PLEASE HAND IN

PLEASE HAND IN

UNIVERSITY OF TORONTO
Faculty of Arts and Science

St. George Campus

April 2012 EXAMINATIONS

CSC 369H1S
Instructor — Angela Demke Brown
Duration — 3 hours

Examination Aids: One double-sided 8.5x11 sheet of paper. No electronic aids.

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

Last (Family) Name(s): _____

First (Given) Name(s): _____

---

*Do **not** turn this page until you have received the signal to start.*
(In the meantime, please fill out the identification section above, and read the instructions below *carefully*.)

This final examination consists of **??** question on **??** page (including this one). *When you receive the signal to start, please make sure that your copy of the examination is complete.*

If you need more space for one of your solutions, use the last pages of the exam and indicate clearly the part of your work that should be marked.

In your written answers, be as specific as possible and explain your reasoning. Clear, concise answers will be given higher marks than vague, wordy answers. **Marks will be deducted for incorrect statements in an answer.** Please make your handwriting legible!

MARKING GUIDE

??	# 0: _____/??
	TOTAL: _____/??

Question 1. True / False [12 MARKS]

[TRUE]	FALSE	Operating systems rely on hardware support for memory protection.
TRUE	[FALSE]	Operating system 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 at least as many disk accesses as 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	Public key cryptography algorithms are much slower than private key algorithms.
[TRUE]	FALSE	Good security protects a system against accidental attacks as well as intentional ones.

Question 2. Fun With Acronyms [15 MARKS]

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**.

Part (a) [3 MARKS] MMU

- Memory Management Unit
- Hardware device that performs memory access permission checks and virtual to physical address translation
- Mechanism

Part (b) [3 MARKS] FCFS

- First Come First Served
- A scheduling policy in which jobs are scheduled in the order that they arrive
- Policy

Part (c) [3 MARKS] PCB

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

Part (d) [3 MARKS] 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

Part (e) [3 MARKS] LRU

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

Question 3. Fill in the Blanks [15 MARKS]

Part (a) [3 MARKS] 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 or features, indicate whether the task will complete faster or if the feature is easier to provide 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 them all.

U Creating new tasks (faster overall completion)

P Preventing accidental modification of other tasks' private memory (e.g. stack) (easier to provide feature)

K,P Making blocking system calls (faster overall completion) (kernel threads are probably better *overall*, but either is ok)

U Switching between tasks (faster overall completion)

K Solving a parallel problem with fine-grained synchronization on a multiprocessor (faster overall completion)

U Customizing task scheduling (easier to provide feature)

Part (b) [3 MARKS] 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**

This question continues on the following page

Part (c) [3 MARKS] 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

I page eviction

F Context switch

N System call other than fork or exec

F fork, implemented with copy-on-write

I copy-on-write fault

Part (d) [3 MARKS] 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

Part (e) [3 MARKS] 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

Question 4. Synchronization Problem [12 MARKS]

In one form of message passing, 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.

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. The definition of the mailbox structure is shown below.

```
struct mailbox {
        char *dest_buf; /* set by receiver */
        int sender_id;  /* set by sender */
        struct semaphore *recvr_ready; /* Initially, count = 0 */
        struct semaphore *sender_done; /* Initially, count = 0 */
};

struct mailbox mailboxes[TID_MAX];
```

Part (a) [8 MARKS] 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.]*

Part (b) [2 MARKS] 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.

Part (c) [2 MARKS] 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.

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];


        /* SOLN: Wait for the receiver to indicate it is ready */
        P(mbox->recvr_ready);


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







        mbox->sender_id = get_current_thread_id();

        /* SOLN: Signal that the sender if finished sending the message */
        V(mbox->sender_done);



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







        mbox->dest_buf = msg;

        /* SOLN: After setting dest_buf, signal ready to get msg */
        V(mbox->recvr_ready);

        /* SOLN: Wait for sender to indicate message is available */
        P(mbox->sender_done);

        *sender_id = mbox->sender_id;




