Operating Systems

Module 3

Process Synchronization

Outline

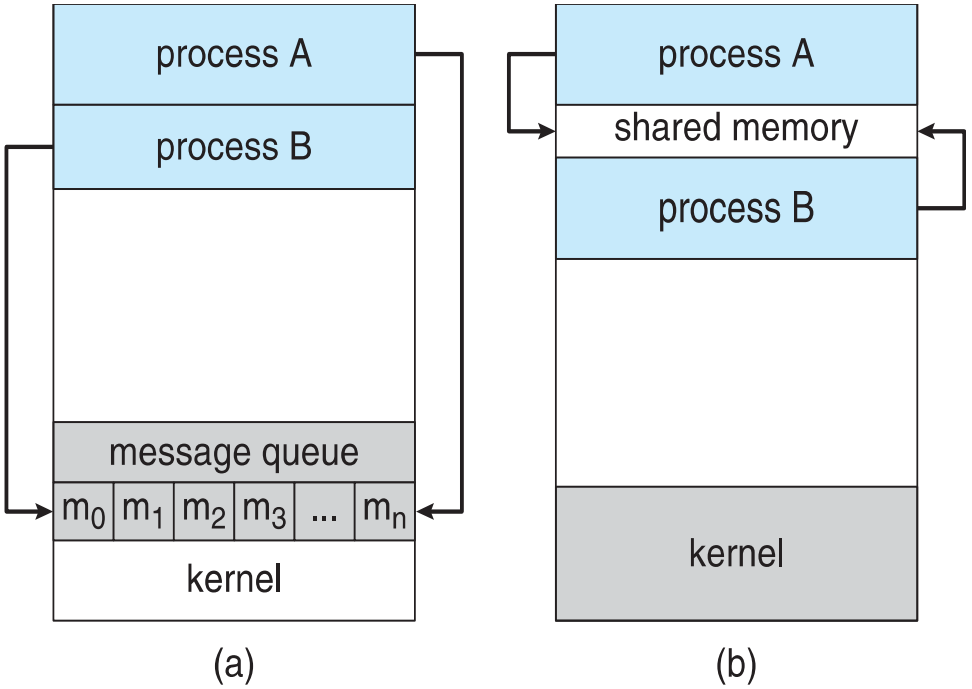
- Interprocess Communication
- Race condition
- Critical Section
- Semaphores

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Message passing. (b) shared memory.



Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Critical section

Motivating Scenario

shared variable

```
int counter=5;
```

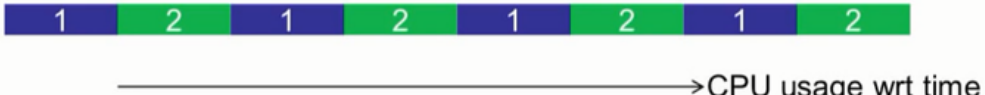
program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Single core
  - Program 1 and program 2 are executing at the same time but sharing a single core



Critical section

Motivating Scenario

Shared variable

```
int counter=5;
```

program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- What is the value of counter?
  - expected to be 5
  - but could also be 4 and 6

Critical section

Motivating Scenario

Shared variable

int counter=5;

program 0

```
{
  *
  *
counter++
  *
}
```

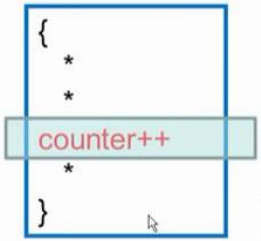
program 1

```
{
  *
  *
counter--
  *
}
```

	R1 ← counter	R1 ← counter	R2 ← counter
	R1 ← R1 + 1	R2 ← counter	R2 ← counter
	counter ←R1	R2 ←R2 - 1	R2 ←R2 + 1
context switch	R2 ← counter	counter ←R2	counter ←R2
	R2 ← R2 - 1	R1 ←R1 + 1	R2 ← R2 - 1
	counter ←R2	counter ← R1	counter ← R2
	counter = 5	counter = 6	counter = 4

Race Conditions

- Race conditions
  - A situation where several processes access and manipulate the same data (*critical section*)
  - The outcome depends on the order in which the access take place
  - Prevent race conditions by synchronization
    - Ensure only one process at a time manipulates the critical data



No more than one process should execute in critical section at a time

Race Conditions in Multicore

shared variable

```
int counter=5;
```

program 0

```
{
  *
  *
counter++
  *
}
```

program 1

```
{
  *
  *
counter--
  *
}
```

- Multi core
  - Program 1 and program 2 are executing at the same time on different cores

Critical Section Problem

- Consider system of ***n*** processes {p_0, p_1, … p_{n-1}}
- Each process has **critical section** segment of code
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in critical section, no other may be in its critical section
- ***Critical section problem*** is to design protocol to solve this
- Each process must ask permission to enter critical section in **entry section**, may follow critical section with **exit section**, then **remainder section**

Critical Section

- General structure of process P_i

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

Algorithm for Process P_i

```
do {
        while (turn == j);
                critical section
        turn = j;
                remainder section
} while (true);
```

Solution to Critical-Section Problem

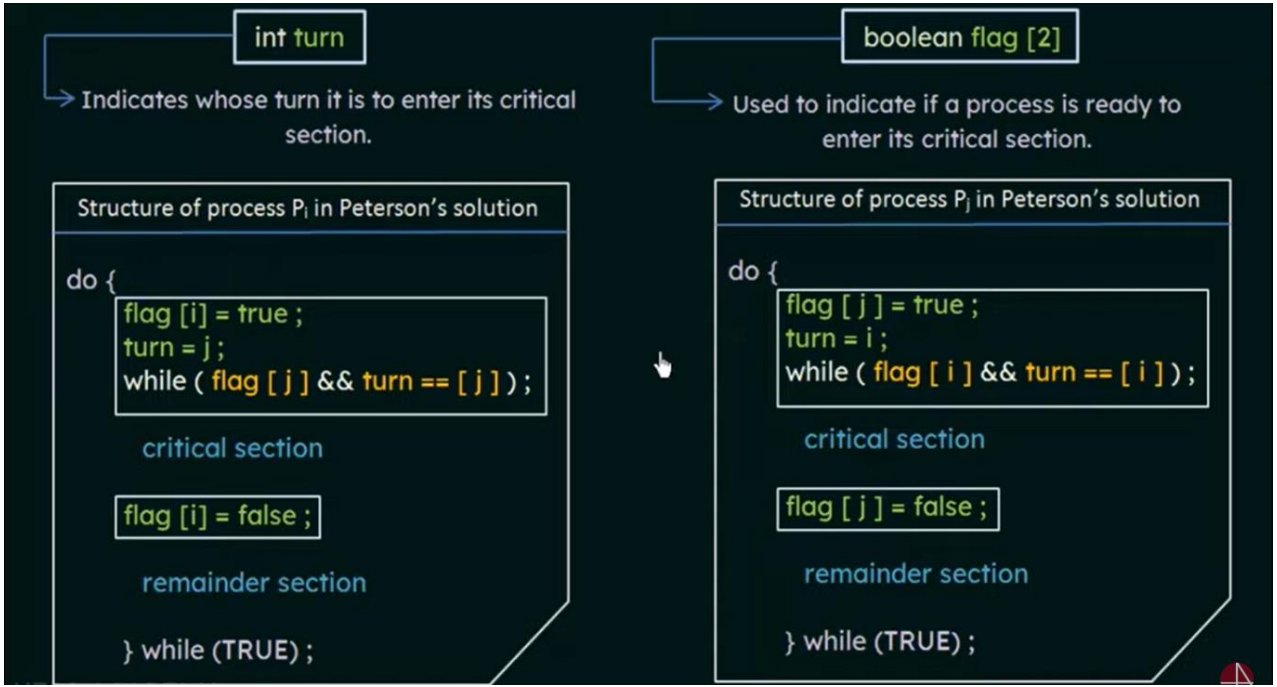
1. **Mutual Exclusion** - If process P_i is executing in its critical section, then no other processes can be executing in their critical sections
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

Peterson's Solution

- Good algorithmic description of solving the problem
- Two process solution
- Assume that the `load` and `store` machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - `int turn;`
  - `Boolean flag[2]`
- The variable `turn` indicates whose turn it is to enter the critical section
- The `flag` array is used to indicate if a process is ready to enter the critical section. `flag[i] = true` implies that process P_i is ready!

