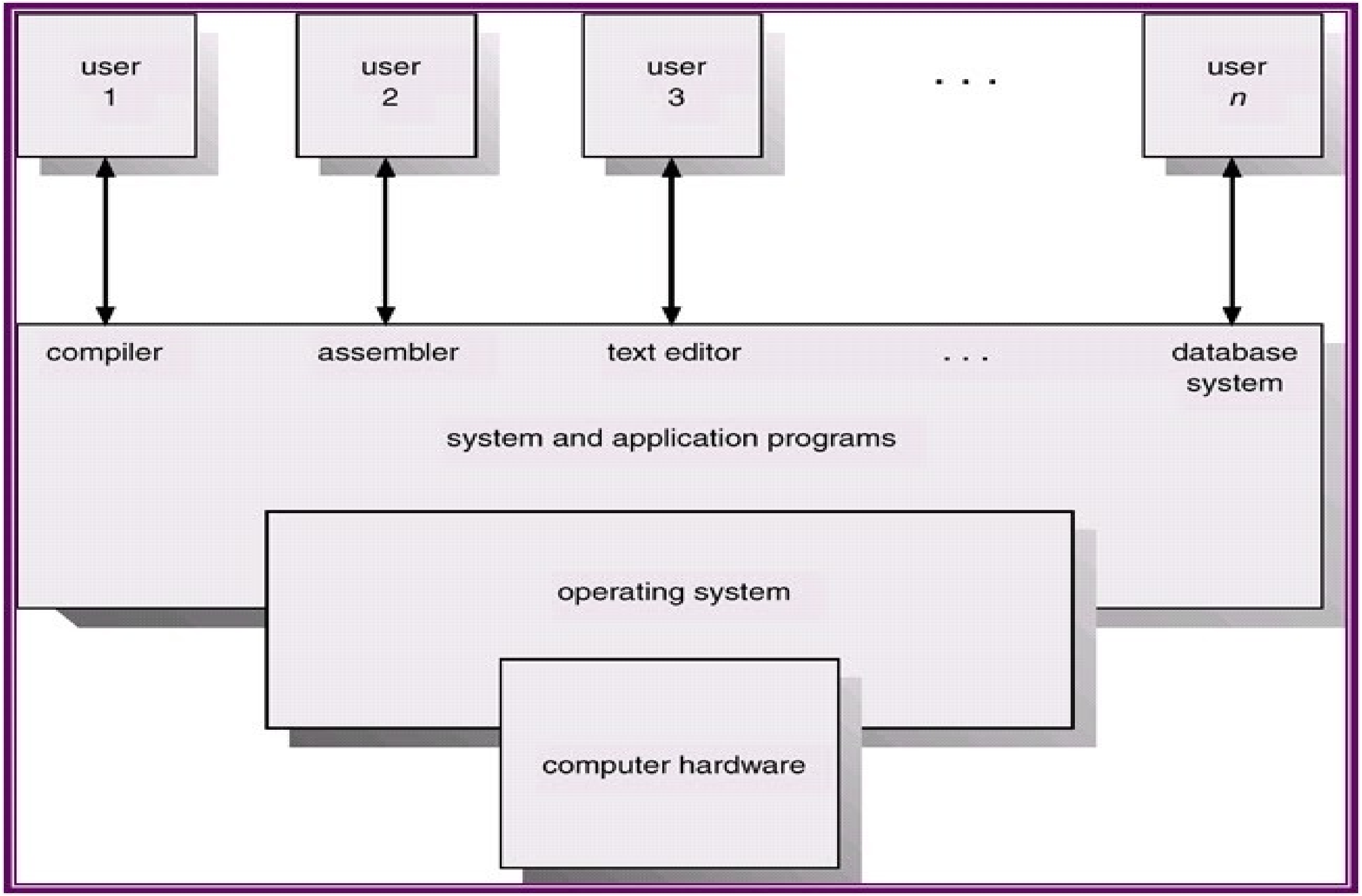
Computer Systems
Concurrency & Threads

A view of an operating System



What does this imply?

- Multiple ‘users’
  - Working independently
  - The operating system:
    - Protects memory from interference from other processes
    - Interleaves execution of each process to:
      - Maximise resource utilisation
      - Responsiveness of each process
    - Preserves the states of each process so it can resume execution

Historically ….

- Computers were expensive:
  - Maximise utilisation

- Now:
  - Users expect concurrent execution of multiple tasks
    - Stream music
    - Update software
    - Edit file
    - Test program
    - Monitor SNS (Social Networking Services)
    - etc.

So?

- The problem is that these processes do not *always* work independently:
  - They can compete for access to resources:
    - Devices, files, data etc. that are local (or remote)
  - They can co-operate – through messages, shared memory, files etc.
  - Applications are distributed:
    - So, potentially, millions of people are reading or updating information *at the same time*
    - SNS, bank accounts, on line shopping ….

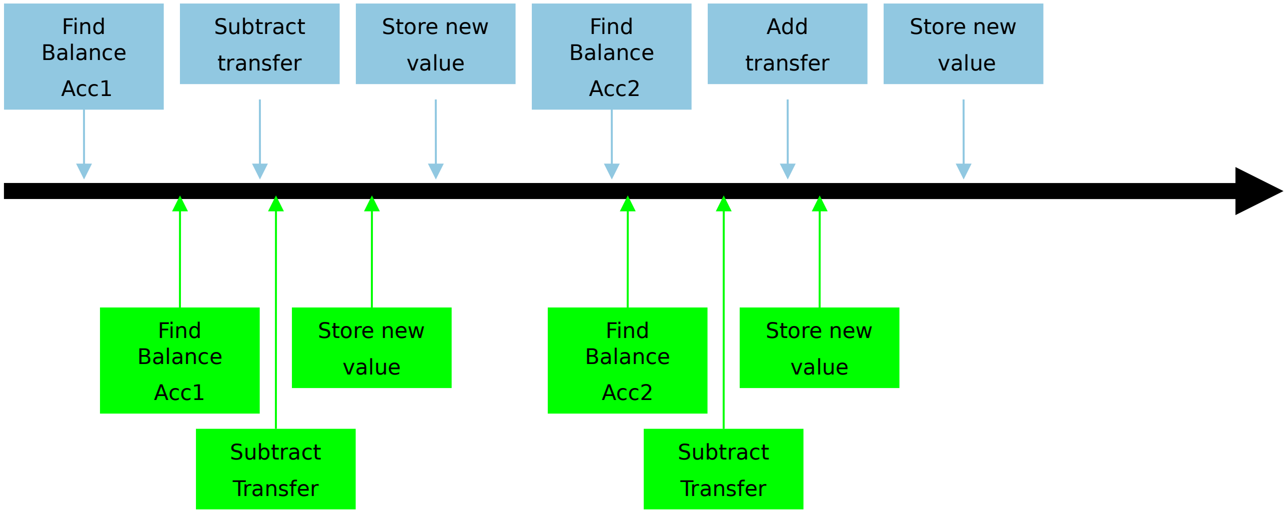
And …

- We need to build applications that can interleave multiple independent (but potentially conflicting) tasks:
  - Consider an app with a GUI:
    - do we want it to freeze when it is performing some time consuming task?
  - What about a database of bank accounts?
    - Do we want to wait in a queue to perform a transaction?
  - What about programmers’ efficiency & effectiveness?
    - Do we want them to have to ‘build an OS’ for every app?
- What about the effectiveness of use of our computing resources?
  - We want to maximise the efficiency

This leads us to two things:

- How can we manage concurrency?
  - How can we ensure that our concurrent execution behaves correctly?
  - How can we do this conveniently?
  - How can we do this efficiently?
    - To maintain responsiveness
- How can we provide programming mechanisms *within a program* to allow concurrency?
  - Threads

Let’s think about a bank transfer



This is unlikely:

- but possible!
- and hard to debug!

