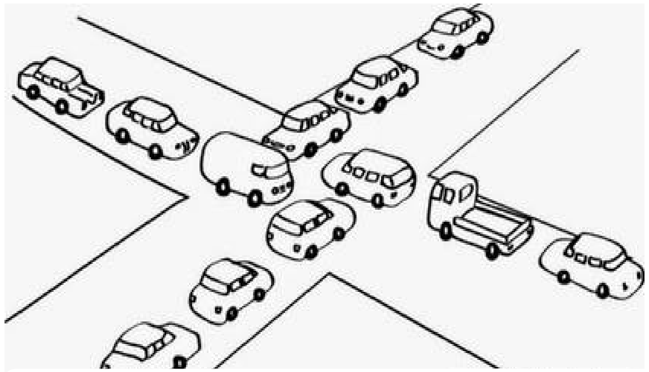
Operating Systems

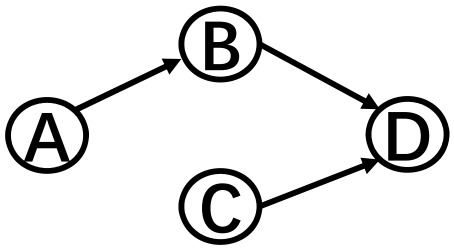
Jinghui Zhong （钟竞辉）
Office：B3-515
Email ： jinghuizhong@scut.edu.cn

Inter Process Communication (IPC)

- How to pass information among processes?
- How to make sure two or more processes do not get into each other's way when engaging in critical activities.



- Proper sequencing when dependencies are present.



Race Conditions

- **Race conditions:** situations in which several processes access shared data and the final result depends on the order of operations.

- With increasing parallelism due to increasing number of cores, race condition are becoming more common.

Example of Race Condition

Out: points to the next file to be printed

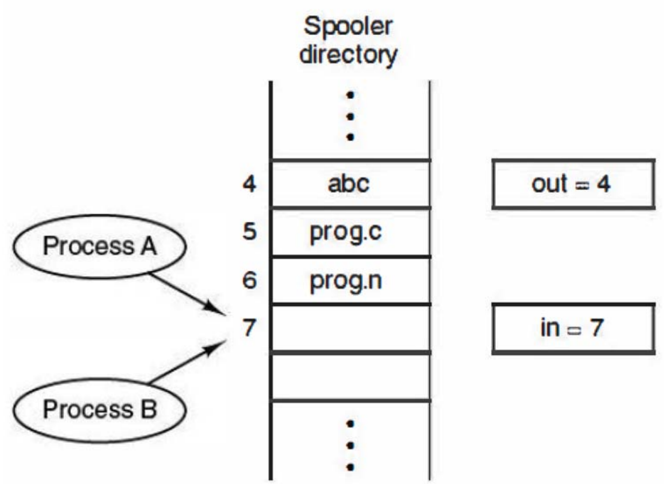
In: points to the next free slot in the directory.

in =7

(1): Process A reads **in** and stores the value 7 in a local variable. A switch to process B happens.

(2): Process B reads **in**, stores the file name in slot 7 and updates **in** to be an 8.

(3): Process A stores the file name in slot 7 and updates **in** to be an 8.



The file name in slot 7 was determined by who finished last. A race condition occurs.

Critical Regions

- Key idea to avoid race condition： **prohibit more than one process from reading and writing the shared data at the same time.**

- **Critical Region:** part of the program where the share memory is accessed。

```
// 临界区结构对象
CRITICAL_SECTION g_cs;
// 共享资源
char g_cArray[10];
UINT ThreadProc10(LPVOID pParam)
{
    // 进入临界区
    EnterCriticalSection(&g_cs);
    // 对共享资源进行写入操作
    for (int i = 0; i < 10; i++)
    {
    g_cArray[i]  = a;
    Sleep(1);
    }
    // 离开临界区
    LeaveCriticalSection(&g_cs);
    return 0;
}
UINT ThreadProc11(LPVOID pParam)
{
    // 进入临界区
    EnterCriticalSection(&g_cs);
    // 对共享资源进行写入操作
    for (int i = 0; i < 10; i++)
    {
        g_cArray[10 - i - 1] = b;
        Sleep(1);
    }
    // 离开临界区
    LeaveCriticalSection(&g_cs);
    return 0;
}
……
void CSample08View::OnCriticalSection()
{
    // 初始化临界区
    InitializeCriticalSection(&g_cs);
    // 启动线程
    AfxBeginThread(ThreadProc10, NULL);
    AfxBeginThread(ThreadProc11, NULL);
    // 等待计算完毕
    Sleep(300);
    // 报告计算结果
    CString sResult = CString(g_cArray);
    AfxMessageBox(sResult);
}
```

