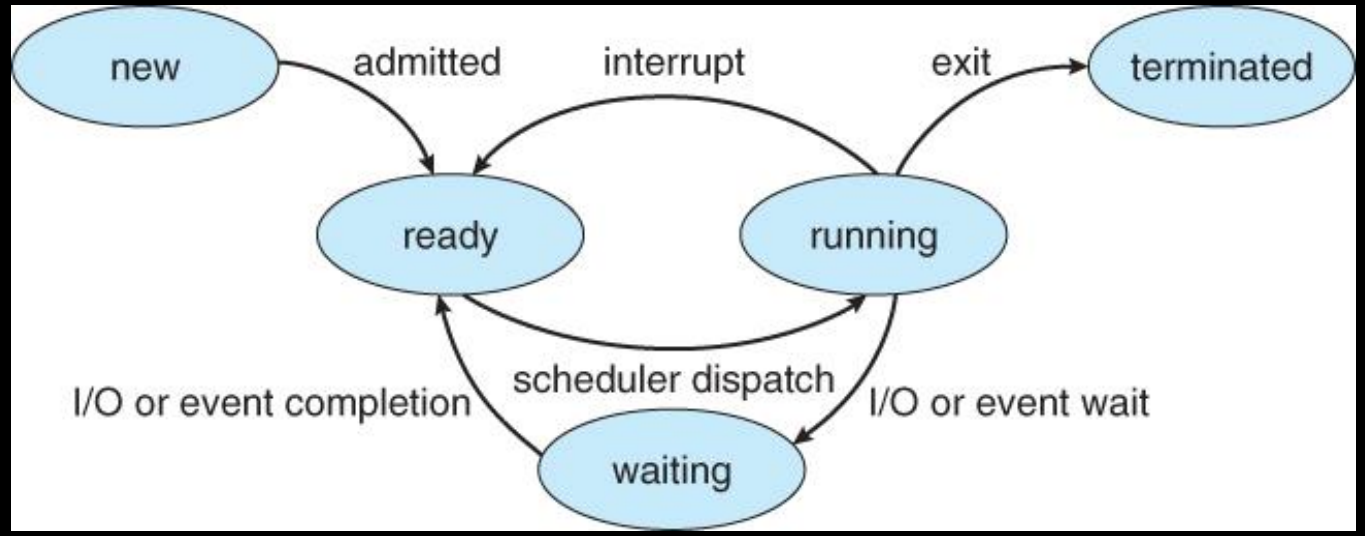
COMUNICACIÓN ENTRE PROCESOS

Scheduler, IPC, mecanismos de sincronización y semáforos.

LEONEL AGUILAR
SISTEMAS OPERATIVOS 1
CLASE 9

AGENDA

1. Scheduler
2. Hilos
3. Comunicación entre procesos
4. Técnicas de sincronización
5. Problema del Productor-Consumidor
6. Semáforos



SCHEDULER

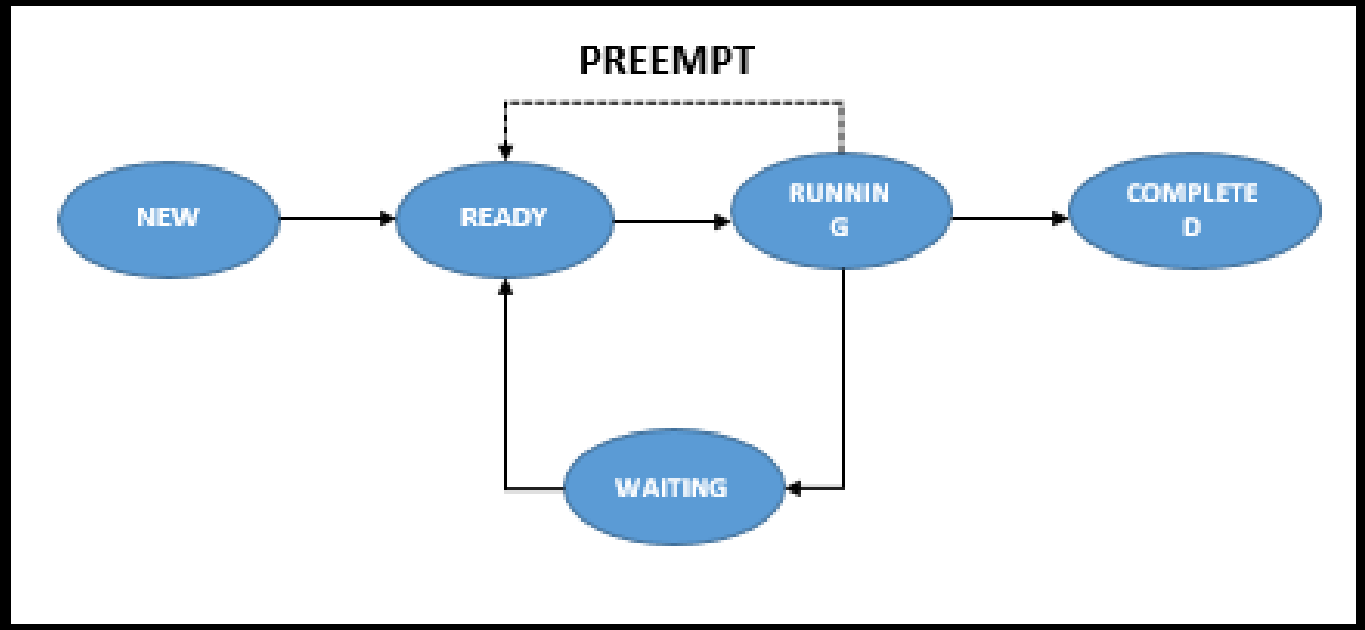
Add your first bullet point here

Add your second bullet point here

Add your third bullet point here

TIPOS DE SCHEDULERS

- First Come First Serve
- Shortest Job-First (SJF)
- Shortest Remaining Time
- Priority Scheduling
- Round Robin Scheduling - RRS
- Multilevel Queue Shcheduling

