

CS330

Operating Systems

SYNCHRONIZATION

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

TOPICS

- Race condition
- Critical Section
 - Software solutions
 - Hardware solutions
- Mutex locks and semaphores
- Classical examples of semaphores
- Monitors

Credit:

These notes are extracted from the following resources:

- Silberschatz, Galvin and Gagne, *Operating System Concepts*, 10 ed, Wiley, 2018
- Remzi H. Arpaci-Dusseau and Andrea C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, CreateSpace, 2018

CS330

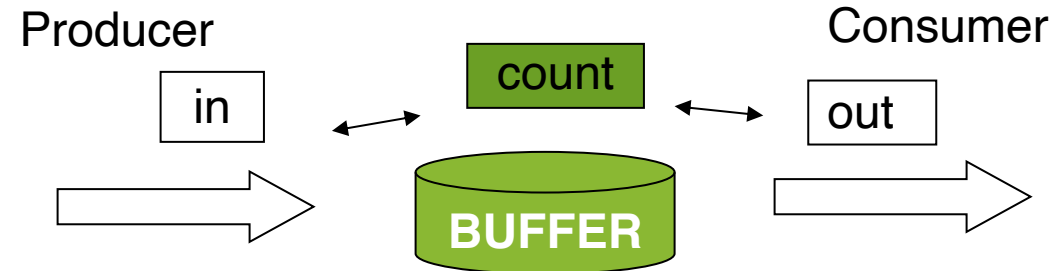
Operating Systems

RACE CONDITION

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

PRODUCER CONSUMER PROBLEM

- A **Producer** is a process that produces information that is consumed by a **Consumer** process.



Example:

- Producer wants to **insert** an item.
 - Consumer wants to **remove** an item.
 - If the **buffer is full**, the producer must wait.
 - If the **buffer is empty**, the consumer must wait.
- **Concurrent** access to **shared data** may result in data **inconsistency**. Maintaining data consistency requires mechanisms to ensure *the orderly execution of cooperating processes*.

PRODUCER CONSUMER PROBLEM

Producer

```
while (true) {  
    /* produce an item in next produced */  
  
    while (count == BUFFER SIZE)  
        {;} /* do nothing */  
  
    buffer[in] = next produced;  
    in = (in + 1) % BUFFER SIZE;  
    count++;  
}
```

Consumer

```
while (true) {  
  
    while (count == 0)  
        {;} /* do nothing */  
  
    next consumed = buffer[out];  
    out = (out + 1) % BUFFER SIZE;  
  
    count--;  
    /* consume the item in next consumed */  
}
```

RACE CONDITION

A **race condition** occurs when two or more processes or threads **access** and **modify *shared data*** **at the same time**, and the final result **depends** on the *order* in which they *execute*.

Lets assume your salary (10000 SR) is deposited in your bank account. You have two expenses: Rent (5000 SR) and Loan payment (2000 SR).

Two bank transactions need to deduct money from your account.

- Process P1: Deduct 5000 SR
- Process P2: Deduct 2000 SR
- Final Balance = 3000 SR



RACE CONDITION

Imagine both processes execute this code:

```
/*  
READ balance  
SUBTRACT amount (balance = balance - amount)  
WRITE balance  
*/
```

Process P1 (Rent)	Process P2 (Loan)
1.Read balance = 10,000	
	2. Read balance = 10,000
3. Calculate $10,000 - 5,000 = 5,000$	
	4. Calculate $10,000 - 2,000 = 8,000$
5.Write balance = 5,000	
	6.Write balance = 8,000

Incorrect final balance = 8000

RACE CONDITION

Why is this a race condition?

- Both processes are racing to update the same shared variable, **balance**.
- The final value depends on which process writes last.
 - If P1 writes last $\rightarrow$ balance = 5,000 SAR.
 - If P2 writes last $\rightarrow$ balance = 8,000 SAR.
- Both are incorrect because one deduction is lost.

RACE CONDITION

- What if P2 executes before P1?

Process P1 (Rent)	Process P2 (Loan)
	1. Read balance = 10,000
2. Read balance = 10,000	
	3. Calculate $10,000 - 2,000 = 8,000$
4. Calculate $10,000 - 5,000 = 5,000$	
	5. Write balance = 8,000
6. Write balance = 5,000	

Incorrect final balance = 5000

RACE CONDITION

The code that modifies the shared salary balance is a **critical section**:

```
balance = balance - amount
```

Solution: Make sure only **ONE** process is in critical section.

The OS uses synchronization mechanisms such as:

- **Mutex** (Mutual Exclusion): Only one process/thread accesses critical section at a time.
- **Semaphore**: Controls access to shared resources.
- **Monitor**: Provides synchronized access to shared data.

CRITICAL SECTION PROBLEM

- Consider system of n processes $\{p_0, p_1, \dots, p_{n-1}\}$
- 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**

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

CRITICAL-SECTION PROBLEM (CONT.)

Requirements for 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. [ONLY ONE process in C.S at a time]**
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 process that will enter the critical section next **cannot be postponed indefinitely [if C.S is available do not WAIT to enter C.S]**
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**[Give each waiting process a turn. No process should WAIT forever (Starvation)]**

Mutual Exclusion ensures **safety**.

Progress ensures **no unnecessary delay**.

Bounded Waiting ensures **fairness**.

