Chapter 4

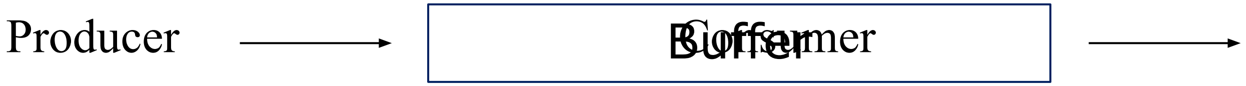
Cont….

Cooperating Processes

- *Concurrent Process* executing in OS may be either independent process or cooperating process
- *Independent* process cannot affect or be affected by the execution of another process.
- Process that does not share any data (temporary or persistent) with other process is independent
- *Cooperating* process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing – several users want to access same file
  - Computation speed-up - Multiprocessing
  - Modularity – through threads
  - Convenience – Even an individual user may have many tasks on which to work at one time Ex:-compiling, editing , printing, etc.

Producer-Consumer Problem

- Example of multi process synchronization problem

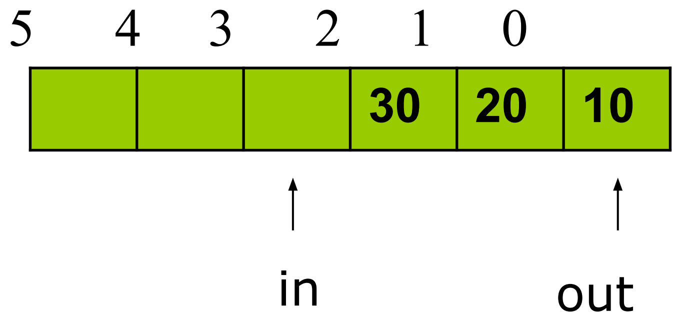


- Here producer and consumer must be synchronized, so that consumer would not consume the item which is not produced yet.
- Consumer must wait until the item is produced.
- Paradigm for cooperating processes, *producer* process produces information that is consumed by a *consumer* process.
  - *unbounded-buffer* places no practical limit on the size of the buffer.
    - 4 Producer keeps on producing items, consumer may have to wait for items

Bounded-Buffer – Shared-Memory Solution

- *bounded-buffer* assumes that there is a fixed buffer size.
  - Consumer must wait if buffer is empty and producer must wait if buffer is full.
- Shared data

```
#define BUFFER_SIZE 6
Typedef struct {
    . . .
} item;
item buffer[BUFFER_SIZE]; // circular array
int in = 0;  //points to next free position
int out = 0;
```

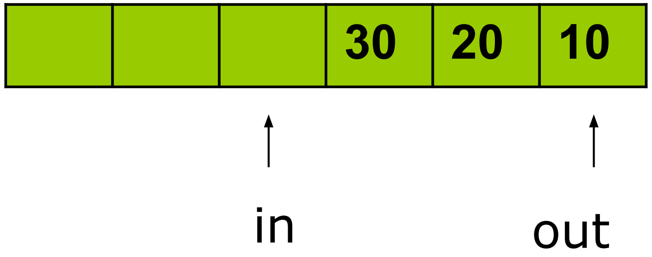


Bounded-Buffer – Producer Process

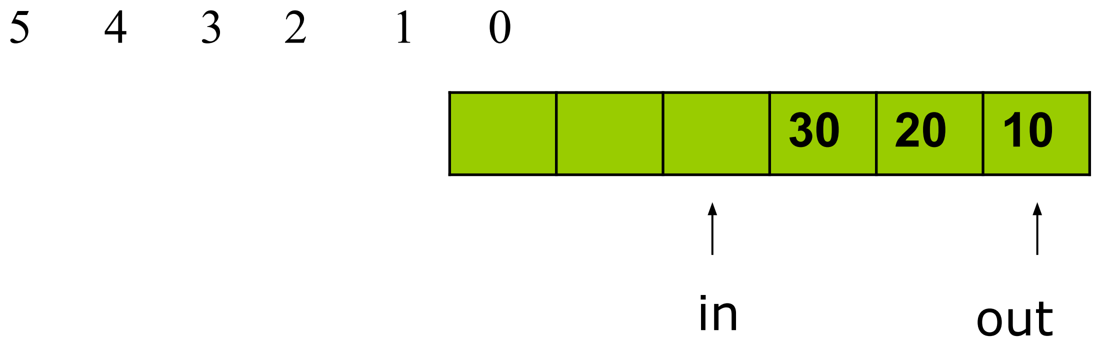
5 4 3 2 1 0

```
item nextProduced;

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



Bounded-Buffer – Consumer Process



```
item nextConsumed;

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

Solution is correct, but can only use BUFFER_SIZE-1 elements

Interprocess Communication (IPC)

- ✓ Mechanism for processes to communicate and to synchronize their actions.
- Message system – processes communicate with each other without resorting to shared variables.
- IPC is useful in a distributed environment where communicating processes may reside on different computers connected with a network. Like chat application
- ✓ IPC facility provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate, they need to:
  - establish a *communication link* between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)

Implementation Questions

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of communicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodate fixed or variable?
- Is a link unidirectional or bi-directional?

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q
- Properties of communication link
  - Links are established automatically.
  - A link is associated with exactly one pair of communicating processes.
  - Between each pair there exists exactly one link.
  - The link may be unidirectional, but is usually bi-directional.

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports).
  - Each mailbox has a unique id.
  - Processes can communicate only if they share a mailbox.
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes.
  - Each pair of processes may share several communication links.
  - Link may be unidirectional or bi-directional.

