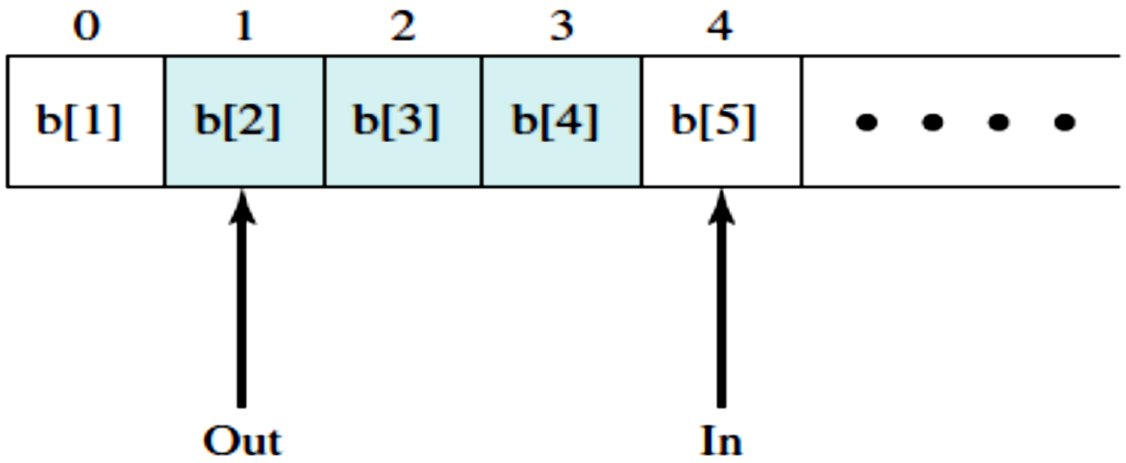
生产者——消费者问题

一到多个生产者和一到多个消费者共享一个可同时容纳多个产品的缓冲区，生产者不断生产产品并放入缓冲区，消费者不断从缓冲区取出产品并消耗它。 要求：

- 任何时刻最多只能有一个生产者或消费者访问缓冲区；
- 禁止生产者向满缓冲区输送产品；
- 禁止消费者从空缓冲区提取产品。

缓冲区无界的情况




```
producer:
while (true) {
    /* produce item v */;
    b[in] = v;
    in++;
}
```

```
consumer:
while (true) {
    while (in <= out)
        /* do nothing */;
    w = b[out];
    out++;
    /* consume item w */;
}
```

Q：缓存区无界时需要考虑哪些并发问题?

1. 互斥;
2. 有多少个可用的满缓冲区?
3. 没有满缓冲区可用时，消费者应该被阻塞。

方案一


```
int n=0;  //可用的满缓冲区的数量
semaphore s = 1;  //互斥信号量
semaphore delay = 0;  //同步信号量
```

```
void producer()
{
    while (true) {
        生产产品;
        P(s);
        放入一个产品;
        n++;
        if (n==1) V(delay);
        V(s);
    }
}
```

```
void consumer()
{
    P(delay);
    while (true) {
        P(s);
        取出一个产品;
        n--;
        V(s);
        消费产品;
        if (n==0) P(delay);
    }
}
```

Q：有什么问题?

	Producer	Consumer	s	n	delay
1			1	0	0
2	P(s)		0	0	0
3	n++		0	1	0
4	if (n==1) V(delay);		0	1	1
5	V(s)		1	1	1
6		P(delay)	1	1	0
7		P(s)	0	1	0
8		n--	0	0	0
9		V(s)	1	0	0
10	P(s)		0	0	0
11	n++		0	1	0
12	if (n==1) V(delay);		0	1	1
13	V(s)		1	1	1
14		if (n==0) P(delay);	1	1	1
15		P(s)	0	1	1
16		n--	0	0	1
17		V(s)	1	0	1
18		if (n==0) P(delay);	1	0	0
18		P(s)	0	0	0
20		n--	1	-1	0

方案二


```
int n=0;
semaphore s = 1, delay = 0;
void producer()
{
  while (true) {
    生产产品;
    P(s);
    放入一个产品;
    n++;
    if (n==1) V(delay);
    V(s);
  }
}
```

```
void consumer()
{
  P(delay);
  while (true) {
    P(s);
    取出一个产品;
    n--;
    if (n==0) P(delay);
    V(s);
    消费产品;
  }
}
```

Q：有什么问题?

