

Threads and Concurrency

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

Zille Huma Kamal

David R. Cheriton School of Computer Science
University of Waterloo

Spring 2022

1 / 47

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 occurring at the same time.
 - Threads may perform the same task.
 - Threads may perform different tasks.

Recall: Concurrency

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

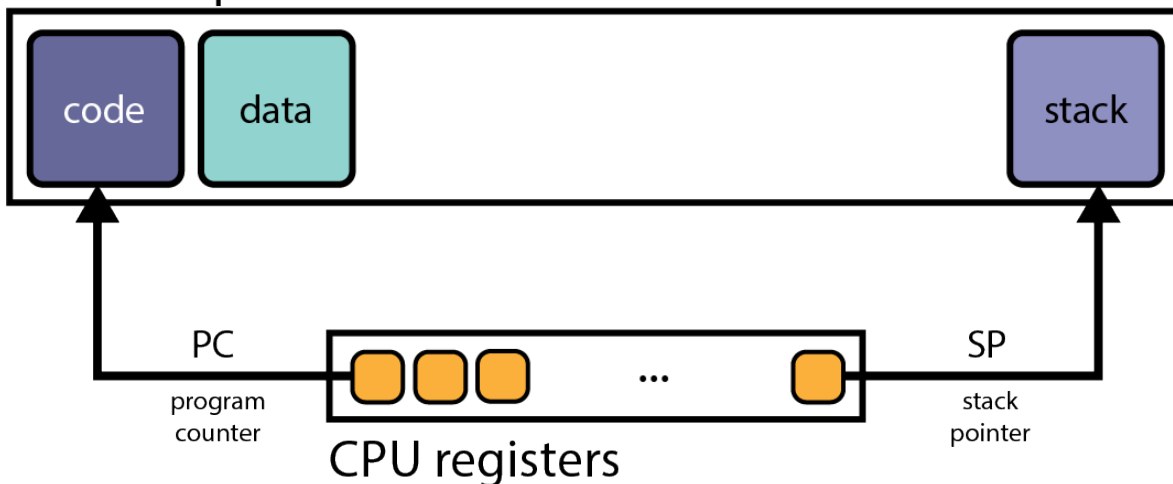
On a multicore system, concurrency \implies parallelism

On a single core system, thread execution is interleaved, **appearing** to be executing at the same time.