Concurrency

- Introduction to Concurrency
- OS Concerns and Process Interaction
- Process Synchronization
- Critical Section Problem
- Requirements for Solutions to CS Problem
- Software Solutions
- Hardware Solutions
- High Level OS Support

Multiple Processes

- Operating System design is concerned with the management of processes and threads
  - Multiprogramming (Multitasking)
  - Multiprocessing
  - Distributed Processing

Concurrency in Different Contexts

Multiple Applications
- invented to allow processing time to be shared among active applications

Structured Applications
- extension of modular design and structured programming

Operating System Structure
- OS themselves implemented as a set of processes or threads

Concurrency: Key Terms

atomic operation	A function or action implemented as a sequence of one or more instructions that appears to be indivisible; that is, no other process can see an intermediate state or interrupt the operation. The sequence of instruction is guaranteed to execute as a group, or not execute at all, having no visible effect on system state. Atomicity guarantees isolation from concurrent processes.
critical section	A section of code within a process that requires access to shared resources and that must not be executed while another process is in a corresponding section of code.
deadlock	A situation in which two or more processes are unable to proceed because each is waiting for one of the others to do something.
livelock	A situation in which two or more processes continuously change their states in response to changes in the other process(es) without doing any useful work.
mutual exclusion	The requirement that when one process is in a critical section that accesses shared resources, no other process may be in a critical section that accesses any of those shared resources.
race condition	A situation in which multiple threads or processes read and write a shared data item and the final result depends on the relative timing of their execution.
starvation	A situation in which a runnable process is overlooked indefinitely by the scheduler; although it is able to proceed, it is never chosen.

Principles of Concurrency

Interleaving and Overlapping

- Can be viewed as examples of **concurrent** processing
- Both present the **same** problems

Uniprocessor – the relative timing of execution of processes cannot be predicted:

- Depends on activities of other processes
- The way the OS handles interrupts
- Scheduling policies of the OS

Difficulties of Concurrency

- Sharing of global resources
- Difficult for the OS to manage the allocation of resources optimally
- Coordination of ‘parallel’ and asynchronous ‘processes’
  - Ensuring correct behaviour
  - Debugging of programming errors:
    - behaviour can be *non-deterministic* and, therefore, hard to identify and reproduce

Operating System Concerns

- Design and management issues raised by the existence of concurrency:
- The OS must:
  - be able to **keep track** of various **processes**
  - **allocate** and **de-allocate** resources for each active process
  - **protect** the data and physical resources of each process against *interference by other processes*
  - ensure that the processes and outputs are **independent** of the **processing speed**

Process Interaction

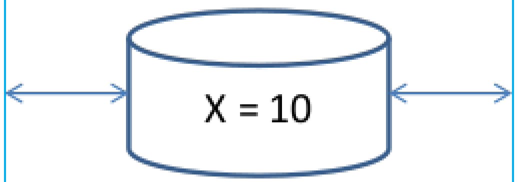
Degree of Awareness	Relationship	Influence that One Process Has on the Other	Potential Control Problems
Processes unaware of each other	Competition	• Results of one process independent of the action of others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation
Processes indirectly aware of each other (e.g., shared object)	Cooperation by sharing	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Mutual exclusion • Deadlock (renewable resource) • Starvation • Data coherence
Processes directly aware of each other (have communication primitives available to them)	Cooperation by communication	• Results of one process may depend on information obtained from others • Timing of process may be affected	• Deadlock (consumable resource) • Starvation

Resource Competition

- Concurrent processes come into conflict when they are competing for use of the same resource
  - for example: I/O devices, memory, processor time, clock
- In the case of competing processes **three control problems** must be faced:
  - Mutual Exclusion – to ensure correct behaviour
  - Deadlocks
  - Starvation

Synchronization of Processes – Example

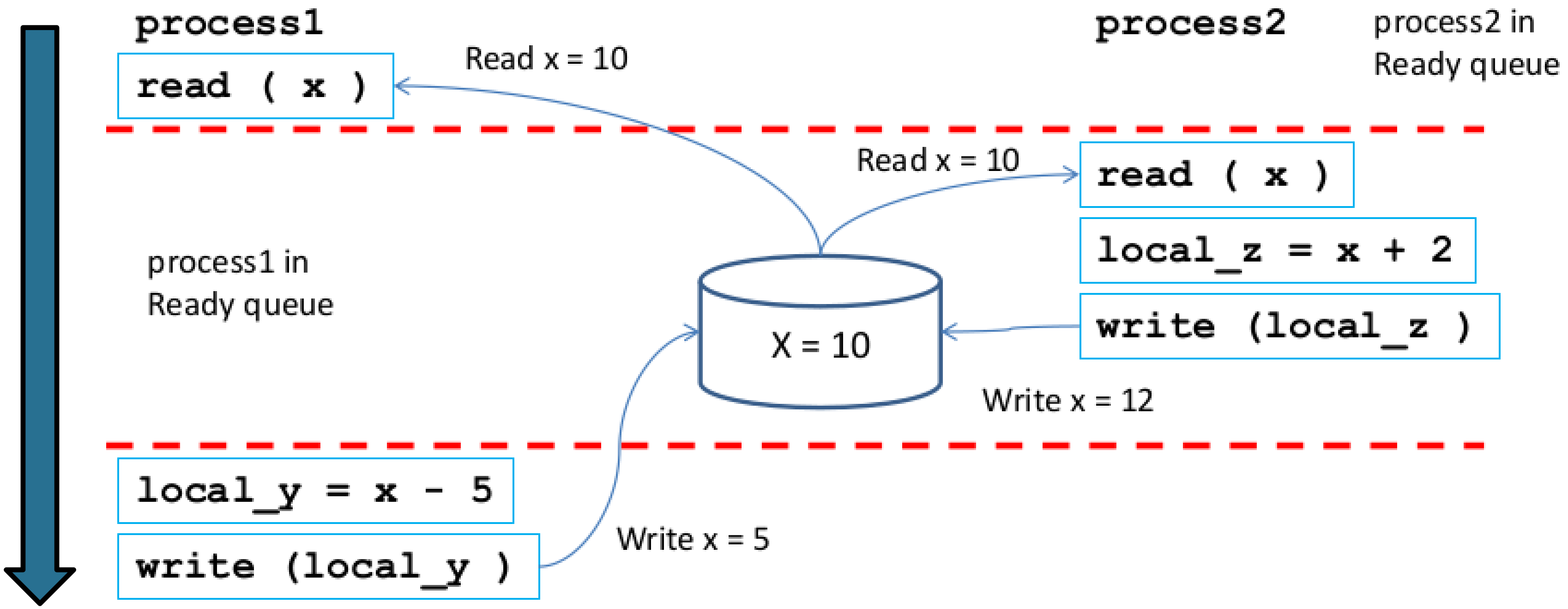
```
process1(){
    read ( x )
    local_y = x - 5
    write (local_y )
}
```




```
process2() {
    read ( x )
    local_z = x + 2
    write (local_z )
}
```

- We expect
  - When process1 finishes, shared variable x is *reduced by 5*
  - When process2 finishes, shared variable x is *increased by 2*

Synchronization of Processes – Example



- Context switches may occur *at any time*
- **Process 2 has its result overwritten by process 1**
- **Process 1 operates with outdated information**

What is a Race Condition?

