Operating System Concepts

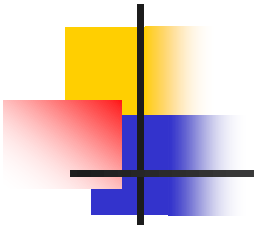
A. Silberschatz,P. B. Galvin, and G. Gagne, 10th edition

Chapter 4: Threads & Concurrency

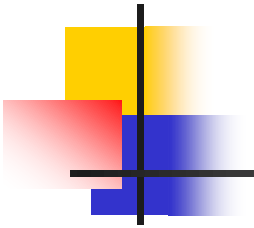
Mei-Ling Chiang, Professor

Dept. of Information Management

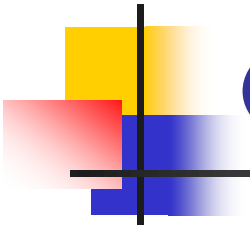
National Chi-Nan University, Taiwan, ROC



- **A process is defined by the resources it uses and by the location at which it is executing.**
- **Processes serve 4 roles:**
  - Unit of resource allocation
  - Memory context
  - Program that is being executed
  - Unit of activity
- **What do you do if your program wants to do more than one thing at a time ?**

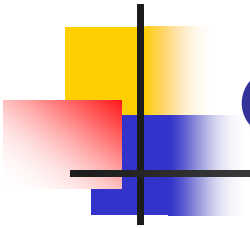


- The process model introduced in Chapter 3 assumed that a process was an executing program with a single thread of control.
- Modern OSes now provide features for a process to contain multiple threads of control.
- If a process has multiple threads of control, it can perform more than one task at a time.



Chapter Objectives

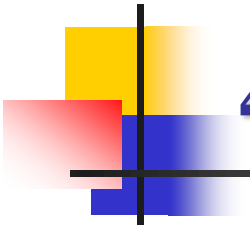
- Identify the basic components of a thread, and contrast threads and processes
- Describe the benefits and challenges of designing multithreaded applications
- Illustrate different approaches to implicit threading including thread pools, fork-join, and Grand Central Dispatch
- Describe how the Windows and Linux OSes represent threads
- Design multithreaded applications using the Pthreads, Java, and Windows threading APIs



Chapter 4: Threads & Concurrency

Outline

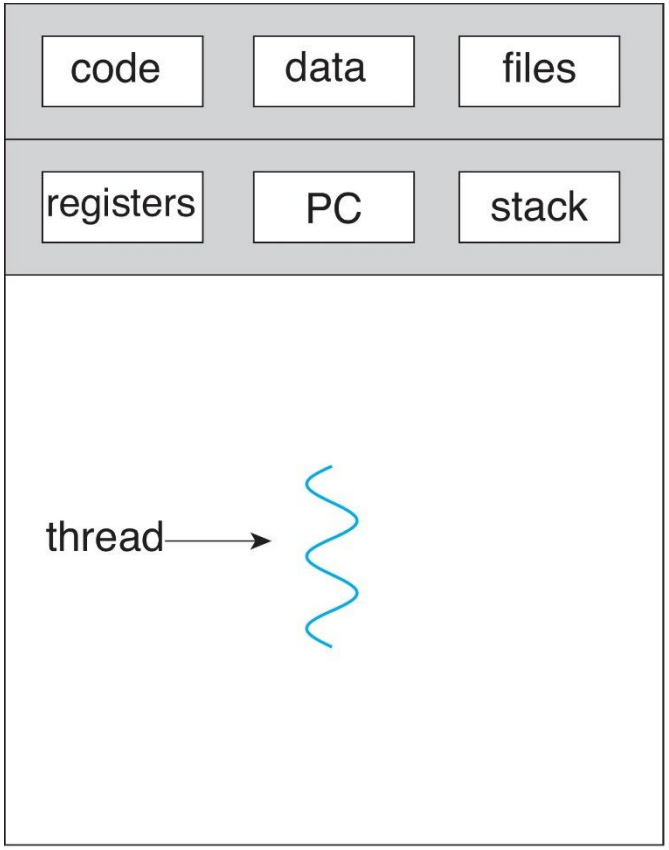
- 4.1 Overview
- 4.2 Multicore Programming
- 4.3 Multithreading Models
- 4.4 Thread Libraries
- 4.5 Implicit Threading
- 4.6 Threading Issues
- 4.7 Operating-System Examples



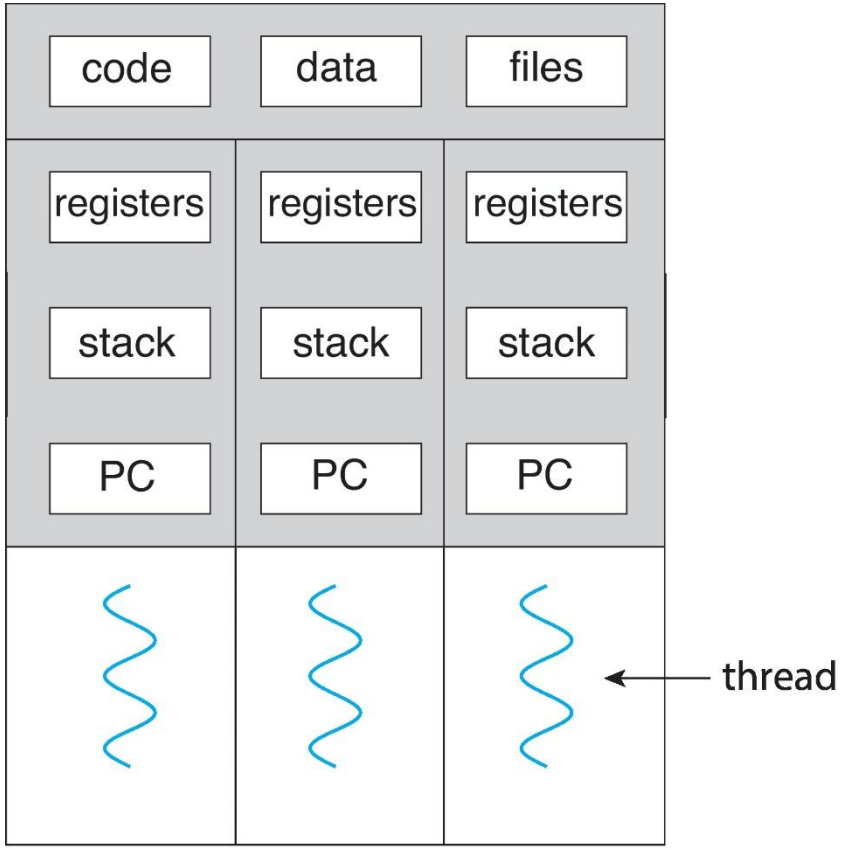
4.1 Overview

- **A *thread* is a basic unit of CPU utilization, comprises**
  - a thread ID, a program counter, a register set, and a stack
- **A thread shares with other threads belonging to the same process its**
  - code section
  - data section
  - other OS resources (open files, signals)
- **A traditional process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.**

Single-threaded and multithreaded processes (Fig. 4.1)



single-threaded process



multithreaded process

4.1.1 Motivation

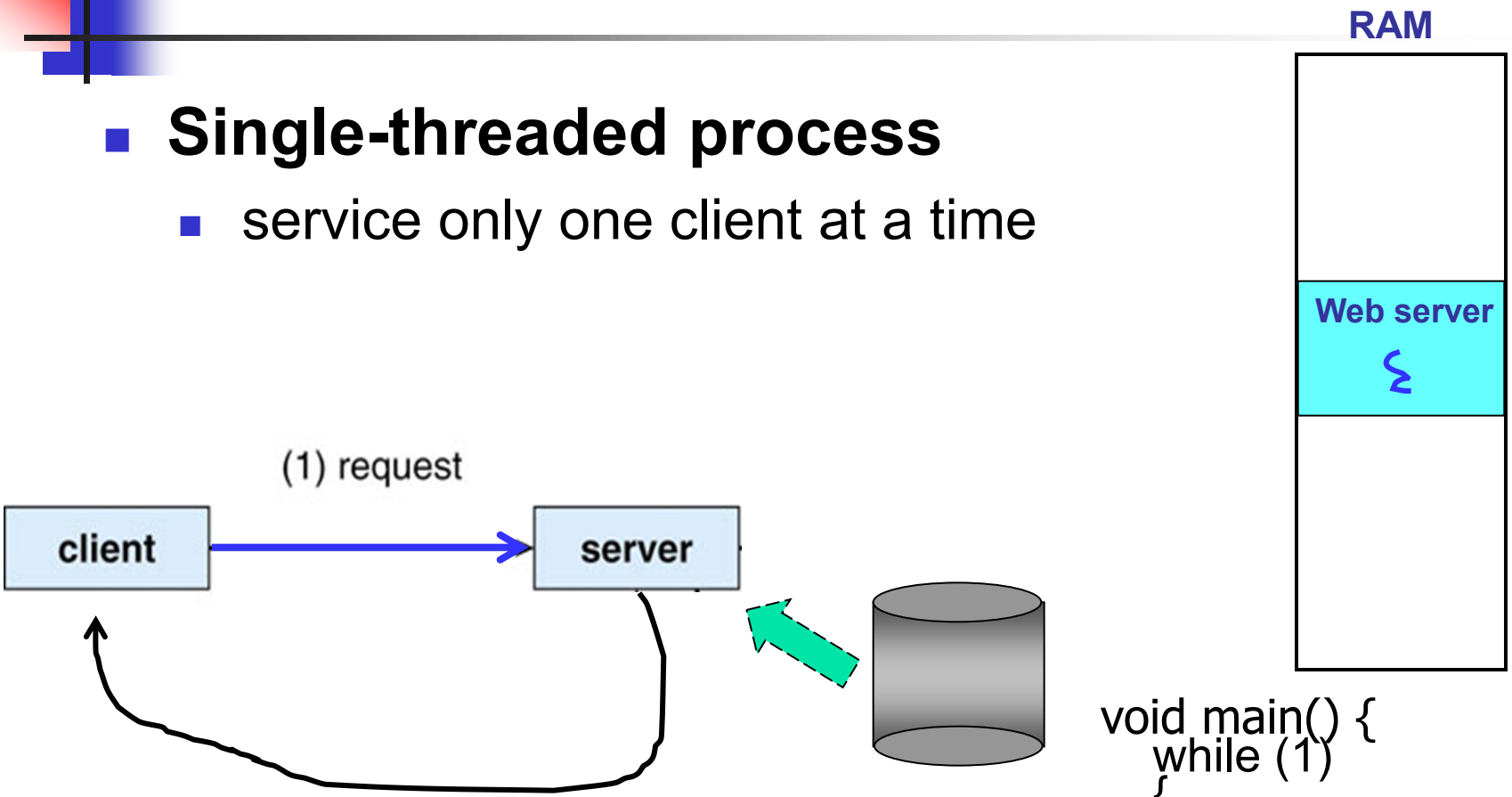
- **Most software applications that run on modern computers and mobile devices are multithreaded.**
- **An application typically is implemented as a separate process with several threads of control.**
- **Example: a web browser**
  - a thread displays images or text while another thread retrieves data from the network
- **Example: a word processor**
  - a thread for displaying graphics, a thread for responding keystrokes from the user, and a thread for performing spelling and grammar checking in the background
- **Example:**
  - applications can be designed and perform several CPU-intensive tasks in parallel on multicore systems.

Motivation (Cont.)

- **In certain situations a single application may be required to perform several similar tasks.**
- **Example: web server**
  - Single-threaded process: service only one client at a time
  - Multiple processes
  - multi-threaded process
    - One process that contains multiple threads to serve the same purpose.
- **Process creation is time consuming and resource intensive.**
- **If the new process will perform the same tasks as the existing process, it is generally more efficient for one process that contains multiple threads to serve the same purpose.**

Example: Web Server

- **Single-threaded process**
  - service only one client at a time

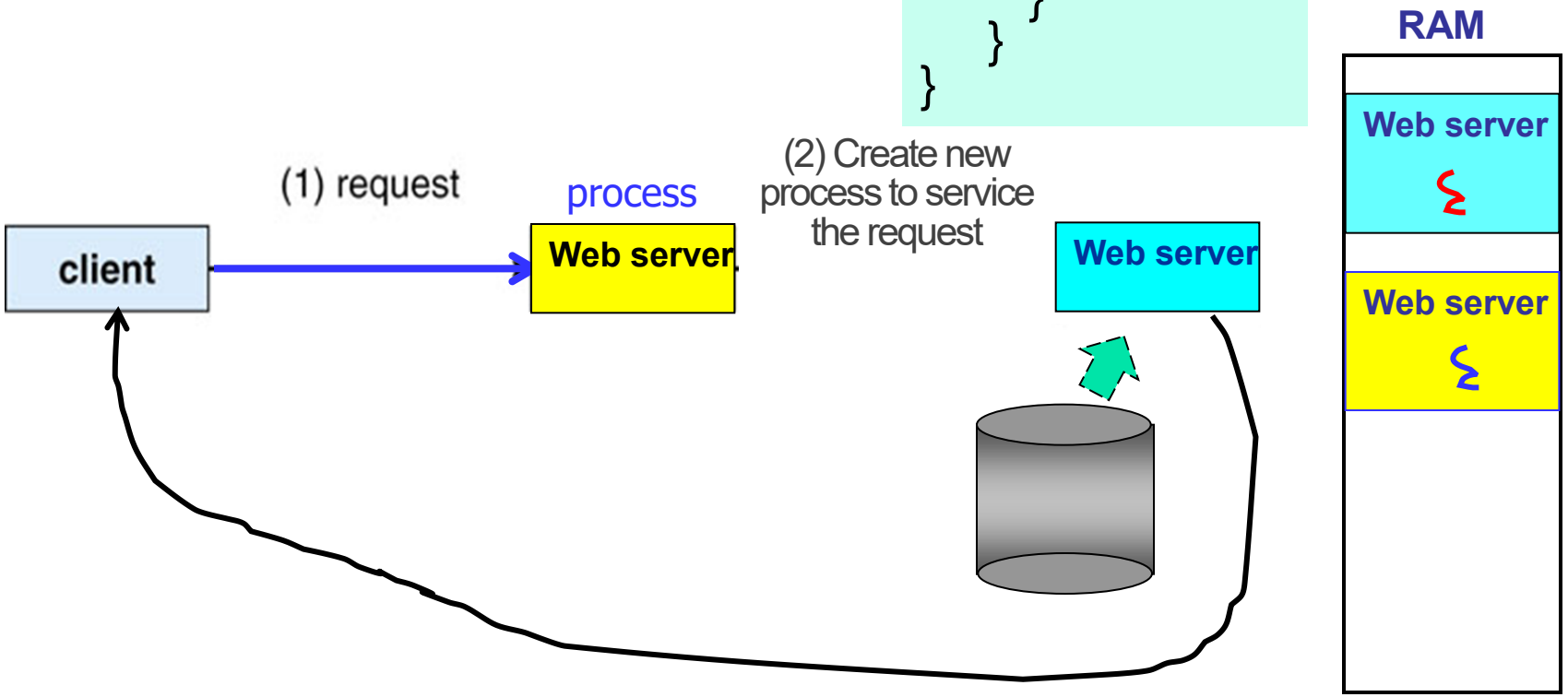



```
void main() {
   while (1)
   {
      receive();
      disk read;
      send();
   }
}
```

Example: Web Server

- **Multiple processes**

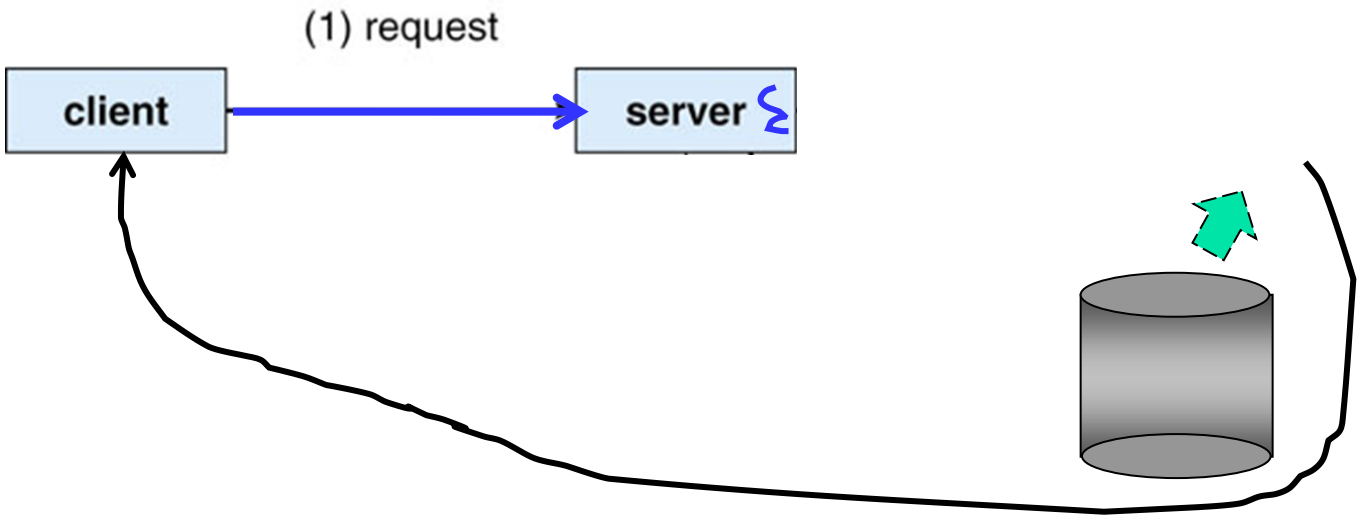
```
void main() {
    while (1)
    {
        receive();
        pid=fork();
        if (pid==0) {
            disk read;
            send();
            exit(0);
        }
    }
}
```

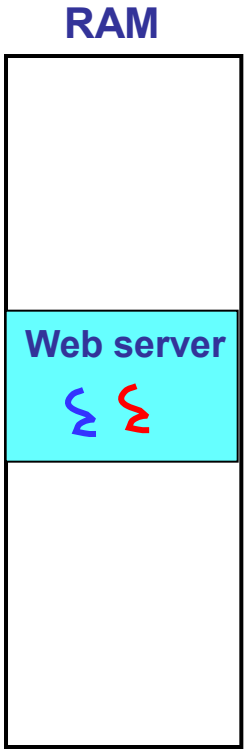


Multithreaded server architecture

- ***Multithreading a web server process***

```
void fun() {
   disk read;
   send();
   thread_exit(0);
}
void main() {
   while (1)
   {
     receive();
     createthread(fun);
   }
}
```





