Inter-process Communication and Synchronization

Contents

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

Producer-Consumer Problem (PCP)

- Paradigm for cooperating processes
- *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.*
- Basic synchronization requirement
  - Producer should not write into a full buffer
  - Consumer should not read from an empty buffer
  - All data written by the producer must be read exactly once by the consumer

Bounded-Buffer – Shared-Memory Solution to PCP

- The shared buffer is implemented as a circular array with two pointers: in and out (in: points to next free position in the buffer; out: points to first full position in the buffer)
- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```


Bounded-Buffer: Producer Process

```
item nextProduced;
while (1) {
  /* Produce an item in nextProduced  */
      while (((in + 1) % BUFFER SIZE)   ==  out)
            ;  /* do nothing */
     buffer[in] = nextProduced;
    in = (in + 1) % BUFFER SIZE;
   }
```

The producer process

Bounded-Buffer: Consumer Process

```
item nextConsumed;
while (1) {
      while (in == out)
           ; // do nothing -- nothing to consume

           // remove an item from the buffer
      nextConsumed = buffer[out];
      out = (out + 1) % BUFFER SIZE;
     /* consume the item in nextConsumed  */
   }
```

The consumer process

Bounded-Buffer – Shared-Memory Solution to PCP

- This solution allows at most BUFFER_SIZE-1 items in the buffer (of size BUFFER_SIZE) at the same time. A solution, where BUFFER_SIZE items can be in the buffer *is not simple*
- Suppose we modify the producer-consumer code by adding a variable *counter*, *initialized to* 0 and incremented each time a new item is added to the buffer

Bounded-Buffer for BUFFER_SIZE items

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```


Bounded-Buffer for BUFFER_SIZE items

- **Producer process**

```
item nextProduced;

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

- **Consumer process**

```
item nextConsumed;

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


The Problem with this solution

- The statement "**counter++**" may be implemented in machine language as:

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

- The statement "**counter--**" may be implemented as:

```
register2 = counter
register2 = register2 – 1
counter = register2
```


The Problem with this solution

- If both the producer and consumer attempt to update *counter concurrently, the assembly* language statements may get interleaved.
- Interleaving depends upon how the producer and consumer processes are scheduled

The Problem with this solution

- Assume **counter** is initially 5. One interleaving of statements is:

  T_0: producer: **register1 = counter** (*register1 = 5*)
  T_1: producer: **register1 = register1 + 1** (*register1 = 6*)
  T_2: consumer: **register2 = counter** (*register2 = 5*)
  T_3: consumer: **register2 = register2 – 1** (*register2 = 4*)
  T_4: producer: **counter = register1** (*counter = 6*)
  T_5: consumer: **counter = register2** (*counter = 4*)

- The value of **count** may be either 4 or 6, where the correct result should be 5.

Race Condition

- A scenario in which several processes may read and write shared data concurrently and the final value of the shared data depends upon which process finishes last
  - So the final output is dependent on the relative speed of the processes
- Race conditions must be prevented
  - concurrent processes must be synchronized

Atomic Operation

- An operation that is either executed fully without interruption, or not executed at all
  - The "operation" can be a group of instructions
  - Ex. the instructions for *counter++ and counter--*
- Note that the producer-consumer problem's solution works if *counter++ and counter-- are made atomic*
- In practice, the process may be interrupted in the middle of an atomic operation, but the atomicity should ensure that no process uses the effect of the partially executed operation until it is completed

The Critical-Section Problem

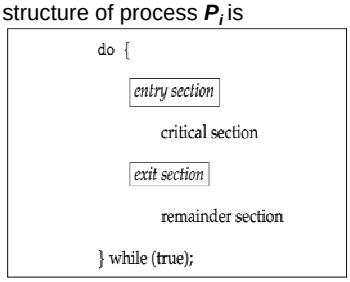
- *n* processes all competing to use some shared data
- Each process has a code segment, called *critical section*, in which the shared data is accessed.
- Critical-Section Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

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 processes that will enter the critical section next cannot be postponed indefinitely.
3. **Bounded Waiting/No Starvation**. A bound (or limit) 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.

