Sincronización en el kernel Linux

LIN - Curso 2016-2017

J.C. Sáez
High Performance Computing Division ArTeCs Group
Universidad Complutense de Madrid

Contenido

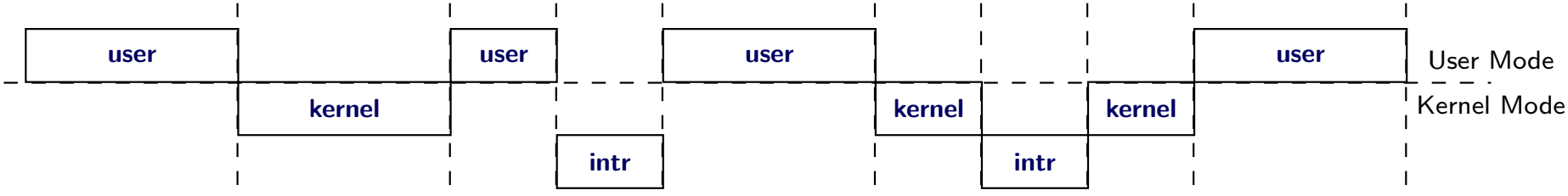
Contenido

Flujos de ejecución en el kernel

- Las funciones del kernel Linux se invocan en respuesta a *eventos*
  - El kernel se comporta como un "servidor" que responde peticiones
    1. Llamadas al sistema
    2. Excepciones de los programas de usuario
    3. Interrupciones de los dispositivos de E/S
- **Flujo de ejecución del kernel**: secuencia de instrucciones que se ejecutan en modo kernel
  - Más ligero que un proceso (menos contexto)
  - No siempre desempeña acciones en nombre de un proceso de usuario

Ejecución en una CPU

- En un instante *t* cada CPU del sistema puede estar ejecutando:
  - Código de un proceso en modo usuario (**Contexto de Proceso en modo usuario**)
  - Código del kernel en una llamada al sistema o manejador de excepción (**Contexto de Proceso en modo kernel**)
  - Código de un *kernel thread* (**Contexto de Proceso en modo kernel**)
  - Código que realiza procesamiento ligado a una interrupción (**Contexto de Interrupción**)



Recuerda: Expropiación de usuario

- El SO puede *expulsar* a un proceso de la CPU mientras se ejecuta en modo usuario y poner a otro proceso en su lugar
  - Planificadores expropiativos: RR, Colas multinivel, CFS, ...
- La expropiación de usuario requiere soporte HW
  - El SO configura el *temporizador del sistema* para que genere interrupciones periódicas (*tick*)
  - Interrupcción del *timer* $\rightarrow$ transición a modo kernel $\rightarrow$ activación del planificador (`scheduler_tick()`)
  - Al retornar a modo usuario tras la interrupción se podría ejecutar otro proceso

Expropiación de kernel

- El kernel Linux es **expropiativo** (*preemptive*)
  - Cualquier proceso ejecutándose en modo kernel (contexto de proceso en modo kernel) podría ser reemplazado por otro proceso *casi* en cualquier instante
    - Por ejemplo, expropiación durante la ejecución de una llamada al sistema (*handler function*)
- **Objetivo expropiación**: reducir el tiempo de respuesta (latencia) de los procesos de usuario
  - **Latencia:** tiempo transcurrido desde que un proceso entra en el estado "listo para ejecutar" hasta que se pone a ejecutar en una CPU
- Durante la ejecución de una llamada al sistema se puede producir expropiación en el kernel siempre y cuando no se haya deshabilitado explícitamente la expropiación
  - `preempt_disable()`, `preempt_enable()`

Origen de la concurrencia en Linux

- Situaciones que provocan la ejecución de flujos concurrentes/entrelazados en el kernel:
  1. **Interrupciones**
     - Una interupción puede ocurrir de forma asíncrona en casi cualquier instante, interrumpiendo al código que está actualmente en ejecución
  2. **Bottom-half processing**
  3. **Expropiación de kernel**
     - Como el kernel es expropiativo, un flujo de ejecución del kernel puede expropiar a otro
  4. **Bloqueos y sincronización con el modo usuario**
     - Un proceso que se ejecuta en modo kernel puede bloquearse y por tanto invocar al planificador (`schedule()`) para permitir la ejecución de otro proceso
  5. **Multiprocesamiento simétrico (SMP)**
     - Dos flujos de ejecución del kernel podrían ejecutarse al mismo tiempo en distintas CPUs

