Introduction

Mr. SIDDESHA K

Assistant Professor

Department of ECE

Dr AIT, Bengaluru.

MULTICORE ARCHITECTURE

Course Plan

- Introduction to Multi-Core Architecture
- System Overview of Threading
- Fundamental Concepts of Parallel Programming
- Threading and Parallel Programming Constructs
- Threading APIs
- Open MP: A Portable Solution for Threading
- Solutions to Common Parallel Programming problems

Course content

- This course content is organized into three major sections.
- The first section (Chapters 1–4) presents an introduction to software threading.
- This section includes background material on, why chipmakers have shifted to multi-core architectures, how threads work, how to measure the performance improvements achieved by a particular threading implementation.
- Overall understanding why hardware platforms are evolving in the way that they are and understanding the basic principles required to write parallel programs.

Course content

- The next section (Chapters 5 and 6) discusses common programming APIs for writing parallel programs. We look at different programming interfaces: Microsoft's APIs for Win32, MFC, and .NET; POSIX Threads; and OpenMP.
- The third and final section is a collection of topics related to multicore programming. Chapter 7 discusses common parallel programming problems and how to solve them.

Text Books

TEXT BOOK:

1. "**Multicore Programming-Increased Performance through Software Multi-threading**", Shameem Akhter and Jason Roberts , Intel Press, 2006.

REFERENCE BOOKS:

1. Calvin Lin, Lawrence Snyder, "Principles of Parallel Programming" Pearson Education, 2009. ISBN-13: 978-0321487902.

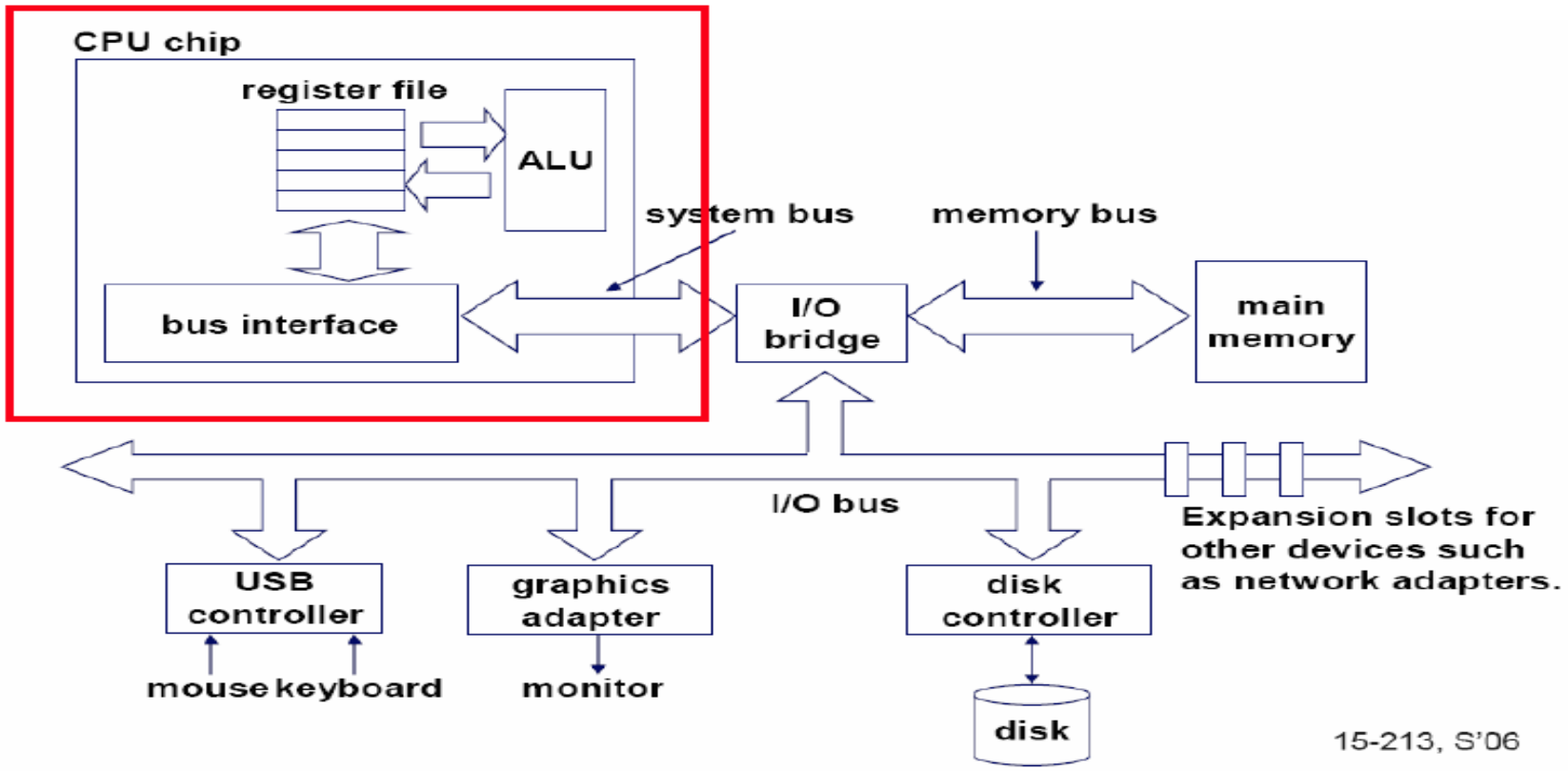
2. Michael J. Quinn ,"Parallel Programming in C with MPI and OpenMP", Tata McGraw Hill, 2004. ISBN 13: 978007058201 9.

3. David E, Culler, Jaswinder Pal Singh with Anoop Gupta "Parallel Computer Architecture A Hardware/ Software Approach", eBook ISBN: 9780080573076 Hardcover ISBN: 9781558603431.

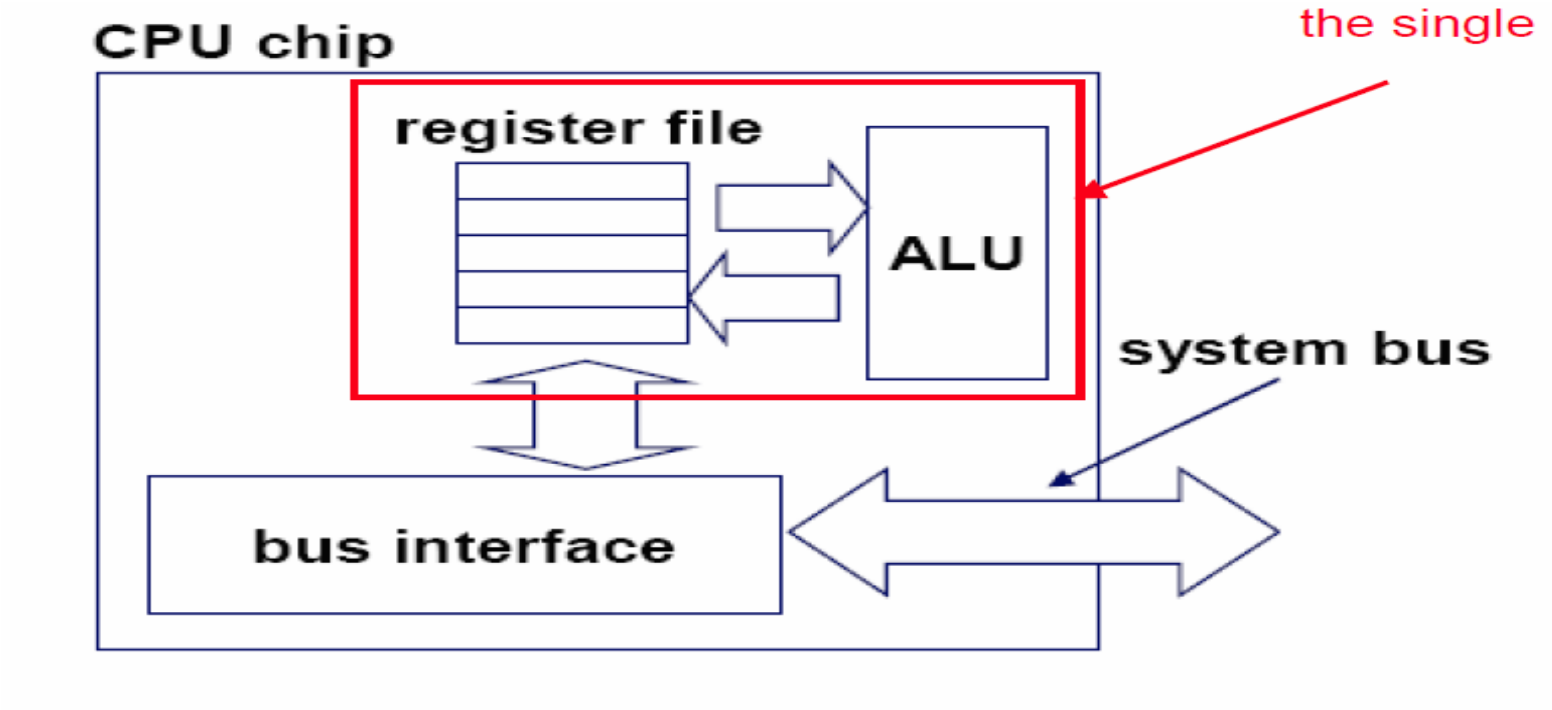
What is MULTICORE ARCHITECTURE?

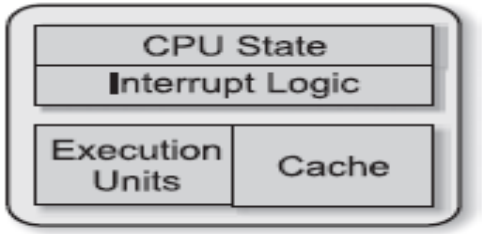
- Most technology professionals have heard of the radical transformation taking place in the way that modern computing platforms are being designed.
- Intel, IBM, Sun, and AMD have all introduced microprocessors that have multiple execution cores on a single chip.
- In the future, computing platforms, whether they are desktop, mobile, server, or specialized embedded platforms are most likely to be multi-core in nature.
- The fact that the hardware industry is moving in this direction presents new opportunities for software developers.
- As a result, multi-threading was an effective illusion.
- With modern multi-core architectures, developers are now presented with a truly parallel computing platform.

Single-core computer



Single-core CPU chip

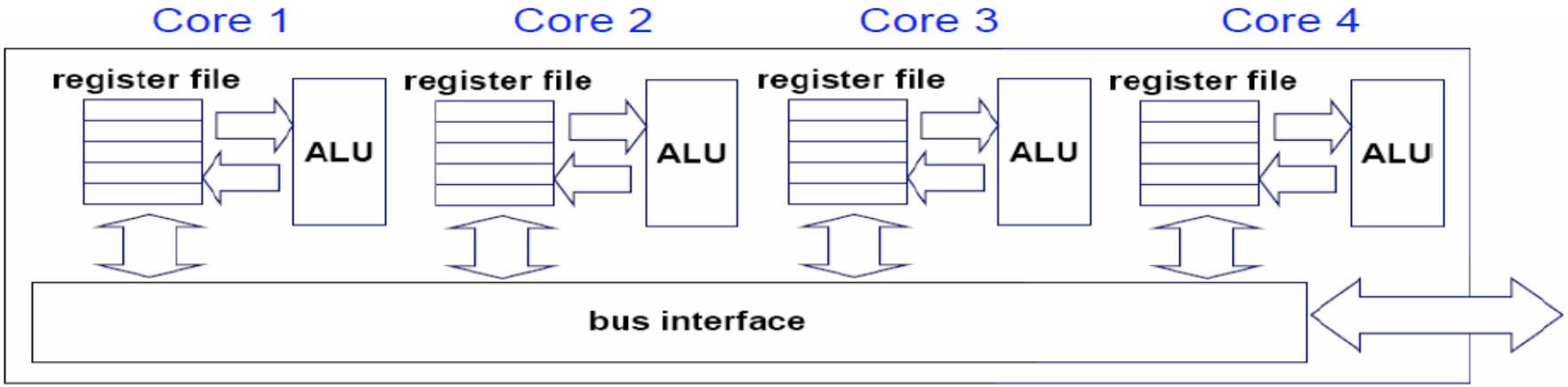




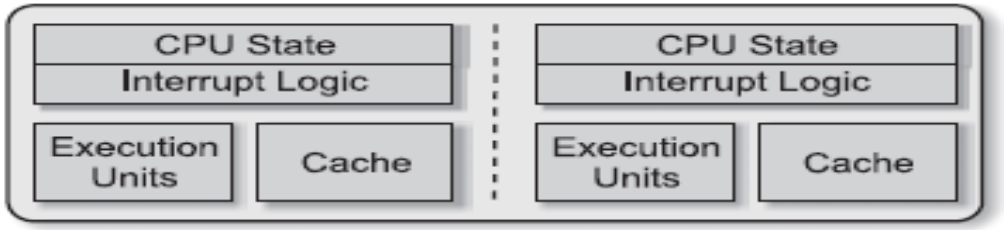
A) Single Core

Multi-core architectures

- This lecture is about a new trend in computer architecture:
  Replicate multiple processor cores on a single die.



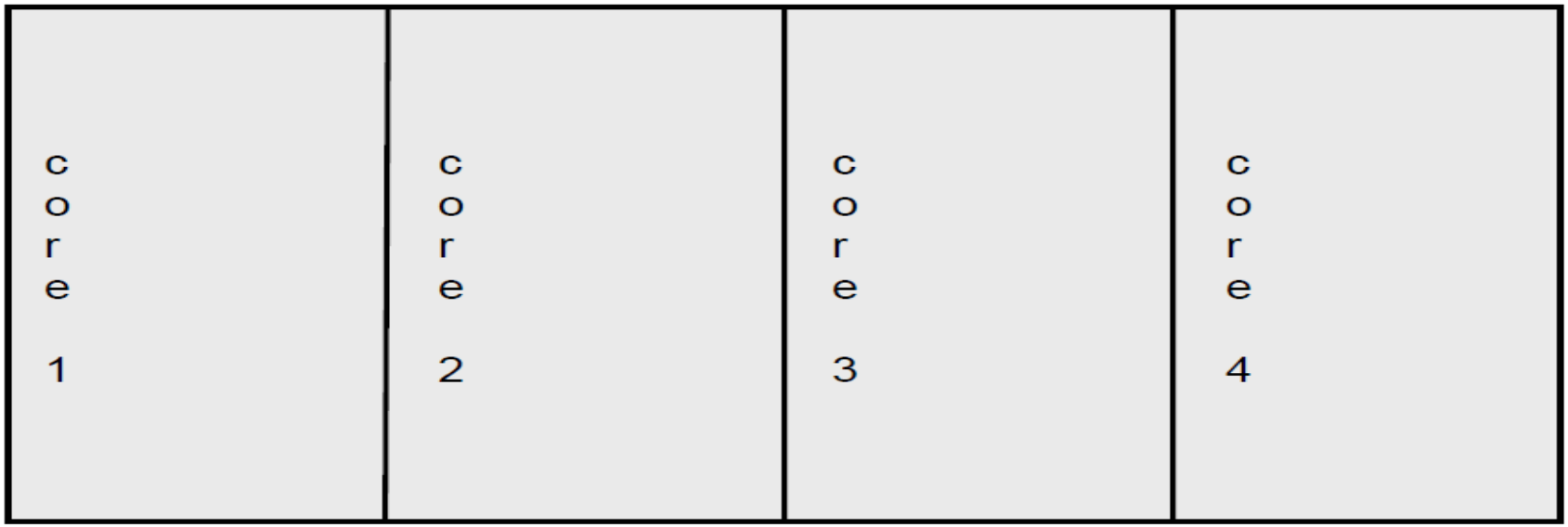
Multi-core CPU chip



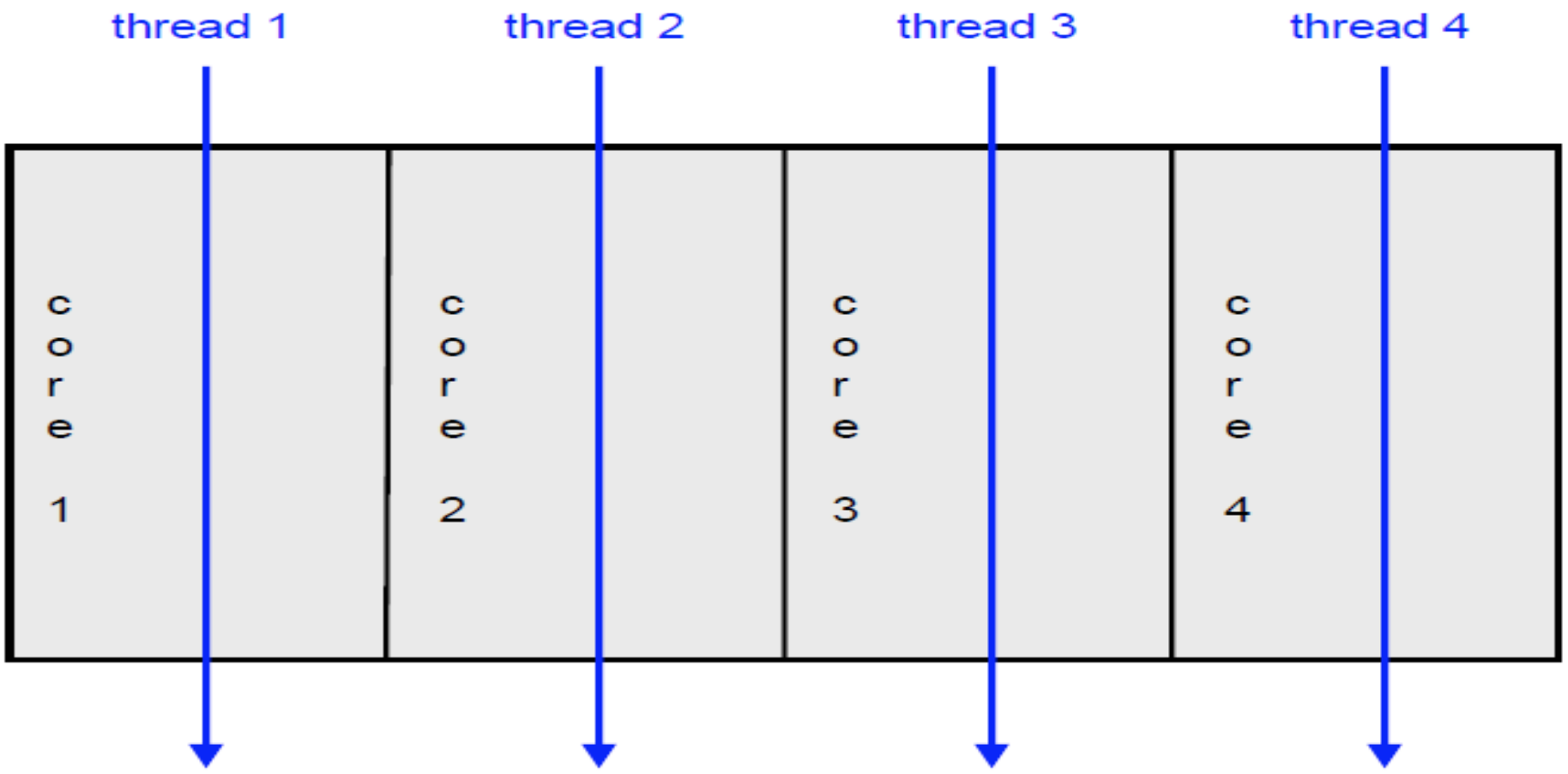
D) Multi-core

Multi-core CPU chip

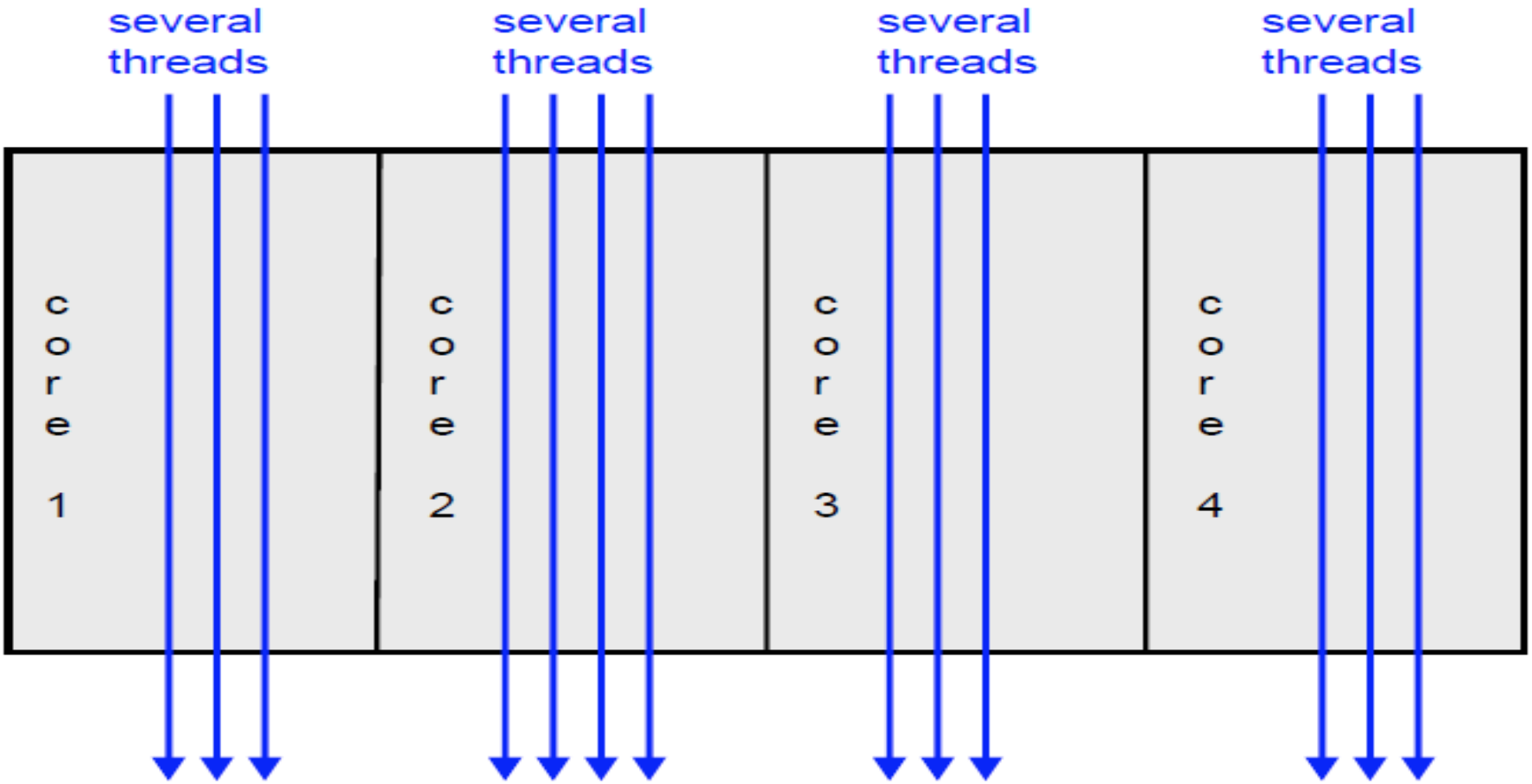
- The cores fit on a single processor socket
- Also called CMP (Chip Multi-Processor)



The cores run in parallel



Within each core, threads are time-sliced (just like on a uniprocessor)



Interaction with OS

- OS perceives each core as a separate processor
- OS scheduler maps threads/processes to different cores
- Most major OS support multi-core today

Why multi-core ?

- Difficult to make single-core clock frequencies even higher
- Deeply pipelined circuits:
  - heat problems
  - speed of light problems
  - difficult design and verification
  - large design teams necessary
  - server farms need expensive air-conditioning

- Many new applications are multithreaded
- General trend in computer architecture (shift towards more parallelism)

Instruction-level parallelism

- Parallelism at the machine-instruction level
- The processor can re-order, pipeline instructions, split them into microinstructions, do aggressive branch prediction, etc.
- Instruction-level parallelism enabled rapid increases in processor speeds over the last 15 years

Thread-level parallelism (TLP)

- This is parallelism on a more coarser scale
- Server can serve each client in a separate thread (Web server, database server)
- A computer game can do AI, graphics, and physics in three separate threads
- Single-core superscalar processors cannot fully exploit TLP
- Multi-core architectures are the next step in processor evolution: explicitly exploiting TLP

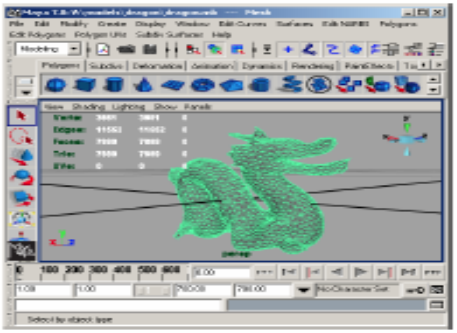
Multiprocessor memory types

- Shared memory:
  In this model, there is one (large) common shared memory for all processors

- Distributed memory:
  In this model, each processor has its own (small) local memory, and its content is not replicated anywhere else

What applications benefit from multi-core?