Algorithm for Process P_i

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


int turn
Indicates whose turn it is to enter its critical section.
boolean flag [2]
Used to indicate if a process is ready to enter its critical section.
Structure of process P_i in Peterson's solution
do {
flag [i] = true ;
turn = j ;
while (flag [j] && turn == [j]) ;
critical section
flag [i] = false ;
remainder section
} while (TRUE) ;
Structure of process P_j in Peterson's solution
do {
flag [j] = true ;
turn = i ;
while (flag [i] && turn == [i]) ;
critical section
flag [j] = false ;
remainder section
} while (TRUE) ;

Peterson's Solution (Cont.)

- Provable that the three CS requirement are met:
  1. Mutual exclusion is preserved

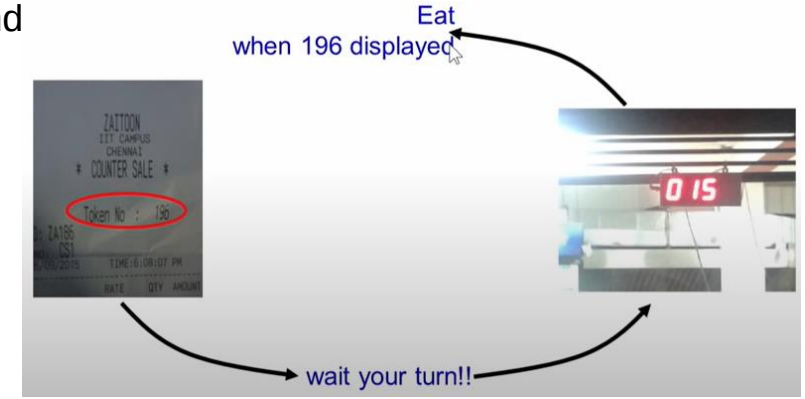
     P_i enters CS only if:

     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Bakery Algorithm in Process Synchronization

- The **Bakery algorithm**, by Leslie Lamport, is one of the simplest known solutions to the mutual exclusion problem for the general case of N process (N>2).
- Bakery Algorithm is a critical section solution for **N** processes. The algorithm preserves the first come first serve property.
- Before entering its critical section, the process receives a number. Holder of the smallest number enters the critical section.
- If processes Pi and

```
if i < j
Pi is served first;
else
Pj is served first.
```



Bakery Algorithm in Process Synchronization

- **Notation –** lexicographical order (ticket #, process id #) – Firstly the ticket number is compared. If same then the process ID is compared next, ie,

– (a, b) < (c, d) if a < c or if a = c and b < d

– max(a [0], . . ., a [n-1]) is a number, k, such that k >= a[i] for i = 0, . . ., n – 1

Shared data – choosing is an array [0..n – 1] of boolean values; & number is an array [0..n – 1] of integer values. Both are initialized to **False & Zero** respectively.

Bakery Algorithm in Process Synchronization

Simplified Bakery Algorithm (example)

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i);

critical section

unlock(i);

Remainder section
```

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1;
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

```
unlock(i){
    num[i] = 0;
}
```

P1	P2	P3	P4	**P5**
0	0	1	0	0

Bakery Algorithm in Process Synchronization

Simplified Bakery Algorithm (example)

Processes numbered 0 to N-1
num is an array N integers (initially 0).
Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

```
unlock(i){
    num[i] = 0;
}
```

P1	P2	P3	P4	P5
0	4	0	2	3

Bakery Algorithm in Process Synchronization

Simplified Bakery Algorithm

- Processes numbered 0 to N-1
- num is an array N integers (initially 0).
  - Each entry corresponds to a process

```
lock(i){
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    for(p = 0; p < N; ++p){
        while (num[p] != 0 and num[p] < num[i]);
    }
}
```

critical section

```
unlock(i){
    num[i] = 0;
}
```

This is at the doorway!!!
It has to be atomic
to ensure two processes
do not get the same token

Original Bakery Algorithm

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

Choosing ensures that a process
Is not at the doorway
i.e., the process is not 'choosing'
a value for num

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

P1	P2	P3	P4	**P5**
0	0	0	0	0

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

43

Original Bakery Algorithm (example)

- Without atomic operation assumptions
- Introduce an array of N Booleans: *choosing*, initially all values False.

```
lock(i){
    choosing[i] = True
    num[i] = MAX(num[0], num[1], …., num[N-1]) + 1
    choosing[i] = False
    for(p = 0; p < N; ++p){
        while (choosing[p]);
        while (num[p] != 0 and (num[p],p)<(num[i],i));
    }
}
```

doorway

critical section

```
unlock(i){
    num[i] = 0;
}
```

P1	P2	P3	P4	P5
0	3	1	2	2

(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))

44

Bakery Algorithm

```
repeat
    choosing[i] := true;
    number[i] := max(number[0], number[1], ..., number[n - 1])+1;
    choosing[i] := false;
    for j := 0 to n - 1
        do begin
            while choosing[j] do no-op;
            while number[j] != 0
                and (number[j], j) < (number[i], i) do no-op;
        end;

        critical section

    number[i] := 0;

        remainder section

until false;
```

Bakery Algorithm

- Firstly the process sets its “choosing” variable to be TRUE indicating its intent to enter critical section.
- Then it gets assigned the highest ticket number corresponding to other processes. Then the “choosing” variable is set to FALSE indicating that it now has a new ticket number. This is in-fact the most important and confusing part of the algorithm. It is actually a small critical section in itself !
- The very purpose of the first three lines is that if a process is modifying its TICKET value then at that time some other process should not be allowed to check its old ticket value which is now obsolete. This is why inside the for loop before checking ticket value we first make sure that all other processes have the “choosing” variable as FALSE.
- After that we proceed to check the ticket values of processes where process with least ticket number/process id gets inside the critical section.
- The exit section just resets the ticket value to zero.

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- 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

- Notation <≡ lexicographical order (ticket #, process id #)
  - $(a,b) < c,d)$ if $a < c$ or if $a = c$ and $b < d$
  - max $(a_0,\ldots, a_{n-1})$ is a number, k, such that $k \geq a_i$ for i - 0, …, $n - 1$
- Shared data

  boolean choosing[n];

  int number[n];

  Data structures are initialized to **false** and **0** respectively

Operating System Concepts

Bakery Algorithm

