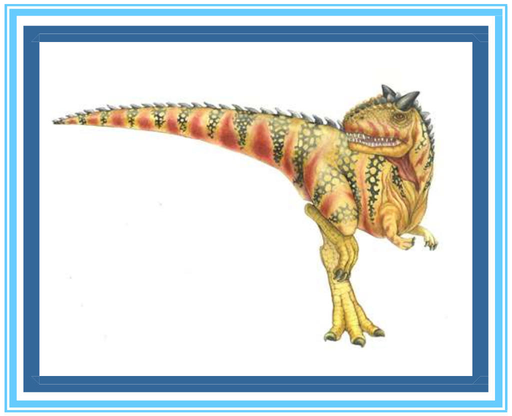
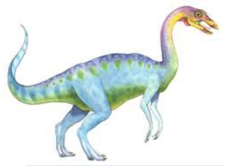
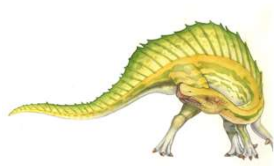
Chapter 6: Synchronization Tools

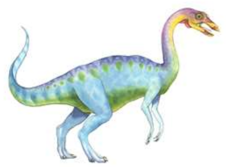




Outline

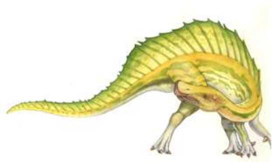
- Background
- The Critical-Section Problem
- Peterson’s Solution
- Hardware Support for Synchronization
- Mutex Locks
- Semaphores
- Monitors

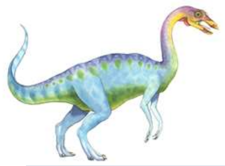




Objectives

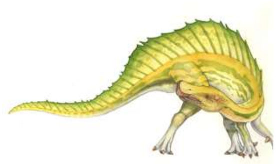
- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem





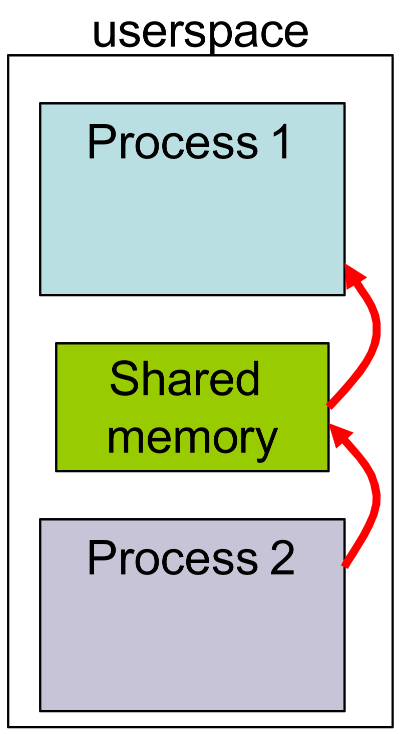
Background

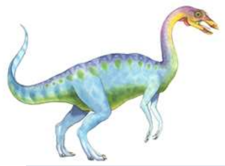
- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Shared memory is a method of IPC.
- **Concurrent access** to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes



Shared Memory

- One process will create an area in RAM which the other process can access
- Both processes can access shared memory like a regular working memory
  - Reading/writing is like regular reading/writing
  - Fast
- Limitation: Error prone. Needs synchronization between processes

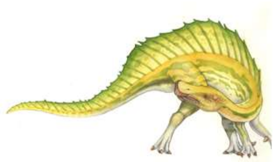




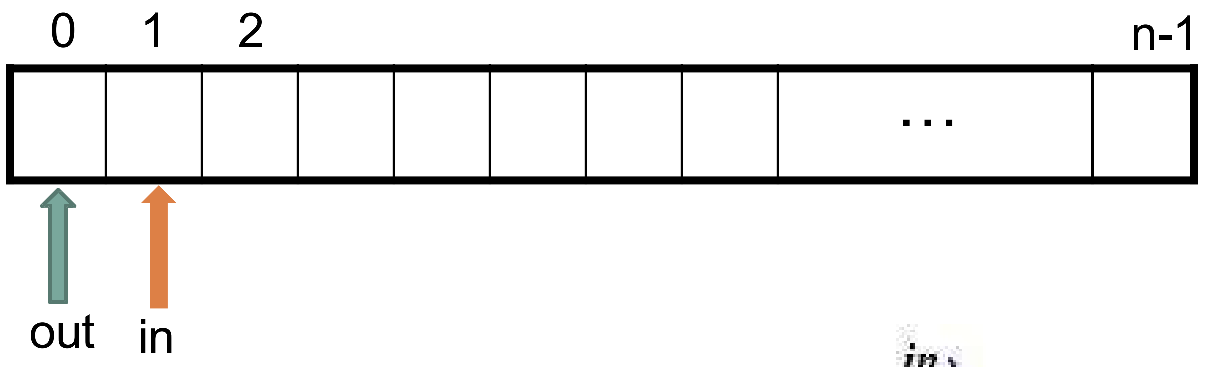
Motivation Example

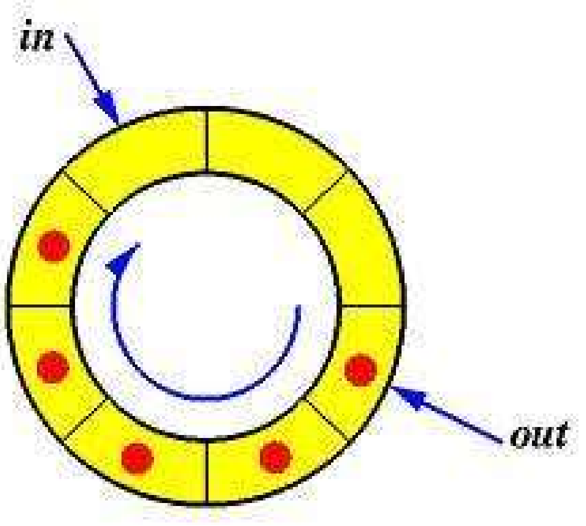
Example: Producer-Consumer Problem

- producer process produces information that is consumed by a consumer process
  - unbounded-buffer places no practical limit on the size of the buffer
  - bounded-buffer assumes that there is a fixed buffer size



