Mutual Exclusion

The Producer-Consumer Problem

int counter = 0;
Item buffer[n];

Producer

```
while(1){
    …
    produce an item in nextp
    …
    while(counter == n)
       do no-op
    buffer[in] = nextp
    in = (in + 1) mod n
    counter = counter + 1

}
```

Consumer

```
while(1){
    …
    while(counter == 0)
       do no-op
    nextc = buffer[out]
    out = (out + 1) mod n
    counter = counter -1
    …
    consume the item in nextc
    …
}
```

Bounded Buffer Solution

```
Shared semaphore: empty = n, full = 0;
Item buffer[n];
```

```
while(1){

  produce an item in nextp

  wait(empty);

  buffer[in] = nextp
  in = (in + 1) mod n

  signal(full);

}
```

Producer

```
while(1){

  wait(full);

  nextc = buffer[out]
  out = (out + 1) mod n


  signal(empty);

  consume the item in nextc

}
```

Consumer

Any problem?

Bounded Buffer Solution

```
Shared semaphore: empty = n, full = 0, mutex = 1;
Item buffer[n]; int in = out = 0;
```

Producer

```
while(1){
  produce an item in nextp

  wait(empty);
  wait(mutex);

  buffer[in] = nextp
  in = (in + 1) mod n

  signal(mutex);
  signal(full);
}
```

Consumer

```
while(1){
  wait(full);
  wait(mutex);

  nextc = buffer[out]
  out = (out + 1) mod n

  signal(mutex);
  signal(empty);

  consume the item in nextc
}
```

Mutex + Synchronization: multiple producers and comusers

Message Passing

- A general method used for interprocess communication (IPC)
  - for processes inside the same computer
  - for processes in a distributed system
- Another means to provide process synchronization and mutual exclusion
- We have at least two primitives:
  - send(destination, message)
  - receive(source, message)
- May or may not be blocking

Synchronization

- For the sender: it is more natural not to be blocked
  - can send several messages to multiple destinations
  - sender usually expects acknowledgment of message receipt (in case receiver fails)

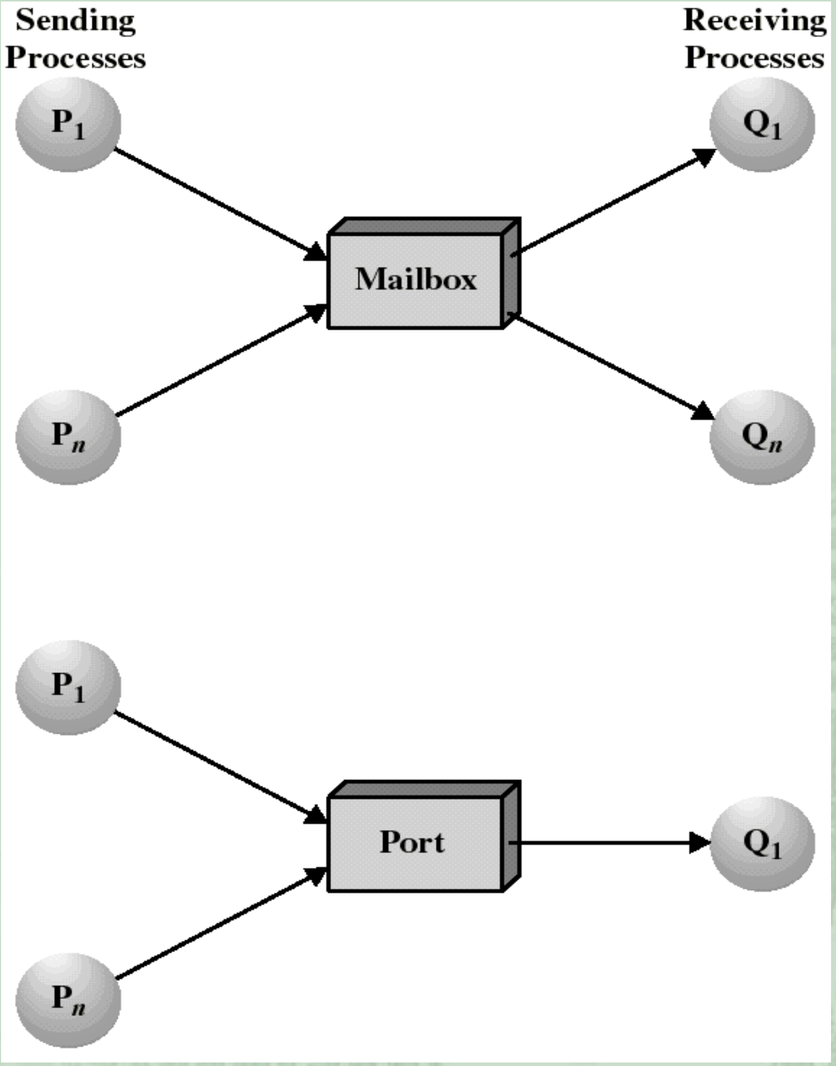
- For the receiver: it is more natural to be blocked after issuing *ReceiveMessage*()
  - the receiver usually needs the info before proceeding
  - but could be blocked indefinitely if there is no sender