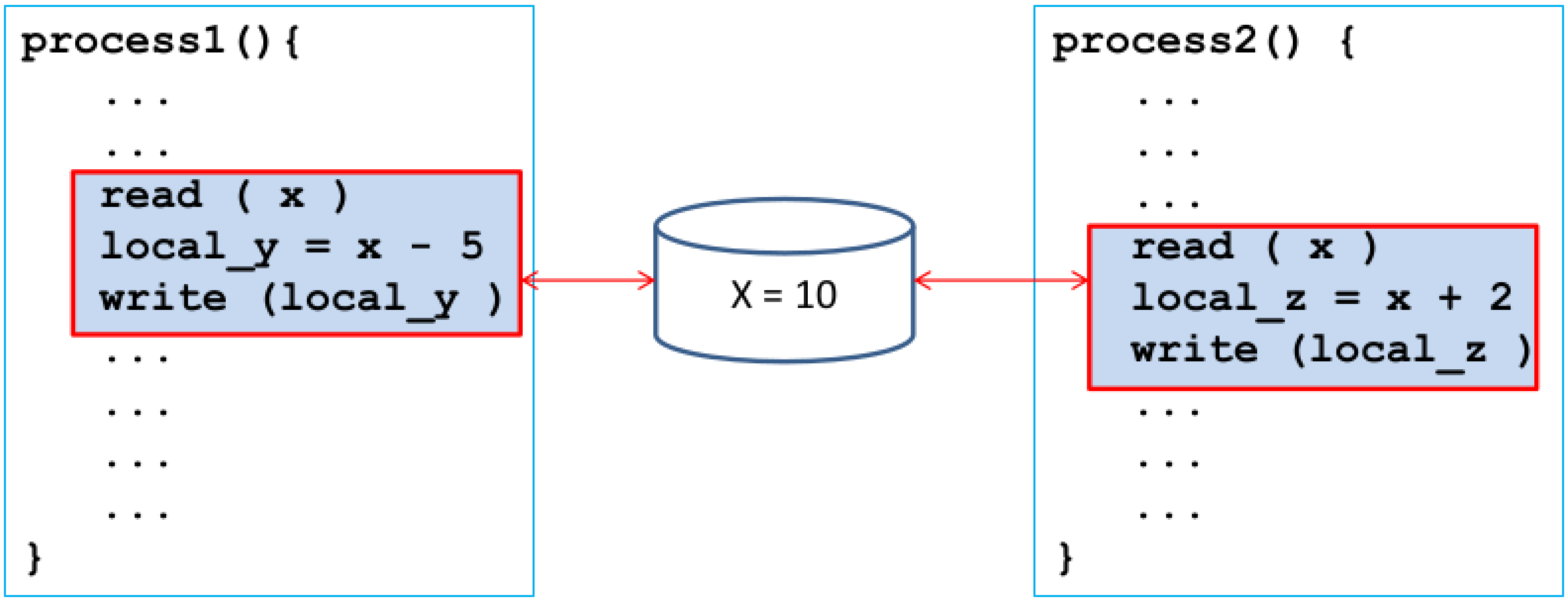
- Occurs when multiple processes or threads **read** and **write** shared data items
- The processes **“race”** to perform their read /write actions
- The final result depends on the order of execution
  - The **“loser”** of the race is the process that performs the last update and determines the final value of a shared data item

Why do Race Conditions Occur?

- **Order of Execution:** Whenever the state of a shared resource depends on the precise execution order of the processes
- **Scheduling:** Context switches at arbitrary times during execution
- **Outdated Information:** Processes / Threads operate with **Stale** or **Dirty** copies of memory values in registers / local variables
  - Other processes may already have changed the original value in the shared memory location
  - **How can we avoid race conditions?**

Critical Section

- Part of the program code that accesses a shared resource



- In order to avoid race conditions, we have to control the concurrent execution of the **critical sections**
  - Strict Serialization i.e. **Mutual Exclusion**

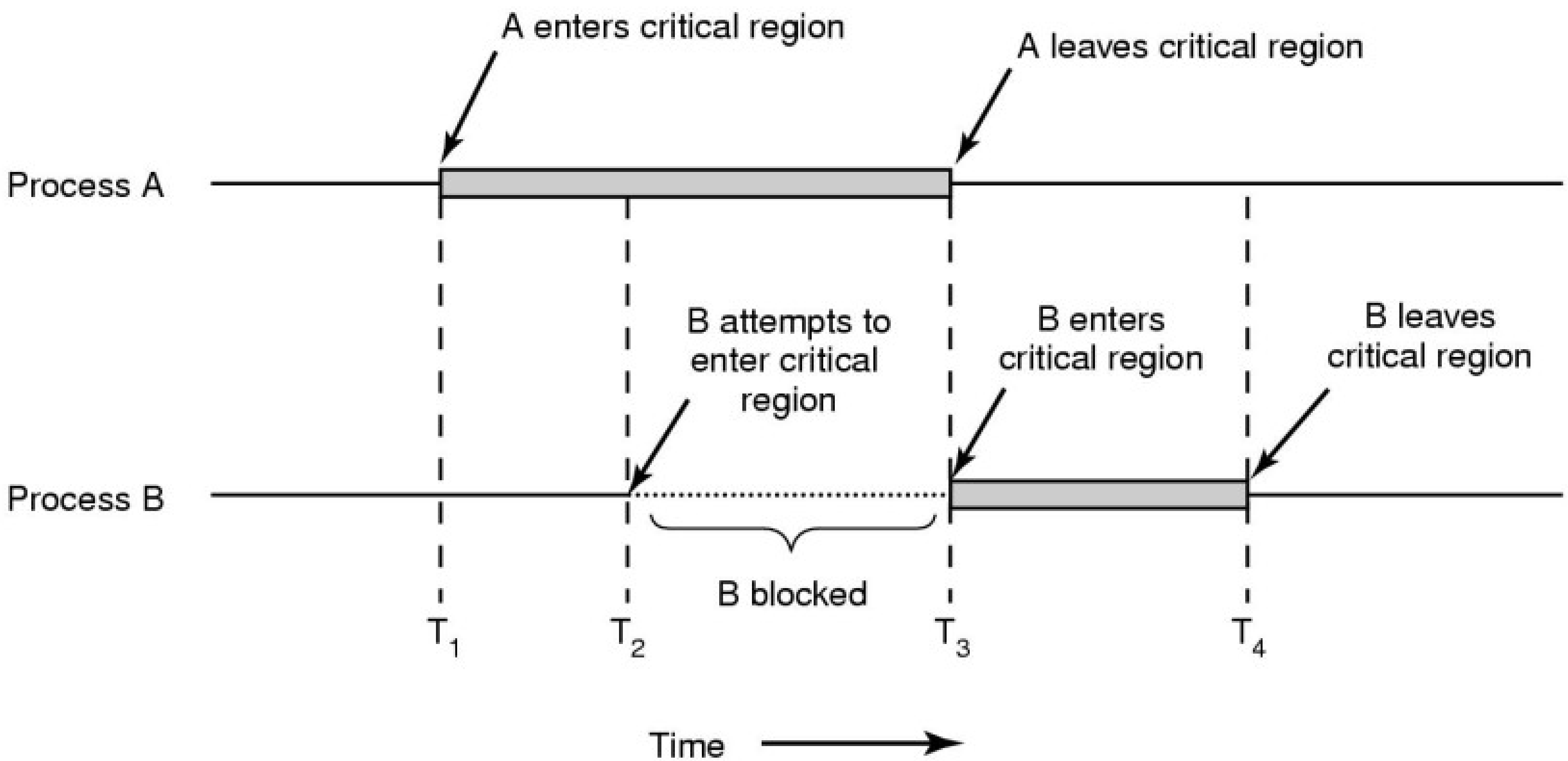
Critical Section

```
process ()
{
    entry_protocol()
        critical_section()
    exit_protocol()
}
```

- **Entry Protocol**
  - Process requests permission for **entering a critical section**; May have to wait / suspended until entry is granted
  - Process has to communicate that it **entered critical section**
- **Exit Protocol**
  - Process communicates to other processes that it has **left the critical section**

A cretical section can be interupted

Critical Section / Mutual Exclusion



- Important: A process can finish the execution of its critical section, even if it is pre-empted or interrupted.

Deadlock and Starvation

- Enforcing mutual exclusion creates two new problems
  - **Deadlocks** – Processes wait forever for each other to free resources
  - **Starvation** – A Process waits forever to be allowed to enter its critical section
- Implementing mutual exclusion has to account for these problems!

Requirements for Solutions to the Critical Section Problem

1) **Serialization of Access:** Only one process at a time is allowed in the critical section for a resource
2) **Bounded Waiting (No Starvation):**
   - A process waiting to enter a critical section, must be guaranteed entry (within some defined limited waiting time)
   - Scheduling algorithm has to guarantee that process is eventually scheduled and can progress

Requirements for Solutions to the Critical Section Problem (cont'd)

3) Progress (Liveness, No Deadlock)

- A process that halts in its noncritical section must do so without interfering with other processes currently waiting to enter their critical section
- Only processes currently waiting to enter their critical section are involved in the selection of the one process that may enter
- A process remains inside its critical section for a finite time only

Solutions to the Critical Section Problem

1) Software Solutions

- Shared lock variables
- Busy Waiting (Polling / Spinning / Strict Alternation)

2) Hardware Solutions

- Disabling Interrupts
- Special H/W Instructions (like compare & swap)

