

# Operating Systems

## CS2006

Lecture 10

### **Process Synchronization**

13th March 2023

Dr. Rana Asif Rehman

# Process Synchronization

- Processes may be **independent** or **cooperating**
- Cooperating process can affect or be affected by other processes
- Cooperating process can share data
  - Inter-Process communication in case of heavy-weight processes
  - Same logical address space in case of threads
  - Message passing

# Why synchronization?

- When we have multiple processes running at the same time (i.e. **concurrently**) we know that we must protect them from one another
  - E.g. protect one process' memory from access by another process
- But, what if we want to have those processes/ threads cooperate to solve a single problem?
  - E.g. one thread that manages, the mouse and keyboard, another that manages the display and a third that runs your programs
  - In this case, the processes/threads must be **synchronized**

# The Synchronization Problem

- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes

# The Synchronization Problem

- One process/thread should not get into the way of another process/thread when doing critical activities
- Proper sequence of execution should be followed when dependencies are present
  - A produces data, B prints it
  - Before printing B should wait while A is producing data
  - B is dependent on A

# If there is no synchronization



# Example (Producer-Consumer)

- Let's revisit an earlier example...
- The producer-consumer problem with a bounded buffer i.e. `BUFFER SIZE - 1` in the original solution.
- Suppose we want to modify the algorithm to remedy this deficiency!
- Add an integer variable `counter` initialized to 0.
- Increment/decrement `counter` every time we add/remove a new item to/from the buffer.

# Producer (Modified code)

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



# Consumer (Modified code)

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

# Synchronization Issue

- Producer and consumer routines shown above are correct separately, they may not function correctly when executed concurrently.
- Suppose that the value of the variable counter is **currently 5** and that the producer and consumer processes concurrently execute the statements “counter++” and “counter--”
- What might be the possible values of counter.... and which one is correct?

# Synchronization Issue (2)

We can show that the value of counter may be incorrect as follows.

□ **counter++** could be implemented in machine language as

```
register1 = counter  
register1 = register1 + 1  
counter = register1
```

□ **counter--** could be implemented as

```
register2 = counter  
register2 = register2 - 1  
counter = register2
```

□ where *register1,2* are the local CPU registers.

# Synchronization Issue (3)

- The concurrent execution of “counter++” and “counter--” is equivalent to a sequential execution.
- Consider this execution interleaving with “counter = 5” initially:

T0: producer execute	register1 = counter	{register1 = 5}
T1: producer execute	register1 = register1 + 1	{register1 = 6}
T2: consumer execute	register2 = counter	{register2 = 5}
T3: consumer execute	register2 = register2 - 1	{register2 = 4}
T4: producer execute	counter = register1	{counter = 6}
T5: consumer execute	counter = register2	{counter = 4}

- Notice that we have arrived at the incorrect state “counter == 4”, indicating that four buffers are full, when, in fact, five buffers are full.
- If we reversed the order of the statements at T4 and T5, we would arrive at the incorrect state “counter == 6”.

