Processes and Threads

- Processes:
  - Also called a task or job
  - Execution of an individual program
  - "Owner" of resources allocated for program execution
  - Encompasses one or more threads

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

Logical Execution Trace

<table>
<thead>
<tr>
<th>Process</th>
<th>Instruction</th>
<th>Time</th>
<th>Instruction</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5000</td>
<td>12000</td>
<td>5001</td>
<td>12001</td>
</tr>
<tr>
<td>B</td>
<td>5002</td>
<td>12002</td>
<td>5003</td>
<td>12003</td>
</tr>
<tr>
<td>C</td>
<td>5004</td>
<td>12004</td>
<td>5005</td>
<td>12005</td>
</tr>
<tr>
<td></td>
<td>5006</td>
<td>12006</td>
<td>5007</td>
<td>12007</td>
</tr>
<tr>
<td></td>
<td>5008</td>
<td>12008</td>
<td>5009</td>
<td>12009</td>
</tr>
<tr>
<td></td>
<td>5010</td>
<td>12010</td>
<td>5011</td>
<td>12011</td>
</tr>
</tbody>
</table>

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

Figure 3.2 Traces of Processes of Figure 3.1
Summary: The Process Model

- Multiprogramming of four programs
- Conceptual model of 4 independent, sequential processes (with a single thread each)
- Only one program active at any instant

Process and thread models of selected OSes

- Single process, single thread
  - MSDOS
- Single process, multiple threads
  - OS/161 as distributed
- Multiple processes, single thread
  - Traditional unix
- Multiple processes, multiple threads
  - Modern Unix (Linux, Solaris), Windows 2000

Note: Literature (incl. Textbooks) often do not cleanly distinguish between processes and threads (for historical reasons)

Process Termination

Conditions which terminate processes
1. Normal exit (voluntary)
2. Error exit (voluntary)
3. Fatal error (involuntary)
4. Killed by another process (involuntary)

Process Creation

Principal events that cause process creation
1. System initialization
   - Foreground processes (interactive programs)
   - Background processes
     - Email server, web server, print server, etc.
     - Called a daemon (unix) or service (Windows)
2. Execution of a process creation system call by a running process
   - New login shell for an incoming telnet connection
3. User request to create a new process
4. Initiation of a batch job

Note: Technically, all these cases use the same system mechanism to create new processes.

Process/Thread States

- Possible process/thread states:
  - running
  - blocked
  - ready

- Transitions between states shown
Some Transition Causing Events

Running > Ready
- Voluntary Yield()
- End of timeslice

Running > Blocked
- Waiting for input
  - File, network,
- Waiting for a timer (alarm signal)
- Waiting for a resource to become available

The Ready Queue

(b) Queuing diagram

The Ready Queue

Using Two Queues

What about blocked processes?

• When an unblocking event occurs, we also wish to avoid scanning all processes to select one to make Ready

Dispatcher

• Sometimes also called the scheduler
  - The literature is also a little inconsistent on this point
• Has to choose a Ready process to run
  - How?
  - It is inefficient to search through all processes
Implementation of Processes

- A processes’ information is stored in a **process control block** (PCB)
- The PCBs form a **process table**
  - Sometimes the kernel stack for each process is in the PCB
  - Sometimes some process info is on the kernel stack
  - E.g., registers in the trapframe in OS/161

Example fields of a process table entry

- Items shared by all threads in a process
- Items private to each thread

The Thread Model

- Local variables are per thread
  - Allocated on the stack
- Global variables are shared between all threads
  - Allocated in data section
  - Concurrency control is an issue
- Dynamically allocated memory (malloc) can be global or local
  - Program defined (the pointer can be global or local)
Thread Usage

A word processor with three threads

• Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread

Thread Usage

A multithreaded Web server

Thread Usage

Three ways to construct a server

Thread Usage

Implementing Threads in User Space

A user-level threads package

Summarising “Why Threads?”

• Simpler to program than a state machine
• Less resources are associated with them than a complete process
  – Cheaper to create and destroy
  – Shares resources (especially memory) between them
• Performance: Threads waiting for I/O can be overlapped with computing threads
  – Note if all threads are compute bound, then there is no performance improvement (on a uniprocessor)
• Threads can take advantage of the parallelism available on machines with more than one CPU (multiprocessor)
User-level Threads

- Implementation at user-level
  - User-level Thread Control Block (TCB), ready queue, blocked queue, and dispatcher
  - Kernel has no knowledge of the threads (it only sees a single process)
  - If a thread blocks waiting for a resource held by another thread, its state is save and the dispatcher switches to another ready thread
  - Thread management (create, exit, yield, wait) are implemented in a runtime support library

