COP 6611: Operating Systems

Project 2 - Process Management and Trap Handling

Due Date: 04/07/2019

Objectives

When you are done with this lab you should have a better picture of how the kernel handles multitasking, and you should understand how system calls and exceptions work.

All work should be done individually.

Assignment Description

This lab is split into two parts. In the first part of this lab, you will implement the basic kernel facilities required to get multiple protected user-mode processes running. You will enhance the kernel to set up the data structures to keep track of user processes, create multiple user processes, load a program image into a user process, and start running a user process. In the second part, you will make the kernel capable of handling system calls/interrupts/exceptions made or triggered by user processes.

Assignment Setup

Please read Chapter 2.3 through 2.8 in the textbook if you haven't already. Additionally, please look through chapters 7 and 9 in the Intel 80386 Programmer's Reference Manual. These are great references if you get stuck!

Inline Assembly Reading
To do this assignment you might find it helpful to understand "inline" assembly in your C files, this guide is a good reference: http://www.delorie.com/djgpp/doc/brennan/brennan_att_inline_djgpp.html.

Code Checkout

Please follow these instructions to fetch the code and merge your solution from project 1. If you

First, go into your working directory i.e. your project directory from project0.

```
$ cd [PATH TO YOUR PROJECT DIRECTORY]
```

From here on, all the instructions will be run in your project directory.

Fetch from remote upstream repository:

```
$ git fetch upstream
```

Checkout branch for project1:

```
$ git checkout -b project2 upstream/project2
$ git config --unset branch.project2.remote
```

This will checkout the starter code from the upstream repository and remove the upstream link so that you can work in your own repository after fetching the code.

Merge your solution:

```
$ git merge project1
```

Testing The Kernel

We will be grading your code with a bunch of test cases, part of which are given in `test.c` in each layer's sub directory. If you have already run `make` before, you have to first run `make clean` before you run `make TEST=1`.

- **Make sure your code passes all the tests for the current layer before moving up to the next.**
- You can write your own test cases to challenge your implementation.

```
$ make clean
$ make TEST=1 [SOL=1]
```

While writing your code, use these commands to run unit tests to verify the working status of your code. The test case output should look like this when you are finished:

```
Testing the PKCtxNew layer…
test 1 passed.
All tests passed.

Testing the PTCBInit layer…
test 1 passed.
All tests passed.

Testing the PTQueueInit layer…
test 1 passed.
test 2 passed.
All tests passed.

Testing the PThread layer…
test 1 passed.
All tests passed.

Test complete. Please Use Ctrl-a x to exit qemu.
```

If you don't see these tests, make sure you have the correct bootloader code and virtual memory code from project 0 and project 1 or use the optional SOL=1 flag!

Normal Execution

Building the image:

```
$ make clean
$ make [SOL=1]
```

These two commands will do all the steps required to build an image file. The make clean step will delete all the compiled binaries from past runs. After the image has been created, use one of the following execution modes:

Executing WITH QEMU VGA Monitor

```
$ make qemu
```

You can use `Ctrl-a x` to exit from qemu.

Executing WITHOUT QEMU VGA Monitor

```
$ make qemu-nox
```

Executing with GDB and QEMU VGA Monitor

```
$ make qemu-gdb
```

Running User Processes

We have implemented a new monitor task **startuser**. Instead of the dummy process in the last assignment, once run, it starts the idle process implemented in `user/idle/idle.c`, which in turn spawns three user processes defined in `user/pingpong/ping.c`, `user/pingpong/pong.c`, and `user/pingpong/ding.c`, at the user level, with full memory isolation and protection.

User Processes

We have implemented some simple procedures in those three processes to show that the memory for different processes are isolated (with page-based virtual memory), and each process owns the entire 4GB of the memory. As before, the programs will not run until you have implemented all the code for the process management and trap handling modules, and the implementations of the functions from the previous assignments are correct. After you can successfully run and understand the user processes we provided, you can replace it with more sophisticated user process implementations.