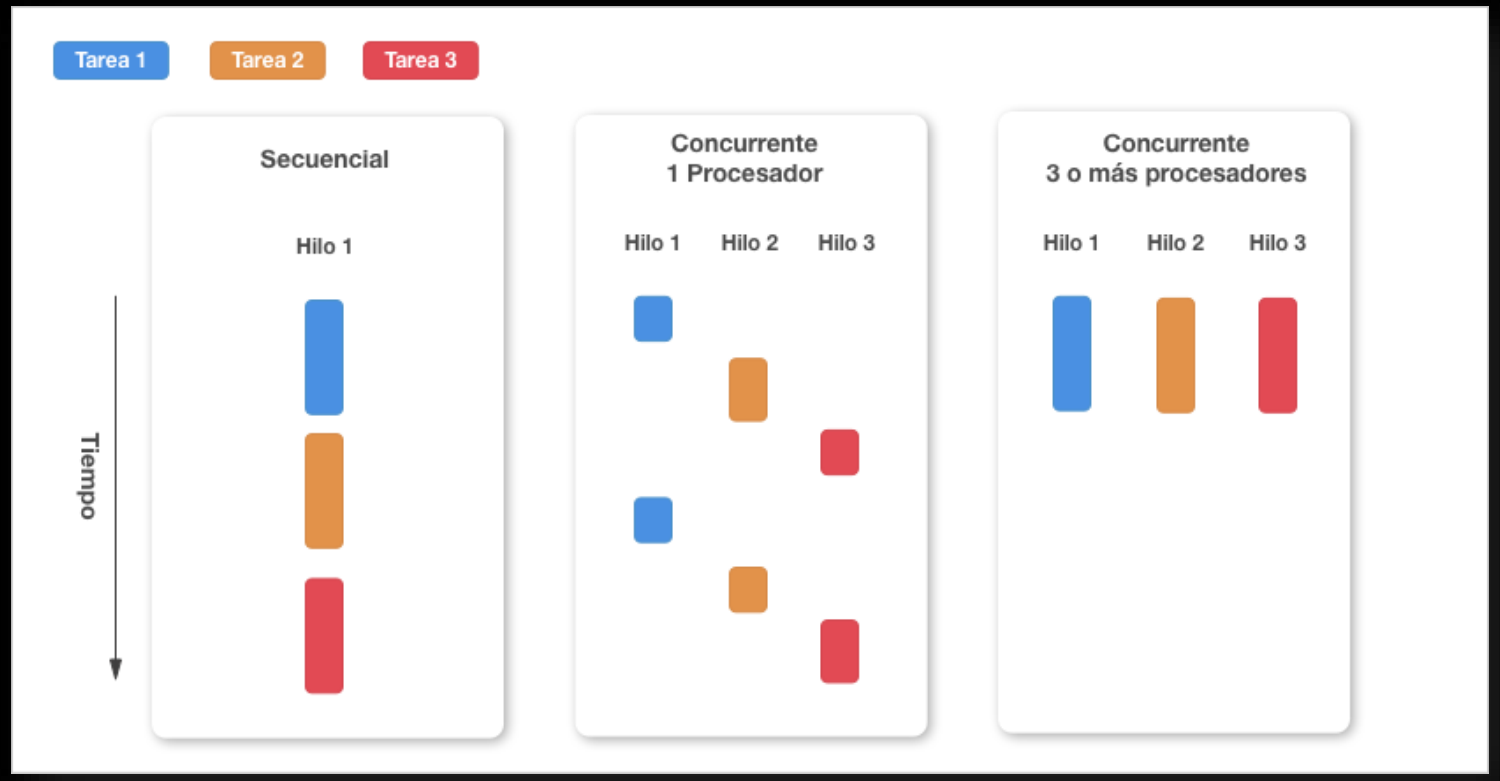


PREEMPTION

Interrumpir un proceso en ejecución y dejarlo en espera.

Esta interrupción es hecha por el Scheduler.

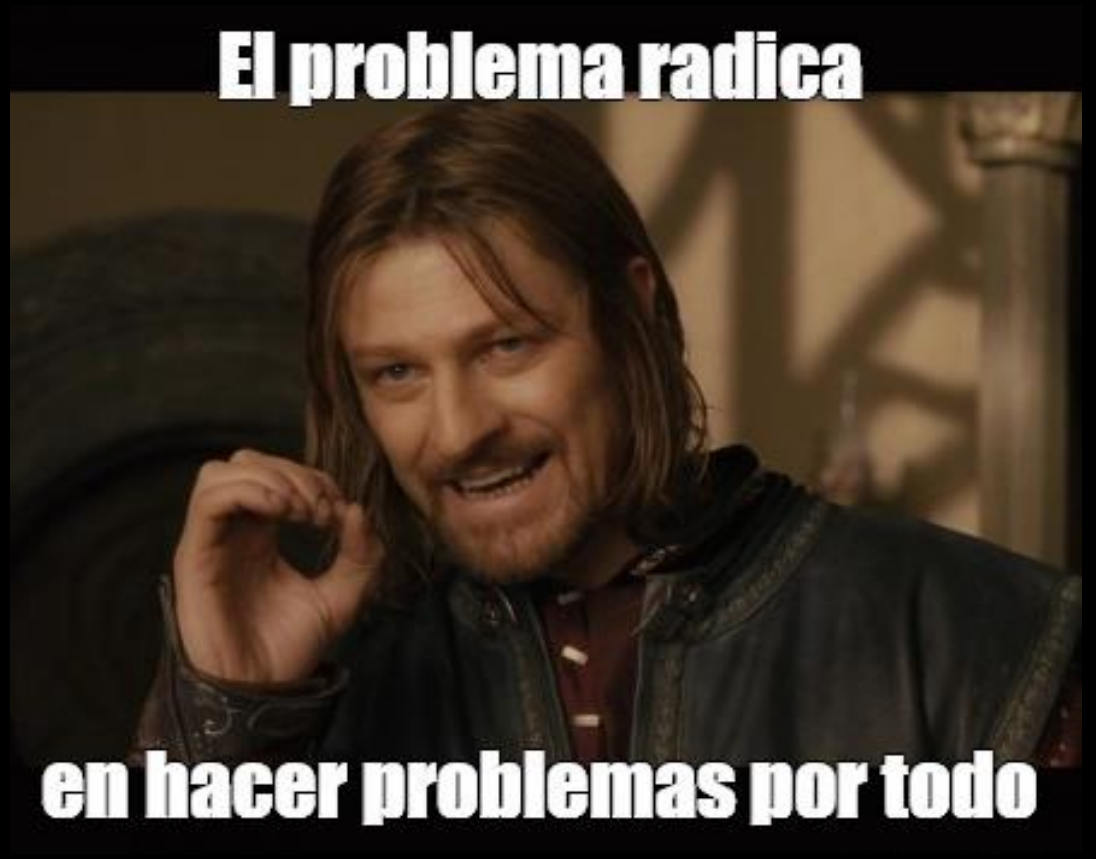
HILOS



- Cada proceso puede tener uno o más hilos.
- Cada proceso tiene un hilo llamado “main thread”.
- La concurrencia son dos procesos o hilos ejecutándose en el mismo tiempo.
- El paralelismo se logra con dos o más núcleos.

¿Qué problema hay entonces?

- Necesitamos compartir recursos.
- Necesitamos que los procesos compartan información y se comuniquen.



CHURRASCO

- Imaginemos que haremos un churrasco
- Darle vuelta a la carne
- Quitar la carne

```
class Churrascon {

    // Al inicio del churrasco, teníamos 0 pedazos cocinados.
    int pedazos_que_quedan = 0;

    int asar (){
        int id_del_pedazo = churrasquera.ponerUnPedazo(new PedazoDeCarnita());
        pedazos_que_quedan = pedazos_que_quedan + 1;

        // Retorno el id para saber cuál estoy volteando
        return id_del_pedazo;
    }

    void quitar(){
        churrasquera.quitarUnPedazo();
        pedazos_que_quedan = pedazos_que_quedan - 1;
    }

    void voltear(int id_del_pedazo){
        churrasquera.voltearPedazo(id_del_pedazo);
    }
```

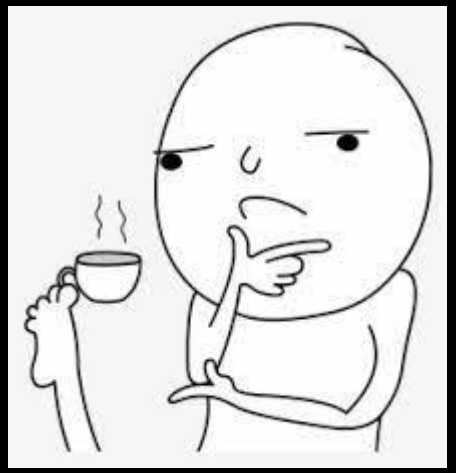
```
public static void main(String args[]){
    Churrascon churrascon = new Churrascon();

    // Este hilo servirá para estar cocinando pedazos de carne
    Thread cocinar_pedazos = new Thread(){
        @Override
        void Run(){
            int id_del_pedazo = churrascon.azar();
            // me tardo 8 minutos por lado
            Thread.sleep(480000);
            churrascon.voltear(id_del_pedazo);
            // me tardo 8 minutos por lado
            Thread.sleep(480000);
        }
    };

    // Este hilo servirá para estar quitando los pedazos de carne
    Thread quitar_pedazos_cocinados = new Thread(){
        @Override
        void Run(){
            // Ya que me tardo 8 minutos por lado, me tardaré 16 cocinando
            Thread.sleep(960000);
            // Quito el primero de la cola
            churrascon.quitar();
        }
    };

    cocinar_pedazos.start();
    quitar_pedazos_cocinados();
}
```

EL PROBLEMA…

- No tiene un comportamiento determinístico.
- Puede mostrar que queda un pedazo de carne, cero, ¡o menos uno!

