Processes and Threads
Implementation

Learning Outcomes
• A basic understanding of the MIPS R3000 assembly
  and compiler generated code.
• 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”

MIPS R3000
• Load/store architecture
  • No instructions that operate on memory except load and
    store
  • Simple load/stores to/from memory from/to registers
    • Store word: sw r4, (r5)
    • Store contents of r4 in memory using address contained in register r5
    • Load word: lw r3, (r7)
    • Load contents of memory into r3 using address contained in r7
    • Delay of one instruction after load before data available in destination
      register
    • Must always an instruction between a load from memory and the
      subsequent use of the register.
  • lw, sw, lb, sb, lh, sh, ...
MIPS Registers

- User-mode accessible registers
- 32 general purpose registers
  - r0 hardwired to zero
  - r31 the link register for jump-and-link (JAL) instruction
- HI/LO
  - 2 * 32-bits for multiply and divide
- PC
  - Not directly visible
  - Modified implicitly by jump and branch instructions

Branching and Jumping

- Branching and jumping have a branch delay slot
- The instruction following a branch or jump is always executed prior to destination of jump

Compiler Register Conventions

- Given 32 registers, which registers are used for
  - Local variables?
  - Argument passing?
  - Function call results?
  - Stack Pointer?

MIPS R3000

- RISC architecture – 5 stage pipeline
- Instruction partially through pipeline prior to jmp having an effect

Jump and Link Instruction

- JAL is used to implement function calls
- r31 = PC+8
- Return Address register (RA) is used to return from function call

Compiler Register Conventions

<table>
<thead>
<tr>
<th>Reg No</th>
<th>Name</th>
<th>Used for</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>zero</td>
<td>Always 0</td>
</tr>
<tr>
<td>1</td>
<td>sp</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>2-3</td>
<td>(SP)</td>
<td>Save Frame Pointer</td>
</tr>
<tr>
<td>4-7</td>
<td>(ARG)</td>
<td>Argument registers for subroutine</td>
</tr>
<tr>
<td>8-15</td>
<td>(FPR)</td>
<td>Temp floating point registers</td>
</tr>
<tr>
<td>16-31</td>
<td>(GPR)</td>
<td>General purpose registers</td>
</tr>
<tr>
<td>32</td>
<td>(AR)</td>
<td>Address register</td>
</tr>
<tr>
<td>33</td>
<td>(LR)</td>
<td>Return Address</td>
</tr>
<tr>
<td>34-35</td>
<td>(ALU)</td>
<td>Arithm/Logic Unit</td>
</tr>
<tr>
<td>36-37</td>
<td>(M)</td>
<td>Memory Unit</td>
</tr>
</tbody>
</table>
Simple factorial

```c
int fact(int n)
{
    int r = 1;
    int i;
    for (i = 1; i < n+1; i++) {
        r = r * i;
    }
    return r;
}
```

Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().

Function Stack Frames

- Each function call allocates a new stack frame for local variables, the return address, previous frame pointer etc.
  - Frame pointer: start of current stack frame
  - Stack pointer: end of current stack frame
- Example: assume f1() calls f2(), which calls f3().

Stack Frame

- MIPS calling convention for gcc
  - Args 1-4 have space reserved for them

Example Code

```c
main ()
{
    int i;
    int sixargs(int a, int b, int c, int d, int e, int f)
    {
        i = sixargs(1,2,3,4,5,6);
        return a + b + c + d + e + f;
    }
}
The Process Model

- Multiprogramming of four programs

<table>
<thead>
<tr>
<th>Process A</th>
<th>Process B</th>
<th>Process C</th>
</tr>
</thead>
</table>

- Minimally consist of three segments
  - Text: contains the code (instructions)
  - Data: global variables
  - Stack: activation records of procedure/function/method
  - Local variables

- Note:
  - Data can dynamically grow up
    - E.g., malloc()-ing
  - The stack can dynamically grow down
    - E.g., increasing function call depth or recursion

- Process Memory Layout

