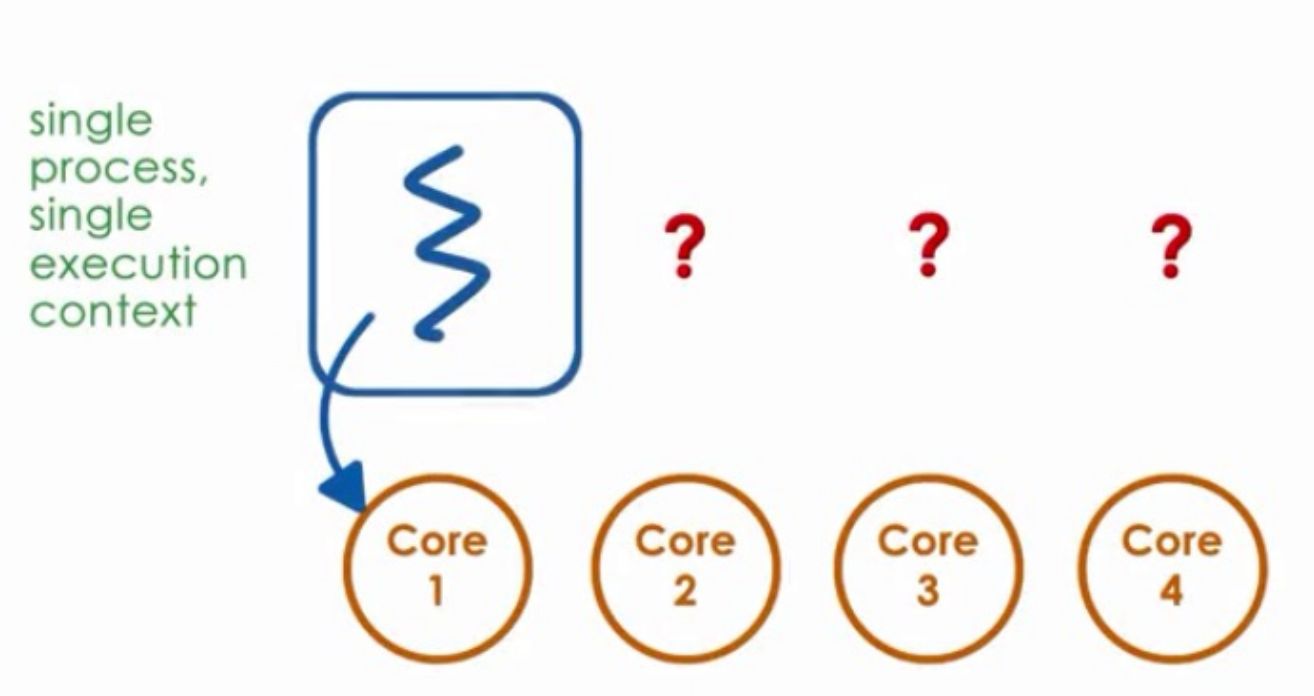
Threads and Concurrency

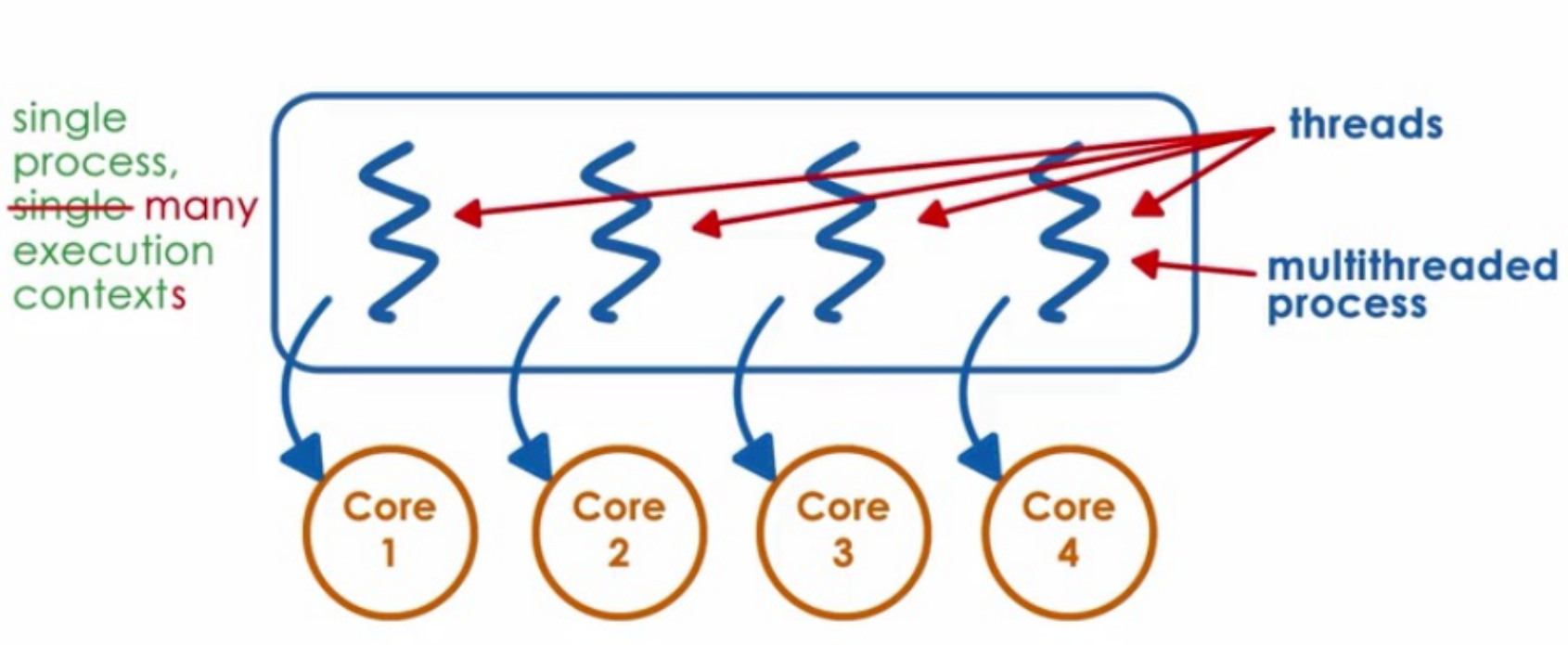
Pertemuan 5 dan 6

Lesson Overview



What if we have multiple CPUs?

Lesson Overview



Threads

- A thread is a **single sequential flow of execution of tasks of a process** so it is also known as **thread of execution** or thread of control (**execution context**). There is a way of thread execution inside the process of any operating system.
- A thread is a single sequence stream within a process. Threads are also called **lightweight processes** as they possess some of the properties of processes. Each thread belongs to exactly one process.

Benefits

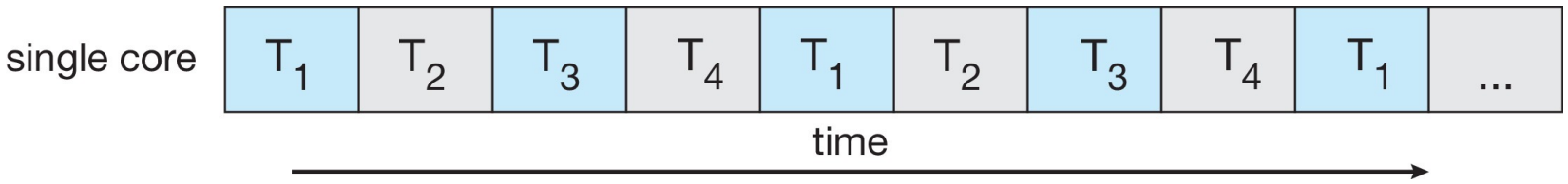