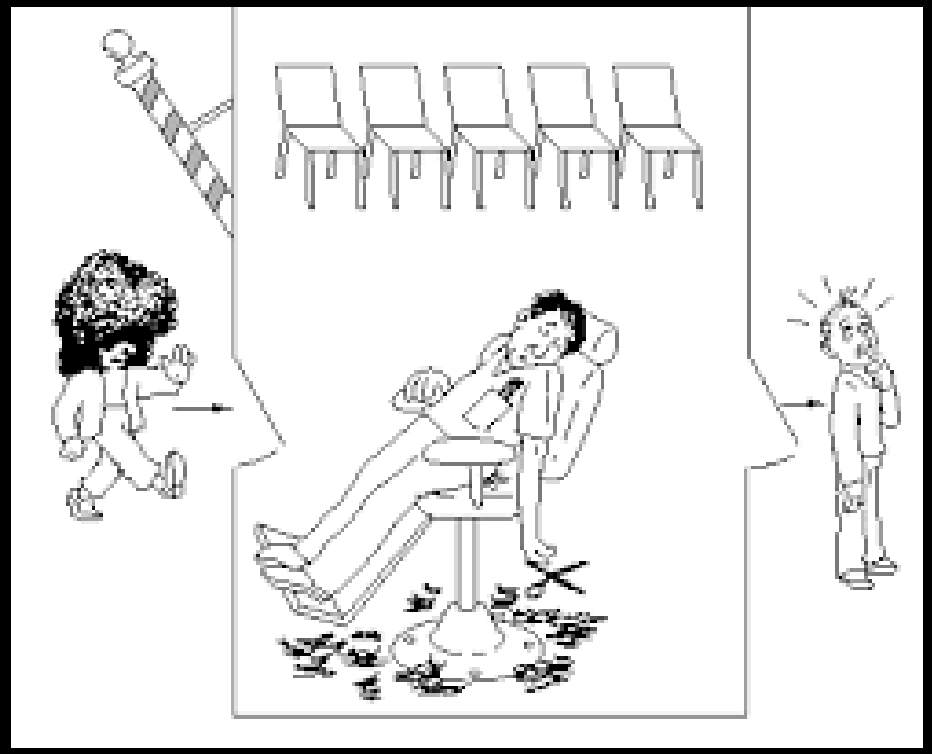
¿CÓMO SOLUCIONAMOS ESTO?

```
int asar (){
    int id_del_pedazo = churrasquera.ponerUnPedazo(new PedazoDeCarnita());
    pedazos_que_quedan = pedazos_que_quedan + 1;

    // Retorno el id para saber cuál estoy volteando
    return id_del_pedazo;
}

void quitar(){
    churrasquera.quitarUnPedazo();
    pedazos_que_quedan = pedazos_que_quedan - 1;
}
```



- Fue necesario desarrollar teoría.
- Se necesitaron 21 años para desarrollar una buena solución.
- Existieron múltiples problemas con variaciones.

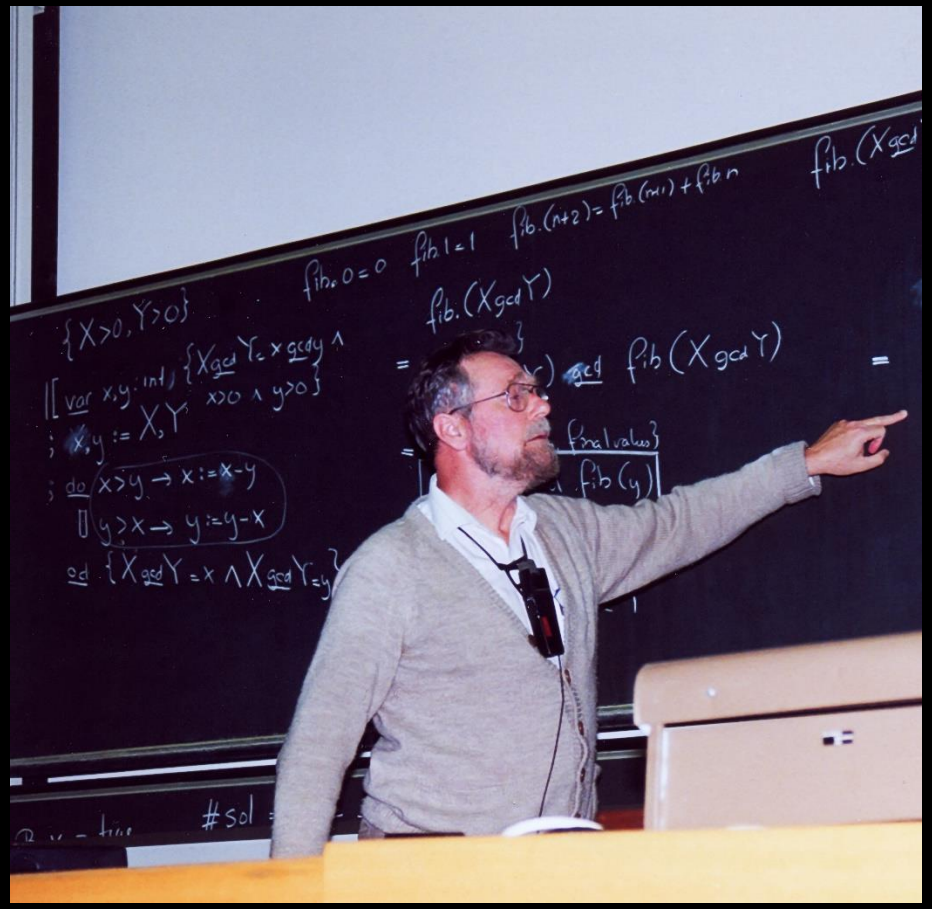
Otros problemas



- Filósofos comensales
- Barbero dormilón
- Productor-Consumidor
- Lectores escritores
- Fumadores

Filósofos Comensales

- Propuesto por Edsger Dijkstra en 1965.
- Demostró los problemas que existen al compartir recursos entre procesos.

