CS 111 Midterm 05/04/2017

Name Nathan Tsai Seat Row D

Student ID # 304575323 Exam # 58 Seat Col 16

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?

non-upwards compatibility means that all programs who have used this OS before the release are not guaranteed to work on this new version. Since its an OS, this could be dangerous, and many problems could occur internally, programs could crash, and adopting new OS without changing code could be catastrophic and render system to be useless.

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

The developer needs to set out specific API/ABI contracts to list and document all compatible versions and their respective time lines. Putting major/minor release numbers can help, so version 3.3 and 3. involve only implementation changes or upwards compatibility, but version 4.0 would need to specify that it could possibly not work with older versions (new parameters, new function names, new ABI hardware requirements)

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

Interface specifications allow a direct line of communication between OS vendor and consumers. They play a role by listing what version is compatible with what system, and tell user that if they want desired output, they need to check that everything on their end is compatible with the version they are trying to use. It can reduce the number of incompatibility issues and reduce the bugs/problems that can arise from releasing a non upwards compatible API/ABI change.

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

A resource contention convoy is when many threads try to access resource, but because it is a critical section and needs to be accessed one by one, a line of waiting threads form; thus, a slow thread or a single thread that halts in critical section (I/O, interrupt) could hold up the line and create a convoy where all other waiting threads remain blocked, making performance bad

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

Likely to form if critical section is long (malloc, long system calls), or if the resource is updated a lot (increases traffic). Also, if the resource is locked with one giant lock (coarse-grained locking), then threads will not be able to access resource at same time, guaranteeing that only one thread can pass at a time, creating a resource contention convoy.

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

1) Make critical section shorter, If we move memory allocation calls out of critical section and minimize code in critical section, threads can pass through much faster, so resource contention is reduced, and the chance of a thread holding up the line and creating a convoy is reduced.

2) Fine-grained locking: by having many small locks lock parts of a resource, then many threads can access different parts of the resource at the same time, reducing resource contention and reducing chance of a convoy forming. This improves performance because of higher [illegible]

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

1) If OS is using the Shortest Job To Completion scheduling policy, a process that still has much to do before completing could be preempted to give a process that joins later but completes faster a chance to run

2) In Round-Robin scheduling, a process will be preempted once its fixed running time is up, giving another process a chance to run.

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

1) A process becomes blocked whenever user process issues I/O, so OS blocks it, takes care of what needs to be done, and then resumes process

2) A process calls a system call to block running process, transfer control to OS to execute privileged instructions, and then resumes once system call finishes

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

Mutual Exclusion (correctness): does mechanism successfully implement locking such that only one thread can access critical sections in all situations?

Performance: does overhead of locking/unlocking not dominate the implementation? How fast does the mechanism do the actual mutual exclusion and locking/unlocking?

Fairness: Do waiting threads have an equal chance to grab lock, and is starvation of a thread successfully avoided in all cases?

(b) Evaluate spin-locks against these criteria.

1) Spin-lock on one CPU
- mutual exclusion: it correctly implements mutual exclusion
  - performance: Very Bad. If a thread locks then preempts, all other threads trying to grab lock will spin for entire cycle, unnecessarily wasting CPU cycles
  - Fairness: not fair, no guarantee that a waiting thread won't starve

2) Spin-lock on multiple CPUs
- mutual exclusion: correctly implements mutual exclusion
- performance: is not too bad as long as critical sections are short
- Fairness: again, not fair, no guarantee a waiting thread won't starve and wait forever

(c) Evaluate interrupt disables against these criteria.

1) Interrupt disabling on 1 CPU
- Mutual exclusion: It is correct on 1 CPU, as it will ensure no simultaneous access to critical section
- Performance: it could be expensive because by nature disabling interrupts is an expensive operation.
- Fairness: not fair, threads not guaranteed to avoid starvation completely

2) Interrupt disabling on multiple CPUs
- Mutual Exclusion: Fails this, disabling interrupts won't stop another thread from entering critical section
- Same as 1 CPU case for Performance and Fairness

(d) Evaluate mutexes against these criteria.

