

Midterm

CMSC 412
Operating Systems
Spring 2007

March 28, 2007

Guidelines

This exam has 8 pages (including this one); make sure you have them all. Put your name on each page before starting the exam. Write your answers directly on the exam sheets, using the back of the page as necessary. Bring your exam to the front when you are finished. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

For questions that ask you to make calculations, you need not actually perform the arithmetic. For example, if the answer to a particular question is $275(0.8) + 45$, you can just write this rather than actually performing the arithmetic to write 265.

You may avail yourself of the punt rule. If you write down punt for any part of a question with a specifically-assigned point value, you will earn $1/5$ of the points for that question (rounded down).

Use good test-taking strategy: read through the whole exam first, and first answer the questions that are easiest for you and are worth the most points.

Question	Points	Score
1	20	
2	20	
3	25	
4	20	
5	15	
Total	100	

1. (Short Answer, 20 points)

In three sentences or less, compare and contrast the following terms.

(a) interrupt vs. system call **Answer:**

System calls are typically implemented using a software-generated interrupt (or trap). Interrupts are used by the hardware to signal other events as well, such as completed I/O, a timer expiry, or exceptional operations (e.g., divide-by-zero). Interrupts, whether signaled by hardware or software, are requests for service from the OS kernel.

(b) thread vs. process **Answer:**

A thread is basically a representation of control, i.e., a stack and a register file. A process is a running program, which includes one or more threads of control, as well as an address space. Thus, threads share memory, whereas processes do not (at least directly). Processes are always scheduled by the OS, whereas threads may be scheduled by their host process (a.k.a. user-level threads) or the OS (a.k.a. system-level threads).

(c) interrupt-driven vs. polling I/O **Answer:**

With polling, the CPU continually checks whether an I/O operation can be initiated (or has completed), whereas the alternative is for the device to raise an interrupt in these cases. Polling avoids the need to context-switch, reducing overhead, but occupies the CPU, inhibiting parallelism.

(d) blocking vs. busy waiting **Answer:**

When some condition cannot be satisfied immediately, the current process/thread can either block or busy-wait. In the former case, it is context-switched and some other process runs until an event indicates the condition may be true, in which case the blocked process can be rescheduled and check the condition again. In the latter case, the process continuously checks the condition until it becomes true. Examples of these two are interrupt-driven and polling I/O, respectively.

2. (Scheduling, 20 points)

- (a) (5 points) Suppose some process P , when run, uses 10 “ticks” of CPU time while having 5 “ticks” for I/O. If P runs alone (i.e., not in parallel with any other processes) on a single-processor system, what will be its turnaround time and waiting time? **Answer:**

The waiting time is 0, since it never sits in the ready queue at all (I/O is not counted in waiting time). The turnaround time is the total running time (not just the CPU time): 15 ticks.

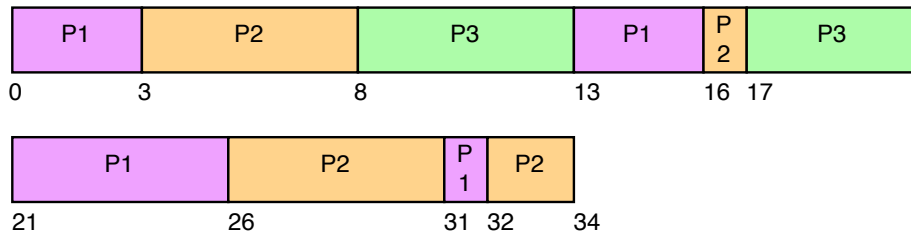
- (b) (14 points) Suppose we have three processes with the following CPU burst times:

Process	Burst Times
p_1	3, 3, 6
p_2	6, 7
p_3	9

Draw a Gantt chart to illustrate how these processes would be scheduled using Round Robin (RR) and MLF scheduling, and calculate the waiting time for each process.

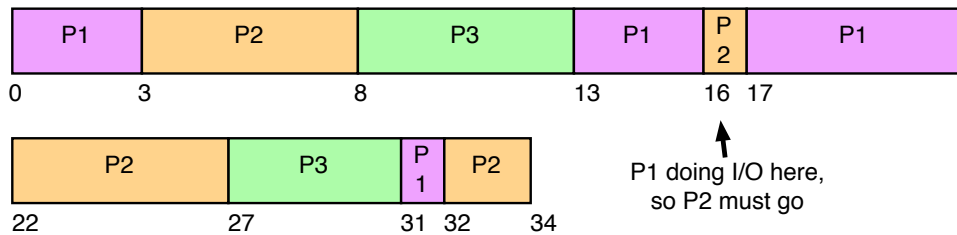
The MLF algorithm to use should be the one you implemented in GeekOS for project 3 *but using only two queues rather than four*. For both RR and MLF, assume the initial run queue is ordered as listed in the chart (i.e., p_1 runs first). *Do not include context-switching time in your chart*. Assume that the quantum is set to 5 time units. Finally, assume that at the completion of each CPU burst the process must wait (at least) one tick to perform I/O before it is eligible to run again.

