

# Chapter 4: Multithreaded Programming

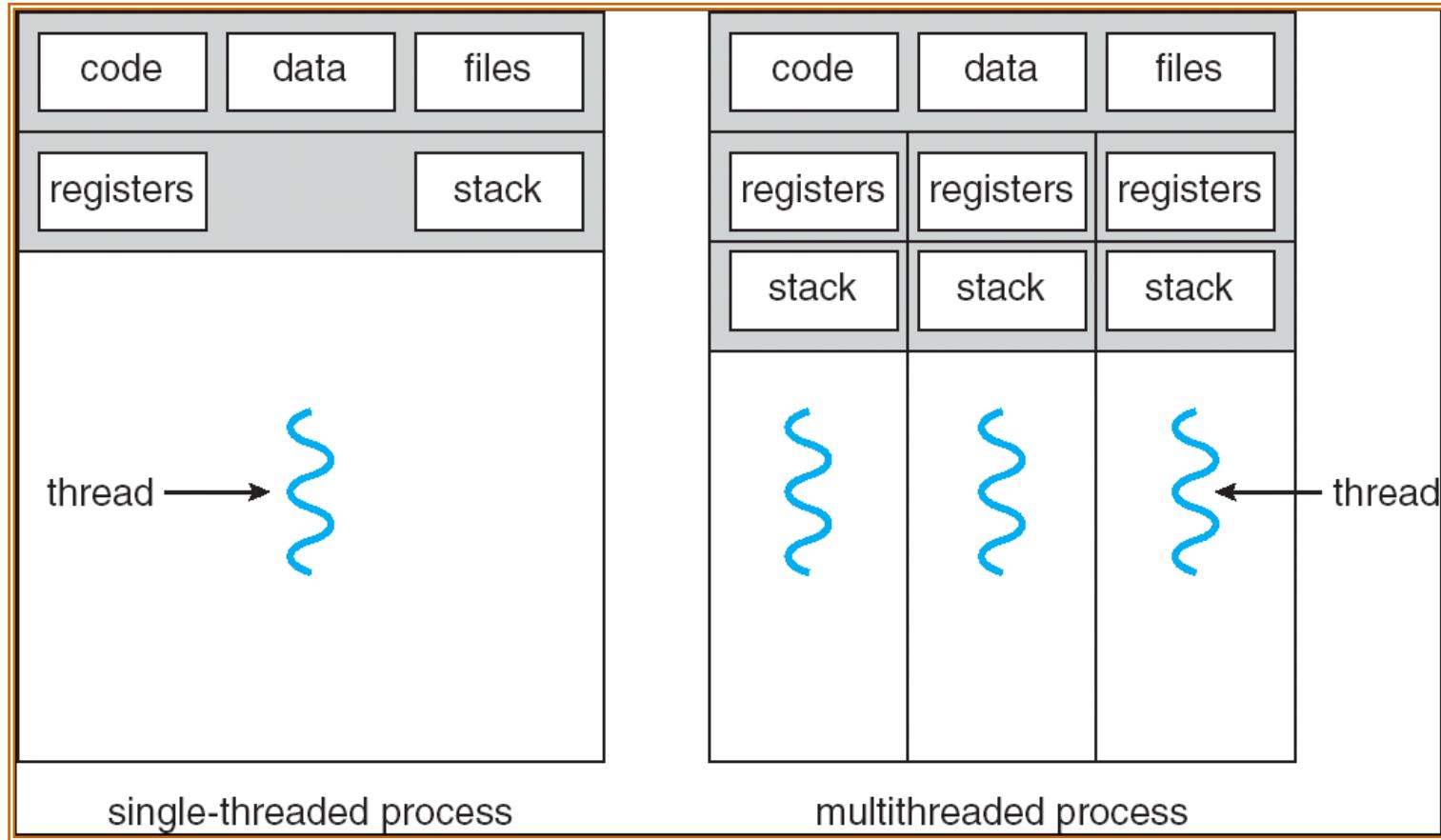
Prof. Li-Pin Chang  
CS@NYCU

# Chapter 4: Multithreaded Programming

- Overview
- Multithreading Models
- Thread Libraries
- Threading Issues
- Operating-System Examples