```
Stack
↓
Gap
↓
Data
↓
Text
```

- Processes
  - User Mode
    - Process’s user-level stack and execution state
    
  - Kernel Mode
    - Scheduler
    - Process’s in-kernel stack and execution state
Processes

- User-mode
  - Processes (programs) scheduled by the kernel
  - Isolated from each other
  - No concurrency issues between each other
- 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

<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 that exist per thread

A Subset of POSIX threads API

```c
int pthread_create(pthread_t *, const pthread_attr_t *,
                    void *(*)(void *), void *);
void  pthread_exit(void *);
int pthread_mutex_init(pthread_mutex_t *,
                       const pthread_mutexattr_t *);
int pthread_mutex_destroy(pthread_mutex_t *);
int pthread_mutex_lock(pthread_mutex_t *);
int pthread_mutex_unlock(pthread_mutex_t *);
int pthread_rwlock_init(pthread_rwlock_t *,
                        const pthread_rwlockattr_t *);
int pthread_rwlock_destroy(pthread_rwlock_t *);
int pthread_rwlock_rdlock(pthread_rwlock_t *);
int pthread_rwlock_wrlock(pthread_rwlock_t *);
int pthread_rwlock_unlock(pthread_rwlock_t *);
```

Where to Implement Application

- Threads?

Note: Thread API similar in both cases

User Mode
- Kernel Mode

System Libraries
- User-level threads implemented in a library?

OS
- Kernel-provided threads implemented in the OS?
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 inside the same process, 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.

- Cons
  - Threads have to yield() manually (no timer interrupt delivery to user-level)
  - Co-operative multithreading
    - A single poorly designed/implemented thread can monopolize the available CPU time
    - There are work-arounds (e.g., a timer signal per second to enable preemptive 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
### Kernel-provided Threads

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

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

### Multiprogramming Implementation

1. Hardware stacks program counter, etc.
2. Hardware load 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 context switch
**Context Switch Terminology**

- A context switch can refer to
  - A switch between threads
    - Involving saving and restoring of state associated with a thread
  - A switch between processes
    - Involving the above, plus extra state associated with a process.
    - E.g. memory maps

**Context Switch Occurrence**

- A switch between process/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**

- Context switch must be *transparent* for processes/threads
  - When dispatched again, process/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 *process/thread context*
- Switching between process/threads consequently results in a *context switch*.

**Simplified Explicit Thread Switch**

\[
\text{thread.switch}(a, b) \rightarrow \text{thread.switch}(b, a)
\]

**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

![Diagram](image1.png)

**Example Context Switch**

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

![Diagram](image2.png)

**Example Context Switch**

- The user-level context is restored

![Diagram](image3.png)

**Example Context Switch**

- The user-level SP is restored

![Diagram](image4.png)

**The Interesting Part of a Thread Switch**

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

![Diagram](image5.png)

**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

switchframe_switch:
/*
 * a0 contains the address of the switchframe pointer in the old thread.
 * a1 contains the address of the switchframe pointer in the new thread.
 *
 * The switchframe pointer is really the stack pointer. The other
 * registers get saved on the stack, namely:
 *
 *    s0-s6, s8
 *    gp, ra
 *
 * The order must match <mips/switchframe.h>.
 *
 * Note that while we'll explicitly need to save 6 registers, because we
 * won't be able to return to the old thread in the middle of a function.
 * switchframe is managed by thread.c, so we won't just let thread.c
 * manage it.
 */

/* 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)

/* and return. */
j ra
addi sp, sp, 40 /* in delay slot */

OS/161 switchframe_switch

/* Get the new stack pointer from the new thread */
lo sp, a1
rep /* delay slot for load */
/* Now, restore the registers */
lo s0, 0(sp)
lo s1, 4(sp)
lo s2, 8(sp)
lo s3, 12(sp)
lo s4, 16(sp)
lo s5, 20(sp)
lo s6, 24(sp)
lo s8, 28(sp)
lo gp, 32(sp)
lo ra, 36(sp)
rep /* delay slot for load */

/* end return */
j lo
addi sp, sp, 40 /* in delay slot */

Revisiting Thread Switch