Process Synchronization

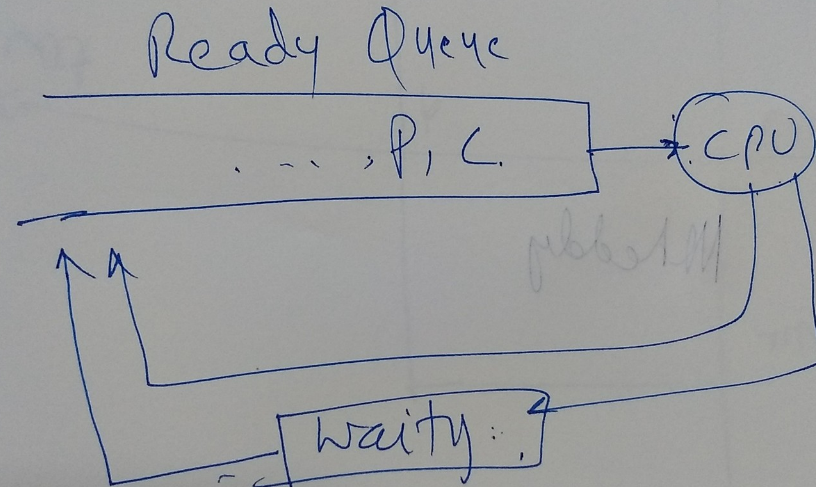
- Background
- The Critical-Section Problem
- Synchronization Hardware
- Semaphores
- Classical Problems of Synchronization
- Monitors
- Synchronization in Solaris 2 & Windows 2000

Process cooperation and synchronization

- Process Synchronization
  - …mechanisms to ensure the orderly execution of cooperating processes that share a logical address space, so that data consistency is maintained.
- Why do process cooperate ?
  - Modularity: breaking up a system into several subsystems
    - E.g, an interrupt handler and device driver that need to communicate.
  - Convenience: users might want to have several processes to share data
  - Speed up: a single program is run as several sub-programs
- How do processes co-operate ?
  - Communication abstraction: producers and consumers
    - Producers produce a piece of information
    - Customers use this information.

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-buffer problem allows at most $n - 1$ items in buffer at the same time. A solution, where all N buffers are used is not simple.
    - 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


Ready Queue
...., P, C.
CPU
waity

Bounded-Buffer

- Shared data

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

Bounded-Buffer

- Producer process

```
item nextProduced;

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

Bounded-Buffer

- Consumer process

```
item nextConsumed;

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

Bounded Buffer

- Although, both the producer and consumer routines are correct separately they may not function correctly when executed concurrently.
- 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:

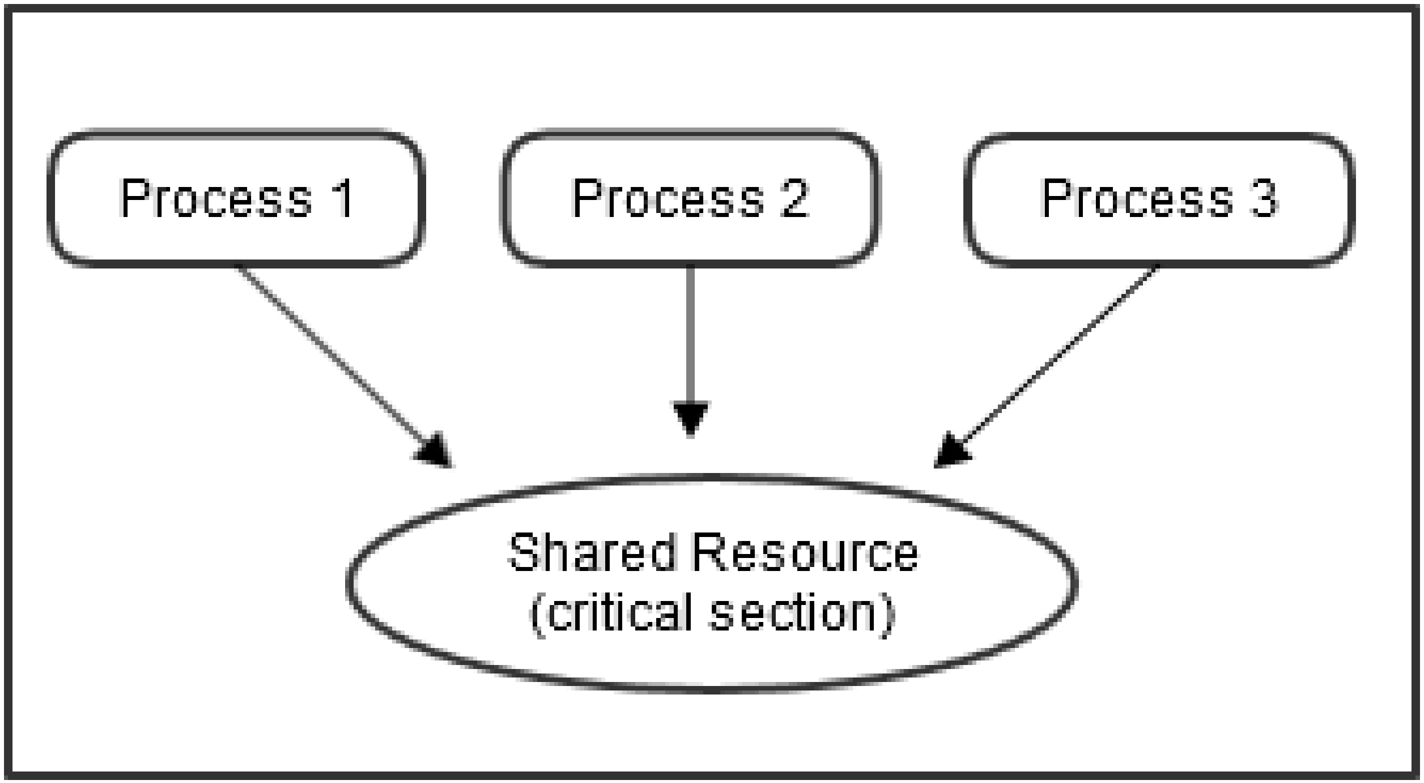
  producer: **register1 = counter** (*register1 = 5*)
  producer: **register1 = register1 + 1** (*register1 = 6*)
  consumer: **register2 = counter** (*register2 = 5*)
  consumer: **register2 = register2 – 1** (*register2 = 4*)
  producer: **counter = register1** (*counter = 6*)
  consumer: **counter =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


Process 1
Process 2
Process 3
Shared Resource
(critical section)

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

- A solution to critical section problem must satisfy the following conditions.
  - **Mutual Exclusion**. If process P_i is executing in its critical section, then no other processes can be executing in their critical section.
  - **Progress**. At least one process requesting entry into CS will be able to enter it if there is no other process in it..
  - **Bounded Waiting**. No process waits indefinitely to enter CS once it has requested entry.
- Assume that each process executes at a nonzero speed
- No assumption concerning relative speed of the n processes.

