CS111: Operating Systems

Professor Xu

Thilan Tran

Winter 2020

Contents

CS111: Operating Systems

Introduction to OS

- **Von Neumann** model of computing:
  - when a program is run, the processor repeatedly *fetches* an instruction from memory, *decodes* it, and *executes* it
- OS Principles
- *complexity* management principles:
  - layered structure and hierarchical decomposition
  - modularity and functional encapsulation
  - appropriately abstracted interfaces and information hiding
  - powerful abstractions
  - interface contracts
  - progressive refinement
- *architectural* paradigms:
  - mechanism / policy separation
  - indirection, federation, and deferred binding
  - dynamic equilibrium
  - criticality of data structures

Responsibilities

- the **operating system (OS)** is in charge of making the system operates correctly and efficiently in an *abstracted*, easy-to-use manner:
  - acts as the software layer between hardware and higher level applications, by abstracting and hiding low level details eg. hardware and ISAs
  - uses technique of **virtualization** to transform a *physical* resource into a generalized *virtual* form:
    - OS thus also known as **virtual machine**
    - eg. in order to virtualize memory, each running program seems to have its own private memory, instead of sharing the actual physical memory
  - provides services through interfaces and **system calls** in a **standard library** that users can use
  - acts as a **resource manager** to manage resources such as the CPU, memory, and disk

        - eg. abstracts physical memory on *disks* as *files*
    - virtualizes the CPU, ie. turning a small number of CPUs into infinite CPUs that can run many programs at once
        - this **concurrency** can lead to different problems for the OS and **multi-threaded** programs that require certain mechanisms to solve
    - handles data **persistence** with the file system and I/O
    - deals with **drivers** and coordination with external devices
- basic OS goals include **abstraction**, minimizing **overhead** (eg. in time or space), providing **protection** and **isolation** between applications, and high **reliability**
- original role has changed over time from harnessing hardware, shielding applications from hardware, to providing an ABI platform, to acting as a "traffic cop":
    - over time, different OSs have converged, since they are so difficult to *maintain*
    - applications have to *choose* to support an OS
    - new OSs must have some clear advantages over alternatives
- **instruction set architectures (ISAs)** are a computer's lowest-level supported instructions/primitives:
    - many different, incompatible ISAs
        - thus OS also responsible for running on *different* ISAs and abstracting them
        - accomplished by separating general frameworks and policies from specialized hardware mechanisms
- the OS code is *unique* from application code:
    - only OS/kernel can work with the *priveleged* ISA set, but standard ISA set is accessible by users
    - eg. applications should not be able to read from anywhere on disk
    - thus, OS should distinguish between:
        - **system calls** that require formal hardware instructions to use the OS (ie. jumping into the priveged *kernel* mode and raising hardware privelege level)
        - **procedure calls** that are provided as a library and are accessible in the *user* mode

Abstractions

- OS handles *abstracting* hardware resources for users
- **resources** have different types:
    - **serial**: used by multiple clients, one at a time, eg. printer or single CPU
        - *serial* multiplexing

        - need *graceful* transitions when switching between clients, mechanisms for exclusive use, cleanup of incomplete operations, etc.
    - **partitionable**: divided into disjoint pieces for multiple clients, eg. memory
        - *spatial* multiplexing
        - need access control for containment and privacy, transitions
    - **shareable**: used by multiple concurrent clients, eg. OS being shared from the perspective of multiple processes
        - no need for transitions, no unique state maintained for a particular client
- the OS should handle *abstractions* in order to:
    - *encapsulate* implementation details
    - provide more *convenient* and powerful behavior
- core OS abstractions include:
    - processor, memory, and communications abstractions
- **process** abstraction:
    - the source code of a program specifies a program's behavior
    - when running as a process, the stack, heap, and register contents form its environment
        - must be independent from other processes
    - but the CPU must be shared across many processes:
        - CPU **schedulers** share the CPU among those processes
        - **memory management** hardware and software gives the illusions of full exclusive memory use for each process
        - **access control** mechanisms to keep processes independent
- **memory** (specifically, filesystem) abstraction:
    - at a low level, there are many different related data storage types and resources:
        - variables, chunks, files, database records, messages
        - all with unique, peculiar characteristics
    - OS must abstract these physical devices to create ones with consistent, more *desirable* properties:
        - **persistence**
        - user desired **size**
        - **coherency** (reads reflect writes) and **atomicity** (full writes and reads)
        - **latency**
    - OS will thus:
        - have a *thorough* file system component
        - optimize **caching**
        - have sophisticated organizations to handle *failures*
- **communications** abstractions:
    - networks and interprocess communication mechanisms

- different from memory:
    - highly *variable* performance
    - **asynchronous**
    - complications from working with *remote* machines

Services

- a **service** is a provided functionality
  - in an OS, the client of its services are applications
- decomposed into:
  - **interface** or *specification* of the service, ie. description of pre- and post-conditions
  - **implementation** of the interface
- main types of OS services include:
  - **CPU/memory**: processes, threads, virtual addresses, lowest latency memory
  - **persistent storage**: disks, files, and file systems, higher latency memory
  - **I/O**: terminals, windows, sockets, networks, signals (interrupts), highest latency memory
  - note each service family can be associated with a memory latency class
    - eg. when CPU is waiting for a process's slow memory access, may use **context switching** to switch to a different process
- *higher* level OS services:
  - used by clients
  - cooperating parallel processes
  - security, eg. authentication and encryption
  - providing a UI
- *lower* level OS services:
  - not as visible
  - hardware handling
  - software updates, config registry
  - resource allocation and scheduling
  - network and protocols

Delivery

- the OS *delivers* these various services at different layers:
  - **subroutines** (functions), eg. `malloc` provided by `libc` library (implementation uses a system call):
    - implemented at higher layers to provide richer operations
    - simplest access to this delivery method, just call subroutines
      - at a lower level: push parameters, jump, place return values in registers
    - *pros*:
      - fastest, can be implemented to use the fewest system calls, eg.

buffered read and writes

      - can bind implementations at runtime
    - *cons*:
      - services are limited to the same virtual address space *associated* with the running program
      - limited to a language
      - less use of priveleged instructions
    - provided in **libraries**
    - *pros* of use of libraries:
      - code reuse, single copy, encapsulating complexity
      - many bind-time options: **static** (included at link time), **shared** (mapped into address space at exec time), **dynamic** (choose and map at load time)
  - **system calls** force an entry into the OS:
    - implementation uses the privileged kernel
    - *pros*:
      - can use previleged resources and operations
      - can communicate with other processes
    - *cons*:
      - very specific use cases, eg. viewing status of a page table
      - slower, the process may have to switch to a priveleged kernel mode
      - requires hardware to **trap** into the OS
  - send **messages** to software that perform services:
    - used in distributed systems, exchange messages with a server
    - *pros*:
      - server can be anywhere
      - service is highly scalable and available
    - *cons*:
      - slowest method
      - limited ability to operate on resources of process

Interfaces

- standardized **interfaces** in software are inspired by the concept of *interchangeable* parts:
  - ie. every part has specifications that allow any collection of parts to be assembled together:
    - *pros*: standards end up being extensively reviewed, platform-neutral, and clear and complete
    - *cons*: standards constrain possible implementations and consumers,

    and can be hard to evolve, leading to obsolescence
    - **proprietary** interfaces are controlled by a single organization, which puts the burden on the org to develop it
    - **open standards** are controlled by a consortium of providers, which may lead to reduced freedom and competitive advantage

- using interfaces for the components of a complex system architecture allows for modularity as well as independent designs
  - interfaces and their implementations should be defined *independently*
- an interface's specifications is a **contract** between developers and the implementation providers:
  - if this contract is broken, programs are no longer portable and solving issues becomes more complex
  - **backwards compatibility** can still be maintained with some strategies:
    - **interface polymorphism** for different versions of a method with unique signatures
    - **versioned interfaces** with micro, minor, or major releases
- an **application programming interface (API)**:
  - defines *subroutines*, what they do, and how to use them
  - ie. a source level interface, helps programmers write programs using the OS
  - includes discussion of signatures, options, return values and errors
  - eg. in a simple *"hello world"*, two system calls are made using their respective APIs:
    - `write(fd, p, num)` writes `num` bytes from the address at `p` to the file descriptor `fd`
    - `exit_group(code)` exits the prgram with exit code `code`
  - allows for software portability:
    - can recompile a program for a different ISA and link with OS-specific libraries supporting the same API
    - API compliant program will *compile* and run on any system compliant with that API
- an **application binary interface (ABI)**:
  - *binds* an API to an ISA
    - applications work *above* the ABI, while the kernel and machine level operations lie *under* or obey the ABI
  - ie. a binary interface specifying the DLLs, data formats, calling sequences, linkage conventions
    - helps install and run binaries on the OS
  - describes the *machine language* instructions and conventions for a specific ISA
    - eg. the binary representation of data types, stack-frame structure, register conventions, routine calling conventions

- usually used by the compiler, linker, loader, and OS
- eg. in the above *"hello world"*, the ABI for Linux x86-64 for making system calls consists of the assembly instruction `syscall`
    - where the register `rax` holds the system call number, and registers `rdi` - `r9` hold the 6 possible arguments
- allows for binary compatibility:
    - one binary serves all customers for a specific hardware ISA
    - ABI compliant program will run, *unmodified*, an any system compliant with that ABI

Program Creation and Linking

- general software file classes:
  - **source** files are editable text files in a programming language
  - **object modules** are relocatable sets of compiled and assembled instructions from source files
  - **libraries** are collections of object modules:
    - other source files can fetch functions from them
    - the order in which libraries are searched can matter
  - **load modules** are complete programs that can be loaded into memory and executed by the CPU
- software generation tool chain:
  - **compiler** produces lower-level assembly language code from source modules
  - **assembler** creates an object module in mostly machine language code from assembly language files:
    - handles lower-level operations including initialization, traps / interrupts, synchronization
    - however, some functions and data may not yet be present and not all memory addresses are finalized
      - ie. references and addresses are only relative to the start of the module addresses
  - **linkage editor** AKA **linker** reads a set of object modules, places them into a **virtual address space (VAS)**, *resolves* external references in the VAS, and finalizes all symbol addresses:
    - creates an executable load module
    - **resolution** searches through specified libraries to find object modules that satisfy unresolved references
    - **loading** lays out text and data segments from modules into one VAS
    - **relocation** fixes relocation entries and updates addresses
  - **program loader** is a part of the OS that creates a virtual address space, loads in instructions and data from executable, resolves references to additional **shared** libraries
    - reads segments into memory, creates a stack, initializes stack pointer
    - program can then be executed by the CPU
- **executable and linkable format (ELF)** is an object module format shared across different ISAs, including:
  - **header** with types, sizes, locations
  - **code** and **data**
  - **symbol table** for external symbols and references

        - used primarily for debugging
    - **relocation** entries

Linking Libraries

- **static** libraries (linktime binding, mapped into memory at linktime):
    - library modules are directly and *permanently* embedded into the load module
    - *cons*:
        - can lead to identical *copies* of the same library code in different programs
        - difficulty keeping static libraries updated (version is *frozen*)
- **shared** libraries (linktime binding, mapped into memory at loadtime):
    - reserve an address
    - linkage edit libraries into code segments
    - includes redirection table (AKA stub library) with addresses for routines
    - at load time, libraries are *mapped* into memory
    - *pros*:
        - only single library copy required (reduced memory consumption, cached libraries)
        - version can be specified at load time
        - easy to update the library (eg. updated size, adding new routines)
        - from client's perspective, *indistinguishable* from static libraries
    - *cons*:
        - cannot use global data storage, since other programs will use this same library copy
        - large, expensive libraries always loaded at startup
        - unlike for a static library, executable will not run on clients without the library
- **dynamic** libraries AKA DLLs (runtime binding, mapped into memory during runtime):
    - not loaded until they are actually needed
    - application:
        - asks OS to load a specific library into its virtual address space
        - receives standard *entry points* to make calls to the DLL through
        - maintains a *table* of entry points for different DLLs
    - when DLL is no longer needed, application asks OS to unload module
    - loading DLLs is done through an API, but the actual loading mechanism is ABI-specific
    - *pros*:
        - runtime binding gives more flexibility for program

    - libaries can be unloaded when no longer required
  - *cons*:
    - more work for the client to load and manage DLLs

Process Virtualization

- the process of **virtualization** takes a *physical*, *limited* resource and creates the illusion of having *virtual*, *unlimited* copies of that resource

- the most fundamental abstraction provided by the OS to users is the **process**, or running instance of a **program**:

  - a program is:
    - *static*, an abstraction stored on disk as a load module with resolved references
    - contains headers, code and data segments, symbol table for the linker
    - but all addresses are unloaded and relative
  - a process has these different **segments** loaded into its address space:
    - statically-sized **code segment** contains code read in from load module:
      - read-only, executable-only, thus different processes can share the same code segment by mapping the addresses
    - **data segment** containing heap, handles *initialized* global data as well as *dynamic* data:
      - read-write, process private
      - can grow and shrink during process, grows upward
    - **stack segment** handles procedure call stack frames (eg. local variables, invocation parameters, saved registers):
      - grows downward
    - **stack overflow** occurs when stack and data segment meet, protects process from possible data corruption
  - in order to run many programs at once, the OS must *virtualize* the CPU:
    - OS uses a **time sharing** approach to virtualizing the CPU, as opposed to a **space sharing** approach (eg. for files)
    - there are low-level **mechanisms** that help achieve this virtualization, eg. **context switching** that allows OS to switch between running programs on a CPU
    - in addition, there are higher-level **policies** or decision algorithms used by the OS to choose which programs to run at a given time (*scheduling* policies)

- a process has an associated **machine state** or properties that it can read or write to at any given time

- this **state** should consistently, uniquely, characterize the process, and includes:

  - **memory** to store its instructions and data
    - every process has an address space of *virtual memory addresses* reserved for itself (illusion of infinite memory)
  - **registers** that are used during execution
    - special registers include the **program counter** that indicates the next instruction, **stack pointer**, and **frame pointer**
  - **I/O information**
    - eg. open persistent storage devices, condition of an I/O operation
  - other OS-related metadata information

Subroutine Stack Frames

- *calling* a subroutine:
  - placing parameters into registers
  - saving the return address on the stack, and transferring control to the entry point
  - saving certain volatile / callee-saved registers so that they can be restored
  - allocate local variables on the stack
- *returning* from a subroutine (symmetric steps):
  - place return value into register
  - pop local storage off stack
  - restoring registers
  - transfer control
- handling traps and interrupts:
  - **interrupts** inform software of an external event
  - **traps** are hardware instructions that inform software of an execution fault (type of interrupt)
  - similar to a procedure call: have to transfer control, save state, restore state and resume process
  - different from procedure call because these events are *hardware*-initiated, so linkage conventions are defined by the hardware
    - after event, computer state should be restored as if event never happend
  - very expensive event to handle, since the CPU is moved to a priveleged mode and new address space
- trap and interrupt *mechanism*:
  - a table associates a **program counter and processor status (PC/PS)** word pair with each possible interrupt/trap
  - when an event triggers an interrupt or trap:
    - CPU uses exception number to index into table and load a new

        PC/PS onto the CPU stack
    - exception continues at new PC address
        - **first level handler** saves registers, polls hardware for cause of exception, chooses and calls a specific **second level handler**
    - on second handler termination, first level handler restores registers, reloads PC/PS, resumes execution

Process Overview

- conceptual process *API*:
    - **create**: OS must provide method to create new processes
        - may create a *blank* process with no initial state (Windows approach)
        - or use the *calling* process as a template (UNIX approach)
            - this approach is useful when making processes with context from parent
        - leads to *parent-child* process relationship
    - **destroy**: OS must provide method to destroy or kill processes, eg. with signals
        - must clean up a terminating process:
            - reclaim resources
            - inform interprocess processes with signals
            - remove process descriptor
    - **wait**: wait for a process to stop running
    - **misc. control**: eg. suspending and resuming processes
    - **status**: retrieve status info for a process
- process *creation* consists of:
    - creating a new address space
    - *loading* and *mapping* code and data into memory/address space of the program
        - programs usually reside on disk, so the OS reads bytes from disk and places them in memory
        - modern OSs use **lazy loading** to load data only when it is needed
    - allocating and initializing the stack for the program (eg. with parameters, `argv` , `argc`)
    - allocating the heap for dynamic memory
    - initializing registers (PC, PS, SP)
    - initializing I/O (eg. opening file descriptors)
    - run program from its **entry point** (eg. `main`)
- in addition, processes may be loaded and *resumed* from a previous blocked state

    - in this case, registers must be loaded from the saved state
- **states** of a processes:
    - **running**: CPU is executing instructions for a process
    - **ready**: process is ready to run, but not currently executing
        - when a process is *scheduled*, it moves from ready state to running
        - when a process is *descheduled*, it moves from running state to ready
    - **blocked**: process has performed some operation that makes it unable to run until some other event takes place (eg. I/O request to disk)
        - the OS recognizes when a running process becomes blocked, and will run a different process to maximize time sharing
    - **initial**, **final/zombie** (not yet cleaned up, allows parent process to check return code)
- the OS maintains key *data structures* called **process descriptors** to track the state of processes. Associated structures include:
    - **process/task table** with descriptors for all ready or running processes, another list for blocked processes
    - **process control block (PCB)** is a C structure maintained in Linux storing information for each process:
        - used for saving the state of process and the registers of a process for **context switching**
        - eg. start and size of memory, process state (eg. scheduled, blocked) and ID, open files, CWD, context, parent, registers
    - certain state of processes is additionally stored on a *per-process* **kernel stack**:
        - retains the stack frames for in-progress OS system calls, and the state of interrupted processes so that the OS can return back to the process
        - must be separate from user stack for *security*, since the kernel stack is used for priveleged operations
        - must be per-process since different processes will experience *different* interrupts and system calls
        - saves essential registers required for switching in and out of the kernel after interrupts, eg. PC, PS, SP, as well as user registers

UNIX Process API

Using `fork`:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

```

```c
int main(int argc, char *argv[]) {
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr,"fork failed\n")
    exit(1);
  } else if (rc == 0) {
    /* child (new process) */
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    /* parent goes down this path (main) */
    printf("hello, I am parent of %d (pid:%d)\n",
           rc, (int) getpid());
  }
  return 0;
}
```

- `fork` creates an almost *exact* copy of the calling process:
  - to OS, there are two programs running, both about to return from `fork`
  - thus, new **child** process starts running after call to `fork` , instead of from start of `main` :
    - child has a new address space
    - child *shares* parent's code segment
    - a new stack is *initialized* to match the parent's
    - data initially points to the parent's *original* data:
      - but when the child modifies the data segment, we need to set up a seprate data segment copy for the child
      - copying large data segment can be expensive, so OS uses **copy-on-write** (lazy operation) to only copy the data once one of the processes has written to it
      - copy-on-write occurs on a low granularity, eg. only copying and remapping specific page that is changed
  - `fork` is non-deterministic, either child or parent will print first depending on the CPU **scheduler**
- new child has a copy of the address space, but the return of `fork` differs:
  - child receives return code of 0
  - parent receives new PID of child

Using `wait` :

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```c
#include <sys/wait.h>

int main(int argc, char *argv[]){
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr,"fork failed\n")
    exit(1);
  } else if (rc == 0) {
    printf("hello, I am child (pid:%d)\n", (int) getpid());
  } else {
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

- `wait` waits for a child process to finish executing:
  - returns PID of finished child process
  - this makes code snippet deterministic, child will always print before parent in this case

Using `exec` :

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

int main(int argc, char *argv[]){
  printf("hello world (pid:%d)\n", (int) getpid());
  int rc = fork();
  if (rc < 0) {
    fprintf(stderr,"fork failed\n")
    exit(1);
  } else if (rc == 0) {
    printf("hello, I am child (pid:%d)\n", (int) getpid());
    char *myargs[3];
    myargs[0] = strdup("wc");
    myargs[1] = strdup("p3.c");
    myargs[2] = NULL;
    execvp(myargs[0], myargs); /* counts words in p3.c */
    printf("this should not print");
  } else {
    int rc_wait = wait(NULL);
    printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
           rc, rc_wait, (int) getpid());
  }
  return 0;
}
```