Critical Regions

- Four conditions to support a good solution

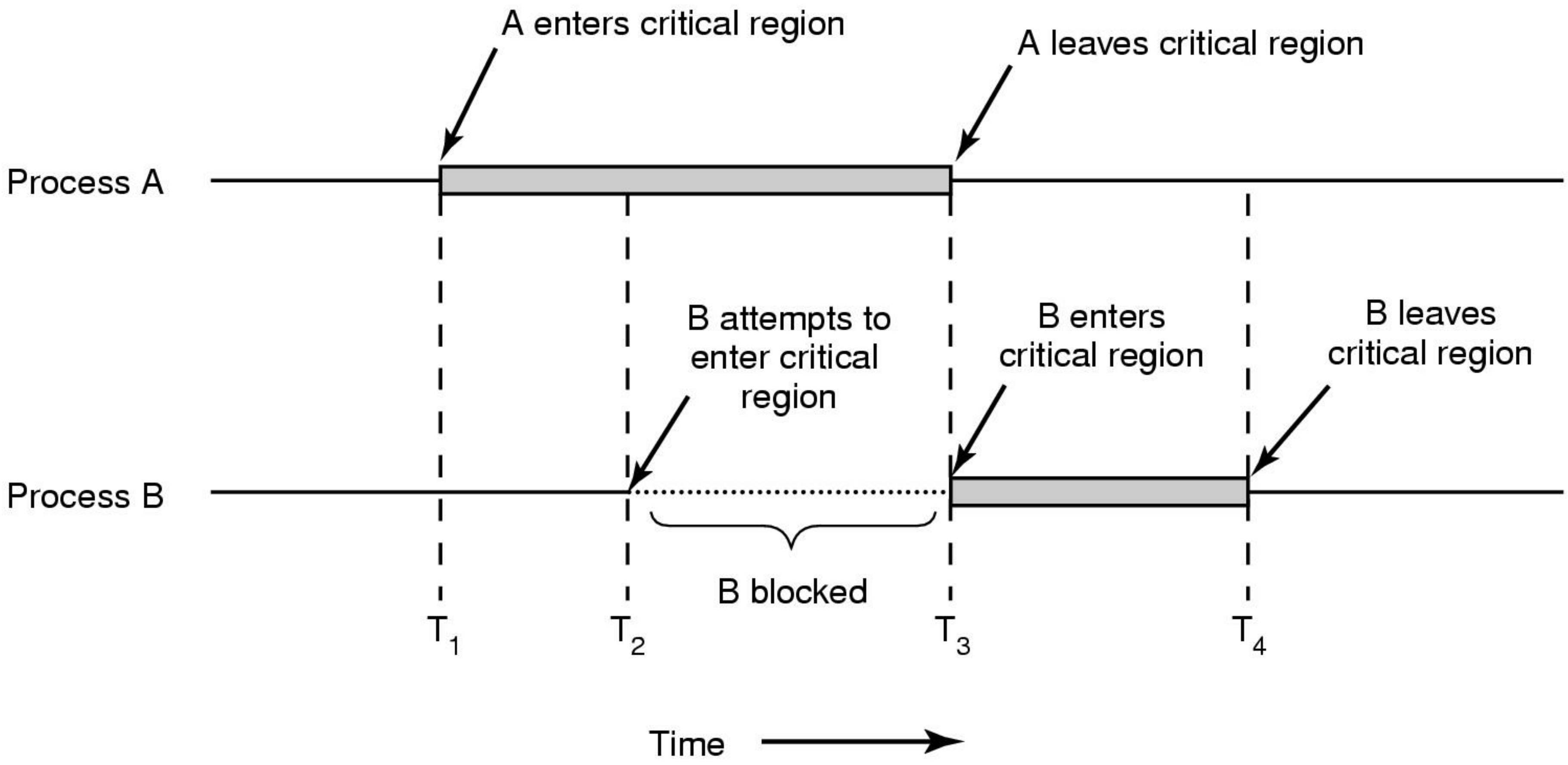
① No two processes may be simultaneously in critical region

