VE482
Introduction to Operating Systems

HOMEWORK 5

November 14, 2021

Yihua Liu 518021910998

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.
   No, a deadlock cannot occur. Since the resources are identical, if each process is allocated with 1 resource, then one process can be allocated with another 1 resource; if process 1 is allocated with 1 resource and process 2 is allocated with 2 resources, after process 2 frees its memory, process 1 can be allocated with 1 resource needed.

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?
   For $0 \leq n \leq 5, n \in \mathbb{N}$ the system is deadlock free. As long as each process is allocated with one tape drive and one of the process can be allocated with 2 tape drives, the system is deadlock free.

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?
   Utilization of a CPU: $U = \sum_{i=1}^{n} \frac{C_i}{P_i}$, where $n = 4$ is the number of tasks.
   Schedulable utiilization bound: $U_b \leq 1$ (refer to ECE4730J Lecture 8).

   $$\frac{35}{50} + \frac{20}{100} + \frac{10}{200} + \frac{x}{250} \leq 1$$

   Solving the equation,

   $$x \leq 12.5$$

   Therefore, the largest value x for which the system is schedulable is 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 a process occurred more than once in the list, it will be executed more than once in one period (it may be executed in adjacent time slices). The

reason for allowing this is that some processes have higher priorities (or strategies decided by the scheduler of the operating system) so that they have to occupy more time slices to finish first.

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?
   If a process is mainly doing a mass of calculations, it is likely to be CPU bound; if it is mainly doing a lot of file inputs and outputs (files can be peripherals, devices, etc.), it is likely to be I/O bound.
   Techniques to determine at runtime:

   - Using `top` or `htop` (an interactive process viewer) to display Linux processes and see their CPU usage and memory usage.
   - Using `iotop` to see the disk read and disk write of commands for Linux kernel version 2.6.20 or later.
   - Using `ps` to see the state of processes (D for disk, R for run, S for sleep).
   - For some Linux systems, using `iostat` to see the CPU statistics and input/output statistics for devices, partitions, and NFS (network filesystems).

Ex. 2 — Deadlocks

1. Determine the content of the Request matrix.
   The request matrix is equal to the maximum matrix minus the allocated matrix:
   $$\begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

2. Is the system in a safe state?
   Process the Banker's algorithm, start and finish P_2, available resource:
   $$\begin{pmatrix} 3 & 3 & 2 \end{pmatrix} - \begin{pmatrix} 1 & 2 & 2 \end{pmatrix} + \begin{pmatrix} 3 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 5 & 3 & 2 \end{pmatrix}$$
   Now availabe resource is enough for both P_4 and P_5. We assume we always start process with less process number first. Start and finish P_4, available resource:
   $$\begin{pmatrix} 5 & 3 & 2 \end{pmatrix} - \begin{pmatrix} 0 & 1 & 1 \end{pmatrix} + \begin{pmatrix} 2 & 2 & 2 \end{pmatrix} = \begin{pmatrix} 7 & 4 & 3 \end{pmatrix}$$
   Start and finish P_1, available resource:
   $$\begin{pmatrix} 7 & 4 & 3 \end{pmatrix} - \begin{pmatrix} 7 & 4 & 3 \end{pmatrix} + \begin{pmatrix} 7 & 5 & 3 \end{pmatrix} = \begin{pmatrix} 7 & 5 & 3 \end{pmatrix}$$
   Start and finish P_3, available resource:
   $$\begin{pmatrix} 7 & 5 & 3 \end{pmatrix} - \begin{pmatrix} 6 & 0 & 0 \end{pmatrix} + \begin{pmatrix} 9 & 0 & 2 \end{pmatrix} = \begin{pmatrix} 10 & 5 & 5 \end{pmatrix}$$

Start and finish P_5, available resource:

$$\begin{pmatrix}10 & 5 & 5\end{pmatrix} - \begin{pmatrix}4 & 3 & 1\end{pmatrix} + \begin{pmatrix}4 & 3 & 3\end{pmatrix} = \begin{pmatrix}10 & 5 & 7\end{pmatrix}$$

Therefore, the system is in a safe state.

3. Can all the processes be completed without the system being in an unsafe state at any stage?
   Yes, all the processes can be completed without the system being in an unsafe state at any stage. The reason is already shown above.

Ex. 3 — Programming

Implement the Banker's algorithm.

See h5ex3.c and README.md.

Ex. 4 — Minix 3

How is scheduling handled in Minix 3? Provide clear explanations on how to find the information just by exploring the source code of Minix kernel.

