Semaphore, Monitor, CCR,…

Semaphores

Semaphores

- 1965 Dijkstra
- In design of OS, we need cooperate sequential processes
- We need efficient and reliable mechanisms for supporting cooperation

Semaphores

- Synchronization tool (provided by the OS) that do not require busy waiting
- A semaphore S is an integer variable that, apart from initialization, can only be accessed through 2 atomic and mutually exclusive operations:
  - P(S)
  - V(S)
- To avoid busy waiting: when a process has to wait, it will be put in a blocked queue of processes waiting for the same event

Semaphores

- Hence, in fact, a semaphore is a record (structure):

```
type semaphore = record
                   count: integer;
                   queue: list of process
                 end;
var S: semaphore;
```

  - When a process must wait for a semaphore S, it is blocked and put on the semaphore’s queue
  - The signal operation removes (according to a fair policy like FIFO) one process from the queue and puts it in the list of ready processes

Semaphore's operations

```
P(S):
  S.count--;
  if (S.count<0) {
    block this process
    place this process in S.queue
  }

V(S):
  S.count++;
  if (S.count<=0) {
    remove a process P from S.queue
    place this process P on ready list
  }
```

`S.count must be initialized to a nonnegative value (depending on application)`

Semaphores: observations

- When S.count >=0: the number of processes that can execute P(S) without being blocked = S.count
- When S.count<0: the number of processes waiting on S is = |S.count|
- Atomicity and mutual exclusion: no 2 process can be in P(S) and V(S) (on the same S) at the same time (even with multiple CPUs)
- Hence the blocks of code defining P(S) and V(S) are, in fact, critical sections

Using semaphores for solving critical section problems

- For n processes
- Initialize S.count to 1
- Then only 1 process is allowed into CS (mutual exclusion)
- To allow k processes into CS, we initialize S.count to k

```
Process Pi:
repeat
  P(S);
   CS
  V(S);
   RS
forever
```

Using semaphores to synchronize processes

- We have 2 processes: P1 and P2
- Statement S1 in P1 needs to be performed before statement S2 in P2
- Then define a semaphore “synch”
- Initialize synch to 0

- Proper synchronization is achieved by having in P1:

  S1;

  V(synch);

- And having in P2:

  P(synch);

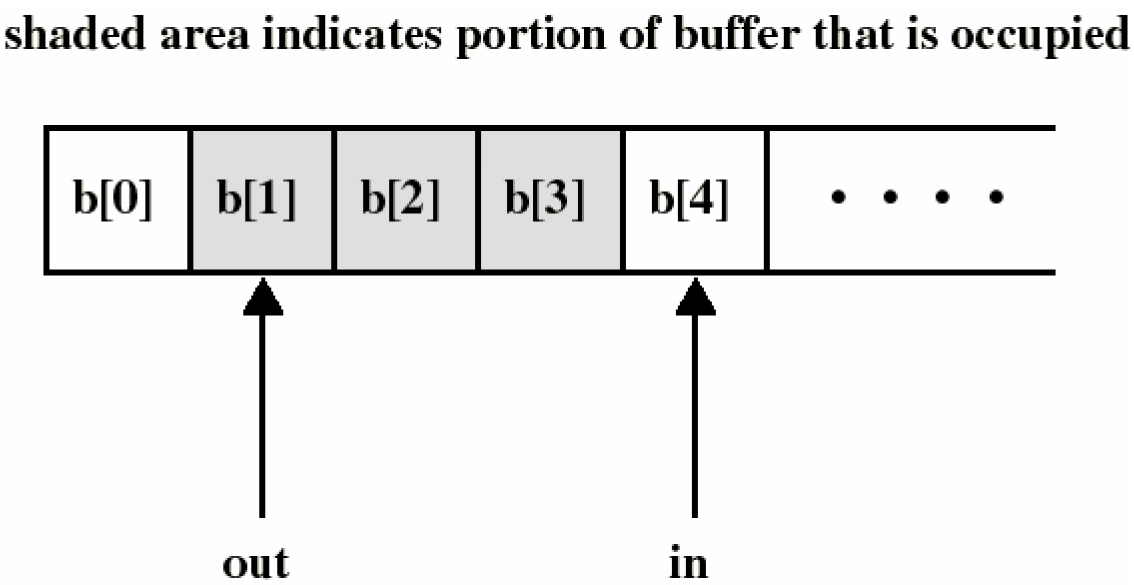
  S2;

