# Threads and Concurrency

**key concepts:** threads, concurrent execution, timesharing, context switch, interrupts, preemption

Lesley Istead, Kevin Lanctot

David R. Cheriton School of Computer Science
University of Waterloo

Winter 2019



## What is a thread?

… a sequence of instructions.

- A normal **sequential program** consists of a single thread of execution.
- Threads provide a way for programmers to express **concurrency** in a program.
- In threaded concurrent programs there are multiple threads of execution, all occuring at the same time.

**Recall: Concurrency**

… multiple programs or sequences of instructions running, or appearing to run, at the same time.

## Why Threads?

1. parallelism exposed by threads enables parallel execution if the underlying hardware supports it; **programs can run faster**
2. parallelism exposed by threads enables better processor utilization; **when one thread blocks, another may be able to run**

**Blocking**

Threads may **block**, ceasing execution for a period of time, or, until some condition has been met. When a thread blocks, it is not executing instructions—the CPU is idle. Concurrency lets the CPU execute a different thread during this time. **CPU time is money!**



## OS/161 Threaded Concurrency Examples

Key ideas from the examples:

- A thread can create new threads using `thread_fork`
- New theads start execution in a function specified as a parameter to `thread_fork`
- The original thread (which called `thread_fork`) and the new thread (which is created by the call to `thread_fork`) proceed concurrently, as two simultaneous sequential threads of execution.
- All threads *share* access to the program's global variables and heap.
- Each thread's function activations are *private* to that thread.

**In the OS**

... a thread is represented as a structure or object.

## OS/161's Thread Interface

- create a new thread:

```
int thread_fork(
 const char *name,          // name of new thread
 struct proc *proc,         // thread's process
 void (*func)               // new thread's function
  (void *, unsigned long),
 void *data1,               // function's first param
 unsigned long data2        // function's second param
);
```

- terminate the calling thread:

```
void thread_exit(void);
```

- volutarily yield execution:

```
void thread_yield(void);
```

See `kern/include/thread.h`



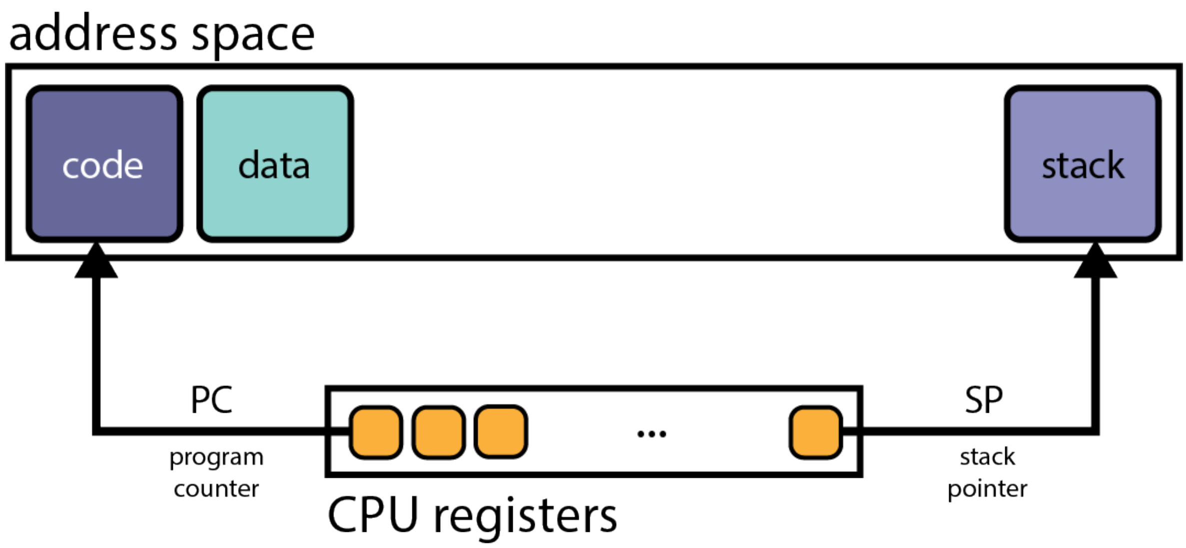
## Other Thread Libraries and Functions

- **join** a common thread function to force one thread to block until another finishes; **NOT** offered by OS/161
- **pthreads** POSIX threads, a well-supported, popular, and sophisticated thread API
- **OpenMP** a cross-platform, simple multi-processing and thread API
- **GPGPU Programming** general-purpose GPU programming APIs, e.g. nVidia's CUDA, create/run threads on GPU instead of CPU

**Concurrency and Threads**

- originated in 1950s to improve CPU utilization during I/O operations
- "modern" timesharing originated in the 1960s

## Review: Sequential Program Execution




**The Fetch/Execute Cycle**

1. fetch instruction PC points to
2. decode and execute instruction
3. advance PC

