Universidade Estadual de Campinas
Instituto de Computação

MC504 Sistemas Operacionais

T21 Condition Variables

Referência principal
Ch.30 of *Operating Systems: Three Easy Pieces* by Remzi and Andrea Arpaci-Dusseau (pages.cs.wisc.edu/~remzi/OSTEP/)

Discutido em classe em 10 de outubro de 2018

Arthur João Catto, PhD

2º semestre de 2018

Waiting for a condition to hold

- Locks aren't the only primitives needed to build concurrent applications.
- Often a thread wishes to check whether a ***condition*** holds before continuing execution.
  - For example, a parent thread might want to wait until a child thread terminates before continuing.
  - This is frequently modelled as a `join()`.
- How should such a wait be implemented?

How would a parent thread wait for its child?

```c
#include <stdio.h>

void *child(void *arg) {
    printf("child\n") ;
    // XXX how to indicate we are done?
    return NULL;
}

int main(int arge, char *argv[]) {
    printf("parent: begin\n") ;
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n") ;
    return 0;
}
```

Expected output

```
parent: begin
child
parent: end
```

Trying to use a shared variable

```
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int arge, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

Does the shared variable solve the problem?

- The shared variable approach will often work.
- However, it is very inefficient as the parent spins and wastes CPU time.
- Instead, we would like to have some way to put the parent to sleep until the condition we are waiting for (e.g. the termination of the child thread) comes true.

How to wait for a condition?

- In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding.
- The simple approach – just spinning until the condition becomes true – is grossly inefficient, wastes CPU cycles, and in some cases, can be incorrect.
- Thus, **how should a thread wait for a condition?**

Condition Variables

- A condition variable is an explicit queue that threads can put themselves on when some state of execution is not as desired.
- Some other thread, when it changes that state, can then wake one (or more) of those waiting threads and thus allow them to continue.
- The approach goes back to Dijkstra's concept of "*private semaphores*" (1968).
- A similar idea was later named a "*condition variable*" by Hoare in his work on monitors (1974).

POSIX[1] Operations on Condition Variables

- The examples below refer to the wrapper functions declared in `mythreads.h`.
- Condition variables are declared by writing something like
  `pthread_cond_t c;`
- A condition variable can be declared and initialized at the same time by
  `pthread_cond_t cv = PTHREAD_COND_INITIALIZER;`
- A thread can wait on a condition variable by calling
  `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);`
- A thread can signal on a condition variable by calling
  `pthread_cond_signal(pthread_cond_t *c);`

[1]The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

The semantics of `wait()` and `signal()`

- You may have noticed that `wait()` also takes a *mutex* as a parameter.
  - It assumes that the calling thread holds this mutex at the point of call.
  - The responsibility of `wait()` is to release the lock and put the calling thread to sleep (atomically).
  - Releasing the lock is mandatory to enable some other thread to change the state, make the condition true and then `signal()` on it.
  - After some other thread's signal, `wait()` will re-acquire the lock before waking up the thread and returning to the caller.
  - This complexity is required to prevent certain race conditions from occurring when a thread is trying to put itself to sleep.
  - Let's take a look at an attempt at solving the "join problem" to understand this better.

Trying to solve the “join problem” using a condition variable

- Let us assume that the two required actions on our previous draft are implemented by
  - a `thr_join()` function that will be called by the parent
  - a `thr_exit()` function that will be called by the child

```
void *child(void *arg) {
    printf("child\n") ;
    thr_exit( );
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Trying to solve the "join problem" using a condition variable

- Let us use a shared variable `done` to indicate that the child has finished.
- Initially, `done` will be `zero`. This will be changed to `one` by the child.
- A condition variable `cv` will be used to synchronize the parent and child threads.
  - The parent will **wait on cv** for the child to finish.
  - The child will **signal on cv** to indicate it has finished.
- Finally, to prevent race conditions, we will use a `mutex m` to protect every access to `done`.

Trying to solve the “join problem” using a condition variable

- Now let us try to implement `thr_exit()` and `thr_join()`, using the shared variable `done`, the condition variable `cv` and a mutex `m`.

```
int done = 0;
mutex_t m = PTHREAD_MUTEX_INITIALIZER;
cond_t cv = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    mutex_lock(&m);
    done = 1;
    cond_signal(&cv);
    mutex_unlock(&m);
}

void thr_join() {
    mutex_lock(&m);
    if (done == 0)
        cond_wait(&cv, &m);
    mutex_unlock(&m);
}
```

Analyzing the proposed solution to the "join problem"

- Consider that there is only one processor and examine two cases
  a. The parent creates the child and carries on, thus calling `thr_join()`.
  b. The child is created and runs immediately, thus printing its message and calling `thr_exit()`.

