Threads
Implementation

Jin-Soo Kim (jinsookim@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Today’s Topics

- How to implement threads?
  - User-level threads
  - Kernel-level threads

- Threading models
Kernel/User-level Threads

- **Who is responsible for creating/managing threads?**
  - The OS (kernel threads)
    - Thread creation and management requires system calls
  - The user-level process (user-level threads)
    - A library linked into the program manages the threads

- **Why is user-level thread management possible?**
  - Threads share the same address space
    - The thread manager doesn’t need to manipulate address spaces
  - Threads only differ in hardware contexts (roughly)
    - PC, SP, registers
    - These can be manipulated by the user-level process itself.
Kernel-level Threads (1)

- OS-managed threads
  - The OS manages threads and processes.
  - All thread operations are implemented in the kernel.
  - The OS schedules all of the threads in a system.
    - If one thread in a process blocks (e.g., on I/O), the OS knows about it, and can run other threads from that process.
    - Possible to overlap I/O and computation inside a process.
  - Kernel threads are cheaper than processes.
    - Less state to allocate and initialize
  - Windows 98/NT/2000/XP/Vista/7
  - Solaris
  - Tru64 Unix
  - Linux
  - Mac OS X
Kernel-level Threads (2)

**Limitations**

- They can still be too expensive.
  - For fine-grained concurrency, we need even cheaper threads.
  - Ideally, we want thread operations as fast as a procedure call.
- Thread operations are all system calls.
  - The program must cross an extra protection boundary on every thread operation, even when the processor is being switched between threads in the same address space.
  - The OS must perform all of the usual argument checks.
- Must maintain kernel state for each thread.
  - Can place limit on the number of simultaneous threads. (typically ~1000)
- Kernel-level threads have to be general to support the needs of all programmers, languages, runtime systems, etc.
Implementing Kernel-level Threads

- **Kernel-level threads**
  - Kernel-level threads are similar to original process management and implementation.
User-level Threads (1)

- Motivation
  - To make threads cheap and fast, they need to be implemented at the user level.
  - Portable: User-level threads are managed entirely by the runtime system (user-level library).

- User-level threads are small and fast
  - Each thread is represented simply by a PC, registers, a stack, and a small thread control block (TCB).
  - Creating a thread, switching between threads, and synchronizing threads are done via procedure calls (No kernel involvement).
  - User-level thread operations can be 10-100x faster than kernel-level threads.
Limitations

- User-level threads are invisible to the OS.
  - They are not well integrated with the OS
- As a result, the OS can make poor decisions.
  - Scheduling a process with only idle threads
  - Blocking a process whose thread initiated I/O, even though the process has other threads that are ready to run.
  - Unscheduling a process with a thread holding a lock.
- Solving this requires coordination between the kernel and the user-level thread manager.
  - e.g., all blocking system calls should be emulated in the library via non-blocking calls to the kernel.
Implementing User-level Threads (1)

- User-level threads
Implementing User-level Threads (2)

- **Thread context switch**
  - Very simple for user-level threads
    - Save context of currently running thread
      - push all machine state onto its stack
    - restore context of the next thread
      - pop machine state from next thread's stack
    - the next thread becomes the current thread
    - return to caller as the new thread
      - execution resumes at PC of next thread
  - All done by assembly languages
    - It works at the level of the procedure calling convention, so it cannot be implemented using procedure calls.
Implementing User-level Threads (3)

- **Thread scheduling**
  - A thread scheduler determines when a thread runs.
    - Just like the OS and processes
    - But implemented at user-level as a library
  - It uses queues to keep track of what threads are doing.
    - Run queue: threads currently running
    - Ready queue: threads ready to run
    - Wait queue: threads blocked for some reason (maybe blocked on I/O or a lock)
  - How can we prevent a thread from hogging the CPU?
Implementing User-level Threads (4)

- Non-preemptive scheduling
  - Force everybody to cooperate
    - Threads willingly give up the CPU by calling `yield()`.  
  - `yield()` calls into the scheduler, which context switches to another ready thread.

```c
Thread ping ()
{
    while (1) {
        printf (“ping\n”);
        yield();
    }
}

Thread pong ()
{
    while (1) {
        printf (“pong\n”);
        yield();
    }
}
```

- What happens if a thread never calls `yield()`?
Implementing User-level Threads (5)

- Preemptive scheduling
  - Need to regain control of processor asynchronously.
  - Scheduler requests that a timer interrupt be delivered by the OS periodically.
    - Usually delivered as a UNIX signal
    - Signals are just like software interrupts, but delivered to user-level by the OS instead of delivered to OS by hardware
  - At each timer interrupt, scheduler gains control and context switches as appropriate.
Threading Models (1)

