

# Operating Systems

---

## **5. Synchronization**

# Synchronization

---

- Many processes/threads interact (shared memory / message passing)
  - Results of interactions not guaranteed to be deterministic
  - Concurrent writes to the same address
  - Result depends on the order of operations
- OS manage
  - Large number of resources
  - Common data structures
- Need to provide good coordination mechanisms
  - Access to the data structures
  - Ensure consistency
- Objective
  - Identify general **synchronization problems**
  - Provide **OS support for synchronization**

## Example: Too Much Milk

---

Alice

3:00 Look in fridge - no milk

3:05 Leave for shop

3:10 Arrive at shop

3:15 Leave shop

3:20 Back home - put milk in fridge

3:25

Bob

Look in fridge - no milk

Leave for shop

Arrive at shop

Leave shop

Back home - put milk in fridge

Oooops!

Problem: Need to ensure that only one process is doing something at a time (e.g., getting milk)

## Example: Money Flies Away ...

---

BALANCE: 2000 €

Bank thread A

Read  $a := \text{BALANCE}$

$a := a + 500$

Write  $\text{BALANCE} := a$

Bank thread B

Read  $b := \text{BALANCE}$

$b := b - 200$

Write  $\text{BALANCE} := b$

Ooops!! BALANCE: 1800 €!!!!

Problem: need to ensure that each process is executing its critical section (e.g updating BALANCE) exclusively (one at a time)

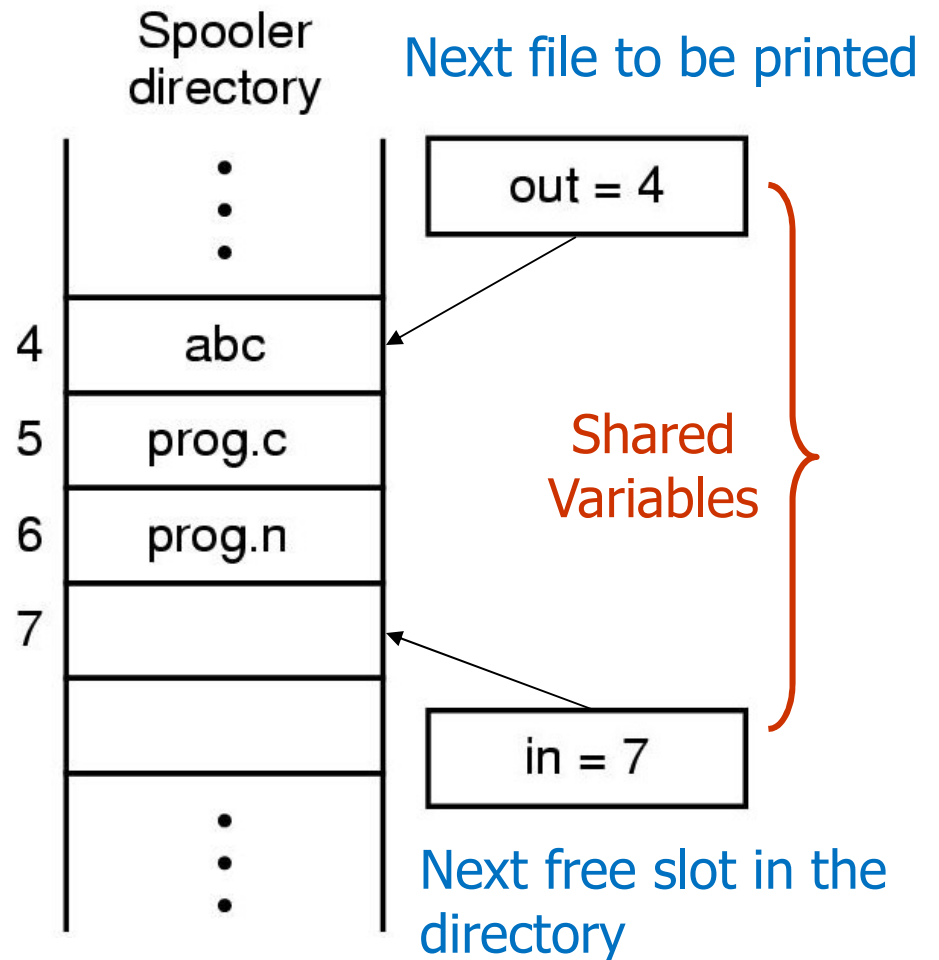
## Example: Print Spooler

---

- 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 who want to print their files

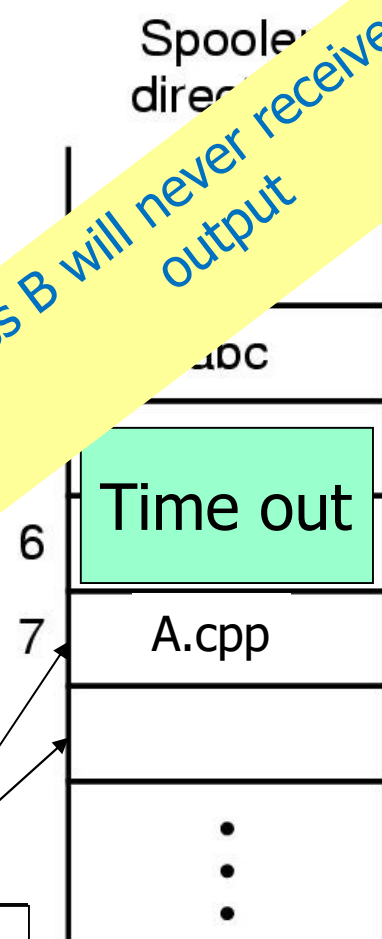
## Process A

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

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`

## Example: Producer/Consumer Problem

---

- An integer count that keeps track of the number of full buffers
- Initially, count is set to 0
  - Count is incremented by the producer after it produces a new buffer
  - Count is decremented by the consumer after it consumes a buffer

### **Producer**

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



## Example: Producer/Consumer Problem

---

- An integer count that keeps track of the number of full buffers
- Initially, count is set to 0
  - Count is incremented by the producer after it produces a new buffer
  - Count is decremented by the consumer after it consumes a buffer

### Consumer:

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

# Producer/Consumer Problem – Race Condition

---

- `counter++` could be implemented as
  - `register1 = counter`
  - `register1 = register1 + 1`
  - `counter = register1`
- `counter--` could be implemented as
  - `register2 = counter`
  - `register2 = register2 - 1`
  - `count = register2`
- Consider this execution interleaving with “count = 5” initially:
  - S0: producer execute `register1 = counter` {`register1 = 5`}
  - S1: producer execute `register1 = register1 + 1` {`register1 = 6`}
  - S2: consumer execute `register2 = counter` {`register2 = 5`}
  - S3: consumer execute `register2 = register2 - 1` {`register2 = 4`}
  - S4: producer execute `counter = register1` {`count = 6`}
  - S5: consumer execute `counter = register2` {`count = 4`}