Bounded Buffer Producer-Consumer





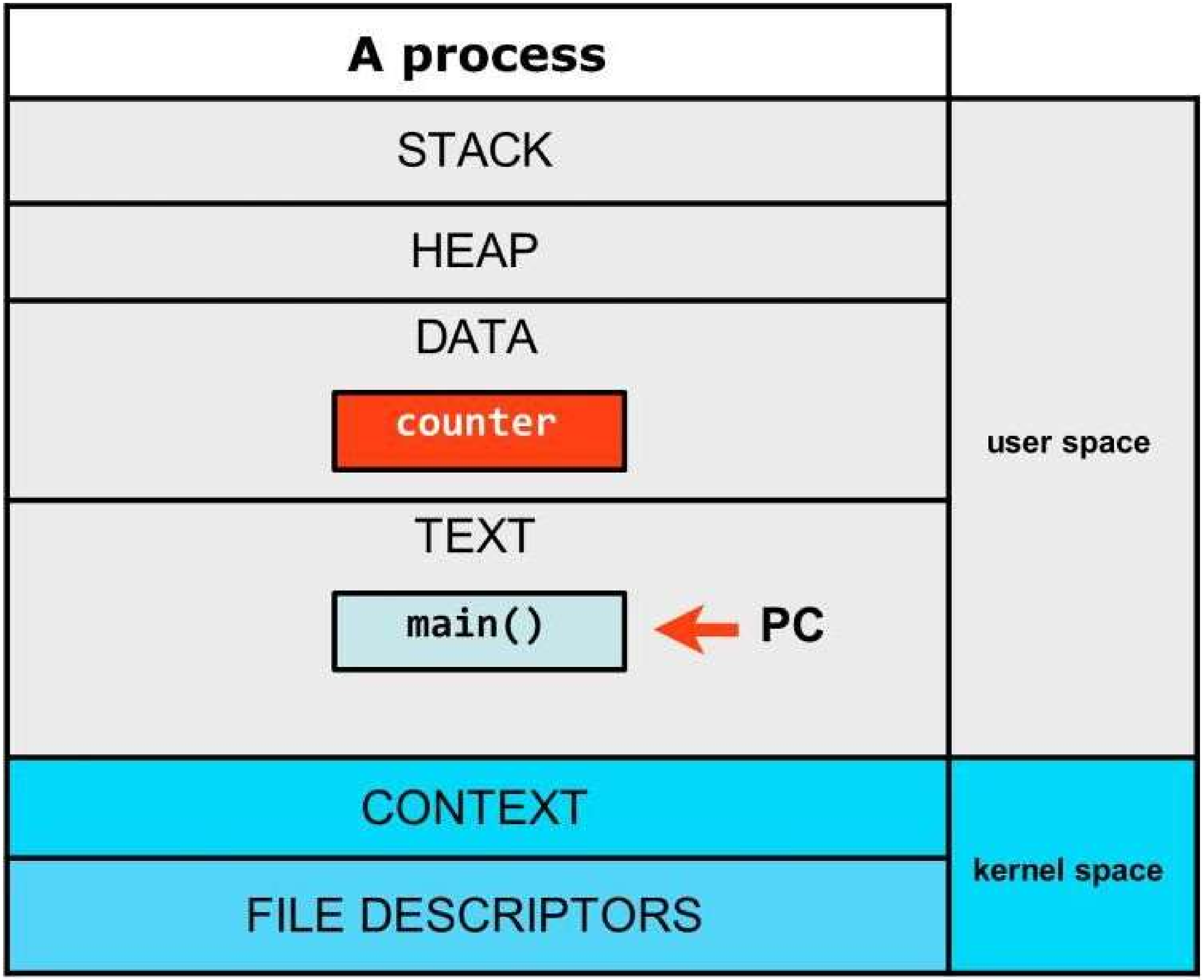
producer $\rightarrow$ in
consumer $\rightarrow$ out
Buffer is circular

Producer Code

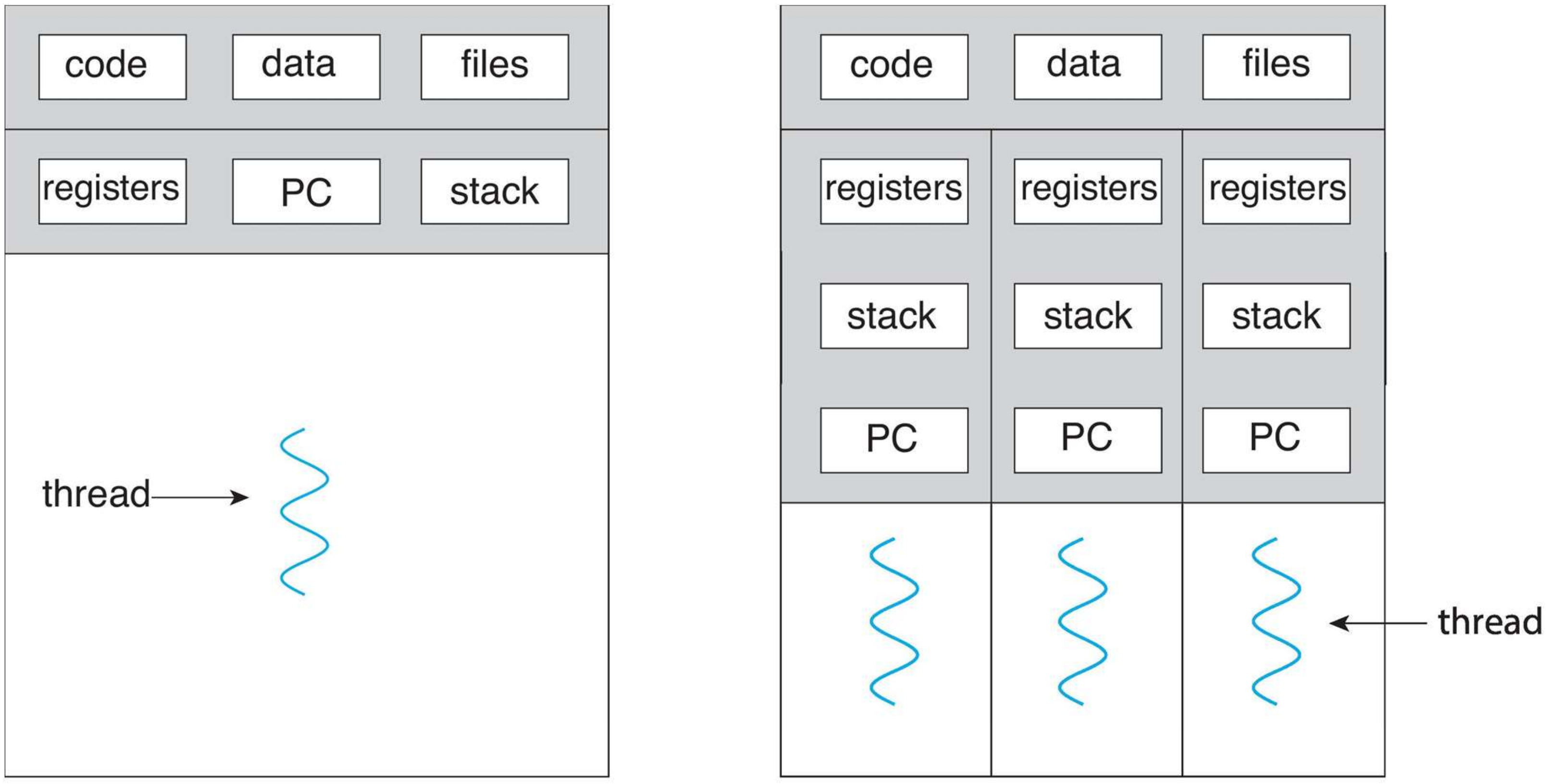
```
while (true) {
        //produce an item and put in nextProduced

        while (counter == BUFFER_SIZE);    //do nothing
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Consumer Code

```
while (true) {

          while (counter == 0);  // do nothing
          nextConsumed = buffer[out];
          out = (out + 1) % BUFFER_SIZE;
          counter--;