The producer/consumer problem

- A producer process produces information that is consumed by a consumer process
  - Ex1: a print program produces characters that are consumed by a printer
  - Ex2: an assembler produces object modules that are consumed by a loader
- We need a buffer to hold items that are produced and eventually consumed
- A common paradigm for cooperating processes

P/C: unbounded buffer

- We assume first an unbounded buffer consisting of a linear array of elements
- `in` points to the next item to be produced
- `out` points to the next item to be consumed



P/C: unbounded buffer

- We need a semaphore S to perform mutual exclusion on the buffer: only 1 process at a time can access the buffer
- We need another semaphore N to synchronize producer and consumer on the number N (= in - out) of items in the buffer
  - an item can be consumed only after it has been created

Solution of P/C: unbounded buffer

```
Initialization:
  S.count:=1;
  N.count:=0;
  in:=out:=0;
```

```
append(v):
b[in]:=v;
in++;
```

```
take():
w:=b[out];
out++;
return w;
```

```
Producer:
repeat
  produce v;
  P(S);
  append(v);
  V(S);
  V(N);
forever
```

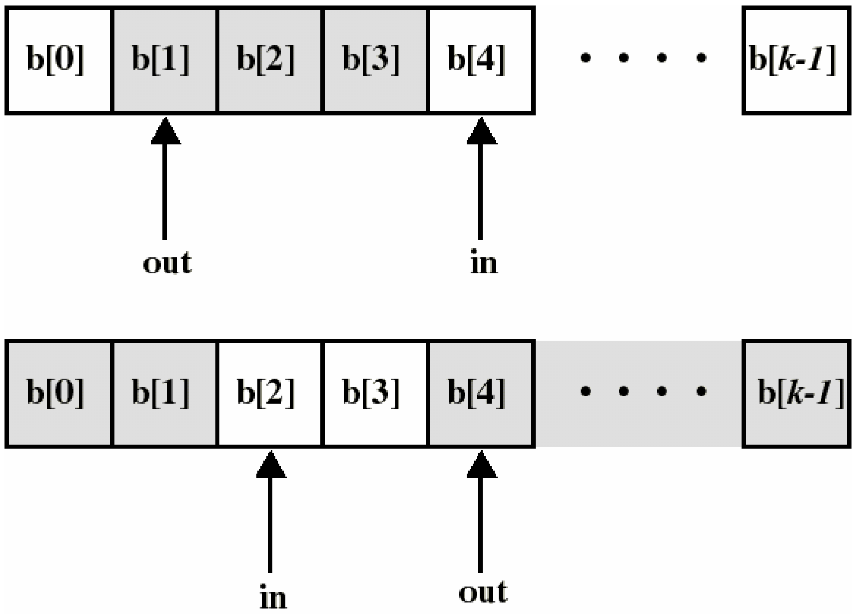
```
Consumer:
repeat
  P(N);
  P(S);
  w:=take();
  V(S);
  consume(w);
forever
```

- critical sections

P/C: unbounded buffer

- Remarks:
  - Putting V(N) inside the CS of the producer (instead of outside) has no effect since the consumer must always wait for both semaphores before proceeding
  - The consumer must perform P(N) before P(S), otherwise deadlock occurs if consumer enter CS while the buffer is empty
- *Using semaphores is a difficult art...*

P/C: finite circular buffer of size k



- can consume only when number N of (consumable) items is at least 1 (now: N!=in-out)
- can produce only when number E of empty spaces is at least 1

