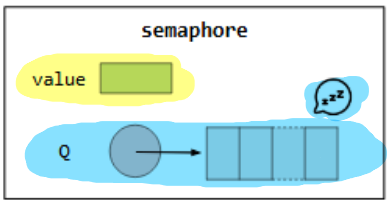
25/11/2025 - Sistemas Operativos (UdeC)

1. Repaso clase anterior

a. Semaforos

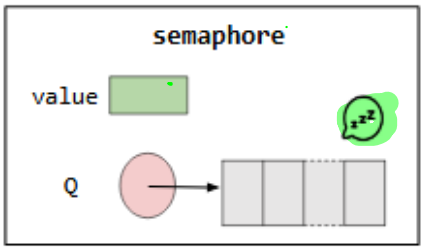
Semáforo

- Objeto con un valor entero no negativo (asociado al estado).
- Cola de hilos que se encuentran dormidos.




```
typedef struct {
  int value;
  struct process *Q;
} sem_t;
```

Semáforo




```
typedef struct {
  int value;
  struct process *Q;
} sem_t;
```

init

```
sem_init(sem_t *s, int value) {
  s->value = initval
}
```

wait

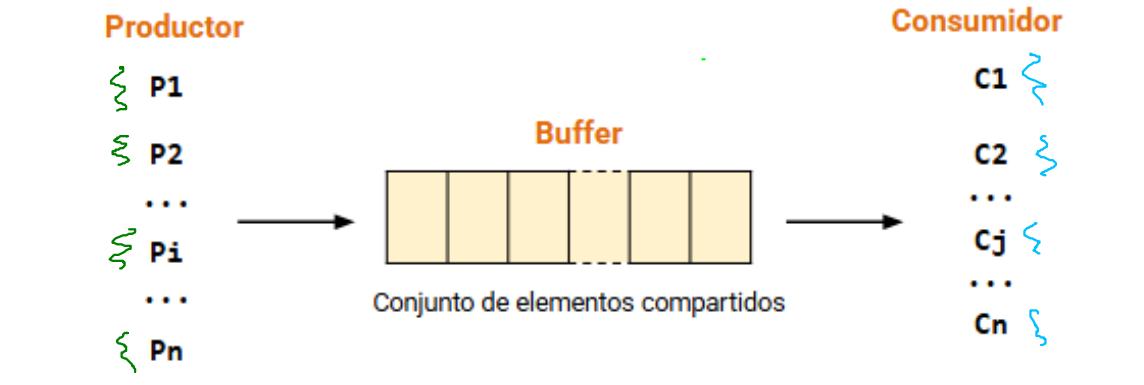
```
void sem_wait(sem_t *s) { // Must be executed atomically
  s->value--;
  if (S->value < 0) {
    add this process to s->Q;
    block();
  }
}
```

post

```
void sem_post(sem_t *s) { // Must be executed atomically
  s->value++;
  if (s->value <= 0) {
    remove a process P from s->Q;
    wakeup(P);
  }
}
```

b. Problema del productor consumidor.

- También conocido como el problema del buffer limitado (Bounded Buffer Problem)
- El productor y el consumidor se ejecutan a tasas diferentes:
  - No hay serialización de uno detrás de otro.
  - Las tareas son independientes.



Implementación definitiva

```
sem_t empty;
sem_t full;
sem_t mutex;

int loops = 50; // Variable global

int main(int argc, char *argv[]) {
  // ...
  sem_init(&empty, 0, MAX); // MAX are empty
  sem_init(&full, 0, 0);    // 0 are full
  sem_init(&mutex, 0, 1);   // mutex = 1
  // ...
  pthread_t C, P;
  pthread_create(P, NULL, producer, NULL);
  pthread_create(C, NULL, consumer, NULL);
  // ...
  return 0;
}
```

```
void *producer(void *arg) {
  int i;
  for (i = 0; i < loops; i++) {
    sem_wait(&empty);   // Line P1
    sem_wait(&mutex);   // line P2
    put(i);             // Line P3
    sem_post(&mutex);   // line P4
    sem_post(&full);    // Line P5

  }
}
```

```
void *consumer(void *arg) {
  int tmp = 0;
  while (tmp != -1) {
    sem_wait(&full);    // Line C1
    sem_wait(&mutex);   // line C2
    tmp = get();        // Line C3
    sem_post(&mutex);   // line C4
    sem_post(&empty);   // Line C5
    printf("%d\n", tmp);
  }
}
```

2. Pasos

i. **Identificar recursos compartidos**: Detectar que datos o estructuras (colas, pilas, arreglos, archivos, etc). son accedidos simultaneamente

ii. **Delimitar la seccion critica**: Identificar el fragmento de codigo donde se leen o modifican dichos recursos

iii. **Garantizar exclusión mutua**: Aplicar mecanismos de bloqueo (Mutex, semaforos, spinlocks) para proteger la seccion critica.

iv. **Coordinacion de hilos**: Definir la logica de espera (wait) y notificacion (signal) segun el estado del recurso

