(VERSION WITH ANSWERS)
CORE EXAMINATION
Department of Computer Science
New York University
September 26, 2008

This is the common examination for the M.S. program in CS. It covers three core computer science topics: Programming Languages, Operating Systems, and Algorithms. The exam has two parts. The Systems Part (PL and OS) lasts two and one half hours and covers the first two topics. The Algorithms Part (Algo), given this afternoon, lasts one and one-half hours, and covers the last topic.

You will be assigned a seat in the examination room.

Use the proper booklet or answer sheet for each question. Each booklet is marked with the Area and Question number, in the form PL1, PL2, OS1, OS2, ALG1, ALG2 and ALG3. DO NOT put your name on the exam booklet or answer sheet. Instead, your exam number must be on every booklet.

You will be graded according to your exam number, shown on the envelope containing the booklets. Remember your exam number: when grades are given out, they will be published according to this number, not by name.

Make sure your name and signature are on the envelope. This is the only place where your name appears. Please include all the booklets inside the envelope. You can keep the exam.

Good luck!

SYSTEMS PART:
Programming Languages and Operating Systems

Question 1—Please use the Exam Booklet labeled PL1

The sum operator, $\sum$, can be thought of as an operator that is parameterized by a starting value, a terminating value, and a function. For example, the mathematical expression $\sum_{i=n}^{m} f(i)$ can be thought of as the sum operator applied to n, m, and f.

1. In the language of your choice, write the $\sum$ operator.

Answer: ML:

```
fun sum n m f = if m <= n then 0
                else f n + sum (n+1) m f
```

Scheme:

```
(define (sum n m f)
   (cond ((<= m n) 0)
         (else (+ (f n) (sum (+ n 1) m f)))
         ))
```

□

2. If you were to implement the product operator, $\prod$ (as used, for example, in the mathematical expression $\prod_{i=n}^{m} f(i)$), much of the code would be the same as the code in $\sum$. Write a function `gen_op` (again, in the language of your choice – it doesn't have to be the same language as in the previous part) such that a call to `gen_op` could return the $\sum$ operator and another call to `gen_op` could return the $\prod$ operator (depending on the parameter(s) to `gen_op`). Also, write the calls to `gen_op` that would return $\sum$ and $\prod$.

Answer: Scheme:

```
(define (gen_op op base)
  (letrec ((iter (lambda (n m f)
                 (cond ((<= m n) base)
                       (else (op (f n) (iter (+ n 1) m f)))
                       ))) )
    iter))
```

ML (curried version):

```
fun gen_op oper base n m f =
    if m <= n then base
    else oper(f n, gen_op oper base (n+1) m f)

val sum = gen_op op+ 0

val prod = gen_op op* 1
```

ML (non-curried version)

```
fun gen_op op base =
    let fun iter n m f =
          if m <= n then base
          else oper(f n, iter (n+1) m f)
```

Instead of `op+` and `op*`, the expressions `(fn (x,y) => x+y)` and `(fn(x,y) => x*y)` can be used, respectively. □

Question 2—Please use the Exam Booklet labeled PL2

Programming Languages: Static and Dynamic Scope

Suppose you have two C compilers, C_S and C_D. C_S compiles according to the standard C semantics. C_D is an experimental compiler that takes C syntax as input, but uses non-standard semantics: it assumes all variables have *dynamic* scope instead of *static* scope.

Consider the following program using C syntax. Assume there is a function called `getValue` that returns an integer:

```
int getValue(void);
void printList(int n);

main()
{
  int n;
  n = getValue();
  printList(n);
}
```

Write code for the function `printList` such that:

- When compiled with C_S, it prints the integers from 1 to n in order.
- When compiled with C_D, it prints the integers from n down to 1 (i.e. in reverse order).

You may add extra code (including additional variables and functions) as needed.

