CS 111 Midterm 05/04/2017

Name [redacted] Seat Row A

Student ID # [redacted] Exam # 12 Seat Col 4

All questions are of equal value. Most questions have multiple parts. You must answer every part of every question. Read each question CAREFULLY; Make sure you understand EXACTLY what question is being asked and what type of answer is expected, and make sure that your answer clearly and directly responds to the asked question.

Many students lose many points for answering questions other than the one I asked. Misunderstanding a question may be evidence that you have not mastered the underlying concepts. If you are unsure about 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, rambling or vague ones. Write carefully. I do not grade for penmanship, spelling or grammar, but if I cannot read or understand your answer, I can't give you credit for it.

1.

2.

3.

4.

5.

6.

7.

8.

9.

10.

SubTotal:

XC

Total:

1: (a) Consider an after-market application, developed and shipped separately from the OS. What could happen if the OS vendor relased a new OS version with a non upwards compatable API (and associated ABI) change?

Future versions of that OS might not be able to support features specified in the current API

(b) What would the application developer have to do to deal with this?

They would have to rewrite their programs to become compliant with the new OS

(c) Explain how/why interface specifications, designed and written independently from the current implementation, might have affected this situation.

If the application was designed to an interface, the developer wouldnt have to rewrite their code because, the code adhered to the interface despite the change in the underlying ISA

2: (a) What is a resource contention convoy?

where threads experience unbounded wait times while waiting to enter a critical section

(b) Under what circumstances is one likely to form?

when trying to enter a coarsely granulated locked section of code
=> a huge piece of code or data structure that is locked

(c) Suggest two distinct approaches to eliminating (or SIGNIFICANTLY improving) the problem.

finely granulated locking – rather than locking off a whole section lock smaller parts within that section so more threads can be in that section at a time

reducing size of critical section – make the code of the critical section smaller so locks are given up more frequently

3: (a) List two different types of events that might cause a running process to be preempted.

A process might be preempted so that
- another process can run
- the current process could be waiting for some I/O to complete before it can run again

(b) List two different types of events that might cause a running process to become blocked.

- A running process may be blocked if it is waiting for some I/O to complete
- A process deliberately chooses to yield the CPU

4: (a) Identify three key criteria in terms of which mutual exclusion mechanisms should be evaluated.

Correctness - implementing mutual exclusion so that only one thread is in a critical section at a time

Fairness - waiting threads have a chance to run in the critical section

Efficiency - low overhead

(b) Evaluate spin-locks against these criteria.

Correctness - yes. spin locks use an atomic check and set mechanism that works even if that thread is preempted

Fairness - no. The mechanisms for spin-locks have nothing to do with the policies that choose the next thread to get the lock

Efficiency - not quite. Spin locks still waste CPU cycles while checking the availability of a lock

(c) Evaluate interrupt disables against these criteria.

Correctness - no.

Fairness - no. does not affect the policies that choose the next thread to get the lock

Efficiency - no. turning of interrupts is a priveleged instruction that must be done in the OS. Switching from user mode to kernel mode is an expensive operation and would add a lot of overhead

(d) Evaluate mutexes against these criteria.

Correctness - not always, checking and setting is not done atomically and this could lead to unpredictable behavior if a thread is preempted in the middle of checking and setting

Fairness - no. does not affect the policies that choose the next thread to get the lock

Efficiency - no. wastes CPU cycles checking for the lock to be available

5: (a) Describe a major capability (not merely memory savings) of DLLs that cannot be achieved w/mere shared libraries.

Dynamically loaded libraries are loaded at runtime when a process decides that it needs some functionality that it was initially intended to do.

(b) Describe a specific situation where and why this capability would be NECESSARY.

A web browser needs to process data from videos but is unfamiliar with the video's format, because this format didnt exist at the time the browser was created. A plug-in would be required to perform the requested functionality

(c) Briefly describe a significant mechanism that is not required to support shared libraries, but is required to support DLLs ... and very briefly explain why this additional mechanism is necessary.

6: (a) What is the primary advantage of shared memory IPC over message communication?

shared memory IPC is faster than message communication

(b) Why does it have this advantage?

loading data into a shared memory space makes that data almost instantly available to other processes
It is as if you are putting that data directly into another process's address space

(c) List TWO advantages of message IPC over shared memory, and why each cannot reasonably be achieved with shared memory IPC.

1) Allow communication between processes that aren't on the same machine
If two processes are not on the same machine, there is no way they can share the same memory

2)

7: (a) list two pieces of information that a variable partition free-list must keep track of that might not be needed with fixed partition memory allocation.

