Slide 1

Threads and Processes — Part 1

COMP3231/COMP9201 Operating Systems

2005/S2

Slide 2

Major Requirements of an OS

- ➜ Interleave the execution of several programs
  - to maximize utilization of CPU and other resources while providing reasonable response time
  - to support multiple user working interactively
  - for convenience (e.g., compile program while editing other file)
- ➜ Allocate resources required for execution of programs
- ➜ Support communication between executing programs

Slide 3

Previously, we listed several definitions of the term Process:

- ✶ A program in execution
- ✶ An instance of a program running on a computer
- ✶ A unit of execution characterised by
  - a single, sequential thread of execution
  - a current state
  - an associated set of system resources (memory, devices, files)
- ✶ Unit of resource ownership

Many applications consist of more than one thread of execution which share resources

$\Longrightarrow$ distinction between thread and process

Slide 4

Processes and Threads

Process:

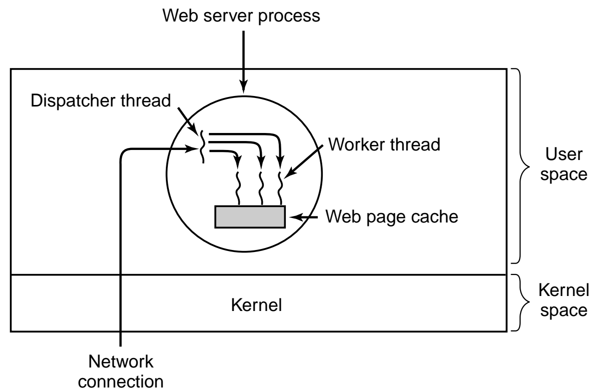
- ➜ "Owner" of resources allocated for individual program execution
- ➜ Can encompass more than one thread of execution
  - Outlook, Evolution: different threads for calendar, mail components etc

Thread:

- ➜ Unit of execution
- ➜ Belongs to a process
- ➜ Can be traced
  - list the sequence of instructions that execute

EXAMPLE: WEB SERVER

Slide 5



SINGLE-THREADED WEB SERVER IMPLEMENTATIONS

Slide 6

➜ Sequential processing of requests:
- web server gets request, processes it, accepts next request
- CPU idle while data retrieved from disk
- Poor performance

➜ Finite-State Machine:
- use non-blocking read
- program records state of current request
- gets next event
- on reply (signal) from disk, fetches and processes data
- good performance, complicated to implement and debug

➜ Processes instead of Threads
- Communicate by sharing data, messages

ADVANTAGES OF THREADS

Slide 7

① Program does not stall when one of its operations blocks
- save contents of a page to disk while downloading other page

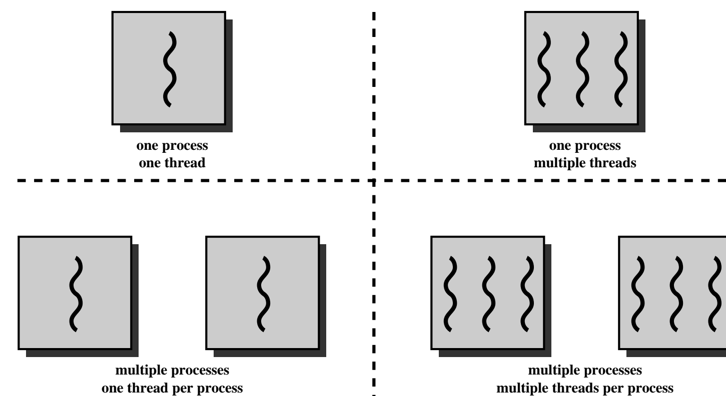
② Overhead for thread creation and destruction is less than for processes (depending on implementation, can be about a factor of 100 faster)

③ Simplification of programming model

④ Performace gains on machines with multiple CPU's

THREADS AND PROCESSES

Slide 8



Slide 9

Threads and Processes

- ➜ Single process, single thread
  - MS-DOS, old MacOS
- ➜ Single process, multiple threads
  - OS/161 as distributed
- ➜ Multiple processes, single thread
  - traditional Unix
- ➜ Multiple processes, multiple threads
  - modern Unices (Solaris, Linux), Windows-2000

Note: Literature (incl. textbooks) often do not cleanly distinguish those concepts (for historical reasons)!

