Slide 1

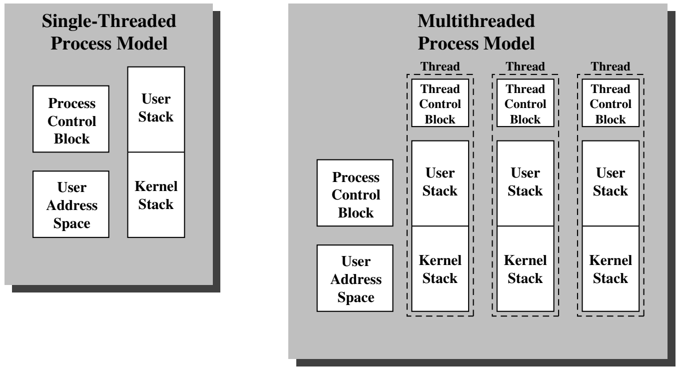
# Threads — Part 2: Internals

COMP3231 Operating Systems

2004/S2

Slide 2

## Thread Switch




Note: Meaning of PCB is different in OS/161!

Slide 3

## Thread Creation (Spawn)

① Assign unique thread identifier (thread ID)
② Allocate and initialise a TCB
③ Allocate a stack and set pointer to it in TCB
④ Set up links to appropriate lists/queues
- lists of threads
- lists of threads belonging to process
- ready queue

⑤ Update appropriate process info
- e.g., accounting (charge for thread's memory)

Correspondingly for thread termination...

Slide 4

## Thread Switch

- can happen after the thread yields the CPU, or
- any time the OS is invoked:
  - on a system call
    - mandatory if system call blocks or on `exit()`
  - on an exception
    - mandatory if offender is killed
  - on an interrupt
    - triggering a dispatch is the main purpose of the timer interrupt

➨ Thread switch can happen between any two instructions!

## Context Switch

Slide 5

- Thread switch must be transparent for threads:
  - When dispatched again, thread should not notice that something else was running in the meantime (except for elapsed time, possibly changes to global data)
- OS must save all state that affects the thread
  - This state is called the thread context
  - Switching between threads consequently results in a context switch
  - Hardware support is necessary in case of exception or interrupt

## Thread Switch

Slide 6

① Save state of executing thread (to running TCB)
- save registers, pc
- perform other updates to TCB of running thread
  - e.g., update total CPU time used
- set state to ready, blocked
- Link TCB to wait queue if appropriate

② Select another thread for execution (scheduling)

③ set running TCB pointer to new thread

④ Activate newly scheduled thread (dispatching)
- update TCB of thread to be scheduled
- Restore context of the selected thread

⑤ Return to user mode (e.g. `rfe` instruction)

## Thread Switch in OS/161

Slide 7

What happpens in OS/161 if a thread uses up all its time?

Main steps:

① timer interrupt

② hardware saves pc, exception cause to co-processor registers

③ general exception handler calls timer interrupt handler

④ timer interrupt handler causes new thread to be activated

## Processor State Example: MIPS R3000

Slide 8

- ➜ 32 general purpose (GP) registers, plus `hi`, `lo`
- ➜ PC
- ➜ remainder of status in co-processor 0 (system co-processor, CP0) registers
  - STATUS register
  - exception CAUSE register
  - EPC: pre-exception PC value
  - MMU registers, etc.
- ➜ accessed via special instructions:
  - `mfc0`: copy CP0 register to GP register
  - `mtc0`: copy GP register to CP0 register

Slide 9

MIPS-32 general purpose registers:

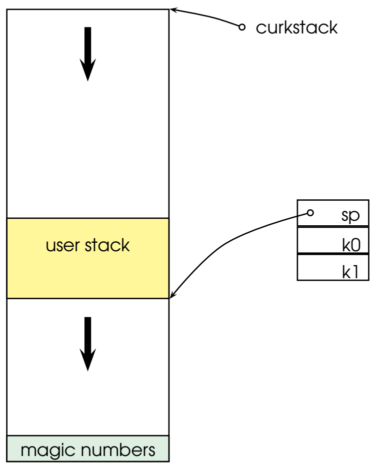
| register | menmonic | convention |
|---|---|---|
| `r0` | `zero` | always zero |
| `r1` | `AT` | assembler temporary |
| `r2–r3` | `v0–v1` | integer function results |
| `r4–r7` | `a0–a3` | first four integer function args |
| `r8–r15` | `t0–t7` | temporary (not preserved) |
| `r16–r23` | `s0–s7` | preserved across calls |
| `r24–r25` | `t8–t9` | temporary (not preserved) |
| `r26–r27` | `k0–k1` | kernel reserved |
| `r28` | `gp` | global (data segment) pointer |
| `r29` | `sp` | stack pointer |
| `r30` | `s8/fp` | frame pointer (preserved) |
| `r31` | `ra` | return address |

