Review: Program Execution

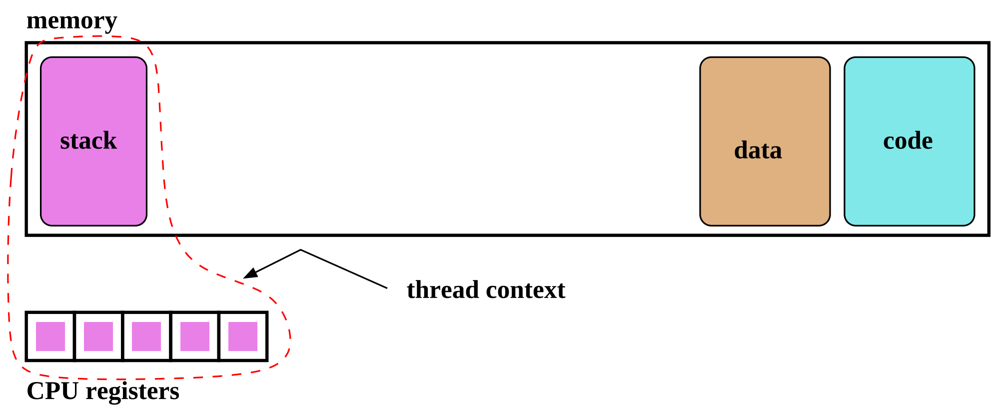
- Registers
  - program counter, stack pointer, . . .
- Memory
  - program code
  - program data
  - program stack containing procedure activation records
- CPU
  - fetches and executes instructions

What is a Thread?

- A thread represents the control state of an executing program.
- A thread has an associated *context* (or state), which consists of
  - the processor's CPU state, including the values of the program counter (PC), the stack pointer, other registers, and the execution mode (privileged/non-privileged)
  - a stack, which is located in the address space of the thread's process

Imagine that you would like to suspend the program execution, and resume it again later. Think of the thread context as the information you would need in order to restart program execution from where it left off when it was suspended.

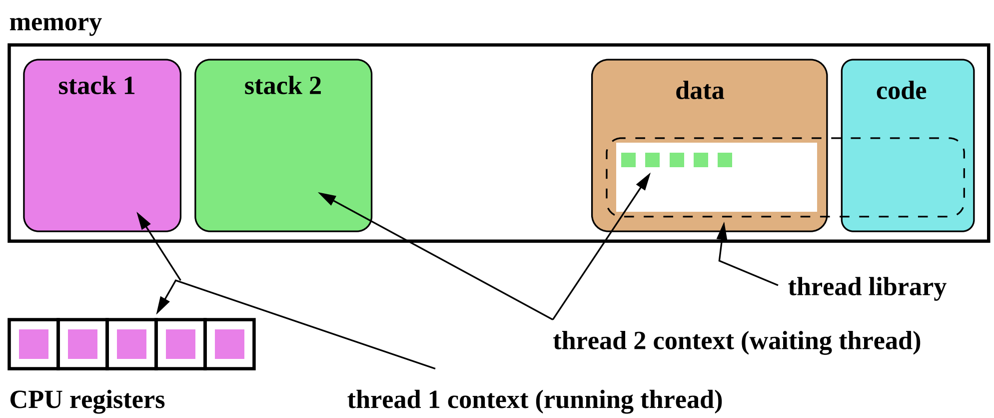
Thread Context



Concurrent Threads

- more than one thread may exist simultaneously (why might this be a good idea?)
- each thread has its own context, though they share access to program code and data
- on a uniprocessor (one CPU), at most one thread is actually executing at any time. The others are paused, waiting to resume execution.
- on a multiprocessor, multiple threads may execute at the same time, but if there are more threads than processors then some threads will be paused and waiting

Two Threads, One Running



Thread Interface (Partial), With OS/161 Examples

- a *thread library* implements threads
- thread library provides a thread interface, used by program code to manipulate threads
- common thread interface functions include
  - create new thread

    ```
    int thread_fork(const char *name, struct proc *proc,
                    void (*func)(void *, unsigned long),
                    void *data1, unsigned long data2);
    ```
  - end (and destroy) the current thread

    ```
    void thread_exit(void);
    ```
  - cause current thread to *yield* (to be discussed later)

    ```
    void thread_yield(void);
    ```
- see `kern/include/thread.h`

Example: Creating Threads Using `thread_fork()`

```
for (index = 0; index < NumMice; index++) {
  error = thread_fork("mouse_simulation thread",
    NULL, mouse_simulation, NULL, index);
  if (error) {
    panic("mouse_simulation: thread_fork failed: %s\n",
     strerror(error));
  }
}
/* wait for all of the cats and mice to finish */
for(i=0;i<(NumCats+NumMice);i++) {
  P(CatMouseWait);
}
```

What `kern/synchprobs/catmouse.c` actually does is slightly more elaborate than this.

