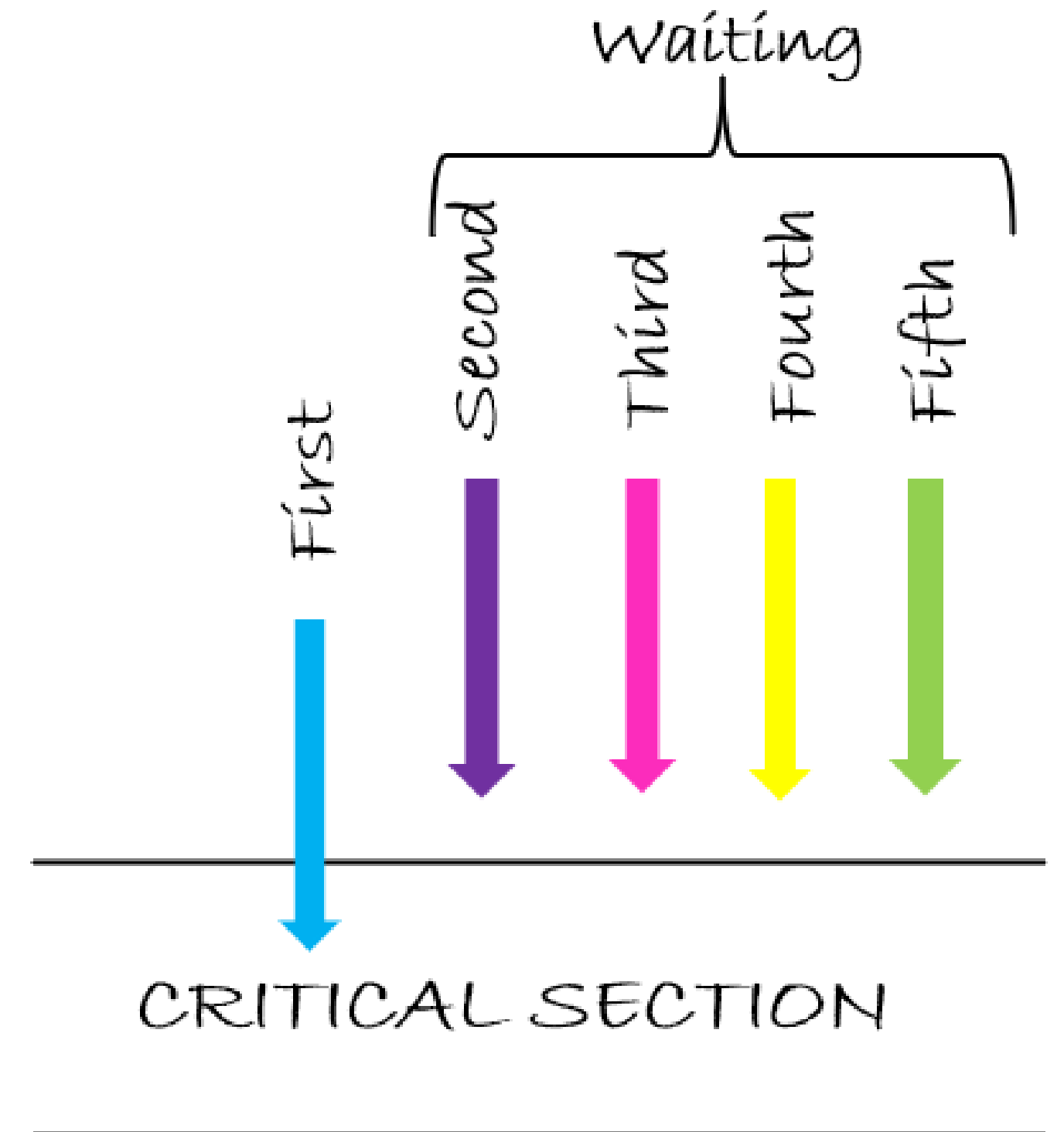


# Monitores

Barajas Ruiz Leslie  
Benites Onofre Fernando Gabriel  
Chaparro Sicardo Tanibeth  
Lerín Hernández Natalia

# Introducción

Un mecanismo de sincronización nos permite a coordinar el acceso a un recurso compartido cuando estamos implementando concurrencia, garantizando la exclusión mutua y evitando las condiciones de carrera.

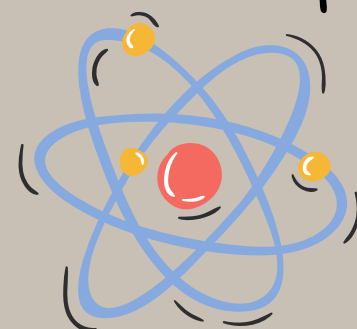


Sección crítica

Condiciones de carrera



Operación atómica



Exclusión Mutua



Espera limitada



# Pero, ¿por qué?

La implementación de los monitores surge de la problemática que nos genera el uso de semáforos, ya que por su estructura no se tiene una visión global de la sincronización, es decir, a cada uno de los procesos se le aplica el `wait()` y `signal()` afectando fuertemente el acceso simultáneo a la sección crítica cayendo en bloqueos mutuos o inanición.

## Semáforos

- Recursos globales.
- Susceptible a errores.
- Código disperso.
- Operaciones no restringidas.