Indirect Communication

- Mailbox owned by OS are independent, no process is attached to it
- OS allows process to do following:-
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox A.
  - P_1, sends; P_2 and P_3 receive.
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes.
  - Allow only one process at a time to execute a receive operation.
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Communication Modes

- Pipes – Communication between two related processes.
  - 4 Mechanism is half duplex.ie. First process communicates with second process.
  - 4 If second process wants to communicate with first process another pipe is required to achieve full duplex
- FIFO – Communication between two unrelated processes.
  - 4 It is full duplex. Ie both the processes can communicate with each other at the same time
- Message Queue - Communication between two or more processes with full duplex capacity.
  - 4 Processes will communicate with each other by posting a message and retrieving it out of the queue. Once retrieved, message is no longer available in queue.

Communication Modes

- Shared memory – Communication between two or more processes is achieved through a shared piece of memory among all processes.
    - 4 The shared memory needs to be protected from each other by synchronizing access to all the processes.
- Semaphores – Meant for synchronizing access to multiple processes.
    - 4 When one process wants to access memory(for read or write) it needs to be locked(protected) and released when the access is removed.
    - 4 This needs to be repeated by all the processes to secure data.
- Signals – Communication occurs via signaling.
    - 4 Source process will send a signal (number) and destination process will handle it accordingly
- Message Passing

Synchronization

- Message passing may be either blocking or non-blocking.
- **Blocking** is considered **synchronous**
- **Non-blocking** is considered **asynchronous**
- **send** and **receive** primitives may be either blocking or non-blocking.
- **Blocking Send**:- Sending process is blocked until a message is received by receiving process or by mail box
- **Non Blocking Send** :- Sending process sends message and resumes operation
- **Blocking Receive** :- Receiver is blocked until a message is available
- **Non Blocking Receive** :- Receiver retrieves either a valid message or NULL.

Buffering

- Queue of messages attached to the link; implemented in one of three ways.

1. Zero capacity – 0 messages
   Link cannot have any messages waiting in it.

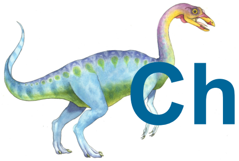
   Sender must wait for receiver (rendezvous).

2. Bounded capacity – finite length of *n* messages
   n messages can reside in it.

   Sender must wait if link full.

3. Unbounded capacity – infinite length
   Any number of messages can wait in queue

   Sender never waits.



Chapter 7: Process Synchronization

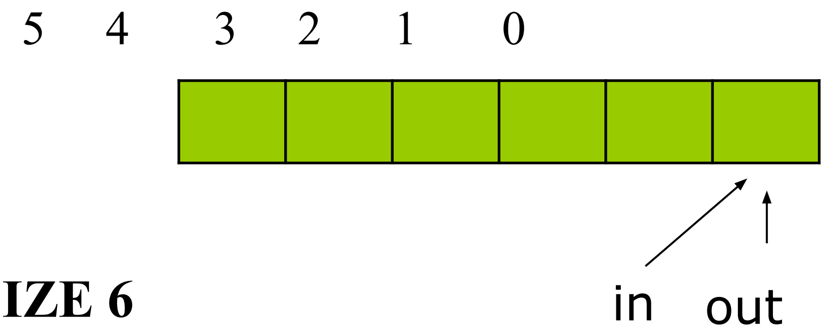
- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Critical Regions
- Monitors
- Synchronization in Solaris 2 & Windows 2000

Background

- 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 solution to bounded-butter problem allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used :
  - Suppose that 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

- Shared data



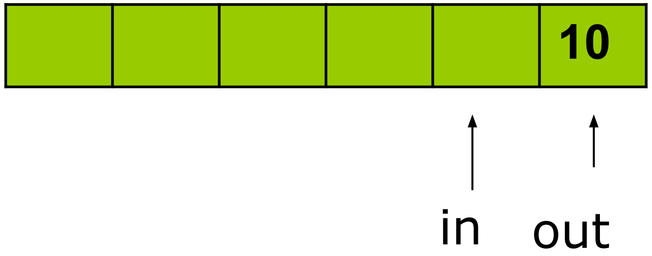
```
#define BUFFER_SIZE 6
typedef struct {
   . . .
} item;
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Bounded-Buffer

- Producer process

```
item nextProduced;

