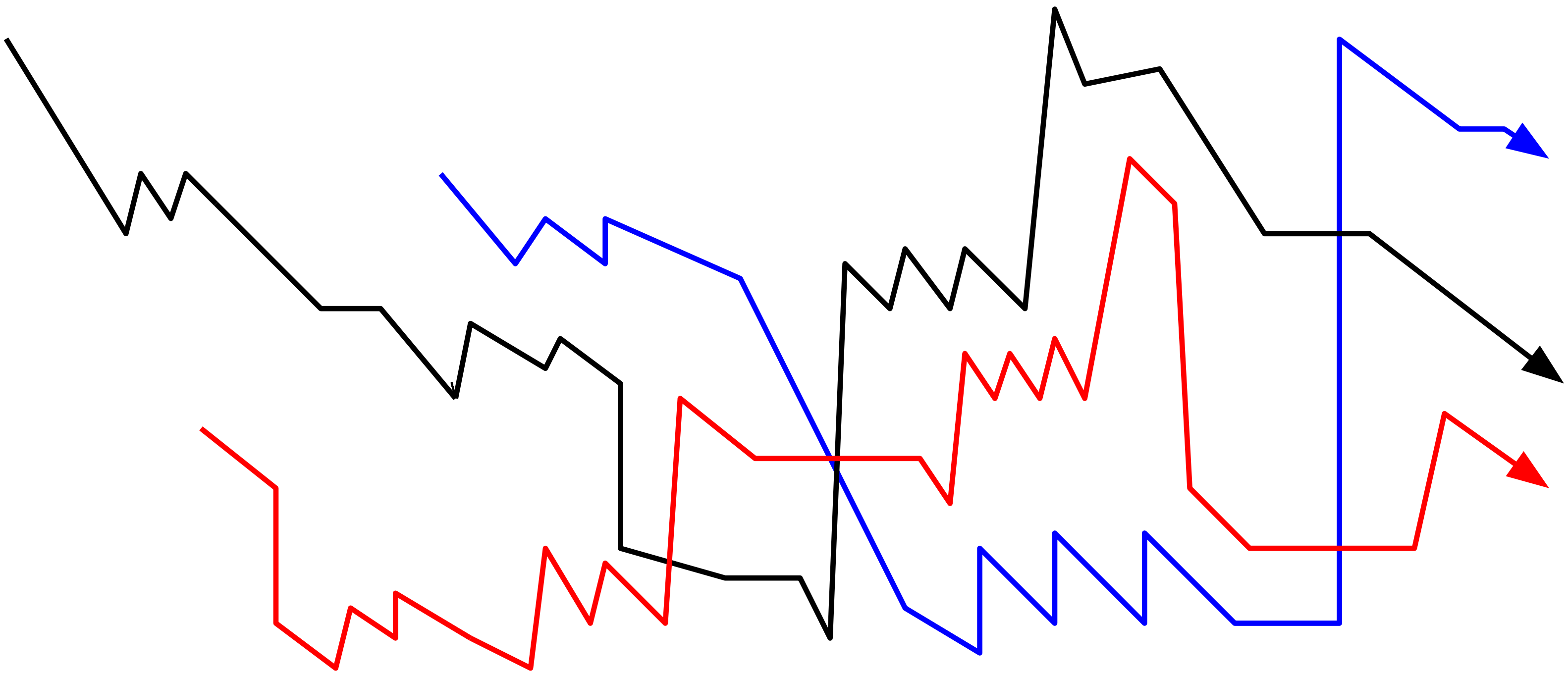
Ravi Mittal

IIT Goa

CS310 Operating Systems

Lecture 19: Need for Synchronization - Part 1: Introduction



Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - CS162, Operating System and Systems Programming, Profs. Natacha Crooks and Anthony D. Joseph, University of California, Berkeley
  - Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II, Chapter 5
  - Book: Modern Operating Systems, Fourth Edition, Andrew Tenenbaum, Herbert Bos, Pearson Publication
    - Chapter 2.3

