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

Name Changfei Shi exam # 3

Student ID # 304441183 seat row A seat col 4

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

total 88 (90)

extra credit 2

ALL IS LOST

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)?

- whether the functionality could result in interference with other ~~wi~~ processes/system resources (disk/memory/etc.)
- whether ~~proces~~ functionality could change the OS behavior (modifying trap table/changing protections/etc.)
- ~~Speed of functionality - OS code~~
- whether functionality requires other OS-specific functionality frequently (ex) very frequent I/O)
- whether it is fundamental to working of an OS ~~[illegible]~~

6/10

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

- an ABI-compatible systems, any program executable ~~wi~~ that conforms to the ABI specification can run on that system, no linking/compilation necessary
- an API-compatible program only guarantees conformity to the methods/objects specified by that interface ~~requires~~ only applies to code, ABI applies to application binary

5/5

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

The two OSs have different instruction sets / support different addressing types / etc., so the actual interactions with hardware are different. However, the OSs want to implement the same functionality guaranteed by the API.

(and support the same programs)

5/5

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

~~At runtime~~ At load time: Trap table with the trap handlers are loaded.

instruction trap occurs

↓

~~First level handler is called~~

OS recognizes kernel-mode instruction and traps

↓

first-level trap handler saves registers, places program counter/etc. onto the stack of the trapping process

↓

calls kernel-mode, ~~the~~ which verifies the process and performs the ~~op~~ instruction

⟶ Return from OS to trap handler which restores program state / to the next instruction

Process data | PS/PC

−2 2nd level handler

8/10

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

infinite wait times on finite operations - a process that ~~blocks for a~~ contends for a resource never gets the resource. (in this case, time)

✓

(b) How can it happen?

- priority scheduling - ~~(SJF~~ (ex) SJF) A ~~program~~ process with low priority may never get to run if ~~programs not~~ processes with higher priority keep arriving ~~and are~~

✓

(c) How can it be prevented?

- queues, which guarantee that each member in queue will eventually reach the front.

✓

- Dynamic priority - adjusting priority of ~~the~~ scheduled processes based on wait time.

10/10

9/10

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

merging of two consecutive chunks of free memory into 1 larger chunk

(b) What problem does it attempt to solve?

external fragmentation

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

~~distributed fragmentation, so that a fragmentation of~~ No two consecutive chunks of memory are ever free at the same time. - caused by high memory utilization.

prob but not cause

(d) What memory allocation design might make it unnecessary? ~~each~~ fixed-size

fixed-size pools of memory ~~for specific~~ ~~for specific~~ chunks that are merely left ~~unused~~ until the next allocation rather than being coalesced

memory

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

~~circular clock that visits~~ "clock" that visits every page in memory ~~and checks the referenced bit~~ [illegible] in a circular fashion and checks the reference bit of every page

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

Working set LRU keeps track of the process id associated with every page.

; timestamp use by that proc

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

Yes Working set LRU requires the addition of a per-page counter variable to store the difference ~~in the process's~~ between the associated process's last runtime and the last accessed time of the page. - this might ~~require extra space in PTEs~~ a more robust clock

-1 not h/w

10

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

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

- need for privacy between the two computations ✓

- ~~simplicity - no need to worry about~~

- robustness - when 1 ~~pa~~ computation fails, we want other to keep running ✓

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

- sharing of resources - file descriptors, memory, etc. ✓

- performance - Thread context switch costs less than a process context switch in terms of performance ✓

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) correctness - locking mechanism allows ~~a~~ only the holder of the lock into its corresponding critical section / object 3

Ex) Interrupt disabling on multiprocessor systems fails to disable interrupts for all processors

(b) fairness - processes/threads waiting on a lock will eventually obtain the lock, ensuring no starvation 3

Ex) Spin lock - waiting process/thread never guaranteed access to the critical section ~~...~~

(c) efficiency / performance - lock runs w/o ~~what~~ slowing down the runtime of the process 3

Ex) Spin lock on a uniprocessor system - spinning by each thread waiting on a lock waste processor time that could be used to run the critical section and unlock of the thread holding lock.

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

7/10

```
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
    }
}
```

& Assume get() & put() are replaced w/ appropriate buffered portion

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
-1 (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

```
consumer( int cunt){
    int i=0;
    while(1){
        pthread_mutex_lock(&lock);
        while(empty)
            pthread_cond_wait(&lock, &full);
        while(!empty && i<cunt){
            get()
            i++;
        }
        pthread_cond_signal(&empty);
        if(i==cunt)
            return(); unlock
    }
}
```

4

```
producer(int cunt){
    while(1){
        pthread_mutex_lock(&lock);
        while(full){
            pthread_cond_wait(&lock, &empty);
        }
        while(!full && i<cunt)
            put();
            i++;
        }
        pthread_cond_signal(&full);
        if(i==cunt)
            return;
    }
}
```

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

10/10

Additional locks for separate parts of the object / program, rather than a single lock

(b) Why does it reduce resource contention?

- A single lock may bottleneck different threads that want access to different parts of the object - ex) a hash table. A lock for different ~~parts ensures that~~ resources ensures that waiting only occurs for multiple threads that want that specific resource.

buckets in

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

- Finer-grade locking may be more prone to error due to the increased number of locks (increasing chances of deadlock, incorrect behavior)
- Additional locks have performance costs - ex) accessing a linked list w/ a lock for each node is much, much slower than a simple list traversal.

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

- Try to move as much code within the lock outside of the lock (using thread-local variables and only running locked section when local variable reach a threshold) reducing the amount of time spent inside the locked region.

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?

- memory allocation fail's due to too little memory available

2

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

(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?

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