In the main functions of the three processes, you may notice that we explicitly call the function `yield()` to yield to another process. Since we do not have preemption implemented in the kernel, unless you explicitly yield to other processes, the current process will not release the CPU to other processes. This is obviously not an ideal situation in terms of protecting the system from bugs or malicious code in user-mode environments, because any user-mode environment can bring the whole system to a halt simply by getting into an infinite loop and never giving back the CPU.

A sample output for `startuser` is shown below.

```
$> help
help - Display this list of commands
kerninfo - Display information about the kernel
startuser - Start the user idle process
$> startuser
[D] kern/lib/monitor.c:45: process idle 1 is created.
Start user-space ...
idle
ping in process 4.
pong in process 5.
ding in process 6.
ping started.
ping: the value at address e0000000: 0
ping: writing the value 100 to the address e0000000
pong started.
pong: the value at address e0000000: 0
pong: writing the value 200 to the address e0000000
ding started.
ding: the value at address e0000000: 0
ding: writing the value 300 to the address e0000000
ping: the new value at address e0000000: 100
pong: the new value at address e0000000: 200
ding: the new value at address e0000000: 300
```

The 3 processes ping, pong and ding start with PID's 4, 5 and 6. Each process writes a different value at the same virtual memory address `e0000000` and reads back from the same location. We see that each process reads back its own value, showing that the page-based virtual memory implemented in project 1 is working and indeed provides an illusion that each process owns the entire 4GB of the memory.

Part 0: Setting up Interrupt Mechanism

In this kernel, we (fairly arbitrarily) use software interrupt vector 48 (0x30) as a system call interrupt vector. At this point, the `int $0x30` system call instruction in user space is a dead end: once the processor gets into user mode, there is no way to get back out. The first thing you should do is thoroughly familiarize yourself with the x86 interrupt and exception mechanism.

Exercise

Read Chapter 9, Exceptions and Interrupts in the 80386 Programmer's Manual (or Chapter 5 of the IA-32 Developer's Manual), if you haven't already.

In this lab we generally follow Intel's terminology for interrupts, exceptions, and the like. However, terms such as exception, trap, interrupt, fault and abort have no standard meaning across architectures or operating systems, and are often used without regard to the subtle distinctions between them on a particular architecture such as the x86. When you see these terms outside of this lab, the meanings might be slightly different.

Basics of Protected Control Transfer

Exceptions and interrupts are both "protected control transfers," which cause the processor to switch from user to kernel mode (`CPL=0`) without giving the user-mode code any opportunity to interfere with the functioning of the kernel or other environments. In Intel's terminology, an *interrupt* is a protected control transfer that is caused by an asynchronous event usually external to the processor, such as notification of external device I/O activity. An *exception*, in contrast, is a protected control transfer caused synchronously by the currently running code, for example due to a divide by zero or an invalid memory access.

In order to ensure that these protected control transfers are actually *protected*, the processor's interrupt/exception mechanism is designed so that the code currently running when the interrupt or exception occurs *does not get to choose arbitrarily where the kernel is entered or how*. Instead, the processor ensures that the kernel can be entered only under carefully controlled conditions. On the x86, two mechanisms work together to provide this protection:

1. **The Interrupt Descriptor Table.** The processor ensures that interrupts and exceptions can only cause the kernel to be entered at a few specific, well-defined entry-points *determined by the kernel itself*, and not by the code running when the interrupt or exception is taken.

The x86 allows up to 256 different interrupt or exception entry points into the kernel, each with a different *interrupt vector*. A vector is a number between 0 and 255. An interrupt's vector is determined by the source of the interrupt: different devices, error conditions, and application requests to the kernel generate interrupts with different vectors. The CPU uses the vector as an index into the processor's *interrupt descriptor table* (`IDT`), which the kernel sets up in kernel-private memory, much like the `GDT`. From the appropriate entry in this table the processor loads:

    a. the value to load into the instruction pointer (`EIP`) register, pointing to the kernel code designated to handle that type of exception.
    b. the value to load into the code segment (`CS`) register, which includes in bits 0-1 the privilege level at which the exception handler is to run. (In this OS, all exceptions are handled in kernel mode, privilege level 0.)
2. **The Task State Segment.** The processor needs a place to save the *old* processor state before the interrupt or exception occurred, such as the original values of `EIP` and `CS` before the processor invoked the exception handler, so that the exception handler can later restore that old state and resume the interrupted code from where it left off. But this save area for the old processor state must in turn be protected from unprivileged user-mode code; otherwise buggy or malicious user code could compromise the kernel.