- `exec` allows us to fork a child of a *different* program:
  - does not *create* a new process, *transforms* currently running program (here, a forked child) into another running program
    - OS loads new code and overwrites current code segment, reinitializes heap and stack, runs new program
  - thus, a successful call to `exec` *never returns*
- separation of `fork` and `exec` have several advantages:
  - allows shell code to be run *after* `fork` and *before* `exec` and thus alter the environment of about-to-run program
  - allows for easy redirection
    - eg. in order to redirect output to some file, after `fork` , close `STDOUT` , open the file, and then `exec`
  - piping is implemented more easily (with `pipe` system call)
- signals and processes:
  - `kill` system call can send signals to a process or **process group**, eg. `SIGINT` for interrupt, `SIGSTP` for stop
  - processes can then use `signal` system call to catch signals

Inter-Process Communication

- OS supports IPC through different system calls:
  - **synchronous** IPC waits until messages are delivered or available
  - **asynchronous** IPC returns immediately, regardless of success
    - requires auxilliary mechanisms to poll for new messages and check for errors on message sends
- data can be shared with **streams**, or **messages** AKA **datagrams**:
  - streams are read and written many bytes at a time
    - the application specifies the format or delimiters in the stream
  - datagrams are a sequence of *distinct* messages, each delivered and read as a unit
    - the IPC mechanism must know about the format
- OS must also take care of **flow control**, to ensure a fast sender doesn't overwhelm a slow receiver:
  - queued or buffered messages are expensive to maintain
  - should be able to limit the buffer space of messages to prevent hogging of OS resources
    - sender may block sender or refuse messages
    - receiver may stifle sender or flush old messages
- OS must be responsible for *reliability* of IPC messages:
  - must retain IPC data for a certain time
    - may allow channels and content to persist through restarts
  - may attempt retransmissions across alternate routes on message send error
- data can be exchanged between processes *uni-directionally* or *bi-directionally*
- uni-directional byte streams are processing pipelines, where processes read from stdin and write to stdout
  - each program accepts a byte-stream input and produces a well defined byte-stream output
  - each program operates independently
- **pipes** are temporary bi-directional buffers (similar to files) from `pipe` that are different from static files in the following ways:
  - simple byte stream buffered between programs
    - no security or privacy controls
  - reader does not get `EOF` until the write side of the pipe is closed
  - `SIGPIPE` from writing to a pipe without an open read end
  - file automatically closed when both ends are closed
- **named-pipes (FIFOs)** are *persistent* pipes that can connect unrelated processes:
  - not destroyed when I/O is finished

  - writes from multiple writers can be interspersed
  - no clean fail-overs on failed reads
- **mailboxes** are not a byte-stream, but distinct, delivered messages:
  - every write has id from sender
  - unprocessed messages saved on death of reader
- **general network connections** provide network communications through **sockets**:
  - connected between addresses and ports
    - operations include connect, listen, accept
  - can use either byte streams (TCP) or datagrams (UDP)
  - much more complexity due to online network
  - complex flow control and error handling
  - has issues of security and privacy
  - high latencies, limited throughput
- **out-of-band** signals are signals that should supersede queued or buffered data:
  - locally, could set up a handler that flushes all buffered data upon receiving an out-of-band signal on a different channel
  - with network services, open multiple communication channels
    - server must periodically poll the out-of-band channel
- **shared memory** is the highest performance communication method:
  - processes must run on the same memory bus on local machine
  - race conditions, sychronization issues
  - no authentication or security from OS
    - applications must provide their own sharing control
  - much faster than other models

Signals

- OS allows for processes to attach *event* callbacks:
  - functions analagously to traps and interrupts, implemented and delivered to process by OS
  - eg. I/O devies and timers
- these *events* are defined by OS through many types of **signals**:
  - processes can then choose to *ignore*, *handle*, or perform a *default* action on certain signals

Process Mechanisms

Direct Execution

- the **scheduler** is the component that actually determines which processes to run
- challenges associated with virtualizing, ie. **time sharing** the CPU:
  - maximizing performance with minimal *overhead*
    - eg. minimal entering the OS, minimal use of the priveleged instruction set (no OS intervention)
  - handling processes while retaining *control*

An initial **direct execution protocol** without limits for maximum efficiency:

OS	Program
create entry for process list	
allocate memory for program	
load program into memory	
set up stack with `argc, argv`	
clear registers	
execute `call` to `main`	
	run `main`
	execute `return` from `main`
free memory of process	
remove from process list	

- this initial approach has some problems:
  - how do we ensure CPU doesn't do anything undesired or restricted, without reducing efficiency?
    - occasional **traps** for syscalls
  - how do we efficiently switch processes in order to actually *virtualize* the CPU?
    - occasional **timer interrupts** for time sharing

Restricted Operations

- some operations should be **restricted** to the OS, eg. I/O or accessing more system resources:

  - eg. if any process could do I/O, all data protections would be lost
  - the solution is to introduce processing modes, a restricted *user mode* and an unrestricted *kernel mode* with elevated priveleges
    - kernel mode also has full access to hardware resources
  - OS provides an ABI to access these operations with **system calls** and **traps**
- some **exceptions** are routine that can be checked in programs:
  - end of file, arithmetic overflow, conversion errors
- however, sometimes will occur that aren't handled by the user:
  - usually **asynchronous** exceptions such as segfaults, user abort, power failure
  - OS must handle these unhandled exceptions with a trap into the OS to perform restricted operations
- when a *user* program wants to perform a priveleged operation, it can use a system call:
  - system calls allow the kernel to expose important functionalities
  - a **system call number** is associated with each syscall, this indirection is a form of protection
  - user arguments/input are placed in ABI-specified registers, and must be validated by OS before performing the actual syscall in kernel mode
- to actually execute a system call:
  - process initiates a syscall with the ABI's specifications
  - this causes a trap or exception that jumps into the kernel
  - hardware:
    - raises the privelege level to kernel/supervisor mode so priveleged operations can be performed
    - uses exception number to index into a **trap vector table** to get the **program counter and processor status (PC/PS)** associated with the first handler of the exception
      - PS usually holds the return/error code from the syscall
    - push PC/PS of the process triggering the trap to kernel stack
    - load the PC/PS onto the kernel stack of the first level trap handler
  - software continues at new PC address:
    - **first level handler**:
      - saves registers
      - polls hardware for cause of exception
      - chooses and calls a **second level handler** from a **dispatch table**
    - **second level handler**:
      - specifies the **trap gate**
      - actually handles the trap/syscall
  - on second handler termination:
    - first level handler restores registers, reloads PC/PS, allows for resuming execution at next instruction

    - when finished, the OS calls a **return-from-trap** instruction that returns to user mode and reduces the privelege level
- a per-process **kernel stack**:
    - acts as a *call stack* for trap handlers and other priveleged operation routines
        - separate from user stack for *security* and *isolation*
    - allows execution of the user process to be *resumed*
    - must push PC/PS, flags, user registers, system call parameters onto kernel stack when entering OS
    - grows and shrinks alongside the syscall handler stack frames
- the kernel should carefully control which code executes on a trap:
    - OS sets up a **trap table** at boot time that informs the hardware of the locations of **trap handlers** to call on a trap
    - note that the machine boots initially in kernel mode

An updated **limited execution protocol** to deal with system calls and traps:

OS at boot	**Hardware**	**Program**
initialize trap table	remember addresses of trap / syscall handlers	

OS at run (kernel mode)	**Hardware**	**Program** (user mode)
create entry in process list		
allocate memory for program		
setup user stack with args		
fill kernel stack with reg / PC		
`return-from-trap`		
	restore regs from kernel stack	
	move to user mode	
	jump to `main`	
		run `main`
		call syscall
		trap into OS
	save regs to kernel stack	
	move to kernel mode	
	jump to trap handler	
handle trap / syscall		
`return-from-trap`		
	restore regs from kernel stack	
	move to user mode	
	jump to PC after trap	
		return from main
		trap (via `exit`)

OS at run (kernel mode)	**Hardware**	**Program** (user mode)
free memory of process remove from process list		

Process Switching

- when a program is running on a CPU, the OS is *not* running:
  - how can the OS *regain control* of the CPU so that it can switch between processes?
  - in a **cooperative** scheduling system, the OS simply *waits* for a program to make a syscall or `yield` in order to regain control
    - this can lead to bugs with infinite loops or malicious programs
  - in a **non-cooperative** scheduling system, a **timer interrupt** is used:
    - every so many milliseconds, an interrupt is raised automatically, and an **interrupt handler** in the OS runs
    - this timer must be started on boot up
    - the hardware must save the state of the program so that execution can resume on a `return-from-trap`
- once OS has control, the **scheduler** decides whether to continue running the current process (process A), or switch to a different one (process B) with a **context switch**:
  - in a context switch:
    - the *hardware* saves the *user* registers for A into kernel stack A so A can resume execution after interrupt
    - the *OS* saves the *kernel* registers for A into memory in the process structure of A so A can resume execution after context switch
    - the *OS* restores the *kernel* registers for B from memory in the process structure of B
    - the *OS* switches from kernel stack A to kernel stack B by changing the stack pointer
    - the *hardware* restores the *user* registers for B from kernel stack B
  - then, after `return-from-trap` , the system has resumed excution of process B
  - context switching is *expensive*: have to enter the OS, switch OS context (stacks, address spaces), loss of caches
- to deal with the issue of **concurrency**, the OS may disable interrupts for a period of time, or use locking schemes

An updated **limited execution protocol** to deal with context switching:

OS at boot	**Hardware**	**Program**
initialize trap table	remember addresses of trap/syscall handlers	
start interrupt timer	start timer	
	interrupt CPU in X ms	

OS at run (kernel mode)	**Hardware**	**Program** (user mode)
		Process A
	timer interrupt	
	save regs(A) → k-stack(A)	
	move to kernel mode	
	jump to trap handler	
handle trap		
call `switch`		
save regs(A) into proc. struct A		
restore regs(B) from proc. struct B		
switch to k-stack(B)		
`return-from-trap`		
	restore regs(B) ← k-stack(b)	
	move to user mode	
	jump to PC of Process B	
		Process B

Process Scheduling

- because CPU is limited as a resource, the OS has to use a **scheduler** in order to schedule processes such that they have the illusion of having full access to the CPU's resources:
  - scheduler has to *choose* which ready processes to run when:
    - current process yields or traps to the OS
    - timer interrupt occurs
    - current process becomes blocked (eg. I/O)
  - *metrics* for performance are the turnaround time, throughput, wait time, response time, degree of fairness, achieving explicit priority goals
  - different scheduling *goals*:
    - **time sharing**: fast response time for interactive programs, every user gets equal CPU share
    - **batch**: maximize throughput, individual delays are unimportant
    - **real-time**: critical operations must happen on time, non-critical operations may be deferred
- *ideal* throughput is impossible:
  - some overhead per dispatch
  - in general, want to reduce the overhead per dispatch (mechanism), as well as the number of dispatches (policy) so that performance *approaches* the ideal throughput
- response time *explodes* at a certain load:
  - finite queue sizes, requests or parts of requests may be *dropped* (infinite response time)
  - dealing with *overloaded* systems:
    - continue service with *degraded* performance
    - alternatively, maintain performance by rejecting work and resume normal service once load drops
    - *avoid* throughput dropping to zero or infinite response time
- in addition to lower level mechanisms associated with the process abstraction, OS also deals with high-level scheduling **policies** for processes:
  - the OS policies should be implemented and chosen *independently* than the mechanisms (eg. dispatching and context switching)
  - scheduler can either be **preemptive** (interruptive) or **non-preemptive**
    - *non-preemptive pros*: low overhead, high throughput, simple (fewer context switches)
    - *non-preemptive cons*: poor response time, freeze at infinite loop bugs, not fair
    - *preemptive pros*: good response time, fair
    - *preemptive cons*: more complex, requires context switch mecha-

        nism, not as good throughputs, higher overhead, can be bad for real-time systems
    - we will explore different scheduling algorithm approaches and how they fail when certain *assumptions* are *relaxed*:
        1. every job runs for the same amount of time
        2. all jobs arrive at the same time
        3. each job runs to completion once started
        4. all jobs only use the CPU
        5. run-time of each job is known
    - initially, aim for optimizing turnaround time (similar to throughput)
- **first in, first out (FIFO)** scheduling:
    - schedules jobs in the order that they arrive
    - highly variable delays
    - useful when response time is not important (*batch* programming), or *embedded* systems (brief processes, simple implementation)
    - effective until assumption 1 is relaxed and jobs no longer run for the same amount of time
        - issues with the **convoy effect**, where many low potential consumers may become queued *behind* a heavyweight resource consumer
- **shortest job first (SJF)** scheduling:
    - runs the shortest job first, next shortest, etc.
    - optimal until assumption 2 is relaxed and jobs no longer arrive at the same time
        - shorter job could arrive while a job is still running
        - ie. SJF is a non-preemptive scheduler that cannot interrupt jobs
- **shortest time-to-completion first (STCF)** scheduling:
    - also relax assumption 3 that jobs will run to completion
    - allow scheduler to use context switching and *preempt* jobs to run another job
    - AKA **preemptive shortest job first (PSJF)**
    - effective until we consider a new metric **response time**, the time for a job to be scheduled for the first time
        - response time deals with interactivity for users
- **round robin (RR)** scheduling:
    - rather than running jobs to completion, run a job for a **time slice** before switching to the next job in the queue
        - more fair CPU sharing and delays, good for interactive processes
    - length of time slice thus must be a multiple of the timer-interrupt period
    - tradeoff between smaller, faster time-slices and the overhead of more context switching, ie. need to find good *amortization*
    - *however*, one of the worst policies in terms of turnaroud time and number of context switches

- when considering other systems, exploit the *overlap* of operations:
  - relaxing assumption 4 that jobs only use the CPU
  - eg. when a process becomes blocked waiting for the completion of an I/O request, schedule another process
  - involves treating each CPU *burst* as an individual job
  - *however*, for SJF and STCF, the run-time of each job is still known

Feedback Priority Scheduling

- **multi-level feedback queue (MLFQ)** scheduling:
  - aims to optimize turnarund time *as well as* response time when assumption 5 is relaxed and the run-time of jobs aren't known
  - an example of a *smart* system that uses the *past* to predict the *future*
- has a number of distinct **priority queues** with different priority levels:
  1. If the priority of A > priority of B, only A runs
  2. If the priority of A = priority of B, A and B run in RR
  - queues may have different time slices depending on priority:
    - shorter time slices for *foreground* high priority tasks, and long time slices for *background* low priority tasks
  - can also have a queue dedicated to real-time tasks that run until completion
    - FIFO for low priority or real-time queue
- the scheduler will *vary* the priorities of a job based on its *observed behavior*:
  - when a job repeatedly makes I/O requests to the keyboard, it will have its high priority *maintained* (interactive process)
  - when a job uses the CPU intensively for long periods of time, it will have its priority *reduced* (response time isn't important)
- translated into rules:
  3. When a job enters the system, it is placed at the *highest* priority
  4. If a job uses an entire time slice, its priority is *reduced*
  5. If a job gives up CPU before time slice is up, it stays at the *same* priority
- when a new job comes along, the scheduler *assumes* it may be a short job with a high priority:
  - if it is short, the job will run quickly and complete
  - otherwise, the job will move down the queues and run in a batch-like process
  - can also profile processes to estimate which queue to place them into in the future
- issues with this initial implementation:
  - with too many interactive jobs, they will consume *all* CPU time and long running jobs will be **starved**

    - a program could *maliciously* issue an I/O operation right before the end of its time slice to *always* run at a high priority
    - no mechanism for a CPU-bound job to transition to interactivity
- using **accounting**:
    6. Once a job uses up its time allotment at a given level, its priority is *reduced*
- using a **priority boost**:
    - in order to guarantee CPU-bound jobs will make progress against *starvation*, boost *all* jobs periodically
    7. After some time period S, move all jobs to the topmost queue
        - S is a **voodoo constant**, if too high, starvation occurs, if too low, interactive jobs would not get a proper share of the CPU
- involves many **parameters** for time slice length, number of queues, etc.
    - many implementations provide configuration files that can adjust these parameters
    - users can also give **advice** to the OS to modify scheduler behavior

Realtime Systems

- priority scheduling is a *best effort* approach:
    - there are other systems whose correctness depend on certain *timing* requirements as well as *functionality*
    - eg. space shuttle during reentry, reading sensor data at high speeds, playing media
- realtime systems are characterized by different metrics:
    - **timeliness**: how closely timing requirements are met
    - **predictability**: deviation in timeliness
- new realtime concepts:
    - **feasibility**: whether or not requirements for a task set can be met
    - **hard real-time**: strong requirements that tasks be run at specific intervals, not recoverable on failures
        - dynamic behavior, unbounded loops should be avoided
        - may have to dissable interrupts and preemptive scheduling
    - **soft real-time**: good response time required, but recoverable on failures
- realtime characteristics that make scheduling *easier*:
    - task length will be known
    - starvation of low priority tasks is acceptable
    - work-load may be fixed
- *differences* between ordinary time-sharing:
    - preemption is no longer an optimal strategy:
        - preempting running tasks will cause it to miss its deadline

- execution time is known, so there is little need for preemption
- real-time systems run fewer and simpler tasks, so code is not malicious or buggy (no infinite loops)

Memory Virtualization

- **physical addresses** are *abstracted* to programs as **virtual addresses**
  - allows for *ease of use* for programmers, and isolation and protection
- an **address space** is a *virtual* abstraction of physical memory:
  - a virtual address is independent from physical address
  - contains all of the memory state of a running program (code, stack, and heap segments)
  - by convention, stack and heap at opposite ends, growing in opposite directions
  - the program is not necessarily in *contiguous* physical memory like the address space, but loaded at *arbitrary* physical addresses
    - eg. printing pointers in C will print virtual, not physical, addresses
  - every process can have an address space of immense size
    - supported using **dynamic paging** and **swapping**
- memory virtualization goals include:
  - *transparency*, ie. an invisible implementation by the OS
  - *efficient* virtualization through hardware support
  - *protecting* processes from one another through isolation

UNIX Memory API

- **heap** memory, as opposed to **stack** ie. *automatic* memory, is explicitly handled by the programmer
- `malloc` dynamically allocates space on the heap:
  - is a library call that uses system calls such as `brk` or `sbrk`
  - `sizeof` is a thus a *compile-time* operator
  - `free` frees heap memory
- `calloc` , `realloc`

```
#include <stdlib.h>

double *d = (double *) malloc(sizeof(double));
free(d);
```

- common errors:
  - failure to allocate memory (eg. `strcpy` into a unallocated pointer) often leads to a **segmentation fault**
  - **buffer overflow**, or not allocating enough memory can have nondeterministic behavior
  - **uninitialized read**, or not initializing allocated memory

- **memory leak**, when memory is not properly freed
- **dangling pointer**, when freed memory is accessed
- **double free**, when allocated memory is freed multiple times
- **invalid free**, when memory not allocated by `malloc` is freed

Memory Mechanisms

General Partition Strategies

- the following are different techiques for partitioning *physical* memory
  - not yet considering virtualization of memory
- **fixed partition**:
  - preallocate partitions for a certain number of processes
  - useful when exact memory needs are known
  - partition sizes are fixed
  - using only *contiguous* physical addresses
  - *pros*:
    - simple implementation
    - allocation/deallocation cheap and easy
  - *cons*:
    - inflexible, limits number of processes and their memory usage
    - can't share memory
    - **internal fragmentation**, or wasted space inside *fixed* blocks
- **dynamic partition**:
  - similar to fixed, except *variable* sized blocks
  - process tells OS how much memory it needs
  - each partition is still contiguous
  - still using physical addresses
  - *pros*:
    - sharable partitions
    - process can use multiple partitions, with different sizes
  - *cons*:
    - still not *expandable*, may not be space nearby
    - still not *relocatable*, pointers will be incorrect
      - partitions are *tied* to address range
    - still subject to some fragmentation
    - still not as large as desired virtual address space
- can use a **free list** to track unallocated memory:
  - allows processes to use *incontiguous*, variable sized partitions
  - each element in the list has a metadata **descriptor** with data such as length, whether it is free, and a pointer to the next chunk
  - to *carve* a chunk:
    - reduce the length, create a new header for leftover chunk, connect new chunk to list, and mark chunk as used

