Operating Systems

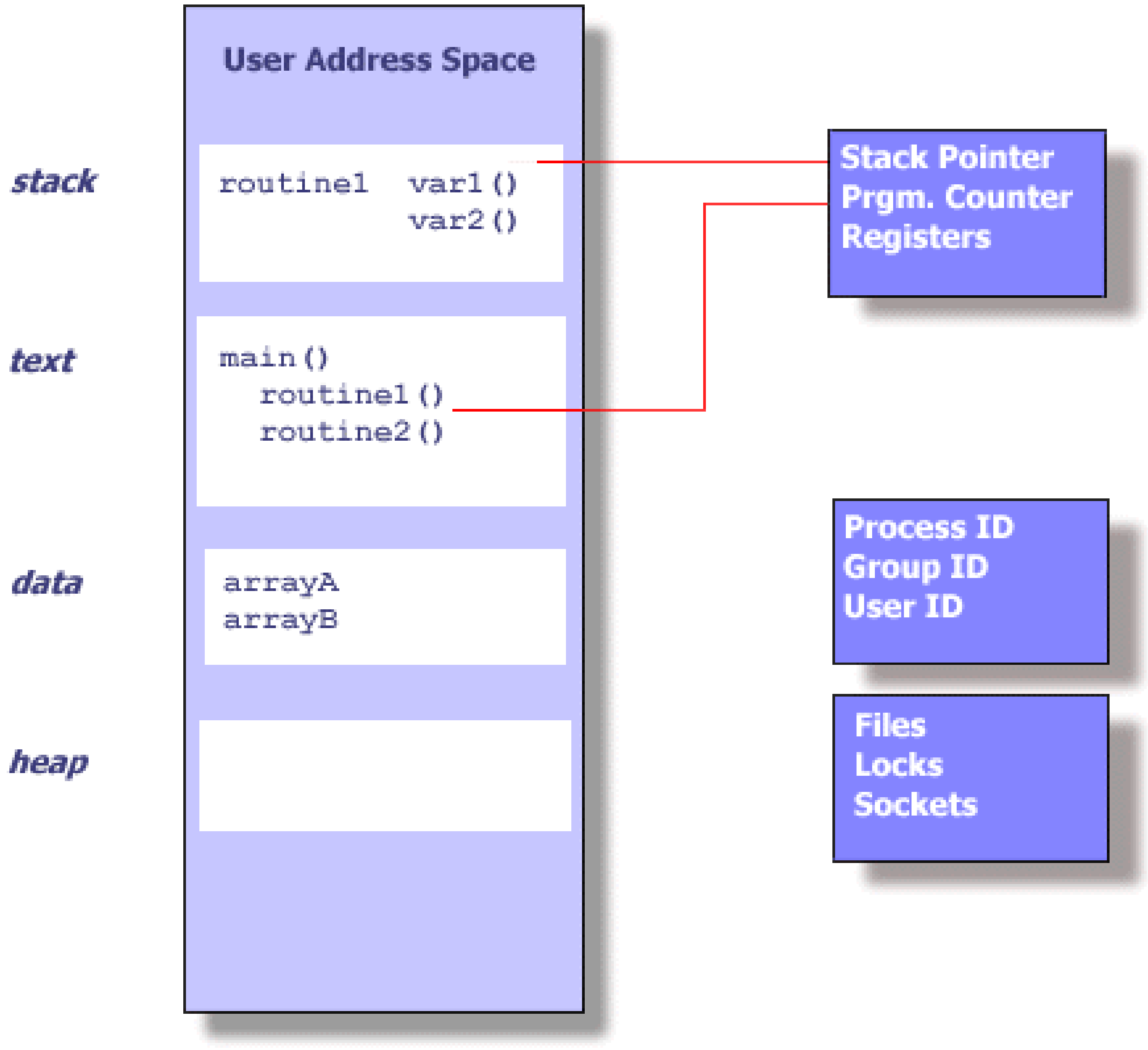
A Biswas

Threads

Threads

```c
#include "csapp.h"

void *thread(void *vargp);

int main()
{
    pthread_t tid;

    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp)
{
    printf("Hello, world!\n");
    return NULL;
}
```

Threads



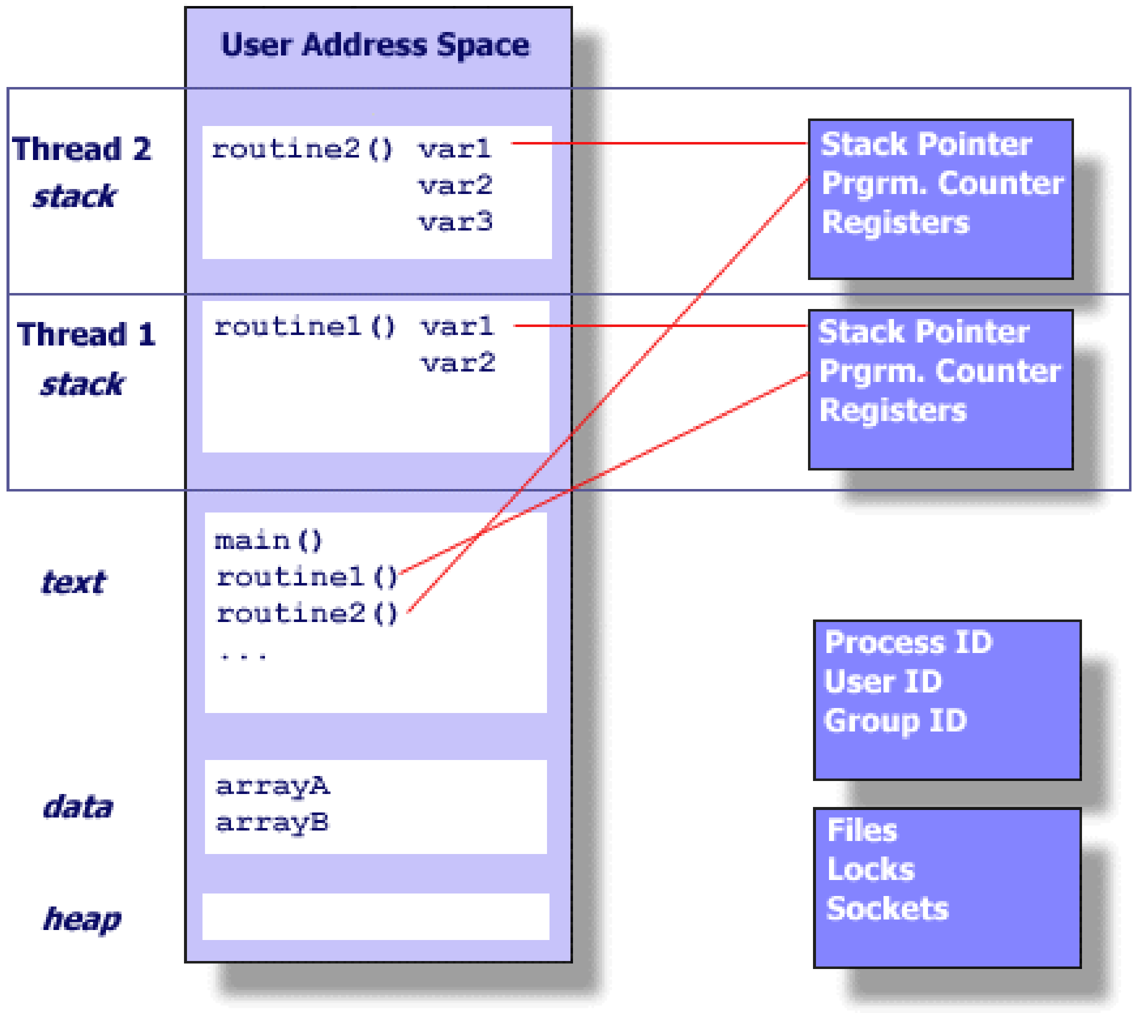
Process

Threads

Processes contain information about program resources and program execution state, including:

Process ID, process group ID, user ID, and group ID
Environment
Working directory
Program instructions
Registers
Stack
Heap
File descriptors
Signal actions
Shared libraries
Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory).

Threads



Threads within a process

Threads

This independent flow of control is accomplished because a thread maintains its own:

Stack pointer

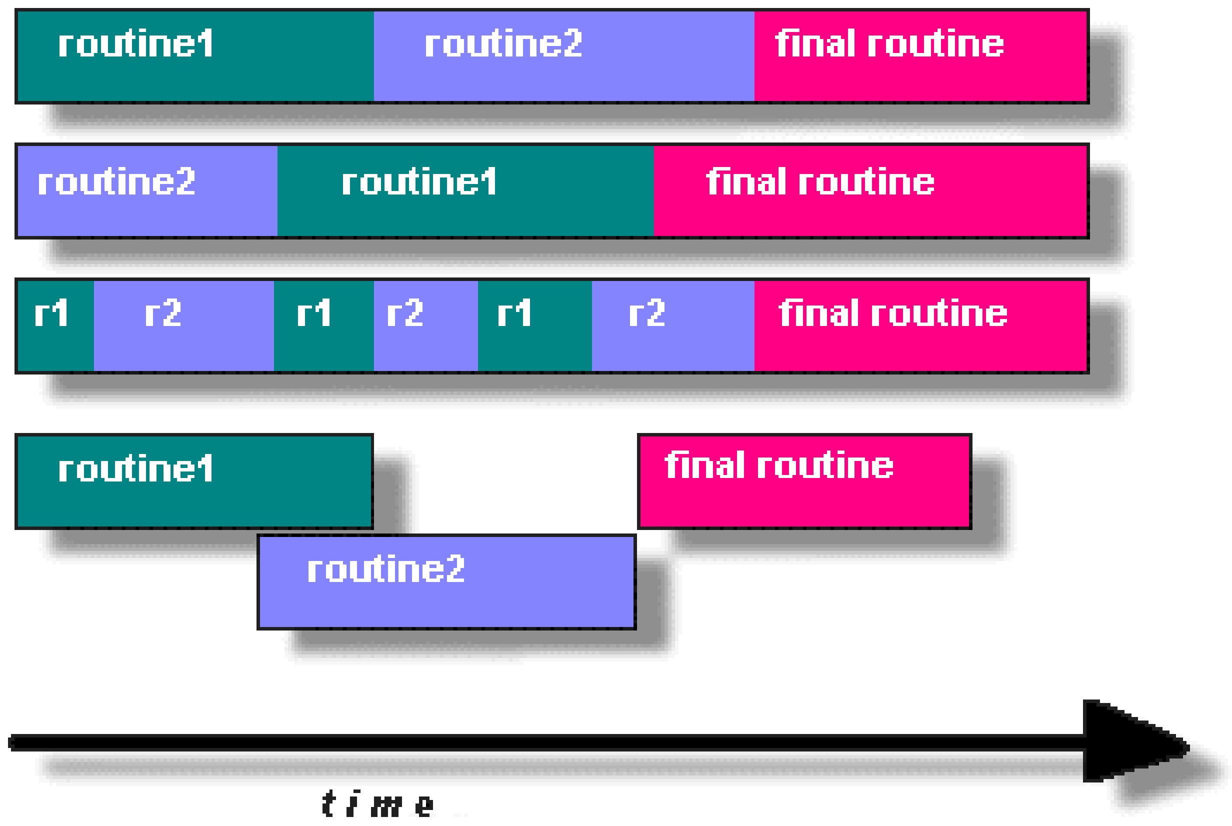
Registers

Scheduling properties (such as policy or priority)

Set of pending and blocked signals

Thread specific data.