Situaciones que requieren sincronización

- Una *condición de carrera* puede ocurrir cuando el resultado de un cómputo/procesamiento depende de cómo se *entrelace* la ejecución de instrucciones de dos o más flujos de ejecución del kernel
- Para evitar condiciones de carrera debe garantizarse **exclusión mutua** entre secciones críticas (SCs)
  - En sistemas monoprocesador, desactivar interrupcciones dentro de la SC garantiza exclusión mutua
  - En sistemas multiprocesador/multicore es más complicado garantizar exclusión mutua
    - Necesario usar primitivas de sincronización

Contenido

Primitivas de sincronización en Linux

Técnica	Descripción	Ámbito
Deshabilitar localmente las interrupciones	Ignorar temporalmente las interrupciones en una CPU	Local
Operaciones Atómicas	Instrucciones RMW (read-modify-write) atómicas	Global
Variables *Per-CPU*	Replicar una estructura de datos en distintas CPUs	Global
Barreras de memoria	Evitar el reordenamiento de instrucciones	Local
Spin locks	Cerrojos de espera activa	Global
Semáforos	Mecanismo de sincronización de espera bloqueante	Global
Read-copy-update (RCU)	Acceso sin cerrojos a datos compartidos vía punteros	Global
Seqlocks	Cerrojo basado en contador de acceso	Global
Variables `completion`	Colas de espera	Global

Deshabilitar interrupciones

Exclusión mutua en UP

```
local_irq_disable();
.. Sección crítica ..
local_irq_enable();
```

Exclusión mutua en UP (seguro)

```
unsigned long flags=0;
local_irq_save(flags);
.. Sección crítica ..
local_irq_restore(flags);
```

Operaciones <linux/irqflags.h>

- `local_irq_disable()`: Deshabilita las interrupciones en la CPU local
- `local_irq_enable()`: Habilita las interrupciones en la CPU local
- `local_irq_save(flags)`: Almacena en `flags` el estado actual de las interrupciones y deshabilita las interrupciones en la CPU local
- `local_irq_restore(flags)`: Restaura el antiguo estado de las interrupciones almacenado en `flags`

Operaciones atómicas sobre enteros

- Tipo `atomic_t`: contador atómico de 32 bits
  - Uso en fragmentos de código del kernel que acceden concurrentemente a una variable entera compartida
  - `atomic_t` soporta conjunto de operaciones atómicas
    - Declarado en `<asm/atomic.h>`
    - Implementación dependiente de arquitectura mediante instrucciones RMW (*read-modify-write*)
- Tipo `atomic64_t`: contador atómico de 64 bits
  - Uso análogo a `atomic_t`
  - Declarado en `<asm/atomic64.h>`

Ejemplo: Operaciones atómicas sobre enteros

```
int contador=0;
```

CPU 0	CPU 1
`contador++;`	`contador+=2;`

Ejemplo: Operaciones atómicas sobre enteros

```
int contador=0;
```

CPU 0	CPU 1
`contador++;`	`contador+=2;`

```
atomic_t contador=ATOMIC_INIT(0);
```

CPU 0	CPU 1
`atomic_inc(&contador);`	`atomic_add(2,&contador);`

Operaciones atómicas: `atomic_t`

Función/Macro	Descripción
`ATOMIC_INIT(i)`	En declaración, inicializa el contador a `i`
`atomic_read(v)`[1]	Devuelve el valor entero asociado
`atomic_set(v,i)`	Asigna `i` al contador
`atomic_add(i,v)`	Incrementa el contador en `i` unidades
`atomic_sub(i,v)`	Decrementa `i` unidades el contador
`atomic_sub_and_test(i,v)`	Decrementa `i` unidades el contador y devuelve 1 si el resultado es 0
`atomic_inc(v)`	Incrementa el contador
`atomic_dec(v)`	Decrementa el contador
`atomic_dec_and_test(v)`	Decrementa el contador y devuelve 1 si el resultado es 0
`atomic_inc_and_test(v)`	Incrementa el contador y devuelve 1 si el resultado es 0
`atomic_add_negative(v)`	Incrementa el contador en `i` unidades y devuelve 1 si el resultado es negativo

[1] `v` ha de ser un puntero a `atomic_t` (valor por referencia).