Example: Concurrent Mouse Simulation Threads (simplified)

```
static void mouse_simulation(void * unusedpointer,
                             unsigned long mousenumber)
{
  int i; unsigned int bowl;

  for(i=0;i<NumLoops;i++) {
    /* for now, this mouse chooses a random bowl from
     * which to eat, and it is not synchronized with
     * other cats and mice
     */
    /* legal bowl numbers range from 1 to NumBowls */
    bowl = ((unsigned int)random() % NumBowls) + 1;
    mouse_eat(bowl);
  }
  /* indicate that this mouse is finished */
  V(CatMouseWait);

  /* implicit thread_exit() on return from this function */
}
```

Context Switch, Scheduling, and Dispatching

- the act of pausing the execution of one thread and resuming the execution of another is called a *(thread) context switch*
- what happens during a context switch?
  1. decide which thread will run next
  2. save the context of the currently running thread
  3. restore the context of the thread that is to run next
- the act of saving the context of the current thread and installing the context of the next thread to run is called *dispatching* (the next thread)
- sounds simple, but . . .
  - architecture-specific implementation
  - thread must save/restore its context carefully, since thread execution continuously changes the context
  - can be tricky to understand (at what point does a thread actually stop? what is it executing when it resumes?)

Scheduling

- scheduling means deciding which thread should run next
- scheduling is implemented by a *scheduler*, which is part of the thread library
- simple *round robin* scheduling:
  - scheduler maintains a queue of threads, often called the *ready queue*
  - the first thread in the ready queue is the running thread
  - on a context switch the running thread is moved to the end of the ready queue, and new first thread is allowed to run
  - newly created threads are placed at the end of the ready queue
- more on scheduling later . . .

Causes of Context Switches

- a call to **thread_yield** by a running thread
  - running thread *voluntarily* allows other threads to run
  - yielding thread remains runnable, and on the ready queue
- a call to **thread_exit** by a running thread
  - running thread is terminated
- running thread *blocks*, via a call to `wchan_sleep`
  - thread is no longer runnable, moves off of the ready queue and into a wait channel
  - more on this later . . .
- running thread is *preempted*
  - running thread *involuntarily* stops running
  - remains runnable, and on the ready queue

Preemption

- without preemption, a running thread could potentially run forever, without yielding, blocking, or exiting
- to ensure *fair* access to the CPU for all threads, the thread library may preempt a running thread
- 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. saves the current thread context (in OS/161, this is saved in a *trap frame* on the current thread's stack)
  2. determines which device caused the interrupt and performs device-specific processing
  3. restores the saved thread context and resumes execution in that context where it left off at the time of the interrupt.

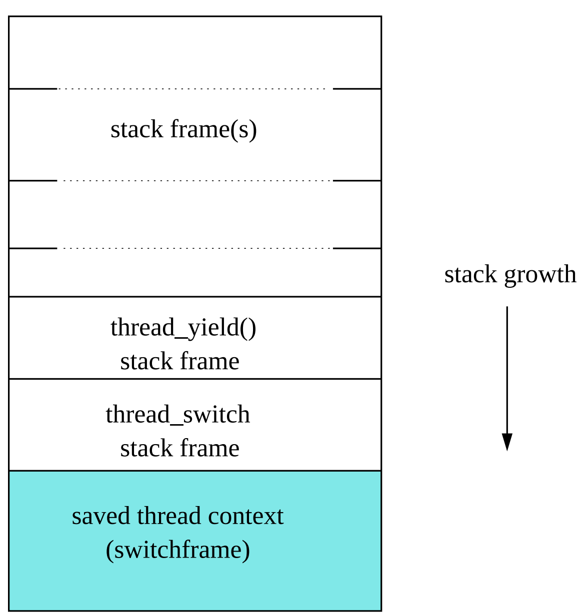
Preemptive Round-Robin Scheduling

- In preemptive round-robin scheduling, the thread library imposes a limit on the amount of time that a thread can run before being preempted
- the amount of time that a thread is allocated is called the scheduling *quantum*
- when the running thread's quantum expires, it is preempted and moved to the back of the ready queue. The thread at the front of the ready queue is dispatched and allowed to run.
- the quantum is an *upper bound* on the amount of time that a thread can run once it has been dispatched
- the dispatched thread may run for less than the scheduling quantum if it yields, exits, or blocks before its quantum expires