CS330

Operating Systems

**CRITICAL SECTION
PROBLEM: SOFTWARE
SOLUTIONS**

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

INTERRUPT-BASED SOLUTION

On a single CPU, the OS scheduler typically relies on timer interrupts to preempt a running process. If interrupts are disabled, the timer interrupt cannot preempt the process during that interval.

- Entry section: **disable** interrupts
- Exit section: **enable** interrupts

Therefore, P_1 can finish its critical section without being interrupted by the scheduler.

- Will this solve the problem?
 - What if the critical section is code that runs for an hour?
 - Can some processes starve – never enter their critical section.
 - What if there are two CPUs?-> Doesnot work for multi-processor systems.

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

```
disable_interrupts() ;  
  
balance = balance - amount;  
  
enable_interrupts() ;
```

SOFTWARE (TURN) SOLUTION

Two process solution

- Assume that the **load** and **store** machine-language instructions are atomic (cannot be interrupted until completed)
- The two processes share one variable:
int turn;
- The variable **turn** indicates whose turn it is to enter the critical section; initially, the value of **turn** is set to **i**

```
//initially turn = i  
while (true){  
  
    while (turn == j); //wait  
  
    /* critical section */  
  
    turn = j;  
  
    /* remainder section */  
  
}
```

```
//code for P0  
while (turn != 0) ; // Wait  
    // Critical section  
    balance = balance - 5000;  
    // Give turn to P1  
    turn = 1;  
    // Remainder section
```

```
//code for P1  
while (turn != 1) ; // Wait  
    // Critical section  
    balance = balance - 2000;  
    // Give turn to P0  
    turn = 0;  
    // Remainder section
```

CORRECTNESS OF THE SOFTWARE SOLUTION

1. Mutual exclusion is preserved

P_i enters critical section only if:

turn = i

and **turn** cannot be both 0 and 1 at the same time. **Mutual Exclusion holds.**

2. What about the Progress requirement?

Consider this situation:

turn = 0.

P0 enters and finishes its critical section.

P0 sets turn = 1.

P1 is not interested in entering the critical section.

P0 wants to enter again.

What happens? -> **Progress is violated.**

3. What about the Bounded-waiting requirement?

P0 enters

↓

P0 exits → turn = 1

↓

P1 enters

↓

P1 exits → turn = 0

↓

P0 gets its turn

Bounded Waiting holds.

PETERSON'S ALGORITHM

- Use two shared variables instead of one!
- 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!

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

PETERSON'S ALGORITHM

```
//code for P0
while (true) {
    flag[0] = true;
    turn = 1;

    while (flag[1] && turn == 1)
    { // Wait }

    // Critical section
    balance = balance - 5000;

    flag[0] = false;
    // Remainder section
}
```

```
//code for P1
while (true) {
    flag[1] = true;
    turn = 0;

    while (flag[0] && turn == 0)
    { // Wait }

    // Critical section
    balance = balance - 3000;

    flag[1] = false;
    // Remainder section
}
```

Lets assume both processes P0 and P1 want to enter the critical section.

flag[0]=true; flag[1]=true

turn = 0 or turn = 1; The last process to write turn gives priority to the other process.

Lets say **turn=1** then

(flag[1] == true && turn==1) // P0 waits

P1 checks, and enters the critical section; when P1 finishes, it sets **flag[1]=false**

P0 waits only if BOTH conditions are true; P0's waiting condition becomes false, so P0 can enter

TURN VS PETERSONS SOLUTION

Feature	turn	Peterson's
Shared variables	turn	flag[] and turn
Mutual Exclusion	Yes	Yes
Progress	No	Yes
Bounded Waiting	Yes	Yes
Knows whether other wants to enter?	No	Yes, using flag[]

CS330

Operating Systems

**CRITICAL SECTION
PROBLEM: HARDWARE
SOLUTIONS**

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

HARDWARE SOLUTIONS

- Many systems provide hardware support for implementing the critical section code.
- **Uniprocessors** – could disable interrupts
 - Currently running code would execute without preemption
 - Generally too **inefficient** on **multiprocessor** systems
 - Operating systems using this *not broadly scalable*
- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or to *swap* the contents of two words atomically (uninterruptedly.)
 - **Test-and-Set** instruction
 - **Compare-and-Swap** instruction

THE TEST_AND_SET INSTRUCTION

- Definition

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

- Properties

- Executed **atomically** (non-interruptable)
- Returns the original value of passed parameter
- Set the new value of passed parameter to **true**

SOLUTION USING TEST_AND_SET()

- Shared Boolean variable **lock**, initialized to **false**
- Solution:

```
do{
    while (test_and_set(&lock))
        {;} /* do nothing */

    /* critical section */

    lock = false;
    /* remainder section */
} while (true);
```

THE COMPARE_AND_SWAP INSTRUCTION

- Definition

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

- Properties

- Executed atomically
- Returns the original value of passed parameter **value**
- Set the variable **value** the value of the passed parameter **new_value** but only if ***value == expected** is **true**.

