Secciones críticas, sincronización con mutex y condiciones, el problema del productor/consumidor.

Basadas en las clases del relator

Rodrigo Arenas A., Luis Mateu B. & Lucas Torrealba A.

18 de marzo de 2024

Contenidos

Secciones críticas

- Motivación: El problema de la cuenta compartida.
- Para la autorización de las transacciones:
  - Si saldo es suficiente se acepta y se modifica el saldo.
  - En caso contrario se rechaza.

```
1 Diccionario dicc;
2 void init(){
3     dicc = nuevoDiccionario();
4 }
5 // Funcion a realizar
6 int autorizar(int cuenta, int monto){
7
8 }
```

Secciones críticas: Cuenta compartida incorrecta

```
1  Diccionario dicc;
2  void init(){
3      dicc = nuevoDiccionario();
4  }
5  int autorizar(int cuenta, int monto){
6      int ret = FALSO;
7
8      int saldo = consultar(dicc, cuenta);
9      if (saldo - monto >= 0){
10         int nuevo_monto = saldo - monto;
11         modificar(dicc, cuenta, nuevo_saldo);
12         ret = VERDADERO;
13     }
14     return ret;
15 }
```

¿Problema? Posibles **datarace**.

Secciones críticas: Cuenta compartida

¿Como garantizar la exclusión mutua en la sección crítica?
Utilizando mutex, donde tenemos las siguientes operaciones:

```
1  #include <pthread.h>
2  pthread_mutex_t mutex;
3  //Creación
4  int pthread_mutex_init(pthread_mutex_t *m, pthread_mutexattr_t *a);
5  //Destrucción y liberar recursos
6  int pthread_mutex_destroy(pthread_mutex_t *m);
7  //Toma de mutex
8  int pthread_mutex_lock(pthread_mutex_t *m);
9  //Liberación de mutex
10 int pthread_mutex_unlock(pthread_mutex_t *m);
```

Secciones críticas: Cuenta compartida correcta

```
1  Diccionario dicc;
2  void init(){
3      dicc = nuevoDiccionario();
4  }
5  pthread_mutex_t m;
6  int autorizar(int cuenta, int monto){
7      int ret = FALSO;
8      // Inicio seccion critica
9      pthread_mutex_lock(&m);
10     int saldo = consultar(dicc, cuenta);
11     if (saldo - monto >= 0){
12         int nuevo_monto = saldo - monto;
13         modificar(dicc, cuenta, nuevo_saldo);
14         ret = VERDADERO;
15     }
16     pthread_mutex_unlock(&m);
17     // Fin seccion critica
18     return ret;
19 }
```

Productor/Consumidor

Motivación: reproducción de un vídeo.

```
1 void reproducir(){
2    for(;;){
3       Cuadro *cuadro = leerCuadro(); //lee de internet
4       if(c==NULL)
5          break;
6       mostrarCuadro(cuadro); //muestra en pantalla
7    }
8 }
```

Problemas? Velocidad variable de internet, lo que provoca:

- pausas
- pérdida de fluidez

Solución:

- Reprogramar `reproducir`
- Leer hasta N cuadros por adelantado
- No más de N
- Si la red se pone lenta, baja la reserva de cuadros

Productor/Consumidor

Implementación: 2 *threads*, un lector de cuadros (put) y un proyector (get).

```
1 void reproducirVideo() {
2     Buffer *buf= nuevoBuffer(30*5);
3     pthread_t t;
4     pthread_create(&t, NULL, proyector, buf);
5     lector(buf);
6     pthread_join(t, NULL);
7 }
```

Productor/Consumidor

```
1  void reproducirVideo() {
2      Buffer *buf= nuevoBuffer(30*5);
3      pthread_t t;
4      pthread_create(&t, NULL, proyector, buf);
5      lector(buf);
6      pthread_join(t, NULL);
7  }
8  void lector(Buffer *buf) {
9      for (;;) {
10       Cuadro *cuadro= leerCuadro();
11       put(buf, cuadro);
12       if (cuadro==NULL)
13         break;
14     }}
15 void *proyector(void *ptr) { // porque se usa en pthread_create
16     Buffer *buf= ptr;
17     for (;;) {
18       Cuadro *cuadro= get(buf);
19       if (cuadro==NULL)
20         break;
21       mostrarCuadro(cuadro);
22     }
23     return NULL;}
```

Productor/Consumidor

```
1  void reproducirVideo() {
2      Buffer *buf= nuevoBuffer(30*5);
3      pthread_t t;
4      pthread_create(&t, NULL, productor, buf);
5      consumidor(buf);
6      pthread_join(t, NULL);
7  }
8  void productor(Buffer *buf) {
9      for (;;) {
10       Item *cuadro= producir();
11       put(buf, cuadro);
12       if (cuadro==NULL)
13         break;
14     }}
15 void *consumidor(void *ptr) { // porque se usa en pthread_create
16     Buffer *buf= ptr;
17     for (;;) {
18       Item *cuadro= get(buf);
19       if (cuadro==NULL)
20         break;
21       consumir(cuadro);
22     }
23     return NULL;}
```

Implementación incorrecta de buffer

```
1 Buffer *nuevoBuffer(int size) {
2     Buffer *buf= (Buffer*)malloc(sizeof(Buffer));
3     buf->size= size;
4     buf->array= (Item**)malloc(size*sizeof(Item*));
5     buf->in= buf->out= buf->cnt= 0;
6     return buf;
7 }
```

```
1 typedef struct {
2     Item **array;
3     int size, in, out, cnt;
4 } Buffer;
```

```
1 Item *get(Buffer *b){
2     while(b->cnt == 0)
3         ; //busy waiting
4     Item *it = b->array[b->out];
5     b->out= b->out  % b->size;
6     b->cnt--;
7     return it:
8 }
```

```
1 void put(Buffer *b, Item *it){
2     while(b->cnt == b->size)
3         ; // busy waiting
4     b->array[b->in] = it;
5     b->in = (b->in + 1)  % b->size
6     b->cnt ++;
7 }
```

¿Es correcta? ¿Es eficiente?

