Assignment 2

Start Assignment

Due Tuesday by 11:59pm **Points** 100 **Submitting** a file upload **File Types** tar

The second assignment has three parts. In Part A, the goal is to understand the flow of a system call more fully. To do this, you will be following instructions to add your own system call to xv6 and a simple program that uses it. Part B is similar, but we will ask you to perform more of the work on your own, and in Part C you'll build a parallel utility that depends on your changes in Part B.

- **Lead TA:** Md Ashfaqur Rahaman
  - Office Hours: Wednesday 9:00 to 10:30 (via Zoom) and Tuesday 20:00 to 21:30 (via Zoom) (https://teams.microsoft.com/l/channel/19%3a49813831552b4141a00cf9b956065ca5%40thread.tacv2/Office%2520Hours?groupId=f0a1a98f-c7fe-4bc9-865c-8c62c2a5948c&tenantId=5217e0e7-539d-4563-b1bf-7c6dcf074f91)

Part A - freemem

You will need to do this lab on the CADE machines, just like you did for Assignment 1. I recommend starting with a fresh copy of xv6 in case you messed anything up from Assignment 1B.

```
$ cd
$ git clone /home/cs5460/xv6-public.git xv6-2
Cloning into 'xv6-2'...
done.
$ cd xv6-2
$ make
...
dd if=kernel of=xv6.img seek=1 conv=notrunc
348+1 records in
348+1 records out
178256 bytes (178 kB) copied, 0.00318416 s, 56.0 MB/s
```

Your Goal

You will implement two things in this part:

- `freemem()`: a new xv6 syscall that returns (as an `int`) the amount of free memory in the system in bytes or -1 if the amount of free memory would overflow an `int`.
- `freemem`: a userspace xv6 program that calls `freemem()` and prints the amount of free memory in bytes followed by a newline.

Getting Started

A major goal of this lab is to have you explore enough to discover what is needed to implement this system call in xv6. That said, if you have never implemented a system call, then it might be hard to guess where to start. Like most other code, learning how to write kernel code is best done with examples, and luckily the kernel includes many system calls.

Investigate `getpid()`

Start by tracking `getpid` through the codebase. `getpid()` is a simple syscall that just returns the ID of the calling process. You can see more details about it on Linux with `man 2 getpid`. xv6's `getpid` is basically the same.

```
$ git grep getpid
syscall.c:extern int sys_getpid(void);
syscall.c:[SYS_getpid]  sys_getpid,
syscall.h:#define SYS_getpid 11
sysproc.c:sys_getpid(void)
user.h:int getpid(void);
usertests.c:  ppid = getpid();
usertests.c:    ppid = getpid();
usys.S:SYSCALL(getpid)
```

So we can already see many of the pieces of the call in a handful of files. Check out each of these pieces and see if you can match them up to what we discussed in class. Recall from class: what pieces do we expect to see?

- There should be a header file somewhere that includes the declaration for a `getpid()` function that user level programs include so that they can use the `getpid()` function (that actually performs the syscall/executes the trap instruction).
- There should be an implementation of the `getpid()` function that is linked into the user program that actually performs the trap to the kernel.
  - Recall Intel x86 uses the `int` instruction for this; make sure you can find this instruction in the code.
  - Hint: this user level `getpid()` code is in assembly, and it is assembly generated by a macro.
- There should be an implementation of the syscall in the kernel.
- There should be a syscall table that contains a function pointer to the implementation of each of the system calls in the kernel. That table should contain a pointer to the `getpid` system call in the kernel.
- There should be a syscall number defined for `getpid` that determines which entry in the syscall table points to the `getpid` implementation.
  - Keep in mind that both user programs and the kernel need to know this number.
  - User programs will need to populate `%eax` with the system call number they want to invoke before they trap to the kernel.
  - The kernel needs to know the number so that it can understand which syscall the user program is trying to invoke, then it uses this number to index into the syscall table to find a pointer to the function that performs that syscall.
- Finally, there are a few calls to `getpid` in `usertests.c`. This is a simple user program that tests out a bunch of xv6's functionality, so this program shows an example of how to call `getpid` from a process.

