# Project 2

**Due** Feb 8 by 11:59pm **Points** 11 **Submitting** a file upload

# CS537 Spring 2023, Project 2

## Updates

- Hint page link fixed
- getnextpid takes no arguments
- getprocstate converts states to strings using the same format as in procdump

## Administrivia

- **Due Date** by February 8th, 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/Classes/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 doing kernel hacking projects in xv6, a port of a classic version of Unix to a modern processor, Intel's x86. It is a clean and small kernel. More information is available **here (https://pdos.csail.mit.edu/6.828/2012/xv6.html)**.
- Some tests will be provided at *~cs537-1/tests/P2*. Read more about the tests, including how to run them, by executing the command `cat ~cs537-1/tests/P2/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/P2* where login is your CS login.

## Kernel hacking with xv6

In this assignment, you will add 2 system calls to a version of xv6.

This project is intended to be a warmup for xv6, and is thus relatively light! You do not need to write many lines of code; instead, a lot of your time will be spent learning where different routines are located in the existing xv6 source code.

Learning Objectives:

- Gain comfort looking through more substantial code bases written by others in which you do not need to understand every line
- Obtain familiarity with the xv6 code base in particular

- Learn how to add a system call to xv6
- Add a user-level application that can be used within xv6
- Become familiar with a few of the data structures in xv6 (e.g., process table)
- Use the gdb debugger on xv6

Summary of what gets turned in:

- A screenshot demonstrating your familarity with `QEMU` and `gdb` to Canvas.
- Your modified version xv6
- The xv6 system should compile successfully with your modified Makefile
- Include a single **README.md** (http://README.html) describing the implementation. This file should include your name, your cs login, you wisc ID and email, and the status of your implementation. It it all works then just say that. If there are things you know doesn't work let us know.

## xv6 System Calls

You will be using our version of xv6. Copy the xv6 source code using the following commands:

```
prompt> cp ~cs537-1/public/xv6.tar.gz .
prompt> tar -xvf xv6.tar.gz
```

If, for development and testing, you would like to run xv6 in an environment other than the CSL instructional Linux cluster, you may need to set up additional software. It is derived from Unix version 6, documented **here** (https://warsus.github.io/lions-/) . You can read **these instructions for the MacOS build environment** (https://github.com/remzi-arpacidusseau/ostep-projects/blob/master/INSTALL-xv6.md) . Note that we will run all of our tests and do our grading on the instructional Linux cluster so you should always ensure that the final code you handin works on those machines.

After you have obtained the source files, you can run `make qemu-nox` to compile all the code and run it using the QEMU emulator. Test out the unmodified code by running a few of the existing user-level applications, like `ls` and `forktest`. With `ls` inside the emulator, you'll be able to see a few other applications that are available (as well as files that have been created within the xv6 environment).

To quit the emulator, type `Ctl-a x`. (You press Ctrl+A at the same time, let go of the keys, and then press the X key.)

You will want to become familiar with the Makefile and comfortable modifying it. In particular, see the list of existing `UPROGS`. See the different ways of running the environment through make (e.g., `qemu-nox` or `qemu-nox-gdb`).

Find where the number of `CPUS` is set and change this to be 1.

For additional information about xv6, we strongly encourage you to look through the code while reading **this book** (https://pdos.csail.mit.edu/6.828/2018/xv6/book-rev11.pdf) by the xv6 authors.

Or, if you prefer to watch videos, the last ten minutes of the **first video (https://www.youtube.com/watch?v=5H5esXbVkC8)** plus a 2nd video from a previous **discussion section (https://www.youtube.com/watch?v=vR6z2QGcoo8)**

**(https://www.youtube.com/watch?v=vR6z2QGcoo8)**
describes some of the relevant files. Note that the code and project in the videos do not exactly match what you are doing. We always recommend that you look at the actual code yourself while either reading or watching (perhaps pausing the video as needed).

