URJC Escuela de Ingeniería de Fuenlabrada

GSyC RoboticsLabURJC Programming Robot Intelligence

Sistemas Distribuidos y Concurrentes

Concurrencia Avanzada

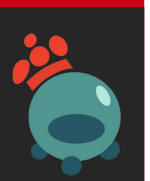
Grado en Ingeniería de Robótica Software

Teoría de la Señal y las Comunicaciones y Sistemas Telemáticos y Computación

Roberto Calvo Palomino
roberto.calvo@urjc.es

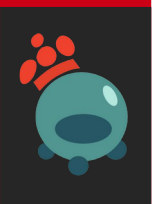
Índice

- Repaso Concurrencia
- Mutex
- Semáforos
- Variables condición
- Productores/Consumidores
- Lectores/Escritores
- Monitores
- Problema de la cena de los filósofos



Repaso Concurrencia

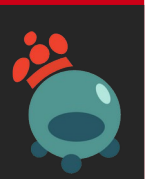
- Condición de carrera
- Operación atómica
- Exclusión mutua
- Región crítica
- Espera activa
- Mutex
- Spinlock



Proteger regiones críticas (Thread-Safe)

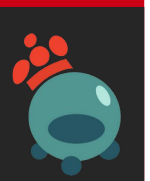
Sincronización a través de señales

Evitar espera activa



Espera Activa

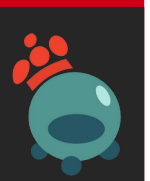
- Técnica donde un proceso está repetidamente comprobando una condición
  - Esperar pulsación de teclado
  - Esperar a habilitación de sección crítica.
- Estrategia válida para sincronización de procesos
  - sistemas con múltiples procesadores (SMP)
- Esta técnica debe evitarse **SIEMPRE** que sea posible
  - Uso innecesario de CPU



Mutex

- Los mutex son mecanismos que se usan en programación concurrente para evitar que entre más de un proceso a la vez en la sección crítica
- La sección crítica es el fragmento de código donde puede modificarse un recurso compartido.
- Modificar un recurso compartido concurrentemente sin protección conlleva resultados indeterminados.
- Operaciones lock() y unlock()

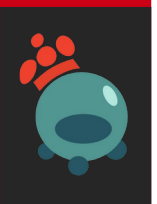
```
pthread_mutex_t mutex;


pthread_mutex_lock(&mutex);
// región crítica
pthread_mutex_unlock(&mutex);
```



Ejemplo

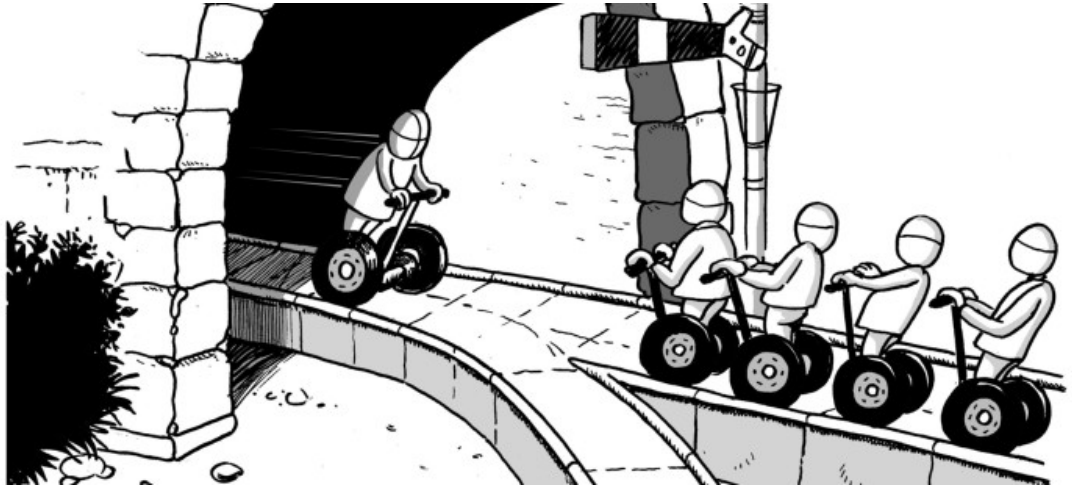
```c
int value = 0, N=1000000;

void thread_function1 (void) {

    for (int i=0; i<N; i++) {
        value = value + i;
    }
}

void thread_function2 (void) {

    for (int i=0; i<N; i++) {
        value = value - i;
    }
}
```

```c
int main (int argc, char* argv[]) {

    pthread_t th1, th2;

    pthread_create(&th1, attr: NULL,
                   (void *)&thread_function1,
                   arg: NULL);
    pthread_create(&th2, attr: NULL,
                   (void *)&thread_function2,
                   arg: NULL);

    pthread_join(th1, thread_return: NULL);
    pthread_join(th2, thread_return: NULL);

    printf ( format: "Resultado final = %d\n", value);

    return 0;
}
```



