CS214-FINAL REVIEW[SPRING 2018]

THIS DOC HAS FINAL REVIEW COLLECTED FROM ALL SOURCES. PLEASE ADD ANY ADDITIONAL IMPORTANT NOTES HERE.
FINAL CHECK LIST IS AT THE END OF THE DOC!!

STUFF I REMEMBER FROM FINAL SPRING 2018:

- Kernel threads vs user threads
- Top vs ps
- Network layout (OSI)
- Difference between UDP and TCP
- Is it possible to ever stop a deadlock from occuring?
- Can you get out of a deadlock in semaphores?
- Can you get out of a deadlock in mutexes?
- LEARN ALL ABOUT DEADLOCK
- Conditions of deadlock
- Distributed Systems extra credit stuff (usually talks about it in last 2 lectures)
- What is a stateless function: identify it
- Can you force a child process to execute a different code?
- Zombie process, orphan process, zombie orphan
- Fork() and exec() - learn them
- What is |, > , >>
- Synchronization
- Internet protocol stuff

A. Required Stuff

0.a Presume the following code has run:

```
int x = 4; int* y = &x; int** z = &y; int a = 10;
```

Write 4 different assignment operations that set `x` equal to the value of `a`.

Int x = 10; int x = &a;

int x = *a;

Int x = *(&a);

0.b Presume the following code has run:

```
int x = 4;
```

```
int* const y = &x;

int const **z =&y;

int a = 10;
```

Which of your assignments (if any) from **0.a** are now invalid?

Int x = &a; ⇒ should be int* const x = &a;

What does the `const` modifier do to each of the pointers it is applied to?

Const char* ⇒ pointer to a char, value can't be changed

char* const ⇒ pointer to a char, value can be changed but pointer can't point to a different char.

Const char* const ⇒ const pointer to const char, neither the value nor where the pointer points to can be changed.

1.Write some code that forks a process and prints "parent" in the parent process and "child" in the child process.

```
#include<stdio.h>

#include <sys/types.h>
#include<unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork()==0)
        printf("Hello from Child!\n");

    // parent process because return value non-zero.
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

OUTPUT:

```
1.

Hello from Child!
```

```
Hello from Parent!
```

```
       (or)
```

```
2.
```

```
Hello from Parent!
```

```
Hello from Child!
```

Negative Value: creation of a child process was unsuccessful.

Zero: Returned to the newly created child process.

Positive value: Returned to parent or caller. The value contains process ID of newly created child process.

I AM NOT SURE IF THIS IS CORRECT! https://www.geeksforgeeks.org/fork-system-call/

2. What is the difference between `fork()` and `exec()`? Can one `fork()` without `exec()`ing?

fork() - creates new processes (return process ID). New processes becomes the *child* process of the caller.

exec() - REPLACES entire current process with a new process.

`fork` and `exec` are often used in sequence to get a new program running as a child of a current process. BUT FORK() CAN WORK WITHOUT EXEC(). For example, if a program knows it wants to execute another program, it doesn't need to exec().

A common programming pattern is to call fork followed by exec and wait. The original process calls fork, which creates a child process. The child process then uses exec to start execution of a new program (exec() replaces the program in the current process with a brand new program, by specifying an executable or or an interpreter script to it). Meanwhile the parent uses wait (or waitpid) to wait for the child process to finish.

3. Assume a parent forks a child immediately on start and then immediately waits on the child. The parent takes 1s of execution time and the child takes 2s of execution time.

a. On a single core processor, what is the total time (wall clock time) for execution? Why?

Wall-clock time is the time that a clock on the wall (or a stopwatch in hand) would measure as having elapsed between the start of the process and 'now'

My guess: 3s

b. On a multi-core processor, what is the total time (wall clock time) for execution? Why?

4. Assume a parent creates a thread immediately on start and does not join in. Assume the thread is scheduled by the kernel. The process' main function takes 1s of execution time, and the thread takes 2s of execution time.

a. On a single core processor, what is the total time (wall clock time) for execution? Why?

On a single core processor, the two threads will run sequentially and the overall time will be 1 + 2 = 3s.. Idk why

b. On a multi-core processor, what is the total time (wall clock time) for execution? Why?

On a multicore processor, the execution can be in parallel and so the execution time will be 2s as both can run in parallel.

5. Are synchronization constructs required for multiprocessing programs? Why or why not?

-The need for synchronization does not arise merely in multi-processor systems but for any kind of concurrent processes; even in single processor systems

https://en.wikipedia.org/wiki/Synchronization_(computer_science)

6. Show how a race condition can occur between 2 threads running the code below at the same time:

```
int i = 0;
void* foo()
{
  printf("%d", i);
  i++;
}
```

We can add another method that is changing the variable i. For Example:

```
void* foo1()
{
printf("%d", i);
  i = i+2;
```

}

https://www.cs.cornell.edu/courses/cs2110/2014fa/L25-ConcurrencyII/cs2110fa14Concurrency2-6up.pdf

A race condition occurs when two or more threads can access shared data and they try to change it at the same time.

Race Condition:

-One thing finishing before another does when the two are consecutively related

-Can be prevented with semaphores, locks, etc..

7. Consider the following communication calls:

Socket, bind, connect, accept, close

If you are building a client program, which calls would you most likely use? What do they do?

GREAT WEBSITE: https://www.geeksforgeeks.org/socket-programming-cc/

socket(): creates an endpoint for communication & returns a file descriptor that can be used in later function calls that operate on sockets

connect(): system call connects the socket referred to by the file descriptor sockfd

Found in lecture notes=> building:(client socket)

hostent (set IP addr, address familty (AF_INET)..htons() the port num)

sockaddr_in

socket()

connect()

8. What is a file descriptor? Why would you need a 'file' descriptor to do socket communication?

A file descriptor is an " int handler" that tells if a file or a socket is opened. A socket is treated like a file so the file descriptor is to reference the socket. For example, -1 if it fails.

9. Why does server code usually involve concurrent programming?

Concurrent programming lets us have *multiple threads of control* in our single program. In other words, multiple separate sections may be executing "simultaneously".The server would

be able to process requests in parallel, and, although the CPU's time would still be divided amongst the *n* tasks, it might be able to finish one or more of the simpler requests while it is concurrently processing the longer request.

Read this. .. very helpful=> https://www.ocf.berkeley.edu/~fricke/threads/threads.html

10. What does `getaddrinfo()` do when a given a host name rather than an IP address?

The getaddrinfo() function combines the functionality provided by the gethostbyname(3) and getservbyname(3) functions into a single interface. The function `getaddrinfo` can convert a human readable domain name (e.g. `www.illinois.edu`) into an IPv4 and IPv6 address. In fact it will return a linked-list of addrinfo structs. A system called "DNS" (Domain Name Service) is used. If a machine does not hold the answer locally then it sends a UDP packet to a local DNS server. This server in turn may query other upstream DNS servers.

http://cs241.cs.illinois.edu/wikibook/networking-part-2-using-getaddrinfo.html

11. What does `gethostbyname()` do when given an IP address rather than a host name?

`-gethostbyname()` takes a string like "www.yahoo.com", and returns a struct hostent which contains tons of information, including the IP address.

https://beej.us/guide/bgnet/html/multi/gethostbynameman.html

12. When might `read()` on a socket return fewer bytes than specified?

On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. **It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal.**

13. A program calls `bind()`, and `bind()` again. What could happen?

A port cannot be bind twice. bind() assigns the address specified by addr to the socket referred to by the file descriptor sockfd. addrlen specifies the size, in bytes, of the address structure pointed to by addr. Traditionally, this operation is called "assigning a name to a socket".

B. Choose Your Own Exam!

Choose and answer 10 of the questions below:

0. Does a mutex lock guarantee thread safety? Why or why not?

Mutex guarantees thread safety partially. Access to shared data is *serialized* using mechanisms that ensure only one thread reads or writes to the shared data at any time. Incorporation of mutual exclusion needs to be well thought out, since improper usage can lead to side-effects like deadlocks, livelocks and resource starvation.

1. When can casting a struct ultimately end in a segmentation fault?

Someone had this issue.. But I don't think that satisfies our questions⇒
https://stackoverflow.com/questions/3026858/segfault-possibly-due-to-casting

2. What information is stored on the stack and heap?

Local variables and Dynamically allocated variables respectively.

3. What processes inherits zombie processes? Why?

A process which is done running, but the parent has not informed the OS it is done. The process will not be deallocated. It will keep running and waste time till its time slice is over is called a Zombie Process. Zombie processes whose parents return or disappear have init as their new parent, which then wait()-s on them and frees the resources.

Orphan process: Parent dies before child. Immediately adopted by init process.

4. How can a parent process communicate information to a child process?

- A parent and child can communicate through any of the normal inter-process communication schemes (pipes, sockets, message queues, shared memory), but also have some special ways to communicate that take advantage of their relationship as a parent and child.
- One of the most obvious is that the parent can get the exit status of the child.
- The child inherits file descriptors from its parent, the parent can open both ends of a pipe, fork, then the parent close one end and the child close the other end of the pipe. This is what happens when you call the `popen()` routine to run another program from within yours, i.e. you can write to the file descriptor returned from `popen()` and the child process sees it as its stdin, or you can read from the file descriptor and see what the program wrote to its stdout.
- the child process inherits memory segments mmapped anonymously (or by mapping the special file `` `/dev/zero' ``) by the parent; these shared memory segments are not accessible from unrelated processes.

http://www.unixguide.net/unix/programming/1.5.shtml

5. What happens to the parent if the child segfaults?

If the child segfaults then it needs to be terminated the parent process get allocated to the new children based on the requirements.

6. What happens to a process when one of its threads segfaults?

The segfault handler is called for the process. If you don't do anything special, the OS will kill the process and reclaim its resources.

7. What is a signal? How are they different from Exceptions in Java?

Signal is like a software version of interrupt. A signal is a small message that notifies a process that an event of some type has occurred in the system.

Exceptions happen during runtime or compile time in a program. It is caused by the program and it can be fixed. Once fixed, it will not reoccur.

8. What is shared among threads in a multithreaded process?

(-threads share the same memory and open files)???

In a multi-threaded process, all of the process' threads share the same memory and open files. Within the shared memory, each thread gets its own stack. Each thread has its own instruction pointer and registers. Since the memory is shared, it is important to note that there is no memory protection among the threads in a process.

9. Make the following code thread safe:

