Lecture 4: Threads

601.418/618 Operating Systems

David Hovemeyer

February 2, 2026

Agenda

- Threads
- Threading models
- Threads in Pintos

Acknowledgments: These slides are shamelessly adapted from Prof. Ryan Huang's Fall 2022 slides, which in turn are based on Prof. David Mazières's OS lecture notes. They also include some content developed by Prof. Phillipp Koehn for CSF.

Processes

Process state:

- PCB
- Address space
- Open files

Significant work required to create a process or tear down a process.

Switching context from one process to another can be expensive:

- Cost to transition between user mode and kernel mode
- TLB misses, cache misses

For processes to cooperate (interprocess communication), kernel must mediate

- Exception: interprocess communication using shared memory

Concurrency

Concurrency is useful in lots of situations. E.g., network server:

```
while (1) {
  fd = accept(ssock, NULL, NULL);
  pid = fork();
  if (pid == 0) {
    chat_with_client(fd); exit(0);
  } else if (pid > 0)
    close(fd);
}
```

The code we want to execute to communicate with a client (`chat_with_client()`) is part of the server executable.

- And yet each child process has its own address space, its own table of open files, etc.

State of child processes is mostly the same as the parent, but lots of resources are duplicated.

When using processes for concurrency

Things that are the same between child processes:

- Code and data are mostly the same (duplicate of parent's address space)
- Privileges are the same
- Open files are mostly the same

Things that are different between child processes:

- CPU registers, program counter, stack pointer
  - Each child executes the same code, but asynchronously, and execution behavior is different in each child process

Idea: lighter-weight abstraction

- Don't duplicate the things that will be the same

Threads

Threads are an abstraction of program counter, CPU registers, and call stack.

- Can be useful to think of a thread as a “virtual CPU”

Threads run *within* a process.

The process is still there to provide

- The address space
- User identity and privileges
- File and I/O resources (table of open files)

Threads and scheduling

Recall everything we discussed last time about processes and scheduling.

- Many processes exist simultaneously
- When the OS kernel makes a scheduling decision, it picks a process to run

With *threads*, the picture changes slightly:

- Many threads exist simultaneously
- When the OS kernel makes a scheduling decision, it picks a thread to run

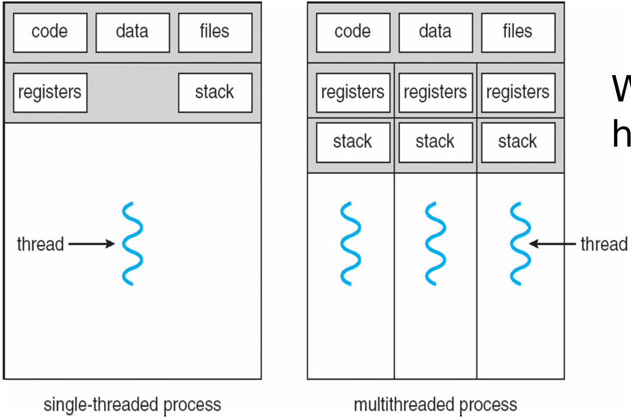
Data structure: *thread control block* (TCB)

Pintos TCB

```c
struct thread
  {
    tid_t tid;                          /* Thread identifier. */
    enum thread_status status;          /* Thread state. */
    char name[16];                      /* Name (for debugging purposes). */
    uint8_t *stack;                     /* Saved stack pointer. */
    int priority;                       /* Priority. */
    struct list_elem allelem;           /* List element for all threads list. */
    struct list_elem elem;              /* List element. */
#ifdef USERPROG
    uint32_t *pagedir;                  /* Page directory. */
#endif
    unsigned magic;                     /* Detects stack overflow. */
  };
```

Contrast: Linux `task_struct` (hundreds of fields!)