Platform	fork()			pthread_create()		
	real	user	sys	real	user	sys
AMD 2.3 GHz Opteron (16cpus/node)	12.5	1.0	12.5	1.2	0.2	1.3
AMD 2.4 GHz Opteron (8cpus/node)	17.6	2.2	15.7	1.4	0.3	1.3
IBM 4.0 GHz POWER6 (8cpus/node)	9.5	0.6	8.8	1.6	0.1	0.4
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.2	30.7	27.6	1.7	0.6	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.5	48.6	47.2	2.1	1.0	1.5
INTEL 2.4 GHz Xeon (2 cpus/node)	54.9	1.5	20.8	1.6	0.7	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.5	1.1	22.2	2.0	1.2	0.6



It must be able to be organized into discrete, independent tasks which can execute concurrently.

For example, if routine1 and routine2 can be interchanged, interleaved and/or overlapped in real time, they are candidates for threading.

Programs having the following characteristics may be well suited for pthreads:

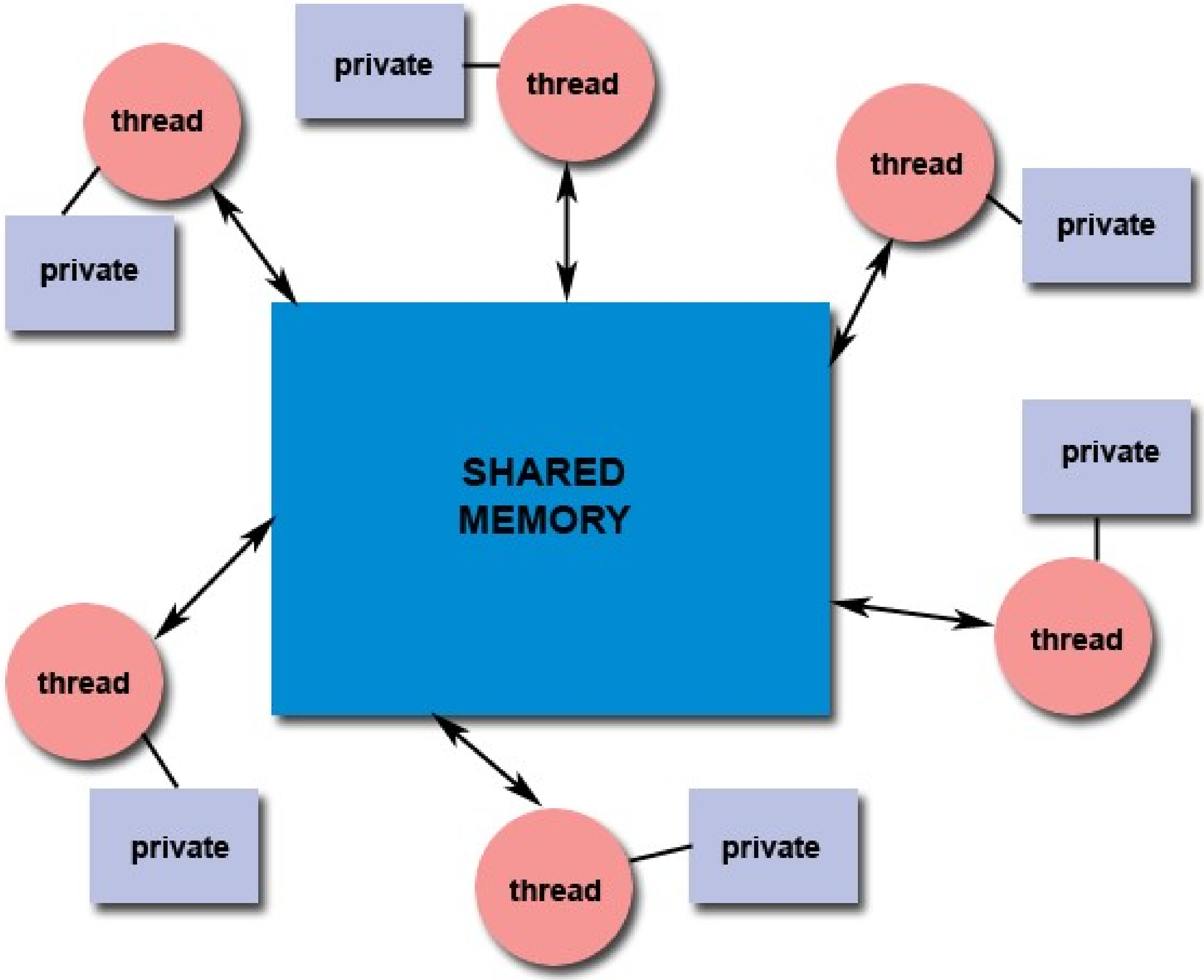
Work that can be executed, or data that can be operated on, by multiple tasks simultaneously

Block for potentially long I/O waits

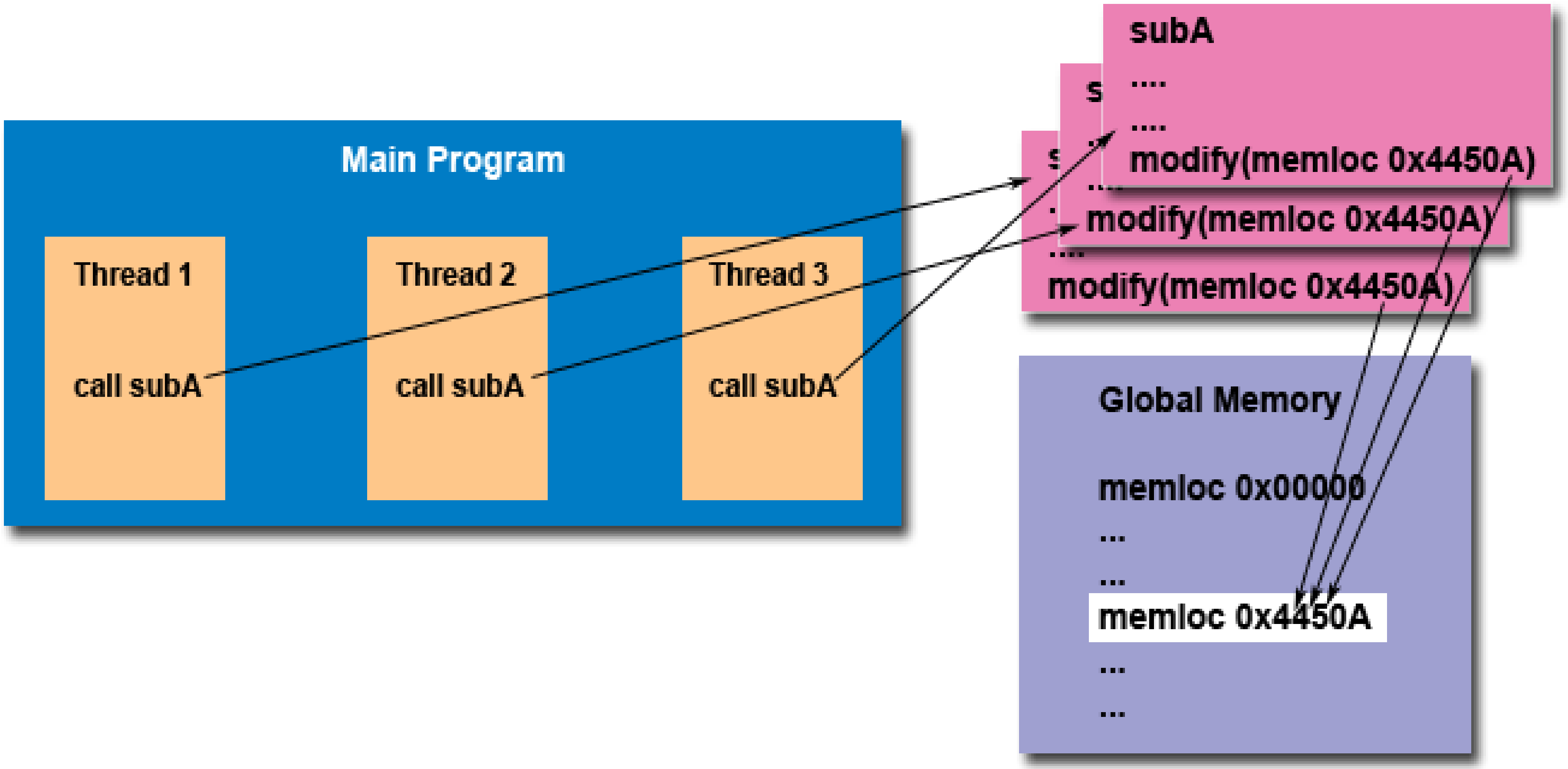
Use many CPU cycles in some places but not others

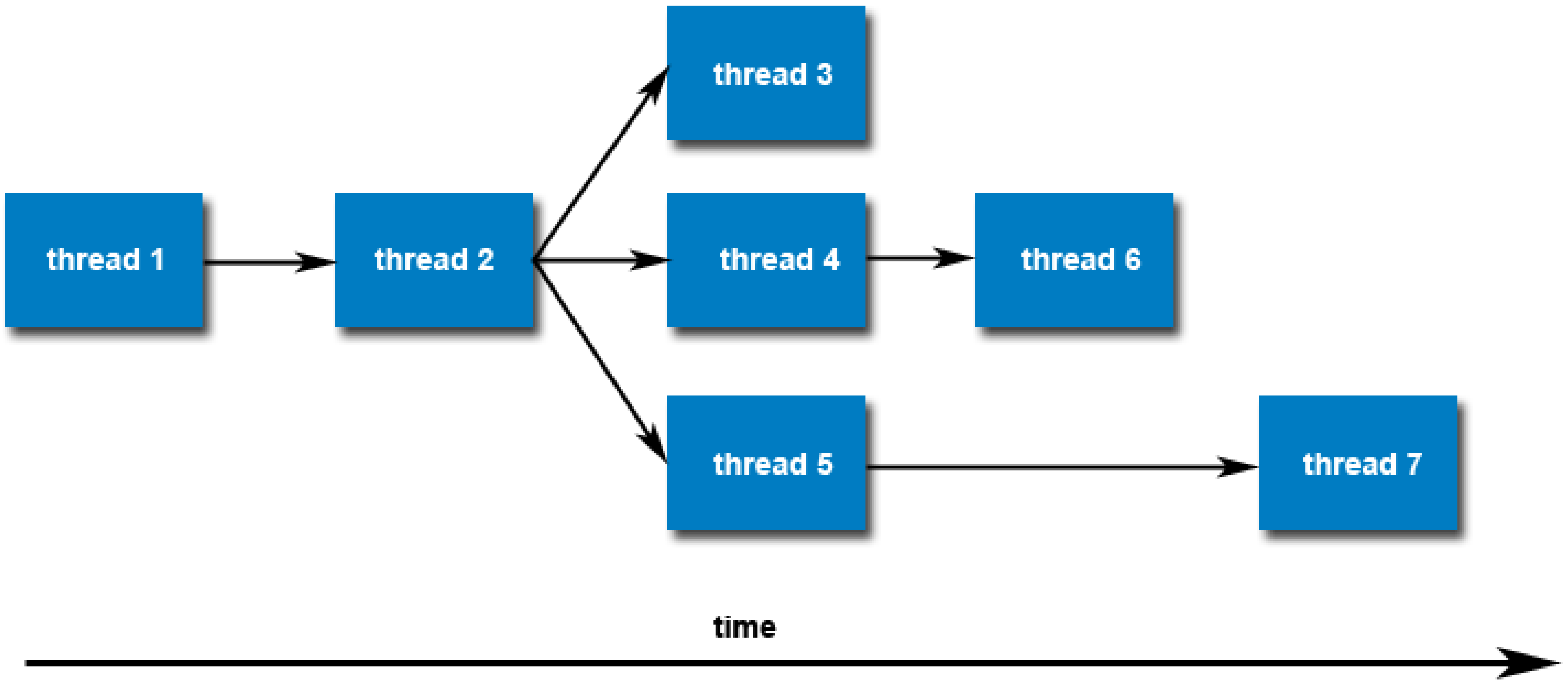
Must respond to asynchronous events

Some work is more important than other work
(priority interrupts)


SHARED MEMORY
thread
private
thread
private
thread
private
thread
private
thread
private
thread
private

Thread-safeness: is an application's ability to execute multiple threads simultaneously without "clobbering" shared data or creating "race" conditions.