Two approaches

- Several kernel-level processes may active at a time
  - Example: Data structure “List of open files”
- Kernel developers should ensure that OS is free from race conditions.
- Two approaches are ued
- Non-preemptive kernel
  - A non-preemptive kernel does not allow a process running in the kernel mode to be preempted.
  - Kernel mode process runs until it exists kernel mode, blocks, or voluntarily yields the control of CPU
  - Free from race conditions
- Preemptive kernel
  - A preemptive kernel allows a process to be pre-empted while it is running in kernel mode.
  - Should be carefully designed
  - Difficult to design especially in SMP
- Why we prefer preemptive kernels ?
  - Suitable for realtime programming
  - More responsive as kernel mode process can not run for a longer time.
- WINDOWS XP, WINDOWS 2000, Prior to LINUX 2.6 are non-preemptive
- Solaris and IRIX are preemptive

Mutual exclusion: Software approaches

- Software approaches can be implemented
- Assume elementary mutual exclusion at the memory access level.
  - Simultaneous access to the same location in main memory are serialized in some order.
- Beyond this, no other support in the hardware, OS, programming language is assumed.

Two process solution
Initial Attempts to Solve Problem
Dekker's algorithm

- Reported by Dijkstra, 1965.
- Only 2 processes, P_0 and P_1
- General structure of process P_i (other process P_j)

do {

entry section

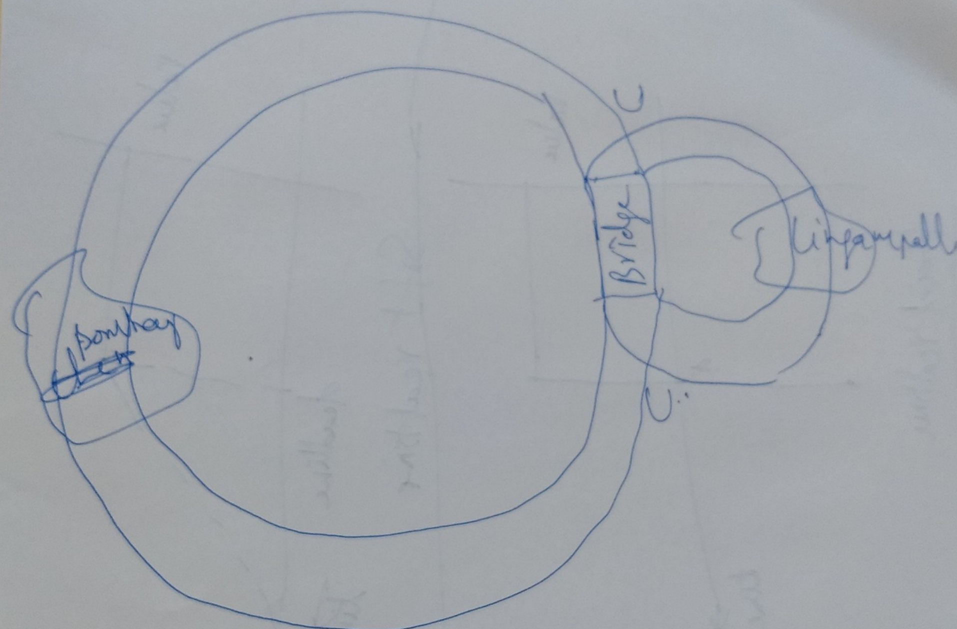
critical section

exit section

reminder section

} **while (1)**;

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


C
Bridge
Lingampalli
Bombay
C

Algorithm 1

- Shared variables:
  - **int turn**;
    initially **turn = 0**
- Turn variable

P0 | **P1**

P0	P1
while (turn != 0) ;	while (turn != 1);
/ Do nothing */*	*/* Do nothing */*
critical section	*critical section*
turn = 1;	turn = 0;
remainder section	remainder section

  - Shared variable *turn* indicates who is allowed to enter next, can enter if *turn = me*
  - On exit, point variable to other process
  - Deadlock if other process never enters
- +Satisfies mutual exclusion: Only one process can enter in CS
- -It does not satisfy the progress requirement, as it requires strict alternation of processes to enter CS.
- The pace of execution is dictated by slower process.
- If turn=0, P1 is ready to enter into CS, P1 can not do so, even though P0 may be in the RS.
- If one process fails in CS or RS, other process is blocked permanently.

Algorithm 2

- Problem with Alg1
  - It does not retain sufficient information about the state of each process.
  - Alg1 remembers only which process is allowed to enter the CS.
- To solve this problem, variable turn is replaced by **boolean flag[2]**; flag[0] is for P0; and flag[1] is for P1.
- Each process may examine the other’s flag but may not alter it.
- When a process wishes to enter CS, it periodically checks other’s flag until that flag is false (other process is not in CS)
- The process sets its own flag true and enters CS.
- When it leaves CS, it sets its flag to false.

Algorithm 2…

- initially **flag [0] = flag [1] = false.**
- **P0** **P1**

P0	P1
while (flag[1]) ;	while (flag[0])
/ Do nothing */*	*/* Do nothing */*
flag[0] = true;	flag[1] = true;
critical section	*critical section*
flag[0] = false;	flag[1] = false;

- Mutual exclusion is satisfied.
- If one process fails outside CS the other process is not blocked.
- Sometimes, the solution is worst than previous solution.
  - It does not even **guarantee ME.**
    - ✔ P0 executes the **while** statement and fins flag[1] set to false.
    - ✔ P1 executes the **while** statement and fins flag[0] set to false.
    - ✔ P0 sets flag[0] to true and enters its CS.
    - ✔ P1 sets flag[1] to true and enters its CS.

Algorithm 3

- Interchange the first two statements.
- Busy Flag Modified

P0	**P1**
flag[0] = true;	flag[1] = true;
while (flag[1]);	while (flag[0]);
/ Do nothing */*	*/* Do nothing */*
critical section	*critical section*
flag[0] = false;	flag[1] = false;

- Guarantees ME
- Both processes set their flags to true before either has executed the **while** statement, then each will think the other has entered CS causing deadlock.

Correct solution (1)

- Combining the key ideas of previous algorithms
- Dekker's Algorithm
  - Use *flags* for mutual exclusion, *turn* variable to break deadlock
  - Handles mutual exclusion, deadlock, and starvation
- Dekker's Algorithm
- Initial state: flag[0]=flag[1]=false; turn=1

P0

```
flag[0] = true;

while ( flag[1] )
    if (turn==1)
    {
        flag[0]=false;
        while (turn==1)
            /* do nothing */
        flag[0]=true;
    }
/* critical section */
turn=1;
flag[0] = false;
remainder section
```

P1

```
flag[1] = true;

while ( flag[0])
    if  (turn==0)
    {
        flag[1]=false;
        while (turn==0)
            /* do nothing */
        flag[1]=true;
    }
/* critical  section */
turn=0;
flag[1] = false;
remainder section
```

Correct solution (2)

- Peterson's Algorithm
- Initial state: flag[0]=flag[1]=false;

P0	**P1**
flag[0] = true;	flag[1] = true;
turn = 1;	turn = 0;
while (flag[1] && turn==1)	while (flag[0] && turn==0)
/* Do Nothing */;	/* Do nothing */;
critical section	*critical section*
flag[0] = false;	flag[1] = false;
remainder section	***remainder section***

Correct solution

- We need to show that
  - ME is preserved
  - The progress requirement is satisfied
  - The bounded-waiting requirement is met.
- **ME is preserved**
  - If both processes enter the CS both flad[0]==flag[1]==true
  - Both could not execute while loop successfully as turn is either 0 or 1.
- **Progress.**
  - While P1 exits CS it sets flag[1]=false, allowing P0 to enter CS.
  - P1 and P0 will enter the CS (Progress)
- **Bounded waiting:** P1 will enter the CS after at most one entry by P0 and vice versa.