- **Responsiveness –** may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource Sharing –** threads share resources of process, easier than shared memory or message passing
- **Economy –** cheaper than process creation, thread switching lower overhead than context switching
- **Scalability –** process can take advantage of multicore architectures

Multicore Programming

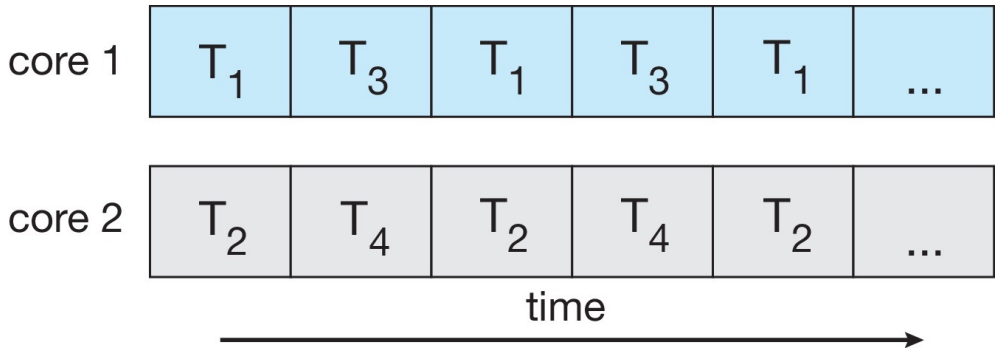
- **Multicore** or **multiprocessor** systems puts pressure on programmers, challenges include:
  - **Dividing activities**
  - **Balance**
  - **Data splitting**
  - **Data dependency**
  - **Testing and debugging**
- ***Parallelism*** implies **a system can perform more than one task simultaneously**
- ***Concurrency*** supports **more than one task making progress**
  - Single processor / core, scheduler providing concurrency

