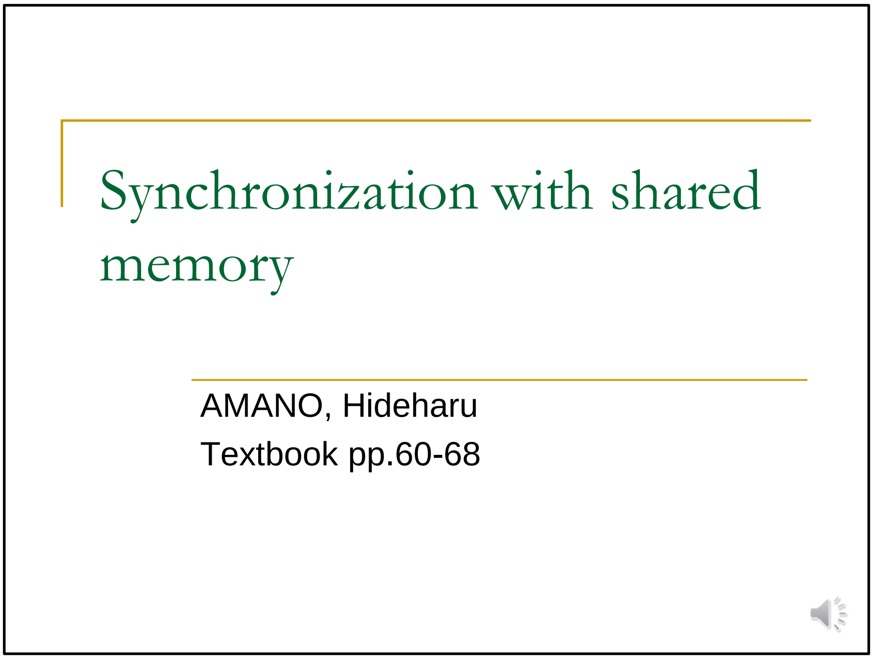
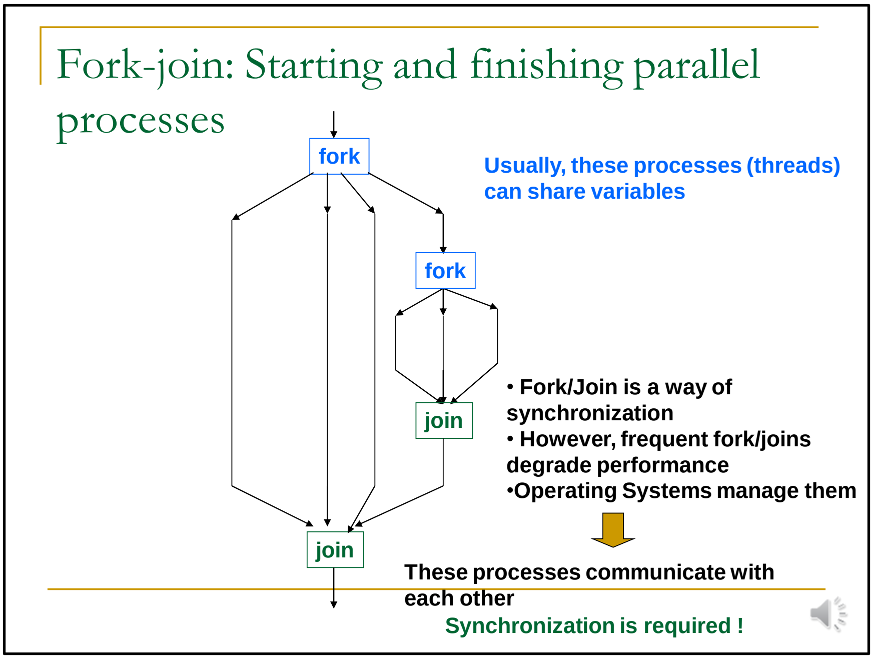


Synchronization with shared memory

AMANO, Hideharu
Textbook pp.60-68

We have learned the structure of shared memory. Now, let’s see the synchronization methods using the shared memory.



Fork-join: Starting and finishing parallel processes

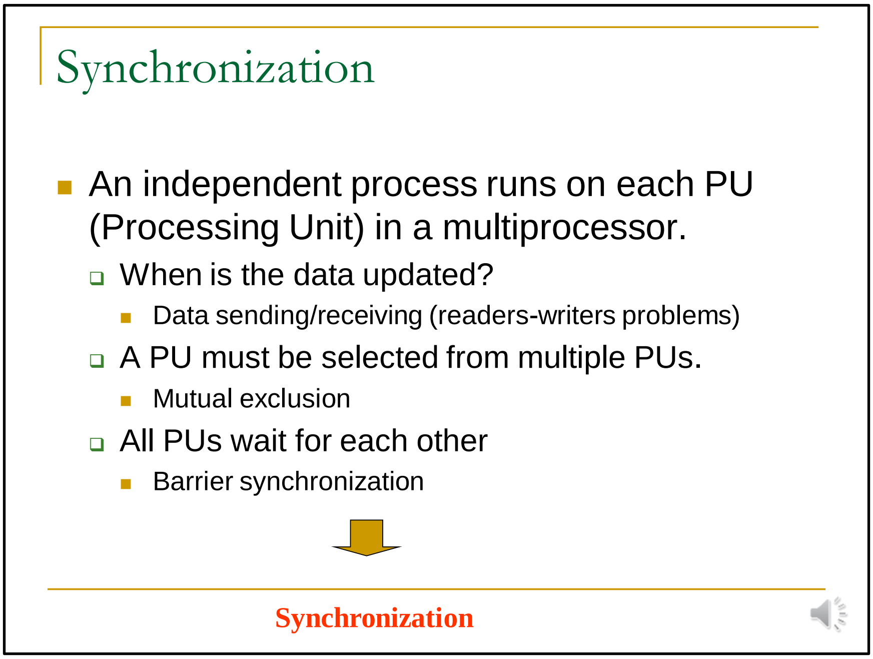
Usually, these processes (threads) can share variables

- Fork/Join is a way of synchronization
- However, frequent fork/joins degrade performance
- Operating Systems manage them

These processes communicate with each other

Synchronization is required !

First of all, let’s check how we start the parallel processing with the shared memory. Usually, a single process starts, and when it executes fork operation to generate multiple processes. Some child-process can execute fork again. After executing in parallel, all processes execute join operation. At that time, processes except for only a process which executes the fork operation are terminated. When all processes are terminated with the join operation, the total program is finished. This join operation is a kind of synchronization. For example, OpenMP which I will explain in the later class uses this method. That is, parallel programming can be done only with fork-join mechanism. But, since they need heavy overhead, we need other method for synchronization.

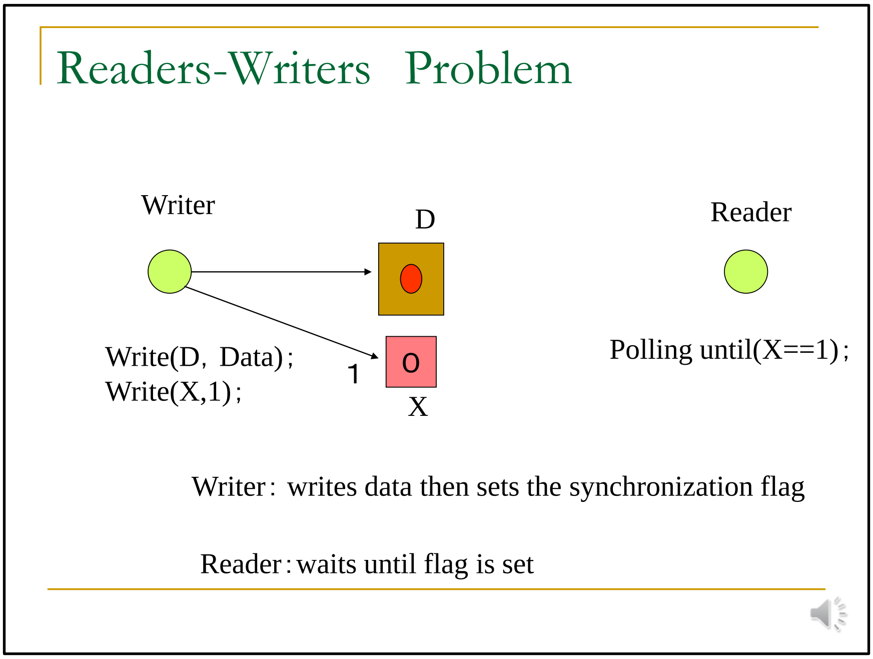


Synchronization

- An independent process runs on each PU (Processing Unit) in a multiprocessor.
  - When is the data updated?
    - Data sending/receiving (readers-writers problems)
  - A PU must be selected from multiple PUs.
    - Mutual exclusion
  - All PUs wait for each other
    - Barrier synchronization

Synchronization

In MIMD processors, an independent process runs on each processing unit. In this case, a processing unit cannot recognize when the data are written into the shared memory from other processing units. Without the synchronization method, data sending/receiving cannot be done. This is called the readers-writers problem. In some cases, a processing unit must be selected from multiple Processing Units. This is called the mutual exclusion. In other cases, all PUs must wait for others. This type of synchronization is called a barrier synchronization.



Readers-Writers Problem

Writer

D

Reader

Write(D, Data);
Write(X,1);

1

0

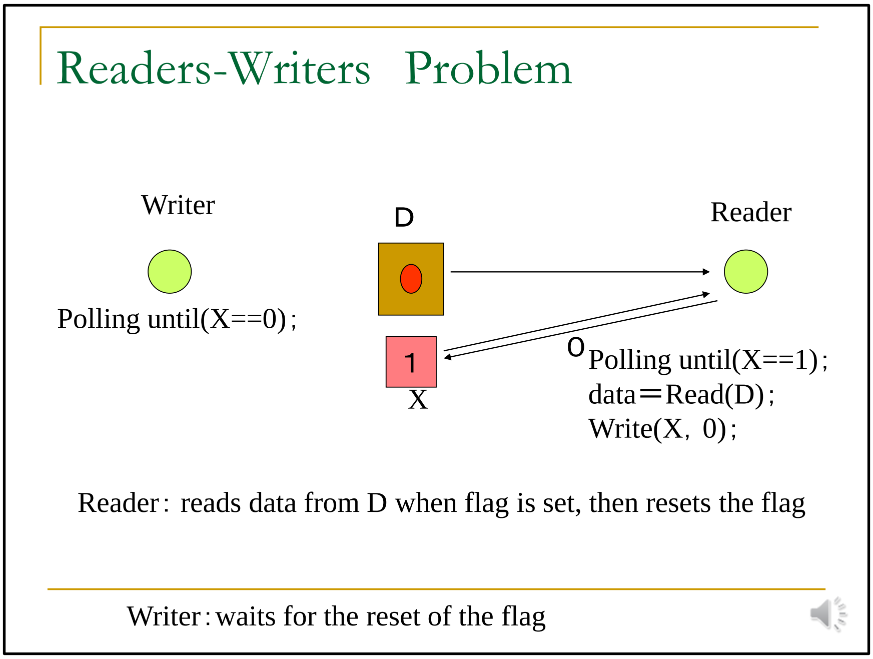
X

Polling until(X==1);

Writer: writes data then sets the synchronization flag

Reader: waits until flag is set

Let me explain the readers-writers problem, first. Assume that a writer wants to send a data block to a reader. In such a case, a writer writes a data into a region in the shared memory. After that it changes the synchronization variable or synchronization flag X into 1. Note that this variable must be initialized to 0.



Readers-Writers Problem

Writer

D

Reader

Polling until(X==0);

1

X

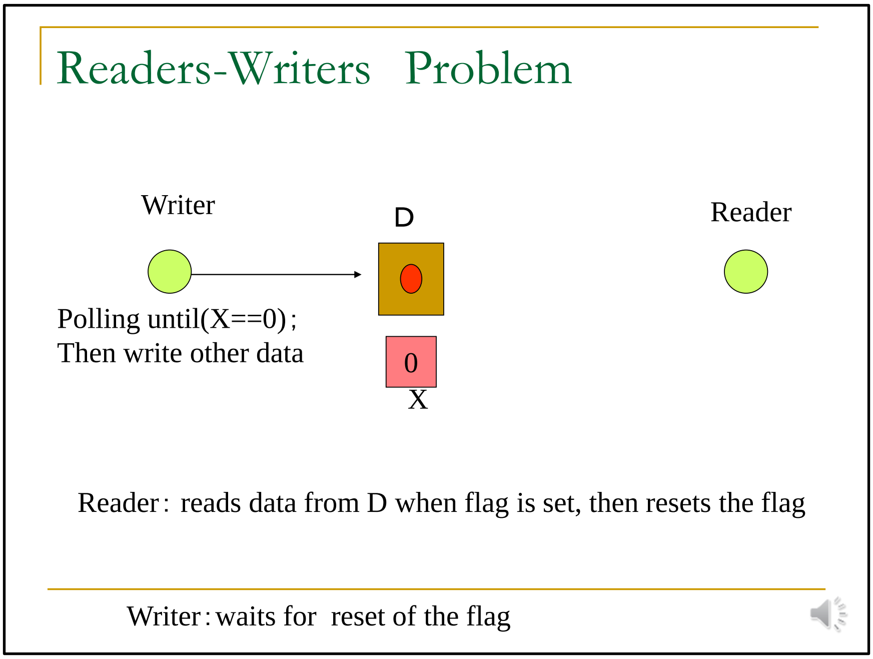
0

Polling until(X==1);
data=Read(D);
Write(X, 0);

Reader: reads data from D when flag is set, then resets the flag

Writer: waits for the reset of the flag

The reader reads X and if it is 0, it must check again. This operation is called poling or busy waiting. When the reader gets 1, it recognizes the sender has written the data. It reads the data block, then it resets the X into 0.