Threads in a process

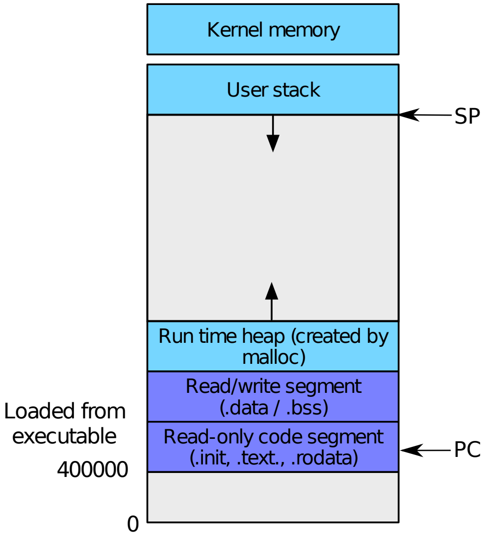


single-threaded process

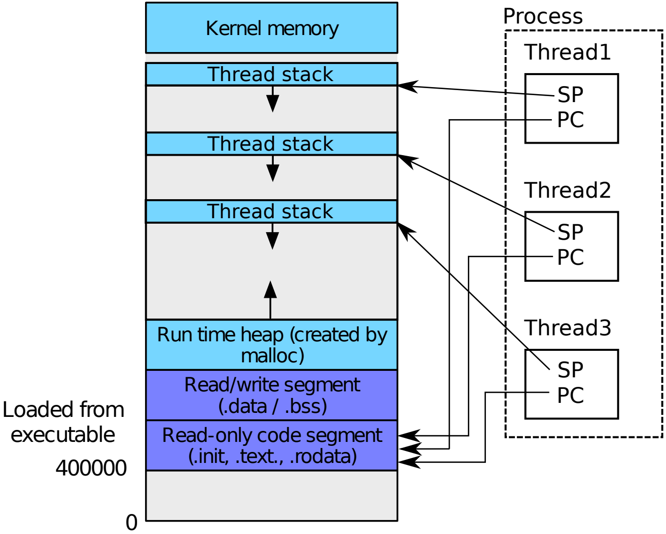
multithreaded process

What about heap memory?

Process address space


Kernel memory
User stack
SP
Run time heap (created by malloc)
Read/write segment (.data / .bss)
Loaded from executable
Read-only code segment (.init, .text., .rodata)
PC
400000
0

Process address space with threads



Utility of threads

Threads are a (fairly) lightweight mechanism for *concurrency* and *parallelism*:

- Concurrency: do many tasks at the same time (e.g., communicate with clients)
- Parallelism: take advantage of multiple CPU cores

While multiple CPU cores are needed for parallelism, threading for concurrency is useful even on a single-core CPU.

Cost of threads

All threads run in the same address space, and have full access to all memory in the address space.

Threads can cooperate by sharing data, but this will require *synchronization*. (Quite challenging to do correctly!)

Bugs in the code a thread is executing could affect other threads.

In general, bugs in multithreaded code can be challenging to identify and fix. (Race conditions, deadlocks, . . .)

Network server

```
/* function to communicate with client */
void chat_with_client(int fd);

/* main loop */
for (;;) {
  int fd = accept(ssock, NULL, NULL);
  if (fd < 0)
    /* handle error */;
  pid_t child = fork();
  if (child < 0)
    /* handle error */;
  else if (child == 0) {
    chat_with_client(fd);
    exit(0);
  }
  close(fd);
}
```

Network server with threads

```
/* function to communicate with client */
void chat_with_client(void *);

/* main loop */
for (;;) {
  int fd = accept(ssock, NULL, NULL);
  if (fd < 0)
    /* handle error */;
  ThreadData *obj = (ThreadData *) malloc(sizeof(ThreadData));
  obj->fd = fd;
  tid thr_id = thread_create(chat_with_client, obj);
  if (thr_id < 0)
    /* handle error */;
}
```

Thread API functions

```
/* Create a new thread, run fn with arg */
tid thread_create (void (*fn) (void *), void *);

/* Destroy current thread */
void thread_exit ();

/* Wait for thread thread to exit */
void thread_join (tid thread);
```

These should look familiar if you know pthreads.

Good reference on threads and thread synchronization: Birell

Creating a thread (in the kernel)

Implementation (in kernel):

1. Create TCB for new thread
2. Allocate kernel stack
3. Prepare stack (push data onto thread stack if necessary)
4. Initialize thread context (initial values of CPU registers)
    - Stack pointer points to appropriate location in thread's stack
    - PC points to first instruction the thread should execute
        - If thread will execute in user mode, will need some "trampoline" code to arrange for the transfer of control to a user space code address
5. Add TCB to ready queue

Note that what values are initially on the stack and what values are initially in CPU registers depends on the calling convention.