- Database servers
- Web servers (Web commerce)
- Compilers
- Multimedia applications
- Scientific applications, CAD/CAM
- In general, applications with *Thread-level parallelism* (as opposed to instruction-level parallelism)



Each can run on its own core

More examples

- Editing a photo while recording a TV show through a digital video recorder
- Downloading software while running an anti-virus program
- “Anything that can be threaded today will map efficiently to multi-core”
- BUT: some applications difficult to parallelize

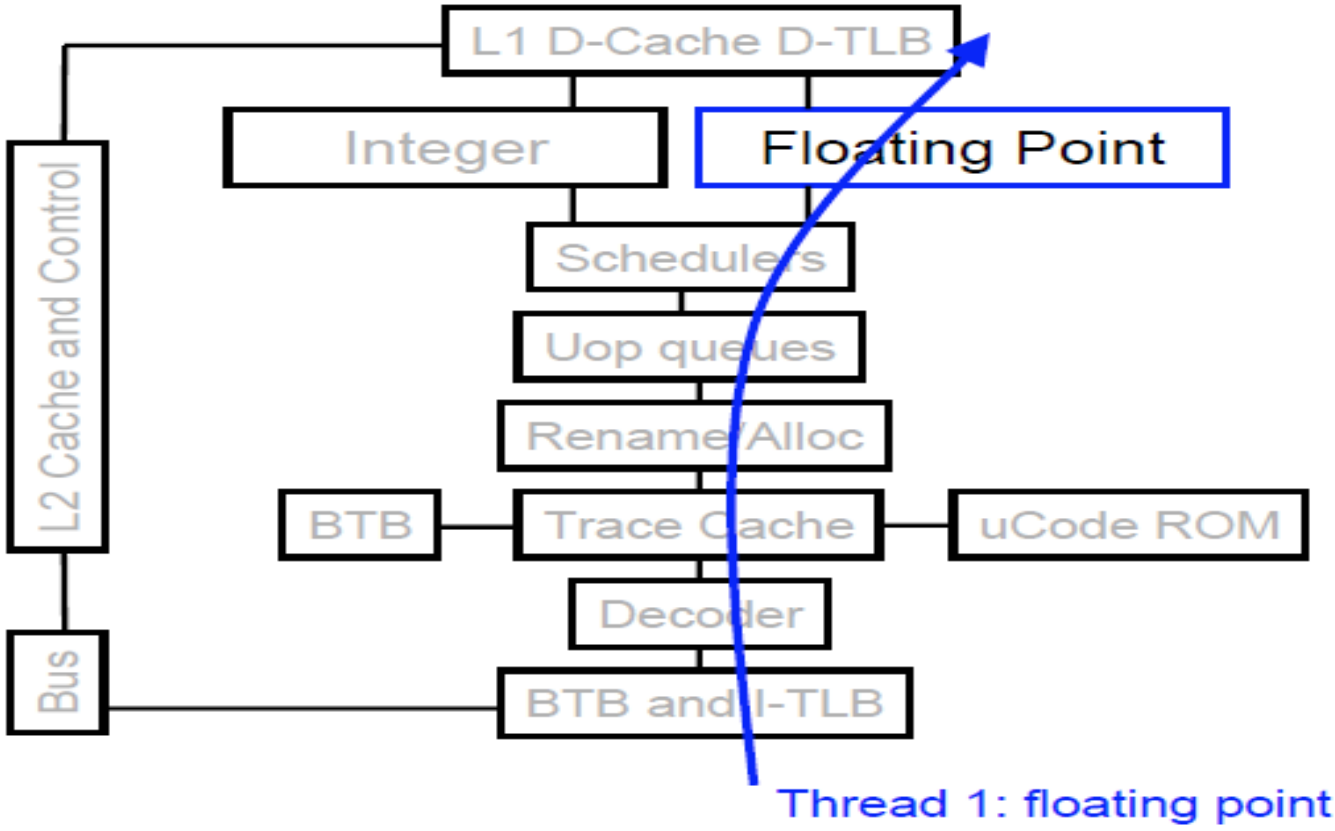
Simultaneous multithreading (SMT)

- Permits multiple independent threads to execute SIMULTANEOUSLY on the SAME core
- Weaving together multiple “threads” on the same core

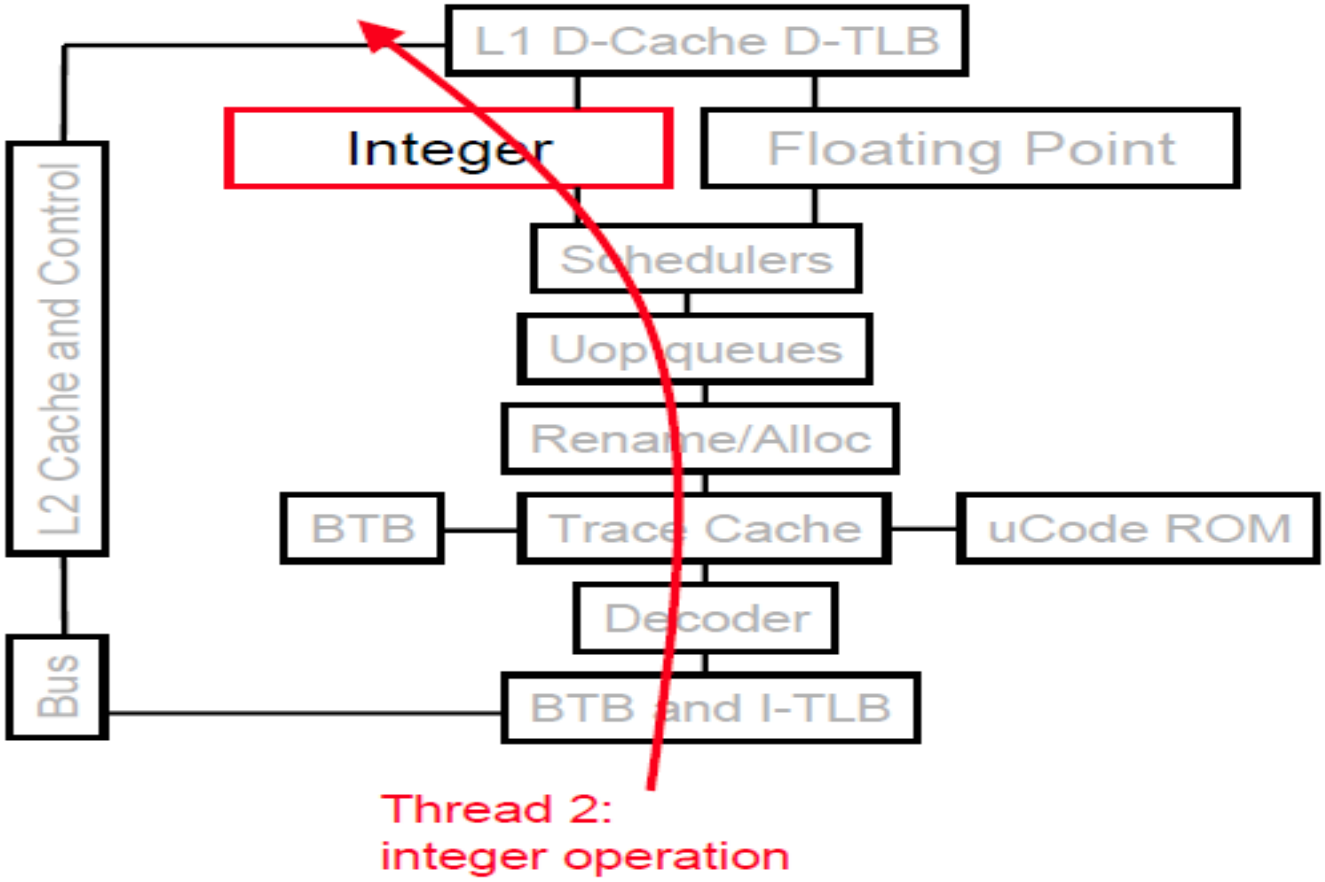
- Example: if one thread is waiting for a floating point operation to complete, another thread can use the integer units

Complementary to multi-core

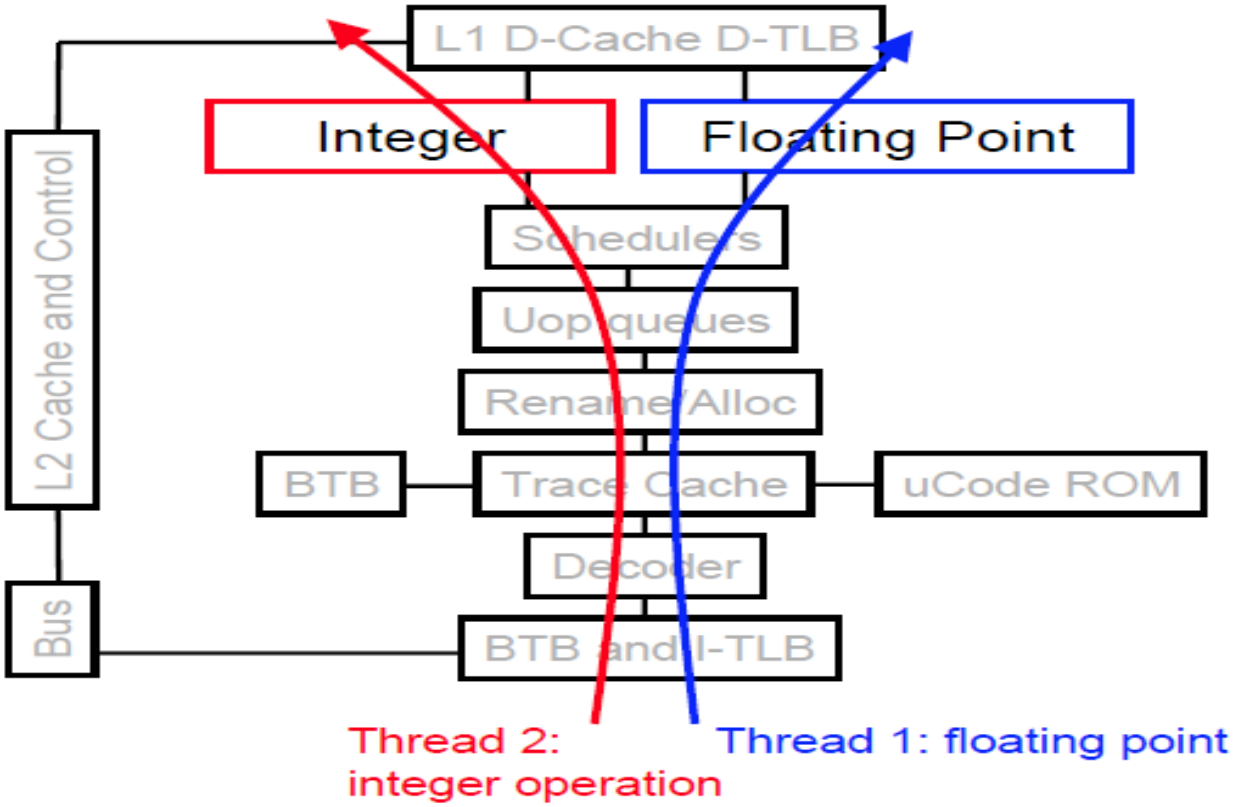
Without SMT, only a single thread can run at any given time

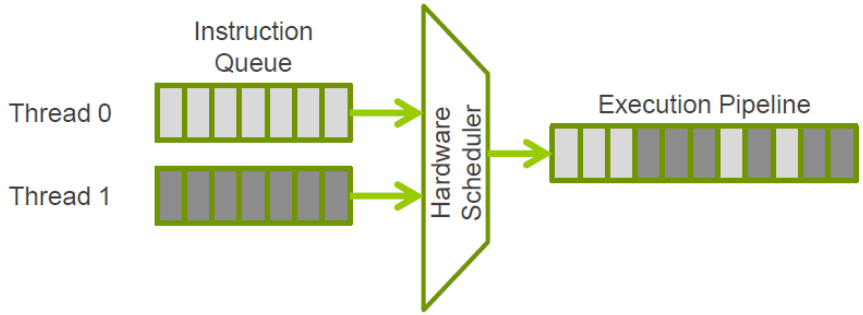


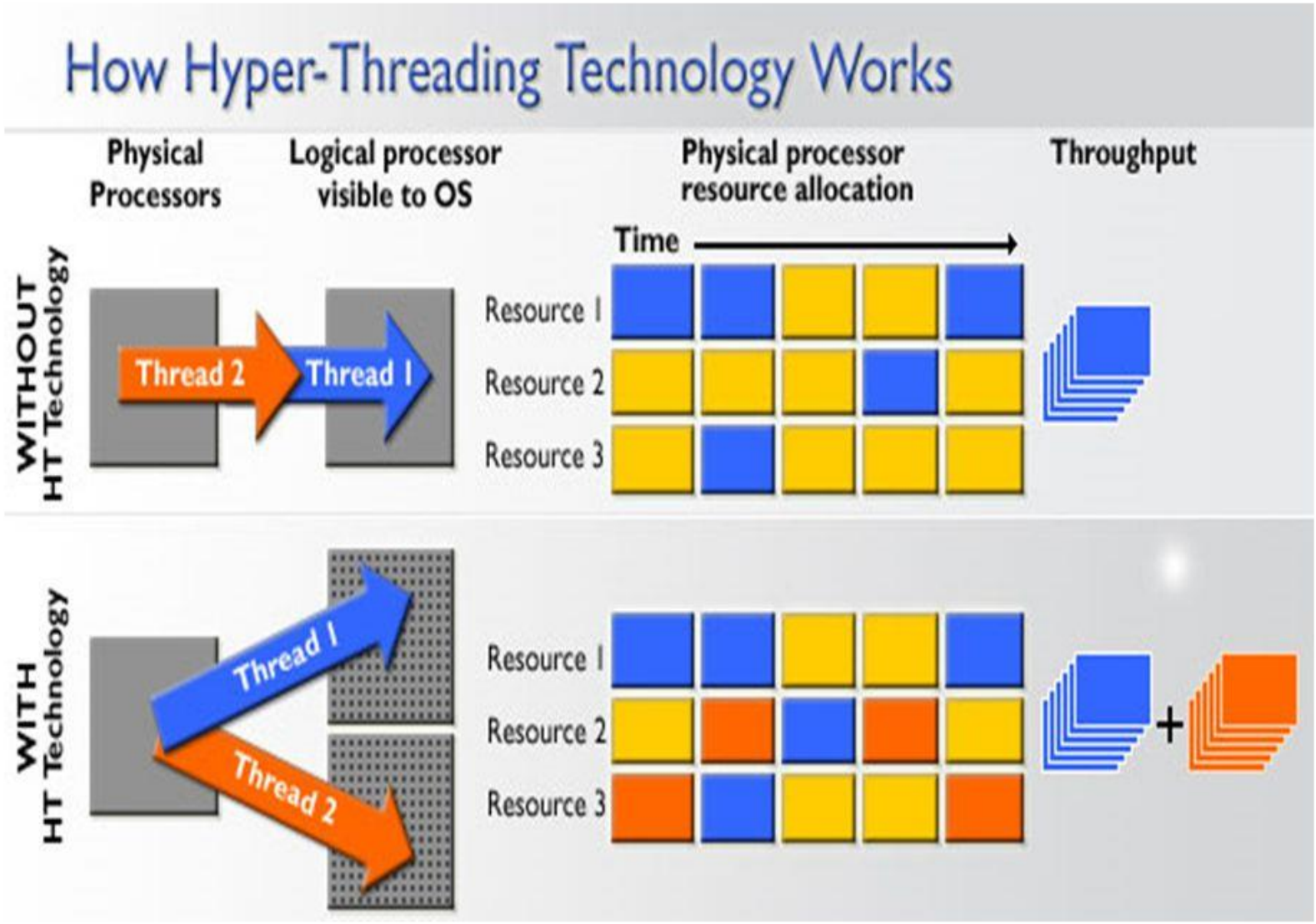
Without SMT, only a single thread can run at any given time



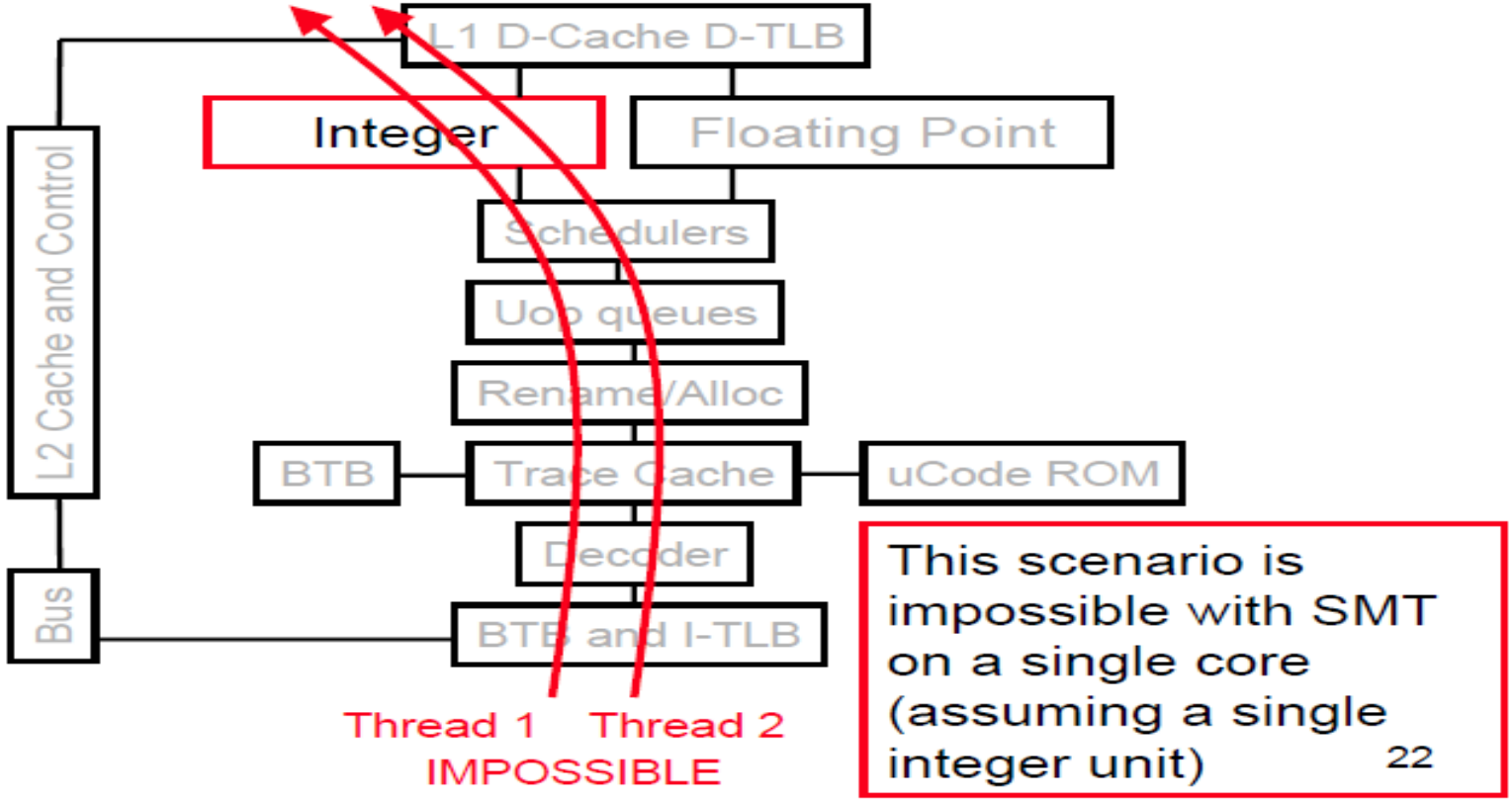
SMT processor: both threads can run concurrently





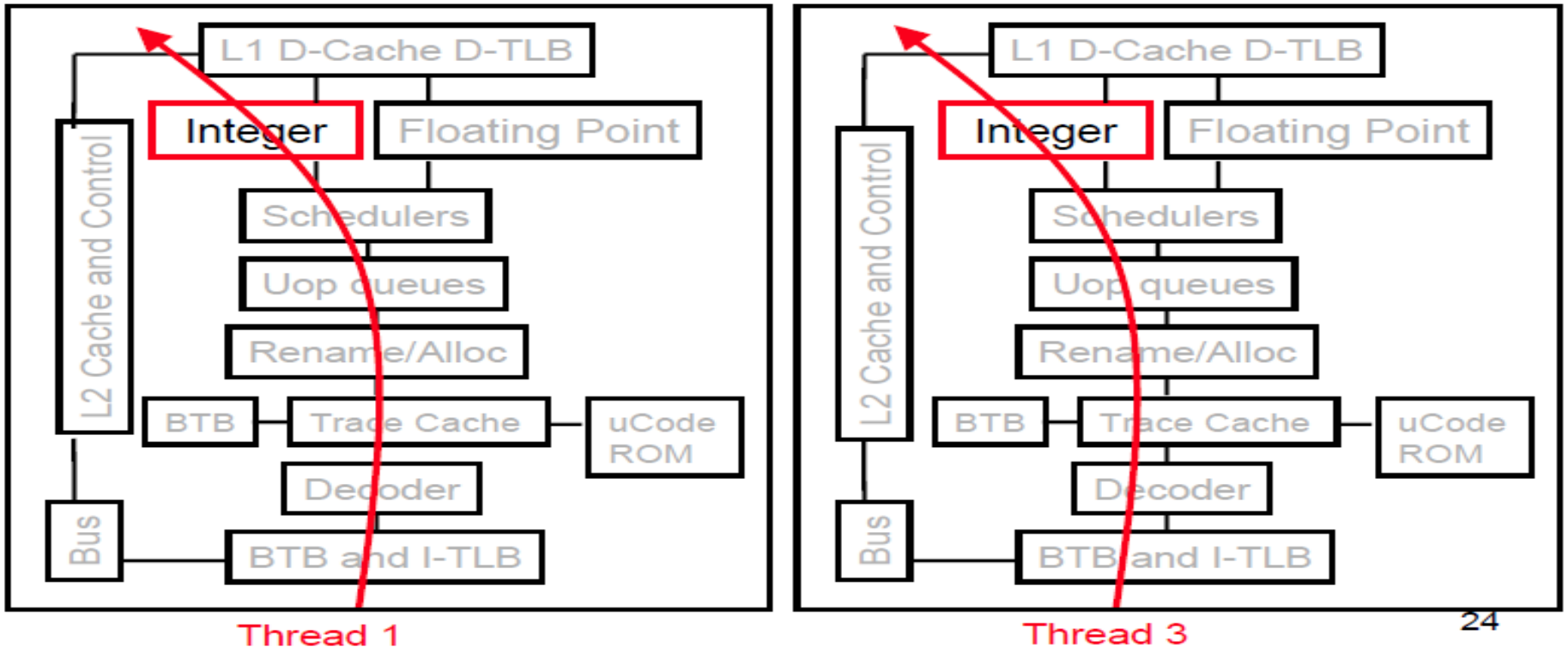

How Hyper-Threading Technology Works
Physical Processors
Logical processor visible to OS
Physical processor resource allocation
Throughput
Time
WITHOUT HT Technology
Thread 2
Thread 1
Resource 1
Resource 2
Resource 3
WITH HT Technology
Thread 1
Thread 2
Resource 1
Resource 2
Resource 3
+

But: Can't simultaneously use the same functional unit



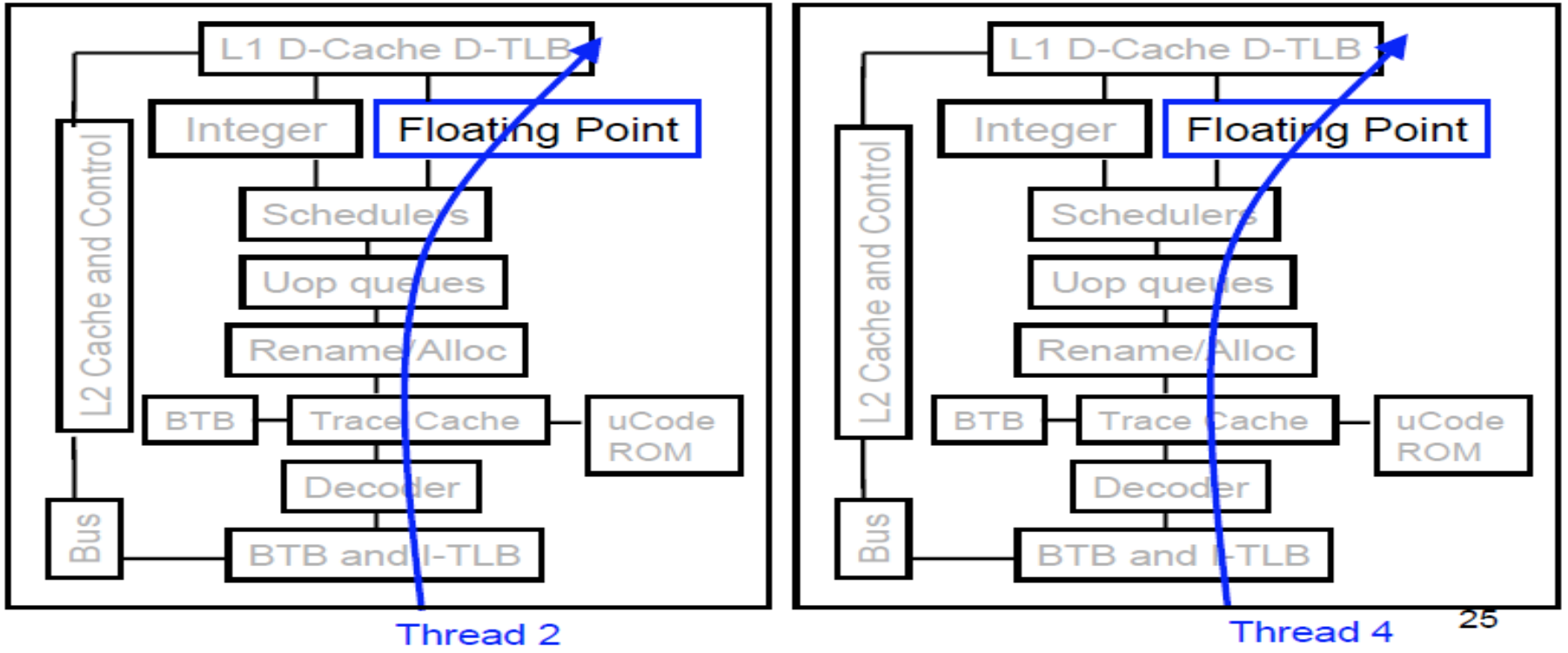
SMT not a "true" parallel processor

Multi-core:
threads can run on separate cores



"The next logical step from simultaneous multi-threading (SMT) is the multi-core processor"

Multi-core:
threads can run on separate cores



Optimal application performance on multi-core architectures will be achieved by effectively using threads to partition software workloads.

Comparison: multi-core vs SMT

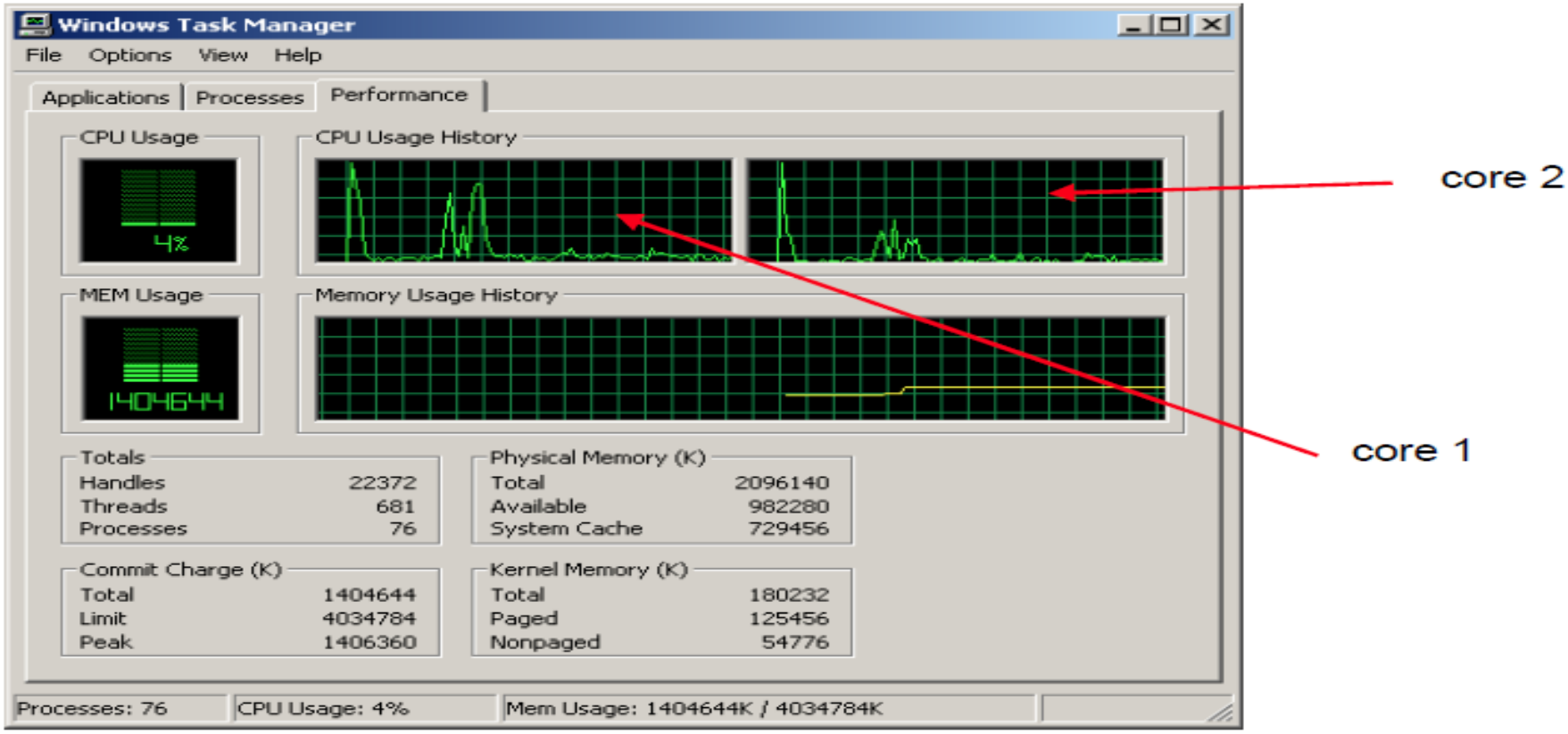
- Multi-core:
  - Since there are several cores, each is smaller and not as powerful (but also easier to design and manufacture)
  - However, great with thread-level parallelism
- SMT
  - Can have one large and fast superscalar core
  - Great performance on a single thread
  - Mostly still only exploits instruction-level parallelism

Programming for multi-core

- Programmers must use threads or processes
- Spread the workload across multiple cores
- Write parallel algorithms
- OS will map threads/processes to cores

We'll take a look at a variety of topics that are relevant to writing software for multi-core platforms.

