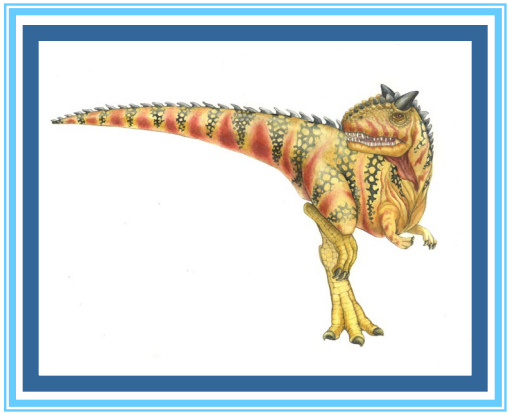
Chapter 6: Synchronization Tools



Introduction

- A **cooperating process** can affect or be affected by other processes executing in the system.
- **Cooperating processes** can either directly share a **logical address** space (both *code* and *data*) or be allowed to **share data** only through **shared memory** or **message passing**.
- **Concurrent access** to **shared data** may result in **data inconsistency**.
- This chapter discusses various **mechanisms to ensure the orderly execution of cooperating processes that share a logical address space**, so that data consistency is maintained.

Background

- We have seen that processes can execute **concurrently** or in **parallel**.
  - A **single CPU scheduler** switches rapidly between processes to provide **concurrent execution**.
  - This is possible by a process that may be interrupted at any point in its instruction stream, and the processing core may be assigned to execute instructions of another process.
- Also noticed how a **parallel execution by a multi-core system** happens:
  - In which multiple instruction streams (representing **threads** from *processes*) execute simultaneously on **separate CPU cores**.
- This chapter explains how **concurrent or parallel execution can contribute to issues involving the integrity of data shared by several processes.**

Background

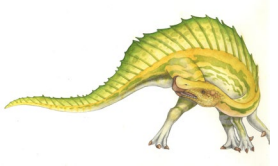
- This section describes how **concurrent access to shared data** may result in **data inconsistency**
  - Maintaining **data consistency** requires mechanisms to ensure the *orderly execution of cooperating processes*
    - **Illustration of the problem:**
      Suppose that we wanted to provide a solution to the **consumer-producer problem** that fills **all the buffers.**
    - an integer `counter` that keeps track of the number of **full buffers**.
    - Initially, the `counter` is set to **0**.
    - The **producer** increments it after it produces a new buffer location, and the **consumer** decrements it after it consumes a **buffer**.