Concurrency vs. Parallelism

- **Concurrent execution on single-core system:**



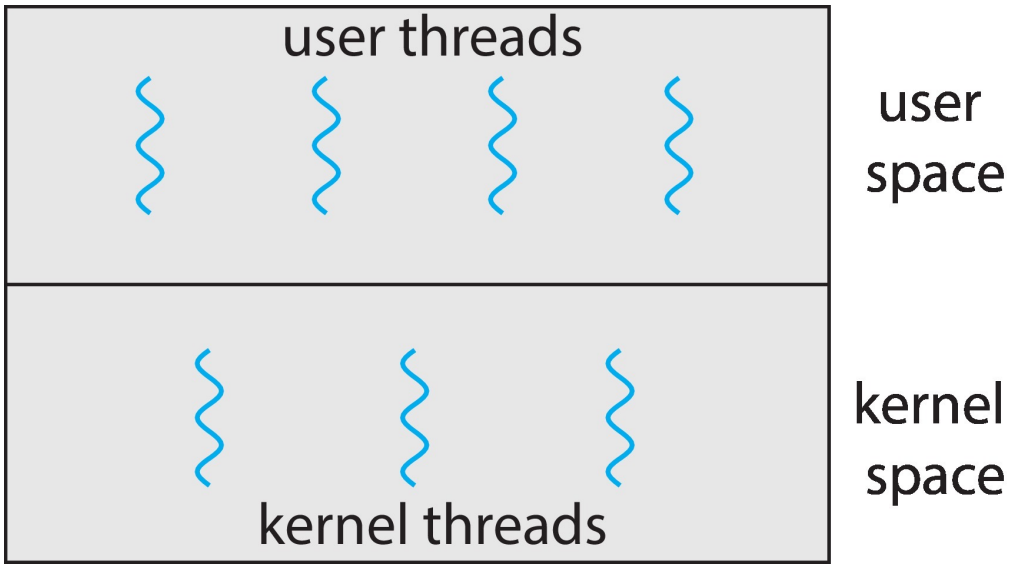
- **Parallelism on a multi-core system:**



User Threads and Kernel Threads

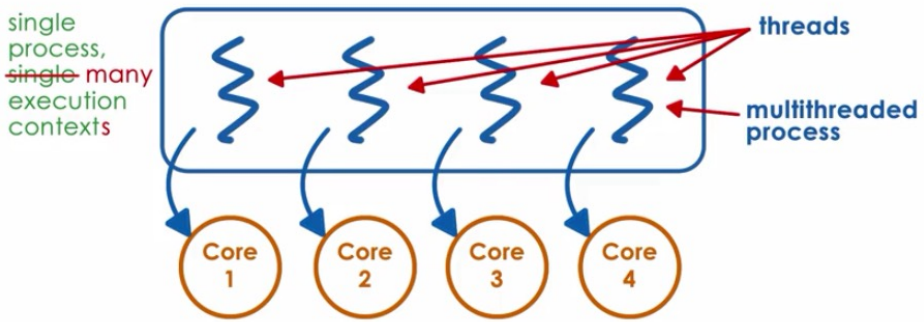
- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - POSIX **Pthreads**
  - Windows threads
  - Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general-purpose operating systems, including:
  - Windows, Linux, Mac OS X, iOS, Android