}
```

Question 5. Scheduling [10 MARKS]

Part (a) [3 MARKS] **Explain** the difference between a preemptive scheduler and a non-preemptive scheduler, and **give one example for each** of the type of system where it 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 (e.g., some embedded systems) ... some real time systems are also non-preemptive, relying on cooperative scheduling to meet deadlines.

Part (b) [2 MARKS] The OS/161 scheduler runqueue is a linked list of pointers to thread structs. What would be the effect on scheduling if we put two pointers to the same thread struct into the runqueue?

A thread gets scheduled whenever its struct reaches the head of the runqueue, so this gives such threads twice as much CPU time as threads that are only in the runqueue once. (Sort of proportional share scheduling.)

Part (c) [2 MARKS] What new problem could occur if we put two pointers to the same thread struct into the runqueue?

A thread could be scheduled to run while it is in the blocked state, because the second pointer to the thread struct gets to the head of the runqueue while the first is on some wait channel. [We would need to be careful to find and remove the second pointer from the runqueue when a thread blocks, or check the thread state when it is selected to run to make sure it is not blocked.]

Part (d) [3 MARKS] **Briefly describe** how you could achieve the same effect without the duplicate pointers.

Record the size of the scheduling quantum for each thread, so that some threads can have more CPU time (same effect as putting the pointer in the queue twice) and on timer interrupt, check if the current thread has used up its quantum, rather than giving all threads the same size of quantum.

Question 6. Page Replacement [8 MARKS]

Two processes, P1 and P2 are executing on a system with 8 pages of physical memory. The OS attempts to maintain a pool of free page frames. The page replacement algorithm will be triggered when the number of free frames drops below the low water mark (1 frame) and it will reclaim pages until the number of free frames reaches the high water mark (**3 frames**). The last free frame has just been allocated.

P1 page table:

Virt	Valid bit			Ref bit			Frame
Pg	(a)	(b)	(c)	(a)	(b)	(c)	(a)
0	1		**0**	1	**0**	**0**	3
1	1			1	**0**	**0**	7
2	1			1	**0**	**0**	4
3	1			1	**0**	**0**	6
4	0			0			—
5	1	**0**	**0**	0			0

P2 page table:

Virt	Valid bit			Ref bit			Frame
Pg	(a)	(b)	(c)	(a)	(b)	(c)	(a)
0	0			0			—
1	1			1	**0**	**0**	2
2	1	**0**	**0**	0			5
3	0			0			—
4	1	**0**		1	**0**	**0**	1
5	0			0			—

Coremap:

Page	In Use			PID	Virt Pg
Frame	(a)	(b)	(c)	(a)	(a)
0	1	**0** vic 1	**0** vic 1	**1**	**5**
1	1	**0** vic 3		**2**	**4**
2	1			**2**	**1**
3	1		**0** vic 3	**1**	**0**
4	1			**1**	**2**
5	1	**0** vic 2	**0** vic 2	**2**	**2**
6	1			**1**	**3**
7	1			**1**	**1**

Part (a) [2 MARKS] Fill in the initial coremap data using the page tables for P1 and P2.

Part (b) [3 MARKS] Show the changes that would result (by filling in column (b) in the tables) if the global clock replacement algorithm is used, and the clock hand is initially pointing at Frame 0.

Part (c) [3 MARKS] Consider the following modification to the global clock algorithm: select victim page frames such that each process loses pages proportional to the amount of memory it is currently allocated. For example, if a process is using half of the total physical memory, then half of the victim frames should come from that process (fractions are rounded to the nearest integer number of pages). **Show the changes that would result (by filling in column(c) in the tables above) using clock with this modification.** Start with the initial state and assume the clock hand begins at Frame 0 again.

For (c) How many pages should be stolen from each process? P1: $3 * 5/7 = 15/7 = 2$
P2: $3 * 3/7 = 9/7 = 1$

Question 7. File Systems [9 MARKS]

Consider a Unix-like file system that maintains a unique index node for each file in the system. Each index node includes 6 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.

Part (a) [3 MARKS] Write an expression for the maximum file size that can be supported by this index node, in terms of **B** and **P**

- *6 direct data blocks, each of size* $B \rightarrow 6B$
- *1 indirect block of size* B*, holding* B/P *pointers to data blocks of size* $B \rightarrow B * B/P$
- *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

$$max\ file\ size = 6B + B\frac{B}{P} + B\frac{B}{P}\frac{B}{P} = B(6 + \frac{B}{P} + \frac{B^2}{P^2})$$

Part (b) [3 MARKS] How many disk operations will be required if a process reads data from the N^{th} 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**. *Hint: Your answer can include multiple expressions to cover different values of N.*

$$\#\ reads = \begin{cases} 1\ if\ N \leq 6 \\ 2\ if\ 6 < N \leq 6 + \frac{B}{P} \\ 3\ if\ N > 6 + \frac{B}{P} \end{cases}$$

Part (c) [3 MARKS] In Unix systems, you are not allowed to create a hard link where the source file is on one file system, and the destination file is on a different file system. **Explain why not**.

A hard link records the inode # of the file being linked to. But each file system has its own inodes (and hence inode numbers), so the inode number alone is not sufficient to identify the file that is being linked to. For example, inode 42 on filesystem1 could be a different file than inode 42 on filesystem 2.

Question 8. Security [6 MARKS]

Part (a) [2 MARKS] Methods for authenticating users when they attempt to log in are based on one of three general principles. **List any two, and give a specific example for each of them.**

- *Something the user **knows**. For example, a password.*
- *Something the user **has**. For example, a swipecard or physical keys.*
- *Something the user **is**. For example, biometrics such as iris, finger, or voice prints.*

Part (b) [4 MARKS] **Explain** the purpose of the random values sent in the ClientHello and ServerHello messages of the TLS Handshake Protocol. **Describe** an attack that is possible during a session resume handshake if the client random number is omitted.

*The random values ensure **liveness** during an exchange. The client supplies a random number to be used in the key generation, so that it knows it is talking to a "live" server, and not simply receiving replayed messages from a previous exchange. Similarly, the server supplies a random number to be used in the key generation, so that it knows it is talking to a "live" client, and not receiving replayed client messages that were recorded by an attacker during an earlier exchange. If the client random were omitted during a session resume handshake, then an attacker can replay previously recorded server messages to resume a session with a client, which believes it is talking to the original server. The attacker can then replay server messages to cause unintended effects, such as acknowledging a request that has not really been handled.*

Note that the attacker does not gain knowledge of the keys in this way, and does not know the content of the messages, but can still cause harm.

Question 9. OS/161 Design: User-level Synchronization [14 MARKS]

Assume that OS/161 has been extended with support for a simple form of user-level shared memory which allows a user process to share parts of its address space with its children. The details of the shared memory support are not important, but processes sharing memory will need a way to coordinate their accesses to shared data. **Your job is to design a user-level lock package that processes can use for this purpose.**

Part (a) [2 MARKS] One option is to build the `lock_acquire` and `lock_release` functions entirely at user-level but this will require busy waiting in the `lock_acquire` function if the lock is already held. **Explain** why some form of busy waiting is needed in this case, and **briefly explain** why busy waiting is undesirable in a uniprocessor system like the one that OS/161 runs on.

A process has to busy wait in lock_acquire because we have no way of suspending the process on a wait queue (or wait channel) from user-level. Busy waiting is bad in a uniprocessor because only one process can be executing at any time, and if a waiter is spinning, there is no way for the lock holder to make progress and release the lock.

A second option is to use system calls to request OS services to assist with user-level lock operations. Four system calls are needed, as listed in the box below:

```
int lock_init(int *lock_id);     /* returns 0 on success, and sets lock_id for
                                  * use in subsequent operations, or returns -1
                                  * on failure with errno set to a suitable value.
                                  */

int lock_acquire(int lock_id);   /* returns 0 when the lock specified by lock_id
                                  * has been acquired by the calling process, or
                                  * -1 on failure with errno set appropriately
                                  * (the lock is not acquired in case of failure).
                                  */

int lock_release(int lock_id);   /* releases the lock specified by lock_id and
                                  * returns 0 on success or -1 on failure with
                                  * errno set appropriately.
                                  */

int lock_destroy(int lock_id);   /* destroys the lock specified by lock_id and
                                  * cleans up any state maintained by the OS,
                                  * returns 0 on success or -1 on failure.
                                  */
```


This question continues on the following page

For each of the following questions, pseudo-code may help to present your answer more clearly, but a point-form English description is also acceptable. You may invent new error codes for this question, as long as they describe errors that might reasonably occur.

Part (b) [3 MARKS] **Describe** the operating system data structure(s) that you would use to keep track of lock ids. Make sure you describe (i) what is stored in the data structure, (ii) how it is accessed, and (iii) where this data structure is located.

You can use an array, similar to the open file table that was used to track file descriptors. The array stores pointers to "struct lock"s, i.e. the existing OS161 lock data structure. The lock_id is the index into the array. We can have either have a global table, which is accessed through a global variable and is shared by all processes, or a per-process table which is accessed through the "struct thread". [Per-process means unrelated threads can't interfere with the locking for a group of parent/child processes, but we have to make sure parent/child share the table after fork.]

Part (c) [3 MARKS] **Outline** an implementation of the system call handler for the `lock_init` call (i.e., `sys_lock_init`). Identify one specific error that might reasonably be returned by `lock_init`.

- *Lock lock_table (either global or current process's table, which might be shared with children)*
- *Search lock_table for an unused entry, e. Return ENOLOCKS if all entries in lock table are in use.*
- *Create OS161 kernel lock. Return ENOMEM if create fails (or return error code returned by lock_create).*
- *Set pointer to new lock, e.g. lock_table[e] = new lock*
- *Unlock lock_table*
- *Return e as result of system call (e.g. *retval = e)*

NOTE: Creating a lock can be omitted if the table is initialized by creating all the possible locks up front, but that is a terrible waste of memory and time.

This question continues on the following page

Part (d) [3 MARKS] **Outline** an implementation of the system call handler for the `lock_acquire` call (i.e. `sys_lock_acquire`). Identify one specific error that might reasonably be returned by `lock_acquire`.

This simply needs to acquire the kernel lock in the lock table at lock_id if one exists, or else return an error indicating that the given lock_id is not valid.

```
if (lock_id < MAX_LOCK_ID && lock_table[lock_id] != NULL) {
    lock_acquire(lock_table[lock_id]);
} else {
    return EINVAL; /* Invented error code like EBADLOCK also ok */
}
```

Part (e) [3 MARKS] **Outline** an implementation of the system call handler for the `lock_release` call (i.e. `sys_lock_release`). Identify one specific error that might reasonably be returned by `lock_release` but **would not** be returned by `lock_acquire`.

This also just needs to check that the lock_id is valid and then call lock_release on the kernel lock stored in the lock table entry for lock_id. The unique error is trying to release a lock that is not held by the caller.

```
if (lock_id < MAX_LOCK_ID && lock_table[lock_id] != NULL) {
    if (lock_do_i_hold(lock_table[lock_id]) {
        lock_release(lock_table[lock_id]);
    } else {
        return EINVAL; /* Invented error code like ENOTOWNER also ok */
    }
} else {
    return EINVAL;  /* Invented error code like EBADLOCK also ok */
}
```