Slide 10

MIPS assembly instructions:

- ➜ General format:
  *operator  destination, source1(, source2)*
- ➜ Store word (`sw`): *source, destination*
- ➜ destination is always a register
- ➜ source is a register or an immediate value
- ➜ for *load* and *store* instructions:
  - *destination* is the register to be loaded/stored
  - *source1* contains the memory address (no *source2*)
- ➜ register-relative address mode with optional constant offset.
  Example : `lw a0 4(a1)`
  - loads `a0`
  - from address ((contents of `a1`) + 4)

Slide 11




Slide 12

## MIPS Timer Interrupt

- MIPS processor does the following:

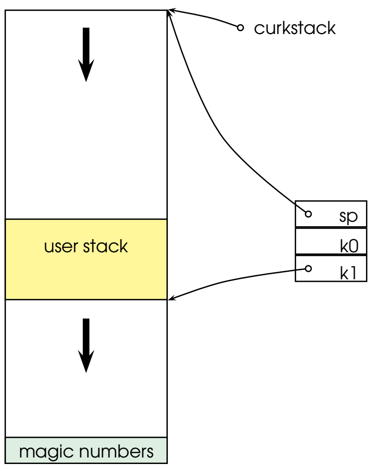
| | | | |
|---|---|---|---|
| CP0.EPC | $\leftarrow$ | PC | |
| CP0.CAUSE.ExcCode | $\leftarrow$ | 0 | ; interrupt |
| CP0.CAUSE.IP | $\leftarrow$ | 0x01 | ; clock interrupt |
| PC | $\leftarrow$ | 0x8000 0080 | ; gen excpt handler |
| CP0.STATUS.EXL | $\leftarrow$ | 1 | |

- Setting CP0.STATUS.EXL sets exception mode:
  - disables interrupts
  - turns on kernel mode

Slide 13

```
exception:
   move k1, sp              /* Save previous stack pointer in k1 */
   mfc0 k0, c0_status       /* Get status register */
   andi k0, k0, CST_KUp     /* Check the we-were-in-user-mode bit */
   beq  k0, $0, 1f          /* If clear, from kernel, already have stack */

   /* Coming from user mode - load kernel stack into sp */
   la k0, curkstack         /* get address of "curkstack" */
   lw sp, 0(k0)             /* get its value */
1:
   mfc0 k0, c0_cause        /* Now, load the exception cause. */
   j common_exception       /* Skip to common code */
   nop
```

Slide 14





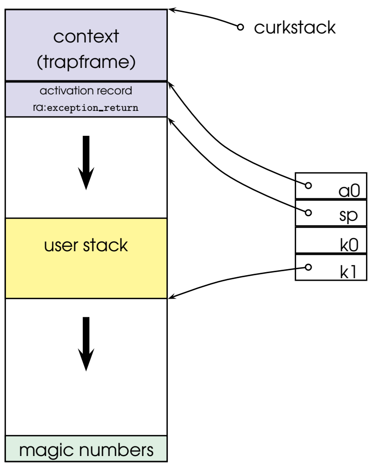

Slide 15

```
common_exception:

  addi sp, sp, -164  /* allocate space for trap frame */
                     /* and minimal argument block    */

  /* Save context information */
  sw s8, 156(sp)       /* save s8 */
  sw gp, 148(sp)       /* save gp */
  sw k1, 152(sp)       /* real saved sp */

  mfc0 k1, c0_epc      /* Copr.0 reg 13 == PC for exception */
  sw k1, 160(sp)       /* real saved PC */

  sw t9, 136(sp)
  .....
  sw AT, 40(sp)
  sw ra, 36(sp)
```

Slide 16

```
  /* Prepare to call mips_trap(struct trapframe *)   */

  addiu a0, sp, 16             /* set argument */
  jal mips_trap                /* call it */
  nop
exception_return:

  /*     16(sp)                   no need to restore tf_vaddr */
  lw t0, 20(sp)                /* load status register value into t0 */
  .....
```

**Slide 17**




**Slide 18**

```
void mips_trap(struct trapframe *tf)
{
  ...

     /*
      * Call the interrupt handler.
      * Note that interrupts are off here so it's ok to access
      * the global lamebus structure.
      */
      lamebus->ls_irqfuncs[slot](lamebus->ls_devdata[slot]);
```