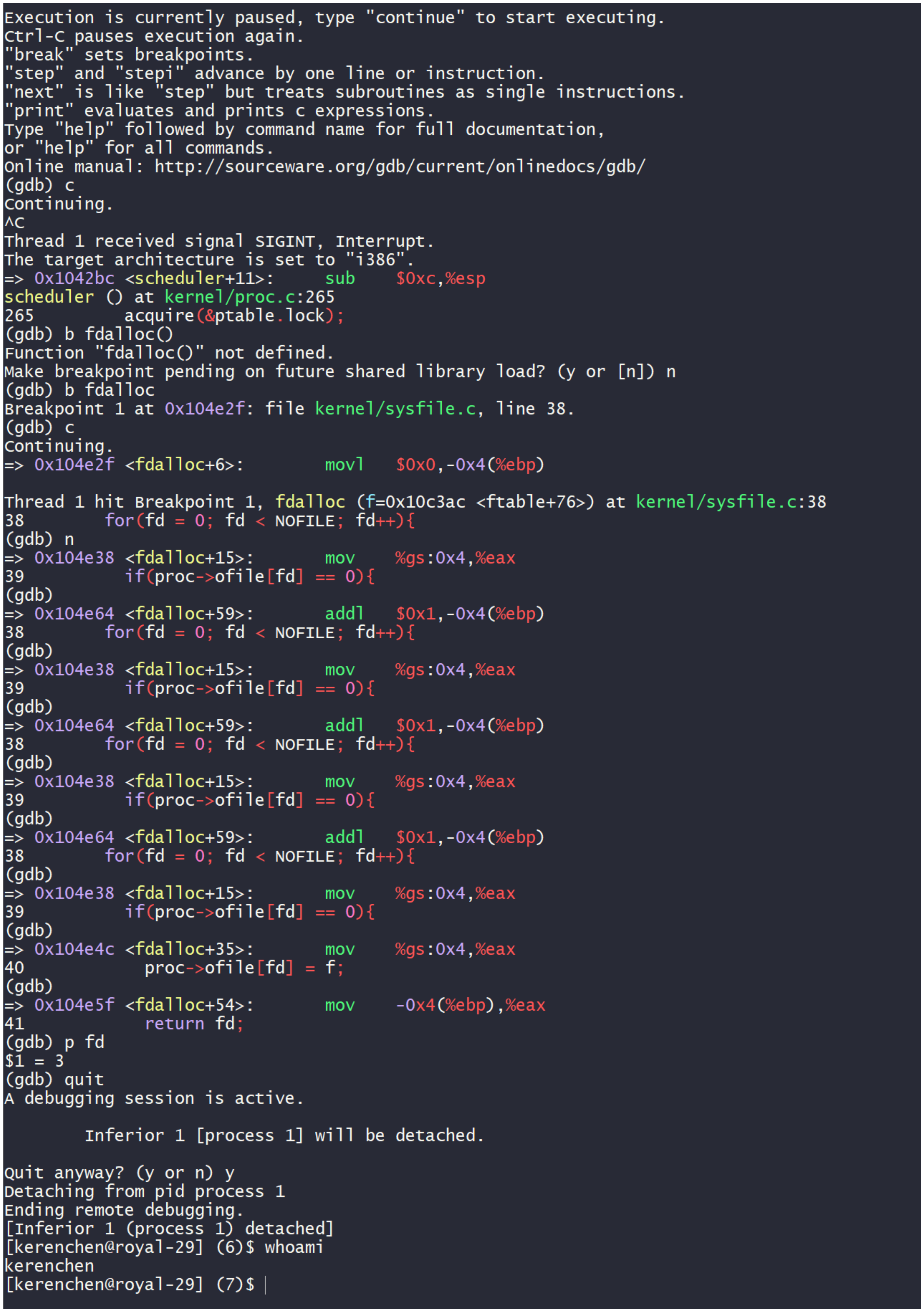
## Debugging

Your first task is to demonstrate that you can use **gdb** to debug xv6 code. You should show the integer value that `fdalloc()` (in `sysfile.c`) returns the first time it is called after a process has been completely initialized.

To do this, you can follow these steps:

1. In one window, start up `qemu-nox-gdb` (using make). This will automatically create a `.gdbinit` file under your current working directory.
2. Open another window **on the same machine** and go to the same working directory. Traditionally gdb will automatically load the `.gdbinit` file under the current working directory and debug xv6 in QEMU. However, in recent versions, this feature is removed for security reasons (e.g. you do not want to load `.gdbinit` file left by someone else). You have two options to manually enable this:
3. `echo "add-auto-load-safe-path $(pwd)/.gdbinit" >> ~/.gdbinit`. This enables the autoloading of the .gdbinit in the current working directory.
4. `echo "set auto-load safe-path /" >> ~/.gdbinit`. This enables the autoloading of all `.gdbinit`.
5. Start up gdb and `continue` it until xv6 finishes its bootup process and gives you a prompt.
6. Now, interrupt (Ctrl+C) gdb and set a breakpoint in the `fdalloc()` routine.
7. Continue gdb, and run the `stressfs` user application at the xv6 prompt since this will cause `fdalloc()` to be called. Your gdb process should now have stopped in `fdalloc`.
8. `step` (or, probably `next`) through the C code until gdb reaches the point just before `fdalloc()` returns and print the value that will be returned (i.e., the value of `fd`).
9. Immediately quit gdb and run `whoami` to display your login name.
10. If gdb gives you the error message that fd has been optimized out and cannot be displayed, make sure that your Makefile uses the flag "-Og" instead of "-O2". Debugging is also a lot easier with a single CPU, so if you didn't do this already: in your Makefile find where the number of `CPUS` is set and change this to be 1.

