Chapter 4
Threads

张竞慧
办公室：计算机楼366室
电邮：jhzhang@seu.edu.cn
主页：http://cse.seu.edu.cn/PersonalPage/zjh/
电话：025-52091017

Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Windows Threads API
- Pthreads

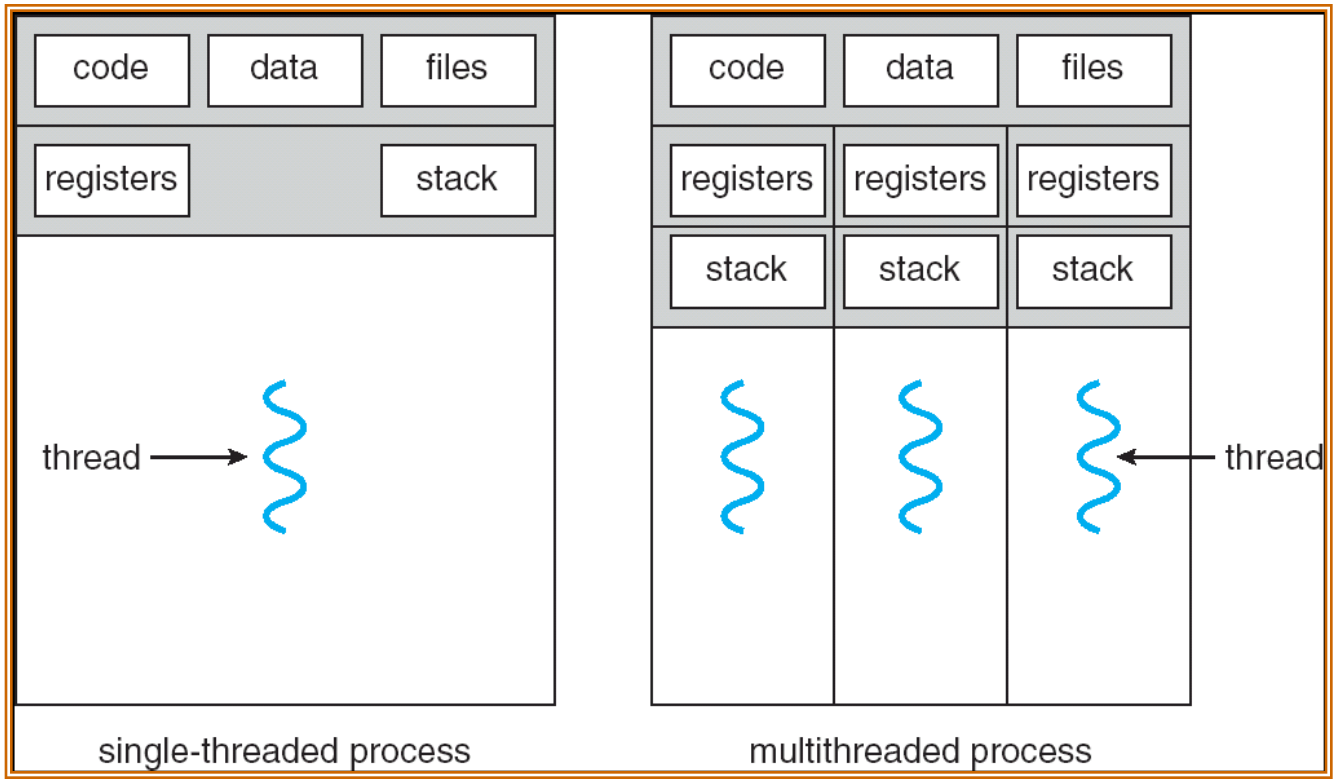
What is a thread?

- A *thread*, also known as *lightweight process* (LWP), is a basic unit of CPU execution.
- A thread has a thread ID, a program counter, a register set, and a stack. Thus, it is similar to a process has.
- However, a thread *shares* with other threads in the *same* process its code section, data section, and other OS resources (*e.g.*, files and signals).
- A process, or heavyweight process, has a *single* thread of control.

Modern Process with Threads

- Thread: *a sequential execution stream within process* (Sometimes called a "Lightweight process")
  - Process still contains a single Address Space
  - No protection between threads
- Multithreading: *a single program made up of a number of different concurrent activities*
  - Sometimes called multitasking, as in Ada …
- Why separate the concept of a thread from that of a process?
  - Discuss the "thread" part of a process (concurrency)
  - Separate from the "address space" (protection)
  - Heavyweight Process ≡ Process with one thread

Single and Multithreaded Processes



- Threads encapsulate concurrency: “Active” component
- Address spaces encapsulate protection: “Passive” part
  - Keeps buggy program from trashing the system
- Why have multiple threads per address space?

Thread State

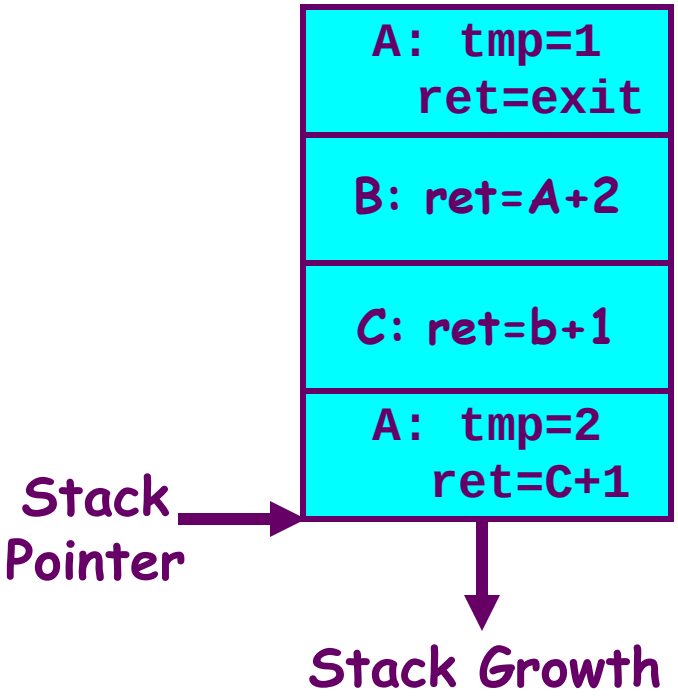
- State shared by all threads in process/addr space
  - Content of memory (global variables, heap)
  - I/O state (file descriptors, network connections, etc)

- State "private" to each thread
  - Kept in TCB ≡ Thread Control Block
  - CPU registers (including, program counter)
  - Execution stack – what is this?

- Execution Stack
  - Parameters, temporary variables
  - Return PCs are kept while called procedures are executing

Execution Stack Example

```
A(int tmp) {
  if (tmp<2)
      B();
 printf(tmp);
      }
    B() {
     C();
      }
    C() {
     A(2);
      }
    A(1);
```



- Stack holds temporary results
- Permits recursive execution
- Crucial to modern languages

Motivational Example for Threads

- Imagine the following C program:

```
main() {
    ComputePI("pi.txt");
    PrintClassList("clist.text");
}
```

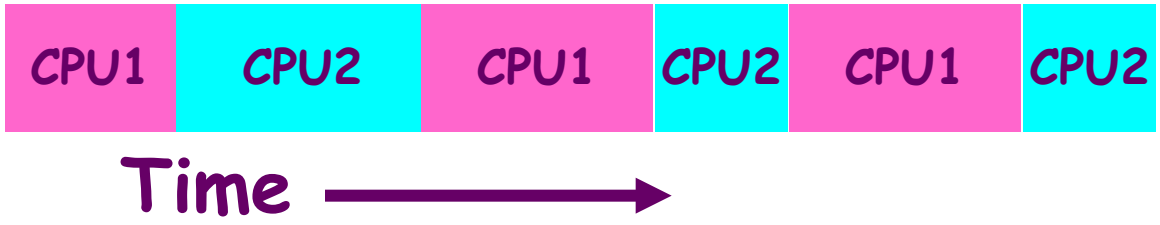
- What is the behavior here?
  - Program would never print out class list
  - Why? ComputePI would never finish

Use of Threads

- Version of program with Threads (loose syntax):

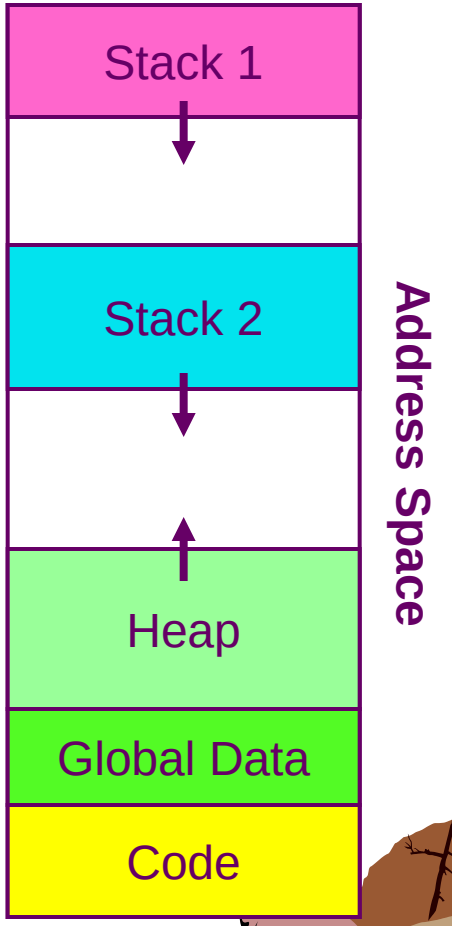
```
 main() {
  ThreadFork(ComputePI("pi.txt"));
  ThreadFork(PrintClassList("clist.text"));
}
```

