Chapter 4
Threads, SMP, and Microkernels
Contents

- Processes and threads
- Symmetric multiprocessing
- Microkernels
- Window 2000 thread and SMP management
- Solaris thread and SMP management
- Linux process and thread management
Processes of Yesterday

- Unit of resource ownership
  - Process is allocated a virtual address space to hold the process image

- Unit of dispatching
  - Scheduled and dispatched by the OS
  - Execution may be interleaved with other processes

- These two characteristics are treated independently by today’s operating system
Processes of today

- Process
  - Unit of resource ownership
  - Virtual address space, main memory, I/O devices, and files

- Thread or light weight process
  - Unit of dispatching
  - Scheduled and dispatched by the OS
Multithreading

- Refers to the ability of an OS to support multiple threads of execution within a single process
- MS-DOS supports a single user process and a single thread
- UNIX supports multiple user processes but only supports one thread per process
- Windows 2000, Solaris, Linux, Mach, and OS/2 support multiple threads
Threads and Processes

- One process, one thread
- One process, multiple threads
- Multiple processes, one thread per process
- Multiple processes, multiple threads per process
Processes in Multithreaded Environment

- A process is the unit of resource allocation and a unit of protection
  - Have a virtual address space that holds the process image
  - Protected access to processors, other processes, files, and I/O resources
  - Per Process Items
    - Address space, global variable, open files, child processes, timers, signals, semaphores, account
Threads

- Within a process, there may be one or more threads, each with the followings
  - Thread execution state (running, ready, ...)
  - Saved thread context
    - Program counter, stack, register set, child threads, memory for local variables
  - Access to the memory and resources of its process
    - All threads of a process share this
Threaded Process Model

Single-Threaded Process Model
- Process Control Block
- User Stack
- User Address Space

Multithreaded Process Model
- Thread
  - Thread Control Block
  - User Stack
  - Kernel Stack
  - User Address Space
- Thread
  - Thread Control Block
  - User Stack
  - Kernel Stack
- Thread
  - Thread Control Block
  - User Stack
  - Kernel Stack
  - Kernel Stack
Benefits of Threads

- Takes less time to create a new thread than a process
  - No need to allocate a virtual address space
- Less time to terminate a thread than a process
- Less time 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 invoking the kernel
Uses of Threads in a Single-User Multiprocessing System

- Foreground and background work
  - One thread displays menus and read user input
  - Another thread executes user commands
- Asynchronous elements in the program can be implemented as threads
- Speed execution
  - On a multiprocessor system, multiple threads from the same process may be able to execute simultaneously
- Modular program structure
Threads

Actions that affect all of the threads in a process

- Suspending a process involves suspending all threads of the process since all threads share the same address space
- Termination of a process terminates all threads within that process
Thread States

- Operations associated with a change in thread state
  - Spawn
    - a thread within a process may spawn another thread within the same process
  - Block
  - Unblock
  - Finish
    - Deallocate register context and stacks
Remote Procedure Call Using Threads

(a) RPC Using Single Thread
Remote Procedure Call Using Threads

Thread A (Process 1)

Thread B (Process 1)

(b) RPC Using One Thread per Server (on a uniprocessor)
Implementation of Threads

- User-level threads (ULTs)
- Kernel-level threads (KLTs)
  - Also referred to as kernel-supported threads or lightweight processes
- Combined approaches
User-level and Kernel-level Threads

(a) Pure user-level

(b) Pure kernel-level

(c) Combined

User-level thread
Kernel-level thread
Process
User-Level Threads

- All thread management is done by the application
  - Kernel is not aware of the existence of threads

- Advantages of ULTs
  - Thread switching does not require kernel mode privileges
  - Scheduling is application specific
  - ULTs can run on any OS

- Disadvantages of ULTs
  - When a thread is blocked, all of the threads within that process are blocked
  - Cannot take advantage of multiprocessing
    - Kernel assigns one process to only one processor at a time
Figure 4.7 Examples of the relationship between User-level thread states and process states
Kernel-Level Threads

- Windows 2000 and OS/2 are examples of this approach
- Kernel maintains the context information for the process and the threads
- Scheduling by the kernel is done on a thread basis
Combined Approaches for Threads

- Example is Solaris
- Thread creation done in the user space
  - As is bulk of scheduling and synchronization
- Multiple ULTs from a single application are mapped onto some (smaller or equal) number of KLTs
# Relationship Between Threads and Processes

<table>
<thead>
<tr>
<th>Threads:Processes</th>
<th>Description</th>
<th>Example Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:1</td>
<td>Each thread of execution is a unique process with its own address space and resources.</td>
<td>Traditional UNIX implementations</td>
</tr>
<tr>
<td>M:1</td>
<td>A process defines an address space and dynamic resource ownership. Multiple threads may be created and executed within that process.</td>
<td>Windows NT, Solaris, Linux OS/2, OS/390, MACH</td>
</tr>
</tbody>
</table>
## Relationship Between Threads and Processes

<table>
<thead>
<tr>
<th>Threads:Process</th>
<th>Description</th>
<th>Example Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:M</td>
<td>A thread may migrate from one process environment to another. This allows a thread to be easily moved among distinct systems.</td>
<td>Ra (Clouds), Emerald</td>
</tr>
<tr>
<td>M:M</td>
<td>Combines attributes of M:1 and 1:M cases</td>
<td>TRIX</td>
</tr>
</tbody>
</table>
Categories of Computer Systems (Flynn)

- Single Instruction Single Data (SISD)
  - Single processor executes a single instruction stream to operate on data stored in a single memory