- 32-bit x86: arguments are passed on the stack

Creating a thread (within a user process)

```
tid thread_create (void (*fn) (void *), void *);
```

Makes a system call to request that the kernel create a kernel thread.

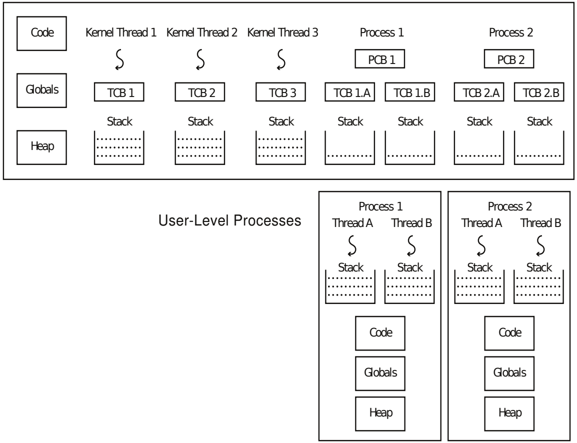
Memory will need to be allocated for the thread's stack in the process address space.

- The `thread_create` function could use `mmap` or a similar system call to do this

Kernel vs. user threads

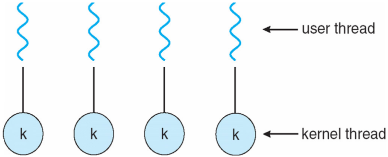
Note that:

- Some kernel threads are "kernel-only": they never execute user-mode code
- Every kernel thread has a *kernel stack* within the kernel address space
- Every user-mode thread has a corresponding kernel-mode thread
- Every user-mode thread has a user-mode stack within its process address space



Implementation strategies: 1:1 threading

In *1:1* threading, each user thread is supported by a single kernel thread:



Scheduling of user threads is therefore handled by the kernel scheduler.

This style of thread is known as a “lightweight process”.

- Windows: thread
- Solaris (a.k.a. “Illumos”): lightweight process (LWP)
- POSIX Threads: `PTHREAD_SCOPE_SYSTEM`

Tradeoffs with 1:1 threading

Pros:

- Simple approach
- User threads are “seen” by the kernel, so they can be scheduled on any CPU core
  - So, they can be used for parallelism

Cons:

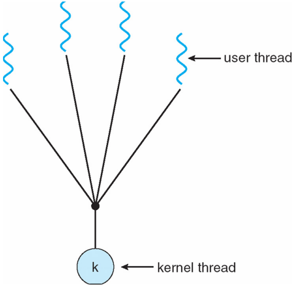
- Thread context switch requires transition into and out of the kernel
- Synchronization operations require kernel support
- Kernel resources (TCB, kernel stack) needed for every thread

Implementation strategies: user-level threading

OS kernel has no concept of threads. They are implemented entirely in user space.

Thread functions (`thread_create()`, etc.) are just library functions. Scheduling can be cooperative or preemptive. (Preemption requires a mechanism for asynchronous interruption, e.g., `SIGALARM` signal.)

- POSIX Threads: `PTHREAD_SCOPE_PROCESS`



Tradeoffs with user-level threading

Pros:

- Very lightweight (no kernel resources, kernel is not required for scheduling or synchronization)
- Context switch cost approaches cost of function call (tens of instructions rather than hundreds)

Cons:

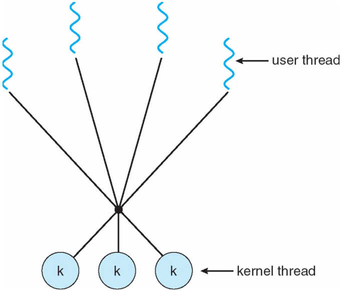
- Kernel can't "see" threads, no way to take advantage of multiple CPUs
- What happens if a thread makes a blocking system call?
  - How does the thread library detect this and schedule a different thread?
  - How would the thread library know when the blocking operation is finished?

Could the thread library cooperate with the kernel?