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

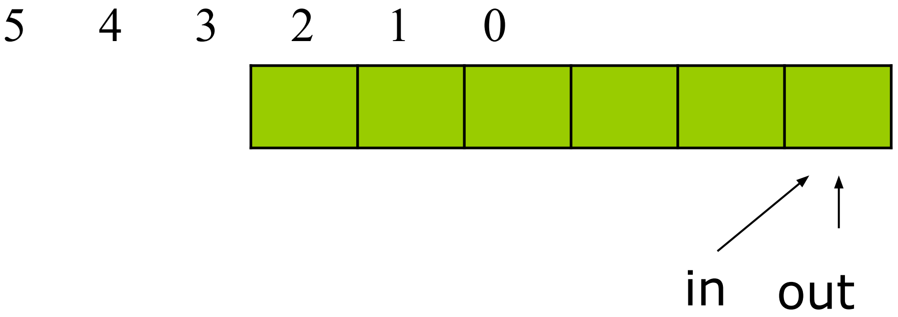


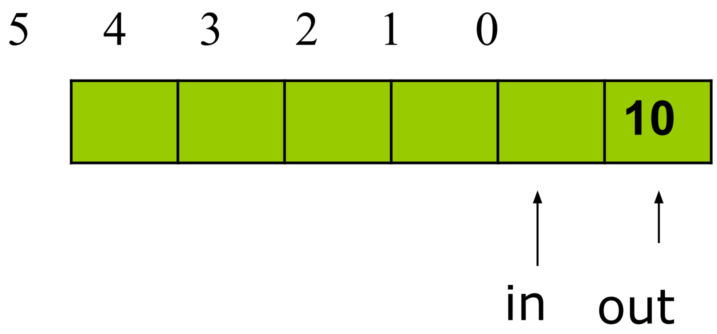
Bounded-Buffer

- Consumer process

```
item nextConsumed;

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





Bounded Buffer

- The statements

  counter++;
  counter--;

  must be performed *atomically*.

- Atomic operation means an operation that completes in its entirety without interruption.

Bounded Buffer

- The statement “**count++**” may be implemented in machine language as:

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

- The statement “**count—**” may be implemented as:

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

Bounded Buffer

- If both the producer and consumer attempt to update the buffer concurrently, the assembly language statements may get interleaved.

- Interleaving depends upon how the producer and consumer processes are scheduled.

Bounded Buffer

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

producer:

register1 = counter (*register1 = 5*)
register1 = register1 + 1 (*register1 = 6*)

counter = register1 (*counter = 6*)

Consumer

register2 = counter (*register2 = 6*)
register2 = register2 – 1 (*register2 = 5*)
counter = register2 (*counter = 5*)

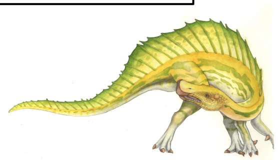
The value of **count** is 5.

Bounded Buffer

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

producer:	**Consumer**
register1 = counter (*register1 = 5*)	
register1 = register1 + 1 (*register1 = 6*)	
	register2 = counter (*register2 = 5*)
	register2 = register2 – 1 (*register2 = 4*)
counter = register1 (*counter = 6*)	
	counter = register2 (*counter = 4*)

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



Race Condition

- **Race condition**: The situation where several processes access and manipulate shared data concurrently. The final value of the shared data depends upon which process finishes last.

- To prevent race conditions, concurrent processes must be **synchronized**.

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.
- ✓ Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.

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**. 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.

Initial Attempts to Solve Problem

- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

```
do {
        entry section
                critical section
        exit section
                reminder section
} while (1);
```

- Processes may share some common variables to synchronize their actions.

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
  - **turn = i** ⇒ P_i can enter its critical section
- Process P_i

```
do {
        while (turn != i) ;
                critical section
        turn = j;
                reminder section
} while (1);
```

- Satisfies mutual exclusion, but not progress

Algorithm 1

Turn = 0

P0

```
While(1)
{
      While(turn !=0);
      CS
       turn =1;
        Remainder section
}
```

Satisfies mutual exclusion, but not progress

P1

```
While(1)
{
       While(turn !=1);
      CS
        turn =0;
        Remainder section
}
```

this solution does not satisfy the progress property, which states that a process outside the CS must not block another process from entering the CS. In your code, if P0 finishes its CS and sets turn to 1, but then goes into an infinite loop in the remainder section, P1 will never be able to enter the CS, even though it is ready to do so. This is a violation of the progress property, and it can lead to starvation of P1.

Algorithm 2

- Shared variables
  - **boolean flag[2]**;
    initially **flag [0] = flag [1] = false.**
  - **flag [i] = true** ⇒ P_i ready to enter its critical section
- Process P_i

```
do {
        flag[i] := true;
        while (flag[j]) ;                              critical section
        flag [i] = false;
                remainder section
} while (1);
```

- Satisfies mutual exclusion, but not progress requirement.

Algorithm 2

flag	F	F
	0	1

```
P0

while (1){
    flag[0] := true;
    while (flag[1]) ; critical
    section
    flag [0] = false;
    };
```

```
P1

while (1){
    flag[1] := true;
    while (flag[0]) ; critical
    section
    flag [1] = false;
    };
```

if both processes set their flags to true and then check the other process's flag before the other process can clear it. In this case, both processes will see the other process's flag as true and will never enter the CS. This is a violation of the progress property, which states that a process outside the CS must not block another process from entering the CS2.

Algorithm 2

flag	T	F
	0	1

P0

```
while (1){
    flag[0] := true;
    while (flag[1]) ; critical
    section
    flag [0] = false;
    };
```

P1

```
while (1){
    flag[1] := true;
    while (flag[0]) ; critical
    section
    flag [1] = false;
    };
```

Algorithm 2

flag	F	F
	0	1

```
P0

while (1){
    flag[0] := true;
    while (flag[1]) ; critical
    section
    flag [0] = false;
    };
```

```
P1

while (1){
    flag[1] := true;
    while (flag[0]) ; critical
    section
    flag [1] = false;
    };
```

Algorithm 2

flag	F	T
	0	1

P0

```
while (1){
    flag[0] := true;
    while (flag[1]) ; critical section
    flag [0] = false;
    };
```

P1

```
while (1){
    flag[1] := true;
    while (flag[0]) ; critical section
    flag [1] = false;
    };
```

Algorithm 2

flag	F	F
	0	1

P0

```
while (1){
    flag[0] := true;
    while (flag[1]) ; critical section
    flag [0] = false;
    };
```

P1

```
while (1){
    flag[1] := true;
    while (flag[0]) ; critical section
    flag [1] = false;
    };
```

- Leads to deadlock

- Satisfies Mutual Exclusion
- Progress not satisfied

Algorithm 3

- Combined shared variables of algorithms 1 and 2.
- Process P_i

```
  do {
          flag [i]:= true;
          turn = j;
          while (flag [j] and turn = j) ;
                  critical section
          flag [i] = false;
                  remainder section
  } while (1);
```

- Meets all three requirements; solves the critical-section problem for two processes.

Operating System Concepts

Algorithm 3

P0

```
While(1)
{
    flag [0]:= T;
    turn = 1;
    while (flag [1] == T && turn ==1);
     critical section
    flag [0] = F;
}
```

P1

