1. (20%, 2% each) Answer each of the following questions concisely but precisely.

- Why is the busy-waiting in the following implementation of semaphore wait() on multiprocessors not considered a problem?

```
void wait(semaphore s)
{
   disable interrupts;
   while (ldl(s->lock) != 0 || !stc(s->lock, 1));
   if (s->count > 0) {
      s->count--;  s->lock = 0; enable interrupts;
      return;
   }
   add(s->q, current_thread);  s->lock = 0;
   enable interrupts;  sleep();
}
```

- Why do we still disable interrupts to prevent context switch in addition to using the lock with TAS in the following implementation of semaphore wait() for multiprocessors?

```
void wait(semaphore s)
{
   disable interrupts;
   while (TAS(s->lock, 1) == 1);
   if (s->count > 0) {
      s->count--;  s->lock = 0; enable interrupts;
      return;
   }
   add(s->q, current_thread);    s->lock = 0;
   enable interrupts;  sleep();
}
```

- Consider CPU-only jobs. Assume all jobs have already arrived. Can Round-Robin ever be the worst possible preemptive CPU scheduling algorithm in terms of average turn around time? If so, under what circumstances? If not, explain why not.

- In a generic storage allocation problem, what are the fundamental causes for external fragmentation?

- What are the tradeoffs in choosing the page size in a paged main memory management system?

- My home PC has 512 MB physical memory. Under its 32-bit architecture, and with a page size of 4KB, a page table takes up about 4MB. So why is the page table size a big deal?

- List at least two benefits of using a two-level page table compared to a one-level page table?

- In a paging system, what is the motivation for using an inverted page table?

- In a paging system, when malloc(16385) invoked by user process P successfully returns, how many physical pages have been allocated to process P, assuming a page size of 4096 bytes? Are they consecutive in the physical memory?

- Assume the page size is chosen as 4 Kbytes. Is a three-level page table a good idea for supporting 32-bit address spaces, and why? What about 64-bit address spaces, and why?

2. (True-or-False - 20%, 2% each) For each of the statements below, indicate in one sentence whether or not the statement is true or false, and why.

   - When a file is being accessed by a process, i.e., read or written, the current position within the file where it is being accessed is stored in the on-disk copy of its inode.

   - When a file is being accessed by a process, i.e., read or written, the current position within the file where it is being accessed is stored in the in-memory copy of its inode.

   - A multi-level indexed file descriptor permits faster random access than a file descriptor with a single level of index.

   - In the Unix File System, the name of a (non-directory) file is stored in its inode.

   - In the Unix File System, the name of a directory is stored in its inode.

   - In the UNIX file system, after file system initialization, there is a fixed upper limit on how large files can be.

   - In the Unix File System, the operation "ls -l" is potentially more expensive and hence slower than "ls".

- In File Systems, the buffer cache is implemented purely in software, unlike demand paging, which is implemented jointly in software and with hardware support (i.e. MMU).

- It is slower to write 1 block in a RAID Level 5 organization with 5 disks than in a RAID Level 1 organization with 2 disks (i.e., mirroring only).

- In disk scheduling, the SSTF (shortest seek time first) scheduling algorithm tends to favor middle cyclinders over the innermost/outmost cylinders.

3. (10 pts) (Deadlock avoidance - Banker's Algorithm) In Deadlock-avoidance algorithms such as the Banker's algorithm, the resource-allocation state is *safe* if the system can allocate resources to each process (up to its maximum) in some oder and still avoid a deadlock.

   Five processes in a system are sharing 4 types of resources A, B, C, and D. Consider the following snapshot of the system:

	Maximal Needs	Currently Held	Remaining Needs	Available Drives:
	A B C D	A B C D	A B C D	A B C D
P0	0 0 1 1	0 0 1 1	0 0 0 0	1 5 2 0
P1	1 7 5 0	1 0 0 0	0 7 5 0	
P2	2 3 5 6	1 3 5 4	1 0 0 2	
P3	0 6 5 2	0 6 3 2	0 0 2 0	
P4	0 6 5 6	0 0 1 4	0 6 4 2	

   Answer the following questions using the banker's algorithm:

   (1) In this system in a safe state? Explain why.

   (2) If P4 requests for (0, 5, 1, 0), is it safe to grant the request? Explain why.

   (3) If instead, P1 requests for (0, 4, 2, 0), is it safe to grant the quest? Explain why.

Write in Exam Book Only

4. (Synchronization - 20 pts) The abstract "too much milk" problem that affects the daily lives of k roommates who share the purchasing and consumption of milk stored in a fridge goes as follows. At any time, when a roommate wants to drink milk, he/she looks into in the fridge. If the bottle of milk is empty, it is his/her responsibility to go to Walmart to buy a new bottle of milk. The challenge arises if the next roommate does the same thing before the first roommate comes back from Walmart, they can end up with two bottles of milk. But since milk does not last long, the roommates agreed that they should develop a scheme so that they never end up with more than one bottle of milk in the fridge.

   Using OS-provided lock API, they arrived at the following simple solution (in pseudo-code) to the "too much milk" problem. However, the action of buying milk, which can potentially take a long time to accomplish (a roommate may decide to walk to Walmart), is inside the critical section. This can cause the next roommate waiting to acquire the lock and enter the critical section (it could be you!) to wait for a long long time.

   Can you revise the pseudo-code to avoid this problem? If you need to use some flags, clearly state if a flag is shared or private. No partial credit for any solution that causes "too much milk" or "no milk".

```
// symmetric code for all roommates

lock mutex;

while(1) // this is their daily life
{

// do other things they do in life

lock_acquire(mutex);

if (no milk)
   go to Walmart to buy milk;

lock_release(mutex);
}
```

5. (Page Replacement - 30%)

   (a) (10%) A page replacement algorithm belongs to the class *stack algorithms* if it can be shown that given any sequence of virtual page references, the set of pages that would be in memory with n physical pages is always a subset of the set of pages that would be in memory with $n+1$ physical pages. Which of the following page replacement algorithms are stack algorithms? (1) OPT (2) LRU (3) FIFO (4) LIFO (5) LFU (Least Frequently Used)

   (b) (6%) Consider the following virtual page reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5, 1, 2. How many page faults will there be under the LRU replacement algorithm on Tom's PC which has 4 physical pages? How many page faults will there be on Jerry's machine which has 3 physical pages?

   (c) (6%) Assuming the LRU replacement algorithm again. Tom has upgraded his machine to 16 MB (4 thousand pages), and Jerry has upgraded his machine to 12 MB (3 thousand pages). Can you come up with a reference string that contains more than 4000 references and still exhibits the above relative performance behavior, i.e., the ratio of the numbers of page faults on the two machines remain the same?

   (d) (8%) Going back to the reference string in (b). What is the minimum number of page faults for an optimal page replacement strategy, assuming 4 physical pages and assuming 3 physical pages, respectively?