(Write anything you guys think the questions could be)
FINISH DECODING THE EXAM
yee

Good Luck Everyone

/* From what I can tell he bases these questions heavily on the questions he asks during class, topics he repeats over and over are probably what he asks here */

Also, here's the study guide I made for the midterm, might be helpful -> https://docs.google.com/document/d/1EeIT-7cuJDikclzjadngGmrDaV0AY2Z2_GYoCNkwaZ8/edit?usp=sharing

==========

0. Kinds of Types:

==========

Answer all of the following 10 parts.

a. **How much memory must the () take up in c?** (pointer?) 4 bytes in 32 bit machines and 8 bytes in a 64 bit. x86 is 32bit or 64bit. But this could also be talking about different types and how much each takes up. Then you could consider unaddressable bits so really if the question is ptr it's a loaded question.
   i. Number of bits depends on the machine
   ii. Pointers are take up as much space as int ???
      1. Reasoning its a number...on ilab it was 4 same as int????

b. **What is a smurf type?** (primitive?/pointer?/enumerated?)
   http://www.tutorialspoint.com/cprogramming/c_data_types.htm
   - **Enumerated type:** They are again arithmetic types and they are used to define variables that can only assign certain discrete integer values throughout the program.
   - **The type void:** The type specifier *void* indicates that no value is available.
   - **Basic Types:** They are arithmetic types and are further classified into: (a) integer types and (b) floating-point types.

c. **What good is a () pointer if you can not () it?** (file pointer, function pointer, maybe const pointer?)
   (what good is a **void** pointer if you cannot **dereference** it?) -> i think this is the question
   opaque structs

You can cast it. So like for threads, you technically send in a void pointer param for thread function, but you can cast it back to the type pointer you want

what good is constant pointer if you can not move it? <-sounds legit

a. A constant pointer is a pointer that cannot change the address its holding. In other words, we can say that once a constant pointer points to a variable then it cannot point to any other variable.
b. When you're designing C programs for embedded systems, or special purpose programs that need to refer to the same memory (multiprocessor applications sharing memory) then you need constant pointers.

d. How are smurfs and smurfs alike, if one references only one smurf and the other smurfs? !!!!

(union and structs alike?)

(macros vs structs?) -> can someone do this? how does that even make sense (i have no idea i saw in the notes he wrote something like why is it better to use a struct rather than a macro to define something)

Macro

```
#define INCREMENT(x) x++
```

Structs

```
struct <name of struct> {
        <type 0> <data 0>
        ...
        | <type n> <data n>|
};
```

A struct, on the other hand, has a separate memory location for each of its elements and they all can be used at once.

Union

```
union {
        <type 0> <data 0>
        ...
        | <type n> <data n>|
} <name of union>;
```

With a union, you're only supposed to use one of the elements, because they're all stored at the same spot. This makes it useful when you want to store something that could be one of several types. Union can only store primitive types

e. If p is a smurf? what does smurf mean?(*If p is a pointer, what does p[i] mean?)*

-> maybe about dereferencing?

i. *It means the same as *(p+i), or, in other words, the contents of the object i locations past the one pointed to by p.*

```
p[i] == *(p+i) // (true)
```

f. The code below is supposed to make p point to the second element of g. It doesn't. Fix it so it does.

```
int array[10];
int * smurf;

* smurf = array[1];
```

OR

```
int * smurf = array;
smurf++;
```

OR

```
smurf = &(array[1]);
```

g. What will the smurf of the both x and y be below? Why will they smurf? !!!

```
smurf datastuff
{
        smurf value0;
        float value1;
        smurf value2;
};

smurf;
```

```
smurf otherdatastuff
{
        int value0;
        smurf value1;
        long value2;
};

smurf;
```

i. (sum? / differ?) -> might be about pointers

ii. (union and structs?) -> why will they differ? (one stores all the variables in one memory location under the same type so if it is an int it will be 4 bytes, struct will be a sum of all the types as it's size so if there are 4 ints it will be 16 bytes)

h. What is the benefit of using smurfs to smurf an old type for a new thing? For instance pthread_t is a smurf type that was smurfed. (typedef?)(pthread attribute??)

i. abstract data types? (yeah i think it's abstract data type)

can someone explain a bit more what a ADT is in C ? did we learn this in class?