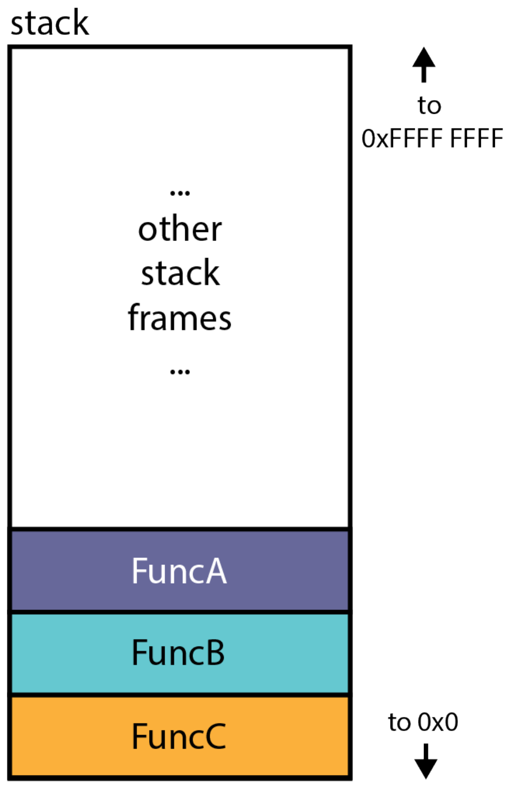


## Review: MIPS Registers

| num | name | use | num | name | use |
|---|---|---|---|---|---|
| 0 | z0 | always zero | 24-25 | t8-t9 | temps (caller-save) |
| 1 | at | assembler reserved | 26-27 | k0-k1 | kernel temps |
| 2 | v0 | return val/syscall # | 28 | gp | global pointer |
| 3 | v1 | return value | 29 | sp | stack pointer |
| 4-7 | a0-a3 | subroutine args | 30 | s8/fp | frame ptr (callee-save) |
| 8-15 | t0-t7 | temps (caller-save) | 31 | ra | return addr (for jal) |
| 16-23 | s0-s7 | saved (callee-save) | | | |

See `kern/arch/mips/include/kern/regdefs.h`

## Review: The Stack

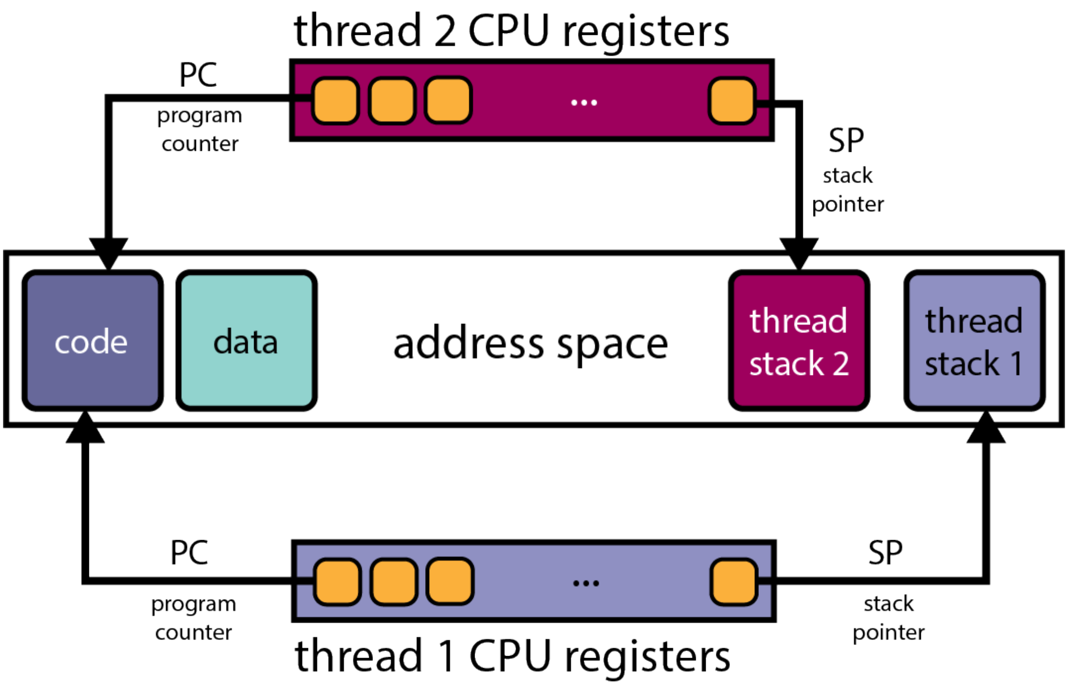



```
FuncA() {
    ...
    FuncB();
    ...
}

FuncB() {
    ...
    FuncC();
    ...
}
```



## Concurrent Program Execution (Two Threads)




Conceptually, each thread executes sequentially using its private register contents and stack.

## Implementing Concurrent Threads

What options exist?

1. Hardware support. $P$ processors, $C$ cores, $M$ multithreading per core $\Rightarrow$ $PCM$ threads can execute **simultaneously**.
2. Timesharing. Multiple threads take turns on the same hardware; rapidly switching between threads so all make progress.
3. Hardware support + Timesharing. $PCM$ threads running simultaneously with timesharing.

**Example: Intel i9-9900X**

... 10 cores, each core can run 2 threads (multithreading degree). Therefore, $P = 1$, $C = 10$, and $M = 2$, so $PCM = 20$ threads can run simultaneously.



## Timesharing and Context Switches

- When timesharing, the switch from one thread to another is called a **context switch**
- What happens during a context switch:
  1. decide which thread will run next (scheduling)
  2. save register contents of current thread
  3. load register contents of next thread
- Thread context must be saved/restored carefully, since thread execution continuously changes the context

**Timesharing**

... each thread gets a small amount of time to execute on the CPU, when it expires, a context switch occurs. Threads **share** the CPU, giving the user the illusion of multiple programs running at the same time.

## Context Switch on the MIPS (1 of 2)

```
/* See kern/arch/mips/thread/switch.S */

switchframe_switch:
  /* a0: address of switchframe pointer of old thread. */
  /* a1: address of switchframe pointer of new thread. */

   /* Allocate stack space for saving 10 registers. 10*4 = 40 */
   addi sp, sp, -40

   sw   ra, 36(sp)  /* Save the registers */
   sw   gp, 32(sp)
   sw   s8, 28(sp)
   sw   s6, 24(sp)
   sw   s5, 20(sp)
   sw   s4, 16(sp)
   sw   s3, 12(sp)
   sw   s2, 8(sp)
   sw   s1, 4(sp)
   sw   s0, 0(sp)

   /* Store the old stack pointer in the old thread */
   sw   sp, 0(a0)
```



## Context Switch on the MIPS (2 of 2)

```
   /* Get the new stack pointer from the new thread */
   lw   sp, 0(a1)
   nop           /* delay slot for load */

   /* Now, restore the registers */
   lw   s0, 0(sp)
   lw   s1, 4(sp)
   lw   s2, 8(sp)
   lw   s3, 12(sp)
   lw   s4, 16(sp)
   lw   s5, 20(sp)
   lw   s6, 24(sp)
   lw   s8, 28(sp)
   lw   gp, 32(sp)
   lw   ra, 36(sp)
   nop                  /* delay slot for load */

   /* and return. */
   j ra
   addi sp, sp, 40      /* in delay slot */
   .end switchframe_switch
```

