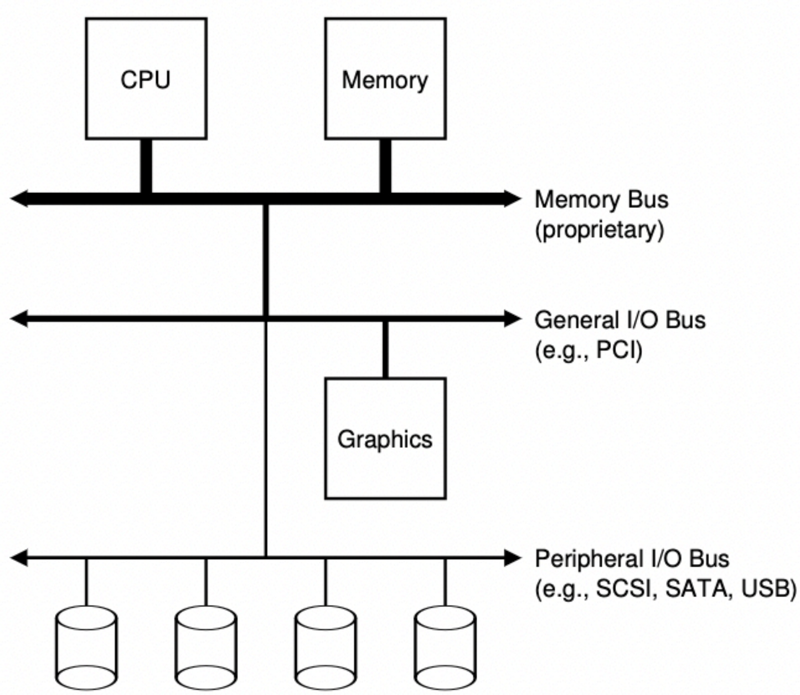
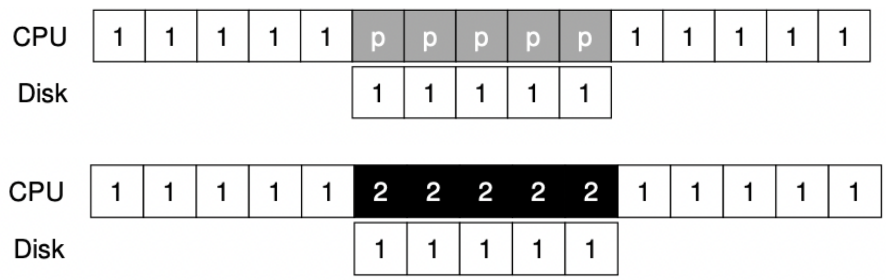
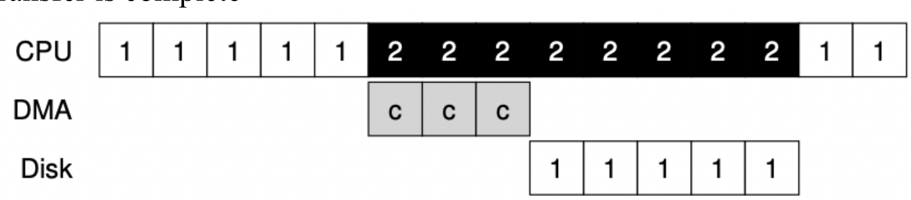
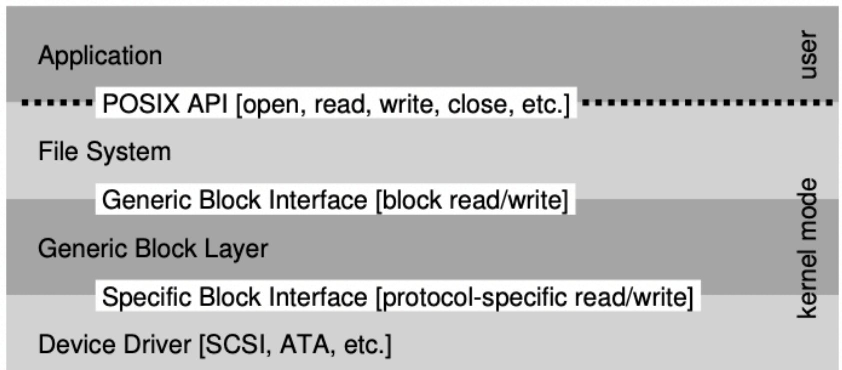
CS111	AD
Chapter 2	Intro to OS
Von Neuman model	Von Neuman model, the processor fetches an instruction, decodes it, and then executes the instruction.
Resource Manager	An OS is in charge of making sure the system operates correctly and efficiently in an easy to use manner.
Virtualizing the CPU	The illusion that a system has many virtual CPU's can turn a small number of physical CPU's into a seemingly infinite number of CPU's which then allows a programs to seemingly run at once is called virtualizing the CPU
Virtualization	Each process accesses its own private **virtual address space** which the OS maps onto the physical memory of the machine The OS manages memory such that one running program does not affect the memory address space of other processes. The program itself thinks that it has all the memory to itself.
Concurrency	A broad term to refer to a host of problems that arise when working on multiple problems or processes at once. In an example correlating to increment a counter by using two threads, for very high values of the counter, we see that the program does not work as expected. This is because the load, increment, and store instructions do not happen automatically (not all at once) the program does not work. Concurrency deals with such issues.
Persistence	Persistence is the idea that even when power goes away we need hardware and software to be able to store data despite this.
File System	The part of the OS that manage the disk is the file system, and it is responsible for storing any files that the user creates in a reliable and efficient manner.
Device Driver	A device driver is a piece of code in the OS that can deal with a specific device.
Journaling, Copy-on-write	The OS tends to bunch up write instructions for files since it becomes more effective, however to better deal with crashes there are intricate protocols called journaling or copy-on-write.