```
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,j] < number[i,i])) ;
    }
        critical section
    number[i] = 0;
        remainder section
} while (1);
```

Mutex Locks

- OS designers build software tools to solve critical section problem
- Simplest is **mutex** lock
- Protect a critical section by first `acquire()` a lock then `release()` the lock
  - Boolean variable indicating if lock is available or not
- Calls to `acquire()` and `release()` must be atomic
  - Usually implemented via hardware atomic instructions
- But this solution requires **busy waiting**
  - This lock therefore called a **spinlock**

Lock Variable

```
while ( Lock != 0 ) ;
Lock = 1
```
Entry Section

```
Critical Section
```

```
Lock = 0
```
Exit Section

- Software solution implemented in user mode
- N>2
- Entry section = acquire()
- Exit section = release()
- No guarantee of mutual exclusion

Synchronization Hardware

- Test and modify the content of a word atomically
- while(test_and_set(&lock));

  CS

  lock = FALSE;

- Boolean test_and_set(Boolean *target){

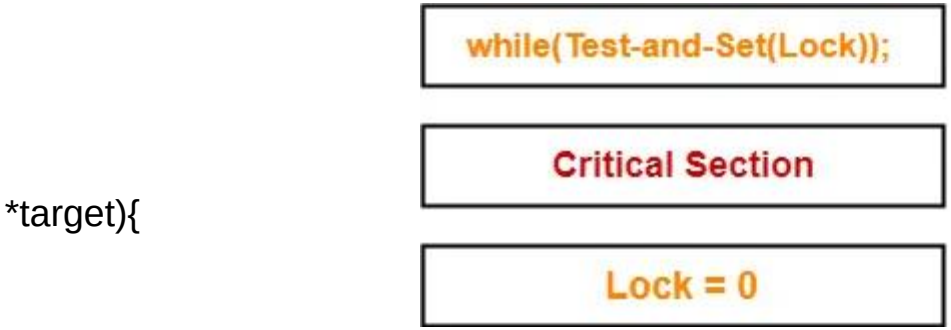
  boolean r = *target;

  *target = TRUE;

  return r;

}

- Mutual Exclusion and progress is achieved.



.

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- Process P_i

```
do {
    while (TestAndSet(lock)) ;
        critical section
    lock = false;
        remainder section
}
```

Operating System Concepts

Synchronization Hardware

- Atomically swap two variables.

```
void Swap(boolean &a, boolean &b) {
    boolean temp = a;
    a = b;
    b = temp;
}
```

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

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

- Process P_i

```
do {
    key = true;
    while (key == true)
            Swap(lock,key);
        critical section
    lock = false;
        remainder section
}
```

Operating System Concepts

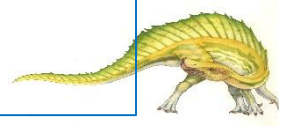
Semaphore

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**
- Definition of the wait() operation

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

- Definition of the **`signal()` operation**

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



Semaphore Usage

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**
- Can solve various synchronization problems
- Must guarantee that no two processes can execute the `wait()` and `signal()` on the same semaphore at the same time
- Thus, the implementation becomes the critical section problem where the `wait` and `signal` code are placed in the critical section

Classical Problems of Synchronization

- Bounded-Buffer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Problem

- Shared data

  Counting semaphore full – no. of filled slots

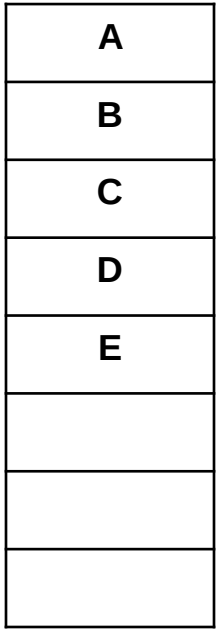
  Counting semaphore empty – no. of empty slots

  Binary semaphore mutex

- Initially:

  full = 0, empty = N, mutex = 1

N=8



Bounded-Buffer Problem

Producer Process

```
do {
        …
    produce an item in nextp
        …
    wait(empty);
    wait(mutex);
        …
    //add nextp to buffer
        Buffer[IN] = nextp;
        IN = (IN+1) % N
        …
    signal(mutex);
    signal(full);
} while (1);
```

Consumer Process

```
do {
    wait(full);
    wait(mutex);
            …
    //remove an item from buffer to nextc
        nextc = Buffer[OUT];
        OUT = (OUT+1) % N
            …
    signal(mutex);
    signal(empty);
            …
    //consume the item in nextc
            …
} while (1);
```

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1,
  int readcount = 0

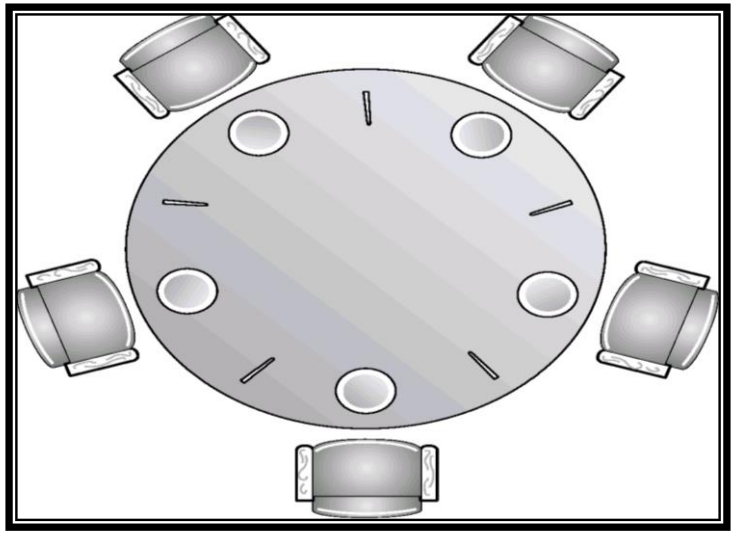
Readers-Writers Problem Reader Process

```
semaphore mutex = 1;
semaphore wrt = 1;
int readcount = 0;
```

```
void Reader(){
    wait(mutex);
    readcount++;
    if (readcount == 1)
        wait(wrt);
    signal(mutex);
        …
        reading is performed
        …
    wait(mutex);
    readcount--;
    if (readcount == 0)
        signal(wrt);
    signal(mutex);
}
```