Answer:

```
int start;
int end;
int inc;

int getValue(void);

void doPrint()
{
  int i;
  for (i = start; i != end; i += inc) {
    printf("%d ", i);
  }
```

```
  printf("\n");
}

void printList(int n)
{
  int start = n;
  int end = 0;
  int inc = -1;
  doPrint();
}

main()
{
  int n;
  start = 1;
  end = n+1;
  inc = 1;
  n = getValue();
  printList(n);
}
```

Question 3—Please use the Exam Booklet labeled OS1

Process Coordination and I/O

Consider the following three processes P1, P2, and P3 written in pseudocode.

```
Process P1          Process P2          Process P3

P(S)                P(T)                P(U)
Compute(10)         Compute(20)         Compute(30)
Disk(10,100)        Disk(100,10)        Disk(1,1000)
V(T)                V(U)                V(S)
```

P(X) and V(X) are binary semaphores that do not busy-wait. Specifically: If the (binary) semaphore X is open, V(X) is a nop and P(X) closes X. If X is closed, P(X) blocks the process and V(X) opens X (thus allowing **one** process blocked on X to proceed and re-close X).

When execution begins the U semaphore is open and the S and T semaphores are closed.

We make several simplifying assumptions

1. Context switching takes **zero** CPU time.
2. Disk(M,N) takes **zero** CPU time.
3. P(X) and V(X) take **zero** CPU time.
4. Compute(Y) takes Y ms of CPU time

Thus we see that the processes require 10ms, 20ms, and 30ms of CPU time and hence the entire job requires 60ms of CPU time. (Recall that ms abbreviates millisecond and 1ms=10^{-3}seconds.)

Another simplification is that this system uses a **non**-preemptive scheduler. So a process is **NOT** involuntarily moved from the running to the ready state. Of course it can be blocked. Two important reasons why a process would be blocked are

1. The process initiates I/O or
2. The process executes P(X) on a closed semaphore X.

Assume that at time zero, P1, P2, and P3 are all created and are the only processes in the system.

Part A (5 Points)

For Part A assume that Disk(M,N) is a nop. When does each of the three Compute(Y) procedures start and when do they end? When does each of the three processes terminate?

Answer

1. Initially, only semaphore U is open so only process P3 can run;
2. P(U) takes 0ms; Compute(30) begins at 0ms.
3. Compute(30) ends at 30ms.
4. Disk(1,1000) and V(S) take 0ms; P3 terminates at 30ms.
5. Only S is open; P(S) takes 0ms; Compute(10) begins at 30ms.

6. Compute(10) ends at 40ms.
7. Disk(10,100) and V(T) take 0ms; P1 terminates at 40ms;
8. Only T is open; P(T) takes 0ms; Compute(20) begins at 40ms.
9. Compute(20) ends at 60ms.
10. Disk(100,10) and V(U) take 0ms; P2 terminates at 60ms.

Part B (4 Points)

For Part B, Disk(M,N) is no longer a nop, but it still is assumed to require zero CPU time. Specifically, Disk(M,N) performs M disk read operations. These M reads are directed at disk locations that are far apart from each other; however, note that each single read itself is directed at a contiguous disk block of N KB (recall that KB abbreviates Kilobyte and 1KB=10^3Bytes). Note that each of the processes reads a total of 1MB (recall that MB abbreviates megabyte and that 1MB=10^6Bytes).

To determine the I/O time required for Disk(M,N) assume all M reads are directed at a single disk with the following characteristics

1. The block size is N.
2. The seek time is 5ms.
3. The rotation rate is 6000 RPM (RPM abbreviates revolutions per minute).
4. The transfer rate is 10MB/second (after the head has reached the beginning of the block).

What is the total I/O time required for Disk(1,100), Disk(1,10), and Disk(1,1000), i.e. for **ONE** I/O operation of each of the three sizes.

Answer

6000 Revolutions per minute = 100 revolutions per second. So one revolution = 1/100 seconds = 10ms. The (average) rotational latency is 1/2 a revolution or 5ms.

Thus each of the operations has a seek of 5ms and a rotational latency of 5ms.

The transfer time for an n KB operation is n KB / (10MB/sec).

For Disk(1,100): each transfer takes 100KB / (10MB/sec) = 10ms,
so the total time for Disk(1,100) is
seek + rot latency + transfer = 5ms+5ms+10ms = 20ms.

