Review: Program Execution

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

Review: MIPS Register Usage

```
See also: kern/arch/mips/include/asmdefs.h
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

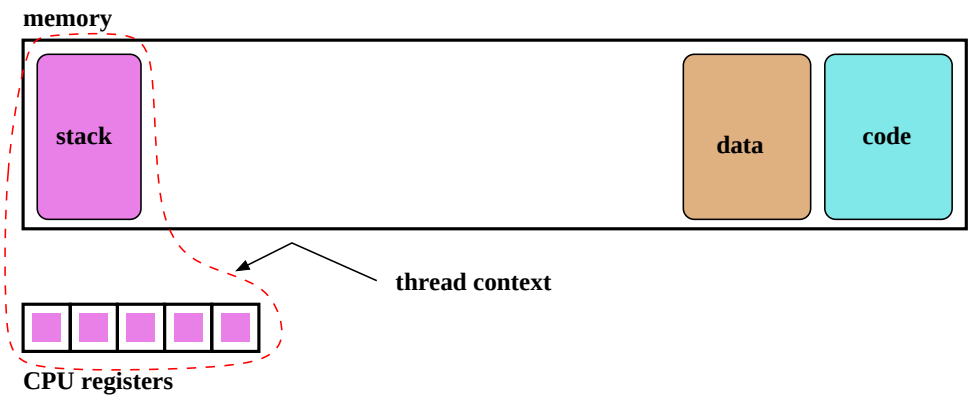
```
R08-R15,  t0-t7 = ## temps (not preserved by subroutines)
R24-R25,  t8-t9 = ## temps (not preserved by subroutines)
                  ##   can be used without saving
R16-R23,  s0-s7 = ## preserved by subroutines
                  ##   save before using,
                  ##   restore before return
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)
```


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 may 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 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

Example: Concurrent Mouse Simulations

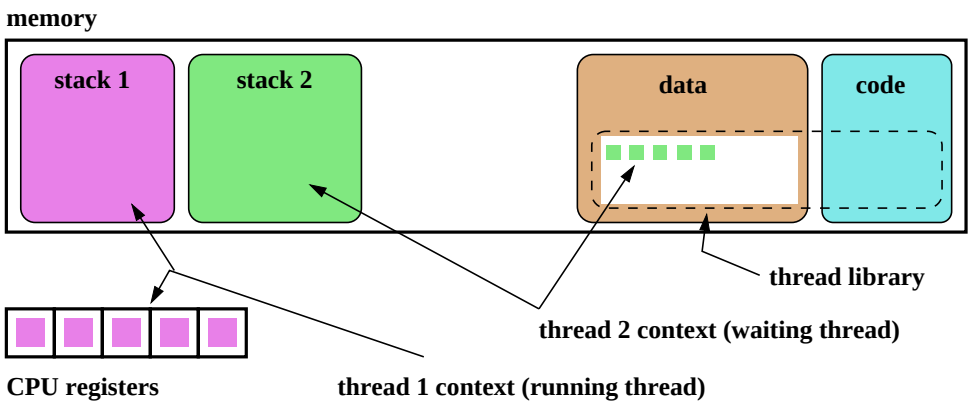
```
static void mouse_simulation(void * unusedpointer,
                             unsigned long mousenumber)
{
  int i; unsigned int bowl;

  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,1);
  }

  /* indicate that this mouse is finished */
  V(CatMouseWait);
}
```


Implementing Threads

- a 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.

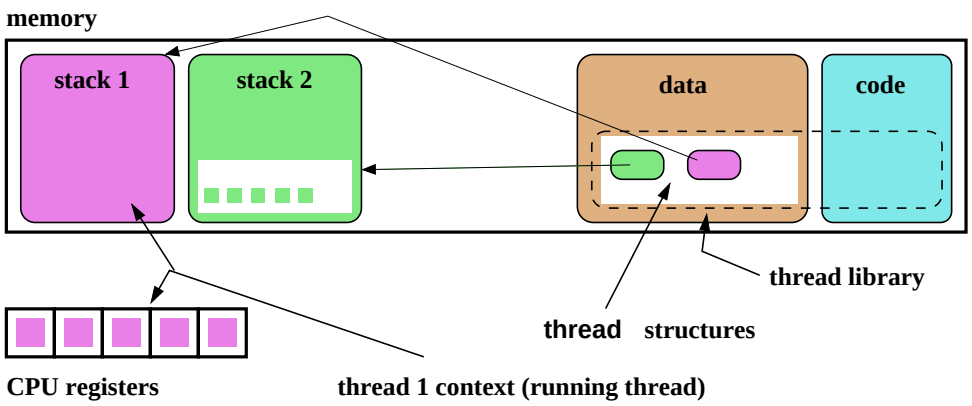
Thread Library and Two Threads



The OS/161 `thread` Structure

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

struct thread {

 /* Private thread members - internal to the thread system */

struct pcb t_pcb;        /* misc. hardware-specific stuff */
char *t_name;            /* thread name */
const void *t_sleepaddr; /* used for synchronization */
char *t_stack;           /* pointer to the thread's stack */


 /* Public thread members - can be used by other code      */

struct addrspace *t_vmspace;  /* address space structure */
struct vnode *t_cwd;          /* current working directory */
};
```

Thread Library and Two Threads (OS/161)



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?)

