REPORT

Requirement 1: System Call:

Strace System Call:

"Strace mask command [args]"

Here mask refers to bit set to 1 that is it will check system calls mapped to that set bit in binary expansion of that mask number .

Command will be a XV6 command for which value is to be checked.

Args refers to argument for that command

CHANGES MADE IN DIFFERENT FILES :

1) Defs.h-
   - Changed the signature of argint, argstr , argaddr (changed void to int)
   -Added declaration of trace as void(trace uint 64);

2) Proc.c
   - Defined the trace function with its definition.

3) kernel/syscall.h
   -Added/Assigned Number to the system call .
4) Syscall.c
   - Added SYS_trace- maps constant to appropriate handler function.

- Prototype declaration extern uint 64 sys_trace(void)

5) Sysproc.c
   - Defined the sys_trace function
6) kernel/proc.h
   - In structproc added the variable for mask .

7) User.h
   - Added prototype int trace(int);
8) user/usys.pl
   - Added a new entry for trace function.
9) Strace.c
   - Added the file with a user level program to get the requirement.
10) Syscall.c
   - Added the syscall names
   - Modified syscall( ) function which retrieves the current process using myproc( ) to print the required result.

Requirement 2: Scheduling:

1. First-come, first-serve

This is a non-preemptive policy that selects the process with the lowest creation time (creation time refers to the tick number when the process was created). The process will run until it no longer needs CPU time.

Modifications :

In kernel/proc.c we ran a loop to search for the process with lowest creation time .

To disable the preemption call to yield( ) in usertrap( ) and kerneltrap( ) is disabled conditionally for FCFS.

2. Priority-based

This is a non-preemptive priority-based scheduling policy that selects the process with the highest priority for execution.

We can set priority by using

“ setpriority [priority] [PID] “

Here, we have static priority and dynamic priority. Dynamic priority varies with running time and sleeping time and decides scheduling. Static priority is used to calculate dynamic priority.

Implementation

- We ran a loop to search a process with highest priority.
- To measure the sleeping time, when the process is sent to sleep via `sleep()` in `kernel/proc.c`, the number of ticks is stored in `struct proc::s_start_time`. Then, when `wakeup()` in `kernel/proc.c` is called, the difference between the current number of ticks and the previously stored time is stored in `struct proc::stime` as the sleeping time.
- Only the static priority (60 by default) is stored in `struct proc`. The niceness and the dynamic priority are calculated in the loop, when the process to be scheduled is being selected.
- As in FCFS, the call to `yield()` has been conditionally disabled for PBS.

- The `set_priority()` system call can be used to change the static priority of a process. It has been implemented in the same manner as in specification 1. A user program has also been implemented.

" setpriority [priority] [PID] "

3. Multilevel Feedback queue scheduling (MLFQ):

Implemented a simplified preemptive MLFQ scheduler that allows processes to move between different priority queues based on their behavior and CPU bursts.
● If a process uses too much CPU time, it is pushed to a lower priority queue,
leaving I/O bound and interactive processes in the higher priority queues.
● To prevent starvation, implement aging.

Implementation

- The five priority queues have not been implemented physically; rather they have been stored as a member variable `current_queue` of `struct proc`. This facilitates adding and removing processes and "shifting between queues" by just changing the number, and eliminates the overhead in popping from one queue and pushing to another.
- Each queue has a time-slice as follows after which they are demoted to a lower priority queue (`current_queue` is decremented).

1. For priority 0: 1 timer tick

2. For priority 1: 2 timer ticks
3. For priority 2: 4 timer ticks
4. For priority 4: 8 timer ticks
5. For priority 4: 16 timer ticks

Aging has been implemented, just before scheduling, via a simple for loop that iterates through the runnable processes, and promotes them to a higher-priority queue

- Demotion of processes after their time slice has been completed is done in `kernel/trap.c`, whenever a timer interrupt occurs. If the time spent in the current queue is greater than $2^{(\text{current\_queue\_number})}$, then it is demoted (`current_queue` is incremented).
- The position of the process in the queue is determined by its `struct proc::entry_time`, which stores the entry time in the current queue. It is reset to the current time whenever it is scheduled, making the wait time in the queue 0.
- If it is relinquished by the CPU, its entry time is again reset to the current number of ticks.

REQUIREMENT 3 : procdump

procdump is a function that is useful for debugging (see kernel/proc.c). It prints a list of processes to the console when a user types ctrl-p on the console. We have extended this function to print more information about all the active processes.

- process ID
- priority (PBS and MLFQ only)
- state
- running time (`struct proc::rtime`)

- waiting time (current ticks or `struct proc::etime` - creation time - running time)
- number of times scheduled (stored in `struct proc::no_of_times_scheduled`)
- q_i (MLFQ only) (number of ticks the process spent in each of the 5 queues)

REQUIREMENT 4 : PERFORMANCE

- Round-robin (default)

Average running time: 173 ticks
Average waiting time: 18 ticks

- First-come, first-serve

Average running time: 174 ticks
Average waiting time: 39 ticks

```
xv6 kernel is booting

init: starting sh
$ schedulertest
Process 0 finishedProcess 1 finishedProcess 2 finishedProcess 3 finishedProcess 4 finishedProcess 5 finishedProcess 6 finishedProcess 7 finishedProcess 8 finishedProcess 9 finishedAverage rtime 174,  wtim
e 39
$ QEMU: Terminated
```

- Priority-based

Average running time: 137 ticks
Average waiting time: 18 ticks

```
xv6 kernel is booting

init: starting sh
$ schedulertest
Process 5 finishedProcess 6 finishedProcess 7 finishedProcess 8 finishedProcess 9 finishedProcess 0 finishedProcess 1 finishedProcess 2 finishedProcess 3 finishedProcess 4 finishedAverage rtime 137,  wtim
e 18
$ QEMU: Terminated
```

- Multi-level feedback queue

Average running time: 172 ticks
Average waiting time: 18 ticks

```
xv6 kernel is booting

init: starting sh
$ schedulertest
Process 8 finishedProcess 9 finishedProcess 5 finishedProcess 6 finishedProcess 7 finishedProcess 0 finishedProcess 1 finishedProcess 2 finishedProcess 3 finishedProcess 4 finishedAverage rtime 172,  wtim
e 18
$
```

Clearly, FCFS performs the worst, as it may do badly if a CPU-bound process which takes a longer time is scheduled first, increasing the waiting time for all other processes. (CONVOY EFFECT)

PBS performs the best, followed by MLFQ, and then RR.

These results have been obtained by the `user/schedulertest.c` benchmark program.