CS3500: Operating Systems

Project topics

Date: 17th Oct 2019

General Instructions

1. Each project is to be done in a team of two. You can choose your partners.
2. Up to 6 teams can take up a given project topic. Allotment of topics will be done based on preferences, with random tie breaking. Each team has to choose three projects in order of preference. This will be announced in the lecture on 21st October.
3. Each project has three levels, which are to be done sequentially. Each team can choosen how many levels to attempt.
4. Level 1 has a maximum of 4 points, Level 2 has a maximum of 3 points, and Level 3 has a maximum of 3 points. Thus, a team attempting Levels 1 and 2 is graded out of 7.
5. Each project has an assigned mentor who will be the contact point for the discussion. Each team is expected to meet the mentor at least twice during the project duration.
6. The project starts on 20th October and will be evaluated in the weekend of 9th and 10th November.
7. Each project will be allotted a presentation time equal to twice the number of points they have attempted. Thus, team doing level 1 gets 8 mins, team doing levels 1 and 2 gets 14 mins, and team doing levels 1, 2, and 3 gets 20 mins.
8. Each presentation is expected to cover the approach and execution results to establish correctness and good design.
9. Links for presentations (same tool as during recitation) will be shared a week before the presentation date.
10. Apart from the presentation, each project requires a code submission, the last date for which will be 8th November. The format will be shared by the respective mentors. There will be no report.
11. Plagairism checks will be performed on submissions on the same topic. Any discrepancy will lead to zero scores for all involved teams.
12. We have been speaking about mechanism-vs-policy in OS. Here, the project is the mechanism, and the policy is to have fun. :-)

1 Project 1, mentor: Kavya

In this project, you will enhance the shell of xv6 with a utility with functionalities similar to `ps` and `top` in standard Linux. We will call this utility `topps`.

1.1 Level 1

Support the command `topps` that prints the following information for all current processes:

1. the pid of the process,
2. the name of the process,
3. the pid of the parent process,
4. the current process state,
5. the size of the virtual memory, and
6. the time elapsed since the creation of the process.

Hint: For the last item, you have to add a new field in the `proc` struct that is updated when a process is allocated in `allocproc()`. You would also need to work with a function like `ssm()` that you build for Lab 10.

1.2 Level 2

Modify the command `topps` to also print the total time executed by each process.
Hint: This requires a new field in the `proc` struct which is updated during context switches in functions `sched()` and `scheduler()`.

1.3 Level 3

Modify the command `topps` to classify the total time executed to one of two classes: amount of time in user space and amount of time in kernel space. Write two user space programs to illustrate how different processes take different amounts of time in user and kernel spaces.
Hint: This requires modifying the interrupt/syscall handling.

2 Project 2, mentor: Nikhilesh

In this project, you will improve upon the simple scheduler used in xv6.

2.1 Level 1

The current scheduler loops through all processes until it finds a process in state `RUNNABLE` and schedules it. This has a $O(n)$ worst-case complexity. Rewrite the scheduler to have the same functionality, but $O(1)$ complexity.
Hint: Maintain a separate list of pids of processes that are in the `RUNNABLE` state.

2.2 Level 2

Clearly, the current implementation is not fair. Modify your implementation to ensure the following fairness property: If process p1 is chosen to be scheduled while p2 is waiting in `RUNNABLE` state, then p1 will not be scheduled again before p2 is scheduled at least once. Write a user-space program that illustrates the benefit that your fairer scheduler brings.
Hint: Use an ordered data structure to store `RUNNABLE` processes, and use a single user-space program with forks and file reads to illustrate the working of your scheduler.

2.3 Level 3

Make your fair implementation more generic by supporting the lottery scheduler which allows you to specify a certain number of lottery tickets for each process. Enable a process to optionally specify the number of lottery tickets to be assigned to a child after forking. Illustrate this scheduler with a user space program.
Hint: Add an additional field to the `proc` struct that specifies the number of lottery tickets.

3 Project 3, mentor: Prasanna

In this question, you will add a fixed priority scheduler to xv6.

3.1 Level 1

Support a `prio` setting for each process, which is randomly assigned between a `MIN_PRIORITY` = 10 and a `MAX_PRIORITY` = 0. In the scheduler, amongst all processes in the `RUNNABLE` state, the one with the highest priority (lowest `prio`) is to be scheduled.
Hint: Add `prio` to the `proc`, and assign it during the allocation of the process. Break ties, if any, randomly.

3.2 Level 2

Support two new system calls `getpriority()` and `setpriority()` which respectively get and set `prio` of the calling process. Enable an immediate scheduling decision upon modification to `prio` values.
Hint: Observe how the scheduler is currently triggered and replicate that in the system calls you write.