- What does "ThreadFork()" do?
  - Start independent thread running given procedure
- What is the behavior here?
  - Now, you would actually see the class list
  - This *should* behave as if there are two separate CPUs



Memory Footprint: Two-Threads

- If we stopped this program and examined it with a debugger, we would see
  - Two sets of CPU registers
  - Two sets of Stacks



Actual Thread Operations

- thread_fork(func, args)
  - Create a new thread to run func(args)
- thread_yield()
  - Relinquish processor voluntarily
- thread_join(thread)
  - In parent, wait for forked thread to exit, then return
- thread_exit
  - Quit thread and clean up, wake up joiner if any

- pThreads: POSIX standard for thread programming

Some Numbers

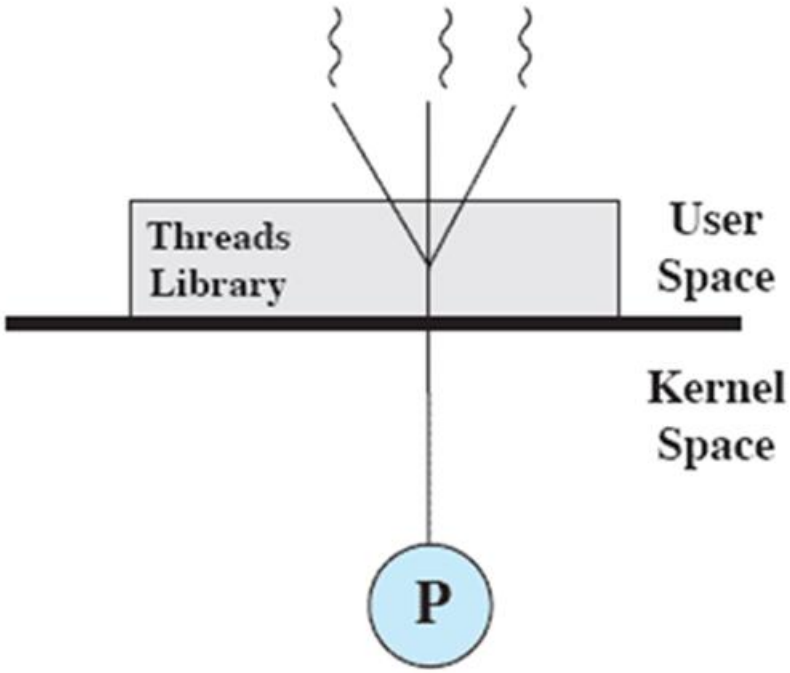
- Frequency of performing context switches: 10-100ms
- Context switch time in Linux: 3-4 μsecs (Current Intel i7 & E5).
  - Thread switching faster than process switching (100 ns).
  - But switching across cores about 2x more expensive than within-core switching.
- Context switch time increases sharply with the size of the working set*, and can increase 100x or more.

  * The working set is the subset of memory used by the process in a time window.
- Moral: Context switching depends mostly on cache limits and the process or thread's hunger for memory.

User Threads

- Thread management done by user-level threads library

- Examples
  - POSIX *Pthreads*
  - Mach *C-threads*
  - Solaris UI-*threads*

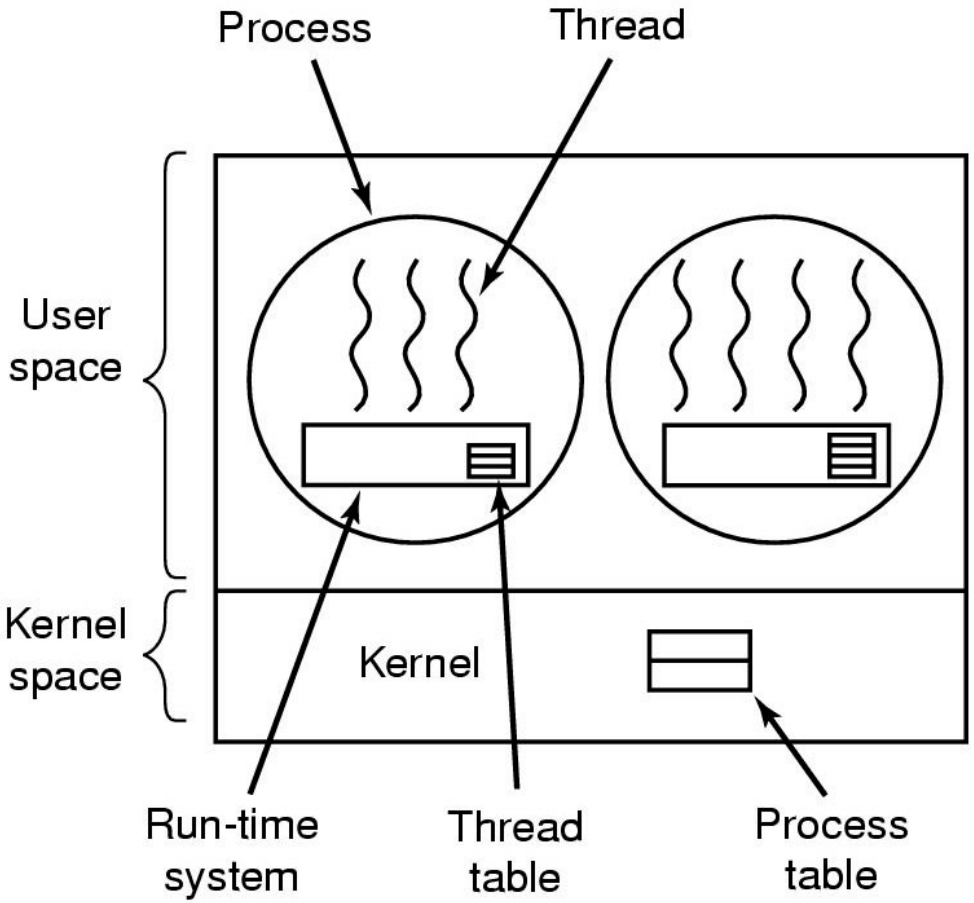


(a) Pure user-level

User Threads (Cont.)

- User threads are supported at the user level. The kernel is not aware of user threads.
- A library provides all support for thread creation, termination, joining, and scheduling.
- There is no kernel intervention, and, hence, user threads are usually more efficient.
- Unfortunately, since the kernel only recognizes the containing process (of the threads), *if one thread is blocked, every other threads of the same process are also blocked* because the containing process is blocked.

Implementing Threads in User Space



A user-level threads package

Implementing Threads in User Space (Cont.)

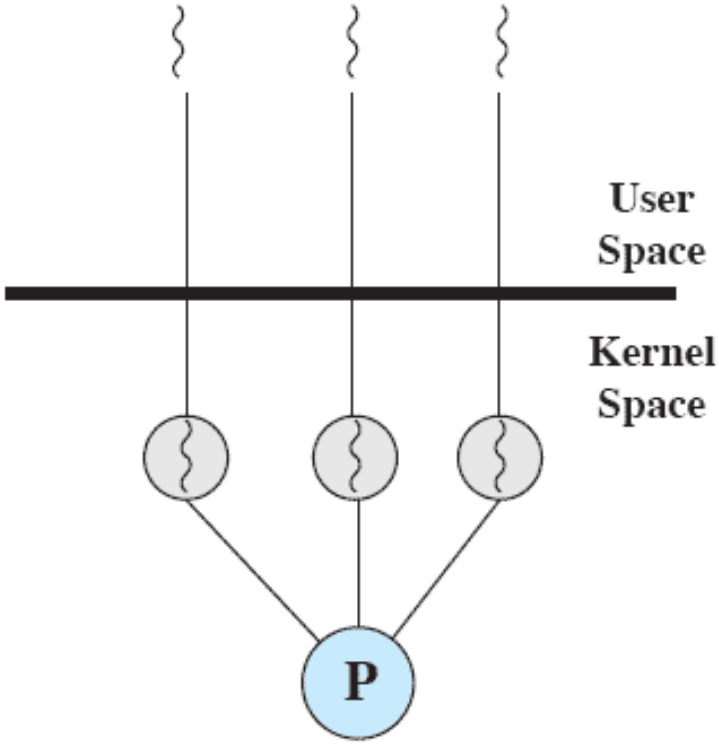
- A user-level threads can be implemented on an operating system that does not support threads
  - When threads are managed in user space, each process needs its own private thread table. It keeps track the per-thread properties (program counter, stack pointer, registers, state etc).
  - The procedure that saves the thread's state and the scheduler are just local procedures, so invoking them is much more efficient than making a kernel call.
- User-level threads allow each process to have its own customized scheduling algorithm.

