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

<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 *);
```

Memory

Where to Implement Application Threads?

Note: Thread API similar in both cases

Implementing Threads in User Space

User-level Threads
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.

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 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)

Kernel-Level Threads

User Mode

Scheduler

Kernel Mode

Process A

Process B

Process C

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

• 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 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

Assume Kernel-Level Threads

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

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

SP

Kernel State
C activation stack
trapframe

SP

Kernel State
C activation stack
trapframe

The Interesting Part of a Thread Switch

- What does the "push kernel state" part do???

SP

Kernel State
C activation stack
trapframe

Kernel State
C activation stack
trapframe

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();
            return;
        }
    } 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

```c
/* Allocate stack space for saving 10 registers. 10*4 = 40 */
addi sp, sp, -40
/* Save the registers */
sw rs, 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)
/* Save the old stack pointer in the old thread */
sw sp, 0(a0)
```

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

```c
/* Get the new stack pointer from the new thread */
w sp, 0(a1)
```

OS/161 switchframe_switch

```c
lw sp, 0(a1)
```

```c
lw rs, 36(sp)
```

```c
lw gp, 32(sp)
```

```c
lw s8, 28(sp)
```

```c
lw s6, 24(sp)
```

```c
lw s5, 20(sp)
```

```c
lw s4, 16(sp)
```

```c
lw s3, 12(sp)
```

```c
lw s2, 8(sp)
```

```c
lw s1, 4(sp)
```

```c
lw s0, 0(sp)
```

```
```

```c
lw sp, 0(a0)
```

```
```
Revisiting Thread Switch

```
/* and return */
jr ra
addi sp, sp, 40 /* in delay slot */
```

```
void switchframe_switch(a, b) {
    switchframe_switch(b, a) {
    }
    switchframe_switch(a, b) {
    }
}
```

Thread a

Thread b