Addressing in message passing

- Direct addressing:
  - when a specific process identifier is used for source/destination
  - but it might be impossible to specify the source ahead of time (ex: a print server)
- Indirect addressing (more convenient):
  - messages are sent to a shared mailbox which consists of a queue of messages
  - senders place messages in the mailbox, receivers pick them up

Mailboxes and Ports

- A mailbox can be private
  - one sender/receiver pair
- A mailbox can be shared among several senders and receivers
  - OS may then allow the use of message types (for selection)
- Port: a mailbox associated with one receiver and multiple senders
  - used for client/server application: the receiver is the server

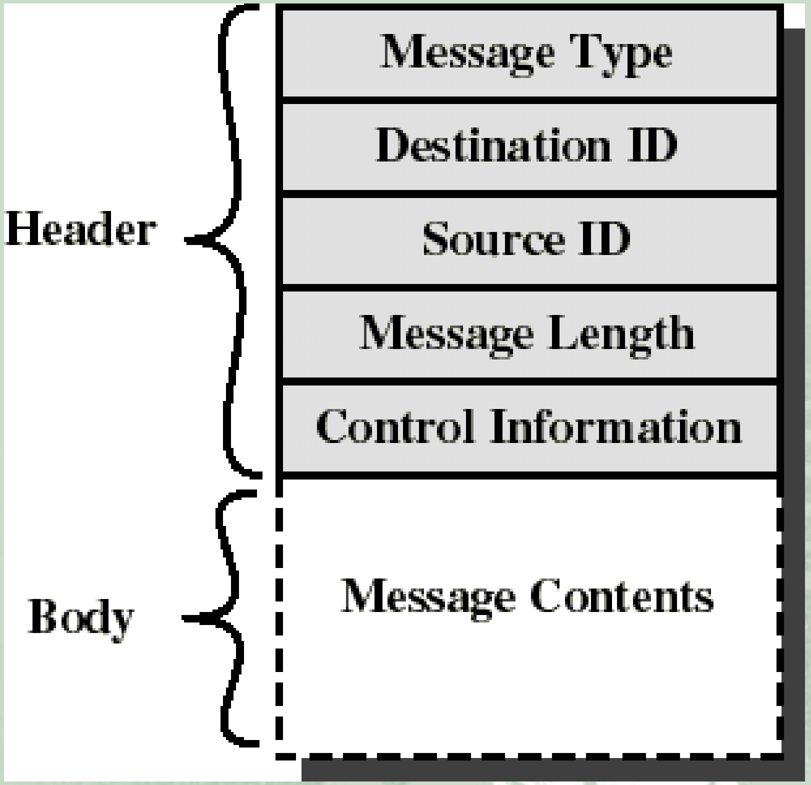


Ownership of ports and mailboxes

- A port is usually owned and created by the receiving process
- The port is destroyed when the receiver terminates
- The OS creates a mailbox on behalf of a process (which becomes the owner)
- The mailbox is destroyed at the owner's request or when the owner terminates

