Chapter 6: Process Synchronization

Course Supervisor: Anaum Hamid

OUTLINE

1. Background
2. The Critical-Section Problem
3. Peterson’s Solution
4. Hardware Support for Synchronization
5. Mutex Locks
6. Semaphores
7. Deadlock and Starvation
8. Classical Problems of Synchonization.

Objectives

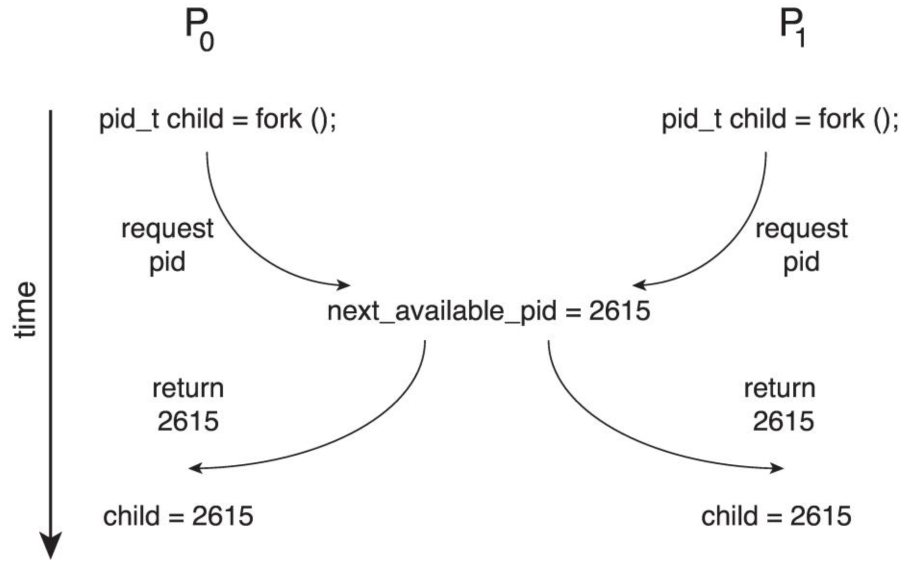
- Describe the critical-section problem and illustrate a race condition
- Illustrate hardware solutions to the critical-section problem using memory barriers, compare-and-swap operations, and atomic variables
- Demonstrate how mutex locks, semaphores, monitors, and condition variables can be used to solve the critical section problem
- Evaluate tools that solve the critical-section problem in low-, Moderate-, and high-contention scenarios

Background

- Processes can execute concurrently
  - May be interrupted at any time, partially completing execution
- Concurrent access to shared data may result in data inconsistency
- Maintaining data consistency requires mechanisms to ensure the orderly execution of cooperating processes
- We illustrated in chapter 4 the problem when we considered the Bounded Buffer problem with use of a counter that is updated concurrently by the producer and consumer,. Which lead to race condition.

Race Condition

- Processes P_0 and P_1 are creating child processes using the `fork()` system call
- Race condition on kernel variable `next_available_pid` which represents the next available process identifier (pid).



- Unless there is a mechanism to prevent P_0 and P_1 from accessing the variable `next_available_pid` the same pid could be assigned to two different processes!

Producer Consumer Problem

- **`counter++`** could be implemented as

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

- **`counter--`** could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = 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` {counter = 6 }
  S5: consumer execute `counter = register2` {counter = 4}

Race Condition

- Several processes access and manipulate the same data concurrently and the outcome of the execution depends on the order in which the access takes place, is called a **RACE CONDITION**.

- To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

Critical Section Problem

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

Critical Section

- General structure of process P_i

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

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
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.
3. **Bounded Waiting** - A bound must exist on the number of times that other processes can 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

Critical-Section Handling in OS

Two approaches depending on if kernel is preemptive or non- preemptive

- **Preemptive** – allows preemption of process when running in kernel mode
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode

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 P_i is ready!

Algorithm for Process P_i

```
while (true){

      flag[i] = true;
      turn = j;
      while (flag[j] && turn = = j)
             ;

         /* critical section */

      flag[i] = false;

      /* remainder section */

}
```

Correctness of Peterson's Solution

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

     P_i enters CS only if:

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

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!

Modern Architecture Example

- Two threads share the data:

```
boolean flag = false;
 int x = 0;
```

- Thread 1 performs

```
while (!flag)
  ;
 print x
```

- Thread 2 performs

```
x = 100;
 flag = true
```

- What is the expected output?

    100

Modern Architecture Example (Cont.)