Condiciones

Una condición permite esperar eficientemente (evitando busy waiting) hasta que ocurra un evento. Las operaciones son:

```
1  #include <pthread.h>
2  pthread_cond_t cond;
3  //Creación
4  int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *a);
5  //Destrucción y liberar recursos
6  int pthread_cond_destroy(pthread_cond_t *cond);
7  // Espera hasta que se cumpla la condición
8  int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *m);
9  // despierta a todos
10 int pthread_cond_broadcast(pthread_cond_t *cond);
11 // despierta a solo uno
12 int pthread_cond_signal(pthread_cond_t *cond);
```

Implementación correcta de buffer

```
Buffer *nuevoBuffer(int size) {
    Buffer *buf= (Buffer*)malloc(sizeof(Buffer));
    buf->size= size;
    buf->array= (Item**)malloc(size*sizeof(Item*));
    buf->in= buf->out= buf->cnt= 0;
    pthread_mutex_init(&buf->m, NULL):
    pthread_mutex_cond(&buf->cond, NULL):
    return buf;
}
```

```
typedef struct {
    Item **array;
    int size, in, out, cnt;
    pthread_mutex_t m;
    pthread_cond_t cond;
} Buffer;
```

```
Item *get(Buffer *b){
    pthread_mutex_lock(&b->m, NULL);
    while(b->cnt == 0)
        pthread_cond_wait(&b->cond, NULL
        ↪ );
    Item *it = b->array[b->out];
    b->out= b->out % b->size;
    b->cnt--;
    pthread_cond_broadcast(&b->cond);
    pthread_mutex_unlock(&b->m);
    return it:
}
```

```
void put(Buffer *b, Item *it){
    pthread_mutex_lock(&b->m, NULL);
    while(b->cnt == b->size)
        pthread_cond_wait(&b->cond, NULL
        ↪ );
    b->array[b->in] = it;
    b->in = (b->in + 1) % b->size
    b->cnt ++;
    pthread_cond_broadcast(&b->cond);
    pthread_mutex_unlock(&b->m);
}
```

Implementación incorrecta de buffer (signal)

```
Buffer *nuevoBuffer(int size) {
    Buffer *buf= (Buffer*)malloc(sizeof(Buffer));
    buf->size= size;
    buf->array= (Item**)malloc(size*sizeof(Item*));
    buf->in= buf->out= buf->cnt= 0;
    pthread_mutex_init(&buf->m, NULL):
    pthread_mutex_cond(&buf->cond, NULL):
    return buf;
}
```

```
typedef struct {
    Item **array;
    int size, in, out, cnt;
    pthread_mutex_t m;
    pthread_cond_t cond;
} Buffer;
```

```
Item *get(Buffer *b){
    pthread_mutex_lock(&b->m, NULL);
    while(b->cnt == 0)
        pthread_cond_wait(&b->cond, NULL
        ↪ );
    Item *it = b->array[b->out];
    b->out= b->out % b->size;
    b->cnt--;
    pthread_cond_signal(&b->cond);
    pthread_mutex_unlock(&b->m);
    return it:
}
```

```
void put(Buffer *b, Item *it){
    pthread_mutex_lock(&b->m, NULL);
    while(b->cnt == b->size)
        pthread_cond_wait(&b->cond, NULL
        ↪ );
    b->array[b->in] = it;
    b->in = (b->in + 1) % b->size
    b->cnt ++;
    pthread_cond_signal(&b->cond);
    pthread_mutex_unlock(&b->m);
}
```

Implementación correcta de buffer (signal)

```
Buffer *nuevoBuffer(int size) {
    Buffer *buf= (Buffer*)malloc(sizeof(Buffer));
    buf->size= size;
    buf->array= (Item**)malloc(size*sizeof(Item*));
    buf->in= buf->out= buf->cnt= 0;
    pthread_mutex_init(&buf->m, NULL):
    pthread_mutex_cond(&buf->cond1, NULL):
    pthread_mutex_cond(&buf->cond2, NULL):
    return buf;
}
```

```
typedef struct {
    Item **array;
    int size, in, out, cnt;
    pthread_mutex_t m;
    pthread_cond_t cond1, cond2;
} Buffer;
```

```
Item *get(Buffer *b){
    pthread_mutex_lock(&b->m, NULL);
    while(b->cnt == 0)
        pthread_cond_wait(&b->cond1,
      ↪ NULL);
    Item *it = b->array[b->out];
    b->out= b->out % b->size;
    b->cnt--;
    pthread_cond_signal(&b->cond2);
    pthread_mutex_unlock(&b->m);
    return it:
```

```
void put(Buffer *b, Item *it){
    pthread_mutex_lock(&b->m, NULL);
    while(b->cnt == b->size)
        pthread_cond_wait(&b->cond2,
      ↪ NULL);
    b->array[b->in] = it;
    b->in = (b->in + 1) % b->size
    b->cnt ++;
    pthread_cond_signal(&b->cond1);
    pthread_mutex_unlock(&b->m);
}
```

Uso correcto de wait

Wait siempre debe ir dentro de un ciclo.
Que pasa si hacemos lo siguiente?

```
1 item get(...){
2     lock(...)
3     if(...)
4         wait(...)
5 }
```

Clasificación de Problemas

- Paralelización: no hay dependencia entre los *threads*
- Sincronización: dependencia entre los *threads*
- Problemas clásicos de sincronización: productor/consumidor, cena de filósofos, lector/escritor.