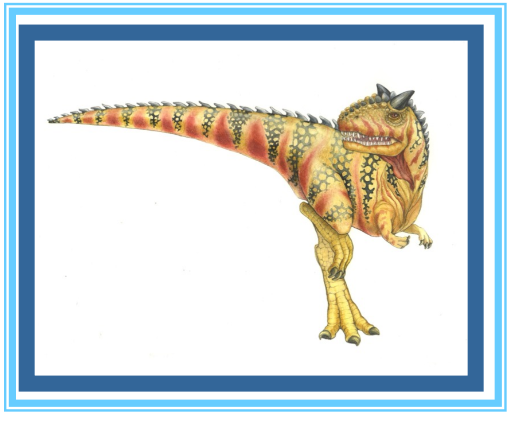
Chapter 4: Threads



Chapter 4: Threads

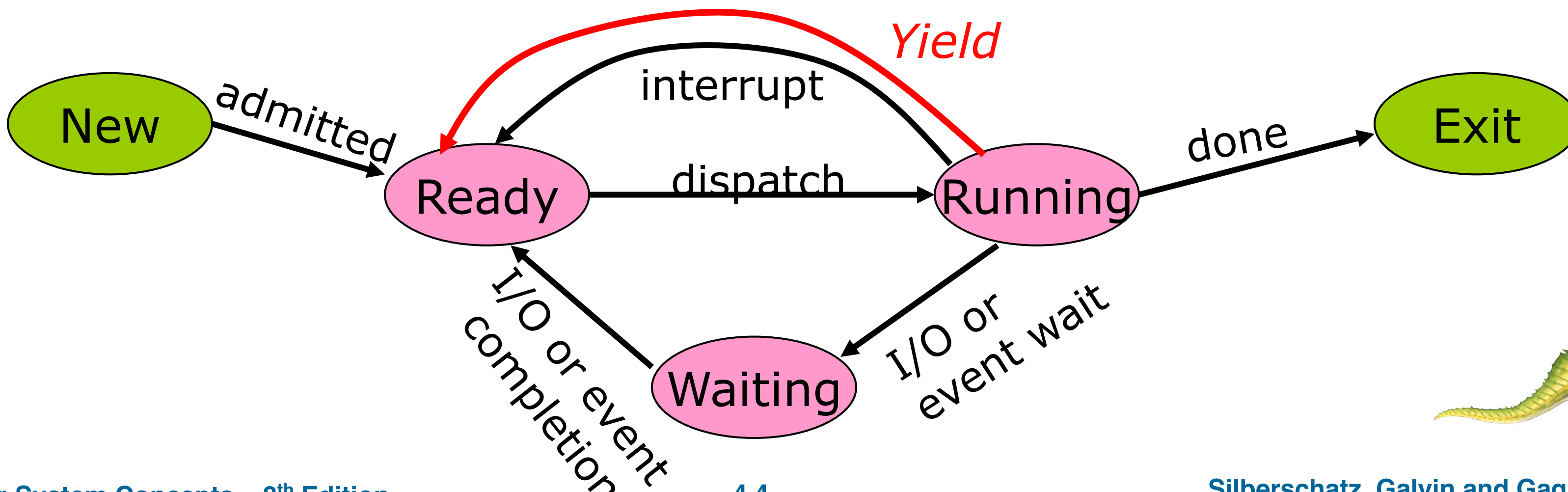
- Overview
- Multithreading Models
- Thread Libraries
- Examples(Java Threads, OSs Threads)
- Threading Issues

Objectives

- To introduce the notion of a thread — a fundamental unit of CPU utilization that forms the basis of multithreaded computer systems
- To discuss the APIs for the Pthreads, Win32, and Java thread libraries
- To examine issues related to multithreaded programming

Review: Processes

- The unit of execution and scheduling
- A task created by the OS, running in a restricted virtual machine environment –a virtual CPU, virtual memory environment, interface to the OS via system calls
- Sequential, instruction-at-a-time execution of a program.
- Abstraction used for protection
  - Main Memory State (contents of Address Space)
- Multiprogramming: overlap IO and CPU
- Context Switches are expensive



Review: Processes

- How does parent know child process has finished if exec does not return?

```
main(int argc, char **argv)
{
   char *myName = argv[1];
   char *progName = argv[2];

   int cpid = fork();
   if (cpid == 0) {
      printf("The child of %s is %d\n", myName, getpid());
      execlp("/bin/ls","ls",NULL);
      printf("OH NO. THEY LIED TO ME!!!\n");

 } else {
      printf("My child is %d\n", cpid);
      wait(cpid);
      exit(0);
   }
}
```

Review: Cooperating Processes

- Processes can be independent or work cooperatively
- Cooperating processes can be used:
  - to gain speedup by overlapping activities or working in parallel
  - to better structure an application as set of cooperating processes
  - to share information between jobs
- Sometimes processes are structured as a pipeline
  - each produces work for the next stage that consumes it

```
main()
  read_data()
  for(all data)
        compute();
        write_data();
  endfor
```

```
main()
    read_data()
    for(all data)
          compute();
          CreateProcess(write_data());
    endfor
```

Case for Parallelism

Consider the following code fragment on a dual core CPU

```
for(k = 0; k < n; k++)
    a[k] = b[k] * c[k] + d[k] * e[k];
```

Instead:

```
CreateProcess(fn, 0, n/2);
CreateProcess(fn, n/2, n);
fn(l, m)
    for(k = l; k < m; k++)
        a[k] = b[k] * c[k] + d[k] * e[k];
```

Case for Parallelism

- Consider a Web server:

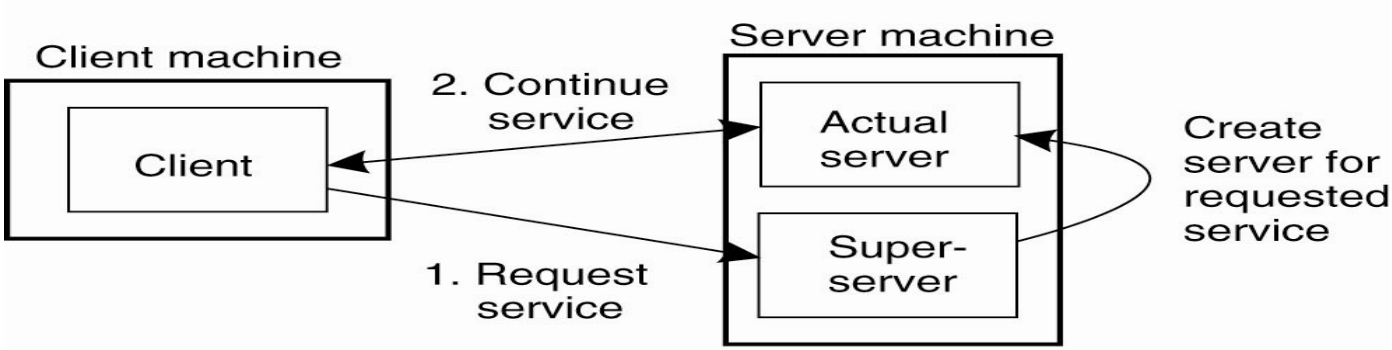
create a number of process, and for each process do

- get network message from client
- get URL data from disk
- compose response
- send response

- Server connections are fast, but client connections may not be (grandma's modem connection)
  - Takes server a loooong time to feed the response to grandma
  - While it's doing that it can't service any more requests（进程占用资源）

Parallel Programs

- To build parallel programs, such as:
  - Parallel execution on a multiprocessor
  - Web server to handle multiple simultaneous web requests
- We will need to:
  - Create several processes that can execute in parallel
  - Cause *each* to map to the *same* address space
    - because they're part of the same computation
  - Give each its starting address and initial parameters
  - The OS will then schedule these processes in parallel



(b)

Processes Overheads

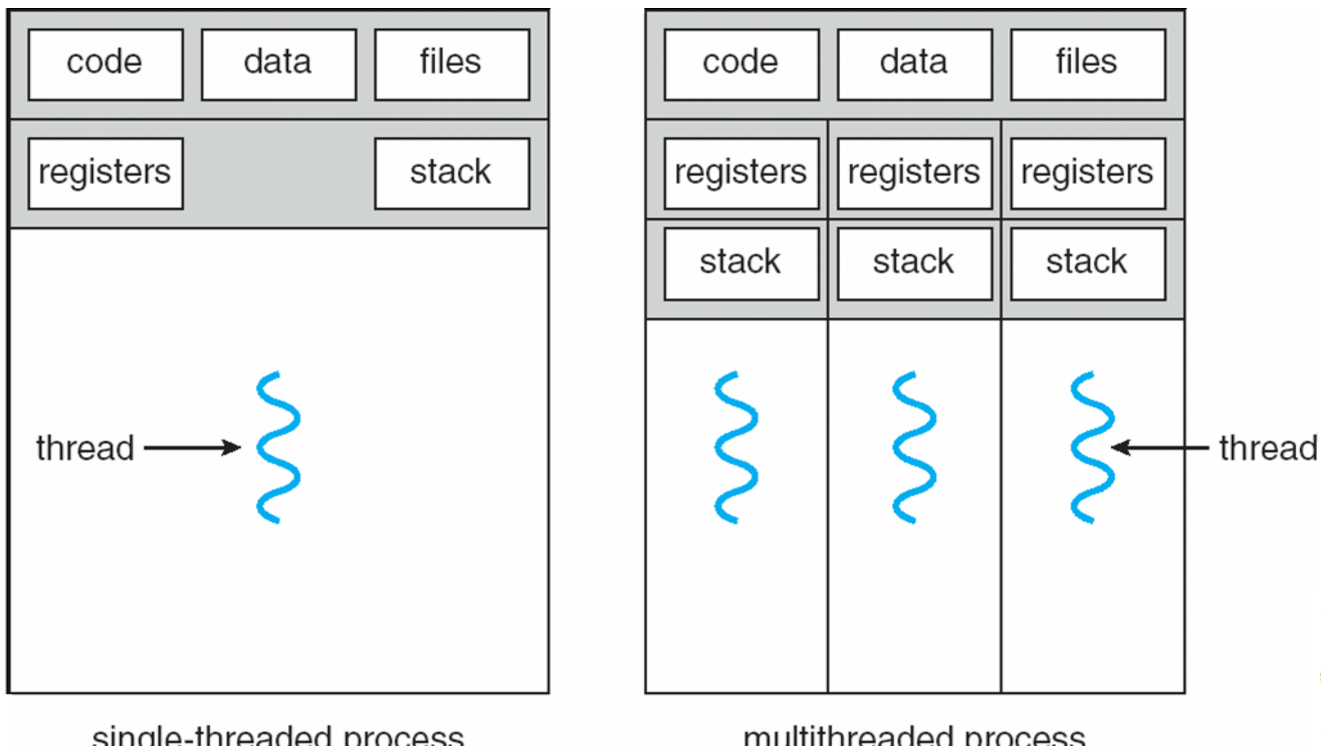
- A full process includes numerous things:
  - an address space (defining all the code and data pages)
  - OS resources and accounting information
  - a “thread of control”,
    - defines where the process is currently executing
    - That is the PC and registers
- Creating a new process is costly
  - all of the structures (e.g., page tables) that must be allocated
- Communicating between processes is costly
  - most communication goes through the OS