Make a note of all of these locations. You will need to replicate each of these pieces of `getpid` in order to create your own syscall.

Implement the Plumbing

To break up the steps of implementing the syscall, first try to implement a syscall by completely duplicating the logic for `getpid()` but replacing names with `freemem` where needed.

You will see that the actual work of the `getpid` syscall is done in `sys_getpid()` in `sysproc.c`. So there you will end up with your own `sys_freemem`. For now you can return a constant from it or you can just copy the logic of `sys_getpid` and return the calling process' PID until you are ready to implement the rest.

Notice, many of these pieces (the parts in `kalloc.c`, `syscall.c`, `syscall.h`, and `sysproc.c` are part of the xv6 kernel. In the kernel you won't have access to any of the normal functions you are used to using in C. Specifically, things like `printf`, `malloc`, and `strcpy` don't exist in the kernel since the libc that these usually come with depend on the kernel. Still, being able to print things out to see what is happening in the kernel is a big help, the xv6 kernel has a `cprintf` function. Since it can't print to `stdout` (since the kernel itself implements `stdout`) it directly prints things to the user's console. So use this liberally to see how code you've added to the kernel is working etc (being sure to comment out extra debugging prints before submitting).

Once these pieces are in place, make sure you can call your new syscall and that things work (aside from returning the actual amount of free memory available).

Recall, to compile and run you can use `make qemu-nox`.

Create the `freemem` Program

Next, you will need to test your new syscall, so you will need to create a new user-level xv6 program.

Again, you can work by example. The `echo` program is a small program you can duplicate to try out your syscall. Copy `echo.c` to `freemem.c`. There are some weird things about writing xv6 programs. Not all of the functions match POSIX or GNU/Linux, so watch out for that. `printf` requires you to pass it a file descriptor number (similar to `fprintf` in `libc`). Also, if the program doesn't explicitly call `exit()` then the program will crash when `main()` returns. You should be able to avoid these pitfalls by modeling your code after the other user space programs in xv6. (The full list of user programs can be gotten via `ls` at the xv6 shell or by looking at the `UPROGS` variable in `Makefile` in the xv6 source).

Adapt your new `freemem.c` to call your `freemem` syscall and have it print the return value (the number of free bytes in the system) followed by a newline. It shouldn't print other text -- just the decimal number and a newline.

You need to tell the build system that you want it to compile `freemem.c` into an xv6 program and put it into filesystem when you recompile. To do that you need to adjust `Makefile`. Copy the last line in the list of `UPROGS` (the line that says `_zombie\`) and adjust it to say `_freemem\`. When you next compile and run xv6, if things all work then you will have a program in xv6's filesystem called `freemem` that you can run from the shell. Check to make sure it works as you expect and returns whatever you coded in the initial version of your syscall (e.g. if you just copied `getpid` it likely still returns the current process' PID).

Complete the Logic for the syscall

Finally, we need to calculate the amount of free memory in the system and return it from the `freemem` syscall. To do this, take a look in `kalloc.c`. This is where the xv6 kernel keeps track of all the free memory pages. Whenever the kernel needs memory or whenever a process asks for more memory (via `sbrk`), xv6 unlinks a page from `kmem.freelist` and uses that (see `kalloc`).

Each page in the free list accounts for `PGSIZE` bytes of free memory (which is 4096 bytes per page on this hardware).

xv6 doesn't keep track of the number of bytes or pages that are in this free list. Instead, you will have to determine how much free memory is represented by the pages in `kmem.freelist`.

Be a bit careful here. Some synchronization is needed to make this safe because multiple CPU cores could be allocating and deallocating pages from the free list at once. `kmem.lock` is used to provide mutual exclusion on free list access to make this safe. Take a look at the `kalloc` function in that file and make sure you only access `kmem.freelist` and the elements in that list while you have `kmem.lock` acquired.

Implement a function in `kalloc.c` (mine is just called `freemem`) that returns the number of bytes represented in `kmem.freelist` as an `int`. If the number of free bytes would overflow an `int` then return -1.

Add your new function's signature to `defs.h`.

Now, go back to your implementation of `sys_freemem` (in `sysproc.c` if you followed the convention of `getpid`). Replace the code in that function so that it calls your new function.