# Race Condition

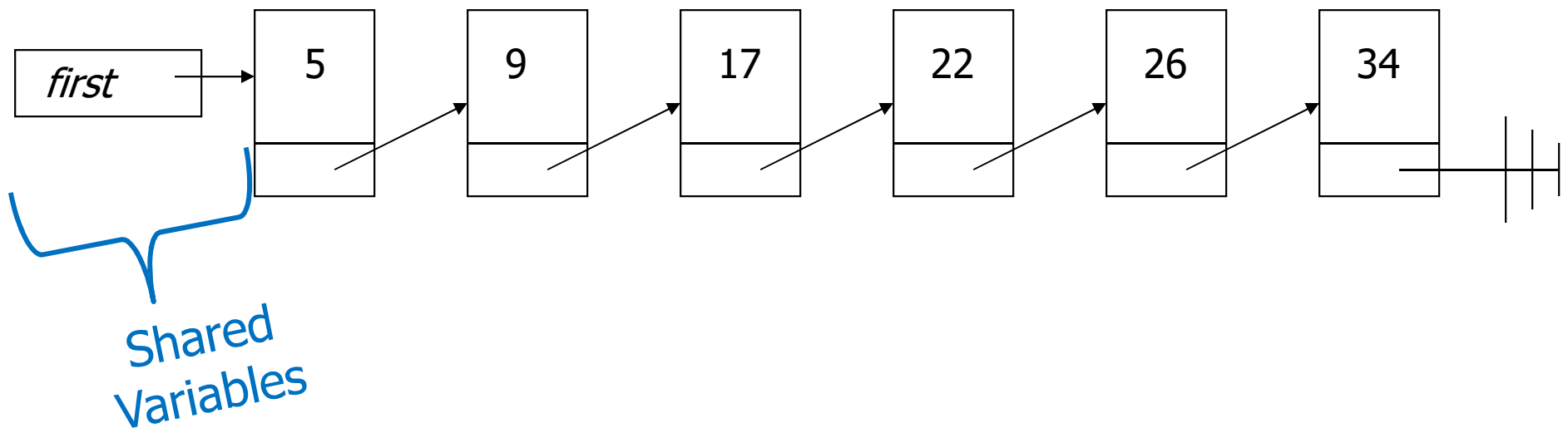
---

- Race condition
  - Several processes access and manipulate the same data concurrently
  - Outcome of the execution depends on the particular order in which the access takes place
    - Debugging is not easy → Most test runs will run fine
- At any given time a process is either
  - Doing internal computation → 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 condition can be avoided
  - If no two processes are in the critical region at the same time

# Examples of Race Condition

---

- Adding node to a shared linked list



# Critical-Section (CS) Problem

---

- $n$  processes all competing to use some shared data / resource
- Each process has a code segment, called **critical section**, in which the shared data is accessed
- **Problem:** ensure that
  - Never two process are allowed to execute in their critical section at the same time
  - Access to the critical section must be an atomic action
- Structure of process  $P_i$ :

```
repeat
    entry section
    critical section
    exit section
    remainder section
until false;
```

# Critical-Section (CS) – Example

---

```
void threadRoutine()
```

```
{
```

```
    int y, x, z;
```

```
    y = 3 + 5x;
```

```
    y = Global_Var;
```

```
    y = y + 1;
```

```
    Global_Var = y;
```

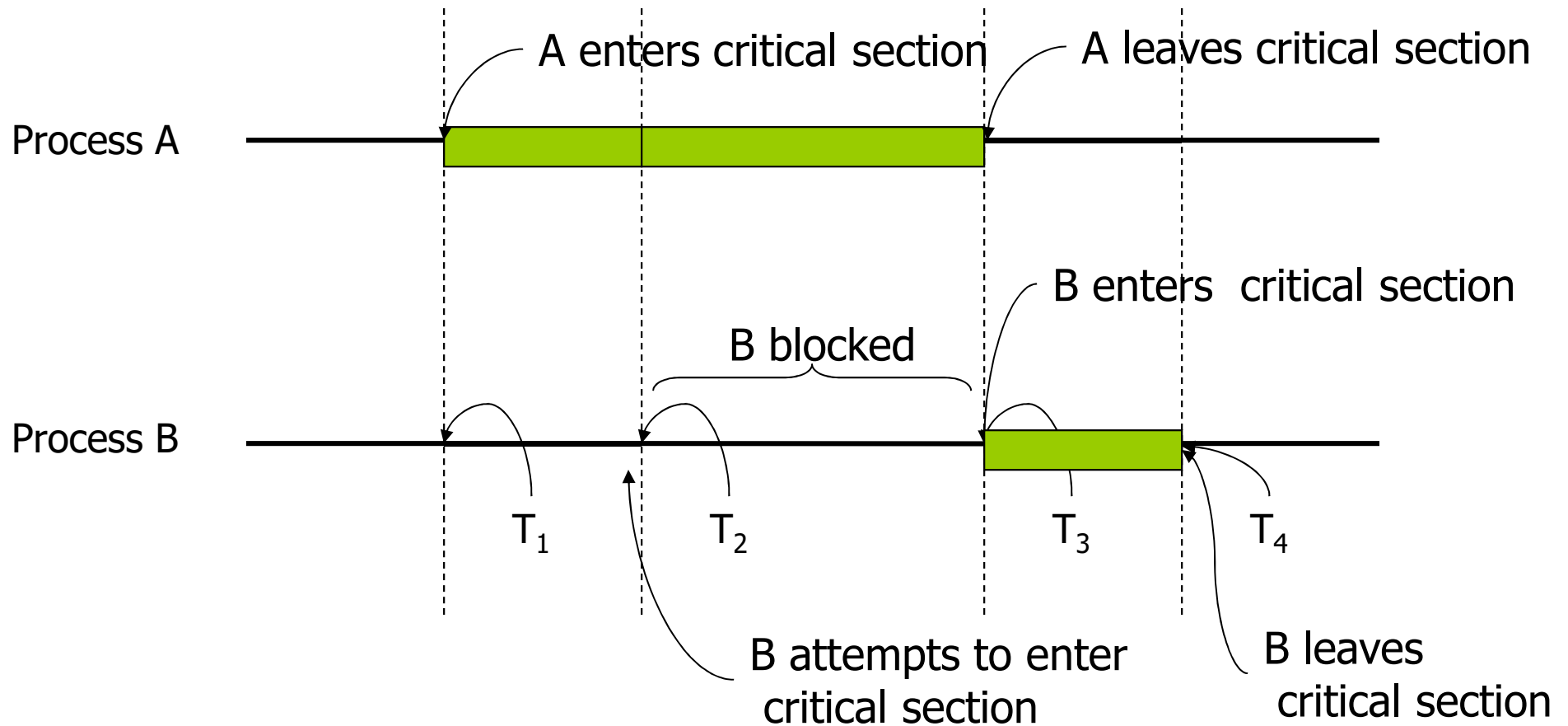


```
    ...
```

```
    ...
```

```
}
```