Need "Lightweight" Processes

- What's similar in these processes?
  - They all share the same code and data (address space)
  - They all share the same privileges
  - They share almost everything in the process
- What don't they share?
  - Each has its own PC, registers, and stack pointer
- Idea: why don't we separate the idea of process (address space, accounting, etc.) from that of the minimal "thread of control" (PC, SP, registers)?按执行线索分离

Threads and Processes

- Most operating systems therefore support two entities:
  - the process: which defines the address space and general process attributes
  - the thread: which defines a sequential execution stream within a process
- A thread is bound to a single process: For each process, however, there may be many threads.
- Threads are the unit of scheduling, Processes are *containers* in which threads execute



Threads vs. Processes

- A thread has no data segment or heap
- A thread cannot live on its own, it must live within a process
- *There can be more than one thread in a process, the first thread calls main & has the process's stack*
- Inexpensive creation
- Inexpensive context switching
- If a thread dies, its stack is reclaimed

- A process has code/data/heap & other segments
- There must be at least one thread in a process
- Threads within a process share code/data/heap, share I/O, but each has its own stack & registers
- Expensive creation
- Expensive context switching
- If a process dies, its resources are reclaimed & all threads die

Separating Threads and Processes

- Makes it easier to support multithreaded applications
  - Different from multiprocessing, multiprogramming, multitasking
- Concurrency (multithreading) is useful for:
  - improving program structure
  - handling concurrent events (e.g., web requests)
  - building parallel programs
  - Resource sharing
  - Multiprocessor utilization

Benefits:

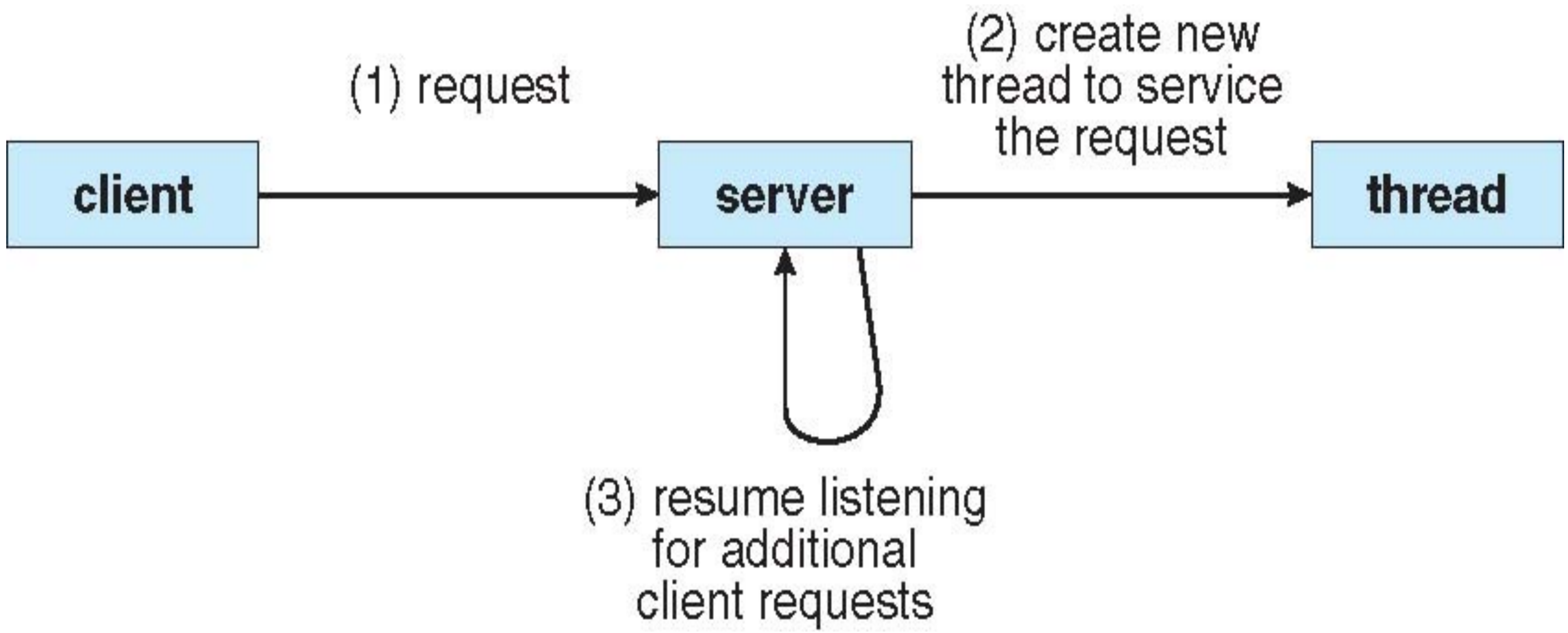
- Responsiveness
- Resource Sharing
- Economy
- Scalability

- Is multithreading useful even on a single processor?

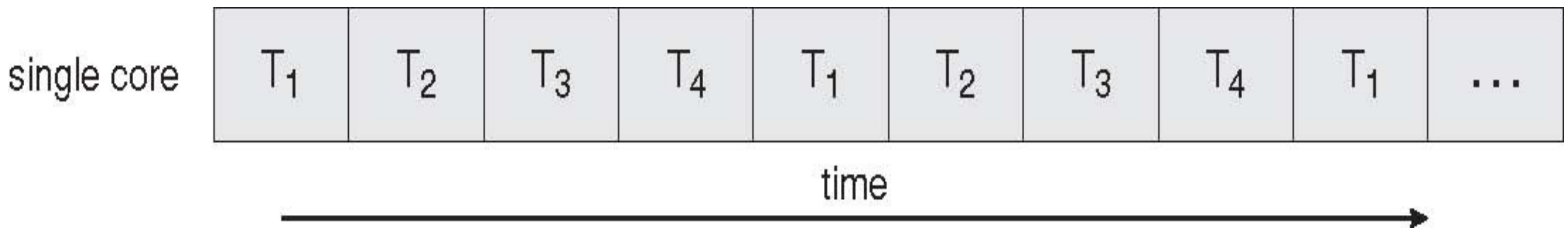
Multicore Programming

- Multicore systems putting pressure on programmers, challenges include
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**

Multithreaded Server Architecture

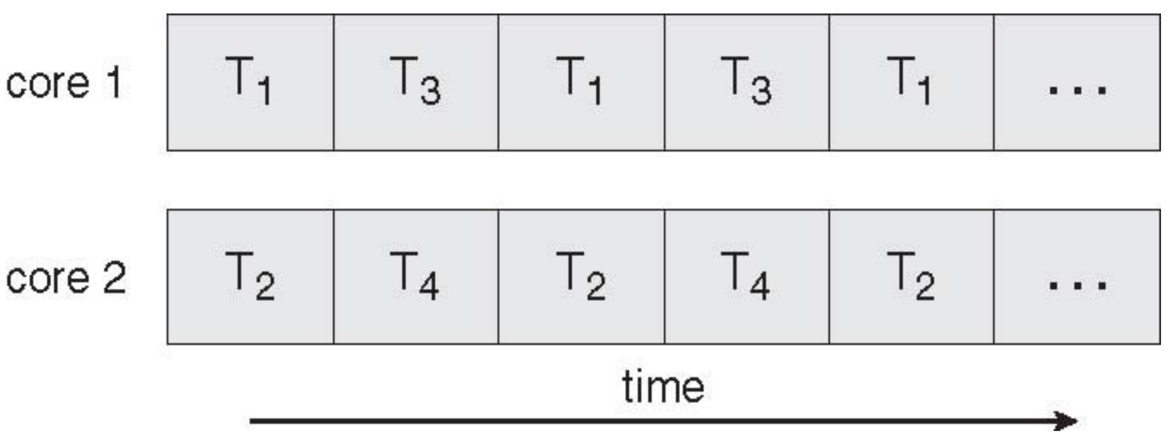


Concurrent Execution on a Single-core System



S=1/(1-a+a/n)

Parallel Execution on a Multicore System



Amdahl's Law（阿姆达尔定律）

- *S* is serial portion， A is parallel portion，*N* processing cores

  Speedup=1/(1-A+A/N) or

$$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$

- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As *N* approaches infinity, speedup approaches 1 / *S*

  Serial portion of an application has disproportionate effect on performance gained by adding additional cores

Implementation of thread

There are actually 2 level of threads:

- Kernel threads:
  - Supported and managed directly by the kernel.
- User threads:
  - Supported above the kernel, and without kernel knowledge.