User and Kernel Threads

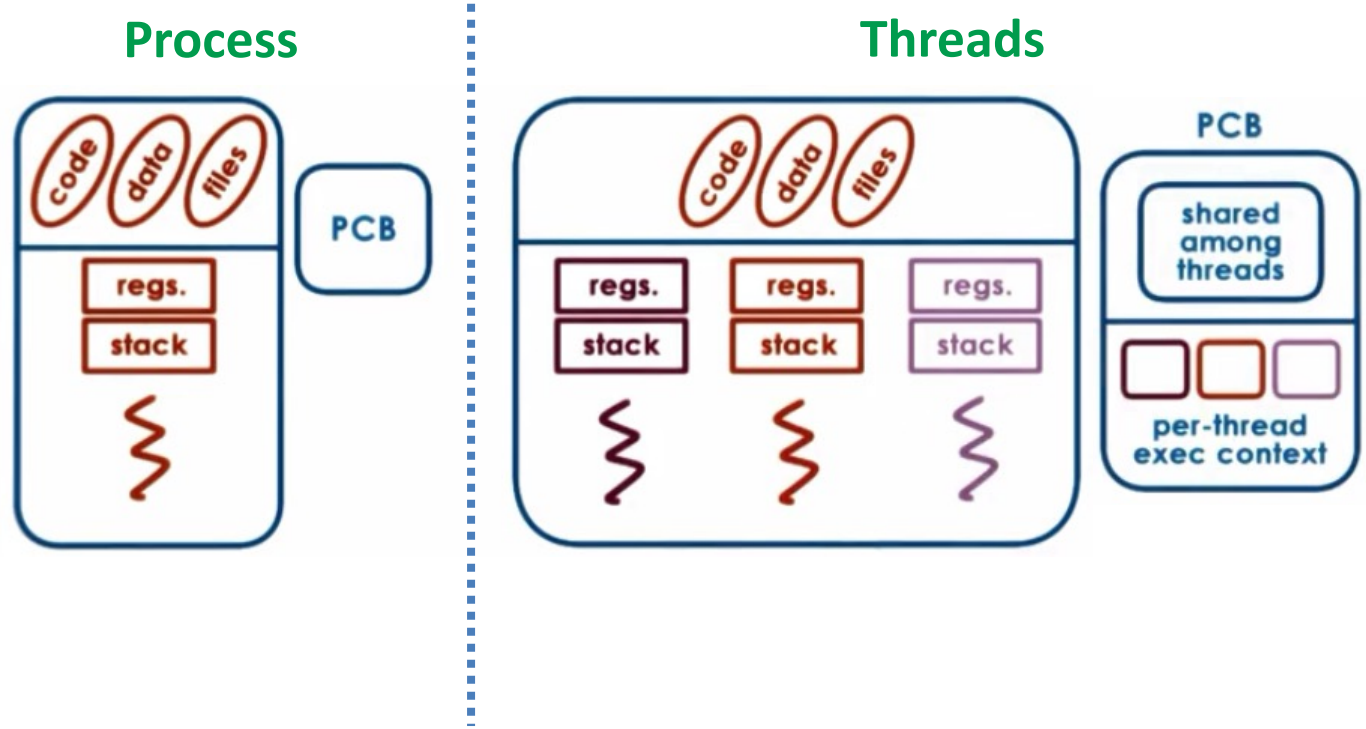


Multithreading

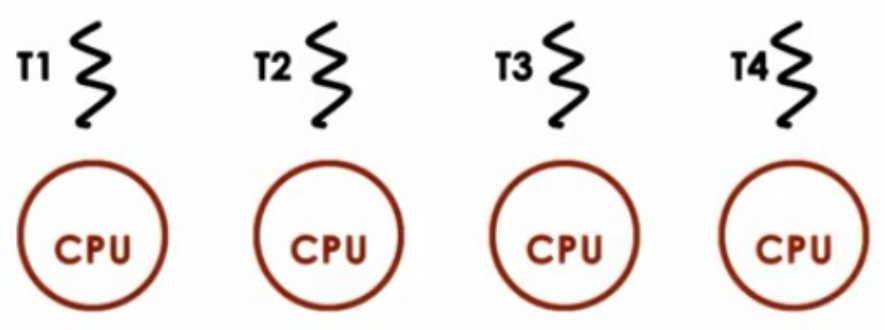
- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
- **Multithreading is the ability of a CPU to execute different portion** (or the exact same code but different subsets of the input) **of the same program at the same time**.

