Compartición de recursos

Sistemes Operatius – pla 2003 (SO)
Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya

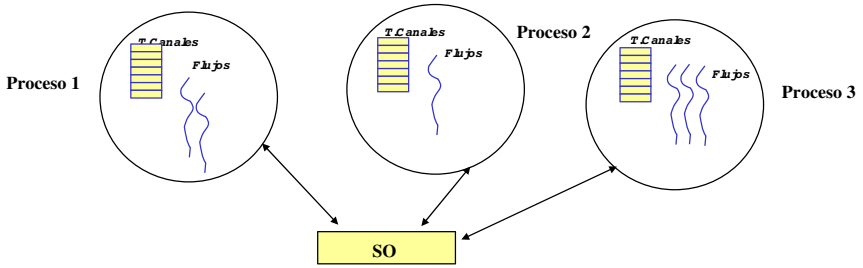
Licencia Creative Commons

Licencia Creative Commons

Índice

Concepto de proceso y flujo

- Un fichero ejecutable es algo estático, código almacenado en disco
  - Cuando se carga en memoria y se empieza a ejecutar es un programa en ejecución o PROCESO.
  - A cada parte del programa (código) que se puede ejecutar de forma independiente se le puede asociar un FLUJO



Gestión de flujos (i)

- Un Flujo (o thread) es la mínima unidad de planificación (CPU) del Sistema Operativo
  - Los flujos de un proceso comparten los recursos del proceso
  - Cada flujo del proceso tiene asociado:
    - Un PC (puntero a código)
    - Un SP (puntero a pila)
    - El estado de los registros
    - Un identificador
- Algunos SO sólo permiten un flujo por proceso (UNIX tradicional)

Gestion de flujos (ii)

- ¿Para qué se utilizan los flujos?
  - Permiten programar de forma modular (encapsular tareas)
  - Permiten explotar el paralelismo (multiprocesadores)
  - Programas que realizan Entrada/Salida
    - Usar flujos dedicados SOLO a la Entrada/Salida
  - Atención a varias peticiones de servicio (Servidores)

¿Que es un flujo (thread)?

- Los diferentes threads son partes del programa que pueden ejecutarse en paralelo (aunque puedan intercambiar información). Habitualmente subrutinas.
- Todos los threads comparten los recursos del proceso al que pertenecen (por ejemplo canales)
- Sólo se tiene una copia del código y los datos del proceso. Todos los threads comparten ese código y esos datos.
  - En cambio hay una pila por cada thread, para las variables locales

Que comparten y que no comparten los flujos?

- Compartido entre todos los threads de un mismo proceso
  - Codigo, Datos (variables globales), PCB (PID, PPID, ...) incluyendo Recursos (tabla de canales, gestión de signals,...)
- Propio de cada thread
  - Identificador thread, pila, registros (PC+SP), la variable errno

Ventajas de threads

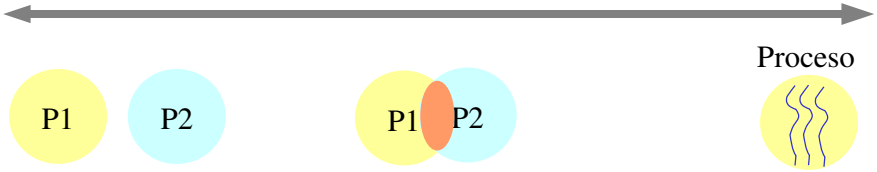
- Es más rápido crear un thread que un proceso
- Es más rápido terminar un thread que un proceso
- Es más rápido cambiar de thread (dentro del mismo proceso) que cambiar de proceso
- Ya que los threads comparten memoria y ficheros, se puede intercambiar información sin llamar a rutinas de sistema
  - Precisamente eso provoca la necesidad de exclusión mutua y sincronización.

Índice

- Concepto de proceso y flujo
- Ampliando el concepto: procesos con recursos compartidos
- Comunicación mediante memoria compartida
  - Sección crítica
  - Mutex
  - Semáforos
  - Deadlock (abrazo mortal)
- Una implementación: pthreads

Procesos con recursos compartidos

- Aunque en esta asignatura nos centremos en
  - Procesos, que NO comparten ningún recurso entre ellos y
  - Threads, donde TODOS los threads de un proceso comparten TODOS los recursos
- ... hay soluciones intermedias



Procesos que no comparten **ningún** recurso

Procesos con algún recurso compartido

Threads que comparten **todos** los recursos

¿Que pueden compartir los procesos?