Kernel Threads

- Supported by the Kernel
  - Kernel maintains context information for the process and the threads
  - No thread management done by application
- Examples

  - Windows 95/98/NT/2000

  - Solaris, Linux



(b) Pure kernel-level

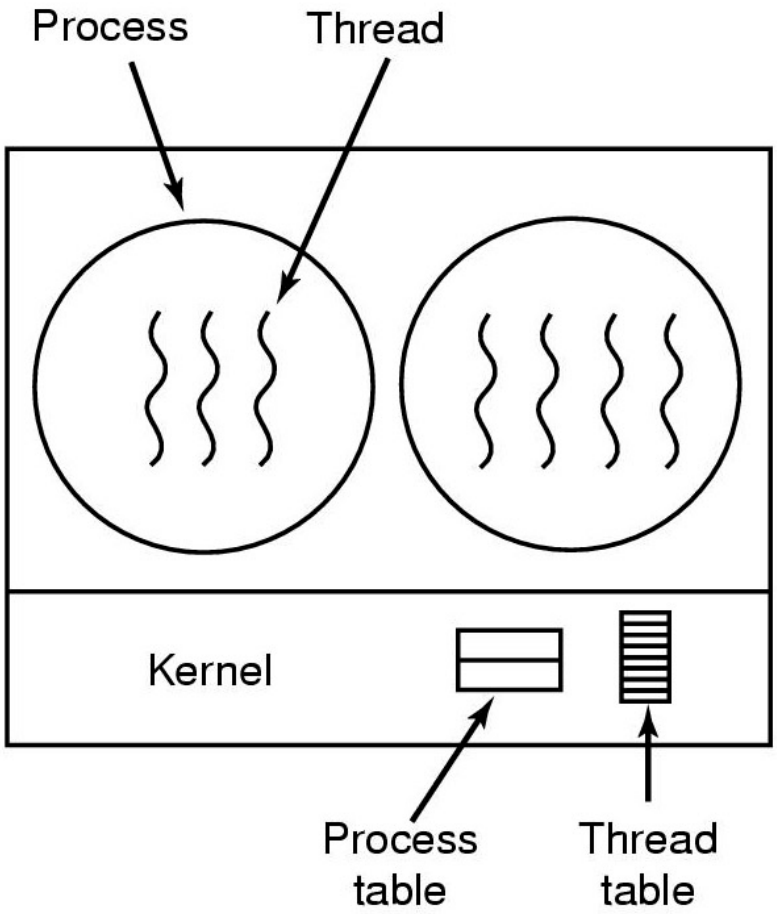
Kernel Threads (Cont.)

- Kernel threads are directly supported by the kernel. The kernel does thread creation, termination, joining, and scheduling in kernel space.
- Kernel threads are usually slower than the user threads.
- However, *blocking one thread will not cause other threads of the same process to block*. The kernel simply runs other threads.
- In a multiprocessor environment, the kernel can schedule threads on different processors

Kernel Threads (Cont.)

- Advantage
  - The kernel can simultaneously schedule multiple threads from the same process on multiple processors.
  - If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- Disadvantage
  - The transfer of control from one thread to another within the same process requires a mode switch to the kernel

Implementing Threads in the Kernel



A threads package managed by the kernel

Implementing Threads in the Kernel (Cont.)

- The kernel has a thread table that keeps track of all the threads in the system.
- All calls that might block a thread are implemented as system calls.
- When a thread blocks, the kernel can run either another thread from the same process, or a thread form a different process.

Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Windows Threads API
- Pthreads

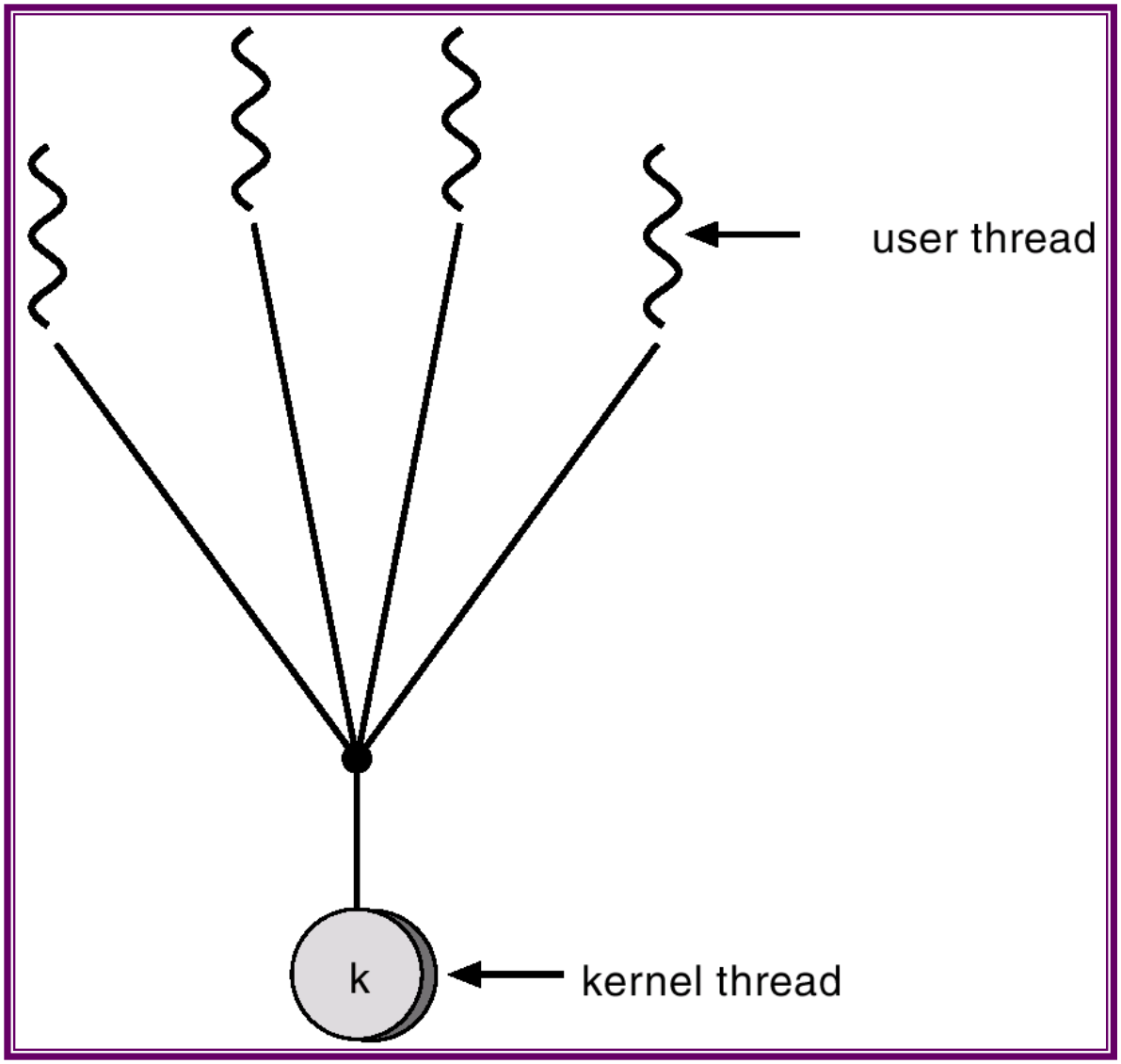
Multithreading Models

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

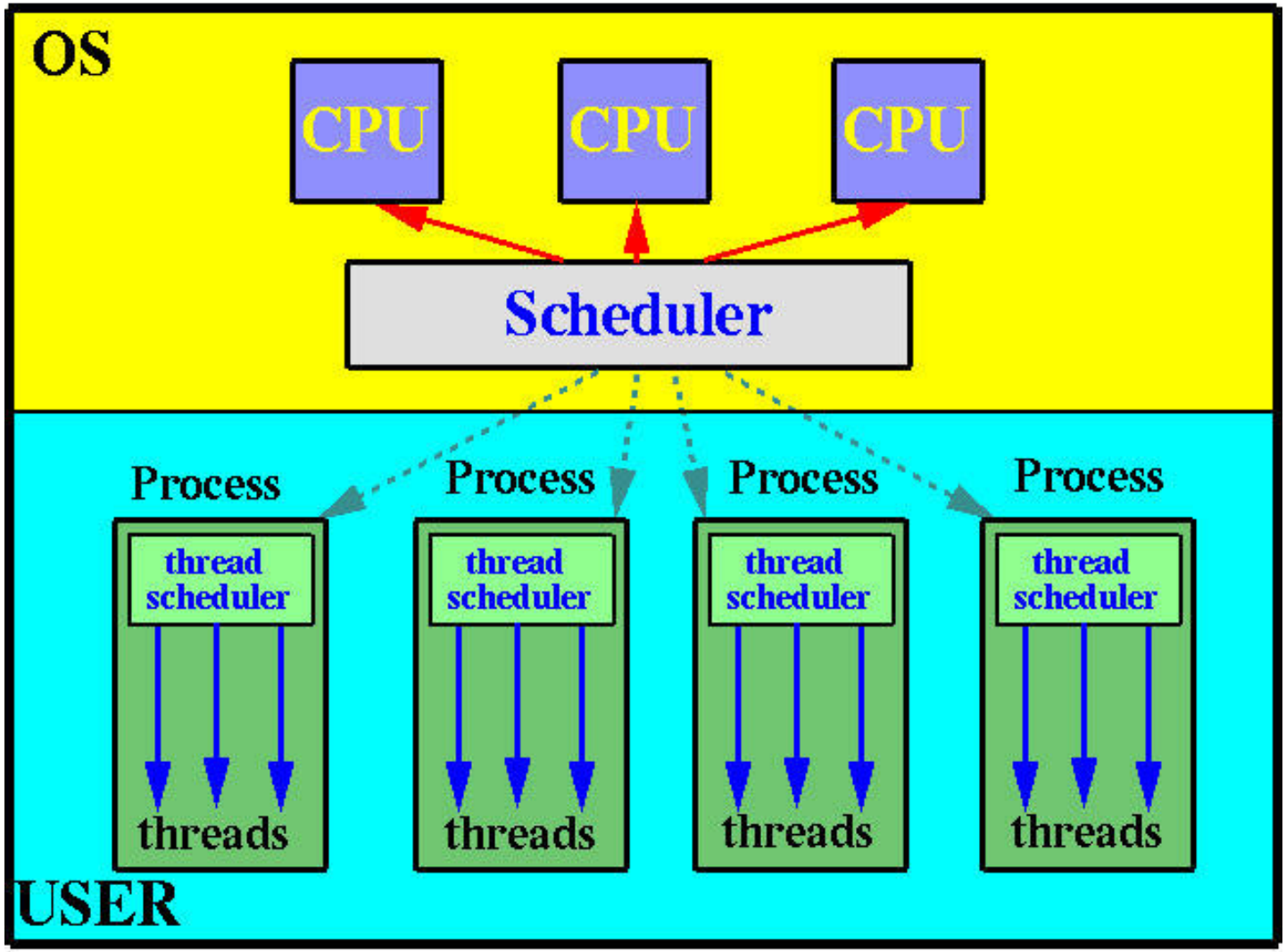
Many-to-One

- Many user-level threads mapped to single kernel thread.
- Used on systems that do not support kernel threads.