         //consume the item in nextConsumed
}
```

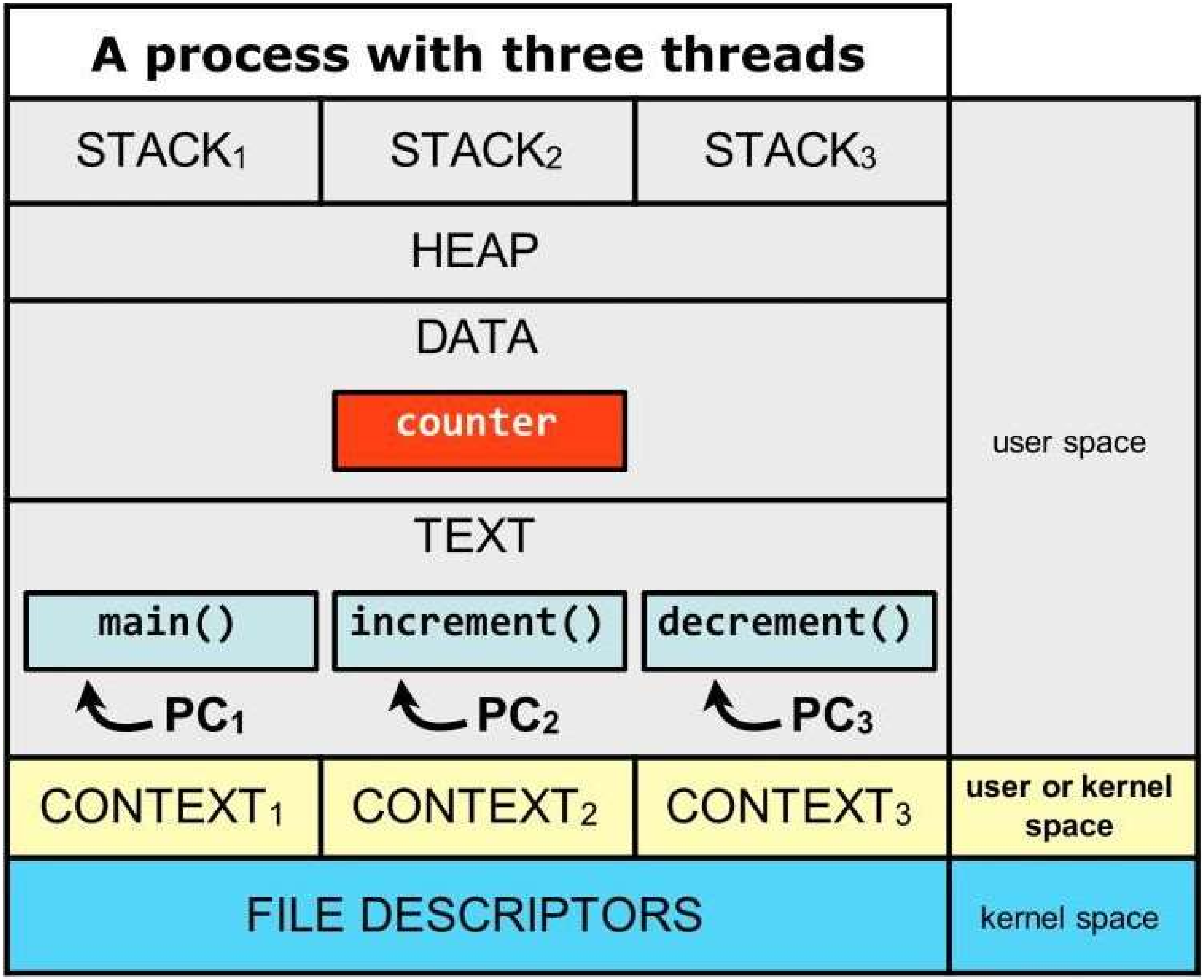

A process
STACK
HEAP
DATA
counter
TEXT
main()
PC
user space
CONTEXT
FILE DESCRIPTORS
kernel space

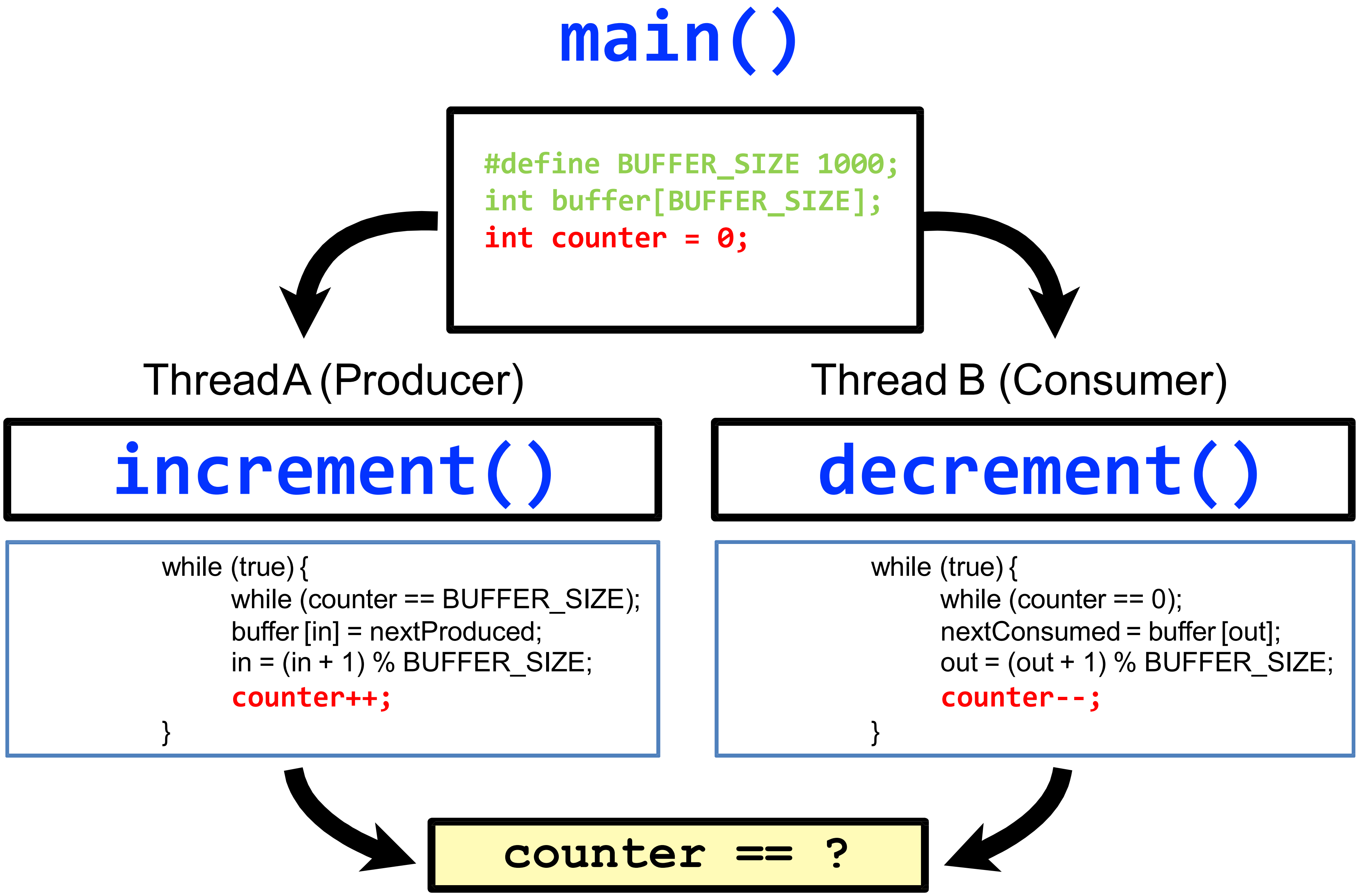
Single and Multithreaded Processes



single-threaded process

multithreaded process


A process with three threads
STACK1
STACK2
STACK3
HEAP
DATA
counter
TEXT
main()
increment()
decrement()
PC1
PC2
PC3
CONTEXT1
CONTEXT2
CONTEXT3
FILE DESCRIPTORS
user space
user or kernel space
kernel space



```
main()
```

```
#define BUFFER_SIZE 1000;
int buffer[BUFFER_SIZE];
int counter = 0;
```

Thread A (Producer)

```
increment()
```

```
while (true) {
        while (counter == BUFFER_SIZE);
        buffer [in] = nextProduced;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

Thread B (Consumer)

```
decrement()
```

```
while (true) {
        while (counter == 0);
        nextConsumed = buffer [out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

