Part 1. Cryptosystems

(a) Alice and Bob are back together. Today Alice wants to send Bob a message that is secret and also authenticated, so that Bob "knows" the message came from Alice. Alice and Bob have keypairs and each knows the other's public key.

How should Alice send the message? How should Bob validate the message? Briefly explain why your scheme works, and note any additional assumptions.

Alice can make the message M secret by encrypting it with Bob’s public key. She can authenticate M by encrypting it with her private key. So one approach is that she encrypts M with Bob’s public key to make M’, and then encrypts M’ with her private key to make M”, and then sends M” to Bob. Bob decrypts M” first with Alice’s public key to obtain M’, then decrypts M’ with Bob’s private key to obtain M. We assume that the private keys are truly private.

Alternatively, Alice can sign M (encrypt a hash with her private key) to authenticate it. Or Alice and Bob can establish a session key K (see 1c) and use K to encrypt and decrypt M, ensuring both properties. Half credit for each property: secrecy and authentication.

(b) One-way hash functions are fast and cheap. But what are they good for? I am looking for two specific examples of how they are used in practice, explaining how their properties are useful for those examples.

Example 1. Storing passwords as one-way hashes on a server. A client submits a cleartext password: the server verifies it by hashing the submitted password and comparing the result to the stored hash. If the hashes match then the submitted password must be valid, because it is “infeasible” that some other string could have the same hash as the real password. The advantage of storing hashes rather than cleartext passwords? If an attacker steals the server’s password file, it is “infeasible” to reverse the hash to obtain the passwords.

Example 2. Digital signature. Hash message M, encrypt the hash with a private key, and append the encrypted hash to the message. A receiver can decrypt the signature with the sender’s public key to obtain the sender’s hash, hash the received message, and compare the hash to the sender’s hash. If the hashes match then the message is authentic: the sender created it, and an attacker could not have corrupted it. An attacker could not corrupt the message without detection, because it is “infeasible” that the corrupted message would have a matching hash, and the attacker cannot fake the hash in the signature because it does not possess the sender’s private key.

(c) What are the defining properties of a session key? Why is it useful to establish a session key? Show how to use asymmetric crypto to establish a session key.

A session key is a symmetric encryption key that is shared by two or more communicating parties and is kept secret from everyone else. A session key is useful because symmetric crypto is cheap. To establish a session key: one side generates the key and uses one of the methods in 1(a) to send it to the other(s) in a message that is secret and authenticated. Example: SSL, HTTPS.
Part 2. Synchronization both ways

This part asks you to write code to synchronize a standard event/request queue, as discussed in class. **As always**: “Any kind of pseudocode is fine as long as its meaning is clear. You may assume standard data structures, e.g., linked lists: don’t write code for those.”

Threads place event records on the queue using the **put** method. (E.g., **put** might be called by a network connection handler.) A pool of worker threads **get** event records from the queue and process/handle those events. When a handler completes, the worker calls **get** again for the next event. The workers sleep if the queue is empty. **Note**: I am only asking you to code up the **put** and **get** methods, and **not** any thread/code that calls those methods.

(a) Implement **put** and **get** using monitors (mutex + condition variable).

```plaintext
put (Event e)
{
    Mx.acquire();
    Put e on queue;
    CV.signal(Mx);
    Mx.release();
}

get() returns Event e
{
    Mx.acquire();
    While (queue is empty)
        CV.wait(Mx);
    Get e from queue;
    Mx.release();
}

Some notes:
(1) It is OK to assume that the queue is unbounded. If you assumed the queue is bounded, then you have to do the wait and signal on both sides. (Equivalent to Soda Machine.)
(2) Strangely, a number of students treated this as a rendezvous problem like Sleeping Professor or Operators Are Standing By. There is no requirement that the event producer waits until an event consumer (worker) receives the message. Imposing such a requirement inhibits concurrency.
```
(b) Implement put and get using semaphores.

```
put (Event e)
{
    Mx.P();
    Put e on queue;
    Mx.V();
    Events.V();
}

get() returns Event e
{
    Events.P();
    Mx.P();
    Get e from queue;
    Mx.V();
}
```

