1524

CS 111 Spring '16 - section 1   Exam #1 (4/27/16)

Name Greg Giebel   exam # 76

Student ID # 004411272   seat row k   seat col 10

All Questions (except the extra credit) are of equal value. Most questions have multiple parts. You must answer every part of every question. Read each question carefully, and make sure you understand EXACTLY what it is asking for. If you are unsure of what a question is asking for, raise your hand and ask.

Spend more time thinking, and less time writing. Short and clear answers get more credit than long and vague ones. Write carefully. I don't grade for grammar, but if I can't read or understand your answer, I can't give you credit for it.

1. 8/10
2. 7/10
3. 8/10
4. 10/10
5. 6/10
6. 9/10
7. 8/10
8. 7/10
9. 4/10
10. 8/10

total 75

extra credit 4

79

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)?

8/10 ~~The main rule to follow here is Mechanism/Policy Separation. If functionality is for a mechanism, meaning that it implements How something is done, it is more likely to end up in the OS. Policies, or who can do something,~~

Code should be in the OS if it uses privileged instructions, controls the allocation of reusable resources, or maintains abstractions for a user.

security?

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

ABIs bind APIs to an instruction set architecture (ISA). If something is ABI compatible, it will run in the same way on any machine with the same ISA. API compatible code may run differently on machines with different ABIs. So if two machines have a piece of ABI compatible code, it will run identically on both. If the code is only API compatible, it may run differently.

2/5

recompilation?

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

It would be necessary if the two OSs had different ISAs. The same ABI cannot be used to bind the API if the instruction sets differ.

5/5

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 something happens in program execution that causes a trap, the following occurs:

1. Save program execution state (registers, program counter, stack pointer, etc.)
2. Syscall and call signal handler for trap into OS
3. Call trap handler
   - push its code onto stack
   - execute
   - return to signal handler
4. Finish executing signal handler which may
   - terminate process
   - restore execution state and return to calling process

8/10 −2 2nd level handler

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

Starvation is when a thread/process never gets to execute because other threads/processes are continuously scheduled before it. ✓

(b) How can it happen?

A thread/process can be starved if
- SJF scheduling with large number of short processes prevent a longer running process from executing ✓
- it gets stuck in a low priority queue with constant jobs occurring in higher priority queues

(c) How can it be prevented?

- FIFO scheduling (has to happen eventually)
- Round Robin (again, will get a turn at some point)
- periodic priority boosts in a priority queue system (boost bumps it to high prio, forcing it to get a turn)

10/10

6/10

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

Coalescing is taking adjacent chunks of free memory and combining them into one chunk.

(b) What problem does it attempt to solve?

external fragmentation

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

- if fragmentation is distributed it may not help much
- if fragmentation is mostly internal it may not help much

-2

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

-2

slab allocation as it makes a uniform chunk size that can be allocated (all fragmentation internal)

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

9/10

Both use a clock algorithm ~~to~~ check which pages are least recently used (and then pages out said page).

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

The Working set algorithm uses a water mark to determine if a process can give up pages ~~that supercedes~~ the LRU search. Global LRU just takes the page used least recently by any process. Working set also does per process LRU, so it adjusts for processes receiving less CPU time.

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

Working set tends to need a few extra bits in the header so that it can check how long h/w? has been since a page has been accessed in terms of the time the process has actually run. LRU just needs an accessed bit.

-1

Given that we need to perform some computations in parallel ... 8

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

- rarely required communication (sending of signals) ✓
- long running program (not worth the added overhead for short programs) -2

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

- need to access or update same data consistently ✓
- need large number of parallel executors ✓
(much cheaper to create lots of threads than lots of processes)

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. 7

(a) Correct - does it properly "lock" data so only 1 thread (or however many you allow) can use it at once

2

does poorly - atomic test and set with sleeping instead of spinning (sleep wakeup races)

(b) Fair - does every thread get the lock in a timely manner if it wants the lock

2

does poorly - mutex; random thread will get the lock

(c). Productive - the threads trying to get the lock do not increase the time until the lock is released

3

does poorly - spin lock; wastes CPU cycles spinning, preventing the thread with the lock from executing

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
    }
}
```

4/10

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:

-2 (a) correct use of pthread_mutex and pthread_cond operations
-2 (b) correct mutual exclusion to protect the critical sections
(c) correct emptied/filled notifications to the producer and consumer
-2 (d) eliminating per character locks and notifications

```
pthread_mutex_t lock;
pthread_cond_t full, empty;

consumer(count){
 while(){
   pthread_cond_wait(full, &lock);
   for(int i=0; i< count; i++){
     get(); }            inside?
pthread_mutex_unlock(lock);   where lock?
pthread_cond_signal(empty);}}

producer(count){
 while(){
   pthread_cond_wait(empty, &lock);
   for(int i=0; i<count; i++)
     put();              inside?
   pthread_mutex_unlock(lock);   where lock
   pthread_cond_signal(full);}}
```

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

8/10

only locking around the critical section of code

ex: locking a for loop within a function that is the critical section as opposed to locking ~~the~~ whole function.

(b) Why does it reduce resource contention?

The lock is held for less time, reducing resource contention

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

Locks and unlocks often occur more frequently, adding over head.

ex: locking a few lines in a for loop as opposed to the whole loop means you lock + unlock every iteration

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

Reduce time using the resource

ex: updating a private local counter and periodically using that to update a shared global counter, as opposed to constantly updating the global counter

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?

When the processes try to send a message that will not fit in the allocated space in memory.

2

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

Note: if only 1 unread at a time, do not need 2nd index. Just free all before the 1st index

add a second index that points to the most recently read ~~message~~ (specifically the start). Any page before that index is garbage. This also keeps the most recently read message in case it is still in use

(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 would have to change the valid bit on the PTEs associated with that data so the process would know it cannot access the data

2

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

Just overwrite old messages. If the message does not fit in available space, write it from the beginning of the memory chunk. This assumes our allocated memory chunk is much larger than the size of a message so we can fit a bunch before overwriting.