Operating System Assignment 2

Eran Aflalo 209343722

Omer Luxembourg 205500390

Part 1 – CPU Scheduling Implementation

In this part, we implemented a scheduling system to work on a given list of tasks.

The file is constructed by rows of task data - <task id>, <priority>, <arrival time>, <burst time>.

Build(char *line) – The functions build convert each line in the file to a struct called task that has the same fields (task id, priority …). The struct of a task is as follows:

```c
struct task
{
    int id;
    int priority;
    int arrival_time;
    int burst_time;
};
```

Table(char * file_name) - The functions return an array of tasks, we assumed that the MAX length of tasks is equal to 10. (MAX is a macro). The table will be created by a given file name on the system.

Display(struct task* array) - The function prints a table that shows (like in a practical session) the tasks in the file.

```
==============PROCESS===TABLE==================
ID      | Priority | Arrival Time   | Burst Time
--------+----------+----------------+-----------
     1  |       4  |             0  |       10
     2  |       3  |             3  |        8
     3  |       3  |             2  |        9
==============================================
```

Schedule(struct task* tasks_orig, enum Algorithm alg, int Q) – This function print the scheduling of an array of tasks.

Each algorithm has its function which implements it.

On the following page, is the Schedule main function. The parameter ‘Q’, is only for Round Robin algorithms.

```c
void Schedule(struct task* tasks_orig, enum Algorithm alg, int Q){
    struct task* tasks = copyTable(tasks_orig);
    int curr_time = 0;                          // cpu current time
    int ntasks = getNumTasks(tasks);            // getting the number of tasks
    switch (alg) {
        case First_Come_First_Serve:
            // First_Come_First_Serve Algorithm
            printf("Scheduling Tasks - First Come First Serve Algorithm:\n");
            for(int i=0; i<ntasks; i++){
                printFirstCome(tasks);  // here we dont need to know curr_time. we need only burst time
            }
            printf("\n");
            break;
        case Shortest_Job_First:
            // Shortest_Job_First Algorithm
            printf("Scheduling Tasks - Shortest Job First Algorithm:\n");
            for(int i=0; i<ntasks; i++){
                curr_time = printShortFirst(tasks, curr_time);     // iterativly searching for the shortest
job, in the times that are minimal than curr_time
            }
            printf("\n");
            break;
        case Priority:
            // Priority Algorithm - NON-PREEMPTIVE (static priority)
            printf("Scheduling Tasks - Priority (Non-Preemptive) Algorithm:\n");
            for(int i=0; i<ntasks; i++){
                curr_time = printPriorityFirst(tasks, curr_time);     // iterativly searching for the shortest
job, in the times that are minimal than curr_time
            }
            printf("\n");
            break;
        case Round_Robin:
            // Round Robin Algorithm - NON-PREEMPTIVE (static priority)
            printf("Scheduling Tasks - Round Robin (Non-Preemptive) Algorithm:\n");
            printRoundRobin(tasks, curr_time, Q, false);     // false is for no priority
            printf("\n");
            break;
        case Priority_With_Round_Robin:
            // Priority with Round Robin Algorithm - NON-PREEMPTIVE (static priority)
            printf("Scheduling Tasks - Priority with Round Robin (Non-Preemptive) Algorithm:\n");
            printRoundRobin(tasks, curr_time, Q, true);      // true is for priority
            printf("\n");
            break;
    }
}
```

Note that we have created a new copy of the tasks table, to work on it freely without erasing important information.

Algorithms:

- **First Come First Serve** – This function is getting the first tasks that had come. This is an iterative function, which means it will only print 1 job at a call.

```
// printing first arrived time task in tasks array
void printFirstCome(struct task* tasks){
    int min_arrival_time = 999999;
    int index = -1;
    for(int i=0; i<MAX; i++){        // finding the most recent arrival time
        if(tasks[i].arrival_time < min_arrival_time && tasks[i].id!=-1) {
            min_arrival_time = tasks[i].arrival_time;
            index = i;
        }
    }
    if(index >= 0){
        printf("<P%d,%d>", tasks[index].id, tasks[index].burst_time);
        tasks[index].id = -1;    // deactivate the task for future use
    }
}
```

Note that we used *task.id = -1* as a task that has been done.