1) the size of the free chunks of memory

2) the location of free chunks of memory

(b) list two operations that a variable-partition free list has to be designed to enable/optimize (zero points for "allocate and free").

carve - take a sufficiently large free chunk, slice off the requested amount of memory, the rest goes back on the free list

coalesce - combining adjacent free chunks of memory into larger chunks to counter the effect of external fragmentation

(c) list an additional piece of information we might want to maintain in the free list descriptors to detect or prevent common errors, and briefly explain how the information would be maintained, and how it would be used to detect or prevent a problem.

Keep a count of how many references are made to a chunk of memory. If the reference count reaches zero, that chunk of memory is freed. This helps counter the effect of memory leaks where a programmer forgets to deallocate memory after they are finished with it

8: Consider a server front-end that receives requests from the network, creates data structures to describe each request, and then queues them for a dozen server-back-end threads that do the real work. Sketch out server and worker-thread algorithms that use semaphores to distribute/await incoming requests, and protect the critical sections in queue updates.

```
server back-end { // consumer
    for the number of requests to receive
        sem_wait(&full)
        sem_wait(&mutex)
          get()
        sem_post(&mutex)
        sem_post(&empty)
}

server front-end { // producer
    for the number of requests to send
        sem_wait(& empty)
        sem_wait(& mutex)
          put()
        sem_post(&mutex)
        sem_post(&full)
}
```

9: Briefly list the sequence of (hint: 8-10) operations that happens, in a demand-paging system, from the page-fault (for a page not yet in main memory) through the (final, successful) resumption of execution.

(a) describe the hardware and low level fault handling.

- trap into the OS, switch from user mode to kernel mode, push PC/PS on kernel stack and go to the page fault handler
- locate where the missing page is on disk

(b) describe the software lookup, selection, I/O, updates.

- once the missing page is found, find a place to swap it in main memory
  - to do this we need to use global LRU or Working Set algorithm to select a page to swap out of main memory
- update the page table to accomodate for the newly swapped page
- schedule the I/O to perform the swap
- if the dirty bit is set, update the swapped out page on disk
  if the clean bit is set, no need to update

(c) describe the return/resumption process.

- 1st level trap handler restores the registers and state of the calling process
- PC is set to retry the instruction that resulted in a page fault
- restore user mode and begin execution

10: (a) Why is each Linux Condition Variable ~~pared~~ paired with a mutex? Briefly ~~escribe~~ describe the race condition that is being managed.

~~So two threads~~ so multiple threads do not change the state of a condition variable at the same time

(b) Write snipppets of signal and wait code, illustruatrating correct use of the condition variable to await a condition.

while (shared variable is not a desired state)
pthread_cond_Wait(&cond, & mutex) // similar to empty or full like in bounded buffer problem
//perform some functionality like put() or get() in bounded buffer problem
pthread_cond_signal(&cond_2) // similar to empty or full like in

(c) What will the operating system do with the mutex, during which system call(s)?

(d) Could we do this for ourselves? If so, how? If not, why not?

We could perform similar behavior like a mutex by keeping a global variable like a flag, which acts like a lock

XC: (a) Heap allocation is much more complex than stack allocation. What key capability do we gain by using heap allocation functions like malloc(3) rather than stack allocation?

The amount of memory a progam needs can be decided at run time. For example, if we are reading from a file, we don't have to know the file's size to store its contents into memory. We can allocate memory as needed

as opposed to stack memory which is allocated at compile time

(b) Heap allocation is much more complex than direct data segment extension and contraction with sbrk(2). Ignoring the higher cost of system calls (vs subroutine calls), what key capability do we gain by using heap allocation functions like malloc(3) rather than sbrk(2)?

dont have to worry about collisions with stack data when expanding

(c) It was briefly mentioned that mmap(2) could be used as an alternative to sbrk(2) to increase the usable data size in a process' virtual address space. What practical benefit/ability might we gain by using mmap(2) rather than sbrk(2) to augment the malloc arena?

Exam Solutions

1. Interface Stability

This was discussed in reading section(s) Interface Stability
This was discussed in lecture section(s) 2C,10F

a. consequences of an incompatible ABI change
   Application programs purchased/obtained by the customer might stop working after the customer upgraded to the new OS version.
b. how those problems would be responded to
   The Independent Software Vendor would have to discover the new interfaces, modify the program to work with the new interfaces, rebuild it, and get the new version out to affected customers. They might also have to distribute different versions of their software to run on different versions of the OS.
c. how clear interface specifications, distinct from implementation, might help

   If the interface specifications were well abstracted from the current implementation this reduces the likelihood that future implementation changes would necessitate incompatible API/ABI changes.

   Additionally having a clear written interface specification would raise the visability of the interface, perhaps making it more obvious to the OS supplier that they were making a change to a committed interface, and that in doing so they were likely to break existing third party applications. If the interfaces were not clearly documented, people would be less likely to consider changes to be important.

