CC3301 Control 3

2 horas

Noviembre de 2009

Pregunta 1 (Sincronización: threads)

Se quiere implementar un sistema de autos que se adelantan en una calle con tres vías: una sólo par el sur, otra sólo para el norte y una al medio que sólo se usa para adelantamientos en ambas direcciones. Para evitar accidentes, implementamos acceso exclusivo a la pista de adelantamiento:

```
/* dir == SUR o NORTE */

pthread_mutex_t pista;

adelantar(int dir) {
   pthread_mutex_lock(&pista);
}

volver(int dir) {
   pthread_mutex_unlock(&pista);
}

/* Un thread por auto */
auto(int dir) {
   for(;;) {
        avanzar_hasta_prox_auto(dir);
        adelantar(dir);
        avanzar_pista_aux(dir);
        volver(dir);
   }
}
```

Parte I

El problema de esta solución es que no permite compartir la vía de adelantamiento entre autos que van en la misma dirección. Modifique el código para que esto sea posible.

```
int dir = SUR;
int count = 0;
pthread_cond_t cnt0;

adelantar(int dir) {
   pthread_mutex_lock(&pista);
   while(dir != cur_dir && count > 0)
      pthread_cond_wait(&cnt0, &pista);

   count++;
   cur_dir = dir;
   pthread_mutex_unlock(&pista);
}

volver(int dir) {
   pthread_mutex_lock(&pista);
   count--;
   if(count == 0) pthread_cond_broadcast(&cnt0);
   pthread_mutex_unlock(&pista);
}
```

Parte II

Modifique su solución para asegurar que la pista de adelantamiento no pueda ser monopolizada en una sola dirección. Para ello, utilice un contador de autos que van entrando en la misma dirección. Llegado a MAX_AUTOS, si hay autos esperando adelantar en la dirección contraria, deben dejar de aceptar nuevos autos, esperar que se desocupe completamente la pista, y aceptar los autos en dirección contraria.

```
int cur_dir = SUR;
int count = 0;
int tot_lado = 0;
pthread_cond_t cntsur0, cntnorte0;
int wait_norte = 0;
int wait_sur = 0;

adelantar(int dir) {
   pthread_mutex_lock(&pista);
   while((dir != cur_dir && count > 0) || tot_lado >= MAX)
        if(dir == SUR) {
            wait_sur++;
            pthread_cond_wait(&cntsur0, &pista);
            wait_sur--;
        }
        else {
            wait_norte++;
```

```
            pthread_cond_wait(&cntnorte0, &pista);
            wait_norte--;
        }

   count++; tot_lado++;
   cur_dir = dir;
   pthread_mutex_unlock(&pista);
}

volver(int dir) {
   pthread_mutex_lock(&pista);
   count--;
   if(count == 0) {
        if(dir == SUR) {
            if(wait_norte > 0)
                pthread_cond_broadcast(&cntnorte0);
            else
                pthread_cond_broadcast(&cntsur0);
        }
        if(dir == NORTE) {
            if(wait_sur > 0)
                pthread_cond_broadcast(&cntsur0);
            else
                pthread_cond_broadcast(&cntnorte0);
        }
        tot_lado = 0;
   }
   pthread_mutex_unlock(&pista);
}
```

Pregunta 2 (jsockets)

Se le pide implementar un servidor multi-cliente que implemente una especie de çhat", que es como un eco múltiple: todo lo que llega desde un cliente debe enviarse a todos los otros clientes (excluyendo al enviador). También se debe poder aceptar nuevos clientes en cualquier momento. Pueden tomar el servidor de eco visto en clases como modelo.

Analice las alternativas de implementación disponibles y elija la que se comporte de mejor manera donde la carga del servidor sea alta y la robustez de la solución no sea un problema. Mencione las ventajas y desventajas de las otras alternativas.