② No assumption made about speeds or numbers of CPUs

③ No process running outside its critical region may block another process

④ No process must wait forever to enter its critical region

Mutual Exclusion using Critical Regions



Mutual Exclusion Solution - Disabling Interrupts

- The CPU is only switch from process to process when clock or other interrupts happen; Hence, by disabling all interrupts, the CPU will not be switched to another process.
- However, it is unwise to allow user processes to disable interrupts.
  - ✓One thread may never turn on interrupt;
  - ✓Problem still exist for multiprocessor systems.

Mutual Exclusion Solution - Lock Variable

```
shared int lock = 0;
/* entry_code: execute before entering critical section */
while (lock != 0) ; // do nothing
lock = 1;
- critical section -
/* exit_code: execute after leaving critical section */
lock = 0;
```

Problem?

- If a context switch occurs after one process executes the while statement, but before setting lock = 1, then two (or more) processes may be able to enter their critical sections at the same time.

Mutual Exclusion Solution – Strict Alternation

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

(a) Process 0.

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

(b) Process 1.

- Since the processes must strictly alternate entering their critical sections, a process wanting to enter its critical section twice will be blocked until the other process decides to enter (and leave) its critical section.

Mutual Exclusion – Peterson's Solution

```
#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 */
}
```

- This solution satisfies all 4 properties of a good solution. Unfortunately, this solution involves busy waiting in the while loop.

Hardware solution: Test-and-Set Locks (TSL)

- The hardware must support a special instruction, TSL, which does **two** things in a single atomic action:
  - (a) copy a value in memory (flag) to a CPU register
  - (b) set flag to 1.

```
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
```

Mutual Exclusion with Busy Waiting

- **BUSY-WAITING**： a process executing the entry code will sit in a tight loop using up CPU cycles, testing some condition over and over, until it becomes true.
- Busy-waiting may lead to the **priority-inversion problem** .

Not only does this approach waste CPU time, but it can also have unexpected effects. Consider a computer with two processes, H, with high priority, and L, with low priority. The scheduling rules are such that H is run whenever it is in ready state. At a certain moment, with L in its critical region, H becomes ready to run (e.g., an I/O operation completes). H now begins busy waiting, but since L is never scheduled while H is running, L never gets the chance to leave its critical region, so H loops forever. This situation is sometimes referred to as the **priority inversion problem**.

