**Translating**

1. **What does the function smurf() do?** **What function does the opposite of smurf()?**
   1. **What does the function (open()/close()) do? What does the opposite of (open()/close()) do?**
      1. 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
   2. What does the function pthread\_mutex\_lock()/pthread\_mutex\_unlock()/pthread\_mutex\_trylock()) do? What function does the opposite? -- **I would expect this question to be about threads more than I would expect to just be about open()/close()**
      1. They return zero on success; that is, if they lock or unlock the mutex successfully. Otherwise, they block the thread until they do. The difference between \_lock() and \_trylock() is that \_trylock() returns immediately, rather than blocks if it can’t get ownership of the mutex
   3. What does the function int pthread\_create(pthread\_t \**thread*, const pthread\_attr\_t \**attr*, void \*(\**start\_routine*) (void \*), void \**arg*);
      1. Pthread\_create: creates a new thread which begins at the start routine function. This start routine function is passed a sole void star argument arg.
      2. Pthread\_detach: detaches the thread, which is the opposite of “joining” a thread (to “join” a thread is to wait until it’s done and catch its return value and stack)
   4. What does the function sem\_wait() do? How does this differ than sem\_post
      1. Sem\_post increments the value of a semaphore will sem\_wait decreases the value. The semaphore is blocked when it is zero. A semaphore can never be negative.
   5. What does the function malloc () do ? What is the opposite?
      1. Memory is set up as a bunch of mementries. Malloc asks the OS for free space in the heap and the OS returns a pointer to free space. Here, malloc needs the size of allocation, next block of free memory, and special code to fill out the mementry. Malloc then returns the pointer to the end of this info.
      2. Free makes the memory free by setting a isFree flag in the mementry.
   6. What does accept() do ? What is the opposite of this?
      1. The accept() function shall extract the first connection on the queue of pending connections, create a new socket with the same socket type protocol and address family as the specified socket, and allocate a new file descriptor for that socket.
   7. Can a thread aquire more than one mutex lock?
      1. Yes it is possible that a thread is in need of more than one recources such as a mutex lock
   8. Can a mutex be locked more than once
      1. A mutex is a lock, only one state is associated with it(locked/unlocked) however a recursive mutex lock can be locked more than once
   9. What happens if a non-recursive mutex lock is locked more than once?
      1. Deadlock will occur because if a thread is in mutex lock and tries to lock the mutex again, it will be in the waiting list of that mutex resulting in a deadlock
2. **How does a file smurf differ from a file smurf?**
   1. **How does a file descriptor differ from a file handler (File \* pointer) ?**
      1. 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.
      2. 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.
   2. **How does child process differ from a parent process?**
      1. 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
   3. How does a thread differ from a process?
      1. 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.
      2. 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.
3. **What does smurf() do?**
   1. what does sem\_init(&mutex, 0, 1) do
      1. Sets up a binary semaphore
   2. What does htons () do?
      1. Everyone's computer has different endianness. So when entering a port number, of another computer you want to connect to, you need it to be in Network Byte Order. To do this we cal htons (host to network short) which translates numbers to Network Byte Order.
   3. What does accept () do ?
      1. Accept() accepts incoming connections and returns a socket number for the process to talk to the new connection.
4. **If smurf is all you need to identify a smurf, what is the use of smurfs?**
   1. f pid is all you need to identify a process/thread, what is the use of ppid?
      1. In the instance the process becomes a zombie process a good way of killing it is by killing the parent which can be identified by the ppid
   2. If IP address is all you need to identify a machine, what is the use of port?
      1. Ports get you to the application on that machine that can understand/ wants the message
   3. **Why is smurfing necessary for any smurf application?**