- Cons
  - Threads have to yield() manually (no timer interrupt delivery to user-level)
    - Co-operative multithreading
      - A single poorly designed/maintained thread can monopolise the available CPU time
    - There are work-arounds (e.g. a timer signal per second to enable pre-emptive multithreading), they are coarse grain and a kludge.
    - Does not take advantage of multiple CPUs (in reality, we still have a single threaded process as far as the kernel is concerned)

User-Level Threads

- Pros
  - Thread management and switching at user level is much faster than doing it in kernel level
  - No need to trap into kernel and back to switch
  - Dispatcher algorithm can be tuned to the application
    - E.g. use priorities
  - Can be implemented on any OS (thread or non-thread aware)
  - Can easily support massive numbers of threads on a per-application basis
    - Use normal application virtual memory
    - Kernel memory more constrained. Difficult to efficiently support wildly differing numbers of threads for different applications.

Implementing Threads in the Kernel

- Threads are implemented in the kernel
  - TCBs are stored in the kernel
    - A subset of information in a traditional PCB
      - The subset related to execution context
    - TCBs have a PCB associated with them
      - Resources associated with the group of threads (the process)
  - Thread management calls are implemented as system calls
    - E.g. create, wait, exit

Kernel Threads

- Threads are implemented in the kernel
  - TCBs are stored in the kernel
    - A subset of information in a traditional PCB
      - The subset related to execution context
    - TCBs have a PCB associated with them
      - Resources associated with the group of threads (the process)
  - Thread management calls are implemented as system calls
    - E.g. create, wait, exit
Kernel Threads

- **Cons**
  - Thread creation and destruction, and blocking and unblocking threads requires kernel entry and exit.
  - More expensive than user-level equivalent

- **Pros**
  - Preemptive multithreading
  - Parallelism
    - Can overlap blocking I/O with computation
    - Can take advantage of a multiprocessor

Multiprogramming Implementation

Skeleton of what lowest level of OS does when an interrupt occurs – a thread/context switch

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)
  - 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*.

Example Context Switch

- Running in user mode, SP points to user-level activation stack

![Simplified Explicit Thread Switch](image)
Example Context Switch

• Take an exception, syscall, or interrupt, and we switch to the kernel stack

Kernel SP

Example Context Switch

• We push a trapframe on the stack
  – Also called exception frame, user-level context....
  – Includes the user-level PC and SP

Kernel SP

Example Context Switch

• Call 'C' code to process syscall, exception, or interrupt
  – Results in a 'C' activation stack building up

Kernel SP

Example Context Switch

• The kernel decides to perform a context switch
  – It chooses a target thread (or process)
  – It pushes remaining kernel context onto the stack

Kernel SP

Example Context Switch

• Any other existing thread must
  – be in kernel mode (on a uni processor),
  – and have a similar stack layout to the stack we are currently using

Kernel SP

• We save the current SP in the PCB (or TCB), and load the SP of the target thread.
  – Thus we have switched contexts

Kernel SP
**Example Context Switch**

- Load the target thread’s previous context, and return to C

  ![Diagram](image)

- The C continues and (in this example) returns to user mode.

  ![Diagram](image)

- The user-level context is restored

  ![Diagram](image)

- The user-level SP is restored

  ![Diagram](image)

**The Interesting Part of a Thread Switch**

- What does the “push kernel state” part do???

  ![Diagram](image)

**OS/161 md_switch**

```c
md_switch(struct pcb *old, struct pcb *nu) {
    if (old==nu) {
        return;
    }
    /*
    * Note: we don’t need to switch curspl, because splhigh() should always be in effect when we get here and when we leave here.
    */
    old->pcb_kstack = curkstack;
    old->pcb_ininterrupt = in_interrupt;
    curkstack = nu->pcb_kstack;
    in_interrupt = nu->pcb_ininterrupt;
    mips_switch(old, nu);
}
```
mips_switch:
/*
 * a0 contains a pointer to the old thread's struct pcb.
 * a1 contains a pointer to the new thread's struct pcb.
 * The only thing we touch in the pcb is the first word, which
 * we save the stack pointer in. The other registers get saved
 * on the stack, namely:
 *      s0-s8
 *      gp, ra
 * The order must match arch/mips/include/switchframe.h.
 */
/* 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 pcb */
sw sp, 0(a0)

/* Restore the registers */
lw s0, 0(sp)  
/ delay slot for load */
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 gp, 36(sp)
lw ra, 40(sp)
/* delay slot for load */
lw sp, 0(sp)
/ delay slot for load */
addi sp, sp, 44 /* in delay slot */

j ra

/* Revisiting Thread Switch */

Thread a

mips_switch(a,b) -------> a
|

mips_switch(b,a) -------> b
|

Thread b

Save the registers that the 'C' procedure calling convention expects preserved