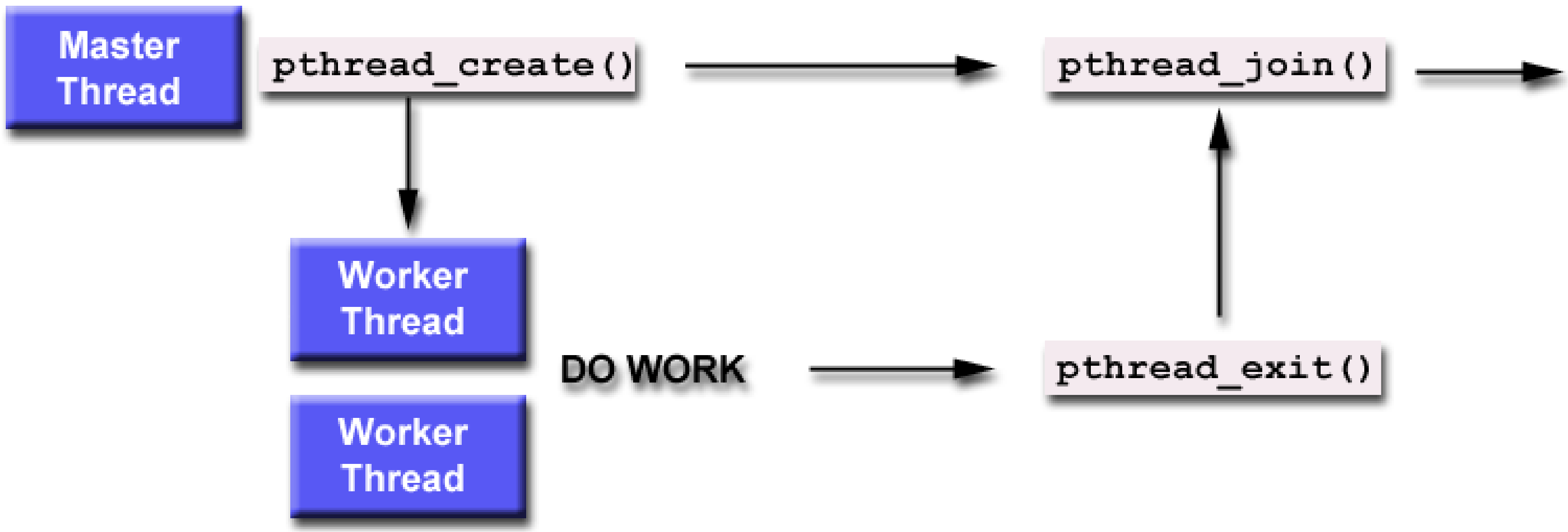




Once created, threads are peers, and may create other threads.

There is no implied hierarchy or dependency between threads.

"Joining" is one way to accomplish synchronization between threads.



•The **pthread_join()** subroutine blocks the calling thread until the specified **threadid** thread terminates.

•The programmer is able to obtain the target thread's termination return **status** if it was specified in the target thread's call to **pthread_exit()**.

•A joining thread can match one **pthread_join()** call. It is a logical error to attempt multiple joins on the same thread.

Threads

What is a thread?

A thread is a unit of execution associated with a process.

It has its own

- thread id;

- stack;

- stack pointer;

- program counter;

- condition codes;

- general purpose registers

Threads

Multiple threads associated with a process run concurrently in the context of the process.

Threads shares

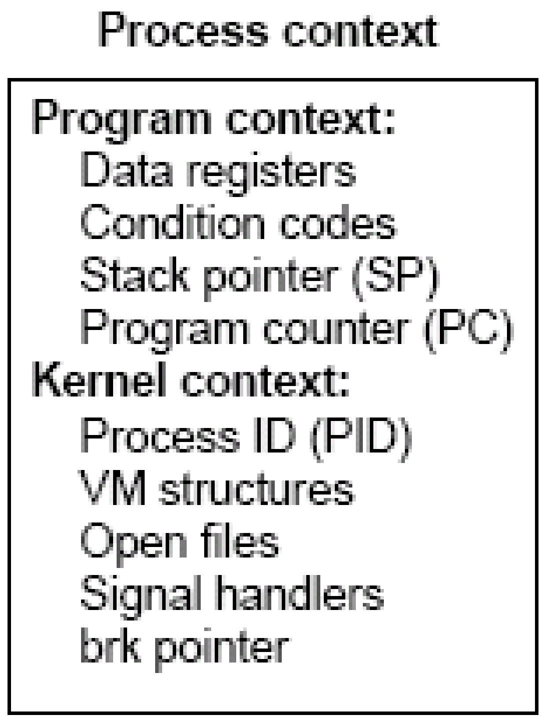
- code;
- data;
- heap;
- shared libraries;
- signal handlers;
- open files

Example: Web server – a separate thread for each open connection.

Basic concepts of threads

Process context can be partitioned into

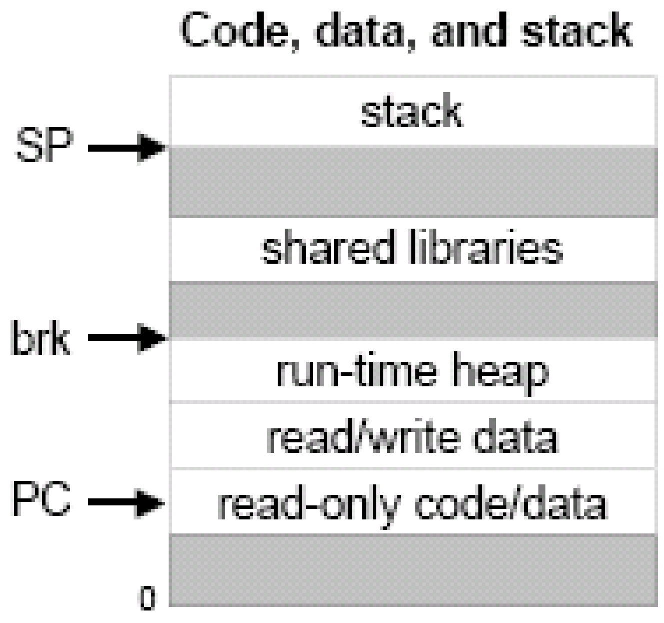
1. **Program context**
2. **Kernel context**



Process context

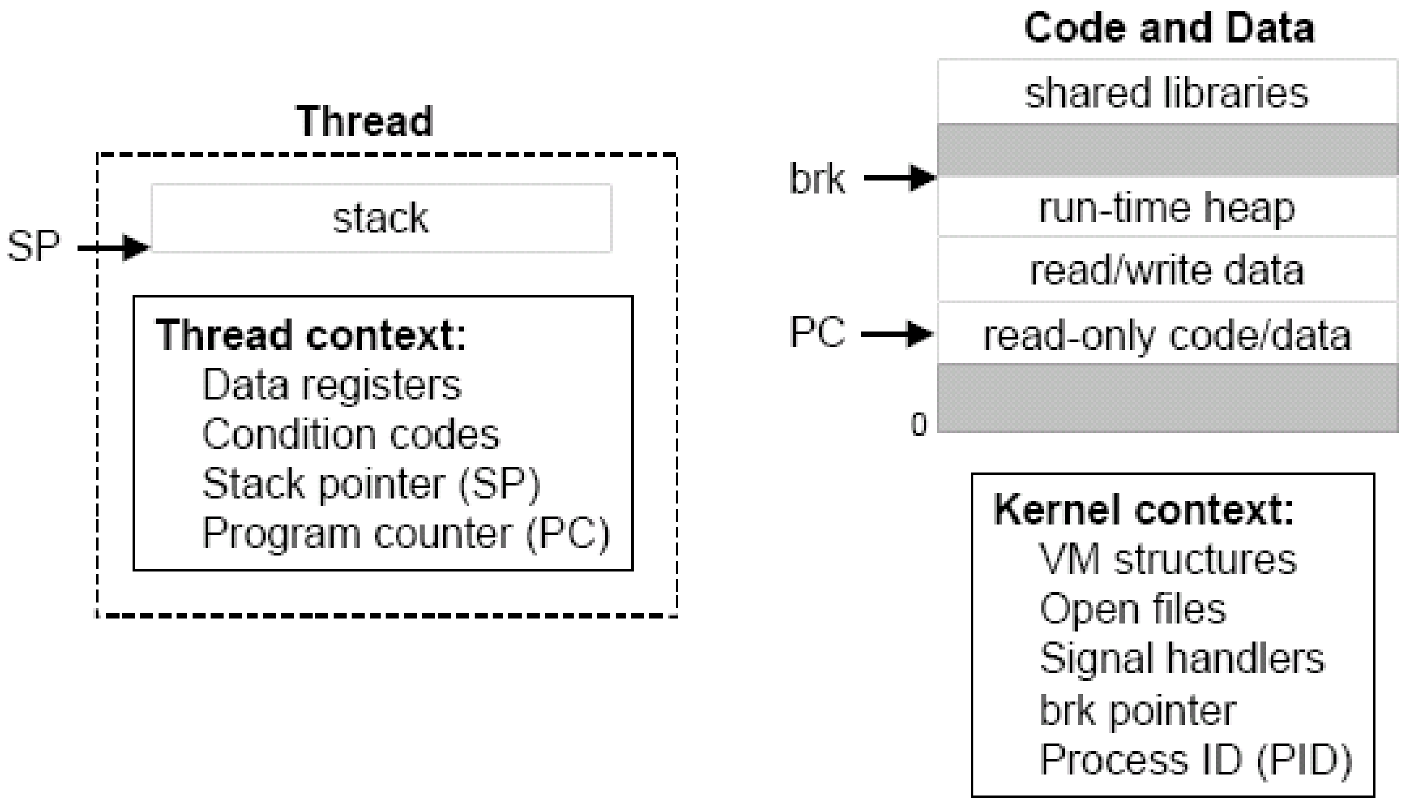
Program context:
- Data registers
- Condition codes
- Stack pointer (SP)
- Program counter (PC)

Kernel context:
- Process ID (PID)
- VM structures
- Open files
- Signal handlers
- brk pointer



