20-EECE-4029   Operating Systems   Spring, 2013   John Franco

First Exam   name: ______________________

Question 1: Semaphores

What is the cause of uncertainty in the output? both threads may have passed the `idx` test which means it is possible that an undefined array location could be accessed.

How can the code be fixed to produce the expected output? protect `idx` (see below)

```
#define LOOPS 100000

int list[LOOPS], idx = 0, cs1 = 0, cs2 = 0;
struct task_struct *t1, *t2;
struct semaphore lock;

int consumer(void *ptr) {
   printk(KERN_INFO "Consumer TID %d\n", (int)ptr);
   while (1) {
      if (!down_interruptible(&lock)) {
         if (idx >= LOOPS) {
            up(&lock);
            break;
         }
         list[idx++] += 1;
         up(&lock);
         if ((int)ptr == 1) cs1++; else cs2++;
      }
   }
   printk(KERN_INFO "Consumer %d done\n",(int)ptr);
   return 0;
}

int init_module (void) {
   int i, id1 = 1, id2 = 2;
   for (i = 0; i < LOOPS; i++) list[i] = 0;
   sema_init(&lock, 1);
   t1 = kthread_create(consumer, (void*)id1, "cons1");
   t2 = kthread_create(consumer, (void*)id2, "cons2");
   if (t1 && t2) {
      printk(KERN_INFO "Starting..\n");
      wake_up_process(t1);
      wake_up_process(t2);
   } else {
      printk(KERN_EMERG "Error\n");
   }
   msleep(2000);
   printk(KERN_INFO "cnt=%d\n",cs1+cs2);

   return 0;
}

void cleanup_module(void) {  printk(KERN_INFO " Inside cleanup_module\n");  }
```

Question 2: Knowledge of terms related to OS

Say something noteworthy about the following

1. OS mitigation of side-channel attacks on cryptosystems
   Side channel attacks exploit properties of modern hardware. For example, an attack might force a cache hit when a cryptoalgorithm sees a 1 keybit (and not a 0). The extra time for the hit can be observed by an attacker. Since caching (and control of branch prediction and other hardware) is the domain of the OS, the OS can be designed to mitigate the effects (e.g. by introducing random delay, by requiring no cache misses (large contiguous memory segments) for cypto algorithms and so on.

2. Why there are so many process scheduling algorithms
   There are many parametric tradeoffs to consider. For example: latency, throughput, response time, and so on.

3. Work queue
   Support the concept of deferred work. When an interrupt is generated it is usually handled by a "top half" segment which sets up a complicated task that must be completed. That task has no restrictions on what it requires (for example, it is probably interruptible) and usually does not have to be done immediately. So it is put into a work queue as a "bottom half." The OS can decide when to pull it from the work queue and execute it.

4. Role of producer-consumer processes in operating systems
   Work queues may be filled by several processes running concurrently. Tasks in the work queue may be pulled and executed alongside those processes. This is the classic producer-consumer relationship with the processes as the producers and the executor of work queue tasks as the consumer.

5. Why there are typically I/O channels
   I/O is much slower than system bus speeds. I/O is handed off to I/O controllers that issue commands to I/O devices and perform I/O independently of the processor, hence avoiding processor wait states. The result of I/O operations is placed in or taken from memory by Direct Memory Access.

6. Why some processors use microcode
   A main advantage of microcode is that it can be changed without the need for changing compiled code. Hence a bug found in the field can be fixed with relatively few side-effects.

