University of Waterloo
Midterm Examination Model Solution
Winter, 2008

1. (10 marks)

Consider a system with two processes, P_1 and P_2. Process P_1 is running and process P_2 is ready to run. The operating system uses preemptive round robin scheduling. Consider the situation in which P_1's scheduling quantum is about to expire, and P_2 will be selected to run next.

List the sequence of events that will occur in this situation, in the order in which they will occur. Create your list by choosing from the events shown below, using the event numbers to identify events. For example, a valid answer might be: "7,5,4,8,5,2". If an event occurs more than once, you may include it more than once in your list.

1. P_1's user-level state is saved into a trap frame
2. P_2's user-level state is saved into a trap frame
3. a timer interrupt occurs
4. the operating system's scheduler runs
5. there is a context switch from thread T_1 (from process P_1) to thread T_2 (from process P_2).
6. P_1's user-level state is restored from a trap frame
7. P_2's user-level state is restored from a trap frame
8. a "system call" instruction is executed
9. thread T_2 (for process P_2) is created
10. thread T_1 (from process P_1 is destroyed

Write your answer here:

3,1,4,5,7

2. (4 marks)

Suppose that an operating system uses preemptive round-robin scheduling with a scheduling quantum of 500 milliseconds. The hardware clock generates timer interrupts once every millisecond. Suppose that a process P is scheduled to run and is dispatched. During its quantum, P makes no system calls. However, address translation exceptions due to TLB misses occur once every 10 milliseconds while P is running.

How many times during its quantum does process P's thread enter the kernel? Briefly explain your answer.

Each exception and interrupt causes control to enter the kernel. There will be 500 timer interrupts during P's quantum, and there will be $500/10 = 50$ address translation exceptions, for a total of 550 kernel entries.

3. (12 marks)

Consider a concurrent system with producer and consumer threads that share access to a buffer with a capacity of `N` items. Producer threads repeatedly call the `Produce` function shown in pseudocode below. Consumer threads repeatedly call the `Consume2` function. `Produce` and `Consume2` make use of a shared variable `NumInBuffer`, which tracks the number of items in the buffer. The buffer is initially empty, and `NumInBuffer` is initially 0. The two functions also make use of an OS/161-style lock, called `mutex`, and two OS/161-style condition variables, called `NotEmpty` and `NotFull`.

```
Produce(item x) {
  lock_acquire(mutex);
  while (NumInBuffer == N) {
    cv_wait(NotFull,mutex);
  }
  put x into the buffer
  NumInBuffer += 1;
  if (NumInBuffer >= 2) {
    cv_signal(NotEmpty,mutex)
  }
  lock_release(mutex);
}
```

```
Consume2() {
  lock_acquire(mutex);
  while (NumInBuffer < 2) {
    cv_wait(NotEmpty,mutex);
  }
  remove two items from the buffer;
  NumInBuffer -= 2;
  cv_broadcast(NotFull,mutex);
  lock_release(mutex);
}
```

Re-implement `Produce` and `Consume2` using OS/161-style semaphores instead of locks and condition variables. Your implementation should provide the same synchronization that is provided by the implementation above, but all synchronization should be accomplished with semaphores.

Show a declaration of every semaphore you use. The declaration should specify the initial value of the semaphore. You may make use of the shared variable `NumInBuffer` in your solution, but you may not introduce any new variables. Write your solution in pseudocode similar to the pseudocode above. Keep your solution as simple as possible - overly complex solutions may be penalized.

Place Semaphore Declarations Here

```
(binary) semaphore Mutex = 1; semaphore Freespace = N; semaphore ItemCount = 0;
(binary) semaphore ClaimTwo = 1;
```

```
Produce(item x) {

  P(Freespace)
  P(Mutex)
  put x into the buffer
  V(ItemCount)
  V(Mutex)

}
```

```
Consume2() {

 P(ClaimTwo)
 P(ItemCount)
 P(ItemCount)
 V(ClaimTwo)
 P(Mutex)
 remove 2 items from the buffer
 V(Freespace)
 V(Freespace)
 V(Mutex)
}
```

4. (15 total marks)

For all parts of this question, you should assume a virtual memory system based on simple paging. All parts of this question refer to the following two page tables, one for process P_1 and one for process P_2. Note that the frame numbers are specified in hexadecimal, as are all virtual and physical addresses used in this question.

P_1's page table

Page Number	Frame Number
0	0x8d10
1	0x1004
2	0x004a
3	0x5500
4	0x2220
5	0x2221
6	0x2222
7	0x222a
8	0x5558

P_2's page table

Page Number	Frame Number
0	0x222b
1	0x010a
2	0x010b
3	0x3008
4	0x3001
5	0x222c

a. (5 marks)

For each of the following virtual addresses from P_1's virtual address space, indicate the physical address to which it corresponds. For the purpose of this part of the question, assume that the page size is 256 (2^8) bytes. Give your answers in hexadecimal. If the specified virtual address is not part of the virtual address space of P_1, write "NO TRANSLATION" instead.

- `0x00003a8` → `0x5500a8`
- `0x0001004` → `NO TRANSLATION`
- `0x0000022` → `0x8d1022`
- `0x0006072` → `NO TRANSLATION`
- `0x00005ff` → `0x2221ff`

b. (5 marks)

Repeat part(a), but this time under the assumption that the page size is 4096 (2^{12}) bytes.

