Process synchronization

Tran, Van Hoai

Faculty of Computer Science & Engineering
HCMC University of Technology

E-mail: hoai@hcmut.edu.vn
(partly based on slides of Le Thanh Van)

Outline

1. Basic concepts
2. Critical-section (CS) problem
3. Synchronization hardware
4. Software tools for synchronization
   - Mutex lock
   - Semaphores
5. Synchronization problems

Outline

1 Basic concepts

2 Critical-section (CS) problem

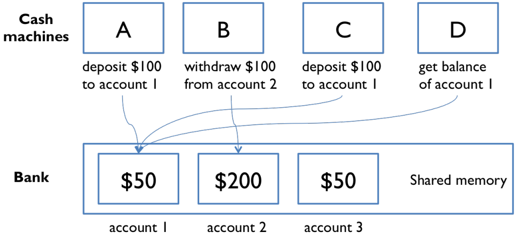
3 Synchronization hardware

4 Software tools for synchronization
- Mutex lock
- Semaphores

5 Synchronization problems

Why do we study synchronization ? (1)

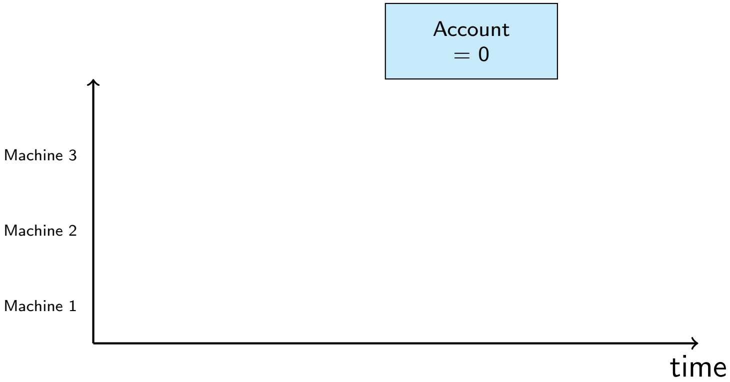
- Different CPU schedulers produce different timings for scheduled processes
- The correctness of a concurrent program should not depend on accidents of timing



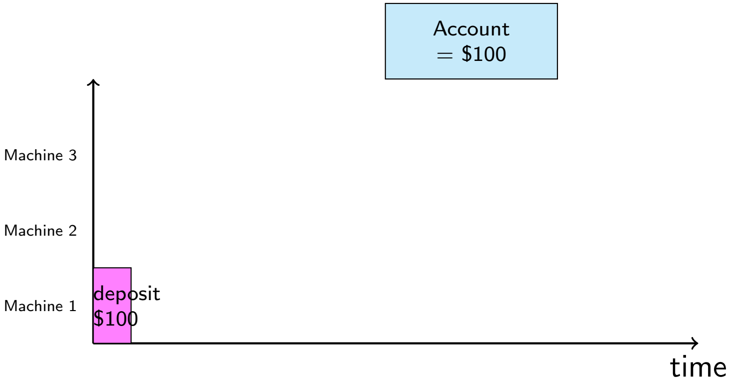
(Source: mit.edu)

Cooperating (concurrent) processes which possibly share a logical address can create the inconsistency in shared data

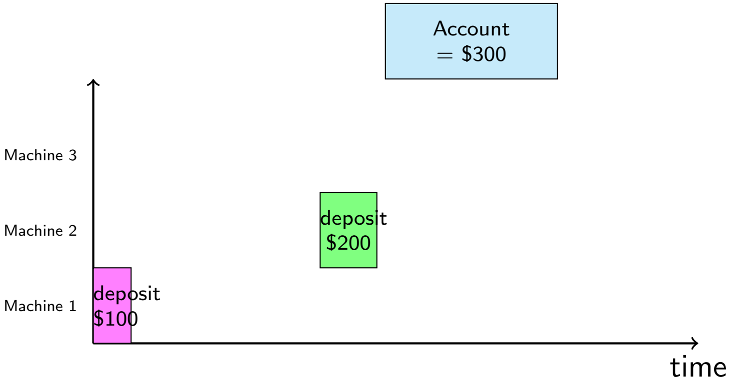
Why do we study synchronization ? (2)



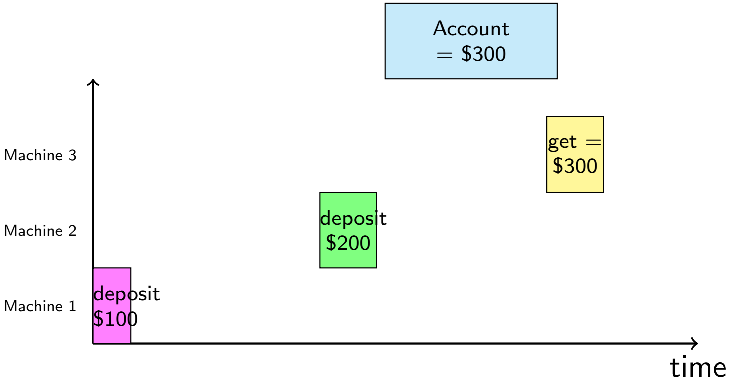
Why do we study synchronization ? (2)