# OVERVIEW

# Single and Multithreaded Processes

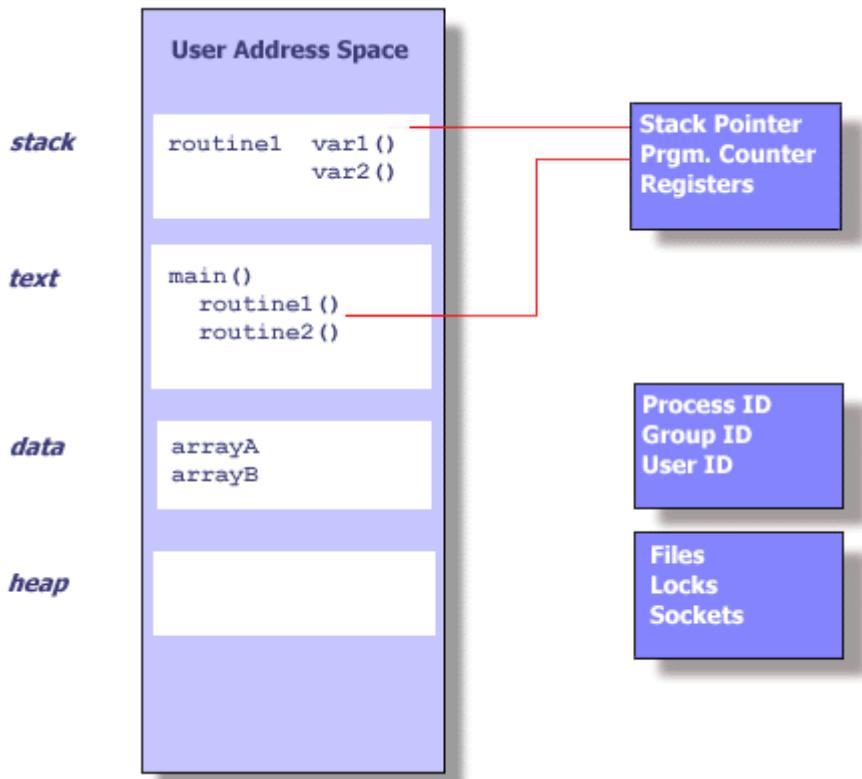


- A process is a “container” of all its threads

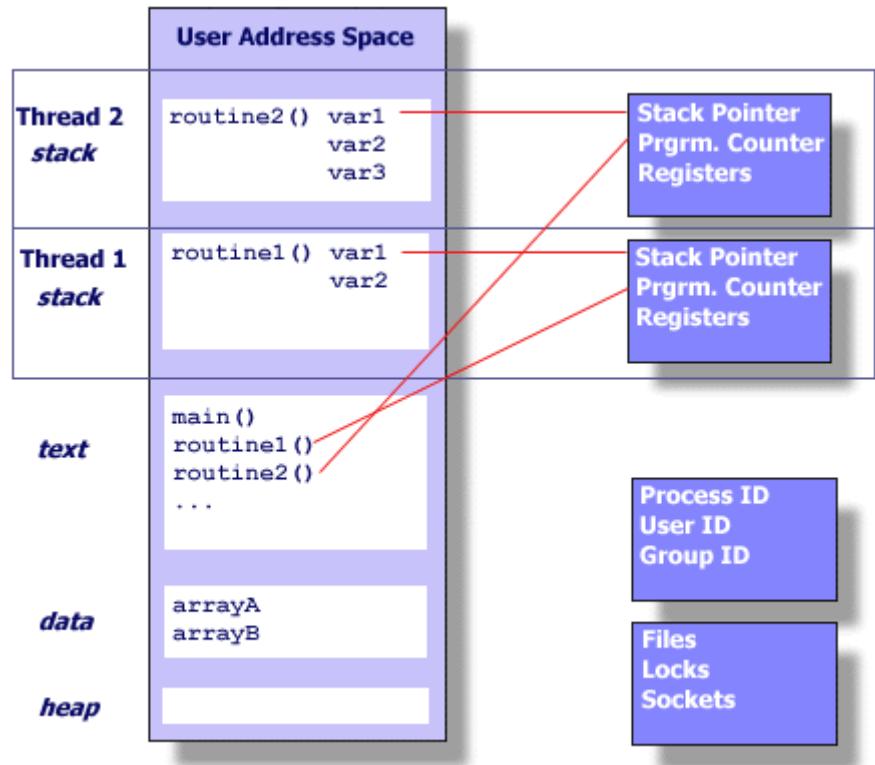
# Benefits

- Responsiveness
  - A thread accepts UI inputs while another does computation
- Resource Sharing
  - To share code and most of the data structures (no need for shared memory)
- Economy
  - A thread is a lightweight process
- Utilization of MP Architectures
  - To utilize multiple cores or to improve ILP

In Solaris it is 5 times slower to context switch a process than to context switch a thread, and 13 times slower for creation



A process



Two threads in a process

- This independent flow of control is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.

