CS 1550
Lab 1 – xv6 Introduction
Setup and exercise
CS 1550 – Kernel Space vs User Space

- OS manages hardware, services and user processes
  - CPU
  - Memory (Address space)
  - I/O devices (Disk, mouse, video card, sound, network, etc.)
CS 1550 – Kernel Space vs User Space

- OS manages hardware, services and user processes
  - CPU
  - Memory (Address space)
  - I/O devices (Disk, mouse, video card, sound, network, etc.)
CS 1550 – Kernel Space vs User Space

- OS is just another software
• OS is just another software
• User applications should not change the kernel (OS software)
User space
  - Less privileged memory space where user processes execute

Kernel space
  - Privileged memory space where the OS main process resides
  - No User application should be able to change
CS 1550 – Kernel Space vs User Space

- **System Call**
  - User processes have to do system calls to access the OS resources and Hardware
CS 1550 – Kernel Space vs User Space

• **System Call**
  • User processes have to do system calls to access the OS resources and Hardware
CS 1550 – Kernel Space vs User Space

• System Call
  • User processes have to do system calls to access the OS resources and Hardware
• **System Call** (OS function)
  • User processes have to do system calls to access the OS resources and Hardware
System Call
-
exercise
CS 1550 – xv6

• Simple Unix-like teaching operating system from MIT
  • Provides basic services to running programs
CS 1550 – Unix is everywhere

- Most operating systems are based on Linux
CS 1550 – xv6

• Simple Unix-like teaching operating system from MIT
  • Has a **subset of traditional** system calls
    • **fork()** Create process
    • **exit()** Terminate current process
    • **wait()** Wait for a child process
    • **kill(pid)** Terminate process pid
    • **getpid()** Return current process’s id
    • **Sleep** for n time units
    • **Load** a file and execute it
    • **sleep(n)**
  • **exec(filename, *argv)**
  • **sbrk(n)**
• Compile and Run xv6 in a cs pitt server
  • Since it is an OS how can we run it?
• Compile and Run xv6 in a cs pitt server
CS 1550 – xv6

• Compile and Run xv6 in a cs pitt server
CS 1550 – xv6

• Compile and Run xv6 in a cs pitt server

[Diagram: xv6 arrow to PC hardware Emulator (Virtual Machine)]
CS 1550 – xv6

• Compile and Run xv6 in a cs pitt server

Diagram:
- xv6
- PC hardware Emulator (Virtual Machine)
- Remote server linux.cs.pitt.edu
• Compile and Run xv6 in a cs pitt server
CS 1550 – Compile and Run xv6

1. Extend disk Quota, if you have less then 500mb free space
   a) Log in to https://my.pitt.edu
   b) Click on "Profile" at the top of the screen
   c) Click on "Manage Your Account"
   d) Click on "Manage Email Quota"
   e) Click on "Increase My UNIX Quota"
1. Extend disk Quota, if you have less then 500mb free space
   a) Log in to https://my.pitt.edu
   b) Click on "Profile" at the top of the screen
   c) Click on "Manage Your Account"
   d) Click on "Manage Email Quota"
   e) Click on "Increase My UNIX Quota"
CS 1550 – xv6

• Log in to linux.cs.pitt.edu
  • `ssh user_name@linux.cs.pitt.edu`

• Use Terminal (MacOS/Ubuntu)
• Use Putty/Powershell (Windows)
CS 1550 – xv6

• Download the xv6 source code from github
  • git clone git://github.com/mit-pdos/xv6-public.git
CS 1550 – xv6

• Got into the cloned xv6 source code folder
  • `cd xv6-public`

• Compile and run the code with
  • `make qemu-nox`
(3) kernighan $ make qemu-nox
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,for
(process:128413): GLib-WARNING **: gmem.c:483: custom memory allocation
xv6...
cpu1: starting 1
cpu0: starting 0
sb: size 1000 nbblocks 941 ninodes 200 nlog 30 logstart 2 inodestart 3
init: starting sh

$
• Compile and run the code with
  • `make qemu-nox`

Compiles and run xv6 with qemu
CS 1550 – xv6

- Compile and Run xv6 in a cs.pitt server
CS 1550 – xv6

• Compile and Run xv6 in a cs pitt server
• Once in xv6 you can call `ls`
First we need to define our new call and its number at `syscall.h`.
• First we need to define our new call and its number at
  • syscall.h

• Add
  • #define SYS_getday 22
Next we need to map the new call in the array pointer of system calls
  • syscall.c

Add
  • extern int sys_getday(void);
  • [SYS_getday] sys_getday,
Then we need to implement the actual method.

In xv6 this is organized in two files.
- `sysfile.c` -> file related system calls
- `sysproc.c` -> all the other syscalls
• Then we need to implement the actual method

• In xv6 this is organized in two files.
  • sysfile.c -> file related system calls
  • sysproc.c -> all the other syscalls
Then we need to implement the actual method.

In xv6 this is organized in two files.
- `sysfile.c` -> file related system calls
- `sysproc.c` -> all the other syscalls
Then we need to implement the actual method

In xv6 this is organized in two files.
- sysfile.c -> file related system calls
- sysproc.c -> all the other syscalls

```c
int sys_getday(void)
{
    return 6;
}
```
Afterwards we define the interface for user programs to call

- Open usys.S

- Add
  - SYSCALL(getday)
CS 1550 – xv6 – Adding a custom Syscall

• Finally we open
  • user.h

• Add
  • int getday(void);

```c
struct stat;
struct rtime;

// system calls
int fork(void);
int exit(void) __attribute__((noreturn));
int wait(void);
int pipe(int*);
int write(int, void*, int);
int read(int, void*, int);
int close(int);
int kill(int);
int exec(char*, char**);
int open(char*, int);
int mknod(char*, short, short);
int unlink(char*);
int fstat(int_fd, struct stat*);
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int close(int);
```
Example user program

```c
#include "types.h"
#include "stat.h"
#include "user.h"

int main(void) {
    printf(1, "Today is %d\n", getday());
    exit();
}
```
• Adding an user program
  • Open Makefile

• Add
  • _todays_date\