- Now, a few questions may have arisen, such as
  - Is `done` really necessary?
  - Is the `mutex` really necessary?
  - Is it the solution we wanted?

- Let us examine each one of them in turn.

Trying to get rid of done

- Assume that there is no done…

```
void thr_exit() {
    mutex_lock(&m);
    cond_signal(&c);
    mutex_unlock(&m);
}

void thr_join() {
    mutex_lock(&m);
    cond_wait(&c, &m);
    mutex_unlock(&m);
}
```

```
int done = 0;
mutex_t m = PTHREAD_MUTEX_INITIALIZER;
cond_t cv = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    mutex_lock(&m);
    done = 1;
    cond_signal(&cv);
    mutex_unlock(&m);
}

void thr_join() {
    mutex_lock(&m);
    if (done == 0)
        cond_wait(&cv, &m);
    mutex_unlock(&m);
}
```

- Now, re-examine case b, where the child is created and runs immediately, thus printing its message and calling `thr_exit()`.
- Does it work?

Trying to get rid of mutex

- Assume that there is no mutex…

```
4.  void thr_exit() {
5.      done = 1;
6.      cond_signal(&c);
7.  }

8.  void thr_join() {
9.      if (done == 0)
10.         cond_wait(&c, &m);
11. }
```

```
int done = 0;
mutex_t m = PTHREAD_MUTEX_INITIALIZER;
cond_t cv = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    mutex_lock(&m);
    done = 1;
    cond_signal(&cv);
    mutex_unlock(&m);
}

void thr_join() {
    mutex_lock(&m);
    if (done == 0)
        cond_wait(&cv, &m);
    mutex_unlock(&m);
}
```

- Now, re-examine case a, where the parent creates the child and continues to run.
- Assume that after checking (done == 0) on line 9, the parent is interrupted just before calling cond_wait()
- Does it work? Or is there a race condition?

Unfortunately, there is still one bug...

- By now, you should have appreciated the need for `done` and the `mutex`.
- Unfortunately, our solution still has one bug.
- To make it evident, we will study a slightly more complicated example: the ***producer/consumer*** or ***bounded buffer*** problem.
- This problem was posed by Dijkstra in 1972 and led to the invention of another synchronization primitive, the ***generalized semaphore***, which we will study later.

The *producer/consumer* or *bounded buffer* problem

- Imagine one or more producer threads and one or more consumer threads.
  - Producers generate data items and place them in a buffer;
  - Consumers grab items from the buffer and use them somehow.
- Because the bounded buffer is a shared resource, access to it must be synchronized, to avoid a race condition.
- Let us try to understand this problem better…
  - We will need a shared buffer, into which a producer puts data, and out of which a consumer takes data.
    - We will use a one-slot buffer, able to hold just one integer, for simplicity. We will relax this constraint later.
  - We will also use two inner routines to put a value into the shared buffer, and to get a value out of the buffer, as shown in the next slide.

The put() and get() functions

- There are two global variables:
  - The buffer and the count of numbers in it.
- put asserts that buffer is not full and puts value in it.
- get asserts that buffer is not empty and returns the value in it.
- In case an assertion fails, an exception will be raised.

```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get(void) {
    assert(count == 1);
    count = 0;
    return buffer;
}
```

The producer and consumer threads (version 1)

- There are two kinds of threads:
  - producer, that puts an integer in the shared buffer loops times;
  - consumer, that gets data out of the shared buffer (forever) and prints out the values it receives.
- An application is supposed to create one or more producers and one or more consumers.
- What do you think of this design? Is it correct? Does it work well?

```
void *producer(void *arg) {
    int loops = (int) arg;
    for (int i = 0; i < loops; i++) {
        put(i);
    }
    return NULL;
}

void *consumer(void *arg) {
    while (1) {
        int tmp = get();
        printf("%d\n", tmp);
    }
    return NULL;
}
```

Unfortunately, version 1 does not work. . .

- You certainly noticed that version 1 has flaws.
  - There is a race condition on **count** and **buffer**, which can be prevented by a **mutex**.
  - Both **producer** and **consumer** may trigger an exception by calling **put** or **get** at the “wrong” time, but their accesses can be synchronized using a **condition variable**.
- Let us create a version 2 by providing these two adjustments.

```c
int buffer;
int count = 0; // initially, empty

void put(int value) {
   assert(count == 0);
   count = 1;
   buffer = value;
}

int get(void) {
   assert(count == 1);
   count = 0;
   return buffer;
}
```

```c
void *producer(void *arg) {
   int loops = (int) arg;
   for (int i = 0; i < loops; i++) {
      put(i);
   }
   return NULL;
}

void *consumer(void *arg) {
   while (1) {
      int tmp = get();
      printf("%d\n", tmp);
   }
   return NULL;
}
```