# MULTITHREADING MODELS

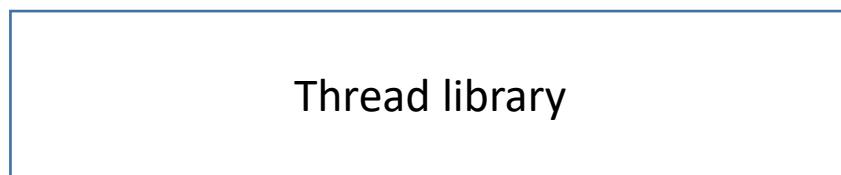
# Multithreading Models

- User threads are supported above the kernel and are managed without kernel support, while
- kernel threads are supported and managed directly by the operating system
- Mapping of user threads to kernel threads:
  - Many-to-One
  - One-to-One
  - Many-to-Many

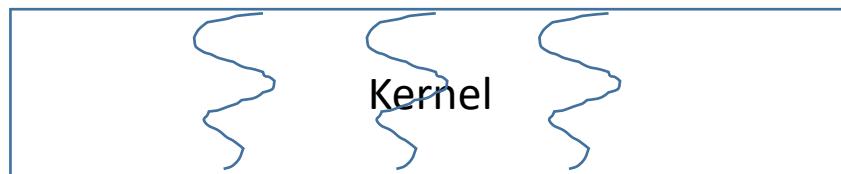
# Multithreading Models



User threads



Threading model  
(mapping)



Kernel threads



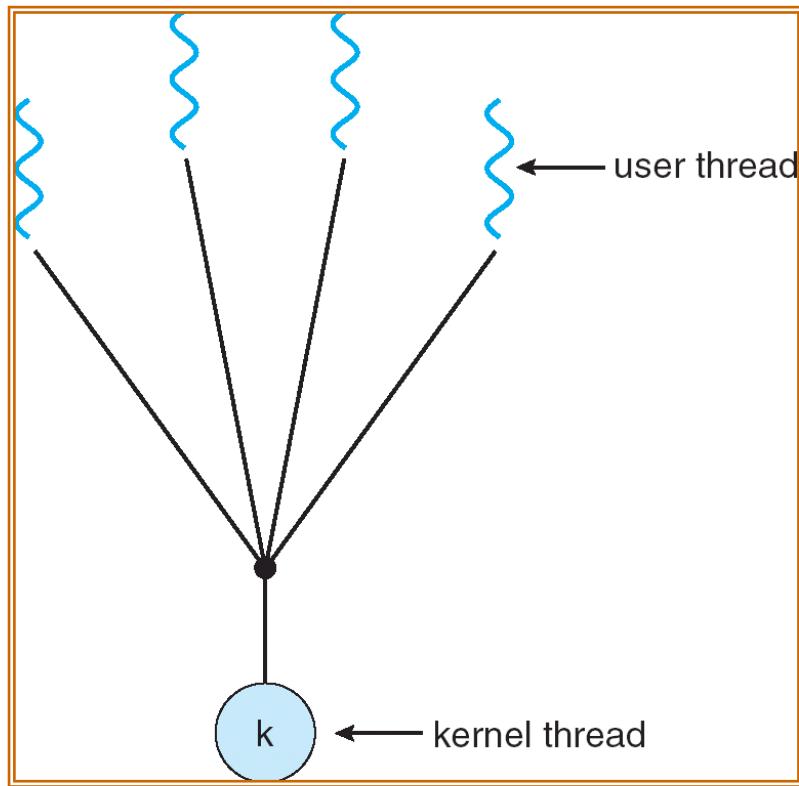
# Specification vs. Implementation

- Major thread libraries
  - Pthreads
  - Win32 threads
  - Java threads
- Pthread is a specification (part of POSIX)
  - Implement all the mandatory thread APIs
  - The thread model is, however, not part of the Pthread specification
  - Pthead can be implemented using 1-1, M-1, or M-M

# Many-to-One

- Many user-level threads mapped to single kernel thread
- Examples:
- Solaris Green Threads
  - For JDK. Options: green or native
- GNU Portable Threads
  - An implementation of the Pthread specification

# Many-to-One Model



- If one single thread issues a blocking system call, then the entire collection of threads become blocked
- Cannot enjoy the physical parallelism of multicore architecture