- ex) stacks & queues, defined by their behaviors instead of the structure used to build them

http://inst.eecs.berkeley.edu/~selfpace/studyguide/9C.sg/Output/ADTs.in.C.html

^^^need to finish this

i. **Why aren't smurf functions smurfed?**
(void,returned)(signal handler, called)(exec, returned)

- Signal handler functions are not called but invoked by the OS on the user's behalf. Could this be it?+
- Exec functions do not return under normal operation-> replaces call to exec in child with new code. Returns int if it fails.
- Void functions do not require a return value, but they can return.

if p is a pointer, write one line of code that does the same thing as smurf without using smurfs.(free)

i. realloc() - be used to deallocate previously allocated memory

```
void *realloc(void *ptr, size_t size);
```

If "size" is zero, then call to realloc is equivalent to "free(ptr)". And if "ptr" is NULL and size is non-zero then call to realloc is equivalent to "malloc(size)".

==========

1. Execution:

==========

Choose and answer 5 of the following 7 parts. Be sure to indicate which 5 you want graded.

A. What is a 'smurf' and how does it differ from a normal C program?

a. (Executable?) is a compiled and linked C program

b. (thread?) A thread is a separate flow of execution in a program. A single program can have multiple threads running within the same address space.

c. process? running executable

B. Right after fork(), but before exec(), what smurfs of the child process smurf the parent?

b. fork() doesn't allocate new memory for the child process, but points the child to the parent's memory, with the instructions to make a copy of a page only if the page is ever written to. This makes fork() not only memory-efficient, but also fast, because it only needs to copy a "table of contents". it forks itself and creates the a process which has the context like init. Only on calling exec(), this child process turns out to be a new process. So why is the intermediate step (of creating a child with same context as parent) -> on an exec call, all the vestiges of a parent are dumped (within the child process) carries over code, variables, file descriptors and file handlers

C. Why is it harder to smurf data between smurfs than smurfs?

(share, threads, processors)(share, user threads,kernel threads)

a. Both stacks share the same heap. When you malloc, you malloc on the heap and if you can access thing on heap from one thread you can access it from all stacks. Can send information back and forth from all your threads. Problem with processes is that it has a different address space, very different to pass information between the process and spend resources of allocating new memory (memory mapped file -> outside the processes).

b. Threads are much faster to allocate, easier to use, easy to communicate between threads. If you want it to be separate and have nothing else mess with it, make a process.

D. What is the difference between smurf and smurf?

a. Process vs Thread
   - Process: On fork(), memory is allocated all over again and is given new address space. Parent & child processes do not share the same heap. Pain in the butt to communicate between (shared memory, mmap stuff).
   - Thread: Fundamentally allows you to compute more than one thing at the same time. OS sees multiple stacks, and threads share the same heap, making it easier for threads to communicate.

b. User thread vs Kernel thread

- User thread: abstraction, As far as the OS is concerned, only the parent process is running. Faster than kernel threads. If the user's thread blocks or breaks, then the entire process and all user threads are blocked or broken. IE. the state of one user thread is the state of all user threads. User is responsible for everything.
- Kernel thread: Kernel can see multiple stacks. Managed by the kernel. The state of the kernel thread is the state of ONLY that thread; it does not affect other threads or the process. Heavier, but more reliable, safety of the kernel.

E. What process has no smurfs? (no parents?)

a. The very first thing that you run. It has no parent. It is the scheduler. It knows how to run on pieces of code, it is done by the OS.
b. Pid 1 is the init process, which is the process that knows how to build processes. Init is the first fork. (deprecated)

F. In what ways are smurfs better than smurfs?

a. It's important to note that a thread can do anything a process can do. But since a process can consist of multiple threads, a thread could be considered a 'lightweight' process. Thus, the essential difference between a thread and a process is the work that each one is used to accomplish. Threads are used for small tasks, whereas processes are used for more 'heavyweight' tasks – basically the execution of applications.

G. In what ways are smurfs better than smurfs?

a. Threads are much faster to allocate, easier to use, easy to communicate between threads. If you want it to be separate and have nothing else mess with it, make a process.
b. (Maybe this could work?) Also threads have access to the heap (dynamic memory-process level access) so that is why it is easier to communicate between threads. For processes, we have to create a memory mapped file outside the processes to allow communication between them or switching data between them.
c. Another difference between a thread and a process is that threads within the same process share the same address space, whereas different processes do not. This allows threads to read from and write to the same data structures and variables, and also facilitates communication between threads. Communication between processes – also known as IPC, or inter-process communication – is quite difficult and resource-intensive

==========

2. Synchronization:

==========

Choose and answer 5 of the following 6 parts. Be sure to indicate which 5 you want graded.