2 / 47

Review: Sequential Program Execution

address space

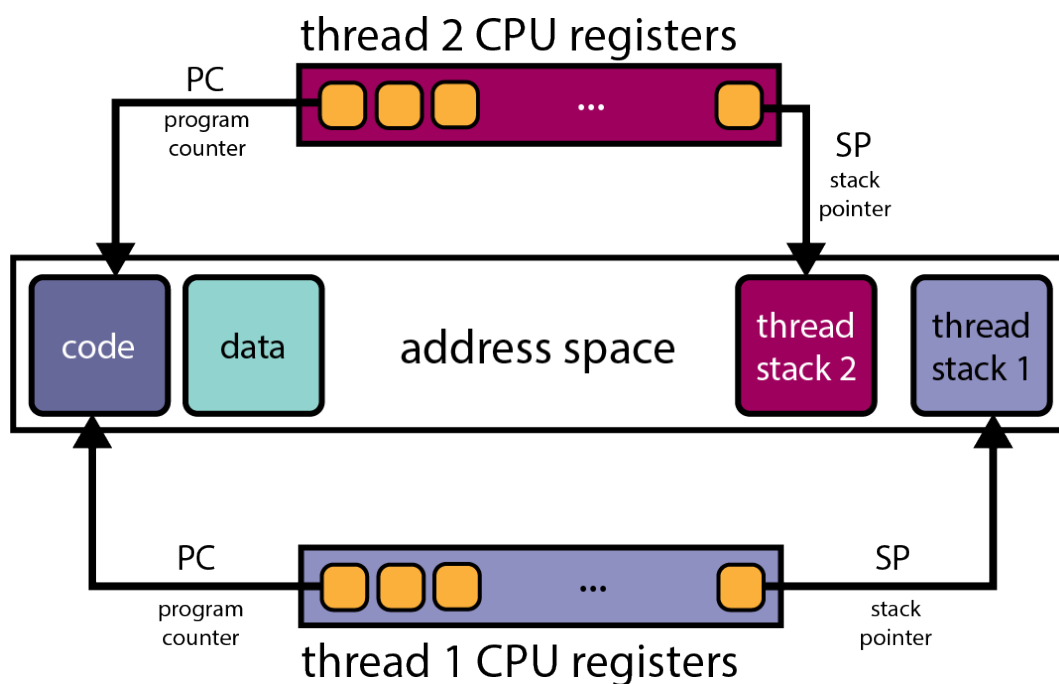


The Fetch/Execute Cycle

- 1 **fetch** instruction PC points to
- 2 decode and **execute** instruction
- 3 increment the PC

3 / 47

Concurrent Program Execution (Two Threads)



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

4 / 47

Why Threads?

- 1 **Resource Utilization:** blocked/waiting threads give up resources, i.e., the CPU, to others.
- 2 **Parallelism:** multiple threads executing simultaneously; improves performance.
- 3 **Responsiveness:** dedicate threads to UI, others to loading/long tasks.

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!**

5 / 47

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.
Note that while cores of a single processor share caches (L2, L3), threads execute separately.

6 / 47

How should we implement threads?

We can implement threads in the kernel through system calls, just like we create processes.

Limitations:

- latency for syscall is 100s of cycles w.r.t to function call, which is a few cycles
- memory requirements are similar to heavier weight processes

Alternatively, implement user-level threads in user-level library.

One kernel thread per process.

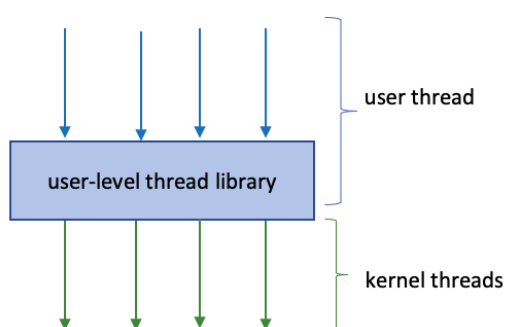
Limitations:

- cannot take advantage of multiple cores

Consider the **n:m** threading model, where we have n user threads and m kernel threads

7 / 47

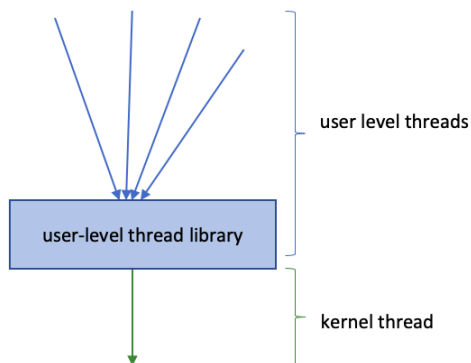
One is to One threading



- **1:1** threading, for each user thread, create a kernel thread
- higher degree of concurrency than $n:1$
- higher degree of parallelism, on a multiprocessor system
- System Contention Scope (SCS)
 - kernel threads compete for CPU
- used by Linux and Windows
- limitation: thread creation and management overhead

8 / 47

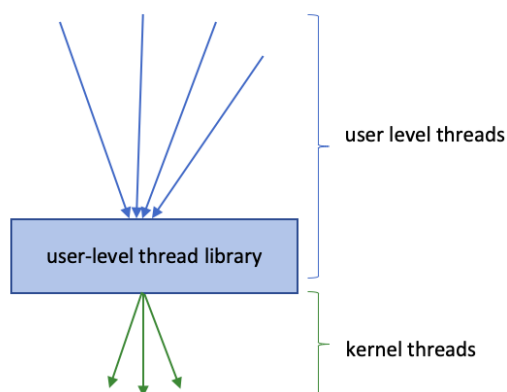
Many is to One threading



- $n > m$ and $m = 1$ many is to one
- n user level threads, only $m=1$ kernel level thread
- blocking system call by a thread, blocks entire process
- Green threads - thread API for Solaris and early Java
- Process Content Scope - threads in the process compete for scheduling
- limitation: cannot leverage parallelism of multicore systems