- eliminates internal fragmentation to a degree, since process can use smaller sized chunks as needed
- however, over time, leads to the inverse issue of **external segmentation** ie. useless, small left-over chunks between carves
- different free space management strategies help counteract external segmentation:
  - different allocation algorithms, eg. first-fit, next-fit
  - **coalescing** adjacent free memory chunks together
  - is it possible to relocate free memory and thus further **compact** incontiguous parts together?
    - compaction requires relocation
    - relocation requires **address translations** and **virtual memory**

Address Translation

- **hardware-based address translation** is a generic, hardware technique that extends the **limited direct execution** model:
  - hardware performs an address translation on every memory reference, ie. *interposes* each memory access
    - the **memory management unit (MMU)** is the CPU component dealing with memory virtualization
  - OS takes care of **managing memory**
  - address translation allows for easier relocating of memory
    - without virtual memory, whenever memory is moved, would have to update all of a process's pointers and memory references
- **dynamic relocation** or *base-and-bounds* technique (one hardware-based translation technique):
  - uses a **base** and **bounds** register in the CPU (different per process)
  - when a program starts running, the OS decides where in physical memory to load it and sets the base register to that value
  - every memory reference (virtual address) from the process gets *translated* by the CPU by adding the base register to produce a physical address
    - occurs at runtime, *interposed*, with little hardware logic required
  - if a virtual address is greater than the bounds register, an exception will be raised
    - *limits* and *protects* address spaces
  - *hardware*:
    - provides extra registers in the MMU and priveleged instructions to modify these registers
    - provides mechanisms for raising exceptions and registering han-

dlers

- *OS*:
    - allocate memory for new processes using the free list
    - cleans up after a process ends by updating the free list and deallocating memory
    - *save-and-restore* base-bounds pair registers using the PCB when context switching
    - install exception handlers at boot time (along with other handlers eg. syscall handlers)
- acts as an *extension* of LDE, where address translating is interposed during direct execution, and OS only intervenes when process misebhaves
- allows for address space to be *relocated* when a process is stopped by copying between locations and updating the saved base register
- **software relocation** is an alternative where the loader rewrites all instructions by adjusting the addresses
    - provides no protection, and difficult to relocate address spaces

- **internal fragmentation** is still an issue *within* the process using this approach:
    - the space inside the allocated unit of a process is wasted, since its stack and heap are small
        - even if the process itself may be dynamically asking for smaller pieces of memory
    - the address space of a process are thus *still* restricted in fixed-size slots
    - cannot run programs where the entire address space doesn't fit into memory
    - issue compounded with even larger, sparse virtual address spaces, eg. 32-bit, 4 GB address spaces

Segmentation

- instead of having a single base-bounds pair in the MMU, instead have a base-bounds pair for every **segment** in the address space:
    - use segments as the *unit* of relocation
    - each segment is already a *continugous* portion of address space
        - this is a **course-grained** segmentation, while **fine-grained** segmentation involves even more smaller segments, usually with a **segment table**
    - allows OS to place segments in different parts of physical memory *independently* and fill *unused fragments*:
        - more flexibility when allocating for processes with large, sparse address spaces

        - also allows specific segments, eg. code segment, to be *shared* between processes
    - now, during address translation, hardware must consider the **offset** *in the specific segment* the address or instruction belongs to, and add the offset to the base register of the segment
        - eg. to read an address in the heap at virtual address `4200`, `offset = 4200 - start(heap)` where `start(heap)` is the virtual address of start of heap compared, `phys_addr = offset + base(heap)` where `base(heap)` is the physical start of the heap
    - to read into the stack which *grows backwards*, the MMU uses a register bit for all segments to keep track of which way the segment grows
    - to allow for sharing of memory segments, the MMU uses register **protection bits**
        - eg. code is read-execute, while a heap and stack are read-write
    - considering an illegal address beyond the end of a segment leads to a **segmentation fault**
- matching an address to its segment and segment base-bounds register:
    - the **explicit** approach is to *slice* the address space into segments based on the *top few bits* of its virtual address
        - eg. if the first two bits of a virtual address is `00`, the hardware will use the code base and bounds pair and the remaining bits as the offset from the segment start
    - the **implicit** approach is to use hardware that examines how the address was formed (eg. program counter, stack pointer, otherwise)

Example address translation process (explicit, slice address into segments):

```
Segment = (VirtualAddr & SEG_MASK) >> SEG_SHIFT
Offset = VirtualAddr & OFFSET_MASK
if (Offset ≥ Bounds[Segment])
  RaiseException(SegmentationFault)
else
  PhysAddr = Base[Segment] + Offset
  Register = AccessMemory(PhysAddr)
```

- segmentation raises some issues that OS must deal with:
    - on a context switch, segment registers must be saved and restored (each process has a virtual address space)
    - OS must find space in physical memory for new address spaces:
        - every segment can now be a different size
    - still must deal with external fragmentation as segments change in size:
        - periodic **defragmentation**: OS can **compact** physical memory by rearranging segments contiguously and updating base registers

      - copying can be very expensive (especially for some types of disks)
    - OS can coalesce as much as possible when free segments are contiguous
      - frequent allocation / deallocations add opportunities to coalesce
    - another approach is to use a free-list **management algorithm** (many algorithms have been used):
      - avoid creating small fragments
      - recombine smaller fragments
  - still not segmented enough, eg. segments *themselves* may be sparse and largely empty (entire heap or stack mostly empty)?

Free-Space Management

- managing free space can be simple with fixed sized chunks:
  - more difficult with variable-sized units, eg. when using segmentation or allocation libraries:
    - supporting variable-sized blocks counteracts internal fragmentation
    - but leads to external fragmentation as the number of small, useless left-over chunks increases
  - note that the **allocator** *itself* in a allocation library cannot utilize **compaction** to combat external fragmentation
    - compactions are expensive, and only ran periodically by the OS
- allocator mechanisms:
  - uses a **free list** to reference free chunks of space on heap:
    - eg. would contain the starting address and length for each chunk in a linked list
    - difficult to **scale** the performance of a linked list, special types of trees may be a better data structure
  - on a small request, **split** a chunk into two and update the chunk's length accordingly in the free list
  - on a memory free, **coalesce** multiple, contiguous chunks together into a single new chunk in the free list
- allocators store metadata for allocated memory in a **header** block immediately before allocated chunk:
  - could store chunk size, additional pointers, magic number for **integrity checking**

  - every allocation of N bytes will require enough space for $N + K$ bytes for the header

- the free list must be *embedded* in the free space itself:
  - free list node minimally holds free chunk size, and a pointer to the next chunk
  - on an allocation, the free chunk is split in two:
    - one chunk large enough for the request and header
    - remaining free chunk with an updated size in the free list node
  - on a memory free:
    - the allocator uses the size in the chunk header to add the free chunk back into the free list,
    - adds a pointer to the next free chunk's node, and redirects the head pointer (requires coalescing and merging to clean up)

- usually, allocator starts with a smaller heap, and uses `sbrk` to ask OS to grow the heap (update bounds)

- free space allocation *strategies*:
  - want to minimze external fragmentation by avoiding smaller fragments
  - **best fit**, ie. smallest fit:
    - find the smallest possible block, waste minimal space
    - actually quickly creates small fragments
    - may involve expensive exhaustive search
  - **worst fit**:
    - find the largest possible block, leaving a large free chunk remaining
    - creates larger fragments, for longer
    - may involve expensive exhaustive search
  - **first fit**:
    - find first block that fits the request
    - fast, but pollutes free list with many small objects
      - searches get longer over time
    - could use **address-based ordering** in the free list to help with coalescing
  - **next fit**:
    - combination of worst fit and first fit
    - maintains a pointer to where allocator was last checking for free space
      - guess pointer acts as a *lazy* cache
    - spreads out searches more uniformly
    - shorter searches

- these strategies combat external fragmentation, but carving and coalescing is also expensive when allocating memory

  - can we minimize these actions?
  - **segregated lists** AKA **buffer pool** (optimize carving):
    - maintain several lists specifically dedicated to a few popular-sized, special requests
    - with a uniform size of requests, allocation is more efficient, and fragmentation is eliminated
    - need to balance how much to reserve in the pools:
      - if too little, buffer pool becomes a *bottleneck*
      - if too much, buffer pool has much unused buffer space
    - can also *dynamically* size buffer pools:
      - get more memory from free list when running low on fixed sized buffers
      - return some buffers to free list if buffer list gets too large
      - can also request services with buffer pools to return space
    - eg. the **slab allocator** uses **object caches** for kernel objects that are likely to be requested frequently (inodes, locks, etc.):
      - requests slabs of memory when the cache is running out of space
      - also keeps objects pre-initialized for even faster performance
  - **buddy allocation** (optimize coalescing):
    - on a request, recursively divides free space in *half* until a small enough block is created
    - allows for extremely *fast* recursive coalescing, can just check if immediate "buddies" are both free and coalesce them
      - because of the recursive division, the address of buddies differ only by a bit (easy arithmetic)
    - suffers from internal fragmentation (fixed powers of 2 sized chunks)

- allocator decides when an allocated resources should be returned to the pool:
  - eg. after `close` , `free` , `delete` , `exit` , or returning from a subroutine
  - if a resource is *sharable* (eg. open file or shared memory segment), resource manager must maintain a count for each resource and free the resource only when the count drops to 0
  - however, keeping track of references to a resource is not always *practical*:
    - some languages support copying references without using OS syscalls
    - some languages don't require programmers to explicitly free memory (Java, Python)
    - some resources may be allocated and freed so often that keeping track of them becomes a significant overhead

- an alternative is **garbage collection**:
    - resources are allocated and *never* explicitly freed
    - only when pool of available resources becomes small does garbage collection occur:
        - **mark-and-sweep** GC:
            - start with a list of original resources
            - scan to find reachable resources by *chasing* pointers
            - remove from original list if reachable
            - free anything still in the original list (unreachable memory)
        - **reference counting** GC:
            - similarly to counting shared resources, count the references to a specific object
            - only free the object when count drops to 0
            - notorious issue of failing to catch *cyclical* references
    - however, must be able to *identify* all *active* resource references:
        - language must *mark* resource references so they can easily be identified on the heap
        - also leads to an overhead when program must *pause* for garbage collection
        - can be mitigated with progressive / incremental background garbage collectors

Paging

- rather than separate memory into *variable* sized pieces for each segment in the **segmentation** approach, separate memory into even smaller, *fixed* sized chunks called **pages**:
    - breaking up segments even *further*, ie. using a finer granularity
    - fixes **external segmentation** caused with segmentation:
        - paging eliminates the requirement of contiguity
        - pages themselves are never *carved* up, granularity is fixed
            - no small, unused memory fragments
    - fixes **internal fragmentation** to a degree:
        - if the page frame is relatively small, internal fragmentation averages only half a page
    - physical memory becomes an array of fixed-sized slots called **page frames**
    - virtual memory (address spaces) is also virtualized with virtual pages
    - pages can still be shared between virtual addresses
    - allows for *flexibility* in abstracting the address space, no more need to keep track of which direction a segment grows

    - provides *simplicity* when allocating space for processes from the free list (fixed sized pages)
- a *per-process* **page table** records where each virtual page of address space is placed in physical memory in a **page table entry (PTE)**:
    - ie. stores address translations for each virtual page (replaces base-bounds registers)
    - simplest implementation is a **linear page table** or array
    - every virtual address can be *translated* by splitting it into components:
        - the **virtual page number (VPN)** indicates which virtual page the address resides on
            - number of bits depends on how many pages in the address space
            - can replace VPN with the **physical frame number (PFN)** to generate the actual physical address by indexing into page table
        - the **offset** indicates the offset within the page
            - stays consistent through address translation
    - also stores meta data such as:
        - **valid bit** that is important for marking unused pages as invalid (no physical frame allocation required)
        - **protection bit** indicating protection
        - **present bit** indicates whether page is in memory or disk (required for page swapping)
        - **dirty bit** if page is modified
        - **reference bit** if page has been accessed
- page tables can become very large
    - aren't stored on-chip, but in physical memory
- since page tables are process specific:
    - need to load pointer to new page table on context switch
    - need to flush previously cached entries
- however for every memory reference, paging requires an *additional* memory reference in order to first fetch the translation from the page table:
    - this can slow down the process by half or more
    - thus the current iteration of paging can cause significant *slowdown* and memory *usage*
        - note that every *instruction fetch* also generates two memory references, one to the page table the instruction is in and then the instruction itself

Example memory access with paging:

```
VPN = (VirtualAddr & VPN_MASK) >> SHIFT // shift by size of offset
PTEAddr = PTBaseReg + (VPN * sizeof(PTE))
PTE = AccessMemory(PTEAddr)
```

```
if (!PTE.Valid)
  RaiseException(SEGMENTATION_FAULT)
else if (!CanAccess(PTE.ProtectBits))
  RaiseException(PROTECTION_FAULT)
else
  offset = VirtualAddr & OFFSET_MASK
  PhysAddr = (PTE.PFN << SHIFT) | offset
  Register = AccessMemory(PhysAddr)
```

Translation Lookaside Buffer (TLB)

- implementing paging made address translation slower with an extra required memory reference:
  - a **Translation Lookaside Buffer (TLB)** is part of the MMU, and is a hardware cache of popular address translations
  - on every virtual memory reference, hardware first checks if the TLB contains the translation
    - if so, translation can be quickly performed without referencing the page table in memory
  - in the common case, translations will be in the cache, and little overhead will be added:
    - want to *avoid* TLB misses as much as possible
    - performance of program is thus as if memory *isn't* virtualized at all
- **caching** in general depends on two principles:
  - **temporal locality** wherean instruction or data that has been recently accessed will be referenced again soon in the future (loop variables or loop instructions)
  - **spatial locality** where programs access memories near each other repeatedly (traversing an array)
  - caches are generally small but fast
    - want to minimize the **miss rate** and maximize **hit rate**
  - types of cache *misses*:
    - a **compulsory miss** occurs because cache is empty to start upon first reference
    - a **capacity miss** occurs because the cache ran out of space and had to evict
    - a **conflict miss** occurs in hardware due to limits on items in a hardware cache
- TLB is usually **fully associative**, ie. one to one mapping between VPN and TLB entries:

  - entry contains VPN, PFN, other bits such as valid bit, protection bits, **ASID**, dirty bit, global bit
  - valid bit for entry indicates if entry contains a valid translation
    - note that the valid bit for the page table indicates if page has been allocated for the process
- address translation with TLB:
  - use the VPN to check if TLB holds translation
  - if so, **TLB hit**:
    - PFN can be found from relevant TLB entry
  - otherwise, **TLB miss**:
    - must go through page table for PFN, and update TLB with the PFN
    - once TLB is updated, hardware retries the translation

Example memory access with TLB:

```
VPN = (VirtualAddr & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success) // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits))
    Offset = VirtualAddr & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else // TLB Miss
  PTEAddr = PTBaseReg + (VPN * sizeof(PTE))
  PTE = AccessMemory(PTEAddr)
  if (!PTE.Valid)
    RaiseException(SEGMENTATION_FAULT)
  else if (!CanAccess(PTE.ProtectBits))
    RaiseException(PROTECTION_FAULT)
  else
    TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
    RetryInstruction()
```

- handling the TLB miss can either be done by hardware or software:
  - with **CISC**, hardware handles TLB miss entirely
    - hardware needs a **page table base register (PTBR)**
  - with **RISC**, software handles the TLB miss:
    - hardware raises an exception, and an OS trap handler updates the TLB and returns from the trap
      - note that this `return-from-trap` must fall back to the original instruction so that it can be *retried* by the hardware
      - must ensure that no infinite chain of TLB misses occurs, so

code for TLB handlers themselves are usually stored in permanent physical memory or permanent translation (*wired* translation) slots

- when context switching, have to somehow clear the TLB since every process has a unique virtual to physical set of translations:
  - can **flush** TLB on context switches, eg. specifically when PTBR is changed
    - however, every context switch will start with many TLB misses
  - can use an **address space identifier (ASID)** in the TLB entries that are process specific
  - when process share physical pages, two TLB entries simply map to the same PFN
    - reduces physical pages in use and lowers overhead
- another issue is **cache replacement**:
  - a common approach is to evict the **least recently used (LRU)** entry in TLB
  - another is to use **random** eviction
- TLB is not a perfect solution:
  - if the number of pages a program frequently accesses exceeds the number of pages in the TLB, there will be many TLB misses
    - known as exceeding the **TLB coverage**
    - one solution is to support larger page sizes
  - TLB can become bottlenecked when using **physically indexed caches**
    - translations must take place *before* cache access
    - one solution is to use a **virtually indexed cache**

Swapping

- another level in the memory hierarchy is the **disk**:
  - with the disk in the memory model, pages no longer *all* reside in physical memory, for very large address spaces, the OS needs to stash away unused parts of these spaces:
    - thus parts of the disk is reserved for swapping and known as **swap space**
    - OS can swap pages in and out of disk in a page-sized granularity
    - OS must also remember the **disk address** of a page
  - disk is *larger* and *slower* than physical memory
  - can also use **demand paging**, which only swaps in pages when they are used
    - because of locality, demand paging is more efficient than swapping in all pages for a process at once

- disk allows for an even larger abstraction of memory, but requires more *machinery* for address translations:
  - when hardware checks the PTE, the page may *not be present* in physical memory
    - stored in a **present bit**
  - if the page is present, the translation can proceed as usual
  - otherwise, this is a **page fault** or **page miss**, the page is in disk
    - page faults never crash, only slow a program down
- on a page fault the OS will always use the **page fault handler** software (even for hardware-managed TLBs):
  - needs to know the disk address, which is additionally stored in the page table
  - look in PTE for disk address, fetches page into memory from disk
    - during I/O request to disk, process becomes blocked
  - update page table to mark page as present
  - update PFN for newly-fetched page in memory
  - backup PC to retry the instruction (could *still* lead to TLB miss, etc.)
- if memory is full, OS may have to **page out** pages to make room:
  - paging and swapping is handled by the **page-replacement policy**
  - OS may also proactively replace pages to maintain bewteen a **low watermark** and **high watermark** number of pages
    - this background replacement thread is called a **swap daemon** or **page daemon**
  - different systems also **cluster** pages together when writing to the swap space, increasing disk efficiency
  - when replacing pages:
    - if in-memory *copy* is **clean**, can replace without writing back to disk
    - if **dirty**, need to write to disk when paging out of memory, very slow operation
    - don't want to be limited to replacing clean pages only
    - can do *pre-emptive* **page laundering** by writing out dirty pages continuously in the background
      - makes the page replacement process much faster
      - ie. *outgoing* equivalent of preloading
      - should only write out dirty, *non-running* pages

Example page-fault exception (hardware):

```
VPN = (VirtualAddr & VPN_MASK) >> SHIFT
(Success, TlbEntry) = TLB_Lookup(VPN)
if (Success) // TLB Hit
  if (CanAccess(TlbEntry.ProtectBits))
    Offset = VirtualAddr & OFFSET_MASK
    PhysAddr = (TlbEntry.PFN << SHIFT) | Offset
    Register = AccessMemory(PhysAddr)
  else
    RaiseException(PROTECTION_FAULT)
else // TLB Miss
  PTEAddr = PTBaseReg + (VPN * sizeof(PTE))
  PTE = AccessMemory(PTEAddr)
  if (!PTE.Valid)
    RaiseException(SEGMENTATION_FAULT)
  else
    if (!CanAccess(PTE.ProtectBits))
      RaiseException(PROTECTION_FAULT)
    else if (PTE.Present)
      TLB_Insert(VPN, PTE.PFN, PTE.ProtectBits)
      RetryInstruction()
    else if (!PTE.Present)
      RaiseException(PAGE_FAULT)
```

Example page-fault handling (software):

```
PFN = FindFreePhysPage()
if (PFN == -1)
  PFN = EvictPage()
DiskRead(PTE.DiskAddr, PFN) // sleep, waiting for I/O
PTE.Present = True
PTE.PFN = PFN
RetryInstruction()
```