- **Shortest Job First** – This function will get the shortest job that has arrived until the current CPU time, and then executes it. This is an iterative function, which prints 1 job at a call.

```
// printing the shortest job in the tasks table for the current cpu time
int printShortFirst(struct task* tasks, int curr_time){
    int min_burst_time = 999999999;
    int index = -1;
    while((index == -1) && (curr_time <= 999999999)) {
        for (int i = 0; i < MAX; i++) {
            if (tasks[i].arrival_time <= curr_time && tasks[i].id != -1 &&  // first check if we can use do this job
                min_burst_time > tasks[i].burst_time) {                      // taking shortest job
                min_burst_time = tasks[i].burst_time;
                index = i;
            }
        }
        if (index >= 0) {
            printf("<P%d,%d>", tasks[index].id, tasks[index].burst_time);
            tasks[index].id = -1;                    // deactivate the task for future use
            curr_time += tasks[index].burst_time;    // updating the current cpu time by this task burst time
        } else {       // index is -1, then there arent any tasks RIGHT NOW...
            curr_time += 1; // clock is ticking
        }
    }
    return curr_time;
}
```

This function will return the current CPU time, for searching for the tasks that have been arrived until that moment.

- **Priority** – This function will get the most prioritized job that has arrived until the current CPU time, and then executes it (higher priority is most urgent). This is an iterative function, which prints 1 job at a call.

```c
// printing the most prioritized job in the tasks table for the current cpu time
int printPriorityFirst(struct task* tasks, int curr_time){
    int max_priority = -1;
    int index = -1;
    while((index == -1) && (curr_time <= 999999999)) {
        for (int i = 0; i < MAX; i++) {
            if (tasks[i].arrival_time <= curr_time && tasks[i].id != -1 &&  // first check if we can use do this job
                max_priority < tasks[i].priority) {                         // taking max priority first
                max_priority = tasks[i].priority;
                index = i;
            }
        }
        if (index >= 0) {
            printf("<P%d,%d>", tasks[index].id, tasks[index].burst_time);
            tasks[index].id = -1;                       // deactivate the task for future use
            curr_time += tasks[index].burst_time;   // updating the current cpu time by this task burst time
        } else {      // index is -1, then there arent any tasks RIGHT NOW...
            curr_time += 1; // clock is ticking
        }
    }
    return curr_time;
}
```

Note that this function is very similar to the shortest job first, only that we check the priority field of the task structure, instead of the burst time.

- **Round Robin** – To implementing Round Robin algorithm we created a Queue system to handle incoming tasks.

```
// -- QUEUE IMPLEMENTATION --
int intArray[MAX];
int front = 0;
int rear = -1;
int itemCount = 0;

void Qinit(){
    front = 0;
    rear = -1;
    itemCount = 0;
}

int Qpeek() {
    return intArray[front];
}

bool QisEmpty() {
    return itemCount == 0;
}

bool QisFull() {
    return itemCount == MAX;
}

int Qsize() {
    return itemCount;
}

void Qinsert(int data) {

    if(!QisFull()) {

        if(rear == MAX-1) {
            rear = -1;
        }

        intArray[++rear] = data;
        itemCount++;
    }
}

int QremoveData() {
    int data = intArray[front++];

    if(front == MAX) {
        front = 0;
    }

    itemCount--;
    return data;
}
```

The queue will save us the index of the wanted task in the tasks table.

After that, we created the following function to handle the scheduling – note that this function is not iterative and prints the whole scheduled events.

```
// printing the task in a round robin manner
void printRoundRobin(struct task* tasks, int curr_time, int quantumTime, bool withPrior){
    int ntasks = getNumTasks(tasks);
    Qinit();
    bool busy = false;
    int busy_time = -1;
    int busy_index = -1;
    while( (!QisEmpty()) || (ntasks >0) || busy ){      // there are tasks in the table or in the queue that did not run yet or
running right now...
        // adding tasks to the queue by curr_time
        for (int i = 0; i < MAX; i++) {
            if (tasks[i].arrival_time == curr_time && tasks[i].id != -1){  // adding the tasks in the curr_time tick
                Qinsert(i);          // in the Q - the indices of the tasks (the table)
                ntasks--;            // this task is gone
            }
        }
        // running job did not finished
        if(busy == true && busy_time > 0){
            curr_time ++;                                    // continue ticking...
            busy_time --;                                    // 1 time ticked for busy curr_time
        }
            // running job finished
        else if(busy == true && busy_time == 0){
            if(tasks[busy_index].burst_time > 0){        // the busy task did not end yet
                Qinsert(busy_index);    // inserting Queue
            }// else: busy task is over - do not add it back to the queue
            busy = false;
            // NOW - it will try to accomplish the busy==true cases
        }

        // no running job, and no one is waiting.
        if(busy == false && QisEmpty()){
            curr_time ++;            // continue ticking...
        }
        // no running job but there is one waiting
        else if(busy == false){
            // getting the next index in the queue
            if (withPrior){
                busy_index = QremovePriorData(tasks);   // removing priority task from queue
            }else {
                busy_index = QremoveData(); // removing from queue
            }
            busy = true;
            busy_time = (quantumTime < tasks[busy_index].burst_time) ? (quantumTime) : (tasks[busy_index].burst_time);
            tasks[busy_index].burst_time -= busy_time;    // updating the task
            printf("<P%d,%d>", tasks[busy_index].id, busy_time);
            curr_time ++;                                    // continue ticking...
            busy_time --;                                    // 1 time ticked for busy curr_time
        }
    }
}
```

