# Midterm Exam – Answers and grading

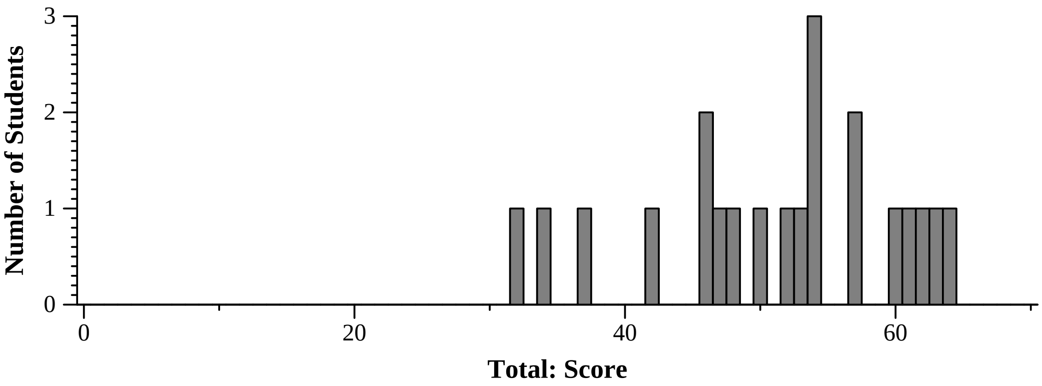
# CS460: Operating Systems

# 19 October, 1995

# Jim Plank

## Overall stuff

So, this test looked ok to me — enough to fill up the time (75 minutes – 15 per question), but not too much. The overall histogram of test scores is as follows:




I would break this down into grades as follows:

- 60 and over: Strong A
- 52 to 59: B+ / A-
- 45 - 51: B
- 35 - 44: C+
- 30 - 34: C

## 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.

### Answer: Part 1

With all interrupt-driven I/O, the I/O devices are connected to the CPU and memory by the memory bus. At certain times, the device will interrupt the CPU, which saves its state and executes from a known location specified by a region of memory known as the *interrupt vector*. Once the interrupt has been processed, the machine can continue running as it was before.

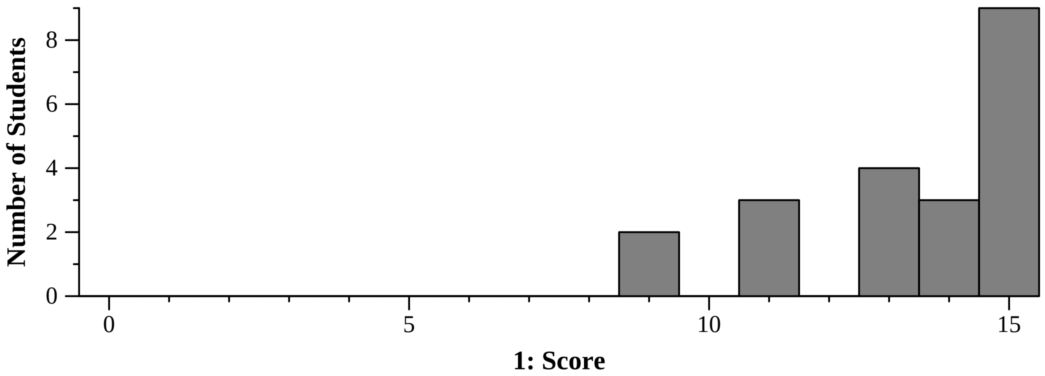
1. When the console has a byte to read, it interrupts the CPU, and then byte is read over the memory bus from the device to the CPU, and then from the CPU to memory. When the console has another byte to read, it interrupts the CPU again. Note that the console might produce bytes before a user process requests them. In such a case the operating system should buffer them.
2. To write a byte, the operating system must wait for the console to be idle. (The console starts in an idle state). It then sends the character to the device, and alerts the device to the existance of that byte. The console is considered busy now processing the character until it interrupts the CPU. The interrupt specifies that the device is again idle.
3. To read a block of data using DMA, the operating system tells the device what data to read, and tells it a location in memory into which to place the data. The device then sends the data along the memory bus, and the data goes directly to memory without any CPU interaction. Once the transfer is done, the device interrupts the CPU.
4. Writing a block of data to a device is just like reading, except the transfer goes from memory to the device, instead of from the device to memory.