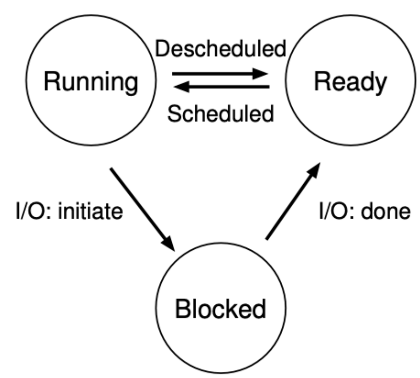
Goals of the OS	The main goals of OS systems are as follows: - Abstraction: make the system easy to use - Performance: minimize the overheads - Protection: between applications, OS and applications. - Isolation: isolating processes from one another is key to protection - Energy-Efficiency - Security - Mobility
History of OS	Initially, the OS was just a set of libraries of commonly used functions used by different programmers. It used a mode called <u>batch</u> processing where a number of jobs were set up and then run in bulk by an operator. Then protection was introduced. It was done by introducing the hardware <u>privilege level</u>, which means the hardware restricts can do, for example a user mode function cannot initiate I/O requests to the disk. When an I/O request is made, an instruction in the hardware called a <u>trap</u> which would raise the the privilege level to <u>kernel mode</u>. In kernel mode, the OS has full access to the hardware of the system. Then a <u>return-from-trap</u> instruction reverts to user <u>mode mode</u>. Then the idea of multiprogramming became more prevalent, which is where issues like concurrency and memory protection became important

Chapter 36	I/O Devices
System Architecture - Lower performance devices components are placed farther away eg. (IO devices etc) - Higher performance devices are generally placed closer to the CPU eg. graphics, memory etc.	

Canonical Device	Hardware interface is something that allows the system's software to control its operation. The second part of any device is its internal structure, which implements the abstraction the device presents to the system. Firmware is the software within a hardware device that implements it's functionality Polling a device essentially means, repeatedly reading the status register, to understand what the state of the machine is. (Programmed I/O PIO)
How can the OS check device status without frequent polling?	When an I/O call is made, the CPU may have to sit idle while not doing anything. To improve this we could use an <u>interrupt handler</u> and interrupt service routine, which allows for overlap with computation and I/O 
Balancing Interrupts, and PIO	Suppose that a device is very fast then what results is that the cost of receiving of an interrupt, handling it, and then switching back to the issuing process may actually end up taking up more time Therefore depending on the device, a hybrid that polls for a little while, and if the device is not yet finished, uses interrupts.(Two phase approach)
Livelock	When a huge stream of incoming packets each generate an interrupt, it is possible for the OS to only end up processing interrupts and then never allowing the user-level processes to run.
Coalescing	In this setup a device waits before raising an interrupt, which while it waits allows other requests to complete, and allows for multiple interrupts to be merged into one.
DMA	Direct Memory Access (DMA) is a specific device that orchestrates transfers between devices and memory without CPU intervention. The OS would first tell the DMA where the data is in memory, and then how much to copy, and which device to send it to. When the DAM is complete the DMA raises an interrupt indicated to the OS that the transfer is complete 