For this reason, when an x86 processor takes an interrupt or trap that causes a privilege level change from user to kernel mode, it also switches to a stack in the kernel's memory. A structure called the *task state segment* (`TSS`) specifies the segment selector and address where this stack lives. The processor pushes (on this new stack) `SS`, `ESP`, `EFLAGS`, `CS`, `EIP`, and an optional error code. Then it loads the `CS` and `EIP` from the interrupt descriptor, and sets the `ESP` and `SS` to refer to the new stack.

Although the TSS is large and can potentially serve a variety of purposes, we only uses it to define the kernel stack that the processor should switch to when it transfers from user to kernel mode. Since "kernel mode" in this OS is privilege level 0 on the x86, the processor uses the ESP0 and SS0 fields of the TSS to define the kernel stack when entering kernel mode. We don't use any other TSS fields.

Types of Exceptions and Interrupts

All of the synchronous exceptions that the x86 processor can generate internally use interrupt vectors between 0 and 31, and therefore map to IDT entries 0-31. For example, a page fault always causes an exception through vector 14. Interrupt vectors greater than 31 are only used by *software interrupts*, which can be generated by the int instruction, or asynchronous *hardware interrupts*, caused by external devices when they need attention. We (fairly arbitrarily) use software interrupt vector 48 (0x30) as its system call interrupt vector.

An Example

Let's put these pieces together and trace through an example. Let's say the processor is executing code in a user environment and encounters a divide instruction that attempts to divide by zero.

1. The processor switches to the stack defined by the `SS0` and `ESP0` fields of the `TSS`, which in our OS will hold the values `GD_KD` and `KSTACKTOP`, respectively.
2. The processor pushes the exception parameters on the kernel stack, starting at address `KSTACKTOP`:

```
                +--------------------+ KSTACKTOP
                | 0x00000 | old SS   |     " - 4
                |      old ESP       |     " - 8
                |     old EFLAGS     |     " - 12
                | 0x00000 | old CS   |     " - 16
                |      old EIP       |     " - 20 <---- ESP
                +--------------------+
```

1. Because we're handling a divide error, which is interrupt vector 0 on the x86, the processor reads `IDT` entry 0 and sets `CS:EIP` to point to the handler function described by the entry.
2. The handler function takes control and handles the exception, for example by terminating the user environment.

For certain types of x86 exceptions, in addition to the "standard" five words above, the processor pushes onto the stack another word containing an *error code*. The page fault exception, number 14, is an important example. See the 80386 manual to determine for which exception numbers the processor pushes an error code, and what the error code means in that case. When the processor pushes an error code, the stack would look as follows at the beginning of the exception handler when coming in from user mode:

```
                +--------------------+ KSTACKTOP
                | 0x00000 | old SS   |     " - 4
                |      old ESP       |     " - 8
                |     old EFLAGS     |     " - 12
                | 0x00000 | old CS   |     " - 16
                |      old EIP       |     " - 20
                |     error code     |     " - 24 <---- ESP
                +--------------------+
```

Nested Exceptions and Interrupts

The processor can take exceptions and interrupts both from kernel and user mode. It is only when entering the kernel from user mode, however, that the x86 processor automatically switches stacks before pushing its old register state onto the stack and invoking the appropriate exception handler through the IDT. If the processor is *already* in kernel mode when the interrupt or exception occurs (the low 2 bits of the `CS` register are already zero), then the CPU just pushes more values on the same kernel stack. In this way, the kernel can gracefully handle *nested exceptions* caused by code within the kernel itself. This capability is an important tool in implementing protection.

On the other hand, in the version of OS provided in this lab, we always turn off the interrupts when we are in the kernel mode, to make our life simpler.

Setting Up the IDT

You should now have the basic information you need in order to understand the code for the IDT set up and exception handling in mCertiKOS. Related code is in `lib/trap.h`, `dev/intr.h`, `dev/intr.c`, `dev/idt.S`.