Operaciones atómicas: `atomic64_t`

Función/Macro	Descripción
`ATOMIC64_INIT(i)`	En declaración, inicializa el contador a `i`
`atomic64_read(v)`[2]	Devuelve el valor entero asociado
`atomic64_set(v,i)`	Asigna `i` al contador
`atomic64_add(i,v)`	Incrementa el contador en `i` unidades
`atomic64_sub(i,v)`	Decrementa `i` unidades el contador
`atomic64_sub_and_test(i,v)`	Decrementa `i` unidades el contador y devuelve 1 si el resultado es 0
`atomic64_inc(v)`	Incrementa el contador
`atomic64_dec(v)`	Decrementa el contador
`atomic64_dec_and_test(v)`	Decrementa el contador y devuelve 1 si el resultado es 0
`atomic64_inc_and_test(v)`	Incrementa el contador y devuelve 1 si el resultado es 0
`atomic64_add_negative(v)`	Incrementa el contador en `i` unidades y devuelve 1 si el resultado es negativo

[2] `v` ha de ser un puntero a `atomic64_t` (valor por referencia).

Recuerda: Cerrojo o mutex

- Mecanismo ideal para resolver el problema de la sección crítica
- Podemos ver un cerrojo como un objeto con dos estados:
  - abierto ó cerrado
- Soporta dos operaciones atómicas:
  - *Adquirir el cerrojo (lock)*

```
lock() {
    while(estado!=abierto)
        .. espera ..

    estado=cerrado;
}
```

  - *Liberar el cerrojo (unlock)*

```
unlock() { estado=abierto; }
```

    - lo ejecuta el flujo de ejecución que lo cerró con `lock()` antes (propietario)

Spin locks (I)

- ***Spin lock***: es un tipo especial de cerrojo diseñado para entornos multiprocesador de memoria compartida (SMP)
  - Cerrojo de espera activa
    - Sólo puede ser adquirido por un flujo de ejecución del kernel al mismo tiempo
    - Si un flujo de ejecución intenta adquirir un cerrojo que no está libre, esperará a que se libere usando un bucle de espera (*busy wait/spin*)
  - En Linux representado por el tipo de datos `spinlock_t`
    - Operaciones declaradas en `<linux/spinlock.h>`
  - El tipo de cerrojo más utilizado en el kernel
    - Única alternativa de cerrojo en aquellos sitios donde no es posible realizar llamadas bloqueantes (p. ej., manejadores de interrupción)

Spin locks (II)

- Garantizan exclusión mutua en regiones de código que acceden a estructuras de datos compartidas entre flujos de ejecución

```
DEFINE_SPINLOCK(sp);
```

```
spin_lock(&sp);
.. Sección crítica 1 ..
spin_unlock(&sp);
```

Flujo de ejecución 1

```
spin_lock(&sp);
.. Sección crítica 2 ..
spin_unlock(&sp);
```

Flujo de ejecución 2

2 consideraciones:

1. `spin_lock()` inhibe la expropiación de kernel dentro de la sección crítica
2. No debemos ejecutar funciones bloqueantes (como `vmalloc()`) dentro de `spin_lock(); ... spin_unlock();`

Spin locks (III)

Función/Macro	Descripción
`DEFINE_SPINLOCK(name)`	Declara e inicializa un *spin lock* como variable (global) con nombre `name`
`spin_lock_init(sp)`[3]	Inicializa un *spin lock*
`spin_lock(sp)`	Adquiere el *spin lock*
`spin_unlock(sp)`	Libera el *spin lock*
`spin_trylock(sp)`	Intenta adquirir el *spin lock*. Si no está libre actualmente, el *spin lock* no se adquiere y la función devolverá un valor distinto de cero
`spin_is_locked(sp)`	Devuelve cero si el *spin lock* está libre actualmente, y un valor distinto de cero en otro caso

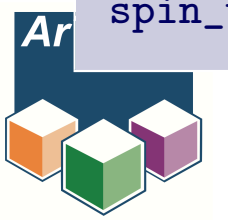
[3] `sp` ha de ser un puntero a `spinlock_t` (valor por referencia).

