Sistemas Operativos

Comunicación y Sincronización - III

Sistemas Operativos

- ☑ Versión: Abril 2017
- ☑ Palabras Claves: Proceso, Comunicación, Mensajes, mailbox, port, send, receive, IPC, Productor, Consumidor

Algunas diapositivas han sido extraídas de las ofrecidas para docentes desde el libro de Stallings (Sistemas Operativos) , el de Silberschatz (Operating Systems Concepts)

Productor - Consumidor

- Es un paradigma de la cooperación de procesos.
- Se basa en dos tipos de procesos (o hilos)
  - un productor
  - un consumidor.
- Se cuenta con área de memoria común donde el productor pone elementos que quedan al servicio del consumidor.

Detalles

- El productor modifica el área con el agregado de elementos.
- El consumidor sólo lee desde el área común.
- El área de memoria es finita. Si se generarán muchos datos conviene buffer circular.

Sincronización

- El consumidor no trate de tomar elementos cuando no los hay
- El productor no debe poner elementos cuando el buffer esta lleno
- Ambos procesos deben esperar cuando sea necesario
- No se asume velocidad de ejecución, ni se presupone alternancia (uno produce, uno consume)

Implementación con sleep y wakeup

```
#define N 100  /* ranuras en buffer*/
int cuenta=0
Void productor(void)
{
  Int elemento
  While (true) {
    Elemento=producir_elemento;
    If (cuenta ==N) sleep();
    Insertar_elemento(elemento);
    Cuenta=cuenta+1;
    If (cuenta==1)
     wakeup(consumidor);
   }
}
```

```
Void consumidor(void)
{
  Int elemento
  While (true) {
    If (cuenta ==0) sleep();
    Elemento=quitar_elemento;
    Cuenta=cuenta-1;
    If (cuenta==N-1)
     wakeup(productor);
    consumir_elemento(elemento);
  }
}
```

Condición de carrera: acceso a *cuenta* no restringido.
Analizar: buffer vacío, testeo de cuenta, detención del Consumidor y arranque del productor. Wakeup del Productor que se pierde.

Semáforos

- Es una herramienta de sincronización
- Sirve para solucionar el problema de la sección crítica.
- Sirve para solucionar problemas de sincronización.

Funcionamiento

- Es una variable entera
  - Inicializada en un valor no negativo
- Dos operaciones:
  - wait (down, p)
    - Decrementa el valor. El proceso no puede continuar ante un valor negativo, se bloquea.
  - signal (up, v)
    - Incrementa el valor. Desbloqueo de un proceso
- Operaciones atómicas
  - Cuando un proceso modifica el valor del semáforo, otros procesos no pueden modificarlo simultáneamente.

Ejemplo implementación wait y signal

- Wait

  While S ≤ 0 do no op;

  S:=S-1

- Signal

  S:=S+1

Sol. Productor/consumidor usando semáforos

- Consideremos un pool de n buffers.
- Se usan los siguientes semáforos: vacios, llenos y mutex.
- *vacios* cuenta la cantidad de buffers vacíos (inicializado en n)
- *llenos* (inicializado en 0) y cuenta la cantidad de buffers llenos.
- *Mutex* es inicializado en 1.

Código productor/consumidor

```
repeat
    ...
    produce un ítem en
    nextp
    ...
    wait(vacios);
    wait(mutex);
    ...
    agrega nextp al buffer
    ...
    signal(mutex);
    signal(llenos);
until false;
```

```
repeat
    wait(llenos);
    wait(mutex);
    ...
    saca el ítem del buffer a
        nextc
    ...
    signal(mutex);
    signal(vacias);
    ...
    consume el ítem en
    nextc
    ...
    until false;
```

Considerar que

- El mutex es para asegurar la exclusión mutua (acceso al buffer)
- "Vacios" y "llenos" se usan para sincronización
  - productor no produzca si buffer lleno
  - consumidor no consuma si buffer está vacío

Alternativas en la espera

- Busy waiting
  - gasta CPU, ejecuta un loop hasta poder entrar a SC.
- Autobloqueo
  - cuando el proceso ve que tiene que esperar, se bloquea.
- Se pone en una cola asociada con el semáforo.
  - Se rearranca por un wakeup cuando se ejecuta el signal en los procesos en SC.

