CS111: Operating Systems

Xu

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, abstracts and hides the low level details eg. hardware and ISAs
  - uses technique of **virtualization** to transform a *physical* resource into a generalized, easy-to-use *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 *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 itself as well as **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
    * separate general frameworks and policies from specialized hardware mechanisms
  - only OS/kernel can work with the *priveleged* ISA set, but standard ISA set is accessible by all
- OS abstracts ISAs into a set of management and abstraction *services* accessible through a **system call interface**
- the OS code is *unique* from application code:
  - 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. jumps into the priveged *kernel* mode and raises hardware privelege level)
    * **procedure calls** that are provided as a library and are accessible in the *user* mode

Abstractions

- OS handles *abstracting* resources
- **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.
  - **partitonable** - 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 for 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 abstractions**:
  - 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 thus 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 abstraction**:
  - at a low level, there are many different related data storage 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** - the *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
    * 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, just call subroutines
      · at a lower level: push parameters, jump, 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 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**:
      · 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**
    * forces 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 performs 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 and implementations
  - but interfaces and 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 system call **ABI** for Linux x86-64 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

Creating Programs and Linking

- general software file classes:
  - **source** files are editable text files in a programming language
  - **object modules** are relocatable sets of compiled or assembled instructions from source files
  - **libraries** are collections of object modules
    * source files can fetch functions from them
    * order in which libraries are searched can matter
  - **load modules** are complete programs that can be loaded into memory and executed into 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 CPU initialization, traps/interrupts, sychronization
    - however, some functions and data may not yet be present and not all memory addresses are finalized
      - ie. references and addresses can only be 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
    - *symbol tables* are used primarily for debugging
- **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
  - **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 pro-

    grams
    * difficulty keeping static libraries updated (version is *frozen*)
- **shared** libraries (linktime binding, mapped into memory at runtime):
  - reserve an address
  - linkage edit libraries into code segments
  - includes redirection table (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
    * library changes (eg. size, new routines) easy to update
    * 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 work on clients without the used 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
  - application receives standard *entry points* to make calls to the DLL through
  - maintains a *table* of entry points for different DLLs
  - on DLL shutdown, application asks OS to unload module
  - loading DLLs is done through an API, but the actual loading mechanism is ABI-specific
  - *pros*:
    * runtime binding, 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 relative, unloaded addresses
  - a process has 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 paramters, saved registers)
      - grows downward
    - **stack overflow** occurs when stack and data segment meet, protects from data corruption
  - can also interpret a process as a virtual, private computer, or an *object*
    - the **state** of a process should consistently, uniquely, characterize the process
    - consists of the metadata, allocated memory, opened files, condition of an I/O operation, etc.
  - 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, comprising of:
  - **memory** to store instructions and data, every process has an **address space**
    - the address space is the *virtual memory addresses* reserved for a pro-

cess (illusion of infinite memory)
  - **registers** that are used during execution
    * some special registers include the **program counter** that indicates the next instruction, **stack pointer**, and **frame pointer**
  - **I/O information** for open persistent storage devices
  - other OS-related state information

Subroutine Stack Frames

- calling a subroutine:
  - *parameter passing* involves placing parameters into registers
  - *subroutine call* involves saving the *return* address on the stack, and transferring control to the entry point
  - *register saving* involves saving certain nonvolatile/callee-saved registers so that they can be restored
  - *space allocation* for local variables
- 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 *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* or **process descriptors** to track the state of processes. These include:
    - **process/task table** 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 iterrupted processes so that the OS can return back to the process
            - must be separate from user stack for *security*, kernel stack used for priveled operations
            - must be per-process since different processes will experience *different* interrupts and system calls
            - saves 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>

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>
#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>
```

```c
#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

Process Mechanisms

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 buffers (similar to files) from `pipe` that are different from static files in the following ways:
  - simple byte stream
    - 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 another inter-process mechanism
    - 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**
  - 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:
  - much faster than other models
  - processes must run on the same memory bus on local machine
  - race conditions, sychronization issues
  - no authentication or security from OS