Kernel Threads

- Also called Lightweight Processes (LWP)
- Kernel threads still suffer from performance problems
- Operations on kernel threads are slow because:
  - a thread operation still requires a system call
  - kernel threads may be overly general
    - to support needs of different users, languages, etc.
  - the kernel doesn't trust the user
    - there must be lots of checking on kernel calls

User-Level Threads

- For speed, implement threads at the user level
- A user-level thread is managed by the run-time system
  - user-level code that is linked with your program
- Each thread is represented simply by:
  - PC
  - Registers
  - Stack
  - Small control block
- All thread operations are at the user-level:
  - Creating a new thread
  - switching between threads
  - synchronizing between threads

User-Level Threads

- User-level threads
  - the thread scheduler is part of a library, outside the kernel
  - thread context switching and scheduling is done by the library
  - Can either use cooperative or pre-emptive threads
    - cooperative threads are implemented by:
      - CreateThread(), DestroyThread(), Yield(), Suspend(), etc.
    - pre-emptive threads are implemented with a timer (signal)
      - where the timer handler decides which thread to run next

Example User Thread Interface

- t = thread_fork(initial context):

  create a new thread of control;

- thread_stop():

  stop the calling thread, sometimes called thread_block;

- thread_start(t): start the named thread;

- thread_yield(): voluntarily give up the processor;

- thread_exit():

  terminate the calling thread, sometimes called thread_destroy;

Key Data Structures

your process address space

your program:

for i (1, 10, I++)
thread_fork(I);
....

your data (shared by
all your threads):

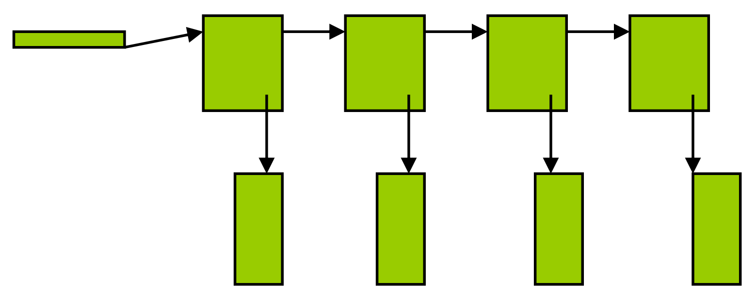
user-level thread code:

proc thread_fork()…

proc thread_block()…

proc thread_exit()…

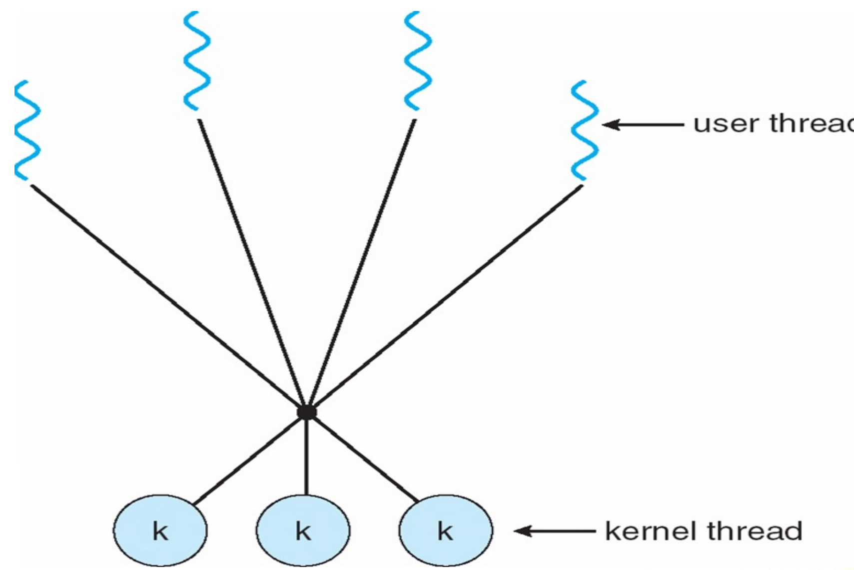
queue of thread control blocks



per-thread stacks

Multiplexing User-Level Threads

- The user-level thread package sees a "virtual" processor(s)
  - it schedules user-level threads on these virtual processors
  - each "virtual" processor is implemented by a kernel thread
- The big picture
  - Create as many kernel threads as there are processors
  - Create as many user-level threads as the application needs
  - Multiplex user-level threads on top of the kernel-level threads
- Why not just create as many kernel-level threads as app needs?
  - Context switching
  - Resources



User-Level vs. Kernel Threads

User-Level

- Managed by application
- Kernel not aware of thread
- Context switching cheap
- Create as many as needed
- Must be used with care

Kernel-Level

- Managed by kernel
- Consumes kernel resources
- Context switching expensive
- Number limited by kernel resources
- Simpler to use

Key issue:

kernel threads provide virtual processors to user-level threads, but if all of kthreads block, then all user-level threads will block *even* if the program logic allows them to proceed

User/Kernel Threads

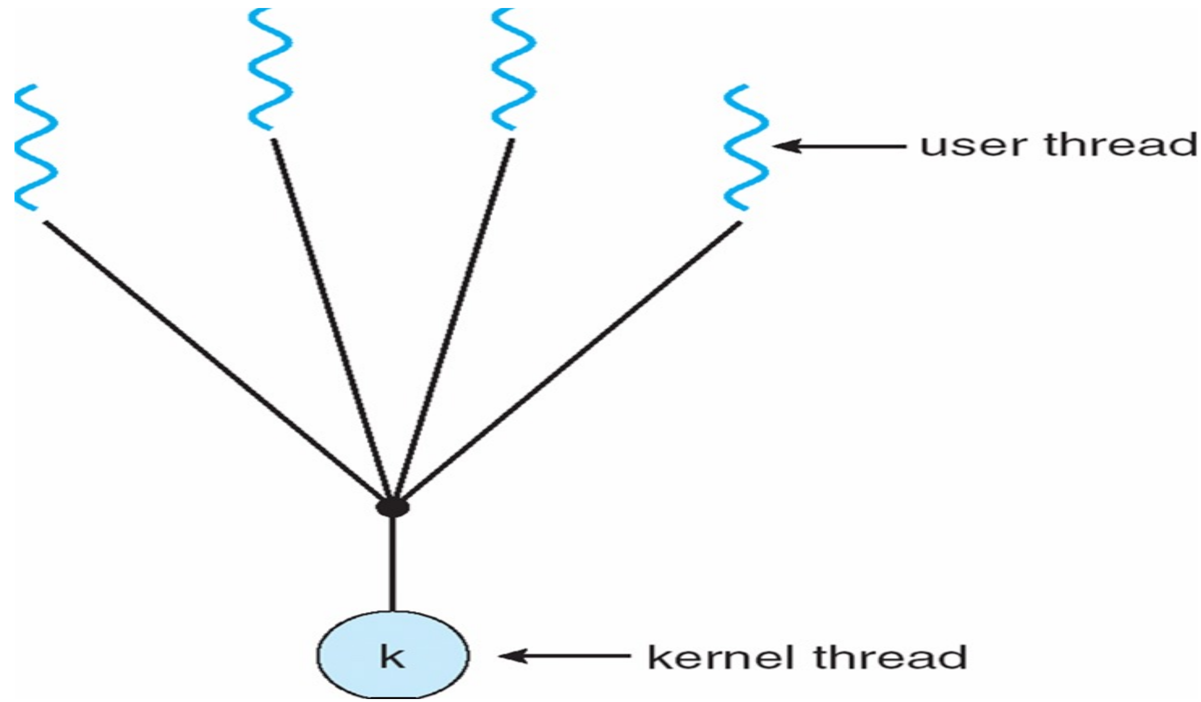
- Thread management done by user-level threads library
- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multithreading Models

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

Many-to-One Model



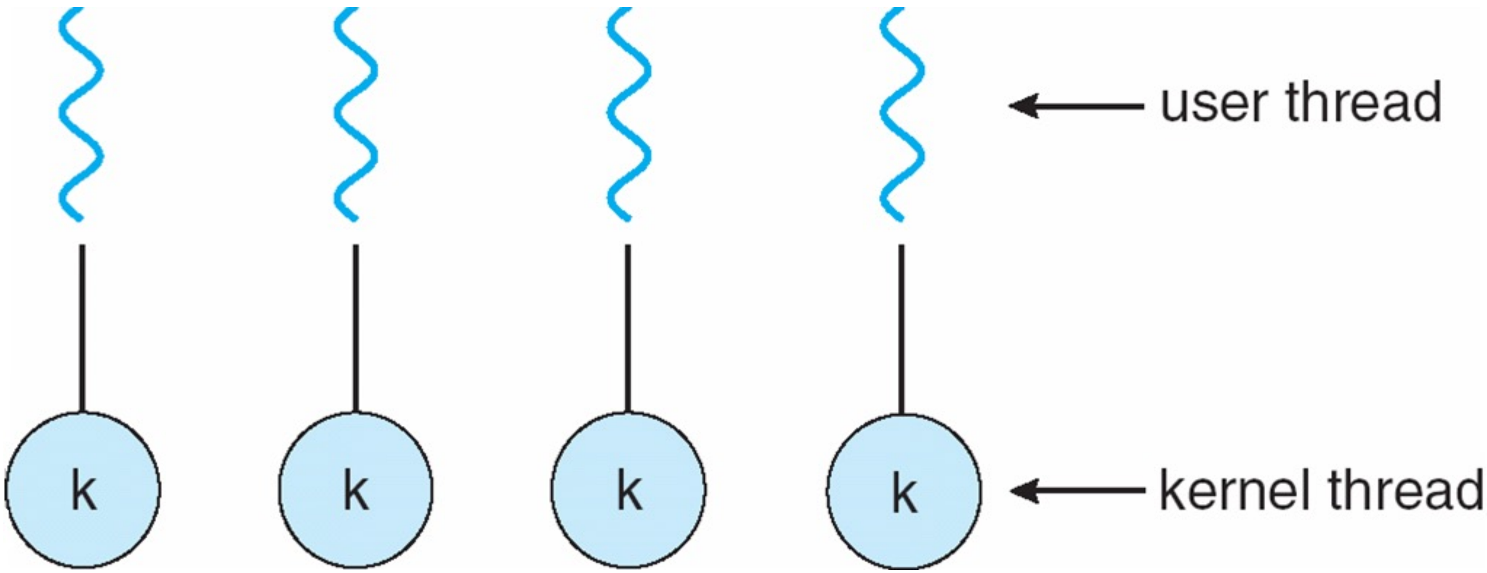
Many user-level threads mapped to single kernel thread

Thread creation, scheduling, synchronization done in user space.Mainly used in language systems, portable libraries.