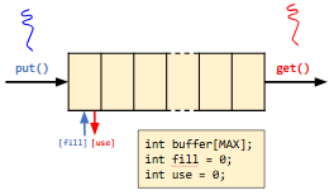
v. **Analisis de la solución**: Verificar y corregir posibles interbloqueos (deadlocks) e inanicion (starvation)

Recordemos el problema del productor – consumidor

Asegurar: safe thread

① Recurso compartido

Buffer (Circular) de tamaño fijo



② Mecanismo

Semaforos { empty, full }

Mutex { mutex }

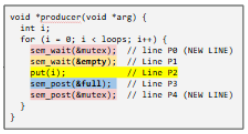
```
sem_t empty = MAX;
sem_t full = 0;
sem_t mutex = 1;
```

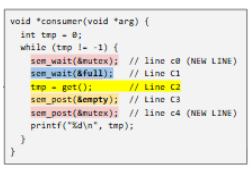
③ Desafio

Sincronizar de modo que:

a. No se escriba en el buffer si esta lleno.

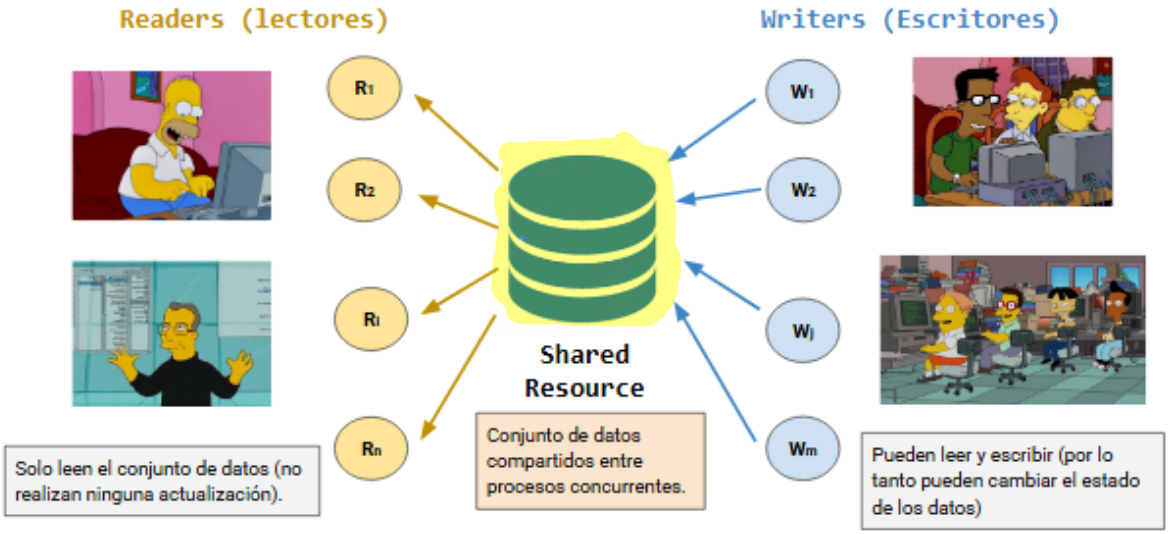
b. No se lea si esta vacio.





3. Problema de los lectores – escritores. (Readers – Writers)

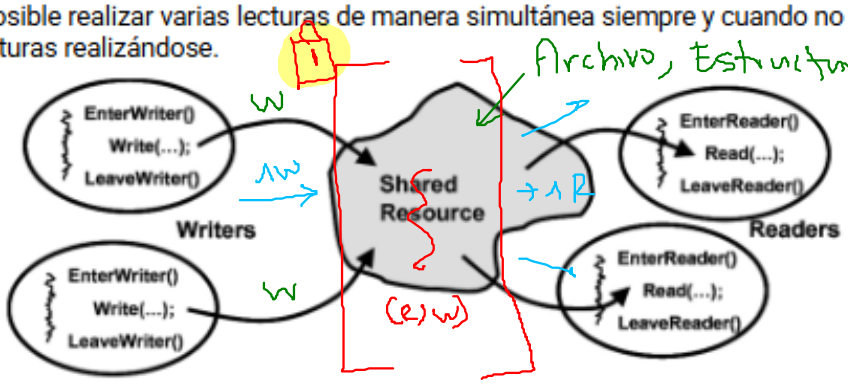
Problema en el cual se tienen un número de operaciones concurrentes sobre los datos (compartidos) que solo son lecturas y escrituras.



Sobre el problema

Las operaciones de lectura y escritura sobre los datos se ejecutan de manera concurrente teniendo en cuenta las siguientes restricciones:

- **Escritura**:
  - Solo un escritor puede acceder a los datos compartidos a la vez.
  - Cambia los datos (acceso a la región crítica)
- **Lectura**:
  - Solo lee los datos.
  - Es posible realizar varias lecturas de manera simultánea siempre y cuando no hayan escrituras realizándose.



① Recurso compartido

Datos / Archivo

② Mecanismo

RW_Lock (cerrojo de Lectura / escritura)

③ Desafío

Evitar la inanición de escritores.

