

**CSC C69H1 F
OPERATING SYSTEMS**

**UNIVERSITY OF TORONTO
FALL 2003**

Midterm Test

NO AIDS ALLOWED

Please **PRINT** in answering the following requests for information:

Family Name: _____

Given Names: _____

Student Number: | _ _ _ | | _ _ _ | | _ _ _ |

Login (@fissure): _____

Notes to students:

1. This test lasts for 60 minutes and consists of 50 marks. Budget your time accordingly.
2. This test has 6 pages and 5 questions.
3. **Write in pen. No pencils.**
4. Write your answers on this “question and answer” paper, in the spaces provided. Be concise. In general, the amount of space provided is an upper bound on the “size” of answer that is expected. If necessary, use space where available and provide explicit pointers.
5. In general, state your assumptions and show your intermediate work, where appropriate.
6. Do **not** go beyond here until instructed to do so. Write your student number at the top of each succeeding page once you get going.

Question	Marks
1	
2	
3	
4	
5	
Total	

1. [10 marks, 2 each] True/False

Circle "T" if the statement is *always* true. Otherwise circle "F".

- a) Multiple threads in the same process share all code and data resources **F**
- b) Hardware support is needed to synchronize more than two processes. **F**
- c) It is possible for a single process to deadlock with itself. **T**
- d) Minimizing turnaround time is the most important criteria in choosing a scheduling algorithm for an interactive system. **F**
- e) With dynamic partitioning, all memory allocated for a process must be contiguous. **T**

2. [9 marks; 3 each]

Define the following terms in the context of this course:

(a) Aging

A mechanism for gradually increasing the priority of processes that wait in the system for a long time to prevent starvation.

(b) Compaction

A way to deal with external fragmentation in dynamic partitioning systems (or contiguous memory allocation systems) by relocating the contents of memory to place all the free memory together in one large block.

(c) Multiprogramming

Interleaving the execution of two or more processes on a single processor to increase CPU utilization. (Giving each process the illusion of a dedicated CPU is not a requirement for multiprogramming; having multiple processes in memory at the same time, however, is a requirement.)

3. [12 marks; 6 each]

Consider the following program to demonstrate the operation of `pthread_create` and `pthread_join`. The `pthread_join` function blocks the calling thread until the specified thread has exited. If the thread has already exited at the time `pthread_join` is called, the caller continues execution without blocking. Assume the existence of a "pthread_attr_setpriority" function which allows the priority for a new thread to be specified. Assume also that higher numbers correspond to higher priorities, and that all `printf` statements are executed without blocking the caller

```
int main() {
    pthread_t t;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setpriority(&attr, 10);
    printf("bam ");
    pthread_create(&t, &attr, A, 0);
    printf("baf ");
    pthread_join(t, 0);
    printf("blam ");
    pthread_exit(0);
}

void *A(){
    pthread_t t2;
    pthread_attr_t attr2;

    pthread_attr_init(&attr);
    pthread_attr_setpriority(&attr, 20);
    printf("umph ");
    pthread_create(&t2, &attr, B, 0);
    printf("ugh ");
    pthread_join(t2, 0);
    printf("urk ");
    pthread_exit(0);
}

void *B() {
    printf("krak ");
    pthread_exit(0);
}
```

(a) Assume that the scheduler runs threads using a FCFS policy, ignoring priorities. What will the program above print out?

bam baf umph ugh krak urk blam

(b) Assume that the scheduler runs threads using a preemptive priority policy, and that the priority of the "main" thread is 0. What will the program above print out now?

bam umph krak ugh urk baf blam

4. [8 marks; 4 each]

The buddy system is a method to do memory allocation, as described below.

Given a memory space of 2^m words, we maintain a separate free list for each block size 2^k , where $0 \leq k \leq m$. If a block of size s is requested such that $2^i < s \leq 2^{i+1}$, then a block of size 2^{i+1} will be allocated. Originally, there is one free block of size 2^m words.

