Mini Project 2 : Introduction to XV6 and Networking

XV6:

System Calls

The implementation of two system calls in xv6:

getSysCount, which allows tracking the usage of specific system calls by a process

sigalarm, which introduces a mechanism for periodically interrupting a process based on CPU time consumption.

1. Gotta Count 'em All (`getSysCount` System Call)

Problem Description:

The objective is to implement a system call getSysCount that tracks how many times a specific system call is invoked by a process and its children, based on a mask provided by the user. This is particularly useful for analyzing the frequency of system calls and understanding the behavior of processes.

Design and Implementation:

To achieve this, we need to make several modifications in both the kernel and user space.

Kernel Changes:

1.Process Structure (`proc.h`): We add a field to track the number of times each system call is invoked by a process:

```
int syscallCount[NSYSCALLS]; // Array to store syscall counts
```

2.System Call Implementation (`sysproc.c`): The `getSysCount` system call will sum the number of times a particular syscall is invoked by both the current process and its children.

```
uint64 sys_getSysCount(void) {}
```

3.**Syscall Count** : Each time a system call is invoked, we update the count. This is done by modifying `syscall()` in `syscall.c`:

```
p->syscallCount[num] ++;
```

4.To add count by children we need to assign parents count to child and continue(proc.c in wait function):

```
for (int i = 0; i < NSYSCALLS; i++) {  p->syscallCount[i] = pp->syscallCount[i];}
```

5.User Program (`syscount.c`) :

The user program `syscount` takes a mask and a command as arguments. It then runs the command and waits for it to complete. After the command exits, it prints the total count of the selected syscall, including calls made by child processes:

→ I made sure that MAKEFILE is updated when a new user file is created.

2. Wake Me Up When My Timer Ends (`sigalarm` and `sigreturn` System Calls)

Problem Description:

The goal of this task is to add a new system call `sigalarm` that allows a process to register a handler function, which will be called at regular intervals (specified by CPU time "ticks"). This mechanism can be used for periodic tasks or monitoring CPU usage.

Design and Implementation:

Kernel Changes:

1.Process Structure (`proc.h`):

I introduced fields to track the alarm state for each process:

uint64 alarm_handler; // Address of the handler function

 int alarm_interval;   // Interval of CPU ticks between each alarm

 int alarm_ticks; // Counter to track the number of ticks since the last alarm

 struct trapframe *alarm_tf; // Pointer to cache the trapframe when the alarm triggers

 int in_alarm; // Flag to indicate if the alarm is currently active

The `sys_sigalarm()` function allows a process to configure an interval-based alarm and specify a handler function that will be invoked when the alarm triggers.

2.System Call Implementation (`sysproc.c`):

The process's alarm state is initialized by setting `alarm_ticks` (which tracks ticks) to 0, `alarm_interval` to the user-provided `interval`, and `alarm_handler` to the address of the handler function.The `in_alarm` flag is set to 0, indicating that the handler is not currently active.

The `sys_sigreturn()` function is used to restore the process's state after the alarm handler has completed execution.

If the process is currently handling an alarm (`in_alarm` flag is set), the saved process state is restored from `alarm_tf`, a copy of the trapframe made when the alarm was triggered.Memory allocated for `alarm_tf` is freed, and the `in_alarm` flag is reset to 0.The `alarm_ticks` counter is also reset to 0, allowing the alarm mechanism to resume.

Both syscalls implementation is in the sysproc.c :

uint64 sys_sigalarm(void){}

uint64 sys_sigreturn(void){}

3.Trap handler(trap.c)

Signal handling occurs in the trap handler. When the device generates an interrupt and there is no active alarm (in_alarm is 0), a new trap frame is created, copying the current one. If the alarm interval (current_ticks) has expired, the epc in the trap frame is set to the address of the handler function, and the in_alarm is set to 1, indicating an active alarm.

Corresponding code is in – usertrap function()

To verify working , just run alarmtest – it outputs all test passed .