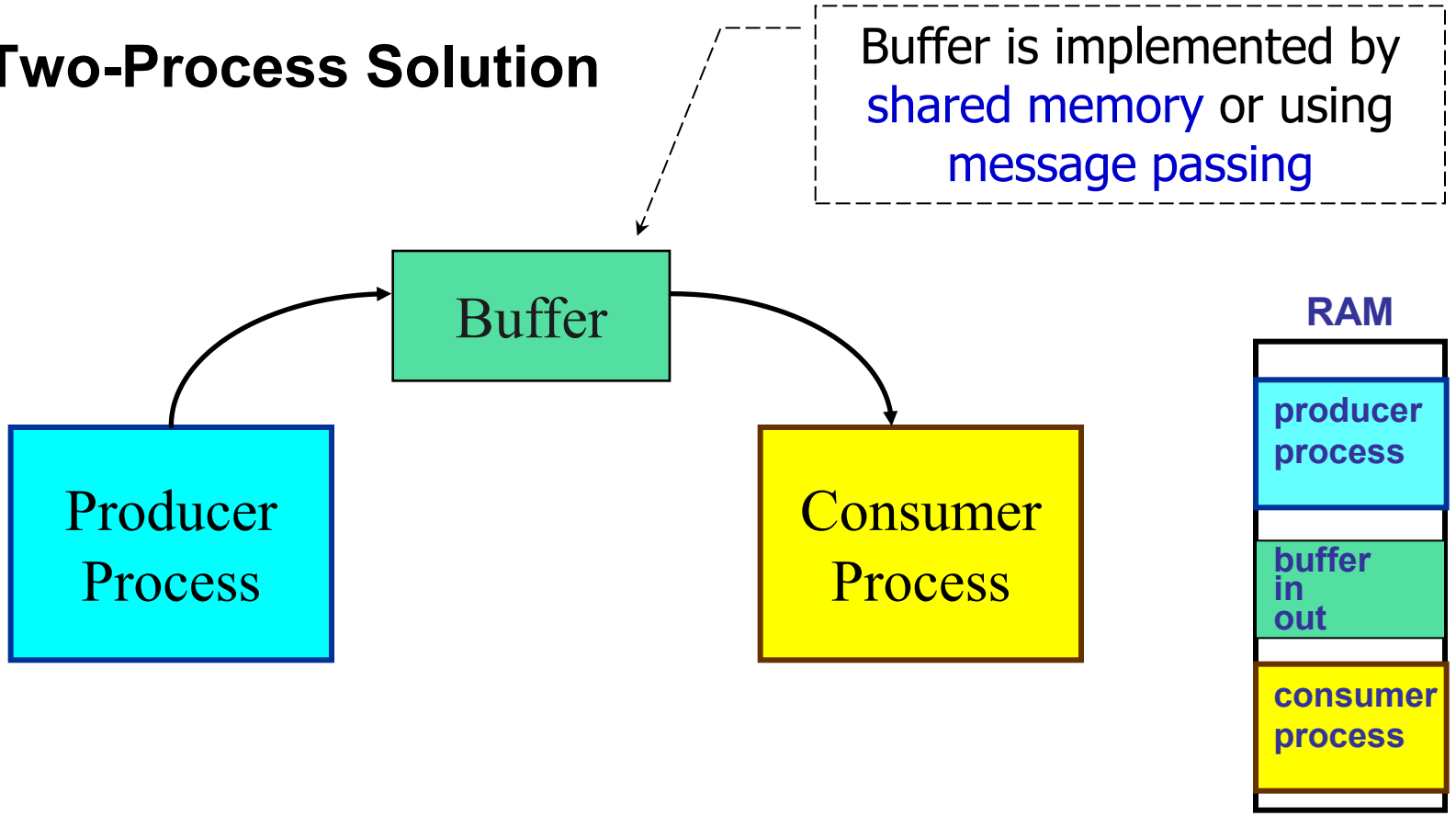
Motivation (Cont.)

- **In a multiple threaded process, while one thread is blocked and waiting, a second thread in the same task can run.**
  - Cooperation of multiple threads in the same job confers higher throughput and improved performance.
  - Applications that require sharing a common buffer (i.e., producer-consumer) benefit from thread utilization.

給予

A Buffering Problem

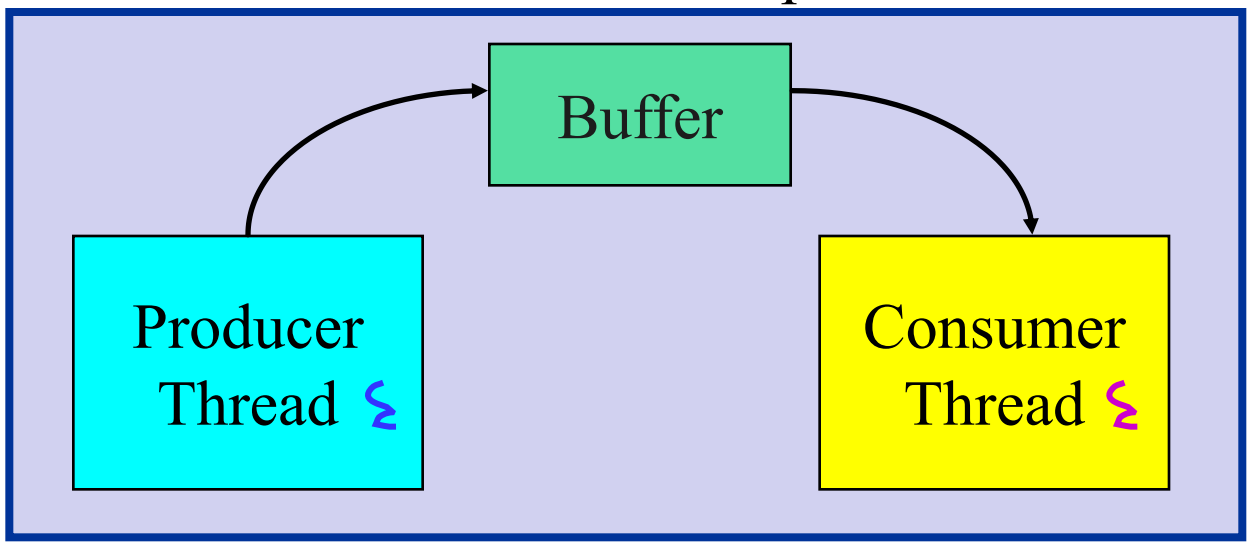
- **Two-Process Solution**

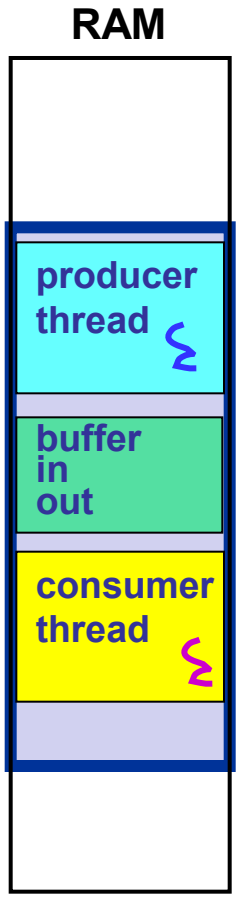


A Buffering Problem

- **Two-Thread Solution**

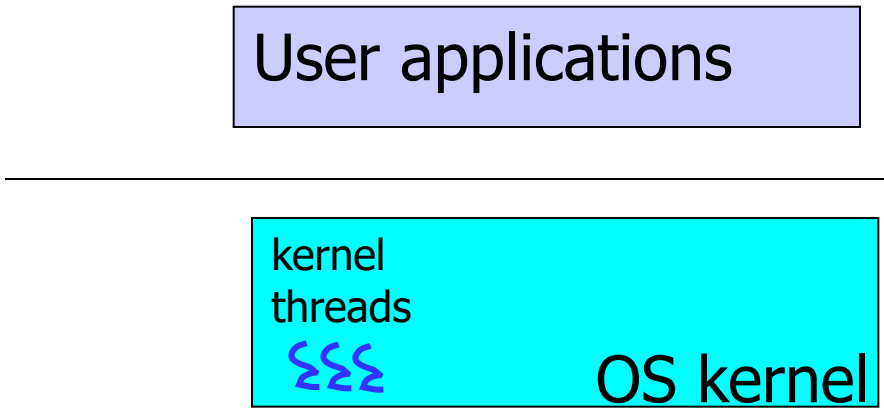
One process with 2 threads



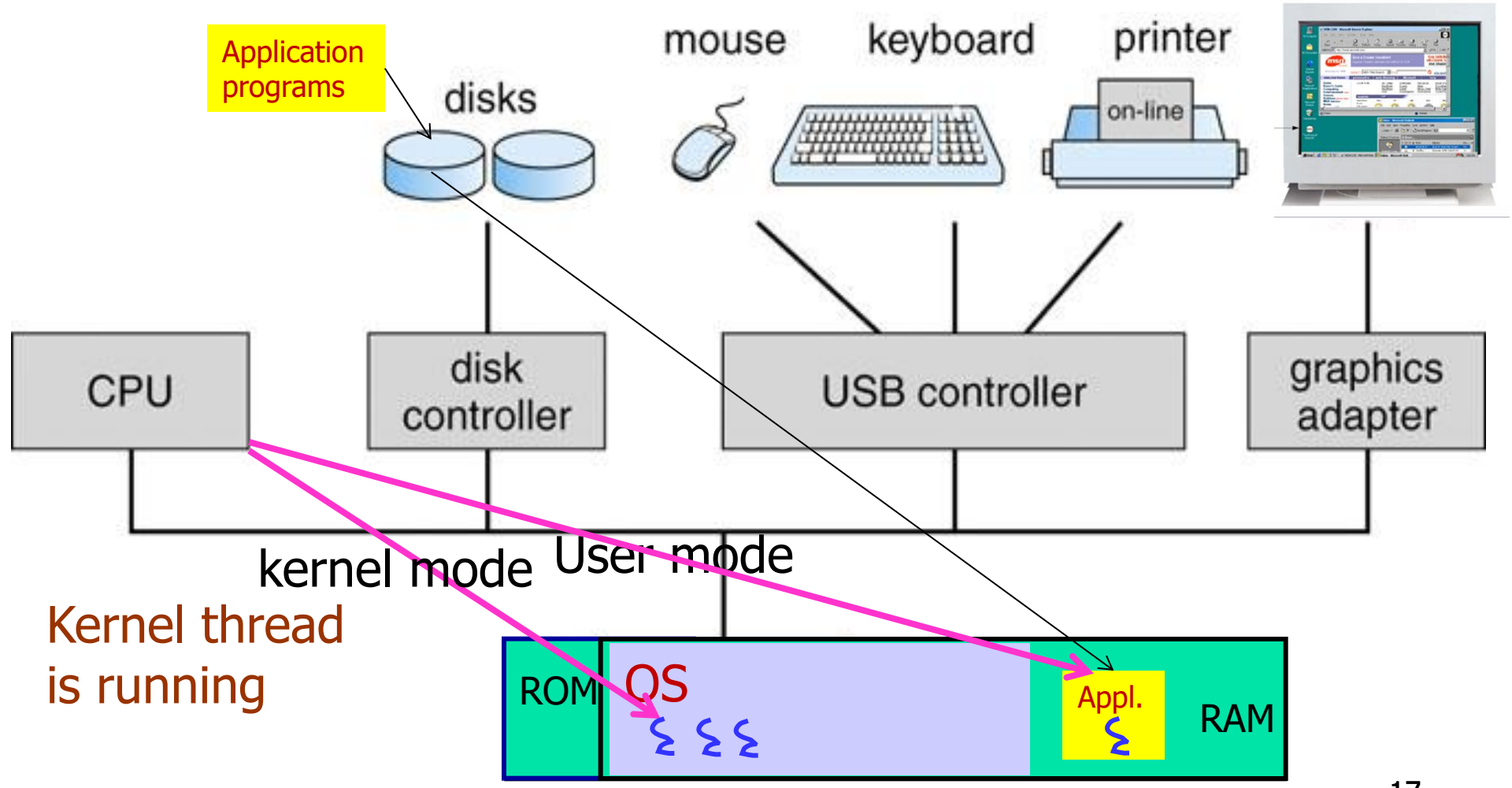


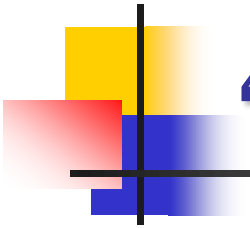
Motivation (Cont.)

- Most OS kernels are also typically multithreaded.
- Example: During boot time on Linux systems, several threads are created. Each thread performs a specific task, such as managing devices, managing memory, or interrupt handling.



When does an OS run?
(5) kernel thread

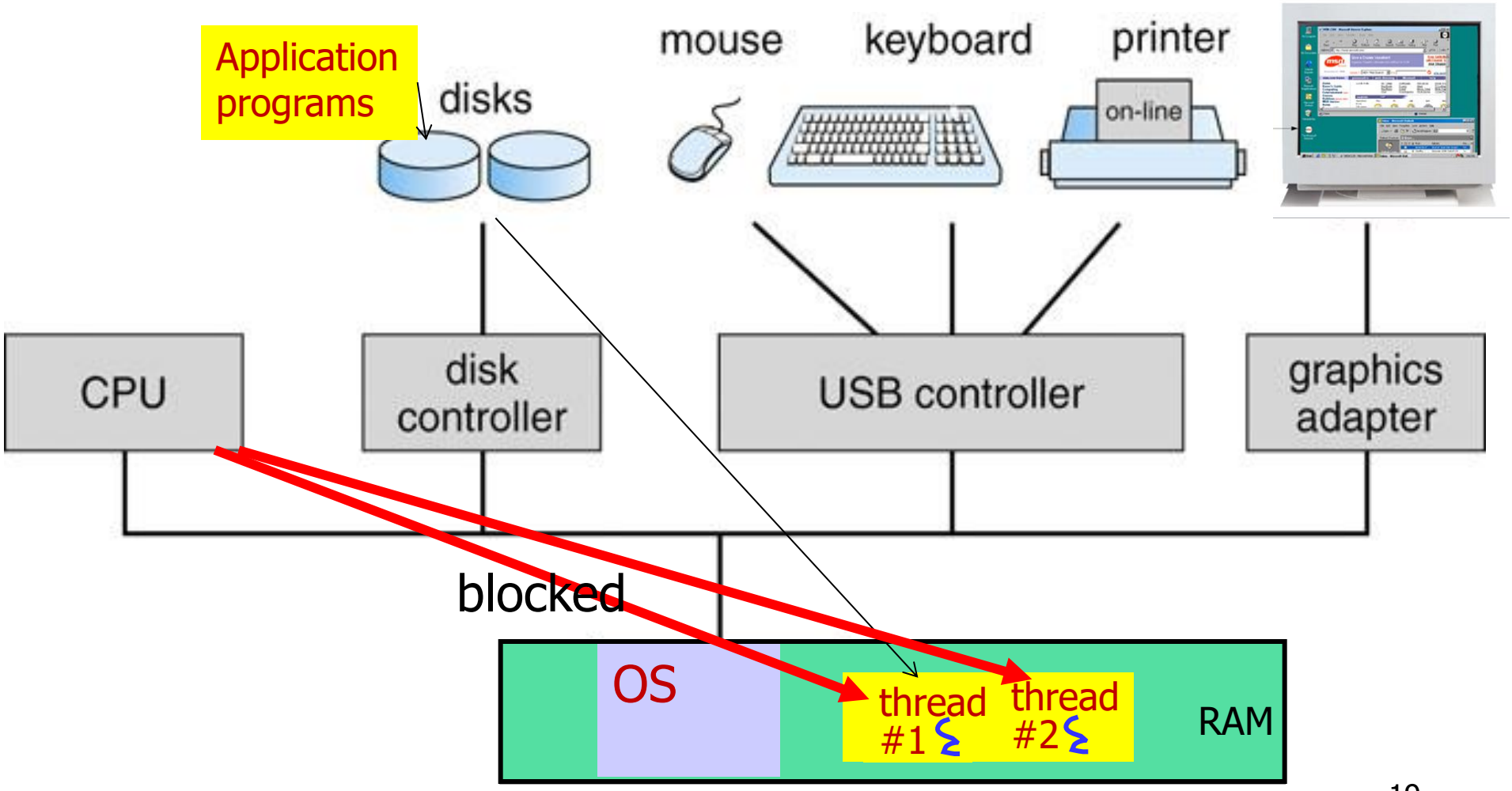




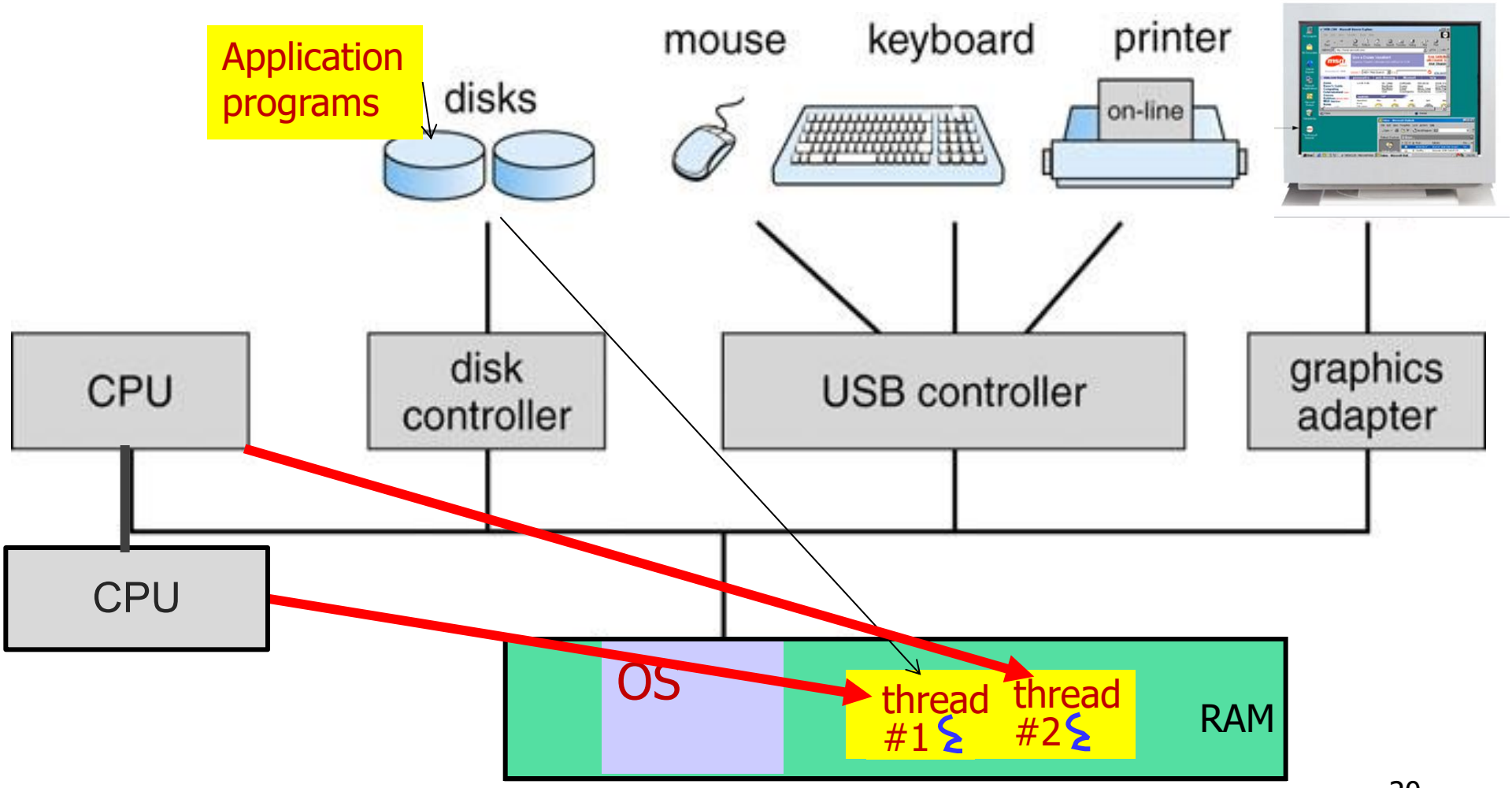
4.1.2 Benefits

- **The benefits of multithreaded programming:**
  - **Responsiveness**
    - Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
  - **Resource Sharing**
    - Threads share the memory and the resources of the process to which they belong.
  - **Economy**
    - More economical to create and context-switch threads.
  - **Scalability**
    - can take advantage of multiprocessor architectures, where threads may be running in parallel.

