csc/cpe 453 Midterm Solutions

Winter 2003

Name: Key

Section: All

Rules:

- Do all your own work. Nothing says your neighbor has any better idea what the answer is.
- Do not discuss this exam outside of class until after 4:00pm.

Suggestions(mostly the obvious):

- When in doubt, state any assumptions you make in solving a problem. **If you think there is a misprint, ask me.**
- **Read the questions carefully.** Be sure to answer all parts.
- Identify your answers *clearly.*
- Watch the time/point tradeoff: 95ts/50min works out to 31.6s/pt.
- Problems are not necessarily in order of difficulty.
- Be sure you have all pages. Pages other than this one are numbered "n of 7".

Encouragement:

- Good Luck!

Problem	**Possible**	**Score**
1	10	
2	10	
3	10	
4	10	
5	15	
6	20	
7	20	
Total:	**95**	

Answer the following questions clearly, concisely, and (where possible) correctly.

1. (10) Busywaiting is often denegrated in the concurrent programming community as a crude and inefficient practice. However, it has its place.

   (a) First, what is busywaiting?
   Solution:

   Busywaiting is the process of repeatedly checking a condition (in a tight loop, e.g.) until it becomes true.

   (b) Under what circumstances would it be appropriate to choose a busywaiting approach?

   Solution:

   It would be appropriate to choose a busywaiting solution:
   (a) when the expected wait time is small,
   (b) when there is nothing else to do until the condition is satisfied, or
   (c) when there is no other mechanism available through which to synchronize.
   The first and third of these points are often true of synchronizing between processors of a multiprocessor.

2. (10) Semaphore waiting lists are often implemented as queues served in FIFO order. Could they be implemented as stacks? What problems might this cause?
   Solution:

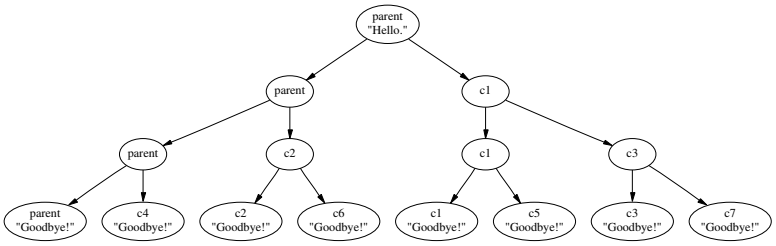
   Certainly one could implement semaphore waiting lists as stacks, but doing so would make concurrent processes synchronizing via semaphores much more vulnerable to starvation.

3. (10) What will be the output of the following program? Explain.

```
int main(int argc, char *argv[]){
  printf("Hello.\n");
  fork();
  fork();
  fork();
  printf("Goodbye!\n");
  return 0;
}
```

Solution:

```
% a.out
Hello.
Goodbye!
Goodbye!
Goodbye!
Goodbye!
Goodbye!
Goodbye!
Goodbye!
Goodbye!
%
```



This program prints "Hello" once in the parent, and "Goodbye!" in each of the 8 resulting processes. There are 7 children created, not 3, because each of the children executes the following `fork()s`*, too.*

4. (10) A programmer dissatisfied with the behavior of a C library function can redefine it without limiting the capabilities of the program. (That is, there is nothing the program could have done before the redefinition that it could not do afterwards.) A system call, however cannot be replaced without limiting the program. Why is this?

Solution:

It all has to do with privilege:

A system call is the means by which the kernel provides access to a particular operating system service, and the services available through system calls (e.g., reading and writing the disk, allocating memory, starting new processes, etc.) are reserved to the kernel; there is no other way of doing these things. Replacing a system call eliminates that particular entry to the kernel, limiting the program.

A library call, in contrast, is code that runs entirely within the user's program requiring only the user's privileges. This code can be replaced by a do-it-yourselfer (masochist?) without giving up anything irreplaceable.

5. (15) Given the following program:

```
int main(int argc, char *argv[]){
  int n;
  for(n = 1; n < argc; n++ )
    execvp( argv[n], &(argv[n]) );
  return 0;
}
```

Solution:

Suppose that it is compiled in the current directory as `myprog` and that one gives the command:

```
% myprog myprog ls myprog myprog myprog
```

(a) How many times will `execvp()` be called during the run of this program?

`execvp()` *will be called twice.*

(b) What will be the expected output of the run?

```
myprog myprog myprog
```

(c) Explain.

The program is called with `argv` *containing the list above. Each time around the loop (*`1...n-1`*) it will* `exec()` `argv[n]` *with arguments* `argv[n+1]` ...`argv[argc-1]`*. Because the* `exec()` *replaces the current program, the loop never actually loops in any of them.*

So, the original instance of `myprog` *calls* `execvp()` *with "*`myprog ls myprog myprog myprog`*" as its arguments. This new instance of* `myprog` *will call* `execvp()` *with "*`ls myprog myprog myprog`*" as its arguments.* `ls` *will proceed to list each of its arguments. Since we know* `myprog` *is in the current directory, it will be found, so the result of the listing will be "*`myprog myprog myprog`*".*

6. (20) Assume you have hacked the kernel and are using a version of MINIX with a 1 second(!) quantum. Suppose there are five user-level jobs, labeled A through E, that require the following amounts of time to run (if they could get the whole CPU to themselves).