# Critical-Section (CS)



## Mutual Exclusion

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

# Requirements: Critical-Section (CS) Problem

---

Informally, the following are the requirements of CS problem

- No two processes may be simultaneously inside their critical sections
- No assumptions may be made about the speed or the number of processors
- No process running outside its critical section may block other processes
- No process should have to wait forever to enter its critical section



# Requirements: Critical-Section Problem

- **Mutual exclusion**
    - Only one process at a time is allowed to execute in its critical section
  - **Progress**
    - Processes cannot be prevented forever from entering CS
- 
- Progress: Distinguish between
    - **Fairness (no starvation):** Every process makes progress
      - **Bounded waiting**
    - **Deadlock freedom:** At least one process can make progress
      - Process running outside the critical section should not have influence
  - Assumptions
    - No process stays forever in CS
    - No assumption on the number of processors
    - No assumption on actual speed of processes

# Requirements: Critical-Section Problem

---

1. Mutual exclusion
2. Progress
3. Bounded waiting
  - Before request by a process to enter its critical section is granted
  - A bound must exist on the number of times that other processes are allowed to enter their critical sections

# Software Solution – Lock Variables

---

- 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();
```

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

FLAG = FALSE

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 turn

# Lock Variables

---

```
// 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();
```

- Algorithm does not satisfy critical section requirements
  - Progress is ok, but does NOT satisfy mutual exclusion

# Software 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

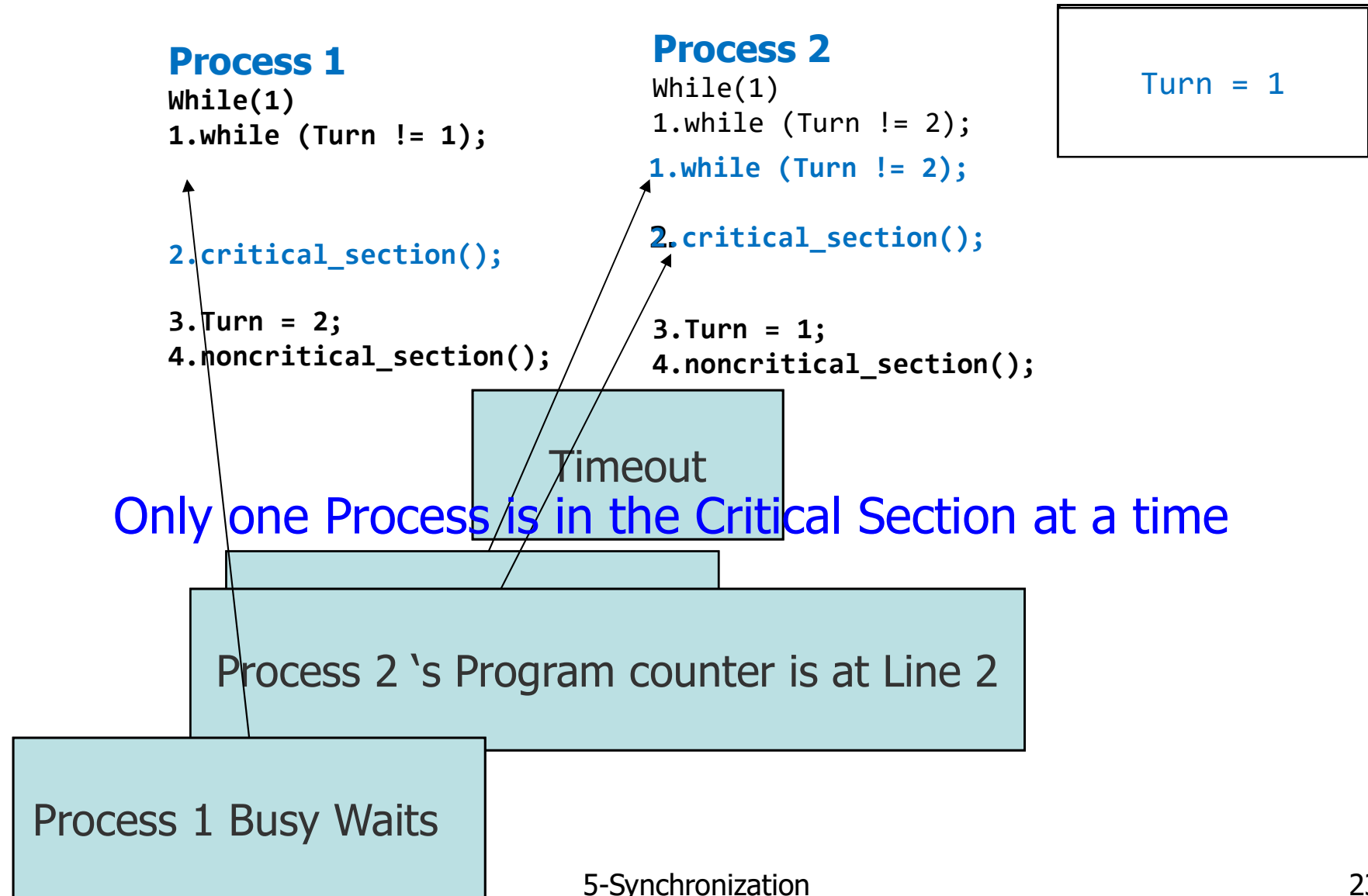
## Process 1

```
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



# Strict Alternation

---

## 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();
}
```

- What is the problem with strict alteration?
  - Satisfies mutual exclusion, but not progress



# Strict Alternation

Turn = 1

## 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();
}
```

- Process 1
  - Runs, Enters its critical section, Exits critical section, Sets 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

# Strict Alternation

Turn = 1

## 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();
}
```

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

# Strict Alternation

## 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();
}
```

## Problem with strict alteration

- **Violation:** 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**

# Yet Another Solution

---

- Strick alternation
  - Although it was Process 1's **turn**
  - But Process 1 was **not interested**
- Solution: **Remember whether a process is interested in CS or not**
  - Replace **int turn** with **bool interested[2]**
  - For example **interested[0] = FALSE** → Process 0 is not interested

## 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();
}
```

# Yet Another Solution

---

## Process 0

```
while(TRUE)
{
    interested[0] = TRUE;

    while(interested[1] != FALSE);
```

## Process 1

```
while(TRUE)
{
    interested[1] = TRUE;

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

Timeout

# DEADLOCK

# Peterson's Algorithm

---

- Combine previous two algorithms
  - `int turn` with `bool interested`

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

# Peterson's Algorithm

---

## Process 0

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

## Process 1

```
while(TRUE)
{
    interested[1] = TRUE;
    turn = 0;
    // wait
    while(interested[1] == 1 && turn == 0);

    critical_section();
    interested[1] = FALSE;
    noncritical_section();
}
```