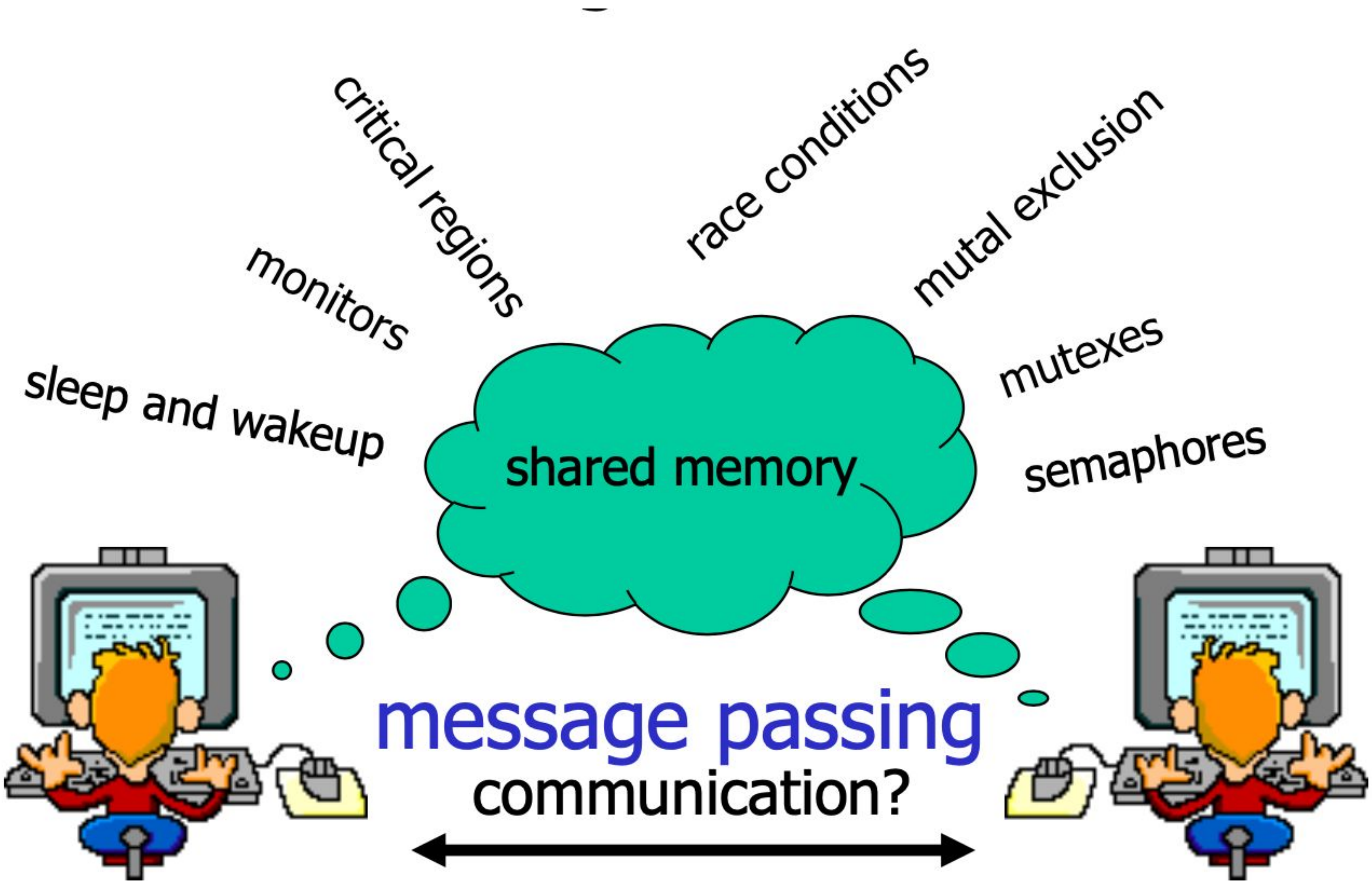
Reading

- Book: Operating Systems: Principles and Practice: Thomas Anderson and Michael Dahlin, Volume II, Chapter 5
- Book: Modern Operating Systems, Fourth Edition, Andrew Tenenbaum, Herbert Bos, Pearson Publication
  - Chapter 2.3