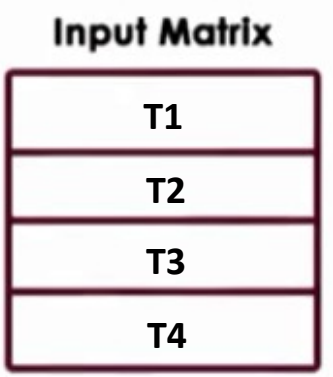


Process vs Thread



Benefits of Multithreading

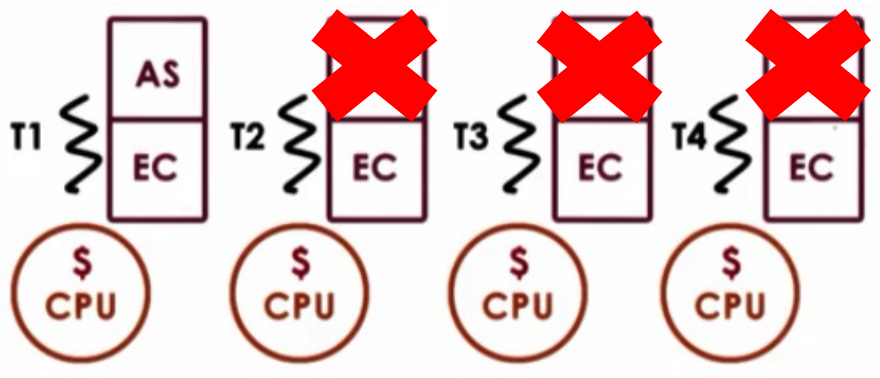


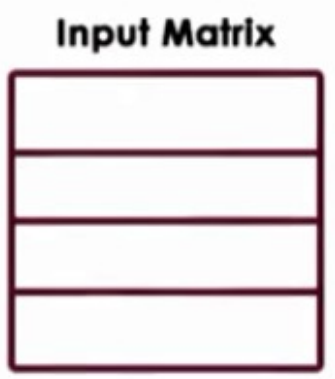


- Parallelization ➔ speed up : *we can process the input much faster than if only a single thread on a single CPU had to process the entire matrix.*
- Specialization ➔ hot cache : *differentiate how we manage those threads (e.g. give higher priority to those threads that handle more important tasks). End up executing with hotter cache or in other words gains performance.*

T1 : task 1
T2 : task 2
T3 : ……
T4 : ……

Benefits of Multithreading





- Parallelization ➔ speed up : *we can process the input much faster than if only a single thread on a single CPU had to process the entire matrix.*
- Specialization ➔ hot cache : *portioning or differentiate how we manage those threads (e.g. give higher priority to those threads that handle more important tasks). End up executing with hotter cache or in other words gains performance.*
- Efficiency ➔ lower memory requirement & cheaper IPC.

T1 : task 1
T2 : task 2
T3 : ……
T4 : ……

Basic Thread Mechanisms

What do we need to support threads?

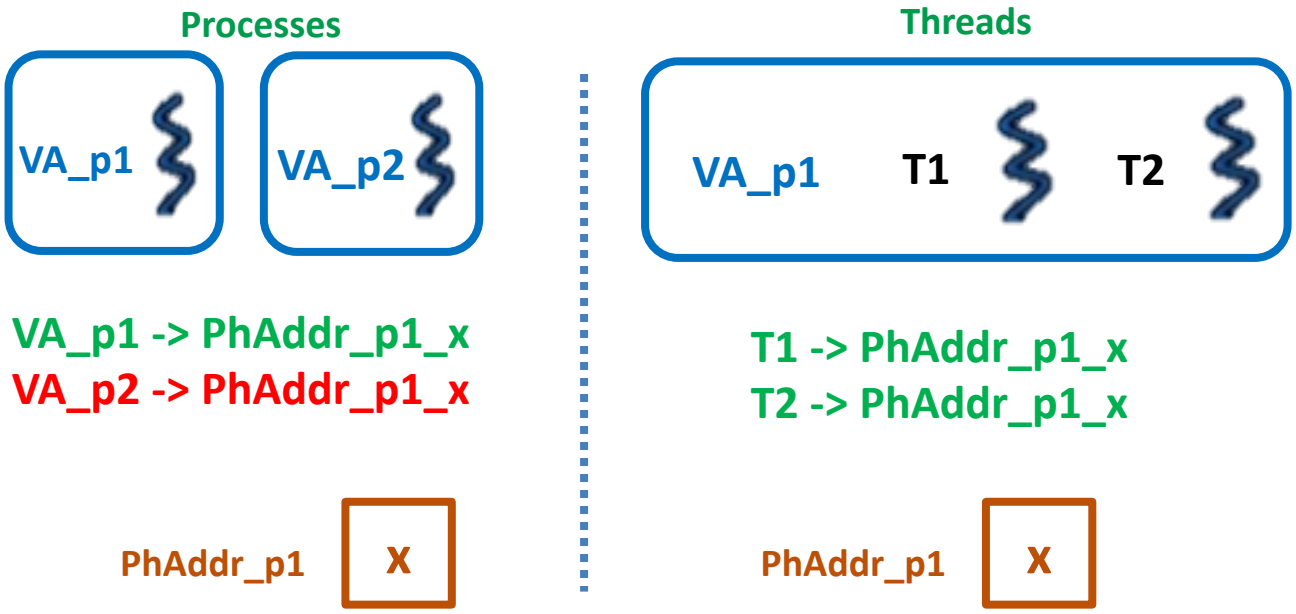
- Thread data structure to identify threads, keep track of resource usage
- Mechanisms to **create** and **manage** threads
- Mechanisms to safely coordinate among threads especially when there are certain dependencies between their execution when these threads are executing concurrently.