For Disk(1,10): each transfer takes 10KB / (10MB/sec) = 1ms,
so the total time for Disk(1,10) is
seek + rot latency + transfer = 5ms+5ms+1ms = 11ms.

For Disk(1,1000): each transfer takes 1000KB / (10MB/sec) = 100ms,
so the total time for Disk(1,1000) is
seek + rot latency + transfer = 5ms+5ms+100ms = 110ms.

Part C (1 Point)

Using the assumptions of part B and the processes of part A, when does each of the three Compute(Y) procedures start and end; when does each of the three Disk(M,N) procedures start and end; and when does each process terminate?

Answer

Now we repeat the analysis of part A plugging in the values for Disk(N,M) that we just computed.

1. Initially, only semaphore U is open so only process P3 can run;
2. P(U) takes 0ms; Compute(30) begins at 0ms.
3. Compute(30) ends at 30ms.
4. Disk(1,1000) starts at 30ms.
5. Disk(1,1000) takes 110ms (Part B) so ends at 140ms.
6. V(S) takes 0ms; P3 terminates at 140ms.
7. Only S is open so P1 runs; P(S) takes 0ms; Compute(10) begins at 140ms.
8. Compute(10) ends at 150ms.
9. Disk(10,100) starts at 150ms.
10. Disk(10,100) takes $10 \times 20ms = 200ms$ so ends at 350ms.
11. V(T) take 0ms; P1 terminates at 350ms;
12. Only T is open so P2 runs; P(T) takes 0ms; Compute(20) begins at 350ms.
13. Compute(20) ends at 370ms.
14. Disk(100,10) starts at 370ms.
15. Disk(100,10) takes $100 \times 11ms = 1100ms$ so ends at 1470ms.
16. V(U) take 0ms; P2 terminates at 1470ms.

Question 4—Please use the Exam Booklet labeled OS2

Virtual Memory:

A user process executes a program that begins as follows:

```
int  fd;                    // file descriptor
char buf[4];                // buffer to hold data from the file

fd = Open("fileX", "r");    // open fileX for reading
Read(fd, buf, 4);           // read 4 bytes from fileX into buf
```

`Open` and `Read` are system calls for opening a file, and reading information from it respectively. Recall that system calls in most modern operating systems (OS) are implemented using a trap (exception) mechanism, which transfers control to the operating system. For the `Read` system call, the OS code retrieves the call arguments, performs the requested operation, and passes back the response by writing into memory locations that are accessible by the user process.

When the OS handles the exception caused by the `Read` system call above, it receives the parameter values **3**, **998**, and **4**, corresponding to the parameters `fd`, `buf` (the address of the buffer), and the number of bytes that need to be read. Assume that the first four bytes of the file `fileX` are the characters `a`, `b`, `c`, and `d` (in that order). The virtual memory system uses a page size of **1000** bytes.

At the time of the `Read` call, the page table for the user process looks like this:

Page Number	*Frame Number*	*Valid*	*Use*	*Dirty*
0	3	1	1	0
1	1	1	0	0
2	0	1	1	1
3	0	0	0	0
4	2	1	0	1

Note that the *Use* and *Dirty* bits (also called the *Referenced* and *Modified* bits) keep track of page usage. These bits are set by hardware and unset in specific ways by page replacement algorithms to control which pages to evict.

Part A (3 points)

The page table above controls how virtual addresses generated by the user process are translated into physical addresses.

To which **physical addresses** will the OS write the characters `a`, `b`, `c`, and `d`, so that they will be placed properly in the process' buffer when the `Read` call returns?

Answer:

Note that the characters `a`, `b`, `c`, and `d` need to be written to the locations corresponding to `buf[0]`, `buf[1]`, `buf[2]` and `buf[3]` in the process' virtual address space. These locations correspond to virtual addresses **998**, **999**, **1000**, and **1001** respectively.