Take a screenshot like the one below showing your gdb session with the returned value of fd printed and your login name displayed. Submit this screenshot to Canvas.



```
Execution is currently paused, type "continue" to start executing.
Ctrl-C pauses execution again.
"break" sets breakpoints.
"step" and "stepi" advance by one line or instruction.
"next" is like "step" but treats subroutines as single instructions.
"print" evaluates and prints c expressions.
Type "help" followed by command name for full documentation,
or "help" for all commands.
Online manual: http://sourceware.org/gdb/current/onlinedocs/gdb/
(gdb) c
Continuing.
^C
Thread 1 received signal SIGINT, Interrupt.
The target architecture is set to "i386".
=> 0x1042bc <scheduler+11>:     sub    $0xc,%esp
scheduler () at kernel/proc.c:265
265           acquire(&ptable.lock);
(gdb) b fdalloc()
Function "fdalloc()" not defined.
Make breakpoint pending on future shared library load? (y or [n]) n
(gdb) b fdalloc
Breakpoint 1 at 0x104e2f: file kernel/sysfile.c, line 38.
(gdb) c
Continuing.
=> 0x104e2f <fdalloc+6>:        movl   $0x0,-0x4(%ebp)

Thread 1 hit Breakpoint 1, fdalloc (f=0x10c3ac <ftable+76>) at kernel/sysfile.c:38
38          for(fd = 0; fd < NOFILE; fd++){
(gdb) n
=> 0x104e38 <fdalloc+15>:       mov    %gs:0x4,%eax
39            if(proc->ofile[fd] == 0){
(gdb)
=> 0x104e64 <fdalloc+59>:       addl   $0x1,-0x4(%ebp)
38          for(fd = 0; fd < NOFILE; fd++){
(gdb)
=> 0x104e38 <fdalloc+15>:       mov    %gs:0x4,%eax
39            if(proc->ofile[fd] == 0){
(gdb)
=> 0x104e64 <fdalloc+59>:       addl   $0x1,-0x4(%ebp)
38          for(fd = 0; fd < NOFILE; fd++){
(gdb)
=> 0x104e38 <fdalloc+15>:       mov    %gs:0x4,%eax
39            if(proc->ofile[fd] == 0){
(gdb)
=> 0x104e64 <fdalloc+59>:       addl   $0x1,-0x4(%ebp)
38          for(fd = 0; fd < NOFILE; fd++){
(gdb)
=> 0x104e38 <fdalloc+15>:       mov    %gs:0x4,%eax
39            if(proc->ofile[fd] == 0){
(gdb)
=> 0x104e4c <fdalloc+35>:       mov    %gs:0x4,%eax
40              proc->ofile[fd] = f;
(gdb)
=> 0x104e5f <fdalloc+54>:       mov    -0x4(%ebp),%eax
41              return fd;
(gdb) p fd
$1 = 3
(gdb) quit
A debugging session is active.

        Inferior 1 [process 1] will be detached.

Quit anyway? (y or n) y
Detaching from pid process 1
Ending remote debugging.
[Inferior 1 (process 1) detached]
[kerenchen@royal-29] (6)$ whoami
kerenchen
[kerenchen@royal-29] (7)$ 
```

## System Calls Details

You will next add 2 new system calls to xv6. For hints on how to how to add a system call, read the **hints document (https://canvas.wisc.edu/courses/330415/pages/p2-hints-dot-html)** .

```
int getnextpid()
```

When called, `getnextpid` returns the next available PID, which is simply `nextpid` in `proc.c`.

```
int getprocstate(int pid, char* state, int n)
```

When called, `getprocstate` finds the state of the first process with that `pid` in `ptable`, and copies the process's `procstate` to the parameter passed into this system call `state`, which points to a block of memory of size `n` bytes; use the same corresponding strings as in `void procdump(void)`; if `n` is not enough to hold the state string, it should return -1, in which case the content of `state` is undefined. It should return 0 on success and -1 if the the process is not found.

In xv6, each process is represented as a `struct proc`, and there are a total of 6 possible states. The definitions of `procstate` and `struct proc` are defined in `proc.h`:

```
enum procstate { UNUSED, EMBRYO, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };

// Per-process state
struct proc {
  uint sz;                     // Size of process memory (bytes)
  pde_t* pgdir;                // Page table
  char *kstack;                // Bottom of kernel stack for this process
  enum procstate state;        // Process state
  volatile int pid;            // Process ID
  ...
};
```

The system keeps track of all processes with `ptable` in `proc.c`. You can check out `procdump` to learn more about how to locate a process in `ptable`.

You are not required to write a xv6 user space program in this assignment. However, we suggest you start with a simple program like hello-world before modifying the kernel to better understand and debug your code.

If you cannot find your user-level application inside xv6, one possible reason is that your application is not compiled into xv6. One easy way to check that is to write some invalid code and see if xv6 still compiles.

## Implementation Hints and Details

The primary files you will want to examine in detail include `syscall.c`, `proc.c`, `proc.h`, and `sysfile.c`. You might also want to take a look at `usys.S`, which (unfortunately) is written in assembly.

To add a system call, find some other very simple system call like `sys_fstat` that also takes a pointer parameter, copy it in all the ways you think are needed, and modify it so it doesn't do anything and has the new name. Compile the code to see if you found everything you need to copy and change. You probably won't find everything the first time you try.

Then think about the changes that you will need to make so your system call acts like it has to.

Good luck! While the xv6 code base might seem intimidating at first, you only need to understand very small portions of it for this project. This project is very doable!