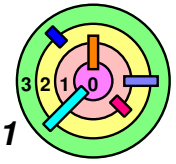
CS 350
PA1: Add System Calls To XV6

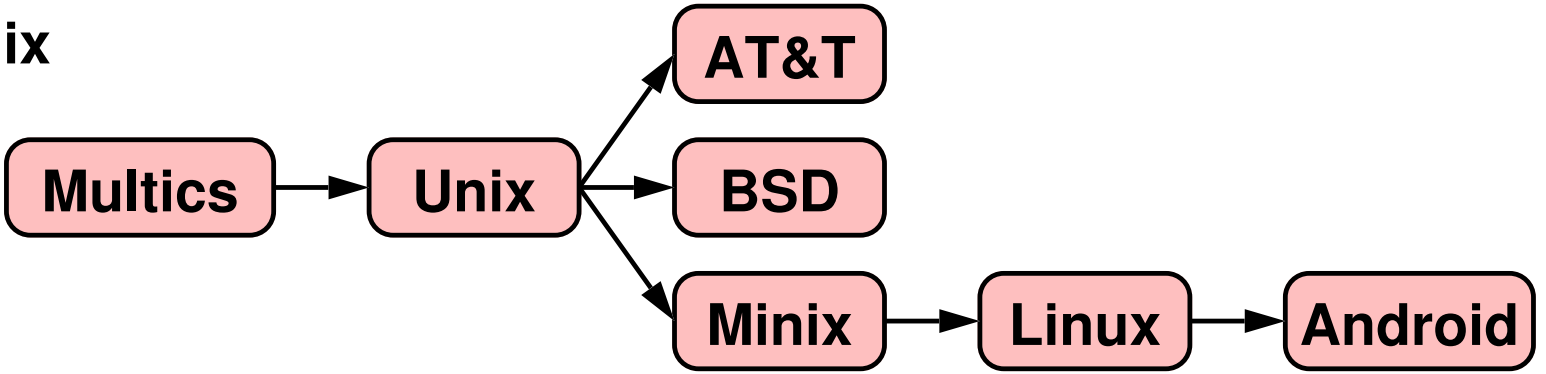
Bill Cheng

http://merlot.usc.edu/william/usc/

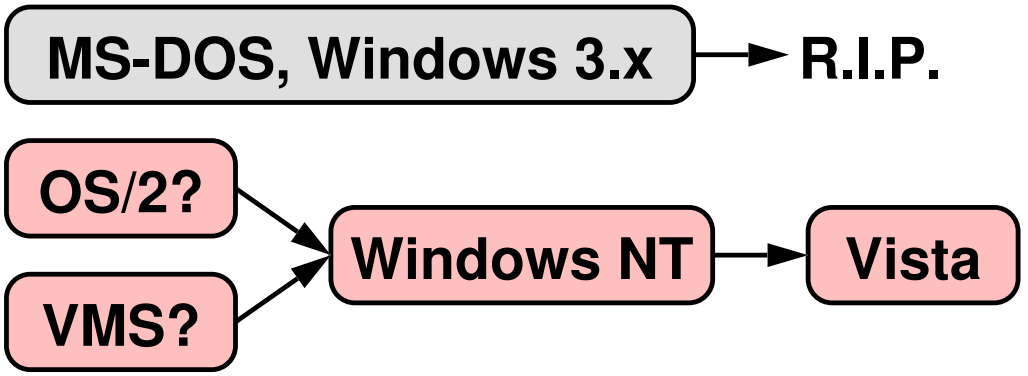


Is Unix/Linux Still Relevant Today?