Multi-process solution: Bakery Algorithm

Critical section for n processes

- Based on scheduling algorithm commonly used in bakeries.
  - On entering the store the customer receives the number.
  - The customer with the lowest number is served.
  - Customers may receive the same number, then the process with the lowest name is served first.
- 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., 1,2,3,3,3,3,4,5...

Bakery Algorithm

- var: choosing: array[0…n-1] of boolean.
- 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 $k \geq a_i$ for $i = 0, \ldots, n-1$
- 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);
```

- **Consider Pi in its CS and Pk is trying to enter CS**
- When Pk enters second while statement for j=i, it finds that
  - number[i] $\neq$ 0
  - (number[i],i) < (number[k].k)
  - So it waits until Pi leaves CS
- FCFS is followed.

Mutual exclusion: hardware solution

- In the uni-processor system, it is sufficient to prevent a process from being interrupted.

```
while (true){
/* disable interrupts */
/* Critical section */
/* enable interrupts */
/* remainder */
}
```

- **Since CS can not be interrupted ME is guaranteed.**
- **The efficiency decreases**
- **It can not work in multi-processor environments**
  - **More than one process is executing at a time.**

Special machine instructions

- In multi-processor configuration, several processes share access to a common main memory.
- At the hardware level, access to a memory location excludes any other access to that same memory location.
- Processor designers have proposed several machine instructions to carry out two actions atomically (single cycle).
  - **Reading and writing**
  - **swapping**

Test and set instruction

- Test and modify the content of a word atomically

```
boolean testset  (int    i)
{
 if (i==0)
   {
       i=1;
       return true;
   }
   else
     {
       return false
     }
}
```

- This instruction sets the value of ‘i’, if the value=0 and returns true. Otherwise the value is not changed and false is returned.

Mutual Exclusion with Test-and-Set

- Shared data:

  boolean lock = false;

- void *P(int i)*

```
do {
        while (TestAndSet(lock)==false)
            /* do nothing*/;
            critical section
        lock = false;
            remainder section
    }

void main()
{
  lock=false;
  parbegin(P1(), P(2),….,P(n));
}
```

Test-and-Set: Correctness

- **Mutual exclusion**
  - A shared variable lock is set to false
  - The only process Pi that enters CS that finds lock as false and sets it to true.
  - All other processes trying to enter CS so   into a busy waiting mode and finds lock as false.
  - When process leaves C it resents lock to false.
  - When Pi exits lock is set to false so the next process Pj to execute instruction find test-and-set=false and will enter the CS.
- **Progress**
  - **Trivially true**
- **Unbounded waiting**
  - **Possible since depending on the timing of evaluating the test-and-set primitive.**
  - **Does not guarantee fairness.**

Swap instruction

- 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];
```

- Process P_i

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

SWAP: Correctness

- Similar to Test-and-set
- Mutual exclusion
- Progress
  - Trivially true
- Unbounded waiting
  - Possible since depending on the timing of evaluating the test-and-set primitive.
  - Does not guarantee fairness.

Can we get bounded waiting ?

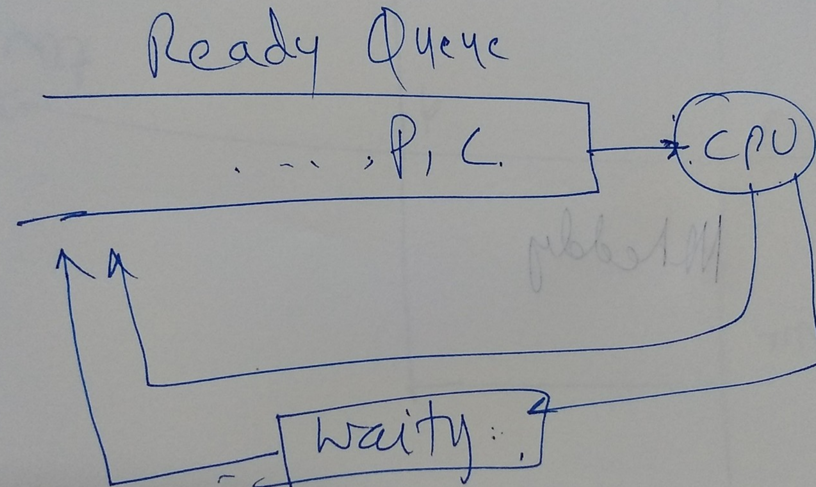
- **Introduce a boolean array called waiting of size n and boolean variable key**
- **Entry**
  - **waiting[i]:=true;**
  - **key:=true;**
  - **while (waiting[i] and key) do**
    - **key := test-and-set(lock);**
  - **waiting[i]:=false;**
  - **execute CRITICAL SECTION**
- **Exit**
  - **Find the next process j that has waiting[j]=1 stepping through waiting.**
  - **Set waiting[j]:=false;**
  - **Process P_j immediately enter the CS.**
  - **If no process exists, set lock=false;**

Can we get bounded waiting ?....

- Every (interested) Pi executes that test&set at least once.
- Pi enters the critical section provided:
  - Key is false in which case there is no process in CS.
- Or
  - If it was waiting, because waiting[i] was reset to false by the unique process that was blocking it in the critical section.
  - Either of the above events occur exactly once and hence mutual exclusion.

Properties of machine instruction approach

- +ve
  - Any number of processes
  - Simple and easy
  - Can support multiple CSs.
- -ve
  - **Busy waiting** **is employed**
    - **The process is waiting and consuming processor time.**
  - Starvation is possible.
    - The selection of waiting process is arbitrary.
  - Deadlock is possible due to priority
    - P1 enters CS and interrupted by higher priority process P2 which is trying to enter CS.
    - P2 can not get CS unless P1 is out and P1 can not be dispatched due to low priority.


Ready Queue
. . . , P, C.
CPU
waity

Mutex Locks

- Test-set locks are also called mutex llocks
- Mutex means mutual exclusion.

Semaphores: Dijkstra; 1965

- Two and more processes can cooperate by means of simple signals, such that a process is forced to stop at a specified place until it has received a specific signal.
- For signaling, special variables called semaphores are used
- A semaphore is a synchronization tool.
- A semaphore is an integer variable that is accessed only through two standard atomic operations: **wait and signal**.
- To transmit a signal to semaphore S, a process executes the primitive *signal(S)* primitive.
- To receive a signal via semaphore S, the process executes *wait(S)* primitive.

