”}$$
$$A \to B \text{ {“<nonce>”}}$$
$$B \to A \text{ “OK”}$$

We can now effectively verify identify, so how do we use this to send messages?

- we use a Hashed Message Authentication Code

$$A ß B M \parallel \text{HMAC}(M, K)$$

how do we find an HMAC? We call our cryptographic friends for a *(Cryptographically) Secure Checksum Algorithm*, defined st if $SHA(M) = N$, it is hard to find $M' \text{st} SHA(M') = N$. We then let HMAC(M, K) = f(SHA(K||M))

Now we are 100% secure, right?

- Of course not: SHA's can be broken — we are already on SHA 2.something

Distributed security can also be done with a virtual network inside of one Computer:

Typical authentication via ssh is multiphase :

1. establish connection with public/private key encryption
2. use the encrypted nonce as a shared secret key

Shared secrets are the model we used above; the public/private model is as such:

1. a message $A \to B$ is encrypted with B's public key

2. person A can then decrypt the message with their private key

So it is better to use this first, since it is slower but more secure. note: verifying a public key is not 100% exact.

This model is utilized by the *RSA encryption algorithm*

- The keys for the RSA algorithm are generated in the following way:
  1. Choose two distinct prime numbers p and q.
     - p & q should be chosen at random
     - p & q should be similar in magnitude but differ in length by a few digits
     - p and q are kept secret
  2. Compute $n = pq$.
     - n is used as the modulus for both keys
     - the length of n expressed in bits, is the key length
  3. Compute $\Phi(n) = (p-1)(q-1)$
  4. Choose an integer e such that $1 < e < \Phi(n)$ and e and $\Phi(n)$ share no factors besides 0.
     - an e with a short bit-length results in more efficiency
     - the most commonly chosen value for e is $216 + 1 = 65,537$
     - the smallest (and fastest) possible value for e is 3
  5. Determine $d \equiv e^{-1} \mod \Phi(n)$
- The *public key* consists of the modulus n and the public (or encryption) exponent e.
- The *private key* consists of the private (or decryption) exponent d.
  - p, q, and $\lambda(n)$ must also be kept secret because they can be used to calculate d.
  - They can all be discarded after d has been computed.

For example:

Suppose $P = 53$ and $Q = 59$.
Then $n = PQ = 3127$.
Let $e = 3$.
Then, $\Phi(n) = 3016$.
For $k = 2$, value of d is 2011.
We need to calculate $\Phi(n)$ such that $\Phi(n) = (P-1)(Q-1)$.
Then $d = (k * \Phi(n) + 1)/e$ for some integer k.
Thus for $k = 2$, value of d is 2011.

21 Cloud Computing

This is a new field, so the terminology is not well defined — the ideas are what is important. We define *cloud computing* as computation done on a network of servers maintained by a cloud service provider

With our model of cloud computing, our goal is effortless elasticity. We can measure this with a few heuristics:

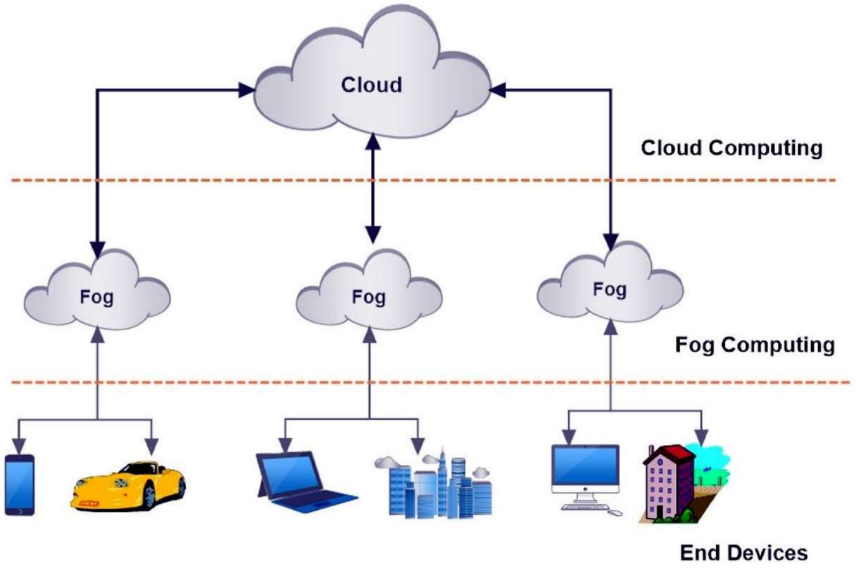
- scalability ≔ (ability to change size)
- availability ≔ (ability to run even with failure)
- fault tolerance ≔ (transparent availability)
- configuration ≔ (ease of setup)

- Quality of Service
- QoS Monitoring Quality

Note: developers often believe the STATE is not part of the QoS metrics but providers disagree.

Cloud computing can be used to the benefit of IOT/Edge Computing, since many home devices now connect to the internet and cannot do heavy computation.

To speed up our computing we use *Fog Computing*.



- localized nodes provide a hybrid between the device and the cloud
- the cloud functions like a cache for computation
- this forms a computation hierarchy

We want to make this *transparent* := to hide the servers from developers entirely, so we run provided functions in the servers, and developers are billed on usage, and it scales to $0.