Producer

```
while (true) {
     /* produce an item in next
produced */

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

Consumer

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



Race Condition

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

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

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

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

Race Condition

The concurrent execution of "`counter++`" and "`counter--`" is equivalent to a sequential execution in which the lower-level statements presented previously are interleaved in some arbitrary order (but the order within each high-level statement is preserved). One such interleaving is the following:

T_0:	*producer*	execute	$register_1 = \texttt{counter}$	$\{register_1 = 5\}$
T_1:	*producer*	execute	$register_1 = register_1 + 1$	$\{register_1 = 6\}$
T_2:	*consumer*	execute	$register_2 = \texttt{counter}$	$\{register_2 = 5\}$
T_3:	*consumer*	execute	$register_2 = register_2 - 1$	$\{register_2 = 4\}$
T_4:	*producer*	execute	$\texttt{counter} = register_1$	$\{counter = 6\}$
T_5:	*consumer*	execute	$\texttt{counter} = register_2$	$\{counter = 4\}$

Notice that we have arrived at the incorrect state "`counter` == 4", indicating that four buffers are full, when, in fact, five buffers are full. If we reversed the order of the statements at T_4 and T_5, we would arrive at the incorrect state "`counter` == 6".

Producer-Consumer Problem: Race condition

- We would arrive at this incorrect state because we allowed both processes to manipulate the **variable counter concurrently.**
- A situation like this, where 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.**
- In a **concurrent system**, to guard against the **race condition,** we need to ensure that only one process at a time can be manipulating the variable counter.

Critical Section (CS)

- A **critical section** (**CS**) is a *code segment* in a process in which a **shared resource is allowed to be accessed**:
  - Only one process can execute its CS at any one time
  - When no process is executing in its CS, any process that requests entry to its CS must be permitted without delay
  - When two or more processes compete to enter their *respective* CSs, the selection cannot be postponed indefinitely
  - No process can prevent any other process from entering its CS indefinitely
  - Every process should be given a **fair chance** to access the shared resource through CS

CS Problem

- Consider system of ***n*** processes $\{p_0, p_1, \ldots p_{n-1}\}$
- Each process has **CS** segment
  - Process may be changing common variables, updating table, writing file, etc
  - When one process in **CS**, no other may be in its **CS**
  - Each process must ask permission to enter CS in **entry section**, may follow critical section with **exit section**, then **remainder section.**
  - **The general structure of** a typical process *Pi* is shown in **Figure 5.1**.
  - **No two processes are executing in their critical sections at the same time**.

CS

- General structure of process P_i

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

Figure 5.1 General structure of a typical process P_i.

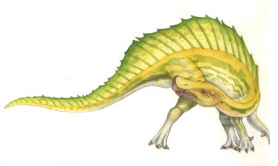
Solution to CS Problem

1. **Mutual Exclusion** - If process P_i is executing in its **critical section** (**CS**), then no other processes can be executing in their **critical sections**
2. **Progress** - If no process is executing in its **CS** and there exist some processes that wish to enter their CS, then the selection of the processes that will enter the **CS** next cannot be postponed indefinitely
3. **Bounded Waiting** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
   - Assume that each process executes at a nonzero speed
   - No assumption concerning **relative speed** of the ***n*** processes

CS Handling in OS

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

- **Preemptive** – allows preemption of process when running in kernel mode
  - **Preemptive kernels** are especially difficult to design for **SMP** architectures, since in these environments it is possible for two kernel-mode processes to run simultaneously on different processors.
- **Non-preemptive** – runs until exits kernel mode, blocks, or voluntarily yields CPU
  - Essentially free of race conditions in kernel mode, as only one process is active in the kernel at a time.

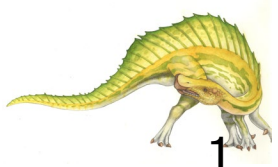


Critical-Section Handling in OS

- **Why, then, would anyone favor a preemptive kernel over a non-preemptive one?**
  - A **preemptive kernel** may be less risk that a **kernel-mode** process can run for an arbitrarily long period before relinquishing it by the processor.
  - A **preemptive kernel** is more suitable for real-time programming, as it will allow a real-time process to preempt a process currently running in the kernel.

Critical-Section Problem: Peterson's Solution

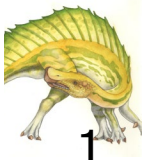
- **Peterson's solution to CS problem** is restricted to two processes that alternate execution between their **critical sections** and **remainder sections**.
  - The processes are numbered P_i and P_j (where $i = 0$ and $j = 1$)
  - **Figure 5.2** The structure of process in Peterson's solution.
  - **int turn;**
    - The integer variable ***turn*** indicates whose turn it is to enter its CS.
      - if *turn* == i, then process *Pi is allowed to execute in its CS*
  - **boolean flag[2];**
    - A **size two array** is used to indicate if a process is ready to enter its CS.
    - For example, if flag[i] is true, this value indicates that *Pi is ready to enter* its critical section.



Critical-Section Problem: Peterson's Solution