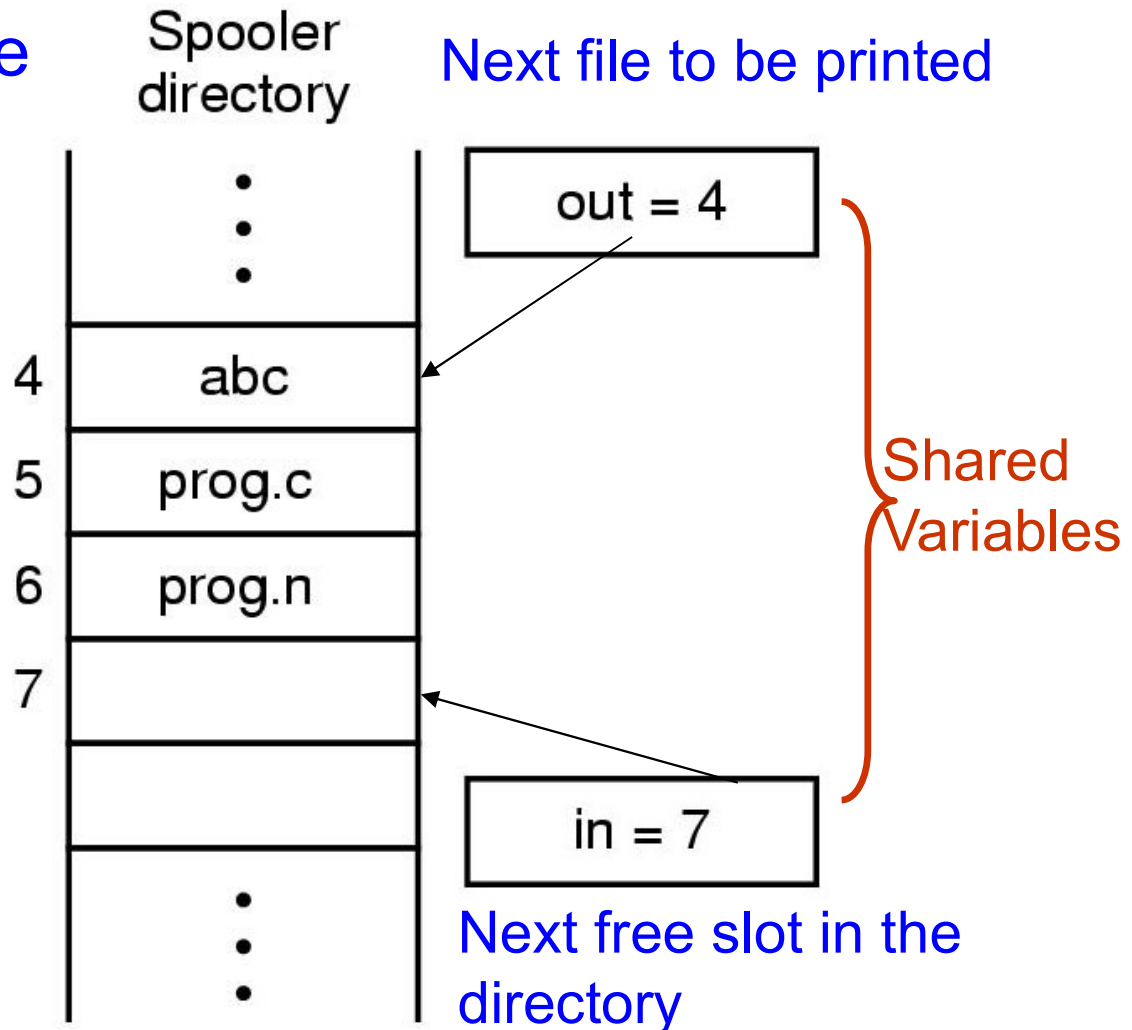
# Example Print Spooler

- If a process wishes to print a file it adds its name in a **Spooler Directory**
- The **Printer process**
  - Periodically checks the spooler directory
  - Prints a file
  - Removes its name from the directory

# Example Print Spooler

If any process wants to print a file it will execute the following code

1. Read the value of `in` in a local variable `next_free_slot`
2. Store the name of its file in the `next_free_slot`
3. Increment `next_free_slot`
4. Store back in `in`