Reader/Writer Lock (`rwlock_t`)

- Tipo especial de lock implementado para operaciones de lectura y escritura.
- Se puede implementar con semáforos:
  - `readers`: Número de hilos de lectura que se encuentran actualmente en la estructura de datos.
  - `writelock`: Lock de escritura.
  - `lock`: Lock de lectura.


```
typedef _struc_t rwlock_t {
  sem_t lock;       // binary semaphore (basic lock)
  sem_t writelock ;  // used to allow ONE writer or MANY readers
  int readers;      // count of readers reading in critical section
} rwlock_t ;
```

Operaciones estructura Reader/Writer Lock (`rwlock_t`)

Inicialización

- `rwlock_init`: Inicializa la estructura `rwlock_t`

```
void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}
```

Operaciones de sincronización básicas para actualización de datos

- `rwlock_acquire_writelock`: Permite al escritor adquirir el `writelock`

```
void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}
```

- `rwlock_release_writelock`: Libera el `writelock`

```
void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```

Estas operaciones usan el semáforo writelock internamente para asegurar que solo un escritor puede ingresar a la región crítica para actualizar los datos.

Operaciones de sincronización básicas la lectura de datos

- `rwlock_acquire_readlock`: Permite al lector adquirir el `lock`

```
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}
```

- `rwlock_release_readlock`: Libera el `Lock`

```
void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```

Resumen estructura y operaciones `rwlock_t`

```
typedef _struc_t rwlock_t {
  sem_t lock;          // binary semaphore (basic lock)
  sem_t writelock ;    // used to allow ONE writer or MANY
                       // readers
  int readers;         // count of readers reading in
                       // critical section
} rwlock_t ;

void rwlock_init(rwlock_t *rw) {
  rw->readers = 0;
  sem_init(&rw->lock, 0, 1);
  sem_init(&rw->writelock, 0, 1);
}
```

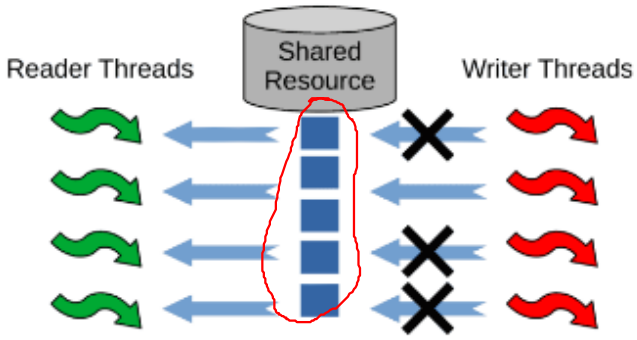
```
void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}

void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}

void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}

void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```

Reader/writer lock

- Solo un **writer** (**escritor**) puede entrar a la **región crítica**. (este writer es el único que adquiere el **look de escritura** (`rw->writelock`))

```
void rwlock_acquire_writelock(rwlock_t *rw) {
  sem_wait(&rw->writelock);
}
```

```
void rwlock_release_writelock(rwlock_t *rw) {
  sem_post(&rw->writelock);
}
```



Reader/writer lock

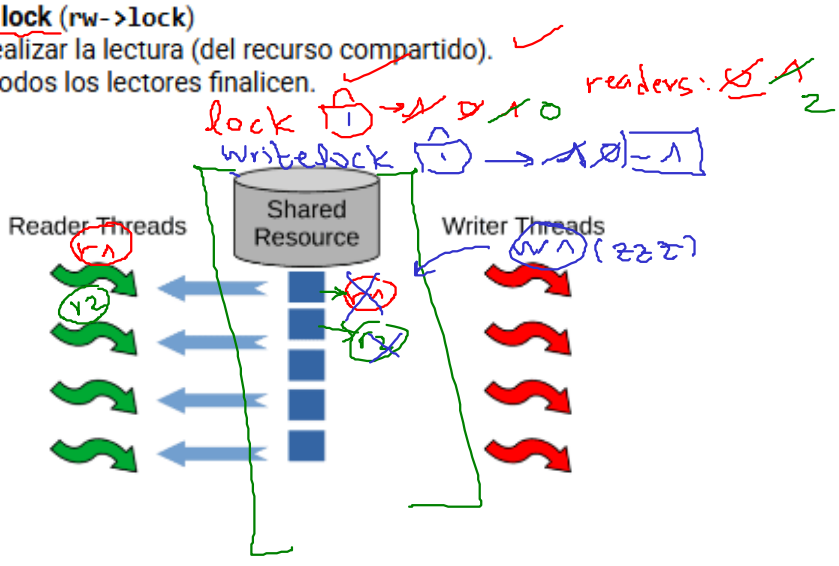
- Una vez que un lector ha adquirido el **write lock** (`rw->lock`)
  - Se permite que más lectores puedan realizar la lectura (del recurso compartido).
  - Cualquier escritor debe esperar a que todos los lectores finalicen.

El primer lector adquiere el **write lock**. (**Open**)

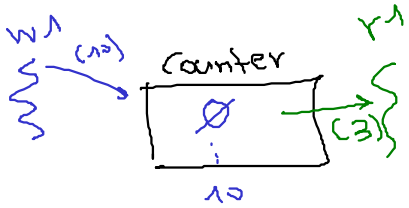
```
void rwlock_acquire_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers++;
  if (rw->readers == 1) // first reader gets writelock
    sem_wait(&rw->writelock);
  sem_post(&rw->lock);
}
```

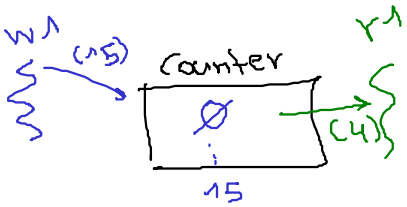
El último lector libera el **write lock** (**Close**)

