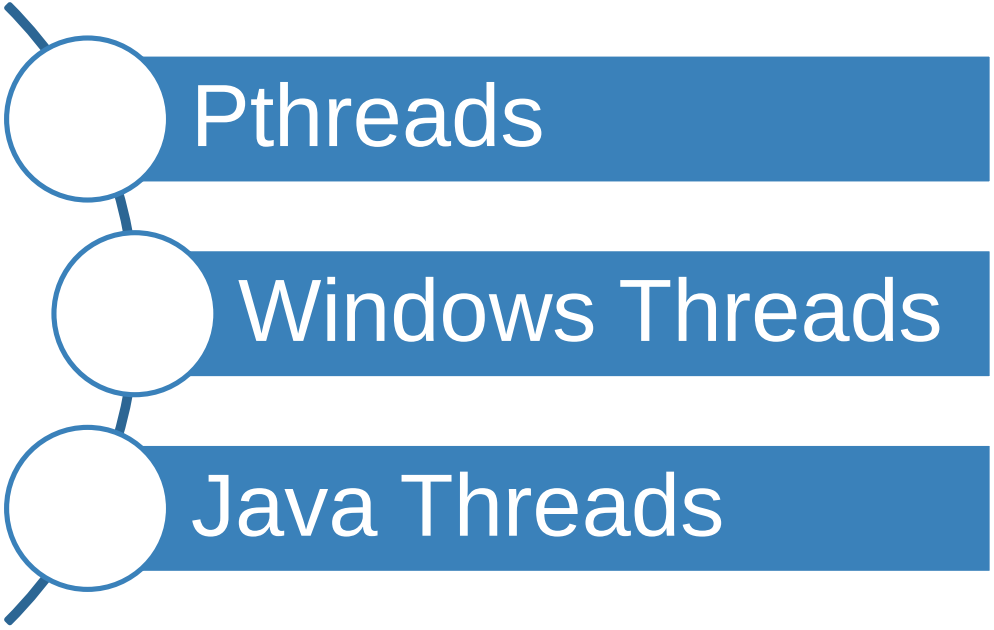
Thread Library



Pthreads

Figure 4.9 Multithreaded C program using the Pthreads API.

```c
#include <pthread.h>
#include <stdio.h>

int sum; /* this data is shared by the thread(s) */
void *runner(void *param); /* threads call this function */
int main(int argc, char *argv[])
{
pthread t tid; /* the thread identifier */
pthread attr t attr; /* set of thread attributes */
if (argc != 2)
    {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0)
    {
        fprintf(stderr,"%d must be >= 0\n",atoi(argv[1]));
        return -1;
    }
/* get the default attributes */
pthread attr init(&attr);
/* create the thread */
pthread create(&tid,&attr,runner,argv[1]);
/* wait for the thread to exit */
pthread join(tid,NULL);
printf("sum = %d\n",sum);
}
/* The thread will begin control in this function */
void *runner(void *param)
{ int i, upper = atoi(param);
sum = 0;
for (i = 1; i <= upper; i++)
sum += i;
pthread exit(0);
}
```

Paralelismo

- Se logra separando procesos en ejecución.
- Existen dos tipos:
  - Data Parallelism
  - Task Parallelism

  Siendo distinto lo que se distribuye entre los núcleos.

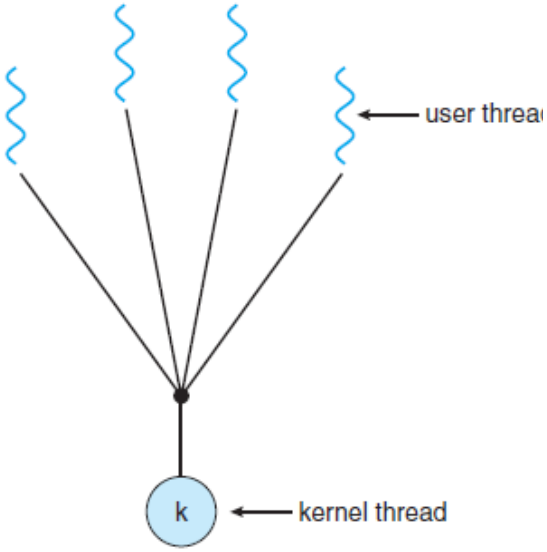


Figure 4.5 Many-to-one model.