```
counter == ?
```

Motivating Scenario

shared variable

```
int counter = 5;
```

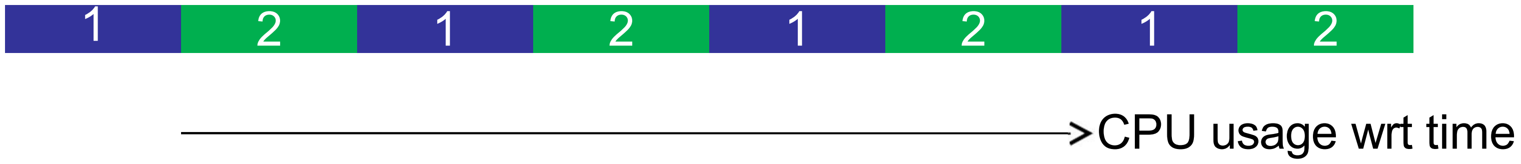
P_1

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

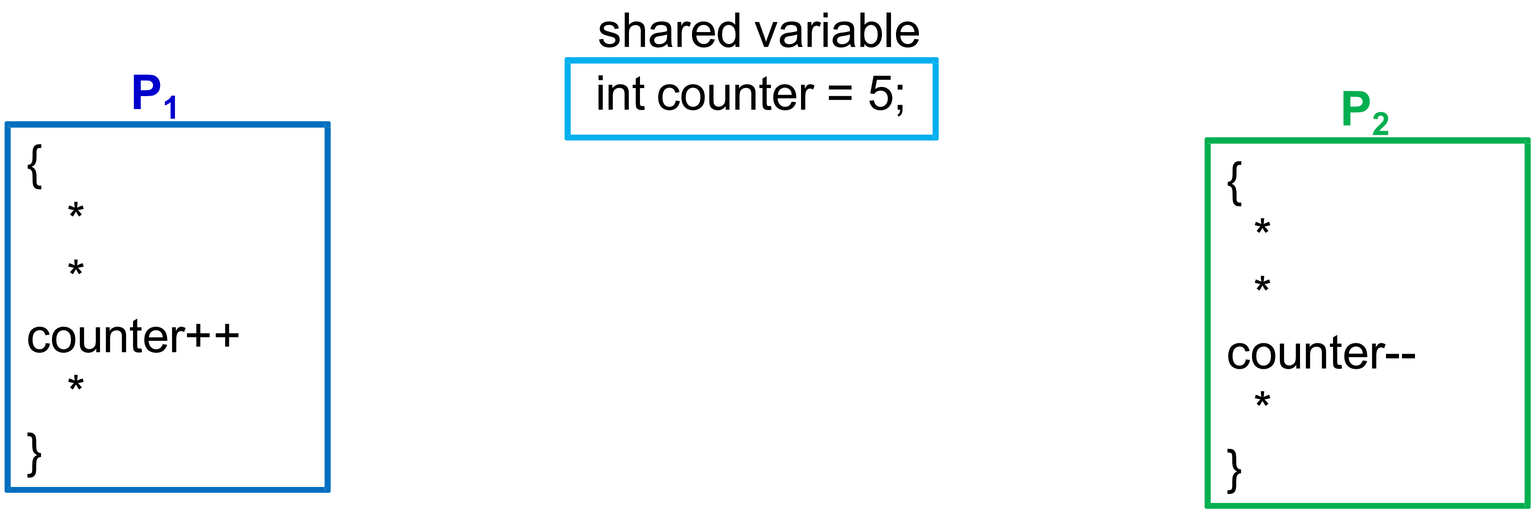
P_2

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

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



Motivating Scenario



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

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 “counter = 5” initially:
- Assume both producer and consumer reach the counter variable at the same time:

	P_0::S0: producer execute register1 = counter	{register1 = 5}
Time Out Switching From P_0 to P_1	P_0::S1: producer execute register1 = register1 + 1	{register1 = 6}
	P_1::S2: consumer execute register2 = counter	{register2 = 5}
	P_1::S3: consumer execute register2 = register2 – 1	{register2 = 4}
	P_1::S4: consumer execute counter = register2	{counter = 4}
	P_0::S5: producer execute counter = register1	{counter = 6}

Motivating Scenario

Shared variable

```
int counter = 5;
```

P_1

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

P_2

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

---

context
switch

```
R1 ← counter
R1 ← R1 + 1
counter ← R
R2 ← counter
R2 ← R2 - 1
counter ← R2
```

counter = 5

```
R1 ← counter
R2 ← counter
R2 ← R2 - 1
counter ← R2
R1 ← R1 + 1
counter ← R1
```

counter = 6

```
R2 ← counter
R1 ← counter
R1 ← R1 + 1
counter ← R1
R2 ← R2 - 1
counter ← R2
```

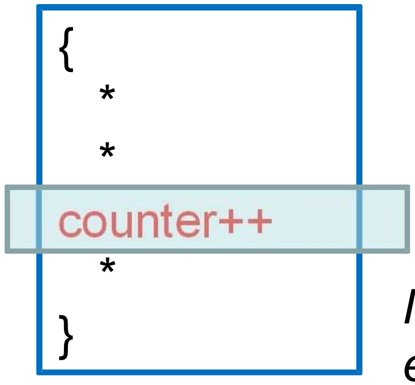
counter = 4

Race Condition

- We would arrive at this incorrect state because we allowed both processes to manipulate the variable counter **concurrently**
- **Race Condition**
  - When 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
- **Synchronization**
  - To ensure that only one process at a time can be manipulating the same data

Race Conditions

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



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

critical section

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

Race Conditions in Multicore

shared variable

```
int counter = 5;
```

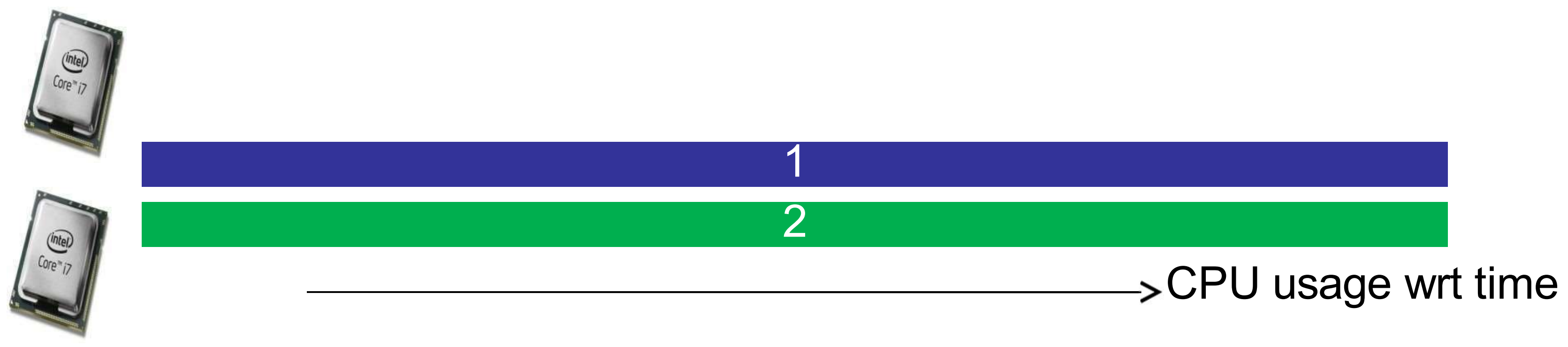
P_1

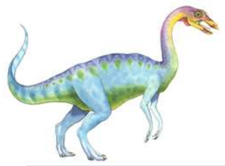
```
{
   *
   *
counter++
   *
}
```

P_2

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

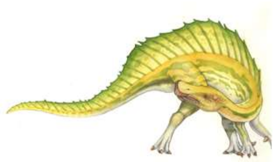
- Multi core
  - Process 1 and process 2 are executing at the same time on different cores





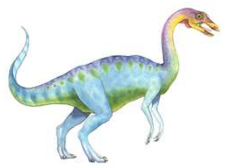
Critical Section Problem

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



The Critical-Section Problem

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



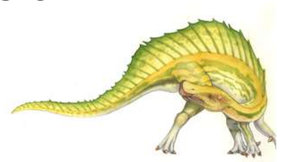
Critical-Section Problem (Cont.)

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. (No more than one process in critical section at a given time)

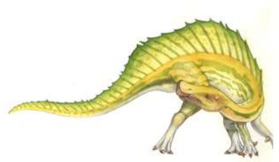
2. **Progress** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the process that will enter the critical section next cannot be postponed indefinitely. (When no process is in the critical section, any process that requests entry into the critical section must be permitted without any delay)

3. **Bounded Waiting (no starvation)** - 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. (There is an upper bound on the number of times a process enters the critical section, while another is waiting)



Three Requirements for a Solution

- Any solution to the critical section problem must satisfy three requirements:
- **Mutual Exclusion**: If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
- **Progress**: If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.
- **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.



Solutions to Critical-Section Problem

- Software-based solutions
- Hardware-based solutions
- Operating system solution (semaphore)
- Programming languages solution (monitor)

Software-based solutions

- Unfortunately, there are no guarantees that software-based solution work correctly on modern architectures
  - Because of the way modern architectures perform basic machine-language instructions, such as load and store
- We present theses solutions because
  - They provides a good algorithmic description of solving the critical-section problem
  - Illustrates some of the complexities involved in designing software that addresses the requirements

Turn-based Solution

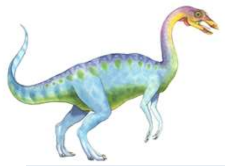
- Assumptions
  - There are only two processes: P_0 and P_1
  - The LOAD and STORE instructions are atomic; that is, cannot be interrupted
- The two processes share a variable turn
  - int turn;
- The variable turn indicates whose turn it is to enter the critical section

Turn-based Solution

- Algorithm for process P_i

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

- Problem
  - What happened if a process wants to enter the critical section again, before the other one needs to enter?
  - The solution does not meet the **Progress requirement**



Correctness of the Software Solution

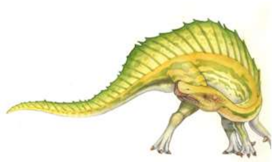
- Mutual exclusion is preserved

  `P`$_i$ enters critical section only if:

  `turn = i`

  and **`turn`** cannot be both 0 and 1 at the same time

- What about the Progress requirement? **No**
- What about the Bounded-waiting requirement? **Yes**



Flag-based Solution

- The two processes share a variable flag
  - boolean flag[2];
- 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!

Flag-based Solution

- Algorithm for process P_i