Increase Responsiveness

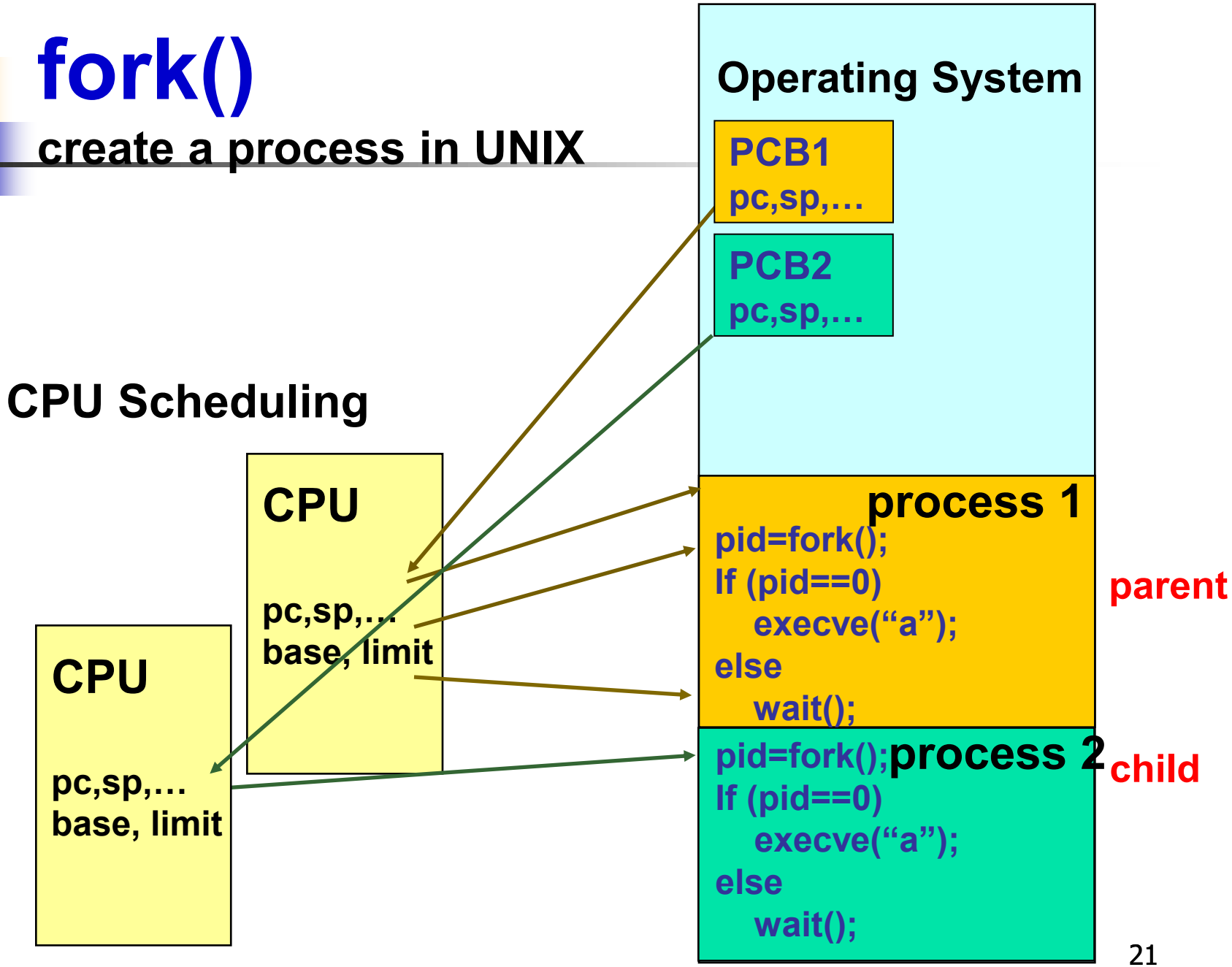


Parallel Execution



fork()

create a process in UNIX

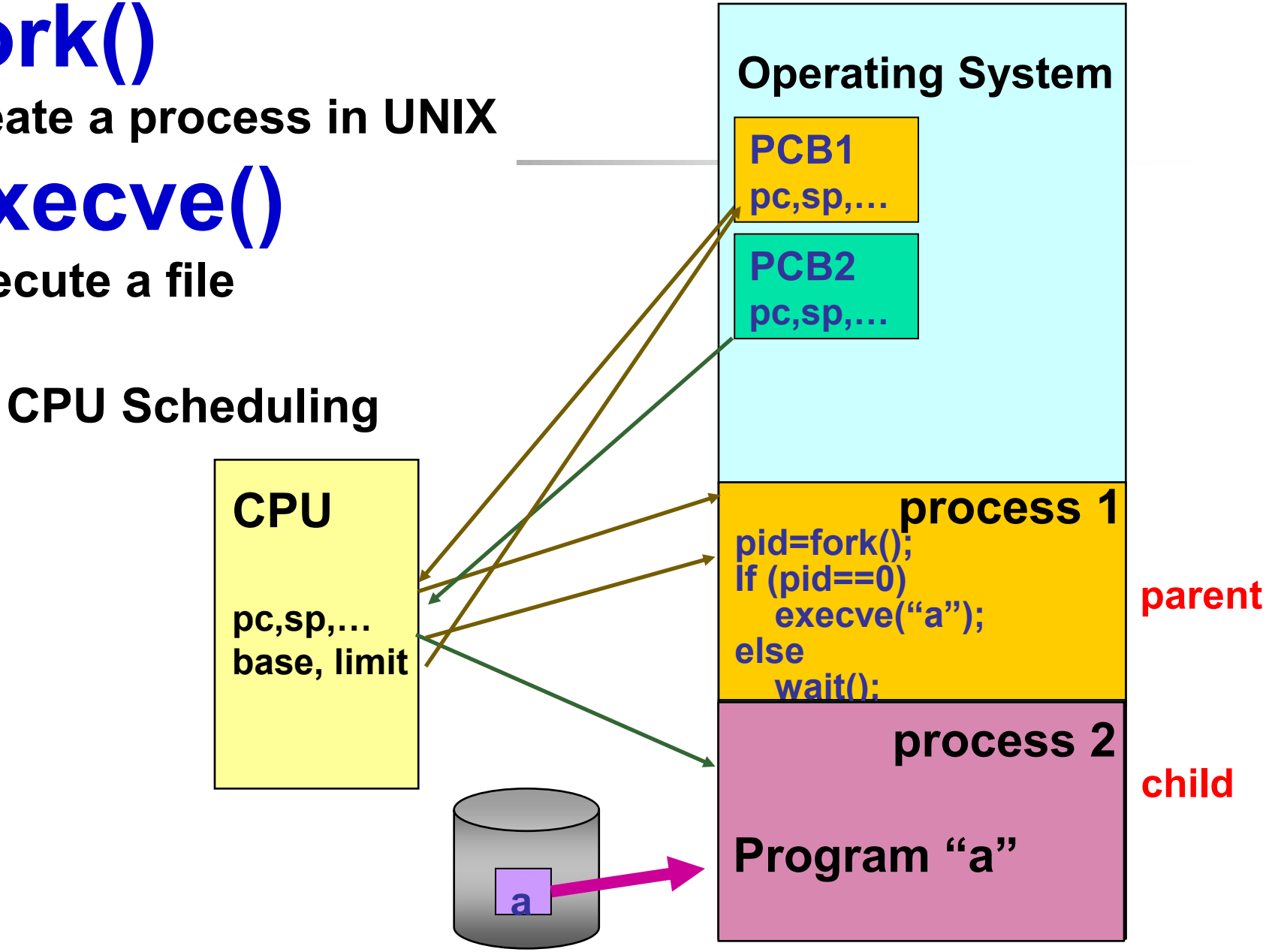


fork()

create a process in UNIX

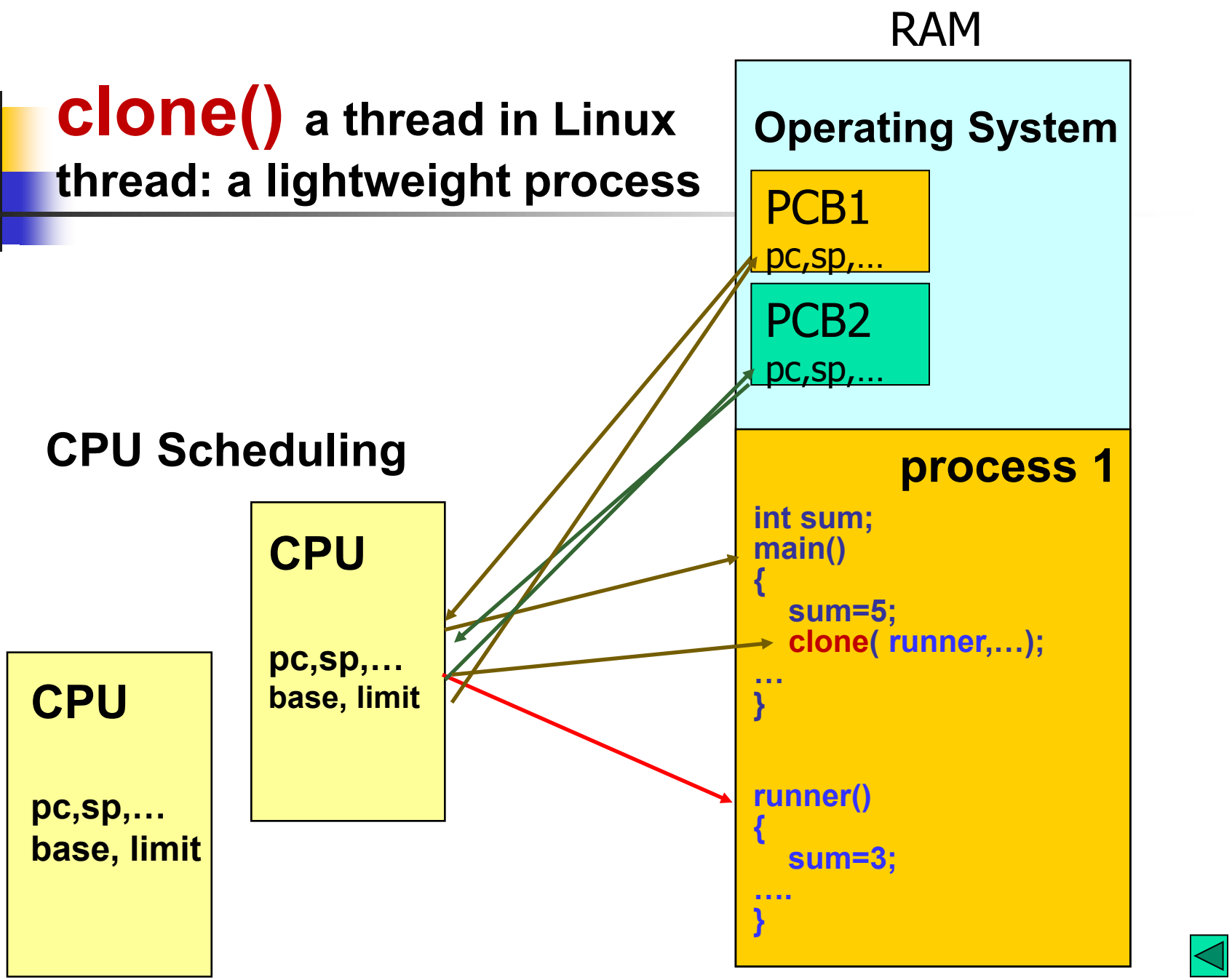
execve()

execute a file



clone() a thread in Linux

thread: a lightweight process

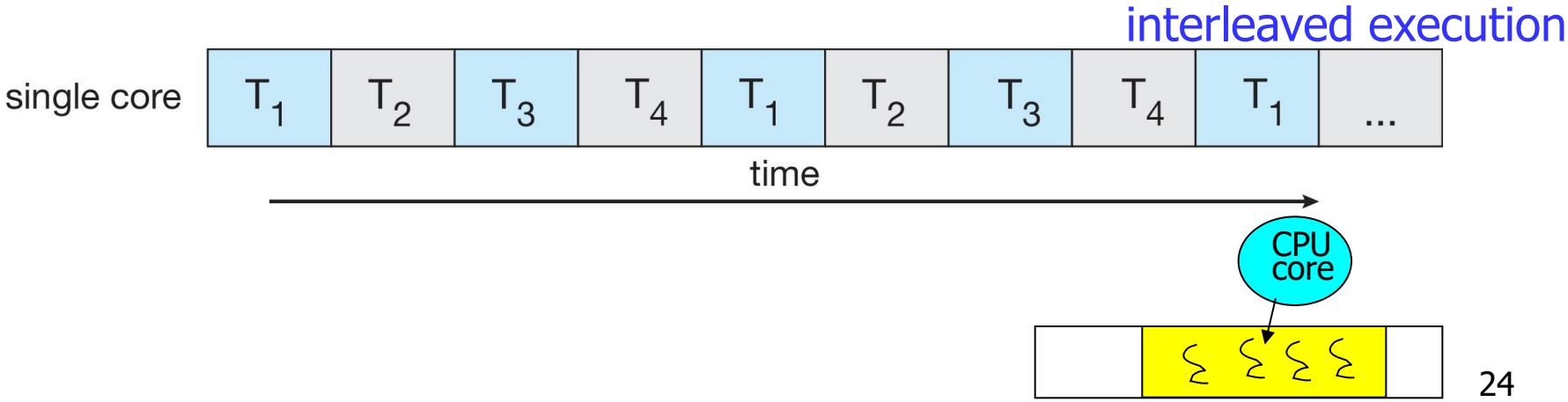


4.2 Multicore Programming

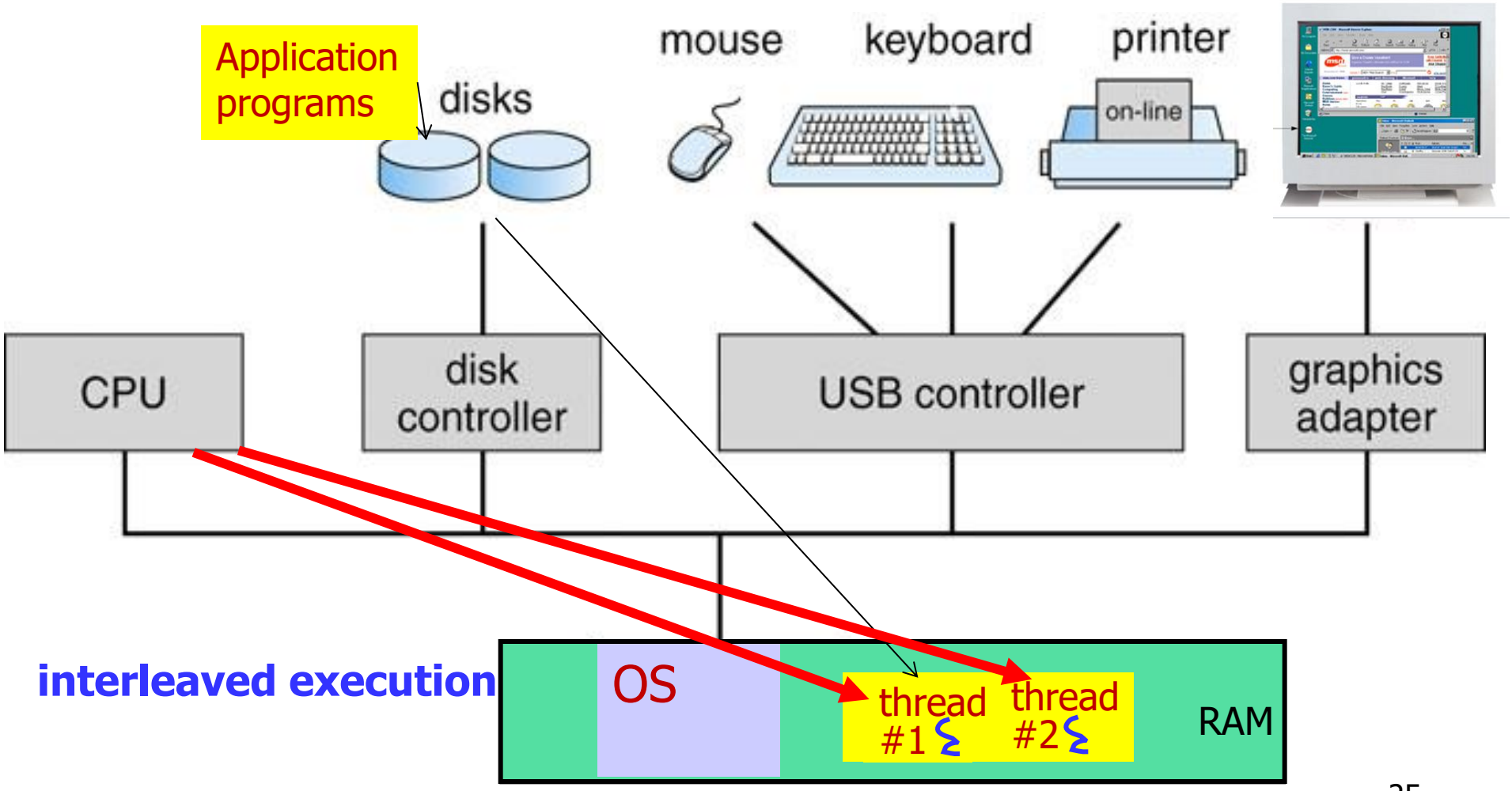
- **Multithreaded programming provides a mechanism for more efficient use of multiple computing cores and improved concurrency.**
- **On a system with a single computing core, concurrency merely means that the execution of the threads will be interleaved over time.**
- **Concurrent execution on single-core system:**

同時發生

交錯



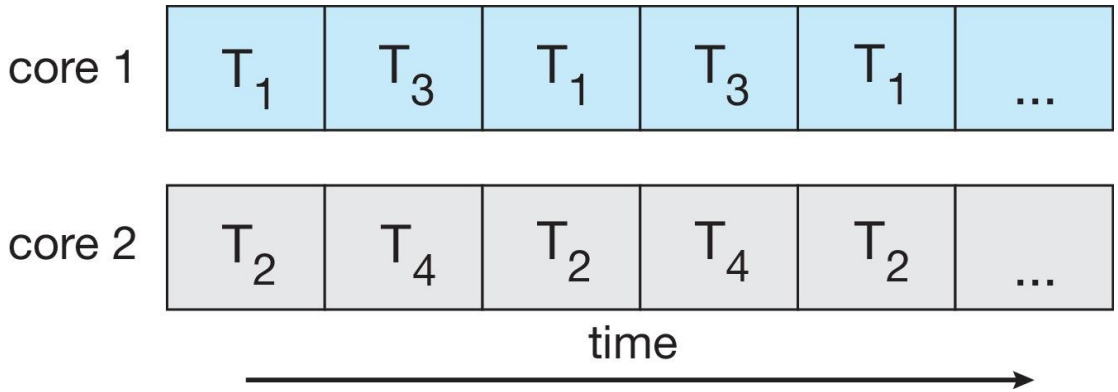
Concurrent Execution
on a Single-Core Processor

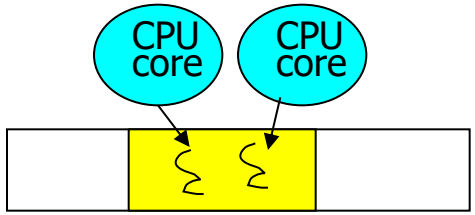


Multicore Programming (Cont.)

