Mini Project 2 : Introduction to XV-6

Deadline: 11:59pm 06 October 2023

xv6 is a simplified operating system developed at MIT. Its main purpose is to explain the main concepts of the operating system by studying an example kernel. xv6 is a re-implementation of Dennis Ritchie's and Ken Thompson's Unix version 6 (v6). xv6 loosely follows the structure and style of v6, but is implemented for a modern RISC-V multiprocessor using ANSI C. In this assignment you will be tweaking the Xv6 operating system as a part of this assignment. Boilerplate code has been provided On Github Classroom. You can see the install instructions here. You might find resource here to be helpful.

GitHub Classroom

Follow this link to accept the assignment. You will then be assigned a private repository on GitHub. This is where you will be working on the mini project. All relevant instructions regarding this project can be found below.

Specification 1: System Calls [20 marks]

System calls provide an interface to the user programs to communicate requests to the operating systems. In this specification, you're tasked to implement the following syscalls:

System Call 1 : `getreadcount` [5 marks]

Add the system call `getreadcount` to xv6.

The system call returns the value of a counter which is incremented every time any process calls the *read()* system call.

Following is the function signature for the call:

```
int getreadcount(void)
```

System call 2 : `sigalarm` and `sigreturn` [15 marks]

In this specification you'll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want to take some periodic action. More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers like the `SIGCHILD` handler in the previous assignment.

You should add a new `sigalarm(interval, handler)` system call. If an application calls `alarm(n, fn)` , then after every n "ticks" of CPU time that the program consumes, the kernel will cause application function `fn` to be called. When `fn` returns, the application will resume where it left off.

Add another system call `sigreturn()` , to reset the process state to before the `handler` was called. This system call needs to be made at the end of the handler so the process can resume where it left off.

NOTE: Instructions to test your code are given in the `initial_xv6/README.md` file present on Github Classroom

Specification 2: Scheduling [40 marks]

The default scheduler of xv6 is round-robin-based. In this task, you'll implement two other scheduling policies and incorporate them in xv6. The kernel shall only use one scheduling policy which will be declared at compile time, with default being round robin in case none is specified.

Modify the `makefile` to support the SCHEDULER macro to compile the specified scheduling algorithm. Use the flags for compilation:-

- First Come First Serve = `FCFS`
- Multilevel Feedback Queue = `MLFQ`

a. FCFS (First Come First Serve) [10 marks]

Implement a policy that selects the process with the lowest creation time (creation time refers to the tick number when the process was created). The process will run until it no longer needs

CPU time.

HINTS:

1. Edit the struct proc (used for storing per-process information) in kernel/proc.h to store extra information about the process.
2. Modify the allocproc() function to set up values when the process starts. (see kernel/proc.h)
3. Use pre-processor directives to declare the alternate scheduling policy in scheduler() in kernel/proc.h.
4. Disable the preemption of the process after the clock interrupts in kernel/trap.c

b. Multi Level Feedback Queue (MLFQ) [30 marks]

Implement a simplified preemptive MLFQ scheduler that allows processes to move between different priority queues based on their behavior and CPU bursts.

- If a process uses too much CPU time, it is pushed to a lower priority queue, leaving I/O bound and interactive processes in the higher priority queues.
- To prevent starvation, implement aging.

Details:

1. Create four priority queues, giving the highest priority to queue number 0 and lowest priority to queue number 3
2. The time-slice are as follows:
    a. For priority 0: 1 timer tick
    b. For priority 1: 3 timer ticks
    c. For priority 2: 9 timer ticks
    d. For priority 3: 15 timer ticks

    NOTE: Here tick refers to the clock interrupt timer. (see kernel/trap.c)

Synopsis for the scheduler:-

1. On the initiation of a process, push it to the end of the highest priority queue.

2. You should always run the processes that are in the highest priority queue that is not empty.

    Example:

Initial Condition: A process is running in queue number 2 and there are no processes in both queues 1 and 0.