### Answer: Part 2

In the absense of DMA, the CPU must be involved in the transfer. For each byte (or perhaps word) of data there must be an interrupt, and two transactions along the memory bus (device to CPU, and CPU to memory). With DMA, the only limiting factor is the speed of the device, the speed of memory, and the capacity of the memory bus. Obviously, given the same system parameters, DMA will support faster devices than using the CPU to do the transfer. There could well be devices for which byte-serial I/O would be too slow to get all the bytes from the device, but for which DMA is easily adequate.

### Grading

**Part 1:** Each question is worth 2.5 points. You had to mention interrupts to receive full credit (i.e. you couldn't just say "you read bytes from a byte-serial device one byte at a time").
**Part 2:** 5 points. To receive full credit you needed to mention that the CPU might not be able to keep up with a fast byte-serial I/O device. If you didn't mention this and instead just said that the CPU is freed up by DMA, 2 points were deducted.




## 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?

### Answer

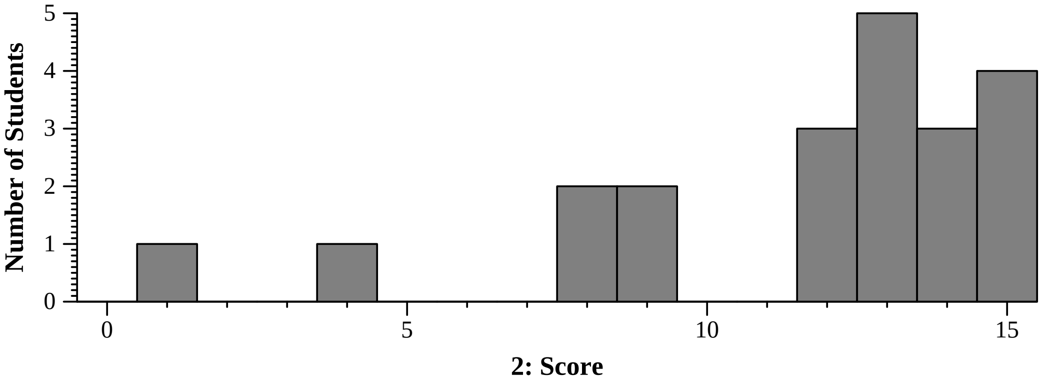
The long-term scheduler is responsible for putting jobs into the system. When a job is first submitted to the system, the long-term scheduler has a choice of loading it into memory and making it available for execution, or storing it on disk to execute later. The long-term scheduler makes decisions when jobs enter and exit the system. It might also make decisions when the CPU is idle too often. The primary responsibility of the long-term scheduler is to keep a good blend of I/O and CPU-bound jobs in the system. The objective is to maximize throughput and CPU utilization. It also must keep track of how much physical memory has been doled out to executing jobs, and make sure that the combined jobs don't need more memory than there is.

The short-term scheduler controls which process of the ones in memory gets the CPU and for how long. It gets executed whenever a job voluntarily gives up the CPU (for preemptive and nonpreemptive schedulers), and whenever the operating system takes control of the CPU via an interrupt (for preemptive schedulers only). The factors that are important to short-term scheduling are CPU Utilization, throughput, process turnaround time, and response time. Since the scheduler is called often, it cannot be too complex. Some schedulers try to approximate shortest job first for maximal throughput/minimal turnaround time. They use job priorities or scheduling queues, and often employ system parameters to determine the individual priorities, or in which queues jobs go. For applications where response time is important, a timer interrupt is employed so that all processes can have the CPU for frequent, small time slices.

## Grading

Here's the point break-down:

- Long-term scheduler
  - What does it do? — 2 points
  - When does it get called? — 2 points
  - What does it care about? — 2 points
  - How does it achieve this (by keeping a good mix of jobs in memory) – 1.5 points
- Short-term scheduler
  - What does it do? — 1.5 points
  - When does it get called? — 1.5 points
  - What does it care about? — 1.5 points
  - How does it achieve this (approximating sjf, using time-slicing) – 3 points



