第四章 进程管理

为了描述程序在并发执行时对系统资源的共享，我们需要一个描述程序执行时动态特征的概念，这就是进程。在本章中，我们将讨论进程概念、进程控制和进程间关系。

4.1 进程(PROCESS)

返回

4.1.1 程序的顺序执行和并发执行

- 程序的执行有两种方式：顺序执行和并发执行。
  - 顺序执行是单道批处理系统的执行方式，也用于简单的单片机系统；
  - 现在的操作系统多为并发执行，具有许多新的特征。引入并发执行的目的是为了提高资源利用率。

- 顺序执行的特征
  - 顺序性：按照程序结构所指定的次序（可能有分支或循环）
  - 封闭性：独占全部资源，计算机的状态只由于该程序的控制逻辑所决定
  - 可再现性：初始条件相同则结果相同。如：可通过空指令控制时间关系。
- 并发执行的特征
  - 间断(异步)性："走走停停"，一个程序可能走到中途停下来，失去原有的时序关系；
  - 失去封闭性：共享资源，受其他程序的控制逻辑的影响。如：一个程序写到存储器中的数据可能被另一个程序修改，失去原有的不变特征。
  - 失去可再现性：失去封闭性 ->失去可再现性；外界环境在程序的两次执行期间发生变化，失去原有的可重复特征。

并发执行的条件：达到封闭性和可再现性

并发执行失去封闭性的原因是共享资源的影响，去掉这种影响就行了。1966年，由Bernstein给出并发执行的条件。（这里没有考虑执行速度的影响。）

- 程序 P(i) 针对共享变量的读集和写集 R(i)和W(i)
- 条件：任意两个程序P(i)和P(j)，有：
  - $R(i) \cap W(j) = \Phi$;
  - $W(i) \cap R(j) = \Phi$;
  - $W(i) \cap W(j) = \Phi$;

前两条保证一个程序的两次读之间数据不变化；最后一条保证写的结果不丢掉。

现在的问题是这个条件不好检查。

4.1.2 进程的定义和描述

1. 进程的定义

一个具有一定独立功能的程序在一个数据集合上的一次动态执行过程。

- 它对应虚拟处理机、虚拟存储器和虚拟外设等资源的分配和回收；
- 引入多进程，提高了对硬件资源的利用率，但又带来额外的空间和时间开销，增加了OS 的复杂性；

2. 进程的特征

- 动态性：进程具有动态的地址空间（数量和内容），地址空间上包括：
  - 代码（指令执行和CPU状态的改变）
  - 数据（变量的生成和赋值）
  - 系统控制信息（进程控制块的生成和删除）
- 独立性：各进程的地址空间相互独立，除非采用进程间通信手段；
- 并发性、异步性："虚拟"
- 结构化：代码段、数据段和核心段（在地址空间中）；程序文件中通常也划分了代码段和数据段，而核心段通常就是OS核心（由各个进程共享，包括各进程的PCB）

3. 进程与程序的区别

- 进程是动态的，程序是静态的：程序是有序代码的集合；进程是程序的执行。通常进程不可在计算机之间迁移；而程序通常对应着文件、静态和可以复制。
- 进程是暂时的，程序的永久的：进程是一个状态变化的过程，程序可长久保存。
- 进程与程序的组成不同：进程的组成包括程序、数据和进程控制块（即进程状态信息）。
- 进程与程序的对应关系：通过多次执行，一个程序可对应多个进程；通过调用关系，一个进程可包括多个程序。

4. 处理机调度器（dispatcher）

处理机调度器是操作系统中的一段代码，它完成如下功能：

- 把处理机从一个进程切换到另一个进程；
- 防止某进程独占处理机；

5. 进程控制块 (PCB, process control block)

进程控制块是由OS维护的用来记录进程相关信息的一块内存。

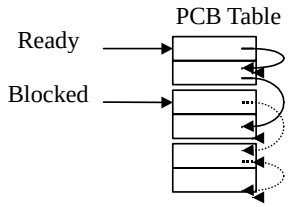
- 每个进程在OS中的登记表项（可能有总数目限制），OS据此对进程进行控制和管理（PCB中的内容会动态改变），不同OS则不同
- 处于核心段，通常不能由应用程序自身的代码来直接访问，而要通过系统调用，或通过UNIX中的进程文件系统(/proc)直接访问进程映象(image)。文件名为进程标识（如：00316），权限为创建者可读写。