```
do {

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

        critical section

    flag[i] = false;
}
        remainder section

} while (true);
```

Figure 5.2 The structure of process P_i in Peterson's solution.



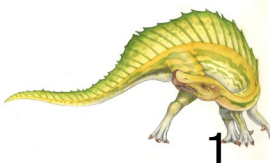
Critical-Section Problem: Peterson's Solution

- Each P_i enters its CS only if either (flag[j] == *false*) or (turn == P_i), where P_i = 0
- If (flag[i] == flag[j] == *true*), then it imply that P_i *and* P_j *could not have* successfully executed their while statements at about the same time, since the value of **turn** can be either 0 or 1 but cannot be both. **So this situation is not valid!**
- If (flag[j] == *true* and turn == P_j), and this condition will cause P_j *is in its critical section*, where P_j = 1
- If P_j is not ready to enter its CS, then (flag[j] == *false*), and P_i can enter its CS.
- Mutual exclusion is preserved

  P_i enters CS only if: either `flag[j] = false` or `turn = i`

Simulation

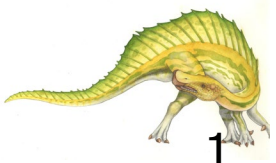
Code in C++



Critical Section

- General structure of process P_i

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



Solution to CS: mutual exclusion locks

- OS designers build software tools to solve **CS** problem, simplest is **mutex lock** (called **mutual exclusion lock**) to protect critical regions and thus prevent **race conditions**.
- That is, a process must **acquire the lock** before entering a **CS**; it **releases the lock** when it exits the **CS**.
  - Protect a **CS** by first **acquire()** a lock then **release()** the lock;

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

Mutex Locks

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

Figure 5.8 Solution to the critical-section problem using mutex locks.

The definition of `release()` is as follows:

```
release() {
   available = true;
}
```

Calls to either `acquire()` or `release()` must be performed atomically.

Disadvantage of Mutex Locks

- The **main disadvantage** of **mutex lock** is that it requires **busy waiting** (**spinlock**)
  - **Spinlock** means processors continuously execute the ***atomic instruction*** to check for the status of the **shared variable**
    - It wastes processor cycles and consumes network bandwidth
- While a process is in its **CS**, any other process that tries to enter its **CS** must loop continuously in the call to **acquire().**
- Calls to **`acquire()`** and **`release()`** must be **atomic**
  - But this solution requires **busy waiting**
    - This lock therefore called a **spinlock**

Atomic hardware instructions for Critical Section

- **Atomic HW instructions for CS through mutex are:**
  - **Test-and-Set ()** instruction
  - **Compare-and-Swap()** Instruction

Atomic hardware instructions for Critical Section (mutex)

- **Test-and-Set ()** instruction
  - "Test" the requested process is sanctioned to enter CS by checking a memory location or a Boolean variable
  - If yes (the memory location match), then "Set" the selected process for CS
- The **Test-and -Set()** instruction can be defined as shown in **Figure 5.3.**
  - If the machine supports the **test and set()** instruction, then we can implement **mutex** by declaring a boolean variable **lock**, initialized to *false*.

test and set() instruction

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

   return rv;
}
```

Figure 5.3 The definition of the `test_and_set()` instruction.

1.Executed atomically
2.Returns the original value of passed parameter **target* and set it as *true*

compare_and_swap Instruction

```
int compare _and_swap(int *value, int
                      expected, int new_value)
{
        int temp = *value;

        if (*value == expected)
           *value = new_value;
     return temp;
    }
```

1. Executed atomically
2. Returns the original value of passed parameter **"*value**" – a **shared variable**
3. If ***value** and **expected** are same, then assigns **new_value** to ***value** and return ***value.** That is, the swap takes place only under this condition.

Semaphores

- **Mutex locks**, are generally considered the simplest of *process synchronization* tools.
- A **semaphore** is a more robust synchronization tool that provides more sophisticated ways (than Mutex locks) for *process synchronization*.
- A *semaphore* ***S*** *is an integer variable* that, apart from initialization, is accessed only through two standard atomic operations:
  - **wait()**
  - **signal()**

Semaphores

- The **wait()** operation was originally termed **P** (“*to test*”).
- The definition of **wait()** is as follows:

  P(S) or wait(S): send a wait to waiting processes until the S access operation (S = S - 1)by the current process(s) has been completed.

- The **signal()** was originally called **V** (“*to increment* ”).
- The definition of **signal()** is as follows:

  V(S) or signal(S): Signal a waiting process if the previous process has been released all S by incrementing (S = S + 1)it.