- Many-to-One (N:1)
  - Many user-level threads mapped to a single kernel thread
  - Used on systems that do not support kernel threads.
  - Solaris Green Threads
  - GNU Portable Threads
Threading Models (2)

- One-to-One (1:1)
  - Each user-level thread maps to a kernel thread.
  - Windows 98/NT/2000/XP, OS/2, Linux, Solaris 9+
Threading Models (3)

- Many-to-Many (M:N)
  - Allows many user-level threads to be mapped to many kernel threads.
  - Allows the OS to create a sufficient number of kernel threads.
  - Solaris prior to v9, IRIX, HP-UX, Tru64
Windows XP Threads (1)

- **One-to-one model**
  - A process in Windows XP is inert; it executes nothing
    - A process simply owns a 4GB address space that contains code and data for an application.
    - In addition, a process owns other resources, such as files, memory allocations, and threads.
  - Every process in Windows XP has a primary thread.
    - Threads in Windows XP are kernel-level threads.
    - Per-thread data structures:
      » Total user/kernel time, kernel stack, thread-scheduling info., thread-local storage array,
      » Thread environment block (TEB),
      » List of objects thread is waiting on, synch. info. etc.
Windows XP Threads (2)

- **Data structures for processes**
  
  - **EPROCESS**: Executive Process Block
    - Process ID, Parent Process ID, Exit status, Create and exit time
    - Memory management info., etc.
  
  - **KPROCESS**: Kernel Process Block
    - Kernel time, User time
    - Scheduling info. (process base priority, default thread quantum)
    - Process state
    - Thread list (KTHREADs), etc.
  
  - **PEB**: Process Environment Block
    - Image info. (base address, version number, etc.)
    - Process heap info., etc.
    - A user-space data structure
Windows XP Threads (3)

Data structures for threads

- **ETHREAD: Executive Thread Block**
  - Create and exit times, Process ID, Pointer to EPROCESS
  - Thread start address, etc.

- **KTHREAD: Kernel Thread Block**
  - Total user time, Total kernel time, Kernel stack info.
  - Thread scheduling info.
  - Synchronization info., etc.

- **TEB: Thread Environment Block**
  - Thread ID
  - User-mode stack info. (stack base, stack limit)
  - Thread local storage, etc.
  - A user-space data structure
Windows XP Threads (4)
Windows XP Threads (5)

Fibers vs. Threads

- Fibers are often called “lightweight” threads.
  - They allow an application to schedule its own “threads” of execution.

- Fibers are invisible to the kernel.
  - They are implemented in user-mode in Kernel32.dll

- Fibers interface
  - ConvertThreadToFiber() converts a thread to a running fiber.
  - A new fiber can be created using CreateFiber().
  - The new fiber runs until it exits or until it calls SwitchToFiber().

- Fibers provide a functionality of the many-to-many model.
**Linux Threads (1)**

- **Original LinuxThreads implementation**
  - http://pauillac.inria.fr/~xleroy/linuxthreads/
  - In Linux, the basic unit is a “task”.
    - In a program that only calls fork() and/or exec(), a task is identical to a process.
    - A task uses the `clone()` system call to implement multithreading.
  - **One-to-one model**
    - Linux creates a task for each application thread using clone().
  - **Resources can be shared selectively in clone()**:
    - CLONE_PARENT ; parent process
    - CLONE_FS ; FS root, current working dir., umask, ...
    - CLONE_FILES ; file descriptor table
    - CLONE_SIGHAND ; signal handler table
    - CLONE_VM ; address space
Linux Threads (2)

- POSIX compatibility problems
  - **POSIX**: A single process that contains one or more threads.
    - The following resources are specific to a thread, while all other resources are global to a process:
      - CPU registers, user stack, blocked signal mask
  - **Linux**: Separate tasks that may share one or more resources.
    - The following resources may be shared between tasks via clone(), while all other resources are local to each task:
      - Address space, signal handlers, open files, working directory
      - PID, PPID, UID, GID, pending signal mask, ...???
  - Problems in signal handling, exec(), exit(), process suspend/resume, etc.
Linux Threads (3)

- Approaches for POSIX compliance
  - NGPT (Next Generation POSIX Threading) – by IBM
    - M:N model
    - Extends GNU Pth library (M:1) by using multiple Linux tasks.
  - NPTL (Native POSIX Threading Library) – by RedHat
    - 1:1 model
    - Adopted for Linux kernel 2.6.