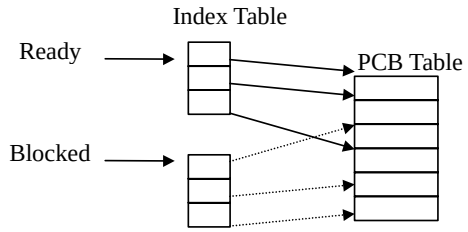
进程控制块的内容

- 进程描述信息：
  - 进程标识符(process ID)，唯一，通常是一个整数；
  - 进程名，通常基于可执行文件名（不唯一）；
  - 用户标识符(user ID)；进程组关系(process group)
- 进程控制信息：
  - 当前状态；
  - 优先级(priority)；
  - 代码执行入口地址；
  - 程序的外存地址；
  - 运行统计信息（执行时间、页面调度）；
  - 进程间同步和通信；阻塞原因
- 资源占用信息：虚拟地址空间的现状、打开文件列表
- CPU现场保护结构：寄存器值（通用、程序计数器PC、状态PSW，地址包括栈指针）

6. PCB的组织方式

- 链表：同一状态的进程其PCB成一链表，多个状态对应多个不同的链表
  - 各状态的进程形成不同的链表：就绪链表、阻塞链表
- 索引表：同一状态的进程归入一个index表（由index指向PCB），多个状态对应多个不同的index表
  - 各状态的进行形成不同的索引表：就绪索引表、阻塞索引表





7. 进程上下文

进程上下文是对进程执行活动全过程的静态描述。进程上下文由进程的用户地址空间内容、硬件寄存器内容及与该进程相关的核心数据结构组成。

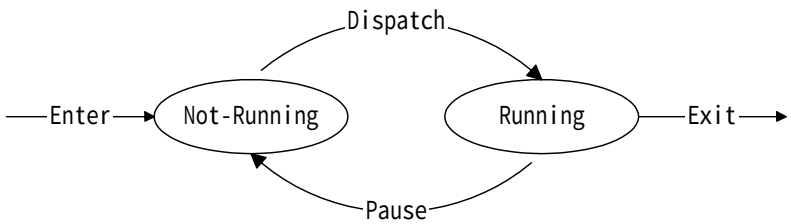
- 用户级上下文：进程的用户地址空间（包括用户栈各层次），包括用户正文段、用户数据段和用户栈；
- 寄存器级上下文：程序寄存器、处理机状态寄存器、栈指针、通用寄存器的值；
- 系统级上下文：
  - 静态部分（PCB和资源表格）
  - 动态部分：核心栈（核心过程的栈结构，不同进程在调用相同核心过程时有不同核心栈）

核心态和用户态

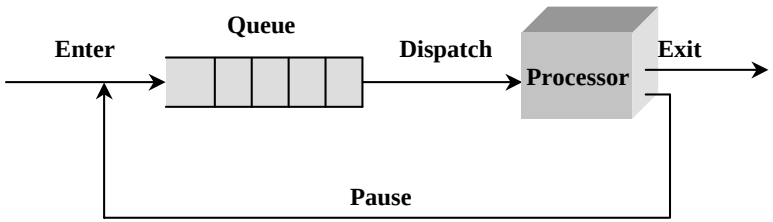
- 用户态时不可直接访问受保护的OS代码；
- 核心态时执行OS代码，可以访问全部进程空间。

4.1.3 进程的状态转换

4.1.3.1 两状态进程模型



(a)状态变迁图



(b)队列结构

1. 状态

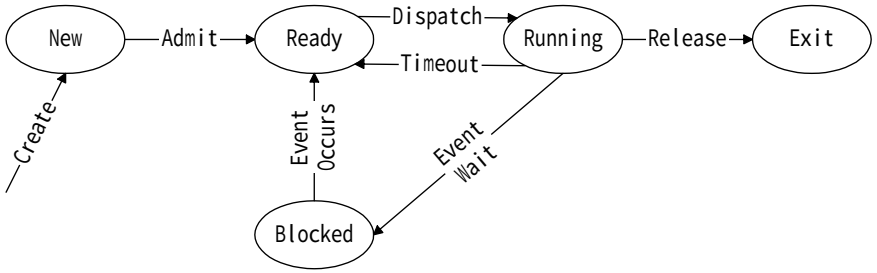
- 运行状态(Running)：占用处理机资源；
- 暂停状态(Not-Running)：等待进程调度分配处理机资源；