Windows Task Manager



Legal licensing issues

- Will software vendors charge a separate license per each core or only a single license per chip?

- Microsoft, Red Hat Linux, Suse Linux will license their OS per chip, not per core

- Multi-core chips an important new trend in computer architecture

- Several new multi-core chips in design phases

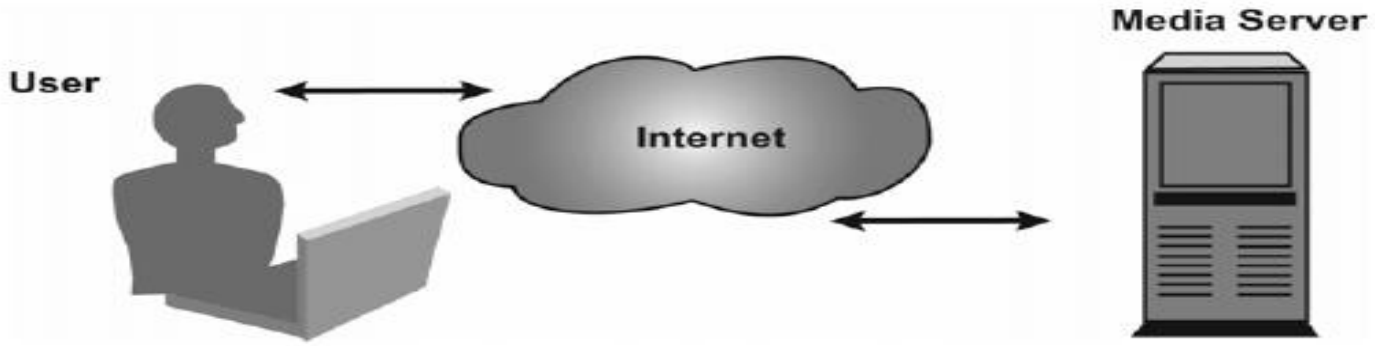
- Parallel programming techniques likely to gain importance

Parallel programming

- Implementing software effectively and efficiently on parallel hardware platforms.
- These platforms include multi-core processors and processors that use simultaneous multi-threading techniques, such as Hyper- Threading Technology (HT Technology).
- This course will focus on programming techniques that allow the developer to exploit the capabilities provided by the underlying hardware platform.

Motivation for Concurrency

- Most end users have a simplistic view of complex computer systems.

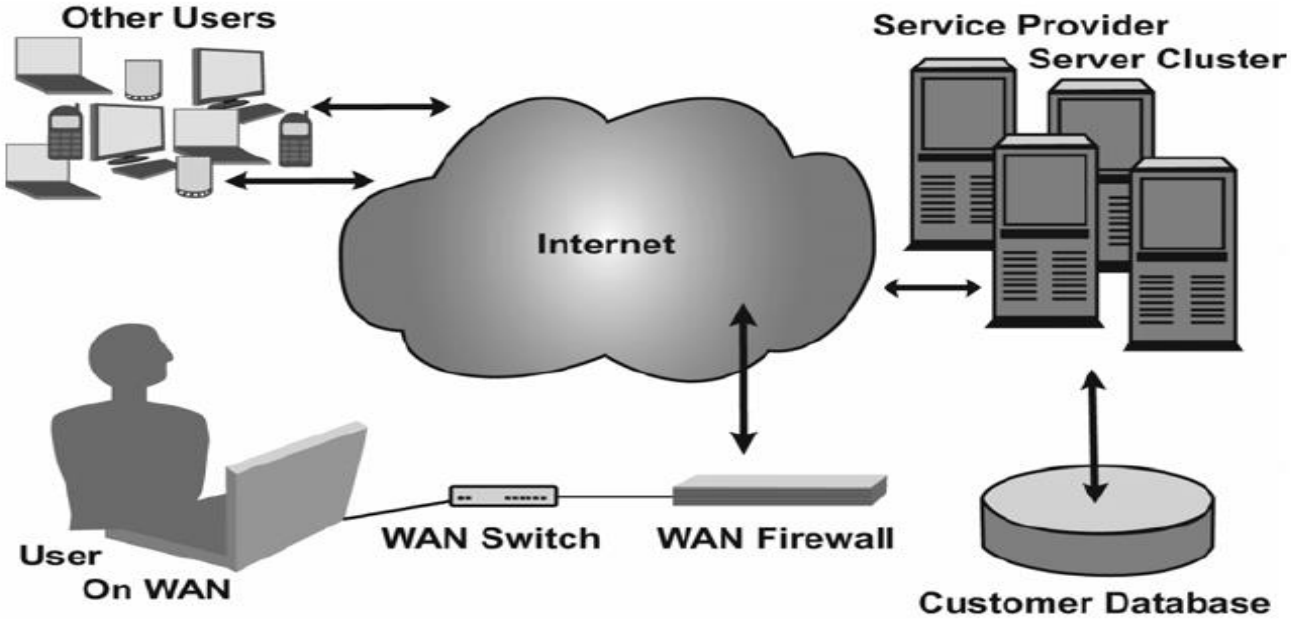


End User View of Streaming Multimedia Content via the Internet

- On the server side, the provider must be able to receive the original broadcast, encode/compress it in near real-time, and then send it over the network to potentially hundreds of thousands of clients.

Motivation for Concurrency

- A system designer who is looking to build a computer system capable of streaming a Web broadcast might look at the system as it's shown,

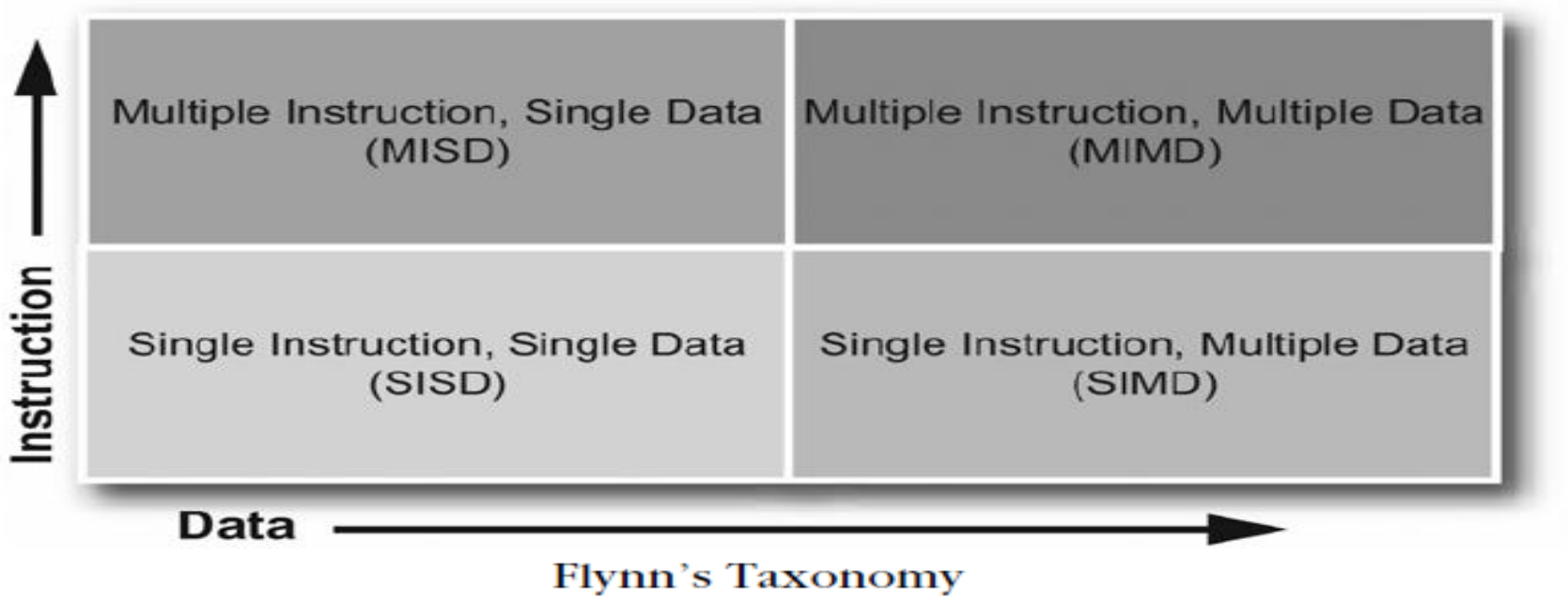


End-to-End Architecture View of Streaming Multimedia Content over the Internet

- In order to provide an acceptable end-user experience, system designers must be able to effectively manage many independent subsystems that operate in parallel.

Parallel Computing Platforms

- In order to achieve parallel execution in software, hardware must provide a platform that supports the simultaneous execution of multiple threads.
- Any given computing system can be described in terms of how instructions and data are processed. This classification system is known as Flynn's taxonomy.



Flynn's Taxonomy

The different processor architectures

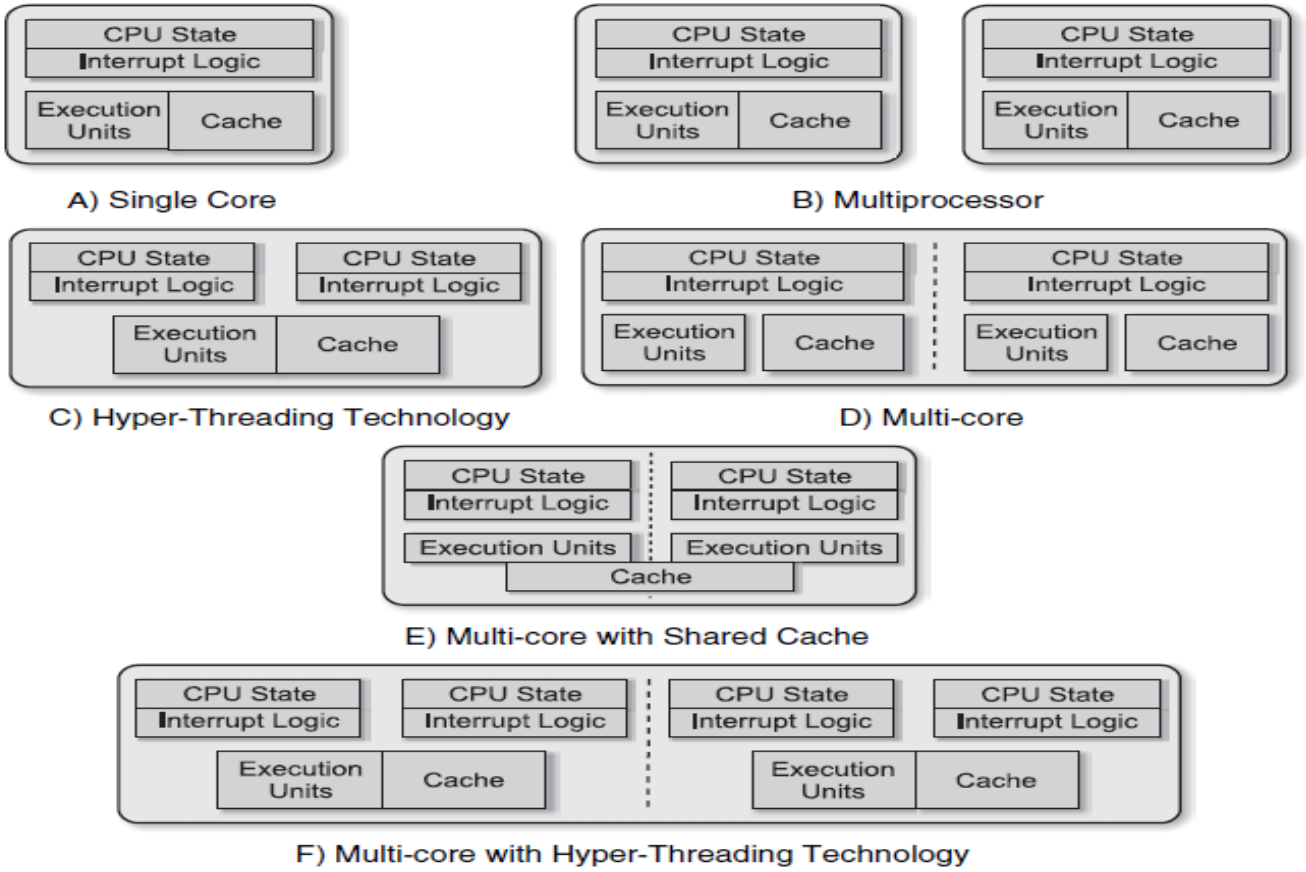


Figure 1.4 Simple Comparison of Single-core, Multi-processor, and Multi-Core Architectures

Assignment questions

1. Differentiate Multi-Core Architectures and Hyper-Threading Technology.

2. Differentiate Multi-threading on Single-Core and multithreading on Multi-Core Platforms.

Understanding Performance

- How do I measure the performance benefit of parallel programming?
- One metric is to compare the elapsed run time of the best sequential algorithm versus the elapsed run time of the parallel program.
- This ratio is known as the speedup and characterizes how much faster a program runs when parallelized.

$$\text{Speedup(nt)} = \frac{Time_{best\_sequential\_algorithm}}{Time_{parallel\_implementation(nt)}}$$

- Speedup is defined in terms of the number of physical threads (nt) used in the parallel implementation.

Amdahl's Law

- In 1967 Gene Amdahl proposed a rule known as Amdahl's Law, examines the maximum theoretical performance benefit of a parallel solution relative to the best case performance of a serial solution.
- Amdahl started with the intuitively clear statement that "program speedup is a function of the fraction of a program that is accelerated and by how much that fraction is accelerated".

Speed up part of the program (Only some Instructions)

$$\text{Speedup} = \frac{1}{(1 - \text{Fraction}_{Enhanced}) + (\text{Fraction}_{Enhanced} / \text{Speedup}_{Enhanced})}$$

So, if you could speed up half the program by 15 percent, you'd get:

$$\text{Speedup} = 1/((1 - .50) + (.50/1.15)) = 1/(.50 + .43) = 1.08$$

This result is a speed increase of 8 percent.

- If half of the program is improved 15 percent, then the whole program is improved by half that amount.

Amdahl's Law example: Make the common case fast

- Fraction = 0.1, Speedup = 10

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \frac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}$$

$$= \frac{1}{(1-0.1) + \frac{0.1}{10}} = \frac{1}{0.91} = 1.1$$

- Fraction = 0.9, Speedup = 10

$$\text{Speedup}_{\text{overall}} = \frac{1}{(1-0.9) + \frac{0.9}{10}} = \frac{1}{0.19} = 5.3$$

Amdahl's Law

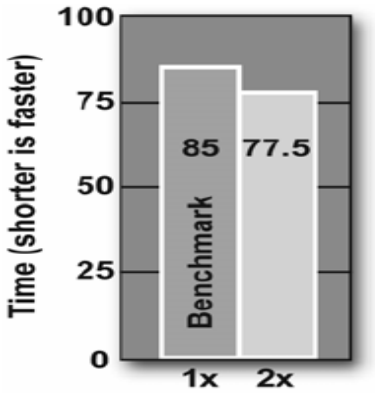
Equation 1.1 Amdahl's Law

$$\text{Speedup} = \frac{1}{S + (1 - S)/n}$$

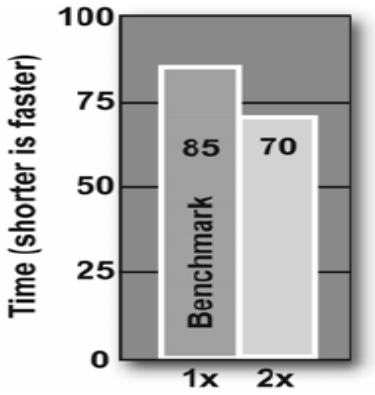
- In this equation, *S* is the time spent executing the serial portion of the parallelized version and *n* is the number of processor cores.
- Setting n = ∞ in Equation 1.1

$$\text{Speedup} = \frac{1}{S}$$

- Decreasing the serialized portion by increasing the parallelized portion is of greater importance than adding more processor cores.



Performance benefit of doubling the number of processor cores



Performance benefit of doubling the amount of parallelism in code

Note: The advantage gained by writing parallel code

Amdahl's Law

- To make Amdahl's Law reflect the reality of multi-core systems, system overhead from adding threads should be included:

$$\text{Speedup} = \frac{1}{S + (1 - S)/n + H(n)}$$

where H(n) = System overhead from adding threads

- This overhead consists of two portions: the actual operating system overhead and inter-thread activities, such as synchronization and other forms of communication between threads.
- Notice that if the overhead is big enough, it offsets the benefits of the parallelized portion.
- This is very common in poorly architected multi-threaded applications.
- The important implication is that the overhead introduced by threading must be kept to a minimum.

Amdahl's Law Applied to Hyper-Threading Technology

- Amending Amdahl's Law to fit HT Technology, then, you get:

$$\text{Speedup}_{\text{HTT}} = \frac{1}{S + 0.67((1-S)/n) + H(n)}$$

where n = number of logical processors.

- This equation represents the typical speed-up for programs running on processor cores with HT Technology performance.
- The value of H(n) is determined empirically and varies from application to application.

Gustafson's Law

- Gustafson's Law has been shown to be equivalent to Amdahl's Law.
- However, Gustafson's Law offers a much more realistic look at the potential of parallel computing on multi-core processors.

$$\text{Scaled speedup} = N + (1 - N) * s$$

where N = is the number of processor cores

s= is the ratio of the time spent in the serial port of the program versus the total execution time.

System Overview of Threading

- When implemented properly, threading can enhance performance by making better use of hardware resources.
- However, the improper use of threading can lead to degraded performance, unpredictable behavior, and error conditions that are difficult to resolve.
- Fortunately, if you are equipped with a proper understanding of how threads operate, you can avoid most problems and derive the full performance benefits that threads offer.
- The concepts of threading starts from hardware and works its way up through the operating system and to the application level.
- In reality threading can be simple, once you grasp the basic principles.

Defining Threads

- A *thread* is a discrete sequence of related instructions that is executed independently of other instruction sequences.
- Every program has at least one thread—the main thread—that initializes the program and begins executing the initial instructions.
- That thread can then create other threads that perform various tasks, or it can create no new threads and simply do all the work itself.
- In either case, every program has at least one thread. Each thread maintains its current machine state.

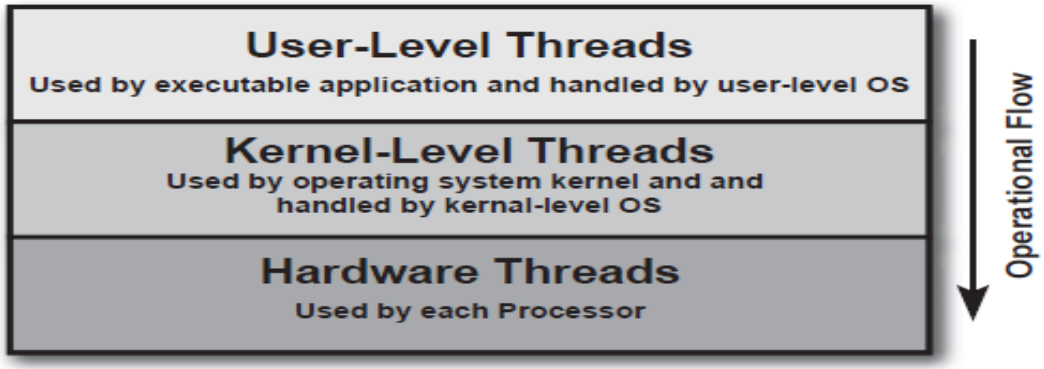
System View of Threads

- The thread computational model is having three layers for threading:

User-level threads: Threads created and manipulated in the application software.

Kernel-level threads: The way the operating system implements most threads.

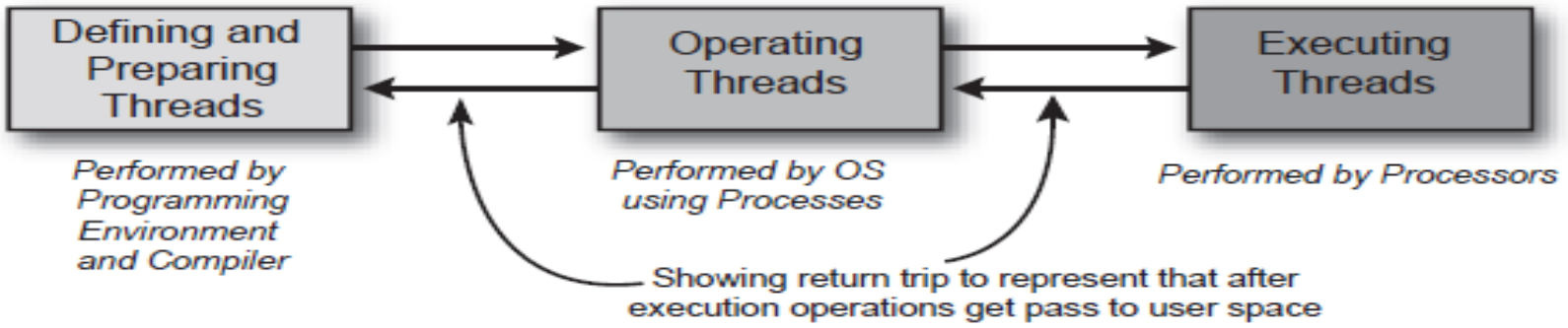
Hardware threads: How threads appear to the execution resources in the hardware.



Computation Model of Threading

Flow of Threads in an Execution Environment

- In the Defining and Preparing stage, threads are specified by the programming environment and encoded by the compiler.
- During the Operating stage, threads are created and managed by the operating system.
- Finally, in the Executing stage, the processor executes the sequence of thread instructions.



Threading above the Operating System

- In general, application threads can be implemented at the application level using established APIs.
- The most common APIs are OpenMP and explicit low-level threading libraries such as Pthreads and Windows threads.
- The choice of API depends on the requirements and the system platform.
- OpenMP, in contrast, offers ease of use and a more developer-friendly threading implementation.
- OpenMP requires a compiler that supports the OpenMP API. Today, these are limited to C/C++ and Fortran compilers.
- To show how threading is used in a program, considered simple “Hello World” programs that use the OpenMP and Pthreads libraries.

"Hello World" Program Using OpenMP

```
#include <stdio.h>
// Have to include 'omp.h' to get OpenMP definitons
#include <omp.h>
void main()
{
     int threadID, totalThreads;
  /* OpenMP pragma specifies that following block is
     going to be parallel and the threadID variable is
     private in this openmp block. */

     #pragma omp parallel private(threadID)
     {
        threadID = omp_get_thread_num();
        printf("\nHello World is from thread %d\n",
                (int)threadID);
           /* Master thread has threadID = 0 */
        if (threadID == 0) {
            printf("\nMaster thread being called\n");
            totalThreads = omp_get_num_threads();

            printf("Total number of threads are %d\n",
                    totalThreads);
        }
    }
}
```

/Compiler directive/

STEPS TO CREATE A PARALLEL PROGRAM

1. **Include the header file**: We have to include the OpenMP header for our program along with the standard header files.

```
//OpenMP header
#include <omp.h>
```

2. **Specify the parallel region**: In OpenMP, we need to mention the region which we are going to make it as parallel using the keyword **pragma omp parallel**. The **pragma omp parallel** is used to fork additional threads to carry out the work enclosed in the parallel. **The original thread will be denoted as the master thread with thread ID 0.** Code for creating a parallel region would be,

```
#pragma omp parallel
{
  //Parallel region code
}
```

So, here we include

```
#pragma omp parallel
{
    printf("Hello World... from thread = %d\n",
           omp_get_thread_num());
}
```

3. **Set the number of threads**:
we can set the number of threads to execute the program using the external variable.

```
export OMP_NUM_THREADS=5
```

4. **Compile and Run**:

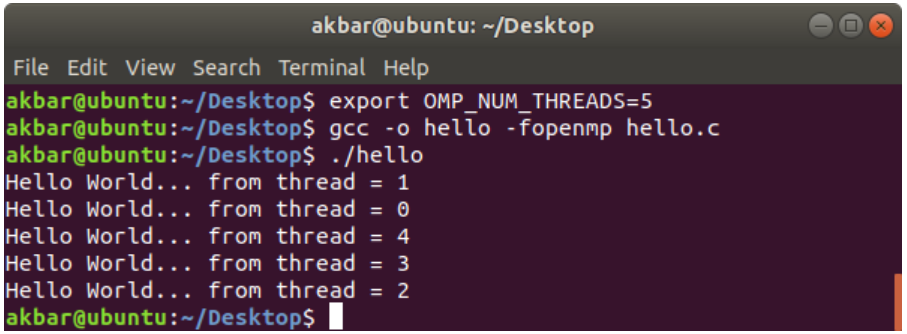
Compile:

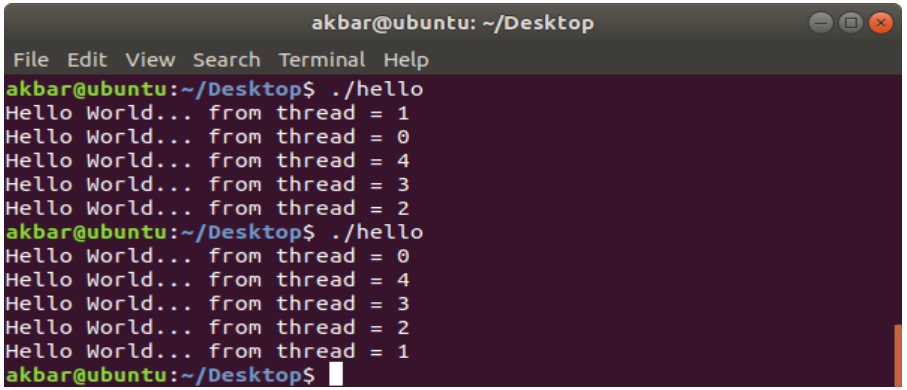
```
gcc -o hello -fopenmp hello.c
```

Execute:

```
./hello
```

```
// OpenMP program to print Hello World
from multiple threads

// using C language

// OpenMP header
#include <omp.h>

#include <stdio.h>
#include <stdlib.h>

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

        // Beginning of parallel region
        #pragma omp parallel
        {

printf("Hello World... from thread = %d\n",

        omp_get_thread_num());
        }
        // Ending of parallel region
}
```





OpenMP: A parallel Hello World Program

```
/* Print Hello World from multiple threads */

/* Include OpenMP header file */
#include <omp.h>
#include <stdio.h>

/* Main function */
int main( int *argc, char *argv[] )
{
    /* Specify the block to be executed in parallel */
    #pragma omp parallel
    {
        /* Print "Hello World" from each thread */
        printf( "Hello World\n" ) ;
    }

    return 0 ;
}
```

```
$ gcc -fopenmp Hello-World-1.c
$ ./a.out
Hello World
Hello World
```

```
$ export OMP_NUM_THREADS=4
$ ./a.out
Hello World
Hello World
Hello World
Hello World
```

"Hello World" Program Using Pthreads

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for (t=0; t < NUM_THREADS; t++) {
      printf("Creating thread %d\n", t);
      rc = pthread_create( &threads[t], NULL,
                          PrintHello,(void *)t);
      if (rc) {
        printf("ERROR return code from pthread_create(): %d\n",
               rc);
        exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

- As can be seen, the OpenMP code has no function that corresponds to thread creation. This is because OpenMP creates threads automatically in the background.
- In Pthreads, where a call to pthread_create() actually creates a single thread and points it at the work to be done in PrintHello().

OUTPUT

```
Compile & run(linux terminal):
gcc -o Pthread_Hello_world pthread_hello_world.c -lpthread
./Pthread_Hello_world


OUTPUT:

In main: creating thread 0
In main: creating thread 1
Hello World! It's me, thread #0!
In main: creating thread 2
Hello World! It's me, thread #1!
In main: creating thread 3
Hello World! It's me, thread #2!
In main: creating thread 4
Hello World! It's me, thread #3!
Hello World! It's me, thread #4!
```

Threads inside the OS

- The key to viewing threads from the perspective of a modern operating system is to recognize that operating systems are partitioned into two distinct layers: the user-level partition (where applications are run) and the kernel-level partition (where system oriented activities occur).