P/C: finite circular buffer of size k

- As before:
  - we need a semaphore S to have mutual exclusion on buffer access
  - we need a semaphore N to synchronize producer and consumer on the number of consumable items
- In addition:
  - we need a semaphore E to synchronize producer and consumer on the number of empty spaces
  - Producer cannot overflow the buffer

Solution of P/C: finite circular buffer of size k

```
Initialization: S.count:=1; in:=0;
                N.count:=0; out:=0;
                E.count:=k;
```

```
append(v):
b[in]:=v;
in:=(in+1)
     mod k;
```

```
take():
w:=b[out];
out:=(out+1)
      mod k;
return w;
```

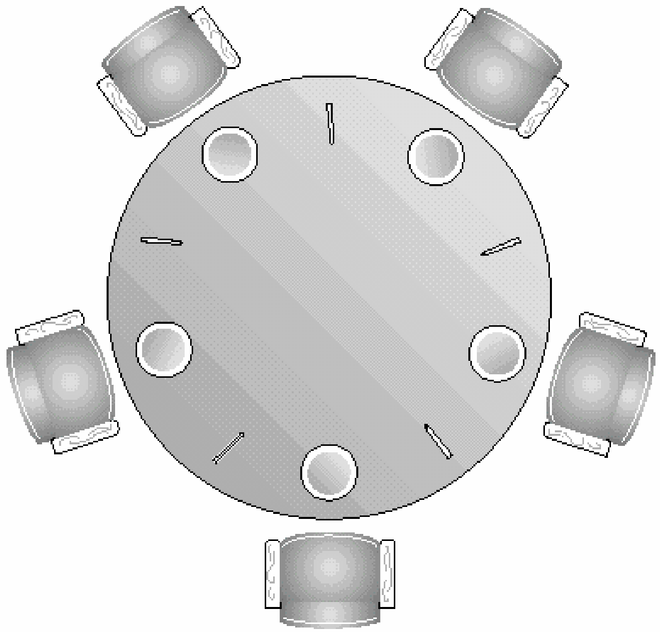
```
Producer:
repeat
  produce v;
  P(E);
  P(S);
  append(v);
  V(S);
  V(N);
forever
```

```
Consumer:
repeat
  P(N);
  P(S);
  w:=take();
  V(S);
  V(E);
  consume(w);
forever
```

critical sections

The Dining Philosophers Problem

- 5 philosophers who only eat and think
- each need to use 2 forks for eating
- we have only 5 forks
- A classical synchronization problem
- Illustrates the difficulty of allocating resources among process without deadlock and starvation



The Dining Philosophers Problem

- Each philosopher is a process
- One semaphore per fork:
  - fork: array[0..4] of semaphores
  - Initialization: fork[i].count:=1 for i:=0..4
- A first attempt:
- Deadlock if each philosopher start by picking his left fork!

```
Process Pi:
repeat
 think;
 P(fork[i]);
 P(fork[i+1 mod 5]);
 eat;
 V(fork[i+1 mod 5]);
 V(fork[i]);
forever
```

The Dining Philosophers Problem

- A solution: admit only 4 philosophers at a time that tries to eat
- Then 1 philosopher can always eat when the other 3 are holding 1 fork
- Hence, we can use another semaphore T that would limit at 4 the numb. of philosophers “sitting at the table”
- Initialize: T.count:=4

