Operating System

A (large-scale) program that allocates and manages resources for other programs
OPERATING SYSTEM == RESOURCE MANAGER

Memory Stuff

- Declaring a float without setting it to anything might lead to the float being something unintended (NOT ALWAYS ZERO)
- All pointers are 8 bytes

Anything not declared with a “new” keyword is ON THE STACK

Datatype Sizes:

i. Use `sizeof` to find the size

- Char: 1 byte
- Short: 2 bytes
- Int: 4 bytes
- Float: 4 bytes
- Long: 8 bytes
- Double: 8 bytes
- Pointer (any): 8 bytes

Strings will not end if there is no *zero byte* at the end of the string then it will keep printing any string in the following memory.

ex)

```
int main() {
    char name[5] = "David";
    char xyz[5] = "xbcne";
    printf("hi again %s", name);
    printf("xyz is %s", xyz);

    return 0;
}
```

Will print out: ”hi again Davidxbcnexyz is xbcne” because there is no end-line character included in the string

However, char* cptr = “KDJHFJFKHDSJHFKDSDFS” will automatically include a terminating char *Note: trying to assign something to this pointer will result in an immediate exit WITH NO ERROR as it is in static memory and therefore can not be changed.* ex) cptr[3] = ‘A’

File Descriptor:

- A small, non-negative integer used in a variety of system calls to refer to an open file (AKA file stream/byte stream)
- Each process has a file descriptor table associated with it that keeps track of it’s open files

ex)

Dynamic Allocation:

For most C functions/features, see my CSCI 2500 (COMPORG) notes but here’s a short demo below

```c
#define O_RDONLY 00;
#define O_WRONLY 01;
#define O_RDWR 02;

int staticAlloc() {
    char fname = "file.txt";
    int fd = open(fname);
    if (fd == 1) {
        perror("open() failed!");
        return EXIT_FAILURE;
    }

    printf("fd is %d\n", fd);
    /*
     * 0 stdin
     * 1 stdout
     * 2 stderr
     * 3 file.txt (O_RDONLY)
     */
```

```
     char buffer[20];
     int rc = read(fd, buffer, 11);
     buffer[rc] = '\0';
     printf("read() returned %d -- read \"%s\"\n", rc, buffer);

    return EXIT_SUCCESS;
}

int dynamicAlloc() {
    char* path = malloc(16);
    if (path == NULL) {
        perror("malloc() failed!");
        return EXIT_FAILURE;
    }
    printf("sizeof path is %lu\n", sizeof(path));

    strcpy(path, "/cs/[REDACTED]/u23");
    printf("length path is %lu\n", strlen(path));

    return EXIT_SUCCESS;
}
```

`ptr = (type*)realloc(ptr, numElements*sizeof(type));` resizes the memory for `ptr`

Valgrind:

- Valgrind
- Valgrind Documentation

Read in args Data:

```
for (int i = 0; *(argv + i) != NULL; i++) {
    //Do smth with the args
}
```

Note that this moves the pointer one over, which is equivalent to EIGHT bytes

Nested arrays:

```
    const int ll = 47;
    char** names = calloc(ll, sizeof(char*));
```

```
    for (int i = 0; i < ll; i++) {
        *(names + i) = calloc(20, sizeof(char));
    }
```

Header Files:

Use `ifndef` and `endif`

Standard Out and Error

If you redirect the buffer to a file (using `./a.out > STDOUT.txt`), the '\n' will no longer flush the buffer

When printing to the terminal (shell) via stdout (fd 1), a '\n' character will "flush" the stdout buffer, i.e., output everything that has been stored in the stdout buffer so far...

==> this is called line-based buffering

TO DO (fix): add a '\n' to the end of each debugging `printf()`

When we instead output fd 1 to a file use `./a.out > STDOUT.txt`

...the '\n' character no longer flushes the stdout buffer

==> this is called block-based buffering (also referred to as fully buffered)

Another approach is to use `fflush(stdout)` or `fflush(NULL)`

A third type of buffering is non-buffered (unbuffered), which is what is used for stderr (fd 2)

`fprintf( stderr, "crashing here?" );`

Missing Lecture Notes for 6/5/23

```
#include <fcntl.h>
#include <sys/types.h>
#include <sys/wait.h>
```

`fork()`

- If the `fork()` function returns a value greater than 0, it means that the calling process is the parent process, and the returned value is the process ID of the newly created child process.
- If the `fork()` function returns 0, it means that the calling process is the child process.
- If the `fork()` function returns -1, it indicates that an error occurred during the fork operation, and no child process was created.

After the `fork()` function is called, both the parent and child processes continue their execution independently. However, there are a few important differences between them:

- ➔ The child process has its own copy of the parent's memory space, which is initially identical to the parent's. However, any changes made to the memory in one process do not affect the other process.
- ➔ The child process gets a new process ID (PID) assigned to it.
- ➔ The child process inherits open file descriptors from the parent, such as open files or network connections.
- ➔ Child processes usually continue executing the same code as the parent, starting from the same point where the `fork()` function was called. However, it's possible to differentiate the behavior of the parent and child processes by examining the return value of `fork()`.

RLIMIT:

???????????

```
struct rlimit rl;
RLIMIT_NPROC:
getPID():
getrlimit(RLIMIT_NPROC, &rl):
```

`usleep(int i)`: puts the process to sleep for i seconds

`pid_t waitpid(pid_t pid, int *status, int options)`: wait for the specified process to hit whatever options is (defaults to 0):

- `WNOHANG`: Return immediately if no child has exited.
- `WUNTRACED`: Also return if a child has stopped. Status for traced children which have stopped is provided even if this option is not specified.
- `WCONTINUED`: Also return if a stopped child has been resumed by delivery of `SIGCONT`.