Slide 10

Logical traces of threads:

Thread A	Thread B	Thread C
5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

5000: Starting address of code for Thread A

8000: Starting address of code for Thread B

12000:Starting address of code for Thread C

Slide 11

Logical traces of threads:

1	5000	
2	5001	
3	5002	
4	5003	
5	5004	
6	5005	
		Time out
7	100	
8	101	
9	102	
10	103	
11	104	
12	105	
13	8000	
14	8001	
15	8002	
16	8003	
17	8004	
		I/O request
18	100	
19	101	
20	102	
21	103	
22	104	
23	105	
24	12000	
25	12001	
26	12002	
27	12003	
28	12004	
29	12005	
		Time out
30	100	
31	101	
32	102	
33	103	
34	104	
35	105	

Slide 12

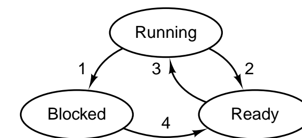
Thread States

Three states (may be more, depending on implementation):

① Running: currently active, using CPU

② Ready: runnable, waiting to be scheduled

③ Blocked: waiting for an event to occur (I/O, alarm)



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

Slide 13

Reasons for Leaving the Running State

- ➜ Thread terminates
  - exit() system call (voluntary termination)
  - killed by another thread
  - killed by OS (due to exception)
- ➜ Thread cannot continue execution
  - blocked waiting for event (I/O)
- ➜ OS decides to give someone else a chance
  - requires the OS to be invoked
    - via system call or exception
    - via interrupt
- ➜ Thread voluntarily gives another thread a chance
  - yield() system call

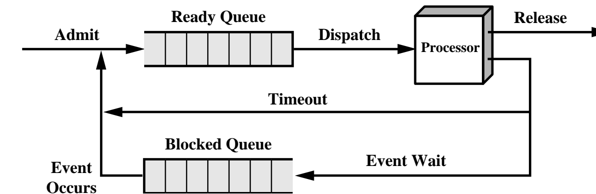
Slide 14

Non-running Threads

- ➜ Many separate reasons for a thread not running
  - another thread is running on the CPU
  - thread is blocked (waiting for an event)
  - thread is in initialisation phase (during creation)
  - thread is being cleaned up (during exit, kill)
- ➜ Dispatching ought to be fast
  - Shouldn't search through all threads to find runnable one
  - Achieved by distinguishing more thread states

Slide 15

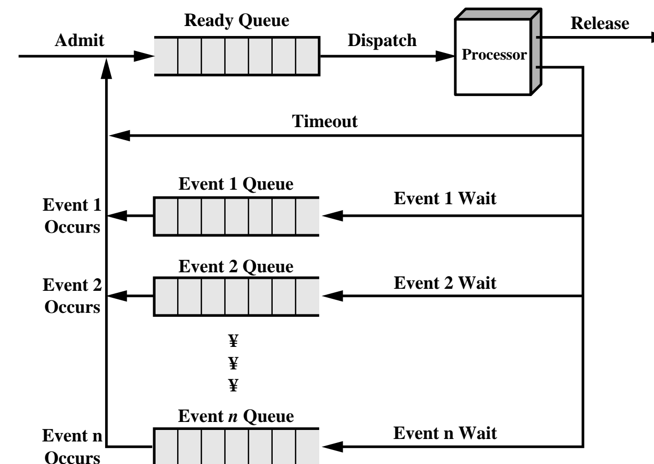
Separate queues



- Simplifies scheduler's job
- How about wakeup of blocked thread when event occurs?

