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

<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>File permissions</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>User ID</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>Group ID</td>
</tr>
<tr>
<td>Stack pointer</td>
<td>Parent process</td>
<td>Working directory</td>
</tr>
<tr>
<td>Process state</td>
<td>Process group</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Priority</td>
<td>Signals</td>
<td>User ID</td>
</tr>
<tr>
<td>Precedence parameters</td>
<td>Time when process started</td>
<td>Group ID</td>
</tr>
<tr>
<td>Process ID</td>
<td>CPU time used</td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td>Child’s CPU time</td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td>Time of next alarm</td>
<td></td>
</tr>
</tbody>
</table>

Threads

The Thread Model

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

Each thread has its own stack

The Thread Model

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

- 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
A word processor with three threads

A multithreaded Web server

Thread Usage

Thread Usage

Thread Usage

Thread Usage

Three ways to construct a server

Rough outline of code for previous slide

(a) Dispatcher thread

(b) Worker thread

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 save 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 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 coarse grain and kludgey.
    - 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

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

Hybrid Schemes

Multiple user threads on a kernel thread

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 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**
Example Context Switch

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

Kernel Stack

(Memory)

Kernel SP

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

trapframe

Example Context Switch

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

Kernel 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

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 State

'C' activation stack

trapframe

Kernel stacks of other threads
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???
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);
}
```

OS/161 mips_switch

```c
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
```

Revisiting Thread Switch