```
void rwlock_release_readlock(rwlock_t *rw) {
  sem_wait(&rw->lock);
  rw->readers--;
  if (rw->readers == 0) // last reader lets it go
    sem_post(&rw->writelock);
  sem_post(&rw->lock);
}
```



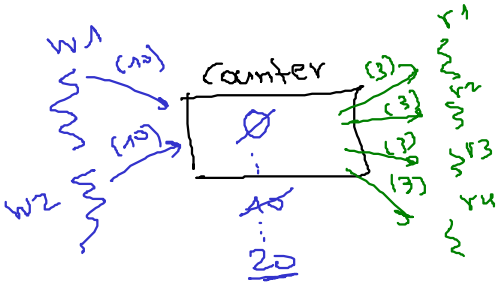
https://github.com/udea-so/apuntes_2025-2_SO-virtual/tree/main/clase_21/code/reader_writer_locks

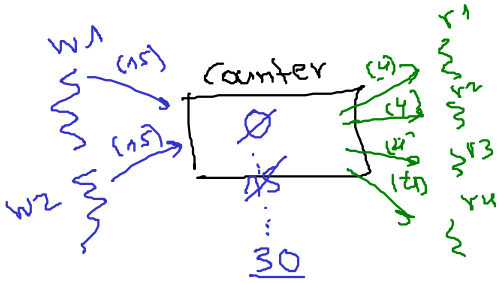





```
de/reader_writer_locks$ ./rwlock 3 10
read 0
read 10
read 10
read done: 10
write done
all done
```

```
de/reader_writer_locks$ ./rwlock 4 15
write done
read 0
read 15
read 15
read 15
read done: 15
all done
```






```
de/reader_writer_locks$ ./rwlock 3 10
read 0
read 10
read 20
read done: 20
write done
read 10
read 20
read 20
read done: 20
read 20
read 20
read 20
read done: 20
read 20
read 20
read 20
read done: 20
write done
all done
```

```
de/reader_writer_locks$ ./rwlock 4 15
read 0
read 15
read 30
read 30
read done: 30
read 30
read 30
read 30
read 30
read done: 30
read 15
read 30
read 30
read 30
read done: 30
write done
write done
read 30
read 30
read 30
read 30
read done: 30
all done
```

Problema de la equidad

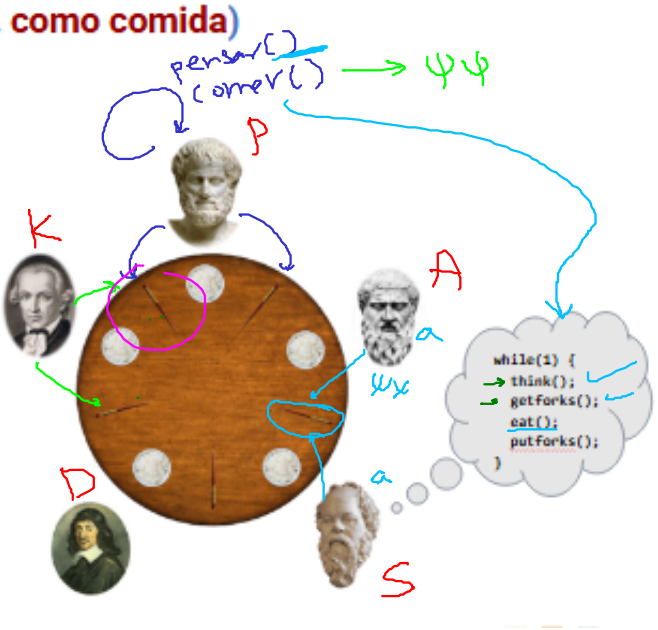
- Muchos lectores pueden fácilmente denegar el acceso al escritor (**starvation**).
- Existen soluciones más sofisticadas a este problema. Por ejemplo: Evitar el acceso de nuevos lectores cuando haya un escritor en espera

4. Problema de la cena de filosofos

Enunciado del problema (Pienso luego... como comida)

El problema de (dining philosophers problem) fue formulado por Dijkstra en 1965 para representar el problema de sincronización de procesos en un Sistema Operativo.

- Se tienen 5 filósofos sentados en una mesa redonda.
- Entre cada filósofo hay un palillo chino (5 en total)
- Cada filósofo pasa su vida pensando (sin necesitar ningún palillo) o comiendo
- Para comer cada filósofo necesita 2 palillos, el que tiene a su derecha y a su izquierda
- Los filósofos compiten por estos palillos

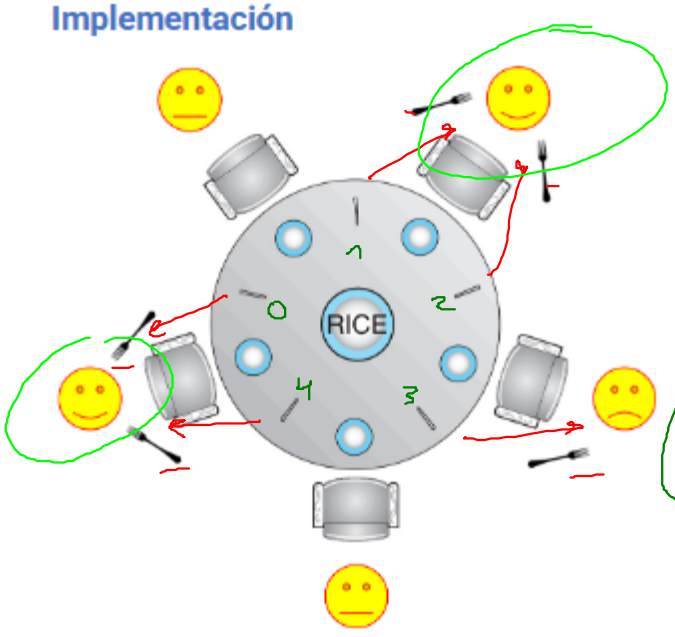


① Recurso compartido — Tenedores

② Mecanismo — Semaforos

③ Desafio — Evitar deadlock

Implementación

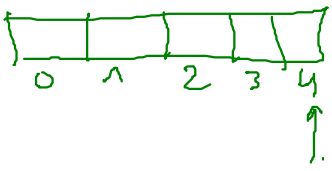


Basic loop of each philosopher