Swapping Policies

- under **memory pressure**, OS must use a **replacement policy** to *evict* pages from main memory out to disk:
  - can't control which pages are read in (dynamically done through demand paging), but we can choose which to kick out
  - the *optimal* policy is to replace the page that will be acessed *furthest in the future*
    - impossible to actually implement, serves as an optimal policy to compare against
- **first-in-first-out (FIFO)** policy:
  - simple to implement
  - doesn't understand the importance of pages

        - even if a page has been accessed more often, it may still be kicked out if it was the first one brought in
    - can lead to **Belady's Anomaly** (increasing cache size still increases miss rate) because it does not obey the **stack property**, where a cache of size N+1 naturally includes the contents of a cache of size N
- **random** policy:
    - simple to implement
    - not intelligent in evicting pages
    - *literally* random performance
- **least-recently-used (LRU)** policy:
    - use *history* to guide decisions, *approximate* future behavior
    - more intelligent evicting, closer to optimal
    - **least-frequently-used (LFU)** also used
    - to support context switching, note that *per-process* LRU should be used instead of *global* LRU
    - implementing LRU requires updating some data structure on *every* page access or memory reference:
        - could have hardware support to update the time field in memory on each access (lower overhead)
        - but scanning all time fields when evicting a page and holding so many time fields is extremely expensive
        - could even lead to extra page faults in a purely software implementation (when saved time field is in memory)
        - is there a way to *approximate* LRU?
- **approximating LRU**:
    - requires hardware support in the form of a **use bit** or **reference bit**:
        - one bit per page of system, stored in the MMU
        - whenever a page is referenced, use bit is set by MMU to 1
    - using the **clock algorithm**:
        - consider all pages are arranged circularly
        - check if the currently pointed page has a use bit of 1 or 0
        - if 1, clear use bit to 0, and increment pointer
        - if 0, this page has not been recently used, so it can be evicted
        - search *continues* at the pointer on the next eviction
        - on worst case, will check all pages in the system
        - the guess pointer acts as a *lazy* cache and a recency approximation
            - if the rate of access on the page is faster than the clock *hand*, it has a lower chance of being evicted
    - *modified* or *dirty* pages are expensive to evict, since they must be written back to disk:
        - thus some systems prefer to evict clean pages
        - hardware should include a **dirty bit** to support this behavior, or perform page laundering

    - note that to use the clock algorithm with *per-process* LRU:
        - hardware needs to also maintain the owning process of a page and accumulated CPU time of each process
    - almost as good performance as true LRU
- performance varies based on different *workloads*:
    - workload with *no locality*:
        - LRU, FIFO, and random all perform the same
        - no locality to exploit
    - 80-20 workload (80% hot reference, 20% cold):
        - LRU performs near optimal
    - *looping workload* (N+1 accesses, N cache size, in a loop):
        - worst case for LRU and FIFO
            - consistently accessing older pages
        - random performs near optimal
- OS also uses a **page selection** policy:
    - determines *when* to bring a page into memory
    - usually **demand paging**, or paged into memory when accessed
    - OS can also **prefetch**, but only on reasonable chance of success

Working Sets and Thrashing

- a **working set** size is an optimal number of pages for a process such that:
    - increasing the number of allocated frames make little difference in performance
    - decreasing the number of allocated frames decreases performance greatly
- don't want to clear out all page frames *every* context switch:
    - single *global* pool:
        - approximate global LRU for the entire set
        - interacts very *poorly* with RR scheduling, since the last process in the queue will have all pages swapped out
        - many page faults for the last few processes
        - however, a global LRU may be useful for handling shared pages and segments
    - *per-process* frame pools:
        - allocate a certain number of frames for each process
        - separate LRU for each
        - but different processes exhibit *different* locality
        - need a more dynamic allocation
    - **working-set** based frame allocations:
        - working set is the set of pages used by a process in a *fixed* length

sampling window in the past

-
  -
    - allocate enough page frames for each process's working set
    - change working set for process over time
    - each process runs LRU replacement within its working set
    - doesn't work well for *shared* pages, which may need special handling
- working set *implementation*:
  - another *predictive* algorithm like the feedback queue for process scheduling
  - an optimal working set would be the number of pages needed for the next time slice
  - need to *observe* and sample process to find ideal working set size:
    - adjust number of assigned frames based on paging behavior, eg. faults per time
    - if a process is experiencing too many page faults, it needs a larger working set
    - if a process is experiencing no page faults, it may have too many allocated frames
  - process will also *automatically* fault to have the optimal pages in its set
  - use a **page-stealing algorithm** to find page least recently used by its process:
    - can use similar *clock* algorithm as naive global LRU approach
    - for clock algorithm, need to:
      - associate each frame with a process
      - keep track of every process's accumulated time
      - keep track of a frame's last referenced time
      - aim for a target age for frames
    - ie. can *steal* unused pages from other processes when swapping pages
    - processes needing more pages get more, processes not using their pages lose them
  - utilizes *dynamic equilibrium*
- if there is not enough physical memory:
  - **thrashing** will occur:
    - whenever any process runs, it will steal another process's page
    - leads to many page faults on every context switch
    - all processes run slow
  - some OS use **admission control** to terminate a subset of running processes
    - hopes that the reduced set of processes' **working sets** do not thrash the system
  - other OS run an **out-of-memory killer** to kill the most intensive process
  - cannot add memory or reduce working set sizes

- can only reduce number of competing processes by swapping some out, ie. swap all of a process's pages to disk
- unfair, but we can RR which processes are swapped in and out
- overall, still *improves* performance by *stopping* thrashing
- to *unswap* or reload a process, we can even *preload* the last working set instead of demand paging
  - much fewer *initial* page faults than pure demand paging in this thrashing model

Concurrency

- processes should be used when:
    - programs should be distinct and isolated from interference or failures of another process
    - priveleges are distinct
    - creation / destruction are rare
    - *limited* interaction and resource sharing
- issues of using only processes:
    - *expensive* to create
    - difficult to *communicate* between processes, since address spaces are not shared
    - no way to parallelize activites in a *single* thread
    - cannot *share* resources
        - programmers do not always want to *isolate* programs by creating distinct processes
- in **multi-threaded** applications, each **thread** runs independently but can access memory *shared* with other threads:
    - a thread is a unit of execution with its own stack, PC, registers
        - but with shared address space, code, data, and resources
    - ie. has *more than one* point of execution for the program
    - issue if these shared resources aren't *coordinated* between threads
        - can lead to **nondeterministic** results
    - *pros*:
        - faster and cheaper to create and run, no need to allocate new address spaces
        - allows for **parallelism** on multiple CPUs
            - parallelism leads to improved *throughput*, *modularity*, and *robustness* (one thread failure does not affect others)
        - enables *overlap* of I/O with other operations *within* a single program
        - easy to *share* data and messages
    - *cons*:
        - up to the programmer to create and manage threads, and serialize resource use
        - avoiding **deadlock**
    - OS must support primitives such as **locks** and **condition variables**
    - support for threads can be provided and scheduled directly by either the *kernel* or by *user*-level libraries
        - kernel threads are scheduled by the kernel and can thus use multiple cores or take more time before context switches

            - but operations are much slower overall, and have increased overhead
        - user threads are scheduled by a library, so the kernel has no knowledge of them
            - much faster, can implement functionality using only procedure calls and no kernel involvement
            - but kernel manages them as a *single*-threaded process
- **thread** abstraction:
    - each thread needs its own private set of registers
    - one *independent* stack per thread, ie. **thread-local storage**
    - switching threads (when using *kernel*-provided threads) thus requires a context switch (save and restore registers)
        - state is stored in a **thread control block (TCB)**
    - however, address space remains the *same*
- issue when reading and writing to shared variables due to *uncontrolled* scheduling:
    - leads to **race condition** or a **data race**, where multiple executing threads enter a critical section at the same time:
        - with multi-step updates to object state, state is inconsistent until operation completes
        - race conditions lead to nondeterministic programs where correctness results depend on execution order
        - the piece of code where threads access a shared resource is a **critical section**
            - can occur in non-multithreaded processes as well when sharing other OS resources, eg. files
    - **atomicity**:
        - prevents overlap of operations
        - guarantees started operations *will* complete
    - eg. incrementing a variable is *not* atomic, since a read and write occurs in sequence:
        - one thread reads a variable, and a context switch occurs *immediately* before the subsequent write
        - the next thread reads the *unincremented* variable, and writes the variable incremented by one
        - the first threads writes the *original* value incremented by only one
        - variable appears to be incremented just once, not twice
    - usually occurs around multi-step object state updates
- race condition solutions:
    - have to use **sychronization** to control key points of interaction
    - composed of two independent problems:
        - *serializing* the critical section
        - *notification* of asynchronous completion

    - have a hardware instruction that read and writes **atomically**:
        - usually very specific, specialized instructions
        - cannot reprogram entire critical section in atomic instructions
    - have hardware provided **sychronization primitives**:
        - ie. **mutual exclusion primitives**
        - also need mechanisms to sleep and wake threads while awaiting I/O blocks

Locks

- **locks** are used around critical sections in order to ensure they are executed as an *atomic* instruction:
    - AKA **mutex**, provides *mutual exclusion*
    - after being declared and initialized, locks start out *available*
    - exactly one thread can *acquire* a lock at a time
    - when a thread calls `lock` when another thread owns the lock in question, the function will not return until the owner calls `unlock` on the lock
    - note that there can be different locks for different critical sections
- when implementing locks, have to consider:
    - *mutual exclusion*: does the lock work?
    - *fairness*: does every thread waiting for a lock have the same chance to acquire it?
    - *performance*: is there significant overhead?
- interrupt **masking**:
    - an initial implementation involved simply disabling interrupts during a lock
        - fast performance for brief interrupts, but long interrupts disables greatly impact OS performance
    - *issues*:
        - process may maliciously use locks to exploit CPU
            - unfair with longer disables
        - fails with multiple CPUs
        - important interrupts and operations can be lost (eg. I/O completion)
        - requires a priveleged instruction
        - infinite loop bug or deadlock during resource requests risk
- avoid using shared data:
    - can try sharing read-only data, where order of reads doesn't matter
    - not always an option
- simple load/store attempt:

    - have a simple *flag* variable that is set to 1 on a lock
    - when another thread tries to lock the flag with a value of 1, **spin-wait** until value becomes 0
    - to unlock the flag, set it to 0
    - *issues*:
        - does not guarantee mutual exclusion!
            - reading and setting the flag itself is *still* not atomic, an interrupt can occur
        - spin-waiting is expensive
- spin-locks with *hardware support*:
    - need some hardware support for a **test-and-set** operation:
        - an **atomic** instruction that returns sets a value and returns its previous value
    - use the same load/store implementation with test-and-set
    - the hardware gauranteed atomicity allows this lock to function correctly
    - *issues*:
        - no guarantee of fairness, eg. a thread may spin forever
            - ie. causing **priority inversion**
        - heavy performance overhead, especially with only one CPU, eg. scheduler only schedules blocked threads
        - spin-waiting is expensive, wastes processor cycles
        - thread may spin-wait until an interrupt goes off as it waits for a lock
- spin locks may be appropriate when:
    - critical sections are generally very short
    - there is not much contention for locks
    - spinning does not delay the awaited operation
- other useful atomic hardware primitives:
    - **compare-and-swap** only updates a value if it has an expected value
    - **load-linked** is a typical load instruction
    - **store-conditional** only updates a value if no intervening store has occurred since its address was load-linked
    - **fetch-and-add** increments and returns a value atomically
        - used in **ticket locks** that guarantee all threads progress
- how to minimize spinning?
    - simply **yielding** to the OS when lock is unavailable:
        - this works well with fewer threads, but with more threads, spinning threads may just *continuously* yield to one another (round robin)
        - extra expensive context switches
        - does not address *starvation* and fairness
- instead, use **queues** and sleeping:
    - place waiting threads into a queue or waiting list:

        - when lock is free, can either broadcast to all waiting threads, or just signal one
        - a queue avoids starvation or fairness issues
    - threads go to sleep when the lock is already held, and is woken up by the OS only when it becomes free:
        - ie. move out of the ready queue
        - implemented with **condition variables**
    - using spin-waiting only *around* the lock itself, so the time spent spinning is limited to a few lock and unlock related instructions
    - *issues*:
        - sleep / wakeup race
        - spurious wakeups can be wasteful
- **two phase locks** are an example of a *hybrid* approach with both a spin and a sleep phase
    - since spinning can be useful if lock is about to be released

Lock example with queues and sleeping:

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

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

void lock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; // spin to acquire guard lock
  if (m->flag == 0) {
    m->flag = 1; // acquire lock itself
    m->guard = 0;
  }
  else {
    queue_add(m->q, gettid());

    // precaution against wakeup / waiting race:
    // if interrupt occurs and other thread releases the lock,
    // we don't want this thread to sleep forever.
    // setpark indicates thread is about to sleep, and if an interrupt occurs
    // and unparks before parks occurs, park immediately returns.
    setpark();

    m->guard = 0;
    park(); // put calling thread to sleep
  }
}

void unlock(lock_t *m) {
  while (TestAndSet(&m->guard, 1) == 1)
    ; // spin to acquire guard lock
  if (queue_empty(m->q))
    m->flag = 0; // let go of lock
  else
    unpark(queue_remove(m->q))
    // lock is not set to 0,
    // since the next thread does not hold guard lock anymore
    // ie. passing on the lock to the next thread
  m->guard = 0;
}
```

Condition Variables

- how to allow a thread to check if a **condition** is true before continuing:
    - eg. parent checking whether a child thread has completed
    - a simple implementation would have the parent spin-wait until a shared variable changes value
- threads can wait and *sleep* on a **condition variable** and be **signaled** to continue when a certain event occurs:
    - may use a **waiting list** with FIFO or priority order to choose which threads to wake up
    - in UNIX, `wait` and `signal` are used in conjunction with a state variable *and* a lock:
        - the lock should be held when calling signal or wait
        - sleeping, waking, and locking is built around the variable
    - without a state variable:
        - child runs before parent, parent ends up spin-waiting for a free resource
            - there was no state variable to record the threads's completion
    - without using a lock:

    - race conditions will occur when reading / writing to the state variable
    - leads to a **sleep-wakeup race**
    - a thread goes to sleep as another thread finishes with a shared resource
      - sleeping thread will never be woken up
- **covering conditions** are conditions where a thread should be woken up conservatively, regardless of the cost that too many threads are woken:
  - eg. **broadcasting** to all threads in the waiting list
  - eg. a memory allocation library that does not know which threads to signal when a certain amount of memory is freed
- the **producer/consumer** or **bounded buffer** problem:
  - multiple producer threads generate data in a buffer, consumers consume data from the buffer, eg. piping I/O
- issues in the initial example below:
  - **Mesa semantics**: after a producer wakes a consumer, but before the consumer runs, the bounded buffer is changed by another consumer:
    - possible because signaling a thread simply *wakes* it up, but this is only a *hint* that the shared state may have changed
      - in reality, when the woken thread runs, the state may not be as desired
    - can fix by replacing `if` with a `while`
      - when the woken thread runs, it rechecks the state
  - all threads may end up asleep if a producer is signaled instead of a consumer and vice versa
    - need to use two conditions, so consumers don't signal consumers and producers don't signal producers

Bounded buffer example:

```c
int buffer;
int count = 0;

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

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

```c
int loops;
// cond_t cond;
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    // if (count == 1)
    while (count == 1)
      // pthread_cond_wait(&cond, &mutex);
      pthread_cond_wait(&empty, &mutex);
    put(i);
    // pthread_cond_signal(&cond);
    pthread_cond_signal(&fill);
    pthread_mutex_unlock(&mutex);
  }
}

void *consumer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    pthread_mutex_lock(&mutex);
    // if (count == 0)
    while (count == 0)
      // pthread_cond_wait(&cond, &mutex);
      pthread_cond_wait(&fill, &mutex);
    int tmp = get();
    // pthread_cond_signal(&cond);
    pthread_cond_signal(&empty);
    pthread_mutex_unlock(&mutex);
    // process tmp here
  }
}
```

Evaluating Locking

- locking has *overhead*, especially when implemented as a system call
  - overhead of locking operation may be much higher than time spent in

critical section

- when a lock is not acquired, thread *blocks*, which is much more expensive than acquiring a lock:
  - must yield to OS and context switch
  - when many threads need a single resource, a resource **convoy** is formed, where parallelism is eliminated
    - the resource becomes a *bottleneck*
  - thus, performance degrades as there is more contention for locks
- **priority inversion** can occur when a higher priority, *scheduled* thread is stuck waiting for a lower priority, *unscheduled* thread to give up its lock:
  - the higher priority thread is effectively reduced to a lower priority
  - to solve, temporarily increase the lower priority thread holding the lock, ie. **priority inheritance**
- introduces potential deadlock issues
- to improve locking performance, can either reduce **overhead** or reduce **contention**
  - overhead is usually already highly optimized
- reducing contention:
  - remove critical sections entirely, eg. give all threads a copy of a resource or use only atomic instructions
    - usually unfeasible
  - reduce time in critical section
    - try and minimize code inside lock
    - eg. do memory allocation and I/O outside of lock
    - complicates the code
  - reduced frequency of entering critical section
    - eg. less high contention resource use, batch operations, *"sloppy"* private and global counters
  - remove requirement for full exclusivity
    - eg. read and write locks
  - spread requests out by changing lock granularity
    - eg. coarse vs. fine grained locks, pool vs. element locks
    - less contention with more finegrained locks, but also leads to more overhead with more locks

Locks with Data Structures

- with objects, lock the *object* instead of *code*
- concurrent counter:
  - to make a counter **thread-safe**, simply wrap each increment and read between a lock and unlock
  - expensive performance cost, using multiple threads makes scaled operations much slower
  - want **perfect scaling**, where threads complete just as quickly as the single thread
  - an approach is **approximate counting** or sloppy counting, where each CPU maintains a *local* counter, and once a certain threshold on a local counter is met, a *global* counter is incremented by the local counter:
    - all operations have locks, but local counters won't be in contention with one another
    - scales well, but the global counter is *inaccurate* and approximate
- concurrent linked lists:
  - to make linked lists thread-safe, make a *big* lock for the list, and surround critical sections of operations with locks
    - make sure to surround the minimal, *actual* critical section, eg. `malloc` for a new node should be outside of the lock in case it fails
  - does not scale well
  - can use **hand-over-hand locking**, where each individual node has its own lock:
    - when traversing, code grabs next node's lock and releases the current node's lock
    - still much overhead for so many locks
- concurrent queues:
  - to make queues thread-safe, make a *big* lock for the queue
  - can also use two locks for head and tail of the queue
    - allows for more concurrent operations
- concurrent hash table:
  - can treat hash table as an array of concurrent linked lists
  - thus, uses an individual lock for every bucket
    - allows for more concurrent operations, scales well

Semaphores

- another possible implementation for locks as well as other synchronization primitives

  - originally a mathematical object / concept invented by Dijsktra

- a **semaphore** is an object with an integer value that can be manipulated with two routines after initialization:
  - uses a *counter* instead of a binary flag as well as a FIFO *queue*
  - `sem_wait` : decrement value by one, and wait in queue if value is negative, otherwise return immediately
    - eg. take the lock, await completion, consume resource
  - `sem_post` : increment value by one, if there are one or more threads waiting, wake one
    - eg. release the lock, signal completion, produce resource
    - no broadcasting, only one thread is woken
  - when negative, the value of the semaphore is equal to the number of waiting threads
  - the semaphore should be initialized to the number of threads that can enter the critical section at once
  - *issues*:
    - easy to deadlock with semaphores
    - cannot check lock without blocking
    - no priority inheritance
- a **binary** semaphore is simply another way to use a lock:
  - the value is initialized to 1
  - to lock, thread calls `sem_wait` , which decrements to 0 and immediately returns
    - if another thread tries to lock here, `sem_wait` would decrement to negative and thread would sleep
  - critical section then executes
  - to unlock, thread calls `sem_post` , which increments back to 0, and wakes any other waiting threads
  - want the waiting thread to execute critical section as soon as possible
    - ie. give away lock immediately after initialization
- using semaphores for *ordering* or notifications (similar to condition variables):
  - eg. parent waiting for completion of child thread
  - here, the value should be initialized to 0
  - if parent runs first:
    - parent calls `sem_wait` , decrements to negative and sleeps
    - child calls `sem_post` , increments back to 0, and wakes the parent
  - if child runs first:
    - child calls `sem_post` , increments to 1
    - parent calls `sem_wait` , decrements to 0 and immediately continues execution
  - want the waiting thread to execute only once a condition has been satisfied

