Concurrency: Mutual Exclusion and Synchronization

Chapter 5

Problems with concurrent execution

- Concurrent processes (or threads) often need to share data (maintained either in shared memory or files) and resources
- If there is no controlled access to shared data, some processes will obtain an inconsistent view of this data
- The action performed by concurrent processes will then depend on the order in which their execution is interleaved

An example

- Process P1 and P2 are running this same procedure and have access to the same variable “a”
- Processes can be interrupted anywhere
- If P1 is first interrupted after user input and P2 executes entirely
- Then the character echoed by P1 will be the one read by P2 !!

```
static char a;

void echo()
{
   cin >> a;
   cout << a;
}
```

Race Conditions

- **Situations like this where processes access the same data concurrently and the outcome of execution depends on the particular order in which the access takes place is called a race condition**
- **How must the processes coordinate (or synchronise) in order to guard against race conditions?**

The critical section problem

- When a process executes code that manipulates shared data (or resource), we say that the process is in it's critical section (CS) (for that shared data)
- The execution of critical sections must be mutually exclusive: at any time, only one process is allowed to execute in its critical section (even with multiple CPUs)
- Then each process must request the permission to enter it's critical section (CS)

The critical section problem

- The section of code implementing this request is called the **entry section**
- The critical section (CS) might be followed by an **exit section**
- The remaining code is the **remainder section**
- The critical section problem is to design a protocol that the processes can use so that their action will not depend on the order in which their execution is interleaved (possibly on many processors)

Framework for analysis of solutions

- Each process executes at nonzero speed but no assumption on the relative speed of n processes
- General structure of a process:

```
repeat
    entry section
     critical section
    exit section
     remainder section
forever
```

- many CPU may be present but memory hardware prevents simultaneous access to the same memory location
- No assumption about order of interleaved execution
- For solutions: we need to specify entry and exit sections

Requirements for a valid solution to the critical section problem

- **Mutual Exclusion**
  - At any time, at most one process can be in its critical section (CS)
- **Progress**
  - Only processes that are not executing in their RS can participate in the decision of who will enter next in the CS.
  - This selection cannot be postponed indefinitely

Requirements for a valid solution to the critical section problem (cont.)

- **Bounded Waiting**
  - After a process has made a request to enter it's CS, there is a bound on the number of times that the other processes are allowed to enter their CS
    - otherwise the process will suffer from starvation
  - Of course also no deadlock

Types of solutions

- **Software solutions**
  - algorithms who's correctness does not rely on any other assumptions (see framework)
- **Hardware solutions**
  - rely on some special machine instructions
- **Operation System solutions**
  - provide some functions and data structures to the programmer

Software solutions

- **We consider first the case of 2 processes**
  - Algorithm 1 and 2 are incorrect
  - Algorithm 3 is correct (Peterson's algorithm)
- **Then we generalize to n processes**
  - the bakery algorithm
- **Notation**
  - We start with 2 processes: P0 and P1
  - When presenting process Pi, Pj always denotes the other process (i != j)

Algorithm 1

- The shared variable turn is initialized (to 0 or 1) before executing any Pi
- Pi's critical section is executed iff turn = i
- Pi is busy waiting if Pj is in CS: mutual exclusion is satisfied
- Progress requirement is not satisfied since it requires strict alternation of CSs
- If a process requires its CS more often then the other, it cannot get it.

```
Process Pi:
repeat
  while(turn!=i){};
     CS
  turn:=j;
     RS
forever
```

```
Process P0:
repeat

while(turn!=0){};
     CS
  turn:=1;
     RS
forever
```

```
Process P1:
repeat

while(turn!=1){};
     CS
  turn:=0;
     RS
forever
```

Algorithm 1 global view

- **Ex: P0 has a large RS and P1 has a small RS. If turn=0, P0 enter its CS and then its long RS (turn=1). P1 enter its CS and then its RS (turn=0) and tries again to enter its CS: request refused! He has to wait that P0 leaves its RS.**

Algorithm 2

- Keep 1 Bool variable for each process: flag[0] and flag[1]
- Pi signals that it is ready to enter it's CS by: flag[i]:=true
- Mutual Exclusion is satisfied but not the progress requirement
- If we have the sequence:
  - T0: flag[0]:=true
  - T1: flag[1]:=true
- Both process will wait forever to enter their CS: we have a deadlock

```
Process Pi:
repeat
  flag[i]:=true;
  while(flag[j]){};
      CS
  flag[i]:=false;
      RS
forever
```