3) Higher Level OS Constructs

- Semaphores, Monitors, Message Passing

Lock Variables

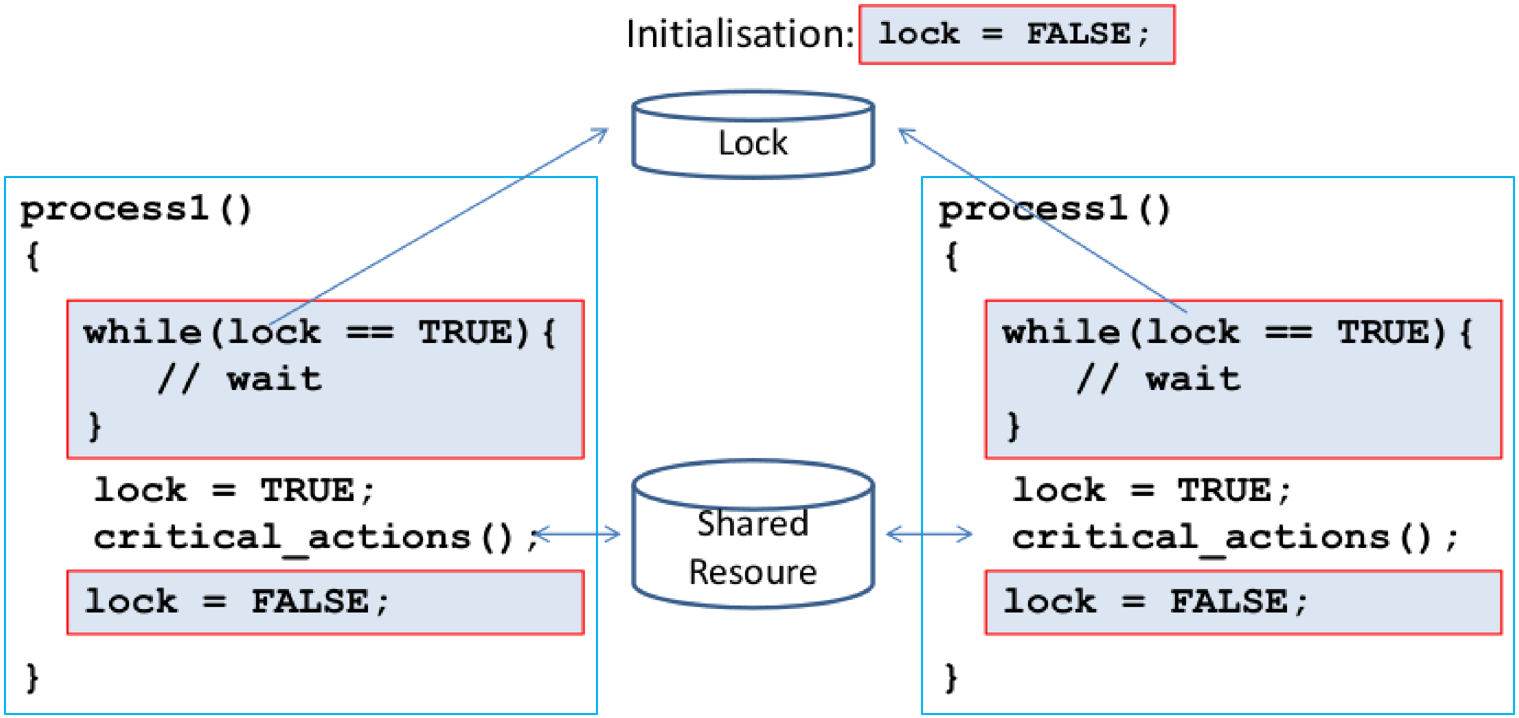
- Critical sections must be protected by some form of a “lock”, where a lock is:
  - A shared data item
  - Processes have to “acquire” such a lock before entering a critical section
  - Processes have to “release” a lock when exiting critical section
- A lock is also called a **Mutex**.

mutual exclusion

```
process ()
{
    acquire lock

        critical_section() ;

    release lock

        remainder_section() ;
}
```

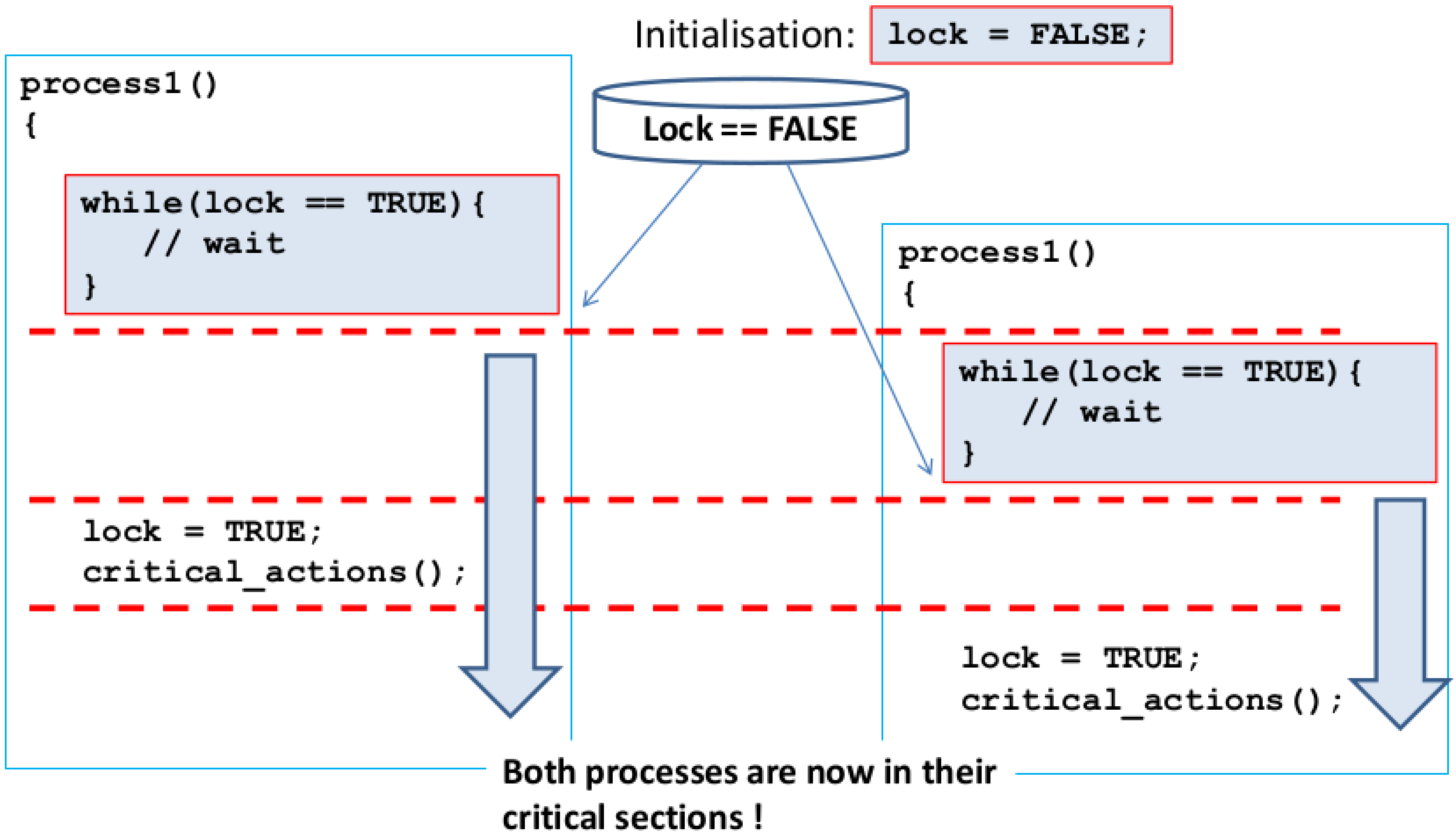
Shared Lock / Mutex

- Two State Lock (shared variable)
  - **True** = Critical Section Locked, **False** = Critical Section Unlocked



Problem: As lock variable is itself a shared resource, race conditions can occur on it!

Shared Lock – Problem



Solution: Use Atomic Instructions for Shared Lock!