```
do {
        flag[i] = TRUE;
        while (flag[j]);
                critical section
        flag[i] = FALSE;
                remainder section
} while (TRUE);
```

- Problem
  - What happened if both processes want to enter the critical section at the same time, and both set their flags true, before entering the critical section
  - The solution does not meet the **Progress requirement**.

Peterson's Solution

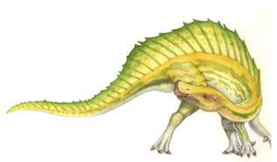
- Algorithm for process P_i

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

- Provable that all requirements are satisfied but works for only two processes => Bakery algorithm was proposed, but it was too complex to check the entry section => hardware solutions

Peterson’s Solution

- Two process solution
- Assume that the **`load`** and **`store`** machine-language instructions are atomic; that is, cannot be interrupted
- The two processes share two variables:
  - **`int turn;`**
  - **`boolean flag[2]`**
- The variable **`turn`** indicates whose turn it is to enter the critical section
- The **`flag`** array is used to indicate if a process is ready to enter the critical section.
  - **`flag[i] =`** ***`true`*** implies that process $\mathbf{P}_i$ is ready!

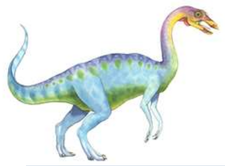
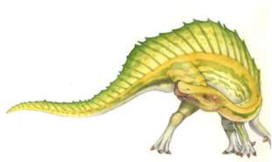


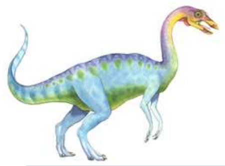
Correctness of Peterson's Solution

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

     `Pi` enters CS only if:

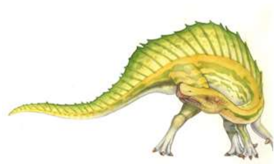
     either **`flag[j] = false`** or **`turn = i`**
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met



Peterson’s Solution and Modern Architecture

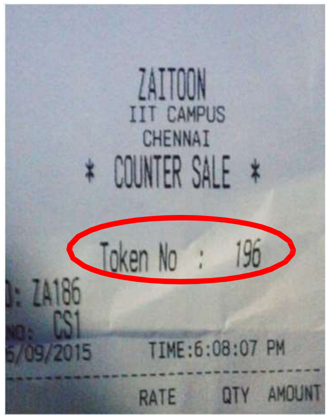
- Although useful for demonstrating an algorithm, Peterson’s Solution is not guaranteed to work on modern architectures.
  - To improve performance, processors and/or compilers may reorder operations that have no dependencies
- Understanding why it will not work is useful for better understanding race conditions.
- For single-threaded this is ok as the result will always be the same.
- For multithreaded the reordering may produce inconsistent or unexpected results!



Bakery Algorithm

- Synchronization between N > 2 processes
- By Leslie Lamport

eat
when 196 displayed


ZAITOON
IIT CAMPUS
CHENNAI
* COUNTER SALE *
Token No : 196
: ZA186
NO: CS1
6/09/2015 TIME:6:08:07 PM
RATE QTY AMOUNT

wait your turn!!

http://research.microsoft.com/en-us/um/people/lamport/pubs/bakery.pdf

Simplified Bakery Algorithm

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

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

critical section

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

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

https://www.youtube.com/watch?v=3pUScfud9Sg

Original Bakery Algorithm

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

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

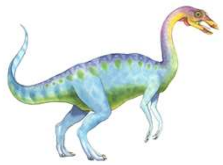
doorway

critical section

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

Choosing ensures that a process is not at the doorway

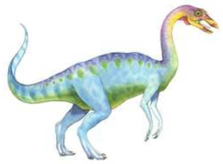
(a, b) < (c, d) which is equivalent to: (a < c) or ((a == c) and (b < d))



The Bakery Algorithm
code of process i , $i \in \{1 ,..., n\}$

```
choosing[i] = true
number[i] = 1 + max {number[j] | (1 ≤ j ≤ n)}
choosing[i] = false
for j = 1 to n {
    await choosing[j] = false
    await (number[j] = 0) ∨ (number[j],j) ≥ (number[i],i)
}
critical section
number[i] = 0
```

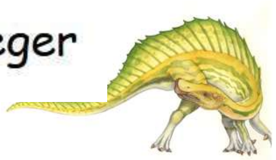
	1	2	3	4	--------	n	
choosing	false	false	false	false	false	false	bits
number	0	0	0	0	0	0	integer



The Bakery Algorithm
code of process i , $i \in \{1 ,..., n\}$

```
choosing[i] = true
number[i] = 1 + max {number[j] | (1 ≤ j ≤ n)}
choosing[i] = false
for j = 1 to n {
      await choosing[j] = false   1. Wait for j to choose a number
      await (number[j] = 0) ∨ (number[j],j) ≥ (number[i],i)
}
critical section
number[i] = 0
```

	1	2	3	4	--------	n	
choosing	false	false	false	false	false	false	bits
number	0	0	0	0	0	0	integer



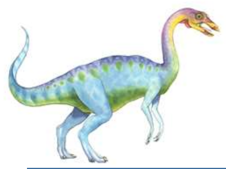
The Bakery Algorithm
code of process i , $i \in \{1 ,..., n\}$

```
choosing[i] = true
number[i] = 1 + max {number[j] | (1 ≤ j ≤ n)}
choosing[i] = false
for j = 1 to n {
    await choosing[j] = false
    await (number[j] = 0) ∨ (
}
critical section
number[i] = 0
```

1. Wait for j to choose a number

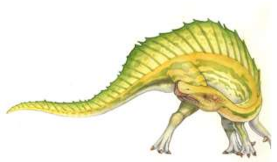
2. Wait for j to finish its critical section

	1	2	3	4	--------	n	
choosing	false	false	false	false	false	false	bits
number	0	0	0	0	0	0	integer



Process Synchronization

Hardware Solutions



Solution Using Locks

- Race conditions are prevented by requiring that critical regions be protected by locks
  - A process must acquire a lock before entering a critical section; it releases the lock when it exits the critical section
- Unfortunately, the design of such locks can be quite sophisticated.