# Many-to-One Model

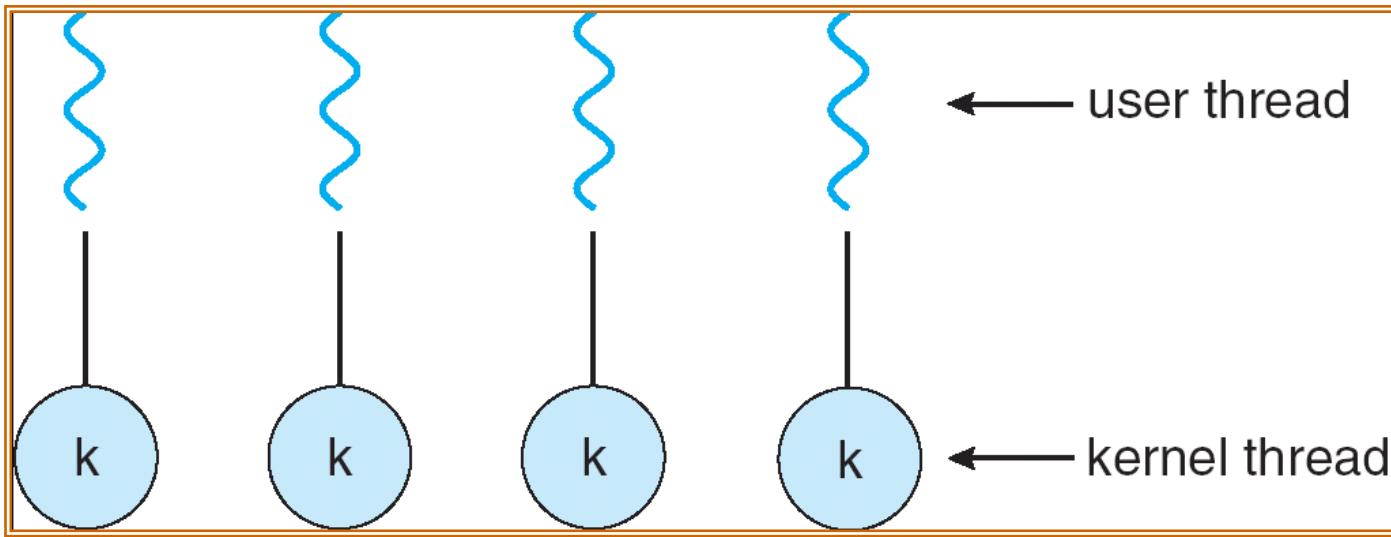
- Use special wrapper functions to prevent blocking (sync) calls from stalling all user threads
- For example, with GNU portable threads, instead of `read()`, we should use

```
ssize_t pth_read(int fd, void *buf, size_t nbytes)
```