Busy Waiting (Polling, Spinning)

- A process **continuously evaluates** whether a lock has become available
- Lock is represented by a data item held in shared memory (IPC via shared memory)
- Process consumes CPU cycles without any progress
- A process busy waiting may prevent another process holding the lock from executing and completing its critical section and from releasing the lock
- **Spin locks** are used at kernel level (special HW instructions)

Busy Waiting (Strict Alternation)

- Strict alternation between two processes – a **process waits** for its **turn**
- Use a "token" as shared variable – usually process ID
  - indicates which process is the next to enter critical section, set by previous process
- **Entry to critical section**
  - Process Pi busy-waits until token == i (its own process ID)
- **Exit from critical Section**
  - Process Pi sets token to next process ID

Busy Waiting (Strict Alternation)

Process 0

```
while(TRUE){

    while(turn != 0){
        // wait
    }

      Critical_Section

    turn = 1;

      Non_Critical_Section
      ...
}
```

Global Variable

```
int turn ;
```

Process 1

```
while(TRUE){

    while(turn != 1){
        // wait
    }

      Critical_Section

    turn = 0;

      Non_Critical_Section
      ...
}
```

- Mutual exclusion is guaranteed!
- Liveness / Progression problem
  - Both process depend on a change of the “turn” variable
  - If one of the processes is held up in its non-critical section, it cannot do that and will block the other process

Hardware Solutions – Disabling Interrupts

- In uniprocessor systems, **concurrent** processes cannot be overlapped, they can only be **interleaved.**
- **Disabling interrupts** guarantees mutual exclusion!
- However, the efficiency of execution could be noticeably degraded.
- This approach will not work in a multiprocessor architecture

```
while (true) {
    /* disable interrupts */;
    /* critical section */;
    /* enable interrupts */;
    /* remainder */;
}
```

Special H/W Instructions

- Applicable to **any number of processes** on either uniprocessor or multiprocessors sharing main memory
- **Simple** and **easy** to verify
- Can be used to **support multiple critical sections**
- **Busy-waiting** is employed
- **Starvation** and **Deadlocks** are possible!

Abstractions

- Usually, we build upon abstractions:
  - The system implements mechanisms correctly and efficiently and we use these:
    - Database systems
    - Python & Java
- There are other solutions to these problems that *may* be more efficient – in some circumstances

Summary

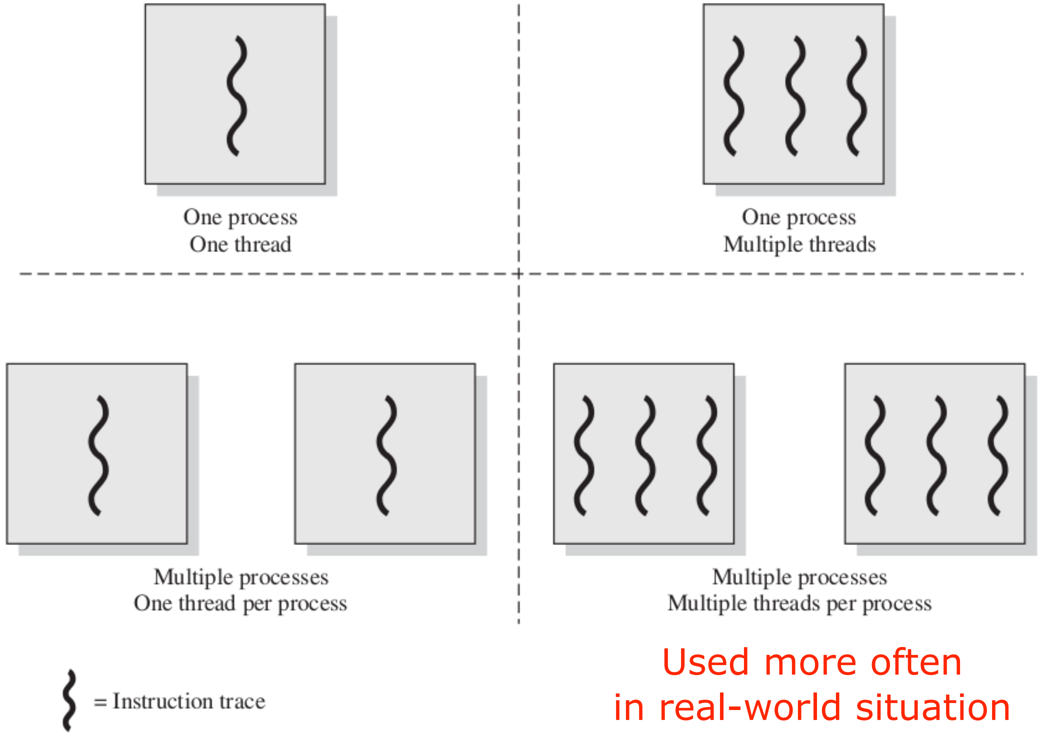
- What is **concurrency** and what are **OS** and **Process concerns** w.r.t to concurrency.
- We have been introduced to the notion of **process synchronization**, its need and the **critical section problem.**
- We have seen **requirements** for **solutions to the CS problem** i.e. mutual exclusion, progress & bounded waiting time.

Threads

- What is a thread?
- Parallelism vs. Concurrency
- Multicore and Multi-threading
- Multi-threading Models

From Processes to Threads

- Process model assumed a process was an executing program with a **single thread** of control
- All modern OS's provide features enabling a process to contain **multiple threads** of control



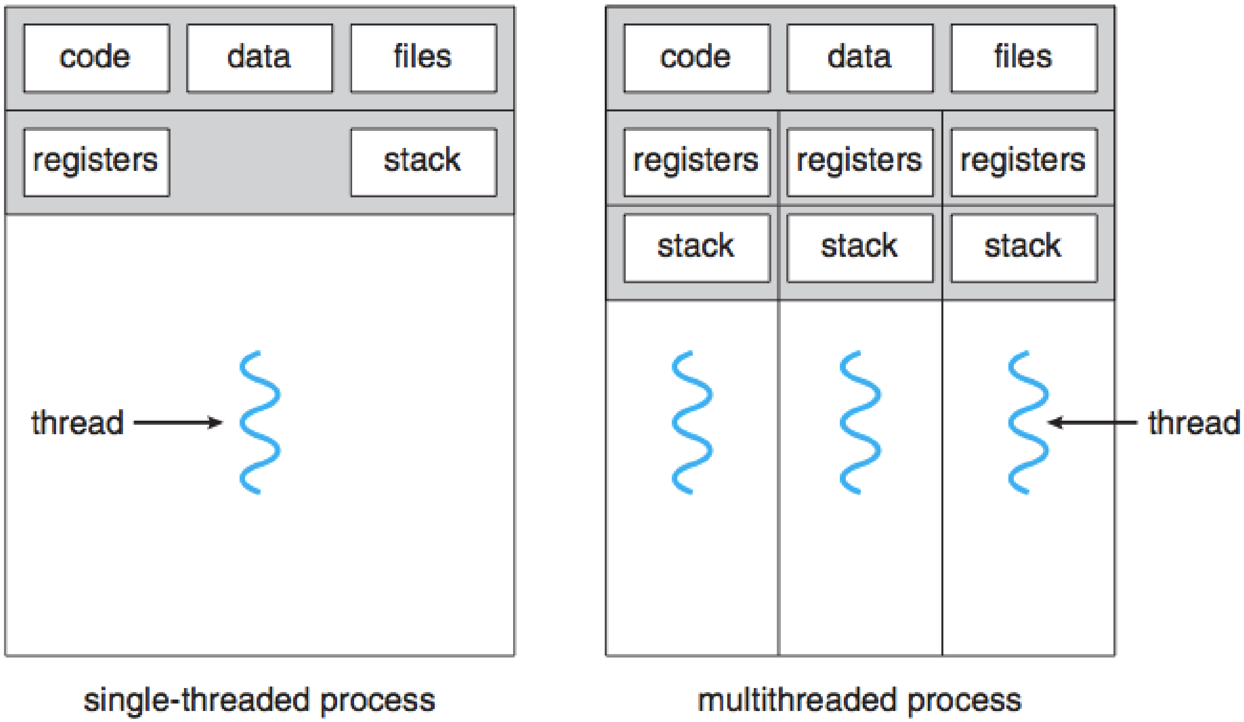
- **Multi-threading** is the ability of an OS to support multiple, concurrent paths of execution *within* a single process.

What is a Thread?

- A thread is part of a process
- Contains:
  - Thread ID (TID)
  - Program Counter (PC)
  - Register Set
  - Stack
- All threads *in the same process*, share:
  - Code section
  - Data section:
    - Objects
    - Open files
    - Network connections
    - Etc.

What is a Thread?

- If a process has multiple threads of control, it can perform more than one task at a time!



