Amirkabir University of Technology

Computer engineering and it department

Operating Systems

Project 2

Authors:

Amir Hossein Rassafi

Mohammad Navid Shahsavari

Dr. Nastooh Taheri Javan

Deadline: 21 May 2019

AMIRKABIR

University of Technology

In this project, you are going to implement four scheduling policies that are described in following sections.

performance measurement

In order to implement the scheduling policies, you should modify the structure of operating system to be able to measure the performance of the system.

1. First, you should add some variables to **struct proc**. Add **etime**, **ctime** and **rtime** to **struct proc**. These variables describe the the creation time, end time and total running time and these data are enough for calculation of turnaround time and waiting time for each process. Each of these fields are described with CPU ticks. which means, you should assign CPU ticks to ctime at creation time of a new process. And at this time you should initialize the other variables. At each CPU tick, you should increase rtime of the running process. And etime should be set at end of the process. Note that the process may be gone to zombie state and the period of being zombie should not affect in turnaround time and waiting time.

2. Now, user should be able to access these data.
   Add a new system call as the following:

   int getPerformanceData(int *wtime, int *rtime)

   waittest.c is attached to test this system call.

Scheduling policies

We suggest to use #**IFDEF** macros to compile the OS. Modify the MakeFile in such way that the **SCHEDFLAG** phrase determines the scheduling policy. for example, the following code compiles the OS based on Round Robin policy:

make qemu SCHEDFLAG=RR

The default value of the SCHEDFLAG in MakeFile should be RR.

Round Robin

SCHEDFLAG = RR
Find the implementation of search algorithm of xv6 OS. This scheduler tries to find the first process with RUNNABLE status in process table and assign the turn to that process. At each clock of CPU, another process is selected to be replaced in CPU. Change this strategy such that the context switch operation occurs at specific clock numbers.
Add the following code to "param.h" and assign 5 to QUANTA.

#define QUANTA <int value>

Your code should do the context switching based on this number.
Test this algorithm using 3.1 section.

FIFO Round Robin

SCHEDFLAG = FRR
This strategy is same as Round Robin except that it selects next process based on FIFO strategy.
Test this algorithm using 3.2 section.

Guarnteed(Fair-Share) scheduling

SCHEDFLAG = GRT
Strategy of this method is based on the following formula. In this method, ratio of total running time to the total time in system is calculated and then, minimum of the ratio will be chosen.

$$\frac{\text{rtime}}{\text{current\_time} - \text{ctime}}$$

Note that, because of descrete time values, denominator could be zero.
Test this algorithm using 3.3 section.

Multi level queue scheduling

SCHEDFLAG = 3Q
this strategy has 3 queues:

- A queue for recording high priority processes that uses Guaranteed scheduling.
- A queue for recording mid priority processes that uses FIFO round robin strategy.
- A queue for recording low priority processes that uses Round robin strategy.

A high priority process should be served earlier than low priority process. This means that, until the first queue is not empty, other queues will not be used.
In order to access the process priority, we should change the struct proc. Our algorithm in this method has only 3 priorities: high, mid and low priority. The added variable should only have these 3 values.
But how to determine the priority of a process? At first, set all processes in start time to high priority. but define a function named **nice** to reduce priority of a process.
Add this system call to xv6:

Int nice()

which returns 1 if the operation is done and 0 for unsuccessful operation. Your system call should support any number of calling nice.
Test this algorithm using section 3.4.

Sanity test

In this section you are going to add some programs to calculate the performance of each algorithm.

3.1: Round Robin Sanity Test

Add a program named **RRsanity**. This program produces 10 childs using fork. Each child should print the following line 1000 times and in new lines.

Child <pid> prints for the <i> time.

which, pid is the process id of the child and i is a counter. After doing that, the child process exits and the parent waits for all children to finish their process and finally prints waiting time, running time and turnaround time for all the children.

3.2: FIFO Round Robin Sanity Test

In this section you should print pids of all processes existing in queue with in same order at each process changing operation.

<pid>,<pid>, ... ,<pid>

Make a program named **frrTest.c** which the parent process produces 10 child processes. Each child waits for a 1000 times loop and terminates. the parent process wait for all children processes to be terminated. Till that, you should print the pids at each process changing as stated above.

3.3: Guaranteed (Fair-Share) Sanity Test

Add a program named **Gsanity**. This program should print:

Father pid is <pid>

then sleeps for 10 seconds. after that, forks one time and each of parent and child processes print the following line in new lines:

process <pid> is printing for the <i> time

and then terminates.

3.4: Sanity Test

Add a program named **Sanity**. This program forks 30 children processes. Each of these processes has their unique id (named cid) from 0 to 29. All processes which their cid is divisible by 3, call nice system call one time. So their priority will reduce to mid. All processes which their remainder in division by 3 is 1, call nice system call 2 times and their priority will reduce to low. and so the processes with remainder 2 will be in high priority. Each child should print its cid 500 times and then calls exit. Parent process should wait for his child to be terminated. After termination of the children print the following:

- Average turnaround and waiting time of all children.
- Average turnaround and waiting time of each queue.
- Turnaround and waiting time for all children.