Sincronización entre tareas.

Agenda.

- Planificación de procesos (repaso).
- Sincronización.
  - Serialización
  - Exclusión Mutua.
- Serialización. Ejecución concurrente y secuencial.
- No determinismo.
- Condiciones de Competencia.
- Zonas Críticas.
- Semáforos.
  - Definición.
  - Sintáxis en FreeRTOS.
  - ¿Por qué semáforos?
  - Exclusión mutua con semáforos.
- Sincronización de dos procesos.
  - Bloqueo Mutuo. Condiciones.
- Inanición
- Inversión de prioridad. Definición. ¿Cómo se da?
- Ejemplo. La cena de los filósofos.
- Semáforos en FreeRTOS.
- Colas de Mensajes.
- Patrones de sincronización. Productor consumidor.
  - Problema de distribución de datos 1 a N.
  - Solución con colas y memoria.
- Problema de Barrera.

Planificación de procesos.

- Uno de los componentes principales de un SO, es el planificador (scheduler).
- El planificador puede funcionar de diferentes maneras.
  - Cooperativo
  - Round-Robin.
  - Apropiativo.

Planificación entre procesos. Manejo del tiempo

```
void vUserTask1(void *pvParameters)
{
    portTickType xMeDesperte;
    xMeDesperte = xTaskGetTickCount();

    while(1)
    {
        if(isActivo(&ledStick))
        {
            Pasivar(&ledStick);
        }
        else
        {
            Activar(&ledStick);
        }
        vTaskDelayUntil(&xMeDesperte,125/portTICK_RATE_MS);
    }
}
```

- Al usar un RTOS (sistema operativo en tiempo real) El manejo del tiempo y de las tareas es una atribución del planificador ***solamente.***
- Las tareas deben tratar las cuestiones temporales a través de las funciones del RTOS.
- Esto se debe a que las tareas ***no tienen información*** temporal precisa (entre dos instrucciones sucesivas no saben cuantos cambios de tareas sucedieron.

Sincronización.

- En un sistema en el que existan dos o más tareas el planificador puede sacar a la misma, en principio en cualquier momento, por lo tanto una tarea no puede asumir que realiza una línea de código de manera completa. Esto da paso a los problemas de sincronización.
- Requerimientos de sincronización. Van a ser los concernientes al orden de los eventos a procesar.

Sincronización.

- Dos de los requerimientos de sincronización más comunes van a ser:
  - La serialización. Un evento A debe ocurrir antes de un evento B.
  - La mutua exclusión. Un evento A y otro B no deben ocurrir al mismo tiempo.

Serialización. Ejemplificando

- Suponiendo que dos personas (A y B) van a sincronizarse para almorzar. La persona A va a almorzar antes que B (Serialización).

Serialización. Ejemplo.

1. **La persona A se despierta.**
2. **Desayuna.**
3. **Trabaja**
4. **Almuerza**
5. **Llama a la persona B**

1. **La persona B se despierta.**
2. **Desayuna**
3. **Espera el llamado de la persona A.**
4. **Almuerza.**

- De este ejemplo, se ve que sólo se puede asegurar que la persona B almuerza después de la persona A (ejecución secuencial).
- Por otro lado, no se puede asegurar quién desayuna primero de los dos. (ejecución concurrente).

Ejecución secuencial y concurrente.

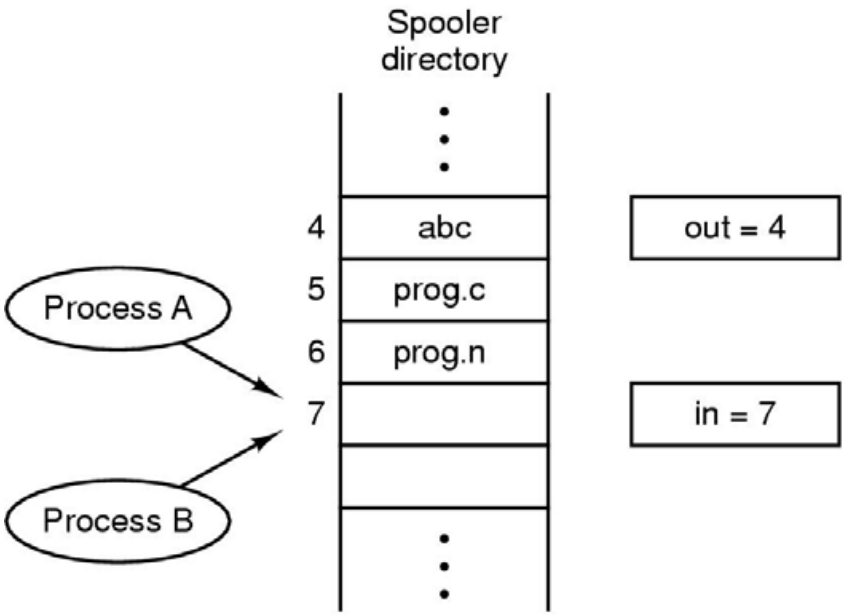
- Se va a definir la ejecución como ***secuencial*** ***cuando se*** ***conoce*** ***que evento sucede antes*** ***de que otro evento***.
- Cuando ***no se puede definir*** ***que evento sucede primero*** se dice que la ejecución va a ser ***concurrente***.

