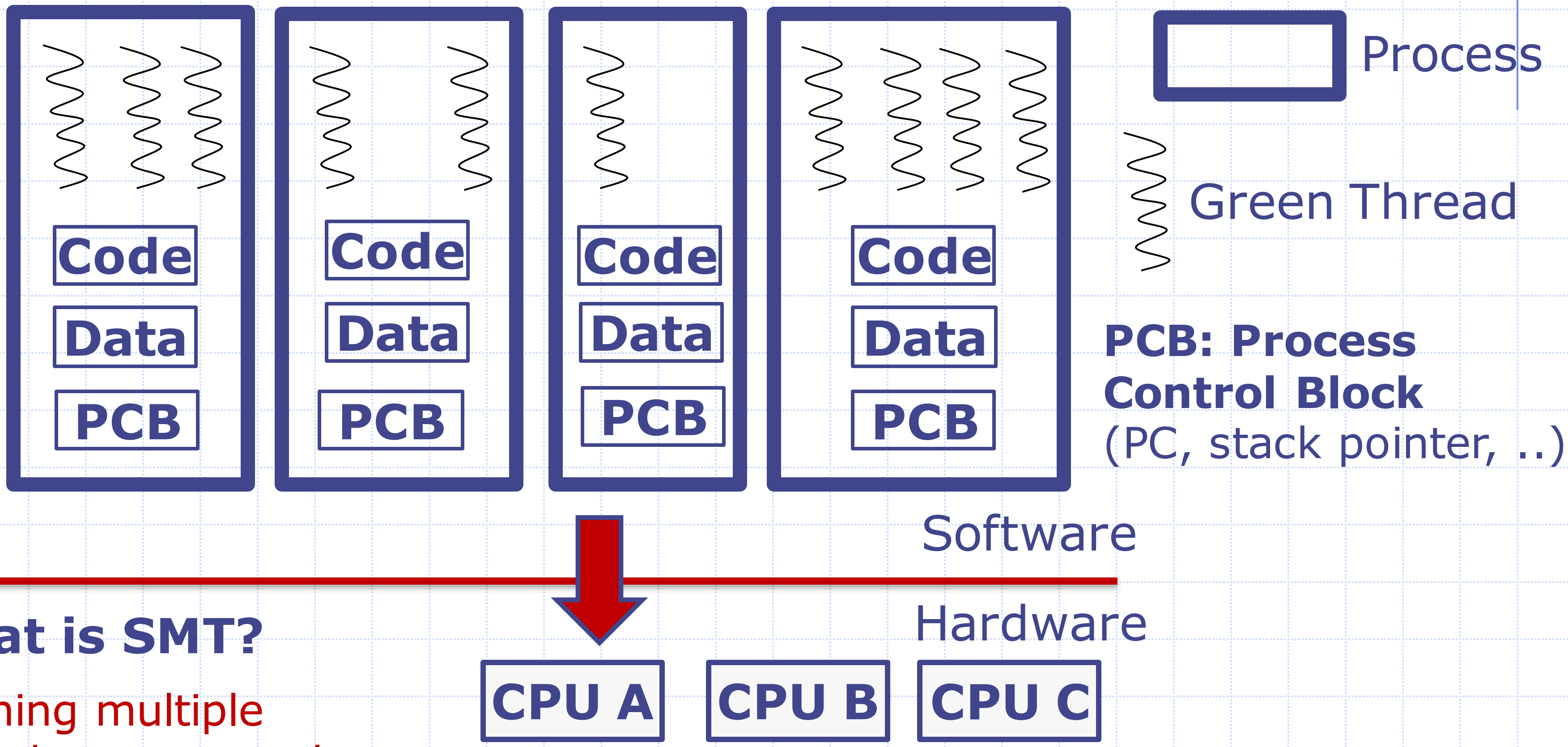
Constructive Computer Architecture

Multithreaded Programming: Synchronization and Sequential Consistency

Thomas – EPFL

Slides adapted from 6.192 Spring '23 with Arvind (MIT) + contributions from Tushar Krishna (Georgia Tech)

Processes and threads



PCB: Process Control Block (PC, stack pointer, ..)

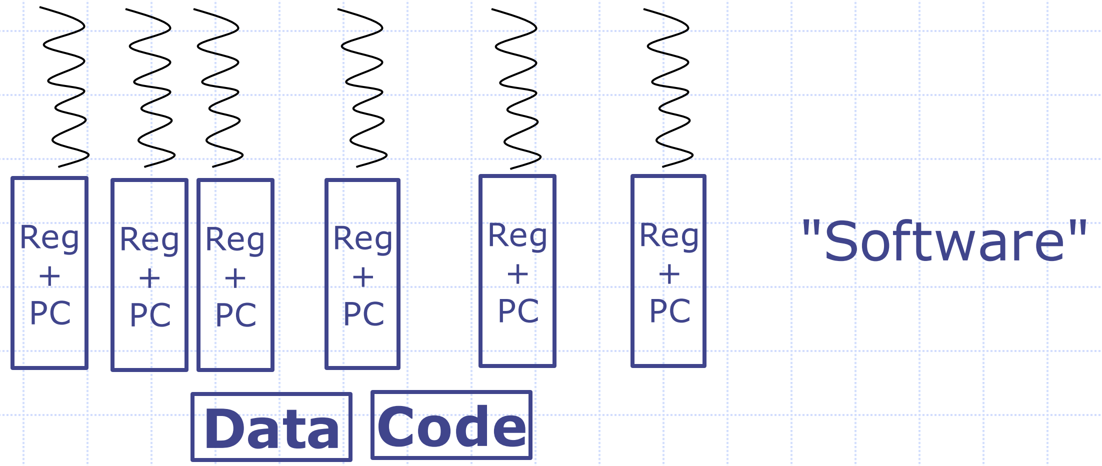
What is SMT?