2. 转换

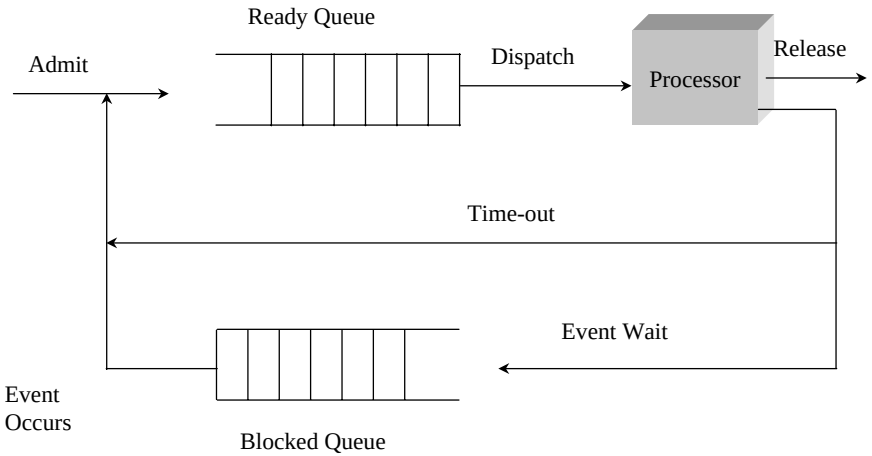
- 进程创建(Enter)：系统创建进程，形成PCB，分配所需资源，排入暂停进程表(可为一个队列)；
- 调度运行(Dispatch)：从暂停进程表中选择一个进程(要求已完成I/O操作)，进入运行状态；
- 暂停运行(Pause)：用完时间片或启动I/O操作后，放弃处理机，进入暂停进程表；
- 进程结束(Exit)：进程运行中止；

4.1.3.2 五状态进程模型

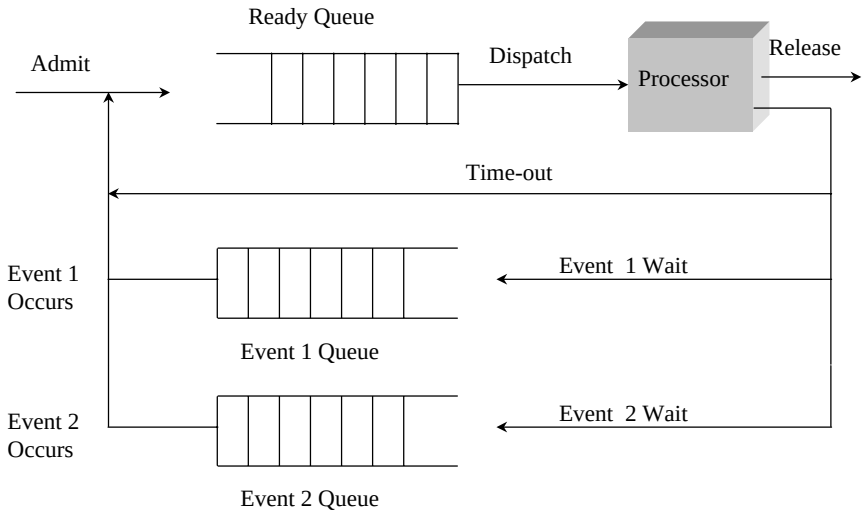
两状态模型无法区分暂停进程表中的可运行和阻塞，五状态模型就是对暂停状态的细化。



五状态进程模型(状态变迁)



五状态进程模型(单队列结构)



五状态进程模型(多队列结构)

1. 状态

- 运行状态(Running)：占用处理机资源；处于此状态的进程的数目小于等于CPU的数目。
  - 在没有其他进程可以执行时（如所有进程都在阻塞状态），通常会自动执行系统的idle进程（相当于空操作）。
- 就绪状态(Ready)：进程已获得除处理机外的所需资源，等待分配处理机资源；只要分配CPU就可执行。
  - 可以按多个优先级来划分队列，如：时间片用完－>低优，I/O完成－>中优，页面调入完成－>高优
- 阻塞状态(Blocked)：由于进程等待某种条件（如I/O操作或进程同步），在条件满足之前无法继续执行。该事件发生前即使把处理机分配给该进程，也无法运行。如：等待I/O操作的完成。