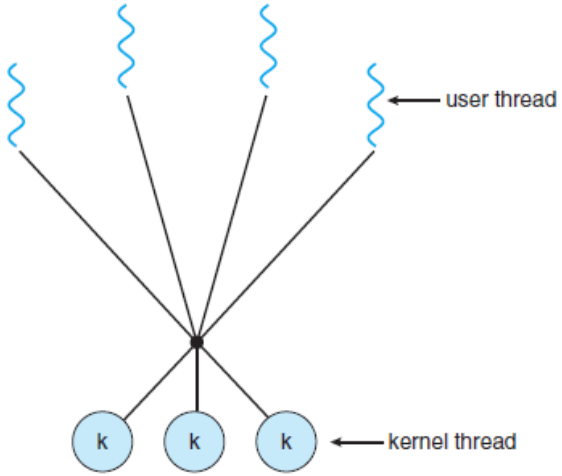


Figure 4.7 Many-to-many model.

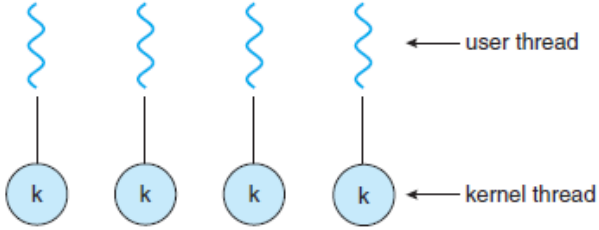
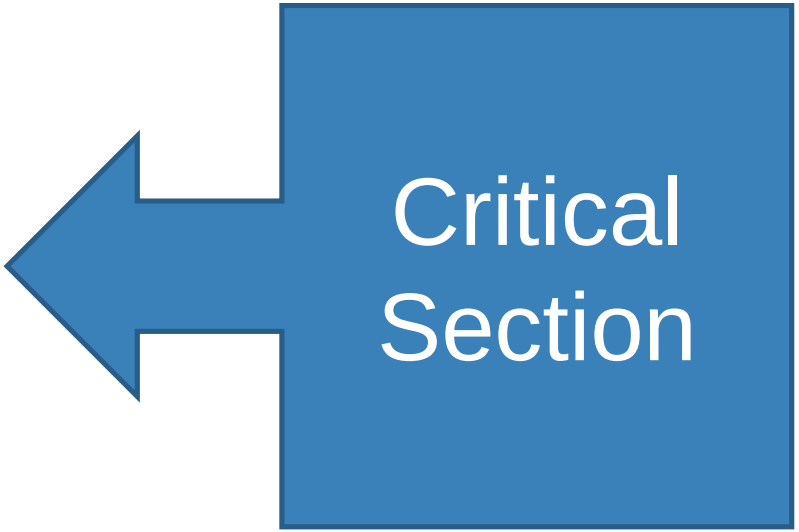