Defunct Process (defunct process): a process that has completed its execution but still has an entry in the process table AKA it terminated, but its parent process has not yet received its exit status

`WIFSIGNALED(status)`: check if the child process was terminated by a signal

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

int main(int argc, char const *argv[]) {
   // create the fork
   pid_t p = fork();

   if (p == -1) {
       perror("fork() failed");
       return EXIT_FAILURE;
   }

   // This is the child process
   if (p == 0) {
       return EXIT_SUCCESS;
   }
   else if (p > 0) {
       // This is the parent process

       /* block the parent until the child is done */
       int status;
       pid_t pid = waitpid(p, &status, 0);
       /* Check to see if pid == -1 here */
       if (WIFEXITED(status)) {
           int estat = WEXITSTATUS(status);
           printf("Child exited with code %d\n", estat);
       } else {
           perror("something went wrong with the child process!");
       }
       return 0;
   }
}
```

Pipe:

- A unidirectional communication channel
- Like a file that's used to communicate between processes (but instead it's two buffers)

Note: if you try to read more from the pipe than is currently there, the program will hang until there is *enough data to finish the read*

```
int pipefd[2];
int rc = pipe(pipefd);
if (rc == -1) {
    perror("pipe() failed!");
    return EXIT_FAILURE;
}

// Write the bytes to the pipe
int bytes_written = write(pipefd[1], "ABCDEFGHIJKLMNOPQRSTUVWXYZ", 26);
printf("wrote %d bytes to a file!\n", bytes_written);

char buffer[20];
int bytes_read = read(pipefd[0], buffer, 10);

// assume the data is char data
buffer[bytes_read] = '\0';
printf("read %d bytes: \"%s\"\n", bytes_read, buffer);

// close the write buffer
close(pipefd[1]);

// close the read buffer
close(pipefd[0]);
```

- Closing the writing pipe will force the number of write descriptors to 0, and will stop the pipe from hanging in the above case if you do it before you read
- Once both the read and write pipes are closed, then the OS automatically gets rid of the pipe buffers (**NOT** the buffers that you might read the data in the pipe into)

★ This pipe can be accessed from other processes

Note: if there is a read open on both the parent and child processes, then there might be read-locks. To avoid this, create on pipe that the child writes to and the parent reads, then one pipe that the parent writes to and the child reads

- ★ **For more info, see pipewithforks.c**

If you want to duplicate a file descriptor, use the `dup()` function like so:

```c
int fd = open("input.txt", O_RDONLY);
if (fd == -1) {
    perror("open() failed!");
    return EXIT_FAILURE;
}