## Monitores

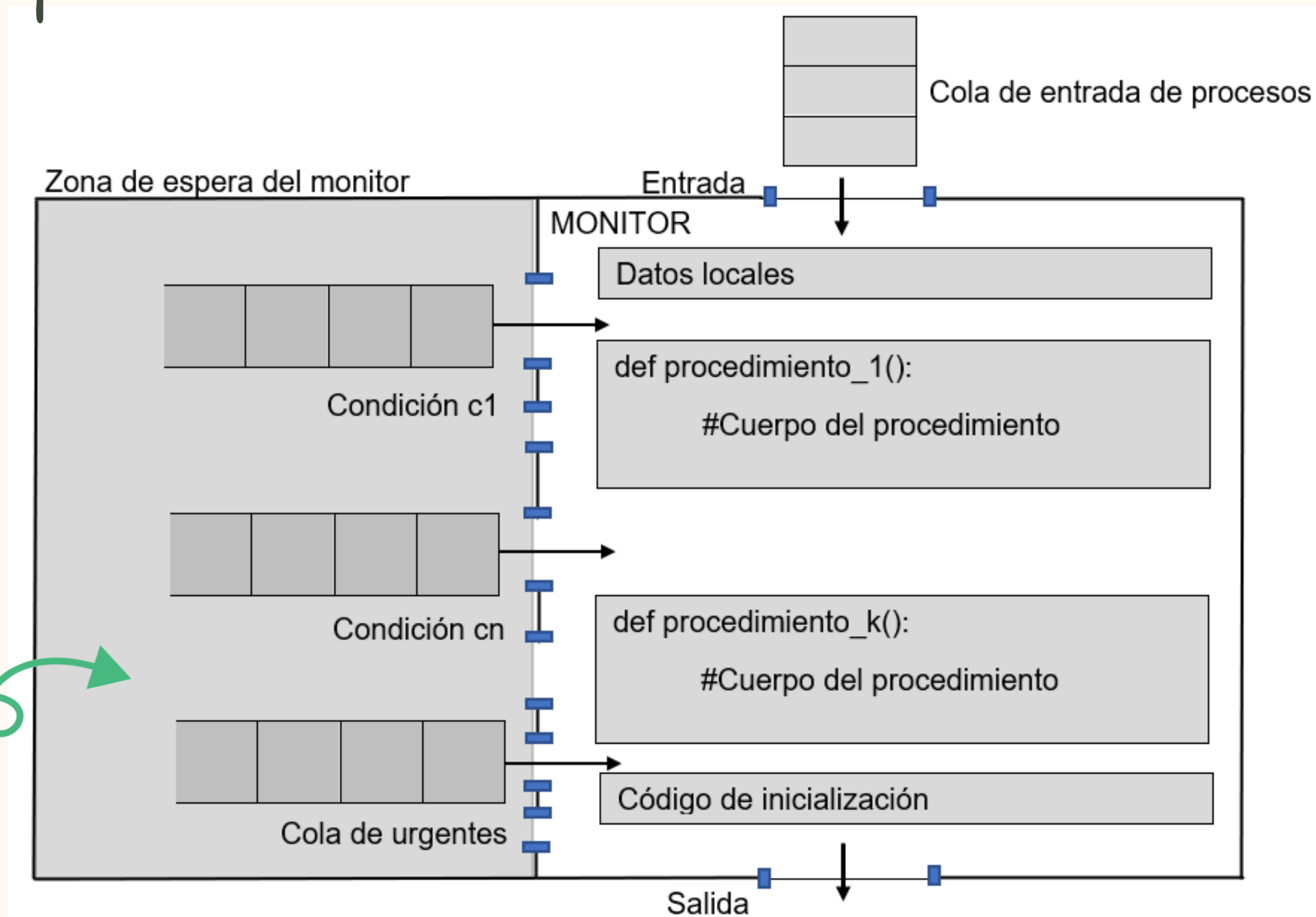
- Recursos locales.
- Código concentrado.
- Asegura la exclusión mutua.
- Asegura que un solo proceso este activo.