- Fast - no system calls required
- Few system dependencies; portable
- No parallel execution of threads - can't exploit multiple CPUs
- All threads block when one uses synchronous I/O

One-to-one Model

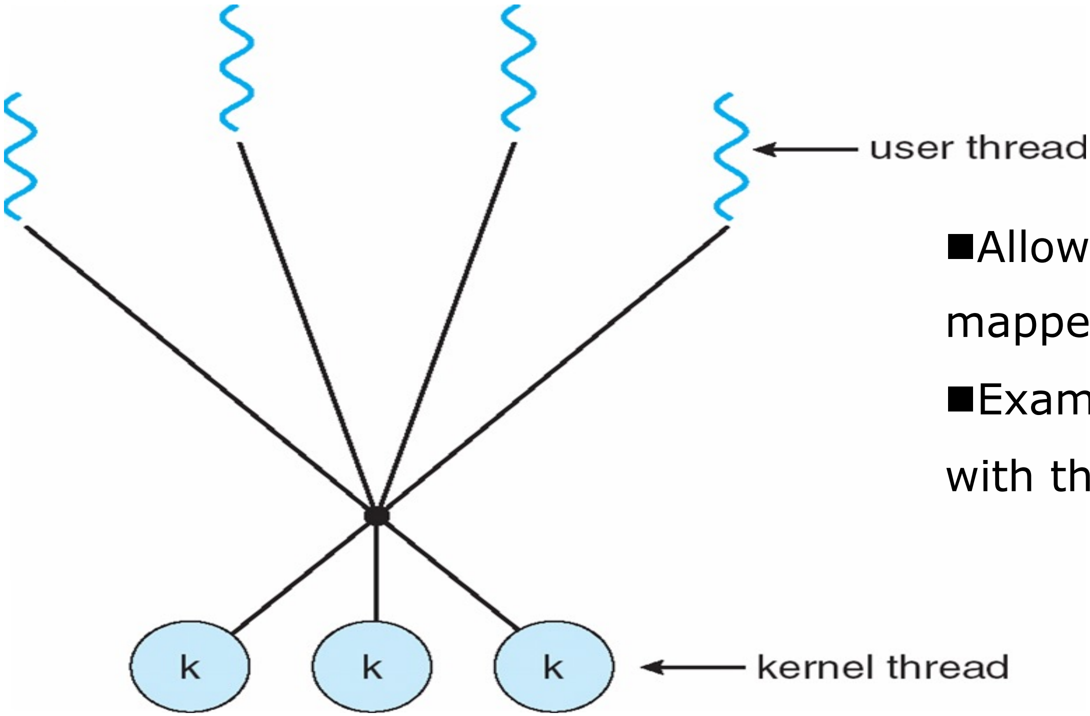
Each user-level thread maps to kernel thread



Thread creation, scheduling, synchronization require system calls. Used in Linux Threads, Windows NT, Windows 2000, OS/2, Solaris 9 and later

- More concurrency
- Better multiprocessor performance
- Each user thread requires creation of kernel thread
- Each thread requires kernel resources; limits number of total threads

Many-to-Many Model



- Allows many user level threads to be mapped to many kernel threads.
- Examples: Solaris, Windows NT/2000 with the *ThreadFiber* package

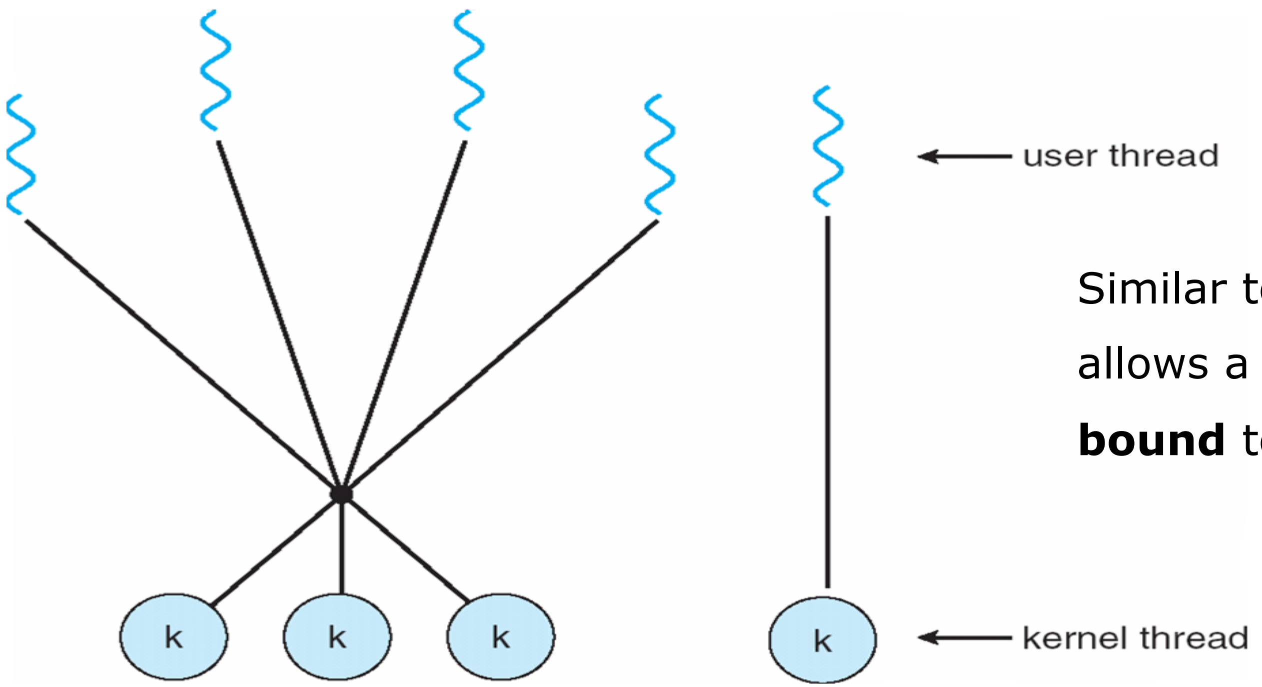
If U<k? No benefits of multithreading
If U>k, some threads may have to wait for an Kthread to run

- Active thread - executing on an Kthread
- Runnable thread - waiting for an Kthread

A thread gives up control of Kthread under the following:

– synchronization, lower priority, yielding, time slicing

Two-level Model



Similar to M:M, except that it allows a user thread to be **bound** to(绑定) kernel thread

- Supports both bound and unbound threads
  - Bound threads - permanently mapped to a single, dedicated kthread
  - Unbound threads - may move among kthreads in set
- Thread creation, scheduling, synchronization done in user space
- Flexible approach, "best of both worlds" (两全其美)

  Used in ,Solaris 8 and earlier implementation of Pthreads and several other Unix implementations (IRIX, HP-UX, Tru64 UNIX)

Thread Libraries

- Thread library provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - Library entirely in user space
  - Kernel-level library supported by the OS
- Examples:
  - Java thread
  - Win32 threads
  - POSIX Pthreads

Java Threads

- Java threads are managed by the JVM, typically implemented using the threads model provided by underlying OS:
  - On windows system, java threads are implemented using Win32 API
  - On UNIX or Linux, use Pthreads
- Java Threads包中定义了Thread类和Runnable接口，两种创建方法：

  ①扩展Thread类，并重置它的RUN()方法；

  ②定义一个类，实现Runnable接口；

Extending the Thread Class

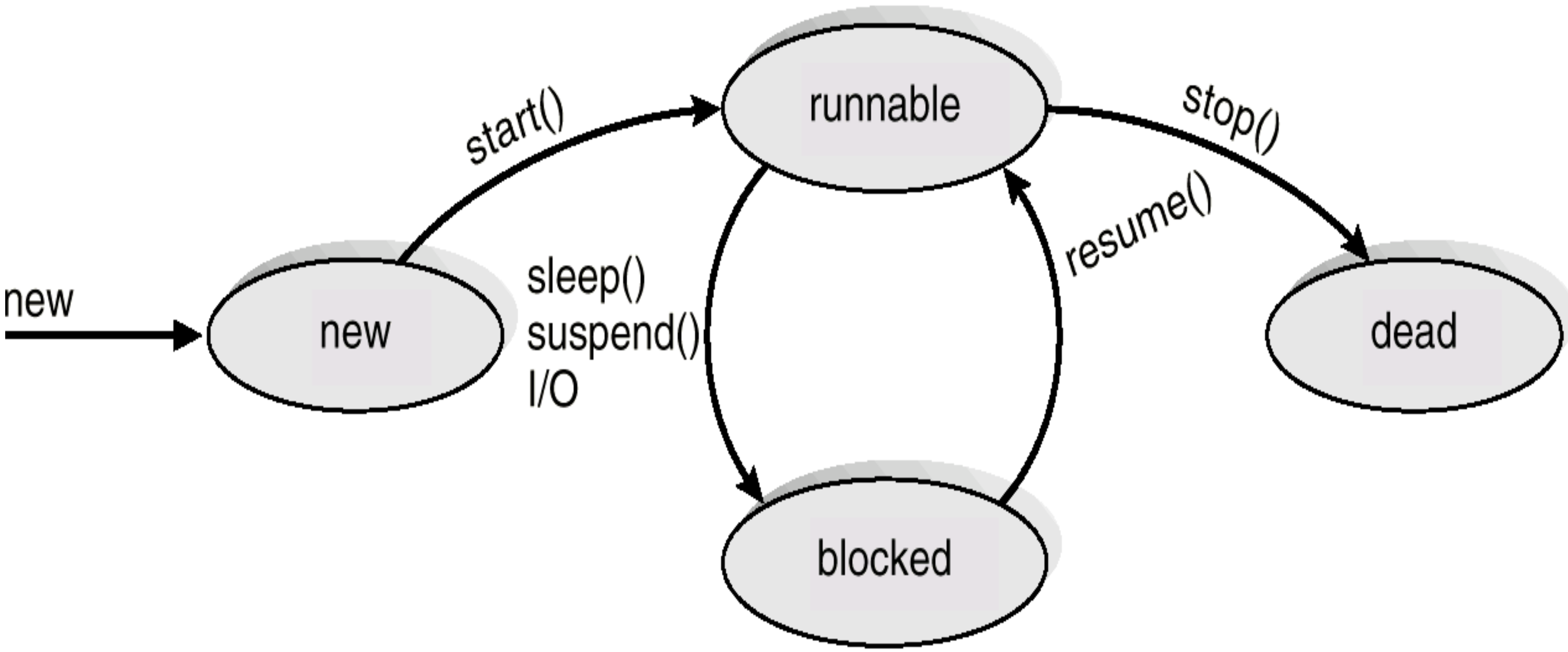
```
class  Worker1 extends Thread
{   public void run() {
            System.out.println(“I am a Worker Thread”);
    }
}
public class First
{   public static void main(String args[]) {
            Worker runner = new Worker1();
            runner.start();
            System.out.println(“I am the main thread”);
    }
}
```