Readers-Writers Problem

Writer

D

Reader

Polling until(X==0);
Then write other data

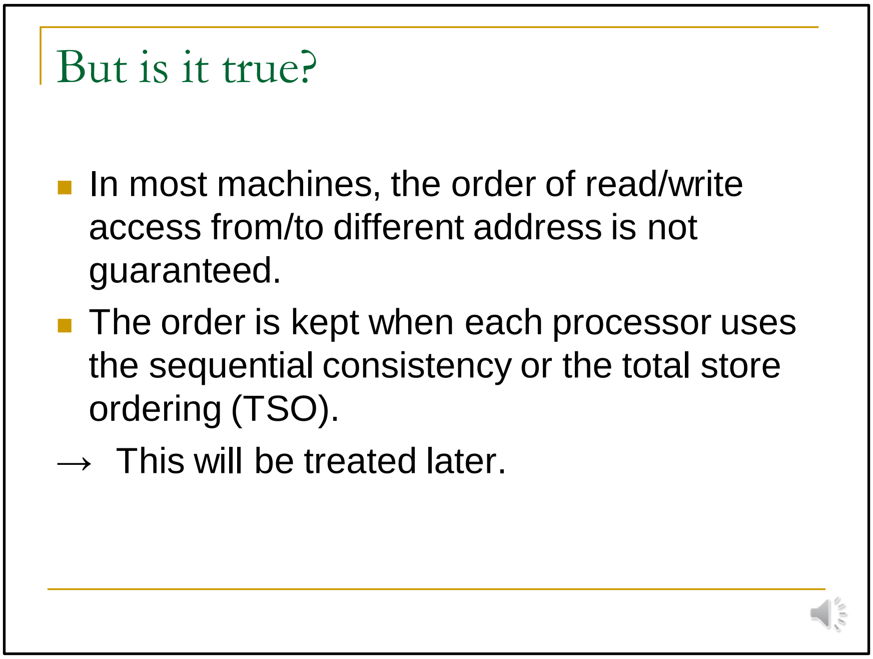
0

X

Reader: reads data from D when flag is set, then resets the flag

Writer: waits for reset of the flag

The writer checks X and if X is 0, it can write the next data if needed.

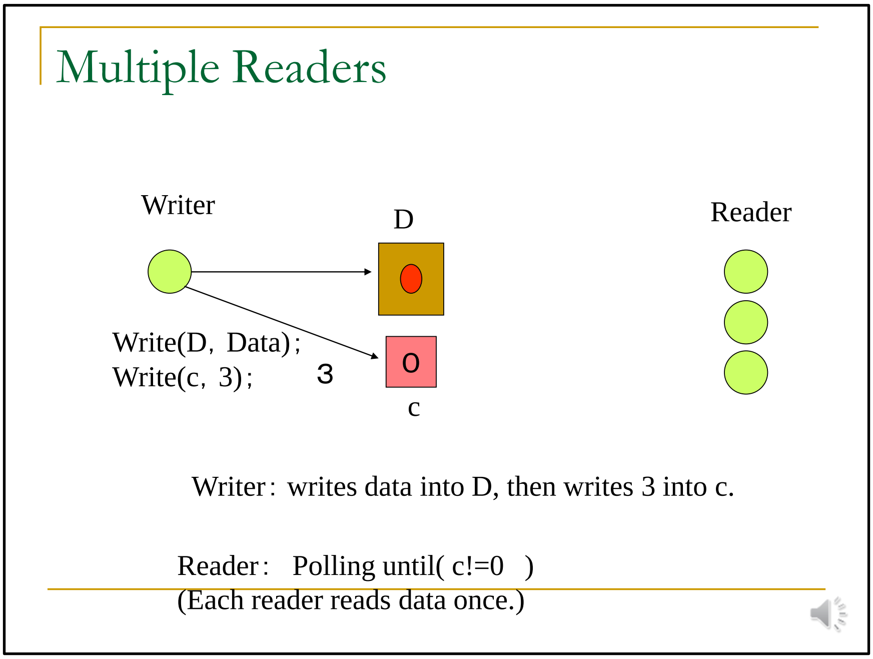


But is it true?

- In most machines, the order of read/write access from/to different address is not guaranteed.
- The order is kept when each processor uses the sequential consistency or the total store ordering (TSO).

→ This will be treated later.

This scenario is true if the order of read/write access from/to different address is guaranteed. Unfortunately, it is not guaranteed in some machines. I will introduce this subject in the next class, but when a processor uses the sequential consistency or the total store ordering (TSO), the scenario is established.



Multiple Readers

Writer

D

Reader

Write(D, Data);
Write(c, 3);

3

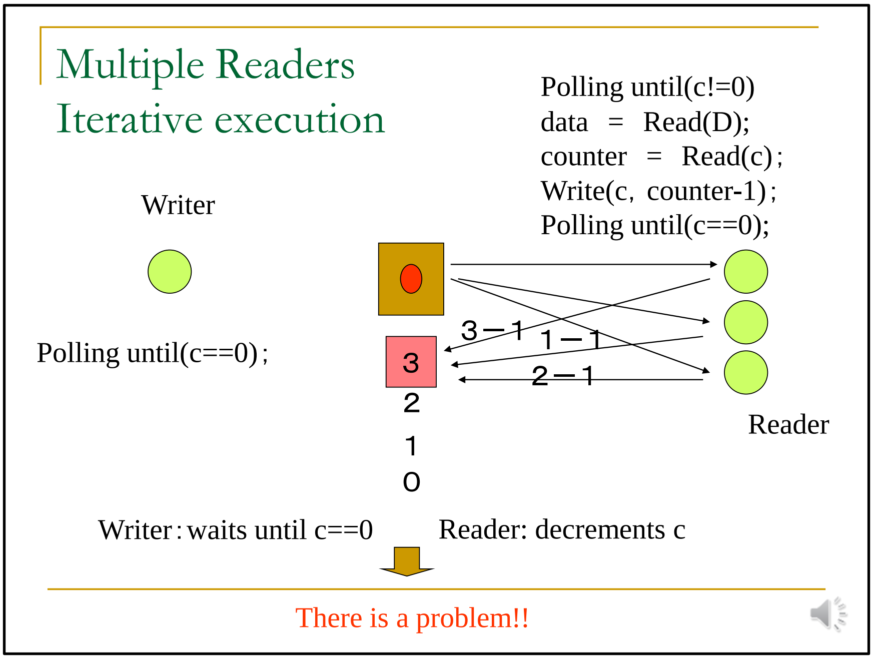
0

c

Writer: writes data into D, then writes 3 into c.

Reader: Polling until(c!=0)
(Each reader reads data once.)

Then, what happens when multiple readers want to receive the data. The simple idea is replacing the synchronization flag X into a counter. When there are three readers, after the write writes the data, it writes a number of readers, three this case, into the counter c.



Multiple Readers
Iterative execution

```
Polling until(c!=0)
data  =  Read(D);
counter  =  Read(c);
Write(c, counter-1);
Polling until(c==0);
```

Writer

Polling until(c==0);

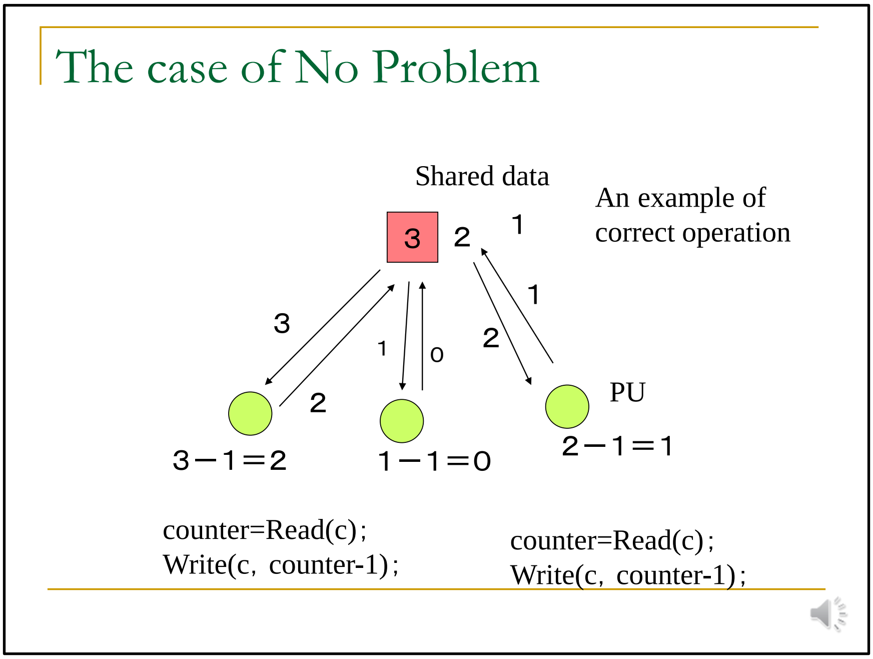
Reader

Writer: waits until c==0

Reader: decrements c

There is a problem!!

The reader can recognize that the written data are available. So, after reading the data, it decrements the counter. When all readers finish to read, the counter becomes zero. The writer waits for the counter being zero, and then it writes the next data if needed. This operation seems to work well. But, there is a problem, since it introduces the mutual exclusion problem.



The case of No Problem

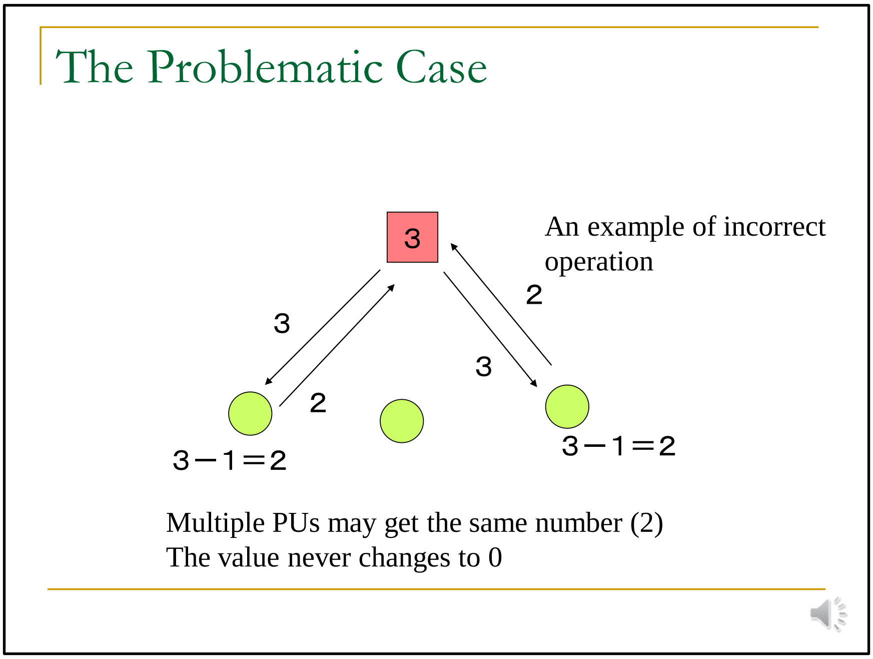
Shared data

An example of correct operation

```
counter=Read(c)；
Write(c，counter-1)；
```

```
counter=Read(c)；
Write(c，counter-1)；
```

This slide shows the case without any problem. If the Processing Unit accesses the counter in order, it does not cause any problem.