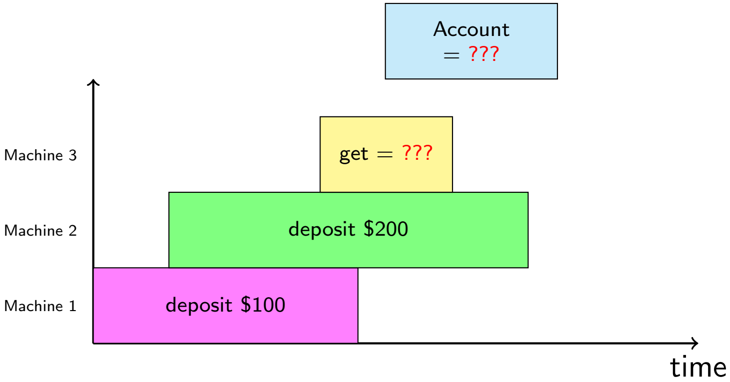
Why do we study synchronization ? (2)



Why do we study synchronization ? (2)



Why do we study synchronization ? (3)



Inconsistency in shared-memory communication (1)

Use a counter to remedy (chữa bệnh) for buffer size of `BUFFER_SIZE - 1`

Producer

```
item next_produced;
while (true){
  while ( counter == BUFFER_SIZE )
    ; /* do nothing */
  buffer[ in ] = next_produced;
  in = ( in + 1 ) % BUFFER_SIZE;
  counter ++;
}
```

Consumer

```
item next_consumed;
while (true){
  while ( counter == 0 )
    ; /* do nothing */
  next_consumed = buffer[ out ];
  out = ( out + 1 ) % BUFFER_SIZE;
  counter --;
}
```

- The codes seem correct, but not
- Suppose currently `counter = 5`, then 2 statements `counter++` and `counter--` executed concurrently
- After those executed, value of `counter` could be 4, 5, or 6 (inconsistently).
  (the expected is 5)

Inconsistency in shared-memory communication (2)

counter++

$register_1$ = `counter`
$register_1 = register_1 + 1$
`counter` = $register_1$

counter--

$register_2$ = `counter`
$register_2 = register_2$ - 1
`counter` = $register_2$

Concurrent execution of 2 processes is interleaved in a low-level sequential execution in any arbitrary order, perhaps as follows.

T_0:	producer	execute	$register_1$ = `counter`	($register_1 = 5$)
T_1:	producer	execute	$register_1 = register_1 + 1$	($register_1 = 6$)
T_2:	consumer	execute	$register_2$ = `counter`	($register_2 = 5$)
T_3:	consumer	execute	$register_2 = register_2$ - 1	($register_2 = 4$)
T_4:	producer	execute	`counter` = $register_1$	(`counter = 6`)
T_5:	consumer	execute	`counter` = $register_2$	(`counter = 4`)

Inconsistency in shared-memory communication (2)

`counter++`

$register_1 = \texttt{counter}$
$register_1 = register_1 + 1$
$\texttt{counter} = register_1$

`counter--`

$register_2 = \texttt{counter}$
$register_2 = register_2 - 1$
$\texttt{counter} = register_2$

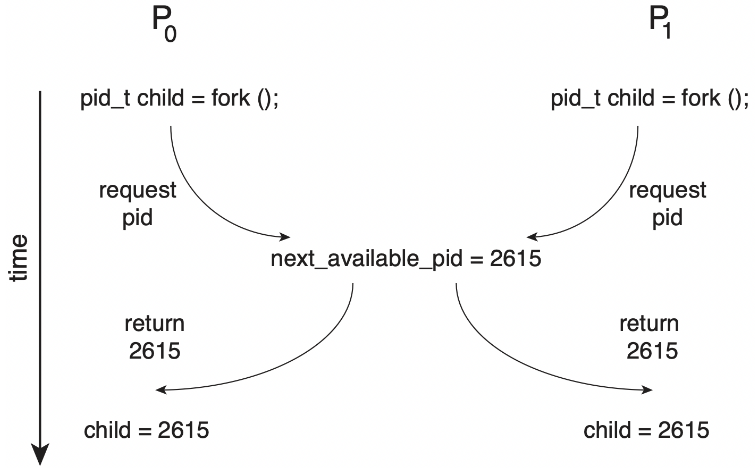
Concurrent execution of 2 processes is interleaved in a low-level sequential execution in any arbitrary order, perhaps as follows.

T_0: producer execute $register_1 = \texttt{counter}$ $(register_1 = 5)$

Race condition

Outcome of executing of multiple cooperating concurrent processes depends on particular order of their access events

Race condition example: `fork()`



Outline

1 Basic concepts

2 Critical-section (CS) problem

3 Synchronization hardware

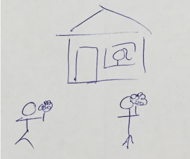
4 Software tools for synchronization
- Mutex lock
- Semaphores