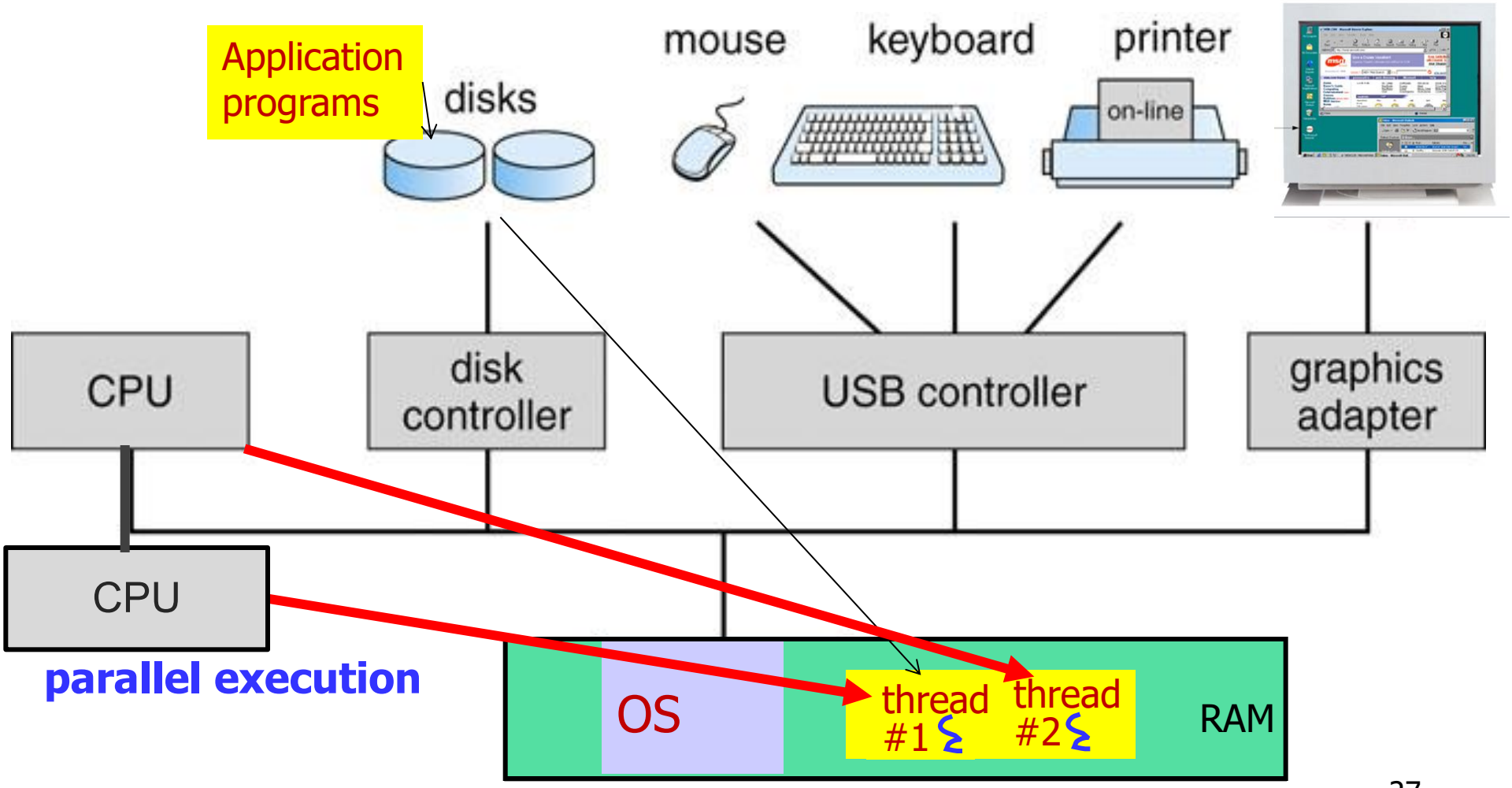
同時發生

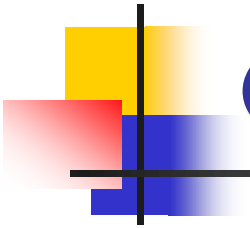
- On a system with multiple cores, **concurrency** means that the threads can run in parallel, as the system can assign a separate thread to each core.
- **Parallel** execution on a multi-core system:





Parallel Execution

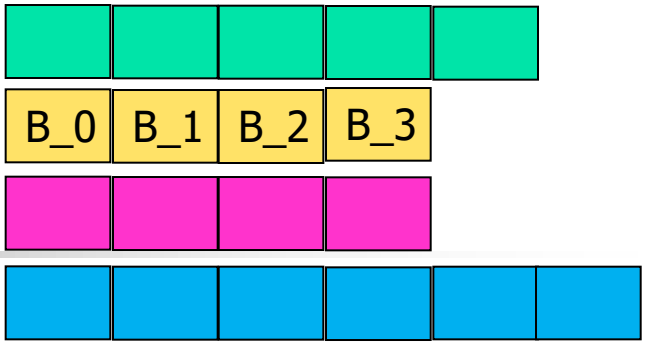


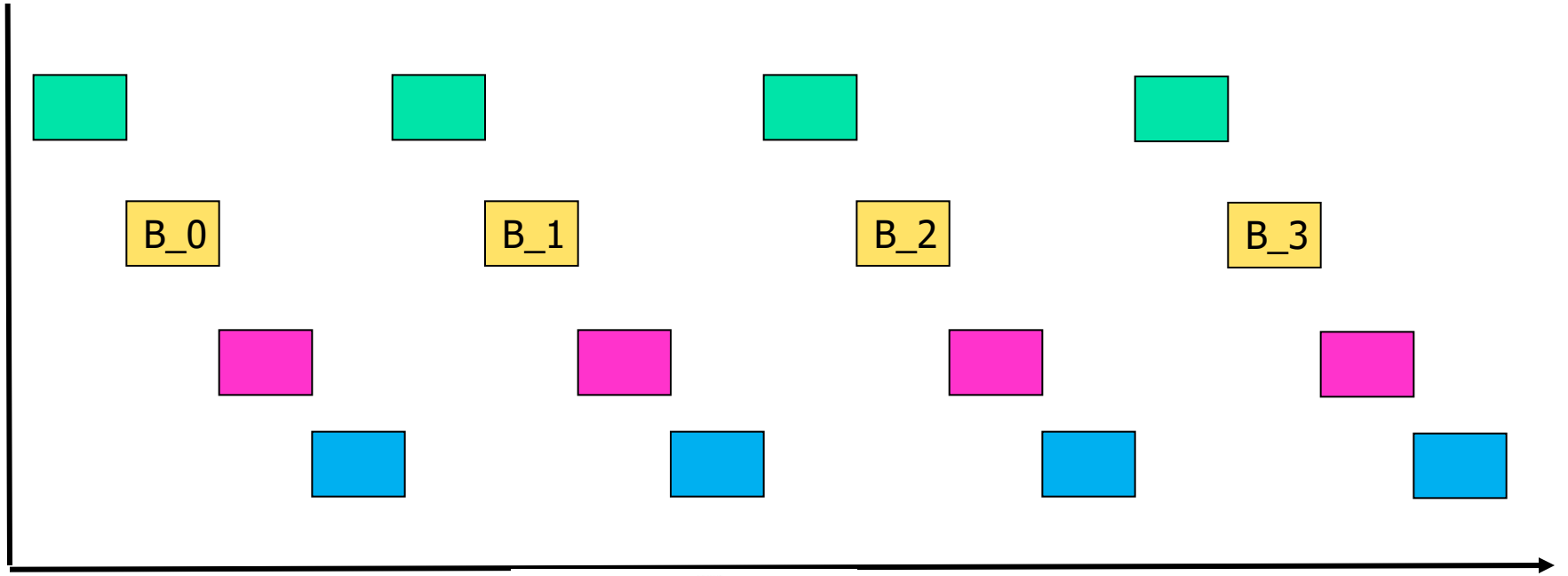


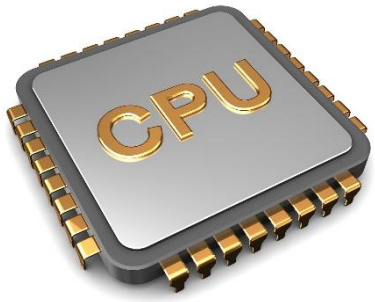
Concurrency vs. Parallelism

- **A system is *parall* if it can perform more than one task simultaneously**
- **A *concurrent* system supports more than one task by allowing all the tasks to make progress**
  - Single processor / core, CPU scheduler providing concurrency
    - Provide the illusion of parallelism by rapidly switching between processes, thereby allowing each process to make progress
    - Processes were running concurrently, but not in parallel

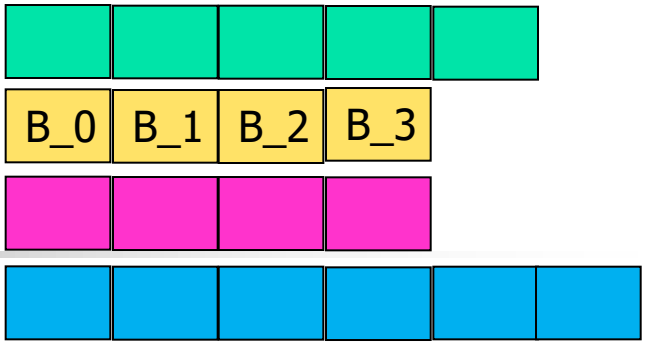
Concurrency

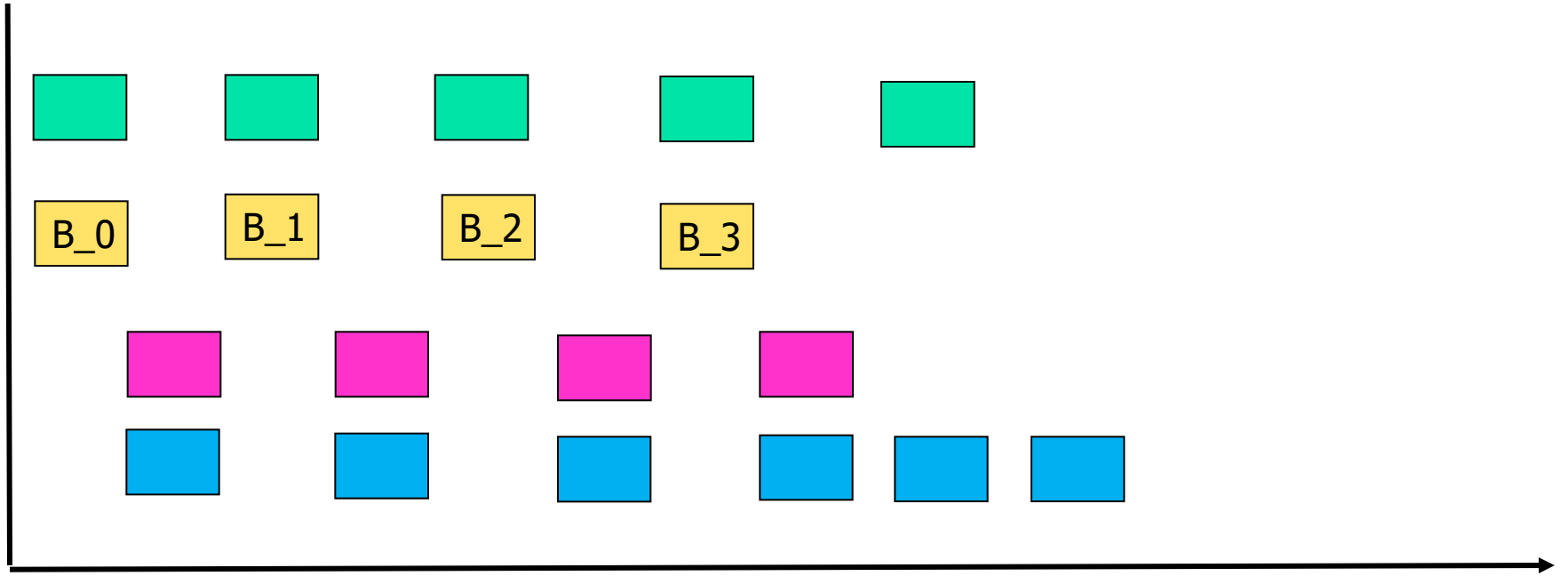


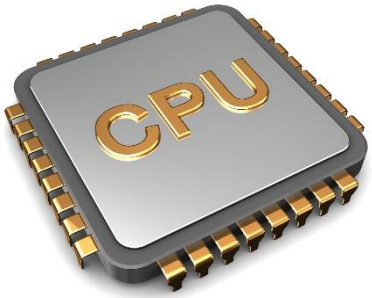


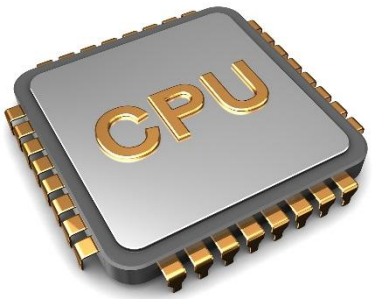


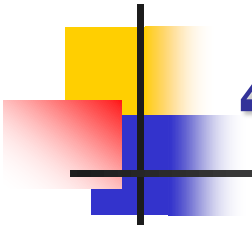
Concurrency







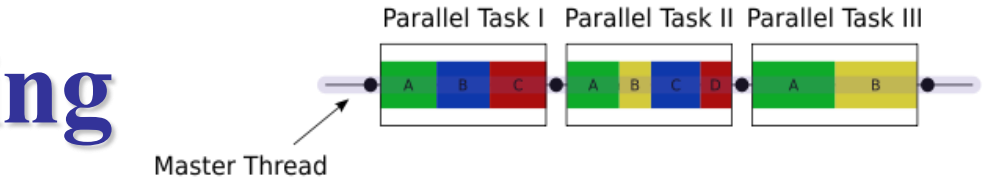


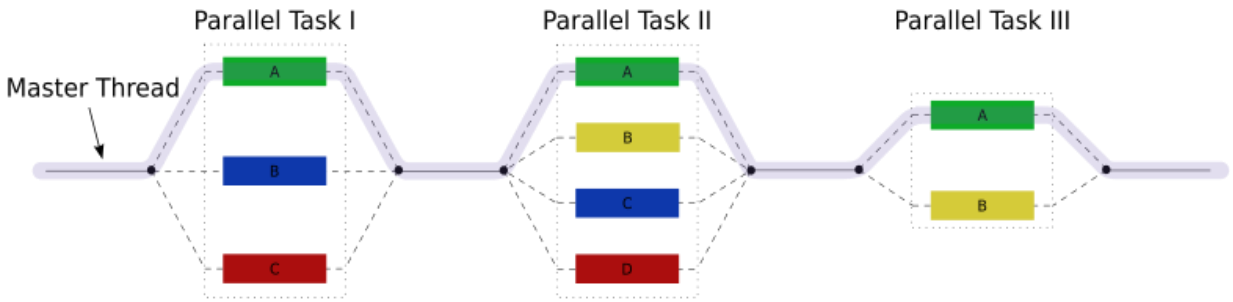


4.2.1 Programming Challenges

- **Multicore systems put pressure on system designers and application programmers to make better use of multiple computing cores.**
  - **OS designers**: write **scheduling algorithms** to allow the parallel execution on multiple processing cores
  - **Application programmers:** modify existing programs and design multithreaded programs

Programming Challenges (Cont.)

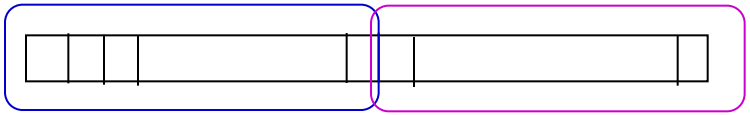




- **Challenges in programming for Multicore systems:**
  - **Dividing activities**
    - Identifying tasks that can run in parallel
  - **Balance**
    - Ensure that the tasks perform equal work
  - **Data splitting**
    - Data accessed by the tasks must be divided to run on separate cores
  - **Data dependency**
    - Ensure that the tasks' execution is synchronized to accommodate the data dependency
  - **Testing and debugging**

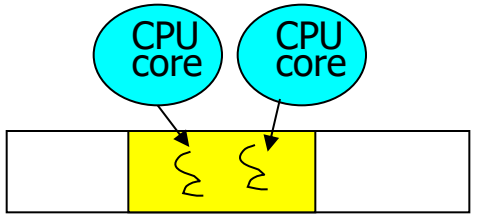
4.2.2 Types of Parallelism

- **Data parallelism – distributes subsets of the same data across multiple computing cores, perform the same operation on each core**
  - e.g., two threads sum array[N] on a dual-core system in parallel

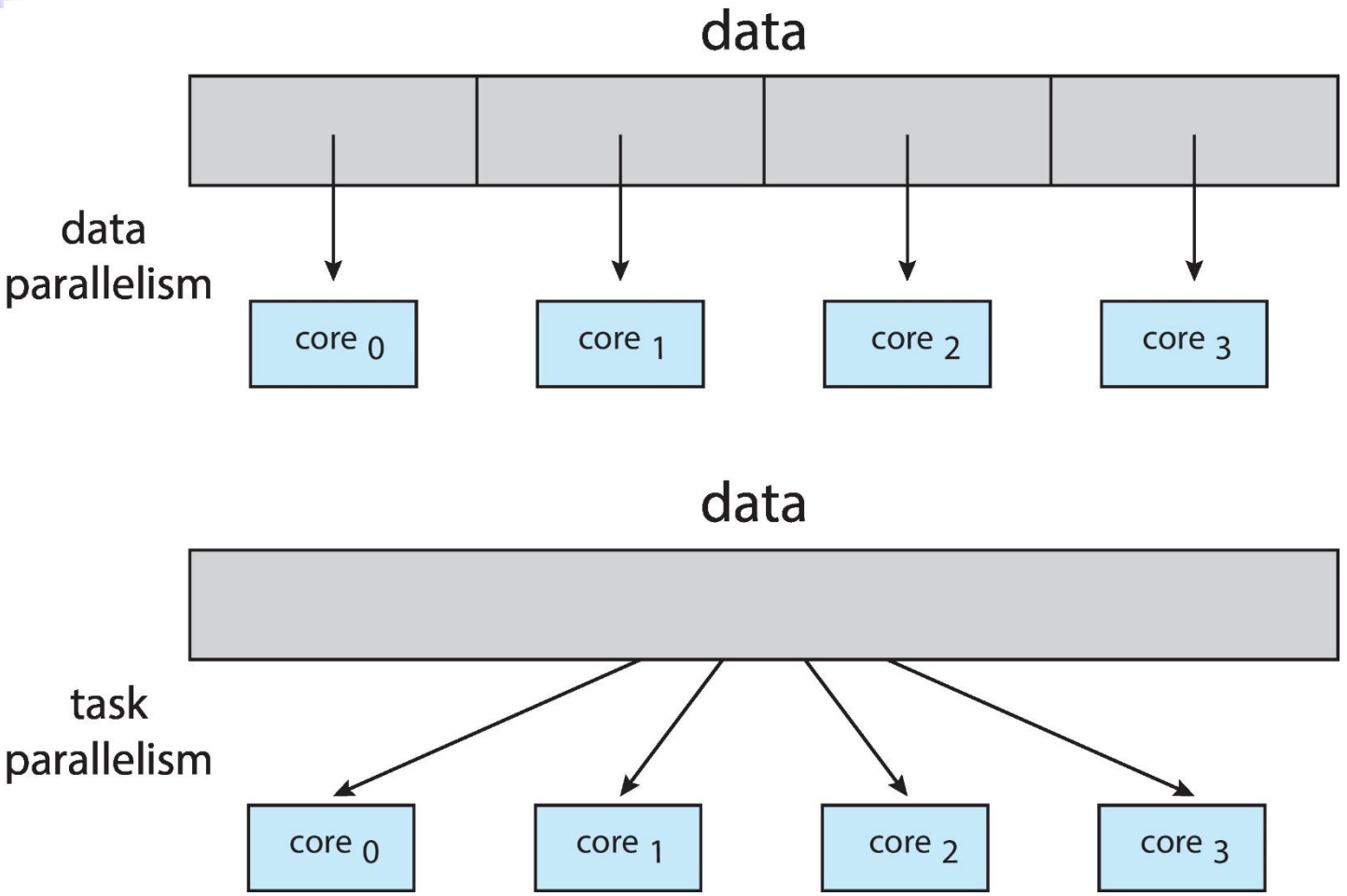


int array[N];

