Xv6
A Unix like teaching operating system.

We are here to help!

TA's for this exercise are

Χρήστος και Λυδία

Intro

- How will we get a good idea on how the operating system works?
- The people at MIT had an idea 
- The xv6 operating system.

It is a reimplementation of Unix v6, implemented for a modern x86–based multiprocessor using ANSI C.

The xv6 kernel

- What is the job of an operating system ?
- How is the job done ?
- An operating system provides services to user programs through an interface.
- Can you name one interface that you use in a Unix machine?

The process example

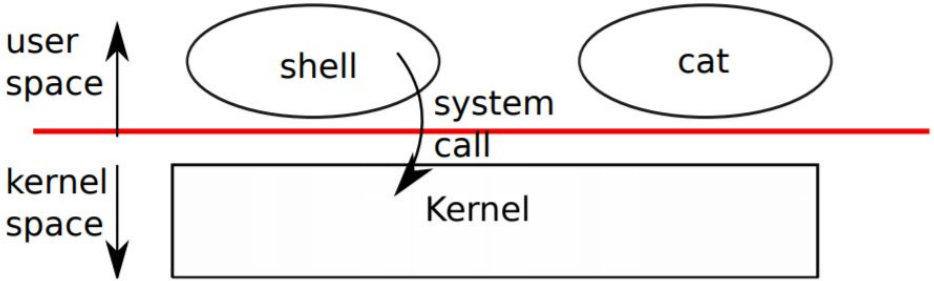


Figure 0-1. A kernel and two user processes.

What will we need?

The source files of xv6 is provided as a printed booklet with line numbers:

https://pdos.csail.mit.edu/6.828/2017/xv6/xv6-rev10.pdf

The main concepts of operating systems by studying the kernel are explained in a book:
https://pdos.csail.mit.edu/6.828/2017/xv6/book-rev10.pdf

Big concept

Understanding xv6 is a good start toward understanding modern operating systems.

2

Kernel Installation instructions

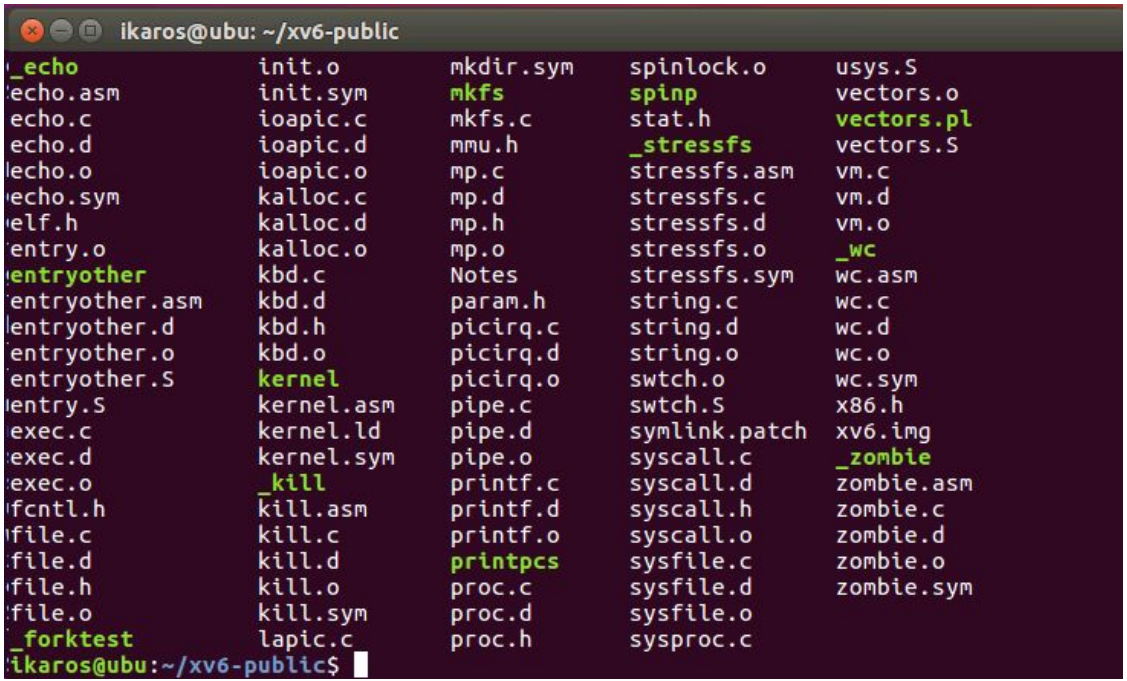
A brief guide on how to install xv6/QEMU

Step 1:install your new OS

- Git clone the repo from here
  https://github.com/mit-pdos/xv6-public

  git clone https://github.com/mit-pdos/xv6-public.git
- Anyone who is not familiar with **git** come find us later...
- Then we run cd xv6-public
- To build xv6 we run make.