# Example Print Spooler

Let A and B be processes with files to print

Process A

Process B

1. Read the value of `in` in a local variable `next_free_slot`

2. Store the name of its file in the `next_free_slot`

3. Increment `next_free_slot`

4. Store back in `in`

1. Read the value of `in` in a local variable `next_free_slot`

2. Store the name of its file in the `next_free_slot`

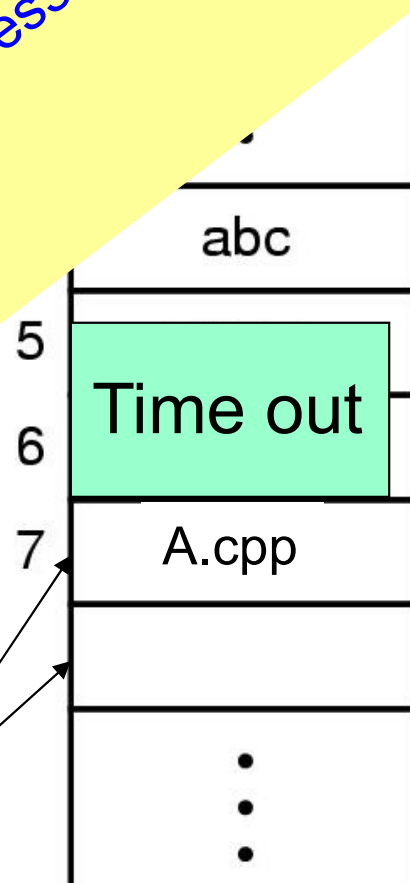
3. Increment `next_free_slot`

4. Store back in `in`

Process B will never receive any output

`next_free_slota = 7`

`in = 8`



`next_free_slotb = 7`

# Race condition

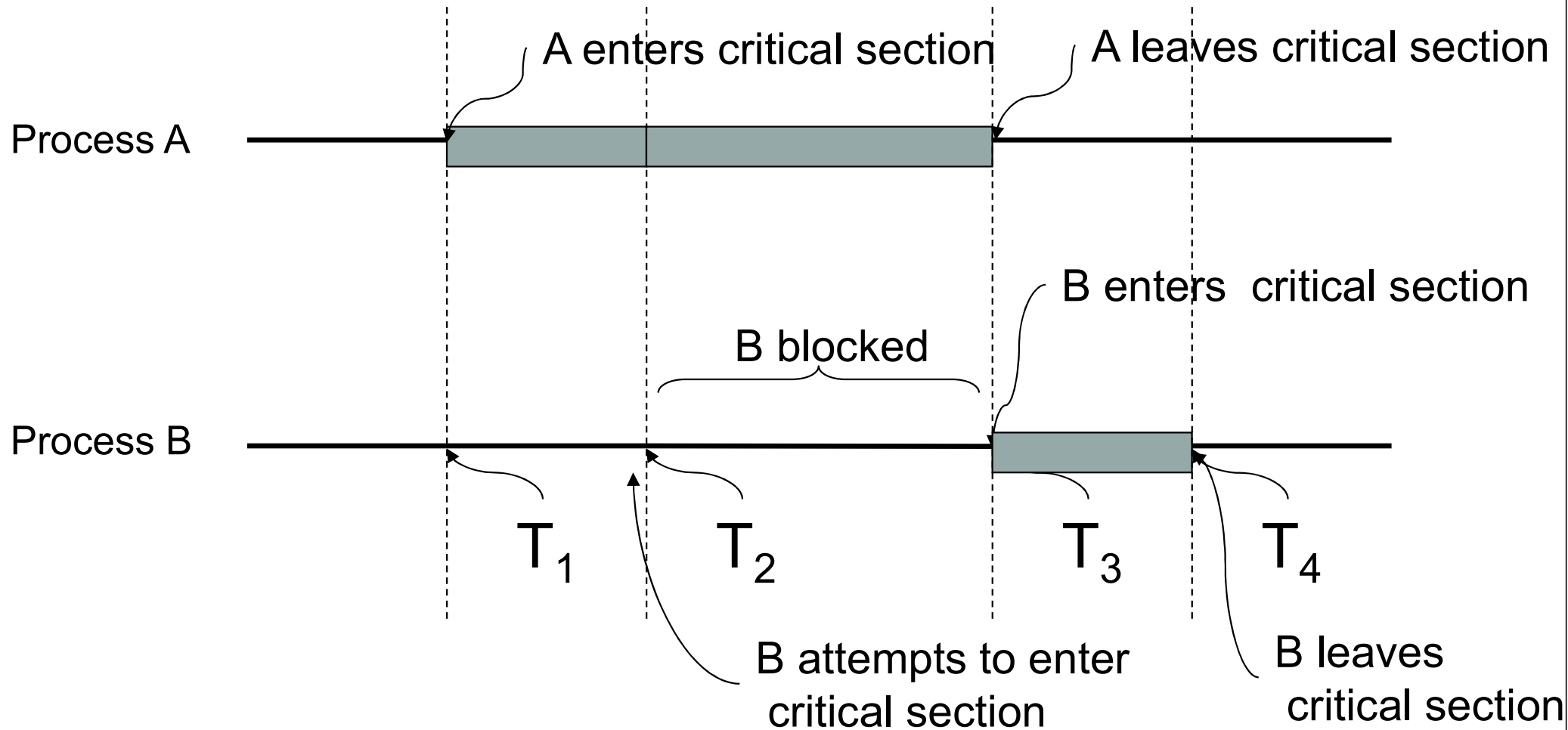
- A situation where several processes access and manipulate the same data concurrently, and the outcome of the execution depends on the particular order in which the access takes place, is called **race condition**.
  - Debugging is not easy
  - Most test runs will run fine
  - To prevent race conditions, concurrent processes must be **synchronized**



# Reason behind Race Condition

- Process B started using one of the shared variables before process A was finished with it.
- At any given time a Process is either
  - Doing internal computation  $\Rightarrow$  no race conditions
  - Or accessing shared data that can lead to race conditions
- Part of the program where the shared memory is accessed is called **Critical Region**
- Races can be avoided
  - **If no two processes are in the critical region at the same time.**

# Critical Section



## Mutual Exclusion

At any given time, only one process is in the critical section

# Critical Section

- Avoid race conditions by not allowing two processes to be in their critical sections at the same time
- We need a mechanism of mutual exclusion
- Some way of ensuring that one process, while using the shared variable, does not allow another process to access that variable

# The Critical-Section Problem

- Each process has a code segment, called *Critical Section (CS)*, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its CS, no other process is allowed to execute in its CS.