- i. (7 points) Round Robin **Answer:**



Waiting times: $p_1 = 32 - (3 + 3 + 6) - 2 = 18$; $p_2 = 34 - (6 + 7) - 1 = 20$; $p_3 = 21 - 9 = 12$

- ii. (8 points) GeekOS MLF with two queues **Answer:**



Waiting times: $p_1 = 32 - (3 + 3 + 6) - 2 = 18$; $p_2 = 34 - (6 + 7) - 1 = 20$; $p_3 = 31 - 9 = 22$

3. (Deadlock, 25 points)

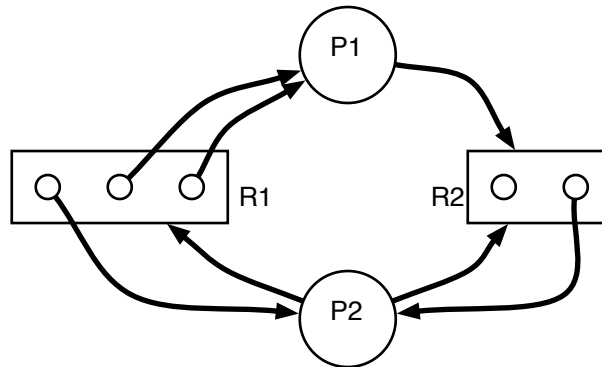
- (a) (5 points) One of the necessary conditions for deadlock is “no preemption.” Indicate a resource that is preemptible and one that is not; use one sentence each to justify your answers.

Answer:

The CPU is a preemptible resource because the relevant computational state can be saved to preempt the CPU and then restored later. Another possibility is a database: a transaction can be aborted (i.e., the DB preempted) and later restarted if two processes attempt to access the same DB entries at the same time.

Just about any I/O device is non-preemptible, since if one process were to start using it while another is, the I/O delivered from/to would be interleaved, which is likely to be nonsensical.

- (b) (6 points) Consider the following resource allocation graph.



- i. Is the system deadlocked? **Answer:**

No, as both P1 and P2 can still make progress (e.g., they can acquire R2).

- ii. What is the status of the system if P1’s request for R2 is granted first? **Answer:**

Safe. After the resource is granted P1 can execute instructions because it is not waiting on any resource. If it terminates it will release its resources and then P2 can make progress.

- iii. What is the status if P2’s request for R2 is granted first? **Answer:**

Deadlock. Both P1 and P2 will be waiting on resources the other holds.

(over)

For GeekOS you implemented semaphores with the following user-side system call API:

```
int Create_Semaphore(const char *name, int ival); // create sem with given name
int P(int sem); // wait on sem
int V(int sem); // signal sem
int Destroy_Semaphore(int sem); // destroy sem
```

- (c) (7 points) The following program can deadlock. Draw a resource allocation graph (where the semaphores are the resources) that represents the results of an execution resulting in a deadlocked state.

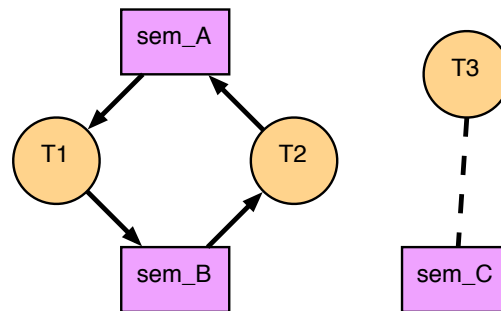
```
/* init. code run before threads start */
int sem_A = Create_Semaphore("a",1);
int sem_B = Create_Semaphore("b",1);
int sem_C = Create_Semaphore("c",1);
int A = 0;
int B = 0;
int C = 0;

thread1() {
    P(sem_A);
    A += 10;
    P(sem_B);
    B += 20;
    A += B;
    V(sem_B);
    A += 30;
    V(sem_A);
}

thread2() {
    P(sem_B);
    B += 10;
    P(sem_A);
    A += 20;
    A += B;
    V(sem_A);
    B += 30;
    V(sem_B);
}

thread3() {
    P(sem_C);
    C += 10;
    P(sem_B);
    B += 30;
    C += B;
    V(sem_B);
    C += 5;
    V(sem_C);
}
```

Answer:



(It really doesn't matter what state thread3 and sem_C are in.)

- (d) (7 points) Show how to fix the above program so that it can no longer deadlock, but is equivalent to the original semantically, and still allows some parallel execution. Indicate which of the four necessary conditions for deadlock you nullified to fix it; justify your answer.

Answer:

You could fix it by avoiding circular waiting between A and B by reordering accesses so that semaphores are decremented in the order $sem_A \leq sem_C \leq sem_B$. Thus you could change thread2 to move the first two statements immediately after the statement $A += 20$ to satisfy this ordering while not changing the semantics of the program.