5 Synchronization problems

Critical section

Multiple processes should have the following structure to avoid race condition

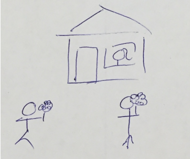
```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```



Critical section

Multiple processes should have the following structure to avoid race condition

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```



Critical section in kernel-mode process

- Kernel-mode processes share some kernel data structures
  list of opened files
- 2 approaches to handle critical sections in OS
  - Nonpreemptive kernel: kernel-mode process cannot be preempted
    → No race conditions
  - Preemptive kernel: kernel-mode process can be preempted
    → more responsive, but prone to critical section

Conditions to critical section

- Mutual exclusion: at most one process allowed in critical section
- Progress: if no process in critical section and some (one or more) wish to enter critical section, the selection of which (of these processes) will enter cannot be postponed indefinitely
- Bounded waiting: a limit on number of times which other processes allowed to enter critical section after a process makes a request and before that request granted.

Algorithm 1 for CS

(assumed for case of 2 processes)

- Processes do not know which is in CS
- $\rightarrow$ use a shared variable "turn" to know which is in CS. (Initially turn=0)

  turn = i $\Rightarrow P_i$ is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Algorithm 1 for CS

(assumed for case of 2 processes)

- Processes do not know which is in CS
- → use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

  `turn = i` ⇒ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here  */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting

Algorithm 1 for CS

(assumed for case of 2 processes)

- Processes do not know which is in CS
- → use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

  `turn = i` $\Rightarrow$ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting

Algorithm 1 for CS

(assumed for case of 2 processes)

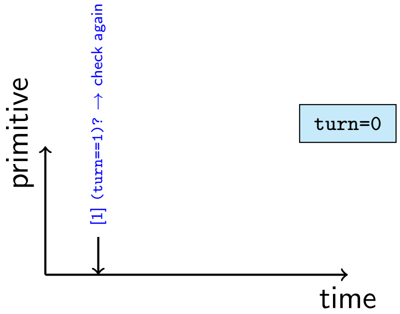
- Processes do not know which is in CS
- → use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

  `turn = i` ⇒ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting



Algorithm 1 for CS

(assumed for case of 2 processes)

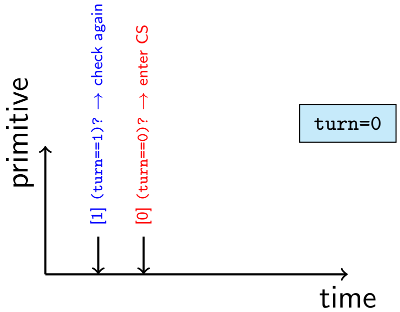
- Processes do not know which is in CS
- $\rightarrow$ use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

  `turn = i` $\Rightarrow P_i$ is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting



Algorithm 1 for CS

(assumed for case of 2 processes)

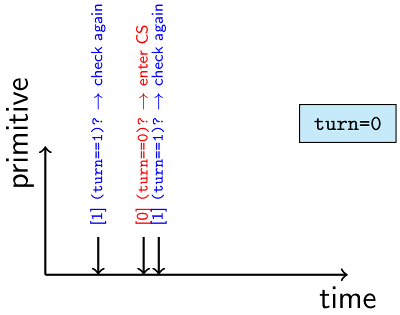
- Processes do not know which is in CS
- → use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

  `turn = i` ⇒ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting



Algorithm 1 for CS

(assumed for case of 2 processes)

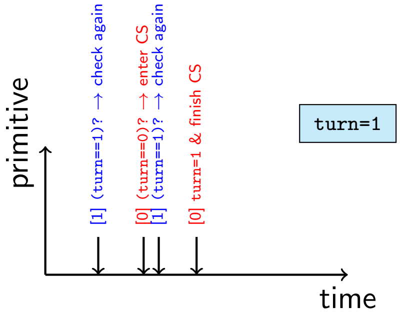
- Processes do not know which is in CS
- → use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

  `turn = i` ⇒ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting



Algorithm 1 for CS

(assumed for case of 2 processes)

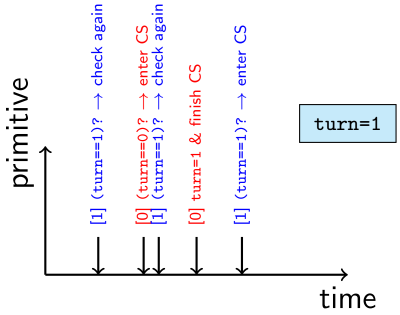
- Processes do not know which is in CS
- $\rightarrow$ use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

  `turn = i` $\Rightarrow P_i$ is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting



Algorithm 1 for CS

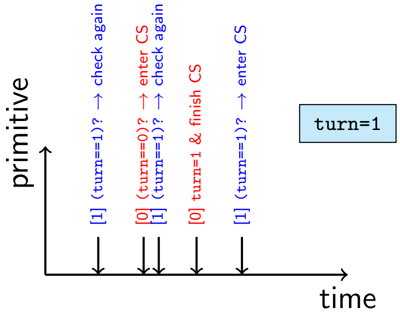
(assumed for case of 2 processes)

- Processes do not know which is in CS
- → use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)
  `turn = i` ⇒ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting , but not progress



Algorithm 1 for CS

(assumed for case of 2 processes)

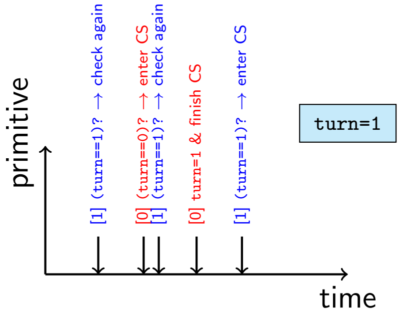
- Processes do not know which is in CS
- $\rightarrow$ use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

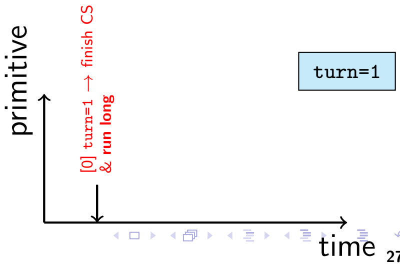
  `turn = i` $\Rightarrow P_i$ is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here  */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting , but not progress





Algorithm 1 for CS

(assumed for case of 2 processes)

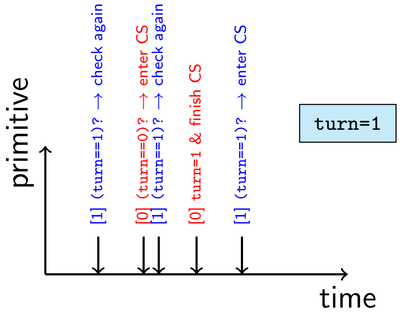
- Processes do not know which is in CS
- → use a shared variable "`turn`" to know which is in CS. (Initially `turn=0`)

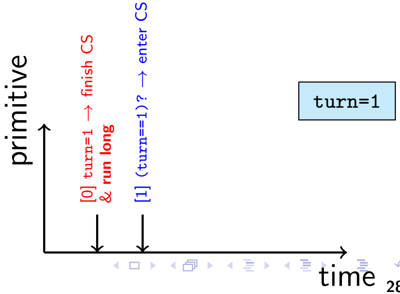
  `turn = i` ⇒ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting , but not progress





Algorithm 1 for CS

(assumed for case of 2 processes)

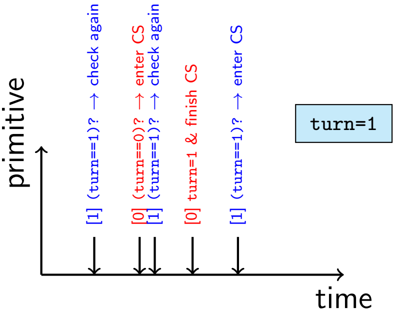
- Processes do not know which is in CS
- → use a shared variable "turn" to know which is in CS. (Initially turn=0)

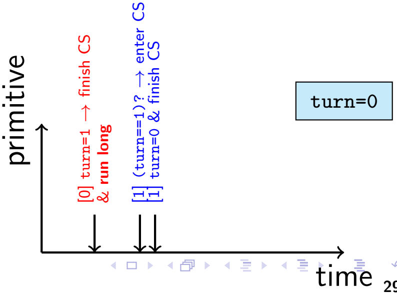
  turn = i $\Rightarrow$ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting , but not progress





Algorithm 1 for CS

(assumed for case of 2 processes)

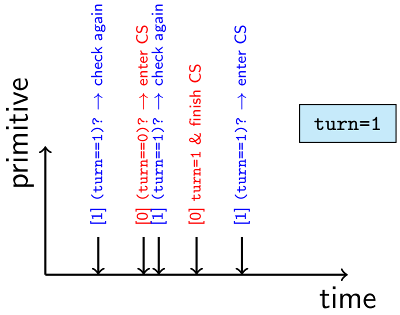
- Processes do not know which is in CS
- → use a shared variable "turn" to know which is in CS. (Initially `turn=0`)

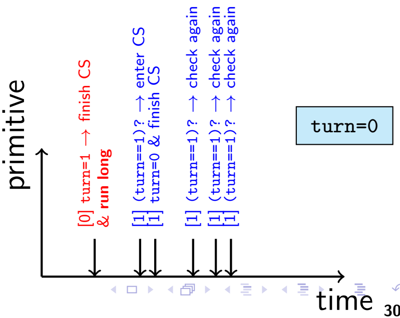
  `turn = i` ⇒ P_i is in its CS

Process P_i

