REQUIREMENT 1: TRACE SYSTEM CALL

Prerequisite Variable Declarations

Declare a variable named **mask** inside the **Proc** structure.

Define a system call **trace** inside **proc.c** whose sole purpose is to assign a **mask** value entered by the user.

Add a system call named **trace** to the **defs.h** file.

Include the system call **trace** in the **usys.pl** file.

Implementation

- Consider an example where the user enters the command: **strace 32 grep hello Readme** in the shell console.
- When the operating system (OS) encounters **strace**, it checks the user-defined file for **strace** and matches it with the number of arguments defined in the file.
- Once this matching is complete, the OS calls the **trace** system call and assigns **p->mask = 32**.
- The remaining part of the command to be parsed is **grep hello Readme**, which can be handled by the **exec** system call.
- The **exec** file forks the process and, at the same time, assigns the parent's **mask** value to the child process.
- Since it's a system call, the transition from user to kernel mode occurs, and this is where **trap.c** comes into play.
- During this transition, all values are saved in various registers named **a0-a7** using the **trapframe**.
- **a0** contains the system call number, and **a7** holds the value returned by the system call.
- Inside the **syscall()** function in **syscall.c**, various values are printed as per the requirements of the question.

REQUIREMENT 2: SCHEDULING

a) FCFS: First Come First Serve

In FCFS, alongside the default Round Robin mechanism, we select the process with the smallest creation time (CTIME) in the PROC table. This involves two iterations:

First Iteration:

- Declare a **`minproc`** variable and initialize it to 0.
- Within the loop, identify the runnable process with the lowest CTIME.
- After the first iteration, **`minproc`** points to the process with the least creation time (CTIME).

Second Iteration:

- In the second iteration, recheck if the process is still runnable.
- If it's runnable, follow these steps:
- Acquire the lock.
- Mark the process as running.
- Make the CPU point to the selected process.
- Switch the context.
- Remove CPU allocation.
- Release the lock.

Important Notes:

- CTIME is defined in **`proc.h`** and allocated to 'ticks' inside the **`allocproc`** function in **`proc.c`**.
- All schedulers run in an infinite loop.
- Interrupts must always be enabled.
- A structure named 'cpu' is maintained, containing a process pointer and context.
- Each process has its own context.
- Context switching is performed using **`swtch(&c->context, &p->context)`**.
-

How to Ensure FCFS is Running:

In the 'cmake' file, specify the scheduler as follows:

```
#ifdef SCHEDULER
SCHEDULER := DEFAULT | FCFS | PBS | MLFQ
#endif
```

To run with FCFS:

- Execute **`make CLEAN`**.

- Execute **`make QEMU SCHEDULER=FCFS`**.

b) PRIORITY BASED SCHEDULING

This scheduling method comprises two priority modes: Static priority and Dynamic priority.

Prerequisites and Declarations:

- In **`proc.h`**, declare **`rtime`**, **`ctime`**, and **`etime`** to store runtime, creation time, and end time.
- For PBS, declare additional variables: **`s_starttime`**, **`stime`**, and **`static priority`**.
- Inside **`allocproc`** in **`proc.c`**, initialize **`rtime=0`**, **`ctime=ticks`**, **`etime=0`**, **`static priority=Default_PRIORITY (60)`**, **`s_start_time=0`**, and **`stime=0`**.
- Utilize **`wakeup()`** and **`sleep()`** functions to handle sleeping and calculate **`stime`**.
- To update **`rtime`**, implement it in **`trap.c`** as follows: **`Trap.c -> devintr() -> clockintr() -> updatetime() -> rtime++`**.

Implementation of PBS

- Declare a **`proc`** pointer **`*hp`** (high priority) and set it to 0. Also, declare **`min_dynamic_priority`** as 101 (greater than max, i.e., 100).
- The scheduler runs in an infinite loop, and the objective is to schedule the process with the lowest Dynamic Priority (DP).
- Iterate through the **`proc`** array until **`NPROC`** to check for any runnable processes.
- Initialize **`niceness`** to 5, and if the following condition is met:

```
 if (p->rtime + p->stime != 0)
niceness = (int)((p->stime / (p->rtime + p->stime)) * 10);
```

- Use **`niceness`** to calculate DP for each process as defined in the question.
- In each iteration, find DP and compare it with the existing DP to find the runnable process with the lowest DP value. After the last iteration, you will have **`hp`**, which is the process with the highest priority.
- Now, schedule **`hp`**.
- Acquire a lock.
- Make the process running.
- Make the CPU point to the selected process.
- Switch the context.
- Remove the CPU allocation.

- Release the lock.