start() 为新线程分配内存并初始化，然后调用run()方法

The Runnable Interface

```
public interface Runnable
{ public abstract void run();
}
class  Worker2 implements Runnable
{ public void run() {/*定义类Worker2实现Runnable接口，定义run()方法*/
        System.out.println("I am a Worker Thread"); }
}
public class Second
{ public static void main(String args[]) {
        Runnable runner = new Worker2();
        Thread thrd = new Thread(runner);
        thrd.start();
        System.out.println("I am the main thread");
    } /* 创建线程对象传递给Runnable对象，新线程由start()方法创建后，
    开始执行Runnable对象的run()方法*/
}
```

Java Thread States



Win32 Threads

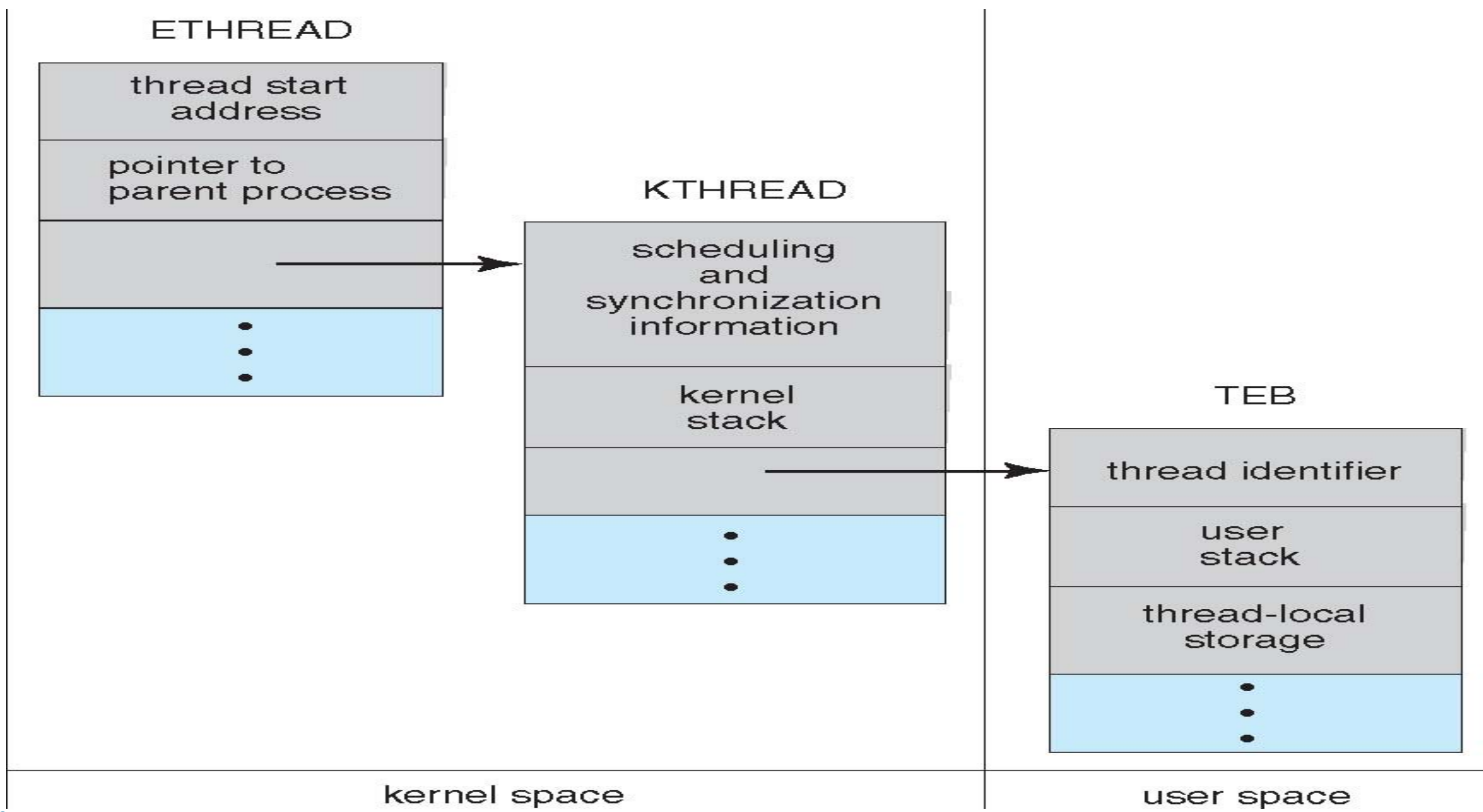
- Win32 API is the primary API for Microsoft OS (Win95,98,NT,2000,XP),A kernel-level library on windows systems

Windows XP Threads:

- Implements the one-to-one mapping, kernel-level
- 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

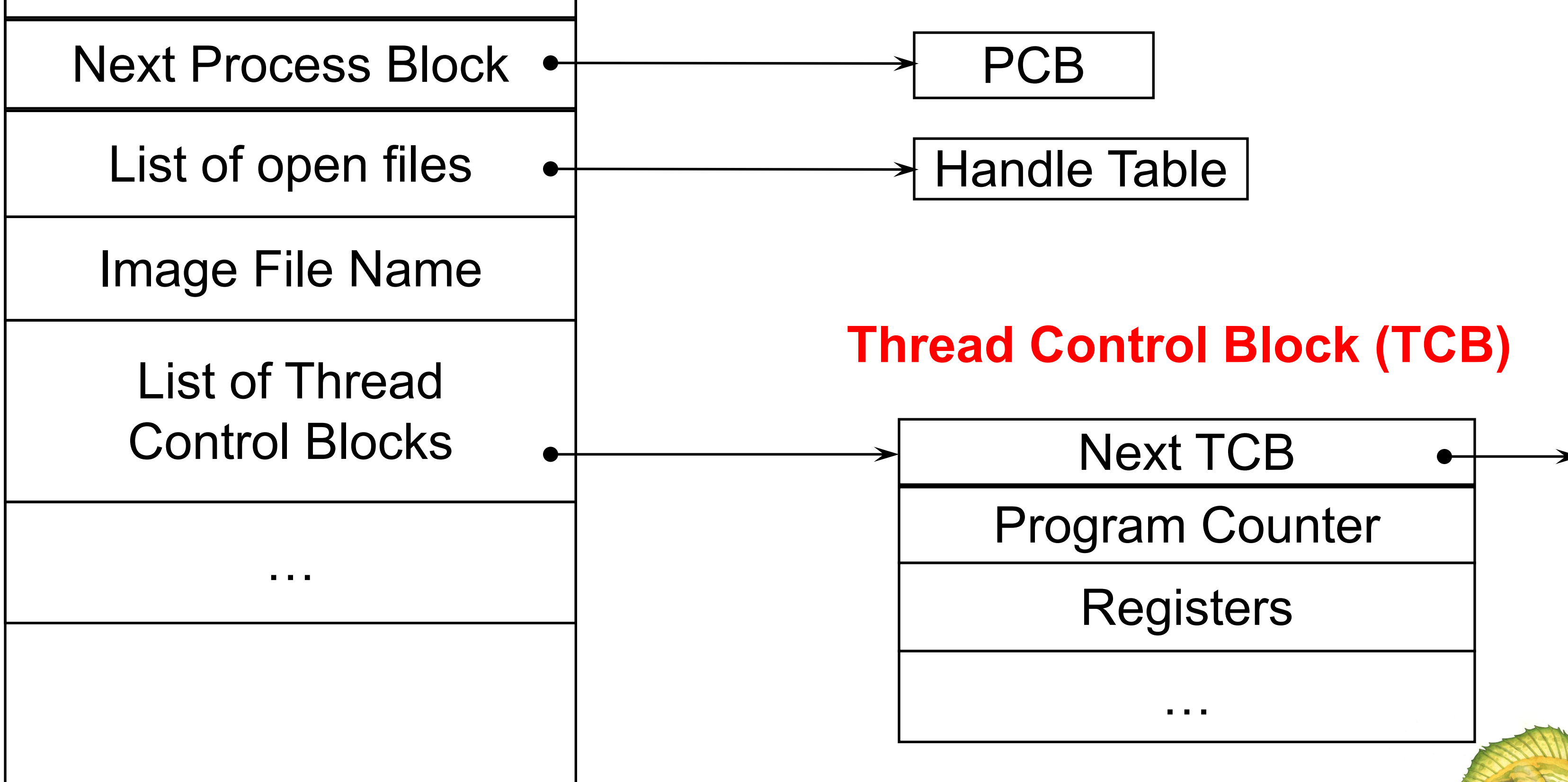
Windows XP Threads

- The primary data structures of a thread include:
  - ETHREAD (executive thread block)
  - KTHREAD (kernel thread block)
  - TEB (thread environment block)