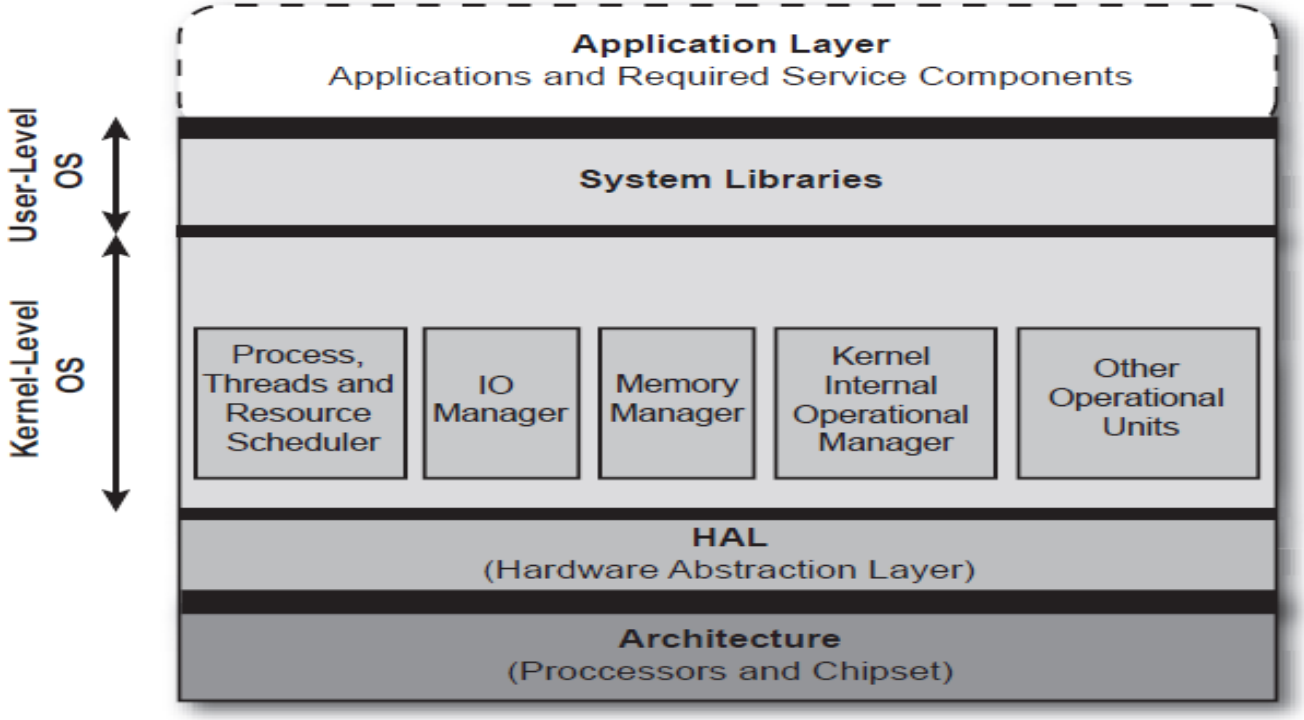
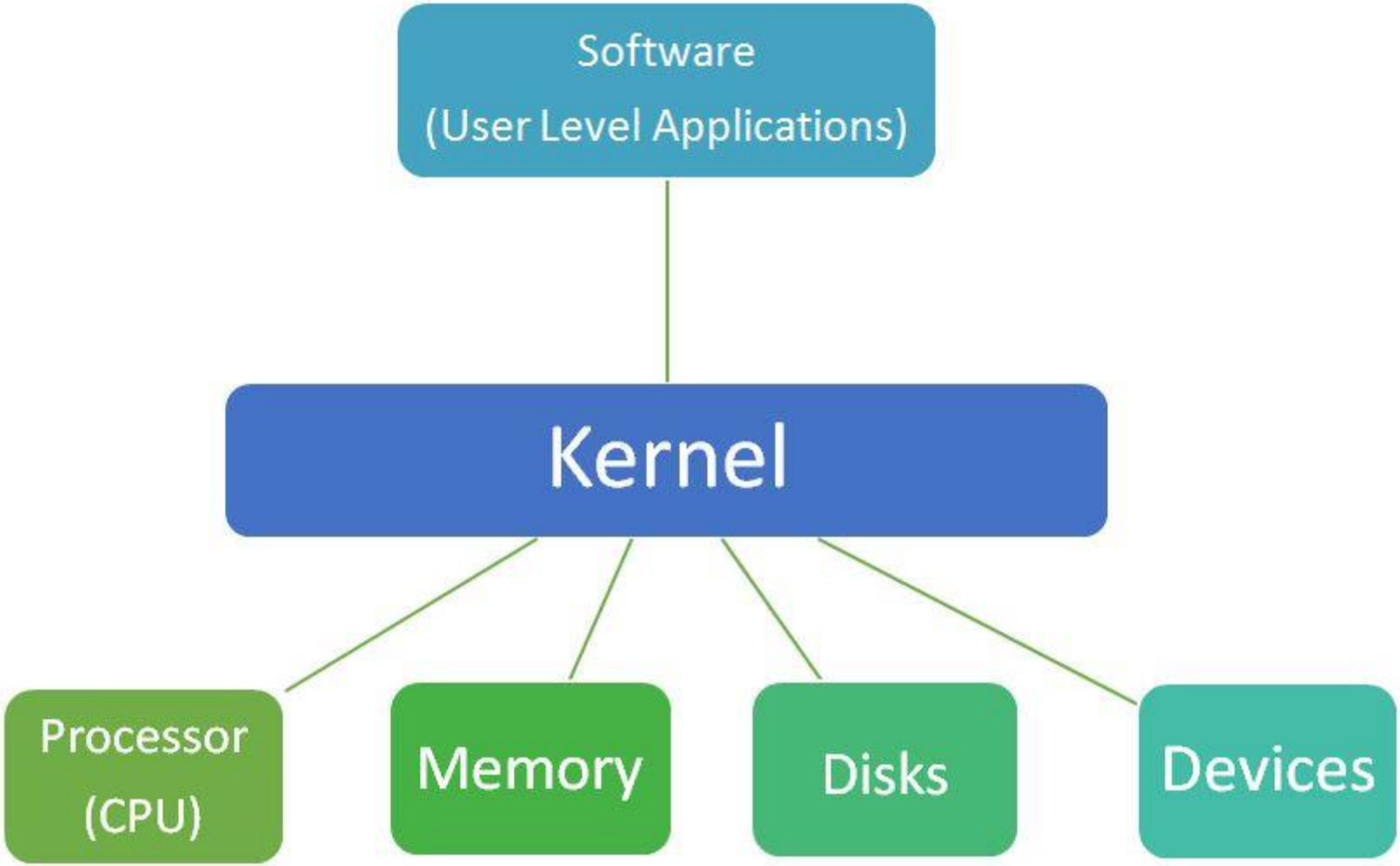
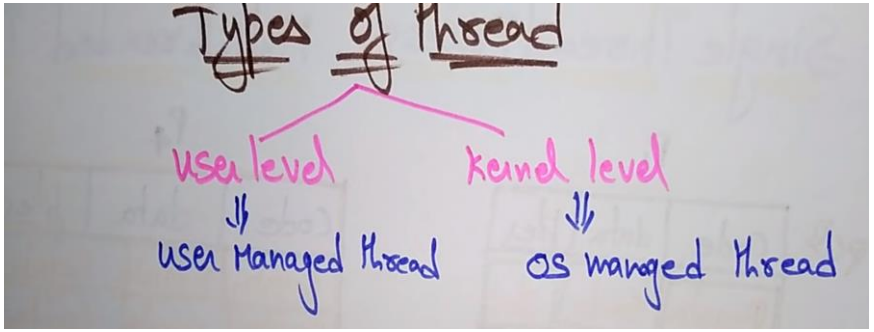


Figure 2.3 Different Layers of the Operating System


Software
(User Level Applications)
Kernel
Processor
(CPU)
Memory
Disks
Devices

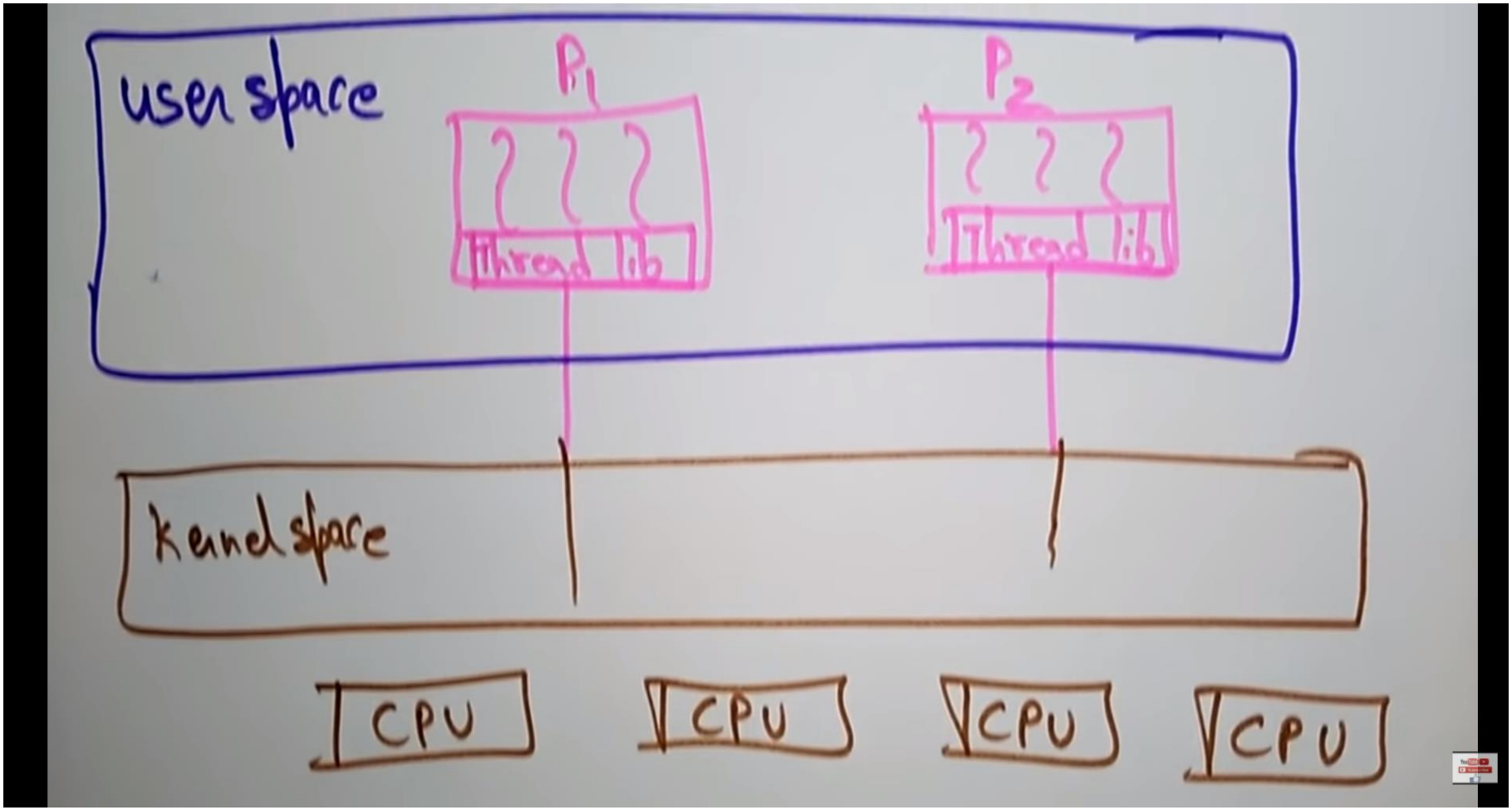
Contd….

- The kernel is the nucleus of the operating system and maintains tables to keep track of processes and threads.
- Threading libraries such as OpenMP and Pthreads (POSIX standard threads) use kernel-level threads.
- User-level threads, which are called *fibers* on the Windows platform, require the programmer to create the entire management infrastructure for the threads and to manually schedule their execution.
- Kernel-level threads provide better performance, and multiple kernel threads from the same process can execute on different processors or cores.



User-level Threads

User-level threads are mapped to kernel threads; and so, when they are executing, the processor knows them only as kernel-level threads.



Contd....

- Below figure shows the relationship between processors, processes, and threads in modern operating systems.
- A processor runs threads from one or more processes, each of which contains one or more threads.
- A program has one or more processes, each of which contains one or more threads, each of which is mapped to a processor by the scheduler in the operating system.

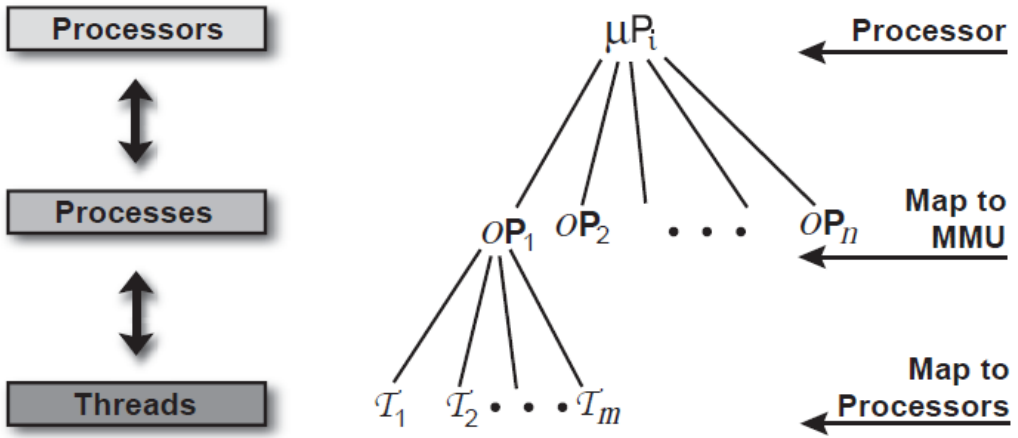
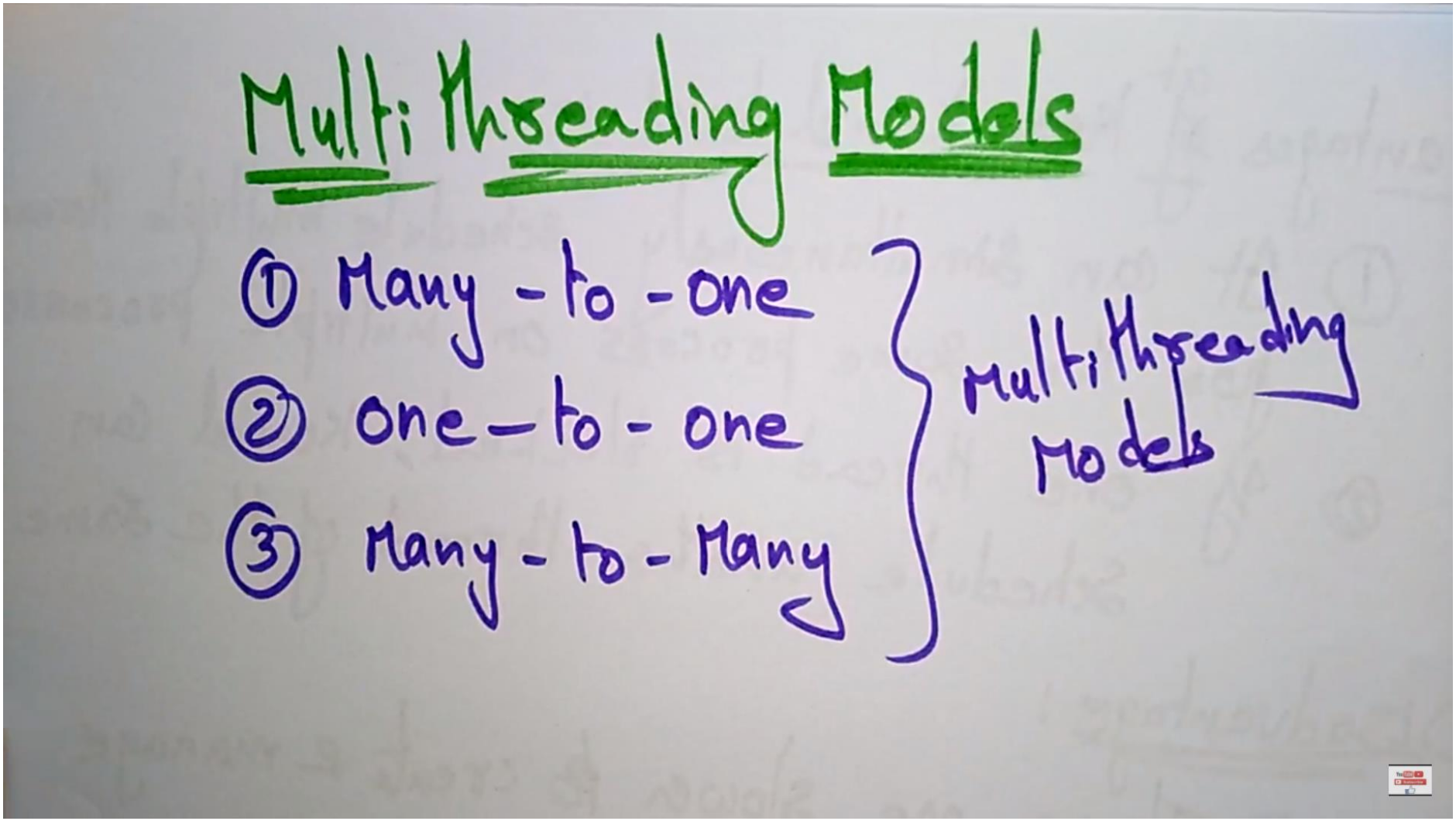


Figure 2.4 Relationships among Processors, Processes, and Threads

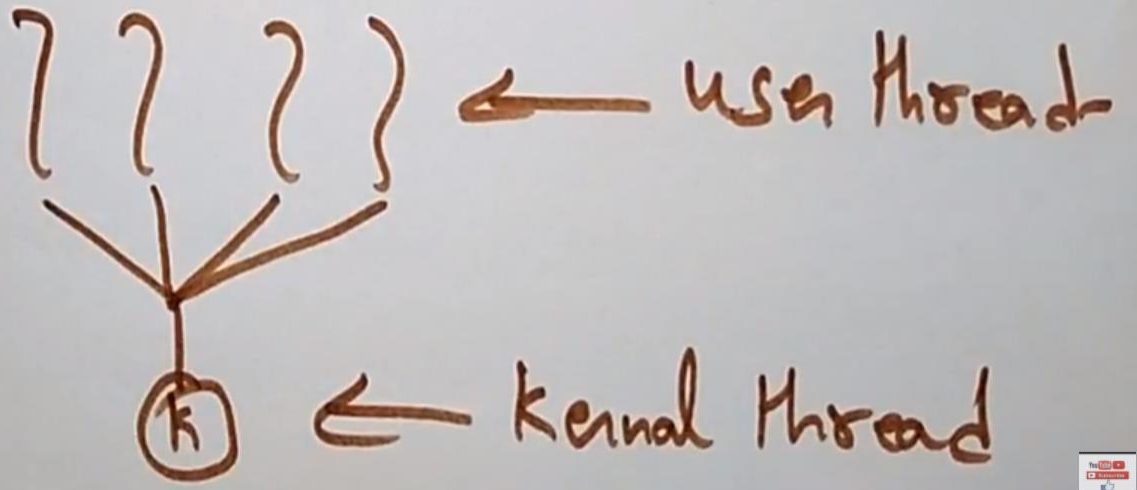
Various mapping models are used between threads and processors



Many-to-one

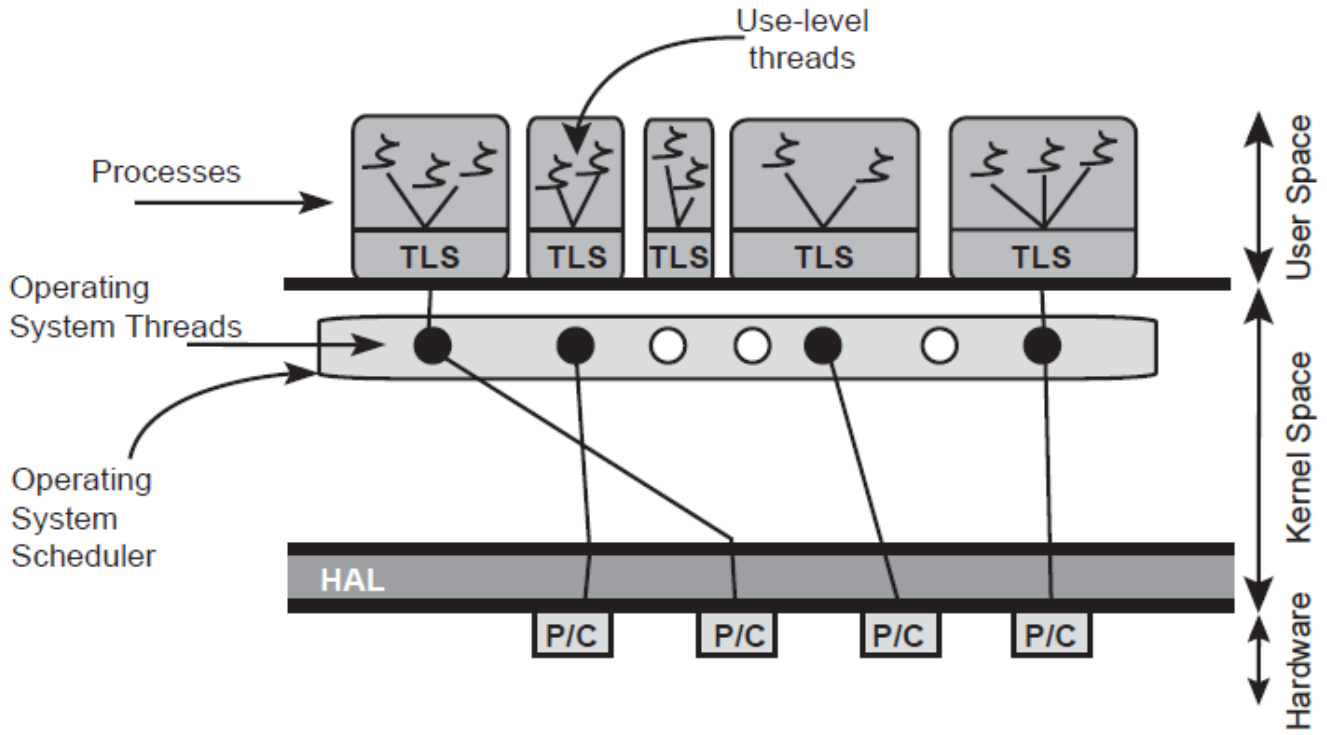
- Many user-level threads maps to single kernel thread

* only one thread can access the kernal at a time



Many to one (M:1)

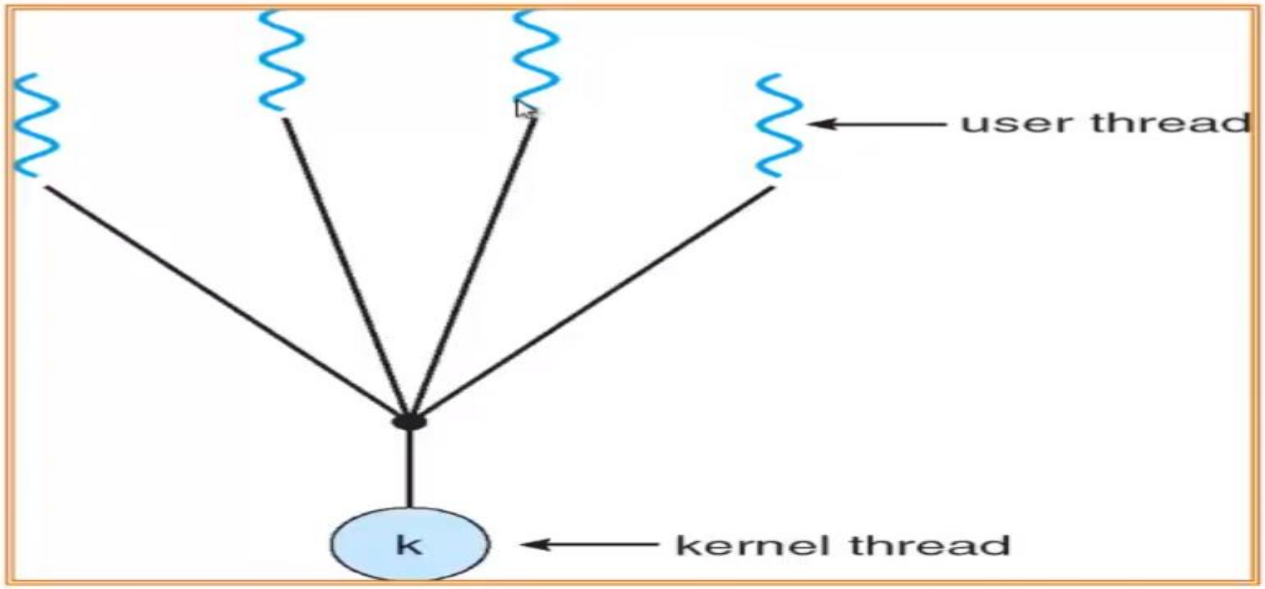
In the M:1 model, the library scheduler decides which thread gets the priority. This is called *cooperative multi-threading*.



