Concurrent Programming

Prof. Sanjeev Setia
Concurrent & Distributed Software Systems
CS 475

Hardware Architectures

- ⌘Uniprocessors
- ⌘Shared-memory multiprocessors
- ⌘Distributed-memory multicomputers
- ⌘Distributed systems

Concurrent Systems

- Essential aspects of any concurrent system
  - Execution context - state of a concurrent entity
  - Scheduling - deciding which context will run next
  - Synchronization - mechanisms that enable execution contexts to coordinate their use of shared resources

Application classes

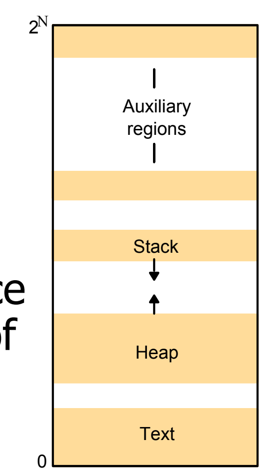
- Multi-threaded Programs
  - Processes/Threads on same computer
  - Window systems, Operating systems
- Distributed computing
  - Processes/Threads on separate computers
  - File servers, Web servers
- Parallel computing
  - On same (multiprocessor) or different computers
  - Goal: solve a problem faster or solve a bigger problem in the same time

Concurrent Programming

- Process = Address space + one thread of control
- Concurrent program = **multiple threads of control**
  - Multiple single-threaded processes
  - Multi-threaded process

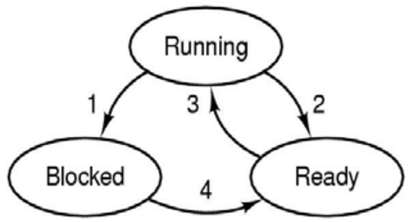
Process Concept

- A process includes:
  - program counter
  - code segment
  - stack segment
  - data segment
- Process = Address Space + One thread of control



Address space

Process States



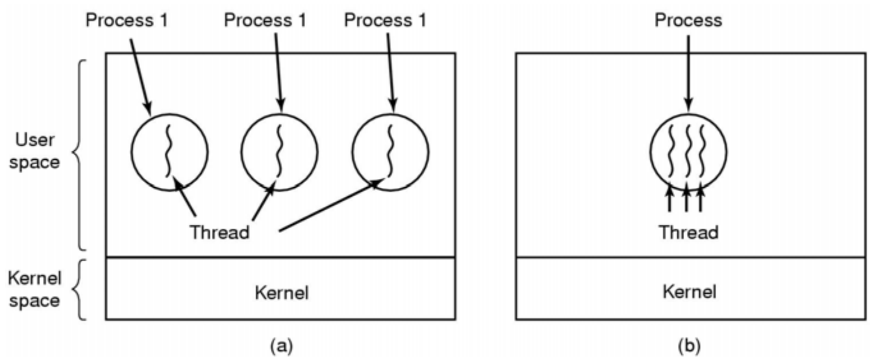
1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

- Possible process states
  - running
  - blocked
  - ready
- Transitions between states shown

Process Scheduling Queues

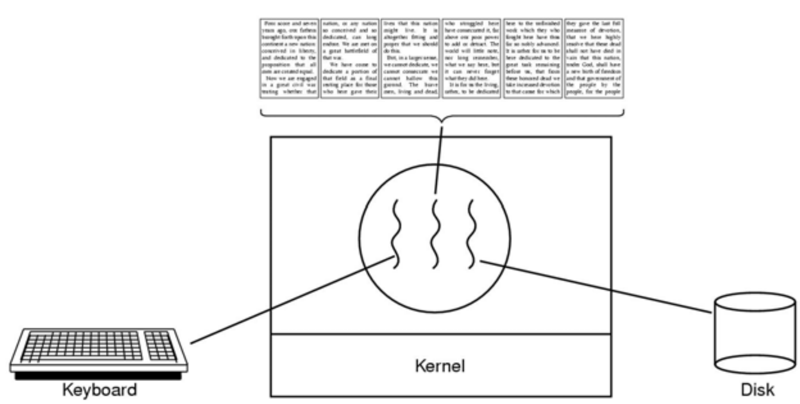
- Ready queue – set of all processes residing in main memory, ready and waiting to execute.
- Device queues – set of processes waiting for an I/O device.
- Processes migrate between the various queues during their lifetime.