- ➜ interrupt handler of timer interrupt: `ltimer_irq`
- ➜ calls `hardclock`, which in turn calls `thread_yield`



**Slide 19**

```
thread_yield(void)
 {
         int spl = splhigh();
         ...
         mi_switch(S_READY);
         splx(spl);
  }
```

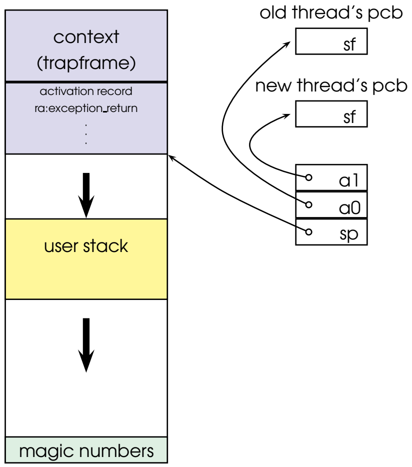
**Slide 20**

```
static void mi_switch(threadstate_t nextstate)
{
     .....
     next = scheduler();

     /* update curthread */
     curthread = next;

     /*
      * Call the machine-dependent code that actually does the
      * context switch.
      */
      md_switch(&cur->t_pcb, &next->t_pcb);
```

Slide 21

```
struct pcb {
        u_int32_t pcb_switchstack;  // stack saved during context switch
        .......
};


struct switchframe {
        u_int32_t sf_s0;
        u_int32_t sf_s1;
        .....
        u_int32_t sf_s8;
        u_int32_t sf_gp;
        u_int32_t sf_ra;
};
```

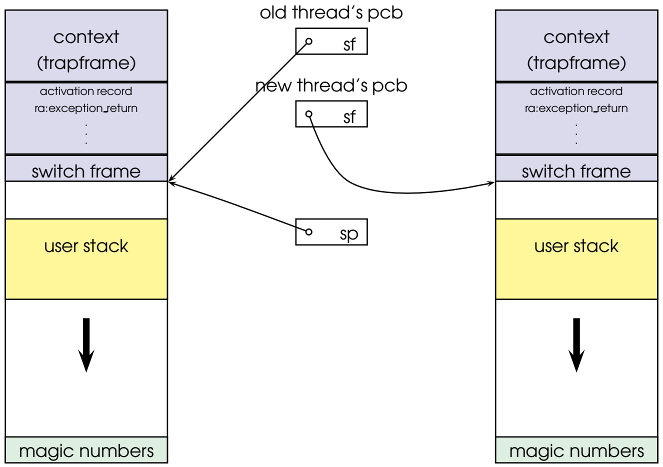
Slide 22




Slide 23

```
/*
 * a0 contains a pointer to the old thread's struct pcb.
 * a1 contains a pointer to the new thread's struct pcb.
 *
 */

/* 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   s0, 0(sp)

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

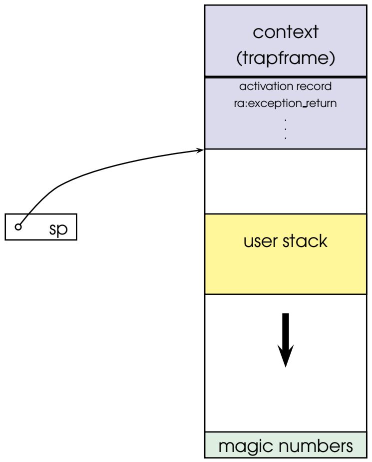
Slide 24



Slide 25

```
/* Get the new stack pointer from the new pcb */
   lw   sp, 0(a1)

   /* Now, restore the registers */
   lw   s0, 0(sp)
        ......
   lw   gp, 36(sp)
   lw   ra, 40(sp)


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

Slide 26

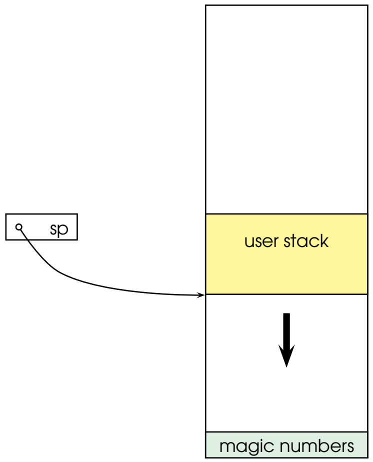





Slide 27

```
exception_return:
  lw t0, 20(sp)               /* load status register value into t0 */

 /* restore special registers */
  lw t1, 28(sp)
  .....

  /* load the general registers */
  lw ra, 36(sp)
  .....
  lw k0, 160(sp)              /* fetch exception return PC into k0 */

  lw sp, 152(sp)              /* fetch saved sp (must be last) */

  /* done */
  jr k0                       /* jump back */
  rfe                         /* in delay slot */
```

Slide 28



Slide 29

## THREADS

Do threads need OS support or is it possible to implement them as user-level library?