```
do {
    acquire lock
            critical section
    release lock
            remainder section
} while (TRUE);
```

Hardware-based Solutions

- Hardware features can make any programming task easier and improve system efficiency
- Many systems provide hardware support for critical section code
  - Disabling interrupts
  - Test and Set instruction
  - Exchange instruction

Disabling interrupts

```
do {
        Disable interrupt
                critical section
        Enable interrupt
                remainder section
  } while (TRUE);
```

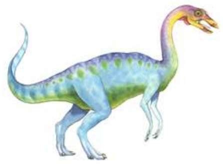
- Critical section code would execute without preemption
- Only works in **uniprocessor** systems
  - Generally too inefficient on multiprocessor systems

Hardware Instructions

- Special hardware instructions that allow us to either *test-and-modify* the content of a word, or two *swap* the contents of two words atomically (uninterruptedly.)
  - **Test-and-Set (TSL)** instruction
  - **Compare-and-Swap** instruction

Test and Set Instruction

- Modern machines provide special atomic hardware instruction to test and modify the content of a word atomically.
  - that is, as one uninterruptible unit (atomic)

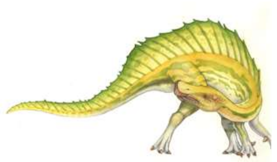


The test_and_set Instruction

- Definition

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

- Properties
  - Executed atomically (uninterruptible unit)
  - Returns the original value of passed parameter
  - Set the new value of passed parameter to **`true`**



Test and Set Instruction

- Algorithm for process P_i

```
boolean lock = FALSE;      //shared variable between all processes
do {
        while (TestAndSet (&lock) ) ;
                critical section
        lock = FALSE;
                remainder section
  } while (TRUE);
```

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

Does lock and test solve the critical-section problem?

Swap Instruction

- Swap contents of two memory words atomically
- Algorithm for process P_i

```
boolean lock = FALSE;  // shared variable between all processes
boolean key;           // local variable for each process
do {
        key = TRUE;
        while (key == TRUE )   swap(&lock, &key);
                critical section
        lock = FALSE;
                remainder section
} while (TRUE);
```

A Bounded-waiting Solution

- Hardware-based solutions do not satisfy the bounded-waiting requirement

```
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);
```

Semaphore

- A synchronization tool which is provided by **OS**
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard **atomic** operations
    - wait() or P()
    - signal() or V()

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

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

Mutual Exclusion using Semaphores

```
Semaphore mutex;    // initialized to 1
do {
    wait (mutex);
        critical Section
    signal (mutex);
        remainder section
} while (TRUE);
```

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

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

Semaphore

- Types of semaphores
  - Binary semaphore – integer value can range only between 0 and 1; can be simpler to implement
    - Also known as mutex locks (Silberschatz definition)
  - Counting (general) semaphore – integer value can range over an unrestricted domain
- Binary semaphores can be used to deal with the critical-section problem (See next slide)
- Counting semaphores can be used to control access to a given resource consisting of a **finite number** of instances (See producer- consumer solution using semaphores)

Semaphore Implementation

- Main disadvantage: busy waiting
  - While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code
- This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock
  - Spinlocks do have an advantage in that no context switch is required when a process must wait on a lock
  - Thus, when locks are expected to be held for short times, spinlocks are useful

Semaphore Implementation

- Toovercome the need for busy waiting, we can modify the definition of the wait() and signal()
- With each semaphore there is an associated waiting queue
- Two operations are provided by the operating system as basic system calls
  - **block** place the process invoking the operation on the appropriate waiting queue and switch it to blocked state
  - **wakeup** remove one of processes in the waiting queue and place it in the ready queue

Semaphore Implementation

```
struct semaphore {
    int value;
    queueType   queue;    //a waiting queue of blocked processes
}
```

```
wait(semaphore S) {
    S.value --;
    if (S.value < 0) {
        add this process to S.queue;
        block();
    }
}
```

```
signal(semaphore S) {
    S.value ++;
    if (S.value <= 0) {
        remove a process P from S.queue;
        wakeup(P);
    }
}
```

Semaphore Implementation

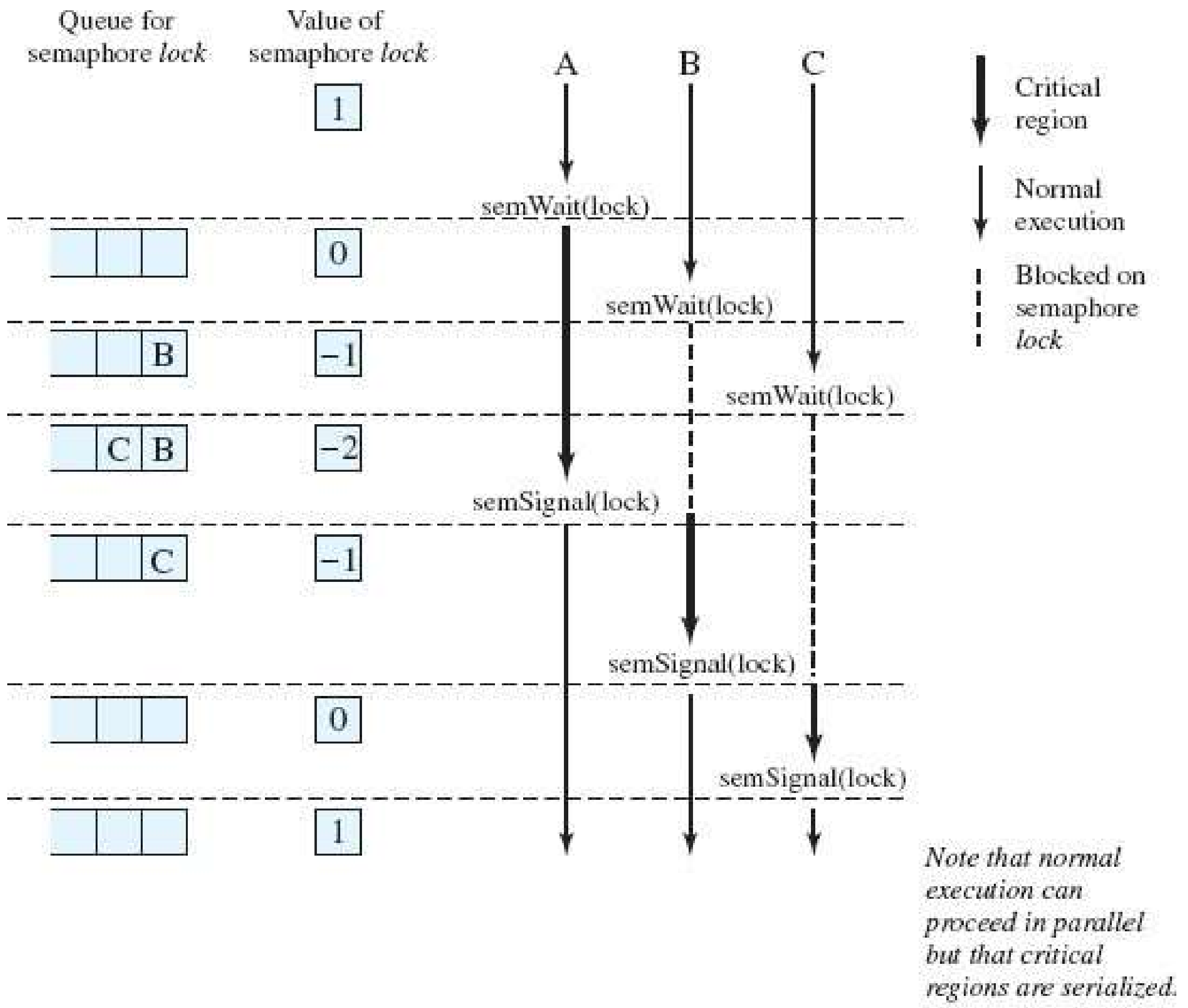

Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
1
semWait(lock)
0
semWait(lock)
B
−1
semWait(lock)
C B
−2
semSignal(lock)
C
−1
semSignal(lock)
0
semSignal(lock)
1
Critical region
Normal execution
Blocked on semaphore *lock*
Note that normal execution can proceed in parallel but that critical regions are serialized.

Figure 5.7 **Processes Accessing Shared Data Protected by a Semaphore**

Semaphore Implementation

- Must guarantee that no two processes can execute wait() and signal() on the same semaphore at the same time
- Thus, 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
  - Thus, the critical section is almost never occupied, and busy waiting occurs rarely, and then for only a short time.

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