How should hardware communicate with a device? Making a device neutral OS Notes for 36.8 and 36.9 not included.	First method is to have explicit I/O instructions; these instructions specify a way for the OS to send data to a specific device. Such instructions are considered privileged, meaning only the OS can use them. The second method is memory mapped I/O. The hardware makes device registers as though they were actually memory locations. So instead the OS issues a load or store instruction to that specific address.  The issue of making a device neutral OS is that device drivers with specialized features may end up being unused since the kernel and OS would require a generalized interface from the driver

Chapter 4	Abstraction of Processes
Process Virtualization, and time sharing Space Sharing Context Switch Mechanism Vs Policy	A process is just the running of a program, the OS manages to execute the bytes and get them running. Gives the illusion of having many CPUs when in reality there is only one CPU. Through time sharing of the CPU, the OS allows users to run as many concurrent processes as they would like ⇒ however the cost is performance since as there are more concurrent processes, the more slowly the CPU will be shared Where a resource is divided (in space) among those who use it. For example a file takes up part of disk storage which is a naturally space shared resource This enables the OS to stop running one program and start running another different program on a given CPU. Mechanism deals with the “how”, policy deals with “Which” question to answer. Having this difference is a form of modularity that allows policies to change without a changing mechanism.

Machine State of a process:	The machine state is what a program can read or update when it is running. ● Memory: the address space of a process, that part of memory that a program can read and write to. ● The instruction pointer, stack pointer, and frame pointer, help tell which instruction should be executed next. And manage the environment the code should get executed in (local variables, function calls etc). ● I/O information files, or external hardware devices
Process API	Create: An OS must have some method to create new processes. Destroy: halt or kill a processes that is runaway (some way to destroy a process forcefully) Wait: Waiting for a process to finish it's execution Miscellaneous Control: Give the ability to pause a process for sometime and then resume running at a later time Status: get information about the process. Ex how long it has run for, or what state it is in.
Process Creation:	The OS will first load code into the address space from the disk so that process can execute its instructions. ● Eager loading: all loading is done at once before running the program ● Lazily: loaded as and when a program needs, uses paging and swapping to properly work The OS will then load and allocate memory for the program's run-time stack, the program's heap. ❖ The OS also will have the ability to give more memory to heap, if a program calls malloc() Finally the OS transfers control the program, by jumping to the main() routine.
Process States:	Running: The processing is executing instructions Ready: the process is ready to run, but the OS is not letting it execute any instructions. Blocked: the process is not ready to run, until some other event take places 

How does the OS keep track of processes?	The OS has multiple data structures that would allow it to store information about each process.
	
	`// the registers xv6 will save and restore`
	`// to stop and subsequently restart a process`
	`struct context {`
	`  int eip;`
	`  int esp;`
	`  int ebx;`
	`  int ecx;`
	`  int edx;`
	`  int esi;`
	`  int edi;`
	`  int ebp;`
	`};`
	
	`// the different states a process can be in`
	`enum proc_state { UNUSED, EMBRYO, SLEEPING,`
	`                  RUNNABLE, RUNNING, ZOMBIE };`
	
