January 2025 CSE 314: Operating System Sessional
Assignment 2: xv6 - System Call and Scheduling

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology

May 2025

Overview

In this assignment, you will implement both a new system call and a new scheduling algorithm for the xv6 operating system. This assignment will help you understand how system calls work, introduce you to some internals of the xv6 kernel, and let you explore advanced scheduling mechanisms. There are two main tasks:

1. Implement a new system call to monitor and collect system call statistics
2. Implement a multilevel feedback queue (MLFQ) scheduler

Change Log

Description	Updated By	Timestamp
Assignment declared	NTD	12 May, 2025, 11:00 PM
Modified syscall description (1.1 & 1.2.1)	NTD	22 May, 2025, 02:00 PM

1 Task 1: System Call History

Suppose our favorite OS xv6 is under virus attack. The virus is calling various system calls on its own. To better understand system behavior and monitor for suspicious activity, we need to track system calls with their relevant history.

1.1 Task Overview

Implement a new system call `history` that will return the aggregated history of system calls (how many times each was called and the system time it consumed). It should take one argument, an integer `syscall_number` (and possibly another argument, described in section 1.2.1) which denotes the system call number to trace. For example, to get history about the fork system call, a user program calls `history(SYS_fork)`, where `SYS_fork` is the syscall number from `kernel/syscall.h`.

1.2 Requirements

1.2.1 System Call Implementation

After the return from your system call, you have a pointer (defined in user program) to a `struct` object. This `struct` object should contain:

1. The name of the system call
2. The number of times this system call was made
3. The total time consumed from boot-up by this system call

A sample structure could be:

```c
struct syscall_stat {
    char syscall_name[16];
    int count;
    int accum_time;
};
```

And the corresponding system call signature - `int history(int, struct syscall_stat*);`

1.2.2 User Program

You must implement a `history.c` user program that calls this system call and displays the results. For example, when you run `history 5`, the history user program will fetch the necessary info from the kernel and then print in the console from user mode.

A sample output might look like this:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ history 12
12: syscall: sbrk, #: 1, time: 0
$ history 5
5: syscall: read, #: 21, time: 58
```