2. Resource Convoys

This was discussed in reading section(s) ~A7.3
This was discussed in lecture section(s) 7K

a. A resource convoy is a persistent queue of processes waiting to get access to a popular resource, which eliminates parallelism, increases delays and reduces system throughput.
b. The key to convoy formation is that processess are no longer able to immediately allocate the required resource, but are always forced to block (until the resource is freed by the current owner and other processes in line run). Once this happens, the mean service time can easily exceed the mean inter-request time, and the line becomes permanent. It may be precipitated by a process becoming bocked or preempted while holding the resource.
c. Techniques for reducing contention include:
   - eliminate mutual exclusion by making the resource truly sharable.
   - reduce mutual exclusion by implementing read/write locks.
   - reduce contention by breaking up the one resource into a number of sub-resources.
   - reduce likelihood of conflict by shortening the protected critical section, or using it less often.
   - reduce the likelihood of preemption by moving potentially blocking operations out of the critical section,

3. Causes of blocking/preemption

This was discussed in reading section(s) AD4.4,A7.5-7
This was discussed in lecture section(s) 3F,4C

a. A running process might be preempted if its time slice ends, if its priority drops, or if a higher priority proces becomes runnable.
b. A running process might become blocked if it requests a resource that is not immediately available, or I/O operation. Also, it is (in some sense) blocked when it is swapped out ... since it cannot run until it is swapped back in.

4. Evaluating Mutual Exclusion

This was discussed in reading section(s) AD28
This was discussed in lecture section(s) 7D,E

a. The text identified the key criteria as successful *mutual exclusion*, *fairness* (vs starvation) and *performance* (single processor, multi-procssor). I added to this *progress*, not blocking for an available resource and likelihood of avoiding convoys and deadlocks.
b. Spin locks work (modulo interrupt), and are prone to starvation. They are likely to be quite wasteful if there is contention, but can be very efficient for uncontended use. But they score well on the progress criterion.
c. Interrupt disables are not usable from user mode and are ineffective against multi-processor parallelism. They are relatively expensive operations, but relatively fair.
d. Mutexes guarantee mutual exclusion. Mutexes work to ensure mutual exclusion. They are (with queuing) relatively fair, but there is a race condition where a new locker can get the mutex before the awakened guy at the front of the queue can do so. But this satisfies the progress criterion. The system call, as well as blocking and dispatching are all relatively expensive operations, but blocking is usually much more efficient than spinning.

5. DLLs vs Shared Libraries

This was discussed in reading section(s) Linking & Libs
This was discussed in lecture section(s) 3Y

a. The major capabilities that come with DLLs are
    - the ability to open and load (at run-time) modules that did not exist at link time,
    - deferring loading until the modules are actually called,
    - the ability to perform per-module initialization and shut-down
    - the ability to resolve references from the loaded module back into the main program.
b. Examples of the exploitation of each capability are
    - explicit selection and loading is exploited by browser plug-ins which can be obtained long after the browser
    - deferred binding can significantly improve performance (by reducing work at initial program load time) and make it possible for a program to get the benefits of modules that become

available after the program starts. This can have a significant performance impact if many plug-ins might be used, but actual use is few and seldom.
  - per module initialization could be used to allocate and initialize private data, register instances, and other complex starup (or shut-down). Device drivers, for instance, require both.
  - the ability to make calls back into the containing program is important if it provides rich services for the plug-in. Here, again, device drivers (which makey heavy use of DKI services) are a very good example.

c. The big extra mechanism that DLLs require is a run-time loader. Why? Because they have to be loaded at run time! They also require a linkage editor that is capable of generating Procedure Linkage Table entries ... but this is a much simpler thing.

6. Messages vs shm IPC

This was discussed in reading section(s) mmap(2),send(2),recv(2)
This was discussed in lecture section(s) 7A

a. The primary advantage of shared memory over message IPC is performance.
b. Shared memory IPC allows large amounts of data can be transferred, at memory speed, with ordinary user-mode instructions, without the need to make expensive calls to operating system.
c. The biggest advantage of messages is that they can easily be sent to processes on other machines, whereas shared memory can only be used between processes on a single machine (it can be turned into messages, but doing so sacrifices its performance advantages). This gives us much greater flexibility in how we structure our applications and systems.

   Messages sent through the operating system can have authenticated sender identity, and the OS can ensure the integrity and privacy of the message contents. This is because the messages are bufferred in, and delivered by the OS ... which does not happen with shared memory.

   Also options like synchronous receive and confirmed delivery may be offered with message system calls, but since applications implement their own shared memory IPC, they would have to provide these services themselves.