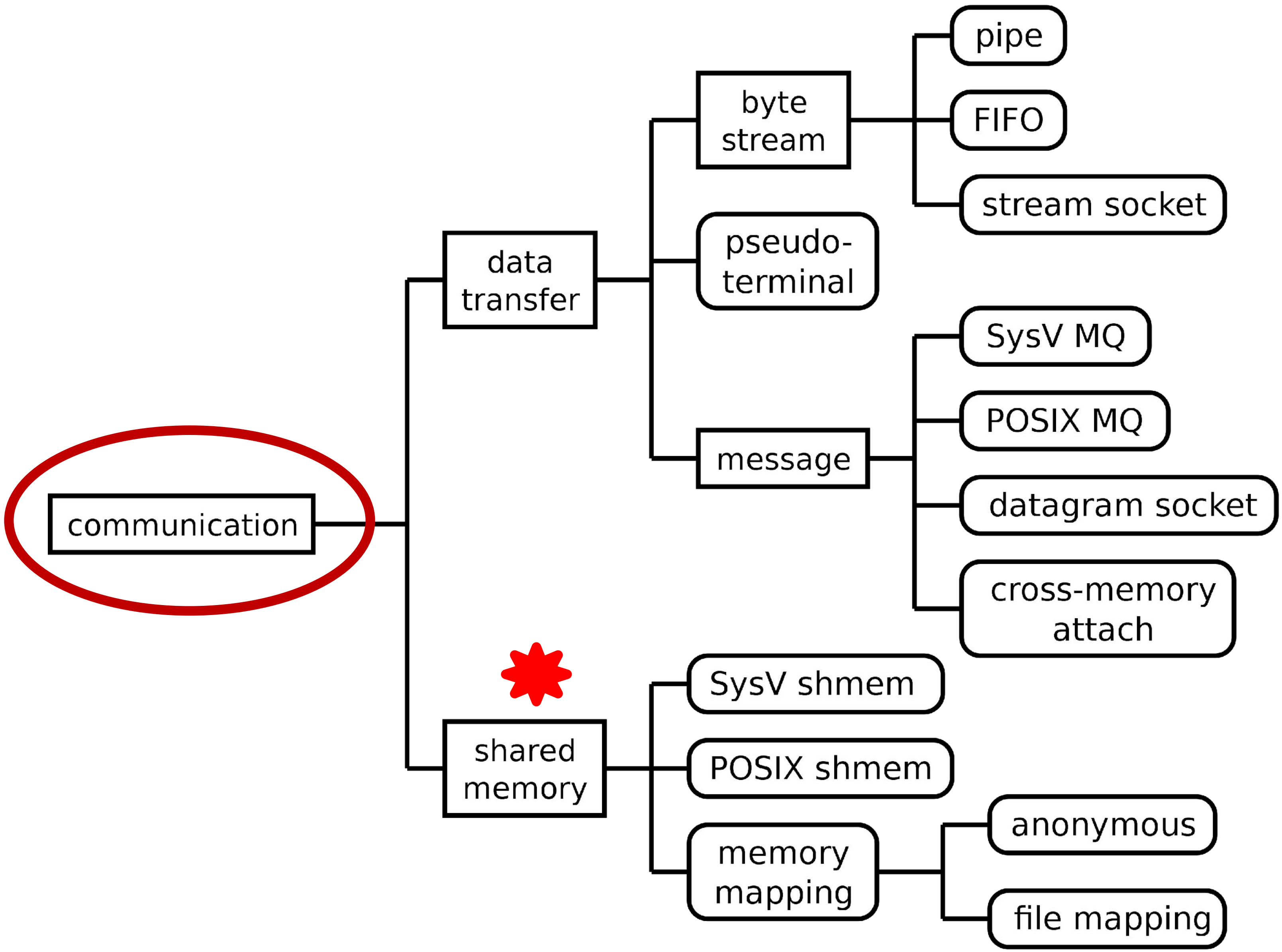
So far we have studied

- Threads
- Processes
- Concurrent execution of Threads and Processes require
  - Communication
  - Synchronization
- Inter-process Communication methods
  - Message Passing
    - Message Queues
    - Pipes
    - Named Pipes or FIFO
  - Shared Memory