Critical Section

- Entry section: a piece of code executed by a process just before entering a critical section
- Exit section: a piece of code executed by a process just after leaving a critical section
- The remaining code is the *remainder section*
- General structure of process P_i is



Peterson's Solution

- The Peterson's solution restricted to two processes P_0 and P_1 that alternate execution between their critical sections and remainder sections.
- The two processes P_i and P_j (j =1-i) share two variables:
  - int turn == 0;
  - boolean flag[2]
- The variable turn indicates whose turn it is to enter the critical section. The variable turn is set to either 0 or 1 randomly (or it can always be set to 0)
  - turn == i implies P_i's turn to enter its critical section
- The flag array is used to indicate if a process is ready to enter the critical section.
  - initially, flag[0] == flag[1] == false
  - flag[i] == true implies that P_i is ready to enter its critical section

Algorithm for Process P_i *in Peterson's solution*

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

- Provethat that
  1. Mutual exclusion is preserved
  2. Progress requirement is satisfied
  3. Bounded-waiting requirement is met

Synchronization Hardware

- Many systems provide hardware support for critical section code
- A process must acquire lock before entering a critical section; it releases the lock when it exits the critical section

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


Synchronization Hardware

- The critical-section problem could be solved in a uniprocessor environment if we could prevent interrupts from occurring while shared variable was being modified
- Unfortunately, this solution is not as feasible in a multiprocessor environment
  - Disabling interrupts on a multiprocessor can be time consuming, as the message passed to all the processors
  - Message passing delays entry into each critical section, & system efficiency decreases
  - The effect on system's clock if the clock is kept updated by interrrupts

Hardware Instruction Based Solutions

- TestAndSet:Test and modify the content of a word atomically
- The TestAndSet() instruction is defined below:

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


Mutual-exclusion with TestAndSet()

- Shared data:

  boolean lock = false
- Process P_i

```
do {
    while(TestAndSet(&lock));
        critical section
    lock = FALSE;
        remainder section
} while(TRUE);
```

Swap() instruction

- *Swap: Atomically swap content of two words.*
- *The Swap() instruction is defined as follows:*

```
void Swap(boolean *a, boolean *b)
  {
    boolean temp = *a;
   *a = *b;
   *b = temp;
  }
```


Mutual-exclusion with swap() instruction

- Uses two Boolean variables lock and Key
- If lock == false then a process can enter the critical section, otherwise it can't
- Initially lock == false
- Process P_i

```
do {
        key = TRUE;
        while ( key == TRUE)
              Swap (&lock, &key );
           critical section
         lock = FALSE;
           remainder section
 } while (TRUE);
```


Bounded-waiting mutual exclusion with TestAndSet()

- Shared data (initialized to false)
  - boolean waiting[n];
  - boolean lock;
- We present another algorithm using TestAndSet() that satisfies all the critical-section requirements

Bounded-waiting Mutual Exclusion with TestAndSet ()

```
do {
    waiting[i] = TRUE;
    key = TRUE;
    while (waiting[i] && key)
        key = TestAndSet(&lock);/ /only first lock==false will set key=false
    waiting[i] = FALSE;
        critical section
    j = (i + 1) % n;        //look for the next P[j] waiting: bound- waiting req.
    while ((j != i) && !waiting[j])
        j = (j + 1) % n;
    if (j == i)
      lock = false;
    else
     waiting[j] = FALSE; //wakeup only one process P[j] without releasing lock
        remainder section
} while (TRUE);
```

Bounded-waiting mutual exclusion with TestAndSet()

- Mutual exclusion
  - Mutual exclusion requirement is met if either waiting[i] == false or key == false
- Progress
  - Exit process sends a process in
- Bounded waiting
  - Any process wait at most n-1 times to enter critical section
- Atomic TestAndSet is hard to implement in multiprocessor environment

Semaphores

- The hardware-based solutions to the critical-section problem are complicated for application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore
- Semaphores
  - A variable used for signaling between the process
  - Originally semaphores were flags for signaling between ships
- Semaphore *S* – integer variable
- Two standard operations modify *S:*
  - wait() (or acquire)
  - signal() (or release)
- These two operations are indivisible (**atomic**)

Semaphore Operations

- The definition of wait () is as follows

```
wait (S) {
        while (S <= 0)
                ; // no-op, called busy waiting, spinlock
         S--;
}
```