Ejecución no determinística.

- Se puede definir la ejecución como no determinística cuando no es posible, al observar el programa, determinar como se va a ejecutar (la ejecución concurrente va a ser no determinística).
- ¿Cuánto vale contador, si ambas tareas tiene la misma prioridad luego de un segundo?

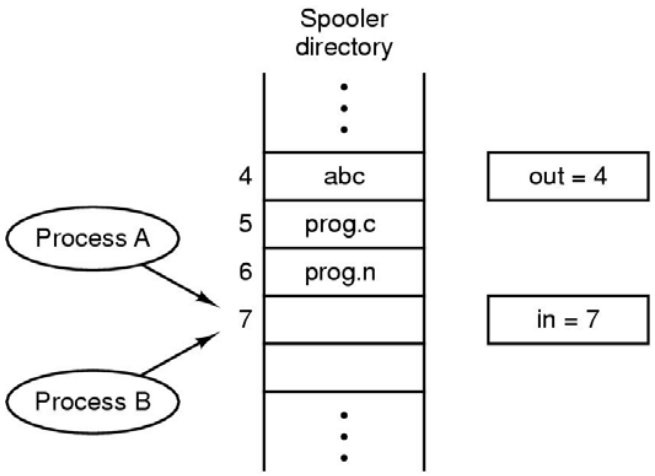
```
int contador = 0;

void TareaArriba(void *pvParameters)
{
    while(1)
    {
        contador++;
    }
}

void TareaAbajo(void *pvParameters)
{
    while(1)
    {
        contador--;
    }
}
```

Otro Problema. Tareas que comparten memoria.



- Existen dos tareas que escriben el nombre de un archivo imprimir en cada una de las posiciones y otra tarea que maneja el puntero de salida y retira el nombre del archivo para imprimirlo.

Tareas compartiendo memoria.



- Suponiendo que la tarea A se está ejecutando, lee el valor del puntero de entrada (7) y luego el planificador para a la tarea B.
- La tarea B también lee el valor del puntero de entrada (7), copia el nombre del archivo en la entrada número 7 y se bloquea.
- La tarea A vuelve donde estaba, (había leído que la posición era la 7. Escribe el valor de su archivo e incrementa el puntero a 8 y se bloquea.
- Como resultado se va a obtener que la tarea B jamás va a generar su salida. Esto se debe a que el resultado depende del orden de ejecución de las tareas, esto se conoce como ***condición de competencia.***

¿Cómo evitar las condiciones de competencia?

- Para evitar las condiciones de competencia es necesario prohibir que más de una tarea lea y escriba los datos compartidos a la vez. Se necesita usar ***exclusión mutua.***
- Las partes del programa que acceden a la memoria compartida se las llaman ***regiones críticas o secciones críticas.***

Condiciones de Competencia

- Para evitar las condiciones se deben cumplir las siguientes cuatro condiciones:
  - Dos procesos nunca pueden estar simultáneamente dentro de sus regiones críticas.
  - No puede suponerse nada acerca de las velocidades o la cantidad de procesadores.
  - Ninguna tarea que se ejecute fuera de su región crítica puede bloquear a otros procesos.
  - Ningún proceso deberá tener que esperar indefinidamente para entrar en su región crítica.

Cómo logro la generación de regiones críticas.

- Inhabilitar Interrupciones.
  - Es la solución más simple. Se deshabilitan las interrupciones (no puede volver a ejecutarse el planificador hasta que vuelvan habilitarse las mismas) al entrar en la región crítica y se vuelven a habilitar al terminar. El problema más grave es que si el proceso no vuelve a habilitar las interrupciones “mata” al planificador.
- Uso de instrucciones especiales.
  - Diferentes arquitecturas tienen instrucciones de assembler especiales (LDREX, STREX) que leen y escriben en memoria de manera atómica (sin poder ser interrumpidas).
- Uso de primitivas de alto nivel (semáforos, colas de mensajes).

