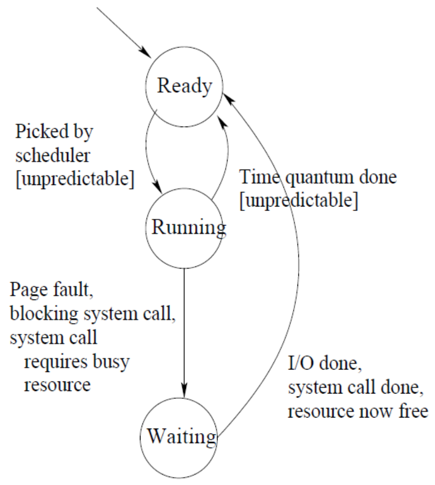
1. Explain the difference between a process and a thread.
    - A process is a program in execution and its resources, whereas a thread is an activation stack, a series of method calls and variables, and each thread has its own execution stack. Use threads when you need many schedulable units — are faster to create, destroy, switch; Much shared data
2. Explain when it is more appropriate to use a process instead of a thread. When is it more appropriate to use a thread instead of a process?
    - A process is more appropriate for containing all of a program's resources and threads for a single instance of the program. Threads are more appropriate for dividing tasks within a program, especially if collaboration is necessary, because threads share an address space while processes do not.
3. Draw the state transition diagram for a process. Label the states and indicate examples of events that cause transitions among states. Show all transitions.


Ready
Picked by scheduler [unpredictable]
Time quantum done [unpredictable]
Running
Page fault, blocking system call, system call requires busy resource
I/O done, system call done, resource now free
Waiting

4. Write C code to create a new UNIX process. Your code should execute function A() if the attempt to create the new process fails. If the attempt succeeds, your code should execute function B() in the parent process and function C() in the child process.

```
void main(){
      int n;
      if((n = fork()) < 0){
            A();
      }else if(n > 0){
            B();
      }else{
            C();
      }
}
```

5. Explain how to wait for input from more than one file descriptor simultaneously.
    - The select() function in C can be used to wait for multiple file descriptors. A timeval struct is used to set a timeout for how long to wait for the file descriptors. When a file descriptor is ready, the execution continues and one or more of the file descriptors can be used.

6. Explain what it means to “block,” “ignore,” or “handle” a UNIX signal. Which system call is used to block a signal? To ignore or handle a signal?
    - To “block” a signal means to block execution of the program until the signal is unblocked. To “ignore” a signal means to receive a signal but do nothing with it, and to “handle” a signal means to call a function or do some task when the signal is received. sigprocmask() is used to block a signal, and sigaction() is used with SIG_IGN or a function to ignore and handle a signal, respectively.
7. Some C library functions should not be called by a signal handler. Explain what aspects of a function make it one that shouldn’t be called by a signal handler and name one such function.
    - Only “signal safe” functions should be called by a signal handler. These are functions which do not write to any static or global memory, and are considered “reentrant” - if the signal is received again, the function should be able to start again from the beginning in the middle of its execution, and then finish its original execution with no adverse effects. It should write only to local variables, but can read from anywhere. One example of a function that is not signal safe is malloc.
8. What does it mean for a section of code to be “reentrant”?The non-reentrancy of some functions can be spotted based only on a description of the function’s interface. Explain how to do so.
    - Reentrant code is code that can have its execution halted in order to be “reentered” from the start by a handler. When this reentered call is finished, it will return to where it halted and continue executing. This reentrance should have no adverse effects. To achieve this, the function cannot write to any static or global variables. So, non-reentrancy can sometimes be easily detected from an interface for a method if the method specifies that it returns a static/global variable or sets errno upon exit.
9. Explain how hardware interrupts are used by the operating system to implement preemptive scheduling.
    - Preemptive scheduling uses hardware interrupts to stop the execution of a thread, and resume the execution of a different thread, usually after a certain amount of time. It differs from non-preemptive scheduling in that it does not rely on the program to handle thread switches, but rather uses a scheduling algorithm to determine which thread should run next and for how long. Preemptive scheduling expects programs to synchronize access to critical sections of the code, which are sections that more than one thread should not be executing at a given time.
10. Write C code to create a new Pthread, wait for it to finish executing, then collect its result. The thread’s start function should accept 3 arguments: an integer, a string, and a floating point number. The thread should return an integer.

```c
#include <pthread.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <ctype.h>

struct arg_t{
       int n;
       char* s;
       float d;
};
void* start(void* args){
       struct arg_t *a = args;
       return &(a->n);
}
```

```c
void main(){
       int n;
       void* y;
       pthread_t st;
       struct arg_t m;
       m.n = 4; m.s = "hey"; m.d = 3.0;
       if((n = pthread_create(&st, NULL, start, &m))){
              printf("Thread creation error.");
              exit(0);
       }
       if((n = pthread_join(st, &y))){
              printf("Thread join error.");
              exit(0);
       }
       printf("Result collected: %i",*((int*)y));
}
```

