

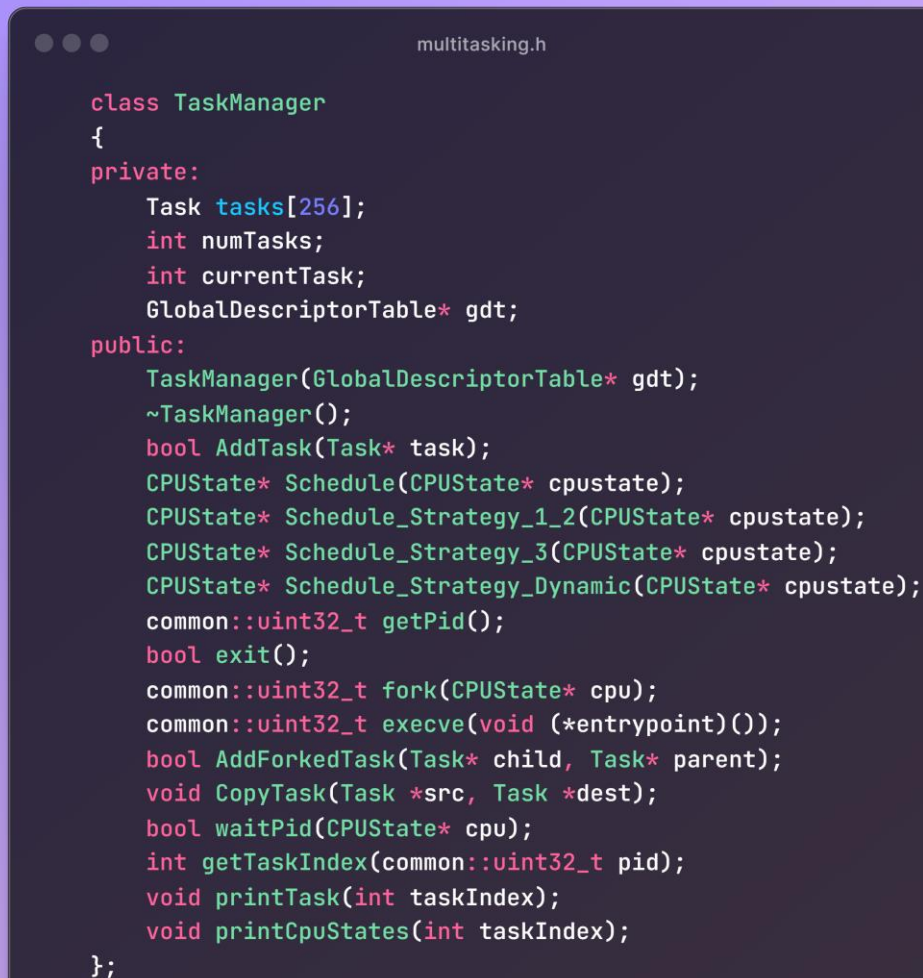
CSE 312 OPERATING SYSTEMS PROJECT 1 – REPORT

Murat Erbilici 200104004007

OVERVIEW

Firstly, I will show my additional functions and changes in the source code for different parts. After that, I will explain the details of my implementations and show outputs of the result for part A and part B (I couldn't implement part C).

1) Task Manager and Task



```
multitasking.h

class TaskManager
{
private:
    Task tasks[256];
    int numTasks;
    int currentTask;
    GlobalDescriptorTable* gdt;
public:
    TaskManager(GlobalDescriptorTable* gdt);
    ~TaskManager();
    bool AddTask(Task* task);
    CPUState* Schedule(CPUState* cpustate);
    CPUState* Schedule_Strategy_1_2(CPUState* cpustate);
    CPUState* Schedule_Strategy_3(CPUState* cpustate);
    CPUState* Schedule_Strategy_Dynamic(CPUState* cpustate);
    common::uint32_t getPid();
    bool exit();
    common::uint32_t fork(CPUState* cpu);
    common::uint32_t execve(void (*entrypoint)());
    bool AddForkedTask(Task* child, Task* parent);
    void CopyTask(Task *src, Task *dest);
    bool waitPid(CPUState* cpu);
    int getTaskIndex(common::uint32_t pid);
    void printTask(int taskIndex);
    void printCpuStates(int taskIndex);
};
```