// Duplicate the file descriptor
int newfd = dup(fd);
if (newfd == -1) {
    perror("dup() failed!");
    return EXIT_FAILURE;
}
```

CPU Scheduling

A process is a "running program" or "program in execution"

Process States:

- RUNNING STATE: process is actually using the CPU, i.e., executing its instructions
- READY STATE: process is ready to use the CPU, i.e., process is idle in the ready queue
- WAITING STATE: process is waiting for I/O operation(s) to complete

bursts

- a CPU burst is a set of assembly/machine instructions that are executed by the CPU for a given process, e.g., P22
- an I/O burst is one or more I/O operations for a given process

CPU Scheduling (a.k.a. Short-Term Scheduling)

- ➔ The scheduling system enables one process to use the CPU while the other processes are waiting in the ready queue to use the CPU (or waiting in the I/O Subsystem).
- ➔ The goals of CPU Scheduling are to make efficient use of the CPU and to minimize the turnaround and wait times for each process.
- ➔ We also want to achieve "fairness" across all processes

Multiprogramming

- Multiple processes reside in memory at the same time
- CPU is shared and managed by the OS
- Addresses the problem of the U being under-utilized
- New problem, we now need to perform “context switches” to switch the CPU’s context from one process to another
- Processes in a multiprogramming compete for the CPU, but they also often need to cooperate with one another via IPC
- ★ A CPU-bound process is one that will primarily have longer CPU bursts (spending most of its time either running with the CPU or waiting in the ready queue until the CPU becomes available)...a.k.a. compute-bound process
- ★ An I/O-bound process is one that will primarily have shorter CPU bursts (spending most of its time waiting on I/O operations to complete)...a.k.a. interactive process

Degree of Multiprogramming

- Given n processes in memory, (i.e. the degree of multiprogramming is n), then the probability that all n processes are waiting for I/O is p^n
- CPU utilization is then 1 - p^n

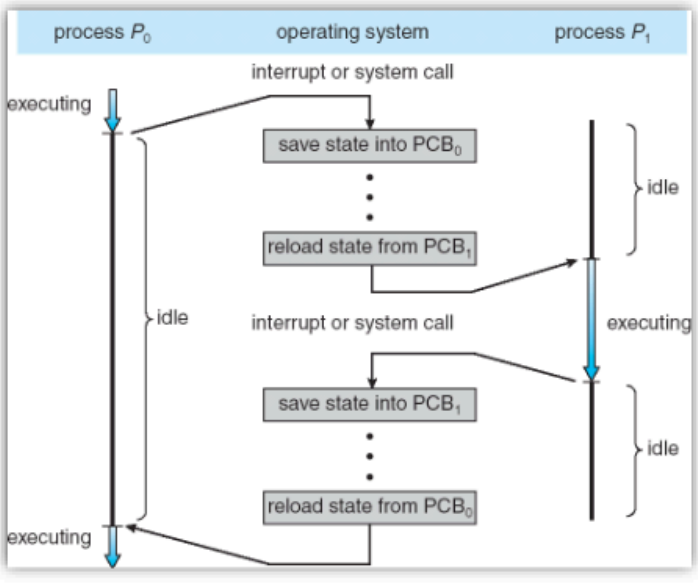
CONTEXT SWITCH

A context switch occurs each time the operating system switches its *context* from one process to another

The operating system maintains a Process Control Block (PCB) for each process, regardless of process state

PCBs contain register values, program counter, file descriptor table, page tables, etc.

$t_{context\text{-}switch} << t_{cpu\text{-}burst}$



Preemption:

- When the currently running process is kicked out while using the CPU
- This might be because a newly arriving (more important) process has switched to the ready state or a timeout
- ★ A non-preemptive algorithm implies that when a process is using the CPU, it will use the CPU as long as necessary to complete its CPU burst

Four Scenarios

- running process "decides" that it is done with its CPU burst (e.g., read() or waitpid() or etc.)
- running process "decides" to terminate
- running process is interrupted (goes back to the ready state/queue)
- preemption? running process is not done with its current CPU burst...another process enters the ready state and might preempt

Prioritizing processes

- batch: no users are waiting (lower priority --- non-preemptive)
- interactive: users are waiting for a (quick) response; also servers serving up files/webpages/etc. (higher priority --- preemptive)
- real-time: preemption is not usually necessary because processes already are designed to "know" that they need to run quickly

CPU Scheduling Criteria and Measures

- CPU utilization (busy versus idle CPU time)
- Throughput
- Fairness
- Arrival and departure rates
- Response times (minimize variance of response times)