- `0x00003a8` → `0x8d103a8`
- `0x0001004` → `0x1004004`
- `0x0000022` → `0x8d10022`
- `0x0006072` → `0x2222072`
- `0x00005ff` → `0x8d105ff`

c. (5 marks)

For each of the following *physical* addresses, indicate which process's virtual address space maps to that physical address, and indicate which specific virtual address maps there. If neither process's virtual address space maps to the given physical address, write "NO MAPPING" instead. For the purposes of this question, assume that the page size is 4096 (2^{12}) bytes.

- `0x3008888` → `0x3888`, process P_2
- `0x222cc01` → `0x5c01`, process P_2
- `0x222d002` → `NO MAPPING`
- `0x2222ffa` → `0x6ffa`, process P_1
- `0x010abcd` → `0x1bcd`, process P_2

5. (11 total marks)

Consider a system with two preemptively scheduled threads. One thread executes the `WriteA` function shown below. The other executes the `WriteB` function, also shown below. Both functions use `kprintf` to produce console output. The `random` function called by `WriteA` returns a randomly-generated non-negative integer. `WriteA` and `WriteB` are synchronized using two semaphores, S_a and S_b. The intial value of both semaphores is zero. Assume that the individual calls to `kprintf` are atomic.

```
WriteA() {
  unsigned int n,i;
  while(1) {
    n = random();
    for(i=0;i<n;i++) {
      kprintf(``A'');
    }
    for(i=0;i<n;i++) {
      V(S_b);
    }
    for(i=0;i<n;i++) {
      P(S_a);
    }
}
```

```
WriteB() {
  while(1) {
    P(S_b);
    kprintf(``B'');
    V(S_a);
  }
}
```

a. (8 marks)

Consider the following 10-character console output prefixes. (Each prefix shows the first 10 characters printed to the console.) Which of these prefixes could possibly be generated by the two threads running in this system? Write "YES" next to the output prefix if it could be generated, otherwise write "NO".

- ABABABABAB **YES**
- BABABABABA **NO**
- AAAAAAAAAA **YES**
- AAABABABAB **NO**
- AAABBBBAAB **NO**
- AAAAAABBBB **YES**
- AAABBBAABB **YES**
- BBBBBAAAAB **NO**

b. (3 marks)

Suppose that the initial value of semaphore S_b is 1, rather than 0. Show a 10-character console output prefix that could be generated in that case, and that could not be generated if the initial value of S_b were 0.

Some examples:

- BABABABABA
- ABABBABABAB
- AAABBBABABB

6. (18 total marks)

a. (4 marks)

Consider the OS/161 address space structure, which is shown below. This structure contains information about the three virtual address space segments in an OS/161 process: the text segment (code and read-only data), the data segment, and the stack segment.

```
struct addrspace
  /* text segment */
  vaddr_t as_vbase1; /* virtual base address */  ELF
  paddr_t as_pbase1; /* physical base address */  KERNEL
  size_t as_npages1; /* number of pages */  ELF

  /* data segment */
  vaddr_t as_vbase2; /* virtual base address */  ELF
  paddr_t as_pbase2; /* physical base address */  KERNEL
  size_t as_npages2; /* number of pages */  ELF

  /* stack segment */
  paddr_t as_stackpbase; /* physical base address */  KERNEL
;
```

Indicate whether the value of each field in this structure is determined by information in the ELF file, is chosen by the kernel, or neither. Give your answer by writing "ELF", "KERNEL" or "NEITHER" next to each field in the structure.

b. (2 marks)

Peterson's algorithm can be used to enforce mutual exclusion without the need for special synchronization instructions such as test-and-set or compare-and-swap. What is the most significant limitation of Peterson's algorithm?

Peterson's algorithm can only be used to synchronize two threads.

c. (2 marks)

Some possible thread execution states are *running*, *ready* and *blocked*. In OS/161, which thread library function is used to move a thread from the *running* state to the *blocked* state?

`thread_sleep()`

d. (2 marks)

What is the difference between a *ready* thread and a *blocked* thread? Explain briefly and clearly.

A ready thread is able to run as soon as it is selected by the scheduler. A blocked thread is waiting for some event, and should not be scheduled to run until that event occurs.

e. (3 marks)

Several techniques for preventing deadlocks by restricting the behaviour of concurrent applications were discussed in class. Name one such technique, and briefly (in one or two sentences) describe it.

No Hold-and-Wait: a thread should not request a resource while it has exclusive access to some other resource. **Resource Ordering:** impose an ordering on the resource types, and require that each new resource the a thread requests be greater (in the resource ordering) than any resources it current holds **Premption:** when a process requests a resource that is already allocated to another process, take it away from the other process and give it to the requestor.

f. (3 marks)

Explain, briefly, what the processor does when it executes a system call instruction, e.g., the `syscall` instruction on a MIPS processor.

- (parts of) the current thread context, e.g., the program counter, are saved
- the processor is switched into privileged execution mode
- control is transferred to a fixed location within the kernel

g. (2 marks)

Consider a process running in a system whose MMU uses a TLB to speed up virtual address translation. Suppose we say that a particular virtual address in the process's virtual address space is part of the current "TLB footprint" if it can be translated to a physical address using the information that is currently in the TLB. The full "TLB footprint" for this process consists of all such virtual addresses. Suppose that the TLB can hold 64 translation entries, and the system page size is 1024 (2^{10}) bytes. What is the maximum size (in bytes) of the TLB footprint?

The size of the TLB footprint is 64 pages, which is $64 * 2^{10} = 2^{16}$ bytes.