CSE 120 Operating Systems
May 11, 2000 – Midterm Exam

1. (**1 point**) Which of the following instructions should be allowed only in kernel (or supervisor) mode? Why?

   (a) Read the time-of-day clock

   This can be executed in the user mode since reading the clock does not affect other processes.

   (b) Disable all interrupts

   This instruction should be executed only in the supervisor mode since execution of this instruction potentially affects all the processes.

2. (**1 point**) Consider the following solution for mutual exclusion. Before entering the critical section, the processes execute the following.

```
void enterCriticalSection(int pid) {
   interested[pid] = 1;
   while (interested[1-pid]);
}
```

   (Assume that the pids of the processes are 0 and 1. The array `int interested[2];` is visible to both the processes.) Does this ensure mutual exclusion? Why?

   The above solution can result in *neither* process entering the critical section. The following sequence of executions results in this condition.

```
       process 0                             process 1
   interested[0] = 1
// process 0 preempted
                                          interested[1] = 1
                                          while (interested[0]); // BUSY WAIT
// process 0 resumes execution
   while (interested[1]); // BUSY WAIT
```

3. (**1 point**) In the previous question, how can the array `interested` be made visible to both the processes?

   The array should be allocated using shared memory techniques. The memory is allocated in the kernel address space. The following is a typical code fragment for shared memory allocation.

```
int *interested, shmid;
shmid = shmget(IPC_PRIVATE, 2*sizeof(int), 0666|IPC_CREAT);
interested = (int *) shmat(shmod, NULL, 0);
```

4. (**1 point**) Consider a computer that does not have a TEST AND SET LOCK instruction, but does have an instruction to swap the contents of a register and a memory word in a single indivisible action. How can this instruction be used to implement mutual exclusion?

```
L1: set register R to 1; // enter critical section
    swap(R, M);
    if R is non-zero, jump to L1;
/* critical section */
    set R to 0; // leave critical section
    swap(R, M);
```

5. (**3 points**) A Multiple semaphore allows the `down` and `up` primitives to operate on several semaphores simultaneously. It is useful for acquiring and releasing several resources in one *atomic* operation. Thus the `DOWN` primitive (for two semaphores) can be defined as follows:

```
void DOWN(u,v) {
    while (u <= 0 or v <=0);
    u--;
    v--;
}
```

How can a multiple semaphore be implemented using regular semaphores? You should give implementations for `DOWN(u)`, `DOWN(u,v)`, `UP(u)`, and `UP(u,v)`.

Let's first fix the data structures.

(a) There are two integers `u` and `v` – each representing the state of the semaphore.

(b) Since these variables are shared, they are protected by a semaphore `mutex`.

(c) Each process has its own semaphore on which it blocks. So there is an array of semaphores, `sem[NPROCESS]`.

(d) Finally, there are three queues: `UQ`, `VQ`, and `UVQ` of processes waiting for each of the semaphores.

Then `DOWN(u)`, `UP(u)`, and `DOWN(u,v)` can be implemented as

```
semaphore mutex=1, sem[NPROCESS];
int u=1, v=1;
pid *UQ, *VQ, *UVQ;

void DOWN(u) {
```

```
        down(mutex);
        if (u==0) {
            enter PID in UQ;
            up(mutex);
            down(sem[PID]);
        }
        u = 0;
    }

    void UP(u) {
        down(mutex);
        u++;
        if ((u==1)&&(v==1)&&(UVQ!=NULL)) {
            u--;
            v--;
            choose PID from UVQ;
            up(sem[PID]);
        } else if ((u==1)&&(UQ!=NULL)) {
                /* choose a process from UQ */
        }
        } else if ((v==1)&&(VQ!=NULL)) {
                /* choose a process from VQ */
        }
        else {
            up(mutex);
        }

    void DOWN(u, v) {
        down(mutex);
        if ((u==0)||(v==0)) {
            enter PID in UVQ;
            up(mutex);
            down(sem[pid]);
        }
        u--;
        v--;
    }
```

6. (**2 points**) Describe how event counters can be used to implement semaphores.

   For each semaphore, we keep two event counters – `x` and `y`. `x` counts the number of `downs` and `y` the number of `ups` on the semaphore.

```
void up(u) {
```

```
   advance(y);
}

void down(u) {
   await(y, x+1); // wait for y >= x+1
   advance(x);
}
```

7. (**1 point**) Consider the following code fragment of a producer thread using mutex and condition variable.

```
pthread_mutex_lock(&mutex);
if (full) pthread_cond_wait(&cvFull, &mutex);
/* access buffer */
```

(The variable `full` is `1` if the buffer is full. `cvFull` is the corresponding condition variable.) Is the above code fragment correct? If not, why?

This does not work if there are multiple producers. It is possible that more than one producer is awakened (after `wait`). In this case, the queue is likely to be full for most of the awakened producers.

To solve this, the `if` statement should be replaced by `while`.

8. (**1 point**) What is a relocatable program? What are the advantages of relocatable programs?

A relocatable program can execute independently of its physical location in memory. Usually relocatable programs do not use absolute addresses – they use only relative addresses. So they can be loaded anywhere in memory.

9. (**1 point**) Explain the difference between internal and external fragmentation.

External fragmentation results when there is sufficient memory to service a request but the memory is divided into holes.

Some memory allocation systems allocate memory in fixed sizes (say, powers of two). In this case, the difference between the allocated size and the requested size is the memory lost in internal fragmentation.

10. (**1 point**) Assume that a computer system has memory of size 1 MB. If the average process size is 100 KB and the average hole size is 50 KB, what is the size of memory wasted in fragmentation?

The ratio between hole size and process size is $50/100 = 0.5$. By *unused memory rule*, the fraction of memory wasted in fragmentation is $0.5/(2+0.5) = 0.2$. The total memory wasted in fragmentation is $0.2 \times 1MB = 200KB$.

11. (**2 points**) A buddy allocator starts with memory of size 256K at address 0. After successive requests for 5K (A), 25K (B), 35K (C), and 20K (D), how many blocks are left and what are their sizes and addresses? If memory is freed in the order B, C, A, D show how the blocks are merged.

After allocation of A

A	8	16	32	64	128

After allocation of B, C, D

A	8	16	B	C	D	32	64

After freeing B, C

A	8	16	32	64	D	32	64

After freeing A

128	D	32	64

After freeing D

256