Scheduling :

This contains 3 scheduling policies :

1.Round Robbin (RR) // DEFAULT

2. Lottery Based Scheduling(`LBS`)

3. Multi-level Feedback Queue (MLFQ)

The Implemented Scheduling policies can be run by:

make clean
make qemu SCHEDULER=<Schedulertype>

Here , LBS for LBS, MLFQ for MLFQ and also can include CPUS = 2 for LBS and CPUS = 1 for MLFQ :

(In makefile I define a MACRO SCHEDULER to specify which to run)

Implementation of Lottery-Based Scheduling (LBS) in xv6

1. Overview:

In this task, I have implemented a **preemptive lottery-based scheduling policy** (LBS) .The LBS scheduler assigns CPU time to processes randomly, with the probability of selection based on the number of tickets each process holds. Processes with more tickets have a higher chance of being chosen to run in each time slice. Additionally, this implementation resolves lottery ties using the arrival time of processes to prioritize older processes with the same number of tickets.

2.1. System Call: `settickets(int number)`

A system call `settickets(int number)` was added, allowing processes to set their ticket count. The default number of tickets is 1, but processes can increase this value to improve their chances of winning the lottery.

Changes:

I modified `proc.h` to include the `tickets` field in the process control block (`struct proc`).In allocproc`()`, I initialized the ticket count for new processes to inherit the parent's ticket count to ensure that child processes start with the same tickets as their parent, in syscall.c I have included the settickets() function.

2.2. Lottery Mechanism:

The main logic for the lottery resides in the `scheduler()` function. Each time the scheduler selects a process to run, it calculates the total number of tickets across all runnable processes and randomly selects a winning ticket. The process that holds the winning ticket is assigned the CPU for 1 tick.

Complete code for this scheduler is in proc.c , in scheduler function.

I also made sure a child process starts with the same number of tickets as its parent. This is done in fork() in proc.c.

2.3. Ticket Tie-Breaker:

To handle the issue of multiple processes with the same number of tickets, I implemented a **tie-breaking mechanism** based on arrival time. If two or more processes have the same number of tickets, the one that arrived earlier is chosen to run.

Changes:

- I have included an `arrival_time` field. This field is set when a process enters the system. If a tie occurs, the scheduler will prioritize processes with earlier `arrival_time`.

3. Lottery-Based Scheduling Logic:

The LBS algorithm works by first calculating the total number of tickets across all runnable processes. A random number is then generated in the range of 0 to the total number of tickets, representing the "winning" ticket. The scheduler iterates through the runnable processes, summing their ticket counts, and selects the process whose ticket sum contains the winning number.

In case two or more processes have the same number of tickets, the scheduler compares their `arrival_time` values and selects the process that has been waiting the longest.

In the LBS scheduler, processes with more tickets receivea larger proportion of the CPU time, which led to shorter waiting times for processes with more tickets. However, the tie-breaking mechanism based on arrival time helped prevent starvation for processes with fewer tickets.

Conclusion:

The implementation of the Lottery-Based Scheduling (LBS) policy in xv6 introduces a dynamic and fair CPU allocation mechanism based on the number of tickets each process holds. The addition of the arrival time as a tie-breaker ensures fairness when multiple processes have the same number of tickets. Performance tests show that processes with more tickets experience shorter waiting times, and the lottery system works efficiently to provide CPU cycles proportional to the ticket count. The system's flexibility to switch between scheduling policies at compile time further enhances its usability.

Implementation of MLFQ Scheduler

1. Implementation Overview:

To implement the Multi-Level Feedback Queue (MLFQ) scheduler, I used four priority queues. Processes are assigned to a queue based on their behavior and CPU bursts. The key mechanisms of the scheduler include time-slice enforcement, priority adjustment based on CPU usage, preemption to ensure higher-priority processes get the CPU first, and priority boosting to avoid starvation.

Each process starts at priority 0 and is assigned different time slices based on its priority. Processes that use their entire time slice are demoted to a lower priority queue. Those that relinquish the CPU voluntarily (e.g., for I/O) return to the same queue upon resuming execution. Priority boosting ensures that no process gets stuck in a lower priority queue indefinitely.