- The definition of signal() is as follows:

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

- 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
- Spinlock: the process "spins" while waiting for the lock
- Must guarantee that no two processes can execute wait () and signal () on the same semaphore at the same time

Semaphore

- **Counting semaphore** – integer value can range over an unrestricted domain
- **Binary semaphore** – integer value can range only between 0 and 1; can be simpler to implement
  - Also known as **mutex locks (are the locks that provide mutual exclusion)**
- Can implement a counting semaphore S as a binary semaphore

Mutual-exclusion with semaphores for n Proceses

- Let the n processes share a semaphore, mutex, initialized to1
- Process P_i

```
do {
    wait (mutex);
        // Critical Section
     signal (mutex);
      // remainder section
} while (TRUE);
```


Semaphore Usage

- 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
- Let P_1 and P_2 share a common semaphore synch, initialized to zero.

```
P1:
    S1;
    signal (synch);
P2:
    wait (synch);
    S2;
```

- **Process synchronization**: S_2 in P_2 should be executed after S_1 in P_1

Semaphore Implementation with no Busy waiting

- Define semaphore as a “C” structure

```
typedef  struct {
                    int value;
                    struct  process *list;
                  } semaphore;
```

- Two operations:
  - **The block** () operation suspends the process that invokes it.
  - **The wakeup(P)** operation resumes the execution of a blocked process P.

Semaphore Implementation with no Busy waiting

- The wait() operation is now defined as

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

- The signal() operation is now defined as

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

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
  - example: process P_0 waits for process P_1, and process P_1 waits for process P_0
- Consider two processes, P_0 and P_1, each accessing 2 semaphores, ***S*** and ***Q*** initialized to 1

P_0	P_1
wait(S); //exec 1st	**wait(Q);** //exec 2nd
wait(Q); //exec 3rd	**wait(S);** //exec 4th
...	**...**
signal(S);	**signal(Q);**
signal(Q);	**signal(S);**

P_0 is waiting for P_1 to execute signal (Q)

P_1 is waiting for P_0 to execute signal (S)

Both Processes P_0 and P_1 are in a deadlock

- **Starvation** or **indefinite blocking**
  - A situation in which processes wait indefinitely within the semaphore. It may occur if we remove processes in LIFO order

Priority Inversion Problem

- Scheduling challenge arises when lower-priority process holds a lock needed by higher-priority process
- Scenario :
  - Processes with priorities **L** < **M** < **H** request resource *R*
  - **L** first locks on *R* ; then **H** requests *R*, but normally H would wait for L to finish using resource R.
  - In the meantime, **M** requests *R* and preempts **L**
  - **H** is waiting longer for lower-priority processes
- Not a problem with only two levels of priorities, but two levels are insufficient for most OSs
- Solved via **priority-inheritance protocol**
  - All lower-priority processes that are accessing resources requested by a higher-priority process, inherits the higher-priority level until it releases the resource

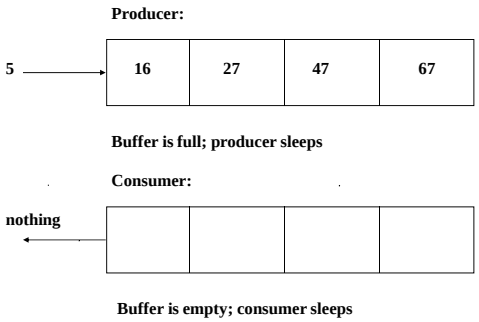
Classical Problems of Synchronization

- Bounded-Buffer Producer-Consumer Problem
- Readers and Writers Problem
- Dining-Philosophers Problem

Bounded-Buffer Producer-Consumer Problem

- Two processes (one producer, one consumer) share a common, fixed-size buffer
- Producer places information into the buffer
- Consumer takes it out
- The producer wants to place a new item into the buffer, but the buffer is already full
  - Solution: producer goes to sleep, wakes up when the consumer has emptied one or more items
- Consumer wants to consume an item, but the buffer is empty
  - Solution: consumer goes to sleep, wakes up when the producer has produced items