Note: Some of the exceptions in the range 0-31 are defined by Intel to be reserved. Since they will never be generated by the processor, it doesn't really matter how you handle them. So we can do whatever we think is cleanest.

The overall flow of control that you should achieve is depicted below:

```
     IDT                   dev/idt.S          trap/TTrapHandler/TTrapHandler.c
+----------------+
|   &handler1    |---------> handler1:          trap (struct tf_t *tf)
|                |             // do stuff      {
|                |             call trap          // handle the exception/interrupt
|                |             // ...           }
+----------------+
|   &handler2    |--------> handler2:
|                |            // do stuff
|                |            call trap
|                |            // …
+----------------+
      .
      .
      .
+----------------+
|   &handlerX    |--------> handlerX:
|                |             // do stuff
|                |             call trap
|                |             // …
+----------------+
```

Each exception or interrupt should have its own handler in `dev/idt.S` and `intr_init_idt()` in `dev/intr.c` should initialize the `IDT` with the addresses of these handlers. Each of the handlers should build a `struct tf_t` (see `lib/trap.h`) on the stack and call `trap()` (in `trap/TTrapHandler/TTrapHandler.c`) with a pointer to the trap frame. `trap()` then handles the exception/interrupt or dispatches to a specific handler function.

Exercise

Read the TSS initialization code in function `seg_init()` of: `kern/lib/seg.c`
Read the IDT initialization code in function `intr_init_idt()` of: `kern/dev/intr.c`

Exercise 0: Trap Frame handling

In the file `kern/dev/idt.S`, we setup the code for the interrupt handlers (see `.text` section from line number 34). These handlers can be of two varieties, depending on whether the CPU automatically pushes the error code onto the stack or not. Please read the macros defined for each:

- `TRAPHANDLER`
- `TRAPHANDLER_NOEC`

All of the interrupt handlers then proceed to the `_alltraps` function. This is the an important function where we save the trap frame on the stack. Then we call the `trap` function (to be described later) and pass the pointer to this trap frame on the stack as the argument. Please carefully read and understand this function.

Implement:

- `trap_return`

This function will be called once the trap/exception/interrupt is completely handled and the kernel is ready to resume the user level process. This function should hence undo the actions of `trap_return` and the interrupt handlers(`TRAPHANDLER`/`TRAPHANDLER_NOEC`) and return execution to the user level process.

Part 1: Thread & Process Management

In this kernel, every process created in the application level has a corresponding stack in the kernel. Any system call requests from a particular process will be using the corresponding kernel stack.

When a process yields, via the `sys_yield` system call we do the following:

- The process first traps into the kernel; we switch the page structure to page structure #0 (so that the kernel can directly use physical memory via the identity page map).
- The kernel saves the current process states (user context/trap frame).
- Then switch to the next ready process (represented by the corresponding kernel thread).
- Then the resumed kernel thread restores the process states, switches the page structure to the corresponding process' page structure, and then returns into the user process.

Exercise 1: The PKCtxIntro Layer

In this layer, we introduce the notion of kernel thread context. When you switch from one kernel thread to another, you need to save the current thread's states (the six register values defined in the `kctx` structure, see below) and restore the new thread's states. **Read `kern/thread/PKCtxIntro/PKCtxIntro.c` carefully to make sure you understand every concept.**

```
// Code segment from kern/thread/PKCtxIntro.c...

struct kctx {
      void  *esp;
      unsigned int edi;
      unsigned int esi;
      unsigned int ebx;
      unsigned int ebp;
      void  *eip;
};
```

In the implementation of `cswitch`, you will first fill a `kctx structure` using the values of all six registers, except `eip`. The eip value you need to save should be the *return address pushed onto the current thread's stack*. By the C calling convention, when function call occurs, return address is first pushed onto the stack before the arguments. Therefore, the first argument of the function is `4(%esp)` instead of `0(%esp)`, since the latter value represents the return address.