Of these, addresses **998** and **999** belong to page 0 and addresses **1000** and **1001** to page 1. Since page 0 maps to frame 3 and page 1 to frame 1, the physical addresses corresponding to these virtual addresses are:

Virtual Address	Physical Address
998	$(3 \times 1000 + 998) = 3998$
999	$(3 \times 1000 + 999) = 3999$
1000	$(1 \times 1000 + 0) = 1000$
1001	$(1 \times 1000 + 1) = 1001$

Part B (3 points)

For the system as in question A above, show the state of the process page table **after** the `Read` call completes.

(Hint: Consider which fields of the page table entry would change as a result of the `Read` system call).

Answer:

The process table after the completion of the `Read` call differs from that shown as part of question A only in the settings of the *Use* and *Dirty* bits associated with the pages hosting the locations of `buf`: (the changes are shown highlighted in bold)

Page Number	*Frame Number*	*Valid*	*Use*	*Dirty*
0	3	1	1	**1**
1	1	1	**1**	**1**
2	0	1	1	1
3	0	0	0	0
4	2	1	0	1

Part C (4 points)

Consider the system and process in question A above, where the virtual memory system employs a local page replacement policy using the CLOCK (Second Chance) algorithm.

The process is statically allocated four physical frames 0-3 (as shown in the first page table): page replacement is restricted to one of these frames.

Assume that the circular list used by the CLOCK algorithm consists of frames inspected in increasing numeric order, i.e., in the order 0, 1, 2, 3. The CLOCK pointer currently points at frame 0.

Starting from the process page table after the `Read` call (which you computed as your answer to question B), give a sequence of three virtual address references in the range 0 - 4999, such that if these are the next three addresses referenced (used) by the process, each reference will cause a **page fault**. Please show all your work.

Answer:

Recall that the CLOCK (Second Chance) algorithm maintains a circular list of frames allocated to the process. When a request for a non-resident page is encountered, the algorithm starts inspecting frames beginning from the one currently pointed to by the CLOCK pointer. If the frame is currently not in use or it's *Use* (reference) bit is 0, that frame is picked as the container for the page. Else, the frame's *Use* bit is reset to 0 and the CLOCK pointer is advanced by one. This last feature of giving used pages at least one additional chance before being evicted, is what gives the algorithm its 'Second Chance' name.

Given the description above, to completely answer the question for full credit, we need to identify both a virtual address reference that will trigger a page fault and describe the functioning of the CLOCK algorithm in sufficient detail to justify which frame is picked by the page replacement algorithm to satisfy the fault.

We start with the frames and their use bits in the following state (all frames are initially in use):

CLOCK pointer	*Frame Number*	*Use*	*Page Number*
*	0	1	2
	1	1	1
	2	0	4
	3	1	0

Our first reference clearly needs to be to a non-resident page to trigger a page fault. Therefore, this address needs to be on page 3, say address **3001**. To service this fault, we run the CLOCK algorithm starting from frame 0 to find a frame for holding this page. The algorithm stops at frame 2, the first frame to not have its *Use* bit set. Page 4, the current occupant of frame 2, is evicted, and the frame is assigned to the page of the faulting access - page 3. At this point, the CLOCK pointer stops at the next frame, frame 3, waiting for the next eviction request. The state of the frames and their use bits looks like the following: (note the resetting of the *Use* bits for frames 0 and 1)

CLOCK pointer	*Frame Number*	*Use*	*Page Number*
	0	0	2
	1	0	1
	2	1	3
*	3	1	0

Following the same reasoning as above, the next reference needs to be to an address on page 4 (the only non-resident page) to trigger a page fault, say address **4001**. The operation of the CLOCK algorithm picks frame 0 as the victim, evicting its occupant — page 2 — and stops at the next frame (frame 1). Note that we make the assumption that there has been no process activity in the interim outside our control that sets the *Use* bits in the table. The state of the frames after this step is the following:

CLOCK pointer	*Frame Number*	*Use*	*Page Number*
	0	1	4
*	1	0	1
	2	1	3
	3	0	0

The third reference therefore needs to be to a location on page 2 (the only non-resident page), say address **2001**.