Slide 16

Multiple wait queues:



Slide 17

Cooperative vs. Preemptive Multithreading

Cooperative multithreading:

- ➜ Threads determine exact order of execution
- ➜ Use yield() to switch between threads
- ➜ Problems if thread doesn't yield (e.g., buggy)

Preemptive multitasking:

- ➜ OS preempts thread's execution after some time
- ➜ Only guaranteed to work if H/W provides timer interrupt
- ➜ Implies unpredictable execution sequence!
  - thread switch can happen between any two instructions
  - threads may require concurrency control

Slide 18

User-level Operations on Threads in OS/161

➜ Start a new thread in OS/161

```
thread_fork(const char *    name,
            void       *    data1,
            unsigned long   data2,
            void       (*   func)(void *, unsigned long),
            struct thread **ret);
```

➜ Terminate thread

- thread_exit()

➜ Yield CPU

- thread_yield()

➜ Synchronisation:

- thread_sleep(const void *addr)
- thread_wakeup(const void *addr)

Slide 19

Processes and Threads

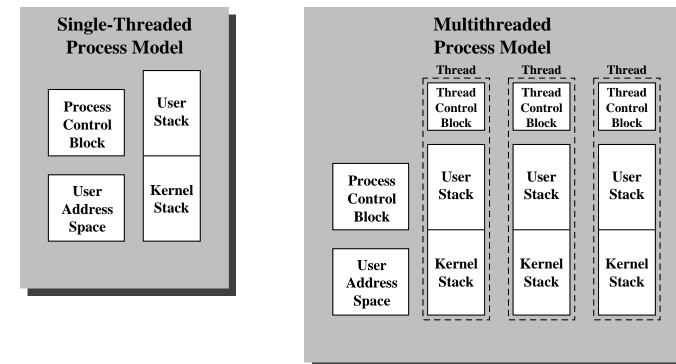
The OS stores information about Threads and Processes in Thread Control Block (TCB) and Process Controll Block (PCB)

- ➜ PCBs stored in process table
- ➜ TCBs stored in thread table

	Process	Thread
Address Space	✔	
Registers		✔
Program Counter		✔
Stack		✔
Open Files	✔	
State		✔
Signals and Handlers	✔	
Accounting Info	✔	✔
Global Variables		

Slide 20

Thread Switch



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

Slide 21

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 22

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!

Slide 23

Context Switch

- 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

Slide 24

Thread Switch

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)

Slide 25

THREAD SWITCH IN OS/161

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

Slide 26

PROCESSOR STATE EXAMPLE: MIPS R3000

- ➜ 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 27

MIPS-32 general purpose registers:

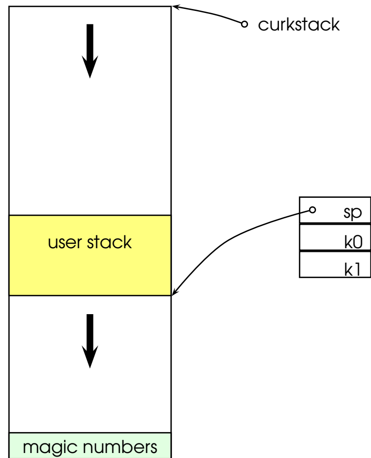
register	menmonic	convention
`r0`	`zero`	always zero
`r1`	`AT`	assembler temporary
`r2–r3`	`v0–v1`	integer function results
`r4–r7`	`a0–a3`	first four integer function args
`r8–r15`	`t0–t7`	temporary (not preserved)
`r16–r23`	`s0–s7`	preserved across calls
`r24–r25`	`t8–t9`	temporary (not preserved)
`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 28

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 29



Slide 30

MIPS Timer Interrupt

- MIPS processor does the following:

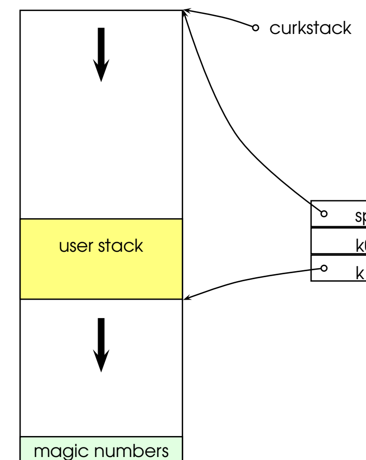
CP0.EPC	←	PC	
CP0.CAUSE.ExcCode	←	0	; interrupt
CP0.CAUSE.IP	←	0x01	; clock interrupt
PC	←	0x8000 0080	; gen excpt handler
CP0.STATUS.EXL	←	1	

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

Slide 31

```
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 32



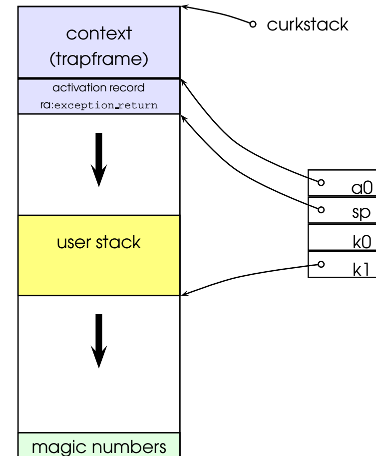
Slide 33

```
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 34

```
  /* 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 35



Slide 36

```
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 37

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

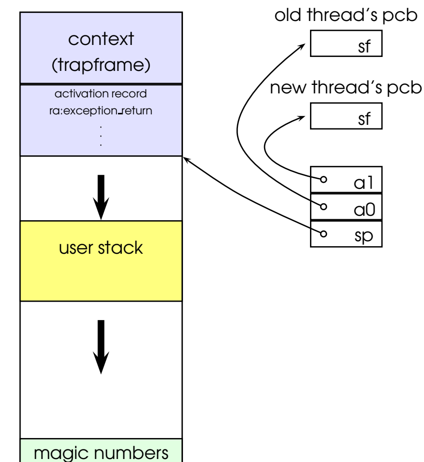
Slide 38

```
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 39

```
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;
};
```

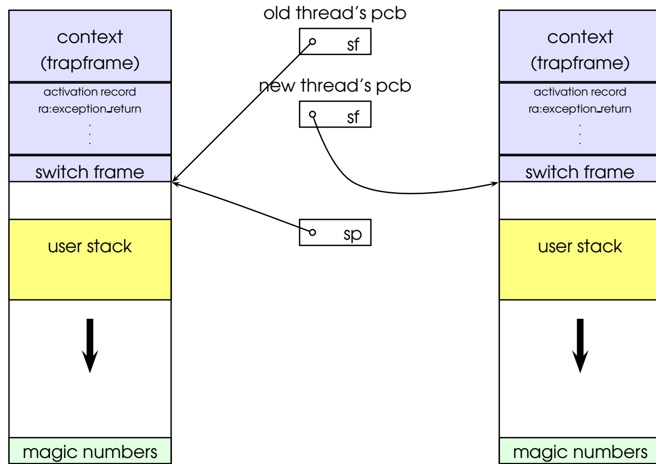
Slide 40



Slide 41

```
/*
 * 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)
```

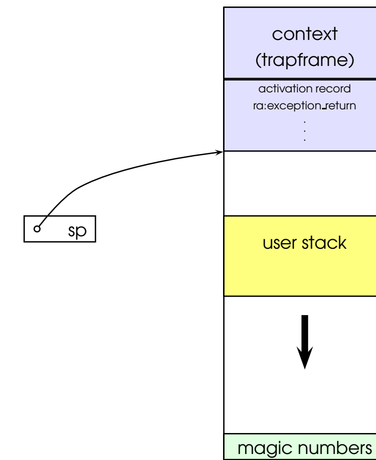
Slide 42



Slide 43

```
/* 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 44



Slide 45

```
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 46