Section 4: Threads and Context Switching

CS162

September 19 - 20, 2017

Contents

1 Warmup

1.1 Hello World

What does C print in the following code?

```
void* identify(void* arg) {
    pid_t pid = getpid();
    printf("My pid is %d\n", pid);
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &identify, NULL);
    identify(NULL);
    return 0;
}
```

```
My pid is 2617
My pid is 2617
```

2 Vocabulary

- **thread** - a thread of execution is the smallest unit of sequential instructions that can be scheduled for execution by the operating system. Multiple threads can share the same address space, but each thread independently operates using its own program counter.
- **pthreads** - A POSIX-compliant (standard specified by IEEE) implementation of threads. A `pthread_t` is usually just an alias for "`unsigned long int`".
- **pthread_create** - Creates and immediately starts a child thread running in the same address space of the thread that spawned it. The child executes starting from the function specified. Internally, this is implemented by calling the clone syscall.

  ```
  /* On success, pthread_create() returns 0; on error, it returns an error
   * number, and the contents of *thread are undefined. */
  int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                     void *(*start_routine) (void *), void *arg);
  ```

- **pthread_join** - Waits for a specific thread to terminate, similar to `waitpid(3)`.

  ```
  /* On success, pthread_join() returns 0; on error, it returns an error number. */
  int pthread_join(pthread_t thread, void **retval);
  ```

- **pthread_yield** - Equivalent to thread_yield() in Pintos. Causes the calling thread to vacate the CPU and go back into the ready queue without blocking. The calling thread is able to be scheduled again immediately. This is not the same as an interrupt and will succeed in Pintos even if interrupts are disabled.

  ```
  /* On success, pthread_yield() returns 0; on error, it returns an error number. */
  int pthread_yield(void);
  ```

3 Problems

3.1 Join

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_yield();
    printf("MAIN\n");
    return 0;
}
```

The output of this program could be "MAIN\nHELPER\n", "HELPER\nMAIN\n" or "MAIN\n".
The actual order could be different each time the program is run.
First, the pthread_yield() does not change the answer, because it provides no guarantee about what order the print statements execute in.
Second, the helper thread may be preempted at any point (e.g., before or after running printf()).
Last, the main() function can return without giving enough time for the helper thread to run, killing the process and all associated threads.

How can we modify the code above to always print out "HELPER" followed by "MAIN"?

Change pthread_yield to pthread_join.

```
void *helper(void *arg) {
    printf("HELPER\n");
    return NULL;
}

int main() {
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, NULL);
    pthread_join(thread, NULL);
    printf("MAIN\n");
    return 0;
}
```

3.2 Stack Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    int *num = (int*) arg;
    *num = 2;
    return NULL;
}

int main() {
    int i = 0;
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, &i);
    pthread_join(thread, NULL);
    printf("i is %d\n", i);
    return 0;
}
```

The spawned thread shares the address space with the main thread and has a pointer to the same memory location, so i is set to 2. "i is 2"

3.3 Heap Allocation

What does C print in the following code?

```
void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    return NULL;
}

int main() {
    char *message = malloc(100);
    strcpy(message, "I am the parent");
    pthread_t thread;
    pthread_create(&thread, NULL, &helper, message);
    pthread_join(thread, NULL);
    printf("%s\n", message);
    return 0;
}
```

"I am the child"

3.4 Threads and Processes

What does C print in the following code?
(Hint: There may be zero, one, or multiple answers.)

```
void *worker(void *arg) {
    int *data = (int *) arg;
    *data = *data + 1;
    printf("Data is %d\n", *data);
    return (void *) 42;
}

int data;
int main() {
    int status;
    data = 0;
    pthread_t thread;

    pid_t pid = fork();
    if (pid == 0) {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
    } else {
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        pthread_create(&thread, NULL, &worker, &data);
        pthread_join(thread, NULL);
        wait(&status);
    }
    return 0;
}
```

One of the following is printed out:

"Data is 1"
"Data is 1"
"Data is 2"

"Data is 1"
"Data is 2"
"Data is 1"

How would you retrieve the return value of worker? (e.g. "42")

You can use the 2nd argument of pthread_join. For example:

```
int return_value;
pthread_join(thread, &return_value);
```

3.5 Context Switching

Refer to the "Pintos Context Switch" section at the end of this discussion worksheet to answer these questions:

How many stacks are involved in a context switch? Identify the purpose of each stack.

```
There are 2 stacks: the kernel stack of the current thread (shares a single page with
the TCB) and the kernel stack of the next thread.
```

The value of `SWITCH_CUR` is 20. The value of `SWITCH_NEXT` is 24. With this information, please draw the stack frame of `switch_threads` for a thread that is about to switch the stack pointer to the next thread's stack. Your stack frame should include the arguments `cur` and `next`.

```
pointer to next thread struct "next" (+24)
pointer to current thread struct "cur" (+20)
the return address (+16)
ebx (+12)
ebp (+8)
esi (+4)
edi (+0)
```

In addition to the code inside `switch_threads`, what other actions are required to perform a context switch between 2 **user program threads**?

```
The most important missing thing to switch is the page table. A new user program
thread could have a different virtual address space.

You might also mention CPU flags or segment registered. However, it's possible
that all kernel threads share the same values for these. The interrupt service
routine (ISR) can restore those registers when it returns to the user program.
```

In order to perform a context switch, the kernel must copy all of a thread's registers onto the CPU's registers. How is the `%eip` (instruction pointer) register copied onto the CPU? Identify which instruction is responsible for this.

```
The "ret" instruction means "pop 4 bytes off the stack and jump to that
location". This is the instruction that applies the thread's %eip, because
the return address would be some code in the parent frame of switch_threads.
```

3.6 Interrupt Handlers

Refer to the "Pintos Interrupt Handler" section at the end of this discussion worksheet to answer these questions:

What do the instructions `pushal` and `popal` do?

