Threads and Concurrency

key concepts

threads, concurrent execution, timesharing, context switch, interrupts, preemption

reading

Three Easy Pieces: Chapter 26 (Concurrency and Threads)

What is a Thread?

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

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.

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:

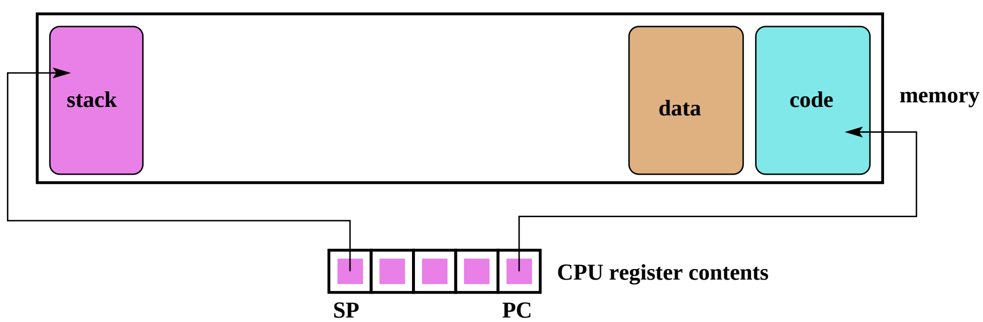
```
void thread_yield(void);
```

See `kern/include/thread.h`

Why Threads?

- **Reason #1**: parallelism exposed by threads enables parallel execution if the underlying hardware supports it.
  - programs can run faster
- **Reason #2**: parallelism exposed by threads enables better processor utilization
  - if one thread has to *block*, another may be able to run

Review: Sequential Program Execution



The Fetch/Execute Cycle

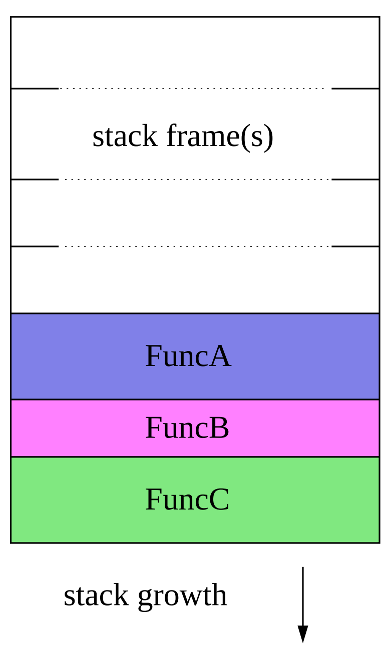
1. fetch instruction PC points to
2. decode and execute instruction
3. advance PC

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)			

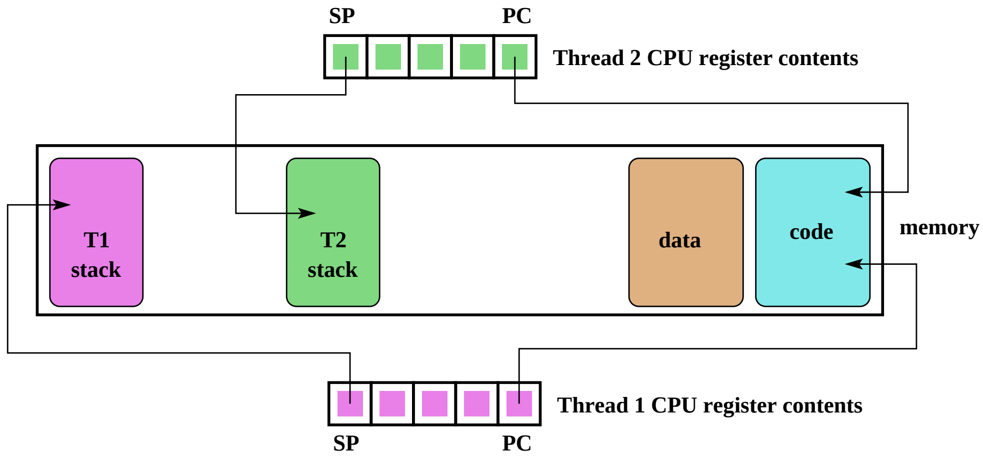
See `kern/arch/mips/include/kern/regdefs.h`

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

- Option 1: multiple processors, multiple cores, hardware multithreading per core
  - P processors, C cores per processor, M multhreading degree per core $\Rightarrow$ PCM threads can execute *simultaneously*
  - separate register set for each running thread, to hold its *execution context*
- Option 2: *timesharing*
  - multiple threads take turns on the same hardware
  - rapidly switch from thread to thread so that all make progress

In practice, both techniques can be combined.