9 / 47

n is to m threading



- $n > m$ and $m > 1$
- for n user threads there are m kernel threads
- most flexible and complicated
- PCS and SCS
- limiting kernel threads not an issue in today's increasingly higher number of processing cores on a system
- a variation is the **2-level model** on Solaris 2.2 to 8, introduced light weight process (LWP)

10 / 47

User level Thread Library: POSIX pthread API

- `int pthread_create (pthread_t *thread_t, pthread_attr_t *attr, void *(*fn)(void *), void *arg);`
 - Create a new thread identified by `thread_t` with optional attributes `attr` within the calling process
 - `attr` includes attributes such as stack size, scheduling priority, etc.
 - `thread_t` is created to start executing function `fn` with argument `arg`
- `void pthread_exit(void *return_value);`
 - Destroy the calling thread and makes `return_value` available for any successful `pthread_join`
- `int pthread_join(pthread_t thread_t, void **return_value);`
 - Wait for thread `thread_t` to exit and retrieve the return value
- `void pthread_yield();`
 - calling thread voluntarily releases CPU
- Plus lots of support for synchronization

11 / 47

Other User level thread libraries

- **OpenMP** a cross-platform library, 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
- **Go routines in Golang**
 - light-weight, running 100k go routines is practical
 - on top of kernel threads (n:m threading model)
 - Multi-core scalability and efficient user-level threads

Concurrency and Threads

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

12 / 47

OS/161 Threaded Concurrency Examples

Key ideas from the examples:

- A thread can create new threads using `thread_fork`
- New threads 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 memory
 - code
 - data: global variables
 - heap
 - open file descriptors
- Each thread's stack frames are **private** to that thread; each thread has its own stack.

In the OS

A thread is represented as a structure or object, known as Thread Control Block (TCB).

13 / 47

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);`
- **join** a common thread function to force one thread to block until another finishes; **NOT** offered by OS/161

See `kern/include/thread.h`

14 / 47

Timesharing and Context Switches

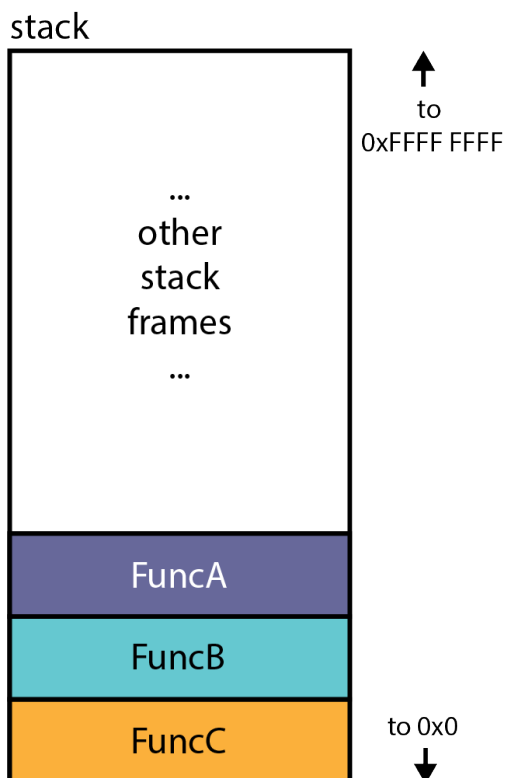
- When **timesharing**, the switch from one thread to another is called a **context switch**
- What happens during a context switch:
 - 1 decide which thread will run next (scheduling)
 - 2 save register contents of current thread
 - 3 load register contents of next thread
- **Thread context** must be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

15 / 47

Review: The Stack



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

Recall:

Functions push arguments (a0-a3), return address, local variables, and temporary-use registers onto the stack.

