CORE EXAMINATION
Department of Computer Science
New York University
February 4, 2005

PL&C: Question 1

Suppose you are building a C compiler and the calling convention is such that, when a procedure is called, the new stack frame should look like (assuming the stack grows down):

```
paramN
...
param2
param1
return address
dynamic link     <------- Frame Pointer
local1
local2
...
localN           <------- Stack Pointer
```

(Note: This is typically what C compilers on the x86 architecture do)

Suppose your compiler is compiling two procedures, as follows:

```
void foo(int x, int y, int z)
{
  int a, b;
  a = x+y;
  b = a+z;
}


void bar()
{
  foo(3,4,5);
}
```

A. Write the assembly code (x86 if possible, if not, any assembly code you like - the syntax doesn't have to be perfect) that your compiler might generate for the entire foo() procedure. Assume that your compiler doesn't do any optimization.

Answer: Here is x86 assembly code in AT&T (Unix) style:

```
_foo:
pushl %ebp         ;; save old frame pointer (dynamic link)
movl %esp, %ebp   ;; set frame pointer to current stack pointer
subl $8, %esp     ;; allocate room on stack for a and b
movl 12(%ebp), %eax  ;; load y
```

```
addl 8(%ebp), %eax   ;;  add x
movl %eax, -4(%ebp)  ;; store in a
movl 16(%ebp), %eax  ;; load z
addl -4(%ebp), %eax  ;; add a
movl %eax, -8(%ebp)  ;; store in b
        addl    $8, %esp        ;; remove a and b from stack
popl    %ebp            ;; restore old frame pointer
ret                     ;; return from procedure
```

B. Write the assembly code for the procedure call to foo() in bar().

Answer:

```
pushl   $5        ; push third argument
        pushl   $4        ; push second argument
        pushl   $3        ; push first argument
call _foo      ; call foo
addl    $12,esp   ; remove the arguments
```

C. On x86 machines, the ebx, esi, edi and ebp registers are generally "callee save", while eax, ecx and edx are typically "caller save". What does this mean?

Answer:
A "callee save" register is a register that, when a procedure call occurs, the called procedure is responsible for making sure that the contents of the register upon exit from the procedure is the same as it was upon entry to the procedure. That is, if the called procedure wants to overwrite the register, it must save the contents (e.g. on the stack) before it does so, and then restore the contents prior to returning.

A "caller save" register is a register that, if the calling procedure wants the contents of the register to be preserved during the function call, the calling procedure has to save the register before the call and restore the register after the call completes. The called function is free to overwrite a "caller save" register as it likes.

D. In order to demonstrate the meaning of "callee save" and "caller save", add code to the C example above (i.e. foo() and bar()) that would require your compiler to save registers. Then, write out the assembly code corresponding to the C code you've added.

Answer:

```
int foo(int x, int y, int z)
{
  int a, b;
  a = x+y;
  b = a+z;
  return b;         // adding a return value here
}
```

```
int bar()
{
  return foo(3,4,5) + foo(6,7,8) + 9;   // a large expression here
}
```

Your compiler might try to keep the value of the large expression in bar() in a register as it is being computed. Depending on which register it chooses to hold the value being computed, that register may have to be saved prior to a call to foo(). For example, the code for computing the large expression in bar() might be:

```
pushl   $5        ; push third argument
        pushl   $4        ; push second argument
        pushl   $3        ; push first argument
call _foo      ; call foo -  result returned in eax
addl    $12,esp   ; remove the arguments

movl eax,ecx   ; copy returned value to ecx
pushl ecx   ; SAVE ecx (caller-saved register)

pushl   $6        ; push third argument
        pushl   $7        ; push second argument
        pushl   $8        ; push first argument
call _foo      ; call foo - result returned in eax
addl    $12,esp   ; remove the arguments

popl    ecx       ; RESTORE ecx
        addl    eax, ecx  ; add the return value of the second call

        addl    $9, ecx   ; add 9
mov ecx, eax  ; move the sum to eax in order to return it
```

PL&C: Question 2

A. Describe briefly the purpose of an iterator over a collection.

Answer: An iterator is an object that encapsulates the process of traversing a collection. The state of an iterator indicates the current element of the collection that is being examined. Because the iterator is an object, it can be stored and passed as parameter, which allows the traversal to be resumed, or continued by another operation at a later time.