```
int total = 0;
void add(int value)
{
	if (value < 1)
		return;
	Else if (total != 1000)
	{
		lock(mutexA);
		total += value;
		unlock(mutexA);
	}
}
void sub(int value)
{
	if (value < 1)
		return;
```

```
        Else
        {
                lock(mutexA);
                total -= value;
                unlock(mutexA);
        }
}
```

10. If we wish to ensure that the value of `total` in the code in **9** NEVER exceeds 1000, what can we do? Insert the necessary constructs/calls to do so:

Look above ^_^

11. Provide an example of code that can lead to deadlock. Describe how deadlock might occur.

Improper use of mutexes can lead to deadlock. For instance,

```
 Thread0                    Thread1

lock(mutexA);               lock(mutexB);

lock(mutexB);               lock(mutexA);

a = b+1;                    a = b-3;

unlock(mutexA);             unlock(mutexA);

unlock(mutexB);             unlock(mutexB);
```

An example of deadlock is when two threads, thread 0 and thread 1, each acquires a mutex lock, A and B, respectively. Suppose that thread 0 tries to acquire mutex lock B and thread 1 tries to acquire mutex lock A. Thread 0 cannot proceed and it is blocked waiting for mutex lock B. Thread 1 cannot proceed and it is blocked waiting for mutex lock A. Nothing can change, so this is a permanent blocking of the threads, and a deadlock

12. There are 2 problems with the code below, what are they?