16 / 47

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)			

- conventions enforced in compiler; used in OS
- **caller-save**: it is the responsibility of the calling function to save/restore values in these registers
- **callee-save**: it is the responsibility of the called function to save/restore values in these registers before/after use

callee/caller save strategy attempts to minimize the callee saving values the caller does not use

17 / 47

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)
```

18 / 47

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
```

19 / 47

Switchframe Notes

- `switchframe_switch` is called by C function `thread_switch`
 - `thread_switch` is the **caller**; it will save/restore the **caller-save** registers
 - `switchframe_switch` is the **callee**; it must save/restore the **callee-save** registers
 - `switchframe_switch`, saves **callee-save** registers to the old thread stack; it restores the **callee-save** registers from the new thread's stack
- MIPS R3000 is pipelined; **delay-slots** are used to protect against:
 - **load-use hazards**, where loaded values are used in the next instruction
 - **control hazards**, where we don't know which instruction to fetch next

20 / 47

What Causes Context Switches?

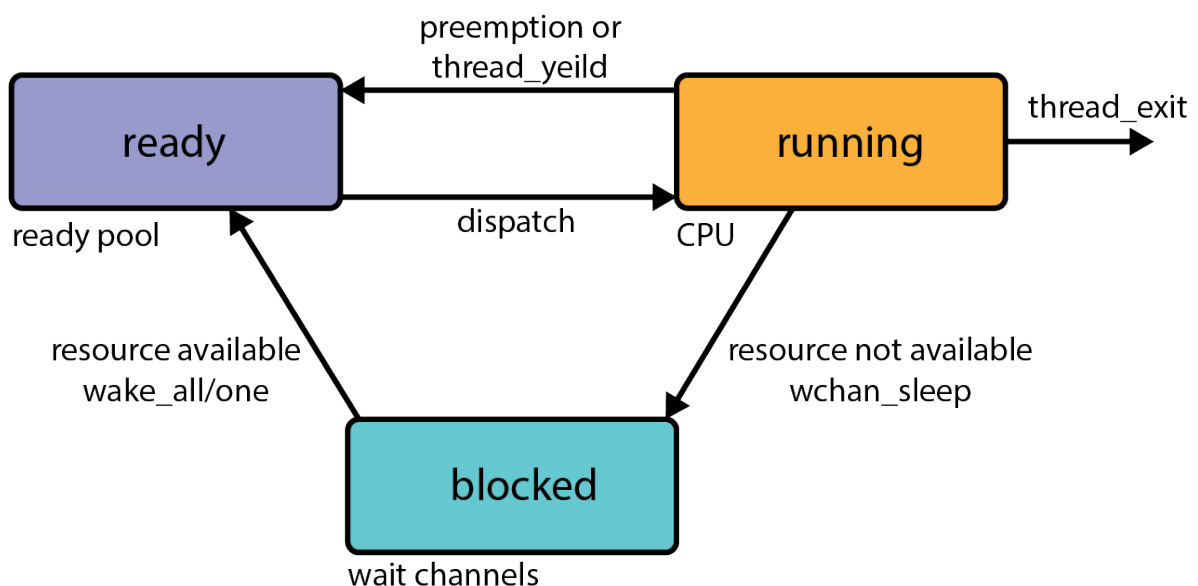
- the running thread calls **thread_yield**
 - running thread **voluntarily** allows other threads to run
- the running thread calls **thread_exit**
 - running thread **is terminated**
- the running thread **blocks**, via a call to **wchan_sleep**
 - more on this later ...
- the running thread is **preempted**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