```
$ history 5
5: syscall: read, #: 31, time: 91
$ history 22
22: syscall: history, #: 3, time: 0
$ history
1: syscall: fork, #: 6, time: 0
2: syscall: exit, #: 0, time: 0
3: syscall: wait, #: 4, time: 1
4: syscall: pipe, #: 0, time: 0
5: syscall: read, #: 50, time: 278
6: syscall: kill, #: 0, time: 0
7: syscall: exec, #: 7, time: 0
8: syscall: fstat, #: 0, time: 0
9: syscall: chdir, #: 0, time: 0
10: syscall: dup, #: 2, time: 0
11: syscall: getpid, #: 0, time: 0
12: syscall: sbrk, #: 5, time: 0
13: syscall: sleep, #: 0, time: 0
14: syscall: uptime, #: 0, time: 0
15: syscall: open, #: 3, time: 1
16: syscall: write, #: 670, time: 2
17: syscall: mknod, #: 1, time: 0
18: syscall: unlink, #: 0, time: 0
19: syscall: link, #: 0, time: 0
20: syscall: mkdir, #: 0, time: 0
21: syscall: close, #: 1, time: 0
22: syscall: history, #: 25, time: 0
23: syscall: settickets, #: 0, time: 0
24: syscall: getpinfo, #: 0, time: 0
```

1.2.3 Explanation

- `history 12` means return history for system call 12. `sbrk` is the system call for index 12 (refer to `syscall.h`). It has been called only once. Moreover, the time (xv6 ticks) consumed by it is 0.
- Notice the difference between the outputs of two `history 5` calls as more system calls occur between them.
- `history 22` confirms our implementation is correct. Here we have used number 22 for the history system call.
- Finally, `history` without any arguments should print history for all system calls. The 23rd and 24th system call `settickets` and `getpinfo` are explained in later sections.

All information must be printed from **user mode**. Printing anything in kernel mode is not allowed.

1.2.4 Implementation Notes

- Please refer to the system call `fstat` to get an idea of how to return a structure object pointer.
- You need to modify `syscall.c` to count the occurrences of each system call and measure their execution time.
- Introduce the appropriate data structures to globally track system call statistics. Initialization of these structures may be necessary; `kernel/main.c` is a suitable location for such initialization (refer to the `procinit()` function for guidance about locks).
- You might need to add functions in the kernel to maintain the counters and timers for each system call.

1.2.5 Locking

Xv6 is a multiprocessor system. There is a variable `CPUS` in `Makefile`. If two processes running the same system call on different CPUs increment the counter for this system call at the exact same time, this may lead to one update not being recorded.

An effective approach to addressing this issue involves the use of locks. Refer to the implementation of `tickslock` in `xv6` as a reference for employing locking mechanisms. To ensure the operating system maintains high efficiency, fine-grained locking techniques should be utilized.

2 Task 2: MLFQ Scheduler

In this task, you will implement a new scheduler for the xv6 operating system. This scheduler will implement an MLFQ (Multilevel Feedback Queue) scheduling algorithm with two queues – one implementing lottery scheduling and the other working in a Round-Robin fashion.

2.1 Multilevel Feedback Queue

An MLFQ scheduler uses multiple queues, each having a prespecified scheduling algorithm and a limit of time slices (clock ticks). Usually, the time limit increases from top to bottom. Processes are scheduled according to the following rules:

- A new process is always inserted at the end (tail) of the topmost queue (queue 1).
- The scheduler searches the queues starting from the topmost one for scheduling processes.
  - When the scheduler finds a non-empty queue, it schedules a process according to the specified algorithm of that queue.
  - After the process leaves the CPU (either voluntarily or being preempted), the scheduler starts searching again from the topmost queue.
- When a process is assigned to the CPU:
  - If it is completed within the time limit of its queue, it leaves the system.
  - If it voluntarily relinquishes control of the CPU, it is inserted at the tail of the immediate higher level queue. For example, if a process voluntarily leaves CPU while it was in queue 2, it is inserted at the tail of queue 1.

  - If it consumes all the time slices, it is preempted and inserted at the end of the next lower level queue. For example, if a process voluntarily consumes all time slices while it was in queue 1, it is preempted and inserted at the tail of queue 2.

- After a certain time interval, all the processes are brought to the topmost queue, which is known as `priority boosting`.

2.2 Lottery Scheduling

The basic idea behind lottery scheduling is quite simple:

- Each process is assigned a fixed (integer) number of tickets.
- A process is probabilistically assigned a time slice based on its number of tickets.
- More specifically, if there are n processes $p_1, p_2, \ldots, p_n$ and they have tickets $t_1, t_2, \ldots, t_n$ respectively at any time, then the probability of a process p_i being scheduled at that time is $\frac{t_i}{\sum_{j=1}^{n} t_j}$.
- Basically, you need to sample based on the probability distribution derived from the ticket counts.
- After each time slice, t_i will be reduced by 1 (i.e., the process has used that ticket).
- Hence, the probabilities need to be recomputed each time using the updated ticket counts.
- All the processes are reinitialized with their original ticket count once the ticket counts of all runnable processes become 0.

2.3 Specifications

For this particular task, we will initially assume that there is only one CPU. This can be implemented by setting `CPUS = 1` in the `Makefile` of xv6.

As we have only two queues:

- The top one (queue 1) will implement lottery scheduling.
- The bottom one (queue 2) will implement the Round-Robin algorithm.

The time limits of the queues are defined using macros:

- `#define TIME_LIMIT_1 1` - one time slice in the top queue
- `#define TIME_LIMIT_2 2` - two **subsequent** time slices in the bottom queue
- `#define BOOST_INTERVAL 64` - priority boosting interval

All the macros should be defined in `kernel/param.h`.

You may assume that the total number of processes will be small enough so that you can check each of them for their queue number instead of implementing the queues rigorously.

2.4 Implementation Details

2.4.1 Scheduling Algorithm

You need to implement the scheduling algorithm mostly in the `kernel/proc.c` file. Try to understand the default Round-Robin algorithm implemented inside the `scheduler()` function. It basically loops over all the active processes and runs the first one which is in a runnable state (states of processes are defined in `kernel/proc.h`) in a Round-Robin format. Watching the two videos Context Switching and Scheduling + swtch.S may come handy for understanding the flow of scheduling in xv6.

For the assignment, you may add additional variables in `struct proc` defined in `kernel/proc.h` to store necessary information like:

- Which queue the process is currently in
- Its original ticket count
- Current ticket count
- Consumed time slots (both total and in current turn)
- Other necessary tracking variables

When a process is created, these variables should be initialized and then updated when needed.

Inside the `scheduler()` function, you need to schedule the processes according to the specifications. After a process returns, check its consumed time slices and change its queue accordingly. To find the consumed time slices, you may update all the running processes after each clock interrupt and do necessary checking for `boosting criteria`. The `clockintr()` function defined in `kernel/trap.c` might be useful.

2.4.2 System Calls

You will need two system calls for your implementation:

settickets The first system call is `int settickets(int number)`, which sets the number of tickets of the calling process. By default, each process should get a default number of tickets; calling this routine makes it such that a process can change the number of tickets it receives and thus receives a different proportion of CPU cycles. This routine should return `0` if successful and `-1` otherwise (if, for example, the caller passes in a number less than one, in which case the default number of tickets are allocated). The default number of tickets will be equal to `DEFAULT_TICKET_COUNT` macro (keep its value as 10) defined in `kernel/param.h`.

getpinfo The second system call is `int getpinfo(struct pstat *)`. This routine returns some information about all active processes, including:

- PIDs
- Statuses
- Which queue each one is currently in
- How many time slices each has been scheduled to run

- Other relevant information

You can use this system call to build a variant of the Linux command line program `ps`, which can then be called to see what is going on.

The structure `pstat` is defined below; note that you cannot change this structure and must use it exactly as is. When the `getpinfo()` routine is called, the `pstat` structure should be updated with the necessary values. The routine should return `0` if successful and `-1` otherwise (if, for example, a bad or NULL pointer is passed into the kernel).

You will need to understand how to fill in the structure `pstat` in the kernel and pass the results to the userspace (like you will do with the `history` system call). The structure should look like what you see below. You need to add a file named `pstat.h` to the xv6 kernel directory.