```
     P0                   P1
wait (S);            wait (Q);
wait (Q);            wait (S);
  ...                  ...
signal (S);          signal (Q);
signal (Q);          signal (S);
```

- Starvation or indefinite blocking: a situation in which processes wait indefinitely within the semaphore.

Classic Problems of Synchronization

- We present a number of synchronization problems as examples of a large class of concurrency-control problems
- Classical problems used to test newly-proposed synchronization schemes
- Examples
  - Producer-Consumer (Bounded-Buffer) Problem
  - Readers and Writers Problem
  - Dining-Philosophers Problem

Producer-Consumer Problem

- *N* buffers, each can hold one item
- Semaphore mutex provides mutual exclusion for accesses to the buffer pool, initialized to the value 1
- Semaphore full counts the number of full buffers, initialized to the value 0
- Semaphore empty counts the number of empty buffers, initialized to the value N

Producer-Consumer Problem

Producer:

```
do  {
    //produce an item in nextp
    wait (empty);
    wait (mutex);

    //add the item to the buffer

    signal (mutex);
    signal (full);
} while (TRUE);
```

Consumer:

```
do {
    wait (full);
    wait (mutex);

    //remove an item from buffer to nextc

    signal (mutex);
    signal (empty);

    //consume the item in nextc

} while (TRUE);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - Readers: only read the data set
  - Writers: can both read and write
- Problem
  - Allow multiple readers to read at the same time
  - Only one single writer can access the shared data at the same time (no other reader or writer)
- Some systems provides reader-writer locks to the users
- Several variations of how readers and writers are treated, all involve priorities

Readers-Writers Problem

- First readers-writers problem
  - No reader should wait for other readers to finish simply because a writer is waiting
- Second readers-writers problem
  - If a writer is waiting to access the object, no new readers may start reading
- Shared Data
  - Semaphore mutex initialized to 1
  - Semaphore wrt initialized to 1
  - Integer readcount initialized to 0

First Readers-Writers Problem

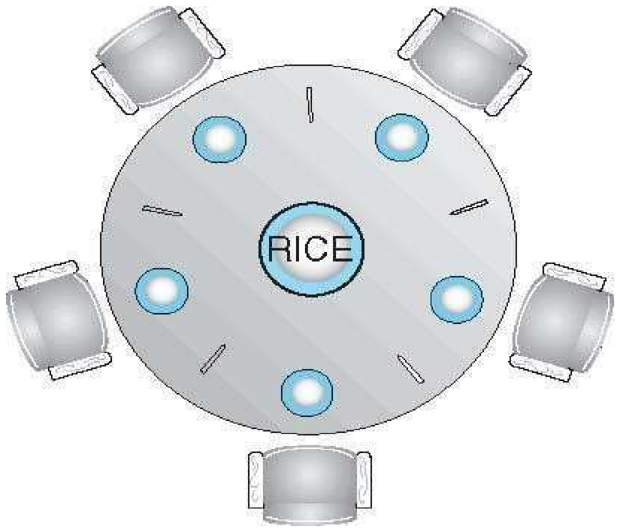
Writer:

```
do {
    wait (wrt);

    //writing is performed


    signal (wrt);

} while (TRUE);
```

Reader:

```
do {
    wait (mutex);
    readcount ++;
    if (readcount == 1)
        wait (wrt);
    signal (mutex);
    // reading is performed
    wait (mutex);
    readcount--;
    if (readcount == 0)
        signal (wrt);
    signal (mutex);
} while (TRUE);
```

The Dining-Philosophers Problem



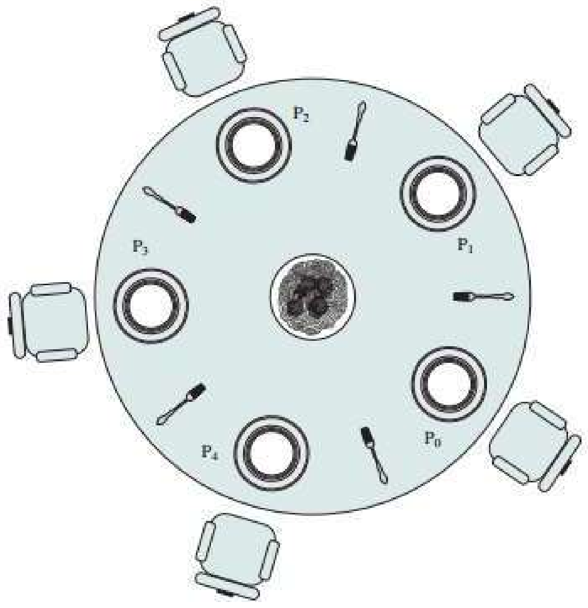
- Consider five philosophers spend their lives thinking and eating
- From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to her (one at a time) to eat from bowl
- Need both to eat, then release both when done

The Dining-Philosophers Problem

- A simple solution: one semaphore for each chopstick
  - Semaphore chopstick[5] initialized to 1

- The structure of philosopher i

```
do {
        wait ( chopstick[i] );
        wait ( chopstick[ (i + 1) % 5] );
            // eating
        signal ( chopstick[i] );
        signal (chopstick[ (i + 1) % 5] );
            // thinking
  } while (TRUE)
```



The Dining-Philosophers Problem

- Problem
  - Deadlock
  - Suppose that all five philosophers become hungry simultaneously
- Solutions
  - Allow at most four philosophers to sit around the table
  - Allow a philosopher to pick up her chopsticks only if both chopsticks are available
  - Use an asymmetric solution
    - An odd philosopher picks up first her left chopstick
    - An even philosopher picks up first her right chopstick

Problems with Semaphores

- Incorrect use of semaphore operations

- Example: semaphore solution to the CS problem
  - signal (mutex) …. wait (mutex) ⍰ ME is violated
  - wait (mutex) … wait (mutex) ⍰ Deadlock
  - Omitting of wait (mutex) or signal (mutex) (or both)

- Solution
  - Monitors: A high-level abstraction that provides a convenient and effective mechanism for process synchronization

Monitors

- The monitor is a programming-language construct that provides equivalent functionality to that of semaphores and that is easier to control.
- Implemented in a number of programming languages, including
  - Concurrent Pascal, Modula-2 and 3, C# and Java
- Monitor is an abstract data type which consists of
  - Internal (private) variables
  - Procedures (public methods)

Monitors

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

- There are two important characteristics
  - Local variables are accessible only by the local methods
  - Only one process may be executing in the monitor at a time

Producer-Consumer Problem

Producer:

```
do  {
    //produce an item in nextp
    wait (empty);
    wait (mutex);

    //add the item to the buffer

    signal (mutex);
    signal (full);
} while (TRUE);
```

