Operating Systems and Networks

Enhanced XV6 Report

System Calls

1. GOTTA COUNT 'EM ALL

In order to implement the system call getSysCount and the user program syscount, firstly the definitions for both have to be added. The system call getSysCount is defined in kernel/syscall.h (SYS_getSysCount) and then it is mapped to an actual function is kernel/syscall.c (sys_getSysCount). This function is defined in kernel/sysproc.c. For the user program syscount, I created a new file called user/syscount.c which takes the command line arguments as specified in the specifications document, executes the given command and using the system call, gets and prints the required count. The system call needs to be added to user/user.h as well as user/usys.pl for the program to run. An entry has to be made in the MAKEFILE as well ($U/_getSysCount) for the user program to compile.

As asked, we need to run the given command and keep count of the specified system call for that command and all of its children. This is done by adding two arrays to the proc struct in kernel/proc.h (syscall_count and child_syscall_count). In kernel/syscall.c, everytime a system call is called, the counter is updated for the corresponding process and system call. This way we are keeping track of calls made by every process. However to also account for children, everytime a process exits, its system call counts are added to the parent's array. When the user program syscount is run, it calls getSysCount after forking a child for the command to be run, so the command which was run is actually a child (the only child) of the current process. Thus, in kernel/sysproc.c, the function actually returns the child_syscall_count for the corresponding system call. In this way, this specification has been implemented.

2. WAKE ME UP WHEN MY TIMER ENDS

For this specification, two new system calls need to be added (sigalarm and sigreturn). For this, I again added definitions required to the important files as explained in the previous specification. To keep track of the alarms, a few more fields were added to the proc struct in kernel/proc.h (alarm, alarm_handler, alarm_t, cur_t, alarm_trapframe, handle). In the system call sigalarm, I set alarm to 1 (to indicate the alarm is on) and specify the alarm_handler as given to the system call. Alarm_t and cur_t keep track of how many ticks to wait for each alarm and how many ticks have gone by already. Alarm_trapframe is for storing the trapframe during the alarm, and handle is 0 while the alarm is being handled.

The main part of this implementation is in kernel/trap.c where in usertrap(), when a time interrupt occurs and the alarm is on (and isn't being handled already), if the cur_t matches alarm_t for the process then I store the trapframe and go to the handler.

The handler always has to end by calling sigreturn, which just switches back to the trapframe stored to continue the program's execution. This has been tested successfully by user/alarmtest.c given.

Scheduling

1. THE PROCESS POWERBALL

This requires the scheduler to conduct a lottery and randomly decide a winning lottery number, using which a process is chosen for execution. Firstly, to enable multiple scheduling algorithms, the MAKEFILE is changed to include a preprocessor directive called SCHEDULER (by default = RR). Now in kernel/proc.c, in the scheduler() function, I introduced #ifdef directives so if RR was specified, then the default given code runs, but if LBS was specified, my new lottery based scheduling runs.

Firstly, the system call settickets has to be added, which takes a number for tickets, and sets the calling process' tickets to that. Thus a field called tickets has been

introduced to the proc structure in kernel/proc.c which is just set to the given number. As also mentioned, any child forked should get the same number of tickets as its parent, so this has been done in fork() in kernel/proc.c.

To implement the lottery, first the total number of tickets is computed across all processes. Then a rand() function (implemented in kernel/rand.h) is used to get a winning lottery ticket number. Now as in the traditional lottery, the processes array is traversed to find the winner (the process containing the winning ticket in their range of tickets). However this needs to be changed so that any process with those many tickets but an earlier arrival time wins instead. So the arrival field is added to the struct proc as well, which stores the ticks of the cpu in allocproc() in kernel/proc.c which enables us to traverse the array once again to find the process with those many tickets as the earlier winner, and also having the earliest arrival. This final winner is run preemptively.