In TaskManager, different from the source code, I have syscall function implementations (Details will be explained).

```
multitasking.cpp

CPUState* TaskManager::Schedule(CPUState* cpustate) {
    return Schedule_Strategy_1_2(cpustate);
    //return Schedule_Strategy_3(cpustate);
    //return Schedule_Strategy_Dynamic(cpustate);
}
```

I have different schedule algorithms for part A and part B. That's why I have one schedule algorithm and it returns the algorithm that will be tested.

```
multitasking.h

class Task
{
    friend class TaskManager;
private:
    static common::uint32_t nextPID;
    common::uint8_t stack[4096]; // 4 KiB
    CPUState* cpustate;
    TaskStatus taskStatus = READY;
    common::uint32_t priority;
    common::uint32_t runningTime;
    common::uint32_t pid = 0;
    common::uint32_t parentPid = 0;
    common::uint32_t waitPid = 0;
public:
    Task(GlobalDescriptorTable *gdt, void entrypoint(), common::uint32_t pid,
    common::uint32_t parentPid, common::uint32_t priority);
    common::uint32_t getPid();
    ~Task();
    Task();
    void InitTask(GlobalDescriptorTable *gdt, void entrypoint(),
    common::uint32_t pid, common::uint32_t parentPid, common::uint32_t priority);
};
```

I have pid, parentPid, waitPid, taskStatus and priority for process scheduling and syscalls. To make the processes' pids unique, I have nextPID.

2) SYSCALL FUNCTIONS AND SYSCALL HANDLER

```
kernel.cpp

int getpid() {
    int pid;
    asm("int $0x80" : "=c" (pid) : "a" (SYSCALLS::GETPID));
    return pid;
}

void execve(void (*entrypoint)()) {
    asm("int $0x80" : : "a" (SYSCALLS::EXECVE), "b" (entrypoint));
}

void exit() {
    asm("int $0x80" : : "a" (SYSCALLS::EXIT));
}

int fork(int *pid) {
    asm("int $0x80" : "=c" (*pid) : "a" (SYSCALLS::FORK));
    return *pid;
}

void waitPid(int pid) {
    asm("int $0x80" : : "a" (SYSCALLS::WAITPID), "b" (pid));
}
```

```

syscalls.cpp

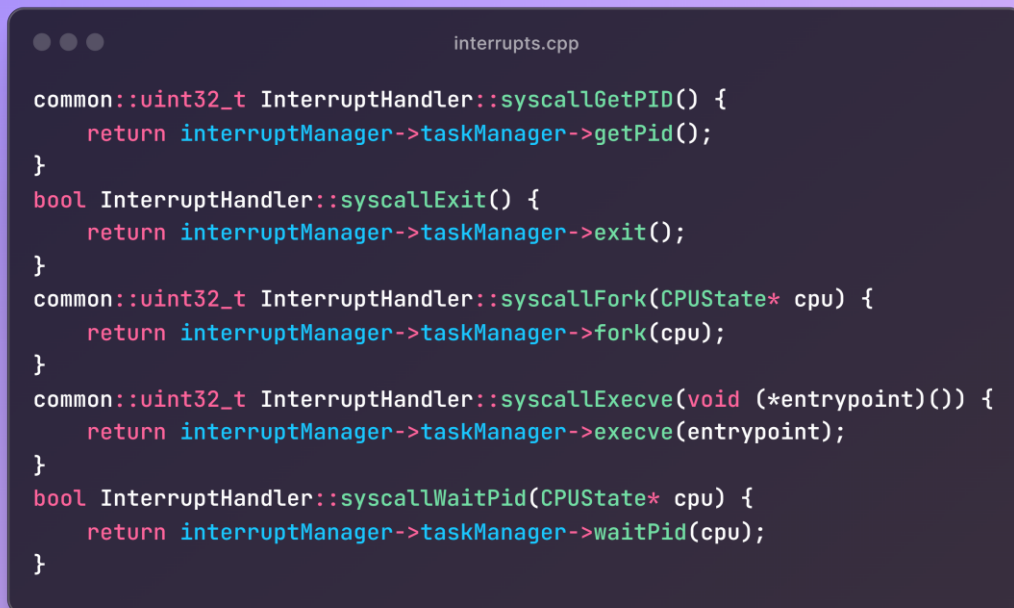
uint32_t SyscallHandler::HandleInterrupt(uint32_t esp)
{
    CPUState* cpu = (CPUState*)esp;

    switch(cpu->eax)
    {
        case SYSCALLS::GETPID:
            cpu->ecx = InterruptHandler::syscallGetPID();
            return InterruptHandler::HandleInterrupt(esp);
            break;
        case SYSCALLS::EXIT:
            InterruptHandler::syscallExit();
            return InterruptHandler::HandleInterrupt(esp);
            break;
        case SYSCALLS::FORK:
        {
            cpu->ecx = InterruptHandler::syscallFork(cpu);
            return InterruptHandler::HandleInterrupt((uint32_t)esp);
            break;
        }
        case SYSCALLS::EXECVE:
            esp = InterruptHandler::syscallExecve((void (*)()) cpu->ebx);
            break;
        case SYSCALLS::WAITPID:
            if(InterruptHandler::syscallWaitPid(cpu))
                return InterruptHandler::HandleInterrupt((uint32_t)esp);
            break;
        default:
            break;
    }

    return esp;
}