	`// the information xv6 tracks about each process`
	`// including its register context and state`
	`struct proc {`
	`  char *mem;                 // Start of process memory`
	`  uint sz;                   // Size of process memory`
	`  char *kstack;              // Bottom of kernel stack`
	`                             // for this process`
	`  enum proc_state state;     // Process state`
	`  int pid;                   // Process ID`
	`  struct proc *parent;       // Parent process`
	`  void *chan;                // If non-zero, sleeping on chan`
	`  int killed;                // If non-zero, have been killed`
	`  struct file *ofile[NOFILE]; // Open files`
	`  struct inode *cwd;         // Current directory`
	`  struct context context;    // Switch here to run process`
	`  struct trapframe *tf;      // Trap frame for the`
	`                             // current interrupt`
	`};`
	
	In this example, we also see the context, where for each process the OS would store the contents of the registers if a process needed to be stopped, and started at a later point ⇒ this helps enable context switching
	
	The OS will contain a process list (a list of all the processes) and each process will have a Process Control Block (PCB) which is a C structure that contains information about each process.

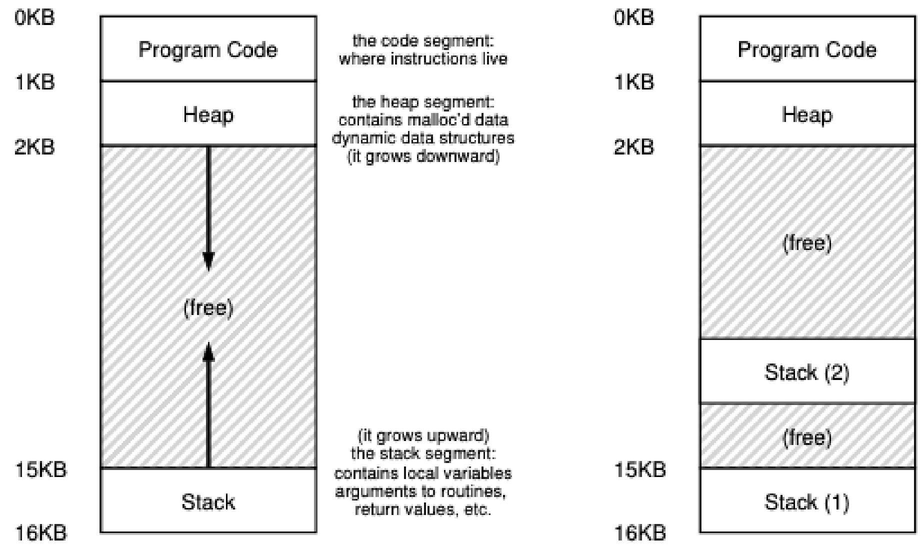
Chapter 6	Limited Direction Execution
Limited Direction Execution	Direct Execution implies that the program runs natively on the CPU, and then transfers control back to the OS. However this brings up two issues, how do we stop the program from doing things we don't want and secondarily how do we implement time sharing.

Problem #1 Creating Restricted Operations	When running a program, we may not want to limit its access for security and efficiency, so how can we do this?
Protected Control Transfer:	The hardware helps create different modes (kernel and user mode) and provides trap, and return from trap instructions to change between the kernel and the user mode programs.
System Call	When a user wants to execute an instruction that is in kernel mode (such as I/O) , they make a system call
Traps/ and Switching	To execute a system call, a trap instruction is executed, which jumps into the kernel and raises the privilege level. To store the user program states, the counter, flags and other information will be pushed onto the kernel stack
Return from Trap	After the kernel has finished running the special instructions, the return from trap pops the above values off the stack which resumes execution of the user systems.
Trap Table	To prevent the user from executing its own code instead of what the trap should do, the kernel sets up a trap table. The trap table tells what code the hardware should execute when certain trap events occur. The kernel then gives the location of these trap handlers to the hardware.
Problem #2 How the OS regains control	A cooperative process: whenever a process calls a system call, (which contains a yield ⇒ specifically for just transferring control) or if a process does something illegal, control will be automatically transferred ex divide by zero
Timer interrupt	An uncooperative process: uses a timer interrupt, every few milliseconds, an interrupt is raised, and a process is halted and the OS runs. (However this much more of a hardware feature not a software one) the OS cannot do anything from a software side without this hardware implementation.