The producer and consumer threads (version 2)

```
void *producer(void *arg) {
    int loops = *((int *)arg);
    for (int i = 0; i < loops; i++) {
        mutex_lock(&mutex);              // p1
        if (count == 1)                  // p2
            cond_wait(&cond, &mutex);    // p3
        put(i);                          // p4
        cond_signal(&cond);              // p5
        mutex_unlock(&mutex);            // p6
    }
    return NULL;
}
```

```
void *consumer(void *arg) {
    int loops = *((int *)arg);
    for (int i = 0; i < loops; i++) {
        mutex_lock(&mutex);              // c1
        if (count == 0)                  // c2
            cond_wait(&cond, &mutex);    // c3
        int tmp = get();                 // c4
        cond_signal(&cond);              // c5
        mutex_unlock(&mutex);            // c6
        printf("%d\n", tmp);
    }
    return NULL;
}
```

- Let us do a thread trace to examine the behavior of this proposal.

Thread trace of version 2 with 2 consumers

t	T_{C1}	State	T_{C2}	State	T_P	State	count	Comment
1	c1	Running		Ready		Ready	0	
2	c2	Running		Ready		Ready	0	
3	c3	Asleep		Ready		Ready	0	Buffer empty; must sleep
4		Asleep		Ready	p1	Running	0	
5		Asleep		Ready	p2	Running	0	
6		Asleep		Ready	p4	Running	1	Buffer now full
7		Ready		Ready	p5	Running	1	T_{C1} awoken
8		Ready		Ready	p6	Running	1	
9		Ready		Ready	p1	Running	1	
10		Ready		Ready	p2	Running	1	
11		Ready		Ready	p3	Asleep	1	Buffer full; must sleep
12		Ready	c1	Running		Asleep	1	T_{C2} sneaks in ...
13		Ready	c2	Running		Asleep	1	
14		Ready	c4	Running		Asleep	0	... and grabs the data
15		Ready	c5	Running		Ready	0	T_P awoken
16		Ready	c6	Running		Ready	0	
17		Ready	c1	Running		Ready	0	
18		Ready	c2	Running		Ready	0	
19		Ready	c3	Asleep		Ready	0	Buffer empty; must sleep
20	c4	Running		Ready		Ready	0	Oops! No data...

What went wrong in version 2 and how to fix it?

- The problem was caused by the assumption that on returning from **wait** the buffer would be **not full** (**producer**) or **not empty** (**consumer**).
- As we have seen this may not be the case when there area multiple producers or consumers.
- The solution is to repeat the test for fullness or emptiness of the **buffer** when returning from **wait**, i.e. let us use a **while** instead of an **if** in lines **p2** (**producer**) and **c2** (**consumer**).
- This will lead us to version 3, shown in the next slide.
- Is version 3 correct? Does it work in any situation?

The producer and consumer threads (version 3)

```
void *producer(void *arg) {
    for (int i = 0; i < LOOPS; i++) {
        mutex_lock(&mutex);                 // p1
        while (count == 1)                  // p2
            cond_wait(&cond, &mutex);       // p3
        put(i);                             // p4
        cond_signal(&cond);                 // p5
        mutex_unlock(&mutex);               // p6
    }
    return NULL;
}
```

```
void *consumer(void *arg) {
    for (int i = 0; i < LOOPS; i++) {
        mutex_lock(&mutex);                 // c1
        while (count == 0)                  // c2
            cond_wait(&cond, &mutex);       // c3
        int tmp = get();                    // c4
        cond_signal(&cond);                 // c5
        mutex_unlock(&mutex);               // c6
        printf("%d\n", tmp);
    }
    return NULL;
}
```

- Let us resort again to a thread trace to examine the behavior of this proposal.

Thread trace of version 3 with 2 consumers

t	T_{C1}	State	T_{C2}	State	T_P	State	count	Comment
1	c1	Running		Ready		Ready	0	
2	c2	Running		Ready		Ready	0	
3	c3	Asleep		Ready		Ready	0	Buffer empty; must sleep
4		Asleep	c1	Running		Ready	0	
5		Asleep	c2	Running		Ready	0	
6		Asleep	c3	Asleep		Ready	0	Buffer empty; must sleep
7		Asleep		Asleep	p1	Running	0	
8		Asleep		Asleep	p2	Running	0	
9		Asleep		Asleep	p4	Running	1	Buffer now full
10		Ready		Asleep	p5	Running	1	T_{C1} awoken
11		Ready		Asleep	p6	Running	1	
12		Ready		Asleep	Pi	Running	1	
13		Ready		Asleep	p2	Running	1	
14		Ready		Asleep	p3	Asleep	1	Buffer full; must sleep
15	c2	Running		Asleep		Asleep	1	Recheck condition
16	c4	Running		Asleep		Asleep	0	T_{C1} grabs data
17	c5	Running		Ready		Asleep	0	Oops! Woke T_{C2} up
18	c6	Running		Ready		Asleep	0	
19	cl	Running		Ready		Asleep	0	
20	c2	Running		Ready		Asleep	0	
21	c3	Asleep		Ready		Asleep	0	Buffer empty; must sleep
22		Asleep	c2	Running		Asleep	0	
23		Asleep	c3	Asleep		Asleep	0	Everyone asleep...

