OSN Assignment 4 - xv6(RISC-V)

Abhinav Reddy Boddu

2021101034

Gnana Prakash Punnavajhala

2021111027

Adding a new system call:

- We set the system call number of the new system call according to the availability, with the maximum being 32 in the `kernel/syscall.h` file. The naming of the system call must be sys_`function_name` and must be followed throughout this process.
- In the `kernel/syscall.c` file, we add the function prototype by adding the line `extern uint64 sys_function_name(void);`. We map the system call number of the function defined in `kernel/syscall.h` to the prototype of it defined previously. This is done in the `syscalls` array by adding the appropriate entry as `[syscall_number] sys_function_name`. We similarly add entries to the `syscall_names` array which keeps track of the names of the system calls and its entry is added as `[syscall_number] "function_name"` and finally another entry is added to the `syscalls_argcnt` array, which keeps track of the number of arguments of each system call. An entry for this array is created as `[syscall_number] number_of_arguments`, where `number_of_arguments` is a non-negative integer.
- To make these system calls available to user-programs, we have to create entries of these system calls in `user/usys.pl` as `entry("function_name");` and in `user/user.h` as the actual prototype of the function including return type and argument type as `return_type function_name(argument types);`

Adding a new user program:

- After creating `filename.c` file, we include the object file dependency as `$U/_filename` in the `UPROGS` section of the Makefile.

Specification 1: System calls

`trace`(system call) and `strace`(user program):

- The `trace` system call is assigned system call number of `22` in the `kernel/syscall.h` file.
- Add an integer named `syscall_tracebits` to the `proc` struct in `kernel/proc.h` file to keep track of the system call to be printed by getting the number of the system call that is called which can be obtained by getting the value in the register `a7`.
- Set the number of arguments required for the `trace` system call to be `1`, the bit of the system call to be traced and add the entry to the `syscall_argcnt` array of the `kernel/syscall.c` file.

- The implementation the `trace` system call in the `kernel/sysproc.c` file sets the `syscall_tracebits` variable using the `argint` function. Then, in the `syscall` function in the `kernel/syscall.c` file, we check for the appropriate system call to be traced by using the condition `if((p->syscall_tracebits) & (1 << num))`, which evaluates to `true` only if `syscall_tracebits` variable is set (`strace` function is called) and the expression evaluates to a valid system call number. Else, it is printed that there is an unknown system call. If the expression evaluates to a valid system call, then the pid is printed through `p->pid`. The system call name is accessed through the syscall number which is stored in the `a7` register of the trapframe of the process. The number of arguments of the system call are known and so, the decimal value of the arguments is obtained through the `argint` function by passing the appropriate index of the argument.
- The `strace` function in the `user/strace.c` file calls the `trace` system call and the function executes as intended.

`sigalarm` and `sigreturn`:

- The `sigalarm` and `sigreturn` system calls are assigned system call numbers of `23` and `24` in the `kernel/syscall.h` file.
- The number of arguments for `sigalarm` and `sigreturn` has been set to `2` and `0` respectively.
- A struct named `sigalarm_struct` has been created which has been defined in the `kernel/proc.h` file and consists of `currticks`, an int variable which stores the number of ticks the process has been running for upto that point of time, `nticks` and `handlerfn` are the arguments provided when a function calls `sigalarm(n, fn)`, where `nticks = n` and `handlerfn = fn`, where `nticks` is an int variable and `handlerfn` points to the function provided as an argument and `trapframe_cpy`, which keeps a copy of the trapframe of the present process.
- A variable named `alarmdata` of the type `struct sigalarm_struct` has been added to the `proc` struct in the `kernel/proc.h` file.
- The `sigalarm` system call first sets the value of the `nticks` variable based on the function call to `sigalarm` as `sigalarm(n, fn)` using `argint` function and the `handlerfn` variable is set to `fn` using the `argaddr` function. For every `nticks` number of ticks, we copy the present state of the process trapframe into `trapframe_cpy`. We check if this is `0 (NULL)` or not, so as to check if the copy exists or not. This ensures that there is no reentrant code.
- The `sigreturn` system call restores the value of the trapframe of the process before calling the `sigalarm` function so as to enable the process to start where it left off before calling `sigalarm`. The values of `currticks` and `trapframe_cpy` are set to `0` before returning from the `sigreturn` function.

