Operating System Project - Scheduler Report

Liu YuBo 18098537-I011-0082
Chen BoYu 1809853V-I011-0036

Content:

Introduction:

1. Project Introduction:

Processes are the basic execution entities in every modern operating system. Process control and management is therefore an important topic. In this project, we are going to implement a workable process scheduler in Linux systems. This process scheduler should read a description file which contains a number of jobs with different parameters (including arrival time, CPU requirements and the job commands).In our program, we need to implement a monitor and a scheduler.

For the monitor process, it stands between the scheduler and the job process. It is created by the scheduler process using the “exec()” system call. Its existence is only to serve one goal: to measure the real (or elapsed) time, the user time, and the system time taken of the job process. In this process, the monitor will receive the signals from scheduler then do the corresponding things to the job processes.

The scheduler “forks and executes” the monitor process with the command string supplied in the job description file as the only program argument of the monitor process. Then, it will execute the job process by the different three policies – FCFS, Non-preemptive SJF and Round-Robin.

2. Linux & Language

Linux: CentOS 6.1

Language: C

3. Task allocation

Chen Boyu: Monitor, FCFS and Non-preemptive SJF policy.

Liu Yubo: Monitor, Round-Robin and three Gantt charts.

● Monitor

In this process, it will have the same effect as the system – timer(). So, we use the timer() function in header file <sys/times.h>. Then, we can get all the times by the struct. Because of that, we create two of that – one is for the begin, other is to record the end time for the process. Also, use the tck to get the system clock(Figure 2.1)

```
struct tms start_t , end_t;//two
long tck = 0;
clock_t time_start , time_end ;//
```

Struct(Figure 2.1)

As the monitor should create the job process, we use fork() to create the child process and then use execvp() to deal with that. So, there is an array to store all the commands. It is noteworthy that because the usage of the execvp, the first character must be same as the file name(./monitor).

Then, the last value is the NULL. So, we add those into the array after we read the command.(figure 2.2)

```
for(i = 1; i < argc; i++){
    array[i-1] = argv[i];
}
array[argc - 1] = NULL;
```

Command array(figure 2.2)

To use the signals to control the job process, we use the signal function to do it. For SIGTERM, SIGTSTP and SIGCONT, we add a function to receive the signals and deal with it.(figure 2.3)

```
void signal_handle(int sig)
{
    if(sig == SIGTERM){
        kill(result, SIGTERM);  //kill job process
        }
    if(sig == SIGTSTP){
        kill(result, SIGTSTP);   //suspends child process
    }
    if(sig == SIGCONT){
        kill(result, SIGCONT);  //continues child process
    }
}
```

Signal(figure 2.3)

● Scheduler

1. Read File

The first thing we need to do is how to deal with the job description file. Because of the sample, we can know that it is made by three parts – Arrive time, Commands and duration time.(Figure 3.1_1)

```
0       ./while1   6
1       ./timer    3
2       ./timer    3
```

Job description file(Figure 3.1_1)

So, we create a struct named “sch” to store that. (Figure 3.1_2)

```
struct stu {
    int pid[MAXJOB];//store the job process pid
    int line[MAXJOB];//line number -> job id
    int start[MAXJOB];//arrival time
    char* cmd[MAXJOB][MAXCMD];//command
    int duration[MAXJOB];//duration time
    int size[MAXJOB];//command size
    char* job[MAXJOB][500];//use for draw the gantt
    int ctime[50];
    int Remain[200];
    int in[200];//flag
    int Finish[200];
}sch, temp_sch;//token for store the each part in job description file
```

Struct(Figure 3.1_2)

According to the demands of project, the max job number is 9. So, we define the MAXJOB to 9. At the same time, we assign the max length of all the commands is MAXCMD = 20. In this struct, we will store the time, the pid for each job, command and its size and so on.