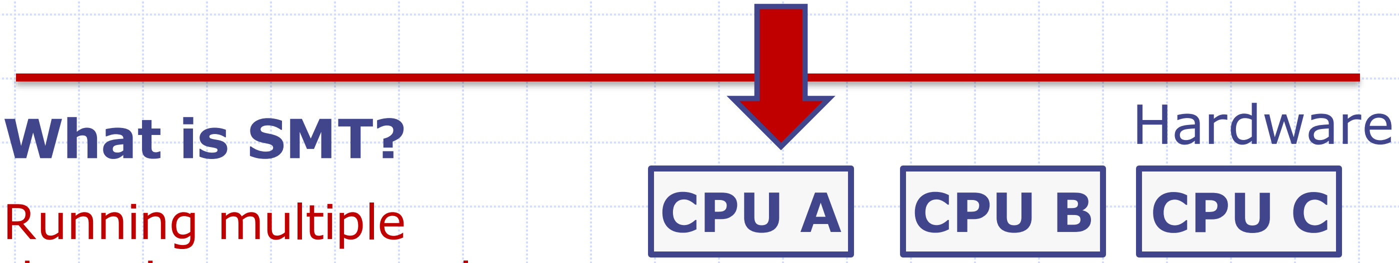
Running multiple threads concurrently on same CPU

What is Multiprocessing?

Running multiple threads concurrently over multiple CPUs

Processes and threads





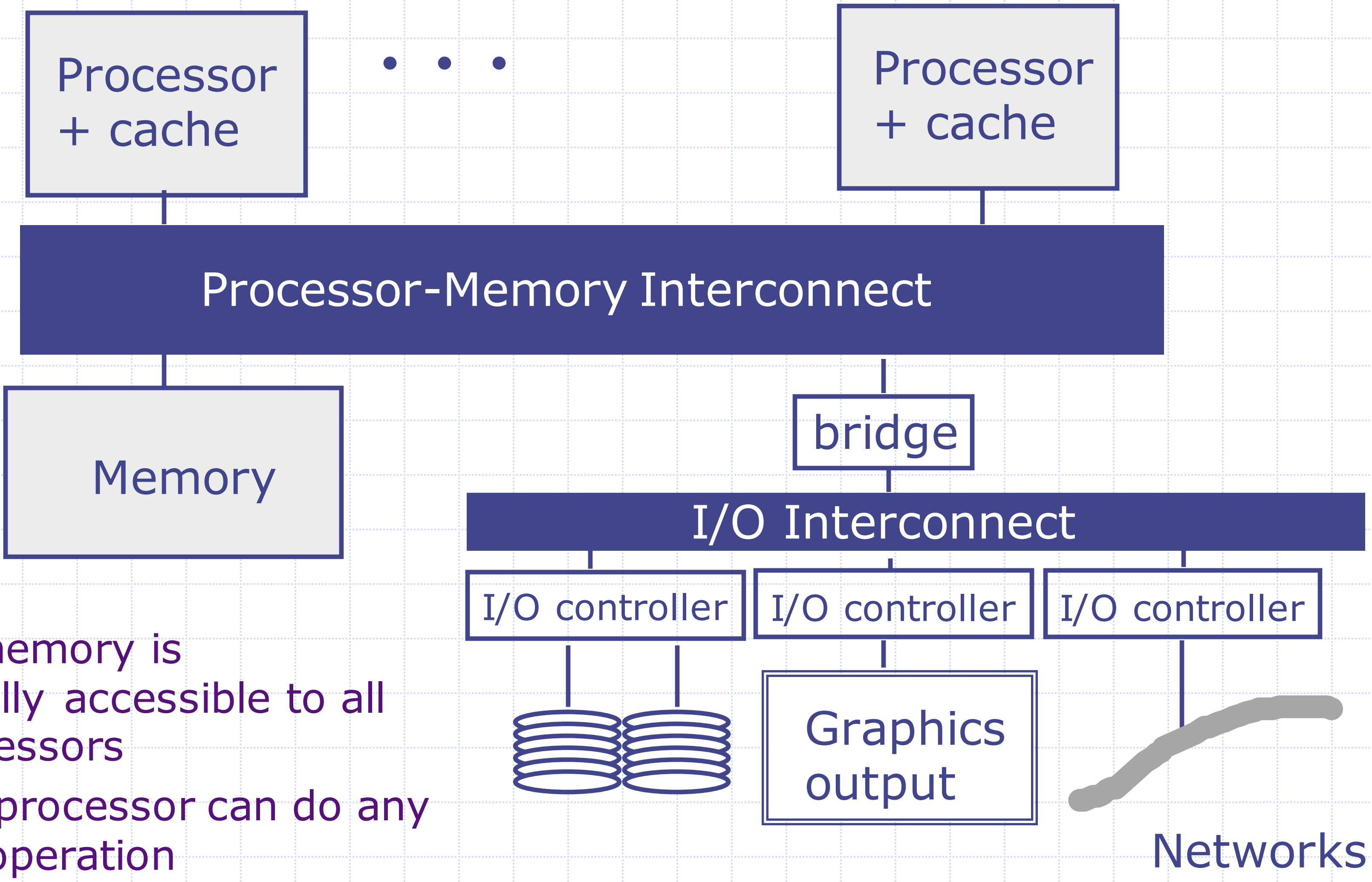
What is SMT?