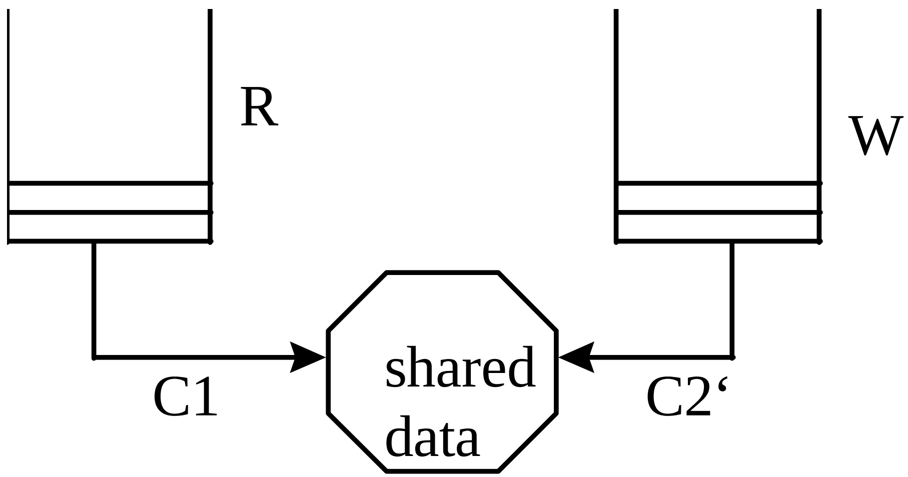
```
Process Pi:
repeat
 think;
 P(T);
 P(fork[i]);
 P(fork[i+1 mod 5]);
 eat;
 V(fork[i+1 mod 5]);
 V(fork[i]);
 V(T);
forever
```

Readers/Writers Problem

- Multiple processes wanting to read an item, and one or more needing to write
  - (Think of airline reservations…)
- Rather than enforce mutual exclusion on every access, use these rules:
  - Any number of readers may simultaneously read the file
  - Only one writer at a time may write to the file
  - If a writer is writing to the file, no reader may read it

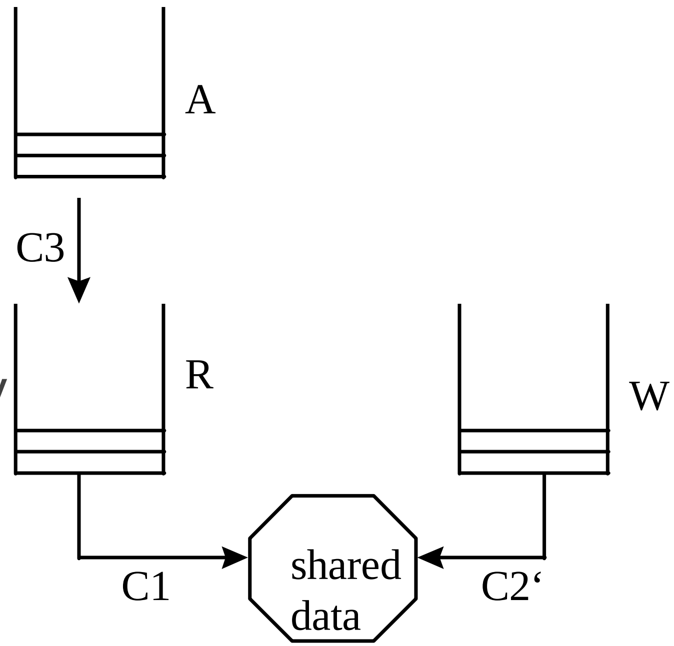
Reader priority

- “First readers/writers” problem (reader priority):
  - No reader will wait (for other readers to finish) even if a writer is waiting
- Rule 1: If writer is writing, nobody can read or write
- Conditions:
  - C1: no writer is writing (for writers)
  - C2: nobody is reading or writing (for readers)
- Writer starvation possible



Fair solution - Writer priority

- "Second reader/writers" problem (writer priority):
  - No new readers allowed once a writer has asked for access
- Rule 2: If readers are reading and at least one writers is waiting, new readers should wait
- Rule3: If writer finishes writing, all waiting readers can start read
- Three queues
- Condotions:
  - C1: no writer is writing
  - C2': nobody is reading and R is empty
  - C3: writer is writing or W is empty



Semaphore Solution to first Readers/Writers problem (readers priority)

- Keep a count of number of current readers (readcount = 0)
- Use two semaphores
  - 1. mutex: mutually exclusive access to readcount (initially 1)
  - 2. wrt: mutual exclusion for writers (initially 1)

```
Reader
P(mutex);
  readcount = readcount + 1;
  if (readcount == 1) then P(wrt);
V(mutext);
readTheFile();
P(mutex);
  readcount = readcount - 1;
  if (readcount == 0) then V(wrt);
V(mutex);
```

```
Writer
P(wrt);
writeTheFile();
V(wrt);
```

