



# SISTEMAS OPERATIVOS

3004610 - 1

**German Sánchez Torres, I.S., M.Sc., Ph.D.**

Profesor, Facultad de Ingeniería - Programa de Sistemas

Universidad del Magdalena, Santa Marta.

Phone: +57 (5) 4214079 Ext 1138 - 301-683 6593

Edificio Docente, Cub 3D401.

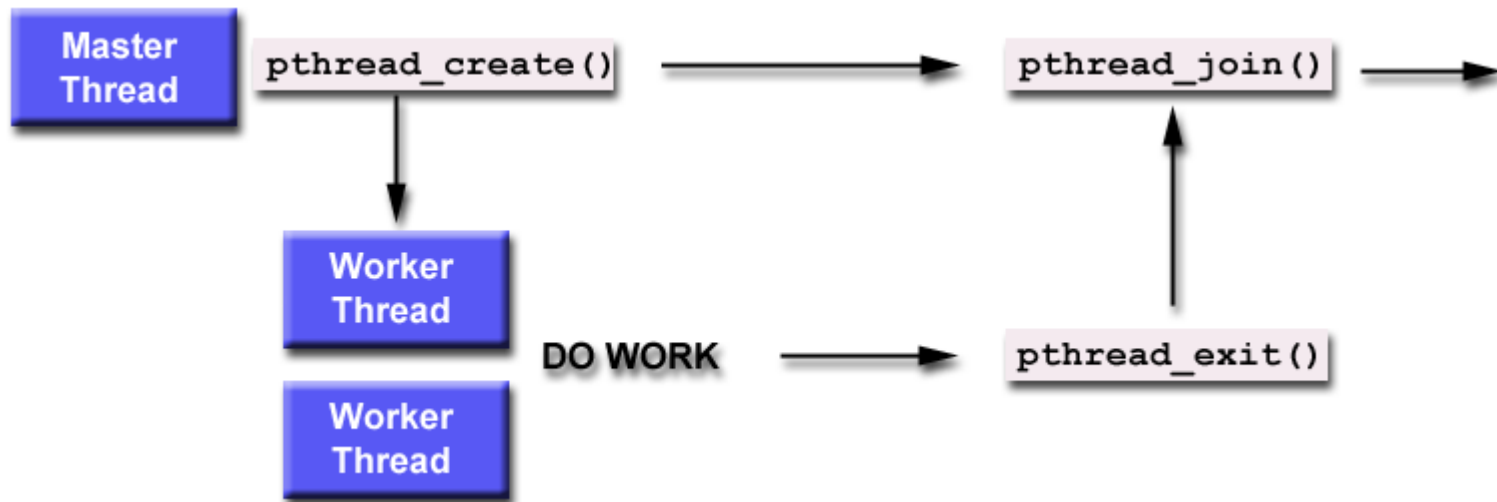
Email: [sanchez.gt@gmail.com](mailto:sanchez.gt@gmail.com) – [gsanchez@unimagdalena.edu.co](mailto:gsanchez@unimagdalena.edu.co)



# HILOS (Sincronización)

**SISTEMAS OPERATIVOS**

3004610 - 1



```
1  #include <stdio.h>
2  #include <pthread.h>
3
4  void *funcion_hilo(void * param);
5  void algo(int );
6  int sum;
7
8
9  main() {
10
11     int i;
12     pthread_t thread_id;
13
14     sum=0;
15     printf("Prevía creacion de hilos sum=%d\n",sum);
16
17     pthread_create( &thread_id, NULL, funcion_hilo, NULL);
18     for (i=0; i<10000; i++) {
19         algo(16);
20         sum ++;
21     }
22     pthread_join( thread_id, NULL);
23     printf("Hilo principal sum=%d\n", sum);
24     return 0;
25 }
26
```

```
27 void * funcion_hilo(void *param)
28 {
29     int i;
30     printf("Hilos %lu \n", pthread_self());
31
32     for (i=0; i<10000; i++) {
33         sum ++;
34         algo(16);
35     }
36     pthread_exit(0);
37 }
38
39 void algo(int n){
40     usleep(n);
41 }
```

```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 void *funcion_hilo(void * param);
5 void algo(int );
6 int sum;
7
8
9 main() {
10
11     int i;
12     pthread_t thread_id;
13
14     sum=0;
15     printf("Previo creacion de hilos sum=%d\n",sum);
16
17     pthread_create( &thread_id, NULL, funcion_hilo, NULL);
18     for (i=0; i<10000; i++) {
19         algo(16);
20         sum ++;
21     }
22     pthread_join( thread_id, NULL);
23     printf("Hilo principal sum=%d\n", sum);
24     return 0;
25 }
26
```

```
27 void * funcion_hilo(void *param)
28 {
29     int i;
30     printf("Hilos %lu \n", pthread_self());
31
32     for (i=0; i<10000; i++) {
33         sum ++;
34         algo(16);
35     }
36     pthread_exit(0);
37 }
38
39 void algo(int n){
40     usleep(n);
41 }
```