```
do {
  while ( turn != i ) ;
     /* critical section code is here   */
  turn = j;
     /* remainder section code is here */
} while(true);
```

Satisfies mutual exclusion, bounded waiting , but not progress





Algorithm 2 for CS

- `turn` cannot keep the state of readiness into CS
- $\rightarrow$ use a shared variable "`flag`" to keep the state of all processes `flag[i] = true` $\Rightarrow P_i$ is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

Algorithm 2 for CS

- `turn` cannot keep the state of readiness into CS
- $\rightarrow$ use a shared variable "`flag`" to keep the state of all processes `flag[i] = true` $\Rightarrow P_i$ is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress

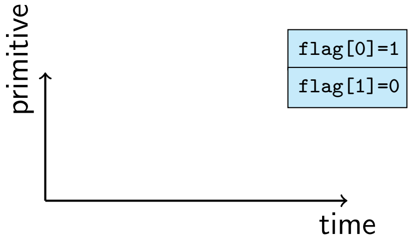
Algorithm 2 for CS

- turn cannot keep the state of readiness into CS
- → use a shared variable "flag" to keep the state of all processes
  flag[i] = true ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress



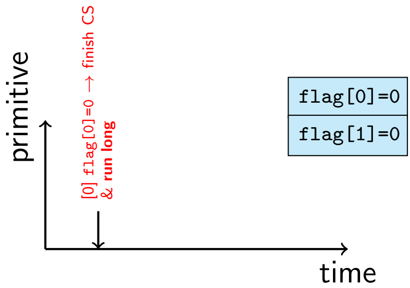
Algorithm 2 for CS

- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress



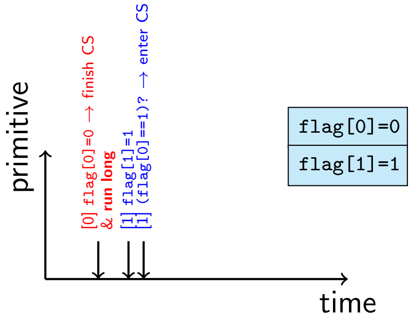
Algorithm 2 for CS

- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress



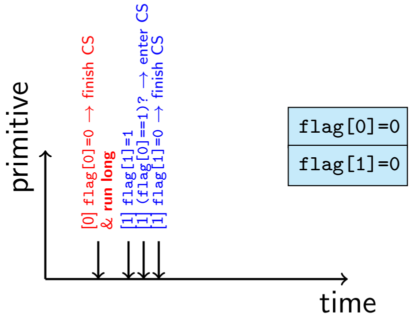
Algorithm 2 for CS

- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress



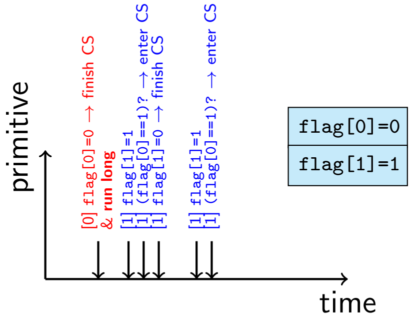
Algorithm 2 for CS

- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress



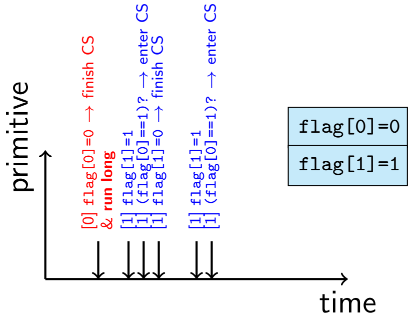
Algorithm 2 for CS

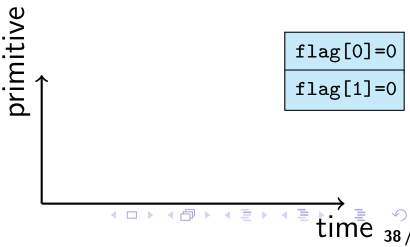
- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress





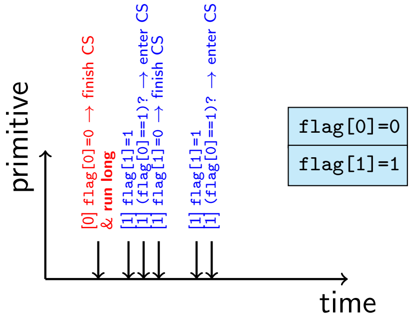
Algorithm 2 for CS

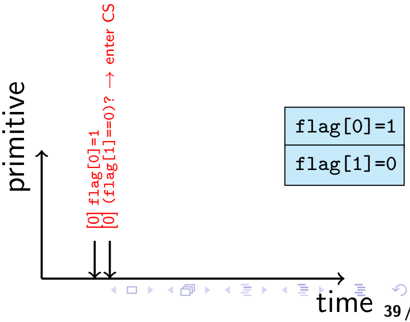
- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress





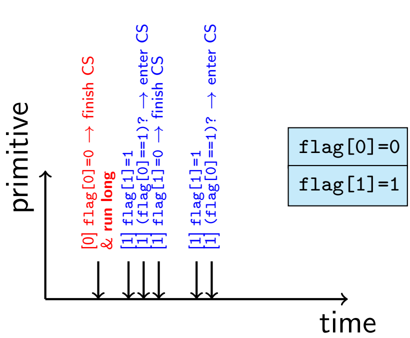
Algorithm 2 for CS

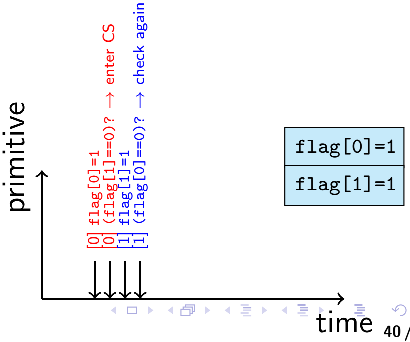
- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress





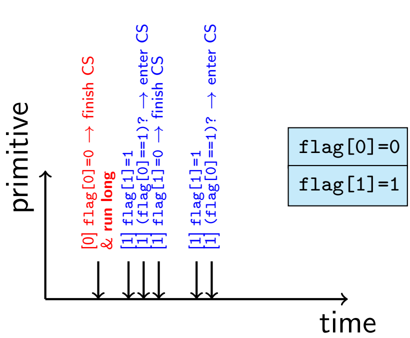
Algorithm 2 for CS

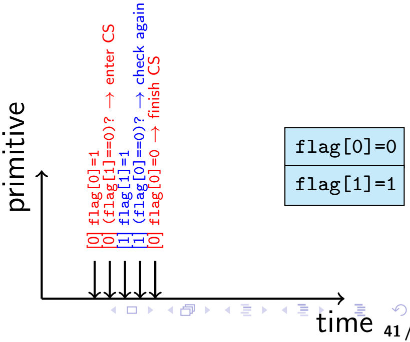
- `turn` cannot keep the state of readiness into CS
- $\rightarrow$ use a shared variable "`flag`" to keep the state of all processes `flag[i] = true` $\Rightarrow P_i$ is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress





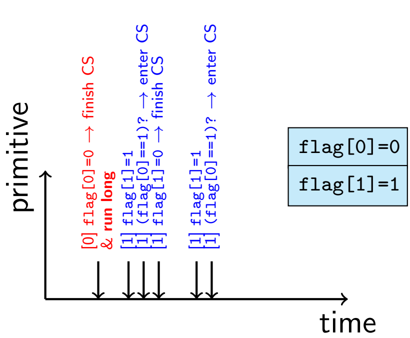
Algorithm 2 for CS

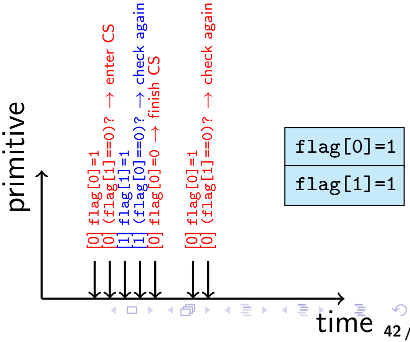
- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes
  `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress





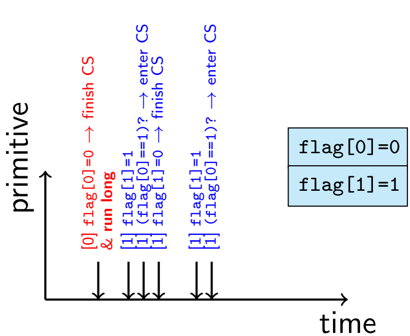
Algorithm 2 for CS

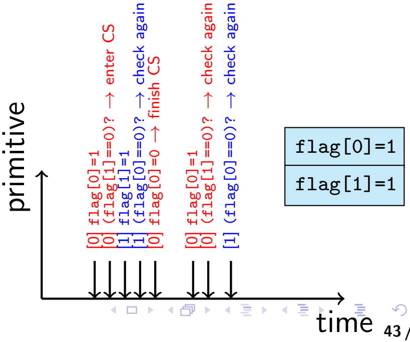
- `turn` cannot keep the state of readiness into CS
- → use a shared variable "`flag`" to keep the state of all processes `flag[i] = true` ⇒ P_i is in CS

Process P_i