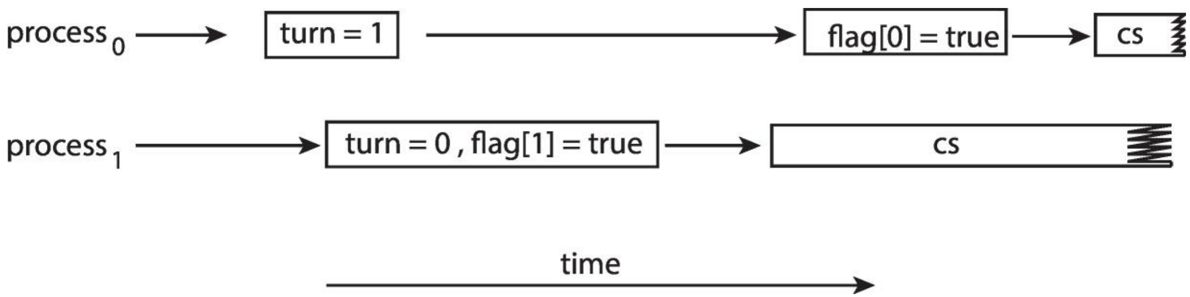
- However, since the variables `flag` and `x` are independent of each other, the instructions:

```
flag = true;
  x = 100;
```

  for Thread 2 may be reordered
- If this occurs, the output may be 0!

Peterson's Solution Revisited

- The effects of instruction reordering in Peterson's Solution



- This allows both processes to be in their critical section at the same time!
- To ensure that Peterson's solution will work correctly on modern computer architecture we must use **Memory Barrier**.

Memory Barrier

- **Memory model** are the memory guarantees a computer architecture makes to application programs.
- Memory models may be either:
  - **Strongly ordered** – where a memory modification of one processor is immediately visible to all other processors.
  - **Weakly ordered** – where a memory modification of one processor may not be immediately visible to all other processors.
- A **memory barrier** is an instruction that forces any change in memory to be propagated (made visible) to all other processors.

Memory Barrier Instructions

- When a memory barrier instruction is performed, the system ensures that all loads and stores are completed before any subsequent load or store operations are performed.
- Therefore, even if instructions were reordered, the memory barrier ensures that the store operations are completed in memory and visible to other processors before future load or store operations are performed.

Memory Barrier Example

- Returning to the example of slides 6.17 - 6.18
- We could add a memory barrier to the following instructions to ensure Thread 1 outputs 100:
- Thread 1 now performs

```
while (!flag)
  memory_barrier();
print x
```

- Thread 2 now performs

```
x = 100;
memory_barrier();
flag = true
```

- For Thread 1 we are guaranteed that that the value of `flag` is loaded before the value of `x`.
- For Thread 2 we ensure that the assignment to `x` occurs before the assignment `flag`.

Synchronization Hardware

- Many systems provide hardware support for implementing the critical section code.
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- We will look at three forms of hardware support:

1. Hardware instructions

2. Atomic variables

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** instruction
  - **Compare-and-Swap** instruction

The test_and_set Instruction

- Definition

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

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

Solution Using test_and_set()

- Shared boolean variable **`lock`**, initialized to **`false`**
- Solution:

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

              /* critical section */

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

- Does it solve the critical-section problem?

The compare_and_swap Instruction

- Definition

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

- Properties
  - Executed atomically
  - Returns the original value of passed parameter **`value`**
  - Set the variable **`value`** the value of the passed parameter **`new_value`** but only if **`*value == expected`** is true. That is, the swap takes place only under this condition.

Solution using compare_and_swap

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

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

  /* critical section */

  lock = 0;

  /* remainder section */
}
```

- Does it solve the critical-section problem?

Bounded-waiting with compare-and-swap

```
Lock = false;
Waiting[n]= false;

while (true) {
   waiting[i] = true;
   key = 1;
   while (waiting[i] && key == 1)
      key = compare_and_swap(&lock,0,1);
   waiting[i] = false;
   /* critical section */
   j = (i + 1) % n;
   while ((j != i) && !waiting[j])
      j = (j + 1) % n;
   if (j == i)
      lock = 0;
   else
      waiting[j] = false;
   /* remainder section */
}
```

Atomic Variables

- Typically, instructions such as compare-and-swap are used as building blocks for other synchronization tools.

- One tool is an **atomic variable** that provides *atomic* (uninterruptible) updates on basic data types such as integers and booleans.

Mutex Locks

- OS designers build **software tools** to solve critical section problem.
- Simplest is mutex lock
  - Boolean variable indicating if lock is available or not
- Protect a critical section by
  - First **acquire()** a lock
  - Then **release()** the lock
- Calls to **acquire()** and **release()** must be atomic
  - Usually implemented via hardware atomic instructions such as compare-and-swap.

- **Busy Waiting means busy**-looping or spinning is a technique in which a process repeatedly checks to see if a condition is true, such as whether keyboard input or a lock is available

- Mutex lock therefore called a **spinlock**

Solution to CS Problem Using Mutex Locks