So, in the main function, to delete the space and table between the command and two times in the job file. We use the strtok() function. In order to use this one, there is a char* named del which contains the \t and space. It will delete when the command has the del. Before that, we need to delete the \n first.(Figure 3.1_3)

```
char* del = "\t ";

while (fgets(temp[line], 100, fp)) {//read by one line
    token = strtok(temp[line], "\n");
    sch.line[line] = line;//get the job id
    temp_sch.line[line] = line + 1;//For SJF
    token = strtok(token, del);
    sch.start[line] = atoi(token);// start time
    temp_sch.start[line] = atoi(token);// start time for SJF

    sch.cmd[line][0] = "./monitor";
    while (token != NULL) {
        if (chno > 0) {
            sch.cmd[line][chno] = token;
        }
        chno++;
        token = strtok(NULL, del);
    }//store the cmd
```

Strtok()(Figure 3.1_3)

After we get all tokens, we need to store them into the struct. The first thing is that the line number is which the job number is, we assign it into the line[MAXJOB]. Then, for the arrival time and duration time, in the description file, they are the first and last items, so, we just to store the

begin and the end into the corresponding array. The important thing is that how to deal with the command. We use a loop to read the token, we skip the first one because it is arriving time, we set a 2-dimension array and the first dimension is MAXJOB which means the job number, the second dimension is each job's command. So, we can store different job's command easily.

We use the fopen() function to read that file. (Figure 3.1_4)

```
FILE* fp;
fp = fopen(filename, "r");

if (fp == NULL) {
    printf("Can't open!\n");
    exit(1);
}
```

Read file function(Figure 3.1_4)

2. FCFS

The basic logic of the FCFS policy is check the arriving time, it will do the job which is the first one coming here. And when we meet an infinite loop like ./while1, the program needs to kill it, also, when some job's time is too small like "ls", it will meet some error when terminate it.

So, we have two function to achieve it.

- Small job first

  In this one, we won't create the job process until is come in and the previous one is done. So we create all the job first and use sleep() function to control that the child processes will wait a corresponding period.(Figure 3.2_1)

```
for (i = 0; i < job; i++) {
    sch.pid[i] = fork();
    if (sch.pid[i] != 0) {
        if (sch.duration[i] != -1) {
            sleep(sch.duration[i]);
            kill(sch.pid[i], SIGTERM);
            wait(NULL);
        }
        else
            wait(NULL);
    }
    else {
        int size = sch.size[i];
        char* cmdarray[size];
        for (j = 0; j < size; j++) {
            cmdarray[j] = sch.cmd[i][j];
        }
        cmdarray[size] = NULL;
        execvp(cmdarray[0], cmdarray);
    }
}
```

(Figure 3.2_1)

Use execvp() to create the monitor and job process. After all job is finished, use wait(NULL) to clean zombie.

◆ Others

For another large job, we will use a loop to control the time for the scheduler, use sleep(1) to make it more reasonable.

The first thing is that when the time is same as the job's arrival time, we will set is as the job need to be running(Running = 1 means it needs to run, 0 means it finished). At the same time, we will add the arrive and duration time as the whole termination time for each job which will be indispensable for the next job's creation.(Figure 3.3_1)

```
for (i = 0; i < line; i++) {
    if (sch.start[i] == time) {
        running[i] = 1;
        if (time == 0) {
            temp_time[i] = sch.duration[i] - 1;
        }
        else {
            temp_time[i] = temp_time[i - 1] + sch.duration[i];
        }
    }
}
```

(Figure 3.3_1)

Then there is an array named finish[MAXJOB] which is used to show how a job is finished or not. When some thing is done, the finish[job] will be 1, oppositely it will be 0.

```
for (i = 0; i < line; i++) {
    if (temp_time[i] != 0 && temp_time[i] + 1 == time && time > 0) {
        finish[i] = 0;//not do the job
    }
}
```

To create the job file, we will separate the time 0 and others. Whatever the other is, the first job will do first. So, we execute it and change the finish[0] to 0 and running[0] into 0.

```
if (sch.start[i] == 0 && fir != 0 && time > 0) {
    int pid = Create(i);
    sch.pid[i] = pid;
    fir = 0;
    running[i] = 0;
    finish[i] = 1;
}
```

