UNIVERSITY OF TORONTO
Faculty of Arts and Science

APRIL/MAY 2007 EXAMINATIONS

CSC 369H1S

Duration - 3 hours

Examination aids allowed: one 8.5 x 11" (letter-sized), double-sided "fact sheet" from the student.

Name: _____

Student #: _____

Notes to students:

1. There are 10 questions and 16 pages in total (including this cover sheet and an extra page at the back) for this exam.
2. Answer all questions directly on the examination paper. Generally, the space allowed is a clue to the size of answer expected. Use the final page of the exam, or the backs of pages if more space is needed, and provide clear pointers to your work.
3. Write your name and student number at the top of each page.
4. Show your intermediate work and BRIEFLY state your assumptions where appropriate.
5. Write clearly and legibly. If we can't read your answer, we can't grade it.
6. **Read the questions carefully and answer the question that is being asked.**
7. **Good luck!**

Question	Marks	Question	Marks
1	/ 12	6	/ 10
2	/ 18	7	/ 12
3	/ 18	8	/ 15
4	/ 12	9	/ 12
5	/ 9	10	/ 20
	Total:		**/ 138**

Continued…

1. True / False [12 points]

TRUE **FALSE** Operating systems rely on hardware support for memory protection.

TRUE **FALSE** Kernel code can disable interrupts to make short critical sections atomic on multiprocessors.

TRUE **FALSE** Starvation happens when some processes are continually preferred over others.

TRUE **FALSE** Priority inversion happens when a low priority process prevents a high priority process from making progress by holding some resource.

TRUE **FALSE** We study dynamic partitioning purely for historical reasons.

TRUE **FALSE** CLOCK is an example of a stack algorithm for page replacement.

TRUE **FALSE** Two-level page tables are an example of a space-time tradeoff.

TRUE **FALSE** Opening a file using a hard link requires more disk accesses than opening the same file using a symbolic link.

TRUE **FALSE** Disks are improving most rapidly in terms of seek time.

TRUE **FALSE** FFS performs synchronous writes for performance reasons.

TRUE **FALSE** RPC provides a general mechanism for writing distributed applications.

TRUE **FALSE** Good security protects a system against accidental attacks as well as intentional ones.

Continued…

2. Acronym Bingo [18 points, 3 each]

For each of the following, (1) expand the acronym, (2) briefly explain what it is, and (3) state whether it relates to an operating system *policy* or *mechanism*.

i) MMU

Memory Management Unit – hardware device that performs memory access checks and virtual-to-physical address translation.
Mechanism.

ii) FCFS

First Come First Served – a scheduling policy in which jobs are scheduled in the order that they arrive.
Policy.

iii) PCB

Process Control Block – software data structure used by operating system to keep track of the state of a single process.
Mechanism.

iv) ACL

Access Control List – a list associated with each object in a protection system (e.g. each file in a file system) that contains the subjects (e.g. users) and the access permissions that they have on the object (e.g. read/write/execute).
Mechanism.

v) LRU

Least Recently Used – a page replacement policy that selects the page whose last access is farthest in the past (i.e., least recent) as the victim.
Policy.

vi) SJF

Shortest Job First – a scheduling policy that selects the job with the smallest processing requirement (i.e., the shortest one); minimizes average wait time but can cause starvation.
Policy.

Continued…

3. Miscellaneous Fill in the Blanks [18 points; 3 each]

(i) Consider a set of cooperating tasks, implemented as either a set of traditional single-threaded processes, or as a set of threads in a multi-threaded process. For each of the following operations, indicate whether it will be *faster or easier overall* if the tasks are implemented as single-threaded processes (**P**), kernel-level threads (**K**), or user-level threads (**U**). Assume a many-to-one mapping of user-level threads to kernel threads. If multiple options are equally good, list all that apply.

_*U*____ **Creating new tasks**

_*P*____ **Preventing accidental modification of other tasks' private memory (e.g. stack)**

_*P,K*__ **Making blocking system calls**

_*U*____ **Switching between tasks**

_*P*____ **Creating a very large number of tasks**

_*U*____ **Customizing task scheduling**

(ii) Fill in the following table by writing "**I**" if the combination of row and column headings for that cell is impossible, or writing "**P**" if the combination is possible.
The first cell (first row, first column) represents "TLB hit and virtual page in memory".