The second step is to set each of these registers using a different process' context (let's call it process B). Since we want to "return" into process B, we want to place the `eip` field of `kctx structure` onto the top of the stack so that it will use that value in the `ret` statement.

In the file `kern/thread/PKCtxIntro/cswitch.S`, you must implement the function listed below:

- `cswitch`

Exercise 2: The PKCtxNew Layer

In this layer, you are going to implement a function that creates new kernel context for a child process. Please make sure you read all the comments carefully.

In the file `kern/thread/PKCtxNew/PKCtxNew.c`, you must implement the function listed below:

- `kctx_new`

For this exercise you will need to use three functions defined in the import.h file.

```
unsigned int alloc_mem_quota(unsigned int id, unsigned int quota);
void kctx_set_esp(unsigned int pid, void *esp);
void kctx_set_eip(unsigned int pid, void *eip);
```

You will also need to use the `STACK_LOC` data structure in `PKCtxNew.c`. This 2D array stores the kernel stack for each process.

```
extern char STACK_LOC[NUM_IDS][PAGESIZE] gcc_aligned(PAGESIZE);
```

The `alloc_mem_quota` function will return a new index of a child. Use this index to set the `eip` and `esp` in the context.

The PTCBIntro Layer

In this layer, we introduce the thread control blocks (TCB). Since the code in this layer is fairly simple, we have already implemented it for you. Please make sure you understand all the code we provide in :

`kern/thread/PTCBIntro/PTCBIntro.c`.

Exercise 3: The PTCBInit Layer

In this layer, you are going to implement a function that initializes all TCBs. Please make sure you read all the comments carefully.

In the file `kern/thread/PTCBInit/PTCBInit.c`, you must implement all the functions listed below:

- `tcb_init`

For this exercise you will need to use two functions defined in the `import.h` file.

```
void paging_init(unsigned int);
void tcb_init_at_id(unsigned int);
```

This function is fairly simple, you will just need to initialize paging and call `tcb_init_at_id` for each possible process (maximum is `NUM_IDS`).

The PTQueueIntro Layer

In this layer, we introduce the thread queues. Please make sure you understand all the code we provide in `kern/thread/PTQueueIntro/PTQueueIntro.c`.

Exercise 4: The PTQueueInit Layer

In this layer, you are going to implement a function that initializes all the thread queues, plus the functions that manipulate the queues. Please make sure you read all the comments carefully.

In the file `kern/thread/PTQueueInit/PTQueueInit.c`, you must implement all the functions listed below:

- `tqueue_init`
- `tqueue_enqueue`
- `tqueue_dequeue`
- `tqueue_remove`

For the `tqueue_init` function we will need to use the following two functions in the `import.h` file.

```
void tcb_init(unsigned int mbi_addr);
void tqueue_init_at_id(unsigned int chid);
```

We first call `tcb_init` to set up all the TCB's. Then we need to loop to initialize the thread queue for each process using the `tqueue_init_at_id` function.

For the `tqueue_enqueue` function we will need to use the following five functions in the `import.h` file.

```
void tqueue_set_head(unsigned int chid, unsigned int head);
unsigned int tqueue_get_tail(unsigned int chid);
void tqueue_set_tail(unsigned int chid, unsigned int tail);
void tcb_set_prev(unsigned int pid, unsigned int prev_pid);
void tcb_set_next(unsigned int pid, unsigned int next_pid);
```

This function inserts the TCB #pid into the tail of the thread queue #chid. We first call `tqueue_get_tail` to get the index of the tail. If the queue is empty then both the head and tail will be set to `NUM_IDS`. In this case, we need to initialize the head. Otherwise we need to insert using the `tcb_set_prev` and `tcb_set_next` functions. In both cases we need to update the tail.

For the `tqueue_dequeue` function we will need to use the following five functions in the `import.h` file.

```
unsigned int tqueue_get_head(unsigned int chid);
void tcb_set_prev(unsigned int pid, unsigned int prev_pid);
unsigned int tcb_get_next(unsigned int pid);
void tcb_set_next(unsigned int pid, unsigned int next_pid);
void tqueue_set_head(unsigned int chid, unsigned int head);
```

This function implements the pop functionality for our thread queue. Note: set the "next" of our removed element to `NUM_IDS` after it has been removed. You will need to handle the case where the queue is empty, it should return `NUM_IDS`.