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

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)
```

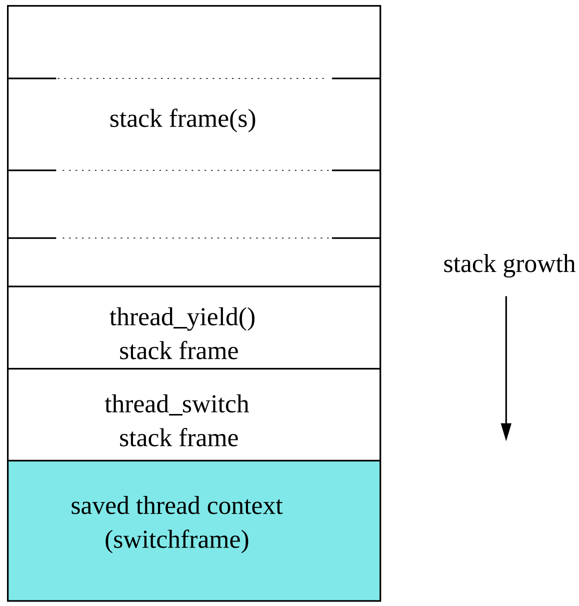
Context Switch on the MIPS (2 of 2)

```
/* 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
```

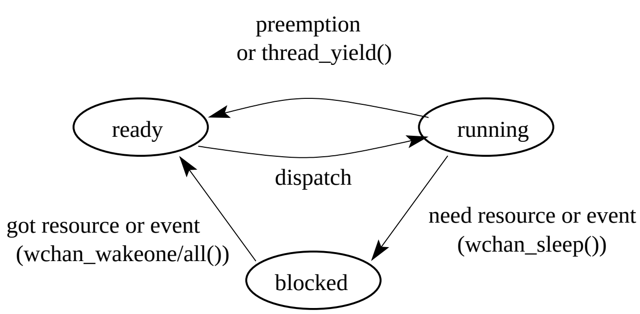

What Causes a Context Switches?

- 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

OS/161 Thread Stack after Voluntary Context Switch (`thread_yield()`)



Thread States



running: currently executing

ready: ready to execute

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

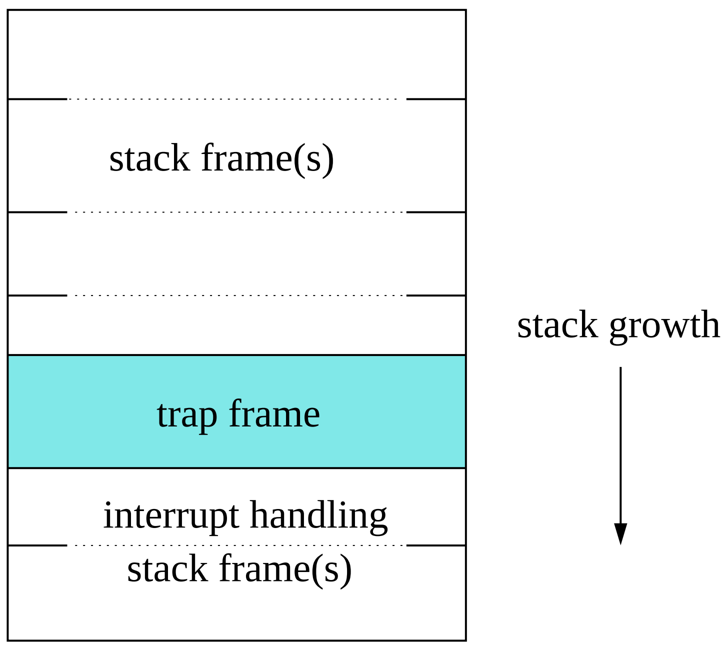
