OSN - Assignment 4 REPORT

Anushka Jain - 2021111008

Nandini Maroo - 2021111012

Specification 1 : System Calls

System Call 1 : `trace`

- Created a user program “strace.c”, `strace` takes in arguments mask and the command to be executed then calls the system call `trace` for that mask and executes the command.
- Added the system call definition in necessary places. Modified the *struct proc*, adding a new variable *trace_mask*.
- Implemented system call “trace” in kernel/sysproc.c. It defines the value of *trace_mask* in the current process as the mask that was input.
- In syscall.c we modify the function `syscall()` such that everytime a system call in called in the execution of the command, we get the bit for the system call being executed if that bit in the mask is 1 then we print the pid, system call name, the decimal values of the arguments and the return value of the system call.

System Call 2 : `sigalarm` and `sigreturn`

- sigalarm system call is already added in program “alarmtest.c” provided to us. The “alarmtest.c” file is added to directory “user”.
- `sigalarm(interval, handler)` specifies that after every *interval* amount of ticks of CPU time, the *handler* is called again.
- Implemented system calls `sys_sigalarm()` and `sys_sigreturn()` in kernel/sysproc.c. `sigreturn()` is called when alarm goes off.
- Added system call definitions in required files.
- Alarm information (int ticks, int cur_ticks, trapframe *alarm_tf, int alarm_on) is added to `struct proc` definition in kernel/proc.h to retain the information in the

process state. The given variables are also initialised in `allocproc()` in kernel/proc.c.

- Now, the trap handler is implemented in `usertrap()` in kernel/trap.c. We ensure that when curr_ticks reach the required ticks value, the handler is called again as control returns to the user space.

Specification 2 : Scheduling

In the `makefile`, we add a macro SCHEDULER which assigns the scheduler algorithm required. The flags for compilation:

● First Come First Serve = `FCFS`
● Priority Based = `PBS`
● Multilevel Feedback Queue = `MLFQ`

The default flag would be Round Robin = `RR`.

a. FCFS (First Come First Serve)

- We edit the struct proc to include a new variable *ctime* which is the time of creation. In `allocproc()` we make the *ctime* of the proc equal to the ticks.
- In `scheduler()` in kernel/proc.c we add the code that is to be executed if SCHEDULER=FCFS.
- There we run a for loop for all the RUNNABLE processes to find the one which has the least creation time i.e. the process that was created first then we switch to that process.
- We also disable the preemtion of the process after clock intervals in trap.c.

b. LBS (Lottery Based Scheduling)

- We make a system call `set_tickets()` which is used to assign tickets to the processes.
- In `scheduler()` , we calculate the total no. of tickets and obtain the ticket probability of the process. Then we generate a random number. (random number generator from a resource on the internet) If our probablility is more than the random number then the process is deemed lucky and we switch to it.

c. PBS (Priority Based Scheduling)

- `waitx()` syscall is implemented by making file user/time.c.
- System call `set_priority()` is added to a file user/setpriotity.c. schedulertest.c uses this function to call PBS.
- Definitions for `set_priority()` are added in required files and implementation is done in kernel/sysproc.c.
- Variables specifying sleep time, nap time (time elapsed since the last sleep), niceness value, static priority, etc. are added to the definition of `struct proc` in proc.h and initialised in `allocproc()` in proc.c.
- The functionality for PBS is added to function `scheduler()` in kernel/proc.c. We go through the list of processes and whenever there's a new process that has not been assigned a priority, we find out it's priority and accordingly place it in our priority queue.

d. MLFQ (Multi Level Feedback Queue)

- Added the required variables to struct proc and created a new struct queue in kernel/proc.h
- NMLFQ and AGING is defined in param.h
- In kernel/proc.c, defined `mlfqueue` which is an array of 5 queues, added functions to push, pop, remove etc. from queue, initialised the values of the queues in `procint()`, initialised in `allocproc()`, in `update_time()` increasing the runtime of each process in that queue and decreasing the set `time_slice` after which it changes the queue to the one with lesser priority.
- In kernel/proc.c, in `scheduler()` we add the code to be executed for SCHEDULER=MLFQ.
- This code loops through all the processes, if it has aged then we dequeue it and increase increase priority and make the in_queue flag 0 so that it gets added to the new prority queue.
- Another for loop is used to loop through all the processes and if they are not in a queue add them to a queue w.r.t. their priorities.
- Now we loop through all the levels of the mlfqueue, if there is a process that has just been added to the queue (preferably of higher priority) [note here 0 is higher

than 1] then that is chosen as the `next_proc`. We switch to the `next_proc`.

- In kernel/trap.c, we handle preemption by pushing the process (that's giving up control) back into the same priority queue.
- If a processes completely uses up the time slice that is allotted for its execution in a certain queue, it is moved to the end of queue with next lower priority for completion of execution. However, if the process voluntarily relinquishes control of the CPU, it is pushed to the end of the same queue. Therefore, we can make use of this if we voluntarily make a process leave the queuing network if its time slice is about to end. As a result, it will stay in the higher priority queue instead of being pushed to the queue with next lower priority.

> 💡 If the process uses the complete time slice assigned for its current priority queue, it
> is preempted and inserted at the end of the next lower level queue.
>
> If a process voluntarily relinquishes control of the CPU(e.g. For doing I/O), 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.
> **How can a process exploit these facts?**
>
> A process have an I/O scheduled just before the last tick after which it uses up its time slice. So if this process keeps on having an I/O like this, every instance it is supposed to go to a lower priority, instead it just stays at the same level just gets added to the tail. A process can stay in the same level until its completed even if it has a huge running time, by having I/Os scheduled like this.

Analysis of Different Scheduling Algorithms

The following table contains the average run times and wait times for various scheduling algorithms.

ALGORITHM	rtime	wtime
RR	152	13
FCFS	119	26
LBS	123	14
PBS	126	13
MLFQ	141	13

Specification 3: Copy-on-Write fork

- In vm.c, we introduce a new function `walk()` which returns of the PTE in page table that corresponds to virtual address of va and if it's not allocated then a page table is created.
- Function `incref()` is added to kalloc.c to increment the reference count of a certain page.
- In trap.c, we have defined `cowfault()` to look for interrupts and respond accordingly.
- Other functions like `copyout()` , `uvmcopy()` are also updated in vm.c so that they respond to `cowfault()` and other changes in flags.