# Monitores

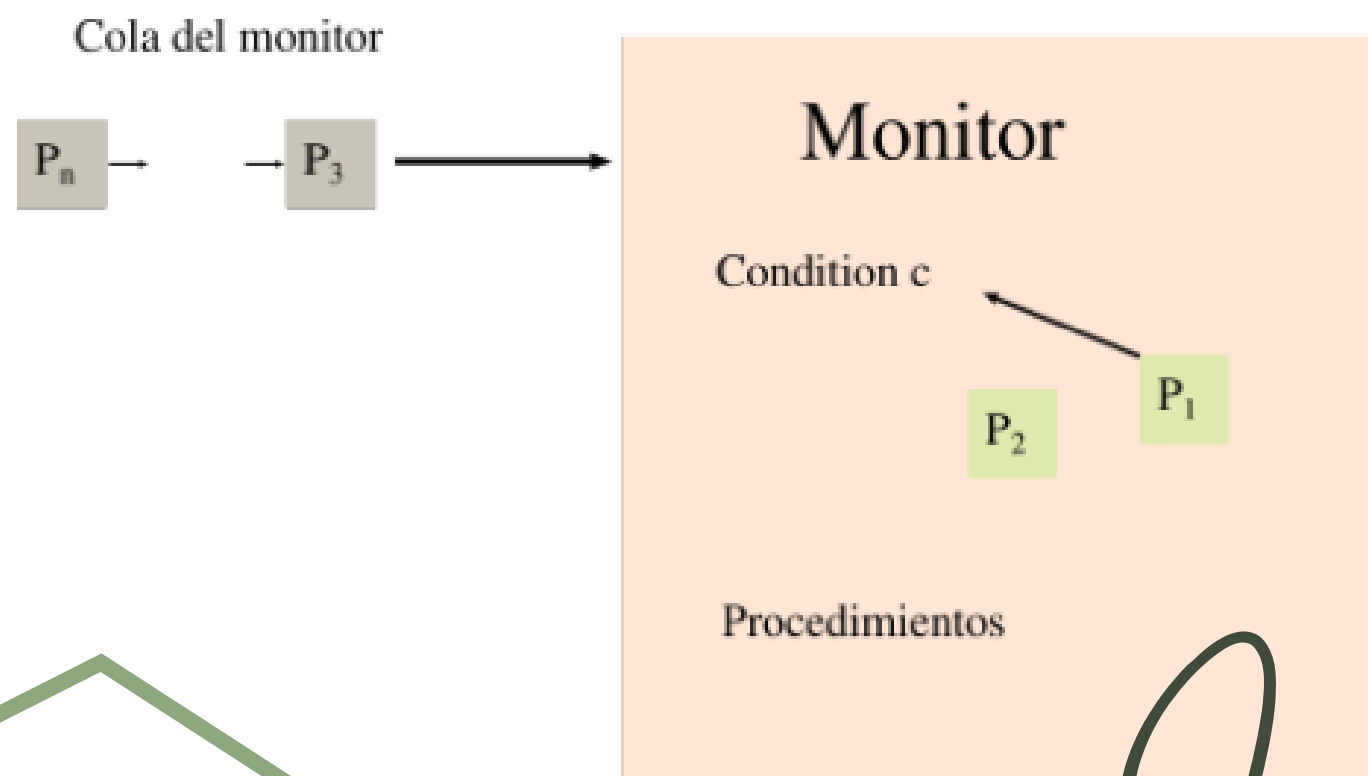
Módulo que encapsula a nuestra sección crítica.

Se construyen de manera en que la ejecución de los procesos dentro de el no se solapen, es decir, no accedan al mismo tiempo a la sección crítica, todo esto haciendo usos de operaciones atómicas.

## Estructura



Los monitores incluyen el método **wait()** el cual se encarga de suspender la ejecución del proceso llamado bajo la condición y **signal()** reanuda exactamente uno de los procesos suspendidos bajo la misma condición.



*En python...*

Para la implementación de monitores se hace uso de variables de condición, las cuales representan una especie de cola FIFO haciendo que los procesos esperen hasta que la condición correspondiente se satisfaga.

- **wait()** -- pone en espera a un proceso
- **start()** -- activa un proceso
- **notify()** -- reanuda a un solo proceso
- **notifyAll()** -- reanuda a todos los procesos

*Implementación*

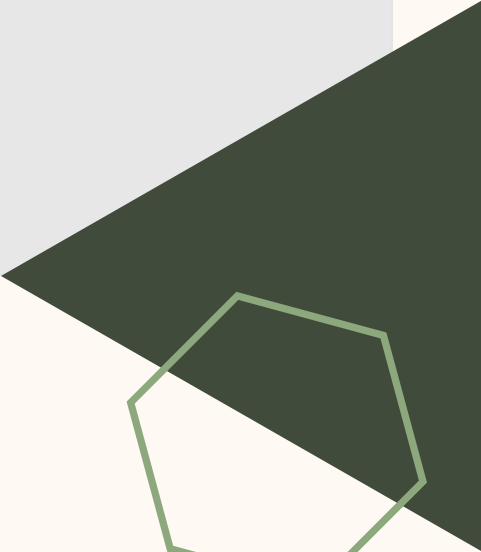


```
buffer_acotado buffer=new buffer_acotado();
```

```
Productor()  
{  
    char x;  
    while(forever)  
    {  
        producir(&x);  
        buffer.añadir(x);  
    }  
}
```

```
Consumidor()  
{  
    char x;  
    while(forever)  
    {  
        buffer.tomar(&x);  
        consumir(x);  
    }  
}
```

```
main()  
{  
    cobegin {  
        Productor();  
        Consumidor();  
    }  
}
```





```
monitor buffer_acotado
```

```
{
```

```
    char buffer[TAM_BUFFER];    //Espacio para N elementos
```

```
    int sigent, sigsal;          //Apuntadores al buffer
```

```
    int contador;               // Número de elementos del buffer
```

```
    condition no_lleno, no_vacio; //Para la sincronización
```

```
    añadir(char x) {
```

```
        if (contador == TAM_BUFFER) cwait(no_lleno);
```

```
                                //Buffer lleno; se impide producir
```

```
        buffer[sigent]=x;
```

```
        sigent = sigent + 1 % TAM_BUFFER;
```

