

# Project 4

---

**Due** Mar 6 by 11:59pm    **Points** 17    **Submitting** a file upload  
**Available** Feb 22 at 12am - Mar 6 at 11:59pm

---

This assignment was locked Mar 6 at 11:59pm.

## CS537 Spring 2023, Project 4


### Updates

- Updated the pstat structure to contain strides[NPROC] and pass[NPROC]

### Administrivia

- **Due Date** by March 6th, 2023 at 11:59 PM
- Questions: We will be using Piazza for all questions.
- Collaboration: The assignment has to be done by yourself. Copying code (from others) is considered cheating. [Read this \(http://pages.cs.wisc.edu/~remzi/Courses/537/Spring2018/dontcheat.html\)](http://pages.cs.wisc.edu/~remzi/Courses/537/Spring2018/dontcheat.html) for more info on what is OK and what is not. Please help us all have a good semester by not doing this.
- We'll be implementing a new scheduling algorithm for the xv6 scheduler in this project.
- Some tests will be provided at `~cs537-1/tests/P4`. Read more about the tests, including how to run them, by executing the command `cat ~cs537-1/tests/P4/README` on any lab machine. Note these test cases are not complete, and you are encouraged to create more on your own.
- Handing it in: Compress your modified version of xv6 using `tar -czvf tar-file-name.tar.gz path-to-your-xv6-dir/` and copy it to `~cs537-1/handin/login/P4` where login is your CS login.

## Stride Scheduling for xv6

In this project, you'll be putting a new scheduler into xv6. It is called a **stride scheduler**, and the algorithm is described in Figure 1 in [this paper](https://web.eecs.umich.edu/~mosharaf/Readings/Stride.pdf)  (<https://web.eecs.umich.edu/~mosharaf/Readings/Stride.pdf>).

The objectives for this project:

- To gain further knowledge of a real kernel, xv6.
- To familiarize yourself with a scheduler.
- To change that scheduler to implement a new algorithm - **stride scheduling**
- To make a graph to show your project behaves appropriately.

## Stride Scheduling

The basic idea in stride scheduling is the following:

- Each process is assigned tickets, and the stride is inversely proportional to the number of tickets assigned, specifically calculated as  $\text{stride} = \text{max\_stride} / \text{tickets}$ , where `max_stride` is the maximum stride, which is a constant number.
- Each process has a `pass` value, which starts from the stride and is incremented by the stride every time the process executes for a time slot
- The scheduler schedules a runnable process with the **minimum pass value** to run in the next time slot.

For example:

- Max stride = 12, tickets for processes: A: 3, B: 2:  
 $\text{stride}(A) = 12/3 = 4$ ,  $\text{stride}(B) = 12/2 = 6$
- Initial values for  $\text{pass}(A) = 4$ ,  $\text{pass}(B) = 6$
- Process A has a lower pass value, so it will be first chosen for scheduling. After process A runs for 1 time slot, the pass values will be  
 $\text{pass}(A) = 4 \text{ (initial)} + 4 \text{ (stride}(A)) = 8$   $\text{pass}(B) = 6$
- Now process B has the lower pass value and will be chosen for scheduling in the next time slot and so on.
- After a while  
 If process A executes 5 time slots,  $\text{pass}(A) = 4 \text{ (initial)} + 4 * 5 \text{ (stride}(A) * \text{slots}) = 24$   
 If process B executes 5 time slots,  $\text{pass}(B) = 6 \text{ (initial)} + 6 * 5 \text{ (stride}(A) * \text{slots}) = 36$

## Implementation/setup details

- Set `CPU := 1` in the Makefile to see the effect of scheduling on a single CPU
- We are not concerned about the complexity of retrieving the process with the minimum pass in each scheduler decision. So, you are free to use any data structure of your choice without worrying about any performance implications.
- The `max_stride` value should be greater than max tickets allocated to any process and preferably divisible by ticket values for getting more accurate strides (assuming an integer `max_stride`)
- If the number of tickets is changed while a process is executing, we update the tickets, stride, but not the pass value. So, the updated tickets will not reflect immediately in actual scheduling due to the existing pass values.
- We will not test for a dynamic allocation of tickets, since the algorithm is meant to handle a static allocation

## Details

You'll need two new system calls to implement this scheduler. The first is `int settickets(int number)`, which sets the number of tickets of the calling process. By default, each process should get one ticket; calling this routine makes it such that a process can raise the number of tickets it receives, and thus receive a higher 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).

