Threads

CSCI 444/544 Operating Systems
Fall 2008

Agenda

Thread concept
Thread vs process
Thread implementation
- user-level
- kernel-level
- hybrid
Inter-process (inter-thread) communication

What is Thread

A thread is a fundamental unit of CPU utilization
- a thread ID
- a program counter
- a register set
- a stack

It shares with other threads belonging to the same process
- code section and data section
- other OS resources, such as open files

Single and Multi-threaded Processes
### Process vs. Thread (1)

<table>
<thead>
<tr>
<th>Process</th>
<th>Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit of allocation</td>
<td>unit of execution</td>
</tr>
<tr>
<td>defines the address space and consumed OS resources</td>
<td>defines a sequential execution stream within a process</td>
</tr>
</tbody>
</table>

- Each process has one or more threads
- Each thread belongs to one process

### Process vs Thread (2)

- A process is just a container in which threads execute
- Sharing data between the threads of a process is cheap: in the same address space
  - one thread may corrupt another thread (no memory protection)
  - one process cannot corrupt another process
- Creating threads is cheap too

### Why Threads Help

- Multi-threading supports concurrency/parallelism
  - concurrent events (e.g., Web servers)
  - parallel program in multi-processor architecture
- But why not multiple processes?
  - memory sharing
  - efficient synchronization between threads
  - less context switch overhead

### Common Multithread Models

- Manager/worker
  - Single manager handles input and assigns work to the worker threads
- Producer/consumer
  - Multiple producer threads create data (or work) that is handled by one of the multiple consumer threads
- Pipeline
  - Task is divided into series of sub-tasks, each of which is handled in series by a different thread
Thread Control Block

- Each thread is described by a thread control block (TCB)
- A TCB usually includes
  - Thread ID
  - Space for saving registers
  - Information about its stack
  - A pointer so it can be chained into a linked list

OS Support for Threads

There are three ways for thread support

- User-level threads
  - Thread data structure is in user-mode memory
  - Scheduling/switching done at user mode
- Kernel-level threads
  - Thread data structure is in kernel memory
  - Scheduling/switching done by the OS kernel
- Hybrid of user-level and kernel-level threads
  - M user threads run on N kernel threads (M>N>1)

User-level Threads

Many-to-one thread mapping

Implemented as user-level runtime library (thread library)
- thread creation, termination, scheduling and switching at user-level

Kernel sees one execution context per process and is unaware of thread activity

Many-to-One

Examples: Solaris Green Threads and GUN Portable Threads
Pros vs Cons of User-level Threads

- **Advantages**
  - Performance: low-cost thread operation since no system calls
  - Flexibility: scheduling policy can be application-specific
  - Portability: does not require OS support, easy to port

- **Disadvantages**
  - Cannot leverage multiprocessors
    - Kernel assigns one process to only one processor
  - Entire process blocks if one thread blocks

---

Pros vs Cons of Kernel-level Thread

Advantages
- each kernel-level thread can in parallel in a multiprocessor
- when one thread blocks, other threads from process can be scheduled

Disadvantages
- Higher overhead for thread operations
- OS must scale well with increasing number of threads

---

Kernel-level Threads

One-to-one mapping:
- map each thread to its own execution context in the kernel

- OS manages threads and sees multiple execution contexts per process

- Each kernel thread scheduled independently

- Thread operations performed by OS
  - Thread operations are all system calls
  - Must maintain kernel state for each thread

---

Hybrid Threads

Hybrid of kernel and user-level threads:
- m-to-n thread mapping (m>n)
  - Application creates m threads
  - OS provides pool of n kernel threads
  - A few user-level threads mapped to each kernel-level thread

- Both kernel and user-level thread library involved in thread management
Many-to-Many Mapping

Examples: Solaris prior to version 9

A Variation (Two-level Model)

Thread/Process Operation Latencies

<table>
<thead>
<tr>
<th>Operation</th>
<th>User-level Thread (ms)</th>
<th>Kernel Threads (ms)</th>
<th>Processes (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null fork</td>
<td>34</td>
<td>948</td>
<td>11,300</td>
</tr>
<tr>
<td>Signal-wait</td>
<td>37</td>
<td>441</td>
<td>1,840</td>
</tr>
</tbody>
</table>


<table>
<thead>
<tr>
<th>Operation</th>
<th>Kernel Threads (ms)</th>
<th>Processes (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null fork</td>
<td>45</td>
<td>108</td>
</tr>
</tbody>
</table>


Interprocess Communication (IPC)

- To cooperate usefully, processes (threads) must communicate with each other
- How do processes and threads communicate?
  - Shared memory
  - Message passing
  - Signals
IPC: Shared Memory

- Process
  - Each process has private address space
  - Explicitly set up shared memory segment within each address space
- Threads
  - Always share address space
- Advantages
  - Fast and easy to share data
- Disadvantages
  - Must synchronize data access; error prone

IPC: Message Passing (1)

- Message passing most commonly used between processes
  - Explicitly pass data between sender (src) and receiver (destination)
  - Example: Unix pipes
- Advantages:
  - Makes sharing explicit
  - Improves modularity (narrow interface)
  - Does not require trust between sender and receiver
- Disadvantages:
  - Performance overhead to copy messages

IPC: Message Passing (2)

- Issues:
  - How to name source and destination?
    - One process, set of processes, or mailbox (port)
  - Does sending process wait (i.e., block) for receiver?
    - Blocking: Slows down sender
    - Non-blocking: Requires buffering between sender and receiver

IPC: Signal

- Signal
  - Like software interrupt that notifies a process of an event
    - By one process to another process
    - By the kernel to a process
- What happens when a signal is received?
  - Catch: Specify signal handler to be called
  - Ignore: Rely on OS default action
    - Example: Abort, memory dump, suspend or resume process
  - Mask: Block signal so it is not delivered
    - May be temporary (while handling signal of same type)
- Disadvantage
  - Does not specify any data to be exchanged
  - Complex semantics with threads
Thread Libraries

- **Pthreads**
  - A POSIX standard API for thread creation and synchronization
  - May be provided as either user- or kernel-level library

- **Win32 Threads**
  - Kernel-level library available on Windows

- **Java threads**
  - Managed by the JVM

Thread interface

- This is taken from the POSIX `pthreads` API:
  - `t = pthread_create(attributes, start_procedure)`
    - creates a new thread of control
    - new thread begins executing at `start_procedure`
  - `pthread_cond_wait(condition_variable)`
    - the calling thread blocks, sometimes called `thread_block()`
  - `pthread_signal(condition_variable)`
    - starts the thread waiting on the condition variable
  - `pthread_exit()`
    - terminates the calling thread
  - `pthread_wait(t)`
    - waits for the named thread to terminate

Thread Switch (1)

- Everyone to cooperate
  - A thread willingly gives up the CPU by calling `yield()`
  - `yield()` calls into the scheduler, which context switches to another ready thread
  - what happens if a thread never calls `yield()`?

Thread Switch (2)

- Use preemption
  - scheduler requests that a timer interrupt be delivered by the OS periodically
    - usually delivered as a signal
  - at each timer interrupt, scheduler gains control and context switches as appropriate
Thread Context Switch

– save context of currently running thread
  • push machine state onto thread stack
– restore context of the next thread
  • pop machine state from next thread’s stack
– return as the new thread
  • execution resumes at PC of next thread