High Level Synchronization Constructs

Critical Regions

- A shared variable *v* of type *T*, is declared as:

  shared *T v;*

- Variable *v* accessed only inside statement

  region *v* **do** *S*

  While statement *S* is being executed, no other process can access variable *v*.

Critical Regions

```
cobegin
region v do S1;
region v do S2;
coend
```

Variable v can be used by a single process at a time

Nested Critical Regions

```
shared V v; shared W w;
    region v do
        begin
          ...
          region w do ... ;
          ...
        end
```

Critical Regions Implementation

shared *T v;*

```
Typedef struct v_tag {
  T  val;
  semaphore_t me;
} v_t;

v_t v;
```

```
Region v do S;

INIT( v.me ) = 1;

wait( v.me );
  S;
signal( v.me );
```


Problems with Nested Critical Regions

P: region v do region w do S1;
Q: region w do region v do S2;

```
(* P *)                (* Q *)
wait( v.me );          wait( w.me );
wait( w.me );          wait( v.me );
S1;                    S2;
signal( w.me );        signal( v.me );
signal( v.me );        signal( w.me );
```


Producer-Consumer with Critical Regions

```
typedef struct BUFFER_tag {
  shared Message buffer[MAX];
  shared int p = 0, c = 0;
  semaphore_t full=0, empty=MAX;
} Buffer;
```

```
send(Message m, Buffer b){
  WAIT( b.empty );
  region b.p do {
    buffer[b.p] = m;
    b.p = (b.p + 1) % MAX;
  }
  SIGNAL( b.full );
}
```

```
receive(Message m, Buffer b){
  WAIT( b.full );
  region b.c do {
    m = buffer[b.c];
    b.c = (b.c + 1) % MAX;
  }
  SIGNAL( b.empty );
}
```

Conditional Critical Regions: when B

region v **when** **B** **do** S;

- Regions referring to the same shared variable exclude each other in time.
- When a process tries to execute the region statement, the Boolean expression *B* is evaluated.
  - If *B* is true, statement *S* is executed.
  - If B is false, the process is delayed until *B* becomes true and no other process is in the region associated with *v*.

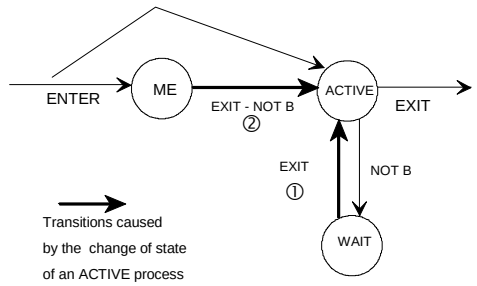
Prod-Cons with Conditional Critical Regions

```
typedef struct BUFFER_tag {
  shared Message buffer[MAX];
  shared int p = 0, c = 0;
  shared int count=0;
} Buffer;
```

```
send(Message m, Buffer b){
  region b.count
    when b.count < MAX do {
    region b.p do {
      buffer[b.p] = m;
      b.p = (b.p + 1)% MAX;
    }
    b.count++;
  }
}
```

```
receive(Message m, Buffer b){
  region b.count
    when b.count > 0 do {
      region b.c do {
        m = buffer[b.c];
        b.c = (b.c + 1)% MAX;
      }
      b.count --;
  }
}
```


Implementation of region **v** when **B** do **S1;**



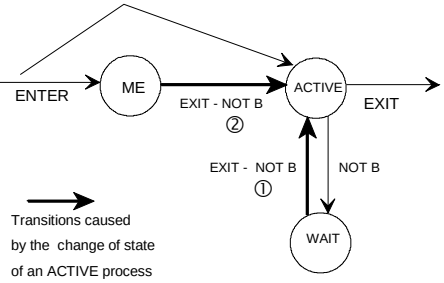
State Diagram of a process

Conditional Critical Regions: await B

region v **do** S1 **await** **B** **do** S2;

- A process can execute S1 – in mutual exclusion – before testing the boolean expression B and possibly be blocked.
- Compare with **region** v **when** **B** **do** S;

Implementation of region **v** do **S1** await **B** do **S2;**



State Diagram of a process

Implementation of region **v** do **when** **B** do **S2;**

For every variable **v** declared **shared** the compiler produces a structure with the following fields:

- **v**: the value of **v**.
- **mutex**: semaphore of mutual exclusion that protect variable v.
- **delay**: semaphore that blocks the processes whose condition **B** is false.
- **count**: number of processes blocked on semaphore **delay**.
- **temp**: counter of the number of processes in the list of semaphore **delay** that have already tested their condition after a process has leaved its critical region.

Implementation of region v do when B do S

```
wait( v.mutex );
if (!B) {
  v.count++;
  SIGNAL( v.mutex );
  WAIT( v.delay );
  while (!B) {
    v.temp++;
    if (v.temp < v.count)SIGNAL( v.delay );
    else SIGNAL( v.mutex );
    WAIT( v.delay );
  }
  v.count--;
}
S;  (* Do something with v.val *)
if (v.count > 0) {
  v.temp = 0;
  SIGNAL( v.delay );
}
else
  SIGNAL(v.mutex);
```


Readers & Writers

- A class of process called Readers that can access a database in parallel
- A class of process called Writers that must access a database in mutual exclusion with other Writers and Readers processes.

