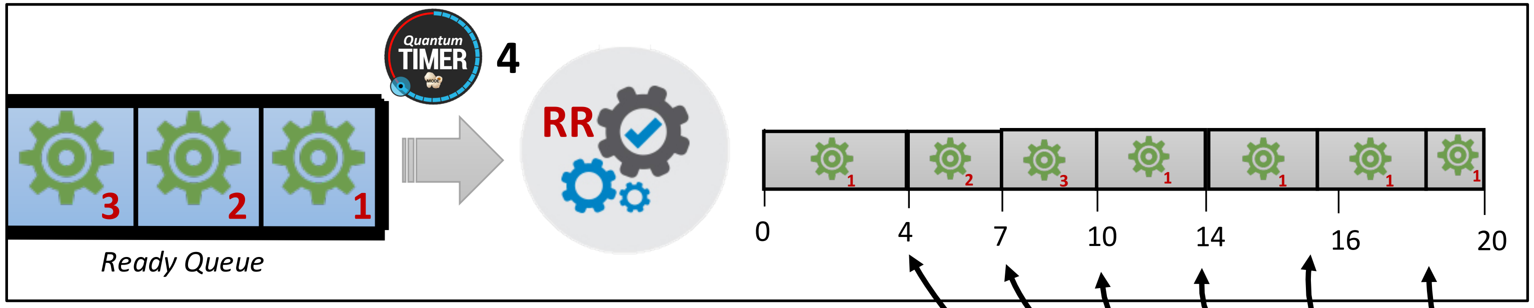
CPE 460 Operating System Design

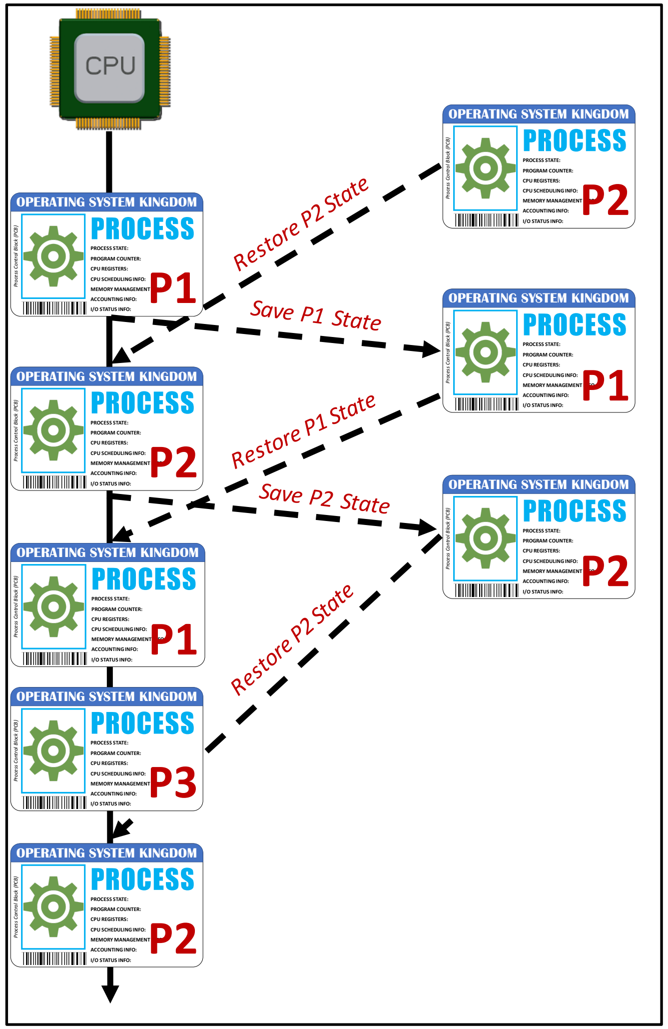
Chapter 5: Process Synchronization

Ahmed Tamrawi

Copyright notice: *care has been taken to use only those web images deemed by the instructor to be in the public domain. If you see a copyrighted image on any slide and are the copyright owner, please contact the instructor. It will be removed.*

CPU Scheduling

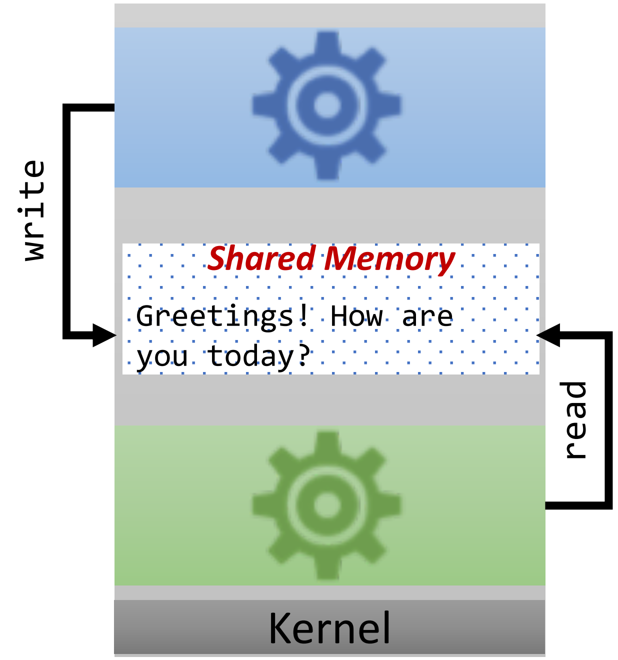




Processes can execute concurrently

May be interrupted at any time, partially completing execution

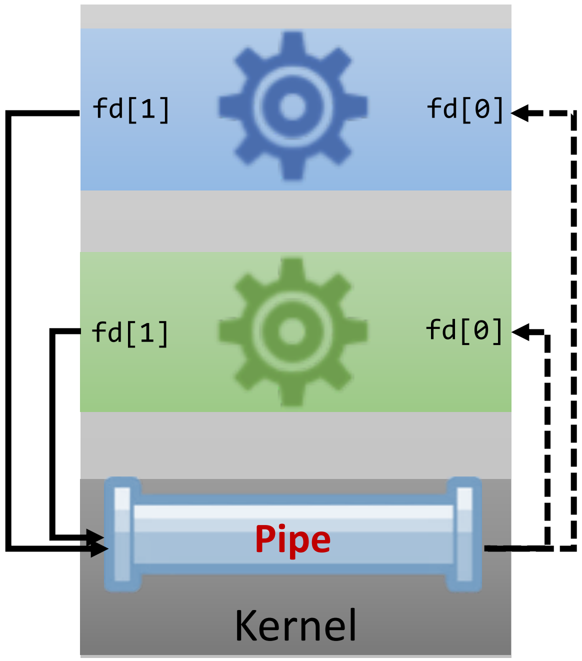
Context Switching



Shared Memory

Synchronization Problems

Concurrent access to shared data may result in data inconsistency



Pipe

Maintaining data consistency requires mechanisms to ensure the **orderly execution of cooperating processes**

How to get free money?


Buma


```
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```


```
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```

withdraw(1000JD)

```
boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}
```

balance = 10000JD

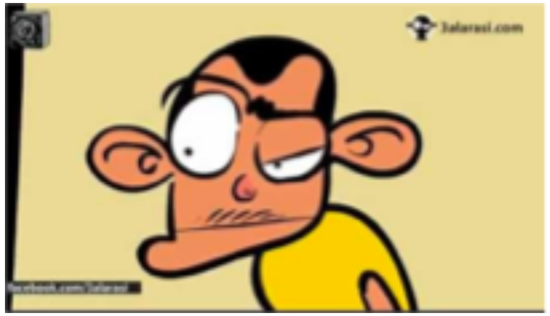
balance = 9000JD

withdraw(1000JD)

balance = 10000JD

balance = 9000JD

```
boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}
```



withdraw(1000JD)