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

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

Chapter 7	Scheduling: Introduction
Workload:	Simplifying Assumptions about the process running in the system
Turnaround time	The time at which the job completes minus the time at which the job arrived: Turnaround $= T_{completion} - T_{arrival}$
First In, First Out (FIFO)	The first task that comes in, will be executed until it is completed and so on and so on. Disadvantage: convoy effect, where a number of much shorter potential processes end up getting queued behind a process that will run for much longer.
Shortest Job First:	The shortest job runs first, this works well under the assumption that all processes enter at the same time. Disadvantage: convoy effect, when processes don't arrive at the same time.
Shortest Time-to-Completion First:	Every time a new process enters the queue it determines which task has the smallest time to completion, and then schedules that. Disadvantage: horrendous response time and very bad for interactivity, when people want to see their results as soon as possible.
Round Robin (RR):	RR runs each jon for a time slice (a quantum) and then switches to the next job in the run queue. However an important note is that the time slice must be a multiple of the timer-interrupt period. RR is a fair policy since it evenly splits up the CPU Disadvantage: RR is horrendous under the metric of turnaround time
Section 7.9 TBD	

Chapter 26	Concurrency: Introduction
Thread	Can be thought of as a process, but threads generally share the same address space, and therefore can access the same data. Switching between threads requires context switches, and we store each thread's information in thread control blocks.
Thread-Local	The local variables of a thread will be in it’s own stack: 
Execution of Threads:	When two threads are created we do not know their execution order, since it can be dependent on the scheduler.
Critical section:	A piece of code that accesses a shared resource (file, variable, etc) and must not be concurrently executed by more than one thread.
Race Condition:	When multiple threads of execution enter the critical section at roughly the same time and both attempt to update a shared data structure.
Mutual Exclusion:	Mutual exclusion, if one thread is running within the critical section, other threads will be prevented from doing so.
Atomically executed:	The entire instruction is executed, or none of the instruction is executed, if we can have this apply to multiple lines of code, our issue of race conditions would be solved.
Indeterminate Execution	Having multiple race conditions changes and causes the program to become indeterminate, where the output will vary from run to run.

Chapter 27	Concurrency: Thread API
Thread creation	Int pthread_create(pthread_t * thread, attr, void * (*start_routine)(void*), void * args); The first argument, is the thread data structure to store information about the thread, the second argument are about the attributes of the thread, the *star routine, is a function pointer to the function that the thread should start executing, and the final parameter is the arguments given to a thread.
Waiting for a thread to complete:	Int pthread_join(pthread * thread, void* return), one argument to specify which thread to wait for, and a second argument to see the return value.
Return values from a thread:	We cannot return a pointer to something that has been allocated on the threads stack, since that would end up returning a pointer to a memory location that is now being used for something else once the thread finishes executing.

Mutex Locks	We can add a mutex lock around a critical section to indicate that only one thread can execute it at a time.
	
	`pthread_mutex_t lock;`
	`Int rc = pthread_mutex_init(&lock, NULL);`
	`pthread_mutex_lock(&lock);`
	`assert(rc == 0); //If success`
	`x = x + 1 //Critical Section here`
	`Ptheard_mutex_unlock(&lock)`

Condition Variables	Condition variables help with signaling between multiple threads, we can have one thread be “sleeping” until a certain condition is met, and when another thread changes the condition, that thread can wake up this other thread.
	
	We must hold the lock, until after we wake up the thread in-order to prevent any race conditions.
	
	To execute a program with multiple threads under gcc we use the #include<pthreads.h> header
	Sleeping Process:
	`pthread_mutex_t lock = PTHREAD_MUTEX_INITALIZER;`
	`Pthread_cond_t init = PTHREAD_COND_INITALIZE;`
	`//Lock the lock while(initialized == 0)`
	`        pthread_cond_wait(&init, &lock)`
	Waking Process:
	`pthread_cond_signal(&init)`