Semáforos.

- En la vida diaria, los semáforos son sistemas de señalización que utilizan banderas, luces u otro mecanismo. Mientras que en el ámbito de los sistemas operativos, los semáforos van a ser estructuras de datos que se van a utilizar para resolver problemas de sincronización.
- Los semáforos fueron inventados por Edsger Dijkstra en 1965.

Semáforos. Definición.

- Un semáforo es como un entero pero con las siguientes diferencias:
  - Cuando se crea un semáforo, se puede inicializar con cualquier valor, luego sólo se lo puede incrementar en uno o decrementar en uno. Sin poder leer su valor.
  - Cuando una tarea decrementa el semáforo y el resultado resulta negativo, la tarea se bloquea hasta que otra tarea incremente al semáforo.
  - Cuando una tarea incrementa el semáforo, si hay otra tarea esperando por este, la misma se desbloquea.
  - Las operaciones de incrementar y decrementar los semáforos son atómicas.
  - En general, no hay forma de saber antes de decrementar si la tarea se va a bloquear o no.

Semáforos en FreeRTOS.

- Las funciones para crear semáforos son:
  - ***xSemaphoreHandle*** xSemaphoreCreateRecursiveMutex(***void***)
  - ***xSemaphoreHandle*** xSemaphoreCreateMutex(***void***)
  - vSemaphoreCreateBinary(***xSemaphoreHandle*** xSemaphore)
  - ***xSemaphoreHandle*** xSemaphoreCreateCounting

    (

    unsigned portBASE_TYPE uxMaxCount,

    unsigned portBASE_TYPE uxInitialCount

)

Semáforos en FreeRTOS (2)

- Funciones para decrementar “tomar” semáforos.
  - xSemaphoreTake(

    xSemaphoreHandle xSemaphore,
    portTickType xBlockTime
)

    el semaforo se bloquea cuando llega a 0
  - xSemaphoreTakeRecursive(

    xSemaphoreHandle xMutex,
    portTickType xBlockTime
)

Semáforos en FreeRTOS (3)

- Funciones para incrementar “liberar” semáforos:
  - xSemaphoreGive(***xSemaphoreHandle*** xSemaphore)
  - xSemaphoreGiveRecursive(***xSemaphoreHandle*** xMutex)
  - xSemaphoreGiveFromISR
    (
    xSemaphoreHandle xSemaphore,
    ***signed portBASE_TYPE ****pxHigherPriorityTaskWoken
)

    si el semáforo estaba en 0 y una tarea lo incrementa, el semáforo se libera.

¿Por qué usar semáforos?

- Es una solución bastante independiente de la arquitectura.
- Es una solución de relativo alto nivel y no atenta contra la prioridad del RTOS (ya que la tarea que lo llama no toca las interrupciones).
- Imponen una serie de restricciones (sólo incrementar o decrementar) que ayudan a no generar errores.

Creando una exclusión mutua con semáforos.

A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks. The word MUTEX originates from 'MUTual EXclusion

```
xSemaphoreHandle mutuaExclusion;

int main(void)
{
    setupHardware();
    mutuaExclusion = xSemaphoreCreateMutex();
    if(mutuaExclusion)
    {
        xTaskCreate(.........);
        xTaskCreate(.........);
        xTaskCreate(.........);
        xTaskCreate(.........);

        vTaskStartScheduler();
    }
    return 1;
}
```

Usando la exclusión mutua.

```
void TareaLed(void *pvParameters)
{
    while(1)
    {
        haceralgo();
        ....
        ....
        if(xSemaphoreTake(mutuaExclusion, portMAX_DELAY) == pdTRUE)
        {
            //Entro en la zona crítico
            hagoalgocritico();
            ....
            ....
            ....
            xSemaphoreGive(mutuaExclusion);
            //Salí de la zona de mutua exclusión.
        }
    }
}
```

Sincronizando dos procesos.

```
void TareaA(void *pvParameters)
{
    while(1)
    {
        funcionA1();
        xSemaphoreGive(semaforoA);
        if(xSemaphoreTake(semaforoB,portMAX_DELAY) == pdTRUE)
        {
            funcionA2();
        }
    }
}
void TareaB(void *pvParameters)
{
    while(1)
    {
        funcionB1();
        xSemaphoreGive(semaforoB);
        if(xSemaphoreTake(semaforoA,portMAX_DELAY) == pdTRUE)
        {
            funcionB2();
        }
    }
}
```