The other jobs need to check the running and finish array. It needs to make sure that this one is ready to run and the previous is finished. It is like the Philosopher question. Then, change the corresponding array number.

```
    if (running[i] > 0 && running[i - 1] == 0 && finish[i - 1] == 0) {
        running[i] = 0;
        finish[i] = 1;//do the job
        run_job = i;
    }
}
if (run_job != 0) {
    int pid = Create(run_job);
    sch.pid[run_job] = pid;
    run_job = 0;
}
```

Also, we will set an array called kpid[MAXJOB] to make the system know which one needs to be terminated. Then, when the kpid is qualified, we use the kill() function to terminate the job process. The pid is stored when we create the job processes.

```
for (i = 0; i < line; i++) {
    if (kpid[i] == 1) {
        if (sch.start[i] == 0) {
            sleep(sch.duration[i] - 1);
            kill(sch.pid[i], SIGTERM);
        }
        else if (sch.start[i] > 0) {
            kill(sch.pid[i], SIGTERM);
        }
        else if (sch.start[i] == 0 && sch.duration[i] == 1) {
            printf("Not kill\n");
        }
        kpid[i] = 0;
    }
    else if (kpid[i] == 0) {
        //do nothing
    }
}
```

3. Non-preemptive SJF

For SJF, we will do the shortest job first after the previous job is finished. So, the key point to achieve this policy is to sort the command and time. Besides, the job 0 is always did first, so we don't sort it.

```
for (j = i + 1; j < job; j++) {
    if ((sch.start[j] >= sch.start[i]) && (sch.duration[i] > sch.duration[j])) {
        temp1 = sch.start[j];//change the arrive time
        sch.start[j] = sch.start[i];
        sch.start[i] = temp1;
        for (k = 0; k < MAXCMD; k++) {
            temp_cmd[i][k] = sch.cmd[j][k];
            sch.cmd[j][k] = sch.cmd[i][k];
            sch.cmd[i][k] = temp_cmd[i][k];
        }//change the command
        temp2 = sch.duration[j];//change the duration time
        sch.duration[j] = sch.duration[i];
        sch.duration[i] = temp2;
        temp_line = temp_sch.line[j];//change the job num
        temp_sch.line[j] = temp_sch.line[i];
        temp_sch.line[i] = temp_line;
    }
}
```

From the figure, we will make sure that all the job is arrived first, then we check the duration time. If the duration time of the previous one is large than the next one, we will change the order of them – arrive and duration, command stored in the struct. There is a temp_sch which is used to

print gantt because it may be wrong if we use the sorted struct.

Then, it is same as the FCFS-small job first. We create all the process first and make them to sleep for a period. The job 0 doesn't be influenced. Use sleep() and execvp() to do that.

```
for (i = 0; i < job; i++) {
    sch.pid[i] = fork();
    if (sch.pid[i] != 0) {
        if (sch.duration[i] != -1) {
            sleep(sch.duration[i]);
            kill(sch.pid[i], SIGTERM);
            wait(NULL);
        }
        else
            wait(NULL);
    }
    else {
        int size = sch.size[i];
        char* cmdarray[size];
        for (j = 0; j < size; j++) {
            cmdarray[j] = sch.cmd[i][j];
        }
        cmdarray[size] = NULL;
        execvp(cmdarray[0], cmdarray);
    }
}
```

After all things are finished, use wait(NULL) to clean zombie.

4. Round-Robin

The RR is a non-preemptive algorithm, we give a slice=2 to RR algorithm, each process works a slice and change process so we need to have control about that.