Binary semaphores

- The semaphores we have studied are called counting (or integer) semaphores
- We have also binary semaphores
  - similar to counting semaphores except that “count” is Boolean valued
  - counting semaphores can be implemented by binary semaphores...
  - generally more difficult to use than counting semaphores (eg: they cannot be initialized to an integer k > 1)

Binary semaphores

```
Pb(S):
   if (S.value = 1) {
     S.value := 0;
   } else {
    block this process
    place this process in S.queue
  }

Vb(S):
  if (S.queue is empty) {
    S.value := 1;
  } else {
    remove a process P from S.queue
    place this process P on ready list
  }
```

- Semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes
- But P(S) and V(S) are scattered among several processes. Hence, difficult to understand their effects
- Usage must be correct in all the processes
- One bad (or malicious) process can fail the entire collection of processes

CR, CCR

Critical Regions

- Declare a shared variable v of type T as
  - **var v: shared T;**
    - The variable v can *only* be accessed inside a `region` statement.
  - **region v do S;**
    - While S is being executed, no other process can access v
- Using a Critical Region For Mutual Exclusion

```
var count: shared integer;

producer {
     .
     .
     region count do
         count = count + 1;
     .
     .
}
```

```
consumer {
     .
     .
     region count do
         count = count - 1;
     .
     .
}
```

Implementation of a Critical Region

- Assign each shared variable, x, a semaphore, xmutex, initialized to 1
- P & V semaphore around any *region* referencing the variable
- *The Code:*
  - ```
    var x: shared T;
              region x do S;
    ```
  - *Becomes:*
    ```
              var x: T;
              var xmutex: semaphore;

              P(xmutex);
              S;
              V(xmutex);
    ```
    ```

Possible problem

– Process 1 performs : **`"region x do region y do S1;"`**
– Process 2 performs : **`"region y do region x do S2;"`**

```
Process1:

   "P(x.mutex); P(y.mutex);"
   "S1;"
   "V(y.mutex); V(x.mutex);"
```

```
Process2:

   "P(y.mutex); P(x.mutex);"
   "S2;"
   "V(x.mutex); V(y.mutex);"
```

Solution: Conditional Critical Region (CCR)

– **region v when B do S;**

» B is a boolean expression which must be true before S is executed

Example: Bounded Buffer

```
var buffer: shared record
               pool: array[0..n-1] of item;
               count, in, out: integer;
            end;
```

Producer:

```
region buffer when count < n
     do begin
         pool[in] := nextp;
         in := in + 1 mod n;
         count := count + 1;
        end;
```

Consumer:

```
region buffer when count > 0
     do begin
         nextc := pool[out];
         out := out + 1 mod n;
         count := count - 1;
        end;
```

Implementation of the CCR

```
region v when B do S
var     e: semaphore; {entry semaphore}
        b: array[1..m] of sema; {delay semaphore}
               { for every condition in program }
        d: array[1..m] of int; {delay counters}
               { for every shared variable }