Many-to-One Model



Many-to-One Model (Cont.)

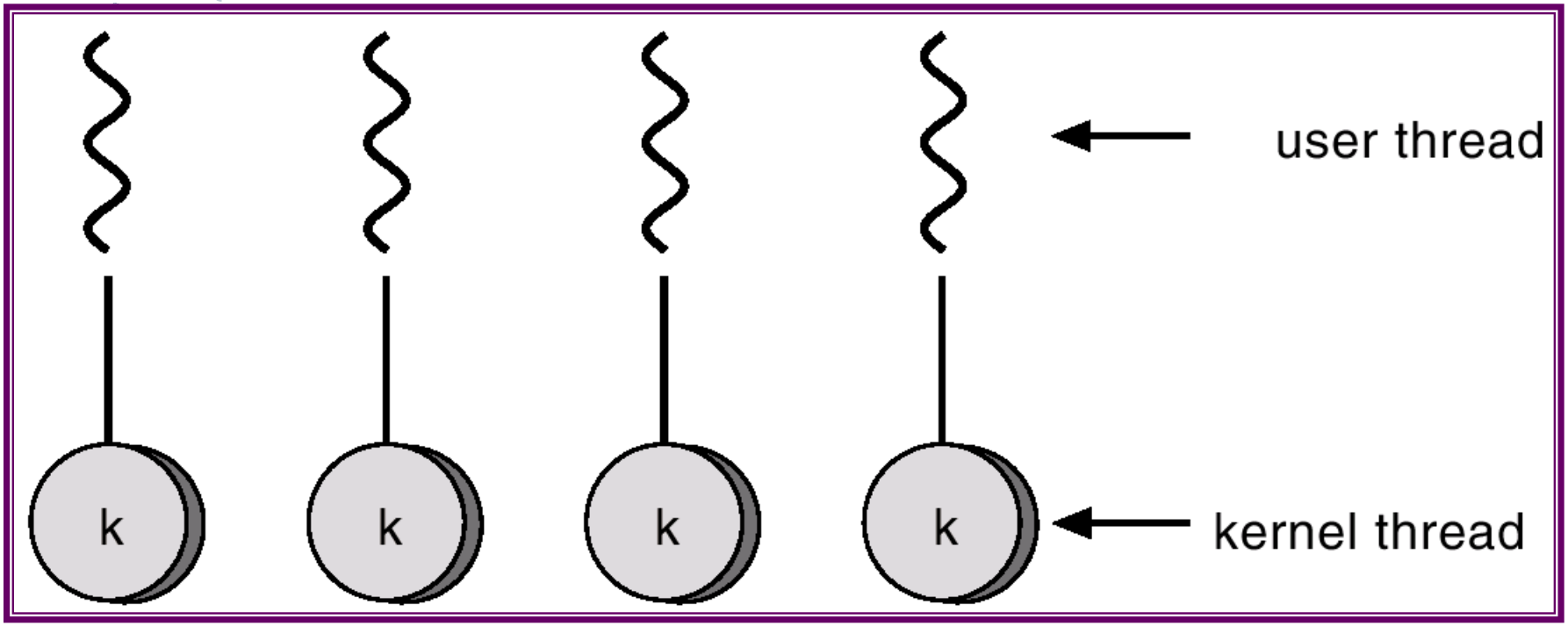


One-to-One

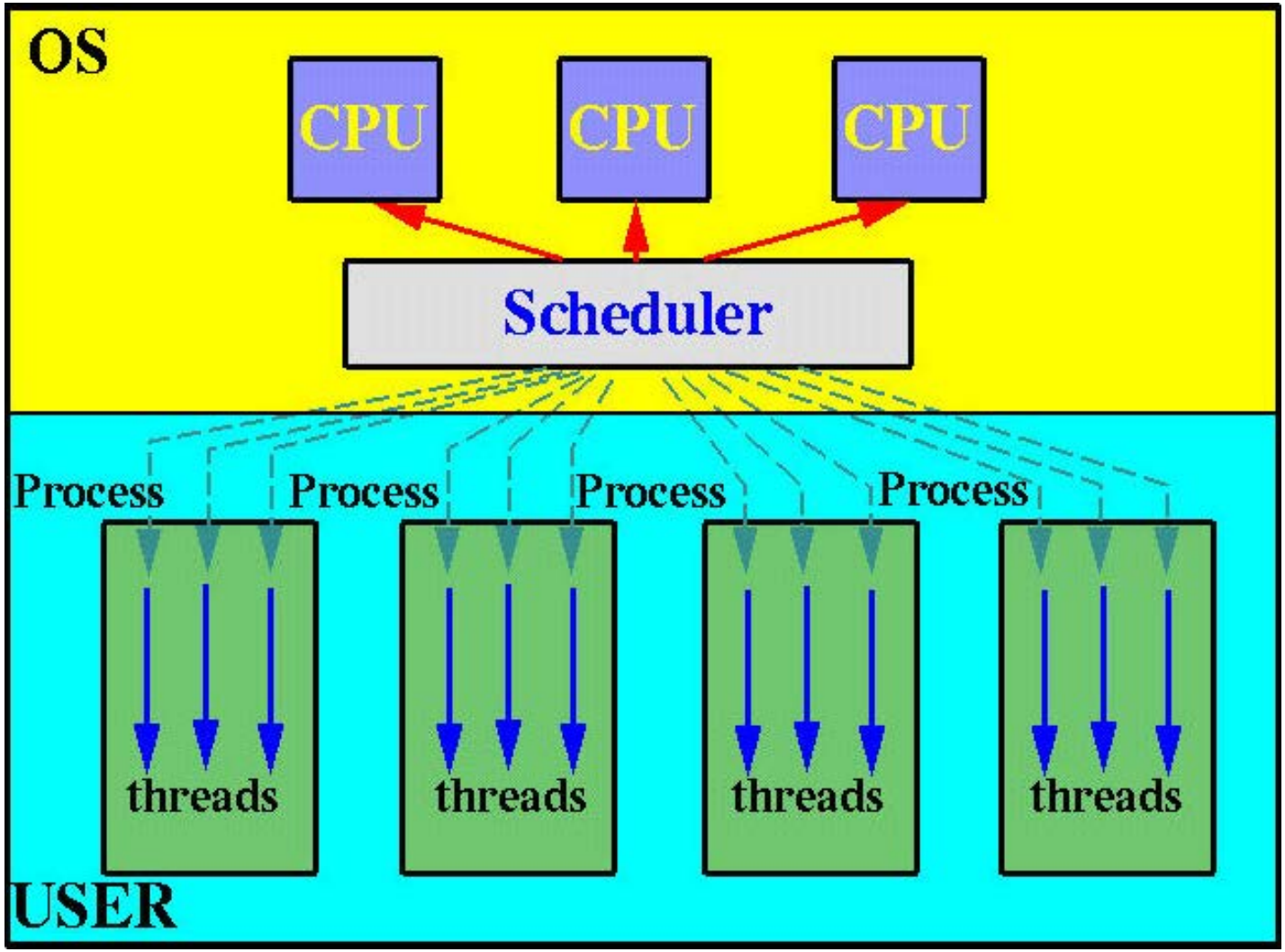
- Each user-level thread maps to kernel thread.