A. Must mutexes be smurfed? Why or why not?

a. initialized or unlocked?

They must be initialized to show what thread / chunk of code is inaccessible. Unlocked after use. Lets multithreaded operations occur and to make sure they don't re-run thread_2 without anything to thread_1 for example.

The real question is if it's not initialized, is it in locked or unlocked state ? -> undefined behaviour.

B. If a thread smurfs a smurf and smurfs, who can smurf?

a. If a thread locks a mutex and the thread crashes who can unlock it?
NO ONE

C. **Presume you have two threads and two mutexes, write a sequence of smurf statements that smurf below:** (create a deadlock?)-> creating a deadlock you would have to create a circular wait here. So like lock stuff in the same order I think

```
int pthread_mutex_unlock(pthread_mutex_t *mutex); //unlock mutex

int pthread_mutex_destroy(pthread_mutex_t *mutex); //destroy locked mutex
```

Thread0:	**Thread 1:**
pthread_mutex_lock(&lockA)	pthread_mutex_lock(&lockB)
pthread_mutex_lock(&lockB)(->i'm waiting for thread 1 to unlock lockB)	
	pthread_mutex_lock(&lockA) (I'm waiting for thread 0 to unlock lockA)
.	.
.	.
.	.

-> CIRCULAR WAIT.

C. If a semaphore is created with initial value 1, wait() and post() will operate much like lock() and unlock() for a mutex.
There is smurf smurfy smurfs smurf and smurf, smurf?

->I don't know what the smurfs are, there are too many. Probably about the different between a binary mutex and semaphore.

Nail it. They are technically the same.

Mutex = binary semaphore.

There is an ambiguity between *binary semaphore* and *mutex*. We might have come across that a mutex is binary semaphore. *But they are not*! The purpose of mutex and semaphore are different. May be, due to similarity in their implementation a mutex would be referred as binary semaphore.

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** ("I am done, you can carry on" kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

D. **A semaphore can be initialized to any non-negative value. What is the utility of initializing it to smurf, which would cause smurf?** (to a negative value/to block it?) (zero/deadlock)

a. A semaphore is an integer variable with the following three operations.
   i. Initialize: You can initialize the semaphore to any non-negative value.
   ii. Decrement: A process can decrement the semaphore, if its value is positive. If value is 0, the process blocks (gets put into queue). It is said to be sleeping on the semaphore. This is the down **(wait)** operation.
   iii. Increment: If value is 0 and some processes are sleeping on the semaphore, one is unblocked. Otherwise, value is incremented. This is the up **(post)** operation.
b. All semaphore operations are implemented as primitive actions (possibly using TSL). In Unix based system, unblocking is done using the signal system call.
c. Semaphores are used by programmers to
   i. ensure mutual exclusion
   ii. send signals from one process to another

d. Example

```
down(S);

    //critical section

up(S);
```

e. If S is negative it would block the crucial section
f. Initialize semaphore to 0 creates a barrier. All threads waiting on semaphore must block until some other process/thread posts. Often used to synchronize computation across multiple threads.

E. Why shouldn't you lock or smurf a mutex smurf?

- Why shouldn't you lock or unlock a mutex within a signal handler? You do not know what thread is running, i.f you lock a mutex and that thread breaks, then that memory space is locked 5EVER.

Semaphore

There are two types of Semaphore

1. Binary Semaphore: Value is either 0 or 1
    a. Often referred to as a mutex <- i thought this was false? http://stackoverflow.com/questions/62814/difference-between-binary-semaphore-and-mutex
2. Counting Semaphore: Value ranges from 0 to N where N can be any integer number

With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:

1. value (of type integer)
2. pointer to next record in the list

A semaphore can be defined as a C struct along these lines:

```
typedef struct {
       int value;
       struct process *list;
 } semaphore
```

When the up is executed a blocked process is woken up. This is done using signals

4 Conditions for deadlock: (one time he mentioned this would be a good exam question)

1. Mutual Exclusion
2. No Preemption
3. **No circular wait** (!!!)<- the only one we can affect
4. Hold and Wait

-> how to prevent a deadlock:
http://www.personal.kent.edu/~rmuhamma/OpSystems/Myos/deadlockPrevent.htm

The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.