Find files of source code that are related to scheduling:

```
grep -R scheduling /usr/src/kernel
find /usr/src -name sched
```

Related source code is:

- `/usr/src/kernel/main.c`
- `/usr/src/server/sched/main.c`

```
1  /* See if this process is immediately schedulable.
2   * In that case, set its privileges now and allow it to run.
3   * Only kernel tasks and the root system process get to run
    ↪  immediately.
4   * All the other system processes are inhibited from running by
    ↪  the
5   * RTS_NO_PRIV flag. They can only be scheduled once the root
    ↪  system
6   * process has set their privileges.
7   */
8   proc_nr = proc_nr(rp);
9  schedulable_proc = (iskerneln(proc_nr) || isrootsysn(proc_nr)
    ↪  ||
10         proc_nr == VM_PROC_NR);
11 if(schedulable_proc) {
12     /* Assign privilege structure. Force a static privilege id.
       ↪  */
13         (void) get_priv(rp, static_priv_id(proc_nr));
```

```
14          /* Priviliges for kernel tasks. */
15      if(proc_nr == VM_PROC_NR) {
16              priv(rp)->s_flags = VM_F;
17              priv(rp)->s_trap_mask = SRV_T;
18          ipc_to_m = SRV_M;
19          kcalls = SRV_KC;
20              priv(rp)->s_sig_mgr = SELF;
21              rp->p_priority = SRV_Q;
22              rp->p_quantum_size_ms = SRV_QT;
23      }
24      else if(iskerneln(proc_nr)) {
25              /* Privilege flags. */
26              priv(rp)->s_flags = (proc_nr == IDLE ? IDL_F :
                ↪  TSK_F);
27              /* Allowed traps. */
28              priv(rp)->s_trap_mask = (proc_nr == CLOCK
29                  || proc_nr == SYSTEM  ? CSK_T : TSK_T);
30              ipc_to_m = TSK_M;                   /* allowed
                ↪  targets */
31              kcalls = TSK_KC;                    /* allowed
                ↪  kernel calls */
32          }
33          /* Priviliges for the root system process. */
34          else {
35              assert(isrootsysn(proc_nr));
36              priv(rp)->s_flags= RSYS_F;         /* privilege
                ↪  flags */
37              priv(rp)->s_trap_mask= SRV_T;      /* allowed traps
                ↪  */
38              ipc_to_m = SRV_M;                  /* allowed
                ↪  targets */
39              kcalls = SRV_KC;                   /* allowed kernel
                ↪  calls */
40              priv(rp)->s_sig_mgr = SRV_SM;      /* signal manager
                ↪  */
41              rp->p_priority = SRV_Q;                  /*
                ↪  priority queue */
42              rp->p_quantum_size_ms = SRV_QT;    /* quantum size
                ↪  */
43          }
44          /* Fill in target mask. */
45          memset(&map, 0, sizeof(map));
46          if (ipc_to_m == ALL_M) {
47              for(j = 0; j < NR_SYS_PROCS; j++)
48                  set_sys_bit(map, j);
49          }
50          fill_sendto_mask(rp, &map);
51
52          /* Fill in kernel call mask. */
```

```
        for(j = 0; j < SYS_CALL_MASK_SIZE; j++) {
            priv(rp)->s_k_call_mask[j] = (kcalls == NO_C ? 0 :
            ↪  (~0));
        }
}
else {
    /* Don't let the process run for now. */
        RTS_SET(rp, RTS_NO_PRIV | RTS_NO_QUANTUM);
}
```

