并发：条件变量

邰颖
南京大学
智能科学与技术学院

条件变量

•在许多情况下，线程在继续执行之前需要**检查某个条件（condition）是否成立**。

•举例：

- 父线程可能希望检查子线程是否已经**完成（completed）**。
- 这种情况通常称作线程的 **join()** 操作。

条件变量（续）

一个父线程等待子线程完成的示例

```c
// 子线程执行的函数
void *child(void *arg) {
    printf("child\n");
    // XXX 如何通知我们已完成任务?
    return NULL;
}

// 主线程（父线程）的main函数
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    Pthread_create(&c, NULL, child, NULL); // 创建子线程
    // XXX 如何等待子线程完成?
    printf("parent: end\n");
    return 0;
}
```

我们期望看到的运行结果

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

父线程等待子线程完成：基于自旋的实现方式

- 这是极其**低效的**实现方式，因为父线程处于自旋状态，**浪费了 CPU 时间**

```c
volatile int done = 0;    // 使用 volatile 修饰的共享变量

void *child(void *arg) {
    printf("child\n");
    done = 1;             // 子线程设置 done 为 1
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL);  // 创建子线程
    while (done == 0)                       // 父线程忙等（自旋）
        ;
    printf("parent: end\n");
    return 0;
}
```

// 让主线程能及时感知子线程的修改

如何等待某个条件变量

- 什么是条件变量（**Condition Variable**）？
  - 是一个**显式队列**：当执行状态（即某个条件）**不满足**时，线程可以将自己加入该队列，等待该条件变为满足状态。
    - **等待条件（Waiting on the condition）**：线程将自己加入等待队列，直到被其他线程发出信号唤醒；避免线程自旋等待
    - **条件唤醒（Signaling on the condition）**：另一个线程在**改变条件状态**后发出信号；信号将唤醒一个正在等待的线程，使其恢复执行

条件变量的定义与常用操作

- 声明条件变量

```
pthread_cond_t c;  // 声明条件变量
```

  - 正确的初始化是必需的

- 操作（POSIX调用）

```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);  // wait()
pthread_cond_signal(pthread_cond_t *c);  // signal()
```

  - **wait()** 调用需要传入一个已锁定的互斥锁（mutex）作为参数
    - wait() 调用会释放互斥锁并使当前线程进入休眠
    - 当线程被唤醒时，必须重新获得锁

父线程等待子线程完成：使用条件变量实现join

```c
int done = 0;  // 共享变量，用于表示子线程是否完成任务
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;  // 互斥锁
pthread_cond_t c = PTHREAD_COND_INITIALIZER;  // 条件变量

void thr_exit() {
    pthread_mutex_lock(&m);  // 锁定互斥锁
    done = 1;  // 设置done为1，表示子线程已经完成
    pthread_cond_signal(&c);  // 向父线程发送信号，通知子线程已完成
    pthread_mutex_unlock(&m);  // 解锁互斥锁
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();  // 子线程完成后调用thr_exit通知父线程
    return NULL;
}

void thr_join() {
    pthread_mutex_lock(&m);  // 锁定互斥锁
    while (done == 0)  // 如果done为0，表示子线程还未完成
        pthread_cond_wait(&c, &m);  // 等待条件变量c的信号
    pthread_mutex_unlock(&m);  // 解锁互斥锁
}
```

```c
int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;

    // 创建子线程，执行child函数
    Pthread_create(&p, NULL, child, NULL);

    // 父线程调用thr_join()，等待子线程发出完成信号
    thr_join();

    // 子线程完成任务后，父线程继续执行
    printf("parent: end\n");
    return 0;
}
```

父线程等待子线程完成：使用条件变量

- **父线程（Parent）：**
  - 创建子线程，并继续运行。
  - 调用thr_join()，等待子线程执行完毕：
    - 获取互斥锁（lock）。
    - 检查子线程是否已完成（通过变量done）。
    - 如果子线程尚未完成，通过调用wait()让自己进入睡眠状态，等待子线程的信号。
    - 释放互斥锁。
- **子线程（Child）：**
  - 输出消息 "child"。
  - 调用thr_exit()来唤醒等待的父线程：
    - 获取互斥锁。
    - 设置状态变量done。
    - 向父线程发出信号（signal），将其唤醒。