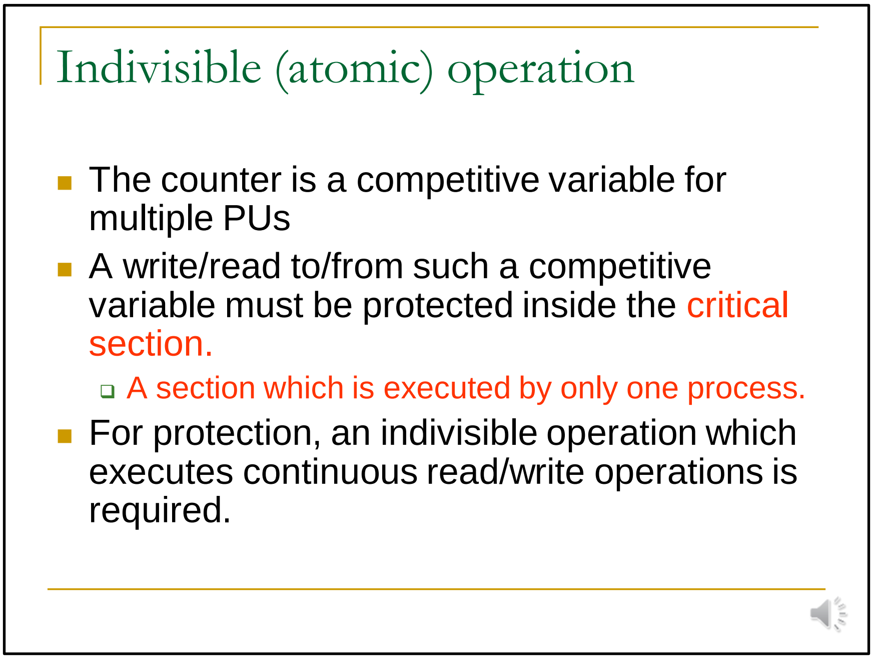


The Problematic Case

An example of incorrect operation

Multiple PUs may get the same number (2)
The value never changes to 0

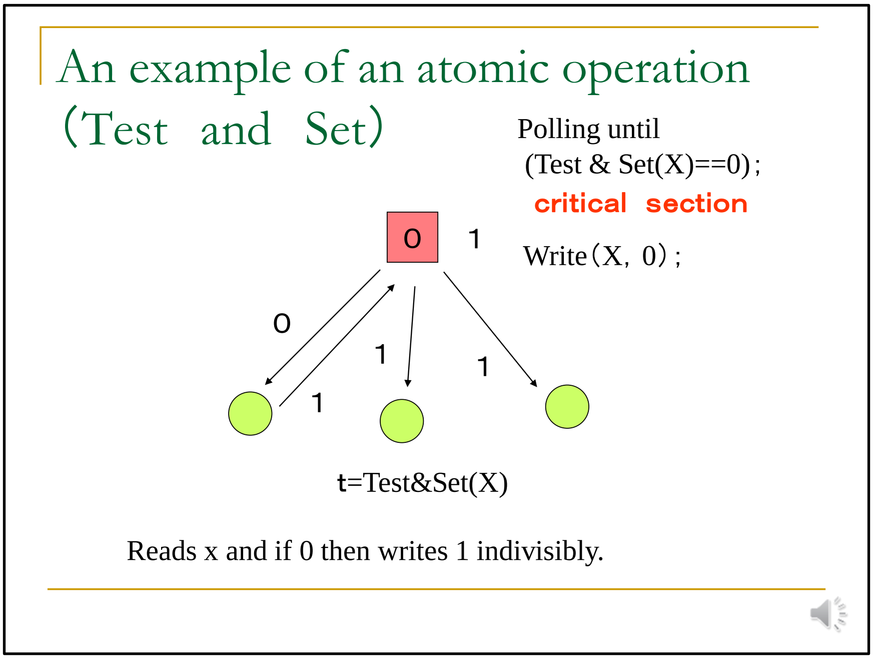
However, what happens when multiple PUs tries to access the counter almost at the same time. When this PU gets three, during it decrements the value, the next PU can read data. It means that two PUs can get the same value 3. In this case, they both write two in the counter, and it never becomes to zero.



Indivisible (atomic) operation

- The counter is a competitive variable for multiple PUs
- A write/read to/from such a competitive variable must be protected inside the critical section.
  - A section which is executed by only one process.
- For protection, an indivisible operation which executes continuous read/write operations is required.

The counter in this example, is called a competitive variable. To such a competitive variable, accesses must be done in the critical section. It means a section which is executed by only a process. This problem can be solved by introducing atomic or indivisible operation.



An example of an atomic operation （Test and Set）

Polling until
(Test & Set(X)==0)；

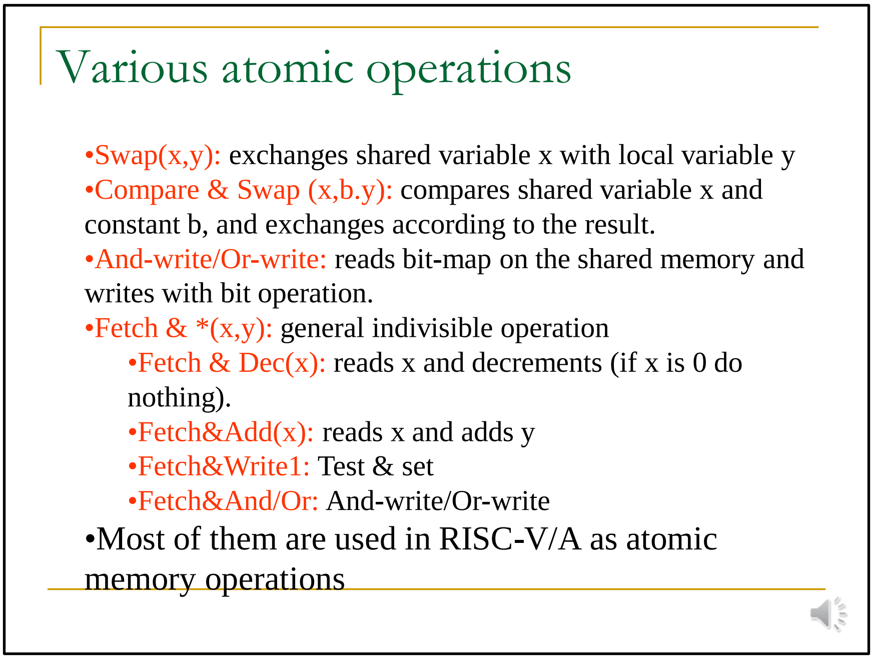
critical section

Write（X，0）；

t=Test&Set(X)

Reads x and if 0 then writes 1 indivisibly.

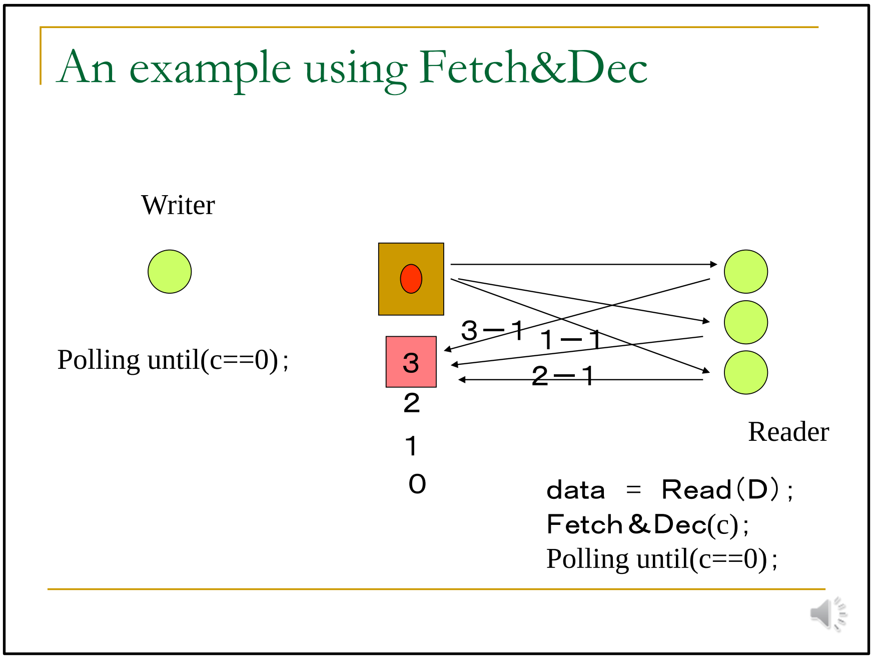
Let me explain the concept of an atomic operation with a simple example Test and Set. This operation reads a variable and writes 1 indivisibly if it is 0. The important point is that reading the data and writing 1 are executed with an action, that is, without interfering by other processors or processes. If multiple processors execute Test and Set to a variable, only one can get 0. It can execute the critical section. After doing it, it must release the critical section by writing 0. Then another processor can get 0 and enter the critical section.



Various atomic operations

- Swap(x,y): exchanges shared variable x with local variable y
- Compare & Swap (x,b.y): compares shared variable x and constant b, and exchanges according to the result.
- And-write/Or-write: reads bit-map on the shared memory and writes with bit operation.
- Fetch & *(x,y): general indivisible operation
    - Fetch & Dec(x): reads x and decrements (if x is 0 do nothing).
    - Fetch&Add(x): reads x and adds y
    - Fetch&Write1: Test & set
    - Fetch&And/Or: And-write/Or-write
- Most of them are used in RISC-V/A as atomic memory operations

Various atomic operations have been used, and most of them are used in RISC-V as atomic memory operations.



An example using Fetch&Dec

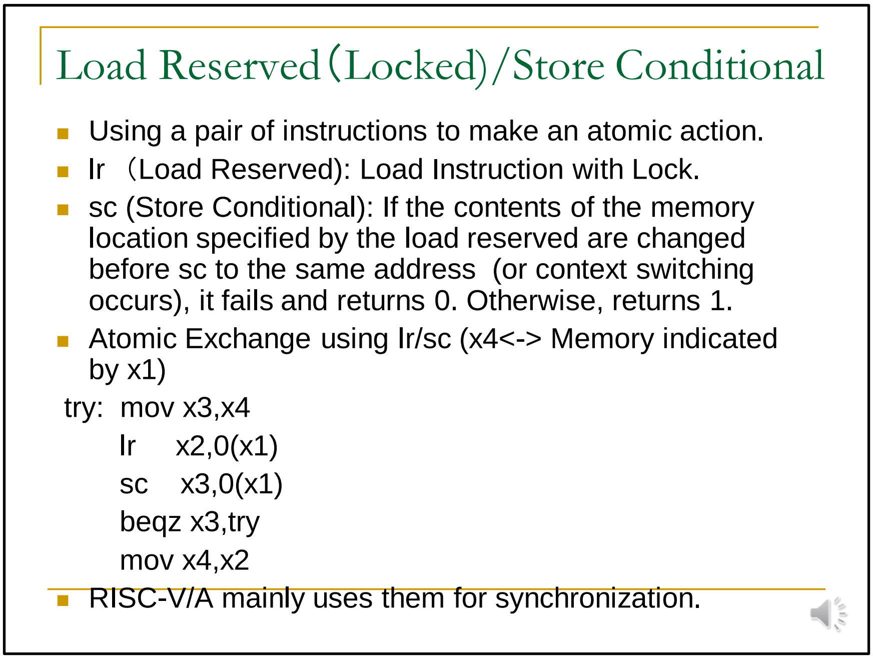
Writer

Polling until(c==0)；

Reader

data　=　Read（D）；
Fetch＆Dec(c)；
Polling until(c==0)；

The convenient operation is Fetch&Dec. It reads the data and decrement it indivisibly. The counter can be directly the target of this operation. So, multiple readers problem can be easily solved without using the critical section. The fetch and decrement has two implementations, one is saturated and the other is non-saturated. In the case of saturated, it the reading value is zero, it is not decremented anymore.