To allocate a block of size 2^k , we find the first available block of size 2^j , such that $k \leq j \leq m$. If $j = k$, then we are done. Otherwise we split the block into two equally-sized buddies which are both placed on the appropriate free list. The search and split repeats until $j = k$. Appropriate coalescing is done upon return of a used block.

a) **Describe** where external fragmentation is present in this scheme.

Each allocation has the potential to split an existing free block in half multiple times. After the first allocation request, the largest available block is only 2^{m-1} (even if the first request was for as small as 1 byte!). The free space becomes broken up into various power-of-2 sizes.

b) **Describe** where internal fragmentation is present in this scheme.

Requests are rounded up to the next larger power-of-2 size, so processes will not use all of the memory that is allocated to them; some of the space in the allocated block will be wasted.

5. [11 marks; breakdown given below]

You have been hired as an OS consultant to solve a problem with a client's system. Their OS has a set of queues, each of which is protected by a lock (implemented as a `pthread_mutex_t`). To enqueue or dequeue an item, a thread must hold the lock associated with the queue.

They have implemented an *atomic transfer* routine that dequeues an item from one queue and enqueues it on another. The client requires that the transfer appear to occur atomically. The transfer routine is being used extensively throughout their multithreaded system, and may be called by many different threads with any of the system queues as inputs.

Unfortunately, their first attempt, shown below, causes the system to deadlock.

```
void transfer(Queue *queue1, Queue *queue2)
{
    Item *thing; /* the thing being transferred */
    pthread_mutex_lock(&queue1->lock);
    thing = Dequeue(queue1);
    if (thing != NULL) {
        pthread_mutex_lock(&queue2->lock);
        Enqueue(queue2, thing);
        pthread_mutex_unlock(&queue2->lock);
    }
    pthread_mutex_unlock(&queue1->lock);
}
```

You have already verified that the Enqueue and Dequeue functions are correct, that the synchronization variables have all been initialized properly, and that the transfer function is never called with the same queue for both inputs (that is, the client has guaranteed that `transfer(queueA, queueA)` will never occur).

(a) [5 marks] Demonstrate how the use of the transfer function can lead to deadlock.

Suppose thread T1 calls `transfer(queueA, queueB)` and T2 calls `transfer(queueB, queueA)`. We could have the following sequence of events:

T1:
`pthread_mutex_lock(&queueA->lock);`

`thing = Dequeue(queueA);`
`if (thing != NULL) {`
 `pthread_mutex_lock(&queueB->lock);`
 `/* blocks waiting for queueB to be free */`

T2:

`pthread_mutex_lock(&queueB->lock);`
`thing = Dequeue(queueB);`

`if (thing != NULL) {`
 `pthread_mutex_lock(&queueA->lock);`
 `/* blocks waiting for queueA to be free */`

(b) [2 marks] Which of the 4 conditions for deadlock could be easily removed to fix this problem?

Either hold-and-wait, or circular-wait could be broken. (Breaking mutual exclusion would violate the atomicity of the transfer function, and could cause the queues to become corrupted; breaking “no preemption” would require a way to roll back a partially complete transfer, which wouldn’t be easy)

(c) [3 marks] Briefly describe (in words) how you would solve the problem.

(i) Break the hold-and-wait condition on queue locks: Introduce a new, global lock (e.g. `pthread_mutex_t transfer_lock`). In the transfer function, acquire the new lock before locking either queue involved in the transfer. Release the new lock only when the transfer is complete. Note that all other operations on the queues would have to hold the new `transfer_lock` as well to really break the hold-and-wait condition.

(ii) Break the circular wait condition: Use the addresses of the queue variables to create a total order of queues in the system. The transfer function could be re-written to always lock the queue with the lower address first. This is preferable to a single global lock as it allows greater concurrency.

(d) [1 mark] Which of the 3 strategies for dealing with deadlock does your solution use?

Both of the strategies above use deadlock **prevention** because they prevent one of the four conditions for deadlock from occurring.

Total marks = (50)

End of test