

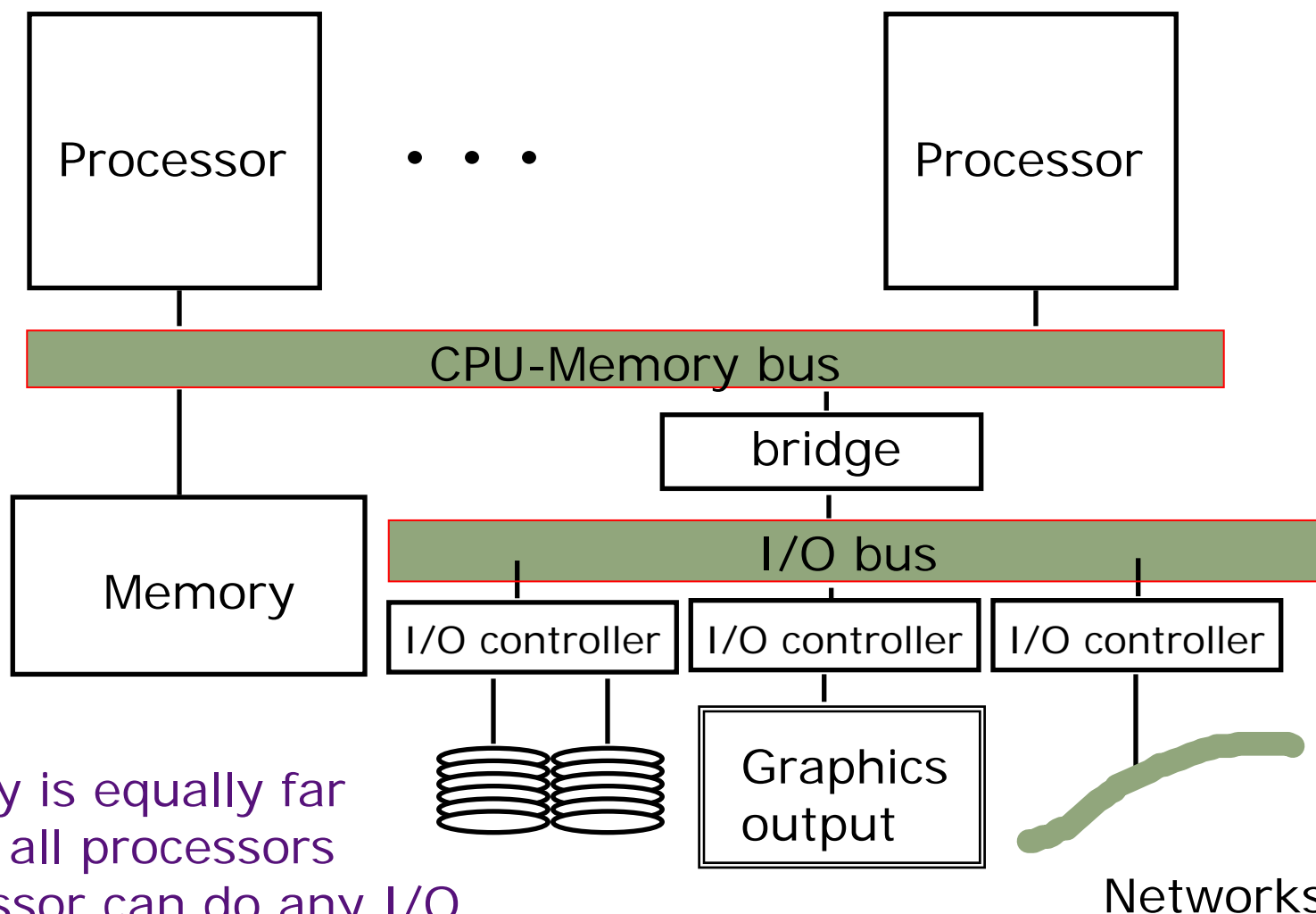
Symmetric Multiprocessors: Synchronization and Sequential Consistency

Arvind

Computer Science and Artificial Intelligence Lab
M.I.T.

