Tweaked xv6

Implementation

Task 1

Strace

```
strace mask command [args]
```

It intercepts and records the system calls which are called by a process during its execution. It should take one argument, an integer mask, whose bits specify which system calls to trace. To trace the ith system call, a program calls strace 1<<i, where i is the syscall number. Added a sys_trace() function in kernel that implements the new system call by remembering its argument in a new variable in the proc structure. The functions to retrieve system call arguments from user space are in kernel. Modified fork to copy the trace mask from the parent to the child process. Modified the syscall function in kernel to print the trace output. Created a user program in user, to generate the user-space stubs for the system call, add a prototype for the system call to user, a stub to user, and a syscall number to kernel. The Makefile invokes the Perl script in user, which produces assembly for that, the actual system call stubs, which use the RISC-V ecall instruction to transition to the kernel.

Task 2

FCFS Scheduler

```
make qemu SCHEDULER=FCFS
```

FCFS scheduler works on a non-preemptive basis. As the name suggests, it is first come first serve logic, and processes are executed in their entirety on the basis of when they were first created. This is implemented by using ctime, added by us in proc structure in task 1. Every iteration of the loop we find the process with the earliest creation time and execute it.

Priority Based Scheduler

```
make qemu SCHEDULER=PBS
```

PBS works on the principal of executing processes having the highest priority, and tie is broken by the time process is scheduled and start time of the process. The priority is added as an extension to the proc structure, and can be modified with the help of the setpriority() system call, that accounts for errors and updates the value accordingly. As PBS is preemptive in nature, we have to ensure that before a process is selected, we check for lower priority each time. This is implemented by iterating through the process table, and before selecting a process, we iterate and find the minimum priority. If the priority is minimum, then the chosen process is executed, else it iterates through the process table again.

Multi-Level Feedback Queue

```
make qemu SCHEDULER=MLFQ
```

Multi-Level feedback queue uses 5 different priority queues to control the order of execution. It is preemptive in nature. The queues are not implemented separately. Instead, the proc structure has been extended to include a pointer to another proc struct. Thus, the queues have been implemented by using linked lists. Push and pop have been implemented as different functions using standard logic of linked lists, and we have 5 extra proc structures that denote the head of each queue. The processes are chosen in order from queues in higher to lower priorities, and their CPU time is monitored in trap.c. In case the elapsed run time exceeds the given number of ticks, then the process is preempted and shifted to a lower queue. On initialization, all processes are pushed into the highest priority queue, and all those waking from sleep into the original queue, using the push() call. Aging is also implemented in the scheduler, by iterating through all queues, and for all processes exceeding the age threshhold, which over here is 30, they are demoted from the given queue to a lower queue. This is again done using linked list and pointer structures. It is ensured that there are no dangling pointers or extra/unnecessary memory left allocated.

Possible Exploitation of MLFQ Policy by a Process If a process voluntarily relinquishes control of the CPU, it leaves the queuing network, and when the process becomes ready again after the I/O, it is inserted at the tail of the same queue, from which it is relinquished earlier. This can be exploited by a process, as just when the time-slice is about to expire, the process can voluntarily relinquish control of the CPU, and get inserted in the same queue again. If it ran as normal, then due to time-slice getting expired, it would have been preempted to a lower priority queue. The process, after exploitation, will remain in the higher priority queue, so that it can run again sooner that it should have.

Task 3

Procdump

Procdump is a function that is useful for debugging. It prints a list of processes to the console when a user types `Ctrl` `P` on the console. I have extend this function to print more information about all the active processes. Sample output is given below. rtime: Total ticks for which the process ran on the CPU till now. wtime corresponds to the total waiting time for the process. However, In the case of the MLFQ scheduler, this is the wait time in the current queue. nrun is the number of times the process was picked by the scheduler. q_i[MLFQ only] is the number of ticks the process has spent in each of the 5 queues.

For MLFQ:

PID	Priority	State	rtime	wtime	nrun	q0	q1	q2	q3	q4
1	2	sleeping	12	10	2	5	7	4	0	0
2	0	running	5	0	1	5	0	0	0	0
3	1	running	8	5	2	5	3	0	0	0
4	-1	zombie	7	8	3	1	2	4	8	0

For PBS:

PID	Priority	State	rtime	wtime	nrun
1	60	sleeping	12	10	2
2	23	running	5	0	1
3	21	running	8	5	2
4	19	zombie	7	8	3

Performance Analysis

Round Robin Scheduler

Average Waiting Time :   115 ticks
Average Running Time : 13 ticks

This performance of round robin scheduler can be expected for the given scheduler tests. As we are preempting many processes frequently so as to give equivalent timeslice to each and every process, our average waiting time comes out to the given value. The runtime is also expected as due to the high amount of swapping, the runtime is more than that of FCFS, which is non-preemptive, and PBS where there was observed to be low runtime.

First Come First Serve Scheduler

Average Waiting Time :   47 tick
Average Running Time : 38 ticks

The first come first serve scheduler apparently seems to have almost equal wait time and run time as the processes are mostly waiting in sleeping state, rather than runnable which is detected by the waitx call. As a result, the average wait time come.

Priority Based Scheduler

Average Waiting Time :   120 tick
Average Running Time : 6 ticks

Priority based scheduler has an equivalent output to that of RR scheduler. This can be concluded from the fact that despite there being preemption, by choosing based on priority, wait time is very high, and it switches less between processes. This can partially be inferred from the method of implementation, as the overhead required to ensure priority with preemption far outweighed the increased throughput of the scheduling algorithm.