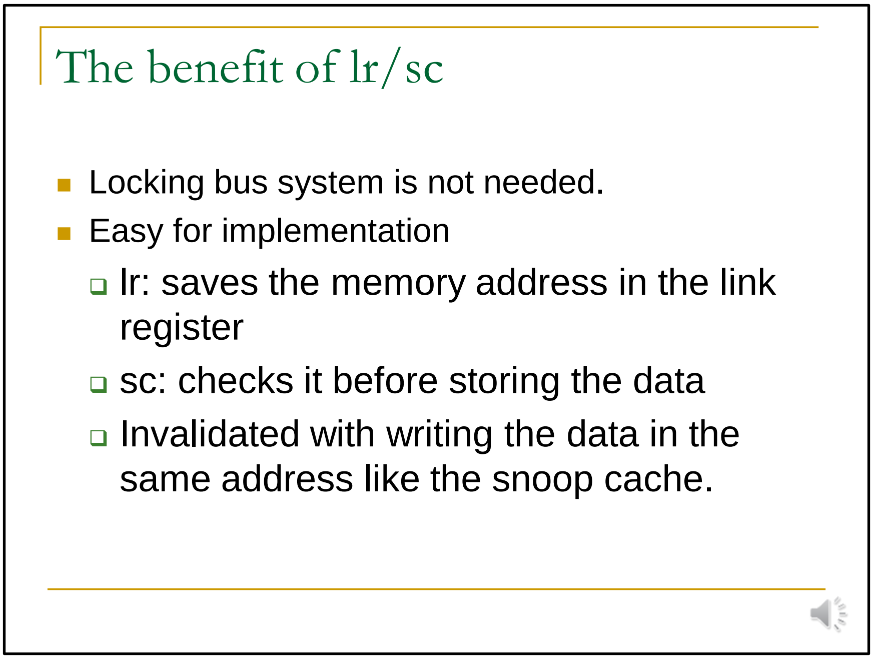


Load Reserved (Locked)/Store Conditional

- Using a pair of instructions to make an atomic action.
- lr （Load Reserved): Load Instruction with Lock.
- sc (Store Conditional): If the contents of the memory location specified by the load reserved are changed before sc to the same address (or context switching occurs), it fails and returns 0. Otherwise, returns 1.
- Atomic Exchange using lr/sc (x4<-> Memory indicated by x1)

```
try:  mov x3,x4
      lr    x2,0(x1)
      sc    x3,0(x1)
      beqz x3,try
      mov x4,x2
```

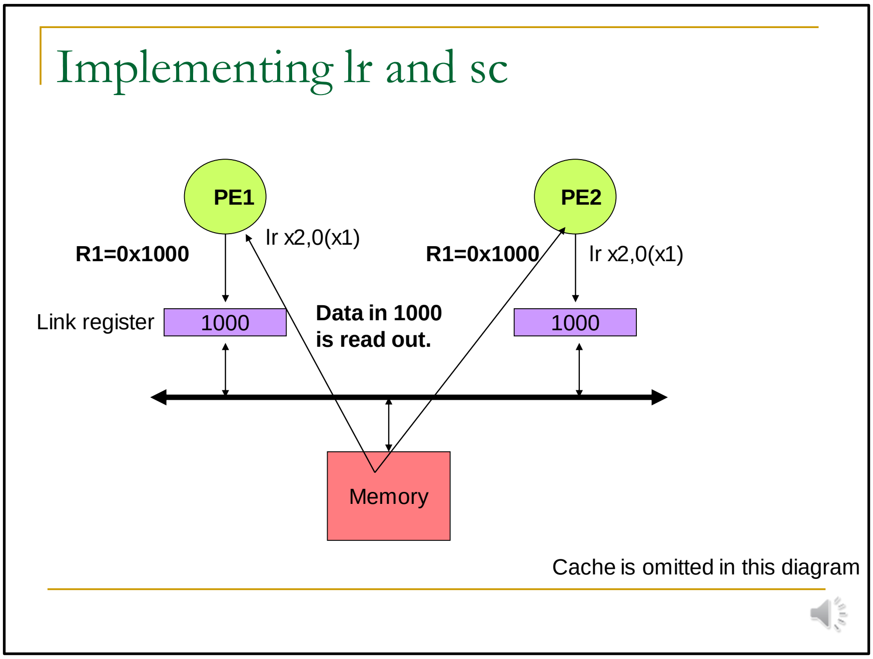
- RISC-V/A mainly uses them for synchronization.



The benefit of lr/sc

- Locking bus system is not needed.
- Easy for implementation
  - lr: saves the memory address in the link register
  - sc: checks it before storing the data
  - Invalidated with writing the data in the same address like the snoop cache.

This pair style of synchronization operations load reserved / store conditional has the following benefits.

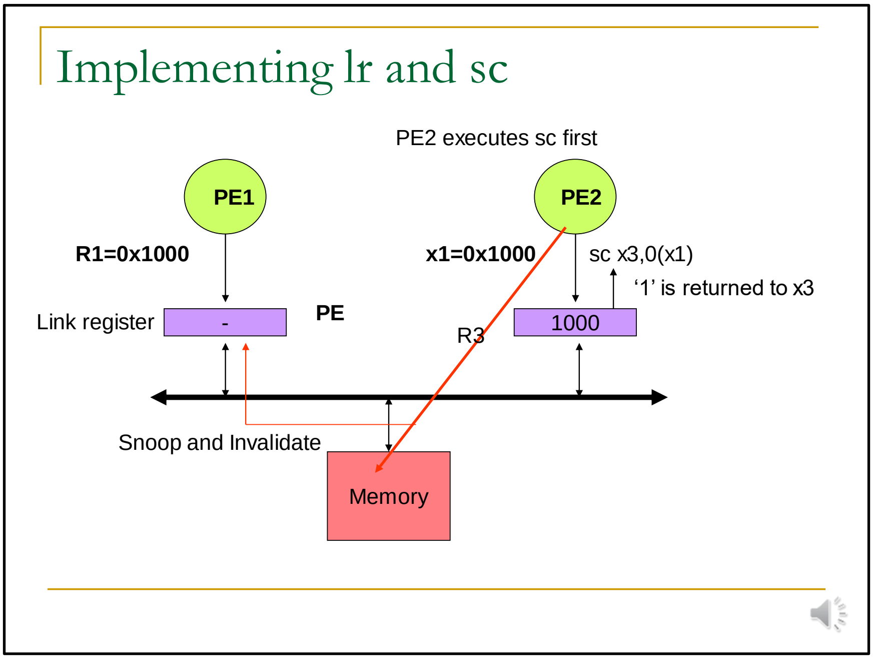


Implementing lr and sc

Data in 1000 is read out.

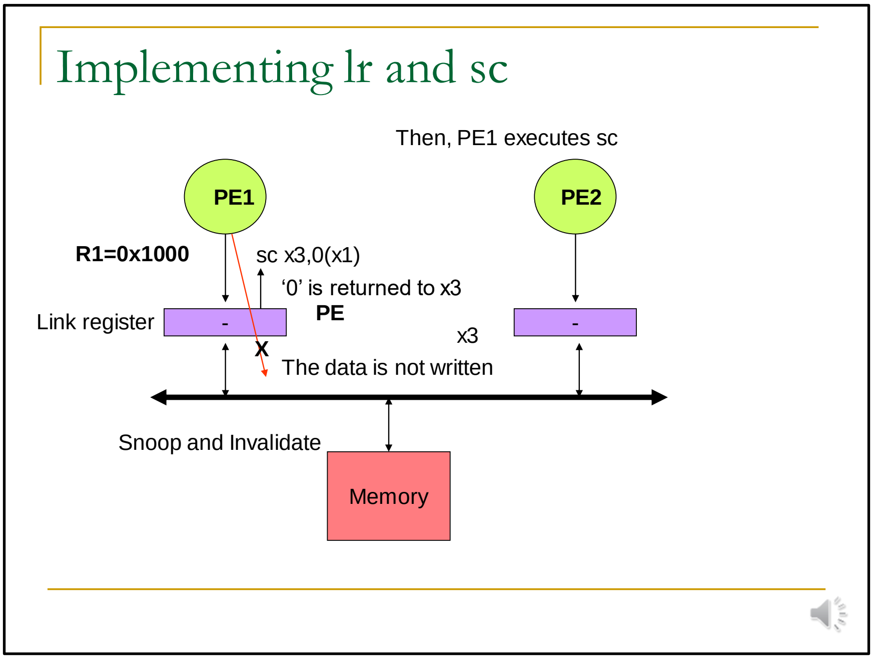
Cache is omitted in this diagram

When lr is executed, the address is set in the link register attached to each processing element.



Implementing lr and sc

When a PE executes store conditional, the link register is cleared if the address matches. If the link register is alive, it can get 1.



Implementing lr and sc

Then, PE1 executes sc

PE1

PE2

R1=0x1000

sc x3,0(x1)

‘0’ is returned to x3

PE

Link register

-

-

x3

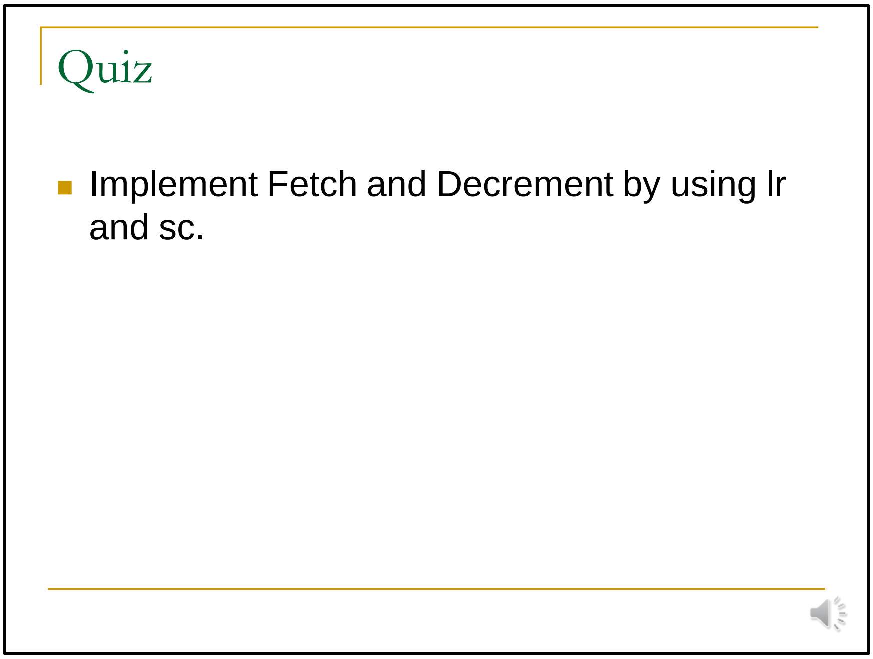
X

The data is not written

Snoop and Invalidate

Memory

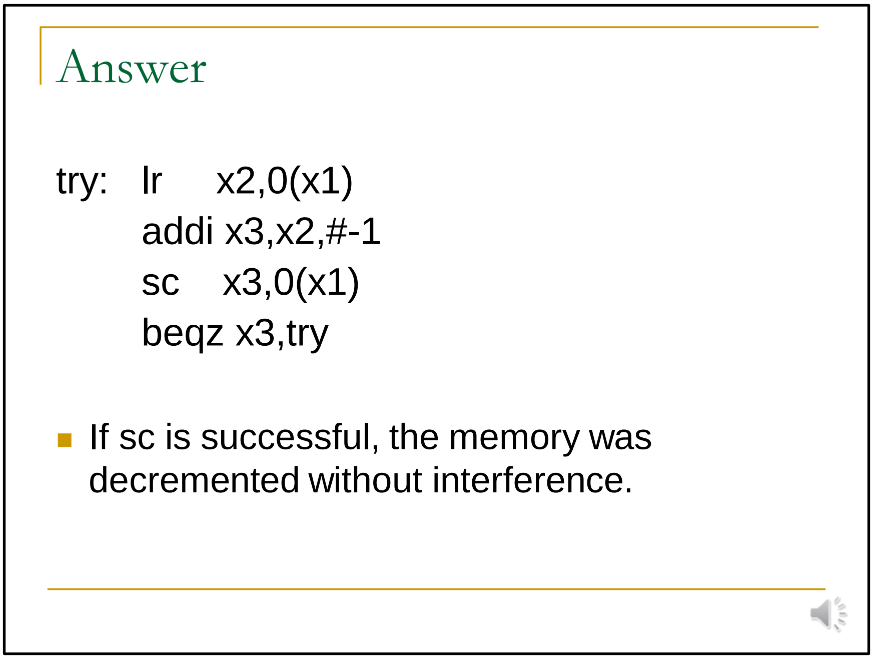
When the other PE executes store conditional operation, it gets 0 since the link register is cleared.



Quiz

- Implement Fetch and Decrement by using lr and sc.

An atomic memory operations can be implanted with a combination of lr and sc.

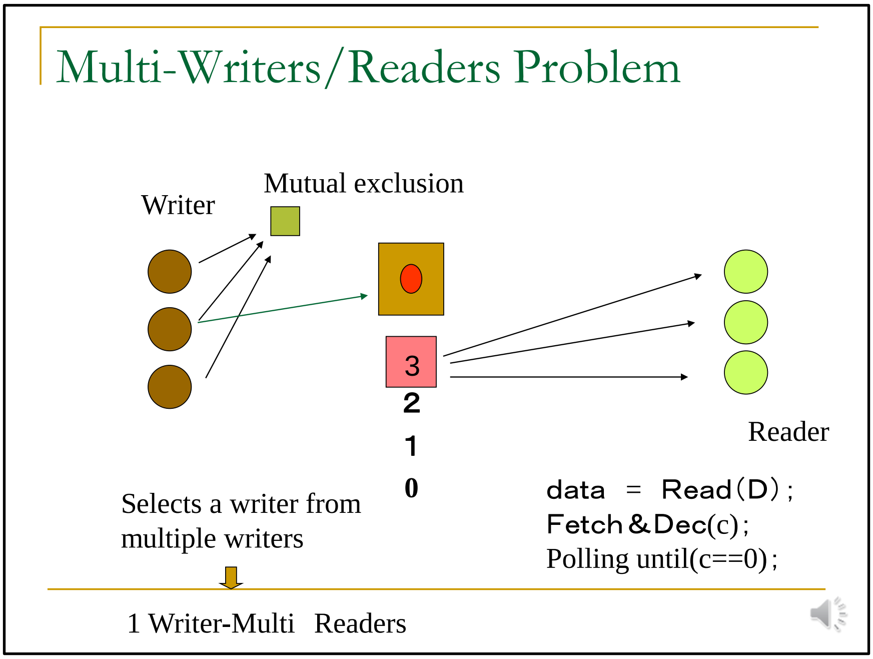


Answer

```
try:   lr     x2,0(x1)
       addi x3,x2,#-1
       sc     x3,0(x1)
       beqz x3,try
```

- If sc is successful, the memory was decremented without interference.