- Anderson et. al., Scheduler Activations

Implementation strategies: *m:n* threading

Multiplex user threads (m) onto some number of kernel threads (n):



Tradeoffs with *m:n* threading

Pros:

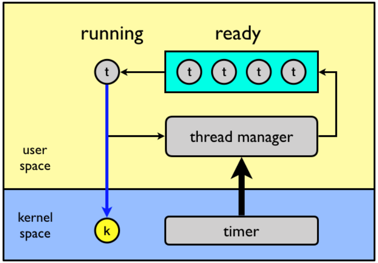
- Only need fixed number of kernel threads (one per CPU core)

Cons:

- Blocking system calls still must be dealt with
- User threads could run on any kernel thread
  - Scheduler data structure is accessed by multiple LWPs, must be synchronized appropriately

Implementing user-level threads

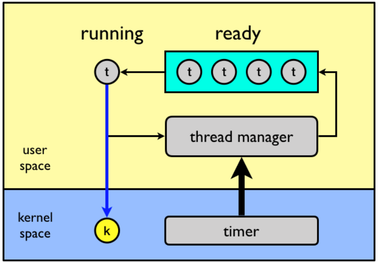
- Allocate a new stack for each `thread_create`
- Keep a queue of runnable threads
- Schedule periodic timer signal (setitimer)
  - Switch to another thread on timer signals (preemption)
- Replace blocking system calls (read/write/etc.) to non-blocking calls
  - If operation would block, switch and run different thread



Implementing user-level threads

The thread scheduler determines when a thread runs It uses queues to keep track of what threads are doing

- Just like the OS and processes
- But it is implemented at user-level in a library



Run queue: Threads currently running (usually one)

Ready queue: Threads ready to run

Are there wait queues?

- How might you implement `sleep(time)`?

Non-preemptive thread scheduling

A.k.a. “cooperative” scheduling.

Threads voluntarily give up the CPU with `yield`:

```c
/* ping thread */
while (1) {
  printf("ping\n");
  yield();
}
```

```c
/* pong thread */
while (1) {
  printf("pong\n");
  yield();
}
```

What is the output of running these two threads?

yield()

Wait a second. How does `yield()` work?

The semantics of yield are that it gives up the CPU to another thread

- In other words, it context switches to another thread

So what does it mean for yield to return?

- It means that another thread called yield!

Execution trace of ping/pong

```
printf("ping\n");   /* ping thread */
yield();            /* scheduling decision, switch to pong thread */
printf("pong\n");   /* pong thread */
yield();            /* scheduling decision, switch to ping thread */
...
```

Coroutines

Coroutines: very similar idea to cooperative threads. Main difference is that the `yield()` function takes an argument specifying which coroutine to transfer control to.

The idea is that the application can determine which thread to schedule next.

Preemptive thread scheduling

Non-preemptive threads have to voluntarily give up CPU

- A long-running thread will take over the machine
- Only voluntary calls to yield, sleep, or finish cause a context switch

Preemptive scheduling causes an involuntary context switch

- Need to regain control of processor asynchronously
- Use timer interrupt
- Timer interrupt handler forces current thread to “call” yield

Thread context switch

The context switch routine does all of the magic

- Saves context of the currently running thread (`old_thread`)
  - Push all machine state onto its stack
- Restores context of the next thread
  - Pop all machine state from the next thread's stack
- The next thread becomes the current thread
- Return to caller as new thread

This is all done in assembly language

- It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls

Background: calling conventions

What

- a standard on how functions should be implemented and called by the machine
- how a function call in C or C++ gets converted into assembly language
  - how arguments are passed to a func, how return values are passed back out of a function, how the func is called, and how the func manages the stack and its stack frame, etc.
- Compilers need to obey this standard in compiling code into assembly
  - set up the stack and registers properly

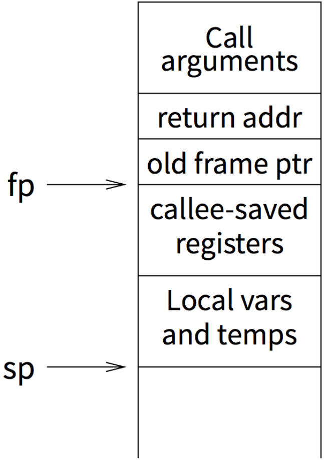
Why

- A program calls functions across many object files and libraries
- For these codes to be interfaced together, we need a standardization for calls

