CS439H: Fall 2011 – Midterm 1

Instructions

- Stop writing when"time" is announced at the end of the exam. I will leave the room as soon as I've given people a fair chance to bring me the exams. I will not accept exams once my foot crosses the threshold.
- This midterm is closed book and notes.
- If a question is unclear, write down the point you find ambiguous, make a reasonable interpretation, write down that interpretation, and proceed.
- For full credit, show your work and explain your reasoning and any important assumptions.

  Write brief, precise, and legible answers. Rambling brain-dumps are unlikely to be effective. **Think before you start writing** so that you can crisply describe a simple approach rather than muddle your way through a complex description that "works around" each issue as you come to it. **Perhaps jot down an outline** to organize your thoughts. And remember, a **picture can be worth 1000 words.**
- **Write your name and uteid on every page of this exam.**

1. (8) Suppose you had a choice between two file systems. System 1's vendor guarantees it will be 99% reliable and 100% available. System 2's vendor guarantees it will be 100% reliable and 99% available. Which would you pick?

   Solution: System 2. An unreliable file system can permanently lose data.

2. (16) Virtualization uses a hypervisor running in privileged mode to create a virtual machine that runs in unprivileged mode. Then, unmodified guest operating systems can run in the virtual machine. The hypervisor can provide the illusion that each guest operating system is running on its own machine in privileged mode.

   Early versions of the x86 architecture (pre-2006) were not *completely virtualizable* – these system could not guarantee to run unmodified guest operating systems properly. One problem was the `popf` "pop flags" instruction. When `popf` was run in privileged mode, it could change both the ALU flags (e.g., `ZF`) and the systems flags (e.g., `IF`, which controls interrupt delivery), and when `popf` was run in unprivileged mode, it could change just the ALU flags.

   Why do instructions like `popf` prevent transparent virtualization of the (old) x86 architecture?

   Solution: Since this instruction behaves differently for user and kernel mode and since it does not cause a trap in user mode, the guest kernel will expect it to do one thing (privileged mode version) but it will actually do something else.

   How would you change the x86 hardware to fix this problem?

   Solution: A simple thing to do is to add a privileged flag that, when set, causes popf in user mode to trap. Then the hypervisor ensures that whenever the guest is running in virtual privileged mode, the popf-trap flag is set; it can be cleared when the guest is running in virtual unprivileged mode.

3. (4) List the four necessary conditions for deadlock.

   Solution: limited resources, wait while holding, circular waiting, no preemption

4. (16) For each of the data structures listed in the table below, indicate (by checking the box in the appropriate column) whether the memory identified is stored in per-thread areas of memory (i.e., it refers to per-thread state) or if it is stored in areas of memory that can be shared by many threads (i.e., it refers to potentially shared state).

```
int max = 42;
char message[] = "Hello world";

int
main(int argc, char **argv)
{
   char *msg = message;

   sthread_t *t = (sthread_t *)malloc(sizeof(sthread_t));
   sthread_init(t, go, msg);
   t = (sthread_t *)malloc(sizeof(sthread_t));
   sthread_init(t, foo, NULL);
}

void
go(char *toPrint)
{
   int OK = 1;
   static int done = 0;
   if(strlen(toPrint) > max){
     OK = 0;
   }
   done = 1;
}

void
foo(void *notUsed){
   // Code omitted
   ...
}
```

	Private, per-thread state	Shared/sharable state
heap	☐	☐
stack	☐	☐
message	☐	☐
msg	☐	☐
toPrint	☐	☐
argc	☐	☐
OK	☐	☐
max	☐	☐
t	☐	☐
go	☐	☐
done	☐	☐

Solution: shared: heap, message, max, go, done; others are private

5. (8) Consider a virtual memory system with 42-bit physical addresses and 6 control bits per page. How large does the page size have to be to allow each page table entry to fit in a 4-byte word?

   Solution: 42 + 6 - 32 = 16. So each page needs to be 2^{16} bytes (64KB).

6. (8) Consider a virtual memory system with 48-bit virtual addresses, 44-bit physical addresses, 16KB pages, and 7 control bits per word. Assuming a multi-level page table arrangement, how many levels of page tables should this system use?