## What Causes Context Switches?

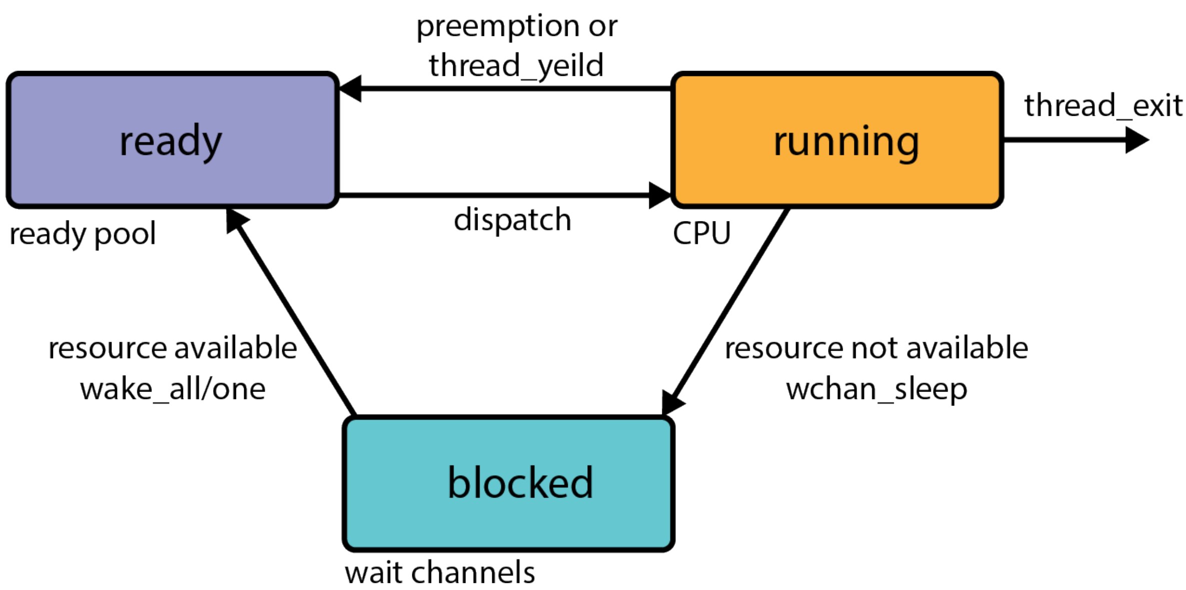
- the running thread calls **thread_yield**
  - running thread *voluntarily* allows other threads to run
- the running thread calls **thread_exit**
  - running thread is terminated
- the running thread *blocks*, via a call to **wchan_sleep**
  - more on this later . . .
- the running thread is *preempted*
  - running thread *involuntarily* stops running

**The OS**

... strives to maintain high CPU utilization. Hence, in addition to timesharing, context switches occur whenever a thread ceases to execute instructions.



## Thread States

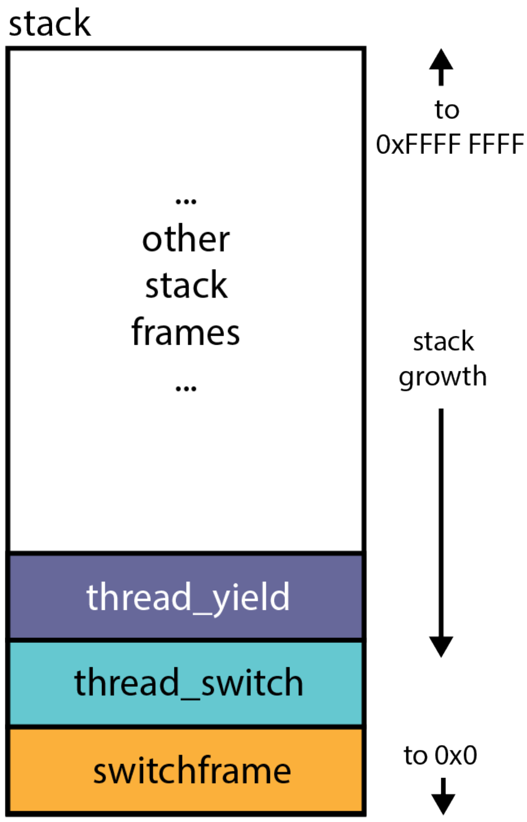



**running:** currently executing

**ready:** ready to execute

**blocked:** waiting for something, so not ready to execute.

## OS/161 Thread Stack after Voluntary Context Switch




- program calls `thread_yield`, to yield the CPU
- `thread_yield` calls `thread_switch`, to perform a context switch
- `thread_switch` chooses a new thread, calls `switchframe_switch` to perform low-level context switch



