CS 511 Final Review

Q:Explain the difference between a process and a thread.

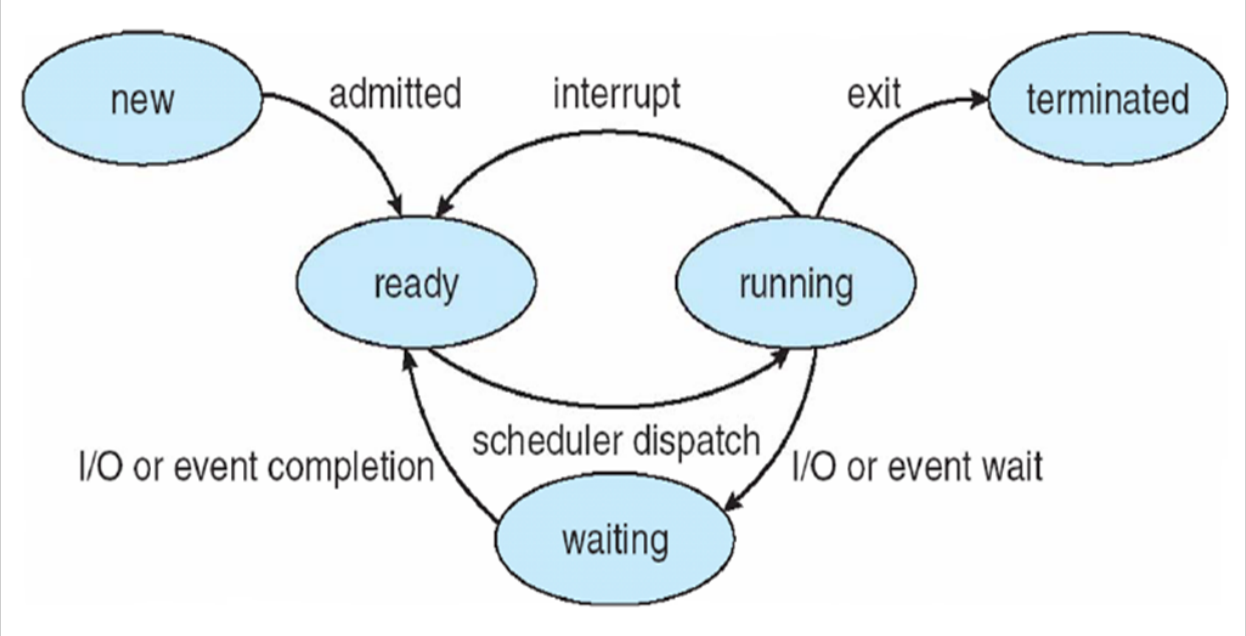
A: **Process - executing instance of an application (can contain multiple threads)**
used for "heavy" tasks
Thread - path of execution *within* a process (threads within same process share same address space)
used for "light" tasks
Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

Q: 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: **1. Threads are easier to create than processes since they don't require a separate address space.**
2. Multithreading requires careful programming since threads share data structures that should only be modified by one thread at a time. Unlike threads, processes don't share the same address space.
3. Threads are considered lightweight because they use far less resources than processes.
4. Processes are independent of each other. Threads, since they share the same address space are interdependent, so caution must be taken so that different threads don't step on each other. (this is really another way of stating 2)
5. A process can consist of multiple threads.

Threads are more appropriate to use than processes when needing to access a particular data set. For example, if you open an application like a text editor, you are starting a process. When you are editing your document, different threads are running in order to execute your commands. It is appropriate to use threads in a case like this because they are all accessing the same document. Processes would not do this as easily.

Q: Draw the state transition diagram for a process. Label the states and indicate examples of events cause transitions among states. Show all transitions.



Q: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.

A:

```
if((pid = fork()) == 0) //in the child
        C();
else if(pid > 0) //in the parent
        B();
else if(pid < 0)
        A();
```

Q:Explain how to wait for input from more than one file descriptor simultaneously.

A: Your main thread would spawn one thread per file descriptor.

Code from Duchamp: this We use the select system call. you give it a range of possible descriptors to listen on and it checks to see if any are ready to be read from. Duchamp's example uses sockets, which are also file descriptors so it works fine.

```
fd_set socketSet;
FD_ZERO(&socketSet);
FD_SET(listenSocket, &socketSet);
for (i=0; i<num_clients; i++)
  FD_SET(client[i].socket, &socketSet);
rc = select(4+MAX_CLIENTS, &socketSet, NULL, NULL, NULL);
```

```
if (rc < 0) {
  perror("select failed");
  exit(1);
} else if (rc == 0) {
  continue;
}
```

Q: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?