This is an answer, if sc is successful, it means that the memory was decremented without interference.



Multi-Writers/Readers Problem

Writer

Mutual exclusion

3

2

1

0

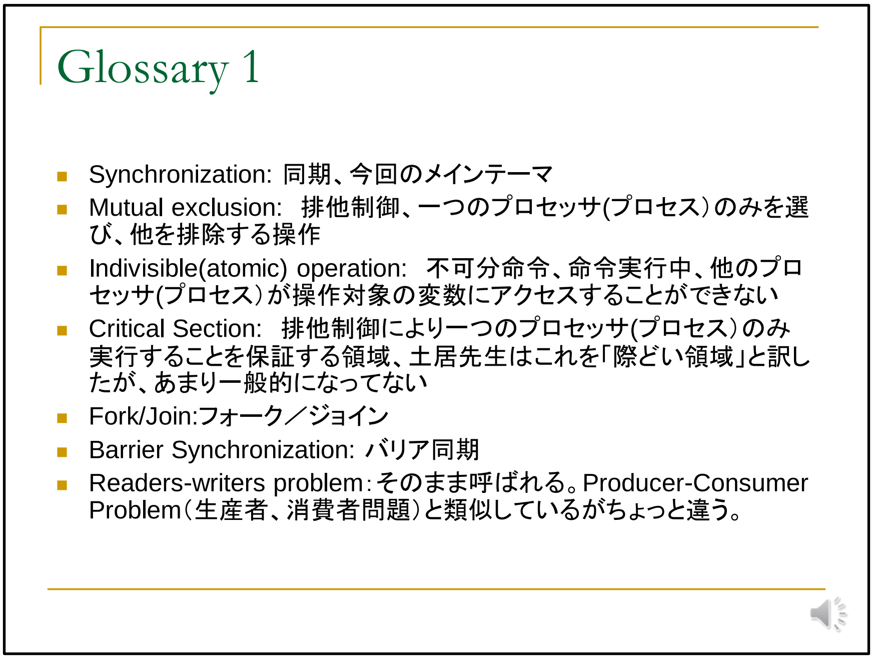
Reader

data　=　Read（D）；
Fetch＆Dec(c)；
Polling until(c==0)；

Selects a writer from multiple writers

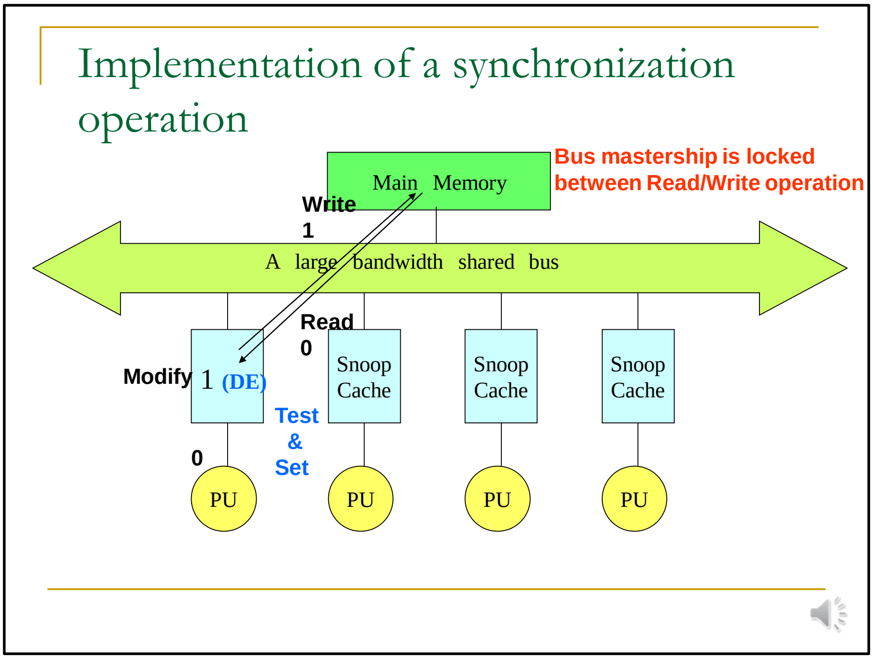
1 Writer-Multi　Readers

Once the critical section can be used, multi-writers problem can be easily solved. First, a writer is selected with a mutual exclusion operation by using Test and Set for example. Then, the writer executes the single-writer multiple readers problem.



Glossary 1

- Synchronization: 同期、今回のメインテーマ
- Mutual exclusion: 排他制御、一つのプロセッサ(プロセス)のみを選び、他を排除する操作
- Indivisible(atomic) operation: 不可分命令、命令実行中、他のプロセッサ(プロセス)が操作対象の変数にアクセスすることができない
- Critical Section: 排他制御により一つのプロセッサ(プロセス)のみ実行することを保証する領域、土居先生はこれを「際どい領域」と訳したが、あまり一般的になってない
- Fork/Join:フォーク／ジョイン
- Barrier Synchronization: バリア同期
- Readers-writers problem：そのまま呼ばれる。Producer-Consumer Problem（生産者、消費者問題）と類似しているがちょっと違う。



Implementation of a synchronization operation

Main Memory

Bus mastership is locked between Read/Write operation

Write 1

A large bandwidth shared bus

Read 0

Modify 1 (DE)

Snoop Cache

Snoop Cache

Snoop Cache

Test & Set

0

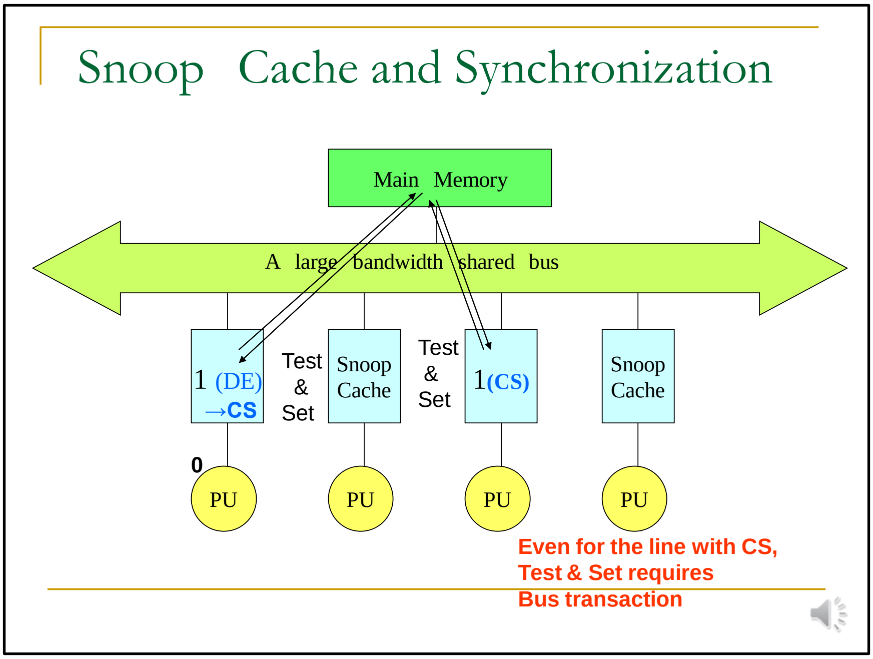
PU

PU

PU

PU

Next, let me explain how the synchronization operations are implemented on the system with private or snoop cache. For example, when Test and Set is implemented on a multi-core system with snoop cache that I explained in the previous class, it will read the main memory and write it locking the bus. Apparently, the value becomes DE since this operation accompanies the write operation.



Snoop Cache and Synchronization

Main Memory

A large bandwidth shared bus

1 (DE) →CS

Test & Set

Snoop Cache

Test & Set

1(CS)

Snoop Cache

0

PU

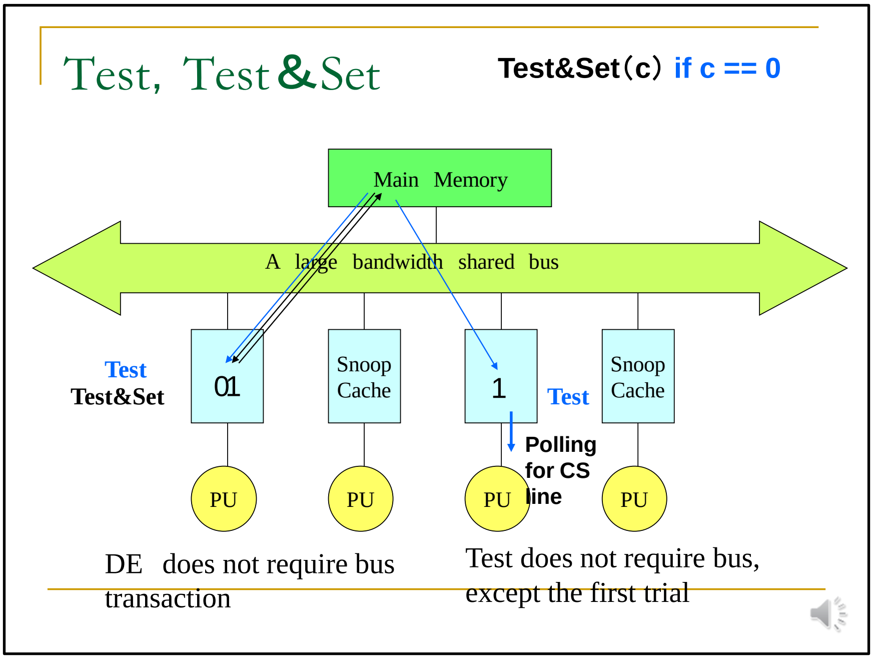
PU

PU

PU

Even for the line with CS, Test & Set requires Bus transaction

If multiple PUs try busy waiting, bus will be congested because multiple PUs try to access the bus for the Test & Set operations.



Test, Test&Set

Test&Set(c) if c == 0

Main Memory

A large bandwidth shared bus

Test
Test&Set

01

Snoop
Cache

1

Test

Snoop
Cache

Polling
for CS
line

PU

PU

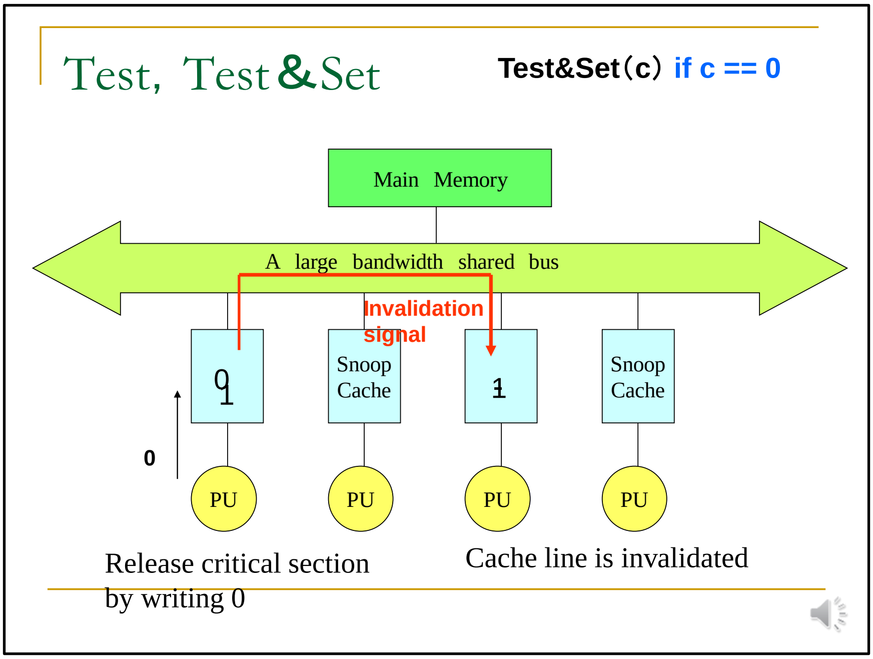
PU

PU

DE does not require bus transaction

Test does not require bus, except the first trial

Just by reading the synchronization variable before executing the atomic operation, we can avoid this bus congestion. It is called test test & set. When reading data from the main memory is 1, it is hopeless to try the test and set. In this case, try just reading again.



