Processes and Threads
Learning Outcomes

• An understanding of fundamental concepts of processes and threads

• An understanding of the typical implementation strategies of processes and threads
  – Including an appreciation of the trade-offs between the implementation approaches
    • Kernel-threads versus user-level threads

• A detailed understanding of “context switching”
Major Requirements of an Operating System

• Interleave the execution of several processes to maximize processor utilization while providing reasonable response time
• Allocate resources to processes
• Support interprocess communication and user creation of processes
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
Execution snapshot of three single-threaded processes (No Virtual Memory)
Logical Execution Trace

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

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.2 Traces of Processes of Figure 3.1
### Combined Traces

(Actual CPU Instructions)

**What are the shaded sections?**

```plaintext
<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5000</td>
<td>5001</td>
<td>5002</td>
<td>5003</td>
<td>5004</td>
<td>5005</td>
<td>5006</td>
<td>5007</td>
</tr>
<tr>
<td>7</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>13</td>
<td>8000</td>
<td>8001</td>
<td>8002</td>
<td>8003</td>
<td>8004</td>
<td>8005</td>
<td>8006</td>
<td>8007</td>
</tr>
<tr>
<td>17</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>23</td>
<td>12000</td>
<td>12001</td>
<td>12002</td>
<td>12003</td>
<td>12004</td>
<td>12005</td>
<td>12006</td>
<td>12007</td>
</tr>
<tr>
<td>29</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>35</td>
<td>5006</td>
<td>5007</td>
<td>5008</td>
<td>5009</td>
<td>5010</td>
<td>5011</td>
<td>5012</td>
<td>5013</td>
</tr>
<tr>
<td>41</td>
<td>100</td>
<td>101</td>
<td>102</td>
<td>103</td>
<td>104</td>
<td>105</td>
<td>106</td>
<td>107</td>
</tr>
<tr>
<td>47</td>
<td>12006</td>
<td>12007</td>
<td>12008</td>
<td>12009</td>
<td>12010</td>
<td>12011</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

100 = Starting address of dispatcher program

shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

---

**Figure 3.3 Combined Trace 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
Figure 4.1  Threads and Processes [ANDE97]
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 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/ssh 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 Termination

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

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

- Transitions between states shown

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

Running $\rightarrow$ Ready
- Voluntary \texttt{Yield()}
- End of timeslice

Running $\rightarrow$ Blocked
- Waiting for input
  - File, network,
- Waiting for a timer (alarm signal)
- Waiting for a resource to become available
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
The Ready Queue

(b) Queuing diagram
What about blocked processes?

• When an *unblocking* event occurs, we also wish to avoid scanning all processes to select one to make *Ready*
Using Two Queues
(b) Multiple blocked queues
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
  – Reality is much more complex (hashing, chaining, allocation bitmaps, …)
# Implementation of Processes

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Example fields of a process table entry**
Threads Analogy

The Hamburger Restaurant
Single-Threaded Restaurant

Customer Arrives → Take Order → Start Fries → Fries Cook

Wait for Customer → Fries Finish

Serve Customer

Assemble Order → Burger Finished

Burger Cooks → Start Burger
Multithreaded Restaurant

Customer Arrives

Wait for Customer

Take Order

Assemble Order

Serve Customer

Start Fries

Fries Finish

Fries Cook

Start Burger

Burger Finished

Burger Cooks
Finite-State Machine Model
(Event-based model)

Input Events:
- Customer Arrives
- Fries Finish
- Burger Finished

Non-Blocking actions:
- Start Fries
- Serve Customer
- Start Burger

External activities:
- Fries Cook
- Burger Cooks
- Wait for Customer
Threads
The Thread Model

(a) Three processes each with one thread
(b) One process with three threads
# The Thread Model

<table>
<thead>
<tr>
<th>Per process items</th>
<th>Per thread items</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address space</td>
<td>Program counter</td>
</tr>
<tr>
<td>Global variables</td>
<td>Registers</td>
</tr>
<tr>
<td>Open files</td>
<td>Stack</td>
</tr>
<tr>
<td>Child processes</td>
<td>State</td>
</tr>
<tr>
<td>Pending alarms</td>
<td></td>
</tr>
<tr>
<td>Signals and signal handlers</td>
<td></td>
</tr>
<tr>
<td>Accounting information</td>
<td></td>
</tr>
</tbody>
</table>

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

Each thread has its own stack
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
Thread Usage

A multithreaded Web server
Thread Usage

while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

(a)

while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}

(b)

