1. What rules should be used to determine whether functionality should be implemented inside the OS, rather than outside of it (e.g. in library or application code)?

There should be as little code implemented in the OS as possible, as writing and maintaining kernel code is very expensive. Thus, the only code that should be implemented in the OS is code that requires the use of privileged instructions, code that requires access to kernel data structures, and code that has to do with maintaining the security and privacy of the system.

8/10

2. (a) Why is ABI compatibility preferable to API compatibility?

ABI compatibility is preferable to API compatibility because the ABI binds the API to a specific Instruction Set Architecture. Thus there is no ambiguity in which instructions should be used to implement the API. Furthermore, the ABI specifies things like sizes of data types and calling conventions that simplifies the implementation of the API.

2/5
portability?

(b) When would it be necessary or reasonable for two OSs that support the same APIs to not support the same ABIs?

It is reasonable when the two OSs use different Instruction Set Architectures. This means that the two OS's supported operations are different, and because of this the 2 may have different ways of implementing the same API.

5/5

3. describe or illustrate (in detail) the sequence of operations involved in the processing of a system call trap, and its eventual return to the calling application.

When there is a system call trap, the application gets switched to kernel mode, and the current PC and PS are saved onto the process's kernel stack. The trap is used as an index into the 1st level trap handler, which saves the current process's registers onto the kernel stack. The 1st level trap handler then indexes into the system call dispatch table, which indexes into the 2nd level trap handler. The 2nd level trap handler actually executes the instructions that deal with the trap. In the case of a system call, it will execute the system call on behalf of the user if the system call is a valid operation. The 2nd level trap handler then returns to the 1st level trap handler, which pops the process's saved registers off the kernel stack and restores them. Finally, a return from trap instruction is executed, which pops the PC and PS off the kernel stack. When the application is switched back to user mode and will continue executing the instruction after the trap.

10/10

4. (a) Define "starvation" (in scheduling)?

Starvation is when a process never gets access to the CPU to run due to other processes always getting the CPU before it.

✓

(b) How can it happen?

Starvation can be the result of a poor scheduling algorithm, in which it is possible for one process to always "lose" when the scheduler is making its decision of which process to run next. For example, in a shortest job first scheduling policy, a process that needs to run for a long time may never get the CPU and will starve if there are always other jobs requiring a shorter runtime that the scheduler can schedule before it.

✓

(c) How can it be prevented?

Starvation can be prevented by preempting processes. Instead of allowing a process to run until it is finished or it yields the CPU, force the process to give up the CPU by using timer interrupts, which performs a context switch and allows other processes to run on the CPU.

✓

10/10

5. (a) What is coalescing (in memory allocation)?

7/10

Coalescing is when multiple free blocks that are nearby in memory are combined to form one bigger free block.

(b) What problem does it attempt to solve?

Coalescing attempts to solve the problem of external fragmentation, in which memory is continuously broken up into chunks to allocate memory for processes, leaving some parts unusable when the chunks left behind after memory allocation are too small to be used for another process.

-1

(c) What memory allocation factors might prevent it from being effective?

One factor that may prevent coalescing from being effective is if the memory allocation policy allocates memory in such a way that the free chunks of memory left behind are either very small. If the chunks left behind are small, then coalescing would not really help because even combining these free chunks may not be enough memory for a new process to be allocated. Also, traversing the whole free list to find free chunks of memory is very slow.

-2

(d) What memory allocation design might make it unnecessary?

Paging may make coalescing unnecessary because it basically eliminates external fragmentation. Paging allocates a fixed size block for each page a process has; thus, because the pages are fixed size, there won't be any cases where there are free chunks of memory in between 2 chunks of allocated memory. There isn't really a need to collect free space to form a bigger free space block because when you need memory for something, you can simply page something out of physical memory and into the swap space.

6. (a) List a key feature that global LRU and Working Set algorithms have in common. (0 points for both are replacement algorithms)

8/10

Global LRU and working set algorithms both try to replace the page that would have the least performance cost by not evicting any pages that will be needed in the near future.

(b) List a key difference between working set algorithms and global LRU.

Global LRU is implemented using a clock algorithm. The clock hands point to different pages in memory, and if that page has not been accessed recently then it is evicted. Working set algorithms are implemented using page stealing algorithms, in which processes that need more pages take pages from processes that don't need as many pages.

-2 also clock

(c) Are there differences in the associated hardware requirements? If so what are they? If not, explain why not.

There are not any differences in the hardware requirements. Both algorithms require a reference bit in the page table entry that tells whether or not a page has been accessed recently. Both the LRU and the working set algorithms base their decisions of who to evict off of this.

8

Given that we need to perform some computations in parallel ...

(a) Give two characteristics that would lead us to choose multiple processes.

One situation in which you would use processes is if you don't want the 2 processes to have the same address space and be able to access each other's data. Another situation in which you would use processes is if you want to communicate with someone else via a socket. ✓ 2

(b) Give two (different) characteristics that would lead us to choose threads.

One situation in which you would use 2 threads is if you want the threads to have access to the same address space. For example, if there is a counter that you want to update in parallel, then you would use threads because both threads need to have access to the shared counter. Another situation in which you want to use threads is if you have some operation which will block, like performing an I/O operation. The scheduler can simply switch to have another thread run on the CPU, without the costs of a process context switch that would need to be done had we used processes. ✓ 10

8. The text gave three criteria in terms of which lock mechanisms should be evaluated. In class this list was expanded to four criteria. List and briefly describe three of those criteria AND provide an example of a real locking mechanism that does poorly on that criteria.