Sleep and Wakeup

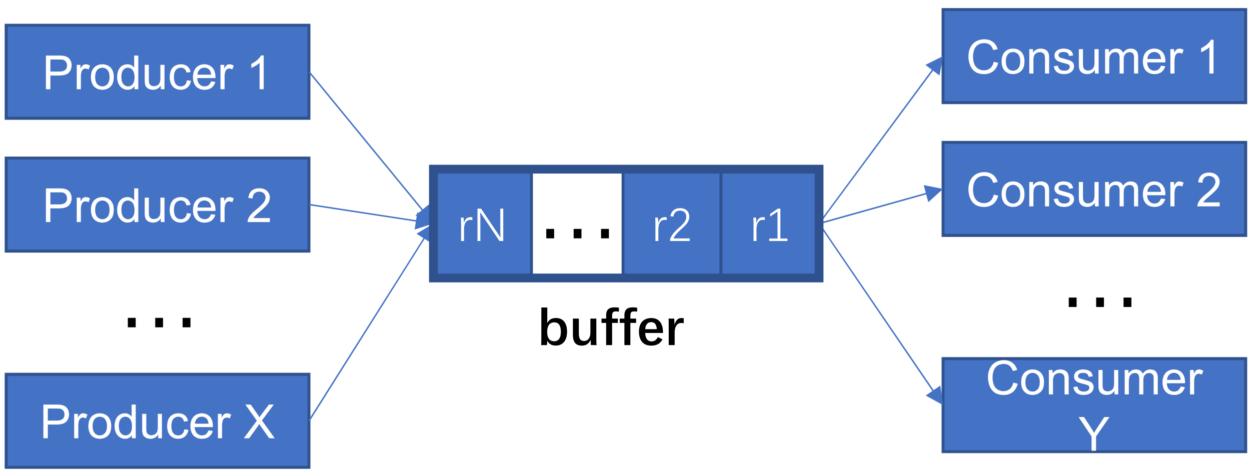
- **From busy waiting to blocking...**

Sleep: is a system call that causes the caller to block, that is, be suspended until another process wakes it up.

Wakeup: has one parameter, the process to be awakened.

Producer-Consumer Problem

- **Producer-Consumer Problem:** Consider a circular buffer that can hold *N* items. Producers add items to the buffer and Consumers remove items from the buffer. The Producer-Consumer Problem is to restrict access to the buffer.



Producer-Consumer Problem

```
#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 */
      }
}
```

Producer-consumer problem with fatal race condition

Semaphores [E.W. Dijkstra, 1965]

- **A SEMAPHORE, S, is a structure consisting of two parts:**
  - **(a) an integer counter, COUNT**
  - **(b) a queue of pids of blocked processes, Q**

- That is,

```
struct sem_struct {
  int count;
  queue Q;
} semaphore;

semaphore S;
```

Semaphores [E.W. Dijkstra, 1965]

- There are two operations on semaphores, **UP** and **DOWN** (PV). These operations must be **executed atomically** (that is in mutual exclusion). Suppose that P is the process making the system call. The operations are defined as follows:

```
DOWN(S):

        if (S.count > 0)
                S.count = S.count - 1;
        else
                block(P); that is,
            (a) enqueue the pid of P in S.Q,
            (b) block process P (remove the pid from the ready queue)
            (c) pass control to the scheduler.
```

Semaphores [E.W. Dijkstra, 1965]

```
UP(S):

    if (S.Q is nonempty)
        wakeup(P) for some process P in S.Q; that is,
            (a) remove a pid from S.Q (the pid of P),
            (b) put the pid in the ready queue, and
            (c) pass control to the scheduler.
    else
        S.count = S.count + 1;
```

Mutual Exclusion Solution

- Semaphores do not require busy-waiting, instead they involve BLOCKING.

```
semaphore mutex = 1;   // set mutex.count = 1

DOWN(mutex);
      - critical section -
UP(mutex);
```

Mutexes

- A mutex is a semaphore that can be in one of two states: unlocked (0) or locked (1).

```
mutex_lock:
      TSL REGISTER,MUTEX          | copy mutex to register and set mutex to 1
      CMP REGISTER,#0             | was mutex zero?
      JZE ok                      | if it was zero, mutex was unlocked, so return
      CALL thread_yield           | mutex is busy; schedule another thread
      JMP mutex_lock              | try again later
ok:   RET | return to caller; critical region entered


mutex_unlock:
      MOVE MUTEX,#0               | store a 0 in mutex
      RET | return to caller
```

Implementation of *mutex_lock* and *mutex_unlock*

Previous Solution

```
#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 */
    }
}
```