(b) M:1 Mapping of Threads to Processors

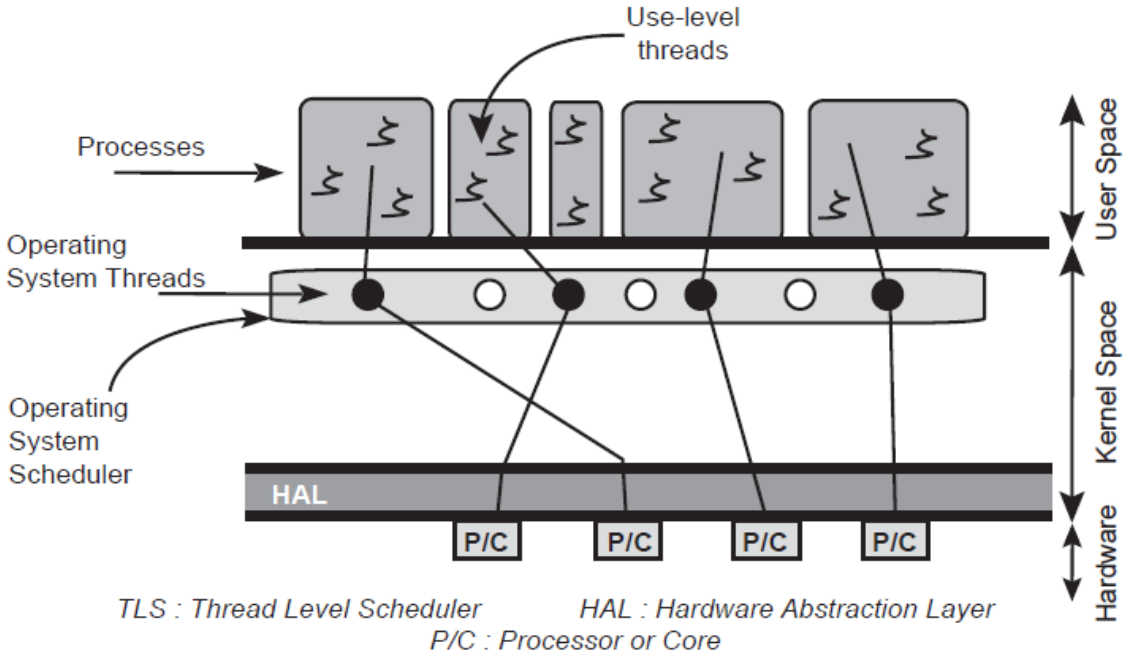
Many-to-one

- Many user-level threads are mapped to one kernel thread



One to one (1:1)

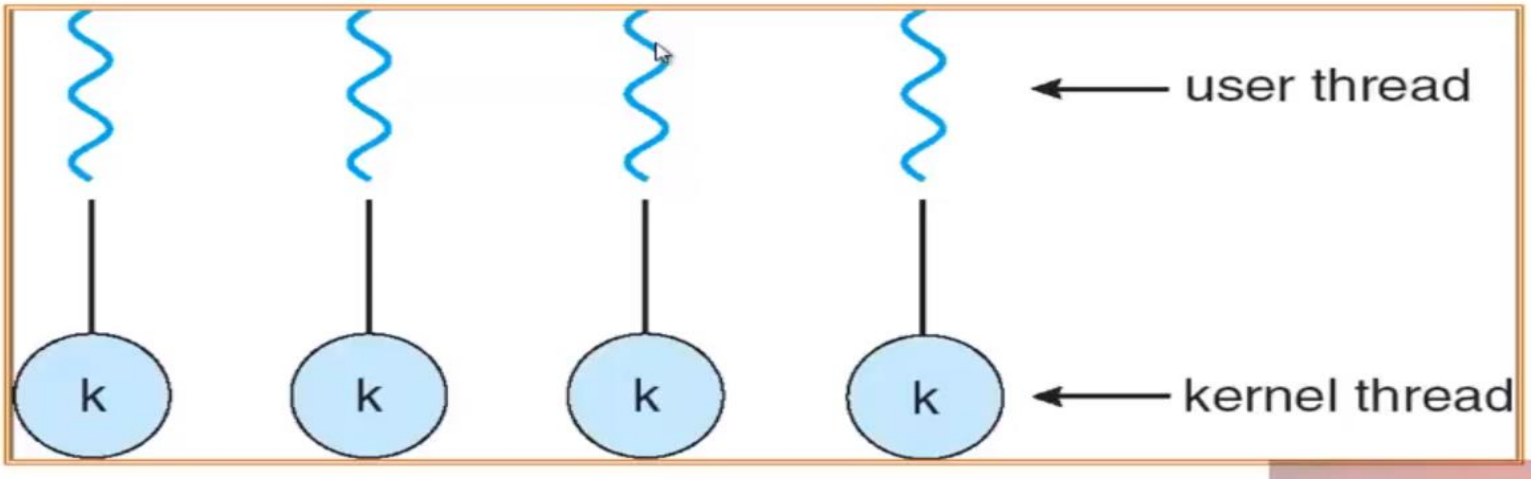
- The 1:1 model requires no thread-library scheduler overhead and the operating system handles the thread scheduling responsibility. This is also referred to as preemptive multi-threading.
- Linux, Windows 2000, and Windows XP use this preemptive multithreading model.



(a) 1:1 Mapping of Threads to Processors

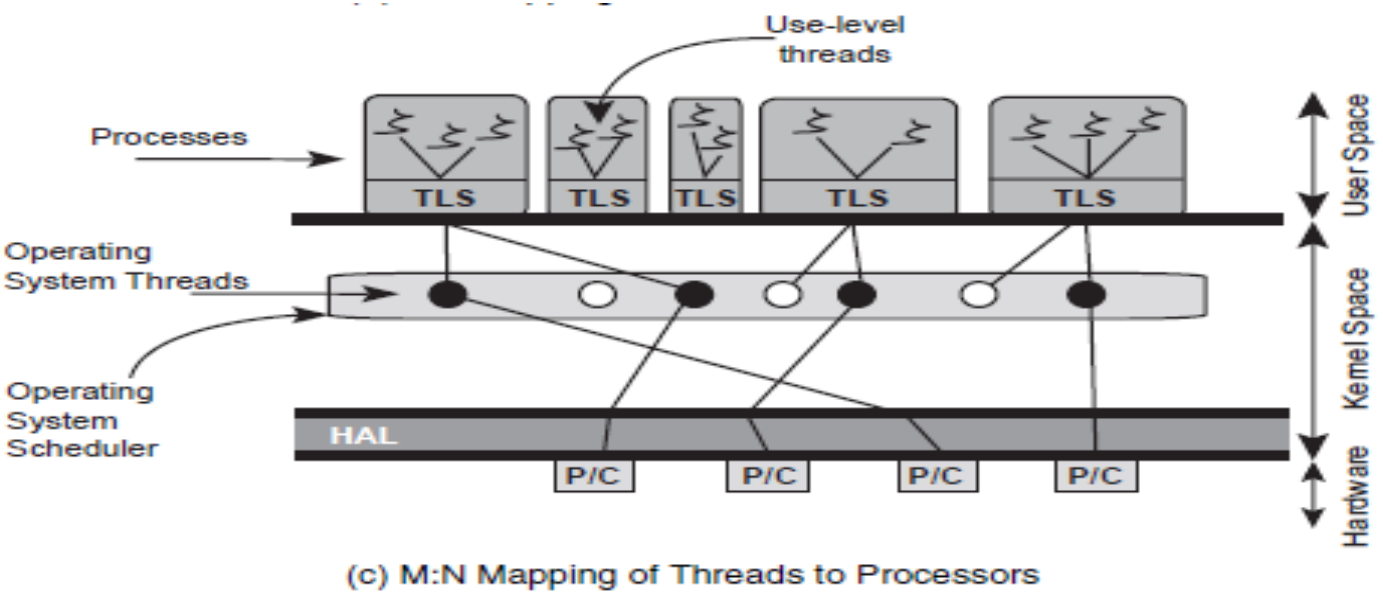
One-to-one

- Each user thread maps to one kernel thread
- Provides more concurrency, if one thread makes a blocking call another thread is allowed to run
- Creating user thread requires creating corresponding kernel thread.



M:N Mapping of Threads to Processors

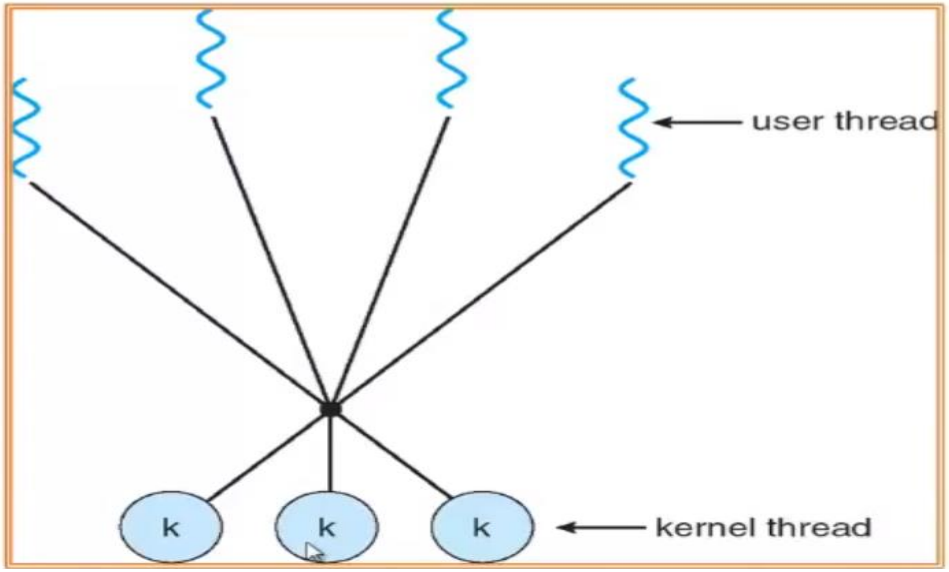
- In the case of M:N, the mapping is flexible.



(c) M:N Mapping of Threads to Processors

Many-to-many

- Multiplexes many user-level threads to a smaller or equal number of kernel threads
- When a thread performs a blocking system call, the kernel schedules another thread for execution.



Threads inside the Hardware

- The hardware executes the instructions from the software levels.
- Instructions of your application threads are mapped to resources and flow down through the intermediate components-the operating system, runtime environment, and to the hardware.
- Threading on hardware once required multiple CPUs to implement parallelism: Multi-core CPUs.
- The CPU might have only one execution engine or core but share the pipeline and other hardware resources among the executing threads: SMT- *concurrent.*

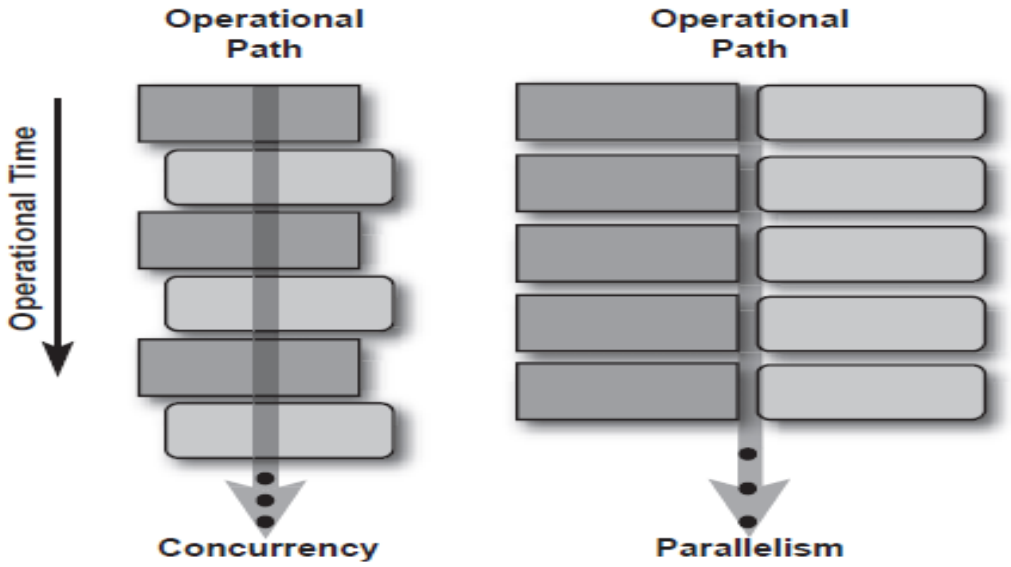


Figure 2.6 Concurrency versus Parallelism

What Happens When a Thread Is Created

- As discussed earlier, Every process has at least one thread. This *initial thread* is created as part of the process initialization.
- There also can be more than one thread in a process; and each of those threads operates independently, even though they share the same address space and certain resources.
- In addition, each thread needs to have its own stack space. These stacks are usually managed by the operating system.
- Once created, a thread is always in one of four states: ready, running, waiting (blocked), or terminated.

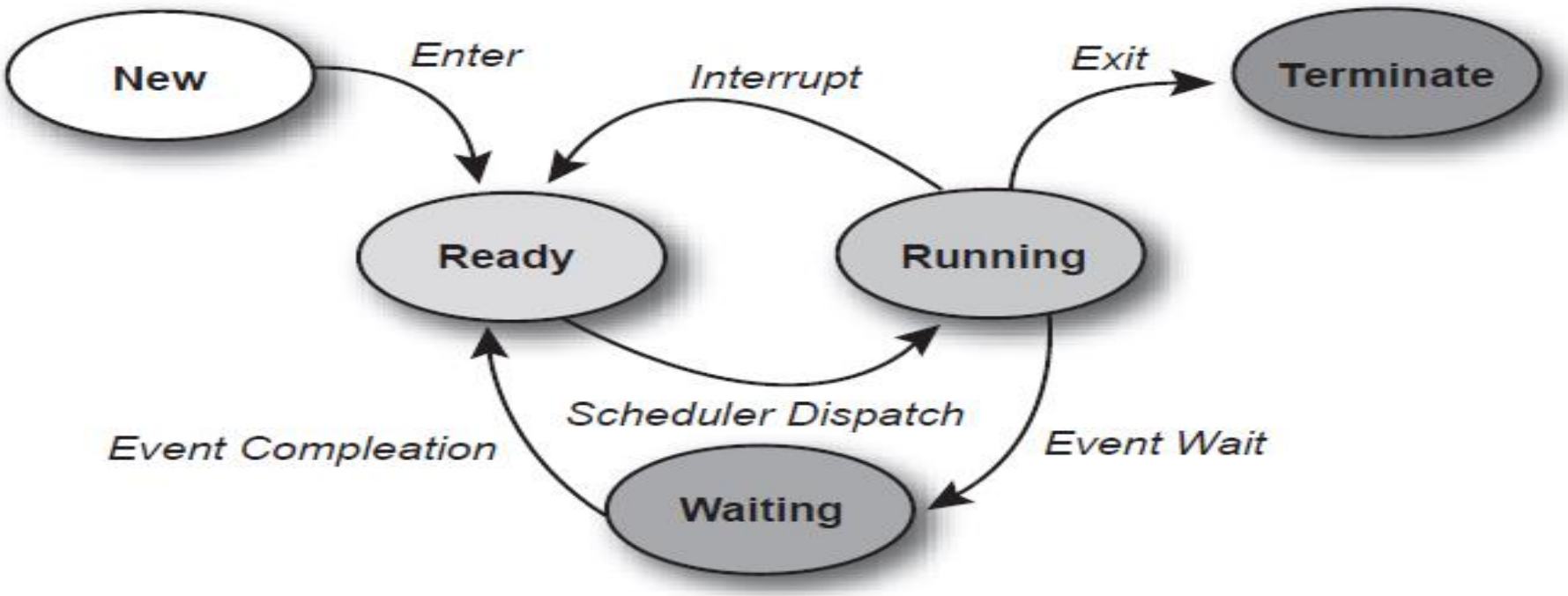


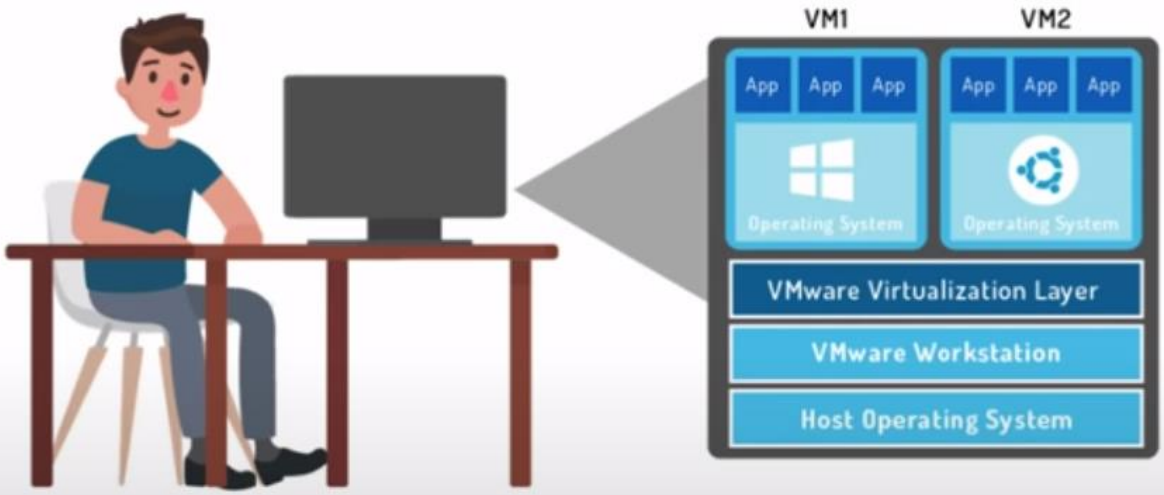
Figure 2.8 State Diagram for a Thread

Virtual Environment: VMs and Platforms

- One of the most important trends in computing today is virtualization.
- Virtualization is the process of using computing resources to create the appearance of a different set of resources.
- System virtualization creates the appearance of a different kind of virtual machine, in which there exists a complete and independent instance of the operating system.
- The virtualization layer that sits between the host system and these VMs is called the *virtual machine monitor* (VMM). The VMM is also known as the hypervisor.

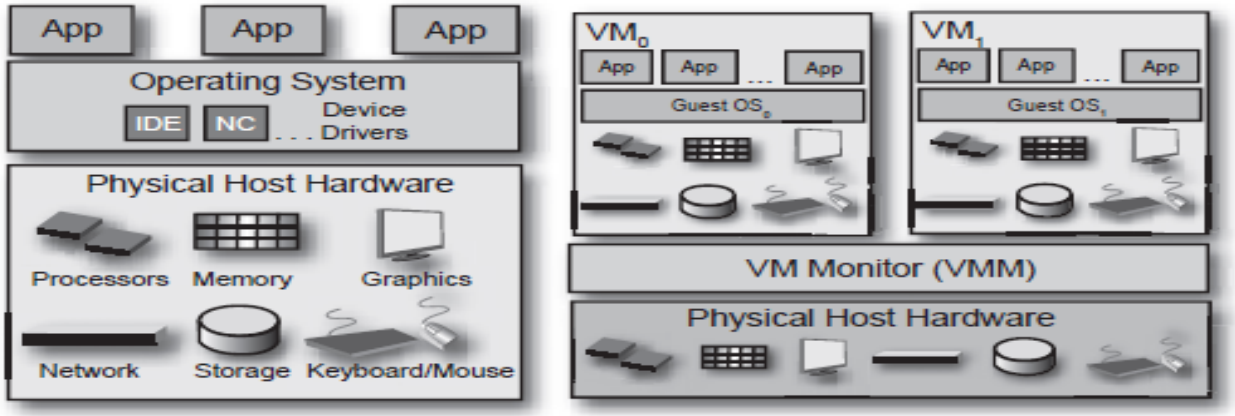
What is Virtualization?

Many virtual machines can be created on a host



System Virtualization

- System virtualization creates a different type of virtual machine.
- A VMM delivers the necessary virtualization of the underlying platform such that the operating system in each VM runs under the illusion that it owns the entire hardware platform.
- When an application running in a VM creates a thread, the thread creation and subsequent scheduling is all handled by the guest operating system. The virtual processor executes the instructions of the thread.



2.9 Comparison of Systems without and with a VMM

Fundamental Concepts of Parallel Programming

- As discussed in previous lectures, parallel programming uses threads to enable multiple operations to proceed simultaneously.
- The entire concept of parallel programming centers on the design, development, and deployment of threads within an application and the coordination between threads and their respective operations.
- This chapter examines how to break up programming tasks into chunks that are suitable for threading.

Contd....

- To move from the linear model to a parallel programming model, designers must rethink the idea of process flow.
- Rather than being constrained by a sequential execution sequence, programmers should identify those activities that can be executed in parallel.
- To do so, designers must see their programs as a set of tasks with dependencies between them.
- Breaking programs down into these individual tasks and identifying dependencies is known as decomposition.

Contd….

- A problem may be decomposed in several ways: by task, by data, or by data flow.
- The below table summarizes these forms of decomposition.

Table 3.1 Summary of the Major Forms of Decomposition

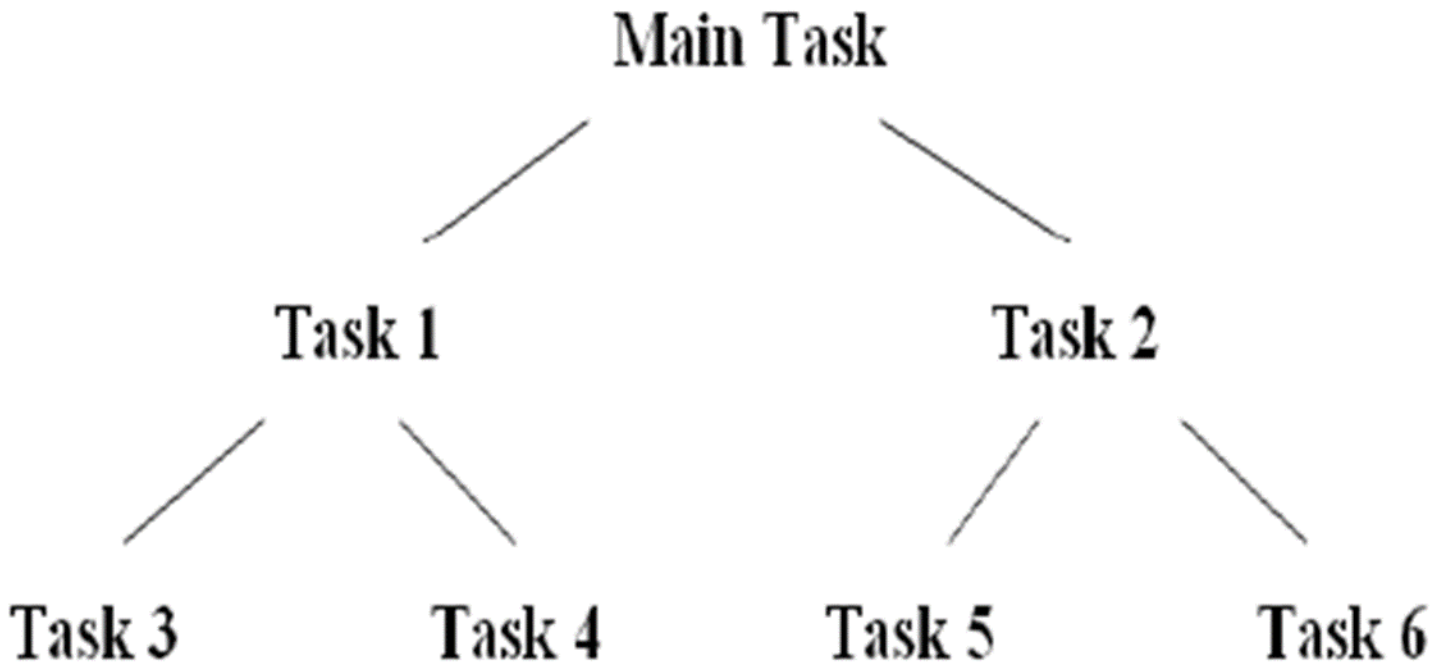
Decomposition	Design	Comments
Task	Different activities assigned to different threads	Common in GUI apps
Data	Multiple threads performing the same operation but on different blocks of data	Common in audio processing, imaging, and in scientific programming
Data Flow	One thread's output is the input to a second thread	Special care is needed to eliminate startup and shutdown latencies

Task Decomposition

- Decomposing a program by the functions that it performs is called task decomposition. It is one of the simplest ways to achieve parallel execution.
- Using this approach, individual tasks are catalogued. If two of them can run concurrently, they are scheduled to do so by the developer.
- As a example consider, gardening, task decomposition would suggest that gardeners be assigned tasks based on the nature of the activity.
- If two gardeners arrived at a client's home, one might mow the lawn while the other weeded.
- Mowing and weeding are separate functions broken out as such.
- To accomplish them, the gardeners would make sure to have some coordination between them, so that the weeder is not sitting in the middle of a lawn that needs to be mowed.

Contd....

In programming terms,

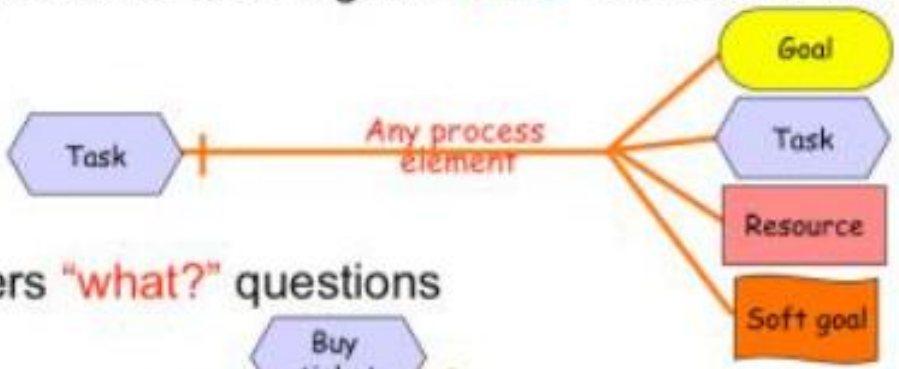


Contd....

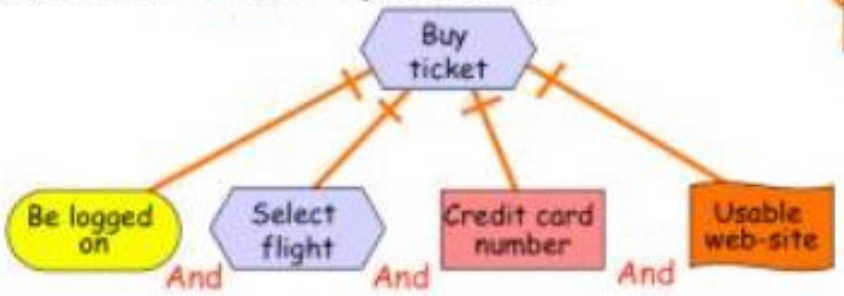
Task Decomposition Links

Decompose task into sub-components of all types

- All sub-components need to be "completed" for a task to be performed, so logical "AND" between them



- Answers "what?" questions



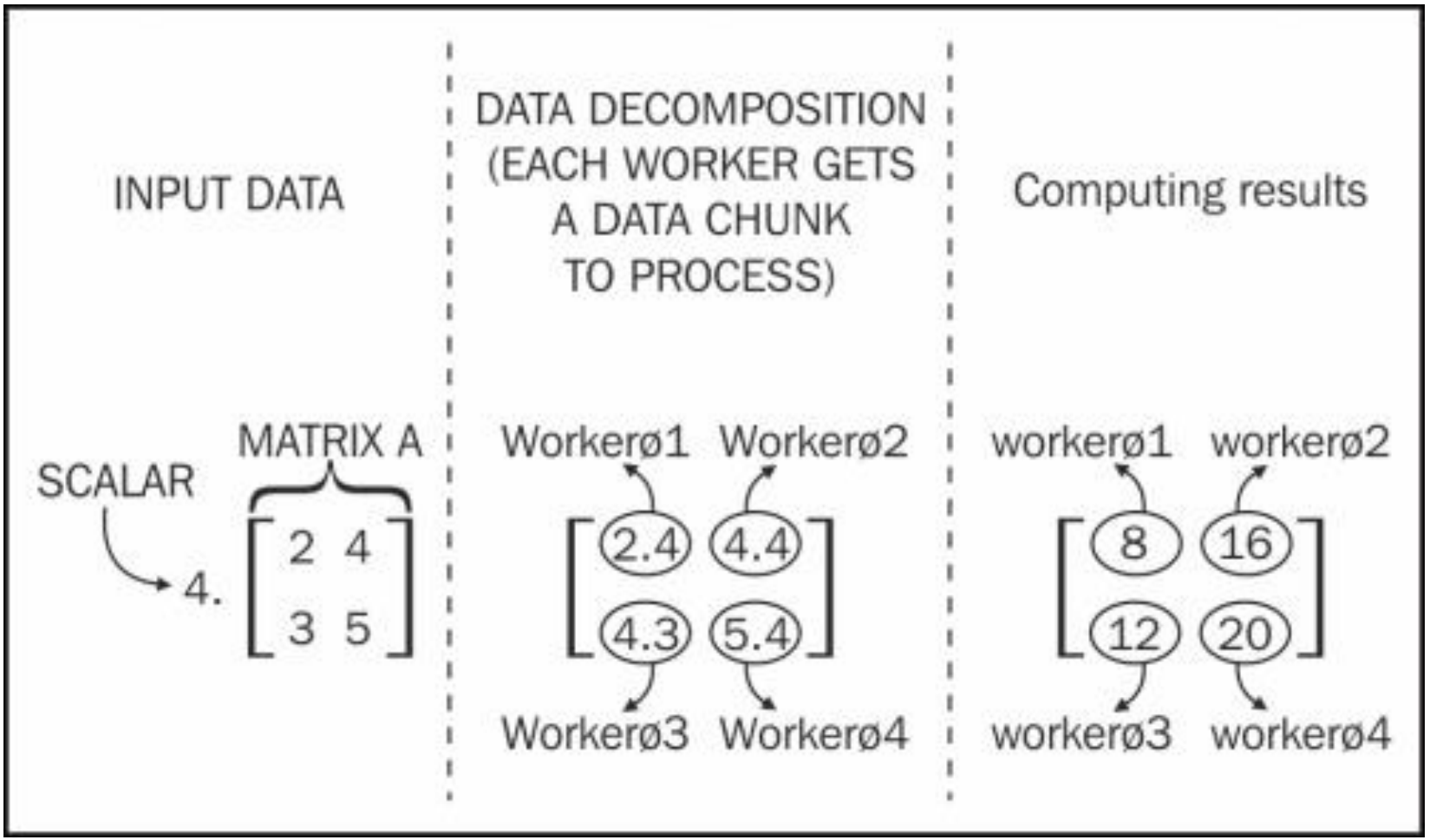
Data Decomposition

- Data decomposition, also known as data-level parallelism, breaks down tasks by the data they work on rather than by the nature of the task.
- Programs that are broken down via data decomposition generally have many threads performing the same work, just on different data items.
- For example, consider recalculating the values in a large spreadsheet. Rather than have one thread perform all the calculations, data decomposition would suggest having two threads, each performing half the calculations, or n threads performing 1/nth the work.