Algorithm 3 (Peterson's algorithm)

- Initialization: flag[0]:=flag[1]:=false turn:= 0 or 1
- Willingness to enter CS specified by flag[i]:=true
- If both processes attempt to enter their CS simultaneously, only one `turn` value will last
- Exit section: specifies that Pi is unwilling to enter CS

```
Process Pi:
repeat
  flag[i]:=true;
    // I want in
  turn:=j;
   // but I let the other in
  while
   (flag[j]&turn=j){};
     CS
  flag[i]:=false;
   // I no longer want in
     RS
forever
```

```
Process P0:
repeat
  flag[0]:=true;
    // 0 wants in
  turn:= 1;
   // 0 gives a chance to 1
  while
   (flag[1]&turn=1){};
     CS
  flag[0]:=false;
   // 0 no longer wants in
     RS
forever
```

```
Process P1:
repeat
  flag[1]:=true;
    // 1 wants in
  turn:=0;
   // 1 gives a chance to 0
  while
   (flag[0]&turn=0){};
     CS
  flag[1]:=false;
   // 1 no longer wants in
     RS
forever
```

Peterson’s algorithm global view

Algorithm 3: proof of correctness

- **Mutual exclusion is preserved since:**
  - P0 and P1 are both in CS only if flag[0] = flag[1] = true and only if turn = i for each Pi (impossible)
- **We now prove that the progress and bounded waiting requirements are satisfied:**
  - Pi cannot enter CS only if stuck in while() with condition flag[j] = true and turn = j.
  - If Pj is not ready to enter CS then flag[j] = false and Pi can then enter its CS

Algorithm 3: proof of correctness (cont.)

- If Pj has set flag[j]=true and is in its while(), then either turn=i or turn=j
- If turn=i, then Pi enters CS. If turn=j then Pj enters CS but will then reset flag[j]=false on exit: allowing Pi to enter CS
- but if Pj has time to reset flag[j]=true, it must also set turn=i
- since Pi does not change value of turn while stuck in while(), Pi will enter CS after at most one CS entry by Pj (bounded waiting)

What about process failures?

- **If all 3 criteria (ME, progress, bounded waiting) are satisfied, then a valid solution will provide robustness against failure of a process in its remainder section (RS)**
  - since failure in RS is just like having an infinitely long RS
- **However, no valid solution can provide robustness against a process failing in its critical section (CS)**
  - A process Pi that fails in its CS does not signal that fact to other processes: for them Pi is still in its CS

n-process solution: bakery algorithm

- **Before entering their CS, each Pi receives a number. Holder of smallest number enter CS (like in bakeries, ice-cream stores...)**
- **When Pi and Pj receives same number:**
  - if i<j then Pi is served first, else Pj is served first
- **Pi resets its number to 0 in the exit section**
- **Notation:**
  - (a,b) < (c,d) if a < c or if a = c and b < d
  - max(a0,...ak) is a number b such that
    - b >= ai for i=0,..k

The bakery algorithm (cont.)

- **Shared data:**
  - choosing: array[0..n-1] of boolean;
    - initialized to false
  - number: array[0..n-1] of integer;
    - initialized to 0
- **Correctness relies on the following fact:**
  - If Pi is in CS and Pk has already chosen its number[k]!= 0, then (number[i],i) < (number[k],k)
  - but the proof is somewhat tricky...

The bakery algorithm (cont.)

```
Process Pi:
repeat
  choosing[i]:=true;
  number[i]:=max(number[0]..number[n-1])+1;
  choosing[i]:=false;
  for j:=0 to n-1 do {
    while (choosing[j]) {};
    while (number[j]!=0
          and (number[j],j)<(number[i],i)){};
  }
  CS
  number[i]:=0;
  RS
forever
```

Drawbacks of software solutions

- Processes that are requesting to enter in their critical section are busy waiting (consuming processor time needlessly)
- If Critical Sections are long, it would be more efficient to block processes that are waiting...