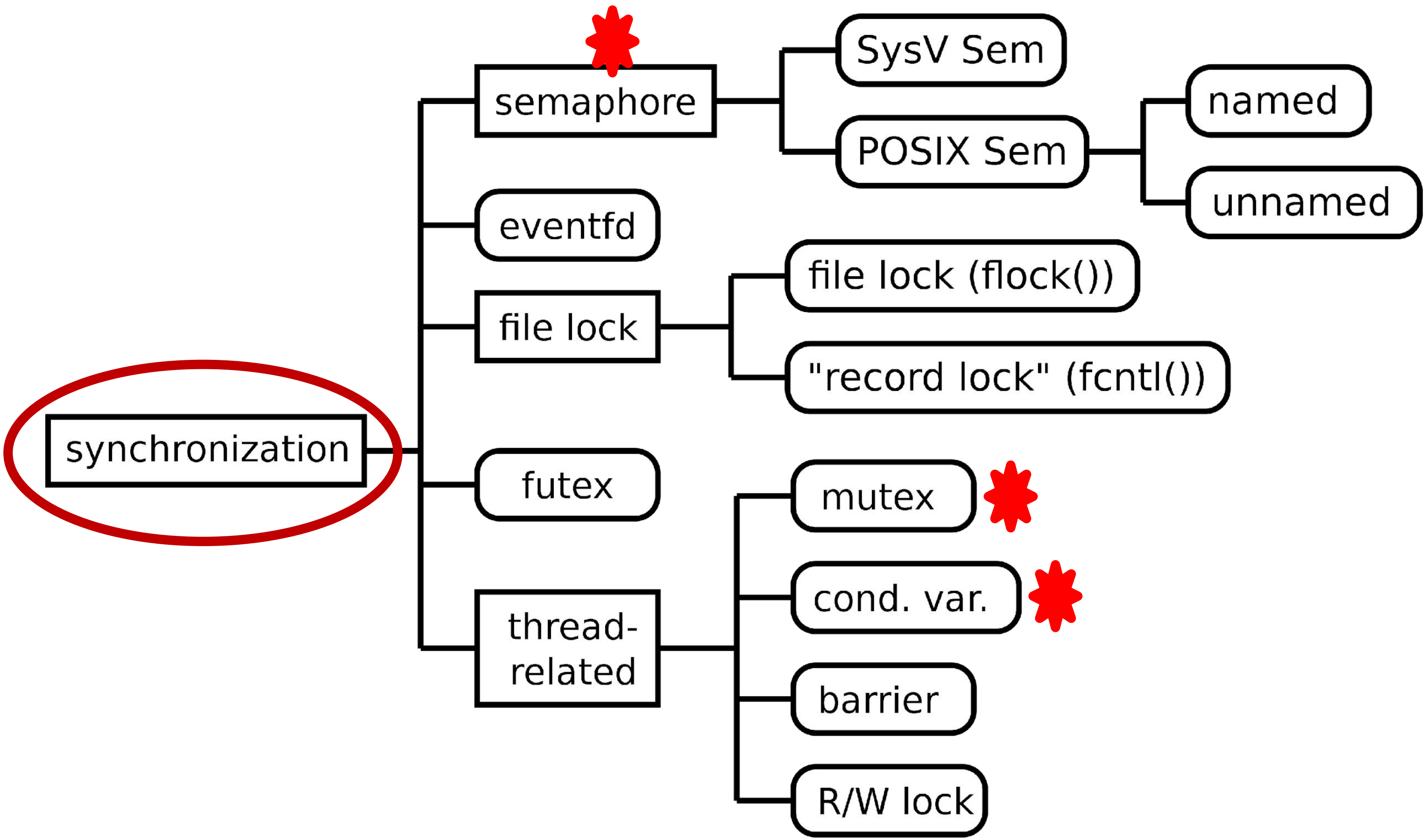
IPC – Big Picture



Communication



Synchronization



We will start with High level primitives

Programs	Shared Programs
Higher-lev el API	Locks Semaphores Monitors Others
Hardware	Disable Ints Test&Set Compare&Swap, others

- Our focus will be on concepts

Top level View of Synchronization

Programs	Shared Programs
Higher-level API	Locks Semaphores Monitors Send/Receive
Hardware	Disable Ints Test&Set Compare&Swap

- We are going to implement various higher-level synchronization primitives using atomic operations
- Need to provide primitives useful at user-level

Today we will study ..

- Race condition in concurrent Processes
- Race condition in Concurrent threads

Concurrent Execution

- Concurrent Execution of Programs or Threads may lead to race conditions

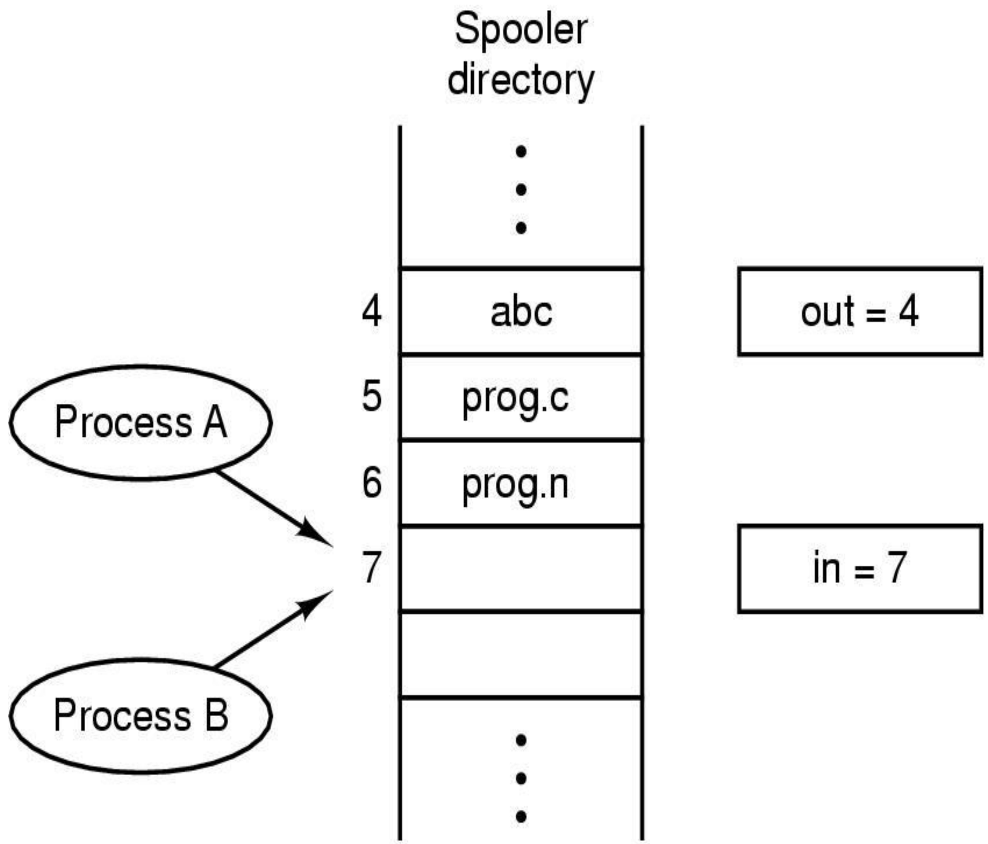
Race Condition in Concurrent Processes

Example 1: Printer Spooler – Processes (1/2)