3.3 Level 3

Improve your dynamic priority scheduler further, by adding feedback. In the scheduler, maintain a log of time executed by each process the last time it was scheduled. If this is less than a threshold, then increase the priority of that process (decrease `prio`). If that is more than the threshold, then decrease the prioity of that process (increase `prio`). Illustrate your scheduler with a user space program.
Hint: You need to maintain only the time the process ran between it being scheduled to being context switched out.

4 Project 4, mentor: Gnanambikai

In this project, you will work with the virtual memory and support a just-in-time approach to growing the heap.

4.1 Level 1

Support two new system calls `count_virtual_pages()` and `count_physical_pages()` which return the count of virtual and physical pages assigned to the calling process.
Hint: The virtual page count is simply based on the allocated virtual memory size, while the physical page count requires walking through the page table and counting the number of entries that are valid.

4.2 Level 2

Currently the system call `sbrk()` grows the allocated memory for a process by allocating a new page, when necessary. A just-in-time approach would be to allocate the new page, only when that memory is accessed. Modify `sbrk` to not allocate the new page, but ensure that the `sz` field of the `proc` struct is updated. With this modification, you should get an exception when running basic commands. Identify the exception by executing an existing xv6 command line utility like `echo` or `cat`.

4.3 Level 3

Now, you need to patch the exception handler for the exception that you identified to do the actual page allocation. This requires allocating a new page and also updating the page table entries. Verify that the exception no more occurs.
Hint: To find out the virtual address where the exception was found, you can use the function `cprintf`. You can then find the start of the page table for this virtual address with the function `PGROUNDDOWN()`.

5 Project 5, mentor: Prasanna

In this project, you will improve the interface between a parent and the forked child process.

5.1 Level 1

The `wait()` system call is supported in xv6. However, there is no support for a `waitpid()` call. Add a `waitpid(pid)` system call which waits for a child of a given pid or return -1 if there is no active child with that pid. Demonstrate this with a user space program.

5.2 Level 2

Add support for `waitpid(0)` which waits for *all* child processes, thereby implementing a barrier sync. Also add support for `waitpid(-1)` which waits for *some one* child process. Demonstrate these with a user space programs.

5.3 Level 3

With the full suite of `waitpid` calls, it will be essential for the parent to know which child processes finished. Support a second pass by reference argument for `waitpid` which contains a -1 terminated list of pids of all child processes that terminated.

6 Project 6, mentor: Kavya

In this project, you will create conditions for race conditions and find ways to avoid them.

6.1 Level 1

Processes in xv6 do not have shared memory and thus no race conditions on shared variables. To simulate concurrent access to shared variables, support three system calls `counter_init()`, `counter_get()`, and `counter_set()`, which set the counter to 0, get the counter value, and set the counter value, respectively. The counter value is a global value maintained in the kernel space. Demonstrate race conditions with two user space programs concurrently accessing the counter.

6.2 Level 2

xv6 supports locks in the kernel space, but these are not exposed to the user space. Create two additional system calls `my_lock` and `my_unlock` which acquire and release a lock in the kernel space, that guards the access to the counter. Demonstrate the use of these calls to avoid race conditions seen in the previous question.
Hint: Define a lock corresponding to the counter and then use standard acquire and release functions when supporting your new system calls.

6.3 Level 3

The current implementation of locks in xv6 are spinlocks. Modify these to *futex* which first busy-loop for a certain time, and after that they yield. Once they are rescheduled the availability of the lock is checked again for a bounded time and if not yet available, they yield again. Evaluate the performance of your futex with user-space programs accessing the shared counter.
Hint: Implement the timed busy-loop as a fixed number of busy for loops.

7 Project 7, mentor: Gnanambikai

In this project, you will work on an aspect of security, namely creation of covert channels between processes. A covert channel is a security vulnerability whereby two processes can communicate with each other without being explicitly allowed to. In xv6, processes do not communicate through shared memory. But because they run on the same architecture, through covert channels they can share information, violating the OS scheme.

7.1 Level 1

First, you will extend the lab exercise on TLBs. Write various programs to estimate the TLB structure: the number of entries in the first level TLB, the number of levels of the TLB, the associativity, etc. Such information is important for user-space processes to enable covert channel attacks.
Hint: You can access data in different strides (contiguous, page size, etc.) to determine different TLB parameters. Carefully ensure that you isolate the effect of cache access.

7.2 Level 2

The next step in the attack is for two processes to be able to create and infer architectural side-effects of their execution. Write a memory access pattern by which two processes have a collision on the TLB. In other words, ensure that the number of TLB hits and misses of one process deterministically depends on whether the other process is running.
Hint: This requires some explanation. Talk to the TA!

7.3 Level 3

The next stage in the attack is to be able to pass meaningful information and also benchmark the rate of flow of this information along the covert channel. Modify the two processes from the previous level, where the two processes take up distinct reader and writer roles. Both processes arrange their memory accesses such that the writer can transmit a specific information, and the reader can then infer this information, based on collisions on the TLB.
Hint: Start with transmission of a single bit, and then expand to multiple bits.