```
do {
  flag[i] = true;
  while ( flag[j] ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

mutual exclusion, but not bounded waiting, and not progress





Peterson's algorithm for CS

Satisfy mutual exclusion

- Use both kinds of variables of Algorithm 1 & 2

Process P_i

```
do {
  flag[i] = true;
  turn = j;
  while ( flag[j] && turn == j ) ;
     /* critical section code is here  */
  flag[i] = false;
     /* remainder section code is here */
} while(true);
```

Bakery algorithm (1)

Peterson's algorithm is only for 2 processes. Bakery algorithm is for n processes

- Before entering its critical section, process receives a ticket number. Holder of the smallest number enters its CS
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5,...

Bakery algorithm (2)

- A process is assigned a pair (pid,ticket #) which is ordered lexicographically
- Shared data
  - `boolean choosing[n]`
  - `number[n]`: ticket #

Process P_i

```
do {
  choosing[i] = true;
  number[i] = max(number[0],number[1],...,number[n-1]) + 1;
  choosing[i] = false;
  for( j=0; j<n; j++ ){
    while( choosing[j] ) ;
    while( number[j] != 0 && (number[j] < number[i] ) ;
  }
     /* critical section code is here  */
  number[i] = 0;
     /* remainder section code is here */
} while(true);
```

Outline

1 Basic concepts

2 Critical-section (CS) problem

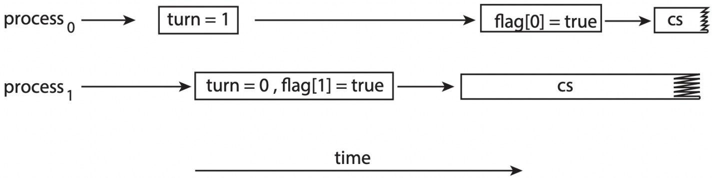
3 Synchronization hardware

4 Software tools for synchronization
- Mutex lock
- Semaphores

5 Synchronization problems

Modern computer architecture

- Multiple cores for multithreaded applications
- Processor or compiler can reorder the execution (if checked no data dependency)



Software-based solutions such as Peterson's are not guaranteed to work on modern computer architectures (mutiple processors/cores)

Hardware synchronization tools

There are 3 forms of hardware instructions to support solving critical-section problem.

- Memory barriers
- Hardware instructions
- Atomic variables

Memory barriers

Memory model

A model that a computer architecture guarantees to application programs on memory.

- **Strongly ordered**: where a memory modification on one processor is immediately visible to all other processors.
- **Weakly ordered**: where modifications to memory on one processor may not be immediately visible to other processors.

Memory barriers

Memory model

A model that a computer architecture guarantees to application programs on memory.

- **Strongly ordered**: where a memory modification on one processor is immediately visible to all other processors.
- **Weakly ordered**: where modifications to memory on one processor may not be immediately visible to other processors.

Memory models vary by processor type. $\Rightarrow$ OS kernel cannot depends on any assumptions regarding the visibility of modifications to memory on a shared-memory multiprocessor.

$\Rightarrow$ Computer architectures provide memory barriers to force any changes in memory to be propagated to all other processors.

Memory barriers - example

Thread 1

```
while (!flag)
  memory_barrier();
print x;
```

Thread 1

```
x = 100;
memory_barrier();
flag = true;
```

`flag` > `x`

`x` > `flag`

The memory barrier will ensure thread 1 will output 100.

We could place a memory barrier into Peterson's algorithm between `flag[i]` and `turn = j`.