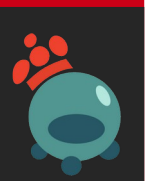
Semáforos

- El concepto de **semáforos** fue el primero en solucionar problemas de sincronización sin espera activa. Fue inventado Edsger W. Dijkstra (finales 1960)
- Los procesos se **bloquean** o **ejecutan** condicionados únicamente por el valor que tiene una variable entera.
- Basados en las señales visuales ferroviarias que indican si un tren puede entrar o no en una vía.



Semáforos

- Típicamente los semáforos permiten las siguientes operaciones:
  - **init()**: Establece el valor inicial del contador del semáforo.
  - **wait(P)**: decrementa (lock) el contador del semáforo.
    - Si el valor del contador > 0, se decrementa el contador y finaliza.
    - Si el valor del contador == 0, la llamada se bloquea
    - wait, acquire,...
  - **signal(V)**: incrementa (unlock) el contador del semáforo.
    - Después de la actualización del contador, si algún proceso está bloqueado en el wait() será despertado y realizará el lock() sobre el semáforo.
    - signal, post, release



Semáforos

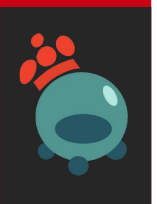
```
CONFORMING TO
       POSIX.1-2001.
```

- Están implementados en Linux desde la versión 2.6

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
```

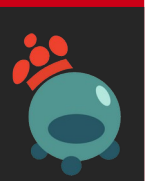
  - Semáforo compartido entre threads del mismo proceso(pshared=0)
  - Semáforo compartido entre procesos (pshared=1)

```
#include <semaphore.h>

int sem_wait    (sem_t *sem);
int sem_trywait (sem_t *sem);
int sem_post    (sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy (sem_t *sem);
```



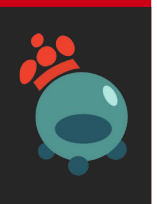
Semáforos

- Semáforos System V (también POSIX)
- Parte del módulo IPC del UNIX System V.
- Fue el estándar de facto durante muchos años y siguen disponibles en las últimas versiones de Linux y Solaris.
- API compleja
- Cada vez se usan menos en favor de los POSIX Semaphores de 2001.
- Operaciones con:
  - `semctl y semop`



Semáforos

- Diferentes usos:
  - Exclusión mutua: La implementación utilizando semáforos se reduce a iniciar el semáforo con el valor ‘1’.
    - Cuando un proceso entra, mediante *wait()* decrementará el contador y ejecutará en la sección crítica. Posteriormente ejecutará *signal()*
    - Si un segundo proceso entra, al ejecutar el *wait()* el contador valdrá 0 y por tanto se bloqueará hasta que el otro proceso ejecute *signal()*
  - Sincronización:
    - El valor del contador del semáforo se puede interpretar como el número de recursos del sistema a utilizar
    - Por ejemplo: Un sistema con N procesadores, se limita el acceso a la sección crítica a N procesos (multiplexes).

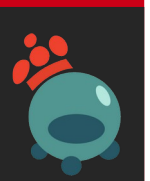


Semáforos

- Ejemplo de exclusión mutua

```
NUM_THREADS=4
MAX_COUNT=1000
counter=0

for (i=0; i<NUM_THREADS; i++){

    id[i].tid = i;

    rc = pthread_create(&threads[i],
                        NULL, count,
                         NULL);

}

for(i=0; i<NUM_THREADS; i++)
    pthread_join(threads[i], NULL);


printf("Contador:%d\n", counter);
```

```
void *count(void *ptr) {

    long i, max = MAX_COUNT/NUM_THREADS;

    for (i=0; i < max; i++) {

        counter += 1;