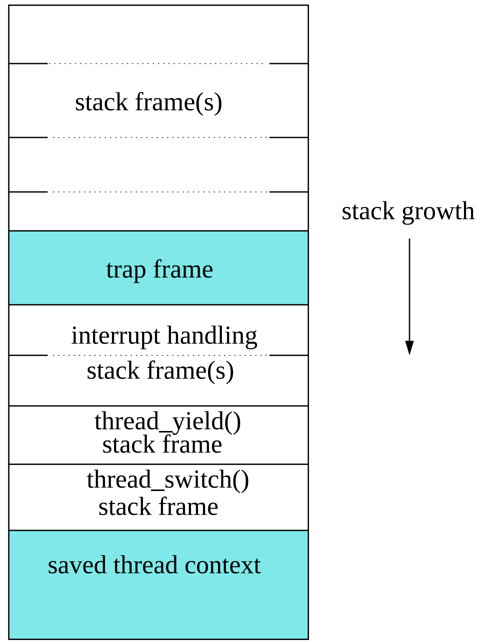
Implementing Preemptive Scheduling

- suppose that the system timer generates an interrupt every t time units, e.g., once every millisecond
- suppose that the thread library wants to use a scheduling quantum $q = 500t$, i.e., it will preempt a thread after half a second of execution
- to implement this, the thread library can maintain a variable called `running_time` to track how long the current thread has been running:
  - when a thread is intially dispatched, `running_time` is set to zero
  - when an interrupt occurs, the timer-specific part of the interrupt handler can increment `running_time` and then test its value
    * if `running_time` is less than q, the interrupt handler simply returns and the running thread resumes its execution
    * if `running_time` is equal to q, then the interrupt handler invokes `thread_yield` to cause a context switch

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



OS/161 Thread Stack after Preemption



Implementing Threads

- the thread library is responsibile for implementing threads
- the thread library stores threads' contexts (or pointers to the threads' contexts) when they are not running
- the data structure used by the thread library to store a thread context is sometimes called a *thread control block*

In the OS/161 kernel's thread implementation, thread contexts are stored in `thread` structures.

The OS/161 `thread` Structure

```
/* see kern/include/thread.h */

struct thread {
 char *t_name;              /* Name of this thread */
 const char *t_wchan_name; /* Wait channel name, if sleeping */
 threadstate_t t_state;    /* State this thread is in */

 /* Thread subsystem internal fields. */
 struct thread_machdep t_machdep; /* Any machine-dependent goo */
 struct threadlistnode t_listnode; /* run/sleep/zombie lists */
 void *t_stack;                   /* Kernel-level stack */
 struct switchframe *t_context; /* Register context (on stack) */
 struct cpu *t_cpu;               /* CPU thread runs on */
 struct proc *t_proc;             /* Process thread belongs to */
 ...
```


Review: MIPS Register Usage

```
R0,  zero = ## zero (always returns 0)
R1,  at   = ## reserved for use by assembler
R2,  v0   = ## return value / system call number
R3,  v1   = ## return value
R4,  a0   = ## 1st argument (to subroutine)
R5,  a1   = ## 2nd argument
R6,  a2   = ## 3rd argument
R7,  a3   = ## 4th argument
```

Review: MIPS Register Usage

```
R08-R15,  t0-t7 = ## temps (not preserved by subroutines)
R24-R25,  t8-t9 = ## temps (not preserved by subroutines)
                  ##   can be used without saving
R16-R23,  s0-s7 = ## preserved by subroutines
                  ##   save before using,
                  ##   restore before return
R26-27,   k0-k1 = ## reserved for interrupt handler
R28,      gp    = ## global pointer
                  ## (for easy access to some variables)
R29,      sp    = ## stack pointer
R30,      s8/fp = ## 9th subroutine reg / frame pointer
R31,      ra    = ## return addr (used by jal)
```


Dispatching 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)
```

Dispatching 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
```


Dispatching on the MIPS (Notes)

- Not all of the registers are saved during a context switch
- This is because the context switch code is reached via a call to `thread_switch` and by convention on the MIPS not all of the registers are required to be preserved across subroutine calls
- thus, after a call to `switchframe_switch` returns, the caller (`thread_switch`) does not expect all registers to have the same values as they had before the call - to save time, those registers are not preserved by the switch
- if the caller wants to reuse those registers it must save and restore them