SOLUTION USING COMPARE_AND_SWAP

- Shared integer **lock** initialized to 0;
- Solution:

```
while (true){  
    while (compare_and_swap(&lock, 0, 1) != 0)  
        {;} /* do nothing */  
  
    /* critical section */  
  
    lock = 0;  
    /* remainder section */  
}
```

CS330

Operating Systems

MUTEX LOCKS AND SEMAPHORES

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

MUTEX LOCKS

- Previous solutions are complicated and generally inaccessible to application programmers. OS designers build software tools to solve critical section problem
- Simplest is **mutex** lock
 - Boolean variable indicating if lock is available or not
- Protect a critical section by
 - First **acquire()** a lock
 - Then **release()** the lock
- Calls to **acquire()** and **release()** must be **atomic**
 - Usually implemented via hardware atomic instructions such as **compare-and-swap**.
- But this solution requires **busy waiting**
 - This lock therefore called a **spinlock**

```
while (true) {  
    acquire lock  
  
    critical section  
  
    release lock  
  
    remainder section  
}
```

SEMAPHORE

- A semaphore is an integer synchronization variable S accessed through atomic **wait()** and **signal()** operations to control concurrent access to shared resources.

- Definition of the **wait()** operation

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

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

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

- **Semaphore types:**

- **Binary semaphore** – integer value can range only between 0 and 1. Same as a **mutex lock**
- **Counting semaphore** – integer value can range over an unrestricted domain

SEMAPHORE USAGE EXAMPLE

- Solution to the CS Problem

- Create a semaphore “**mutex**” initialized to 1

- `wait(mutex) ;`

- `CS`

- `signal(mutex) ;`

- Consider P_1 and P_2 that with two statements S_1 and S_2 and the requirement that S_1 to happen before S_2

- Create a semaphore “**sync**” initialized to 0

- `P1 :`

- `S1 ;`

- `signal(sync) ;`

- `P2 :`

- `wait(sync) ;`

- `S2 ;`

SEMAPHORE IMPLEMENTATION

- 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
 - Could now have **busy waiting** in critical section implementation
 - But implementation code is short
 - Little busy waiting if critical section rarely occupied
- Note that applications may spend lots of time in critical sections and therefore this is **not a good solution**

SEMAPHORE IMP. WITH NO BUSY WAITING

- With each semaphore there is an associated waiting queue. Each entry in a waiting queue has two data items:
 - Value (of type integer)
 - Pointer to next record in the list
- Two operations:
 - **block** – place 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

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

IMPLEMENTATION WITH NO BUSY WAITING (CONT.)

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

Process requests a resource,
if it is available, the process acquires it.
else add to the waiting queue and block

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

Process finished using the resource,
Remove it from the queue and wakeup

EXAMPLE

- Assume P1, P2 and P3 are processes with Semaphore S=1. Initially Q is empty

S=1	C.S[]
Q=Empty	

P1.wait(S)

S=0	C.S[P1]
Q=Empty	

P2.wait(S)

S=-1	C.S[P1]
Q=P2	

P3.wait(S)

S=-2	C.S[P1]
Q=P2, P3	

P1.signal(S)

S=-1	C.S[P2]
Q=P3	

S=-1 indicates, one process waiting!
P2 blocks and sleeps

S=-2 indicates, two processes waiting!
P3 blocks and sleeps

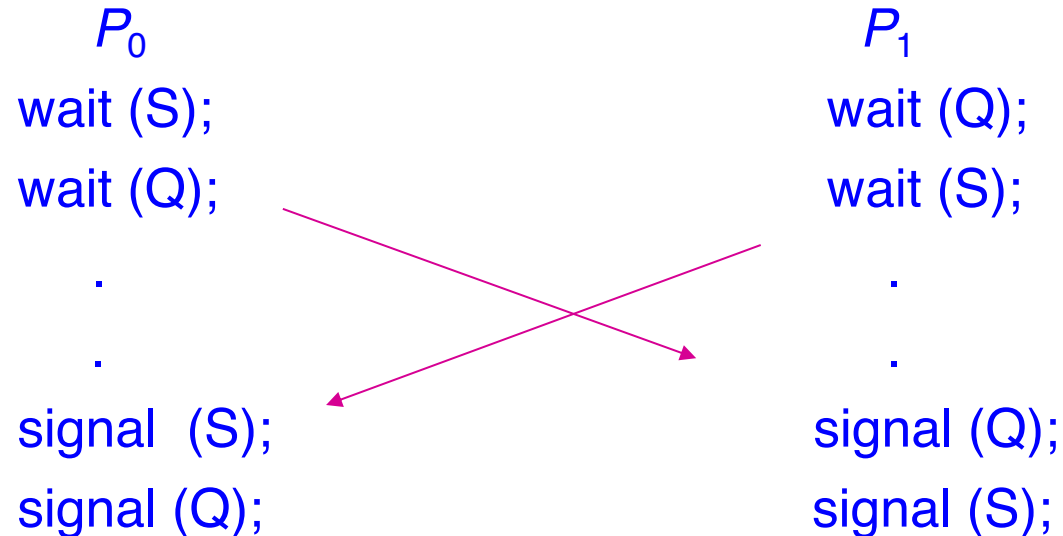
P2 wakes and enters Critical section
P3 is still blocked and sleeping

PROBLEMS WITH SEMAPHORES

- Incorrect use of semaphore operations:
 - `signal(mutex) ... wait(mutex)`
 - `wait(mutex) ... wait(mutex)`
 - Omitting of `wait(mutex)` and/or `signal(mutex)`
- These – and others – are examples of what can occur when semaphores and other synchronization tools are used incorrectly.