方案三


```
int n=0;
semaphore s = 1, delay = 0;
void producer()
{
  while (true) {
    生产产品;
    P(s);
    放入一个产品;
    n++;
    if (n==1) V(delay);
    V(s);
  }
}
```

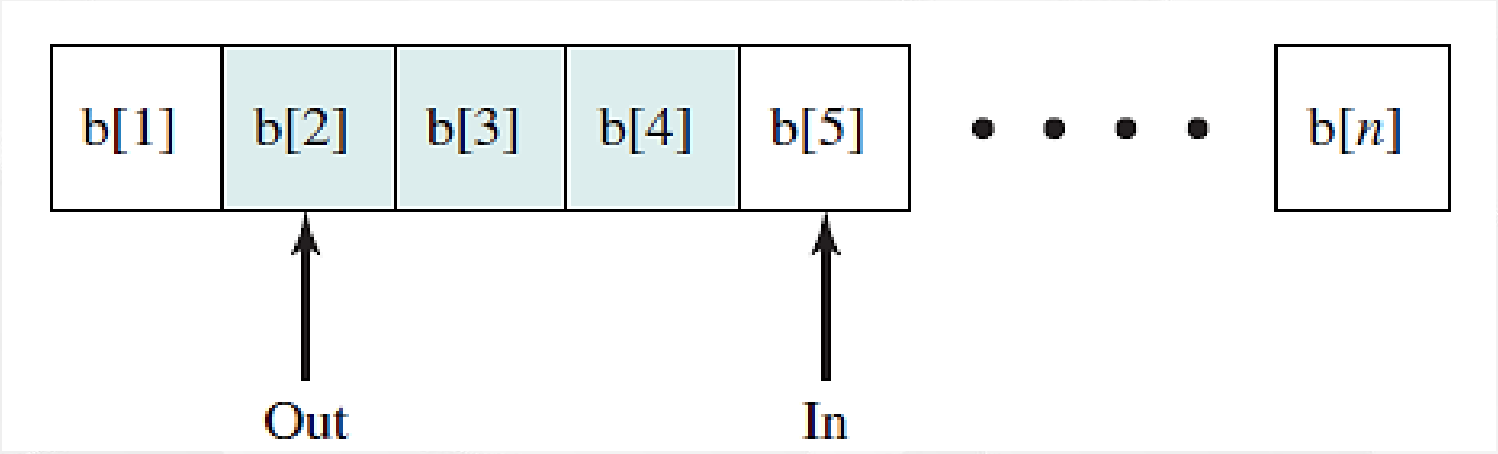
```
void consumer()
{
  int m;
  P(delay);
  while (true) {
    P(s);
    取出一个产品;
    n--;
    m=n;
    V(s);
    消费产品;
    if (m==0) P(delay);
  }
}
```

方案四


```
semaphore s = 1; //互斥
semaphore n = 0; //满缓冲区数量
void producer()
{
  while (true) {
    生产产品;
    P(s);
    放入一个产品;
    V(n);
    V(s);
  }
}
```

```
void consumer()
{
  while (true) {
    P(n);
    P(s);
    取出一个产品;
    V(s);
    消费产品;
  }
}
```

缓冲区有界的情况




```
producer:
while (true) {
    /* produce item v */
    while ((in + 1) % n == out)
        /* do nothing */;
    b[in] = v;
    in = (in + 1) % n;
}
```

```
consumer:
while (true) {
    while (in == out)
        /* do nothing */;
    w = b[out];
    out = (out + 1) % n;
    /* consume item w */;
}
```

缓存区有界时需要考虑的问题：

1. 互斥
2. 有多少个可用的空缓冲区?
3. 没有可用的空缓冲区时，生产者应该被阻塞
4. 有多少个可用的满缓冲区?
5. 没有可用的满缓冲区时，消费者应该被阻塞

如何设置信号量?