状态变量done的重要性

- 未使用状态变量 done 的 thr_exit() 和 thr_join() 实现：

```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

// 子线程调用thr_exit()

// 父线程调用thr_join()

- 设想以下情况：**子线程立刻运行**
  - 子线程会立即发出信号（signal），但此时父线程**尚未进入等待状态（wait）**。
  - 当父线程随后运行时，它调用wait()进入睡眠，但此时子线程的信号已经发送完成，**不会再有线程唤醒它**。
  - 结果导致父线程被**永久阻塞**。

另一个糟糕的实现方式：在发信号和等待时都不加锁

- 该代码存在一个隐蔽的**竞争条件**：
  - 父线程调用thr_join()时：
    - 父线程首先检查变量done的值。
    - 如果父线程看到done为0，准备调用wait()进入睡眠状态。
    - 但就在父线程调用wait()前的一瞬间被打断，子线程执行。
  - 子线程将状态变量done设置为1并调用signal()发出信号：
    - 此时还没有线程处于等待状态，因此信号无法唤醒任何线程
    - 当父线程恢复执行时，**它进入wait()并永久睡眠**，因为它将永远无法接收到唤醒信号。

```
void thr_exit() {                    // 子线程调用thr_exit()
    done = 1;
    Pthread_cond_signal(&c);
}

void thr_join() {                    // 父线程调用thr_join()
    if (done == 0)
        Pthread_cond_wait(&c);
}
```

生产者 / 消费者（有界缓冲区）问题

- 生产者（**Producer**）
  - •生产数据项。
  - •希望将数据项放入一个缓冲区中。
- **消费者（Consumer）**
  - •从缓冲区中取出数据项并以某种方式**消费**它们。
- **案例：多线程Web服务器**
  - •**生产者**将HTTP请求放入一个任务队列。
  - •**消费者线程**从该队列取出请求并进行处理。

有界缓冲区（Bounded Buffer）其他应用：管道

- 当一个程序的输出通过管道（pipe）传送给另一个程序作为输入时，会使用到有界缓冲区
  - 示例：grep foo file.txt | wc –l
    - grep 进程是生产者（producer）
    - wc 进程是消费者（consumer）
    - 它们之间存在一个**内核中的有界缓冲区（即管道）**
  - 有界缓冲区是一个共享资源，因此需要进行**同步访问**。

put 和get 函数（第1版）：缓冲区存储一个整数

```
1   int buffer;
2   int count = 0; // 初始为空
3
4   void put(int value) {
5       assert(count == 0);
6       count = 1;          // 表示缓冲区已满
7       buffer = value;
8   }
9
10  int get() {
11      assert(count == 1);
12      count = 0;          // 表示缓冲区已空
13      return buffer;
14  }
```

- 仅当**count为0**时才能将数据放入缓冲区
  - 缓冲区**为空**时才可写入数据。
- 仅当**count为1**时才能从缓冲区取出数据
  - 缓冲区**满时**才可读取数据。

实际中，缓冲区状态不满足时**应等待而不是**终止进程（如assert）

生产者/消费者线程（第1版）

```
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) {
10      int i;
11      while (1) {
12          int tmp = get();
13          printf("%d\n", tmp);
14      }
15  }
```

- **生产者（producer）** 将一个整数放入共享缓冲区，共执行loops次。

- **消费者（consumer）** 从共享缓冲区取出数据。

- 需要引入同步机制：如条件变量
  - 当缓冲区满时，让生产者线程等待（阻塞）；
  - 当缓冲区空时，让消费者线程等待（阻塞）；

生产者/消费者：单一条件变量与if语句实现

- 使用一个单独的条件变量 cond 和关联的互斥锁 mutex
- 当系统中只有单个生产者和单个消费者时，这段代码是正确工作的

```
1   int loops; cond_t cond;
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);            // p1
8           if (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  }
```

p1-p3: 生产者等待缓冲区变为空