Chapter 28	Concurrency: Locks
Locks	The lock() attempts to acquire the lock, if there is somebody else holding onto the lock, then this attempt will fail. The unlock() will release the lock, so that somebody else may acquire the lock.
Coarse versus Fine grained	In a coarse-grained approach we will protect a critical section by using one large lock, in a fine-grained approach we will have different locks protecting different data and different data structures, thereby increasing concurrency.
Turning off Interrupts	The first approach for mutual exclusion was to disable interrupts for critical sections, thereby ensuring that the code will be executed fully. Negatives: we have to give a program a privileged operation, which can allow a program to monopolize a system and thereby run forever. Lastly, disabling and enabling can significantly slow down a system.
Test-and-set	An atomic instruction that enables us to test a value by reutring it, while also changing it to a different new value
Spin Lock:	Uses the test and set function to implement a lock, and will have a thread just spin until it can acquire the lock. However spin locks are not fair, and can lead to starvation since a waiting thread may never get the lock
Compare and Swap:	Similar to a spin lock, however, instead of always setting the value to 1, we check whether the lock is free and then set the value to 1.
Load and Store	Load the value and check if it is free, if so, attempt to store the value with a conditional check to see if it has already been acquired.
Fetch and Add, Ticket Lock	Helps implement a ticket lock, whenever a thread wants to access a lock it will increment a global counter, and store it as it's turn, when it reaches that number, it gets to use the resource. Ticket locks, still have spinning, but they prevent starvation.
Solution#1 to Spinning: Yield	Instead of having a process just constantly be in a while loop spinning, within the while loop we will yield the CPU to a different process. While this does help reduce wasted time, it also requires a lot of context switching.
Lock with Queues:	We will use a queue, to hold onto a lock, and this way we have control over who next gets the lock
Two Phase Locks:	First phase, spin for a bit, second phase yield, and put caller to sleep

<table>
<tr><td>Chapter 30</td><td>Concurrency: Condition Variables</td></tr>
<tr><td>Condition Variables</td><td>Allows each thread to wait for a certain condition to be met, before it continues its execution. However, spinning and waiting for execution is a gross waste of time.

We can wake one or more threads that are waiting by using a signal

Whenever we would like to call a signal or wait we would need to hold onto the lock, in-order to avoid race conditions.</td></tr>
<tr><td>Product/Consumer Problem</td><td><pre>
cond_t empty, fill;
mutex_t mutex;

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

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);   // c1
        while (count == 0)   // c2
            Pthread_cond_wait(&fill, &mutex); // c3
        int tmp = get();   // c4
        Pthread_cond_signal(&empty);   // c5
        Pthread_mutex_unlock(&mutex);   // c6
        printf("%d\n", tmp);
    }
}
</pre>
A producer will only sleep, if all buffers are filled, and a consumer will sleep only if all the buffers are empty.</td></tr>
<tr><td>Covering Condition: Lampson and Redell Memory allocation</td><td>A condition where all cases where a thread needs to be woken up are covered. Take the Lampson and Redell example about memory allocation, the code may not know what thread to wake up at a certain point, and hence the best course of action would be to wake up all sleeping threads.</td></tr>
</table>

Chapter 31	Concurrency: Semaphores

Initializing A Semaphore

```
#include <semaphore.h>
sem_t s;
sem_init(&s, 0, 1);
```

The third argument will initialize the value of the semaphore to that value, and the second argument value of 0 is used to indicate that the semaphore is shared between threads, in the same process.

Value of a Semaphore

A negative value for a semaphore indicates the number of threads that are waiting.

```
int sme_wait(sem_t *s) {
     Decrement the value of semaphore s by one
     Wait if the value of semaphore s is negative
}

int sme_post(sem_t *s) {
     Increment the value of semaphore s by one
     If there are more thread waiting wake one
}
```

Binary Semaphore (Lock)

We give our semaphore an initial value of 1 to make in a binary semaphore. To try and acquire the lock we will use sem_wait(), and then to release the lock we will use sem_post()

Condition Variable

We give our semaphore an initial value of 0. The parent (or the waiting process) will call sem_wait(), while the child process or thread (the signaling process) will call sem_post()