The Minix kernel firstly check whether the process is immediately schedulable. If the process is immediately schedulable, the kernel will set its privileges and lets it run. Besides, only kernel task or root system process are able to run immediately, otherwise the kernel does not let it run for now.

```
/* Initialize scheduling timers, used for running
↪  balance_queues */
init_scheduling();

/* 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. */
    if (sef_receive_status(ANY, &m_in, &ipc_status) != OK)
        panic("SCHED sef_receive error");
    who_e = m_in.m_source;        /* who sent the message */
    call_nr = m_in.m_type;        /* system call number */

    /* Check for system notifications first. Special cases. */
    if (is_ipc_notify(ipc_status)) {
        switch(who_e) {
            case CLOCK:
                expire_timers(m_in.NOTIFY_TIMESTAMP);
                continue;        /* don't reply */
            default :
                result = ENOSYS;
        }

        goto sendreply;
    }

    switch(call_nr) {
    case SCHEDULING_INHERIT:
    case SCHEDULING_START:
        result = do_start_scheduling(&m_in);
        break;
```

```
32      case SCHEDULING_STOP:
33          result = do_stop_scheduling(&m_in);
34          break;
35      case SCHEDULING_SET_NICE:
36          result = do_nice(&m_in);
37          break;
38      case SCHEDULING_NO_QUANTUM:
39          /* This message was sent from the kernel, don't reply
            ↪  */
40          if (IPC_STATUS_FLAGS_TEST(ipc_status,
41              IPC_FLG_MSG_FROM_KERNEL)) {
42              if ((rv = do_noquantum(&m_in)) != (OK)) {
43                  printf("SCHED: Warning, do_noquantum "
44                      "failed with %d\n", rv);
45              }
46              continue; /* Don't reply */
47          }
48          else {
49              printf("SCHED: process %d faked "
50                  "SCHEDULING_NO_QUANTUM message!\n",
51                      who_e);
52              result = EPERM;
53          }
54          break;
55      default:
56          result = no_sys(who_e, call_nr);
57      }
58
59  sendreply:
60      /* Send reply. */
61      if (result != SUSPEND) {
62              m_in.m_type = result;                   /* build
                ↪  reply message */
63              reply(who_e, &m_in);                  /* send it away
                ↪  */
64      }
65  }
```

After initializing timers used for running `balance_queues`, the scheduler starts the SCHED's main loop. During the loop, it will wait for the next message and check for system notifications first. Then, it will do different operations depending on the value of `call_nr`, the system call number.

Ex. 5 — The reader-writer problem

1. Explain how to get a read lock, and write the corresponding pseudocode. To get a read lock, we have to first lock the count, increase the count. If now the count is 1, lock the database. Finally, unlock the count.

```c
int count = 0;
sem_t count_lock;
sem_t db_lock;
sem_init(&count_lock, 0, 1);
sem_init(&db_lock, 0, 1);

void read_lock() {
    sem_wait(&count_lock);  // down(count_lock);
    count++;
    if (count == 1)
        sem_wait(&db_lock);  // down(db_lock);
    sem_post(&count_lock);  // up(count_lock);
}
```

2. Describe what is happening if many readers request a lock.
   If many readers request a lock, the readers may starve the writer. It is unfair, because once a reader acquires the `count_lock`, other readers can also acquire this lock, but if a writer wants to acquire `db_lock`, it must wait for all the readers to finish. The last exiting reader `ups` `db_lock` so that the waiting writer can acquire this lock.

3. Explain how to implement this idea using another semaphore called `read_lock`.
   Before acquiring `db_lock`, the program should block other readers. After acquiring `db_lock`, recover `read_lock`. Besides, before readers acquiring the `count_lock`, it has to wait for the writer to release `read_lock`.

```c
int count = 0;
sem_t count_lock;
sem_t db_lock;
sem_t read_lock;
sem_init(&count_lock, 0, 1);
sem_init(&db_lock, 0, 1);
sem_init(&read_lock, 0, 1);

void read_lock() {
    sem_wait(&read_lock);  // down(read_lock);
    sem_wait(&count_lock);  // down(count_lock);
    count++;
    if (count == 1)
        sem_wait(&db_lock);  // down(db_lock);
    sem_post(&count_lock);  // up(count_lock);
    // read_lock can also be posted immediately after
    ↪  waiting count_lock or before posting count_lock.
    sem_post(&read_lock);  // up(read_lock);
}

void read_unlock() {
    sem_wait(&count_lock);  // down(count_lock);
```

```
22      count--;
23      if (count == 0)
24          sem_post(&db_lock);  // up(db_lock);
25      sem_post(&count_lock);  // up(count_lock);
26  }
27
28  void write_lock() {
29      sem_wait(&read_lock);  // down(read_lock);
30      sem_wait(&db_lock);  // down(db_lock);
31  }
32
33  void write_unlock() {
34      // read_lock can also be posted at the end of
        ↪  write_lock() after waiting for db_lock.
35      sem_post(&read_lock);  // up(read_lock);
36      sem_post(&db_lock);  // up(db_lock);
37  }
```

4. Is this solution giving any unfair priority to the writer or the reader? Can the problem be considered as solved?
   No, this solution is not giving any unfair priority to the writer or the reader.
   No, the problem cannot be considered as solved because this solution can ensure no thread starving if and only if the semaphores preserve first-in first-out ordering when blocking and releasing threads in case there are multiple writers.