Test, Test&Set

Test&Set(c) if c == 0

Main Memory

A large bandwidth shared bus

Invalidation signal

0
1

Snoop Cache

1

Snoop Cache

0

PU

PU

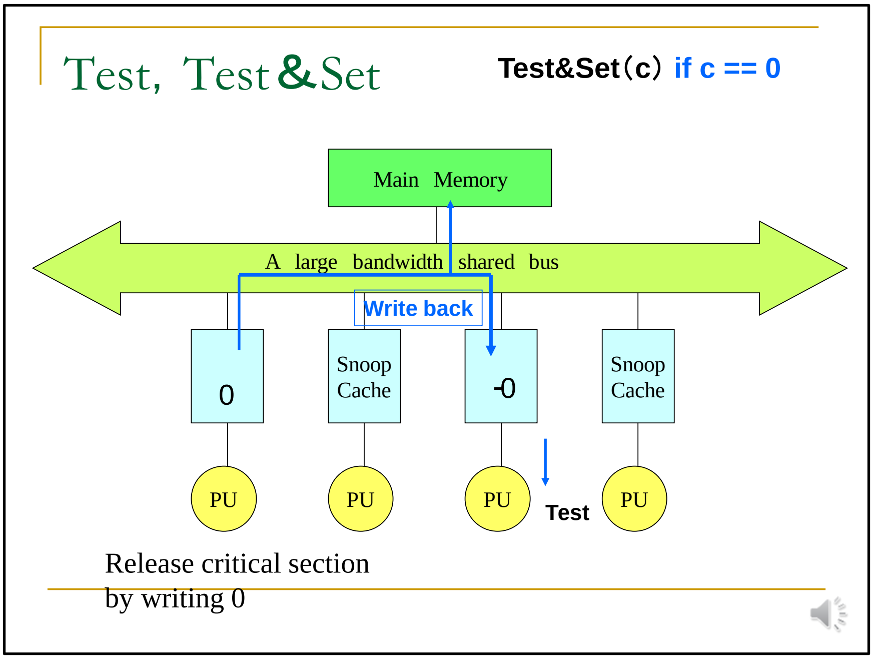
PU

PU

Release critical section by writing 0

Cache line is invalidated

After executing the critical section, a PU writes 0 to release the critical section. At that time, the invalidation signal is transferred on the bus, and other copies are invalidated.



Test, Test&Set

Test&Set(c) if c == 0

Main Memory

A large bandwidth shared bus

Write back

0

Snoop Cache

-0

Snoop Cache

PU

PU

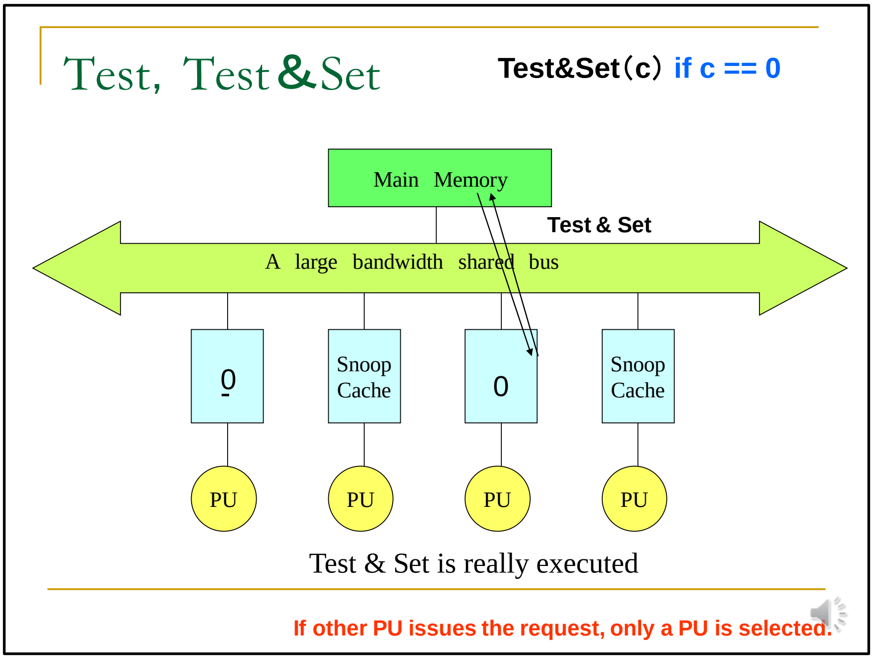
PU

Test

PU

Release critical section
by writing 0

When other PUs reads the synchronization variable. Since it was invalidated, it causes cache-miss, and the value 0 is transferred from the cache which released the critical section. Since the requesting PU gets 0, it executes Test&Set operation. Of course, there may be multiple PUs which get the value 0, but they can be resolved by executing Test&Set.



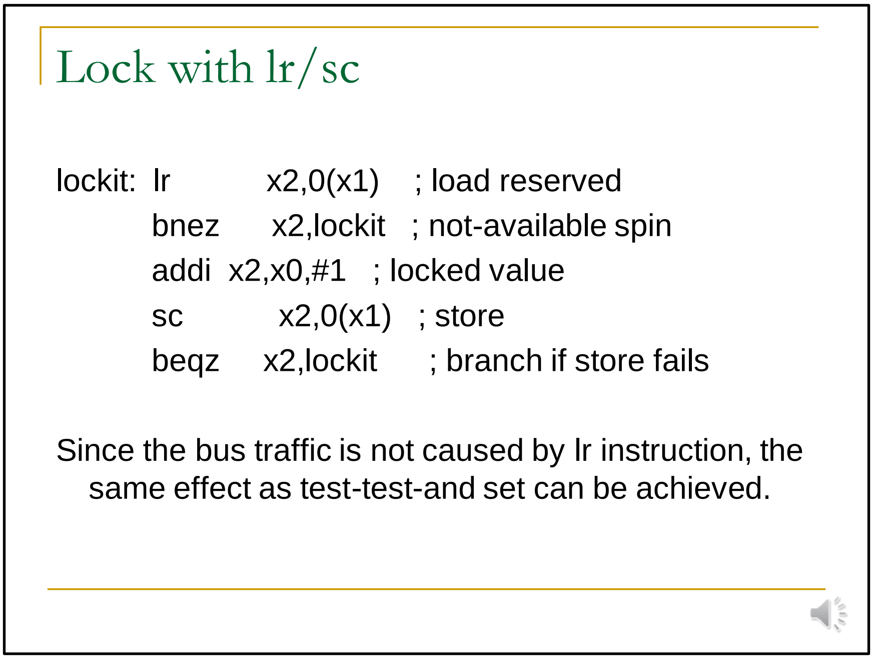
Test, Test&Set

Test&Set(c) if c == 0

Test & Set is really executed

If other PU issues the request, only a PU is selected.

As a result, only a PU can be selected.

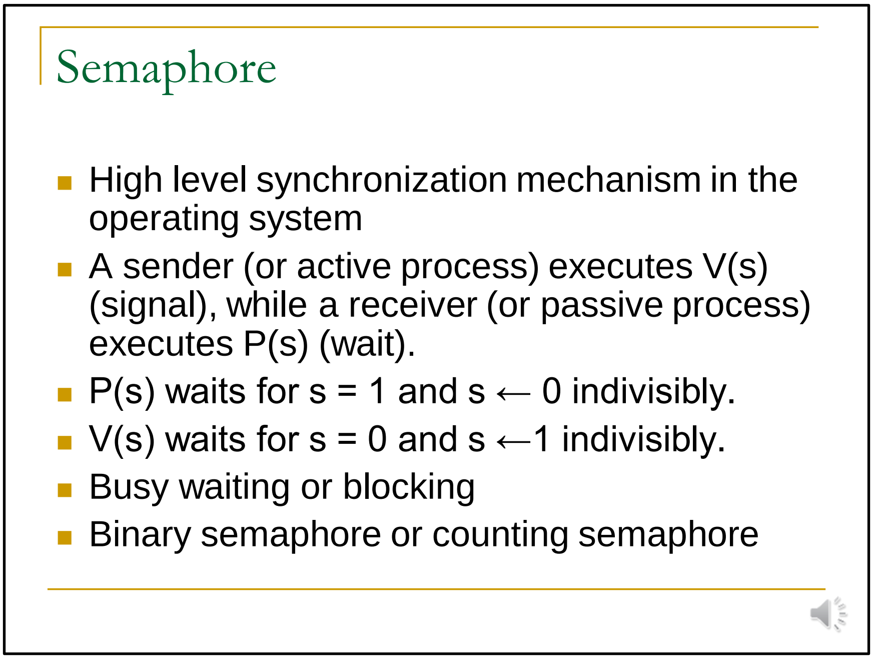


Lock with lr/sc

```
lockit: lr        x2,0(x1)    ; load reserved
        bnez      x2,lockit   ; not-available spin
        addi  x2,x0,#1   ; locked value
        sc        x2,0(x1)    ; store
        beqz      x2,lockit      ; branch if store fails
```

Since the bus traffic is not caused by lr instruction, the same effect as test-test-and set can be achieved.

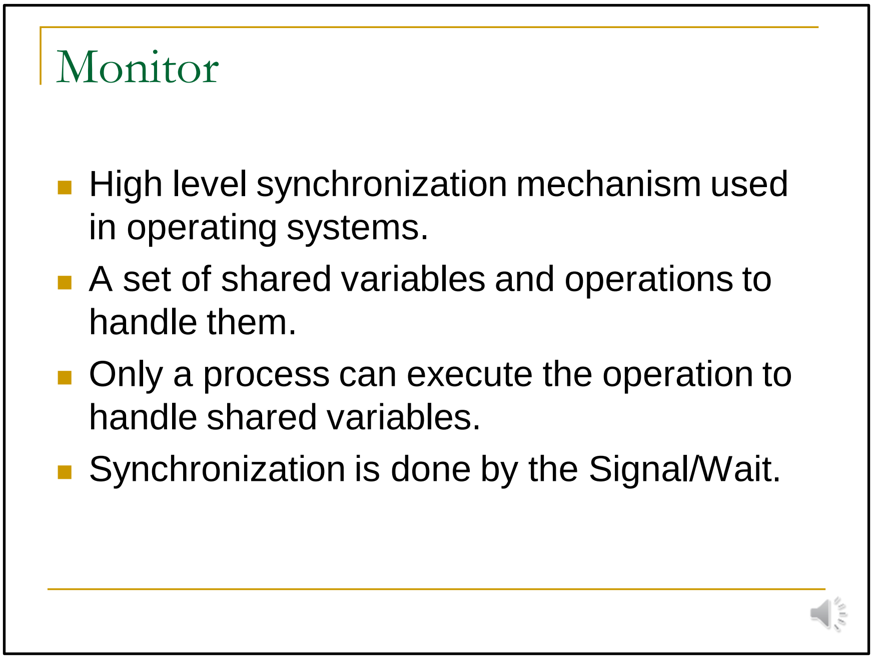
This code is a lock operation by using lr and sc. Since the bus traffic is not caused by lr instruction, the same effect as test-test-and-set can be achieved.



Semaphore

- High level synchronization mechanism in the operating system
- A sender (or active process) executes V(s) (signal), while a receiver (or passive process) executes P(s) (wait).
- P(s) waits for s = 1 and s ← 0 indivisibly.
- V(s) waits for s = 0 and s ←1 indivisibly.
- Busy waiting or blocking
- Binary semaphore or counting semaphore

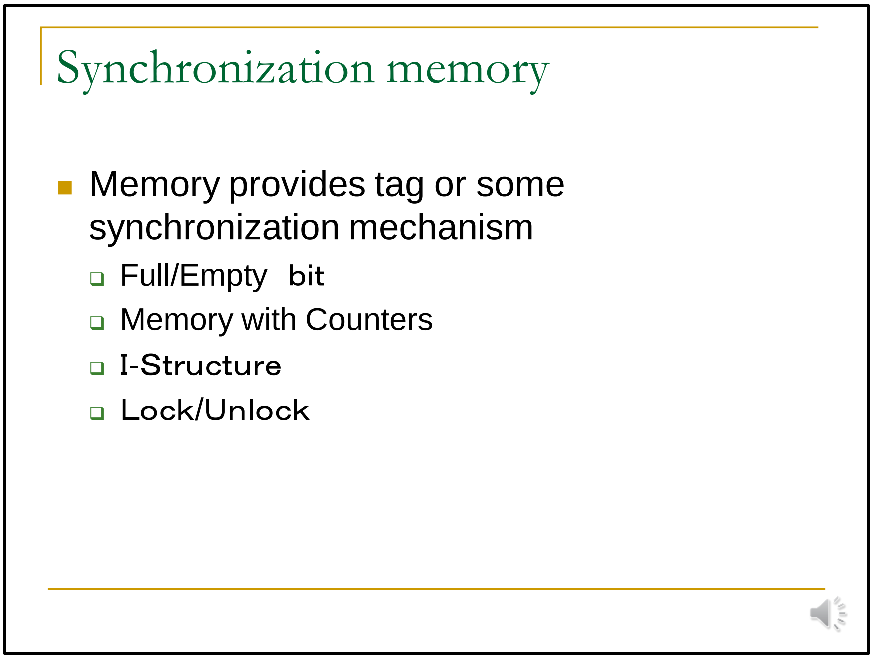
I have introduced basic synchronization mechanisms for multi-cores, but the synchronization operation is needed for operating system which handles concurrent processing of multiple processes. That is, concurrent processes must be treated as the same way of multiprocessors. However, since synchronization operations are implemented mainly with software, sophisticated mechanisms are used. Semaphore is famous synchronization mechanism.