    - thread A sums array[0]..[N/2-1] on core 0
    - thread B sums array[N/2]..[N-1] on core 1
- **Task parallelism – distributing tasks (threads) across multiple computing cores, each thread is performing a unique operation**
  - e.g., thread A sums, thread B multiplies
  - e.g., producer thread, consumer thread



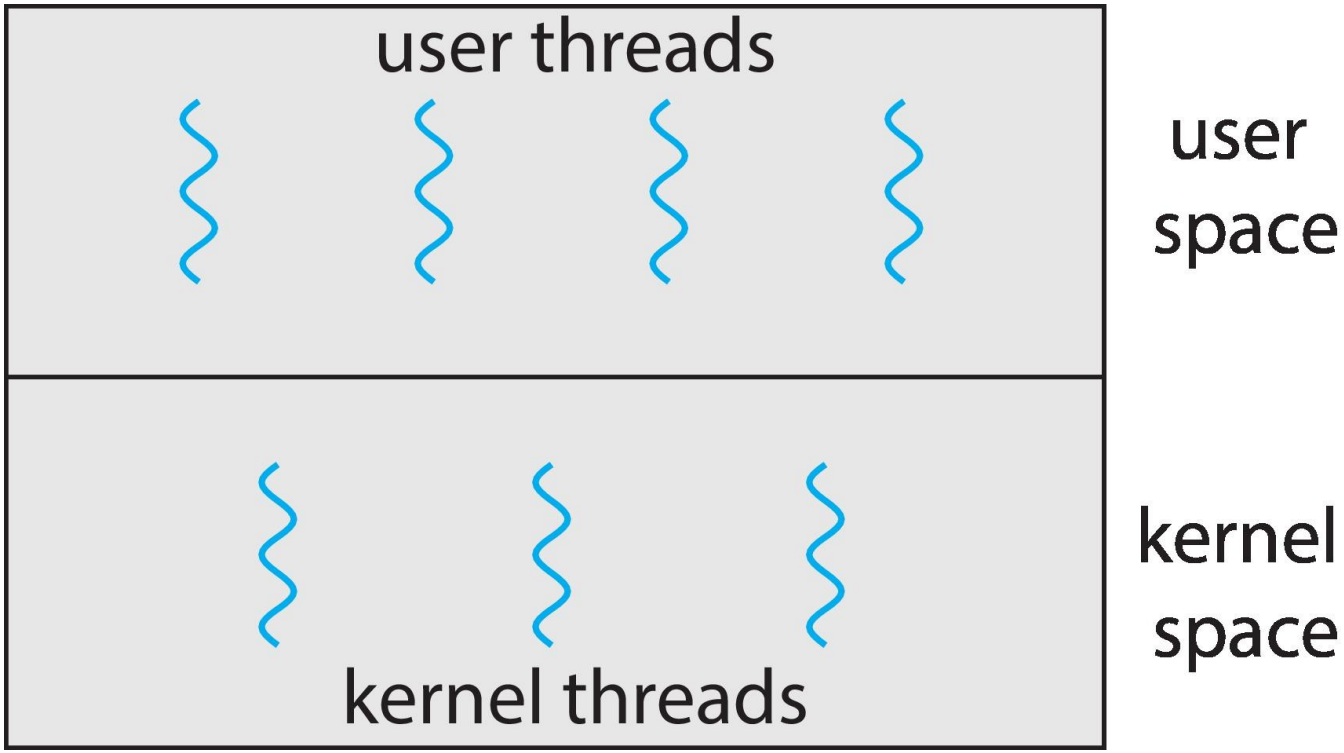
Data and task parallelism (Fig. 4.5)



4.3 Multithreading Models

- **Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.**
- **User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the OS.**
- **Virtually all contemporary OSes support kernel threads:**
  - Windows, Linux, macOS

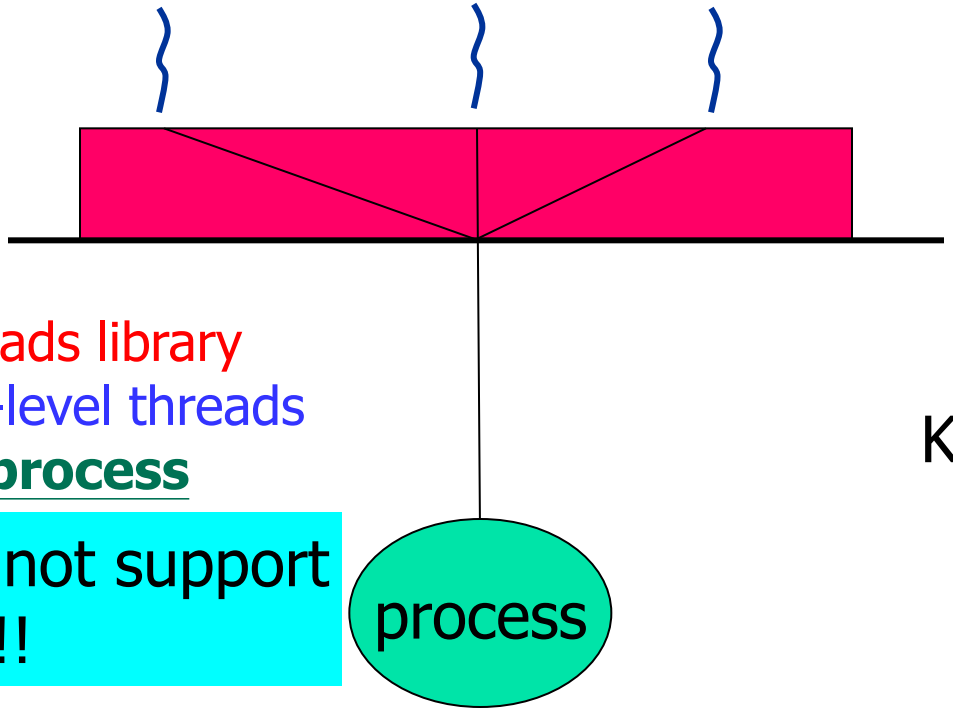
User and kernel threads (Fig. 4.6)



User-level Threads (1)

- **User-level threads are generally faster to create and manage than are kernel threads**
  - Because no intervention from the kernel is required

介入

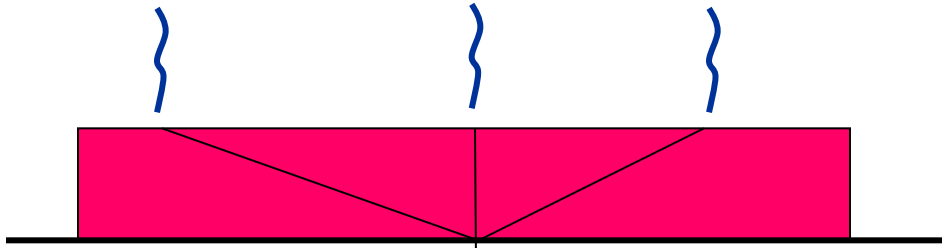


User-level threads library multiplex user-level threads above **an OS process**

If kernel does not support kernel thread!!!

User-level Threads (2)

- **User-level threads are generally faster to create and manage than are kernel threads**
  - Because no intervention from the kernel is required



Thread library

User space

User-level threads library multiplex user-level threads above an OS process

a kernel thread

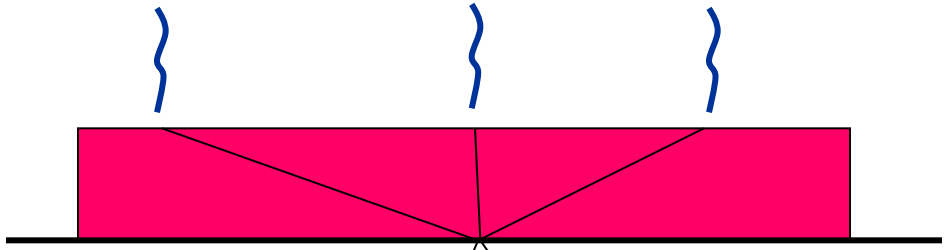
Kernel space

If kernel supports kernel thread!!!

kernel thread

User-level Threads (3)

- **User-level threads are generally faster to create and manage than are kernel threads**
  - Because no intervention from the kernel is required



Thread library

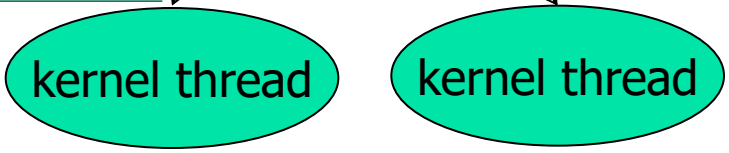
User space

User-level threads library multiplex user-level threads above an OS process

a kernel thread

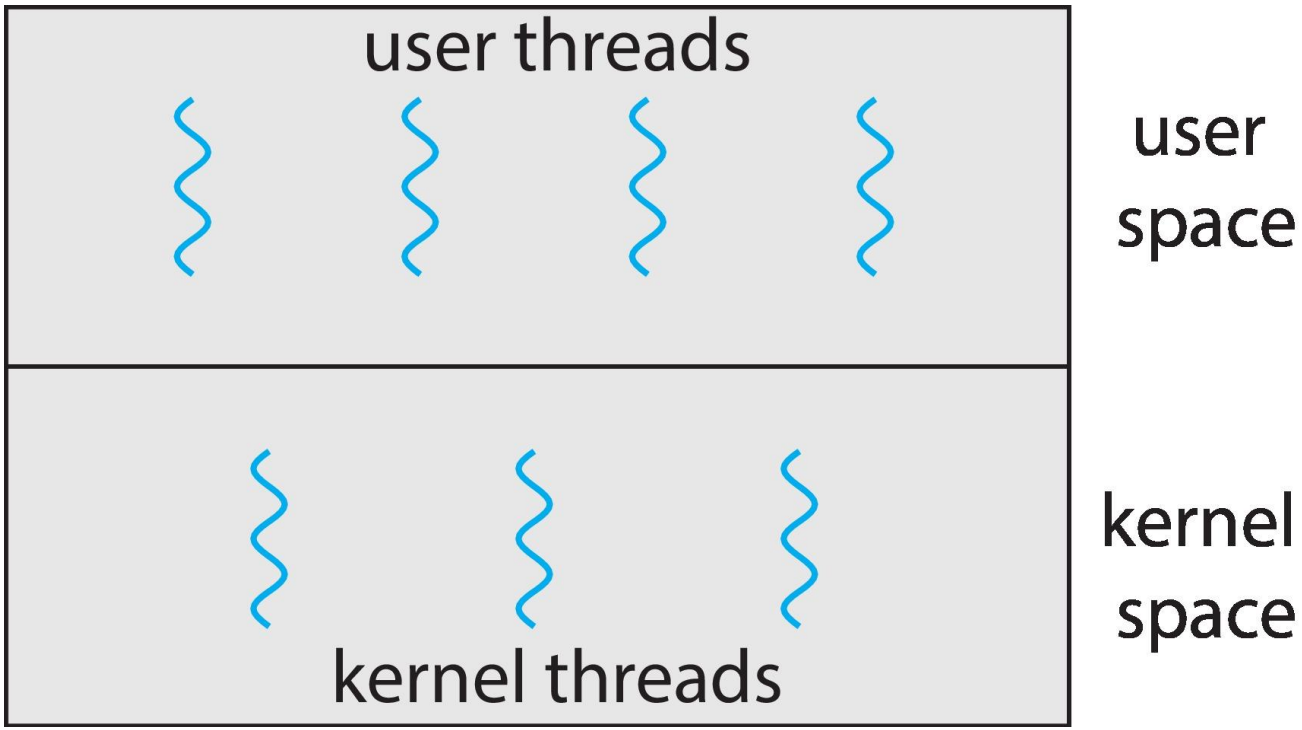
a set of kernel thread

Kernel space

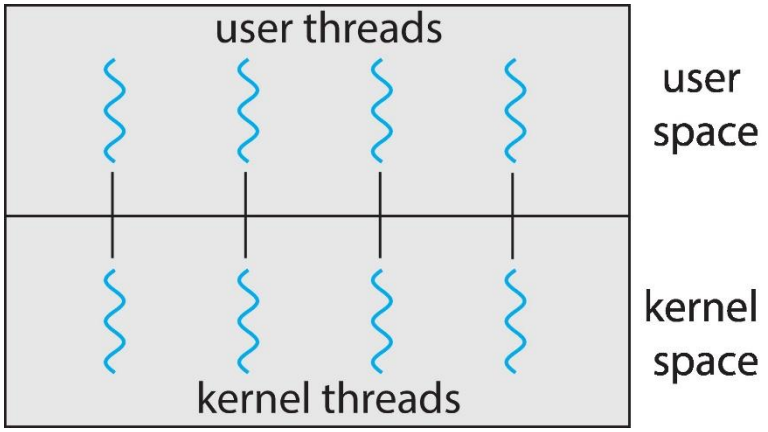


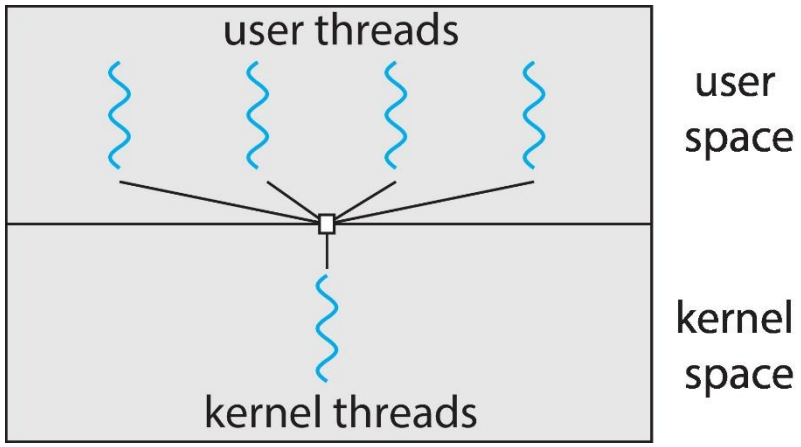
If kernel supports kernel thread!!!

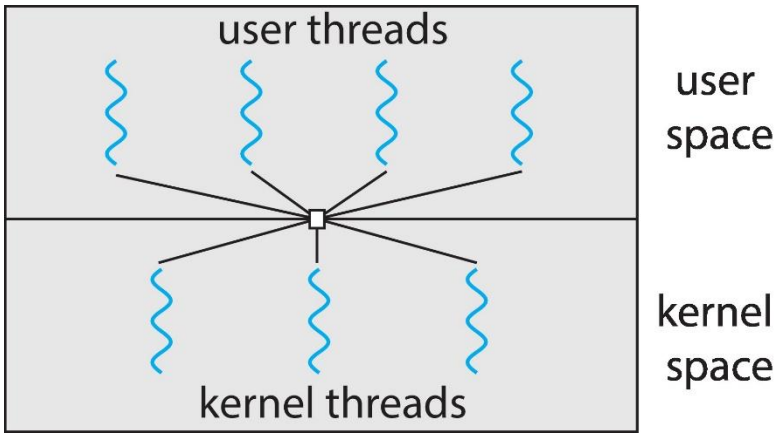
Relationship between user threads and kernel threads

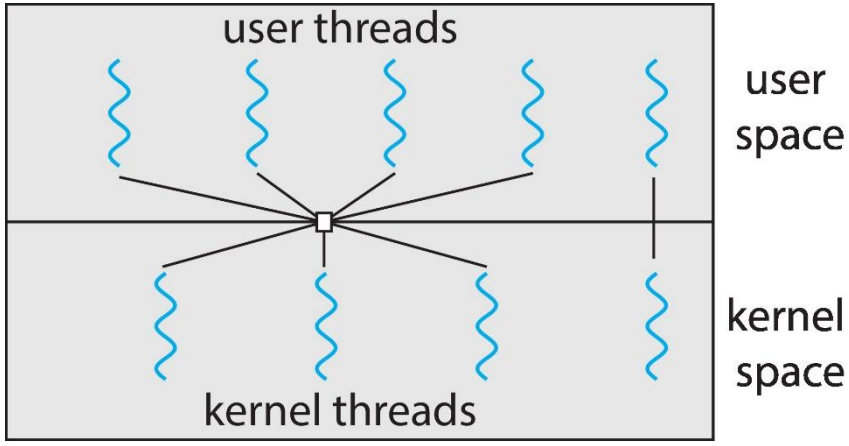


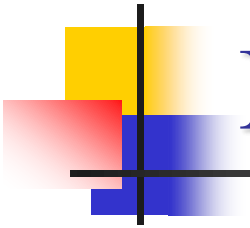
Relationship between user threads and kernel threads (Cont.)











Multithreading Models (Cont.)

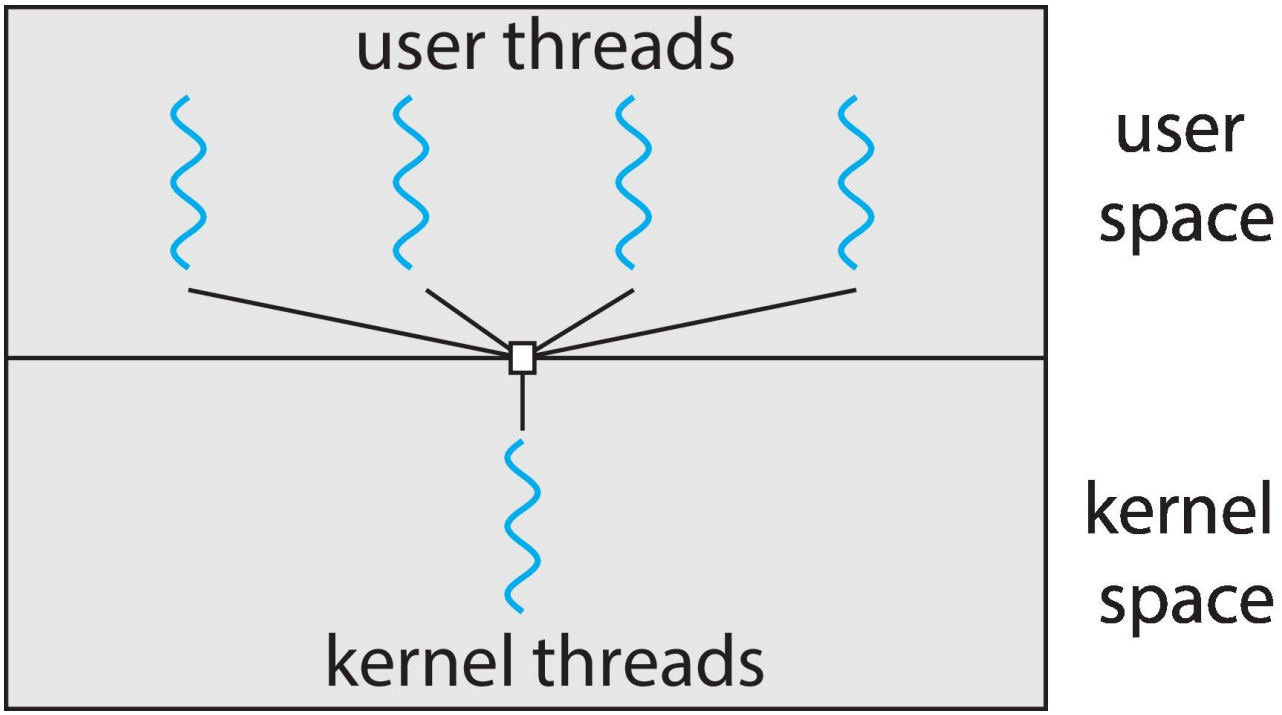
- Can threads run in parallel on multicore systems?
- If a thread makes a blocking system call, is the entire process still runnable?