Spin locks (IV)

- Si usamos un *spin lock* para proteger estructura de datos compartida entre manejador de interrupción y cualquier otro flujo de ejecución del kernel → desactivar localmente las interrupciones antes de intentar adquirir el lock
  - No hacer esto puede provocar interbloqueo (*deadlock*)

Funciones adicionales

Función/Macro	Descripción
`spin_lock_irq(sp)`	Desactiva las interrupciones en la CPU actual y adquiere el *spin lock*
`spin_unlock_irq(sp)`	Reactiva las interrupciones en la CPU actual y libera el *spin lock*
`spin_lock_irqsave(sp,flags)`	Guarda en flags (`unsigned long`) el estado actual de las interrupciones en la CPU actual, deshabilita localmente las interrupciones y adquiere el *spin lock*
`spin_unlock_irqrestore(sp,flags)`	Libera el *spin lock* y restaura el antiguo estado de las interrupciones locales (almacenado en flags)



Código no *SMP-safe*

```
typedef struct{
    int data;
    struct list_head links;
}list_item_t;

LIST_HEAD(shared_list);

void list_insert_front(list_item_t* item)
{
        list_add(&item->links,&shared_list);
}

void list_print(char* buf)
{
        char* dst=buf;
        list_item_t* cur_entry=NULL;
        struct list_head *cur_node;

        list_for_each(cur_node,&shared_list)
        {
            cur_entry = list_entry(cur_node, list_item_t, links);
            dst+=sprintf(dst,"%i\n",cur_entry->data);
        }
}
```

Spin locks: ejemplo

```
LIST_HEAD(shared_list);
DEFINE_SPINLOCK(sp);

void list_insert_front(list_item_t* item)
{
       spin_lock(&sp);
       list_add(&item->links,&shared_list);
       spin_unlock(&sp);
}

void list_print(char* buf)
{
       char* dst=buf;
       list_item_t* cur_entry=NULL;
       struct list_head *cur_node;

       spin_lock(&sp);
       list_for_each(cur_node,&shared_list)
       {
           cur_entry = list_entry(cur_node, list_item_t, links);
           dst+=sprintf(dst,"%i\n",cur_entry->data);
       }
       spin_unlock(&sp);
}
```

Reader-Writer Spin Locks

- Tipo especial de *spin lock* que permite incrementar el grado de concurrencia en una sección crítica
  - Paradigma lectores-escritores:
    - Permite lecturas simultáneas pero sólo una escritura
    - Exclusión mutua entre lectura-esctritura y escritura-escritura
  - En Linux representado mediante el tipo de datos `rwlock_t`
    - Los lectores tienen prioridad→ inanición de los escritores

Funciones/Macros (`<linux/rwlock.h>`)

- **Inicialización:** `DEFINE_RWLOCK()`, `rwlock_init()`
- **Lector:** `read_lock()`, `read_unlock()`, `read_trylock()`, `read_lock_irq()`, `read_unlock_irq()`, `read_lock_irqsave()`, `read_unlock_irqrestore()`.
- **Escritor:** `write_lock()`, `write_unlock()`, `write_trylock()`, `write_lock_irq()`, `write_unlock_irq()`, `write_lock_irqsave()`, `write_unlock_irqrestore()`.

Reader-Writer Spin Locks: ejemplo

```
LIST_HEAD(shared_list);
DEFINE_RWLOCK(rwl);

void list_insert_front(list_item_t* item)
{
       write_lock(&rwl);
       list_add(&item->links,&shared_list);
       write_unlock(&rwl);
}

void list_print(char* buf)
{
       char* dst=buf;
       list_item_t* cur_entry=NULL;
       struct list_head *cur_node;

       read_lock(&rwl);
       list_for_each(cur_node,&shared_list)
       {
           cur_entry = list_entry(cur_node, list_item_t, links);
           dst+=sprintf(dst,"%i\n",cur_entry->data);
       }
       read_unlock(&rwl);
}
```

Semáforos (I)

Dos tipos de semáforos en Linux:

1. *Semáforos POSIX*: Utilizados por procesos de usuario
2. *Semáforos del kernel*: Usados por flujos de ejecución del kernel