amount = 1000JD
balance = 10000JD

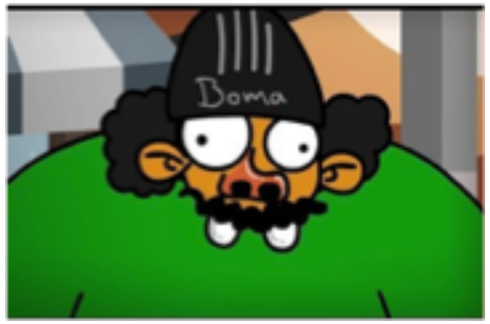
register1 = balance

register1 = register1 - amount

balance = register1

balance = 9000JOD

```java
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```



withdraw(1000JD)

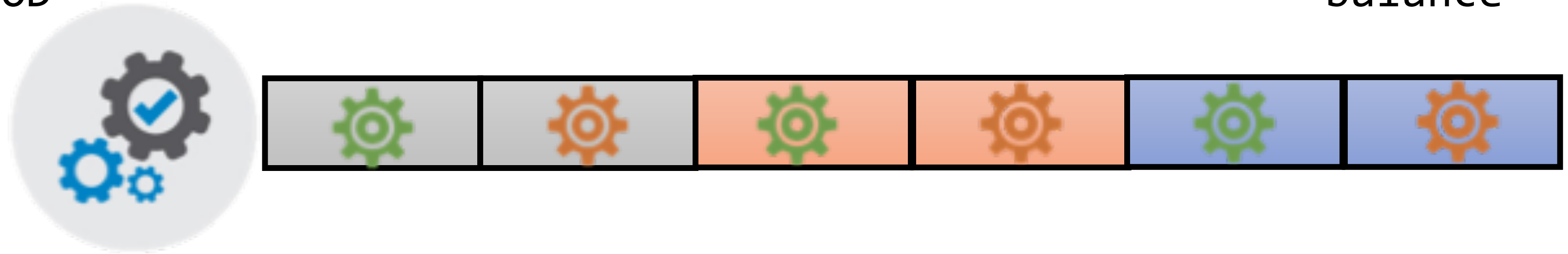
amount = 1000JD
balance = 10000JD

register2 = balance

register2 = register2 - amount

balance = register2

balance = 9000JOD



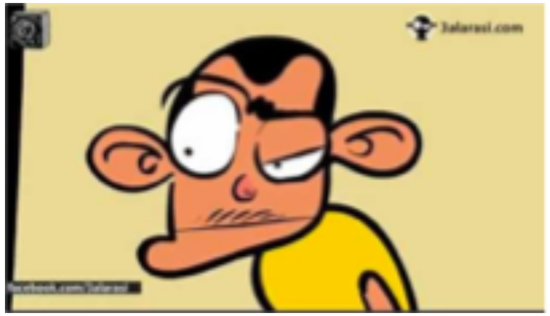

```
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```

Why did this trick work?

We allowed both processes to manipulate the balance counter concurrently.

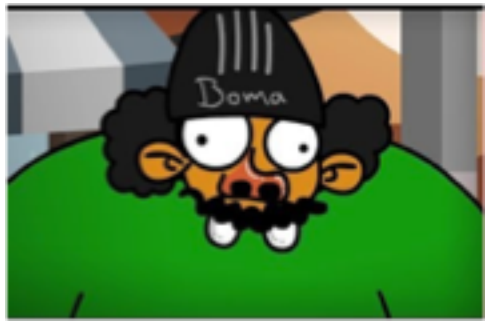
Race Condition

Several processes access and manipulate the **same** data **concurrently** and the outcome of the execution **depends** on the particular order in which the access takes place



withdraw(1000JD)

To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the balance



withdraw(1000JD)

CRITICAL section

```
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```

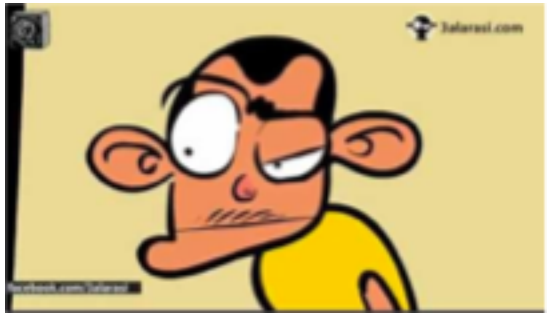
CRITICAL section

When one process in critical section, **no other** may be in its critical section

Each process must **ask permission** to enter critical section

When one process in critical section, **no other** may be in its critical section

Each process must **ask permission** to enter critical section



withdraw(1000JD)

amount = 1000JD
balance = 10000JD

register1 = balance

register1 = register1 - amount

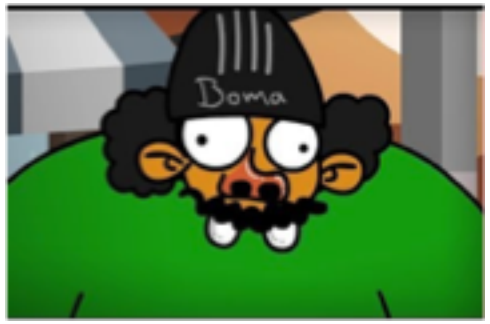
balance = register1

balance = 9000JD

CRITICAL section

```
double balance = 10000;

boolean withdraw(int amount){
    if(amount < 0){
        return false;
    }
    if (balance < amount) {
        return false
    } else {
        balance = balance - amount;
        return true
    }
}

boolean deposit(int amount){
    if(amount < 0){
        return false;
    }
    balance = balance + amount;
    return true;
}
```

CRITICAL section



withdraw(1000JD)

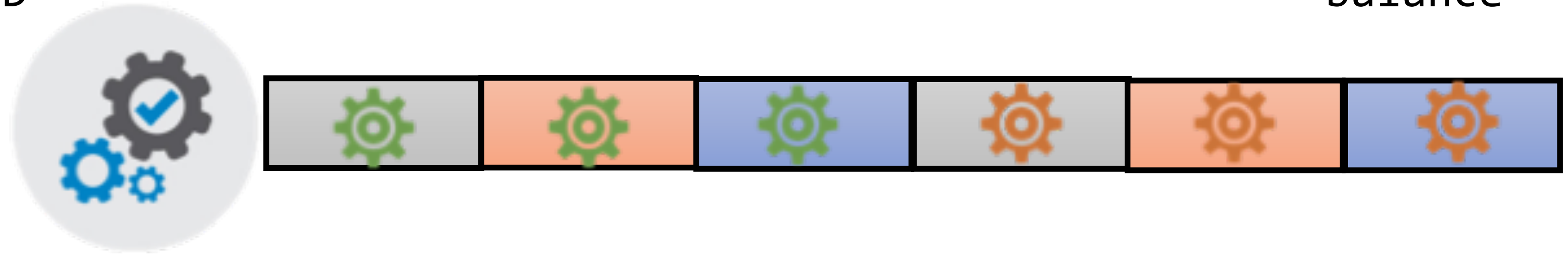
amount = 1000JD
balance = 9000JD

register2 = balance

register2 = register2- amount

balance = register2

balance = 8000JD



Concurrent accesses to **shared resources/variables** must be protected in such a way that ***it cannot be executed by more than one process***.

CRITICAL section

A code segment that accesses shared variables or resources and has to be executed as an atomic action that does not allow multiple concurrent accesses

CRITICAL section PROBLEM

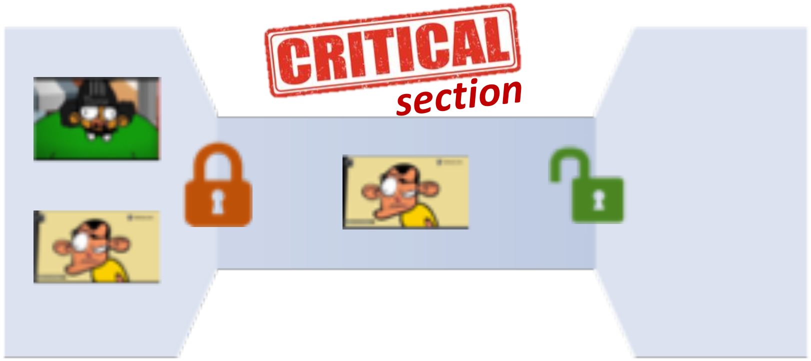
The problem of how to ensure that at most one process is executing its critical section at a given time.

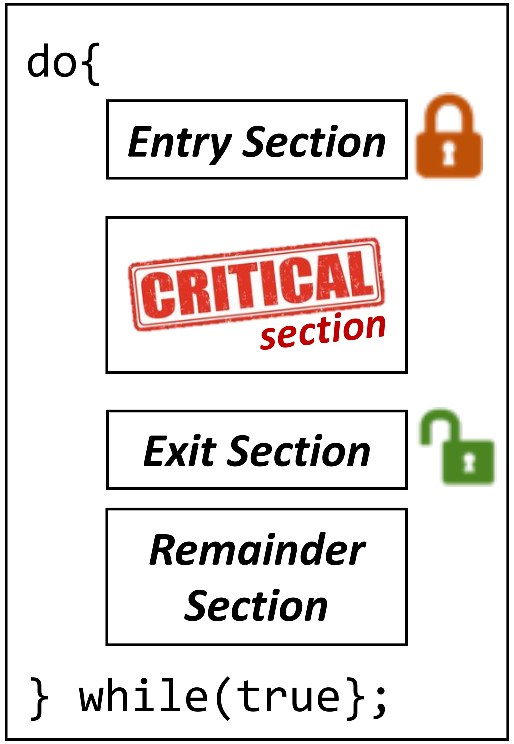
```
do{
    Entry Section