They push and pop all the general-purpose 32-bit x86 registers onto/from the stack.

The interrupt service routine (ISR) must run with the kernel's stack. Why is this the case? And which instruction is responsible for switching the stack pointer to the kernel stack?

The user program's stack pointer may be invalid, or the user program could be using memory below the stack pointer. The CPU itself will switch the stack to the kernel stack (either because of an external interrupt, a trap, or a programmed interrupt). We do not need to write an instruction in the ISR to do this.

The `pushal` instruction pushes 8 values onto the stack (32 bytes). With this information, please draw the stack at the moment when "`call intr_handler`" is about to be executed.

ds
es
fs
gs
pushal's 8 general purpose registers
pointer to (%esp + 4)

What is the purpose of the "`pushl %esp`" instruction that is right before "`call intr_handler`"?

It is a pointer to the part of the stack that contains all the registers. In pintos, this is accessed as the "intr_frame" struct.

Inside the `intr_exit` function, what would happen if we reversed the order of the 5 `pop` instructions?

The pop instructions need to be in their current order. They are exactly the reverse order of the corresponding push instructions, because our stack is First-In-Last-Out.

3.7 Pintos Context Switch

```
#### struct thread *switch_threads (struct thread *cur, struct thread *next);
####
#### Switches from CUR, which must be the running thread, to NEXT,
#### which must also be running switch_threads(), returning CUR in
#### NEXT's context.
####
#### This function works by assuming that the thread we're switching
#### into is also running switch_threads().  Thus, all it has to do is
#### preserve a few registers on the stack, then switch stacks and
#### restore the registers.  As part of switching stacks we record the
#### current stack pointer in CUR's thread structure.

.globl switch_threads
.func switch_threads
switch_threads:
        # Save caller's register state.
        #
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
        # [SysV-ABI-386] pages 3-11 and 3-12 for details.
        #
        # This stack frame must match the one set up by thread_create()
        # in size.
        pushl %ebx
        pushl %ebp
        pushl %esi
        pushl %edi

        # Get offsetof (struct thread, stack).
.globl thread_stack_ofs
        mov thread_stack_ofs, %edx

        # Save current stack pointer to old thread's stack, if any.
        movl SWITCH_CUR(%esp), %eax
        movl %esp, (%eax,%edx,1)

        # Restore stack pointer from new thread's stack.
        movl SWITCH_NEXT(%esp), %ecx
        movl (%ecx,%edx,1), %esp

        # Restore caller's register state.
        popl %edi
        popl %esi
        popl %ebp
        popl %ebx
        ret
.endfunc
```

3.8 Pintos Interrupt Handler

```
/**
* An example of an entry point that would reside in the interrupt
* vector. This entry point is for interrupt number 0x30.
*/
.func intr30_stub
intr30_stub:
      pushl %ebp     /* Frame pointer */
      pushl $0       /* Error code */
      pushl $0x30    /* Interrupt vector number */
      jmp intr_entry
.endfunc
/* Main interrupt entry point.

   An internal or external interrupt starts in one of the
   intrNN_stub routines, which push the 'struct intr_frame'
   frame_pointer, error_code, and vec_no members on the stack,
   then jump here.

   We save the rest of the 'struct intr_frame' members to the
   stack, set up some registers as needed by the kernel, and then
   call intr_handler(), which actually handles the interrupt.

   We "fall through" to intr_exit to return from the interrupt.
*/
.func intr_entry
intr_entry:
        /* Save caller's registers. */
        pushl %ds
        pushl %es
        pushl %fs
        pushl %gs
        pushal

        /* Set up kernel environment. */
        cld                     /* String instructions go upward. */
        mov $SEL_KDSEG, %eax    /* Initialize segment registers. */
        mov %eax, %ds
        mov %eax, %es
        leal 56(%esp), %ebp     /* Set up frame pointer. */

        /* Call interrupt handler. */
        pushl %esp
.globl intr_handler
        call intr_handler
        addl $4, %esp
.endfunc
```

```
48 /* Interrupt exit.
49
50    Restores the caller's registers, discards extra data on the
51    stack, and returns to the caller.
52
53    This is a separate function because it is called directly when
54    we launch a new user process (see start_process() in
55    userprog/process.c). */
56 .globl intr_exit
57 .func intr_exit
58 intr_exit:
59         /* Restore caller's registers. */
60         popal
61         popl %gs
62         popl %fs
63         popl %es
64         popl %ds
65
66         /* Discard `struct intr_frame' vec_no, error_code,
67            frame_pointer members. */
68         addl $12, %esp
69
70         /* Return to caller. */
71         iret
72 .endfunc
```