# The Critical-Section Problem

- Only 2 processes, P0 and P1
- General structure of process P<sub>i</sub> (other process P<sub>j</sub>)

**do {**

*entry section*

*critical section (CS)*

*exit section*

*reminder section*

**} while (1);**

- Processes may share some common variables to synchronize their actions

# Solution to Critical-Section Problem

- There are 3 requirements that must stand for a correct solution:
  1. **Mutual Exclusion**
  2. **Progress**
  3. **Bounded Waiting**



# Solution to CS Problem – Mutual Exclusion

1. **Mutual Exclusion** – If process  $P_i$  is executing in its critical section, then no other processes can be executing in their critical sections.
- Implications:
  - Critical sections better be focused and short.
  - Better not get into an infinite loop in there.

## Solution to CS Problem – Progress

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 only one process wants to enter, it should be able to.
  - If two or more want to enter, one of them should succeed.
  - No deadlock
  - No process in its remainder section can participate in this decision



## Solution to CS Problem – Bounded Waiting

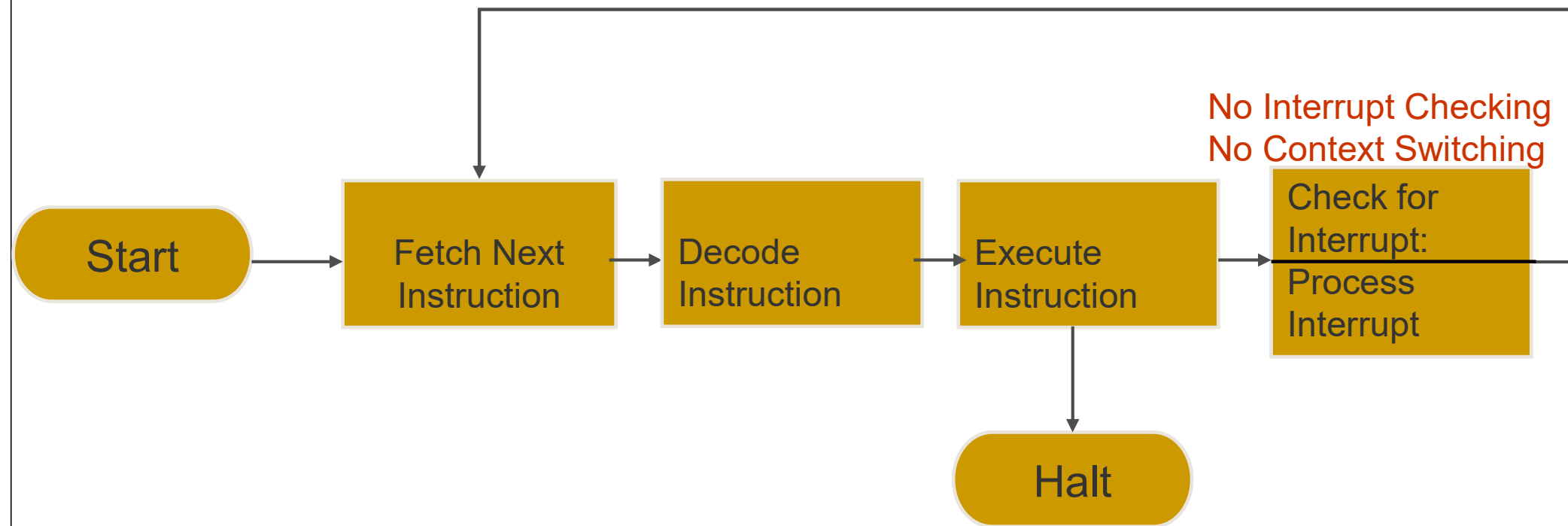
3. **Bounded Waiting** – A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.
- Assume that each process executes at a nonzero speed.
  - No assumption concerning relative speed of the  $n$  processes.
  - Deterministic algorithm, otherwise the process could suffer from starvation

# Implementing Mutual Exclusion

1. Disabling Interrupts
2. Lock Variables
3. Strict Alternation

# Disabling Interrupts

- The problem occurred because the CPU switched to another process due to clock interrupt
- Remember the CPU cycle



# Disabling Interrupts

- Solution: A Process
  - **Disable interrupts** before it enters its critical section
  - **Enable interrupts** after it leaves its critical section
- CPU will be unable to switch a process while it is in its **critical section**
- Guarantees that the process can use the shared variable without another process accessing it
- **Disadvantage:**
  - Unwise to give user processes this much power
  - The computer will not be able to service useful interrupts
  - The process may never enable interrupts, thus (effectively) crashing the system
- However, the kernel itself can disable the interrupts