Semaphores

The definition of `wait()` is as follows:

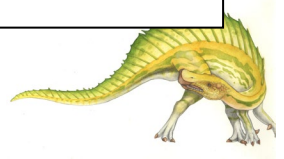
```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

The definition of `signal()` is as follows:

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

Semaphores

```
P(S)
        Semaphore S; // initialize resource size S
        {
          while (S > 0) ;  // wait until  resource accessed
          S = S - 1; // resource access
        }
V(S)
        Semaphore S; // initialize resource size S
        {
          S = S+1; // free resources by processes
        }
Init(S, v)  // initialize S with v values before a new request
        Semaphore S; // initialize resource size S
        Int v;
        {
        S = v;
         }
```



Semaphores

- All modifications to the integer value of the semaphore in the **wait()** and **signal()** operations must be executed **indivisibly**.
  - That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value.
- In addition, in the case of **wait(S)**, the testing of the integer value of **S** (**S ≤ 0**), as well as its possible modification (**S--**), must be executed **without interruption**.

Semaphore Usage

- Operating systems often distinguish between **counting** and **binary** semaphores.
- The value of a **counting semaphore** can range over an unrestricted domain.
- The value of a **binary semaphore** can range only between 0 and 1.
  - Thus, binary semaphores behave similarly to **mutex locks**.
    - on systems that do not provide **mutex locks**, binary semaphores can be used instead for providing mutual exclusion.

Semaphores

- Use **semaphores** to solve **synchronization problems**: For example, consider two **concurrently** running processes:
  - ***P*1** with a statement ***S*1** and ***P*2** with a statement ***S*2**. Suppose we require that ***S*2** be executed only after ***S*1** has completed.
  - We can implement this scheme readily by letting ***P*1** and ***P*2** share a common semaphore **synch**, initialized with its size.
  - In process ***P*1**, we insert the statement

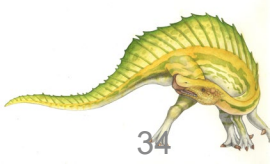
```
S1;
wait(synch); // P1 using semaphore synch (P1
                in CS) and P2 should wait
```

  - In process *P*2, we insert the statements:

```
signal(synch); // Signal to P2 indicates P1 has
                  completed its semaphore synch
                  (means, its CS).
```

Semaphores

- As shown in **Table 6.3**, ***P*3** is placed in the **WAIT** state (for the semaphore) on **State 4**:
  - As shown in **Table 6.3**, for **States 6** and **8**, when a process exits the **CS**, the value of ***s*** is ***reset* to 1,** indicating that the **CS** is free.
  - This, in turn, triggers the awakening of one of the blocked processes, its entry into the **CS**, and the resetting of ***s*** to **0**.
  - In **State7**, ***P*1** and ***P*2** are not trying to do processing in that **CS** and ***P*4** is still **blocked**.



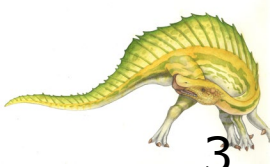
Semaphores

State Number	Actions Calling Process	Operation	Running in Critical Region	Results Blocked on *s*	Value of *s*
0					1 (CS available)
1	P_1	test(*s*)	P_1 (P1 In CS)	No CS wait	0 (CS not available)
2	P_1	increment(*s*)	P1 exits CS	No CS wait	1 (CS available)
3	P_2	test(*s*)	P_2 (P2 In CS)	No CS wait	0 (CS not available)
4	P_3	test(*s*)	P_2 (P2 In CS)	P_3	0 (CS not available)
5	P_4	test(*s*)	P_2 (P2 In CS)	P_3, P_4	0 (CS not available)
6	P_2	increment(*s*)	P_3 (P3 In CS)	P_4	0 (CS not available)
7			P_3 P3 In CS)	P_4	0 (CS not available)
8	P_3	increment(*s*)	P_4 (P4 In CS)	No CS wait	0 (CS not available)
9	P_4	increment(*s*)	(P4 exits from CS)	No CS wait	1 (CS available)

(table 6.3) *The sequence of states for four processes calling test and increment (P and V) operations on the binary semaphore s. (Note: The value of the semaphore before the operation is shown on the line preceding the operation. The current value is on the same line.)*

Semaphores

- **Advantages**
  - Semaphores impose deliberate constraints that help programmers to avoid errors
  - Solutions using semaphores are often clean and organized
  - Semaphore can be implemented efficiently on many systems, so solutions that use semaphores are portable and usually efficient



Semaphores

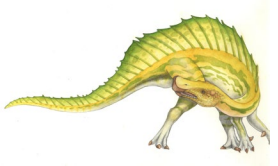
- **Drawbacks**
  - A process that uses a semaphore has to know which other processes use the semaphore (or waiting for semaphore)
    - the semaphore operations of all interacting processes have to be coordinated
  - Semaphore operations must be carefully positioned in a process
    - The omission of a P or V operation may result in deadlock or inconsistencies
  - Programs with semaphores extremely hard to verify for their correctness

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

P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
`...`	`...`
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

- Suppose that ***P*0** executes **wait(S)** and then ***P*1** executes **wait(Q)**. When ***P*0** executes **wait(Q)**, it must wait until ***P*1** executes **signal(Q)**.
- Similarly, when ***P*1** executes **wait(S)**, it must wait until ***P*0** executes **signal(S)**. Since these **signal()** operations cannot be executed, ***P*0** and ***P*1** are **deadlocked**.



Deadlock and Starvation

- **Starvation – indefinite blocking**
  - A process may never be removed from the semaphore queue in which it is suspended
- **Priority Inversion** – Scheduling problem when lower-priority process holds a lock needed by higher-priority process
  - Solved via **priority-inheritance protocol**

Classical Problems of Synchronization

- Classical problems used to test newly-proposed synchronization schemes
  - **Bounded-Buffer Problem**
  - **Readers and Writers Problem**
  - **Dining-Philosophers Problem**

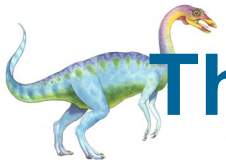
Bounded-Buffer Problem

- ***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

The Producer-Consumer - Semaphore

- The structure of the **producer process**

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



The Producer-Consumer - Semaphore

- The structure of the **consumer process**

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

The Producer-Consumer - Semaphore

- The structure of the **producer process**

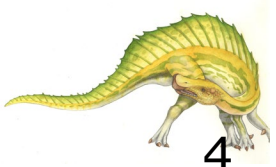
```
do {
    // produce an item
    wait (empty); // consumer should wait because buffer is empty
    wait (mutex); // consumer wait, mutex semaphore is with producer
    add the item to the buffer; // fill buffer (CS)
    signal (mutex); // signal to consumer that 'mutex' is free and get it
    signal (full); // signal to consumer that buffer is full
} while (TRUE);
```

- The structure of the **consumer process**