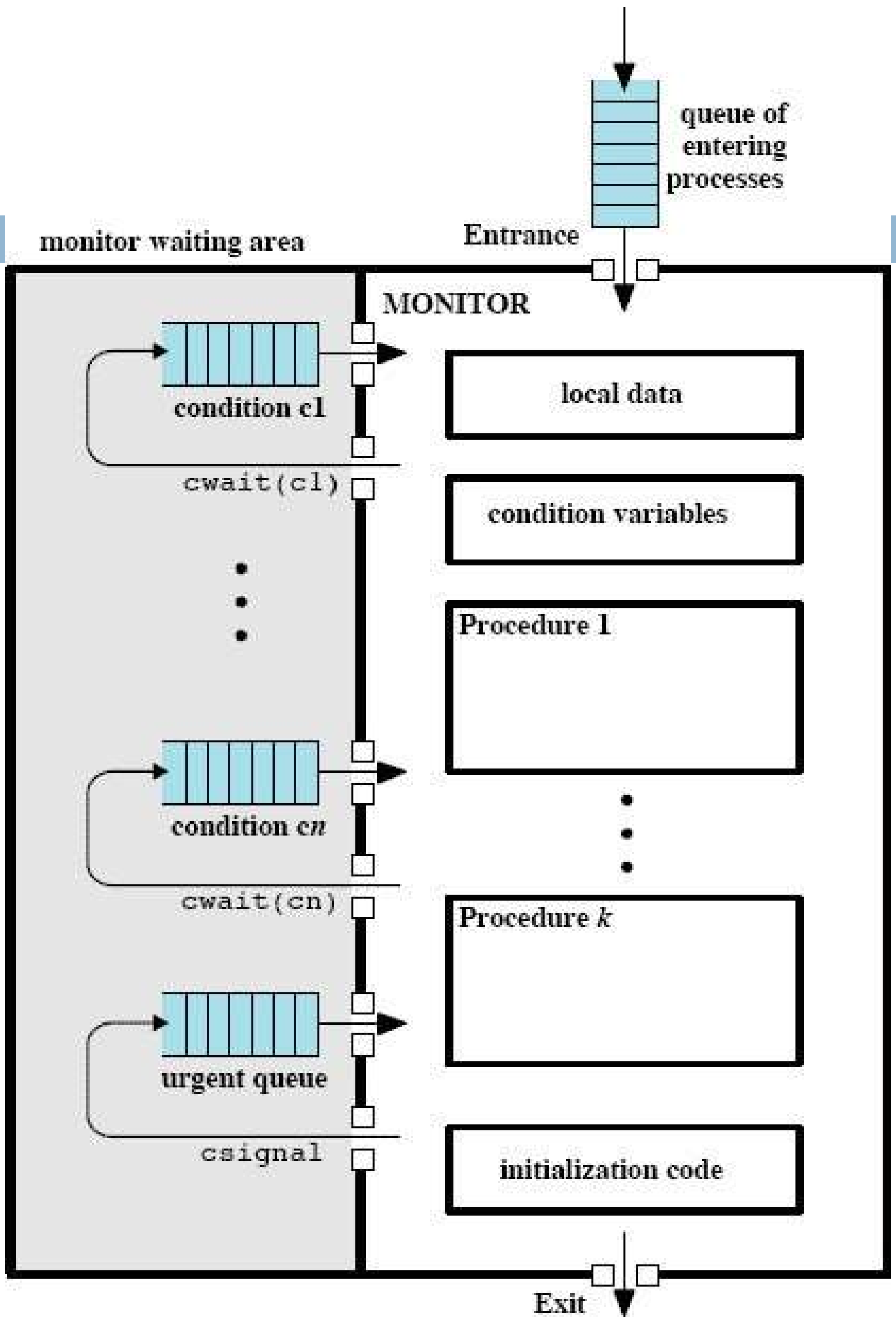
Consumer:

```
do {
    wait (full);
    wait (mutex);

    //remove an item from buffer to nextc

    signal (mutex);
    signal (empty);

    //consume the item in nextc

} while (TRUE);
```

Monitors

- Mutual Exclusion can be achieved by this construct
- But it is not powerful enough to model some synchronization schemes
- We need to define additional synchronization mechanisms: condition variables
  - Example: condition x, y;
- There are Two operations on a condition variable:
  - x.wait() – a process that invokes the operation is suspended on condition x until another process invokes x.signal()
  - x.signal() – resumes exactly one suspended process on x
    - If no process is suspended, then the signal() operation has no effect

Monitors



queue of entering processes
Entrance
monitor waiting area
MONITOR
condition c1
cwait(c1)
condition cn
cwait(cn)
urgent queue
csignal
local data
condition variables
Procedure 1
Procedure k
initialization code
Exit

Producer-Consumer with Monitors

```
monitor ProducerConsumer {
    int buffer[N];
    int count, in, out;
    condition empty, full;

    void produce(int nextp) {
        if (count == N) full.wait();
        buffer[in] = nextp;
        in = (in + 1) % N;
        count = count + 1;
        If (count == 1) empty.signal();
    }
```

Producer-Consumer with Monitors

```
int consume() {
        int nextc;
         if (count == 0) empty.wait();
             nextc = buffer[out];
             out = (out + 1) % N;
             count = count - 1;
             If (count == N - 1) full.signal();
             return (nextc);
    }

    initialization_code() {
        count = in = out = 0;
    }
}
```

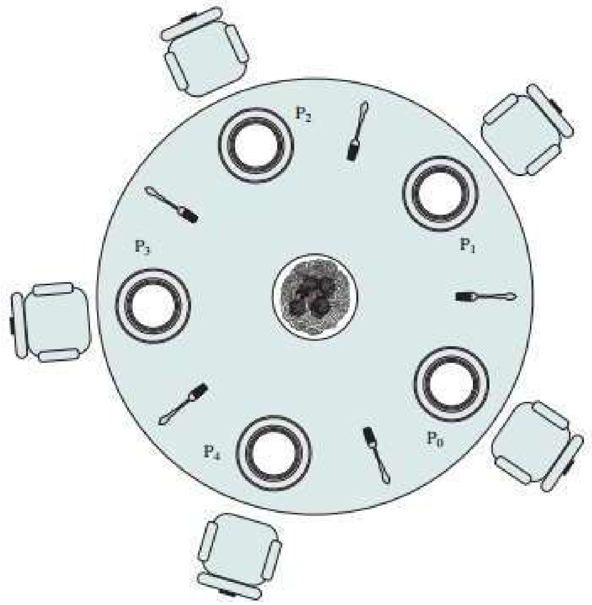
Producer-Consumer with Monitors

```
monitor  ProducerConsumer  pc;

Producer:
        do {
          // produce an item in nextp
            pc.produce(nextp);
        } while (TRUE);

Consumer:
        do {
          nextc = pc.consume();
          // consume the item in nextc
        } while (TRUE);
```

Dining-Philosophers with Monitors

Allow a philosopher to pick up her chopsticks only if both chopsticks are available

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

    void pickup(int i) {    // philosopher i
            state [i] = HUNGRY;
            test(i);    // check if the two neighbors are eating or not
            if (state[i] != EATING)
                self[i].wait();
    }
```



Dining-Philosophers with Monitors

```
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();
        }
}
```

Dining-Philosophers with Monitors

```
    initialization-code () {
            for (int i = 0; i < 5; i++)
                    state[i] = THINKING;
    }
} // end monitor dp

philosopher i:
    do {
            …
            dp.pickup(i);
            …
            dp.putdown(i);
    } while (TRUE);
```

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 must wait
- Options include
  - **Signal and wait (Hoare)** – P immediately leaves the monitor (blocked), Q is resumed
  - **Signal and continue (Lampson/Redell**) – P continues and Q waits until P leaves the monitor or waits for another condition

Condition Variables Choices

- Both have pros and cons – language implementer can decide
- The Producer-Consumer code is written for Hoare's proposal
- For Lampson/Redell's method, we should replace
  - `if (count == N) full.wait();` with `while (count == N) full.wait();`
  - And we should do the same for consume()

Further Reading

- A solution to Dining Philosophersusing monitors (without deadlock)
- Implementing a monitor using semaphores
- Synchronization examples in Solaris, Windows XP and Linux
- Producer-Consumer using message passing (Stallings)
- A solution to the Readers/Writersproblem using semaphore: writers have priority (Stallings)

End of Chapter 6