*Based on the material prepared by
Arvind and Krste Asanovic*

Symmetric Multiprocessors



symmetric

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

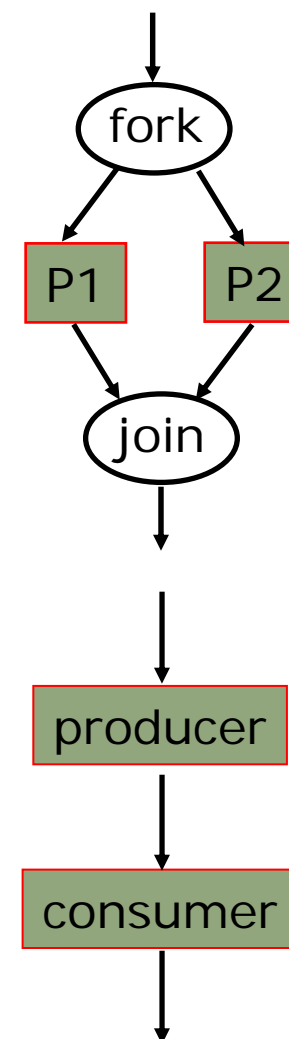
Synchronization

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

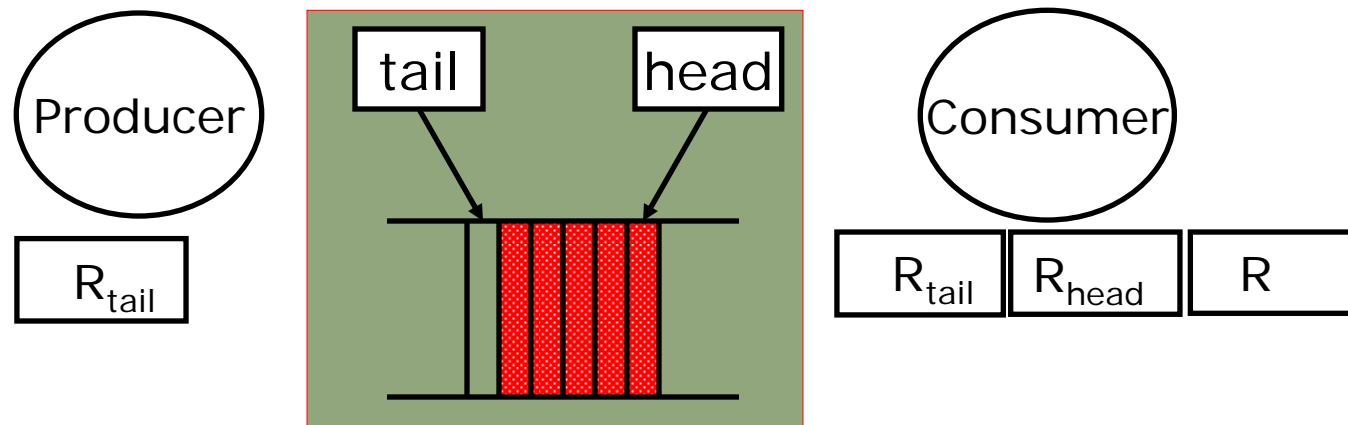
Forks and Joins: In parallel programming a parallel process may want to wait until several events have occurred

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:

```
Load(Rtail, tail)
Store(Rtail, x)
Rtail = Rtail + 1
Store(tail, Rtail)
```

Consumer:

```
Load(Rhead, head)
spin: Load(Rtail, tail)
if Rhead == Rtail goto spin
Load(R, Rhead)
Rhead = Rhead + 1
Store(head, Rhead)
process(R)
```

The program is written assuming instructions are executed in order.

Problems?

A Producer-Consumer Example

continued

Producer posting Item x:

```

Load(Rtail, tail)
1 Store(Rtail, x)
Rtail = Rtail + 1
2 Store(tail, Rtail)

```

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

Consumer:

```

Load(Rhead, head)
spin: Load(Rtail, tail) 3
if Rhead == Rtail goto spin
Load(R, Rhead) 4
Rhead = Rhead + 1
Store(head, Rhead)
process(R)

```

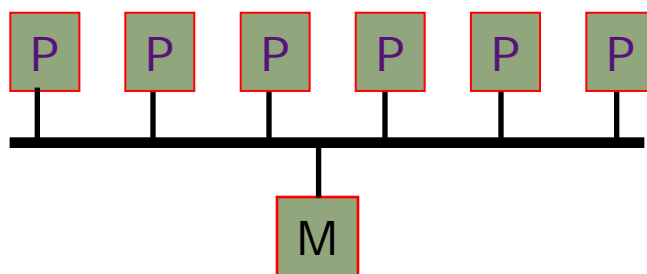
Programmer assumes that if 3 happens after 2, then 4 happens after 1.