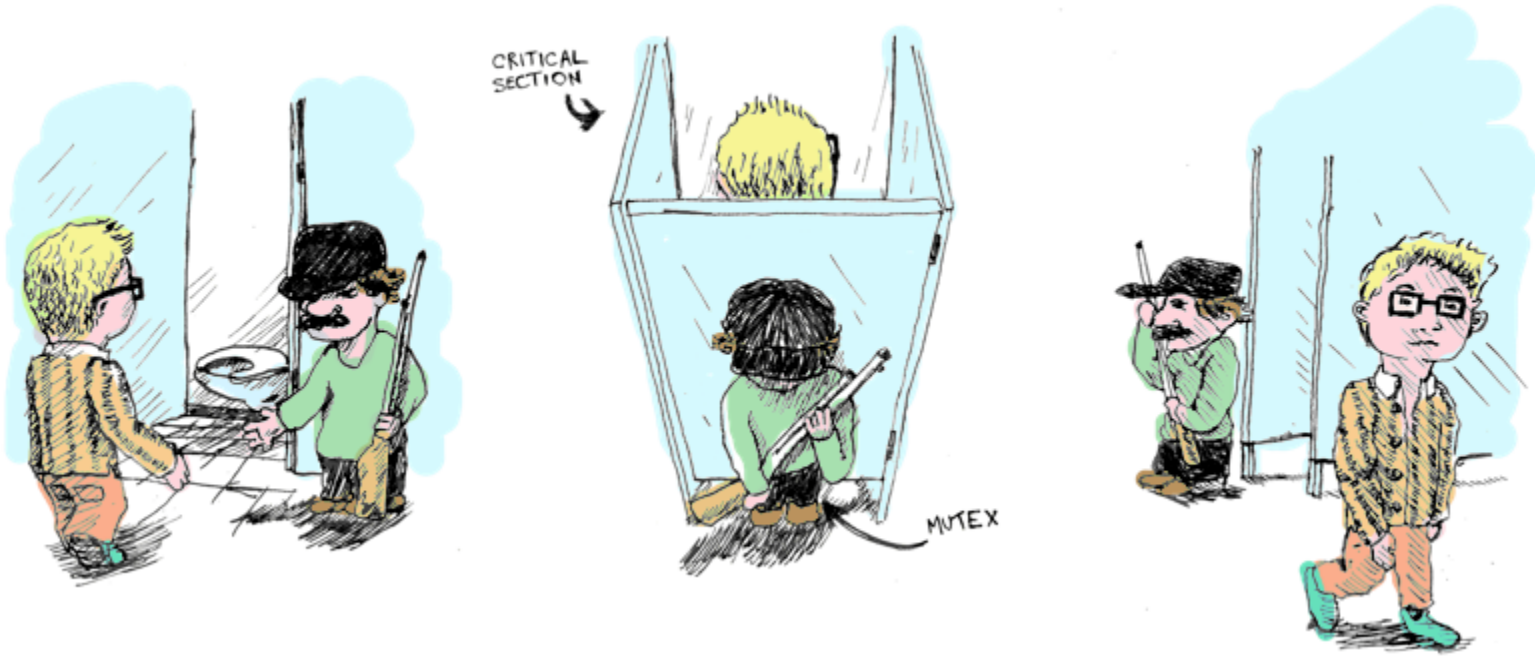
Figure 4.6 One-to-one model.

Mutual Exclusion

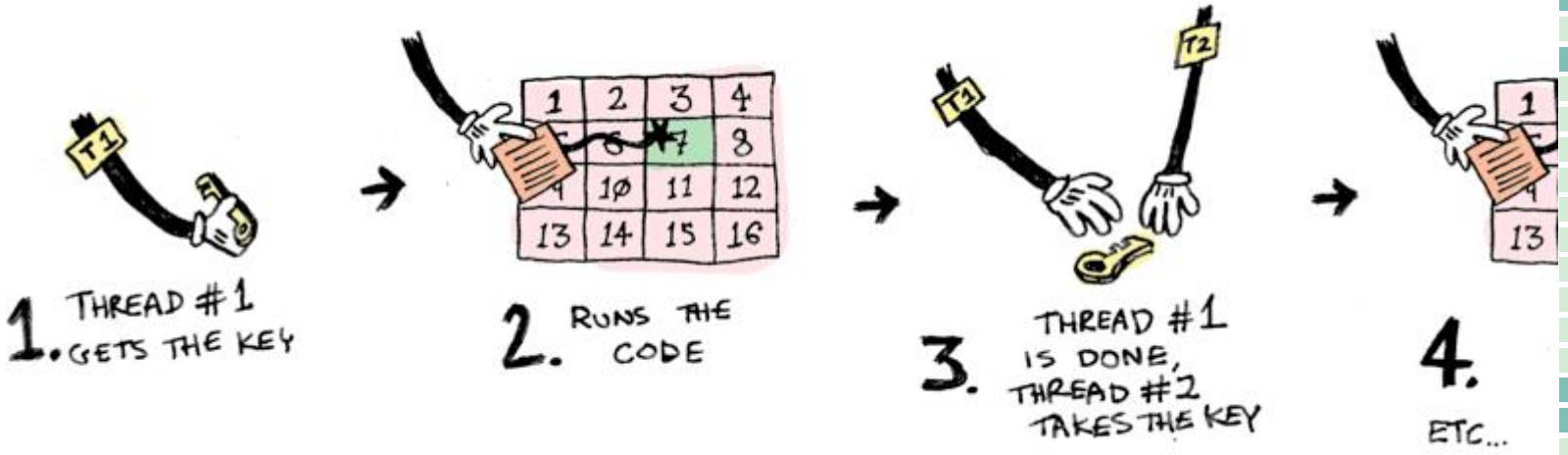
```
mutex:
      lock(mutex)
      {

      }
```