- Examples
  - Windows 95/98/NT/2000
  - OS/2

One-to-one Model



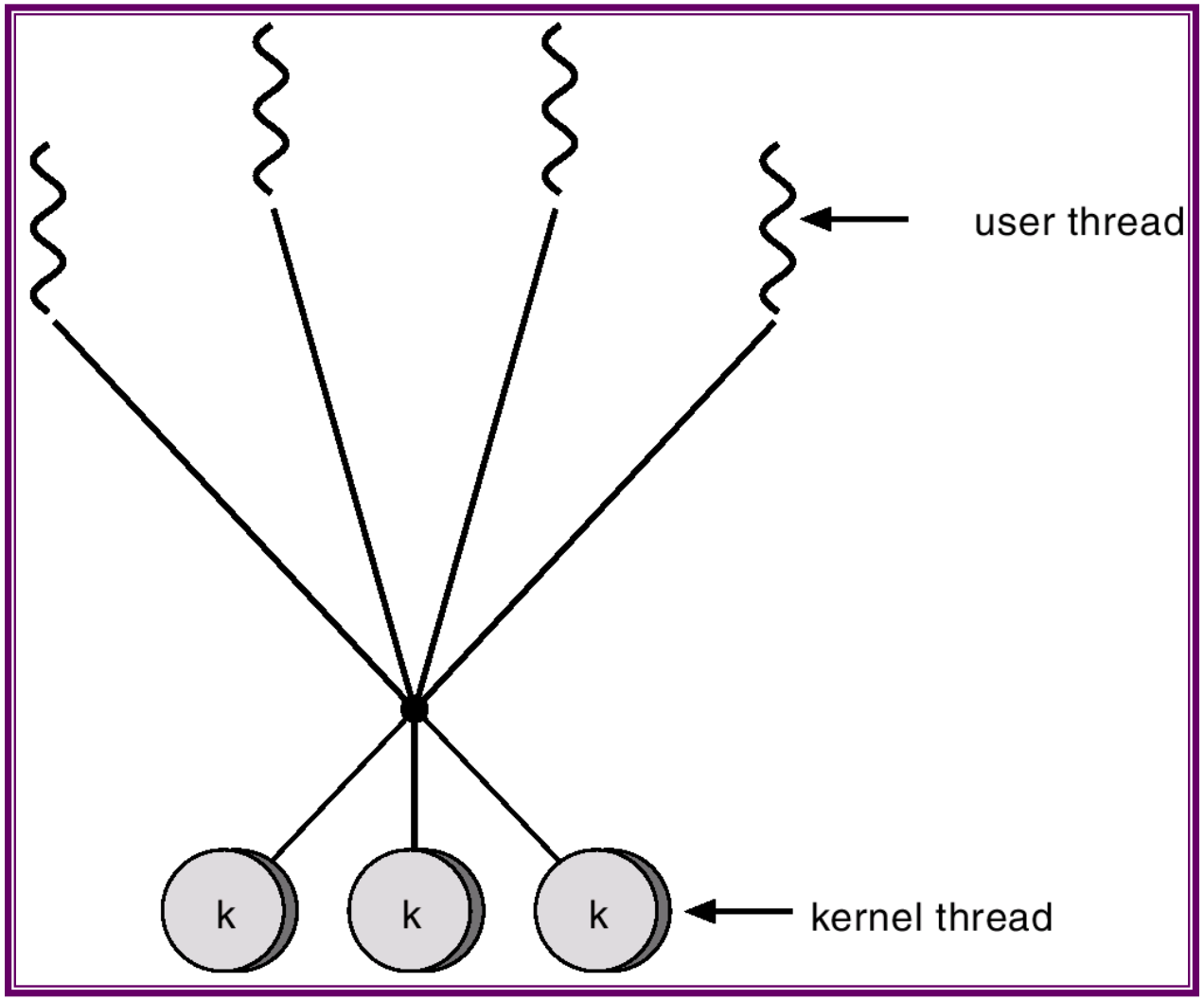
One-to-one Model (Cont.)



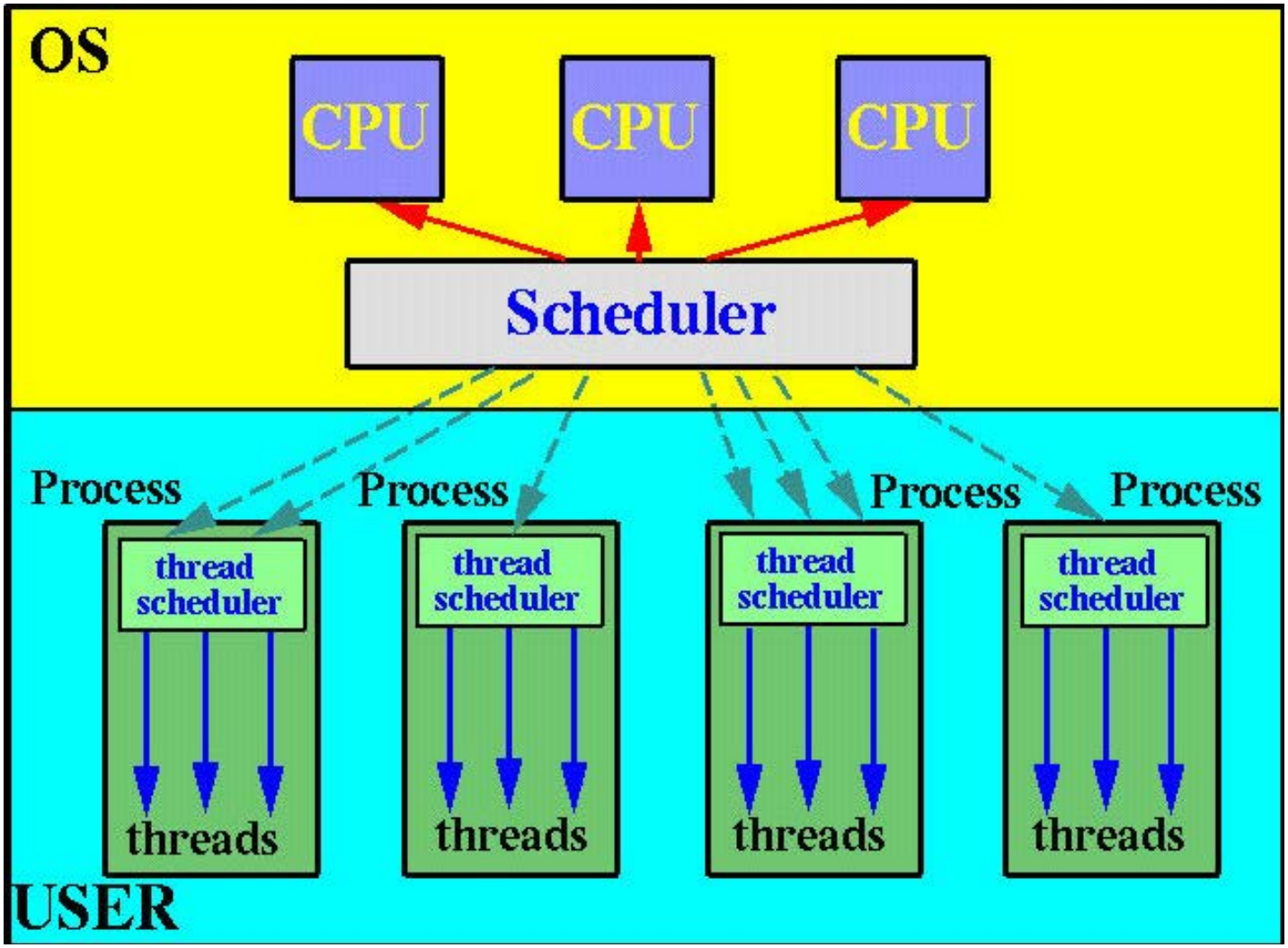
Many-to-Many Model

- 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



Many-to-Many Model (Cont.)



Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Windows Threads API
- Pthreads

Threading Issues

- Semantics of fork() and exec() system calls.
- Thread cancellation.
- Signal handling
- Thread pools
- Thread specific data
- Scheduler Activations

Semantics of fork() and exec()

- Does **fork()** duplicate only the calling thread or all threads?

Thread Cancellation

- Terminating a thread before it has finished
- Two general approaches:
  - **Asynchronous cancellation** terminates the target thread immediately
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - ✓The point a thread can terminate itself is a ***cancellation point*.**

Thread Cancellation (Cont.)

- With asynchronous cancellation, if the target thread owns some system-wide resources, the system may not be able to reclaim all recourses
- With deferred cancellation, the target thread determines the time to terminate itself. Reclaiming resources is not a problem.
- Most systems implement asynchronous cancellation for processes (e.g., use the kill system call) and threads.
- Pthread supports deferred cancellation.

Signal Handling

- Signals are used in UNIX systems to notify a process that a particular event has occurred
- All signals follow the same pattern:
  1. Signal is generated by particular event
  2. Signal is delivered to a process
  3. Signal is handled
- A **signal handler** is used to process signals

Signal Handling (Cont.)

- Options:
  - Deliver the signal to the thread to which the signal applies
  - Deliver the signal to every thread in the process
  - Deliver the signal to certain threads in the process
  - Assign a specific thread to receive all signals for the process
- Demo: signal handler (SIGINT, kill -2 or Ctrl + c)

Thread Pools

- Create a number of threads in a pool where they await work
- Advantages:
  - Usually slightly faster to service a request with an existing thread than create a new thread
  - Allows the number of threads in the application(s) to be bound to the size of the pool