We have a full working kernel!



Step 2: QEMU

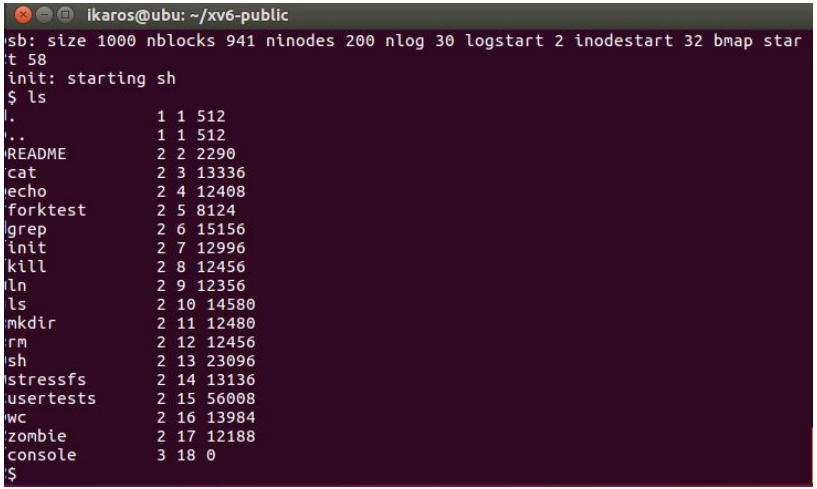
- QEMU is a modern and fast PC emulator available from https://www.qemu.org/index.html:

  You can install the emulator with the following command:

  sudo apt-get install qemu

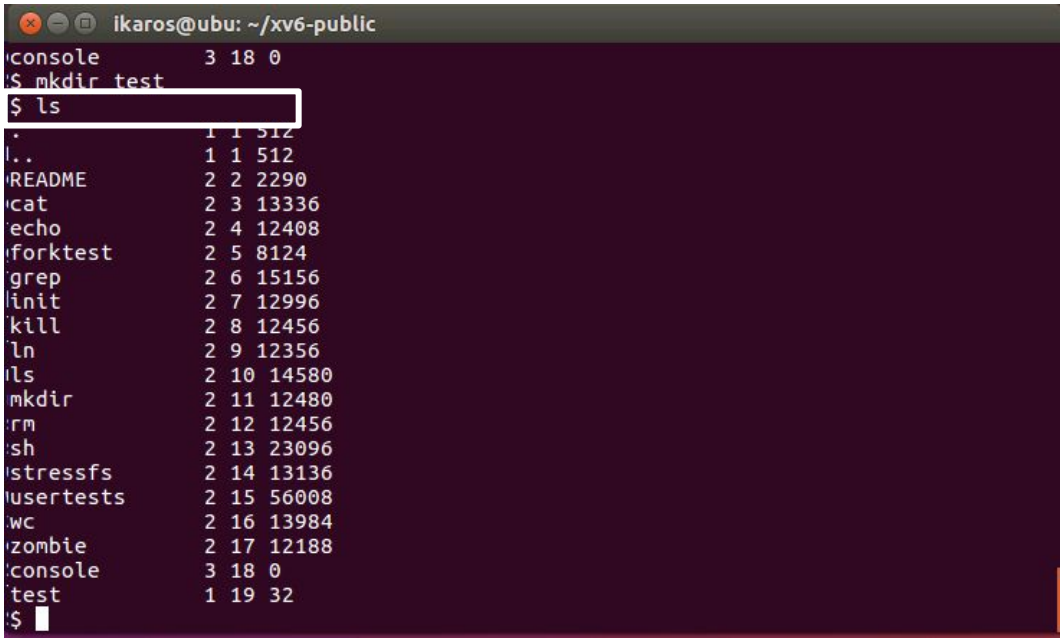
How do we use the kernel

- Now that we have installed xv6 and QEMU…
- Run make qemu-nox
- This is our operating system!
- Run ls