The second is `int getpinfo(struct pstat *)`. This routine (detailed below) returns information about all running processes, including how many times each process has been chosen to run and the process ID of each process. You can use this system call to build a variant of the command line program `ps`, which can then be called to see what is going on. The structure `pstat` is defined below; note, you cannot change this structure, and must use it exactly as is. This routine should return 0 if successful, and -1 otherwise (if, for example, a bad or NULL pointer is passed into the kernel).

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 and then try some small changes.

Finally, you'll need to understand how to fill in the structure `pstat` in the kernel and pass the results to user space. The structure should look like what you see here, in a file you'll have to include called `pstat.h` (as `include/pstat.h`):

```
#ifndef _PSTAT_H_
#define _PSTAT_H_

#include "param.h"

struct pstat {
    int inuse[NPROC]; // whether this slot of the process table is in use (1 or 0)
    int tickets[NPROC]; // the number of tickets this process has
    int strides[NPROC]; // the stride of each process
    int pass[NPROC]; // the current pass value of each process
    int pid[NPROC]; // the PID of each process
    int ticks[NPROC]; // the number of ticks each process has accumulated
};

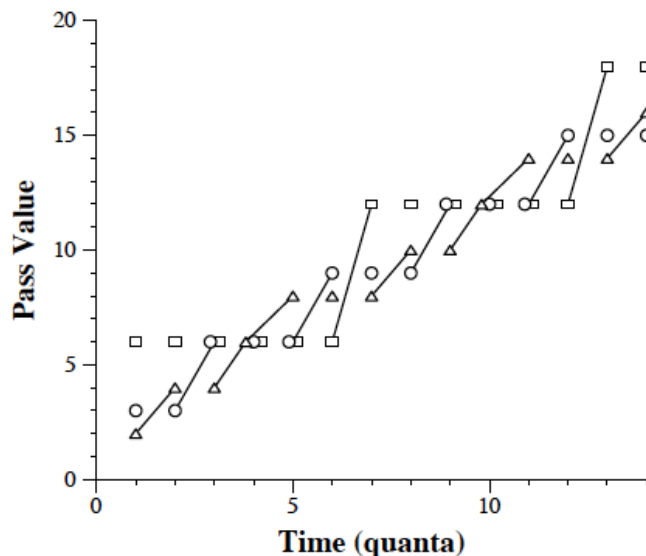
#endif // _PSTAT_H_
```

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 `argptr()` (and related calls) to obtain a pointer that has been passed into the kernel. Note how careful the kernel is with pointers passed from user space -- they are a security threat(!), and thus must be checked very carefully before usage.

You'll need to assign tickets to a process when it is created. Specifically, A child process *inherits* the same number of tickets as its parents. Thus, if the parent has 10 tickets and has a stride of 2, and calls `fork()` to create a child process, the child should also get 10 tickets, will have stride 2, and its pass value starts from its **stride** value (2). The first process - **init** starts with 1 ticket, and stride of `max_stride`, and pass value of `max_stride`. The pass value of a process is never reset for the purposes of this project.

## Graph

Beyond the usual code, you'll have to make a graph for this assignment similar to Figure 2 from [this paper](https://web.eecs.umich.edu/~mosharaf/Readings/Stride.pdf) , as shown here:



but for 2 processes. The graph should show the pass value of two processes over 10 time slots, where the processes have a 3:2 ratio of tickets (e.g., process A might have 3 tickets, and process B 2). Also, please specify the max\_stride and tickets assigned to each process in the graph.

Using GDB for this part can be useful. The graph can be generated using plotting software or be hand-drawn clearly showing the point coordinates. Submit a screenshot/image of the graph to Canvas.