- Espacio de memoria
  - Si un proceso modifica una variable, se modifica para todos
- La tabla de descriptores
  - Abrir/cerrar/moverse por un fichero afecta a todos los procesos
- La información del file system
  - Un chdir (por ejemplo) afecta a todos
- Manipulación de los signals
  - Si un proceso reprograma un signal, afecta a todos
- etc...

Un ejemplo: clone de Linux

- Propio de Linux. Lo usa para crear threads.
  - De hecho, cuando creas un proceso en Linux con clone decides cuanto quieres compartir entre padre e hijo.
  - En Linux no se hace distinción threads/procesos a la hora de la planificación: todo son tasks que pueden compartir (o no) recursos con otras tasks.
  - No portable. Si se quieren programar threads y exportarlo a otro entorno, debe usarse pthreads (al final del capítulo)

Clone

```
int clone(int (*fn)(void *), void *child_stack, int
  flags, void *arg);
```

- Devuelve el PID del proceso creado
- El proceso ejecuta la rutina fn(arg) – es diferente de fork()!!
- Se le debe pasar una pila child_stack (lo único que debe tener cada task y que no puede compartir)
- Flags:
  - CLONE_PARENT: el padre del proceso creado es el mismo que el del proceso creador
  - CLONE_FS: compartición de la información de file system
  - CLONE_FILES: compartición de la tabla de canales
  - CLONE_SIGHAND: compartición de la tabla de gestión de SIGNALS
  - ...

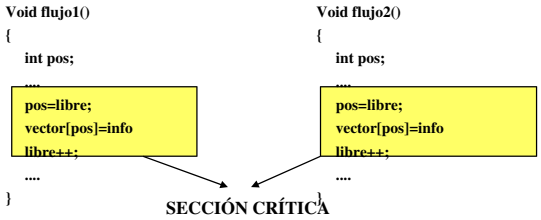
Índice

Comunicación mediante memoria compartida

- Dos flujos de un mismo proceso pueden usar la memoria física que comparten para comunicarse: memoria compartida
- Hay que asegurar que el resultado no dependa del orden de ejecución ya que no lo conocemos .Esta situación se llama “race condition”
  - Asegurar que dos flujos no puedan leer/escribir en la misma posición de memoria a la vez

Sección crítica

- El conjunto de instrucciones que acceden a variables compartidas es una “sección crítica”



Exclusión mutua

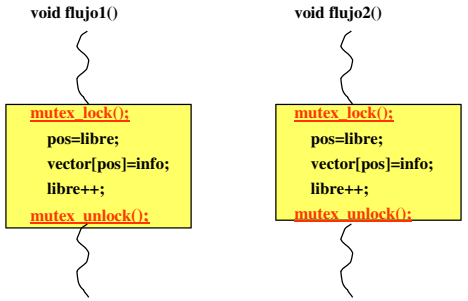
- Exclusión mutua
  - Mecanismo para garantizar que dos flujos no acceden simultáneamente a una sección crítica
- El S.O nos proporciona las herramientas pero es el programador el que ha de usarlas adecuadamente
- Las secciones críticas tienen un punto de entrada y un punto de salida

Acceso a una sección crítica

- Condiciones que garantizan un acceso correcto (Dijkstra)
  - Si un flujo está ejecutando una sección crítica, ningún otro flujo podrá entrar en la misma sección crítica
  - Un flujo no puede esperar indefinidamente para entrar en una sección crítica
  - Un flujo que está ejecutando fuera de una sección crítica no puede impedir que los otros flujos entren en ella
  - No se pueden hacer hipótesis sobre el número de procesadores, ni la velocidad de ejecución

Mutex

- La exclusión mutua se ha de cumplir aunque haya cambios de contexto dentro de la sección crítica
- Se secuencializa el acceso a las variables compartidas



Implementación mutex: busy waiting (i)