Use an uninterruptable (atomic) sequence of instructions which modifies a shared variable `lock` (initially `lock=false`).

test_and_set()

```
boolean test_and_set( boolean *target )
{
  boolean rv = *target;
  *target = true;
  return rv;
}
```

Mutual-exclusion by test_and_set()

```
do {
  while( test_and_set( &lock ) ) ;
     /* critical section code is here  */
  lock = false;
     /* remainder section code is here */
} while (true};
```

Swaping the contents of two words (compare_and_swap()) on a shared variable (lock) (initially lock=0)

compare_and_swap()

```
boolean compare_and_swap( int *value,
                          int expected,
                          int new_value )
{
  int temp = *value;
  if ( *value == expected )
    *value = new_value;
  return temp;
}
```

Mutual-exclusion by compare_and_swap()

```
do {
  while( compare_and_swap( &lock, 0, 1 ) != 0 ) ;
     /* critical section code is here  */
  lock = 0;
     /* remainder section code is here */
} while (true};
```

Swaping the contents of two words (compare_and_swap()) on a shared variable (lock) (initially lock=0)

compare_and_swap()

```
boolean compare_and_swap( int *value,
                          int expected,
                          int new_value )
{
  int temp = *value;
  if ( *value == expected )
    *value = new_value;
  return temp;
}
```

Mutual-exclusion by compare_and_swap()

```
do {
  while( compare_and_swap( &lock, 0, 1 ) != 0 ) ;
     /* critical section code is here  */
  lock = 0;
     /* remainder section code is here */
} while (true};
```

The solution compare_and_swap() above is not bounded waiting.

Process P_i

```
while (true) {
  waiting[i] = true;
  key = 1;
  while (waiting[i] && key == 1)
    key = compare and swap(&lock,0,1);
  waiting[i] = false;

  /* critical section */

  j = (i + 1) % n;
  while ((j != i) && !waiting[j])
    j = (j + 1) % n;

  if (j == i)
    lock = 0;
  else
     waiting[j] = false;

  /* remainder section */
}
```

- `waiting` initialized to `false`, `lock` initialized to 0.
- On leaving CS, P_i scans in the cyclic ordering $(i+1, i+2, \ldots, n-1, 0, \ldots, i-1)$ (that `waiting[j] == true`), and then set `waiting[j] == true` to allow P_j enter CS.
  $\Rightarrow$ Bounded waiting.

Atomic variables

Increment and decrement an integer value may produce race condition.

- `compare_and_swap()` is used as **basic** building blocks for constructing synchronization tools.
- **Atomic variable**: providing atomic operations (`increment()`, `decrement()`) on basic data types such as integers and booleans.

```
void increment(atomic int *v)
{
  int temp;
  do {
    temp = *v;
  } while (temp != compare and swap(v, temp, temp+1));
}
```

Outline

1. Basic concepts
2. Critical-section (CS) problem
3. Synchronization hardware
4. Software tools for synchronization
   - Mutex lock
   - Semaphores
5. Synchronization problems

Mutex locks

Application programmers do not like using hardware-synchronization mechanisms.

OS designers must provide tools to deal with critical-section with 2 functions

- `acquire()`

```
acquire{} {
  while (!available) ; /* busy wait */
  available = false;
}
```

- `release()`

```
release() {
  available = true;
}
```

Before entering CS, programmers call `acquire()`, and call `release()` when leaving CS. Both are atomic which can be implemented by hardware synchronization instructions.

Mutex locks

Application programmers do not like using hardware-synchronization mechanisms.

OS designers must provide tools to deal with critical-section with 2 functions

- `acquire()`

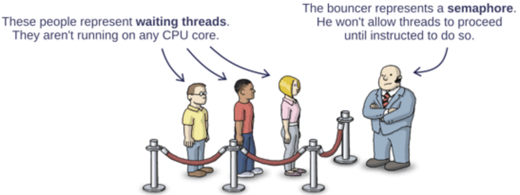
```
acquire{} {
  while (!available) ; /* busy wait */
  available = false;
}
```

- `release()`

Busy waiting = when a process is in CS, other processes loop continuously in order to enter CS (spinlock)

Before entering CS, programmers call `acquire()`, and call `release()` when leaving CS. Both are atomic which can be implemented by hardware synchronization instructions.

Semaphores

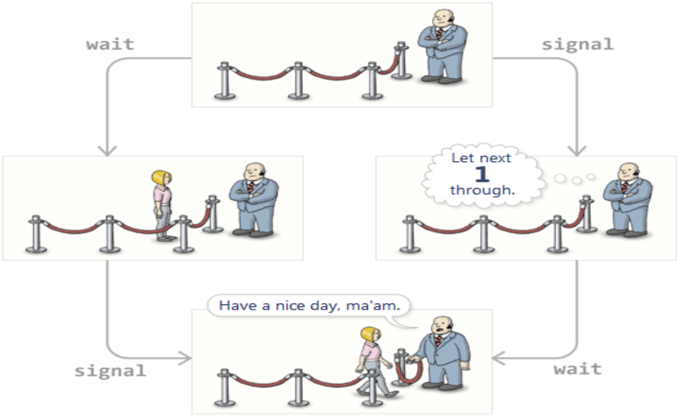


(source:preshing.com)

- Semaphore S: an integer variable
- can only be accessed by indivisible (atomic) operations

```
wait(S) {
  while ( S <= 0 ) ; /* busy wait */
  S --;
}
```

```
signal(S) {
  S ++;
}
```



(source:preshing.com)

Same outcome for different orders of calling `wait()` and `signal()`.