```c
#ifndef _PSTAT_H_
#define _PSTAT_H_
#include "param.h"
struct pstat {
int pid[NPROC]; // the process ID of each process
int inuse[NPROC]; // whether this slot of the process table is being used (1 or 0)
int inQ[NPROC]; // which queue the process is currently in
int tickets_original[NPROC]; // the number of tickets each process originally had
int tickets_current[NPROC]; // the number of tickets each process currently has
int time_slices[NPROC]; // the number of time slices each process has been
    scheduled
};
#endif // _PSTAT_H_
```

2.4.3 Files

Most of the code for the scheduler is quite localized and can be found in `proc.c`; the associated header file, `proc.h`, is also quite useful to examine. To change the scheduler, not much needs to be done; study its control flow (maybe from the xv6 book or the playlist) and then try some small changes.

2.4.4 Ticket Assignment

You will need to assign tickets to a process when it is created. Specifically, you will need to ensure that a **child inherits the same number of tickets as its parent**. Thus, if the parent originally had 17 tickets and calls `fork()` to create a child process, the child should also get 17 tickets initially.

2.4.5 Argument Passing

Good examples of how to pass arguments into the kernel are found in existing system calls. In particular, follow the path of `read()`, which will lead you to `sys_read()`, which will show you how to use `argaddr()` (and related calls) to obtain a pointer that has been passed into the kernel. Note how careful the kernel is with pointers passed from the userspace – they are a security threat(!) and thus must be checked very carefully before usage.

2.4.6 Random Number Generation

You will also need to figure out how to generate (pseudo)random numbers in the kernel; you can implement your own random number generator or use any off-the-shelf implementation from the web. **Include the link of the article or website that you take your random number**

generator from. You must make sure that the random number generator uses a deterministic seed (so that the results will be reproducible) and is implemented as a kernel-level module.

2.5 Testing

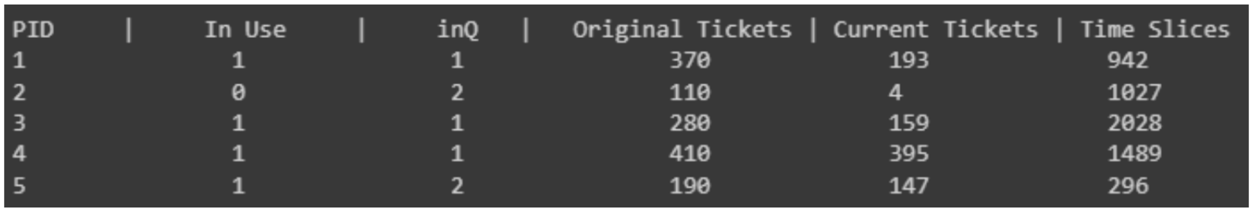
2.5.1 User level programs

You need to write two user-level programs to test your implementation:

dummyproc.c This program should be used to test the ticket assignment and scheduling behavior. **It should have provisions to test forked processes**. Its calling syntax should be like `dummyproc 43` where the parameter is used to set the number of tickets. If nothing is passed as argument, system call `settickets` needs to be made with `-1`.

testprocinfo.c This program should update the `pstat` structure and then print its contents in a nice format. Its calling syntax should be `testprocinfo` as it should not need any parameter. All statistics information must be printed from **user mode**. Printing anything in kernel mode is not allowed.

Executing multiple instances of `dummyproc.c` (by appending `&;` after each call) followed by a single instance of `testprocinfo.c` should print the relevant statistics defined in the `pstat` structure like a format shown in Figure 1.