Semaphores: Dijkstra 1965
Classical or first definition

- A semaphore is initialized to a non-negative value
- The **wait** operation decrements the semaphore value. If the integer value is negative the process waits.
- The **signal** operation increments the semaphore value. If the value is not positive, then process which is blocked by a wait operation is gets the access to CS.
- **The wait and signal are assumed to be atomic.**
- Semaphore *S* – integer variable
- can only be accessed via two indivisible (atomic) operations

wait (*S*):
while $S \leq 0$ **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);

- **Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly.**

Semaphore Implementation

- The classical definition requires busy waiting.
- While a process is in CS, the other process must loop continuously in the entry code.
- Busy waiting wastes CPU cycles.
- This type of semaphore is called **spinlock**: process spins while waiting for a lock.
  - Advantage of spinlock: no context switch
  - When locks are expected to be held for short times, spinlocks are useful.
- To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations.
- If a process executes wait operation and finds the semaphore operation is not positive, it must wait.
  - Rather than busy waiting it must **block** itself.
  - The **block** operation puts the process into waiting queue of semaphore and process is switched to waiting state.
- A process that is blocked waiting on a semaphore S, should be restarted when some other process executes signal operation.
- The process is restarted with **wakeup** operation.

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);
  }
  ```

- **Wait and signal operations are system calls.**

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);
⋮	⋮
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.

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.

Binary Semaphores

- A binary semaphore is a semaphore with an integer value that can range only between 0 and 1
- It is simple to implement.
- Type binary semaphore =**record**

  value:(0,1)

  queue: list of processes

  end;
- var s: binary semaphore
- **waitB(s):**

  If s.value=1 **then**

  s.value=0

  else

  begin

  place this process in s.queue;

  block this process;

  end;
- **signalB(s):**

  If s.queue is empty **then**

  s.value=1

  else

  begin

  remove (wakeup) the process from s.queue;

  place this process in the ready list.

  end;

Implementing *S* as a Binary Semaphore

- Can implement a counting semaphore *S* as a binary semaphore.
- Data structures:
  binary-semaphore S1, S2;
  int C:
- Initialization:
  S1 = 1
  S2 = 0
  C = initial value of semaphore **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);
```

Counting semaphores

```
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);
       }
```

Classical Problems of Synchronization

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

Bounded-Buffer Problem

- Used to to illustrate the power of synchronization techniques
- We assume that the buffer consists of n buffers, each capable of holding an item.
- The mutex semaphore provides mutual exclusion access to buffer which is initialized to the value 1.
- The **empty** and **full** semaphores count the number of empty and full buffers which are initialized to n and zero respectively.
- Shared data
  semaphore full, empty, mutex;
- Initially:
  full = 0, empty = n, mutex = 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);
```

- **Producer is producing full buffers for the consumer and consumer is producing empty buffers for the consumer.**

Readers-Writers Problem

- Problem: A data object (file or record) is shared among several concurrent processes.
  - Some want to read and others want to update it.
- **Readers:** processes interested in reading.
- **Writers:** processes interested in writing.
- Two readers can access shared data object simultaneously.
- But a writer and reader can access shared data object simultaneously
  - problems may occur!
- To protect from these problems, writers should have an exclusive access to the shared object.
- This synchronization problem is referred to as readers-writers problem.
- It is a different kind of synchronization problem.
- The readers-writers problem has several variations.
  - Simple one: No reader will be kept waiting unless writer has obtained permission to write.

Readers-Writers Problem

- Semaphores used: **mutex** and **wrt**
- The semaphore **wrt** is common to reader and writer.
- Semaphore **mutex** is used to update **readcount**.
- **readcount** keeps track of how many are reading the object.

- Shared data

  semaphore mutex, wrt;

  Initially

  mutex = 1, wrt = 1, readcount = 0

Readers-Writers Problem Writer Process

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

Readers-Writers Problem Reader Process

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

- **Writers can be starved if there is a continuous sequence of readers.**

Readers-Writers Problem

- Can the producer/consumer problem is considered as case of the readers/writers problem with a writer is a producer and reader is a consumer ?

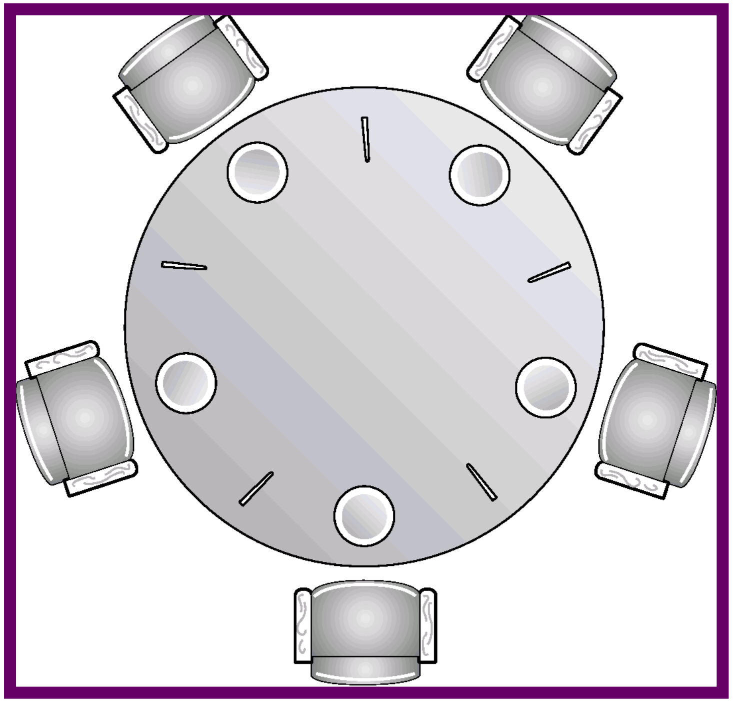
- The answer is no

- The producer is not just a writer
  - It must read queue of pointers to determine where to write the next item and it must determine if the buffer is full.

- Similarly the consumer is not a reader
  - It must adjust queue pointers to show that it has removed a unit from the buffer.

Dining-Philosophers Problem

- Five philosophers spend their lives on thinking and eating.
- They share a common circular table surrounded by five chairs.
- Five single chopsticks are available.
- Whenever a philosopher wants to eat, he tries to pick up two chopsticks that are closest to him/her.
- A philosopher can not pick the chopstick in the hand of neighbor.
- After finishing, the philosopher puts back the chopsticks and starts thinking.



- It is simple representation of the need to allocate several resources among several processes in a **deadlock and starvation free manner.**

Dining-Philosophers Problem

- Shared data

  semaphore chopstick[5];
  Initially all values are 1

- Philosopher *i*:

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

- **The solution creates a deadlock**

Barbershop Problem

- 3 barbers, each with a barber chair
  - Haircuts may take varying amounts of time
- Sofa can hold 4 customers, max of 20 in shop
- Customers wait outside if necessary
- When a chair is empty:
  - Customer sitting longest on sofa is served
  - Customer standing the longest sits down
- After haircut, go to cashier for payment
  - Only one cash register
  - Algorithm has a separate cashier, but often barbers also take payment
    - ✔ This is also a critical section

Barbershop Problem

- The main body of the program activates 50 customers, 3 barbers, and the cashier process. Synchronization operators.
  - Shop and sofa capacity: the capacity of shop and the capacity of the sofa are governed by the semaphores **max_capacity** and **sofa.**
    - When customer enters max_capacity decremented by one.
    - When a customer leaves it is incremented.
    - Wait and signal operations are surround the actions of sitting and getting_up from sofa.
  - **Barber chair capacity:**
    - There are three barber chairs; the semaphore barber_chair assures that no more than three customers attempt to obtain service at a time.
    - A customer will not get up from the sofa until at least one chair is free.
  - **Ensuring customers are in the barber chair:** The semaphore cust_ready provides a wakeup signal for a sleeping barber indicating that the customer has just taken the chair.
  - **Holding customers in barber chair:** once seated the customer remain in the chair until the barber gives the signal that haircut is complete, using the semaphore finished.
  - **Limiting one customer to a barber chair:** the semaphore barber_chair is intended to limit the number of customers in barber chairs to three. The semaphore leave_b_chair is used to synchronize sitting.
  - **Paying and receiving:** payment and receipt semaphores are used   to synchronize the operations.
  - **Coordinating barber and cashier functions:** To save money the barber shop does not employ a separate cashier. Each barber is required to perform that task when not cutting hair. The semaphore coord ensures the barbers perform only one task at a time.

Barbershop Problem

Semaphore	Wait operation	Signal operation
max_capacity	Customer waits for a room to enter shop.	Exiting customer signals customer waiting to enter
sofa	Customer waits for seat on sofa	Customer leaving sofa signals customer waiting for sofa
barber_chair	Customer waits for empty barber chair	Barber signals when that barber's chair is empty
Cust_read	Barber waits until customer is in the chair	Customer signals barber that customer is in the chair
finished	Customer waits until his haircut is complete.	Barber signals when done cutting hair of his customer.
leave_b_chair	Barber waits until customer gets up from the chair	Customer signals barber when customer gets up from chair.
payment	Cashier waits for a customer to pay	Customer signals cashier that he has paid.
receipt	Customer waits for a receipt for a payment	Cashier signals that payment has been accepted.
coord	Wait for a barber resource to be free to be free perform either the hair cutting or cashiering function.	Signal that a barber resource is free.

Barbershop