```
while (true) {
      acquire lock

          critical section

      release lock

remainder section
}
```

Pthreads Synchronization – Mutex Lock

```
#include <pthread.h>
Pthread_mutex_t
mutex;

/* create the mutex lock */
Pthread_mutex_ init(&mutex, NULL);
```

1st Arg: Mutex initiliazer

2nd Arg: Attributes, NULL means no error checks will be performed

Pthreads Synchronization – Mutex Lock

- The mutex is acquired with the pthread_mutex_lock()
- and released pthread_mutex_unlock() functions.

- If the mutex lock is unavailable when pthread_ mutex_lock() is invoked, the calling thread is blocked until the owner invokes pthread_mutex_ unlock().

Semaphore

- **Semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent **system** such as a multiprogramming **operating system**.
- Synchronization tool that provides more sophisticated ways (than Mutex locks) for processes to synchronize their activities.

Semaphore

- Semaphore ***S*** – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - **`wait()`** and **`signal()`**
    - Originally called **`P()`** and **`V()`**

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

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

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

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

Semaphore (Cont.)

- **Counting semaphore** – integer value can range over an unrestricted domain.

- **Binary semaphore** – integer value can range only between 0 and 1
  - Same as a **mutex lock**

- Can implement a counting semaphore ***S*** as a binary semaphore
- With semaphores we can solve various synchronization problems like resource allocation, order of execution among processes.

Semaphore Usage

- consider two concurrently running processes: *P*1 with a statement *S*1 and *P*2 with a statement *S*2.

- It is required that *S*2 be executed only after *S*1 has completed. We can implement this scheme readily by letting *P*1and *P*2 share a common semaphore synch, initialized to 0.

Sem Synch = 0

P1:

S_1;

signal(synch)

;

P2:

wait(sync);

S_2;

- Because synch is initialized to 0, *P*2 will execute *S*2 only after *P*1 has invoked signal(synch), which is after statement *S*1 has been executed.

Semaphore Implementation

- Must guarantee that no two processes can execute the **`wait()`** and **`signal()`** on the same semaphore at the same time.
- Thus, the implementation becomes the critical section problem where the **`wait`** and **`signal`** code are placed in the critical section.
- Could now have **busy waiting** in critical section implementation
  - But implementation code is short
  - Little busy waiting if critical section rarely occupied.
- Note that applications may spend lots of time in critical sections and therefore this is not a good solution

Semaphore Implementation with no Busy waiting

- With each semaphore there is an associated waiting queue.

- Each entry in a waiting queue has two data items:
  - Value (of type integer)
  - Pointer to next record in the list.

- Two operations:
  - **block** – place the process invoking the operation on the appropriate waiting queue
  - **wakeup** – remove one of processes in the waiting queue and place it in the ready queue

Implementation with no Busy waiting (Cont.)

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

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

POSIX - Semaphores