	Virtual page in memory	**Virtual page not in memory**	**Invalid address exception**
TLB Hit	*P*	*I*	*I*
TLB Miss	*P*	*P*	*P*

(iii) Consider a system with a hardware-loaded TLB, which provides no software interface to update entries. The only software options are to flush the TLB (invalidate all entries) or invalidate a specific entry. For each of the following events, indicate whether the OS software should flush the TLB (**F**), invalidate an entry in the TLB (**I**), or take no action with respect to the TLB (**N**).

__*N*___ **CPU interrupt**

__*F*___ **Context switch**

__*N*___ **System call other than fork or exec**

__*F*___ **fork, implemented with copy-on-write**

__*I*___ **copy-on-write fault**

__*I*___ **page eviction**

This question continues on the following page.

(iv) For each of the following, indicate whether it relates more closely to a technique for deadlock *prevention* (**P**), *avoidance* (**A**), or *detection and recovery* (**D**).

__*A*___ **Determining safe and unsafe states**

__*P*___ **Requesting all resources at once**

__*D*___ **Constructing the resource allocation graph**

__*A*___ **the Banker's Algorithm**

__*D*___ **Killing some or all processes involved in a cycle**

__*P*___ **Assigning a linear ordering to resource types**

(v) For each of the following, indicate whether it improves performance mainly by *increasing throughput* (**T**), *decreasing response time* (**R**), or both (**B**).

__*R*___ **Using a short CPU scheduling quantum**

__*T*___ **Striping files across disks in a RAID** *(I decided to take "B" as well)*

__*T*___ **Using SSTF disk scheduling**

__*R*___ **Raising the priority of processes that block before using up their quantum**

__*B*___ **Caching file blocks in memory**

__*T*___ **Log-structured file systems (consider data writes specifically)**

(vi) For each of the following, indicate whether it relates more closely to an attack on *confidentiality* (**C**), *authenticity* (**A**), or *integrity* (**I**). The relation may be a either a type of attack, or a defense against one of these attacks.

__*C*___ **Eavesdropping**

__*A*___ **Digital Signatures**

__*C*___ **Message Encryption**

__*I*___ **Cryptographic checksums**

__*A*___ **Masquerade**

__*C*___ **Traffic Analysis**

Continued…

4. Synchronization Problem: Message Passing [12 points]

Message passing implementations come in many forms. In one variant, called a *rendezvous*, both sending and receiving a message are blocking operations. That is, a thread calling the msg_send() function will block until the recipient calls msg_recv(). Similarly, if a thread calls msg_recv(), it must block until some other thread calls msg_send(). When sending a message, the intended destination must be specified, however, messages can be received from any source, with the id of the sender returned along with the message.

To simplify the problem, we assume that messages are of a fixed length, MSG_LEN, and that thread ids are chosen from a small range of integer values, [0..TID_MAX-1]. We define an array of *mailboxes*, one per thread, indexed by thread id, to help implement the functions.

```
struct mailbox {
   char *dest_buf;     /* set by receiver */
   int sender_id;      /* set by sender */
   struct semaphore mutex;       /* initialized to 1 */
   struct semaphore recvr_ready; /* initialized to 0 */
   struct semaphore sender_done; /* initialized to 0 */
};

struct mailbox mailboxes[TID_MAX];
```

(i) [8 points] Complete the implementation of the message passing functions msg_send() and msg_recv() on the following page. Remember that a thread may receive messages from multiple senders. [You should at least read the partial implementation before answering the following two questions.]

(ii) [2 points] What is the main *advantage* of using blocking sends and receives in this way?

No intermediate storage is needed to hold sent messages until the receiver is ready to receive them.

Also acceptable: The sender knows the receiver has the message when it returns from send (or else it has any error immediately, although the functions on the following page don't return errors), which may simplify programming since we don't need to check separately that the receiver got the message.

(iii) [2 points] What is the main *disadvantage* of using blocking sends and receives?

The sender must wait for the receiver before the message can be sent, instead of going on to other work. The receiver also waits, but this may be less of a problem if we assume the receiver needs to get the message before it can continue with its work anyway.

Also acceptable: less fault-tolerant, as sender will block forever if receiver dies or doesn't ask to receive message for any reason.