        - ie. nothing to give away at the start, waiting for child's completion
- using semaphores for **bounded buffer** problem in below example:
    - ie. using semaphores for counting resources:
        - value should be initialized to number of resources
        - wait consumes resource, while post produces a resource
    - initially, when `MAX = 1` , example works
    - when `MAX` is increased, need to add mutex locks to make `put` and `get` atomic
        - need to ensure scope of mutex lock is correct
        - **deadlock** can occur if mutex lock is outside the conditional variable semaphores

Bounded buffer with semaphores example:

```
sem_t empty, full;
sem_init(&empty, 0, MAX); // 0 indicates semaphores are shared
sem_init(&full, 0, 0);

void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    // sem_wait(&mutex); // leads to deadlock
    sem_wait(&empty);
    sem_wait(&mutex);
    put(i);
    sem_post(&mutex);
    sem_post(&full);
    // sem_post(&mutex); // leads to deadlock
  }
}

void *consumer(void *arg) {
  int i, tmp = 0;
  while (tmp != -1)
    sem_wait(&full);
    sem_wait(&mutex);
    tmp = get();
    sem_post(&mutex);
    sem_post(&empty);
    // process tmp
  }
}
```

- using semaphores for *reader-writer locks*:

- split up locks between reading and writing operations
    - ie. many lookups can proceed concurrently *as long as* no insert is on going
- the write lock functions as an ordinary binary lock
- for readers:
    - the first reader acquires the write lock
    - the last reader to release read lock releases the write lock as well
- not always useful, can introduce excessive overhead
    - readers may *starve* writers

Reader-writer locks example:

```c
typedef struct __rwlock_t {
  sem_t lock;      // basic binary semaphore lock
  sem_t writelock; // allow ONE writer but MANY readers
  int readers;     // # readers
} rwlock_t;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_relase_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last writer lets writelock go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}
```

```c
void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

- the **dining philosopher's problem**:
  - philosophers around a table with a fork on either side
    - each needs a pair of forks to eat
  - broken solution:
    - every philospher grabs left fork and then right fork
    - leads to deadlock
  - solution:
    - one philospher has to grab forks in a *different* order

Semaphore Implementation

```c
typedef struct __sem_t {
  int value;
  pthread_cond_t cond;
  pthread_mutex_lock lock;
} sem_t;

void sem_init(sem_t *s, int value) {
  s->value = value;
  cond_init(&s->cond);
  mutex_init(&s->lock);
}

void sem_wait(sem_t *s) {
  mutex_lock(&s->lock);
  while (s->value <= 0)
    cond_wait(&s->cond, &s->lock);
  s->value--;
  mutex_unlock(&s->lock);
}

void sem_post(sem_t *s) {
  mutex_lock(&s->lock);
  s->value++;
  cond_signal(&s->cond);
  mutex_unlock(&s->lock);
}
```

Deadlock and Other Bugs

- early work on concurrency focused on solving issues of **deadlock**
  - modern applications face more *non-deadlock* rather than deadlock bugs
- non-deadlock bugs:
  - **atomicity violation** bugs occur when a code region is intended to be *atomic*, but is not enforced
    - eg. checking if a struct pointer is `NULL` before dereferencing it:
      - the pointer could be set to `NULL` immediately *after* the check
    - simple solution of wrapping all shared variable references with a **lock**
  - **order violation** bugs occur when the desired order of a group of memory accesses is flipped
    - eg. reading from a struct pointer *before* the struct is initialized
    - simple solution of using **condition variables** to enforce synchronization
- deadlock bugs:
  - **deadlock** occurs can occur in systems with many locks or complex locking protocols
  - two entities each have locked some resource, and each needs the other's *locked* resource to continue
    - eg. thread 1 holds lock A and waits for lock B, thread 2 holds lock B and waits for lock A
    - eg. main memory is exhausted, the OS must swap some processes to disk, swapping processes to disk requires the creation of I/O request descriptors, which must be allocated on main memory
  - because of **encapsulation**, the details of implementations are often hidden, and deadlock can occur even with simple, innocuous interfaces
    - difficult to find or diagnose
  - resource types:
    - *commodity*, ie. clients need a specific amount, avoid deadlock with resource manager
    - *general*, ie. specific file or lock, avoid deadlock at design time
  - 4 conditions are required for deadlock to occur:
    - **mutual exclusion**: threads claim exclusive control of resources
    - **hold-and-wait**: threads hold resources while awaiting additional ones
    - **no preemption**: resources cannot be forcibly removed
    - **circular wait**: there is a circular chain of threads controlling resources a previous thread is requesting
- *avoiding* deadlock:
  - the OS can keep track of free resources, and refuse to grant requests

that would put the system into a dangerously resource-depleted state
    - use **reservations** for commodity resources:
        - resource manager only grants reservations if resources are available
            - eg. `sbrk` , `malloc` are all *failable* requests that allows the OS to avoid resource exhaustion deadlock
        - thus, over-subscriptions are detected before resources are distributed to processes
        - balance between overbooking and under-utilization
        - client apps must be able to handle failure to reserve resources
            - eg. report failure and continue running (reserve critical resources at startup)
            - better to immediately reject request than run into with deadlock or undoing complex half-performed operations
    - use **scheduling** to combat deadlock:
        - with global knowledge of which threads grab which locks, the scheduler can decide if two threads should never run at the same time
        - however, degrades performance and concurrency
- *preventing* deadlock:
    - to counter mutual exclusion:
        - use *shareable* or *private* resources
            - not always feasible
        - design **lock-free** data structures without any locks that utilize atomic hardware instructions
            - eg. using compare and swap to change values or data structures without locks
            - difficulty implementing
    - to counter hold-and-wait:
        - *wrap* lock acquisition in another lock so that all locks are acquired **atomically** at once
            - order of grabbing locks would no longer matter (atomic)
            - requires an additional *global* lock
            - this approach is still hindered by encapsulation, and may decrease concurrency performance
        - force threads to acquire *all* required resources before running
            - all or nothing
        - use **non-blocking** requests, ie. an unsatisfied request will immediately fail
        - *disallow* blocking while holding resources
            - release all locks before blocking, reacquire them in an all-or-none operation after returning
    - to counter no preemption:

        - more *flexible* interfaces can return an error code when locking a held lock, so process can continue and try for the lock again later
            - eg. `pthread_mutex_trylock`
            - order of grabbing locks would no longer matter
            - can still lead to **livelock**, where process still makes no progress since the sequence continuously fails
            - difficult to determine where process should *start over* from if the locking fails
        - allow for resource **confiscation**:
            - have resource leases with *timeouts*, or even *lock-breaking*
            - invalidate a process's resource handle, not always possible (eg. embedded in address space)
            - resources must be designed with revocation in mind
            - last resort
    - to counter circular waits:
        - provide a **total ordering** on lock acquisition so that no cyclical wait occurs
            - may require a convoluted *lock dance* to satisfy ordering
            - eg. release a lock, acquire a lock, reacquire the first lock
        - **partial orderings** can still be used to structure lock acquisition when total ordering is unfeasible
- *monitoring* deadlock:
    - allow deadlocks to occur, but then detect and recover from them
    - for the OS to formally *detect* deadlock:
        - it would have to identify all blocking resources, the owners of the resources, and check if the dependency graph had any loops
        - difficult to identify, process may not actually be blocked
        - OS does not have a way to *repair* the deadlock anyway (killing a random process is not recommended)
    - **health monitoring**:
        - better than deadlock detection, can detect different types of problems eg. live-locks, infite loops and waits, crashes
        - OS uses a combination of different monitoring methods
        - obvious failures such as core dumps, passive observation for hangs
        - eg. monitoring agent watches message traffic or transaction logs for system slowdown
            - however, agent itself can fail
        - eg. requiring servers to send periodic **heartbeat** messages
            - however, application may be running but not responding to requests
        - eg. external services or clients send periodic test requests
            - however, although application is responding to requests, some requests may still be deadlocked

        - have to avoid **false reports** with a certain **mark-out threshold** in order to not overzealously restart functioning processes
    - **managed recover**:
        - services should be designed to be easily restart and reestablish communications
            - eg. different restart types, *cold-start* or reboot, *warm-start* (restore state)
        - restarts should be allowed to occur at different scopes, from single process to list to entire system

Monitor Classes

- implementing parallelism with locks and mutexes can be complex:
    - is there an easier synchronization method for programmers?
    - *encapsulate* sychronization concerns, provide automatic protection
- use **monitor** classes:
    - identify shared resources
    - write code that operates on that object/resource under the assumption that all critical sections are serialized
    - compiler *generates* the serialization
    - every monitor object has a semaphore
        - acquired on any method invocation and released on method return
    - *issues*:
        - monitor locking is conservative and coarse-grained
        - leads to performance problems and reduced parallelism
            - eg. resource convoys and bottlenecks
- Java **synchronized** methods:
    - object's mutex is only acquired for specific methods
    - automatically released on final return
    - finer lock granularity, reduced deadlock risk
    - *issues*:
        - up to developer to identify serialized methods

Event-Based Concurrency

- other approaches to concurrency *without* using threads
    - would allow developers to retain control over concurrency, instead of leaving it up to the OS to schedule threads
- eg. **event-based concurrency**, where the system simply waits for an **event** to occur, then handles it with a specific **event handler**:
    - gives explicit control over scheduling, since when an event is handled,

    it is the only action occuring in the system
  - no locks are needed, since *one* event is handled at a time
  - however, no calls that block the execution of the *caller* can be made
    - would lead to a blocked event-based server

Example of an **event loop**:

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

- *issues*:
  - **blocking system calls**:
    - eg. if the server must fulfill an I/O request for a client, the *entire* server will block until the call completes
    - since only the main event loop is running, ie. no other threads to *overlap* with
  - solution is to use **asynchronous** I/O (AIO):
    - some OS allow I/O requests to return control immediately to caller, before I/O has been completed
    - pass in an **AIO control block** describing what to read from disk to where
    - how to inform user when AIO has completed?
      - user *polls*, ie. periodically checks an AIO error routine to test whether the AIO request has completed
      - OS uses the interrupts and *signals* to inform applications when AIO completes
  - **state management**:
    - AKA **manual stack management**
    - when event handler issues AIO, it must package up some program state for the next handler to use when AIO completes:
      - event-based server would not know how to handle AIO completion otherwise
      - this extra work is not needed in thread-based programs, since state is stored on the thread's stack
  - solution is to use a **continuation**:
    - *record* some information and state in a data structure, and look up the state to process event when AIO completes
    - eg. writing back to client after AIO:
      - save socket descriptor in hash table associated by file descriptor
      - when AIO completes, handler looks up file descriptor
      - writes data to associated socket descriptor

    - harder to exploit **multiprogramming**:
        - multiple event handlers would have to run in parallel
        - would require locks
    - event-based server may still block from *implicit* blocking, such as page faults
    - difficult to manage over time as routines evolve (eg. from non-blocking to blocking)
- for UNIX, there are the `select` and `poll` APIs address receiving events:
    - both check if there is any incoming I/O to process
    - `int select(int nfds, fd_set *readfds, fd_set *writefds, timeval *timeout)` :
        - checks whether `nfds` file descriptors can be read to or written to
        - can be used to build a *non-blocking* event loop

Example using `select` :

```c
int main() {
  while (1) {
    fd_set readFDs;
    FD_ZERO(&readFds);

    // setting bits for descriptors
    for (int fd = minFD; fd < maxFD; fd++)
      FD_SET(fd, &readFDs);

    int rc = select(maxFD+1, &readFDs, NULL, NULL, NULL);