What about

- ➜ Context switch?
- ➜ Preemption?
- ➜ Scheduling?
- ➜ Handling of blocking system calls?

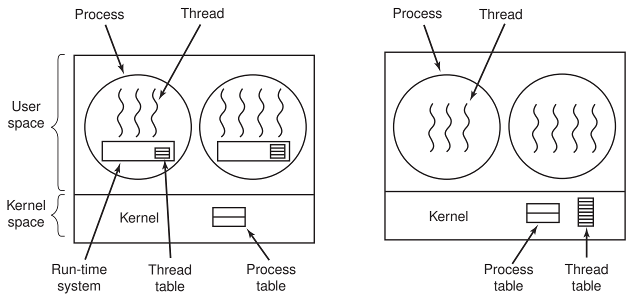
Slide 30

## USER-LEVEL THREADS

- ➜ All thread management is done by the application (library)
- ➜ Runs in usermode
- ➜ The kernel is not aware of the existence of threads
  - User-level threads are not scheduled by the kernel
  - Pure application/library level construct
- ➜ Used to enhance modularisation
- ➜ Also called co-routines



Slide 31




Slide 32

## USER-LEVEL THREADS VS KERNEL LEVEL THREADS

- ✔ Scheduling policy tailored to specific application
- ✔ Extremely low overhead
- ✘ Process blocks if one of its threads blocks
  - can be avoided by using wrapper functions for all possibly blocking operations, introduces extra system calls
- ✘ No inter-process parallelism possible on machines with multiple CPUs

Slide 33

## Combined Approaches

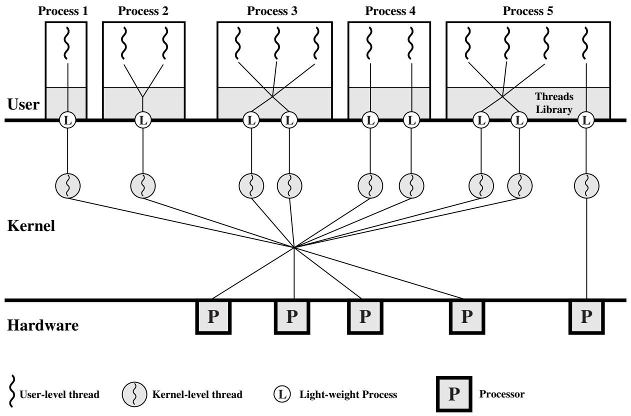
Try to get best of both worlds!

Example: Solaris

- ➜ Library offers (user-level) thread interface
- ➜ OS supports kernel-level threads
- ➜ Thread library provides API for binding one or more user-level threads to a kernel-level thread
- ➜ Most thread management done explicitly at user level

Slide 34

## Solaris Thread Architecture




Slide 35

## Relationship Threads vs. Processes

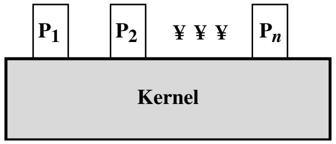
| T:P | Description | Example Systems |
|---|---|---|
| 1:1 | Each thread of execution has its own address space and resources. | Traditional UNIX, legacy OSes |
| m:1 | A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process. | Modern Unices, Win2k, OS/2, OS/390, Mach |
| 1:n | Threads migrate from one process to another. Thread can change its protection context dynamically | Ra (Clouds), Emerald |
| m:n | Combination of m:1 and 1:n | TRIX, Pebble, Opal, Mungi |

Slide 36

## Execution Context of the Operating System

Non-process Kernel:

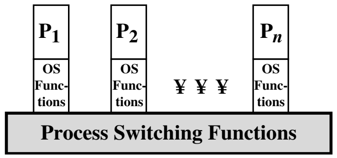
- ➜ Execute kernel outside of any process context
- ➜ Separate context for execution of OS code
  - OS context may be automatically switched by hardware
- ➜ Traditional model



