20-EECE-4029 Operating Systems Spring, 2015 John Franco

First Exam name: ______________________________

Question 1: Semaphores

The code below produces a line consisting of 10 each of the letters `A`, `B`, and `C`. What other relationship of the occurrence of these letters in the output is enforced by the semaphore?

```
#include <pthread.h>
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <unistd.h>
#include <semaphore.h>

#define X 3
sem_t avail;

void waitIt (void *id) {  sem_wait(&avail);  }

void moveItW (void *id) {  sem_post(&avail);  }

void *W (void *id) {
   printf("A "); fflush(stdout);
   waitIt(id);
   printf("B "); fflush(stdout);
   sleep(1);

   moveItW(id);
   printf("C "); fflush(stdout);
   pthread_exit(NULL);
}

int main () {
   int i;
   pthread_t w[10];
   int id[10];

   for (i=0 ; i < 10 ; i++) { id[i] = i+1; }

   sem_init(&avail, 0, X);

   for (i=0 ; i < 10 ; i++)
      pthread_create(&w[i], NULL, W, (void*)&id[i]);
   for (i=0 ; i < 10 ; i++) pthread_join(w[i], NULL);
   printf("\n");
   pthread_exit(NULL);
}
```

Answer:

At any time, no more than three threads can have passed through `waitIt()` yet not through `moveIt()`. Therefore, from left to right up to any point in the output list of characters, the number of `B` characters is never greater than the number of `C` characters plus three and there are never fewer `B` characters than `C` characters.

Question 2: Knowledge of terms related to OS

Say something noteworthy about the following

1. kernel/interrupt/user context

   Kernel context - kernel runs system calls on behalf of some application. Interrupts may occur in kernel context.
   Interrupt context - entered when an interrupt is generated (e.g. by `procfs`). Interrupts are turned off.
   User context - state of normal application when interrupted, say by the end of a time-slice.

2. Why there are so many process scheduling algorithms

   An OS intends to support far more processes than there are processors in hardware. Some time should probably be given to each active process but typically a system runs better if the decision to preempt a running process and dispatching a suspending process in its place is allowed to vary, depending on the type of process and other factors such as competing performance criteria: 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. `procfs`

   A special filesystem that presents information about processes and other system information in a file-like structure, providing a convenient and standardized method for dynamically accessing process data held in the kernel.

6. `spinlock`

   A spinlock does not let a thread through until a condition is true. This condition is tested in a tight loop. Spinlocks can offer higher performance than other locks but are subject to a different set of constraints: 1) the critical section holding the spinlock must be atomic; 2) when the kernel holds a spinlock, preemption is disabled; 3) the locked code must finish as fast as it possibly can.

Question 3: Mutex

What is likely to happen when this executes?
No output - the `watcher` looks for a signal from a `doer` thread too late.

What can be done to fix this?
Lots of possibilities. A simple one is to declare a global integer variable `done`, initialized to 0. Every time a `doer` thread finishes, `done` is incremented by 1. Then, if the value of `done` is at least three, a signal is sent to the `watcher` thread via `pthread_cond_signal(&condvar);`. Meanwhile, after `sleep`, the `watcher` thread tests the value of `done` and waits via `pthread_cond_wait(&condvar, &mutex);` if the value is less than 3.

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

pthread_mutex_t mutex;
pthread_cond_t condvar;

float result = 0.0;

void *watcher(void *id) {
   sleep(1);
   pthread_mutex_lock(&mutex);
   pthread_cond_wait(&condvar, &mutex);
   printf("done watcher: %f\n", result);
   pthread_mutex_unlock(&mutex);
   pthread_exit(NULL);
}

void *doer(void *id) {
   long i;
   for (i=0 ; i < (long)id ; i++) {
      pthread_mutex_lock(&mutex);
      result += (22.0/7.0)*i;
      pthread_mutex_unlock(&mutex);
   }
   pthread_cond_signal(&condvar);
   pthread_exit(NULL);
}

int main(int argc, char *argv[]) {
   long id[] = { 0,1,2,3,4 };
   pthread_t threads[4];

   pthread_mutex_init(&mutex, NULL);
   pthread_cond_init (&condvar, NULL);

   pthread_create(&threads[0], NULL, watcher, (void*)id[0]);
   pthread_create(&threads[1], NULL, doer, (void*)id[1]);
   pthread_create(&threads[2], NULL, doer, (void*)id[2]);
   pthread_create(&threads[3], NULL, doer, (void*)id[4]);

   pthread_exit (NULL);
}
```

Question 4: Process Scheduling

1. State how the Multilevel Feedback Queue works
   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. State three competing measures of CPU scheduling quality and say what they are
   Throughput: the number of processes completed per unit time
   Latency: the time to wait until the process starts
   Fairness: the percentage of time a CPU is given a process

3. What difficulty might a non-preemptive scheduler cause?
   Starvation

4. How is a red-black tree used in CPU scheduling?
   The completely fair scheduler selects the job with the least spent processing time. This is managed by a red-black tree with $O(\log(n))$ insertion time.

Question 5: Memory barriers in device drivers The author of the code below used a memory barrier. What type of barrier and where was it put, do you suppose? Explain. Even if you have a different explanation, you will be marked correct if you have a good argument. It is impractical to give all the struct declarations that are used in this code but you should have a pretty good idea of what these are trying to accomplish from the names.

Put `smp_wmb();` before `wq->head = next;` to make sure the queue entry is written before the head index.

```
/**
 * qib_post_receive - post a receive on a QP
 * @ibqp: the QP to post the receive on
 * @wr: the WR to post
 * @bad_wr: the first bad WR is put here
 *
 * This may be called from interrupt context.
 */
static int qib_post_receive(struct ib_qp *ibqp, struct ib_recv_wr *wr, struct ib_recv_wr **bad_wr) {
   struct qib_qp *qp = to_iqp(ibqp);
   struct qib_rwq *wq = qp->r_rq.wq;
   unsigned long flags;
   int ret;

   /* Check that state is OK to post receive. */
   if (!(ib_qib_state_ops[qp->state] & QIB_POST_RECV_OK) || !wq) {
      *bad_wr = wr;
      ret = -EINVAL;
      goto bail;
   }

   for (; wr; wr = wr->next) {
      struct qib_rwqe *wqe;
      u32 next;
      int i;

      if ((unsigned) wr->num_sge > qp->r_rq.max_sge) {
         *bad_wr = wr;
         ret = -EINVAL;
         goto bail;
      }

      spin_lock_irqsave(&qp->r_rq.lock, flags);
      next = wq->head + 1;
      if (next >= qp->r_rq.size) next = 0;
      if (next == wq->tail) {
         spin_unlock_irqrestore(&qp->r_rq.lock, flags);
         *bad_wr = wr;
         ret = -ENOMEM;
         goto bail;
      }

      wqe = get_rwqe_ptr(&qp->r_rq, wq->head);
      wqe->wr_id = wr->wr_id;
      wqe->num_sge = wr->num_sge;
      for (i = 0; i < wr->num_sge; i++)
         wqe->sg_list[i] = wr->sg_list[i];
      wq->head = next;
      spin_unlock_irqrestore(&qp->r_rq.lock, flags);
   }
   ret = 0;

bail:
   return ret;
}
```

Question 6: Explorer question

What is the point of hyper-threading considering hyper-threads share execution resources such as a cache, buses, the ALU, the branch predictor, instruction fetch mechanism, and so on?

Execution resources are shared but state resources are duplicated. This allows for fast context switching from one hyperthread to another if one hyperthread becomes stalled waiting for a resource.