CCR:    P(e);
        if not Bi then
               d[i]++;
               V(e);
               P(b[i]);
        end if
        S;
             if ((B1) and (d[1] > 0)) then {d[1]--; V(b[1]); }
          elseif ((B2) and (d[2] > 0)) then {d[2]--; V(b[2]); }
                        ...
          else V(e);
```

– Problems
  - large number of semaphores
  - re-evaluation of all conditions
  - every process must evaluate conditions of all other processes (locals!)

Monitors

Monitors

- Are high-level language constructs that provide equivalent functionality to that of semaphores but are easier to control
- Found in many concurrent programming languages
  - Concurrent Pascal, Modula-3, C++, Java...
- Can be implemented by semaphores...

Monitor

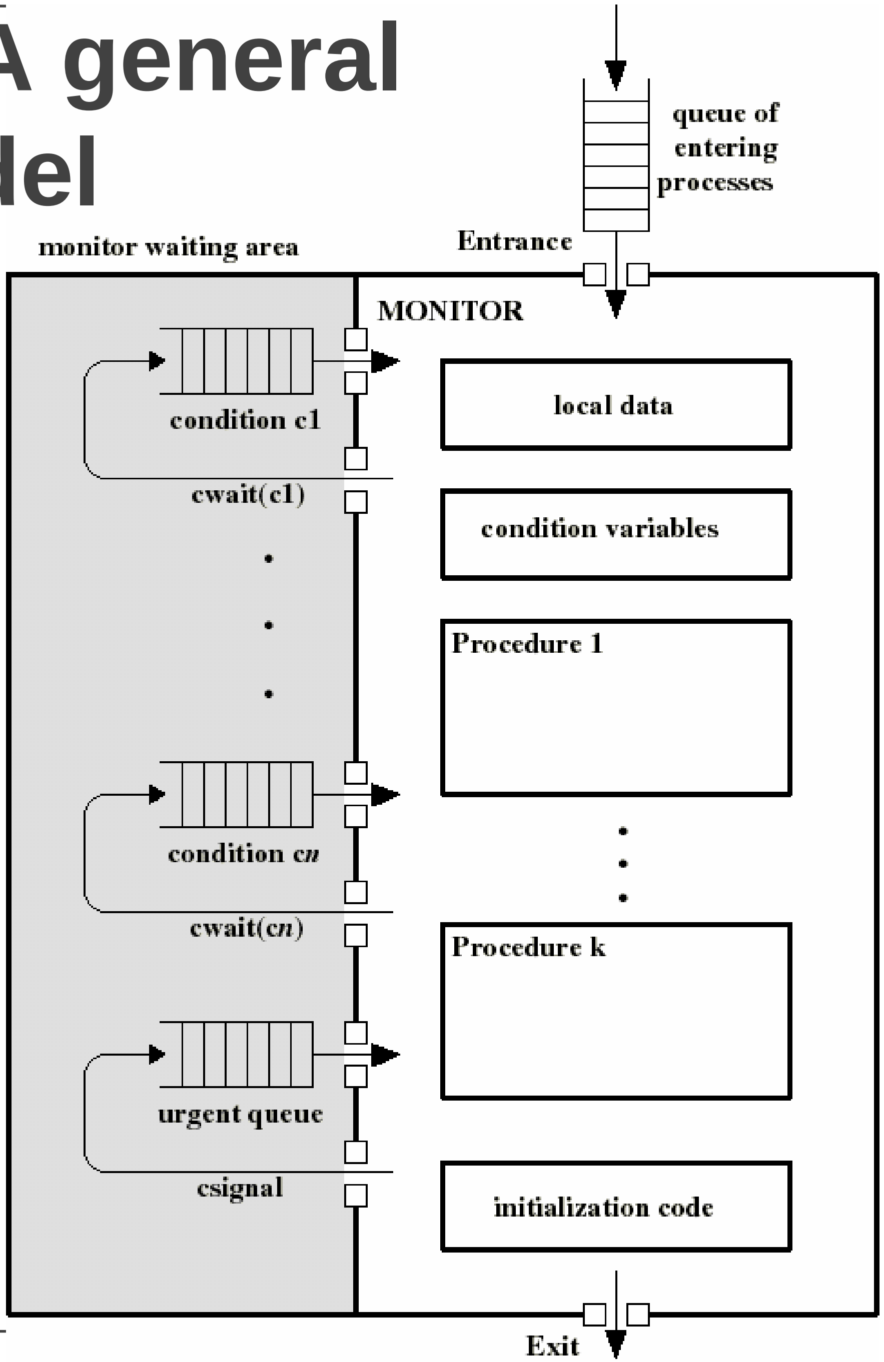
- Is a software module containing:
  - One or more procedures
  - An initialization sequence
  - Local data variables
- Characteristics:
  - Local variables accessible only by monitor’s procedures
  - A process enters the monitor by invoking one of it’s procedures
  - Only one process can be in the monitor at any one time

Monitor

- The monitor ensures mutual exclusion: no need to program this constraint explicitly
- Hence, shared data are protected by placing them in the monitor
  - The monitor locks the shared data on process entry
- Process synchronization is done by the programmer by using condition variables that represent conditions a process may need to wait for before executing in the monitor

Monitor – A general model

- Awaiting processes are either in the entrance queue or in a condition queue
- A process puts itself into condition queue cn by issuing wait(cn)
- signal(cn) brings into the monitor 1 process in condition cn queue
- Hence signal(cn) blocks the calling process and puts it in the urgent queue (unless signal is the last operation of the monitor procedure)



Condition variables

- Are local to the monitor (accessible only within the monitor)
- Can be access and changed only by two functions:
  - wait(a): blocks execution of the calling process on condition (variable) a
    - the process can resume execution only if another process executes csignal(a)
  - signal(a): resume execution of some process blocked on condition (variable) a.
    - If several such process exists: choose any one
    - If no such process exists: do nothing
  - Optional:
    - bool empty (c)
      - Returns true if delay queue is empty
    - signal_all (c)
      - while not empty (c) do signal (c);

Implementing monitor using semaphore