```
while(1) {
  think();
  getforks();
  eat();
  putforks();
}
```

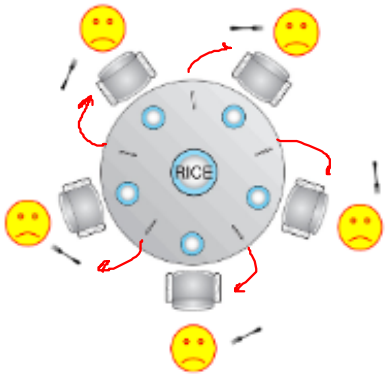
Helper functions (Downey's solutions)

```
// helper functions
int  left(intp) {
  return p;
}

int right(int p) {
  return (p + 1) % 5;
}
```

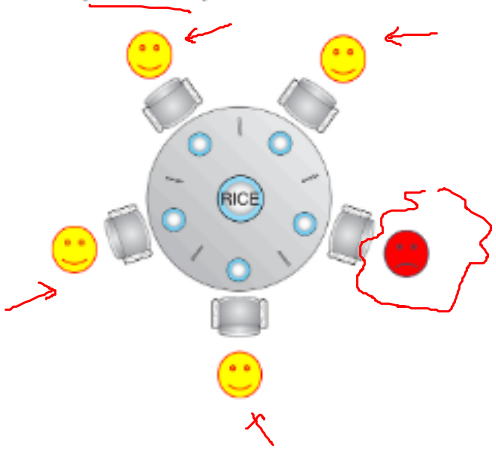


Objetivos

Objetivo 1: Que no hayan interbloqueos (**deadlocks**)



Objetivo 2: Que no se quede ningún filósofo sin comer (**starvation**)



Objetivo 3: Máxima **concurrencia**

Implementación

Funciones de utilidad

Filósofo **p** quiere adquirir palillo de la izquierda:
→ Llamar **`left(p)`**

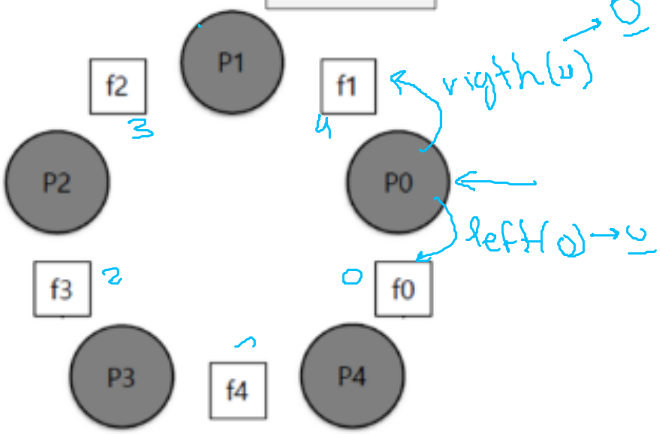
```
int  left(intp) {
  return p;
}
```

Filósofo **p** quiere adquirir palillo de la derecha:
→ Llamar **`right(p)`**

```
int right(int p) {
  return (p + 1) % 5;
}
```

Basic loop of each philosopher