## Question 3: 15 Points

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 */
```

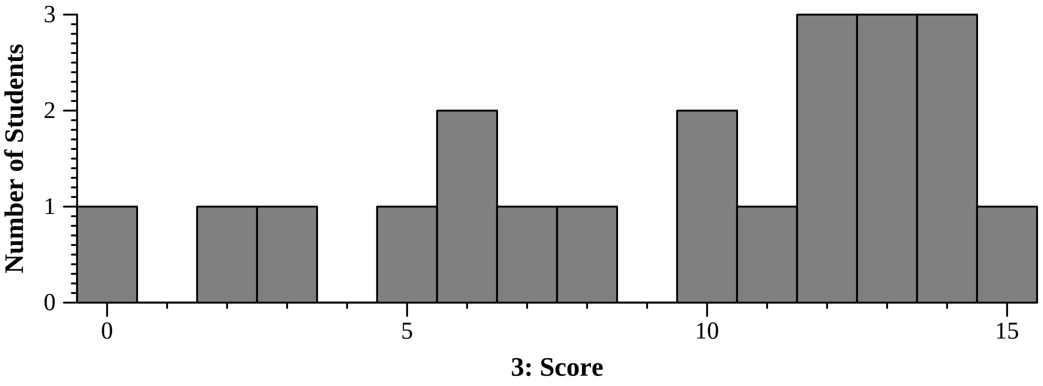
### The Answer

You simply need a monitor and a condition variable. The monitor protects access to the semaphore's value, and the condition variable lets you block/wakeup threads.

Some of you tried to use the binary semaphore solution from the book, substituting monitors with binary semaphores. The problem with that is that one thread cannot exit a monitor that another thread has entered. This was a deduction of 4 points.

Note too the $< 0$ in **gsem_P** and $\leq 0$ in **gsem_V**. You can argue that the conditional in **gsem_V** is unnecessary, for if the value is greater than zero, there will be no threads waiting on the conditional varible.

```
typedef struct {
  mon_t m;
  cv_t c;
  int i;
} gsem;

void *gsem_create(int initval)
{
  gsem *g;

  if (initval < 0) return NULL;

  g = (gsem *) malloc(sizeof(gsem));

  mon_create(&(g->m));
  cv_create(&(g->c), g->m);
  g->i = initval;
  return g;
}

gsem_P(void *v)
{
  gsem *g;

  g = (gsem *) v;
  mon_enter(g->m);
  g->i--;
  if (g->i < 0) cv_wait(g->c);
  mon_exit(g->m);
}

gsem_V(void *v)
{
  gsem *g;

  g = (gsem *) v;
  mon_enter(g->m);
  g->i++;
  if (g->i <= 0) cv_notify(g->c);
  mon_exit(g->m);
}
```




## Question 4: 15 Points

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.

## Answer

**Part A / 1**:

```
I am thread 0
I am thread 1
I am thread 2
I am thread 3
I am thread 4
I am thread 5
```

```
I am thread 6
I am thread 7
I am thread 8
I am thread 9
All done
```

**Part A / 2**: On a non-preemptive uniprocessor, `mut` is unnecessary, because no thread will be preempted between the `gsem_P(mut)` and `gsem_V(mut)` calls.

**Part A / 3**: You can answer either yes or no to this as long as you give the right reason. The answer is yes if you don't want to change the program further. Otherwise, the main thread (the one that makes all the `thread_fork` calls) will call `thread_join(done[i])`, but `done[i]` will not have been initialized yet. This would be an error (probably a core dump, but maybe just deadlock). The answer can be no if you say that none of this is really necessary because the program doesn't gain you anything on a non-preemptive uniprocessor system: you should just write a normal non-threaded C program. It was not acceptable to say that `sem` was unnecessary because the system is non-preemptive – on a standard non-preemptive uniprocessor system like SunOS threads, we can remove `mut` without altering the program, but the program will not work if you remove `sem`.

**Part B**: The first trouble spot is the `printf` call, because `printf` writes to global memory, and in a preemptive thread system, leaving that call unprotected can lead to errors. The solution to this problem is to guard the `printf` call with another semaphore (or `mut` for that matter), or to use `sprintf/write` instead of `printf`.

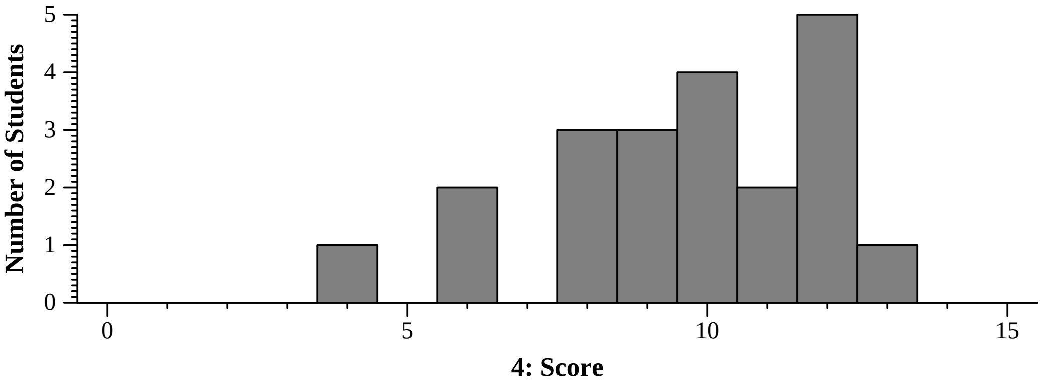
The second trouble spot is subtle. In a non-preemptive thread system, a thread can be created and start running before the forking thread returns from the `thread_fork` call. Therefore, it is quite possible that a thread will reach the `done[ndone] = T[i]` line before the main thread's `thread_fork()` call has returned. This makes `T[i]` an uninitialized variable, and again can get you in trouble. A solution to this would be to make sure that all threads are blocked until the main one has done all its `thread_fork()` calls. E.g.:

```
Gsem start;

proca(void *v)
{
  int i;

  gsem_P(start);

  ...
}

main()
{
  int i;

  start = gsem_create(0);
  ...
  for (i = 0; i < 10; i++) {
    T[i] = thread_fork(proca, (void *) i);
```

```
  }
  for (i = 0; i < 10; i++) { gsem_V(start); }
 ...
}
```

### Grading

**Part A / 1**: 3 points. You lost a point if you said the order may vary – the system is preemptive, so you know exactly which threads get called when.
**Part A / 2**: 3 points.
**Part A / 3**: 3 points.
**Part B**: 6 points – 3 for the first problem, and 3 for the second.




## Question 5: 10 Points

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?

## The Answer

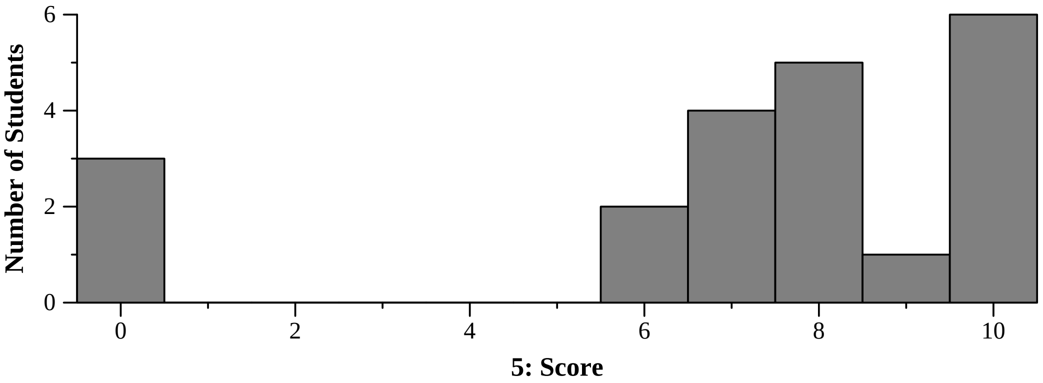
### Part A

```
do_stuff()
{
  while (test_and_set(&key)) ;

  do_body()

  key = 0;
}
```

You can have starvation if there are more than two threads calling `do_stuff()` at a time. Since there is no fairness scheme built into the `test_and_set()` primitive, it could well happen that if you have three threads, A, B, and C repeatedly calling `do_stuff()`, then threads A and B can alternate getting the key while thread C remains starved. This is more likely to happen if `do_stuff()` is long because then it is more likely that there will be multiple threads waiting for entrance to `do_stuff()`.

## Grading

Each part is worth 5 points. Some of you said that it will happen more if `do_stuff()` is small, because then it's more likely that the threads will get back to calling `do_stuff()` again. Actually, what you meant was that it's more likely to happen if the interval between the end of `do_stuff()` and the next call to `do_stuff()` is small. That's true, but that's not what the question is asking.


Number of Students
0
2
4
6
0
2
4
6
8
10
5: Score