Debe ser un servidor multi-clientes, y la opción de usar fork() queda descartada, ya que los hijos no comparten memoria con el padre y no podrán entonces usar los sockets de los otros procesos para replicar los mensajes. La mejor opción es usar threads, y atender a un cliente en cada uno.

```
int sock[MAX_CLIENTS];

void serv(int j) {
   int cnt, size = BUF_SIZE;
   char buf[BUF_SIZE];
   int i;

    fprintf(stderr, "cliente conectado\n");
    while((cnt=read(sock[j], buf, size)) > 0) {
        for(i=0; i < MAX_CLIENTS; i++)
            if(sock[i] != -1 && i != j) {
            if(write(sock[i], buf, cnt) < 0) /* broken pipe */
                sock[i] = -1;
            }
    }
   close(sock[j]);
   sock[j] = -1;

   fprintf(stderr, "cliente desconectado\n");
}

main() {
   int s, s2;
   pthread_t pid;
   int i;

   signal(SIGPIPE, SIG_IGN);

   for(i=0; i < MAX_CLIENTS; i++)
        sock[i] = -1;

   s = j_socket();

   if(j_bind(s, 1818) < 0) {
        fprintf(stderr, "bind failed\n");
        exit(1);
   }

/* Cada vez que se conecta un cliente le creo un thread */
   for(;;) {
        s2 = j_accept(s);
        for(i=0; i < MAX_CLIENTS; i++)
            if(sock[i] == -1) break;
        if(i == MAX_CLIENTS) { close(s2); continue; }

        sock[i] = s2;
```

```
        if( pthread_create(&pid, NULL, (void *)serv, (void *)i) != 0) {
            fprintf(stderr, "No pude crear thread!!!\n");
            exit(1);
        }
    }
}
```

Pregunta 3 (Deadlocks, inanición y eficiencia)

Se propone la siguiente solución a N buffers para múltiples productores y consumidores:

```
void putbox(BOX *b, char c)
{
    pthread_mutex_lock(&b->mutex);
    sem_wait(&b->vacios);
    sem_post(&b->llenos);
    b->buf[b->in] = c;
    b->in = (b->in+1)%NBUFS;
    pthread_mutex_unlock(&b->mutex);
}

char getbox(BOX *b)
{
    char c;

    pthread_mutex_lock(&b->mutex);
    sem_wait(&b->llenos);
    sem_post(&b->vacios);
    c = b->buf[b->out];
    b->out = (b->out+1)%NBUFS;
    pthread_mutex_unlock(&b->mutex);
    return(c);
}
```

Comente si está correcta, si permite paralelismo entre productores y consumidores y si no genera deadlocks. Explique su análisis y haga las menos correcciones posibles que generen una solución correcta.

Como el mutex es compartido, generamos un deadlock, ya que si nos bloqueamos en un wait, el otro thread no podrá desbloquearnos ya que no podrá tomar el mutex. Lo primero, entonces, es separar los mutex en dos o acercarlos al código interno (la primera genera más paralelismo entre prods/cons, pero la segunda también es correcta):

```
void putbox(BOX *b, char c)
{
```

```
    pthread_mutex_lock(&b->mutex_put);
    sem_wait(&b->vacios);
    sem_post(&b->llenos);
    b->buf[b->in] = c;
    b->in = (b->in+1)%NBUFS;
    pthread_mutex_unlock(&b->mutex_put);
}

char getbox(BOX *b)
{
    char c;

    pthread_mutex_lock(&b->mutex_get);
    sem_wait(&b->llenos);
    sem_post(&b->vacios);
    c = b->buf[b->out];
    b->out = (b->out+1)%NBUFS;
    pthread_mutex_unlock(&b->mutex_get);
    return(c);
}
```

Ahora igual tenemos un error: como hacemos post de llenos antes poner el dato, puede despertarse un get y sacar un dato que aun no ha sido escrito (pasa cuando in == out). Al revés (post de vacios) también ocurre lo mismo. El mover el mutex (o usar uno compartido) no arregla el problema. Debemos mover el post después del código que pone/saca el dato, pero el incremento de in/out puede quedar después. Esto es porque in/out no se comparte entre prod/cons y la protección entre productores y entre consumidores ya está dada por el mutex:

```
void putbox(BOX *b, char c)
{
    pthread_mutex_lock(&b->mutex_put);
    sem_wait(&b->vacios);
    b->buf[b->in] = c;
    sem_post(&b->llenos);
    b->in = (b->in+1)%NBUFS;
    pthread_mutex_unlock(&b->mutex_put);
}

char getbox(BOX *b)
{
    char c;

    pthread_mutex_lock(&b->mutex_get);
    sem_wait(&b->llenos);
    c = b->buf[b->out];
```

```
    sem_post(&b->vacios);
    b->out = (b->out+1)%NBUFS;
    pthread_mutex_unlock(&b->mutex_get);
    return(c);
}
```