VE482 — Introduction to Operating Systems Homework 5

Shengyuan Xu 518370910200

VE482 — Introduction to Operating Systems Homework 5

- Ex. 1 — Simple questions
- Ex. 2 — Deadlocks
- Ex. 3 — Programming
- Ex. 4— Minix3
- Ex. 5 — The reader-writer problem

Ex. 1 — Simple questions

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

   The system is a safe one. Suppose each process has one resource. There's a resource free. Either process can ask for the available resource, finish its work and release both resources. Thereby, deadlock is impossible.

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?**

   $1 \leq n \leq 5$

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?**

   The fraction of CPU use is: $35/50 + 20/100 + 10/200 + x/250$, which should be less than 1. Therefore, we can get $x < 12.5\ msec$

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?**

   The process will have more opportunities to be executed. Allowing such operation is equivelant to give the process a higher priority.

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. 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.

Ex. 2 — Deadlocks

1. **Determine the content of the Request matrix.**

```
R =
     7     4     3
     1     2     2
     6     0     0
     0     1     1
     4     3     1
```

2. **Is the system in a safe state?**

   Yes, P_4 can run first, then $A = [5, 4, 3]$. After that we can run P_2, and $A = [7, 4, 3]$. At this moment, any other process can be run successfully.

3. **Can all the processes be completed without the system being in an unsafe state at any stage?**

   Yes, for example $P_4,\ P_2,\ P_1,\ P_3,\ P_5$.

Ex. 3 — Programming

```
#include <iostream>
#include <vector>

using namespace std;

#define PS_CNT 5 // process count
#define RS_CNT 4 // resource count

// current allocation matrix
vector <vector<int>> C = {
        {3, 0, 1, 1},
        {0, 1, 0, 0},
        {1, 0, 1, 0},
        {1, 1, 0, 1},
        {0, 0, 0, 0}
};

// request matrix
vector <vector<int>> R = {
```

```
        {2, 1, 0, 0},
        {0, 1, 1, 1},
        {3, 1, 0, 0},
        {0, 0, 1, 0},
        {3, 1, 1, 0}
};

// available matrix
vector<int> A = {1, 1, 2, 0};

// existing matrix
vector<int> E = {6, 3, 4, 2};

#define S_RUNNING 1
#define S_SUCCESS 2
#define S_FAIL 0

// check if request R_i can be satisfied
bool isRequestSatisfied(vector<int> R, vector<int> A) {
    for (int i = 0; i < RS_CNT; ++i) {
        if (R[i] > A[i]) {
            return false;
        }
    }
    return true;
}

// update available source
void freeSrc(vector<int> C, vector<int> &A) {
    for (int i = 0; i < RS_CNT; ++i) {
        A[i] += C[i];
    }
}

bool isTerminated(int PID, vector<int> terminated) {
    for (int p: terminated) {
        if (p == PID) {
            return true;
        }
    }
    return false;
}


int main() {
    int currPSCnt = PS_CNT;
    int currState = S_RUNNING;
    vector<int> terminated;
```

```
    while (currState == S_RUNNING) {
        // find a process whose request can be satisfied,
        int PID;
        for (PID = 0; PID < PS_CNT; ++PID) {
            // if the process has already been satisfied
            if (isTerminated(PID, terminated)) {
                continue;
            }

            // if an unsatisfied process can be satisfied
            if (isRequestSatisfied(R[PID], A)) {
                break;
            }
        }

        // mark it as terminated, and free all of its resources
        if (PID < PS_CNT) {
            terminated.push_back(PID);
            std::cout << "executing process #" << PID << "...\n";
            freeSrc(C[PID], A);
            currPSCnt--;
        } else { // not find
            currState = S_FAIL;
            std::cout << "fail execute all the processes\n";

        }

        // check all requests are satisfied
        if (currPSCnt == 0) {
            currState = S_SUCCESS;
            std::cout << "successfully execute all the processes\n";
        }
    }

    return 0;
}
```

The result is:

```
executing process #3...
executing process #0...
executing process #1...
executing process #2...
executing process #4...
successfully execute all the processes
```

Ex. 4— Minix3

Scheduling in Minix is simple multi-priority round robin. Each runnable process is in a priority queue and has a quantum assigned. The assigned quantum of each process is periodically decreased. Each process is associated with a scheduler, e.g. a process that is responsible for making scheduling decisions for such a process.

The *kernel* retains a very simple round robin scheduler which always picks the process at the head of the highest priority queue. This does not guarantee that processes will not starve. It also contains a default policy. If a process is not associated with any userspace scheduler and runs out of its quantum, the quantum is renewed. If a process is associated with a scheduler, a message generated by the kernel on behalf of the process is sent to the scheduler every time the process runs out of its quantum. A process that runs out of its quantum is blocked (not runnable) until the scheduler decides.

The scheduler can change the process priority and time quantum. *sys_schedule()* is used to change the priority and quantum and thus can make the process runnable again.

The userspace scheduler can use *sef_receive_status()* to check whether the message was generated by the kernel or whether the massage is a fake sent by a process. This information is reliable.

In the source code of Minix kernel `/usr/src/kernel/main.c`, I find 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"));
  }
}
```

Also in `/usr/src/servers/sched/main.c`, I find:

```
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);
        result = EPERM;
      }
      break;
    default:
      result = no_sys(who_e, call_nr);
    }

sendreply:
    /* Send reply. */
    if (result != SUSPEND) {
      m_in.m_type = result;     /* build reply message */
      reply(who_e, &m_in);    /* send it away */
    }
  }
  return(OK);
}
```

Ex. 5 — The reader-writer problem

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

   The first one get `read_lock` can also get the `db_lock` and increment the `count`, while following ones can only increment the `count`.

   ```
   int count = 0;
   semaphore cnt_lock = 1, db_lock = 1;

   void read_lock() {
       down(&cnt_lock);
       if(count == 0) {
           down(&db_lock);
       }
       ++count;
       up(&cnt_lock);
   }
   ```

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

   The reader has a higher priority on getting the `db_lock`. If there are a lot of readers, the writer will go into starvation.

To overcome the previous problem we will block any new reader when a writer becomes available.

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

```
void write_lock() {
  down(&read_lock);
    down(&db_lock);
}

void write_unlock() {
    up(&db_lock);
    up(&read_lock);
}

void reader_lock() {
    down(&read_lock);
    up(&read_lock);
    down(&cnt_lock);
    if(count == 0) {
        down(&db_lock);
    }
    ++count;
    up(&cnt_lock);
}

void reader_unlock() {
    down(&cnt_lock);
    if(--count == 0) {
        up(&db_lock);
    }
    up(&cnt_lock);
}
```

As shown in the pseudocode above, we allow the writer to lock the reader_lock, so that when the writer wants to access the DB, other readers will be blocked until the writer completes its task. That way, the author doesn't wait for it to start working.

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.