21 / 47

Thread States

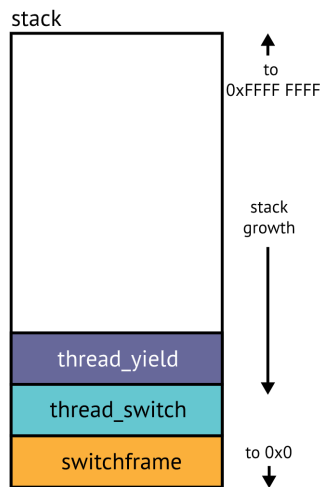


running: currently executing

ready: ready to execute

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

22 / 47



- 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

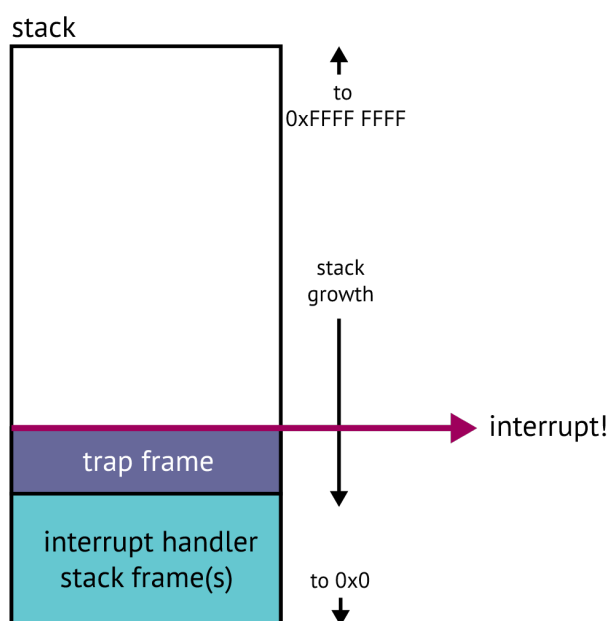
- **timesharing**—concurrency achieved by rapidly switching between threads
 - how rapidly? impose a limit on CPU time, the **scheduling quantum**
 - the quantum is an **upper bound** on how long a thread can run before it must yield the CPU
- how do you stop a running thread, that never yields, blocks 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 OS must have a means of “getting control”
 - 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 kernel memory
- at that memory location, is a procedure called an **interrupt handler**
- the interrupt handler normally:
 - 1 creates 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

25 / 47

OS/161 Thread Stack after an Interrupt



26 / 47

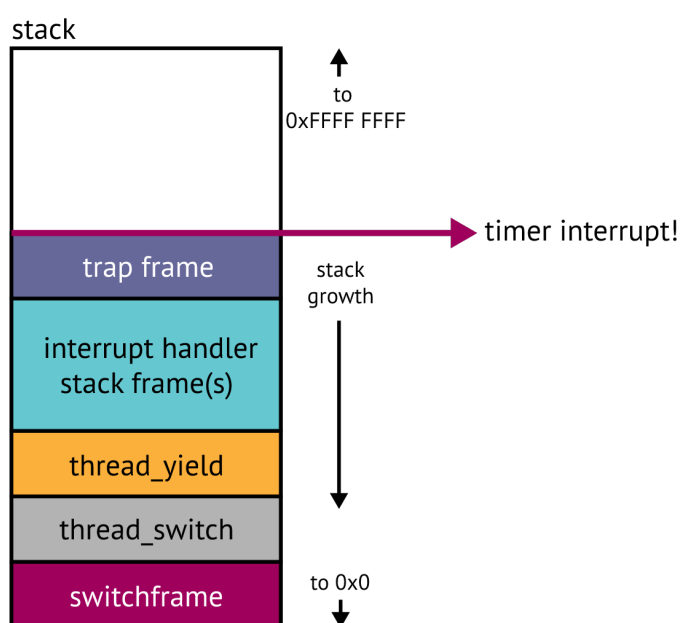
Preemptive Scheduling

- 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 in 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**.