```
 char* xpx(char* src)
{
        char result[sizeof(src)];
        strcpy(result, src);
        return result;
}
```

```
Src is a char* which holds an
address. It does not carry the
data value so that is an error.



sizeof(src) should be malloced.
char* result = (char*)
(malloc(sizeof(src)));
```

Scratch/Additional space:

What does the function (open()/close()) do? What does the opposite of

(open()/close()) do?

Open is for pipes or network sockets; fopen for files Open() opens a file descriptor(int) for reading or writing, which is used to refer to the file. If unsuccessful it returns -1 and sets the global variable errno to indicate the error type. Whereas, **close**() closes the file descriptor. Everything associated with that file descriptor is removed.

b. What does the function

pthread_mutex_lock()/pthread_mutex_unlock()/pthread_mutex_trylock()) do? What function does the opposite?

They return zero on success; that is, if they lock or unlock the mutex(mutual exclusion object) successfully. Otherwise, they block the thread.The difference between _lock() and _trylock() is that _trylock() returns immediately and acquires the mutex, rather than blocks if it can't get ownership of the mutexmutex_lock() waits until it acquires mutex.

What is join()?

to "join" a thread is to wait until it's done and catch its return value and stack

How does a file descriptor differ from a file handler (File * pointer) ?

A **file descriptor**, like a socket, is an int that identifies as a file at the kernel level. It is passed in methods like read() and write(). Array name + pointer name.

A **file handler** is a struct that includes the file descriptor, along with other useful things like the end-of-file. Used for methods like fscanf(). Array index + pointer offset.

A **file pointer** is c standard library level struct used to represent a file.

How does child process differ from a parent process?

Address space, pids and ppids are different, they are similar in the sense that they have the exact same code, same variables/data, file descriptors/handlers, and signal disposition

Right after fork(), but before exec(), what attributes of the child process are different?

PId is different as well as PPid, address space and return value of fork().

What process has a PID of 0

the init has a PID of 1, scheduler has a PID of 0

Why is it harder to share information(memory) between processes than threads?

Threads share memory (same addr space), therefore if you change something one Thread, it'll be the same for the rest. As for processes, each process has it's own address space. So sending information from one Process to another would require allocating/reallocating space. Specifically, threads share heap space

Which of the four smurfs(deadlock conditions) can we avoid with smurfs(mutexes)(timeouts)

i. Mutual exclusion- Can be avoided by: Using different variables or not using mutexes

ii. Hold and wait or partial allocation-Can be avoided by: allowing the holds to time out

iii. No pre-emption- Can be avoided by: A scheduler or moderator of the system

iv. Circular wait or resource waiting(the only one we, as users, can really avoid) Can be avoided by: Having a global order to the mutexes that are locked

Why shouldn't you unlock/lock a mutex twice?

Locking a mutex after having already locked it means that the thread is blocking, waiting for the mutex to be unlocked, and so can't unlock it itself. No one can (or should be able to) unlock the mutex and so it's game over.

a. What are the differences between (mutexes) and (condition?) variables:

I. A condition variable is a more advanced / high-level form of synchronizing primitive which combines a lock with a "signaling" mechanism. It is used when threads need to wait for a resource to become available. These are what's used by monitors to create thread-safe programs. A monitor is a combination of a mutex and condition variables.

ii. A mutex is the actual low-level synchronizing primitive. You can take a mutex and then release it, and only one thread can take it at any single time (hence it is a synchronization primitive)

f. how do signal and signal handlers differ

i. Signals - indicates some situation that must be dealt with before the code continues

Ii. Signal handlers - reroute certain signals to your own code

a. Why is it dangerous to lock a mutex inside a signal handler in a thread(?) that signals(?)?

i.Locking a mutex inside a signal handler is dangerous because, other than the alarm signal, you don't' know when signals will go off, possibly not allowing the mutex to ever be unlocked and creating deadlock.

b. Why is it dangerous to malloc a space inside a signal handler in a _____

i. Since signal handler can be called at any time, if there is another call to malloc, then:

ii. Deadlock: malloc cannot get heap lock

c. Why is it dangerous to free a space inside a signal handler in a process that mallocs.