Now if another process enters in queue 0, then the current running process (residing in queue number 2) must be preempted and the process in queue 0 should be allocated the CPU.(The kernel can only preempt the process when it gets control of the hardware which is at the end of each tick so you can assume this condition)

3. When the process completes, it leaves the system.
4. 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.
5. If a process voluntarily relinquishes control of the CPU(eg. 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
6. A round-robin scheduler should be used for processes at the lowest priority queue.
7. To prevent starvation, implement aging of the processes:
   a. If the wait time of a process in a priority queue (other than priority 0) exceeds a given limit (assign a suitable limit to prevent starvation), their priority is increased and they are pushed to the next higher priority queue.
   b. The wait time is reset to 0 whenever a process gets selected by the scheduler or if a change in the queue takes place (because of aging).

NOTE

Procdump:

This will be useful for debugging (you can refer to it's code in kernel/proc.c). It prints a list of processes to the console when a user types Ctrl-P on the console. You can modify this functionality to print the state of the running process and display the other relevant information on the console.

Use the procdump function to print the current status of the processes and check whether the processes are scheduled according to your logic. You are free to do any additions to the given file, to test your scheduler.

Getting runtime and waittime for your schedulers:

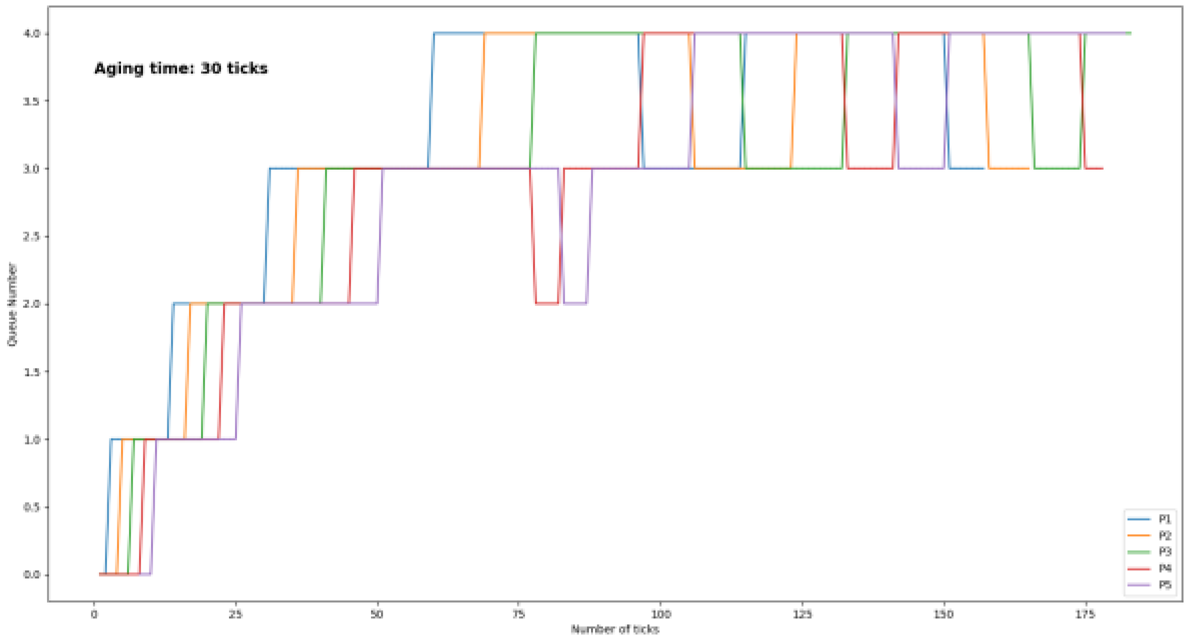
Run the `schedulertest` command once your implementation is complete.

Specification 3 : Report [10 marks]

- The report must contain explanation about the implementation of the various implemented scheduling algorithms.
- Include the performance comparison between the default and 2 implemented policies in the README by showing the average waiting and running times for processes. Set the processes to run on only 1 CPU for this purpose.

  ## MLFQ scheduling analysis [5 marks]

- Create timeline graphs for processes that are being managed by MLFQ Scheduler. 'Variate the length of time that each process consumes the CPU before willingly quitting using the benchmark from Specification 2. The graph should be a timeline/scatter plot between queue_id(y-axis) and time elapsed(x-axis) from start with color-coded processes.



Specification 4 : Networking [30 marks]

UDP and TCP, but mostly TCP

Part A : Using library functions [12 marks]

Implement the following within the `<mini-project2-directory>/networks/partA/basic` directory

1. TCP server program
2. TCP client program

3. UDP server program
4. UDP client program

The client program must send some text to the server and then receive some text from the server.

