Stanford University
Computer Science Department
CS 140 Midterm Exam
Dawson Engler
Fall 1999

Name: _____

Please initial the bottom left corner of each page.

This is an open-book exam. You have 50 minutes to answer as many questions as possible. The number in parenthesis at the beginning of each question indicates the number of points given to the question, and about how many minutes you should spend on your answer. Write all of your answers directly on the paper. Make your answers as concise as possible. Sentence fragments ok.

Stanford University Honor Code:
In accordance with both the letter and the spirit of the Honor Code, I did not cheat on this exam.

Signature: _____

| **Problem** | **Points** | **Score** |
|---|---|---|
| 1 | 15 | |
| 2 | 10 | |
| 3 | 10 | |
| 4 | 10 | |
| 5 | 10 | |
| 6 | 20 | |
| Total | 75 | |

**Question 1) [15 points]** V. A. Pid is peddling his spiffy new VAs-R-Us computer, which sports128-bit virtual addresses. He wants you to exploit this large address space to make traditional linkers obsolete. On his machine, routines and variables must be located at the address formed by treating their name as the high 64-bits of their virtual address. (Names longer than 8 bytes are not allowed; names shorter than 8 bytes have trailing 0's for padding.) The stack and heap segments are located in the low half of the address space, while code and data get the high half.

A) **[10 points]** In a few sentences, briefly describe how this organization changes linking and loading. In particular, what traditional linking steps can the new system now do locally, when object files generated, instead deferring them until link time? (And, how does it do them?) What linking steps must still occur at link time to preserve traditional semantics? What information does the loader need, and when is this information computed? Finally, what traditional constructs will no longer work on this system?

*3pts: can resolve reference immediately (name gives address). Symbol table just needs to record names used in references and definitions (but not their offsets).*

*4pts: need to create a list of all definitions to check for multiple defs and unresolved refs. The list is given to the loader (along with the size of each object) so that it can determine what it needs to map.*

*3pts: static names won't work, or routines/data larger than 2^64. Static shared linking still works, but name conflicts a severe problem.*

B) **[5 points]** Programs run significantly slower and consume significantly more memory on this machine than on a traditional system. What is the likely reason for this? What do you expect the increased memory consumption to be proportional to?

.

*Most likely: internal fragmentation. We'd expect this consumption to be proportional to the number of routines and variables: in realistic cases each will have it's own segment of virtual memory (rather than all code sharing one segment and data another). The "tail" end of this segment will be unused, typically wasting (at least) ½ page (easily much worse, since we blow a page for every variable).*

*Some points were given for pointing out that the page table overhead would be larger . While true, for reasonable page table structures this overhead will be dwarfed by the overhead of internal fragmentation.*

**Question 2) [10 points]** In the following code, three processes produce output using the routine "putc" and synchronize using two semaphores "L" and "R."

```
   semaphore L = 3, R = 0;   /* initialization */

/* Process 1 */          /* process 2 */          /* process 3 */
L1:                      L2:                      L3:
        P(L);                    P(R);                    P(R);
        putc('C');               putc('A');
        V(R);                    putc('B');               putc('D');
                                 V(R);
        goto L1;                 goto L2;                 goto L3;
```

a) How many D's are printed when this set of processes runs?

3

b) What is the smallest number of A's that might be printed when this set of processes runs?

0

c) Is CABABDDCABCABD a possible output sequence when this set of processes runs?

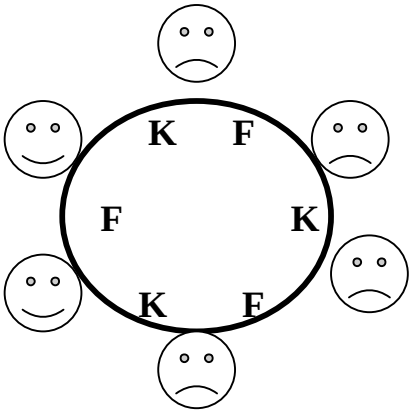
no

d) Is CABACDBCABDD a possible output sequence when this set of processes runs?

yes

**Question 3) [10 points]** The notorious dining gourmand problems is as follows. Six gourmands sit around a table, with a large hunk of well-done roast beef in the middle. Forks (represented as 'F') and knives (represented as 'K') are arranged as shown below. (Gourmands that like their roast beef rare have frowny faces.) Each gourmand obeys the following algorithm:

1) Grab the closest knife.
2) Grab the closest fork.
3) Carve and devour a piece of beef.
4) Put down the knife and fork where they used to be.




Can deadlock ever occur? Please be convincingly concrete: either indicate why this algorithm satisfies our deadlock conditions, or which one(s) it avoids. If you said that the system could deadlock, describe a reasonable deadlock avoidance scheme. If you said the scheme could not deadlock, say which one it uses.