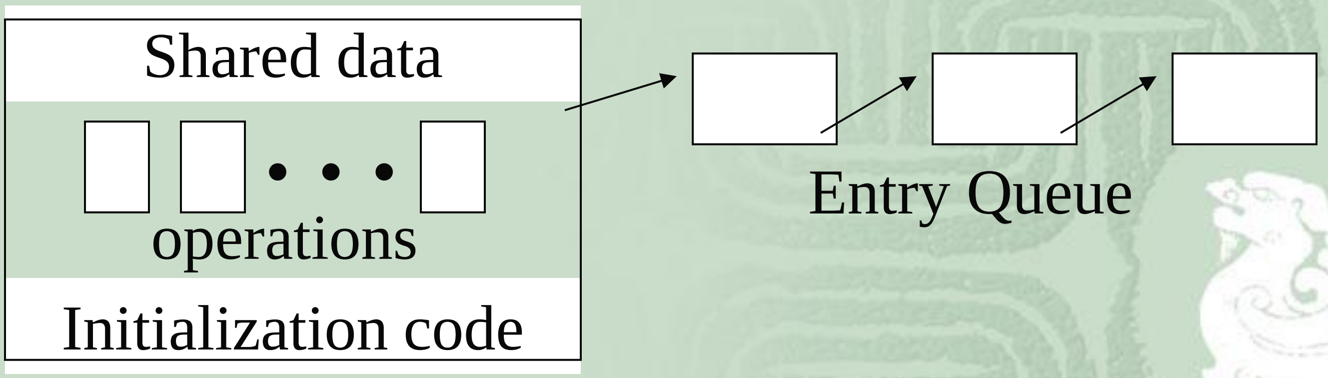
Message format

- Consists of header and body of message
- Control information:
  - what to do if run out of buffer space
  - sequence numbers
  - priority...
- **Queuing discipline: usually FIFO but can also include priorities**



Monitor

- 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 its procedures
  - only one process can be in the monitor at any one time