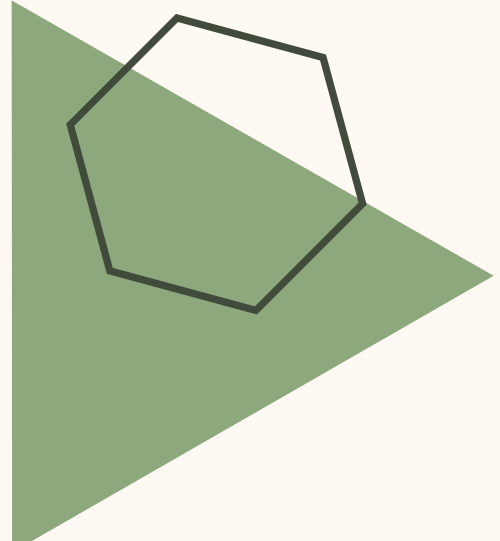
```
        contador++;             //Un elemento más en el buffer
```

```
        csignal(no_vacio);      //Reanudar un consumidor en espera
```

```
    }
```







```
    tomar(char x)
    {
        if (contador == 0) cwait(no_vacio);

                                //Buffer vacio; se impide consumir

        x=buffer[sigsal];
        sigsal=(sigsal+1) % TAM_BUFFER;
        contador--;

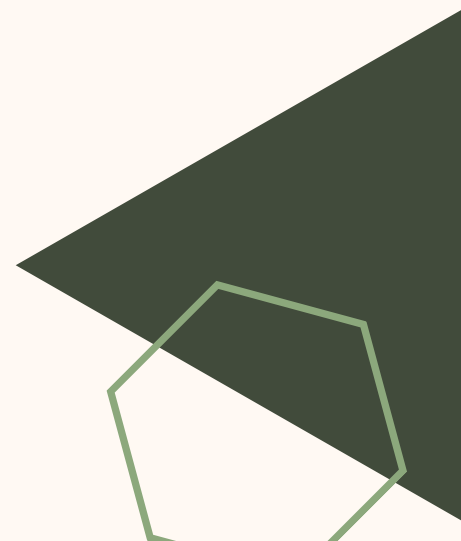
                                //Un elemento menos en el buffer

        csignal(no_lleno); //Reanudar un productor en espera
    }

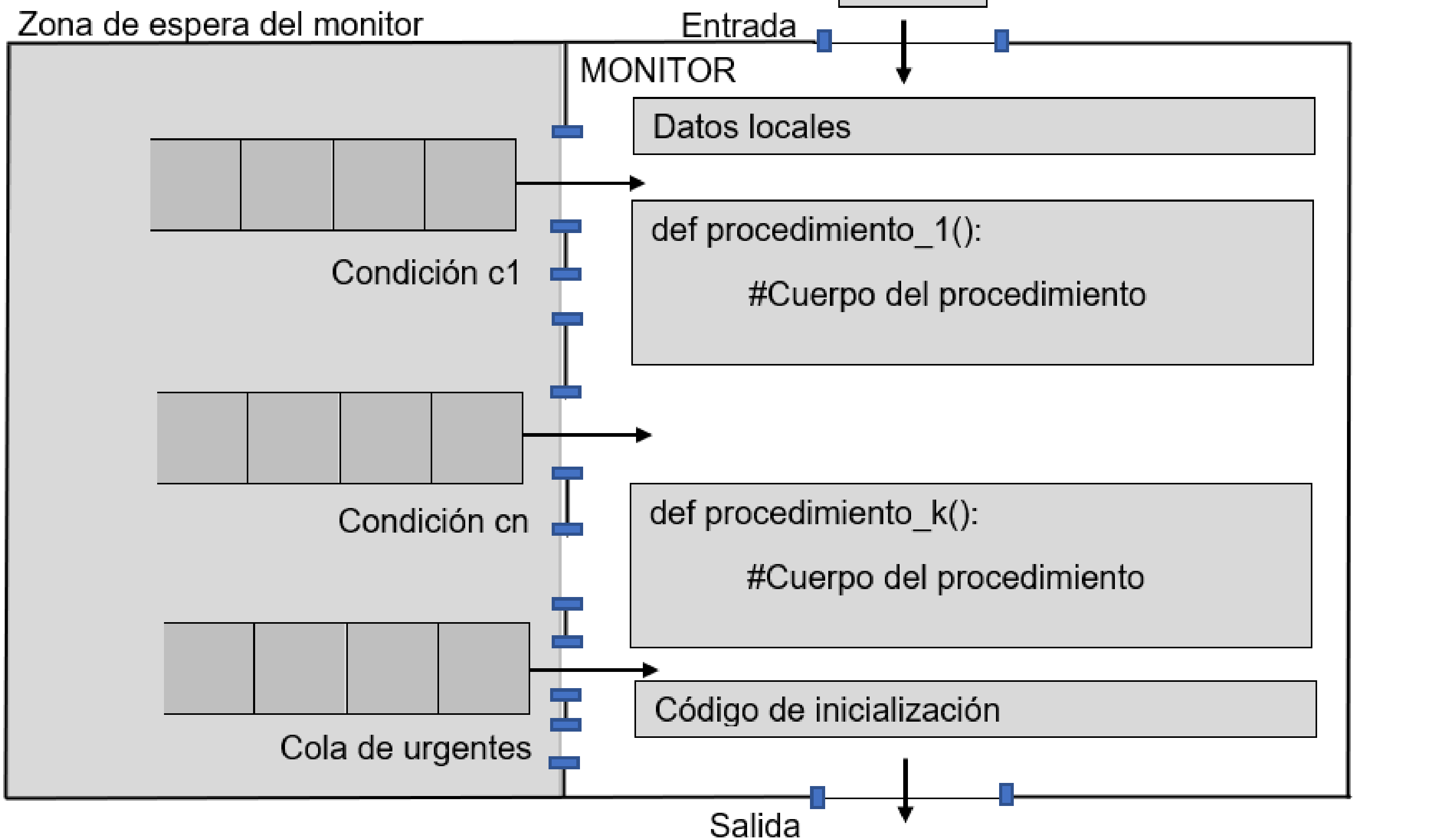
    initialize
    {
        //Cuerpo del monitor

        sigent=0;sigsal=0;contador=0; //Buffer inicia vacio
    }

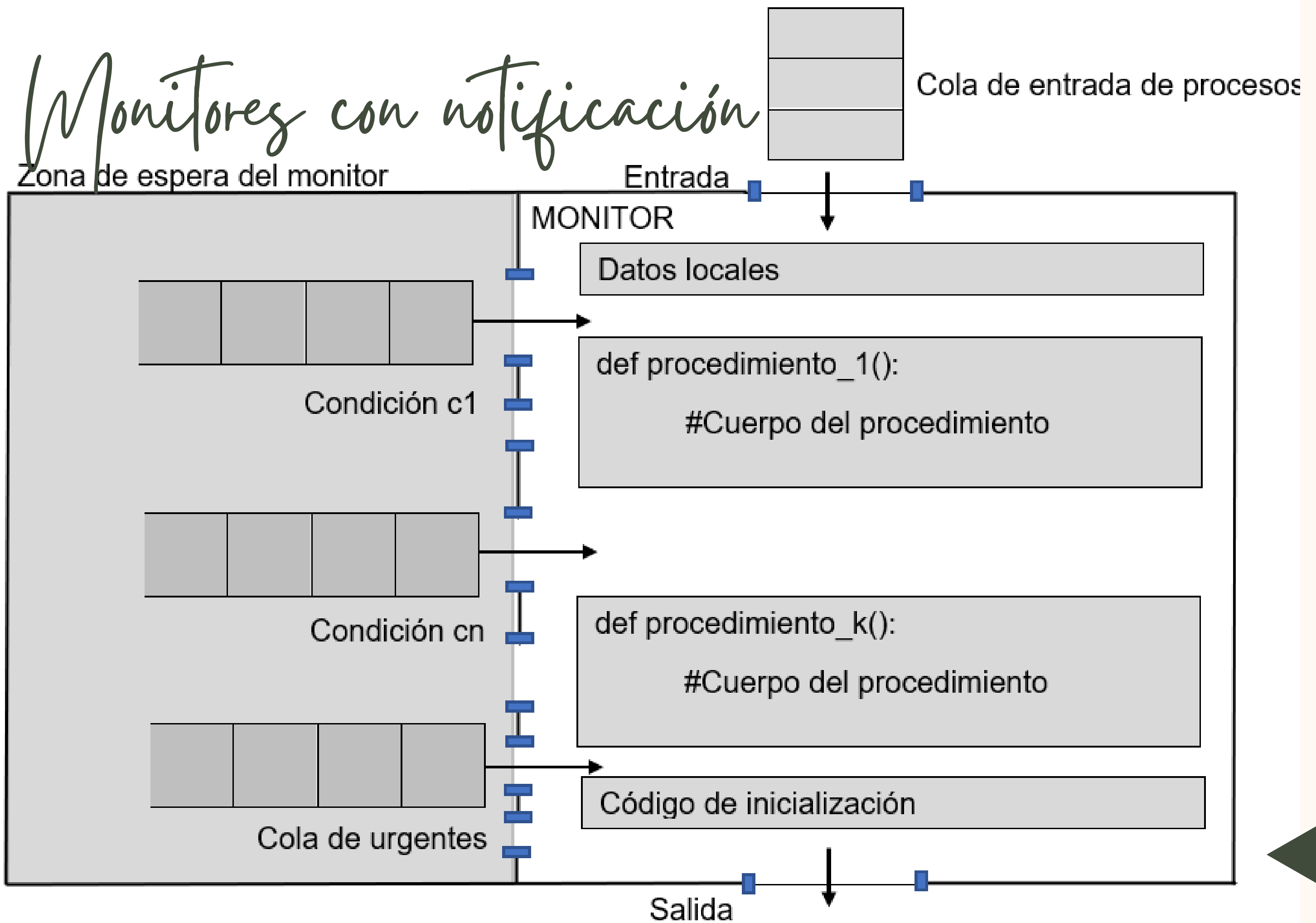
} // Termina el monitor
```



