Assignment - 4

Report

Aditya Sehgal(2020112013)
Anish Mathur(2020102044)

This report summarizes our approach towards the various specifications implemented in the current xv-6 Operating System. In this assignment we have implemented the following specifications:

---

1. *Syscalls* : We implemented 2 system calls which involves the following steps:
   1. Modifying *syscall.c* to include the function pointer for the particular system call and adding the function prototype.
   2. Modifying *syscall.h* to include the indexing of the system call in the array of function pointers.
   3. Modifying *sysproc.c* to include the basic implementation of the system call.
   4. Modifying *usys.S* to include the user program which will be used to call the system call and act as an interface between the user and the system call.
   5. Modifying *user.h* to include the prototype of the user function for the system call.
   6. Modifying *makefile* by adding name of user program without the extension in the UPROGS section of the makefile.

   A) *Trace* :

   Create trace syscall as discussed above.

   In sysproc.c, the trace_flag variable is assigned the value of first register. $(p \rightarrow trapframe \rightarrow a0)$.

   In syscall.c we store the value of $a0$ in a temporary variable as the value of $a0$ changes to return value after execution of syscall.

   After syscall execution, if the condition of a0 value right shifted with syscall number is true then print the relevant information about that syscall.

   B) *Sigalarm and Sigreturn* : This system call takes in 2 arguments, a timer *interval* and a function *handler*. Once those many ticks (val) have occured, it saves the current state of the system using *kalloc* and *memmove*, executes the function *handler* by changing it's program counter to that of the handler function, which contains the system call *sigreturn*, that returns back to the original state.

We modify the *proc* struct to store the interval, number of ticks left, a pointer to the *trapframe* and a pointer to the handler function struct to save the current state and jump to the handler.

---

2. *Schedulers* : By default, xv-6 used the round robin scheme to schedule its processes. In our version, we implement 4 different standards of scheduling:
   1. **First-Come-First-Serve:** The FCFS algorithm is extremely intuitive and non-preemptive in nature. The processes are scheduled based on their time of arrival in the queue. In xv-6, we modified the *proc* struct in file *proc.h* to simply store the 'tick' at which the process was scheduled.

We then iterate through all processes in a single for loop, comparing the creation time of each process and storing the process with the least time in a *struct proc* p* variable. At the end of the for loop we execute said process.

   2. **Lottery Based Scheduler:** This scheduler is implemented as following:

   proc.h

   1. A variable tickets is initialized in the process state. Now, every process is assigned a default number of tickets which would dictate it's CPU time share.

proc.c
1. In scheduler function, a loop increments over each process' tickets and storing the total number of tickets in variable $max\_tickets$.
2. A random ticket is generated within the range $[0, max\_tickets]$ using random function.
3. A counter is then run through the process table to obtain the desired process for which $counter \geqslant rand\_num$.
4. Here, the counter is incremented with process' ticket number.

trap.c
5. To make it preemptive, a yield call similar to round robin is implemented in trap.c

Hence, this solves the problem of starvation as each process with at least one ticket is guaranteeed to execute.

3. **Priority based Scheduler:** In the PBS system, we assign an initial priority of 60 to each process. Although preemptive in some systems, our version is non-preemptive. We also have 2 different tie breakers. If the priority is the same, we check the number of times the process has alsready been scheduled. If that also does not resolve the tie, we check the creation time.

We first initialize the following variables in the proc struct,
1. *start_time_pbs* : *stores the creation time in case of the tie.*
2. *priority* : stores the priority of the process.
3. *niceness_var* : *stores the niceness of the process.*
4. *last_run_time* : stores the number of ticks the process was in RUNNING state.
5. *last_sleep_time* : stores the number of ticks the process was in SLEEPING state. The last two parameters are known to calculate the dynamic priority, which is stored in a temp variable.
In this version of the PBS, we iterate through all processes to find the minimum dynamic priority in a for loop, checking for the tie breaker conditions. At the end we simply execute that process.

4. **Multi-Level Feedback Queue:** This scheduler is implemented as following:
proc.h
1. Each process is initialized with four variables:
- priority: stores the priority level the process belongs to
- in_queue: stores binary value to tell if the process is in a queue or not.
- curr_rtime: stores current run time since the process last entered the CPU
- curr_wtime: stores current wait time since the process last became RUNNABLE

2. The multi-level priority queue is created through five structs containing:
- A rotating array of processes
- The front process index
- The last or 'back' procedd index
- The length of the current number of processes in the queue

proc.c
1. In scheduler function, a loop increments over process table and enqueues the first RUNNABLE process wrt its priority level. This loop ensures that whenever a process becomes RUNNABLE and is not present in the queue, it is added to it.
2. Furthermore, a second loop increments over our created multi-level queue to find the first RUNNABLE process. It dequeues it and sends it to execution.

trap.c

1. In usertrap and kerneltrap function, a process is yielded if:

- it's $curr\_rtime \geqslant assigned\ queue's\ time\ slice$: decrease it's priority and yield.
- if there's any process in the higher queues then the current queue: To quickly serve a higher priority process.

proc.c

3. Aging is continuously checked in update time function, such that if:

- $curr\_wtime \geqslant AGING\_TIME$: delete it from the queue and increase it's priority.
- $curr\_wtime$ is incremented in the same update time function when the process is RUNNABLE.
- This will be caught by the enqueue loop in the scheduler function as it is constantly running.

Exploitation of MLFQ Policy:

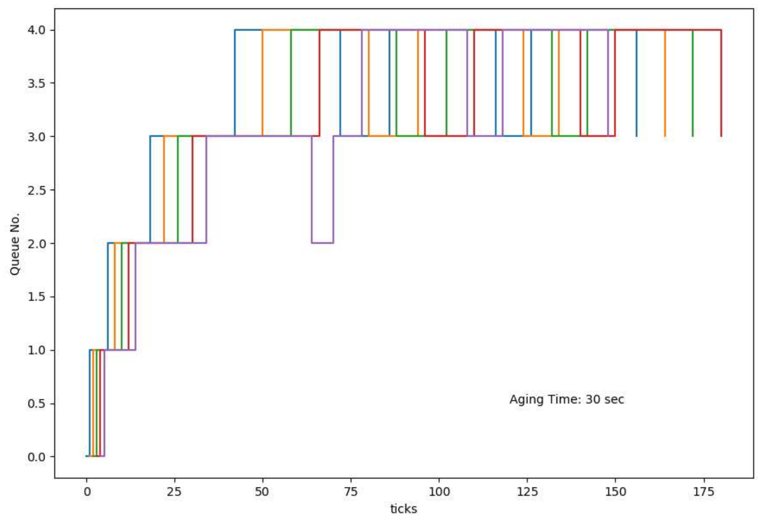
Since, we have defined a policy that if a process voluntarily relinquishes control of the CPU, it leaves and enter at the last of the same priority queue after I/O completion. This could be exploited by the process as it can deliberatly relinquishes itself just before the end of time slice. Hence, it will be enqueued again in the same queue instead of a one lower queue.

Run-time and Wait-time for different schedulers:

Schedulers	***Run time***	***Wait time***
Round-Robin	17	168
FCFS	18	185
LBS (tickets = 1)	18	168
PBS (priority = 60)	22	159
Multi-Level Feedback Queue	18	171

Here, all the schedulers are been run on a single CPU.

Timeline graph for MLFQ:



Here, we can observe that process yielding and it's priority increase is working perfectly. Furhtermore for aging time $= 30$ sec, one of the process increases it's priority from $4 \rightarrow 3$.
Hence, confirming that aging function is also working. (The code to create this graph is included with xv-6 files)