```
program    barbershop1;
var        max_capacity: semaphore (:=20);
           sofa: semaphore (:=4);
           barber_chair, coord: semaphore (:=3);
           cust_ready, leave_b_chair, payment, receipt: semaphore (:=0)
```

```
procedure customer;
var custnr: integer;
begin
   wait (max_capacity );
   enter shop;
   wait( sofa );
   sit on sofa;
   wait( barber_chair );
   get up from sofa;
   signal( sofa );
   sit in barber chair;
   wait( mutex2 );
  signal( cust_ready );
   wait( finished[custnr] );
   leave barber chair;
   signal( leave_b_chair );
   pay;
   signal( payment );
   wait( receipt );
   exit shop;
   signal( max_capacity );
end;
```

```
procedure barber;
var b_cust: integer
begin
   repeat
  wait( cust_ready );
      wait( coord );
cut hair;
signal( coord );
      signal( finsihed[b_cust] );
      wait( leave_b_chair );
      signal( barber_chair );
   forever
end;
```

```
procedure cashier;
begin
   repeat
wait( payment );
wait( coord );
      accept payment;
      signal( coord );
      signal( receipt );
   forever
end;
```

```
Void main()
{
 count=0;
Parbegin {customer… 50 times,…customer,
Barber, barber,barber, cashier)
}
```

```
/* program barbershop1 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3;
semaphore coord = 3;
semaphore cust_ready = 0, finished = 0, leave_b_chair = 0, payment= 0, receipt = 0;

void customer ()
{
    wait(max_capacity);
    enter_shop();
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair;
    signal(cust_ready);
    wait(finished);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity)
}

void barber()
{
    while (true)
    {
        wait(cust_ready);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}

void cashier()
{
    while (true)
    {   wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}

void main()
{
    parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber,          cashier);
}
```

Figure 16: An unfair barbershop

Another method to avoid the unfairness is to number the barber chairs so that less semaphores are needed, but how? Think about it!

Barbershop Problem

- The preceding solution is unfair.
- The customers are servers in the order they enter the shop.
- If one barber is very fast and one of the customer is quite bald.
- The problem can be solved with more semaphores.
  - We assign unique customer number is to each customer.
  - The semaphore mutex1 protects access to global variable count.
- The semaphore finished is refined to be an array of 50 semaphores.
  - Once a customer seated in a barber chair, he executes wait(finished[custnt]) to wait in his own unique semaphore.
- Please see the solution in Willium Stallling book (pp 229-234)

Fair Barbershop

```
program         barbershop2;
var             max_capacity: semaphore (:=20);
                sofa: semaphore (:=4);
                barber_chair, coord: semaphore (:=3);
                mutex1, mutex2: semaphore (:=1);
                cust_ready, leave_b_chair, payment, receipt: semaphore (:=0)
                finished: array [1..50] of semaphore (:=0);
                count: integer;
```

```
procedure customer;
var custnr: integer;
begin
   wait (max_capacity );
   enter shop;
   wait( mutex1 );
   count := count + 1;
   custnr := count;
   signal( mutex1 );
   wait( sofa );
   sit on sofa;
   wait( barber_chair );
   get up from sofa;
   signal( sofa );
   sit in barber chair;
   wait( mutex2 );
   enqueue1( custnr );
   signal( cust_ready );
   signal( mutex2 );
   wait( finished[custnr] );
   leave barber chair;
   signal( leave_b_chair );
   pay;
   signal( payment );
   wait( receipt );
   exit shop;
   signal( max_capacity );
end;
```

```
procedure barber;
var b_cust: integer
begin
   repeat
      wait( cust_ready );
      wait( mutex2 );
      dequeue1( b_cust );
      signal( mutex2 );
      wait( coord );
      cut hair;
      signal( coord );
      signal( finsihed[b_cust] );
      wait( leave_b_chair );
      signal( barber_chair );
   forever
end;
```

```
procedure cashier;
begin
   repeat
      wait( payment );
      wait( coord );
      accept payment;
      signal( coord );
      signal( receipt );
   forever
end;
```

```
Void main()
{
 count=0;
Parbegin {customer… 50 times,…customer,
Barber, barber,barber, cashier)
}
```

```
/* program barbershop2 */
semaphore max_capacity = 20;
semaphore sofa = 4;
semaphore barber_chair = 3, coord = 3;
semaphore mutex1 = 1, mutex2 = 1;
semaphore cust_ready = 0, leave_b_chair = 0, payment = 0, receipt = 0;
semaphore finished [50] = {0};
int count;

void customer()
{
    int custnr;
    wait(max_capacity);
    enter_shop();
    wait(mutex1);
    count++;
    custnr = count;
    signal(mutex1);
    wait(sofa);
    sit_on_sofa();
    wait(barber_chair);
    get_up_from_sofa();
    signal(sofa);
    sit_in_barber_chair();
    wait(mutex2);
    enqueue1(custnr);
    signal(cust_ready);
    signal(mutex2);
    wait(finished[custnr]);
    leave_barber_chair();
    signal(leave_b_chair);
    pay();
    signal(payment);
    wait(receipt);
    exit_shop();
    signal(max_capacity)
}