Models of Cloud Computing

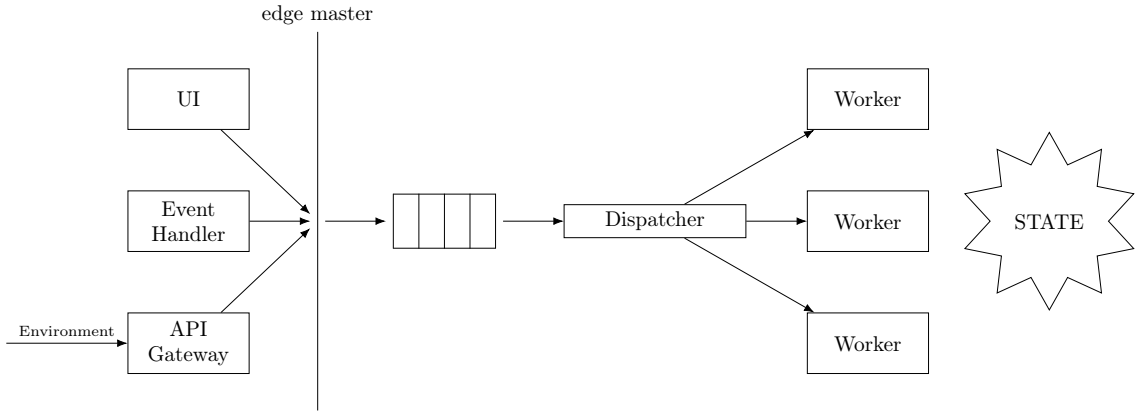
Function at a Service (FaaS)

- A function is the primary unit of computation.
- Functions are executed in response to a trigger (such as an HTTP/HTTPS request).
- Rules:
  - a function must execute relatively quickly
  - a function can have no persistent state
  - functions can, however, refer to storage or database servers to find a state)

For an example of FaaS, consider:

```
function main(params, context) {
  return { payload: 'Hello ' + params.name };
}
// params and context are JSON objects: a marshaling of a data structure
// context is a JSON object that represents metadata like security
```

If we are worried about performance, we would never use FaaS: (un)marshaling is slow. We instead use FaaS for processing speed or functionality.



The steps of event execution are:

1. arrival
2. validation
   (a) authentication
   (b) authorization
   (c) resource limit checking
3. enqueue
4. dispatch
5. allocate a container (a cheap VM)
6. copy function code into the container *
7. execute the function
8. deallocate the container

* our bottleneck is here, since Linux booting is slow relative to all the other operations.

This is called the *cold start problem*.

- This causes our latency to skyrocket.
- We can temper the throughput by adding more workers, but how do we temper latency?
  - pre-load stem cell containers with modules we expect to use
  - do not clear warm containers post use
- We can get to an almost reasonable speed with these techniques!

We call this model the *Action and Trigger Model*

- An event triggers the execution of an action.
- One event can trigger multiple actions. We can use this to implement parallel execution.
- One action can triggers an action. We can use this to implement serialized execution
- this has problems:
  - debugging
    - GDB is too big to use! We have to use their logs!
  - fixing bottlenecks
    - We can't use ps! We have to check usage in the logs post-run!

- atomicity(?)
  - We are sometimes guaranteed atomicity of function calls
- change
  - We have difficulty refactoring (breaking functions) and reverting to old versions.
  - The tools are evolving too quickly for us to get used to them!

We have a few other basic models of cloud computing which provide different levels of services

Infrastructure as a Service (IaaS)

- The provider provides supples virtual machines
- A stripped OS called a hypervisor hosts each VM; common ones include Zen and Virtual Box
- The programmer is responsible for the OS, configuration, and applications
- This is 2/3 of enterprise level IT spending. Security concerns keep it from reaching %100.
- There are some major issues, though:
  - We have to anticipate resource usage.
  - It is hard to adjust usage on the fly.

Platform as a Service (PaaS)

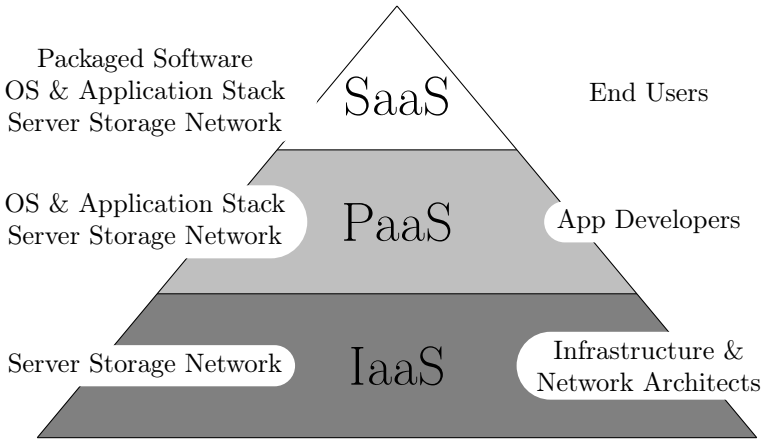
- The provider supplies some of the software stack in addition to VMs
- Less expertise is needed, but provisioning is still pushed on the developer
- Developers still have to autoscale.

Backend as a Service (BaaS)

- the provider supplies a framework for an (often mobile) app on the backend and client
- avoids heavy computation on mobile devices
- scales, but allows little flexibility

Software as a Service (SaaS)

- provider supplies the entire platform and stack
- the developer calls the service's API (which is application specific)
- can be tailored with callbacks in config files



Our models form a pyramid of increasing ease but decreasing flexibility.