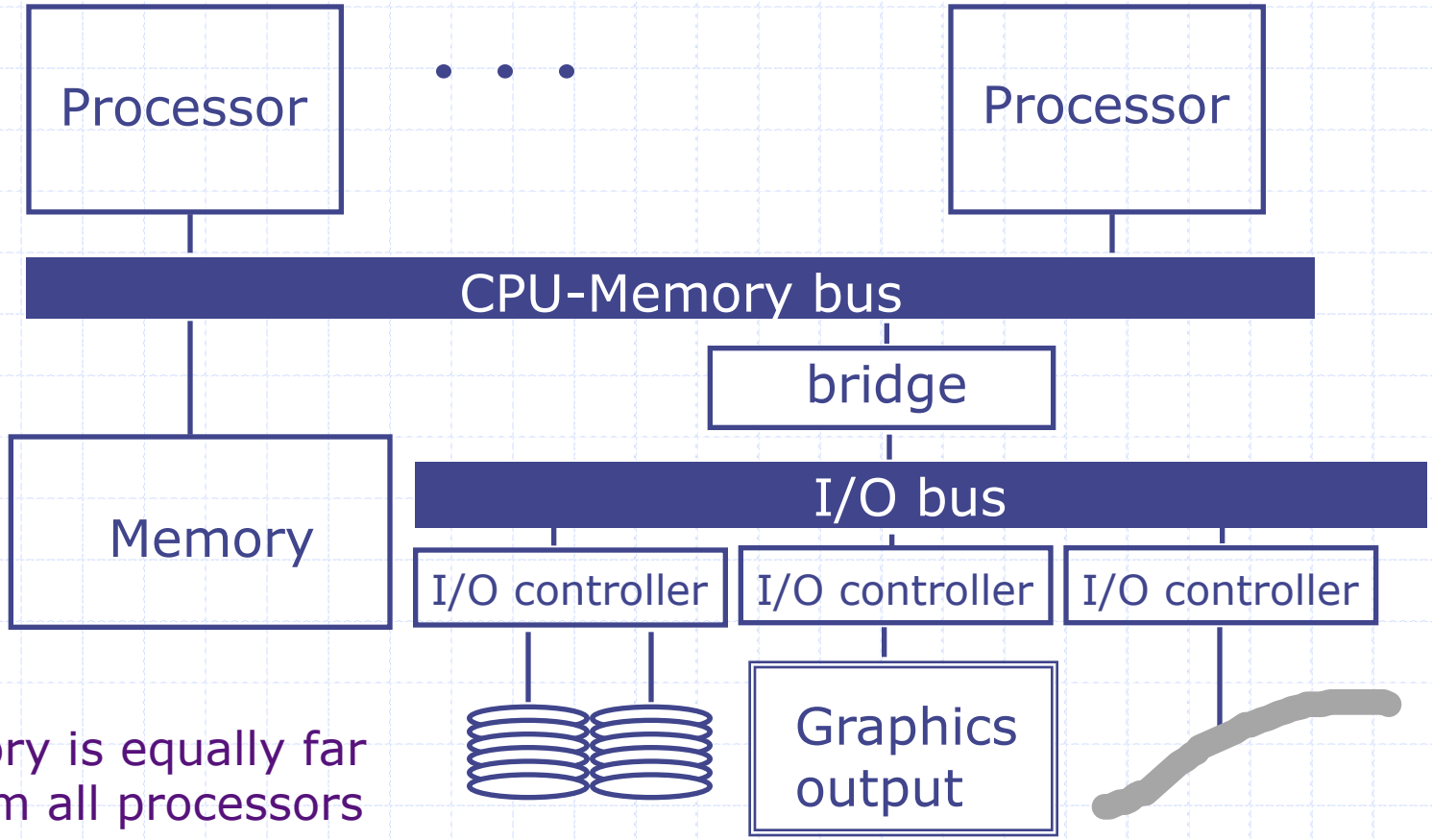
Constructive Computer Architecture

Symmetric Multiprocessors: Synchronization and Sequential Consistency

Arvind
Computer Science & Artificial Intelligence Lab.
Massachusetts Institute of Technology