Basic Thread Mechanisms

Threads and Concurrency

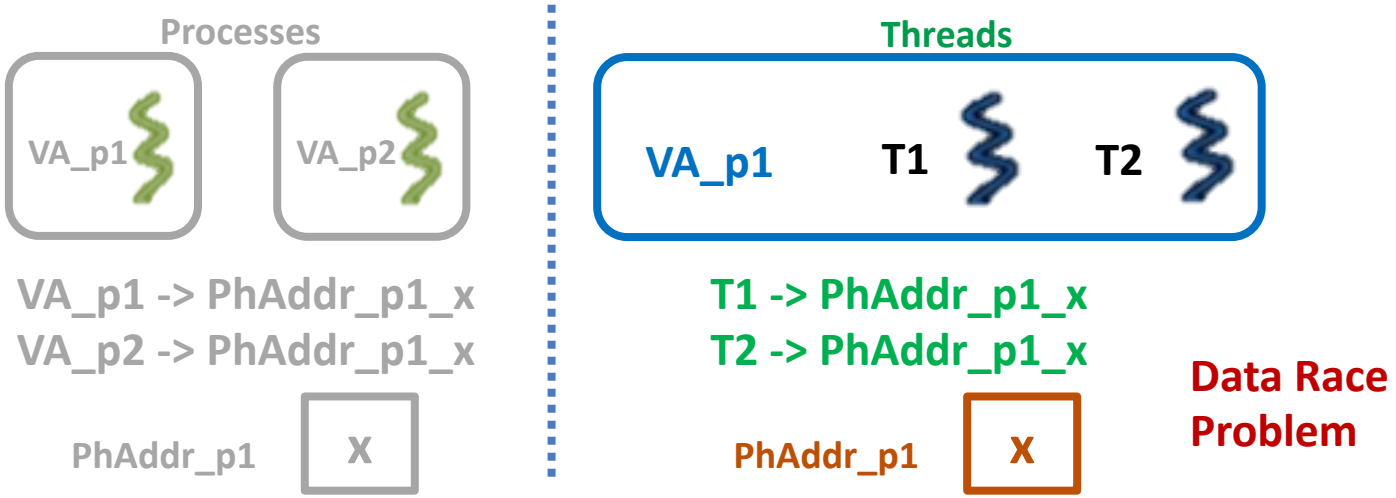
Issues associated with concurrent execution.

- When processes run concurrently: they each operate within their own address space.
- The OS together with the underlying hardware will make sure no access from one address space is allowed to perform on memory that belongs to the other address space.



Basic Thread Mechanisms

Threads and Concurrency



Issues associated with concurrent execution.

- **Problems**: if both T1 and T2 are allowed to access the data at the same time and modify at the same time, could end up with some inconsistencies.
  - *Example 1: one trying to read the data while the other is modifying it ---> just read the garbage*
  - *Example 2: both Threads are trying to update the data at the same time ---> update overlap*