The Thread Model (1)



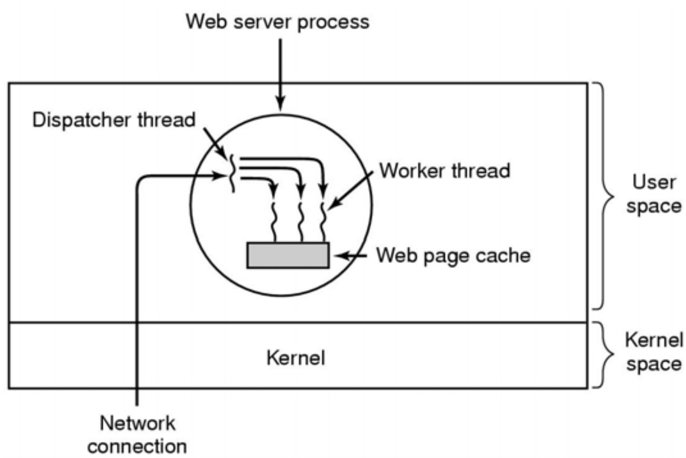
(a) Three processes each with one thread
(b) One process with three threads

Thread Usage (1)



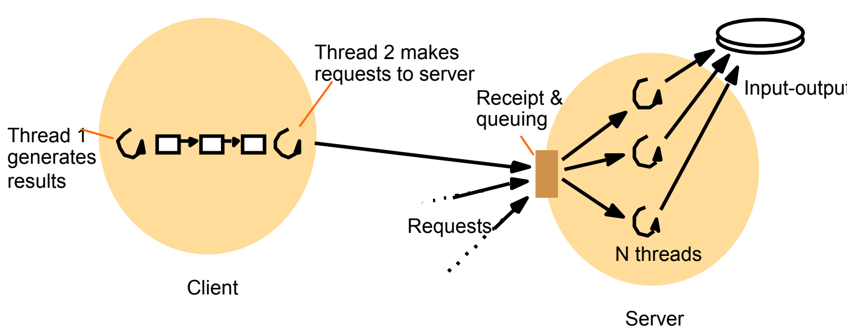
A word processor with three threads

Thread Usage (2)

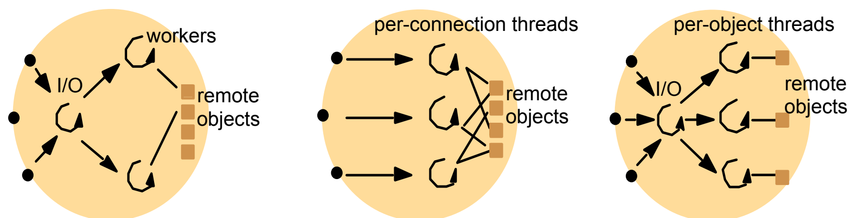


A multithreaded Web server

Client and server with threads



Alternative server threading architectures



a. Thread-per-request b. Thread-per-connection c. Thread-per-object

Threads: Motivation

- ⌘Traditional UNIX processes created and managed by the OS kernel
- ⌘Process creation expensive - fork system call
- ⌘Context switching expensive
- ⌘Cooperating processes - no need for protection (separate address spaces)

Threads

- Execute in same address space
  - separate execution stack, share access to code and (global) data
- Smaller creation and context-switch time
- Can exploit fine-grain concurrency
- Easier to write programs that use asynchronous I/O or communication

State associated with processes and threads

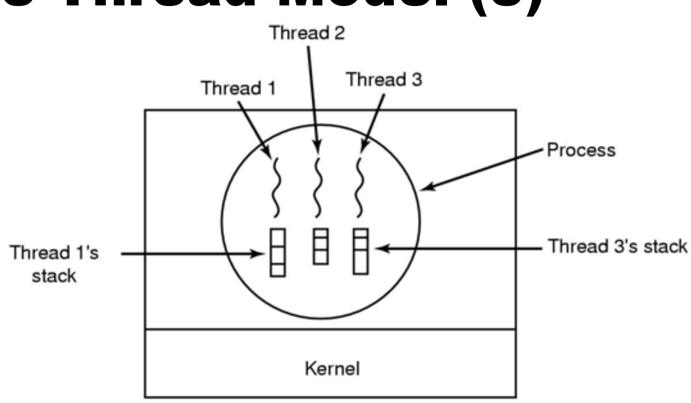
Process	*Thread*
Address space tables	Saved processor registers
Communication interfaces, open files	Priority and execution state (such as *BLOCKED*)
Semaphores, other synchronization objects	Software interrupt handling information
List of thread identifiers	Execution environment identifier
Pages of address space resident in memory; hardware cache entries	

The Thread Model (2)

Per process items	Per thread items
Address space Global variables Open files Child processes Pending alarms Signals and signal handlers Accounting information	Program counter Registers Stack State