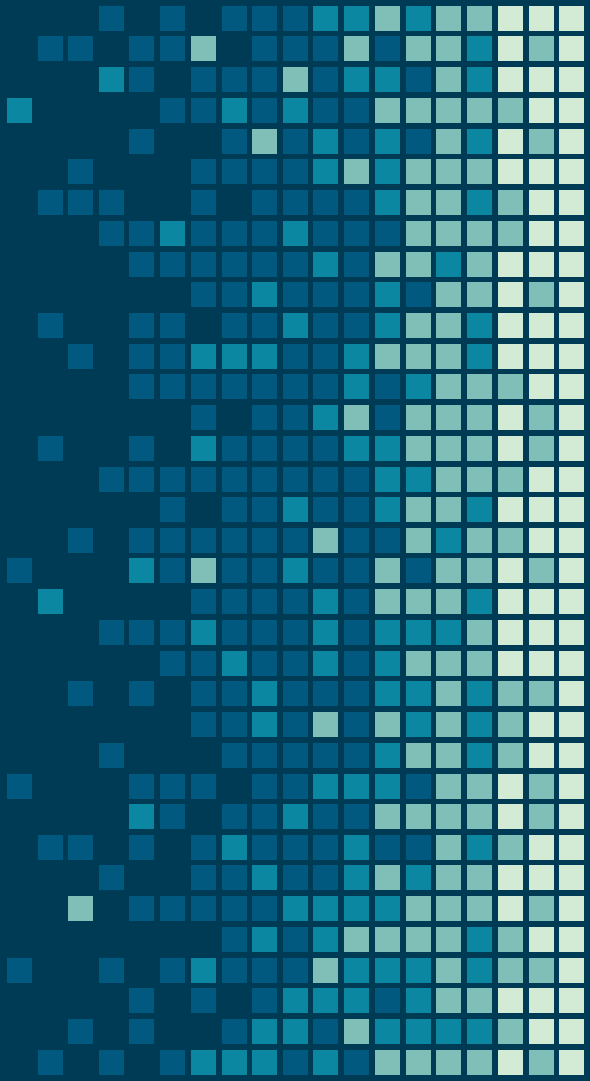
http://www.rudyhuyn.com/blog/2015/12/31/synchroniser-ses-agents-avec-lapplication/mutex/