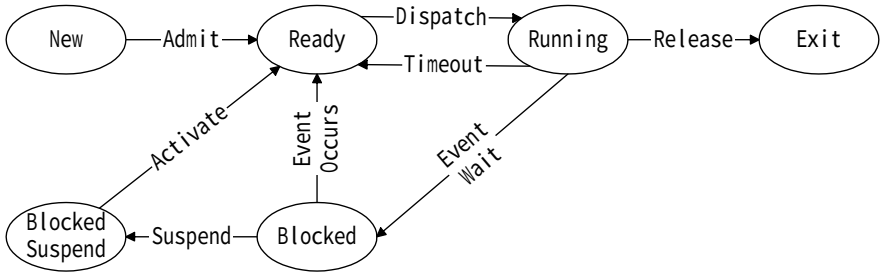
- 创建状态(New)：进程刚创建，但还不能运行(一种可能的原因是OS对并发进程数的限制)；如：分配和建立PCB表项（可能有数目限制）、建立资源表格（如打开文件表）并分配资源，加载程序并建立地址空间表。
- 结束状态(Exit)：进程已结束运行，回收除PCB之外的其他资源，并让其他进程从PCB中收集有关信息（如记帐，将退出码exit code传递给父进程）。

2. 转换

- 创建新进程：创建一个新进程，以运行一个程序。可能的原因为：用户登录、OS创建以提供某项服务、批处理作业。
- 收容(Admit, 也称为提交)：收容一个新进程，进入就绪状态。由于性能、内存、进程总数等原因，系统会限制并发进程总数。
- 调度运行(Dispatch)：从就绪进程表中选择一个进程，进入运行状态；
- 释放(Release)：由于进程完成或失败而中止进程运行，进入结束状态；
  - 运行到结束：分为正常退出Exit和异常退出abort（执行超时或内存不够，非法指令或地址，I/O失败，被其他进程所终止）
  - 就绪或阻塞到结束：可能的原因有：父进程可在任何时间中止子进程；

- 超时（Timeout）：由于用完时间片或高优先进程就绪等导致进程暂停运行；
- 事件等待（Event Wait）：进程要求的事件未出现而进入阻塞；可能的原因包括：申请系统服务或资源、通信、I/O操作等；
- 事件出现（Event Occurs）：进程等待的事件出现；如：操作完成、申请成功等；

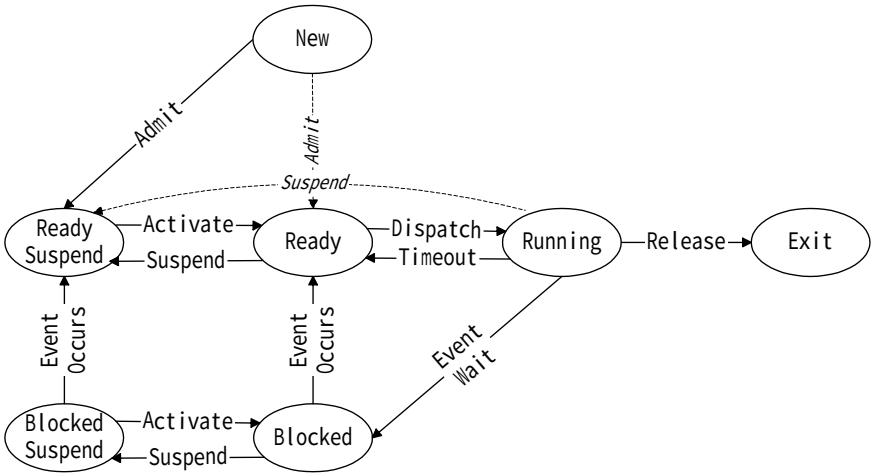
注：对于五状态进程模型，一个重要的问题是当一个事件出现时如何检查阻塞进程表中的进程状态。当进程多时，对系统性能影响很大。一种可能的作法是按等待事件类型，排成多个队列。



单挂起进程模型

4.1.3.3 挂起进程模型

- 这个问题的出现是由于进程优先级的引入，一些低优先级进程可能等待较长时间，从而被对换至外存。这样做的目的是：
  - 提高处理机效率：就绪进程表为空时，要提交新进程，以提高处理机效率；
  - 为运行进程提供足够内存：资源紧张时，暂停某些进程，如：CPU繁忙（或实时任务执行），内存紧张
  - 用于调试：在调试时，挂起被调试进程（从而对其地址空间进行读写）