Interbloqueo.

```
void TareaA(void *pvParameters)
{
    while(1)
    {
        xSemaphoreTake(semaforoA,portMAX_DELAY);
        xSemaphoreTake(semaforoB,portMAX_DELAY);
        a = b;
        xSemaphoreGive(semaforoB);
        xSemaphoreGive(semaforoA);
        }
    }
}
```

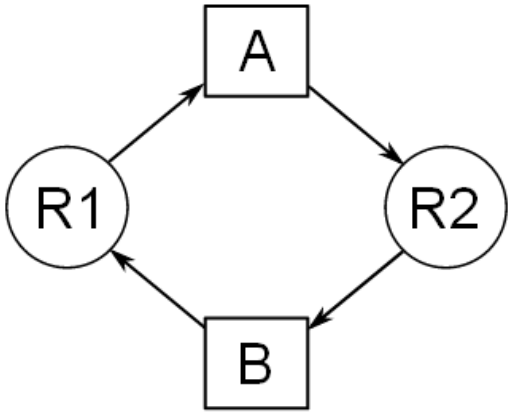
```
void TareaB(void *pvParameters)
{
    while(1)
    {
        xSemaphoreTake(semaforoB,portMAX_DELAY);
        xSemaphoreTake(semaforoA,portMAX_DELAY);
        b = a;
        xSemaphoreGive(semaforoA);
        xSemaphoreGive(semaforoB);
        }
    }
}
```

Interbloqueo.

- En el caso anterior se supone que empieza a ejecutar la tareaA, toma el primer semáforo y se produce el cambio de contexto a la tarea B.
- La tarea B toma el semáforo B y se bloquea (ya que el semáforo A está tomado por la tarea A).
- Se vuelve a la tarea A y no puede desbloquearse ya que B tiene recursos que ella necesita bloqueados y viceversa, esta situación se la denomina ***interbloqueo.***

Condiciones para Interbloqueos (Condiciones de Coffman).

- Dadas las tareas T1, T2, .. TN y los recursos R1, R2, … RM.
  - ***Exclusión Mutua***. existencia de al menos de un recurso compartido por los procesos, al cual sólo puede acceder uno simultáneamente.
  - **Condición de retención y espera**: al menos un proceso *Pi* ha adquirido un recurso *Ri*, y lo retiene mientras espera al menos un recurso *Rj* que ya ha sido asignado a otro proceso
  - **Condición de no expropiación**: los recursos no pueden ser expropiados por los procesos, es decir, los recursos sólo podrán ser liberados voluntariamente por sus propietarios.
  - **Condición de espera circular**: dado el conjunto de procesos *P0...Pm*(subconjunto del total de procesos original),*P0* está esperando un recurso adquirido por *P1*, que está esperando un recurso adquirido por *P2*,... ,que está esperando un recurso adquirido por *Pm*, que está esperando un recurso adquirido por *P0*. Esta condición implica la condición de retención y espera



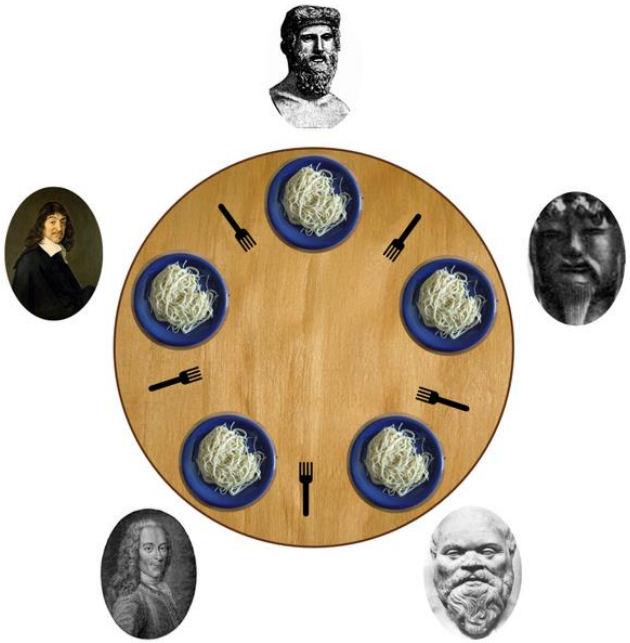
Inanición.

- La inanición de una tarea es cuando a esta se le niega siempre un recurso del sistema. Nunca puede terminar de hacer su tarea y “muere de hambre” a la espera del recurso.
- La inanición es una situación similar al interbloqueo, pero las causas son diferentes. En el interbloqueo, dos tareas en de ejecución llegan a un punto muerto cuando cada uno de ellos necesita un recurso que es ocupado por el otro. En cambio, en este caso, uno o más procesos están esperando recursos ocupados por otros procesos que no se encuentran necesariamente en ningún punto muerto.

