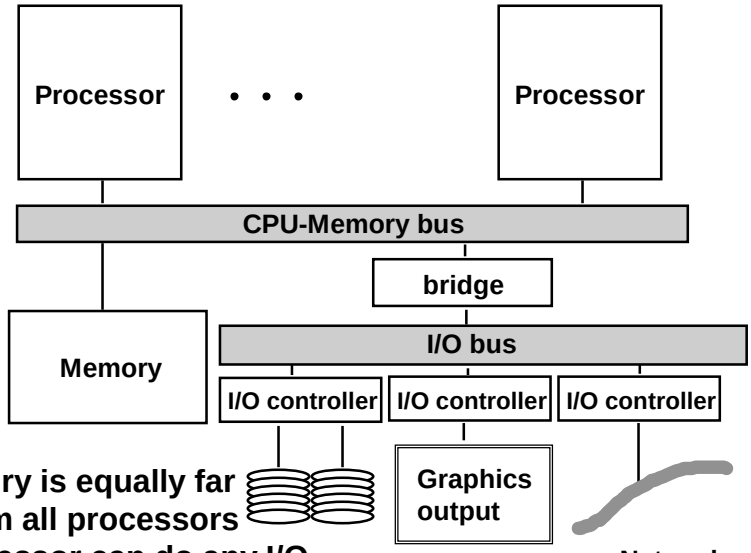
Symmetric Multiprocessors: Synchronization and Sequential Consistency

Krste Asanovic
Laboratory for Computer Science
M.I.T.

http://www.csg.lcs.mit.edu/6.823

Symmetric Multiprocessors

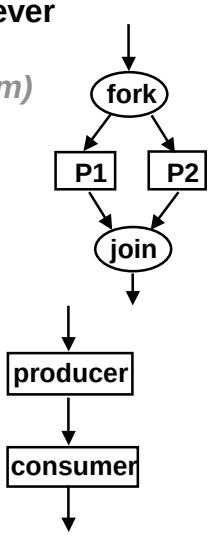


symmetric

- **All memory is equally far away from all processors**
- **Any processor can do any I/O (set up a DMA transfer)**

Synchronization

The need for synchronization arises whenever there are parallel processes in a system
(even in a uniprocessor system)

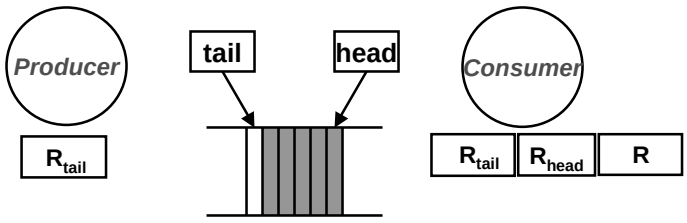


Forks and Joins: **In parallel programming a parallel process may want to wait until several events have occurred**

Producer-Consumer: **A consumer process must wait until the producer process has produced data**

Exclusive use of a resource: **Operating system has to ensure that only one process uses a resource at a given time**

A Producer-Consumer Example



Producer posting Item x:

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

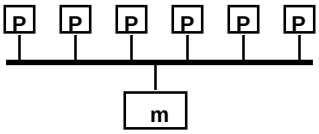
Consumer:

```
        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 problem: **The program is written assuming instructions are executed in order.** ***Suppose the tail pointer gets updated before the item x is stored.***

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

Concurrent sequential tasks: T1, T2
Shared variables: X, Y (initially X = 0, Y = 10)

T1:

Store(X, 1) *(X = 1)*
Store(Y, 11) *(Y = 11)*

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') ∈ { (1, 11), (0, 10), (1, 10), (0, 11) } ?

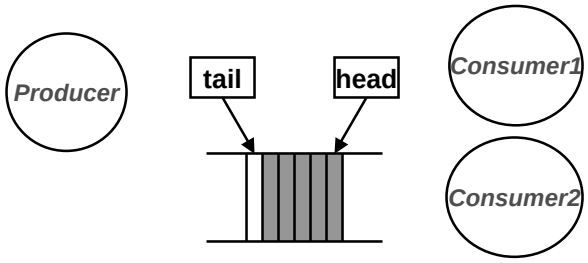
Sequential Consistency

Sequential consistency imposes additional memory ordering constraints in addition to those imposed by uniprocessor program dependencies

Does (can) a system with caches or out-of-order execution capability provide a *sequentially consistent* view of the memory ?

More on this later

Multiple Consumer Example



Producer posting Item x:

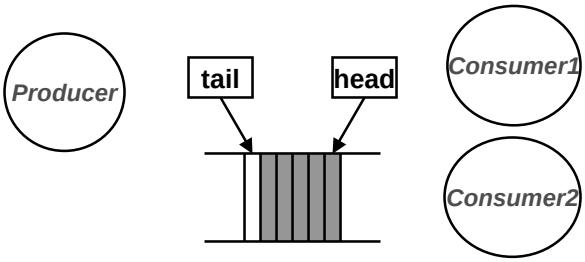
Load(R_{tail}, tail)
Store(R_{tail}, x)
R_{tail}=R_{tail}+1
Store(tail, R_{tail})

Consumer:

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?

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:

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)

Critical Section
Needs to be executed atomically by one consumer

⇒ *locks*

Locks or Semaphores

E. W. Dijkstra, 1965

A *semaphore* is a non-negative integer, with the following operations:

P(s): *if s>0 decrement s by 1 otherwise wait*

V(s): *increment s by 1 and wake up one of the waiting processes*

P's and V's must be executed atomically, i.e., without

- ***interruptions* or**
- ***interleaved accesses to s* by other processors**

Process i

```
P(s)
   <critical section>
V(s)
```

initial value of s determines the maximum no. of processes in the critical section

Implementation of Semaphores

Semaphores (mutual exclusion) can be implemented using ordinary Load and Store instructions in the Sequential Consistency memory model. However, protocols for mutual exclusion are difficult to design...

Simpler solution:

atomic read-modify-write instructions

Examples: ***(m is a memory location, R is a register)***

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

Fetch&Add(m, R_V, R):
R ← M[m];
M[m] ← R + R_V;

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

Multiple Consumers Example

using the Test&Set Instruction

```
P:    Test&Set(mutex,R_temp)
      if (R_temp!=0) goto P
      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)
V:    Store(mutex,0)
      process(R)
```

Critical Section

Other atomic read-modify-write instructions (Swap, Fetch&Add, etc.) can also implement P's and V's

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

Nonblocking Synchronization

Compare&Swap(m,R_t,R_s): *implicit argument* - status

```
if (Rt==M[m])
   then     M[m]=Rs;
            Rs=Rt ;
            status ← success;
   else     status ← fail;
```

```
try:   Load(Rhead, head)
spin:  Load(Rtail, tail)
       if Rhead==Rtail goto spin
       Load(R, Rhead)
       Rnewhead = Rhead+1
       Compare&Swap(head, Rhead, Rnewhead)
       if (status==fail) goto try
       process(R)
```

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(Rhead, head)
spin:  Load (Rtail, tail)
       if Rhead==Rtail goto spin
       Load(R, Rhead)
       Rhead = Rhead + 1
       Store-conditional(head, Rhead)
       if (status==fail) goto try
       process(R)
```

Mutual Exclusion Using Load/Store

A protocol based on two shared variables c1 and c2. Initially, both c1 and c2 are 0 *(not busy)*

Process 1

```
    ...
    c1=1;
L:  if c2=1 then go to L
     < critical section>
    c1=0;
```

Process 2

```
    ...
    c2=1;
L:  if c1=1 then go to L
     < critical section>
    c2=0;
```

What is wrong? _____

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c1 to 0) while waiting.