http://mishadoff.com/blog/programming-digest-3/

Sistemas Operativos I

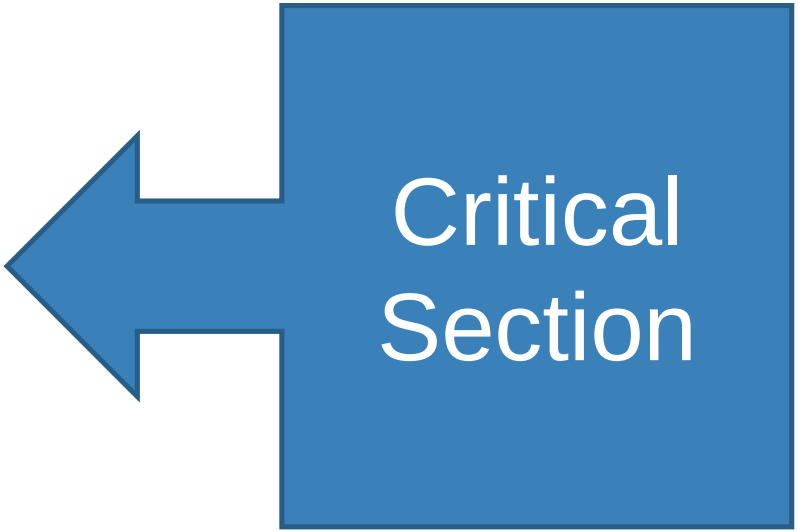
Synchronization Tools



Mutual Exclusion

```
mutex:
      lock(mutex)
      {

      }
```

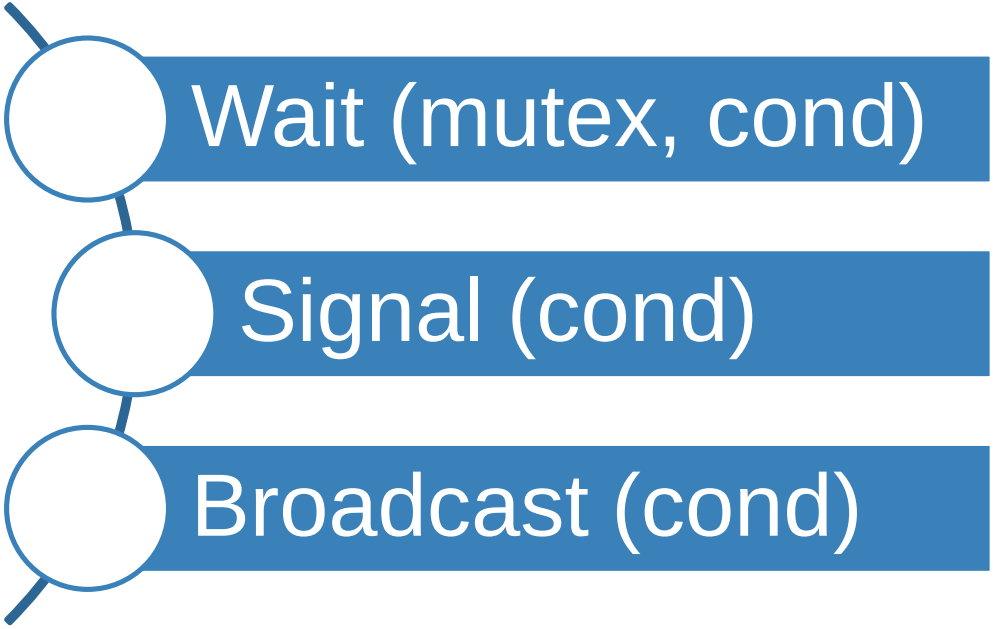


Mutual Exclusion

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (true);
```

Figure 5.8 Solution to the critical-section problem using mutex locks.

Condition Variable



Semáforos

- Upgrade de Mutex
- Representa una variable entera con valor de 0 o superior.
- Se basa en dos operaciones atómicas “Signal” y “Wait”.

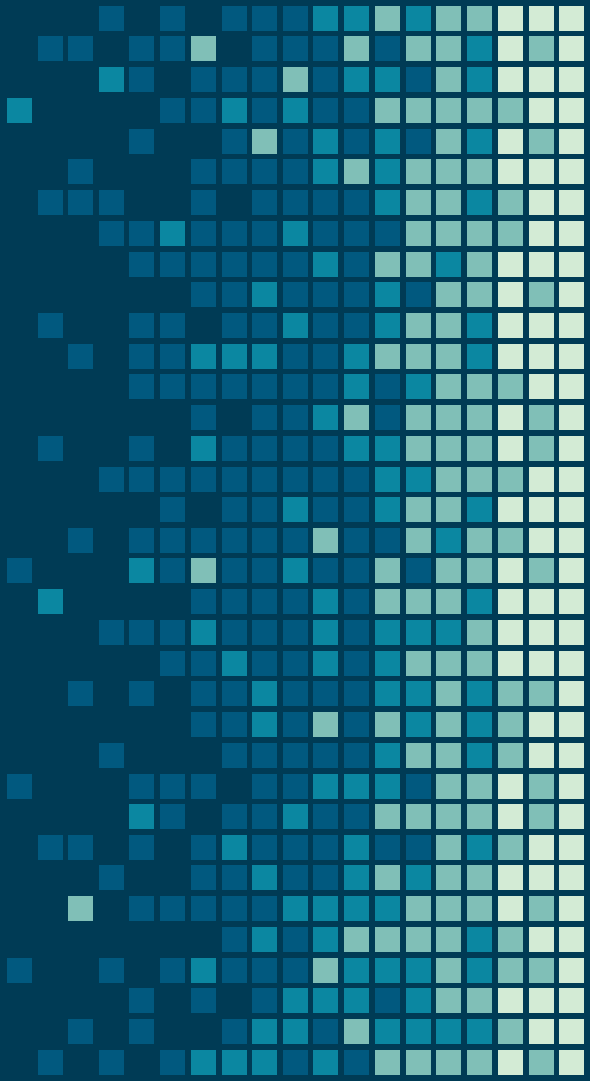
```
wait(S) {
    while (S <= 0)
       ; // busy wait
    S--;
}
```

```
signal(S) {
    S++;
}
```

Semáforos

- Semáforo contador: permite el acceso a mas de un elemento a la vez.