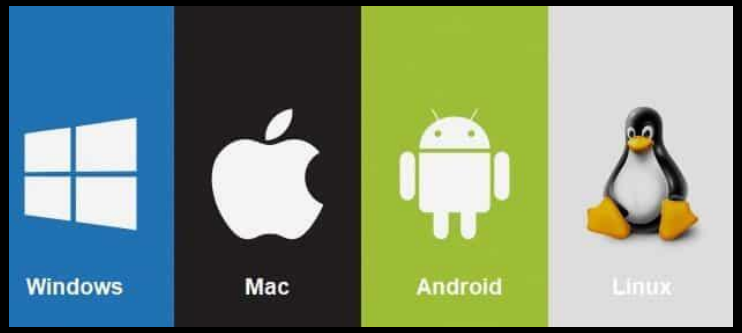


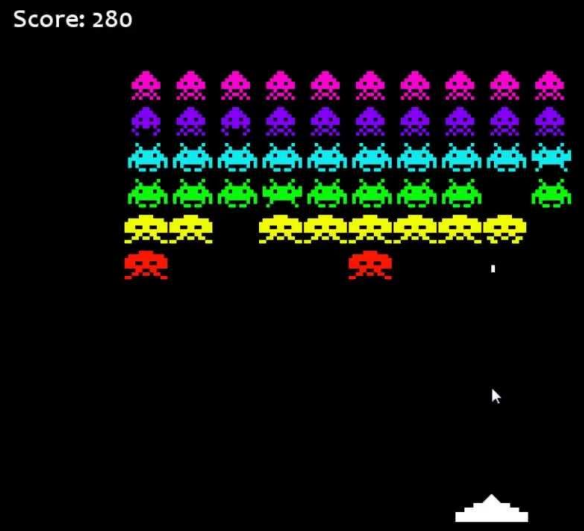
"Simplicity is prerequisite for reliability."

FILÓSOFOS COMENSALES

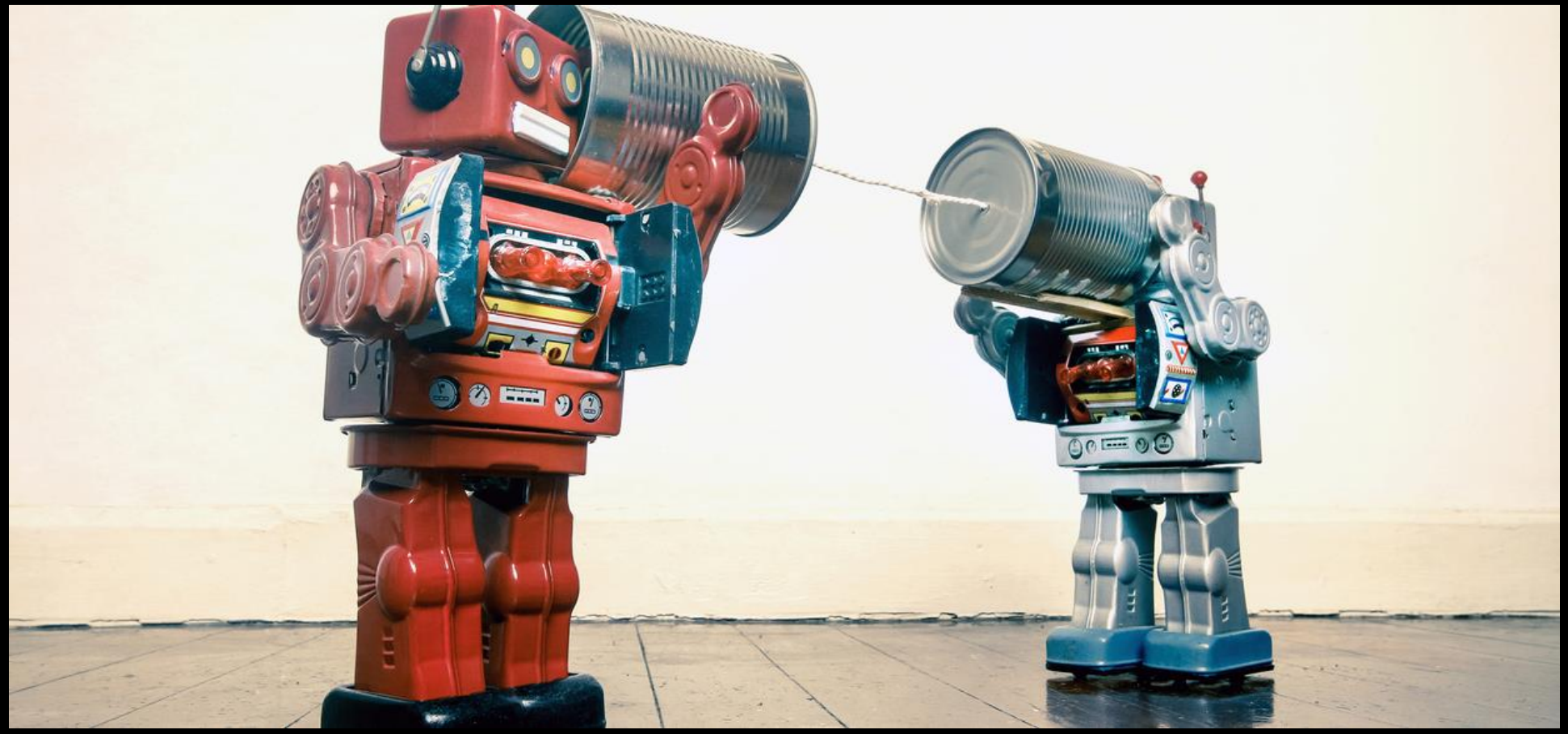
- Cada filósofo tiene un plato
- Necesitan dos palillos para comer.
- Hay 5 palillos.
- Evitar que se mueran de hambre.

¿DÓNDE MÁS LO PUEDO UTILIZAR?



