Report

This document describes the implementation of the specifications as mentioned in the assignment.

Specification 1: syscall tracing

SETUP:
- Added $U/_strace to UPROGS in Makefile
- Added sys_trace() in kernel/sysproc.c that reads the syscall argument and calls the trace() defined in proc.c with the argument
- trace() sets the mask field of the process struct
- Modified fork() to copy mask from parent to child
- Modified syscall() to print syscall info if the syscall is traced
- Created user program strace in user/strace.c that calls the trace system call to set the mask of the process and run the function passed.

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.

Specification 2: scheduling

SETUP:
- Modified Makefile to take argument which then defines a macro with the compiler to identify the scheduling algorithm

FCFS Policy

SETUP:
- Edited struct proc to store the time it was created
- Edited allocproc() to initialise the new variable created above
- Edited scheduler() to run the process with the lowest time created
- Edited kerneltrap() in kernel/trap.c to disable premption with timer interrupts

IMPLEMENTATION:
First Iteration:
- Declare a minproc variable and initialize it to 0.
- Within the loop, identify the runnable process with the lowest creation time.
- After the first iteration, minproc points to the process with the least creation time.

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.

PBS Policy

SETUP:
- Edited struct proc to store the priority, time dispatched, runtime during allocated time, and time when it ready to run
- Edited allocproc() to initialise the new variables created above
- Edited scheduler() to run the process with the highest priority
- Edited clockintr() to track runtime and wait time
- Added a new sycall set_priority to change the priority of a process

IMPLEMENTATION:
- 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.

MLFQ Policy

SETUP:
- Edited struct proc to store the priority, allocated time, times dispatched, time added to queue, and time spent in each queue
- Edited allocproc() to initialise the new variables created above
- Created 5 queues of different priority
- Edited scheduler() to run the process with the highest priority
- Edited clockintr() to track runtime, add processes to queue and handle aging
- Edited kerneltrap() and usertrap() to yield when process has exhausted its time slice

IMPLEMENTATION:
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.

Answer to question in specification 2:
A malicious process can exploit the given condition by yielding the CPU before finishing its allocated time, retaining its priority and blocking lower priority processes from running unless aging is implemented or Suppose a very important process is to be run. Ideally, it should be placed in a high-priority queue. However, as each process has a certain time slice, which would be lesser in high-priority queues. After a certain time, it would be pushed to the lowest-priority queue and would not be given the priority it needs. A solution to this is to make it spend as much time as it can

in the highest-priority queue, and then just before its time slice gets over, make it relinquish the CPU and re-insert it in the same queue. In this manner, the process can exploit this feature to remain in the highest-priority queue despite the time slice.

Specification 3: procdump

SETUP:
- Edited procdump() in kernel/proc.c to print data from the process struct

IMPLEMENTATION:

Additional Implementations

SETUP & IMPLEMENTATION:
- waitx() syscall is implemented that functions similar to wait() but also returns the runtime and wait time of the child process
- We fork 10 new dummy process, and let it execute for a constant amount of time.
- For PBS we call the set_priority() function to check it.
- It simulates IO bound process by calling sleep() command.
- Waitx is called to calculate the average run and wait times
- It exists after printing average wait and run times

Scheduler	<rtime>	<wtime>
Round robin	125	29
First come first serve	79	66
Priority based scheduler	113	33

RR :
```
$ schedulertest
Process 6 finished
Process 7 finished
Process 8 finished
Process 9 finished
Process 5 finished
PPPrrooccersess s2 oc 0feisn ifsihneids
sh ePdr
oc1ePsrs  fo3ci ensiss hfeidn
i4s hfeidn
ished
Average rtime 125,  wtime 29
$
```

FCFS:
```
$ schedulertest
```

```
Process 0 finished
Process 2 finished
Process 1 finished
Process 3 finished
Process 5 finished
Process 4 finished
Process 6 finished
Process 8 finished
Process 7 finished
Process 9 finished
Average rtime 79,  wtime 66
$
```

PBS:

```
$ schedulertest
Process 6 finished
Process 7 finished
Process 5 finished
Process 9 finished
Process 8 finished
PPPrroorcoecsecsess ss 02  f1  ffiinniinsihsehdi
sehdedP
ro
cePsrso c3es fs i4n ifsihneids
hed
Average rtime 113,  wtime 33
$
```

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.

Round-robin performs the best, followed closely by MLFQ, and then PBS.

The above results are obtained by running schedulertest program in console