void barber()
{
    int b_cust;
    while (true)
    {
        wait(cust_ready);
        wait(mutex2);
        dequeue1(b_cust);
        signal(mutex2);
        wait(coord);
        cut_hair();
        signal(coord);
        signal(finished[b_cust]);
        wait(leave_b_chair);
        signal(barber_chair);
    }
}

void cashier()
{
    while (true)
    {
        wait(payment);
        wait(coord);
        accept_pay();
        signal(coord);
        signal(receipt);
    }
}

void main()
{   count := 0;
    parbegin (customer, . . . 50 times, . . . customer, barber, barber, barber,
        cashier);
}
```

Figure 17: An fair barbershop

Implementing wait() and signal() in Multi-processor Systems

- Disabling interrupts will not work.

- Spinlock is the solution
  - With this we have moved busy waiting from entry section to critical sections of application programs.

Implementation of Semaphores

- wait and signal operations are atomic.
- No two processes should execute wait and signal on the same semaphore at the same time
- Good Solution: implement through hardware or firmware.
- Other solutions
  - Ensure that only process manipulates “wait” and “signal” operations.
  - One can use Dekker’s algorithm or Peterson’s algorithm
    - ✔ Substantial processing overhead
  - Use one of the hardware supported schemes
    - ✔ Test and set
    - ✔ disabling interrupts (single processor)

About busy waiting

- Note: we have not eliminated the busy waiting with wait() and signal() completely.
  - Moved busy waiting from entry section to critical sections of application programs.
  - Furthermore, we have limited busy waiting to critical sections of wait() and signal() operations.
  - The wait and signal code is very short the amount of busy waiting involved is short.

Two possible implementations of Semaphores

```
Wait(s)
{
   while(!testset(s.flag)
      /* do nothing */
    s.count--;
    if  (s.count <0)
      {
       place this process in s.queue;
       block this process (set s.flag to 0)
      }
    else
      s.flag=0;
 }
```

```
Signal(s)
{
   while(!testset(s.flag)
      /* do nothing */
    s.count++;
    if  (s.count <= 0)
      {
       remove a process P from  s.queue;
       Place a process P in the ready list
      }
      s.flag=0;
 }
```

With TestSet Instruction

```
Wait(s)
{
   Inhibit interrupts
    s.count--;
    if  (s.count <0)
      {
       place this process in s.queue;
       block this process allow interrupts
      }
    else
      allow interrupts;
 }
```