Producer Consumer Problem



6

Bounded-Buffer Producer-Consumer Problem

- The binary semaphore mutex provides mutually exclusive access to the buffer & is initialized to the value 1
- The counting semaphore full used for counting the number of slots that are full & is initialized to 0
- The counting semaphore empty used for counting the number of slots that are empty & is initialized to n, the buffer size
- The empty and full semaphores are used for process synchronization

Bounded-Buffer Producer-Consumer Problem

- The structure of the producer process

```
do {
        …
        //   produce an item
      wait (empty);     //wait for an empty space
      wait (mutex);
        …
        //  add the item to the  buffer
        …
      signal (mutex);
      signal (full);      // report the new full slot

} while(TRUE);
```


Bounded-Buffer Producer-Consumer Problem

- The structure of the consumer process

```
do {
      wait (full);    // wait for a stored item
      wait (mutex);
          …
          //  remove an item from  buffer
          …
      signal (mutex);
      signal (empty);  //report the new empty slot
          …
          //  consume the removed item
          …
} while (TRUE)
```

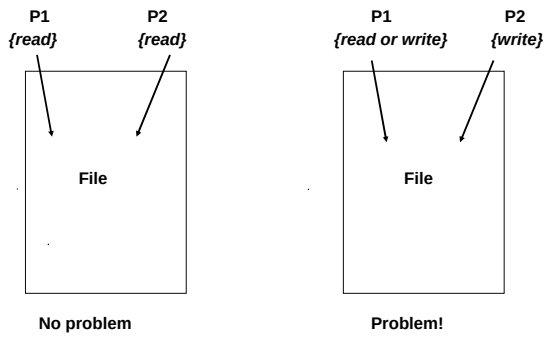
Bounded-Buffer Producer-Consumer Problem

Producer	Consumer
produce an item	
wait (empty)	
• loose processor	
	wait(full) % cannot proceed because there are no items in the buffer
wait(mutex)	• loose processor
add item to buffer	
signal(mutex)	
signal(full)	
produce an item	
wait(empty)	
wait(mutex)	
add item to buffer	
signal (mutex)	
• loose processor	
	wait(full) is now successful because something has been added to buffer

Readers-Writers

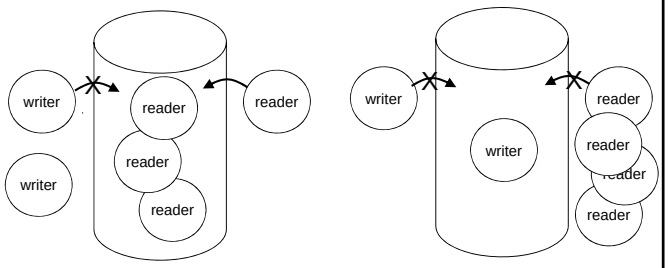
- Concurrent processes share a file, record, or other resources
- Some may read only (readers), some may both read and write (writers)
- Two concurrent reads have no adverse effects
- Problems if
  - concurrent reads and writes
  - multiple writes

Readers-Writers Diagram



The Readers-Writers Problem

Multiple readers or a single writer can use DB.



Readers-Writers Problem

- Two Variations
  - First Readers-Writers problem: No reader be kept waiting unless a writer has already obtained exclusive write permissions (Readers have high priority)
  - Second Readers-Writers problem: If a writer is waiting/ready , no new readers may start reading (Writers have high priority)
- A solution to either problem may result in starvation
  - In the first case , writers may starve; in the second case, reader may starve
- We present solution to first readers writers problem using semaphores

Solution to First Readers-Writers Problem

- The reader processes share the following data structures:
  semaphore mutex , wrt;
  int readcount;
  - The binary semaphores mutex and wrt are initialized to 1; readcount is initialized to 0;
- Semaphore wrt is common to both reader and writer process
  - wrt functions as a mutual exclusion for the writers
  - It is also used by the first or last reader that enters or exits the critical section
  - It is not used by readers who enter or exit while other readers are in their critical section
- The readcount variable keeps track of how many processes are currently reading the object
- The mutex semaphore is used to ensure mutual exclusion when readcount is updated

Solution to first Readers-Writers Problem

- The structure of a writer process

```
do {
        wait (wrt) ;
            …
           //    writing is performed
            …
        signal (wrt) ;
} while (TRUE);
```