Bounded Buffer Problem

```
sem_t empty;
sem_t full;
sem_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&empty);           // line p1
        sem_wait(&mutex);           // line p1.5 (MOVED MUTEX HERE...)
        put(i);                     // line p2
        sem_post(&mutex);           // line p2.5 (... AND HERE)
        sem_post(&full);            // line p3
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        sem_wait(&full);            // line c1
        sem_wait(&mutex);           // line c1.5 (MOVED MUTEX HERE...)
        int tmp = get();            // line c2
        sem_post(&mutex);           // line c2.5 (... AND HERE)
        sem_post(&empty);           // line c3
        printf("%d\n", tmp);
    }
}

int main(int argc, char *argv[]) {
    // ...
    sem_init(&empty, 0, MAX); // MAX buffers are empty to begin with...
    sem_init(&full, 0, 0);    // ... and 0 are full
    sem_init(&mutex, 0, 1);   // mutex=1 because it is a lock
    // ...
}
```

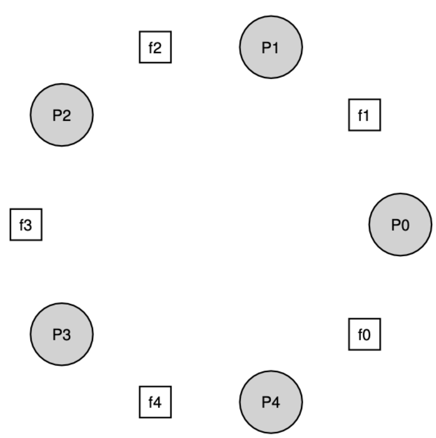
Read Write locks

Allow as many readers to read the data without modification without letting any writers in. Then when a writer enters, there can be only one person writing to a data structure at a time.

```c
typedef struct _rwlock_t {
  sem_t lock;      // binary semaphore (basic lock)
  sem_t writelock; // used to allow ONE writer or MANY readers
  int   readers;   // count of readers reading in critical section
} 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)
    sem_wait(&rw->writelock); // first reader acquires writelock
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0)
    sem_post(&rw->writelock); // last reader releases 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);
}
```

Dining Philosophers

If each philosopher were to grab the fork on the left, none of them would have the ability to eat since they would each have exactly one fork. TO AVOID THIS DEADLOCK WE SIMPLY NEED TO CHANGE ONE PHILOSOPHER to try and grab a fork on the right first.



Implementing A Zempahore

```c
typedef struct __Zem_t {
    int value;
    pthread_cond_t cond;
    pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
    s->value = value;
    Cond_init(&s->cond);
    Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
    Mutex_lock(&s->lock);
    while (s->value <= 0)
        Cond_wait(&s->cond, &s->lock);
    s->value--;
    Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
    Mutex_lock(&s->lock);
    s->value++;
    Cond_signal(&s->cond);
    Mutex_unlock(&s->lock);
}
```

A Zemaphore is almost exactly same to a semaphore, however a zemapahore does not maintain the invariant that if the value is negative it will represent the number of waiting processes. In this case it will always be positive

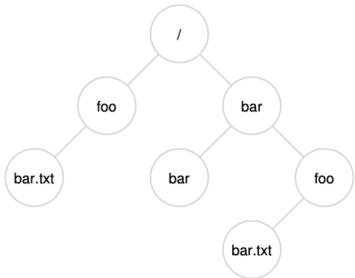
Signals

```c
#include <signal.h>
void handle(int arg) {
   printf("Handle the signal")
}

int main(){
   signal(SIGHUP, handle); //set up the signal
   ...
}
```

Whenever our program catches a signal it will stop, call the handle function, and then once again resume execution.

Chapter 32	Concurrency: Common Concurrency Problems
Non Deadlock bugs	Bugs in concurrency programs that don't actually have the system end up in deadlock.
Atomicity Violation	When we memory access and modify a shared location, atomicity is not enforced. This can be fixed by adding locks around the respective critical sections.
Order Violation Bugs	The memory access order of two groups is flipped, this can be easily fixed by adding a conditional variable to indicate when the status has changed.
Conditions for Deadlock (all four must occur)	Mutual Exclusion: Threads claim exclusive control of resources that they require Hold and Wait: Threads hold resources allocated to them, while waiting for additional resources No preemption: Resources cannot be forcibly removed from threads that are holding them. Circular Wait: There exists a circular chain of threads such that each thread holds one more resources that are being requested by the next thread in the chain.
Solving Deadlock:	Avoid circular wait: by making sure that locks can only be acquired in a very certain order. Avoid hold and wait: acquire all locks at once atomically. No preemption: we can use a trylock, which will tell use if a lock is available, and if it not release our current resource as well, however this can lead to an issue of livelock The final option is to avoid deadlock via scheduling, however this would require the scheduler to know what threads would need which resources.