```
While(1)
{
    flag [1]:= T;
    turn = 0;
    while (flag[0] == T && turn==1);
     critical section
    flag [1] = F;
}
```

Turn = 1

flag	F	F
	0	1

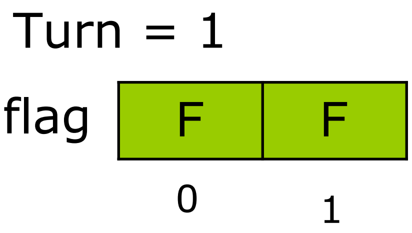
Algorithm 3(Peterson's Algorithm)

P0

```
While(1)
{
     flag [0]:= T;
     turn = 1;
     while (flag [1] == T && turn ==1);
       critical section
     flag [0] = F;
}
```

P1

```
While(1)
{
     flag [1]:= T;
     turn = 0;
     while (flag[0] == T && turn==0);
       critical section
     flag [1] = F;
}
```



Turn = 1

flag	F	F
	0	1

- Satisfies Mutual Exclusion
- Satisfies Progress
- Satisfies Bounded Wait

Bakery Algorithm

Critical section for n processes

- Before entering its critical section, process receives a number. Holder of the smallest number enters the critical section.
- If processes P_i and P_j receive the same number, if $i < j$, then P_i is served first; else P_j is served first.
- The numbering scheme always generates numbers in increasing order of enumeration; i.e., 1,2,3,4,5...

Simplified Bakery Algorithm

```
Lock(i) {
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        for (j = 0; j < n; j++) {
                while ((number[j] != 0) && (number[j] < number[i])) ;
                }

      critical section

Unlock(i){
        number[i] = 0;
        remainder section
        }
```

p1	p2	p3	p4	p5
0	0	0	0	0

Simplified Bakery Algorithm

```
Lock(i) {
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        for (j = 0; j < n; j++) {
                while ((number[j] != 0) && (number[j] < number[i])) ;
               }

      critical section

Unlock(i){
        number[i] = 0;
        remainder section
        }
```

p1	p2	p3	p4	p5
0	1	2	4	3

Simplified Bakery Algorithm

```
Lock(i) {
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        for (j = 0; j < n; j++) {
                while ((number[j] != 0) && (number[j] < number[i])) ;
                }

      critical section

Unlock(i){
        number[i] = 0;
        remainder section
        }
```

p1	p2	p3	p4	p5
0	0	2	4	3

Simplified Bakery Algorithm

```
Lock(i) {
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        for (j = 0; j < n; j++) {
                while ((number[j] != 0) && (number[j] < number[i])) ;
               }

      critical section

Unlock(i){
        number[i] = 0;
        remainder section
        }
```

p1	p2	p3	p4	p5
0	0	0	4	3

Simplified Bakery Algorithm

```
Lock(i) {
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        for (j = 0; j < n; j++) {
                while ((number[j] != 0) && (number[j] < number[i])) ;
                }

      critical section

Unlock(i){
        number[i] = 0;
        remainder section
        }
```

p1	p2	p3	p4	p5
0	0	0	0	0

Simplified Bakery Algorithm

```
Lock(i) {
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        for (j = 0; j < n; j++) {
                while ((number[j] != 0) && (number[j] < number[i])) ;
                }

      critical section

Unlock(i){
        number[i] = 0;
        remainder section
        }
```

p1	p2	p3	p4	p5
0	1	2	2	3

Simplified Bakery Algorithm

Must be in atomic fashion

```
Lock(i) {
        number[i] = max(number[0], number[1], …, number [n – 1])+1;
        for (j = 0; j < n; j++) {
                while ((number[j] != 0) && (number[j] < number[i])) ;
                }

      critical section

Unlock(i){
        number[i] = 0;
        remainder section
        }
```

p3

p4

p1	p2	p3	p4	p5
0	0	2	2	3

- Mutual Exclusion is not satisfied

Bakery Algorithm

- Notation <≡ lexicographical order (ticket #, process id #)
  - (a,b) < c,d) if $a < c$ or if $a = c$ and $b < d$
  - max ($a_0,\ldots, a_{n-1}$) is a number, k, such that $k \geq a_{\mathrm{i}}$ for i - 0, …, $n - 1$
- Shared data

  boolean choosing[n];

  int number[n];

Data structures are initialized to **false** and **0** respectively

Bakery Algorithm

```
do {
      choosing[i] = true;
      number[i] = max(number[0], number[1], …, number [n – 1])+1;
      choosing[i] = false;
      for (j = 0; j < n; j++) {
            while (choosing[j]) ;
            while ((number[j] != 0) && ((number[j],j) < (number[i],i)) ;
                              }
       critical section

      number[i] = 0;
       remainder section
} while (1);
```

p1	p2	p3	p4	p5
0	1	2	2	3

Bakery Algorithm

```
do {
      choosing[i] = true;
      number[i] = max(number[0], number[1], …, number [n – 1])+1;
      choosing[i] = false;
      for (j = 0; j < n; j++) {
            while (choosing[j]) ;
            while ((number[j] != 0) && ((number[j],j) < (number[i],i)) ;
                          }
        critical section

      number[i] = 0;
        remainder section
} while (1);
```

p1	p2	p3	p4	p5
0	1	2	~~2~~	

Synchronization Hardware

- Many systems provide hardware support for critical section problem.
- Uniprocessor
  - Could disable interrupt
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor system
- Modern machines provide special **atomic** hardware instructions

Synchronization Hardware

- Test and modify the content of a word atomically

```
boolean TestAndSet(boolean &target) {
        boolean rv = target;
        target = true;

        return rv;
}
```

Mutual Exclusion with Test-and-Set

- Shared data:

```
boolean lock = false;
```

- Process P_i

```
do {
        while (TestAndSet(lock)) ;
                critical section
        lock = false;
                remainder section
}while(true)
```

Mutual Exclusion with Test-and-Set

P1

- Shared data:
  boolean lock = false;
- Process *p1*

```
do {
while(TestAndSet(lock)) ;
  critical section
lock = false;
  remainder section
}while(true)
```

Prempt

P2

- Shared data:
  boolean lock = false;
- Process P_2

```
do {
while(TestAndSet(lock)) ;
  critical section
lock = false;
  remainder section
}while(true)
```

- Satisfies Mutual Exclusion
- Satisfies Progress

P1 -> lock = false -> r=false -> lock = true -> prempt
P2 -> lock = true -> r = true -> lock = true -> infinite loop

Synchronization Hardware

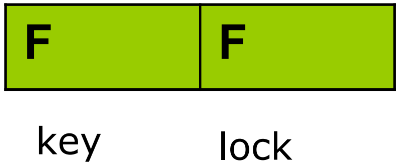
- Atomically swap two variables.

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

Mutual Exclusion with Swap

- Shared data (initialized to **false**):

```
 boolean lock;
        boolean waiting[n];
```



F	F
key	lock

- Process P_i

```
 do {
        key = true;
        while (key == true)
                Swap(lock,key);
                critical section
        lock = false;
                remainder section
 }
```

Mutual Exclusion with Swap

F	F
key	lock

- Shared data (initialized to **false**):
  boolean lock;
- Process *P1*

```
do {
        key = true;
        while (key == true)
        Swap(lock,key);
                critical section
        lock = false;
                remainder section
}
```

- Shared data (initialized to **false**):
  boolean lock;
- Process *P_2*

```
do {
        key = true;
        while (key == true)
        Swap(lock,key);
                critical section
        lock = false;
                remainder section
}
```

Bounded Waiting Mutual Exclusion with Test and Set

Semaphores

- Synchronization tool that does not require busy waiting.
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

```
wait (S):
        while $S \le 0$ do no-op;
                S--;

signal (S):
        S++;
```

Critical Section of *n* Processes

- Shared data:

  semaphore mutex; //initially *mutex* = 1

- Process *Pi:*

```
do {
    wait(mutex);
        critical section
    signal(mutex);
        remainder section
} while (1);
```

Semaphore Implementation

- Define a semaphore as a record

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

- Assume two simple operations:
  - **block** suspends the process that invokes it.
  - **wakeup(*P*)** resumes the execution of a blocked process **P**.

Implementation

- Semaphore operations now defined as

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

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

Semaphore as a General Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
- Use semaphore *flag* initialized to 0
- Code:

```
P_i        P_j
 □          □
A          wait(flag)
signal(flag)     B
```

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);
prempt
  □              □
signal(S);     signal(Q);
signal(Q)      signal(S);
```

- **Starvation** – indefinite blocking. A process may never be removed from the semaphore queue in which it is suspended.

```
  P1                          P2
Signal (s)                  Signal (s)
Cs;                         Cs;
Wait(s)                     Wait(s)
```

S = 1 (initially)

Two Types of Semaphores

- *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.
- Can implement a counting semaphore *S* as a binary semaphore.

Implementing *S* as a Binary Semaphore

- Data structures:

  binary-semaphore S1, S2;

  int C:

- Initialization:

  S1 = 1

  S2 = 0

  C = initial value of semaphore **S**

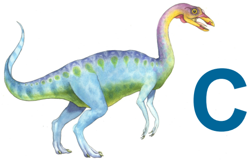
Implementing *S*

- *wait* operation

```
wait(S1);
C--;
if (C < 0) {
            signal(S1);
            wait(S2);
}
signal(S1);
```

- *signal* operation

```
      wait(S1);
      C ++;
      if (C <= 0)
            signal(S2);
      else
            signal(S1);
```



Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Shared data

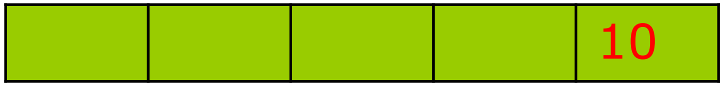
semaphore full, empty, mutex;

Initially:

full = 0, empty = n, mutex = 1

Bounded-Buffer Problem

full = 0,
empty = n = 5,
mutex = 1



				10

Producer

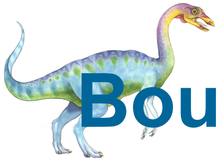
```
do {
              …
    produce an item in nextp
              …
    wait(empty);
    wait(mutex);
              …
          add nextp to buffer
              …
    signal(mutex);
    signal(full);
    } while (1);
```

Consumer

```
do {
    wait(full)
    wait(mutex);
      …
    remove an item from buffer to nextc
      …
    signal(mutex);
    signal(empty);
      …
    consume the item in nextc
      …
    } while (1);
```

Bounded-Buffer Problem Producer Process

```
do {
            …
        produce an item in nextp
            …
        wait(empty);
        wait(mutex);
            …
        add nextp to buffer
            …
        signal(mutex);
        signal(full);
} while (1);
```



Bounded-Buffer Problem Consumer Process

```
do {
        wait(full)
        wait(mutex);
                …
        remove an item from buffer to nextc
                …
        signal(mutex);
        signal(empty);
                …
        consume the item in nextc
                …
} while (1);
```

