Processes and Threads
Implementation

Learning Outcomes

• 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”

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

Processes

• User-mode
  • Processes (programs) scheduled by the kernel
  • Isolated from each other
  • No concurrency issues between each other
• System-calls transition into and return from the kernel
• Kernel-mode
  • Nearly all activities still associated with a process
  • Kernel memory shared between all processes
  • Concurrency issues exist between processes concurrently executing in a system call

Threads

The Thread Model

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

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

Implementing Threads in User Space

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

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 designed/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 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

- **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 – exactly when efficiency matters!

Implementing Threads in the Kernel

- A threads package managed by the kernel

Kernel-Level Threads

- **Kernel-Level 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
**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**

1. Hardware stack 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. l: interrupt service runs (typically reads and outputs input)
6. Scheduler decides which process to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure sets 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.

**Kernel-Level Threads**

User Mode

Kernel Mode

Scheduler

Thread Switch...

Simplified Explicit Thread Switch

Thread a

Thread b
Example Context Switch
- Running in user mode, SP points to user-level stack (not shown on slide)

Representation of Kernel Stack (Memory)

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

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
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???
Simplified OS/161 thread_switch

```c
static
void
thread_switch(threadstate_t newstate, struct wchan *wc)
{
struct thread *cur, *next;

cur = curthread;
do {
    next = threadlist_remhead(&curcpu->c_runqueue);
    if (next == NULL) {
        cpu_idle();
    }
} while (next == NULL);
/* do the switch (in assembler in switch.S) */
switchframe_switch(&cur->t_context, &next->t_context);
}
```

Lots of code removed – only basics of pick next thread and run it remain.

OS/161 switchframe_switch

```asm
/* Allocate stack space for saving 10 registers. 10*4 = 40 */
addi sp, sp, -40
/* Save the registers */
sw   ra, 36(sp)
sw   gp, 32(sp)
sw   s8, 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 thread */
sw   sp, 0(a0)
/* Get the new stack pointer from the new thread */
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   s8, 28(sp)
lw   gp, 32(sp)
lw   ra, 36(sp)
/* delay slot for load */
```

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

OS/161 switchframe_switch

```asm
/* and return. */
jmp addi, sp, 40 /* In delay slot */
```

OS/161 switchframe_switch

```
/* and return. */
jmp addi, sp, 40 /* In delay slot */
```