   Solution: entry size = 44 + 7 - 14 = 37 round up to 8 bytes
   entries per page = 2K
   bits per level = 11
   number vpage bits = 48 - 14 = 34
   number of levels needed = 4 (the top level has just 2 entries, though)

7. (8) The Bryant and O'Halloran book says that it is safe to do conservative mark and sweep garbage collection in a C program. This is not quite true. Write a short 'C' program (detailed pseudocode is fine) that can dereference a pointer that could point to garbage collected memory after a mark and sweep pass.

   The key thing here is to do some pointer arithmatic such that allocated memory has no pointer pointing into it, but you can construct such a pointer.

```
char *hidden = malloc(100 * sizeof(char));
sprintf(hidden, "Hidden.");
hidden = hidden/2;
...
// mark and sweep happens
...
hidden = hidden * 2;
printf(hidden);
```

8. (32) Implement a *priority condition variable.* A priority condition variable (PCV) has 3 public methods:

```
void PCV::wait(Lock *lock, int priority);
void PCV::signal(Lock *lock);
void PCV::broadcast(Lock *lock, int priority);
```

These methods are similar to those of a standard condition variable. The one difference is that a PCV enforces both *priority* and *ordering*.

In particular, `signal(Lock *lock)` causes the currently waiting thread with the highest priority to return from `wait()`; if multiple threads with the same priority are waiting, then the one that is waiting the longest should return before any that have been waiting a shorter amount of time.

Similarly, `broadcast(Lock *lock, int priority)` causes all currently waiting threads whose priority equals or exceeds `priority` to return from `wait()`.

For full credit, you must follow the *thread coding standards* discussed in class.

Solution: This problem ended up being tougher than intended. The intention was for this to be a standard "implement a shared object" problem – a PCV is just another type of shared object and the standard techniques apply. A large number of people interpreted this as a "implement from atomic operations" problem or did a hybrid atomic operations/locking-based solution. I can see why that might appear to be a reasonable interpretation, so I graded those accordingly.
Also, in retrospect, I should have given the `Lock *lock` parameter a more descriptive name to make it more clear what was going on. (Some people used that lock for mutex in this object; that does give the ownership pattern, so I gave credit.

class PCV Member variables

type	name	initial value (if any)

Solution:

type	name	initial value (if any)
Lock	myLock	NA
CV	cv	NA
SortedList	waiting	empty

PCV::wait(Lock *lock, int priority){

```
// Wait should atomically release the callerLock
// and start waiting until signalled. And it should
// reacquire the lock before returning.
//
// NOTE: Since we (a) want to follow the Standards and
// (b) don't want to risk deadlock, we break wait()
// into two pieces
  doWait(callerLock, priority);
  lock->acquire();
}

void
PCV::doWait(Lock *lock, int priority)\{
  myLock.acquire();
  lock->release();
  WaitRecord *wr = new WaitRecord(priority);
  waiting.insertSortedByPriorityAndInsertOrder(wr);
  while(!wr.okToGo){
    cv.wait(&myLock);
  }
  free(wr);
  myLock.release();

}
```

PCV::signal(Lock *lock){

```
  myLock.acquire();
  waiting.markFirstOkToGo();
  cv.broadcast(&myLock);
  myLock.release();

}
```

PCV::broadcast(Lock *lock, int priority){

```
  myLock.acquire();
  waiting.markPriorityOkToGo(priority);
  cv.broadcast(&myLock);
  myLock.release();
```

}

Note:

WaitRecord has 4 fields: priority (initialized) and okToGo (initially false), and next/prev

Here are the helper functions (it was OK to include much less detail than this.)

```
SortedList::insertSortedByPriorityAndInsertOrder(WaitRecord *e)
{
   WaitRecord *cur = head;
   while(cur && cur->next && cur->next >= e->priority){
        cur = cur->next;
   }
   e->next = cur->next;
   e->prev = cur;
   if(cur->next){
```

```
      e->next->prev = e;
    }
    if(cur != head){
      e->prev->next = e;
    }
    else{
      head = e;
    }
}

// Note: also need to remove first from
// list to avoid remarking it
SortedList::markFirstOKToGo()
{
    WaitRecord *cur = head;
    if(cur){
      cur->OKToGo = true;
      head = cur->next;
      if(head){
        head->prev = NULL;
      }
    }
    return;
}

SortedList::markFirstOKToGo(int pri)
{
  while(head && head->priority >= pri){
       markFirstOKToGo();
    }
    return;
}
```

This page intentionally left almost blank

This page intentionally left almost blank