# VE482 Homework 5

Liu Yihao 515370910207

## Ex. 1 — Simple questions

1. Each process takes up at most one resource at a moment. When a process asks for the second resource, it can always take a free resource. So a deadlock can't occur.

2. The maximum value of $n$ is 5. The reason is similar to the previous question, each process takes up at most one resource, so there is always at least one free resource when $n \leqslant 5$.

3. $$\frac{35}{50} + \frac{20}{100} + \frac{10}{200} + \frac{x}{250} < 1$$
$$x < 12.5$$

   So the largest value for $x$ is 12.5.

4. It can be activated twice in a round. It can be used to implement a scheduler that each process have a priority. Then the more times a process occurs, the higher priority it is.

5. Yes. In source code, the program is likely to be I/O bound if it processes data along with I/O. Otherwise, it is likely to be CPU bound. In runtime, we can use monitor commands like iostat (monitor I/O) and top (monitor CPU) to determine it.

## Ex. 2 — Deadlocks

1. $$\begin{bmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{bmatrix}$$

2. Yes, the process can be finished in the order $P_2, P_4, P_5, P_1, P_3$.

3. Yes, for example, it can be completed as shown in the table.

| Process | Allocated | Maximum | Available |
|---|---|---|---|
| $P_2$ | 2 0 0 | 3 2 2 | 5 3 2 |
| $P_4$ | 2 1 1 | 2 2 2 | 7 4 3 |
| $P_5$ | 0 0 2 | 4 3 3 | 7 4 5 |
| $P_1$ | 0 1 0 | 7 5 3 | 7 5 5 |
| $P_3$ | 3 0 2 | 9 0 2 | 10 5 7 |

## Ex. 3 — Research

### Viruses

A computer virus is a type of malicious software program (”malware”) that, when executed, replicates itself by modifying other computer programs and inserting its own code. Infected computer programs can include, as well, data files, or the ”boot” sector of the hard drive. When this replication succeeds, the affected areas are then said to be ”infected” with a computer virus.

Virus writers use social engineering deceptions and exploit detailed knowledge of security vulnerabilities to initially infect systems and to spread the virus. The vast majority of viruses target systems running Microsoft Windows, employing a variety of mechanisms to infect new hosts, and often using complex anti-detection/stealth strategies to evade antivirus software. Motives for creating viruses can include seeking profit (e.g., with ransomware), desire to send a political message, personal amusement, to demonstrate that a vulnerability exists in software, for sabotage and denial of service, or simply because they wish to explore cybersecurity issues, artificial life and evolutionary algorithms.

Computer viruses currently cause billions of dollars' worth of economic damage each year, due to causing system failure, wasting computer resources, corrupting data, increasing maintenance costs, etc. In response, free, open-source antivirus tools have been developed, and an industry of antivirus software has cropped up, selling or freely distributing virus protection to users of various operating systems. As of 2005, even though no currently existing antivirus software was able to uncover all computer viruses (especially new ones), computer security researchers are actively searching for new ways to enable antivirus solutions to more effectively detect emerging viruses, before they have already become widely distributed.

### Worms

A computer worm is a standalone malware computer program that replicates itself in order to spread to other computers. Often, it uses a computer network to spread itself, relying on security failures on the target computer to access it. Worms almost always cause at least some harm to the network, even if only by consuming bandwidth, whereas viruses almost always corrupt or modify files on a targeted computer.

Many worms that have been created are designed only to spread, and do not attempt to change the systems they pass through. However, as the Morris worm and Mydoom showed, even these ”payload-free” worms can cause major disruption by increasing network traffic and other unintended effects.

### Trojans

In computing, a Trojan horse, or Trojan, is any malicious computer program which misleads users of its true intent. The term is derived from the Ancient Greek story of the deceptive wooden horse that led to the fall of the city of Troy.

Trojans are generally spread by some form of social engineering, for example where a user is duped into executing an e-mail attachment disguised to be unsuspicious, (e.g., a routine form to be filled in), or by drive-by download. Although their payload can be anything, many modern forms act as a backdoor, contacting a controller which can then have unauthorized access to the affected computer. Trojans may allow an attacker to access users' personal information such as banking information, passwords, or personal identity (IP address). It can infect other devices connected to the network. Ransomware attacks are often carried out using a Trojan.

Unlike computer viruses and worms, Trojans generally do not attempt to inject themselves into other files or otherwise propagate themselves.