Question 3: Thread problems

What is likely to happen when this executes? threads will hang

What can be done to fix this? see below

```
int count = 0;
double finalresult=0.0;
pthread_mutex_t count_mutex;
pthread_cond_t count_condvar;

void *sub1(void *t) {
   long tid = (long)t;
   double myresult=0.0;
   sleep(1);
   pthread_mutex_lock(&count_mutex);
   if (count < 12) {
      pthread_cond_wait(&count_condvar, &count_mutex);
      count++;
      finalresult += myresult;
   }
   pthread_mutex_unlock(&count_mutex);
   pthread_exit(NULL);
}

void *sub2(void *t) {
   long tid = (long)t, i, j;
   double myresult=0.0;
   for (i=0 ; i < 10 ; i++) {
      for (j=0 ; j < 100000 ; j++) myresult += sin(j)*tan(i);
      pthread_mutex_lock(&count_mutex);
      finalresult += myresult;
      count++;
      if (count == 12) pthread_cond_signal(&count_condvar);
      pthread_mutex_unlock(&count_mutex);
   }
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   int i, rc;
   long t[] = { 1, 2, 3 };
   pthread_t threads[3];
   pthread_mutex_init(&count_mutex, NULL);
   pthread_cond_init (&count_condvar, NULL);
   pthread_create(&threads[0], NULL, sub1, (void*)t[0]);
   for (i=1; i<3; i++) pthread_create(&threads[i], NULL, sub2, (void*)t[i]);
   for (i = 0; i < 3; i++) pthread_join(threads[i], NULL);
   printf ("Result=%e\n", finalresult);
   pthread_mutex_destroy(&count_mutex);
   pthread_cond_destroy(&count_condvar);
   pthread_exit (NULL);
}
```

Question 4: Memory hierarchy

1. Why is there a memory hierarchy to worry about?
   Memory devices that have low cost per bit are slow and devices that are fast have high cost per bit. Hence, it is economical to have small amounts of fast memory to directly support processor operations, feed that fast memory from some intermediate, slower but cheaper and larger memory, feed that memory from even slower and even cheaper very large memory and so on.

2. Improve the spatial locality of this code:

```
int main () {
   int arr[100], i, k=0;
   for (i=0 ; i < 100 ; i++) {
      arr[k%100] = 9;
      k += 11;
   }
}
```

```
int main () {
  int arr[100], i;
  for (i=0 ; i < 100 ; i++) arr[i] = 9;
}
```

3. Why do memory hierarchies work?
   The principle of locality makes cache misses relatively rare.

Question 5: Memory barriers in device drivers

What does `barrier` do in the following (that is, what could go wrong if `barrier` were not put into this statement)?

```
static inline void incr_bp(volatile unsigned long *index, int d) {
   unsigned long new = *index + d;
   barrier();
   *index = (new >= (buffer + PAGE_SIZE)) ? buffer : new;
}
```

The compiler might optimize the two lines into one like this:
`*index = (*index + d >= (buffer + PAGE_SIZE)) ? buffer :  *index + d;`
which, if interrupted, might lead to an unpredictable value for `index`.

Why is `rmb` put into the following piece of code?

```
ssize_t readIt (char *buf, size_t count) {
   int retval = count;
   unsigned long port = 0x378;
   unsigned char *kbuf = kmalloc(count, GFP_KERNEL);
   ...
   insb(port, kbuf, count);
   rmb();
   ...
   return retval;
}
```

To make sure the port is fully read before continuing to process the result.

The following snippet is taken from a Mellanox Infiniband HCA low-level driver. The object `sq` is a work queue whose `head` is `sq.head`. The object `sq.db` is a "doorbell" interrupt that is used to notify the device that some work is to be done. The call to `mthca_write64` raises the interrupt. Put `wmb` or `rmb` call(s) wherever you think they should go, if at all, and say why.

```
   qp->sq.head += MTHCA_ARBEL_MAX_WQES_PER_SEND_DB;
   wmb();  /* set up the doorbell record first */
   *qp->sq.db = cpu_to_be32(qp->sq.head & 0xffff);
   wmb();  /* translate doorbell record to hw format */
   mthca_write64(dbhi, (qp->qpn << 8) | size0,
                 dev->kar + MTHCA_SEND_DOORBELL,
                 MTHCA_GET_DOORBELL_LOCK(&dev->doorbell_lock));
   /* finally ring the doorbell */
```