# Report

Divij 2021101001

Bhargav 2021101065

## Directions to run the code :

```
1. Run make clean to make sure that there is no garbage before actually compiling.
2. Run the following commands depending which scheduler you want to execute :
  a. make qemu                  - Run this for RR
  b. make qemu SCHED=FCFS       - Run this for FCFS
  c. make qemu SCHED=LBS        - Run this for LBS
  d. make qemu SCHED=PBS        - Run this for PBS
  e. make qemu SCHED=MLFQ       - Run this for MLFQ
```

## Specification 1 : System Calls

### System Call 1 : trace

1. 1. The basic flow of syscalls is, when user calls a syscall, it triggers an assembly function generated by usys.pl which save the args to registers and uses ecall to trigger an trap which calls uservec in trampoline.s which calls usertrap in trap.c after storing the current state in trapframe. From there syscall in syscall.c is called which has access to the array of syscall wrappers which retrieve args from registers using argraw and its wrappers, which call appropriate syscalls.
2. So add the strace syscall, and other corresponding things. The strace user program takes the bitmask and the command to be executed, forks and uses exec to execute the command as the child process.
3. A variable trace_mask is added to the proc structure and this is passed to each of its child upon further forks so all are traced. Whenever syscall in syscall.c is called and the process has a nonzero trace_mask then the syscall is printed along with the paramenters the number of which is calculated using a global variable.

4. The index of the set bits of trace_mask represent the sys_calls that will be traced.
5. The parent in the strace user program calls the trace syscall which sets the trace_mask for a given pid. It is present in proc.c.

## System Call 2 : sigalarm and sigreturn

- Add an extra trapframe to the struct proc in **proc.h.** Also, add interval, time_passed, in_handler and fnptr as int and last one as uint64 in the struct proc.
- Initialize interval as 0, time_passed to 0, in_handler to 0 in allocproc().
- Add sigalarm syscall by adding it to syscall.c and syscall.h
- In sysproc.c, add the code which retrieves the two parameters and calls sigalarm from proc.c
- This sets the two fields of process to interval and fnptr.
- Add a check_alarm function which basically checks if the current process's interval is more than 0 and if it the function is already in handler (via in-handler to make sure it is not called again while the handler is already executing). If conditions are met, save the trapframe of the function by copying it in the extra trap_frame created before and call the handler by setting the epc as the fnptr of the handler. When the trap exits, it goes to the handler to start executing that now.
- The handler calls sigreturn() which is implemented as a syscall similarly to sigalarm. It restores the trap frame from the extra frame and calls usertrapret() to return which basically fixes all the other registers like a0.
- Create two user functions which processes can call to do the sigalarm and sigreturn routine.
- Update the time_passed every time tick and compare it with the interval to call the check_alarm function in which_dev ==2 conditional block of usertrap and kerneltrap both.

# Specification 2 : Scheduling

## 1. FCFS

- **proc.h and proc.c**
  Add “creation_time” to struct proc in proc.h which stores the amount of ticks that have passed since the beginning till when the process was created by calling sys_uptime() in allocproc();

- **proc.c**
  Declare a struct proc pointer “this_proc” and initialize it to 0. This stores the process with the minimum creation time. Loop through all the processes in the scheduler, acquire locks at the right places and do the following if the process’s state is found RUNNABLE :
  1. If this_proc = 0, then assign that process’s address to this_proc i.e. this_proc = p
  2. Otherwise, compare the creation time of the current process and this_proc, if current process has lower creation_time, then update this_proc;

- **proc.c**
  After looping through all the processes in the previous bullet, check if this_proc is still 0 (as no process might be RUNNABLE at a given time and thus it stays 0). If it is not, acquire it’s lock, then check if it is RUNNABLE even now as it might have changed it’s state because of multiprocessing.
  If it is still RUNNABLE, then switch context to the process after taking the appropriate steps.

- **trap.c**
  In trap.c, put the yield() call in usertrap() and kerneltrap() functions in a condition such that it is not executed when FCFS is used as the scheduler, #ifdef can be used for this. This makes it so that the process doesn’t yield the CPU even when there is a timer interrupt.

## 2. LBS

- **proc.h and proc.c**
  Add “tickets” to struct proc in proc.h to hold the number of tickets a process currently has. In proc.c, set it to 1 by default in allocproc() function call. In the fork() function, set it to the number of tickets the parent has for the process that’s getting forked (child process).

- **proc.c**
  Declare a int “sum” and initialize it to 0. This stores the process with the minimum creation time. Loop through all the processes in the scheduler, acquire

and release locks at the right places and add it's p→tickets to sum if a process is found RUNNABLE.

- **proc.c**
  Declare a new variable int new_sum = 0 and struct proc pointer this_proc = 0 right after the previous loop ends. Generate a random number using freeBSD code given in xv6 in grind.c and take it's modulo with the sum variable from previous bullet.
  Loop over all the processes now and similarly, check if they are RUNNABLE. If yes, add their tickets to new_sum.
  Logic : we generated a random number between [0, sum-1] by taking modulo previously. Now the idea is that if the number is in the first x tickets (where x is the number of tickets held by first runnable process in the process table), then we run that process. Otherwise, we check for the next process.
  When a process is selected, choose it and assign it's address to this_proc. For similar reasons as before, check the zero condition for this_proc. If not zero, check if still RUNNABLE and if yes, run the process.