    }
}
```

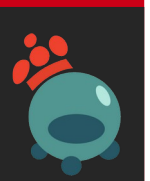


Semáforos

- Ejemplo de sincronización: Sólo 4 hilos pueden

```
NUM_THREADS=4
NUM_MAX_THREAD=1000
int i;

for (i=0; i<NUM_MAX_THREAD; i++){

    rc = pthread_create(&threads[i],
                        NULL, count,
                         NULL);

}

for(i=0; i<NUM_THREADS; i++)
    pthread_join(threads[i], NULL);
```

```
void *func(void *ptr) {


    // Comprimir datos


}
```

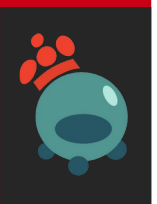


Semáforos

- El uso inadecuado de semáforos:
  - Puede dejar varios procesos/hilos bloqueados
  - Si el único proceso/hilo que puede despertar a otros y éste se queda bloqueado, puede no despertar nunca al resto.
- La situación más grave ocurre cuando se producen **interbloqueos**, 2 hilos se quedan bloqueados esperando a que el otro lo despierte.

```
void *thread1(void *ptr) {

    …
    sem_wait(&mutex1);
    sem_wait(&mutex2);
    …

}
```

```
void *thread2(void *ptr) {

    …
    sem_wait(&mutex2);
    sem_wait(&mutex1);
    …

}
```



Problemas de sincronización

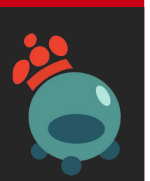
- Dado un *buffer* compartido, un thread quiere escribir de él y otro thread distinto quiere leer.
  - El que escribe no puede hacerlo si el buffer está lleno.
  - El que lee no puede hacerlo si el buffer está vacío.

Escribe en el buffer

```
for (;;) {

  pthread_mutex_lock(&mutex);

  if ( size(BUFFER) != MAX_BUFFER)
    writeBuffer(elem);

  pthread_mutex_unlock(&mutex);

}
```

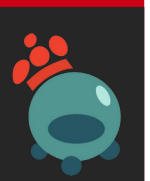
Lee del buffer

```
for (;;) {

  pthread_mutex_lock(&mutex);

  if ( size(BUFFER) != 0)
    elem = readBuffer();

  pthread_mutex_unlock(&mutex);

}
```



Variables condición

- Mutex o semáforos binarios tienen la **limitación** que se ha de esperar si el cerrojo ya está cogido, sin poder continuar con la ejecución.
- Las variables condición ofrecen una extensión sobre el comportamiento de los mutex ofreciendo respuesta ante determinados eventos.
- Nos permiten bloquear el hilo ejecución y a la vez liberar el mutex para que otro proceso pueda continuar.
- Es una de las múltiples soluciones propuestas para evitar realizar espera activa.



Variables condición

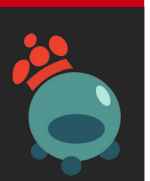
- La espera activa genera un bloqueo mutuo (deadlock)
- Las variables condición son una solución idónea y eficiente en este tipo de escenarios.
- Las variables condición deben ser llamadas con el **lock** del mutex asociado.
- La librería pthread ofrece variables condición