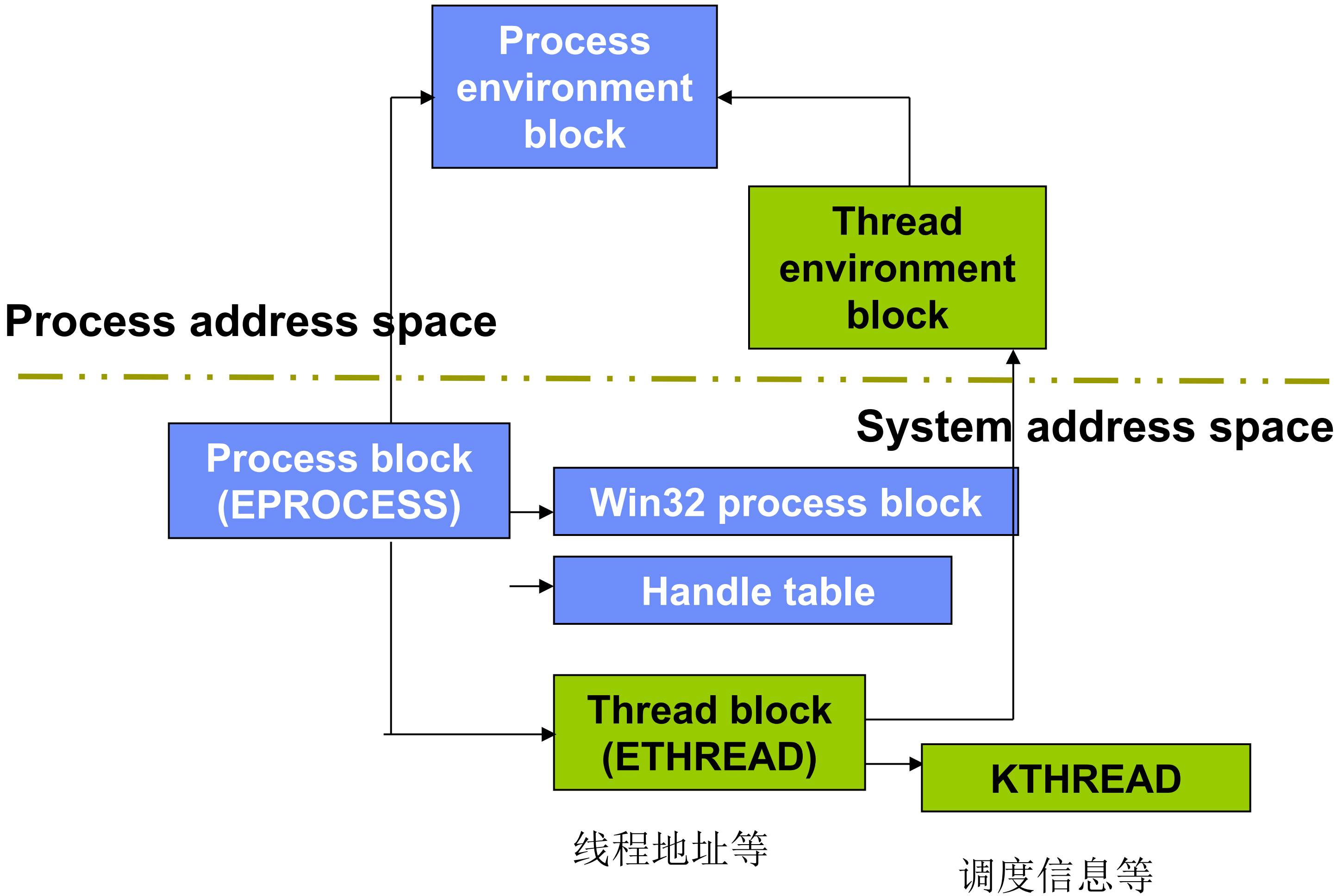


Process Control Block (PCB)

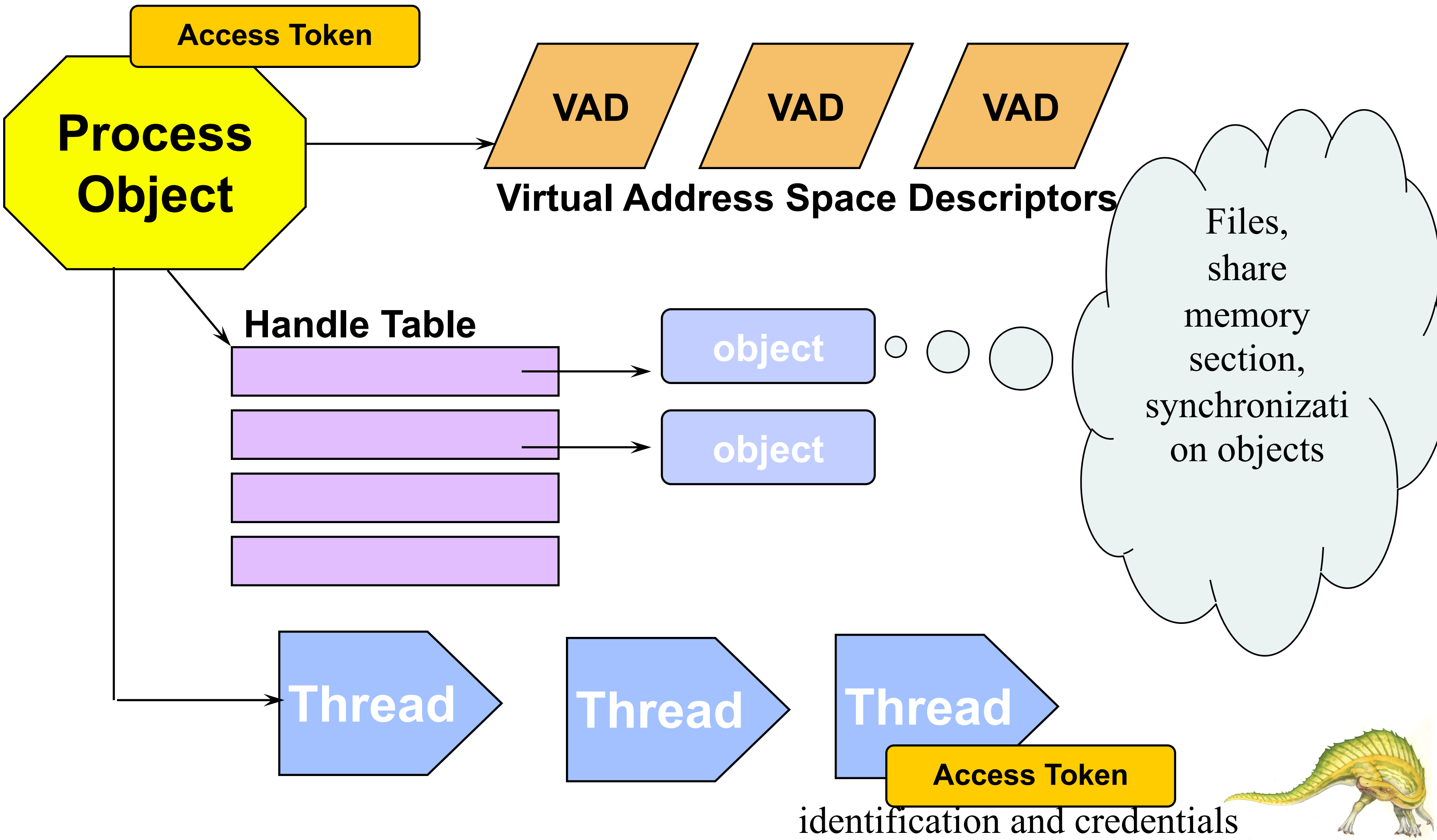
- This is an abstract view
- Windows implementation of PCB is split in multiple data structures