1. 一个互斥信号量

   mutex：表示有界缓冲区是否被占用，初值 = 1
2. 两个同步信号量

   s_a ： 表示满缓冲区 的数目，初值 = 0

   s_b ： 表示空缓冲区的数目，初值 = n

生产者:

生产;

$p(s_b)$;

p(mutex);

将数据放入有界缓冲区;

v(mutex);

$v(s_a)$;

消费者:

$p(s_a)$;

p(mutex);

从有界缓冲区中取数据;

v(mutex);

$v(s_b)$;

消费;

讨论：下面这个流程有没有问题?

生产者:

```
生产;
p(mutex);
p(sb);
将数据放入有界缓冲区;
v(sa);
v(mutex);
```

消费者:

```
p(mutex);
p(sa);
从有界缓冲区中取数据;
v(sb);
v(mutex);
消费;
```

程序描述

```
main( )
{
    semaphore s_a=0;        /*满缓冲区的数目   */
    semaphore s_b=n;        /*空缓冲区的数目   */
    semaphore mutex=1;  /*对有界缓冲区进行操作的互斥信号灯*/
    cobegin
        p1 ( );  p2 ( );  … pm ( );
        c1 ( );  c2 ( );  … ck ( );
    coend
}
```

```
pi( )
{
    while(生产未完成)
    {
        ⋮
        生产一个产品;
        p(sb);
        p(mutex);
        送一个产品到有界缓冲区 ;
        v(mutex);
        v(sa);
    }
}
```

```
cj( )
{
    while(还要继续消费)
    {
        p(sa);
        p(mutex);
        从有界缓冲区中取产品;
        v(mutex);
        v(sb);
        消费一个产品;
        ⋮
    }
}
```

讨论：这些并发进程的瓶颈在哪里？

课堂练习：

某公园有一个长凳，其上最多可以坐5个人。游客按以下规则使用长凳：

（1）如果长凳上还有空间可以坐，就坐到长凳上休息，直到休息结束，离开长凳。

（2）如果长凳上没有空间，就转身离开。

```
Main()
{
    semaphore s=5;     //长凳上的空位数
    cobegin
        p1(); p2(); ……pn();
    coend
}
pi()
{
    p(s);
    在长凳上休息;
    v(s);
    离开长凳;
}
```

对吗?

```
Main()
{
    int n=5;
    semaphore s=1;   //互斥信号量
    cobegin
        p1(); p2(); ……pn();
    coend
}
```

```
pi()
{
    p(s);
    if (n==0) {
        v(s);
        离开;
    }
    else {
        n--;
        v(s);
        在长凳上休息;
        p(s);
        n++;
        v(s);
        离开长凳;
}
```

课堂练习：

某商场有一个地下车库，有N个停车位，车辆只能通过一个指定的通道进出该车库，通道处只能容一辆车通过。请设计信号灯和P、V操作，给出**进库车辆**和**出库车辆**这两种进程的程序描述。

分析同步关系：

- 通道是临界资源，所有车辆必须互斥使用；
- 车库里有空的停车位的时候，进库车辆才能进库；
- 出库车辆出库后应该通知正在等待停车位的车辆。

```
main()
{
    semaphore n=N; //空位数量
    semaphore s=1;  //互斥信号量
    cobegin
        p1(); p2(); ……pn();
        q1(); q2(); ……qm();
    coend
}
```

```
pi()   //入库车辆
{
    ……
    p(n);
    p(s);
    进入车库;
    v(s);
    ……
}

qi()   //出库车辆
{
    ……
    p(s);
    开出车库;
    v(s);
    v(n);
    ……
}
```