Hardware solutions: interrupt disabling

- **On a uniprocessor: mutual exclusion is preserved but efficiency of execution is degraded: while in CS, we cannot interleave execution with other processes that are in RS**
- **On a multiprocessor: mutual exclusion is not preserved**
  - CS is now atomic but not mutually exclusive
  - Generally not an acceptable solution

```
Process Pi:
repeat
   disable interrupts
    critical section
   enable interrupts
    remainder section
forever
```

Hardware solutions: special machine instructions

- Normally, access to a memory location excludes other access to that same location
- Extension: designers have proposed machines instructions that perform 2 actions atomically (indivisible) on the same memory location (ex: reading and writing)
- The execution of such an instruction is also mutually exclusive (even with multiple CPUs)
- They can be used to provide mutual exclusion but need to be complemented by other mechanisms to satisfy the other 2 requirements of the CS problem (and avoid starvation and deadlock)

The test-and-set instruction

- A C++ description of test-and-set:

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

- An algorithm that uses testset for Mutual Exclusion:
- Shared variable b is initialized to 0
- Only the first Pi who sets b enter CS

```
Process Pi:
repeat
  repeat{}
  until testset(b);
     CS
  b:=0;
     RS
forever
```

The test-and-set instruction (cont.)

- Mutual exclusion is preserved: if Pi enter CS, the other Pj are busy waiting
- Problem: still using busy waiting
- When Pi exit CS, the selection of the Pj who will enter CS is arbitrary: no bounded waiting. Hence starvation is possible
- Processors (ex: Pentium) often provide an atomic xchg(a,b) instruction that swaps the content of a and b.
- But xchg(a,b) suffers from the same drawbacks as test-and-set

Using xchg for mutual exclusion

- Shared variable b is initialized to 0
- Each Pi has a local variable k
- The only Pi that can enter CS is the one who finds b=0
- This Pi excludes all the other Pj by setting b to 1

```
Process Pi:
repeat
   k:=1
   repeat xchg(k,b)
   until k=0;
      CS
   b:=0;
      RS
forever
```

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:
  - wait(S)
  - signal(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 (acc. to a fair policy like FIFO) one process from the queue and puts it in the list of ready processes

Semaphore's operations

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

signal(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 wait(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 wait(S) and signal(S) (on the same S) at the same time (even with multiple CPUs)
- Hence the blocks of code defining wait(S) and signal(S) are, in fact, critical sections

Semaphores: observations

- **The critical sections defined by wait(S) and signal(S) are very short: typically 10 instructions**
- **Solutions:**
  - uniprocessor: disable interrupts during these operations (ie: for a very short period). This does not work on a multiprocessor machine.
  - multiprocessor: use previous software or hardware schemes. The amount of busy waiting should be small.

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
  wait(S);
   CS
  signal(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;
  - signal(synch);

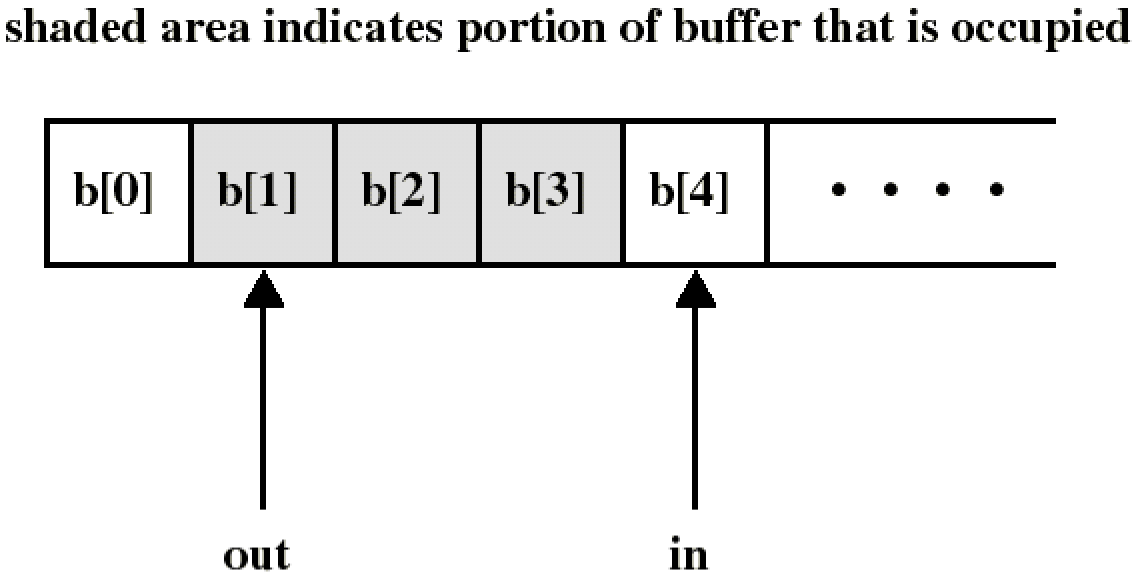
- **And having in P2:**
  - wait(synch);
  - S2;