Running multiple threads concurrently on same CPU

What is Multiprocessing?

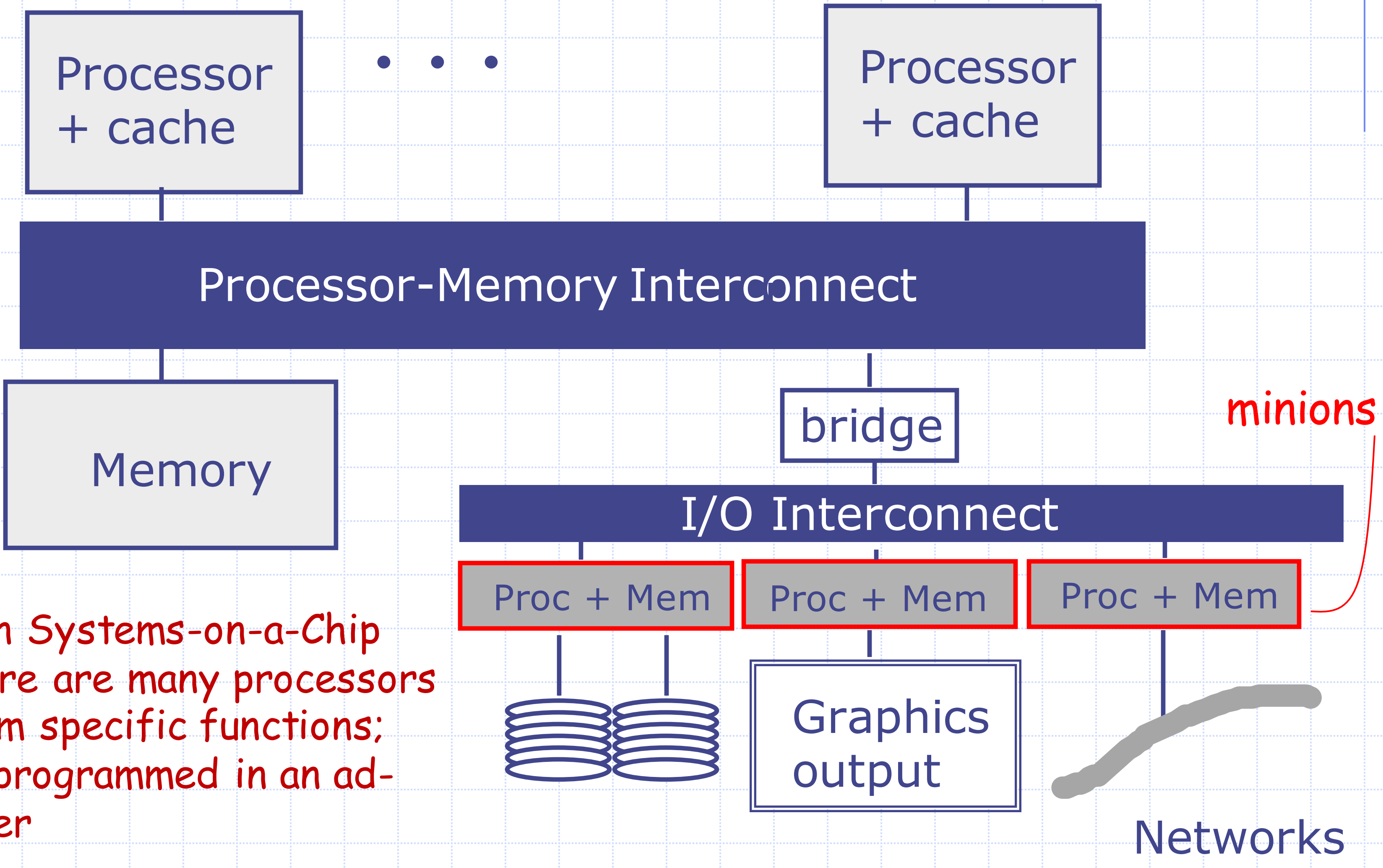
Running multiple threads concurrently over multiple CPUs

Symmetric Multiprocessors



- All memory is equally accessible to all processors
- Any processor can do any I/O operation

Heterogeneous Systems



In modern Systems-on-a-Chip (SoC) there are many processors to perform specific functions; they are programmed in an ad-hoc manner

Multithreaded Programming

- Multiple *independent sequential threads* which compete for shared resources such as memory and I/O devices
  - usually managed by the operating system
  - OS often runs multiple threads for efficient management of resources even on a single processor
- Multiple *cooperating sequential threads*, which communicate via the shared memory system, i.e., shared data structures
  - an application can often be completed faster by decomposing it into multiple threads and running them on multiprocessors