# Lock Variables: Software Solution

- Before entering a critical section a process should know if any other is already in the critical section or not
- Consider having a FLAG (also called lock)
- **FLAG = FALSE**
  - A process is in the critical section
- **FLAG = TRUE**
  - No process is in the critical section

```
// wait while someone else is in the
// critical region
1. while (FLAG == FALSE) ;
// stop others from entering critical region
2. FLAG = FALSE;
3. critical_section() ;
// after critical section let others enter
//the critical region
4. FLAG = TRUE;
5. noncritical_section() ;
```

FLAG = FALSE

## Lock Variables

### Process 1

```
1.while (FLAG == FALSE);  
2.FLAG = FALSE;  
3.critical_section();  
4.FLAG = TRUE;  
5.noncritical_section();
```

### Process 2

```
1.while (FLAG == FALSE);  
2.FLAG = FALSE;  
3.critical_section();
```

Timeout

No two processes may be simultaneously inside  
their critical sections

Process 2 's Program counter is at Line 2

Process 1 forgot that it was Process 2's turn

# Solution: Strict Alternation

- We need to remember “Who’s turn it is?”
- If its Process 1’s turn then Process 2 should wait
- If its Process 2’s turn then Process 1 should wait

## Process 1

```
while (TRUE)
{
    // wait for turn
    while (turn != 1);
    critical_section();
    turn = 2;
    noncritical_section();
}
```

## Process 2

```
while (TRUE)
{
    // wait for turn
    while (turn != 2);
    critical_section();
    turn = 1;
    noncritical_section();
}
```

# Strict Alternation

Turn = 1

## Process 1

```
While(1)
1.while (Turn != 1);
2.critical_section();
3.Turn = 2;
4.noncritical_section();
```

## Process 2

```
While(1)
1.while (Turn != 2);
2.critical_section();
3.Turn = 1;
4.noncritical_section();
```

Timeout

Only one Process is in the Critical Section at a time

Process 2 's Program counter is at Line 2

Process 1 Busy Waits

CS-2006 Operating Systems



# Strict Alternation

## Process 1

```
while (TRUE)
{
    // wait
    while (turn != 1);
    critical_section();
    turn = 2;
    noncritical_section();
}
```

## Process 2

```
while (TRUE)
{
    // wait
    while (turn != 2);
    critical_section();
    turn = 1;
    noncritical_section();
}
```

- Can you see a problem with this?
- Hint : What if one process is a much faster than the other

# Strict Alternation

turn = 1

## Process 1

```
while (TRUE)
{
    // wait
    while (turn != 1);
    critical_section();
    turn = 2;
    → noncritical_section();
}
```

## Process 2

```
while (TRUE)
{
    // wait
    while (turn != 2);
    critical_section();
    turn = 1;
    noncritical_section();
}
```



### Process 1

- Runs
- Enters its critical section
- Exits; setting **turn** to 2.

Process 1 is now in its **non-critical section**.

Assume this non-critical procedure takes a long time.

Process 2, which is a much faster process, now runs

Once it has left its critical section, sets **turn** to 1.

Process 2 executes its **non-critical section** very **quickly** and returns to the top of the procedure.

turn = 1

### Process 1

```
while (TRUE)
{
    // wait
    while (turn != 1);
    critical_section();
    turn = 2;
    → noncritical_section();
}
```

### Process 2

```
while (TRUE)
{
    // wait
    → while (turn != 2);
    critical_section();
    turn = 1;
    noncritical_section();
}
```

- Process 1 is in its non-critical section
- Process 2 is waiting for turn to be set to 2
- In fact, there is no reason why process 2 cannot enter its critical region as process 1 is not in its critical region.

## Strict Alternation

- What we have is a violation of one of the conditions that we listed above

No process running outside its critical section may block other processes

- This algorithm requires that the processes *strictly alternate* in entering the critical section
- Taking turns is not a good idea if one of the processes is *slower*.