For the `tqueue_remove` function we will need to use the following six functions in the `import.h` file.

```
unsigned int tcb_get_prev(unsigned int pid);
unsigned int tcb_get_next(unsigned int pid);
void tcb_set_prev(unsigned int pid, unsigned int prev_pid);
void tcb_set_next(unsigned int pid, unsigned int next_pid);
void tqueue_set_head(unsigned int chid, unsigned int head);
void tqueue_set_tail(unsigned int chid, unsigned int tail);
```

This function should remove an element from a thread queue. It will need to update its neighboring elements so they reference each other. You will need to handle the case of removing the head or the tail.

The PCurID Layer

In this layer, we introduce the current thread id that records the current running thread id. The code is provided in `kern/thread/PCurID/PCurID.c`.

Exercise 5: The PThread Layer

In this layer, you are going to implement a function to spawn a new kernel thread, or to yield to another kernel thread. Please make sure you read all the comments carefully.

In the file `kern/thread/PThread/PThread.c`, you must implement all the functions listed below:

- `thread_spawn`
- `thread_yield`

For the `thread_spawn` function we will need to use the following three functions in the `import.h` file.

```
unsigned int kctx_new(void *entry, unsigned int id, unsigned int quota);
void tcb_set_state(unsigned int pid, unsigned int state);
void tqueue_enqueue(unsigned int chid, unsigned int pid);
```

This function should do three things: 1) set up a new context for this thread, 2) set the state to "ready to run" or `TSTATE_READY`, and 3) enqueue the new thread into the ready queue. The ready queue has a special id of `NUM_IDS` which does not correspond to any thread.

For the `thread_yield` function we will need to use the following functions in the `import.h` file.

```
unsigned int get_curid(void);
```

```
void set_curid(unsigned int curid);
void tcb_set_state(unsigned int pid, unsigned int state);
void tqueue_enqueue(unsigned int chid, unsigned int pid);
unsigned int tqueue_dequeue(unsigned int chid);
void kctx_switch(unsigned int from_pid, unsigned int to_pid);
```

This function should take the following steps:

1. Get the id of the currently running thread
2. Set the state of the current thread to ready (`TSTATE_READY`).
3. Enqueue the thread into the ready queue
4. Dequeue the next ready thread
5. Set its state to running (`TSTATE_RUN`).
6. Set the new thread as the current id.
7. Execute the context switch.

The PProc Layer

In this layer, we introduce the functions to create a user level process.

In the file `kern/proc/PProc/PProc.c`, you must implement:

- `proc_start_user`

This function sets up the `tss` and page pables to switch to user level from the kernel level.

Function `proc_create` is already implemented for you. Please make sure you understand it.

Part 2: Trap Handling

At this point, the first `int $0x30` system call instruction in user space is a dead end. You will now need to implement basic exception and system call handling, so that it is possible for the kernel to recover control of the processor from user-mode code.

Handling Page Faults

The page fault exception, interrupt vector 14 (`T_PGFLT`), is a particularly important one. When the processor takes a page fault, it stores the linear (i.e., virtual) address that caused the fault in a special processor control register, `CR2`. In `trap/TTrapHandler/TTrapHandler.c` we have provided the implementation of `pgflt_handler(),` to handle page fault exceptions.

System calls

User processes ask the kernel to do things for them by invoking system calls. When the user process invokes a system call, the processor enters kernel mode, the processor and the kernel cooperate to save the user process' state, the kernel executes appropriate code in order to carry out the system call, and then resumes the user process. The exact details of how the user process gets the kernel's attention and how it specifies which call it wants to execute vary from system to system.

In our kernel, we will use the `int` instruction, which causes a processor interrupt. In particular, we will use `int $0x30` as the system call interrupt. We have defined the constant `T_SYSCALL` to 48 (0x30) for you. You will have to set up the interrupt descriptor to allow user processes to cause that interrupt.