Specification 2: Scheduling

- First, we define a macro `NON_PREEMPT`, which we use in the `usertrap` function in `kernel/trap.c` file to decide if we want processes to be preempted. If this macro is not defined, then we enable preempting by the CPU.
- We also added the variables `ctime, rtime and etime` to the `proc` struct in the `kernel/proc.h` file, which denote the creation time of the process, number of ticks for which it executed and time when it exited.

- `ctime` is set to the variable `ticks` in `allocproc` function and `rtime` and `etime` are set to `0`. `rtime` is incremented by `1` in the `update_time` function if the process is in `RUNNING` state. `etime` is set to the variable `ticks` in the `exit` function in the `kernel/proc.c` file.
- For, `FCFS`, `PBS` and `MLFQ` scheduling algorithms, we define the `NON_PREEMPT` macro as a compiler flag in `CFLAGS` using `$SCHED_FLAGS`. Otherwise, only the scheduling algorithm is defined (in the case of `LBS` scheduling algorithm). The `NON_PREEMPT` macro is defined in the case of `FCFS` and `PBS` as they are non-preemptive and it is defined in the case of `MLFQ` as the preemption method in the case of `MLFQ` is different.

First Come First Serve (FCFS) Scheduling:

- We iterate over all the `RUNNABLE` processes in the `proc` array and select the process which has the least creation time (checked using the `ctime` variable in the `proc` struct) stored in the variable `next_process` of type `struct proc *`, initially set to `0`.
- If a valid process has been selected, i.e., `next_process` is non-zero, we allocate CPU to that process pointed to by `next_process` by using the `swtch` function to perform context switch.
- This implementation naturally guarantees that the process with the least creation time is allocated CPU time first and thus the principle of `FCFS` is followed.

Lottery Based Scheduler (LBS):

- In the case of `LBS` scheduling algorithm, we add another int variable named `tickets` to the `proc` struct in the `kernel/proc.h` file to keep track of the number of tickets the process possesses.
- The value of `tickets` is set to `1` by default in the `allocproc` function in the `kernel/proc.c` file. The number of tickets a process has can be modified by the `settickets` system call.
- The `settickets` system call is assigned the system call number of `26` in the `kernel/syscall.h` file.
- The `settickets` system call sets the tickets variable of the process to the argument provided to it, where settickets is called as `int settickets(int tickets)`, where `p->tickets` is set to `tickets` using the `argint` function.
- In the scheduling algorithm, we choose the process randomly but in a biased manner with more bias towards processes that have higher number of tickets. For this, we use the `rand` function in the `user/grind.c` file.
- To choose the process, we use the concept of Stochastic simulation. We first calculate the total number of tickets across all `RUNNABLE` processes and store it in the `total_tickets` variable of type int, initially set to `0`. Then, we randomly generate a number between `0` and `total_tickets - 1` and store it in the int variable `randominteger` and then, we select the process which has cumulative tickets greater than or equal to `randominteger`. The cumulative tickets are stored in the int variable `preftickets`. The locks of the processes with cumulative tickets less than `randominteger` and those process which are not the very first process having cumulative tickets greater than or equal to `randominteger` are released. The selcted process is allotted CPU time accordingly.
- To ensure that every child process receives the same number of tickets as its parent, the `fork` function in `kernel/proc.c` is modified such that the tickets of the child are set to the same value as that of its parent if the `LBS` scheduling algorithm is active.

Priority Based Scheduler (PBS):

- When the `PBS` scheduling algorithm is active, we add the fields `static_priority`, `ntimesscheduled`, `nsleeping` and `nrunning` to the `proc` struct of the `kernel/proc.h` file, each of which represent the static priority of the process, the number of times the process has been scheduled, the number of ticks for which the process has been in `SLEEPING` state and the number of ticks for which the process has been in `RUNNING` state, which are initially set to `60`, `0`, `0` and `0` in the `allocproc` function of the `kernel/proc.c` file
- Every time the `update_time` function is called, `nsleeping` and `nrunning` are incremented if the process is in `SLEEPING` or `RUNNING` state respectively.
- The `calc_niceness` function in the `kernel/proc.c` file calculates which returns the niceness of the process provided to it as a paramter and is calculated according to the given formula.
- Then, we iterate over all the `RUNNABLE` processes and select the process with the maximum dynamic priority. The selected process is stored in the variable `selected` of type `struct proc *` and the corresponding maximum dynamic priority is stored in the variable `selected_DP` of type `int`. The dynamic priority of each process is stored in the variable `DP` of type `int`. Dynamic priority is also calculated according to the given formula and ties in selecting the process with the highest priority are followed accordingly. In the case where tie-breaking is done on the basis of creation times, the process which is created recently is selected.
- The selected process is then allocated CPU time accordingly and thus the process with the highest priority is selected.
- The `set_priority` system call is assigned the system call number of `27` in the `kernel/syscall.h` file.
- If the system call is called as followed `int set_priority(int new_priority, int pid)`, then `-1` is returned in the case of invalid value of `pid` or invalid value of `new_priority`, i.e., `< 0 or > 100`. Else, after finding the process in the proc with its pid equal to `pid`, we set its `static_priority` to `new_priority`. Since we have to reschedule once the priority of a process chamges, we set the values of `nrunning` and `nsleeping` of that process whose priority has been changed to `0`.

