二. 进程与线程

A.进程

1.进程就是指一个正在运行中的程序，有输入，输出，程序，状态，一个单核 CPU 在某一个瞬间只能运行一个程序。

创建进程的方式有这几种：

系统的初始化

一个正在运行中的进程创建的另一个进程的系统调用

用户请求创建一个新的进程

一个批处理作业的初始化

前台进程：用于与用户进行交互

后台进程:与特定的用户无关,却执行着专门的任务,如守护进程(daemon)。

UNIX 中使用 fork()语句创建进程。这个系统调用会创建一个相同的副本，父进程和子进程，拥有两个不同的地址空间。进程，地址空间，文件是操作系统中最要的三个概念。

Windows 中使用 CreateProcess()创建进程。

进程的终止也有 4 种情况：

正常退出

正常错误推出

非正常严重推出

被其他进程杀死

UNIX 中使用 exit()退出进程，Windows 使用 ExitProcess()。第三种退出可能是发生了异常，程序抛出异常结束，等待运行命令。

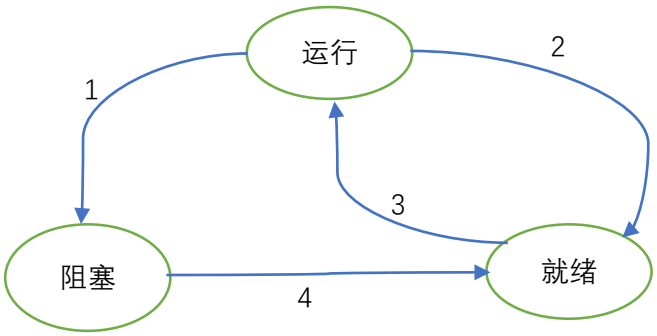
这是一个 fork 的简单应用：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
int main(void)
{
    int n = 0;
    printf("before fork: n = %d\n", n);

    pid_t fpid = fork();
    if(fpid < 0)
    {
        perror("fork error");
        exit(EXIT_FAILURE);
    }
    else if(fpid == 0)
```