Semáforos del kernel

- **Semáforos generales**
  - Contador asociado
  - Dos operaciones básicas:
    - `down()`: operación *wait()* del semáforo
    - `up()`: operación *signal()* del semáforo
- **Espera bloqueante**
  - Si un flujo de ejecución ejecuta operación *wait()* cuando contador del semáforo ≤ 0, el flujo se bloquea (*sleep*)
  - Solo puede utilizarse en **Contexto de Proceso** en modo kernel

Semáforos (II)

```
wait(s){
  s = s - 1;
  if(s < 0){
     <Bloquear al proceso>
  }
}

signal(s){
  s = s + 1;
  if (hay_procesos_bloqueados){
     <Desbloquear a un proceso
         bloqueado por wait>
  }
}
```

Significado de s

- $s \geq 0 \rightarrow s$ es el número de veces que se puede invocar a wait() sin que ningún proceso invocador se bloquee
- $s \leq 0 \rightarrow |s|$ es el número de procesos bloqueados en el semáforo

Semáforos (III)

Semáforos del kernel (Cont.)

- En Linux representado mediante `struct semaphore`
  - Operaciones declaradas en `<linux/semaphore.h>`
- Muy utilizados en implementación de llamadas al sistema (*handler functions*)
- Uso típico de los semáforos:
  1. Garantizar exclusión mutua
     - Inicializar contador del semáforo a 1
     - Encerrar las secciones críticas entre `down()` y `up()`
  2. Implementar sincronización basada en condiciones sobre números enteros
  3. Cola de espera
     - Inicializar contador del semáforo a 0
     - Bloqueo incondicional de proceso con `down()`
     - Despertar a proceso bloqueado con `up()`

Semáforos (IV)

Función/Macro	Descripción
`DEFINE_SEMAPHORE(name)`	Declara e inicializa a 1 un semaforo (`struct semaphore`) como variable (global) con nombre `name`
`sema_init(sem,n)`[4]	Inicializa un semáforo y establece su contador a `n`
`down(sem)`	Operación *wait()* del semáforo. En el caso de que se produzca un bloqueo, el kernel pone el proceso en estado `TASK_UNINTERRUPTIBLE` (el proceso NO responde a señales). No es recomendable utilizar esta operación
`up(sem)`	Operación *signal()* del semáforo.
`down_interruptible(sem)`	Variante de la operación *wait()* del semáforo. En el caso de que se produzca un bloqueo, el kernel pone el proceso en estado `TASK_INTERRUPTIBLE` (el proceso SÍ responde a señales.). Si el proceso es despertado por una señal antes de decrementar el semáforo, la función devuelve un valor distinto de 0 y 0 en caso contrario.

[4] `sem` ha de ser un puntero a `struct semaphore` (valor por referencia).

Semáforos (V)

Función/Macro	Descripción
`down_killable(sem)`	Variante de la operación *wait()* del semáforo. En el caso de que se produzca un bloqueo, el proceso se pondrá en estado especial `TASK_KILLABLE` (el proceso solo responde a señales que fuerzan su terminación). Sí el proceso es despertado por una señal antes de decrementar el semáforo, la función devuelve un valor distinto de 0 y 0 en caso contrario.
`down_trylock(sem)`	Intenta decrementar atómicamente el valor del semáforo y devuelve 1 si lo consigue. Si no es posible (contador≤ 0) devuelve 0.

Semáforos: Ejemplo 1

```
LIST_HEAD(shared_list);
DEFINE_SEMAPHORE(sem);

int list_insert_front(list_item_t* item)
{
        if (down_interruptible(&sem)){
            return -EINTR;
        }

        list_add(&item->links,&shared_list);
        up(&sem);
        return 0;
}
```

Semáforos: Ejemplo 1 (cont.)