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 *default* action on certain signals

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 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 executes 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 (PC)/program status (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 for 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	

OS at run (kernel mode)	**Hardware**	**Program** (user mode)
	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`)
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 resumes execution of *another* process
  - 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) $\to$ 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) $\leftarrow$ 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, **real-time** scheduling
  - 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
  - maintain performance by rejecting work
  - 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, difficult for real-time and priority scheduling
    * *preemptive pros*: good response time, fair, good for real-time and priority scheduling
    * *preemptive cons*: more complex, requires context switch mechanism, not as good throughputs, higher overhead

- 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
  - 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 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 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*
    - eg. when a job repeatedly makes I/O requests to the keyboard, it will have its high priority *maintained* (interactive process)
    - eg. when a job uses the CPU intensively for long periods of time, it will have its priority *reduced* (response time isn't important)
    - (3) When a job enters the system, it is placed at the *highest* priority
    - (4a) If a job uses an entire time slice, its priority is *reduced*
    - (4b) 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
  - 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**:
  - (4) 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
  - (5) 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, interact 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 paramters
  - 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

- 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
  - virtual address 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, grow in opposite directions
  - the program is not in *contiguous* physical memory like the address space, but loaded at *arbitrary* physical addresses
    - eg. printing pointers in C prints virtual 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** or *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 *compile-time* operator
  - `free` frees heap memory
- `calloc`, `realloc`

```c
#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**
  - **dangling pointer**
  - **double free**
  - **invalid free**

Memory Mechanisms

General Partition Strategies

- **fixed partition**:
  - preallocate 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** - wasted space inside *fixed* blocks
- **dynamic partition**:
  - similar to fixed, except *variable* sized blocks
  - each partition is 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 tied to address range
  - still subject to fragmentation
  - not as large as virtual address space

- can use a **free list** to track unallocated memory
  - allow processes to use incontiguous, *variable* sized partitions
  - each element in the list has a metadata **descriptor** with data such as length, free or not, 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, since process can use smaller sized chunks as needed
  - however, over time, leads to **external segmentation** - useless, small leftover chunks
  - different free space management strategies help counteract external segmentation:
    - different allocation algorithms, eg. first-fit, next-fit
    - **coalesce** adjacent free memory chunks together
    - is it possible to relocate free memory and **compact** it 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:
  - 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*, 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 handlers
  - *OS*:
    - allocate memory for new processes using a **free list** (data structure holding free slots in physical memory)
    - 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
- dynamic relocation can lead to **internal fragmentation**, where the space inside the allocated unit of a process is wasted, since its stack and heap are small
  - restricts address spaces in fixed-size slots
  - cannot run programs where the entire address space doesn't fit into memory
  - issue compounded with larger, **sparse 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, **fine-grained** segmentation involves more smaller segments, usually with a **segment table**
  - allows OS to place segments in different parts of physical memory *indepedently* and fill *unused fragments*
    - more flexibility when allocating for processes with large, sparse address spaces
    - 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 - virtual address of start of heap, physical address = offset + base(heap)
  - to read into the stack that *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, 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:

```
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
  - issue of **external fragmentation**, when physical memory becomes full of small holes of free space 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 the opportunity 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 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
- 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
    - 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 coalesce
    - **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 expensive when allocating memory
    - can we minimize these actions?
    - **segregated lists** AKA **buffer pool**
        * 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**
        * 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 to count based freeing is **garbage collection**:
  - resources are allocated and *never* explicitly freed
  - only when pool of available resources becomes small does garbage collection occur:
    * 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)
  - however, must be able to *identify* all *active* resource references
    * language must *mark* resource references so they can easily be identified on the heap
    * thus leads to an overhead when program must *pause* for garbage collection
    * can be mitigated with progressive background garbage collectors