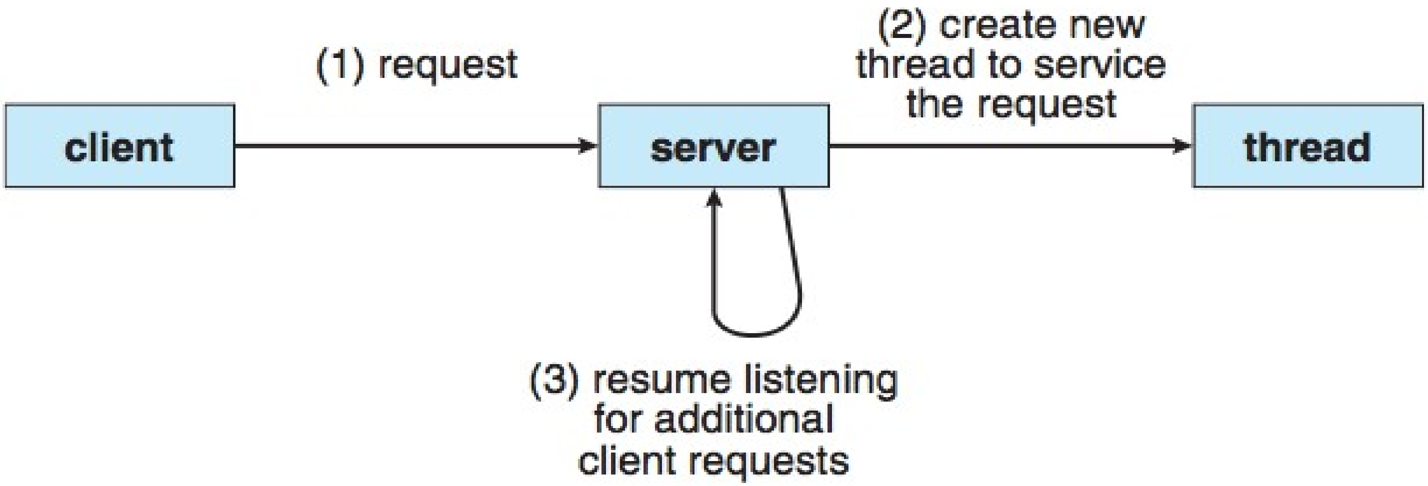
Why Threads?

Most applications *are* multi-threaded

- **Application:**
  - Process with several threads of control
  - Web browser:
    - Threads for each tab
    - Threads for each extension/helper
    - Utility threads
  - Program: *main* thread + service threads
    - + GUI threads
  - Server Processes

Multi-threaded Server Architecture

- Threads have a critical role in systems which provide services:
  - Servers: web servers, database servers ….
- When a server receives a message, that request is (usually) serviced using a separate thread:

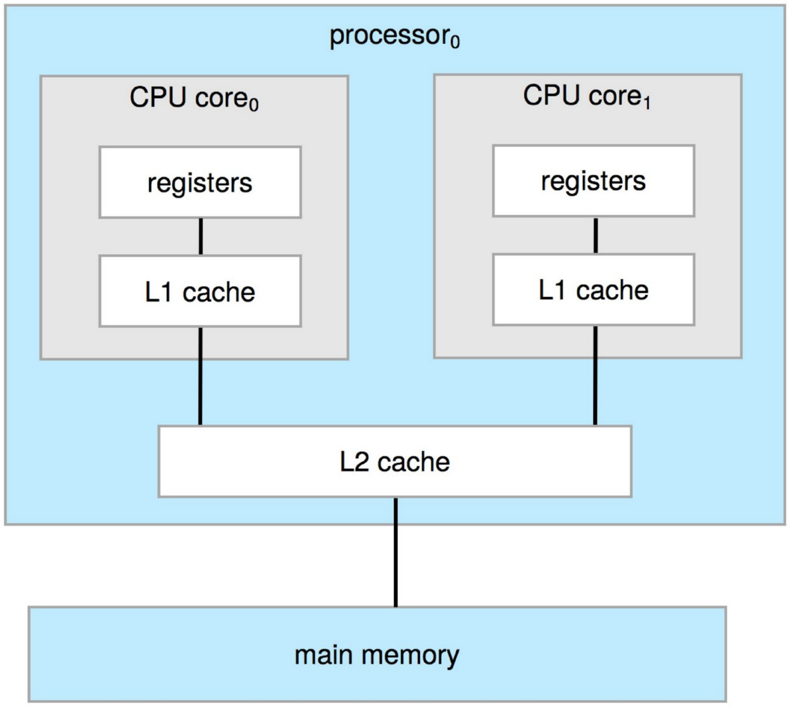


Benefits of Threads

- **Responsiveness:** user interaction in a GUI can be responded to by a separate thread from one running a computationally intense algorithm.
- **Resource Sharing:** Variables & objects can be shared. This is especially important for things like network or DB connections.
- **Economy:** “Light-Weight Processes” due to smaller memory footprint than using multiple processes & efficiency of switching between threads.
- **Scalability:** Easier to make use of parallel processing in a multi-threaded processor
- **Reduce programming complexity:** Problems often are better broken down into independent parallel tasks, rather than having to build complex mechanisms to interleave their execution.

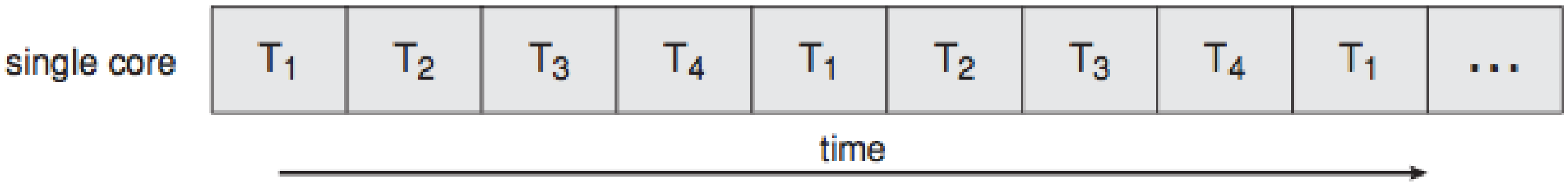
Multicore Programming

- Multiple processing elements on a single chip
- Multi-threaded programming:
    - Allows efficient use of the multiple processing elements and improved parallelism



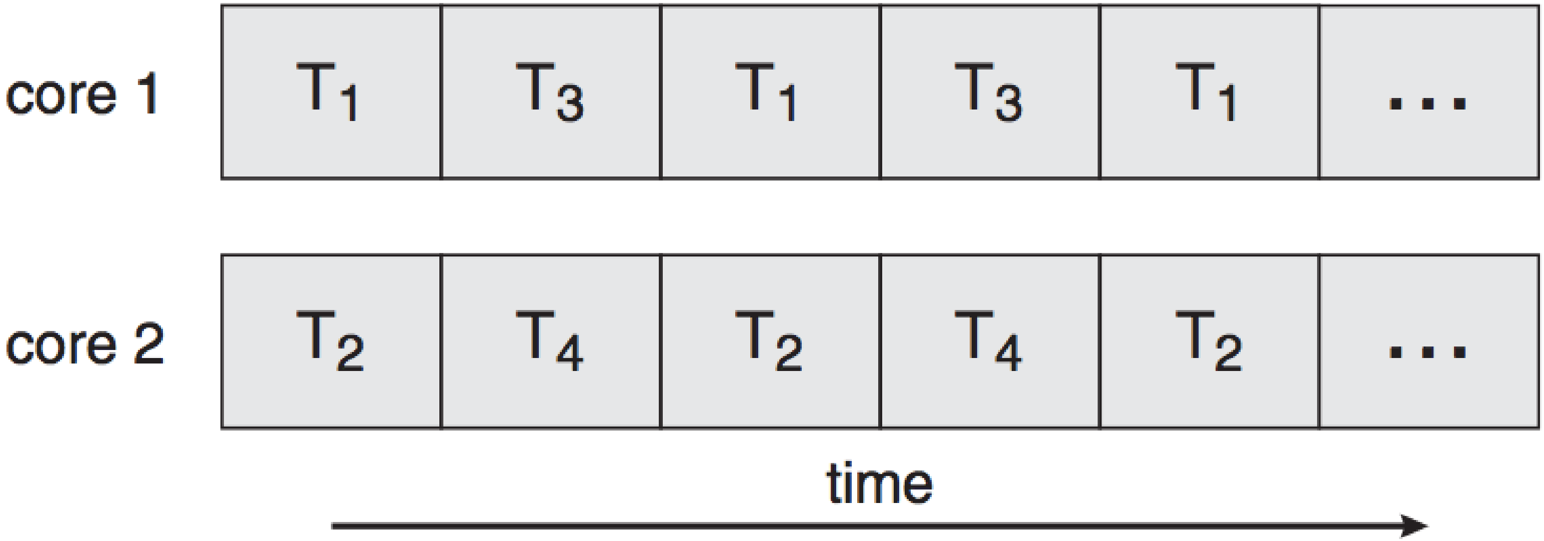
Concurrency

- On a single Processing element, **concurrency** means that the execution of threads will be interleaved over time.