```
shared var
        e: semaphore = 1;
        c: semaphore = 0;                    // for each condition
        nc: integer = 0;                     // for each condition

monitor entry
        P(e)
wait(cond)
        nc = nc + 1;
        V(e); P(c); P(e);
signal(cond)
        if (nc > 0) then nc = nc - 1; V(c);
        end if
monitor exit
        V(e)
```

Implementing semaphore using monitor

```
monitor Semaphore
        var     s: integer = 1;
                pos: condition;
        procedure P();
        begin
                while (s=0) do wait (pos);
                s = s-1;
        end;
        procedure V();
        begin
                s = s+1;
                signal(pos);
        end;
end;
```

Monitor for the bounded P/C problem

- Monitor needs to hold the buffer:
  - buffer: array[0..k-1] of items;
- Needs two condition variables:
  - notfull: signal(notfull) indicates that the buffer is not full
  - notempty: signal(notempty) indicates that the buffer is not empty
- Needs buffer pointers and counts:
  - nextin: points to next item to be appended
  - nextout: points to next item to be taken
  - count: holds the number of items in buffer

Monitor for the bounded P/C problem - the general model

```
Monitor boundedbuffer:
  buffer: array[0..k-1] of items;
  nextin:=0, nextout:=0,
          count:=0: integer;
  notfull, notempty: condition;

  Procedure Append(v);
    begin
    if (count=k) wait(notfull);
    buffer[nextin]:= v;
    nextin:= nextin+1 mod k;
    count++;
    signal(notempty);
    end;

Procedure Take(v);
    begin
    if (count=0) wait(notempty);
    v:= buffer[nextout];
    nextout:= nextout+1 mod k;
    count--;
    signal(notfull);
    end;
```

Deadlock Free Dining Philosophers

```
Monitor dining-philosophers

   var state: array[0..4] of (thinking,hungry,eating);
   var philosopher: array[0..4] of condition;

   procedure entry pickup( i: 0..4 );
       begin
          state[i] := hungry;
          test(i);
          if state[i] != eating
             then philosopher[i].wait;
       end;

   procedure entry putdown( i: 0..4 );
       begin
          state[i] := thinking;
          test( i-1 mod 5 );
          test( i+1 mod 5 );
       end;
```

Deadlock Free Dining Philosophers

```
procedure test( k: 0..4 );
    begin
       if state[k-1 mod 5] != eating
          and state[k] = hungry
          and state[k+1 mod 5] != eating
          then begin
                 state[k] := eating;
                 philosopher[k].signal;
               end;
    end;

begin
   for i := 0 to 4 do state[i] := thinking;
end.
```

The End