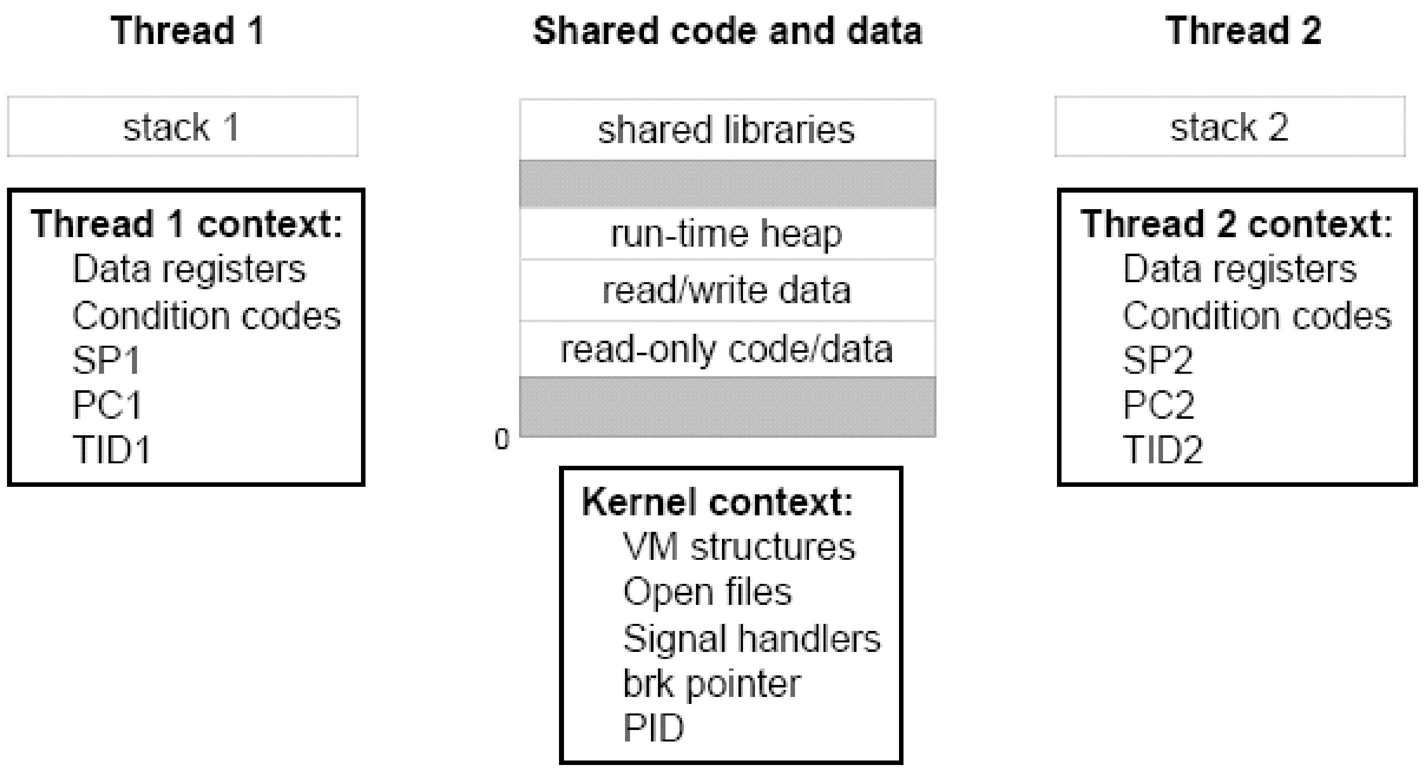
Basic concepts of threads

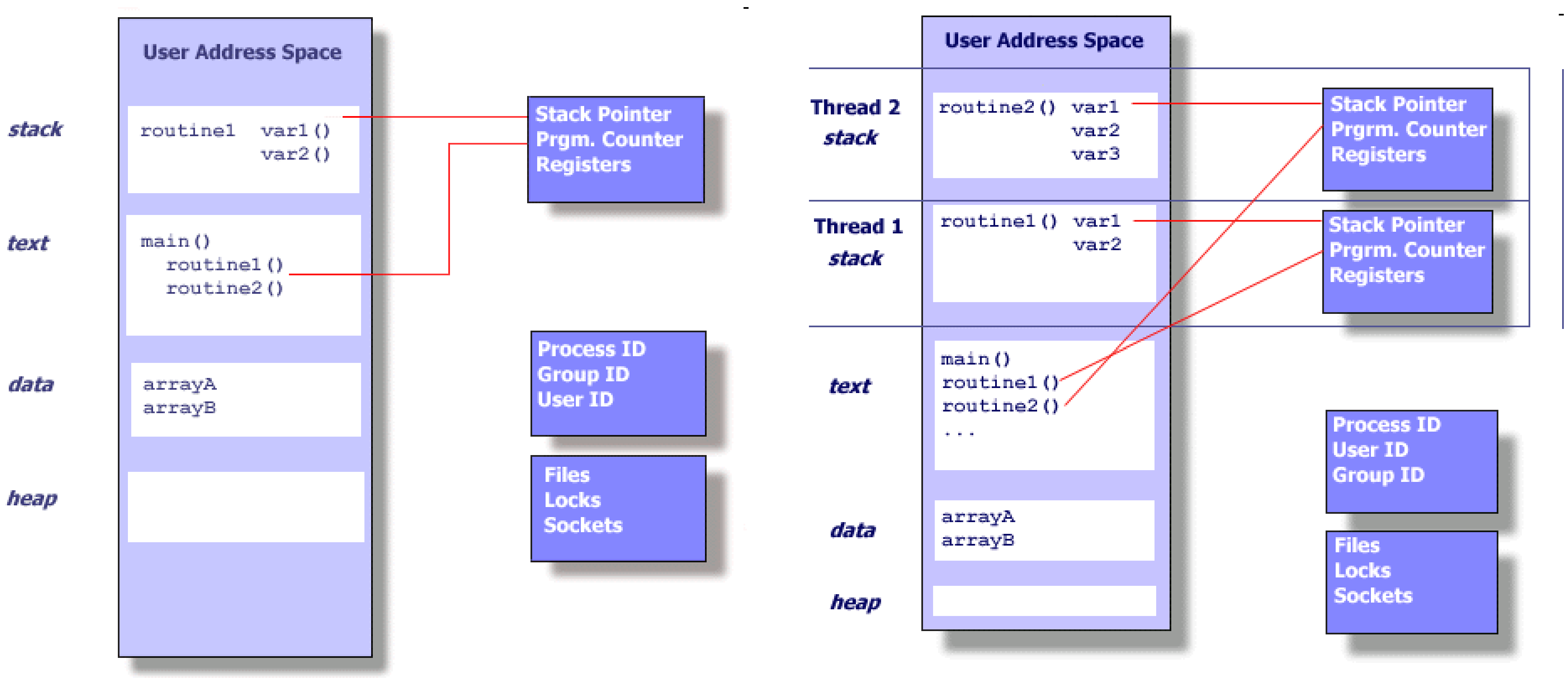
Alternative view of the process context



Basic concepts of threads

Associating multiple threads with a process

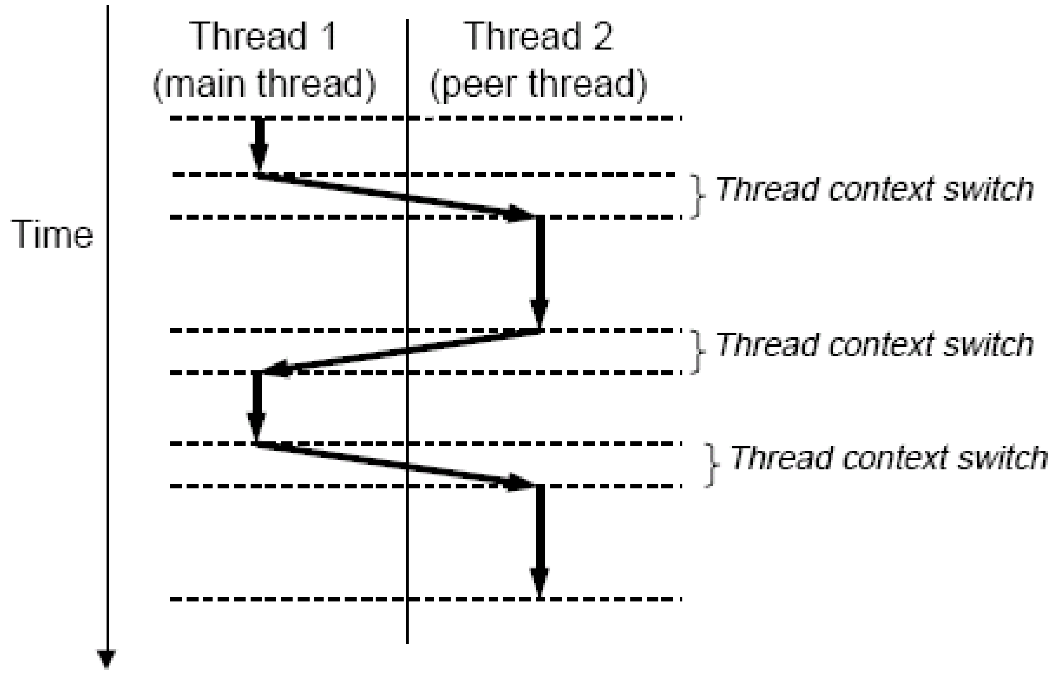




User Address Space
stack
routine1 var1() var2()
text
main() routine1() routine2()
data
arrayA arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets
User Address Space
Thread 2 stack
routine2() var1 var2 var3
Thread 1 stack
routine1() var1 var2
text
main() routine1() routine2() ...
data
arrayA arrayB
heap
Stack Pointer
Prgrm. Counter
Registers
Stack Pointer
Prgrm. Counter
Registers
Process ID
User ID
Group ID
Files
Locks
Sockets

Basic concepts of threads

Concurrent thread execution:



Basic concepts of threads

Notes:

1. **Thread context is much smaller. Hence thread context switch is much faster than the process context switch.**

2. **Threads are not organized into a rigid parent-child hierarchy. They are peer threads. A thread can kill any of its peers and wait for any of its peers.**

3. **Each peer can read and write the same shared data.**

Thread control

Example hello world:

```
 1 #include "csapp.h"
 2
 3 void *thread(void *vargp);
 4
 5 int main()
 6 {
 7     pthread_t tid;
 8
 9     Pthread_create(&tid, NULL, thread, NULL);
10     Pthread_join(tid, NULL);
11     exit(0);
12 }
13
14 /* thread routine */
15 void *thread(void *vargp)
16 {
17     printf("Hello, world!\n");
18     return NULL;
19 }
```

9:The main thread creates a new peer thread by calling the pthread_create function

10: The main thread waits for the newly created thread to terminate.

11:The main thread terminates itself and the entire process by calling exit

Thread Creation

```
#include <pthread.h>
typedef void *(func)(void *);

int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
                                              returns: 0 if OK, non-zero on error
```

The pthread_create function creates a new thread and runs the *thread routine* f in the context of the new thread and with an input argument of arg.

```
#include <pthread.h>

pthread_t pthread_self(void);
```

The new thread can determine its own thread ID by calling the pthread_self function.

Thread Termination

A thread terminates in one of the following ways:

1. The thread terminates *implicitly* when its top-level thread routine returns.

2. The thread terminates *explicitly* by calling the pthread exit function, which returns a pointer to the return value thread return.