Use `freemem` Program to Test `freemem()`

Rerun your `freemem` program to make sure it looks like it is working. Even though we allocated 512 MB of memory to our virtual machine, xv6 can't make use of all of it. The short story is that on x86 there isn't a single simple way for xv6 to discover how much physical memory there is in the system, so it assumes there is a fixed amount `PHYSTOP`. Check to see an idle xv6 at least reports free memory close to `PHYSTOP` bytes.

It is also worth trying to allocate some memory from xv6 to make sure that when you do it looks like the free memory lowers as expected (see the `sbrk` syscall).

Part A Modified Files

When you are done, you will likely have changed the following files:

- `Makefile`
- `defs.h`
- `freemem.c`
- `kalloc.c`
- `syscall.c`
- `syscall.h`
- `sysproc.c`
- `user.h`
- `usys.S`

If things are working at this point, I highly recommend you `git add -u && git commit` so that you've got a commit that you can revert to in case things stop working later on in the lab.

Part B - A Better exit() and wait()

In this part you'll implement two new system calls to make `exit()` and `wait()` work more like they do in POSIX/UNIX. You probably already noticed that `exit()` in xv6 is different than it is in POSIX since it takes no exit status code. Likewise, in xv6 `wait()` does return the pid of the child that exited, but it doesn't pass back the exit status code from the exited child process. Your goal is to create `exit2()` and `wait2()` which will fix this. Here are the details for each new syscall:

`int exit2(int exit_status) __attribute__((noreturn))`

This syscall terminates the calling program's process, and it records the `exit_status` so that calls to `wait2()` by the parent process can collect its value. The return value is irrelevant, as indicated by its `__attribute__`. Calls to `exit()` should have identical behavior as calling `exit2(0)`.

`int wait2(int* exit_status)`

This syscall blocks until a child process exits (reaches ZOMBIE state). Once the calling process has an exited child whose exit status has not yet been collected by a prior call to `wait2()`, the call returns the pid of that exited child and populates `exit_status` with the child's exit status. Just like `wait()`, `wait2()` should deallocate the child process's process control block. More specifically, `wait()` should have identical behavior as calling `wait2(NULL)` (in this case `wait2()` just discards the exited child process's exit status and performs the same steps at `wait()`).

Follow similar steps as you did in implementing the `freemem()` syscall above, though here it is much more helpful to start with the code for the `exit()` and `wait()` syscalls respectively. There are a few tips and new things you will run into along the way.

1. `freemem()` only returned a value, but it didn't take any arguments. Passing arguments into xv6's syscall handlers is a bit trickier than handing the return values back.
   A. `exit2()` takes an integer argument. See `sys_kill` for an example of how to fetch an integer argument from the syscall argument list. It uses `int argint(int n, int* ip)` to do this. `n` tells which argument you want to extract from the user space trapframe for the syscall, with 0 being the first argument. `ip` is a pointer to an integer that `argint()` will fill in with the value of the `n`th argument. If `argint` returns 0, then the value pointed to by `ip` contains the `n`th argument; if it returns -1 then no action was taken because the arguments which should have been on the user

space process's stack can't be accessed because it is out of bounds. Note if `argint()` fails you will need to return -1 from `exit2()`.

B. `wait2()` takes a pointer argument that points to an integer-sized region that the kernel will fill in with the child's exit status. Fetching a pointer argument from a syscalls argument list is similar to the case above but a bit trickier. The kernel should only populate the exit status passed by the userspace program if **the whole region** that would be filled in is accessible by the userspace process. `int argptr(int n, char** pp, int size)` to do this. See `sys_read` for an example of its use. `n` tells which argument you want to extract from the user space trapframe for the syscall, with 0 being the first argument. `pp` is a pointer to a pointer that `argptr()` will fill in with the value of the `n`th argument. `size` indicates how big the region pointed to by `*pp` needs to be. In this case, the calling userspace process passes a pointer to an integer that should be filled in by the kernel on success, so the `size` should be `sizeof(int)`. If `argptr` returns 0, then the value pointed to by `pp` contains the `n`th argument; if it returns -1 then no action was taken because the argument couldn't be retrieved from the process's stack (it was out of bounds) or the pointer retrieved from the stack didn't point to a valid location whose entire following `size` bytes is in bounds for the process. Note if `argptr()` fails you will need to return -1 from `wait2()`.