Because of that, we need to consider sending the signal SIGTSTP and SIGCONT to stop the process and restart it. So, we can achieve the RR policy.

```
for (k = 0; k <jnum; k++) {
    if (nk == stu1.start[k] && stu1.in[k] == 0) {

            int id;
                stu1.in[k] = 1;
            stu1.pid[k] = fork();
            if (stu1.pid[k]!= 0) {
                if(stu1.duration[k]!=-1){
                    sleep(stu1.duration[k]);
                    kill(stu1.pid[k],SIGTERM);
                    wait(NULL);
                }
                else
                    wait(NULL);

            }
            else {

                execvp(command_array[0], command_array);
                }

    }
}
```

5. Gantt Chart

For the gantt chart, we make the first job to execute, and calculate the finish time, if the finish time is bigger to the arrival time of other process, we compare the duration time, if the duration time of process three bigger to process two, we will exchange all information, and store the final result, we also calculate the waiting time of process and store it to m.

It will be executed when the job processes are finished.

```
while(stu1.start[num]<=(stu1.start[t-1]+stu1.duration[t-1]+m)&&num<line){
            num++;
        }
        for(x=1;x<num-1;x++){
            for(c=t;c<num-x;c++){
                if(stu1.duration[c]>stu1.duration[c+1] && stu1.duration[c+1] != 0){
                    temp_arr = stu1.start[c];
                    stu1.start[c] = stu1.start[c+1];
                    stu1.start[c+1] = temp_arr;
                    temp_dur = stu1.duration[c];
                    stu1.duration[c] = stu1.duration[c+1];
                    stu1.duration[c+1] = temp_dur;
                }
            }
        }
    }
    if(i==0){
        m=(stu1.start[t-1]+stu1.duration[t-1]+m)-stu1.start[t];
    }
    if(i<stu1.start[t]){
        stu1.job[t+1][i]=" ";
    }
    else if(i<(m+stu1.start[t])&&i>=stu1.start[t]){
        stu1.job[t+1][i]=".";
    }
    else if(i>=(stu1.start[t]+m)&&i<(stu1.start[t]+stu1.duration[t]+m)){
        stu1.job[t+1][i]="#";
    }
```

● Operation Manual

For the Scheduler, we just need to input "./scheduler + filename + policy". Such as:

./scheduler job1 FCFS

./scheduler job1 SJF

./scheduler job1 RR

In the result, it will show the time of each job processes and after all processes are finished, it prints the gantt charts. One of the results is that(Figure 4.1)

```
[admin@localhost Desktop]$ ./1 job1 FCFS
PID 21384:     time elapsed is: 3.010000
             User time: 2.86000
            Sys time: 0.14000
Child TERM
PID 21386:     time elapsed is: 5.000000
             User time: 4.77000
            Sys time: 0.22000
Child TERM
PID 21388:     time elapsed is: 4.000000
             User time: 3.80000
            Sys time: 0.18000
Child TERM
PID 21390:     time elapsed is: 3.000000
             User time: 2.86000
            Sys time: 0.12000
Gantt Chart
Time  0                   1
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4
Job1  # # #
Job2    . . # # # # #
Job3      . . . . . # # # #
Job4        . . . . . . . . . # # #
Mixed 1 1 1 2 2 2 2 3 3 3 3 4 4 4
```

FCFS(Figure 4.1)

For the monitor, we just enter "./monitor + ./command". For example:

./monitor ./timer 3

./monitor ./while

./monitor ./sum

It will show the process id and three times. One of the results(figure 4.2)

```
[liu19991222@localhost Desktop]$ ./monitor ./timer 5
PID 5763:     time elapsed is: 5.020000
             User time: 4.77000
            Sys time: 0.23000
[liu19991222@localhost Desktop]$
```

./monitor ./timer 5(figure 4.2)

● Disadvantage

In Round-Robin, the time for different job may not be so explicit as the project demands. Also, we don't plan the time successful that it is difficult for us to design the Preemptive SJF. So, it will hint us to do anything regularly. Maybe next time, we should begin to do the program earlier.

● Conclusion

For this project, we meet many problems, like how to control the job process execution and how to draw the gantt chart.

Through our own efforts, we solved all the problems. During this project, we learn something more about the monitor and

scheduler. We can deal with different file format and store the information to struct. And learn to use signal and how to simulate the FCFS, SJF, RR algorithms.

So this is an unforgettable experience for us to do such a different project as a junior.

All in all, Thanks for the teacher’s guidance.