A: **<u>Block</u> - keep pending until signal unblocked (sigprocmask(2))**
E.g. if a 2nd thread tries to execute its critical section while lock is held, 2nd thread’s attempt to get lock will be blocked, until the 1st thread drops it
<u>Ignore</u> - delivered and immediately dropped (sigaction(2))
only one thread can hold lock at a time
<u>Handle</u> - delivered and handled (sigaction(2))

Q: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.
If you call a function in the handler, make sure it is reentrant with respect to signals, or else make sure that the signal cannot interrupt a call to a related function. ex: malloc or printf.

Q: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.

A: **Reentrant (same as Pthreads):**
***can be interrupted in the middle of its execution and then safely called again (“re-entered”) before its previous invocations complete execution**

A reentrant function must be:
-function does not return pointer to static
-function does not write to shared errno
-function does not use globals OR function gets lock before accessing globals

A function is NOT reentrant if:
-calls another non-reentrant function
-holds any static (or global) non-constant data
-modifies its own code (itself)

Q:Explain how hardware interrupts are used by the operating system to implement preemptive scheduling.

A:

In order to achieve multi-process/thread scheduling in an operating system, the hardware contains a clock. This clock will pulse at set time intervals, which in turn cause hardware interrupts. These interrupts are caught by the operating system which in turn knows that it should probably switch to running a different process/thread. It saves all registers, halts all current operations, and then loads another task's registers in its place. Processing then continues.

Q:Explain how to pass an unbounded amount of data to a Pthread even though its start function accepts only one "void *" argument.

A:

Make the void* argument point to a struct that contains the data you want the function to have access to.

Q:Explain how a "zombie" UNIX process is created. Explain why Pthreads does not include the zombie concept.

A:

Zombie processes are created when a process completes its execution but its parent has not called wait on it, so it still has an entry in the process table.
Slides: "If child terminates before parent waits: it becomes a zombie - OS saves its exit code so that parent can later wait for it"
Pthreads do not include the concept because they are threads waiting on a pthread_join() call to have their memory cleared, not an entire process waiting on a wait() call to have all its memory cleared.

Q:Explain how to make a function thread-safe.

A:

The function must not modify any non-local memory, or if it does, that memory must be protected. A mutex to lock it before each access and unlock it after would make the function thread-safe. It must also not return pointer to static, or write to shared errno.

Q:Explain how to make a function cancel-safe.

A:

Either have the thread's cancel "state" be disabled, or if enabled, have its cancel "type" be deferred. Disabled means it cannot be canceled. Deferred means that the thread may be canceled only at certain "cancellation points" where implementation checks "should I kill this thread?" 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.

Q:What is the result if an N-thread process forks? That is, how many threads does the child process have and what code are they executing with the child process begins?

A:

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.

Q:What happens to each of a process's threads when a system call in the "exec" family is invoked?

A:

When any exec call happens, all existing threads are terminated, and a new thread is created to run main of new executable file.

Q: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.

A:

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.

Q:If a signal is generated by an external process, to which thread in a multi-threaded process is it delivered?

A:

If a signal is generated by an external process, the "effective target" is the process, so the signal is delivered to an arbitrary thread that does not have signal blocked. The OS likely chooses the thread based on what's simplest to implement.

Q:Give an example of a race condition. Explain how the race condition may lead to incorrect results.

A:

A student swipes their duckbills card at two vending machines at the same time, then selects something in each machine. Both machines will try to subtract the dollar amount of the item selected from the student's total, however they will not wait on each other to go first (their critical sections are not protected). They will both subtract their items values from the student's original total and only the latter will remain permanent. As in, Student has $10 in account, buys a soda for $1.50 and a snack for $1.00 at the same time. Their total should end up as 10-1-1.5 = $7.50. However, one machine will do 10-1 = $9 and the other will do 10-1.5 = $8.50, and whichever happened last will be what gets saved.

[ex) credit card number without any security OR 84/85]

Q:Explain how preemptive scheduling may cause race conditions to occur.

A:

In non-preemptive scheduling, threads switch among themselves only at safe times, and the OS does not preempt thread execution. In preemptive scheduling, the OS schedules threads at unpredictable times, and if the threads do not protect their critical sections with some mutual exclusion mechanism, race conditions may occur.