2. Detailed Changes and Specifications:

2.1. Time Slice Assignment:Priority 0: 1 timer tick,**Priority 1:** 4 timer ticks,**Priority 2:** 8 timer ticks.,**Priority 3:** 16 timer ticks.

The time slice assignment is handled through a time slice counter for each process. If a process uses its full time slice, it is demoted to the next lower-priority queue.

→ All queue operations are created in proc.c file , which makes deque,enqueue operations easier.

2.2. Preemption: Processes in lower-priority queues are preempted when a new process enters a higher-priority queue. This was implemented by checking for new arrivals or the reactivation of processes in higher queues at the end of each tick.

In trap.c file , the modification is made in usertrap() function.

2.3. Voluntary I/O Release:

When a process performs I/O, it relinquishes the CPU voluntarily. Upon reactivation, it returns to the same queue it was in. I modified sleep and the wake-up routine to remove and place the process back into the correct queue, ensuring its behavior was correctly simulated.

2.4. Priority Boosting:

After 48 ticks, the system boosts all processes to the highest priority queue (queue 0). This prevents starvation of lower-priority processes and allows interactive processes a chance to execute frequently.

2.5. Round-Robin Scheduling for the Lowest Queue:

In the lowest priority queue (queue 3), a round-robin scheduling policy is implemented. This ensures that processes get a fair share of the CPU if they are at the bottom of the queue structure. This is implemented in the scheduler function itself , just pushing back into the same queue.

Totally , I added the required fields used in proc.h file , the complete scheduler part is added in scheduler function in proc.c , and Like changing priority is added in usertrap function() in trap.c file.

→ when process request for IO , I remove the process from queue and add it when it wakes up . This is done in both functions.

Performance Comparison:

I ran the `schedulertest` command to measure the average waiting and running times of processes.

-the processes run on only 1 CPU .

Round Robbin :(DEFAULT)

- **Average Waiting Time: 136**
- **Average Running Time:** 9

LBS Scheduler:

- **Average Waiting Time:** 133
- **Average Running Time:** 8

MLFQ Scheduler:

- **Average Waiting Time:** 135
- **Average Running Time:** 9

Analysis:

The **LBS scheduler** showed a slightly lower average waiting time and running time compared to both round-robin and MLFQ, indicating that processes with higher ticket counts received CPU time more efficiently, reducing the overall wait.

The **Round Robin** and **MLFQ** schedulers had similar performance, with MLFQ showing slightly better efficiency due to its prioritization of short bursts and interactive processes, though their average times were close to the default round-robin.

Q. Answer the following in 3-4 lines, and optionally provide an example: What is the implication of adding the arrival time in the lottery based scheduling policy? Are there any pitfalls to watch out for? What happens if all processes have the same number of tickets?

Implication of Adding Arrival Time to Lottery Scheduling:

Adding arrival time to lottery-based scheduling adds a fairness aspect by prioritizing processes that have been waiting longer when a ticket tie occurs. This modification ensures that older processes are not unfairly delayed simply because newer processes with the same number of tickets are introduced.