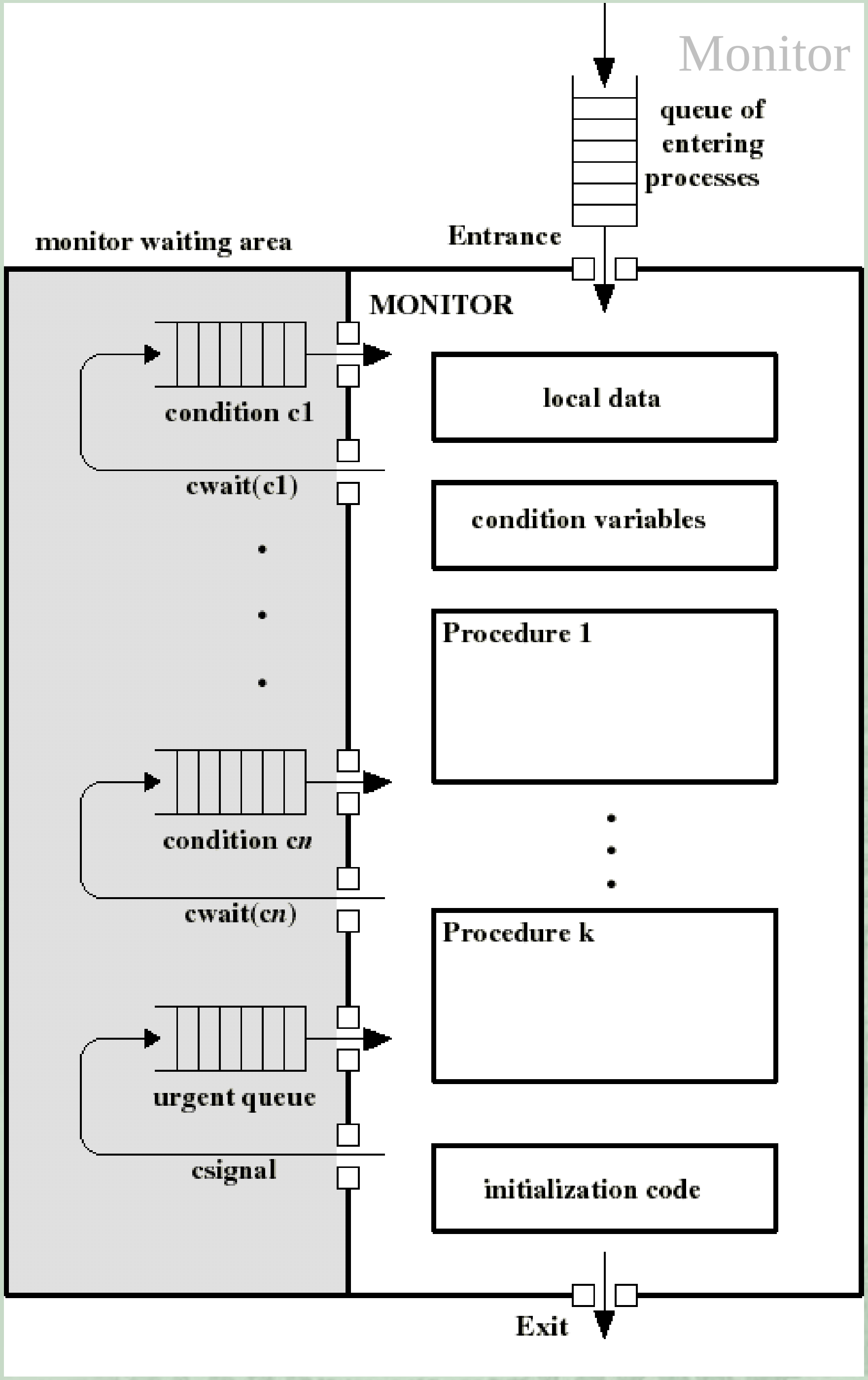


Monitor Mutual Exclusion

- **The monitor ensures mutual exclusion - no need to program this constraint explicitly.**
- **The monitor locks (protects) shared data on process entry.**
- **Process synchronization is done by the programmer by using condition variables.**
  - conditions needing to be satisfied before entering monitor
  - local to the monitor - accessible only within the monitor
    - **cwait(*a*):** blocks execution of the calling process on condition variable *a*. The process can resume execution only if another process executes csignal(*a*)
    - **csignal(*a*):** resume execution of some process blocked on condition variable *a*.
      - If several such processes exist: choose any one
      - If no such process exists: do nothing

Monitor

- Waiting processes are
  - in the entrance queue or
  - in a condition queue
- A process puts itself into condition queue c_i by issuing *cwait*(c_i)
- *csignal*(c_i) brings into the monitor one process in condition c_i queue
- *csignal*(c_i) blocks the calling process and puts it in the urgent queue
  - unless *csignal* is the last operation of the monitor procedure



Monitor implementation

- Semaphore to enter monitor
- Semaphore for each condition variable
- Must allow multiple processes in the monitor
  - only one active
  - others are in condition variable queues

Monitor for the P/C problem

- Monitor holds the buffer:
  - buffer: array[0..k-1] of items;
- Two condition variables:
  - notfull: csignal(notfull) indicates that the buffer is not full
  - notemty: csignal(notempty) indicates that the buffer is not empty
- 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 P/C problem

```
Monitor boundedbuffer:
  item buffer[0..k-1];
  int nextin=0, nextout=0, count=0;
  condition notfull, notempty;

  Append(item v){
    if (count==k) cwait(notfull);
    buffer[nextin] = v;
    nextin = (nextin + 1) mod k;
    count++;
    csignal(notempty);
  }
  Take(){
    if (count==0) cwait(notempty);
    v = buffer[nextout];
    nextout = (nextout + 1) mod k;
    count--;
    csignal(notfull);
    return v;
  }
```