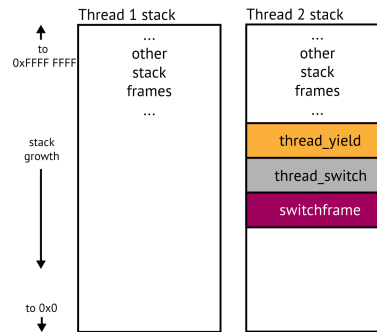
27 / 47

OS/161 Thread Stack after Preemption



28 / 47

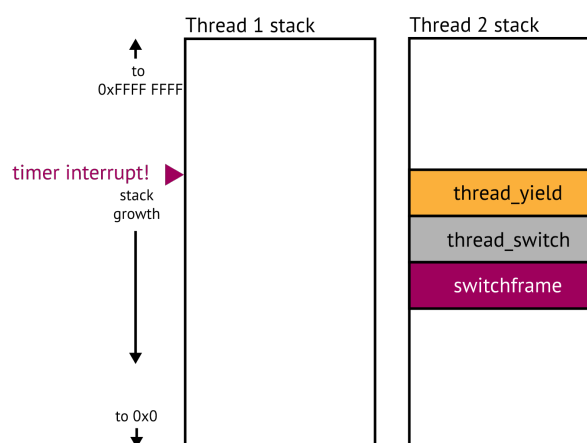
Two-Thread Example - 1



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

29 / 47

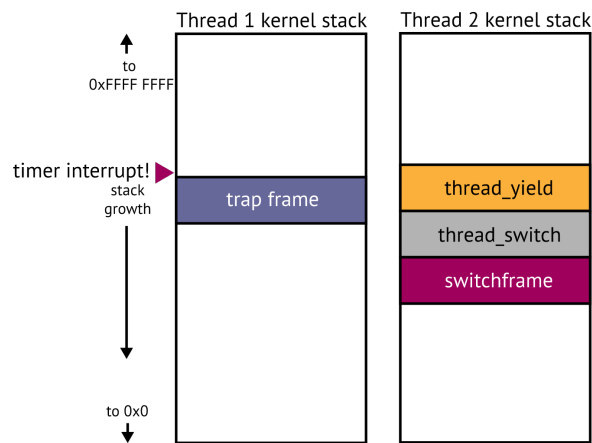
Two-Thread Example - 2



A timer interrupt occurs.