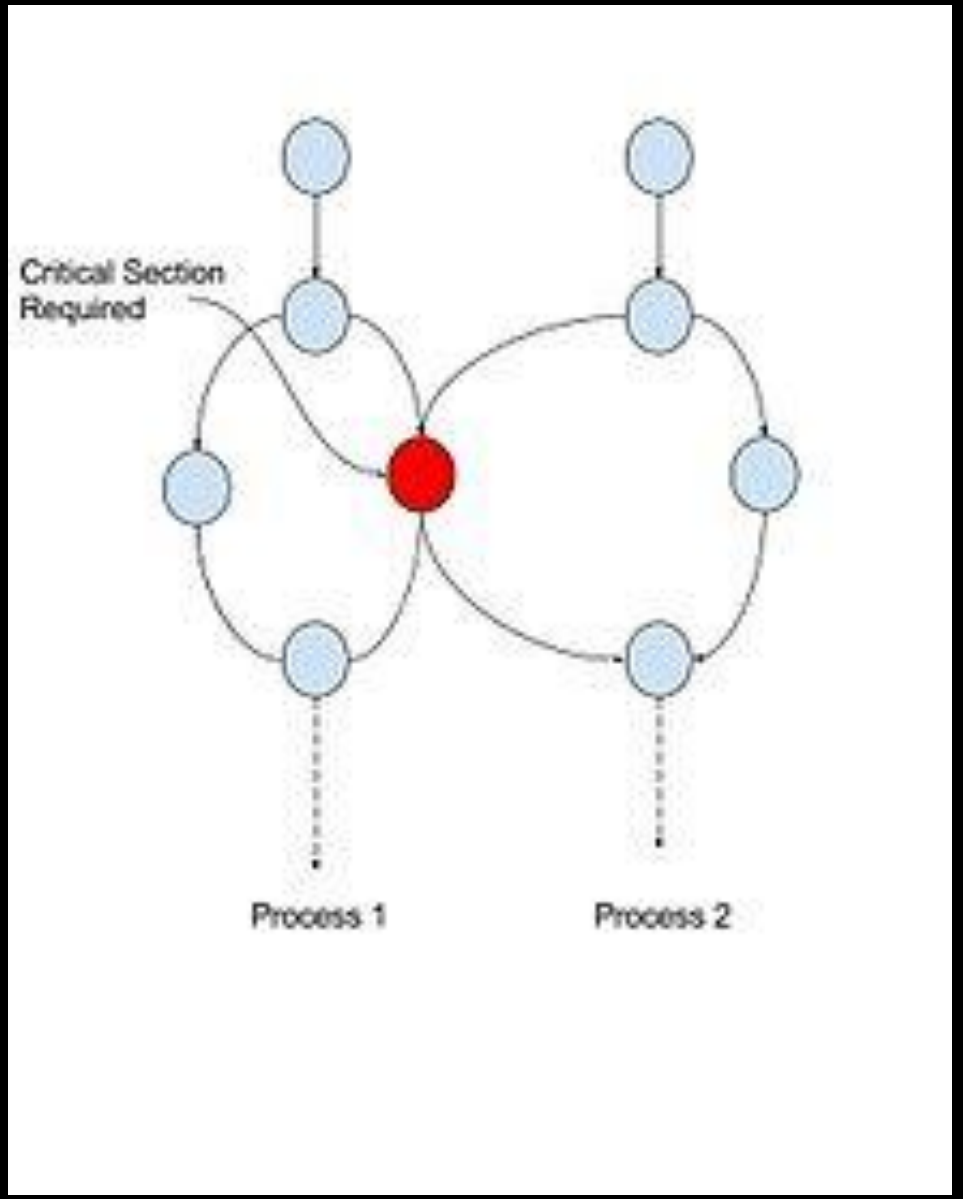


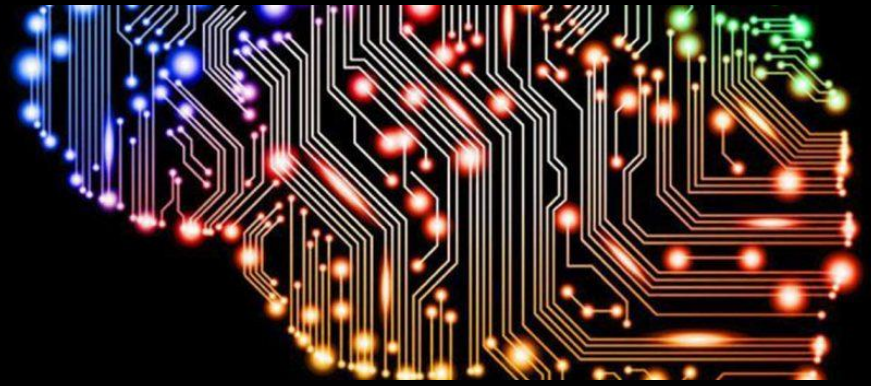
COMUNICACIÓN ENTRE PROCESOS

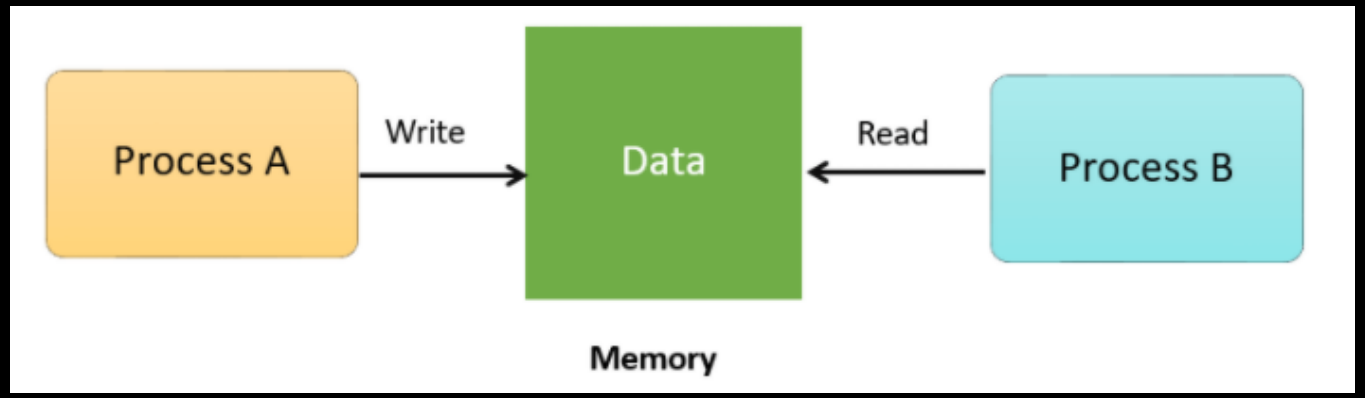


IPC

- Inter-process communication.
- Mecanismo para comunicación entre procesos.
- Necesidad de sincronizar.
- Dos procesos se comunican cuando acceden al mismo espacio de memoria.
- Estas instrucciones son denominadas Región Crítica.