Monitor for the P/C problem

```
procedure producer() {
        while (true) {
          item v = produceItem();
          boundedbuffer.Append(v);
        }
}
procedure consumer() {
        while (true) {
          item v = boundedbuffer.Take();
          consumeItem(v);
         }
}
```

Classical Synchronization Problems

Readers and Writers Problem

- Data object is shared (file, memory, registers)
  - many processes that only read data (readers)
  - many processes that only write data (writers)
- Conditions needing to be satisfied:
  - many can read at the same time (concurrency)
  - only one writer at a time and no one allowed to read while someone is writing (mutual exclusion)
- Solutions result in reader or writer priority

Readers/Writers (priority?)

```
Semaphore rmutex=1, wmutex = 1;
integer readcount = 0;
```

WRITERS:
```
while(true)
{   wait(wmutex);
    <write to the data object>
    signal(wmutex);
};
```

Only one writer at a time

The first reader makes sure no one can write

READERS:
```
while(true)
{   wait(rmutex);
    readcount++;
    if (readcount == 1) wait(wmutex);
    signal(rmutex);
    <read the data>
    wait(rmutex);
    readcount--;
    if (readcount == 0) signal(wmutex);
    signal(rmutex);
};
```

More than one reader at a time

Last one out allows writing again

Writers/Readers (priority?)

```
Semaphore outerQ = rsem = rmutex = wmutex = wsem = 1;
```

```
while(true)
{ wait(outerQ);
    wait(rsem);
      wait(rmutex);
        readcnt++
        if (readcnt == 1)
          wait(wsem);
      signal(rmutex);
    signal(rsem);
  signal(outerQ);

  READ

  wait(rmutex);
    readcnt--;
    if(readcnt == 0)
      signal(wsem);
  signal(rmutex);
};
```

Additional readers queue here allowing writers to jump ahead of the readers

Disable writers

Last reader out allows writers

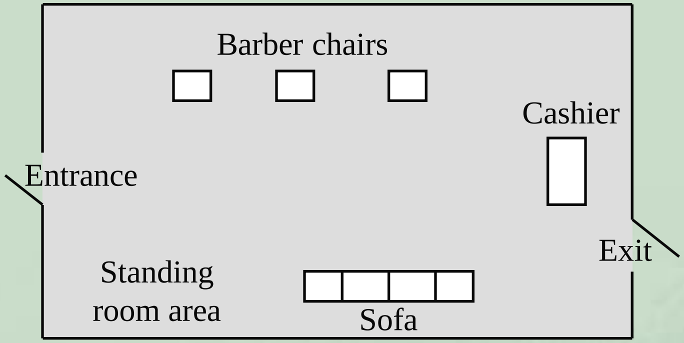
```
while(true)
{ wait(wmutex);
    writecnt++;
    if (writecnt == 1)
      wait(rsem);
  signal(wmutex);
  wait(wsem);

  WRITE

  signal(wsem);
  wait(wmutex);
    writecnt--;
    if (writecnt == 0)
      signal(rsem);
  signal(wmutex);
};
```

Once a writer wants to write – no new readers allowed

Wait here until all readers done

Last writer out allows readers

Barbershop Problem



- 3 barbers, each with a barber chair
  - Haircuts vary in time
- Sofa can hold 4 customers
- Maximum of 20 customers 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 on sofa
- After haircut, customer pays cashier at cash register
  - Algorithm has a separate cashier, but often barbers also take payment

Fair Barbershop

```
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 );
   enqueue( 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 );
      dequeue( b_cust );
      signal( mutex2 );
      wait( coord );
      /* cut hair */
      signal( coord );
      signal( finished[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;
```

```
program    barbershop;
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;
```