Symmetric Multiprocessors



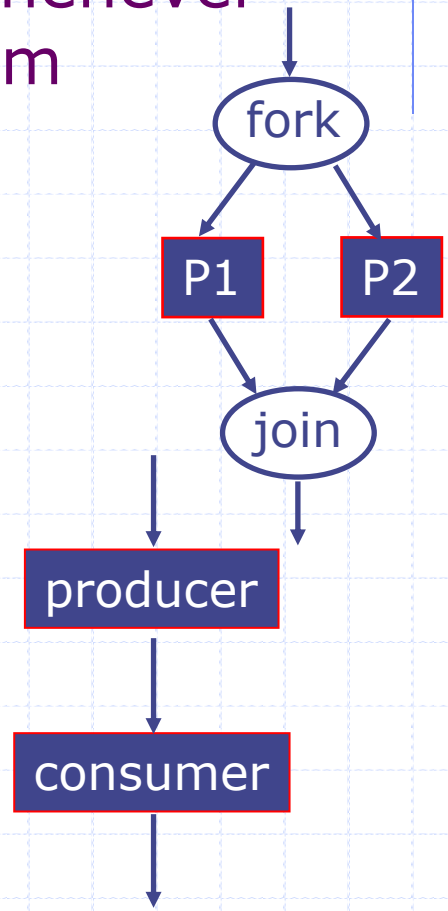
Symmetric?

- All memory is equally far away from all processors
- Any processor can do any I/O operation

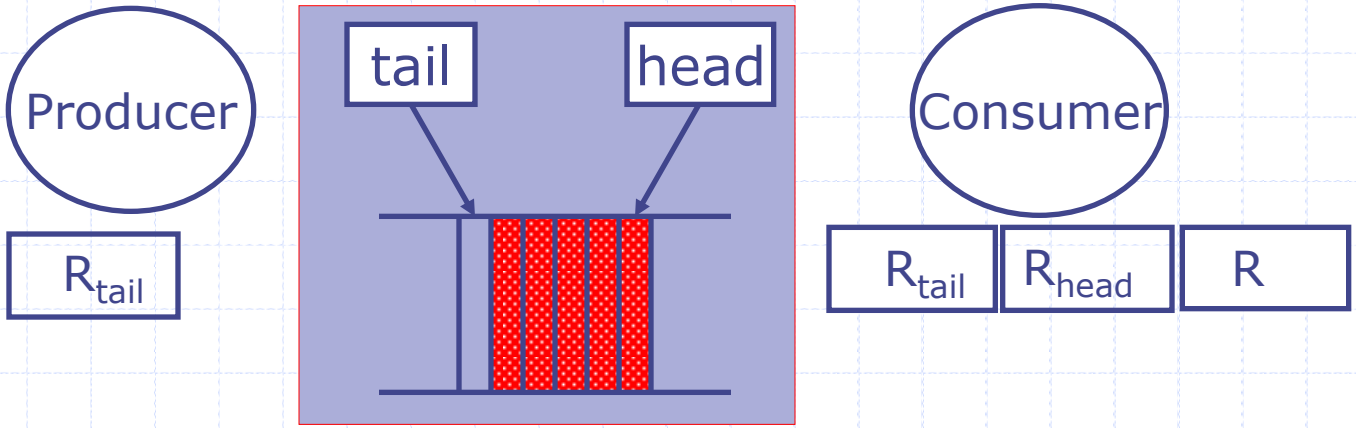
Synchronization

needed even in single-processor systems

- The need for synchronization arises whenever there are parallel processes in a system
  - *Forks and Joins: A* parallel process may want to wait until several events have occurred
  - *Producer-Consumer:* A consumer process 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 at a given time