    for (int fd = minFd; fd < maxFD; fd++)
      if (FD_ISSET(fd, &readFDs))
        processFD(fd);
  }
}
```

Persistent Storage

- devices are connected to the CPU through a *hierarchy* of buses:
    - connected to memory through a **memory bus**
    - connected to some high performance devices (eg. graphics) through a general **I/O bus** (eg. PCI)
    - connected to slower peripheral devices (eg. keyboard, mouse) through a **peripheral bus** (eg. SATA or USB)
    - the *faster* a bus is, the *smaller* it is
    - the *larger* a bus is, the *farther* away from the CPU it is
    - buses can send and receive interrupts, and transfer data and commands between the CPU and connected devices
    - on more modern systems:
        - PCIe graphics or NVMe drives connect directly to the CPU
        - a specialized **I/O chip** is used for the rest of device connections

Devices

- many different *peripheral* devices for a computer
    - each needs code to perform its operations and integrate with the rest of the system
- a general **device** is made of:
    - a hardware **interface** presented to the rest of the system
        - eg. certain registers such as **status**, **command**, and **data**
        - is connected to a bus
    - a device-specific **internal structure**
    - a canonical **protocol** to use a device, for example:
        1. OS polls device until the status register is not busy
        2. OS writes data to data register
            - AKA **programmed I/O (PIO)**
        3. OS writes command to command register
        4. OS polls device until the status register is not busy and device is finished
- to avoid frequent polling of device status, and lower CPU overhead:
    - instead of polling, put calling process to sleep
    - device can raise a **hardware interrupt** (transmitted by the bus to the CPU) when its operation is completed:
        - allows a device to do slower work at their own time
        - OS will call a corresponding **interrupt handler** that wakes awaiting process

        - can also **coalesce** multiple interrupts together to lower the overhead of interrupt processing
            - ie. unlike traps, special CPU instructions and operations can enable, disable, or group interrupts
    - if device is *slow*, faster to use interrupts
    - if device is *fast*, better to poll to avoid interrupt and context switching overhead
    - alternatively, use a **hybrid** or **two-phase** approach with both
- avoid time spent writing data and commands with PIO:
    - use a **direct memory access (DMA)** device to orchestrate memory transfers without CPU intervention
    - CPU can run a different process, instead of direct PIO
- OS can interact with device using:
    - explicit, hardware supported, **I/O instructions**:
        - eg. x86 `in` and `out` can write to a specific port and register on a device
        - usually *priveleged* operations
    - **memory mapped I/O**:
        - hardware makes device registers available as if they were memory locations
        - no new instructions required
- to *abstract* devices and keep the OS *device neutral*:
    - only the **device driver**, the lowest level OS software, knows the details of a device:
        - handles converting general application instructions into detailed hardware-specific instructions
        - encapsulates knowledge of how to use, optimize, and handle faults on the device
        - highly specific and modular
            - allows for a “plug and play” interface
    - thus, OS uses a hierarchal file system:
        - high level applications are oblivious to any devices
        - eg. idealized file system class / abstraction makes *generic* requests of standard behavior to a **generic block interface**
        - requests are *routed* through a **specific block interface** to the appropriate device driver that implements the standard behavior:
            - AKA **device driver interface (DDI)**, a standard device driver entry point that can be associated with different OS frameworks eg. disk, network
            - opposite the **driver / kernel interface (DKI)** that specifies the services OS can provide to drivers
    - high end application interface as well as low-level, well-defined hardware interactions at the bus are similar between different related devices

        - device driver acts between these layers
    - core OS code contains common functionalities such as caching, while specialized functionality belongs in the drivers
    - however, a **raw interface** also allows special applications to directly read and write without using file abstraction
    - device driver code dominates OS code (70%)
- UNIX device driver *abstractions*:
    - provides a *class-based* system of different device types
        - ensures all devices in a class have a *minimal* functionality
    - eg. *character* device superclass for devices that transfer one byte at a time, *block* device superclass for devices that deal with a block of data at a time, *network* device superclass for transferring data in packets

Device Performance

- want to achieve better device **performance** during device utilization
- *techniques*:
    - key system device will limit system performance, so it is important to *minimize* their idle time and keep them *busy*
    - exploit CPU and device parallelism:
        - use **direct memory access (DMA)** to move data between any two device on the same bus *without* going through the CPU:
            - designed for large contiguous transfers
            - can *gather* writes from mulitple pages in memory into a contiguous stream on the device
            - can *scatter* reads from a contiguous stream on the device to multiple pages in memory
            - more transfer overhead
        - use **memory mapped I/O** instead when device is executing small sparse transfers:
            - treat registers/memory in device as part of normal memory space
            - more difficult to share, but little overhead
        - can be used whenever CPU is not using the bus
        - minimal overhead imposed on CPU by device
    - allow multiple device requests to be queued, creating *deep* request queues:
        - device is kept busy, maximizing throughput
        - deep queues achieved by many processes and parallele threads making requests
            - or from read-ahead for expected data requests, write-back

        cache flushing
  - consolidate queued I/O requests into fewer, larger transfers:
    - eg. accumulate small writes into a write-out cache of disk blocks, read entire blocks but deliver smaller segments
    - large transfers reduce per-operation overhead

Disks

- the **hard disk drive** is the main form of data persistence:
  - consists of many **sectors** (512-byte blocks):
    - blocks that are near each other are faster to access
    - only single sector operations are guaranteed to be *atomic*
    - muti-sector operations may not complete, ie. **torn writes**
  - can view the **address space** of the disk as an array of its sectors
- *physical* components consist of:
  - **platters** with two **surfaces** each that data can be stored persistently on using magentic charges
  - platters are bound together and spun around the **spindle**
    - **rotations per minute (RPM)** can range from 7k to 15k
    - platters spun at a constant RPM whenever drive is powered
  - data is encoded on platters in concentric circles of sectors called **tracks**
    - a surface can have thousands and thousands of tracks
    - drives may use a **track skew** to account for the seeking of the disk head
    - since outer tracks naturally have more sectors than inner tracks, drives may organize tracks into **zones** with the same number of sectors per track
  - the **disk head** is attached to the **disk arm**, and can read and modify the magnetic atterns on disk
  - the **cache** or **track buffer** holds a small amount of memory for reads and writes
  - drive can acknowledge a succesful write after data is written to memory ie. **write back** or **immediate reporting** caching, or after data is actually written to disk ie. **write through** caching
- accessing sectors from disk:
  - with just a *single* track, the disk would simply wait for the desired sector to *rotate* under the disk head
    - latency is known as **rotational delay**, on average waiting half of the full rotational delay
  - with *multiple* tracks, the disk arm may have to **seek** or move to the correct track:

    - the seek is composed of an acceleration, coasting, and settling time
    - settling time can be quite significant
    - average seek is a third of the full seek time
  - the next phase is the actual **transfer** of data
- *metrics*:
  - the actual *time* for an I/O operation is made up of the time for seek, rotation, and transfer
    - can divide transfer size by operation time for a *rate*
  - drives will perform differently under a **random** (small, random requests) vs. **sequential** (many consecutive sectors) workload
    - eg. high-end *performance* drives vs. low-end *capacity* drives

Disk Scheduling

- the OS decides the order of I/Os issued to disk with the **disk scheduler**:
  - unike job scheduling, can easily estimate how long a disk request will take
    - no preemption either
  - thus disk scheduler will try to follow **shortest job first (SJF)**
- **shortest seek time first (SSTF)**:
  - early approach
  - order queue of I/O requests by the nearest track / shortest seek
  - *cons*:
    - drive geometry not available to OS
      - simply use nearest block to approximate
    - can lead to **starvation** of far tracks
    - does not take rotation into account
- **elevator** or SCAN:
  - move back and forth across disk and services requests in order across tracks
    - *sweeping* motion
  - variants include:
    - **circular** SCAN only sweeps one way before resetting to avoid favoring middle tracks
    - **freeze** SCAN freezes the queue during a sweep to avoid starvation of far-away requests
  - *cons*:
    - does not take rotation into account
- **shortest positioning timing first (SPTF)**:
  - considers both rotation and seeking times
  - if seeking is much slower than rotating, can use previous algorithms

    - in modern drives, rotation and seeking times are similar, so SPTF is used
- other considerations:
    - difficult to implement these algorithms since the OS has no idea on *device-specific* track specifications or current disk head location:
        - in modern systems, disk scheduler selects best *several* requests and issues them to disk
            - disk then uses *internal* knowledge to service requests in SPTF order
    - the disk scheduler should handle **I/O merging** of adjacent requests to reduce number of requests to disk
    - how long should OS wait before issuing I/O request to disks?
        - **work-conserving** approach issues immediately
        - **non-work-conserving** waits so that a "better" request may arrive

RAID

- general *issues* with disks:
    - *slow* and bottlenecking operations
    - relatively *limited* space
    - can be *unreliable*
- **redundant array of inexpensive disks (RAID)** is a technique to use multiple disks together to build a better disk system:
    - looks like a disk externally, but internally includes multiple disks, memory, and even multiple processors
        - ie. specialized computer system running software to operate the RAID
    - *pros*:
        - faster *performance* by using multiple disks in parallel
        - *larger* capacity
        - uses **redundancy** to improve reliability
        - *transparent* deployment due to identical external interface
- **RAID Level 0** AKA **striping**:
    - **stripe** blocks across the disks of the system in a round-robin fashion:
        - the **chunk size** is the size of the stripe (eg. multiple blocks)
        - chunk size affects the *performance* of RAID:
            - small chunk sizes increases read/write parallelism, but the time for positioning also increases
            - opposite for larger chunk sizes
    - want to extract the most parallelism when requests are made for large, contiguous chunks
    - can easily map *logical* memory requests to *physical* requests

        - use mod and integer division operations to find disk number and offset values
    - best capacity, worst reliability (no redundancy)
    - excellent performance:
        - metrics for performance are **single-request latency** and **steady-state throughput**:
            - good latency, essentially redirecting to a single disk
            - best throughput for both sequential and random
        - note for throughput, can consider performance for **sequential** or **random** workloads
- **RAID Level 1** AKA **mirroring**:
    - to tolerate disk failures, make a physical copy of every block in the system
    - different ways to place block copies:
        - **RAID-10** stripes on top of copies
        - **RAID-01** mirrors on top of striping arrays
    - can read either copy, but must write (in parallel) to *both* copies:
        - what if a crash leads to the two copies being *inconsistent*?
            - use a **write-ahead log** so that all pending transactions can be replayed after a crash
            - can use non-volatile RAM for cheaper, faster logging on every write
    - *halved* capacity, best reliability (handles at least one disk failure)
    - okay performance:
        - good reading latency, but slower writing latency (can be parallelized, but have to wait for the slowest disk positioning)
        - halved sequential throughput for both reading and writing (writing copy as well)
            - still losing throughput on reads since disks must skip over copied blocks when positioning
        - good random throughput for reading, but halved random throughput for writing (writing copy as well)
- **RAID Level 4** using **parity**:
    - an attempt to add redundancy with less capacity, but at the cost of performance
    - every stripe across the disks has a **parity** block on another disk that stores redundant information from the stripe
        - parity is calculated using the **XOR** function bitwise on stripe blocks:
            - for a set of bits, returns 0 for even number of 1's
            - returns 1 for odd number of 1's
    - thus, on a disk failure, can easily *reconstruct* the lost bits a column by reading and doing an XOR on each row

  - good capacity (only one disk dedicated to protection), good reliability (tolerates only one disk failure)
  - worst performance:
    - good reading latency, but doubled writing latency
    - good sequential reading, utilizing almost all disks
    - good sequential writing, when writing large chunks of data, can use **full-stripe writes** to perform XORs and write to parity block in parallel
    - good random throughput for reading, utilizing almost all disks
  - worst random throughput for writing:
    - have to update parity block correctly and efficiently when data is changed
    - with **additive parity**, simply read (in parallel) and XOR all the blocks in the targeted stripe, and write new data and parity block in parallel
      - the number of extra reads necessary scales with the *number of disks*
    - with **subtractive parity**, bitwise XOR old against the new data, and update the parity block accordingly
      - the number of extra reads necessary scales with the *size of the changed data*
  - ie. the issue with random writes is the **small-write problem**:
    - even though all the data disks can be accessed in parallel, the parity disk becomes a *bottleneck*, since every write must also write to the parity disk
    - throughput under small, random writes *does not* improve as disks are added
- **RAID Level 5** using **rotated parity**:
  - instead of keeping all parity blocks on a single drive, **rotate** the parity blocks across all drives
  - identical capacity and reliability to RAID Level 4
  - identical sequential throughput and single operation latency as RAID Level 4
  - even better random throughput for reading, can use all disks
  - much better random throughput for writing, since all writing operations can proceed in parallel
    - parity block no longer bottlenecks writing as heavily
    - however, still not as fast as RAID Level 1 performance

File Systems

- a **file system** is an *abstraction* using of raw storage blocks on disk for storing persistent data:
  - an organized way of structuring persistent data for users *and* programmers
  - use directories to put together related data
  - have to deal with both actual **data** and **metadata**, the attributes of a file of data
- want to achieve portability, performance, security, reliability
- general file system structures:
  - divided into fixed-sized **blocks**:
    - 0th block is the **boot block** that would allow machine to boot an OS
    - file systems starts at block 1
  - some blocks are used for storing metadata, eg. file system description, file control blocks, free blocks
  - space is allocated for files in blocks ie. pages:
    - different file systems organize a file's allocated space in different ways
    - eg. linked extents vs. indexed blocks
- layered abstractions are used in order to achieve portability and ease of use:
  - the high level **file system API** lies under the system call layer:
    - independent of underlying file system implementation
    - separated into file container (manipulating files as objects), directory (finding or creating files), and file specific (openning and reading data from a file) operations
  - the lower level **virtual file system (VFS)** layer allows OS to generalize file systems:
    - plug-in interface for different file system implementations
    - clients only see standard methods and properties
  - the even lower level **file system** layer gives support to different file systems by implementing on top of block I/O:
    - all file systems have the same basic functions
    - multiple file systems for different services and purposes
  - the lowest level **device independent block I/O** layer:
    - makes all disks look the same
    - implements standard operations on each block for a device, eg. reads, writes, maps

        - handles device-specific support
        - block I/O is a better abstraction than generic disks:
            - allow for a unified LRU buffer cache for disk data (vital for exploiting locality in memory hiearchy)
            - easier block-sized buffer management

- challenge of file system **control structures**:
    - file system must manage millions of blocks and many files on device
    - have to track and quickly find the data assigned to each file
        - deal with updating, deleting, and adding blocks
    - file control strucutres store these important attributes of a file
    - note that there is an *on-disk* persistent control structure, as well as an *in-memory* structure that points to RAM pages and tracks dirty pages
        - in addition, this in-memory structure should be per-process
- file system free space and allocation:
    - need to maintain a free list of unused disk blocks
    - creating, extending, and deleting files
        - allocating and deallocation should be fast operations
- caching is essential for performance:
    - *read-ahead* caching requests blocks from disk ahead of process requesting them:
        - useful when client is reading sequentially, depends on locality
        - may waste I/O reading unwanted blocks or waste buffer space
    - *write* caching aggregates small writes into larger writes:
        - can eliminate some unecessary writes and build a deeper queue for better disk scheduling
- file system must associate names with low level inode numbers:
    - in **flat name spaces**, all names exist in a single level
    - in **hiearchical name spaces**, more of a graphical representation of files, one unique name per directory
    - also issues of different names / nicknames

UNIX API

- the main *abstraction* associated with persistent storage is the **file**, or linear array of accessible bytes:
    - can be created, read, written, deleted

  - each has a *low-level* name or number (in UNIX, this is the **inode number**, short for index node)
  - can have a file **type**, but this is an *unenforced convention*
    - OS doesn't care about file contents, only delivering the bytes within

- another important abstraction is the **directory**:

  - also has an inode
  - but its content is a list of file name/inode tuple pairs for files contained in the directory
    - UNIX directories have current directory entries `.` and parent directory entries `..`
  - by nesting directories, users can build a **directory tree**
  - everything lies inside the **root directory**
  - **sub-directories** are separated by a **separator** in their **pathnames**
  - file system depends on integrity of directories, so only the OS can write them

- OS keeps track of open files in a *system-wide* **open file table**:

  - entries track information for a process-specific open file:
    - number of references, readability, writability, inode, offset
  - the same file open in different processes may have different file entries in the table:
    - eg. opened with different permissions, accessing different offsets

- at the highest level however, processes can share **file descriptors**:

  - ie. the open-file references stored in process descriptor
  - allows cooperating processes to share a file cursor
  - file descriptors *share* file table entries:
    - in a forked parent/child relationship (more than one reference to same file entry)
    - when using `dup` to redirect I/O

- the full UNIX descriptor *hiearchy*:

  - UNIX user file descriptor
  - open file instance descriptor (file entry in open file table)
  - UNIX struct inode (in-memory file descriptor)
  - UNIX struct diode (on-disk file descriptor)

- `open(filename, flags, permissions)` : open or create a file

  - flags eg. `O_CREAT` , `O_WRONLY` , `O_TRUNC`
  - returns a **file descriptor**:
    - private *per-process* integer

            - every process has an array of file descriptors pointers in its process control block
            - each pointer refers to an entry in the open file table
        - acts as an **opaque handler** to work with the file
        - standard file descriptors are `stdin` , `stdout` , and `stderr`
    - two steps of creation:
        - making a structure associated with an inode that tracks all relevant information for a file
        - linking a *human-readable* name to the file, and placing link into a directory

- `lseek(filedes, offset, whence)` : reposition file descriptor
    - whence eg. `SEEK_SET` , `SEEK_CUR` , `SEEK_END`
    - every open file has an associated current offset
    - does not actually directly cause any disk I/O
- `fsync(filedes)` : force all (buffered) writes fo a file descriptor
    - OS buffers writes using `write` for performance
    - ie. forces all *dirty* data to disk
- `rename(old, new)` : renames files atomically
    - eg. editors use .tmp files when writing out files
- `stat` : get metadata for a file
    - eg. inode, protection, links, owner IDs, size and blocksize, times
- `mkdir` : makes directories
    - cannot directly modify content of directories
    - OS updates directories as new files and subdirectories are created within it
    - on creation, directory has entries refering to itself and its parent
- `rmdir` : removes directories
    - only works with empty directories
- `link` : create alternative links to a file
    - **hard links** create another name in the directory and refers it to the *same* inode as the original file:
        - file is *not* copied
        - both names link to the same underlying metadata associated with the same inode
        - on an unlink, the **reference count** of an inode is decremented

        - inode is only **freed** when this count reaches 0
        - *limitations*:
            - no hard links to directory, can lead to a cycle in the tree
            - can't hard link to files in other file partitions, inodes are unique to a file system
    - with `-s` option, a **symbolic link** is created:
        - a symbolic link is a different type of file
        - holds the *pathname* to the linked to file in its data
            - data size depends on pathname length
        - removing original file leads to a **dangling reference**
        - *not* a reference to the inode, similar to an URL

- `unlink` : unlinks a file
    - removes link between name and inode
    - decrements the reference count of an inode

- files are *shared* between different users and processes:
    - need to offer *varying* degrees of sharing
    - eg. UNIX **permission bits**:
        - three groupings: **user**, **group**, **other**
        - read, write, or execute permissions
        - executing a directory means being able to change directory into them
    - **superuser** or **root** has access to all files regardless of priveleges
    - alternatively, distributed file systems such as AFS use **access control lists**
    - **time of check to time of use (TOCTTOU)** problem:
        - when a malicious user switches a file they have permission to access to another sensitive file before the permissions are rechecked
        - can reduce number of services that needs root permission to run, or prevent following symbolic links

- `chmod` : change permission bits of a file

- assembling a full directory tree from *underlying* file systems:
    - `mkfs` makes a file system on a device (eg. disk partition) with a file system type
        - writes empty file system starting with root directory
    - need to **mount** new file system into the tree:
        - `mount` pastes a file system into an existing directory
        - *unifies* many different systems into a single tree

VSFS Implementation

- many different file system implementations:
  - **vsfs**, or "very simple file system" is a simplified UNIX file system
  - revolves around the data structures and access methods used
- organization of vsfs:
  - divided into **blocks**
  - majority of disk will be the **data region**
  - **inode table** holds inode structures
  - **allocation structures** that track whether blocks are free or allocated
    - eg. free list, or **bit map** where each bit maps to a block
  - **superblock** with information for a particular file system:
    - eg. how many inodes and blocks, file system type
    - used when mounting
  - file control block acts as an index or pointer to all blocks in the file:
    - faster access to particular blocks in file
    - but limited number of extents for a file?
      - use hierarchically structured index blocks, AKA **indirect pointers**
      - some files index blocks contain pointers that point to more blocks
      - with triple indirect pointers, limit can be extremely large
- inode is enough to calculate where on disk corresponding inode structure is located:
  - will contain metadata such as disk pointers to different blocks belonging to file, size, type, protection and time information
  - different ways to refer to data blocks:
    - **direct** disk-address directly point to data blocks (10 direct)
    - **indirect** pointers point to a block with more pointers and allow file size to grow:
      - double, triple indirect pointers are part of the **multi-level index** approach
      - this *imbalanced* tree design works because most files and directories in system are *small*
      - 1 indirect, 1 double indirect, 1 triple indirect
    - linked **extents** include a pointer as well as a length in blocks:
      - still need multiple for incontiguous blocks
      - less flexible, but more compact than indirect pointers
  - alternate **linked list** approach to inodes (DOS-FAT approach):
    - only one initial pointer to first data block, with more pointers at end of each block
      - bad for random, non-sequential access

        - keep an in-memory table of link information that can be easily scanned through
            - structure used by **file allocation table (FAT)** file systems
- directory organization consists of just a list of string name/inode pairs:
    - unlinking a file may leave an empty space, marked with some reserved inode number
    - other file systems use a tree form, which may change the speed of operations
- **free space management** is done with bitmaps:
    - have to scan through to find a free inode or data block, and update bitmap accordingly
    - other approaches include using a **free list**, **B-tree**, or **preallocation** policies
- **access path** when reading a pathname from disk:
    - first, file must be opened with `open`
    - assume file system is mounted and superblock is in memory
    - file system must *traverse* the pathname and locate the desired inode:
        - starting from **root directory**, which is conventionally inode 2
        - look inside (ie. read) inodes recursively for pointers to data bocks
        - no need to check the bitmap, file and its data is *already* allocated
    - once found, permissions are checked, inode is read into memory