```
while(1) {
  think();
  getforks();
  eat();
  putforks();
}
```

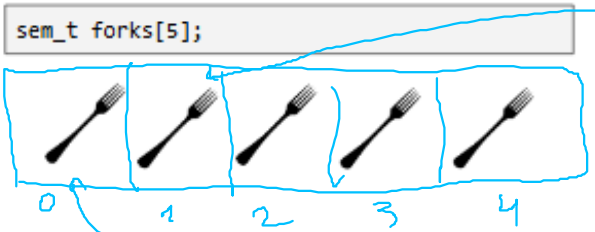


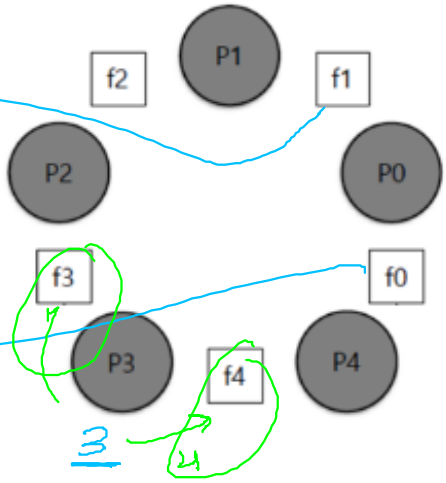
Implementación

Semáforos

Necesitamos varios semáforos para resolver este problema. Para el caso 5. Uno por cada palillo:

```
sem_t forks[5];
```





Solución simple - Primera solución

Implementación inicial del problema

```
// initialized to 1
sem_t forks[5];
```

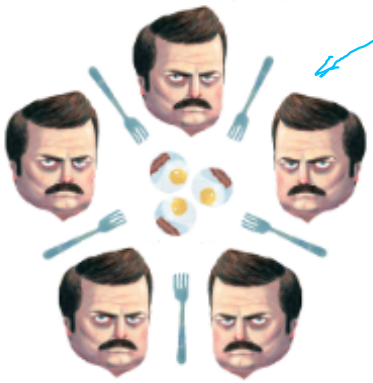
```
void philosopher(int i) {
  while(1) {
    think();
    getforks(i);
    eat();
    putforks(i);
  }
}
```

```
void get_forks(int p) {
  sem_wait(&forks[left(p)]);
  sem_wait(&forks[right(p)]);
}
```

```
void put_forks(int p) {
  sem_post(&forks[left(p)]);
  sem_post(&forks[right(p)]);
}
```

Solución simple - Primera solución

Implementación inicial del problema

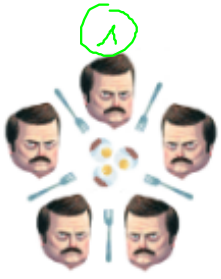


```
int main(int argc, char *argv[]) {
  // ...
  for (int i = 0; i < 5; i++) {
    sem_init(&forks[i], 0, 1); // initialized to 1
  }
  // ...
}
```

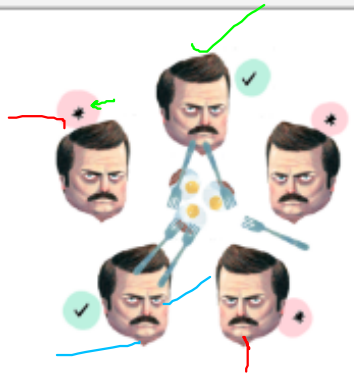
Solución

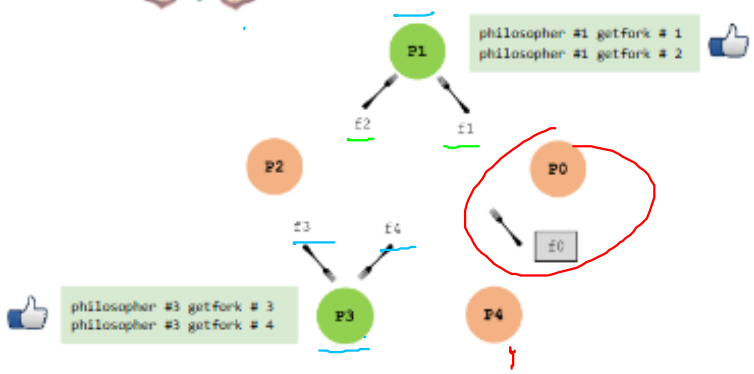
Solución simple - Primera solución

```
int main(int argc, char *argv[]) {
  // ...
  for (int i = 0; i < 5; i++) {
    sem_init(&forks[i], 0, 1); // initialized to 1
  }
  // ...
}
```



```
void philosopher(int i) {
  while(1) {
    think();
    getforks(i);
    eat();
    putforks(i);
  }
}
```



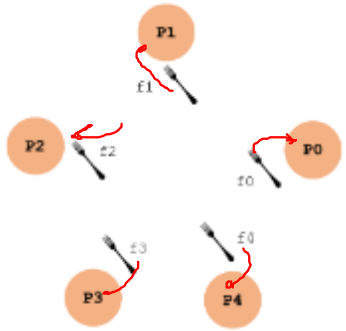


Solución

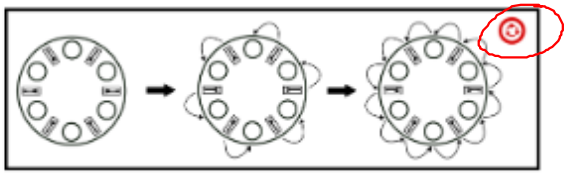
Solución simple - Primera solución

```
int main(int argc, char *argv[]) {
  // ...
  for (int i = 0; i < 5; i++) {
    sem_init(&forks[i], 0, 1); // initialized to 1
  }
  // ...
}
```