双挂起进程模型

1. 状态

注：这里只列出了意义有变化或新的状态。

- 就绪状态(Ready)：进程在内存且可立即进入运行状态；
- 阻塞状态(Blocked)：进程在内存并等待某事件的出现；
- 阻塞挂起状态（Blocked, suspend）：进程在外存并等待某事件的出现；
- 就绪挂起状态（Ready, suspend）：进程在外存，但只要进入内存，即可运行；

2. 转换

- 挂起（Suspend）：把一个进程从内存转到外存；可能有以下几种情况：
  - 阻塞到阻塞挂起：没有进程处于就绪状态或就绪进程要求更多内存资源时，会进行这种转换，以提交新进程或运行就绪进程；
  - 就绪到就绪挂起：当有高优先级阻塞（系统认为会很快就绪的）进程和低优先级就绪进程时，系统会选择挂起低优先级就绪进程；
  - 运行到就绪挂起：对抢先式分时系统，当有高优先级阻塞挂起进程因事件出现而进入就绪挂起时，系统可能会把运行进程转到就绪挂起状态；
- 激活（Activate）：把一个进程从外存转到内存；可能有以下几种情况：
  - 就绪挂起到就绪：没有就绪进程或挂起就绪进程优先级高于就绪进程时，会进行这种转换；
  - 阻塞挂起到阻塞：当一个进程释放足够内存时，系统会把一个高优先级阻塞挂起（系统认为会很快出现所等待的事件）进程；

- 事件出现（Event Occurs）：进程等待的事件出现；如：操作完成、申请成功等；可能的情况有：
  - 阻塞到就绪：针对内存进程的事件出现；
  - 阻塞挂起到就绪挂起：针对外存进程的事件出现；
- 收容(Admit)：收容一个新进程，进入就绪状态或就绪挂起状态。进入就绪挂起的原因是系统希望保持一个大的就绪进程表（挂起和非挂起）；

4.1.4 操作系统代码的执行

通常，OS核心不是一个进程，其执行不被调度。

- OS和进程的关系：
  - OS不作为进程地址空间的一部分：传统方法。
  - OS作为进程地址空间的一部分：如UNIX
  - OS功能分别在核心和系统服务进程中，只有OS核心作为进程地址空间的一部分：如Windows NT

4.2 进程控制

返回

4.2.1 进程控制的功能

完成进程状态的转换。

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－－要么全都完成，要么全都不做。许多系统调用就是原语。

注意：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。

4.2.2 进程的创建和退出

1. 创建

	产生新进程	不产生新进程
复制现有进程的上下文	fork（新进程的系统上下文会有不同）	–
加载程序	spawn（等待子进程完成，）	exec（加载新程序并覆盖自身）

- 继承(inherit)：子进程可以从父进程中继承用户标识符、环境变量、打开文件、文件系统的当前目录、控制终端、已经连接的共享存储区、信号处理例程入口表等
- 不被继承：进程标识符，父进程标识符
- spawn创建并执行一个新进程；新进程与父进程的关系可有多种：覆盖(_P_OVERLAY)、并发(_P_NOWAIT or _P_NOWAITO)、父进程阻塞(_P_WAIT)、后台(_P_DETACH)等。

2. 退出

- 也称为“终止”或主程序返回：调用exit()可终止进程。
- 释放资源：
  - 释放内外存空间
  - 关闭所有打开文件
  - 释放共享内存段和各种锁定lock

4.2.3 UNIX进程的阻塞和唤醒

- 阻塞：
  - 暂停一段时间(sleep)；
  - 暂停并等待信号(pause)；
  - 等待子进程暂停或终止(wait)；
- 唤醒：发送信号到某个或一组进程(kill)

- 调用wait挂起本进程以等待子进程的结束，子进程结束时返回。父进程创建多个子进程且已有某子进程退出时，父进程中wait函数在第一个子进程结束时返回。
  - 其调用格式为"pid_t wait(int *stat_loc);"；返回值为子进程ID。
  - waitpid()等待指定进程号的子进程的返回并修改状态；
  - waitid()等待子进程修改状态；
- 调用pause挂起本进程以等待信号，接收到信号后恢复执行。当接收到中止进程信号时，该调用不再返回。
  - 其调用格式为"int pause(void);"；