    CRITICAL section

    Exit Section

    Remainder Section
} while(true};
```

*It controls the entry into critical section and gets a **LOCK** on required resources. Each process must ask permission to enter critical section*

A code segment that accesses shared variables or resources and has to be executed as an atomic action that does not allow multiple concurrent accesses

*Removes the **LOCK** from the resources and let the other processes know that its critical section is over*





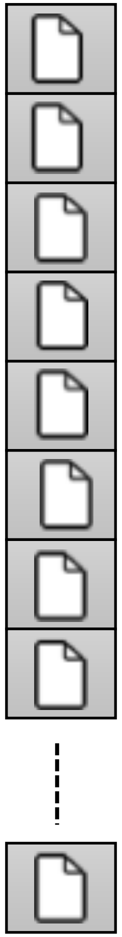
Any solution to the critical-section problem must satisfy:

1. **Mutual Exclusion** - If a process is executing in its critical section, then *no other processes can be executing in their critical sections.*

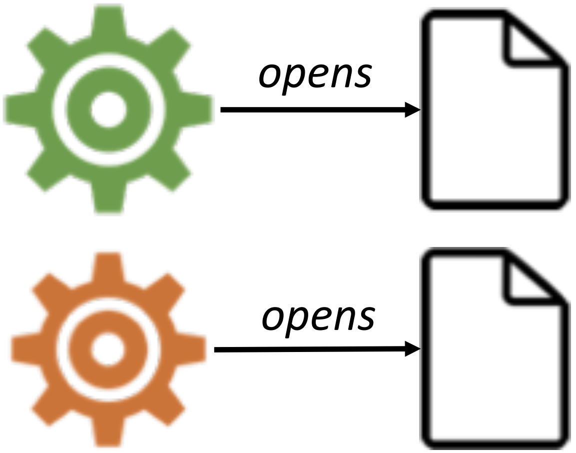
2. **Progress** - If no process is executing in its critical section, and if there are some processes that wish to enter their critical sections, then one of these processes will get into the critical section.

3. **Bounded Waiting** - After a process makes a request to enter its critical section, there is a bound on the number of times that other processes are allowed to enter their critical sections, before the request is granted.

Critical Section Handling in OS



List of Open Files



Concurrent modification to the list may result in **race condition**

It is up to kernel developers to ensure that the OS is free from such race conditions.

Two general approaches are used to handle critical sections in operating systems:

Preemptive
allows preemption of process when running in kernel mode

Non-preemptive
runs until exits kernel mode, blocks, or voluntarily yields CPU

Non-preemptive is *essentially free of race conditions in kernel mode*

Why, then would anyone favor a preemptive kernel over a nonpreemptive one?


شو
الحل يا
خال

CRITICAL *section* PROBLEM SOLUTIONS

- Peterson’s Algorithm
- Synchronization Hardware
- Mutex Locks
- Semaphores

CRITICAL section PROBLEM SOLUTIONS

Peterson's Algorithm

https://en.wikipedia.org/wiki/Peterson's_algorithm

ECONOMICAL SOLUTIONS FOR THE
CRITICAL SECTION PROBLEM IN A DISTRIBUTED SYSTEM*

extended abstract

Gary L. Peterson and Michael J. Fischer

Department of Computer Science
University of Washington
Seattle, Washington 98195

1. Introduction

A solution to the critical section problem, first posed by Dijkstra [1], is a fundamental requirement for concurrent program control. The problem is to ensure that no two processes are in a specified area of their programs (the critical section) at the same time. Improvements to Dijkstra's solution were made by Knuth [2], deBruijn [3], and Eisenberg and McGuire [4]. The situation for a distributed system was considered by Lamport [5]. Rivest and Pratt [6] presented a solution for a distributed system where processes may repeatedly fail. The algorithms to be presented will be further improvements, where the comparisons will be made according to three measures: message size -- the number of values the variable for interprocess communication can take on; fairness -- the sequence in which waiting processes enter their critical sections; and time -- the amount of time a process spends attempting to enter its critical section.

A read occurring simultaneously with a write returns either the old or new value. A process' state value is automatically set to a prespecified value on process failure. When a process later restarts, it begins at a specified control point and its state remains dead.

The processors run totally asynchronously, and we make no assumptions about the relative speeds of any processors at any time. Thus it is possible for one processor to execute thousands of steps while another executes just a few, and then the speeds may suddenly reverse. We assume only that each active process is always executing instructions, although possibly very slowly.

A formalization of this model would be essentially an n-tuple of random access machines, augmented with the visible states and instructions for manipulating them. Our notion of a computation, however, must be considerably more complicated, for it is necessary to consider

CRITICAL section PROBLEM SOLUTIONS

Peterson's Algorithm

Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two.

Information common to both processes:

```
boolean flag[2] = {false, false};
int turn;
```

A `flag[n]` value of `true` indicates that the process n wants to enter the **critical section**

The variable `turn` indicates whose turn it is to enter the critical section

```
do{
    flag[i] = true;
    int j = 1 - i;
    turn = j;
    while(flag[j] && turn == j){
        // busy wait
    }

    CRITICAL section

    flag[i] = false;

    Remainder Section

} while(true};
```

flag[0]	false
flag[1]	false
turn	0

i = 0

Ready to Running

i = 1

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	false
turn	0

i = 0

Ready to Running

i = 1

do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);

do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);

flag[0]	true
flag[1]	false
turn	1

i = 0

Ready to Running

i = 1

do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);

do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);

flag[0]	true
flag[1]	false
turn	1

i = 0

i = 1

Ready to Running

Switch Context to P1

Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j

CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j

CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	1

i = 0

Ready to Running

Switch Context to P1

Running to Ready

i = 1

Ready to Running

P0 (i = 0):

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

P1 (i = 1):

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1

Running to Ready

i = 1

Ready to Running

do{
flag[i] = true
turn = j
flag[j] && turn == j


flag[i] = false;
// Remainder Section
}while(true);

do{
flag[i] = true
turn = j
flag[j] && turn == j


flag[i] = false;
// Remainder Section
}while(true);

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1

Running to Ready

i = 1

Ready to Running

P0 (i = 0):

do{
flag[i] = true
turn = j
flag[j] && turn == j


flag[i] = false;
// Remainder Section
}while(true);

P1 (i = 1):

do{
flag[i] = true
turn = j
flag[j] && turn == j


flag[i] = false;
// Remainder Section
}while(true);

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1

Running to Ready

i = 1

Ready to Running

P0:

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

P1:

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

P0 (i = 0):

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

P1 (i = 1):

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

i = 1

Ready to Running

Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

i = 1

Ready to Running

Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```
CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```
CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```

CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```

CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	false
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	false
flag[1]	true
turn	0

i = 0

i = 1

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Switch Context to P1
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	false
flag[1]	true
turn	0

i = 0

i = 1

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Switch Context to P1
Running to Ready

Ready to Running

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	false
flag[1]	true
turn	0

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```

CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```

CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	false
flag[1]	true
turn	0

P0 (i = 0)

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Switch Context to P1
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

P1 (i = 1)

Ready to Running
Switch Context to P0
Running to Ready

Ready to Running
Switch Context to P0
Running to Ready

Ready to Running
Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
⌛
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Switch Context to P1
Running to Ready

i = 1

Ready to Running
Switch Context to P0
Running to Ready

Ready to Running
Switch Context to P0
Running to Ready

Ready to Running
Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0] true

flag[1] true

turn 1

i = 0

i = 1

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```

CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j
```

CRITICAL section

```
flag[i] = false;
// Remainder Section
}while(true);
```

flag[0]	true
flag[1]	true
turn	1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Switch Context to P1
Running to Ready

Switch Context to P1
Running to Ready

i = 1

Ready to Running
Switch Context to P0
Running to Ready

Ready to Running
Switch Context to P0
Running to Ready

Ready to Running
Switch Context to P0
Running to Ready

```
do{
flag[i] = true
turn = j
flag[j] && turn == j

CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

```
do{
flag[i] = true
turn = j
flag[j] && turn == j

CRITICAL section
flag[i] = false;
// Remainder Section
}while(true);
```

CRITICAL section PROBLEM SOLUTIONS

Peterson's Algorithm

Any solution to the critical-section problem must satisfy:

1. **Mutual Exclusion** - If a process is executing in its critical section, then no other processes can be executing in their critical sections.

```
(flag[j] && turn == j) = false
```

```
flag[0] == true
turn == 0 ||
flag[1] == false
```

```
flag[1] == true
turn == 1 ||
flag[0] == false
```

2. **Progress** - If no process is executing in its critical section, and if there are some processes that wish to enter their critical sections, then one of these processes will get into the critical section.

A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section.

3. **Bounded Waiting** - After a process makes a request to enter its critical section, there is a bound on the number of times that other processes are allowed to enter their critical sections, before the request is granted.

A process will never wait longer than one turn for entrance to the critical section:

```
do{
    flag[i] = true;
    int j = 1 - i;
    turn = j;
    while(flag[j] && turn == j){
        // process busy wait
    }
```

CRITICAL section

```
    flag[i] = false;
```

Remainder Section

```
} while(true};
```

CRITICAL *section* PROBLEM SOLUTIONS

Peterson's Algorithm

Peterson's original formulation worked with only two processes, the algorithm can be generalized for more than two.

Because of the way modern computer architectures perform basic machine-language instructions, such as `load` and `store`, there are no guarantees that Peterson's solution **will work correctly on such architectures**.

Assume that the `load` and `store` machine-language instructions are **atomic**; that is, cannot be interrupted

However, it provides a good algorithmic description of solving the critical-section problem and illustrates some of the complexities involved in designing software that addresses the requirements of mutual exclusion, progress, and bounded waiting.

