Problem 1. Simple questions

1. No. Each process could has one resource at first, and for the free resource, any of the process could ask for it. This can be later released and to be used for the other process.

2. $n \leq 5$. When the number of processes exceeds 5, no single process can have 2 maximum resources without leaving a deadlock, since all of them beforehand has one resource.

3. To schedule the real time system, it should satisfy

$$1 - \frac{35}{50} - \frac{20}{100} - \frac{10}{200} - \frac{x}{250} > 0$$

$$x > 12.5$$

4. The same process occurring twice in the list can be activated twice. Allowing such occurrence may be related to designing a scheduling algorithm based on priority, where the number of occurrence denotes priority level.

5. View the source code, the computation, algorithm are more likely to be CPU bound. The data to be sent/received when input and output is IO-bound. During the running time, command line tool sar, top or iostat can be used to view the CPU usage and IO usage.

Problem 2. Deadlocks

1. The requested resouces is just Maximum - Allocated

Process	Request
P_1	743
P_2	122
P_3	600
P_4	011
P_5	431

Table 1: Requested matrix

2. Yes. First the resources can be given to P_2 or P_4, then the resource gain the allocated part and become 743, so it can give P_1 resources and get back allocated resources. Now it become 753, finally it can give P_3 and P_5 the requested resources, and end up with 1057.

3. As stated above, the process can be completed without being in an unsafe state.

Problem 3. Programming

See README.md for more information. The source code is attached below (python)

```python
class process:
    def __init__(self):
        self.max = []
        self.allocation = []
        self.need = []
        self.completed = False

    def printstat(self):
        print("Max resources: ", str(self.max))
        print("Allocated resources: ", str(self.allocation))
        print("Need resources: ", str(self.need))

if __name__ == '__main__':
    available = []  # Record available resources
    processes = []  # Store all the processes
    process_num = int(input())
    resource_num = int(input())
    for i in range(process_num):
        tmp = process()
        maxstr = input().split()
        allstr = input().split()
        neestr = input().split()
        for j in range(resource_num):
            tmp.max.append(int(maxstr[j]))
            tmp.allocation.append(int(allstr[j]))
            tmp.need.append(int(neestr[j]))
        processes.append(tmp)
    avastr = input().split()
    for i in range(resource_num):
        available.append(int(avastr[i]))

    print("Initialization Completed!")

    ### Testing
    print("process_num: {}".format(process_num))
    print("resource_num: {}".format(resource_num))
    print("available resource: {}".format(str(available)))
    cnt = 0
    for i in processes:
        print("------ Process {} Status -------".format(cnt))
        cnt += 1
        i.printstat()

    ### Banker's algorithm
    print("------ Banker's Algorithm ------")
    complete_process = 0
    while (complete_process < process_num):
        flag = False
        for p in processes:
            if all(j >= 0 for j in [available[i] - p.need[i] for i in
             ↪  range(resource_num)]):
                flag = True
                p.completed = True
                complete_process += 1
                available += p.allocation
        if not flag:
            print('Status: Unsafe')
            exit()
    print("Status: safe")
```

Problem 4. Minix 3

The related code about scheduling in MINIX3 is included in `/minix/kernel/main.c`
The source code about deciding whether a process get to be scheduled is in below.

```
/* 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);
            /* Init flags. */
            priv(rp)->s_init_flags = TSK_I;
            /* 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_init_flags = SRV_I;   /* init 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);

```

```
72  /* scheduling functions depend on proc_ptr pointing somewhere. */
73  if(!get_cpulocal_var(proc_ptr))
74          get_cpulocal_var(proc_ptr) = rp;
75
76  /* Process isn't scheduled until VM has set up a pagetable for it. */
77  if(rp->p_nr != VM_PROC_NR && rp->p_nr >= 0) {
78          rp->p_rts_flags |= RTS_VMINHIBIT;
79          rp->p_rts_flags |= RTS_BOOTINHIBIT;
80  }
81
82  rp->p_rts_flags |= RTS_PROC_STOP;
83  rp->p_rts_flags &= ~RTS_SLOT_FREE;
84  DEBUGEXTRA(("done\n"));
85  }
```

As is written in comments, the schedulable process's result is recorded in `schedulable_proc`, once its true, which means this process can run right away, minix calls `get_priv` function to set the privilege and let it to run. From line 25, kernel tasks and root system process will be set to run immediately, in line 66, all other process are set to be waiting until all the previous process running is over. Finally, after the pagetable is set up, the process can be scheduled and set flag to be stop and free, otherwise the flag is set to be vminhibit and bootinhibit.

Problem 5. The reader-writer problem

1. To get a read lock we want to protect the counter, and focus on its value to lock or unlock the db. The pseudocode is given as follows

```
// count records current reader number
void read_lock() {
    down(count_lock);
    if (count == 0) {
        count++;
        down(db_lock);
    }
    up(count_lock);
}

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

2. In the above program, the reader has higher priority, which means that even if one reader is reading, no writer is allowed to access, and all other readers can read as well. If many readers request a read lock, the read lock won't be unlocked until all the reader requests are over, such that writer could be starving and waiting endlessly.

3. We just need to add one new semaphore, such that once a write lock is requested, the read has to end to process write. Only if no write lock is requested, the new read can be processed.

```
void read_lock() {
    down(read_lock);
    down(count_lock);
    if (count == 0){
        count++;
        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);
    up(read_lock);
}

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

4. The new approach clearly gives more priority to writers, in case one wants a balanced solution to this problem, it cannot be considered as solved. Otherwise we have to provide the above two solutions.