CS 3204
Operating Systems

Lecture 5

Godmar Back

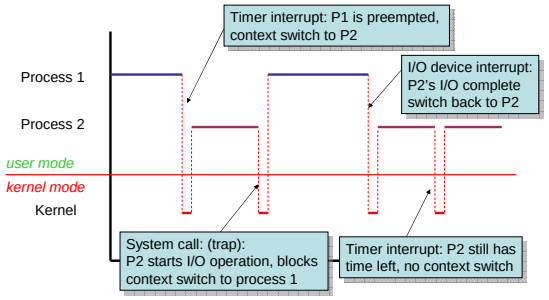
Announcements

- Posted stack.cc example on website
- Project 0 is due Feb 1, 11:59pm
- This project is done individually
- Curator instructions will be posted on website

Overview

- Definitions
- How does OS execute processes?
  - How do kernel & processes interact
  - How does kernel switch between processes
- Implementing Processes
- Next:
  - Scheduling of threads
  - Synchronization of threads

Context Switching



Switching Procedures

- Inside kernel, context switch is implemented in schedule() function called from C code
  - Called when kernel decides to context switch
- Must understand how to switch procedures (call/return)
- Dictated by procedure calling conventions
  - Architecture-specific
  - Part of ABI (application binary interface), implemented by compiler
  - Pintos uses SVR4 ABI

x86 Calling Conventions

- Caller saves caller-saved registers as needed
- Caller pushes arguments, right-to-left on stack via push assembly instruction
- Caller executes CALL instruction: save address of next instruction & jump to callee

- Callee executes:
  - Saves callee-saved registers if they'll be destroyed
  - Puts return value (if any) in eax
- Callee returns via RET instruction: pop return address from stack & jump to it

- Caller resumes: pop arguments off the stack
- Caller restores caller-saved registers, if any

Example

```
int globalvar;

int
callee(int a, int b)
{
   return a + b;
}

int
caller(void)
{
   return callee(5, globalvar);
}
```

```
callee:
        pushl   %ebp
        movl    %esp, %ebp
        movl    12(%ebp), %eax
        addl    8(%ebp), %eax
        leave
        ret

caller:
        pushl   %ebp
        movl    %esp, %ebp
        pushl   globalvar
        pushl   $5
        call    callee
        popl    %edx
        popl    %ecx
        leave
        ret
```


Pintos Context Switch (1)

```
static void
schedule (void)
{
  struct thread *cur = running_thread ();
  struct thread *next = next_thread_to_run ();
  struct thread *prev = NULL;
  if (cur != next)
    prev = switch_threads (cur, next);
retlabel:  /* not in actual code */
  schedule_tail (prev);
}

uint32_t thread_stack_ofs = offsetof (struct thread, stack);
```

- threads/thread.c, threads/switch.S

Pintos Context Switch (2)

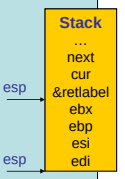
```
switch_threads:
        # Save caller's register state.
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.
        pushl %ebx; pushl %ebp; pushl %esi; pushl %edi

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

        # Save current stack pointer to old thread's stack.
        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
```




```
#define SWITCH_CUR 20
#define SWITCH_NEXT 24
```


Famous Quote For The Day

If the new process paused because it was swapped out, set the stack level to the last call to savu(u_ssav). This means that the return which is executed immediately after the call to aretu actually returns from the last routine which did the savu.

You are not expected to understand this.

- Source: Dennis Ritchie, Unix V6 slp.c (context-switching code) as per The Unix Heritage Society (tuhs.org)

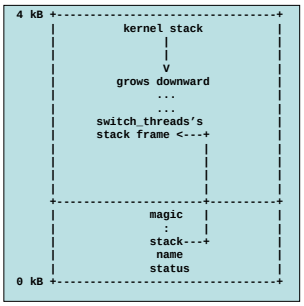
Pintos Context Switch (3)

- All state is stored on outgoing thread's stack, and restored from incoming thread's stack
  - Each thread has a 4KB page for its stack
  - Called "kernel stack" because it's only used when thread executes in kernel mode
  - Mode switch automatically switches to kernel stack
    - x86 does this in hardware, curiously.
- switch_threads assumes that the thread that's switched in was suspended in switch_threads as well.
  - Must fake that environment when switching to a thread for the first time.
- Aside: none of the thread switching code uses privileged instructions:
  - that's what makes user-level threads (ULT) possible

Pintos Kernel Stack

- One page of memory captures a process's kernel stack + PCB
- Don't allocate large objects on the stack:

```
void
kernel_function(void)
{
    char buf[4096]; // DON'T
    // KERNEL STACK OVERFLOW
    // guaranteed
}
```



Context Switching: Summary

- Context switch means to save the current and restore next process's execution context
- Context Switch != Mode Switch
  - Although mode switch often precedes context switch
- Asynchronous context switch happens in interrupt handler
  - Usually last thing before leaving handler
- Have ignored so far when to context switch & why → next

Intermezzo

Just enough on concurrency to get through Project 0

A lot more later.

Concurrency

- Access to shared resources must be mediated
  - Specifically shared (non-stack) variables
- Will hear a lot more about this
- For now, simplest way to protection is mutual exclusion via locks (aka mutexes)
- For Project 0, concurrency is produced by using PThreads (POSIX Threads), so must use PThread's mutexes.
  - Just an API, idea is the same everywhere

pthread_mutex example

```
/* Define a mutex and initialize it. */
static pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

static int counter = 0; /* A global variable to protect. */

/* Function executed by each thread. */
static void *
increment(void *_)
{
    int i;
    for (i = 0; i < 1000000; i++) {
        pthread_mutex_lock(&lock);
        counter++;
        pthread_mutex_unlock(&lock);
    }
}
```