7. free lists

This was discussed in reading section(s) AD17.2
This was discussed in lecture section(s) 5C,5G

a. In variable-partition allocation we need to know the size, locations, and neighbors of each chunk. In fixed partition allocation, all of these are constants.
b. The free list data structures must be designed to optimize:
   - searching for a piece of desired size.
   - breaking a large piece into smaller pieces.
   - coalescing neighbors back together.
c. we discssued several types of diagnostic information that could be added to free list descriptors and chunks:
   - if we keep allocated memory on a list (as well as free memory) we can audit that list to find

memory that has not yet been freed, and perhaps detect memory leaks.
- address of the allocater (and perhaps time of allocation. This can be recorded at allocation time. If a subsequent audit finds this chunk to be lost, we will know who allocated it (and hence what it was used for).
- we can put pattern-data guard-zones before and after each chunk (at allocation time) and do periodic audits to see that they still contain the correct patterns. This will detect bufer under- or over-run.

8. prod/cons w/sems

This was discussed in reading section(s) AD31.4
This was discussed in lecture section(s) 7I

This application probably calls for two different semaphores:

a. a work semaphore to allow back-end threads to await requests, The front-end would V the work queue whenever a new request was added to it, and the back-end threads would P the work queue to await work.
b. a mutex semaphore to serialize access to the shared queue. All threads (front-end and back-end) would have to P to lock the mutex, and V to release it when adding or removing requests to/from the queue.

The trick is to avoid deadlock (holding one semaphore and then blocking on the other). Nobody holds the mutex while doing a P on the work queue.

```
server:
        P(mutex)
        append to work queue
        V(mutex)
        V(work queue)

worker:
        P(work queue)
        P(mutex)
        take item off queue
        V(mutex)
```

Note that a two-semaphore solution invites deadlock (much like we saw in the semaphore producer/consumer solution we examined in class. I address this by avoiding hold-and-block on the mutex (release the mutex before P'ing the work semaphore).

9. Page Fault process

This was discussed in reading section(s) AD21.3-5
This was discussed in lecture section(s) 6C

a. the trap and low level handling:
    - process reference address that is not yet mapped in

- CPU generates a page fault exception and traps into the OS
- first level handler is selected from an in-memory trap vector
- the PC/PS at time of trap is pushed onto the supervisor mode stack
- first level handler saves registers and forwards to 2nd level handler.

b. software looup, selection, I/O:
   - page fault handler determines that address does indeed refer to a valid, but paged out, page in the process's address space.
   - a free page frame is found, perhaps requiring some other page to be written out
   - I/O request is scheduled to bring in the required page, and we await completion
   - process's page table is adjusted to show location of newly fetched page.

c. return/resmption::
   - back-up the failed instruction
   - return through the first level handler, which will restore the saved registers.
   - return to usermode with a **return from trap** instruction that will restore the saved PC/PS.
   - resumed process will re-attempt the instruction that had page faulted.

10. using Condition Variables

This was discussed in reading section(s) AD30.1
This was discussed in lecture section(s) 7F,7I

a. The mutex prevents us from missing a wake-up because the condition was signaled, after we checked it, but before we went to sleep.
b. Sample signal and wait code is:

```
waiter:
        pthread_mutex_lock(&mutex);
        while (!condition)
                pthread_cond_wait(&cv, &mutex);
        pthread_mutex_unlock(&mutex);

signaler:
        pthread_mutex_lock(&mutex);
        condition = True;
        pthread_cond_signal(&cv);
        pthread_mutex_unlock(&mutex);
```

   Note that the mutex is held whenever the condition is maniuplated or a call is made to either *signal* or *wait*.

c. The OS will release the mutex after blocking the process (in a call to pthread_cond_wait) and reacquire the mutext before returning to the user-mode process.
d. If the waiter released the mutex prior to calling pthread_cond_wait, the signal could be sent before we went to sleep, and we would have missed the wake-up.

XC. memory allocation mechanisms

This was discussed in reading section(s) AD14
This was discussed in lecture section(s) 5B

Note: this was intended to be *hard question* on this exam, requiring more than mere recollection. Only part (a) was answered in class. Parts (b) and (c) require you to to contemplate how the mechanisms might be used.

a. Stack allocated storage is automatically deallocated when the allocating block exits. Heap storage persists after exiting the block, until it is explicitly freed.
b. The *sbrk(2)* system call can extend or shink the data segment but only at its end. We cannot free individually allocated chunks from the middle.
c. Two likely applications are:
   - allocating very large blocks of memory in their own segments ... where the malloc arena adds little value
   - creating multiple malloc arenas (for different clients) each in its own segment).

The first two points were discussed in class. The last point calls for imagination, which is what made this an extra credit problem.