```

I used InterruptHandler class to call syscall functions because I needed to reach TaskManager and Engelmann's implementation has the TaskManager pointer in the InterruptManager and InterruptHandler has the InterruptManager pointer.



```
interrupts.cpp

common::uint32_t InterruptHandler::syscallGetPID() {
    return interruptManager->taskManager->getPid();
}

bool InterruptHandler::syscallExit() {
    return interruptManager->taskManager->exit();
}

common::uint32_t InterruptHandler::syscallFork(CPUState* cpu) {
    return interruptManager->taskManager->fork(cpu);
}

common::uint32_t InterruptHandler::syscallExecve(void (*entrypoint)()) {
    return interruptManager->taskManager->execve(entrypoint);
}

bool InterruptHandler::syscallWaitPid(CPUState* cpu) {
    return interruptManager->taskManager->waitPid(cpu);
}
```

3) IMPLEMENTATION DETAILS OF SYSCALLS

A-FORK

Some part, I took it from the teams page. Additionally, I invoke `initTask` because I need to copy process information to the tasks array as a new task.

```

multitasking.cpp

common::uint32_t TaskManager::fork(CPUState* cpu) {
    printf(" *FORK* ");

    if (numTasks >= 256)
        return -1;

    Task *parent = &tasks[currentTask];
    Task *child = &tasks[numTasks];
    child->InitTask(gdt, (void(*)()) parent->cpustate->eip, Task::nextPID, parent->pid, parent->priority);
    *(child->cpustate) = *cpu;

    tasks[numTasks].taskStatus=READY;
    tasks[numTasks].parentPid=tasks[currentTask].pid;
    tasks[numTasks].pid=Task::nextPID;

    for (int i = 0; i < sizeof(tasks[currentTask].stack); i++)
    {
        tasks[numTasks].stack[i]=tasks[currentTask].stack[i];
    }

    //Stackten yer alında cpustate'in konumu değişiyor bu nedenle şuanki taskın
    offsetini hesaplayıp yeni oluşan process'in cpu statenin konumunu ona göre
    düzenliyorum. Bu işlemi yapmazsam process düzgün şekilde devam etmiyor.
    common::uint32_t currentTaskOffset=((common::uint32_t)cpu - (common::uint32_t)
    tasks[currentTask].stack));
    tasks[numTasks].cpustate=(CPUState*)((common::uint32_t) tasks[numTasks].stack
    + currentTaskOffset);

    //Burada ECX' yeni taskın process id'sini atıyorum. Syscall'a return edebilmek
    için.
    tasks[numTasks].cpustate->ecx = 0;
    ++Task::nextPID;
    ++numTasks;
    return tasks[numTasks-1].pid;
}

```

B-EXIT

It makes the process status FINISHED. Before that, checks if it is child process and checks if it has parent or not. If it does, change waitPid to 0 in parent process. It is necessary because waitPid keeps the child pid that parent is waiting for. Making it zero allows us to change parent process's status to READY from BLOCKED (in the schedule function).

```

multitasking.cpp

bool TaskManager::exit() {
    if(tasks[currentTask].parentPid != 0) {
        int index = getTaskIndex(tasks[currentTask].parentPid);
        if(index != -1 && tasks[index].waitPid == tasks[currentTask].pid) {
            tasks[index].waitPid = 0;
        }
    }
    tasks[currentTask].taskStatus = FINISHED;
    return true;
}