A Producer-Consumer Example



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)
```

The program is written assuming instructions are executed in order.

Problems?

A 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.

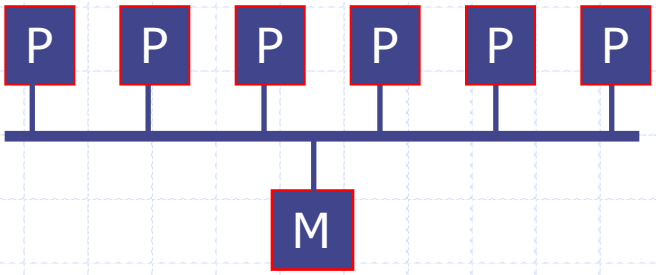
Problem sequences are:

2, 3, 4, 1

4, 1, 2, 3

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

Sequential Consistency

Sequential 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 1

Sequential Consistency

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

What are these in our example ?

additional SC requirements (⟶)

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)

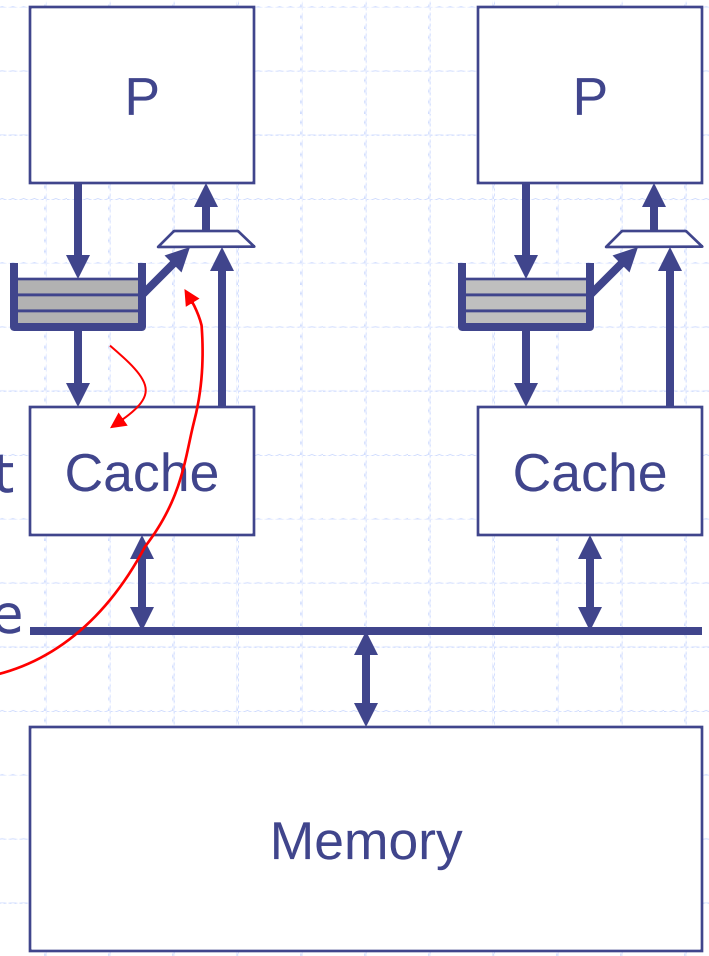
Store X', R_2 *(X'= X)*

High-performance processor implementations often violate SC

Example Store Buffer

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 L1
- Stores can be moved from the store buffer to L1 in a different order
- A load can read values from the local store buffer (forwarding)



The net effect of store buffers is that Loads/Stores can appear to be ordered differently to different processors – breaks SC

Violations of SC

Example 1

Process 1	*Process 2*
Store ($flag_1$),1;	Store ($flag_2$),1;
Load r_1, ($flag_2$);	Load r_2, ($flag_1$);

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 !*

Total Store Order (TSO):
IBM 370, Sparc's TSO memory model, x86

Initially, all memory locations contain zeros

Violations of SC

Example 2: Non-FIFO Store buffers

Process 1	*Process 2*
Store (a), 1;	Load r_1, (flag);
Store (flag), 1;	Load r_2, (a);

Question: Is it possible that r_1=1 but r_2=0?

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

Sparc's PSO memory model

Violations of SC

Example 3: Non-Blocking Caches

Process 1	*Process 2*
Store (a), 1;	Load r_1, (flag);
Store (flag), 1;	Load r_2, (a);

Question: Is it possible that r_1=1 but r_2=0?

- *Sequential consistency:* *No*
- *Assuming stores are ordered:*

  Yes because Loads can be reordered

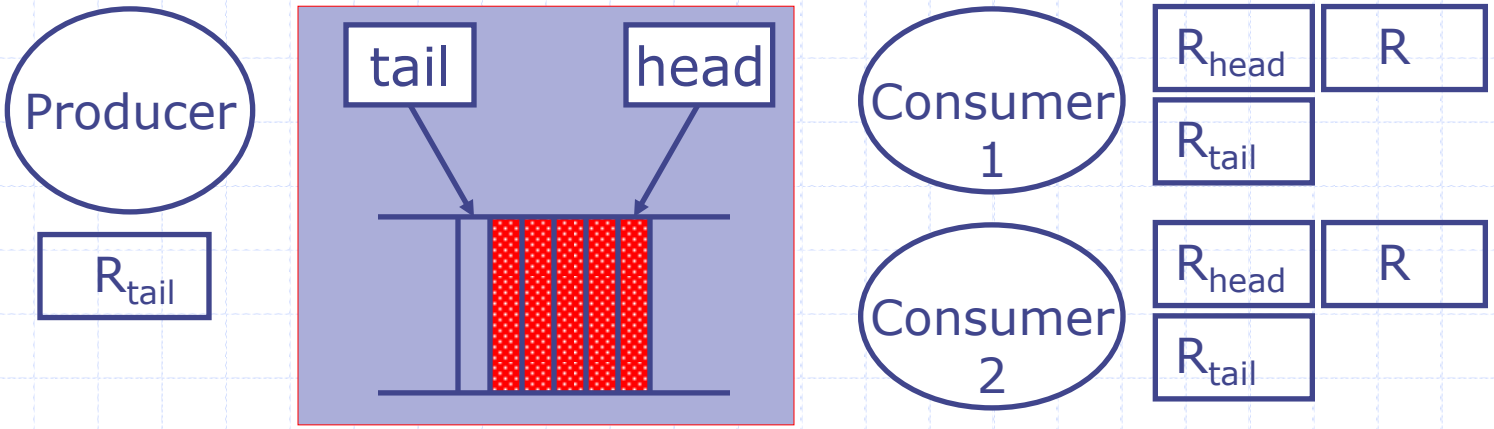
  Sparc's RMO, PowerPC's WO, Alpha

Memory Model Issue

- 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 because most ISA specifications (X86, ARM, PowerPC, Sparc, MIPS) are ambiguous

For the rest of the lecture we will assume the architecture is SC and focus on synchronization issues

Multiple Consumer Example



Producer posting Item x:

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

Consumer:

spin: Load R_{head}, (head)
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)

Critical section:
Needs to be executed atomically by one consumer ⇒ locks

What is wrong with this code?

Locks or Semaphores

E. W. Dijkstra, 1965