If the main thread calls pthread exit, it waits for all other peer threads to terminate, and then terminates the main thread and the entire process with a return value of thread return.

```
#include <pthread.h>

int pthread_exit(void *thread_return);
```

Reaping threads

pthread join

blocks until thread tid terminates,

assigns the (void *) pointer returned by the thread routine to the location pointed to by thread return, and then

***reaps* any memory resources held by the terminated thread.**

```
#include <pthread.h>

int pthread_join(pthread_t tid, void **thread_return);
```

Detaching threads

At any point in time, a thread is *joinable* or *detached*.

A joinable thread can be reaped and killed by other threads. Its memory resources (such as the stack) are not freed until it is reaped by another thread.

In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.

```
#include <pthread.h>

int pthread_detach(pthread_t tid);
```

The pthread detach function detaches the joinable thread tid. Threads can detach themselves by calling pthread detach with an argument of pthread self().

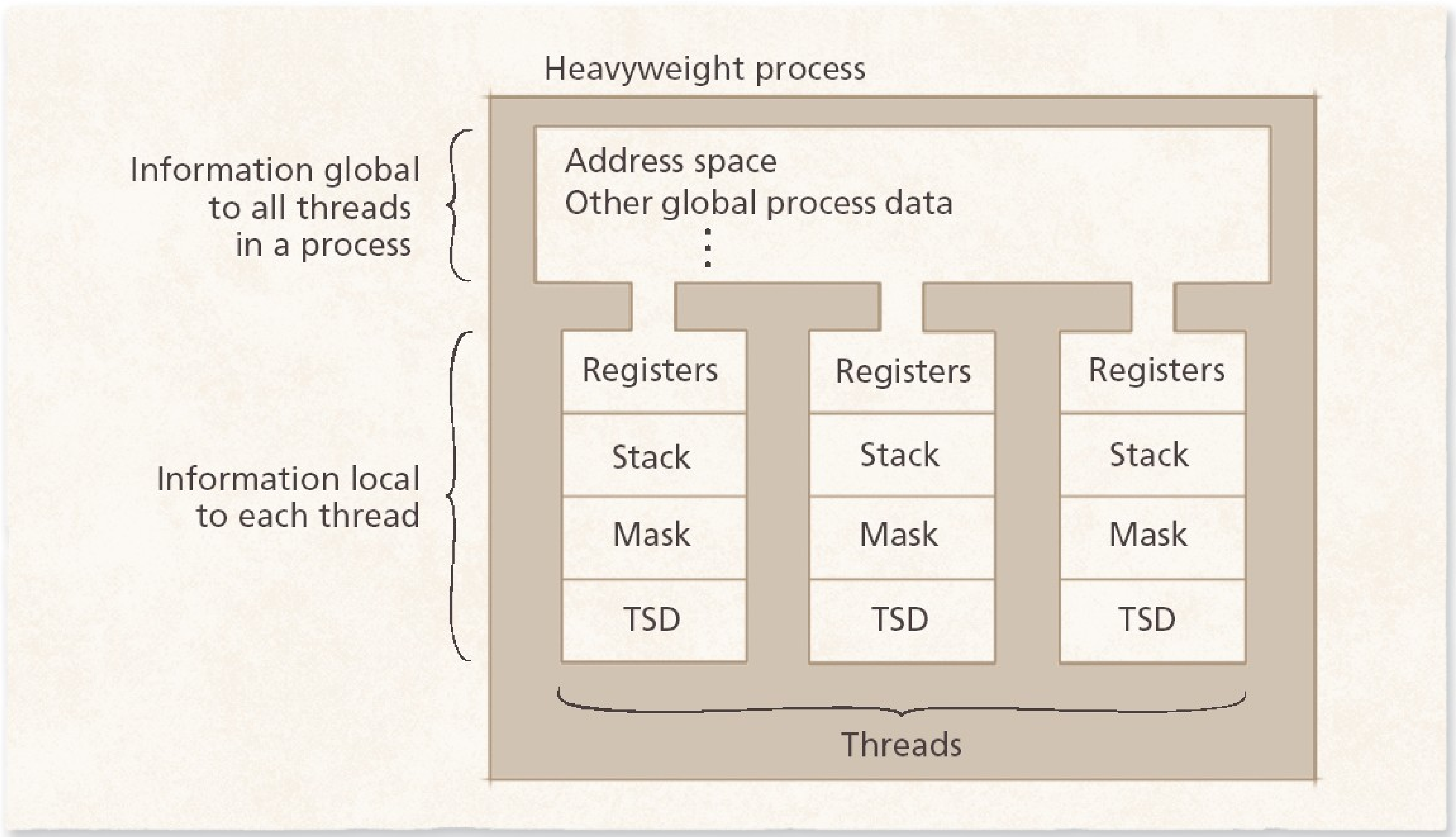
Detaching threads

For example,
a high-performance Web server
- create a new peer thread each time it receives a connection request from a Web browser.
- each connection is handled independently by a separate thread, it is unnecessary and indeed undesirable for the server to explicitly wait for each peer thread to terminate.

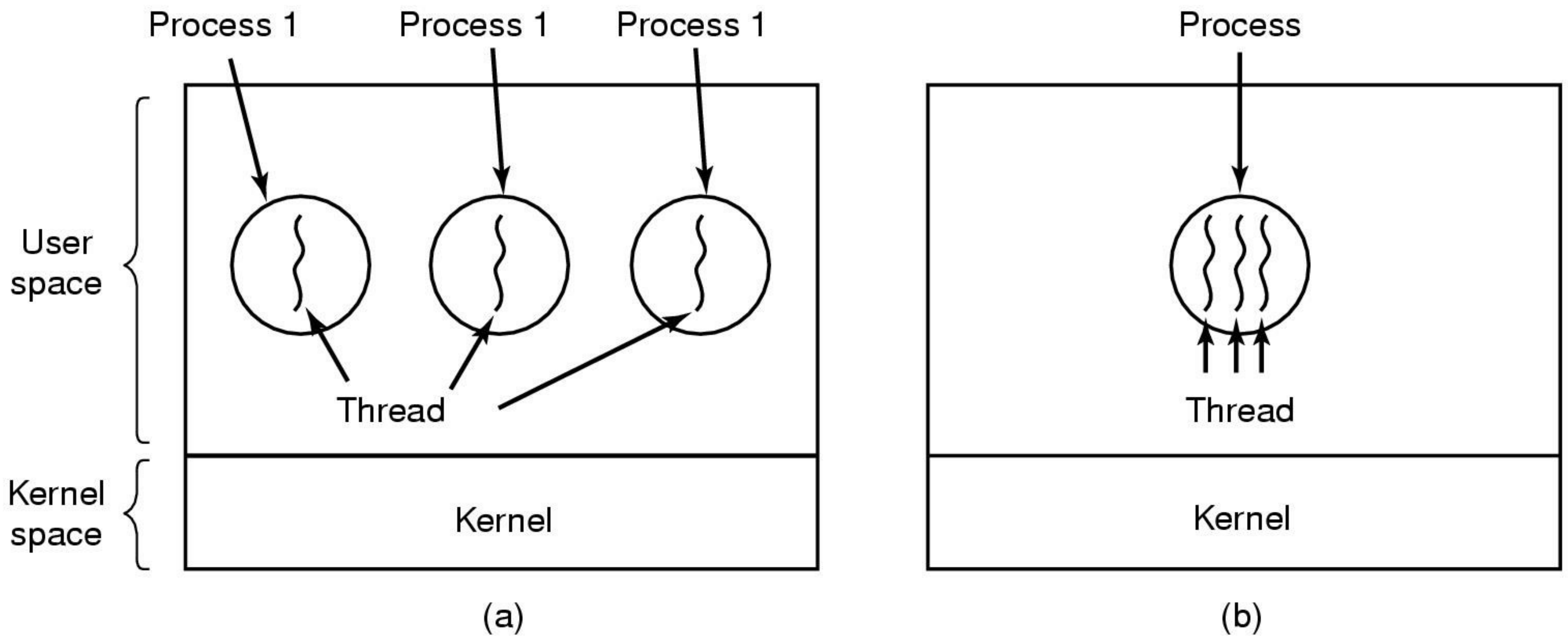
In this case, each peer thread should detach itself before it begins processing the request so that its memory resources can be reclaimed after it terminates.

```c
#include "csapp.h"
#define N 2

char **ptr;  /* global variable */

void *thread(void *vargp);

int main()
{
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;

    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}

void *thread(void *vargp)
{
    int myid = (int)vargp;
    static int cnt = 0;

    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
}
```

Threads


Heavyweight process
Information global to all threads in a process
Address space
Other global process data
⋮
Information local to each thread
Registers
Stack
Mask
TSD
Registers
Stack
Mask
TSD
Registers
Stack
Mask
TSD
Threads

Threads


Process 1
Process 1
Process 1
User space
Thread
Kernel space
Kernel
(a)
Process
Thread
Kernel
(b)

Motivation for Threads

- Threads have become prominent due to trends in
  - **Software design**
    - **More naturally expresses inherently parallel tasks**
  - **Performance**
    - **Scales better to multiprocessor systems**
  - **Cooperation**
    - **Shared address space incurs less overhead than IPC**

Motivation for Threads

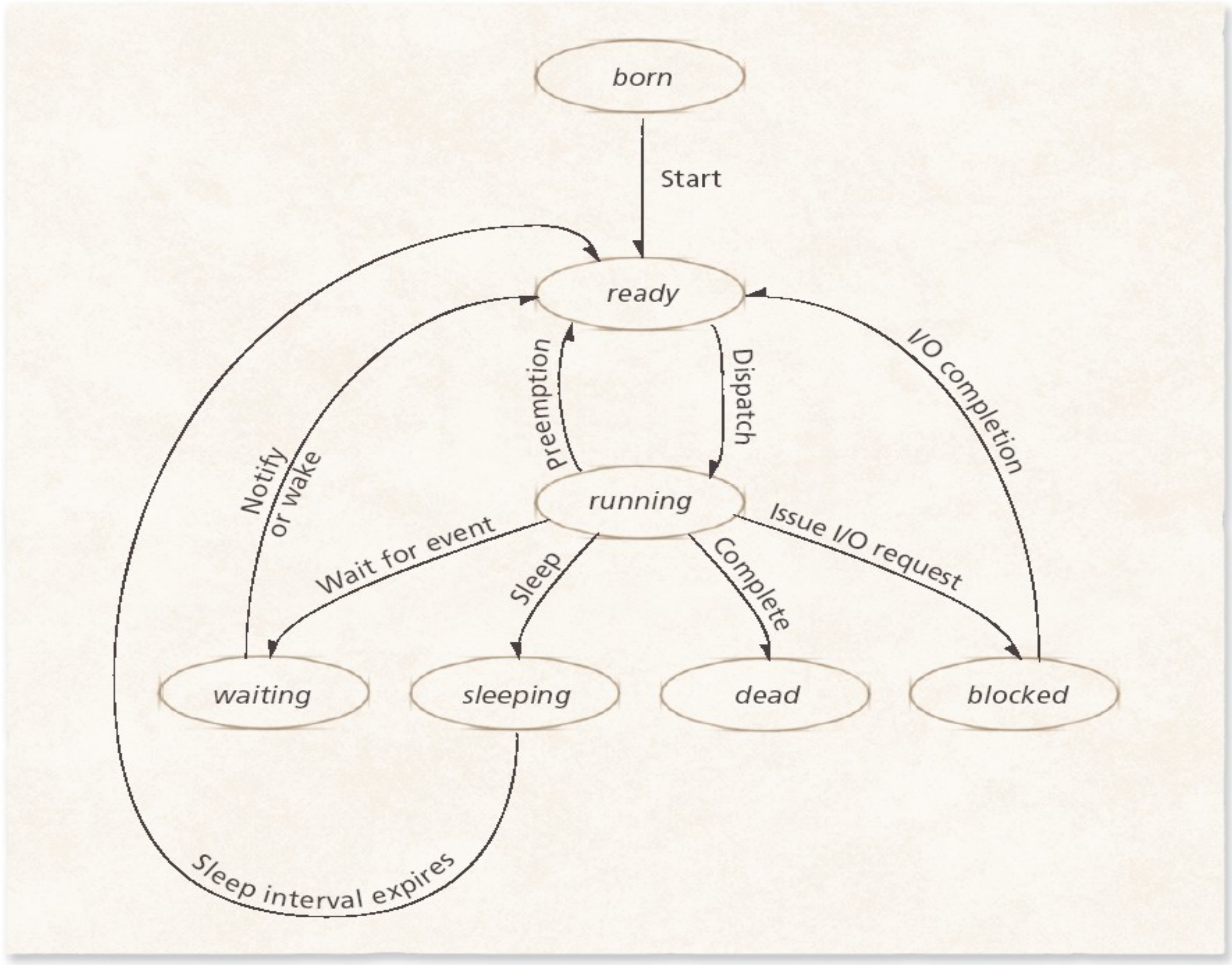
- Each thread transitions among a series of discrete thread states