11. Explain how to pass an unbounded amount of data as the argument to a Pthread even though its start function accepts only a single "void *" argument.
    - You can make a struct containing space for multiple types of arguments, and then pass a pointer to the struct in pthread_create.
12. Explain how a "zombie" UNIX process is created. Explain why Pthreads does not include the zombie concept.
    - A zombie process occurs when a process has terminated but is still waiting for its parent to read its exit status using wait(). Zombie processes can last a long time if the parent exits without waiting for the child to finish. Pthreads do not have this concept, because when the main thread exits, the entire program is finished, so all threads exit as well - unlike child processes, they do not live longer than the parent thread.
13. Explain how to make a function thread-safe.
    - The function must be reentrant. It should not write to any static or global variables without first getting a lock. It should not return a pointer to a static variable. It also should not write to a shared errno. Thread-safe means that any number of threads should be able to execute this function simultaneously without side effects.
14. Explain how to make a function cancel-safe.
    - Sometimes a function being cancelled at the wrong time can lead to half-written data structures, or unfinished updating of sensitive information. So, a solution to this, barring simply not updating sensitive things, is to set the cancellation status to "deferred". The thread can then finish its sensitive operations, and cancel itself later, when it won't be detrimental to do so.
15. What is the result if an N-thread process forks? That is, how many threads does the child process have and what code is each executing when the child process begins?
    - The fork() results in a single threaded process which is an exact copy of the thread which called fork(). So, the child process duplicates the address space of the current thread, as well as any state information created by other threads.
16. What happens to each of a process's threads when a system call in the "exec" family is invoked?
    - All existing threads are terminated, and a new main thread is created to run the executable specified as a parameter to exec().

17. If a signal is generated by hardware or software exception, to which thread in a multi-threaded process is it delivered? Give an example of one such signal type.
    - If a signal generated by a hardware or software exception occurs, it is delivered to the thread responsible for creating that exception. Examples of this are SIGILL or SIGSEGV.
18. If a signal is generated by an external process, to which thread in a multi-threaded process is it delivered?
    - Since the target of the signal is just the process as a whole, the signal is delivered to an arbitrary process that does not have that signal blocked. Examples of this are SIGCHLD or SIGUSR1
19. Give an example of a race condition. Explain how the race condition may lead to incorrect results.
    - An example of a race condition is in the code "foo = foo + 1", where foo is 10 before hand. If multiple threads perform this action to this variable in this way, it could be scheduled like so:
        1. thread A reads 10 from foo, then descheduled
        2. thread B reads 10 from foo, then descheduled
        3. thread A adds 1 to 10, which is 11.
        4. thread A stores 11 in foo, then descheduled
        5. thread B adds 1 to 10, which is 11.
        6. thread B stores 11 in foo.
    - Now even though two threads were supposed to each add 1 to the shared variable foo, instead of a total of two being added, the race condition makes it so that only one is added. And the result could be either 11 or 12, depending entirely on how the scheduling is done.
20. Explain how preemptive scheduling may cause race conditions to occur.
    - Preemptive scheduling can cause race conditions if critical sections or shared variables are not accessed atomically. Since preemptive scheduling uses interrupts to stop a thread's execution, if it happens to interrupt a thread during a section that accesses shared variables, it can change the expected output to be a result of the scheduling algorithm's choice.
21. Which statements of a high level programming language (e.g., C or Java) may a programmer correctly assume to be atomic on typical computer architectures? Which assembly language statements?
    - Single word reads and writes are typically atomic. Getting and dropping a lock is usually atomic, because it's usually just testing and setting or just setting a one word space in memory. Data type atomically read and written in Java - int, short, char. Typically same for C, but depends on architecture. At the assembly level, linked loading and store conditional (MIPS) exists to allow for a form of compare and swap (x86/x86_64). Compare and swap of single words (machine specific) are also usually allowed (Compare and Swap on Motorola 68K). Weaker form of CAS is Test and Swap (TAS)..
22. Explain the major advantage and major disadvantage of non-preemptive scheduling.
    - The major advantage of non-preemptive scheduling is that it does not unschedule threads at unsafe times - so race conditions don't happen. The major disadvantage is that the responsibility for switching threads falls on the threads themselves. So the OS trusts the threads to schedule themselves, and the threads have to trust each other to "yield" at the right times.
23. Explain what mutual exclusion is. What problem does it solve?List and explain the necessary conditions for a satisfactory solution to mutual exclusion.