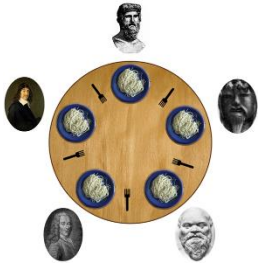
Inversión de prioridades

- La inversión de prioridades se da cuando dos tareas de ***distinta prioridad comparten un recurso*** y la tarea de menor prioridad bloquea el recurso antes que la de prioridad mayor, quedando bloqueada esta última tarea en el momento que precise el uso del recurso compartido.
- Esto hace que queden invertidas de forma efectiva las prioridades relativas entre ambas ya que la tarea que originalmente tenía mayor prioridad queda supeditada a la tarea de menor prioridad.

La cena de los filósofos.



- La cena de los filósofos es un problema clásico de sincronización entre tareas.
- El problema asume que los filósofos alternan períodos aleatorios de comer y pensar.
- Cuando están en condiciones de comer necesitan dos tenedores para comer, luego de comer los liberan y vuelven a pensar.



Problemas.

- Si todos los filósofos toman un tenedor a la vez se llega a una condición de ***interbloqueo***, ya que todos tienen un recurso que otros necesitan.
- Una posible solución sería, que el filósofo tome un tenedor y espere un tiempo por el otro. Con algo de mala suerte, puede darse que todos tomen y suelten los tenedores a la vez, teniendo un problema de ***inanición***.
- Otra posible solución es generar los tiempos de retención de tenedores ***aleatorios***, reduciendo la probabilidad de inanición. No todas las aplicaciones permiten este comportamiento (por ej. Sistema de control de una central nuclear).
- También se podría hacer que siempre como un solo filósofo, independientemente de que se pueda alimentar a otro, con lo que se hace una ***subutilización de recursos.***
- La idea de este problema es mostrar la complejidad que puede alcanzar un problema de programación concurrente.

Solución al problema con semáforos.

```
#define N           5
#define IZQ         ((i-1)%N)
#define DER         ((i+1)%N)
#define PENSANDO    0
#define HAMBRIENTO  1
#define COMIENDO    2

int estado[N];
xSemaphoreHandle mutex;
xSemaphoreHandle s[N];
```

```
void filosofo(void *pvParameters)
{
    while(1)
    {
        pensar();
        tomar_tenedores();
        comer();
        devolver_tenedores();
    }
}
```

Solución al problema con semáforos (2).

```
void tomar_tenedores(int i)
{
    xSemaphoreTake(mutex, portMAX_DELAY);
    estado[i] = HAMBRIENTO;
    chequear(i);
    xSemaphoreGive(mutex, portMAX_DELAY);
    xSemaphoreTake(s[i], portMAX_DELAY);
}
```

```
void devolver_tenedores(int i)
{
    xSemaphoreTake(mutex, portMAX_DELAY);
    estado[i] = PENSANDO;
    chequear(IZQ);
    chequear(DER);
    xSemaphoreGive(mutex, portMAX_DELAY);
}
```

Solución al problema con semáforos (3).

```
void chequear(int i)
{
    if( (estado[i]==HAMBRIENTO) &&
        (estado[IZQ]!=COMIENDO) &&
        (estado[DER]!=COMIENDO))
    {
        estado[i] = COMIENDO;
        xSemaphoreGive(s[i], portMAX_DELAY);
    }
}
```

Colas de mensajes.

- Las colas de mensajes son un mecanismo de comunicación entre procesos. A diferencia de los semáforos, las colas de mensajes permiten enviar datos directamente entre tareas.
- En FreeRTOS son el mecanismo para comunicación entre:
  - Tareas con Tareas.
  - Tarea a interrupción
  - Interrupción a tarea.

Características de las colas de mensajes.

- Almacenamiento de datos. Una cola puede almacenar un número finito de elementos de tamaño fijo.
- Las colas de mensajes de FreeRTOS van a utilizar normalmente como colas “primero en entrar – primero en salir” (FIFO).
- Escribir en una cola va a generar una copia byte a byte del mensaje transferido a los datos de la cola.
- Acceso por muchas tareas. Las colas de mensajes pueden ser escritas por muchas tareas y leídas por varias tareas también. El uso más común es el que varias tareas escriben sobre una cola y una única tarea lee los datos (registro o logging).

Lectura de una cola de mensajes.