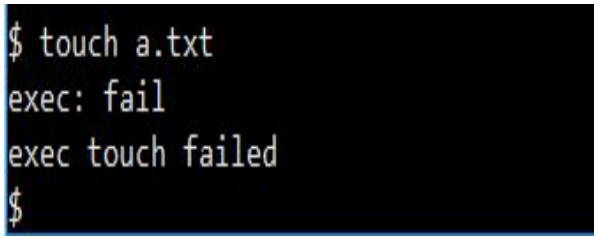
The xv6 kernel

- Some basic functionality is implemented on the kernel.



The xv6 kernel

- Some other are not implemented!!!



Big concept

The collection of system calls that a kernel provides is the ***interface*** that user programs see.

2

Writing our first program

A test program we are going to run inside the kernel

Creating the first program in xv6

- Next we are going to demonstrate **how to add a new program** to xv6 (for example a hello.c program)
- This step involves editing the **Makefile** and adding the line of the newly created program.
- Then we are going to **run qemu** to test the hello.c program.

hello.c

Step 1: We save this program inside the source code directory of xv6 operating system. (TIP: copy an already existing xv6 program, keep the #includes,main, then add your source)

```
// A simple program which just prints something on screen

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

int
main(void)
{
 printf(1, "Hello on xv6, myprocess id is %d\n", getpid() );
 exit();
}
```

hello.c

- We are calling printf here and what is different from what we know from printf is that we add “1”

  printf(1, "Hello on xv6, myprocess id is %d\n", getpid());
- We have to specify to which file descriptor this print will go to.
- In Unix environment file descriptors are used for input and output – files or the terminal for example
  - 0: is for things we read in
  - 1: is for things we print out

hello.c

Step 2: Edit the Makefile of the xv6 source code

```
mkfs: mkfs.c fs.h
	gcc -Werror -Wall -o mkfs mkfs.c

# Prevent deletion of intermediate files, e.g. cat.o, after first build, so
# that disk image changes after first build are persistent until clean.  More
# details:
# http://www.gnu.org/software/make/manual/html_node/Chained-Rules.html
.PRECIOUS: %.o

UPROGS=\
	_cat\
	_echo\
	_forktest\
	_grep\
	_init\
	_kill\
	_ln\
	_hello\
	_ls\
	_mkdir\
	_rm\
	_sh\
	_stressfs\
	_usertests\
	_wc\
	_zombie\

fs.img: mkfs README $(UPROGS)
	./mkfs fs.img README $(UPROGS)

-include *.d
```

hello.c

Step 3: Then start xv6 system on QEMU with

make qemu

or

make qemu-nox

Then, check whether our program is available for the user (using ls)

```
t 58
init: starting sh
$ ls
.               1 1 512
..              1 1 512
README          2 2 2290
cat             2 3 13336
echo            2 4 12408
forktest        2 5 8124
grep            2 6 15156
init            2 7 12996
kill            2 8 12456
ln              2 9 12356
hello           2 10 12212
ls              2 11 14580
mkdir           2 12 12480
rm              2 13 12456
sh              2 14 23096
stressfs        2 15 13136
usertests       2 16 56008
wc              2 17 13984
zombie          2 18 12188
console         3 19 0
```

hello.c

- If yes, give the name of that executable program, which is in my case is ***hello*** to see the program output on the terminal.

```
console          3 19 0
$ hello
Hello on xv6, myprocessid is 4
$
```

4

gdb

A brief guide on how to debug programs in C

gdb - how to debug a C program

gdb offers some very useful commands to assist you in the difficult task of debugging your code:

- Pausing the execution
- Printing values in execution time
- Breakpoints!

How to use them? Here is a basic tutorial:

gdb command shortcuts

l : list command - Use gdb command l or list to print the source code in the debug mode. Use l line-number to view a specific line number (or) l function to view a specific function.
bt : backtrack command – Print backtrace of all stack frames, or innermost COUNT frames.

p : print command – prints out the value of the variable given as argument. This is great for inspecting your code's behavior!

gdb command shortcuts

s : step command – Moves the program counter to the next line. Just a little step so you may always know “what just happened”

c : continue command – Resumes execution until it’s over **OR** to the next breakpoint.