2. MLF WHO? MLFQ!

Similar to LBS, MLFQ is passed as a preprocessor directive. To implement the queueing, instead of using actual queues I have used a field in each proc structure called queue, which stores the queue number the process is in currently. Another field has been used to keep track of the number of ticks the process has spent in the current queue (to enable demotion to lower queues). Another field called run is used for the lowest queue to run in round robin.

In the scheduler, every 48 ticks all processes are boosted to queue 0. Then I check the first 3 queues sequentially, finding the first process in the lowest possible queue which can be run. This is run, and if it consumes a tick it is incremented in the respective fields. Now, if the process has used up its time slice in the queue, it is demoted. If no process was found in this, I loop through the processes and run them in a round robin fashion by utilizing the run field of the proc struct. This is done by setting run to 1 for every process run in the round robin, and if all processes have run set to 1, reset them all to 0. Thus the MLFQ has been implemented, with major changes to kernel/proc.c.

3. Additional Comparison

I edited user/schedulertest.c a bit, and I've made 2 IO bound processes and 3 CPU bound processes. Running this for all 3 scheduling policies with 1 CPU each, the following average running and waiting times were obtained:
RR - Average running time 23, waiting time 97
LBS - Average running time 20, waiting time 105
MLFQ - Average running time 28, waiting time 100

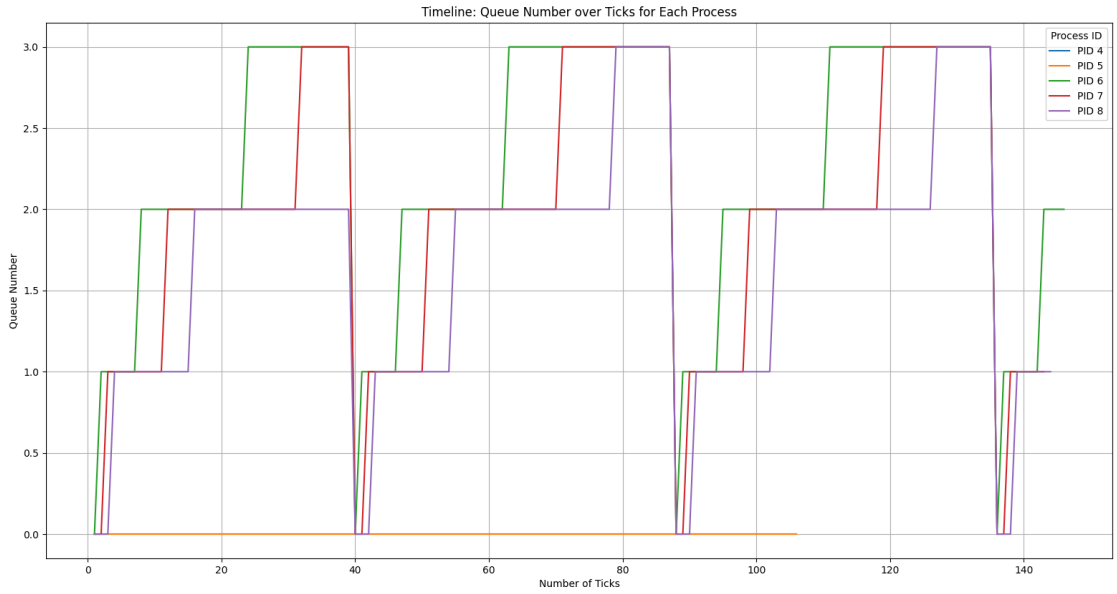
RR has the lowest waiting time but slightly longer running times.
LBS results in the shortest running times but the highest waiting time, indicating that it benefits CPU-bound processes but IO-bound ones wait longer.
MLFQ results in longer running times, but it balances between RR and LBS in terms of waiting time.

This comparison highlights how each policy affects process performance differently, with LBS favoring CPU-bound processes and MLFQ providing a balanced approach.