- Cuando se intenta leer una cola, esta va a bloquear la tarea que la trata de leer hasta que haya al menos un dato válido.
- Si hay varias tareas esperando leer datos de una cola de mensajes, se va a desbloquear la tarea de mayor prioridad. En el caso de que tengan la misma prioridad se desbloquea la que lleve mayor tiempo bloqueada.

Escritura de colas de mensajes.

- Una tarea que intenta escribir una cola de mensajes se va a bloquear si la cola está llena.
- Si hay múltiples tareas tratando de escribir una cola bloqueada va a entrar primero la de mayor prioridad. Si son de igual prioridad la que lleve mayor tiempo bloqueada.

Creando una cola de mensajes en FreeRTOS.

xQueueHandle xQueueCreate

(

unsigned portBASE_TYPE uxQueueLength,

unsigned portBASE_TYPE uxItemSize

);

Poniendo datos en una cola de mensajes. FreeRTOS.

- ***portBASE_TYPE*** xQueueSend(
  xQueueHandle xQueue,
  ***const void ** ** pvItemToQueue,
  portTickType xTicksToWait
  );
- ***portBASE_TYPE*** xQueueSendToBack(
  xQueueHandle xQueue,
  ***const void ** ** pvItemToQueue,
  portTickType xTicksToWait
  );
- ***portBASE_TYPE*** xQueueSendToFront(
  xQueueHandle xQueue,
  ***const void ** ** pvItemToQueue,
  portTickType xTicksToWait
  );

Sacando datos de las colas de mensajes en FreeRTOS.

- ***portBASE_TYPE*** xQueueReceive(

  xQueueHandle xQueue,
  ***void ****pvBuffer,
  portTickType xTicksToWait

  );

- ***portBASE_TYPE*** xQueuePeek(

  xQueueHandle xQueue,
  ***void ****pvBuffer,
  portTickType xTicksToWait

  );

Otras funciones de colas de mensajes.

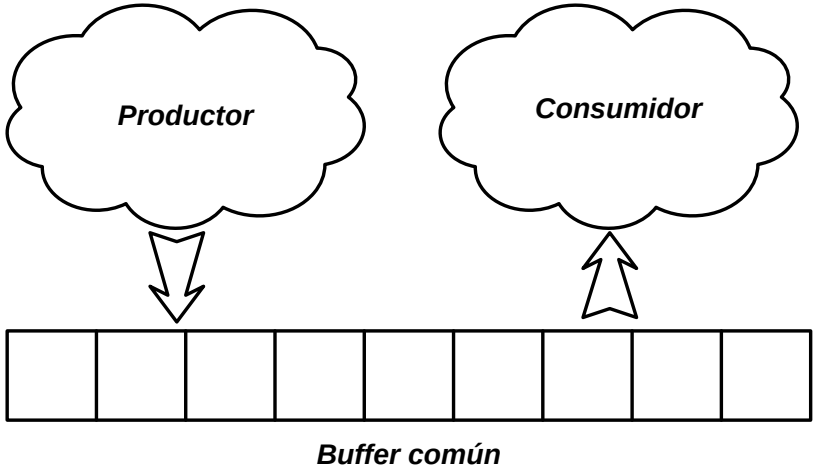
- ***unsigned portBASE_TYPE*** uxQueueMessagesWaiting (***xQueueHandle*** xQueue);
- ***void*** vQueueDelete(***xQueueHandle*** xQueue);
- ***portBASE_TYPE*** xQueueReset(***xQueueHandle*** xQueue);

Ejemplo cola de mensajes

```
/*  Fragmento de la tarea que
    escribe datos en una cola   */
for(;;)
{
    datos = generarDatos();
    xQueueSend(Cola,&datos,portMAX_DELAY);
    taskYIELD();
}
```

```
/*  Fragmento de la tarea que
    escribe datos en una cola   */
for(;;)
{
    xQueueReceive(Cola,&datos,portMAX_DELAY);
    consumir(&datos);
    taskYIELD();
}
```

Patrones de Sincronización. Productor - Consumidor



- El problema del productor consumidor es un problema donde una tarea intenta escribir datos en un buffer común (productor) y otra tarea retira los datos de este buffer común (consumidor).

Productor – Consumidor. Condiciones.

