Chapter 4: Threads
Threads

Conventional View of Process

Conceptually, a Process has two fundamental characteristics:

- **A unit of Resource Ownership**
  - Allocated an Address Space
  - May have Open Files
  - May utilize I/O Devices

- **A unit of Scheduling**
  - The CPU Scheduler dispatches one Process at a Time to the CPU for execution
  - Each Process has an execution state (Recall PCB)

- **But Modern OS concepts treats the two as Independent tasks**
  - Process ~ Unit of Resource Ownership
  - Thread ~ Unit of Scheduling
Process vs. Thread

Process as a Unit of Resource Ownership
A Process has:
- Address Space to store the Executable Code
- Global Variables, heap, stack
- OS Resources (files, I/O Devices)

Thread is a Unit of Scheduling
A thread is a single sequential execution stream within a process
- A thread has its own:
  - Program Counter (PC)
  - Stack Pointer (PC)
  - Registers
- A thread shares with other threads in the Process:
  - Address Space
  - Program Code
  - OS Resources (files, I/O Devices)
Single Thread and Multi-Threaded Processes

A Thread is a Single Sequential Flow of Control

What does it mean to have a single thread in a program?

- There is only one point of execution, at any instance, within a single threaded Process

Single Threaded Process

<table>
<thead>
<tr>
<th>code</th>
<th>data</th>
<th>files</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>registers</th>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

thread ➔
Single Thread and Multi-Threaded Processes

A Process may contain multiple threads of control

What does it mean to have multiple threads within a process?

- Conceptually, all the threads in the process can execute concurrently
- All the threads share a common **address space**

Since the threads execute within the single address space:

- Hardware must be configured to allow threads to read/write the same memory locations
- Each thread executes on a separate stack with its local variables
- Global variables (off-stack) are shared among all threads ~ Common data
Threads

Stack

Each thread has its own Execution Stack

- Suppose P calls Q, and Q calls R
- Assume P needs to wait for Q and R to complete executions, before resuming his, then P’s execution stack will hold P’s:
  - Parameters
  - PC
  - Local variables
Why Threads?

**Can we do without concurrency?**

- A Process with multiple threads makes a great server (e.g., print server) – why?
  - Have one server process and many “worker” threads
  - If one thread blocks (such as read request, others will continue executing)
  - Threads interact/communicate via shared memory (common data), no need for IPC
    - Global variables

- Threads are inexpensive (unlike Processes)
  - Only need stack and register
  - Use minimum resources:
    - No new address space, global data or program data

- But, no protection between threads within the Process
  - May interfere with each other since they share common address space
Multithreading Models

User-Level Threads

- User-Level Threads: Library of functions (to create, fork, switch..) that allow user processes to create and manage their own threads

Benefits

- Does not require modification to the OS
- Simple representation – each thread is represented by a PC, registers, stack, and a small control block, all stored in the process’ address space
- Fast – Creating a new thread, switching between threads and synchronization are done by function calls, no kernel intervention is needed
Multithreading Models

User-Level Threads

- User-Level Threads: Library of functions (to create, fork, switch..) that allow user processes to create and manage their own threads

Disadvantages

- Lack of coordination between threads and the OS Kernel
  - Process gets one time slice, independent of number of threads within the process

- Requires non-blocking system call
  - Entire process blocks if a single thread blocks

- If one thread causes a page fault, the entire process blocks
Multithreading Models

Kernel-Level Threads

Kernel-Level Threads: Kernel provides system calls to create and manage threads

Advantages

- Kernel has full knowledge of all threads
  - Scheduler may choose to give a process with several threads more time slice than a process with one thread
- Good for applications that frequently block
  - Entire process blocks if a single thread blocks

Disadvantages

- Slow – Thread operations are 100s of times slower than for user-level threads
- Significant overhead and kernel complexity
  - Kernel must manage and schedule both threads and Processes
  - Requires a full TCB for each thread
Relationship Between User/Kernel –Level Threads

Many-to-One Model

- Many user-level threads map to one kernel thread
- Thread management is done by thread library in User space
- Entire Process blocks if a thread makes a blocking system call
Relationship Between User/Kernel –Level Threads

