University of Waterloo
Midterm Examination Model Solution

Winter, 2009

1. (12 total marks)

Consider a virtual memory system that uses paging. Virtual and physical addresses are both 32 bits long, and the page size is 4KB = 2^{12} bytes. A process P_1 has the following page table. Page and frame numbers are given in hexadecimal notation (recall that each hexadecimal digit represents 4 bits).

Page Number	Frame Number
0x00000	0x00234
0x00001	0x00235
0x00002	0x0023f
0x00003	0x00ace
0x00004	0x00bcd

a. (3 marks)

For each of the following virtual addresses, indicate the physical address to which it maps. If the virtual address is not part of the address space of P_1, write `NO TRANSLATION` instead. Use hexadecimal notation for physical addresses.

- `0x00001a60` $\rightarrow$ `0x00235a60`

- `0x002351ff` $\rightarrow$ `NO TRANSLATION`

- `0x00000fb5` $\rightarrow$ `0x00234fb5`

b. (3 marks)

If the page size was 16KB = 2^{14} bytes, how many bits would be needed for the page number, frame number, and offset?

- Page number: 18

- Frame number: 18

- Offset: 14

c. (2 marks)

Why does the page size have to be a power of 2?

In address translation under paging, the rightmost k bits are used as the offset. To get the page (or frame) number we can look at the bits to the left of the offset, without needing division. This works only if the page size is a power of 2.

d. (4 marks)

The page size is one of the key parameters of a virtual memory system. List one advantage and one disadvantage of larger page sizes.

- Advantage:

One of:

  - Smaller page table
  - Bigger TLB footprint
  - Fewer TLB misses
  - Wasted work paging in data that is not needed

- Disadvantage:

One of:

  - More internal fragmentation
  - Run out of physical memory sooner, so page faults more likley

2. (10 marks)

a. (2 marks)

Why does a thread need two stacks?

A thread needs one stack for executing in user mode and one stack for executing in the kernel.

b. (2 marks)

What is the difference between the ready and blocked thread states?

A ready thread will be considered by the scheduler for dispatching. A blocked thread will not be considered for scheduling.

c. (2 marks)

Why would a thread be in the blocked state?

A thread would be blocked if it is waiting for an event to happen (at which point another thread would unblock it).

d. (2 marks)

What is the role of the timer interrupt in round-robin thread scheduling?

The timer interrupt ensures that the kernel periodically gains execution so that it can preempt a thread when its quantum expires.

e. (2 marks)

The MIPS register usage convention specifies that registers `t0` to `t9` are temporary registers that are not preserved by subroutines. In OS/161, `thread_yield()` does not save these temporary registers as part of the thread context, while the exception handler does save them as part of the thread context. Why does the exception handler need to save these registers?

When a thread calls `thread_yield()`, it does not expect the values of `t0` to `t9` to be preserved when the function returns. An exception, on the other hand, can happen at any time, and the thread may be using t0 to t9 when the exception happens. The exception handler must therefore save these registers so that the thread can continue execution after the return from exception without seeing the values of these registers change unexpectedly.

3. (10 marks)

a. (5 marks)

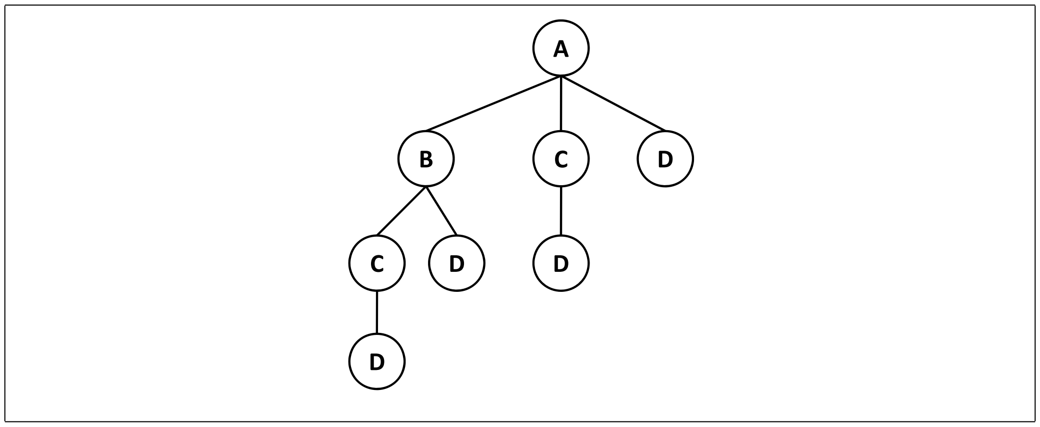
Consider the following list of actions. Put a check mark in the blank beside those actions that should be performed only by the kernel (i.e., in privileged mode), and not by user programs.

- _**X**__ Creating a shared memory segment.
- ____ Issuing a `syscall` instruction.
- ____ Changing the value of the `sp` register (the stack pointer).
- ____ Changing the value of the program counter.
- ____ Using a test-and-set instruction for synchronization.

b. (5 marks)

Consider a system with a process, `ProcA`, that executes the following program. Draw the tree rooted at `ProcA` representing the processes created by this program. Every node in the tree should represent a process, and there should be an edge from node `A` to node `B` if process `A` creates process `B`. Label each node of the tree with the label of the `fork()` statement that created it.

```
int main () {
  fork();   /* Label: B */
  fork();   /* Label: C */
  fork();   /* Label: D */
  return 0;
}
```



4. (6 total marks)

a. (2 marks)

If `ProgramA` has a bigger ELF file than `ProgramB`, does this mean that `ProgramA` will have a bigger virtual address space than `ProgramB`? Briefly explain your answer.

