Section 4

Thursday, April 25, 2019 6:47 PM

section4

Section 4: Threads and Context Switching

CS162

September 18 - 19, 2018

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;
}
```

Takes a generic pointer

allocate a pointer to variable thread

Pointer to a pointer function

Pass pointer to thread value.

Prints my pid is something, from same pids. Or once if main function exits

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;
}
```

HELPER\nMAIN\n. MAIN\nHELPER\n

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

Pthread_join(thread);

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;
}
```

Return generic pointer

Cast into integer pointer

Store 2 at memory location that *num refers to.

Store reference here

I is 2, we wait for the helper thread to finish, which stores 2 in variable num

3.3 Heap Allocation

What does C print in the following code?

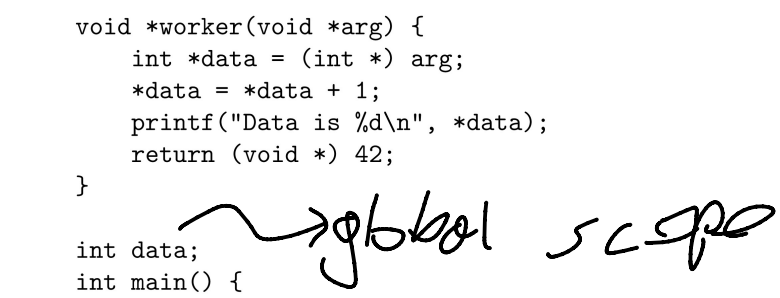
```
void *helper(void *arg) {
    char *message = (char *) arg;
    strcpy(message, "I am the child");
    return NULL;
}

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;
}
```

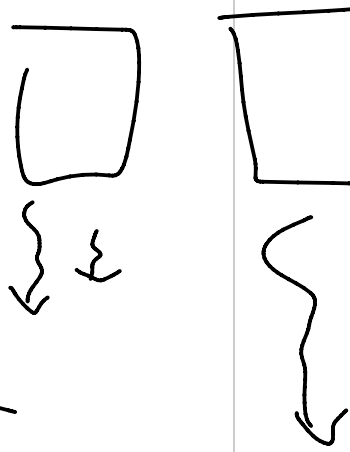
I am the child

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;
}
```





1	1
1	2
2	1 (late child)

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

Replace NULL with &&someInt

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.

KERNEL stack of current thread, KERNEL stack of 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`.

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

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.

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?

Push and pop all general uprpose registers

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?

Why ISR run on KERNEL stack? Needs to disable ints to do work?
Which instruction handles switch of SP to Kstack? Addl $4 %esp

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
gf
Eax
Ecx
Edx
Ebc
Temp
Ebp
Esi
edi

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

Debug info?

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

Thigns save in the wrong place
chaos

3.7 Pintos Context Switch

```
3 #### struct thread *switch_threads (struct thread *cur, struct thread *next);
4 ####
5 #### Switches from CUR, which must be the running thread, to NEXT,
6 #### which must also be running switch_threads(), returning CUR in
7 #### NEXT's context.
8 ####
9 #### This function works by assuming that the thread we're switching
10 #### into is also running switch_threads().  Thus, all it has to do is
11 #### preserve a few registers on the stack, then switch stacks and
12 #### restore the registers.  As part of switching stacks we record the
13 #### current stack pointer in CUR's thread structure.
14
15 .globl switch_threads
16 .func switch_threads
17 switch_threads:
18         # Save caller's register state.
19         #
20         # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
21         # but requires us to preserve %ebx, %ebp, %esi, %edi.  See
22         # [SysV-ABI-386] pages 3-11 and 3-12 for details.
23         #
24         # This stack frame must match the one set up by thread_create()
25         # in size.
26         pushl %ebx
27         pushl %ebp
28         pushl %esi
29         pushl %edi
30
31         # Get offsetof (struct thread, stack).
32 .globl thread_stack_ofs
33         mov thread_stack_ofs, %edx
34
35         # Save current stack pointer to old thread's stack, if any.
36         movl SWITCH_CUR(%esp), %eax
37         movl %esp, (%eax,%edx,1)
38
39         # Restore stack pointer from new thread's stack.
40         movl SWITCH_NEXT(%esp), %ecx
41         movl (%ecx,%edx,1), %esp
42
43         # Restore caller's register state.
44         popl %edi
45         popl %esi
46         popl %ebp
47         popl %ebx
48         ret
49 .endfunc
```

Switch threads symbol exported to where it points in object code generated

Application Binary Interface -
Specifications for calling conventions, obj file formats, exec file for

Lcaol reg
Stack base pointer
Local reg
Local reg

Copy dat from thread_stack_fs to %edx

3.8 Pintos Interrupt Handler

```
1 /**
2 * An example of an entry point that would reside in the interrupt
3 * vector. This entry point is for interrupt number 0x30.
4 */
```

called function
ling function.

Figure 3-14: Proc

Type

General

Floating-po

In addition to re
stack grows dow
organization.

Figure 3-15: Star

Position
4*n*+8(%ebp
8(%ebp
4(%ebp
0(%ebp
-4(%ebp
0(%esp

mats, semantics, etc. used my major Unix OS

. Registers %ebp, %ebx, %edi, %esi, and %esp "belong" to the cal-
In other words, a called function must preserve these registers'

cessor Registers

	Name	Usage
	%eax	Return value
	%edx	Dividend register (divide operations)
	%ecx	Count register (shift and string operations)
	%ebx	Local register variable
	%ebp	Stack frame pointer (optional)
	%esi	Local register variable
	%edi	Local register variable
	%esp	Stack pointer
	%st(0)	floating-point stack top, return value
oint	%st(1)	floating-point next to stack top
	. . .	
	%st(7)	floating-point stack bottom

G

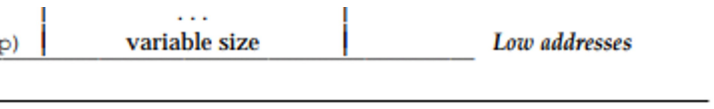
gisters, each function has a frame on the run-time stack. This
nward from high addresses. Figure 3-15 shows the stack frame

ndard Stack Frame

	Contents	Frame	
p)	argument word *n*	Previous	*High addresses*
	. . .		
p)	argument word 0		
p)	return address	Current	
p)	previous %ebp (optional)		
p)	unspecified		
	. . .		
p)	variable size		*Low addresses*

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
```


p)
. . .
variable size
Low addresses

```
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
```