Job	Time
A	2 sec.
B	3 sec.
C	4 sec.
D	5 sec.
E	2 sec.

Supposing that these five jobs become runnable at very close to the same time and are initially added to the run queue in alphabetical order, answer the following questions.

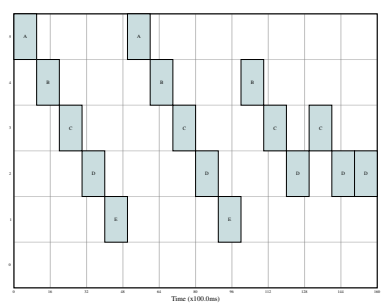
Recall that within a priority class, MINIX uses round-robin scheduling. You may assume that these are the only five jobs in the system and that the overhead required for context switches is negligible.

(a) After how many seconds will each job terminate? Briefly explain your reasoning.

Job	Finish Time
A	*6 sec.*
B	*11 sec.*
C	*14 sec.*
D	*16 sec.*
E	*10 sec.*

Solution

Time Ending	A	B	C	D	E
0.0s	—	—	—	—	—
1.0s	1.0s	—	—	—	—
2.0s	—	1.0s	—	—	—
3.0s	—	—	1.0s	—	—
4.0s	—	—	—	1.0s	—
5.0s	—	—	—	—	1.0s
6.0s	2.0s	—	—	—	—
7.0s		2.0s	—	—	—
8.0s		—	2.0s	—	—
9.0s		—	—	2.0s	—
10.0s		—	—	—	2.0s
11.0s		3.0s	—	—	
12.0s			3.0s	—	
13.0s			—	3.0s	
14.0s			4.0s	—	
15.0s				4.0s	
16.0s				5.0s	



(b) Now suppose that job D issues an I/O request after each 1.4s it is allowed to run. This I/O request takes 200ms (0.2s) to complete and causes D to block. Now how long will it take for each job to complete? Again, explain your reasoning.

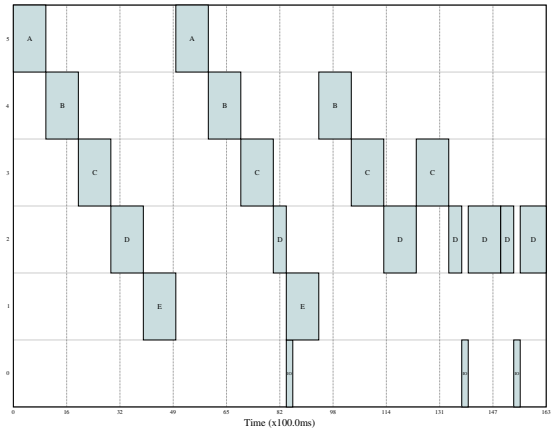
Job	Finish Time
A	*6.0 sec.*
B	*10.4 sec.*
C	*13.4 sec.*
D	*16.4 sec.*
E	*9.4 sec.*

Solution:

D's first IO is masked by E's execution, but for the second two, there is nothing else to do.

Solution

Time End	A	B	C	D	E
0.0s	—	—	—	—	—
1.0s	1.0s	—	—	—	—
2.0s	—	1.0s	—	—	—
3.0s	—	—	1.0s	—	—
4.0s	—	—	—	1.0s	—
5.0s	—	—	—	—	1.0s
6.0s	2.0s	—	—	—	—
7.0s		2.0s	—	—	—
8.0s		—	2.0s	—	—
8.4s		—	—	1.4s	—
9.4s		—	—	—	2.0s
10.4s		3.0s	—	—	
11.4s			3.0s	—	
12.4s			—	2.4s	
13.4s			4.0s	—	
13.8s				2.8s	
14.0s				—	
15.0s				3.8s	
15.4s				4.2s	
15.6s				—	
16.4s				5.0s	



7. (20) Consider the following situation: A very narrow hurricane has washed out all but one lane of the Lake Pontchartrain Causeway[1]. Given that it is a very large lake, going around is impractical, so it is necessary to come up with a system to keep the bridge open. The conditions:

   - Cars arrive at random intervals from either the north or south.
   - The remains of the bridge are only one car wide and cars cannot back up. That is, a car that meets another car is stuck forever.
   - Whenever a car wants to enter the bridge, it calls the function `enter_bridge(int direction)` with a pre-defined integer constant indicating the direction. This will be either `NORTH` or `SOUTH`. When it wants to leave, it calls `exit_bridge(int dir)`.

