Sistemas Operativos 4º Semestre. Grados II y MI

Cuarto Parcial. Sistema de Ficheros. 2 de Junio de 2014.

Dispone de 50 minutos. Publicación de notas el Jueves 5 de Junio 2014 .

Revisión el Viernes 6 de Junio de 2014 12:00 a.m. Despacho 4202

Primera cuestión (2 puntos)

Explique lo más conciso posible

1. ¿Por qué el sistema operativo necesita ofrecer mecanismos de comunicación y sincronismo (C&S) para los procesos? ¿Cuál es el problema que resuelven dichos mecanismos de C&S?
2. Explique concisamente los conceptos "condición de carrera" e "interbloqueo".
3. ¿Qué característica común tienen todos los mecanismos de C&S para evitar una condición de carrera entre dos o más procesos?
4. ¿Qué problema nuevo aparece al utilizar los mecanismos de C&S para la asignación de recursos? ¿Cómo lo podemos resolver?

Segunda cuestión (2 puntos)

A continuación se muestra el fragmento de código de un proceso ligero escritor, del ejemplo de lectores y escritores con mutex y variables de condición que aparece en las transparencias de clase. Suponiendo que el hilo principal crea tres lectores y tres escritores, responda a las siguientes preguntas:

1. ¿Qué hace la línea 04? ¿Por qué está en un while?
2. En la línea 06, ¿por qué antes de acceder al recurso compartido se suelta el mutex? ¿Qué estaba protegiendo entonces el mutex?
3. El la línea 08, ¿por qué se vuelve a coger el mutex previo a la señalización ?
4. En la línea 10 y 11, ¿por qué se señaliza con "signal" a los escritores y con "broadcast" a los lectores?

```
01 void Escritor (void)
02 {
03   pthread_mutex_lock (&mutex);
04   while (leyendo!=0 || escribiendo!=0)  // cond. Espera
            pthread_cond_wait (&a_escribir, &mutex);
05   escribiendo++;
06   pthread_mutex_unlock (&mutex);

07    << Acceso en exclusiva al recurso compartido >>

08   pthread_mutex_lock (&mutex);
09   escribiendo--;
10   pthread_cond_signal (&a_escribir);
11   pthread_cond_broadcast (&a_leer);
12   pthread_mutex_unlock (&mutex);
13 }
```

Ejercicio (6 puntos)

Se desea diseñar un sistema informático de reservas de entradas para una sala de cine de 500 plazas, empleando una arquitectura cliente servidor. Por simplicidad suponemos que sólo se puede comprar una entrada en cada operación. También por simplicidad suponemos la sala como una matriz lineal de asientos, de 1 a 500.

El proceso servidor tiene acceso a un fichero donde se almacena para cada asiento la estructura de datos "asiento", que incluye para cada butaca el estado de la butaca (libre, reservado u ocupado), número de butaca (1 a 500) y los datos del comprador como; hora de la venta, medio de pago, nº de tarjeta de crédito, etc. Para el control de acceso concurrente al fichero se emplean cerrojos.

El proceso servidor tiene un único proceso pesado que escucha peticiones en "venta.entradas.es" en el puerto 1294. Cuando llega un nuevo cliente, el servidor crea un nuevo proceso pesado, servidor dedicado, que atiende la venta completa, realizando la modificación del nuevo estado de la sala en el fichero.

Los procesos clientes remotos dialogan con su servidor dedicado. El cliente recibirá el estado actual de la sala, ocultando como es lógico todos los datos privados de los compradores, y mostrando en pantalla al comprador la ocupación de las butacas.

Para minimizar el número de consultas al fichero se cuenta en el servidor con una matriz **char estado_de_la_sala** (línea 04) que mantiene en memoria el estado de libre, ocupado o reservado de cada una de las plazas. Al iniciar el servidor esta matriz se marcarán todos los asientos como libres, y en cada venta de entrada se irá actualizando a ocupado.

A continuación se muestra parte del código de los procesos cliente y servidor. Algunas líneas y parte del código se han omitido de forma intencionada para que responda las siguientes preguntas.