- Mutual exclusion is the requirement of ensuring that no two processes or threads are in their critical section at the same time. It solves the problem of more than one process or thread attempting to access something in shared memory at the same time and miscommunicating information.
    i. No two threads may be simultaneously inside their critical sections. (partial correctness)
    ii. No thread should wait arbitrarily long to enter its critical section (liveness - freedom from starvation)
    iii. No thread stopped outside its critical section should block other threads.
    iv. No assumptions about relative speeds of threads or number of CPUs
    v. No knowledge about number/identity of other threads pre-execution. (Threads can learn about other threads during execution).

24. “Peterson’s Solution” and “Strict Alternation” provide mutual exclusion assuming only atomic single-word read and write operations. Explain how these solutions are inadequate.
    - Strict alternation - alternate between two threads using a boolean to tell whose turn it is. This is not adequate because a thread outside of its critical section can block other threads - one thread may need to access a critical section more often than the other so it violates condition 3. It also violates condition 5, because it has to have 2 threads, and the number must be known at the time of coding. (The idea could be expanded to more than 2 threads, though.
    - Peterson’s solution - instead of a boolean to tell whose turn it is, this checks whether each thread is interested, and if both threads are interested, whichever sets turn_to_wait last will be the thread that has to wait. It does not violate condition 3, but it still requires the knowledge of 2 threads beforehand, so it violates condition 5.
25. Explain why the disabling of interrupts is no longer a viable synchronization technique, even inside operating systems.
    - Disabling interrupts works by simply not allowing a thread to be interrupted until the critical section is complete, and then the interrupts are reenabled. This is dangerous, because if the interrupts are not reenabled, it could crash the system, and that shouldn’t be something that’s a responsibility of the programmer. Also, this requires privileges, and while it ensures that no two critical sections are run at the same time, it also ensures that NOTHING ELSE is run at the same time (not every thread always needs to be in its critical section).
26. Explain each of these terms: liveness, starvation, livelock.
    - liveness - the algorithm always does something (liveness + partial correct = correctness)
    - partial correctness - the algorithm either does the right thing or does nothing at all
    - starvation - a thread never or very rarely gets its turn to do anything, ex: largest job first can make the smallest jobs never get to run
27. What does it mean to “busy-wait” or “spin” waiting for a lock? What is disadvantageous about this technique?
    - A process will be looping to check if a condition is true to take the lock. Repeatedly checking to see if they can obtain the lock wastes CPU time. A better solution is to have the process sleep or be suspended until the lock is available, and then woken up or alerted, to prevent the wasting of CPU resources.
28. Explain how to use the “load-linked” and “store-conditional” machine instructions to implement a lock, and explain why this solution is suited to a shared memory multiprocessor.

- The load linked atomically loads a memory location into register then marks location with CPU's ID (possible to mark with > 1 CPU IDs). 'LL r1, lock' is the load-linked machine instruction using register r1 and the lock.
- The store-conditional atomically: checks if memory location is marked by this CPU then store new value and remove **all** marks, else do nothing. 'SC r2, lock' is the store-conditional machine instruction using register r2 and the lock.
- Any changes to memory are prevented if another CPU has already changed changed the value of the memory location. In the example with the lock, if two CPUs read the value of the lock and attempt to set it, CPUs that try to set the lock following the first will have to read it again before setting it. This makes it suited for shared memory multiprocessor.

