Preview

- The Thread Model
- Motivation of Threads
- Benefits of Threads
- Implementation of Thread
  - Implement thread in User’s Mode
  - Implement thread in Kernel’s Mode
The Thread Model

- In traditional operating systems, each process has an address space and a single thread of control.
- But in modern software, it is desirable to have multiple threads of control in the same address space.
- Each thread inside a process needs its independent space for running, but shares the same address space as the program.
The Thread Model

Threads

- Threads are processes in a process!!! – multiple executions in the same process environment.
- Each thread has its own: thread ID, program counter, register set, and stack.
- Different threads are not quite as independent as different processes since they share same address space.
- It shares with other threads belonging to the same process its code section, data section and other operating system resources such as files and child processes.
The Thread Model

- Multiple processes running on a computer—Processes share physical memory, disks, printers and other resources.
- Multiple threads running on a process – the threads share an address space, open files, and other resources.
- Multiple threads working together – By sharing a set of resources, threads can work together closely to perform some task.
The Thread Model

(a) Each of them operates in a different address space.

(b) All three of them share the same address space.
The Thread Model

- The CPU switches rapidly back and forth among the threads providing the illusion that the threads are running in parallel. (Same idea as multiprogramming)

- No protection between threads – using same address space. (e.g. one thread can completely wipe out another thread’s stack)

- A thread can be in any one of several states: running, blocked, ready. (like a traditional process)
The Thread Model

<table>
<thead>
<tr>
<th><strong>Per process items</strong></th>
<th><strong>Per thread items</strong></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>
The Thread Model

- Each thread has its own stack – Each thread’s stack contains one frame for each procedure called but not returned.

- Multithreading – processes normally start with a single thread present. This thread has the ability to create new threads by calling a library procedure (e.g. thread_create).

- When a thread has finished its work, it can exit by calling a library procedure (e.g. thread_exit).
Each thread has its own stack for storing its execution history.
Motivation of Threads

- With threads
  - Instead of thinking about interrupts and context switches, we can think about parallel executions.
  - Ability for the parallel entities to share an address space and all of its data. This is essential for certain applications.
  - Faster creation: easier to create/destroy than processes.
  - Useful on system with multiple CPUs, where real parallelism is possible.
Motivation of Threads

- Many software packages that run on modern desktop PC are multithreaded, e.g. web server, word processor.
- An application typically is implemented as a separate process with several threads of control.
Motivation of Threads

Ex) Web browser
1. A thread for displaying images or text
2. A thread for retrieving data from network

Ex) Word processor
1. A thread for displaying graphics
2. A thread for reading input from keyboard
3. A thread for performing spelling and grammar checking in the background
Benefits of Thread

- Threads share resources.
- Light weighted – allocating memory and resources for each process creation is expensive.
- Utilization of multi-processor system – each thread in a process can run on one CPU in multi-processor system. It can greatly shorten the run time.
Threads Implementation

- Implementing thread in user space
  - Threads are handled by the run-time system
  - OS does not know the existence of threads

- Implementing thread in the kernel
  - Thread are handled by OS
Implementing Threads in User’s space
Implementing Threads in User’s space

- The kernel does not know anything about threads. It is just managing single-thread process.
- The first advantage – user-level threads package can be implemented on an operating system that does not support threads.
- The threads run on top of a run-time system (correction of procedures that manage threads thread_create, thread_exit, thread_yield)
Implementing Threads in User’s space

- Each process needs its own private thread table to keep track of the threads in that process (managed by the run-time system).
- Thread table – stack pointer, program counter, registers, state, and so on.
- When a thread’s state become ready or blocked state, all information regarding the thread is stored in its own thread table.
Implementing Threads in User’s space

- When a running thread needed to be blocked (waiting for a job done by other thread),
  1. it calls a run-time system procedure.
  2. The run-time system procedure saves the thread’s information
  3. Choose one thread from the ready queue
  4. Load new thread’s information and run it

- When a thread is finished running for the moment, thread_yield is called (which is running in user’s space) saves the thread’s information and call scheduler, which is also running on user’s space.
Implementing Threads in User’s space

- User-level thread allows each process to have its own customized scheduling algorithm
- User-level threads are generally fast to create and manage, but they have drawbacks.
  - If the kernel is single-threaded, then any user-level thread performing a blocking system call will cause the entire process to block, even though other threads are available to run within the application.
- POSIX (Pthreads), Math (C-threads), Solaris 2 (UI-threads)
Implementing Threads in Kernel’s Space
Implementing Threads in Kernel’s Space

- It is directly supported by operating system.
- The kernel performs thread creation, scheduling and management (keeps track of a thread table for each thread) in kernel’s space.
- Because thread management is done by the operating system, kernel threads are generally slower to create and manage than the user-level thread.
Implementing Threads in Kernel’s Space

- Since the kernel is managing the threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution.
- In a multiprocessor environment, the kernel can schedule threads on different processors.
- Windows NT, Windows 2000, Solaris 2, BeOS, Tru64 UNIX
Interprocess Communication

- Three issues in interprocess communication

1. How one process can pass information to another?
2. How to make sure two or more processes do not get into the critical region?
3. Proper sequencing when dependencies are present (e.g. A create outputs, B consume the outputs).
Interprocess Communication
(Race Condition)

Race Condition
- A situation where two or more processes are reading or writing some shared data and the final result depends on the order in which the processes run.
Interprocess Communication
(Race Condition)

When a process wants to print a file, it enters a file name in a special spooler directory.

Printer daemon periodically checks spooler directory if any file needs to be printed.
Interprocess Communication (Race Condition)

- How to avoid race condition?
  
  **Mutual exclusion** – some way of making sure that if one process is using a shared variable or file, the other processes will be excluded from using the same variable or file.

- The choice of the algorithm for achieving mutual exclusion is a major design issue in any operating system.
Interprocess Communication
(Race Condition)
Interprocess Communication
(Race Condition)

- Critical section (critical region) – The part of program where the shared memory is accessed.
- If we could arrange matters such that no two processes were ever in their critical regions at the same time, we can avoid race conditions.
Interprocess Communication
(Race Condition)

A solution for the race condition should have following conditions

1. No two processes may be simultaneously inside their critical regions – mutual exclusion
2. No process running outside its critical region may block other processes
3. No process should have to wait forever to enter critical region
4. No assumptions may be made about speeds or the number of CPUs.