Semaphore usage

- Binary semaphore ($\{0,1\}$): similar to mutex locks
- Counting semaphore: to control access to resource with a finite number of instances
  - `S` initialized to number of resource instances
  - Wish to use an instance, call `wait(S)`
  - Release the resource, call `signal(S)`
  - All instances are in use, processes blocked until `S` > 0

Example of using semaphore `synch` to synchronize 2 processes (similar to `memory_barrier()`).

```
synch = 0;
```

P_1

```
S1;
signal( synch );
```

P_2

```
wait( synch );
S2;
```

Semaphores without busy waiting

Semaphore

Semaphore mentioned before could have busy waiting

Idea to avoid busy waiting

- After the checking of the semaphore fails, the process will be put into waiting queue
- The process is restarted by `wakeup()` when other processes send `signal()` (i.e., put the process into ready queue)

Semaphores without busy waiting

Implementation

Semaphore data structure

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

wait()

```
wait( semaphore *S }{
  S->value --;
  while ( S->value < 0 ){
    add this process to S->list;
    sleep();
  }
}
```

signal()

```
signal( semaphore *S ){
  S->value ++;
  if ( S->value >= 0 ) {
    remove a process P from S->list;
    wakeup( P );
  }
}
```

- `block()`, `wakeup()`: system calls

Semaphore data structure

```
typedef struct {
  int value;
  struct process *list;
} semaphore;
```

wait()

```
wait( semaphore *S ){
  S->value --;
```

```
}
```

signal()

```
signal( semaphore *S ){
  S->value ++;
```

Remarks

- Busy waiting just moved into critical section, not completely eliminated
- `wait()`, `signal()` must be executed atomically (on SMP, other techniques could be used additionally)

What is deadlock ?

P_0

```
wait(S);
wait(Q);
 ...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
 ...
signal(Q);
signal(S);
```

- Two semaphores `S` and `Q` initialized to 1
- After `wait(S)` of P_0 and `wait(Q)` of P_1 executed, both processes wait for corresponding `signal()`s.

What is deadlock ?

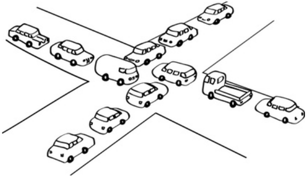
P_0

```
wait(S);
wait(Q);
 ...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
 ...
signal(Q);
signal(S);
```

- Two semaphores `S` and `Q` initialized to 1
- After `wait(S)` of P_0 and `wait(Q)` of P_1 executed, both processes wait for corresponding `signal()`s.



Deadlock

Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

What is deadlock ?

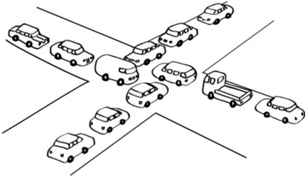
P_0

```
wait(S);
wait(Q);
 ...
signal(S);
signal(Q);
```

P_1

```
wait(Q);
wait(S);
 ...
signal(Q);
signal(S);
```

- Two semaphores S and Q initialized to 1
- After wait(S) of P_0 and wait(Q) of P_1 executed, both processes wait for corresponding signal()s.



Deadlock

Two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes.

Starvation (indefinite blocking)

A process may never be removed from the semaphore queue in which it is suspended

Outline

1 Basic concepts

2 Critical-section (CS) problem

3 Synchronization hardware

4 Software tools for synchronization
- Mutex lock
- Semaphores

5 Synchronization problems

Classic synchronization problems

- Only classic problems considered
  - Bounded-buffer problem
  - Readers-writers problem
  - Dining-philosophers problem
- Use semaphore for synchronization

Bounded-buffer problem

Semaphore data structure

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0;
```

producer

```
do {
  /* produce an item in      */
  /* next produced           */
  wait( empty );
  wait( mutex );
  /* add next_produced to    */
  /* the buffer              */
  signal( mutex );
  signal( full );
} while (true);
```

consumer

```
do {
  wait( full );
  wait( mutex );
  /* remove an item from the */
  /* buffer to next consumed */
  signal( mutex );
  signal( empty );
  /* consume the item in     */
  /* next_consumed           */
} while (true);
```

Readers-writers problem

- A database shared among several processes: some are readers, some are writers
- If a writer and some other process access database simultaneously chaos may occur

First readers-writers problem

no reader kept waiting unless a writer has already obtained permission to use shared objects

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

Writer

```
do {
  wait( rw_mutex );
  .../* writing performed */...
  signal( rw_mutex );
} while (true);
```

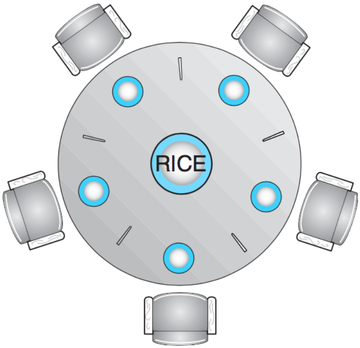
Reader

```
do {
  wait( mutex );
  read_count ++;
  if ( read_count == 1 )
    wait( rw_mutex );
  signal( mutex );
  .../* reading performed */...
  wait( mutex );
  read_count --;
  if ( read_count == 0 )
    signal( rw_mutex );
  signal( mutex );
} while (true);
```

Dining-philosopher

Dining-philosopher

- Resource: 5 single chopsticks
- 5 Philosophers sitting around the table, with actions
    - Think: do nothing, not critical
    - Eat: need 2 chopsticks (left and right)



Semaphores can be used to solve dininig-philosophers synchronization problem, using 5 semaphores

```
semaphore chopstick[5];
```

Philosopher *i*

```
do {
  wait( chopstock[i] );
  wait( chopstick[(i+1) % 5] );
  /* eat eat eat */
  signal( chopstick[i] );
  signal( chopstick[(i+1) % 5] );
  /* think think think */
} while (true);
```