Timeout

# Peterson's Algorithm

## Process 0

```
while(TRUE)
{
    interested[0] = TRUE;
    turn = 1;

    // wait
    while(interested[0] == TRUE && turn == 1);

    critical_section();
    interested[0] = FALSE;
    noncritical_section();
}
```

## Process 1

```
while(TRUE)
{
    interested[1] = TRUE;
    turn = 0;
    // wait
    while(interested[1] == TRUE && turn == 0);

    critical_section();
    interested[1] = FALSE;
    noncritical_section();
}
```

Timeout

Can not be **TRUE** at the same time.  
Thus used to break tie



# Lamport's Bakery Algorithm (For n processes)

---

## Idea

- Before entering its critical section, each process receives a number
- Holder of the smallest number enters the critical section
  - Fairness
  - No deadlock
- 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
- While processes are trying to enter CS
  - The numbering scheme generates numbers in increasing order of enumeration; i.e., 1,2,3,3,3,3,4,5

# Lamport's Bakery Algorithm

- Peterson's algorithm solves the critical-section problem for two processes
  - For multiple processes, bakery algorithm is used

```
Shared var choosing: array [0..n - 1] of boolean (init false);
        number: array [0..n - 1] of integer (init 0),

Process Pi
1: repeat
2:   choosing[i] := true;
3:   number[i] := max(number[0], number[1], ..., number [n - 1]) + 1;
4:   choosing[i] := false;
5:   for j := 0 to n - 1 do begin
6:     while choosing[j] do no-op;
7:     while number[j] ≠ 0 and (number[j], j) < (number[i], i) do
8:       no-op;
9:   end;
10:  critical section
11:  number[i] := 0;
12:  remainder section
13: until false;
```

## CS Problem: Software-based Solutions

---

- Peterson and Bakery algorithms solve critical section problem
- These algorithms are not used in practice
  - Mutual exclusion depends on the order of steps in the software based algorithms
  - However, modern Compilers often reorder instructions to enhance performance

# Mutual Exclusion: Hardware Support

---

- Interrupt Disabling

- A process runs until it invokes an operating-system service or until it is interrupted
- Disabling interrupts guarantees mutual exclusion

```
while(true){  
    /* disable interrupts */  
    critical section  
    /* enable interrupts */  
    remainder section  
}
```

- Disadvantage

- Processor is limited in its ability to interleave programs
- Multiprocessors: Disable interrupts on one processor will not guarantee mutual exclusion
- User programs cannot directly disabled interrupts

# Mutual Exclusion: Other Hardware Support

---

- Special Machine Instructions
  - Performed in a single instruction cycle: Reading and writing in one atomic step
- Not subject to interference from other instructions
  - Uniprocessor system: Executed without interrupt
  - Multiprocessor system: Executed with locked system bus

# Test and Set (TSL)

---

## Test and Set Instruction

```
1: boolean testset (int & i) {  
2:     if (i == 0) {  
3:         i = 1;  
4:         return false;  
5:     }  
6:     else return true;  
7: }
```

## Test and Set Instruction

```
1: boolean testset (Boolean *target) {  
2:     boolean rv = *target;  
3:     *target = true;  
4:     return rv;  
5: }
```

- Atomic processor instructions
- Idea: Only one process at a time succeeds to set lock=1

# Test and Set (TSL)

## Test and Set Instruction

```
1: boolean testset(int& i) {  
2:     if (i == 0) {  
3:         i = 1;  
4:         return false;  
5:     }  
6:     else return true;  
7: }
```

```
1: void P(int i){  
2:     while (true) {  
3:         while (testset(bolt));  
4:         critical section  
5:         bolt = 0;  
6:         remainder section  
7:     }  
8: }
```

```
9: void main()  
10: {  
11:     bolt =0;  
13:     parbegin(P(1), ..., P(N));  
14: }
```

- Atomic processor instructions
- Idea: Only one process at a time succeeds to set lock=1

# Exchange (Swap)

## Exchange Instruction

```
1: void exchange(int& mem1,int& mem2)
2: {
3:     int temp = mem1;
4:     mem1 = mem2;
5:     mem2 = temp;
6: }
```

```
1: void P(int i){
2:     while (true) {
3:         int key = 1;
4:         while(key) {
5:             exchange(bolt,key);
6:         }
7:         critical section
8:         exchange(bolt,key);
9:         remainder section
10:    }
11:}

12: void main(){
13:     bolt =0;
14:     parbegin(P(1), ..., P(N));
15: }
```

- Atomic processor instructions
- Idea: Only one process at a time will be able to set key = 0



# Mutual Exclusion Using Machine Instructions

---

## Advantages

- Applicable to any number of processes on single or multiple CPUs sharing main memory
- It is simple and therefore easy to verify

## Disadvantages

- **Busy-waiting** consumes processor time
- **Starvation** is possible
  - A process leaves a CS
  - More than one process is waiting
- Deadlock possible **if used in priority-based scheduling systems**: Ex. scenario
  - Low priority process has the critical region
  - Higher priority process needs it
  - **The higher priority process will obtain the CPU to wait for the critical region**

# Avoiding Starvation

---

- Bounded waiting with **Test and Set instruction**

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);
    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);
```

# Priority Inversion

---

- Consider three processes
  - Process L with low priority that requires a resource R
  - Process H with high priority that also requires a resource R
  - Process M with medium priority
- If H starts after L has acquired resource R
  - H has to wait to run until L relinquishes resource R
- Now when M starts
  - M is higher priority unblocked process, it will be scheduled before L
    - Since L has been preempted by M, L cannot relinquish R
  - M will run until it is finished
    - Then L will run - at least up to a point where it can relinquish R
    - Then H will run
- **Priority inversion:** Process with the medium priority ran before a process with high priority

# Priority Inheritance

---

- If high priority process has to wait for some resource shared with an executing low priority process
- Low priority process is temporarily assigned the priority of the highest waiting priority process
- Medium priority processes are prevented from pre-empting (originally) low priority process avoiding priority inversion

# Semaphores

---

- Higher-level synchronization construct
  - Designed by Edsger Dijkstra in 1960's



# Semaphores



- Synchronization variable accessed by
  - System calls of the operating system
  - Associated with a queue of **blocked processes**
- Accessible via atomic `wait()` and `signal()` operations
  - Originally called `P()` and `V()`

**wait():** Process blocks until a signal is received

**signal():** Another process can make progress: Process in the queue, next process calling wait()

- Different semaphore semantics possible
  - **Binary:** Only one process at a time makes progress
  - **General (counting):** n processes at a time make progress

# Binary Semaphore

---

- **Wait:** Process blocks if state of semaphore is 0
- **Signal:** Unblocks a process or sets the state of the semaphore to 1