Contd….

- If the gardeners used the principle of data decomposition to divide their work, they would both mow half the property and then both weed half the flower beds.
- As in computing, determining which form of decomposition is more effective depends a lot on the constraints of the system.
- For example, if the area to mow is so small that it does not need two mowers, that task would be better done by just one gardener- that is, task decomposition is the best choice

Contd....



A key aim is to solve problems faster/Performance

Data Flow Decomposition

- Many times, when decomposing a problem, the critical issue isn't what tasks should do the work, but how the data flows between the different tasks.
- In these cases, data flow decomposition breaks up a problem by how data flows between tasks.
- The producer/consumer problem is a well known example of how data flow impacts a programs ability to execute in parallel.
- Here, the output of one task, the producer, becomes the input to another, the consumer. The two tasks are performed by different threads, and the second one, the consumer, cannot start until the producer finishes some portion of its work.

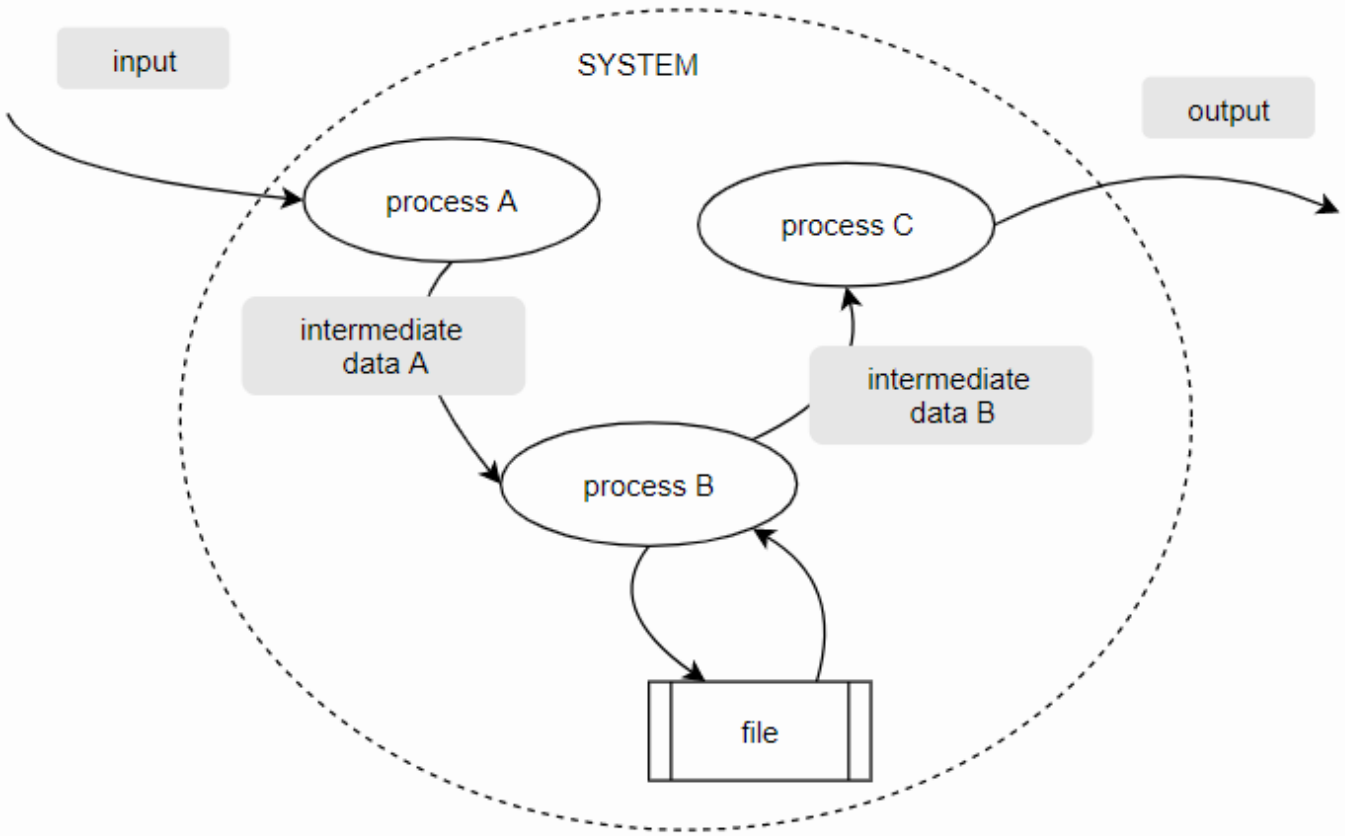
Contd….

- Using the gardening example, one gardener prepares the tools—that is, he puts gas in the mower, cleans the shears, and other similar tasks— for both gardeners to use.
- No gardening can occur until this step is mostly finished, at which point the true gardening work can begin.
- The delay caused by the first task creates a pause for the second task, after which both tasks can continue in parallel.

Contd....

- In common programming tasks, the producer/consumer problem occurs in several typical scenarios.
- For example, programs that must rely on the reading of a file fit this scenario.
- The results of the file I/O become the input to the next step, which might be threaded. However, that step cannot begin until the reading is either complete or has progressed sufficiently.

Contd....




```
for (i=1; i<100; i++)
    a[i] = a[i-1] + 100;
```

"Tightly coupled" problems require lots of interaction between their parallel tasks

Contd....

The producer/consumer(Data Flow Decomposition) has several interesting dimensions:

- The dependence created between consumer and producer can cause significant delays if this model is not implemented correctly.
- If the consumer is finishing up while the producer is completely done, one thread remains idle while other threads are busy working away.
- This issue violates an important objective of parallel processing, which is to balance loads so that all available threads are kept busy.
- A performance-sensitive design must aim to avoid situations of threads are in idle while waiting for related threads.

Implications of Different Decompositions

- Different decompositions provide different benefits.
- The most common reason for threading an application is performance, meanwhile, the choice of decompositions is more difficult.
- In many instances, the choice is dictated by the problem domain.
- In some cases, the answer comes only through careful analysis of the constituent activities.
- Ultimately, you determine the right answer for your application's use of parallel programming by careful planning, timing, evaluation and testing.

Challenges You'll Face

- The use of threads enables you to improve performance significantly by allowing two or more activities to occur simultaneously.
- However, developers cannot fail to recognize that threads add a measure of complexity that requires thoughtful consideration to navigate correctly.
- This complexity arises from the inherent fact that more than one activity is occurring in the program.

Contd....

Managing simultaneous activities and their possible interaction leads you to confronting four types of problems:

1. *Synchronization* is the process by which two or more threads coordinate their activities. For example, one thread waits for another to finish a task before continuing.
2. *Communication* refers to the bandwidth and latency issues associated with exchanging data between threads.
3. *Load balancing* refers to the distribution of work across multiple threads so that they all perform roughly the same amount of work.
4. *Scalability* is the challenge of making efficient use of a larger number of threads when software is run on more-capable systems. For example, if a program is written to make good use of four processor cores, will it scale properly when run on a system with eight processor cores?

Each of these issues must be handled carefully to maximize application performance. Subsequent chapters describe many aspects of these problems and how best to address them on multi-core systems.

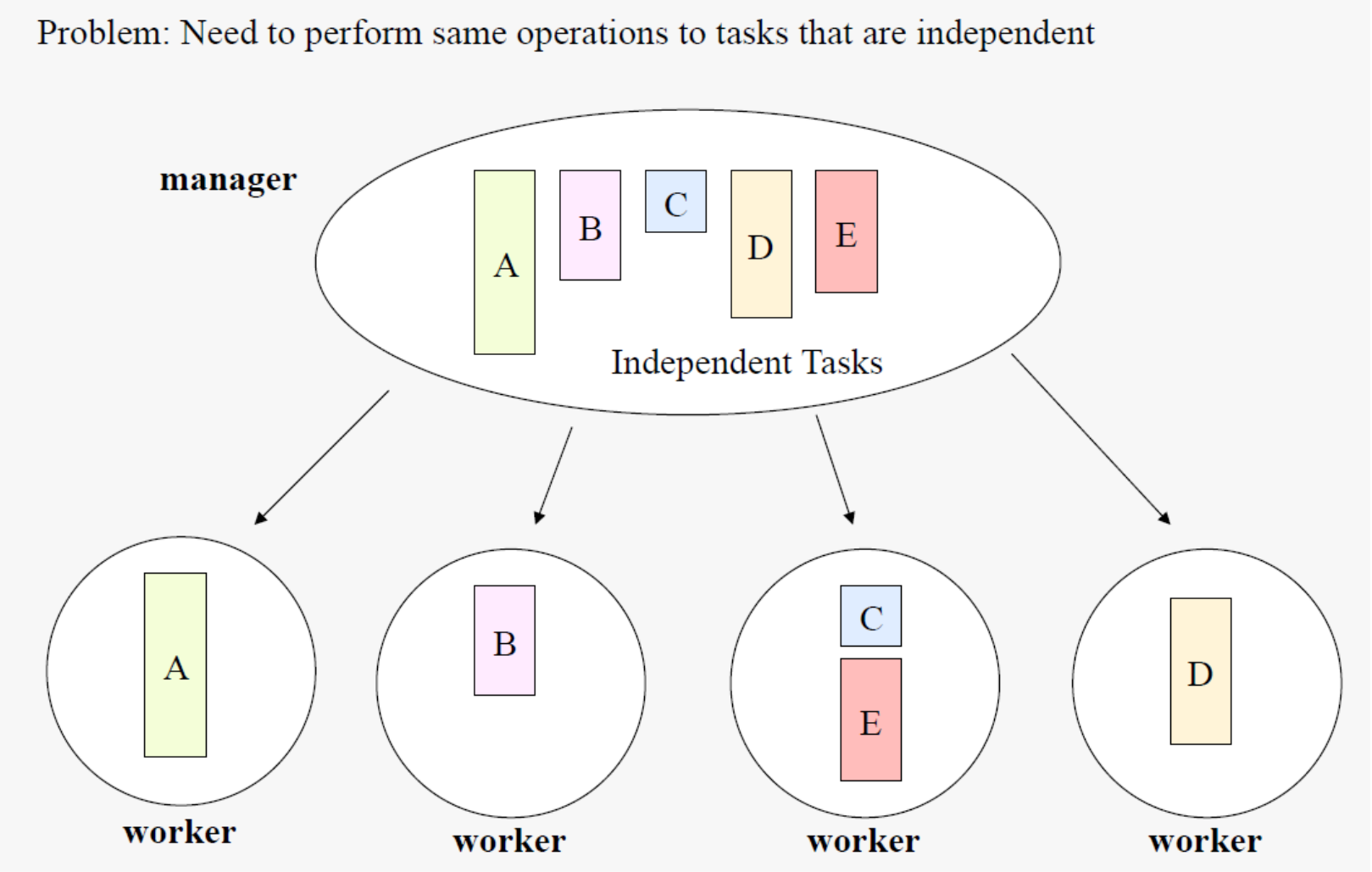
Parallel Programming Patterns

- Parallel programming patterns are design patterns to logically design applications, through Parallel programming/Solve parallel programming problems.
- Need a "cookbook" that will guide the programmers systematically to achieve peak parallel performance.
- Provide common vocabulary to the programming comunity.
- A few of the more common parallel programming patterns and their relationship to the aforementioned decompositions are shown in below table.

Table 3.2 Common Parallel Programming Patterns

Pattern	Decomposition
Task-level parallelism	Task
Divide and Conquer	Task/Data
Geometric Decomposition	Data
Pipeline	Data Flow
Wavefront	Data Flow

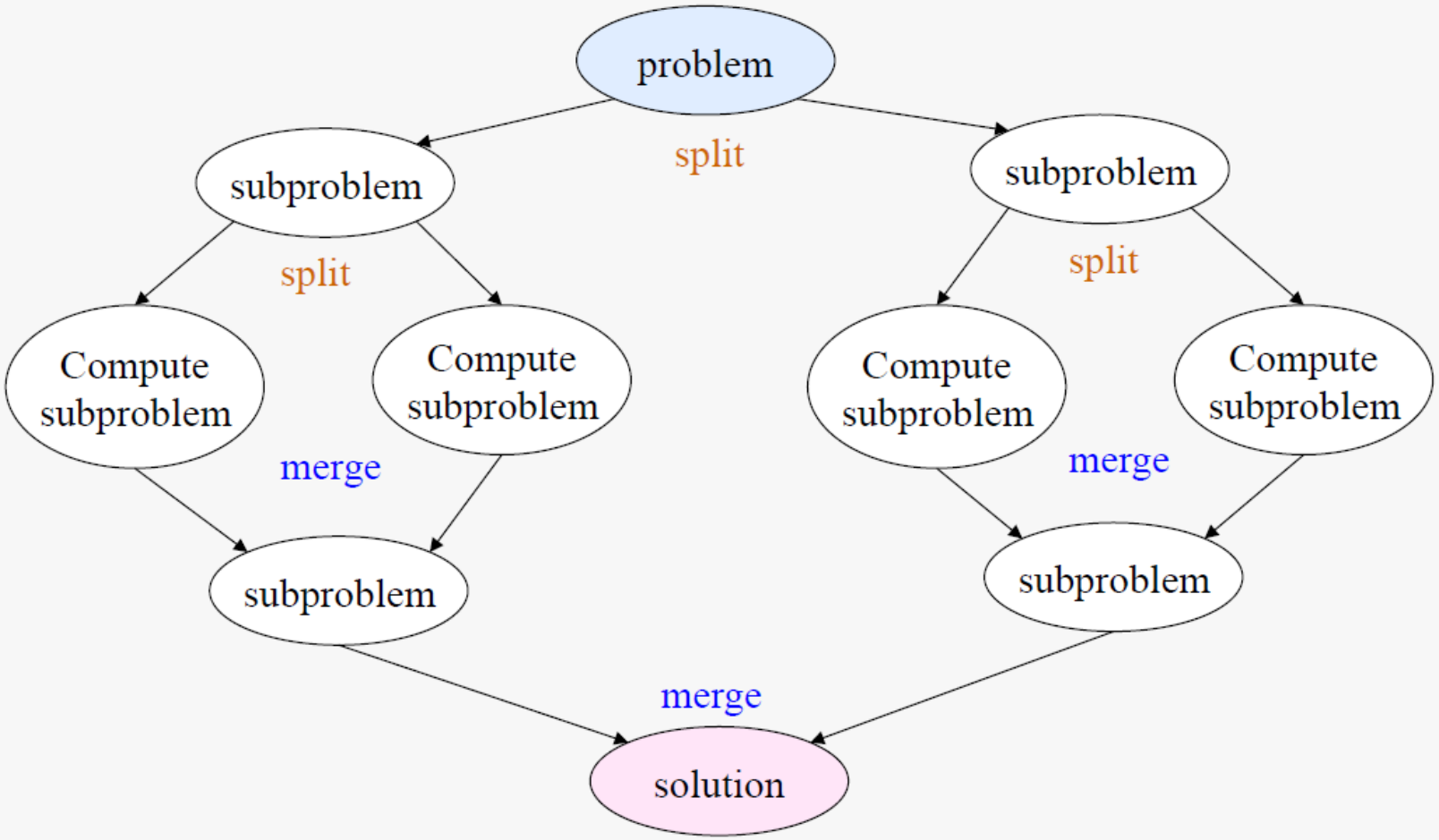
Task-level parallelism



In this pattern, the problem is decomposed into a set of tasks that operate independently. Problems that fit into this pattern include the so-called *embarrassingly parallel* problems, those where there are no dependencies between threads.

Divide and Conquer

A problem is structured to be solved in sub-problems independently, and merging them later.



Split level needs to be adjusted appropriately.

- *Geometric Decomposition Pattern*
- *Pipeline Pattern*
- *Wavefront Pattern*

A Motivating Problem: Error Diffusion

- To see how you might apply the discussed methods to a practical computing problem, consider the **error diffusion** algorithm (multi-level image into a binary image) that is used in many computer graphics and image processing programs.
- *Error diffusion* is a technique for displaying continuous-tone digital images on devices that have limited color (tone) range. Originally proposed by Floyd and Steinberg (Floyd 1975).
- The problem seems to break down into a **data-flow decomposition** and follow **wavefront pattern**.

The key points to keep in mind when developing solutions for parallel computing architectures

- Decompositions fall into one of three categories: task, data, and data flow.
- Task-level parallelism partitions the work between threads based on tasks.
- Data decomposition breaks down tasks based on the data that the threads work on.
- Data flow decomposition breaks down the problem in terms of how data flows between the tasks.
- Most parallel programming problems fall into one of several well known patterns.
- The constraints of synchronization, communication, load balancing, and scalability must be dealt with to get the most benefit out of a parallel program.

Many problems that appear to be serial may, through a simple transformation, be adapted to a parallel implementation.

Threading and Parallel Programming Constructs

- Here we describe the theory and practice of the principal parallel programming constructs that focus on threading and begins with the fundamental concepts of synchronization, critical section, and deadlock.

- Basic parallel programming constructs
- Topics :
  - Synchronization & primitives used
  - Critical Section
  - Deadlock
  - Use of messages
  - Concepts based on flow control

Synchronization

- In simple terms, synchronization is used to coordinate thread execution and manage shared data.
- Two types of synchronization operations are widely used: mutual exclusion and condition synchronization.
- In the case of ***mutual exclusion***, one thread blocks a critical section—a section of code that contains shared data—and one or more threads wait to get their turn to enter into the section.
- This helps when two or more threads share the same memory space and run simultaneously.
- ***Condition synchronization***, on the other hand, blocks a thread until the system state specifies some specific conditions. The condition synchronization allows a thread to wait until a specific condition is reached.

Contd....

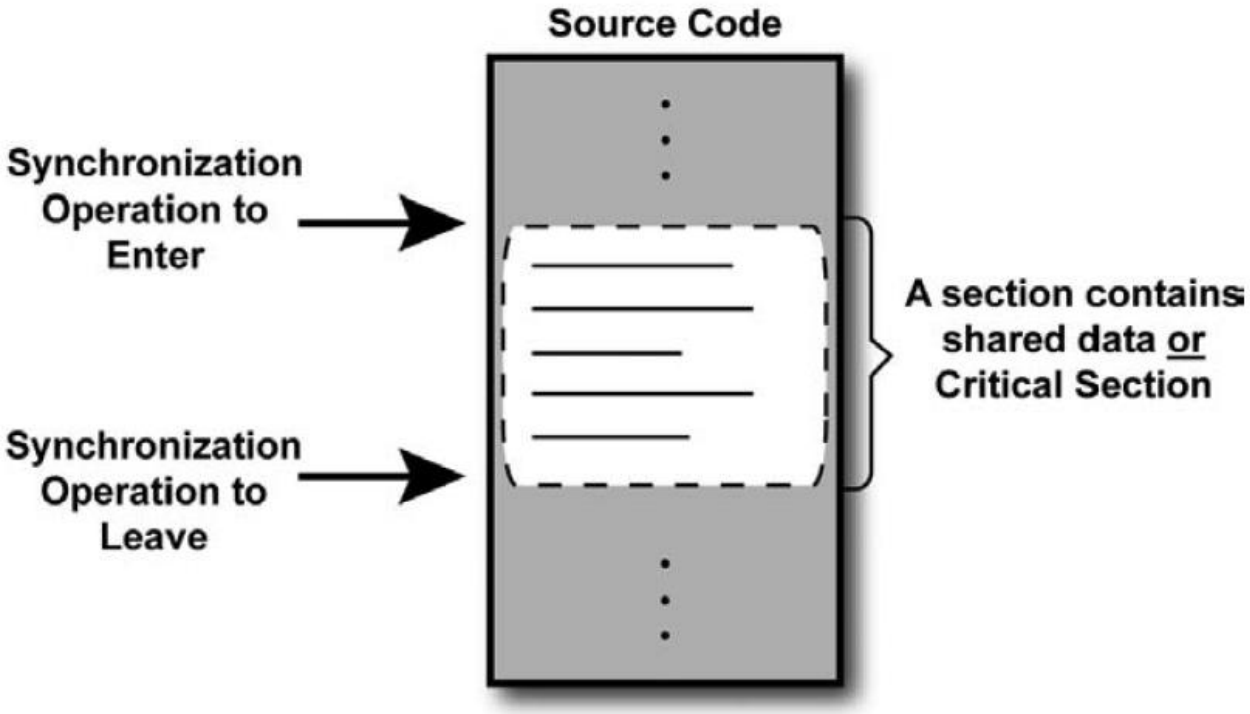


Figure 4.1 Generic Representation of Synchronization Block inside Source Code

Contd....

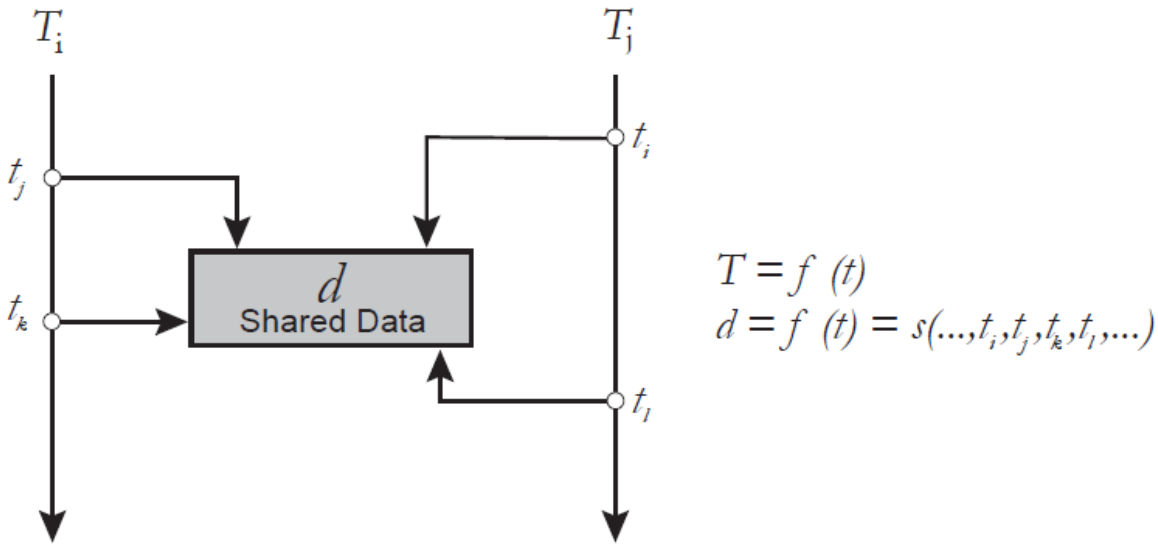


Figure 4.2 Shared Data Synchronization, Where Data *d* Is Protected by a Synchronization Operation

The scope of synchronization is broad. Proper synchronization orders the updates to data and provides an expected outcome. In Figure 4.2, shared data *d* can get access by threads Ti and Tj at time ti, tj, tk, tl, where ti ≠ tj ≠ tk ≠ tl and a proper synchronization maintains the order to update *d* at these instances and considers the state of *d* as a synchronization function of time. This synchronization function, *s*, represents the behavior of a synchronized construct with respect to the execution time of a thread.

Synchronization operations in an actual multi-threaded implementation

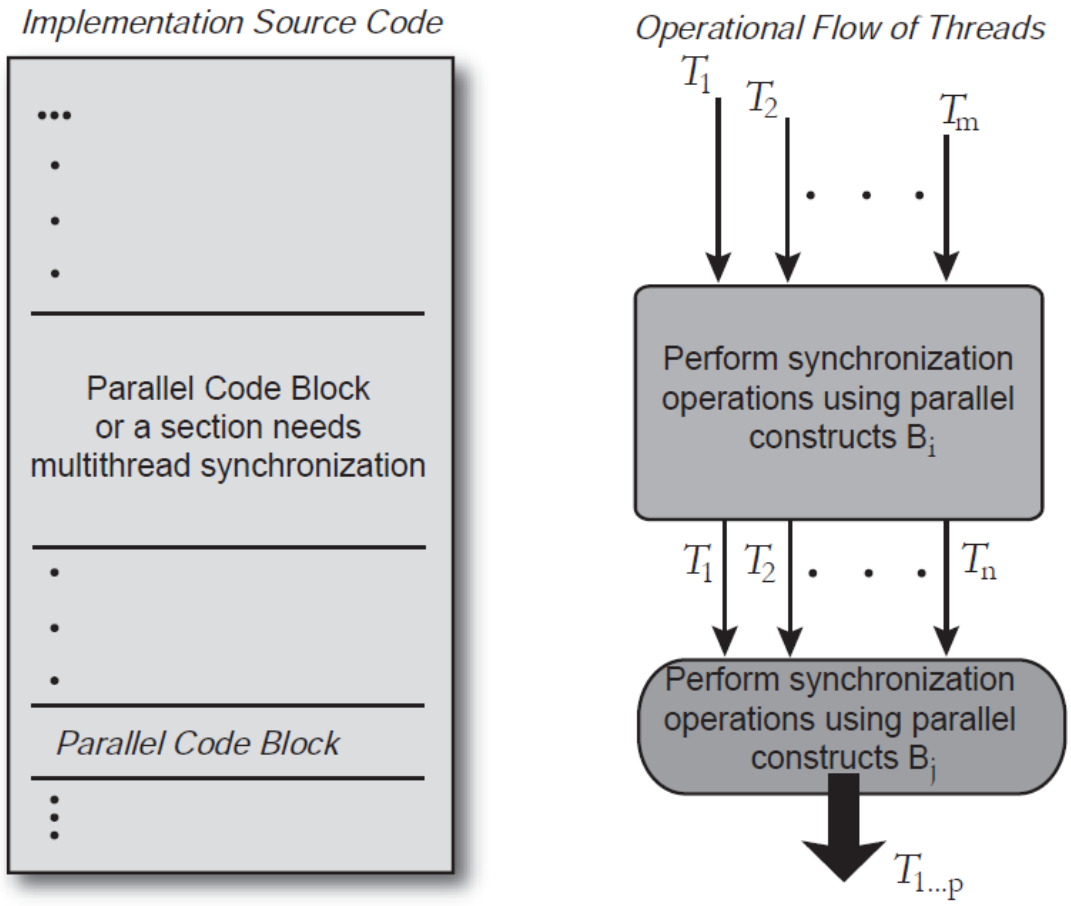
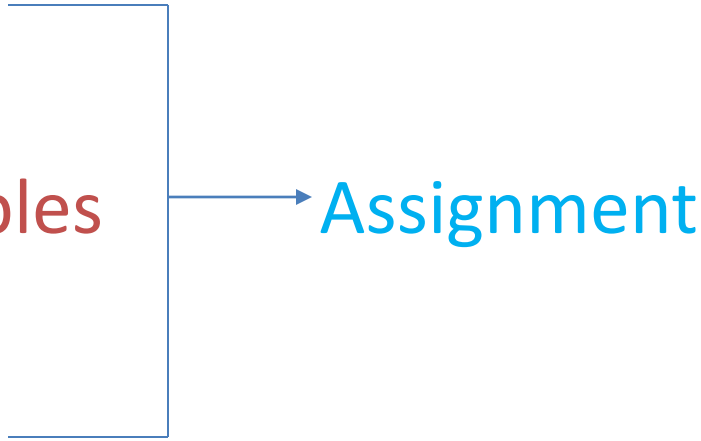


Figure 4.3 Operational Flow of Threads for an Application

Synchronization Primitives

- Synchronization is typically performed by different types of primitives:

1. Semaphores
2. Locks
3. Condition variables
4. Fence
5. Barrier



- The use of these primitives depends on the application requirements.

