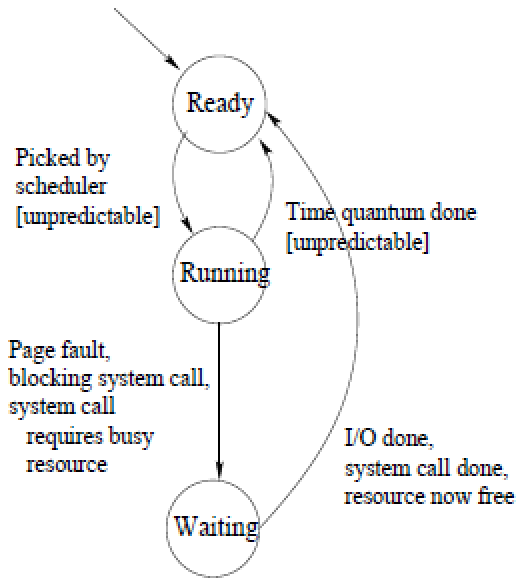
CS511 Fall 2014 Possible Questions

* Page numbers next to questions are based off of combined PDF with all slides printed 4x4 on each page.

1. Explain the difference between a process and a thread. (1)
    - Process - program in execution with all the resources it needs for its execution (PID, virtual address space, instructions, …)
    - Thread - execution stack within a process (registers, stack, PSW)
    - Processes have their own virtual address space while threads share the address space of their process.
    - Process is heavyweight with its own address space. Threads are tasks within a process and share memory with each other.
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? (1, 4)
    - Processes are more appropriate when you want a robust application because they are independent of each other and one process crashing will not crash the other one like with threads in a process.
    - Threads are more appropriate when there are many schedulable units or much shared data because they are more lightweight with less overhead making them faster to create, destroy, or switch and shared memory is one of their features.
    - Thread are easier to use/create. They have less overhead memory wise and can share memory with each other due to shared address space. Processes independent and don't require careful multithreading due to separate address space. Use threads is need to share memory / creating a lot of threads.
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. (1)
    - 
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. (1)

```
pid_t child;
if ((child = fork()) < 0) {
        A();
} else if (child == 0) {
        C();
} else {
        B();
}
```

5. Explain how to wait for input from more than one file descriptor simultaneously. (2-3)
   You use the select function which takes sets of file descriptors as arguments: read set, write set, exception set. It also takes a time out argument. The first argument is the maximum fd to check plus 1.

```
int rc, max_fd;
struct timeval timeout;
struct fd_set call_set;
timeout.tv_sec = 60;
timeout.tv_usec = 0;
FD_ZERO(&call_set);
FD_SET(fd, &call_set);
max_fd = fd; /* only 1 fd in this example */
num = select(max_fd + 1, &call_set, NULL, NULL, &timeout);
/* this code handles any number of "ready" fds */
for (i=0; i <= max_fd; i++) {
        if (FD_ISSET(i, &call_set)) {
                …
        }
}
if (--num <= 0)
break;
}
```

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? (3)
   - Block - keep the signal pending until it is unblocked (sigprocmask)
   - Ignore - receive the signal and immediately drop it (sigaction)
   - Handle - receive the signal and handle it by calling a function (sigaction)
   - Blocking a signal with sigprocmask means that you will keep waiting until the signal can be handled. Ignore is to receive signal and then do nothing with sigaction. Handling a thread means to receive a signal and then handle it via a function call. This is also done through sigaction.
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. (4)
   - Such “signal safe” function should not access (read or write) static data, like malloc.
   - Make sure that functions handling signals are reentrant for signals. Otherwise, make sure that the signal cannot interrupt a call to another function. Printf, Malloc, etc are examples of functions that should not be called by signal handlers.