- Wait times
- Turnaround times

Note that for each CPU burst per each process

WAIT TIME: How much time does a process spend in the ready queue, waiting for time with the CPU? (Note that the process here is in the READY state.)

TURNAROUND TIME: How much time is required for a process to complete its CPU burst, from the time it first enters the ready queue (i.e., the READY state) through to when it completes its CPU burst?

TURNAROUND TIME = WAIT TIME + CPU BURST TIME + OVERHEAD (context switches)

Scheduling Algorithms [LINK]

First-Come-First-Served (FCFS)

When a process arrives or transitions to the ready state, we simply add it to the end of the READY queue

- **advantages:** very simple; easy to implement; very low overhead
- **disadvantages:** processes with larger CPU burst times will cause longer delays for other processes (e.g., CPU-bound versus I/O-bound process mix)

Shortest Job First (SJF)

processes with the smallest execution time execute first

Advantages:

1. lower average wait/turnaround times (versus FCFS)
2. good low turnaround times (and therefore low response times) for interactive or I/O-bound processes

Disadvantages:

1. processes with larger CPU burst times might end up facing INDEFINITE BLOCKING (or STARVATION)
2. increased overhead due to the priority queue (log time)
3. we have no way of knowing ahead of time exactly what CPU burst times will be for each process (we can only predict these CPU burst times...)

STARVATION: a process faces starvation if we know that the process will NEVER complete its CPU burst

INDEFINITE BLOCKING: a process faces indefinite blocking if (or POSTPONEMENT) the time to complete its CPU burst is (likely) comparatively long…

Both FCFS and SJF are non-preemptive algorithms (AKA once a process starts using the CPU for its CPU burst, it will continue uninterrupted until the burst is complete)

Shortest Remaining Time (SRT) - SJF with preemption

- when process B arrives in the ready queue, it can potentially preempt and replace the currently running process A iff B's CPU burst time is less than the remaining CPU burst time for process A

Advantages:

1. lower average wait/turnaround times (versus FCFS)
2. SRT is "better" at getting I/O-bound or interactive processes through the CPU even more quickly than SJF
3. good low turnaround times (and therefore low response times) for interactive or I/O-bound processes

Disadvantages:

1. processes with larger CPU burst times might end up facing INDEFINITE BLOCKING (or STARVATION)
2. increased overhead due to the priority queue (log time)
3. we have no way of knowing ahead of time exactly what CPU burst times will be for each process (we can only predict these CPU burst times...)
4. more context switches are likely (versus SJF)

Round Robin (RR) Algorithm

- Essentially FCFS with a fixed time limit on each CPU burst (i.e. a timeslice)
    - When a process starts using the CPU for its current CPU burst, a timer is set timer (timeslice) expires
    - Or...the process is PREEMPTED by the expiration of this timer, in which case the process is added back to the end of the ready queue

- Selection of the timeslice is crucial
    - too large of a timeslice and we end up with FCFS
    - too small of a timeslice and we have way too many context switches
    - aim is to minimize turnaround times if we can get "most" of the processes finishing their respective CPU burst times within ONE timeslice
    - heuristic is ~80% of CPU burst times should be less than the timeslice

★ With N processes, each process gets $\frac{1}{N}th$ of CPU time (which should ensure fairness)

★ If a process arrives at some later time (i.e., not all processes start at time 0), we need to decide where the process should be placed in the ready queue