B. What are the basic operations that a useful iterator must provide?

Answer: An iterator is linked to a given collection, and therefore the constructor for an iterator must designate the collection over which the iterator will operate. In addition, we need at least a way of advancing the iterator to the next element in a collection, and a predicate to determine whether there are any elements left. So the minimal set is the equivalent of:

```
Init ( );
Next ();
Previous ();
Done ();
```

If the collection is ordered, it might make sense to define an insertion operation (insert new element after the one currently designated by the iterator), deletion, etc. Finally, we need to get from the iterator to the element that it currently designates (in C++ this is often a dereference).

C. Suppose that we need to manipulate a set collection, that is to say an unordered collection without duplicates. Operations on a set include membership, element insertion, element deletion, and equality.

We need to provide different representations for these collections (bit vectors, hash-tables, linked lists, trees, etc.). We want to provide a general-purpose equality routine that takes two sets of any two representations, and returns True if the two sets contain the same elements. In the language of your choice, explain how you would organize your types or classes so that you have to write this equality routine only once, regardless of how many different representations are at hand.

Answer: Given that our set collection can have different representations, we want to define an interface (or an abstract class) that defines the required operations. Any explicit representation must implement this interface. In Java, this might be:

```
interface Seti {
    boolean Member (Object X);
    void Insert (Object X);
    Seti Union (Seti S);
    int Size ();
}
```

Similarly, iterators over these various collections must implement a common interface:

```
interface Iterator {
```

```
    void Init ( );
    Object Next ( );
    boolean Done ( );
    Object Elem ( );
};
```

Given that the code for an iterator depends on the internal structure of the collection, for each implementation of a collection we declare an inner class that implements the Iterator interface, and an operation newIterator that constructs an instance of that class.

D. In the language of your choice, using iterators, write the body of the equality routine. (You may assume that the iterators exist; you do not have to write them. Also, it is not important that the equality routine be efficient, just that it be correct.)

Answer: The body of the equality routine can be written as follows:

```
Boolean Equals (Seti S) {
   if (Size() != S.Size()) return False;
   iterator I1 = S.newIterator();
   I1.Init( );
   while (! I1.Done()) {
     Object obj = I1.Elem ();
     if (!Member (obj)) return False;
     I1.Next ();
   }
   return True;
}
```

PL&C: Question 3

A. Write a function reverseEach in scheme that will take a list of lists and will reverse each one. You may assume the existence of the reverse and map functions. For example if there is a list x

((a b c d) (1 2 3 4))

then (reverseEach x) will return

((d c b a) (4 3 2 1)).

B. Write a function that creates a set of pairs of an atom with each element of a list. For example if there is a list

(define y '(4 5 6 7 8))

then (createPair 'x y) will generate

((x 4) (x 5) (x 6) (x 7) (x 8))

C. Now create a function crossProduct that will generate every possible combination of ordered pairs from two lists. For example, if we define z as follows:

(define z '(my your))

then (crossProduct y z) will generate

((4 my) (4 your) (5 my) (5 your) (6 my) (6 your) (7 my) (7 your) (8 my) (8 your))

Answers:

```
(define reverseEach (lambda (x) (map reverse x)))


(define createPair (lambda (x y)
   (cond ( (null? y) '())
          (else (cons (list x (car y)) (createPair x (cdr y))))
   )
 )
)

(define crossProduct (lambda (x y)
   (cond ((null? x) '())
        (else (append (createPair (car x) y) (crossProduct (cdr x) y)))
   )
 )
)
```

Operating Systems

OS 1: Process Scheduling

Part A

Consider the following set of processes, each of which performs no I/O (i.e., no process ever blocks). All times are in milliseconds. The CPU time is the total time required for the process. The creation time is the time when the process is created. So P1 is created when the problem begins, P2 is created 16 milliseconds later, and P3 is created 12 milliseconds after P2.