```

C-EXECVE, GETPID, WAITPID

```

multitasking.cpp

common::uint32_t TaskManager::execve(void (*entrypoint)()) {
    printf(" *EXECVE* ");
    Task *curr = &tasks[currentTask];
    curr->InitTask(gdt, entrypoint, curr->pid, curr->parentPid, curr->priority);
    return (common::uint32_t) curr->cpustate;
}

common::uint32_t Task::getPid() {
    return pid;
}

bool TaskManager::waitPid(CPUState* cpu) {
    printf(" *WAIT* ");
    common::uint32_t pid = cpu->ebx;
    if(pid != tasks[currentTask].pid){
        int index = getTaskIndex(pid);
        if(index != -1 && tasks[index].taskStatus != FINISHED) {
            tasks[currentTask].waitPid = pid;
            tasks[currentTask].taskStatus = BLOCKED;
            return true;
        }
    }
    return false;
}

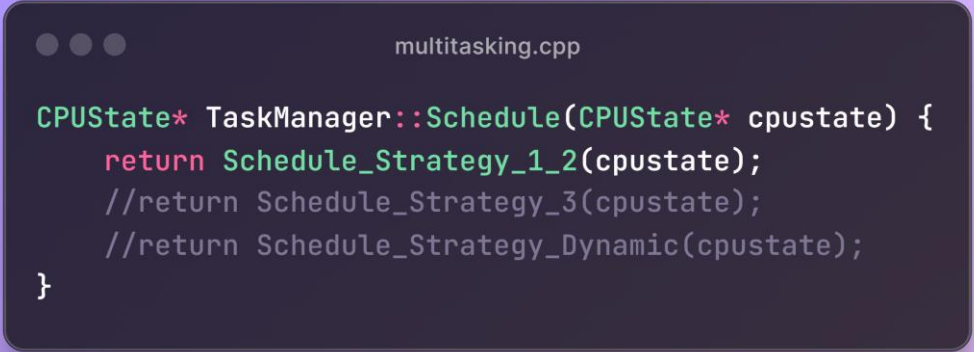
```

Execve is similar to fork but this time, we change the current process states.

Waitpid changes the parent process's status to BLOCKED if there is such a parent process.

4) SCHEDULING

I have a generic schedule function. Please test each part commenting out the other schedule strategies (The details are in the ReadMe.txt).



```
multitasking.cpp

CPUState* TaskManager::Schedule(CPUState* cpustate) {
    return Schedule_Strategy_1_2(cpustate);
    //return Schedule_Strategy_3(cpustate);
    //return Schedule_Strategy_Dynamic(cpustate);
}
```

Schedule_Strategy_1_2 is for Part A and the first and second strategy in Part B. This is a simple round robin algorithm like Engelmann's source code.

Schedule_Strategy_3 is for third strategy in Part B.

Schedule_Strategy_Dynamic is for Dynamic Priority Strategy in Part B.

The difference between third and dynamic one is that dynamic changes the priority of collatz each time some number of interrupts happen, third strategy executes only collatz until some number of interrupts happen and allows the other programs to be in schedule.


```

multitasking.cpp

CPUState* TaskManager::Schedule_Strategy_1_2(CPUState* cpustate)
{
    for(int i=0;i<numTasks;++i){
        //printTask(i);
        //printCpuStates(i);
    }
    if(numTasks <= 0) {
        return cpustate;
    }
    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    int nextTaskIndex = -1;
    if(tasks[currentTask].taskStatus == RUNNING) {
        tasks[currentTask].taskStatus = READY;
    }
    for (int i = 0; i < numTasks; ++i)
    {
        currentTask = (currentTask + 1) % numTasks;
        if(tasks[currentTask].taskStatus == BLOCKED) {
            if(tasks[currentTask].waitPid == 0) {
                tasks[currentTask].taskStatus = READY;
                nextTaskIndex = currentTask;
                break;
            }
        }
        if (tasks[currentTask].taskStatus == READY && currentTask != nextTaskIndex)
        {
            nextTaskIndex = currentTask;
            break;
        }
    }

    if (nextTaskIndex == -1) {
        tasks[currentTask].taskStatus = READY;
        return cpustate;
    }
    tasks[nextTaskIndex].taskStatus = RUNNING;
    return tasks[nextTaskIndex].cpustate;
}