```
void philosopher(int i) {
  while(1) {
    think();
    getforks(i);
    eat();
    putforks(i);
  }
}
```



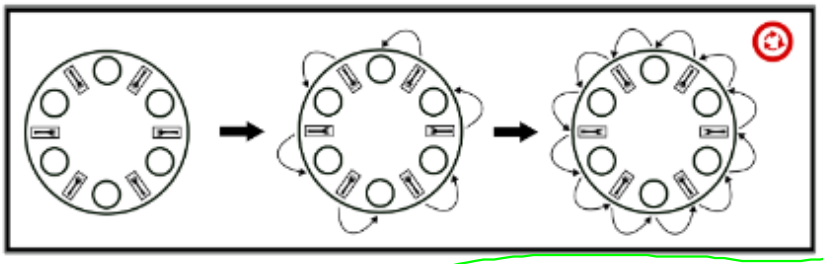

```
philosopher #0 getfork # 0
philosopher #1 getfork # 1
philosopher #2 getfork # 2
philosopher #3 getfork # 3
philosopher #4 getfork # 4
```



Solución simple - Interbloqueo

El interbloqueo se puede dar:

1. Si cada filósofo toma el palillo a su izquierda antes de que cualquier otro filósofo tome el palillo a su derecha.
2. Cada filósofo se quedará bloqueado para siempre, sosteniendo el tenedor y esperando a que otro suelte el palillo.



Reescribir el programa o mirar la lógica

Solución al problema del interbloqueo

Cambiar el orden en que los semáforos con adquiridos

```
// initialized to 1
sem_t forks[5];
```

```
void philosopher(int i) {
  while(1) {
    think();
    getforks(i);
    eat();
    putforks(i);
  }
}
```


```
void get_forks(int p) {
  if (p == 4) {
    sem_wait(&forks[right(p)]);
    sem_wait(&forks[left(p)]);
  }
  else {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
  }
}
```

4 (D,I)

0,1,2,3 (I,D)

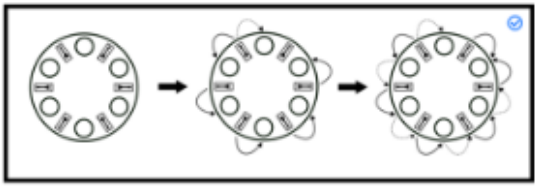

```
void put_forks(int p) {
  sem_post(&forks[left(p)]);
  sem_post(&forks[right(p)]);
}
```

Solución al problema del interbloqueo

Cambiar el orden en que los semáforos con adquiridos:

- Supongamos que para el caso, el último filósofo (4) adquiere los palillos en un orden diferente.
- En este caso no se da la situación en la cual un filósofo que coja un palillo se quede esperando a otro.
- Hay otras soluciones diferentes a esta.
- Existen otros problemas famosos de sincronización como el problema de los fumadores o el problema del barbero dormilón.


```
void get_forks(int p) {
  if (p == 4) {
    sem_wait(&forks[right(p)]);
    sem_wait(&forks[left(p)]);
  }
  else {
    sem_wait(&forks[left(p)]);
    sem_wait(&forks[right(p)]);
  }
}
```



https://github.com/udea-so/apuntes_2025-2_SO-virtual/tree/main/clase_21/code/dining_philosophers

```
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_clases/2025-2/repos/apuntes_2025-2_SO-virtual/clase_21/co
de/dining_philosophers$ make
gcc -Wall -Werror -pthread -c dining_philosophers_deadlock.c
gcc dining_philosophers_deadlock.o -o dining_philosophers_deadlock  -pthread
gcc -Wall -Werror -pthread -c dining_philosophers_deadlock_print.c
gcc dining_philosophers_deadlock_print.o -o dining_philosophers_deadlock_print  -pthread
gcc -Wall -Werror -pthread -c dining_philosophers_no_deadlock.c
gcc dining_philosophers_no_deadlock.o -o dining_philosophers_no_deadlock  -pthread
gcc -Wall -Werror -pthread -c dining_philosophers_no_deadlock_print.c
gcc dining_philosophers_no_deadlock_print.o -o dining_philosophers_no_deadlock_print  -pthread
rm -f dining_philosophers_deadlock.o dining_philosophers_deadlock_print.o dining_philosophers_no_deadlock.o dining_philo
sophers_no_deadlock_print.o
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_clases/2025-2/repos/apuntes_2025-2_SO-virtual/clase_21/co
de/dining_philosophers$ ls
Makefile                    dining_philosophers_deadlock        dining_philosophers_no_deadlock
README.md                   dining_philosophers_deadlock.c      dining_philosophers_no_deadlock.c
common.h                    dining_philosophers_deadlock_print  dining_philosophers_no_deadlock_print
common_threads.h            dining_philosophers_deadlock_print.c  dining_philosophers_no_deadlock_print.c
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_clases/2025-2/repos/apuntes_2025-2_SO-virtual/clase_21/co
de/dining_philosophers$
```