i. Signal Handler:You do not want to access global variables when you are in signal handler (or call methods) because: -signal handler can stop the program at any point. For example, you call add(2,3) in signal handler. However, another call to add(2,3) is happening at that very moment. Data is corrupted.

Basic C syntax:

So if y = 2;

&y,....addr

*(&y)...value

Y….value

what causes seg faults and how to avoid them, what the mem layout is

- Seg fault - program has attempted to access an area of memory that it is not allowed to access.
- When using array make sure the array doesn't go out of bounds. When freeing or deleting memory, make sure u malloc first. And Do not free memory twice.
- Freeing already freed pointers, dereferencing null pointers, accessing past valid array indices, accessing past string boundaries

BASICALLY= GOING OUT OF SCOPE.

Segmentation fault handler:

-use the signum to construct a pointer to flag on stored stack,

-increment pointer down to the stored PC

-increment value at pointer by length of bad instruction.

syntax: signal(SIGSEGV, segment_fault_handler);

Free can fail, how?

- If there is nothing to free..
- In other words, if pointer is already free (freeing twice will result in a dangling pointer) or if pointer does not exist (was not malloced in the first place).

- signal()
    - ***typedef void (*sighandler_t)(int);***
    - ***sighandler_t signal(int signum, sighandler_t handler);***
    - ***signal****() sets the disposition of the signal signum to handler, which is either* ***SIG_IGN****,* ***SIG_DFL****, or the address of a programmer-defined function (a "signal handler").*
        - *If the disposition is set to* ***SIG_IGN****, then the signal is ignored.*
        - *If the disposition is set to* ***SIG_DFL****, then the default action associated with the signal (see signal(7)) occurs.*
        - *If the disposition is set to a function, then first either the disposition is reset to* ***SIG_DFL****, or the signal is blocked , and then handler is called with argument signum. If invocation of the handler caused the signal to be blocked, then the signal is unblocked upon return from the handler.*

The signals ***SIGKILL*** *and* ***SIGSTOP*** *cannot be caught or ignored.*

Scheduler/init-PID 0

systemwide foster parent... takes over for parent processes that have for some reason. left the system without calling wait() in their child..

Semaphores

-Structure that increases/decreases in value using sem_post(sem_t * sem)/sem_wait(sem_t * sem), respectively. When the value of the semaphore reaches 0 the calling process blocks; one of the blocked processes wakes up when another process calls sem_post

-A semaphore is a data structure used to help threads work together without interfering with each other.

-When to use semaphores?

When trying to solve the Dining Philosophers Problem, semaphores represent a suitable locking mechanism.

CONDITION VARIABLES:

Condition variables should be used as a place to wait and be notified

Ex: pthread_cond_wait: (condition, mutex)

Ex: pthread_cond_signal (condition)

Ex: pthread_cond_init (and so on)

- lets threads synchronize on the VALUE of data

-condition variables are used together with mutexes.

-pthread_cond_wait() is passed a mutex and condition variable

-mutex is released by the system when the thread waits and reacquired by the system when the thread awakes.

How to use a semaphore as a condition variable?

Substitute the condition variable with a binary semaphore… since semaphores can be incremented/decremented from any thread or process depending on the flag set during initializing.

Saturation. What happens when you have a test program that allocates all of available dynamic memory?

-Memory saturation is when all of the memory on a machine is used (dynamically allocated) and cannot be used for future mallocs.

-In this case, any malloc after the saturation threshold has been reached will cause the machine to crash completely. Some compilers kill the process that caused saturation to prevent a machine crash.

Process: a series of actions or steps taken in order to achieve a particular end. It is like a procedure and has its own memory. Basically a single running program.

Thread: a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system.

- Saturation. What happens when you have a test program that allocates all of available dynamic memory?
    - Memory saturation is when all of the memory on a machine is used (dynamically allocated) and cannot be used for future mallocs.
    - In this case, any malloc after the saturation threshold has been reached will cause the machine to crash completely.
    - Some compilers kill the process that caused saturation to prevent a machine crash.
- FRAGMENTATION: when storage space is used inefficiently reducing performance. When a request for a large block is made, the request fails even though the total amount of free memory is larger than the requested block size. There are ways to deal with this problem. First, one part of your memory resource can be reserved for small blocks, leaving the rest available for larger blocks. Second, large blocks can be allocated from one end of your memory resource and small blocks can be allocated from the opposite end.