进程通信

以下代码可能的输出结果是什么?

```
main()
{
    int i=0;
    int child;

    child=fork();
    if(child==0)
        i=i*2;
    else
        i=i+5;
    i=i*2;
    printf("i=%d\n", i);
}
```

1. i=0
   i=10

2. i=10
   i=0

进程通信

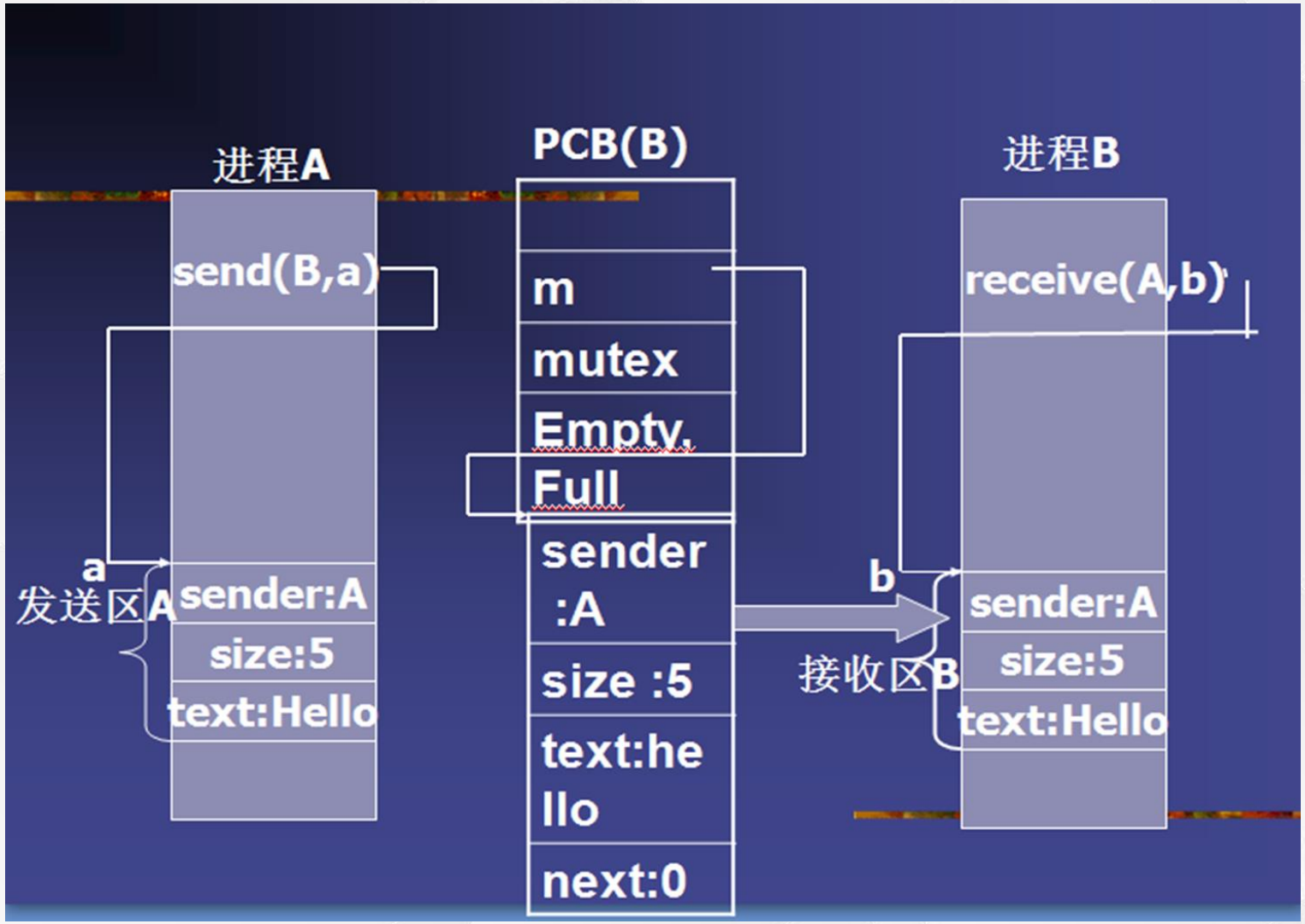
- **进程通信**（Inter Process Communication，IPC）是指进程之间传递信息及同步的机制。
- IPC的基本形式为以下两个操作：
  - 发送信息：**send(destination，message)**
  - 接收信息：**receive(source，message)**
- 基本通信流程
  - 在通信进程间建立通信链路
  - 通过send/receive交换信息