- Two processes A and B, are trying to use a print spooler
- When a process wants to print a file, it enters the file name in a special special spooler directory
- Another process Printer Daemon periodically checks if there is any file to be printed. If so, it prints the file and removes the file's name from spooler directory
- Imagine that there are two shared variables: `in` and `out`
  - `In` points to the next free slot in the director
  - `Out` points to the next file to be printed
- As shown in the figure slots 4, 5, 6 are occupied; so `out` = 4 and `in` = 7
- Process A reads `in` = 7 and stores it in local variable *next_free_slot* ; Now process A is context switched

Example 1: Printer Spooler – Processes (2/2)

- Now, process B reads `in` = 7 and it stores it in local variable *next_free_slot*
- Process B stores the file name to be printed, into slot 7 and updates `in` = 8; Process is now context switched
- Process A comes back and stores the file to be printed in slot 7 and updates `in` = 8
- Process B will never receive any output



Race Condition in Threads

A simple piece of code

```
unsigned counter = 0;

void *do_stuff(void * arg) {
  for (int i = 0 ; i < 200000000 ; ++ i) {
    counter ++;
  }
  return arg;
}
```

adds one to counter

How long does this program take?

How can we make it faster?

A simple piece of code

```
unsigned counter = 0;

void *do_stuff(void * arg) {
   for (int i = 0 ; i < 200000000 ; ++ i) {
      counter ++;
   }
   return arg;
}
```

adds one to counter

How long does this program take? Time for 200000000 iterations

How can we make it faster? Run iterations in *parallel*

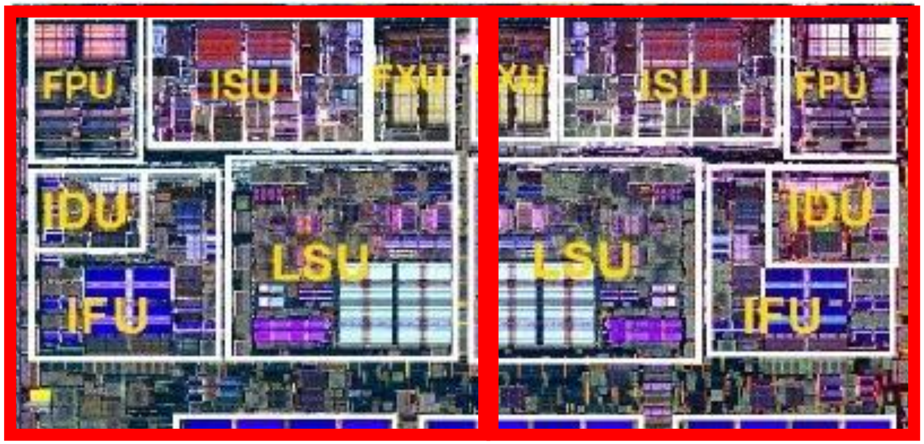
Exploiting a multi-core processor

Concurrently run this on multiple threads running on separate cores

```
unsigned counter = 0;

void *do_stuff(void * arg) {
   for (int i = 0 ; i < 200000000 ; ++ i) {
      counter ++;
   }
   return arg;
}
```

#1

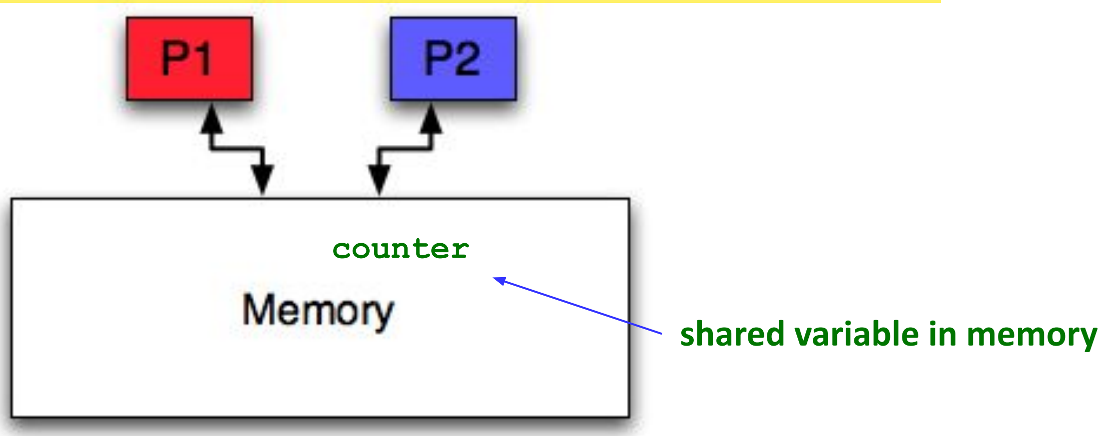


#2

What is the speedup?