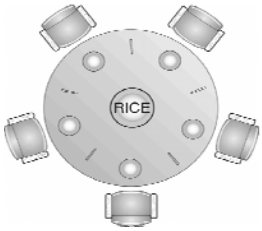

Solution to first Readers-Writers Problem

- The structure of a reader process

```
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);
```

Dining Philosophers Problem

- Five philosophers sit at a round table - thinking and eating
- Each philosopher has one chopstick
  - five chopsticks total
- A philosopher needs two chopsticks to eat
  - philosophers must share chopsticks to eat
- No interaction occurs while thinking



Nature of Problem

- Problem:
  - starvation
    - a philosopher may never get the two chopsticks necessary to eat
  - deadlocks
    - two neighboring philosophers may try to eat at same time

Faulty Solution

Assume that no two neighbors are eating simultaneously

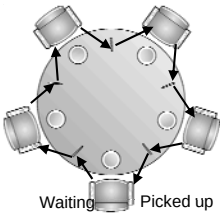
Shared data

binary semaphore chopstick [5] ;
where all elements of chopstick are initialized to 1

- The structure of Philosopher *i*:

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


Deadlock



❖Clearly deadlock is possible (and starvation possible.)
❖Several possible solutions for deadlock:

- ❖ Allow only 4 philosophers to be hungry at a time.
- ❖ Allow pickup only if both chopsticks are available. (Done in critical section)
- ❖ Odd # philosopher always picks up left chopstick1st, even # philosopher always picks up right chopstick 1st.

Problems with Semaphores

- Semaphores must be used with care. Incorrect use of semaphore results in following types of errors:
  - Inversed order: Suppose a process interchanges the order of wait() and signal() operations on the semaphore mutex

```
signal (mutex);
    …
critical section
    …
wait(mutex);
```

    - In this case, multiple processes may be executing in their critical sections , violating the mutual-exclusion requirement

Problems with Semaphores

- Repeated calls: Suppose a process replaces signal(mutex) with wait(mutex)

```
wait(mutex);
  …
  critical section
  …
wait(mutex);
```

    - In this case, processes may block forever (deadlock will occur)
- Omitted calls: Suppose a process omits the wait(mutex), or the signal (mutex) or both
    - In this case, either mutual exclusion is violated or a deadlock will occur

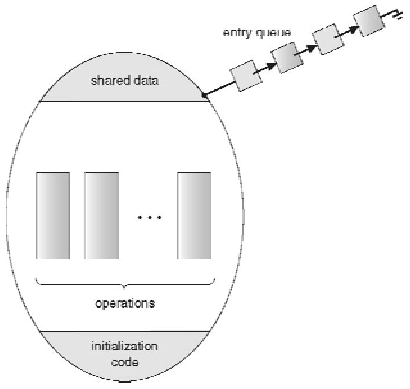
Monitors

- Monitors provides a software solution to synchronization problems
- Monitor is a high-level abstraction that provides a convenient and effective mechanism for process synchronization

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

- Only one process may be active within the monitor at a time
- *Abstract data type*, internal variables only accessible by code within the procedure

Schematic view of a Monitor



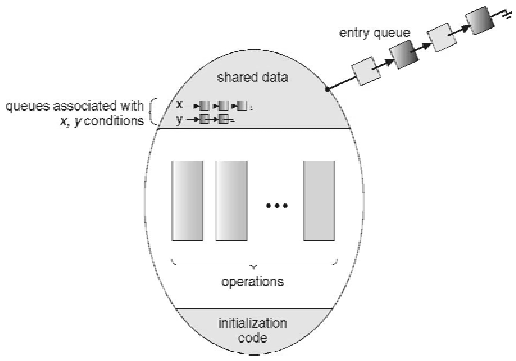
Monitors

- To allow a process to wait within the monitor, one or more variables of type *condition* must be declared, as:

  condition x, y;

- Two operations on a condition variable:
  - **x.wait()** – a process invoking this operation is suspended until another process invokes **x.signal()**
  - **x.signal()** – resumes exactly one suspended process. If no process is suspended, then signal operation has no effect: the state of x is same as if the operation had never been executed