Focus of Today’s Lecture

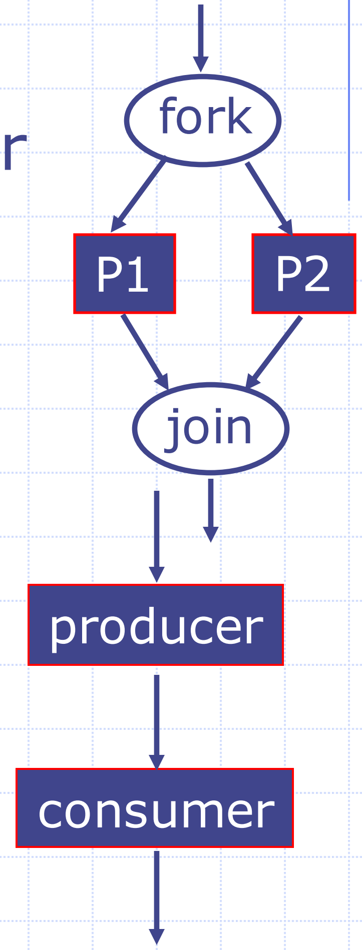
- Supporting Multi-threaded Programming
    - Synchronization
    - Sequential Consistency

Focus of Today’s Lecture

- Supporting Multi-threaded Programming
  - Synchronization
  - Sequential Consistency

Synchronization

- Need for synchronization arises whenever there are parallel processes or threads in a system
  - *Forks and Joins: A* parallel process/thread may want to wait until several events have occurred
  - *Producer-Consumer:* A consumer process/thread must wait until the producer process has produced data
  - *Mutual Exclusion:* Operating system has to ensure that a resource is used by only one process/thread at a given time

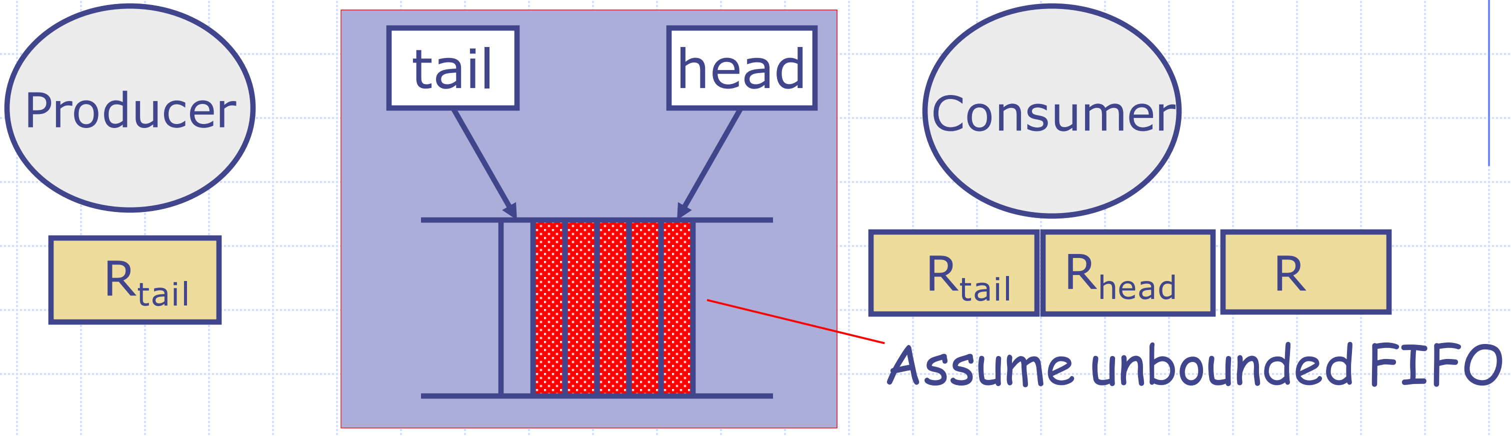


Thread-safe programming

- Multithreaded programs can be executed on a uniprocessor by *timesharing*
    - Each thread is executed for a while (timer interrupt) and then the OS switches to another thread, repeatedly
- *Thread-safe* multithreaded programs behave the same way regardless of whether they are executed on multiprocessors or a single processor

In the rest of lecture we will assume that each thread has its own processor to run

Example: Producer-Consumer communicate via a FIFO



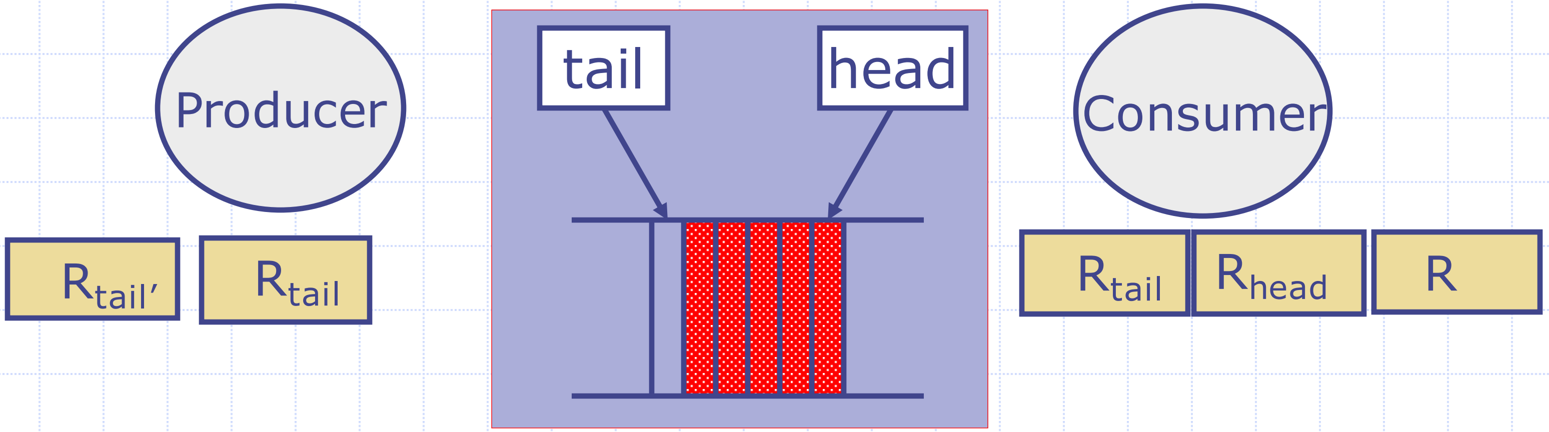
Producer posting Item x:

```
        Load $R_{tail}$, tail              src addr
        Store ($R_{tail}$), x
dest addr
in a reg  $R_{tail}$=$R_{tail}$+1
        Store tail, $R_{tail}$
      dest addr
```

