Concurrency
Condition Variables
2º MIEIC

Pedro F. Souto (pfs@fe.up.pt)

March 24, 2020

Introduction

- Locking ensures mutual exclusion
- But sometimes all we need is synchronization
  - E.g waiting for a thead to terminate

```
1   void *child(void *arg) {
2       printf("child\n");
3       // XXX how to indicate we are done?
4       return NULL;
5   }
6
7   int main(int argc, char *argv[]) {
8       printf("parent: begin\n");
9       pthread_t c;
10      Pthread_create(&c, NULL, child, NULL); //
11      // XXX how to wait for child?
12      printf("parent: end\n");
13      return 0;
14  }
```

Synchronizing with Shared Variables

```
1   volatile int done = 0;
2
3   void *child(void *arg) {
4       printf("child\n");
5       done = 1;
6       return NULL;
7   }
8
9   int main(int argc, char *argv[]) {
10      printf("parent: begin\n");
11      pthread_t c;
12      Pthread_create(&c, NULL, child, NULL); // create
13      while (done == 0)
14          ; // spin
15      printf("parent: end\n");
16      return 0;
17  }
```

- But this requires busy-waiting
  - In this case, it is OK not to use locks to access `done`

Sumário

A Solution: Condition Variables

Condition Variable is a queue on which the threads put themselves while waiting for some **condition**

- When another thread changes the state so that the condition is satisfied, it should wake up one (or more) threads waiting on the condition

`libpthread` API

```
pthread_cond_t c = PTHREAD_COND_INITIALIZER;
```

`pthread_cond_wait()`

- The thread must hold the mutex, when calling `pthread_cond_wait()`
- Upon waiting `pthread_cond_wait()` releases the lock
- Upon returning from `pthread_cond_wait()`:
  1. The thread holds the lock
  2. **But** the condition may not be satisfied any more
     - This is known as the **Mesa semantics**

`pthread_cond_signal()`

- Wakes up one thread waiting on the condition variable, if any

`pthread_cond_broadcast()`

- Wakes up all threads waiting on the condition variable, if any

Joining a Thread with Condition Variables

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

What if we did not use **done**?

```
1   void thr_exit() {
2       Pthread_mutex_lock(&m);
3       Pthread_cond_signal(&c);
4       Pthread_mutex_unlock(&m);
5   }
6
7   void thr_join() {
8       Pthread_mutex_lock(&m);
9       Pthread_cond_wait(&c, &m);
10      Pthread_mutex_unlock(&m);
11  }
```

- Can you see what may go wrong?

What if we did not use `done`?

```
1   void thr_exit() {
2       Pthread_mutex_lock(&m);
3       Pthread_cond_signal(&c);
4       Pthread_mutex_unlock(&m);
5   }
6
7   void thr_join() {
8       Pthread_mutex_lock(&m);
9       Pthread_cond_wait(&c, &m);
10      Pthread_mutex_unlock(&m);
11  }
```

- Can you see what may go wrong?
  - `pthread_cond_signal()` has no effect if no thread is waiting
    - Condition variables are not counters

What if `thr_exit()` did not use mutexes?

```
 1:     void thr_exit() {
 2:         done = 1;
 3:         Pthread_cond_signal(&c);
 5:      }
 6:     void thr_join() {
 7:         Pthread_mutex_lock(&m);
 8:         while (done == 0)
 9:             Pthread_cond_signal(&c, &m);
10:         Pthread_mutex_unlock(&m);
11:      }
```

- Can you see what may go wrong?

What if `thr_exit()` did not use mutexes?

```
 1:     void thr_exit() {
 2:         done = 1;
 3:         Pthread_cond_signal(&c);
 5:      }
 6:     void thr_join() {
 7:         Pthread_mutex_lock(&m);
 8:         while (done == 0)
 9:             Pthread_cond_signal(&c, &m);
10:         Pthread_mutex_unlock(&m);
11:      }
```

- Can you see what may go wrong?
- Note that the testing of `done` and the waiting are not atomic anymore

What if `thr_exit()` did not use mutexes?

```
 1:     void thr_exit() {
 2:         done = 1;
 3:         Pthread_cond_signal(&c);
 5:      }
 6:     void thr_join() {
 7:         Pthread_mutex_lock(&m);
 8:         while (done == 0)
 9:             Pthread_cond_signal(&c, &m);
10:         Pthread_mutex_unlock(&m);
11:      }
```

- Can you see what may go wrong?
- Note that the testing of `done` and the waiting are not atomic anymore
- In is not always necessary to hold the lock while calling `pthread_cond_signal()`
  - In any case, holding the lock may be safer than not holding the lock

Sumário

The Problem of the Bounded Buffer

- This is a classical problem in which:

  Producer (one or more) threads "generate" data items and put them on a **queue/buffer**

  Consumer (one or more) threads grab data items from the queue and "consume" them
- If the buffer has an unlimited capacity, the problem is known as the **producer/consumer** problem
- If the buffer has limited capacity, the problem is known as the **bounded buffer**
- Both problems are of very practical importance. E.g. consider the multi-threaded implementation of a web server
  - One of more threads receive the HTTP requests from the clients and put them on a queue
  - One or more threads get the requests from the queue, process them and send back the HTTP responses to the clients

A simple BB Abstract Data Type

```
1   int buffer;
2   int count = 0; // initially, empty
3
4   void put(int value) {
5       assert(count == 0);
6       count = 1;
7       buffer = value;
8   }
9
10  int get() {
11      assert(count == 1);
12      count = 0;
13      return buffer;
14  }
```

- This is a very simple example:
  - The buffer has capacity for only one data item
  - The data item passed through the buffer is only an `int`
    - How could we pass an arbitrary data type?