DEADLOCK AND STARVATION

- **Deadlock** – two or more processes are waiting indefinitely for an event that can be caused by only one of the waiting processes
- Let S and Q be two semaphores initialized to 1



- **Starvation** – *indefinite blocking*. A process may never be removed from the semaphore queue in which it is suspended.
- When P_0 executes Wait(Q), it must wait until P_1 executes Signal (Q) and when P_1 executes Wait(S)), it must wait until P_0 executes Signal(S) .
- Solution: Always acquire semaphores in the same order. E.g. wait(S); wait(Q); signal(Q), signal(S)

CS330

Operating Systems

CLASSICAL EXAMPLES OF SEMAPHORES

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

CLASSICAL EXAMPLES OF SEMAPHORES

- Bounded buffer problem
- Readers and Writers problem
- Dining Philosophers problem

BOUNDED BUFFER PROBLEM

- The **bounded-buffer problem** is the classic **Producer-Consumer** problem in Operating Systems. It demonstrates how **semaphores** can synchronize processes or threads that share a fixed-size buffer.
- Imagine a buffer that can store only 5 items.
 - A producer adds items to the buffer.
 - A consumer removes items from the buffer.
 - The buffer has a limited capacity.
- Problem:
 - The producer must not add an item when the buffer is full.
 - The consumer must not remove an item when the buffer is empty.
 - Producer and consumer must not modify the shared buffer simultaneously.

BOUNDED BUFFER PROBLEM

- We use 3 synchronization variables

```
semaphore mutex = 1; //protects C.S
```

```
semaphore empty = 5; //counts empty buffer slots
```

```
semaphore full = 0; //counts full buffer slots
```

```
//Producer
```

```
do {  
    // Produce an item  
    wait(empty);  
    wait(mutex);  
  
    // Add item to buffer  
    signal(mutex);  
    signal(full);  
} while (true);
```

Step 1: **wait(empty)**

The producer checks empty slots.

- If empty > 0, it decreases the count and continues.
- If empty == 0, the buffer is full and the producer blocks.

Step 2: **wait(mutex)**

The producer acquires exclusive access to the buffer.

Step 3: Add item

The producer inserts the item into the buffer.

Step 4: **signal(mutex)**

The producer releases the critical section.

Step 5: **signal(full)**

The producer indicates that one more item is available.

BOUNDED BUFFER PROBLEM

```
//Consumer
```

```
do {  
    wait(full);  
    wait(mutex);  
  
    // remove item from buffer  
    signal(mutex);  
    signal(empty);  
  
    //consume the item  
} while (true);
```

Step 1: **wait(full)**

The consumer checks if item is available.

- If $full > 0$, it decreases the count and continues.
- If $full == 0$, the buffer is full and the consumer blocks.

Step 2: **wait(mutex)**

The consumer acquires exclusive access to the buffer.

Step 3: remove item

The consumer removes the item.

Step 4: **signal(mutex)**

The consumer releases the critical section.

Step 5: **signal(empty)**

The consumer indicates that one more item is available.

Semaphores solve the bounded-buffer problem by using **empty** to prevent overflow, **full** to prevent underflow, and **mutex** to ensure mutual exclusion when accessing the shared buffer.

READERS WRITERS PROBLEM

- Multiple processes have access to shared data (file, database etc)
- Two types of processes
 - Readers-> only read the shared data
 - Writers-> only modify shared data
- Rules:
 - READ: allows multiple processes to read concurrently,
 - WRITE: only ONE process is allowed to write at a time.
 - A WRITER cannot write if a reader is reading
 - No reader can read when WRITER is writing

READERS WRITERS PROBLEM

- Semaphore solution:

```
semaphore rw_mutex = 1; // controls access to shared data
semaphore mutex = 1; //protects read_count variable
int read_count = 0; //Number of readers currently reading
```

READERS WRITERS PROBLEM

```
do{
    wait(mutex) ;
    read_count++;

    if (read_count == 1)
        wait(rw_mutex) ;

    signal(mutex) ;

    // READ DATA

    wait(mutex) ;
    read_count--;

    if (read_count == 0)
        signal(rw_mutex) ;

    signal(mutex) ;
}while(true) ;
```

Suppose nobody is reading:

`read_count = 0`
`rw_mutex = 1`

First reader:

`read_count = 1`
`wait(rw_mutex)` becomes 0

Reader can read but writers are prevented from writing

For each next reader

`read_count++`

Each reader can read

If a writer arrives `rw_mutex=0`

Writer is **blocked**

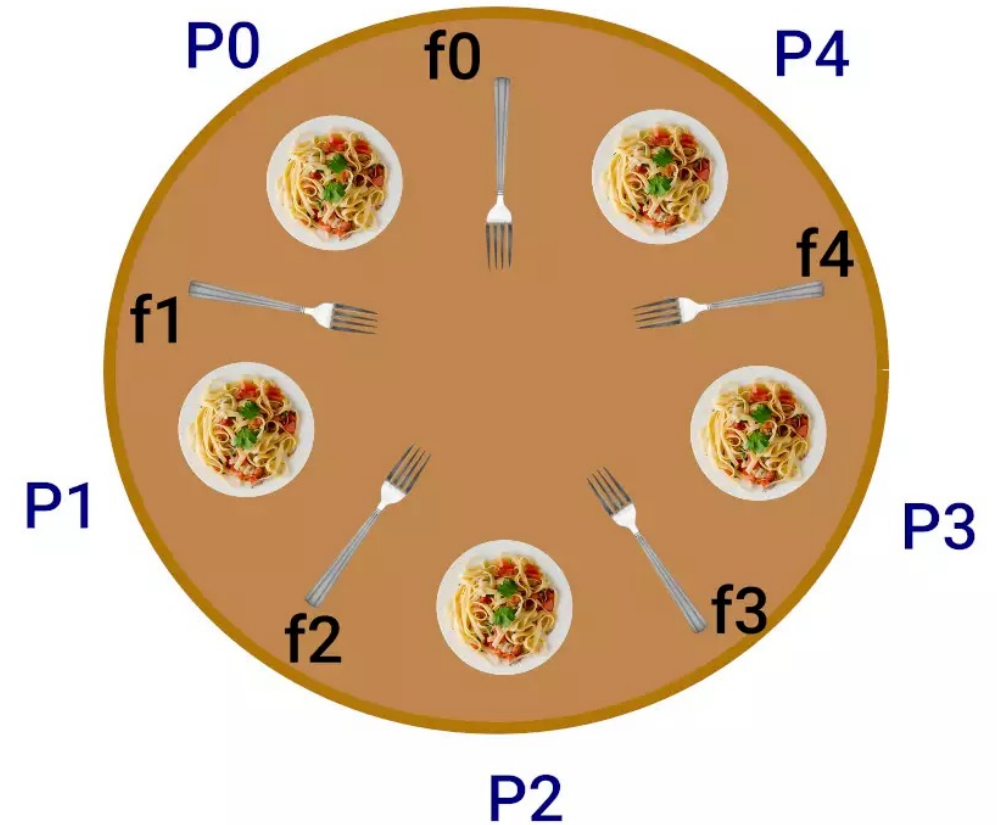
The last reader calls `signal(rw_mutex)`
`rw_mutex` becomes 1, writer wakes up

DINING PHILOSOPHERS PROBLEM

5 philosophers who only **eat** and **think**
each need to use 2 **forks** for eating

Problem: we have only 5 forks

- A classical synchronization problem
- Shows the difficulty of allocating resources among processes without deadlock and starvation

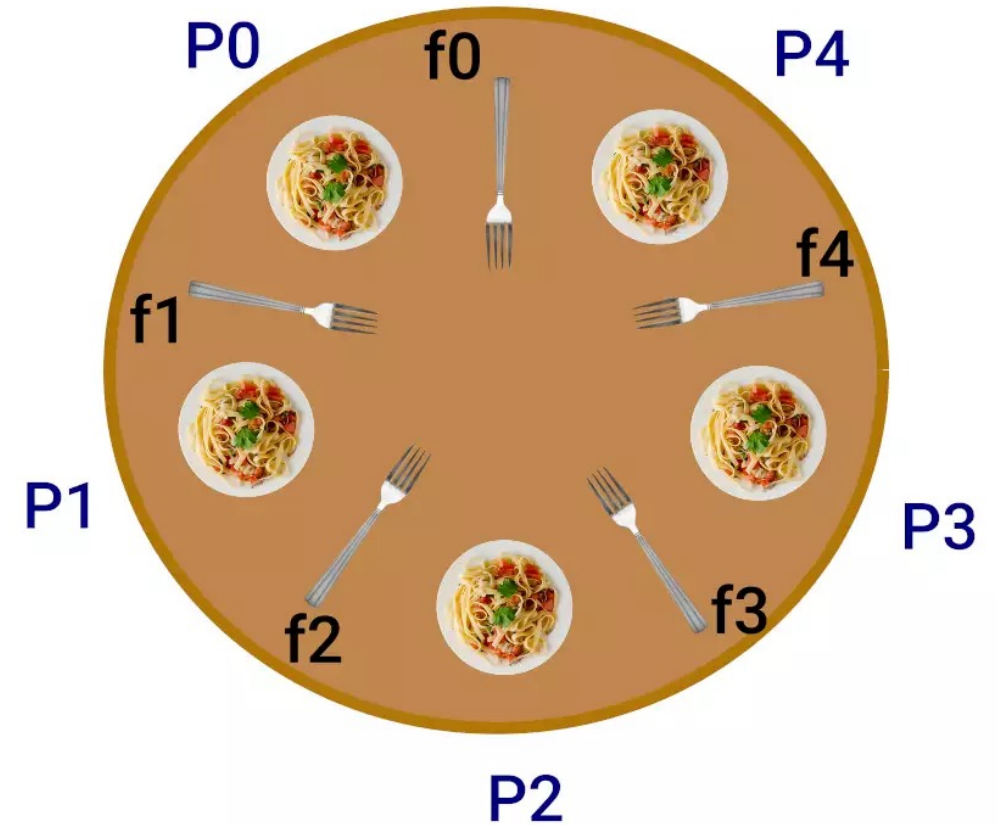


DINING PHILOSOPHERS PROBLEM

Each philosopher (Process) repeatedly:

1. Thinks
2. Becomes hungry
3. Picks up two forks
4. Eats
5. Puts the forks down

A fork is a shared resource between philosophers. The problem is that each philosopher needs **two forks**, but each fork can be used by only one philosopher at a time.



DINING PHILOSOPHERS PROBLEM

Suppose P0 becomes hungry. It needs f0 and f1.

