A certain Duke professor likes to sleep in the Perfect Chair™ in his office when there are no students around. The professor wakes up when students visit the office to ask questions.

Write two procedures to synchronize threads for the professor and students. When the professor thread has nothing to do, it calls `IdleProf()`. `IdleProf` sleeps if there are no students waiting, otherwise it signals one student to enter the office, and returns. A student with a question to ask calls `ArrivingStudent()`, which waits until the professor is idle; if no students are waiting, then the student wakes up the sleeping professor.

The idea is that the professor and exactly one student will return from their procedures “at the same time”. (They then discuss a topic of mutual interest, then the student goes back to studying and the professor calls `IdleProf` again.)

a) Implement `IdleProf` and `ArrivingStudent` using a mutex and condition variable(s). [30 points]

This problem is a variant of Rendezvous, like *Operators Are Standing By*, which is discussed in detail on the concurrency page. In this variant there are multiple customers/students, but only one operator/professor. The solution below also handles multiple professors. The fundamentals are: single mutex, each side has its own count and CV, with a wait on its CV and a signal of the other side’s CV. The counts are crucial, and there are many ways to mess it up. In particular, looping-before-leaping is tricky here unless you have each side consume a count from the other side: otherwise it’s not clear which of the looping waiters “wins” the right to rendezvous with a newly available partner. I accepted solutions that skipped the loop-before-leap, but I don’t like them as much because they presume that each signal wakes at most one waiter, which is not true in all CV implementations. If the fundamentals are right, then errors with the counts generally cost only 5 points.

```java
IdleProf() {
    mx.acquire();
    professors++;
    student.signal();
    while (students == 0)
        professor.wait();
    students--;
    mx.release();
}

ArrivingStudent() {
    mx.acquire();
    students++;
    professor.signal();
    while (professor == 0)
        student.wait();
    professors--;
    mx.release();
}
```
b) Implement *IdleProf* and *ArrivingStudent* using semaphores. [30 points]

**Rendezvous problems are simple and elegant with semaphores.** The solution is similar to producer/consumer: a semaphore for each side, with a V of the other side’s semaphore to signal that this side is ready, and then a P on the local semaphore to wait for someone on the other side. Note that at least one side must V before P: they deadlock if they both P first, unless you mess with the initial counts (which is confusing to the grader).

The most common error is to fail to wield the full power of semaphores effectively. Semaphores have their own internal counts, and if you use them properly, then you don’t need any counts or locking outside of the semaphores. Adding flags or counts DOES YOU NO GOOD because they are useless with semaphores. You can’t use semaphores like CVs! In particular, there is no way to coordinate the counts with blocking on your semaphore: any attempt to do so either results in a race or in deadlock. These errors are worth at least 10 points.

```c
Semaphore students (init 0);
Semaphore profs (init 0);

IdleProf() {
    profs.V();
    students.P();
}

ArrivingStudent() {
    students.V();
    profs.P();
}
```
Part 2. Blocks

For each of the following terms, write a short piece of text explaining what it is and why the thing exists (or why it is important). Feel free to add pictures. [30 points]

(a) parity block

A parity block stores redundant data for a RAID stripe, to enable recovery of lost data after a disk failure. A stripe is a sequence of blocks, one on each disk. In RAID-4/5, one of the blocks of each stripe is a parity block, and the others are data blocks. Each bit of the parity block contains even/odd parity over the corresponding bits of the data blocks. [I got some good pictures for this, which helped earn full credit.]

(b) block map

A block map represents a storage object (a variable-length sequence of bytes) as an array of “pointers” to fixed-sized blocks of storage. Examples include page tables and inodes. Page tables represent the storage of virtual memory segments in machine memory: the “pointers” are Page Frame Numbers. Inodes represent the storage of files on a storage volume (“disk”): the “pointers” are block numbers (blockIDs) on the disk. To find block N of the object, index the block map array at N and retrieve the “pointer”. [The obvious picture earns full credit.]

(c) indirect block

An indirect block is a block containing an array of block map entries. A standard inode contains a “pointer” to an indirect block to extend the block map for larger files. For very large files, the inode contains a pointer to a double indirect block, which is a block of pointers to indirect blocks. [The obvious picture earns full credit.]

(d) dirty block

A block or page is dirty if it has been modified in memory but the update has not yet been written (pushed) back to disk. The disk write “cleans” the page, so it is no longer dirty.

(e) victim block

A block or page is a victim when it is selected for eviction from memory (e.g., from the buffer cache). The block or page is cleaned (if it is dirty) and then discarded, and its memory is then used to store/cache a different page or block. The choice of victims is driven by a replacement policy, such as LRU.
Part 3. Elevator

These questions pertain to your elevator project code. If you don’t remember the details of what your group did then make something up that sounds good. Please confine your answers to the space provided. [30 points]

(a) How many EventBarriers does your code use? What purpose does each EventBarrier serve?

These kinds of questions are useful for me to be able to gauge the internal dynamics of a group. (If I compare your answers.) In general, I gave full credit for any reasonable answer, even if it would result in incorrect elevator behavior. I knocked off a few points for garble or strategic vagueness.

(b) The elevator is a real-world problem in which starvation is a concern. Is your elevator code free from starvation? State what it means for the elevator to be free of starvation, and outline how your elevator assures this property.

Here I looked for a clear definition of starvation and a convincing explanation of why your elevator was safe. Most answers received full credit.
Part 4. Back of the napkin

Suppose I have a service that runs on a server $S$ and receives requests from a network. Each request on $S$ reads a randomly chosen piece of data from a disk attached to $S$, and also does some computing. On average, a request arriving when $S$ is idle spends $T$ time units computing on the CPU and $T$ time units waiting for data from the disk.