- Semáforo binario: permite el acceso a un elemento a la vez.

Deadlocks



Deadlock

- Se crean cuando un thread necesita utilizar recursos que están siendo utilizados por otro.
- Ejemplo básico:
  - Thread que ejecuta un ciclo en su Zona Crítica.

Deadlock

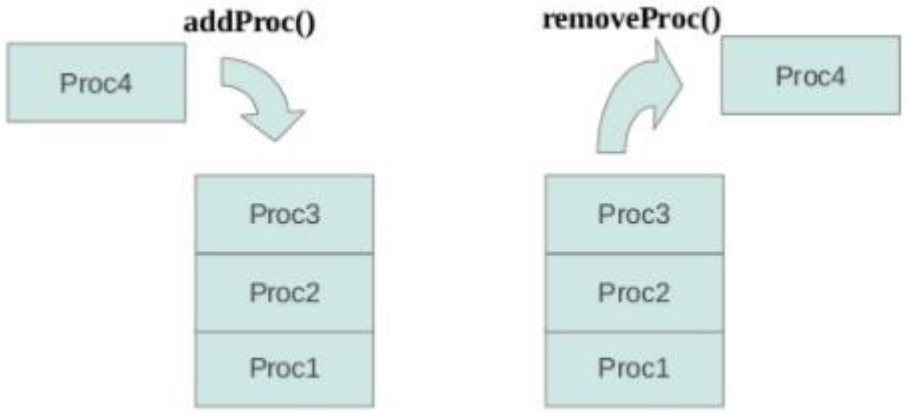
P_0	P_1
`wait(S);`	`wait(Q);`
`wait(Q);`	`wait(S);`
.	.
.	.
.	.
`signal(S);`	`signal(Q);`
`signal(Q);`	`signal(S);`

¿Qué se puede hacer al respecto?

- Prevención (alto en recurso).
- Detección y recuperación (rollback)
- Aplicar el algoritmo del avestruz (Ostrich algorithm)

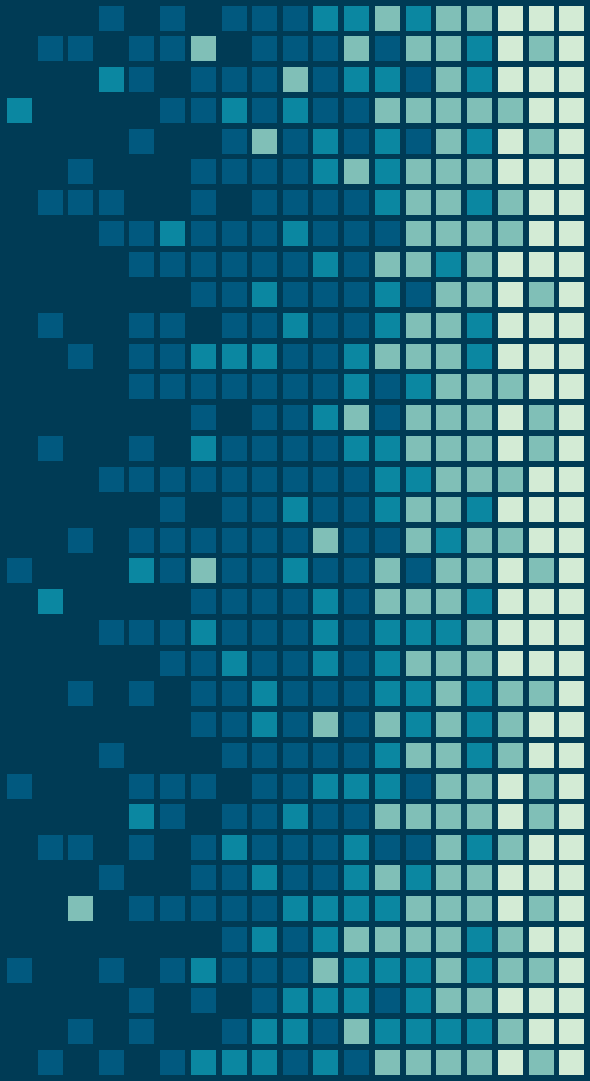
Indefinite Blocking

- Postergación Indefinida. También conocido como Starvation.



Synchronization Primitives cont…

…Classic Problems



The Bounded-Buffer Problem