Se pide:

a) (1 punto) Indique qué le parece la solución elegida. Si es adecuado el uso de comunicación TCP o sería más adecuado UDP y por qué.

b) (1 punto) Codifique para el programa cliente la función de la línea 17 **recibir_del_servidor (cd, estado_de_la_sala)**,

c) (1 punto) Codifique para el programa servidor la función de la línea 30 **enviar_al_cliente (cd, estado_de_la_sala);**

d) (1 punto) Explique la diferencia entre las funciones reservar () y ocupar () del servidor. Líneas 32 y 35 respectivamente.

e) (1 punto) Codifique una de las dos funciones anteriores.

f) (1 punto) La matriz **char estado_de_la_sala** del servidor pretende ser una variable compartida entre todos los procesos padre e hijos del servidor que refleje el estado libre, ocupado o reservado. Indique si es correcta su utilización al ser modificado por los procesos hijo (línea 37) .

Común

```
01 # define BUTACA_LIBRE 'L'
02 # define BUTACA_OCUPADA 'O'
03 # define BUTACA_RESERVADA 'R'
04 # define BUTACAS 500
05 struct asiento {
        char estado;    /* tres posibles estados L=libre, O=ocupado, R=reservada */
        unsigned n_butaca;                              /* 0 < n_butaca < BUTACAS */
        struct datos_t comprador;     /*nombre, nº tarjeta de crédito, hora, etc */
       };
```

Cliente

```
01 int main() {
02 int size, ret;
03 struct asiento mi_butaca;        /* datos de la butaca que va a ser comprada */
04 char estado_de_la_sala[BUTACAS];    /* butacas L=libre O=ocupado R=reservada */
05 struct hostent * hp;
06 struct sockaddr_in s_ain;
07 int cd;                            /* descriptor para el socket de cliente */
08
09 hp = gethostbyname ("venta.entradas.es");
10 bzero ((char *) &s_ain, sizeof (s_ain));
11 s_ain.sin_family = AF_INET;
12 memcpy (&(s_ain.sin_addr), hp->h_addr, hp->h_length);
13 s_ain.sin_port = htons (1294);
14 cd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
15 connect (cd, (struct sockaddr *) &s_ain, sizeof (s_ain));
16 do{
17     recibir_del_servidor(cd, estado_de_la_sala);  /*se pide al servidor estado
18                                                          ocupación sala */
19     elegir_entrada(estado_sala, &mi_butaca); /*se muestra estado sala y usuario
20                                                    elige butaca y paga */
21     ret = comprar(cd, &mi_butaca);   /* se intenta comprar la opción en el
22                                  servidor. Si lo consigue comprar devuelve 0 */
23   } while (ret!=0);
24 confirmacion_compra(&mi_butaca);       /* muestra al usuario la confirmación */
25 close (cd);
26 return 0;
27 }
```