- Rough outline of code for previous slide
  (a) Dispatcher thread
  (b) Worker thread
# Thread Usage

<table>
<thead>
<tr>
<th>Model</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads</td>
<td>Parallelism, blocking system calls</td>
</tr>
<tr>
<td>Single-threaded process</td>
<td>No parallelism, blocking system calls</td>
</tr>
<tr>
<td>Finite-state machine</td>
<td>Parallelism, nonblocking system calls, interrupts</td>
</tr>
</tbody>
</table>

### Three ways to construct a server
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)
Implementing Threads in User Space

A user-level threads package
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 saved and the dispatcher switches to another ready thread
  – Thread management (create, exit, yield, wait) are implemented in a runtime support library
User-Level Threads

• Pros
  – Thread management and switching at user level is much faster than doing it in kernel level
    • No need to trap (take syscall exception) 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.
User-level Threads

• Cons
  – Threads have to yield() manually (no timer interrupt delivery to user-level)
    • Co-operative multithreading
      – A single poorly design/implemented 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 course 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

• Cons
  – If a thread makes a blocking system call (or takes a page fault), the process (and all the internal threads) blocks
    • Can’t overlap I/O with computation
    • Can use wrappers as a work around
      – Example: wrap the `read()` call
      – Use `select()` to test if read system call would block
        » `select()` then `read()`
        » Only call `read()` if it won’t block
        » Otherwise schedule another thread
      – Wrapper requires 2 system calls instead of one
        » Wrappers are needed for environments doing lots of blocking system calls?
    • Can change to kernel to support non-blocking system call
      – Lose “on any system” advantage, page faults still a problem.
Implementing Threads in the Kernel

A threads package managed by the kernel
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
User-level Threads

User Mode

- Scheduler
- Process A

Kernel Mode

- Scheduler
- Process B
- Process C

Scheduler
User-level Threads

✔ Fast thread management (creation, deletion, switching, synchronisation…)

✗ Blocking blocks all threads in a process
  – Syscalls
  – Page faults

✗ No thread-level parallelism on multiprocessor
Kernel-Level Threads

User Mode

Kernel Mode

Scheduler

Process A

Process B

Process C
Kernel-level Threads

❌ Slow thread management (creation, deletion, switching, synchronisation…)
  • System calls
✔ Blocking blocks only the appropriate thread in a process
✔ Thread-level parallelism on multiprocessor
Multiprogramming Implementation

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

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

• A switch between threads can happen 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*

A thread switch can happen between any two instructions

Note instructions do not equal program statements
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*. 
Simplified
Explicit
Thread Switch

Thread a

thread_switch(a,b) {

}

Thread b

thread_switch(b,a) {

}

thread_switch(a,b) {

}
Example Context Switch

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

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

- We push a *trapframe* on the stack
  - Also called *exception frame, user-level context*....
  - Includes the user-level PC and SP
Example Context Switch

- Call ‘C’ code to process syscall, exception, or interrupt
  - Results in a ‘C’ activation stack building up

```
  SP
  └── 'C' activation stack
      └── trapframe
```
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
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 stacks of other threads

<table>
<thead>
<tr>
<th>Kernel State</th>
<th>‘C’ activation stack</th>
<th>trapframe</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example Context Switch

- We save the current SP in the PCB (or TCB), and load the SP of the target thread.
  - Thus we have *switched contexts*
Example Context Switch

• Load the target thread’s previous context, and return to C
Example Context Switch

• The C continues and (in this example) returns to user mode.
Example Context Switch

- The user-level context is restored
Example Context Switch

• The user-level SP is restored
The Interesting Part of a Thread Switch

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

<table>
<thead>
<tr>
<th>SP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel State</td>
</tr>
<tr>
<td>Kernel State</td>
</tr>
</tbody>
</table>
OS/161 md_switch

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)
/* Get the new stack pointer from the new pcb */
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 */

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

```
mips_switch(a,b)
{
}
```

Thread b

```
mips_switch(b,a)
{
}
```

Revisiting Thread Switch