2. Inside the xv6 kernel your implementation of `exit2()` and `wait2()` should share almost all of its implementation with `exit()` and `wait()`. You can find the kernel functions that perform most of the `exit()` and `wait()` logic in `proc.c`. Think about how to change those so that the syscall kernel syscall handlers `sys_exit()`, `sys_exit2()`, `sys_wait()`, and `sys_wait2()` in `sysproc.c` can both use them.
3. Process control blocks are defined in `proc.h`. You'll need to change them.
4. When you change the kernel's `wait()` function in `proc.c`, you'll need to be careful. It is only safe to access and modify fields in a process control block while `ptable.lock` is held, since multiple CPU cores in xv6 (yes, its multicore!) could be accessing process control blocks concurrently. Make sure your accesses and changes to fields in process control blocks fall between the lines of code in that function that have that lock acquired.
5. The same warning applies to `exit()` in `proc.c`; in that function, however, the lock is released after the call to `sched()` which never returns.
6. Several places in the kernel call `exit()` in `proc.c` particularly in `trap.c` where processes might be killed due to unexpected traps, etc. If you change the function signature for `exit()` in `proc.c` and its declaration in `defs.h` be aware you'll have to update all of the calls to it in the kernel (these are easy to find by just changing the function signature for `exit()` in `proc.c` and `defs.h` and just recompiling to let the compiler point these calls out.

Once you have your syscalls implemented, write a small userspace program called `testexit` that checks to make sure `exit2()` and `wait2()` work like you expect. We will supply our own programs that call `exit2()` and `wait2()`, so feel free to implement `testexit` however you please.

Part B Modified Files

When you are done, you will likely have changed the following files:

- `defs.h`
- `proc.c`
- `proc.h`
- `syscall.c`
- `syscall.h`
- `sysproc.c`
- `trap.c`
- `user.h`
- `usys.S`
- `testexit.c`

If things are working at this point, I highly recommend you `git add -u && git commit` so that you've got a commit that you can revert to in case things stop working later on in the lab.

Part C - crc32

In the final part of the assignment you'll be using your new `exit2()` and `wait2()` syscalls to implement a parallel checksumming utility. This userspace utility program is called `crc32`, so create a `crc32.c` and add it to the `UPROGS` variable in `Makefile` so it can be compiled. I highly recommend copying `ls.c` to `crc32.c` to get started, since much of the functionality you'll need in `crc32.c` is similar to what already exists in `ls.c`. We **give you the code (https://utah.instructure.com/courses/756700/files/127552624/download?download_frd=1)** for a `crc32()` function which you should use to compute the CRC32 of a file by reading the file's bytes and passing them into `crc32()`. Include it however you like; it is probably easiest just to copy-paste the code right into your `crc32.c` file.

`crc32` is already an existing utility in Linux that is very similar to the one you will implement. You can try it out by running it on a file:

```
$ crc32 _ls
41b5561e
$ crc32 /bin/ls
f1833563
```

`crc32` computes a CRC32 checksum over all of the bytes in the specified file. If any byte in the file changes (or its file length changes), then the CRC32 that `crc32` computes will change. So, `crc32` can be used to check the integrity of files to see if they've been changed or corrupted.

Your version of `crc32` is very similar to Linux's with a few differences. You should be able to run your program by running `crc32 <path>` If the `<path>` argument is missing or it doesn't represent a file or directory then `crc32` should print an error message and exit with exit status code -1. If `<path>` does exist the behavior depends on whether `<path>` specifies the name of a file or a directory.

Computing crc32 for One File

If `<path>` indicates a file, then `crc32` reads the contents of the file and computes the CRC32 using `crc32()`. It should print out the resulting CRC32 on a single line followed by a newline using the `"%x"` format specifier to `printf()` (`printf("%x\n", crc);`). (This output differs slightly from Linux's `crc32` because xv6's `"%x"` doesn't pad with zeroes and it uses upper case letters in its hex output. Here is an example output (note your CRCs for `ls` may not match mine but the CRC32 for `README` should).

```
$ crc32 ls
41B5561E
$ crc32 README
73B388BA
```

(Note that from your bash shell in Linux the file `_ls` is the same file that appears in your xv6 VM as `ls`, which is why this run of `crc32` outputs the same number as the first one run from Linux above. The two files have the same content. Hence, you can use Linux's `crc32` to check the output of your xv6 `crc32`.)