Multithreading Models (Cont.)

- **Three common ways of establishing relationship between user threads and kernel threads:**
  - 4.3.1 Many-to-One Model 
  - 4.3.2 One-to-One Model 
  - 4.3.3 Many-to-Many Model 

4.3.1 Many-to-One Model

- **Maps many user-level threads to one kernel thread.**

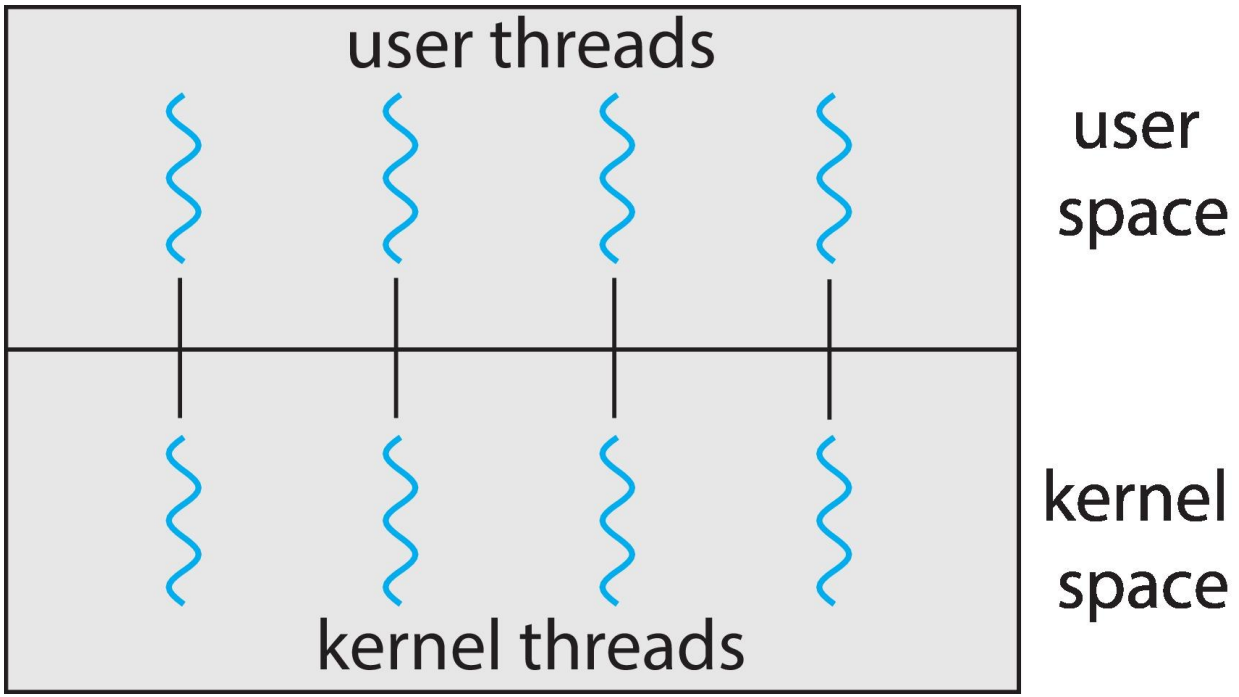


Many-to-One Model (Cont.)

- **Thread management is done by the thread library in user space, so it is efficient**
- **The entire process will block if a thread makes a blocking system call.**
  - because only one thread can access the kernel at a time
  - multiple threads are unable to run in parallel on multicore systems
- **Cannot take advantage of multiple processing cores**
    - very few systems currently use this model
- **Example:**
  - Green threads – a thread library for Solaris systems

4.3.2 One-to-One Model

- Maps each user thread to a kernel thread.



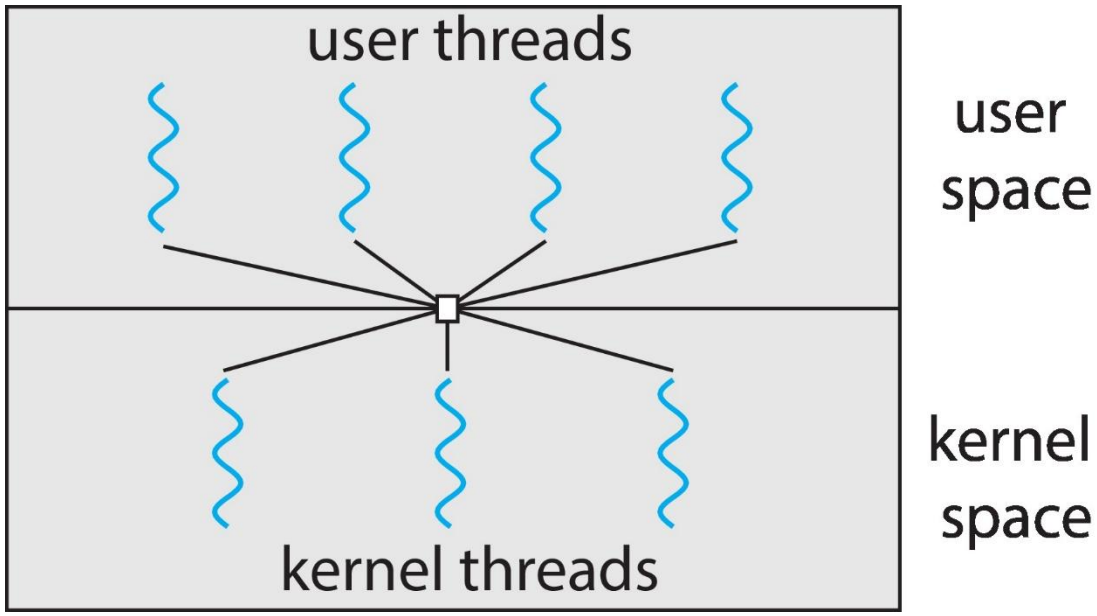
One-to-One Model (Cont.)

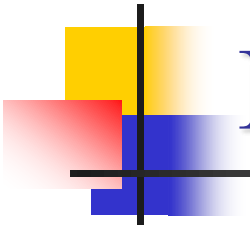
- **Provides more concurrency than many-to-one model by allowing another thread to run when a thread makes a blocking system call.**
- **Allows multiple threads to run in parallel on multiprocessors.**
- **Creating a user thread requires creating the corresponding kernel thread, and a large number of kernel threads may burden the performance of a system.**
- **Examples**
  - Linux, the family of Windows OSes

4.3.3 Many-to-Many Model

- **Multiplexes many user-level threads to a smaller or equal number of kernel threads.**

|user threads| >= |kernel threads|





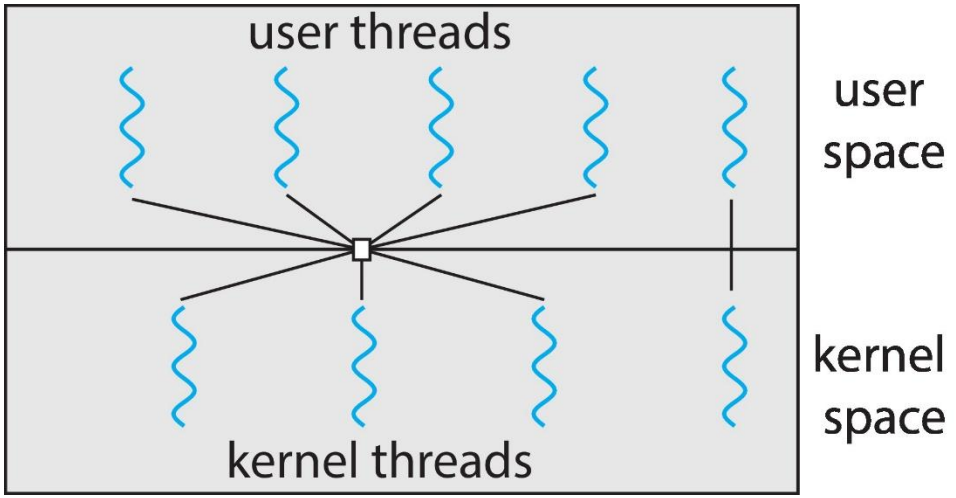
Discussion of Concurrency

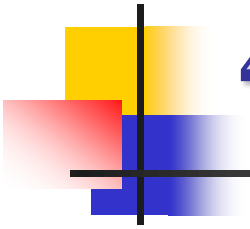
- **many-to-one model**
  - Allows the developer to create as many user threads as she wishes, it does not result in parallelism, because the kernel can schedule only one kernel thread at a time.
- **one-to-one model**
  - Allows greater concurrency, but the developer has to be careful not to create too many threads within an application.
  - When a thread performs a blocking system call, the kernel can schedule another thread for execution.
- **many-to-many model**
  - Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor.
  - When a thread performs a blocking system call, the kernel can schedule another thread for execution.

Two-level Model

變種

- **One variation on the many-to-many model**
  - still multiplexes many user-level thread to a smaller or equal number of kernel threads
  - also allows a user-level thread to be bound to a kernel thread
  - Appears to be the most flexible one, but is difficult to implement
- **With an increasing number of processing cores, limiting the number of kernel threads has become less important.**
  - Most OSes now use one-to-one model



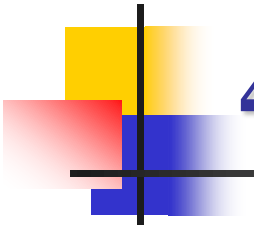


4.4 Thread Libraries

- **A thread library provides the programmer an API for creating and managing threads.**
- **Two primary ways of implementing a thread library:**
  - (1) Provide a library entirely in user space with no kernel support
    - All code and data structures for the library exist in user space.
      - Invoking a function in the library results in a local function call in user space and not a system call
  - (2) Implement a kernel-level library supported directly by OS
    - code and data structures for the library exist in kernel space.
      - Invoking a function in the API for the library results in a system call to the kernel

Thread Libraries (Cont.)

- **Three main thread libraries are in use today:**
  - (1) **POSIX Pthreads**
    - The threads extension of the POSIX standard, may be provided as either a user-level or kernel-level library.
  - (2) **Windows**
    - A kernel-level library available on Windows systems
  - (3) **Java**
    - The **Java thread API** allows thread creation and management directly in Java programs.
    - The Java thread API is typically implemented using a thread library available on the host system.
      - Windows API on Windows systems
      - Pthreads API on UNIX, Linux, macOS



4.4.1 Pthreads

- **Pthreads refers to the POSIX standard (IEEE 1003.1c) defining an API for thread creation and synchronization.**
- ***Specification*, not *implementation***
- **API specifies behavior of the thread library, implementation is up to development of the library.**
- **Common in UNIX-type systems:**
  - Linux, macOS

Multithreaded C program using the Pthread API (Fig. 4.11)

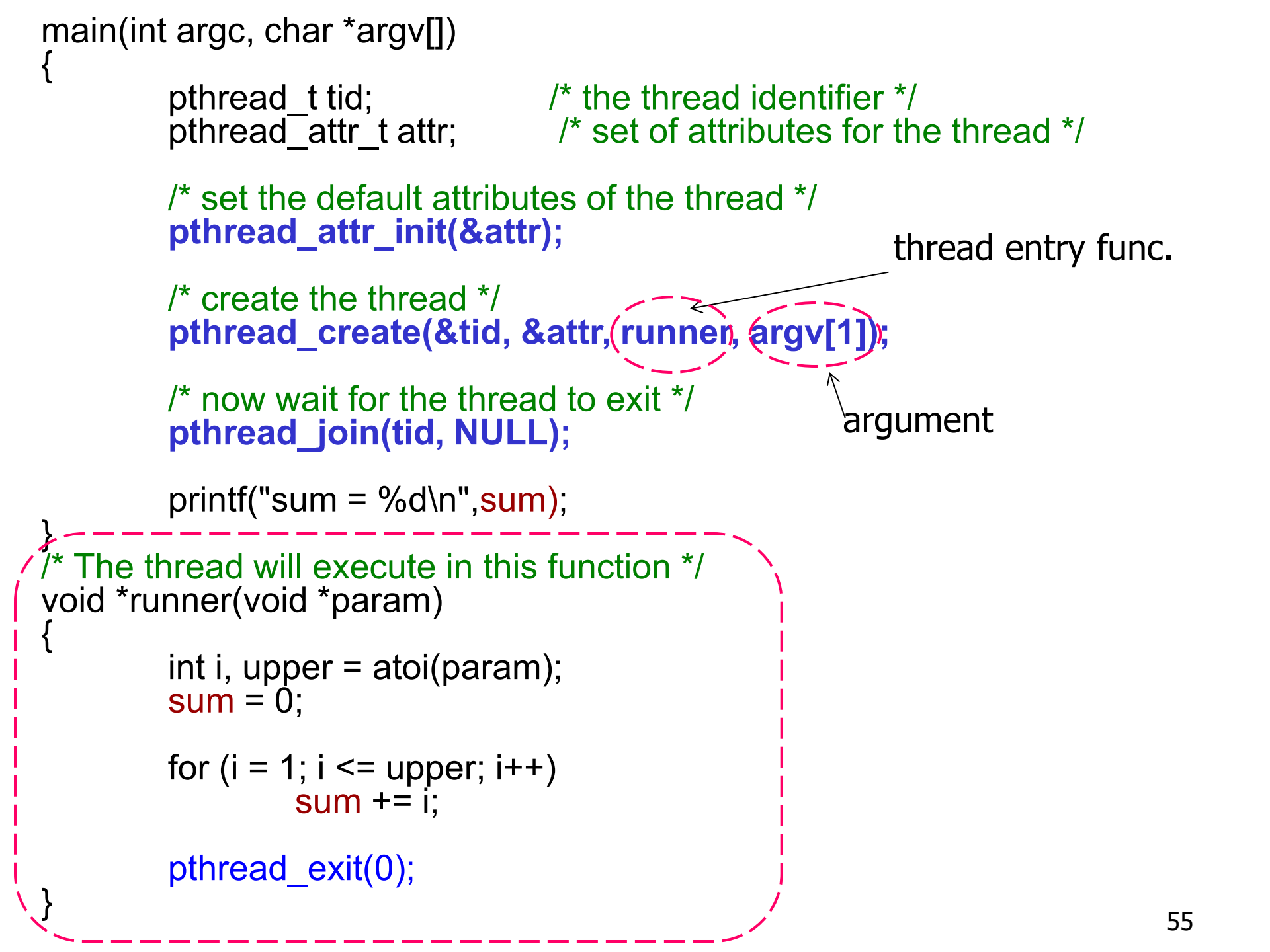
```
/**
 * A pthread program illustrating how to
 * create a simple thread and some of the pthread API
 * This program implements the summation function where
 * the summation operation is run as a separate thread.
 * Usage on Solaris/Linux/Mac OS X:
 *
 * gcc thrd.c -lpthread
 * a.out <number>
 */
```

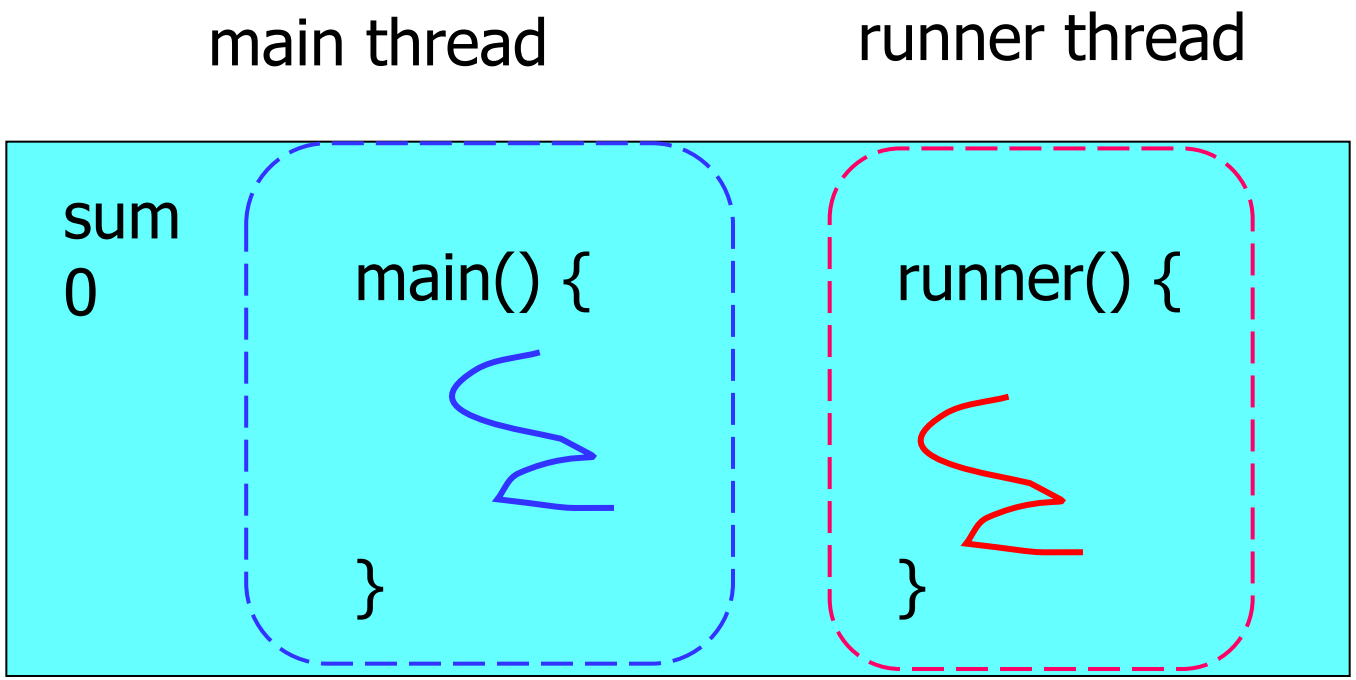
← compile & link with pthread lib.