30 / 47

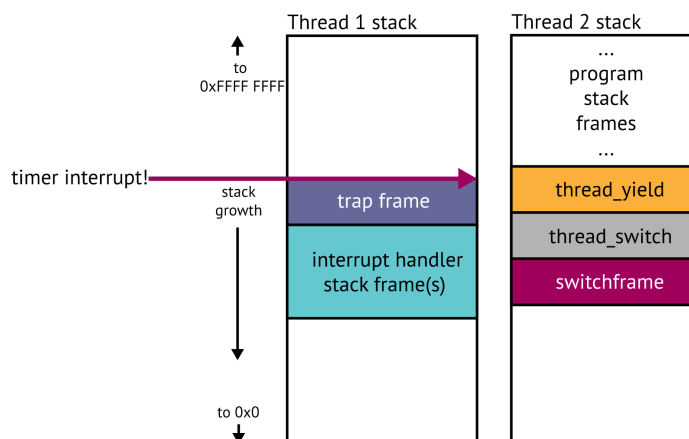
Two-Thread Example - 3



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

31 / 47

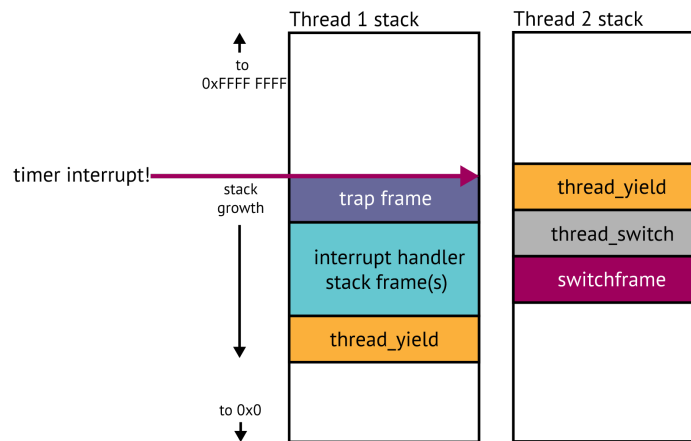
Two-Thread Example - 4



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

32 / 47

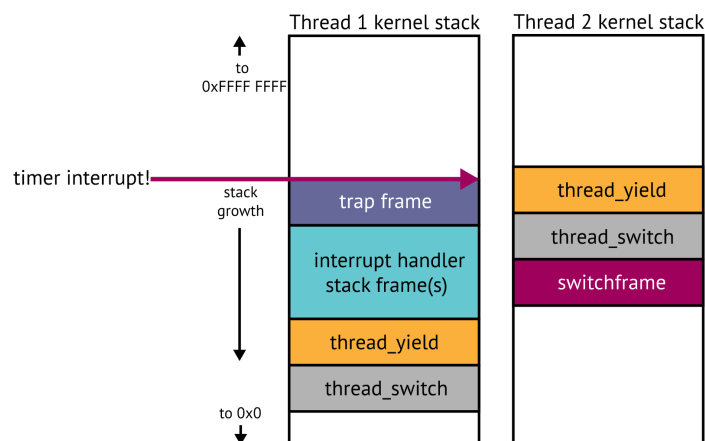
Two-Thread Example - 5



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

33 / 47

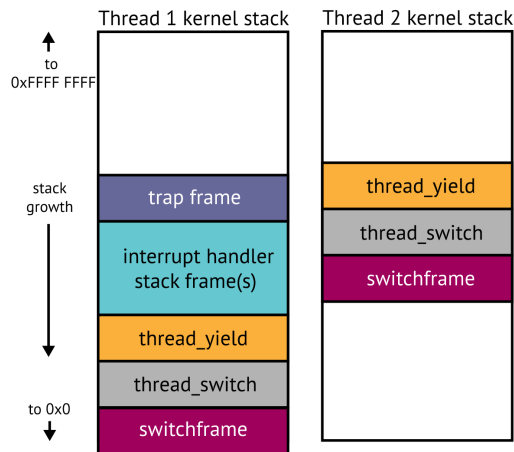
Two-Thread Example - 6



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

34 / 47

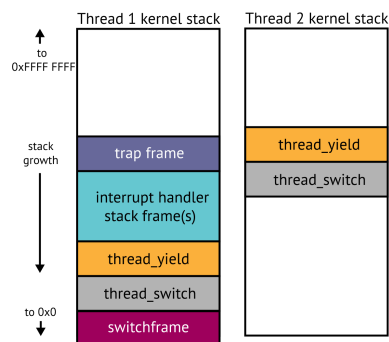
Two-Thread Example - 7



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

35 / 47

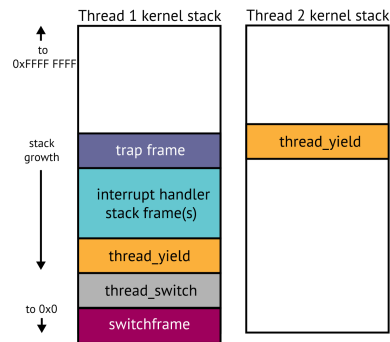
Two-Thread Example - 8



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

36 / 47

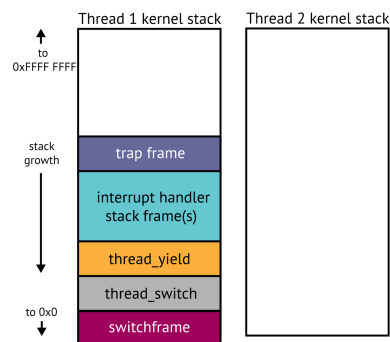
Two-Thread Example - 9



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

37 / 47

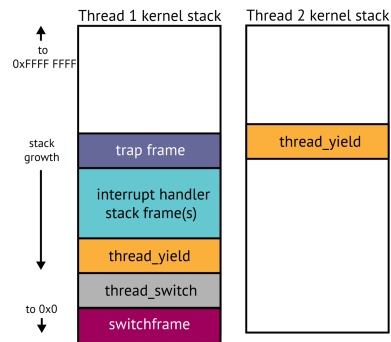
Two-Thread Example - 10



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

38 / 47

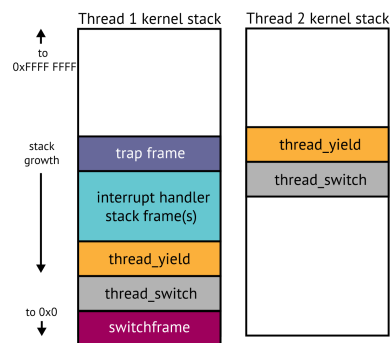
Two-Thread Example - 11



Thread 2 yields.

