Threads, SMP, and Microkernels

Chapter 4
Multithreaded Programming

- So what is multithreaded programming? Basically, multithreaded programming is implementing software so that two or more activities can be performed in parallel within the same application.
- This is accomplished by having each activity performed by its own thread.
- A thread is a path of execution through the software that has its own call stack and CPU state.
- Threads run within the context of a process, which defines an address space within which code and data exist, and threads execute.
Multithread Programming

Example Program (sequential version)

#include <stdio.h>

void do_oneThing(int *);
void do_another_thing(int *);
void do_wrap_up(int, int);

int r1 = 0, r2 = 0;

void main()
{
    do_one_thing( &r1 );
    do_another_thing( &r2 );
    do_wrap_up(r1, r2);
}
void do_one_thing(int *pnum_times)
{
    int i, j, x;

    for (i=0; i<4; i++) {
        printf("doing one thing\n");
        for (j=0; j<10000; j++) x += i;  // wait some time
        (*pnum_times)++;
    } 
}
void do_another_thing(int *pnum_times)
{
    int i, j, x;

    for (i=0; i<4; i++) {
        printf("doing another thing\n");
        for (j=0; j<10000; j++) x += i;  // wait some time
          (*pnum_times)++;
    }
}
void do_wrap_up(int one_times, int another_times)
{
    int total;

    total = one_times + another_times;
    printf("wrap up: one thing %d, another %d, total %d\n",
           one_times, another_times, total);
}
Tasking/Threading/Processing

- The terms multitasking, multithreading, and multiprocessing are different technologies often used interchangeably.
- Multitasking is the ability of an operating system to switch among tasks quickly to give the appearance of simultaneous execution of those tasks.
- Multithreading extends the idea of multitasking to apply within applications, giving applications the capability to separate their own tasks into individual threads.
- Multiprocessing refers to multiple processors in one computer, executing the threads that are ready to run.
Process

- Unit of resource ownership
  - given temporary ownership of certain resources
  - e.g. allocated virtual address space, set of opened files

- Unit of scheduling/dispatching
  - is given the processor for execution
  - its execution path may be interleaved with that of other processes
Processes and Threads

- The unit of scheduling is referred to as a thread (or lightweight process)

- The unit of resource ownership is referred to as a process or task
Figure 4.1  Threads and Processes [ANDE97]
Process

- Has a virtual address space which holds the process image
- Protected access to files, I/O resources and other processes
Thread

- has execution state (running, ready, etc.)
- thread context saved when thread not running
- has an execution stack
- some per-thread static storage for global variables
- access to the entire memory and all resources of its process
  - all threads of a process share this
Uni-, Multithreaded Process Models

Single-Threaded

Multithreaded
Benefits of Threads

- process = a heavy-weight entity
  thread = a lighter weight entity
- less time needed
  - to create a new thread than a process
  - to terminate a thread than a process
  - to switch between two threads within the same process
- since threads within the same process share memory and files, they can communicate with each other without involving the kernel
- penalty: lesser inter-thread protection
The main benefit of threads (as compared to multiple processes) is that the context switches are much cheaper than those required to change current processes. Sun reports that a fork() takes 30 times as long as an unbound thread creation and 5 times as long as a bound thread creation.
Thread States and Operations

- key thread states: running, ready, blocked
- basic thread operations
  - spawn
    - spawn another thread
  - block
  - unblock
  - finish
    - deallocate space for register context and stacks
Thread-Process Coupling

- **Process → Thread**
  - suspending a process involves suspending all threads of the process
  - termination of a process terminates all threads within the process

- **Thread → Process**
  - relationship between thread blocking and process blocking; two alternatives
Example (RPC requests)

Time

Process 1

Server

Server

Thread A
(Process 1)

Server

Thread B
(Process 1)

Blocked

Running
Thread Design Alternatives

- User-level threads
  - all thread management is done by the application process
  - kernel is not aware of the existence of threads

- Kernel-level threads
  - kernel maintains context information for the threads (and the process)
  - scheduling is done by kernel on a thread basis
  - W2K, Linux, and OS/2 are examples of this approach
Threads

- Kernel threads can be independently scheduled on CPUs. Context switching between kernel threads is very fast because memory mappings do not have to be flushed.

- User threads are scheduled on their LWPs via a scheduler in libthread. This scheduler does implement priorities, but does not implement time slicing. If time slicing is desired, it must be programmed in.