- Only one thread at a time is actually running

Parallelism



Parallelism vs Concurrency

- A system is **parallel** 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.
- It is possible to have concurrency without parallelism

This is incredible!
I can do so much more!
There must be a catch ...

Multicore systems and Multithreading

- Performance of software on Multicore systems
- Amdahl's law states that:

$$\text{Speedup} = \frac{\text{time to execute program on a single processor}}{\text{time to execute program on } N \text{ parallel processors}} = \frac{1}{(1 - f) + \dfrac{f}{N}}$$

- **(1-f)** : Inherently serial code
- **f** : parallelizable code with no scheduling overhead

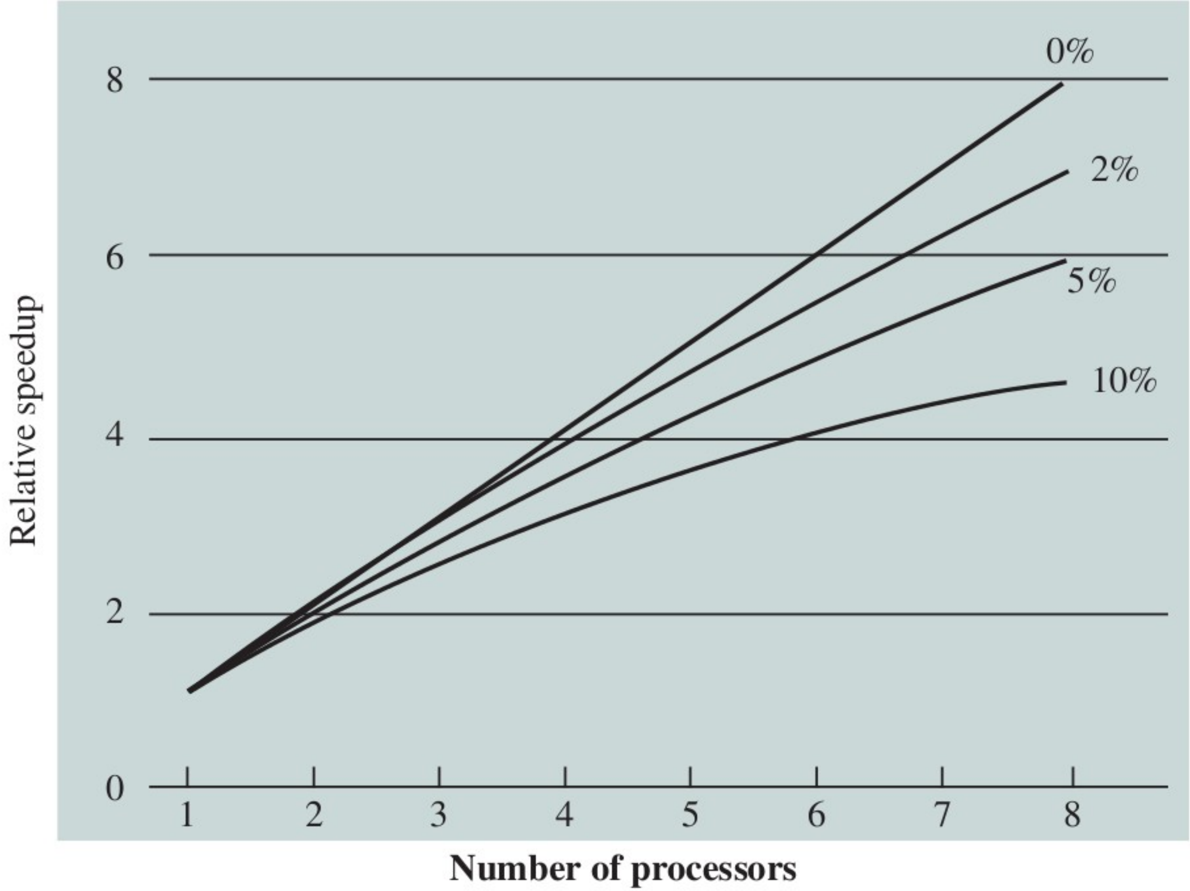
Multicore and Multithreading (cont'd)

- Amdahl's law makes the multicore organizations **look attractive!**
- But even a small amount of **serial code** has noticeable impact on the overall performance

  Example:
  10% serial, 90% parallel, 8 CPUs → ~4.7x speedup

- Also: There are overheads to parallelism:
  - Memory access
  - Cache load
  - Etc.

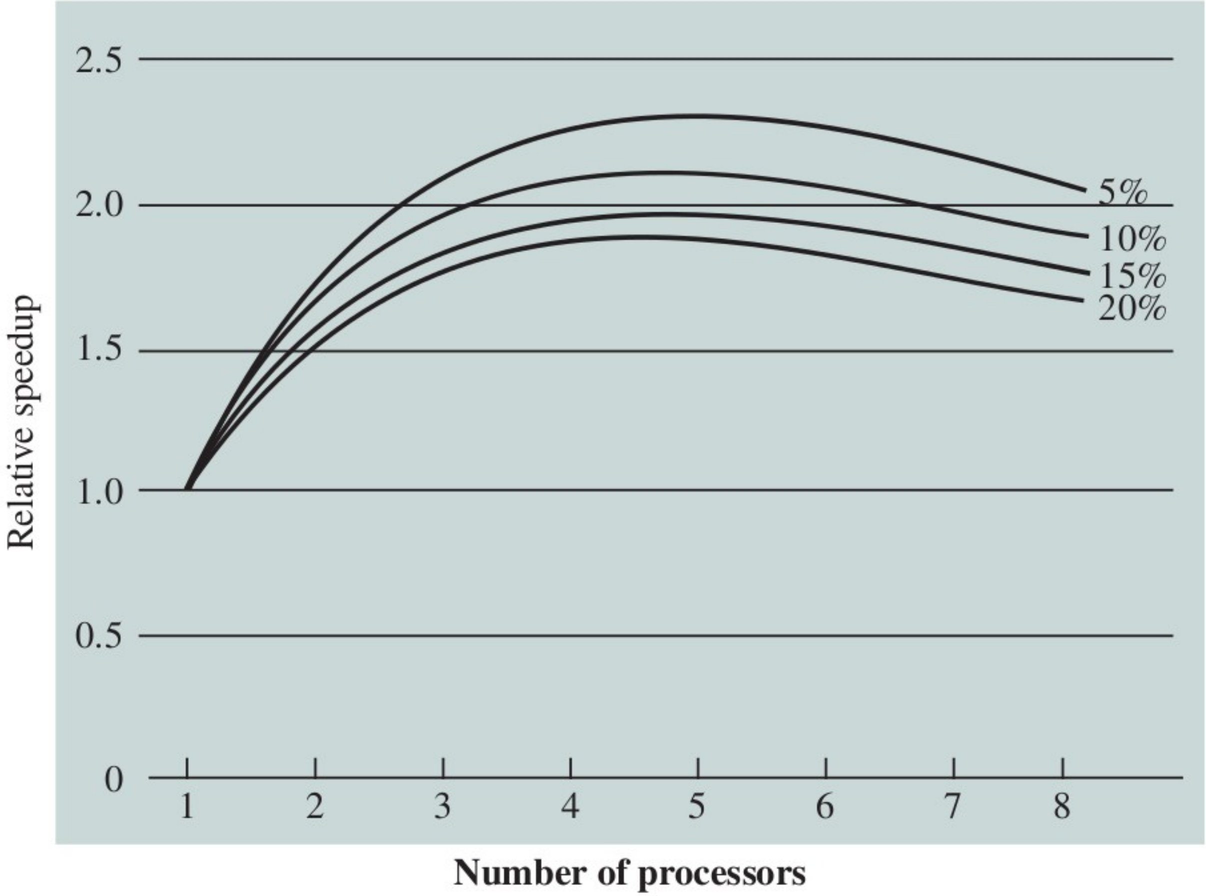
Multicore and Multi-threading (cont’d)