Consumer:

```
        Load $R_{head}$, head
spin:   Load $R_{tail}$, tail
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
        process(R)
```

Multithreaded programming is subtle



Producer posting Item x:

```
        Load $R_{tail}$, tail
        Store ($R_{tail}$), x
        $R_{tail'}$=$R_{tail}$+1
        Store tail, $R_{tail'}$
```

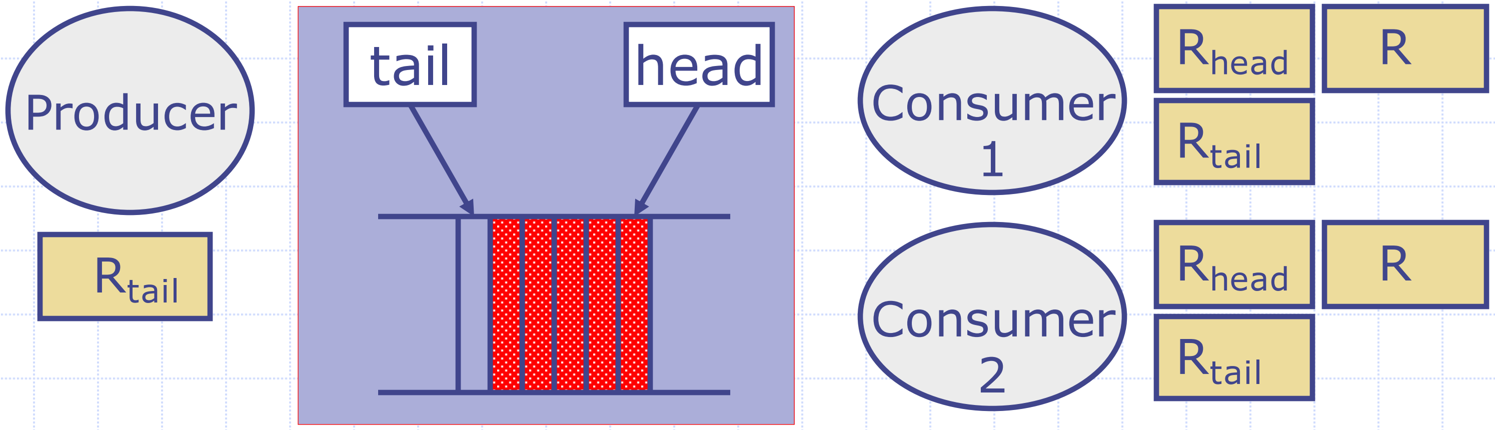
Consumer:

```
        Load $R_{head}$, head
spin:   Load $R_{tail}$, tail
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
        process(R)
```

What is wrong with this code?

reordering the stores can cause the consumer to see stale data

Multiple Consumers



Producer posting Item x:

```
        Load $R_{tail}$, tail
        Store ($R_{tail}$), x
        $R_{tail}$=$R_{tail}$+1
        Store tail, $R_{tail}$
```

Consumer:

```
        Load $R_{head}$, head
spin:   Load $R_{tail}$, tail
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
        process(R)
```

What is wrong with this code?

Critical section:
Needs to be executed atomically by one consumer $\Rightarrow$ locks

Locks or Semaphores

E. W. Dijkstra, 1965

```
Process i
    acquire(s)
        <critical section>
    release(s)
```

The execution of the critical section is protected by locks; Only one process can hold the lock at a time

- Lock s has two values:
  - *Unlocked (s=0):* means that no process has the lock
  - *Locked (s=1):* means that exactly one process has the lock and it can access the critical section
- Once a process successfully acquires a lock, it executes the critical section and then sets s to zero by releasing the lock

Implementing locks is quite difficult; ISAs provide special atomic instructions to implement locks

Atomic read-modify-write instructions

m is a memory location, R is a register

```
Test&Set m, R:
    R ← M[m];
    if R==0 then
        M[m] ← 1;
```

Location m can be set to one only if it contains a zero; the old value is returned in R

```
Swap m, R:
    $R_t$ ← M[m];
    M[m] ← R;
    R ← $R_t$;
```

Location m is first read and then set to the new value; the old value is returned in R