同步方式——阻塞 vs 非阻塞

- **发送端阻塞，接收端阻塞**：两端都阻塞，消息被接收后再将两端唤醒。
  - 常用于要求进程进行强同步的场合。
- **发送端不阻塞，接收端阻塞**：发送端在发送完消息后继续执行，而接收端必须阻塞，直到接收到消息后才被唤醒。
  - 最广泛使用的方案。
  - **问题**：可能导致消息重复发送，消耗系统资源，难以确定消息是否被接收。
- **发送端不阻塞，接收端不阻塞**：两端都不等待。
  - **问题**：可能导致消息接收失败。

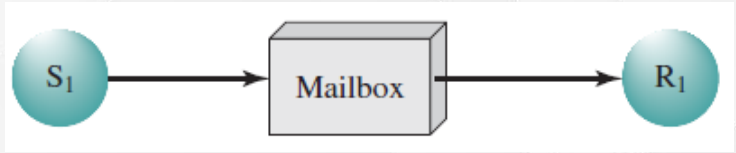
寻址方式——直接 vs 间接

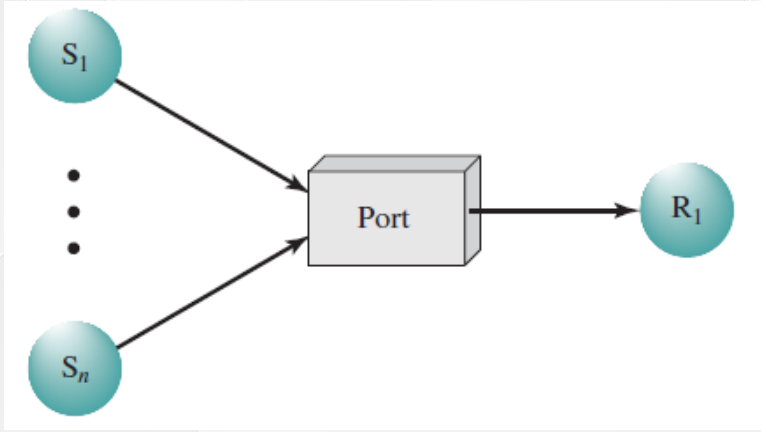
- **直接通信**：发送/接收消息时必须指明消息的接收/发送进程。
- **send(P，message)**
  发送消息到进程P
- **receive(Q，message)**
  从进程Q接收消息
- 通信双方必须同时存在。

