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.

(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.

(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.
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).

```c
put (Event e) {
}
```

```c
get() returns Event e {
}
```
(b) Implement **put** and **get** using semaphores.

```java
put (Event e) {

get() returns Event e {

}
```
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.

(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?

(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.
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?

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

(c) What is the purpose of indirect blocks? Why do some files have them and others do not?
Part 4. File systems and storage (continued)

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

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

(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.
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.