Preemption

- without preemption, a running thread could potentially run forever, without yielding, blocking, or exiting
- *preemption* means forcing 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

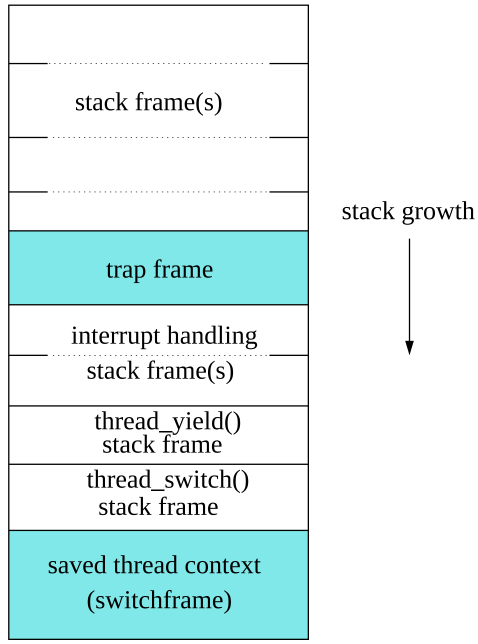


Preemptive Scheduling

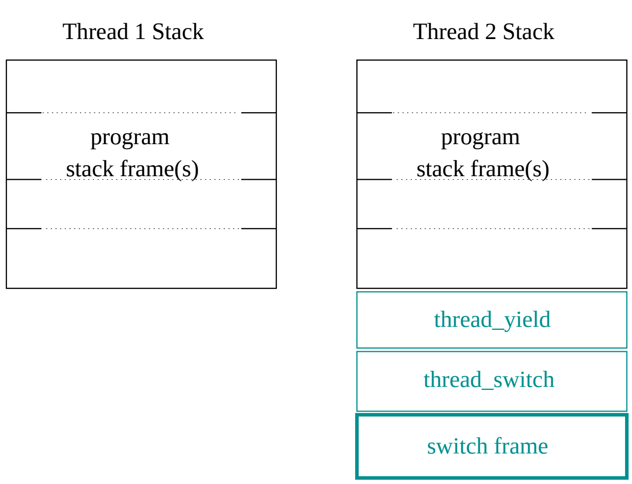
- A preemptive scheduler imposes a limit, called the *scheduling quantum* on how long a thread can run before being preempted.
- The quantum is an *upper bound* on the amount of time that a thread can run. It may block or yield before its quantum has expired.
