Mutual Exclusion Concepts

- **Mutual Exclusion Problem**
  - ♦ **Two or more processes want to use same set of resources. How can we ensure that they gain the access of the resource only one at a time?**
- **Critical Section**
  - ♦ **A section of a program in which the process wants to have exclusive access to one or more shared resources.**
- **Race Condition**
  - ♦ **A situation where several processes access and manipulate the same date item concurrently, and the outcome depends on the order of their execution.**

Three Conditions of a Good Solution

1. **MUTUAL EXCLUSION: Only one process at a time enters CRITICAL SECTION of code.**

2. **PROGRESS: A process which is not requesting entry to CRITICAL SECTION should not block the process(es) who are requesting the entry.**

3. **BOUNDED WAIT: A process should not wait indefinitely before it can enter the CRITICAL SECTION.**

Attempt-1

var turn: 0..1;

Process 0
while turn =! 0 do {nothing};

<CRITICAL SECTION>

turn ;=1;

Process 1
while turn =! 1 do {nothing};

<CRITICAL SECTION>

turn ;=0;

Attempt-2

var flag: array [0..1] of boolean;

[flag[0]=false; flag[1]=false]

Process 0
while flag[1] do {nothing};
flag[0]:=true;

<CRITICAL SECTION>

flag[0] ;=false;

Process 1
while flag[0] do {nothing};
flag[1]=true;

<CRITICAL SECTION>

flag[1] ;=false;

Attempt-3

var flag: array [0..1] of boolean;

[flag[0]=false; flag[1]=false]

Process 0

flag[0]:=true;
while flag[1] do {nothing};

<CRITICAL SECTION>

flag[0] ;=false;

Process 1

flag[1]=true;
while flag[0] do {nothing};

<CRITICAL SECTION>

flag[1] ;=false;

Attempt-4

var flag: array [0..1] of boolean;

[flag[0]=false; flag[1]=false]

Process 0

flag[0]:=true;
while flag[1] do
    begin
    flag[0];= false;
    <delay a while>
    flag[0];=true;
    end
<CRITICAL SECTION>
flag[0] ;=false;

Process 1

flag[1]=true;
while flag[0] do
    begin
    flag[1];= false;
    <delay a while>
    flag[1];=true;
    end
<CRITICAL SECTION>
flag[1] ;=false;

Problem with bounded wait

Dekker's Solution

```
var flag: array [0..1] of boolean;
turn:0..1;
```

[flag[0]=false; flag[1]=false; turn=1]

Process 0

```
begin
  repeat
    flag[0]:=true;
    while flag[1] do if turn=1 then
        begin
        flag[0];= false;
        while turn =1 do {nothing};
        flag[0];=true;
    end
    <CRITICAL SECTION>
    turn=1;
    flag[0] ;=false;
    <REMINDER>
  forever
end;
```

Process 1

```
begin
  repeat
    flag[1]:=true;
    while flag[0] do if turn=0 then
        begin
        flag[1];= false;
        while turn =0 do {nothing};
        flag[1];=true;
    end
    <CRITICAL SECTION>
    turn=0;
    flag[1] ;=false;
    <REMINDER>
  forever
end;
```


Peterson's Solution

```
var flag: array [0..1] of boolean;
turn:0..1;
```

[flag[0]=false; flag[1]=false; turn=1]

Process 0

```
begin
  repeat
    flag[0]:=true;
    turn=1;
    while flag[1] and turn=1 do {nothing};
    <CRITICAL SECTION>
    flag[0] ;=false;
    <REMINDER>
  forever
end;
```

Process 1

```
begin
  repeat
    flag[1]:=true;
    turn=0;
    while flag[0] and turn=0 do {nothing};
    <CRITICAL SECTION>
    flag[1] ;=false;
    <REMINDER>
  forever
end;
```

Hardware Supported Solution-1: Test-and-Set

Test the value of lock and try to lock it if it was open

If the target is 1 it sets it to 1 and returns success (true).
If the target is 0 it sets it to 1 and returns fail (false).

```
function test-and-set( var target ; boolean) : boolean;
begin
        if (target) test-and-set:=true;
        else test-and-set:=false;
        target:=true;
end;
```


Mutual Exclusion by Test-and-Set

Wait Until the “lock” is open (0)

```
Repeat

        while test-and-set(lock) do no-op;
        <CRITICAL SECTION>
        lock:=false;
        <REMINDER>

until false;
```

Hardware Supported Solutions-2: Swap

Swaps the value of target with your own value
Always returns 1

```
function swap( var target, key ; boolean);
var temp: boolean;
begin
        temp=target;
        target=key;
        key=temp;
end;
```


Mutual Exclusion by Swap

Make a key and wait until the “lock” is open (0)

```
Repeat

        key:=true;
        repeat
                swap((lock, key);
        until key=false;
        <CRITICAL SECTION>
        lock:=false;
        <REMINDER>

until false;
```

Mutual Exclusion?
Progress?
Bounded Wait?

Synchronization Concepts (more..)