```
wait () / P() {  
    while S <= 0 ; // no-op  
    S--;  
}
```

**Atomic**

An atomic operation that waits for semaphore to become positive then decrements it by 1

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

**Atomic**

An atomic operation that increments semaphore by 1

**How to avoid busy waiting?**

# Busy-waiting

---

- Problem with hardware and software based implementations
  - If one thread already has the lock, and another thread T wants to acquire the lock
  - Acquiring thread T waste CPU's time by executing the Idle loop

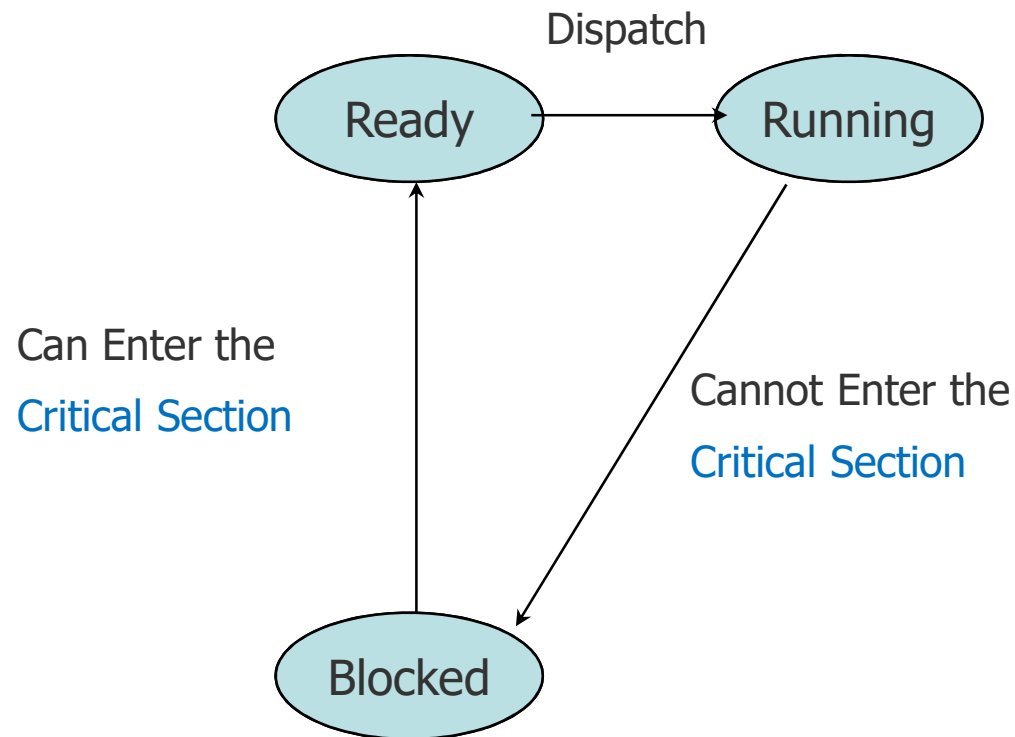
## To avoid CPU time

- If a thread (process) T cannot enter critical section
  - T's status should be changed from **Running** to **Blocked**
  - T should be removed from the **ready list** and entered in a **block list**
- When T enters critical section
  - T's status should be changed from **Blocked** to **Ready/Running**
  - T should be entered in the **ready list**



# Busy-waiting

---



No CPU time wastage

# Sleep/Block and Wakeup Implementation

---

- `sleep()/block()` and `wakeup()` **can** be two system calls
- When a process  $P_i$  determines that it can not enter the CS it calls `sleep()/block()`
  - $P_i$  entered in the block list
- When the other process finishes with the CS, it wakes up  $P_i$ , `wakeup( $P_i$ )`
  - $P_i$  entered in the ready list
- `wakeup()` takes a single input parameter, which is the ID of the process to be unblocked

# Binary Semaphore

- **Wait:** Process blocks if state of semaphore is 0
- **Signal:** Unblocks a process or sets the state of the semaphore to 1

```
struct bin_sem_t {
    enum {zero, one} value;
    queueType queue;
    /* Can be linked list of PCB */
};

int wait(bin_sem_t s) {
    if (s.value==1)
        s.value=0;
    else{
        /* block and place current
           process in s.queue */
        state(current) = BLOCKED
        append(s.queue, current)
    }
}
```

```
int signal(bin_sem_t s) {
    if (isempty(s.queue))
        s.value=1;
    else{
        /* unblock and remove first
           process from s */
        p=pop(s.queue);
        state(p) = READY/RUNNING
    }
}
```

# Counting Semaphore

- **Count:** Determines no. of processes that can still pass semaphore
- **Wait:** Decreases count and blocks if count is  $< 0$
- **Signal:** Unblocks a process or increases count

```
struct sem_t {
    int count;
    queueType queue;
};

int wait(sem_t s) {
    s.count--;
    if (s.count < 0)
        /* block and place current
           process in s.queue */
        state(current) = BLOCKED
        append(s.queue, current)
    }
}
```

```
int signal(sem_t s) {
    s.count++;
    if (s.count ≤ 0){
        /* unblock and remove
           first process from s */
        p=pop(s.queue);
        state(p) = RUNNING/READY
    }
}
```

**How to ensure atomicity?**

# Implementing Semaphores

---

- **Problem:** Concurrent calls of signal and wait
- **Solution:** Use hardware synchronization!
- **Properties:** Still limited busy waiting

```
int wait(sem_t s)
{
    /* mode switch */
    while (testset(s.flag));
    s.count--;
    if (s.count < 0) {
        /* block and place current
           process in s.queue */
        state(current) = BLOCKED
        append(s.queue, current)
    }
    s.flag=0;
}
```

```
int signal(sem_t s) {
    /* mode switch */
    while (testset(s.flag));
    s.count++;
    if (s.count ≤ 0) {
        /* unblock and remove
           first process from s */
        p=pop(s.queue);
        state(p) = RUNNING
    }
    s.flag=0;
}
```

# Critical Section Using Semaphores

---

Shared var mutex of semaphore (init =1)

Process  $P_i$ :

```
1: repeat
2:   wait(mutex)
3:   critical section
4:   signal(mutex);
5:   remainder section
6: until false;
```

Properties:

- Again simple
- No busy waiting
- Fair if queues of semaphores are implemented fair

# Bad Use of Semaphores Can Lead to Deadlocks!

---

- Let S and Q be two semaphores initialized to 1

$P_0$	$P_1$
1 wait(S);	2 wait(Q);
3 wait(Q);	4 wait(S);
⋮	⋮
signal(S);	signal(Q);
signal(Q);	signal(S);

# Synchronization Problems

---

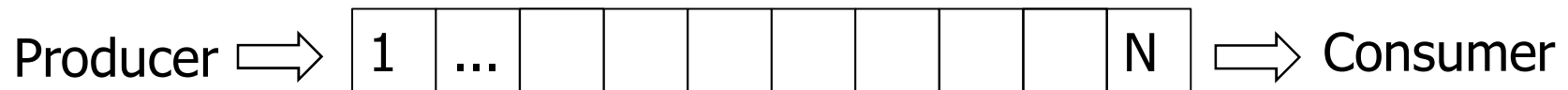
- Critical section synchronization very common
- **Problems**
  - Synchronization on many variables becomes increasingly complicated
  - Deadlock because of bad programming likely to happen
- **Idea:** Understand useful synchronization patterns that match IPC patterns
  - Consumers/Producer problem
    - High degree of concurrency by decoupling of tasks
  - Dining Philosophers
    - Coordinate access to resources
  - Readers Writers Problem
    - Coordinate access to shared data structures



# Producer/Consumer Problem Revisited

---

- Also known as **bounded buffer problem**



- Overall problem description
  - A buffer can hold N items
  - Producer pushes items into the buffer
  - Consumer pulls items from the buffer
  - Producer needs to wait when buffer is full
  - Consumer needs to wait when the buffer is empty

# Producer/Consumer Problem Revisited