29. Explain what a semaphore is and how it is used to provide mutual exclusion.
    - A semaphore is essentially an integer. It uses two atomic operations P() and V() (decrement and increment). P() tries to decrement the semaphore. If the semaphore is already zero, it puts the thread to sleep until it is not 0. V() adds 1 to the semaphore, and also wakes up one sleeping thread if any exist. Used like:
```
int main() {
  P();
  crit-section;
  V(); }
```
30. Explain the primary disadvantage of the semaphore concept.
    - The programmer has to keep track of the calls to wait and to signal the semaphore. If they're not done in the correct order, it will cause deadlock. Also, the programmer has to somehow convert the idea behind their code into the terms of a single integer.
31. Explain why it is not a good idea for more than one thread to share a single file descriptor.
    - If multiple threads share a single file descriptor, then multiple threads will all have the ability to write to it. Since these write are happening at the same time, the bytes written may be interleaved. To get around this requires explicit locking of the file descriptor. Note that depending on the OS and machine architecture, it may be safe to write certain amounts of memory without worrying about interleaving.
32. Explain what is the "monitor invariant"? Name the three major operations on a monitor and explain their operation.
    - Monitor invariant - no more than one monitor procedure will run at any given time.
    - Wait: atomically releases lock then waits on condition. When thread is reawakened, thread requests the lock. Wait returns after it reacquires the lock.
    - Broadcast - enables all waiting threads to run. When broadcast returns, lock is still held.
    - Signal - enable one waiting thread to run. When signal returns, lock is still held.
33. Why should an application-specific condition—failure of which to be satisfied would lead a monitor procedure to call wait—be tested by a while loop rather than tested by an if statement?
    - A while loop should be used because it is possible for a thread to be woken, but for the condition to not be met. So, if a thread is woken, it needs to check the condition again before moving on to the next part of the code.
34. How is a "recursive" Pthread mutex lock different from a standard Pthread mutex lock?
    - Re-locking a mutex in a standard pthread mutex lock causes deadlock, and re-unlocking or unlocking an unlocked mutex will cause undefined behavior. In recursive mode, a count is kept which monitors how many times a thread has locked a mutex, and it must unlock it that many times before the mutex is available to other threads. Unlocking a mutex not owned by the current thread, or unlocking a mutex which was never locked also results in undefined behavior in recursive locks.

- The difference between a recursive and non-recursive mutex has to do with ownership. In the case of a recursive mutex, the kernel has to keep track of the thread who actually obtained the mutex the first time around so that it can detect the difference between recursion vs. a different thread that should block instead. As another answer pointed out, there is a question of the additional overhead of this both in terms of memory to store this context and also the cycles required for maintaining it.

35. What is dangerous about cancelling a Pthread? How can these dangers be avoided or mitigated?
    - When a Pthread is cancelled and the cancel function returns, the thread is added to a queue to be cleaned up. There is no assurance that when the function returns the thread has been cleaned up or the thread has left work partially completed. It can be mitigated by making the thread deferred meaning it can only be cancelled at predefined points where it checks if it is cancelled.
36. What is “fairness” (wrt accessing a resource, such as a lock)?
    - A fair lock favors the thread that has been waiting the longest.
    - ReentrantLock has an optional fairness parameter. When set true, locks favor granting access to the longest waiting thread. Otherwise, there is no guarantee to any particular access order.
    - Can cause a drag on performance, but it prevents starvation
37. What is the purpose of having multi-mode locks? Give an example of a moded lock.
    - It allows multiple threads to hold a lock in compatible modes. For example, multiple threads can read from a variable (read & read), or multiple threads can read OR a single thread can write (read & write).
38. Consider the Runnable and Callable Java interfaces. Explain what they have in common and what are their differences.
    - The Runnable and Callable interfaces in Java are both used for executing instances which potentially need to be executed in another thread. The Callable interface allows for returning a result and throwing Exceptions, but the Runnable interface allows for neither.
39. Explain how the Java Future interface is useful.
    - The Future interface represents the eventual result of an asynchronous computation. It is useful because the Future object can be passed around, and when the computation is finished, the Future object will contain the result. It can also be used to wait for the result, by blocking the calling thread of Future.get().
40. Suppose that int foo(String a, float b) is a Java method that should be executed in a separate thread. Write Java code to create a new thread, execute foo with the proper arguments, and retrieve foo’s result.

```
class CallThread implements
Callable<int> {
        private String a;
        private float b;
        public CallThread(String a,
        float b) {
                this.a = a;
                this.b = b;
        }
        public int call() {
                return foo(a, b);
        }
}
```

```
public void main(String[] args) throws
Exception {
        FutureTask<int> task = new
        FutureTask<int>(new
        CallThead(PUT_ARGS_HERE));
        Thread t = new Thread(task);
        t.start();
        int result = task.get();
}
```