```
pthread_mutex_lock(&mutex);
if (CONDICION)
    pthread_cond_wait(&not_full, &mutex);
```

cond_wait libera el mutex y deja bloqueada la ejecución esperando el signal de la variable condición.

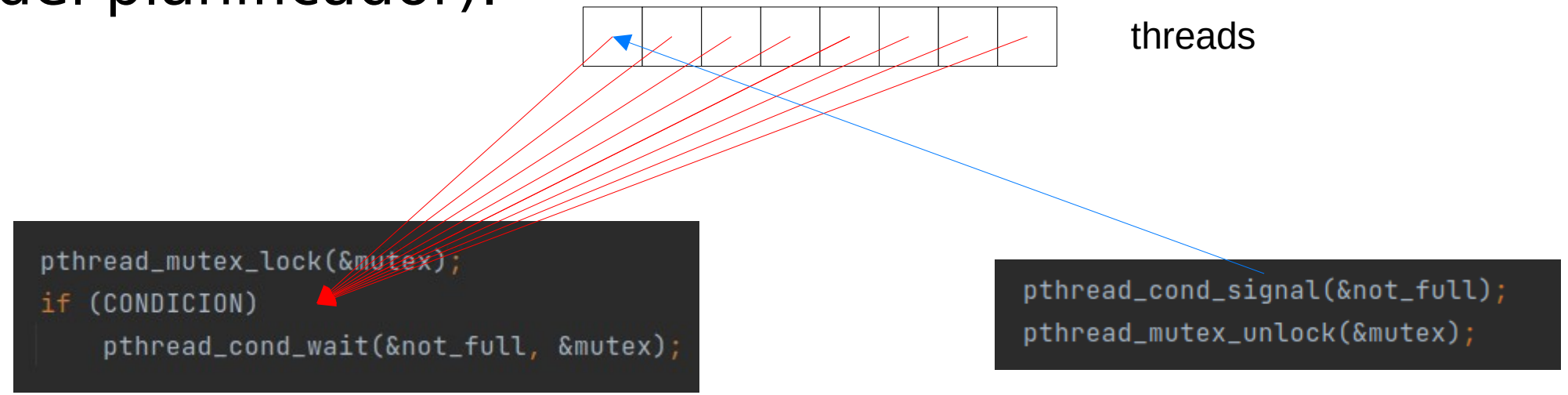
```
pthread_cond_signal(&not_full);
pthread_mutex_unlock(&mutex);
```

cond_signal libera el mutex y deja bloqueada la ejecución esperando el signal de la variable condición.

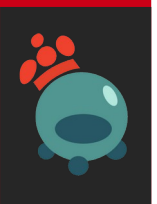


Variables condición

- Si muchos threads se quedan bloqueados esperando a la variable condición, cada cond_signal ejecutado desbloqueará un único thread (dependerá de la política del planificador).



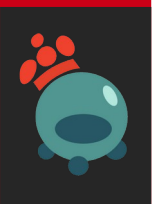
- *pthread_cond_broadcast()* desbloquea todos los threads bloqueados en la variable condición.



Problemas de sincronización

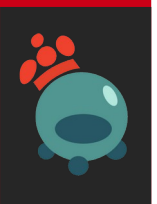
- Veremos en detalle los siguientes problemas de sincronización y como resolverlos utilizando mutex, semáforos o monitores.

- Productores/Consumidores
- Lectores/Escritores
- El problema de los filósofos cenando



Productores/Consumidores

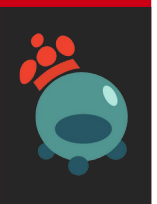
- El problema de **productores/consumidores** puede resolverse utilizando mutex + variables condición.
- Productores/consumidores es un ejemplo clásico de problema de **sincronización** de multiprocesos.
- El programa describe dos procesos, productor y consumidor, ambos comparten un buffer de tamaño finito.
- El productor genera y escribe información en el buffer, y el consumidor recibe y lee información del buffer.
- **Problema**: El productor no puede añadir más elementos que la capacidad del buffer y el consumidor no puede leer elementos de un buffer que está vacio.



Productores/Consumidores

```c
void producer (void) {

  int pos, aux_pos = 0;

  for (int i=0; i<MAX_DATA_PRODUCER; i++) {

    pthread_mutex_lock(&mutex);

    while (n_elems == MAX_BUFFER) {}

    buffer[pos] = i;
    aux_pos = pos;
    pos = (pos + 1) % MAX_BUFFER;
    n_elems ++;

    pthread_mutex_unlock(&mutex);

    printf("Producer[%d] %d \n", aux_pos, i);
  }
  pthread_exit(0);
}
```

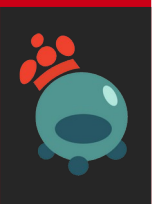
```c
void consumer (void) {

  int data, pos, aux_pos = 0;

  for (int i=0; i<MAX_DATA_PRODUCER; i++) {

    pthread_mutex_lock(&mutex);

    while (n_elems == 0) {}

    data = buffer[pos];
    aux_pos = pos;
    pos = (pos + 1) % MAX_BUFFER;
    n_elems--;

    pthread_mutex_unlock(&mutex);

    printf("Consumer[%d] %d \n", aux_pos, data);
  }
  pthread_exit(0);
}
```



Productores/Consumidores

- La solución pasa por utilizar 2 variables condición para modelar las acciones de
  - “buffer no lleno”
  - “buffer no vacío”

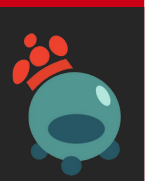
- Importante.
- El uso de las variables condición **siempre** debe realizarse dentro de la región crítica, una vez adquirido el mutex.



Productores/Consumidores