```
#include<semaphore.h>
        Sem_t sem;
          /* Create the semaphore and initialize it to 1 */
                Sem_init(&sem, 0, 1);
```

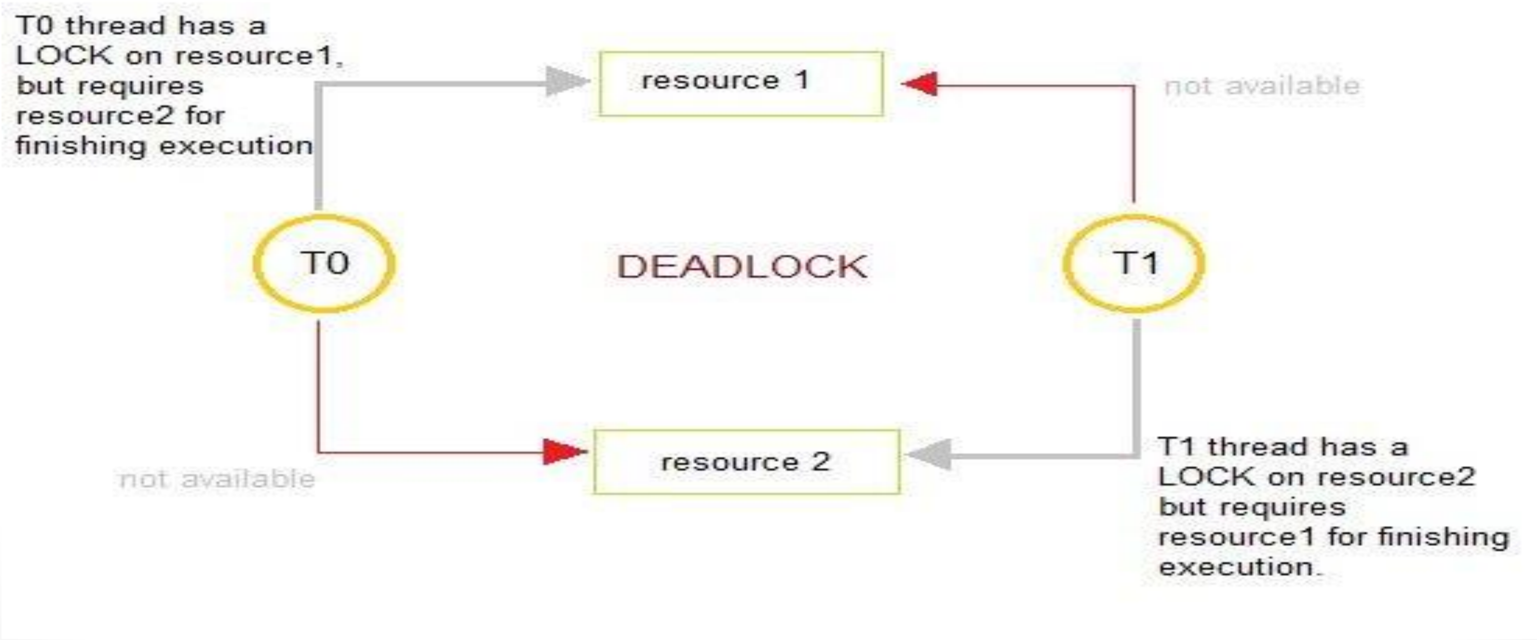
The sem_init() function is passed three parameters:

1. A pointer to the semaphore
2. A flag indicating the level of sharing
3. The semaphore's initial value

```
/* acquire the semaphore */
Sem_wait(&sem);

        /* critical section */

/* release the semaphore */
Sem_post(&sem);
```

Deadlock

- Deadlocks are a set of blocked processes each holding a resource and waiting to acquire a resource held by another process



Starvation

- **Starvation** is the name given to the indefinite postponement of a process because it requires some resource before it can run, but the resource, though available for allocation, is never allocated to this process.

Deadlock Vs. Starvation

- Deadlock refers to the situation when processes are stuck in circular waiting for the resources.

- On the other hand, starvation occurs when a process waits for a resource indefinitely.

- *Deadlock implies starvation but starvation does not imply deadlock*

Priority Inversion

- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Priority Inheritance Protocol

- when a job blocks one or more high-priority jobs, it ignores its original priority assignment and executes its critical section at an elevated priority level.

- After executing its critical section and releasing its locks, the process returns to its original priority level

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes

1. Bounded Buffer Problem
2. Dining-Philosophers Problem
3. Readers and Writers Problem

Bounded-Buffer Problem

- ***n*** buffers, each can hold one item
- Semaphore **`mutex`** initialized to the value 1
- Semaphore **`full`** initialized to the value 0
- Semaphore **`empty`** initialized to the value n

Bounded Buffer Problem (Cont.)

- The structure of the producer process

```
while (true) {
      ...
     /* produce an item in next_produced */
      ...
   wait(empty);
   wait(mutex);
       ...
     /* add next produced to the buffer */
       ...
   signal(mutex);
   signal(full);
}
```

Bounded Buffer Problem (Cont.)

- The structure of the consumer process

```
while (true) {
   wait(full);
   wait(mutex);
       ...
   /* remove an item from buffer to next_consumed */
       ...
   signal(mutex);
   signal(empty);
       ...
     /* consume the item in next consumed */
       ...
 }
```

Bounded Buffer Problem (Cont.)

- mutex = 1
- full = 2
- empty =3
- n= 5

100
101

```
wait(empty);
        wait(mutex);
            ...
       buffer[in] = add;
             in = in+1
            ...
        signal(mutex);
        signal(full);
```

```
wait(full);
        wait(mutex);
           ...
     Var A = buffer[out];
     out = out+1
           ...
        signal(mutex);
        signal(empty);
```

Readers-Writers Problem

- A data set is shared among a number of concurrent processes
  - **Readers** – only read the data set; they do ***not*** perform any updates
  - **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.
- Several variations of how readers and writers are considered – all involve some form of priorities

Readers-Writers Problem (Cont.)

- Shared Data
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1
  - Semaphore **`mutex`** initialized to 1
  - Integer **`read_count`** initialized to 0

  - Cases: RR, WW, RW, WR

Readers-Writers Problem (Cont.)

- The structure of a writer process