$$sum = \sum_{i=1}^{N} i$$

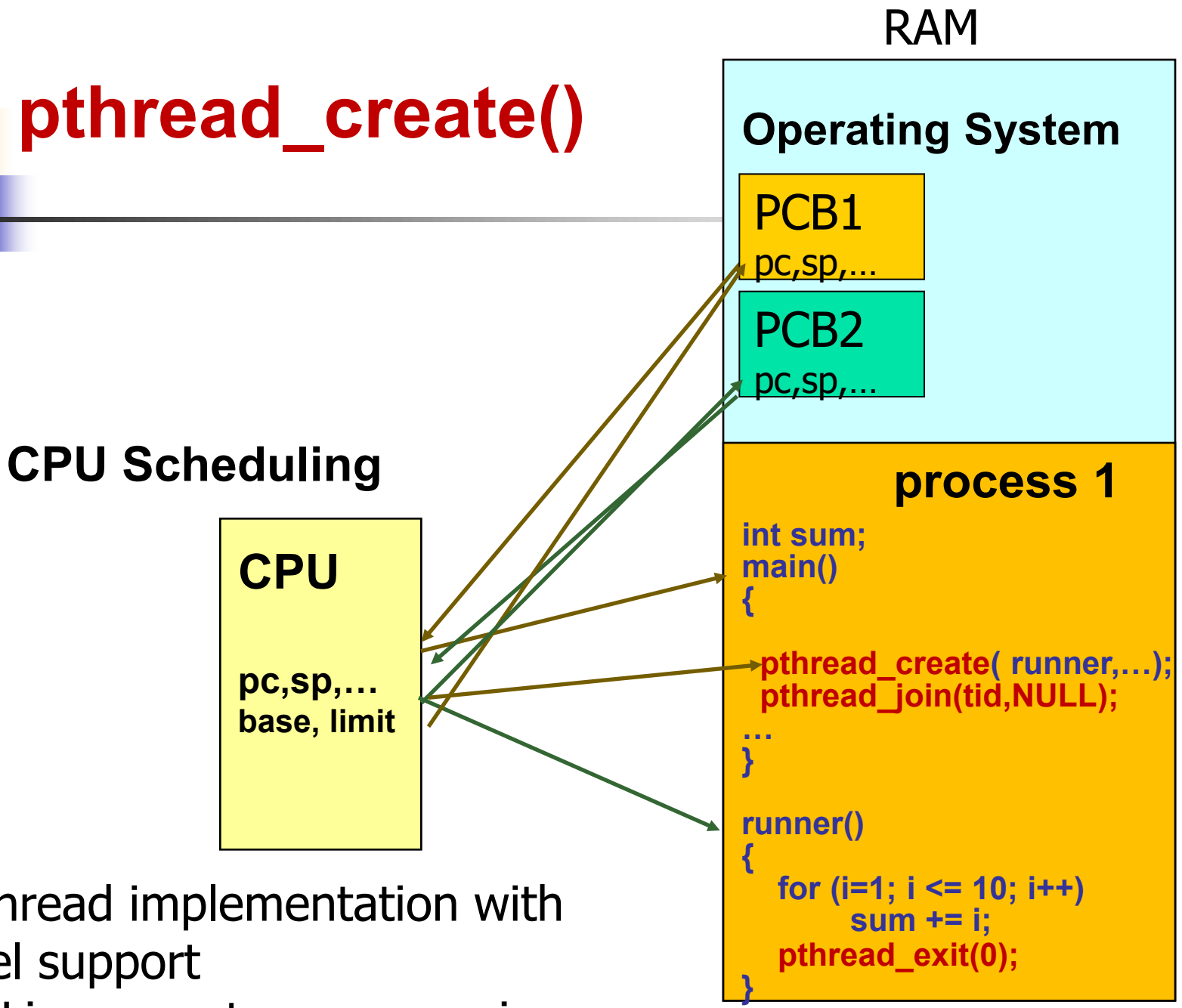
```
#include <pthread.h>
#include <stdio.h>

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

void *runner(void *param);          /* the thread */
```

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

        /* set the default attributes of the thread */
        pthread_attr_init(&attr);

        /* create the thread */
        pthread_create(&tid, &attr, runner, argv[1]);

        /* now wait for the thread to exit */
        pthread_join(tid, NULL);

        printf("sum = %d\n",sum);
}
/* The thread will execute 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);
}
```




main thread
runner thread
sum
0
main() {
}
runner() {
}

pthread_create()

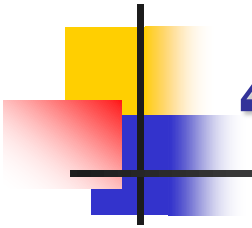


* if thread implementation with kernel support
e.g., Linux one-to-one mapping

Pthreads Code for Joining 10 Threads (Fig. 4.12)

```
#define NUM_THREADS 10

/* an array of threads to be joined upon */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
   pthread_join(workers[i], NULL);
```

Figure 4.12 Pthread code for joining ten threads.



4.4.2 Windows Threads

- The technique for creating threads using the Windows thread library is similar to the Pthreads technique.
- Figure 4.13: Multithreaded C program using the Windows API.

Multithreaded C program using the Windows API (Fig. 4.13)

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

$$sum = \sum_{i=0}^{N} i$$

Multithreaded C program using Win32 API (Fig. 4.13) (Cont.)

```
int main(int argc, char *argv[])
{
        DWORD ThreadId;
        HANDLE ThreadHandle;
        int Param;

        Param = atoi(argv[1]);
        // create the thread
        ThreadHandle = CreateThread(NULL, 0, Summation,
                                    &Param, 0, &ThreadId);

        // now wait for the thread to finish
        WaitForSingleObject(ThreadHandle, INFINITE);

        // close the thread handle
        CloseHandle(ThreadHandle);

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


4.4.3 Java Threads

- **Java provides support at the language level for the creation and management of threads.**
- **Java threads are managed by Java Virtual Machine.**
- **Typically implemented using the threads model provided by underlying OS**
- **Java threads may be created by:**
  - (1) Extending Thread class
  - (2) Defining a class that implements the Runnable interface

```
public interface Runnable
{
    public abstract void run();
}
```

Standard practice is to implement Runnable interface

(1) Java program for the summation of a non-negative integer

```
public class Summation extends Thread
{
        public Summation(int n) {
                upper = n;
        }

        public void run() {
                int sum = 0;

                if (upper > 0) {
                        for (int i = 1; i <= upper; i++)
                                sum += i;
                }

                System.out.println("The summation of " + upper + " is " + sum);
        }

        private int upper;
}
```

$$sum = \sum_{i=0}^{N} i$$

(1) Java program for the summation of a non-negative integer (Cont.)

```
public class ThreadTester
{
    public static void main(String[] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) < 0)
                throw new IllegalArgumentException(args[0] + "
    must be non-negative.");
            else {
                Summation summationThread = new
                    Summation(Integer.parseInt(args[0]));
                summationThread.start();
            }
        }
        else
            System.err.println("Usage: Summation <integer value>");
    }
}
```


(2) Java program for the summation of a non-negative integer (Fig. 4.12)

```
/**
 * This program creates a separate thread by implementing the Runnable interface.
  * To compile, enter
 *  javac Driver.java
*/
class Sum
{
    private int sum;
    public int getSum() {
        return sum;
    }
    public void setSum(int sum) {
        this.sum = sum;
    }
}
```

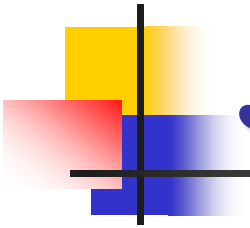
(2) Java program for the summation of a non-negative integer (Cont.)

```
class Summation implements Runnable
{
    private int upper;
    private Sum sumValue;

    public Summation(int upper, Sum sumValue) {
        if (upper < 0)
            throw new IllegalArgumentException();
        this.upper = upper;
        this.sumValue = sumValue;
    }

    public void run() {
        int sum = 0;

        for (int i = 0; i <= upper; i++)
            sum += i;
        sumValue.setSum(sum);
    }
}
```

$$sum = \sum_{i=0}^{N} i$$

(2) Java program for the summation of a non-negative integer(Cont.)

```
public class Driver
{
    public static void main(String[] args) {
        if (args.length != 1) {
            System.err.println("Usage Driver <integer>");
            System.exit(0);
        }

        // create the object to be shared
        Sum sumObject = new Sum();
        int upper = Integer.parseInt(args[0]);

        Thread thrd = new Thread(new Summation(upper, sumObject));
        thrd.start();
        try {
            thrd.join();
        } catch (InterruptedException ie) { }
        System.out.println("The sum of " + upper + " is " + sumObject.get());
    }
}
```



Java Threads

Implementing Runnable interface:

```
class Task implements Runnable
{
   public void run() {
      System.out.println("I am a thread.");
   }
}
```

Creating a thread:

```
Thread worker = new Thread(new Task());
worker.start();
```

Waiting on a thread:

```
try {
   worker.join();
}
catch (InterruptedException ie) { }
```

Java Executor Framework

- **Rather than explicitly creating threads, Java also allows thread creation around the Executor interface:**

```
public interface Executor
{
   void execute(Runnable command);
}
```

The Executor is used as follows:

```
Executor service = new Executor;
service.execute(new Task());
```

Java Executor Framework

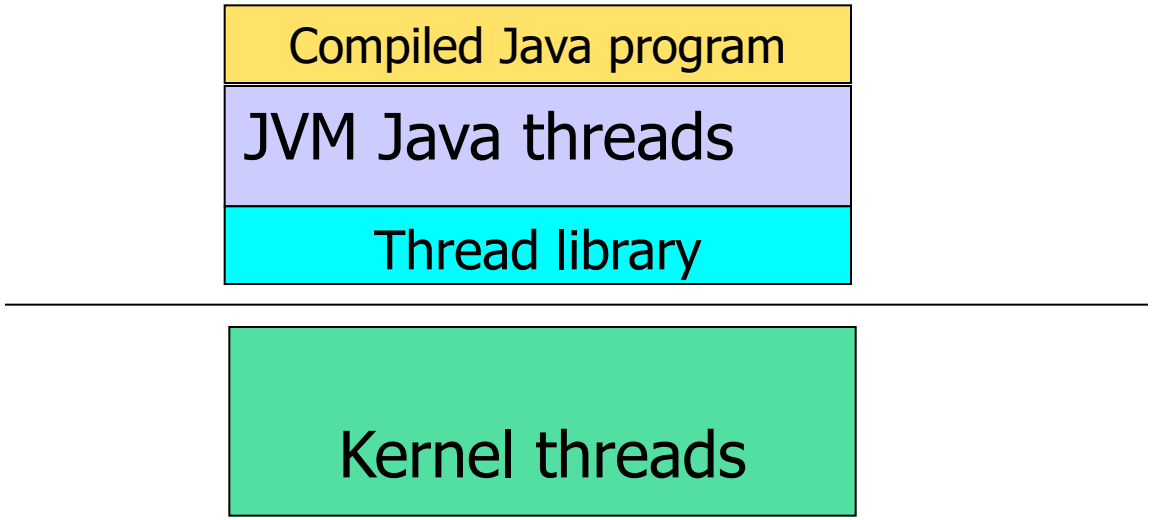
```
import java.util.concurrent.*;

class Summation implements Callable<Integer>
{
   private int upper;
   public Summation(int upper) {
      this.upper = upper;
   }

   /* The thread will execute in this method */
   public Integer call() {
      int sum = 0;
      for (int i = 1; i <= upper; i++)
         sum += i;

      return new Integer(sum);
   }
}
```

Java Executor Framework (cont.)

```
public class Driver
{
 public static void main(String[] args) {
    int upper = Integer.parseInt(args[0]);

    ExecutorService pool = Executors.newSingleThreadExecutor();
    Future<Integer> result = pool.submit(new Summation(upper));

    try {
        System.out.println("sum = " + result.get());
    } catch (InterruptedException | ExecutionException ie) { }
  }
}
```

Java Threads (Cont.)

- The specification for the JVM does not indicate how Java threads are to be mapped to the underlying OS, instead leaving that decision to the particular implementation of JVM.



内含的

4.5 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**
- **Five methods explored**
  - Thread Pools
  - Fork-Join
  - OpenMP
  - Grand Central Dispatch
  - Intel Threading Building Blocks

4.5.1 Thread pools

- ***a multithreaded server***
  - Whenever the server receives a request, it creates a separate thread to service the request.
  - Potential problems:
    - thread creation time, unlimited threads could exhaust system resources
- ***Thread pools***
  - Create a number of threads at process startup and place them into a pool, where they sit and wait for work.
  - When a server receives a request, it awakens a thread from this pool, passing it the request for service.
  - Once the thread completes its service, it returns to the pool awaiting more work.

Thread pools (Cont.)

- **Benefits of thread pools**
  - Faster to service a request with an existing thread than waiting to create a thread
  - Limits the number of threads that exist at any one point.
  - Separating task to be performed from mechanics of creating task allows different strategies for running task
    - i.e. tasks could be scheduled to run periodically
- **Examples:**
  - Windows thread pool API

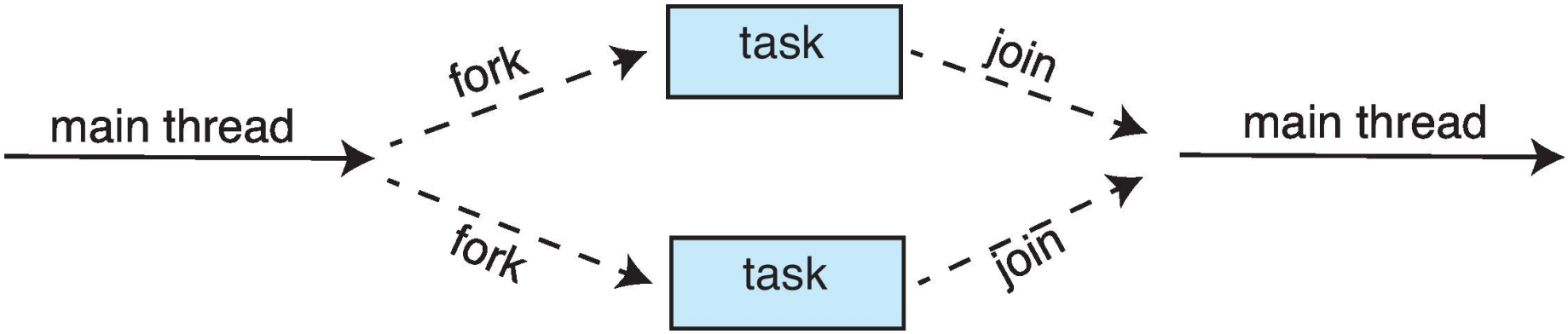
```
DWORD WINAPI PoolFunction(AVOID Param) {
    /*
    * this function runs as a separate thread.
    */
}

QueueUserWorkItem(&PoolFunction, NULL, 0);
```

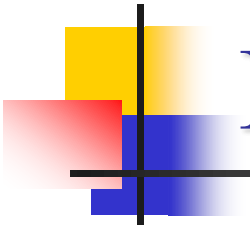
  - Java provides a thread pool utility

4.5.2 Fork Join

Fork-Join Parallelism

- **Multiple threads (tasks) are forked, and then joined.**



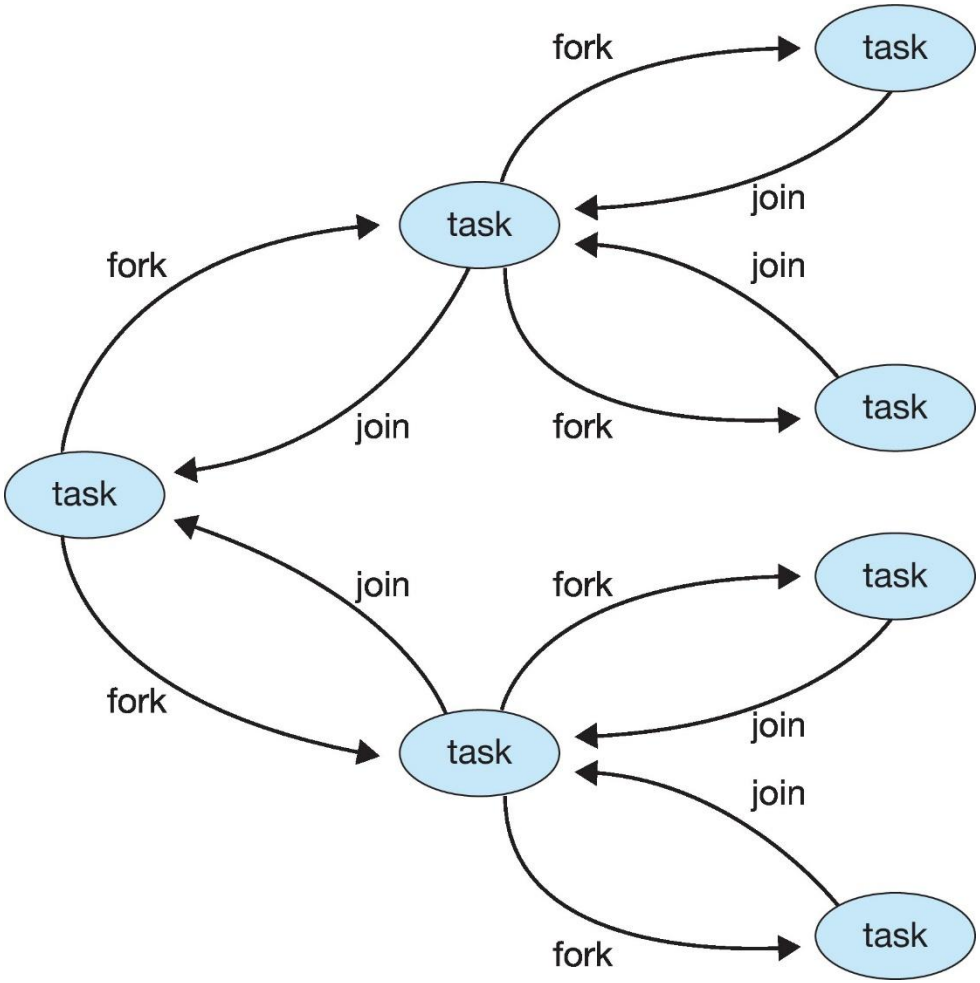


Fork-Join Parallelism

- **General algorithm for fork-join strategy:**

```
Task(problem)
   if problem is small enough
      solve the problem directly
   else
      subtask1 = fork(new Task(subset of problem)
      subtask2 = fork(new Task(subset of problem)

      result1 = join(subtask1)
      result2 = join(subtask2)

      return combined results
```

Fork-Join Parallelism



Fork-Join Parallelism in Java

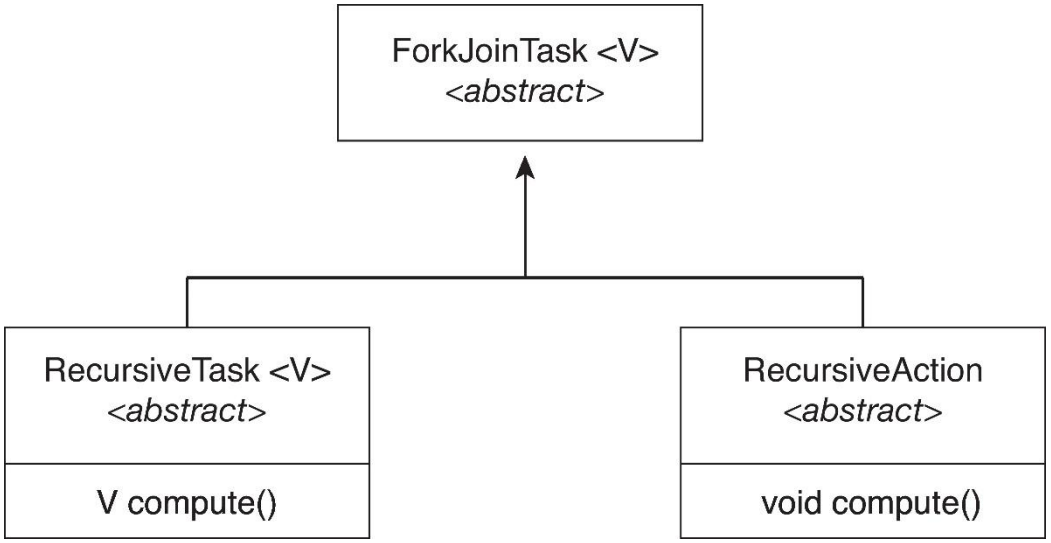
```
ForkJoinPool pool = new ForkJoinPool();
// array contains the integers to be summed
int[] array = new int[SIZE];

SumTask task = new SumTask(0, SIZE - 1, array);
int sum = pool.invoke(task);
```