```c
void producer (void) {

  int pos = 0;
  int aux_pos = 0;

  for (int i=0; i<MAX_DATA_PRODUCER; i++) {

    pthread_mutex_lock(&mutex);

    while (n_elems == MAX_BUFFER)
      pthread_cond_wait(&not_full, &mutex);

    buffer[pos] = i;
    aux_pos = pos;
    pos = (pos + 1) % MAX_BUFFER;
    n_elems ++;

    pthread_cond_signal(&not_empty);
    pthread_mutex_unlock(&mutex);

    printf("Producer[%d] %d \n", aux_pos, i);
    usleep(SLEEP_TIME*3);
  }
  pthread_exit(0);
}
```

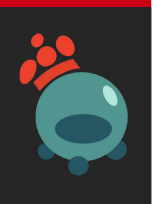
```c
void consumer (void) {

  int data = 0;
  int pos = 0;
  int aux_pos = 0;

  for (int i=0; i<MAX_DATA_PRODUCER; i++) {

    pthread_mutex_lock(&mutex);

    while (n_elems == 0)
      pthread_cond_wait(&not_empty, &mutex);

    data = buffer[pos];
    aux_pos = pos;
    pos = (pos + 1) % MAX_BUFFER;
    n_elems--;

    pthread_cond_signal(&not_full);
    pthread_mutex_unlock(&mutex);