Monitor

- High level synchronization mechanism used in operating systems.
- A set of shared variables and operations to handle them.
- Only a process can execute the operation to handle shared variables.
- Synchronization is done by the Signal/Wait.

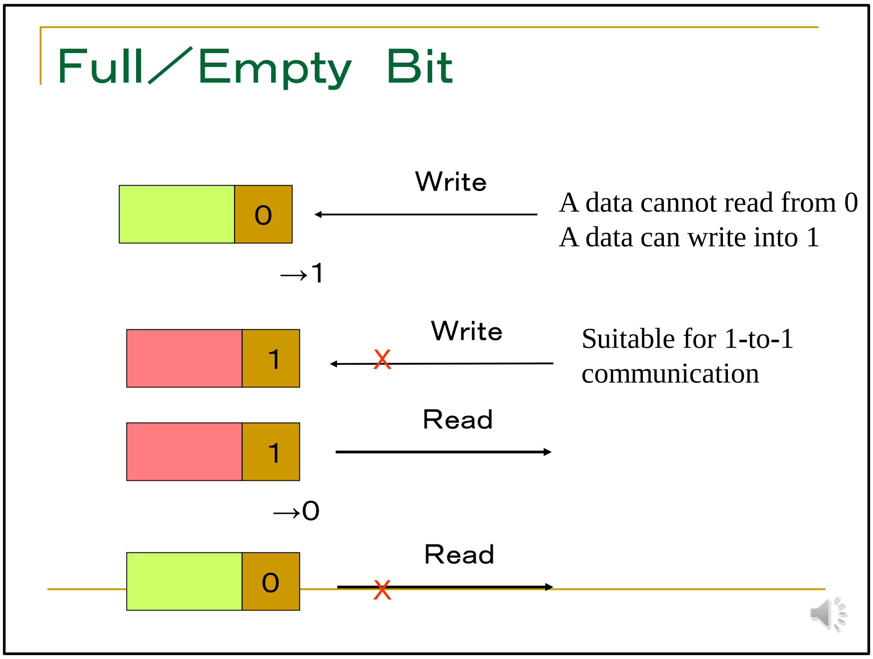
Monitor is also famous example of synchronization. For the safe operation, a set of shared variables and operations to handle them are defined.



Synchronization memory

- Memory provides tag or some synchronization mechanism
  - Full/Empty bit
  - Memory with Counters
  - I-Structure
  - Lock/Unlock

Next, let me explain the synchronization memory. Instead of providing atomic operation, the synchronization mechanism can be provided on the memory.

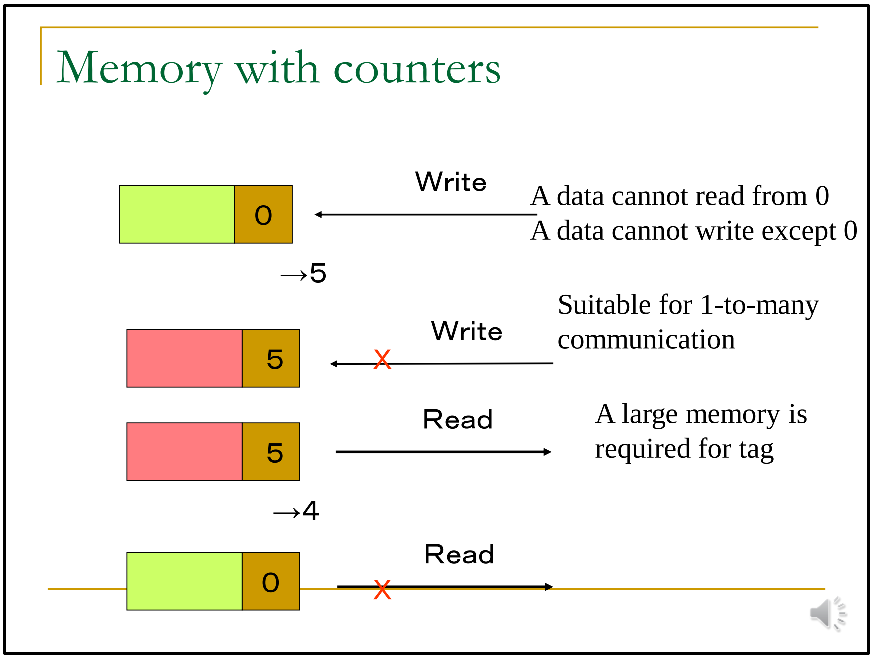


Full／Empty Bit

A data cannot read from 0
A data can write into 1

Suitable for 1-to-1
communication

Full/Empty bit is the simplest one. There is a flag for each word of the memory.



Memory with counters

0

Write

A data cannot read from 0

A data cannot write except 0

→5

5

Write

Suitable for 1-to-many communication

5

Read

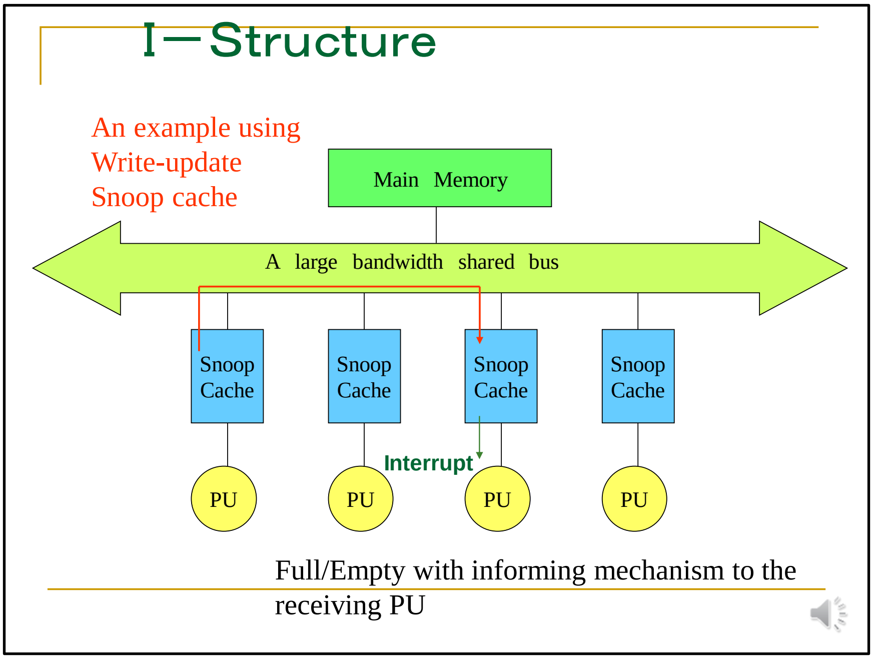
A large memory is required for tag

→4

0

Read

Instead of providing the flag, we can provide the counter.

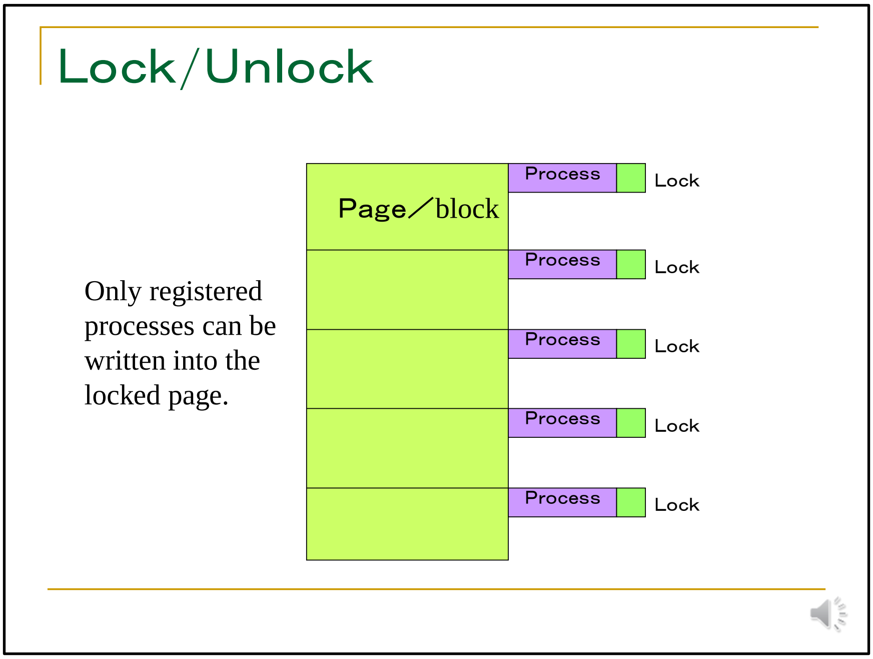


I－Structure

An example using
Write-update
Snoop cache

Full/Empty with informing mechanism to the receiving PU

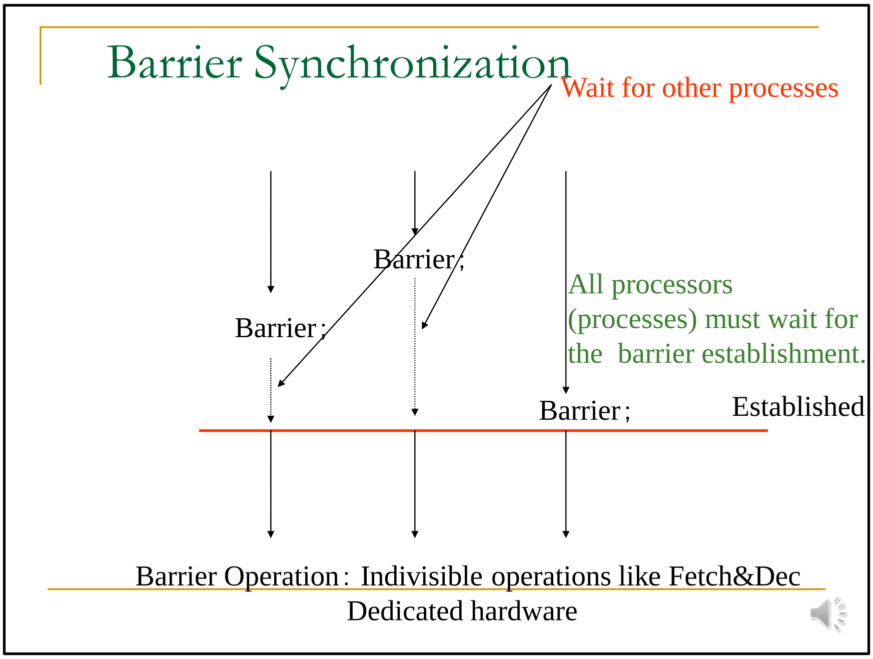
I-structure is a mechanism with informing system with the synchronization memory. This slide shows an example of implementing write-update style snoop cache. This idea was proposed for data flow machines.



Lock/Unlock

Only registered processes can be written into the locked page.

Lock/unlock register can be provided for a certain block of the memory, for example, a page or block of the cache. When a process can write the page when lock bit is not set. At the first write, the lock bit is set and the process number is registered. Only the process whose id matches the register can write the block.



Barrier Synchronization

Wait for other processes

Barrier;

Barrier;

All processors (processes) must wait for the barrier establishment.