```
void Writer(){
    wait(wrt);
        …
        writing is performed
        …
    signal(wrt);
}
```

Dining-Philosophers Problem



- Shared data

  semaphore chopstick[5];

  Initially all values are 1

Dining-Philosophers Problem

- Philosopher *i*:

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

Classical synchronization problems

READERS-WRITERS PROBLEM

CRITICAL SECTION PROBLEM

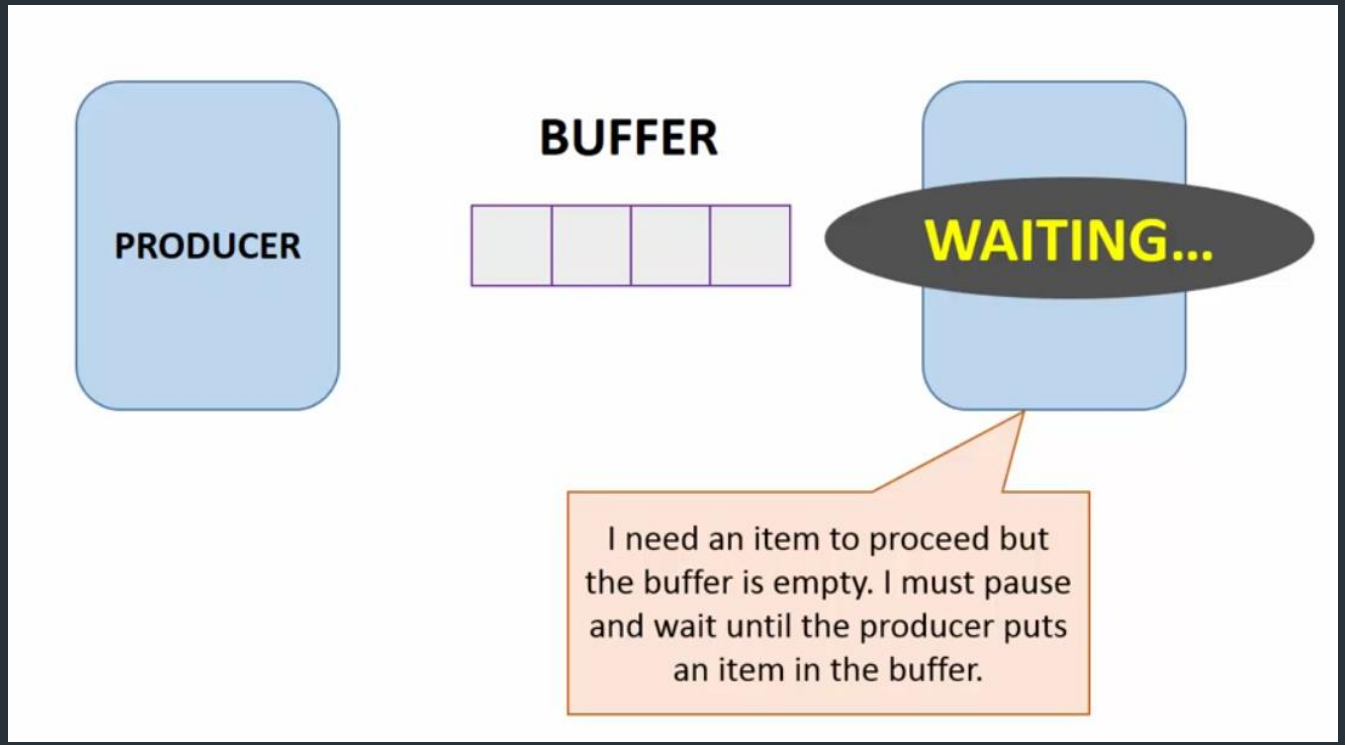
SLEEPING BARBER PROBLEM

CIGARETTE SMOKERS PROBLEM