Chapter 39	File and Directories
File	A file is simply a linear array of bytes, each of which has the read and write permissions. Each file has a "name" which is called the inode number, which is a number to identify the file that the user cannot see
Directory	A directory also has an inode number, and its contents are a list of (file_name, inode number). The file_name is what the user can see.



We can place directories within directories to create create a <u>Directory Tree </u>or a <u>Directory Hierarchy</u>

Directories and files can have the same name as long as they are in different locations.

Directories start at the root directory: “/”. Each file has an absolute pathname.

Creating Files:

```
int fd = open(“foo”, O_CREAT | O_WRONLY | O_TRUNC);
```

open returns a file descriptor, if the file foo does not exist, it will create it with the flag O_CREAT, and the O_WRONLY can only be written too. If the file already exists the O_TRUNC states to set the newly created file to have zero bytes

Reading a file:

```
open(“foo”, O_RDONLY | O_LARGEFILE);
```

Writing a file:

We first open a file using `open` then we use the `write()` call to actually write our data to a file.

Non-sequential read and write

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

this simply changes a variable in the OS kernel to tell where to start the reading file from (this is not same as the seek done by a hard disk.

Writing immediately to files

```
int fd = open(“foo”, O_CREAT | O_WRONLY | O_TRUNC);
int rc = write(fd, buffer, size);
assert (rc == size);
rc = fsync(fd);
assert(rc == 0);
```

In this case fsync forces all bytes to be written by the write call before moving on.

Renaming Files:

We can use the mv command which then calls the rename function.

```
int fd = open(“foo.txt.tmp”, O_CREAT | O_WRONLY |
O_TRUNC);
write(fd, buffer, size);
fsync(fd);
close(fd);
rename(“foo.txt.tmp”, “foo.txt”);
```

The rename operation is atomic, so we don't lose a file during a system crash

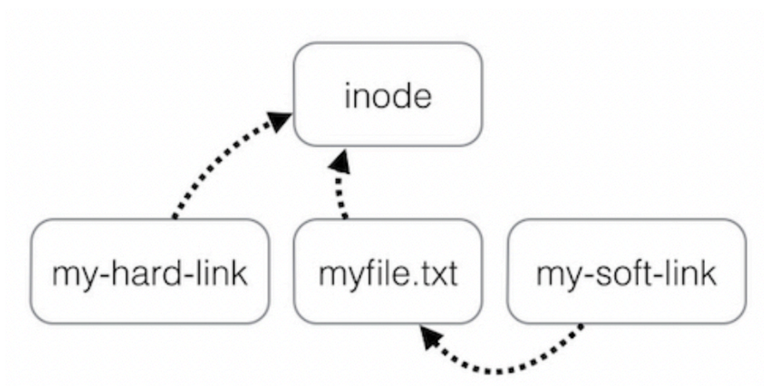
Metadata

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

Inode	An Inode is a persistent data structure kept by the file system that has information about all each of the files.
Removing files	`unlink("foo")` there is a reference count (aka link count) that will count the number of hard links to a file. Only when the reference count is zero does the OS free the data and inode at the location.
Making/Delete a Directory	`mkdir("foo"), rmdir()`

Reading Directory:	`opendir(), readdir(), closedir()`
	`DIR *dp = opendir(".");`
	`struct dirent *d;`
	`while ((d = readdir(dp)) != NULL)`
	`   printf("%d %s\n", (int) d->d_ino, d->d_name)`
	`closedir(dp)`

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

Symbolic and Hard Links	 If a file gets deleted and there is a symbolic link to it, it becomes a dangling reference. Symlinks can "point" to a directory, hard links cannot go to a directory
Mounting a file system	The mkfs command creates and writes an empty file system starting with a root directory onto that specific disk partition Mount: it takes an existing directory as a target mount point, and pastes a new file system onto the directory tree at that point.