**Executing**

1. Right after smurf(), but before smurf(), what smurfs of the smurf are smurf?
   1. Right after fork(), but before exec(), what attributes of the child process are different?
      1. PId is different as well as PPid, address space and return value of fork()
2. **What process has smurf?**
   1. **What process has a PID of 0**
      1. **the init has a PID of 1, scheduler has a PID of 0**
3. **Why is it harder to smurf between processes than smurfs?**
   1. Why is it harder to share information(memory) between processes than threads?
      1. 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
4. **In what ways are smurfs better than smurfs?**
   1. In what ways are kernel threads better than user threads? (Or vice versa)
      1. Kernel can schedule multiple threads of the same process on multiple processors, Kernel routines can be multithreaded
   2. In what ways are threads better than processes?
      1. Far less time to create a new thread than a new process
      2. Less time to terminate a thread than a process
      3. Less time to switch between two threads within the same process than to switch between processes
      4. Threads can communicate via shared memory - processes have to rely on kernel services for IPC
5. **What is a smurf?** 
   1. Thread
      1. Function that runs independently of the calling function
      2. Can run at the same time as other code
      3. Separate flow of execution in a program
   2. Zombie process
      1. A child process that has returned but was never wait()-ed on by the parent. Wait() is pretty much the parent saying that it is done with the child and doesn’t need its data anymore. This means that the child process is done doing what it needs to do but its resources can’t be reclaimed. Zombie processes whose parents return or disappear have init as their new parent, which then wait()-s on them and frees the resources.
   3. Orphan process
      1. Parent dies before child. Immediately adopted by init process.

Syncronizing

1. **Which of the four smurfs(deadlock conditions) can we avoid with smurfs(mutexes)(timeouts)**
   * 1. Mutual exclusion- Can be avoided by: Using different variables or not using mutexes
     2. Hold and wait or partial allocation-Can be avoided by: allowing the holds to time out
     3. No pre-emption- Can be avoided by: A scheduler or moderator of the system
     4. 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

1. Why is it impossible to completely avoid smurf in the dining smurf problem using smurfs?
   1. Why is it impossible to completely avoid (starvation?/deadlock/livelock?) in the dining (philosopher) problem using (semaphores?/mutexes?/For starvation ->timeouts?)?
2. If a thread smurfs a mutex and smurfs, who can smurf?
   1. **if a thread locks a mutex and crashes, who can unlock it**
      1. No one can unlock it. Only the thread that owned the mutex can unlock it. This case is a deadlock
   2. if a thread creates a mutex and locks, who can access shared resources?
      1. Only that specific thread
3. **Why shouldn't you smurf a smurf twice?**
   1. Why shouldn’t you unlock/lock a mutex twice?
      1. 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.
4. **What are the differences between smurfs and smurf variables:**
   1. What are the differences between (mutexes) and (condition?) variables:
      1. 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.
      2. 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 synchronizing primitive)

Signaling

1. What is the difference between smurf and \_smurf?
   1. What is the difference between exit and \_exit atexit()?
      1. The basic difference between exit() and \_exit() is that the former performs clean-up related to user-mode constructs in the library, and calls user-supplied cleanup functions, whereas the latter performs only the kernel cleanup for the process."
2. **How do smurfs and smurfs differ?**
   1. How do alarms and signals differ?
   2. how do forks and exec differ
   3. how do signal() and sigaction() differ?
      1. signal() - doesn’t necessarily block other signals
      2. sigaction() - can block other signals until current handler returns
   4. how do parent and child processes differ?
   5. how do relative and absolute paths differ?
      1. Relative - reference to current directory
      2. Absolute - reference to root directory
   6. how do signal and signal handlers differ
      1. Signals - indicates some situation that must be dealt with before the code continues
      2. Signal handlers - reroute certain signals to your own code
3. **Why is it dangerous to smurf a smurf inside a smurf handler in a smurf that smurfs?.**
   1. Why is it dangerous to lock a mutex inside a signal handler in a thread(?) that signals(?)?
      1. 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.
   2. Why is it dangerous to malloc a space inside a signal handler in a \_\_\_\_\_
      1. Since signal handler can be called at any time, if there is another call to malloc, then:
      2. Deadlock: malloc cannot get heap lock
   3. Why is it dangerous to free a space inside a signal handler in a process that mallocs.
      1. Signal Handler:You do not want to access global variables when you are in signal handler (or call methodis) 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.