```
Signal(s)
{
   Inhibit interrupts;
    s.count++;
    if  (s.count <= 0)
      {
       remove a process P from  s.queue;
       Place a process P in the ready list
      }
      allow interrupts;
 }
```

With Interrupts

Problem with semaphores

- Incorrect use may result in timing errors
- These errors are difficult to detect as these occur if only particular sequence occurs.
- Missing or reverse order.
- It is difficult to produce correct program using semaphores.
- The wait and signal operations are scattered throughout the program and it is difficult to see overall effect of these operations on the semaphores.

Problem with semaphores (cont.)

- Problems
  - Suppose a process interchanges the order in which wait and signal operations on the semaphore are executed

    signal (mutex)
    CS
    wait (mutex)

    - ✔ Several processes may be executing in their CS simultaneously.
  - Suppose that a process replaces signal(mutex) with wait(mutex)

signal (mutex)	wait(mutex)
CS	CS
signal (mutex)	wait(mutex)

    - ✔ Deadlock will occur.
  - Suppose a process omits wait(mutex) or signal(mutex) or both.
    - ✔ ME is violated or deadlock occurs.
- A critical region and monitor concept is introduced to address this problem

High level synchronization constructs

- To deal with the type of errors caused by semaphores, high-level constructs have been introduced.
  - Critical region
  - Monitor
- Assumption
  - Process contains some local data
  - Local data can be accessed by only the sequential program that is encapsulated within the same process.
  - Process can not access the local data of another process.

Monitors

- It is an high level synchronization construct.
- A monitor is a software module consisting of one or more procedures, an initialization sequence and local data.
- Main characteristics
  1. The local data variables are accessible only by the monitor’s procedures and not by any external procedure.
  2. A process enters monitor by entering one of its procedures.
  3. Only one process may be executing (active) in the monitor at any time; any other process that has invoked the monitor is suspended while waiting for the monitor to become available.
- 1&2 → object oriented characteristics.
- By enforcing one procedure at a time, the monitor enforces ME facility.

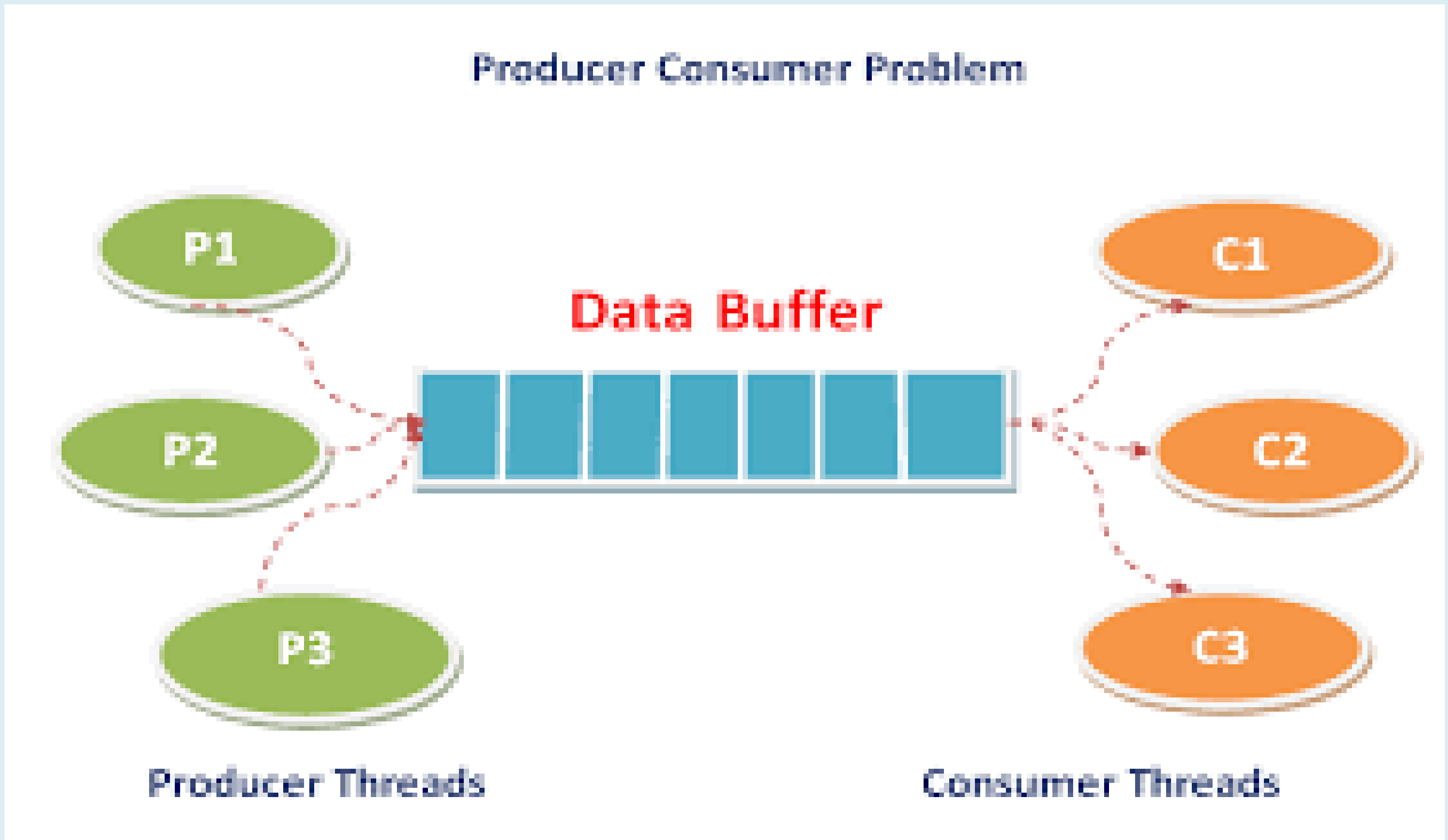
Monitors

- High-level synchronization construct that allows the safe sharing of an abstract data type among concurrent processes.
- A shared data resource can be protected by placing in the monitor.

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

Two kinds of waiting:

- Mutual exclusion waiting: to avoid race condition
- Conditional waiting: to avoid inconsistency



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

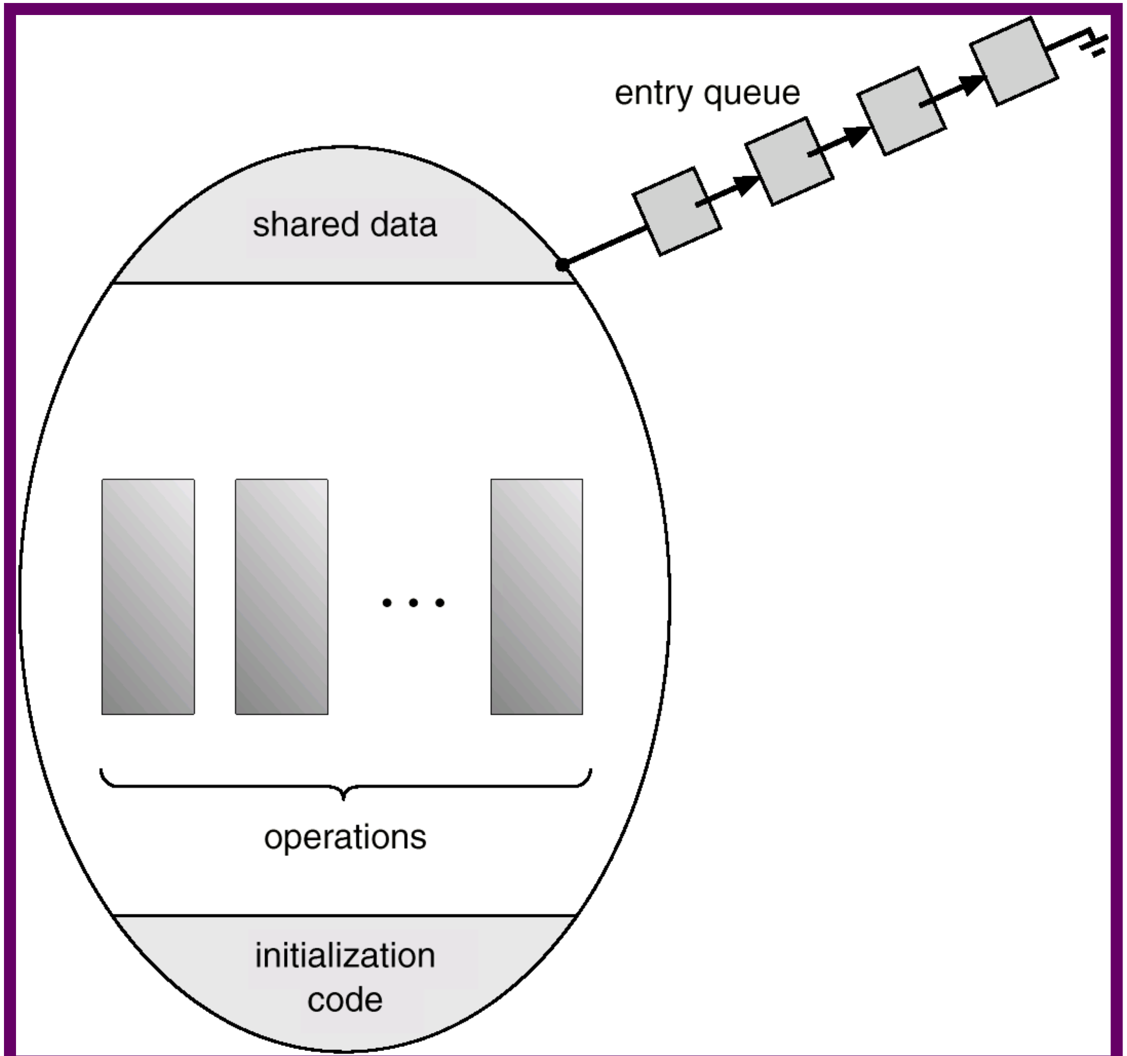
    x.wait();

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

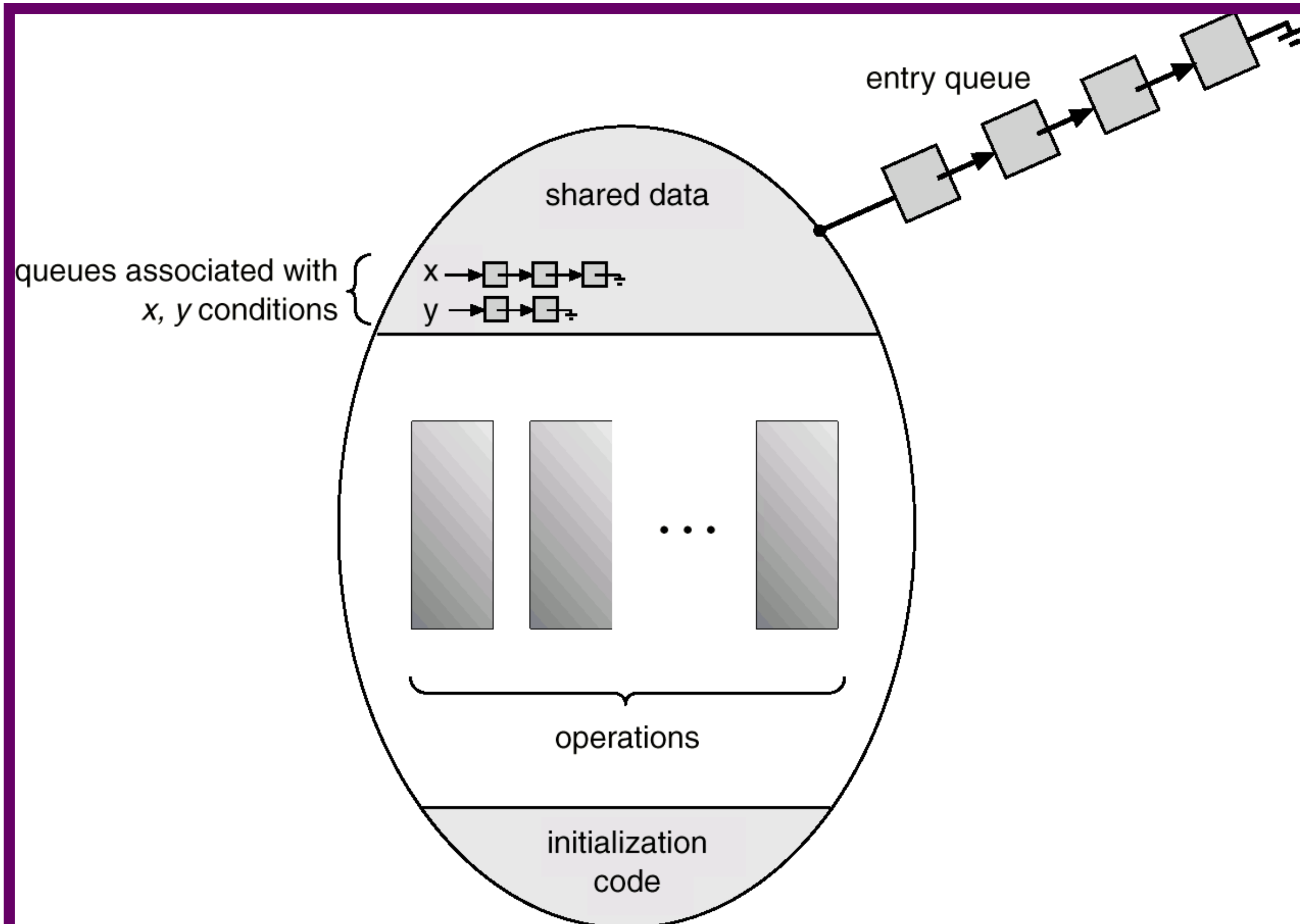
    x.signal();

  - 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



Monitors

- In case of monitors, the monitor construct itself provides ME, but synchronization is provided by the programmer.
- In case of semaphore, both ME and synchronization are provided by the programmer.
- Also , in case of monitors also, it is possible to make mistakes in the synchronization of monitors.
- For example if csignal function is omitted, the processes entering corresponding queue are permanently hung up.
- However, since all synchronization functions are confined to monitor, it is easier to verify the synchronization and detect bugs.
- Once a monitor is correctly programmed, access to the protected resource is correct from all processes.
- With semaphores, resources access is correct only if all of the processes that access the resource are programmed correctly.

Dining Philosophers Example
Deadlock free solution

- A philosopher is allowed to pick up his chopsticks only if both of them were available.
- We introduce three states:
  - 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 declare condition self[5];
  - Philosopher i can delay himself when he is hungry, but unable to obtain the chopsticks he needs.

Dining Philosophers Example
Deadlock free solution

- 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;
        }
}
```