## Timesharing and Preemption

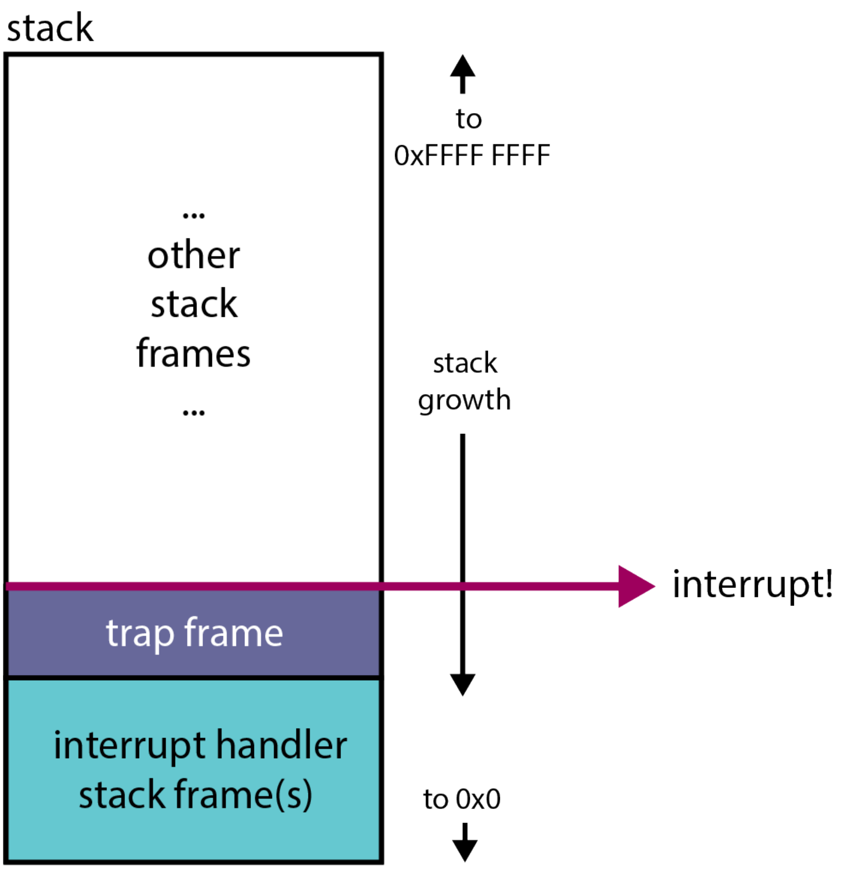
- **timesharing**—concurrency achieved by rapidly switching between threads
  - how rapidly? impose a limit on CPU time, the **scheduling quantum**
  - the quantum is an **upper bound** on how long a thread can run before it must yield the CPU
- how do you stop a running thread, that never yields, blocks or exits when the quantum expires?
  - **preemption** forces a running thread to stop running, so that another thread can have a chance
  - to implement preemption, the thread library must have a means of "getting control" (causing thread library code to be executed) even though the running thread has not called a thread library function
  - this is normally accomplished using **interrupts**

## Review: Interrupts

- an **interrupt** is an event that occurs during the execution of a program
- interrupts are caused by system devices (hardware), e.g., a timer, a disk controller, a network interface
- when an interrupt occurs, the hardware automatically transfers control to a fixed location in memory
- at that memory location, the thread library places a procedure called an *interrupt handler*
- the interrupt handler normally:
  1. create a *trap frame* to record thread context at the time of the interrupt
  2. determines which device caused the interrupt and performs device-specific processing
  3. restores the saved thread context from the trap frame and resumes execution of the thread



## OS/161 Thread Stack after in Interrupt



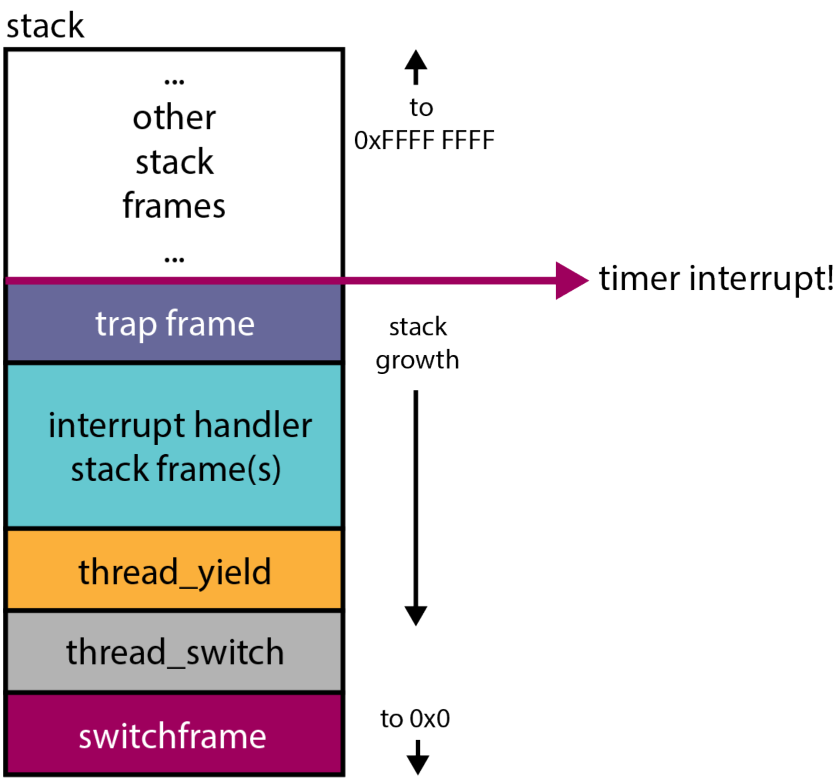
## Preemptive Scheduling

- A preemptive scheduler uses the **scheduling quantum** to impose a time limit on running threads
- Threads may block or yield before their quantum has expired.
- Periodic timer interrupts allow running time to be tracked.
- If a thread has run too long, the timer interrupt handler preempts the thread by calling `thread_yield`.
- The preempted thread changes state from running to ready, and it is placed on the *ready queue*.
- Each time a thread goes from ready to running, the runtime starts out at 0. Runtime does not accumulate.

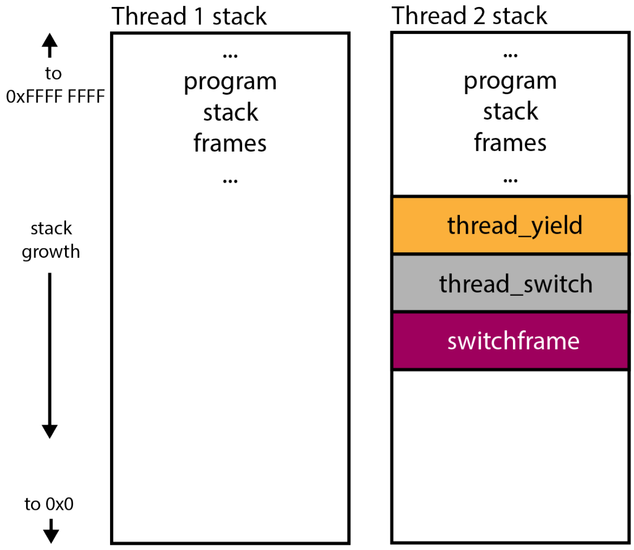
> OS/161 threads use *preemptive round-robin scheduling*.



