Practice Final Exam

Exam Facts

Wednesday, December 13th from 8:30 until 11:30 a.m.
Maricia, Eric, and Erin's students: TCSeq201
Miler and Karen's students: Bishop Auditorium

Recall that the exam is closed-book and closed-notes, and that the time you'll be given will be an ample amount of time for the exam. Because some students have exams after ours, I will need to force students to respect the 3 hour limit and just design the exam to be two hours long instead of three hours long. In practice, I will probably let students take 3 hours and 15 minutes, since students can still get to their next exam even if I were to go a little over. Most students gave a big thumbs up to the untimed midterm, but a sizable portion of you though it was overkill, so I'm trying to satisfy both camps here.

Material

The final is comprehensive but will emphasize topics covered after the midterm, so that is the best place to concentrate your studying efforts. Want to see where you've been? Here's the pretty impressive list of things you've learned in 107:

- Implementation— stack-heap diagrams, memory layout, structures, arrays and pointers, function calls, parameter passing, local variables, code generation.
- C— arrays, pointers, `malloc`, `&`, `*`, `void*`, typecasts, function pointers, preprocessor, compiler, linker.
- Concurrency— threads, race conditions, mutual exclusion, efficient blocking, binary semaphores, general semaphores, recognizing and avoiding deadlock conditions.
- Java— OOP paradigm, classes, objects, messages, inheritance, overriding, structuring an inheritance tree to factor code, class variables and methods, simple Java built-ins.
- Lisp— functional paradigm, list operations, recursion, `mapcar`, lambda, lexical closures, run-time typing.

This handout is intended to give you practice solving problems that are comparable in format and difficulty to those which will appear on the exam. This is the final that was given last autumn, with the space left for answers removed in order to conserve trees. I highly recommend working through these problems under exam-like conditions to prepare. Note, however, that this was written to be full 3-hour, open-notes, open-book exam.

1) C and code generation

a) Generate code for the single marked line of C using our stack layout and code generation conventions.

```
struct list {
   int data[2];
   struct list *next;
};


char arr[32];

((struct list *)arr)->next += arr[0];  // generate code for this line
```

b) In order for different C compilers to work together, a standard arrangement for parameters is necessary so that both the caller and callee agree about the layout of the activation record. In the C language specification, it is required that a compiler push the arguments to a function onto the stack from right to left (i.e. the lastmost parameter is placed on the stack first, followed by the earlier parameters). Although it was important to choose a single standard, this particular arrangement was not arbitrarily chosen. What problem would be caused if all C compilers instead agreed to push parameters from left to right?

c) The `GroupByLength` function below takes a DArray of words and groups the words by lengths into separate DArrays. The `words` parameter is a DArray of char*. The `allGroups` parameter is a DArray of DArrays of char*. At the end of the function, the 0^{th} entry of the allGroups DArray contains the group with the length-0 words, the 1^{st} entry holds all the length-1 words, and so on up to a maximum word length of 25. Assume that the allGroups parameter has already been configured to have the necessary empty DArrays as elements.

Your job is to supply the two missing helper functions. The `EnterWordIntoGroup` map function is used to add each word from the words DArray into the correct DArray group for its length. The `SummarizeGroup` map function is used to summarize each DArray group, printing the word length and count, e.g. "10 words of length 4." You may assume there is at least one word of every length represented. You do not need to make copies of the words when adding them to the DArray groups.

```
static void GroupByLength(DArray words, DArray allGroups)
{
     ArrayMap(words, EnterWordIntoGroup, allGroups);
     ArrayMap(allGroups, SummarizeGroup, NULL);
}

static void EnterWordIntoGroup(void *elem, void *clientData)
{


}
```

```
static void SummarizeGroup(void *elem, void *clientData)
{


}
```

2) LISP coding

a) Write the function (`weave list1 list2`) which takes two lists and returns a list of the elements from the two lists alternately arranged (i.e. the first element from list1, then the first from the list2, then the second from list1, and so on). If one list runs out before the other, the rest of the remaining list is tacked onto the end.

```
? (weave '(1 2 3) '(a b c))
(1 A 2 B 3 C)
? (weave '(1 2) '(A B C D))
(1 A 2 B C D)
```

Write this function recursively.

```
(defun weave (list1 list2)
```

b) Write the function (`closest-to-average nums`) that given a non-empty list of numbers returns the number in the list that is closest in absolute value to the overall list average. The LISP built-in `abs` gives the absolute value of a number. Your function can assume the list will not be empty.

```
? (closest-to-average '(12 3 7 9))
7
```