- **Busy Wait**
  - ♦ **A type of wait when a waiting process still occupies CPU time by continuous checking.**

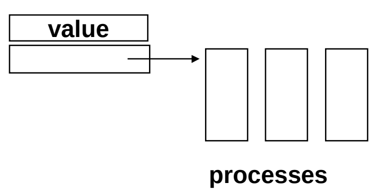
- **Solution:**
  - ♦ **OS assisted wait: OS puts a waiting process into sleep and wakes it up when the special event occurs.**

Operating Systems Lecture 9

Semaphores

- **A special synchronization variable S called semaphore (defining the event) is announced to OS.**
- **The process want to wait for the event use a call Wait(S)**
- **The process executing the event, after the event completes use a call Signal(S)**
- **OS makes sure that only one process can modify S at a time.**

```
Type semaphore = record
        value:integer;
        L: list of process;
    end;
```



Semaphore Implementation

```
Wait(S) S.value=S.value-1
if S.value <0
then begin
      add this process to S.L;
      block();
end;
```

```
Signal(S) S.value=S.value+1
if S.value <=0
then begin
     remove a process P from S.L;
     wakeup(P);
end;
```

•OS must ensure that no two processes can execute wait() and signal() on the same semaphore at the same time.
•OS can use hardware or SW solutions.

[redacted]

```
// synch.cc
#include "copyright.h"
#include "synch.h"
#include "system.h"
//------------------------------------
// Semaphore::Semaphore
//------------------------------------
Semaphore::Semaphore(char* debugName, int
initialValue)
{
    name = debugName;
    value = initialValue;
    queue = new List;
}
//------------------------------------
// Semaphore::Semaphore
//------------------------------------
Semaphore::~Semaphore()
{
    delete queue;
}
//------------------------------------
// Semaphore::P or Wait
//------------------------------------
void
Semaphore::P()
{
    IntStatus oldLevel = interrupt->SetLevel(IntOff);
            // disable interrupts

    while (value == 0) {
            // semaphore not available
            queue->Append((void *)currentThread);
            // so go to sleep
            currentThread->Sleep();
    }
    value--;
                    // semaphore available,

                    // consume its value

    (void) interrupt->SetLevel(oldLevel);       // re-
enable interrupts
}
```

```
//------------------------------------
// Semaphore::V  or Wakeup
//------------------------------------
void
Semaphore::V()
{
    Thread *thread;
    IntStatus oldLevel = interrupt->SetLevel(IntOff);

    thread = (Thread *)queue->Remove();
    if (thread != NULL)    // make thread ready,
consuming the V immediately
            scheduler->ReadyToRun(thread);
    value++;
    (void) interrupt->SetLevel(oldLevel);
}
// Dummy functions -- so we can compile our later
assignments
// Note -- without a correct implementation of
Condition::Wait(),
// the test case in the network assignment won't work!
Condition::Condition(char* debugName) { }
Condition::~Condition() { }
void Condition::Wait(Lock* conditionLock) {
ASSERT(FALSE); }
void Condition::Signal(Lock* conditionLock) { }
void Condition::Broadcast(Lock* conditionLock) { }
```

Semaphore Solution: Mutual Exclusion

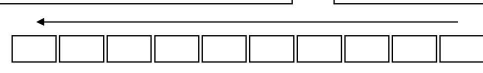
```
Var mutex: semaphore;

Repeat
        wait(mutex);
        <CRITICAL SECTION>
        signal(mutex);

        <REMINDER>

until false;
```

Semaphores can be binary type. Three binay semaphores can be used to implements one counting semaphore.

Semaphore Solution: Bounded-Buffer Problem

```
Var empty, full, mutext: semaphore;
Producer
Var nextp: item;
Repeat
        produce item nextp;
        wait(empty);
        wait(mutex);
        add nextp to buffer;
        signal(mutex);
        signal(full);
until false;
```

```
Var empty, full, mutext: semaphore;
Consumer
Var nextc: item;
Repeat
  wait(full);
  wait(mutex);
  remove item from buffer to nextc;
  signal(mutex);
  signal(empty);
until false;
```



Semaphore Solution: Readers and Writers Problem

Wnen one writer is in no reader or writer is allowed
More that one reader can be active at a time
Readers don't wait for waiting writer

```
Var mutex, wrt: semaphore;
Var readcount;

Writer

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

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


Language Construct: Conditional Critical Region

Shared variable — Critical section code

Declare the shared variables: **var v: shared T;**
Declare the critical section: **region v when B do S;**

condition

Example: Producer Consumer Problem

Shared Variable:

```
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;
```

Semaphore Implementation of Critical Region

```
Var mutex, first-delay, second-delay: semaphore;
first-count, second-count: integer;
region x when B do S;
wait(mutex);
while not B
   do begin
         first-count=first-count+1;
         if second-count>0
                  then signal(second-delay)
                  else signal(mutex);
         wait(first-delay);
         first-count=first-count-1;
         second-count=second-count+1;
         if first-count>0
                  then signal(first-delay);
                  else signal(second-delay);
         wait(second-delay);
         second-count=second-count-1;
   end;
   (continued…)
```

Mutex == only one executes region
First Delay == just checked B
Second Delay == ready to check B again