1 CPU and multiple CPU case

- mutual exclusion: correct! locking/unlocking on 1 CPU/more CPUs is correct because mutexes implemented with atomic hardware instructions
- Performance: depends. If program locks/unlocks very often, then performance overhead could happen, and slow program down
- Fairness: Fails, no guarantee that a waiting thread won't starve, need semaphores and queue-based waiting system to ensure there is some ordering to which thread gets lock next

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

DLL's can have implementations that may not have existed when the program was created, so DLL is powerful in that you can implement libraries/functionalities on the fly during run time. Can achieve much more flexibility in this sense.

TBD calls to
TBD implementations

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

Let's say I have a program that runs on my self-driving car, I know flying cars will eventually come about in 3 years, but I want program to constantly run on this cars OS. If it stops running stuff will break, like braking system. But I want to be able to support flying car operations when it comes. With DLL, I can dynamically load new software when it exists and use it to fly my car.

(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.

Shared libraries allow for the performance boost of referencing a single physical location, but it won't allow the the calls to global references or other libraries.

DLL's can make calls to other libraries and user process, so if you want mechanism of callbacks and complex bi-direction communication, you need to use DLL's over shared libraries

Necessary to have communication or wait for asynchronous results.

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

Advantage is speed and performance; if communication happens on a shared file, then both process can map same code to their virtual address spaces.

(b) Why does it have this advantage?

It must be fast because there are applications where speed of communication is necessary for survival or crucial operations.

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

—message IPC (like mailboxes) achieve:

1) once a reader ceases to exist, more readers can still read unread messages that sit in mailbox
   —shared memory can't do this, once a reader/process dies, communication dies with it
2) the sender of message can be authenticated, so reader knows who sent what messages
   —shared memory doesn't know where messages came from, only knows that here are byte-streams and it can only process it. Malicious messages will be [illegible] credibility of source

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.

- needs to know size of free chunk
  - not needed in fixed partition cuz size is uniform
- needs to know where memory is relative to each other because we want to coalesce, with fixed memory allocation, no need to coalesce, so don't need to know if two chunks are contiguous

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

- needs to deal with external fragmentation and coalescing to make sure that no chunks of memory become useless because too small
  - It can use algorithms like Best Fit, Next Fit, Worst Fit, and First Fit
- needs to optimize how to allocate memory to reduce fragmentation, so may need to relocate memory to separate free/unfree chunks.

(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.

- a magic number provided by hardware will help us identify the free chunk and ensure that it is indeed what we freed, and no malicious program can pretend an address is a free chunk when it isn't. It prevents the accident of OS thinking a certain block is free or if it isn't, and having this extra unique identifier only known by OS helps prevent this problem.

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.

wait
post

```
//server algorithm
server(){
  queue<request> reqQueue;
  requests[NUM_REQUESTS];
  getRequests(requests);
  while(1) {
    for(int i=0; i<NUM_REQUESTS; i++){
      reqQueue.push(requests[i]);
      for (int i=0; i<12; i++)
        pthred_create(&dOWork, requests[i]);
    }
  }
}
```

```
// thread algorithm
void* doWork(request MyRequest) {
  // do all work that involves read
  // or private computations
  read(MyRequest);
  privateComputations();
  //critical section when doing updates
  semaphore_wait(&count); //decrement counter
  updateDataStructure();   //critical section
  semaphore_post(&count); //increment counter
  finishReading(MyRequest);
  return;
}
```

```
// request data structure
struct request { int info1; int info2; }
// All functions below are listed above
int main {
  sem_s count; //represents count of semaphore
  semaphore(&count, 1); //initialize semaphore to 1
  server(); //queves requests, makes 12 threads per reqrest
}
```

```
// wrapper that implements semaphores
semaphore(count, init) {
  //initialize value of
  // semaphore counter to init
}
```

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.