Basic Thread Mechanisms

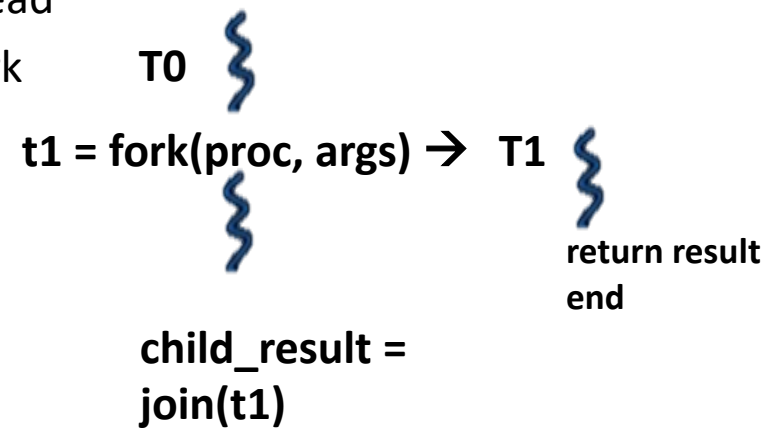
Concurrency Control & Coordination

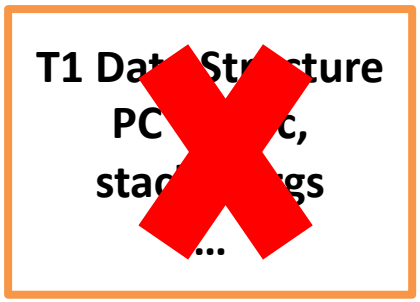
Synchronization Mechanisms:

- **Mutual Exclusion** mechanism:
  - Exclusive access to only one thread at a time.
  - **Mutex**
- **Waiting** on other threads
  - Specific condition before proceeding
    - Example: A thread that's dealing with shipment processing must wait on all the items in a certain order to be processed before that order can be shipped.
  - **Condition variable** to handle interthread coordination.
- Waking up other threads from wait state

Thread Creation

- **Thread type**:
  - Thread data structure: thread ID, PC, SP, registers, stack, attributes
- **Fork (proc, args)**
  - Create a thread
  - Not UNIX fork





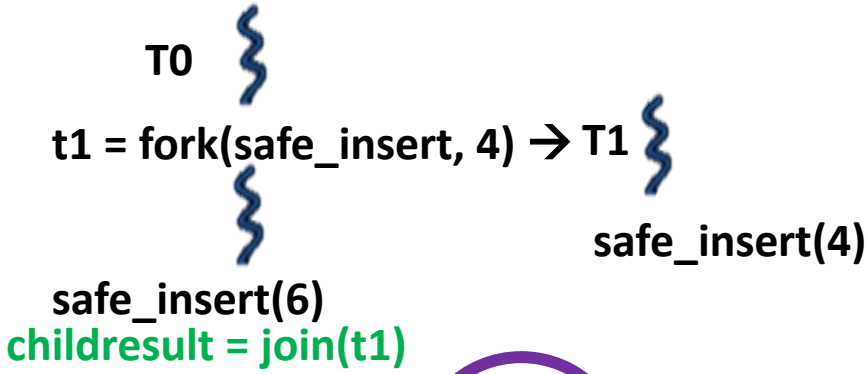
- **Join(thread)**
  - Terminate a thread

Thread Creation Example

```
Thread thread1;
Shared_list list;
thread1 = fork(safe_insert, 4);
safe_insert(6);
join(thread1); //optional
```

4 -> 6 -> null

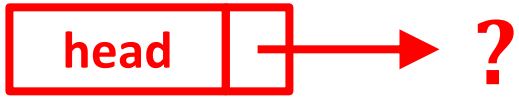
6 -> 4 -> null



How the list is updated

Can be accessed by reading shared variable

Problems:



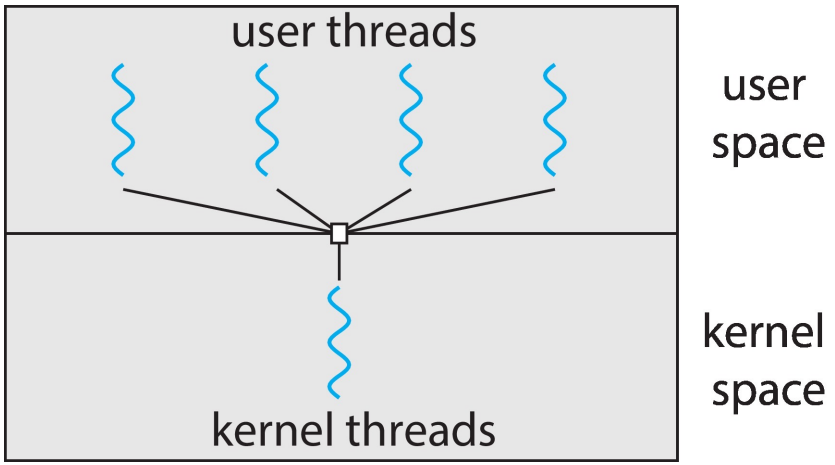

```
create new list element e
set e.value = X
read list and list.p_next
set e.pointer = list.p_next
set list.p_next = e
```