Alternate approach, when a process arrives, add it to the front of the queue (i.e., have it cut the line)

- this "breaks" the fairness idea
- the advantage here is that I/O-bound or interactive processes get through their CPU bursts quickly and get back to more I/O
- one other idea: maybe we only have certain processes that we've identified as I/O-bound cutting the line

e.g., apply the RR algorithm to the processes listed below using a timeslice of 3ms

Priority Scheduling Algorithms

Each process is assigned a priority based on

- Predicted CPU burst times (e.g. SJF/SRT)
- Ratio of I/O activity (predicted or expected)
- System resource usage
- Arbitrary or hard-coded (user-defined?)

The process with the highest priority gets scheduled with the CPU first

★ When multiple processes have the same priority, we need a tie-breaker, which is a secondary algorithm on that subset (e.g., just use FCFS)

★ We decide (ahead of time) whether the algorithm is preemptive (e.g., SRT) or non-preemptive (e.g., SJF)

★ To help avoid starvation or indefinite blocking, we can use AGING: if a given process is in the READY state (i.e., in the ready queue) for some X amount of time, then we increase the priority of that process by Y

As a process is running with the CPU, change (lower?) its priority

Algorithm	Preemption?	Advantages	Disadvantages
FCFS	non-preemptive	Easy to implement Minimal overhead No starvation	long wait times long turnaround times
SJF	non-preemptive	Optimal (fastest) (least average wait time)	starvation Requires us to predict CPU burst times
SRT	preemptive	None?	starvation requires us to predict CPU burst times
Priority	non-preemptive or preemptive	finer control over process order	starvation (also increased overhead)
Priority with Aging (PWA)	non-preemptive or preemptive	no starvation	we still have long wait times for low
RR	preemptive	favors CPU-bound overhead and I/O-bound processes? no starvation	

For **SJF/SRT***, we can predict the CPU burst times for each process based on historical data, e.g., measures of previous actual CPU burst times*

Exponential Averaging (for each process)

alpha: constant in the range [0.0,1.0], often at least 0.5 or higher

tau: estimated CPU burst time

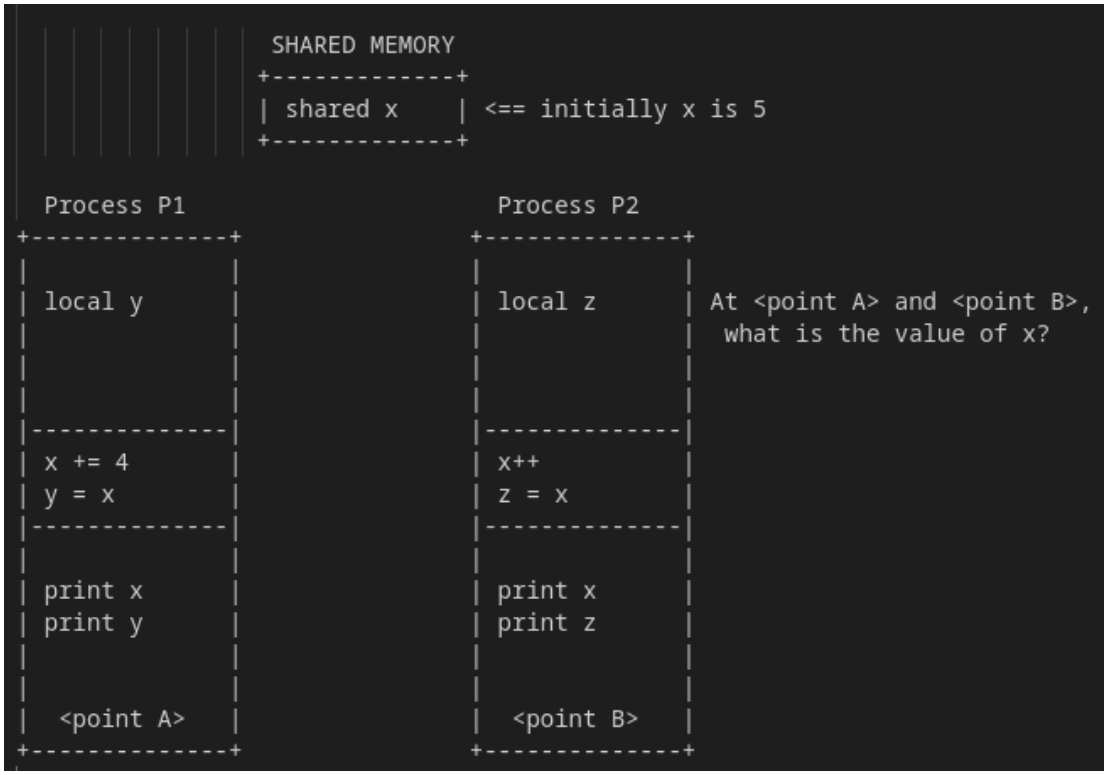
t: actual CPU burst time

Apply the formula below separately to each process so we keep track of a tau value per process

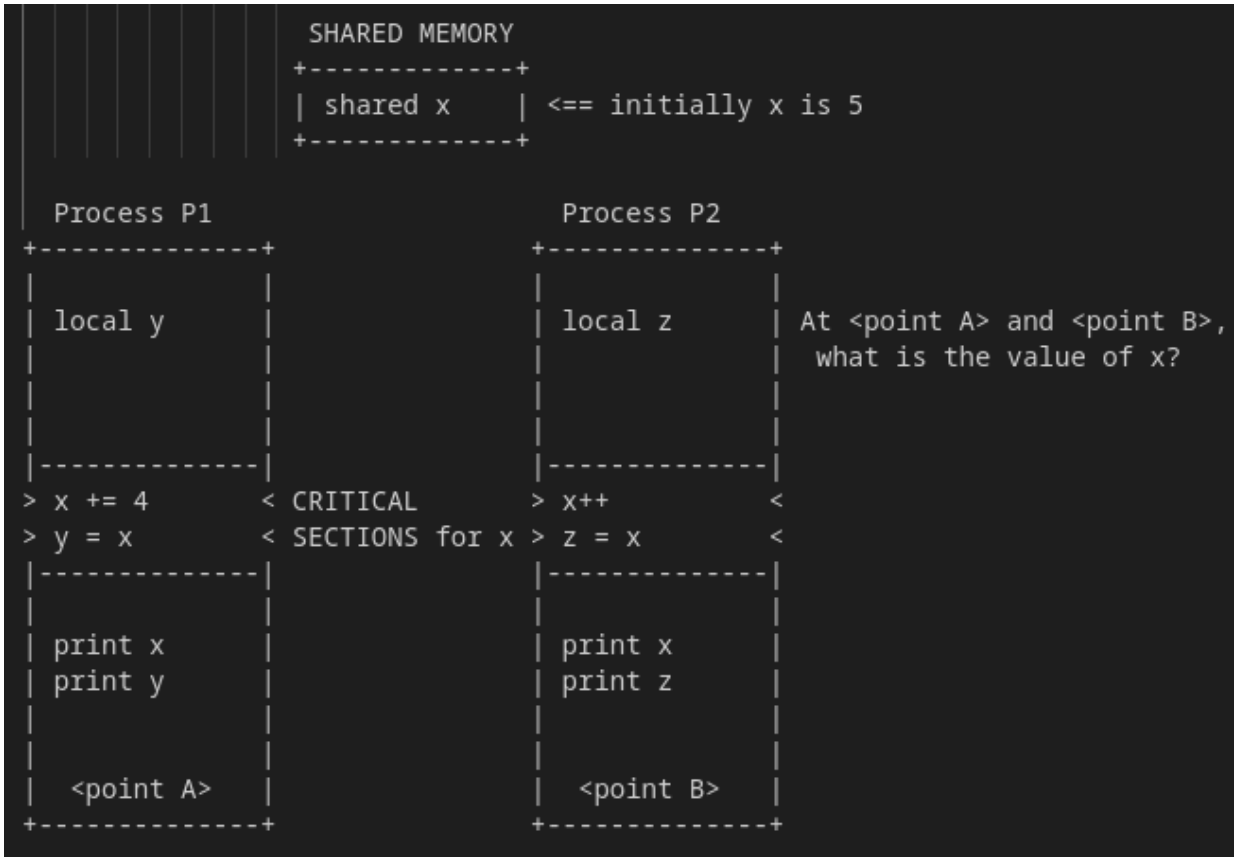
$$tau_{i+1} = alpha * t_i + (1 - alpha) * tau$$

Did you know, grade inquiries opened as of 11:59PM Jul 10 (due within one week)

Synchronization



There are multiple possibilities, what section of code is critical for splitting those possibilities?



The highlighted CRITICAL SECTIONS are shown above

- To synchronize these two processes, only one (and no more than one) process can be in its critical section at any given time
- A critical section guarantees mutual exclusion among multiple processes for access to one or more shared resources, e.g., shared variable x
- To synchronize processes, first we must identify the critical sections of code within each process
- The OS must control access to the critical sections by providing us (as programmers) some mechanism to define these critical sections, etc. (e.g., semaphores)

START LECTURE 7-13

Note: the files with "alarm" in the name are kinda…useless…

`sleep(int s)`

- puts the current process to sleep for `s` seconds
- returns 0 if the requested time has elapsed

`int usleep(useconds_t useconds)` sections

- Put the current process to sleep for `useconds` microseconds
- returns the value 0 if the execution was successful (execution successfully suspended) else returns -1

C Struct