The application will pass the system call number and the system call arguments in registers. This way, the kernel won't need to grub around in the user environment's stack or instruction stream. The system call number will go in `%eax`, and the arguments (up to five of them) will go in `%ebx`, `%ecx`, `%edx`, `%esi`, and `%edi`, respectively. A system call always returns with an error number via register `%eax`. All valid error numbers are listed in `__error_nr` defined in `kern/lib/syscall.h`. `E_SUCC` indicates success (no errors). A system call can return at most 5 32-bit values via registers `%ebx`, `%ecx`, `%edx`, `%esi` and `%edi`. When the trap happens, we first save the corresponding trap frame (the register values of the user process) into memory (`uctx_pool`), and we restores the register values based on the saved ones.

In this part of the assignment, you will implement:

1. User level library function for spawn and yield system calls, following the calling conventions above.
2. Various kernel functions to handle the user level requests inside the kernel.

Exercise 6: System call Library

The implementation of the system call library in the user level, following the calling conventions above, is in `user/include/syscall.h`. A sample implementation for `sys_puts` system call is provided. Implement the remaining system calls following the calling conventions mentioned above.

- `sys_spawn`
- `sys_yield`

```
// Code segment from user/include/syscall.h …

static gcc_inline void
sys_puts(const char *s, size_t len)
{
      asm volatile("int %0":
                  : "i" (T_SYSCALL),
                    "a" (SYS_puts),
                    "b" (s),
                    "c" (len)
                  : "cc", "memory");
}
```

This code segment demonstrates how a user process initiates a system call. The `%eax` is loaded with `SYS_puts`, the `%ebx` is loaded with `s` (pointer to char array to print) and the `%ecx` is loaded with `len` (length of the char array). Then the `int T_SYSCALL` is executed to start the kernel side. The processor then transfers execution to the appropriate SYSCALL handler. Please review Part 0 if necessary.

Exercise 7: The TSyscallArg Layer

In this layer, you are going to implement the functions that retrieves the arguments from the user context, and ones that sets the error number and return values back to the user context, based on the calling convention described above.

In the file `kern/trap/TSyscallArg/TSyscallArg.c`, you must implement all the functions listed below. Note that syscall_get_arg1 should return the system call number (`%eax`), not the actual first argument of the function (`%ebx`).

- `syscall_get_arg1`
- `syscall_get_arg2`
- `syscall_get_arg3`
- `syscall_get_arg4`
- `syscall_get_arg5`
- `syscall_get_arg6`
- `syscall_set_errno`
- `syscall_set_retval1`
- `syscall_set_retval2`
- `syscall_set_retval3`
- `syscall_set_retval4`
- `syscall_set_retval5`

These functions should be very simple lookups into `uctx_pool` (array of user contexts) based on the return value of `get_curid`. See `trap.h` below for the type definition of `uctx_pool`.

```
// Code segment from trap.h ...

typedef
struct pushregs {
      uint32_t edi;
      uint32_t esi;
      uint32_t ebp;
      uint32_t oesp;          /* Useless */
      uint32_t ebx;
      uint32_t edx;
      uint32_t ecx;
      uint32_t eax;
} pushregs;

typedef
struct tf_t {
      /* registers and other info we push manually in trapasm.S */
      pushregs regs;
      uint16_t es;      uint16_t padding_es;
      uint16_t ds;      uint16_t padding_ds;
      uint32_t trapno;

      /* format from here on determined by x86 hardware architecture */
      uint32_t err;
      uintptr_t eip;
      uint16_t cs;      uint16_t padding_cs;
      uint32_t eflags;

      /* rest included only when crossing rings, e.g., user to kernel */
      uintptr_t esp;
      uint16_t ss;      uint16_t padding_ss;
} tf_t;
```

Now from here we can see that the `uctx_pool` array has a `regs` field for each element. This `regs` field has a number of fields, one for each register we will need in our calling convention. Each “get” function we write in this layer will retrieve the corresponding field of `regs` in the `uctx_pool` array. Each “set” function will set the corresponding field of `regs`.