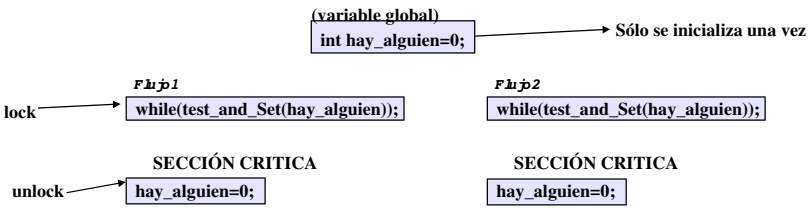
- Espera activa (busy waiting)
  - Necesitaremos soporte de la arquitectura: instrucción atómica, es decir, ininterrumpible (instrucción de lenguaje máquina)
    - Consulta y modificación de una variable de forma atómica
  - El equivalente en alto nivel sería...

```
int test_and_set(int *a)
{
   int tmp=*a;
   *a=1;
   return(tmp);
}
```

Es necesario hacerlo
Por hardware para que
Sea atómico

Implementación mutex: busy waiting (ii)

- ¿Cómo se usa?



Implementación mutex: busy waiting (iii)

- Inconvenientes:
  - Ocupamos la cpu mientras esperamos poder entrar en una sección crítica, (trabajo no útil)
    - Podríamos no dejar avanzar al flujo que ha de liberar la sección crítica
  - Podríamos colapsar el bus de memoria (siempre accediendo a la misma instrucción y la misma variable)
  - El resto de los usuarios no pueden aprovechar la cpu, el proceso no está bloqueado
- Posible solución
  - Consultar si podemos acceder a la sección crítica, y si no podemos, bloquear el proceso hasta que nos avisen que podemos acceder, en lugar de estar consultando continuamente como en espera activa

Implementación: bloqueo

- Idea:
  - Evitar el consumo inútil de tiempo de cpu.
  - Reducir el número de accesos a memoria
- Propuesta:
  - Bloquear a los flujos que no pueden entrar en ese momento
  - Desbloquearlos cuando quede libre la sección crítica
- Utilizaremos SEMAFOROS
  - sem_init(sem,n): crea un semáforo
  - sem_wait(sem):entrada en exclusión mutua (equivale al lock)
  - sem_signal(sem): salida de exclusión mutua (equivale al unlock)

Semáforos

- Qué es un semáforo??
  - Es una estructura de datos del SO que tendrá asociado un contador y una cola de procesos bloqueados. Sirve para proteger el acceso a recursos.
  - El contador indica la cantidad de accesos simultáneos que permitimos al recurso que protege el semáforo
    - Si usamos el semáforo para hacer exclusión mútua: Recurso=sección crítica, n=1.
    - Se pueden usar para más cosas

Una posible implementación de semáforos

- sem_init(sem,n);

```
sem->count=n;
Ini_queue(sem->queue);
```

- sem_wait(sem);

```
Sem->count--;
If (sem->count<0{
   bloquear_flujo(sem->queue);
}
```

/* bloquea al flujo que hace la llamada*/

- sem_signal(sem);

```
sem->count++;
If (sem->count<=0){
   despertar_flujo(sem->queue);
}
```

/* despierta un flujo de la cola */

Uso de semáforos

- En función del valor inicial del contador usaremos el semáforo para distintos fines
  - sem_init(sem,1): MUTEX (permitimos que 1 flujo acceda de forma simultanea a la sección crítica)
  - sem_init(sem,0): SINCRONIZACIÓN
  - sem_init(sem,N): RESTRICCIÓN DE RECURSOS, genérico
- Habitualmente usaremos:
  - Espera activa si los tiempos de espera se prevén cortos
  - Bloqueo si se prevén largos
    - Bloquear un flujo es costoso (entrar a sistema)
- Ejemplo lectores/escritores

Problemas concurrencia: deadlock

- Se produce un abrazo mortal entre un conjunto de flujos, si cada flujo del conjunto está bloqueado esperando un acontecimiento que solamente puede estar provocado por otro flujo del conjunto

```
Flujo 1
Conseguir(impresora)
Conseguir(cinta)
   imprimir_datos_cinta()
Liberar(cinta)
Liberar(impresora)
```

```
Flujo2
Conseguir(cinta)
Conseguir(impresora)
   imprimir_datos_cinta()
Liberar(cinta)
Liberar(impresora)
```