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 = 10)

T1:	T2:
Store(X, 1) (X = 1)	Load(R ₁ , Y)
Store(Y, 11) (Y = 11)	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' ?

$(X', Y') \in \{(1, 11), (0, 10), (1, 10), (0, 11)\}$?

If y is 11 then x cannot be 0

Sequential Consistency

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

What are these in our example ?

T1:

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

T2:

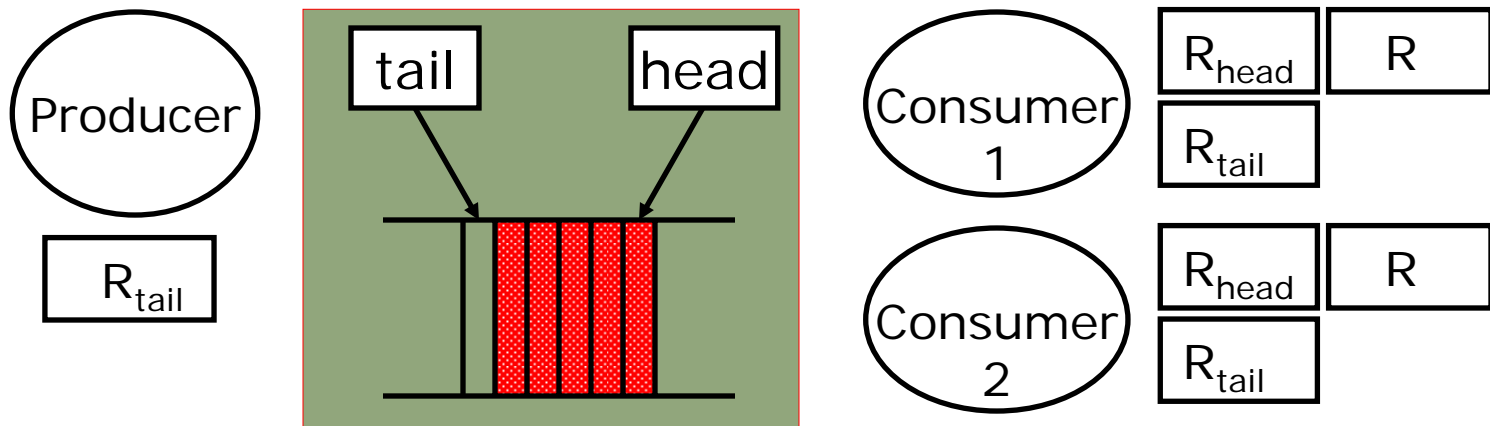
Load(R₁, Y)
Store(Y', R₁) (Y' = Y)
Load(R₂, X)
Store(X', R₂) (X' = X)

\rightarrow additional SC requirements

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(Rtail, tail)
Store(Rtail, x)
Rtail = Rtail + 1
Store(tail, Rtail)
```

Consumer:

```
spin: Load(Rhead, head)
      Load(Rtail, tail)
      if Rhead == Rtail goto spin
      Load(R, Rhead)
      Rhead = Rhead + 1
      Store(head, Rhead)
      process(R)
```

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

What is wrong with this code?

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, Rv, R):  
  R ← M[m];  
  M[m] ← R + Rv;
```

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

Multiple Consumers Example

using the Test&Set Instruction

```
P:   Test&Set(mutex, Rtemp)
      if (Rtemp != 0) goto P
      Load(Rhead, head)
spin: Load(Rtail, tail)
      if Rhead == Rtail goto spin
      Load(R, Rhead)
      Rhead = Rhead + 1
      Store(head, Rhead)
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, Rt, Rs):  
  if (Rt == M[m])  
    then      M[m] = Rs;  
             Rs = Rt ;  
             status ← success;  
    else      status ← fail;
```

status is an
implicit
argument

```
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)
```

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:

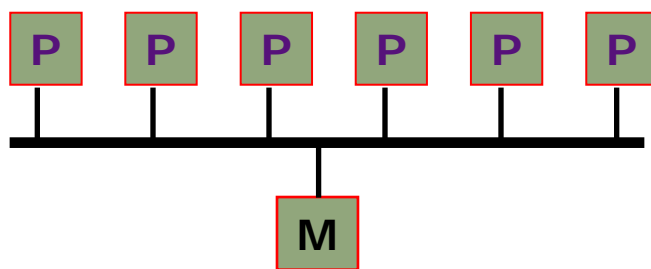
degree of contention,

caches,

out-of-order execution of Loads and Stores

later ...