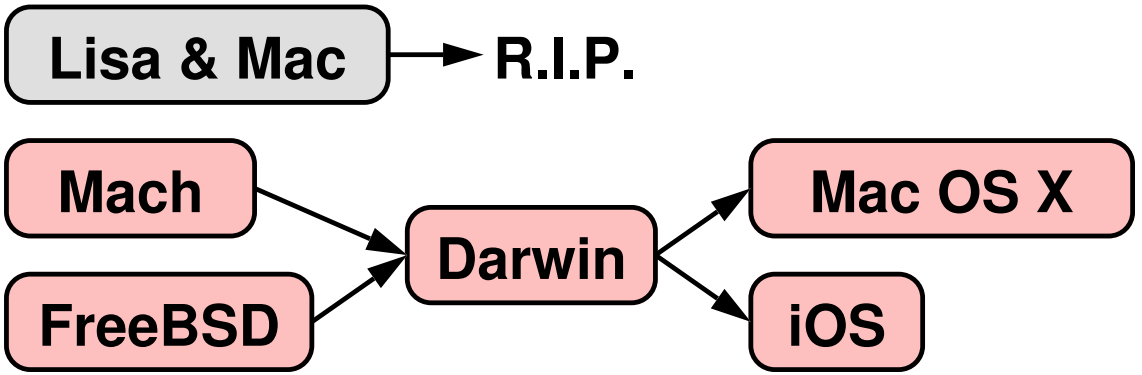
- Linux & Unix

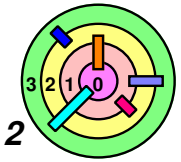


- Microsoft



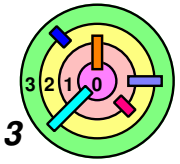
- Apple





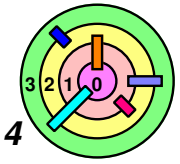
What Is C?

- Unix/Linux *system calls* have a C functional interface
  - must use a system call to use hardware
- Early 1960s: CPL (Combined Programming Language)
  - developed at Cambridge University and University of London
- 1966: BCPL (Basic CPL): simplified CPL
  - intended for systems programming
- 1969: B: simplified BCPL (stripped down so its compiler would run on minicomputer)
  - used to implement earliest Unix
- Early 1970s: C: expanded from B
  - motivation: they wanted to play "Space Travel" on minicomputer
  - used to implement all subsequent Unix OSes
- Unix has been written in C ever since



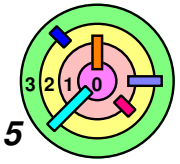
PA1

- Set up a standard 32-bit Ubuntu 16.04 system
  - download xv6 and get familiar with xv6
- Part 1 - add a new system call
  - `trace()`
- Part 2 - add a second system call
  - `date()`



Some Basic Linux Commands

```
ls
ls -a
ls -l
echo "hello"
echo -n "hello"
echo `date`
echo `date +%m%d%y-%H%M%S`
cat /etc/os-release
more /etc/os-release
mkdir tmp
pwd
cd tmp
pwd
ls
cd ..
cp /etc/os-release tmp
cp tmp/os-release tmp/abc
ls -aF tmp
mv tmp/os-release tmp/xyz
ls -aF tmp
diff tmp/abc tmp/xyz
man gcc
rm tmp/abc
touch tmp/defg
ls -alF tmp
ps -x
ps -auxw
pico tmp/xyz
rm tmp/defg tmp/xyz
ls -alF tmp
rmdir tmp
ls -alF tmp
df
top
exit
```

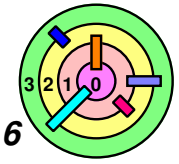


Notes On gdb

- The debugger is your friend! Get to know it *NOW*!

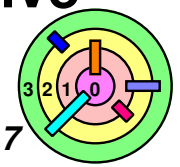
start debugging:	`gdb`
list source code:	`(gdb) list`
set breakpoint:	`(gdb) break foo.c:123`
list all breakpoints:	`(gdb) info breakpoints`
continue:	`(gdb) cont`
clear breakpoint:	`(gdb) clear`
stack trace:	`(gdb) where`
print field:	`(gdb) print f.BlockType`
print in hex:	`(gdb) print/x f.BlockType`
single-step at same level:	`(gdb) next`
single-step into a function:	`(gdb) step`
print field after every cmd:	`(gdb) display f.BlockType`
assignment:	`(gdb) set f.BlockType=0`
quit:	`(gdb) quit`