Readers-Writers Problem

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process

- mutex = 1,
- wrt = 1,
- readcount = 0

Writer

```
wait(wrt);
   …
  writing is performed
   …
signal(wrt);
```

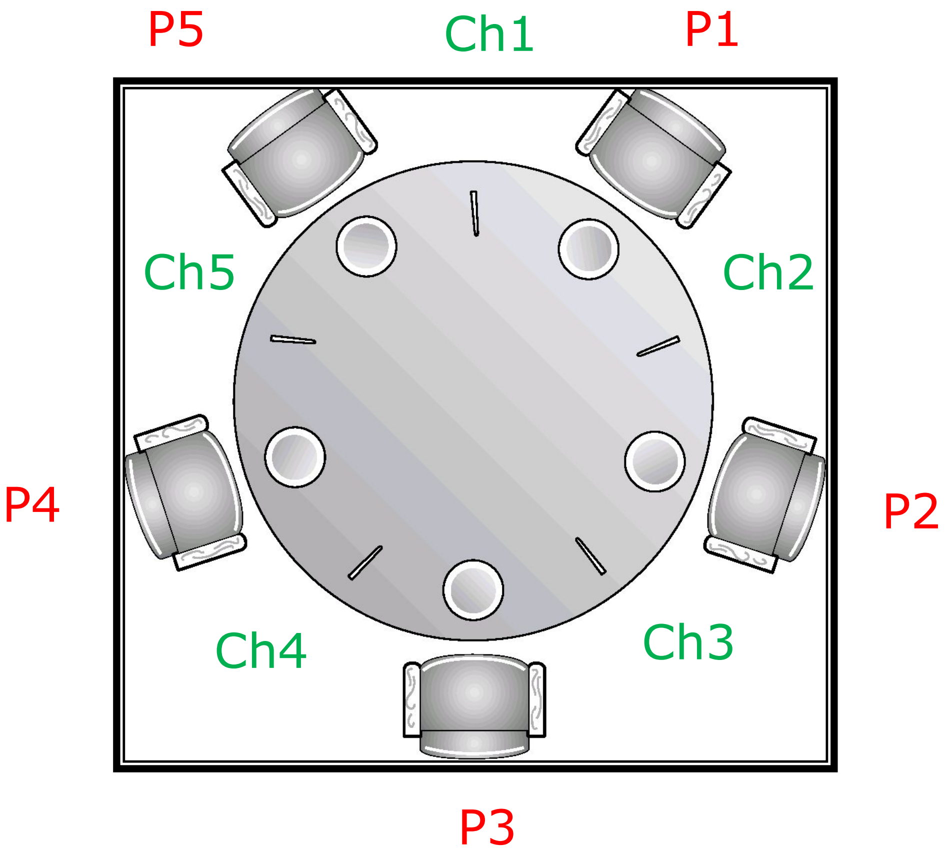
Reader

```
wait(mutex);
      readcount++;
      if (readcount == 1)
     wait(wrt);
signal(mutex);
       …
      reading is performed
       …
wait(mutex);
      readcount--;
      if (readcount == 0)
      signal(wrt);
signal(mutex):
```

Readers-Writers Problem Reader Process

```
wait(mutex);
readcount++;
if (readcount == 1)
            wait(rt);
signal(mutex);
            …
      reading is performed
            …
wait(mutex);
readcount--;
if (readcount == 0)
      signal(wrt);
signal(mutex):
```

Dining-Philosophers Problem



- Shared data

  semaphore chopstick[5];

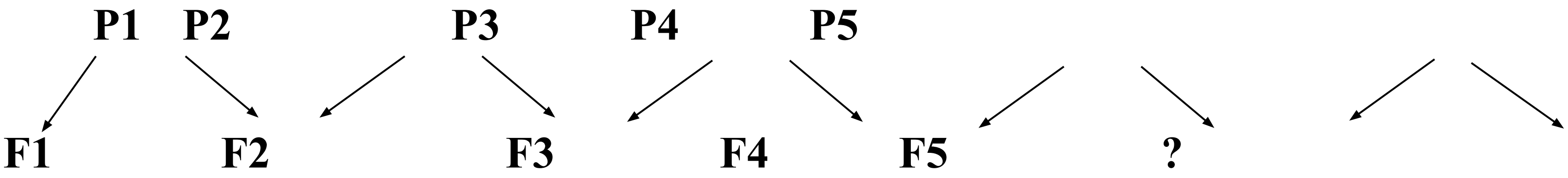
Initially all values are 1

Dining-Philosophers Problem

- Philosopher *i*:

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

Dining-Philosophers Problem



P1 P2 P3 P4 P5

F1 F2 F3 F4 F5 ?

Dining-Philosophers Problem

- Use Binary Semaphore -> S[i]
- Philosopher *i*:

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

Deadlock Situation

- After getting left fork all the processes get preempted.
-

P0	**P1**	**P2**	**P3**	**P4**
F0	**F1**	**F2**	**F3**	**F4**

- All are waiting for another fork to be released.
- Thus deadlock occurs
- Soln:-
  - Reverse the sequence of last philosopher
  - Allow n-1 philosophers to sit and eat
  - Philosophers will pick chopstick if both chopsticks are available

Critical Regions

- High-level synchronization construct
- A shared variable ***v*** of type ***T***, is declared as:

  v: shared T
- Variable ***v*** accessed only inside statement

  region v when B do S

  where ***B*** is a boolean expression.
- While statement ***S*** is being executed, no other process can access variable ***v***.

Critical Regions

- Regions referring to the same shared variable exclude each other in time.

- When a process tries to execute the region statement, the Boolean expression *B* is evaluated. If *B* is true, statement *S* is executed. If it is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*.