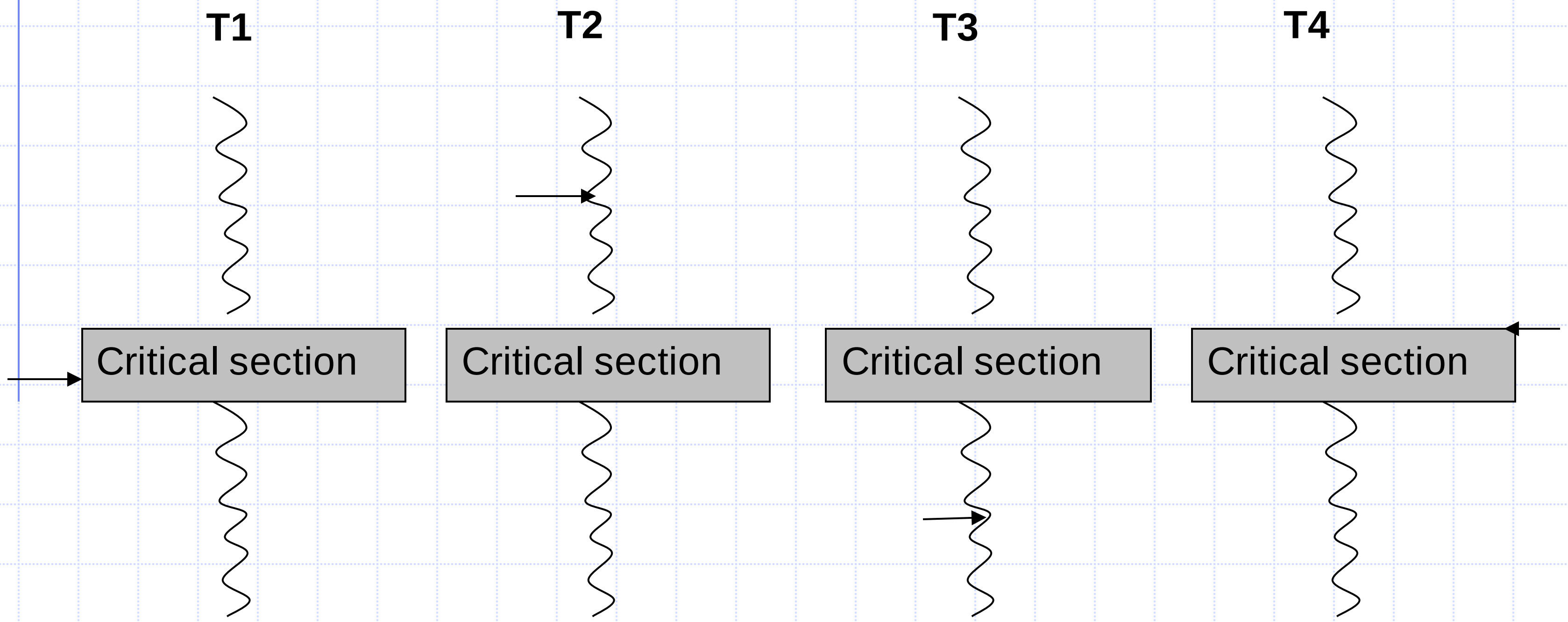
Multiple Consumers Example *using the Test&Set Instruction*

A consumer acquires the lock (mutex) before reading the head value

```
lock:   Test&Set mutex, $R_{temp}$
        if ($R_{temp}$==1) goto lock
        Load $R_{head}$, head
spin:   Load $R_{tail}$, tail
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
unlock: Store mutex, 0
        process(R)
```

← *Critical Section*

Synchronization and Concurrency



- T1 is **active** and executing code **inside** its **critical section**.
- T2 is **active** and executing code **outside** its **critical section**.
- T3 is **active** and executing code **outside** its **critical section**.
- T4 is **blocked** and **waiting** to get into its **critical section**.
    - (It will get in once the lock is released by T1).

Programming Challenges

- How to decide what part of code is the “critical section”?
  - What happens if critical section is too large?

- Our example was written assuming that the instructions per thread are executed *in order*
  - *An architecture may execute instructions out of order to gain higher performance*

    $\Rightarrow$ *Gives rise to memory model issues*

Implementation Challenges

- Atomic instructions (read-modify-write) are quite disruptive in pipelined machines
- What if the process stops or is swapped out *while* in the critical section?
  - Lock may never get released! Every thing stops
  - More sophisticated programming is needed to deal with such eventualities

Focus of Today’s Lecture

- Supporting Multi-threaded Programming
  - Synchronization
  - Sequential Consistency

Memory Ordering Nuances

Concurrent tasks: T1, T2

Shared variables: X, Y (initially X = 0, Y = 0)

T1:

Store X, 1 *(X = 1)*

Store Y, 2 *(Y = 2)*

T2:

Load R_1, Y

Store Y′, R_1 *(Y′= Y)*

Load R_2, X

Store X′, R_2 *(X′= X)*

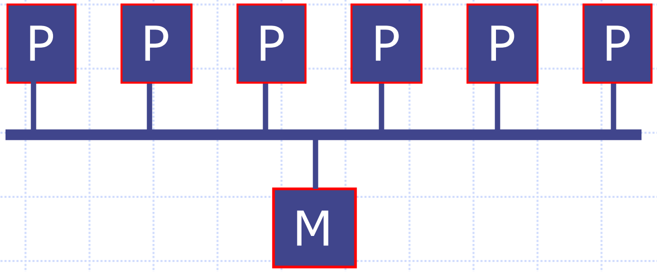
What are the legitimate answers for X′ and Y′ ?