Exercise 7: The TSyscall Layer

In the file `kern/trap/TSyscall/TSyscall.c`, you must correctly implement all the functions listed below:

- `sys_spawn`
- `sys_yield`

For the `sys_spawn` function we will need to use the following five functions in the `import.h` file.

```
unsigned int syscall_get_arg2(void);
unsigned int syscall_get_arg3(void);
void syscall_set_errno(unsigned int errno);
void syscall_set_retval1(unsigned int retval);
unsigned int proc_create(void *elf_addr, unsigned int);
```

The goal of this toy system call is to introduce you to how an program could be loaded into memory.

In the `user/include/syscall.h` file we can see that this function has two arguments `exec` and `quota`:

```
// Code segment from user/include/syscall.h …

static gcc_inline pid_t
sys_spawn(uintptr_t exec, unsigned int quota)
{
      // TODO
}
```

This code segment should initiate a system call. The registers will be loaded with `exec` and the `quota` based on the calling convention.

Now to write the kernel side, `sys_spawn` function needs to use the `syscall_get_N` functions to retrieve the `exec` and `quota` arguments. Then based on the number of `exec` we should call `proc_create` based on this table.

exec	Elf to load
1	_binary___obj_user_pingpong_ping_start
2	_binary___obj_user_pingpong_pong_start
3	_binary___obj_user_pingpong_ding_start

On successfully loading the process we should set the errno to `E_SUCC` and if an invalid `exec` is given use `E_INVAL_PID`. The return value should be the ID of the child returned by `proc_create`.

The `sys_yield` system call is much simpler. You will need to use the following two functions in the `import.h` file.

```
void thread_yield(void);
void syscall_set_errno(unsigned int errno);
```

Exercise 8: The TDispatch Layer

In the file `kern/trap/TDispatch/TDispatch.c`, you must correctly implement the function that dispatches the system call requests to appropriate handlers we have implemented in the previous layer, based on the system call number passed in the user context.

- `syscall_dispatch`

Exercise 9: The TTrapHandler Layer

In the file `kern/trap/TTrapHandler/TTrapHandler.c`, we will begin to handle all traps/interrupts/exceptions.

- `trap`

This function will be called when the hardware generates an exception or interrupt as well as when we trigger a software interrupt. The main task of this function are:

1. Save the current trap frame into uctx_pool
2. Switch to the appropriate page structure
3. Dispatch the trap/exception/interrupt to appropriate handlers.
4. After the handler is done, the call to `proc_start_user` will return to the user space via the `trap_return` function.

The `interrupt_handler` and `exception_handler` functions are provided to you. Please read through them.
System call needs further dispatching based on the syscall number and was implemented in `syscall_dispatch` in TDispatch layer.

See `intr.h` for the possible values of the trapno field.

Hand in Procedure

You will submit your assignment via git by creating a tag. The following instructions will explain the needed steps. To get started first change into the root directory of your repository.

Git: Adding Changes AND Git: Committing Changes

These steps are same are same as the previous project. Please `git add` and `git commit` all your changes.

Git: Push Branch: FIRST TIME ONLY

This is a checkpointing step and submits your local branch and commit history to the remote git server. Doing this means your changes will not be lost.
First time push for the new project 2 branch:

```
$ git push -u origin project2
```

The additional -u flag has to be used for the first time you push the branch. This sets the proper remote tracking branch.
Subsequent push:

```
$ git push origin project2
```

Submitting: Creating a tag

```
$ git tag -a v2 -m "Submission for project 2."
```

This will create a `tag` named `v2` from the latest commit on your local.

Submitting: Pushing the tag

```
$ git push origin v2
```

This will push the `tag` named `v2` to the remote repository. Note that this step is different from "Git: Push Branch" step above.

Re-Submission

If you make changes after creating the tag, you will have to forcefully update the tag and (if already pushed) forcefully push the tag.

Follow steps for Adding Changes, Committing Changes and Push Branch as above.

Re-Submitting: Updating the tag

```
$ git tag -a -f v2 -m "Re-Submission for project 2."
```

This will forcefully update the existing tag named `v1` from the latest commit on your local.

Re-Submitting: Forcefully Pushing the tag

```
$ git push -f origin v2
```

This will forcefully push the `tag` named `v2` to the remote repository.