- Items shared by all threads in a process
- Items private to each thread

The Thread Model (3)



Each thread has its own stack

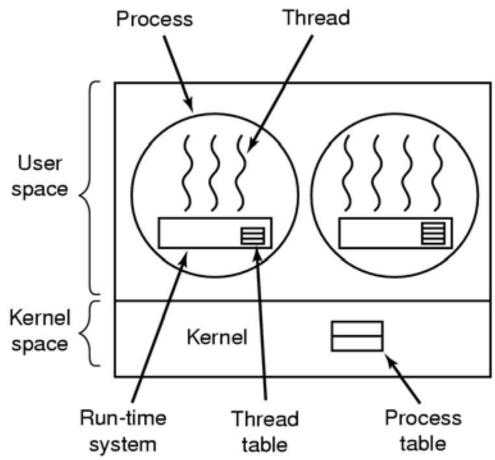
Threads cont'd

- Less protection against programming errors
- User-level vs kernel-level threads
  - kernel not aware of threads created by user-level thread package (e.g. Pthreads), language (e.g. Java)
  - user-level threads typically multiplexed on top of kernel level threads in a user-transparent fashion

User-Level Threads

- Thread management (scheduling, thread creation) done by user-level threads library
- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris *threads*
  - Java threads

Implementing Threads in User Space

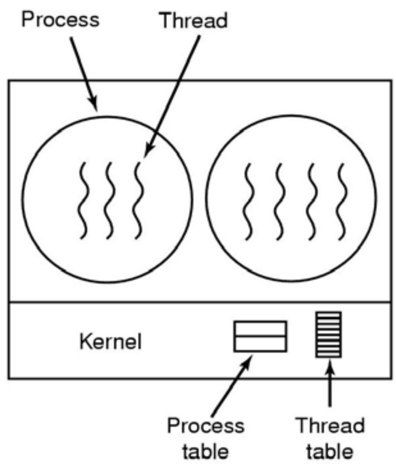


A user-level threads package

Kernel Threads

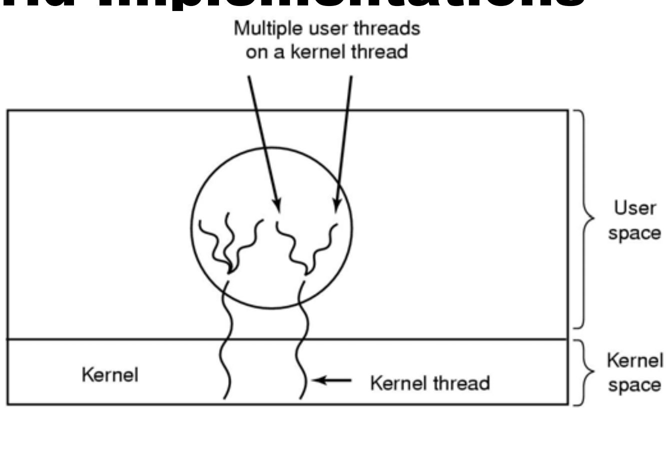
- Supported by the Kernel
- Examples
  - Windows 95/98/NT/2000
  - Solaris
  - Tru64 UNIX
  - BeOS
  - Linux

Implementing Threads in the Kernel



A threads package managed by the kernel

Hybrid Implementations



Multiplexing user-level threads onto kernel-level threads

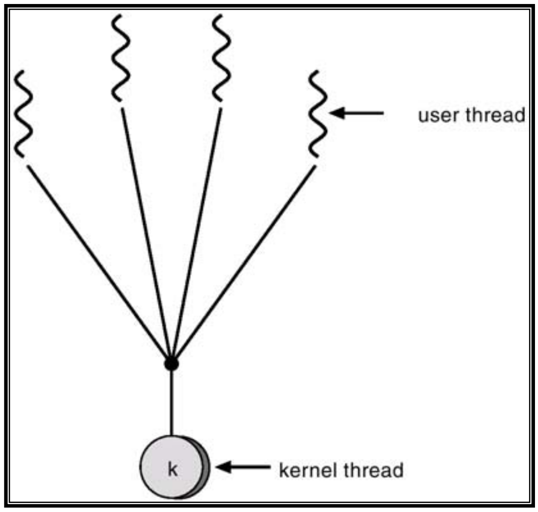
Multithreading Models

- ⌘Many-to-One
- ⌘One-to-One
- ⌘Many-to-Many

Many-to-One

- ⌘Many user-level threads mapped to single kernel thread.
  - If one user-level thread makes a blocking system call, the entire process is blocked even though other user-level threads may be "ready"
- ⌘Used on systems that do not support kernel threads.