## OS/161 Thread Stack after Preemption



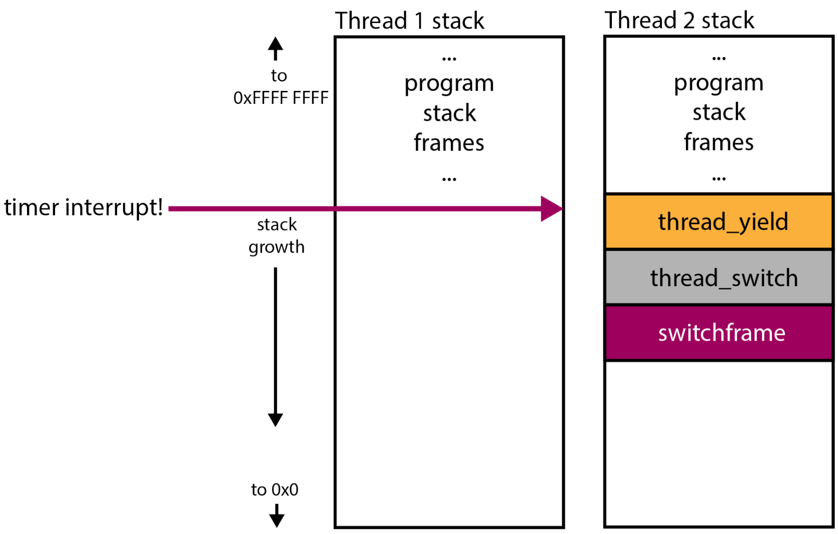
## Two-Thread Example - 1




Thread 1 is **RUNNING**. Thread 2 is **READY**, having called thread_yield previously.

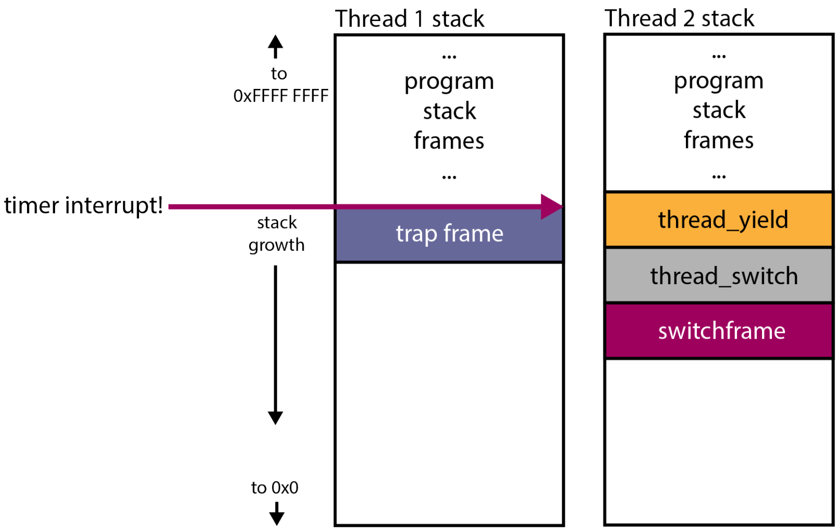


## Two-Thread Example - 2




A timer interrupt occurs.

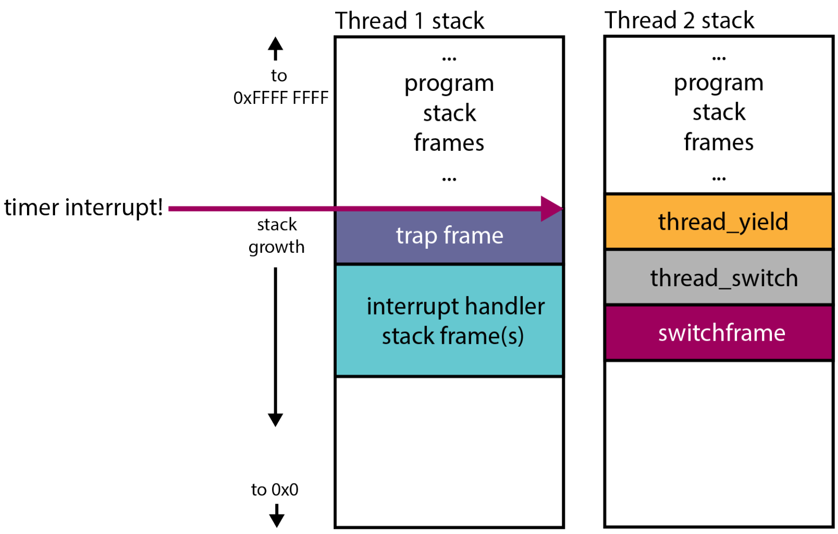
# Two-Thread Example - 3




Thread 1 is preempted, a trapframe is created to save its context.

