CORE EXAMINATION
Department of Computer Science
New York University
January 16, 2004

This is the common examination for the M.S. program in CS. It covers core computer science topics: Languages and Compilers, Operating Systems, and Algorithms. The exam has two parts. The first part lasts three hours and covers the first two topics. The second part, given this afternoon, lasts one and one-half hours, and covers algorithms.

Attempt all of the questions. Use the proper booklet for each question. Each booklet is marked with the Area and Question number, in the form PLC1, PLC2, PLC3, OS1, OS2, ALGS1, ALGS2, ALGS3. Use the appropriate booklet for each question. DO NOT put your name on the exam booklet. 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!

Programming Languages and Compilers

Question 1

1. A function that one often writes in Scheme is **`flatten`**, which takes a list *L* possibly containing nested lists and returns a list with no nesting which contains all the atoms within *L*. For example,

   ```
   (flatten '(a b (c (d e) f) g))
   ```

   will return `(a b c d e f g)`. Write `flatten` in Scheme or LISP

   (Note: the function `(pair? x)` returns true of *x* is a non-empty list. For our purpose, the empty list can be considered an atom).

   Write a procedure **`unflatten`** which takes a list *L* containing no nested lists and returns a list that does contain nesting, such that the *i*th element of *L* is nested inside *i* lists in the result. For example,

   ```
   (unflatten '(a b c d e))
   ```

   would return `(a (b (c (d (e)))))`. You can assume that L has at least one element.

Answer 1

```
(define (flatten L)
  (cond ((null? L) '())
        ((pair? (car L)) (append (flatten (car L)) (flatten (cdr L))))
        (else (cons (car L) (flatten (cdr L))))))
```

```
(define (unflatten L)
    (cond ((null? (cdr L)) L)
          (else (cons (car L) (list (unflatten (cdr L)))))))
```

Question 2

1. Consider the following Java declarations;

```
class Parent {
  public int func (int x) {..};
  public void proc (String s) {..};
  ... constructors, data, no other methods
}

class Child extends Parent {
   private void modify (int x) {..};
   public int func (int x) {..};
   ..  other data, no other methods
}
Parent P = new...    (some constructor call)
P = ...              (some other assignment)
```

   Show the contents of the dispatch table (the vtable in C++ jargon) for these two classes. Explain how the dispatching call in:

```
int val = P.func (123);
```

   is implemented at run-time.

2. Consider now the interface:

```
interface Ordered {
   Boolean lessThan (Ordered x);
}
```

   Complete the following class declarations:

```
class NewP extends Parent  implements Ordered {..};
class NewC extends Child   implements Ordered {..};
```

Show the dispatch tables for these two new classes.

3. Consider the following method, declared elsewhere:

```
Boolean Smaller (Ordered first, Ordered second) {
   return first.lessThan (second);
};
```

first can be an instance of any class that implements the interface Ordered, for example NewP or NewC. Explain how the call to LessThan is implemented at run-time.

Answer 2

1. The dispatch table is an array of pointers to methods. A class extension inherits the layout of the dispatch table from is parent. If a method is overridden, the corresponding pointer now points to the overriding method, but the position of the pointer in the table is unaffected. In our case, the dispatch table for Parent looks like:

```
pointer to code for func (declared in Parent)
pointer to proc
```

while the dispatch table for Child looks like:

```
pointer to code for func (overrides)
pointer to proc          (inherited)
pointer to modify        (new)
```

a dispatching call is an indirect call through a statically known entry in the dispatch table. A call to func on a polymorphic reference becomes a call through the first entry in the dispatch table: Tab[0](123) in C++. (Of course Java has no syntax for pointers to functions).

2. In order to satisfy the interface, the new classes must include a method declaration with the proper signature:

```
class NewP extends Parent  implements Ordered {
      Boolean lessThan (Ordered x) {
      // typically there will be a cast (Newp)x
      // to obtain the data members on which the
      // ordering is computed.
```

Same for class NewC. As a result, the method lessThan occupies the **third** slot in the dispatch table for NewP, and the **fourth** slot in NewC.

3. Because an interface method occupies different positions in the dispatch tables of unrelated classes, a call to such a method cannot be translated into a simple indirect call, as for a class method. The original JVM carried the

names of methods explicitly, and performed a sequential search over a run-time table to find the position of a given interface method. A better approach is to add a separate dispatch table for every interface that a class implements. The call is transformed into a double indirection: locate the interface table for the given class, extract the position of the method, and use this value to access the dispatch table.

A final note: the naive implementation of the JVM does things inefficiently, to avoid the fragile base class problem, but this is alleviated by just-in-time compilation, which does transform a dispatching call into an indexed call through an array of pointers. The central point remains that a call to an interface methods will be slightly more complicated that a call to a class method, because it has to deal with the layout of unrelated classes.