Issues in Implementing Sequential Consistency



Implementation of SC is complicated by two issues

- *Our-of-order execution capability*

Load(a); Load(b)	yes
Load(a); Store(b)	yes if $a \neq b$
Store(a); Load(b)	yes if $a \neq b$
Store(a); Store(b)	yes if $a \neq b$

- *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 serialization of memory accesses

Examples of processors with relaxed memory models:

Sparc V8 (TSO,PSO): Membar

Sparc V9 (RMO):

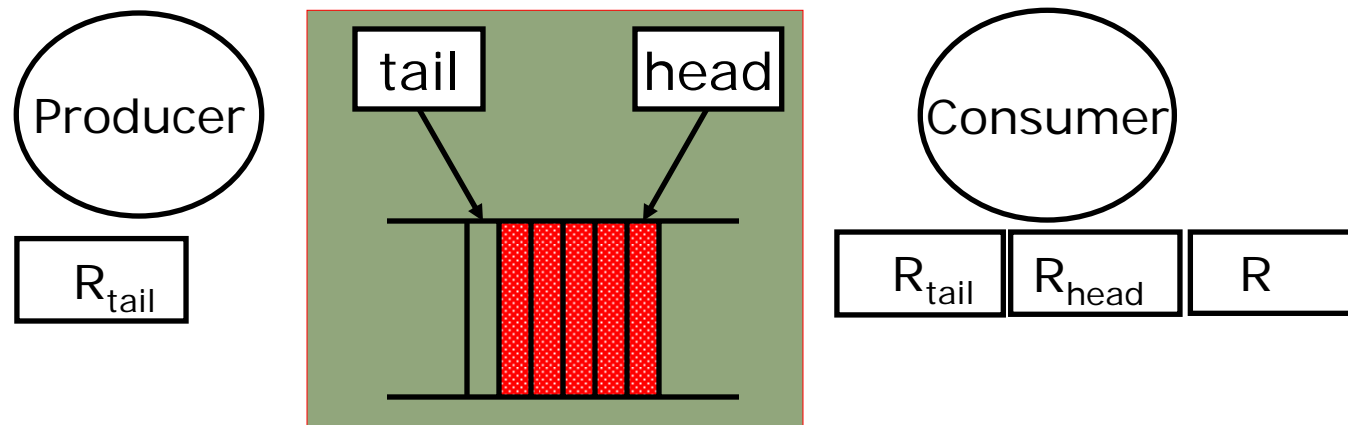
Membar #LoadLoad, Membar #LoadStore

Membar #StoreLoad, Membar #StoreStore

PowerPC (WO): Sync, EIEIO

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

Using Memory Fences



Producer posting Item x :

```

Load( $R_{tail}$ , tail)
Store( $R_{tail}$ ,  $x$ )
MembarSS
 $R_{tail} = R_{tail} + 1$ 
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
MemberLL
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 disjoint from data variables

Accesses to writable shared data variables are protected in critical regions

⇒ *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

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

Process 2

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



Five-minute break to stretch your legs

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?

Deadlock!

Mutual Exclusion: *second attempt*

To avoid *deadlock*, let a process give up the reservation (i.e. Process 1 sets c_1 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 c_1 , c_2 and $turn$.
Initially, both c_1 and c_2 are 0 (*not busy*)

Process 1

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

Process 2

```
...  
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 c_1 and c_2 ensure *mutual exclusion*

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

Analysis of Dekker's Algorithm

Scenario 1

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

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

Scenario 2

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

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

N-process Mutual Exclusion

Lamport's Bakery Algorithm

Process i

Initially $\text{num}[j] = 0$, for all j

Entry Code

```
choosing[i] = 1;
num[i] = max(num[0], ..., num[N-1]) + 1;
choosing[i] = 0;

for(j = 0; j < N; j++) {
    while( choosing[j] );
    while( num[j] &&
           ( ( num[j] < num[i] ) ||
             ( num[j] == num[i] && j < i ) ) );
}
```

Exit Code

```
num[i] = 0;
```

next time

Effect of caches on Sequential Consistency



Thank you !