One-to-One Model

- Each user-level threads map to a kernel thread
- Increased concurrency
- Allows another thread to run if a user thread makes a blocking system call
- Creating a user thread is expensive
  - Requires creating another kernel thread (Hence restrictions on number of threads enforced)
- Examples
  - + Windows 95/200/NT
  - + OS/2
Relationship Between User/Kernel –Level Threads

Many-to-Many Model

- Maps user-level threads to a smaller or equal number of kernel threads
- Allows the OS to create a sufficient number of kernel threads
- Kernel schedules another thread for execution when a user thread makes a blocking system call
- Examples
  - Windows NT/200 with ThreadFiber package
Threading Issues  
fork() and exec() System Calls

- If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded?

- Some Unix systems have two versions of fork()
  - One duplicates all threads
  - Other duplicates only the thread that invoked the fork() system call

- If a thread invokes the exec() system call, program specified in parameter to exec will replace the entire process – including all threads
Thread Cancellation

- Thread Cancellation is the task of terminating a thread before it has completed

- Why cancel a thread?
  - Multiple threads are concurrently searching through a dB, and one thread returns result. Remaining threads might be canceled
  - Multiple threads are utilized to load a single web page. If a user clicks on the stop icon while the page is loading, all threads loading the page are cancelled
Thread Cancellation

Two scenarios

- **Asynchronous Cancellation**
  - One thread immediately terminates the target thread

- **Deferred Cancellation**
  - The target thread periodically checks if it should terminate. Thread terminates itself in an orderly fashion
How do threads accomplish concurrency in a single CPU environment?

1) **Thread Control Block (TCB)**
   - holds
     - Registers
     - PC
     - Stack Pointer
     - Dispatcher Info
       - Attributes unique to the thread’s execution (e.g., priority)
   - TCB info defines the thread’s execution “state”
Threads
Concurrency

2) **Scheduler**

- Schedules each thread to run by loading pertinent "Ready State" TCB info onto the CPU
  - Registers
  - PC
  - Stack Pointer
- Save "state" in the TCB (and wait for an event ~ sleep/timer)
- Select another thread (when event occurs)
  - Load its state into TCB and repeat above steps

The Scheduler maintains a list of threads in the Ready state
Threads

Execution States

- Threads exists in one of three states
  - Running – Owns the CPU
  - Blocked – Waiting for I/O, synch
  - Ready – Waiting for CPU (Ready List)

Thread State Diagram
Threads
Scheduling

- How does Scheduler select a thread, in the Ready state list to run?
  - **FIFO**
    - Each Ready thread is placed on tail end of list, and threads are scheduled from the leading end of the list
  - **LIFO**
    - Each Ready thread is placed on leading end of list, and threads are scheduled from the leading end of the list
  - **TCB Priority Field**
    - Sort ready list based on priority setting
When I/O request Completes, an interrupt is generated
- This causes CPU to stop intermittently
- The CPU runs interrupt handler

Handler saves state of interrupted thread
- PC, register, stack pointer

Handler continues to run
- Restore state of Interrupted/new thread

CPU resumes execution for the thread in Restored State
Signals

- Signals are used to notify a Process that an event has occurred (Unix systems)
  - Internal Events
    - I/O Request causes a thread to block
      - I/O disk, Keyboard entry
    - When a thread waits for another thread to complete execution, the waiting thread blocks
  - External Events
    - Interrupt ~ sent at completion of I/O requests
    - Timer ~ when timer expires
Thread Libraries

- **Thread Library**
  - Set of API to Create and Manage Threads
    - User Space Library:
      - POSIX Pthreads (can also be in Kernel space)
        - No System call
    - Kernel Space Library
      - Win32
      - POSIX Pthreads
        - API invokes system call to Kernel
  - Thread “fork” creates a new thread
    - pthread_create (…) POSIX API
    - CreateThread (…) Win32 API
Thread Libraries
Creating a New Thread

- What does it mean to create a new thread?
  - Create a new TCB and Execution Stack
  - Initialize values for register and PC’s initial address in the TCB
  - Place thread on Ready List (Scheduler’s task)

So what is the difference between Thread fork and UNIX fork from Chapter 3?