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 users working interactively
  - for convenience (e.g., compile program while editing other file)

- Allocate resources required for execution of programs
- Support communication between executing programs

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

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

⇒ distinction between thread and process

Example: Web Server
**Slide 5**

**Example: Web Server**

- Dispatcher thread
- Worker thread
- Web page cache
- Network connection
- User space
- Kernel space
- Web server process

**Slide 6**

**Single-Threaded Web Server Implementations**

- 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

**Slide 7**

**Advantages of Threads**

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. Performance gains on machines with multiple CPU’s

**Slide 8**

**Threads and Processes**

- One process
  - one thread
- Multiple processes
  - multiple threads

= Instruction trace
**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)!

---

**Logical traces of threads:**

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

---

**Thread States**

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

- **Running:** currently active, using CPU
- **Ready:** runnable, waiting to be scheduled
- **Blocked:** waiting for an event to occur (I/O, alarm)

---

**Slide 11**

5000: Starting address of code for Thread A
8000: Starting address of code for Thread B
12000: Starting address of code for Thread C

Thread A  Thread B  Thread C

---

**Slide 12**

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

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

SEPARATE QUEUES

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

MULTIPLE WAIT QUEUES:

- Achieved by distinguishing more thread states
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

User-level Operations on Threads in OS/161

- Start a new thread in OS/161
  ```c
  thread_fork(const char * name, 
              void * data1, 
              unsigned long data2, 
              void (* func)(void *, unsigned long), 
              struct thread ** ret);
  ```

- Terminate thread
  ```c
  thread_exit();
  ```
- Yield CPU
  ```c
  thread_yield();
  ```
- Synchronisation:
  ```c
  thread_sleep(const void * addr)
  thread_wakeup(const void * addr)
  ```

Processes and Threads

The OS stores information about Threads and Processes in Thread Control Block (TCB) and Process Control Block (PCB)
- PCBs stored in process table
- TCBs stored in thread table

<table>
<thead>
<tr>
<th></th>
<th>Process</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address Space</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Registers</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Program Counter</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Stack</td>
<td>✔</td>
<td></td>
</tr>
<tr>
<td>Open Files</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>State</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Signals and Handlers</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Accounting Info</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>Global Variables</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Thread Switch

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

---

**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)

**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

**Thread Switch in OS/161**

- 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!
**Thread Switch in OS/161**

What happens 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**

- 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

---

**MIPS-32 General Purpose Registers:**

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

---

**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
  - Source? 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)
MIPS Timer Interrupt

- MIPS processor does the following:
  - CP0.EPC ← PC
  - CP0.CAUSE.ExcCode ← 0 ; Interrupt
  - CP0.CAUSE.IP ← 0x01 ; clock interrupt
  - PC ← 0x80000080 ; gen exception handler
  - CP0.STATUS.EXL ← 1

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

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 */
lk 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
```
common_exception:

```assembly
class 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 ra, 36(sp)

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

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

exception_return:

    /* 16(sp) */
    lw t0, 20(sp)    /* load status register value into t0 */
.....
```

```

MIPS Timer Interrupt

17

MIPS Timer Interrupt

18

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

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);

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;
};
*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)

/* 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. */
jr ra
addi sp, sp, 44    /* in delay slot */
.end mips_switch
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 */