Question 3

1. In many programming languages (Fortran, Pascal, Ada..) multidimensional arrays are stored as one contiguous object, either in row-major or column-major order. Consider the declaration:

   ```
   A : array (1..100, 1..100) of float;
   ```

   Indicate with a simple diagram how elements of A are placed in memory. Assuming that A is stored in row-major order, write the assembly code (for the machine of your choice) that will be generated for the loop:

   ```
   for J in 1..100 loop
      A (J, J) := 0;
   end loop;
   ```

2. In other languages, such as Java, multidimensional arrays are implemented with indirection, as arrays of arrays. Given the declaration:

   ```
   int [][] A = new int [100][100];
   ```

   provide a diagram to indicate how A is stored, and write the assembly code that will be generated for the loop:

   ```
   for (int j=0; j < 100; j++) A[j][j] = 0;
   ```

Answer 3 For a discussion of array layout, see the text by Scott, sec. 7.4.3.

Question 1

Consider the two processes shown below, which are run concurrently. The variable `X` is shared between the processes and is initially 0. The variables `i` and `j` are local to the process in which they appear. The shared binary semaphore `b` is initially open. The shared counting (also called general) semaphore `C` is initially 3. The shared counting semaphore `D` is initially 0.

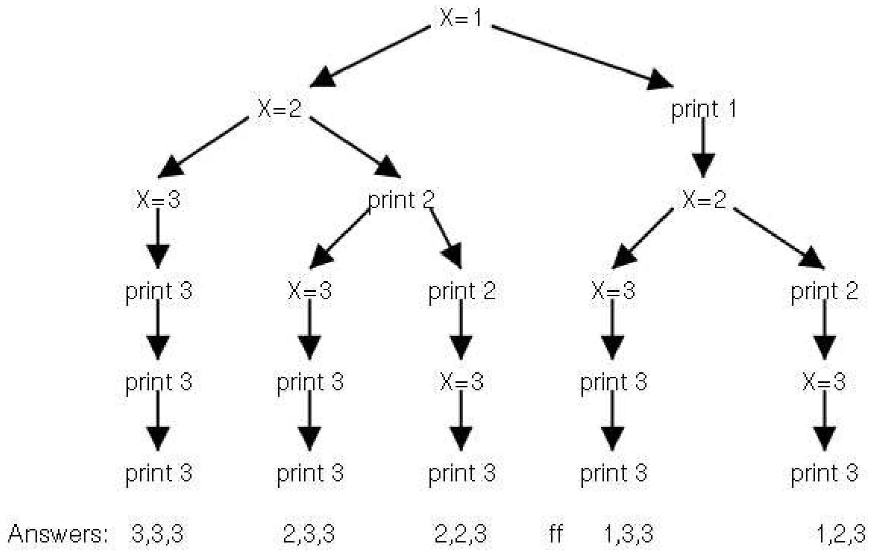
There are several possible outputs for this pair of concurrent programs. Give all the possibilities and explain your reasoning.

```
    Process 1                         Process 2

for i = 1 to 3                   for j = 1 to 3
    P(C)                              P(D)
    P(b); X = X + 1; V(b)             P(b); print X; V(b)
    V(D)                              V(C)
```

Answer to OS Question 1:

This is basically a producer (increment) and consumer (print) problem. The consumer cannot run more often than the producer, and the producer cannot get more than three iterations ahead of the consumer since C is initially 3. The picture below shows the possible executions.



Question 2

1. Consider a UNIX-like inode-based file system. Recall that whereas a file contains arbitrary contents, a directory, however, has a prescribed format. It consists of a number of entries, each of which contains the name of a file or a subdirectory and a pointer to the corresponding inode. Also recall that, when a (hard) link L is created to an (existing) file f, a directory entry is allocated for L and it points to the **same** inode as f does.

   A particular user starts in her home directory (`/home/user`), which initially contains a single file, `file1`, and executes the following sequence of UNIX commands:

```
      mkdir dirA;
      mkdir dirB;
S1 ----------------
      cd dirA;
      cp ../file1 fileA;
S2 ----------------
      cd ../dirB;
      ln ../dirA/fileA fileB; # hard link named fileB (to fileA)
S3 ----------------
      cd ../dirA;
      rm fileA;
S4 ----------------
```

   For each of the points marked $S1$, $S2$, $S3$, and $S4$, in the above sequence, draw a graph showing the user's files and directories and drawing arrows from each directory D to each subdirectory of D and to each file contained in D. Start at `/home/user`.

2. Refining question (a), we note that the file system consists of 4 kinds of objects: (1) **data blocks** (which store file contents), (2) **directory blocks** (which store the contents of a directory), (3) **indirect blocks** (described below), and (4) **inodes** (also described below).