- Thread creation does not require operating system to initialize resources that are shared between parent processes and its threads
  - Reduces overhead of thread creation and termination compared to process creation and termination

Thread States: Life Cycle of a Thread

- Thread states
  - *Born* state
  - *Ready* state (*runnable* state)
  - *Running* state
  - *Dead* state
  - *Blocked* state
  - *Waiting* state
  - *Sleeping* state
    - Sleep interval specifies for how long a thread will sleep

Thread States: Life Cycle of a Thread



Thread life cycle.

Thread Operations

- Threads and processes have common operations
  - Create
  - Exit (terminate)
  - Suspend
  - Resume
  - Sleep
  - Wake

Threading Models

- Three most popular threading models
  - User-level threads
  - Kernel-level threads
  - Combination of user- and kernel-level threads

User-level Threads

- User-level threads perform threading operations in user space
  - Threads are created by runtime libraries that cannot execute privileged instructions or access kernel primitives directly

User-level Threads

- User-level thread implementation
  - Many-to-one thread mappings
    - Operating system maps all threads in a multithreaded process to single execution context

User-level Threads

Advantages

- User-level libraries can schedule its threads to optimize performance
- Synchronization performed outside kernel, avoids context switches
- More portable

User-level Threads

Disadvantage

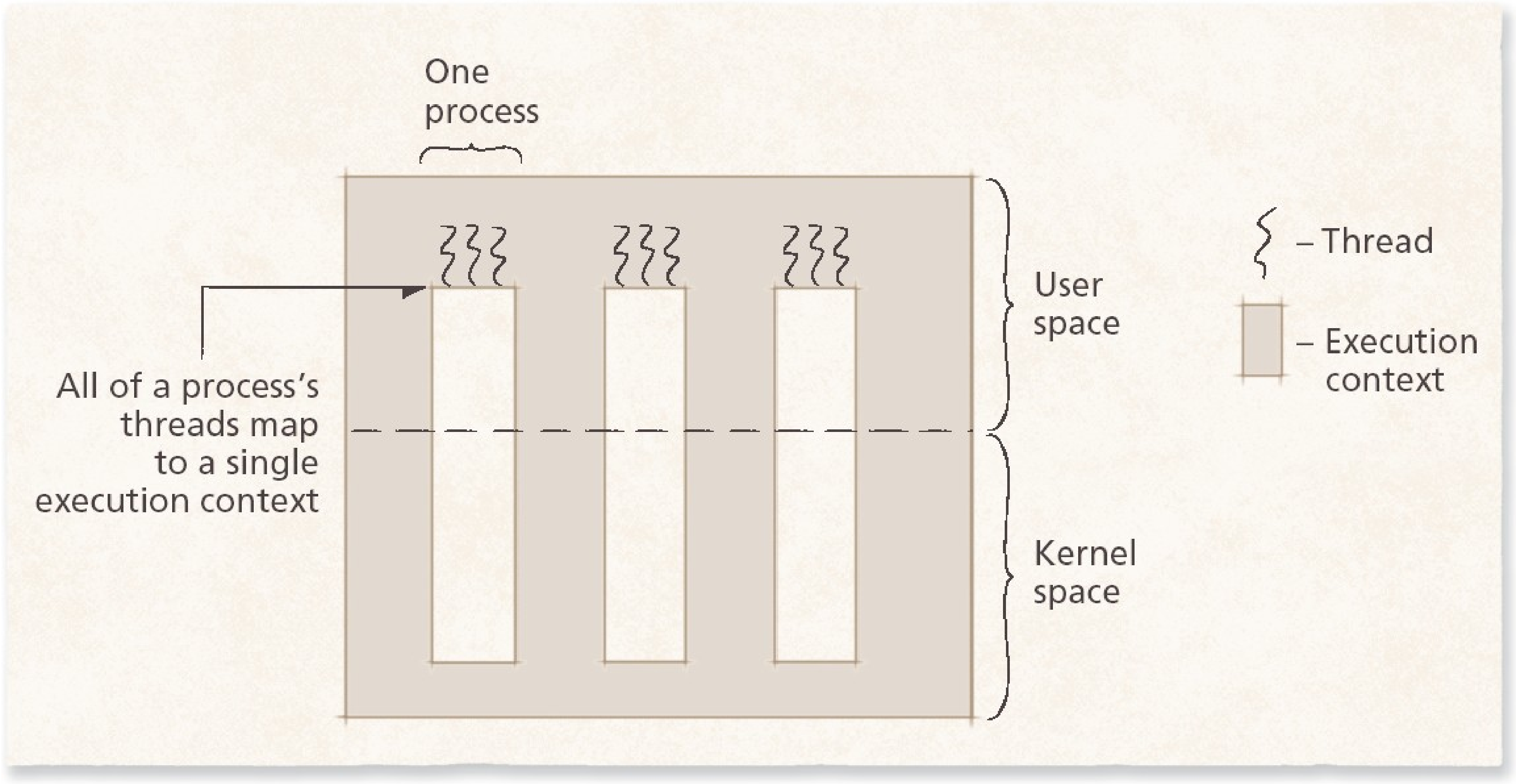
- –Kernel views a multithreaded process as a single thread of control
  - »Can lead to suboptimal performance if a thread issues I/O
  - »Cannot be scheduled on multiple processors at once

User-level Threads

Entire process blocks when any of its threads requests a blocking I/O operation.

User-level Threads

User-level threads.


One process
All of a process's threads map to a single execution context
User space
Kernel space
– Thread
– Execution context

Kernel-level Threads

Kernel-level threads attempt to address the limitations of user-level threads by mapping each thread to its own execution context

Kernel-level Threads

– Kernel-level threads provide a one-to-one thread mapping

- Advantages:

  Increased scalability,

  interactivity, and

  throughput

- Disadvantages:

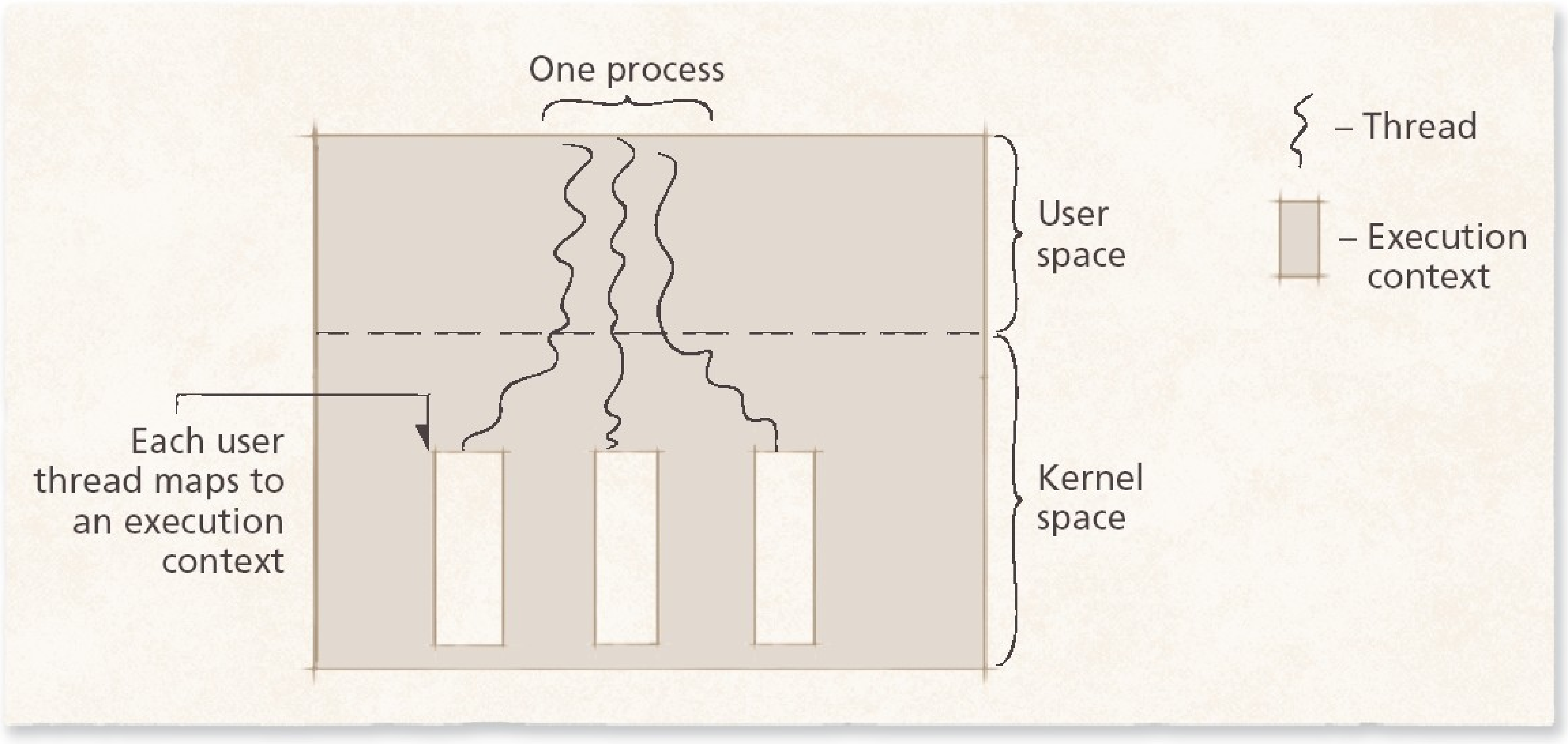
  Overhead due to context switching and reduced portability due to OS-specific APIs

Kernel-level Threads

- Kernel-level threads are not always the optimal solution for multithreaded applications

Kernel-level Threads

Kernel-level threads.



