PCB vs TCB

In 1:1 systems (Pintos), TCB==PCB

- **`struct thread`**
- add information there as it gets progress

```
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. */
};
```

In 1:n systems:

- TCB contains execution state of thread + scheduling information + link to PCB for process to which thread belongs
- PCB contains identifier, plus information about resources shared by all threads

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

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];
}
```

Q.: which of these variables are stored on the stack, and which are not?

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]

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

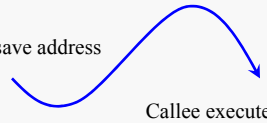
- Architecture-specific
- Defined by ABI (application binary interface), implemented by compiler
- Pintos uses SVR4 ABI

x86 Calling Conventions

Caller saves caller-saved registers as needed

Caller pushes arguments, right-to-left on stack via push assembly instruction

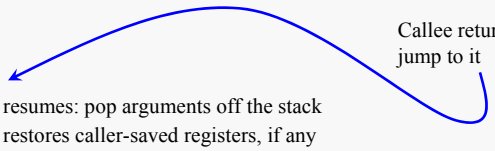
Caller executes CALL instruction: save address of next instruction & jump to callee



Callee executes:

- Saves callee-saved registers if they'll be destroyed
- Puts return value (if any) in eax

Callee returns: 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)

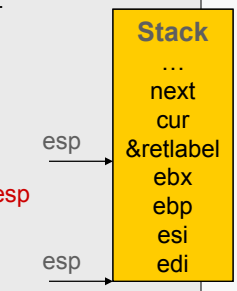
```
switch_threads:  // switch_thread (struct thread *cur, struct thread *next)
        # Save caller's register state.
        # Note that the SVR4 ABI allows us to destroy %eax, %ecx, %edx,
        # but requires us to preserve %ebx, %ebp, %esi, %edi.
        pushl %ebx; pushl %ebp; pushl %esi; pushl %edi

        # Get offsetof (struct thread, stack).
        mov 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,%edx,1), %esp        } esp = next->stack

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




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


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

`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.

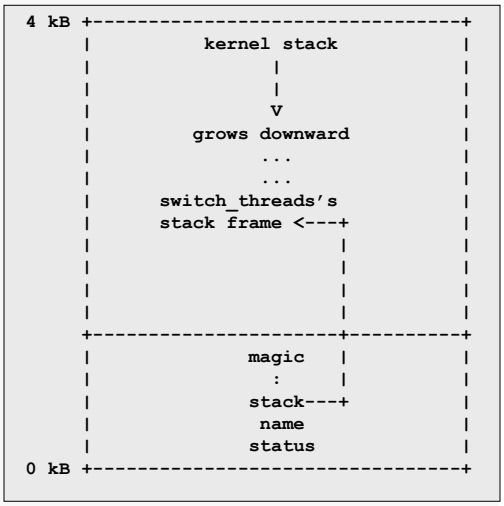
Aside: none of the thread switching code uses privileged instructions:
- that's what makes user-level threads (ULT) possible

Pintos Kernel Stack

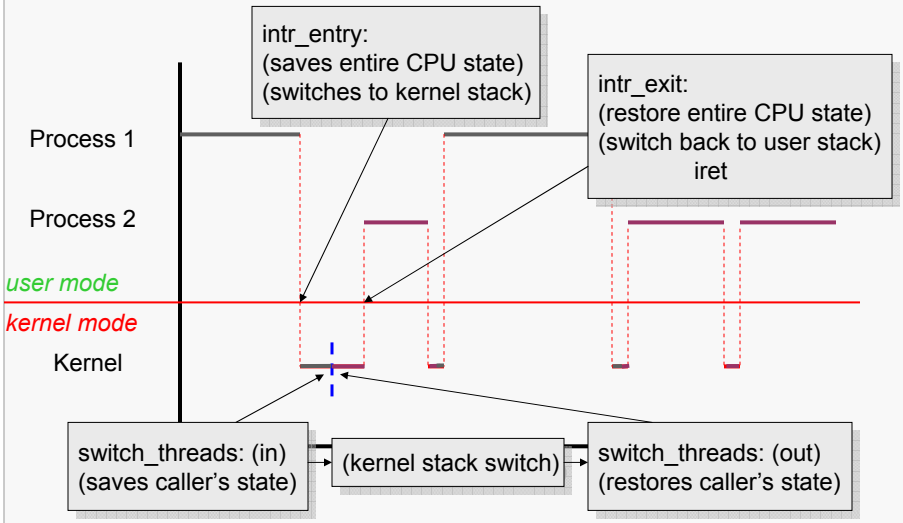
One page of memory captures a process's kernel stack + PCB