- Periodic timer interrupts allow running 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*.

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

OS/161 Thread Stack after Preemption

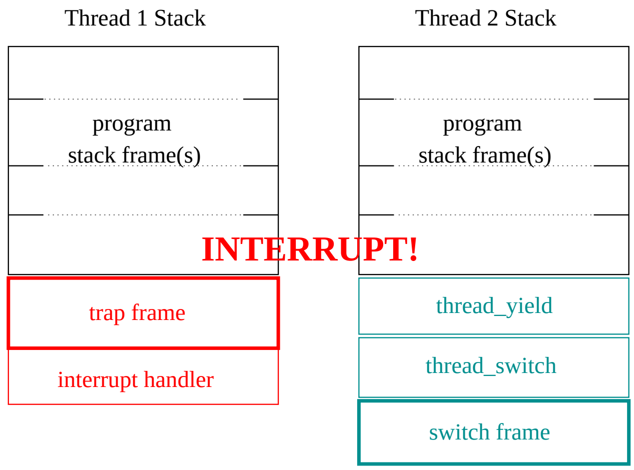


Two-Thread Example (Part 1)



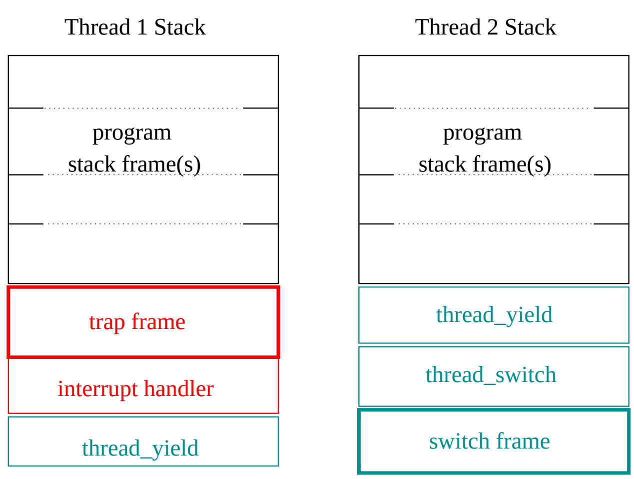
Thread 1 is running, thread two had previously yielded voluntarily.

Two-Thread Example (Part 2)



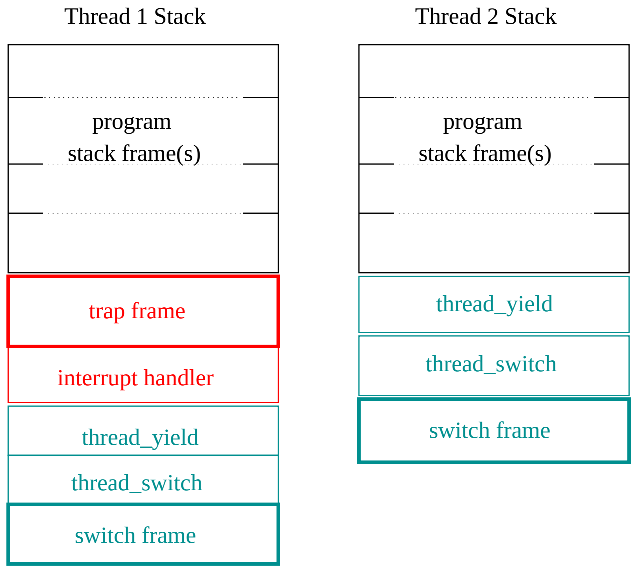
A time interrupt occurs! Interrupt handler runs.

Two-Thread Example (Part 3)



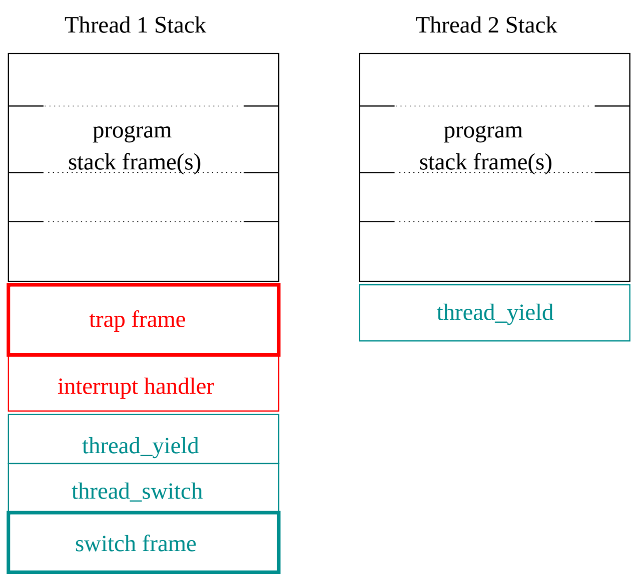
Interrupt handler decides Thread 1 quantum has expired.

Two-Thread Example (Part 4)