Fork-Join Parallelism in Java

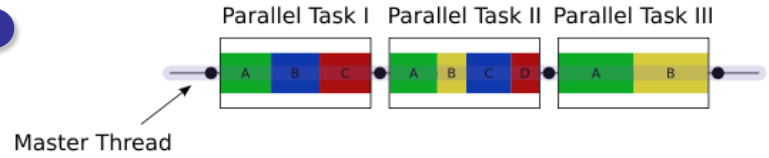
```
import java.util.concurrent.*;

public class SumTask extends RecursiveTask<Integer>
{
   static final int THRESHOLD = 1000;

   private int begin;
   private int end;
   private int[] array;

   public SumTask(int begin, int end, int[] array) {
      this.begin = begin;
      this.end = end;
      this.array = array;
   }

   protected Integer compute() {
      if (end - begin < THRESHOLD) {
         int sum = 0;
         for (int i = begin; i <= end; i++)
            sum += array[i];

         return sum;
      }
      else {
         int mid = (begin + end) / 2;

         SumTask leftTask = new SumTask(begin, mid, array);
         SumTask rightTask = new SumTask(mid + 1, end, array);

         leftTask.fork();
         rightTask.fork();

         return rightTask.join() + leftTask.join();
      }
   }
}
```

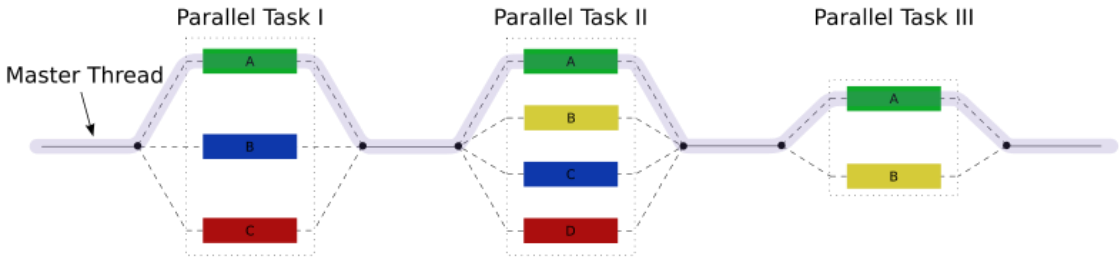
Fork-Join Parallelism in Java

- The ForkJoinTask is an abstract base class
- RecursiveTask and RecursiveAction classes extend ForkJoinTask
- RecursiveTask returns a result (via the return value from the compute() method)
- RecursiveAction does not return a result



4.5.3 OpenMP





- A 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 may run in parallel
- OpenMP is available on several open-source and commercial compilers for Linux, Windows, and Mac OS X systems.

OpenMP (Cont.)

#pragma omp parallel

Create as many threads as there are cores

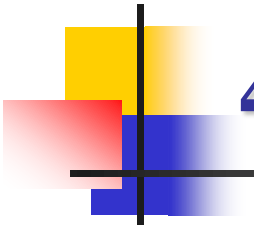
#pragma omp parallel for

```
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;
}
```

4.5.4 Grand Central Dispatch



- **A technology for Apple Mac OS X and iOS**
- **Extensions to C, C++ languages, API, and run-time library**
- **Allows application developers to identify 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 (Cont.)

- **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."); });
```

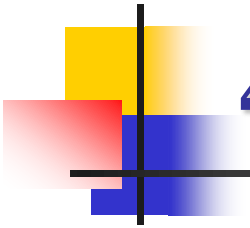
4.5.5 Intel Threading Building Blocks (TBB)

- Template library for designing parallel C++ programs
- A serial version of a simple for loop

```
for (int i = 0; i < n; i++) {
   apply(v[i]);
}
```

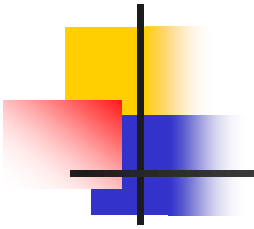
- The same for loop written using TBB with `parallel_for` statement:

```
parallel_for (size_t(0), n, [=](size_t i) {apply(v[i]);});
```



4.6 Threading Issues

- 4.6.1 fork() and exec() System Calls
- 4.6.2 Signal Handling
- 4.6.3 Thread Cancellation
- 4.6.4 Thread-Local Storage
- 4.6.5 Scheduler Activations



4.6.1 fork() & exec() System Calls

- **fork()**
  - What if one thread in a program calls fork():
    - Some UNIX systems have chosen to have two versions of fork():
      - The new process duplicates all threads
      - The new process is single-threaded (duplicates only the thread that invoked the fork()
- **exec()**
  - What if a thread invokes the exec() system call?
    - The program specified in the parameter to exec() will replace the entire process – including all threads.

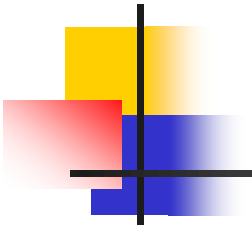
4.6.2 Signal Handling

- ***signal***
  - Used in UNIX systems to notify a process that a particular event has occurred.
- **All signals follow the same pattern:**
  - A signal is generated by the occurrence of a particular event.
  - A generated signal is delivered to a process.
  - Once delivered, the signal must be handled.

Signal Handling (Cont.)

exception

- ***Synchronous signal***
  - A running program performs some actions, e.g., illegal memory access or division by zero, a signal is generated.
  - Synchronous signals are delivered to the same process that performed the operation causing the signal
- ***Asynchronous signal***
  - A signal generated by an event external to a running process, that process receives the signal asynchronously.
  - e.g., terminate a process with CTRL-C or timer expire
  - Typically, an asynchronous signal is sent to another process.



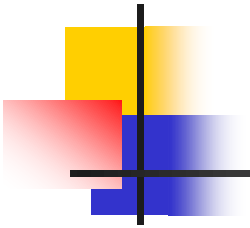
Signal Handling (Cont.)

- **Every signal may be handled by one of two possible handlers:**
  - A default signal handler
    - that kernel runs when handling signal
  - A user-defined signal handler
    - can override default

```c
#include <stdio.h>
#include <signal.h>

void sighup();
void sigint();
void sigquit();

int main(int argc, char *argv[]) {
    int pid;

    if ((pid = fork()) == 0) {
        printf("this is child process for receiving signal\n");
        signal(SIGHUP, sighup);
        signal(SIGINT, sigint);
        signal(SIGQUIT, sigquit);
        for(;;);
    } else {
        printf("this is parent process for sending signal\n");
        printf("\nPARENT: sending SIGHUP\n\n");
        kill(pid,SIGHUP);
        printf("\nPARENT: sending SIGINT\n\n");
        kill(pid,SIGINT);
        printf("\nPARENT: sending SIGQUIT\n\n");
        kill(pid,SIGQUIT);
        kill(pid,SIGHUP);
        kill(pid,SIGINT);
    }
}
void sighup() {
        printf("CHILD: I have received a SIGHUP\n");
}
void sigint() {
        printf("CHILD: I have received a SIGINT\n");
}
void sigquit() {
    printf("My DADDY has Killed me!!!\n");
}
```

user-defined signal handlers

sending signals to process



Signal Handling (Cont.)

- **How to deal with signals in a multithreaded environment?**
  - Deliver the signal to the thread to which the signal applies.
    - e.g., synchronous signals
  - Deliver the signal to every thread in the process.
    - e.g., CRTL-C
  - Deliver the signal to certain threads in the process.
    - e.g., Pthreads provides **pthread_kill(threadid, signal)**
  - Assign a specific thread to receive all signals for the process.
    - e.g., Solaris 2

4.6.3 Thread Cancellation

- **The task of terminating a thread before it has completed.**
- **Thread to be canceled is target thread**
- **Asynchronous cancellation**
  - One thread immediately terminates the target thread
- **Deferred cancellation**
  - The target thread periodically checks whether it should terminate
  - Allows a thread to check if it should be canceled at a point at which it can be canceled safely.
  - Example: Pthreads refers to such points as cancellation points.

Thread Cancellation

- **Pthread code to create and cancel a thread:**

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

  . . .

/* cancel the thread */
pthread_cancel(tid);

/* wait for the thread to terminate */
pthread_join(tid,NULL);
```

Thread Cancellation (Cont.)

- **Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state**

Mode	State	Type
Off	Disabled	–
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- **If thread has cancellation disabled, cancellation remains pending until thread enables it**
- **Default type is deferred**
  - Cancellation only occurs when thread reaches **cancellation point**
    - i.e., `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- **On Linux systems, thread cancellation is handled through signals**

Thread Cancellation in Java

- Deferred cancellation uses the `interrupt()` method, which sets the interrupted status of a thread.

```
Thread worker;

 . . .

/* set the interruption status of the thread */
worker.interrupt()
```

- A thread can then check to see if it has been interrupted:

```
while (!Thread.currentThread().isInterrupted()) {
    . . .
}
```

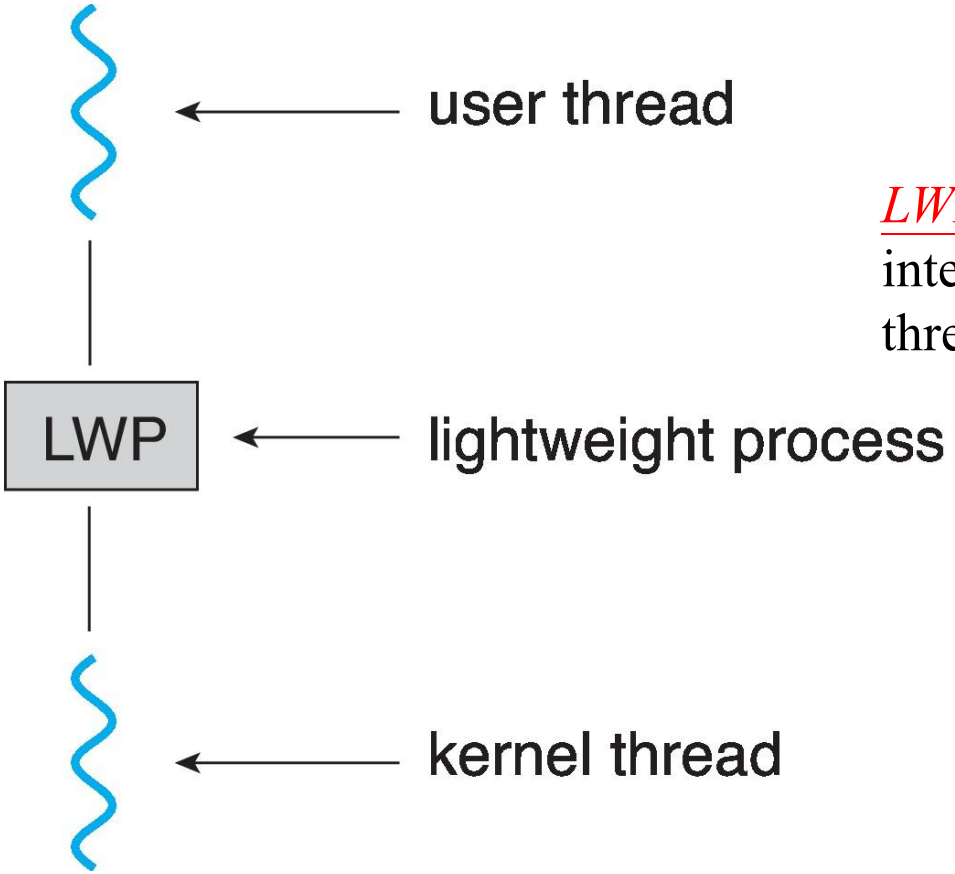
4.6.4 Thread-Local Storage

- **Each thread might need its own copy of certain data in some circumstances – thread-local storage (TLS).**
  - Visible across function invocations
- **Most thread libraries – including Windows and Pthreads – provide some form of support for thread-local storage.**
- **Java provides support as well.**

4.6.5 Scheduler Activations

- **Concerns communication between the kernel and the thread library.**
- **Both many-to-many and two-level models require communication to maintain the appropriate number of kernel threads allocated to the application**
- **Typically use an intermediate data structure between user and kernel threads – lightweight process (LWP)**
  - Appears to be a virtual processor on which the application can schedule a user thread to run
  - Each LWP is attached to a kernel thread

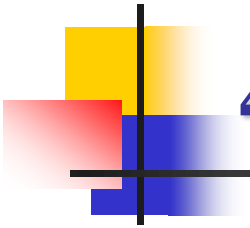
Lightweight process (LWP)



LWP (lightweight processes) - intermediate level between user-level threads and kernel-level threads.

Scheduler Activations (Cont.)

- **Scheduler activations provide upcalls - a communication mechanism from the kernel to the thread library**
  - Upcalls are handled by the thread library with an *upcall handler*
- **This communication allows an application to maintain the correct number kernel threads**



4.7 Operating-System Examples

- **Explore how threads are implemented in Windows and Linux systems.**
  - 4.7.1 Windows Threads
  - 4.7.2 Linux Threads

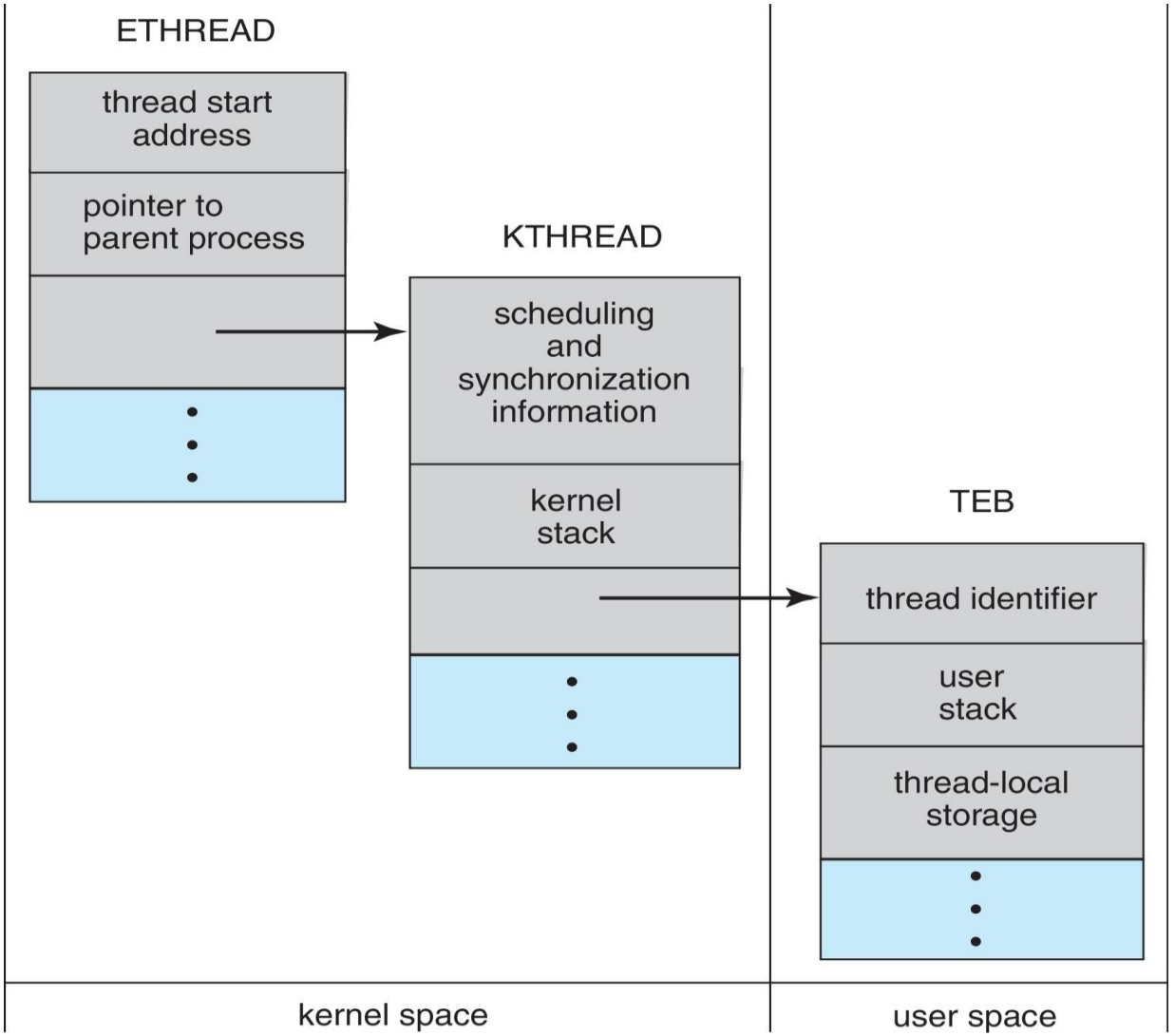
4.7.1 Windows Threads

- Windows implements the Windows API.
- A Windows application runs as a separate process, and each process may contain one or more threads.
- Implements the one-to-one mapping, where each user-level thread maps to an associated kernel thread.

Windows Threads (Cont.)

- **Each thread contains**
  - A thread id
  - Register set representing state of processor
  - Separate user and kernel stacks for when thread runs in user mode or kernel mode
  - Private data storage area used by run-time libraries and dynamic link libraries (DLLs)
- **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)

Data structures of a Windows thread



4.7.2 Linux Threads

- **Linux does not distinguish between processes and threads.**
  - Linux generally uses the term ***tasks*** rather than *process* or *threads* when referring to a flow of control within a program.
- **Thread creation is done through clone() system call.**
- **clone() allows a child task to share the address space of the parent task (process).**
  - the passed flags determine how much sharing between parent & child

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.

Homework

- **10th edition**
  - Ex. 4.4, 4.5, 4.6, 4.9, 4.11, 4.17, 4.19
  - Ex. 4.19 手寫 (40%) 或程式實作 (50%)
    - if (手寫)
      - 最高得 40分
    - else
      - 最高得 50分
    - 請印出執行結果畫面 (使用自己的帳號和虛擬機).
    - 請交原始程式

EX 4.19

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

int value = 0;
void *runner(void *param); /* the thread */

int main(int argc, char *argv[])
{
pid_t pid;
pthread_t tid;
pthread_attr_t attr;

   pid = fork();

   if (pid == 0) { /* child process */
      pthread_attr_init(&attr);
      pthread_create(&tid,&attr,runner,NULL);
      pthread_join(tid,NULL);
      printf("CHILD: value = %d",value); /* LINE C */
   }
   else if (pid > 0) { /* parent process */
      wait(NULL);
      printf("PARENT: value = %d",value); /* LINE P */
   }
}

void *runner(void *param) {
   value = 5;
   pthread_exit(0);
}
```