- **syscall.h, syscall.c, sysproc.c, proc.c and some header files**
  Add the settickets systemcall to syscall.c. sysproc.c just retrieves the value of the argument and passes it to settickets function in proc.c which sets it to that many number of tickets.
  Add these functions appropriately to the headerfiles.
  Create a function in user.h which calls the systemcall settickets. For this, add it to usys.pl and other required header files.

## 3. PBS :

- Add priority, scheduled, sleeping_time and running_time variable to struct proc in **proc.h**
- Assign the value 0 to sleeping_time, scheduled and running_time in allocproc() and assign 60 to priority in the same in **proc.c**
- Write a function which updates the sleeping_time and running_time on every clock interrupt and call this funciton in clockintr() in **trap.c**
- In **proc.c,** declare struct max_pr which is the same this_proc() inside the scheduler function, loop over all the processes and choose the one with the minimum dynamic priority which is calculated using two functions (that you implement) max and min. In case of tierbreakers, use the given rules in assignment to choose a process (using p→scheduled and p→screation_time).

Save this process's address in max_pr.
Do the usual 0 and RUNNABLE check on this, and otherwise update it's running_time and sleeping_time to 0 and increment it's p→scheduled by 1 as it's scheduled 1 more time.

- Implement user_process setpriority which takes a pid and a priority number and then calls a system call which sets the priority of the process with the given pid to that. For this, add the required syscall.c etc and in the function setpriority of proc.c, loop over all the process looking for the process with the given pid. If found, set it's priority to the given number and set sleeping_time and running_time to 0. If the old value of priority is more than the new, the reschedule the CPU.
- Disable pre-emption by making a change similar to that in **trap.c** and removing the call to yield()

## 4. MLFQ :

- Add curr_ticks, queue_number and last_scheduled variable to struct proc in **proc.h**
  curr_ticks stores how many ticks the process has consumed in the same queue, last_scheduled store sys_uptime() of when it was last scheduled.
- In **proc.c,** assign queue_number and curr_ticks as 0 and last_scheduled as sys_uptime() in allocproc()
- Write enqueue and deque functions to do their jobs and create 5 global queue and 5 variables to store each of their sizes (as an array) in **proc.c**
- Enqueue init to queue 0 in userinit, enqueue any new forked process to queue_0 as well in fork(). Dequeue a process in wait() as then it should not be present in the queues (as not runnable anymore)
- Write helper functions check_exceed and check_higher which are called every time there is a timer interrupt. check_exceed checks if allowed ticks have exceeded and if yes, yields for scheduling and also increases the queue_number by 1 if not already 4. It also sets curr_ticks back to 0. Check_higher checks if there is a process with higher priority (in one of the higher queues) and yields for scheduling if one is found.
- In the scheduler function in **proc.c,** first push all the processes to their respective queue numbers. Enqueue is written such that, it pushes a process in a queue only if it's not already there.

After this, iterate over all process in queues 1 to 4 checking for aging. Calculate age as sys_uptime() - last_scheduled and compare it with the MAX_AGE which is #defined in **proc.h**
Update the queue_number of the process accordingly and dequeue it from the old queue and push to the new one.

- After doing the previous work, start from queue0 and choose the front process of the first queue that is non-empty. Dequeue this process from it's queue, do the usual checks on it before running it. Update the last_scheduled to the sys_uptime() before switching context.

**Note :** Since it was not mentioned if a process should have it's curr_ticks set to 0 or not after relinquishing the CPU voluntarily, we are assuming that it was not supposed to be made 0. More is explained about this in the answer below.

### Answer to Q : How MLFQ policy can be exploited by a process?

In our implementation, we run the process which comes back to the same queue for the leftover amount of ticks (i.e. if a process relinquishes CPU voluntarily in favor of some interrupt and it still hasn't reached the limit of the number of ticks it can have in that particular queue, then when it is inserted at the back of the queue, it will run only for the amount of ticks it had left.)

This prevents the exploitation where processes can keep relinquishing the CPU intentionally to make sure they stay in the top queues. However, **an alternative way to exploit this would be** to fork more processes to execute your code in parts. As we put any new created process in the top queue, this would mean that the process will have more number of ticks to execute in the top queue as compared to a normal process which does not do this (essentially user-level threads).

## Specification 4 : Copy-on-write fork

### COW Fork

1. When `fork` is called, in the usual case it creates a copy of all the pages referenced by the pagetable of the calling process. But when `COW` is enabled, only a new pagetable is created and it still points to the same pages as the parent.

2. An array `page_refs` is used to keep track of the references to each page. `page_refs_lock` is used as a lock for this array. An array of locks would be more efficient time wise as it won't block access to someother page.