```
typedef struct {
 int seconds;
 char msg[MAXLINE];
} alarm_t;
```

Pthreads

```
#include <pthread.h>
int pthread_create(pthread_t * thread,
                   const pthread_attr_t * attr,
                   void * (*start_routine)(void *),
                   void *arg);
```

- **thread:** pointer to an unsigned integer value that returns the thread id of the thread created.
- **attr:** pointer to a structure that is used to define thread attributes like detached state, scheduling policy, stack address, etc. Set to NULL for default thread attributes.
- **start_routine:** pointer to a subroutine that is executed by the thread. The return type and parameter type of the subroutine must be of type void *. The function has a single attribute but if multiple values need to be passed to the function, a struct must be used.
- **arg:** pointer to void that contains the arguments to the function defined in the earlier argument

```
void pthread_exit(void *retval);
```

- This method accepts a mandatory parameter retval which is the pointer to an integer that stores the return status of the thread terminated

```
int pthread_join(pthread_t th,
                 void **thread_return);
```

- ★ *used to wait for the termination of a thread*
- **th:** thread id of the thread for which the current thread waits.
- **thread_return:** pointer to the location where the exit status of the thread mentioned is stored.

```
pthread_t pthread_self();
```

gets the thread id of the current thread

```
int pthread_equal(pthread_t t1,
                  pthread_t t2);
```

- ★ compares whether two threads are the same or not. If the two threads are equal, the function returns a non-zero value otherwise zero
- t1: the thread id of the first thread
- t2: the thread id of the second thread

mutex-example.c

Shared Memory

```
#include <sys/shm.h>
key_t key = 74832; //any number works

/* create the shared memory segment with a size of 4 bytes */
int shmid = shmget(key, sizeof(int), IPC_CREATE | IPC_EXCL | 0660);
if (shmid == -1) { /* failure */ }

int * data = shmat( shmid, NULL, 0 ); // attach to the shared memory segment
if ( data == (void *)-1 ) { /* failure */ }
*data += 12; // add 12 to the data segment

int rc = shmdt( data ); // detach from the shared memory segment
if ( rc == -1 ) { /* failure */ }

int detRes = shmctl( shmid, IPC_RMID, 0 );
if (detRes == -1) { /* failure */ }
```

ATTY FILES

```
#define SHM_SHARED_KEY 8999
 struct termios ttyraw, ttyrestore;

 /* if stdin is a terminal/shell, switch to raw mode to avoid buffering */
 if ( isatty( STDIN_FILENO ) ) {
   tcgetattr( STDIN_FILENO, &ttyrestore );
   cfmakeraw( &ttyraw );
   tcsetattr( STDIN_FILENO, TCSANOW, &ttyraw );
 }
/* create the shared memory segment with a size of 16 bytes */
 key_t key = SHM_SHARED_KEY;
 int shmid = shmget( key, 16, IPC_CREAT | IPC_EXCL | 0660 );

 if ( shmid == -1 ) {
   perror( "shmget() failed" );
```

```
  return EXIT_FAILURE;
}
printf( "\rCHILD: type some characters (enter '!' to end)\n" );
int c;
char * ptr = data;

do {
    printf( "\r" );
    c = getchar();
    *ptr = c;
    ptr++;
  }
  while ( c != '!' );
int rc = shmdt( data );
if ( isatty( STDIN_FILENO ) ) {
  tcsetattr( STDIN_FILENO, TCSANOW, &ttyrestore );
}
```

Mutex and Semaphores

- A semaphore is an operating system construct that serves as a mechanism for synchronizing access to one or more shared resources
- an integer variable, shared among multiple processes where
    - A value of zero indicates the shared resource is not available
    - A value greater than zero indicates that that number of shared resources is available

Two atomic uninterruptable operations are:

- P() blocks until a resource is available, then acquires the resource
- V() relinquishes a resource