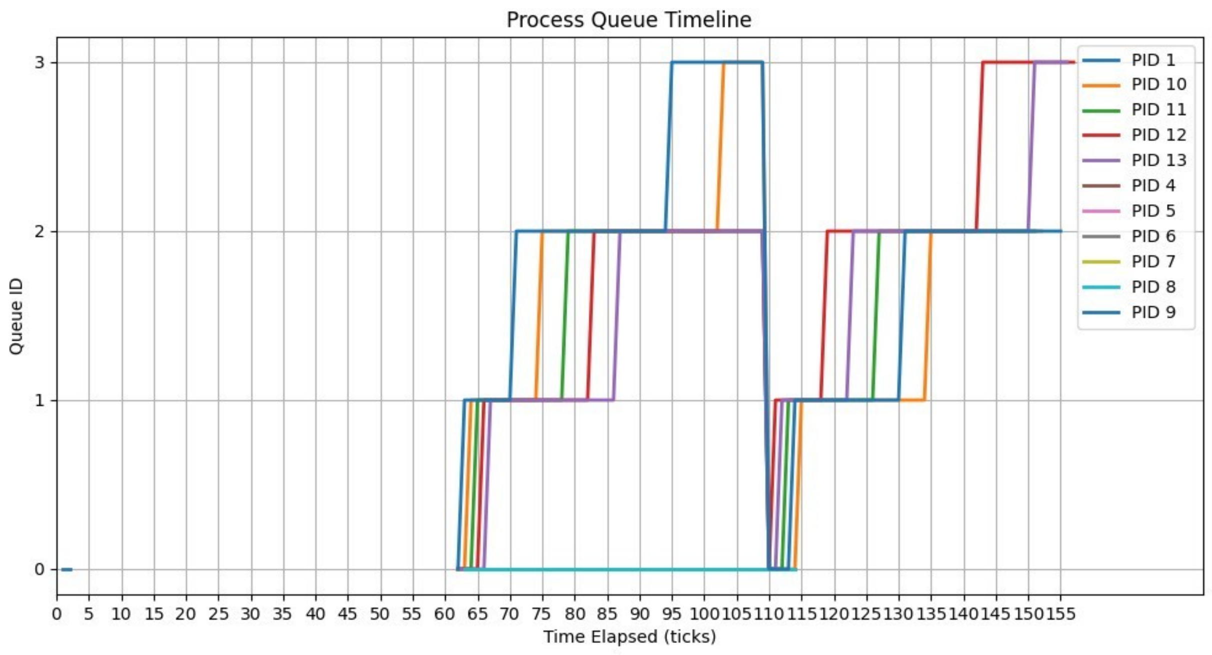
Pitfalls:

- **Starvation Risk for Low-Ticket Processes:** While using arrival time as a tie-breaker helps ensure fairness between processes with the same ticket count, processes with significantly fewer tickets can still face extended wait times. This is especially true in systems with many high-ticket processes, as the lower-ticket processes have a much lower probability of being selected, potentially leading to starvation.
- **Overhead:** Tracking and comparing arrival times introduces additional overhead, especially in a system with a large number of processes. The scheduler must check both ticket counts and arrival times, potentially adding extra computational load to each scheduling decision. But it is manageable with low no.of processes , given the fairness it introduces.

When All Processes Have the Same Number of Tickets:
In this scenario, the system effectively operates like a **round-robin** scheduler, as all processes have an equal chance of being selected. However, with the tie-breaking mechanism based on arrival time, the scheduler behaves similarly to **First-Come First-Serve (FCFS)** for processes with the same ticket count. The oldest process, or the one that arrived first, is given priority, ensuring fairness and preventing any process from being favored purely by chance.

MLFQ Analysis:

The following timeline graph shows the queue evolution over time for each process:



- **X-axis (time elapsed):** The total time elapsed since the scheduler started.
- **Y-axis (queue ID):** The priority queue a process belongs to (0 to 3).
- **Color-coded Processes:** Each process is represented by a different color.

In the graph, you can observe:

Dynamic Queue Movement: Processes are moved between queues based on CPU usage, with CPU-bound processes dropping to lower queues, while I/O-bound or interactive processes stay in higher-priority queues. Interactive processes relinquishing the CPU remain in higher queues, improving system responsiveness.

Priority Boosting: The priority boost mechanism occurs every 48 ticks, moving all processes back to Queue 0 to prevent starvation of long-running tasks.

Round-Robin in Lowest Queue: Processes in the lowest-priority queue (Queue 3) are scheduled in a round-robin fashion, ensuring fairness among CPU-bound tasks.

Balanced System:MLFQ provides quick CPU access to short tasks while effectively managing long-running CPU-bound processes without monopolization.

MLFQ Analysis Summary:

The MLFQ scheduler shows a clear advantage in terms of prioritizing I/O-bound and interactive processes, ensuring they remain in higher queues. This leads to faster response times for those processes, making the system more interactive-friendly. CPU-bound processes are pushed to lower queues, ensuring they do not monopolize the CPU. However, the priority boost ensures that long-running processes are not starved. The timeline graph highlights these dynamics effectively.