Servidor

```
01 int main() {
02 int i, size, error;
03 struct asiento su_butaca;        /* datos de la butaca que va a ser comprada */
04 char estado_de_la_sala[BUTACAS]; /*   L=libre O=ocupado R=reservado
05                                      el estado de la sala se almacena en esta
06                                      variable, además de en el fichero      */
07 struct hostent * hp;
08 struct sockaddr_in s_ain, c_ain;
09 int cd, sd;            /* descriptor para el socket de cliente y de servidor */
10 int fd;           /* descriptor del fichero que refleja el estado de la sala */
11
12 fd = open ("fichero_estado_de_la_sala",O_RDWR);
13 for (i=0, i<BUTACAS, i++) estado_de_la_sala[i] = BUTACA_LIBRE;
```

```
14 sd = socket (AF_INET, SOCK_STREAM, IPPROTO_TCP);
15 bzero ((char *) &s_ain, sizeof (s_ain));
16 s_ain.sin_family = AF_INET;
17 s_ain.sin_addr.s_addr = INADDR_ANY;                    /* Cualquier origen */
18 s_ain.sin_port = htons (1294);
19 bind (sd, (struct sockaddr *)&s_ain, sizeof(s_ain));
20 listen(sd, 100);
21 while (1) {
22    size = sizeof(c_ain);
23    cd = accept (sd, (struct socaddr *)&c_ain, &size);
24    switch (fork()) {
25    case -1:
26         perror ("error en servidor");
27         return 1;
28    case 0:              /* Comienza código del hijo que atiende al cliente */
29         close (sd);
30         do { enviar_al_cliente(cd, estado_de_la_sala);  /* Comienza la venta */
31              pedir_cliente(cd, &su_butaca);
32             if (reservar (fd, &su_butaca) == 0){ /*reservar devuelve 0 si ok */
33                 estado_de_la_sala[su_butaca.n_butaca] = BUTACA_RESERVADA;
34                 if (cobrar (&su_butaca) == 0){  /* cobrar devuelve 0 si ok */
35                     ocupar (fd, &su_butaca);/* se escribe en fichero ocupada */
36                     error = notificar_al_cliente_ok(cd);  /* envía cliente 0 */
37                     estado_de_la_sala[su_butaca.n_butaca] = BUTACA_OCUPADA;
38                     }
39                 else {
40                     error = notificar_al_cliente_error_pago(cd); /* error -1 */
41                     liberar (fd , &su_butaca);
42                     estado_de_la_sala[su_butaca.n_butaca] = BUTACA_LIBRE;
43                     }
44                 } /* fin del primer if */
45             else {
46                  liberar (fd , &su_butaca);
47                  error = notifica_al_cliente_butaca_ocupada(cd); /* error -2 */
48                  }
49         while (error != 0);           /* Fin del do-while y fin de la venta */
50         exit (0);                                  /* Fin del código del hijo */
51    default:                                      /* Comienza código del padre */
52         close (cd);
53         return (0);
54       }  /* switch */
55    }     /* while  */
56   }      /* main   */
```

Resolución

Primera cuestión

P1.a *¿Por qué el sistema operativo necesita ofrecer mecanismos de C&S?* Cada proceso ve su propio espacio de memoria virtual y no ve nada del resto de procesos. Ya en capítulos anteriores al de C&S se introdujeron medios de comunicarse como son la memoria compartida y con ficheros. No obstante aparece el problema de la condiciones de carrera y es necesario sincronizar los procesos para evitarlo. El sistema operativo ofrece otros mecanismos de comunicación aparte de los descritos, que ya incluyen la sincronización, y que evitan por tanto el problema de condición de carrera. Además la sincronización es necesaria si queremos por ejemplo que varios procesos cooperantes se ejecuten en un orden determinado, ya que desde la programación no podemos controlar la planificación de los procesos.

P1.b *¿Cuál es el problema que resuelven los mecanismos de C&S?* Introducíamos la concurrencia para optimizar el aprovechamiento de la CPU. Con ella surge la necesidad de comunicar los procesos para que puedan cooperar entre si. Al introducir mecanismos de comunicación **aparece el problema de la la condición de carrera** que es resuelto con los mecanismos de sincronización introduciendo operaciones indivisibles o atómicas. Por tanto el problema que resuelven es la condición de carrera.

P2.a *Condición de carrera.* Se dice que dos o más procesos están en condición de carrera si el resultado depende del orden en que se ejecuten. Se da principalmente cuando dos o más procesos acceden al mismo tiempo a un recurso compartido, por ejemplo una variable, cambiando su estado y obteniendo de esta forma un valor no esperado de la misma.

P2.b *Interbloqueo*, bloqueo mutuo, abrazo de la muerte o *deadlock* es cuando dos o más procesos se bloquean de forma permanente porque todos ellos necesitan para poder continuar algo que tiene alguno de los procesos bloqueados. No tiene una solución general.

P3 *Características de los mecanismos de C&S.* Todos ellos implementan instrucciones atómicas o indivisibles, que garantizan que el proceso que comienza a ejecutarlas, la acabe, antes de que el proceso sea desalojado de la CPU. Así, por ejemplo, son atómicas las operaciones de hacerse o deshacerse con un mutex o semáforo, de leer o escribir en una tubería, *fifo* o *socket*, etc.

