第二章　进程管理

翁楚良

https://chuliangweng.github.io

2023 春　ECNU

第二章进程管理 提纲

- 2.1 进程
- 2.2 进程间通信
- 2.3 经典并发问题
- 2.4 进程调度
- 2.5 MINIX3进程概述
- 2.6 MINIX3进程实现
- 2.7 MINIX3系统任务
- 2.8 MINIX3时钟任务

进程间通信

- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

可能的方法(1)

- 关中断
  - 使每个进程在进入临界区后先关中断，在离开之前再开中断
  - 关中断CPU只有在发生时钟或其他中断时才会进行进程切换，这样关中断之后CPU将不会被切换到其他进程
  - 关中断对于操作系统本身是一项很有用的技术，但对于用户进程则不是一种合适的通用互斥机制
- 锁变量
  - 设想有一个共享（锁）变量，初值为0。当一个进程想进入其临界区时，它首先测试这把锁。如果锁的值为0，则进程将其置为1并进入临界区。若锁已经为1，则进程将等待直到其变成0。于是，0就表示临界区内没有进程，1表示已经有某个进程进入了临界区。
  - 仍会发生竞争条件

可能的方法(2)

- 严格交替法
  - 确保两个进程严格轮流进入临界区
  - 容易导致临界区外的进程阻塞其他进程

```
while (TRUE){
    while(turn != 0)        /* loop* /;
    critical_region();
    turn  = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while(turn != 1)        /* loop* /;
    critical_region();
    turn  = 0;
    noncritical_region();
}
```

(b)

忙等待形式互斥

- 当一个进程想进入临界区，首先检查是否允许进入，若不允许，则该进程将忙等待，直至许可为止。
  - Peterson互斥解法
  - TSL解法

Peterson解法

```
#define FALSE 0
#define TRUE  1
#define N     2                         /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE)*/
void enter_region(int process)          /* process is 0 or 1 */
{
     int other;                         /* number of the other process */
     other = 1 - process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE);/* null statement */
}
void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

TSL解法

- 测试并上锁（TSL）
  - 它将一个存储器字读到一个寄存器中，然后在该内存地址上存一个非零值。
  - 读数和写数操作保证是不可分割的 - 即该指令结束之前其他处理机均不允许访问该存储器字。
  - 执行TSL指令的CPU将锁住内存总线以禁止其他CPU在本指令结束之前访问内存。

```
enter_region:
        TSL REGISTER,LOCK       |copy LOCK to register and set LOCK to 1
        CMP REGISTER,#0         |was LOCK zero?
        JNE ENTER_REGION        |if it was non zero, LOCK was set, so loop
        RET                     |return to caller; critical region entered


leave_region:
       MOVE LOCK,#0             |store a 0 in LOCK
       RET                      |return to caller
```

进程间通信

- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

优先级翻转问题

- 考虑一台计算机有两个进程，H优先级较高，L优先级较低。
- 调度规则规定只要H处于就绪态它就可以运行。
- 在某一时刻，L处于临界区中，此时H变到就绪态准备运行（例如，一条I/O操作结束）。现在H开始忙等待，但由于当H就绪时L不会被调度，也就无法离开临界区，所以H将永远忙等待下去。
- 这种情况有时被称作优先级翻转问题(priority inversion problem)。

睡眠和唤醒

- 无法进入临界区时将阻塞，而不是忙等待。
- 最简单的是睡眠（SLEEP）和唤醒（WAKEUP）。
  - SLEEP系统调用将引起调用进程阻塞，即被挂起，直到另一进程将其唤醒。
  - WAKEUP调用有一个参数，即要被唤醒的进程。

含竞争条件的生产者消费者问题

[View full width]