While we have tasks to do, we first add the tasks that are in the same arrival time as the CPU time, to the queue.
Next, we have 2 main conditions – busy, which means the CPU is running on a job, or not busy, which means we can do tasks from the queue if there are tasks.
Each task is being executed as the quantum time Q, or less (if it needs less time than Q…).

- **Priority with Round Robin** – To implement it, we only needed to change the way we remove data from the queue. Instead of getting the first item in the queue, we searched for the first item with the highest priority that exists in the queue.

```
int QremovePriorData(struct task * tasks){
    // this is liek removeData, only that we get the higher priority
FIRST, and fix the queue if needed
    int highestPrior = 0;
    for (int i=front; i!=(rear+1)%MAX; i=(i+1)%MAX){
        int taskIndex = intArray[i];
        if (tasks[taskIndex].priority > highestPrior)
            highestPrior = tasks[taskIndex].priority;
    }
    // now we got the highest priority in highestPrior
    int queueIndex = front;
    // searching for the first task with priority = highestPrior:
    while(tasks[intArray[queueIndex]].priority != highestPrior){
        queueIndex = (queueIndex + 1)%MAX;
    }
    int taskIndex = intArray[queueIndex];
    // now we got the index of the task we want to return

    // fixing the array (from front to the task index)
    for (int i=(queueIndex-1)%MAX; i!=(front-1)%MAX; i=(i-1)%MAX){
        intArray[(i+1)%MAX] = intArray[i];
    }
    front = (front+1)%MAX;
    itemCount--;

    return taskIndex;
}
```

Note that this function is an extended version of the QremoveData function(..) we showed earlier.

Results:

We created the main function to run on simple examples:

```c
int main() {
    // TEST FOR BUILD
    char *line = "15, 13, 2, 5";
    struct task task = Build(line);
    printf("id '%d', priority '%d', arrival_time '%d', burst_time '%d'\n",
           task.id, task.priority, task.arrival_time, task.burst_time);

    // TESST FOR TABLE
    char *file_name = "../processes.txt";
    struct task *tasks = Table(file_name);

    // TEST FOR DISPLAY
    Display(tasks);

    // TEST FOR SCHEDULER
    file_name = "../processes.txt";
    tasks = Table(file_name);
    Schedule(tasks, First_Come_First_Serve, -1);
    file_name = "../processes_SJF.txt";
    tasks = Table(file_name);
    Schedule(tasks, Shortest_Job_First, -1);
    file_name = "../processes_prior.txt";
    tasks = Table(file_name);
    Schedule(tasks, Priority, -1);
    file_name = "../processes_roundrobin.txt";
    tasks = Table(file_name);
    Schedule(tasks, Round_Robin, 2);
    file_name = "../processes_priorroundrobin.txt";
    tasks = Table(file_name);
    Schedule(tasks, Priority_With_Round_Robin, 2);
    return 0;
}
```

processes.txt

```
1, 4, 0, 10
2, 3, 3, 8
3, 3, 2, 9
```

processes_SJF.txt

```
1, 4, 1, 7
2, 3, 3, 5
3, 3, 3, 1
4, 0, 4, 2
5, 0, 5, 8
```

processes_prior.txt

```
1, 2, 0, 4
2, 4, 1, 2
3, 6, 2, 3
4, 10, 3, 5
5, 8, 4, 1
6, 12, 5, 4
7, 9, 6, 6
```

processes_roundrobin.txt

```
1, 0, 0, 4
2, 0, 1, 5
3, 0, 2, 2
4, 0, 3, 1
5, 0, 6, 3
6, 0, 6, 3
```

processes_priorroundrobin.txt

```
1, 0, 0, 4
2, 1, 1, 5
3, 2, 2, 2
4, 0, 3, 1
5, 0, 6, 3
6, 0, 6, 3
```