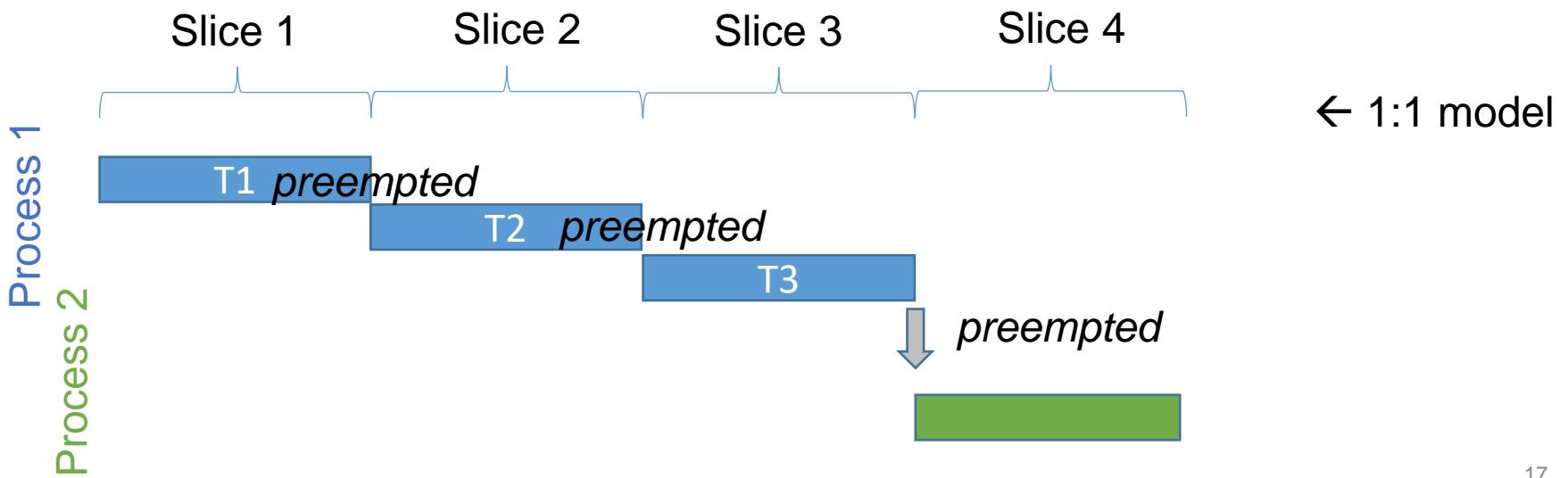
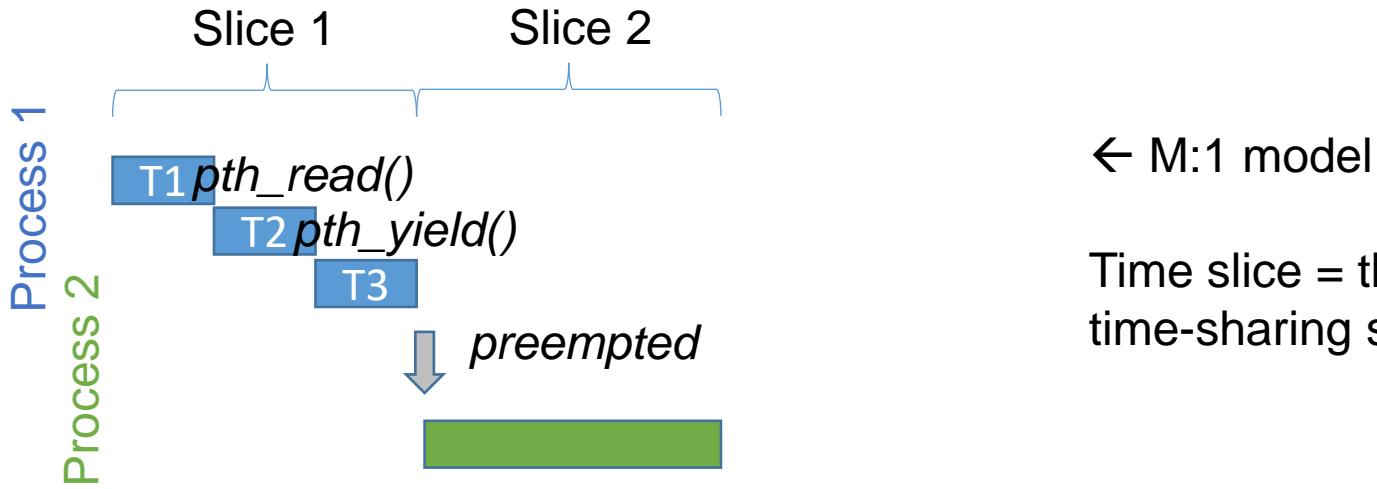
# One-to-One

- Each user-level thread maps to kernel thread
- Examples
  - Windows NT/XP/2000
  - Linux Pthread
  - Solaris 9 and later

# One-to-one Model



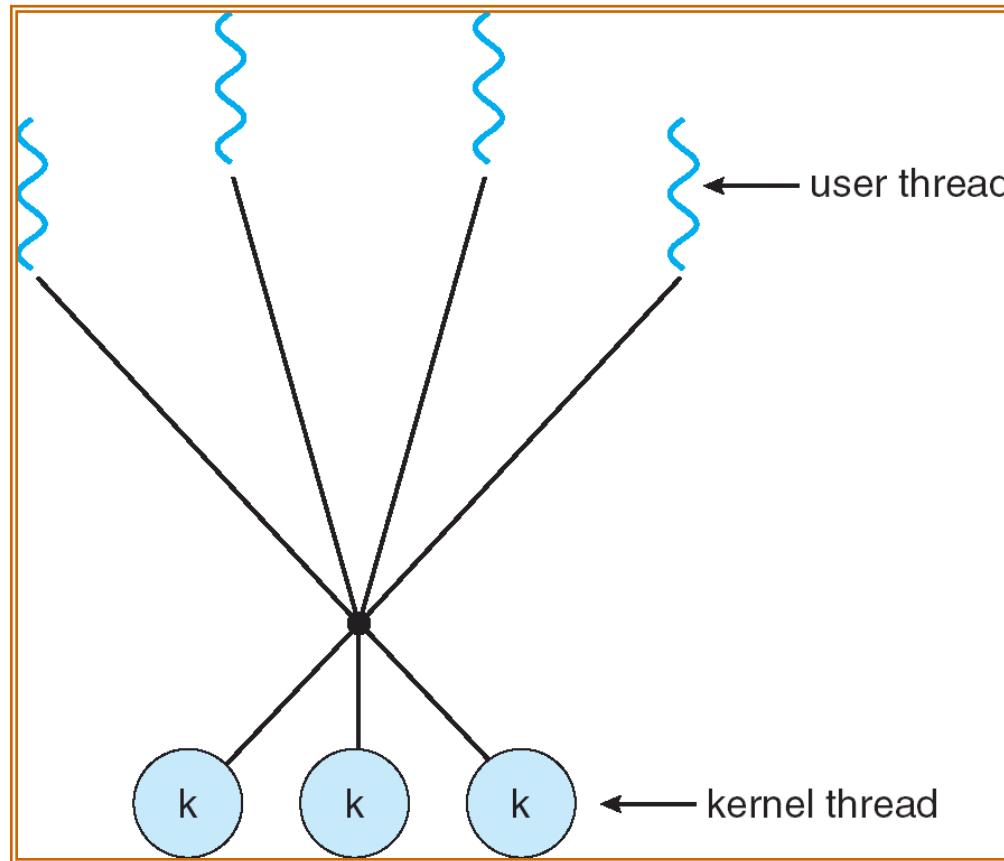
- Require kernel support; less portable
- Can exploit the parallelism of multicore architecture
- A blocking call will not block the whole thread set



# Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
  - One thread won't block the entire process
- Allows the operating system to create a sufficient number of kernel threads
  - More economic than 1-1 model
- Solaris prior to version 9
- Windows NT/2000 with the ThreadFiber package

# Many-to-Many Model

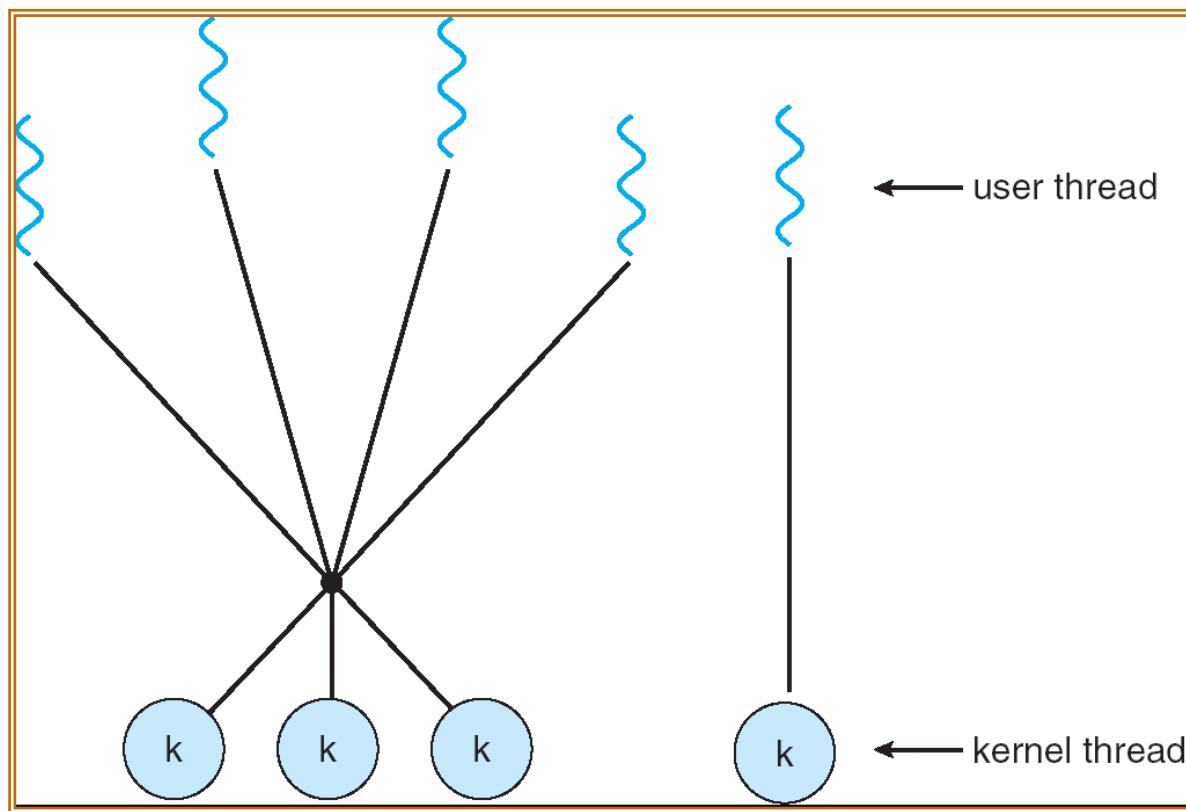


- A mixture of 1-1 and M-1

# Two-level Model

- Similar to M:M, except that static binding between user threads and kernel threads is permitted
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