```
#define N 100                                        /* number of slots in the buffer */
int count = 0;                                       /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE){                                   /* repeat forever */
          item = produce_item();                     /* generate next item */
          if (count == N) sleep();                   /* if buffer is full, go to sleep */
          insert_item(item);                         /* put item in buffer */
          count = count + 1;                         /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);          /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE){                                   /* repeat forever */
          if (count == 0) sleep();                   /* if buffer is empty, got to sleep */
          item = remove_item();                      /* take item out of buffer */
          count = count - 1;                        /* decrement count of items in
   buffer */
          if (count ==N - 1) wakeup(producer);      /* was buffer full? */
          consume_item(item);                        /* print item */
     }
}
```

- 由于对count的访问未加限制，会导致生产者添满整个缓冲区，最终二进程将永远睡眠

- 唤醒等待位。当向一个清醒进程发送一个唤醒信号时，将该位置位。当进程睡眠时，若唤醒等待位为1，则进程仍保持清醒

进程间通信

- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

原语

- 原语(primitive)：由若干条指令构成的“原子操作(atomic operation)”过程，作为一个整体而不可分割，即要么全都完成，要么全都不做。许多系统调用就是原语。

- 系统调用并不都是原语。
  - 进程A调用read()，因无数据而阻塞，在read()里未返回。然后进程B调用read()，此时read()被重入。系统调用不一定一次执行完并返回该进程，有可能在特定的点暂停，而转入到其他进程。

信号量和P、V原语

- 1965年，由荷兰学者Dijkstra提出（所以P、V分别是荷兰语的test(proberen)和increment(verhogen)），是一种卓有成效的进程同步机制
- 一个信号量的值可以为0，表示没有积累下来的唤醒操作；或者为正值，表示有一个或多个被积累下来的唤醒操作。
- "二进制信号量(binary semaphore)"：只允许信号量取0或1值

down 和 up

- DOWN操作检查信号量值是否大于0
  - 若是则将其值减1（即，用掉一个保存的唤醒信号）并继续
  - 若值为0，则进程将睡眠，而且此时DOWN操作并未结束
    - 检查数值、改变数值、以及可能发生的睡眠操作均作为一个单一的、不可分割的原子操作(atomic action)完成
- UP操作递增信号量的值
  - 如果一个或多个进程在该信号量上睡眠，无法完成一个先前的DOWN操作，则由系统选择其中的一个（例如，随机挑选）并允许其完成它的DOWN操作
  - 对一个有进程在其上睡眠的信号量执行一次UP操作之后，该信号量的值仍旧是0，但在其上睡眠的进程却少了一个。递增信号量的值和唤醒一个进程同样也是不可分割的
  - 不会有进程因执行UP而阻塞

生产者-消费者问题(信号量)(1)

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
void producer(void)
{
      int item;

      while (TRUE){
            item = produce_item();
            down(&empty);
            down(&mutex);
            insert_item(item);
            up(&mutex);
            up(&full);
      }
}
```

生产者-消费者问题(信号量)(2)

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;
```

```
void consumer(void)
{
    int item;

    while (TRUE){
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```

进程间通信

- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

利用信号量实现互斥

- 为临界资源设置一个互斥信号量mutex(MUTual Exclusion)；在每个进程中将临界区代码置于P(mutex)和V(mutex)原语之间
- 必须成对使用P和V原语：遗漏P原语则不能保证互斥访问，遗漏V原语则不能在使用临界资源之后将其释放（给其他等待的进程）；P、V原语不能次序错误、重复或遗漏

加锁

```
P(mutex);
```

```
critical section
```

解锁

```
V(mutex);
```

```
remainder section
```

进程间通信

- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

管程

- 信号量同步操作分散，易读性差、不利于修改和维护、正确性难以保证
- Brinch Hansen (1973) and Hoare (1974)分别提出管程(monitor)
  - 其基本思想是把信号量及其操作原语封装在一个对象内部。即：将共享变量以及对共享变量能够进行的所有操作集中在一个模块中。
  - 是过程、变量和相关数据结构的集合，构成一个特殊的模块或软件包
  - 引入管程可提高代码的可读性，便于修改和维护，正确性易于保证。

```
monitor example
 integer i;
 condition c;

 procedure producer (x);
 .
 .
 .
 end;

 procedure consumer (x);
 .
 .
 .
 end;
end monitor;
```

管程的主要特性

- 模块化：一个管程是一个基本程序单位，可以单独编译
- 抽象数据类型：管程是一种特殊的数据类型，其中不仅有数据，而且有对数据进行操作的代码
- 信息封装：管程是半透明的，管程中的过程（函数）实现了某些功能，至于这些功能是怎样实现的，在其外部则是不可见的

管程的实现要素

- 管程中的共享变量在管程外部是不可见的，外部只能通过调用管程中所说明的过程（函数）来间接地访问管程中的共享变量
- 为了保证管程共享变量的数据完整性，规定管程互斥进入
- 管程通常是用来管理资源的，因而在管程中应当设有进程等待队列以及相应的等待及唤醒操作

条件变量(condition variable)

- 由于管程通常是用于管理资源的，因而在管程内部，应当存在某种等待机制
- 当进入管程的进程因资源被占用等原因不能继续运行时使其等待
- 为此在管程内部可以说明和使用一种特殊类型的变量 — 条件变量
  - 每个表示一种等待原因 — 相当于每个原因对应一个队列

管程例子

```
monitor ProducerConsumer
     condition full, empty;
     integer count;

     procedure insert(item: integer);
     begin
          if count = N then wait(full);
          insert_item(item);
          count := count + 1;
          if count = 1 then signal(empty)
     end;

     function remove: integer;
     begin
          if count = 0 then wait(empty);
          remove = remove_item;
          count := count  1;
          if count = N  1 then signal(full)
     end;

     count := 0;
end monitor;
```

基于管程的生产者-消费者

```
procedure producer;
begin
      while true do
      begin
            item = produce_item;
            ProducerConsumer.insert(item)
      end
end;
```

```
procedure consumer;
begin
      while true do
      begin
            item = ProducerConsumer.remove;
            consume_item(item)
            end
end;
```

进程间通信

- 竞争条件
- 临界区
- 忙等待形式互斥
- 睡眠和唤醒
- 信号量
- 互斥
- 管程
- 消息传递

消息传递

- 消息传递(message passing)进行进程间通信，使用两条原语SEND和RECEIVE
  - 可以以库函数的方式实现
    - send(destination, &message, &status);
    - receive(source, &message, &status);