# Monitores con señales



# Monitores con notificación



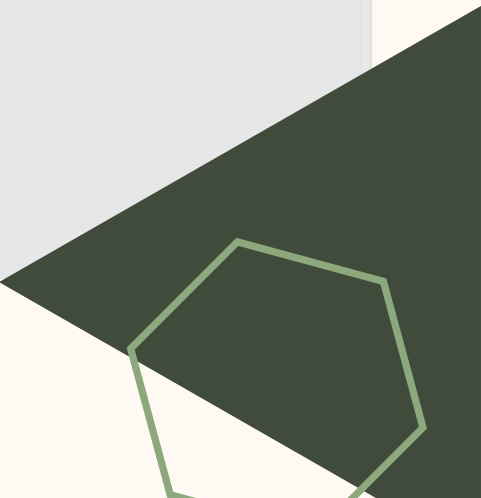


```
buffer_acotado buffer=new buffer_acotado();
```

```
Productor()  
{  
    char x;  
    while(forever)  
    {  
        producir(&x);  
        buffer.añadir(x);  
    }  
}
```

```
Consumidor()  
{  
    char x;  
    while(forever)  
    {  
        buffer.tomar(&x);  
        consumir(x);  
    }  
}
```

```
main()  
{  
    cobegin {  
        Productor();  
        Consumidor();  
    }  
}
```





```
monitor buffer_acotado
```

```
{
```

```
    char buffer[TAM_BUFFER];    //Espacio para N elementos
```

```
    int sigent, sigsal;          //Apuntadores al buffer
```

```
    int contador;               // Número de elementos del buffer
```

```
    condition no_lleno, no_vacio; //Para la sincronización
```

```
    añadir(char x) {
```

```
        while (contador == TAM_BUFFER) cwait(no_lleno);
```

```
        //Buffer lleno; se impide producir
```

```
        buffer[sigent]=x;
```