P0 needs f0 and f1

P1 needs f1 and f2

P2 needs f2 and f3

P3 needs f3 and f4

P4 needs f4 and f0

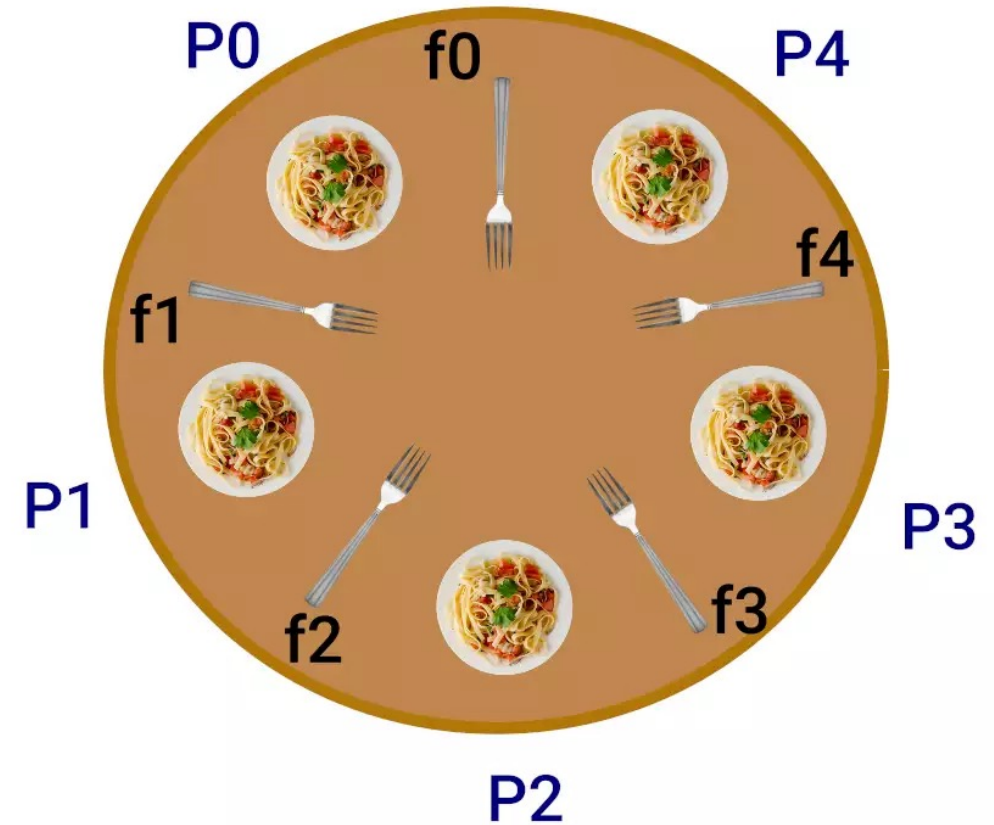
Notice that F0 is shared between P0 and P1.

Two neighboring philosophers cannot eat simultaneously.

Using semaphores:

```
semaphore fork[5] = {1, 1, 1, 1, 1};
```

1-> fork is available, 0-> Not available



DINING PHILOSOPHERS PROBLEM

A philosopher might run this:

```
do{  
    wait(fork[i]);           // Pick up left fork  
    wait(fork[(i+1)%5]);    // Pick up right fork  
    // EAT  
    signal(fork[i]);  
    signal(fork[(i+1)%5]);  
}while(true)
```

P0 takes f0 (**f0** = 0)

P1 takes f1 (**f1** = 0)

P2 takes f2 (**f2** = 0)

P3 takes f3 (**f3** = 0)

P4 takes f4 (**f4** = 0)

Everyone is holding one fork (left).
But everyone needs a second fork.

P0 waiting for f1
P1 waiting for f2
P2 waiting for f3
P3 waiting for f4
P4 waiting for f0

Nobody can proceed! This is **deadlock**.

PROBLEMS WITH SEMAPHORES

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes
- But `wait(S)` and `signal(S)` are scattered among several processes. Hence, difficult to understand their effects. **Usage must be correct in all the processes**
- Incorrect use of semaphore operations:
 - `signal(mutex) .... wait(mutex)`
 - `wait(mutex) ... wait(mutex)`
 - Omitting of `wait(mutex)` or `signal(mutex)` (or both)
- Causes Deadlock and starvation

CS330

Operating Systems

MONITORS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

MONITORS

- A **monitor** is a **high-level synchronization mechanism** used to control access to shared data.
- A monitor is a protected object that allows only one thread/process to execute inside it at a time. It combines **shared data + procedures + synchronization** into one unit.
- Semaphores require programmer to remember and correctly use **wait()** and **signal()** everywhere.
- A **monitor makes this easier** by providing mutual exclusion automatically.

MONITORS

A monitor is a software module containing:

- one or more procedures
- an initialization sequence
- local data variables

Characteristics:

- local variables accessible only by monitor's procedures
- a process enters the monitor by invoking one of its procedures
- only **one process** can be in **the monitor** at any given time

```
monitor monitor-name
{
    // shared variable declarations
    procedure P1 (...) { ... }
    ...
    procedure Pn (...) {.....}
    Initialization code ( ...) { ... }
    ...
}
}
```