消息传递系统要点

- 为了防止消息丢失，发送方和接收方可以达成如下一致：一旦信息被接收到，接收方马上回送一条特殊的应答(acknowledgement)消息。如果发送方在一段时间间隔内未收到应答，则进行重发
- 区分新消息和一条重发的老消息是非常重 要的。通常采用在每条原始消息中嵌入一个连续的序号来解决该问题
- 消息系统还需要解决进程命名的问题，这样才能明确在SEND和RECEIVE调用中所指定的进程

基于消息传递的生产者-消费者

```
#define N 100

void producer(void)
{
     int item;
     message m;

     while (TRUE) {
          item = produce_item();
          receive(consumer, &m);
          build_message(&m, item);
          send(consumer, &m);
     }
}
```

```
void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);
     while (TRUE) {
          receive(producer, &m);
          item = extract_item(&m);
          send(producer, &m);
          consume_item(item);
     }
}
```

MPI例子

- 其中一个进程(进程0) 向另一个进程(进程1)发送一条消息
- 该消息是一个字符串"Hello, process 1 “
- 进程1在接收到该消息后将这一消息打印到屏幕上

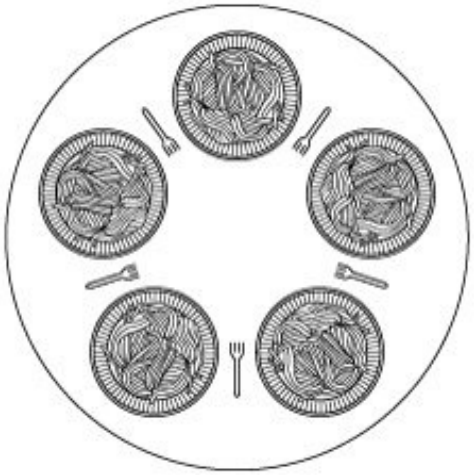
```
#include "mpi.h"
main( argc, argv )
int argc;
char **argv;
{
   char message[20];
   int myrank;
   MPI_Init( &argc, &argv );   /* MPI程序的初始化*/
   MPI_Comm_rank( MPI_COMM_WORLD, &myrank );/* 得到当前进程的标识*/
   if (myrank == 0) /* 若是 0 进程*/
   {
      strcpy(message,"Hello, process 1");
      MPI_Send(message, strlen(message), MPI_CHAR, 1, 99, MPI_COMM_WORLD);
   }
   else if(myrank==1) /* 若是进程 1 */
   {
      MPI_Recv(message, 20, MPI_CHAR, 0, 99, MPI_COMM_WORLD, &status);
      printf("received :%s:", message);
   }
   MPI_Finalize();  /* MPI程序结束*/
}
```

第二章进程管理 提纲

- 2.1 进程
- 2.2 进程间通信
- 2.3 经典并发问题
- 2.4 进程调度
- 2.5 MINIX3进程概述
- 2.6 MINIX3进程实现
- 2.7 MINIX3系统任务
- 2.8 MINIX3时钟任务

哲学家进餐问题

- 五个哲学家围坐在一张圆桌周围，每个哲学家面前都有一碟通心面，由于面条很滑，所以要两把叉子才能夹住，相邻两个碟子之间有一把叉子。
- 哲学家的生活包括两种活动：即吃饭和思考。
- 当一个哲学家觉得饿时，他就试图分两次去取他左边和右边的叉子，每次拿一把，但不分次序。
  - 如果成功地获得了两把叉子，他就开始吃饭，吃完以后放下叉子继续思考。



死锁?
饥饿?
效率？

解决方案(1)

```
#define N                5
#define LEFT             (i+N-1)%N
#define RIGHT            (i+1)%N
#define THINKING         0
#define HUNGRY           1
#define EATING           2
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
     while (TRUE){
          think();
          take_forks(i);
          eat();
          put_forks(i);
     }
}
```

解决方案(2)

```
void take_forks(int i)
{
        down(&mutex);
        state[i] = HUNGRY;
        test(i);
        up(&mutex);
        down(&s[i]);
}
```

```
void put_forks(i)
{
        down(&mutex);
        state[i] = THINKING;
        test(LEFT);
        test(RIGHT);
        up(&mutex);
}
```

```
void test(i)                        /* i: philosopher number, from 0 to N1* /
{
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
}
```

读者-写者问题

- 对共享资源的读写操作，任一时刻“写者”最多只允许一个，而“读者”则允许多个
  - "读 - 写"互斥
  - "写 - 写"互斥
  - "读 - 读"允许

解决方案

```
typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;
```

```
void reader(void)
{
     while (TRUE){
          down(&mutex);
          rc = rc + 1;
          if (rc == 1) down(&db);
          up(&mutex);
          read_data_base();
          down(&mutex);
          rc = rc - 1;
          if (rc == 0) up(&db);
          up(&mutex);
          use_data_read();
     }
}
```

```
void writer(void)
{
     while (TRUE){
          think_up_data();
          down(&db);
          write_data_base();
          up(&db);
     }
}
```