20-EECE-4029/4069
20-ELTN-4022

Operating Systems **Fall, 2013**

First Exam name: _____

Question 1: Knowledge of OS principles/objects

Say something significant about each of the OS principles and objects stated below

1. The principle of locality (why crucial?) - due to the high cost of fast memory, a memory hierarchy is necessary to keep computers useful and affordable. Since managing data and instructions between memory levels is costly in time, performance would be poor if such moves were not rare. Locality infers the moves are rare in a well designed system.

2. Spin lock (why useful?) - mutex locks and semaphores require that a running thread enter a wait state. This means removing the thread from the ready queue and performing a context switch. With spin locks, a thread does not have to go to sleep so that overhead is saved.

3. Work queue (why important?) - code that is executed in kernel modules is usually run with interrupts turned off. If such code results in a call to a function that interrupts itself (common with device drivers - we saw this with ACPI calls) it will hang. The solution is to place such work into a work queue where the OS will run it when it is ready to and with interrupts turned on.

4. Memory barrier (why needed?) - modern CPUs with several cores are capable of executing instructions out of sequence to improve performance, provided certain rules are obeyed. Sometimes delays in the availability of data cause such executions to give the wrong results. A memory barrier is a mechanism that causes execution to wait until all needed data is updated.

5. Co-routine (why useful?) - in a multi-threaded environment with huge speed differences between components, queues are necessary for reliable transfer of work from a producer (say an application) to a consumer (say a device driver) and back. Co-routines facilitate such transfers.

Question 2: Kernel Thread Troubles

What's up (or down) with this code and say why?

There is no waiting by any thread - some thread is going to go full speed through the monitor and probably hang the computer.

Pencil in statements that fix it, show where they belong with arrows (shown in red)

```
struct semaphore empty;
struct semaphore full;
struct semaphore mutex;
int value;

int get(void) {
    int ret;
    if (down_interruptible(&full)) return -1;
    if (down_interruptible(&mutex)) return -1;
    ret = value;
    up(&mutex);
    up(&empty);
    return ret;
}

int put(int v) {
    if (down_interruptible(&empty)) return -1;
    if (down_interruptible(&mutex)) return -1;
    value = v;
    up(&mutex);
    up(&full);
    return 0;
}

int successor (void *unused) {
    int i;
    for (i=1 ; ; i++) { put(i); }
    return 0;
}

int consumer (void *unused) {
    int i;
    for (i=0 ; i < 10 ; i++) printk("%d ", get());
    return 0;
}

int init_module () {
    struct task_struct *s1, *c1;
    sema_init(&mutex, 1);
    sema_init(&empty, 1);
    sema_init(&full, 0);
    s1 = kthread_run(successor, NULL, "suc");
    c1 = kthread_run(consumer, NULL, "cons");
    return 0;
}

void cleanup_module () { }
```

Question 3: Userland Thread Control

What is the strongest true statement that can be made about the numbers output?

The numbers 1,2,3 follow the numbers 4,5,6

What is the difference in execution between `selector=true` and `selector=false`?

If `selector=true` the `W` threads are allowed to run simultaneously, otherwise only one `W` thread is allowed to run at a time.

```
int n = 3, count = 3;
sem_t avail, protect, mutex;
bool selector;

void *W (void *id) {
   sem_wait(&avail);
   usleep(rand() % 10000);  /* sleep for random time */
   printf("%d ", (int)id);
   if (!selector) sem_post(&avail);
   pthread_exit(NULL);
}

void *N (void *id) {
   int i;
   usleep(rand() % 10000);  /* sleep for random time */
   printf("%d ", (int)id);
   sem_wait(&protect);
   count--;
   if (count == 0 && !selector) sem_post(&avail);
   else if (count == 0 && selector) {
      sem_wait(&mutex);
      for (i=0 ; i < n ; i++) sem_post(&avail);
      sem_post(&mutex);
   }
   sem_post(&protect);
   pthread_exit(NULL);
}

int main (int argc, char **argv) {
   if (!strcmp(argv[1],"true")) selector=true; else selector=false;
   pthread_t w1, w2, w3, n1, n2, n3;
   srand(time(NULL));
   sem_init(&avail, 0, 0);
   sem_init(&protect, 0, 1);
   sem_init(&mutex, 0, 1);
   pthread_create(&w1, NULL, W, (void*)1);
   pthread_create(&w2, NULL, W, (void*)2);
   pthread_create(&n1, NULL, N, (void*)4);
   pthread_create(&n2, NULL, N, (void*)5);
   pthread_create(&w3, NULL, W, (void*)3);
   pthread_create(&n3, NULL, N, (void*)6);
   pthread_exit(NULL);
}
```

Question 4: Scheduling Algorithms

1. Describe how a process scheduler uses a multi-level feedback queue

   Lots of process-ready FIFOs with level numbers 1,2,...N-1 and a Round Robin queue at level N. Processes at lower level numbers have higher priority. A process entering the MFQ is placed at the end of the lowest level FIFO. When it runs, if it does not complete after a certain time interval it is placed at the end of the FIFO at the next higher level. This keeps happening until it completes or enters the RR queue where it will stay until it finishes. A process waiting too long in a higher level may be moved to a lower level.

2. Start with an empty Round Robin queue. Suppose 4 jobs become ready (enter the queue) at the same time, and are placed in the queue in no particular order. The total number of cycles needed to complete each is given in the table below:

job	**cycles needed**
1	16
2	32
3	8
4	24

   Assume a time slice consists of exactly 12 cycles.
   What is the number of context switches needed for the RR queue to complete all jobs (must be an integer)?

   1:12,4(2) 2:12,12,8(3) 3:8(1) 4:12,12(2) ans: 8

   What is the minimum possible number of time slices needed to complete all jobs (must be an integer)?

   ceiling((16+32+8+24)/12) ans: 7 but 4 cycles are unused

   Explain the difference

   total wasted cycles is 8+4+4+0 = 16 = 12 + 4 (the 12 is the extra time slice)