```
int BUFFER_SIZE = 100; int count = 0;
```

```
void producer(void) { int item;
```

```
    while(TRUE) {
```

```
        produce_item(&item);
```

```
        if(count == BUFFER_SIZE)
            sleep ();
```

```
        enter_item(item);
```

```
        count++;
```

```
        if(count == 1)
            wakeup(consumer);
```

```
    }
```

```
}
```

```
void consumer(void) { int item;
```

```
    while(TRUE) {
```

```
        if(count == 0)
            sleep ();
```

```
        remove_item(&item);
```

```
        count--;
```

```
        if(count == BUFFER_SIZE - 1)
            wakeup(producer);
```

```
        consume_item(&item);
```

```
    }
```

```
}
```

1 Producer must wait for consumer to empty buffers

2 Consumer must wait for producer to fill buffers

3 A thread can manipulate buffer queue at a time

# Producer/Consumer Problem Revisited

---

- Synchronization between producer and consumer
  - Use of three separate semaphores
- 1. Producer must wait for consumer to empty buffers, if all full
  - Semaphore **empty** initialized to the value N
- 2. Consumer must wait for producer to fill buffers, if none full
  - Semaphore **full** initialized to the value 0
- 3. Producer/consumer can manipulate buffer queue at a time
  - Semaphore **mutex** initialized to the value 1

# Producer/Consumer Problem Revisited

---

- Semaphore **empty** initialized to the value N
- Semaphore **full** initialized to the value 0
- Semaphore **mutex** initialized to the value 1

## Producer:

```
void producer(void) {
    int item;
    while(TRUE) {
        produce_item(&item);
        wait (empty);
        wait(mutex);
        enter_item(item);
        signal(mutex);
        signal(full);
    }
}
```


## Consumer:

```
void consumer(void) {
    int item;
    while(TRUE) {
        wait(full);
        wait(mutex);
        remove_item(&item);
        signal(mutex);
        signal(empty);
        consume_item(&item);
    }
}
```

# Changing Order of Semaphores


## Producer:

```
void producer(void) {  
    int item;  
    while(TRUE) {  
        produce_item(&item);  
        wait (mutex);  
        wait(empty);  
        enter_item(item);  
        signal(mutex);  
        signal(full);  
    }  
}
```



## Consumer:

```
void consumer(void) {  
    int item;  
    while(TRUE) {  
        wait(full);  
        wait(mutex);  
        remove_item(&item);  
        signal(mutex);  
        signal(empty);  
        consume_item(&item);  
    }  
}
```

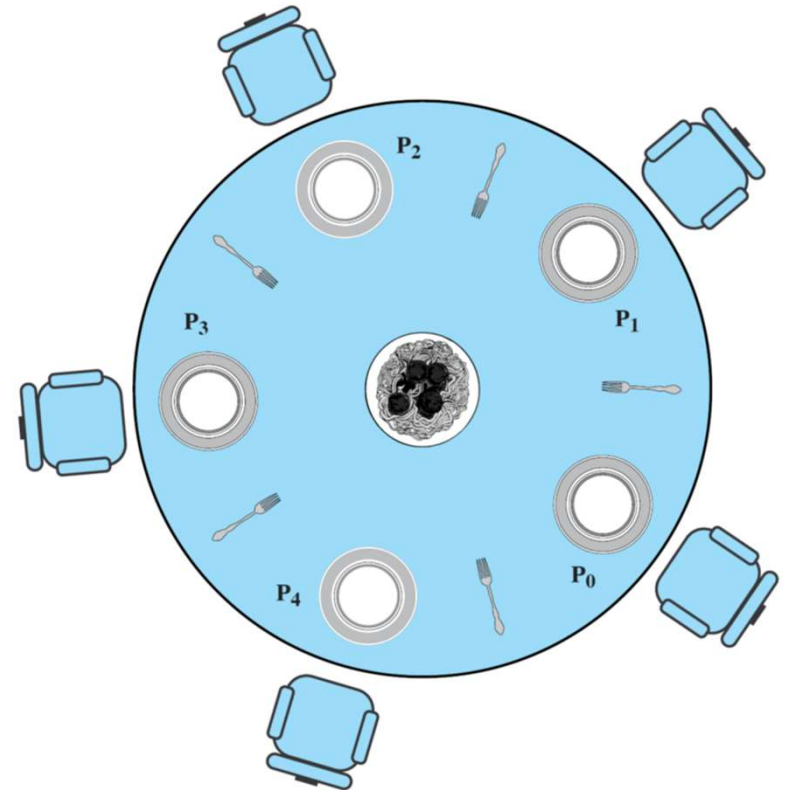


# DEADLOCK

# Dining Philosophers Problem [E.W. Dijkstra, 1965]

---

- Simplest form of the resource allocation problem
- **Intuition:**
  - **Philosophers** ~ Processes which aim to execute code
  - **Food** ~ code which requires exclusive access to resources
  - **Forks** ~ conflicts on shared resources
- **Hungry** philosophers
  - Try to **exclusively acquire** forks shared with neighbors
- **Eat** after they have acquired both forks
  - **Release** forks after eating
- **Think** if not interested to eat

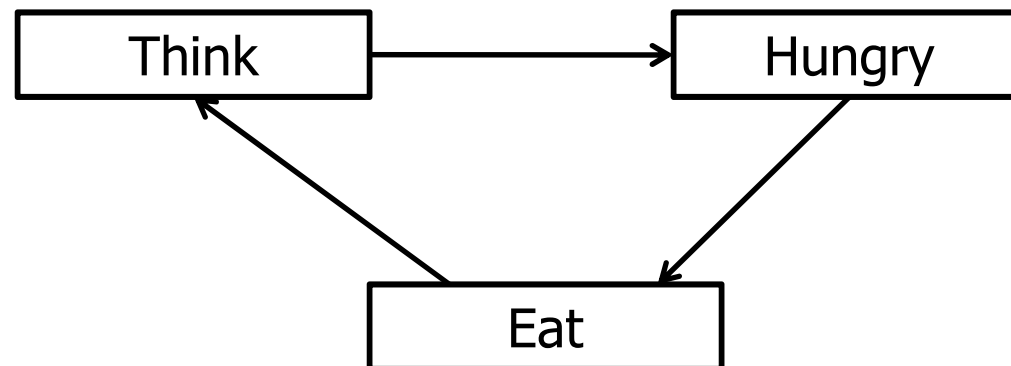


# Dining Philosophers Problem: States and Properties

---

## Goal

- No deadlocks
- Hungry philosophers will eventually be able to eat



States of a philosopher

# Dining Philosophers Problem Using Semaphores

---

- Represent each fork with a semaphore
  - Semaphore `fork[5]` initialized to 1

## Structure of Philosopher *i*

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

- Solution guarantees that no two neighbors eat simultaneously
  - Deadlock can be occur



# Semaphores Summary

---

- Semaphores can be used to solve any of traditional synchronization problems
- Semaphore have some **drawbacks**
  - They are essentially shared global variables
    - Can potentially be accessed anywhere in program
  - No connection between the semaphore and the data being controlled by the semaphore
  - Used both for critical sections (mutual exclusion) and coordination (scheduling)
  - No control or guarantee of proper usage
- Sometimes hard to use and prone to bugs
  - Another approach: Use programming language support

# Monitors

---

**Goal:** Avoid problems of managing multiple semaphores

**Monitor** is a programming language construct (not OS)

- Controls access to shared data
- Synchronization code added by compiler, enforced at runtime