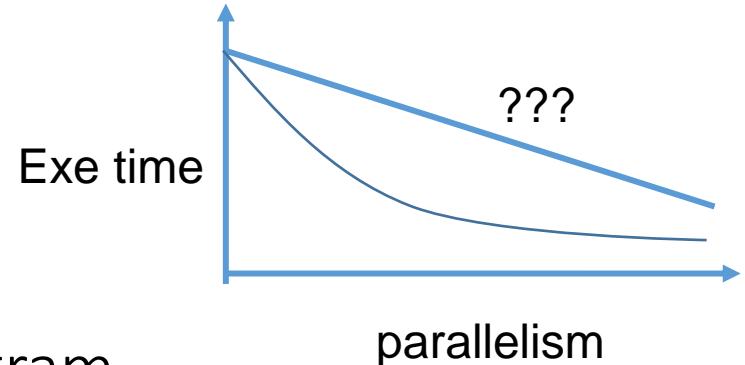
# Two-level Model



# Threading Models

- Specifications do not cover threading model
  - Pthread may be many-to-one, one-to-one, or many-to-many
  - Java green threads are many-to-one but Java threads are not defined
- Different thread library may adopt different threading models
  - GNU Portable Thread is many-to-one
  - Linux Pthread?
- Check the programmers' manual first, or write a small test program to make sure

# Amdahl's Law



- P: parallelizable portion of a program
- S: non-parallelizable portion of a program
- [-----P(16)-----][----S(8)----]
  
- Deg. Of parallelism=2
- [-----P(8)-----][----S(8)----] : speedup=1.5
- Deg. Of parallelism=16
- [P(1)][----S(8)----] : speedup=2.66
- Deg. Of parallelism → infinity
- [→0][----S(8)----] : speedup=3

## Definition [\[edit\]](#)

---

Amdahl's law can be formulated in the following way:

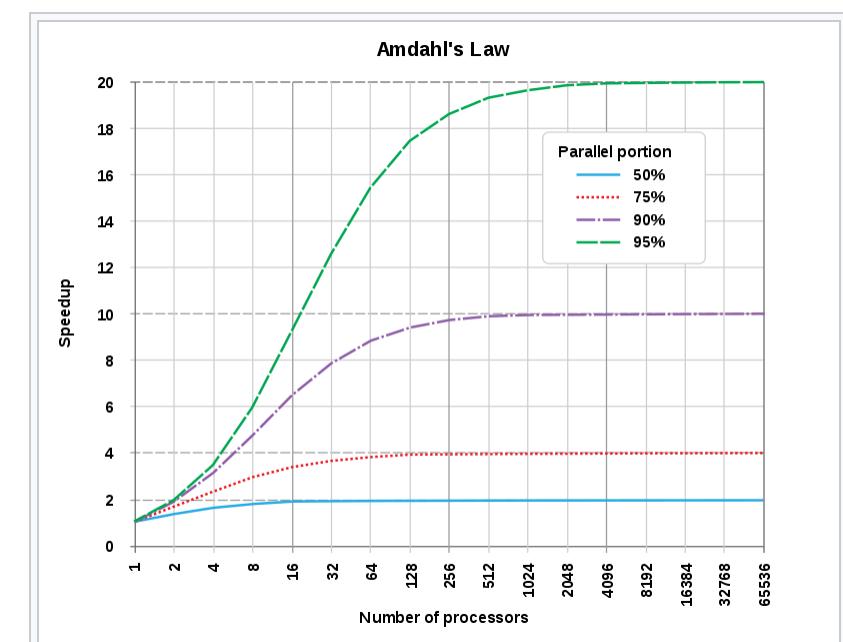
$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

If 30% of the execution time may be the subject of a speedup,  $p$  will be 0.3; if the improvement makes the affected part twice as fast,  $s$  will be 2. Amdahl's law states that the overall speedup of applying the improvement will be:

$$S_{\text{latency}} = \frac{1}{1 - p + \frac{p}{s}} = \frac{1}{1 - 0.3 + \frac{0.3}{2}} = 1.18.$$


---

[Notice] In real cases, as there are synchronization overhead among processes/threads, the speedup degrades when # of processes/threads is very large



