Midterm Exam

CS460: Operating Systems

19 October, 1995

Jim Plank

Question 1: 15 Points

Part 1

This question concerns interrupt-driven I/O. Specifically, explain what is involved in each of the following:

1. Reading from a byte-serial device like the console.
2. Writing to a byte-serial device like the console.
3. Reading from a DMA device like a disk.
4. Writing to a DMA device like a disk.

Part 2

A colleague makes the following assertion: "DMA enables a system to handle faster I/O devices." Explain what he/she means.

Question 2: 15 Points

Explain what a long-term scheduler and a short-term scheduler do. For each, explain:

- What kind of decisions does it make?
- When must it make decisions?
- What factors are important for its decision and what does it do to ameliorate those factors?

Question 3

You are programming on a multiprocessor system using threads. The system includes monitors and condition variables, with primitives exactly like those in the SunOS lightweight process library:

```
#include <lwp.h>

mon_create(mon_t *m)           /* Creates a new monitor */
mon_enter(mon_t m)             /* Enters the monitor */
mon_exit(mon_t m)              /* Exits the monitor */

cv_create(cv_t *cv, mon_t m)   /* Creates a condition variable associated
                                  with the monitor m */
cv_wait(cv_t cv)               /* Blocks on the condition variable */
cv_notify(cv_t cv)             /* Wakes up on thread waiting on cv if there
                                  is such a thread */
```

Implement general semaphores under this system. In other words, write the file **gsem.c**, which implements the following procedures:

```
typedef void *Gsem;

Gsem gsem_create(int initval); /* Create and return a semaphore with
                                  an initial value of initval */
gsem_P(Gsem g);                /* Call P() on the semaphore */
gsem_V(Gsem g);                /* Call V() on the semaphore */
```

Question 4

You are programming on a threads system with general semaphores that conform to the syntax given in question 3. Look at the following program, where the thread creation/manipulation primitives are:

```
thread_t thread_fork(void (*proc)(), void *arg)     /* Create a thread running procedure proc
                                                       called on the argument arg.
                                                       Returns a pointer to the thread */

thread_join(thread tid)                             /* Waits for thread t to return */

thread_t thread_self()                              /* Returns a pointer to the calling thread */
```

```
#include "gsem.h"
#include "thread.h"

Gsem mut;
Gsem sem;
thread_t T[10], done[10];
int ndone;

proca(void *v)
{
  int i;

  i = (int) v;

  printf("I am thread %d\n", i);

  gsem_P(mut);
  done[ndone] = T[i];
  ndone++;
  gsem_V(mut);
  gsem_V(sem);
  return;
}
```

```
main()
{
  int i;

  sem = gsem_greate(0);
  mut = gsem_greate(1);
  ndone= 0;

  for (i = 0; i < 10; i++) {
    T[i] = thread_fork(proca, (void *) i);
  }

  for (i = 0; i < 10; i++) {
    gsem_P(sem);
    thread_join(done[i]);
  }

  printf("All done\n");
}

.
```

Part A

On a non-preemptive uniprocessor:

1. What is the output of the program?
2. Do we need the semaphore **mut**? Why or why not?
3. Do we need the semaphore **sem**? Why or why not?

Part B

On a multiprocessor threads system, there are two potential trouble spots for this program. Identify them and fix them.

Question 5

You are programming on a multiprocessor system with threads where the test and set function is supported with the following syntax:

```
int test_and_set(int *key)
```

You are writing the procedure `do_stuff()`, which has the following skeleton:

```
static int key = 0;

do_stuff()
{
  /* ... */

  do_body();

  /* ... */
}
```

Part A

Fill in the parts marked `/* ... */` so that the procedure `do_body()` is never executed by two threads simultaneously. There can be any number of threads in the system.

Part B

Given your solution, explain how a thread can be starved by calling `do_stuff()`. Is this more likely to happen if `do_body()` is short or long?