```
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          if (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);                   // c7
26      }
27  }
```

c1-c3: 消费者等待缓冲区变为满

追踪线程：有问题的方案（问题1）

T_{c1}	状态	T_{c2}	状态	T_p	状态	count	注释
c1	运行		就绪		就绪	0	
c2	运行		就绪		就绪	0	
c3	睡眠		就绪		就绪	0	没数据可取
	睡眠		就绪	p1	运行	0	
	睡眠		就绪	p2	运行	0	
	睡眠		就绪	p4	运行	1	缓冲区现在满了
	就绪		就绪	p5	运行	1	T_{c1} 唤醒
	就绪		就绪	p6	运行	1	
	就绪		就绪	p1	运行	1	
	就绪		就绪	p2	运行	1	
	就绪		就绪	p3	睡眠	1	缓冲区满了，睡眠
	就绪	c1	运行		睡眠	1	T_{c2} 插入……
	就绪	c2	运行		睡眠	1	
	就绪	c4	运行		睡眠	0	……抓取了数据
	就绪	c5	运行		就绪	0	T_p 唤醒
	就绪	c6	运行		就绪	0	
c4	运行		就绪		就绪	0	啊！没数据

- 两个消费者（T_{c1}和T_{c2}）
- 一个生产者（T_p）
  - ✓ T_{c1}执行，队列空，睡眠
  - ✓ T_p执行，唤醒T_{c1}后睡眠
  - ✓ T_{c2}抢先执行，消费后唤醒T_p后睡眠
  - ✓ T_{c1}发现队列为空

由于count为1，c3被跳过

追踪线程：有问题的方案（问题1）

- 问题的产生原因：
  - 在生产者唤醒 T_{c1} 线程后，但在 T_{c1} 线程运行之前，边界缓冲区的状态被 **T_{c2} 线程改变**
  - 没有保证唤醒线程运行时，状态仍然是所期望的 → Mesa 语义
    - 几乎所有系统都采用了 Mesa 语义
  - **Hoare 语义**提供了更强的保证，确保唤醒的线程会在被唤醒后立即运行

生产者/消费者：单一条件变量和 while 循环

- 消费者 T_{c1} 唤醒并重新检查共享变量的状态

```c
cond_t cond;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);   // p1
        while (count == 1) {          // p2
            Pthread_cond_wait(&cond, &mutex);  // p3
        }
        put(i);                       // p4
        Pthread_cond_signal(&cond);   // p5
        Pthread_mutex_unlock(&mutex); // p6
    }
}
```

```c
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);   // c1
        while (count == 0) {          // c2
            Pthread_cond_wait(&cond, &mutex);  // c3
        }
        int tmp = get();              // c4
        Pthread_cond_signal(&cond);   // c5
        Pthread_mutex_unlock(&mutex); // c6
    }
}
```

- 简单的规则：使用条件变量时，始终使用 while 循环
- 然而，这段代码仍然存在一个问题【与只使用一个条件变量有关】

追踪线程：有问题的方案（问题2）

T_{c1}	状态	T_{c2}	状态	T_p	状态	count	注释
c1	运行		就绪		就绪	0	
c2	运行		就绪		就绪	0	
c3	睡眠		就绪		就绪	0	没数据可取
	睡眠	c1	运行		就绪	0	
	睡眠	c2	运行		就绪	0	
	睡眠	c3	睡眠		就绪	0	没数据可取
	睡眠		睡眠	p1	运行	0	
	睡眠		睡眠	p2	运行	0	
	睡眠		睡眠	p4	运行	1	缓冲区现在满了
	就绪		睡眠	p5	运行	1	T_{c1} 唤醒
	就绪		睡眠	p6	运行	1	
	就绪		睡眠	p1	运行	1	
	就绪		睡眠	p2	运行	1	
	就绪		睡眠	p3	睡眠	1	必须睡（满了）
c2	运行		睡眠		睡眠	1	重新检查条件
c4	运行		睡眠		睡眠	0	T_{c1} 抓取数据
c5	运行		就绪		睡眠	0	啊！唤醒 T_{c2}
c6	运行		就绪		睡眠	0	
c1	运行		就绪		睡眠	0	
c2	运行		就绪		睡眠	0	
c3	睡眠		就绪		睡眠	0	没数据可取
	睡眠	c2	运行		睡眠	0	
	睡眠	c3	睡眠		睡眠	0	大家都睡了……

- 两个消费者（T_{c1}和T_{c2}）
- 一个生产者（T_p）
  - ✓ 两个消费者先运行至睡眠
  - ✓ 生产者运行，唤醒T_{c1}后睡眠
    （T_{c2}和T_p均在同一个条件变量上等待）
  - ✓ T_{c1}执行，消费后唤醒了T_{c2}后睡眠
  - ✓ T_{c2}发现队列为空，睡眠

消费者不应唤醒其他消费者，只能唤醒生产者，反之亦然。

单缓冲区生产者/消费者：可用解决方案

- 使用两个条件变量 和 **while** 循环
  - **生产者线程** 等待条件变量 empty，并发出信号 fill
  - **消费者线程** 等待条件变量 fill，并发出信号 empty

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1) {
            Pthread_cond_wait(&empty, &mutex);
        }
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0) {
            Pthread_cond_wait(&fill, &mutex);
        }
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

最终的生产者/消费者解决方案：更好的并发和效率

- 更好的 并发性 和 效率 → 增加更多的缓冲区槽位
  - •允许生产或消费同时进行
  - •减少上下文切换

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

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

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

// 修改1: 缓冲区结构本身

最终的生产者/消费者解决方案（续）

- **p2**: 生产者仅在所有缓冲区已满时才会休眠。
- **c2**: 消费者仅在所有缓冲区为空时才会休眠。