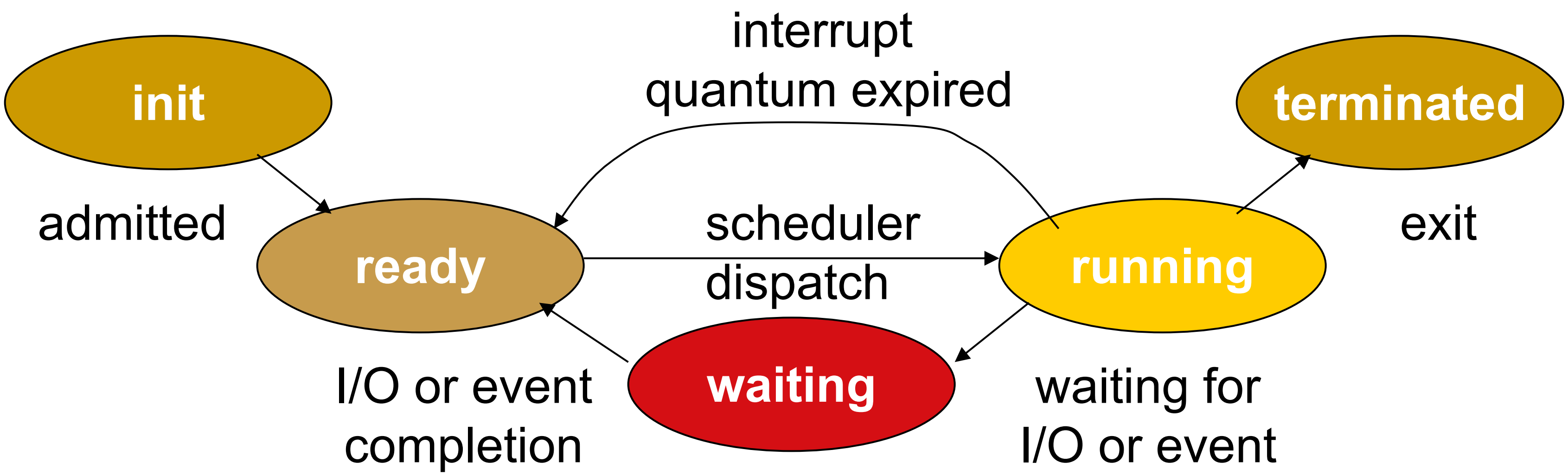
Windows Process and Thread Internals



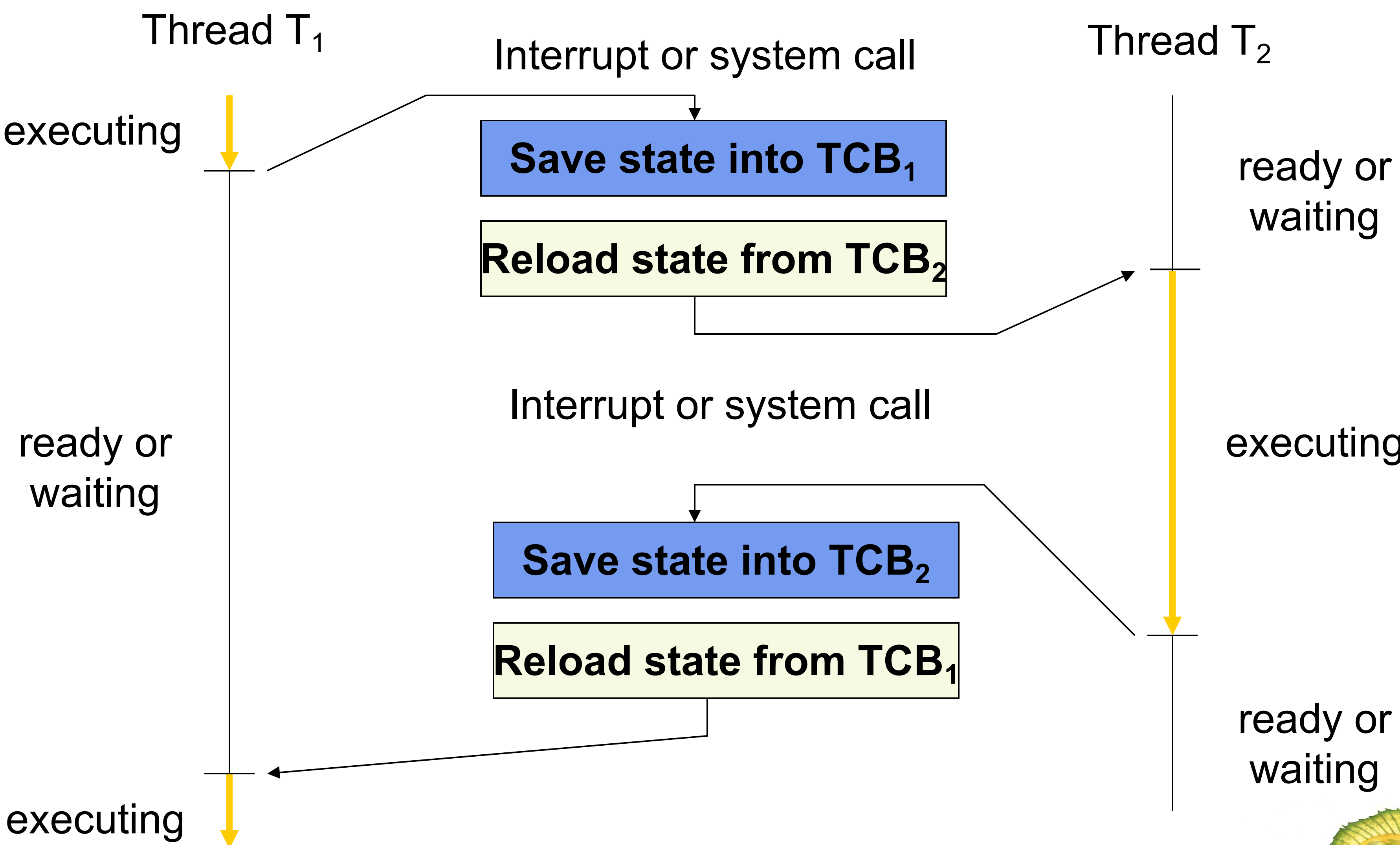
Windows Processes & Threads Internal Data Structures



Windows XP Threads



CPU Switch from Thread to Thread



POSIX Pthreads

- May be provided either as user-level or kernel-level
- 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

  (Linux, Solaris, Mac OS X)

Linux Threads

- Linux does not distinguish between processes and threads; Linux refers to them as *tasks* rather than *process* or *thread;*
- Thread creation is done through system call clone() or pthread_create() ;

◆调用fork()创建一个新进程，它具有父进程所有相关数据结构的拷贝；

◆调用clone()创建新进程，但它不复制父进程的数据结构，而是指向了父进程的数据结构，从而允许子进程共享父进程的内存和其他资源。或者定义一个函数作为线程的入口，然后调用pthread_create() 创建线程。

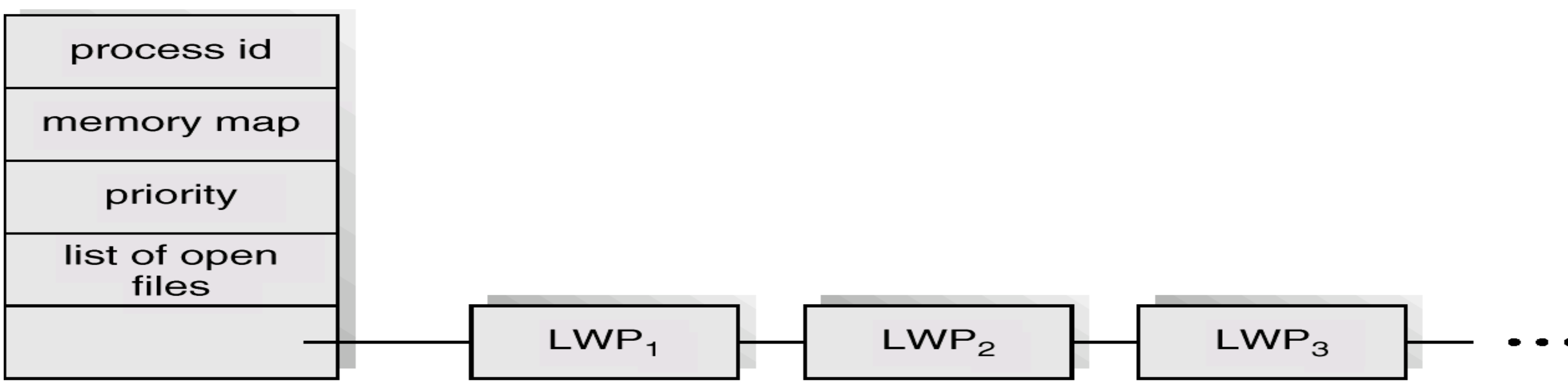
Linux Threads

- clone() allows a child task to share the address space of the parent task (process)
- Flags determine how much sharing is to take place between the parent and child
- If no flag is set when clone(), no sharing take place, resulting in functionality similar to fork()

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.

Solaris threads

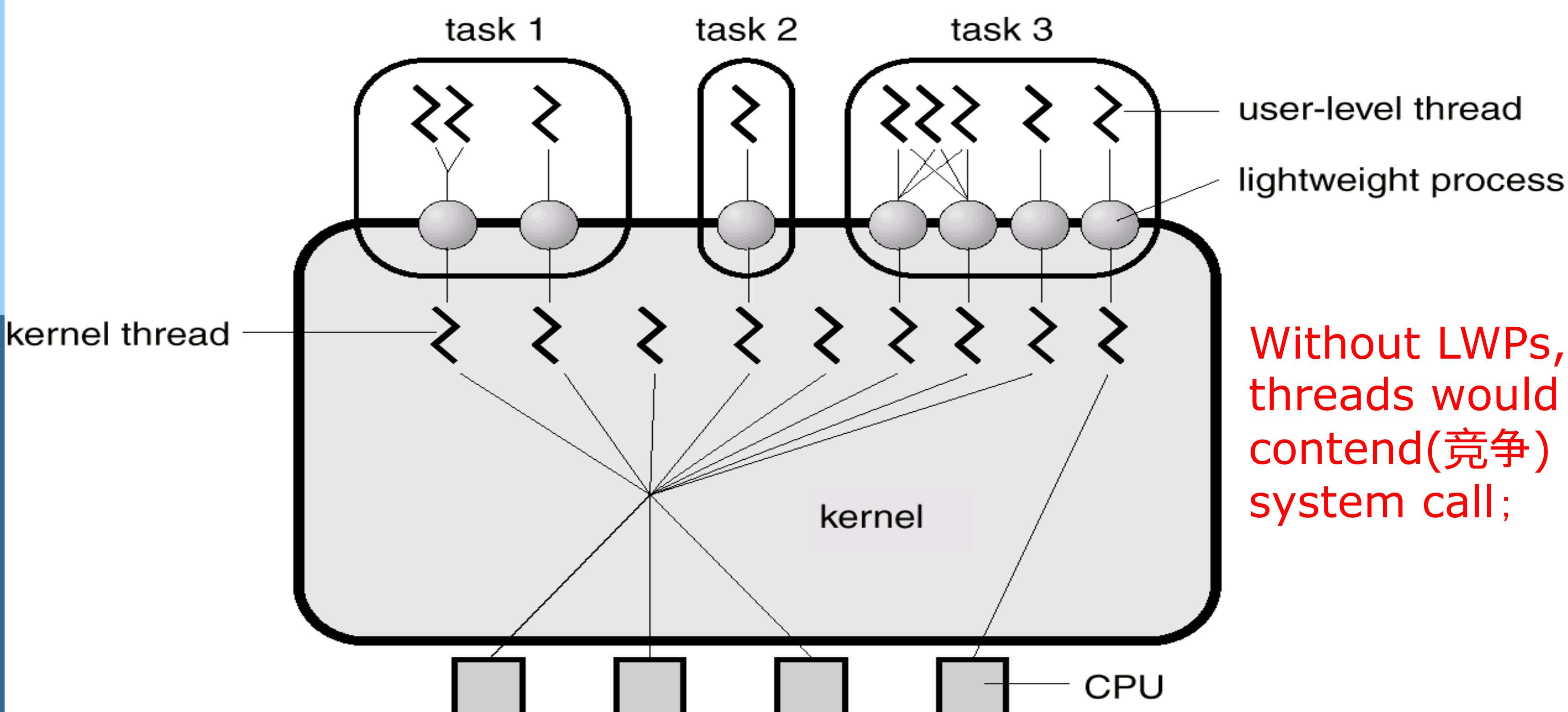
- Solaris is a version of UNIX with support for threads at the kernel and user levels, symmetric multiprocessing, and real-time scheduling.
  - Kernel threads;
  - Lightweight Processes;
  - User Level Threads;
- LWP – (lightweight processes)intermediate level between user-level threads and kernel-level threads.