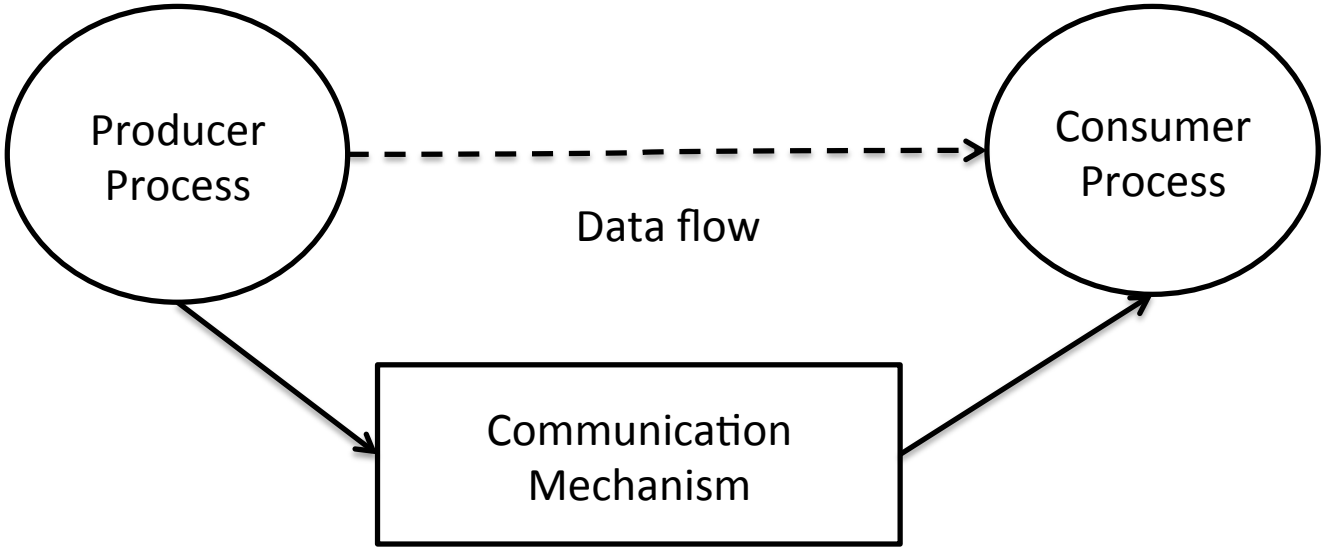

```
int list_print(char* buf)
{
        char* dst=buf;
        list_item_t* cur_entry=NULL;
        struct list_head *cur_node;

        if (down_interruptible(&sem)){
            return -EINTR;
        }

        list_for_each(cur_node,&shared_list)
        {
            cur_entry = list_entry(cur_node, list_item_t, links);
            dst+=sprintf(dst,"%i\n",cur_entry->data);
        }
        up(&sem);
        return 0;
}
```