Combining User- and Kernel-level Threads

- The combination of user- and kernel-level thread implementation
  - Many-to-many thread mapping (*m*-to-*n* thread mapping)
    - Number of user and kernel threads need not be equal
    - Can reduce overhead compared to one-to-one thread mappings by implementing thread pooling

Combining User- and Kernel-level Threads

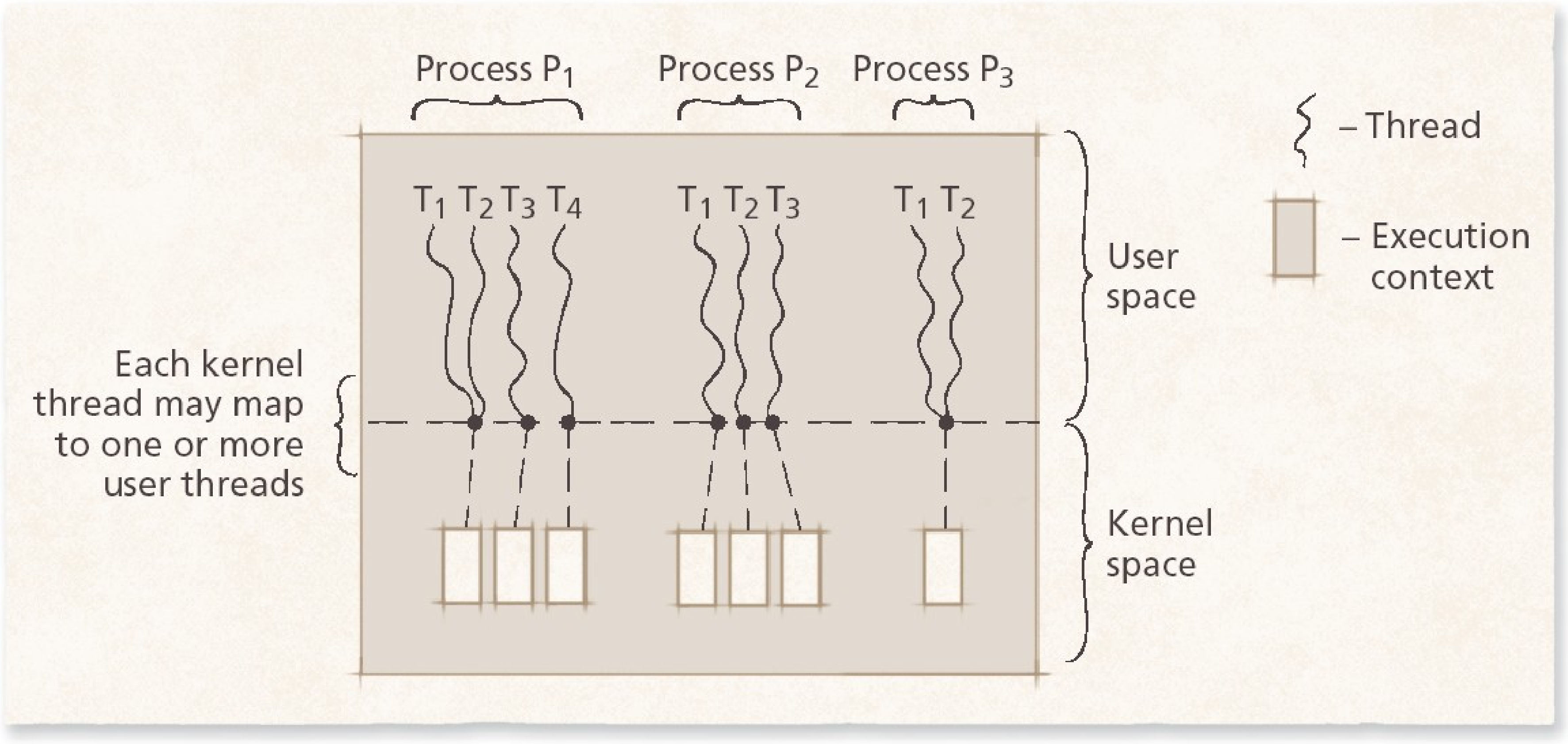
- Worker threads
  - Persistent kernel threads that occupy the thread pool
  - Improves performance in environments where threads are frequently created and destroyed
  - Each new thread is executed by a worker thread

Combining User- and Kernel-level Threads

- Scheduler activation
  - Technique that enables user-level library to schedule its threads
  - Occurs when the operating system calls a user-level threading library that determines if any of its threads need rescheduling

Combining User- and Kernel-level Threads

Hybrid threading model.



Thread Termination

- Thread termination (cancellation)
  - Differs between thread implementations
  - Prematurely terminating a thread can cause subtle errors in processes because multiple threads share the same address space
  - Some thread implementations allow a thread to determine when it can be terminated to prevent process from entering inconsistent state

POSIX and Pthreads

- Threads that use the POSIX threading API are called Pthreads
  - POSIX states that processor registers, stack and signal mask are maintained individually for each thread
  - POSIX specifies how operating systems should deliver signals to Pthreads in addition to specifying several thread-cancellation modes

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)

Threads: Benefits

- Responsiveness
- Resource Sharing
- Economy
- Utilization of MP Architectures

Threading models

- Three most popular threading models
  - User-level threads
  - Kernel-level threads
  - Combination of user- and kernel-level threads

User Threads

- Thread management done by user-level threads library

- Three primary thread libraries:
  - POSIX Pthreads
  - Win32 threads
  - Java threads

Kernel Threads

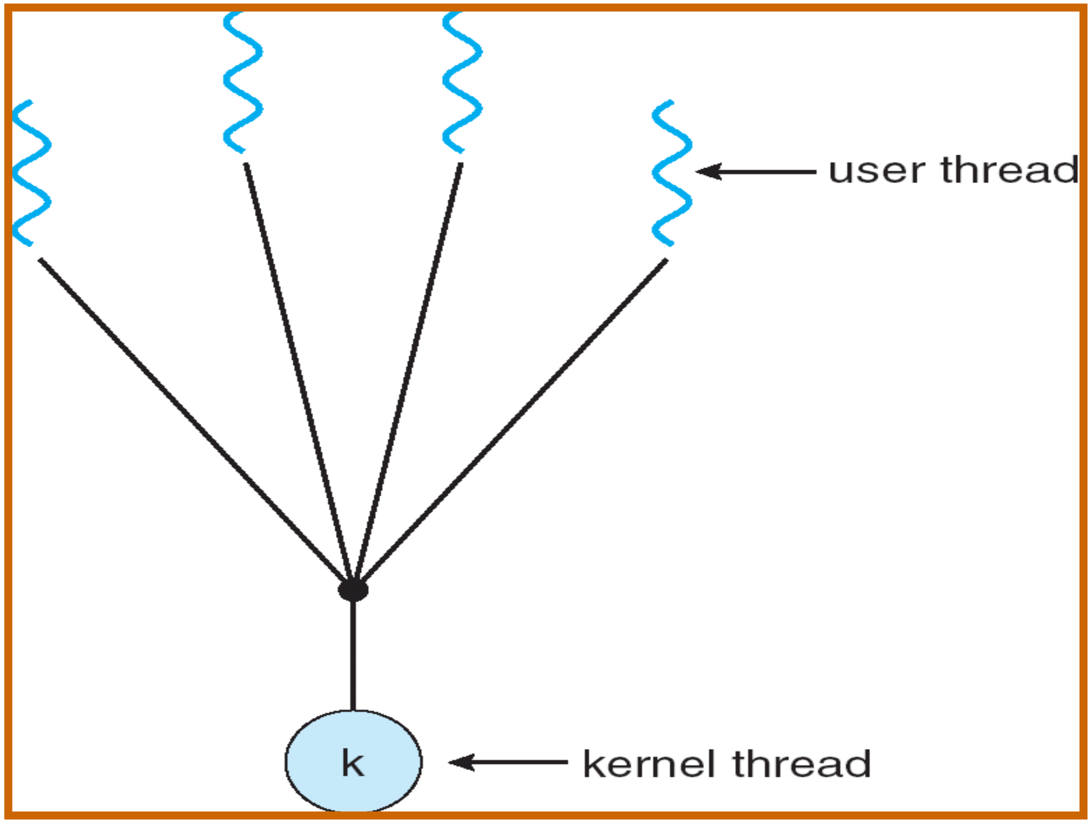
- Supported by the Kernel
- Examples
  - Windows XP/2000
  - Solaris
  - Linux
  - Tru64 UNIX
  - Mac OS X

Multi-threading Models

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

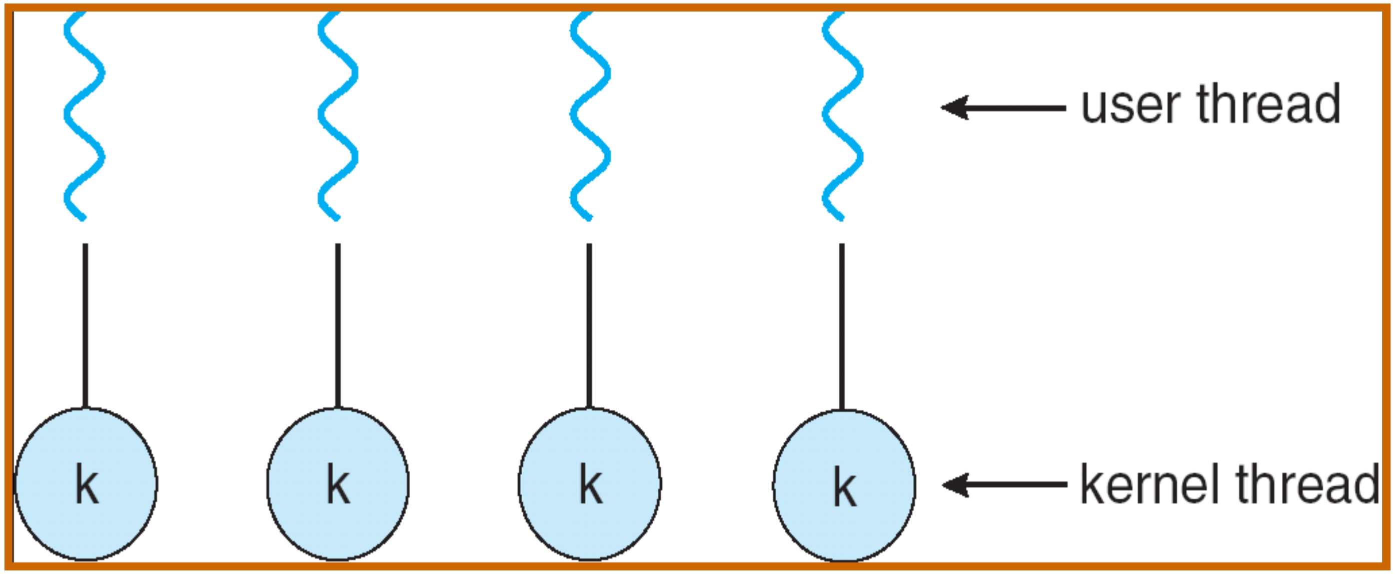
Many-to-One

- Many user-level threads mapped to single kernel thread
- One thread may block the processs
- Does not reap the benefit of multiprocessors