## Reason

- Although it was Process 1's **turn**
- But Process 1 was not **interested**.
- Solution:
  - We also need to remember
    - **“Whether it is interested or not?”**

## Algorithm 2

- Replace
  - **int turn;**
- With
  - **bool Interested[2];**
- **Interested[0] = FALSE**
  - Process 0 is not interested
- **Interested[0] = TRUE**
  - Process 0 is interested
- **Interested[1] = FALSE**
  - Process 1 is not interested
- **Interested[1] = TRUE**
  - Process 1 is interested

# Algorithm 2

## Process 0

```
while (TRUE)
{
    interested[0] = TRUE;
    // wait for turn
    while(interested[1] != FALSE) ;
    critical_section();
    interested[0] = FALSE;
    noncritical_section();
}
```

## Process 1

```
while (TRUE)
{
    interested[1] = TRUE;
    // wait for turn
    while(interested[0] != FALSE) ;
    critical_section();
    interested[1] = FALSE;
    noncritical_section();
}
```

## Algorithm 2

### Process 0

```
while (TRUE)
```

```
{
```

```
    interested[0] = TRUE;
```

```
    while (interested[1] !=
```

### Process 1

```
while (TRUE)
```

```
{
```

```
    interested[1] = TRUE;
```

```
    while (interested[0] != FALSE) ;
```

Timeout

DEADLOCK



# Peterson's Solution

Combine the previous two algorithms:

```
int turn;
```

```
bool interested[2];
```

- **Interested[0] = FALSE**
  - Process 0 is not interested
- **Interested[0] = TRUE**
  - Process 0 is interested
- **Interested[1] = FALSE**
  - Process 1 is not interested
- **Interested[1] = TRUE**
  - Process 1 is interested

# Algorithm 3: Peterson's Solution

- Two process solution (Software based solution)
- The two processes share two variables:
  - `int turn;`
  - `Boolean interested[2]`
- The variable `turn` indicates whose turn it is to enter the critical section.
- The `interested` array is used to indicate if a process is ready to enter the critical section. `interested[i] = true` implies that process `Pi` is ready

# Algorithm 3: Peterson's Solution

- Process  $P_i$

```
while (TRUE)
{
    interested[i] = TRUE;
    turn = j;
    // wait
    while (interested[j] == TRUE && turn == j );
    critical_section();
    interested[i] = FALSE;
    noncritical_section();
}
```

# Algorithm 3: Peterson's Solution

- Meets all three requirements:
  - **Mutual Exclusion:** 'turn' can have one value at a given time (0 or 1)
  - **Bounded-waiting:** At most one entry by a process and then the second process enters into its CS
  - **Progress:** Exiting process sets its 'flag' to false ... comes back quickly and set it to true again ... but sets turn to the number of the other process

# Two-Process Solution to the Critical-Section Problem -- Peterson's Solution

```
flag[0], flag[1] := false
```

```
turn := 0;
```

Process P0:

```
repeat
```

```
    flag[0] := true;
```

```
    // 0 wants in
```

```
    turn := 1;
```

```
    // 0 gives a chance to 1
```

```
    while(flag[1] && turn == 1) {};
```

```
        CS
```

```
    flag[0] := false;
```

```
    // 0 is done
```

```
        RS
```

```
forever
```

Process P1:

```
repeat
```

```
    flag[1] := true;
```

```
    // 1 wants in
```

```
    turn := 0;
```

```
    // 1 gives a chance to 0
```

```
    while(flag[0] && turn == 0) {};
```

```
        CS
```

```
    flag[1] := false;
```

```
    // 1 is done
```

```
        RS
```

```
forever
```

- ➡ The algorithm proved to be correct. Turn can only be 0 or 1 even if both flags are set to true

# Issue with Peterson solution

- Peterson solution isn't practical because it can not solve critical section problem for more than two processes at the same time
- It is only a reference solution that helps understand the background for developing a new solution

# N-Process Critical Section Problem

- In this section we extend the critical section problem of two processes to include  $n$  processes. Consider a system of  $n$  processes ( $P_0, P_1, \dots, P_{n-1}$ ).
- Each process has a segment of code called a critical section in which the process may be changing common variables, updating a table, writing a file and so on. The important feature of the system is that, when one process enters its critical section, no other process is allowed to execute in its critical section.
- Thus the execution of critical sections by the processes is mutually exclusive in time. The critical section problem is to design a protocol to serialize executions of critical sections. Each process must request permission to enter its critical section.
- Many solutions are available in the literature to solve the  $N$ -process critical section problem. We will discuss a simple and elegant solution, known as the **Bakery algorithm**.



BAKERY

THE BAKERY ALGORITHM



# THE BAKERY ALGORITHM

- The bakery algorithm is due to **Leslie Lamport** and is based on a scheduling algorithm commonly used in bakeries, ice-cream stores, and other locations where order must be made out of chaos.
- On entering the store, each customer receives a number.
- The customer with the lowest number is served next.
- Before entering its critical section, process receives a ticket number.
- Holder of the smallest ticket number enters its critical section.
- Unfortunately, the bakery algorithm cannot guarantee that two processes (customers) will not receive the same number.