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

Continue from previous slide

Multicore and Multi-threading (cont’d)



(b) Speedup with overheads

Parallel Programming Challenges

- Pressure placed on system designers and application programmers to **better use multiple cores**
- System designers must write scheduling algorithms that use multiple processing core to allow **parallel execution**
- Challenge to **modify existing programs**
- Challenge to **design new programs** that are multithreaded

Parallel Programming Challenges (cont'd)

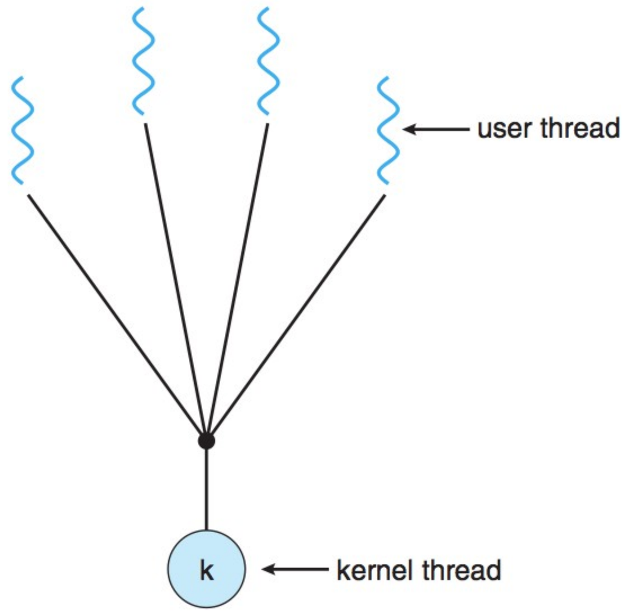
- **Identifying tasks:** which areas of an application can be divided into separate, concurrent (and ideally independent) tasks?
- **Balance:** how do we balance the tasks on the multiple cores to achieve maximum efficiency?
- **Data splitting:** how can data sets be split for processing in parallel?
- **Data dependency:** Certain tasks will rely on data from another. In this case, how do we synchronise the access to this data?
- **Testing and debugging:** Due to complexities of concurrent and parallel execution (multiple pathways etc.), how do we debug such an application?

Multithreading Models

- Threads may either be at the **user** or **kernel** thread level
- User threads managed above kernel
- Kernel threads are managed directly by the OS
- Three types of relationships between user and kernel threads are possible.
  - Many-to-One
  - One-to-One
  - Many-to-Many

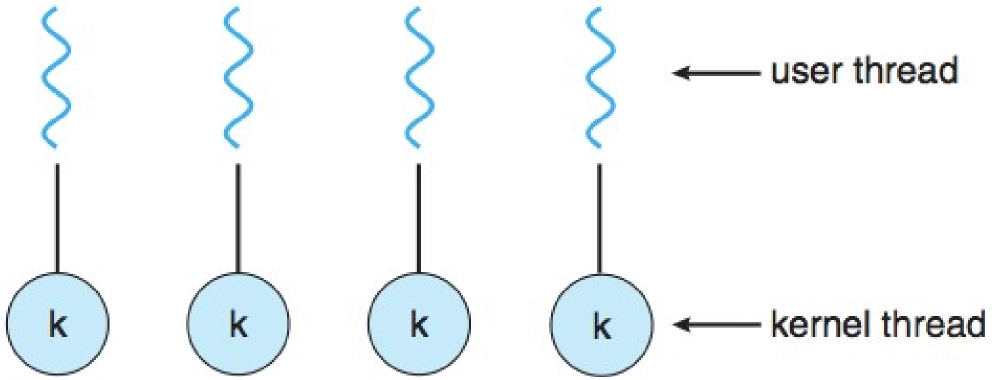
Many-to-One Model

- Maps **many user-level** threads to **one kernel** thread
- Thread management done by the **thread library**, in user space. Therefore, efficient!
- But, entire process will block if a thread makes a blocking system call
- Only one thread can access kernel at a time - lacks parallelism on multicore systems
- Not many systems use this due to inability to take advantage of multiple cores



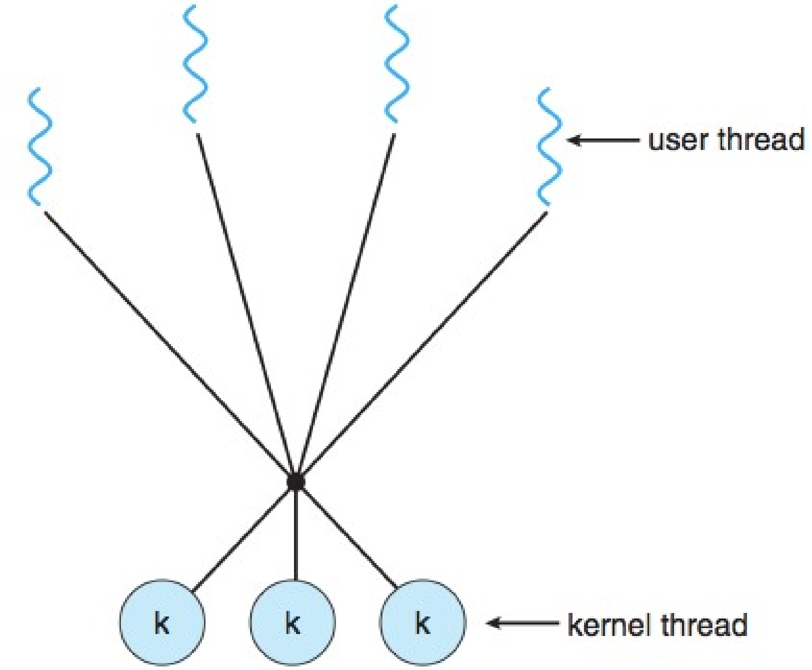
One-to-One Model

- Maps **each user thread** to **a kernel thread**
- Overcomes blocking issue - so more concurrent
- Allows for parallelism
- But, for each user thread, there has to be a kernel thread



Many-to-Many Model

- A software developer can create **as many user threads** as necessary i.e. a Thread Pool.
- Corresponding kernel threads can **run in parallel** on a multiprocessor
- If a thread blocks, kernel can schedule another for execution



How can we use Threads?

- Through use of a **thread library**
  - POSIX Pthreads (user/kernel level)
  - Windows (kernel)
  - Java Threads (on top of Windows/Pthreads)
  - Python Threads
    - (See https://realpython.com/intro-to-python-threading/#what-is-a-thread
- API for **creating** and **managing** threads

Inter-thread Communication

- Threads communicate by sharing data:
  - They are part of the same process
  - **Efficient!**
- We now have the same problems that we discussed earlier!
- Solution?
  - **Use tools (within your programming language or package) to manage thread synchronisation**

An Example

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

Why is there a problem?

- Functions seem simple and atomic

- But, to increment, there are three steps
  - Retrieve the current value of integer
  - Add 1 to this value
  - Store the incremented value back to the variable

Thread Interference

- If multiple threads all reference the same object:
  - **thread interference** can occur
- Happens when two functions, operating on the same data, **interleave**
- Unpredictable
- Difficult to debug

Managing Threads and Thread interaction

- These topics will be covered next term:
  - Python thread management
  - Data base transactions

  - These both provide effective, efficient and abstract ways to manage concurrent access to shared data

Summary

In this part, we have introduced:

- The concept of threads and multithreading.
- The differences between parallelism and concurrency
- The Amdahl's law for performance on multicore
- Multithreading Models
- Threading Issues including thread interference & memory consistency

References / Links

- Chapter # 4: Threads, Operating System Concepts (9th edition) by Silberschatz, Galvin & Gagne
- Chapter # 4: Threads, Operating System Internals and Design Principles (7th edition) by William Stallings
- https://realpython.com/intro-to-python-threading/#what-is-a-thread
- https://docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html
- http://tutorials.jenkov.com/java-concurrency/creating-and-starting-threads.html