```
do {
    wait (full); // producer wait because buffer is full
    wait (mutex); // producer wait 'mutex' is with consumer
    remove an item from buffer; // consume buffer (CS)
    signal (mutex); // signal to producer that mutex is free and get it
    signal (empty); // signal to producer that buffer is empty
    } while (TRUE);
```

The Producer-Consumer problem

- In multithreaded programs there is often a division of **labor** between processes (or threads)
- And some processes are **producers** and some are **consumers**
  - **Producers** create **items** of some kind and add them to a **data structure (buffer);**
  - **Consumers** remove (consume) the items from **buffer** and process them



Producer-Consumer Problem

- There are several *synchronization constraints* that we need to enforce to make this **producer-consumer** system work correctly
  - **Producer** processes (or threads) supplies/produce messages
  - **Consumer** processes (or threads) consume messages
  - Both processes (*asynchronous in nature*) share a **common buffer**
  - conditional synchronization as well as mutual exclusion needed:
    - No **consumer** can access the buffer **when it is empty** and no producer can access the buffer **when it is full**
    - If a **consumer** process (or thread) arrives while the **buffer** is empty, it blocks until a producer adds a new item into it

Producer-Consumer Problem

- **Event-driven** programs are a good example:
  - Whenever and **event** occurs, a **producer thread** creates an **event object** and adds it to the **event buffer**
  - Concurrently, **consumer threads** take **events** out of the **buffer** and execute them
  - In this case, the **consumers** are called "**event handlers**"

Readers-Writers Problem

- Here 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* and *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
- **Shared Data**
  - Data set
  - Semaphore **`rw_mutex`** initialized to 1 // *this one for writes*
  - Semaphore **`mutex`** initialized to 1 // both readers and writers needed
  - Integer **`read_count`** initialized to 0

Readers-Writers Problem (Cont.)

- The structure of a **writer process**

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

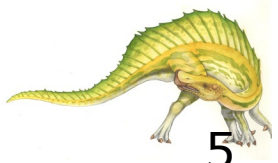
Readers-Writers Problem (Cont.)

- The structure of **a reader process**

```
do {  //  first finding readers using 'mutex'
      wait(mutex);  // mutex is with a reader other readers/writers wait
         read_count++;  // get readers
         if (read_count == 1)  //if at least one reader
          wait(rw_mutex);  // send  wait (rw_mutex) writers
       signal(mutex);  // send signal (mutex) next waiting reader

         /* reading is performed */
      wait(mutex);  // reading is going on (readers/writers waits)
         read count--;  // get count of read readers
         if (read_count == 0)  // if no more readers
       signal(rw_mutex);  // signal 'rw_mutex' to writers to synch
       signal(mutex);  // signal 'mutex' to synchronized writers
 } while (true);
```

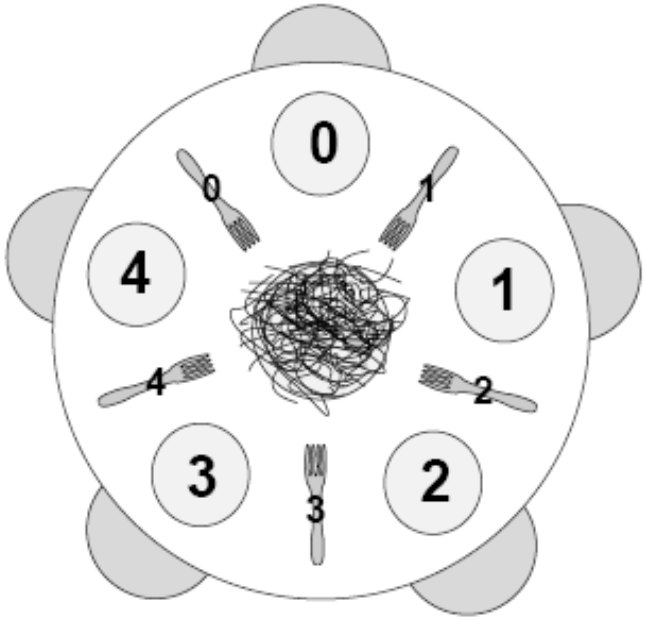
The Dining Philosophers (cont.)

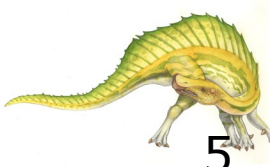
- **Five philosophers** are sitting in a circle, attempting to eat spaghetti with the help of forks
- A common bowl of spaghetti for each philosopher but there are only **five forks** (forks are placed between left and right of each philosopher) to share among them
- But both forks are needed at a time to consume spaghetti
- A philosopher alternates between **two phases:**
  - **Thinking** and **eating**
- In the **thinking mode**, a philosopher does not hold a fork
- When hungry, a philosopher attempts to pick up both forks on left and right sides
  - A philosopher can start eating only after **obtaining both forks**
  - Once start eating the **forks are not relinquished** until the eating phase is over



The Dining Philosophers (cont.)

- Note that **no two neighboring philosophers can eat simultaneously**
- In order to find any solution, *the act of picking up a fork by a philosopher must be a* ***critical section***
- The solution should be a *deadlock free one*, in which no philosopher starves





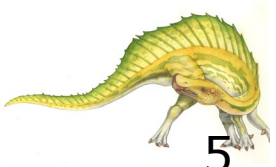
The Dining Philosophers (cont.)

- Five philosophers, who represent interacting threads, come to the table and execute the following loop:

```
while true
    think()
    get_forks()
    eat()
    put_forks()
```

The program should satisfy the following constraints:

1. Only one philosopher can hold a fork at a time.
2. It must be impossible for a deadlock to occur.
3. It must be impossible for a philosopher to starve waiting for a fork.
4. It must be possible for more that one philosopher to eat at the same time.

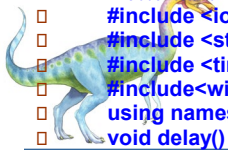


Windows Synchronization

- Windows uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses **spinlocks** on multiprocessor systems
  - **Spinlocking-thread** will never be preempted
- Also provides **dispatcher objects** user-land which may act as mutexes, semaphores, events, and timers
  - **Events**
    - An event acts much like a condition variable
  - Timers notify one or more thread when time expired
  - Dispatcher objects either **signaled-state** (object available) or **non-signaled state** (thread will block)

Linux Synchronization

- **Linux:**
  - Prior to kernel Version 2.6, disables interrupts to implement short critical sections
  - Version 2.6 and later, fully preemptive
- **Linux provides:**
  - Semaphores
  - atomic integers
  - spinlocks
  - reader-writer versions of both
- On single CPU system, **spinlocks** replaced by enabling and disabling kernel preemption.



```
#include <cstdlib>
#include <iostream>
#include <string>
#include <time.h>
#include<windows.h>
using namespace std;
void delay()
{ time_t start_time, cur_time;
  time(&start_time);
  do{ time(&cur_time);}
  while((cur_time - start_time) < 2.5); // 2.5 ms delay
}
int main(int argc, char *argv[])
{
    bool flag[2]= {false, false};
    int turn;
    int count = 1;
    int bount = 1;
    string turnVal;
    do{
       system("CLS");
        srand (time(NULL));
        int ran = rand() % 2; // generate 0 = i or 1 = j
       flag[ran] = true;// i == true
       turn = ran; //j = 1
    while(flag[ran] && turn == ran)// while both flag and turn are same then
      {
          cout << "\n\n";
           for (int i = 0; i < 2; i++)
          { if (i == 0 && flag[i] == 1) {cout << "  flag[Pi] = true  " ;}
           else if(i == 0 && flag[i] == 0){cout << "  flag[Pi] = false "  ;}
           else if(i == 1 && flag[i] == 1){cout << "  flag[Pj] =  true  ";}
           else {cout << "  flag[Pj] =  false  ";}}//for ends
           if(turn == 0)turnVal = " Pi ( value is 0) ";
           else turnVal = " Pj (value is 1) ";
           cout << "   turn = " << turnVal <<  endl;
    if(ran == 0) // print the equivalent process critical status
    { if (count == 1)
    {Beep(1568, 300); //cout << '\a';// window beep
    count = 0;}
    cout << "\n ==========================================\n";
    cout << " \n  Process Pi is in its Critical Section (CS).\n";
    cout << "\n ==========================================\n" ;
    bount = 1;}
    else { if(bount == 1){
     Beep(1275, 400); //cout << '\a';// window beep
    bount = 0;}
    cout << "\n ==========================================\n";
    cout << " \n  Process Pj is in its Critical Section (CS).\n";
    cout << "\n ==========================================\n" ;
    count = 1;
    }
    flag[ran] = false; // clear the flag status for next process
    delay();// call 2 ms delay
    }
}while(true);
    system("PAUSE");
    return EXIT_SUCCESS;
}
```