Slide 37

## Execution Within User Processes:

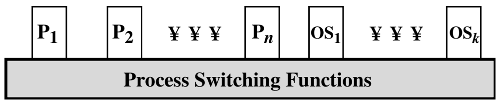
- ➜ OS software executes within context of a user process
- ➜ Process in privileged mode while executing OS code
  - has access to additional (kernel) memory
- ➜ E.g: UNIX




Slide 38

## Execution Within Separate Process(es):

- ➜ Process-based (server-based) OS
- ➜ Separate process for major OS functions
- ➜ Clients use message-passing IPC to invoke services
  - Aids distribution
- ➜ OS processes may execute in unprivileged (user) mode
- ➜ E.g: Windows-2000





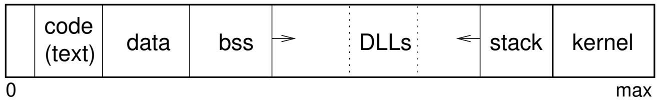

Slide 39

# UNIX SVR4 PROCESS MANAGEMENT

- ➜ Mostly follows in-process model
- ➜ In addition has "kernel processes" (daemons)
- ➜ Several parts of kernel data:
  - user-level context (text, data, stack)
  - register context
  - system-level context
    - process table entry:
      - global table, always entirely accessible by kernel
      - process state, IDs, prio, links to other data
    - "U area": process resources

Slide 40

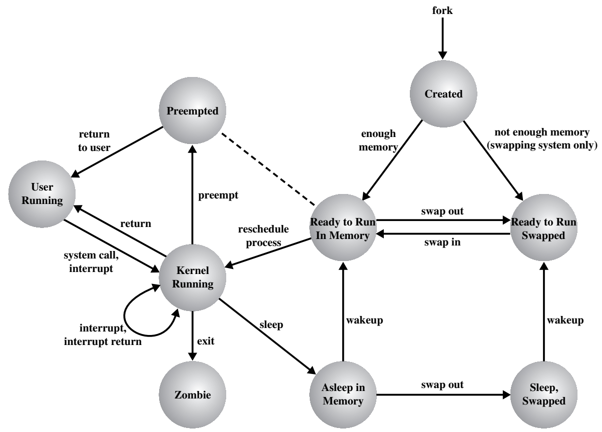
## Typical Address Space Layout (UNIX):




- ➜ 0-th page typically not used
- ➜ text segment is read-only
- ➜ data segment is initialised data (part R/O)
- ➜ bss segment is uninitialised data (heap), can grow
- ➜ shared libraries (DLLs) allocated in free middle region
- ➜ stack at top of user space, grows downward
- ➜ kernel space is in reserved (shared) region

Slide 41

## UNIX process states (single-threaded):




Slide 42

## Main process system calls:

| | |
|---|---|
| `fork()` | Creates process with copy of parent's process image |
| `exec()` | Replaces image of calling process with new executable file |
| `exit()` | Terminate calling process. Return exit status to parent |
| `wait()` | Wait for (specific or any) child to terminate. Collect exit status |
| `kill()` | Sends signal to process. Default action for many signals is to kill recipient, but may install signal handler |

Slide 43

## Typical use:

```
#include <stdio.h>
int pid;
pid = fork();
 if (pid < 0) {
   perror("fork() failed");
   exit(1);
}  else  if (pid > 0)            /* We're the parent! */
    printf("child PID=%d\n", pid);
}  else {                        /* We're the child! */
   execve(file);
   sprintf(STDERR, "Exec failed!");
   exit(1);
}
```

Slide 44

## Threads in Linux:

- ➜ Linux has no real threads (with reduced context)
- ➜ Provides `clone()` system call
  - generalisation of Unix `fork()`
  - creates new process
  - parent and child can share (part of) address space
  - effectively a shortcut for `fork(); mmap()`
  - child has complete process context
  - `LinuxThreads` lib implemented using `clone`
- ➜ PThreads package provides user-level threads
  - can be bound to Linux "threads"
  - gives some approximation of lightweight threads
  - similar to Solaris
    - but Solaris' "lightweight processes" are faster

## LINUX clone SYSTEM CALL

Slide 45

- ➜ CLONE_PARENT: (Linux 2.4 onwards) parent of new task same as parent of caller
- ➜ CLONE_FS: share file system information
- ➜ CLONE_FILES: share file descriptor table
- ➜ CLONE_SIGHAND: share table of signal handler
- ➜ CLONE_VFORK: parent suspended until child releases vm resources (exit(), execve())
- ➜ CLONE_VM: parent and child run in the same memory space
- ➜ CLONE_THREAD: (Linux 2.4 onwards) parent and child share the same thread id