Your function should use the `most` function listed below. Your function must run in linear time.

```
; MOST (list comparator)
; Takes a non-empty list and two argument comparator which returns T
; if its first argument was "more than" its second argument.  Returns
; the element in the list which was "most" according to the comparator.
; Runs in time linear to the number of elements in the list.

? (most '(3 46 7 -72 6 -8) #'>)
46
```

```
(defun closest-to-average(nums)
```

c) LISP has built-in functions for mapping over the elements of a list (`mapcar`) as well as mapping over the sublists of a list (`maplist`). Demonstrate that is not necessary to have both by writing your own version of `mapcar` in terms of `maplist`.

```
? (my-mapcar #'evenp '(4 5 7))
(T NIL NIL)
? (my-mapcar #'length '((a "hi") (3)))
(2 1)
```

Your function should use the LISP built-in `maplist` function:

```
; MAPLIST (fn list)
: Like mapcar, but maps over sublists instead of elements. Invokes fn
; once for each sublist of decreasing length. For example, maplist over
; (a b c) calls fn once passing (a b c), once passing (b c) and once
; passing (c). Results are gathered and returned in a list.

? (maplist #'length '(30 apple 12 "Hi!"))
(4 3 2 1)
```

You should write this function using mapping, not recursion.

```
(defun my-mapcar (fn list)
```

d) Write the function `(map-unique-pairs fn list)` which takes a list and a function of two arguments and invokes the function once for each unique pairing of two different elements chosen from the list. An element should not be paired with itself, nor should it be paired with the same element more than once. For example, if the function has been called on the pair A and B, it should not be called again B and A. Thus, the passed-in function is called exactly once per distinct combination of two non-equal elements. Like `mapcar`, the results are gathered and returned in a list. The order of the results in the list is not defined.

```
? (map-unique-pairs #'list '(3 10 4 1 3))
((3 10) (3 4) (3 1) (10 4) (10 1) (4 1)))
? (map-unique-pairs #'* '(3 5 10 5))
(15 30 50)
```

You should use the `maplist` built-in (described in part c) and the `uniquify` function presented here.

```
; UNIQUIFY (list)
; Returns a list with only one instance of each distinct element
; in the original list (i.e. removes all duplicates). Duplicates
; determined by using the equal function.

? (uniquify '(2 4 2 3 8 4))
(2 4 3 8)
```

You should write this function using mapping, not recursion.

```
(defun map-unique-pairs (fn list)
```

3) Concurrency coding

You're writing a simulation for the ever-popular dorm ski trip. There are the RAs who bring their residents to the slopes, the residents who rent skis and go skiing, and the clerk who runs the ski lodge. At the beginning of the simulation, one thread is launched for each RA and one for the ski lodge clerk. Threads for the residents are launched later in the simulation.

An RA first drives the van of residents to the slopes. Once they arrive, the RA dispatches a separate thread for each resident, so they go off and ski independently. Each resident wants to take as many runs down the mountain as possible before the slopes close. Before each run, the resident has to get some rental skis. The available rental skis are heaped in a pile inside the lodge and as long as the lodge is open, residents can come in grab any set. If the lodge is closed, no more skis will be given out and thus the resident calls it a day and goes back to the van. If all skis are currently rented the resident waits until a set is returned. When the resident gets skis, they head to the slopes and make a run. Because there are so many more skiers than available skis, rental skis are turned in after each run to give others a chance.

The RA doesn't ski, but instead waits with the van until all the residents return and then drives home. Note that the RA should be careful to wait for exactly those residents it spawned, not just a collection of random residents to finish.

The ski lodge clerk sleeps all day, doing nothing. At closing time, he wakes up to shut down the ski lodge. Once the lodge is closed, no more rental skis will be checked out. The clerk waits for all rental skis to be returned and then he is done.

An RA thread finishes after all its residents come back to the van. A resident finishes after skiing as much as they can until the lodge closes. The lodge clerk finishes at the end of the day after all rental skis have been returned.

Here is the starting main function for the ski trip simulation. You cannot change any of this code.

```
#define NUM_DORMS 25
#define NUM_RENTAL_SKIS 75
#define NUM_RESIDENTS 12            // each RA brings 12 residents in van
#define ALL_DAY (1000*60*8)   // time used for clerk sleep period

void main(void)
{
   int i;

   InitThreadPackage(false);

   for (i = 0; i < NUM_DORMS; i++)        // create all RA threads
      ThreadNew("RA", RA, 0);
   ThreadNew("Ski Lodge Clerk", Clerk, 0);
   RunAllThreads();
}

      // these simulation functions don't do anything, just "fake"
static void Drive(void);      // for RA, drive to slopes
static void OneRun(void);     // for Resident, to take one ski run
```

Assume the above helpers are already written and are thread-safe, you can just call them when you need to. Your job will be to write the `RA`, `Resident`, and `Clerk` functions to properly synchronize the different activities and efficiently share the common resources.

Declare your global variables and semaphores. You don't need to write out SemaphoreNew calls or explicitly initialize globals, but **clearly indicate the initial value** for each semaphore or global variable.

Write `RA`, `Resident`, and `Clerk` functions. There is another blank page after this one. You can abbreviate SemaphoreWait and SemaphoreSignal as Signal and Wait if you like.