Some notes:
1. If you assume the queue is bounded (OK) then the problem is equivalent to Soda Machine: you need another semaphore that counts the empty slots. You must initialize that semaphore to the number of empty slots. If you initialize it to 1, then you are making it a rendezvous problem again: the producers are forced to block if there is already an event in the queue. No good.
2. A surprising number of answers got the counts right but did not lock the queue. Cost: 10 points.
3. A correct solution uses semaphores alone. Bad errors: using anything that looks like a synchronized, wait, signal, or sleep, or manipulating the semaphore counts directly.
4. It is not necessary to maintain other counts, but I did not deduct points for it if done right.
5. The position of the ops on the Events semaphore are crucial. If you P on the Events semaphore while holding the Mx binary semaphore, the system deadlocks because a producer cannot acquire the Mx to put a new event. If you V on the Events semaphore before placing the new event on the queue, then a consumer may wake up too soon and get from an empty queue.

(c) Is this system vulnerable to deadlock? Why or why not?

I scored this with equal weight (13 points). I was looking for clear reasoning about deadlock. It was good enough to summarize the Four Prerequisites for deadlock and say that they are not all satisfied. In general, any convincing argument got full credit, and an argument is convincing if and only if it makes some reference to a circular waiting pattern, and shows awareness of various ways that might happen. If you just said "no" (or "yes"), then you lost some points, and you lost all of them if you were wrong. Wrong answers with some correct substance got partial credit. Note: it’s not deadlock when no events are produced (that’s degenerate but correct behavior), or when events are produced so fast that a consumer/worker cannot acquire the lock (that’s livelock).
Part 3. Condition variables

The following is a list of statements about condition variables. For each statement, discuss a specific example to show that is true (at least sometimes) or false (at least sometimes), or both.

(a) Code that waits on a condition variable must “loop before leaping”: check a condition after returning from wait, and wait in a loop until the condition is satisfied.

Key phrase: Mesa semantics. I was looking for a clear statement that a waiter awakened by a signal/notify might not run immediately: some other thread may acquire the mutex first and change the condition (predicate) that led to the signal. So the awakened waiter must verify the condition before proceeding. Of course there are exceptions: for example, in ping-pong there is no other thread that might change the wakeup condition after the signal. You got a “plus” for mentioning that, but I didn’t take off points for omitting it. Some answers talked about changing the value of the condition variable itself, which doesn’t make sense. (“Condition variables are not variables! They have no value!”) But I knew what you meant. I imagined that answers might refer to 2(a) as an example, but few did.

(b) Calls to wait on a condition variable may appear only in a locked critical section. It is given that Java (for example) throws an exception if this requirement is not met. I want you to explain why condition variables have this requirement, i.e., give an example of what could go wrong without it. Can condition variables ever be used safely without holding a lock?

Key phrase: missed wakeup or “wake up waiter” race. I was looking for a clear statement that if a thread checks a condition and then calls wait, it could miss the wakeup (signal, notify) if the wakeup comes before the wait. Holding the lock defends against this, presuming that the waker (the thread calling signal) also holds the lock at least when updating the condition state before the signal. In the case where there is no condition state (e.g., ping-pong), the waker must hold the lock through the signal call itself (see the problem “Tweedledum and Tweedledee”). CVs or sleep/wakeup cannot be safe without some form of lock. Some answers boiled down to general statements that mutexes are important to avoid races, or generally reiterated a “loop before leap” case of 3a, or repeated that wait requires a lock by definition without engaging the question of why. These answers got some partial credit.

(c) “Condition variable signal/notify is just a performance hint”: it is generally safe to use broadcast/notifyAll instead of signal/notify, but it may be slower.