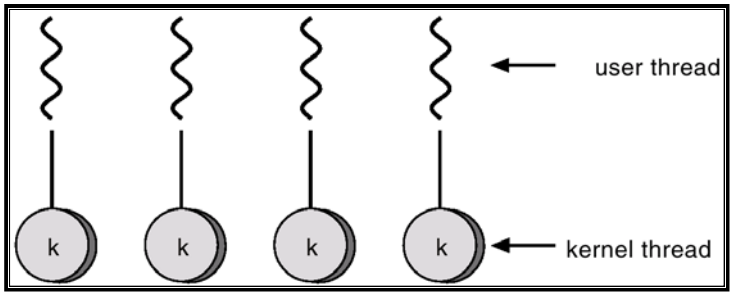
Many-to-One Model



One-to-One Model

- Each user-level thread maps to kernel thread.
- Examples
  - Windows 95/98/NT/2000
  - OS/2

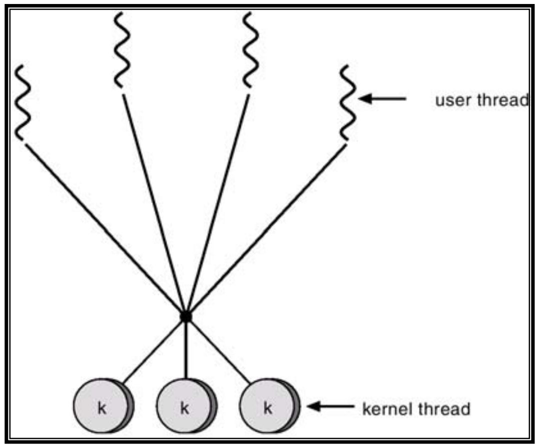
One-to-one Model



Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads.
- Allows the operating system to create a sufficient number of kernel threads.
- Solaris 2
- Windows NT/2000 with the *ThreadFiber* package

Many-to-Many Model



Pthreads

- a POSIX standard (IEEE 1003.1c) API for thread creation and synchronization.
- API specifies behavior of the thread library, implementation is up to development of the library.
- Common in UNIX operating systems.

Java Threads

- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Java threads are managed by the JVM.

Creating and Using threads

- Solaris Multi-threading Library
  - supports Pthreads API + own Solaris threads API
  - pthread_create, pthread_join, pthread_self, pthread_exit, pthread_detach
- Java
  - provides a Runnable interface and a Thread class as part of standard Java libraries
    - users program threads by implementing the Runnable interface or extending the Thread class

Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)
Creates a new thread in the *SUSPENDED* state, which will belong to *group* and be identified as *name*; the thread will execute the *run()* method of *target*.

setPriority(int newPriority), getPriority()
Set and return the thread's priority.

run()
A thread executes the *run()* method of its target object, if it has one, and otherwise its own *run()* method (*Thread* implements *Runnable*).

start()
Change the state of the thread from *SUSPENDED* to *RUNNABLE*.

sleep(int millisecs)
Cause the thread to enter the *SUSPENDED* state for the specified time.

yield()
Enter the *READY* state and invoke the scheduler.

destroy()
Destroy the thread.

Creating threads

```
class Simple implements Runnable {
  public void run() {
    System.out.println("this is a thread");
  }
}

Runnable s = new Simple();
Thread t = new Thread(s);
t.start();
```

Alternative strategy: Extend Thread class (not recommended unless you are creating a new type of Thread)

Cooperating concurrent processes

- Shared Memory
  - Semaphores, mutex locks, condition variables, monitors
  - Mutual exclusion
- Message-passing
  - Pipes, FIFOs (named pipes)
  - Message queues

Synchronization Mechanisms

- Pthreads
  - Semaphores
  - Mutex locks
  - Condition Variables
  - Reader/Writer Locks
- Java
  - Each object has an (implicitly) associated lock and condition variable

Race Conditions

Consider two threads T1 and T2 repeatedly executing the code below

```
int count = 100;   // global

increment ( ) {
    int  temp;

    temp = count;
    temp = temp + 1;
    count = temp;
}
```

Time ↓

Thread T1	Thread T2
temp = 100	
count = 101	
	temp = 101
	count = 102
temp = 102	
	temp = 102
count = 103	
	count = 103

We have a *race condition* if two processes or threads want to access the same item in shared memory at the same

Assignment 1

- Three experiments
  - All you have to do is compile and run programs
  - Linux/Solaris
- First two experiments illustrate differences between processes and threads
- Third experiment shows a race condition between two threads