4) Java

On the next page, you'll find code for the starting implementation of a generic Employee class. An employee tracks a Vector of their skills, their boss (who is another Employee) and the number of tasks they have completed. An employee responds to messages to find out whether they are skilled in a task and to perform a task. Our employees are specialized into Workaholics, BrownNosers, and Schmoozers. Workaholics are known for working long hours perfecting their work, BrownNosers are constantly trying to move ahead by pleasing the boss, and Schmoozers get along by doing favors for others in hopes of getting something in return.

We're interested in what happens when you ask an employee to perform a list of tasks by sending a `doAllTasks(Employee asker, String tasks[])` message to an Employee object. The `asker` parameter is another Employee who is making the request. The `tasks` parameter is an array of task names, given as Strings. For each task in this list, the employee will consider it and either perform the task or skip it.

In general, Employees agree to perform a task if they are qualified, they have time available, and they find the task "interesting." Any employee is qualified for a task if it appears in their list of skills and has time if they have not yet completed over 100 tasks. Workaholics differ in that they don't stick to this fixed bound. Any time a workaholic refuses to do a task (for any reason), they increase their task limit by one so that they may agree to sign up for more tasks in the future (.e.g after refusing 3 tasks, a workaholic will make time for 103, not just 100, tasks).

The different employees also have varying ways of deciding whether a task is interesting. When asked, a Schmoozer's response alternates between finding a task interesting and finding it not, regardless of the task being asked about. Workaholics love working, and thus find all tasks interesting. Since BrownNosers are always have their eye on the next level, they only find a task interesting if their boss has that skill. BrownNosers also make a special exception in that they always agree to do any task if their boss is the one asking, whether or not they have the skill, time, or find the task interesting.

The employees differ in some other ways as well. For example, when asked to perform a task, Workaholics not only do the original task, but they follow up by performing a second task of "Checking" to make sure they got it right. Whenever a Schmoozer performs a task for someone, he adds the asker to his list of folks who "owe him a favor." Whenever a Schmoozer is asked to do a task, he first tries to delegate the task by asking the list of folks who owe him a favor. If one performs the task, the Schmoozer removes that person from his list, and credits himself with performing the task of "Delegating". A Schmoozer considers doing the task himself only if no one in his list will accept his delegation. The "Checking" and "Delegating" tasks are performed by the employee just like any other tasks: printing the message, incrementing the count of tasks done, and adding skill if not present.

Your job is to design and implement the employee classes as described above. You are free to add any other helper classes and can change or add to the generic Employee class as well. Note that we are not going to worry about allocation or initialization. You do not have to write any constructors. Where needed you can indicate the starting value for variables.

Your most important design goal is to avoid code duplication. It is recommended you think through the entire design before making any decisions.

```
public class Employee {

   public void performTask(Employee asker, String task)
   {
      System.out.println("Performing task " + task + " for " + asker);
      numTasksDone++;
      if (!hasAbility(task)) skills.addElement(task); // just gained new skill
   }

   public boolean hasAbility(String task)
   {
      return skills.contains(task);
   }

   public boolean hasTime()
   {
      return (numTasksDone < MaxTasks);
   }

                  // instance & class variables, assume they are
                  // set with correct values as needed
   protected int numTasksDone;
   protected Vector skills;
   protected Employee boss;
   protected static final int MaxTasks = 100;
}
```

Make it clear what modifications you want to make to the above Employee class. You can change the code inside the given methods, add new methods, add new variables, make things abstract, change the access specifiers, etc. whatever you need to get the job done.

5) Short answer

Each of these questions can be answered fairly briefly, just one or two sentences is all it takes.

· C and LISP differ in their treatment of functional arguments. The `look` function from the LISP assignment can be seen as a rough equivalent of `ArraySearch` on a DArray and both take a functional argument as a parameter. Give an example of a type of error you can make involving the functional argument to look / ArraySearch that is caught at compile-time by C yet not caught until runtime in LISP. Give an example of a type of mistake involving the functional argument to `look` detected by LISP that is not caught by C's error-checking at all.

b) C, LISP, and Java are all intended to be portable (each with varying degrees of success), but only Java is designed to be architecture-neutral. Describe one advantage and one disadvantage this gives Java.

c) You have a Java Vector containing String elements. What happens if you don't cast the result from Vector's `elementAt()` before trying to send that element a String-specific method?

d) Consider the following similar code fragments, one in LISP, and one in C:

```
(let ((a (<some-expression>)))      a = <some-expression>;
  (+ (func1 a) (func2 a)))          return func1(a) + func2(a);
```

You are trying to build a wonder compiler that can transform sequential programs (i.e. authored with no explicit concurrency) into ones that operate in parallel. To convert the above fragments, you propose dispatching two concurrent threads, one to compute func1 on argument a, the other to compute func2, wait for the two results and then do the addition. Explain why this transformation will be safe and correct for the LISP code but not for the C version.