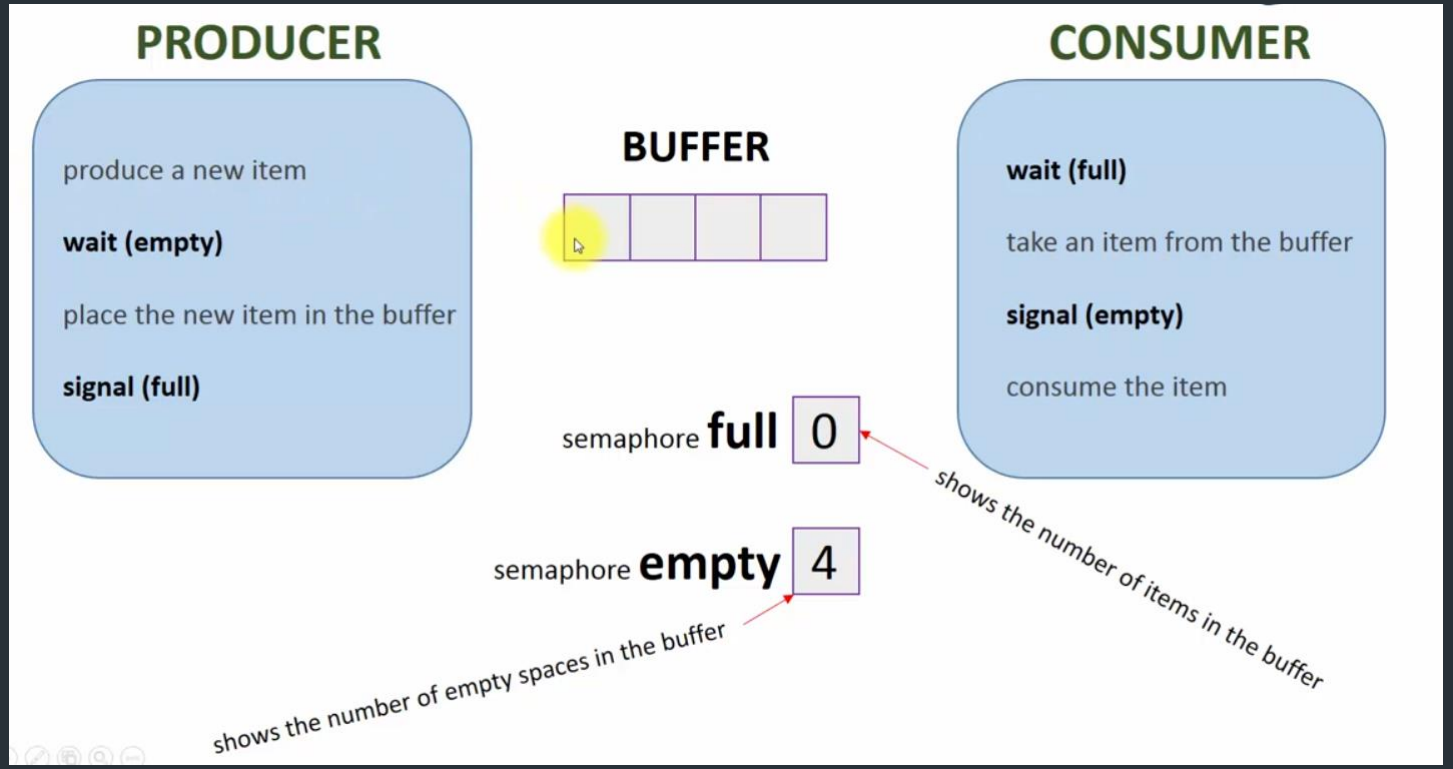
DINING PHILOSOPHERS PROBLEM

PRODUCER-CONSUMER PROBLEM

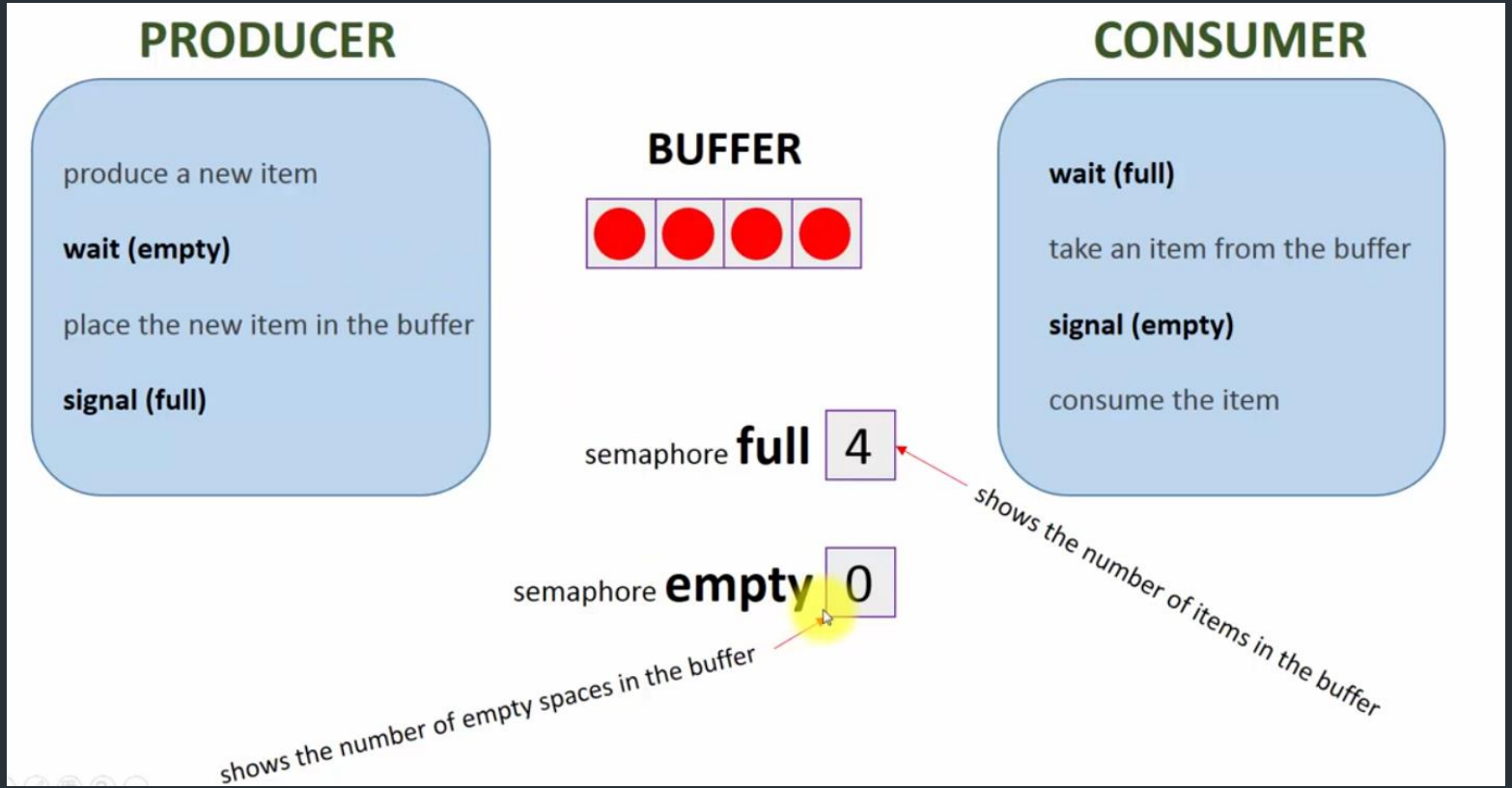
Producer consumer problem



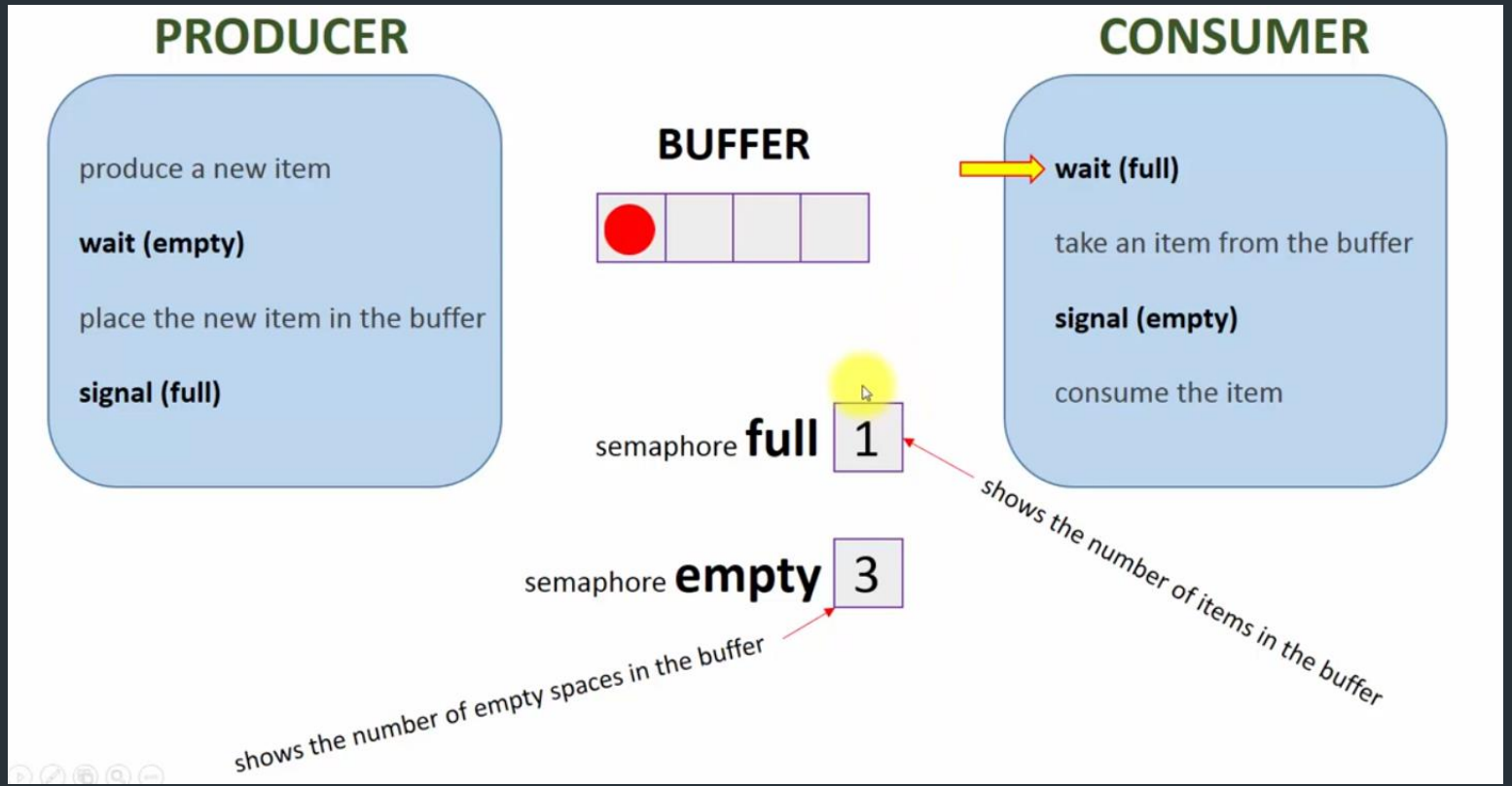
Producer consumer problem



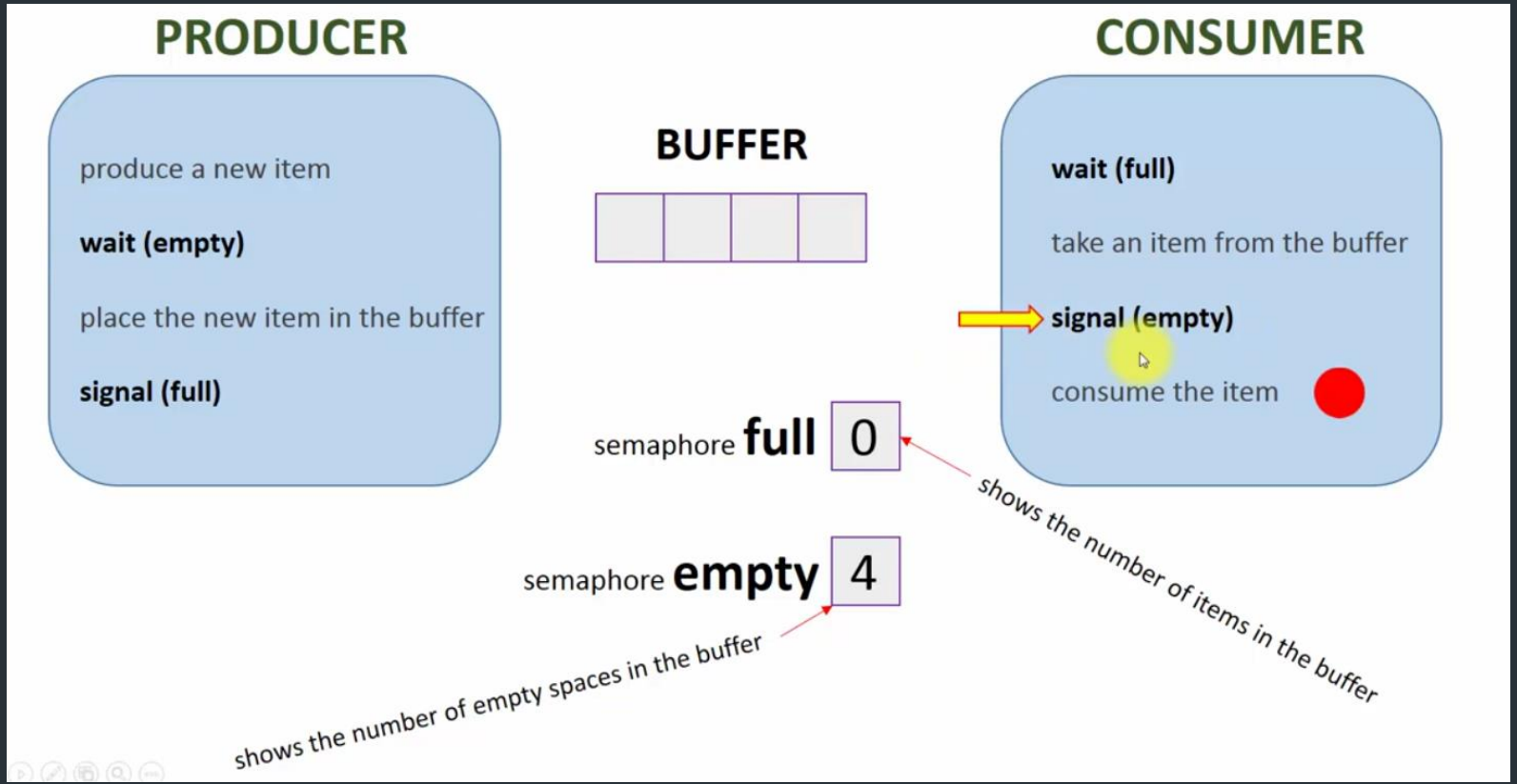
Producer consumer problem



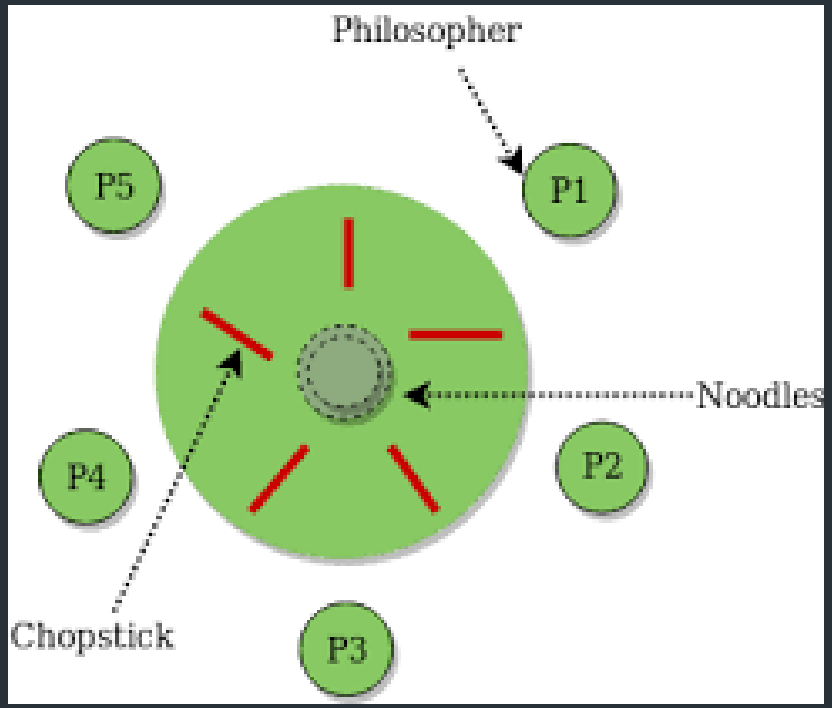
Producer consumer problem



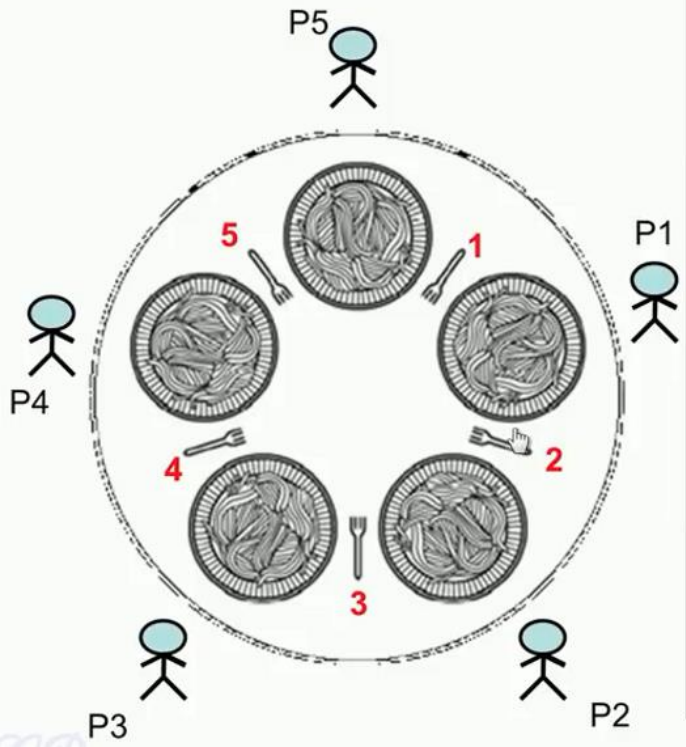
Producer consumer problem



Dining philosophers problem



Dining Philosophers Problem



- **Philosophers either think or eat**
- To eat, a philosopher needs to hold both forks (the one on his left and the one on his right)
- If the philosopher is not eating, he is thinking.
- Problem Statement : Develop an algorithm where no philosopher starves.

Dining-Philosophers Problem

Solution using Mutex

- Protect critical sections with a mutex
- Prevents deadlock
- But has performance issues
  - Only one philosopher can eat at a time