P4.a *¿Problema de los mecanismos de C&S?* Aparece el bloqueo de los interbloqueos.

P4.b *¿Cómo lo podemos resolver?* No tiene una solución general. Existen algunos algoritmos como el del banquero, aunque no dan una solución genérica. Lo normal es seguir una serie de buenas praxis a la hora de programar para evitarlos, aunque esto tampoco garantiza que no aparezcan.

Segunda cuestión

P.1.a *¿Qué hace la línea 04?* Partimos de una situación en la línea anterior en la que se acaba de conseguir el *mutex*. En la línea 04 primer lugar comprueba una cierta condición sobre las barreras para ver si debe conservar o no el *mutex*. En este caso debe conservar el *mutex* si no hay ningún lector y ningún escritor. Es decir si las variables “leyendo” y “escribiendo” valen cero. Si es cierta la condición se llama a “pthread_cond_wait” que hace dos cosas; En primer lugar suelta el *mutex*. En segundo lugar bloquea al proceso en espera de que algún otro proceso señalice sobre la variable de condición “a_escribir”.

P1.b *¿Por qué está en un while?* Cuando algún otro proceso señalice sobre la variable de condición “a_escribir” con un “pthread_cond_signal”, el proceso volverá a despertar, haciéndose de nuevo con el *mutex*. Está en un *while* porque debe volver a evaluarse las condiciones sobre las barreras, por si estas han cambiado. En caso de que vuelvan a ser ciertas volverá a soltar el *mutex* y bloquearse, como se describió anteriormente. Si son falsas seguirá avanzando dentro de la sección crítica.

P2.a *¿Por qué se suelta mutex línea 06?* Al incrementar la variable barrera “escribiendo” en la línea anterior, ya se está accediendo al recurso en exclusividad. No tiene sentido seguir manteniendo el *mutex*. Téngase en cuenta que la llamada *pthread_mutex_lock()* no es bloqueante. Los procesos estarán en esa llamada realizando espera activa. La filosofía es siempre soltar el *mutex* lo antes posible para que el resto

de procesos ligeros, lectores o escritores, puedan comprobar las barreras, bloquearse y soltar el *mutex* en su correspondiente *pthread_cond_wait()*.

P2.b *¿Qué estaba protegiendo el mutex?* Protege la modificación de las variables barrera, "leyendo" y "escribiendo". En este caso el código del escritor modifica la barrera "escribiendo" por lo que se debe garantizar su modificación y su comprobación en exclusividad.

P3 *¿Por qué se vuelve a coger el mutex línea 08?* Por que se va modificar de nuevo la variable barrera "escribiendo" que registra que el proceso escritor abandona el recurso compartido.

P4 *¿Por qué signal y broadcast? Signal* se usa porque va a despertarse a un sólo escritor, el primero de la lista de espera. Ya que sólo puede entrar un escritor en el recurso compartido.

Broadcast en cambio va a despertar a todos los lectores, ya que podrían entrar todos ellos en el recurso compartido.

Ejercicio

a) *¿Es adecuado el uso de TCP/IP?* Si, es necesario un protocolo fiable que garantice la comunicación, ya que estamos haciendo una transacción económica, y si se pierden datos, alguien puede perder dinero. UDP no garantiza que los mensajes lleguen al destinatario.