```
int n;
semaphore mutex = 1;
semaphore empty = n;
semaphore full = 0
```

```
while (true) {
     /* produce an item in next_produced */

     while (counter == BUFFER_SIZE)
        ; /* do nothing */

     buffer[in] = next_produced;
     in = (in + 1) % BUFFER_SIZE;
     counter++;
}
```

```
while (true) {
     while (counter == 0)
        ; /* do nothing */

     next_consumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     counter--;

     /* consume the item in next_consumed */
}
```

The Readers–Writers Problem

- Uno o mas lectores pero no escritores tienen acceso al recurso. Un escritor sin lectores tienen acceso al recurso.

```
semaphore rw_mutex = 1;
semaphore mutex = 1;
int read_count = 0;
```

The Readers–Writers Problem

```
do {
   wait(rw_mutex);
      . . .
   /* writing is performed */
      . . .
   signal(rw_mutex);
} while (true);
```

Figure 5.11 The structure of a writer process.

```
do {
   wait(mutex);
   read_count++;
   if (read_count == 1)
      wait(rw_mutex);
   signal(mutex);
      . . .
   /* reading is performed */
      . . .
   wait(mutex);
   read_count--;
   if (read_count == 0)
      signal(rw_mutex);
   signal(mutex);
} while (true);
```

Figure 5.12 The structure of a reader process.

The Dining-Philosophers Problem

- También conocido como Algoritmo de los Filósofos Comensales.

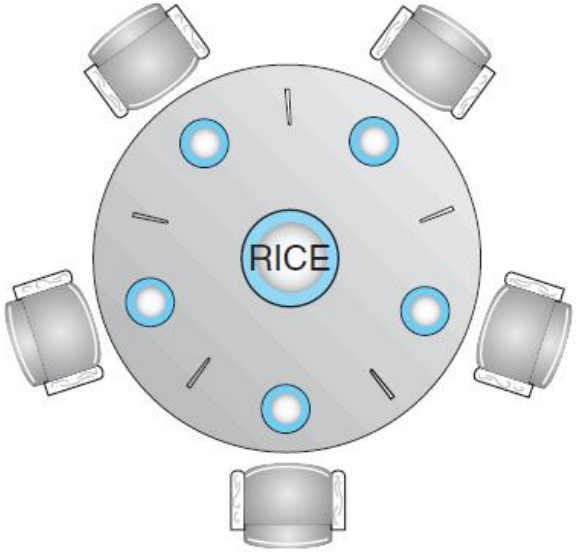


Figure 5.13 The situation of the dining philosophers.