- How to prevent fragmentation:
    - Depending on the workload, these algorithms are very useful when trying to prevent fragmentation of memory:
    - *Malloc algorithms:
        - First fit: looking for the first free block to allocate.
        - Best fit: traverse the entire list looking for the closest fit to what you try to malloc(least amount of space that can fit).
        - Worst fit: looking for most space. (allows you to have larger space if workloads were similar to block size)

MUTEX AND SEMAPHORES:

Mutex is locking mechanism- locks and unlocks threads, whereas semaphores are synchronization mechanism- signal the waiting list

Network Programming:

- IP address how diff from hardware address
    - Hardware address is always unique to every single machine
    - IP address can be relative to a specific router and can be the same if another router is used somewhere else

Networking Layers:

ISO OSI 7 layer stack

Application =>

-program does stuff with data!! (wheee)

Presentation =>

-managing format of data to make it useful..encrypt/decrypt. trans/en/de-code

Session =>

-manages information over multiple sockets.. synch data streams..multiple streams

Transport (TCP: Transmission control protocol) =>

-manage delivery of a message ...reliability(data will be sent in the correct order), retransmission,integrity,size..

-create simulated connections

Network (IP: Internet Protocol) =>

-manage/discover/forward messages along routes to other hosts.. maintain routing table, change routes when a link is congested

–IPv4 ⇒ 32 bits split into octets (group of 8)

–IPv6 ⇒ 128 bits

Data Link =>

-Checks the information, checks for any interuptions!

-Manage a point to point link to other machine.. data integrity..data identity

[Data integrity refers to maintaining and assuring the accuracy and consistency

of data over its entire life-cycle, and is a critical aspect to the design,

implementation and usage of any system which stores, processes, or retrieves data]

Physical layer =>

-physically transmits bits by converting them to some radiation

CHANGE MODE:

a = all= Everyone

u = user = owner

g = group = everyone in file owners group

o = other = users not in file owner's group

chmod a+rw test.c //everybody read and write

chmod a-rw test.c //everybody can't read write

chmod u+rw test.c //user can read write

chmod a+rwx test.c //nobody can do anything

chmod g+r

chmod o-rwx //anyone not me and not in group can't do anything

IMPORTANT LECTURE NOTES:
https://content.sakai.rutgers.edu/access/content/group/353569d4-8cf4-4d54-8fc8-faec8d25fc69/Recitations/Gang_Section3/rec5/Important_Notes_v3.pdf
lecture 4/26/2018
/*
********** FINAL REVIEW ************
Definition:
function
design

cat stuff.txt|grep printf > logfile
cat stuff.txt >> logfile

semaphores:
monitors
barriers

locks

condtion variables
0. not in use
1. in use

locks guranties:
only locker can unlock
mutual exclusion implies that ->
 code in one critical section will be run before another (atomically)
 will block all other lock()ers until unlock() is called..soo.. must call lok only once and before unlock in a given context
 (mutexes) waiting (blocked) contexts will resolve in lock order

(binary) semaphore guarantee:
mutual exclusion (atomic set and test)
 will block all other produce callers until consume is called
 any context can call produce or consume at any time.
 can call produce or consume any number of times in the same context in any order
 -> will block on consume at min value and at produceat max value
*don't necessarily have to be resolved in lock order (because there are no locks)

deadlock conditions:
0. mutual exclusion
1. non-preemption
2. hold and wait
3. circular wait

Avoiding deadlock:
-Global ordering of mutex locking
-Rising/falling permission phases

NOTES:
~sometimes total ordering is not possible...usually when dealing with machine events
~Task for monitors - Object-oriented synchronisation mechanisms that can be programmed with the logic between or requirements or other
    dependencies between resources and will make sure to only allocate them in a non-deadlocking order.

~File:
file descriptor
blocking and non blocking

~file permissions:
both the mnemonic and numeric representations
u g o
rwx rwx rwx
110 000 000

6 0 0
open("./somefile",O_RDWR|O_CREAT, 00600)

chmod
seek
touch

~THREADS:
attributes, how do u create a thread?
void*
know what is required to build thread..not necessarily format , other than the parts you need to write/supply
know about k u v-threads
..what threads are
problems that can result from threads.. (accessing memory..synchronization stuff.. why do u need special instruction for synchronisation?..)
what is join() and why?
-blocks until thread exits
-join either if you need some resource or action from the thread to be complete before continuing ..or you want to make sure it is done doing
 whatever it is doing before you return/exit

~know errno,perrno
*/