Lecture

- In the lecture on March 23 we will mainly discuss Chapter 6 (Process Scheduling) and start with Chapter 7 (Deadlocks). Example were be shown for the simulation of the Dining Philosopher problem, a solution with monitors was shown.
- Next Lecture is on April 7th at 10.15.
  There is no lecture on April 7th 14.15.
  Exercises and lectures on 8th and 9th of April are switched, i.e., there is a lecture on Wednesday April 8th at 12.15 and there is a tutorial section on Thursday April 9th.

Exercises

Note, as usual, that you find even more exercises including solutions here :
`http://codex.cs.yale.edu/avi/os-book/OS9/practice-exer-dir/index.html`

Prepare for the Tutorial Session on Wednesday, March 25, 2015:
All exercises not discussed so far. In addition:

5.21 Under what circumstances is rate-monotonic scheduling inferior to earliest-deadline-first scheduling in meeting the deadlines associated with processes?

5.22 Consider two processes, P_1 and P_2 , where $p_1 = 50, t_1 = 25, p_2 = 75$, and $t_2 = 30$.

a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using Gantt chart.

b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

5.23 Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.

6.1 Race conditions are possible in many computer systems. Consider a banking system with two methods: `deposit(amount)` and `withdraw(amount)`. These two methods are passed the amount that is to be deposited or withdrawn from a bank account. Assume that a husband and wife share a bank account and that concurrently the husband calls the `withdraw()` method and the wife calls `deposit()`. Describe how a race condition is possible and what might be done to prevent the race condition from occurring.

6.2 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P_0 and P_1 , share the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i ($i = 0$ or 1) is shown in the Figure below; the other process is P_j, ($j = 1$ or 0). Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
  flag[i] = true;
  while (flag[j]) {
    if (turn == j) {
      flag[i] = false;
      while (turn == j)
        ; /* do nothing */
      flag[i] = true;
    }
  }
  /* critical section */
  turn = j;
  flag[i] = false;
  /* remainder section */
 } while (true);
```

6.3 The first known correct software solution to the critical-section problem for n processes with a lower bound on waiting of $n - 1$ turns was presented by Eisenberg and McGuire. The processes share the following variables:

```
enum pstate {idle, want_in, in_cs};
pstate flag[n];
int turn;
```

All the elements of flag are initially idle; the initial value of turn is immaterial (between 0 and $n - 1$). The structure of process P_i is shown in the Figure below. Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
  while (true) {
    flag[i] = want in;
    j = turn;
    while (j != i) {
      if (flag[j] != idle) {
        j = turn;
        else
          j = (j + 1) % n;
      }
      flag[i] = in cs;
```

```
      j = 0;
      while ( (j < n) && (j == i || flag[j] != in cs))
        j++;
    }
    if ( (j >= n) && (turn == i || flag[turn] == idle))
      break;
    /* critical section */
    j = (turn + 1) % n;
    while (flag[j] == idle)
      j = (j + 1) % n;
    turn = j;
    flag[i] = idle;
    /* remainder section */
  } while (true);
```

6.4 Explain why implementing synchronization primitives by disabling interrupts is not appropriate in a single-processor system if the synchronization primitives are to be used in user-level programs.

6.5 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

6.6 The Linux kernel has a policy that a process cannot hold a spinlock while attempting to acquire a semaphore. Explain why this policy is in place.

6.7 Describe two kernel data structures in which race conditions are possible. Be sure to include a description of how a race condition can occur.

6.8 (modified) Describe how the `compare_and_swap()` (not described in detail in the lecture) instruction can be used i.) to provide mutual exclusion and ii.) to provide mutual exclusion that satisfies the bounded-waiting requirement.

6.9 Consider how to implement a mutex lock using an atomic hardware instruction. Assume that the following structure defining the mutex lock is available:

```
   typedef struct {
   int available;
} lock;
```

where (`available == 0`) indicates the lock is available; a value of 1 indicates the lock is unavailable. Using this struct, illustrate how the following functions may be implemented using the `test_and_set()` and `compare_and_swap()` instructions.

– `void acquire(lock *mutex)`

– `void release(lock *mutex)`

Be sure to include any initialization that may be necessary.

6.11 Assume that a system has multiple processing cores. For each of the following scenarios, describe which is a better locking mechanism-a spinlock or a mutex lock where waiting processes sleep while waiting for the lock to become available:

– The lock is to be held for a short duration.

– The lock is to be held for a long duration.

– The thread may be put to sleep while holding the lock.

6.12 Assume a context switch takes T time. Suggest an upper bound (in terms of T) for holding a spin lock and that if the spin lock is held for any longer duration, a mutex lock (where waiting threads are put to sleep) is a better alternative.

6.14 Consider the code example for allocating and releasing processes shown in the Figure below.

```
#define MAX PROCESSES 255
int number of processes = 0;
/* the implementation of fork() calls this function */
int allocate process() {
  int new pid;
  if (number of processes == MAX PROCESSES)
    return -1;
  else {
    /* allocate necessary process resources */
    ++number of processes;
  }
}
return new pid;
/* the implementation of exit() calls this function */
void release process() {
  /* release process resources */
  --number of processes;
}
```

a. Identify the race condition(s).

b. Assume you have a mutex lock named mutex with the operations acquire() and release(). Indicate where the locking needs to be placed to prevent the race condition(s).

c. Could we replace the integer variable

```
int number_of_processes = 0
```

with the atomic integer

```
atomic_t number_of_processes = 0
```

to prevent the race condition(s)?

6.19 Demonstrate that monitors and semaphores are equivalent insofar as they can be used to implement the same types of synchronization problems.

6.22 Discuss the tradeoff between fairness and throughput of operations in the readers-writers problem. Propose a method for solving the readers-writers problem without causing starvation.

6.23 How does the `signal()` operation associated with monitors differ from the corresponding operation defined for semaphores?

6.28 Suppose we replace the `wait()` and `signal()` operations of monitors with a single construct `await(B)`, where `B` is a general Boolean expression that causes the process executing it to wait until `B` becomes true.

a. Write a monitor using this scheme to implement the readers-writers problem. b. Explain why, in general, this construct cannot be implemented efficiently.

b. Why is it important for the scheduler to distinguish I/O-bound programs from CPU-bound programs?