8. What does it mean for a section of code to be “reentrant”? (4)
   - Reentrant means that it can be safely interrupted in the middle of what it’s doing by another call to itself.
9. The non-reentrancy of some functions can be spotted based only on a description of the function’s interface. Explain how to do so. (4)
   - Reentrant functions must not return pointers to static data, write to shared errno, use globals in non-multithread-safe fashion (aka use locks for globals). Reentrant functions can not call non-reentrant functions, hold non-constant data for static vars/globals, or modify its own code.
10. Explain how hardware interrupts are used by the operating system to implement preemptive scheduling. (Notes 8/25/14)
    - Hardware interrupts used by OS in form of clock. Clock sends pulse at set intervals, causing interrupts. This lets OS know it should switch to another process.
11. 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.

```
struct args_t {
        int i;
        char *s;
        float f;
};

void *start(void *args) {
        int *retval;
        struct args_t *a = (struct args_t *) args;
        retval = (int *) malloc(sizeof(int));
        *retval = a->i;
        return (void *)retval;
}

int main(int argc, char *argv[]) {
        pthread_t tid;
        struct args_t args;
        void *retvalLocation;
        args.i = 3;
        args.s = argv[0];
        args.f = 2.0;
        (void) pthread_create(&tid, NULL, start, (void *)args);
        (void) pthread_join(tid, &retvalLocation);
        printf("thread returned %i\n", *(int *)retvalLocation);

}
```

12. 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.
    - Pass the thread a struct pointer where that struct holds all the data you need to access.
13. Explain how a "zombie" UNIX process is created. Explain why Pthreads does not include the zombie concept.
    - Zombie processes are created when a child process terminates but the parent has not yet called wait. As such, the process's memory just sits around with an entry in the process table. Pthreads does not have zombie processes since at termination the memory just waits to be cleared. There is no table of processes.
14. Explain how to make a function thread-safe. (6)
    - To make a function thread-safe, only modify local memory. If you need to touch globals, use a mutex. Don't return pointers to static data or write to shared errno.
15. Explain how to make a function cancel-safe. (6)
    - Set thread cancel state to be disabled. If enables, have cancel type be deffered. Disabled does not allow for canceling. Deferred means that it can only be cancelled at a cancellation point. If the type is asynchronous and not deferred, it is killed immediately when canceled, and there is no way to ensure that the thread was not in the middle of something important.
16. 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? (7)
    - A new 1-thread process is created. That thread is a replica of the specific thread in the parent process that called fork. The address space of the child duplicates that of the parent, including all states created by other threads in parent. It does not matter how many threads you have in the parent process, forking always behaves the same way.
17. What happens to each of a process's threads when a system call in the "exec" family is invoked? (7)
    - All threads are terminated and new thread started to run main in new file.
18. 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. (7)
    - If a signal generated by hardware or software exception (e.g., SIGKILL or SIGSEGV), then "effective target" of signal is thread that caused exception, so the signal is delivered to offending thread.
19. If a signal is generated by an external process, to which thread in a

multi-threaded process is it delivered? (7)
   - Signal is delivered to arbitrary thread that does not have signal blocked.

20. Give an example of a race condition. Explain how the race condition may lead to incorrect results. (7)
   - foo = foo + 1
   - Two threads try to increment a global variable. First thread gets current value, stores it in a register, and increments that register. Then second thread runs, gets unchanged value, stores it in a register, and increments that register. First thread stores value. Second thread stores value. Global only shows 1 added instead of 2.
21. Explain how preemptive scheduling may cause race conditions to occur. (7)
   - OS schedules threads at unpredictable times
   - With pre-emptive scheduling, you don't know when your thread will be switched out for another one. As such, without taking measures to protect critical sections you may unknowingly cause race conditions.
22. 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? (7, 10, 11, 19)
   - In higher level languages, we can assume that the writing of a value of the machine word size is atomic (not addition but just writing an individual value will not cause bits to be interleaved). For java, this applies to types int, short, and char. In C, that depends on the machine architecture but is typically the same. Additionally, there are things like Java's AtomicInteger class that allow for each compare and swap operations (increment and get for example). 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 Set (TAS).
   - LL, SC
23. Explain the major advantage and major disadvantage of non-preemptive scheduling. (7)
   - Major advantage is there will never be race conditions, threads switch among themselves only at safe times. Major disadvantage is perceived user slowdown and less concurrency. If all programs are waiting on each other to yield, then everything will experience massive slowdowns.
24. Explain what mutual exclusion is. What problem does it solve? (8, 9)
   - 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.
25. List and explain the necessary conditions for a satisfactory solution to mutual exclusion. (8)
   - No two threads may be simultaneously inside their critical section (partial correctness)

- No thread should wait arbitrarily long to enter its critical section (liveness - freedom from starvation)
- No thread stopped outside its critical section should block other threads
- No assumptions about relative speeds of threads or number of CPUs
- No knowledge (at coding time) about number/identity of other threads