41. In Java what is an “object lock”? How is it useful for controlling concurrency?
    - An object lock is a hidden lock inside every object, such that upon entering any synchronized method, this lock is obtained, and dropped upon return/exception from a synchronized method.
    - It is useful because synchronized methods can be used to access critical sections and global variables, without explicit locking.
42. Contrast the functionality provided by a monitor versus the functionality provided by Java’s synchronized keyword.
    - Monitors allow for waiting on a condition while the synchronized keyword does not. With the synchronized keyword, it’s essentially just locking and dropping the lock on a mutex before and after a function - a monitor should be used if you only want to get the lock if a condition is true, like an queue becoming empty.
43. Evaluate the functionality provided by Java’s synchronized keyword as a solution for mutual exclusion.
    - When static methods are made synchronized, it allows for mutual exclusion between objects of this class. This is because no two objects can be executing a static synchronized method at the same time. It works for any number of threads and any CPU/speed assumptions. Also, since the lock is dropped upon exit of the function, it prevents other threads from waiting for a thread which is not in its critical section. Lastly, threads will not wait an arbitrarily long time to run, because essentially a random waiting thread is picked to run the synchronized method.
44. Explain the usefulness of a synchronized block (block only, not an entire method).
    - The main downside of using synchronized methods is that it blocks other threads via holding of the object lock even when not touching shared state. The benefit of using synchronized blocks is that the locking is still handled for you like a synchronized method, however it will only hold the object lock for the portion you specify. This way you can avoid blocking when doing a thread safe but long computation, and only block when updating a variable.
45. Write Java code for a synchronized block (block only, not an entire method). Write “BODY” to indicate where the statements of the block’s body would be.

```
        synchronized (this){
                BODY
        }
```

46. Write Java code that gets a lock using Java’s ReentrantLock class and is guaranteed to drop the lock even if an exception is thrown between lock-obtaining statement and the lock-dropping statement.

```
class Test {
        private ReentrantLock lock = new ReentrantLock();
        public void doesSomething() {
                try {
                        lock.lock();
                        DO_SOMETHING_HERE
                } catch (Exception ex) {
                        HANDLE_EXCEPTION HERE
                } finally {
                        lock.unlock();
                }
        }
}
```