Example – Bounded Buffer

- Shared data:

```
struct buffer {
        int pool[n];
        int count, in, out;
}
```

Operating System Concepts

Bounded Buffer Producer Process

- Producer process inserts **nextp** into the shared buffer

```
region buffer when( count < n) {
        pool[in] = nextp;
        in:= (in+1) % n;
        count++;
}
```

Bounded Buffer Consumer Process

- Consumer process removes an item from the shared buffer and puts it in **nextc**

```
region buffer when (count > 0) {                    nextc = pool[out];
        out = (out+1) % n;
        count--;
}
```

Operating System Concepts

Implementation region *x* when *B* do *S*

- Associate with the shared variable *x*, the following variables:

  semaphore mutex, first-delay, second-delay;
  int first-count, second-count;

- Mutually exclusive access to the critical section is provided by **mutex**.

- If a process cannot enter the critical section because the Boolean expression **B** is false, it initially waits on the **first-delay** semaphore; moved to the **second-delay** semaphore before it is allowed to reevaluate *B*.

Implementation

- Keep track of the number of processes waiting on **first-delay** and **second-delay**, with **first-count** and **second-count** respectively.
- The algorithm assumes a FIFO ordering in the queuing of processes for a semaphore.
- For an arbitrary queuing discipline, a more complicated implementation is required.

Monitors

- ✓ High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.

```
monitor monitor-name
{
          shared variable declarations
          procedure body P1 (…) {
                    . . .
          }
          procedure body P2 (…) {
                    . . .
          }
          procedure body Pn (…) {
                     . . .
          }
          {
                    initialization code
          }
}
```

Monitors

- To allow a process to wait within the monitor, a **condition** variable must be declared, as

  condition x, y;

- Condition variable can only be used with the operations **wait** and **signal**.
  - The operation

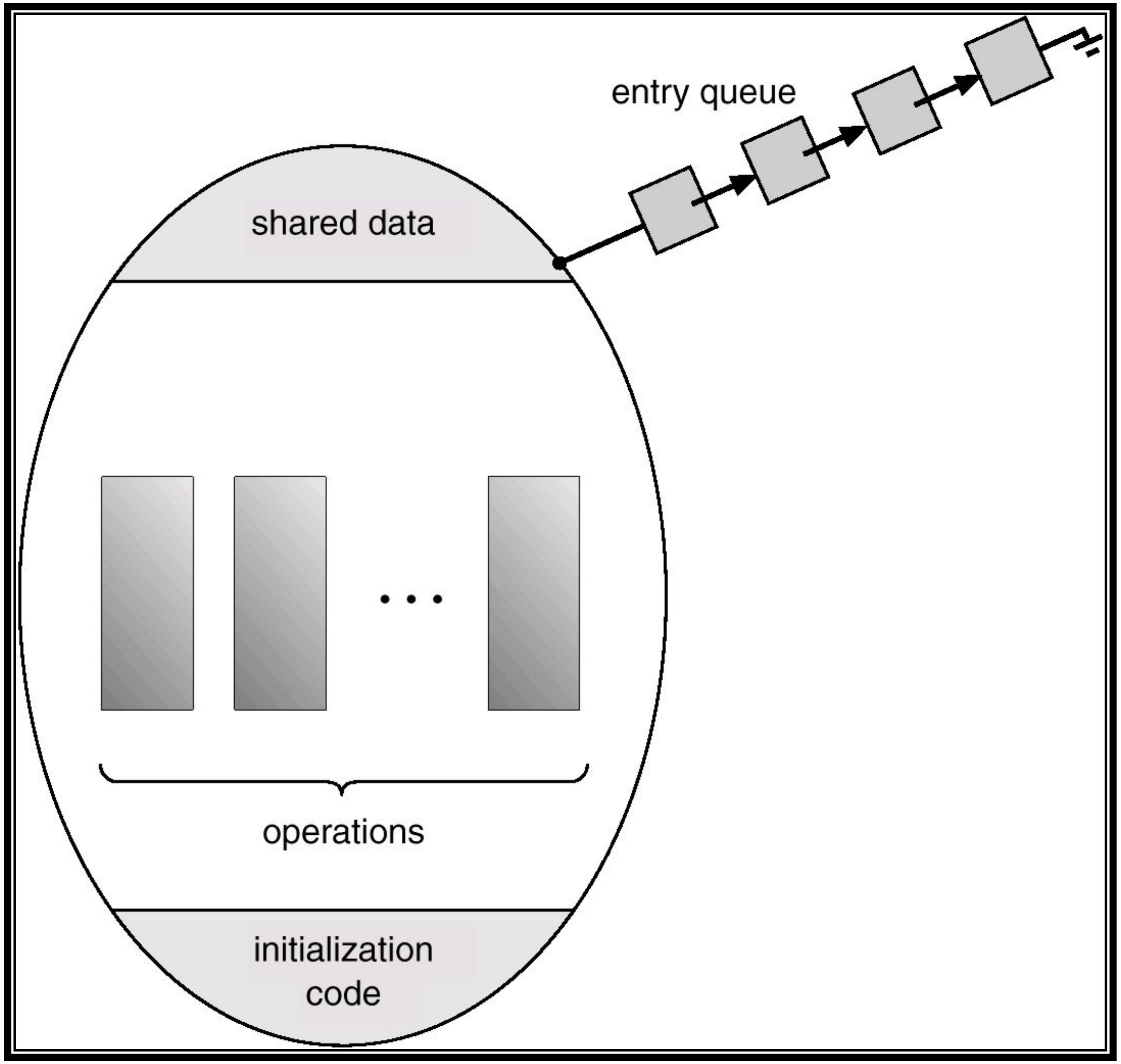
    x.wait();

    means that the process invoking this operation is suspended until another process invokes