This question continues on the following page.

```
/* Add the necessary calls to P() and V() on the mailbox semaphores.
You do not need to add code to every blank space. */

void msg_send(char *msg, int recvr_id) {
   struct mailbox *mbox = &mailboxes[recvr_id];

   /* mutex was an unintentional red herring.  The recvr_ready
    * semaphore will serve the purpose of only allowing one
    * sender to copy its message into the provided buffer at
    * a time.  Use of mutex is ok as long as it doesn't cause
    * a deadlock.
    */

   /* wait for the receiver to set dest_buf */
   P(mbox->recvr_ready);

   memcpy(mbox->dest_buf, msg, MSG_LEN);




   mbox->sender_id = get_current_thread_id();

   /* signal that sender is done copying message and setting id */
   V(mbox->sender_done);

}

void msg_recv(char *msg, int *sender_id) {
   struct mailbox *mbox = &mailboxes[get_current_thread_id()]);







    mbox->dest_buf = msg;

    /* Signal sender that receiver is ready to get message */
    V(mbox->recvr_ready);

    /* Wait for sender to send message */
    P(mbox->sender_done);

   *sender_id = mbox->sender_id;




}
```

5. Scheduling [9 points; 3 each]

(i) <u>Explain</u> the difference between a preemptive scheduler and a non-preemptive scheduler, and **<u>give one example</u>** of the type of system where each might be used.

A preemptive scheduler will force a context switch upon certain events (time slice expires, higher priority process becomes runnable, etc.) while a non-preemptive scheduler will wait for the running process to voluntarily yield the CPU (blocking or exiting) before scheduling a new process.

Preemptive systems: real time systems, time shared systems, interactive systems
Non-preemptive systems: batch systems, single-application systems (some embedded systems for example) … some real-time systems are also non-preemptive, relying on cooperative scheduling to meet deadlines, for greater predictability.

(ii) Are synchronization problems easier to solve when preemptive scheduling is used, or when non-preemptive scheduling is used? **<u>Explain why.</u>**

Non-preemptive scheduling makes synchronization easier, because the execution of other processes can't interfere at arbitrary points. Any section of code that does not contain an explicit yield, or a request that could block, automatically becomes atomic.

(iii) What is the main purpose of *lottery scheduling*?

Lottery scheduling provides a way to implement fair share scheduling; the number of tickets a process holds is proportional to the share of the CPU it should be allocated. It allows a very flexible division of CPU between processes belonging to different users or groups.

6. Replacement Algorithms [10 points]

(i) [4 points] **Describe** a specific situation where LRU replacement (or any reasonable approximation of it) will maximize the page fault rate for an application. Refer to the application data access pattern, its working set size, and the size of available memory in your answer.

LRU is bad for looping access patterns (where the same set of data is accessed repeatedly in the same order) when the working set site (i.e. the amount of data accessed in the loop) is larger than the available memory. [The page fault rate for the application will be the same regardless of how much larger the working set is than the available memory]

(ii) [6 points] Consider a new replacement algorithm called *Most Recently Used* (MRU), which evicts the most recently used page. MRU is attractive because it handles the cases where LRU is the worst possible algorithm. **(a) Explain** why exact MRU is unsuitable for virtual memory systems, *even if you had a fast, cheap way to determine the MRU page order*. **(b) Explain** why this problem does not arise if you want to use MRU for file buffer cache replacement.

Continued…

In virtual memory systems, data on pages is always accessed by at most a word at a time. Thus the MRU page is likely to still be in use when the replacement is performed, and the process will immediately fault on the page that was replaced too soon. [What is wanted is the MRU page that is no longer in use, but that is very hard to determine.] The situation is different for the file buffer cache because accesses happen on a block basis – data read is copied from the buffer cache to an application-level buffer for example – so after the access completes the block is no longer in use and can be replaced. [Note that it is possible for an application to read/write a file at a very fine granularity, making the problem identical to the virtual memory system, but at least this is under the application's control.]

Name:_____
Student #:_____

7. File Systems [12 points; 4 each]

Consider a Unix-like file system that maintains a unique index node for each file in the system. Each index node includes 8 direct pointers, a single indirect pointer, and a double indirect pointer. The file system block size is **B** bytes, and a block pointer occupies **P** bytes.