(X′,Y′) ε {(1,2), (0,0), (1,0), ~~(0,2)~~} ?

If y is 2 then x cannot be 0

Sequential Consistency

A Memory Model



"A system is *sequentially consistent* if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in the order specified by the program"

Leslie Lamport

Sequential Consistency =
arbitrary *order-preserving interleaving*
of memory references of sequential programs

Why SC may be violated

Sequential consistency imposes more memory ordering constraints than those imposed by uniprocessor program dependencies ($\longrightarrow$)

What are these in our example ?

additional SC requirements ($\longrightarrow$)

T1:

$$\begin{tikzcd}[row sep=small] \text{Store } X, 1 \quad \mathit{(X = 1)} \arrow[d, bend right] \\ \text{Store } Y, 2 \quad \mathit{(Y = 2)} \end{tikzcd}$$

T2:

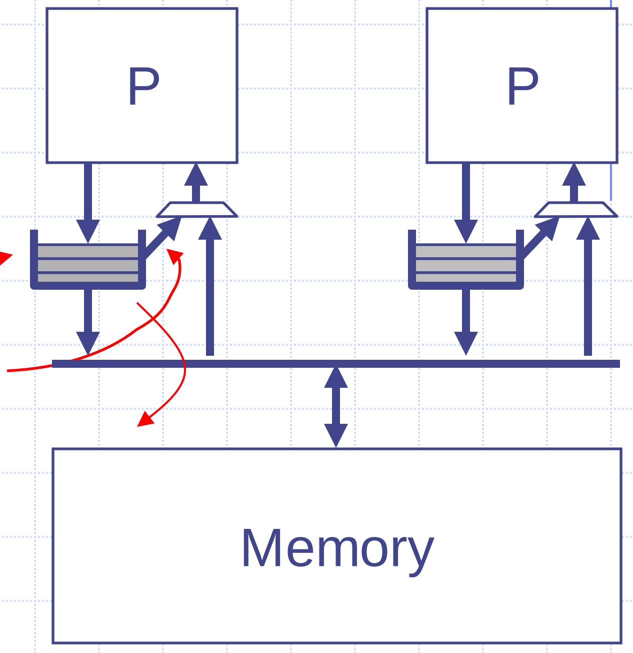
$$\begin{tikzcd}[row sep=small] \text{Load } R_1, Y \arrow[d, bend right] \\ \text{Store } Y', R_1 \quad \mathit{(Y' = Y)} \arrow[d, bend right] \\ \text{Load } R_2, X \arrow[d, bend right] \\ \text{Store } X', R_2 \quad \mathit{(X' = X)} \end{tikzcd}$$

High-performance processor implementations often violate SC by not enforcing the extra dependencies required by SC

Example Store Buffers

Store Buffers

- A processor considers a Store to have been executed as soon as it is stored in the Store buffer, that is, before it is put in memory
- A load can read values from the local store buffer (forwarding)



Loads/Stores can appear to be ordered differently to other processors

==> violate SC

Some systems only enforce *FIFO ordering for the stores to the same address* while moving a store from store buffer to memory

Violations of SC

Example 1

Process 1	*Process 2*
Store X, 1;	Load R_1, Y;
Store Y, 2;	Load R_2, X;

Question: Is it possible that $R_1=2$ but $R_2=0$?

- *Sequential consistency:* *No*
- *With FIFO store buffers:* *No*
- *With non-FIFO store buffers:* *Yes*

Initially, all memory locations contain zeros

Violations of SC
Example 2

Process 1	*Process 2*
Store X, 1;	Store Y, 2;
Load R_1, Y;	Load R_2, X;

Question: Is it possible that $R_1=0$ and $R_2=0$?

- *Sequential consistency:* *No*
- *Suppose Stores don't leave the store buffers before the Loads are executed:* *Yes !*

Initially, all memory locations contain zeros

A Practical Producer-Consumer Example *continued*

Producer posting Item x:

```
        Load $R_{tail}$, (tail)
*1*     Store ($R_{tail}$), x
        $R_{tail}$=$R_{tail}$+1
*2*     Store tail, $R_{tail}$
```

Consumer:

```
        Load $R_{head}$, head
spin:   Load $R_{tail}$, tail              *3*
        if $R_{head}$==$R_{tail}$ goto spin
        Load R, ($R_{head}$)               *4*
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
        process(R)
```

Can the tail pointer get updated before the item x is stored?

Programmer assumes that if 3 happens after 2, then 4 happens after 1.