- Single Instruction Multiple Data (SIMD)
  - One instruction is executed on a different set of data by the different processors
  - Vector and array processors
Categories of Computer Systems

- Multiple Instruction Single Data (MISD)
  - A sequence of data is transmitted to a set of processors, each of which executes a different instruction sequence. Never implemented

- Multiple Instruction Multiple Data (MIMD)
  - A set of processors simultaneously execute different instruction sequences on different data sets
Parallel Processor Architectures

- Parallel Processor
  - SIMD (single instruction multiple data stream)
  - MIMD (multiple instruction multiple data stream)
    - Shared-Memory (tightly coupled)
      - Master/Slave
      - Symmetric Multiprocessors (SMP)
    - Distributed-Memory (loosely coupled)
      - Clusters
Symmetric Multiprocessing

- Kernel can execute on any processor
- Typically each processor does self-scheduling from the pool of available processes or threads
Symmetric Multiprocessor Organization

```
Processor
  ↓
L1 Cache
  ↓
L2 Cache
  ↓

Processor
  ↓
L1 Cache
  ↓
L2 Cache
  ↓

Processor
  ↓
L1 Cache
  ↓
L2 Cache
  ↓

. . .

Main Memory
  ↓
I/O Subsystem
  ↓
I/O Adapter
  ↓
I/O Adapter
  ↓
I/O Adapter
```
Multiprocessor OS Design Considerations

- Simultaneous concurrent processes or threads
  - Kernel routines need to be reentrant
- Scheduling
  - May be performed by any processor
- Synchronization
- Memory Management
- Reliability and Fault Tolerance
Microkernel

- Small operating system core
- Contains only essential operating systems functions
- Many services traditionally included in the operating system are now external subsystems
  - Device drivers
  - File systems
  - Virtual memory manager
  - Windowing system
  - Security services
Figure 4.10  Kernel Architecture
Benefits of Microkernel Organization

● Uniform interface on request made by a process
  ♦ All services are provided by means of message passing

● Extensibility
  ♦ Allows the addition of new services

● Flexibility
  ♦ Not only can new features be added to OS, but existing features can be subtracted

● Portability
  ♦ Almost all processor-specific code is in the microkernel
    – Changes needed to port the system to a new processor are fewer and tend to be arranged in logical groupings
Benefits of Microkernel Organization

● Reliability
  ◆ Modular design
  ◆ Small microkernel can be rigorously tested

● Distributed system support
  ◆ Message can be sent without knowing what the target machine is

● Object-oriented operating system
  ◆ Components are objects with clearly defined interfaces that can be interconnected to form software
Microkernel Design

- Functions that must be included
  - Low-level memory management
    - Mapping each virtual page to a physical page frame
  - Inter-process communication
    - Message is the basic form
    - Message passing between separate processes involves memory-to-memory copying
      + current research on thread-based IPC and memory sharing scheme
  - I/O and interrupt management
Page Fault Processing

Application

Page fault

resume

Pager

Address-space function call

Microkernel
Windows 2000 Processes

- Implemented as objects
- An executable process may contain one or more threads
- Both process and thread objects have built-in synchronization capabilities
Process and its Resources

- Security access token
  - Used to validate the user’s ability to access secured objects

- Virtual address space
  - A series of blocks

- Object table
  - Have handles to other objects known to this process
  - One handle exists for each thread contained in this object
W2K Process and Its Resources

- Access token
- Virtual address space description
- Object Table
  - Handle1
  - Handle2
  - Handle3
  - ...
- Available objects
  - Thread x
  - File y
  - Section z
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
- Create process
- Open process
- Query process information
- Set process information
- Current process
- Terminate process

(a) Process object
# Windows 2000 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
Multithreading

- Threads in different processes may execute concurrently
- Multiple threads within the same process may be allocated to separate processors and execute concurrently
- Threads in different processes can exchange information through shared memory that has been set up between the two processes
Windows 2000 Thread States

- Ready
- Standby: selected to run next
- Running
- Waiting: blocked
- Transition: ready to run but the resources are not available (e.g: stack may be paged out of memory)
- Terminated
Thread-related concepts used by Solaris

- **Process**
  - Includes the user’s address space, stack, and process control block

- **User-level threads**
  - Implemented through a threads library

- **Lightweight processes**
  - Can be viewed as a mapping between ULTs and kernel threads

- **Kernel threads**
Solaris Multithreaded Architecture

User

Kernel

Hardware

User-level thread  Kernel-level thread  Light-weight Process  Processor


Threads Library
Figure 4.17  Solaris User-Level Thread and LWP States
Solaris Thread Execution

- A ULT in the active state is currently assigned to a LWP

- Events that cause ULT to leave active state
  - Synchronization
    - Invoke a concurrency action and go into the sleeping state
  - Suspension
    - Go into the stopped state
  - Preemption
    - Go into the runnable state
  - Yielding
    - Yield to another runnable thread
Linux Process

- State
- Scheduling information
- Identifiers
- Interprocess communication: SVR4 is supported
- Links
- Times and timers
- File system
- Virtual memory
- Processor-specific context
Linux States of a Process

- Running
- Interruptable
  - Blocked state
- Uninterruptable
  - Another blocked state, but will not accept any signals
- Stopped
  - Halted, and can only resume by positive action from another process
- Zombie
Linux Process/Thread Model

- **Stopped**
- **Ready**
- **Executing**
- **Zombie**

States and Transitions:
- Creation
- Signal
- Signal or Event
- Event
- Scheduling
- Termination

States:
- Running State
- Interruptible
- Uninterruptible