Solaris process

Solaris threads

- Kernel Threads: small data structure and a stack; thread switching relatively fast. Kernel only sees the LWPs that support user-level threads.
- LWP: a register set for the user-thread it is running, accounting and memory information, switching between LWPs is relatively slow.
- User Threads: only need stack and program counter; no kernel involvement means fast switching.

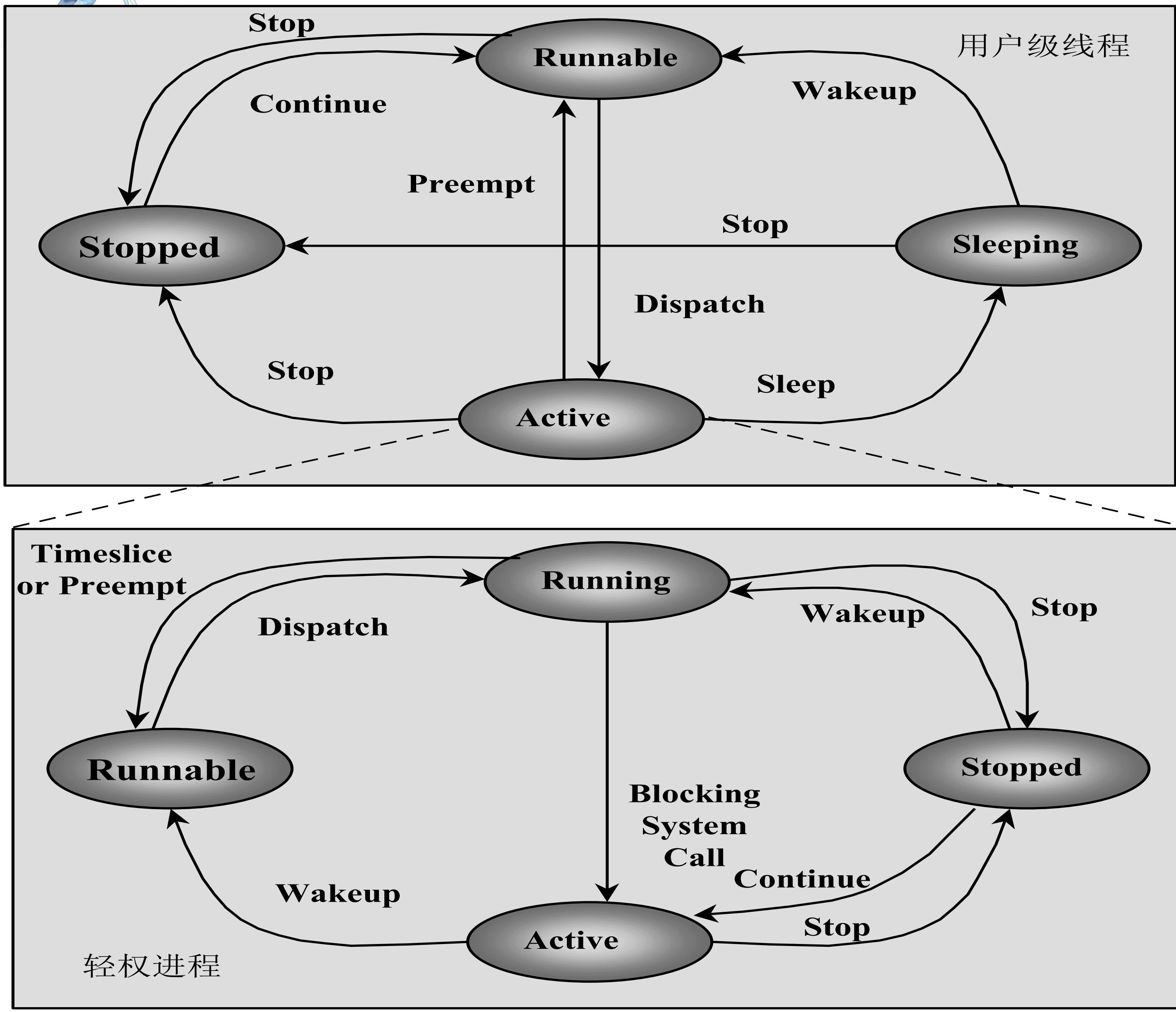


Without LWPs, user threads would contend(竞争) at system call;

Solaris threads

- User level threads may be either bound or unbound ;
- Bound: A user thread is permanently attached to a LWP;
- All unbound threads in an application are multiplexed onto the pool of available for the application;
  - Threads are unbound by default .
- The thread library adjusts LWPs in the pool
  - The thread library ages LWPs and deletes them when they are unused for a long time, typically about 5 minutes.

Solaris: User Threads and LWP



C library functions on Solaris threads

•Create a User Threads

• int thr_create(void *stack_base, size_t stack_size,

void *(*start_routine)(void *), void *arg, long flags, thread_t *new_thread_id);

flags: THR_BOUND（永久捆绑）

THR_NEW_LWP（创建新LWP放入LWP池）

两者同时指定则创建两个新LWP，一个永久捆绑而另一个放入LWP池；

•Create a LWP

int _lwp_create(ucontext_t *contextp, unsigned long flags, lwpid_t *new_lwp_id);

• Make a context on LWP

• void _lwp_makecontext(ucontext_t *ucp,

void (*start_routine)(void *), void *arg,

void *private, caddr_t stack_base, size_t stack_size);

Implicit Threading

- Growing in popularity as numbers of threads increase, program correctness more difficult with explicit threads
- Creation and management of threads done by compilers and run-time libraries rather than programmers
- Three methods explored
  - Thread Pools
  - OpenMP
  - Grand Central Dispatch
- Other methods include Microsoft Threading Building Blocks (TBB), **java.util.concurrent** package

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
  - Separating task to be performed from mechanics of creating task allows different strategies for running task（执行与创建分离）
    - i.e.Tasks could be scheduled to run periodically
- Windows API supports thread pools:

```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
     * this function runs as a separate thread.
     */
}
```

OpenMP

- Set of compiler directives and an API for C, C++, FORTRAN
- Provides support for parallel programming in shared-memory environments
- Identifies **parallel regions** – blocks of code that can run in parallel

#pragma omp parallel

Create as many threads as there are cores

#pragma omp parallel for
for(i=0;i<N;i++) {

c[i] = a[i] + b[i];

}

Run for loop in parallel

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

int main(int argc, char *argv[])
{
   /* sequential code */

   #pragma omp parallel
   {
      printf("I am a parallel region.");
   }

   /* sequential code */

   return 0;
}
```

Grand Central Dispatch

- Apple technology for Mac OS X and iOS operating systems
- Extensions to C, C++ languages, API, and run-time library
- Allows identification of parallel sections
- Manages most of the details of threading
- Block is in “^{ }” - **^{ printf("I am a block"); }**
- Blocks placed in dispatch queue
  - Assigned to available thread in thread pool when removed from queue

Grand Central Dispatch

- Two types of dispatch queues:
  - serial – blocks removed in FIFO order, queue is per process, called **main queue**
    - Programmers can create additional serial queues within program
  - concurrent – removed in FIFO order but several may be removed at a time
    - Three system wide queues with priorities low, default, high

```
dispatch_queue_t queue = dispatch_get_global_queue
   (DISPATCH_QUEUE_PRIORITY_DEFAULT, 0);

dispatch_async(queue, ^{ printf("I am a block."); });
```

Threading Issues

- Semantics of **fork()** and **exec()** system calls
- Thread cancellation of target thread
  - Asynchronous or deferred
- Signal handling
- Thread pools
- Thread-specific data
- Scheduler activations

Thread Hazards

```
int a = 1, b = 2, w = 2;
main() {
    CreateThread(fn, 4);
    CreateThread(fn, 4);
    while(w)  ;
}
fn() {  int v = a + b;
        w--;
}
```

- A statement like "w--" in C (or C++) is implemented by several machine instructions:

```
ld      r4, #w
add     r4, r4, -1
st      r4, #w
```

- Now, imagine the following sequence, what is the value of w?

```
ld      r4, #w
____________            ____________
____________            ld      r4, #w
____________            add     r4, r4, -1
add     r4, r4, -1      st      r4, #w
st      r4, #w
```

作业

- 编写一个C/S架构的分布式程序， Server接收Client发来的请求,执行一个计算F(X)并给Client返回结果;分别用进程与线程作为服务器Server实现,并比较服务器的开销.可以在一台机器上模拟.

End of Chapter 4