# VE482 Homework 5

**Due: Nov.1**

**Name: Wu Qinhang**

**ID: 518370910041**

**Email: william_wu@sjtu.edu.cn**

## Ex1

1. **A system has two processes and three identical resources. Each process needs a maximum of two resources. Can a deadlock occur? Explain.**

No. Since the three resources are all identical, one of the process can always get two resources (that meet the running requirement). Assume that the first process has two resources, so that when it finishes, the second resource can get the desired resource immediately, and the deadlock cannot occur. Indeed, the circular wait condition is not satisfied.

2. **A computer has six tape drives, with n processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?**

n can be any integers between 1 and 5.

3. **A real-time system has four periodic events with periods of 50, 100, 200, and 250 msec each. Suppose the four events require 35, 20, 10, and x msec of CPU time, respectively. What is the largest value x for which the system is schedulable?**

$35/50 + 20/100 + 10/200 + x/250 < 1$

$x < 12.5$

Answer: 12.5 (less than 12.5)

4. **Round-robin schedulers normally maintain a list of all runnable processes, with each process occurring exactly once in the list. What would happen if a process occurred more than once in the list? Would there be any reason for allowing this?**

If so, it will be granted more than one turn to run. The reason is that letting one process occurring more than once in the round-robin list can be treated as increasing the priority of that process (so that it can have longer running time each turn).

5. **Can a measure of whether a process is likely to be CPU bound or I/O bound be detected by analyzing the source code. How to determine it at runtime?**

Yes. For example, a CPU-bounded process will be involved with a large amount of calculations while an I/O-bounded process will be involved with more reading or writing operations.

## Ex2 Deadlocks

1. The Request Matrix:

```
743
122
600
011
431
```

2. Yes. With the Available Matrix, either P2 or P4 can run. Assume we let P2 runs first, so after it completes, we can load P4. After that, it is able to execute any other processes. A sample order: P2, P4, P1, P3, P5
3. Yes, just as the sample order listed in 2).

```
Available Matrix:
332
(execute P2)
210
532
(execute P4)
521
743
(execute P1)
000
753
(execute P3)
153
A55
(execute P5)
624
A57
```

## Ex3 Banker's Algorithm

see `./banker.cpp`.

The program should be compiled with `g++ .\banker.cpp -o banker`.

## Ex4 Minix3 Scheduling

In order to discover the scheduling implementation of Minix 3, first I use grep command to search for the keyword `scheduling`:

```
grep -R "scheduling" /usr/src
```

I find two key scheduling-related file: `/usr/src/kernel/main.c`, `/usr/src/servers/sched/main.c`