# THREAD LIBRARIES

# Pthread

- =POSIX thread
- Pthread is a “specification”, not an implementation
- Implementations:
  - GNU portable thread (M-1)
  - Linux Pthread (1-1)
  - Mac OS X Pthread (?)

```

#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
    pthread_t tid; /* the thread identifier */
    pthread_attr_t attr; /* set of thread attributes */

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }

    /* get the default attributes */
    pthread_attr_init(&attr);
    /* create the thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* wait for the thread to exit */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* The thread will begin control in this function */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}

```

# Win32 Thread

- Again Win32 thread is a specification, implementation varies among WinXP, Win7, etc.
- Win32 thread APIs are very similar to Pthread APIs
- Win32 threads are referred to as objects/handle
  - WaitForSingleObject
  - CloseHandle

```

#include <windows.h>
#include <stdio.h>
DWORD Sum; /* data is shared by the thread(s) */
/* the thread runs in this separate function */

DWORD WINAPI Summation(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Sum += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    /* perform some basic error checking */
    if (argc != 2) {
        fprintf(stderr, "An integer parameter is required\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr, "An integer >= 0 is required\n");
        return -1;
    }

    // create the thread
    ThreadHandle = CreateThread(
        NULL, // default security attributes
        0, // default stack size
        Summation, // thread function
        &Param, // parameter to thread function
        0, // default creation flags
        &ThreadId); // returns the thread identifier

    if (ThreadHandle != NULL) {
        // now wait for the thread to finish
        WaitForSingleObject(ThreadHandle, INFINITE);

        // close the thread handle
        CloseHandle(ThreadHandle);
        ...
        printf("sum = %d\n", Sum);
    }
}

```

# OPERATING-SYSTEM THREAD SUPPORT

# Windows XP Threads

- Implements the one-to-one mapping
- Each thread contains
  - A thread id
  - Register set
  - Separate user and kernel stacks
  - Private data storage area
- The register set, stacks, and private storage area are known as the context of the threads
- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

# Linux Threads

- Linux refers to them as tasks rather than threads
- Thread creation is done through **clone()** system call
- **clone()** allows a child task to share the address space of the parent task (process)
  - But the stacks are separate
  - Different from **vfork()**

| flag          | meaning                            |
|---------------|------------------------------------|
| CLONE_FS      | File-system information is shared. |
| CLONE_VM      | The same memory space is shared.   |
| CLONE_SIGHAND | Signal handlers are shared.        |
| CLONE_FILES   | The set of open files is shared.   |

# Java Threads

- Java threads are managed by the JVM
- Java threads may be created by:
  - Extending Thread class
  - Implementing the Runnable interface
- Threading model
  - Java green thread: many-to-one
  - Java (native) thread: many-to-many (old) or one-to-one (recent)

# The JVM and the Host Operating System

The Java Virtual Machine (JVM) is generally built on top of a host operating system, which enables it to abstract the underlying operating system's complexities and offer a consistent environment for Java programs to run on any platform with JVM support. The JVM specification doesn't dictate how Java threads should be mapped to the operating system's threads, leaving that to the specific JVM implementation. For instance, on Windows XP, which uses a **one-to-one** threading model, each Java thread corresponds to a kernel thread. Conversely, operating systems that employ a **many-to-many** model, such as Tru64 UNIX, map Java threads accordingly. Solaris initially used a **many-to-one** model (via the green threads library) for JVM implementation, later adopting the **many-to-many** model, and starting with Solaris 9, moved to a **one-to-one** model. Additionally, the threading library used by the JVM might correspond to the thread library on the host OS; for example, a JVM on Windows might utilize the Win32 API for thread creation, while Linux, Solaris, and Mac OS X systems might use the Pthreads API.

-- Adapted from “Operating System Concepts”

# THREADING ISSUES

# Semantics of fork() and exec()

- Does fork() duplicate only the calling thread or all threads?
  - Undefined, but in many UNIX variants the **entire process** (including all its threads) is **duplicated**
- How about exec()
  - Again undefined, but in many UNIX variants the **entire process** (including all its threads) is **replaced**

# Signal Handling

- Synchronous signal
  - Always delivered to the thread that causes the signal
  - E.g., access violation
- Asynchronous signal
  - Typically delivered to the first thread that does not block the signal
  - E.g., process termination
- Varies from implementation to implementation

# End of Chapter 4

# Review Questions

1. Compare thread and process in terms of resource requirement and fault isolation
2. Modern operating systems are capable of the 1-1 thread model. Why does the M-1 model still exist?
3. What are the memory sections shared among threads?
4. Verify Amdahl's Law with your programming assignment