```

```

CPUState* TaskManager::Schedule_Strategy_3(CPUState* cpustate)
{
    int isCollatz = 0;
    for(int i=0;i<numTasks;++i){
        // printTask(i);0
        //printCpuStates(i);
    }
    if(numTasks <= 0) {
        return cpustate;
    }
    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    int highestPriority = 40;
    int nextTaskIndex = -1;
    if(tasks[currentTask].taskStatus == RUNNING) {
        tasks[currentTask].taskStatus = READY;
    }
    for (int i = 0; i < numTasks; ++i)
    {
        currentTask = (currentTask + 1) % numTasks;
        if(tasks[currentTask].taskStatus == BLOCKED) {
            if(tasks[currentTask].waitPid == 0) {
                tasks[currentTask].taskStatus = READY;
            }
        }
        if(tasks[currentTask].pid == 1) {
            isCollatz = 1;
        }
        else {
            isCollatz = 0;
        }
        if (isCollatz == 1 && time <= 50)
        {
            if(tasks[currentTask].taskStatus == READY && currentTask !=
nextTaskIndex)
            {
                nextTaskIndex = currentTask;
                break;
            }
        }
        if(isCollatz == 0 && time <= 50 && tasks[currentTask].pid != 0)
            continue;
        if (tasks[currentTask].taskStatus == READY && tasks[currentTask].priority <
highestPriority)
        {
            highestPriority = tasks[currentTask].priority;
            nextTaskIndex = currentTask;
        }
    }
    if(time < 500)
        time++;
    currentTask = nextTaskIndex;
    if (nextTaskIndex == -1) {
        tasks[currentTask].taskStatus = READY;
        return cpustate;
    }
    tasks[nextTaskIndex].taskStatus = RUNNING;
    return tasks[nextTaskIndex].cpustate;
}

```

```

multitasking.cpp

CPUState* TaskManager::Schedule_Strategy_Dynamic(CPUState* cpustate)
{
    int isCollatz = 0;
    for(int i=0;i<numTasks;++i){
        // printTask(i);0
        //printCpuStates(i);
    }
    if(numTasks <= 0) {
        return cpustate;
    }
    if(currentTask >= 0)
        tasks[currentTask].cpustate = cpustate;

    int highestPriority = 40;
    int nextTaskIndex = -1;
    if(tasks[currentTask].taskStatus == RUNNING) {
        tasks[currentTask].taskStatus = READY;
    }
    for (int i = 0; i < numTasks; ++i)
    {
        currentTask = (currentTask + 1) % numTasks;
        if(tasks[currentTask].taskStatus == BLOCKED) {
            if(tasks[currentTask].waitPid == 0) {
                tasks[currentTask].taskStatus = READY;
            }
        }
        if(tasks[currentTask].pid == 1) {
            isCollatz = 1;
        }
        else {
            isCollatz = 0;
        }
        if (isCollatz == 1 && time <= 50)
        {
            if(tasks[currentTask].taskStatus == READY && currentTask !=
nextTaskIndex)
            {
                nextTaskIndex = currentTask;
                break;
            }
        }
        if(isCollatz == 1 && time % 20 == 0)
            if(tasks[currentTask].priority >= 1)
                tasks[currentTask].priority-=12;
            if (tasks[currentTask].taskStatus == READY && tasks[currentTask].priority <
highestPriority)
            {
                highestPriority = tasks[currentTask].priority;
                nextTaskIndex = currentTask;
            }
        }
    }
    if(time < 1000000)
        time++;
    else {
        time = 100;
    }
    currentTask = nextTaskIndex;
    if (nextTaskIndex == -1) {
        tasks[currentTask].taskStatus = READY;
        return cpustate;
    }
    tasks[nextTaskIndex].taskStatus = RUNNING;
    return tasks[nextTaskIndex].cpustate;
}

```

I didn't implement the third and dynamic strategy according to 5th interrupt. Since it is so fast, I tried to find optimum number to see better that it works properly. I use 40 as the highest number for priority (means lowest priority). Collatz has 39 as priority, other processes have 3 or 10.