# THE BAKERY ALGORITHM

- In the case of a tie, the process with the lowest ID is served first.
- 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 ticket numbering scheme always generates numbers in the increasing order of enumeration; i.e., 1, 2, 3, 4, 5 ...
- Since process names are unique and totally ordered, our algorithm is completely deterministic.

# THE BAKERY ALGORITHM

- The common data structures are:
  - boolean choosing [n];
  - int number[n];
- Initially these data structures are initialized to false and 0, respectively.
- The following notation is defined for convenience:
  - (ticket #, process id #)
  - $(a,b) < (c,d)$  if  $a < c$  or if  $a == c$  and  $b < d$
  - $\max(a_0, \dots, a_{n-1})$  is a number,  $k$ , such that  $k \geq a_i$  for  $i=0, \dots, n-1$

# THE BAKERY ALGORITHM

The structure of process  $P_i$  used in the bakery algorithm is as follows:

```
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)));
    }

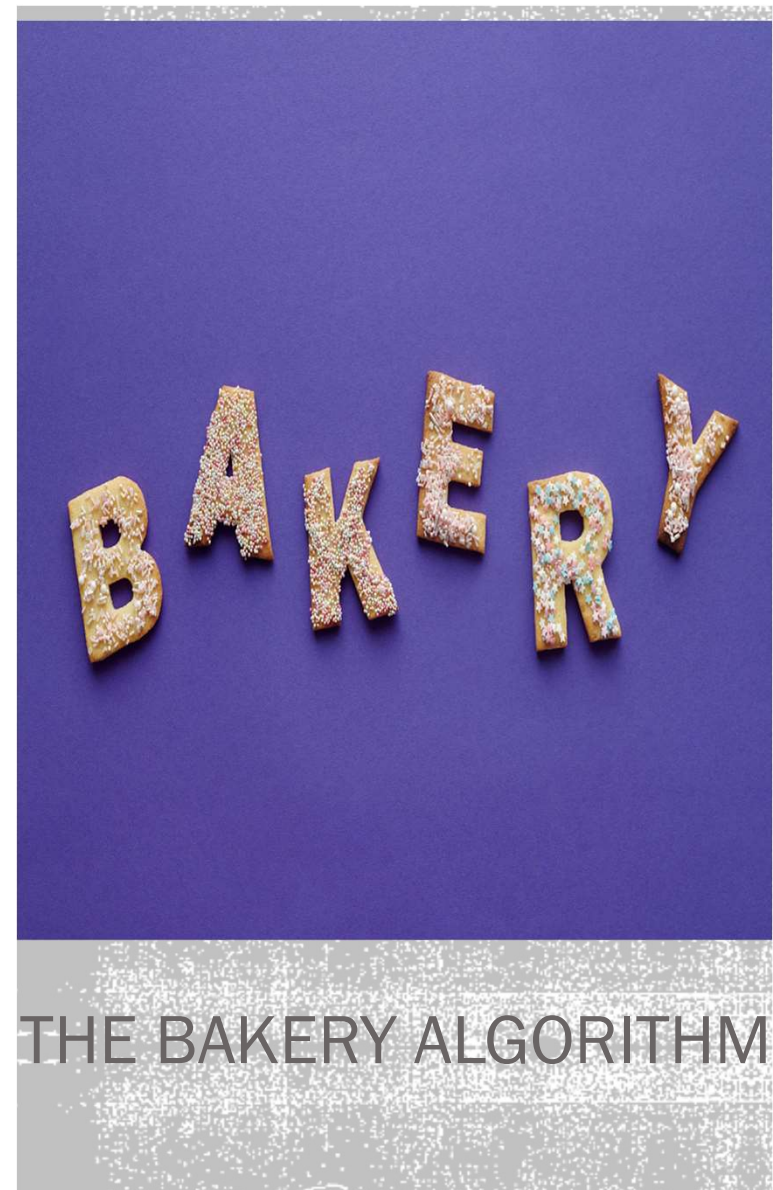
    Critical section

    number[i] = 0;

    Remainder section

} while(1);
```

Process	Number
P0	3
P1	0
P2	7
P3	4
P4	8



- The following table shows the status of all the processes as they execute the 'for' loops in their entry sections.
- The gray cells show processes waiting in the second while loops in their entry sections.
- The table shows that P0 never waits for any process and is, therefore, the first process to enter its critical section,
- while all other processes wait in their second while loops for  $j = 0$ , indicating that they are waiting for P0 to get out of its critical section and then they would make progress (i.e., they will get out the while loop, increment  $j$  by one, and continue their execution).