Don't allocate large objects on the stack:

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



Context Switching, Take 2



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. */
```


Context Switching: Summary

Context switch means to save the current and restore next process's execution context

Context Switch != Mode Switch

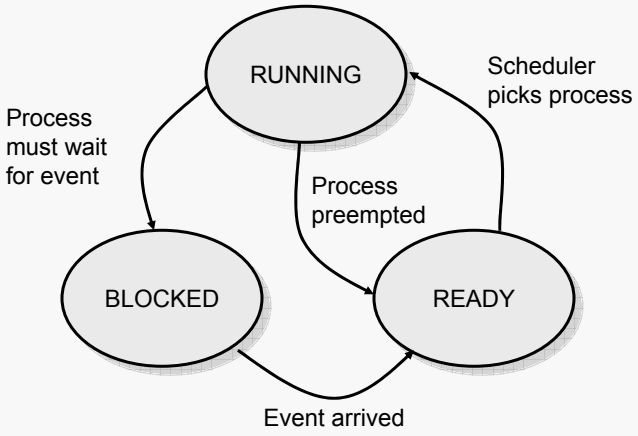
- Although mode switch often precedes context switch

Asynchronous context switch happens in interrupt handler

- Usually last thing before leaving handler

Have ignored so far when to context switch & why → next

Process States



Only 1 process (per CPU) can be in RUNNING state

Process Events

What's an event?

- External event:
  - disk controller completes sector transfer to memory
  - network controller signals that new packet has been received
  - clock has advanced to a predetermined time
- Events that arise from process interaction:
  - a resource that was previously held by some process is now available (e.g., lock_release)
  - an explicit signal is sent to a process (e.g., cond_signal)
  - a process has exited or was killed
  - a new process has been created

Process Lists

All ready processes are inserted in a "ready list" data structure

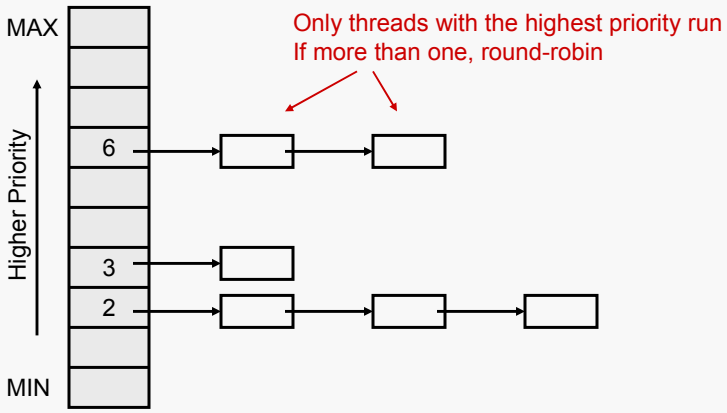
- Running process typically not kept on ready list
- Can implement as multiple (real) ready lists, e.g., one for each priority class

All blocked processes are kept on lists

- List usually associated with event that caused blocking – usually one list per object that's causing events

Most of scheduling involves simple and clever ways of manipulating lists

Priority Based Scheduling



Done in Linux, Windows, Pintos (after you complete Project 1), …

Priority Based Scheduling (2)

Advantage:

- Dead simple: the highest-priority process runs
- Q.: what is the complexity of finding which process that is?

Disadvantage:

- Not fair: lower-priority processes will never run
- Hence, must adjust priorities somehow

All schedulers used in today's general purpose OS work like this

- Only difference is how priorities are adjusted to provide fairness and avoid starvation