*Deadlock cannot occur (-6 if this was wrong) since the algorithm defines a partial ordering, removing the ability to have circular requests. The easiest way to see the partial order is to label knives 1,2, 3 and forks 4,5,6. Clearly we only acquire in ascending order, thereby preventing circularity.*

**Question 4). [10 points]** Pleased with his company's success, Mr. P decides to expand operations by going into the scheduling business. He can't understand multi-level feedback, so asks you to implement a simpler scheduling system: bi-polar scheduling. In the bi-polar model, there are two queues, one for I/O jobs, one for CPU jobs. If I/O devices are idle, the scheduler runs an I/O job, otherwise it runs a CPU job. The system uses time slicing, and decides which queue a job belongs on based on whether or not it blocks before its time slice expires. Give two concrete weaknesses of this scheme as compared to a multi-level feedback approach. Give one job mix for which this scheduler will run better.

*+7 points for two of:*

- *Starvation: many I/O devices and I/O jobs = CPU job never gets to run*
- *To short view of the past: if a job alternates between CPU and I/O it will bounce between the (wrong) queues.*
- *I/O job will likely run for a bit before blocking (say T cycles). Since we only run an I/O job when the I/O devices are idle, the device will remain idle for another T cycles before the job launches an I/O request.*
-

*+3 points for job mix (had to say why better). The easiest answer is that since the algorithm is unfair we can exploit this to keep more I/O devices busy than an algorithm that was more fair and would try to run CPU jobs more often (thereby allowing the I/O device to become idle). One such mix: one I/O device, several I/O jobs (where each job immediately blocks on I/O), and many CPU jobs. Bi-polar will keep I/O devices more busy than the MLF scheme, which to prevent starvation will periodically run the CPU jobs (leaving I/O idle).*

**Question 5). [10 points]** Virtual memory systems can map multiple virtual pages to the same physical page ("aliasing"). Does this create problems for either virtually or physically tagged data caches? (And why?) Why might it matter whether the page refers to code or data?

*It creates problems for virtual tagged caches, which tag cache entries based on their virtual (rather than physical) address. Thus, data mapped at multiple virtual addresses can be cached at multiple places in the cache. If we write to one location, the others will be inconsistent. This organization does not cause problems for read-only data (such as code).*

**Question 6). [20 points]** You are sleeping too much, so decide to work for a startup, "no-locks.com." The no-lock hardware associates every memory byte with a "watch bit" that can be set by calling the routine "mem_watch":

```
/* set the watch bit for memory address addr */
void mem_watch(char *addr);
```

This location can then be modified using the routine "mem_set:"

```
int mem_set(char *addr, char x):
```

Which atomically:

1) Stores "x" into "addr" if its watch bit is 1.
2) Sets the watch bit to 0.
3) Returns the original watch bit value.

The following sequential code adds and removes characters from an infinite buffer. Rewrite the code to work correctly when multiple threads are adding and removing elements from the buffer, using nothing other than the two provided routines "mem_watch" and "mem_set." You may rewrite the data structures if you want. Provide a short, intuitive correctness argument for your modifications.

```
char buf[];         /* infinite buffer */
int  head = 0,      /* producers' position in buf */
     tail = 0,      /* consumers' position in buf */
     n = 0;         /* number of characters in buf */

char get(void) {
        /* no characters = infinite loop */
        assert(n > 0);
        /* take character */
        c = buf[tail];
        tail++;
        n--;
        return c;
}

void put(char c) {
        buf[head] = c;
        head++;
        n++;
}
```

*Solution:*

```
char buf[];
int head = 0, tail = 0;
char getlock, putlock;

void acquire(char *l) {
        while(!mem_set(l, 0))
                ;
}

void release(char *l) {
        mem_watch(l);
}
```

```
/* called before buf can be used. */
void init() {
        mem_watch(&getlock);
        mem_watch(&putlock);
}

char get(){
        while(1) {
        acquire(&getlock);
        if(head != tail) {
                break;
                release(&getock);
        }
        c = buf[tail];
        tail++;
        release(&getlock);
        return c;
}

void put(char c) {
        acquire(&putlock);
        buf[head] = c;
        head++;
        release(&putlock);
}
```

*Grading was mostly bi-modal:*

*If mostly correct:*

*-1 forgotten initialization*
*-2 for not removing the assertion*
*-1 or –2 for random logic errors (e.g., testing for true rather than false)*

*if mostly incorrect:*

*0 if no answer given*
*2-3 if answer provided doesn't work at all*
*4-5 if answer works in very limited cases*
*6-8 if answer forks for more cases*
*9-12 if answer fails to synch some of the accesses*

*codes:*

*A means updates to tail could be lost*
*B means that updates to head could be lost*
*C means that it's possible to return the same value twice (from two different gets)*
*D means that it's possible to overwrite the same buffer location twice (from two different puts)*
*E means you forgot to initialize variables.*