47. Write a monitor solution for the producer/consumer problem using Pthreads pthread_mutex_* and pthread_cond_* functions. Do the same using Java's ReentrantLock and Condition classes.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <pthread.h>
#include <string.h>
#include <semaphore.h>
#include <unistd.h>
static pthread_mutex_t monLock; /* monLock protects all the following ... */
#define BUFFER_SIZE 128
static char *buffer;static pthread_cond_t stringAvailable;
static pthread_cond_t spaceAvailable;
static int outptr;
static int inptr;
void *drainThread(void *outfile) {
       char *f = (char *) outfile;
       FILE *out;
       if ((out = fopen(f, "w")) == NULL) {
              fprintf(stderr, "could not open output file %s\n", f);
              exit(1);
       }
       do {
              char local[BUFFER_SIZE];
              pthread_mutex_lock(&monLock);
              /* while there is no new string, wait */
              while (buffer[outptr] == 0)
                     pthread_cond_wait(&stringAvailable, &monLock);
              strcpy(local, &buffer[outptr]);
              printf("drain thread: read [%s] from buffer\n", local);
              if (strcmp(local, "QUIT") == 0) {
                     pthread_mutex_unlock(&monLock); /* unlock before returning!
                     */
                     return NULL;
              } else {
                     if (fwrite(local, 1, strlen(local), out) != strlen(local))
                            fprintf(stderr, "failed to write line %s to output\n",
                     local), exit(1);
              }
              outptr = outptr + strlen(local) + 1;
              pthread_cond_signal(&spaceAvailable);
              pthread_mutex_unlock(&monLock);
       } while (1);
       return NULL;
}
int putLineIntoBuffer(FILE *in) {
       char *line;
       size_t ignored;
       ssize_t nread;
       line = NULL;
       nread = getline(&line, &ignored, in);
       if (nread != 1) {
              strcpy(&buffer[inptr], line);
              inptr = inptr + nread + 1;
              printf("fill thread: wrote [%s] into buffer\n", line);
              free(line);
       } else {
              strcpy(&buffer[inptr], "QUIT");
              inptr = inptr + 5;
              printf("fill thread: wrote QUIT into buffer\n");
       }
       return (int)nread;
}
struct args_t {
       char *file;
       useconds_t usecs;
};
void *fillThread(void *args) {
       struct args_t *a = (struct args_t *) args;
       char *f = a>file;
       useconds_t sleepTime = a>usecs;
       FILE *in;
       int nread;
       int previous_outptr;
       if ((in = fopen(f, "r")) == NULL) {
              fprintf(stderr, "could not open input file %s\n", f);
              exit(1);
       }
       /* fillThread must act first ... write 1st line before waiting for signal
       */
       pthread_mutex_lock(&monLock);
       nread = putLineIntoBuffer(in);
       previous_outptr = outptr; /* remember where outptr was when last line added
       */
       pthread_mutex_unlock(&monLock);
       do {
              pthread_mutex_lock(&monLock);
              /* while last string hasn't been read, wait */
              while (previous_outptr == outptr)
                     pthread_cond_wait(&spaceAvailable, &monLock);
              nread = putLineIntoBuffer(in);
              previous_outptr = outptr; /* remember where outptr was when last
              line added */
              pthread_cond_signal(&stringAvailable);
              pthread_mutex_unlock(&monLock);
              usleep(sleepTime);
       } while (nread != 1);
       (void) fclose(in);
       return NULL;
}
int main(int argc, char *argv[]) {
       int rc;pthread_t tid;
       struct args_t fillArgs;
       /* check command line arguments */
       if (argc != 4) {
              fprintf(stderr, "usage: %s infile outfile sleeptime\n", argv[0]);
              exit(1);
              }
       /* initialize shared buffer */
       if ((rc = pthread_mutex_init(&monLock, NULL)) != 0)
```