```
/* P() blocks until a resource is available, then acquires the resource */
P( semaphore S )
{
  while ( S == 0 ) { /* busy wait */ }
  S--;
}

/* V() relinquishes a resource */
V( semaphore S ) { S++ }
```

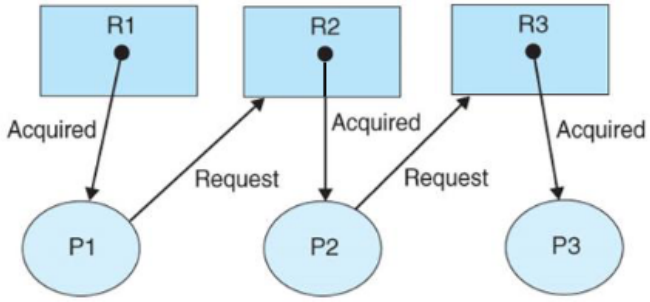
Note: a mutex is just a binary semaphore

Deadlock

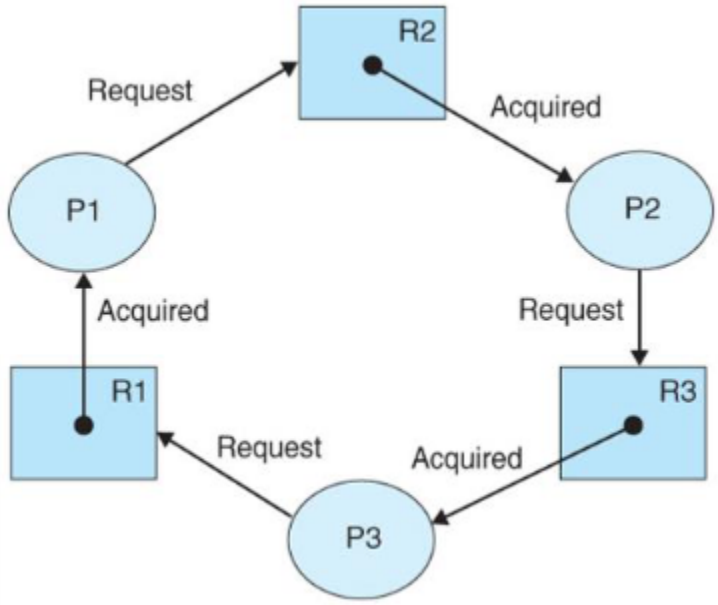
- no thread can make any further progress in its execution
- All threads are blocked on a P() operation, but the requested resource(s) will never become available
- Deadlock requires four conditions:
    - Mutual exclusion: threads require mutually exclusive access
    - Hold and wait: after acquiring a resource via P(), a thread will hold on to that resource indefinitely (forever!)
    - No preemption: once a thread has obtained a lock, we cannot preempt that lock
    - Circular wait: a cycle exists in the corresponding resource allocation graph…

Resource Allocation Graph

- A directed graph in which directed edges represent processes requesting and acquiring resources



A cycle indicates deadlock



Open Systems Interconnection (OSI) Reference Model

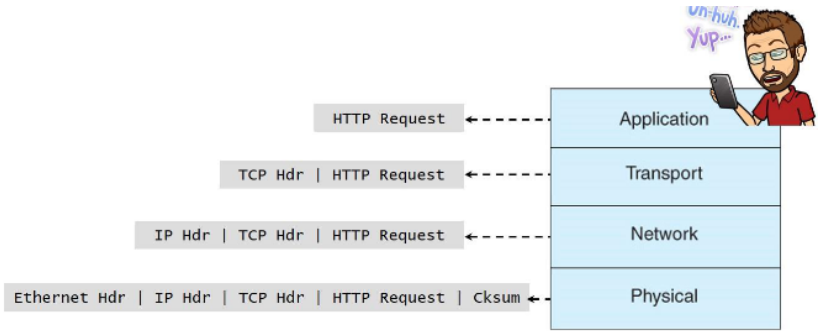
Standardization of how communication should occur across a network,

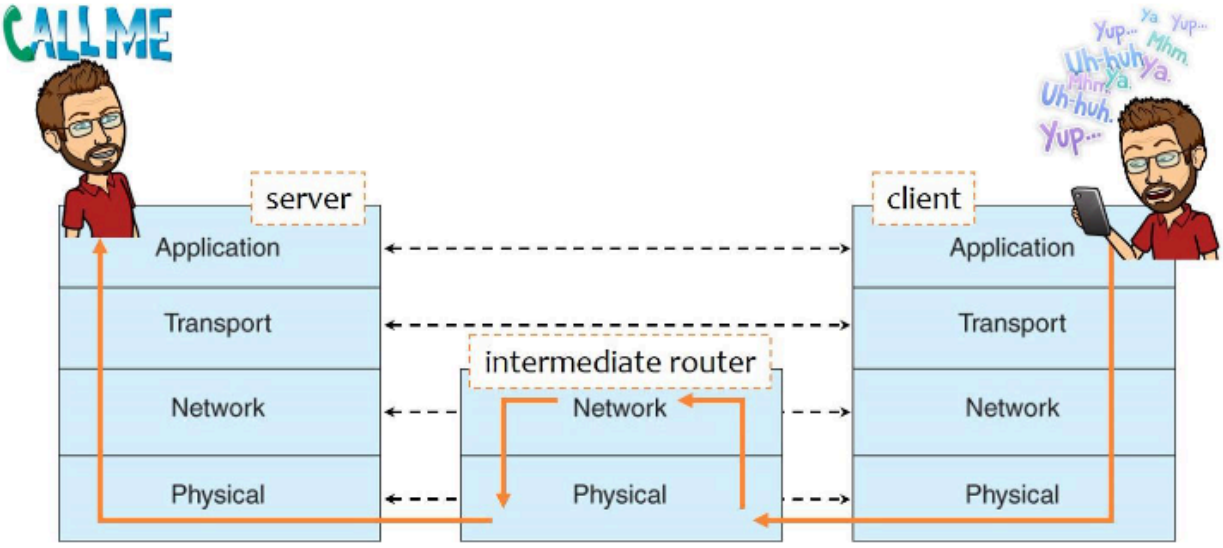
describing where and how network protocols fit together with one another

A seven-layer protocol stack that supports interoperability of networking components:

- Layer 7: Application (e.g., HTTP, HTTPS, NFS, SMTP, SNMP, TELNET)
- Layer 6: Presentation (e.g., SSL, FTP, SSH)
- Layer 5: Session (e.g., RPC)
- Layer 4: Transport (e.g., TCP, UDP)
- Layer 3: Network (e.g., IP, ICMP, ARP, OSPF)
- Layer 2: Data Link (e.g., MAC)
- Layer 1: Physical (e.g., Ethernet, Frame Relay, IEEE 802.11)

COMMUNICATING VIA THE OSI REFERENCE MODEL



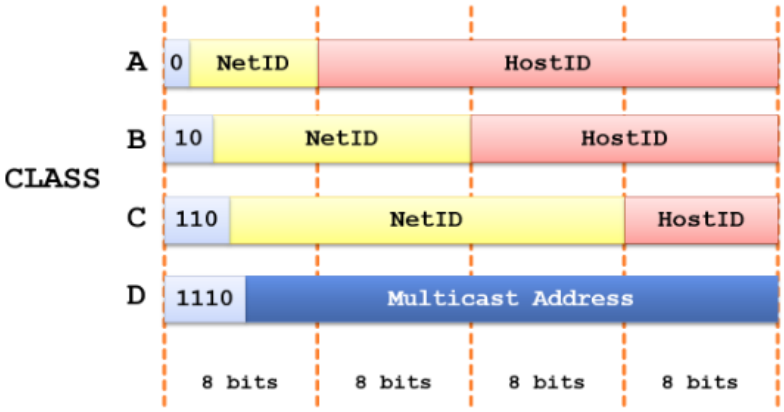


IP Addresses

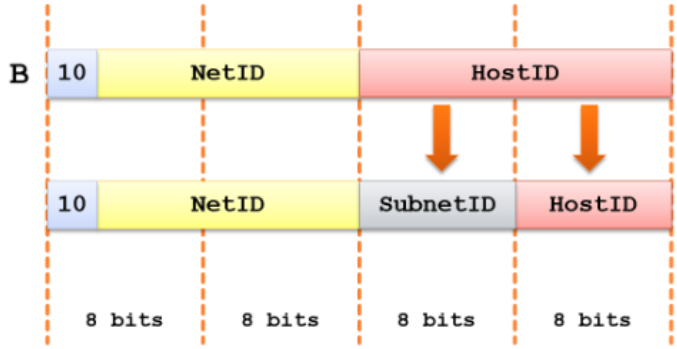
Each IP address contains information about what network the destination host is on, which enables routing to occur at intermediate “hops” (i.e., routers) along the path from a source endpoint to the destination endpoint

NETWORK CLASS	LEADING BITS	# of NETWORKS	# of HOSTS	NETWORK/HOST BIT FIELDS
CLASS A	0...	128	16,777,214	8 / 24 bits
CLASS B	10...	16,384	65,534	16 / 16 bits
CLASS C	110...	2,097,152	254	24 / 8 bits
CLASS D	1110...	n/a	n/a	n/a
MULTICAST				

Decoding IP Addresses



Subdividing into Subnets



TCP and UDP

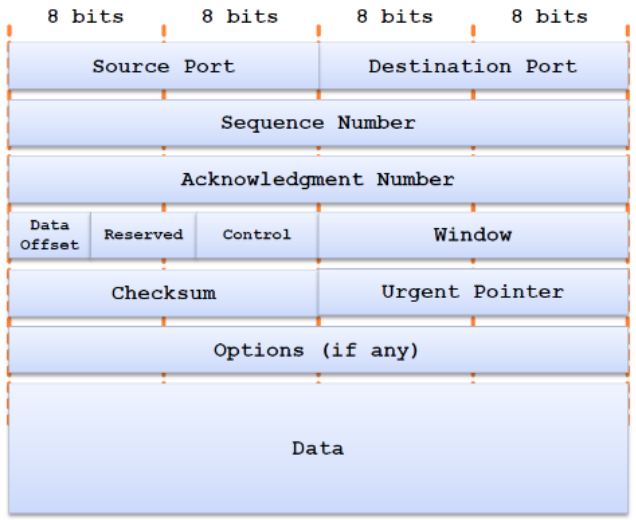
User Datagram Protocol (UDP)

- User Datagram Protocol (UDP):
- Connection-less
- Unreliable (i.e., no re-sending
- of dropped datagram)
- Simple
- Low overhead
- ★ delivery and duplicate protection are not guaranteed. Applications requiring ordered reliable delivery of streams of data should use the TCP [per this reference]

Transmission Control Protocol (TCP)

- Transmission Control Protocol (TCP):
- Connection-oriented
- Reliable (i.e., re-sending of dropped packets, sequencing and reordering of packets, general error checking)
- Overhead

TCP segment Structure



INTER-PROCESS COMMUNICATION (IPC)

Requires:

1. Synchronization
2. Protocol (i.e., how is communication to occur between the endpoints?)
3. Precision
4. Data marshaling (i.e., translating from “host format” to “network format”)

Socket-based Communication

