Sistemas Operativos – GII

Cuarto Parcial. Sincronización y comunicación. 18 de diciembre de 2017

Cuestiones:

Responder **brevemente pero de forma razonada** a las siguientes preguntas.

1) [1 punto] Se desea diseñar un sistema de reservas online de entradas de cine. ¿Qué modelo de comunicación sería preferible usar?

Este tipo de sistema se ajusta mejor a un modelo de comunicaciones cliente-servidor, frente a un modelo P2P, dado que la base de datos de reservas estaría centralizada en el servidor, facilitando el control del acceso sincronizado de los clientes y, aunque se tendrán múltiples clientes, el número no será tan elevado.

2) [1 punto] Se desean comunicar y asegurar exclusión mutua entre N procesos ligeros. Indicar qué opciones existen.

Para comunicar se podría usar las regiones de la imagen de memoria que comparten o bien un fichero, siendo la primera opción la más sencilla y más rápida.
Para asegurar exclusión mutua se podrían usar mutex/condición o bien semáforos.

3) [1 punto] Se desea comunicar dos procesos remotos. Uno de ellos recogerá imágenes de una cámara conectada por USB y le enviará los fotogramas adquiridos al otro proceso. Detallar qué mecanismo de comunicación se debe usar.

El mecanismo adecuado para comunicar dos procesos remotos serían los sockets. Y, si en este caso asumimos que la pérdida o llegada desordenada de algún mensaje con un fotograma no supone un problema, el tipo de sockets que se usarían serían de tipo datagrama.

4) [1 punto] Explique qué ocurre si un proceso solicita leer 10 bytes de una tubería cuando dicha tubería: (a) está vacía y hay un descriptor de escritura asociado a la misma; (b) está vacía y no hay ningún descriptor de escritura asociado a la misma; (c) contiene 5 bytes; (d) contiene 15 bytes.

a) **La operación de lectura se queda bloqueada**
b) **La lectura devuelve 0 (EOF)**
c) **Se leen 5 bytes**
d) **Se leen 10 bytes**

Problema

La barrera es un mecanismo que permite sincronizar múltiples flujos de ejecución, tengan éstos el mismo código o diferente, tal que cuando un flujo llega a una barrera, no proseguirá su ejecución hasta que todos los demás flujos alcancen la misma. Este mecanismo ofrece una función para iniciar la barrera (`inicio_barrera(barr_t *b, int nflujos)`), que recibe como parámetros el descriptor de la barrera y el número total de flujos que usarán la misma, y otra función que se invoca cada vez que se quiere realizar la sincronización propiamente dicha (`barrera(barr_t *b, int flujoID)`), que recibe como parámetros el descriptor de la barrera y el identificador del flujo que invoca la función, que será un valor entre 0 y *`nflujos-1`*. Este mecanismo puede implementarse usando cualquiera de las herramientas de sincronización que ofrece el sistema operativo.

a) Solución basada en semáforos sin nombre pero **sólo para dos flujos de ejecución**. Considere un escenario en el que el programa inicia la barrera y, a continuación, crea **dos** *threads* a cada uno de los cuales les pasa como parámetros el descriptor de la barrera y su identificador de flujo. Se plantea una solución que usa un semáforo por cada flujo. **Se pide identificar a qué corresponde cada uno de los 5 símbolos que incluyen una interrogación** (**NOTA**: Tenga en cuenta que, al tratarse de sólo 2 flujos, si uno tiene como identificador *`x`*, al otro le corresponderá *`1-x`*).

```
typedef struct {
  int nflujos;
  sem_t *sems; //vector de semáforos
} barr_t;
void inicio_barrera(barr_t *b, int nflujos){
  // implementada de forma genérica
  b->nflujos = nflujos;
  b->sems = malloc(nflujos * sizeof(sem_t));
  for (int i=0; i<nflujos; i++)
    sem_init(&(b->sems[i]), 0, ?1);
}
```

```
void barrera(barr_t *b, int flujoID){
  // implementada sólo para 2 flujos (0 y 1)
  sem_?2(&(b->sems[?3]));
  sem_?4(&(b->sems[?5]));
}
```

b) Solución basada en *mutex* y variables de condición. Considere el mismo escenario que en el apartado anterior, pero **para un número cualquiera de flujos**. Se plantea una solución que usa un *mutex* y una variable de condición, así como una variable (`nflujos_sinc`) que refleja cuántos flujos han alcanzado la barrera en cada momento (inicialmente, ninguno). **Se pide identificar a qué corresponde cada una de las dos operaciones de sincronización**.

```
typedef struct {
  int nflujos; int nflujos_sinc;
  pthread_mutex_t m; pthread_cond_t c;
} barr_t;
void inicio_barrera(barr_t *b,int nflujos){
  b->nflujos = nflujos;
  b->nflujos_sinc = 0;
  pthread_mutex_init(&(b->m), NULL);
  pthread_cond_init(&(b->c), NULL);
}
```

```
void barrera(barr_t *b, int flujoID){
  pthread_mutex_lock(&(b->m));
  if (++b->nflujos_sinc == b->nflujos) {
     b->nflujos_sinc=0;
     // OP1 de sincronización
  } else {
     // OP2 de sincronización
  }
  pthread_mutex_unlock(&(b->m));
}
```

c) Solución basada en *sockets stream*. Considere un escenario en el que se ejecutan en máquinas distintas **dos programas** independientes que reciben los argumentos pertinentes (dirección IP, puerto, …). Se plantea una solución asimétrica en el sentido de que cada proceso ejecuta un código

diferente. Suponga que el código que aparece a continuación corresponde a la iniciación de la barrera en cada flujo.

Código de inicio de la barrera del flujo 0

```
a = socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);

struct sockaddr_in dir = {
   .sin_family = AF_INET,
   .sin_port = htons(atoi(argv[1])),
   .sin_addr ={INADDR_ANY}};

b = bind(a,(struct sockaddr *)&dir,
         sizeof(dir));
c = listen(a,1);
d = accept(a, NULL, NULL);
```

Código de inicio de la barrera del flujo 1

```
a = socket(PF_INET,SOCK_STREAM,IPPROTO_TCP);

struct sockaddr_in dir = {
   .sin_family = AF_INET,
   .sin_port = htons(atoi(argv[1])),
   .sin_addr={inet_addr(argv[2])}};

b = connect(a,(struct sockaddr *)&dir,
            sizeof(dir));
```

Una vez realizada esa iniciación, cada vez que se tengan que sincronizar los procesos en la barrera, se intercambiarán entre sí un mensaje que contenga un byte con un valor cualquiera usando un esquema cliente-servidor. **Se pide detallar únicamente el código de cada flujo que realiza esa sincronización siguiendo el esquema propuesto.**

Solución

a) En la solución planteada en el enunciado para el caso de dos flujos de ejecución con un semáforo por cada uno, cada flujo debe notificar al otro que ha llegado a la barrera y esperar a que suceda la misma circunstancia con el otro flujo. Por tanto, en primer lugar, cada flujo realizará un `sem_post` del semáforo del otro flujo y, a continuación, un `sem_wait` de su semáforo, que, al estar iniciado a 0, lo hará esperar hasta que haya llegado la notificación del otro flujo.

`?1 → 0`

`?2 → sem_post`

`?3 → 1-flujoID`

`?4 → sem_wait`

`?5 → flujoID`

Dando como resultado, el siguiente código:

```
  . . . . .
  sem_init(&(b->sems[i]), 0, 0);
}
void barrera(barr_t *b, int flujoID){
  // implementada sólo para 2 flujos (0 y 1)
  sem_post(&(b->sems[1-flujoID]));
  sem_wait(&(b->sems[flujoID]));
}
```

Nótese que también sería válida una solución donde los índices del vector de semáforos se usen justo al contrario: `?3 → flujoID` y `?5 → 1-flujoID`

b) En la solución propuesta con *mutex* y variables de sincronización, la rama `else` del `if` corresponde a la llegada a la barrera de los sucesivos flujos, exceptuando el último, y, por tanto, la acción de sincronización debe significar el bloqueo del flujo mediante `pthread_cond_wait`. Por otro lado, la rama afirmativa del `if` corresponde a cuando alcanza la barrera el último flujo y, por tanto, tiene que desbloquear a todos los demás flujos que están esperando mediante `pthread_cond_broadcast`.

```
void barrera(barr_t *b, int flujoID){
```

```
  pthread_mutex_lock(&(b->m));
  if (++b->nflujos_sinc == b->nflujos) {
     b->nflujos_sinc=0;
     pthread_cond_broadcast(&(b->c));        // OP1 de sincronización
  } else {
     pthread_cond_wait(&(b->c), &(b->m));    // OP2 de sincronización
  }
  pthread_mutex_unlock(&(b->m));
}
```

c) En la solución basada en *sockets stream* planteada en el enunciado, en la fase inicial los dos flujos establecen una conexión entre sí. A la hora de realizar la sincronización asociada a la barrera, tal como establece el enunciado, los flujos tienen que intercambiar un byte siguiendo un esquema cliente-servidor.

Código de la barrera del flujo 0

```
char c;
read(d, &c, 1);
write(d, &c, 1);
```

Código de la barrera del flujo 1

```
char c;
write(a, &c, 1);
read(a, &c, 1);
```

Tenga en cuenta que el envío de datos por un *socket*, sea mediante `write` o usando `send`, no conlleva ningún tipo de sincronización entre el emisor y el receptor: los datos se copian a un *buffer* local y se completa la llamada de petición de envío.

Nótese que en el flujo 0, que actúa de servidor, hay que usar en la comunicación el valor devuelto por la llamada `accept`, que corresponde al descriptor de *socket* conectado con el cliente, mientras que en el flujo 1 se utiliza directamente el valor devuelto por la llamada `socket`.