```
        fprintf(stderr, "mutex init failed: %s\n", strerror(rc)), exit(1);
    if ((rc = pthread_cond_init(&spaceAvailable, NULL)) != 0)
        fprintf(stderr, "spaceavailable condition variable init failed:
        %s\n", strerror(rc)), exit(1);
    if ((rc = pthread_cond_init(&stringAvailable, NULL)) != 0)
        fprintf(stderr, "stringavailable condition variable init failed:
        %s\n", strerror(rc)), exit(1);
    if ((buffer = calloc(BUFFER_SIZE, 1)) == NULL)
        fprintf(stderr, "buffer allocation failed\n"), exit(1);
    outptr = 0;
    inptr = 0;
    /* start thread that takes from buffer & writes to output file */
    if ((rc = pthread_create(&tid, NULL, drainThread, (void *)argv[2])) != 0)
        fprintf(stderr, "thread create failed (%s)\n", strerror(rc)),
        exit(1);
    /* this thread reads from input file & puts into buffer */
    fillArgs.file = argv[1];
    fillArgs.usecs = atoi(argv[3]);
    (void) fillThread((void *)&fillArgs);
    /* wait for other thread to terminate then free resources */
    (void) pthread_join(tid, NULL);
    (void) pthread_cond_destroy(&stringAvailable);
    (void) pthread_cond_destroy(&spaceAvailable);
    free(buffer);
    return 0;
}
```

- Explanation: The problem describes two processes, the producer and the consumer, who share a common, fixed size buffer used as a queue. The producer's job is to generate a piece of data, put it into the buffer and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer) one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer

48. Suppose that a Java thread is using a read-write lock and already holds a lock in read mode. Explain the steps involved to correctly “upgrade” the lock to write mode.
    - It is not OK to update a read lock to a write lock. You should drop the read lock, then attempt to acquire the write lock. This is because only one writer can exist at a time, but multiple readers can exist, so if multiple threads tried to upgrade it would be possible to have multiple writers.
    - To downgrade from a write lock: 1. acquire write lock 2. acquire read lock. 3. drop write lock
49. Java provides a variety of thread pools. Explain how “fixed” and “cached” thread pools operate. What is the relative advantage of each over the other?
    - Fixed thread pools have a fixed number of threads. New tasks will either be assigned to a free thread, or delayed until a thread is available. The advantage is that at most n threads will be running at a given time, so with limited resources it provides assurance that it will only use n threads.
    - Cached thread pools create new threads when a task comes in, unless an older thread is free to use, in which case it reuses that one. Each thread is discarded after being unused for 60 seconds. Good for programs which use a lot of short-lived asynchronous tasks, because it will do them simultaneously and then finish.

50. Java provides a variety of synchronized Collection classes. What is the major advantage of using such classes? What is the major disadvantage?
    - Synchronized Collection Classes: Java Collections Framework
        i. set of typeparameterized classes and interfaces for storing groups of objects○ 14 Interfaces
    - many classes that implement them each class is a data structure
        i. set, list, hash table, queue
    - java.util contains all collection classes
        i. [not in .concurrent] - Vector, Hashtable
    - java.util.concurrent contains some but not all thread safe collection classes
        i. ArrayBlockingQueue, ConcurrentHashMap, ConcurrentLinkedQueue, Concurrent SkipListMap, ConcurrentSkipListSet, CopyOnWriteArrayList, CopyOnWriteArraySet, LinkedBlockingDeque, LinkedBlockingQueue, PriorityBlockingQueue,
    - Advantages - thread safe collection classes help in some synchronization situations
    - Disadvantage - there is no single convenient place to find all concurrent collections
    - classes, thread safe collection classes impose overhead (locking) all the time
51. Explain how to use a "synchronization wrapper" to convert an unsynchronized Collection class into a synchronized class.
    ```
    List<E> syncArrayList = Collections.synchronizedList (new ArrayList<E>());
    Map<K, V> syncHashmap = Collections.synchronizedMap( new HashMap<K, V>());
    ```
    - Just convert existing unsynchronized classes using the corresponding methods in the Collections class.
52. Java's volatile keyword is sometimes said to be useful for concurrent programming. Explain why. What is the main danger of using volatile? What programming approach can be used instead of using volatile?
    - Volatile can be used on atomic variables, such as int, char, and short. Volatile allows the variable not to be cached - it will always be read and written to from memory. Sometimes this can prevent the need for the synchronized blocks and methods. However, it does not make every action with the item atomic - for example, a volatile int myInt will not have an atomic myInt++. The increment uses a read and a write, as myInt = myInt + 1, and so even though it may look like a single operation, it won't be, which makes volatile a bit misleading. Further more, declaring an array volatile only makes the pointer volatile and not the things inside the array. A better approach is to simply synchronize access to these variables, so that it is guaranteed any code within the synchronize blocks will be called atomically.
53. Write Erlang code that starts a new thread. The thread should initially execute function "func" from module "mod" with three arguments a, b, and c. There is no need to retrieve the function's result.Repeat the exercise above except that the new thread should be "monitored" by the initial thread.
    ```
    MyThread = spawn(mod, func, [a, b, c]).

    {MyThread, Ref } = spawn_monitor(mod, func, [a, b, c]).