26. “Peterson’s Solution” and “Strict Alternation” provide mutual exclusion assuming only atomic single-word read and write operations. Explain how these solutions are inadequate. (9)
    - Strict Alternation violates 2 conditions of mutual exclusion because non-CS thread can block others (e.g. 0 can’t enter its CS until 1 has had its turn; what if 0 needs to enter its CS more often than 1?), and it must have a fixed number of threads, known at program creation time. Also, it specifically works only for 2 threads.
    - In Peterson’s Solution: unlike strict alternation, turn_to_wait tells whose turn it is to WAIT; if both execute enter() simultaneously, whichever sets turn_to_wait LAST will wait. It is an improvement over Strict Alternation, because it does not violate the first condition that S.A. does, but still only works for exactly 2 threads.
27. Explain why the disabling of interrupts is no longer a viable synchronization technique, even inside operating systems. (9)
    - Disabling of interrupts is no longer a viable technique because there is no guarantee that the interrupts will ever be re-enabled. This could cause the computer to crash. It is wasteful because it halts ALL threads, even if only SOME want mutex. Doesn’t work on multiprocessor.
28. Explain each of these terms: liveness, starvation, livelock. (8)
    - Liveness - algorithm always does something
    - Starvation - one particular thread can never make progress
    - Livelock - (as opposed to deadlock) when a set of threads are starved
29. What does it mean to “busy-wait” or “spin” waiting for a lock? What is disadvantageous about this technique? (10)
    - Looping over and over, attempting to acquire the lock each time, rather than waiting to do so until the lock is unlocked. This technique is wasteful of CPU time, and a better solution might be to “suspend” the waiter by placing it on a queue and “resume” it once the lock is available.
30. 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. (11)

    To get lock:
```
r2 = 1;
while (1) {
        LL r1, lock
        if (r1 == 0)
                if (SC r2, lock)
                break;
}
```

- SC returns 1 iff lock was STILL marked by this CPU; i.e., no other CPU executed SC before this one did. To drop lock: STORE lock, 0. This is efficient for multi-threaded programs since it does not block the processor memory bus like compare and swap (CAS) does.

31. Explain what a semaphore is and how it is used to provide mutual exclusion. (12, 13)
    - A semaphore is ultimately a variable that holds an integer. In order to provide mutual exclusion, two atomic operations are provided P() and V(). P tries to subtract 1 from the semaphore. If the semaphore is already 0, then the thread is put to sleep until the variable is positive. V() adds one to the variables atomically and wakes up a sleeping thread if any.
32. Explain the primary disadvantage of the semaphore concept. (13)
    - The programmer must keep track of all calls to wait and to signal the semaphore. If this is not done in the correct order, it will cause deadlock.
33. 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 worry about interleaving.
34. Explain what is the “monitor invariant”? (14)
    - A monitor invariant is a behavioral constraint in which no more than one monitor procedure will “run” at any time. It is an assertion which is true whenever no thread is executing in the monitor. It is permitted to be false while a thread is executing but must be true when the thread leaves the monitor by method return or wait().
35. Name the three major operations on a monitor and explain their operation. (14)
    - Wait / await - atomically releases lock then waits on condition. When thread re-awakens (due to some broadcast/signal), thread requests lock. Wait returns only after lock has been re-acquired (this allows thread to enter monitor)
    - Signal / signal - an optimization of broadcast; enables ONE waiting thread to run (thread is picked according to some unspecified policy). When operation returns, lock is still held
    - Broadcast / signalAll - enables ALL waiting threads to run. When operation returns, lock is still held
36. 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? (16)
    - Other threads may run between signal=er and signal-ee
    - Between when thread A’s signal() awakens thread B and when thread B waits for the lock, some thread C may have run and undone the application-specific condition that A established for B

    - Therefore, condition must be re-checked when signal-ee awakens
    - You should test in the condition of a while loop rather than in the condition of an if statement because it is possible for a thread to be woken while the condition is not met. Signaling a thread does not mean the thread will immediately execute. Given scheduling constraints, it's possible to signal a thread and have the condition change again before the thread actually executes.