```
monitor name
{
    // shared variable

    procedure P1 ( ... ) {
        ...
    }
    ...
    procedure PN (...){
        ...
    }
    initialization (...) {
        ...
    }
}
```

# Monitors

A monitor guarantees **mutual exclusion**

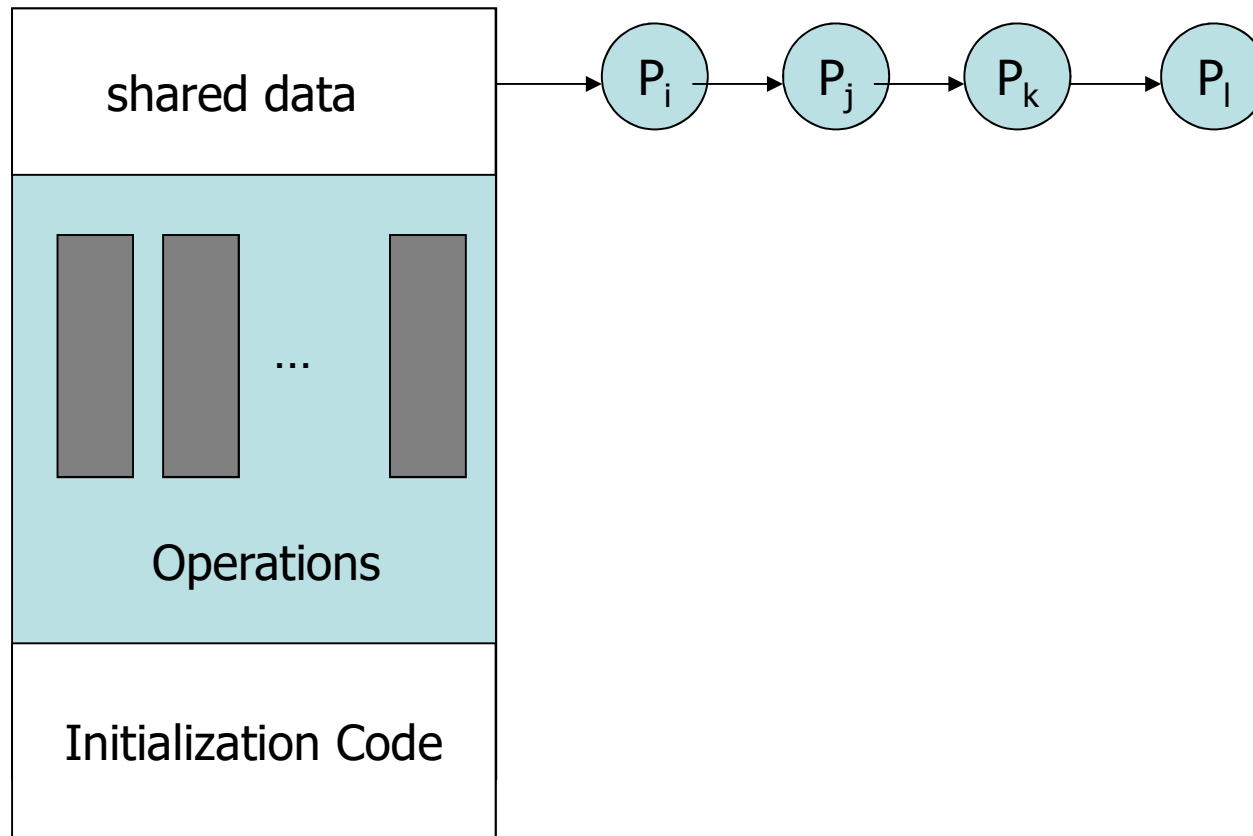
- Only **one thread** can **execute** any monitor **procedure at any time**
  - The thread is “in the monitor”
- If a **second thread invokes** a monitor **procedure** when a **first thread** is already executing one, it **blocks**
  - So the monitor has to have a **wait queue**...
- If a thread within a monitor blocks, another one can enter

```
monitor name
{
    // shared variable

    procedure P1 ( ... ) {
        ...
    }
    ...
    procedure PN (...){
        ...
    }
    initialization (...) {
        ...
    }
}
```

# Monitors

---



# Monitors

---

- A monitor has four components
  - Initialization
  - Share (private) data
  - Monitor procedures
  - Monitor entry queue
- A monitor looks like a class with
  - Constructors, private data and methods
  - Only major difference is that classes do not have entry queues

# Mutual Exclusion With a Monitor

---

```
monitor mutual_exclusion{  
    procedure CS();  
        Critical Section  
    end;  
}
```

```
Process  $P_i$ :  
1: REPEAT  
2:     CS();  
3:     Remainder Section  
4: UNTIL false;
```

# Monitors Example

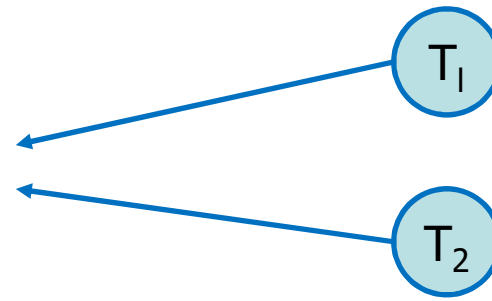
---

```
Monitor account {
```

```
    double balance;
```

```
    double withdraw(amount) {  
        balance = balance - amount;  
        return balance;  
    }
```