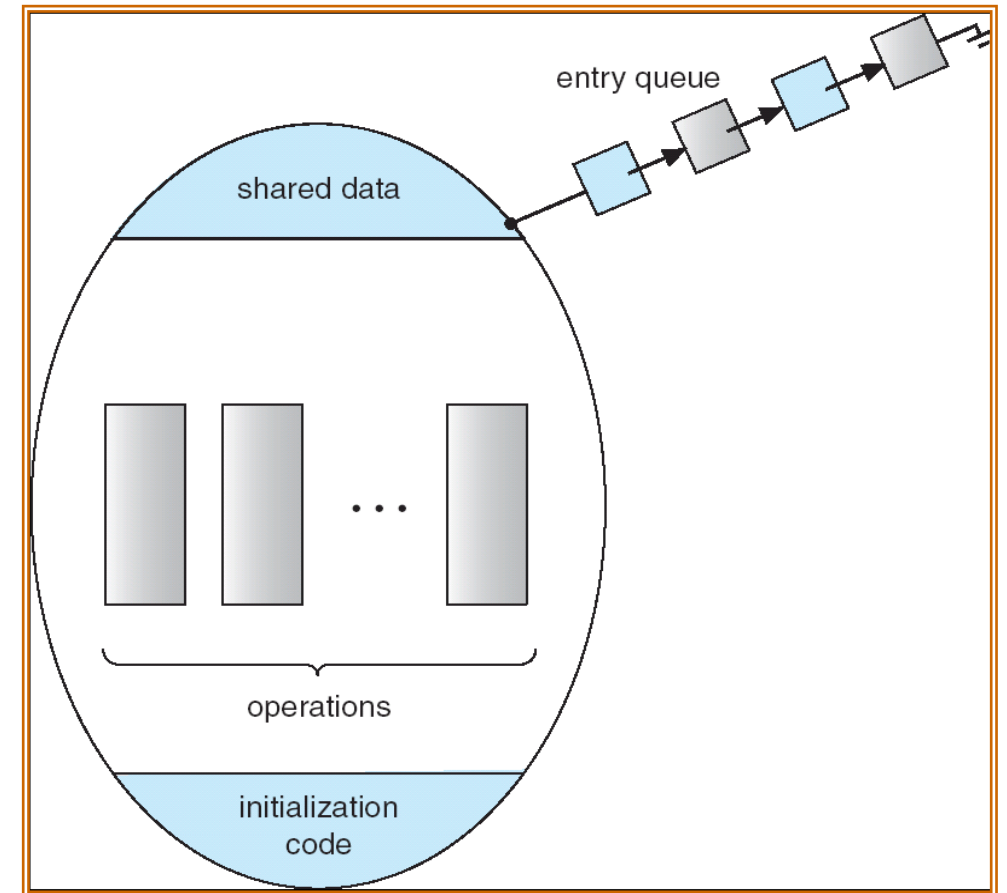
MONITORS

A monitor is a software module containing:

- one or more procedures
- an initialization sequence
- local data variables

Characteristics:

- local variables accessible only by monitor's procedures
- a process enters the monitor by invoking one of its procedures
- only **one process** can be in **the monitor** at any given time