- El productor se bloqueará si el buffer compartido se llena.
- El consumidor se bloqueará esperando hasta que haya al menos un dato disponible.
- Tener en cuenta que no se puede suponer en que orden se van a ejecutar las tareas ni cuanto tiempo tarda en generar o consumir un dato

Inicialización.

```
#define N   100
int buffer[N];
int indice = 0;
xSemaphoreHandle mutex;
xSemaphoreHandle vacios;
xSemaphoreHandle llenos;

void InicializaPC(void)
{
     mutex  =  xSemaphoreCreateMutex();
     vacios = xSemaphoreCreateCounting(N,N);
     llenos = xSemaphoreCreateCounting(N,0);

     /*Crear tareas Productor y consumidor*/
     .....
     .....

}
```

Código del productor.

```
void Productor(void *parametros)
{
    int dato;

    for(;;)
    {
        dato = generarDato();
        xSemaphoreTake(vacios,portMAX_DELAY);
        xSemaphoreTake(mutex,portMAX_DELAY);
        buffer[indice++]=dato;
        xSemaphoreGive(mutex);
        xSemaphoreGive(llenos);
    }
}
```

Código del consumidor.

```
void Consumidor(void *parametros)
{
     int dato;

     for(;;)
     {
          xSemaphoreTake(llenos,portMAX_DELAY);
          xSemaphoreTake(mutex,portMAX_DELAY);
          dato = buffer[--indice];
          xSemaphoreGive(mutex);
          xSemaphoreGive(vacios);
          consumir(dato);
     }
}
```

Alcance de la solución.

- Esta implementación del algoritmo del productor consumidor, permite la implementación de n consumidores y m productores.
- Los consumidores “compiten” por los datos (el dato que usa un consumidor, no lo pueden acceder otros).
- Este patrón de sincronización es una muy buena solución para N productores y 1 consumidor.

Productor Consumidor con colas de mensajes (inicialización).

```
#define NPRODUCTORES    2
#define NCONSUMIDORES   3
#define LENCOLA         8

xQueueHandle    qproductor;
xQueueHandle    qconsumidor;

void Inicializar()
{
    qproductor = xQueueCreate(LENCOLA,sizeof(int));
    qconsumidor = xQueueCreate(LENCOLA,sizeof(int));

    for(i=0;i<LENCOLA; i++)
    {
        xQueueSend(qproductor,&i,0);
    }
}
```

Código del productor.

```
void Productor(void *parametros)
{
     int dato;
     int dummy;
     for(;;)
     {
          dato = generar();
          xQueueReceive(qproductor,&dummy,portMAX_DELAY);
          xQueueSend(qconsumidor,&dato,portMAX_DELAY);
     }
}
```

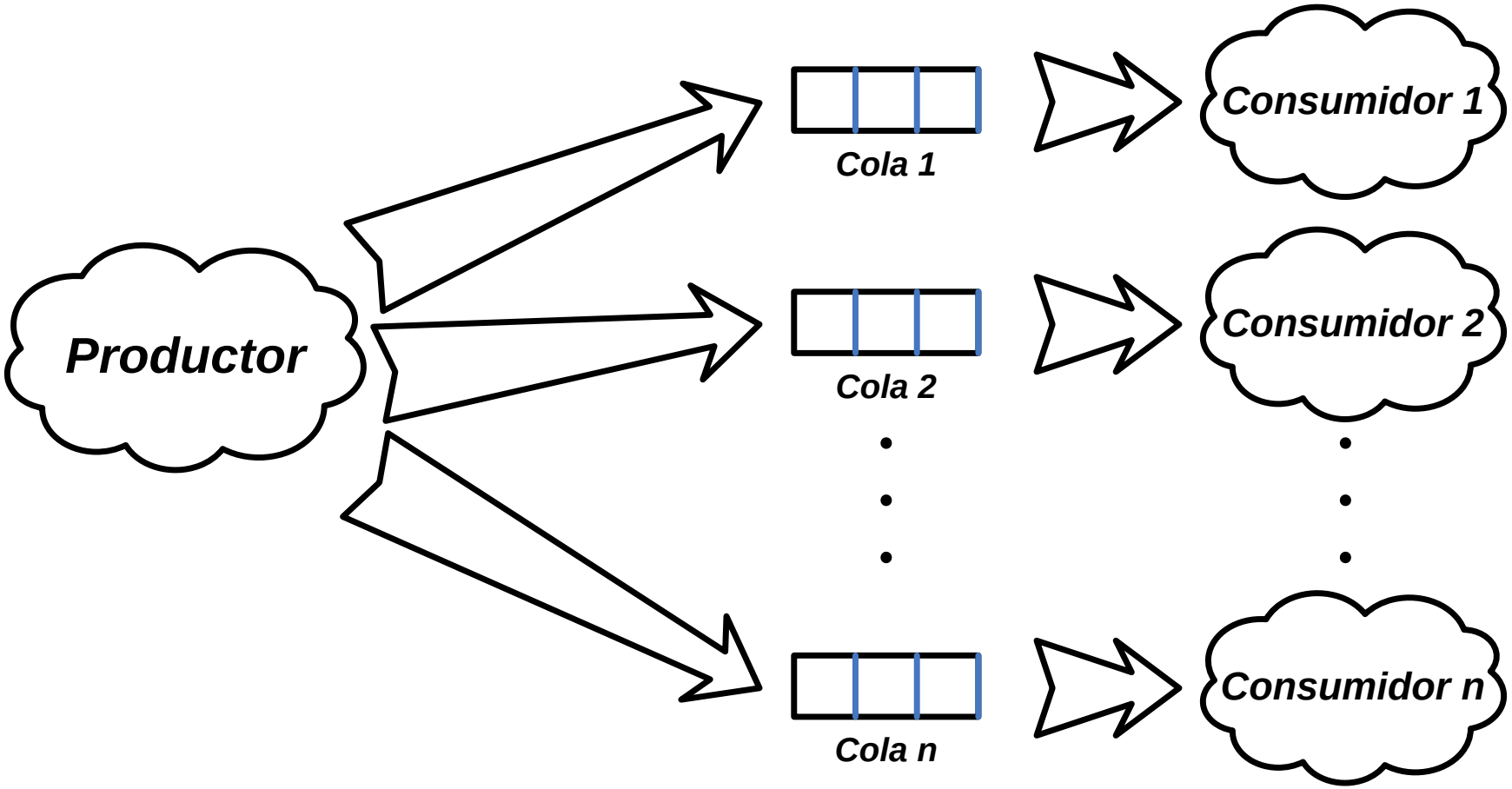
Código del consumidor.

```
void Consumidor(void *parametros)
{
     int dato;
     int dummy = 0;
     for(;;)
     {
          xQueueReceive(qconsumidor,&dato,portMAX_DELAY);
          xQueueSend(qproductor,&dummy,portMAX_DELAY);
          consumir(&dato);

     }
}
```