Producer-consumer problem with fatal race condition

Semaphore Based Solution

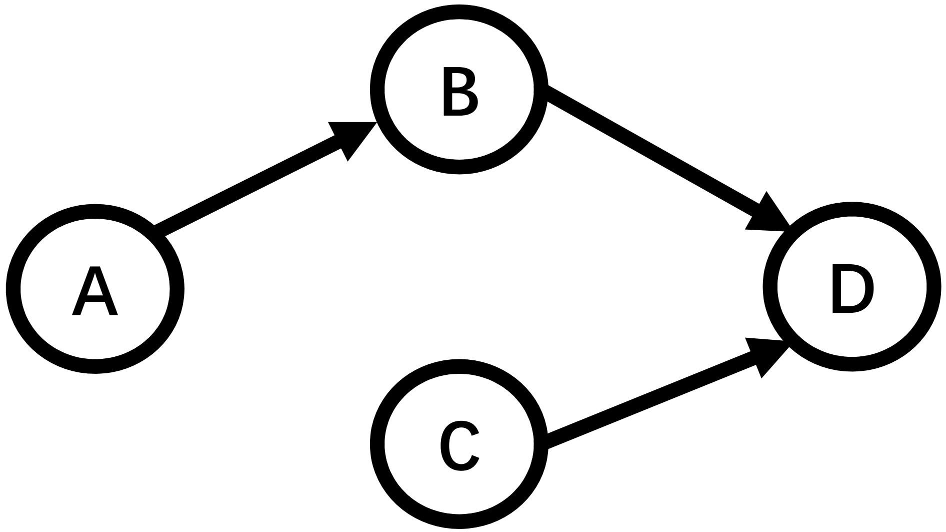
```
#define N 100                           /* number of slots in the buffer */
typedef int semaphore;                  /* semaphores are a special kind of int */
semaphore mutex = 1;                    /* controls access to critical region */
semaphore empty = N;                    /* counts empty buffer slots */
semaphore full = 0;                     /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                     /* TRUE is the constant 1 */
          item = produce_item( );       /* generate something to put in buffer */
          down(&empty);                 /* decrement empty count */
          down(&mutex);                 /* enter critical region */
          insert_item(item);            /* put new item in buffer */
          up(&mutex);                   /* leave critical region */
          up(&full);                    /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                     /* infinite loop */
          down(&full);                  /* decrement full count */
          down(&mutex);                 /* enter critical region */
          item = remove_item( );        /* take item from buffer */
          up(&mutex);                   /* leave critical region */
          up(&empty);                   /* increment count of empty slots */
          consume_item(item);           /* do something with the item */
     }
}
```

The producer-consumer problem using semaphores

Using Semaphores

- **Process Synchronization:**

Suppose we have 4 processes: A, B, C, and D. A must finish executing before B start. B and C must finish executing before D starts.



How many semaphores should be used to achieve the above goal?

Using Semaphores

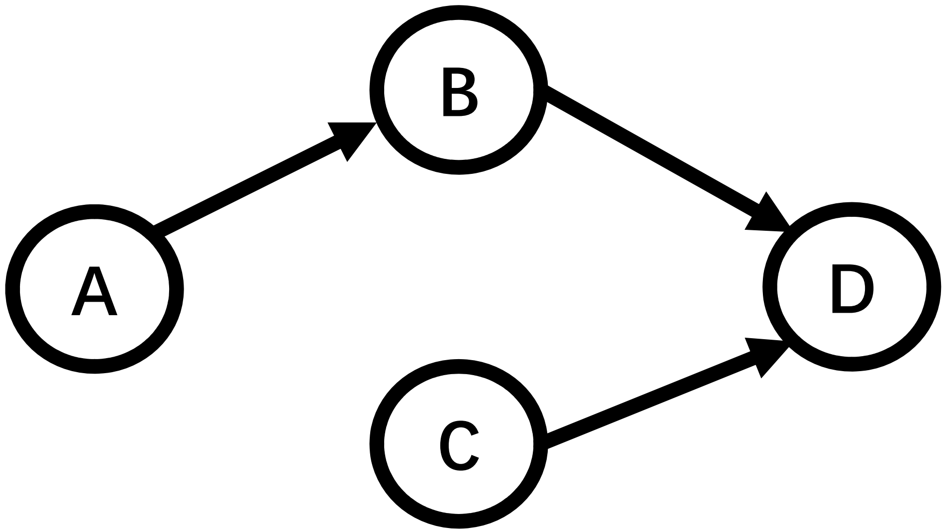
- Process Synchronization:

Process A:
- do work of A
UP(S1); /* Let B or C start */

Process B:
DOWN(S1); /* Block until A is finished */
- do work of B
UP(S2);

Process C:
- do work of C
UP(S3);

Process D:
DOWN(S2);
DOWN(S3);
- do work of D



Check Points

- What is Race Condition?
- What is Critical Region?
- What is Busy Waiting?
- What is Semaphore?
- What is Mutex?