```
cond_t empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);    // p1
        while (count == MAX) {         // p2
            Pthread_cond_wait(&empty, &mutex); // p3
        }
        put(i);                        // p4
        Pthread_cond_signal(&fill);    // p5
        Pthread_mutex_unlock(&mutex); // p6
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);    // c1
        while (count == 0) {           // c2
            Pthread_cond_wait(&fill, &mutex);  // c3
        }
        int tmp = get();               // c4
        Pthread_cond_signal(&empty);   // c5
        Pthread_mutex_unlock(&mutex); // c6
        printf("%d\n", tmp);           // c7
    }
}
```

覆盖条件

- 假设当前没有空闲字节。
  - 线程 T_a 调用 allocate(100)。
  - 线程 T_b 调用 allocate(10)。
  - 线程 T_a 和 T_b 都在条件变量上等待，并进入休眠。
  - 线程 T_c 调用 free(50)。

哪个等待的线程应该被唤醒？

```
// 堆中有多少字节是空闲的?
int bytesLeft = MAX_HEAP_SIZE;

// 需要锁和条件变量
cond_t c;
mutex_t m;

void *allocate(int size) {
    Pthread_mutex_lock(&m);      // 获取锁
    while (bytesLeft < size) {  // 如果剩余的字节数小于请求的内存
        Pthread_cond_wait(&c, &m);  // 等待条件变量
    }
    void *ptr = ...;             // 从堆中获取内存
    bytesLeft -= size;           // 更新剩余字节数
    Pthread_mutex_unlock(&m);    // 释放锁
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);      // 获取锁
    bytesLeft += size;           // 更新剩余字节数
    Pthread_cond_signal(&c);     // 唤醒一个等待线程
    Pthread_mutex_unlock(&m);    // 释放锁
}
```

覆盖条件解决方案：使用pthread_cond_broadcast()

•解决方案（由 **Lampson** 和 **Redell** 提出）

- 将 pthread_cond_signal() 替换为 pthread_cond_broadcast()

•**pthread_cond_broadcast()**：

- 唤醒**所有等待的线程**。
- **缺点**：可能会唤醒太多线程。
- 被唤醒的线程将重新检查条件，若条件不满足，则会再次进入睡眠。

小结

- 介绍了什么是条件变量
- 介绍了线程同步的一些方法
  - 基于自旋的实现方式（低效）
  - 基于条件变量和互斥锁的实现方式
- 介绍了基于条件变量的生产者/消费者问题的实现
  - 单一条件变量与 if 语句实现（有问题）
  - 单一条件变量和 while 循环（仍存在问题）
  - 两个条件变量和 while 循环（可用）
  - 更多缓冲区方案（更好的并发和效率）
- 介绍了覆盖条件问题和解决方案