

# CS 3204 Operating Systems

Lecture 4  
Godmar Back



## Announcements

- **Project 0 Jan 29 (Monday) 11:59pm**
- Additional office hours this week – check forum for hours
- Start forming groups
  - (but don't collaborate on project 0!)
- Project 1 help sessions next week (probably Tu+We evening)



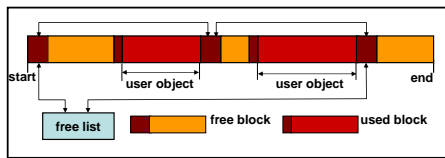
CS 3204 Spring 2007

1/30/2007

2

## Project 0

- Implement User-level Memory Allocator
  - Use address-ordered first-fit



CS 3204 Spring 2007

1/30/2007

3

## Processes & Threads



## Threads vs Processes

- Thread:
  - Execution sequence + CPU state (registers + stack)
- Process:
  - $n$  Threads + Resources shared by them (specifically: accessible heap memory, global variables, file descriptors, etc.)
- In most contemporary OS,  $n \geq 1$ .
- In Pintos,  $n=1$ : a process is a thread – as in traditional Unix.
  - Following discussion applies to both threads & processes.

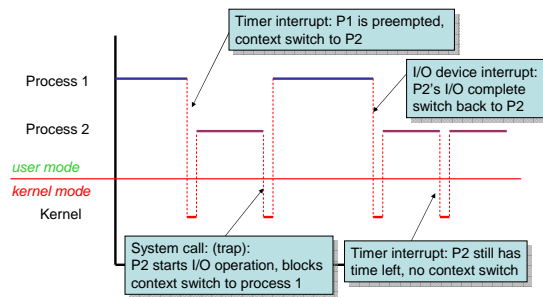


CS 3204 Spring 2007

1/30/2007

5

## Context Switching



CS 3204 Spring 2007

1/30/2007

6

## Mode Switching

- User → Kernel mode
  - For reasons external or internal to CPU
- External (aka hardware) interrupt:
  - timer/clock chip, I/O device, network card, keyboard, mouse
  - asynchronous (with respect to the executing program)
- Internal interrupt (aka software interrupt, trap, or exception)
  - are synchronous
  - can be intended: for system call (process wants to enter kernel to obtain services)
  - or unintended (usually): fault/exception (division by zero, attempt to execute privileged instruction in user mode)
- Kernel → User mode switch on iret instruction

## Context vs Mode Switching

- Mode switch guarantees kernel gains control when needed
  - To react to external events
  - To handle error situations
  - Entry into kernel is controlled
- Not all mode switches lead to context switches
  - Kernel code's logic decides when – subject of scheduling
- Mode switch always hardware supported
  - Context switch (typically) not – this means many options for implementing it!

## Implementing Processes

- To maintain illusion, must remember a process's information when not currently running
- Process Control Block (PCB)
  - Identifier (\*)
  - Value of registers, including stack pointer (\*)
  - Information needed by scheduler: process state (whether blocked or not) (\*)
  - Resources held by process: file descriptors, memory pages, etc.

(\*) applies to TCB (thread control block) as well

## PCB vs TCB

- In 1:1 systems (Pintos), TCB==PCB
  - `struct thread`
  - add information as projects progress
- In 1:n systems
  - TCB contains scheduling information
  - PCB contains process state
  - about resources

```
struct thread
{
    tid_t tid; /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16]; /* Name. */
    uint8_t *stack; /* Saved stack pointer. */
    int priority; /* Priority. */
    struct list_elem elem; /* List element. */
    /* others you'll add as needed. */
};
```

## Steps in context switch: high-level

- Save the current process's execution state to its PCB
- Update current's PCB as needed
- Choose next process N
- Update N's PCB as needed
- Restore N's PCB execution state
  - May involve reprogramming MMU

## Execution State

- Saving/restoring execution state is highly tricky:
  - Must save state without destroying it
- Registers
  - On x86: eax, ebx, ecx, ...
- Stack
  - Special area in memory that holds activation records: e.g., the local (automatic) variables of all function calls currently in progress
  - Saving the stack means retaining that area & saving a pointer to it ("stack pointer" = esp)

## The Stack, seen from C/C++

```
int a;
static int b;
int c = 5;
struct S
{
    int t;
} s;

void func(int d)
{
    static int e;
    int f;
    struct S w;
    int *g = new int[10];
}
```

A.: On stack: d, f, w (including w.t), g  
Not on stack: a, b, c, s (including s.t), e, g[0]...g[9]



CS 3204 Spring 2007

1/30/2007

13

## Switching Procedures

- Inside kernel, context switch is implemented in some procedure (function) called from C code
  - Appears to caller as a procedure call
- Must understand how to switch procedures (call/return)
- Procedure calling conventions
  - Architecture-specific
  - Defined by ABI (application binary interface), implemented by compiler
  - Pintos uses SVR4 ABI



CS 3204 Spring 2007

1/30/2007

14

## 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
  - Caller resumes: pop arguments off the stack
  - Caller restores caller-saved registers, if any
- Callee executes:
- Saves callee-saved registers if they'll be destroyed
  - Puts return value (if any) in eax
- Callee returns: pop return address from stack & jump to it



CS 3204 Spring 2007

1/30/2007

15

## 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
```



CS 3204 Spring 2007

1/30/2007

16

## 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



CS 3204 Spring 2007

1/30/2007

17

## Pintos Context Switch (2)

```
switch_threads: // switch_thread(struct thread *cur, struct thread *next)
# 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

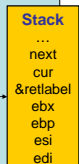
# Get offsetof (struct thread, stack).
movl thread_stack_ofs, %edx

# Save current stack pointer to old thread's stack.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax, %edx, 1) } cur->stack = esp

# Restore stack pointer from new thread's stack.
movl SWITCH_NEXT(%esp), %ecx
movl %ecx, (%ecx, 1), %esp } esp = next->stack

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

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



CS 3204 Spring 2007

1/30/2007

18

## 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](http://The Unix Heritage Society) (tuhs.org); gif by Eddie Koehler.

```
2230 /*
2231  * If the new process paused because it was
2232  * swapped out, set the stack level to the last call
2233  * to savu(u_ssav). This means that the return
2234  * which is executed immediately after the call to aretu
2235  * actually returns from the last routine which did
2236  * the savu.
2237  *
2238  * You are not expected to understand this.
2239  */
```



CS 3204 Spring 2007

1/30/2007

19

## 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



CS 3204 Spring 2007

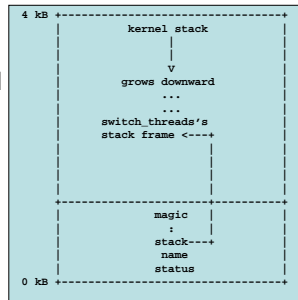
1/30/2007

20

## 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
}
```



CS 3204 Spring 2007

1/30/2007

21

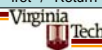
## External Interrupts & Context Switches

```
intr_entry:
/* Save caller's registers. */
pushl %ds; pushl %es; pushl %fs; pushl %gs; pushal

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

pushl %esp
call intr_handler /* Call interrupt handler. Context switch happens in there! */
addl $4, %esp
/* FALL THROUGH */

intr_exit: /* Separate entry for initial user program start */
/* Restore caller's registers. */
popal; popl %gs; popl %fs; popl %es; popl %ds
iret /* Return to current process, or to new process after context switch. */
```



CS 3204 Spring 2007

1/30/2007

22

## 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



CS 3204 Spring 2007

1/30/2007

23