Using the simple BB Abstract Data Type

```
 1 int loops;                  // initialized somewhere
 2 void *producer(void *arg) {
 3     int i;
 4     for( i = 0; i < loops: i++ ) {
 5         while(count == 1);
 6         put(i);
 7     }
 8 }
 9 void *consumer(void *arg) {
10     int i;
11     while(1) {
12         while( count == 0);
13         int tmp = get();
14         printf("%d\n", tmp);
15     }
16 }
```

- The problem with the previous ADT is that it is not **thread-safe**. I.e.
  - It may suffer race conditions when used by more than one thread

Thread-safe BB ADT: 1st try

```
1  int loops; // must initialize somewhere...
2  cond_t  cond;
3  mutex_t mutex;
4
5  void *producer(void *arg) {
6      int i;
7      for (i = 0; i < loops; i++) {
8          Pthread_mutex_lock(&mutex);            // p1
9          if (count == 1)                        // p2
10             Pthread_cond_wait(&cond, &mutex);  // p3
11         put(i);                                // p4
12         Pthread_cond_signal(&cond);            // p5
13         Pthread_mutex_unlock(&mutex);          // p6
14     }
15 }
16
17 void *consumer(void *arg) {
18     int i;
19     for (i = 0; i < loops; i++) {
20         Pthread_mutex_lock(&mutex);            // c1
21         if (count == 0)                        // c2
22             Pthread_cond_wait(&cond, &mutex);  // c3
23         int tmp = get();                       // c4
24         Pthread_cond_signal(&cond);            // c5
25         Pthread_mutex_unlock(&mutex);          // c6
26         printf("%d\n", tmp);
27     }
28 }
```

- This has a race-condition. Can you see it?

Use `while` rather than `if`

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

- The issue is that upon return from `Pthread_cond_wait()` the condition may not be satisfied any more
  - Thead c_2 made it false again
- Recheck the condition rather than taking it for granted

Thread-safe BB ADT: 2nd try

```
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);             // p1
8           while (count == 1)                      // p2
9               Pthread_cond_wait(&cond, &mutex); // p3
10          put(i);                                 // p4
11          Pthread_cond_signal(&cond);             // p5
12          Pthread_mutex_unlock(&mutex);           // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);             // c1
20          while (count == 0)                      // c2
21              Pthread_cond_wait(&cond, &mutex); // c3
22          int tmp = get();                        // c4
23          Pthread_cond_signal(&cond);             // c5
24          Pthread_mutex_unlock(&mutex);           // c6
25          printf("%d\n", tmp);
26      }
27  }
```

Thread-safe BB ADT: 2nd try

```
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);            // p1
8           while (count == 1)                     // p2
9               Pthread_cond_wait(&cond, &mutex); // p3
10          put(i);                                // p4
11          Pthread_cond_signal(&cond);            // p5
12          Pthread_mutex_unlock(&mutex);          // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          while (count == 0)                     // c2
21              Pthread_cond_wait(&cond, &mutex); // c3
22          int tmp = get();                       // c4
23          Pthread_cond_signal(&cond);            // c5
24          Pthread_mutex_unlock(&mutex);          // c6
25          printf("%d\n", tmp);
26      }
27  }
```

- But this still has a race condition. Can you spot it?

Thread-safe BB ADT: 2nd try

```
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);            // p1
8           while (count == 1)                     // p2
9               Pthread_cond_wait(&cond, &mutex); // p3
10          put(i);                                // p4
11          Pthread_cond_signal(&cond);            // p5
12          Pthread_mutex_unlock(&mutex);          // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          while (count == 0)                     // c2
21              Pthread_cond_wait(&cond, &mutex); // c3
22          int tmp = get();                       // c4
23          Pthread_cond_signal(&cond);            // c5
24          Pthread_mutex_unlock(&mutex);          // c6
25          printf("%d\n", tmp);
26      }
27  }
```

- Think about waking up the wrong thread!

Thread-safe BB ADT: 2nd race

T_{c1}	State	T_{c2}	State	T_p	State	**Count**	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Thread-safe BB ADT: Fixes to 2nd race?

How to fix this?

Thread-safe BB ADT: Fixes to 2nd race?

How to fix this?

Use **`pthread_cond_broadcast()`**

- Sometimes threads are awoken unnecessarily

Thread-safe BB ADT: Fixes to 2nd race?

How to fix this?

Use `pthread_cond_broadcast()`

- Sometimes threads are awoken unnecessarily

Use two condition variables

- One for empty buffer, to be used by producers;
- One for filled buffer, to be used by consumers;

Thread-safe BB ADT: No races

```
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);
8           while (count == 1)
9               Pthread_cond_wait(&empty, &mutex);
10          put(i);
11          Pthread_cond_signal(&fill);
12          Pthread_mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);
20          while (count == 0)
21              Pthread_cond_wait(&fill, &mutex);
22          int tmp = get();
23          Pthread_cond_signal(&empty);
24          Pthread_mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```