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. To meet the running requirement, one process needs to have two resources until it's finished. When it finished, the second resource will have its second resource. Therefore, the deadlock will not 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?**

   n = [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?**

   We have $35/50 + 20/100 + 10/200 + x/250 < 1$ and we can get $x < 12.5$.

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?**

   This process will be runned more than one turn because allowing one process running more than once is increasing it priority and it will be granted longer running time.

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 most straight forward way to identify is to recognize that a CPU-bounded process will be composed of a large amount of calculations while an I/O-bounded process will consist a lot of Input/Ouput related operations like reading or writing.

Ex. 2 - Deadlocks

1. $[743 \quad 122 \quad 600 \quad 011 \quad 431]$.
2. Yes. According to the Available Matrix, we can run either P2 or P4. Let's assume we run P2 first. Then we can load P4 after we finish running P2. Next we run other process accordingly.
3. Yes. One possible order is: $[P_2, P_4, P_1, P_3, P_5]$.

   Calls in detail are as follows:

   332 -> (P2) -> 210 -> 532 -> (P4) -> 521 -> 743 -> (P1) -> 000 -> 753 -> (P3) -> 153 -> A55 -> (P5) -> 624 -> A57

Ex. 3 - Banker's Algorithm

Check `./banker.cpp` .

Compile:

```
gcc banker.c -o banker
```

Ex. 4 - Minix3 Scheduling

By searching the internet, I find two vital files related to scheduling of Minix 3:

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

In the first file `/usr/src/kernel/main.c` , we check function `void kmain(kinfo_t *local_cbi)` for more information.

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

```c
/* 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"));
  }
}
```

According to the code, we can see the kernel in Minix 3 will first set up proc table entries for processes in boot image. Then it will see if each process is immediately schedulable. 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.

Then we check another file in `/usr/src/servers/sched/main.c`, the `main` function. It will by default loop and work forever until a certain process require scheduling.

```
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.**

   Two semaphores, one to control the reader count and one to control the database, are required. We have the pseudocode as follows:

```
int count = 0; // reader count
semaphore count_lock = 1; // control the reader count
semaphore db_lock = 1; // control the database

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

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

   If many readers request a lock, they may wait for count to be released. When a single reader has the control of `count`, it will first call `read_lock`, and read the data in the db. Then it will unlock it using a `read_unlock` function. However, a writer may wait for a very long time before all readers finish their job.

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

   We may let a writer to take control of the reader_lock s.t. it can prevent other readers from reading the database until the writer finished its job. In this case, the writer does not need to wait for a long time.

```
int count = 0; // reader count
semaphore count_lock = 1; // control the reader count
semaphore db_lock = 1; // control the database
```

```
void write_lock(){
  down(reader_lock);
  down(db_lock);
}

void write_unlock(){
  up(reader_lock);
  up(db_lock);
}

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

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

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

   Obviously, the writer is given more priority over the reader. Thus, the problem cannot be considered as solved.