Here are some important tips on implementing this part:

1. If you start by copying the `ls.c` code to get started, the the `ls()` function will do much of the work for you when you get to the next step and need to compute checksums over multiple files in a directory. If you look in the `ls()` function, you will see that it takes a `path` string, it opens that path, and then it switches to do something different depending on whether `path` indicates a file or a directory. You can replace the part that handles the case that the opened path has

type `T_FILE` with code that opens the file, reads its content, computes its CRC32, and prints it. The `T_DIR` case you will change a bit later.

2. If you run into an error opening or reading the file just print an error message and `exit2(-1)`.
3. In general you can't guarantee that you can read a whole file into memory; it could be too large. To solve this, the `crc32()` function that we gave you can incrementally compute a CRC32 a little bit at a time. The example below shows how this works:

The function we give you has the signature:

`unsigned long crc32(unsigned long crc, const void *buf, unsigned long size)`

It returns a CRC32 (as a `unsigned long`) of the `size` bytes pointed to by `buf`. If the entire contents of a file started at `buf` and `buf` was the length of the file contents then `crc32(0, buf, size)` would return the correct CRC32 for the file. To process the file in smaller chunks, the first argument of `crc32()` is a CRC32 itself. It's easiest to explain how this works by example:

```
const char* whole_buf = "abcdef";
const size_t len = 6;

unsigned long crc_whole = crc32(0, whole_buf, len);

unsigned long crc_byte_by_byte = 0;
for (size_t i = 0; i < len; ++i)
    crc_byte_by_byte = crc32(crc_byte_by_byte, &whole_buf[i], 1);

printf("%x\n", crc_whole);
printf("%x\n", crc_byte_by_byte);
```

Running this gives:

```
$ ./a.out
4b8e39ef
4b8e39ef
```

Notice that the "for loop" approach to computing the CRC32, though, doesn't require the whole buffer; it can process data a chunk at a time.

Once you've implemented support for printing the CRC32 of a single file, test it on several of the files you find in the xv6 root directory or files you've created yourself in xv6 (e.g. `echo mytest > testfile` keeping in mind that this will create a file with a trailing newline) and compare with what `crc32` on Linux gives you back. `README` is a good file to test it on; if you haven't changed the file its CRC32 should be 73B388BA.

Parallel CRC32 on a Directory

If `<path>` indicates a directory, then `crc32` computes the CRC32 of each file in the directory and prints each CRC32 on its own line. For full credit your implementation must parallelize its work in the following way. The initial process (which I'll call the parent here), iterates through all of the file paths in the directory. For each file the parent encounters in the specified directory it must fork off a child process that (in the background) reads that file, computes its CRC32, and then returns the CRC32 to the parent via `exit2`. The parent should start **all** children before calling `wait2` to wait on any of its children, so that all of the children read their respective files and compute a single CRC32 in parallel.

A good strategy here is to first implement the code that iterates over the directory and reads each file, computes its CRC32, and prints it all in the parent process. That will let you run your code to be sure it its working correctly before you place `fork`, `exit2`, and `wait2` calls in the right places to implement the parallel computation in the children. If you run into a problem on `fork` just have the parent print an error message and exit. If a child runs into a problem (e.g. a failure on `read`, it can print an error message and then `exit2(-1)`). The parent can simply print that specific file's CRC32 as `FFFFFFFF`.

```
$ crc32 .
73B388BA
1EB8140C
C9C00FD6
ABC11C3D
DAEB0A86
3E015350
F07CEB04
D4396B43
B1026A77
B546FBBE
D1F752D7
A9FBF6B
5A41FD3F
41B5561E
ED729CBD
DF492395
C8B6047B
E3C23DE7
$ mkdir test
$ cat ls > test/ls
$ crc32 test
41B5561E
```