```
#define N 5

void philosopher(int i){
  while(TRUE){
     think(); // for some_time
     lock(mutex);
     take_fork(Ri);
     take_fork(Li);
     eat();
     put_fork(Li);
     put_fork(Ri);
     unlock(mutex);
  }
}
```

Dining-Philosophers Problem

Solution with Semaphores

Uses N semaphores (s[1], s[2], …., s[N]) all initialized to 0, and a mutex
Philosopher has 3 states: HUNGRY, EATING, THINKING
A philosopher can only move to EATING state if neither neighbor is eating

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

Dining-Philosophers Problem

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	T	T	T
semaphore	0	0	0	0	0

Dining-Philosophers Problem

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	H	T	T
semaphore	0	0	0	0	0

Dining-Philosophers Problem

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

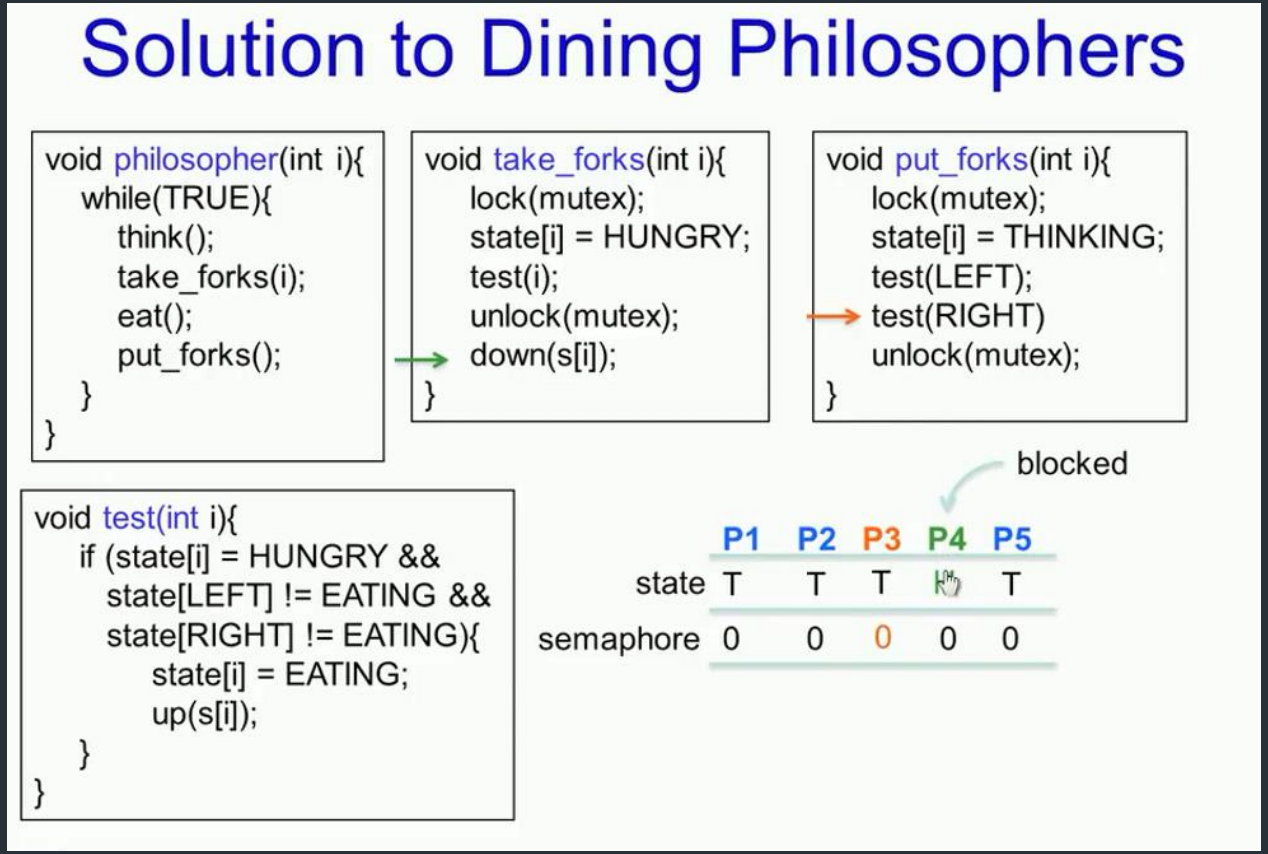
```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

s[i] is 1, so down will not block.
The value of s[i] decrements by 1.

```
void test(int i){
    if (state[i] = HUNGRY &&
       state[LEFT] != EATING &&
       state[RIGHT] != EATING){
           state[i] = EATING;
           up(s[i]);
    }
}
```

	P1	P2	P3	P4	P5
state	T	T	E	T	T
semaphore	0	0	1	0	0

Dining-Philosophers Problem



Dining-Philosophers Problem

Solution to Dining Philosophers

```
void philosopher(int i){
    while(TRUE){
        think();
        take_forks(i);
        eat();
        put_forks();
    }
}
```

```
void take_forks(int i){
    lock(mutex);
    state[i] = HUNGRY;
    test(i);
    unlock(mutex);
    down(s[i]);
}
```

```
void put_forks(int i){
    lock(mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT)
    unlock(mutex);
}
```

```
void test(int i){
    if (state[i] = HUNGRY &&
        state[LEFT] != EATING &&
        state[RIGHT] != EATING){
            state[i] = EATING;
            up(s[i]);
    }
}
```