Multi Level Feedback Queue (MLFQ) Scheduler:

- A queue data structure has been created of type `struct Queue` in `kernel/proc.h` having fields `front` and `back` of type `struct proc *` and `no_of_processes` of type `int`, which represent the processes in the front and back of the queue and the number of processes in the queue respectively.
- The queues are `NQUEUES(5)` in number defined in `kernel/proc.c` and are initialised using the `queue_init` function in `kernel/main.c` file implemented in `kernel/proc.c` by setting all of the fields of the Queue struct to `0`.
- In the `proc` struct in `kernel/proc.h`, the following fields are added: `isQueued` of type `int` to keep track if a process is present in any queue or not, `Queue_Num` of type `int` to keep track of the queue in which the process is present, `int` array `slices_used[NQUEUES]` to keep track of the number of time slices the process spent in each of the `NQUEUE` number of queues, `ctime_queue` of type `int` to keep track of the time at which it has been inserted into the queue which it presently is in, `wtime_queue` of type `int` to keep track of the number of ticks for which it has been waiting in the queue which it presently is in and `queue_next` and `queue_prev` of type `struct proc *` to keep track of the processes which are after and before it in the present queue respectively.
- The queue function prototypes of `add_to_queue` and `remove_from_queue` are defined in `kernel/proc.h` file and implemented in `kernel/queue.c` file. Accordingly, the object file dependency for the `kernel/queue.c` file has been added to the Makefile as `$K/queue.o`

- Now, all the processes in the `proc` array of `kernel/proc.c` file are iterated and those processes which are not in any queue and are `RUNNABLE` are added to the queue according to their `Queue_Num` value using the `add_to_queue` function.
- Then, we iterate over all the queues starting from the `0` queue till we find a non-empty queue and select the queue and then allot it CPU time.
- With every clock interrupt, we call the `aging` function implemented in `kernel/proc.c` which puts the process in the queue with priority `p->Queue_Num - 1` (only for processes not in the `0`th queue). Also, we downgrade the present process if it has used up all of its time slice in the present queue by placing it in the queue with priority `p->Queue_Num + 1` (only for processes not in the `NQUEUE`th queue). After that, we iterate over all the queues with priority higher than that of the present process and if there are processes in them, we preempt the presently running process and reschedule again.