No. `ProgramB` may have more uninitialized global data than `ProgramA`, and so have a larger virtual address space. Data in the uninitialized global data sections (.bss and .sbss) does not take up space in the ELF file, but space is created for it in the virtual address space.

b. (2 marks)

Does an ELF file contain information about the stack to be created for the program? If yes, what is this information? If no, why not?

No. The stack is created by the kernel for use during run time. The kernel does not need information from the ELF file about where the stack is located or how it is initialized.

c. (2 marks)

Does an ELF file contain information about the physical addresses at which the different segments are loaded? If yes, what is this information? If no, how are these physical addresses chosen?

No. The kernel manages physical memory, and it chooses the physical addresses at which each segment is loaded.

5. (10 total marks)

a. (4 marks)

Multiple threads simultaneously access `N` data items. To ensure mutual exclusion in accessing these items, we define an array of locks as follows:
`struct lock L[N];`
When a thread accesses item `i`, it first acquire's the lock `L[i]`. The thread releases this lock after it is done using item `i`. A thread may need to use multiple items (and hence acquire multiple locks) at the same time.

Is it possible that a deadlock will occur among these threads? Answer yes or no. If deadlock is possible, give a concrete example of a sequence of events that would result in a deadlock. Be sure to indicate which thread performs each event in your example. If deadlock is not possible, indicate which of the deadlock prevention techniques (if any) discussed in class is being used.

Yes, deadlock can occur:
Thread 1: lock_acquire(L[1]);
Thread 2: lock_acquire(L[2]);
Thread 1: lock_acquire(L[2]);
Thread 2: lock_acquire(L[1]);

b. (3 marks)

In a concurrent program with multiple threads, one thread repeatedly calls `ProcedureA`, shown below. Another thread repeatedly calls `ProcedureB`. The procedures use two locks, `foo_lock` and `bar_lock`, and a condition variable, `CV1`, all of which are shared. What is wrong with the way that the procedures are using these synchronization primitives?

```
ProcedureA() {
   lock_acquire(foo_lock);
   // Access foo data structure
   ...
   cv_signal(CV1, foo_lock);
   lock_release(foo_lock);
}
```
	```
	ProcedureB() {
	   lock_acquire(bar_lock);
	   // Access bar data structure
	   ...
	   cv_signal(CV1, bar_lock);
	   lock_release(bar_lock);
	}
	```

The two threads are using the same condition variable with two different locks.

c. (3 marks)

Describe how a `Swap(A,B)` instruction can be used to enforce mutual exclusion. Fill in your solution below.

- Shared variables:

```
int lock = 0;
```

- Code for entering critical section (executed by each thread):

```
int other_in_cs = 1;

while(other_in_cs == 1) {
   Swap(lock, other_in_cs);
}
```

- Code for exiting critical section:

```
lock = 0;
```

6. (12 total marks)

Consider a concurrent program with three threads. Thread 1 repeatedly calls `ProcedureA`, shown on the next page, to produce items of type `A` and place them in an unbounded list `ABuff`. Thread 2 repeatedly calls `ProcedureB`, to produce items of type `B` and place them in an unbounded list `BBuff`. Thread 3 repeatedly calls `ProcedureC`, which consumes one item of type `A` or `B`, depending on what items are available. If an item of type `A` is available, `ProcedureC` consumes it. If no item of type `A` is available but an item of type `B` is available, `ProcedureC` consumes the `B` item. If no `A` or `B` items are available, `ProcedureC` will block until some item is available.

Code for `ProcedureA`, `ProcedureB`, and `ProcedureC` is given on the next page, but without synchronization. Add synchronization primitives to this code as needed to ensure that: (1) `ABuff` and `BBuff` are accessed by only one thread at a time, and (2) `ProcedureC` blocks if no items are available and wakes up when items are available.

Answer Question 6 in the template below.

List Lock, Condition Variable and Global Variable Declarations Here

```
list *ABuff, *BBuff;
```

```
ProcedureA() {

   item *A;



   produce(A);


   list_add(ABuff, A);



}
```

```
ProcedureB() {

   item *B;



   produce(B);


   list_add(BBuff, B);



}
```

```
ProcedureC() {

   item *C;

   // Add code here to check whether to (i) remove an item from ABuff by
   // calling list_remove(ABuff, C), (ii) remove an item from BBuff by calling
   // list_remove(BBuff, C), or (iii) block.



   consume(C);


}
```

Solution on next page.

List Lock, Condition Variable and Global Variable Declarations Here

```
list *ABuff, *BBuff;
int NumA = 0; int NumB = 0;
lock *ALock, *Block;
semaphore *ItemAvailable; // Initially 0
```

```
ProcedureA() {

   item *A;

   produce(A);

   lock_acquire(ALock);
   list_add(ABuff, A);
   NumA++;
   lock_release(ALock);

   V(ItemAvailable);
}
```

```
ProcedureB() {

   item *B;

   produce(B);

   lock_acquire(BLock);
   list_add(BBuff, B);
   NumB++;
   lock_release(BLock);

   V(ItemAvailable);
}
```

```
ProcedureC() {

   item *C;

   // Add code here to check whether to (i) remove an item from ABuff by
   // calling list_remove(ABuff, C), (ii) remove an item from BBuff by calling
   // list_remove(BBuff, C), or (iii) block.

   P(ItemAvailable);

   lock_acquire(ALock);

   if(NumA>0){
      list_remove(ABuff, C);
      NumA--;
      lock_release(ALock);
   }else{
      lock_release(ALock);

      assert(NumB>0);

      lock_acquire(BLock);
      list_remove(BBuff, C);
      NumB--;
      lock_release(BLock);
   }

   consume(C);
}
```