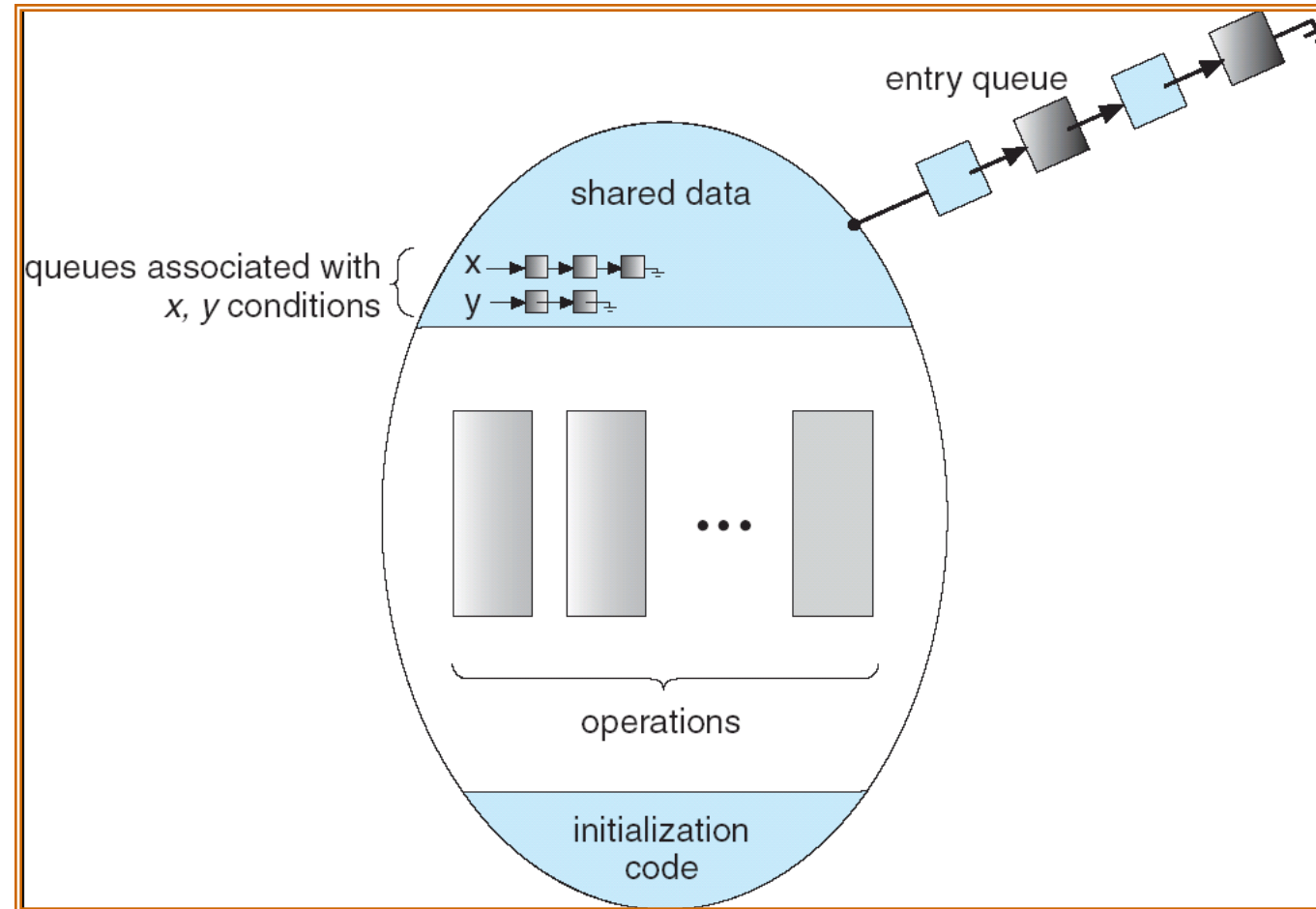
CONDITION VARIABLES

- The monitor construct is not sufficiently powerful for modeling some synchronization schemes-→ need to define additional mechanisms.
- These mechanisms are provided by the **condition construct**.
- A programmer who needs to write a tailor-made synchronization scheme can define one or more variable of type **condition**:

condition x, y;

- Two operations on a condition variable:
 - **x.wait()** – a process that invokes the operation is suspended.
 - **x.signal()** – resumes one suspended processes that invoked **x.wait()**
 - If no **x.wait()** on the variable, then it has no effect on the variable

MONITOR WITH CONDITION VARIABLES



CONDITION VARIABLES CHOICES

- If process P invokes `x.signal ()`, with Q in `x.wait ()` state, what should happen next? If Q is resumed, then P (signaling process) must wait
 - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
 - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition

MONITORS EXAMPLE

- Lets revisit the scenario presented earlier and define a monitor **MS** for deduction of rent and loan from salary. Assume the company has a condition, deduct loan, rent only if the salary has been credited.
- Only one thread can execute inside the Salary Monitor at a time.
- Lets assume there are 3 threads, Rent and Salary.
- Lets assume **deductRent(5000)** is called first. It enters the monitor MS

```
Rent Thread
  ↓
salaryCredited?
  ↓ no
salaryCredited.wait()
  ↓
Rent threads is blocked
```

```
monitor MS
{
  // shared variables
  salary = 10000

  // procedures
  deductRent()
  getSalary()
  creditSalary()

  //condition variables
  salaryCredited
}
```

After the deduction is completed, the rent thread leaves MS

MONITORS EXAMPLE

- Next the Salary thread is activated. It enters the monitor MS.
- It calls creditSalary(); After the salary is credited, salaryCredited.signal() is called. This wakes the Rent Thread

Salary Thread
↓
creditSalary(10000)
↓
salaryCredited.**signal**()
↓ Rent thread wakes

Rent threads was **blocked**

↓
deductRent(5000)
↓

SALARY MONITOR

salary = 30,000

salary -= 5,000

salary = 25,000

↓ **wait**()

After the deduction is completed, the rent thread leaves MS

```
monitor MS
{
    // shared variables
    salary = 10000

    // procedures
    deductRent()
    deductLoan()
    getSalary()

    //condition variables
    salaryCredited
}
```

MONITORS: DINING PHILOSOPHERS SOLUTION

- Instead of allowing philosophers to directly manipulate forks, we put the entire dining system inside a **monitor**.
- The monitor keeps track of the state of every philosopher: **THINKING HUNGRY EATING**
- We also create one condition variable for every philosopher: **condition self[5];**
- Only one philosopher can execute inside the monitor at a time.

DINING MONITOR

```
state[0]
state[1]
state[2]
state[3]
state[4]

pickup(i)
putdown(i)
test(i)

self[0] ... self[4]
```

MONITORS: DINING PHILOSOPHERS SOLUTION

```
monitor DiningPhilosophers{
    enum { THINKING, HUNGRY, EATING } state[5];
    condition self[5];

    void pickup(int i)    {
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i)   {
        state[i] = THINKING;

        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();
        }
    }
}
```

Only one philosopher can execute:

`pickup()` `putdown()` `test()`

inside the monitor at a time.

If a philosopher cannot eat, it sleeps.

`self[i].wait();`

`self[i].signal();`

Monitor → automatic mutual exclusion +
condition variables → synchronization
without explicit semaphore management.

SUMMARY

- **Race Conditions & Critical Section Problem:** A race condition occurs when concurrent processes access shared data without synchronization, leading to inconsistent results. Resolving this requires protecting the critical section.
- **Software & Hardware Solutions:** Software algorithms (such as Peterson's algorithm) manage process access using shared flags and turn variables, while hardware mechanisms rely on atomic instructions like Test-and-Set and Compare-and-Swap or disabling interrupts on single processors.
- **Mutex Locks & Semaphores:** Operating systems provide abstractions like binary mutex locks and integer semaphores (wait() and signal() operations) to manage resource access. Semaphores can be implemented without busy waiting by placing waiting processes into a queue to block and wake up as needed.
- **Classical Synchronization Problems:** Standard concurrency challenges include the Bounded-Buffer (Producer-Consumer), Readers-Writers, and Dining Philosophers problems, which highlight real-world synchronization needs along with risks like deadlock and starvation.
- **Monitors & Condition Variables:** Monitors provide a higher-level synchronization mechanism that encapsulates shared data and guarantees automatic mutual exclusion. They utilize condition variables with wait() and signal() operations to manage custom signaling and thread blocking safely.