Specification 3: Copy-on-write (COW) fork

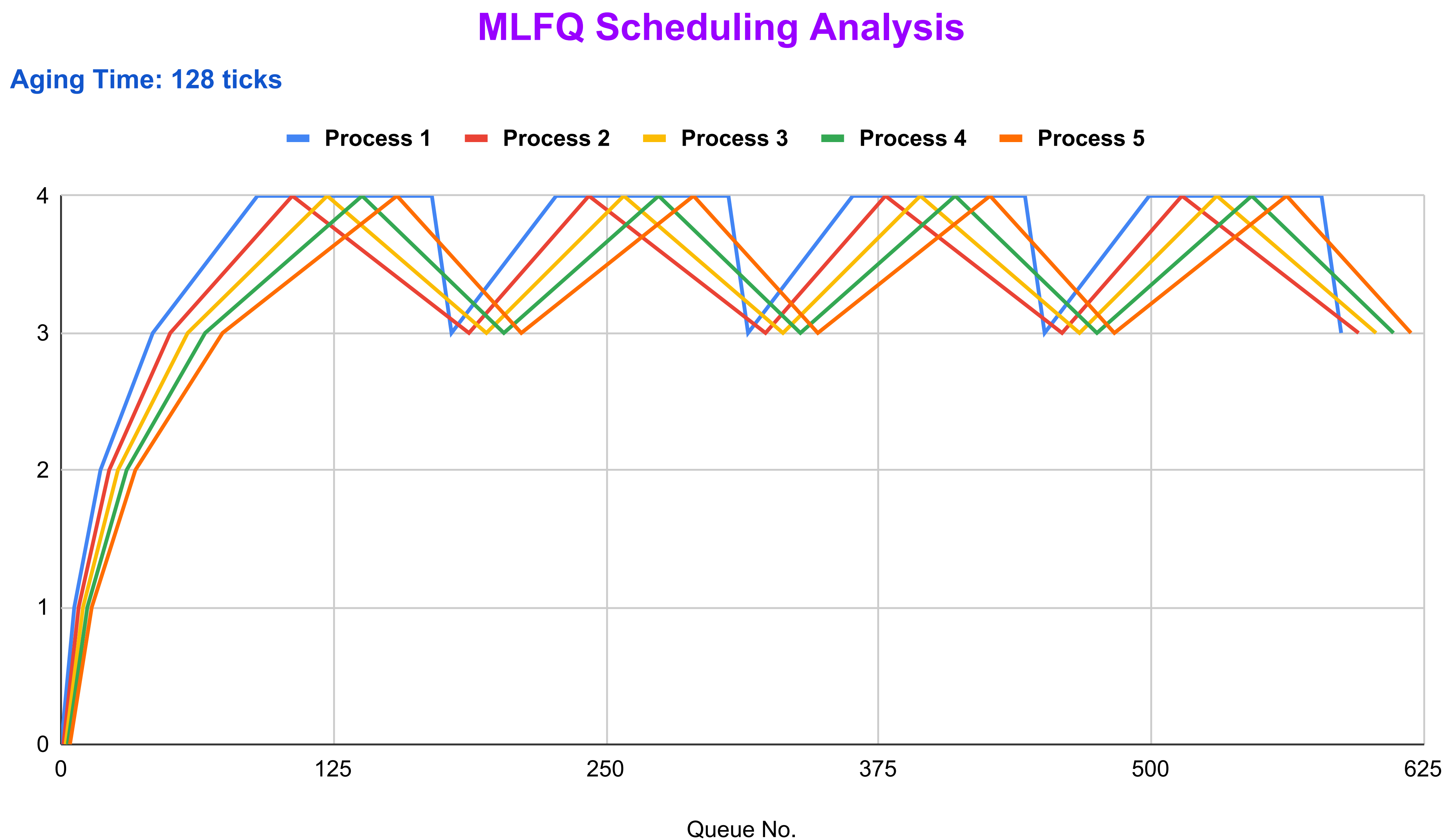
- In the `uvmcopy` function, we point the address space of the child to the same address space provided to the parent. Then, we update the permissions of the page table entries pointed to by the physical address by removing write permissions and adding an extra `PTE_COW` bit to help identify that the process is allocated address based on the principle of Copy-on-write fork. This bit is used later to identify if page-fault exceptions have occured due to COW. If yes, then these addresses are changed and different addresses are allocated to child and parent processes with write permissions.
- After every clock tick the value of `r_scause` (supervisor trap cause) is checked if it is a page-fault exception. If yes, then we have values of either `13` or `15` of `r_scause` after which we call the `COW_handler` function.
- In the `COW_handler` function, we first check if the page-fault exception is due to accessing of memory not allocated to write to, i.e., out of bounds memory access, in which case an error is returned and the process is killed using the `setkilled` function. Else, we obtain the address of the pagetable entry and the corresponding physical address allocated. We then update the permissions to access the address space of the parent by removing the `PTE_COW` bit and adding the `PTE_W` bit. Here, we essentially use the `PTE_COW` bit to identify page-fault exceptions that have appeared due to COW and resolve them by allocating physical address space to the child process and update the permissions of both the child and parent processes to be able to write to that address space. After this, the process executes normally.

Performance Comparision:

- The comparision between the implemented scheduling algorithms, viz, `FCFS`, `LBS`, `PBS`, `MLFQ` and the default scheduling algorithm, `RR` was done using the `schedulertest.c` user program.
- As given in the assignment document, `MLFQ` and `LBS` scheduling algorithms were run on `1` CPU while others (`PBS`, `FCFS` and `RR`) were run on `3` CPUS.

Policy	Average run time	Average wait time
RR	49	39
FCFS	28	22

Policy	Average run time	Average wait time
LBS	52	157
PBS	55	21
MLFQ	54	186


MLFQ Scheduling Analysis
Aging Time: 128 ticks
Process 1
Process 2
Process 3
Process 4
Process 5
4
3
2
1
0
0
125
250
375
500
625
Queue No.