Background: calling conventions

x86 calling convention stack setup:

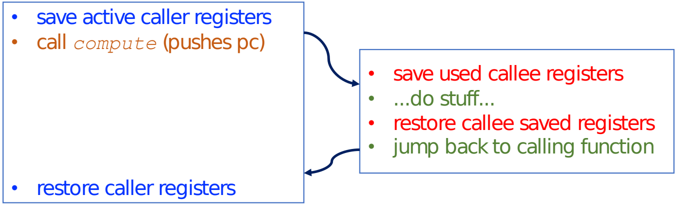
```
int compute(int a, int b) {
  int i, result;
  result = 0;
  for (i = 0; i < a; i++)
    result = result + b - i;
  return result;
}

void foo() {
  int x, y, z;
  x = 3;
  y = 5;
  z = compute(x, y);
  printf("compute(%d, %d)=%d\n", x, y, z);
}
```



Background: calling conventions

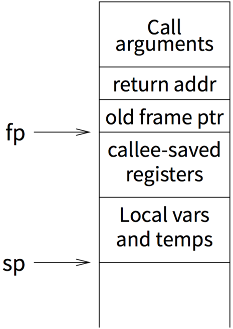
Registers divided into 2 groups:

- caller-saved regs: callee function free to modify
  - on 32-bit x86, `%eax` (return val), `%edx`, and `%ecx`
- callee-saved regs: callee function must restore to original value upon return
  - on 32-bit x86, `%ebx`, `%esi`, `%edi`, plus `%ebp` and `%esp`



- save active caller registers
- call *compute* (pushes pc)
- restore caller registers

- save used callee registers
- ...do stuff...
- restore callee saved registers
- jump back to calling function



Call arguments

return addr

fp → old frame ptr

callee-saved registers

Local vars and temps

sp →

Pintos thread implementation

```
struct thread {
  tid_t tid;
  enum thread_status status;
  char name[16];
  uint8_t *stack; /* Saved stack pointer. */
  struct list_elem allelem;
  struct list_elem elem;
  unsigned magic; /* Detects stack overflow. */
};

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

```
/* Each thread structure is stored in its own 4 kB page.  The
   thread structure itself sits at the very bottom of the page
   (at offset 0).  The rest of the page is reserved for the
   thread's kernel stack, which grows downward from the top of
   the page (at offset 4 kB) */
                  4 kB +---------------------------------+
                       |          kernel stack           |
                       |                |                |
                       |                |                |
                       |                V                |
                       |         grows downward          |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
                       |                                 |
sizeof (struct thread) +---------------------------------+
                       |              magic              |
                       |                :                |
                       |                :                |
                       |              status             |
                       |               tid               |
                  0 kB +---------------------------------+
```

Pintos switch_threads

C declaration for thread-switch function:

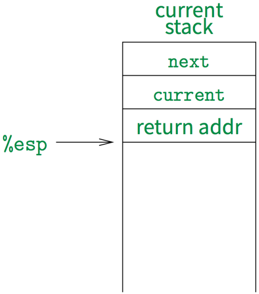
```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

Actual implementation is in i386 assembly.

i386 switch_threads

```
struct thread *switch_threads (struct thread *cur, struct thread *next);
```

```
pushl %ebx; pushl %ebp     # Save callee-saved regs
pushl %esi; pushl %edi
mov thread_stack_ofs, %edx # %edx = offset of stack field
                           # in thread struct
movl 20(%esp), %eax        # %eax = cur
movl %esp, (%eax,%edx,1)   # cur->stack = %esp
movl 24(%esp), %ecx        # %ecx = next
movl (%ecx,%edx,1), %esp   # %esp = next->stack
popl %edi; popl %esi       # Restore calle-saved regs
popl %ebp; popl %ebx
ret                        # Resume execution
```

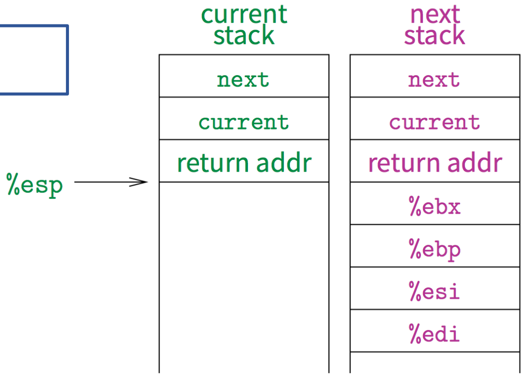


This is actual code from Pintos switch.S (slightly reformatted)