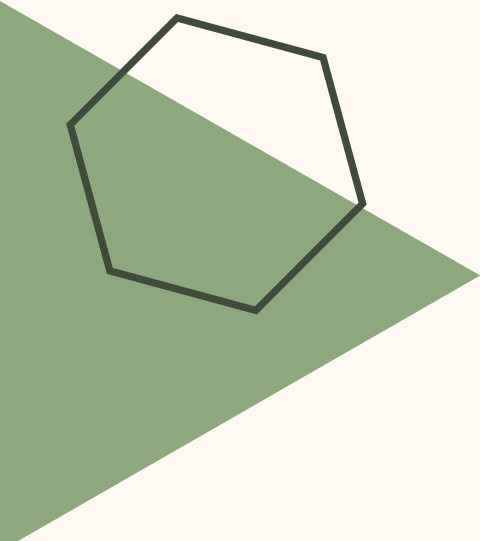
```
        sigent = sigent + 1 % TAM_BUFFER;
```

```
        contador++;             //Un elemento más en el buffer
```

```
        cnotify(no_vacio);      //Envía notificación a consumidor en espera
```

```
    }
```





```
tomar(char x)
{
    while (contador == 0) cwait(no_vacio);

    //Buffer vacio; se impide consumir

    x=buffer[sigsal];
    sigsal=(sigsal+1) % TAM_BUFFER;
    contador--;

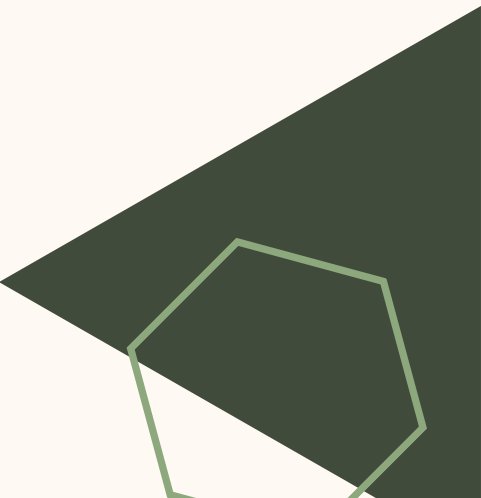
    //Un elemento menos en el buffer

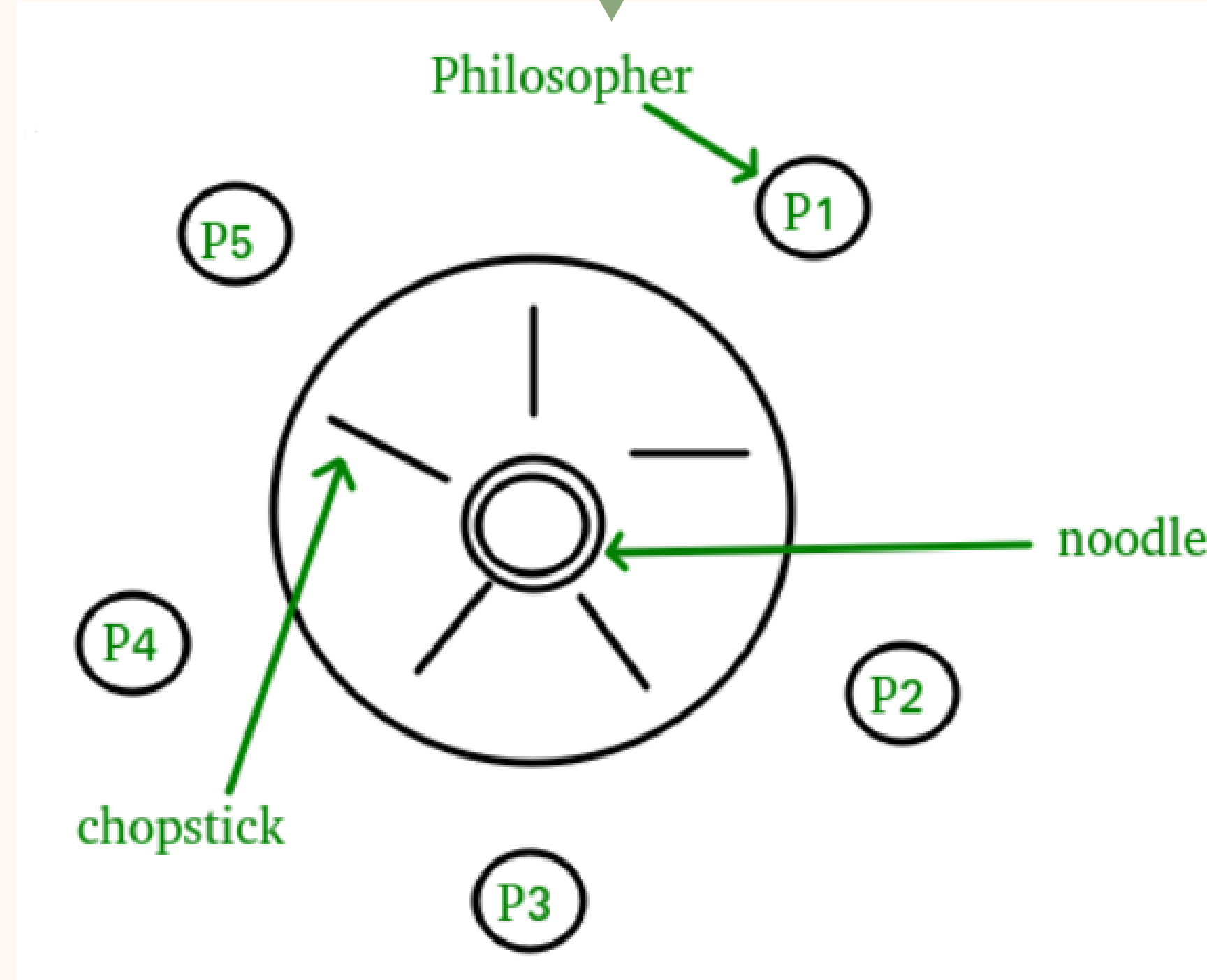
    cnotify(no_lleno); //Envía notificación a un productor en espera
}

initialize
{
    //Cuerpo del monitor

    sigent=0;sigsal=0;contador=0; //Buffer inicia vacio
}

} // Termina el monitor
```