```
semaphore chopstick[5];

do {
   wait(chopstick[i]);
   wait(chopstick[(i+1) % 5]);
      . . .
   /* eat for awhile */
      . . .
   signal(chopstick[i]);
   signal(chopstick[(i+1) % 5]);
      . . .
   /* think for awhile */
      . . .
} while (true);
```

Figure 5.14 The structure of philosopher *i*.

Monitores

- Sincronización a alto nivel.

```
monitor monitor name
{
   /* shared variable declarations */

   function P1 ( . . . ) {
      . . .
   }

   function P2 ( . . . ) {
      . . .
   }

         .
         .
         .
   function Pn ( . . . ) {
      . . .
   }

   initialization_code ( . . . ) {
      . . .
   }
}
```

Figure 5.15 Syntax of a monitor.

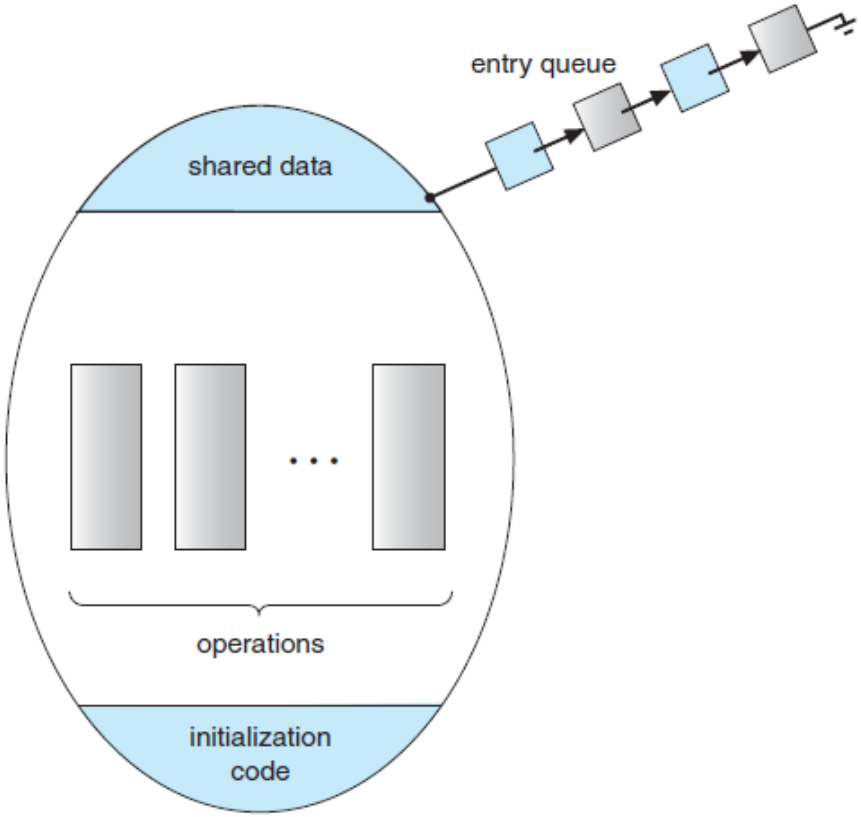


Figure 5.16 Schematic view of a monitor.

Proyecto:

THANKS!

Any questions?