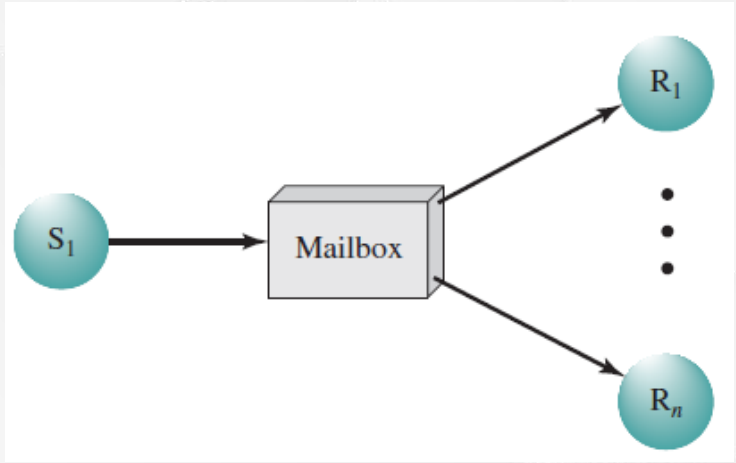


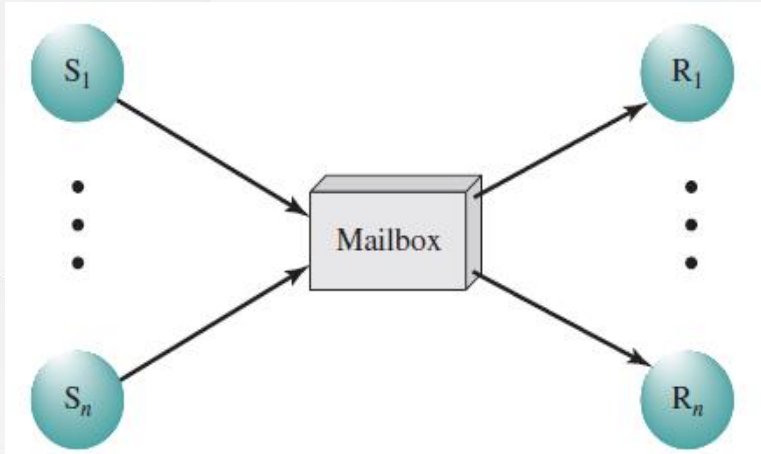
寻址方式——直接 vs 间接

- **间接通信**：消息被发送到一个操作系统维护的消息队列（信箱）中，接收端从其中取出消息。
- 每个消息队列具有一个唯一的标识。
- 只有使用相同消息队列的进程才能够通信。









信箱的归属权——进程 vs 操作系统

- **信箱属于创建它的进程**
  - 随着创建它的进程结束而结束。
  - **例如**：端口
- **信箱属于操作系统**
  - 与使用它的进程是否结束无关，必须通过明确的命令撤销。
  - **例如**：消息队列