```
S;
if first-count>0
   then signal(first-delay);
   else  if second-count>0
         then signal(second-delay);
         else signal(mutex);
end
.
.
```


Language Construct: Monitors with Condition Variables

Declare the exclusive routines inside the Monitor
Declare the shared variables inside the Monitor

Declare the condition variables
Monitor routines can temporarily wait on conditional variables (cwait(x) and csignal(x))

```
Type monitor-name= monitor
variable declarations;
 procedure entry P1(…);
      begin…end;
 procedure entry P2(…);
      begin…end;
 ...
 procedure entry Pn(…);
      begin…end;
end;
```

Monitor Example: Producer Consumer Problem

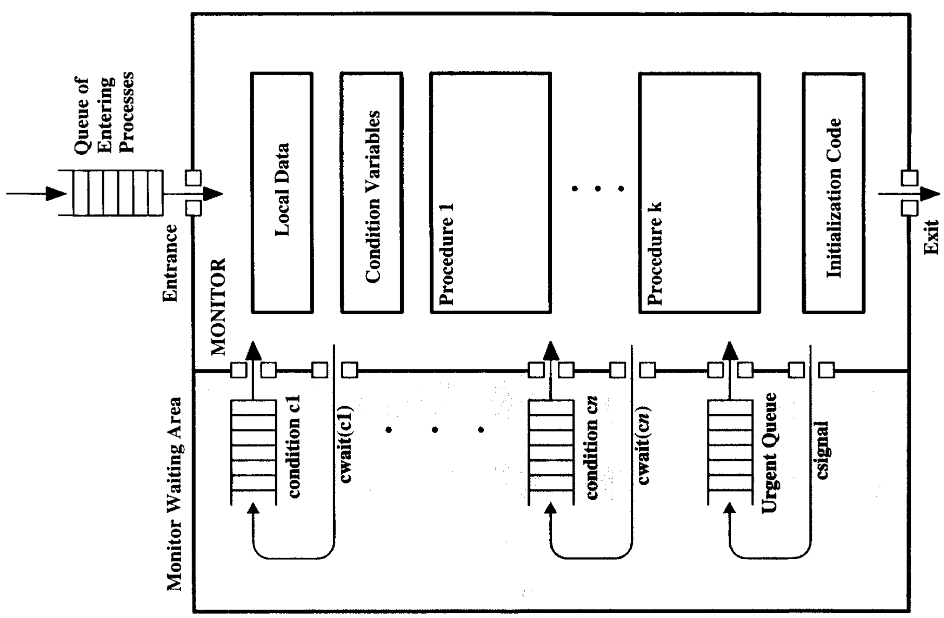
```
Program producerconsumer
monitor boundedbuffer;
        pool:array[0..N] of item;
        in, out:integer;
        count:integer;
        notfull, notempty:condition;
begin in=0; out=0; count=0; end;
procedure append(x: item);
        begin
                if count=N then cwait(notfull);
                pool[in]:=x;
                in=in+1 mod N;
                count=count+1;
                csignal(notempty);
        end;
procedure take(x: item);
        begin
                if count=0 then cwait(notempty);
                x=pool[out];
                out=out+1 mod N;
                count=count-1;
                csignal(notfull);
        end;
```

Monitor

Shared Variables

Condition variables

```
Procedure producer;
var x: item;
begin
        repeat
                produce(x);
                appen(x);
        forwever;
end;
Procedure consumer;
var x: item;
begin
        repeat
                take(x);
                consume(x);
        forwever;
end;
```

Source: Fig-5.22, OS Design Principles, Stalling, 1998

Condition Variables: Finer Issues

- **Difference between semaphore wake() and signal() and condition variables cwait() and csignal().**
  - ♦ **If no process is waiting csignal() has no effect.**
  - ♦ **Semaphore signal() still increments the semaphore value.**

- **Performs a csignal(). Q is waked up. Who gets to execute?**
  - ♦ **Logical: P should contimue and leave monitor and then Q gets a chance.**
  - ♦ **But what if P now changes the condition again? P should wait.**
  - ♦ **Middle ground: any one executing csignal() must exit immediately.**

Hoare's choice for simplicity of proof

Example: Solaris 2

Data Sharing & Synchronization

- ♦ **Pipes, Messages, Shared memory**

Technique	Primitives	Actions
Mutual Exclusion Lock	Mutex_enter() Mutex_exit() Mutex_tryenter()	Acquire lock, block if it already held. Release lock, unblock waiter if any. Non blocking enter.
Semphores	Sema_p() Sema_v() Sema_tryp()	 Decrements the semaphore if blocking not required.
Reader/Writer Lock	Rw_enter() RW_exit() Rw_tryenter() Rw_downgrade() Rw_tryupgrade()	 Write lock to read lock Try for write lock from readlock
Condition Variables	Cv_wait() Cv_signal() Cv_broadcast()	 Wakeup all waiters