You shall be awarded `1 mark per program` listed above *only* if you check for all the errors that could come up, i.e., check the return status of all the relevant functions.

Now, for the rest of the 8 marks - you must make modifications to the programs listed above and add your submissions to `<mini-project2-directory>/networks/partA/rpc`

Here's the task - Implement rock, paper, scissor between two clients (4 marks for the UDP implementation and the other four for using TCP)

- Start server which listens for two clients on two different ports.
- Start two clients - clientA and clientB
- Users enter their decision (Rock, Paper or Scissors) in clientA and clientB.
- Server receives the decisions from clientA and clientB.
- Server deliberates and returns it's judgement to both the clients.
- Clients display the judgement ("Win", "Lost", "Draw")
- Implement a loop such that the clients are prompted for another game after the judgement (The next game starts only if both the clients say yes - handle this using the server)

NOTE: we are NOT expecting anything fancy here - using 0 for Rock (i.e., expecting the user to input 0 for rock), etc., is also fine.

Libraries you could use are - arpa/inet.h, sys/types.h, sys/socket.h, netinet/in.h

Resources that you could refer to -

1. https://www.csd.uoc.gr/~hy556/material/tutorials/cs556-3rd-tutorial.pdf - slides on socket programming
2. https://github.com/nikhilroxtomar/TCP-Client-Server-Implementation-in-C - TCP client and server in C using library functions (be wary as they haven't used `htons()`)
3. https://github.com/nikhilroxtomar/UDP-Client-Server-implementation-in-C - UDP client and server in C
4. man pages :)

Part B : Implement some TCP (?) functionality from scratch [18 marks]

> Seems daunting but really isn't (read on)

We can't really ask you to implement the entire TCP/IP stack from scratch for about twenty marks (But, for the ones interested here's a repo on it - https://github.com/saminiir/level-ip)

So, here's what you actually need to do -

Implement *some* TCP functionality using UDP sockets

Seems hack-y? Very. But, it's OSN not just N and the point of this course is to hopefully make you understand some stuff.

Functionalities that you have to implement are (10 marks):

1. **Data Sequencing**: The sender (client or server - both should be able to send as well as receive data) must divide the data (assume some text) into smaller chunks (using chunks of fixed size or using a fixed number of chunks). Each chunk is assigned a number which is sent along with the transmission (use structs). The sender should also communicate the total number of chunks being sent [1]. After the receiver has data from all the chunks, it should aggregate them according to their sequence number and display the text.

2. **Retransmissions**: The receiver must send an ACK packet for every data chunk received (The packet must reference the sequence number of the received chunk). If the sender doesn't receive the acknowledgement for a chunk within a reasonable amount of time (say 0.1 seconds), it must resend the data. However, the sender shouldn't wait for receiving acknowledgement for a previously sent chunk before transmitting the next chunk [2].

   [1] Regardless of whether you use a fixed number of chunks

   [2] For implementation's sake, send ACK messages randomly to check whether retransmission is working - say, skip every third chunk's ACK. (Please *comment* out this code in your final submission)

Edit: You may simulate only one client and one server as there is no specification asking for demonstrating a connection between a client and a server.

To make your life simpler, the rest of the 8 marks is a report -

1. How is **your** implementation of data sequencing and retransmission different from traditional TCP? (If there are no apparent differences, you may mention that) (3 marks)
2. How can you extend **your** implementation to account for flow control? You may ignore deadlocks. (5 marks)

Submit your implementation for the server and client in `<mini-project2-directory>/networks/partB`

Guidelines:

1. Do not change the basic file structure given on Github Classroom.
2. No deadline extensions will be granted.
3. We will use more than 2 CPUs to test for FCFS. But for the MLFQ scheduler, we will only use 1 CPU.
4. Whenever you add new files do not forget to add them to the Makefile so that they get included in the build.
5. Make sure to include a detailed **report, describing the implementation of the scheduling algorithms [specification 2] , failing which will result in direct 0 marking for those questions.**

Do NOT take codes from seniors or your batch mates, by any chance. We will extensively evaluate cheating scenarios along with the previous few year's submissions.

Viva will be conducted during the evaluations, related to your code and also the logic/concept involved. If you're unable to answer them, you'll get no marks for that feature/topic that you've implemented.