Dispatching on the MIPS (1 of 2)

```
/* see kern/arch/mips/mips/switch.S */
mips_switch:
    /* a0/a1 points to old/new thread's control block */

   /* Allocate stack space for saving 11 registers. 11*4 = 44 */
   addi sp, sp, -44

   /* Save the registers */
   sw ra, 40(sp)
   sw gp, 36(sp)
   sw s8, 32(sp)
   sw s7, 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 control block */
   sw sp, 0(a0)
```


Dispatching on the MIPS (2 of 2)

```
   /* Get the new stack pointer from the new control block */
   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 s7, 28(sp)
   lw s8, 32(sp)
   lw gp, 36(sp)
   lw ra, 40(sp)
   nop /* delay slot for load */

   j ra             /* and return. */
   addi sp, sp, 44 /* in delay slot */
   .end mips_switch
```

Thread Library Interface

- the thread library interface allows program code to manipulate threads
- one key thread library interface function is *Yield()*
- Yield() causes the calling thread to stop and wait, and causes the thread library to choose some other waiting thread to run in its place. In other words, Yield() causes a context switch.
- in addition to `Yield()`, thread libraries typically provide other thread-related services:
  - create new thread
  - end (and destroy) a thread
  - cause a thread to *block* (to be discussed later)

The OS/161 Thread Interface (incomplete)

```
/* see kern/include/thread.h */
/* create a new thread */
int thread_fork(const char *name,
                void *data1, unsigned long data2,
                void (*func)(void *, unsigned long),
                struct thread **ret);

/* destroy the calling thread */
void thread_exit(void);

/* let another thread run */
void thread_yield(void);

/* block the calling thread */
void thread_sleep(const void *addr);

/* unblock blocked threads */
void thread_wakeup(const void *addr);
```

Creating Threads Using `thread_fork()`

```
/*  from catmouse() in kern/asst1/catmouse.c */
/*  Start NumMice mouse_simulation() threads. */
for (index = 0; index < NumMice; index++) {
  error = thread_fork("mouse_simulation thread",NULL,index,
                      mouse_simulation,NULL);
  if (error) {
    panic("mouse_simulation: thread_fork failed: %s\n",
          strerror(error));
  }
}

/* wait for all of the cats and mice to finish before
   terminating */
for(i=0;i<(NumCats+NumMice);i++) {
  P(CatMouseWait);
}
```


Scheduling

- scheduling means deciding which thread should run next
- scheduling is implemented by a *scheduler*, which is part of the thread library
- simple FIFO 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 . . .

Preemption

- `Yield()` allows programs to *voluntarily* pause their execution to allow another thread to run
- sometimes it is desirable to make a thread stop running even if it has not called `Yield()`
- this kind of *involuntary* context switch is called *preemption* of the running thread
- to implement preemption, the thread library must have a means of "getting control" (causing thread library code to be executed) even through the application 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.

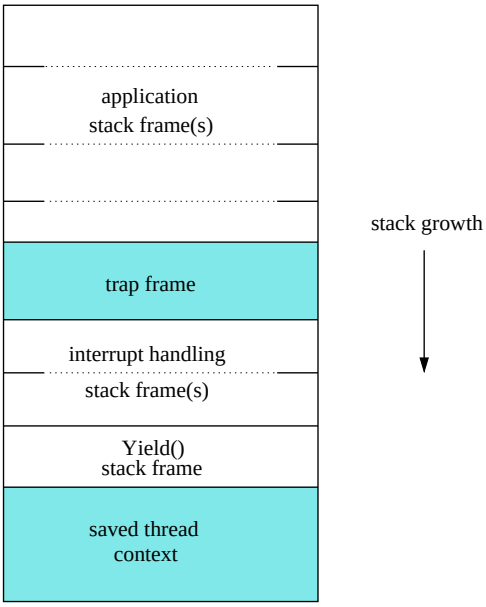
Round-Robin Scheduling

- *round-robin* is one example of a preemptive scheduling policy
- round-robin scheduling is similar to FIFO scheduling, except that it is preemptive
- as in FIFO scheduling, there is a ready queue and the thread at the front of the ready queue runs
- unlike FIFO, a limit is placed on the amount of time that a thread can run before it is 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.

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 `Yield()` to cause a context switch

OS/161 Stack after Preemption



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