Enter: repeat command - This re-executes the last command.

Also, there are always:

help : help command – View help for a particular gdb topic — help TOPICNAME.

quit : quit command – Exit from the gdb debugger.

How to run gdb with QEMU

- Open two terminals in the directory of the xv6 source code.
- In the first, enter **make qemu-nox-gdb**. This starts up QEMU, but it stops just before the processor executes the first instruction and waits for a debugging connection from gdb.
- In the other, enter **gdb**.

How to run gdb with QEMU(2)

Along with the xv6 source code, there is a gdb configuration file, named .gdbinit, to enable its connection with QEMU.

```
warning: File "/home/lydia/Desktop/xv6-public/.gdbinit" auto-loading has been de
clined by your `auto-load safe-path' set to "$debugdir:$datadir/auto-load".
To enable execution of this file add
        add-auto-load-safe-path /home/lydia/Desktop/xv6-public/.gdbinit
line to your configuration file "/home/lydia/.gdbinit".
To completely disable this security protection add
        set auto-load safe-path /
line to your configuration file "/home/lydia/.gdbinit".
For more information about this security protection see the
"Auto-loading safe path" section in the GDB manual.  E.g., run from the shell:
        info "(gdb)Auto-loading safe path"
(gdb)
```

If you encounter the above error, create a file in your home directory with the name “.gdbinit” and type **“add-auto-load-safe-path <path_to_xv6_.gdbinit>”** in order to convince gdb to process the .gdbinit provided by xv6.

How to run gdb with QEMU(3)

Finally, you should see something like this:

```
GNU gdb (Ubuntu 7.7.1-0ubuntu5~14.04.3) 7.7.1
Copyright (C) 2014 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word".
+ target remote localhost:26000
warning: A handler for the OS ABI "GNU/Linux" is not built into this configuration
of GDB.  Attempting to continue with the default i8086 settings.

The target architecture is assumed to be i8086
[f000:fff0]    0xffff0: ljmp   $0xf000,$0xe05b
0x0000fff0 in ?? ()
+ symbol-file kernel
(gdb)
```

Step 3: QEMU

In order to build the patched version of QEMU on Linux:

- You may need to install the following libraries:

  sudo apt-get install libsdl1.2-dev && libtool && libglib2.0-dev && libz-dev && libpixman-1-dev

- Configure the source code:

  In the QEMU directory run **./configure --disable-kvm [--prefix=PFX] [--target-list="i386-softmmu x86_64-softmmu"]**

  The arguments in the brackets are optional. The target-list argument slims down the architectures QEMU will built support for. The prefix argument determines where to install QEMU, otherwise the default location is /usr/local.

- Run **make && make install**

5

System calls

A guide on how to add your own system call to the kernel

Adding our own system call

- It is useful to review the usys.S : SYSCALL file, this is where all the system calls are actually defined.
- We will next show you which files to edit in order to add a new system call to xv6.
- We are going to add a system call named **getyear** which will return **1975** from the kernel always.

Where do I begin ?

We start this work from **syscall.h** file where a number is assigned to every system call in this xv6 system.

As you can see, there are 21 system calls already defined in this file. Let's go ahead and add the following line to reserve a system call number for our own system call.

#define SYS_getyear 22

```
// System call numbers
#define SYS_fork    1
#define SYS_exit    2
#define SYS_wait    3
#define SYS_pipe    4
#define SYS_read    5
#define SYS_kill    6
#define SYS_exec    7
#define SYS_fstat   8
#define SYS_chdir   9
#define SYS_dup    10
#define SYS_getpid 11
#define SYS_sbrk   12
#define SYS_sleep  13
#define SYS_uptime 14
#define SYS_open   15
#define SYS_write  16
#define SYS_mknod  17
#define SYS_unlink 18
#define SYS_link   19
#define SYS_mkdir  20
#define SYS_close  21
#define SYS_getyear 22
```

Moving on ...

Open **syscall.c** file.
There's an array of function pointers inside this file with the function prototype **static int (*syscalls[])(void)**.
It uses the numbers of system calls defined above as indexes for a pointer to each system call function defined elsewhere.
At the end of this function pointer array, let's add the following line.