Part C Modified Files

When you are done, you will likely have changed the following files:

- `crc32.c`

Fun Stuff to Try [Optional]

Here we list a few things you can build on top of what you've done already just in case you are looking for some extra fun. Don't feel obligated to try them, but these are a few things we've found interesting hacking into xv6 ourselves in the past that you might find interesting.

xv6 programs crash if you return from `main()` instead of calling `exit()`. See if you can figure out why, and see if you can fix it. Now that you have `exit2()` you can use that to implement the same scheme that Linux/POSIX processes have where returning from `main()` implicitly exits with using the return value from main at the exit status code. Part of the trick here is that in Linux, programs start in a symbol called `_start()` that actually calls `main()`. You can define a similar function in xv6's libc (which is implemented in `ulib.c`). You have to adjust the entry point to programs to programs as well in xv6's `Makefile`; look for calls to the linker (`$(LD)`) and look for the `-e` argument, which specifies where a compiled programs entry point is.

Handin

You will turn in your entire xv6 directory *without* build files, etc. To do that run the following steps.

```
$ make clean
$ cd ..
$ ls xv6-2
...
date.h      file.h        ioapic.c    ls.c       Notes        rm.c      sleeplock.h  sysfile.c   ulib.c
```

Make sure no `.o` files or `.img` files are in there since they are large. Then bundle up all the files to send to us.

```
tar cvf xv6-2.tar --exclude=.git xv6-2/
```

Afterward, upload the resulting xv6-2.tar to Canvas. (For reference, my xv6-2.tar was 370 KB; yours should be right in that ballpark.)

Scoring

- Part A (35 Points)
  - (5 Points) `freemem()` symbol defined correctly; user programs referring to it compile.
  - (5 Points) `freemem()` can be called and the process/kernel doesn't crash.
  - (5 Points) `freemem()` syscall enters kernel successfully and returns.
  - (10 Points) `freemem()` returns actual available memory in bytes.
  - (5 Points) `freemem` program compiles and runs without crash.
  - (5 Points) `freemem` program prints actual available memory in bytes.
- Part B (35 Points)
  - (5 Points) `exit2()` and `wait2()` symbols defined correctly; user programs referring to them compile.
  - (5 Points) `exit2()` and `wait2()` can be called and the process/kernel doesn't crash.
  - (5 Points) `exit2()` and `wait2()` syscalls enter kernel successfully and return.
  - (10 Points) `exit2()` records the exit status code of a process and correctly terminates the process.
  - (10 Points) `wait2()` returns the exit status code of an exited child (when one exists) and correctly cleans up its child's process control block.
- Part A (30 Points)
  - (5 Points) `crc32` compiles and produces an xv6 executable
  - (5 Points) `crc32` exits with an error message when it isn't given one command line argument
  - (5 Points) `crc32` correctly computes and prints the CRC32 of a file in the specified format when the command line argument specifies a file path
  - (5 Points) `crc32` correctly computes and prints the CRC32 of all files in the specified directory in the specified format when the command line argument specifies a directory path
  - (5 Points) `crc32` uses `fork`, `exit2`, and `wait2` to compute each file's CRC32 in the background when the command line argument specifies a directory path
  - (5 Points) `crc32` passes the CRC32 of each file from each child process to its parent using `exit2`/`wait2`
- Total 100 points

You will have submitted one file.

- `xv6-2.tar` with all of the files needed to build and run your new xv6 `freemem` and `crc32` programs and their respective syscalls.

If you update your files multiple times in Canvas it adds a number onto the end of the file names. Don't worry about that; we'll grade the most recent version that was submitted before the deadline when we grade.