    printf("Consumer[%d] %d \n", aux_pos, data);
    usleep(SLEEP_TIME);
  }
  pthread_exit(0);
}
```



Ejercicio:

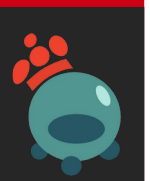
Productores/Consumidores

Implementado con semáforos



Lectores / Escritores

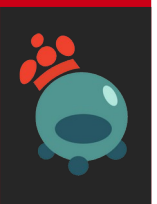
- Problema de exclusión mutua donde numerosos procesos compiten por entrar en la sección crítica.
- Existen 2 tipos de procesos: Lectores y Escritores
- En aplicaciones reales, la mayoría de las operaciones sobre la memoria son lecturas. Y deben ser consistentes.
- Los procesos “lectores” pueden compartir la sección crítica con otros procesos “lectores”, pero nunca con “escritores”
- Los procesos “escritores” necesitan ejecutar con acceso exclusivo y de 1 en 1, ya que realizarán operaciones de escritura.
- Diferentes enfoques:
  - Prioridad para los lectores o prioridad para los escritores



Monitores

- Los semáforos se inventaron para resolver problemas de sincronización sin espera activa.
- Sin embargo, son primitivas de bajo nivel.
- No está estructuradas, y son propensas a errores de programación por parte de los desarrolladores.
- La omisión accidental de un *signal* o *wait* provoca fallos.

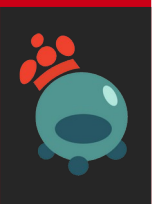