-> Whenever you have code that use mutexes, always lock them in the same relative order and unlock in the opposite order (To avoid deadlock)<-- can't u unlock in any order? (this would cause a circular wait in some cases)

Race conditions arise in software when an application depends on the sequence or timing of processes or threads for it to operate properly. As with electronics, there are critical race conditions that result in invalid execution and bugs as well as non-critical race conditions that result in unanticipated behavior

A mutex is essentially the same thing as a binary semaphore and sometimes uses the same basic implementation. The differences between them are in how they are used. While a binary semaphore may be used as a mutex, a mutex is a more specific use-case, which allows extra guarantees:

1. Since only the process that locked the mutex is supposed to unlock it, a mutex may store the id of process that locked it and verify the same process unlocks it.
2. Mutexes may provide priority inversion safety. If the mutex knows who locked it and is supposed to unlock it, it is possible to promote the priority of that process whenever a higher-priority task starts waiting on the mutex.
3. Mutexes may also provide deletion safety, where the process holding the mutex cannot be accidentally deleted.
4. A mutex may be recursive, in that it allows it the process that locked it, to lock it multiple times without causing a deadlock.

d. If a semaphore is created with initial value 1, wait() and post() will operate much like lock() and unlock() for a mutex. There is smurf smurfy smurfs smurf and smurf, smurf?

Differences between a semaphore initialized with a value of 1 and a mutex?

e. A semaphore can be initialized to any non-negative value. What is the utility of initializing it to smurf, which would cause smurf?

What is the utility of initializing it to [0], which would cause [deadlock || barrier]?

On barrier: Used to synchronize computation across multiple threads.

==========

3. Files:

==========

Answer all of the following 4 parts.

A. The smurf function moves the file pointer, smurf smurf?

a. fread/fseek (also what is the difference between lseek, fseek, read and fread ?)

i. lseek is used with open/close/read/write system calls.

ii. fseek is used with fopen/fgets/fread/fputs/fprintf etc. buffered I/O C-library calls They're a higher-level system of I/O that does its own buffering, and calls the lower-level calls to do the actual I/O.

b. What is lseek? Move/go to a number of bytes within a file

B. What happens if you smurf past the smurf?

a. What happens if you read past the (EOF)?

i. It returns the number of bytes actually read. It will return a number greater than zero until the read cursor has reached EOF. At the point any subsequent call will return 0. This is not the case for lseek which will happily read past the end of file and increase the size of the file

C. Can more than one process smurf the same file in smurfy smurf?

a. Can more than one process (open/read/write) the same file in ()?

D. Can more than one process smurf the same file in smurfy smurf?

a. Can more than one process (open/read/write) the same file in ()

-> i think threads if kernel level, more than one thread can access the same file at the same time. but at user level if a thread needs to do something with a file is stops all the other threads to do whatever it needs to do. i think in terms of processes, more than one process can read/open a file at a time but only one at a time can write to the file.

==========

pipe creates a pipe just before it forks one or more child processes, the pipe is then used for communication either between the parent or child processes, or between two sibling processes

If one kernel thread wants to write a file, it goes off and does it, it does not affect the other threads.

What is the difference between file descriptors and file handlers?
A file descriptor is a data structure that is used to access a file.
A file handler indicates a position to reference within the file. (ex. read 10 bytes, read another 10 bytes. this is the file handler, the position of the
file handler in the parent process will be in the same position)
Do not use file descriptors or handlers directly after you fork

Kernel vs User Threads:

Kernel threads are visible by the kernal, kernal can see and schedule them. User threads are internal to the process, kernel can not see them. If one kernel thread wants to write a file, it goes off and does it, it does not affect the other threads. User thread is an abstraction supplied by your own code, kernel is not aware of these threads and does not schedule them, as far as the OS is concerned there is only one thread. If your thread wants to write a file, OS gives process time to run and if the process wants to write file, none of the other user threads will run. If you block on some I/O everything else stops until it is done running. Be careful with user threads, if one thread blocks or breaks something, the entire processes is stopped. Kernel threads can be scheduled threads to run independently of one another. User threads are very fast, just moving program counter to different places in the code. Kernel threads are slow, OS takes time to run schedule but are more robust. (change pthread_scope to switch between kernel and user) User threads not used that often.

Strange functions

1. Exec - called once, normally does not return (replaces the call to exec in the child with new code) if it returns : **error** (BAD JUJU)
2. Fork - called once, returns twice (parent- childPID and child- 0)
3. Signal Handlers: is not called, but does return (because it is invoked by the OS on your behalf)

Good luck everyone