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

P/C: unbounded buffer

- The producer is free to add an item into the buffer at any time: it performs wait(S) before appending and signal(S) afterwards to prevent customer access
- It also performs signal(N) after each append to increment N
- The consumer must first do wait(N) to see if there is an item to consume and use wait(S)/signal(S) to access the buffer

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;
   wait(S);
   append(v);
   signal(S);
   signal(N);
forever
```

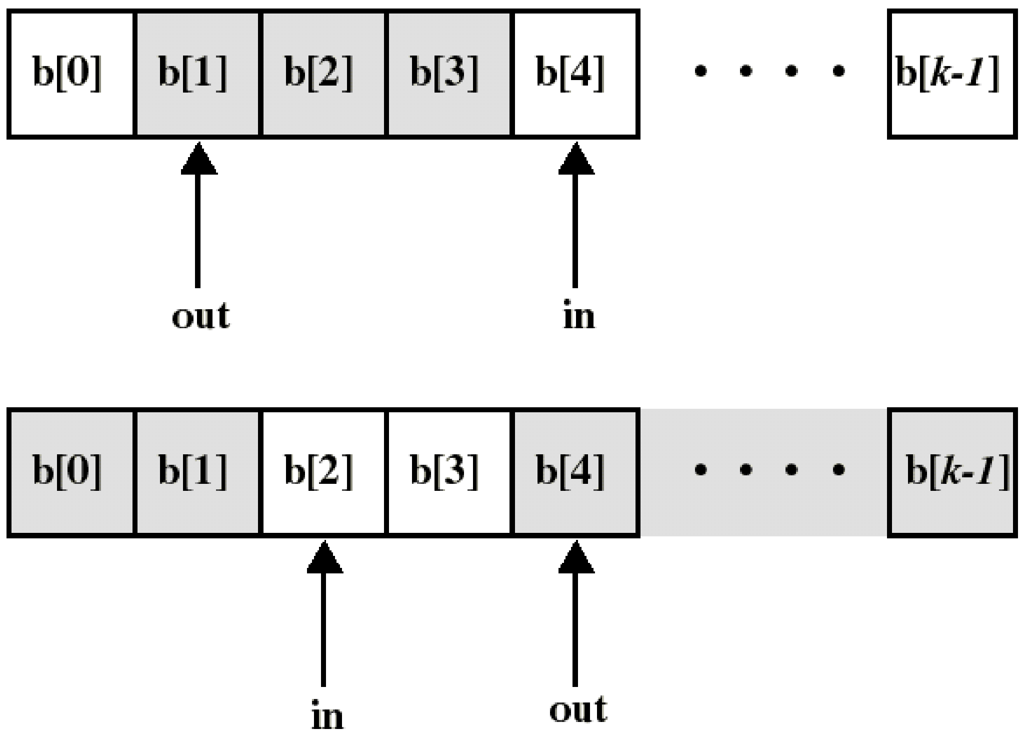
```
Consumer:
repeat
   wait(N);
   wait(S);
   w:=take();
   signal(S);
   consume(w);
forever
```

- critical sections

P/C: unbounded buffer

- **Remarks:**
  - Putting signal(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 wait(N) before wait(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

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;
```

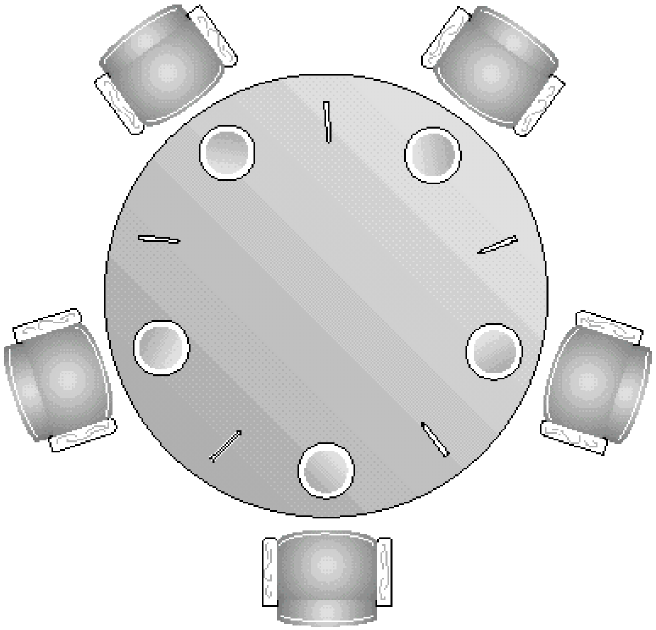
```
Producer:
repeat
   produce v;
   wait(E);
   wait(S);
   append(v);
   signal(S);
   signal(N);
forever
```

```
Consumer:
repeat
   wait(N);
   wait(S);
   w:=take();
   signal(S);
   signal(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 synchron. 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;
  wait(fork[i]);
  wait(fork[i+1 mod 5]);
  eat;
  signal(fork[i+1 mod 5]);
  signal(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;
 wait(T);
 wait(fork[i]);
 wait(fork[i+1 mod 5]);
 eat;
 signal(fork[i+1 mod 5]);
 signal(fork[i]);
 signal(T);
forever
```

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

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

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

Problems with semaphores

- semaphores provide a powerful tool for enforcing mutual exclusion and coordinate processes
- But wait(S) and signal(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

Readers/Writers Problem

- **A data area is shared among many processes**
  - Some processes only read the data area, (readers) and some only write to the data area (writers)
- **Conditions that must be satisfied:**
  - 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

```
/* program readersandwriters */
int readcount;
semaphore x = 1, wsem = 1;
void reader()
{
     while (true) {
       semWait (x);
       readcount++;
       if (readcount == 1) semWait (wsem);
       semSignal (x);
       READUNIT();
       semWait (x);
       readcount--;
       if (readcount == 0) semSignal (wsem);
       semSignal (x);
     }
 }
void writer()
{
     while (true) {
       semWait (wsem);
       WRITEUNIT();
       semSignal (wsem);
     }
}

void main()
{
     readcount = 0;
     parbegin (reader, writer);
}
```

Figure 5.25 A Solution to the Readers/Writers Problem Using Semaphores: Readers Have Priority

The semaphore wsem is used to enforce mutual exclusion. As long as one writer is accessing the shared data area, no other writers and no readers may access it. The global variable readcount is used to keep track of the number of readers, and the semaphore x is used to assure that readcount is updated properly.

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, uC++, 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

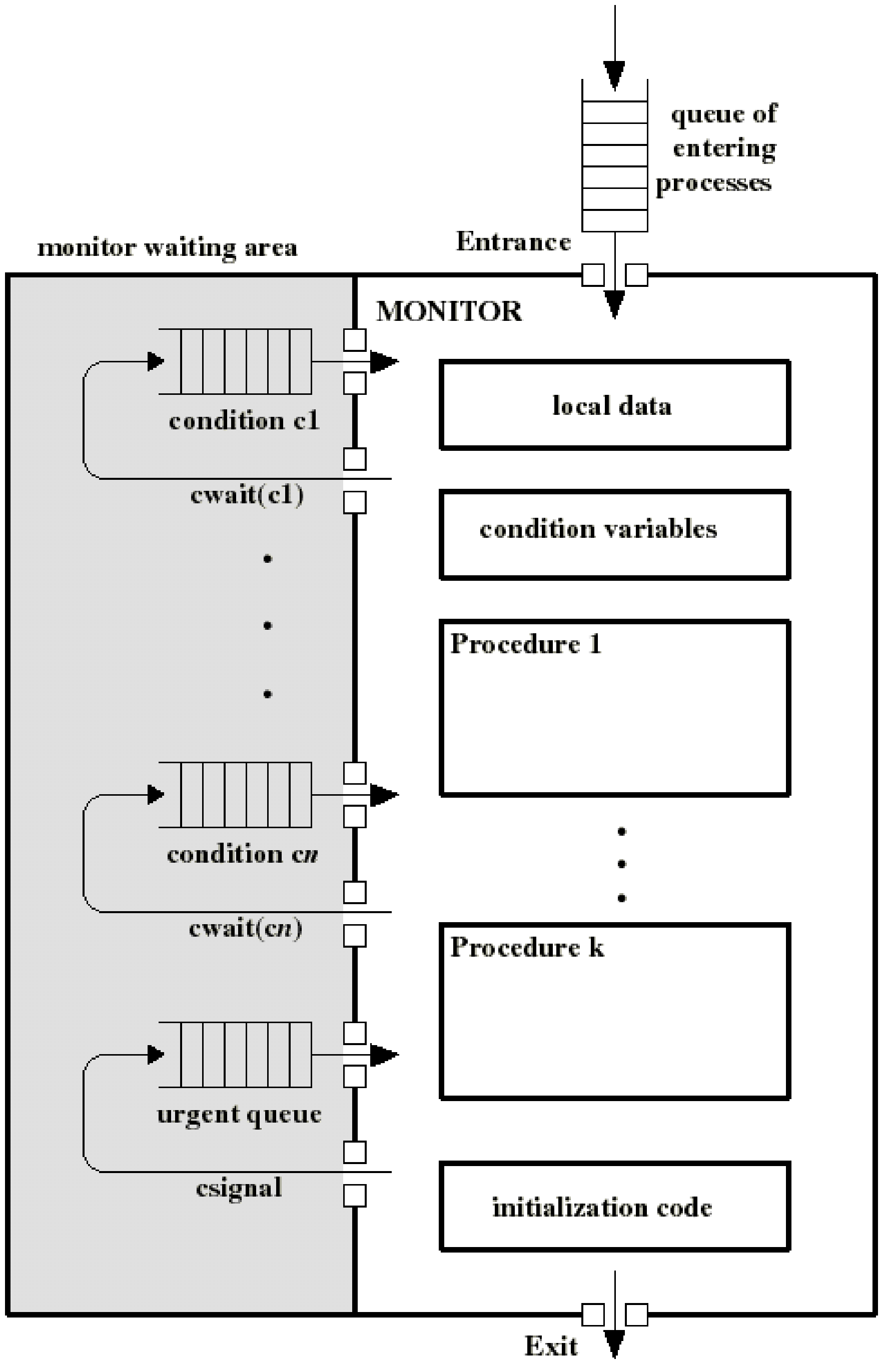
- **The monitor ensures mutual exclusion: no need to program this constraint explicitly**
- **Hence, shared data are protected by placing them in the monitor**
  - The monitor locks the shared data on process entry
- **Process synchronization is done by the programmer by using condition variables that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

Condition variables

- **are local to the monitor (accessible only within the monitor)**
- **can be access and changed only by two functions:**
  - 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 process exists: choose any one
    - If no such process exists: do nothing

Monitor

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



Producer/Consumer problem

- Two types of processes:
  - producers
  - consumers
- Synchronization is now confined within the monitor
- append(.) and take(.) are procedures within the monitor: are the only means by which P/C can access the buffer
- If these procedures are correct, synchronization will be correct for all participating processes

```
ProducerI:
repeat
  produce v;
  Append(v);
forever

ConsumerI:
repeat
  Take(v);
  consume v;
forever
```

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: csignal(notfull) indicates that the buffer is not full
  - notemty: csignal(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

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

  Append(v):
    if (count=k) cwait(notfull);
    buffer[nextin]:= v;
    nextin:= nextin+1 mod k;
    count++;
    csignal(notempty);

  Take(v):
    if (count=0) cwait(notempty);
    v:= buffer[nextout];
    nextout:= nextout+1 mod k;
    count--;
    csignal(notfull);
```

Message Passing

- **Is a general method used for interprocess communication (IPC)**
  - for processes inside the same computer
  - for processes in a distributed system
- **Yet another mean to provide process synchronization and mutual exclusion**
- **We have at least two primitives:**
  - send(destination, message)
  - received(source, message)
- **In both cases, the process may or may not be blocked**

Synchronization in message passing

- **For the sender: it is more natural not to be blocked after issuing send(.,.)**
  - can send several messages to multiple dest.
  - but sender usually expect acknowledgment of message receipt (in case receiver fails)
- **For the receiver: it is more natural to be blocked after issuing receive(.,.)**
  - the receiver usually needs the info before proceeding
  - but could be blocked indefinitely if sender process fails before send(.,.)

Synchronization in message passing

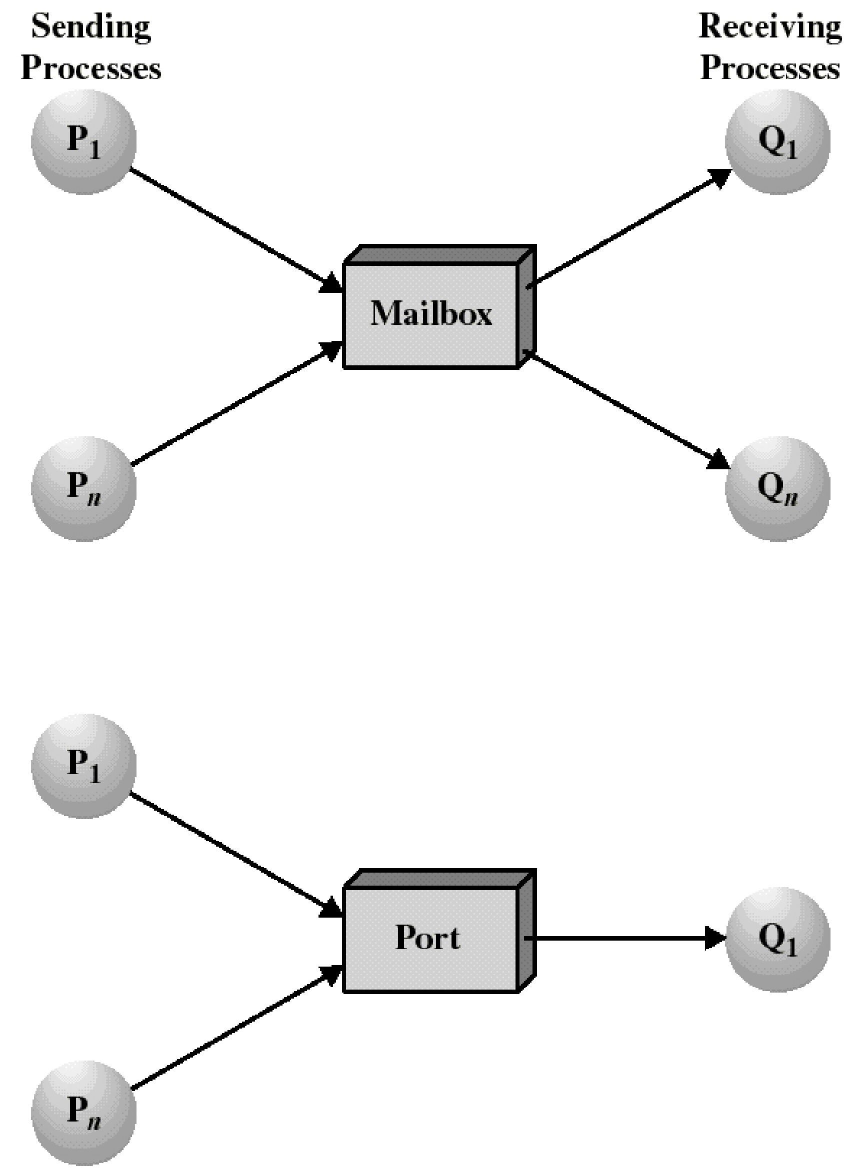
- **Hence other possibilities are sometimes offered**
- **Ex: blocking send, blocking receive:**
  - both are blocked until the message is received
  - occurs when the communication link is unbuffered (no message queue)
  - provides tight synchronization (*rendez-vous*)

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

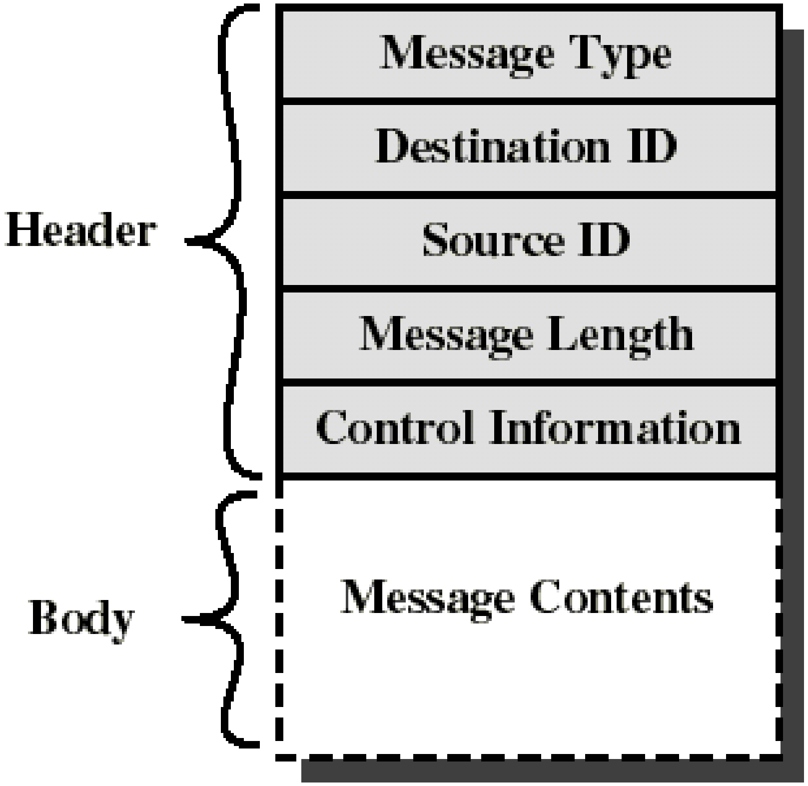


Ownership of ports and mailboxes

- A port is usually own 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**
- **In Unix: no ID, only message type**
- **control info:**
  - what to do if run out of buffer space
  - sequence numbers
  - priority...
- Queuing discipline: usually FIFO but can also include priorities



Enforcing mutual exclusion with message passing

- **create a mailbox *mutex* shared by n processes**
- **send() is non blocking**
- **receive() blocks when *mutex* is empty**
- **Initialization: send(*mutex*, “go”);**
- **The first Pi who executes receive() will enter CS. Others will be blocked until Pi resends msg.**

```
Process Pi:
var msg: message;
repeat
   receive(mutex,msg);
   CS
   send(mutex,msg);
   RS
forever
```

The bounded-buffer P/C problem with message passing

- We will now make use of messages
- The producer place items (inside messages) in the mailbox *mayconsume*
- *mayconsume* acts as our buffer: consumer can consume item when at least one message is present
- Mailbox *mayproduce* is filled initially with k null messages (k= buffer size)
- The size of *mayproduce* shrinks with each production and grows with each consumption
- can support multiple producers/consumers

The bounded-buffer P/C problem with message passing

```
Producer:
var pmsg: message;
repeat
  receive(mayproduce, pmsg);
  pmsg:= produce();
  send(mayconsume, pmsg);
forever

Consumer:
var cmsg: message;
repeat
  receive(mayconsume, cmsg);
  consume(cmsg);
  send(mayproduce, null);
forever
```

Unix SVR4 concurrency mechanisms

- **To communicate data across processes:**
  - Pipes
  - Messages
  - Shared memory
- **To trigger actions by other processes:**
  - Signals
  - Semaphores

Unix Pipes

- **A shared bounded FIFO queue written by one process and read by another**
  - based on the producer/consumer model
  - OS enforces Mutual Exclusion: only one process at a time can access the pipe
  - if there is not enough room to write, the producer is blocked, else he writes
  - consumer is blocked if attempting to read more bytes that are currently in the pipe
  - accessed by a file descriptor, like an ordinary file
  - processes sharing the pipe are unaware of each other's existence

Unix Messages

- **A process can create or access a message queue (like a mailbox) with the** *msgget* **system call.**
- *msgsnd* **and** *msgrcv* **system calls are used to send and receive messages to a queue**
- **There is a "type" field in message headers**
  - FIFO access within each message type
  - each type defines a communication channel
- **Process is blocked (put asleep) when:**
  - trying to receive from an empty queue
  - trying to send to a full queue

Shared memory in Unix

- A block of virtual memory shared by multiple processes
- The *shmget* system call creates a new region of shared memory or return an existing one
- A process attaches a shared memory region to its virtual address space with the *shmat* system call
- Mutual exclusion must be provided by processes using the shared memory
- Fastest form of IPC provided by Unix

Unix signals

- **Similar to hardware interrupts without priorities**
- **Each signal is represented by a numeric value. Ex:**
  - 02, SIGINT: to interrupt a process
  - 09, SIGKILL: to terminate a process
- **Each signal is maintained as a single bit in the process table entry of the receiving process: the bit is set when the corresponding signal arrives (no waiting queues)**
- **A signal is processed as soon as the process runs in user mode**
- **A default action (eg: termination) is performed unless a signal handler function is provided for that signal (by using the *signal* system call)**

Unix Semaphores

- **Are a generalization of the counting semaphores (more operations are permitted).**
- **A semaphore includes:**
  - the current value S of the semaphore
  - number of processes waiting for S to increase
  - number of processes waiting for S to be 0
- **We have queues of processes that are blocked on a semaphore**
- **The system call** *semget* **creates an array of semaphores**
- **The system call** *semop* **performs a list of operations: one on each semaphore (atomically)**

Unix Semaphores

- **Each operation to be done is specified by a value sem_op.**
- **Let S be the semaphore value**
  - if sem_op > 0:
    - S is incremented and process awaiting for S to increase are awaken
  - if sem_op = 0:
    - If S=0: do nothing
    - if S!=0, block the current process on the event that S=0

Unix Semaphores

- if sem_op < 0 and |sem_op| <= S:
  - set S:= S + sem_op (ie: S decreases)
  - then if S=0: awake processes waiting for S=0
- if sem_op < 0 and |sem_op| > S:
  - current process is blocked on the event that S increases

- **Hence: flexibility in usage (many operations are permitted)**