```
    {
        n++;
        printf("child_proc(%d, ppid=%d): n = %d\n", getpid(), getppid(), n);
    }
    Else
    {
        n--;
        printf("parent_proc(%d): n = %d\n", getpid(), n);
    }
    printf("quit_proc(%d) ... \n", getpid());
    return 0;
}
```

每个进程虽然都有自己的程序计数器和其状态，但也可能会受其他状态的影响。若一个进程准备运行打印程序，但此时 CPU 正在忙碌状态还没有响应过来，则该程序就会被挂起呈就绪状态。若一个进程正准备调用一个打印机，但此时另一个进程正在使用打印机，此时该进程会被阻塞直到打印机可以用为止。图(1)为进程间各状态的转换图。



1.进程为等待输入而阻塞
2.进程调度程序选择另一个进程
3.调度程序选择这个进程
4.出现有效输入

图 1 进程状态转换图

操作系统会拥有这样一张表格，里面记录着每一个进程重要的状态，包括程序计数器，堆栈指针，内存分配情况，调度信息，账号等，这张表叫进程表(process table)，或称为进程控制块（process control block)。当发生中断时，或该进程被操作系统置为就绪或阻塞态时，该表将记录进程运行前的所有状态以便在下次运行时可以继续之前的操作。

中断向量：即一个数组，里面包含有中断程序的入口地址，该地址指向发生中断的进程首地址。

B. 线程

线程就是轻量级的进程，一个进程可以包含多个线程，这些线程可以并行，使得一个复杂的进程拆分为更小的程序，线程的创建和删除都不占用太多内存。多线程经常用于 I/O 密集型程序。每个线程会公用一个内存空间。线程的状态和进程的状态是相同的。

实现线程的方式有两种：用户空间和内核中。

用户级线程包图(3a)可以在不支持线程的操作系统上运行，此时每个进程都拥有各自的线程表。也允许用户自行定制自己的调度算法，有良好的扩展性。但最大的问题是线程的阻塞是很难实现的，且有很大风险使线程停止运行。

内核中实现图(3b)，内核中存在线程表，并跟踪每个线程的运行情况，但由于在内核中创建和删除线程成本过高，把线程完全放置在内核中是不妥当的。图(3b)

还有一种混合式的，就是把用户级线程和内核级线程混合起来。

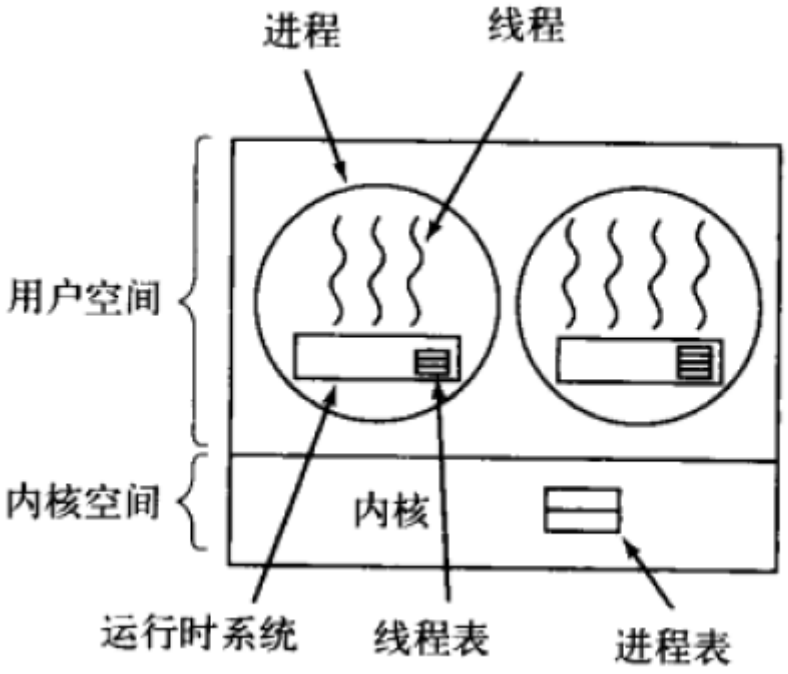


图 3a 用户级线程

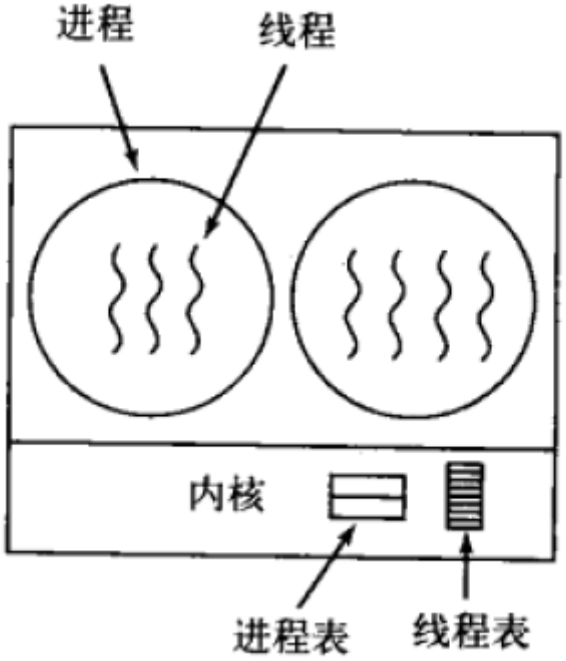


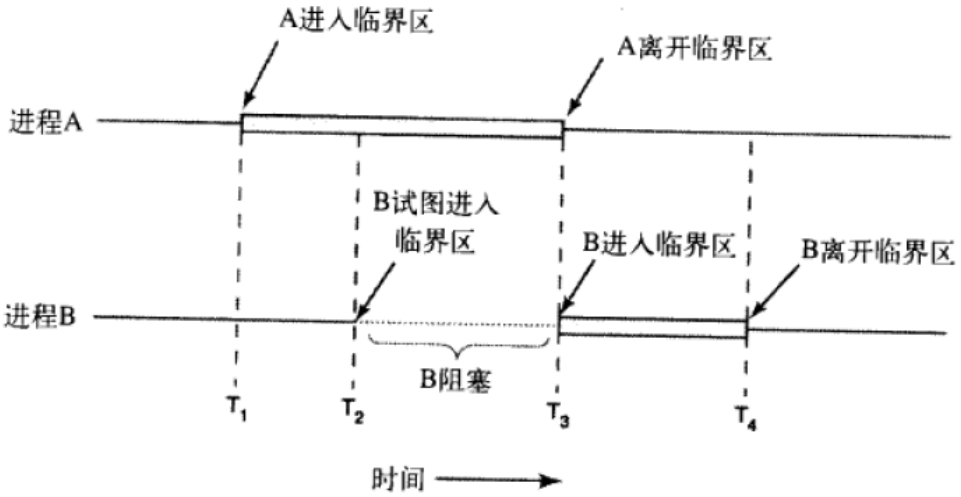
图 3b 内核级线程

C.进程间通信(Inter Process Communication, IPC)

指的是各个进程不是互相独立的，而是相互合作，进行通信联系的。进程间通信可以移植到线程间通信。有三个问题需要解决：如何传递消息，多个进程传递消息过程中如何保证不会出现影响，如果顺序很重要的话如何排列。前两个问题对于线程同样重要。

1)竞争条件：两个或两个以上的进程在读写共享数据时，最后的结果与进程运行的精确时间有关。这可能会导致这几个进程都由于无法获得有效输入而被阻塞。

该共享数据称为临界区。避免竞争条件的条件：a.任何两个进程不能同时处于临界区，b.不应对 CPU 的数量和速度做任何假设，c.临界区外运行的进程不能阻塞其他进程，d.进程等待进入临界区的时间必须是有限的。例如有 A,B 两个进程，当 A 进入临界区时，B 试图进入，则操作系统会检测发现临界区已经有一个进程存在，则阻塞 B，当系统检测到临界区无进程时则允许 B 进入临界区。A，B 进程互斥。



2)信号量(semaphore)：使用一个整型变量来累计唤醒的次数。down 睡眠，up 唤醒，有些书中写为 PV 量。用来表示进程是否运行的标志。当信号量为零时，说明此时没有进程在运行，可以允许进程进入临界区并将信号量置为非零。当信号量不是零时，说明有进程在临界区中，令其他进程进入等待状态。

生产者-消费者问题（又称有界缓冲区问题），是两个或多个进程共享同一块缓冲区的问题。生产者是将数据放入缓冲区，消费者将数据取出缓冲区。