```
Process i
    lock(s)
        <critical section>
    unlock(s)
```

The execution of the critical section is protected by lock s. Only one process can hold the lock.

- Suppose the lock s can have only two values:
  - s=0 means that no process has the lock
  - s=1 means that exactly one process has the lock and therefore can access the critical section
  - Once a process successfully acquires a lock, it executes the critical section and then sets s to zero by executing unlock(s)
- Implementation of locks is quite difficult using just Loads and Stores. 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

```
Swap (m), R:
    Rt ← M[m];
    M[m] ← R;
    R ← Rt;
```

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

Multiple Consumers Example *using the Test&Set Instruction*

```
lock:    Test&Set (mutex),Rtemp
         if (Rtemp=1) goto lock
         Load Rhead, (head)
spin:    Load Rtail, (tail)
         if Rhead==Rtail goto spin
         Load R, (Rhead)
         Rhead=Rhead+1
         Store head, Rhead
unlock:  Store mutex, 0
         process(R)
```

Critical Section

What if the process stops or is swapped out while in the critical section?

Nonblocking Synchronization

Load-reserve & Store-conditional

Special register(s) to hold reservation flag and address, and the outcome of store-conditional

```
Load-reserve R, (m):
        <flag, adr> ← <1, m>;
        R ← M[m];
```

```
Store-conditional (m), R:
    if <flag, adr> == <1, m>
    then  cancel other procs' reservation on m;
          M[m] ← R;
          status ← succeed;
    else  status ← fail;
```

```
try:    Load-reserve 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-conditional head, R_head
        if (status==fail) goto try
        process(R)
```

Nonblocking Synchronization

```
Load-reserve R, (m):
      <flag, adr> ← <1, m>;
      R ← M[m];
```

```
Store-conditional (m), R:
      if <flag, adr> == <1, m>
      then  cancel other procs'
                reservation on m;
            M[m] ← R;
            status ← succeed;
      else  status ← fail;
```

- The flag is cleared in other processors on a Store using the CC protocol's invalidation mechanism
- Usually address m is not remembered by Load-reserve; the flag is cleared on *any* invalidation
  - works as long as the Load-reserve instructions are used in a nested manner
- These instructions won't work properly if Loads and Stores can be dynamically reordered

Memory Fences

Instructions to sequentialize memory accesses

Processors with *weak* or non-sequentially-consistent *memory models* need to provide *memory fence* instructions to force the serialization of memory accesses

Producer posting Item x:

```
        Load R_tail, (tail)
        Store (R_tail), x
        Membar_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
        Membar_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