- 调用sleep将在指定的时间seconds内挂起本进程。其调用格式为："unsigned sleep(unsigned seconds);"；返回值为实际的挂起时间。
- 调用kill可发送信号sig到某个或一组进程pid。其调用格式为："int kill(pid_t pid, int sig);"。信号的定义在文件"/usr/ucbinclude/sys/signal.h"中。命令"kill"可用于向进程发送信号。如："kill -9 100"将发送SIGKILL到ID为100的进程；该命令将中止该进程的执行。

实例：UNIX_wait

演示子进程与父进程的关系和fork、exec、wait的使用；

程序main.c

功能是进行10次循环，创建2个子进程。循环到第3次时，等待子进程结束。

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
void perror(const char *s);
#include <errno.h>
int errno;
int global;
```

```
main() {
  int local,i;
  pid_t child;

  if ((child=fork()) == -1)
  {       // 创建失败
    printf("Fork Error.\n");
  }
  if (child == 0)
  {// 子进程
    printf("Now it is in child process.\n");
    if (execl("/home/xyong/work/ttt","ttt",NULL) == -1)
            {       // 加载程序失败
              perror("Error in child process");
            }
    global=local + 2;
    exit();
  }
```

```
// 父进程
 printf("Now it is in parent process.\n");
 for (i=0; i<10; i++)
 {
            sleep(2);
            printf("Parent:%d\n",i);
            if (i==2)
            {
 if ((child=fork()) == -1)
 {          // 创建失败
   printf("Fork Error.\n");
 }
 if (child == 0)
 {// 子进程
   printf("Now it is in child process.\n");
   if (execl("/home/xyong/work/ttt","ttt",NULL) == -1)
              {          // 加载程序失败
                perror("Error in child process");
              }
   global=local + 2;
   exit();
 }
            }
```

```
            if (i==3)
            {
              pid_t temp;
              temp=wait(NULL);
              printf("Child process ID: %d\n", temp);
            }
        }
        global=local + 1;
        exit();
      }
```

程序test.c

功能是进行10次循环。

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
int global;
main() {
  int local;
  int i;
  pid_t CurrentProcessID, ParentProcessID;
  CurrentProcessID=getpid();
  ParentProcessID=getppid();
  printf("Now it is in the program TEST.\n");
  for (i=0; i<10; i++)
    {
          sleep(2);
          printf("Parent: %d, Current: %d, Nunber:%d\n",ParentProcessID,
                CurrentProcessID,i);
    }
  global=local + 1;
  exit();
}
```

Now it is in parent process.
Now it is in child process.
Now it is in the program TEST.
Parent:0
Parent: 7072, Current: 7073, Nunber:0
Parent:1
Parent: 7072, Current: 7073, Nunber:1
Parent:2
Parent: 7072, Current: 7073, Nunber:2
Now it is in child process.
Now it is in the program TEST.
Parent: 7072, Current: 7073, Nunber:3
Parent:3
Parent: 7072, Current: 7074, Nunber:0
Parent: 7072, Current: 7073, Nunber:4
Parent: 7072, Current: 7074, Nunber:1
Parent: 7072, Current: 7073, Nunber:5
Parent: 7072, Current: 7074, Nunber:2
Parent: 7072, Current: 7073, Nunber:6
Parent: 7072, Current: 7074, Nunber:3
Parent: 7072, Current: 7073, Nunber:7

结果

Parent: 7072, Current: 7074, Nunber:4
Parent: 7072, Current: 7073, Nunber:8
Parent: 7072, Current: 7074, Nunber:5
Parent: 7072, Current: 7073, Nunber:9
Child process ID: 7073
Parent: 7072, Current: 7074, Nunber:6
Parent:4
Parent: 7072, Current: 7074, Nunber:7
Parent:5
Parent: 7072, Current: 7074, Nunber:8
Parent:6
Parent: 7072, Current: 7074, Nunber:9
Parent:7
Parent:8
Parent:9

4.2.4 NT线程的挂起和激活

NT中处理机的分配对象为线程，一个线程可被多次挂起和多次激活。在线程内有一个挂起计数（suspend count），挂起操作使该计数加1，激活操作便该计数减1；当挂起计数为0时，线程恢复执行。

(1) 挂起：Windows NT中的SuspendThread可挂起指定的线程；

```
DWORD SuspendThread( HANDLE hThread
 // handle to the thread
);
```

(2) 激活：Windows NT中的ResumeThread可恢复指定线程的执行；

```
DWORD ResumeThread( HANDLE hThread
 // identifies thread to restart
);
```