Key phrase: thundering herd. The point here is that if a thread waits for a condition (predicate) to become true, and if it checks the condition on return from wait before proceeding as in 3(a), then it can handle spurious wakeups safely. If every waiter thread loops before leaping in this fashion then it is always safe to signal a waiter thread, whether or not the thread should proceed past the wait. So it is safe to replace any signal with a broadcast, which has the effect of signal, but signals all waiting threads and not just one. Again, 2(a) is an example. But: broadcast may be less efficient (slower) due to unnecessary context switching. Some answers addressed half of the issue (slow or safe) and ignored the other half, and/or waffled on whether it is always safe. These answers received partial credit.
Part 4. File systems and storage

Please answer the following questions crisply without writing a book. Key words will do. Feel free to draw pictures!

(a) What factors influence **access time** for storage media? How?

Key phrases: seek time, seek distance, platter width, rotational delay, rotational speed (RPM), solid-state storage (SSD, e.g., flash) vs. rotational media (disk, HDD). This question was asking about dynamics of storage media: disks. Mentioning SSDs was a plus, but wasn’t necessary. Any clear reference to mechanical delays in accessing a spinning disk got full credit, perhaps with a minus. Many answers touched on various related but orthogonal issues: caching, transfer time (I/O bus speed, DMA, spindle speed and/or block size), disk scheduling policies and/or queuing delays. Some drifted further afield into file system placement choices (e.g., cylinder groups), or file system layout choices (e.g., levels of indirection). These were worth varying degrees of partial credit. As always, irrelevant information was not penalized, but I knocked some points for false statements or unacceptable vagueness suggesting a lack of confidence in the topic.

(b) [Optional extra credit] How does **adaptive internal priority** help to improve **disk bandwidth utilization**?

Key phrases: multi-level feedback queue, SJF or SCTF scheduling. This question was asking about the common scheduling heuristic of increasing a thread’s priority when the thread blocks before its quantum expires (e.g., waiting for an I/O completion). These threads are often given priority for the CPU when they wake up, since they often repeat the pattern: run for a brief period to initiate another I/O request before blocking again. This policy is a common way to reduce the likelihood that a disk with work to do sits idle because of delays in issuing the I/O requests to it. The key to maximizing throughput is to keep all resources busy (utilized) whenever there is work for them to do.

I got a few blank answers, a few confessions of puzzlement, and more than a few creative wild guesses issued with a straight poker face. Thanks for the entertainment.

(c) What is the purpose of **indirect blocks**? Why do some files have them and others do not?

Key phrases: skewed tree, large files. Classic inodes contain sufficient block map entries to represent small files, but no more than that. Small files don’t need them, and omitting them keeps the inode small. If a file is large, then its inode points to an indirect block, which is a block containing additional map entries. With this structure small files are cheap and large files are possible. Some other points worth a plus: very large files use double indirect blocks; the hierarchical block maps are similar to (or equivalent to) page tables; block maps represent sparse files (and sparse block spaces or address spaces) efficiently. As always, irrelevant information was not penalized, but I knocked some points for false statements or unacceptable vagueness.
Part 4. File systems and storage (continued)

(d) How does the classical Unix file system structure use reference counts?

Key phrase: hard link. A name entry in a directory (folder) references an inode and is called a “hard link” in Unix. Regular files may have multiple names, created and destroyed with the link and unlink system calls. Each on-disk inode holds a link count of the number of links that reference it: the system call code increments and decrements this count as it creates and destroys links. When the count reaches zero the file may be destroyed: the inode and blocks are freed. Various related tidbits are worth a plus. Unix also supports soft links (symbolic links), which are not reference-counted. That makes them more flexible (the target of a link can change without the need to change the link itself) and also more dangerous (the target may be destroyed while a link to it exists, leaving a dangling reference). In addition, active (in-memory) inodes have a second reference count that goes to zero only when no processes/threads have the file open or are operating on the inode in any other way. But few mentioned that and it wasn’t necessary to do so.

(f) What is an inode file? What purpose does it serve?

A majority of answers answered a different question that is related but distinct: what is an inode? Those answers were worth half credit minus, if correct and sufficiently detailed, and may also win back some points that might have been lost from vagueness on 4(c).