Condiciones del deadlock

- Se han de cumplir 4 condiciones a la vez para que haya abrazo mortal
  - Mututal exclusion: mínimo de 2 recursos no compartibles
  - Hold&Wait: un flujo consigue un recurso y espera por otro
  - No preempción: si un flujo consigue un recurso sólo él puede liberarlo y nadie se lo puede quitar
  - Circular wait: ha de haber una cadena circular de 2 o más flujos donde cada uno necesita un recurso que esta siendo usado por otro de la cadena

Evitar deadlocks

- Como evitarlos????, evitar que se cumpla alguna de las condiciones anteriores
  - Tener recursos compartidos
  - Poder quitarle un recurso a un flujo
  - Poder conseguir todos los recursos que necesitas de forma atómica
  - Ordenar las peticiones de recursos (tener que conseguirlos en el mismo orden)

Índice

- Concepto de proceso y flujo
- Ampliando el concepto: procesos con recursos compartidos
- Comunicación mediante memoria compartida
  - Sección crítica
  - Mutex
  - Semáforos
  - Deadlock (abrazo mortal)
- Una implementación: pthreads

Implementación: Pthreads

- POSIX Threads (Portable Operating System Interface, definido por la IEEE)
  - Librería de flujos de usuario
  - Define la interfície de gestión de flujos
    - Creación, destrucción
    - Prioridades
    - Planificación
  - Estándar definido para conseguir portabilidad (POSIX 1003.1c - 1995)
- Disponible a la mayoría de los SO (p.ej. Linux y W2K)
- Sólo veremos unas pocas

Creación

- Creación de un pthread

```
#include <pthread.h>
int pthread_create(
    pthread_t       * thread ,
    pthread_attr_t * attr ,
    void *          (* start_routine) (void *),
    void            * arg );
```

- Parámetros
  - **thread** Identificador de thread.
  - **attr** Atributos del thread, si ponemos NULL se inicializa con los atributos por defecto
  - **start_routine,** función que ejecutará el nuevo thread
  - **Arg** dirección que recibe como parámetro el nuevo thread.
- Qué devuelve?
  - **0 si OK**
  - **Código de error**

Identificación y destrucción

- Identificación del pthread

```
#include <pthread.h>
pthread_t pthread_self (void);
```

  - Devuelve el identificador del pthread
- Destruir flujo

```
#include <pthread.h>
void pthread_exit(void * status);
```

  - Lo realiza el flujo que va a morir (equivalente al exit de procesos)
  - status,valor que recibirá la función que espera la finalización del flujo

Esperar flujo

- Esperar flujo

```
#include <pthread.h>
int pthread_join (pthread_t thread,void  ** status);
```

  - Parámetros
    - thread, identificador del flujo que estamos esperando.
    - status, estado de finalización del flujo que estábamos esperando.
  - Qué devuelve?
    - 0 si OK
    - Código de error
  - Join puede ser bloqueante (hasta que acabe el flujo esperado)

Mutex en Pthreads

- Iniciar mutex

```
int pthread_mutex_init (pthread_mutex_t *mutex, cons
  pthread_mutexattr_t *mutexatr);
```

  - Inicia la variable mutex. Una variable mutex sólo tiene dos estados: locked y unlocked.
  - El atributo lo consideraremos NULL (inicialización por defecto a unlocked)
  - Siempre devuelve 0 (el resto de funciones relacionadas con mutex devuelve 0 si todo ha ido bien o diferente de 0 si error)

Mutex en Pthreads : lock y unlock

- Lock

```
int pthread_mutex_lock(p_thread_mutex_t *mutex);
```

  - Funciona como se espera de un lock
  - Es bloqueante, si se quiere no bloqueante, puede usarse

```
int pthread_mutex_trylock(p_thread_mutex_t *mutex);
```

  - Hace lo mismo que lock, pero no bloquea y devuelve -1 con errno=EBUSY si está locked

- Unlock

```
int pthread_mutex_unlock(p_thread_mutex_t *mutex);
```