[SYS_getyear] sys_getyear,

```
static int (*syscalls[])(void) = {
[SYS_fork]    sys_fork,
[SYS_exit]    sys_exit,
[SYS_wait]    sys_wait,
[SYS_pipe]    sys_pipe,
[SYS_read]    sys_read,
[SYS_kill]    sys_kill,
[SYS_exec]    sys_exec,
[SYS_fstat]   sys_fstat,
[SYS_chdir]   sys_chdir,
[SYS_dup]     sys_dup,
[SYS_getpid]  sys_getpid,
[SYS_sbrk]    sys_sbrk,
[SYS_sleep]   sys_sleep,
[SYS_uptime]  sys_uptime,
[SYS_open]    sys_open,
[SYS_write]   sys_write,
[SYS_mknod]   sys_mknod,
[SYS_unlink]  sys_unlink,
[SYS_link]    sys_link,
[SYS_mkdir]   sys_mkdir,
[SYS_close]   sys_close,
[SYS_getyear] sys_getyear,
};
```

What does this mean?

This means, when a system call occurred with the system call number 22, the function pointed by the function pointer **sys_getyear** will be called.

So, we will have to implement this function. (However, this file is not the place we are going to implement it).

What next ?

We will just put the function prototype here inside this file (**syscall.c)**.

So, find the suitable place inside this file and add the following line. You can see that all the other 21 system call functions are defined similarly.

extern int sys_getyear(void);

```
extern int sys_chdir(void);
extern int sys_close(void);
extern int sys_dup(void);
extern int sys_exec(void);
extern int sys_exit(void);
extern int sys_fork(void);
extern int sys_fstat(void);
extern int sys_getpid(void);
extern int sys_kill(void);
extern int sys_link(void);
extern int sys_mkdir(void);
extern int sys_mknod(void);
extern int sys_open(void);
extern int sys_pipe(void);
extern int sys_read(void);
extern int sys_sbrk(void);
extern int sys_sleep(void);
extern int sys_unlink(void);
extern int sys_wait(void);
extern int sys_write(void);
extern int sys_uptime(void);
extern int sys_getyear(void);
```

Implement the actual system call

There are two files inside xv6 system where system calls are defined.

sysproc.c and **sysfile.c** are those two places.

If you open and check, you will see that many system calls related to file system are located in **sysfile.c** while the rest is in **sysproc.c**.

To implement our system call, we use the file **sysproc.c**. Open it and add the following function at the end of the file.

```
int
sys_getyear(void)
{
        return 1975;
}
```

Finishing up ...

Now we have just two little files to edit and these files will contain the interface for our user program to access the system call.

Open the file called **usys.S** and add a line line the below at the end.

SYSCALL(getyear)

```
SYSCALL(fork)
SYSCALL(exit)
SYSCALL(wait)
SYSCALL(pipe)
SYSCALL(read)
SYSCALL(write)
SYSCALL(close)
SYSCALL(kill)
SYSCALL(exec)
SYSCALL(open)
SYSCALL(mknod)
SYSCALL(unlink)
SYSCALL(fstat)
SYSCALL(link)
SYSCALL(mkdir)
SYSCALL(chdir)
SYSCALL(dup)
SYSCALL(getpid)
SYSCALL(sbrk)
SYSCALL(sleep)
SYSCALL(uptime)
SYSCALL(getyear)
```

One last thing!

Then, open the file called **user.h** and add the following line. This is the function that the user program will be calling. As you know now, there's no such a function implemented in the system.

int getyear(void);

Instead, a call to the above function from a user program will be simply mapped to the system call number 22 which is defined as **SYS_getyear** preprocessor directive. The system knows what exactly is this system call and how to handle it.

```
// system calls
int fork(void);
int exit(void) __attribute
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, sh
int unlink(char*);
int fstat(int fd, struct s
int link(char*, char*);
int mkdir(char*);
int chdir(char*);
int dup(int);
int getpid(void);
char* sbrk(int);
int sleep(int);
int uptime(void);
int getyear(void);
```

Now time to test this!

Having done all the above stuff, our xv6 operating system now contains a new system call. Let's write a user program which will try to make that new system call to see whether the xv6 kernel respond to it.