- Start using the debugger with PA1!
  - get help from TAs and me



Set Up A Standard System

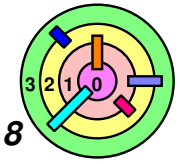
- Go to class home page, scroll all the way to the bottom and click on the *install a standard 32-bit Ubuntu 16.04 system* link
- You have two choices
  1) install a virtual machine hypervisor, then install my *virtual appliance* into the VM hypervisor to create a virtual machine
     - install *VirtualBox* if you have Intel/AMD CPU
     - install *UTM* if you have Apple CPU (M1/M2/M3)
  2) create an VM instance from *my AMI* on *AWS Free Tier* (free for one year if you don't go over the usage limit)
- if you like to do development on your host machine, you need to figure out a way to transfer files between your host machine and the "standard" system
  - my recommendation is to use FileZilla
  - some would like to use SSH in VScode
    - this would only work with (1) above and you need to give 4 GB RAM becuase VScode is memory hungry



Download XV6

- Follow the instructions on the PA1 spec
- Open a *terminal* and type the following

```
cat /etc/os-release
pwd
mkdir cs350
cd cs350
mkdir pa1
cd pa1
wget --user=USERNAME --password=PASSWORD \
  http://merlot.usc.edu/cs350-m25/programming/pa1/xv6-pa1-src.tar.gz
tar xvf xv6-pa1-src.tar.gz
cd xv6-pa1-src
ls
```



Run XV6

- Three ways to run xv6:
  1) run xv6 console in a separate window:

```
make qemu
```

  - I do not recommend this way

  2) run xv6 in commandline mode:

```
make qemu-nox
$ ls
$ echo Hello
$ cat README
```

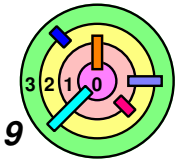
  3) debug xv6 commandline mode:

```
make qemu-nox-gdb
```

  - in a separate terminal, do:

```
gdb
(gdb) source .gdbinit
(gdb) list exec.c:11
(gdb) break exec
(gdb) break sys_open
(gdb) cont
```

  - by default, you are debugging the kernel



Debugging User Space Code In XV6

- The last line in .gdbinit says to debug the kernel

```
(gdb) symbol-file kernel
```

  - sometimes, you may need the assembly listing of the kernel

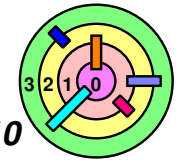
```
objdump --disassemble --section=".text" kernel > kernel.txt
objdump --disassemble --section=".text" -S kernel > kernel.txt
```

    - you can open `kernel.txt` with your favorite text editor
  - switch to debug the "`ls`" user space program

```
(gdb) symbol-file _ls
(gdb) list 25
(gdb) break ls
(gdb) break open
(gdb) cont
```

  - in the first window, when you get the xv6 prompt, type "`ls`" to run the "`ls`" program
    - you should break at the beginning of the `ls()` function

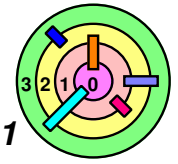
```
(gdb) c
Continuing.
```



Debugging User Space Code In XV6

- you should break at the `open()` functionsystem call
  - since `open()` is a ***system call***, things would look different

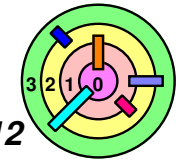
```
[  1b: 5c2]    0x772 <printf+162>:      in     (%dx),%al

Thread 1 hit Breakpoint 2, open () at usys.S:20
20      SYSCALL(open)
(gdb) list usys.S:20
15      SYSCALL(read)
16      SYSCALL(write)
17      SYSCALL(close)
18      SYSCALL(kill)
19      SYSCALL(exec)
20      SYSCALL(open)
...
(gdb) list usys.S:9
4       #define SYSCALL(name) \
5         .globl name; \
6         name: \
7           movl $SYS_ ## name, %eax; \
8           int $T_SYSCALL; \
9           ret
10
11      SYSCALL(fork)
...
```