La cena de los filósofos

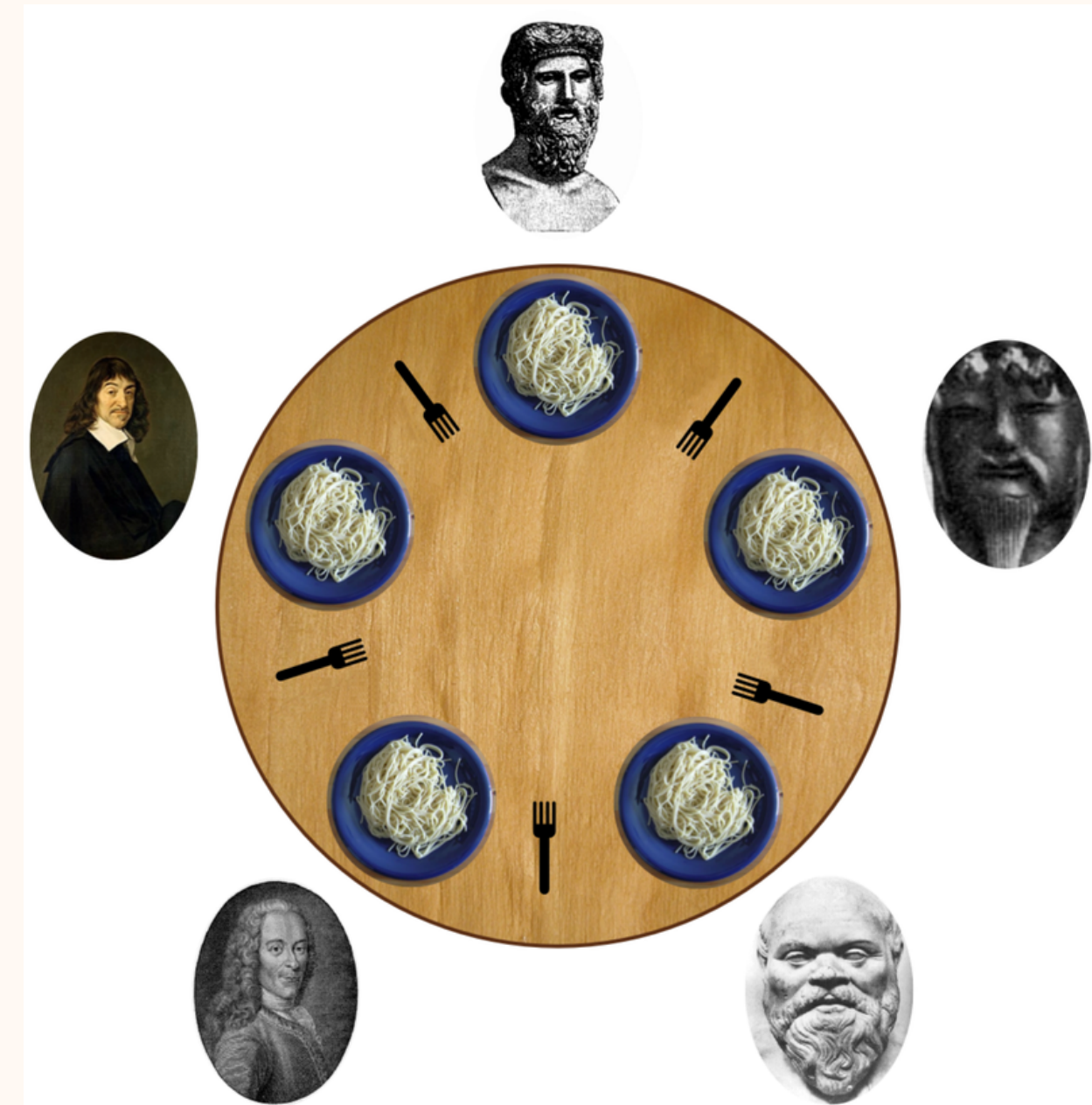
# Observaciones

Implementaremos una solución haciendo uso de monitores para controlar el acceso a las variables de estado y de condición. Solo indica cuándo entrar y salir del segmento.