```
}
```



- Threads are blocked while waiting to get into monitor
- When first thread exists, another can enter

# Condition Variables in Monitors

- Similar concept to semaphores
- Condition variable associated with a queue

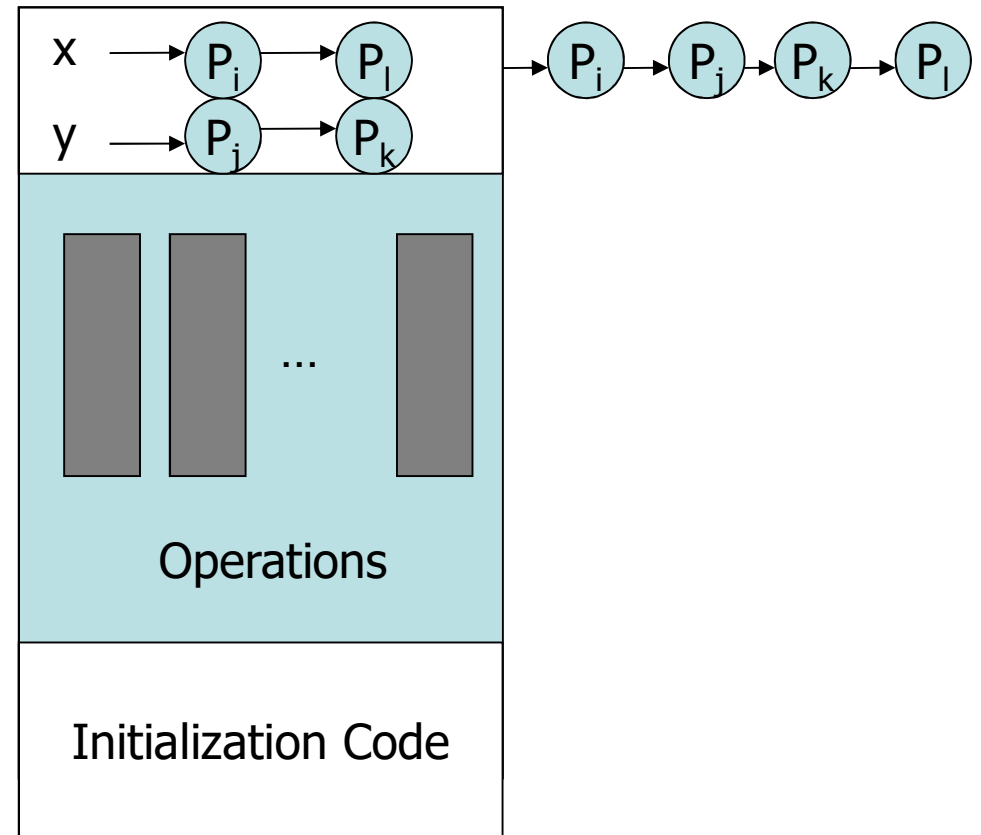
**condition** x; // declaration

**x.wait()**

- Process which invokes this operation is blocked

**x.signal()**

- A blocked process can resume execution
  - If no process issue x.wait () on variable x, then it has no effect on the variable





# Producer/Consumer Problem Revisited

```
monitor ProducerConsumer {  
    int itemCount;  
    condition full, empty;  
  
    procedure initialization( ){  
        itemCount = 0;  
    }  
    procedure add(item) {  
        if (itemCount == BUFFER_SIZE)  
            full.wait( );  
  
        putItemIntoBuffer(item);  
        itemCount = itemCount + 1;  
  
        if (itemCount == 1)  
            empty.signal();  
    }  
    procedure remove() {  
        if (itemCount == 0)  
            empty.wait( );
```

```
        item = removeItemFromBuffer();  
        itemCount = itemCount - 1;  
  
        if (itemCount == BUFFER_SIZE - 1)  
            full.signal( );  
        return item;  
    }  
}  
  
procedure producer() {  
    while (true) {  
        item = produceItem();  
        ProducerConsumer.add(item);  
    }  
}  
  
procedure consumer() {  
    while (true) {  
        item = ProducerConsumer.remove();  
        consumeItem(item);  
    }  
}
```

# Signal Semantics

---

What happens if P executes `x.signal()` & has still code to execute?

Two possibilities

- **Hoare monitors** (Concurrent Pascal, original)
  - Signal and wait: P either waits for Q to leave the monitor or waits for some other condition
  - The condition that Q was anticipating is guaranteed to hold
    - `if (empty) wait(condition);`
- **Mesa monitors** (Mesa, Java)
  - Signal and continue: Q either waits until P leaves the monitor or for some other condition
  - Condition is not necessarily true when Q runs again
    - `while (empty) wait(condition);`

# Condition Variables vs. Semaphores

## Semaphore

- Can be used anywhere in a program, but should not be used in a monitor
- `wait()` does not always block the caller (i.e., when the semaphore counter is greater than zero)
- `signal()` either releases a blocked thread, if there is one, or increases the semaphore counter
- If `signal()` releases a blocked thread, the caller and the released thread both continue

## Condition variables

- Can only be used in monitors
- `wait()` always blocks the caller
- `signal()` either releases a blocked thread, if there is one, or the signal is lost as if it never happens
- If `signal()` releases a blocked thread. Only one of the caller or the released thread can continue, but not both

# Dining Philosophers Problem Revisited

```
monitor DiningPhilosophers {
    enum {THINK,HUNGRY,EAT} 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 left & right neighbors
        test((i + 4) % 5);
        test((i + 1) % 5);
    }
    initialization_code() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }
}
```

```
void test (int i) {
    if ((state[(i+4)%5] != EATING) &&
        (state[i] == HUNGRY) &&
        (state[(i+1)%5] != EATING)) {
        state[i] = EATING ;
        self[i].signal() ;
    }
}
```

## Function of Philosopher i

```
Void philosopher(int i) {
    while (true) {
        DiningPhilosophers.pickup (i);
        // EAT
        DiningPhilosophers.putdown (i);
        // THINK
    }
}
```

**No deadlock but starvation is possible**

# Condition Variables and Locks

---

- Condition variables are also used without monitors in conjunction with blocking locks

```
procedure( ) {  
    Acquire(lock);  
    /* Do something including signal and wait*/  
    Release(lock);  
}
```

- A monitor is “just like” a module whose state includes a condition variable and a lock
  - Difference is syntactic; with monitors, compiler adds the code
- It is “just as if” each procedure in the module calls `acquire()` on entry and `release()` on exit
  - But can be done anywhere in procedure, at finer granularity

# Condition Variables and Locks – Example

---

```
Lock lock;
Condition cond;
int AddToQueue() {
    lock.Acquire();        // Lock before using shared data
    put item on queue;     // Ok to access shared data
    cond.signal();
    lock.Release();        // Unlock after done with shared data
}
int RemoveFromQueue() {
    lock.Acquire();
    while nothing on queue
        cond.wait(&lock); //Atomically release lock and block until signal
                          //Automatically re-acquire lock before it returns
                          // Difference in syntax, e.g., wait(cond, lock)
    remove item from queue;
    lock.Release();
    return item;
}
```

# Any Question So Far?

---