The terminal output is as we learned and solved by hand:

```
id '15', priority '13', arrival_time '2', burst_time '5'
==============PROCESS===TABLE==================
 ID     | Priority | Arrival Time   | Burst Time
--------+----------+---------------+------------
     1  |       4  |           0  |      10
     2  |       3  |           3  |       8
     3  |       3  |           2  |       9
===============================================
Scheduling Tasks - First Come First Serve Algorithm:
<P1,10><P3,9><P2,8>
Scheduling Tasks - Shortest Job First Algorithm:
<P1,7><P3,1><P4,2><P2,5><P5,8>
Scheduling Tasks - Priority (Non-Preemptive) Algorithm:
<P1,4><P4,5><P6,4><P7,6><P5,1><P3,3><P2,2>
Scheduling Tasks - Round Robin (Non-Preemptive) Algorithm:
<P1,2><P2,2><P3,2><P1,2><P4,1><P2,2><P5,2><P6,2><P2,1><P5,1><P6,1>
Scheduling Tasks - Priority with Round Robin (Non-Preemptive) Algorithm:
<P1,2><P3,2><P2,2><P2,2><P2,1><P1,2><P4,1><P5,2><P6,2><P5,1><P6,1>
```

Part 2 – XV6 Scheduling implementation

2.1: Priority-based Scheduler for XV6

In this part, we replaced the round-robin scheduler for xv6 with a priority-based scheduler. We have changed the *proc.c* and *proc.h* to work with priority – first added the priority field to process structure, then we changed the *scheduler* function in the *proc.c* file to work first on the highest prioritized processes.

proc.c

```
void
scheduler(void)
{
    // ------start edit : 20.12.2020------
   struct proc *p;
    struct proc *p_find;
    int highestPriority = 200;
    // ------ end edit ------------------
    struct cpu *c = mycpu();
    c->proc = 0;

    for(;;){
        // Enable interrupts on this processor.
        sti();
        // Loop over process table looking for process with highestPriority to run.
        acquire(&ptable.lock);
      for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
         // ------start edit : 20.12.2020-----
            if(p->state !=  RUNNABLE)  // not a runnable process
                continue;

         highestPriority = p->priority;
         for(p_find = ptable.proc; p_find < &ptable.proc[NPROC]; p_find++) {
            if (p_find->state != RUNNABLE)
               continue;
            if (p_find -> priority < highestPriority){
               // we found a more prioritised process
               p = p_find;        // jump to that priortised process
               highestPriority = p_find->priority;       // save the highest priority number
            }
         }
         // ------ end edit -------------------
            // Switch to chosen process.  It is the process's job
            // to release ptable.lock and then reacquire it
            // before jumping back to us.
            c->proc = p;
            switchuvm(p);
            p->state = RUNNING;

            swtch(&(c->scheduler), p->context);
            switchkvm();

            // Process is done running for now.
            // It should have changed its p->state before coming back.
            c->proc = 0;
        }
        release(&ptable.lock);
    }
}
```

As you can see, we used the same pattern as we implemented the priority with round-robin, in the previous section, to find the most prioritized task.

2.2: Add a Syscall to Set Priority

As we did in task 1, we added the system call *setpriority* to the XV6 system call and to the user space – by updating *syscall.h, user.h, usys.S, syscall.c, sysproc.c* according to the new changes (as done in task 1…).

To test the priority setter, and the scheduler, we created a user space program – *test_scheduling.c*.

```
#include "types.h"
#include "stat.h"
#include "user.h"

#include "fcntl.h"

////passing command line arguments

int main(int argc, char **argv)
{
    int pid = fork();
   for(int j=0; j<1000000; j++){}

    if(pid == 0){
        int newprior = atoi(argv[2]);
        int myPid = getpid();
        int oldPrior = setpriority(newprior);
        printf(1, "I'm the child - pid %d! change my priority to %d from %d\n",myPid,newprior,oldPrior);
      sleep(4);
        for(int i=0;i<10;i++){
         for(int j=0; j<1000000; j++){}
            printf(1, "<pid = %d run number %d: prior= %d>\n ", myPid,i,newprior);
      }
        exit();
    } else if(pid < 0){
        printf (1, "This is fork failed\n");
        exit();
    }else{
       int newprior = atoi(argv[1]);
       int myPid = getpid();
       int oldPrior = setpriority(newprior);
       printf(1, "I'm the father - pid %d! change my priority to %d from %d\n",myPid,newprior,oldPrior);
      sleep(4);
      for(int i=0;i<10;i++){
         for(int j=0; j<1000000; j++){}
            printf(1, "<pid = %d run number %d: prior= %d>\n ", myPid,i,newprior);
      }
    }
    wait();
    exit();
}
```

We created a user program that will spend a lot of CPU time on an empty for loop, to see when the CPU is given for each task.