The major part of the driver of scheduling (kernel mode) in Minix 3 is in the file `/usr/src/kernel/main.c`, function `void kmain(kinfo_t *local_cbi)`.

```
void kmain(kinfo_t *local_cbi){
  /* boot the system and init */
  // ... (omitted)

  /* Set up proc table entries for processes in boot image. */
  for (i=0; i < NR_BOOT_PROCS; ++i) {
  int schedulable_proc;
  proc_nr_t proc_nr;
  int ipc_to_m, kcalls;
  sys_map_t map;

  ip = &image[i];       /* process' attributes */
  // ... (omitted debug and cache)

  reset_proc_accounting(rp);

  /* See if this process is immediately schedulable.
   * In that case, set its privileges now and allow it to run.
   * Only kernel tasks and the root system process get to run
immediately.
   * All the other system processes are inhibited from running by the
   * RTS_NO_PRIV flag. They can only be scheduled once the root system
   * process has set their privileges.
   */
  proc_nr = proc_nr(rp);
  schedulable_proc = (iskerneln(proc_nr) || isrootsysn(proc_nr) ||
    proc_nr == VM_PROC_NR);
  if(schedulable_proc) {
      /* Assign privilege structure. Force a static privilege id. */
            (void) get_priv(rp, static_priv_id(proc_nr));

            /* Priviliges for kernel tasks. */
      if(proc_nr == VM_PROC_NR) {
                priv(rp)->s_flags = VM_F;
                priv(rp)->s_trap_mask = SRV_T;
    ipc_to_m = SRV_M;
    kcalls = SRV_KC;
                priv(rp)->s_sig_mgr = SELF;
                rp->p_priority = SRV_Q;
                rp->p_quantum_size_ms = SRV_QT;
      }
      else if(iskerneln(proc_nr)) {
                /* Privilege flags. */
                priv(rp)->s_flags = (proc_nr == IDLE ? IDL_F : TSK_F);
                /* Allowed traps. */
                priv(rp)->s_trap_mask = (proc_nr == CLOCK
```

```
                    || proc_nr == SYSTEM  ? CSK_T : TSK_T);
                ipc_to_m = TSK_M;                   /* allowed targets */
                kcalls = TSK_KC;                    /* allowed kernel
calls */
            }
            /* Priviliges for the root system process. */
            else {
        assert(isrootsysn(proc_nr));
                priv(rp)->s_flags= RSYS_F;         /* privilege flags */
                priv(rp)->s_trap_mask= SRV_T;      /* allowed traps */
                ipc_to_m = SRV_M;                  /* allowed targets */
                kcalls = SRV_KC;                   /* allowed kernel
calls */
                priv(rp)->s_sig_mgr = SRV_SM;      /* signal manager */
                rp->p_priority = SRV_Q;            /* priority queue */
                rp->p_quantum_size_ms = SRV_QT;    /* quantum size */
            }

            /* Fill in target mask. */
            // ... (omitted)

            /* Fill in kernel call mask. */
            // ... (omitted)
  }
  else {
      /* Don't let the process run for now. */
            RTS_SET(rp, RTS_NO_PRIV | RTS_NO_QUANTUM);
  }

  /* Arch-specific state initialization. */
  arch_boot_proc(ip, rp);

  /* scheduling functions depend on proc_ptr pointing somewhere. */
  if(!get_cpulocal_var(proc_ptr))
    get_cpulocal_var(proc_ptr) = rp;

  /* Process isn't scheduled until VM has set up a pagetable for it. */
  if(rp->p_nr != VM_PROC_NR && rp->p_nr >= 0) {
    rp->p_rts_flags |= RTS_VMINHIBIT;
    rp->p_rts_flags |= RTS_BOOTINHIBIT;
  }

  rp->p_rts_flags |= RTS_PROC_STOP;
  rp->p_rts_flags &= ~RTS_SLOT_FREE;
  DEBUGEXTRA(("done\n"));
  }
}
```

From the code, we can see that in Minix 3, the kernel will first set up the process table. Next, it will check the processes one by one to see whether it is immediately schedulable. Then there are three cases:

1. **it is immediately schedulable**. Then the kernel will set its privileges and allow it to run.
2. **no, but it is a kernel task / root system process**. Then it will get to run immediately.
3. **other cases**. Then it is inhibited from running.

Note that the processes can only be scheduled once their privileges are set by the system.