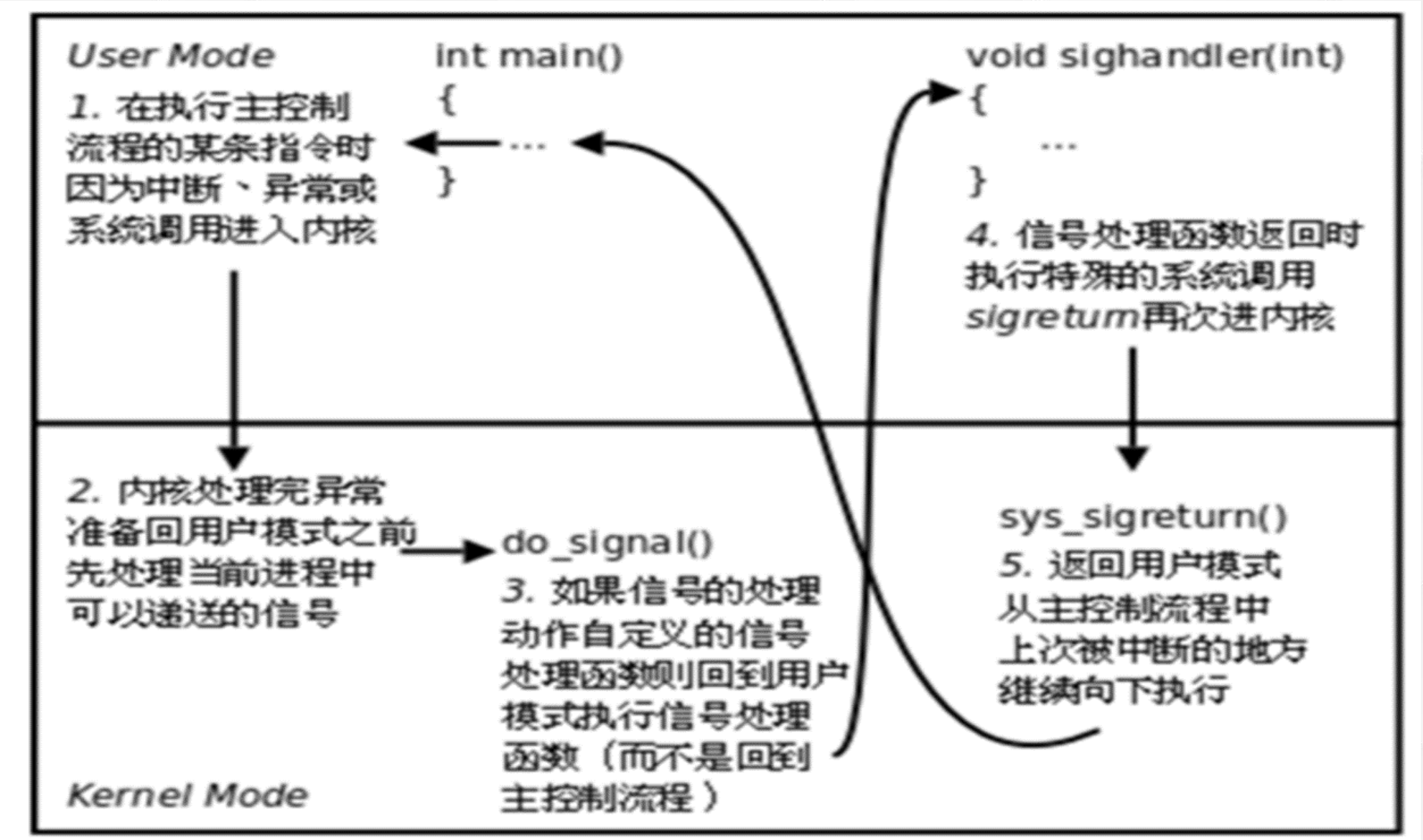
Linux——信号

- 信号机制是一种进程间的软中断信号通知和异步处理机制。
- 信号机制是在软件层次上对中断机制的一种模拟。
- 进程在很多情况下会都接收到软中断信号：
  - **用户在终端按下某些特殊键**：例如Ctrl-C，终端进程会发送SIGINT信号给前台进程。
  - **硬件异常**：例如当前进程执行了除零指令，CPU产生异常，内核处理完该异常后会发送SIGFPE信号给进程。
  - **内核检测到某种软件条件**：如闹钟超时，会发送SIGALRM信号给进程。
  - **显式调用kill函数或kill命令发送信号给某个进程。**

信号机制的功能

- **预置对信号的处理方式**
  - **忽略**：忽略该信号不做处理，SIGKIL和SIGSTOP不能被忽略
  - **捕获**：通知内核在某种信号发生时，调用用户指定的函数进行处理
  - **执行默认动作**：编译器预置了各个信息的处理函数
  - 例如：**signal(SIGINT, func1)**
- **信号发送**：发送进程把信号送到指定进程的信号域的某一位上，如果目标进程正在一个可被中断的优先级上睡眠，系统将其唤醒。
  - 例如：**kill(getppid, SIGUSR1)**
- **信号处理**：收到信号的进程在由核态返回到用户态前，按预置的处理方式完成对相应事件的处理。
- **不足**：传送的信息量小，只有一个信号类型。

信号捕获过程


User Mode
1. 在执行主控制流程的某条指令时因为中断、异常或系统调用进入内核
int main()
{
...
}
void sighandler(int)
{
...
}
4. 信号处理函数返回时执行特殊的系统调用sigreturn再次进内核
2. 内核处理完异常准备回用户模式之前先处理当前进程中可以递送的信号
do_signal()
3. 如果信号的处理动作自定义的信号处理函数则回到用户模式执行信号处理函数（而不是回到主控制流程）
sys_sigreturn()
5. 返回用户模式从主控制流程中上次被中断的地方继续向下执行
Kernel Mode

```
void main() {
    int pid;
    signal(SIGINT, my_func);
    pid=fork();
    if (pid==0) {
       printf("I'm child process.\n");
       kill(getppid(), SIGINT);
       exit(0);
    }
    else {
        wait(0);
        printf("I'm parent process.\n");
        exit(0);
    }
}
```

```
void my_func()
{
    printf("I catch a signal.\n");
}
```