```
do{
    flag[i] = true;
    int j = 1 - i;
    turn = j;
    while(flag[j] && turn == j){
        // process busy wait
    }

    CRITICAL section

    flag[i] = false;

    Remainder Section

} while(true};
```

CRITICAL *section* PROBLEM SOLUTIONS

Synchronization Hardware

Hardware support for implementing the critical section code

All solutions below based on idea of locking
protecting critical regions via locks

```
do{
    // acquire lock;
        CRITICAL section
    // release lock;
        Remainder Section
} while(true};
```

Uninterruptible Operations

Uniprocessors Architecture
Could simply disable interrupts so that running code would execute without preemption

Multiprocessors Architecture
Generally too inefficient making the OS not broadly scalable

Atomic (Uninterruptible) hardware instructions

`test_and_set`

`compare_and_swap`

CRITICAL section PROBLEM SOLUTIONS

test_and_set

executed atomically

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

test_and_set Instruction

```
*target = true
test_and_set(&target) = true
*target = true
```

```
*target = false
test_and_set(&target) = false
*target = true
```

Information common to processes:

```
boolean lock = false;
```

```
do{
    while(test_and_set(&lock)){
        // busy waiting
    }
    CRITICAL section
    lock = false;
    Remainder Section
} while(true};
```

https://github.com/torvalds/linux/blob/master/arch/sparc/lib/bitops.S

CRITICAL *section* PROBLEM SOLUTIONS compare_and_swap

executed atomically

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

compare_and_swap Instruction

```
value = 0
compare_and_swap(&value, 0, 1) = 0
value = 1
```

```
value = 1
compare_and_swap(&value, 0, 0) = 1
value = 1
```

```
value = 0
compare_and_swap(&value, 1, 1) = 0
value = 0
```

```
value = 1
compare_and_swap(&value, 1, 0) = 1
value = 0
```

Information common to processes:

```
int lock = 0;
```

```
do{
    while(compare_and_swap(&lock, 0, 1)){
        // busy waiting
    }
```

CRITICAL *section*

```
    lock = 0;
```

Remainder Section

```
} while(true};
```

```
do{
    while(test_and_set(&lock)){
        // do nothing
    }
    CRITICAL section
    lock = false;
    Remainder Section
} while(true};
```

```
do{
    while(compare_and_swap(&lock, 0, 1)){
        // do nothing
    }
    CRITICAL section
    lock = 0;
    Remainder Section
} while(true};
```

Any solution to the critical-section problem must satisfy:

1 **Mutual Exclusion** - If a process is executing in its critical section, then no other processes can be executing in their critical sections.


2 **Progress** - If no process is executing in its critical section, and if there are some processes that wish to enter their critical sections, then one of these processes will get into the critical section.


3 **Bounded Waiting** - After a process makes a request to enter its critical section, there is a bound on the number of times that other processes are allowed to enter their critical sections, before the request is granted.


Information common to processes:

```
boolean waiting[n];
boolean lock = false
```

```
do{
    waiting[i] = true;
    while(waiting[i] && test_and_set(&lock)){
        // busy waiting
    }
    waiting[i] = false;
```


```
    j = (i + 1) % n;
    while((j != i) && !waiting[j]){
        j = (j + 1) % n;
    }
    if (j == i){
        lock = false;
    }else{
        waiting[j] = false;
    }
```

Remainder Section

```
} while(true};
```

Nice Explanation: http://stackoverflow.com/questions/31084724/bounded-waiting-mutual-exclusion-with-test-and-set

waiting[0]	false
waiting[1]	false
waiting[1]	false
lock	false

i = 0

i = 1

i = 2

Ready to Running

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

waiting[0]	true
waiting[1]	false
waiting[2]	false
lock	true

i = 0

i = 1

i = 2

Ready to Running

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

waiting[0]	false
waiting[1]	false
waiting[2]	false
lock	true

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1

Running to Ready

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

waiting[0]	false
waiting[1]	false
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

Ready to Running

Switch Context to P1

Running to Ready

i = 1

Ready to Running

i = 2

waiting[0]	false
waiting[1]	true
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

Ready to Running

Switch Context to P1

Running to Ready

i = 1

Ready to Running

i = 2

waiting[0]	false
waiting[1]	true
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P0

Running to Ready

waiting[0]	false
waiting[1]	true
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

waiting[0]	false
waiting[1]	true
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

i = 1

Ready to Running

Switch Context to P0

Running to Ready

i = 2

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] && test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
(j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 2

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

i = 1

Ready to Running

Switch Context to P0
Running to Ready

i = 2

Ready to Running

Switch Context to P0
Running to Ready

waiting[0]	false
waiting[1]	true
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

i = 1

i = 2

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

i = 2

Ready to Running

Switch Context to P0
Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

i = 2

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

i = 2

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

i = 1

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

i = 2

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

Switch Context to P1

Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

i = 2

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 2

j = 1

i = 0

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

i = 1

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

i = 2

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

Switch Context to P1

Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 2

j = 1

i = 0

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

i = 1

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

i = 2

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

Switch Context to P1

Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	true
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⌛
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 2

j = 1

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

i = 2

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)
⧗
waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1 j = 2

i = 0

Ready to Running

Switch Context to P1

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

i = 1

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

Switch Context to P2

Running to Ready

i = 2

Ready to Running

Switch Context to P0

Running to Ready

Ready to Running

Switch Context to P1

Running to Ready

waiting[0]	false
waiting[1]	false
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1 j = 2

i = 0

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 2

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

waiting[0]	false
waiting[1]	false
waiting[2]	false
lock	true

```
do{
waiting[i] = true;
waiting[i] &&
test_and_set(&lock)

waiting[i] = false;
CRITICAL section
j = (i + 1) % n;
while((j != i) && !waiting[j]){
    j = (j + 1) % n;
}
if (j == i)
lock = false;
waiting[j] = false;
}while(true);
```

j = 1 j = 2

i = 0

Ready to Running

Switch Context to P1
Running to Ready

i = 1

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

i = 2

Ready to Running

Switch Context to P0
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Switch Context to P1
Running to Ready

Ready to Running

Switch Context to P2
Running to Ready

Ready to Running

Any solution to the critical-section problem must satisfy:

1 **Mutual Exclusion** - If a process is executing in its critical section, then no other processes can be executing in their critical sections.

2 **Progress** - If no process is executing in its critical section, and if there are some processes that wish to enter their critical sections, then one of these processes will get into the critical section.

3 **Bounded Waiting** - After a process makes a request to enter its critical section, there is a bound on the number of times that other processes are allowed to enter their critical sections, before the request is granted.

```
do{
    waiting[i] = true;
    while(waiting[i] && test_and_set(&lock)){
        // do nothing
    }
    waiting[i] = false;

    CRITICAL section

    j = (i + 1) % n;
    while((j != i) && !waiting[j]){
        j = (j + 1) % n;
    }
    if (j == i){
        lock = false;
    }else{
        waiting[j] = false;
    }

    Remainder Section

} while(true};
```

```
boolean lock = false;
```

```
do{
    while(test_and_set(&lock)){
        busy_wait();
    }
    operation_1();
    operation_2();
    lock = false;
    Remainder Section
} while(true};
```

CRITICAL section

```
*lock= true
test_and_set(&lock) = true
*lock = true
```

```
*lock= false
test_and_set(&lock) = false
*lock= true
```

Time	P_0 $(i = 0)$	$P_1 (i = 1)$
1	Context-switching to P_0 (*Ready to Running*)	
2	**test_and_set**(&lock) = false	
3	operation_1();	
4	Context-switching to P_1 (*Ready to Running*)	
5		**test_and_set**(&lock) = true
6		busy_wait();
7		**test_and_set**(&lock) = true
8		busy_wait();
9	Context-switching to P_0 (*Ready to Running*)	
10	operation_2();	
11	Context-switching to P_1 (*Ready to Running*)	
12		**test_and_set**(&lock) = true
13	Context-switching to P_0 (*Ready to Running*)	
14	lock = false;	
15	Context-switching to P_1 (*Ready to Running*)	
16		busy_wait();
17		**test_and_set**(&lock) = false
18		operation_1();
19	Context-switching to P_0 (*Ready to Running*)	
20	**test_and_set**(&lock) = true	
21	busy_wait();	

CRITICAL *section* PROBLEM SOLUTIONS

Mutex Locks

Previous solutions are complicated and generally **inaccessible** to application programmers

OS designers provide developers with mechanism to build software tools to solve critical section problem

```
do{
    acquire_lock();