Process	CPU Time	Creation Time
P	40	0
Q	32	16
R	8	28

Assume RR (Round Robin) scheduling with a quantum of 12 milliseconds. Context switching from one process to **ANOTHER** takes 0.1ms, but "switching" from a process to itself requires **ZERO** time. Assume that starting the initial process at time zero takes **ZERO** time. At what time does each process finish? Show your work.

Solution

Part A was easy, everyone should have done well. P finished at 72.5; Q finished at 80.6; and R finished at 56.3.

interval	run	ready list at end	time remaining at end
0.0–12.0	P		28—32—8
12.0–24.0	P	Q	16—32—8
24.1–36.1	Q	P R	16—20—8
36.2–48.2	P	R Q	4—20—8
48.3–56.3	R	Q P	4—20—0
56.4–68.4	Q	P	4— 8 —0
68.5–72.5	P	Q	0— 8 —0
72.6–80.6	Q		0— 0 —0

Part B

For part B you are observing a system that uses a variation of RR scheduling, which is sometimes called Selfish Round Robin (SRR) and is defined below. For this part we make the simplifying assumption that no process ever blocks or terminates so the only states encountered are ready and running. We also assume (unlike part A) that context switching always takes **ZERO** time.

SRR is a **preemptive** scheduling algorithm characterized by two non-negative values a and b. When a process is created it is given an initial priority of 0 (zero) and its priority increases at a rate of a per millisecond. When a process has the highest priority in the system (others may have equal priority) the process becomes "accepted" and its priority now increases at a rate of b per millisecond. The accepted processes (note that they all have the same priority) are run using a RR scheduler with a quantum of q. When a process becomes accepted it is inserted at the rear of the RR queue.

The system we are observing has three processes P, Q, and R. If two or three processes are to be inserted in the RR queue simultaneously, the tie-breaking rules are that P is inserted before Q or R, and Q is inserted before R.

The system is turned on at time zero and we are able to monitor all process creation and context switching events. During the first 30ms (ms abbreviates milliseconds) the following events occurred.

Time	Event
10ms	Process R created
10ms	Process R runs
14ms	Process Q created
16ms	Context switch: process Q runs
18ms	Process P created
18ms	Context switch: process R runs
20ms	Context switch: process Q runs
22ms	Context switch: process R runs
24ms	Context switch: process P runs
26ms	Context switch: process Q runs
28ms	Context switch: process R runs
30ms	Context switch: process P runs

Determine a possible set of values for a, b, and q (there is more than one correct answer). Be sure to show you work carefully.

Solution

¿From the last few events we see that q=2ms. Calculating values for a and b is not as easy.

When R is created, it is the only process so is immediately accepted and starts running with an initial priority of 0, which increases at the rate of b per millisecond. Q is created 4ms later at which point R has priority $4b$ and Q has priority 0. Since Q is initially not accepted, its priority grows at the rate of a per millisecond. Assume Q becomes accepted at time $14 + x$. Since Q starts running

at the next quantum (16ms), $0 \leq x \leq 2$. At time $14 + x$ the priority of Q is xa and R has priority $(x+4)b$. Since P and Q have equal priority at time $14+x$, $xa = (x+4)b$ or $a/b = (x+4)/x$. Since $x \leq 2$, $a/b \geq (2+4)/2 = 3$.

Since P runs at 24ms it must have been accepted (and hence entered the ready queue) no latter than 22ms. Since P didn't run at 22ms, it must not have been accepted at 20ms. So P became accepted at a time $18 + y$ with $2 < y \leq 4$. At time $18 + y$ P has priority ya and R has priority $(y + 8)b$. Since all the priorities are equal when P is accepted, $ya = (y + 8)b$ or $a/b = (y + 8)/y$. Since $2 < y \leq 4$, $(4 + 8)/4 \leq a/b < (2 + 8)/2$ or $3 \leq a/b < 5$. Combined with $a/b \geq 3$ from the preceding paragraph we see that $a/b = 3$. So one solution is $a = 1$ and $b = 3$.