b) *Codifique en el cliente línea 17* `recibir_del_servidor (cd, estado_de_la_sala)`; Lo que hace esta función sería leer del servidor el estado de la sala a través del descriptor del *socket* "cd". A través de este descriptor el cliente se puso en contacto con el servidor ejecutando "connect" y "accept" respectivamente. El proceso cliente mostraría dicho estado al comprador para que elija la butaca que desea comprar. Sin entrar en mucho detalle sería una función con la línea de código:

```
recv ( cd, estado_de_la_sala, BUTACAS, 0);
```

c) *Codifique para el servidor línea 30* `enviar_al_cliente(cd, estado_de_la_sala);` Lo que hace esta función es mandarle al proceso cliente el estado de la sala que se va almacenando en la variable global estado_de_la_sala. Para ello necesitamos escribir esta variable en el descriptor del socket que lo comunica con el cliente. Sin entrar mucho en detalle sería una función con la línea de código:

```
send ( cd, estado_de_la_sala, BUTACAS, 0);
```

d) *Diferencia entre función reservar() y ocupar():* Ambas funciones escriben en el fichero y por tanto, en principio, deberían manejar cerrojos para acceder a la zona del fichero que están escribiendo. Sin embargo la reserva se realiza sobre el fichero en primer lugar, accediendo en exclusividad gracias al cerrojo, y luego en la variable global estado_de_la_sala, donde no es necesario ya la exclusividad porque ya se hizo en el fichero, y nadie va a reservar ya esa plaza. El tener la reserva ya marcada en el fichero impide que otros procesos puedan ya reservarla escribiendo en esa posición del fichero. La compra escribe en el fichero una vez que se ha garantizado el cobro de la entrada, y luego modifica también la variable global. No sería necesario poner cerrojo en la compra por que al estar reservada nadie la puede ya reservar, y menos comprar. Así que no va a haber nadie que compita por modificar dicha posición en el fichero. Tampoco habría condición de carrera al pasar de "R" (reservado) a "O" (ocupado) en la variable global estado_de_la_sala.

e) *Códifique una de ellas.* Aunque es más fácil codificar la función comprar, porque como se ha dicho en el apartado anterior, no hay que establecer cerrojo, a continuación se muestra la codificación de la función reservar ()

```
/* el fichero de la base de datos está abierto y el descriptor fd viene como
   parámetro */
/* el parámetro butaca es un puntero de struct asiento a los datos de la butaca que
   se desea reservar, con el número de la butaca todos los datos del cliente */

int reservar (int fd, struct asiento *butaca)
{
  struct flock cerrojo;
  int posicion;  /* posición donde empiezan los datos de la butaca a reservar */
  int resultado; /* 0 = todo va bien 1= butaca ya ocupada*/
  char estado;   /* estado de la butaca 'L' libre, 'O' ocupada, 'R' reservada */

  posicion = (butaca->n_butaca)*sizeof (struct asiento);

  /* establezco el cerrojo desde el principio del fichero, en la butaca que se desea
     reservar y por una longitud de la estructura asiento */

  cerrojo.l_whence= SEEK_SET;
  cerrojo.l_start = posicion;
  cerrojo.l_len = sizeof (struct asiento);
  cerrojo.l_pid = getpid();
  cerrojo.l_type = R_WRLCK; /* Cerrojo compartido voy a leer si está ya reservado */
  fcntl (fd, F_SETLKW, &cerrojo);   /* establezco cerrojo exclusivo porque no quiero
                                       que me lo modifique otro proceso */
  lseek (fd, posicion, SEEK_SET);   /* me posiciono */
  read (fd, &estado, sizeof(char)); /* leo estado de la butaca, primer campo de la
                                       estructura butaca */
  if (estado != 'L')  /* si es cierto es que otro proceso lo reservó antes que yo */
        resultado = 1;              /* devuelvo 1 indicando que ya está reservada */
  else {
        lseek (fd, posicion, SEEK_SET);             /* me vuelvo a posicionar */
        write (fd, asiento, sizeof(struct butaca)); /* reservo butaca */
        resultado = 0;               /* 0 = reserva hecha correctamente */
        }
  cerrojo.l_type = F_UNLCK;
  fcntl (fd, F_SETLK, &cerrojo);  /* abre el cerrojo */
  return resultado;
} /* fin de la función reservar() */
```

f) *¿Es correcta la utilización de la variable global estado_de_la_sala"?* No, no es correcta. Dicha variable no es vista por los hijos. Debería ser una zona de memoria compartida o los procesos fueran procesos ligeros.