例子NT_thread.cpp

```
#include <stdio.h>
#include <windows.h>
#include <iostream.h>
#include <winbase.h>

void SubThread(void)
{
        int i;
        for (i=0;i<5;i++)
        {
                cout << "SubThread" << i << endl;
                Sleep(2000);
        }
}
```

```
void main(void)
{
   cout << "CreateThread" << endl;

      // Create a thread;
      DWORD IDThread;
      HANDLE hThread;
      hThread = CreateThread(NULL, // no security attributes
    0,                    // use default stack size
    (LPTHREAD_START_ROUTINE) SubThread, // thread function
    NULL,                 // no thread function argument
    0,                  // use default creation flags
    &IDThread);           // returns thread identifier

      // Check the return value for success.
      if (hThread == NULL)
            cout << "CreateThread error" << endl;
```

```
        int i;
        for (i=0;i<5;i++)
        {
                cout << "MainThread" << i << endl;
                if (i==1)
                {
                        if (SuspendThread(hThread)==0xFFFFFFFF)
                        {
                                cout << "Suspend thread error." << endl;
                        }
                        else
                        {
                                cout << "Suspend thread is ok." << endl;
                        }
                }
                if (i==3)
                {
                        if (ResumeThread(hThread)==0xFFFFFFFF)
                        {
                                cout << "Resume thread error." << endl;
                        }
                        else
                        {
                                cout << "Resume thread is ok." << endl;
                        }
                }
                Sleep(4000);
        }
}
```

运行结果

```
CreateThread
MainThread0
SubThread0
SubThread1
MainThread1
Suspend thread is ok.
MainThread2
MainThread3
Resume thread is ok.
SubThread2
SubThread3
MainThread4
SubThread4
```

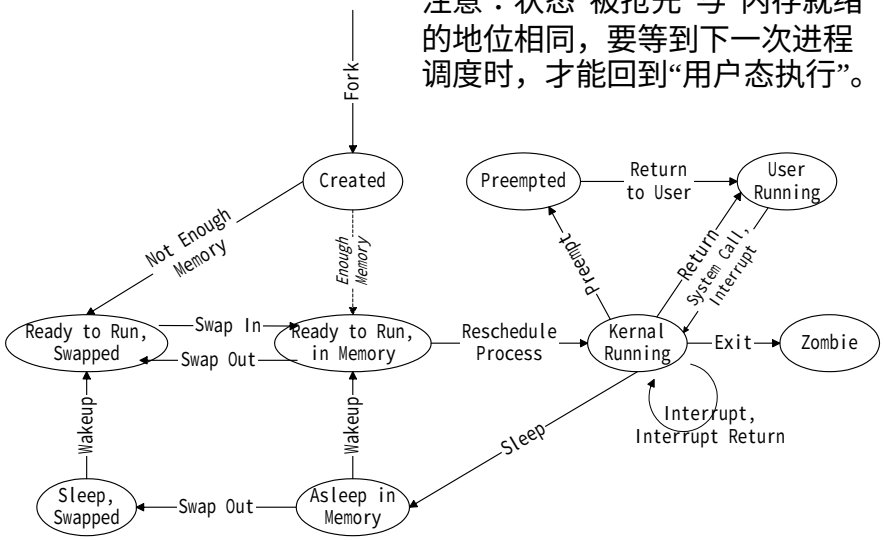
4.2.5 UNIX进程管理举例

1. 进程结构

- 进程上下文：指进程的用户地址空间内容、寄存器内容及与进程相关的核心数据结构；包括三部分"上下文"：用户级、寄存器级、系统级
- 用户级上下文：正文段即代码(text)；数据段(data)；栈段(user stack)：用户态执行时的过程调用；共享存储区(shared memory)
  - 把地址空间的段称为"区(region)"：进程区表和系统区表（前者索引指向后者）
- 系统上下文：
  - proc结构：总在内存，内容包括阻塞原因；
  - user结构：可以调出到外存，进程处于执行状态时才用得着，各种资源表格；
  - 进程区表：从虚拟地址到物理地址的映射；
  - 核心栈：核心态执行时的过程调用的栈结构；