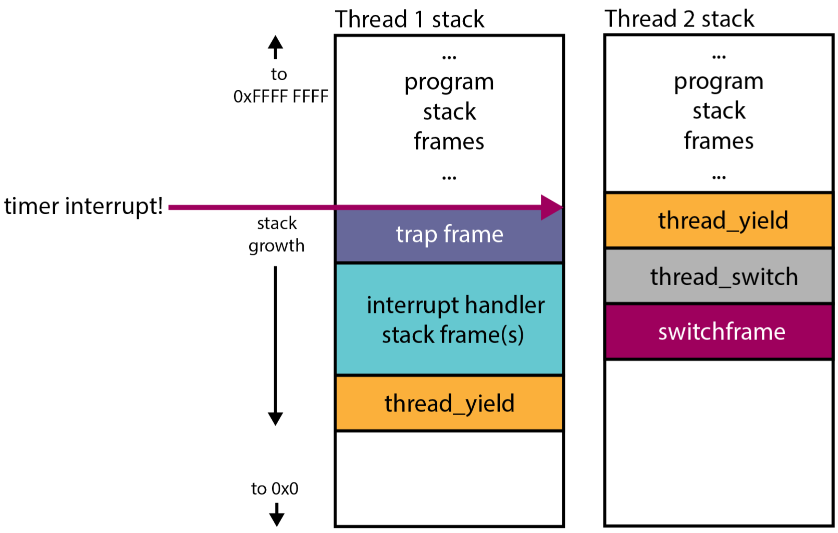


# Two-Thread Example - 4




The timer interrupt handler determines what happened, and, calls the appropriate handler.

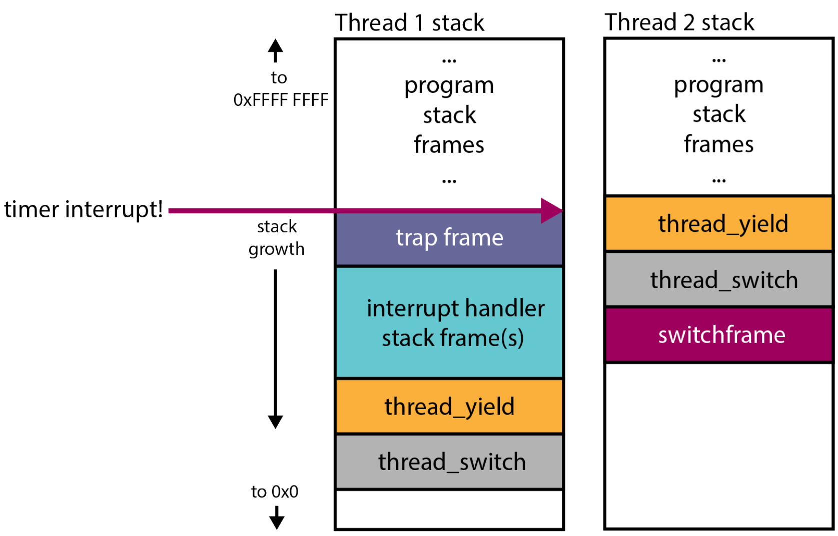
# Two-Thread Example - 5




Thread 1 has exceeded its quantum. Yield the CPU to another thread, call thread_yield.

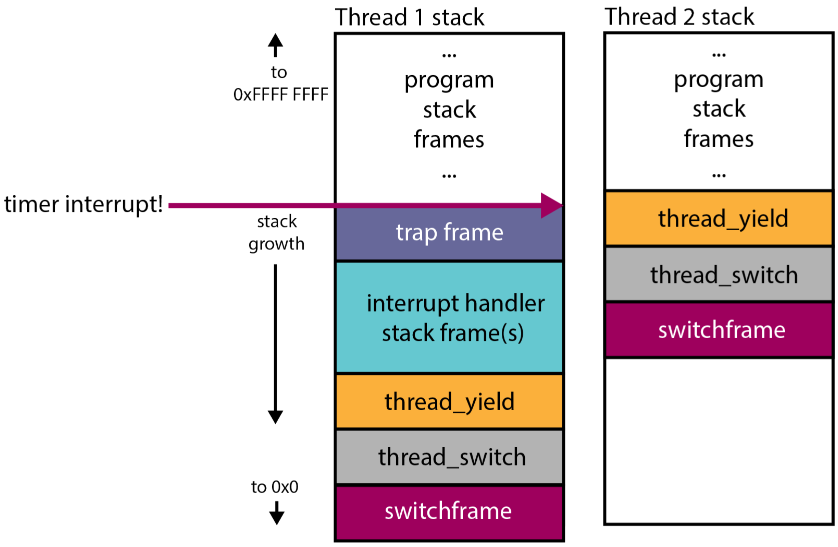


# Two-Thread Example - 6




High-level context switch: choose new thread, save caller-save registers.

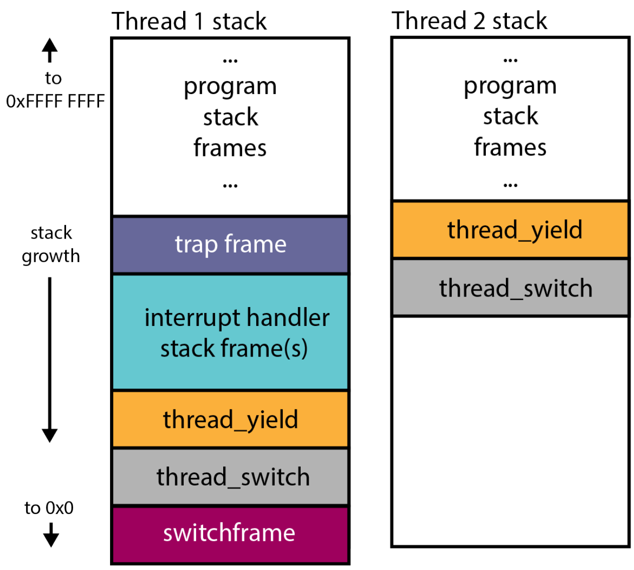
# Two-Thread Example - 7




Low-level context switch. Save callee-save registers.