        - file descriptor is allocated in the open-file table and returned to user
    - can now use `read` to read from the file:
        - file system consults inode to see where certain blocks lie
        - may update inode with a last accessed time
        - updates file offset in the open-file table
    - on file close, deallocate and remove file descriptor
    - many *auxilliary* reads to read the actual file:
        - must read through directories to locate actual file inode
        - reading a block requires reading the inode first and updating inode after
- **access path** when writing a file to disk:
    - similar recursive opening process with `open`
    - writing to a file may **allocate** more blocks:
        - thus every write consists of generally five I/Os:
            - read bitmap to check which block to allocate to the file
            - write bitmap to reflect the new state on disk
            - read and write the inode to update it with the new block's location
            - write the actual block itself
    - worse write traffic when creating a new file:
        - allocate a new inode:
            - read inode bitmap for free inode

            - write inode bitmap to mark it allocated
            - write inode to initialize it
            - write directory data to link the parent directory to new inode
            - read and write directory inode to update it
        - allocate space inside directory containing new file
- even simple reads and writes are made up of many auxilliary I/O requests
    - every level in pathname hierarchy involves at least two reads to the inode and then data
- **caching** is used to remedy this performance problem:
    - early file systems used a **fixed-size cache** to hold popular blocks:
        - allocated at boot time to be 10% of total memory
        - such **static partitioning** can be wasteful if file system does not use all of cache
    - modern systems use **dynamic partitioning**:
        - give out differing amounts of memory over time
        - eg. integrate virtual memory pages and file system pages into a **unified page cache**
    - caching helps greatly for the file open access path
        - but write traffic *must* go all the way to disk in order to be persistent
    - there are other special-purpose caches such as directory and inode caches
- **buffered writes** are used to help optimize the writing access pattern:
    - lazily delay writes in order to **batch** updates into a smaller set of I/Os
    - also allows OS to *schedule* I/Os for increased performance
    - some writes may be avoided entirely if delayed (file created and then deleted)
    - most file systems buffer writes between 5s and 30s
        - however on system crash, updates may be lost
    - for applications such as databases, correctness is key:
        - file systems offer **direct I/O** interfaces that work around the cache
        - slower, but guaranteed atomic writes

DOS-FAT File System

- **file allocation table (FAT)** for **disk operating system (DOS)**:
    - early file system standard
    - still used heavily, especially for Windows OS
    - divides space into **clusters** (ie. blocks):
        - simple linked list space allocation implementation
        - file control block contains exactly one pointer to the first chunk
        - chunks point to the next chunk

  - characteristics:
    - to find a particular block in file, may have to follow long chain of pointers through FAT
      - but FAT is kept in memory and an auxiliary **chunk linkage** table can be used for faster searches
    - no support for *sparse* files
    - width of FAT determines max file system size
    - only single true qualified file name for a file (local vs. fully qualified name)

- main data structures:
  - **bootstrap**, code executed on startup
  - **volume descriptors** that describe the size, type, and layout of system
  - **file descriptors** that describe a file and points to its data
  - **free space descriptors** that lists unused blocks
  - **file name descriptors** with user-chosen names
- first block of DOS FAT volume contains:
  - bootstrap and volume description
  - file allocation table that works as both a free list and tracking allocated blocks to files
- remaining blocks are data clusters for files and directories
- space is allocated in terms of *physical* single blocks, but also *clustered*, *logical* blocks
- boot block specifically contains:
  - branch instruction to real bootstrap code
  - BIOS parameter block or volume description, with device geometry and file system layout
    - sometimes its own block
  - bootstrap code
  - FDISK table that allows for multiple file system **partitions** on disk:
    - partition table details partition type, active boot indication, start and end disk addresses, number of sectors
    - sometimes included in the **master boot record (MBR)** and bootstrap to choose and find active partition to boot from
- file descriptors for DOS combine both file description and file naming together:
  - every entry contains:
    - a fixed length name
      - if null, directory entry is unused

        - byte of attribute bits such as type, read-only, system, etc.
        - time of creation, last modification, last access
        - pointer to first logical block in the file
        - length of file
    - main drawbacks of simplistic DOS implementation is fixed byte length for names and attributes options

- file allocation table has one entry for every logical block in the volume:
    - if block is free, this is indicated in the entry by a 0
    - otherwise, it is associated to a file, and the entry gives the logical block number of the next *block* in the file
        - -1 is reserved to indicate the **end of file**
    - backup FATs in case of a corrupted FAT

- some versions of DOS have a unique *lazy* deletion feature:
    - instead of freeing blocks when a file is deleted, simply mark the file name as 'deleted' without freeing the data blocks
    - when space runs out, initialize **garbage collection** to recursively free these marked files
    - allows for *recovering* of deleted file contents

- to account for the fixed file name lengths:
    - put extended filenames in *auxilliary* directory entries, tagged with unusual set of attributes
    - new systems would recognize the new, longer directory names and be able to access files both ways
    - this clunky solution retains *backwards compatibility* with older systems

Fast File System

- issues with initial VSFS implementation:
    - disk is treated as if it is random access memory, so seeks between incontiguous memory is not taken into account
    - external fragmentation occurs over time as allocated blocks for files are split up
        - **defragmentation** deals with this, and rearranges data and updates inodes so that disk files are placed contiguously
    - the original block size was too small:
        - small block size minimizes internal fragmentation
        - but leads to more overhead since more blocks must be are transfered

- the **fast file system (FFS)** sought to design file system structures and allocation policies to be *disk aware*:
  - treat the disk like it's a disk
  - while maintaing the same *interface* as VSFS
- structure organization:
  - FFS divides the disk into **cylinder groups**, where each **cylinder** is a set of tracks on *different* surfaces that are the *same* distance from the center of the hard drive
    - consecutive cylinders make up a cylinder group
  - but since modern drives do not export the details of their geometry, the drive can also divided into **block groups**
    - each is a consecutive portion of the disk's address space
  - FFS wants to ensure that accessing two files in the same group will *not* result in long seeks
  - *each* group contains all the structures associated with a single file system:
    - a copy of the **super block** (acts as a backup)
    - pergroup **inode bitmap** and **data bitmap**
    - **inode** and **data block** regions
- allocation policies:
  - FFS tries to *keep related stuff together*
  - placing *directories*:
    - find group with a low number of allocated directories and a high number of free inodes
  - placing *files*:
    - ensure data blocks and inodes of a file are allocated in the same group (preventing long seeks)
    - all files in the same directory in the group of the directory (preserving locality)
- large file exception:
  - large files would entirely fill out a block group, which would prevent locality in this group from being used
  - after a certain amount of blocks for a file are allocated into one block group, the next large **chunk** is placed in another group
  - *spreading* large files will hurt sequential performance on the large files
  - but if the **chunk size** is large enough, file system will spend most of its time transferring data from disk, instead of seeking
    - an example of **amortization**, ie. reducing overhead by doing more work per overhead
- block sizes:
  - FFS used a 4KB block size, which was good for transferring data, but lead to more internal fragmentation
  - FFS introduced **sub-blocks** of 512-byte size that the file system could

use instead:
    - small files would use multiple sub-blocks, until the file size reached 4KB
    - FFS would then find a 4KB block, copy in the sub-blocks, and free the sub-blocks
  - in order to avoid the inefficiencies of these copies, FFS would buffer writes so that the sub-block specification was mostly avoided anyway
- other improvements:
  - special disk layouts:
    - in sequential reads, sometimes the disk would rotate past the subsequent block before the file system issued a read request for its
      - would have to wait for a full rotation before the next block rotated back
    - FFS solved this with a technique called **parameterization**, or skipping a certain number of blocks in the disk layout:
      - for modern drives, the read speed has caught up to the rotation speed, so parameterization is not as useful
      - modern drives may have **track buffers** or internal disk caches to optimize for the locality of a sequential read
  - allowed for **long file names**
  - introduced **symbolic links** and the `rename` operation

Crash Consistency

- a key aspect of file systems is **persistence**:
  - the **crash-consistency problem**:
    - how should file system data structures be updated in the event of a **system crash**?
  - eg. a simple write to a file consists of three operations:
    - writing to the inode to point to the new block
    - writing to the bitmap to mark the new block as allocated
    - writing to the data block with new data
  - if only one or two of these writes occur, the file system may be in a corrupted state:
    - only data block is written: no inode points to new data, no bitmap marks it as allocated, as if write never occured
    - only updated inode is written: we would read garbage from disk, **file system inconsistency** between inode and bitmap
    - only updated bitmap is written: can lead to a **space leak**, where file system never uses data block
    - inode and bitmap are written: no inconsistency, but would read

garbage from disk

-
  -
    - inode and data are written: inconsistency with bitmap
    - bitmap and data are written: inconsistency, no inode points to the data
  - ideally, we would move the file system from one consistent state to another *atomically*
    - impossible, disk only commits one write at a time
- generally, want to order writes:
  - write out data before writing pointers to it
    - unreferenced objects can still be garbage collected
  - write out deallocations before allocations
    - improperly shared data is more serious than missing data
  - still does not eliminate incomplete writes
- early solution, the **file system checker** or `fsck` :
  - allow inconsistencies to occur, and then audit and fix them later (eg. at reboot, using redundant info)
  - can only fix inconsistencies, garbage can still be read from an unwritten data block
  - summary of its checks:
    - make sure references and free lists are correct and consistent
    - check **superblock**, eg. file system size is greater than allocated blocks
      - may decide to use a backup copy of superblock
    - check **inodes** and indirect pointed data blocks to produce correct version of allocation bitmaps
      - trusts information in inodes for bitmaps
    - check **inode state** for corruption and validity of type fields
    - check **inode links** by counting and checking the number of links
    - check for **duplicates**
    - check for **bad block pointers** if they point outside valid ranges or partition sizes
    - check **directories** for integrity of entires and links
  - requires intricate knowledge of the file system
  - are much *too slow*, brute force solution
- modern solution, **journaling** or **write-ahead logging**:
  - before updating the file system, record a note describing the update into a **log** on disk
    - in the event of a crash, can check the log to fix inconsistencies
  - eg. **Linux ext3**, a journaling file system built over **Linux ext2**
    - only additional structure is the *journal* itself
  - a **journal write** consists of:
    - transaction begin block (TxB) with addresses to write to and **transaction identifier (TID)**

        - middle blocks containing the exact contents of the blocks to write too
            - example of **physical logging** vs. **logical logging** which logs a compact representation of the write
        - transaction end block (TxE) marking the end
    - once the transaction is logged, we can overwrite the old structures on disk
        - called **checkpointing**
    - note that journal writing and journal reading (for repair) is always done sequentially and relatively fast
- dealing with crashes *during* a journal write:
    - *slow* solution: each write in the journal would be issued one after the other, guaranteeing that each previous write completed
    - ideally, want to issue all writes to the log at once to turn them into a single, efficient sequential write:
        - but such large writes may be scheduled and completed in different orders
        - if a crash occurs, file system may replay the transaction and write *garbage* data to disk
    - solution: write all journal blocks except for the TxE block, only write TxE block on completion
        - if TxE block is written, journal entry is guaranteed to be in a final, safe state
        - since only 512-byte writes are atomic, TxE block should be a 512-byte block
    - full protocol:
        - **journal write**
        - **journal commit** (write TxE)
        - **checkpoint**, ie. update writes to disk
- using the journal to **recover** from crashes:
    - if a crash occurs before the journal is committed, the pending update is simply skipped
        - don't want to write garbage to disk
    - if a crash occurs after the transaction has been committed to the log, but before checkpointing is complete, the update can be recovered:
        - the log is scanned and committed transactions are **replayed** in order to disk
            - a very small journal is *idempotent*
        - an example of **redo logging**
        - because recovery is a rare event only triggered after crashes, redundant writes from partially complete checkpointing are not an issue
- other journaling *optimizations*:

    - can **buffer** all updates to the log to reduce redundant writes to the same structures (eg. creating multiple files in the same directory)
        - ie. wait longer before commiting to the journal, in case the same disk blocks are updated soon after
    - issues when the **finite** log becomes full:
        - recovering all transactions takes longer
        - no further transactions can be logged to disk
    - treat the log as a **circular log** so that it can be reused:
        - once a transaction has been successful checkpointed, the file system should truncate or *free* this log space from the journal
        - eg. track non-checkpointed transactions in a **journal superblock**, can free all other space in journal
        - achieves both faster recovery and a circular data stucture
    - issues can occur when files are *deleted* and their allocated blocks are immediately *reused*:
        - have to log the change in the journal, otherwise the reallocated blocks may be overwritten with the deleted blocks
        - eg. never reuse blocks until their *deletion* is logged
        - eg. add a new type of **revoke** record to the journal that indicates not to replay certain journal entries
- improving *performance*:
    - journaling doubles every write to the disk
        - must write the data block to the journal first, and also seek between the journal and main file system
    - use **metadata / ordered journaling** instead of the above **data journaling**:
        - ie. log updated inode and bitmap blocks, not data blocks (only written once to proper file system)
        - *order* is key, the data block must be written *before* the metadata blocks are committed and then checkpointed:
            - otherwise, could have a consistent file system with inodes pointing to garbage
            - obeying crash consistency rule to write the pointed-to object *before* the object that points to it
    - since majority of blocks written are data blocks, this greatly improves performance
    - can also issue data block writes and metadata logging in parallel, as long as both complete before the commit
- other solutions:
    - **soft updates**: order all writes to file system such that file structures are *never* in an inconsistent state
        - difficult to implement, requires intriciate knowledge of file system data structures

  - **copy-on-write (COW)**: never *overwrites* files or directories on disk, just places new updates to previously unused locations on disk
    - * after a number of updates are completed, the root structure of the file system is updated with new pointers to the new structures
    - * eg. used in the **log-structured file system**
  - **backpointer-based consistency**: no ordering is enforced, but a **back pointer** is added to every block in the system
    - * can check if a file is consistent if its foward pointer points to a block pointing back to it
    - * another example of *lazy* crash consistency
  - **optimistic crash consistency**: uses **transaction checksums** to issue as many writes at once when journaling (instead of waiting for certain critical writes to succeed)
    - * can greatly improve performance

Data Integrity and Protection

- **reliability** of data in modern file systems is important:
  - RAID systems used a **fail-stop** model of failure, where an entire disk is either working or completely failed
  - other types of single-block, **fail-partial** failures are **latent-sector errors (LSE)** and **block corruption**
  - an LSE occurs when a sector or group of sectors has been damaged:
    - * eg. **head crash** or disk head touches surface, or comsic rays flip bits
    - * **error correcting codes (ECC)** are used to determine whether the on-disk bits in a block are good
    - * rate of LSE increases with *age* of drive and disk *size*
  - block corruptions are not detectable by the disk:
    - * eg. buggy firmware writes to the wrong locations
    - * an example of a **silent** fault
    - * not correlated with LSEs, can vary greatly depending on disk model, even with the same age or disk size
- solving LSEs:
  - since LSEs are easily detected by the file system, can use whatever *redundancy* mechanism to access the correct data
  - eg. using RAID-4 parity bits, or alternate copy in mirrored systems
  - **RAID-DP** uses multiple parity disks for extra reliability
- detecting corruption:
  - with corruption, the issue lies with detecting the silent failure
  - use a **checksum** that hashes a chunk of data into a small 4 or 8-byte summary

    - tradeoff between the *strength* (try and minimize collisions) and *speed* of the checksum function:
        - **bitwise XOR** is fast, but may not detect double-bit errors (differences cancel out)
        - **two's complement addition** is fast, but not good when data is *shifted*
        - **Fletcher's checksum** uses two check bytes and modulo:
            - simple to compute, also catches all single-bit, double-bit, and burst errors
            - not as strong as CRC
        - **cyclic redundancy check (CRC)** is the most commonly used checksum:
            - treat block as large binary number find modulo with some agreed value k
            - efficient implementation, effective
    - storing checksums on disk:
        - can add a single checksum for each block, but this leads to *shifted* sectors:
            - only requires single write when updating disk
            - some drives format with 520-byte sectors to account for checksum instead of usual 512-bytes
        - pack checksums together into 512-byte blocks
            - requires a seek and multiple writes when updating disk
    - using checksums:
        - whenever a block is read, file system needs to also read the **stored checksum** and calculate a **computed checksum** for the block
        - if the checksums do *not* match, a corruption has occured
            - use redundancy mechanism to restore data, or return an error
        - but most data is *rarely* accessed, and would thus remain unchecked:
            - dangerous if unchecked, *bit rot* can occur and could affect all copies
            - use **disk scrubbing** to periodically (eg. nightly or weekly) read *every* block and perform checksum comparisons
- other possible unhandled failure modes:
    - **misdirected write** to the wrong location (eg. wrong block, or right address but wrong RAID disk):
        - add a **physical identifier (PID)** to the checksum that holds the disk and sector numbers of the block
        - lots of extra redundancy on disk, but redundancy is the key for corruption detection
        - allows file system to detect these misdirected writes
    - **lost write** when device confirms that a write has completed, but the write does not physically go through:

    - checksum does not solve this issue, since the checksum and PID will match
    - perform a **write-verify** and immediately reading back data after a write
      - slow, doubles I/Os for a write
    - add a checksum elsewhere, eg. **Zettabyte file system (ZFS)** uses checksums for inodes and indirect blocks in addition to data blocks
      - will only fail if both inode and data writes are lost simultaneously

- **overhead** of checksumming:
  - *space* overhead:
    - extra space for checksums for every block on disk, about 0.2% disk overhead
    - extra space in memory when reading checksums
      - but if checksums are discarded from memory immediately, this overhead is not a concern
  - *time* overhead:
    - much more noticeable than the space overhead
    - CPU must compute checksums when storing *and* accessing data
      - can combine data copying and checksumming
    - extra I/O overhead when writing to checksums that are stored separately

Log-Structured File System

- *motivations*:
  - as memory and cache sizes increase, file system performance becomes more dependent on *writes*, which are not serviced by cache
  - sequential I/O is significantly more performant than random I/O
  - FFS performs poorly for even common workloads such as creating a small file (lots of metadata to update)
  - file systems are not **RAID-aware** (the small-write problem)
- **log-structured file system (LFS)**:
  - ie. the journal *is* the file system
  - good write performance requires not just *sequential* writes, but also large *contigious* writes
  - all writes (even metadata) are **write buffered** in an in-memory **segment**
    - similar to FFS buffering, want to *amortize* the costs of writing
  - when the segment is full, it is then written to disk in one long, sequential transfer to an unused part of the disk
    - eg. when writing out a data block to disk, also *sequentially* write

        out its metadata or inode
            - similar process when creating files, the directory data and inode must also be written sequentially
        - data is never *overwritten*, and segments are always written to *free* locations
            - ie. **shadow paging**, or **copy-on-write**
            - once written, blocks and inodes are immutable
    - *issues*:
        - recovery time to reconstruct index
        - garbage collection and defragmentation
- finding inodes:
    - inodes are no longer in *fixed* locations, but *scattered* all over disk
    - introduce a layer of **indirection** with a **inode map** or index:
        - an array that takes in an inode number and returns the most *recent* version of the inode
        - on new write, index is updated
    - placing the imap on disk:
        - if it resides in a *fixed* part of the disk, this brings us back to square one
        - would have to await seeks between the fixed imap and the newly written free disk locations
        - instead, place *chunks* of the imap right in the log where all the other updates are being written, as it updates *periodically*
        - in addition, imap is cached *entirely* in memory
    - imap solves the **recursive update problem**:
        - problem with any file system that doesn't update *in-place*
        - when an inode is updated, this requires an update to its directory, the parent of its directory, etc.
        - but with an imap, the change is not reflected in the parent directories, since the inode remains the same
            - only the imap structure has to be updated
    - finding and updating the imap:
        - file system must have a fixed and known location to begin a **file lookup**
        - fixed place is the **checkpoint region (CR)**:
            - contains pointers to the *latest* chunks of the inode map (which spread out over disk)
            - updated *periodically*, every 30 seconds, less performance limiting
            - the CR will also be *cached*, so the same number of I/Os are made when reading a file from disk
            - during clean shutdown, the entire in-memory imap is flushed to the checkpoint region

      - after clean reboot, the checkpoint regions is read to create the in-memory imap
- **garbage collection**:
  - LFS has to handle the old, garbage versions of inodes, data blocks, etc.
  - could inventory and maintain old versions in a **versioning** system
  - or periodically *clean* up any dead versions using garbage collection:
    - the cleaner works on a *segment-by-segment* asis, so that large chunks are cleared for later writes
    - *compacts* segments into only their live blocks, write these new, smaller segments to disk, and free the old segments
  - to check for **block liveness**:
    - consult the **segment summary block** at the head of segments to get a list of inodes for written data blocks
    - cross-reference with the imap to see if the data block is still in use, ie. live
    - can also store **version numbers** and check those directly
- policy of cleaning blocks:
  - options for *when* to clean:
    - periodically, idle time, or disk is full
  - options for *what* to clean:
    - segregate **hot** segments whose contents are frequently overwritten from **cold** segments whose contents are more stable
    - better to wait longer before cleaning hot segments, since more of the blocks will be dead and freed for use
- crash recovery and the **log**:
  - segment writes are organized in a log
  - ie. the checkpoint region points to a head and tail segment, and each segment points to the next segment to be written
  - in case of a CR write crash:
    - LFS keeps two CRs at either end of the disk and writes them alternately
    - CRs are written in a special order, ending with a closing timestamp
    - LFS can thus detect inconsistencies in the CRs
  - in case of a segment write crash:
    - read in the latest CR, which is only written periodically for the last consistent write
    - **roll forward** any recent, lost updates by reading the latest segments at the end of the log and finding valid updates

Solid-State Drives

- **solid-state drives (SSDs)** are made of transistors (like DRAM and memory), and have no mechanical or moving parts:
  - however, still retain information despite power loss, ideal for data **persistence**
  - built with **flash** technology, with some unique properties:
    - to write a chunk or **flash page**, a bigger chunk or **flash block** must be erased first
    - fewer *reliability* issues:
      - no mechanical head crashes
      - writing to a page too often can cause **wear out**, ie. over time, harder to differentiate between a 0 and 1 as extra charge on the flash builds up
      - **disturbance** or bit flips can occur
  - **flash chips** store one or more bits in a single transistor:
    - can store one bit **single-level cell (SLC)**, two bits in a **multi-level cell**, three bits in a **triple-level cell**
    - SLC has highest performance and are the most expensive