4. Question About LBS

Adding arrival time in the lottery-based scheduling policy means that among processes with the same number of tickets, the one that arrived earlier is prioritized. This ensures fairness, preventing late-arriving processes with the same tickets from overtaking early ones just due to random chance. However, a pitfall is that this could lead to a "starvation" scenario where newer processes are constantly delayed by older ones, especially if many processes have similar ticket counts. If all processes have the same number of tickets, the scheduler will essentially behave like a first-come-first-served policy, since earlier processes will always win in case of a tie.

5. MLFQ Analysis



Copy-On-Write Fork

1. Page Fault Frequency

In this analysis, we evaluate the Copy-On-Write (COW) mechanism implemented in `xv6` by measuring page fault frequency under various scenarios. Using different types of memory operations—read-only, partial writes, full writes, and multiple child processes modifying memory—we analyzed how often the COW mechanism was triggered.

To record the page faults, we implemented a `get_page_faults()` system call, which allowed us to track and print page fault counts at different stages in each test scenario. Below are the findings for each scenario:

1. **Initial State**

   Before any tests were run, the system reported **2 page faults**. This baseline provides insight into the system's initial page management overhead, which we can compare against further operations.

2. **Read-Only Operations**
   In this scenario, a child process forked from the parent performs read-only operations across the allocated memory. The child process encountered **1 page fault** before beginning its reads and showed no increase in page faults after reading all pages. This confirms that read-only operations do not trigger additional page faults under COW, as no pages are duplicated unless a write occurs. After the child exited, the parent process reported **3 page faults** in total, showing a minimal overhead increase.
3. **Partial Page Writes with Varying Steps**
   Here, a child process is modified every third page, simulating a sparse write pattern. The child process started with **1 page fault** before modifications, and each page modification incrementally increased the page fault count, reaching **4 page faults** by the end. This behavior demonstrates the effectiveness of the COW mechanism, as only the pages that were modified by the child process were duplicated. Upon the child's exit, the parent maintained a total of **4 page faults**, indicating efficient memory conservation and minimal duplication.
4. **Full Page Writes with Unique Values**
   This test involved a child process modifying all pages, with each page receiving a unique value to avoid optimization. The child process started with **1 page fault** and encountered additional faults for each page it modified, resulting in **9 page faults** by the end of its execution. This outcome is expected under COW: every written page is duplicated to prevent shared modifications. The parent process retained **5 page faults** after the child exited, reflecting a moderate increase in memory usage but maintaining efficient page management.
5. **Multiple Children with Staggered Modifications**
   To test concurrency and independent modifications, three child processes were forked, each modifying staggered pages to minimize overlap. Each child encountered page faults as it modified pages, with totals of **4 page faults** for each child by the end of their respective executions. After all child processes completed, the parent process had a final total of **6 page faults**, indicating that each child's modifications were isolated efficiently through COW without significant page duplication in the parent.

2. Benefits of COW Fork

Copy-on-Write (COW) fork brings substantial memory efficiency by allowing parent and child processes to share memory pages initially rather than duplicating them immediately. This technique significantly reduces memory usage in systems with numerous forked processes, as pages are duplicated only when a write operation occurs. This lazy duplication ensures memory is allocated on an as-needed basis, conserving resources. Furthermore, COW enhances performance by accelerating fork operations, as memory copying is deferred until necessary. Efficient resource utilization means that the system can support more processes simultaneously, a boon for applications that require extensive forking without exhausting memory.

Optimizations to improve COW focus on more efficient management of memory pages and fault handling. Implementing page reference counting can improve tracking of shared pages, ensuring duplications happen only when needed. Batching page fault handling, rather than processing each fault individually, could also reduce overhead during high-frequency write operations. Additionally, selective COW could be applied by protecting only essential pages, reducing page faults and further enhancing system efficiency. Integrating COW with memory swapping mechanisms could prove particularly beneficial for systems with limited physical memory, as it would help manage resources more effectively. Finally, adopting memory allocation strategies in user-space applications that are COW-friendly can enhance the effectiveness of COW, making it a more robust and scalable solution for large applications.