39 / 47

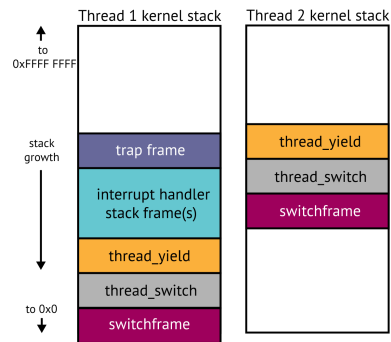
Two-Thread Example - 12



High-level context switch.

40 / 47

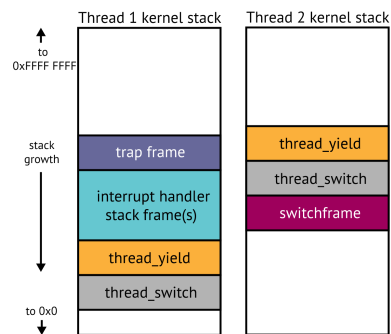
Two-Thread Example - 13



Low-level context switch.

41 / 47

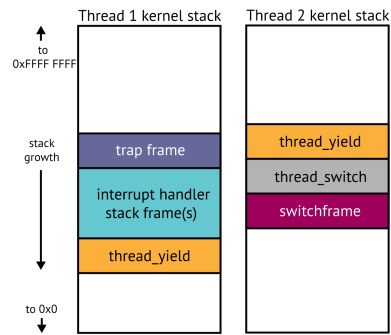
Two-Thread Example - 14



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

42 / 47

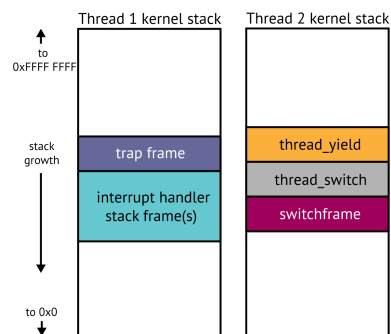
Two-Thread Example - 15



Return from high-level context switch.

43 / 47

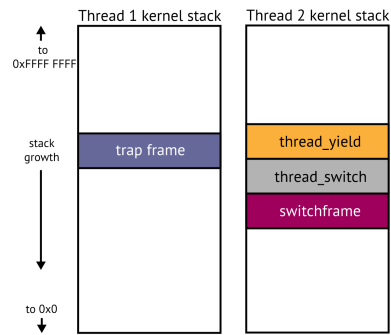
Two-Thread Example - 16



Return from yield.

44 / 47

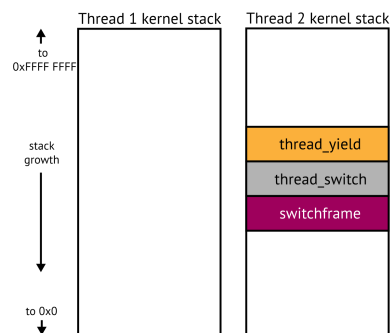
Two-Thread Example - 17



Return from interrupt handling functions.

45 / 47

Two-Thread Example - 18



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

46 / 47

How to implement process management system calls in a multi-threaded process

- if a calling thread calls **fork**, should we
 - 1 create a new process with all threads of the parent process?
 - 2 create a new process with only the calling thread from the parent process duplicated?

Linux implements option 2, since Native POSIX Threading Library(NPTL) assumes this. There are variants of `fork`, such as `clone` and `rfork` found in Linux and FreeBSD

- if a thread calls **execv**, should we replace entire process, including all threads?
- signal handling, should we
 - 1 deliver signals to all threads of a process?
 - 2 deliver signals to the thread that generated the signal?
 - 3 deliver to a specific thread that receives all signals for the process?