Multithreading Models

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

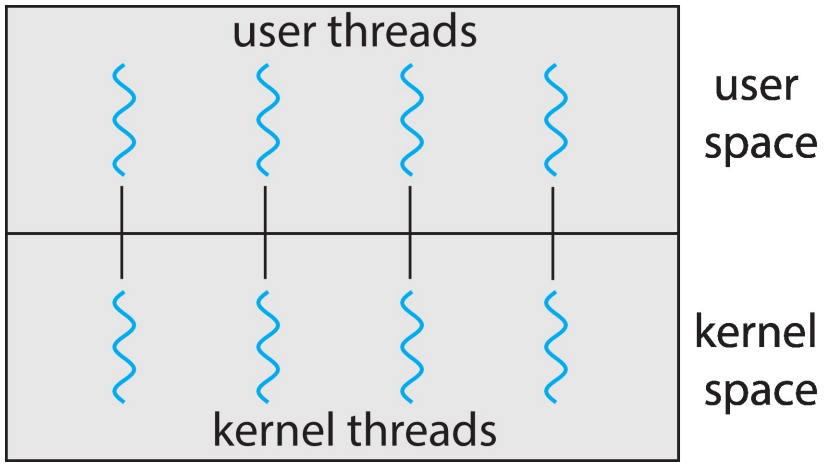
Many-to-One

- **Many user-level threads mapped to single kernel thread**
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples:
  - **Solaris Green Threads**
  - **GNU Portable Threads**



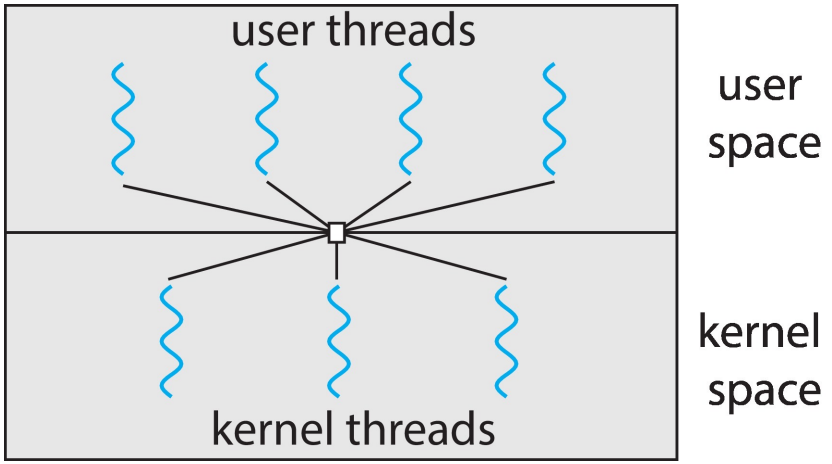
One-to-One

- **Each user-level thread maps to kernel thread**
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples
  - Windows
  - Linux



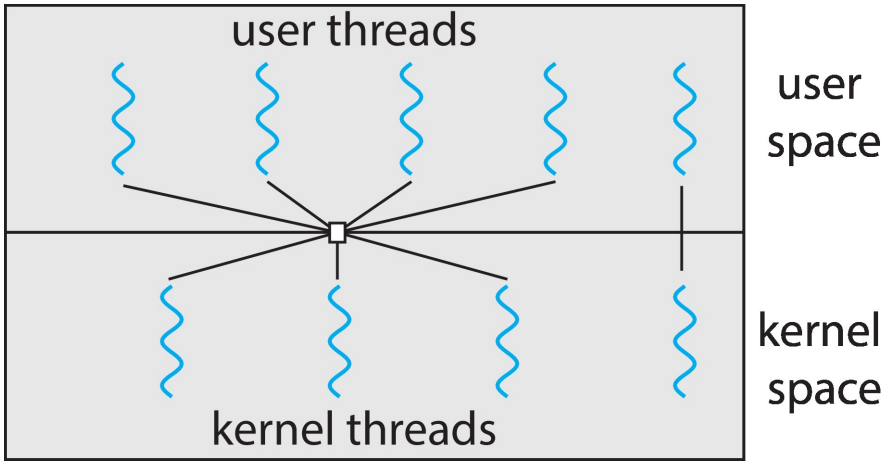
Many-to-Many Model

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Windows with the *ThreadFiber* package
- Otherwise not very common



Two-level Model

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



Thank You