Readers & Writers : Readers Precedence

During the access of a Writer to the database, several Readers and Writers processes can be blocked outside their CCRs waiting the end of the Writer operation.

In this situation to give precedence to the Readers means favour the access of the waiting Readers rather than of the waiting Writers.

Implementation of R & W with Semaphores Readers Precedence

```
nr = 0  (* number of Readers currently reading *)
INIT(w) = INIT(me) = INIT(me1) = 1
```

READER

```
WAIT(me);
  nr++;
  if(nr == 1) WAIT(w);
SIGNAL(me);
⋮
//READING
⋮
WAIT(me);
  nr--;
  if(nr == 0) SIGNAL(w);
SIGNAL(me);
```

WRITER

```
WAIT(me1);
  WAIT(w);
  ⋮
   //WRITING
  ⋮
  SIGNAL(w);
SIGNAL(me1);
```


Implementation of R & W with Semaphores Writers Precedence

```
nr=nw=0; INIT(r)=INIT(w)=INIT(me)=INIT(me1)=INIT(me2)=1
```

READER

```
WAIT(me2);
  WAIT(r);
    WAIT(me);
      nr++;
      if(nr == 1) WAIT(w);
    SIGNAL(me);
  SIGNAL(r);
SIGNAL(me2);
     // READING
WAIT(me);
  nr--;
  if(nr == 0) SIGNAL(w);
SIGNAL(me);
```

WRITER

```
WAIT(me1);
  nw++;
  if(nw == 1) WAIT(r);
SIGNAL(me1);
  WAIT(w);
   ⋮
   //    WRITING
   ⋮
  SIGNAL(w)
WAIT(me1);
  nw--;
  if(nw == 0) SIGNAL(r);
SIGNAL(me1);
```


Class

- Is an abstract data type that encapsulates both data and the functions that operates on them.
- The functions are not visible outside the class if they are preceded by the keyword **`private`**.

Class Example: memory

```
const int N = 100; // number of memory pages
class memory {
private:
  char free[N];
public:
  void init(void);
  int acquire(void);
  void release (int index);
};
```


Class Example: memory

```
int acquire () {
  int index;
  for (index=0; index < N; index++) {
     if ( free[index] ) {
        free[index] = FALSE;
        return (index);
  }
  return (-1);
}
```


Class Example: memory

```
void release (int index) {
  free[index] = TRUE;
}
void init(void) {
 for (index=0; index < N; index++)
   free[index] = TRUE;
}
```

Class memory Use Example

```
main() {
  memory mem1;
  memory mem2;
  int k1, k2;
  ⋮
  mem1.init();
  mem2.init();
  ⋮
  k2=mem2.acquire();
  ⋮
  k1=mem1.acquire();
  ⋮
```

```
// continue here
  ⋮
  mem1.release(k1);
  ⋮
  mem2.release(k2);
  ⋮

}
```


Extension of a class to a concurrent environment

```
int acquire () {
  int index;
  region free {
    for (index=0; index < N; index++) {
       if ( free[index] ) {
          free[index] = FALSE;
          return (index);
    }
  }
  return (-1);
}
```


R & W implementation with CCR

```
class RW_Readers_Precedence {
private:
  int nr, nw;
  char busy;
public:     // executed in Mutual Exclusion
  void start_read(void);
  void start_write(void);
  void end_read(void);
  void end_write(void);
  void (init(void));
};
```

R & W : Readers Precedence

```
class RW_Readers_Precedence {
private:
  int nr, nw;
  char busy;
public:
  void start_read(void);
  void start_write(void);
  void end_read(void);
  void end_write(void);
  void init(void);
};
```


R & W: Readers Precedence

```
void start_read(void) {
 nr++;   // S1
 await (nw == 0);
}
```

```
void start_write(void) {
  await (!busy && nr ==0);
  nw++;
  busy = TRUE;
}
```

```
void end_read(void) {
  nr--;
}
```

```
void end_write(void) {
   busy = FALSE
   nw--;
}
```

```
void init (void){
  busy = FALSE;
  nr = nw = 0;
}
```


Example of Use of Class R&W

```
main() {
RW_Readers_Precedence db;

db.init();
db.start_read();
  READ ( …… );
db.end_read();

}
```

R & W: Writers Precedence

```
class RW_Writers_Precedence {
private:
  int nr, nw;
  char busy;
public: // executed in Mutual Exclusion
  void start_read(void);
  void start_write(void);
  void end_read(void);
  void end_write(void);
  void init(void);
};
```


R & W: Writers Precedence

```
void start_read(void) {
  await (nw == 0);
  nr++;
}
```

```
void start_write(void) {
  nw++;
  await (!busy && nr ==0);
  busy = TRUE;
}
```

```
void end_read(void) {
  nr--;
}
```

```
void end_write(void) {
  busy = FALSE
  nw--;
}
```

```
void init (void){
  busy = FALSE;
  nr = nw = 0;
}
```