sum++  
sum = sum + 1

---

```
mov ax,[sum]
add ax,1
mov [sum], ax
```

### Hilo1

```
mov ax,[sum]
add ax,1
mov [sum], ax
```

### Hilo2

```
mov ax,[sum]
add ax,1
mov [sum], ax
```

Toda solución debe cumplir tres condiciones:

**Exclusión mutua:** Si el proceso  $P_i$  se está ejecutando en su sección crítica, ningún otro proceso puede estar ejecutándose en su sección crítica.

**Progreso:** Sin ningún proceso se está ejecutando en su sección crítica y hay procesos que desean ejecutar la sección crítica deben poder ejecutarla.

**Espera limitada:** Hay un límite para el número de veces que se permite a otros procesos ingresar en sus secciones críticas y antes de que se le otorgue la autorización para hacerlo.

## Problema de la Sección Crítica

Esquema general de solución:

```
do {  
    protocolo de entrada  
    sección crítica  
    protocolo de salida  
    resto de la sección  
} while (TRUE);
```

Pi y Pj comparten la variable **turno**

```
do{  
while (turno != i) nada();  
    sección crítica  
    turno = j;  
    resto de la sección  
}while(TRUE);
```



Variables compartidas:

listo[i] = FALSE;

listo[j] = FALSE;

```
do{  
    listo[i] = TRUE;  
    while (listo[j]) nada();  
    sección crítica  
    listo[i] = FALSE;  
    resto de la sección  
}while(TRUE);
```

Variables compartidas:

int turno, listo[2];

listo[i] = listo [j] = FALSE;

Turno = 1 o 0;

```
do{  
    listo[i] = TRUE;  
    turno = j;  
    while(listo[j] && (turno == j)) nada();  
    sección crítica  
    listo[i] = FALSE;  
    resto de la sección  
}while(TRUE);
```

Es un tipo especial de variable con dos únicas operaciones P y V

**P (semáforo):** Operación atómica que espera hasta que el semáforo sea positivo, en este momento lo decrementa en 1.

**V (semáforo):** Operación atómica que incrementa el semáforo en 1.

```
P(sem)  //probar
```

***Sección crítica***

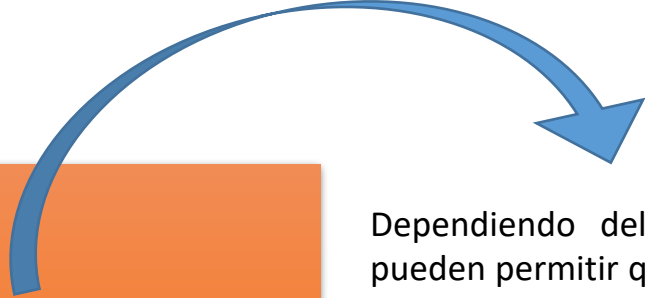
```
V(sem)  //incrementar
```

```
typedef struct{  
    int contador;  
    cola q;  
}SEMAFORO;
```

```
P (SEMAFORO *s) {  
    s->contador-=1;  
    if(s->contador < 0) {  
        Añadimos proc. a s->q;  
        Bloquear;  
    }  
}
```

```
V (SEMAFORO *s) {  
    s->contador+=1;  
    if(s->contador <= 0) {  
        Sacar proceso k de s->q;  
        Despertar(k) ;  
    }  
}
```

```
P (SEMAFORO *s) {  
    s->contador-=1;  
    if(s->contador < 0) {  
        Añadimos proc. a s->q;  
        Bloquear;  
    }  
}
```



Dependiendo del valor inicial de *contador*,  
pueden permitir que varios procesos entren en  
la sección crítica al mismo tiempo en caso de  
necesitarse esta posibilidad

Doble uso de los semáforos:  
Exclusión mutua  
Sincronización

Mutex = abreviatura de "exclusión mutua"

Principal medio de la implementación de sincronización de **hilos de ejecución** y la protección de los datos compartidos con múltiples escrituras concurrentes.