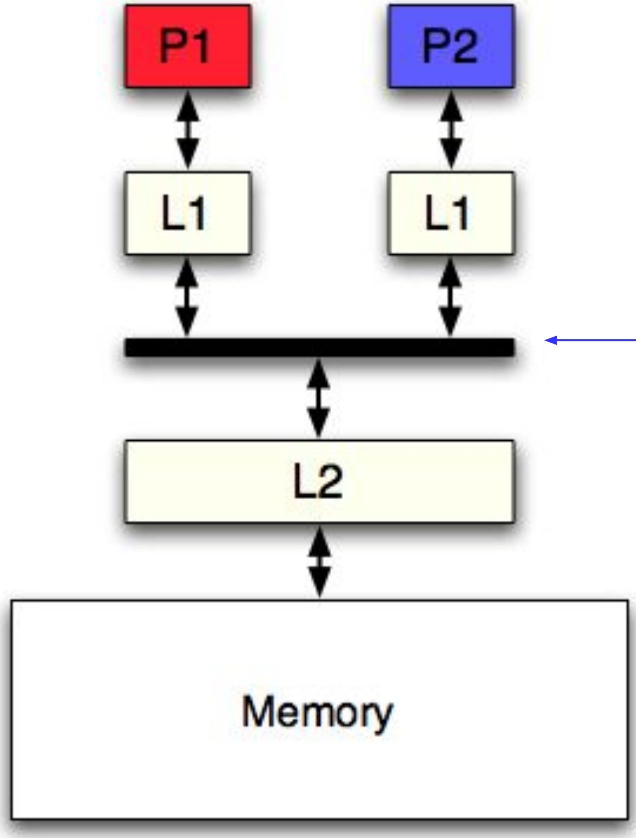
How much faster?

- We're expecting a speedup of 2
- OK, perhaps a little less because of Amdahl's Law
  - Overhead for forking and joining multiple threads
- But its actually slower!! Why??
- Here's the mental picture that we have – two processors, shared memory



This mental picture is wrong!

- We've forgotten about caches!
  - The memory may be shared, but each processor has its own L1 cache
  - As each processor updates `counter`, it bounces between L1 caches



Multiple bouncing slows performance

The code is not only slow, its WRONG!

- Since the variable `counter` is *shared*, are we getting in to race condition?
- Look back at the code !
- Two threads are independently incrementing counter value
  - Counter value is increasing as expected ?
- Then, where is the issue ?

The code is not only slow, its WRONG!

- Increment operation: `counter++` MIPS equivalent:

```
lw   $t0, counter
addi $t0, $t0, 1
sw   $t0, counter
```

Sequence 1

Processor 1	Processor 2
`lw   $t0, counter`	
`addi $t0, $t0, 1`	
`sw   $t0, counter`	
	`lw   $t0, counter`
	`addi $t0, $t0, 1`
	`sw   $t0, counter`

Sequence 2

Processor 1	Processor 2
`lw   $t0, counter`	
	`lw   $t0, counter`
`addi $t0, $t0, 1`	
	`addi $t0, $t0, 1`
`sw   $t0, counter`	
	`sw   $t0, counter`

`counter` increases by 2

`counter` increases by 1 !!

What is the minimum value of counter at the end of the execution?

```
unsigned counter = 0;

void *do_stuff(void * arg) {
  for (int i = 0 ; i < 200000000 ; ++ i) {
     counter ++;
  }
  return arg;
}
```

adds one to counter

What is the minimum value at the end of the program?