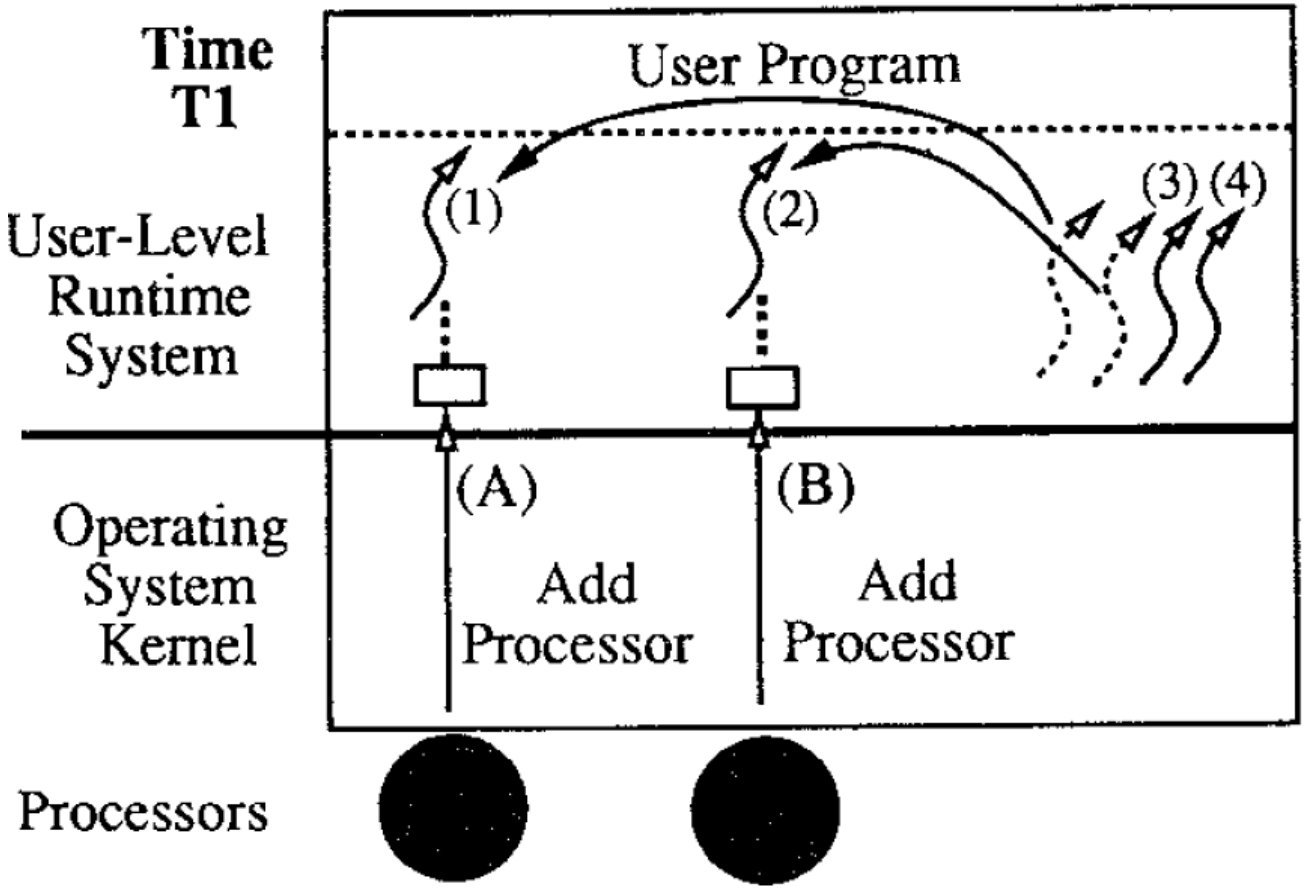
Thread Specific Data

- Allows each thread to have its own copy of data
- Useful when you do not have control over the thread creation process (i.e., when using a thread pool)

Scheduler Activations

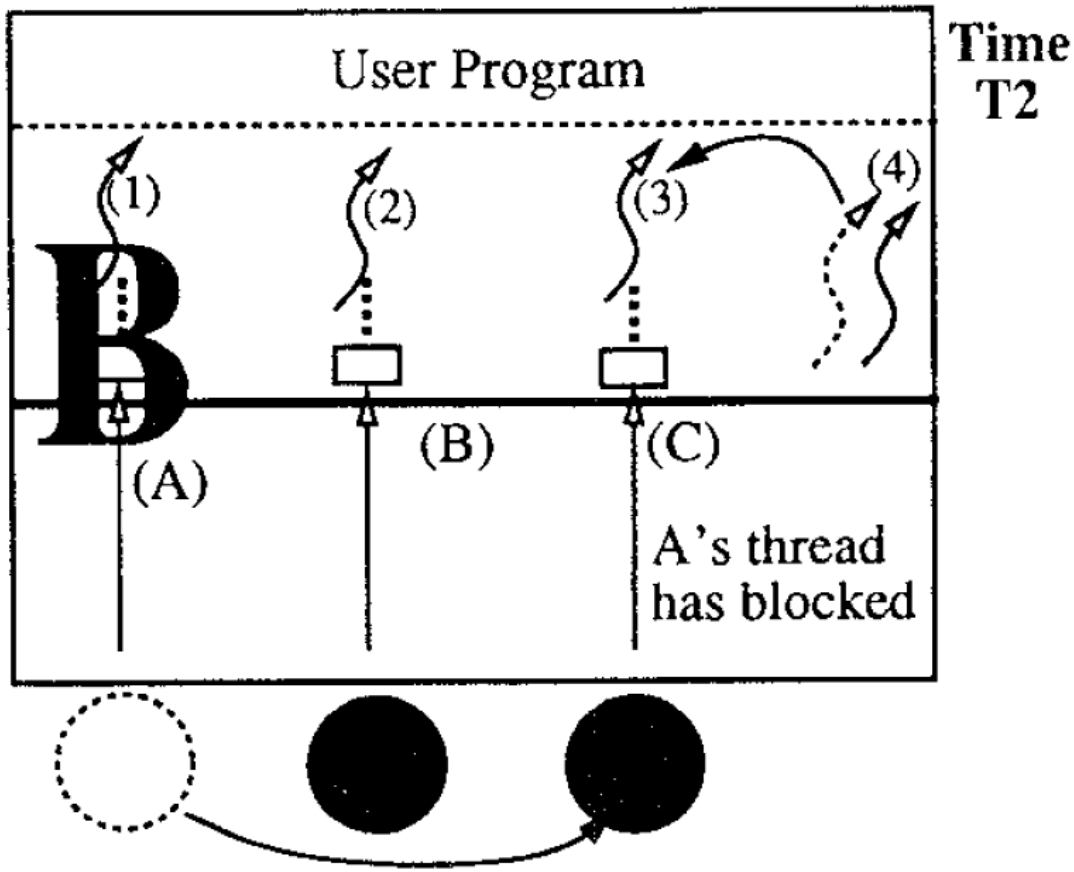
- Both M:M and Two-level models require communication to maintain the appropriate number of kernel threads allocated to the application
- Scheduler activations provide **upcalls** - a communication mechanism from the kernel to the thread library
- This communication allows an application to maintain the correct number kernel threads

Example (T1)



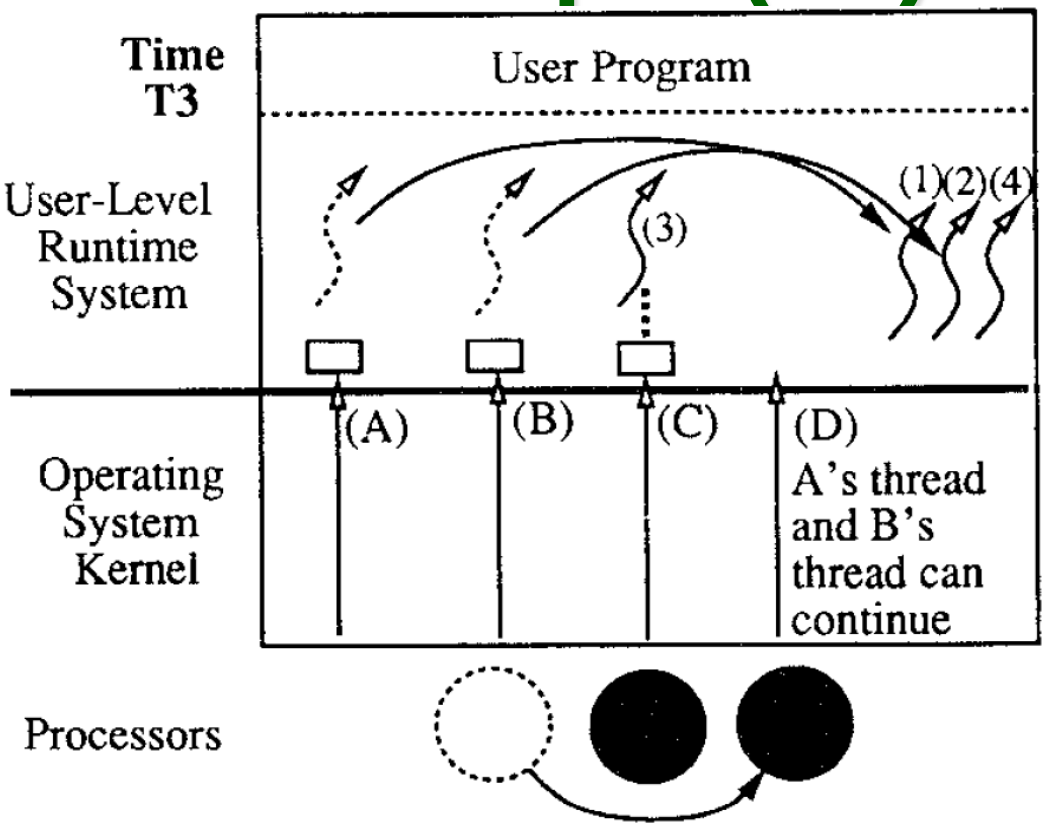
At time Tl, the kernel allocates the application two processors. On each processor, the kernel upcalls to user-level code that removes a thread from the ready list and starts running it.