5) TESTING AND OUTPUTS

Please test the program according to instructions in ReadMe.txt file.

I used characters between each collatz number while printing. Each process that runs collatz has different characters like process1: a10a a5a, process2: b30b b15b. In this way, we can easily see which process is running. In beginning of each schedule functions, to make the output clear, I commented out the functions that print process tables. If you want to print process tables, please uncomment these lines.

Part A Output:

```
*FORK* a30a *FORK* b60b *FORK* c90c *FORK* a15a b30b *LRP Result: 1 * *FOR
K* c45c *LRP Result: 36 * *FORK* a46a b15b *LRP Result: 225 * *WAIT* c136c a
23a b46b c68c a70a b23b c34c b70b a35a c17c a106a b35b c52c a53a b106b c26c a160
a b53b c13c a80a b160b c40c a40a b80b c20c a20a b40b c10c a10a b20b c5c a5a b10b
c16c a16a b5b c8c a8a b16b c4c a4a b8b c2c b4b a2a c1c b2b a1a b1b *WAIT* *WA
IT* *WAIT* *WAIT* *WAIT*
```

Part B First Strategy:

```
*FORK* a1a *FORK* b2b *FORK* c3c *FORK* d4d *FORK* b1b c10c e5e *FORK* d2d
f6f *FORK* e16e g7g *FORK* c5c d1f3f h8h *FORK* d g22g i9i *FORK* c16c e8e
h4h j10j *WAIT* *WAIT* *WAIT* f10f g11g i28i e4e j5j c8c h2h f5f g34g i14i c4
c e2e h1h j16j f16f e1e g17g i7i j8j c2c g52g c1c f8f i22i j4j g26g i11i j2j f4f
g13g i34i j1j f2f g40g i17i f1f g20g i52i g10g i26i g5g i13i g16g i40i g8g i20i
g4g i10i g2g i5i g1g i16i i8i i4i i2i i1i *WAIT* *WAIT* *WAIT* *WAIT* *WAI
T* *WAIT* *WAIT* ===== ST-1 End =====
```

Part B Second Strategy:

```

*FORK* *LRP Result: 0 * *FORK* *LRP Result: 1 * *FORK* *LRP Result: 9 * *
FORK* a1a *FORK* b2b *FORK* c3c *WAIT* *WAIT* *WAIT* *WAIT* *WAIT* b1b c1
0c c5c c16c c8c c4c c2c c1c *WAIT* ===== ST-2 End =====

```

Part B Third Strategy:

```

*FORK* -200- -100- -50- -25- -76- -38- -19- -58- -29- -88- -44- -22- -11- -34-
-17- -52- -26- -13- -40- -20- *FORK* *The Element(40) is found, index: 4.(Line
ar Search)* *FORK* *The Element(10) is found, index: 3.(Binary Search)* *FORK
* LRP Result: 36 *FORK* *200* *WAIT* *100* *50* *25* *76* *38* *19* *58* *2
9* *88* *44* *22* *11* *34* *17* *52* *26* *13* *40* *20* *10* *5* *16* *8* *4*
*2* *1* -10- -5- -16- -8- -4- -2- -1- *WAIT* *WAIT* *WAIT* *WAIT* *WAIT* E
ND

```

Part B Dynamic Priority Strategy:

```

*FORK* -200- -100- -50- -25- -76- -38- -19- -58- -29- -88- -44- -22- -11- -34-
-17- -52- -26- -13- -40- -20- *FORK* *The Element(40) is found, index: 4.(Line
ar Search)* *FORK* *The Element(10) is found, index: 3.(Binary Search)* *FORK
* LRP Result: 36 *FORK* *200* *WAIT* *100* *50* *25* *76* *38* *19* *58* *2
9* *88* *44* *22* *11* *34* *17* *52* *26* -10- -5- -16- -8- -4- -2- -1- *13* *4
0* *20* *10* *5* *16* *8* *4* *2* *1* *WAIT* *WAIT* *WAIT* *WAIT* *WAIT* E
ND

```

NOTE: In output screenshots, collatz numbers between ‘-’ character are from the collatz process that we set it the priority 39 (lowest priority). Other collatz numbers between ‘*’ are from the collatz process that has same priority as other programs like LRP(long_running_program), linear search etc. I use two collatz program to make program longer.