```
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_
de/dining_philosophers$ ./dining_philosophers_deadlock 10
dining: started
dining: finished
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_
de/dining_philosophers$ ./dining_philosophers_deadlock 100
dining: started
dining: finished
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_
de/dining_philosophers$ ./dining_philosophers_deadlock 1000
dining: started
dining: finished
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_
de/dining_philosophers$ ./dining_philosophers_deadlock 10000
dining: started
```

```
hilosophers_deadlock_print 1
dining: started
0: start
0: think
0: try 0
0: try 1
0: eat
0: done
                    2: start
                    2: think
                    2: try 2
                    2: try 3
                    2: eat
                    2: done
                              3: start
                              3: think
                              3: try 3
                              3: try 4
                              3: eat
                              3: done
          1: start
          1: think
          1: try 1
          1: try 2
          1: eat
          1: done
                                        4: start
                                        4: think
                                        4: try 4
                                        4: try 0
                                        4: eat
```

```
/dining_philosophers$ ./dining_philosophers_deadlock_print 10000
```

```
                                        4: try 0
                    2: eat
                    2: done
                    2: think
                    2: try 2
          1: eat
          1: done
          1: think
          1: try 1
                                  3: try 4
0: eat
0: done
0: think
0: try 0
                                        4: eat
                                        4: done
                                        4: think
                                        4: try 4
                                  3: eat
                                  3: done
                                  3: think
                                  3: try 3
                                  3: try 4
                                        4: try 0
          1: try 2
                    2: try 3
0: try 1
```

```
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_clases/2025-2/repo
lase_21/code/dining_philosophers$ ./dining_philosophers_no_deadlock 100
dining: started
dining: finished
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_clases/2025-2/repo
lase_21/code/dining_philosophers$ ./dining_philosophers_no_deadlock 100000
dining: started
dining: finished
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_clases/2025-2/repo
lase_21/code/dining_philosophers$ ./dining_philosophers_no_deadlock 1000000
dining: started
dining: finished
tigarto@DESKTOP-EAVDF2C:/mnt/c/Users/Usuario/Documents/UdeA/SO_clases/2025-2/repo
lase_21/code/dining_philosophers$
```

```
lase_21/code/dining_philosophers$ ./dining_philosophers_no_deadlock_print 10000
```

```
4: done
4: think
4 try 0
4 try 4
4: eat
4: done
4: think
4 try 0
4 try 4
4: eat
4: done
dining: finished
```

5. Zemaforos (Implementacion alternativa)

Implementación alternativa

A continuación se muestra una implementación alternativa de los **Semaforos** conocida como **Zemaforos** y construida usando Locks y variables de condición.

```
void sem_wait(sem_t *s) {
  s->value--;
  if (s->value < 0) {
    wait;
  }
}

void sem_post(sem_t *s) {
  s->value++;
  if (s->value <= 0) {
    wake one waiting thread;
  }
}
```

```
typedef struct __Zem_t {
  int value;
  pthread_cond_t cond;
  pthread_mutex_t lock;
} Zem_t;

// only one thread can call this
void Zem_init(Zem_t *s, int value) {
  s->value = value;
  Cond_init(&s->cond);
  Mutex_init(&s->lock);
}

void Zem_wait(Zem_t *s) {
  Mutex_lock(&s->lock);
  while (s->value <= 0)
    Cond_wait(&s->cond, &s->lock);
  s->value--;
  Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
  Mutex_lock(&s->lock);
  s->value++;
  Cond_signal(&s->cond);
  Mutex_unlock(&s->lock);
}
```

Implementación alternativa

Comentarios sobre la implementación

- El valor del semáforo en esta implementación no representa el número de hilos bloqueados en el semáforo (cuando el valor es negativo).
- Fácil implementación, similar a la implementación actual en linux.

```
void sem_wait(sem_t *s) {
  s->value--;
  if (s->value < 0) {
    wait;
  }
}

void sem_post(sem_t *s) {
  s->value++;
  if (s->value <= 0) {
    wake one waiting thread;
  }
}
```

```
void Zem_wait(Zem_t *s) {
  Mutex_lock(&s->lock);
  while (s->value <= 0)
    Cond_wait(&s->cond, &s->lock);
  s->value--;
  Mutex_unlock(&s->lock);
}

void Zem_post(Zem_t *s) {
  Mutex_lock(&s->lock);
  s->value++;
  Cond_signal(&s->cond);
  Mutex_unlock(&s->lock);
}
```

6. Conclusiones

- Los **semáforos** son equivalentes a **locks + CV** (Variables de condición).
  - Pueden ser usados para exclusión mutua y ordenamiento.
- Los semáforos tienen un estado:
  - La forma como serán inicializados depende de cómo serán usados.
    - **Init sem = 1**: Mutex
    - **Init sem = 0**: Join.
    - **Init sem = N**: Número de recursos disponibles.
  - **Operaciones**:
    - **sem→wait()**: Espera hasta que el valor del semáforo sea mayor que 0, entonces decrementa (atómico)
    - **sem→post()**: Incrementa el valor del semáforo, entonces despierta a uno de los hilos que se encuentra esperando (Atómico).
- **Problemas de sincronización analizados**:
  - Buffer limitado
  - Escritores y Lectores
  - Cena de filosofos.