```
PID     |      In Use     |      inQ   |   Original Tickets | Current Tickets | Time Slices
1                1                1              370              193            942
2                0                2              110              4              1027
3                1                1              280              159            2028
4                1                1              410              395            1489
5                1                2              190              147            296
```

Figure 1: Demo format of output when `testprocinfo` command is executed

2.5.2 Guidelines for Testing the Scheduling Algorithm

To verify the correctness of your scheduling algorithm, your implementation should include a mechanism for producing diagnostic output at the kernel level. This can be achieved by defining a compile-time flag using a preprocessor macro, which enables conditional printing for debugging and evaluation purposes. An example is shown below:

```
#define PRINT_SCHEDULING 1
...
...

if (PRINT_SCHEDULING)
    printf("Scheduling output: ...");
```

Prior to submission, ensure that the flag is set to `0`. During evaluation, we will modify this flag as needed to inspect the internal behavior of your scheduling logic.

3 Bonus Tasks

Multi-CPU Support in Scheduling Modify the implementation to ensure that the Multi-Level Feedback Queue (MLFQ) scheduler functions correctly in a multi-CPU environment. It is important to note that support for multiple CPUs in the `history` system call is a mandatory requirement, not a bonus task.

Hint: Consider the use of appropriate locking mechanisms in sections of the code where concurrent access by multiple CPUs could lead to race conditions or inconsistent state (protect the queue).

4 General Guidelines

- Don't forget to acquire and release locks when needed. Look out for the `proc` struct in `kernel/proc.h` and `kmem` struct in `kernel/kalloc.c`. You should look at how other existing functions use the fields of those structs to get an idea.
- Remember xv6 is multi-core, so proper synchronization is essential.
- Make incremental changes and test them thoroughly to avoid debugging complex issues.
- Read the xv6 book to better understand the internals and the existing implementation.

5 Submission Guidelines

Start with a fresh copy of xv6 from the original repository. Make necessary changes for this assignment. In this offline, you will submit just the changes done (i.e., a patch file), not the entire repository.

Don't commit. Modify and create files that you need to. Then create a patch using the following command:

```
git add --all
git diff HEAD > {studentID}.patch
```

Where `studentID` = your own seven-digit student ID (e.g., 2105192). Just submit the patch file, do not zip it.

In the lab, during evaluation, we will start with a fresh copy of xv6 and apply your patch using the command:

```
git apply {studentID}.patch
```

Make sure to test your patch file after submission the same way we will run it during the evaluation.

Please DO NOT COPY solutions from anywhere (your friends, seniors, internet, etc.). Any form of plagiarism (irrespective of source or destination), will result in getting -100% marks in this assignment. You have to protect your code.

Submission Deadline

Friday, May 30, 2025, 11:45 PM

Mark Distribution

Task	**Sub-task**	**Marks**
History System Call	Designing history.c	5
	Counting system calls	8
	Calculating system call time	7
	Using appropriate fine-grained locking	10
MLFQ Scheduler	Random number generation	5
	Argument passing	10
	Implementing lottery scheduling	20
	Implementing MLFQ	20
	Testing programs	10
Proper submission		5
Bonus tasks	Scheduling Multi-CPU support	10
Total		**100+10**