Thread 1	Thread 2
lw to, counter	
addi to, to + 1 //counter = 0	
	lw, to, counter // counter = 0
	addi, to, 1 //
	sw to, counter // counter = 1
	1M – 1 times counter = 1M-1
t0 = 1	
sw to, counter // counter = 1	
	lw to, counter // to = 1
	addi to, 1 //to = 2
lw, to, counter // counter = 1	
addi, to, 1	
sw to, counter // counter =	
1M – 1 times counter = 1M	
	sw to, counter //counter =2

Atomic operations

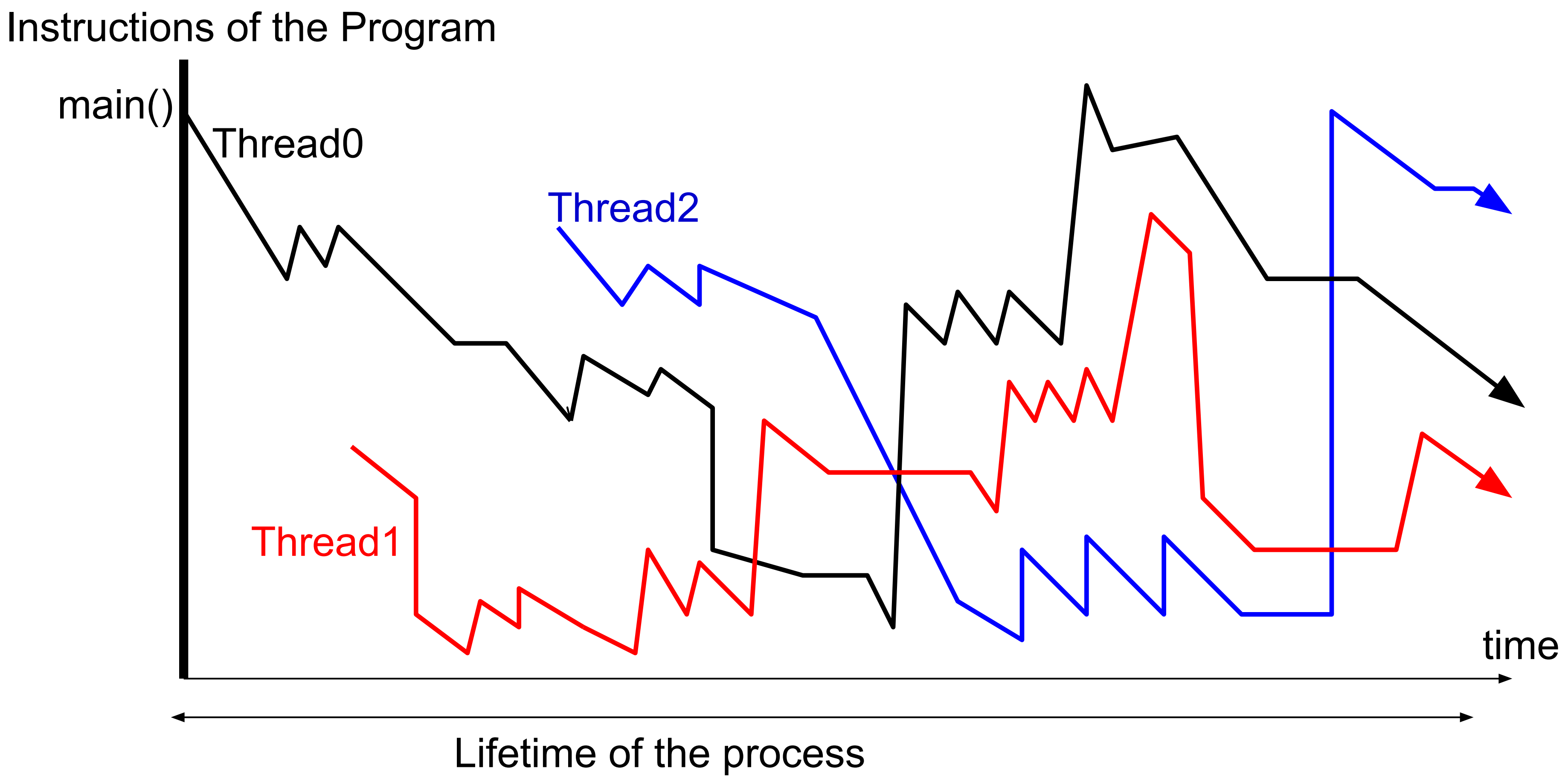
- You can show that if the sequence is particularly nasty, the final value of `counter` may be as little as 2, instead of 200000000
- To fix this, we must do the load-add-store in a *single* step
  - We call this an atomic operation
  - We're saying: Do this, and don't get interrupted while doing this
- Atomic in this context means all or nothing
  - Either we succeed in completing the operation with no interruptions or we fail to even begin the operation
    - Fail because someone else is doing atomic operation

Atomic Operations (Repeat)

- Indivisible operations that cannot be interleaved with or split by other operations

- The problem in the above example happened due to
  - `add` operation is not an atomic operation
  - So interleaving of assembly instructions caused race condition

A multithreaded process' execution flows: threads



Achieving correctness with Concurrent Threads

- Non-determinism
  - Scheduler can run threads in any order
  - Scheduler can switch threads at any time
  - Non-determinism can make testing very difficult
- Independent Threads
  - No state shared with other threads
  - Deterministic, reproducible conditions
- Cooperating Threads
  - Shared state between multiple threads