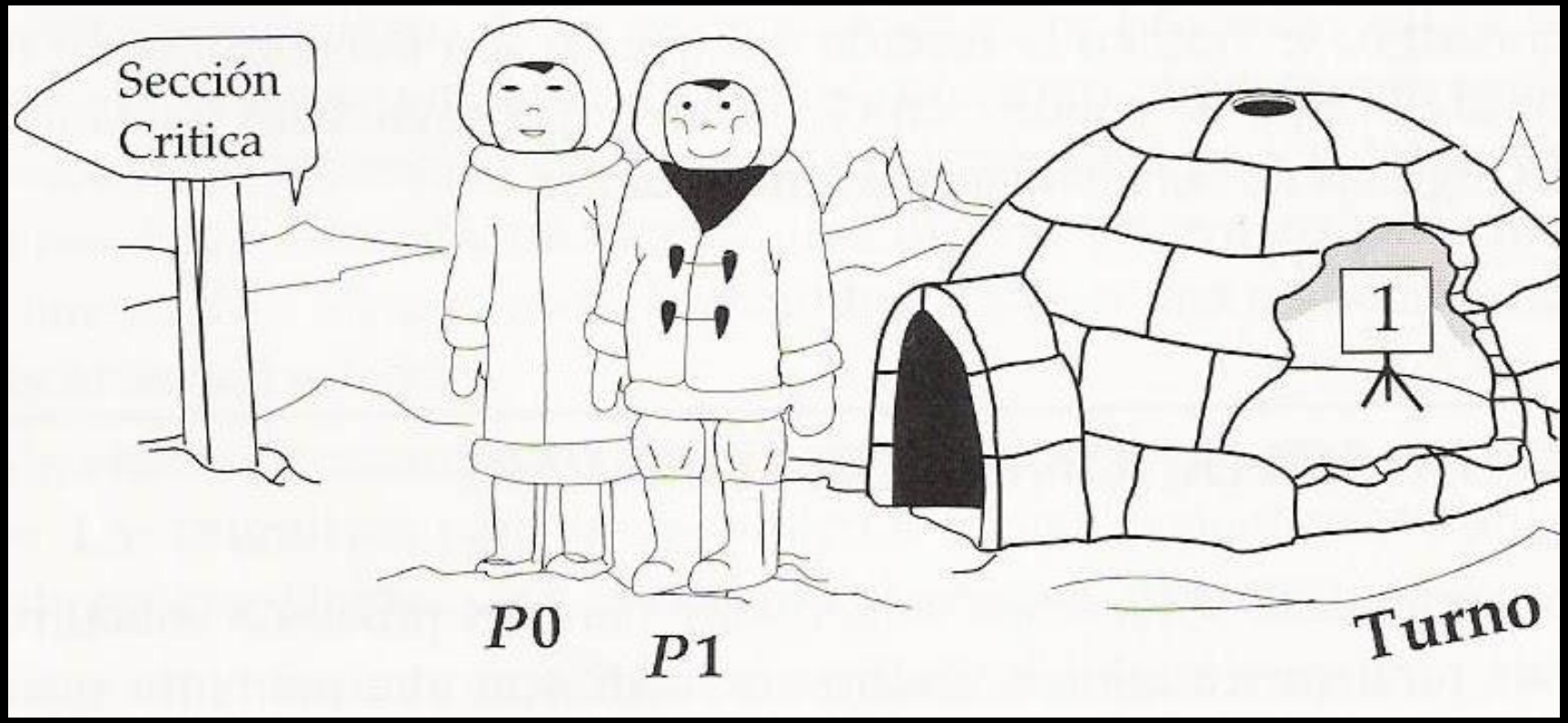




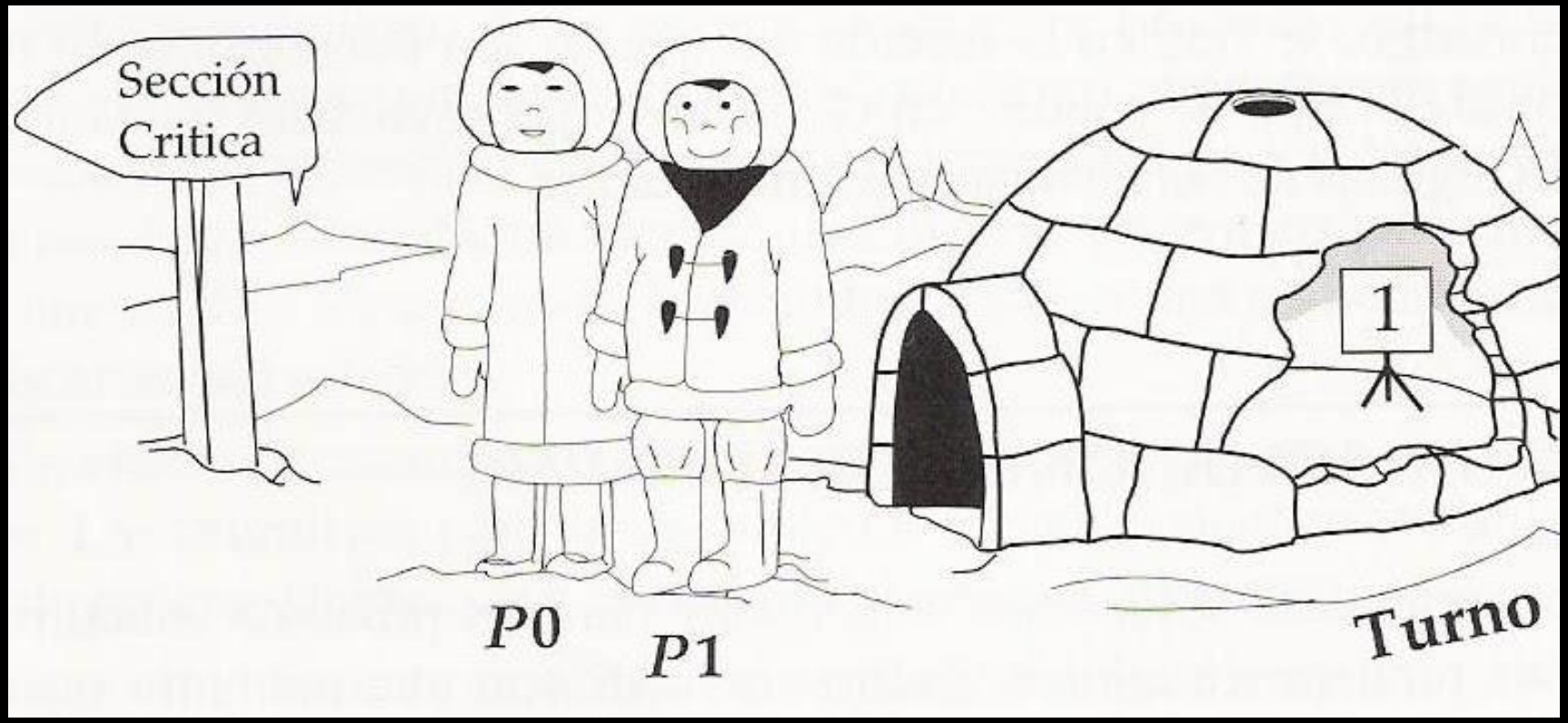
Región crítica

- Se accede a la memoria compartida.
- Garantizar **exclusión mutua**.

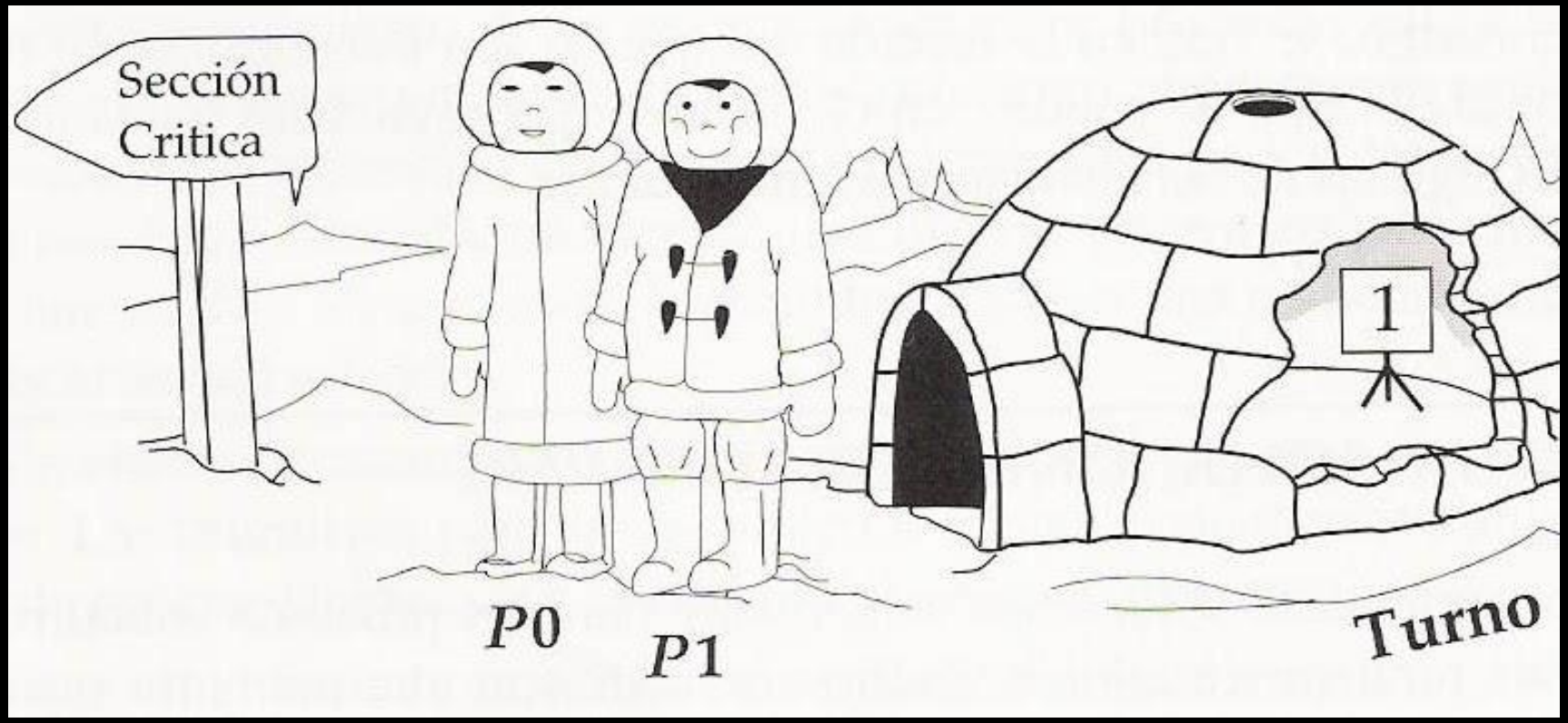
Debemos garantizar 4 cosas:
1. Exclusión Mutua