(i) Write an expression for the maximum file size that can be supported by this index node, in terms of **B** and **P**.

- *8 direct data blocks, each of size B → 8B*
- *1 indirect block of size B, holding B/P pointers to data blocks of size B → B B/P*
- *1 double indirect block of size B, holding B/P pointers to indirect blocks, each with B/P pointers to data blocks of size B*

$$\text{max size} = 8B + B\frac{B}{P} + B\frac{B}{P}\frac{B}{P} = 8B + \frac{B^2}{P} + \frac{B^3}{P^2} = B\left(8 + \frac{B}{P} + \frac{B^2}{P^2}\right)$$

(ii) How many disk operations will be required if a process reads data from the **Nth** block of a file? Assume that the file is already open, the buffer cache is empty, and each disk operation reads a single file block. Your answer should be given in terms of **N**, **B**, and **P**.

$$\#\,reads = \begin{cases} 1 & N \le 8 \\ 2 & 8 < N \le 8 + \frac{B}{P} \\ 3 & N > 8 + \frac{B}{P} \end{cases}$$

(iii) As a general rule, the internal structure of the data in Unix files is defined by user code and not known to the kernel (or file system). There are three key exceptions to this rule: directories, symbolic links, and executables. **Pick any two** and **explain** how each one is an exception to the rule and why this exception exists.

1) directories are an exception because their internal structure (directory entries) is defined by the file system and used by the file system to locate files given file names.

2) symbolic links are an exception because their content is expected to be a path (string) giving the real name of the file to which the link points. This must be known and understood by the file system in order to resolve symbolic links.

3) executables are an exception because the kernel must know the internal format to load an executable into an address space (e.g. knowing size and start location of code and data sections) to that the program can be run.

8. Distributed File Systems [15 points; 3 each]

Consider a distributed file system accessed by multiple clients. File system data can be cached in server memory, in client memory, or on client disk (or some combination of these).

(i) <u>Briefly describe</u> at least three (**3**) **<u>distinct factors</u>** that need to be considered when deciding where to cache file data.

A) How will files be shared? If different clients never share the same files, little benefit to server caching. If write sharing is common, client caching may be undesirable.
B) What are the consistency requirements? If we need strict Unix local FS semantics, client caching may be undesirable.
C) What is the client configuration? Do they have large memories? Are they diskless?
D) Scalability – how many clients must the server support?
E) What are the network characteristics? (i.e., bandwidth, latency, likelihood of connection failure)
F) What are the file/application characteristics? Are files large or small? Read sequentially or randomly? Frequently updated? Dynamic content generated by server?

For each of the following scenarios, <u>describe</u> *where* files should be cached and *why*.
(ii) A system that supports *disconnected* operation.

Client disk – allows client to continue using files when server is unreachable; client memory is unsafe because we might not be able to flush modifications back to server for a long time, and RAM is volatile.

(iii) A system that provides a video-on-demand service.

Extra assumptions are needed here. Popular movies are likely to be requested by multiple clients making server memory a good choice. Caching in client memory could facilitate a rewind feature, but if the service does not allow rewind, then client caching is pointless. Client disk can similarly be used for rewind with a larger coverage (i.e., can rewind farther, start from beginning, etc.) but depending on network and server characteristics, re-fetching from server may be faster than the client disk. Also, for digital rights management, service providers may be uncomfortable with caching on client disk.

This question continues on the following page.

(iv) A web server that provides mostly static content.

All three should be used. Web servers expect their content (particularly static content) to be accessed by many clients, so we should definitely cache in server memory. Clients browsing the web site will often visit the same pages multiple times in a short time period, so we want to keep the most recently visited pages in client memory. However, clients will also often visit the same web sites / pages repeatedly over extended periods of time (days or weeks), so caching on client disk is also a good idea.

(v) A system with a very small number of file servers and a very large number of clients.

Client caching is critical to reduce the load on the servers, client disk or client memory doesn't really matter. The servers should also do caching to reduce the service time for frequently-requested files.

9. Security [12 points; 4 each]

(i) Explain what is meant by *security through obscurity*, and **why** it tends to be a bad idea. Give **one example** of a security system that used this strategy and failed.