        CRITICAL section

    release_lock();

    Remainder Section

} while(true};
```

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

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

(Atomic) Uninterruptible Operations

Usually implemented via hardware atomic instructions

```
do{
    acquire_lock();

    CRITICAL section

    release_lock();

    Remainder Section

} while(true};
```

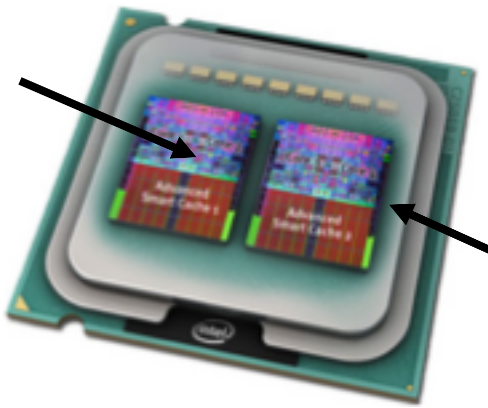
```
acquire_lock() {
    while (!available){
        // busy wait
    }
    available = false;
}
```

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

The main disadvantage of the implementation given here is that it requires **busy waiting**

We call it **spinlock** because the process “spins” while waiting for the lock to become available.

Busy waiting wastes valuable CPU time, let the waiting “spinning” happen on different processor



Examples from the Linux kernel for mutex and spin locks

http://kcsl.ece.iastate.edu/linux-results/linux-kernel-3.19-rc1/

CRITICAL section PROBLEM SOLUTIONS

Semaphores

Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities

The **Semaphore S** is an integer variable and can only be accessed via two **indivisible (atomic)** operations `wait()` and `signal()`

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

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

The **Semaphore S** is an integer variable and can only be accessed via two **indivisible (atomic)** operations `wait()` and `signal()`

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

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

Binary Semaphore

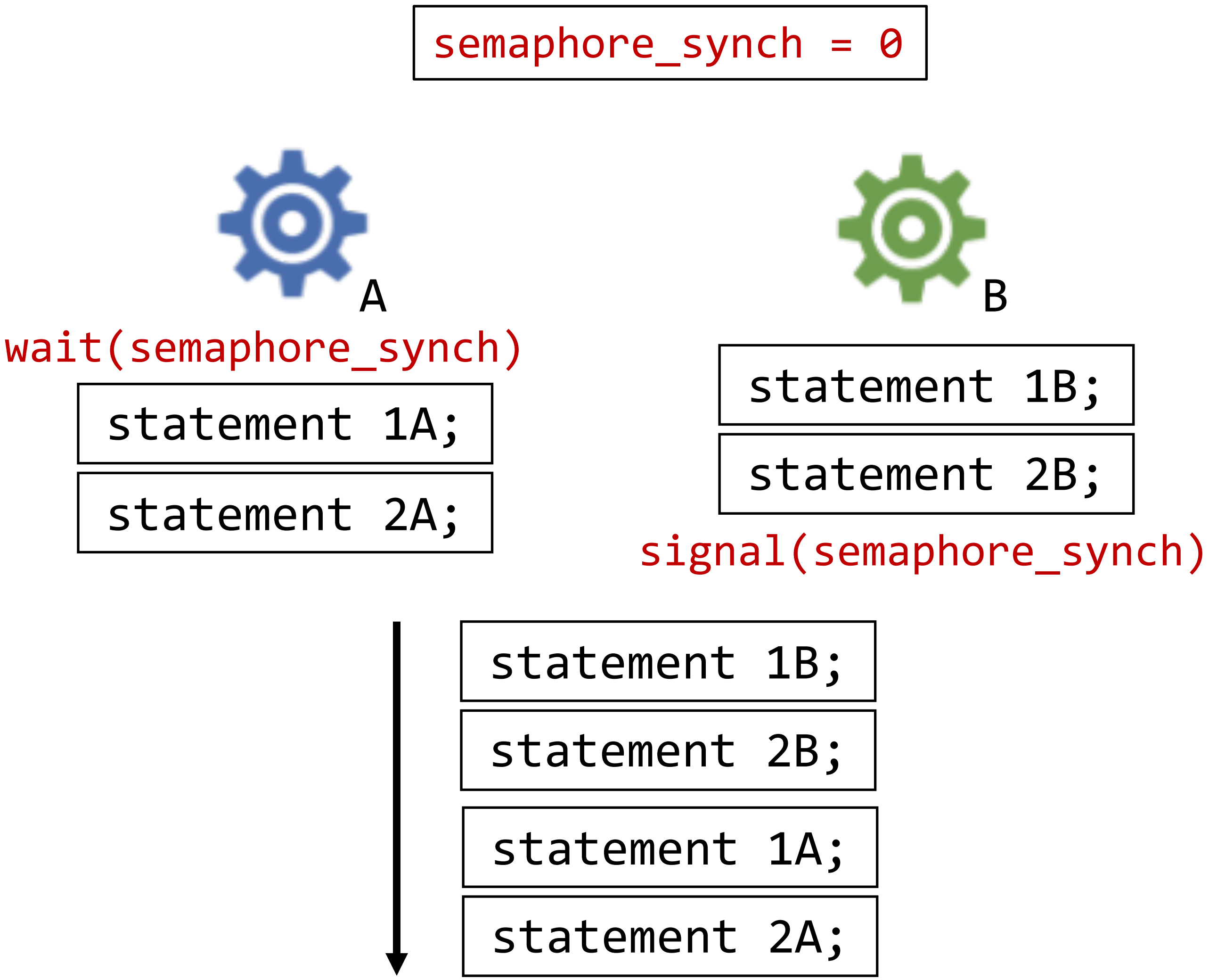
Semaphore S can be either 0 or 1 (Similar to mutex locks)

Counting Semaphore

Semaphore S can range over some domain values. For example: number of available resources to a set of processes

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

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



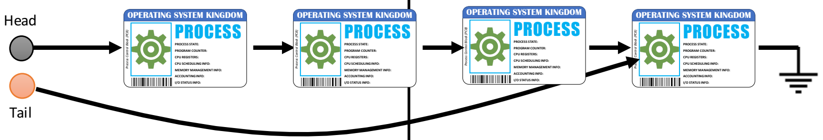
```
wait(S){
    while (S <= 0){
        // busy wait
    }
    S--;
}
```

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

This is very naive implementation that requires busy waiting
Wasting CPU Time

Can we implement a solution that **blocks** "*switches the process from running to waiting*" when its waiting for acquire the resource?

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

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

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



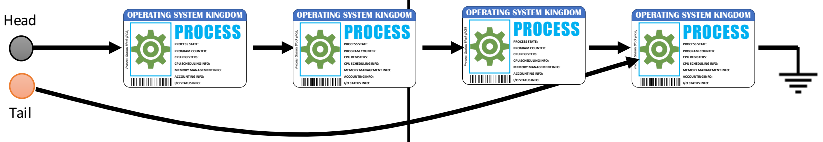
`list` – *A waiting queue of processes waiting for the semaphore*

`block()` – *places the process invoking the operation on the appropriate waiting queue*

`wakeup()` – *remove one of processes in the waiting queue and place it in the ready queue*

https://github.com/torvalds/linux/blob/master/include/linux/semaphore.h

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

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

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



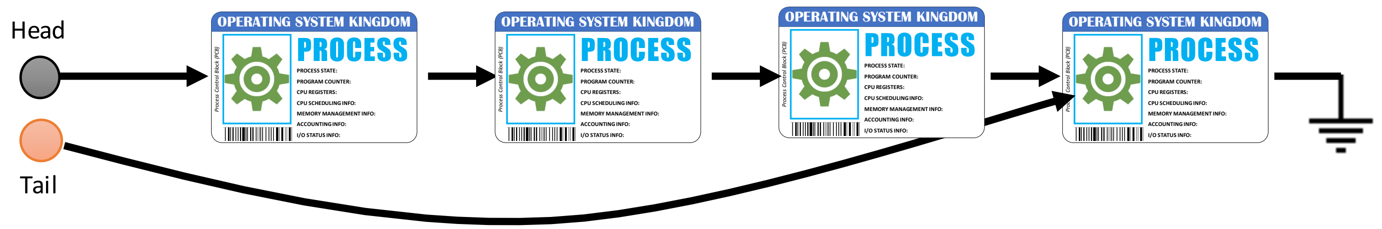
Can we have negative `value` for semaphore? and What does that represent?

The list should represent a queue that ensures **bounded-waiting** such as FIFO

Starvation

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

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

wait(semaphore *S) {
    S->value--;
    if (S->value < 0) {
        add this process to S->list;
        block();
    }
}

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



LIFO Queue

Classical Problems of Synchronization

test newly-proposed synchronization schemes

Bounded-Buffer Problem

Readers and Writers Problem

Dining-Philosophers Problem

Bounded-Buffer Problem