One-to-One

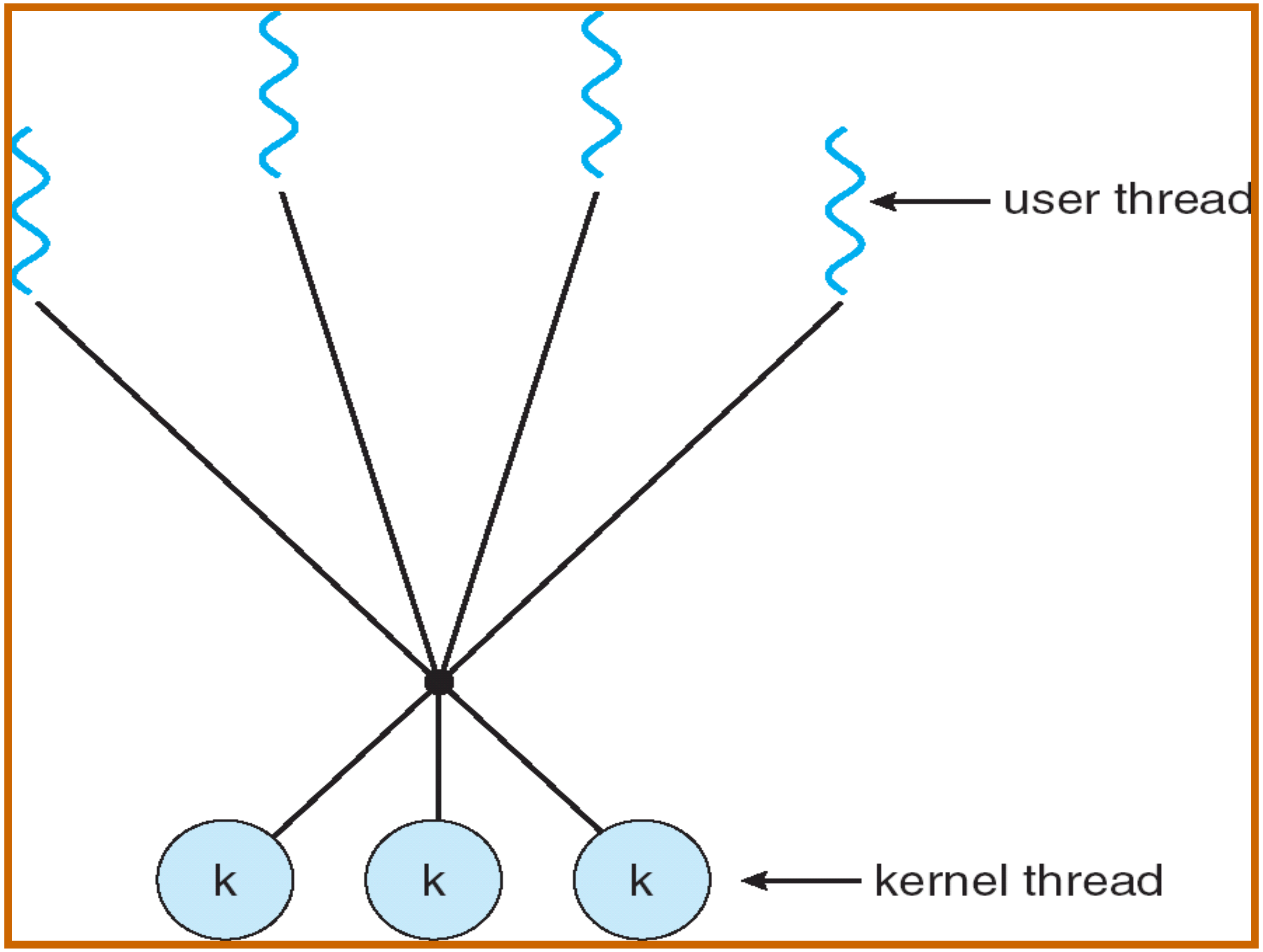
- **Each user-level thread maps to kernel thread**
- **One kernel thread per user thread**
- **Multiple threads run in parallel on multiprocessors**
- **Linux, Windows NT/XP, Solaris 9 or later**



Many-to-Many

- **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 prior to version 9**
- **Windows NT/2000 with the *ThreadFiber* package**

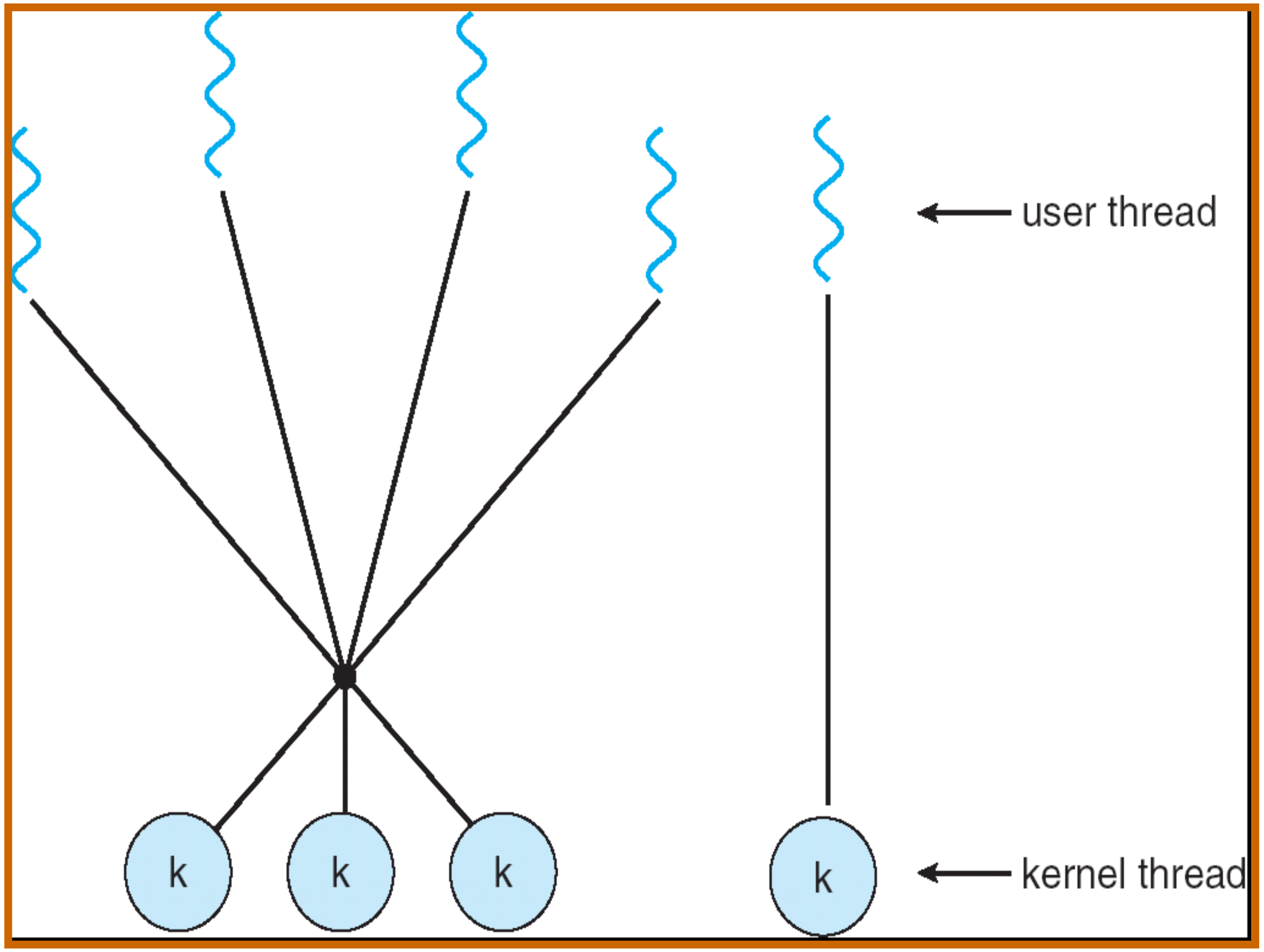
Many-to-Many



Two-level model

- Similar to M:M, except that it allows a user thread to be **bound** to kernel thread
- Examples
  - IRIX
  - HP-UX
  - Tru64 UNIX
  - Solaris 8 and earlier

Two-level model



Solaris Threads

- Process includes the user’s address space, stack, and process control block
- User-level threads
- Lightweight processes (LWP)
- Kernel threads

Solaris Threads

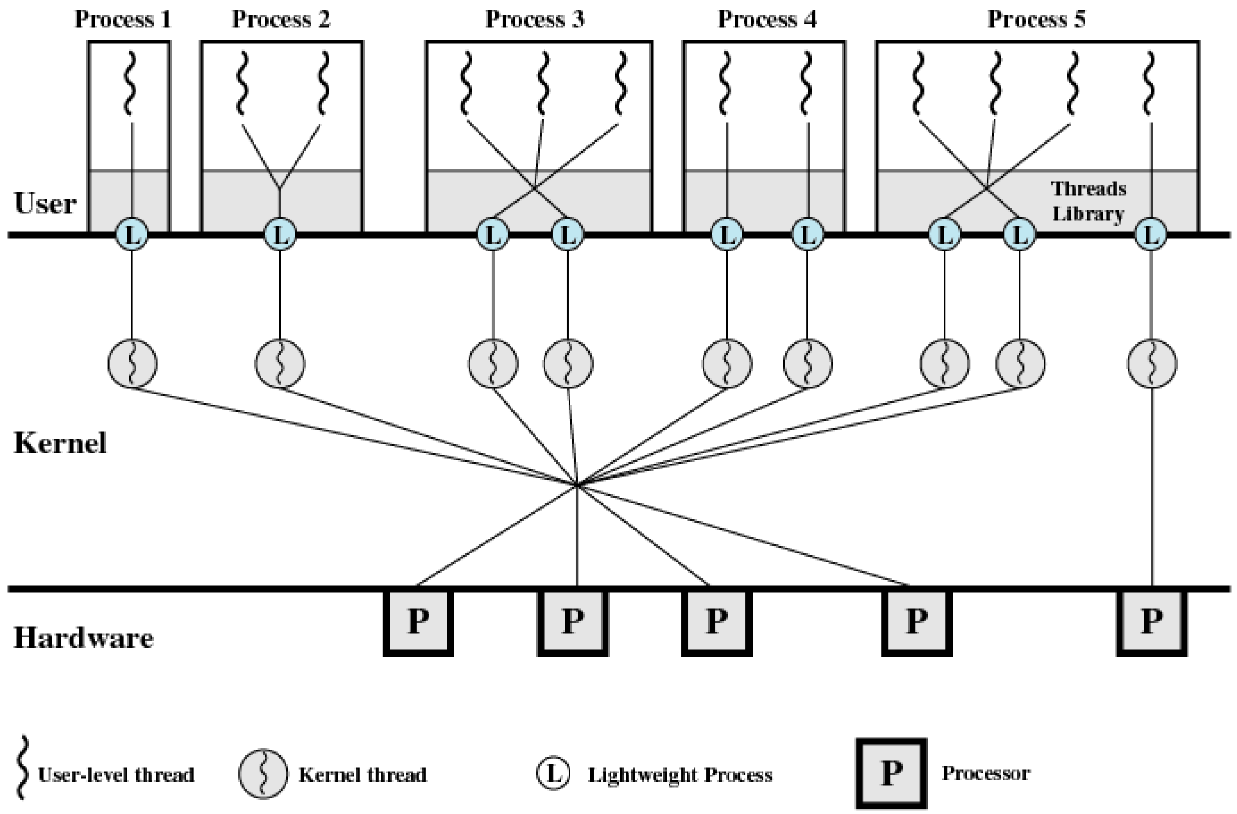


Figure 4.15 Solaris Multithreaded Architecture Example