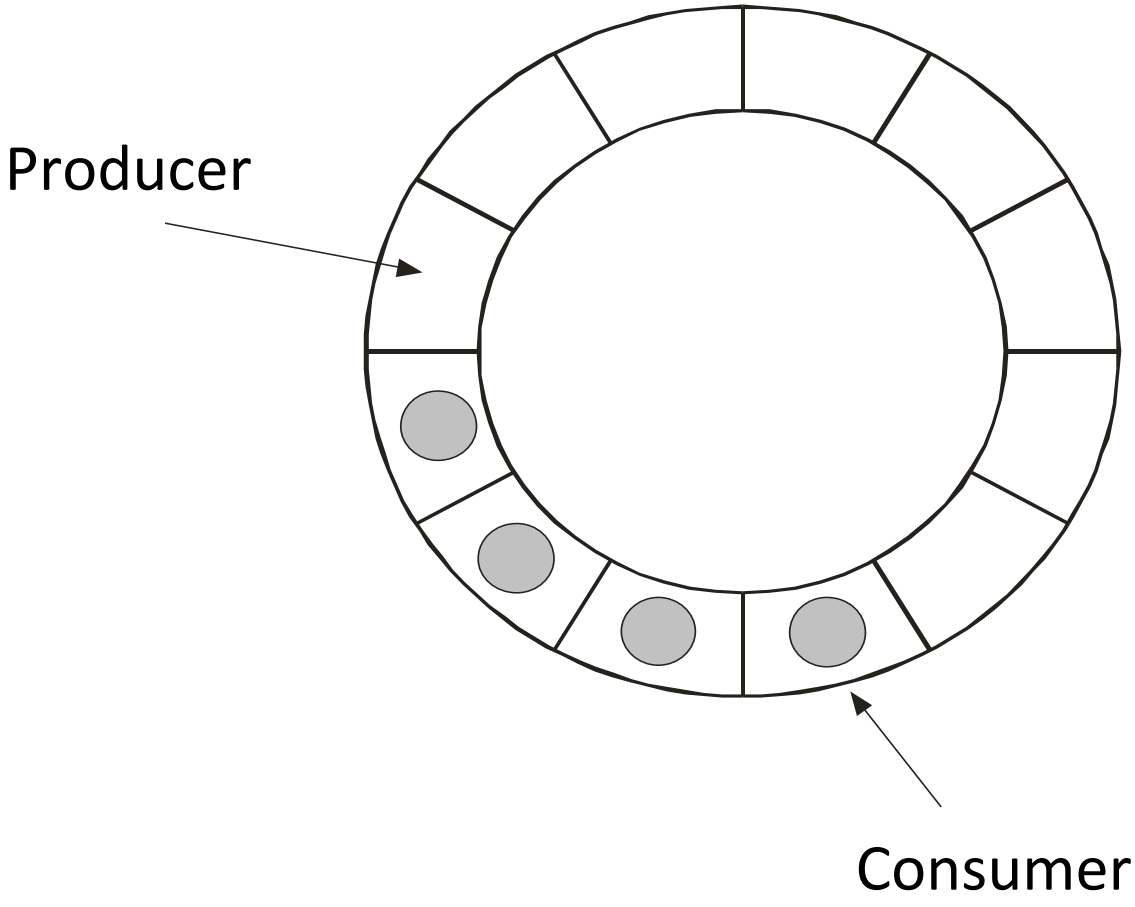
Semáforos: Ejemplo 2

Problema del productor-consumidor



- El consumidor se bloquea hasta que haya datos que leer
- El productor se bloqueará al intentar enviar datos a través del mecanismo de sincronización cuando esté lleno

Productor-consumidor: buffer circular



Semáforos: Ejemplo 2

Productor/consumidor

- Implementado mediante un módulo del kernel que gestiona buffer circular de punteros a enteros
- Módulo exporta entrada `/proc/prodcons`
  - Productor: `$ echo 7 > /proc/prodcons`
    - Inserta elemento al final del buffer
    - Se bloquea si no hay espacio en el buffer
  - Consumidor: `$ cat /proc/prodcons`
    - Consume el primer elemento del buffer y lo devuelve al programa de usuario
    - Si no hay elementos que consumir se bloquea

Semáforos: Ejemplo 2 (Inic.)

```
#define MAX_ITEMS_CBUF 5
static struct proc_dir_entry *proc_entry;
static cbuffer_t* cbuf;
struct semaphore elementos,huecos;

int init_prodcons_module( void )
{
  cbuf = create_cbuffer_t(MAX_ITEMS_CBUF);
  sema_init(&elementos,0);
  sema_init(&huecos,MAX_ITEMS_CBUF);

  if (!cbuf)
    return -ENOMEM;

  proc_entry = proc_create("prodcons",0666, NULL, &proc_entry_fops);

  if (proc_entry == NULL) {
      destroy_cbuffer_t(cbuf);
      return -ENOMEM;
  }

  return 0;
}
```

Semáforos: Ejemplo 2 (Productor)

```
ssize_t prodcons_write(struct file *filp, const char __user *buf,
                       size_t len, loff_t *off)
{
  char kbuf[MAX_CHARS_KBUF+1];
  int val=0;
  int* item=NULL;

  .. copy_from_user() + Convertir char* a entero y almacenarlo en val ..

  item=vmalloc(sizeof(int));
  (*item)=val;

  if (down_interruptible(&huecos)){
    vfree(item);
    return -EINTR;
  }

  insert_cbuffer_t(cbuf,item);

  up(&elementos);

  return len;
}
```

Semáforos: Ejemplo 2 (Consumidor)

```
ssize_t prodcons_read(struct file *filp, char __user *buf, size_t len, loff_t *off)
{
  char kbuf[MAX_CHARS_KBUF+1];
  int nr_bytes=0;
  int* item=NULL;

  if (offset>0)
    return 0;

  if (down_interruptible(&elementos)){
    return -EINTR;
  }

  item=head_cbuffer_t(cbuf);
  remove_cbuffer_t(cbuf);

  up(&huecos);

  nr_bytes=sprintf(kbuf,"%i\n",*item);

  vfree(item);

  ... copy_to_user() ...

  return nr_bytes;
}
```

Semáforos: Ejemplo 2

- El código anterior no garantiza exclusión mutua en el acceso al buffer
  - Si la estructura de datos no soportara acceso concurrente → potencial condición de carrera
  - **Solución:** Añadir semáforo adicional para garantizar exclusión mutua