```
   while (true) {
        wait(rw_mutex);
             ...
        /* writing is performed */
             ...
      signal(rw_mutex);
 }
```

Readers-Writers Problem (Cont.)

- The structure of a reader process

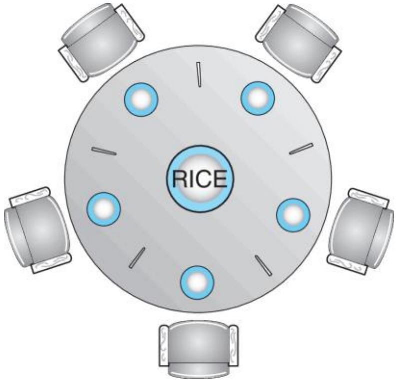
```
while (true){
        wait(mutex);
        read_count++;
        if (read_count == 1) /* first reader */
              wait(rw_mutex);
              signal(mutex);
         ...
        /* reading is performed */
         ...
        wait(mutex);
        read count--;
        if (read_count == 0) /* last reader */
                 signal(rw_mutex);
        signal(mutex);
}
```

Readers-Writers Problem Variations

- The solution in previous slide can result in a situation where a writer process never writes. It is referred to as the "First reader-writer" problem.
- The "Second reader-writer" problem is a variation the first reader-writer problem that state:
  - Once a writer is ready to write, no "newly arrived reader" is allowed to read.
- Both the first and second may result in starvation. leading to even more variations
- Problem is solved on some systems by kernel providing reader-writer locks

Dining-Philosophers Problem

- N philosophers' sit at a round table with a bowel of rice in the middle.



- They spend their lives alternating thinking and eating.
- They do not interact with their neighbors.
- Occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
  - Need both to eat, then release both when done
- In the case of 5 philosophers, the shared data
  - Bowl of rice (data set)
  - Semaphore chopstick [5] initialized to 1

Dining-Philosophers Solution 1

- When a philosopher wants to eat the rice, he will wait for the chopstick at his left and picks up that chopstick.

- Then he waits for the right chopstick to be available, and then picks it too. After eating, he puts both the chopsticks down

Dining-Philosophers Problem

- But if all five philosophers are hungry simultaneously, and each of them pickup one chopstick, then a deadlock situation occurs:

Dining-Philosophers Solution 2

- Two Possible solutions are :

- A philosopher must be allowed to pick up the chopsticks only if both the left and right chopsticks are available.

- Allow only four philosophers to sit at the table. That way, if all the four philosophers pick up four chopsticks, there will be one chopstick left on the table. So, one philosopher can start eating and eventually, two chopsticks will be available. In this way, deadlocks can be avoided.

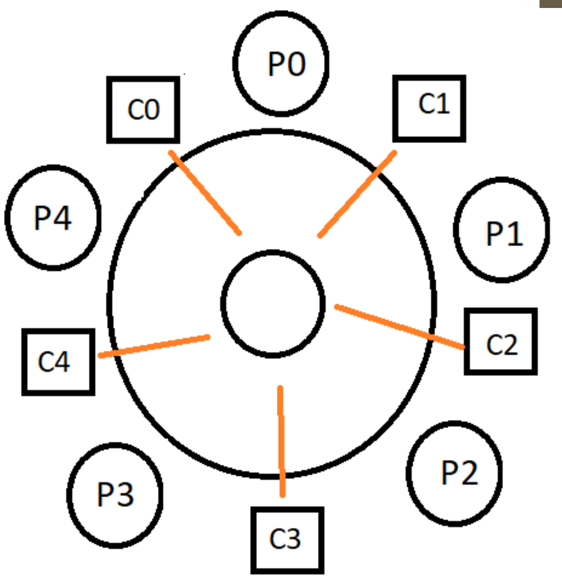
Dining-Philosophers Problem Algorithm

- Semaphore Solution
- The structure of Philosopher *i*:

```
while (true){

    wait (chopStick[ (i + 1) % 5] );
wait (chopstick[i] );

     /* eat for awhile */

    signal (chopstick[i] );
    signal (chopstick[ (i + 1) % 5] );

     /* think for awhile */


}
```

Dining-Philosophers Problem Algorithm

- Case 1: A Philosopher at a time
- Case 2: Two Consecutive arrivals
- Case 3: Context Switching and Preemption



Philosopher	Chop[i]	Chop[i+1]
Ph0	C0	C1
Ph1		
Ph2		
Ph3		
Ph4		

c0	c1	c2	c3	c4
1	1	1	1	1

Thank you