Process 1

```
    ...
L:  c1=1;
    if c2=1 then
        { c1=0; go to L}
     < critical section>
    c1=0
```

Process 2

```
    ...
L:  c2=1;
    if c1=1 then
        { c2=0; go to L}
     < critical section>
    c2=0
```

Deadlock is not possible but with a low probability a *livelock* may occur

An unlucky process may never get to enter the critical section $\Rightarrow$ *starvation*

A Protocol for Mutual Exclusion

T. Dekker, 1966

A protocol based on 3 shared variables c1, c2 and turn.
Initially, both c1 and c2 are 0 *(not busy)*

Process 1	***Process 2***
... **c1=1;** **turn = 1;** **L: *if* c2=1 & turn=1** **then go to L** **< critical section>** **c1=0;**	**...** **c2=1;** **turn = 2;** **L: *if* c1=1 & turn=2** **then go to L** **< critical section>** **c2=0;**

- **turn = *i* ensures that only process *i* can wait**
- **variables c1 and c2 ensure *mutual exclusion***

Solution for n processes was given by Dijkstra and is quite tricky!

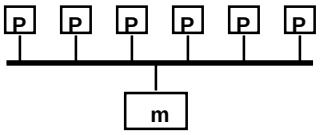
Performance of Locks

Blocking atomic read-modify-write instructions
e.g., Test&Set, Fetch&Add, Swap
vs
Non-blocking atomic read-modify-write instructions
e.g., Compare&Swap,
Load-reserve/Store-conditional
vs
Protocols based on ordinary Loads and Stores

Performance depends on several interacting factors:
degree of contention,
caches,
out-of-order execution of Loads and Stores

later ...

Issues in Implementing Sequential Consistency



Implementation of SC is complicated by two issues

- ***Out-of-order execution capability***

Load(a); Load(b)	***yes***
Load(a); Store(b)	***yes if*** **a ≠ b**
Store(a); Load(b)	***yes if*** **a ≠ b**
Store(a); Store(b)	***yes if*** **a ≠ b**

- ***Caches***

  Caches can prevent the effect of a store from being seen by other processors

Memory Fences

Instructions to sequentialize memory accesses

Processors with *relaxed or weak memory models* (i.e., permit Loads and Stores to different addresses to be reordered) need to provide *memory fence* instructions to force the sequentialization of memory accesses

Examples of processors with relaxed memory models:

Sparc V8 (TSO,PSO): Membar

Sparc V9 (RMO):

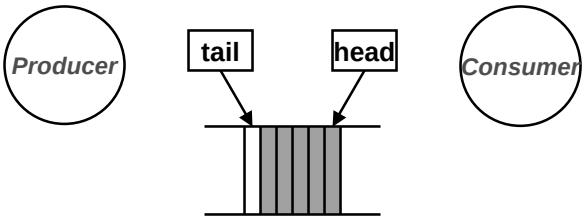
Membar #LoadLoad, Membar #LoadStore

Membar #StoreLoad, Membar #StoreStore

PowerPC (WO): Sync, EIEIO

Memory fences are expensive operations, however, one pays the cost of sequentialization only when it is required

Using Memory Fences



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

Data-Race Free Programs

a.k.a. Properly Synchronized Programs

Process 1

```
...
Acquire(mutex);
  < critical section>
Release(mutex);
```

Process 2

```
...
Acquire(mutex);
  < critical section>
Release(mutex);
```

Synchronization variables (e.g. mutex) are separate from data variables

Accesses to writable shared data variables are protected in critical regions

$\Rightarrow$ no data races except for locks

(Formal definition is elusive)

In general, it cannot be proven if a program is data-race free.

Fences in Data-Race Free Programs

Process 1	*Process 2*
...	...
Acquire(mutex);	Acquire(mutex);
membar;	*membar;*
< critical section>	< critical section>
membar;	*membar;*
Release(mutex);	Release(mutex);

- ***Relaxed memory model allows reordering of instructions by the compiler or the processor as long as the reordering is not done across a fence***
- ***The processor also should not speculate or prefetch across fences***

next time

Effect of caches on Sequential Consistency