Q: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?

A:

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 Swap (TAS).

Q:Explain the major advantage and major disadvantage of non-preemptive scheduling.

A:

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.

Q:Explain what mutual exclusion is. What problem does it solve?

A:

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.

Q:Explain the necessary conditions for a satisfactory solution to mutual exclusion.

A:

1) No two threads may be simultaneously inside their critical section (partial correctness)

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

Q:"Peterson's Solution" and "Strict Alternation" provide mutual exclusion assuming only atomic single-word read and write operations. Explain how these solutions are inadequate. Explain why the disabling of interrupts is no longer a viable synchronization technique, even inside operating systems.

A:

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. 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.

Q:Explain each of these terms: liveness, starvation, livelock.

A:

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

Q:What does it mean to "busy-wait" or "spin" waiting for a lock? What is disadvantageous about this technique?

A:

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.

Q:Explain what the "load linked" and "store conditional" MIPS instructions do.

A :

Load Linked atomically does
1. load memory location into address register

2. mark location with CPU's ID

Store Conditional atomically does

1. if memory location is marked by the CPU, store new value and remove ALL marks; else do nothing

2. return the indication of which case occurred.

Q:Explain how to use load-linked and store-conditional for locking, and explain why this solution is suited to a shared memory multiprocessor.

A:

```
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.

Q:Explain what a semaphore is and how it is used to provide mutual exclusion.

A:

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.

Q:Explain the primary disadvantage of the semaphore concept.

A:

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.

Q:Explain why it is not a good idea for more than one thread to share a single file descriptor.

A:

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.

Q:What is the “monitor invariant”?

A:

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().

Q:Name the three major operations on a monitor and explain their operation.

A: **Wait (await in Java) - 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 in Java) - 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 in Java) - enables ALL waiting threads to run. When operation returns, lock is *still held*

Q:Why should an application-specific condition—failure of which to be satisfied would lead a monitor function to call wait—be tested in the condition of while loop rather than in the condition of an if statement?

A:

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, its possible to signal a thread and have the condition change again before the thread actually executes.

Q:How is a “recursive” Pthread mutex lock different from a standard Pthread mutex lock?

A:

- **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.**

Q:What is dangerous about cancelling a Pthread? How can these dangers be avoided or mitigated?

A:

- **there is no way to ensure that a canceled thread won't be half done with some operation that should not be left partially done (such as transfer of funds)**
- **to prevent, set thread cancel state to "deferred" before any "dangerous" operations**

Q:What is "fairness" (w.r.t. accessing a resource, such as a lock)?

A:

- **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.**

Q:What is the purpose of having multi-mode locks?

A:

- **read and write: allow single writer or multiple readers**
- **only one can write at a time, but multiple can be reading**

Q:Consider the Runnable and Callable Java interfaces. Explain what they have in common and what are their differences.

A:

The Callable interface is similar to Runnable, in that both are designed for classes whose instances are potentially executed by another thread. A Runnable interface however, does not return a result and cannot throw a checked exception.

Q:Explain how the Java Future interface is useful.

A:

- **Represent a tasks eventual result.**

- **Useful for instantiating Future object.**
- **Holds the result of an asynchronous computation.**
- **“You can start a computation, give someone the Future object, and forget about it. The owner of the Future object can obtain the result when it is ready” (text 774 / pdf 506)**
- **Program can move on with code and continue with computations while waiting for the return value of the future value**
- **A Future represents the result of an asynchronous computation. Methods are provided to check if the computation is complete, to wait for its completion, and to retrieve the result of the computation. The result can only be retrieved using method get when the computation has completed, blocking if necessary until it is ready. Cancellation is performed by the cancel method. Additional methods are provided to determine if the task completed normally or was cancelled. Once a computation has completed, the computation cannot be cancelled. If you would like to use a Future for the sake of cancellability but not provide a usable result, you can declare types of the form Future<?> and return null as a result of the underlying task.-**

Q: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, pass arguments to it, execute foo, and retrieve foo’s result.

A:

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

Q:In Java what is an “object lock”? How is it useful for controlling concurrency? Contrast the functionality provided by a monitor versus the functionality provided by Java’s synchronized keyword.

A:

Object Lock:

- **each object has one hidden object lock that is**
  - **obtained upon entry to synchronized method**
  - **dropped upon exit from synchronized method**
    - **dropped no matter how the function exits:**
      - **By return.**
      - **By throwing exception..**
      - **By failing to catch thrown exception.**
    - **How is it useful for controlling concurrency?**
      - **If all methods in an object that touch mutable state are marked synchronized, than this makes it so you do not need to do explicit locking. All synchronized methods are thread safe.**
    - **Contrast the functionality vs a monitor:**
      - **This functionality is just like explicitly locking a mutex (this is exactly what happens on this happens without explicit code on the programmer's part). However, this does not allow for waiting on a condition. If you are trying to retrieve something from a queue or wait for something if it is empty, then synchronized will not work. Object lock will be held entire time. Additionally, synchronized holds lock for entire method call, not just when touching shared state.**

Q:Evaluate the functionality provided by Java’s synchronized keyword as a solution for mutual exclusion.

A:

- **Static methods can be synchronized.**
- **Entry into synchronized static method gets class lock.**
- **At most, one of all of a class’s methods declared synchronized may be executing at a time.**
- **→ This ensures that no two processes or threads will be in the critical section at the same time (mutual exclusion).**

Q:Explain the usefulness of a synchronized block (block only, not an entire method).

A: **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.

Q: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.

A:

```
// Any arbitrary method.
synchronized(this)
{
    // BODY
}
```

Q: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.

A:

```
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();
        }
    }
}
```

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

A:

```
#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;
}
```

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.

Posted on Moodle - Week 10 Oct 28 - Nov 10 in-class exercise example solution . Java Implementation

Q:Suppose that a thread already contains a lock in read mode. Explain the steps involved to correctly “upgrade” the lock to write mode.

A: **According to slides:**
Read/Write Lock
Semantics, I
Reentrant: lock can be multiply acquired by same thread
Downgrade OK:
1. Acquire write lock
2. Acquire read lock
3. Drop write lock
***Upgrade NOT OK* — must first drop read lock, then try to acquire write lock**

Q: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?

A:

Fixed:

-create fixed number of threads
-when given a task to execute:
-if a thread is free: assign task to the thread
-if no thread is free: wait until a thread becomes free
Reuses a fixed number of threads operating off a shared unbounded queue, at most *n* threads will be actively processing tasks, if additional tasks are submitted when all threads are active, new tasks will wait in the queue until a thread is available

Cached:

-when given a task to execute:
-if a thread is free: assign task to the thread
-if no thread is free: create new thread & assign task to it
-cached threads discarded after 60 seconds of disuse
Creates threads as needed, but will reuse previously constructed threads when they are available, typically improves performance of programs that execute many short-lived asynchronous tasks, will create new thread if no existing thread is available

Q:Java provides a variety of synchronized Collection classes. What is the major advantage of using such classes? What is the major disadvantage?

A:

- **Synchronized Collection Classes: Java Collections Framework**
  - **set of type-parameterized classes and interfaces for storing groups of objects**

    - **14 Interfaces**
    - **many classes that implement them - each class is a data structure**
        - **set**
        - **list**
        - **hash table**
        - **queue**
    - **java.util - contains all collection classes**
        - **[not in .concurrent]**
        - **Vector**
        - **Hashtable**
    - **java.util.concurrent - contains some but not all thread safe collection classes**
        - **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**

Q:Explain how to use a “synchronization wrapper” to convert an unsynchronized collection class into a synchronized class.

A:

List<E> syncArrayList = Collections.synchronizedList (new ArrayList<E>())j
Map<K, v> syncHashmap = Collections.synchronizedMap(new HashMap<K, V>());

Q: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?

A:

ATOMIC instances can be declared volatile--volatile values will not be cached, meaning (does not apply to non-atomic variables such as long, double):

-current value always read from memory

-new value always written to memory

GOOD: Can make a class thread-safe without "synchronized" keyword
BAD: Is a hack, easily misused, if used incorrectly it will not provide synchronized access

ex 1. If you declare volatile int foo, then perform foo++, you are compiling to instructions that load AND store foo, not just store)
ex 2. if you declare volatile int[] foo and try to assign a value to an element of foo, foo alone is volatile (it is a reference to the array), but individual elements are NOT made volatile by the declaration

Q: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.

A:

NewThread = spawn(mod, func, [a, b, c]).

Q:What facilities does Erlang provide to control concurrency? Explain how these facilities could be used to implement a semaphore.

A:

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.