Example (T2)



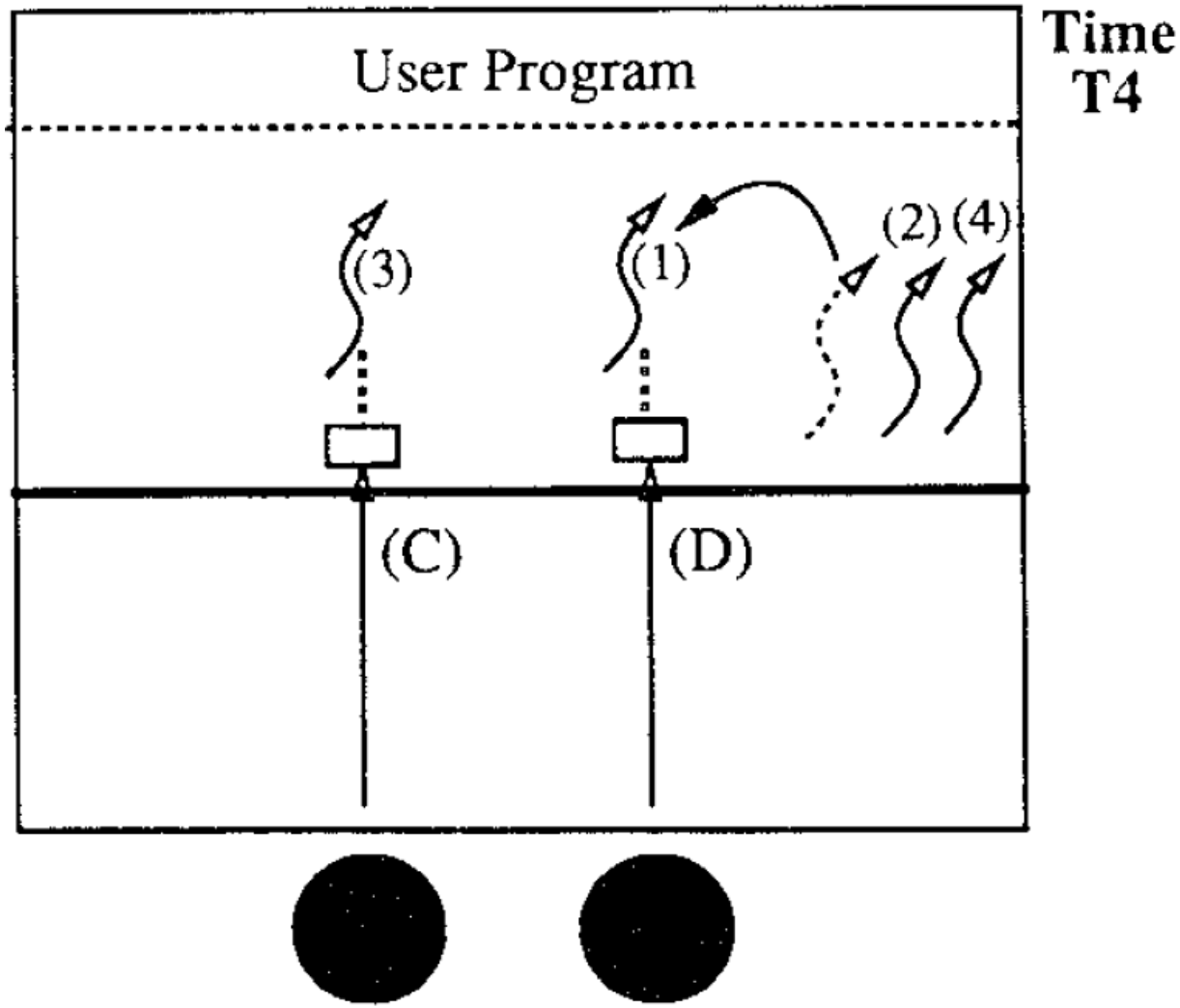
At time T2, one of the user-level threads (thread 1) blocks in the kernel. To notify the user level of this event, the kernel takes the processor that had been running thread 1 and performs an upcall in the context of a fresh scheduler activation. The user-level thread scheduler can then use the processor to take another thread off the ready list and start running it.

Example (T3)



At time T3, the I/O completes. Again, the kernel must notify the user-level thread system of the event, but this notification requires a processor. The kernel preempts one of the processors running in the address space and uses it to do the upcall. (If there are no processors assigned to the address space when the I/O completes, the upcall must wait until the kernel allocates one). This upcall notifies the user level of two things: the I/0 completion and the preemption. The upcall invokes code in the user-level thread system that (1) puts the thread that had been blocked on the ready list and (2) puts the thread that was preempted on the ready list. At this point, scheduler activations A and B can be discarded.

Example (T4)



Finally, at time T4, the upcall takes a thread off the ready list and starts running it.

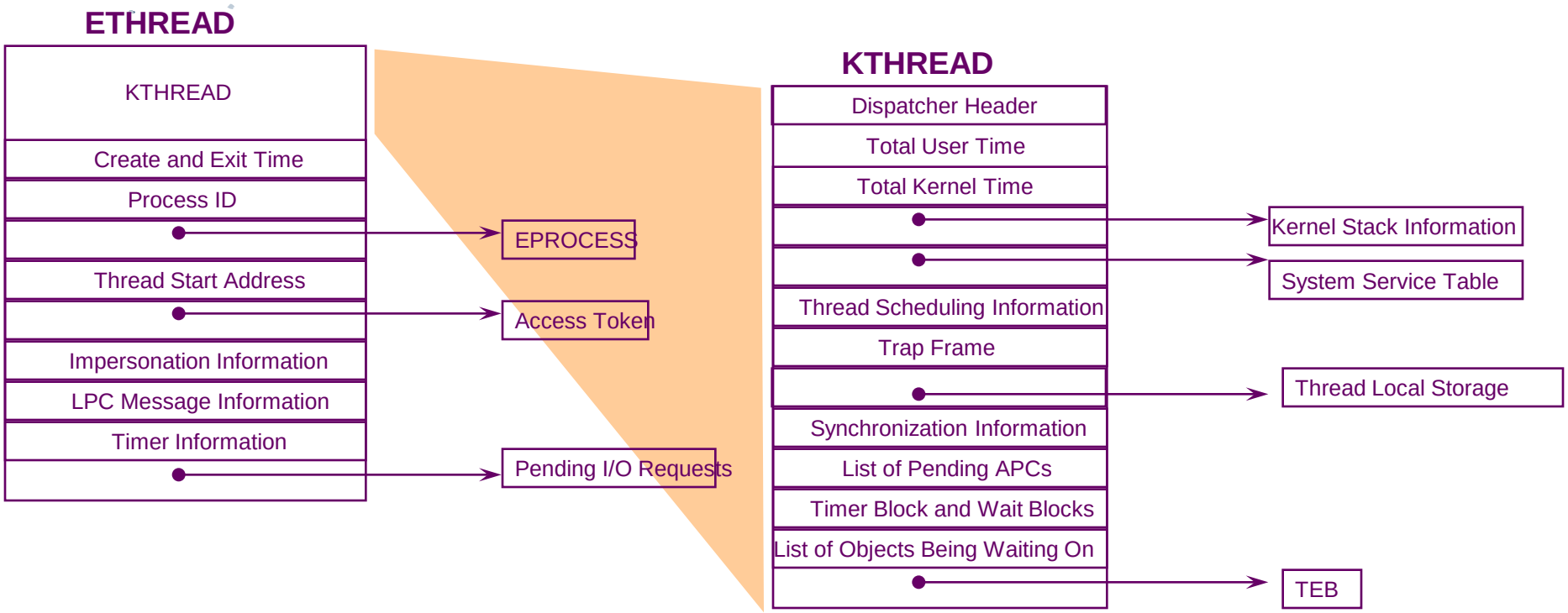
Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Windows Threads API
- Pthreads

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

Thread Block



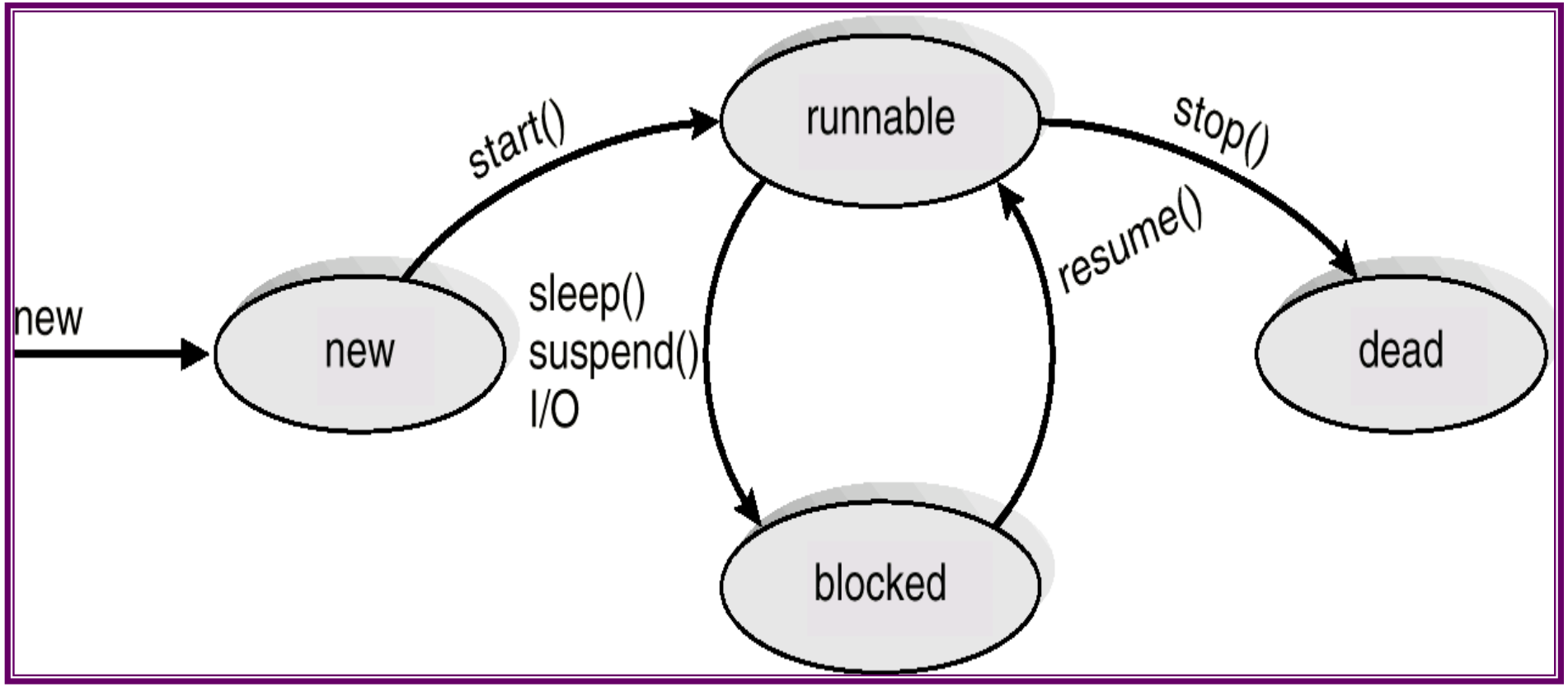
Linux Threads

- 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)