- **Goal: Correctness by Design**

Relevant Definitions

- Synchronization: Coordination among threads/processes, usually regarding shared data
- Mutual Exclusion: Ensuring only one thread/Process does a particular thing at a time (one thread/Process *excludes* the others)
  - Type of synchronization
- Critical Section: Code exactly one thread/process can execute at once
  - Result of mutual exclusion
- Lock: An object only one thread/process can hold at a time
  - Provides mutual exclusion

Multicore programming and multithreading challenges

- Programming in multicore processors is difficult
  - Threading can utilize Multicore systems better, but it has to overcome challenges
- Threading Challenges include
  - **Dividing activities**
    - Come up with concurrent tasks
  - **Balance**
    - Tasks should be similar importance and load
  - **Data splitting**
    - Data may need to be split as well
  - **Data dependency**
    - Data dependencies should be considered; need synchronization of activities
  - **Testing and debugging**
    - Debugging is more difficult

Mutual Exclusion: Token exchange system of Railways on single track

Only one train on the track can have the token

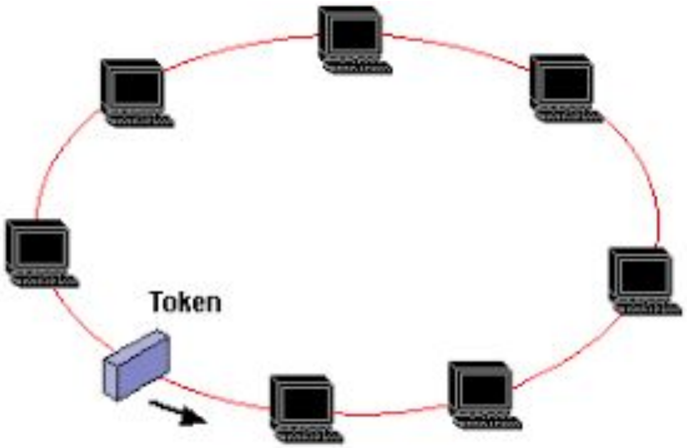
Lecture Summary

- Concurrent execution of processes /Threads is important activity for performance improvements
- However, programming utilizing concurrency needs to be done carefully due to race condition
- We have looked at examples where race conditions give wrong results
- In the next class we will look at another example

backup

Examples: Mutual Exclusion

Token Ring Network



Locks

- Lock - acquire
  - wait until lock is free, then take it
  - Atomically make the lock busy
    - Checking the state to see if it is FREE and setting the state to BUSY are together an atomic operation
  - Even if multiple threads try to acquire the lock, at most one thread will succeed

- Lock - release
  - release lock, waking up anyone waiting for it

Note:

1. At most one lock holder at a time (safety)
2. If no one holding, acquire gets lock (progress)
3. If all lock holders finish and no higher priority waiters, waiter eventually gets lock (progress)

Rules for Using Locks

- Lock is initially free
- Always acquire before accessing shared data structure
  - Beginning of procedure!
- Always release after finishing with shared data
  - End of procedure!
  - Only the lock holder can release
  - DO NOT throw lock for someone else to release
- Never access shared data without lock
  - Danger!

Lock - Examples

- Locking to group multiple operations
  - Bank Transactions – account modification, account data reading etc …
  - Acquire Lock – do transactions – release lock
- Printing files from different users
  - Printf uses a lock

Lock Example: Malloc/Free

- Malloc acquire and free can be made thread safe by acquiring lock before accessing the heap

```
char *malloc (n) {
    heaplock.acquire();
    p = allocate memory
    heaplock.release();
    return p;
}
```

```
void free(char *p) {
    heaplock.acquire();
    put p back on free list
    heaplock.release();
}
```

Lock properties

- Mutual Exclusion
  - At most one thread holds the lock
- Progress
  - If no thread holds the locks, many threads may attempt acquiring the lock. One thread succeeds
- Bounded Waiting
  - If thread T attempts to acquire a lock, then there exists a bound on the number of times other threads can successfully acquire the lock before T does

Locks

- Locks provide two **atomic** operations:
  - Lock.acquire() – wait until lock is free; then mark it as busy
    - After this returns, we say the calling thread *holds* the lock
  - Lock.release() – mark lock as free
    - Should only be called by a thread that currently holds the lock
    - After this returns, the calling thread no longer holds the lock

Definitions

- Atomic Operations
  - An operation that runs to completion or not at all
  - These are the primitives on which to construct various synchronization primitives
- Mutual Exclusion: Ensuring only one thread does a particular thing at a time (one thread *excludes* the others)
  - Type of synchronization
- Critical Section: Code exactly one thread can execute at once
  - Result of mutual exclusion
- Lock: An object only one thread can hold at a time
  - Provides mutual exclusion