What went wrong in version 3 and how to fix it?

- This time the problem was caused by waking up the “wrong” thread.
- As it may be clear now, our threads are interested in two different conditions: the **producers** need to wait for **buffer not full**, while the **consumers** need to wait for **buffer not empty**.
- Thus, we will use two different condition variables: **not_empty** and **not_full**.
- This will lead us to version 4, shown in the next slide.

The producer and consumer threads (version 4)

```
cond_t not_empty, not_full;
mutex_t mutex;
```

```
void *producer(void *arg) {
    for (int i = 0; i < LOOPS; i++) {
        mutex_lock(&mutex);                    // p1
        while (count == 1)                     // p2
            cond_wait(&not_full, &mutex);      // p3
        put (i);                               // p4
        cond_signal(&not_empty);               // p5
        mutex_unlock(&mutex);                  // p6
    }
    return NULL;
}
```

```
void *consumer(void *arg) {
    for (int i = 0; i < LOOPS; i++) {
        mutex_lock(&mutex);                    // c1
        while (count == 0)                     // c2
            cond_wait(&not_empty, &mutex);     // c3
        int tmp = get();                       // c4
        cond_signal(&not_full);                // c5
        mutex_unlock(&mutex);                  // c6
        printf("%d\n", tmp);
    }
    return NULL;
}
```

- Study the code and then answer…
  - Is it correct? Does it work in any situation?
  - Why two condition variables but just one mutex?

The Final Producer/Consumer Solution

- Our last version works but it is not a general solution to the problem.
- To increase concurrency and efficiency, let us enlarge the buffer, turning it into an array of integers.
- This asks for only a couple of simple changes to our current design:
  - The **buffer** structure must be reflected on both **put()** and **get()**.
  - The **while** statement of the producer (line **p2**), must cater for the new **buffer** (shown in the next slide).

```
#define MAX 100

int buffer[MAX];
int fill_ptr = 0;
int use_ptr = 0;
int count = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get(void) {
    int tmp = buffer[use_ptr]
    use_ptr = (use_ptr +1) % MAX;
    count—-;
    return tmp;
}
```

The producer and consumer threads (final version)

```
cond_t not_empty, not_full;
mutex_t mutex;
```

```
void *producer(void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&mutex);                      // p1
        while (count == MAX)                     // p2
            cond_wait(&not_full, &mutex);        // p3
        put (i);                                 // p4
        cond_signal(&not_empty);                 // p5
        mutex_unlock(&mutex);                    // p6
    }
    return NULL;
}
```

```
void *consumer(void *arg) {
    for (int i = 0; i < loops; i++) {
        mutex_lock(&mutex);                      // c1
        while (count == 0)                       // c2
            cond_wait(&not_empty, &mutex);       // c3
        int tmp = get();                         // c4
        cond_signal(&not_full);                  // c5
        mutex_unlock(&mutex);                    // c6
        printf("%d\n", tmp);
    }
    return NULL;
}
```

Covering Conditions

- Consider a multi-threaded memory allocation library, from which this code snippet was taken.
- Assume that there are no **bytesLeft** and that two threads call **allocate(100)** and **allocate(10)**, respectively.
  - As a result, both will wait on **c** (line 7).
- Now assume that a third thread calls **free(50)**.
  - If the first thread catches the signal, everyone will remain asleep.
- If we turn **cond_signal()** (line 16) into **code_broadcast()**, all sleeping threads could have a chance to awake and continue.

```
1.  int bytesLeft = MAX_HEAP_SIZE;

2.  cond_t c;
3.  mutex_t m;

4.  void *allocate(int size) {
5.      mutex_lock(&m);
6.      while (bytesLeft < size)
7.          cond_wait(&c, &m);
8.      void *ptr = ...;        // get mem from heap
9.      bytesLeft -= size;
10.     mutex_unlock(&m);
11.     return ptr;
12. }

13. void free(void *ptr, int size) {
14.     mutex_lock(&m);
15.     bytesLeft += size;
16.     cond_signal(&c);        // whom to signal??
17.     mutex_unlock(&m);
18. }
```