Paging

- rather than separate memory into *variable* sized pieces in the **segmentation** approach, separate memory into smaller, *fixed* sized chunks called **pages** or **page frames**
  - 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)

- paging makes 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 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 a **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
  PTEAddre = 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**
  - with **RISC**, software handles the TLB miss
    - hardware raises an exception, and a 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 TLB handlers 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 is process specific
  - when process share physical pages, two TLB entries simply map to the same PFN
    - reduces physical pages in use and lowers overhead
- an 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 hiearchy** is the **disk**
    - 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 uses 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
        * when 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 water-**

mark 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 (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 a comparison policy
- **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 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 (no saved time field)
        * 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 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
    - to use the clock algorithm with per-process LRU:
        * hardware also needs to maintain the owning process of a page and accumulated CPU time of each process
    - almost as good performance as true LRU
- performance based on *workload*:
    - 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
- OS also has a policy to deal with **thrashing**, when memory is oversubscribed and constant paging occurs:
    - 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
    - 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

Working Sets

- 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 last few processes
    - *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*:
    - 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
        * thus, can *steal* 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
    - cannot add memory or reduce working sets
        * can only reduce number of competing processes by swapping some out, ie. swap all of its 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 **pre-load** the last working set instead of demand paging
            · much fewer *initial* page faults than pure demand paging

Concurrency

- processes should be used when:
  - programs should be distinct, and be isolated from interference or failures of another process
  - priveleged are distinct
  - creation/destruction are rare
  - *limited* interaction and resource sharing
- issue of just using 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 access memory *shared* with other threads
  - ie. has *more than one* point of execution for the program
    - but *share* the same address space and data
  - issue if these shared resources aren't *coordinated* between threads
    - can lead to **inndeterministic** results
  - multithreading *benefits*:
    - faster and cheaper to create and run, no need to allocate 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
  - multithreading *issues*:
    - up to the programmer to create and manage threads, and serialize resource use
  - OS must support primitives such as **locks** and **condition variables**
  - support for threads can be provided and scheduled directly by the kernel or by user-level libraries
- **thread** abstraction:
  - each thread needs its own private set of registers
  - one *independent* stack per thread, ie. **thread-local storage**
  - switching threads thus requires a context switch (save and retore 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
        * ie. when execution order of threads in parallel affects correctness of the program
        * race conditions lead to **nondeterministic** programs, where results depend on the timing execution
        * the piece of code where threads access a shared resource is a **critical section**
    - **atomicity**:
        * prevents overlap of operations
        * guarantees started operations *will* complete
    - eg. incrementing a variable is not **atomic**, 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:
    - 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**
        * would 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 disables greatly impact OS performance
  - many issues:
    - process may maliciously use locks to exploit CPU
    - fails with multiple CPUs
    - important interrupts and operations can be lost (eg. I/O completion)
    - requires a priveleged instruction
    - infinite loop bug
- 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
    * heavy performance overhead, especially with only one CPU, eg. scheduler only schedules blocked threads
    * spin-waiting is expensive, wastes processor cycles
- other useful atomic hardware primitives:
  - **compare-and-swap** only updates a value if it has an expected value
  - **load-linked** is a typically 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
- spin locks may be appropriate when:
  - critical sections are generally very short
  - there is not much contention for locks
- issues with spin locks:
  - thread may spin-wait until an interrupt goes off as it waits for a lock
  - **priority inversion** may occur where a higher priority, *scheduled* thread is stuck waiting for a lower priority, *unscheduled* thread to give up its lock
    * to solve, temporarily increase the lower priority thread holding the lock, ie. **priority inheritance**
- how to minimize spinning?
  - simply **yield** to the OS
    * this works well with fewer threads, but with more threads, spinning threads may just *continuously* yield to one another (round robin)
    * extra context switches
    * does not address *starvation* and fairness
  - instead use **queues** and sleeping:
    * a queue avoids starvation
    * threads go to sleep when the lock is already held, and woken up by the OS when it becomes free
    * using spin-waiting only *around* the lock itself, so the time spent spinning is limited to few lock and unlock related instructions
- **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;
}
```

Contention

- 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. course vs fine grained locks, pool vs element locks
    - but more finegrained locks leads to more overhead

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**, where each CPU maintains a *local* counter, and once a certain threshold on a local counter is met, a *global* counter is incremeneted 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

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
  - may use a **waiting list** with FIFO or priority order to choose threads to wake up
  - use `wait` and `signal` in UNIX, used in conjunction with a state variable and 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
- **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:

```
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;
}

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
  }
}
```

Semaphores

- 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
    - thus, intrinsictly incorporates 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`: release the lock, ie. 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);
}

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

```
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);
}
```

Common Concurrency Problems

- early work on concurrency focused on solving issues of **deadlock**
  - modern applications face more **non-deadlock** 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
  - 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
  - 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
- *preventing* deadlock:
  - to counter circular waits:

        * provide a **total ordering** on lock acquisition so that no cyclical wait occurs
        * **partial orderings** can still be used to structure lock acquisition when total ordering is unfeasible
    - 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
        * however, this approach is still hindered by encapsulation, and may decrease concurrency performance
    - 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
        * however, difficult to determine where process should *start over* from if the locking fails
            · not true preemption
    - to counter mutual exclusion:
        * 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
    - can also 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
    - the OS can keep track of free resources, and refuse to grant requests that would put the system into a dangerously resource-depleted state
        * eg. `sbrk`, `malloc` are all *failable* requests that allows the OS to avoid resource exhaustion deadlock
- *monitoring* deadlock:
    - 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**:
        * OS uses a combination of different monitoring methods
        * 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 responded 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

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, fd_set *errorfs, timeval *timeout)`
        * checks whether `nfds` file descriptors can be read to or written to
        * can use to build a *non-blocking* event loop

Example using `select`:

```
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
  - 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

- 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**
  - a device-specific **internal structure**
- a canonical **protocol** to use a device:
  - OS polls device until the status register is not busy
  - OS writes data to data register
    * AKA **programmed I/O (PIO)**
  - OS writes command to command register
  - 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** when its operation is completed
    * OS will call a corresponding **interrupt handler** that wakes awaiting process
    * can also **coalesce** multiple interrupts together to lower the overhead of interrupt processing
  - 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
        * *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 adevice
    - thus, OS uses a hierarchal file system:
        * high level applications are oblivious to any devices
        * **file system** abstraction makes *generic* requests to a **generic block interface**
        * requests are *routed* through a **specific block interface** to the appropriate device driver
    - however, a **raw interface** also allows special applications to directly read and write without using file abstraction
    - device driver code dominates OS code (70%)
    - abstraction of devices can lead to loss of *special* device capabilities

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 or **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 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 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, identical external interface
- **RAID Level 0** AKA **striping**:
  - **stripe** blocks across the disks of the system in a round-robin fashion
    * the **chunk size** if 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 performanceare **single-request latency** and **steady-state throughput**:
    * for throughput, can consider performance for **sequential** and **random** workloads
    * good latency, essentially redirecting to a single disk
    * best throughput for both sequential and random
- **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 the 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*
        * the issue with random writes is the **small-write problem**:
            - even though all the data disks can be accessed in parallel, the par-

        ity 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

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
  - 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**
- 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
    - 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

- `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
    - `fsync` 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**
- `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

Implementation

- many different file system implementations
  - **vsfs**, or "very simple file system" is a simplified UNIX file system
  - revolves around the **data structures** used and the **access methods**
- 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
- vsfs: 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 pointers limit the maximum size of the file
    * **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*
    * **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
  – **linked list** approach to inodes:
    * 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
- vsfs: 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
- vsfs: **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

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, fd_set *exceptfds, 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:

```
#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);
}
```