Mutex

- Podemos ver el mutex como una versión simplificada del semáforo.
- Son buenos SÓLO para garantizar exclusión mutua.
- Variable que está en estado abierto (desbloqueado) o cerrado (bloqueado).
- Se pueden implementar en espacio de usuario.
- Útil para sincronización de ULTs

Procedimientos del mutex

- Mutex_lock y mutex_unlock
- Cuando un hilo/proceso necesita acceder a la SC invoca a mutex_lock
- Si el mutex está abierto, puede entrar a la SC.
- Si está cerrado, se bloquea hasta que el hilo que está usando libere la SC invocando a mutex_unlock.
- Si hay varios hilos bloqueados por el mutex, se selecciona uno y se le da el mutex.

Implementación (I)

- Uso de test-and-set/swap
- El hilo no puede quedarse en espera ocupada pues no permite que se ejecute otro hilo.
- Cuando no puede adquirir un mutex, invoca a thread_yield para darle la CPU a otro hilo.
- No hay espera ocupada
- thread_yield es un planificador de hilos a nivel de usuario.
- No hay llamadas al kernel.

Implementación (II)

- mutex_lock:

```
    TSL registro,mutex
    CMP registro,0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok: RET
```

- mutex_unlock:

```
    move mutex,0
    ret
```

IPC – Semáforos - SysCalls

- semget

```
#include <sys/types.h>
#include<sys/ipc.h>
#include<sys/sem.h>
int semget (key_t key, int nsems,int semflg);
```

- Crear/obtener un grupo de semáforos.
- nsems: cantidad de semáforos en el grupo
- flags:
  - IPC_CREATE: Lo crea si no existe

IPC_EXCL: Error si lo intenta crear y existe

  - Permisos (User, Group, Others)
- Retorna descriptor del semáforo, o un error

IPC – Semáforos - SysCalls

- semop

```
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/sem.h>
int semop(int semid, struct sembuf __user * sops, unsigned nsops);
```

- Operaciónes sobre semáforos del grupo
- sembuf: puntero a un vector de operaciones
- nsops: cantidad de operaciones
  - Longitud de sembuf

IPC – Semáforos - SysCalls

semop (cont.)

```
struct sembuf {
   unsigned short sem_num; //Número de semáforo del grupo
   short sem_op;           /*Operación sobre el semáforo*/
   short sem_flg;          /* Opciones */
};
```

sem_op:

- >0 : Se suma al valor del semáforo y se despiertan todos los procesos que esperan.
- <0: Si es posible se resta al valor del semáforo, sino se bloquea el proceso.
- Nulo: Comprobar el semáforo, si es nulo se bloquea al proceso.

sem_flg:

- IPC_NOWAIT: No se bloquea al proceso

IPC – Semáforos - SysCalls

semclt

```
#include <sys/types.h>
#include <sys/ipc.h>
#include<sys/sem.h>
int semctl(int semid, int semnum, int cmd,union semun arg);
```

- Consulta, Modificación y Borrado de un grupo de semáforos.
- semnum: id del semáforo del grupo
- cmd:
  - SEM_INFO/ SEM_STAT
  - SETVAL
  - IPC_SET
  - IPC_RMID, etc

IPC – Semáforos - SysCalls

☑ semclt (cont.)

```
union semun {
  int val; /* Valor para SETVAL */

  struct semid_ds __user *buf; /* Memoria de datos para IPC_STAT e
IPC_SET */

  unsigned short __user *array; /* Tabla para GETALL y SETALL */

  struct seminfo __user *__buf; /* Memoria de datos para IPC_INFO */

  void __user *__pad; /*Puntero de alineación de la estructura*/
};
```

Ejemplo

```
int semaforo; //Valor del semáforo
struct sembuf P,V; //Estr. para las op’ s de incremento y decremento

/*Creamos un semáforo con semget después de obtener una clave mediante
ftok. Solo se creará un semáforo para este conjunto*/
void crear_semaforo()
{
    key_t key=ftok("/bin/ls",1); //Crea la clave
    //Sem para controlar el acceso exclusivo al recurso compartido
    semaforo = semget(key, 1, IPC_CREAT | 0666);
    //Se inicializa el semáforo a 1.
    semctl(semaforo,0,SETVAL,1);
    //P decrementa en 1 y V lo incrementa abrirlo.
//El flag SEM_UNDO hace que si un proceso termina inesperadamente
//deshace las operaciones que ha realizado sobre el sem.
        P.sem_num = 0;
        P.sem_op = -1;
        P.sem_flg = SEM_UNDO;
        V.sem_num = 0;
        V.sem_op = 1;
        V.sem_flg = SEM_UNDO;
}
```

Ejemplo (cont.)

```
void Imprimir_por_pantalla ()
{
pid_t pid; /*Creamos un proceso hijo que imprime en pantalla*/
pid = fork(); // Comprobamos que ha sido posible crear el proceso hijo
if (pid != -1) {
    if (pid == 0) {//Código del Hijo
        sleep(1);
// El proceso hijo adquiere el semáforo para acceder al recurso
// compartido (pantalla) antes de imprimir el texto por pantalla.
        semop(semaforo,&P,1);
        cout << “Soy el proceso hijo” <<endl;
        //Libera el semáforo
        semop(semaforo,&V,1);
    }
    else { //Código del Padre
        ….
```

Ejemplo (cont.)

```
        else { //Código del Padre
                sleep(2);
// El proceso padre adquiere el semáforo para acceder al recurso
// compartido (pantalla)antes de imprimir el texto por pantalla.
                semop(semaforo,&P,1);
                cout << “Soy el proceso padre” <<endl;
                //Libera el semáforo
                semop(semaforo,&V,1);
        }
}
else {
        cout<<"No se ha podido crear el proceso hijo"<<endl;
    exit(-1);
  }
}
int main () {
        crear_semaforo();
        while (true) {
                Imprimir_por_pantalla();
        }
}
```

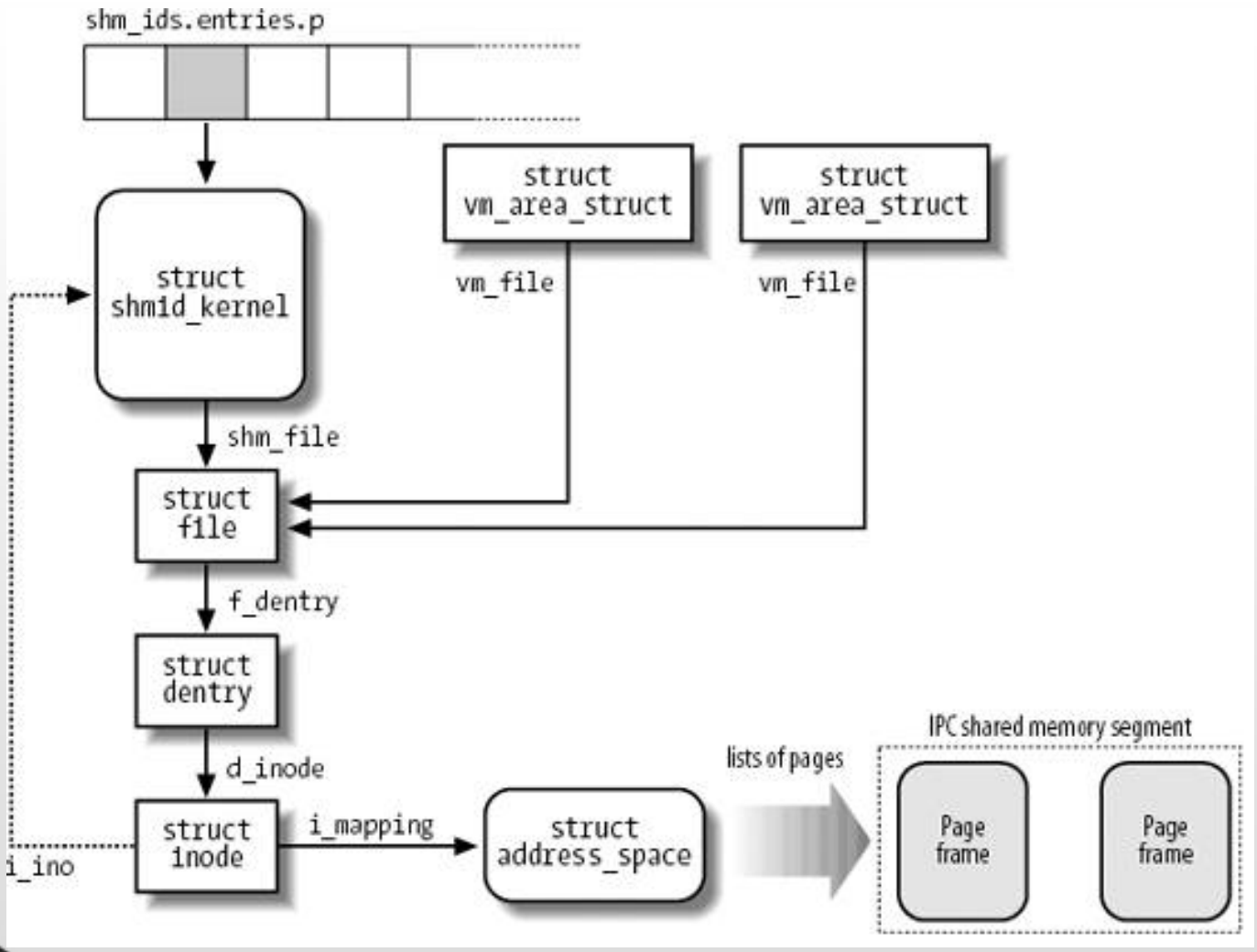
Memoria Compartida

- Tradicionalmente cada proceso cuenta con su espacio de direcciones
  - Direcciones Virtuales
- Un proceso NO puede acceder al espacio de otro
  - Protección de la memoria
- Los procesos siguen “viendo” un espacio virtual
  - Cada región compartida puede estar en diferente lugar del Espacio de Direcciones de cada proceso.

Memoria Compartida (cont.)

- La técnica permite a dos o mas procesos compartir un segmento de memoria.
- Permite la transferencia de datos entre procesos
  - Comunicación
- Se requieren mecanismos de Sincronización
  - Semáforos

Memoria Compartida - Estructura



Memoria Compartida - SysCalls

- shmget

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

- Permite obtener/crear un segmento de memoria compartida
- Flags:
  - IPC_CREATE: Crearlo si no existe
  - IPC_EXCL: Falla si al crear uno, ya existe
  - MODOS: RWX (Owner, Group, Others)

Memoria Compartida - SysCalls

- shmctl

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
```

- Operaciones de control
- cmd:
  - IPC_STAT: Estado
  - SHM_LOCK: Bloquear la zona de memoria en memoria principal (Solo si es super usuario)
  - SHM_UNLCK: Desbloquear la zona de la memoria
  - Etc

Memoria Compartida - SysCalls

- shmat

```
#include <sys/types.h>
#include <sys/shm.h>
void* shmat (int shmid, const void *shmaddr, int option);
```

- Adjuntar la zona de memoria compartida al espacio de dir. del proceso.
- shmaddr:
  - Dirección especifica del espacio virtual
  - Null : el SO decide donde
- option:
  - SHM_RDONLY

Memoria Compartida - SysCalls

- shmdt

```
#include <sys/shm.h>
#include <sys/types.h>
int shmdt(void *shmaddr);
```

- Quitar la zona de memoria compartida del espacio del proceso.
- shmaddr:
  - Dirección obtenida en shmat

Comandos IPC

- ipcs
  - Obtener información de los objetos IPC del sistema

```
------ Shared Memory Segments --------
key        shmid      owner      perms      bytes      nattch
status
0x0052e2c1 557056     postgres  600        29278208   4

------ Semaphore Arrays --------
key        semid      owner      perms      nsems
0x00000000 34144256   www-data  600        1
0x00000000 34177025   www-data  600        1
0x0052e2c1 7798786    postgres  600        17

------ Message Queues --------
key        msqid      owner      perms      used-bytes   messages
```

Comandos IPC (cont.)

- ipcmk
  - Crear un objeto IPC
- ipcrm
  - Eliminar un objeto IPC

Referencias

- System V IPC

  http://www.tldp.org/LDP/lpg/node21.html (05/2014)