   Using semaphores and the C-like syntax used for semaphore examples in class and in Tanenbaum and Woodhull, develop a solution to the problem. Implement `enter_bridge()` and `exit_bridge()` and whatever auxilliary data and functions you may need. **Be sure to explain briefly why your solution works.**

   For partial credit: produce a solution that allows cars to cross the bridge without risking meeting another car on the way (and getting stuck forever).

   For more partial credit: produce a solution that guarantees that no car will have to wait forever to cross.

   For full credit: produce a solution that does all of the above and allows multiple cars travelling in the same direction to be on the bridge at a time. (It is 24 miles long, after all.)

   Write your code below (and on page 7 if necessary):

   Solution:

   The first two partial credit levels above can be handled by a very simple solution that uses a single semaphore for exclusive access to the bridge. Exclusive access guarantees that there will be no accidents or deadlocks, and the semaphore's own queueing mechanisms guarantee that nobody will have to wait forever. See below. A better solution is presented on page 7.

```
semaphore mutex;        /* initialized to 1 */

void enter_bridge(direction d) {
  /* make sure nobody else is on the bridge */
  DOWN(mutex);
}

void exit_bridge(direction d) {
  /* indicate that the bridge is free */
  UP(mutex);
}
```

A semaphore-based solution to problem 7 that allows one car at a time.

[1] Lake Pontchartrain is a lake 41 miles long and 24 miles wide north of New Orleans, La. The causeway is the bridge that spans the "short" direction and is one of the two roads out of the city.

Extra space for problem 7

The full solution is much trickier: To meet all three requirements, we must not allow traffic in one direction to hold up traffic in the other direction forever, but we can't just take turns, because a car may never arrive on the other side, and we still have to allow multiple cars to cross in the same direction.

The technique is to check to see if there are cars on the other side before entering the bridge. If there are, we act as if they were already on the bridge and wait. The last car crossing in the current direction wakes up all the waiting cars on the other side.

The only subtle part if the solution is the question of what happens if `enter_bridge()` *is interrupted at line 41 (marked at right). Could it cause problems if a process were to decide to wait until cars have cleared the bridge, but be interrupted before it an down* `waitsem`*?*

Consider: The only way a car(C) could decide to wait is if there is at least one car on the bridge, and before the C releases `mutex`*, it has already been added to the count of waiting cars. So, the worst that could happen during an interruption is that the only car on the bridge could leave. On its way out, however, it would put all waiting cars onto the bridge and up* `waitsem` *the appropriate number of times. So, all that will happen is that when C does get to run and downs* `waitsem`*, the operation will succeed without blocking.*

```
typedef int direction;
#define NORTH 0
#define SOUTH 1

semaphore mutex;              /* initialized to 1 */
semaphore waitsem[2];         /* initialized to 0,0 */

direction bridgedir = NORTH;   /* direction cars are currently moving   */
                               /* how it is initialized doesn't matter  */
int    onbridge    = 0;        /* there are no cars initially on the bridge */   10
int    waiting[2]  = {0,0};    /* nobody is waiting in either direction */


direction otherdir(direction d) {
  direction ret;
  if ( d == NORTH )
    ret = SOUTH;
  else
    ret = NORTH;
  return ret;                                                                    20
}

void enter_bridge(direction d) {
  direction other = otherdir(d);
  int willwait = 0;

  DOWN(mutex);
  if ( onbridge == 0 ) {        /* there's nobody on the bridge, take it */
    onbridge     = 1;
    bridgedir    = d;                                                           30
    waiting[d]   = 0;
  } else if ( (onbridge > 0) &&         /* if somebody's on the bridge, */
              (bridgedir == d) &&       /* it's going our way, and nobody's */
              (waiting[other] == 0) ) { /* waiting on the other side */
    onbridge  = onbridge + 1;           /* add ourselves to the bridge */
  } else {
    waiting[d] = waiting[d] + 1; /* otherwise, we have to wait. */
    willwait = 1;
  }
  UP(mutex);                                                                     40
  /*  HERE HERE HERE */
  if ( willwait )               /* if we have to wait, wait */
    DOWN(waitsem[d]);
}

void exit_bridge(direction d) {
  /* indicate that the bridge is free */
  direction other = otherdir(d);

  DOWN(mutex);                                                                   50
  onbridge = onbridge−1;        /* decrement the counter */
  if ( onbridge == 0 ) {        /* if we were the last */
    if ( waiting[other] != 0 ) {
      /* if someone is waiting to go the other way, put them
        * on the bridge, then wake those cars up
        */
      onbridge       = waiting[other];
      bridgedir      = other;
      waiting[other] = 0;
                                                                                 60
      for(i=0;i<waiting[other];i++) { /* wake up the waiting cars */
        UP(waitsem[other]);
      }
    }
  }
  UP(mutex);
}
```