37. How is a "recursive" Pthread mutex lock different from a standard Pthread mutex lock? (12)
    - PTHREAD_MUTEX_NORMAL - "Attempting to re-lock the mutex causes deadlock. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, undefined behavior results."
    - PTHREAD_MUTEX_RECURSIVE - "the mutex shall maintain ... a lock count ... Every time a thread relocks this mutex, the lock count shall be incremented by one. Each time the thread unlocks the mutex, the lock count shall be decremented by one. When the lock count reaches zero, the mutex shall become available for other threads to acquire. If a thread attempts to unlock a mutex that it has not locked or a mutex which is unlocked, an error shall be returned."
    - The re-locking deadlock that can occur in the normal cannot occur in the recursive
    - A thread attempting to re-lock the mutex without first unlocking will deadlock in normal but will NOT deadlock in recursive
    - 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.
38. What is dangerous about cancelling a Pthread? How can these dangers be avoided or mitigated? (6)
    - There is no way to ensure that a canceled thread won't be in the middle of a function that is not safe to interrupt. To prevent this, set thread cancel state to deferred before any unsafe operations.
39. What is "fairness" (wrt accessing a resource, such as a lock)? (20)
    - "fair" access means FCFS - goal is to avoid starvation
    - 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
40. What is the purpose of having multi-mode locks? Give an example of a moded lock. (20)
    - Multi-mode locks allow you to have multiple readers at once. You would

still need to prevent concurrent write, but concurrent reads is fine.

41. Consider the Runnable and Callable Java interfaces. Explain what they have in common and what are their differences. (22)
    - Callable is the same as Runnable in terms of executing tasks in another thread, however the Callable interface allows for returning a result and throwing exceptions.
42. Explain how the Java Future interface is useful. (22)
    - The Future interface is a representation of a task that will eventually be completed. If allows you to run tasks in the background at your leisure while providing an easy way to retrieve return values. Cancellation is also possible in this interface to stop asynchronous operations before they execute.
43. 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.

```
public class CallableFoo implements Callable<int> {
        private String a;
        private float b;

        public CallableFoo(String a, float b) {
                this.a = a;
                this.b = b;
        }

        public int call() {
                return foo(a, b);
        }
}

public static void main(String[] args) throws Exception {
        String a = "Hello";
        float b = 2.0;
        CallableFoo<int> c = new CallableFoo<int>(a, b);
        FutureTask<int> ft = new FutureTask<int>(c);
        Thread t = new Thread(ft);
        t.start();
        t.join();
        int res = ft.get();
}
```

44. In Java what is an "object lock"? How is it useful for controlling concurrency? (18)
    - Every java object has a hidden lock built into it. This lock is useful for concurrency since it is obtained and dropped at the entrance and exit of synchronized methods regardless of how you exit the method. This way the programmer does not have to worry about accidentally forgetting to drop a lock at the end of a method. If all methods that touch global state in

a class are declared synchronized, this automatically makes the class thread-safe. In regards to monitors, this is exactly like locking a mutex. You do not have the ability to wait on certain conditions, however. Each thread will block just like as if it was waiting for a spin-lock. Additionally, this object lock is held the entire time, not just when touching mutable shared state.

45. Contrast the functionality provided by a monitor versus the functionality provided by Java's synchronized keyword. (18)
    - synchronized provides only the "monitor lock" part of the monitor concept as it does not require "wait" and "signal".
46. Evaluate the functionality provided by Java's synchronized keyword as a solution for mutual exclusion. (18)
    - If all methods are declared synchronized, they the class is automatically thread safe (via mutual exclusion by obtaining and later dropping the object lock. Static methods can also be declared synchronized. Only one synchronized method may run at a time.
47. 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. (19)

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

48. 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. (17)

```
ReentrantLock aLock = new ReentrantLock();

aLock.lock();
try {
        // access shared variables
        // that are protected by this lock
} finally {
        aLock.unlock();
}
```

49. 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.
C example the professor posted (In-class exercise 10/10/14):

```
#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, "space-available condition variable init failed: %s\n", strerror(rc)),
exit(1);
  if ((rc = pthread_cond_init(&stringAvailable, NULL)) != 0)
    fprintf(stderr, "string-available 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;
}
```