Java Threads

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

Java Thread States



Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Windows Thread APIs
- Pthreads

Windows Thread APIs

- CreateThread
- GetCurrentThreadId - returns global ID
- GetCurrentThread - returns handle
- SuspendThread/ResumeThread
- ExitThread
- TerminateThread
- GetExitCodeThread
- GetThreadTimes

Windows API Thread Creation

```
HANDLE CreateThread (
    LPSECURITY_ATTRIBUTES lpsa,
    DWORD cbStack,
    LPTHREAD_START_ROUTINE lpStartAddr,
    LPVOID lpvThreadParm,
    DWORD fdwCreate,
    LPDWORD lpIDThread)
```

cbStack == 0: thread‘s stack size defaults to primary thread‘s size

- lpstartAddr points to function declared as

  DWORD WINAPI ThreadFunc(LPVOID)
- lpvThreadParm is 32-bit argument
- LPIDThread points to DWORD that receives thread ID non-NULL pointer !

Windows API Thread Termination

```
VOID ExitThread( DWORD devExitCode )
```

- When the last thread in a process terminates, the process itself terminates

```
BOOL GetExitCodeThread (
      HANDLE hThread, LPDWORD lpdwExitCode)
```

- Returns exit code or STILL_ACTIVE

Suspending and Resuming Threads

- Each thread has suspend count
- Can only execute if suspend count == 0
- Thread can be created in suspended state

```
DWORD ResumeThread (HANDLE hThread)
DWORD SuspendThread(HANDLE hThread)
```

- Both functions return suspend count or 0xFFFFFFFF on failure

Example: Thread Creation

```
#include <stdio.h>
#include <windows.h>

DWORD WINAPI helloFunc(LPVOID arg ) {
        printf("Hello Thread\n");
        return 0;
}

main() {
        HANDLE hThread =
                CreateThread(NULL, 0, helloFunc,
                    NULL, 0, NULL );
        }
```

What's Wrong?

Example Explained

- Main thread is process
- When process goes, all threads go
- Need some methods of waiting for a thread to finish

Waiting for Windows* Thread

```
#include <stdio.h>
#include <windows.h>
BOOL thrdDone = FALSE;
DWORD WINAPI helloFunc(LPVOID arg ) {
        printf("Hello Thread\n");
        return 0;
}

main() {
        HANDLE hThrea
                Create
                        NULL, 0, NULL );
        }
```

thrdDone = TRUE;

Not a good idea!

while (!thrdDone);

Waiting for a Thread

Wait for one object (thread)

```
DWORD WaitForSingleObject(
        HANDLE hHandle,
        DWORD dwMilliseconds );
```

Calling thread waits (blocks) until

- Time expires
  - Return code used to indicate this
- Thread exits (handle is signaled)
  - Use `INFINITE` to wait until thread termination

Does not use CPU cycles

Waiting for Many Threads

Wait for up to 64 objects (threads)

```
DWORD WaitForMultipleObjects(
        DWORD nCount,
        CONST HANDLE *lpHandles, // array
        BOOL fWaitAll, // wait for one or all
        DWORD dwMilliseconds)
```

Wait for all: `fWaitAll==TRUE`

Wait for any: `fWaitAll==FALSE`

- Return value is first array index found

Notes on WaitFor* Functions

- Handle as parameter
- Used for different types of objects
- Kernel objects have two states
  - Signaled
  - Non-signaled
- Behavior is defined by object referred to by handle
  - Thread: signaled means terminated

Example: Waiting for multiple threads

```
#include <stdio.h>
#include <windows.h>
const int numThreads = 4;

DWORD WINAPI helloFunc(LPVOID arg ) {
  printf("Hello Thread\n");
  return 0; }

main() {
  HANDLE hThread[numThreads];
  for (int i = 0; i < numThreads; i++)
    hThread[i] =
      CreateThread(NULL, 0, helloFunc, NULL, 0, NULL );
  WaitForMultipleObjects(numThreads, hThread,
                                        TRUE, INFINITE);

}
```

Example: HelloThreads

- Modify the previous example code to print out
  - appropriate “Hello Thread”message
  - Unique thread number
    - ✓use for-loop variable of CreateThread loop
- Sample output:

```
Hello from Thread #0

Hello from Thread #1

Hello from Thread #2

Hello from Thread #3
```

What's Wrong?

```
DWORD WINAPI threadFunc(LPVOID pArg) {
  int* p = (int*)pArg;
  int myNum = *p;
  printf( "Thread number %d\n", myNum);
}
. . .
// from main():
for (int i = 0; i < numThreads; i++) {
  hThread[i] =
      CreateThread(NULL, 0, threadFunc, &i, 0, NULL);
}
```

What is printed for myNum?

Hello Threads Timeline

Time	*main*	*Thread 0*	*Thread 1*
T_0	i = 0	---	----
T_1	create(&i)	---	---
T_2	i++ (i == 1)	launch	---
T_3	create(&i)	p = pArg	---
T_4	i++ (i == 2)	myNum = *p myNum = 2	launch
T_5	wait	print(2)	p = pArg
T_6	wait	exit	myNum = *p myNum = 2

Race Conditions

- Concurrent access of same variable by multiple threads
  - Read/Write conflict
  - Write/Write conflict
- Most common error in concurrent programs
- May not be apparent at all times
- How to avoid data races?
  - Local storage
  - Control shared access with critical regions

Hello Thread: Local Storage solution

```
DWORD WINAPI threadFunc(LPVOID pArg)
{
  int myNum = *((int*)pArg);
  printf( "Thread number %d\n", myNum);
}
. . .

// from main():
for (int i = 0; i < numThreads; i++) {
  tNum[i] = i;
  hThread[i] =
     CreateThread(NULL, 0, threadFunc, &tNum[i],
                  0, NULL);
}
```

Chapter 4: Threads

- Overview
- Multithreading Models
- Threading Issues
- Windows XP Threads
- Linux Threads
- Java Threads
- Windows Threads API
- Pthreads

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.

pthread_create

int pthread_create(tid, attr, function, arg);

- pthread_t *tid
  - handle of created thread
- const pthread_attr_t *attr
  - attributes of thread to be created
- void *(*function) (void*)
  - function to be mapped to thread
- void *arg
  - single argument to function

pthread_create explained

spawn a thread running the function

thread handle returned via pthread_t structure

- specify *NULL* to use default attributes

single argument sent to function

- If no argument to function, specify *NULL*

check error codes!

EAGAIN – insufficient resources to create thread

EINVAL – invalid attribute

Threads states

- pthread threads have two states
  - joinable and detached
- threads are joinable by default
  - Resources are kept until *pthread_join*
  - can be reset with attribute or API call
- detached thread can not be joined
  - resources can be reclaimed at termination
  - cannot reset to be *joinable*

Waiting for a thread

int pthread_join(tid, val_ptr);

- pthread_t *tid
  - handle of joinable thread
- void **val_ptr
  - exit value returned by joined thread

pthread_join explained

calling thread waits for the thread with handle tid to terminate

- only one thread can be joined
- thread must be joinable

exit value is returned from joined thread

- Type returned is (void *)
- use NULL if no return value expected

ESRCH –thread not found

EINVAL – thread not joinable

Example: Multiple threads

```
#include <stdio.h>
#include <pthread.h>
const int numThreads = 4;

void *helloFunc(void * pArg)
{   printf("Hello Thread\n"); }

main()
{   pthread_t hThread[numThreads];
    for (int i = 0; i < numThreads; i++)
            pthread_create(&hThread[i], NULL, helloFunc,
    NULL);
    for (int i = 0; i < numThreads; i++)
          pthread_join(hThread[i], NULL);
    return 0;
}
```

Demo with pthreads

Thread Termination

- void pthread_exit(void *status);
  - terminate the current thread
- int pthread_cancel(pthread_t thread);
  - the thread may:
    - ✓ ignore the request
    - ✓ terminated immediately (Asynchronous cancellation)
    - ✓ deferred terminated (Deferred cancellation)
- int pthread_kill(pthread_t thread, int sig);

Deferred Cancellation

- int pthread_setcancelstate(int state, int *oldstate);
  - PTHREAD_CANCEL_ENABLE
  - PTHREAD_CANCEL_DISABLE
- int pthread_setcanceltype(int type, int *oldtype);
  - PTHREAD_CANCEL_ASYNCHROUS
  - PTHREAD_CANCEL_DEFERRED
- void pthread_testcancel(void);