COP 6611: Operating Systems

Project 2 - Process Management and Trap Handling

Due Date: 11/14/2017

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

Execute the following instructions if you want to retrieve a fresh copy of the assignment. `git clone` will create a new folder for you named with your USF NET ID. As always make sure you are connected to the VPN, and use your USF NETID and Password.

```
$ git clone [YOUR NETID]@osnode01.csee.usf.edu:/home/os-student-repos/[YOUR NETID]

$ cd [YOUR NETID]

$ git checkout -b project2 origin/project2
```

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
```

While writing your code, use these: command 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!

Normal Execution

Building the image:

```
$ make clean

$ make
```

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 I 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. In the next assignment, we will learn how to utilize the timer hardware interrupt to allow the kernel to preempt a running process, forcefully retaking control of the CPU from it.

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 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 kernel service thread based on the scheduler.
- Then the resumed kernel thread restores the process states, switches the page structure to the corresponding process's page structure, and then jumps into to 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, i.e., the `eip` you need to read from or write to.

The second step is to set each of these registers using a different process's 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 thread, or to yield to another 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`.

For the `thread_yield` function we will need to use the following three 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. Please make sure you understand all the code we provide in `kern/proc/PProc/PProc.c`

Part 2: Trap Handling

At this point, the first `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. 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. The first thing you should do is thoroughly familiarize yourself with the x86 interrupt and exception mechanism. If you haven't read Chapter 9 in the 80386 Intel Programmer's Manual do so now.

Exercise 6: The TSyscallArg Layer

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 table (IDT) to allow user processes to cause that interrupt.

The Calling Convention:

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.

The implementation of the system call library in the user level, following the calling conventions above, can be found in `user/include/`syscall`.h`. In this part of the assignment, you will implement various kernel functions to handle the user level requests inside the kernel.

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)
{
      int errno;
      pid_t pid;

      asm volatile("int %2"
                 : "=a" (errno),
                   "=b" (pid)
                 : "i" (T_SYSCALL),
                   "a" (SYS_spawn),
                   "b" (exec),
                   "c" (quota)
                 : "cc", "memory");

      return errno ? -1 : pid;
}
```

This code segment demonstrates how a user process initiates a system call. The `%eax` is loaded with `SYS_spawn`, the `%ebx` is loaded with `exec` and the `%ecx` is loaded with `quota`. Then the `int T_SYSCALL` is executed to start the kernel side.

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
4	_binary___obj_user_fork_fork_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);
```

The TDispatch Layer

This layer implements 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. Make sure you fully understand the code in `kern/trap/TDispatch/TDispatch.c`.

Exercise 8: The TTrapHandler Layer

In the file `kern/trap/TTrapHandler/TTrapHandler.c`, carefully review the implementation of the function trap, then implement all the functions listed below:

- `exception_handler`
- `interrupt_handler`

These functions will be called if the hardware generates an exception or interrupt. In form they are very similar.

For the `exception_handler` function we will need to use the following three functions:

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

You will need to lookup the current process id in `uctx_pool` and access the `trapno` field to determine what exception is being triggered. Since we currently only support the page fault exception (`T_PGFLT`), send everything else to the default exception handler.

For the `interrupt_handler` function we will need to use the following four functions:

```
unsigned int get_curid(void);
static int spurious_intr_handler (void);
static int timer_intr_handler (void);
static int default_intr_handler (void);
```

You will need to lookup the current process id in `uctx_pool` and access the `trapno` field to determine what interrupt is being triggered.

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

```
// Code segment from kern/dev/intr.h …

// Hardware IRQ numbers. We receive these as (T_IRQ0 + IRQ_WHATEVER)
/* (32 ~ 47) ISA interrupts: used by i8259 */
/* (48 ~ 55) reserved for IOAPIC extended interrupts */
#define T_IRQ0          32 /* Legacy ISA hardware interrupts: IRQ0-15. */
#define IRQ_TIMER       0  /* 8253 Programmable Interval Timer (PIT) */
#define IRQ_KBD         1  /* Keyboard interrupt */
#define IRQ_SLAVE       2  /* cascaded to slave 8259 */
#define IRQ_SERIAL24    3  /* Serial (COM2 and COM4) interrupt */
#define IRQ_SERIAL13    4  /* Serial (COM1 and COM4) interrupt */
#define IRQ_LPT2        5  /* Parallel (LPT2) interrupt */
#define IRQ_FLOPPY      6  /* Floppy interrupt */
#define IRQ_SPURIOUS    7  /* Spurious interrupt or LPT1 interrupt */
#define IRQ_RTC         8  /* RTC interrupt */
#define IRQ_MOUSE       12 /* Mouse interrupt */
#define IRQ_COPROCESSOR 13 /* Math coprocessor interrupt */
#define IRQ_IDE1        14 /* IDE disk controller 1 interrupt */
#define IRQ_IDE2        15 /* IDE disk controller 2 interrupt */
#define IRQ_EHCI_1      16
#define IRQ_ERROR       19
#define IRQ_EHCI_2      23
```

We don't need to handle all of these, just spurious and timer for now. Otherwise just call the default interrupt handler.

Hand in Procedure

You will submit your assignment via git. The following instructions will explain the needed steps. Make sure you are connected to the USF VPN (using Junos Pulse). To get started first change into the root directory of your os-class-node.repo folder.

Submitting: adding changes

```
$ git add .
```

This will add the changes that you have made to these files to the next *commit*.

Submitting: committing changes

```
$ git commit -m "project 2 hand in"
```

This step will create a commit of your current changes. If you want to make multiple commits while developing, that's fine, we will just grade the latest commit in your submission branch.

Submitting: push branch

```
$ git push origin project2
```

This is the final, and most important step in the process. This push submits your local branch and commit history to our server.