3. When `fork` is called all the pages are `stripped off` their `write` bit.
4. When a process tries to refer to an address in such page it creates a page fault with `scause` = `0xf` (Store Page Fault), upon which `cow_handler` function springs into action and duplicates the page using `kalloc` and `memmove` and gives write permission to the new page. Error cases will kill the process using `setkilled`
5. The references to the original page are decreased by calling `kfree`.
6. The `kfree` function decreases reference count and actually frees only if the references are zero.
7. When `kalloc` is called the page returned has its ref count set to 1, as only pages with zero refs enter the free list.
8. When xv6 first starts with the main function, `kinit` is called and in `kinit` all the pages are freed so that they get added to the page table so I set their pagerefs to 1 before they are called so that the pagerefs stay at 0 after `kinit`.

## Scheduling Analysis :

For this, **schedulertest** was used with a few modification to run it properly.

The following analysis in comparison was collected, we collected 2 instances of data for each scheduler which are in the following table. **Note that all policies except MLFQ are for 3 CPUs hence it is evident then MLFQ works better as it performs very well even in case of only 1 CPU.**

| Scheduling Policy | rtime1 | wtime1 | rtime2 | wtime2 |
|---|---|---|---|---|
| RR | 40 | 96 | 45 | 97 |
| FCFS | 55 | 83 | 53 | 84 |
| LBS | 28 | 114 | 30 | 112 |
| PBS | 51 | 84 | 54 | 83 |
| MLFQ | 44 | 165 | 40 | 158 |

By this analysis, we can see that MLFQ works the best while the general order is something like this:

**MLFQ > PBS > FCFS > RR > LBS**

This is mostly because while RR does not differentiate between the processes, LBS makes it such that it pays attention to number of tickets which might take in a different direction compared to being optimal. Hence, there will be a lot of inefficient scheduling in LBS depending on which process has more tickets (which again, has nothing to do with optimality).

FCFS works just slightly better than RR as it also pays attention to a parameter creation_time which does not have much of something to do with optimal scheduling most of the time. Depending on the processes being ran, it might be worse than RR as well.

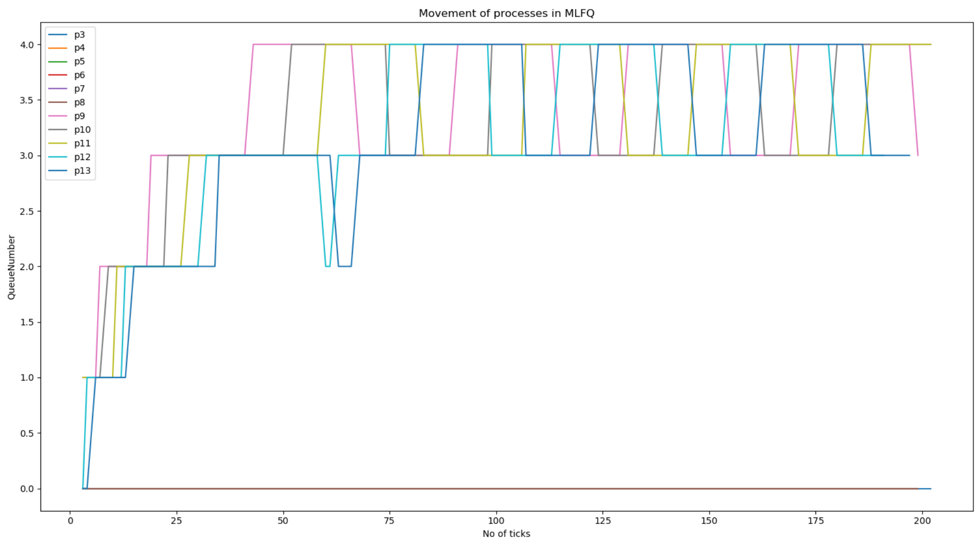
PBS is better than these as it calculates the priority of processes as they are being run. It calculates the niceness and uses that as a parameter along with other other things to schedule it. This gives us more optimal scheduling than the others as we keep the running_time and the sleeping_time of the processes in mind here while scheduling them.

MLFQ works better than all of these as it puts a strict condition on how many ticks a process can run for. It makes sure that I/O bound processes are passed first before a process that completely hogs CPU time is run. This ends up in making our scheduling algorithm more optimal as we now have 5 queues with each putting a constraint on how many ticks a process can run for. Depending on factors like consuming the ticks, the processes can move down the queue all the way to queue4 which is essentially RR in itself. This also makes sure that a process with higher priority takes precedence over one with lower priority and via aging, we make sure that starvation doesn't take place as we upgrade the queue of a process as it ages.

Hence, this way we can explain the trend in the data we have collected.

## MLFQ Analysis

Following is the graph obtained by doing the MLFQ  analysis, which we ran with the MAX_AGE parameter defined as 30 ticks for aging a process.


Movement of processes in MLFQ
QueueNumber
No of ticks
p3
p4
p5
p6
p7
p8
p9
p10
p11
p12
p13
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
25
50
75
100
125
150
175
200