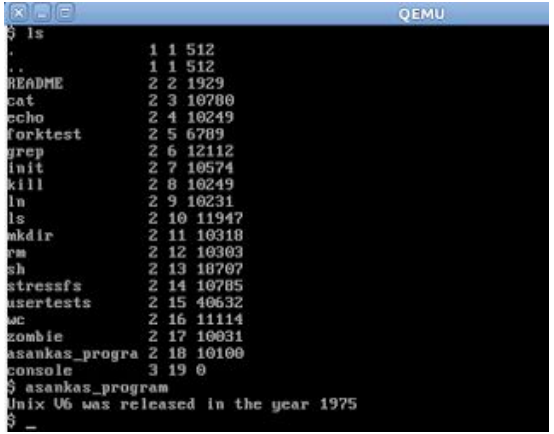
```
// A simple program which just print something on screen
#include "types.h"
#include "stat.h"
#include "user.h"
int main(void)
{
        printf(1, "Unix V6 was released in the year %d\n", getyear());
        exit();
}
```

Look back a few slides if you don't remember how to test your source code

After adding this user program properly into the xv6 source code directly and making the necessary changes in **Makefile**, we can attempt to compile and run it. So, issue the following commands inside the source code directory of xv6.

```
make
```

Now, start the xv6 system on QEMU and wait till it boots up to the shell prompt. There you can issue **ls** command to see whether your user program is available. If yes, just go ahead and run the program.



6

More programs!

Additional practice examples to get more familiar with the kernel

The xv6 kernel, things to remember!

- xv6 kernel provides a subset of the services and system calls that Unix kernels traditionally offer.
- A kernel is a special program that provides services to running programs.
- A running program is called a process.
- Do we know anything about processes ? 
- A process has memory that contains: instructions, data and stack.

Code: creating the first process

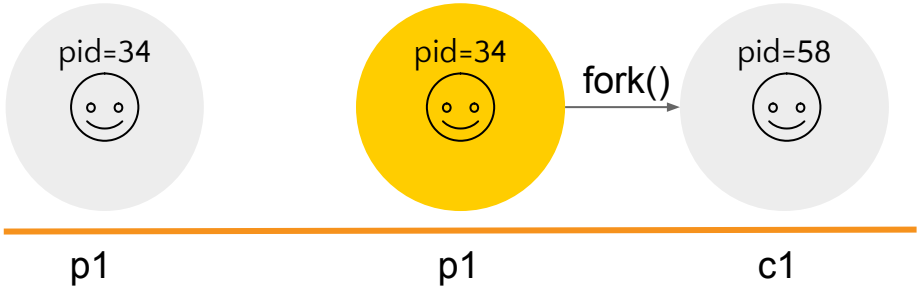
- The xv6 process consists of
  - user-space memory (instructions,data and stack)
  - the per-process state private to the kernel.
- Xv6 can **time-share** processes.
- When a process is not executing xv6 saves its CPU registers.
- It restores them when it next runs this process.

Code: creating the first process

- Each process is assigned a process identifier, the pid, by the kernel.
- A process may create a new process using the fork system call.
- Fork creates a new process, called the child process, with exactly the same memory contents as the calling process, called the parent process.

The fork system call

- After fork() both the parent and the child are executing the same program.
  - pid<0: the creation of a child process was unsuccessful.
  - pid==0: to the newly created child process.
  - pid>0: the process ID of the child process, to the parent.



Practice example!

Write this example (simpfork.c) using xv6 and notice what is happening

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

int
main(void)
{
        int pid = fork();

        if(pid > 0){
        printf(1,"parent: child=%d\n", pid);
        pid = wait();
        printf(1,"child %d is done\n", pid);
        } else if(pid == 0){
        printf(1,"child: exiting\n");
        exit();
        } else {
        printf(1,"fork error\n");
}

}
```

References

This presentation was inspired by

- http://www.fotiskoutoulakis.com/2014/04/28/introduction-to-xv6-adding-a-new-system-call.html
- http://moss.cs.iit.edu/cs450/assign01-xv6-syscall.html
- http://pdos.csail.mit.edu/6.828/2012/xv6/book-rev7.pdf
- http://zoo.cs.yale.edu/classes/cs422/2010/lec/l2-hw
- http://zoo.cs.yale.edu/classes/cs422/2010/xv6-book/trap.pdf

Thanks!

Any questions ?

You can find us on Piazza!