SET_PRIORITY SYSTEM CALL

- This system call takes two arguments: Priority and process ID.
- It searches through the entire **PROC** table and checks for the entered process ID.
- Once a match is found, it sets the current priority to the old priority (**old_sp**) and sets **curr->priority** as the entered priority.
- Then, it calculates new DP and old DP and compares them. If **new_dp < old_dp**, it calls **yield()** to give up the CPU.
- Important note: This process occurs for running processes.
- Define **set_priority** in **def.h**.
- Define it in **entry()** in **usersys.pl**.
- Also, define it in **syscall.h** to make the system recognize it as a system call.
-

MULTILEVEL FEEDBACK QUEUE (MLFQ)

Prerequisites and Declarations:

- Define **p->entrytime** as **ticks**.
- Create a **queueticks** array for each process capable of storing values from 0 to 4.
- Declare a **chosenproc** pointer of **proc** type and initialize it to 0.
- Set the highest queue value to 5.
- Define a predefined **waiting_limit** variable.
- Maintain a **times_scheduled** variable for each process.

Implementation:

- One crucial aspect of MLFQ is the implementation of aging.
- Iterate through the loop to find runnable processes.
- When a runnable process is found, check the **p->entrytime** of that specific process. Subtract it from the total ticks, which gives you the waiting time of the process in that queue. If that value is greater than the **waiting_limit**, then:
- Acquire the lock.
- Upgrade the queue of the process.
- Store the new **p->entrytime** of the process in that queue.
- Release the lock.
- Next, choose the process:
- Run a loop for the **PROC** array.
- Find the runnable process.
- Identify the process in the lowermost queue and set it as the chosen process. Also, set the corresponding queue as the highest queue.

- If two processes are in the same queue of the highest priority, break the tie using the one with the lesser **p->entrytime**.
- Schedule the process:
- Acquire a lock.
- If **chosen_proc** is runnable, increase the **times_scheduled** variable and store the entry time of that process.
- Make the process run.
- Switch the context.
- Once completed, increase the **queue_ticks** variable of that process.
- Release the lock.

REQUIREMENT 3: PROCDUMP( )

Prerequisites

1. Ensure that all the required variables, including **rtime**, **endtime**, **no_of_times_Scheduled**, and the **queue_ticks** array for MLFQ, are defined within the respective scheduler implementations.

Implementation

- The printing of process information on the console is handled by the **procdump()** function, which is defined within **Proc.c**.
- To display process information, iterate through each process in the **proc** array.
- For the default or FCFS scheduler, print **P->id**, **state**, **rtime**, **etime**, and **no_of_times_scheduled**.
- For the PBS scheduler, additionally print **DP** (dynamic priority) along with the previous information.
- For the MLFQ scheduler, include the **queue_ticks[I]** value along with the other details.
- This information should only be printed when the user presses **Ctrl + P**. The **procdump()** function is called from a case in **console.c**.

REQUIREMENT 4: PERFORMANCE COMPARISON

Prerequisites

- Implement the **`waitx()`** system call to calculate the waiting time (**`wtime`**) and running time (**`rtime`**) of exited processes.
- Create a file named **`schedulertest`** and fork approximately 10 dummy processes for performance testing
- Simulate both CPU-bound and I/O-bound processes in your test cases.

Implementation

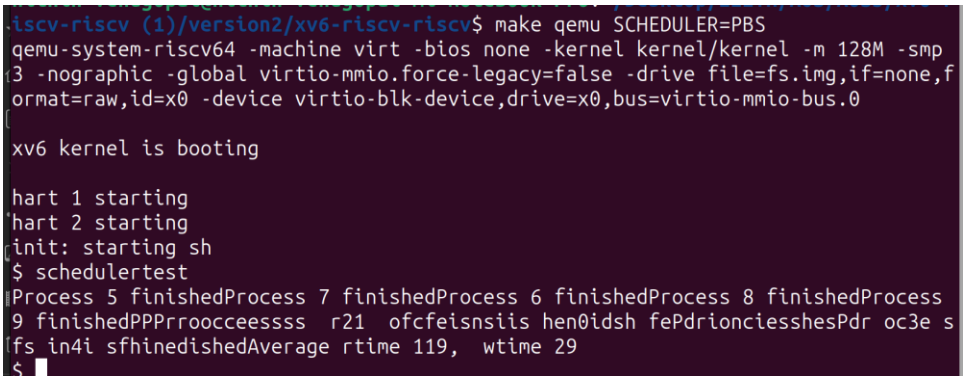
- To initiate the performance comparison, enter the command **`schedulertest`** in the console.

- For PBS scheduler, invoke the **`set_priority`** function within your test cases.

FCFS

MLFQ

PBS


iscv-riscv (1)/version2/xv6-riscv-riscv$ make qemu SCHEDULER=PBS
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ schedulertest
Process 5 finishedProcess 7 finishedProcess 6 finishedProcess 8 finishedProcess
9 finishedPPPrroocceessss r21 ofcfeisnsiis hen0idsh fePdrionciesshesPdr oc3e s
fs in4i sfhinedishedAverage rtime 119, wtime 29
$