Productor

```
 if (down_interruptible(&huecos)) {
    vfree(item);
    return -EINTR;
 }
/* Entrar a la SC */
if (down_interruptible(&mtx)) {
        vfree(item);
        up(&huecos);
        return -EINTR;
}
 insert_cbuffer_t(cbuf,item);
/* Salir de la SC */
up(&mtx);
up(&elementos);
```

Consumidor

```
if (down_interruptible(&elementos)) {
    return -EINTR;
 }
/* Entrar a la SC */
if (down_interruptible(&mtx)){
    up(&elementos);
    return -EINTR;
}
item=head_cbuffer_t(cbuf);
remove_cbuffer_t(cbuf);
/* Salir de la SC */
up(&mtx);
up(&huecos);
```

Recuerda: Variables condición

- Una variable condición puede verse como una **cola de espera con un cerrojo asociado**
  - Se llaman variables condición porque los hilos esperan en la cola hasta que se haga cierta una condición de nuestro programa

Las variables condición soportan 3 operaciones atómicas:

- `cond_wait()`: El hilo invocador se bloquea incondicionalmente en la cola de espera
- `cond_signal()`: se despierta a un solo hilo de los que esperan en la cola (si hay alguno)
- `cond_broadcast()`: se despierta a todos los hilos bloqueados en la cola de espera (si hay alguno)

Recuerda: Variables condición

Aspectos esenciales

- Las operaciones de las VC se han de invocar entre código encerrado entre `lock()` y `unlock()` del mutex asociado
  - Solo el propietario del mutex puede invocar operaciones sobre una VC
- Operaciones que realiza `cond_wait(vcond, mutex)`:
  1. unlock(mutex);
  2. El hilo se bloquea en la cola de `vcond`
  3. lock(mutex);
- Típicamente `cond_wait()` se invoca dentro de un while para poder recomprobar la condición de espera al salir del bloqueo:

```
while(condicion==false)
    cond_wait(vcond, mutex);
```

Variables condición con semáforos

- Las variables condición pueden emularse mediante dos semáforos y un contador
  1. Semáforo `sem_mtx` actúa como *mutex* asociado a la "variable condición"
     - Contador del semáforo se inicializa a 1
  2. Semáforo `sem_queue` que sirve de cola de espera
     - Contador del semáforo se inicializa a 0
  3. Contador `nr_waiting` lleva la cuenta del número de hilos esperando en la cola
     - Se inicializa a 0

Variables condición con semáforos

- `cond_wait()`

Pseudocódigo

```
lock(mutex);
..
while(condición==false)
    cond_wait(vcond,mutex);
..
unlock(mutex);
```

Implementación con semáforos

```
/* "Adquiere" el mutex */
if (down_interruptible(&sem_mtx))
    return -EINTR;
..
while(condición==false) {
    nr_waiting++;
    up(&sem_mtx); /* "Libera" el mutex */
    /* Se bloquea en la cola */
    if (down_interruptible(&sem_queue)){
        down(&sem_mtx);
        nr_waiting--;
        up(&sem_mtx);
        return -EINTR;
    }
    /* "Adquiere" el mutex */
    if (down_interruptible(&sem_mtx)
        return -EINTR;
}
..
/* "Libera" el mutex */
up(&sem_mtx);
```

Variables condición con semáforos

- `cond_signal()`

Pseudocódigo

```
lock(mutex);
..
cond_signal(vcond);
..
unlock(mutex);
```

Implementación con semáforos

```
/* "Adquiere" el mutex */
if (down_interruptible(&sem_mtx))
    return -EINTR;
..
if (nr_waiting>0) {
    /* Despierta a uno de los hilos
         bloqueados */
    up(&sem_queue);
    nr_waiting--;
}
..
/* "Libera" el mutex */
up(&sem_mtx);
```

Referencias

- Linux Kernel Development
  - Cap. 9 *"An Introduction to Kernel Synchronization"*
  - Cap. 10 *"Kernel Synchronization Methods"*
- Understanding the Linux Kernel
  - Cap. 11 *"Kernel Synchronization"*

Licencia

LIN - Sincronización en el kernel Linux
Versión 0.5

Este documento (o uno muy similar) está disponible en
`https://cv4.ucm.es/moodle/course/view.php?id=75410`