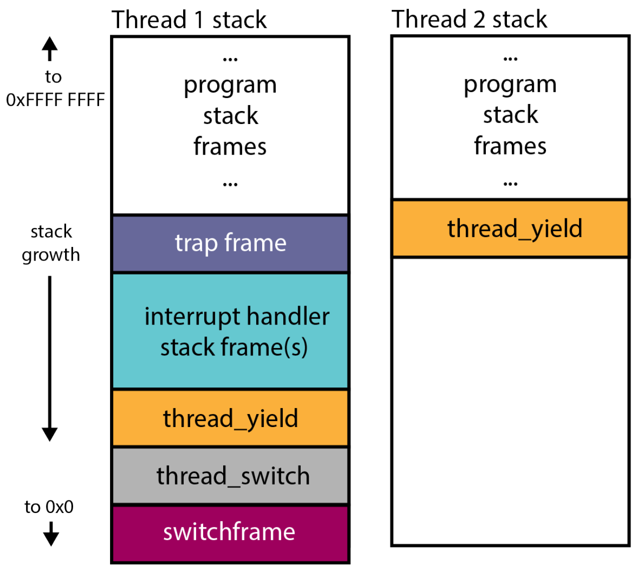


# Two-Thread Example - 8




Thread 2 is now **RUNNING**, Thread 1 is now **READY**. Thread 2 returns from low-level context switch, restoring callee-save registers.

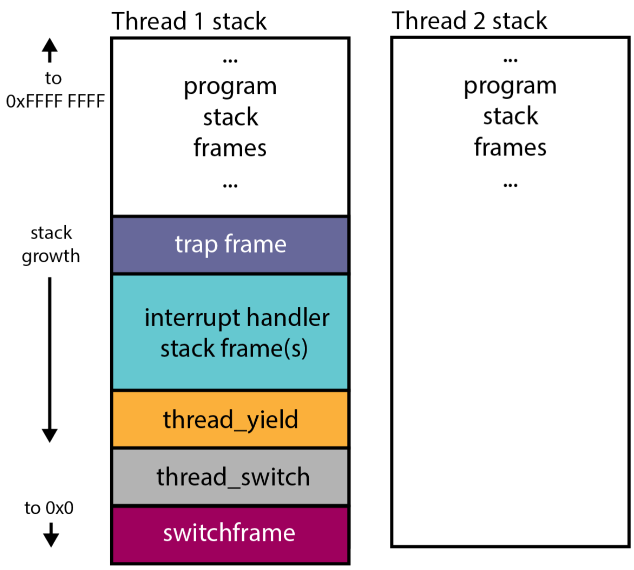
# Two-Thread Example - 9




Return from high-level context switch, restoring caller-save registers.

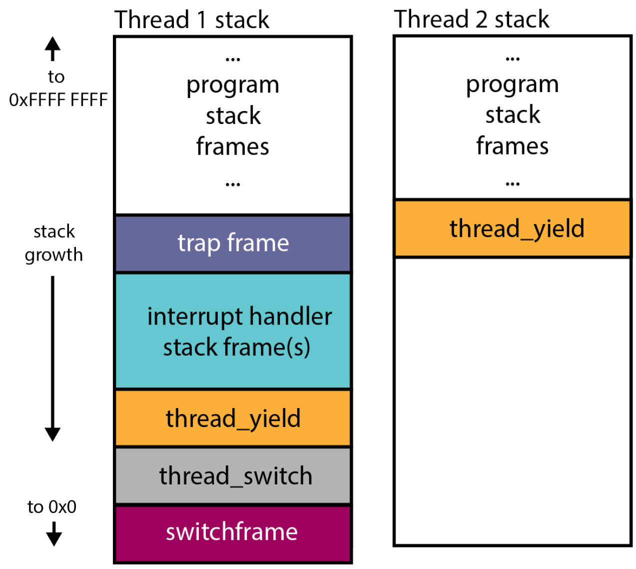


# Two-Thread Example - 10




Return from yield. Context is fully restored. Thread 2 is now running its regular program.

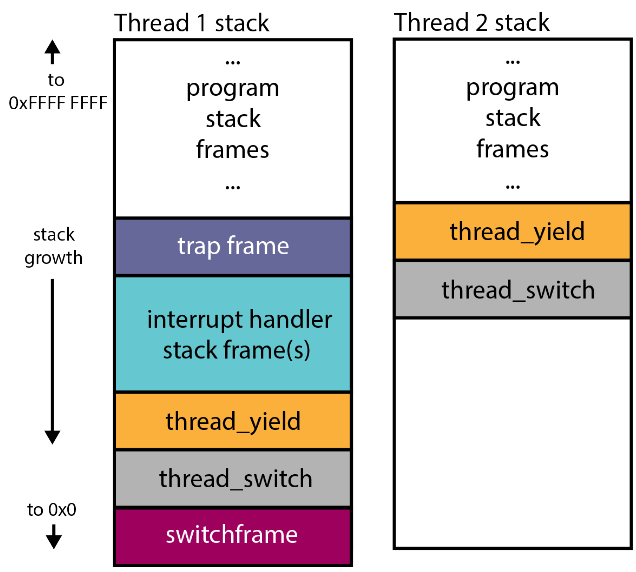
## Two-Thread Example - 11




Thread 2 yields.

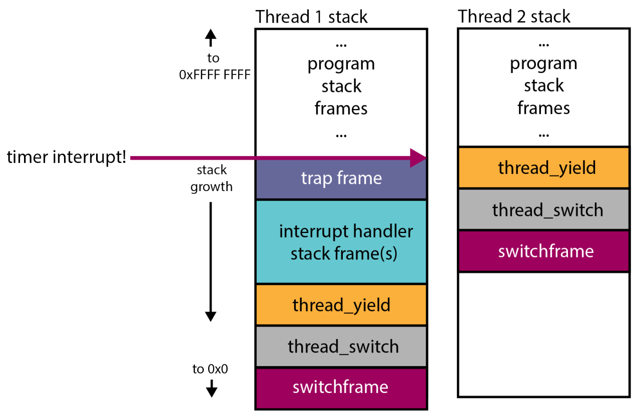


## Two-Thread Example - 12




High-level context switch.