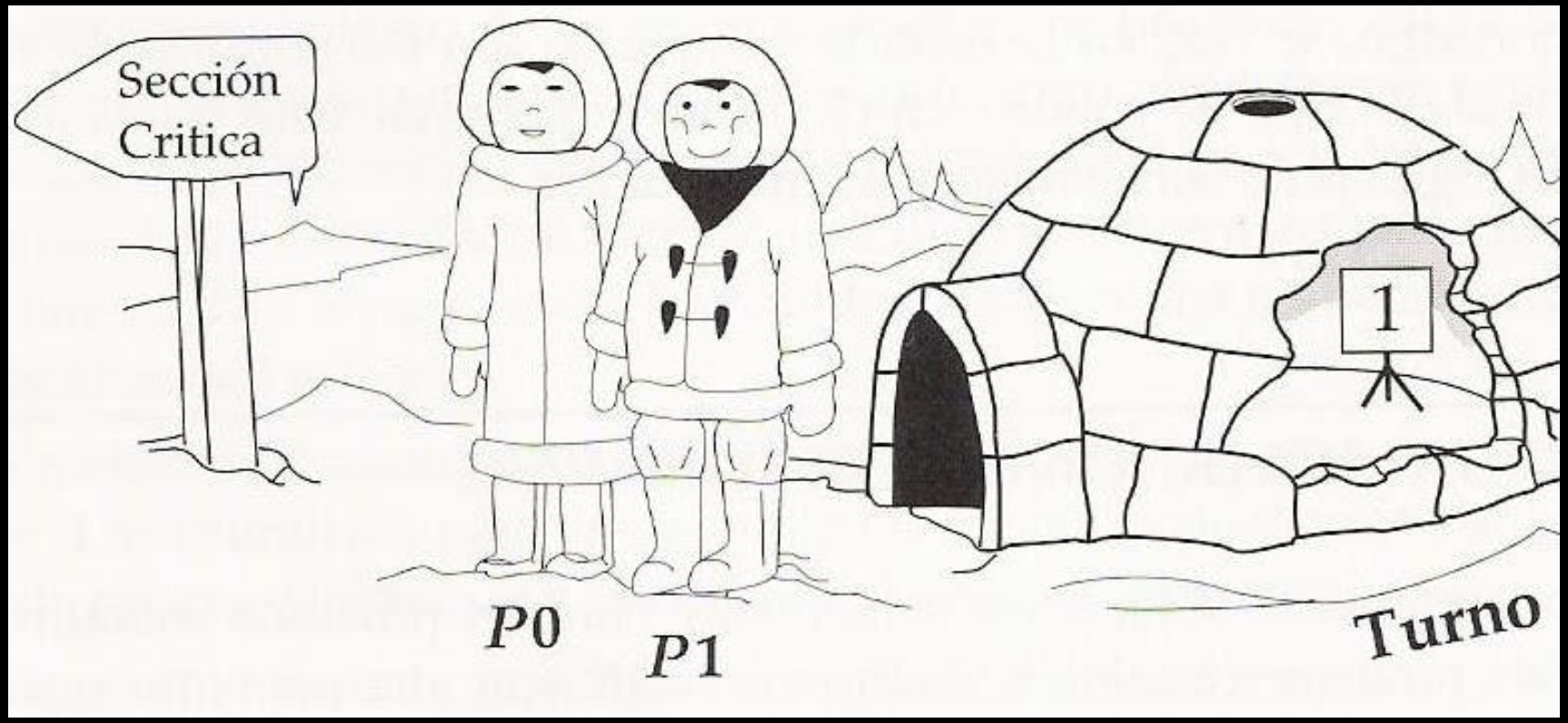
Debemos garantizar 4 cosas:
2. Progreso / Continuidad



Debemos garantizar 4 cosas:
3. No asumpciones



Debemos garantizar 4 cosas:
4. Espera limitada



TÉCNICAS DE SINCRONIZACIÓN

- Existen numerosas técnicas de sincronización entre procesos. Estos a continuación se denominan “busy-waiting techniques”.

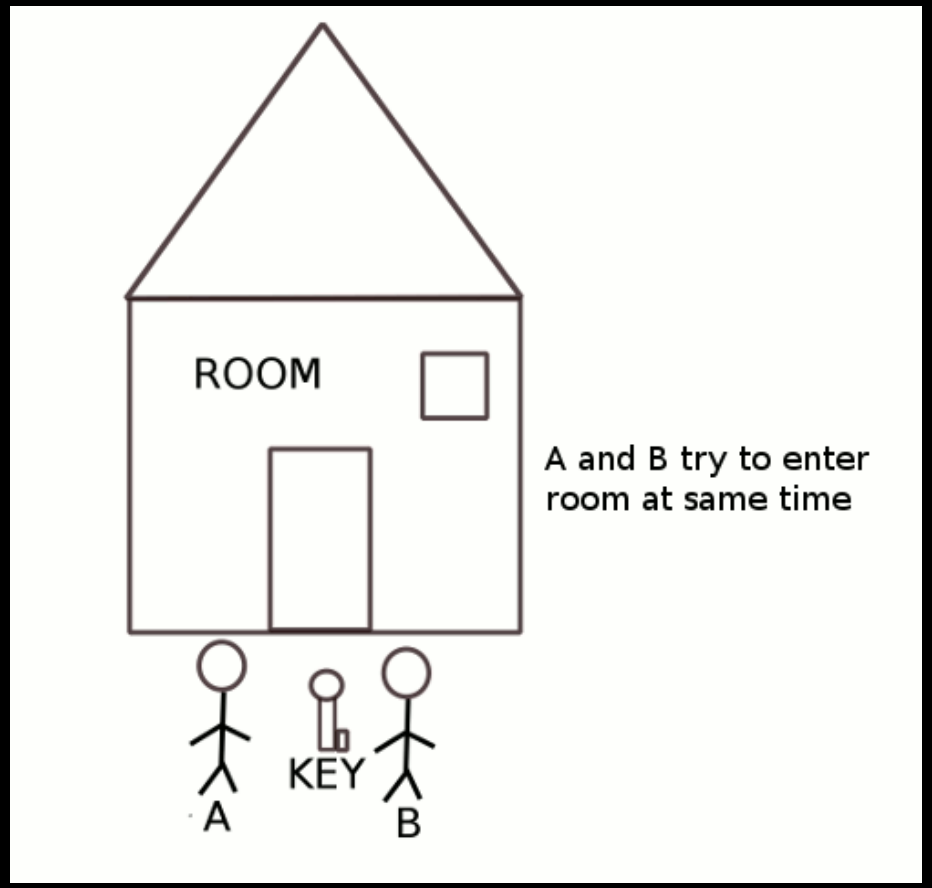
- Por interrupciones
- Por variables de lock
- Por alternancia estricta
- TSL
- Por variables interesadas
- Peterson

Deshabilitar Interrupciones

- Deshabilitar las interrupciones antes de ingresar a una región crítica.
- No funciona con varios núcleos.