Another file in `/usr/src/servers/sched/main.c` contains the program of the SCHED scheduler. It will by default sit idle until it is activated by the process manager to take over scheduling a particular process. [1] [2]

```
int main(void){
  /* Initialize scheduling timers, used for running balance_queues */
  // ... (omitted)
  /* This is SCHED's main loop - get work and do it, forever and
forever. */
  while (TRUE) {
    int ipc_status;

    /* Wait for the next message and extract useful information from it.
*/
    // ... (omitted)

    /* Check for system notifications first. Special cases. */
    // ... (omitted)

    switch(call_nr) {
    case SCHEDULING_INHERIT:
    case SCHEDULING_START:
      result = do_start_scheduling(&m_in);
      break;
    case SCHEDULING_STOP:
      result = do_stop_scheduling(&m_in);
      break;
    case SCHEDULING_SET_NICE:
      result = do_nice(&m_in);
      break;
    case SCHEDULING_NO_QUANTUM:
      /* This message was sent from the kernel, don't reply */
      if (IPC_STATUS_FLAGS_TEST(ipc_status,
        IPC_FLG_MSG_FROM_KERNEL)) {
        if ((rv = do_noquantum(&m_in)) != (OK)) {
          printf("SCHED: Warning, do_noquantum "
            "failed with %d\n", rv);
        }
        continue; /* Don't reply */
      }
      else {
        printf("SCHED: process %d faked "
          "SCHEDULING_NO_QUANTUM message!\n",
            who_e);
```

```
39            result = EPERM;
40          }
41          break;
42        default:
43          result = no_sys(who_e, call_nr);
44        }
45
46  sendreply:
47        /* Send reply. */
48        if (result != SUSPEND) {
49          m_in.m_type = result;     /* build reply message */
50          reply(who_e, &m_in);    /* send it away */
51        }
52      }
53      return(OK);
54  }
```

# Ex5 The reader-writer problem

1. **Explain how to get a read lock, and write the corresponding pseudocode.**

In order to get a read lock, we may first lock the reader count. Then, we check whether it is the first reader. If so, we lock db. Then, we increase the reader count and unlock the access to it. After that, we release the reader count.

```
1  /* pseudocode for read lock */
2  int rc = 0; // reader count
3  semaphore db_lock = 1; // control on access to db
4  semaphore rc_lock = 1; // control on access to rc
5
6  void read_lock(){
7    down(&rc_lock);
8    rc++;
9    if(rc==1) down(&b_lock);
10   up(&rc_lock);
11 }
```

2. **Describe what is happening if many readers request a lock.**

If many readers are requesting a lock, they may wait for rc to be released. Once they gain the control on writing to rc, they will first call `read_lock`, then accessing the data in the db, and then unlock it. Since when there are readers accessing the db, writer will keep waiting until no readers are accessing the db.

3. **Explain how to implement this idea using another semaphore called read_lock.**

```
1  void write_lock(){
2    down(&reader_lock);
3    down(&db_lock);
4  }semaphore reader_lock = 1; // control on reader
5  // ... (omitted)
6
```

```
void read_lock(){
  down(&reader_lock);
  down(&rc_lock);
  rc++;
  if(rc==1) down(&b_lock);
  up(&rc_lock);
  up(&reader_lock);
}
void write_lock(){
  up(&reader_lock);
  up(&db_lock);
}
void read_unlock(){
  down(&reader_lock);
  down(&rc_lock);
  rc--;
  if(rc==0) down(&b_lock);
  up(&rc_lock);
  up(&reader_lock);
}
```

As is shown in the pseudocode above, we allow the writer to lock the reader_lock, so that other readers will be blocked when the writer wants to access db until it finishes the job. In this way, the writer won't wait forever until it gets to work.

4. **Is this solution giving any unfair priority to the writer or the reader? Can the problem be considered as solved?**

Yes, the writer is granted much priority over the reader. The problem cannot be considered as solved.

---

1. "developersguide:userspacescheduling [Wiki]," *Minix3.org* , 2017. https://wiki.minix3.org/doku.php?id=developersguide:userspacescheduling#:~:text=Userspace%20scheduling,-How%20does%20it&text=Scheduling%20in%20Minix%20is%20simple,and%20has%20a%20quantum%20assigned.&text=A%20process%20that%20runs%20out,process%20priority%20and%20time%20quantum. (accessed Nov. 01, 2020). ↩

2. Hiroaki Machida, "What does sys_schedule() do in Minix 3.1.8?," *Stack Overflow* , Nov. 27, 2019. https://stackoverflow.com/questions/59065429/what-does-sys-schedule-do-in-minix-3-1-8 (accessed Nov. 01, 2020). ↩