OS 2: I/O Systems

Consider a computer, which consists of a CPU and an I/O device **D** connected to main memory **M** via a shared bus. The only I/O operation accessible to user processes is **read**, invoked using a system call, which can be implemented using one of two strategies:

The **polling** strategy involves the following sequence of operations: (1) D's device driver writes the command and any arguments to control registers in D; (2) the driver repeatedly reads a status register on D, waiting for D to complete the requested operation and store the results in its data registers; (3) the driver reads the contents of D's data registers and writes them into M.

The **interrupt** strategy proceeds as follows: (1) D's device driver writes the command and any arguments to control registers in D; (2) D performs the requested operation, stores the results in its data registers, and interrupts the CPU upon completion causing control to get passed to D's driver; (3) the driver reads the contents of D's data registers and writes them into M.

There is only one user process running on this computer, so the OS does not switch to another process while processing a system call. The latency of an I/O system call is primarily determined by the cost of crossing from user mode to kernel mode and vice-versa, the time taken by D to perform the requested operation, the cost of processing an interrupt, and the bus interactions involving the CPU, device D, and memory M. All other operations (non-memory CPU actions) incur negligible cost.

Answer the following questions, assuming the following costs:

user-kernel (and kernel-user) crossing: 50 units
writing a command to D: 20 units
reading a status register on D: 20 units
time taken by D to perform requested operation: 200 units
interrupt processing: 100 units
reading a word from D's data registers: 1 unit
writing a word to main memory: 1 unit

1. If the maximum amount of data that can be read using a single I/O command (one interaction with the device) is 100 words, what is the time that a user process call to read 400 words takes to return using the polling strategy? What portion of this time is available for the OS to perform other activities?
2. If the interrupt strategy was used instead, what is the time taken for the system call in part (a), and what portion of this time is available for other activities?

Your solution in each case should provide the expression for computing the times of interest and clearly identify the different contributors (it is okay to just write the expression without simplifying it).

Solution

1. The overall sequence of events that happen when using the polling strategy is as follows: (1) the user program issues a system call to read 400 words; (2) the system call code invokes the

driver code four times, each time reading 100 words from the device; (3) the system call ends, returning control to the user program.

Therefore, since steps (1) and (2) each involve one user-kernel or kernel-user crossing,

$$T_{\text{call}} = \underbrace{50}_{\text{user}-\text{kernel}} + (4 \times T_{\text{driver}}) + \underbrace{50}_{\text{kernel}-\text{user}}$$

where T_{driver} is the time required by the driver code to perform one interaction with the device (which results in 100 words read into memory). Following the steps in the polling strategy sequence described above,

$$T_{\text{driver}} = \underbrace{(20 + 200 + 20)}_{\text{command, operation, status}} + \underbrace{100 \times (1 + 1)}_{\text{read/write contents}}$$

Although the driver repeatedly checks the status, only the last one (indicating completion of the I/O) adds to the time required since the others (indicating the I/O is still in progress) are overlapped with the device operation.

Combining the two expressions above,

$$T_{\text{call}} = 1860 \text{ units}$$

Since the driver code busy-waits for the completion of the I/O operation, so **no** portion of the overall system call time is available for the OS to perform other activities.

2. When the interrupt strategy is used instead, the only difference is in the computation of the T_{driver} time. In particular, following the sequence of steps for the interrupt strategy described above,

$$T_{\text{driver}} = \underbrace{(20 + 200 + 100)}_{\text{command, operation, interrupt}} + \underbrace{(100 \times (1 + 1)}_{\text{read/write;contents}}$$

Thus,

$$T_{\text{call}} = 2180 \text{ units}$$

Of this time, the CPU can be used for performing other activities during the **800 time units** when the device is completing its operation.