Question 4(f) is instead asking about the inode file structure used in various advanced file systems, primarily NetApp’s WAFL (discussed in some depth) and log-structured file systems (mentioned only briefly). The inode file is a file containing all on-disk inodes in the volume (except for the inode of the inode file itself, which is linked from the file system root). The inode file has a block map that gives a level of indirection, enabling the file system to change the number of inodes and/or their placement on the disk. This technique is a cornerstone of the “write anywhere” file layout (WAFL).

(e) Raw disk bandwidth (“spindle speed”) continues to improve exponentially, while seek latencies are not improving much at all. What is the impact of this trend on effective bandwidth? Sketch a graph, assuming a constant stream of requests with some fixed block size.

This wasn't intended as a trick question, but I had to grade it leniently. It contains the same information as a graph on the slides, but is “asked backwards” so that the line starts near one and asymptotically approaches zero, like another graph on the slides. A slide gives the relevant formula as \( \frac{b}{s(B+b)} \): as spindle speed (B) grows this value approaches zero if access time (s) and block size (b) are held constant. I gave half credit minus for any suitably shaped (asymptotic) line, and full credit if the answer had enough information to demonstrate understanding of the relevant factors (e.g., fragments of a formula). You got an extra plus if you actually got it right.
Part 5. Scheduling for response time

This question asks you to modify your answer for Part 2 (event queue put and get) to reduce average response time. Suppose there are three types of incoming events with different average service demands: the handler for type A events takes one time unit to complete, type B events take 2 units, and type C events take 3 units. You may assume each event record is tagged with exactly one type code (A, B, or C). Your solution should be free from starvation.

Key phrase: SJF. The best approach is to give scheduling preference to the cheaper events first, which is a variant of SJF scheduling. Any switch of scheduling order for a pair of jobs to give priority to a short job over a longer job provably reduces average case response time. (A before B and B before C. A few answers got the order backwards and lost some points.)

However, a pure SJF approach will starve. If you did not address starvation then the answer was worth 25 points at most.

I accepted any reasonable approach to addressing starvation. These answers give a good reading on clarity and creativity. A common approach is to fall back to FIFO/FCFS if the number of queued B or C jobs exceeds some bound. There are various other approaches (e.g., bump priority based on wait times, or add in some randomness). In general these should receive full credit.

Another common answer is to suggest preemption, e.g., round robin, rather than giving priority to the short events. This alternative complicates things and raises a number of issues that the answers generally didn’t address. For example, there is a possibility of deadlock if the queue is bounded, since a worker can block inserting a preempted event back on the queue. Also, the effect of preemption on response time may depend on the request stream. I gave at most 75% credit for that answer.

Another answer that showed up was to use different worker pools for the different event types. The problem here is that it doesn’t work. For example, other things being equal, this approach will increase response times if all of the requests that arrive have the same type. But it doesn’t starve! I think I scored these at 15-20 points.

A few answers gave me a bunch of code with no explanation. You might lose points if the synchronization is broken, or if it isn’t clear how the code meets your goals.
Top students at Duke are competitive with the top students at any university in the country or the world. Learning is a lifelong process, but you will be well-prepared for anything you want to do in computer systems. You made some mistakes, but I’m rooting for you.

Grading is not an exact science. These scores leave plenty of room for improvement, but they’re satisfactory. Swings of 10 or even 25 points aren’t really significant unless there’s a clear pattern. I consider these exams to be in a wide B range. I see lots of evidence of talent, hard work, and accomplishment in many of these exams.

These scores are disappointing. I think some of you are slackers, and some are making an effort but are struggling or lost a bunch of points somewhere for whatever reason. I know that you are all stressed and busy and people make mistakes under pressure. But depending on other factors, I consider these scores consistent with C range, maybe up to B-.

These are people who are making it in a difficult class. These are grades to be proud of. There is always room for improvement. Depending on other factors, I consider these to be A- grades.

These scores indicate real trouble, but given the nature of the specific mistakes, I regard them as passing grades.