```
mutex = 1, full = 0, empty = n
```

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 buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next consumed */
} while (true);
```

n-buffers

mutex	1
full	0
empty	3

Producer

Consumer

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

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

Readers-Writers Problem

Read and writes to the database; they do perform updates

Only one single writer can access the database at the same time

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

Shared Dataset

```
rw_mutex = 1
mutex = 1
read_count = 0
```

Information shared among processes

Only read the database; they do not perform any updates

allow multiple readers to read at the same time

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

rw_mutex	1
mutex	1
read_count	0

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

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

Dining-Philosophers Problem

https://en.wikipedia.org/wiki/Dining_philosophers_problem

Philosophers spend their lives alternating **thinking** and **eating**

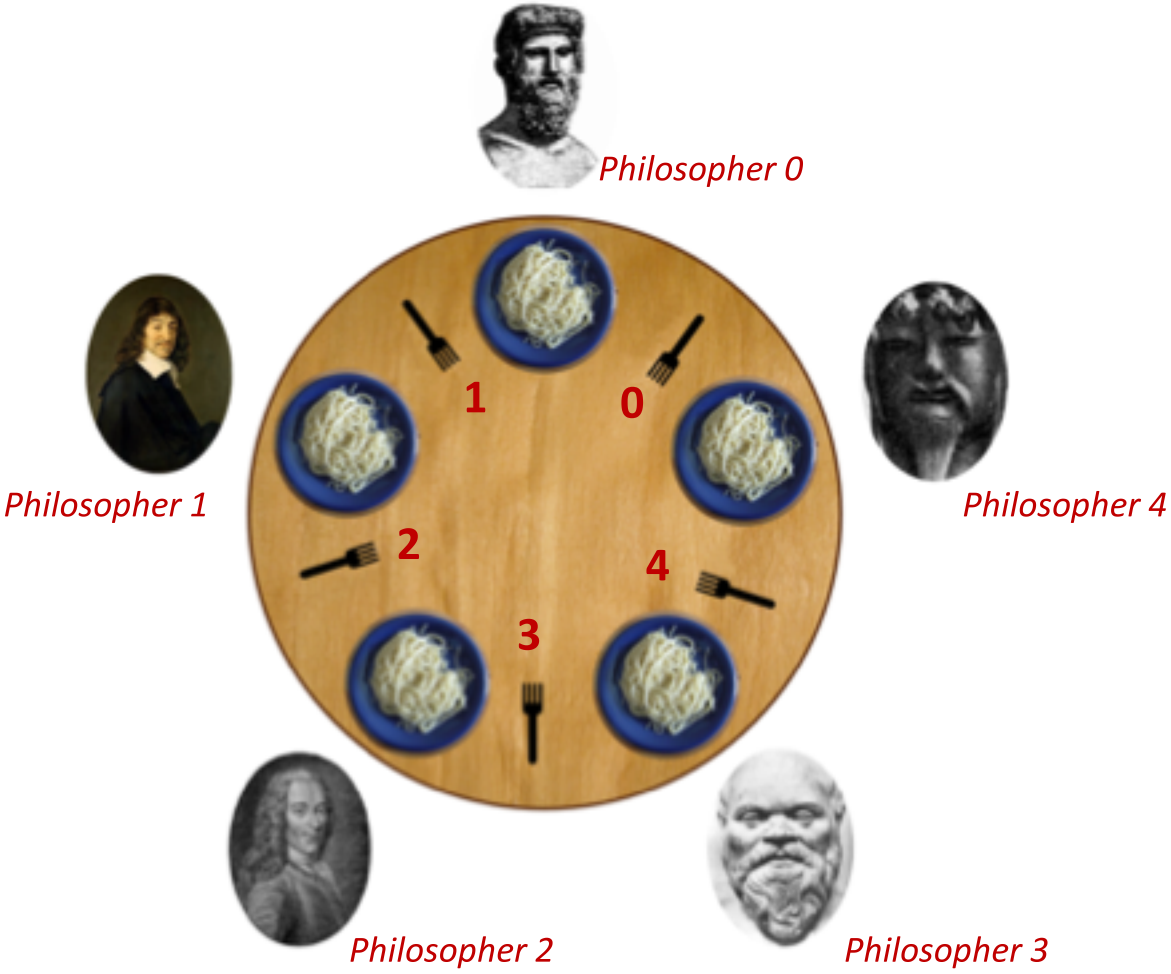
Need 2 forks (one at a time) to eat from bowl

Dining-Philosophers Problem

```
int fork[5] = {1, 1, 1, 1, 1}
```

Philosopher-i

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



Suppose that all five philosophers become **hungry at the same time** and each grabs her **left chopstick**. All the elements of chopstick will now be equal to 0.

When each philosopher tries to grab her right chopstick, she will be delayed forever

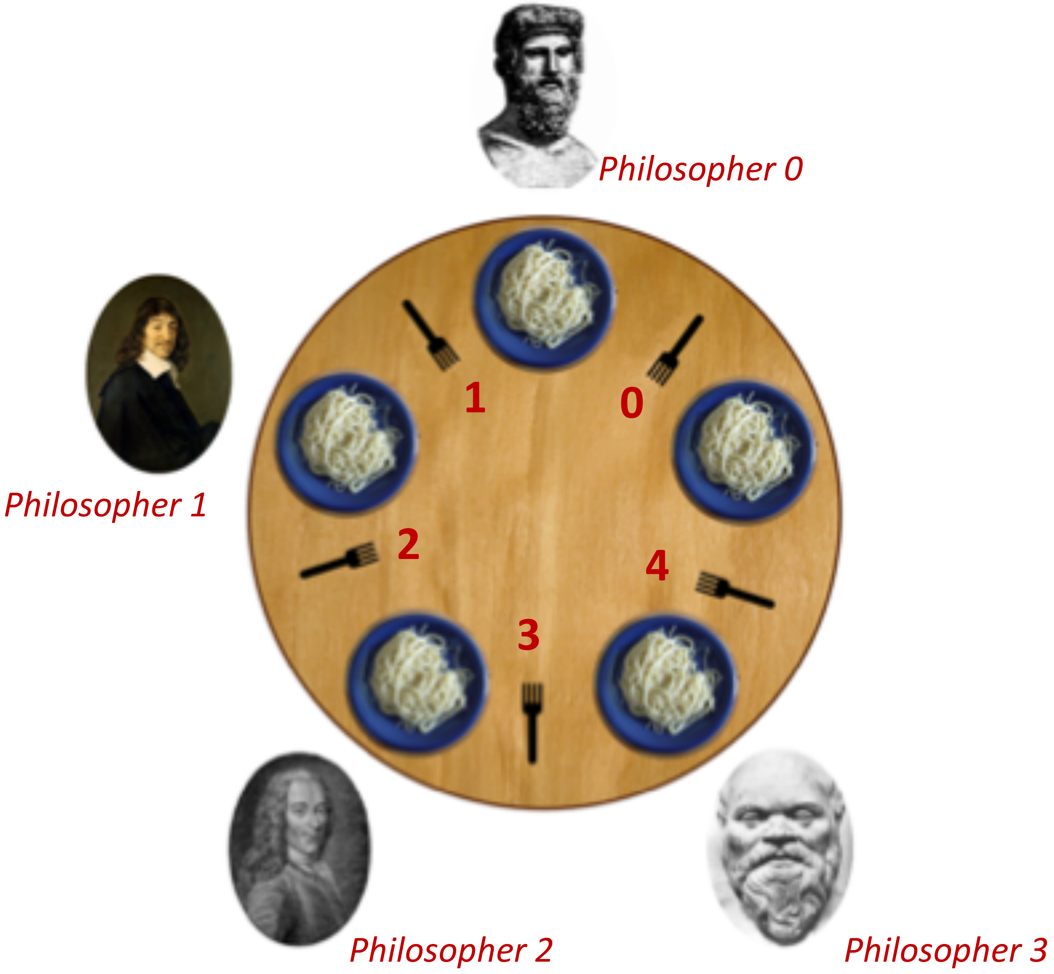
Dining-Philosophers Problem

Allow at most 4 philosophers to be sitting simultaneously at the table

```
int fork[5] = {1, 1, 1, 1, 1}
```