    ```
54. What facilities does Erlang provide to control concurrency? Explain how these facilities could be used to implement a semaphore.
    - In Erlang, there is no shared state. Instead, you have multiple processes that communicate using message passing. To implement a semaphore, first spawn a thread to act as the semaphore. This thread will run a function that takes an integer (the

semaphore starting number). To do semaphore operations, this thread will continually receive messages. The spawned thread will keep track of a list of waiting threads and the passed in number. After sending a message to the semaphore, a thread will go into an infinite loop that waits for messages until it receives a message from the semaphore saying its ok to continue. The semaphore will handle sending messages to those waiting processes when it gets a V() message.

55. When an Erlang thread that is being monitored crashes, how does the monitoring thread learn about the crash?
    - When a thread crashes, a 5-tuple is sent to the monitoring thread that looks like this: {'DOWN', Ref, process, Pid, Reason}. Ref is the value returned by spawn_monitor when the thread was created. Pid is the thread's id, and Reason is the crash message.
56. Erlang does not permit threads to share memory. Explain how this feature aids the construction of reliable concurrent code.
    - Since Erlang uses an Actors model, as opposed to a threads and locks model, it doesn't share data between different "actors". There are no mutual exclusion problems or race conditions because all values are either local or passed via message to other actors. If one thread fails unexpectedly in a threads and locks model, it leaves the address space unpredictable state, and other threads have to cope. If an actor fails, other threads just no longer receive messages which isnt a problem. Furthermore, it is easily distributable to large systems, because any messages sent just become network messages, whereas the typical threads model requires some way of having a shared address space on different machines.
57. If many Erlang threads each had to read and make updates to a single very large data structure, how would you design the code to do so?
    - There would most likely be a single thread which took input from many threads, and updated the data structure recursively - it would recursively receive messages from the many erlang threads. To manage the reading and writing, it could first receive a request to read, then write back the response, then read messages specifically from that one thread to get the update. When the update was finished, it could get an update from a new thread.
58. Explain what "model checking" is.
    - Model checking is a formal method for verifying the correctness of programs, which uses software to analyze the states and possible execution path. Model checking involves creating a model of a program, and attempting to prove properties about that model. If it is clear how pieces of the model relate to the program, then the properties can give some proof of the program's correctness.
59. What advantage does model checking have compared to testing?
    - Testing is flawed. It involves just running lots of tests in the hopes of covering every case or at least the main cases where things could go wrong. The more complex a program is, the more difficult it can be to encompass all the tests. Furthermore in concurrent programs, it is nearly impossible to check all the different interleavings of the program. Model checking involves proving a "model" of the program rather than the program itself. In this way, it proves a slightly simpler version of the program is correct - by running through every initial state and execution of the program. So, as long as the model correlates in an obvious way to the program, it is a good indicator of the program's correctness, and has the benefit of being able to test different interleavings in concurrent programs.
60. What disadvantage does model checking have compared to testing?

- Model checking is harder than testing, and slower. Testing is good for finding bugs and other obvious sources of inaccuracy. Model checking will find all of the incorrect assertions, but testing can quickly lead the programmer to incorrect portions of the code. While the program may not be proven to be completely correct, it can be shown that it works in at least X cases correctly, and it is possible to reason about the program based on those X cases.

61. What disadvantage does model checking have compared to program proof?
    - Program proof involves creating specifications for each statement in the code, and then showing that (state1 && statement A) -> state2, (state2 && statement B) -> state 3, etc. If this is done for many small theorems such that it composes the entirety of the program, the program is shown to be correct.
    - Program proof, by definition, completely verifies correctness of the given program, and not of a model of a program, and not of a subset of its executions. While program proof is very difficult in most cases, a program proof assures complete correctness.

Peterson's solution

```
int turn_to_wait;
int interested[2];

enter(proc) {
   int other = 1 - proc;     /* assumes 0,1 */
   interested[proc] = TRUE;
   turn_to_wait = proc;
   while ((turn_to_wait == proc) &&
          (interested[other] == TRUE))
      ;   /* wait by looping */
}


leave(proc) {
   interested[proc] = FALSE;
}
```

Strict Alternation

Thread 0:

```
int turn_to_run;     /* shared variable */

while (TRUE) {
   while (turn_to_run != 0)
      ;     /* wait */
   <critical section>
   turn_to_run = 1;
   <other, non-critical code>
}
```

Thread 1:

```
while (TRUE) {
   while (turn_to_run != 1)
      ;     /* wait */
   <critical section>
   turn_to_run = 0;
   <other, non-critical code>
}
```