Debugging User Space Code In XV6

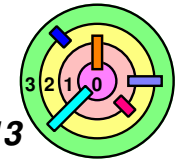
- An application program doesn't know how to open a file
  - why now?
  - only the OS kernel knows how to do that
  - to ask the OS kernel for help, you make a system call
  - in xv6, the convention is that if `foo()` is a *system call*, the corresponding OS kernel function is called `sys_foo()`
- To go from user space code to the kernel requires a *context switch*
  - there are different types of context switches
  - here we switch from the *user space context* to the *kernel space context*
    - we will talk more about this in class
    - for now, you need to switch to debug kernel code

```
(gdb) delete
(gdb) symbol-file kernel
(gdb) break sys_open
(gdb) cont
```



Debugging User Space Code In XV6

- The kernel versions of the system calls that are related to the file system is in `sysfile.c`
  - `sys_open()` looks like regular C code
    - but remember, you are now in the all power kernel, you can really mess things up if you are not careful
- Gdb is designed to mainly debug regular C code
  - it's not comfortable with switching contexts
    - single-step gdb command (i.e., "`next`" and "`step`") may not work as expected when a context switch is involved
    - if you know that a context switch will happen, you should set a breakpoint and use the "`cont`" gdb command to get there
  - although if you switch to assembly level debugging, then gdb will just be debugging machine code and it won't worry about context switching
    - why not?
    - context switching is just an ***abstraction***



Debugging User Space Code In XV6

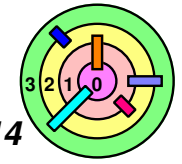
- How to get back to user space code?
  - you need to get back to where you made the `open()` system call
  - open `ls.c` and see that you called `open()` on line 33

```
(gdb) delete
(gdb) symbol-file _ls
(gdb) list ls.c:34
29          int fd;
30          struct dirent de;
31          struct stat st;
32
33          if((fd = open(path, 0)) < 0){
34            printf(2, "ls: cannot open %s\n", path);
35            return;
36          }
37
38          if(fstat(fd, &st) < 0){
```

  - therefore, you should do:

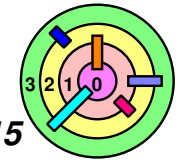
```
(gdb) break ls.c:34
(gdb) break ls.c:38
(gdb) cont
```

  - now you are back in user space



Advanced Debugging In XV6 (Optional)

- ***if you really want to know*** **how context switching works from user space to kernel space, you need to switch to debug assembly code (you probably have seen this in CS 356)**
  - ***"Abandon all hope, ye who enter here".***



Advanced Debugging In XV6 (Optional)

- ***if you really want to know*** **how context switching works from user space to kernel space, you need to switch to debug assembly code (you probably have seen this in CS 356)**
  - ***"Abandon all hope, ye who enter here".***

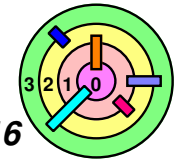
```
(gdb) layout asm
```

- **the window splits and the top panel would look like:**

```
   +-------------------------------------------
B+>|0x5c2 <open>              mov    $0xf,%ax
   |0x5c5 <open+3>            add    %al,(%bx,%si)
   |0x5c7 <open+5>            int    $0x40
   |0x5c9 <open+7>            ret
   |0x5ca <mknod>             mov    $0x11,%ax
   |0x5cd <mknod+3>           add    %al,(%bx,%si)
   |0x5cf <mknod+5>           int    $0x40
   |0x5d1 <mknod+7>           ret
   +-------------------------------------------
```

- **single step at the assembly code level:**

```
(gdb) si
(gdb) si
```



Advanced Debugging In XV6 (Optional)

- the top panel now looks like:

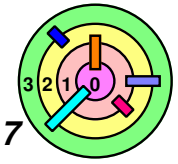
```
 +-------------------------------------------
>|0x80105cf9      push   $0x40
 |0x80105cfb      jmp    0x8010560a
 |0x80105d00      push   $0x0
 |0x80105cf9      push   $0x41
 |0x80105cfb      jmp    0x8010560a
 |0x80105d00      push   $0x0
 |0x80105cf9      push   $0x42
 |0x80105cfb      jmp    0x8010560a
 |0x80105d00      push   $0x0
 +-------------------------------------------
```