Philosopher-i

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



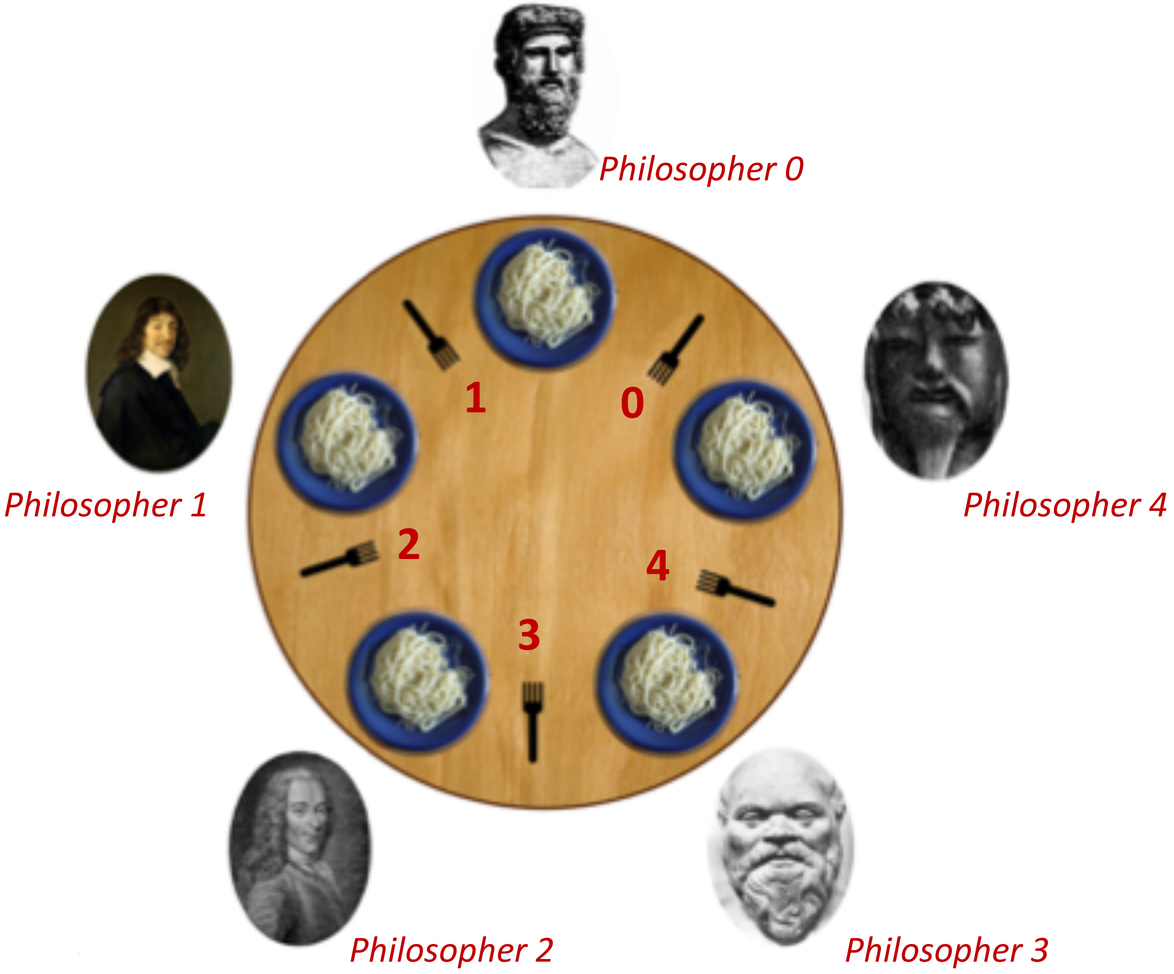
Dining-Philosophers Problem

Allow a philosopher to pick up the forks only if both are available (picking must be done in a critical section)

```
int fork[5] = {1, 1, 1, 1, 1}
int mutex = 1;
```

Philosopher-i

```
do{
    wait(mutex);
    // Start Critical Section
    wait(fork[i]);
    wait(fork[(i + 1) % 5]);
    signal(mutex);
    // End Critical Section
    eat();
    signal(fork[i]);
    signal(fork[(i + 1) % 5]);
    think();
} while(true);
```



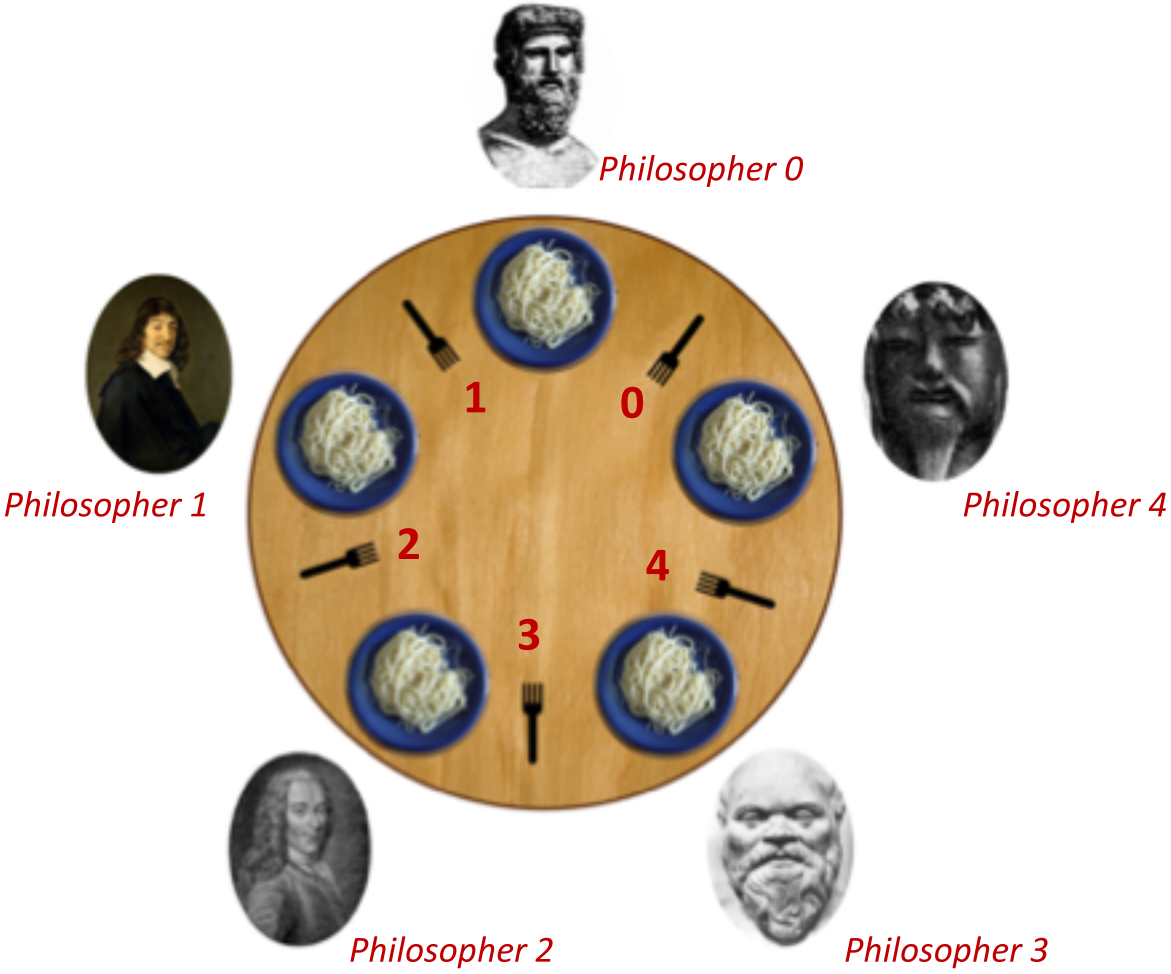
Dining-Philosophers Problem

Odd-numbered picks up left then right chopstick. Even-numbered picks up right then left chopstick

```
int fork[5] = {1, 1, 1, 1, 1}
```

Philosopher-i

```
do{
    if(i % 2 == 0){
        wait(fork[(i + 1) % 5]);
        wait(fork[i]);
    }else{
        wait(fork[i]);
        wait(fork[(i + 1) % 5]);
    }
    eat();
    signal(fork[i]);
    signal(fork[(i + 1) % 5]);
    think();
} while(true);
```




You release the lock first Once I have finished my task, you can continue.
Why should I? You release the lock first and wait until I complete my task.

Deadlock

Deadlock

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

```
co_printer = 1, bw_printer = 1
```

A

B

```
wait(co_printer);
```

```
wait(bw_printer);
```

```
wait(bw_printer);
```

```
wait(co_printer);
```

Deadlock Characterization

Deadlock can arise if four conditions hold simultaneously.