LOCKS

- Imaginemos que entraremos a un baño público.
- Únicamente Podemos entrar si temenos la llave.
- Dejamos la llave cuando terminamos.



```
int candado = 0;
int contador = 0;
```

```
void contarPaArriba {
    /**********************
    * REGIÓN NO-CRÍTICA   *
    **********************/

    // ...

    /*********************************
    * MECANISMO DE SINCRONIZACIÓN   *
    *********************************/
    while (candado == 1) {
        ;
    }
    candado = 1;

    /**********************
    * REGIÓN CRÍTICA      *
    **********************/
    contador++;


    /**********************
    * REGIÓN NO-CRÍTICA   *
    **********************/
    candado = 0;

    // ...
}
```

```
void contarPaAbajo {
    /**********************
    * REGIÓN NO-CRÍTICA   *
    **********************/

    // ...

    /*********************************
    * MECANISMO DE SINCRONIZACIÓN   *
    *********************************/
    while (candado == 1) {
        ;
    }
    candado = 1;

    /**********************
    * REGIÓN CRÍTICA      *
    **********************/
    contador--;


    /**********************
    * REGIÓN NO-CRÍTICA   *
    **********************/
    candado = 0;

    // ...
}
```

Análisis

- No cumple con la exclusión mútua.
- Cumple con la continuidad.
- No hicimos asumpciones.
- No cumple con el criterio de límite de tiempo.

```
void contarPaArriba {
    while(true){

        while (candado == 1) {
            ;
        }
        candado = 1;

        /**********************
        * REGIÓN CRÍTICA      *
        **********************/
        contador++;

        /**********************
        * REGIÓN NO-CRÍTICA   *
        **********************/
        candado = 0;
        // ...
    }
}
```

```
void contarPaAbajo {
    /**********************
    * REGIÓN NO-CRÍTICA   *
    **********************/

    // ...

    /********************************
    * MECANISMO DE SINCRONIZACIÓN   *
    ********************************/
    while (candado == 1) {
        ;
    }
    candado = 1;

    /**********************
    * REGIÓN CRÍTICA      *
    **********************/
    contador--;


    /**********************
    * REGIÓN NO-CRÍTICA   *
    **********************/
    candado = 0;

    // ...
}
```

Alternancia estricta

1. Únicamente para dos procesos.
2. Definimos turnos.

```
turno = 0;
```

```
void proceso1 {

    while(1){
        while(turno == 1);

        /**********************
        * REGIÓN CRÍTICA      *
        **********************/

        turno = 1;
    }
}
```

```
void proceso2 {

    while(1){
        while(turno == 0);

        /**********************
        * REGIÓN CRÍTICA      *
        **********************/

        turno = 0;
    }
}
```

Análisis

- Cumple con exclusión mútua.
- No cumple con continuidad.
- Cumple con espera limitada.
- No hicimos asumpciones.

```
void proceso1 {

    while(1){
        while(turno == 1);

        /**********************
        * REGIÓN CRÍTICA      *
        **********************/

        turno = 1;
    }
}
```

```
void proceso2 {
    while(1){
        /**********************
        * REGIÓN NO-CRÍTICA   *
        **********************/

        while(turno == 0);

        /**********************
        * REGIÓN CRÍTICA      *
        **********************/

        turno = 0;
    }
}
```

Solución de Peterson

- Ideado en 1981 por Gary Peterson.
- Volvió obsoleto a los algoritmos de Dekker.
- Basado en llamadas, y tiene dos controladores.

```
int[] ejecutandose = { false, false };
int turno;
```

```
void entrar(int pid){
    int otro = 1 - pid;
    ejecutandose[pid] = true;
    turno = pid;

    while(ejecutandose[otro] == true && turn == pid) {
        //esperar a que el otro proceso termine su turno...
        ;
    }
}

void salir(int pid){
    ejecutandose[pid] = false;
}
```

```
void process1 {
    entrar(1);

    /**********************
    * REGIÓN CRÍTICA      *
    **********************/

    salir(1);
}

void process2 {
    entrar(0);

    /**********************
    * REGIÓN CRÍTICA      *
    **********************/

    salir(0);
}
```

Análisis

- Cumple con exclusión mútua.
- Cumple con continuidad.
- Cumple con espera limitada.
- No se realizaron asumpciones.

Dormir y despertar

- La solución de Peterson require mucho tiempo de espera.
- Se utiliza otro concepto, Sleep y Wakeup.

SLEEP

- Llamada al sistema
- Cambia el estado de proceso a dormido

WAKEUP

- Llamada al sistema
- Cambia el estado de proceso a activo

PRODUCTOR-CONSUMIDOR

- Existe un productor y un consumidor.
- El productor pone productos en el almacén.
- El consumidor, obtiene los productos.

PROBLEMA Y SOLUCIÓN

- Tenemos el mismo problema de las **condiciones de carrera.**

ESTADO	PRODUCTOR	CONSUMIDOR
NO HAY PRODUCTOS		
HAY UN PRODUCTO Y SE ESTÁ SACANDO		
HAY 9 PRODUCTOS Y SE ESTÁ INGRESANDO		
ESTÁ LLENO		

```
int productosEnAlmacen = 0;
```

```
void productor() {
    while (true) {
        producto = crearUnProducto();

        // Nuestro búfer es de tamaño 10
        if (productosEnAlmacen == 10) {
            sleep();
        }

        guardarEnAlmacen(producto);
        productosEnAlmacen = productosEnAlmacen + 1;

        if (productosEnAlmacen == 1) {
            wakeup(consumidor);
        }
    }
}
```

```
void consumidor() {
    while (true) {

        if (productosEnAlmacen == 0) {
            sleep();
        }

        agarrarProducto();
        productosEnAlmacen = productosEnAlmacen - 1;

        if (productosEnAlmacen == 9) {
            wakeup(productor);
        }
    }
}
```

SEMÁFOROS

- Ideado por Dijkstra
- Un nuevo tipo de variable.
- Dos operaciones, down y up.

```
down( ){
    while(s == 0) {
        ; //esperar
    }
    s--;
}
```

```
up( ){
    s++;
}
```

```
Semaforo productosEnAlmacen = 0;
Semaforo espaciosLibres = 10;

void productor() {
    while (true) {
        producto = crearUnProducto();
        down(espaciosLibres);
            guardarEnAlmacen(producto);
        up(productosEnAlmacen);
    }
}

void consumidor() {
    while (true) {
        down(productosEnAlmacen);
        agarrarProducto();
        up(espaciosLibres);
    }
}
```

SOLUCIÓN CON SEMÁFOROS