Critical Sections

- A section of a code block called a *critical section* is where shared dependency variables reside and those shared variables have dependency among multiple threads.
- Different synchronization primitives are used to keep critical sections safe.
- With the use of proper synchronization techniques, only one thread is allowed access to a critical section at any one instance.
- The major challenge of threaded programming is to implement critical sections in such a way that multiple threads perform mutually exclusive operations for critical sections and do not use critical sections simultaneously.

Contd....

- Minimize the size of critical sections when practical.
- Each critical section has an entry and an exit point.

```
<Critical Section Entry,
 to keep other threads in waiting status>
 ...
Critical Section
 ...
<Critical Section Exit,
 allow other threads to enter critical section>
```

Figure 4.4 Implement Critical Section in Source Code

Deadlock

- Deadlock occurs whenever a thread is blocked waiting on a resource of another thread that will never become available.
- According to the circumstances, different deadlocks can occur:

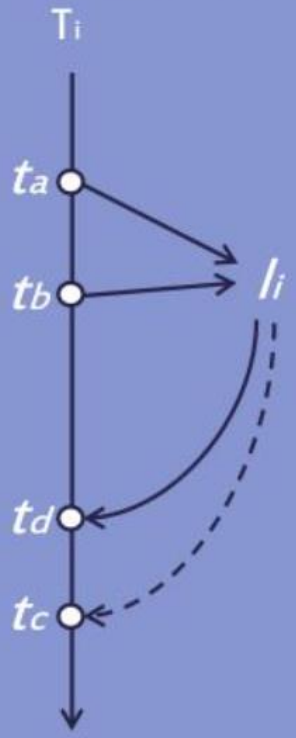
1. self-deadlock,
2. recursive deadlock,
3. lock-ordering deadlock.

Contd....

The self-deadlock is the instance or condition when a thread, T_i, wants to acquire a lock that is already owned by thread T_i. In Figure 4.5 (a), at time t_a thread T_i owns lock l_i, where l_i is going to get released at t_c. However, there is a call at t_b from T_i, which requires l_i. The release time of l_i is t_d, where t_d can be either before or after t_c. In this scenario, thread T_i is in self-deadlock condition at t_b. When the wakeup path of thread T_i, resides in another thread, T_j, that condition is referred to as *recursive deadlock*, as shown in Figure 4.5 (b). Figure 4.5 (c) illustrates a lock-ordering thread, where thread T_i locks resource r_j and waits for resource r_i, which is being locked by thread T_j. Also, thread T_j locks resource r_i and waits for resource r_j, which is being locked by thread T_i. Here, both threads T_i and T_j are in deadlock at t_d, and w is the *wait-function* for a lock.

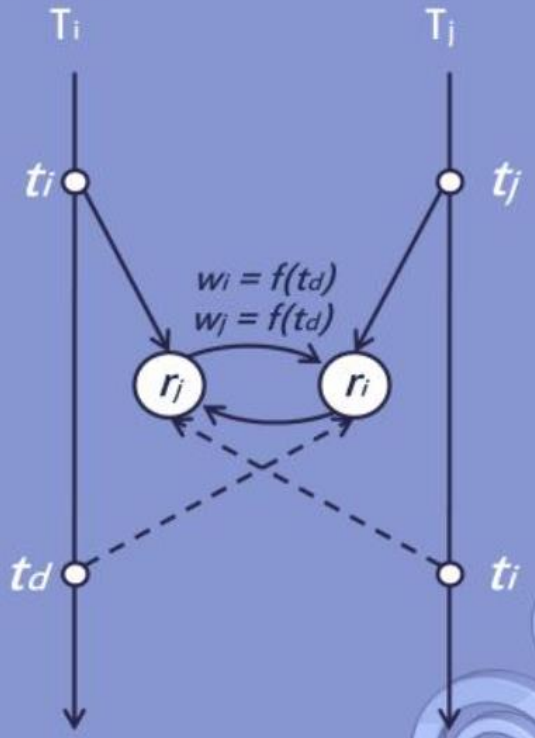
Deadlock

Thread is blocked waiting on a resource of another thread that will never become available



Self-deadlock

Recursive Deadlock



Lock-ordering Deadlock

Contd....

- Avoiding deadlock is one of the challenges of multi-threaded programming.
- There must not be any possibility of deadlock in an application.
- One recommendation is to use the appropriate number of locks when implementing synchronization.

Messages

- The *message* is a special method of communication to transfer information or a signal from one domain to another.
- For multi-threading environments, the domain is referred to as the *boundary* of a thread.
- In general, the conceptual representations of messages get associated with processes rather than threads.
- From a message-sharing perspective, messages get shared using an intra-process, inter-process, or process-process approach.

Contd....

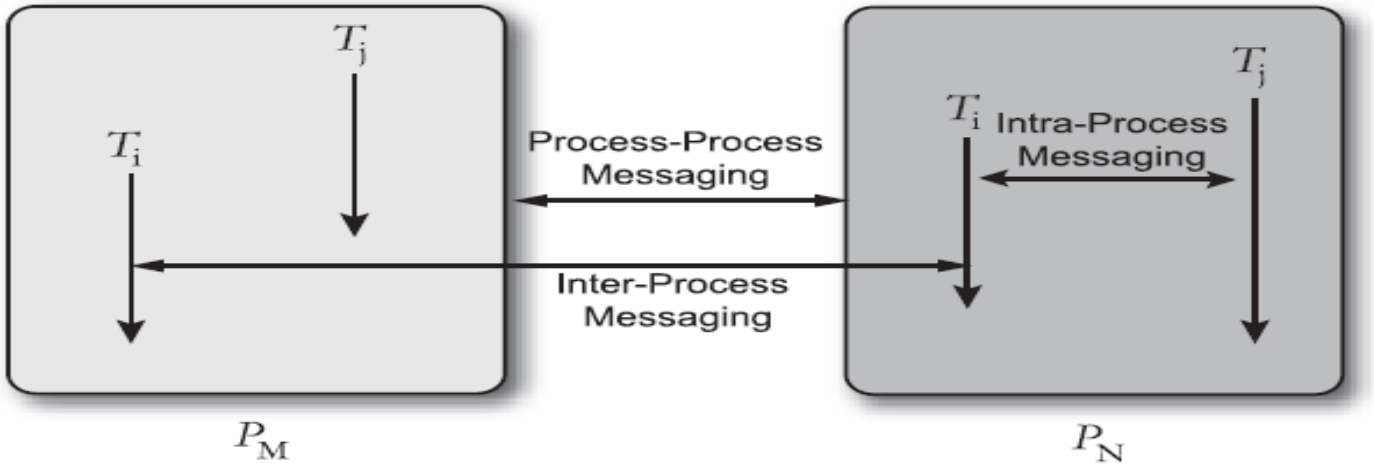


Figure 4.12 Message Passing Model

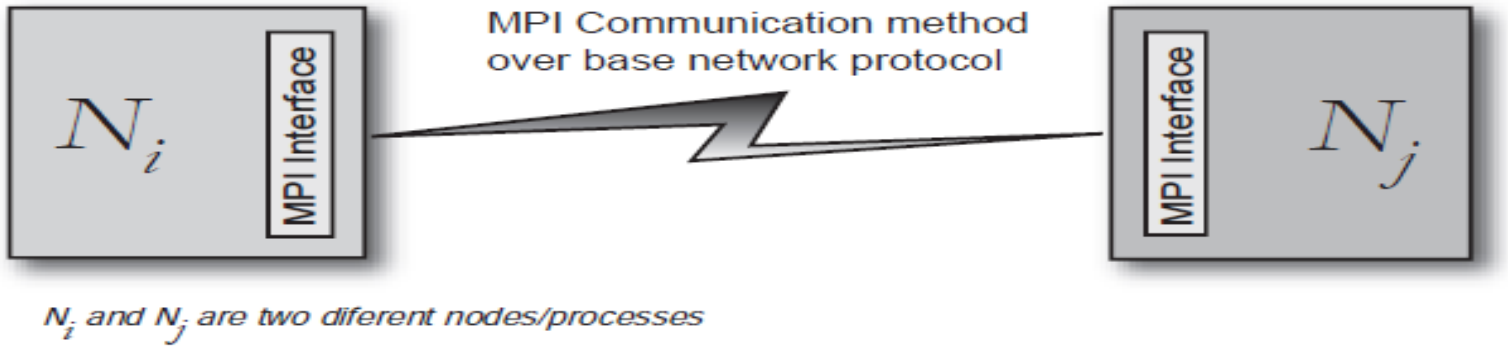
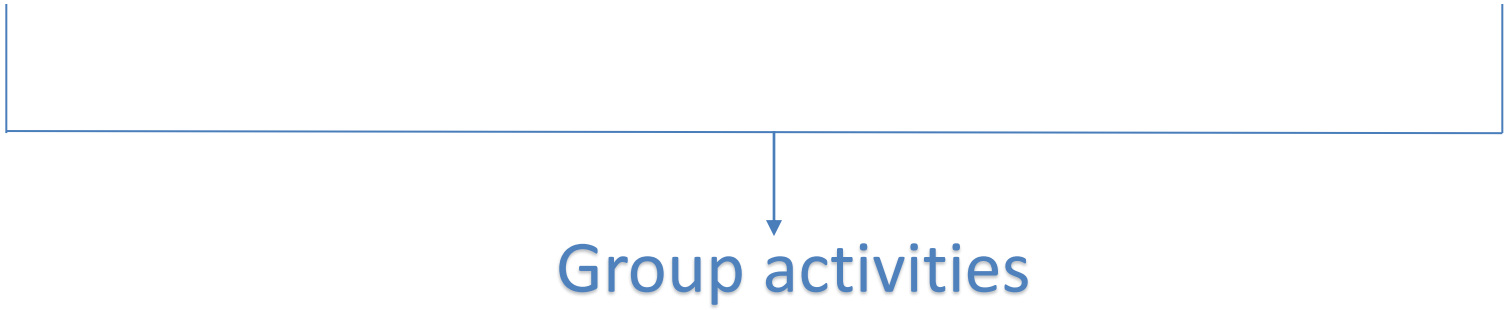


Figure 4.13 Basic MPI Communication Environment

Threading APIs

- This topic will provide an overview of several popular thread packages used by developers today.

1. Threading APIs for Microsoft Windows
2. Threading APIs for Microsoft .NET Framework
3. POSIX Threads



OpenMP

A Portable Solution for Threading

- OpenMP plays a key role by providing an easy method for threading applications without burdening the programmer with the complications of creating, synchronizing, load balancing, and destroying threads.
- The OpenMP standard was formulated in 1997 as an API for writing portable, multithreaded applications.
- The current version is OpenMP Version 2.5, which supports Fortran, C, and C++. Intel C++ and Fortran compilers support the OpenMP Version 2.5 standard.
- The OpenMP programming model provides a platform-independent set of compiler pragmas, directives, function calls, and environment variables that explicitly instruct the compiler how and where to use parallelism in the application.
- Many loops can be threaded by inserting only one pragma right before the loop. The full potential of OpenMP is realized when it is used to thread the most time consuming loops.

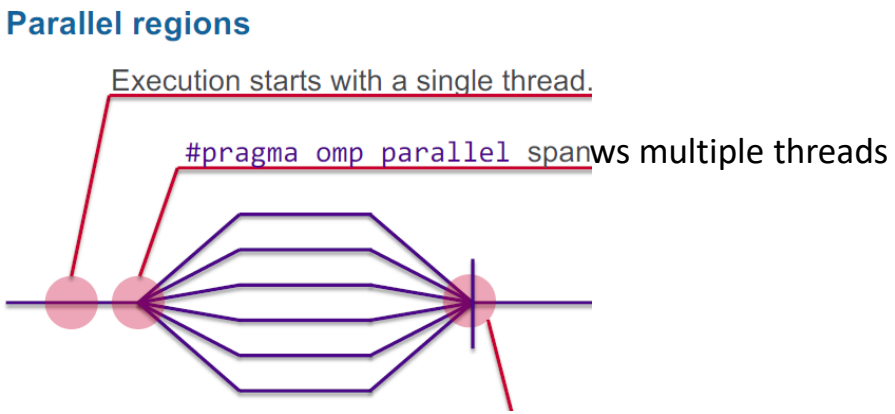
Contd....

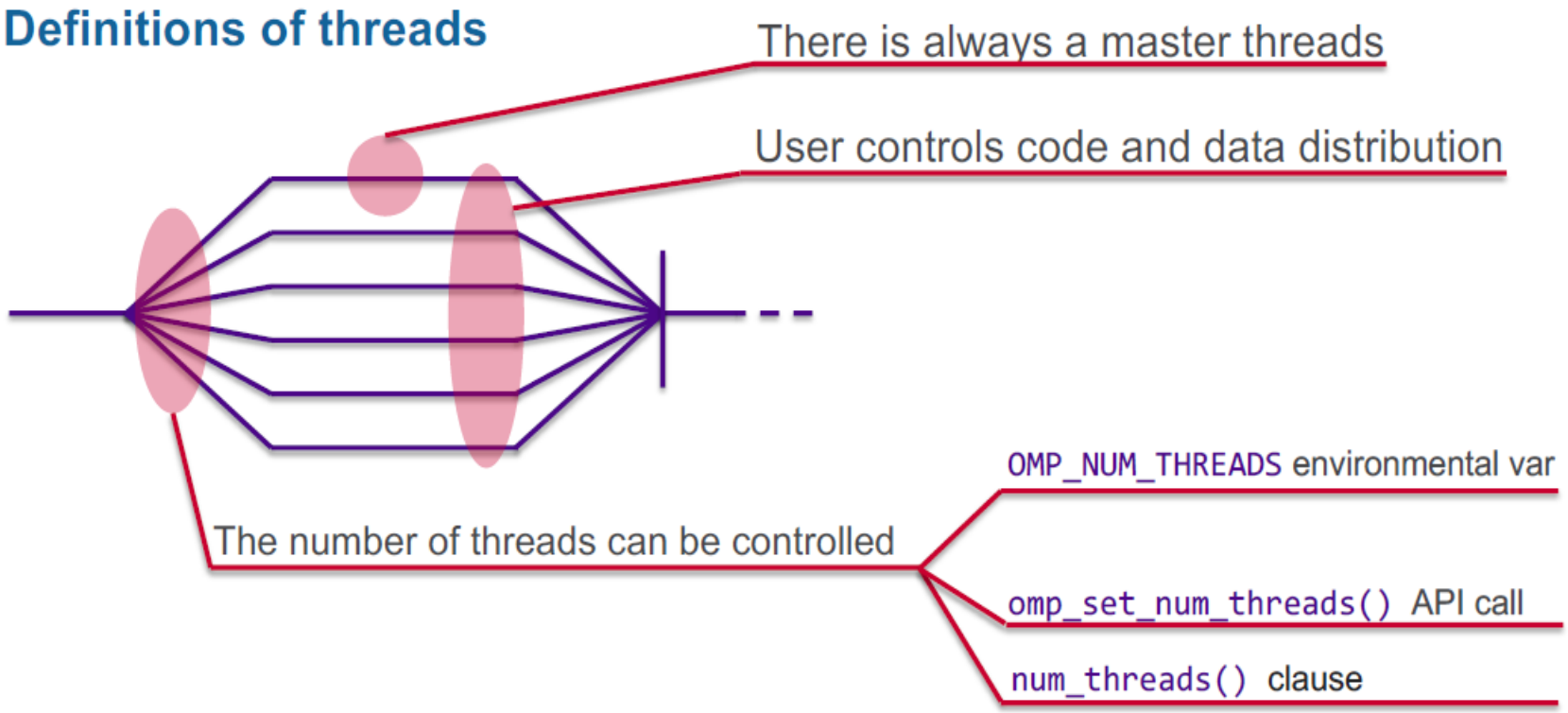
- The simplest way to create parallelism in OpenMP is to use the `parallel` pragma.

```
#pragma omp parallel
{
  // this is executed by a team of threads
}
```

```
// hello.c
#pragma omp parallel
  {
    int t = omp_get_thread_num();
    printf("Hello world from %d!\n",t);
  }
```

Contd....


Parallel regions
Execution starts with a single thread.
#pragma omp parallel spanws multiple threads


Definitions of threads
There is always a master threads
User controls code and data distribution
The number of threads can be controlled
OMP_NUM_THREADS environmental var
omp_set_num_threads() API call
num_threads() clause

Contd....

For instance, if you program computes

```
result = f(x)+g(x)+h(x)
```

you could parallelize this as

```
double result,fresult,gresult,hresult;
#pragma omp parallel
{ int num = omp_get_thread_num();
  if (num==0)      fresult = f(x);
  else if (num==1) gresult = g(x);
  else if (num==2) hresult = h(x);
}
result = fresult + gresult + hresult;
```

Contd....

- The `for loop` construct (or simply the `loop` construct) specifies that the iterations of the following `for loop` will be executed in parallel. The iterations of the loop are distributed among multiple threads.

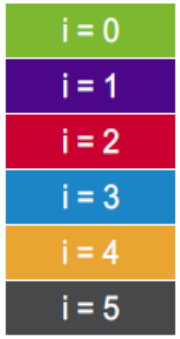
```
#pragma omp parallel for
for(int i = 1; i < 100; ++i)
{
    ...
}
```

- `#pragma omp parallel` spawns a group of threads, while `#pragma omp parallel for` divides loop iterations between the spawned threads.

Contd....

Parallel for/do loop directive

Iteration space:



- Loop is executed in parallel
- Each thread gets a chunk of the iteration space
- **How to distribute the iterations?**
  - A: *schedule()* clause

ile: parallel_for.c

```
1 #pragma omp parallel for
2 for (int i = 0; i < 6; i++)
3 {
4     printf("Hi from %d iteration %d \n", omp_get_thread_num(), i);
5 }
```

Contd....

- The OpenMP implementation determines how many threads to create and how best to manage them.
- All the programmer needs to do is to tell OpenMP which loop should be threaded.
- No need for programmers to add a lot of codes for creating, initializing, managing, and killing threads in order to exploit parallelism.
- OpenMP compiler and runtime library take care of these and many other details behind the scenes.

Challenges in Threading a Loop

The challenges you must identify or restructure the hot loop according to these challenges before adding OpenMP pragmas.

- Loop-carried Dependence
- Data-race Conditions
- Managing Shared and Private Data
- Loop Scheduling and Partitioning
- Effective Use of Reductions

Loop-carried Dependence

- Even if the loop meets all required criteria and the compiler threaded the loop, it may still not work correctly, given the existence of data dependencies that the compiler ignores due to the presence of OpenMP pragmas.
- When a statement in one iteration of a `loop` depends in some way on a statement in a different iteration of the same `loop`, a **loop-carried dependence** exists.

```
for (i = 1; i < length; i++) {
  a[i] = a[i-1] + b[i];
}
```

Here, each iteration of the loop depends on the previous:
iteration `i=3` depends on iteration `i=2`,
iteration `i=4` depends on iteration `i=3`,
iteration `i=5` depends on iteration `i=4`, etc.

This is sometimes called a ***loop carried dependency***.

There is no way to execute iteration `i` until after iteration `i-1` has completed, so this loop can't be parallelized.

Contd....

Table 6.1 The Different Cases of Loop-carried Dependences

	iteration k	iteration $k + d$
	Loop-carried flow dependence	
statement S_1	write L	
statement S_2		read L
	Loop-carried anti-dependence	
statement S_1	read L	
statement S_2		write L
	Loop-carried output dependence	
statement S_1	write L	
statement S_2		write L

In order for `S2` to depend upon `S1` , it is necessary for some execution of `S1` to write to a memory location `L` that is later read by an execution of `S2` . This is also called flow dependence. Other dependencies exist when two statements write the same memory location `L`, called an output dependence, or a read occurs before a write, called an anti-dependence.

Contd....

- Compiler must reason about dependence between instructions
- Three kinds of dependencies:
  - True dependence:

```
(s1)  x = ...
(s2)  ... = x
```

  - Anti dependence:

```
(s1)  ... = x
(s2)  x = ...
```

  - Output dependence:

```
(s1)  x = ...
(s2)  x = ...
```

- Cannot reorder instructions in any of these cases!

Contd....

Limitations to Parallelism

Dependencies:

- RAW: Read-after-write dependences

```
sum = a+1;                          /* << */
first_term = sum * scale1; /* << */
sum = sum+b;
second_term = sum * scale2;
```

- These are true data dependences

Loop-carried dependences

- The following loop cannot (without rewriting) be parallelised with OpenMP

```
a[0] = 1;
for (i = 1; i < N; i++) {
  a[i] = a[i] + a[i-1];
}
```

- i=1: a[1] = a[1] + a[0];
- i=2: a[2] = a[2] + a[1];

Contd....

Detecting dependences

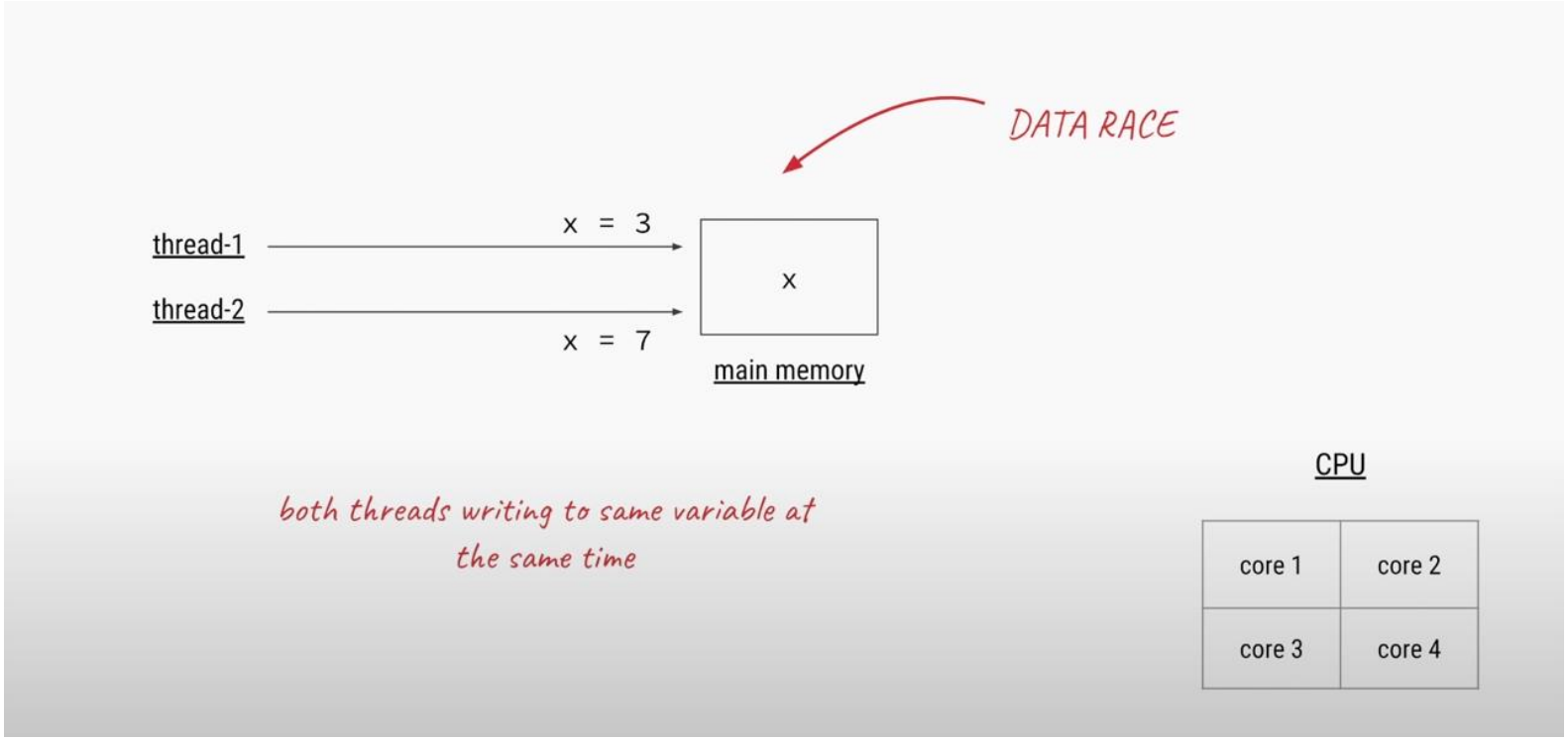
- Analyze how each variable is used within a loop iteration
- Is the variable only read and nevery written? => no dependences
- For each variable written: can there be any accesses in other iterations than the current?
  => there are dependences!

Contd....

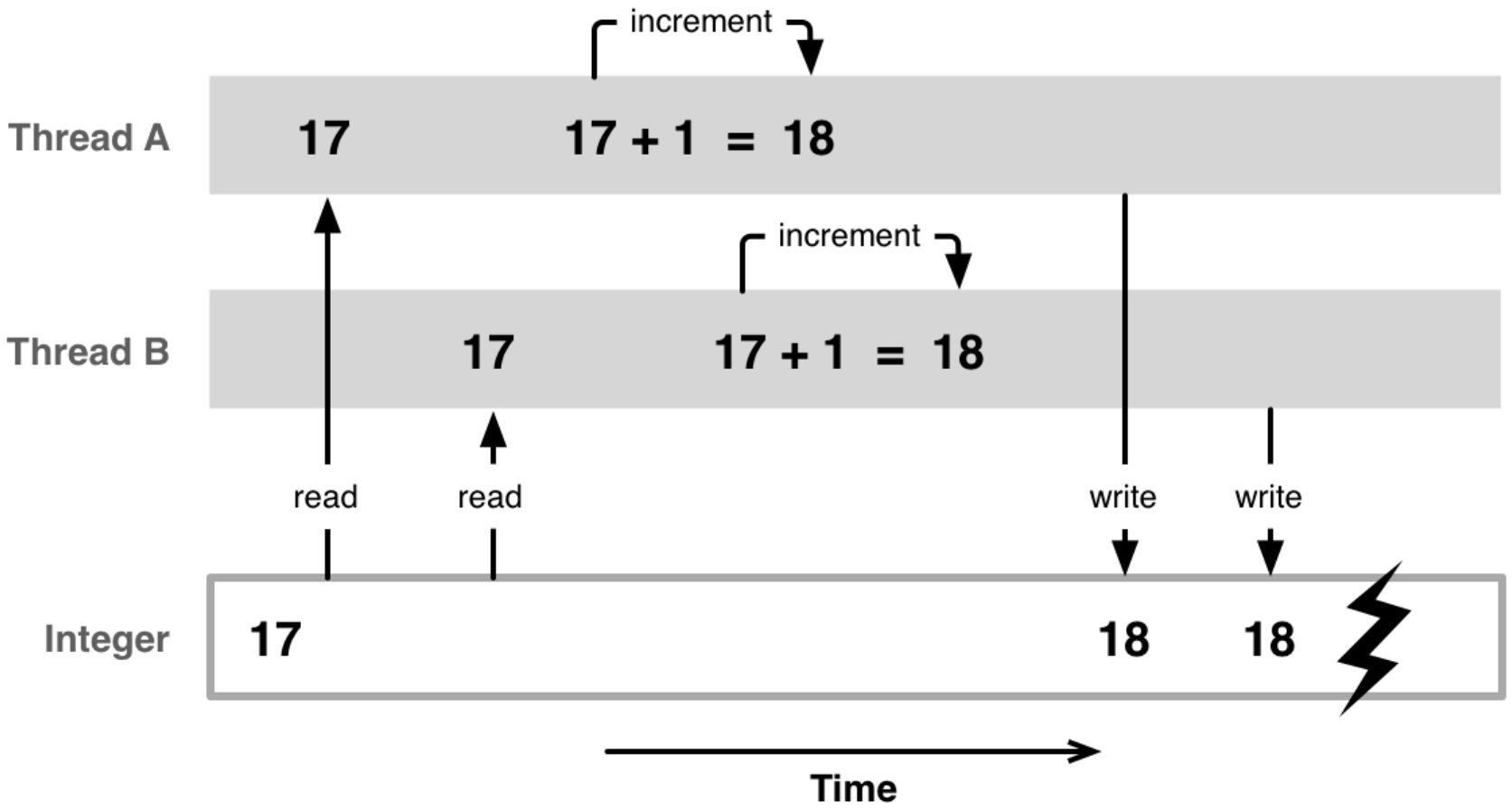
- Because OpenMP directives are commands to the compiler, the compiler will thread the loop.
- However, the threaded code will fail because of loop-carried dependence.
- The only way to fix this kind of problem is to rewrite the loop or to pick a different algorithm that does not contain the loop-carried dependence.