Now suppose I redeploy the service on a new server $S'$. $S'$ is the same as $S$, except that the CPU is 20% faster and $S'$ has two disks, each containing a copy of the data.

(a) Estimate the throughput and mean response time of $S$ and $S'$ as the request arrival rate increases. Sketch the graphs, label the axes, and annotate key features of the graphs. What are the peak throughputs of $S$ and $S'$ at saturation? Note any other assumptions you need to make to answer the question. [30 points]
Part 4. Back of the napkin (continued)

Getting this right requires a little insight into the behavior of systems of queues. Many students ran the numbers by plugging in obvious values for service demand $D$: $2T$ for $S$ and $(1.8T)$ for $S'$. This led to a little bit of ugly arithmetic. I was OK with these answers, and was generally forgiving with the arithmetic.

But the important point is that this system is two separate stages/centers, and the CPU stage is the bottleneck. I spent a lot of our class time talking about the behavior of a single center, and modeling systems as a single center, but you should always remember that any complex system is a queue network. Some of the cases are easy, and this is one of them. Of course, the CPU can overlap with the disk in pipelining fashion:

That means that the peak throughput of $S$ is double what you would get without pipelining: $D = T$. Moreover, once we add a disk to make $S'$, we can have any two disk reads going in parallel. So for $S'$, the CPU is the bottleneck: it limits the performance of the system. So to determine the peak throughput or average queuing delay (the major component of response time), we need only to solve for the CPU.

The peak throughput of $S'$ is 25% higher than $S$ ($1/0.8$).

(b) Now suppose I add memory to $S'$ so that it can cache half of the data stored on the disks. What impact does this change have on the peak throughput? What impact does it have on the utilization of the disks? Again, please note any additional assumptions made in your answer. [20 points]

The cache cuts disk bandwidth utilization in half: if reads are random, then 50% of the reads will hit in the buffer cache and won’t go to disk. However, throughput is unaffected, since the CPU is the bottleneck. Capacity utilization is also unaffected: we still store all the data on both disks, even if some of the data is also in the cache.

Getting the bottleneck right was worth 10 points. There were other deductions for errors in the graphs in part (a), or omissions and/or garbling beyond some acceptable threshold.
Part 5. Quick thinking

You can answer each of these questions for full credit in a single sentence, or maybe two (with some commas). If you don't recall the relevant systems, just make something up based on what you know about the underlying principles. [30 points]

(a) Consider a SEDA (staged event-driven architecture) service implemented on a single server as a set of software components (stages) that interact via asynchronous messaging. SEDA works with the OS to allocate CPU time to stages. Each request to the service requires processing from some subset of the stages. Question: how does SEDA know if a stage is overloaded?

Each stage has a queue of incoming/pending tasks for that stage. If the stage is overloaded, the queue grows. **If the queue grows beyond some threshold**, then SEDA takes corrective action.

(b) If a stage of a SEDA service is overloaded, how does/should SEDA rectify the situation?

The corrective action is to (1) **drop tasks off the queue** (load shedding or admission control), or (2) **add another thread to the stage’s threadpool** (elastic provisioning). Thrashing is also an option, but it will not “rectify the situation”. When a task is dropped, the system should respond to the original requester with a “system too busy” message. When a thread is added, the stage gets (in essence) a 1/N boost in CPU time, where N is the total number of threads: a SEDA instance runs on a single node, and the OS allocates CPU “fairly” among the threads.

(c) How does Android prevent apps from spying on you, interfering with other apps, or stealing your data?

Android apps are isolated in sandboxes/lockboxes, of course, but they can communicate via binder messaging (RPC and intents). All cross-app interactions or platform data access (e.g., GPS sensors) are **initiated by firing an intent: at install each app declares the permissions required for the intents it receives or sends**. The user must approve any “dangerous” permission requests at app install time. The system blocks an intent if the sender or receiver does not have sufficient permission.

(d) Consider a program for classic Unix that does the following: create a new file, lseek to a large offset L (say 1 gigabyte) in the file, write a single byte, and exit. What is the size of the file? How much disk space is consumed to represent the file?

**The size of the file is L+1**: this is the size recorded in the inode and returned by `ls -l`: reads beyond this byte offset deliver an EOF. The disk space consumed is **one block containing the written byte, plus any metadata**: one inode and any double/indirect blocks needed for the map, depending on L.
(e) You remember your Lab #2 shell: it supports piped commands like `cat | cat | cat`. Why is it important for the parent shell process to close the pipe descriptors after forking? What could go wrong if the parent fails to close them?

Everybody should know this: I asked it on the first midterm. So I gave little or no partial credit on this. All file descriptors including pipe descriptors are reference-counted by the kernel. In the case of pipes, the kernel delivers an EOF on a read if all the data in the pipe has been read and no process has the pipe open for writing. The kernel delivers a signal (SIGPIPE) to notify/kill a writer if no process has the pipe open for reading. These mechanisms ensure clean shutdown of the pipeline if any of the processes exits. If the parent holds the pipe open, then these mechanisms will not work: the kernel does not know that the parent has no intention of reading or writing on the pipe descriptors it has open. If a child process exits, the downstream processes “hang”, and the upstream processes run to completion instead of stopping early. All bytes written into a pipe just “fall on the floor” if the intended reader is dead.

Note: for parts 2 and 5, the answer elements in bold were generally sufficient to get full credit. The rest of the solution answer is supporting detail, which was appreciated. In some cases I marked particularly good answers with a + and forgave some omissions on other answers.