- See Thread Switching in documentation

i386 switch_threads

```
pushl %ebx; pushl %ebp
pushl %esi; pushl %edi
```

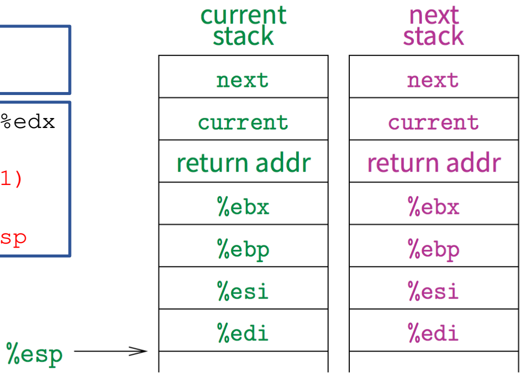


i386 switch_threads

```
pushl %ebx; pushl %ebp
pushl %esi; pushl %edi
```

```
mov thread_stack_ofs, %edx
movl 20(%esp), %eax
movl %esp, (%eax,%edx,1)
movl 24(%esp), %ecx
movl (%ecx,%edx,1), %esp
```

```
# cur->stack = %esp
# %esp = next->stack
```

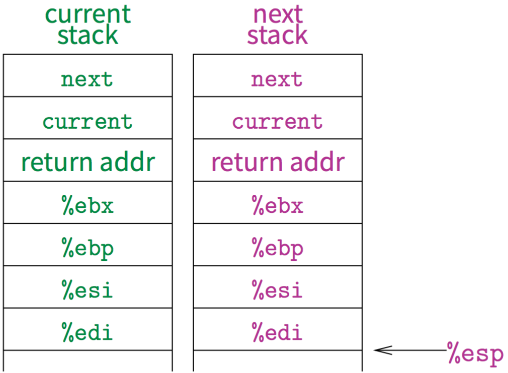


i386 switch_threads

```
pushl %ebx; pushl %ebp
pushl %esi; pushl %edi
```

```
mov thread_stack_ofs, %edx
movl 20(%esp), %eax
movl %esp, (%eax,%edx,1)
movl 24(%esp), %ecx
movl (%ecx,%edx,1), %esp
```

```
popl %edi; popl %esi
popl %ebp; popl %ebx
```



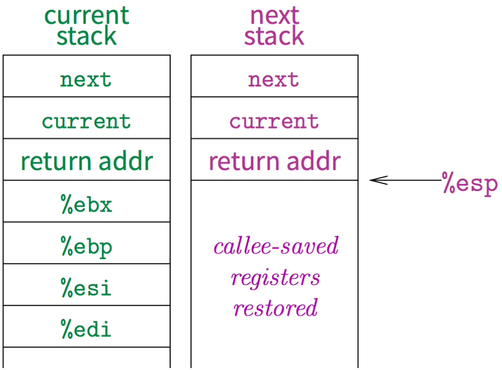
i386 switch_threads

```
pushl %ebx; pushl %ebp
pushl %esi; pushl %edi
```

```
mov thread_stack_ofs, %edx
movl 20(%esp), %eax
movl %esp, (%eax,%edx,1)
movl 24(%esp), %ecx
movl (%ecx,%edx,1), %esp
```

```
popl %edi; popl %esi
popl %ebp; popl %ebx
```

```
ret
```



current stack

next

current

return addr

%ebx

%ebp

%esi

%edi

next stack

next

current

return addr

callee-saved registers restored

%esp

Conclusions

The operating system as a large multithreaded program

- Each process executes as (at least one) thread within the OS

Multithreading is also very useful for applications

- Efficient multithreading requires fast primitives
- Processes are too heavyweight (for some applications)

Solution is to separate threads from processes

- Kernel-level threads much better, but still significant overhead
- User-level threads even better, but not well integrated with OS

Now, how do we get our threads to correctly cooperate with each other?

- Synchronization. . .