blocked

	P1	P2	P3	P4	P5
state	T	T	T	E	T
semaphore	0	0	0	1	0

Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
        shared variable declarations
        procedure body P1 (...) {
            . . .
        }
        procedure body P2 (...) {
            . . .
        }
        procedure body Pn (...) {
            . . .
        }
        {
            initialization code
        }
}
```

Monitors

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

    x.wait();

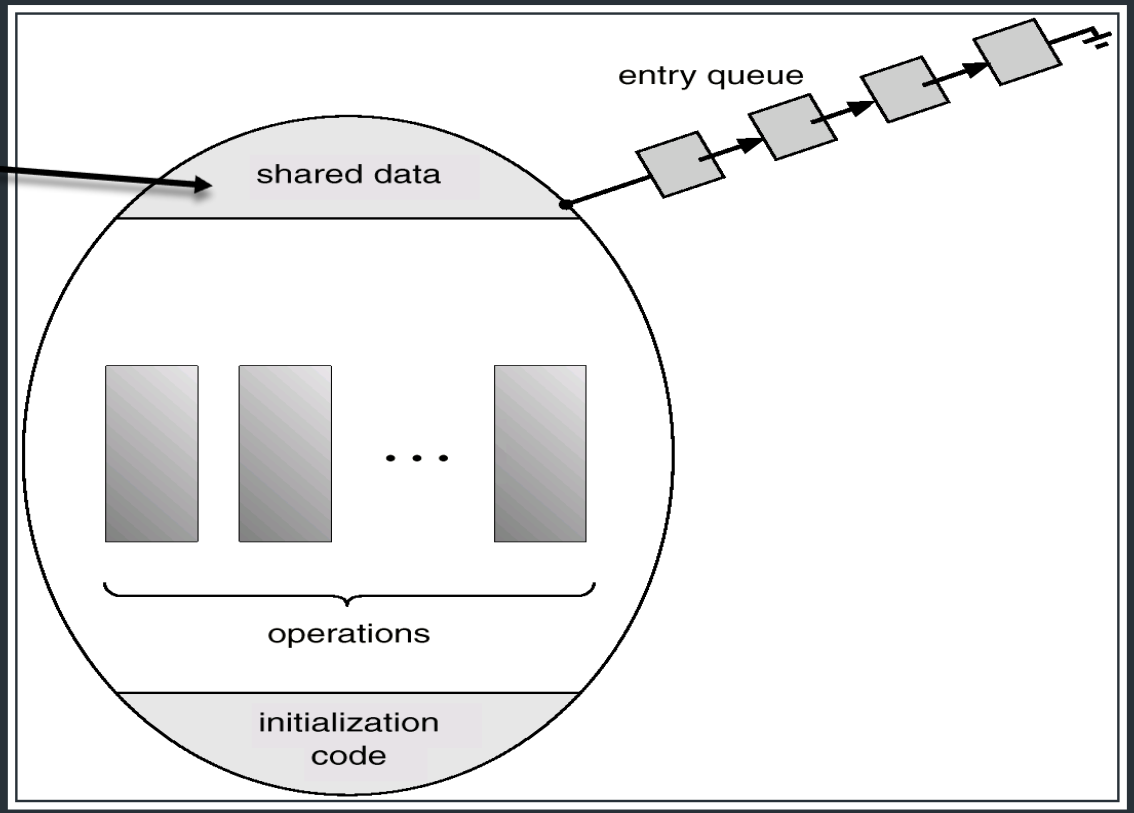
    means that the process invoking this operation is suspended until another process invokes

    x.signal();

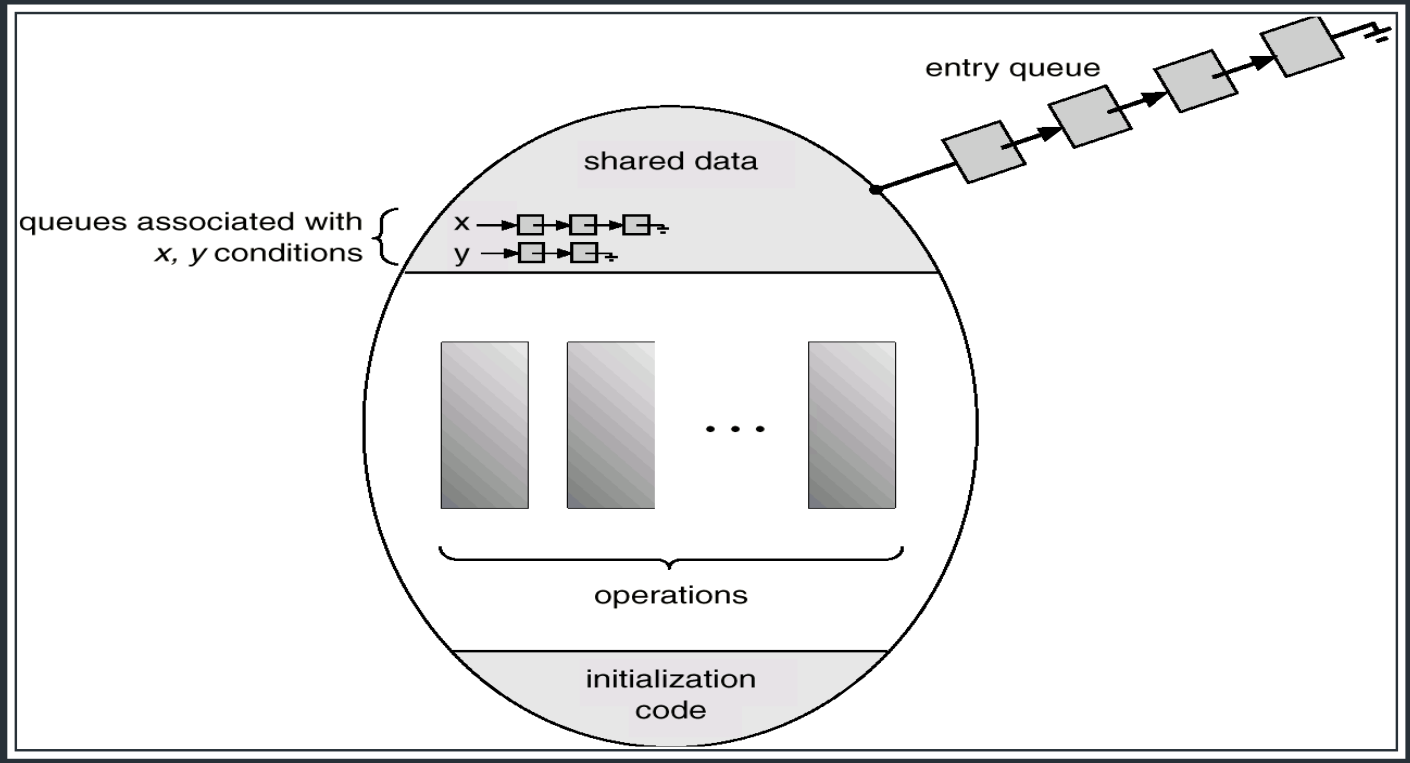
  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Schematic View of a Monitor

Queues associated with x, y conditions



Monitor With Condition Variables



Dining Philosophers Example


```
monitor dp
{
    enum {thinking, hungry, eating} state[5];
    condition self[5];
    void pickup(int i)
    void putdown(int i)
    void test(int i)
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = thinking;
    }
}
```

```
void pickup(int i) {
        state[i] = hungry;
        test[i];
        if (state[i] != eating)
            self[i].wait();
}

void putdown(int i) {
        state[i] = thinking;
        // test left and right neighbors
        test((i+4) % 5);
        test((i+1) % 5);
}
void test(int i) {
        if ( (state[(I + 4) % 5] != eating) &&
          (state[i] == hungry) &&
          (state[(i + 1) % 5] != eating)) {
            state[i] = eating;
            self[i].signal();
        }
}
```

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next-count = 0;
```

- Each external procedure ***F*** will be replaced by

```
wait(mutex);
   …
  body of F;
   …
if (next-count > 0)
   signal(next)
else
   signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation

- For each condition variable ***x***, we have:

```
semaphore x-sem; // (initially = 0)
int x-count = 0;
```

- The operation **x.wait** can be implemented as:

```
x-count++;
if (next-count > 0)
    signal(next);
else
    signal(mutex);
wait(x-sem);
x-count--;
```

Monitor Implementation

- The operation **x.signal** can be implemented as:

```
if (x-count > 0) {
    next-count++;
    signal(x-sem);
    wait(next);
    next-count--;
}
```

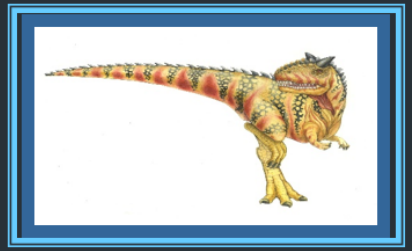
Monitor Implementation

- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - when **x.signal** is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

References

1. Abraham Silberschatz, Peter B. Galvin, Greg Gagne-Operating System Concepts, Wiley (2018).
2. Ramez Elmasri, A.Gil Carrick, David Levine, Operating Systems, A Spiral Approach - McGrawHill Higher Education (2010).
3. https://pdos.csail.mit.edu/6.828/2012/lec/l-lockfree.txt
4. https://en.wikipedia.org/wiki/Non-blocking_algorithm

Thank you



RR solution
Gantt chart

Process	A.T	B.T
A	0	6
B	1	5
C	3	4