(a) One criteria is correctness. This means whether or not the locking mechanism actually provided mutual exclusion, meaning only one thread was allowed to enter the critical section at a time. A mechanism that does poorly in this criteria is disabling interrupts. This doesn't work in the context of multiprocessor systems. Disabling interrupts and letting one thread run non-preempted on one core doesn't mean a thread on another core can't access the critical section. 3

(b) Another criteria is fairness. This means that all threads that are waiting on a lock will eventually get the lock at some point. An example of a locking mechanism that fails this criteria is spinning and yielding. In this mechanism, a thread periodically checks if the lock is available. If it's not, then the thread yields the CPU to another thread. This mechanism is not fair because it all depends on when the thread wakes up. A lock could be available a lot, but if a thread happens to wake up everytime when the lock is held by some other thread, then it will never acquire the lock. 3

(c) Another criteria is performance. This means that there is not any significant overhead added in acquiring and releasing the lock. A locking mechanism that performs poorly is spinning and yielding. Everytime a thread wakes up and the lock isn't available, it yields to another thread, and there is significant overhead associated with this because on switching to another thread, the OS needs to save and restore things since PC, registers, etc, which take up a lot of time. 3

9. Arpaci-Dusseau developed a simple producer/consumer implementation along the general lines of:

```
consumer() {
    for( int i = 0; i < count; i++ ) {
        while(empty)
            wait for data to be added
        get()
        wake the producer
    }
}

producer() {
    for( int i = 0; i < count; i++ ) {
        while(full)
            wait for data to be drained
        put()
        wake the consumer
    }
}
```

5/10

He went through several steps (exploring deadlocks and other race conditions) to develop a correct implementation based on pthread_mutex and pthread_cond operations. While correct, his final implementation seemed quite expensive, getting and releasing locks, and signaling condition variables for each and every get/put operation.

Update/Rewrite the above code to include all of the following:
(a) correct use of pthread_mutex and pthread_cond operations
(b) correct mutual exclusion to protect the critical sections
(c) correct emptied/filled notifications to the producer and consumer
(d) eliminating per character locks and notifications

```
void consumer(int count) {
    while (empty)
        pthread_cond_wait (&empty, 0);
    pthread_mutex_lock (&lock);
    get();
    pthread_mutex_unlock (&lock);
    pthread_cond_signal (&full, 0);
}

void producer (int count) {
    while (full)
        pthread_cond_wait (&full, 0);
    pthread_mutex_lock (&lock);
    put();
    pthread_mutex_unlock (&lock);
    pthread_cond_signal (&empty, 0);
}
```

(a) What is meant by "finer grained locking"?

10/10

Finer grained locking means instead of having one lock for a big critical section, break the critical section up into parts and give each of these sections their own individual lock.

(b) Why does it reduce resource contention?

It reduces resource contention because now more threads can enter the critical section instead of locking all other threads out when one thread goes into a lengthy critical section, shorter critical sections allow more threads to come in and do the work they need to be still maintaining mutual exclusion. This, more work can be done in parallel.

(c) What are the costs of finer grained locking?

The costs of finer grained locking is that there is more overhead associated with acquiring and releasing the locks, since there are now more of them. Also, finer grained locking makes the program more complicated and harder to keep track of because of all the additional locking and unlocking that needs to take place.

(d) Suggest another way of reducing contention on a single (unpartitionable) resource.

Another way of reducing contention is to make the number of accesses to the critical section less frequent. An example of this is the sloppy counter, where multiple threads are trying to update a shared counter. To reduce the frequency of accessing the critical section of updating the counter, each thread can have its own local counter that can be updated without synchronization problems. Then, when the threads' local counters reach a certain threshold, the value of the local counter is pushed to the shared global counter. This reduces contention to the shared counter because threads are now accessing it less frequently.

XC. We are designing an inter-process communication mechanism that provides very efficient (zero-copy) access to very large messages by mapping newly received network message buffers directly into a reserved set of page frames in the user's address space. As new messages are received (and the buffers mapped-in) the OS updates a shared index at the beginning of the reserved area to point to the newly added pages.

The problem we are currently wrestling with is how to reclaim/recycle old buffers and page frames after the application has processed them. One group of engineers asserts that garbage collection would provide the most convenient interface. Another group engineers asserts that garbage collection would be expensive to implement and result in poorer memory utilization.

(a) When, specifically, would the OS initiate garbage collection?

The OS initiates garbage collection when there is no more free memory in the system. It goes through the memory and finds portions of memory that are no longer being accessed, and then frees them.

2

(b) Describe an approach that would permit the OS to automatically determine which buffers/page-frames were "garbage" (be specific).

One way would be to have a bit in the page table entry that specifies whether or not we are done using it. When the bit is set to 0, then the OS would know that the page frame is garbage.

2

(c) What would the OS have to do to make sure that the process would not attempt to re-use a buffer that had been garbage collected?

The OS could turn off the valid bit for that page frame so that when a process tries to access it, it will go into a trap.

2

(d) Describe an alternative implementation (without garbage collection)?