4. (Semaphores, 20 points)

- (a) (5 points) Suppose the two threads must execute so that code block **Block1** always executes prior to **Block3** and **Block2** executes prior to **Block4**. Enforce this ordering using semaphores while allowing the two threads to execute in parallel as much as possible. Do this by annotating the code below with where the semaphore operations must be inserted.

```
thread1() {                                thread2() {

                                            P(sem);

    Block1                                Block3

    V(sem);

    ... other code here ...                ... other code here ...

                                            P(sem);

    Block2                                Block4

    V(sem);

    ... other code here ...                ... other code here ...

}                                           }
```

Answer:

Change the program to insert semaphore operations as above, initializing the semaphore to be 0.

- (b) (7 points) Semaphores can be used to implement mutual exclusion locks, which are often used to enforce critical sections, e.g., as follows:

```
lock_t x;
init(&x);
lock(x);
... critical section ...
unlock(x);
```

Illustrate how you can implement the type `lock_t` and the functions `init`, `lock`, and `unlock` above using the semaphore API for GeekOS presented earlier.

```
typedef int lock_t;

void init(lock_t *x) {
    *x = Create_Semaphore(...,1);
}

void lock(lock_t x) {
    P(x);
}

void unlock(lock_t x) {
    V(x);
}
```

- (c) (8 points) In Java, locks are *reentrant*, which means that the same thread can acquire a lock more than once (i.e., without blocking on the second acquire). Modify your implementation of `lock` and `unlock` above to allow them to be reentrant. *Hint: you may want to associate a process owner and a counter with each lock, where the counter tracks the number of times the lock has been acquired by the current owner.*

```
typedef struct {
    int mutex;
    PID *p;
    int sem;
    int *cnt;
} lock_t;

void init(lock_t *x) {
    x->mutex = Create_Semaphore(...,1);
    x->sem = Create_Semaphore(...,1);
    x->cnt = malloc(sizeof(int));
    *x->cnt = 0;
    x->pid = malloc(sizeof(PID));
    *x->pid = -1;
}

void lock(lock_t x) {
    P(x.mutex);
    if (currentPID() == *(x.pid)) {
        *(x.cnt)++;
    } else {
        V(x.mutex);
        P(x.sem);
        P(x.mutex);
        *(x.cnt)++;
        *(x.pid) = currentPID();
    }
    V(x.mutex);
}

void unlock(lock_t x) {
    P(x.mutex);
    if (currentPID() == *(x.pid)) {
        *(x.cnt)--;
        if (*(x.cnt) == 0) {
            *(x.pid) = -1;
            V(x.sem);
        }
    } // else error!
    V(x.mutex);
}
```

5. (Memory Management, 15 points)

- (a) (4 points) Describe one reason for having a logical address space per process (as opposed to all processes sharing the same physical address space). **Answer:**

Possible answers: (1) improves security (both confidentiality and integrity) since each process now has its own address space and so cannot refer to addresses in another process; (2) simplifies program construction, since the compiler can construct programs to assume they start at address 0, while the operating system can load them at different physical addresses; (3) enables memory management strategies like paging that can be used to share physical memory and/or reduce fragmentation while allowing the process to have a contiguous address range; etc.

- (b) (5 points) Suppose we are using two-level paging with a translation lookaside buffer (TLB), where we have empirically measured the hit rate for the TLB to be $\alpha = 0.8$. If each access to memory is 6 time units, and the time to access the TLB is 1 time unit, what is the effective access time (EAT) of each memory access? **Answer:**

$$EAT = 1 + 6 + (0.2)(12)$$

The first element is the time to access the TLB and the second is to access the actual memory; these costs are incurred at every access. The last term is the time to access the page table; this happens $(1-\alpha)$ times, and the cost is two memory accesses, one for each level of the table.

- (c) (6 points) Suppose we have an 16 bit address space (i.e., 64 KB of memory, max) with single-level paging (where each process has its own single-level page table). Allocated memory for a process can be wasted due to internal fragmentation, unused page table entries, or both. Assume that each page table entry is 2 B (2^4 bits), and the typical process requires about 1 KB (2^{10} bits) of memory.

We wish to determine the page size $p = 2^n$ bits for some n that would minimize the amount of wasted memory for the average process. How would you make this determination? Show mathematical formulas that justify your answer.

Answer:

The goal is to minimize the sum of the average internal fragmentation and the average wasted page table entries, where each is a function of n , exponent in the page size. Thus we need to calculate both components of the formula.

Internal Fragmentation *Assuming processes are at least a page, the internal fragmentation comes from the unused bit on the last page allocated to the process. On average, we can assume half of this last page is wasted, i.e., the wasted space is $(p/2) = (2^n/2) = 2^{n-1}$ bits.*

Wasted PT entries *The total number of page table entries is $(2^{16}/p) = (2^{16}/2^n) = 2^{16-n}$. The number of entries used by a 1 KB process is $(2^{10}/p) = 2^{10-n}$. The number of unused entries is thus $\lfloor 2^{16-n} - 2^{10-n} \rfloor$. Finally, since each entry is 2 B, the wasted space is $\lfloor 2^{16-n} - 2^{10-n} \rfloor 2^4$ bits.*

Putting these together, we wish to solve for n in the formula:

$$\min(2^{n-1} + \lfloor 2^{16-n} - 2^{10-n} \rfloor 2^4)$$