- Locking issues must also be carefully considered by the programmer in order to prevent several threads from blocking on a single resource.
Combined ULT/KLT Approaches

Idea is to combine the best of both approaches

- Solaris is an example of an OS that combines both ULT and KLT
- Thread creation done in the user space
- Bulk of scheduling and synchronization of threads done in the user space
- The programmer may adjust the number of KLTs
- User-level threads (threads library) invisible to the OS are the interface for application parallelism
- Kernel threads the unit that can be dispatched on a processor
- Lightweight processes (LWP) each LWP supports one or more ULTs and maps to exactly one KLT
Difference between UT and KT

- User threads are those which the application creates,
- kernel threads are those which the kernel can "see" and schedule.

- A user application can implement a multi-threaded application without kernel threads by implementing a user-space scheduler to switch between the various threads for the process.
- Since this exists only in user-space, the kernel still sees only a single thread -- these threads are therefore referred to as unbound, since they do not correspond to a thread the kernel can see and schedule.
- If each of these threads are bound to a single kernel thread, there is no need for the user-space scheduler, and these threads are referred-to as bound.
- Notice that only bound threads (those which the kernel can see) can give the benefits of parallelism on a multi-processor machine.
The API for handling user threads is provided by a library, the *threads library*.

A user thread only exists within a process; a user thread in process A cannot reference a user thread in process B.

The user threads API, unlike the kernel threads interface, is part of a portable programming model. Thus, a multi-threaded program can easily be ported to other systems.
Combined Approaches

- example: Solaris
- both kernel and user-level threads (KLTs and ULTs)
- one or more ULTs mapped onto one KLT
- most activity at user level
  - thread creation is done completely at user level
  - also, the bulk of scheduling and synchronization of threads done at user level
Figure 4.6  User-Level and Kernel-Level Threads
Microkernels

- historical evolution
  - monolithic OS $\rightarrow$ micro-kernel based OS
- microkernel contains only the basic operating systems functions
- many services traditionally included in the operating system are now separate external subsystems
  - device drivers
  - file systems
  - virtual memory manager
  - windowing system
  - security services
Kernel Architectures

Layered Kernel

- Users
- File System
- IPC
- I/O Device Mgmnt
- Virtual Mem.
- Proc. Mgmnt

Hardware

User mode

Kernel mode

Microkernel

- client process
- device drivers
- file server
- process server
- virtual memory

Hardware

User mode

Kernel mode
Microkernel Organization Benefits

- Uniform interface on requests made by a process
  - no need to differentiate user/kernel-level services
  - all services provided via message passing
- Extensibility
  - incremental addition of new services is possible
- Flexibility
  - possible to add/remove features
- Portability
  - all processor specific code is in microkernel
  - changes needed to port are fewer, grouped
Examples

- Windows 2000
- Solaris
Figure 4.12  Windows 2000 Process and Its Resources
Windows 2000 Process Object

- **Object Type**
  - Process ID
  - Security Descriptor
  - Base priority
  - Default processor affinity
  - Quota limits
  - Execution time
  - I/O counters
  - VM operation counters
  - Exception/debugging ports
  - Exit status

- **Object Body Attributes**

- **Services**
  - Create process
  - Open process
  - Query process information
  - Set process information
  - Current process
  - Terminate process

(a) Process object
W2K Thread Object

Object Type
- Thread ID
- Thread context
- Dynamic priority
- Base priority
- Thread processor affinity
- Thread execution time
- Alert status
- Suspension count
- Impersonation token
- Termination port
- Thread exit status

Object Body Attributes

Services
- Create thread
- Open thread
- Query thread information
- Set thread information
- Current thread
- Terminate thread
- Get context
- Set context
- Suspend
- Resume
- Alert thread
- Test thread alert
- Register termination port

(b) Thread object
Figure 4.14  Windows 2000 Thread States
Solaris

- Process includes the user’s address space, stack, and process control block
- User-level threads
- Lightweight processes
- Kernel threads
Lightweight Processes

- A lightweight process can be considered as the swappable portion of a kernel thread.
- Another way to look at a lightweight process is to think of them as "virtual CPUs" which perform the processing for applications. Application threads are attached to available lightweight processes, which are attached to a kernel thread, which is scheduled on the system's CPU dispatch queue.
- LWPs can make system calls and can block while waiting for resources. All LWPs in a process share a common address space.
Figure 4.15  Solaris Multithreaded Architecture Example
Figure 4.16  Process Structure in Traditional UNIX and Solaris [LEWI96]
Figure 4.17  Solaris User-Level Thread and LWP States
Figure 4.18 Linux Process/Thread Model