Una variable mutex actúa como una cerradura

Varios **hilos de ejecución** pueden intentar bloquear un mutex, sólo uno será un éxito; otros hilos serán bloqueados hasta que el subproceso propietario desbloquee que mutex.

Una secuencia típica en el uso de un mutex es como sigue:

1. Crear e inicializar una variable mutex
2. Varios hilos intentan bloquear el mutex
3. **Sólo uno tiene éxito y ese hilo posee el mutex**
4. El hilo propietario realiza un conjunto de acciones
5. **El propietario desbloquea el mutex**
6. Otro hilo adquiere el mutex y repite el proceso
7. Finalmente, el mutex se destruye

- Creation:

```
pthread_mutex_t my = PTHREAD_MUTEX_INITIALIZER
```

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *restrict);
```

- Destroying:

```
pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Locking and unlocking mutexes

```
pthread_mutex_lock(pthread_mutex_t *mutex);
```

```
pthread_mutex_trylock(pthread_mutex_t *mutex);
```

```
pthread_mutex_unlock(pthread_mutex_t *mutex);
```



```
1 #include <stdio.h>
2 #include <pthread.h>
3
4 void *funcion_hilo(void * param);
5 void algo(int );
6 int sum;
7 pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
8
9 main() {
10
11     int i;
12     pthread_t thread_id;
13
14     sum=0;
15     printf("Previo creacion de hilos sum=%d\n",sum);
16
17     pthread_create( &thread_id, NULL, funcion_hilo, NULL);
18     for (i=0; i<10000; i++) {
19         algo(16);
20         pthread_mutex_lock(&mutex);
21         sum ++;
22         pthread_mutex_unlock(&mutex);
23     }
24     pthread_join( thread_id, NULL);
25     printf("Hilo principal sum=%d\n", sum);
26     pthread_mutex_destroy(&mutex);
27     return 0;
28 }
```

```
29
30 void * funcion_hilo(void *param)
31 {
32     int i;
33     printf("Hilos %lu \n", pthread_self());
34
35     for (i=0; i<10000; i++) {
36         pthread_mutex_lock(&mutex);
37         sum ++;
38         pthread_mutex_unlock(&mutex);
39         algo(16);
40     }
41     pthread_exit(0);
42 }
43
44 void algo(int n){
45     usleep(n);
46 }
```

```
1  #include <stdio.h>
2  #include <pthread.h>
3  #include <unistd.h>
4
5  #define CHILOS 20
6  int global=0;
7  void * funcion_hilo(void *);
8  void hacer_algo(int );
9
10 int main(){
11     int i;
12     pthread_t hilos[ CHILOS ];
13     for(i=0; i < CHILOS; i++){
14         pthread_create(&hilos[i], NULL, funcion_hilo, NULL);
15     }
16     for(i=0; i < CHILOS; i++){
17         pthread_join(hilos[i], NULL);
18     }
19     return 0;
20 }
21
22 void * funcion_hilo(void *args){
23     hacer_algo(global);
24     printf("%d \n", global);
25     global = global + 1;
26     pthread_exit(NULL);
27 }
28
29 void hacer_algo(int n){ usleep(10000);}
```

```
1 #include <stdio.h>
2 #include <pthread.h>
3 #include <unistd.h>
4 #define CHILOS 40
5
6 int global=0;
7 void * funcion_hilo1(void *);
8 void hacer_algo(long int );
9
10 int main(){
11     int i;
12     pthread_t hilos[ CHILOS ];
13     for(i=0; i < CHILOS; i++){
14         pthread_create(&hilos[i], NULL, funcion_hilo1, NULL);
15     }
16     for(i=0; i < CHILOS; i++){
17         pthread_join(hilos[i], NULL);
18     }
19     printf("%d\n", global);
20     return 0;
21 }
22
23 void * funcion_hilo1(void *args){
24     int myturno;
25     hacer_algo(100);
26     myturno = global;
27     printf("%lu %d\n", pthread_self(), myturno);
28     global = global+1;
29     pthread_exit(NULL);
30 }
31
32 void hacer_algo(long int n){    usleep(n); }
```