Problema de distribución de datos.

- El patrón de productor consumidor resuelve la situación del registro de datos (muchas fuentes de datos y un único registrador).
- El patrón de distribución de dato maneja el problema opuesto se genera un dato y lo tienen que consumir (sin competir por él) varias tareas (ej: se toma un dato del ADC y se muestra en un display, se graba en memoria y se envía por línea serie.

Solución para distribuir datos con mucha memoria.



Problema de barrera

- Este tipo de problema trata de que un número de tareas se ejecuten todos antes de que se vuelva a repetir la ejecución de una tarea.

Inicialización

```
#define NPROC 4

int n;
xSemaphoreHandle mutex;
xSemaphoreHandle torniquete;
xSemaphoreHandle torniquete2;

void Inicialización
{
     n = 0;
     mutex = xSemaphoreCreateMutex();
     vSemaphoreCreateBinary(torniquete);
     vSemaphoreCreateBinary(torniquete2);
     xSemaphoreTake(torniquete,portMAX_DELAY);
}
```

Esquema de la Tarea.

```
void Sincronizado(void *parametros)
{
    for(;;)
    {
        xSemaphoreTake(mutex,portMAX_DELAY);
        n++;
        if(n==NPROC)
        {
            xSemaphoreTake(torniquete2,portMAX_DELAY);
            xSemaphoreGive(torniquete,portMAX_DELAY);
        }
        xSemaphoreGive(mutex);
        xSemaphoreTake(torniquete,portMAX_DELAY);
        xSemaphoreGive(torniquete,portMAX_DELAY);
```

Esquema de la Tarea (2).

```
            xSemaphoreTake(mutex,portMAX_DELAY);
            n--;
            if(n==0)
            {
                xSemaphoreTake(torniquete,portMAX_DELAY);
                xSemaphoreGive(torniquete2,portMAX_DELAY);
            }
            xSemaphoreGive(mutex);
            xSemaphoreTake(torniquete2,portMAX_DELAY);
            xSemaphoreGive(torniquete2,portMAX_DELAY);
            //Hacer tarea sincronizada
        }
    }
```

Bibliografía.

- Sistemas Operativos. Diseño e Implementación. Andrew S. Tanenbaum.
- Sistemas operativos modernos. Andrew Tanembaum. Segunda Edición.
- Using the FreeRTOS Real Time Kernel. NXP LPC17xx Edition. Richard Barry.
- FreeRTOS http://www.freertos.org/
- The Little Book of Semaphores. Allen B. Downey. http://www.greenteapress.com/semaphores/