Java example from official javadocs pages:

```
class BoundedBuffer {
   final Lock lock = new ReentrantLock();
   final Condition notFull  = lock.newCondition();
   final Condition notEmpty = lock.newCondition();

   final Object[] items = new Object[100];
   int putptr, takeptr, count;

   public void put(Object x) throws InterruptedException {
     lock.lock();
     try {
          while (count == items.length)
          notFull.await();
          items[putptr] = x;
          if (++putptr == items.length) putptr = 0;
          ++count;
          notEmpty.signal();
     } finally {
          lock.unlock();
     }
   }

   public Object take() throws InterruptedException {
     lock.lock();
     try {
          while (count == 0)
          notEmpty.await();
          Object x = items[takeptr];
          if (++takeptr == items.length) takeptr = 0;
          --count;
          notFull.signal();
          return x;
     } finally {
          lock.unlock();
     }
   }
 }
```

50. 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. (20)
    - If you have a read lock, you must drop it to acquire a write lock. Upgrade not allowed. So readLock.unlock(); writeLock.lock(); writeLock.unlock();
51. 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? (22)
    - Fixed thread pools creates a set number of threads at creation time. If a thread is free, it will execute a waiting task. Otherwise, that task will just sit in a queue. Guarantees that at most N threads will be running at a time. Cached thread pools, in comparison, create new threads to handle tasks if none are available. After a thread executes a task, it will stay around to execute future tasks for only a set amount of time (60 seconds). Cached thread pools will create new threads to keep up with additional load but will start a large number of threads if many simultaneous tasks are submitted all at the same time.
52. Java provides a variety of synchronized Collection classes. What is the major advantage of using such classes? What is the major disadvantage? (23, 24)
    - Major advantage is that they handle concurrency and thread safety for you. They are already optimized to be used from multiple threads at once. The main downside is that there is extra overhead and latency with using thread safe classes. Usually worth it if using multiple threads, but in single threaded situations it is a different story.
53. Explain how to use a "synchronization wrapper" to convert an unsynchronized Collection class into a synchronized class. (24)
    - List<E> syncArrayList = Collections.synchronizedList (new ArrayList<E>());
    - Map<K, V> syncHashmap = Collections.synchronizedMap(new HashMap<K, V>());
54. 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? (19)
    - Ints, shorts, and chars can be declared volatile to make them atomic for

reading/writing. They will then always be written and read from memory as opposed to from caches. Does not apply to longs or doubles. This allows for a form of thread-safe behavior without the usage of locks or the synchronized keyword. It is bad because it is misunderstood and will not provide synchronized access. Declaring an array volatile will not make individual element inside the array volatile. Additionally, volatile int a = 0; a++; the a++ part will still not be thread safe. Same issues as if you did this in C++.

55. 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. (27)
    - Pid = spawn(mod, func, [a, b, c]).
56. Repeat the exercise above except that the new thread should be "monitored" by the initial thread. (27)
    - { Pid, _ } = spawn_monitor(mod, func, [a, b, c]).
57. What facilities does Erlang provide to control concurrency? Explain how these facilities could be used to implement a semaphore. (25)
    - 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.
58. When an Erlang thread that is being monitored crashes, how does the monitoring thread learn about the crash? (27)
    - When monitorED process dies, moniterING process receives this tuple as a message: { 'DOWN', Ref, process, Pid, Reason }
    - 'DOWN' and process are atoms; i.e., literal values
    - Ref is variable; has same value as returned by spawn_monitor/3
    - Pid and Reason are variables
59. Erlang does not permit threads to share memory. Explain how this feature aids the construction of reliable concurrent code. (27)
    - Since the Erlang threadsdon't share memory, there is no need to maintain mutual exclusion for critical sections simply because there are no critical sections to begin with. With this feature, the threads are independent of each other and there is no concurrent access on shared variables because they don't exist.
60. 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?
    - I would have 1 thread that managed the reads and writes to the data

structure by listening and responding to messages from other worker threads trying to read or make updates to the data structure.

61. Explain what “model checking” is. (28, 29)
    - Model checking is a technique for verifying that a given program is correct. It relies on creating a model of the program to prove certain properties of the program.
62. What advantage does model checking have compared to testing? (28, 29)
    - There are lots of cases to test and code, but model checking avoids this by automatically checking all possible execution paths through a program.
    - It allows for the testing of all possible interleavings of a concurrent program.
63. What disadvantage does model checking have compared to testing? What disadvantage does model checking have compared to program proof?
    - Model checking relies on a simplified modeling language which may not be able to model all kinds of programs to prove them correct.
    - A model does not necessarily prove that the program will be correct if it is mistranslated from the model into code. A program proof would prove that a real program written in a real language is correct, instead of in a limited modeling language.