   Data blocks, directory blocks, and indirect blocks are each 1 disk block; whereas, 10 inodes fit in one disk block. Assume each disk block is 4000 bytes and every pointer (including a null pointer) is 4 bytes.

   The inode for a file contains 76 bytes of attribute information followed by 80 pointers to the first 80 data blocks of the file followed by a pointer to the indirect block for the file. If the file has exactly 80 data blocks, the data block pointers are all valid, the indirect block pointer is null, and no indirect block is allocated. If the file has fewer than 80 data blocks the unused data block pointers as well as the indirect block pointer are all null and no indirect block is allocated. If the file has more than 80 data blocks, the indirect block is allocated and contains pointers to the excess data blocks. We assume no file is so big that it overflows the indirect block.

   The inode for a directory is essentially the same, the only relevent difference is that the pointers are now to directory blocks instead of data blocks.

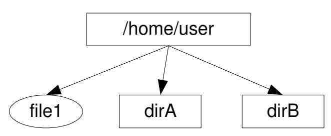
Assume that the file `/home/user/file1` contains **100** data blocks.

For each of the points marked *S*1, *S*2, *S*3, and *S*4, in the above sequence, how many **additional** inodes and data, directory, and indirect blocks would have been created?

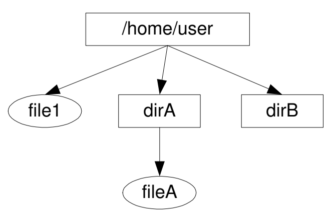
Answer to OS Question 2:

1. The figures below show the state of the user's files and directories at each of the points *S*1 through *S*4. To obtain full credit, the answer somehow needs to indicate that `fileA` and `fileB` both refer to the same file on disk, and that this file continues to exist at point S4.

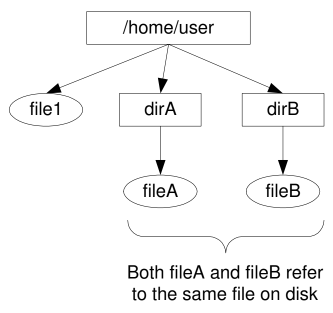
At point S1:



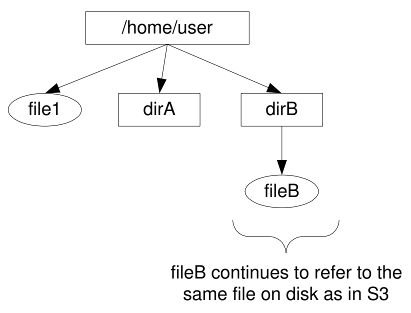
At point S2:



At point S3:



At point S4:



2. This question required one to reason about how files and directories are stored on disk. Most of the information for this was provided in the text of the question itself.

   At point *S*1, two new directories `dirA` and `dirB` would have been created. In terms of disk storage, this would require two the use of additional directory entries in the directory block corresponding to the `/home/user` directory. Each of these directory entries would point to a directory inode, which in turn would point to a directory block storing the entries for the corresponding directory. Given the size of a directory block (4000 bytes), no additional blocks need to be allocated at the `/home/user` level for the two additional entries. Thus, the additional storage that would be created at point *S*1 would be **two inodes** (one each for `dirA` and `dirB`), and **two directory blocks** (to store the directory entries of `dirA` and `dirB` respectively). Note that we are assuming that a directory block is allocated as soon as the directory is created; in some instances, this allocation may be deferred until the directory has some non-default entries.

At point *S*2, we would have created a new file, `fileA`, under directory `dirA`. In terms of disk storage, this would require the use of a directory entry in `dirA`'s directory block (again, given the size of a directory block, we would not need to allocate any additional storage for this). This directory entry would contain a pointer to an inode for `fileA`. This inode would in turn point to the data blocks storing the contents of the file. Since the file is specified as requiring 100 blocks, we would have 80 direct pointers from the inode, and would need to allocate an indirect block to point to the 20 remaining blocks. Thus, at point *S*2, we would need to create (in addition to the storage at point *S*1), **one inode**, **one indirect block**, and **100 data blocks**.

At point *S*3, we would have created a hard-link, `fileB`, under directory `dirB`, which points to `fileA`. As described in the question text, a hard-link merely corresponds to a directory entry that points to the inode of the original file. Since the only allocation required is that of the directory entry for `fileB`, as compared to the storage at point *S*2, we do not need to create **any additional** inodes, directory blocks, indirect blocks, or data blocks.

At point *S*4, we remove the original file, `fileA`. Following the semantics of hard links in Unix-like systems, only the corresponding directory entry in `dirA` is deleted. The storage for the file, including the inode allocated at point *S*2 and the indirect block and data blocks continue to be valid (and are pointed to by `fileB`'s directory entry). Thus, there is **no additional storage** that needs to be created, and equally importantly, no disk storage is freed up as compared to point *S*3.