- they correspond to the following in "`usys.S`":

```
20      SYSCALL(open)
21      SYSCALL(mknod)
22      SYSCALL(unlink)
```

- set a breakpoint at virtual address `0x8010560a`

```
(gdb) break *0x8010560a
(gdb) cont
```



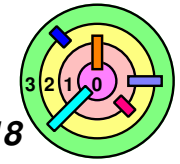
Advanced Debugging In XV6 (Optional)

- what's at `0x8010560a`?
- open `kernel.txt` and do a string search for `8010560a`

```
8010560a <alltraps>:
8010560a:       1e                      push   %ds
8010560b:       06                      push   %es
8010560c:       0f a0                   push   %fs
8010560e:       0f a8                   push   %gs
80105610:       60                      pusha
80105611:       66 b8 10 00             mov    $0x10,%ax
80105615:       8e d8                   mov    %eax,%ds
80105617:       8e c0                   mov    %eax,%es
80105619:       54                      push   %esp
8010561a:       e8 e1 00 00 00          call   80105700 <trap>
```

- in `kernel.txt`, `<trap>` looks code generated by a C compiler

```
80105700 <trap>:
80105700:       55                      push   %ebp
80105701:       89 e5                   mov    %esp,%ebp
...
```



Advanced Debugging In XV6 (Optional)

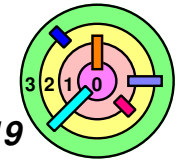
- to set a breakpoint there, you need to clear all breakpoints, switch to debug the kernel, and set a breakpoint there

```
(gdb) delete
(gdb) symbol-file kernel
(gdb) break trap
(gdb) layout src
(gdb) cont
```

- to get rid of the top panel:

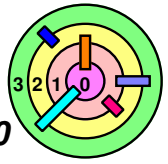
```
(gdb) tui disable
```

⇨ The assembly code level debugging is optional for now
  - hopefully, you won't need to do that in this class
  - one day, when you have a really tough bug and the only way to debug your code is to debug context switching at the assembly code level, then you need to come back here and review all this



Part 1: Add A System Call

- You need to read the xv6 book in the spec to understand how xv6 works
  - Ch 3 contains details on traps and system calls (although most of the low level details *are not needed* for you to complete this assignment)
- Your job is to add a new system call called `trace()`
  - since you know the basic flow from `open()` to `sys_open()` and back to `open()`, you should be able to add a `trace()` system call to reach `sys_trace()` and get back to `trace()`
    - of course, you need to implement `sys_trace()` according to the spec



Part 1: Add A System Call

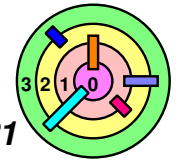
- Which files do you need to modify?
  - open a terminal and type the following:

```
pwd
cd cs350/pa1/xv6-pa1-src
make -n pa1-submit
```

    - you should see:

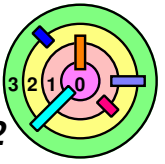
```
tar cvzf pa1-submit.tar.gz \
 Makefile \
 pa1-README.txt \
 proc.c \
 proc.h \
 syscall.c \
 syscall.h \
 sysproc.c \
 user.h \
 usys.S
```

  - these are the only files are are supposed to submit
    - if you submit additional files, the grader will have to delete them before grading
    - if you submit binary files, points will be deducted



Part 1: Add A System Call

- `test_project1.c` is a test program for part 1
  - need to modify `Makefile` get it compiled so the grader can run it
- Please take a look at the grading guidelines to see what the grader will do to grade part 1
  - `grade_pa1.c` is another test program for part 1
    - need to include that in your `Makefile`



Part 2: Add Another System Call

- Your job is to add a new system call called `date()`
  - need to call `cmostime()` to get read the real time clock (which is the current UTC time
- `date.c` is a test program for part 2
  - need to modify `Makefile` get it compiled so the grader can run it