Dining Philosophers

```
void pickup(int i) {
        state[i] = hungry;
        test[i]; // if left and right of i are not eating, then eat.
        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();
        }
}
```

Dining Philosophers

```
dp.pickup[i];
….
eat
….
dp.putdown(i);
```

- It is easy to show that no two neighbors are eating simultaneously and no deadlocks will occur.
- However, it is possible for a philosopher to starve to death.

Monitor Implementation Using Semaphores

- Variables
  semaphore mutex; // (initially = 1)
  semaphore next; // (initially = 0)
  int next-count = 0;
  // number of processes suspended on next.
- 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

- Solaris2 OS is designed to provide real-time computing, be multithreaded and support multiple processors.
- To control access to critical regions Solaris 2 implements
  - Adaptive mutexes
  - Condition variables
  - Semaphores
  - Reader-writer locks
  - Turnstiles
- Adaptive mutex protects access to every critical data item.
  - On multiprocessor system an adaptive mutex starts as a standard semaphore implemented as a spinlock.
  - Adaptive mutex is used to protect only data that are accessed by short-code segments (few hundred instructions).
- For longer code segments, condition variables and semaphores are used.
- Reader-writers locks are used to access data that is accessed frequently in read-only manner.
  - Semaphores serialize the access
  - When there are many readers and few writers r-w locks are efficient.
- Solaris 2 uses turnstiles to order list of threads waiting to acquire either an adaptive mutex or a reader-writer lock.

Solaris 2 Synchronization

- Uses *turnstiles* to order the list of threads waiting to acquire either an adaptive mutex or reader-writer lock.
  - It is a queue structure containing threads blocked in a lock.
- To prevent a priority inversion, turnstiles are organized into priority inheritance protocol.
  - When a lower priority thread holds a lock, and higher priority thread blocks, the lower priority thread inherits the priority of the higher-priority thread.

Windows 2000 Synchronization

- Multi-threaded kernel
  - Real-time applications and multiple processors
- Uses interrupt masks to protect access to global resources on uni-processor systems.
- Uses *spinlocks* on multiprocessor systems.
  - Kernel ensures that a thread will never be preempted while holding a spinlock.
- Also provides *dispatcher objects* which may act as mutexes and semaphores and events.
- Dispatcher objects may also provide *events*. An event acts much like a condition variable which may notify a waiting thread when a desired condition occurs.

Transactional Memory

- Multi-core applications increases risk of race conditions and deadlocks.
- Techniques proposed: locks, semaphores and monitors
- Transactional memory provides alternative strategy
- A memory transaction is a sequence of memory read and write transactions that are atomic. If all the operations are completed the memory transaction is committed. Otherwise, the operations must be aborted. Such a feature can be added to programming language.
- Traditional way
  - Update () {

    Acquire();

    /* Modify shared data */

    release(); }
- Suppose we add an atomic operation
  - Update() {

    Atomic { }

    }

Atomic transactions

- A group pf statements should be executed as a logical unit.
- More than mutual exclusion
- Notion of transaction has been emerged
  - ACID properties
  - Atomicity, consistency, Isolation and durability
- Atomicity: all or nothing
- Consistency: one consistent state to another consistent state
- Isolation: Parallel execution is serial
- Durability: Changes are permanent
- Two-phase locking and log based recovery methods are followed.
- Will be studied in database systems course

Transactional Memory

- Advantage
  - System is responsible for guaranteeing the atomicity.
- Transactional memory systems can be implemented in either software or in software.
  - STM: software transactional memory
    - ✔ appropriate code is inserted by compiler
  - HTM: Hardware transactional memory
    - ✔ Uses cache coherency protocols to support transaction memory

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***.
- The expression B is Boolean expression which governs the access to the critical region.

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*.
- If the following two statements are executed concurrently, it will be equivalent to the serial execution “S1 followed by S2” or “S2 followed by S1”.
  - Region v when (true) S1;
  - Region v when (true) S2;
- CR construct guards against simple errors associated with the semaphore solution.

Example – Bounded Buffer

- Shared data:

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

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--;
}
```

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

- Can be implemented using semaphores.

Implementing *S* as a Binary Semaphore

- Can implement a counting semaphore *S* as a binary semaphore.
- Data structures:
  binary-semaphore S1, S2;
  int C:
- Initialization:
  S1 = 1
  S2 = 0
  C = initial value of semaphore **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);
```

Counting semaphores

```
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);
        }
```