Operating System (OS)
CS232

Concurrency: Condition Variables

Dr. Muhammad Mobeen Movania
Dr. Muhammad Saeed

Condition Variables

- Issues with Mutex Locks
- Condition Variables
- PThread Condition Variables API
- Producer/Consumer problem
- Need of synchronization
- Summary

Example - Lock Issues

```
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);
13      while (done == 0)
14          ; // spin
15      printf("parent: end\n");
16      return 0;
17  }
```

Issues with Mutex Locks

- Locks provide basic synchronization (mutual exclusion) which does not work in all cases (as discussed in last slide)
  - Spin waiting wastes a lot of CPU cycles
  - What if a thread wanted to wait for a condition?

- Solution
  - Condition variables

Condition Variables

- Are provided by OS Kernel
- Provide two operations
  - wait() – puts a thread in a queue of waiting threads
  - signal() – informs one or more waiting threads
- If a thread wants to be informed of a change in state of a CV
  - it calls wait() on CV and it will be put in a queue associated with that CV
- If a thread wants to inform other threads of a change it CV
  - it calls signal() on CV
  - The OS will awake one (or more) threads waiting in the queue associated with that CV

PThread Condition Variable API

```
pthread_cond_t c;   // initialization

//wait function
pthread_cond_wait(pthread_cond_t *c,
                  pthread_mutex_t *m);

//signal function
pthread_cond_signal(pthread_cond_t *c);
```

Example Code

```
pthread_cond_t cv;
pthread_mutex_t mutex;
long long  balance = 50000000;
```

```
int main (){
   pthread_t tid;
   pthread_cond_init(&cv, NULL);
   pthread_mutex_init(&mutex, NULL);
   pthread_create (&tid, NULL, debit, NULL)

   pthread_mutex_lock(&mutex);
   if (balance != 0)
      pthread_cond_wait(&cv, &mutex);
   pthread_mutex_unlock(&mutex);

   printf("balance = %lld \n", balance);
   pthread_mutex_destroy(&mutex);
   pthread_cond_destroy(&cv);
   return 0;
}
```

```
void* debit(void* arg){
    while(balance > 0){
        pthread_mutex_lock(&mutex);
         balance--;
        pthread_mutex_unlock(&mutex);
    }
    pthread_cond_signal(&cv);
    return NULL;
}
```

```
balance = 0
```

Producer-Consumer

- A classic synchronization problem

- One (or more) Producer thread(s) put values in a buffer
- One (or more) Consumer threads(s) consume values from the buffer
- The buffer is bounded

- How do we synchronize them?

Needs synchronization

```
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  }
```

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
0       int i;
1       while (1) {
2           int tmp = get();
3           printf("%d\n", tmp);
4       }
5   }
```

Needs synchronization … contd.

- Problems?

```
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  }
```

Example Run (c1,p,c2,c1)

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

Needs synchronization … contd.

- Scenario
  - 1 Producer
  - 2 Consumers
- C0 runs, sleeps.
- P0 runs, produces, sleeps.
- C1 runs, consumes signals the CV.
- Which thread is woken?

```
1   int loops;
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           while (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          while (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  }
```

Example Run (c1,c2,p,c1,c2)

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...

Use 2 CVs

- Solution
  - Use 2 CVs

```
1   cond_t  empty, fill;
2   mutex_t mutex;
3
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  }
```

Multiple buffers

```
1   int buffer[MAX];
2   int fill_ptr = 0;
3   int use_ptr  = 0;
4   int count    = 0;
5
6   void put(int value) {
7       buffer[fill_ptr] = value;
8       fill_ptr = (fill_ptr + 1) % MAX;
9       count++;
10  }
11
12  int get() {
13      int tmp = buffer[use_ptr];
14      use_ptr = (use_ptr + 1) % MAX;
15      count--;
16      return tmp;
17  }
```

```
1   cond_t empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);
8           while (count == MAX)
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  }
```

Covering conditions

- Solution
  - Broadcast signal

```
1   // how many bytes of the heap are free?
2   int bytesLeft = MAX_HEAP_SIZE;
3
4   // need lock and condition too
5   cond_t  c;
6   mutex_t m;
7
8   void *
9   allocate(int size) {
10      Pthread_mutex_lock(&m);
11      while (bytesLeft < size)
12          Pthread_cond_wait(&c, &m);
13      void *ptr = ...; // get mem from heap
14      bytesLeft -= size;
15      Pthread_mutex_unlock(&m);
16      return ptr;
17  }
18
19  void free(void *ptr, int size) {
20      Pthread_mutex_lock(&m);
21      bytesLeft += size;
22      Pthread_cond_signal(&c); // whom to signal??
23      Pthread_mutex_unlock(&m);
24  }
```

Summary

- We have seen a new synchronization primitive called a condition variable
- Unlike locks, condition variable does not spin wait, instead, it provides two function wait and signal
  - wait() call puts the calling thread in a wait queue associated with the condition variable
  - signal() call awakes one or more waiting thread in the condition variable waiting queue
- Condition variables enable us to neatly solve producer/consumer problem, as well as covering conditions.