Problem sequences:

2, 3, 4, 1
4, 1, 2, 3

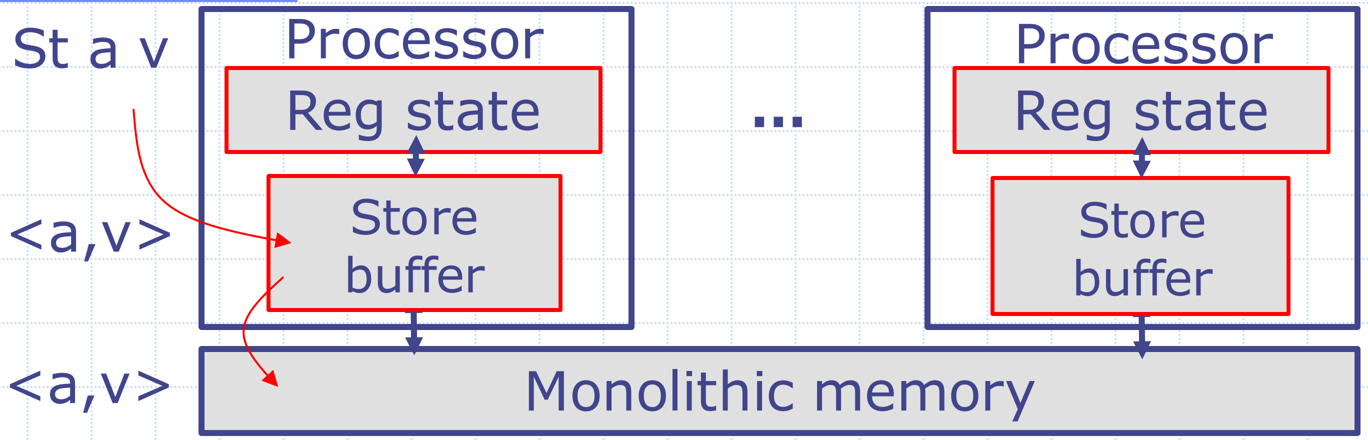
How do we determine which sequences are allowed?

Memory Consistency Model

- In practice, processors use “weaker” memory models than SC
  - Why?

- The “Memory Consistency Model” *(aka Memory Model)* describes what values can be returned by load instructions across concurrent processes/threads

TSO: A memory model for machines with store buffers



simple and vendor independent

TSO allows loads to overtake stores

- A store first goes into the Store buffer (SB)
- A load reads the youngest corresponding entry from SB before reading the memory
- A store is dequeued from the SB in FIFO order to update the monolithic memory (*background rule*)
- A *commit fence* stalls local execution until SB is empty

Memory Fences

Instructions to sequentialize memory accesses

Processors which do not support SC memory model provide *memory fence* instructions to force the serialization of memory accesses as needed

Producer posting Item x:
```
        Load $R_{tail}$, (tail)
        Store ($R_{tail}$), x
        $Fence_{SS}$
        $R_{tail}$=$R_{tail}$+1
        Store tail, $R_{tail}$
```

ensures that tail ptr is not updated before x has been stored

Consumer:
```
        Load $R_{head}$, (head)
spin:   Load $R_{tail}$, (tail)
        if $R_{head}$==$R_{tail}$ goto spin
        $Fence_{LL}$
        Load R, ($R_{head}$)
        $R_{head}$=$R_{head}$+1
        Store head, $R_{head}$
        process(R)
```

ensures that R is not loaded before x has been stored

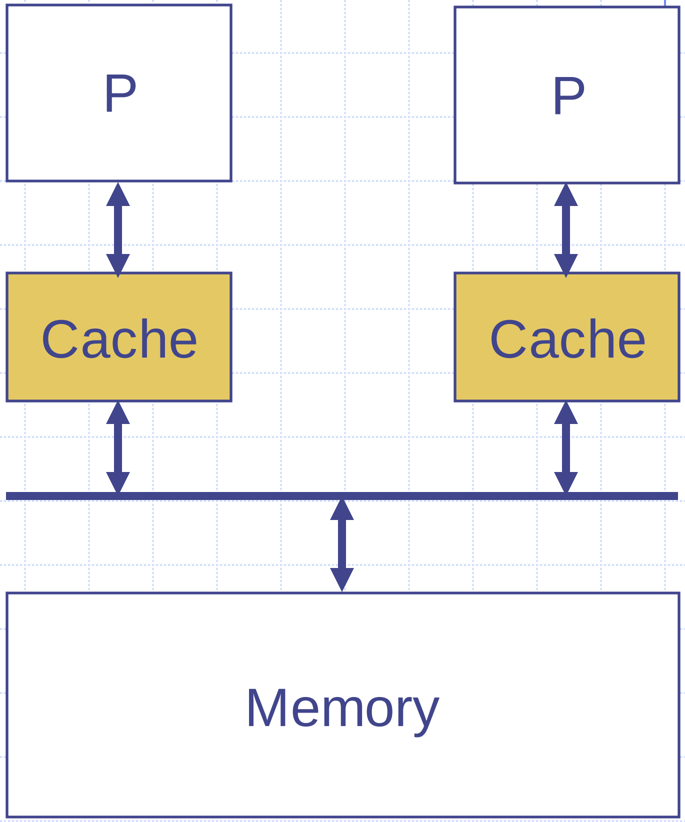
RISC-V has one instruction called "fence"

Memory Model Challenges

- Architectural optimizations that are correct for uniprocessors, often violate sequential consistency and result in a new memory model for multiprocessors
- Memory model issues are subtle and contentious
  - x86 ISA uses TSO, whose definition is easy to understand and unambiguous
  - ISAs for ARM, PowerPC etc. use weaker memory models and even experts don’t agree on their definitions

SC and Caches

- Caches present a similar problem as store buffers – stores in one cache will not be visible to other caches automatically
- Cache problem is solved differently – *caches are kept coherent*



How to build coherent caches is the topic of next lecture