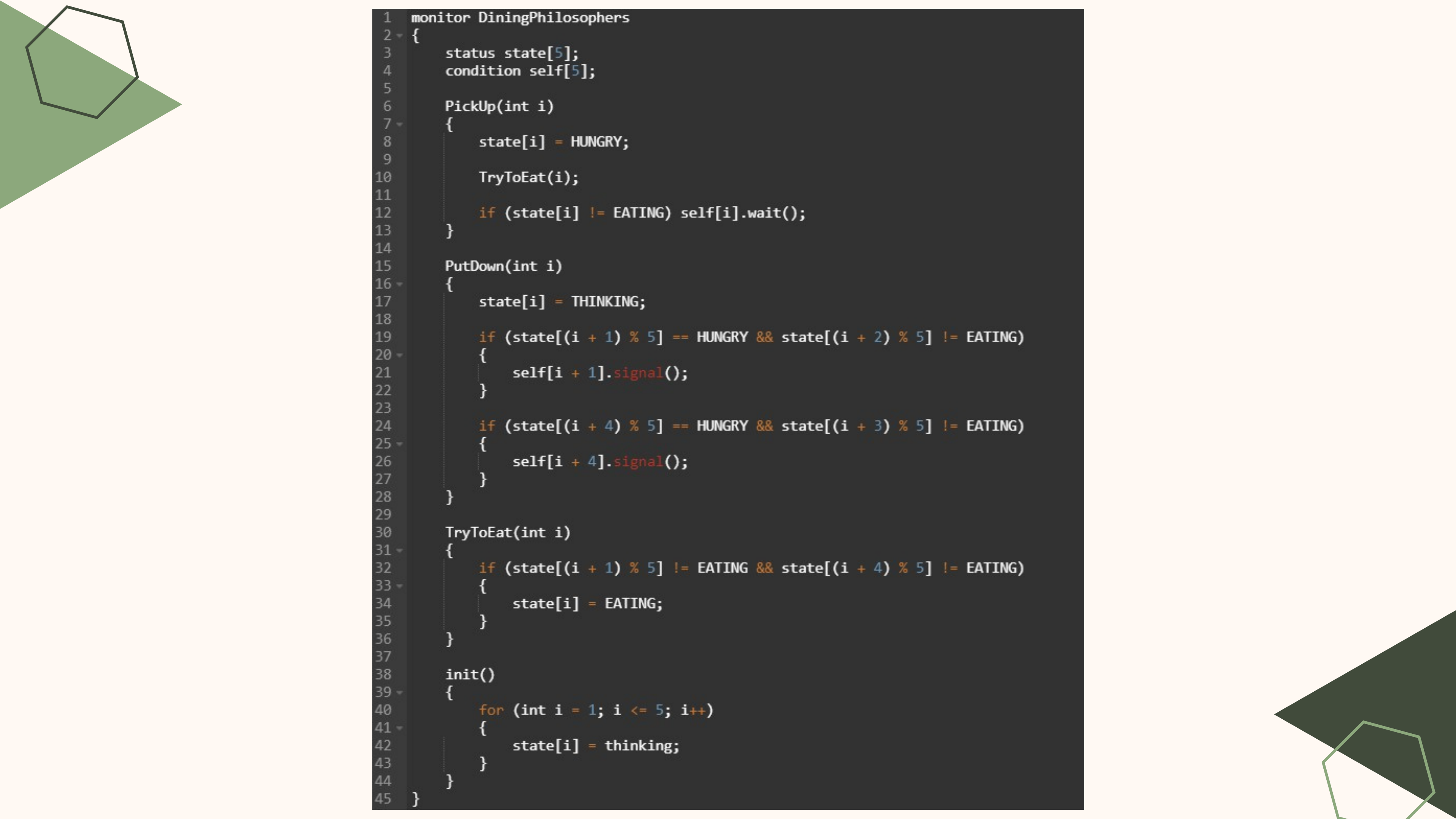
**PENSAMIENTO** - Cuando el filósofo no quiere acceder a ninguna de las bifurcaciones.

**HAMBRE** - Cuando el filósofo quiere entrar en la sección crítica.

**COMER** - Cuando el filósofo tiene las dos horquillas, es decir, ha entrado en la sección.







```
1  monitor DiningPhilosophers
2  {
3      status state[5];
4      condition self[5];
5
6      Pickup(int i)
7      {
8          state[i] = HUNGRY;
9
10         TryToEat(i);
11
12         if (state[i] != EATING) self[i].wait();
13     }
14
15     PutDown(int i)
16     {
17         state[i] = THINKING;
18
19         if (state[(i + 1) % 5] == HUNGRY && state[(i + 2) % 5] != EATING)
20         {
21             self[i + 1].signal();
22         }
23
24         if (state[(i + 4) % 5] == HUNGRY && state[(i + 3) % 5] != EATING)
25         {
26             self[i + 4].signal();
27         }
28     }
29
30     TryToEat(int i)
31     {
32         if (state[(i + 1) % 5] != EATING && state[(i + 4) % 5] != EATING)
33         {
34             state[i] = EATING;
35         }
36     }
37
38     init()
39     {
40         for (int i = 1; i <= 5; i++)
41         {
42             state[i] = thinking;
43         }
44     }
45 }
```