

VE482 Homework 5

Weili Shi | 519370910011 | Nov 11, 2021



Ex 1.

1. A system has two processes and three identical resources.

Each process needs a maximum of two resources. Can a deadlock occur? Explain.

No. Assume 2 process both need 2 resources. According to the pigeonhole principle, there must be one process which is able to obtain 2 resources. Then, the process can continue executing, until it finishes and releases the resources. Then, the other process will be allowed to run. No deadlock can ever occur.

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?

0, 1, 2, 3, 4, 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?

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

largest value for x is 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 be given more time to run. It is possible that the process is important and it is given more 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. CPU bound processes will do lots of calculations, while I/O bound will do lots of reading and writing. To determine at runtime, we can calculate the percentage of CPU time from the total execution time.

Ex 2. Deadlocks

1. Determine the request matrix.

(743

122

600

11

431)

2. Is the system in a safe state?

- allocate P2 122 resources, P4 11 resources ----- 199 available
- P2, P4 finish executing ----- 743 available

- o allocate P5 431 resources ----- 312 available
- o P5 finishes executing ----- 745 available
- o allocate P1 743 resources ----- 12 available
- o P1 finishes executing ----- 755 available
- o allocate P3 600 resources ----- 155 available
- o P3 finishes executing ----- 1057 available
- o Done! It is in a safe state.

3. Yes. A possible sequence of execution is shown above.

Ex 3.

see ./ex3/README.md

Ex 4. Minix 3 Scheduling

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

After running a command, a list of files with the keyword are listed. Here is part from /usr/src/kernel/main.c

```
/* 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));

    /* Privileges 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 */
    }
    /* Privileges 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. */
memset(&map, 0, sizeof(map));

if (ipc_to_m == ALL_M) {
    for(j = 0; j < NR_SYS_PROCS; j++)
        set_sys_bit(map, j);
}

fill_sendto_mask(rp, &map);

/* 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);
}

/* 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"));

```

We first check if a process is schedulable. This is done by `schedulable_proc = (iskerneln(proc_nr) || isrootsysn(proc_nr) || proc_nr == VM_PROC_NR);`

The privileged processes are either kernel processes or root processes, their privileges are then set and are allowed to run. Other processes are not allowed to run for now. After the privileged processes are run,

Ex 5. The reader-writer problem

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

```
int count = 0;
semaphore count_lock = 1;
semaphore db_lock = 1;

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

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

Writers will only be able to access the resource when no reader is inside, that is when count == 0. However, if reader keeps coming, count can never be 0. When it is the writer's turn, it gets blocked, however the readers after her gets to come in and read. The writer is likely to be blocked forever.

3. To overcome the previous problem we will block any new reader when a writer becomes available. Explain how to implement this idea using another semaphore called read_lock.

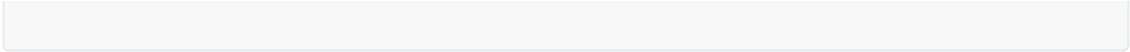
```
int count = 0;
semaphore count_lock = 1;
semaphore db_lock = 1;
semaphore read_lock = 1;

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

void read_unlock() {
    down(&count_lock);
    count--;
    if (count == 0) up(&db_lock);
    up(&count_lock);
}

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

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

- 
4. As soon as a writer is ready, no reader can access the resource even if she is in front of the writer in the queue. Writer is given much more priority and the problem is not solved.