1. 程序的执行结果是什么?
2. 如果删除signal(SIGINT, my_func)，

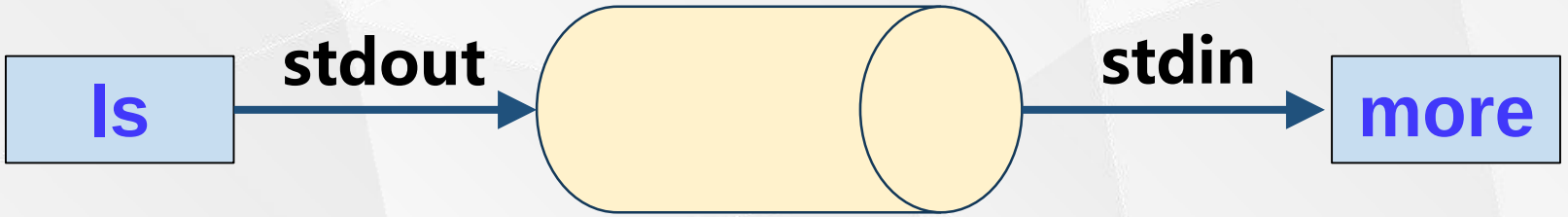
   程序的执行结果是什么?

Linux——管道

- 管道是进程间基于文件的一种间接通信机制。
- 管道是半双工的，数据只能单向流动。
- 管道对于管道两端的进程而言就是一个文件，写管道时内容添加在文件末尾，读管道则从文件头部读出。
- **无名管道**
  - 无名管道没有名字，只有一组文件描述
  - 子进程从父进程继承文件描述符
  - 只能用于具有亲缘关系的进程间的通信
- **命名管道**
  - 管道有文件名，允许无亲缘关系的进程通过访问管道文件进行通信。

管道示例

$ ls | more



与管道相关的系统调用

- **创建无名管道**：**int pipefd[2]; int pipe(pipefd);**
  - pipefd[0]只能用于读，接收进程应关闭写入端，即close(pipefd[1])。
  - pipefd[1]只能用于写，发送进程应关闭读取端，即close(pipefd[0]) 。
- **将数据写入管道**：**write()**
  - 管道长度受到限制，管道满时写操作将被阻塞，直到管道中的数据被读取。
  - fcntl()可将管道写模式设置为非阻塞模式。
- **从管道读取数据**：**read()**
  - 数据被读取后将自动从管道中清除，若管道中没有数据则读操作将被阻塞。
  - fcntl()可将管道读模式设置为非阻塞模式。
- **关闭管道**：**close()**
  - 所有读端口都关闭时，在管道上进行写操作的进程将收到SIGPIPE信号。
  - 所有写端口都关闭时，在管道上进行读操作的read()函数将返回0

与管道相关的系统调用

- **创建命名管道**
  - **int mknod(char *pathname, mode_t mod, dev_t dev);**
  - **int mkfifo(char *pathname, mode_t mode);**
- **命名管道的使用**
  - 命名管道必须先调用open()将其打开，其他与无名管道相似
  - **pipe_read = open("./my_pipe", O_RDONLY, 0);**
  - **pipe_write = open("./my_pipe", O_WRONLY, 0);**
  - 以只读方式(O_RDONLY)打开时，进程将被阻塞直到有写方打开管道
  - 以只写方式(O_WRONLY)打开时，进程将被阻塞直到有读方打开管道
- **命名管道的删除**
  - **int unlink(char *pathname);**