    - performance vs. capacity tradeoff
  - flash chip organization:
    - made up of **banks** or **planes** with a large number of cells
    - bank can be accessed in terms of larger **blocks** (128 or 256KB) or smaller **pages** (4KB)
    - to write a page within the block, the entire block must be erased first
- basic flash operations:
  - pages start out in an *invalid* state
  - **read** a flash page, very fast operation regardless of location:
    - does not affect the state of the page
    - device is **random access**, no expensive mechanical seeks and rotations
  - **erase** a flash block so that a page within can be rewritten:
    - state of all pages is *erased*, which makes them *programmable*
    - destroys contents of the block, most expensive operation
  - **program** a flash page, less expensive:
    - can only program *erased* blocks
    - state of the page becomes *valid*, which makes it readable but no longer programmable
- from flash to flash-based SSDs:
  - SSD is made up of flash chips, some volatile memory for caching, some control logic for device operation
  - the **flash translation layer (FTL)** in the control logic translates I/O requests on *logical blocks* to low level requests on the *physical blocks* and *physical pages*

  - basic *goals*:
    - fully utilize the flash chips in **parallel**
    - reduce **write amplification** that can occur (extra write traffic by FTL compared to original write traffic to SSD)
    - utilize **wear leveling** to spread out flash writes to reduce wear out
    - avoid disturbances by programming pages sequentially, *in order*
- FTL organization:
  - naive approach:
    - **direct map** logical pages to physical pages
    - ie. to write to a logical page, have to erase the entire physical block and reprogram all the pages
    - very high write amplification, high wear out, terrible performance and reliability
  - log-structured approach:
    - use **logging**, ie. on logical writes, append the write to the next free spot in the currently-written-to block
    - use a **mapping table** (in memory and persistent in some form on device) that stores physical addresses of logical blocks
    - takes care of wear leveling, ie. writes are spread out
- **garbage collection**:
  - ie. dead-block reclamation, have to periodically clear old versions of data on disk
  - specifically, find block with one or more garbage pages, read in the live pages and write them to the log, and finally erase and reclaim the block
  - store some information in each block about which logical blocks are stored in each flash page
    - compare with the mapping table to find dead pages
  - the **trim** operation can also be used to inform the device that certain blocks are no longer needed
  - expensive operation, usually delayed and **backgrounded**
    - some SSDs **overprovision** the device with extra flash capacity so garbage collection can be delayed
- dealing with mapping table size:
  - a **page-level** FTL scheme is *impractical* because the mapping table will become huge in memory (eg. 1GB map for 1TB SSD)
  - **block-level** FTL mapping:
    - FTL only records the mapping between logical blocks and physical blocks
    - the logical address provided by the client consists of a **chunk number** (the logical block) and an **offset**
    - extract chunk number, look up mapping to physical block, and add offset to get physical address
    - however, large performance issues with **small writes**:

        - the entire block must be read and written out, and the mapping table updated
        - must rewrite entire block so that the block-level mapping is *maintained*
    - **hybrid** mapping:
        - aims to achieve flexible *writing* while reducing mapping costs
        - maintain a *per-page* mapping for **log blocks** in the **log table**
            - log blocks are always erased and FTL directs all writes into them
        - maintan a *per-block* mapping in the **data table**
        - FTL must periodically examine log blocks (which have a pointer per page) and **switch** them into blocks that can be pointed to by a single block pointer
    - switching *strategies*:
        - **switch merge**: pages are written to the log in the exact way that their physical pages lie in a block
            - update the data table block pointer, erase the old data block and use it as a log block
            - best case
        - **partial merge**: some pages are written to the log in the same physical layout
            - must reunite all the pages of the physical block
            - read in other pages in the block , append them to the log, and perform a switch merge
            - some write amplification
        - **full merge**:
            - must pull together pages from many other blocks
            - read in many other pages from different blocks, append them all in order, perform multiple switch merges
            - much more write amplication, can seriously harm performance
    - page mapping with **caching**:
        - avoid the complexities of the hybrid mapping
        - reduce the memory used for the map, only cache the active parts of the FTL in memory
        - revolves around exploiting locality and the **working set**
- more wear leveling:
    - although log-structuring approach inherently uses wear leveling, long-lived data may not be overwritten and reclaimed
    - want to ensure *fair* sharing of the write load
    - periodically read out *live* data of such long-lived blocks and write them elsewhere
- SSD *performance* and *cost*:
    - much faster (10x, 100x faster) random access performance than **hard**

  disk drives (HDD)
    - random write faster than random read, log-structured approach turns random write into sequential write
  - slightly better sequential performance than HDD
  - however, sequential still better than random performance
  - 10x more expensive than hard drives

Security

- **security** of OS is especially important since all programs run on some OS:
  - program may have no security flaws, but if OS is exploitable, program will be at risk
  - OS is large, and must deal with complexities such as providing many abstractions and supporting multiple independent processes
    - has complete control of all of its hardware
  - virtualization helps provide some security, eg. virtual memory protects physical memory addresses from processes that are not authorized to access the physical memory
  - system calls to the OS provide another opportunity to implement security measures
    - can use **access control mechanisms** to check if process is **authorized**
  - attackers can **exploit** certain **vulnerabilities**
- a **secure** OS has certain goals and assurances for defined behavior:
  - **confidentiality**: secrets should be hidden
    - eg. process-private memory space
  - **integrity**: state of data is fixed and uncorruptable, related to **authenticity**
    - eg. other users cannot change a file without permissions
  - **availability**: data and services are available, ie. cannot be disabled by an attacker
    - eg. a process should not be able to hog the CPU
  - **sharing**: allow for controlled sharing, ie. different functionality for different users or processes
- *general* OS goals should be converted into more *specific* policies that can actually be implemented:
  - OS provides certain usable security mechanisms, eg. only users A and B can write to file X
  - the **protection** mechanisms then must be *applied* to specific **security** policies in order to help satisfy a security goal
- general **system design** principles:
  1. economy of mechanism: keep system small and simple
  2. fail safe defaults
  3. complete mediation: check security policies every time the same action is take
  4. open design: assume attacker knows all details of the design
  5. separation of privelege, eg. two-factor authentication
  6. minimum privelege: avoid giving unecessary, superfluous priveleges

7. least common mechanism: separate data structures should be used for different users or processes
8. acceptability by users

Authentication

- OS should not allow certain priveleged operations to occur, *unless* the principal is **authenticated** to request the operation:
  - **principals** ie. users request access through an **agent** ie. process to **objects** ie. resources or services
  - OS needs to know the context of *who* making a request:
    - at the start of a system call, OS can check the process data structure to find user and identity of the process
    - OS makes a *policy-based decision* to perform or reject the request
    - can track previous access decisions for future reference in **credentials**
  - principals can be categorized into users, groups, or a specific program
- *attaching* identities to processes:
  - processes inherit user IDs
  - when a process is created, check the *creating* processes control block for its identity, and copy over the identity
    - eg. on a `fork`
  - the ownership of a user's *initial* shell or window manager process is set by the priveleged OS to the new user when they join the system
    - can only do this if the user is **authenticated** to be who they claim they are
- authentication by *what you know*:
  - eg. passwords or challenge questions:
    - the longer and more possibilities for a character in a password, the harder to brute force
    - attackers may use a **dictionary attack** with common or familiar words
      - most systems lock accounts or drastically slow down password checking after a certain amount oftries
  - we cannot control if other people know a user's password, but what about the OS knowing the user's password?
    - store and compare a **hash** of the password
    - **cryptographic hashes** cannot be reverse-hashed, and also cannot be analyzed to find clues about the original password

  - what if attacker creates a dictionary by hashing the common words?
    - the same cryptographic hashing algorithm is used by most systems
    - use a **salt**:
      - generate a large random number, concatenate it to the password, hash the result
      - random number must also be hashed and stored
  - *cons*:
    - if not properly hashed, susceptible to *network sniffing*

- authentication by *what you have*:
  - eg. ID cards
  - some systems accept hardware security **tokens** over USB, eg. **dongles**:
    - some smart tokens display a key that the user then types into the computer to authenticate
    - tokens must *change* the information they pass to the OS frequently
      - otherwise, attacker can learn this *static* information required from the token
  - *pros*:
    - two-factor authentication with smartphones is a very common authentication option
      - one method compensates for the others drawbacks
      - smartphones are ubiquitous
  - *cons*:
    - user may not have their token with them
    - attacker can obtain the physical token

- authentication by *what you are*:
  - eg. fingerprints, face ID
  - systems should distrust preconstructed biometric information, since attackers may simply *create* a pattern of bits
    - hardware performing the scanning should be *physically* attached to the machine
  - multiple authentication mechanisms can be used in conjunction, to correct for failures of one type
  - *cons*:
    - very complex to implement, requires special hardware
      - not a simple bit-by-bit equivalence check, pictures can be taken in different lightings, different haircuts
    - these *biometric* implementations will have a *false-positive* and *false-negative* rate:
      - impossible to minimize both, inversely related
      - different contexts seek to minimize a different rate

- can also authenticate by location, ie. *where you are*
- authenticating *non-humans*:
  - eg. a web server or smart embedded devices
  - allow priveleged users to create processes that do not belong to them, or processes to change their identity
    - require strong controls

Access Control

- once a process has an identity associated with it, OS needs to figure out if its requests fit within the security policy:
  - **authorization**: determine whether a **subject** is allowed to perform a particular mode of **access** on an **object**
  - the code that performs authorization is known as a **reference monitor**
    - tradeoff between efficiency costs and security benefits
  - to actually provide access, OS may map a resource into a process directly, or provide a capability
- using **access control lists (ACL)** for access control:
  - eg. doorman with list of all approved entrants
  - opening a file in OS using ACL:
    - every file has its own access control list, shorter lists, faster searches
    - OS obtains requesting process's owning user
    - check file's ACL if user is listed for the type of requested access
  - other considerations:
    - *size* of the ACL
      - large, fixed per-file lists can be wasteful, long searches
      - don't have to reserve enough space in the ACL for many users
    - storing *location* of the ACL:
      - alongside metadata locations being read anyway, such as the inode, directory, or first data block of the file
  - tried-and-true UNIX solution:
    - with limited size file system, only 9 bits available for ACL
    - allow for three entries, with three *modes* of access (read, write, execute) for each
    - no room to store what user the entry pertains to?
      - partition the ACL into three preset *categories*:
      - the user (stored in inode already), group ID (also stored in inode), everybody else (find by complement)
    - embed 9 bit ACL in the inodes, no extra reads or seeks to obtain

        - no expensive search through arbitrary sized list
        - however, unable to express more complex access modes and sharing relationships
    - *pros*:
        - determining the set of principals who can access a resource is easy
        - to authorize, simply check the ACL
        - to update authorized subjects, only need to update the ACL
        - ACL stored in or near file, ie. can get relevant control information if OS can get to the file
    - *cons*:
        - have to store ACL near file and deal with long ACL and getting to ACL
            - or use UNIX solution but be limited in modes and sharing
        - difficult to express the opposite relationship, ie. list all resources a principal is permitted to access
        - difficulties translating user name spaces across different domains, eg. different web servers
- using **capabilities** for access control:
    - eg. keys or tickets for an event
    - open a file in OS using capabilities:
        - application provides a capability or OS finds one
        - OS checks if capability allows user to access the file
    - a **capability** is a set of bits:
        - have to be fairly long and complex, unforgeable
        - danger of a process being able to maliciously recreate or redistribute capabilities
        - could be implemented as just the pointer to the resource
        - thus, process should never be able to access its capabilities
            - OS mediates all operations involving the capabilities, eg. through system calls
    - storing capabilities:
        - OS can maintain per-process lists in the PCBs
            - check for relevant capabilities for a specific operation
            - these lists can get large, so only part of a user's set is stored in the PCB
        - can also cryptographically protect capabilities and store out of the OS
    - *pros*:
        - easy to determine which resources a principal can access
        - can be faster than ACLs (no search)
        - easier to transfer a subset of parent's priveleges
            - for ACL, to change from the parent's priveleges, need to create a new principal and modify its ACLs

    - *cons*:
        - determining the set of principals who can access a resource is more expensive
        - have to provide mechanism for revocation
        - may have to use cryptography to prevent forgery
    - ACLs are used more frequently than capabilities
    - some systems use both:
        - eg. file descriptors act as a capability, while file permissions act as the ACL
        - where as long as a process keeps a file open that it has been authorized for, the ACL is not consulted
        - instead a file descriptor (ie. capability) is attached to the PCB, that is checked instead
- usually, the decision of what the access control for a resource is falls to the *owner* of the resource
    - **discretionary access control**
- sometimes, access control decisions are mandated by an authority, eg. like secret, confidential data
    - **mandatory access control**
- **role-based access control (RBAC)** is used when a system has distinct, specialized roles that users are in
    - similar to groups in ACL, but more strict type enforcement with specialized permissions / allowed operations
    - would use mandatory access control
    - can be implemented using *privelege escalation* with `setuid`

Cryptography

- when data is sent between machines, want to preserve:
    - **authentication** of the sender
    - **integrity** of secret data
    - **privacy** of secret data
    - cryptography helps will all of these goals
- **cryptography** is a set of techniques to convert data from one form to another:
    - if done correctly, impossible to determine the original data by examining the protected form
    - must allow for reversing transformation, while making it difficult for an attacker to do so
    - computationally expensive, have to be selective about where to implement cryptography
- uses an algorithm called a **cipher** and a **key** to encrypt some data:

    - translate *plaintext* into *ciphertext* and vice versa
    - there is a reverse transformation that takes the key and encrypted data to return the decrypted data
    - very difficult to tell what plaintext a specific ciphertext decrypts to
    - must be a *deterministic* operation
- however, attackers will know which cryptography cipher is used:
    - a weak cipher would allow an attacker to decrypt without the key
    - there are very few strong ciphers, eg. AES
- thus, the cryptography's benefit relies *entirely* on the *secrecy* of the key:
    - if key is secure, our encrypted data is secure
    - can check for altered, tampered data using hashes
- **symmetric** cryptography:
    - the same key is used to encrypt and decrypt data
    - can use for authentication, since users can only decrypt if they have the correct key
    - selecting keys should be done *randomly*
        * on Linux, need to gather *entropy* for an approxmiation of true randomness
    - eg. **Data Encryption Standard (DES)**, **Advanced Encryption Standard (AES)**, Blowfish
    - cracked by **brute force**, too expensive for AES
    - *pros*:
        * encryption and authentication performed in a single operation
        * very fast
        * no centralized authority required
    - *cons*:
        * hard to separate encryption from authentication, which complicates signatures
        * key distribution is an issue
        * scaling, especially on the internet
- **asymmetric** or **public key** cryptography:
    - keys are created in pair
        * the key to decrypt is different from the key to encrypt data
    - one key can be *public*, while the other *secret* one remains private and secure
        * in addition, can encrypt with the public key, in which case, only the secret key can decrypt
    - safer for authentication, no need to ever distribute the secret key
    - encrypting with secret key allows *authentication*, while encrypting with public key allows *secret communication*:
        * eg. sign a message by encrypting with your own private key
        * eg. send an encrypted message to A by encrypting with his public key

  - for both authentication and private communication, multiple key pairs are required
    - eg. A encrypts with A's private key to authenticate only A could have created the message
    - A also encrypts their message with B's public key so only B can read the message
  - eg. RSA, elliptic curve cryptography
  - *cons*:
    - need key distribution infrastructure or certificates
    - PK cryptography is hundred times slower than symmetric cryptography
- common to use both types of cryptography in a single session:
  - asymmetric crypto used to *bootstrap* symmetric crypto
  - use RSA to authenticate and establish a **session key**, use AES for the rest of the transmission for faster speed
- integrity checks on data uses hashes to see if data has been corrupted:
  - hash the data and encrypt the hash
  - however hash functions still have hash **collisions**
- with **cryptographic hashes**:
  - it is infeasible to find two inputs that hash to the same value
  - change to the input causes an unpredictable change to the hash value
  - infeasible to infer properties of the input from the hash value
- thus, if integrity, not secrecy, is important:
  - take a cryptographic hash of the data, and encrypt only the hash
  - share the data and the encrypted hash
- **at-rest** data encryption:
  - preserving secrecy and integrity of disk data
  - store data in its encrypted, not plaintext, form:
    - eg. in **full-disk encryption** the entire contents of the storage device are encrypted
    - at boot time, key is obtained, and encryption and decryption occurs continuously, transparent to users
  - protects against reading of storage hardware after being transferred or stolen from another system
  - does *not* protect against:
    - users accessing data they shouldn't, disk encryption is separate from access control mechanisms
    - application flaws that divulge data
    - OS security flaws
- **cryptographic capabilities**:
  - solves the issue of forgeable capabilities
  - an encrypted data structure that indiciates access to a particular resource is created and given to a users

    - the user would use the capaility to access the resource
  - with *symmetric* cryptography, the capability creator and system need to share the same key
    - so both are usually are the same system
  - with PK cryptography, creator and resource controller do not need to be co-located

Appendix

UNIX Syscalls

- `sighandler_t signal(int signum, sighandler_t handler)` : handles signals, registers signal catchers
  - in `signal.h`
  - if `signum` is delivered to the process:
    - if `handler` is set to `SIG_IGN` , the signal is ignored
    - if `handler` is set to `SIG_DFL` , the default action occurs
    - if `handler` is set to a function, the function is called with argument `signum`
  - note that the signals `SIGKILL` and `SIGSTOP` cannot be caught or ignored
  - returns the previous value of the signal handler, or `SIG_ERR`
    - `errno` set on errors
- `int kill(pid_t pid, int sig)` : sends signals to a process
  - in `sys/types.h` , `signal.h`
  - if `pid` is positive, signal `sig` is sent to process with matching PID
  - if `pid` is 0, `sig` is sent to every process in the process group of the calling process
  - if `pid` is -1, `sig` is sent to every process possible
  - if `sig` is 0, no signal is sent, but existence and permission checks still occur
  - returns 0 on success, returns -1 and `errno` set on error
- `send(int sockfd, const void *buf, size_t len, int flags)` : sends message on a socket
  - in `sys/socket.h`
  - used with a connected socket
  - supports various flag options
  - returns number of bytes sent on success, return -1 and `errno` set on error
- `recv(int sockfd, const void *buf, size_t len, int flags)` : receive message from a socket
  - in `sys/socket.h`
  - supports various flag options
  - if message is too long, excess bytes may be discarded
  - returns number of bytes received, return -1 and `errno` set on error
- `mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)` : map or unmap files or devices into memory

  - creates a new mapping in the virtual address space of the calling process, starting at `addr` for `length`
  - the contents of the file mapping are initialized by `length` bytes from `fd` starting at `offset`
  - `prot` specifies memory protections
  - returns a void pointer to the mapped area, return -1 and `errno` set on error
- `flock(int fd, int operation)` : apply or remove advisory lock on an open file
  - `operation` can be `LOCK_SH` to place a shared lock, `LOCK_EX` for exclusive lock, and `LOCK_UN` to remove an existing lock
  - file can only have one type of lock
  - duplicate file descriptors refer to the same lock
  - returns 0 on success, returns -1 and `errno` set on error
- `lockf(int fd, int cmd, off_t len)` : apply, test, or remove POSIX lock on an open file
  - applies to only a section of a file, starting at the current file position for `len` bytes
  - `cmd` can be:
    - `F_LOCK` to set an exclusive lock, blocks until release if already locked
      - overlapped locks are *merged*
    - `F_TLOCK` same but call never blocks
    - `F_ULOCK` unlocks section
      - file locks released on file close
      - may split into two locked sections
    - `F_TEST` tests the lock
      - 0 if unlocked or locked by process, -1 if other process holds lock
  - returns 0 on success, returns -1 and `errno` set on error
- `select(int nfds, fd_set *readfds, fd_set *writefds, struct timeval *timeout)` : synchronous I/O multiplexing
  - monitor multiple file descriptors until they become ready
  - returns number of file descriptors on success, returns -1 and `errno` set on error
- `poll(struct pollfd *fds, nfds_t nfds, int timeout)` : wait for one of a set of file descriptors to become ready for I/O
  - `struct pollfd` includes a file descriptor, requested events, and then actual returned events from `poll`
  - events can include:
    - `POLLIN` for reading, `POLLOUT` for writing, `POLLERR` for errors, `POLLHUP` for hangup
  - returns number of structures with nonzero events on success, returns

    -1 and `errno` set on error

- `sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` : examine and change a signal action
    - install action from `act` if not `NULL`
    - previous action is stored in `oldact` if not `NULL`
    - returns 0 on success, returns -1 and `errno` set on error

Sockets Example

- **stream** socket vs. **datagram** socket:
    - datagrams are more *unreliable*, ie. packets can be lost
        - TCP protocol with streams will detect and *retransmit* lost messages
    - datagrams preserve message *boundaries*
        - stream sockets may divide messages into chunks
    - much less *overhead* (no initialization/breakdown, no package acknowledgement), so used for short services

Server example:

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(int argc, char *argv[])
{
     int sockfd, newsockfd, portno, clilen;
     char buffer[256];
     struct sockaddr_in serv_addr, cli_addr;
     int n;

     /* create socket:
      * AF_UNIX local, AF_INET network
      * SOCK_STREAM continuous stream, SOCK_DGRAM chunks */
     sockfd = socket(AF_INET, SOCK_STREAM, 0);

     bzero((char *) &serv_addr, sizeof(serv_addr));
     portno = atoi(argv[1]);
     serv_addr.sin_family = AF_INET;
     serv_addr.sin_addr.s_addr = INADDR_ANY;
     serv_addr.sin_port = htons(portno);
     /* bind socket to an address: */
     if (bind(sockfd, (struct sockaddr *) &serv_addr,
              sizeof(serv_addr)) < 0)
              error("ERROR on binding");

      /* listen for connections: */
     listen(sockfd,5);

     clilen = sizeof(cli_addr);
     /* handle multiple connections */
     while (1) {
         /* repeatedly accept a connection, return new fd: */
         newsockfd = accept(sockfd,
               (struct sockaddr *) &cli_addr, &clilen);
         if (newsockfd < 0)
             error("ERROR on accept");
         pid = fork();
         if (pid < 0)
             error("ERROR on fork");
         if (pid == 0)  {
             close(sockfd);
             /* write and read from new fd */
             dostuff(newsockfd);
             exit(0);
         }
         else close(newsockfd);
     }
}
```

Client example:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>

int main(int argc, char *argv[])
{
    int sockfd, portno, n;
    struct sockaddr_in serv_addr;
    struct hostent *server;
    char buffer[256];
    portno = atoi(argv[2]);
    /* create socket */
    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    server = gethostbyname(argv[1]);
    if (server == NULL) {
        fprintf(stderr,"ERROR, no such host\n");
        exit(0);
    }

    bzero((char *) &serv_addr, sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    bcopy((char *)server->h_addr,
         (char *)&serv_addr.sin_addr.s_addr,
         server->h_length);
    serv_addr.sin_port = htons(portno);

    /* connect to server: */
    if (connect(sockfd,(struct sockaddr *)&serv_addr,sizeof(serv_addr)) < 0)
        error("ERROR connecting");

     /* write and read from new fd */
    bzero(buffer,256);
    printf("Please enter the message: ");
    fgets(buffer,255,stdin);
    n = write(sockfd,buffer,strlen(buffer));
    bzero(buffer,256);
    n = read(sockfd,buffer,255);
    printf("%s\n",buffer);
}
```