Monitor with Condition Variables



Condition Variables Choices

- If process P invokes **`x.signal(),`** with process Q in **`x.wait()`** state, what should happen next?
  - If Q is resumed, then P must wait. Otherwise, both P and Q would active simultaneously within the monitor
- Two Options include
  - **Signal and wait** – P waits until Q leaves monitor or waits for another condition
  - **Signal and continue** – Q waits until P leaves the monitor or waits for another condition
- Both have pros and cons – language implementer can decide
- A compromise between these two choices was adopted in the Concurrent Pascal language
  - P executing signal immediately leaves the monitor, hence Q is resumed
- Implemented in other languages including Mesa, C#, Java

Solution to Dining Philosophers using Monitors

- We need to distinguish among three states in which we may find a philosopher. For this purpose we introduce the following data structure:

  enum{THINKING, HUNGRY, EATING} state[5]

- Philosopher i can set the variable state[i] =EATING only if her two neighbors are not eating:

  (state[(i+4)%5] != EATING and (state[(i+1)%5] != EATING

- We also need to declare

  condition self[5];

  in which philosopher i can delay herself when she is hungry but is unable to obtain the chopsticks she needs
- The distribution of the chopsticks is controlled by the monitor DiningPhilosophers, defined below:

Solution to Dining Philosophers

```
monitor DiningPhilosophers
  {
    enum { THINKING; HUNGRY, EATING) state [5] ;
    condition self [5];

    void pickup (int i) {
          state[i] = HUNGRY;
          test(i);
          if (state[i] != EATING)
              self [i].wait();
    }

     void putdown (int i) {
          state[i] = THINKING;
                // test left and right neighbors
           test((i + 4) % 5);
           test((i + 1) % 5);
      }
```


Solution to Dining Philosophers (Cont.)

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

      initialization_code() {
          for (int i = 0; i < 5; i++)
              state[i] = THINKING;
     }
  }
```


Solution to Dining Philosophers (Cont.)

- Each philosopher *i* invokes the operations **`pickup()`** and **`putdown()`** in the following sequence:

```
DiningPhilosophers.pickup(i);
            …
           eat
            …
DiningPhilosophers.putdown(i);
```

- No deadlock, but starvation is possible

Monitor Implementation Using Semaphores

- Variables

```
semaphore mutex;  // (initially  = 1)
semaphore next;   // (initially  = 0)
int next_count = 0;
```

- Each procedure ***F*** will be replaced by

```
wait(mutex);
   …
  body of F
   …
if (next_count > 0)
  signal(next);
else
  signal(mutex);
```

- Mutual exclusion within a monitor is ensured.

Monitor Implementation Using Semaphores

- For each condition variable ***x***, we have:

```
semaphore x_sem; // (initially = 0)
int x_count = 0;
```

- The operation x.wait can be implemented as:

```
x_count++;
if (next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

- The operation x.signal can be implemented as:

```
if (x_count > 0) {
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
}
```


Resuming Processes within a Monitor

- If several processes queued on condition x, and an x.signal() operation is executed by some process, then how do we determine which of the suspended processes which should be resumed next?
- FCFS frequently not adequate
- **conditional-wait** construct of the form x.wait(c)
  - Where c is **priority number**
  - Process with lowest number (highest priority) is scheduled next
- ResourceAllocator monitor, which controls the allocation of a single resource among competing processes is given next.

A Monitor to Allocate Single Resource

```
monitor ResourceAllocator
{
    boolean busy;
    condition x;
    void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = TRUE;
    }
    void release() {
        busy = FALSE;
        x.signal();
    }
initialization code() {
    busy = FALSE;
    }
}
```


A Monitor to Allocate Single Resource

- Each process, when requesting an allocation of this resource, specifies the maximum time it plans to use the resource
- Monitor allocates the resource to the process that has the shortest time-allocation request
- A process that needs to access the resource in question must observe the following sequence:

```
R.acquire(t);
    …
access the resource
    …
R.release();
```

where R is an instance of type ResourceAllocator

Problems with Monitors

- A process might access a resource without first gaining access permission to resource
- A process might never release a resource once it has been granted access to resource
- A process might attempt to release a resource that it never requested
- A process might request the same resource twice(without first releasing the resource)