```
#include <arpa/inet.h>
```

- A socket is an endpoint for communication, so (at least) two endpoints are required
- A server process creates one or more sockets that it then listens on for incoming connection requests or incoming datagrams
- Server processes listen on specific port numbers, which are 2-byte values
- Sockets-based communication can be connection-oriented or connection-less

Data Marshalling

- Ensures we send the correct data
- Big endian stores the most significant byte (MSB) first, i.e., in the lowest memory address
- Little endian stores the least significant byte (LSB) first, i.e., in the lowest memory address
- Network format (i.e., network byte order) is standardized to use big endian see htons(), ntohs(), htonl(), ntohl(), etc. (also try “man endian”)

Note: The `htons()` function makes sure that numbers are stored in memory in network byte order, which is with the most significant byte first

Demo Code - UDP Server

```
int sd;
sd = socket(AF_INET, SOCK_DGRAM, 0);
if (sd == -1) { /* socket failed */ }
struct sockaddr_in udp_server;
```

```
int length = sizeof(udp_server);
udp_server.sin_family = AF_INET; // IPv4
// any remote IP can send us a datagram
udp_server.sin_addr.s_addr = htonl(INADDR_ANY);
// for bind(), the 0 here means let the OS assign a port number
udp_server.sin_port = htons(0); // htons( 12345 );
if (bind(sd, (struct sockaddr *)&udp_server, length) == -1) {/* bind fail */}

// CLIENT CODE
char buffer[MAXBUFFER + 1];
struct sockaddr_in remote_client;
int addrlen = sizeof(remote_client);
// read an n-byte datagram from the remote client side (BLOCKING)
int n = recvfrom(sd, buffer, MAXBUFFER, 0,(struct sockaddr *)&remote_client,
                  (socklen_t *)&addrlen);
if (n == -1) { /* recvfrom() failed */ }
buffer[n] = '\0';
sendto(sd, buffer, n, 0, (struct sockaddr *)&remote_client, addrlen);
```

Demo Code - TCP Server

```
// Create the listener socket as TCP socket (SOCK_STREAM)
int listener = socket( AF_INET, SOCK_STREAM, 0 );
if (listener == -1) { /* socket() failed */ }
// populate the socket structure for bind()
struct sockaddr_in tcp_server;
tcp_server.sin_family = AF_INET;   /* IPv4 */
// allow any remote IP address to connect to our socket
tcp_server.sin_addr.s_addr = htonl( INADDR_ANY );
unsigned short port = 8192;
tcp_server.sin_port = htons( port );
if ( bind( listener, (struct sockaddr *)&tcp_server, sizeof( tcp_server ) ) ==
                                                        -1 ) { /* fail */ }
if ( listen( listener, 5 ) == -1 ) { /* listen fail */ }
while ( 1 ) {
      struct sockaddr_in remote_client;
      int addrlen = sizeof( remote_client );
      // SERVER BLOCKING CALL
```

```
      int newsd = accept( listener, (struct sockaddr *)&remote_client,
(socklen_t *)&addrlen );
      if ( newsd == -1 ) { perror( "accept() failed" ); continue; }

      // congrats, you're connected
      pid_t p = fork();
      if ( p > 0 ) { close( newsd ); } /* PARENT */
      else {
            int n;
            do {
                  // CLIENT BLOCKING CALL (CHILD)
                  n = recv( newsd, buffer, MAXBUFFER, 0 );
                  buffer[n] = '\0';
                  char* addr = inet_ntoa((struct
                                    in_addr)remote_client.sin_addr), buffer);
                  n = send( newsd, "ACK\n", 4, 0 ); // or we can use write()
            } while (n > 0);
            close( newsd );
            return EXIT_SUCCESS;
      }
      close( listener );
}
```

Note: read is a BLOCKING call

Signals

```
signal( SIGINT, SIG_IGN );   /* ignore SIGINT (CTRL-C) */
signal( SIGTERM, SIG_IGN );  /* ignore SIGTERM */

/* redefine SIGINT to call signal_handler() */
signal( SIGINT, signal_handler );

/* restore SIGINT (CTRL-C) to default behavior -- see signal(7) */
signal( SIGINT, SIG_DFL );
```

Server Types

(0) Iterative server (often UDP): accept incoming UDP datagrams on

a specified port number (udp-server.c); or accept and handle
TCP connections one at a time or serially (tcp-server-iterative.c)

1. fork()-based server (tcp-server-fork.c): a child process is created to handle each rcvd/accepted TCP client connection
2. fork()-based server with pre-forked processes: pre-fork() N processes, then as incoming TCP client connection requests come in, assign them to child processes (which requires shared memory, synchronization, etc.)
3. thread-based server (Homework 3): a child thread is created to handle each rcvd/accepted TCP client connection
4. thread-based server with pre-spawned threads: pre-spawn N threads, then as incoming TCP client connection requests come in, assign them to child threads (which requires synchronization!)
5. select()-based (multiplex) server
    a. a single process (no multi-threading) that uses the select() system call, which enables the process to poll multiple descriptors and check for activity on any of them (tcp-server-multiplex.c)
    b. this approach eliminates the need (complexity/overhead) of multiple processes or multiple threads...but this is then an iterative server (and we can use select() on ANY type of descriptor)