- Los **monitores** son una primitiva con estructura cuya responsabilidad de funcionamiento recae en los módulos del programa.



Monitores

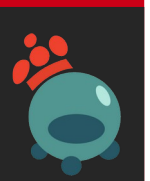
- El nombre de monitor fué acuñado por C.A.R. Hoare [1973]
- B. Hansen incorpora los monitores al lenguaje Pascal Concurrente [1975]
- Proporcionan un mecanismo de abstracción
- Los monitores contienen variables y métodos/funciones.
- Las variables son unicamente accesibles desde los métodos/funciones.
- Ningún método/función del monitor se puede ejecutar si algún otro está ejecutando.

El monitor debe asegurar la exclusión mutua de la ejecución de sus métodos/funciones.



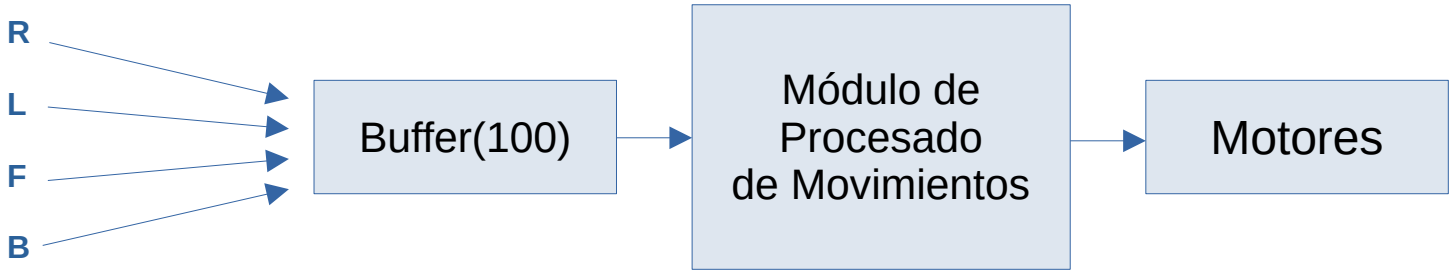
Monitores

- La idea de los monitores es separar estos dos ámbitos: se usan cerrojos para la exclusión mutua y variables condición para restringir la ejecución.
- La mayoría de los lenguajes modernos ya incluyen monitores como construcción sintáctica.
- Java se utiliza la palabra reservada *synchronized.*

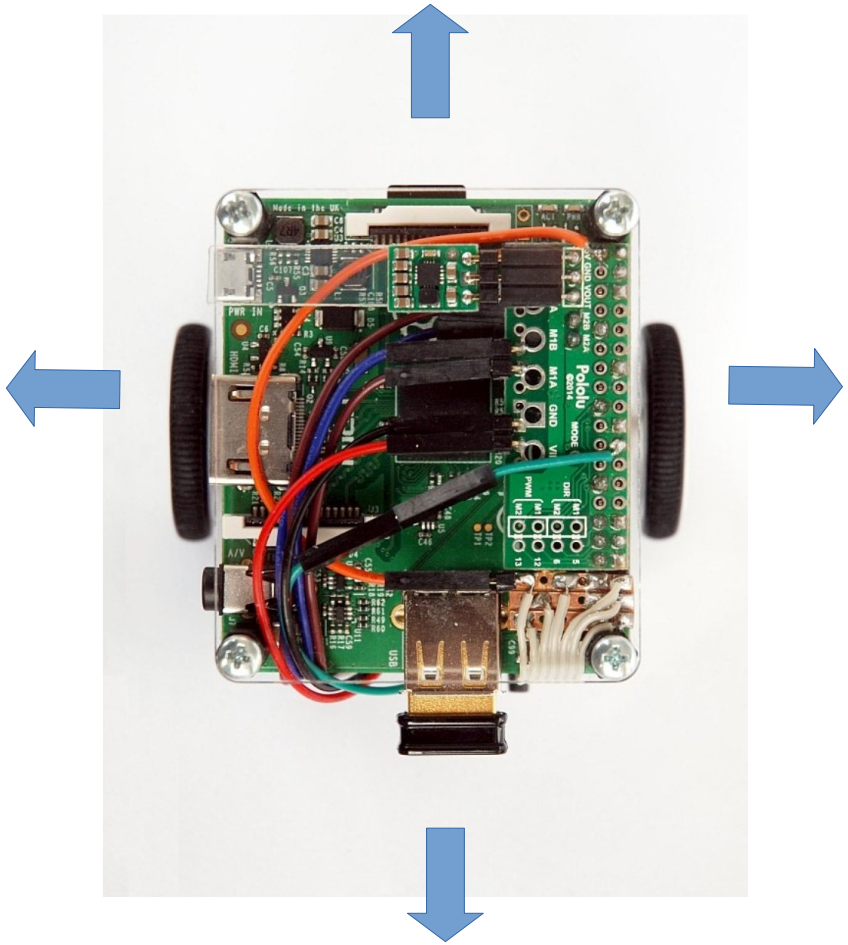


Monitores (ejemplo)

- Librería thread-safe para controlar un robot.

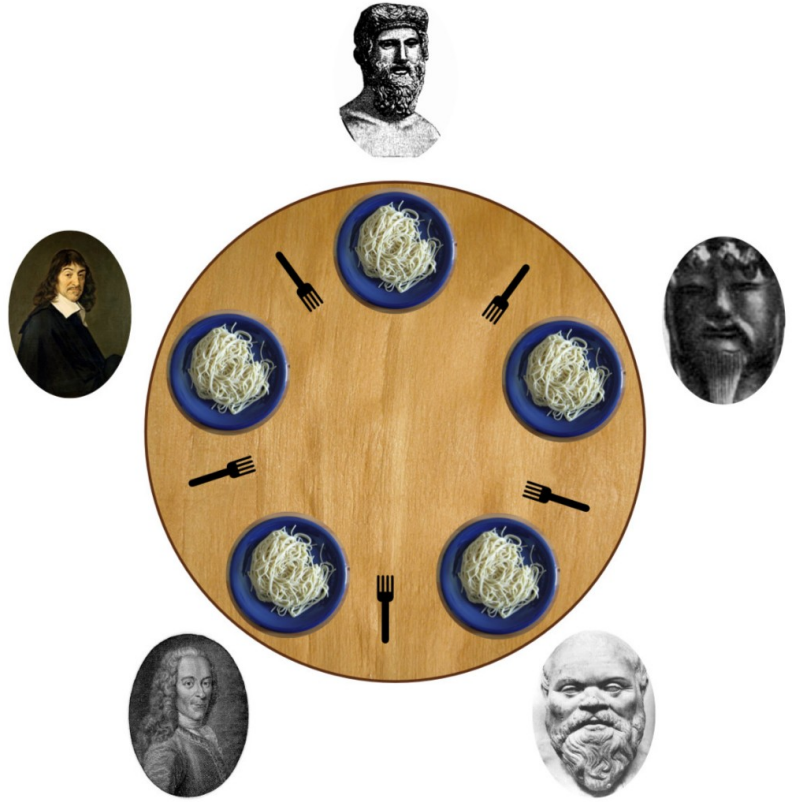


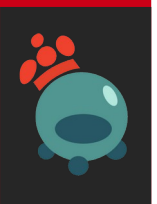