The first try is to give the father lower priority (30) and the child higher priority (20), and the second try is the other way around:

father_priority = 30, child_priority=20:

```
cpu0: starting 0
sb: size 1000 nblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 32 bmap start
 58
init: starting sh
$ test_schedule 30 20
I'm the father - pid 3! change my priority to 30 from 50
I'm the child - pid 4! change my priority to 20 from 50
<pid = 3 run number 0: prior= 30>
 <pid = 3 run number 1: prior= 30<pid = 4 run number 0: prior= 20>
 <pid = 4 run number 1: prior= 20>
 <pid = 4 run number 2: prior= 20>
 <pid = 4 run number 3: prior= 20>
 <pid = 4 run number 4: prior= 20>
 <pid = 4 run number 5: prior= 20>
 <pid = 4 run number 6: prior= 20>
 <pid = 4 run number 7: prior= 20>
 <pid = 4 run number 8: prior= 20>
 <pid = 4 run number 9: prior= 20>
 >
 <pid = 3 run number 2: prior= 30>
 <pid = 3 run number 3: prior= 30>
 <pid = 3 run number 4: prior= 30>
 <pid = 3 run number 5: prior= 30>
 <pid = 3 run number 6: prior= 30>
 <pid = 3 run number 7: prior= 30>
 <pid = 3 run number 8: prior= 30>
 <pid = 3 run number 9: prior= 30>
 $
```

Here the child is stealing the CPU from the father, after it woken up from sleep.

father_priority = 20, child_priority=30:

```
 $ test_schedule 20 30
I'm the father - pid 5! change my priority to 20 from 50
I'm the child - pid 6! change my priority to 30 from 50
<pid = 5 run number 0: prior= 20>
 <pid = 5 run number 1: prior= 20>
 <pid = 5 run number 2: prior= 20>
 <pid = 5 run number 3: prior= 20>
 <pid = 5 run number 4: prior= 20>
 <pid = 5 run number 5: prior= 20>
 <pid = 5 run number 6: prior= 20>
 <pid = 5 run number 7: prior= 20>
 <pid = 5 run number 8: prior= 20>
 <pid = 5 run number 9: prior= 20>
 <pid = 6 run number 0: prior= 30>
 <pid = 6 run number 1: prior= 30>
 <pid = 6 run number 2: prior= 30>
 <pid = 6 run number 3: prior= 30>
 <pid = 6 run number 4: prior= 30>
 <pid = 6 run number 5: prior= 30>
 <pid = 6 run number 6: prior= 30>
 <pid = 6 run number 7: prior= 30>
 <pid = 6 run number 8: prior= 30>
 <pid = 6 run number 9: prior= 30>
 $
```

As we can see, after changing the priority and going to sleep, the CPU is given to the lower number priorty – higher priority process.

We can notice that the default priority is 50 as demanded.

Now we tested the same priority process number to check that they “share” the CPU time:

```
$ test_schedule 1 1
I'm the father - pid 7! change my priority to 1 from 50
I'm the child - pid 8! change my priority to 1 from 50
<pid = 7 run number 0: prior= 1>
 <pid = 7 run number 1: prior= 1>
 <pid = 7 run number <pid = 8 run number 0: prior= 1>
 <pid = 8 run number 1:2: prior= 1>
 <pid = 7 run numb prior= 1>
 <pid = 8 run number 2: prior= 1>
 <pid = 8 run number 3: prior= 1>
 <pier 3: prior= 1>
 <pid = 7 run number 4: prior= 1>
 <pid = 7 run number 5: prior= 1>
 <pid = 7 run nud = 8 run number 4: prior= 1>
 <pid = 8 run number 5: prior= 1>
 <pmber 6: prior= 1>
 <pid = 7 run number 7: prior= 1>
 <pid = 7 run numbid = 8 run number 6: prior= 1>
 <pid = 8 run number 7: prior= 1>
 <pid = 8 run number er 8: prior= 1>
 <pid = 7 run number 9: prior= 1>
 8: prior= 1>
 <pid = 8 run number 9: prior= 1>
 $
```

As we can see, they share the CPU and interrupting each other.