```
#define FALSE  0
#define TRUE   1
#define N      2                                    /* 进程数量 */

int turn;                                           /* 现在轮到谁？ */
int interested[N];                                  /* 所有值初始化为0（FALSE） */

void enter_region(int process);                     /* 进程是0或1 */
{
     int other;                                     /* 其他进程号 */

     other = 1 - process;                           /* 另一方进程*/
     interested[process] = TRUE;                    /* 表明所感兴趣的 */
     turn = process;                                /* 设置标志 */
     while (turn == process && interested[other] == TRUE); /*空语句 */
}

void leave_region(int process)                      /* 进程：谁离开？ */
{
     interested[process] = FALSE;                   /* 表示离开临界区 */
}
```

图2-24　完成互斥的Peterson解法

```
#define N 100                                   /* 缓冲区中的槽数目 */
typedef int semaphore;                          /* 信号量是一种特殊的整型数据 */
semaphore mutex = 1;                            /* 控制对临界区的访问 */
semaphore empty = N;                            /* 计数缓冲区的空槽数目 */
semaphore full = 0;                             /* 计数缓冲区的满槽数目 */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE是常量1 */
          item = produce_item();                /* 产生放在缓冲区中的一些数据 */
          down(&empty);                         /* 将空槽数目减1 */
          down(&mutex);                         /* 进入临界区 */
          insert_item(item);                    /* 将新数据项放到缓冲区中 */
          up(&mutex);                           /* 离开临界区 */
          up(&full);                            /* 将满槽的数目加1 */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* 无限循环 */
          down(&full);                          /* 将满槽数目减1 */
          down(&mutex);                         /* 进入临界区 */
          item = remove_item();                 /* 从缓冲区中取出数据项 */
          up(&mutex);                           /* 离开临界区 */
          up(&empty);                           /* 将空槽数目加1 */
          consume_item(item);                   /* 处理数据项 */
     }
}
```

图2-28 使用信号量的生产者-消费者问题

Pthread 中的互斥，基本机制是使用一个可以被锁定和解锁的互斥量来保护每个临界区。

3)管程，由变量，过程及数据结构等组成的集合，形成一个特殊的软件包。一个很重要的性质是**任意时刻管程中只有一个运行的进程**。便于对进程进行管理。

当一个进程准备运行时，它将调用管程，管程此时检测其内部是否有正在运行的进程，如果没有则该进程可以进入，如果有，则阻塞该进程直到管程内部无进程运行。

```
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;

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;
```

图2-34 用管程实现的生产者-消费者问题的解法框架。一次只能有一个管程过程活跃。其中的缓冲区有N个槽