```
void init_module ();
void finish_module ();


// Las siguientes funciones son thread-safe
void move_robot_right();
void move_robot_left();
void move_robot_forward();
void move_robot_backward();

void print_commands();
```



El problema de los filósofos cenando

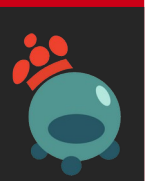
- Problema de sincronización propuesto por Dijkstra en 1965 para explicar los problemas de sincronización de procesos en un sistema operativo.
- Cinco filósofos se sientan alrededor de una mesa
- Cada filósofo tiene un plato de fideo y un tenedor a la izquierda de su pla
- Para comer los fideos son necesarios dos tenedores y cada filósofo sólo puede tomar los que están a su izquierda y derecha



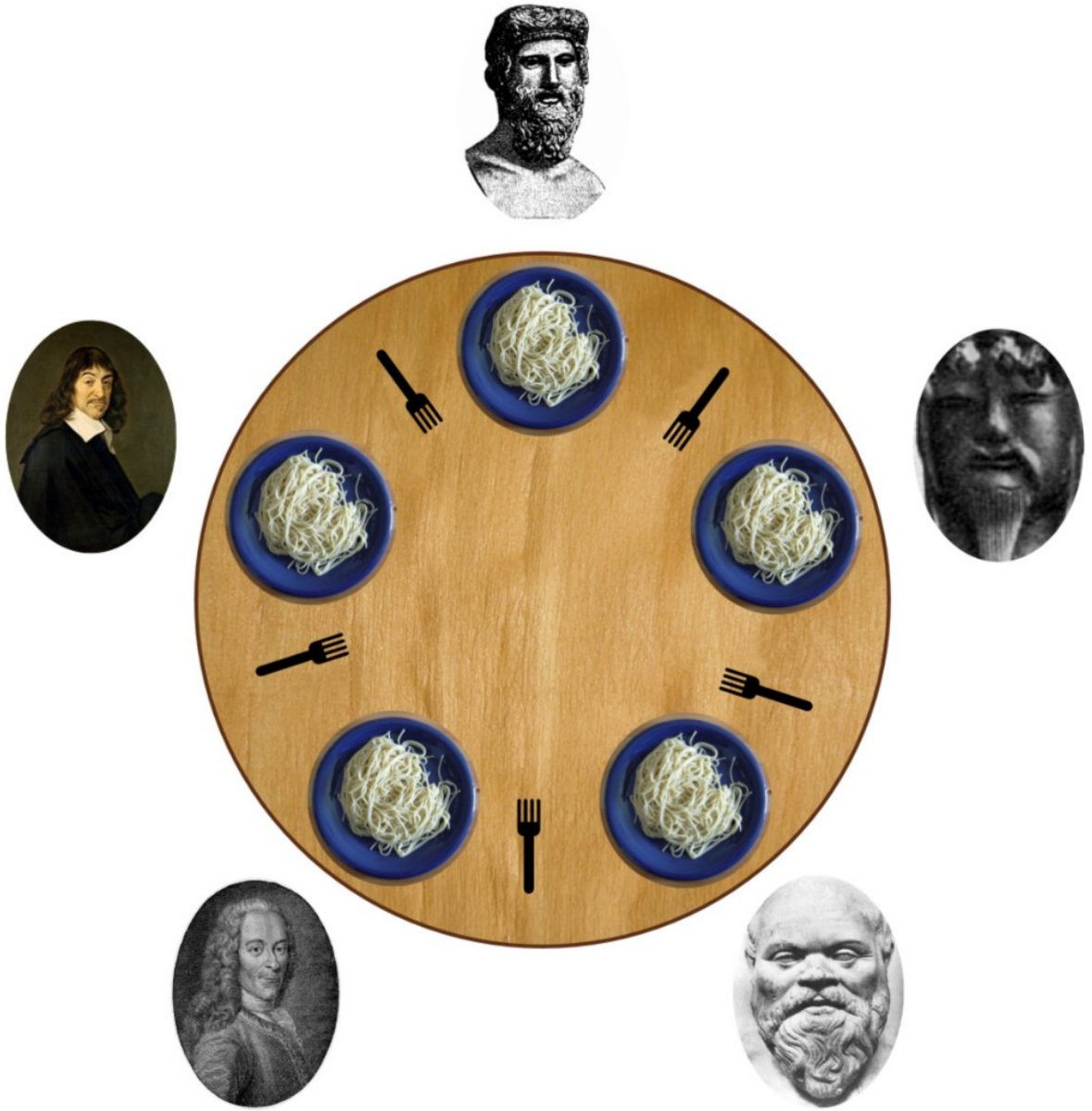


El problema de los filósofos cenando

- Se deben cumplir los siguientes requisitos
  - Un filósofo solo puede comer si tiene los dos tenedores.
  - Exclusión mutua, un tenedor solo puede ser usado por un filósofo a la vez.
  - Se debe asegurar progreso y espera limitada.
- Si todos los filósofos toman el tenedor que está a su derecha al mismo tiempo, entonces todos se quedarán esperando eternamente
- El problema consiste en **encontrar** un algoritmo que permita que los filósofos nunca se mueran de hambre.



Soluciones

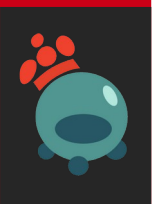


Extra

- **Barreras**
  - Elemento de sincronización que nos permite esperar a que un numero determinado de tareas finalicen.
  - Útil para sincronizar el comienzo de sub-tareas dentro del mismo thread.
  - Implementados en la librería pthread.

    - Inicializa la barrera con el número de tareas a esperar.
      - `pthread_barrier_init(&barrier, …, count)`

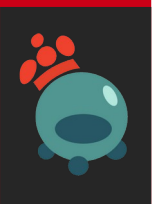
    - Cada tarea espera hasta que el resto haya terminado.
      - `pthread_barrier_wait(&barrier)`



Extra

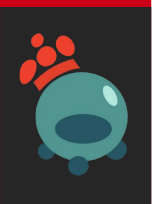
- **Futex** (fast user-space mutex)
  - Es utilizado en contextos de memoria compartida
  - Implementación eficiente y rápida en espacio de usuario.
  - Usados en la implementación de pthread_mutex

```
#include <linux/futex.h>
#include <stdint.h>
#include <sys/time.h>

long futex(uint32_t *uaddr, int futex_op, uint32_t val,
           const struct timespec *timeout,   /* or: uint32_t val2 */
           uint32_t *uaddr2, uint32_t val3);
```



Bibliografía

- GITLAB de la asignatura
  - https://gitlab.etsit.urjc.es/roberto.calvo/sdc
- Sistemas operativos distribuidos (Andrew S. Tanenbaum)
- Libro de Concurrencia Gallir.
  - https://barrgroup.com/embedded-systems/how-to/rtos-mutex-semaphore
- POSIX Semaphores in Linux
  - https://www.softprayog.in/programming/posix-semaphore
- Readers and Writers [Courtois et al., 1971]
  - https://dl.acm.org/doi/10.1145/362759.362813



URJC Escuela de Ingeniería de Fuenlabrada

GSyC

RoboticsLabURJC

Programming Robot Intelligence