### Reference

https://en.wikipedia.org/wiki/Computer_virus
https://en.wikipedia.org/wiki/Computer_worm
https://en.wikipedia.org/wiki/Trojan_horse_(computing)

## Ex. 4 — Programming

```cpp
//
// Created by liu on 17-11-7.
//

#include <ctime>
#include <vector>
#include <list>
#include <algorithm>
#include <iostream>
#include <random>

using namespace std;

void printRes(ostream &os, const vector<pair<size_t, size_t> > &resource) {
    os << "allocated ( ";
    for_each(resource.begin(), resource.end(), [&os](const pair<size_t, size_t>
    ↪ &p) {
        os << p.first << " ";
    });
    os << ") maximum ( ";
    for_each(resource.begin(), resource.end(), [&os](const pair<size_t, size_t>
    ↪ &p) {
        os << p.second << " ";
    });
    os << ")";
}

int main() {
    minstd_rand randEngine;
    randEngine.seed((size_t) time(0));

    size_t typeNum, processNum, availableInit;
    cout << "Input the number of resource types: ";
    cin >> typeNum;
    cout << "Input the number of processes: ";
    cin >> processNum;
    cout << "Input the number of resource initially: ";
    cin >> availableInit;
    cout << "Generate data: " << endl;
    list<pair<size_t, vector<pair<size_t, size_t> > > > data(processNum);
    vector<size_t> available(typeNum, 2);
    size_t i = 0;
    for (auto &process : data) {
        process.first = i++;
        process.second = vector<pair<size_t, size_t> >(typeNum);
        for (auto &resource: process.second) {
            size_t num = randEngine() % 10;
            resource.second = num;
            resource.first = num == 0 ? 0 : randEngine() % num;
        }
        cout << "P" << process.first << "\t";
        printRes(cout, process.second);
        cout << endl;
    }
    cout << "Begin simulation: " << endl;
    bool flag;
    do {
        flag = false;
        for (auto it = data.begin(); it != data.end();) {
            bool processable = true;
            auto it2 = available.begin();
            for_each(it->second.begin(), it->second.end(), [&it2,
            ↪   &processable](const pair<size_t, size_t> &p) {
                processable &= (p.second - p.first <= *(it2++));
            });
            if (processable) {
                flag = true;
                it2 = available.begin();
                for_each(it->second.begin(), it->second.end(),
                ↪   [&it2](pair<size_t, size_t> &p) {
                    *(it2++) += p.first;
                });
                cout << "Run P" << it->first << "\t";
                printRes(cout, it->second);
                cout << " available ( ";
                for_each(available.begin(), available.end(), [](const size_t &p)
                ↪   {
                    cout << p << " ";
                });
                cout << ")" << endl;
                it = data.erase(it);
            } else {
                ++it;
            }
        }
    } while (flag);
    if (data.empty()) {
        cout << "All completed" << endl;
    } else {
        cout << "Not completed, available( ";
        for_each(available.begin(), available.end(), [](const size_t &p) {
            cout << p << " ";
        });
        cout << ")" << endl;
        cout << "Remain processes:" << endl;
        for (auto &process : data) {
            cout << "P" << process.first << "\t";
            printRes(cout, process.second);
            cout << endl;
        }
    }
    return 0;
}
```

## Ex. 5 — Minix 3

For this part, we can search for the keyword “scheduling” in the minix source code folder and find the file “minix/kernel/main.c”. I enclosed some of the relating source code below:

```
/* 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 */
    DEBUGEXTRA(("initializing %s... ", ip->proc_name));
    rp = proc_addr(ip->proc_nr);          /* get process pointer */
    ip->endpoint = rp->p_endpoint;         /* ipc endpoint */
    rp->p_cpu_time_left = 0;
    if (i < NR_TASKS)               /* name (tasks only) */
        strlcpy(rp->p_name, ip->proc_name, sizeof(rp->p_name));

    if (i >= NR_TASKS) {
        /* Remember this so it can be passed to VM */
        multiboot_module_t *mb_mod = &kinfo.module_list[i - NR_TASKS];
        ip->start_addr = mb_mod->mod_start;
        ip->len = mb_mod->mod_end - mb_mod->mod_start;
    }

    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));

            /* 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. */
```

```c
        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"));
}
```

From the source code, we can simply find out that when the system starts, the kernel 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.

## Ex. 6 — The reader-writer problem

1. We can first use the semaphore count_lock to protect the counter, and then proceed with the counter and decide to lock or unlock the db according to the value of counter.

```c
void read_lock() {
    down(count_lock);
    if (counter++ == 0) down(db_lock);
    up(count_lock);
}

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

2. If there are so many readers that their reading time fill out the total time, no writer can enter the db because there are readers continuously coming.

3. We can down read_lock when a writer comes at the moment some readers are in the db. For new readers, down both count_lock and read_lock so that when a writer is waiting, the new reader can no longer enter the db. After the writer locks the db, we can up read_lock so that new readers can enter the db again after the writer leaves.

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

void read_unlock() {
    down(count_lock);
    if (--counter == 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. No, the priority of writer is higher, it is not solved.