Data-race Conditions

- Data-race condition occurs when, multiple threads attempt to update the same memory location, or variable, without proper synchronization, after threading.



Contd....


increment
Thread A
17
17 + 1 = 18
increment
Thread B
17
17 + 1 = 18
read
read
write
write
Integer
17
18
18
Time

Contd....

- The following example, in which multiple threads are updating the variable `x` will lead to undesirable results.
- In such a situation, the code needs to be modified via privatization or synchronized using mechanisms like Mutexes.
- For example, you can simply add the `private(x)` clause to the parallel for pragma to eliminate the data-race condition on variable `x` for this loop.

```
// A data race condition exists for variable x;
// you can eliminate it by adding private(x) clause.

#pragma omp parallel for
for ( k = 0; k < 80; k++ )
{
    x = sin(k*2.0)*100 + 1;
    if ( x > 60 ) x = x % 60 + 1;
    printf ( "x %d = %d\n", k, x );
}
```

Managing Shared and Private Data

- In writing multithreaded programs, understanding which data is shared and which is private becomes extremely important, not only to performance, but also for program correctness.
- OpenMP makes this distinction apparent to the programmer through a set of clauses such as *`shared`*, *`private`*, and *`default`*, and it is something that you can set manually.
- With OpenMP, it is the developer's responsibility to indicate to the compiler which pieces of memory should be shared among the threads and which pieces should be kept private.
- When memory is identified as shared, all threads access the exact same memory location.
- When memory is identified as private, however, a separate copy of the variable is made for each thread to access in private.

Contd....

- The following loop fails to function correctly because the variable `x` is shared. It needs to be private.
- Given example below, it fails due to the loop-carried output dependence on the variable `x`.
- The `x` is shared among all threads based on OpenMP default shared rule, so there is a data-race condition on the `x` while one thread is reading `x`, another thread might be writing to it.

Contd....

```
#pragma omp parallel for
        for ( k = 0; k < 100; k++ ) {
            x = array[k];
            array[k] = do_work(x);
        }
```

This problem can be fixed in either of the following two ways, which both declare the variable x as private memory.

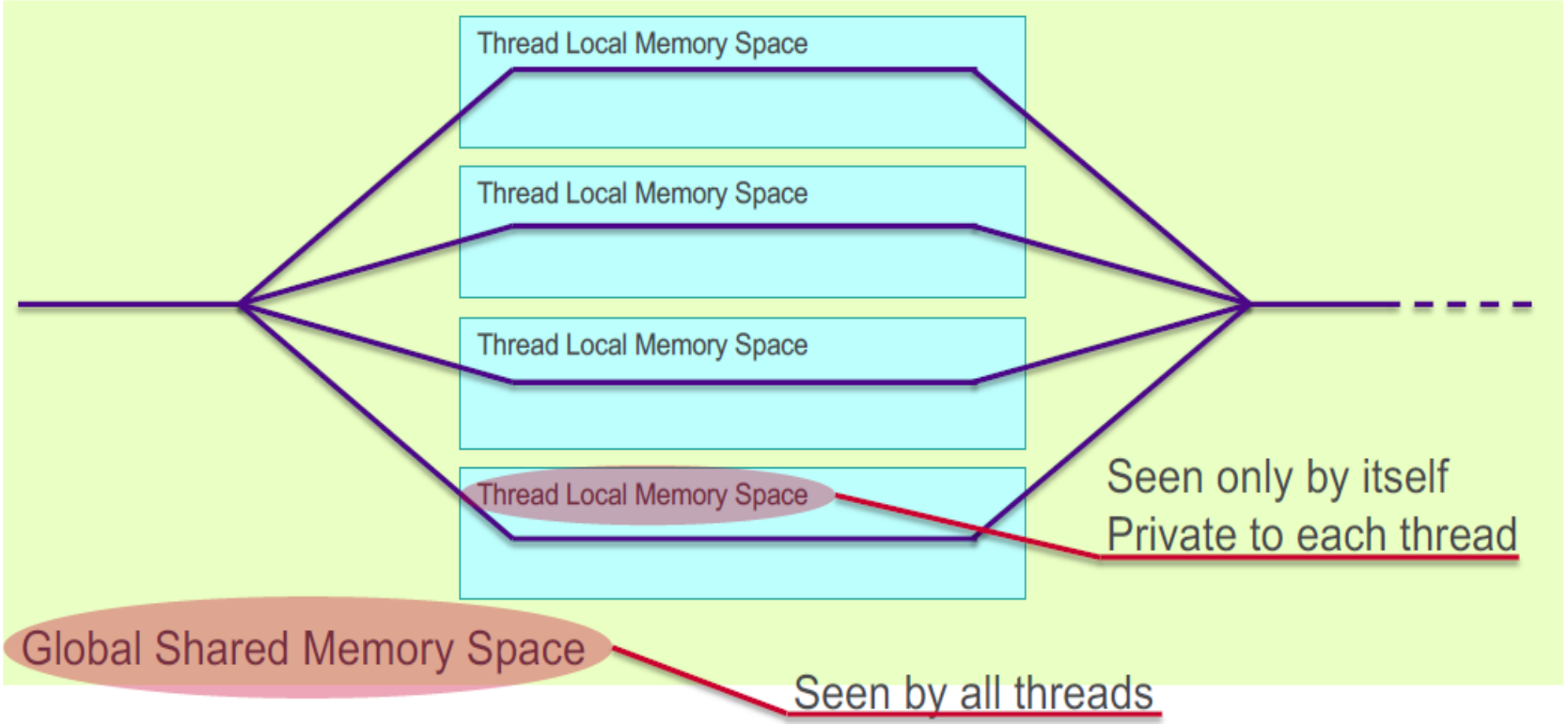
```
// This works. The variable x is specified as private.

#pragma omp parallel for private(x)
for ( k = 0; k < 100; k++ )
{
     x = array[i];
     array[k] = do_work(x);
}


// This also works. The variable x is now private.

#pragma omp parallel for
for ( k = 0; k < 100; k++ )
{
     int x; // variables declared within a parallel
            // construct are, by definition, private
     x = array[k];
     array[k] = do_work(x);
}
```

Every time you use OpenMP to parallelize a loop, you should carefully examine all memory references, including the references made by called functions.

Contd....

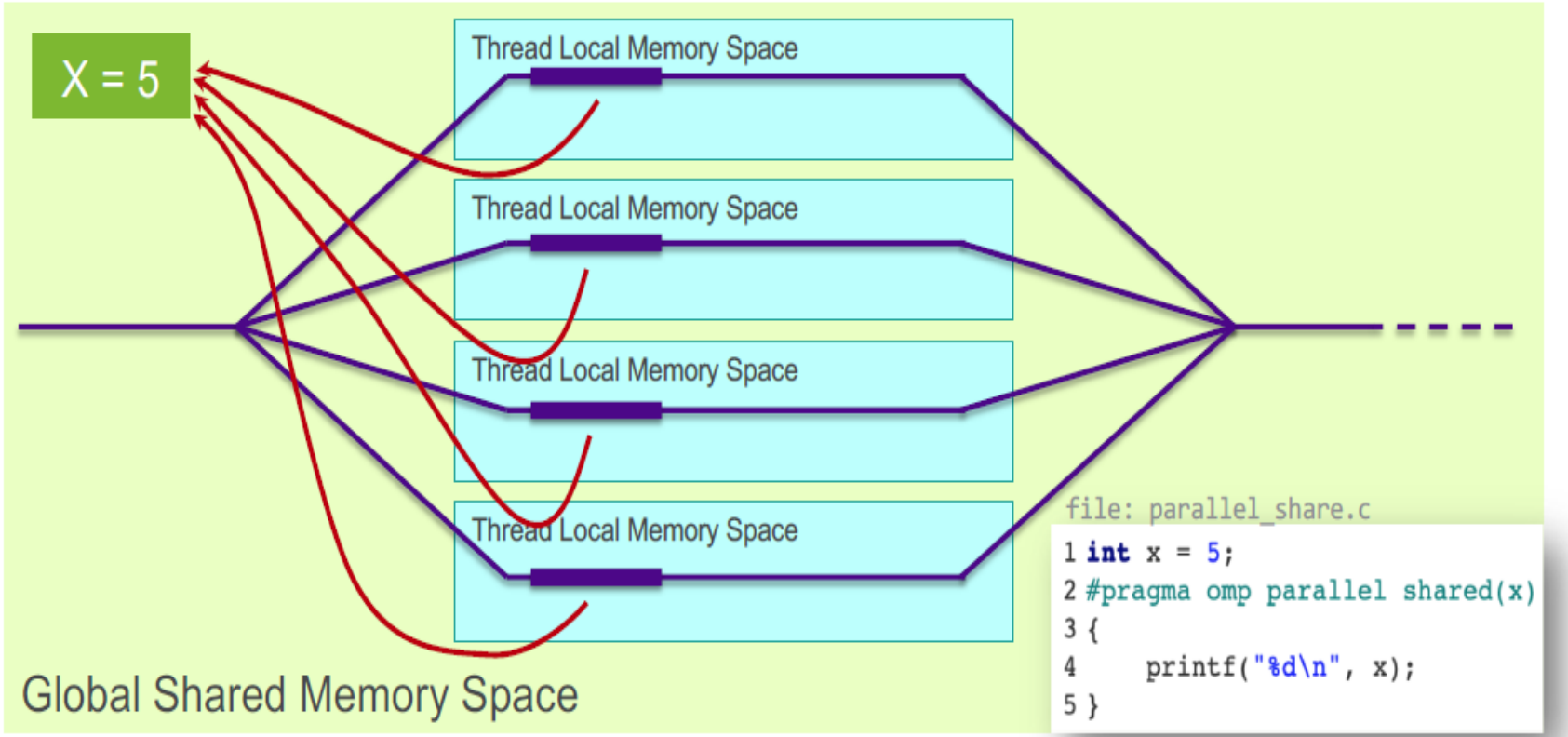
THE OPENMP MEMORY MODEL

Global shared vs thread local memory



Contd....

Shared() clause

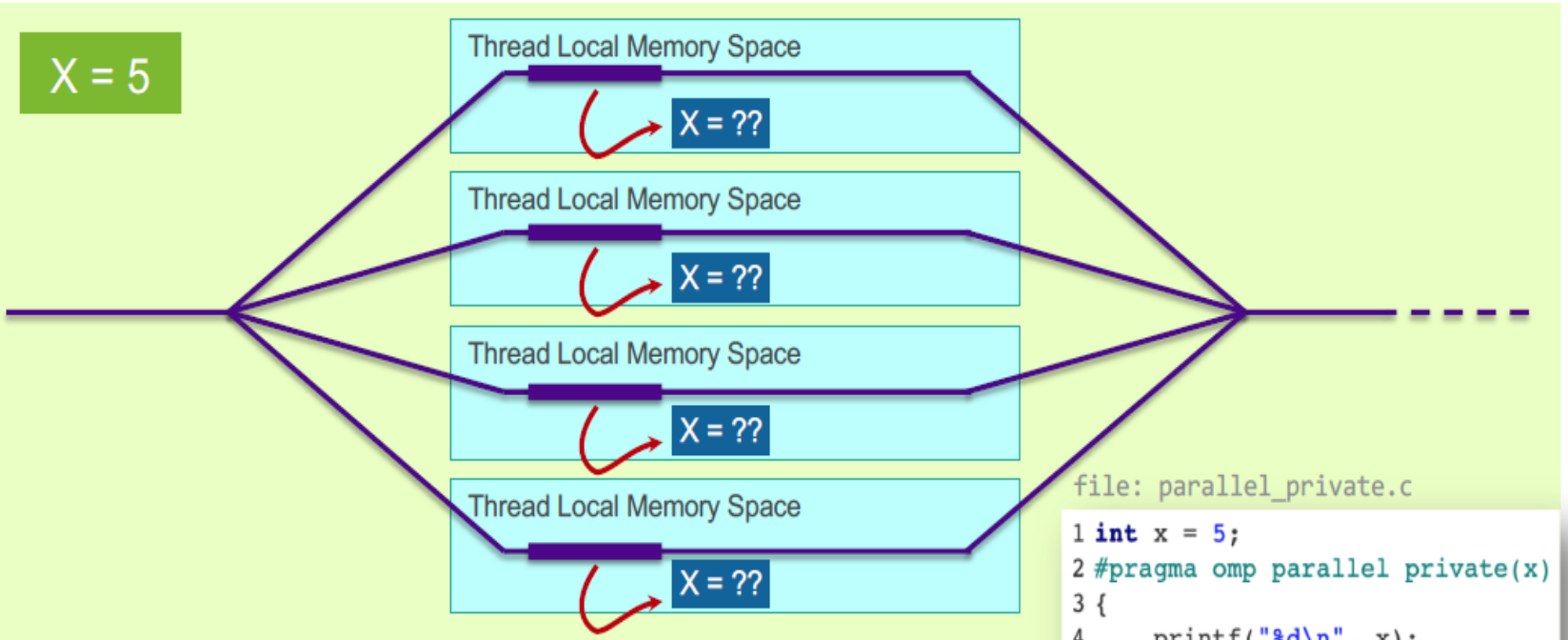


file: parallel_share.c

```
1 int x = 5;
2 #pragma omp parallel shared(x)
3 {
4     printf("%d\n", x);
5 }
```

Contd....

Private() clause



file: parallel_private.c

```
1 int x = 5;
2 #pragma omp parallel private(x)
3 {
4     printf("%d\n", x);
5 }
```

Contd....

```
1  int A, B;
2
3  #pragma omp parallel for private(A) shared(B)
4  for (int i = 0; i < n; i++) {
5    printf("Iteration %d is processed by thread %d\n",
6           i, omp_get_thread_num());
7    // ... iterations will be distributed across available threads...
8    // Each thread has a private copy of variable A
9    // All threads access the same memory location for variable B
10 }
```

Loop Scheduling and Partitioning

- To have good load balancing and thereby achieve optimal performance in a multithreaded application, you must have effective loop scheduling and partitioning.
- The ultimate goal is to ensure that the execution cores are busy most of the time.
- With a poorly balanced workload, some threads may finish significantly before others, leaving processor resources idle and wasting performance opportunities.
- In order to provide an easy way for you to adjust the workload among cores, OpenMP offers four scheduling schemes that are appropriate for many situations: static, dynamic, runtime, and guided.
- The Intel C++ and Fortran compilers support all four of these scheduling schemes.
- In any case, you can provide loop scheduling information via the scheduling clause, so that the compiler and runtime library can better partition and distribute the iterations of the loop across the threads, and therefore the cores, for optimal load balancing.

Contd....

Table 6.2 The Four Schedule Schemes in OpenMP

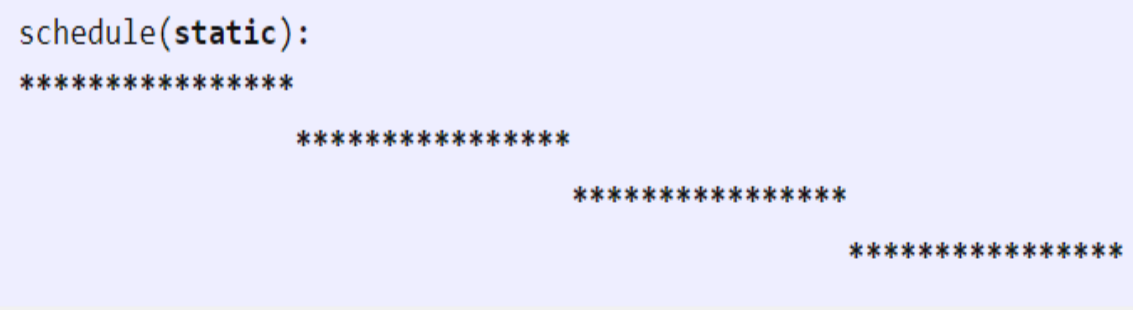
Schedule Type	Description
static (default with no chunk size)	Partitions the loop iterations into equal-sized chunks or as nearly equal as possible in the case where the number of loop iterations is not evenly divisible by the number of threads multiplied by the chunk size. When chunk size is not specified, the iterations are divided as evenly as possible, with one chunk per thread. Set chunk to 1 to interleave the iterations.
dynamic	Uses an internal work queue to give a chunk-sized block of loop iterations to each thread as it becomes available. When a thread is finished with its current block, it retrieves the next block of loop iterations from the top of the work queue. By default, chunk size is 1. Be careful when using this scheduling type because of the extra overhead required.
guided	Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work queue to get more work. The optional chunk parameter specifies the minimum size chunk to use, which, by default, is 1.
runtime	Uses the OMP_SCHEDULE environment variable at runtime to specify which one of the three loop-scheduling types should be used. OMP_SCHEDULE is a string formatted exactly the same as it would appear on the parallel construct.

```
#pragma omp for schedule(kind [, chunk-size])
```

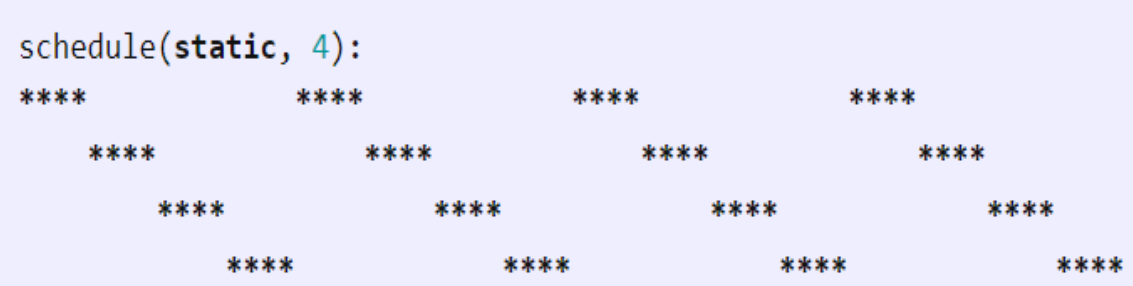
Static

- The `schedule(static, chunk-size)` clause of the loop construct specifies that the for loop has the static scheduling type.
- OpenMP divides the iterations into chunks of size `chunk-size` and it distributes the chunks to threads in a circular order.
- When no `chunk-size specified`, OpenMP divides iterations into chunks that are approximately equal in size and it distributes at most one chunk to each thread.
- Here are three examples of static scheduling. We parallelized a `for loop` with 64 iterations and we used 4 threads to parallelize the `for loop`.

schedule(**static**):



schedule(**static**, 4):



schedule(**static**, 8):

```
#pragma omp parallel for
for ( k = 0; k < 1000; k++ ) do_work(k);
```

Dynamic

- The `schedule(dynamic, chunk-size)` clause of the loop construct specifies that the `for loop` has the dynamic scheduling type. OpenMP divides the iterations into chunks of size `chunk-size`.
- Each thread executes a chunk of iterations and then requests another chunk until there are no more chunks available.
- The dynamic scheduling type has higher overhead then the static scheduling type because it dynamically distributes the iterations during the runtime.

For example, if the chunk size is specified as 16 with the `schedule(dynamic,16)` clause and the total number of iterations is 100, the partition would be `16,16,16,16,16,16,4` with a total of seven chunks.

Here are four examples of dynamic scheduling.

```
schedule(dynamic):
*   ** **  * * *  *      *  *    **   *  *  * *       *  *   *
  *       *     *    * *     * *   *    *        * *   *    *
 *       *    *     * *   *   *     *  *       *  *  *  *  *   *
   *  *     *    * *    *  *    *    *    ** *  *   *     *   *
```

```
schedule(dynamic, 1):
    *    *     *        *   *    * *  *  *         *  * *  * *
*  *  *   * *     *  * * *    * *      *   ***  *   *         *
 *   *  *  *  *    ** *    *      *  *  * *   *  *   *   *
  *    *     *  **        *  * *    *          *  *    *  * *  *
```

```
schedule(dynamic, 4):
           ****                    ****                    ****
****            ****    ****            ****        ****
    ****            ****    ****            ****        ****
        ****                    ****            ****
```

```
schedule(dynamic, 8):
               ********                                ********
                       ********        ********
********                       ********        ********
        ********
```

Guided

- For the guided scheduling, the way a loop is partitioned depends on the number of threads (N), the number of iterations (β0) and the chunk size (S).
- Threads dynamically grab block of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
- For example, given a loop with β0=800, N= 2, and S=80, the loop partition is {200, 150, 113, 85, 80, 80, 80, 12}.
- Similar to dynamic scheduling, but the chunk size starts off large and shrinks in an effort to reduce the amount of time threads have to go to the work queue to get more work.

With dynamic and guided scheduling mechanisms, you can tune your application to deal with those situations where each iteration has variable amounts of work or where some cores (or processors) are faster than others.

Runtime

- The runtime scheduling scheme is actually not a scheduling scheme per se.
- The runtime scheduling type defers the decision about the scheduling until the runtime.
- When runtime is specified in the schedule clause, the OpenMP runtime uses the scheduling scheme specified in the `OMP_SCHEDULE` environment variable for this particular for loop.
- Using `runtime` scheduling gives the end-user some flexibility in selecting the type of scheduling dynamically among three previously mentioned scheduling mechanisms through the `OMP_SCHEDULE` environment variable.

```
export OMP_SCHEDULE=dynamic,16
```

Effective Use of Reductions

- OpenMP provides the `reduction` clause that is used to efficiently combine certain associative arithmetical reductions of one or more variables in a loop.
- The following loop uses the reduction clause to generate the correct results.

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
    for (k = 0; k < 100; k++) {
        sum = sum + func(k);
    }
```

- Given the `reduction` clause, the compiler creates private copies of the variable `sum` for each thread, and when the loop completes, it adds the values together and places the result in the original variable `sum`.

Contd....

- How does OpenMP parallelize a `for loop` declared with a reduction clause?
- OpenMP creates a team of threads and then shares the iterations of the `for loop` between the threads.
- Each thread has its own **local** copy of the reduction variable. The thread modifies only the local copy of this variable.
- Therefore, there is no data race. When the threads join together, all the local copies of the reduction variable are combined to the global shared variable.

Contd....

```
sum = 0;
#pragma omp parallel for shared(sum, a) reduction(+: sum)
for (auto i = 0; i < 9; i++)
{
    sum += a[i]
}
```

and let there be three threads in the team of threads. Each thread has `sumloc`, which is a local copy of the reduction variable. The threads then perform the following computations

- Thread 1

```
sumloc_1 = a[0] + a[1] + a[2]
```

- Thread 2

```
sumloc_2 = a[3] + a[4] + a[5]
```

- Thread 3

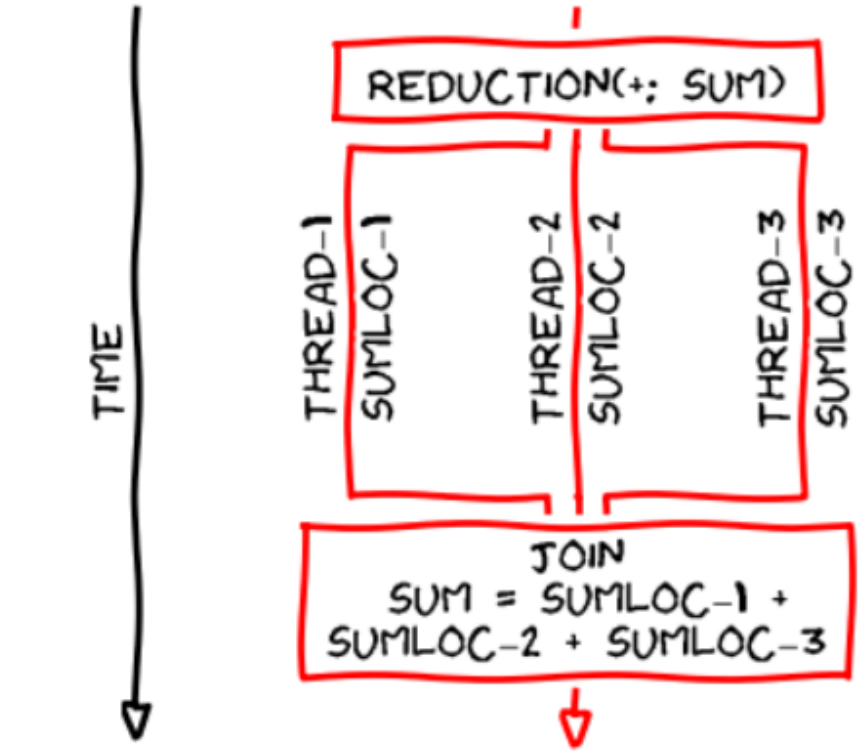
```
sumloc_3 = a[6] + a[7] + a[8]
```

In the end, when the treads join together, OpenMP reduces local copies to the shared reduction variable

```
sum = sumloc_1 + sumloc_2 + sumloc_3
```

Contd....

The following figure is another representation of this process.



How to combine values into a single accumulation variable (avg)?

Contd....

Table 6.3 Reduction Operators and Reduction Variable's Initial Value in OpenMP

Operator	Initialization Value
+ (addition)	0
- (subtraction)	0
* (multiplication)	1
& (bitwise and)	~0
| (bitwise or)	0
^ (bitwise exclusive or)	0
&& (conditional and)	1
|| (conditional or)	0

Minimizing Threading Overhead

- Using OpenMP, you can parallelize loops, regions, and sections or straight-line code blocks, whenever dependences do not forbids them being executed in parallel.
- In addition, because OpenMP employs the simple fork-join execution model, it allows the compiler and run-time library to compile and run OpenMP programs efficiently with lower threading overhead.
- However, you can improve your application performance by further reducing threading overhead.

Contd....

- Consider the following example:

```
#pragma omp parallel for for
( k = 0; k < m; k++ ) {
    fn1(k); fn2(k);
}

#pragma omp parallel for // adds unnecessary overhead
for ( k = 0; k < m; k++ ) {
    fn3(k); fn4(k);
}
```

The overhead can be removed by entering a parallel region once, then dividing the work within the parallel region. The following code is functionally identical to the preceding code but runs faster, because the overhead of entering a parallel region is performed only once.

Contd....

```
#pragma omp parallel
{
    #pragma omp for
    for ( k = 0; k < m; k++ ) {
       fn1(k); fn2(k);
    }

    #pragma omp for
    for ( k = 0; k < m; k++ ) {
       fn3(k); fn4(k);
    }
}
```

Work-sharing Sections

- A work-sharing section is a construct used to handle non-loop code.
- The work-sharing `sections` construct directs the OpenMP compiler and runtime to distribute the identified sections of your application among threads in the team created for the parallel region.
- The following example uses work-sharing `for` loops and work-sharing `sections` together within a single parallel region. In this case, the overhead of forking or resuming threads for `parallel sections` is eliminated.

Contd....

```
#pragma omp parallel
{
    #pragma omp for
    for ( k = 0; k < m; k++ ) {
       x = fn1(k) + fn2(k);
    }

    #pragma omp sections private(y, z)
    {
       #pragma omp section
          { y = sectionA(x); fn7(y); }
       #pragma omp section
          { z = sectionB(x); fn8(z); }
    }
}
```

- Here, OpenMP first creates several threads. Then, the iterations of the loop are divided among the threads.
- Once the loop is finished, the sections are divided among the threads so that each section is executed exactly once, but in parallel with the other sections.
- If the program contains more sections than threads, the remaining sections get scheduled as threads finish their previous sections.

Thank you