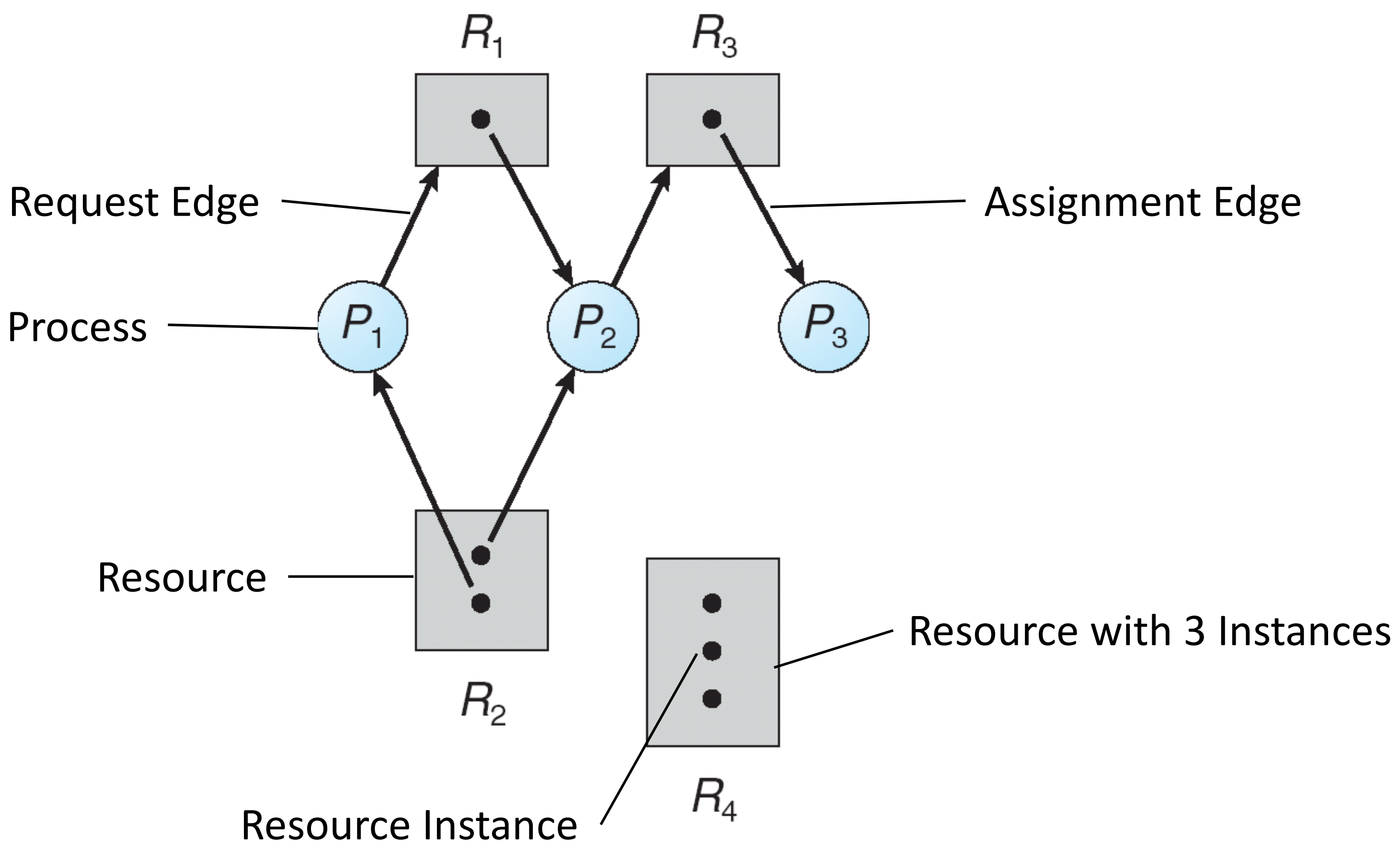
Mutual exclusion: only one process at a time can use a resource

Hold and wait: a process holding at least one resource is waiting to acquire additional resources held by other processes

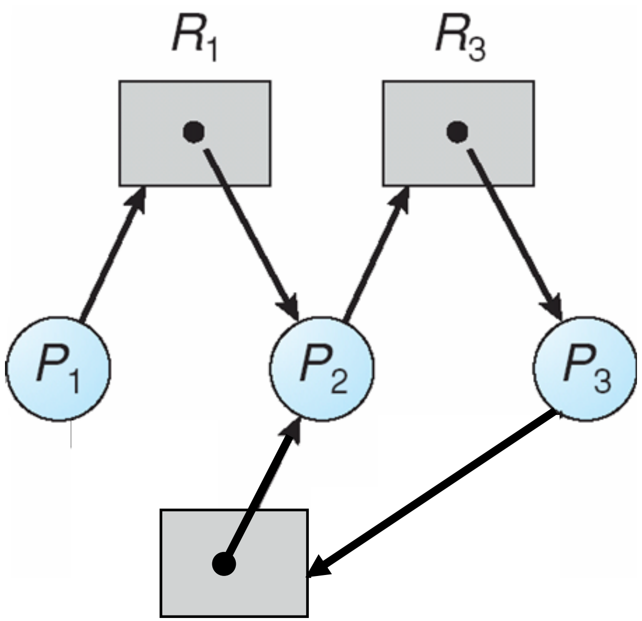
No preemption: a resource can be released only voluntarily by the process holding it, after that process has completed its task

Circular wait: there exists a set $\{P_0, P_1, \ldots, P_n\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by P_2, …, P_{n-1} is waiting for a resource that is held by P_n, and P_n is waiting for a resource that is held by P_0.

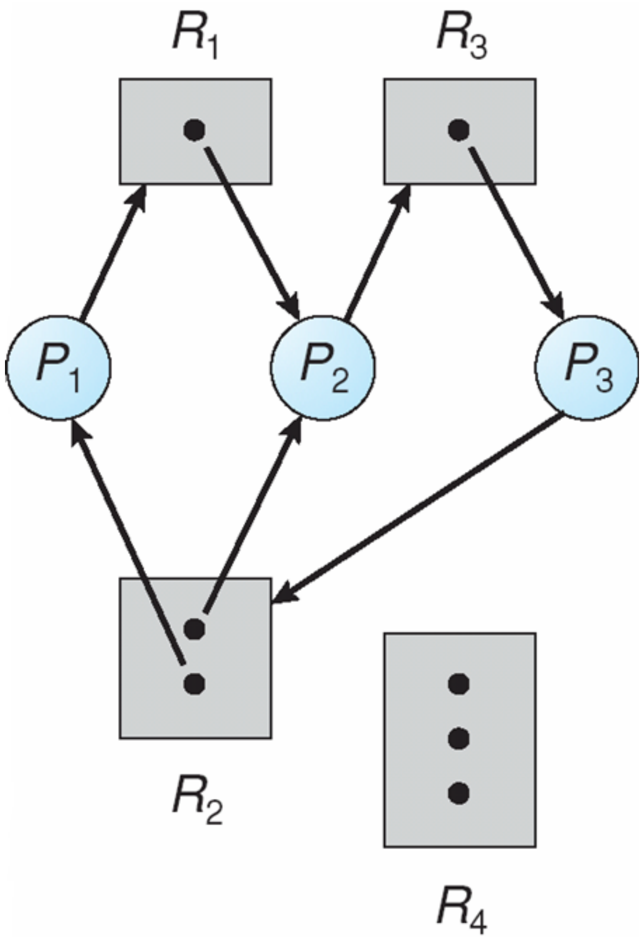
Resource Allocation Graph

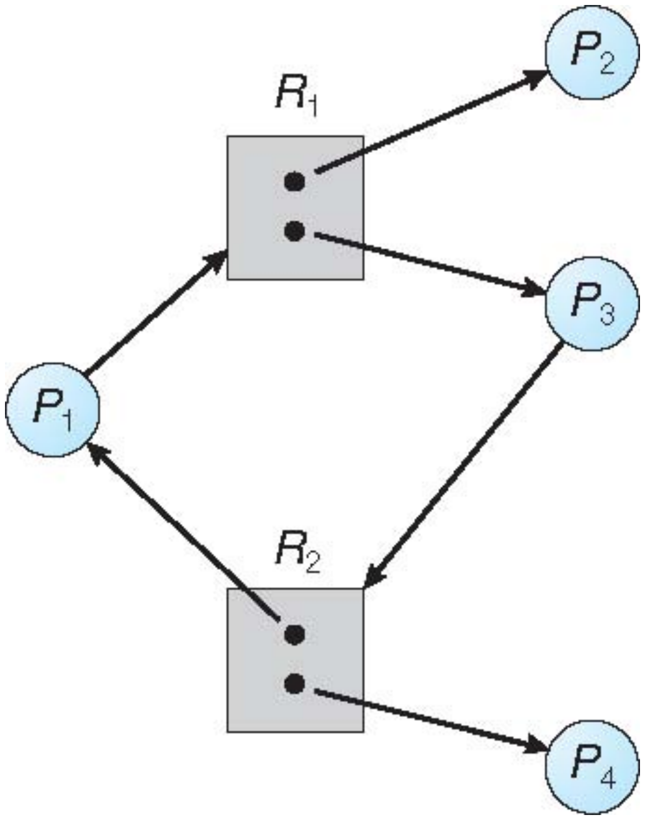


No Cycles ⇒ No Deadlock



If graph contains a **cycle** and **one instance** per resource $\Rightarrow$ **Deadlock**





If graph contains a **cycle** with **many instances** per resource $\Rightarrow$ Deadlock **possibility**

Methods for Handling Deadlocks

Ensure that the system will never enter a deadlock state via **Deadlock prevention** and **Deadlock avoidance**

Allow the system to enter a deadlock state and then recover

Ignore the problem and pretend that deadlocks never occur in the system; used by most operating systems, including UNIX

つづく