Lab 5 New Scheduler in xv6
Report

-- 140050002, 140050007

Modification in Data Structure :

We have added two member (**prio** and **cprio**) of type int to struct proc.

prio stands for priority of process. If not mentioned same as parent.

cprio stands for current priority. (explained during policy)

Scheduling Policy :

We are implementing weighted round robin policy with O(1) complexity.

Let one unit of time be time after which timer interrupt occurs after process start executing and process did not go through blocking system calls nor any other interrupts. This is usually on every clock tick in a simple round robin scheduler.

Whenever process timer interrupt occurs, we check for how many units of time (say T_curr) process has run so far.

If T_curr is equal to multiple of (1 + priority) then process yields.

Otherwise we let process continue without yielding.

T_curr is maintained in **cprio** as (priority - time_run_so_far) modulo (priority + 1)

By this we yield a process with say priority 3 only after it runs for 4 clock ticks as opposed to another process with priority 0 which will yield immediately after every clock tick.
Thus we ensure that a higher priority process is given more processing time as compared to a lower priority process

Implementation Details :

1. To create custom syscalls first we add our new sys_setprio and sys_getprio functions in "**sysproc.c**"(line 93).
2. Next we add their system call numbers in "**syscall.h**"(line 23) and create the mappings to the sys_functions in "**syscall.c**"(line 129).
3. Then we expose the function to users in "**user.h**"(line 26) and finally add the corresponding new macros in "**usys.S**"(line 32).
4. The "**testmyscheduler.c**" code calls both the syscalls viz. getprio() and setprio(int) and we compile this user program by adding this file in EXTRA of "**Makefile**"(line 247) and set the user program executables in UPROGS of "**Makefile**"(line 176).
5. Every process now has two new parameters, namely **prio**(int) and **cprio**(int) set in "**proc.h**"(line 66).
6. The default value of the priority of a process is the same as that of its parent set in "**proc.c**"(line 164) and we set our "**initcode**" process to **priority 0** via "**proc.c**"(line 93). Hence any user program will have the default priority of 0.
7. To create the logic of running the process for (priority+1) number of clock ticks, we maintain the value of "**cprio**" to the remaining **number of clock ticks** which the process **can run before it yields**(forcefully make the process to give up CPU) "**trap.c**"(line 106). We forcefully yield a process only when **cprio becomes zero**.
8. The **testmyscheduler** user program generates child processes and we print the approx. time each child process spends to end.
    a. It generates 3 CPU bound processes with priority 0, 1, 2 and we observe that higher priority takes less time.
    b. It also generates 2 I/O bound processes with priority 0 and 10 and we observe that the higher priority takes less time.

Corner Cases :

Default priority is explained in the 6th point of Implementation. Basically we set the "**initcode**" process to **priority 0** and any other process to initialize to the **same priority as its parent**.

In case of multi-core cpu, when a process is scheduled, it is yielded only after it runs for time proportional to (1 + priority). For new process that is runnable, it runs on idle core or waits in the way same as in round robin policy.

When process blocks before it's quantum finishes same behaviour is default xv6 case since we are not modifying any trape frame or code related to blocking. Same way in other corner cases we will observe same behaviour as we do in default xv6.

Test Case and Observations :

To run simply replace all the given files in the source directory of xv6 code and then type "**make qemu**" in terminal.
In the qemu terminal that opens type "**testmyscheduler**" to start the scheduler testing user program.

The following test cases are on a 2 core system
CPU - Bound Process :
$ testmyscheduler
Testing schedular for CPU-Bound processes
Child 1 has priority 0
Child 2 has priority 2
Child 3 has priority 1
Child 2 runs for 1878 (0)
Child 3 runs for 2045 (0)
Child 1 runs for 2983 (0)

Clearly the Child with higher priority finishes before other lower priority processes.
In child 1 we actually do not set any priority and it gets the default priority.
Only 2 CPU bound processes are not sufficient to test as we chose a 2 core processor else every process will get assigned to a particular core which we do not wish for.

I/O - Bound Process :
Testing schedular for IO-Bound processes
Child 1 has priority 4
Child 2 has priority 0
Child 1 runs for 1016
Child 2 runs for 2534

Clearly the Child with higher priority finishes before other lower priority processes.
In child 2 we actually do not set any priority and it gets the default priority.
We simply **write** a lot of bytes to a file in both the children which makes these processes I/O bound. The created files can be seen if we do "**ls**" inside the QEMU simulation terminal.