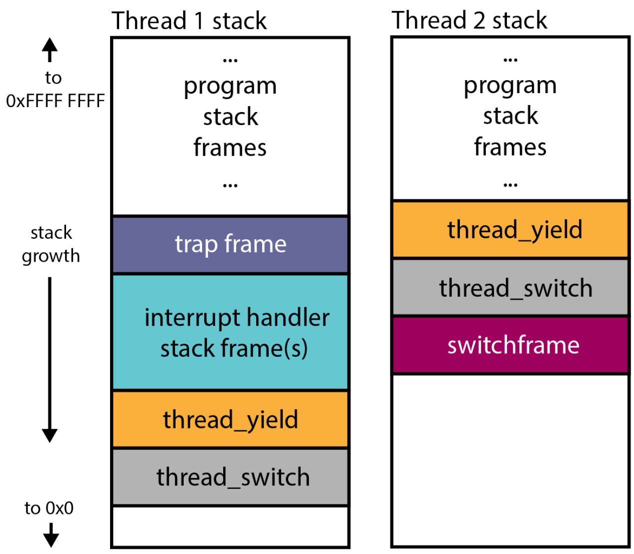
# Two-Thread Example - 13




Low-level context switch.

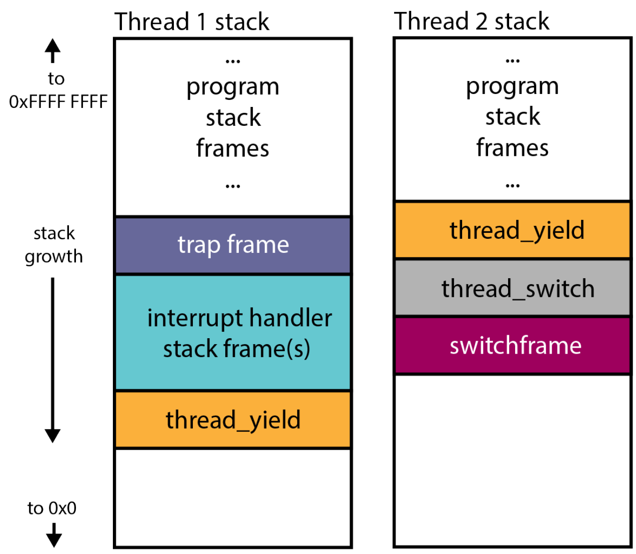


# Two-Thread Example - 14




Thread 1 is now **RUNNING**. Thread 2 is now **READY**. Return from low-level context switch.

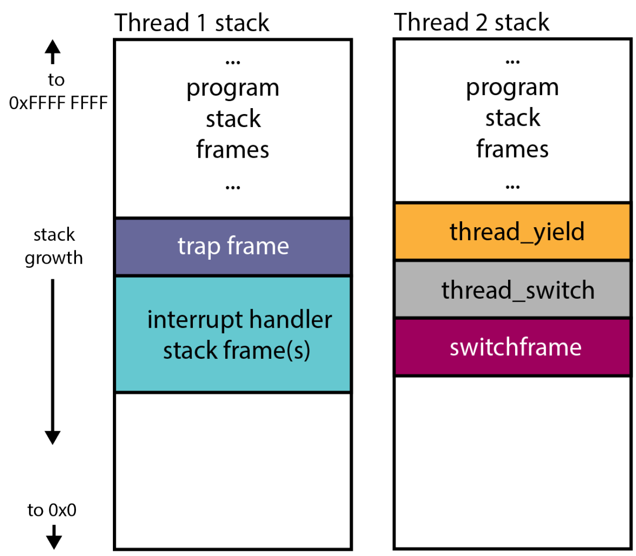
# Two-Thread Example - 15




Return from high-level context switch.

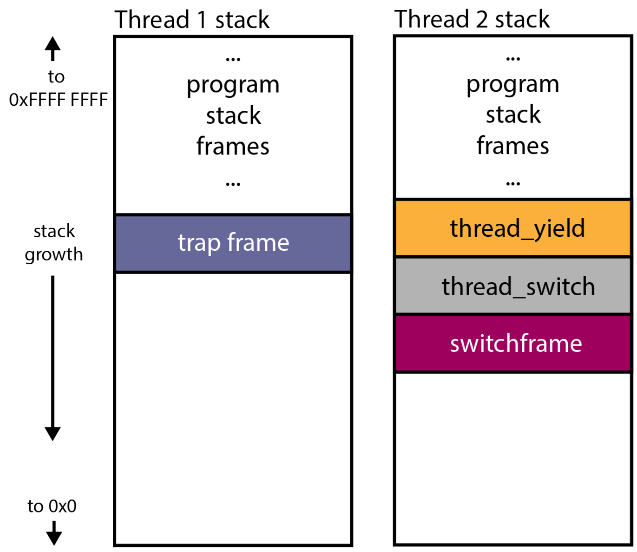


# Two-Thread Example - 16




Return from yield.

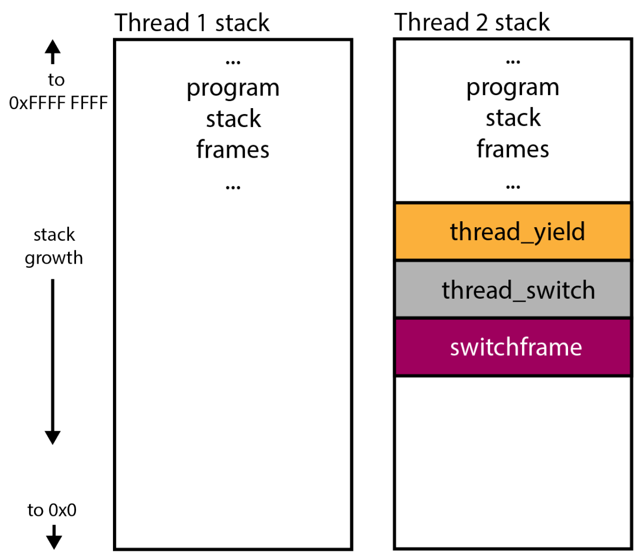
# Two-Thread Example - 17




Return from interrupt handling functions.



# Two-Thread Example - 18




Restore thread 1's context (stored in the trapframe), return to regular program.