FAQ & Tips for For xv6 Programming and Debugging

Q: What does trap -- kill proc mean?

xv6's trap handler isn't very informative about what went wrong when a process crashes. It looks something like this when a process crashes in xv6:

```
pid 3 myproc: trap 14 err 6 on cpu 1 eip 0x16 addr 0xffffff--kill proc
```

This is saying that pid 3 which was running program "myprog" experienced trap 14 while execute the instruction at address 0x16 and it tried to access address 0xffffff. As a result, xv6 killed the process. So what does a trap 14 mean (or any other number for that matter). That gets back to what is inside the interrupt descriptor table that we talked about in class. Luckily, xv6 has a set of constants and a tiny bit of documentation that describes what each trap is (in trap.h):

```
#define T_DIVIDE         0      // divide error
#define T_DEBUG          1      // debug exception
#define T_NMI            2      // non-maskable interrupt
#define T_BRKPT          3      // breakpoint
#define T_OFLOW          4      // overflow
#define T_BOUND          5      // bounds check
#define T_ILLOP          6      // illegal opcode
#define T_DEVICE         7      // device not available
#define T_DBLFLT         8      // double fault
// #define T_COPROC      9      // reserved (not used since 486)
#define T_TSS           10      // invalid task switch segment
#define T_SEGNP         11      // segment not present
#define T_STACK         12      // stack exception
#define T_GPFLT         13      // general protection fault
#define T_PGFLT         14      // page fault
// #define T_RES        15      // reserved
#define T_FPERR         16      // floating point error
#define T_ALIGN         17      // aligment check
#define T_MCHK          18      // machine check
#define T_SIMDERR       19      // SIMD floating point error
```

Here you can see trap 14 is a "page fault". A page fault occurs when the process tries to access a page/address in its address space that isn't mapped. So, this likely means my program tried to de-reference a pointer to an address that it shouldn't have (0xffffff in this case).

Q: How do I understand unexpected trap messages?

If you accidentally do something that raises a trap that shouldn't happen while in kernel code, the kernel will panic. This can happen, for example, if you dereference a bad pointer in the kernel that accesses an unmapped or inaccessible virtual address. Here's an example:

```
unexpected trap 14 from cpu 1 eip 80103d09 (cr2=0xffffff)
lapicid 1: panic: trap
 80105b2a 801057cf 8010559d 80104999 80105a7d 801057cf 0 0 0 0
```

Here this is saying that trap 14 (a page fault, see the question above) was raised while the CPU was executing in kernel code. 0x80103d09 points to the instruction that triggered the trap. cr2=0xffffff indicates which address access triggered the trap, if any. The sequence of hex numbers that it spits out afterward represent the return addresses of the call stack in the kernel. If you're struggling to find where a fault is occurring, you can use "objdump -dS kernel" to get a disassembly of the kernel that overlays the kernel source code. If you search for these addresses in that disassembly you can find which function and which instructions triggered the fault. (In general, this is a bit painful, so commenting things/using cprintf are often easier in the kernel.)

Q: I always get errors when I run make qemu-nox!!

Revisit the qemu setup section in Assignment 1.

If you see this messages like this when running later on, then you have a problem with your PATH environment variable; it isn't set up properly so that make can find qemu-system-i386 (which we have to install in a custom location on the CADE cluster):

```
$ make qemu-nox
which: no qemu in (/bin:/usr/bin)
which: no qemu-system-i386 in (/bin:/usr/bin)
which: no qemu-system-x86_64 in (/bin:/usr/bin)
***
*** Error: Couldn't find a working QEMU executable.
*** Is the directory containing the qemu binary in your PATH
*** or have you tried setting the QEMU variable in Makefile?
***
nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -smp 2 -m 512
make: nographic: Command not found
make: [qemu-nox] Error 127 (ignored)
```

Similarly, you have a problem with your LD_LIBRARY_PATH (which tells the OS loader where it can find the additional libraries our custom qemu is linked against) if you see something like this later:

```
$ make qemu-nox
...
qemu-system-i386 -nographic -drive file=fs.img,index=1,media=disk,format=raw -drive file=xv6.img,index=0,media=disk,format=raw -
smp 2 -m 512

qemu-system-i386: error while loading shared libraries: libfdt.so.1: cannot open shared object file: No such file or directory
make: *** [Makefile:231: qemu-nox] Error 127
```

Associated Learning Outcomes

Primary:

A. describe how kernels typically implement standard OS abstractions (e.g. processes, threads, files) and system calls (e.g. fork, exec) and modify their implementations

Secondary:

F. describe the underlying mechanisms behind recent OS innovations (e.g. hardware virtualization support and virtual machines, Flash storage and flash translations layers)