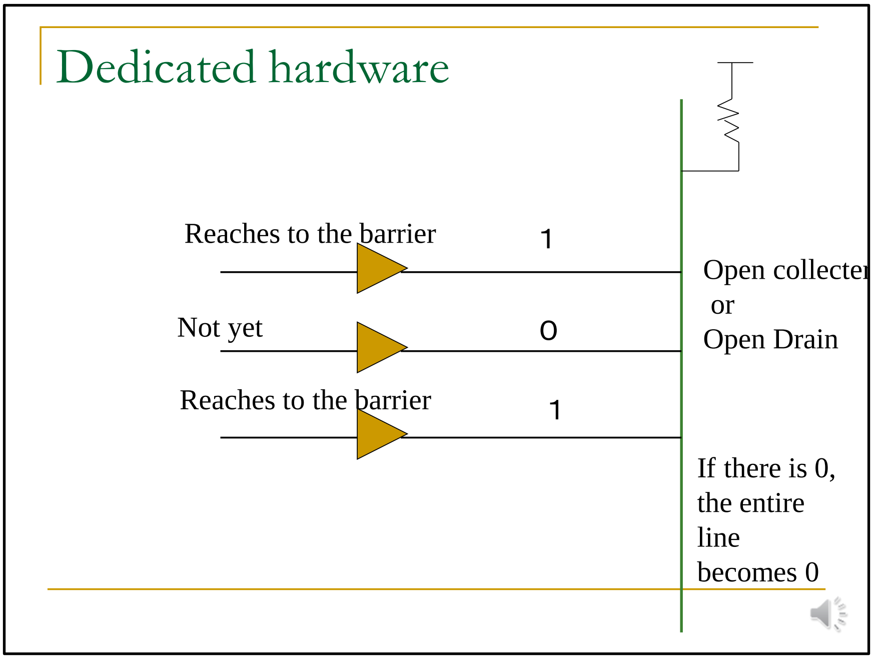
Barrier;

Established

Barrier Operation: Indivisible operations like Fetch&Dec

Dedicated hardware

Barrier synchronization is a simple and easy to use. When a processor executes a barrier operation, it must wait other processors execute the barrier operation. When all processors execute the barrier operation, they can go forward. When multiple processors compute different part of a matrix with iteration, after writing results, they execute the barrier operation. When the barrier is established, all results are available, so they can go to the next iteration. Barrier operation can be implemented with atomic instructions, but since it is so popularly used, the special hardware is sometimes provided.



Dedicated hardware

Reaches to the barrier

1

Not yet

0

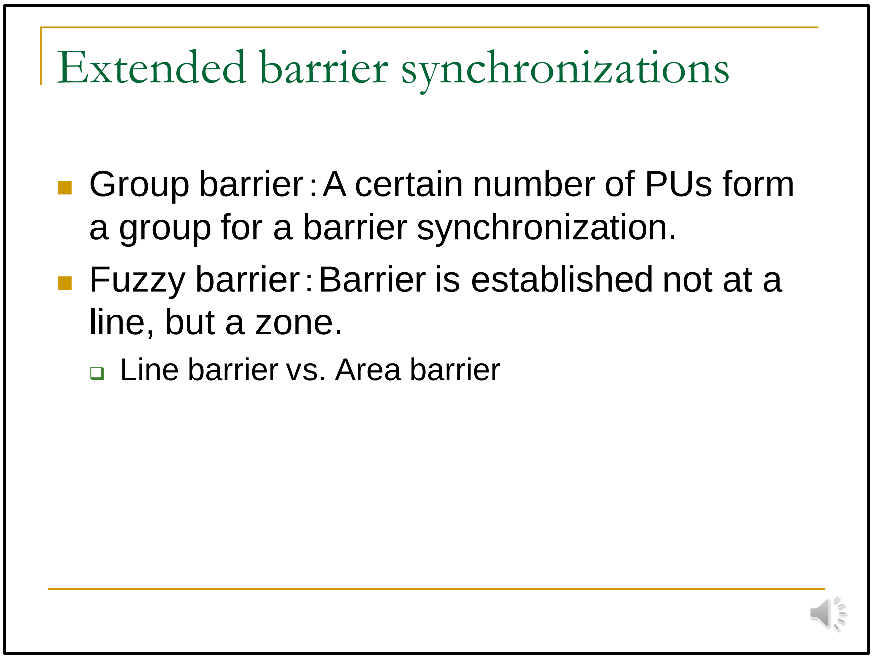
Reaches to the barrier

1

Open collecter
or
Open Drain

If there is 0,
the entire
line
becomes 0

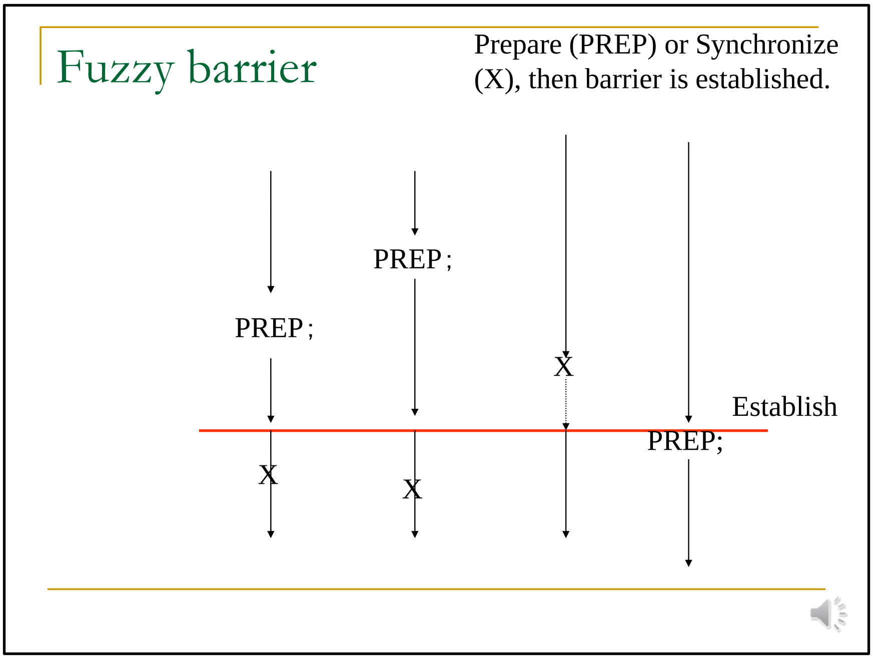
This is an example of the simplest barrier implementation. Only a wire for the open drain output is enough. It is also useful for the debugging.



Extended barrier synchronizations

- Group barrier：A certain number of PUs form a group for a barrier synchronization.
- Fuzzy barrier：Barrier is established not at a line, but a zone.
  - Line barrier vs. Area barrier

Extended barrier synchronizations have been proposed.



Fuzzy barrier

Prepare (PREP) or Synchronize (X), then barrier is established.

PREP ;

PREP ;

X

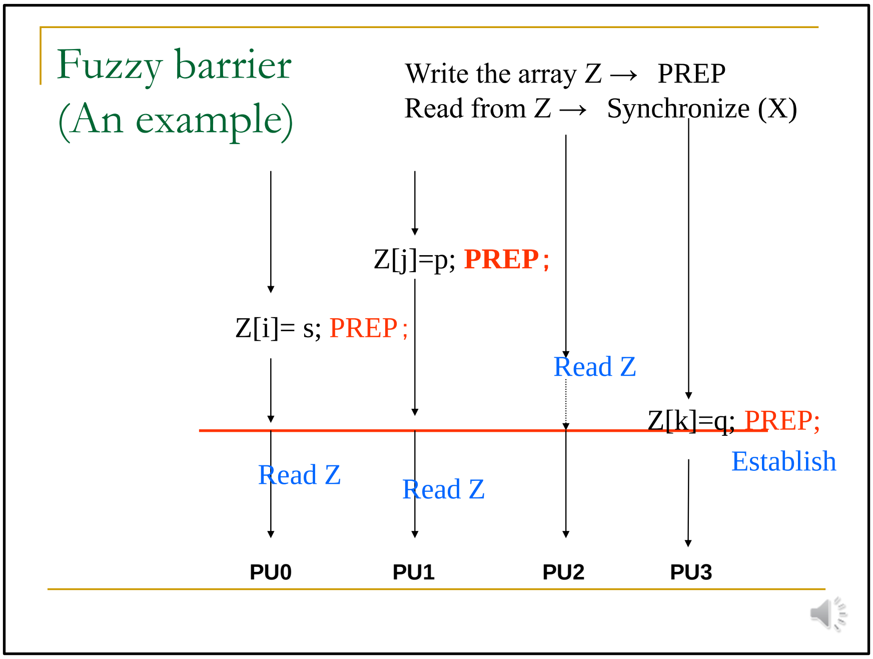
Establish

PREP;

X

X

This diagram explains the fuzzy barrier.



Fuzzy barrier (An example)

Write the array Z → PREP
Read from Z → Synchronize (X)

Z[j]=p; **PREP ;**

Z[i]= s; PREP ;

Read Z

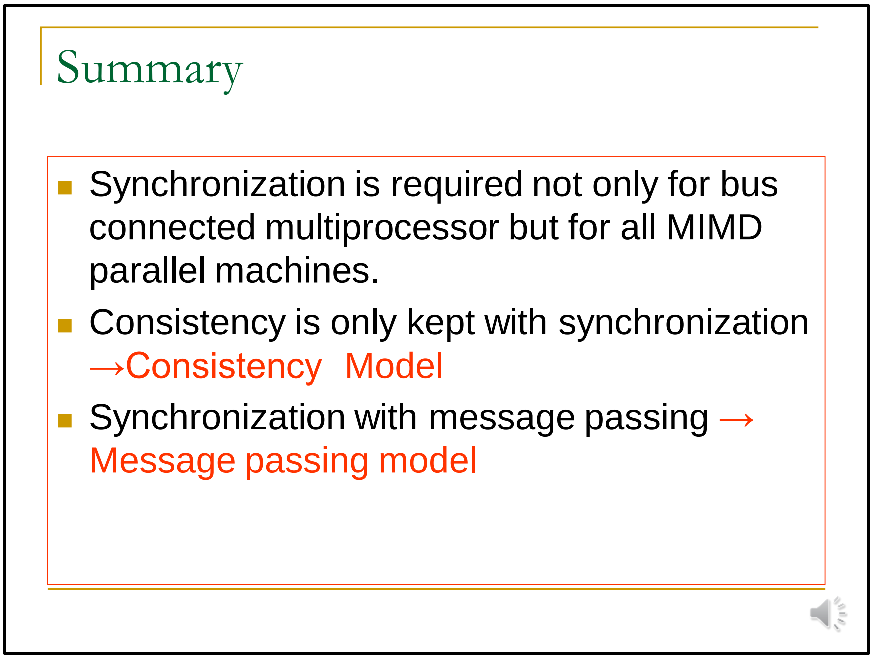
Z[k]=q; PREP;

Establish

Read Z

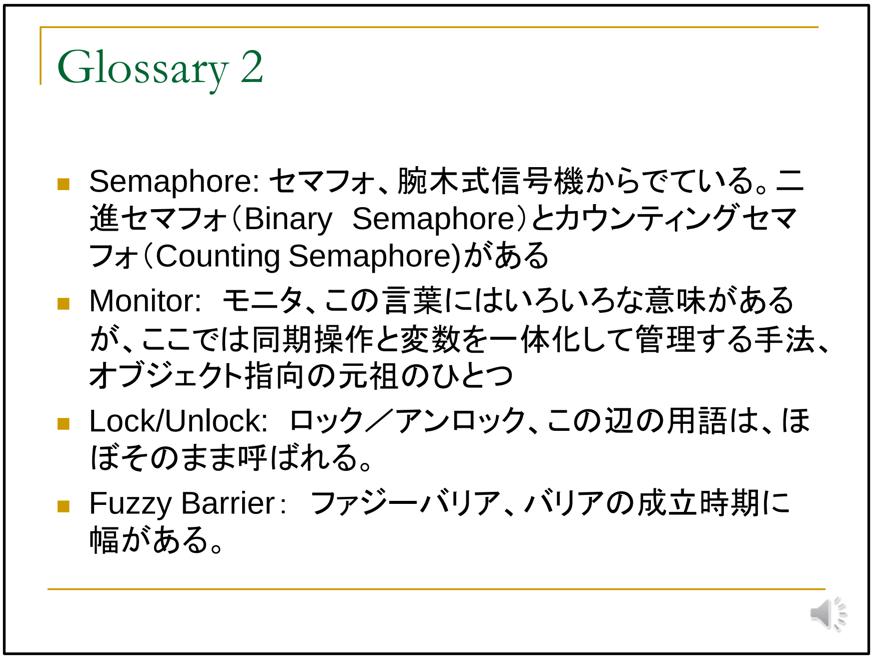
Read Z

PU0 **PU1** **PU2** **PU3**



Summary

- Synchronization is required not only for bus connected multiprocessor but for all MIMD parallel machines.
- Consistency is only kept with synchronization →Consistency Model
- Synchronization with message passing → Message passing model



Glossary 2

- Semaphore: セマフォ、腕木式信号機からでている。二進セマフォ（Binary　Semaphore）とカウンティングセマフォ（Counting Semaphore)がある
- Monitor:　モニタ、この言葉にはいろいろな意味があるが、ここでは同期操作と変数を一体化して管理する手法、オブジェクト指向の元祖のひとつ
- Lock/Unlock:　ロック／アンロック、この辺の用語は、ほぼそのまま呼ばれる。
- Fuzzy Barrier：　ファジーバリア、バリアの成立時期に幅がある。

Exercise

- Write a program for sending a data from PU A to PU B,C,D only using Test & Set operations.