Demand Paging

Requirements: TLB (hardware cache) lives in MMU (Memory Management Unit) of CPU
Memory (Physical memory)
Page-Table (per process structure with address translations)
Disk
Page-Table Entry (VPN | PFN | bits like valid bit, present bit, dirty bit)

STEPS: In fault handling, we check if the present bit is high in the page table entry, If it isn't, then raise exception and proceed to swap page from disk into memory, First need to save process state into OS data structure
In CISC machines, hardware takes care of all page fault handling
In RISC machines, hardware raises exception, and OS does the rest
Once it is swapped to memory, change present bit to be high, and retry instruction

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

Continuing from page fault handling:
Software lookup:
IF page is now in memory and is present, the instruction is retried. OS looks in TLB cache first, If it is a miss, then we look in page table entry to see if present, Since it is present, because we handled page fault handling and swapped into memory, we now swap page existing in memory to TLB. Now, we retry instruction. The OS looks in TLB, and it is a hit, so now we can successfully translate the virtual address into the physical address.

Selection: need policies to choose which pagees to evict from TLB or which to evict from memory to swap in desired page. Use working sets algorithm or LRU/global LRU depending on needs to do selection.

I/O: Once page fault happens, I/O issued, which blocks calling process to handle the page fault

updates: mechanism to actually update/swap pages in done by low-level hardware and managed by OS. Details listed above.

(c) describe the return/resumption process.

Now that the page that we wanted is successfully in TLB hardware cache, we know that access to address will be fast. To give process running power again, we need to restore process state, such as PC, PS, stack pointer, stack frame, and registers, PC (Program Counter) holds next executing instruction that was saved in OS data structures, Update PS (Processor Status) to represent the fact that we are in user mode. Restore registers and stack pointer such that process state is completely restored. Meaning, process now resumes normal execution, starting right where it left off. That's the beauty of virtualization: all of this is transparent to process, so process just continues without knowing OS did all this for it. All future calls to memory [illegible]

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

Condition variables need to be paired with a lock (mutex) to avoid race conditions. We want to avoid spurious wake ups (one signal call wakes up 2 waiting threads) If lock is implied, that means hopefully only one thread wakes up from another thread's signal call. Race condition of spurious wake up can be bad, threads that wake up assume state is the same from waking up to point of execution, but if 2 threads wake, 1 thread will find unexpected results.

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

```
cond_t empty, full;
mutex_t lock;

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

producer (int count) {
  pthread_mutex_lock(&lock);
  int i=0;
  while (full)
    pthread_cond_wait(&empty, &lock);
  while (!full && i<count) {
    put();
    i++;
  }
  pthread_cond_signal(&full);
  if (i==count) {
    pthread_mutex_unlock(&lock);
    return;
}}}
```

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

The OS will atomically compare and test mutex condition to ensure that the mutex is held by only one process.

System calls:

test-and-set();
compare-and-put(); } ATOMIC SYSTEM CALLS that check and update mutex value

By returning old value of pointer while simultaneously testing new value with expected value, we can ensure that even if process pre-empted, no race conditions occur. See part (d) for race condition that the above atomic system calls prevent!

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

We can't because high level languages like C are made up of multiple machine/assembly instructions.

counter = counter + 1 becomes

```
movq 0x1234, %eax
addq 0x1, %eax
movq %eax, 0x1234
```

if multiple threads run concurrently, race condition can happen. if thread 1 executes to addq but pre-empts, and thread 2 executes all 3 instructions then thread 1 finishes execution, our counter will only increment by 1 even though it was supposed to increment by 2. Not atomic instructions

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?

we are allowed to set a user-determined size when we use malloc, so heap gives user more flexibility over size compared to stack.

Also, heap lives longer, meant for longer-lasting data, while stack is short lived and data gets popped off frequently.

(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)?

Using heap allows us re-allocate when we need to, and gives user flexibility to allocate. Also, to deallocate, user can call free to re-use that space.

Sbrk(2) just directly allocates memory by extending the memory size we get. However, we can't recycle the memory easily, and there's no equivalent to the free call that recycles memory usage.

(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?

mmap maps to possibly a different location