Attempting to gain security by hiding implementation details. This tends to be a bad idea because the details often don't stay hidden (reverse engineering, social engineering, or a combination exposes them) and the security is compromised.
Example 1: GSM cell phones
Example 2: eBooks (Adobe and others)
Example 3: DVD encryption

(ii) Explain the principle of *complete mediation*, and **describe** the general category of security flaws that result when the principle is not followed, including the name given to this type of bug.

You must check permissions on every access to every object to ensure that nothing has changed since the check was performed. Ignoring this principle leads to "Time of check to time of use" bugs (aka TOCTTOU), which are in reality race conditions that can be exploited by an attacker.

(iii) Define the term *intrusion detection*, and **describe** one method for providing it.

Techniques for determining when a system has been compromised.

One method: signature-based detection, in which signatures of known attacks are constructed and system activity is monitored to see if it matches a known signature. (low false positives, hard to defend against new/unknown attacks)

Another method: anomaly-based detection, in which system activity is monitored to establish a baseline "normal" behavior, and significant deviations are flagged as intrusions. (high false positives, better able to deal with new attacks)

Continued…

10. Paging and OS/161 [20 points]

Modern operating systems use a separate *paging* thread to select pages for eviction and maintain a pool of free physical memory pages. The main disadvantage of this approach is that pages may be freed while they are still needed, increasing the number of page faults.

(i) [2 points] What is the main advantage of using such a paging thread?

Page fault handling can be much faster because we do not incur the overhead of running our page replacement algorithm to select a victim, and the overhead of writing dirty victims to disk on every page fault. We can amortize this cost across a large number of evictions, and we can do it in the background when no process is waiting for the eviction to complete before it can make progress

(ii) [2 points] How does a modern OS reduce the cost of the extra page faults caused by freeing physical memory pages early?

If a process faults on a freed page, and the page frame has not been reallocated to a different process, then the same page frame is reallocated to the original owner avoiding the cost of reading the page contents back in from disk.

Suppose a paging thread were added to OS/161. Describe the following in words, making specific reference to existing OS/161 data structures:

(iii) [4 points] What changes to existing data structures are needed to allow a paging thread to free physical pages, while supporting the standard OS technique for reducing the cost of extra page faults? *For full marks, do not increase the size of existing structures, or add any new data structures.*

Mainly, we need a way to distinguish a truly invalid virtual page (one that requires a new page to be allocated and data read in from disk) from one that has been freed, but could still be "rescued" or reclaimed. We can borrow another bit from the low-order part of the lp_paddr field in the lpage structure, similar to the lpage lock and dirty bits, e.g. #define LP_FREED 0x4 Nothing else is needed, although we will need to change how we update/modify the existing fields as well.

This question continues on the following page.

For each of the following, focus on how data structures are updated and ignore synchronization concerns.

(iv) [4 points] What happens when the paging thread frees a physical memory page?

We need to set the LP_FREED flag in the lp_paddr field, but NOT set the lp_paddr to INVALID_PADDR (the current behavior). We also need to mark the page free in the coremap, but NOT clear the pointer to the lpage in the coremap entry (because if the page is allocated to a different process or virtual page, we will need to update the lpage again, marking it truly invalid this time). We can interpret a coremap entry marked free with a non-NULL lpage as a freed page that might still be needed by its previous owner.

(v) [4 points] What happens when a freed page is reallocated to hold a different virtual page?

If we allocate a freed page with a non-NULL lpage pointer, we must mark the lp_paddr field in the lpage as INVALID_PADDR, so that a subsequent fault on the lpage will get a fresh page frame and read the missing data in from disk.

(vi) [4 points] What happens when a process faults on a virtual page and the physical page to which it was previously mapped has been freed by the paging thread but not yet reallocated?

If the lp_paddr field has the LP_FREED bit marked, we can use the rest of the lp_paddr field to locate the coremap entry for the page frame and mark that frame as allocated again. We then clear the LP_FREED bit and proceed with updating the TLB as if we had faulted on a page that was valid.

Continued…

Extra Space – Use if needed. Indicate clearly what questions you are answering here.

Bonus: 2 marks – Hand in your info sheet (make sure your name and student number is on it) with your final exam.

End of Exam
Total Pages = (16)
Total Marks = (138) (not counting bonus)