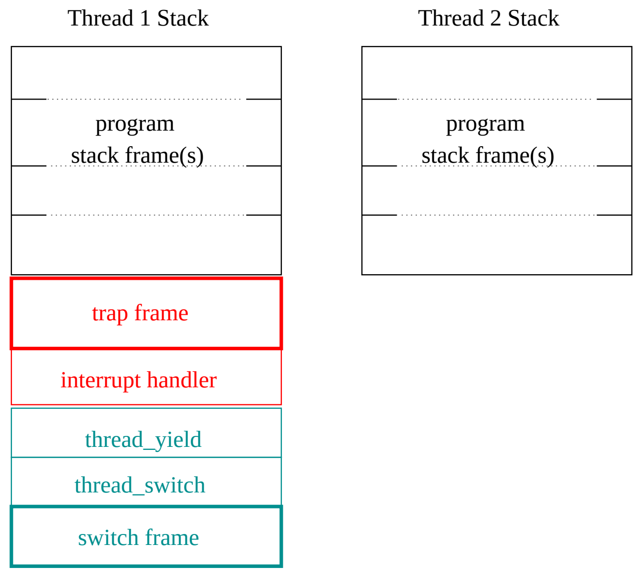
Scheduler chooses Thread 2 to run. Context switch.

Two-Thread Example (Part 5)



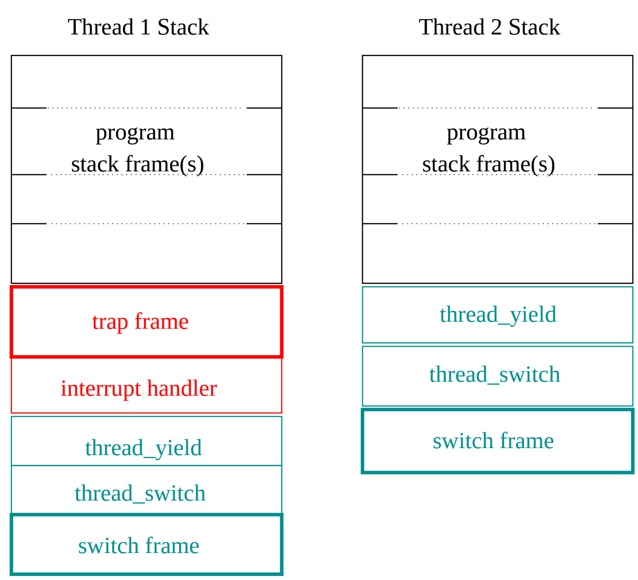
Thread 2 context is restored.

Two-Thread Example (Part 6)



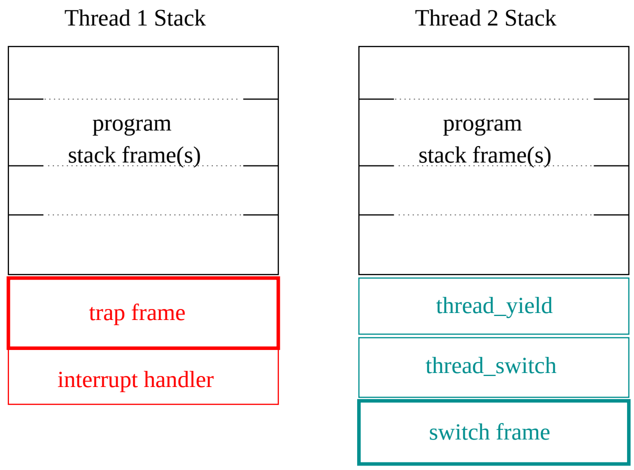
`thread_yield` finishes, Thread 2 program resumes.

Two-Thread Example (Part 7)



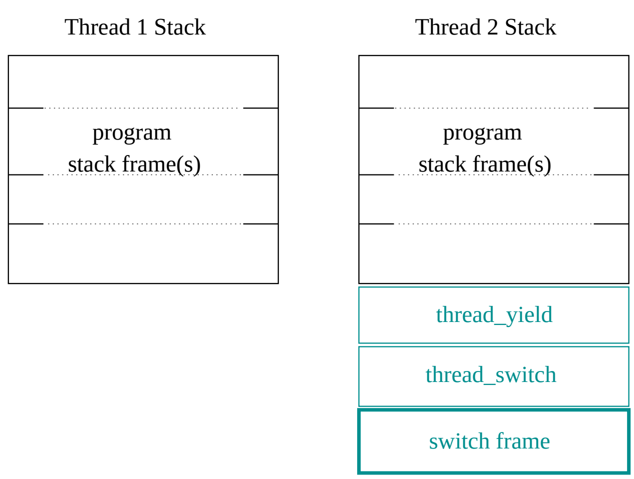
Later, Thread 2 yields again. Scheduler chooses Thread 1.

Two-Thread Example (Part 8)



Thread 1 context is restored, interrupt handler resumes.

Two-Thread Example (Part 9)



Interrupt handler restores state from trap frame and returns.