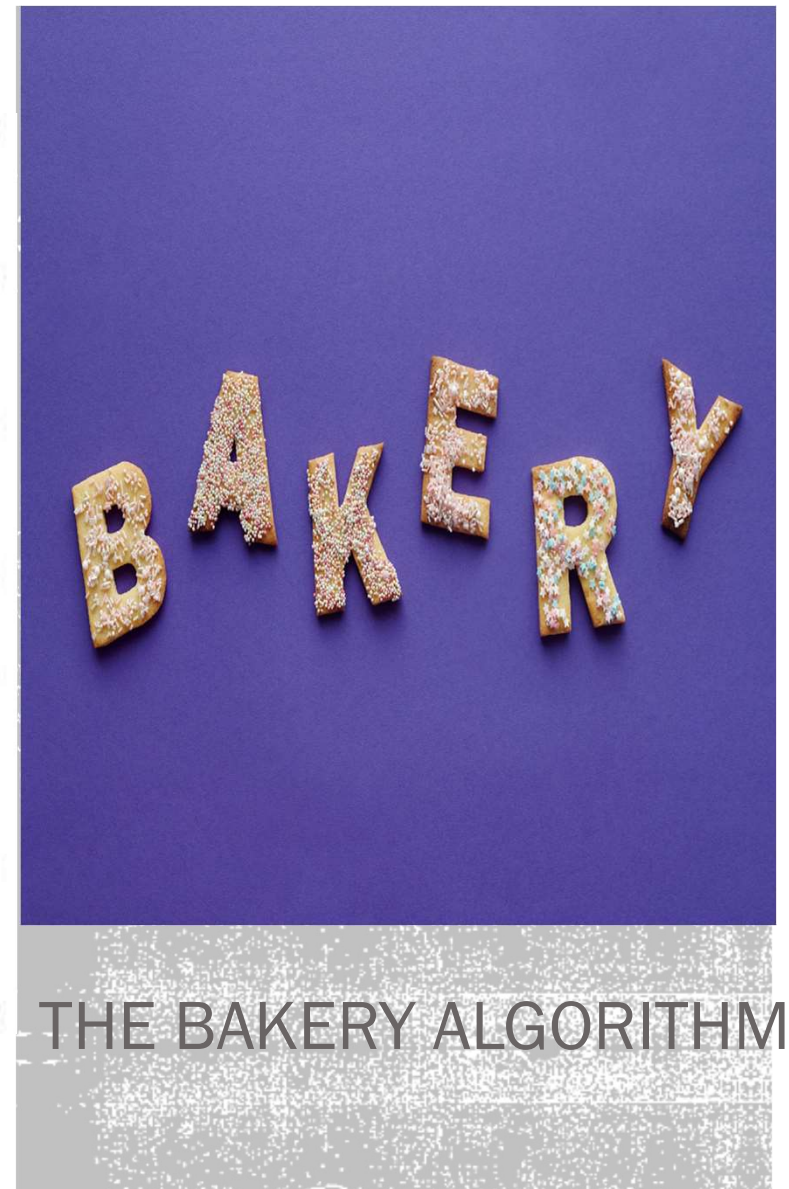


- You can make the following observations by following the Bakery algorithm closely with the help of the following table:
- P1 not interested to get into its critical section  $\Rightarrow$  number[1] is 0
- P2, P3, and P4 wait for P0
- P0 gets into its CS, get out, and sets its number to 0
- P3 get into its CS and P2 and P4 wait for it to get out of its CS
- P2 gets into its CS and P4 waits for it to get out
- P4 gets into its CS
- Sequence of execution of processes:  **$\langle P_0, P_3, P_2, P_4 \rangle$**



THE BAKERY ALGORITHM

j	P0	P2	P3	P4
0	$(3,0) < (3,0)$	$(3,0) < (7,2)$	$(3,0) < (4,3)$	$(3,0) < (8,4)$
1	Number[1] = 0	Number[1] = 0	Number[1] = 0	Number[1] = 0
2	$(7,2) < (3,0)$	$(7,2) < (7,2)$	$(7,2) < (4,3)$	$(7,2) < (8,4)$
3	$(4,3) < (3,0)$	$(4,3) < (7,2)$	$(4,3) < (4,3)$	$(4,3) < (8,4)$
4	$(8,4) < (3,0)$	$(8,4) < (7,2)$	$(8,4) < (4,3)$	$(8,4) < (8,4)$





# References

- Operating System Concepts (Silberschatz, 8<sup>th</sup> edition)  
Chapter 6