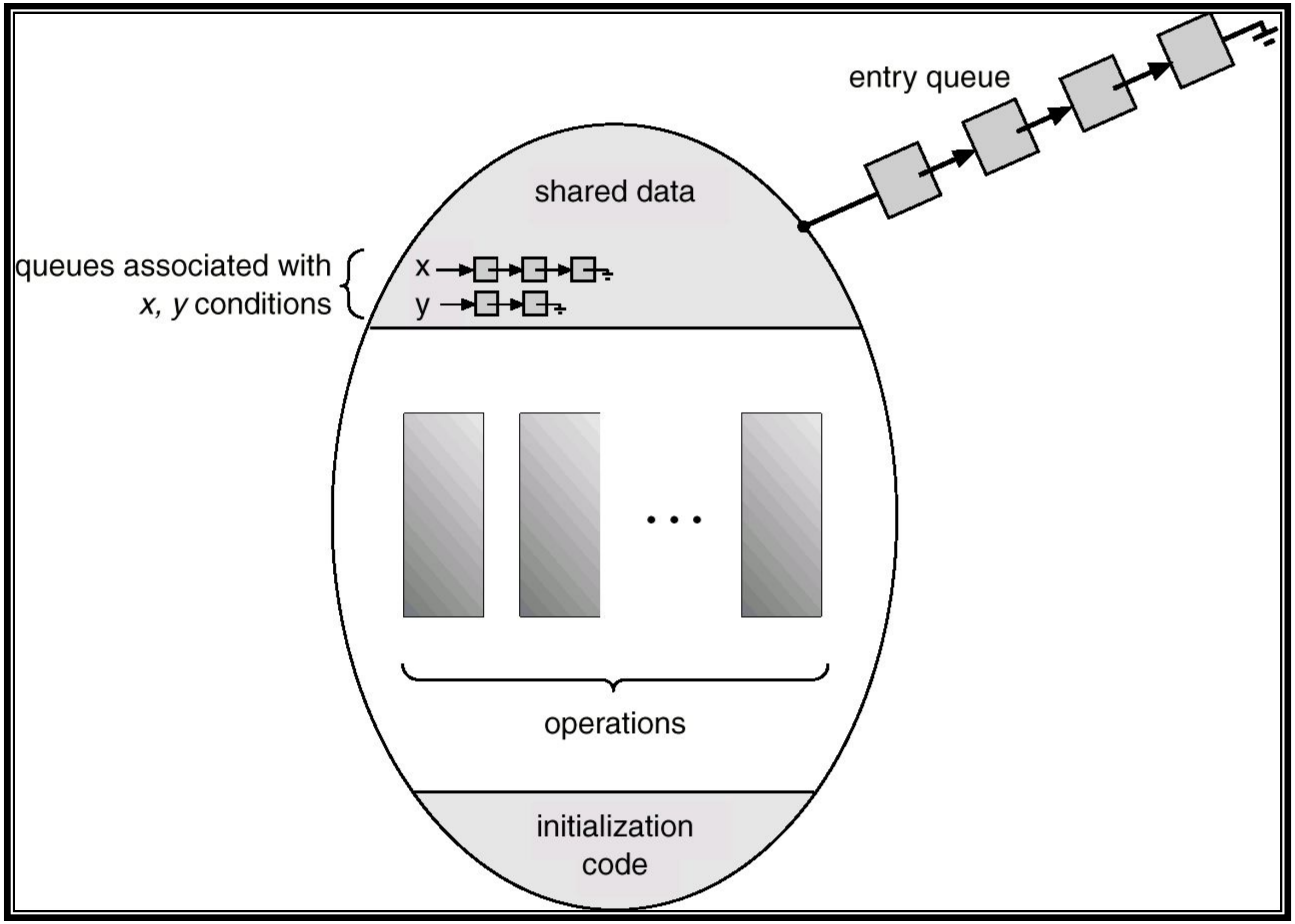
    x.signal();

  - The **x.signal** operation resumes exactly one suspended process. If no process is suspended, then the **signal** operation has no effect.

Schematic View of a Monitor



Monitor With Condition Variables



Dining Philosophers Example

```
monitor dp
{
  enum {thinking, hungry, eating} state[5];
  condition self[5];
  void pickup(int i)          // following slides
  void putdown(int i)         // following slides
  void test(int i)            // following slides
  void init() {
         for (int i = 0; i < 5; i++)
                state[i] = thinking;
  }
}
```

Here philosopher i can delay himself when he is hungry but unable to obtain the chopstick he needs.

Dining Philosophers

```
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);
}
```

Dining Philosophers

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

Monitor Implementation Using Semaphores

- Variables

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

- Each external 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

- 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--;
```

Monitor Implementation

- The operation **x.signal** can be implemented as:

```
if (x-count > 0) {
        next-count++;
        signal(x-sem);
        wait(next);
        next-count--;
}
```

Monitor Implementation

- *Conditional-wait* construct: **x.wait(c);**
  - **c** – integer expression evaluated when the **wait** operation is executed.
  - value of **c** (a *priority number*) stored with the name of the process that is suspended.
  - when **x.signal** is executed, process with smallest associated priority number is resumed next.
- Check two conditions to establish correctness of system:
  - User processes must always make their calls on the monitor in a correct sequence.
  - Must ensure that an uncooperative process does not ignore the mutual-exclusion gateway provided by the monitor, and try to access the shared resource directly, without using the access protocols.

Solaris 2 Synchronization

- Implements a variety of locks to support multitasking, multithreading (including real-time threads), and multiprocessing.

- Uses *adaptive mutexes* for efficiency when protecting data from short code segments.

- Uses *condition variables* and *readers-writers* locks when longer sections of code need access to data.

- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.

Windows 2000 Synchronization

- Uses interrupt masks to protect access to global resources on uniprocessor systems.
- Uses *spinlocks* on multiprocessor systems.
- Also provides *dispatcher objects* which may act as wither mutexes and semaphores.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable.