1 Process Interaction

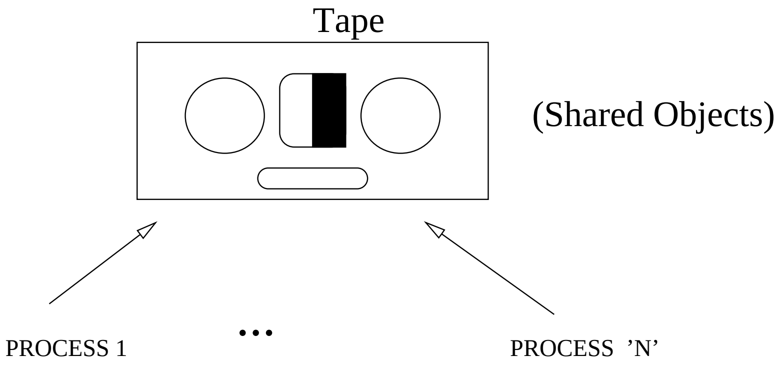

Figure 1:

- Processes are *concurrent* (exist at the same time) - need to cooperate and synchronise activities with each other
- Share resources (such as CPU and I/O) and results with each other - How ?

Shared resources could be :

Data : Program counter, variables or queues
Devices : Printers, tapes, terminals

- Race Conditions (in printing and memory)
- Multiple process writing or modifying simultaneously

2 Too much milk problem

Consider the scenario :

	Person A	Person B
4:30	Look in fridge. Out of milk	
4:35	Leave for Sainsbury	
4:40	Arrive at Sainsbury	Look in fridge. Out of milk
4:45	Buy milk	Leave for Sainsbury
4:50	Arrive home, put milk away	Arrive at Sainsbury
4:55		Buy milk
4:00		Arrive home, put milk ...
4:01		Oh no!!

- Will consider different ways to solve this problem

3 No more milk - some solutions

- Never more than one person buys milk, and someone buys if needed
- Basic Idea (Sol 1) :
  Leave a note (kind of like a "lock")
  Remove note (kind of like an "unlock")
  if note present - don't buy (wait)
- Use *atomic* load and store as building blocks

```
if (noMilk) {
   if (noNote) {
leave Note;
buy milk;
remove note; } }
```

- Process (thread) can get context switched after checking milk and leaving note, but before buying milk
- Bad solution - occasional failure - hard to detect

Solution 2

```
Process A

leave noteA
if (noNoteB) {
 if (noMilk)
  buy milk }
remove noteA

ProcessB

leave noteB
if (noNoteA) {
 if (noMilk)
  buy milk }
remove noteB
```

- Context switch at the wrong time - possible for neither process to buy milk - as each thinks the other is buying milk
- **Starvation** : processes wait forever

Solution 3

```
Process A

leave note A
while (note B) // Pos: 1
  do nothing ; if
 (noMilk)
   buy milk ;
 remove noteA

Process B

leave note B
if (noNoteA) { // Pos: 2
  if (noMilk)
   buy milk }
  remove note B
```

- At Pos1: if noNote A, safe for B to buy (means A has not started yet). If note, A is either buying, or waiting for B to quit, so ok for B to quit
- At Pos2: if noNote B, safe to buy. If note B, don't know. A hangs around. Either: if B buys, done. If B doesn't buy, A will.
- Too complicated - must be simpler way

4 Critical Section

- Critical section or region is the part of a program, where the program accesses the shared (modifiable) resources
- Only one process can be in the critical section at any one time, hence the general concept of :
  entry protocol
  critical section
  exit protocol
- Concept of **mutual exclusion** : processing *mutually* agreeing to *exclude* one another (software) to allow the shared resource to be used efficiently - some solutions are enforced (hardware) however
- Consider the concept of a narrow waterway : *Suez Canal*, *Panama Canal*
- Must be carefully coded to prevent one process from sabotaging the machine, and remaining in the critical section forever
- If a process in mutual exclusion terminates - the operating system must release the mutual exclusion held by that process

Requirements and Implementation :

1. **Mutual Exclusion** : If process P_i is executing in its critical section, then no other process can be executing in that critical section
2. **Progress** : Only processes outside their critical sections can participate in deciding who will enter the critical section - and selection of a waiting one cannot be postponed indefinitely. However, processes cannot prevent any process from entering their critical section if requirements are valid
3. **Bounded Waiting** : There must be a bound on the number of times that other processes are allowed to enter their critical section, after a process has made a request to enter is critical section and before that request is granted
4. No assumption may be made or should be necessary about the relative speed of asynchronous concurrent processes
5.

Two common algorithms :

- *Dekker's Algorithm*
- *Petersons Algorithm.*

5 Example 1

```
var enter1, enter2 = false : boolean ;

Process P1
  loop
  {entry protocol}
  enter1 := true ; {announce intent to enter}

  while enter2
    do null end;
  {wait if other process is in CS}

  {critical section}

  enter1 := false {exit protocol}
  end
end ;

Process P2
 loop
 {entry protocol}
 enter2 := true {announce intent to enter}

 while enter1
   do null end ;
   {wait if other process is in CS}

 {critical section}

   enter2 := false {exit protocol}
   end
end ;
```

6 Example 2

```
var enter1, enter2 = false : boolean ;
    turn = 1 : integer ; {i.e. P1}

Process P1
 loop
 {entry protocol}
 enter1 := true; {announce intent to enter}
 turn := 2 ; {set priority to P2}

 while enter2 and turn=2
   do nothing ; {wait if other process in CS}

 {critical section}

 enter1 := false {exit protocol}
 end
end

Process P2
 loop
 {entry protocol}
 enter2 := true ; {announce intent to enter}
 turn := 1 ; {set priority to other process}

 while enter1 and turn=1
  do nothing ; {wait if other process is in CS}

 {critical section}

 enter2 := false ; {exit protocol}
 end
end
```

From Example 1 :

- Busy Wait
- Deadlock ?
  P1 sets enter1 to true
  P2 sets enter2 to true
  Both processes in busy wait loop
- Busy wait acceptable if anticipated waits are brief ? Costly !
- Alternation (entry into CS) between processes

From Example 2 :

- Break Deadlock - concurrent assignment to turn results in one of the processes having priority
- The *favoured process* concept - and alternate the favoured process (turn) variable

Aim to achieve synchronisation : simplest way -

7 Disable interrupts

- Prevent context switch from occurring in the middle of an operation (locking out external or internal interrupting events)
- Prevent *dispatcher* from getting control
- **Dangerous** for *user* processes (as they may never turn on interrupts again) - crashing machine
- **Hazardous** but generally used by *kernel* commands to update shared areas (variables, data structures etc)

8 Lock and Unlock

- Use *binary* indicators to control access to a shared resource
- L = 1 (means resource is unavailable (*locked*)
  L = 0 (means resource is free (*unlocked*)
- Primitive operations on locks
  LOCK(L) ::= when L=0 do L=1 ;
  UNLOCK(L) :: L=0 ;
- Mutual exclusion can be obtained using locks

```
PROCESS P[i] ;
 begin
   LOCK(L) ;
   {critical section}
   UNLOCK(L) ;
 end
```

Implementation of Locks

- The *indivisible* TESTandSET instruction
- single instruction that reads a variable, stores its value in an allocated area (critical section), and then sets the variable to a certain value
- Instruction once initiated will complete all operations without interruption
- It is difficult to use, understand and verify -and is also wasteful

9 Semaphores

- *Protected* variable, which can only be accessed and modified by *P* and *V* operations, plus some initialisation code
- *Binary* semaphores (values 0 or 1) and *general or counting* semaphores (assume ONLY non-negative integer values)
- P and V are indivisible (cannot be interrupted)
- If several processes attempt *P* simultaneously, only one will be allowed to proceed
- Implementation of P and V ensure that none suffer indefinite postponement

```
P(s) ::= if s>0
 then s := s-1
 else -- suspend process on s ;

V(s) ::= if (process waiting on s)
 then -- resume one of these processes
 else s := s+1;
```

- Binary Semaphore : like a lock
- General semaphore : like a counter (items in a buffer)
- Initial value = number of processes in critical section simultaneously
- For mutual exclusion : initial value = 1 (only 1 process in critical section)

- Synchronisation is dispersed in processes
- Difficult to use, understand and verify (rather unstructured)
- Mistakes are disastrous
- No explicit textual association between semaphore and the date item to which it is synchronising access
- Main synchronisation primitive used in unix

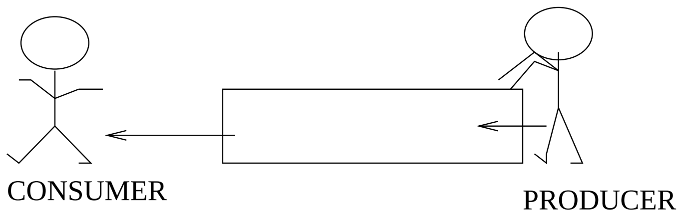

Figure 2:

Producer - Consumer relation

Producer puts things into a buffer, consumer takes them out - need synchronisation for coordination
the Unix pipe :

```
ls | sort | uniq
```

- Producer and Consumer operating at different rates - hence the user of an intermediate buffer
- *Coke Machine* : Producer is delivery person, consumer are staff and students

10 Example : Producer Consumer

```
var mutex = 1 : semaphore ; {binary}
    space = N : semaphore ;
    cans = 0  : semaphore ;

Process PRODUCER;
loop
  {pick can off crate}
  P(space) ; {wait for slot in the coke machine}
  P(mutex) ; {get mutual exclusion}
  {put one can in machine}
  V(mutex} ; {release mutual exclusion}
  V(cans} ; {signal cans available}
end ;

Process CONSUMER;
loop
 P(cans) ; {wait for any cans}
 P(mutex) ; {obtain mutual exclusion}
 {take one can}
 V(mutex) ; {release mutual exclusion}
 V(space) ; {signal space available}
end ;
```

- Solution holds for multiple producers and consumers
- Does order of P and V matter ?
- Does it matter is can is consumed before or after V in consumer ?