2. 进程状态转换

注意：状态“被抢先”与“内存就绪”的地位相同，要等到下一次进程调度时，才能回到“用户态执行”。

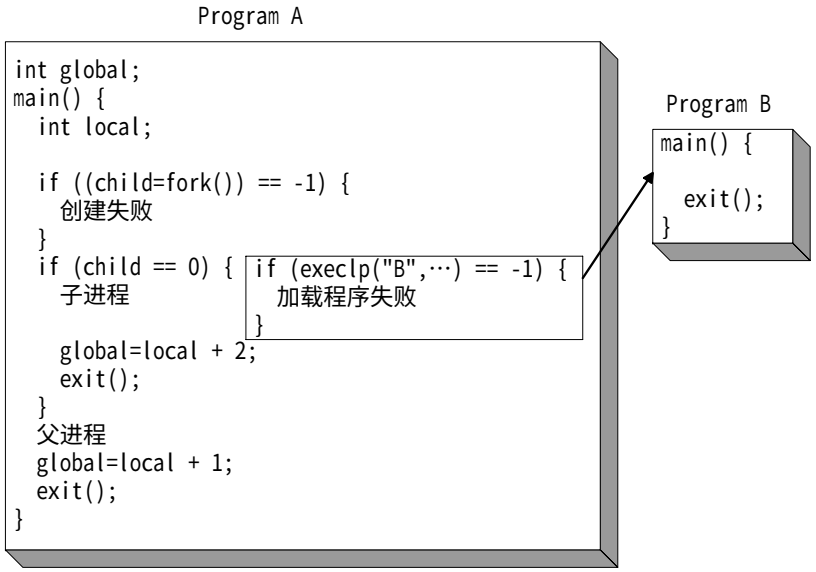


3. 进程控制

创建：fork(), exec()

- 父子进程的fork()返回值不同，Parent PID的值不同 getppid()
- fork()创建子进程之后，执行返回父进程，或调度切换到子进程以及其他进程
- fork创建一个新进程（子进程），子进程是父进程的精确复制。在子进程中返回为0；在父进程中，返回子进程的标识。子进程是从fork调用返回时在用户态开始运行的。父进程的返回点与子进程的开始点是相同的。
- exec用一个新进程覆盖调用进程。它的参数包括新进程对应的文件和命令行参数。成功调用时，不再返回；否则，返回出错原因。

UNIX进程创建



- 退出：exit()向父进程给出一个退出码（8位的整数）。父进程终止时如何影响子进程：
  - 子进程从父进程继承了进程组ID和终端组ID（控制终端），因此子进程对发给该进程组或终端组的信号敏感。终端关闭时，以该终端为控制终端的所有进程都收到SIGHUP信号。
  - 子进程终止时，向父进程发送SIGCHLD信号，父进程截获此信号并通过wait3()系统调用来释放子进程PCB。

- 阻塞：暂停一段时间sleep；暂停并等待信号pause；等待子进程暂停或终止wait：可以取得子进程的退出码（16位整数：exit退出时exitCode*256，信号终止时coreFlag*128 + signalNumber，信号停止时signalNumber*256 + 0x7f）
- 唤醒：发送信号到某个或一组进程kill：使得接收方从阻塞的系统调用中返回（如read返回并给出失败值 -1），并随即调用相应的信号处理例程
- 调试：设置执行断点(breakpoint)，读写进程映象中的数据（从而修改进程的上下文）。系统调用ptrace()，信号SIGTRAP（由调试方发送到被调试方）ptrace允许父进程控制子进程的运行，可用于调试。子进程在遇到signal时暂停，父进程可访问core image。

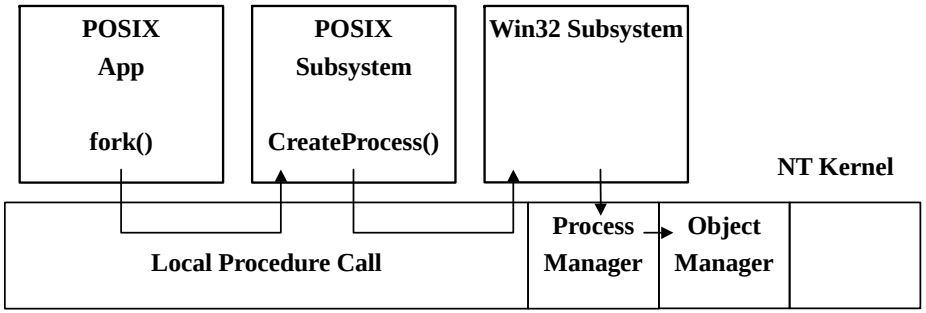
4.2.6 Windows NT进程管理举例

1. 概述

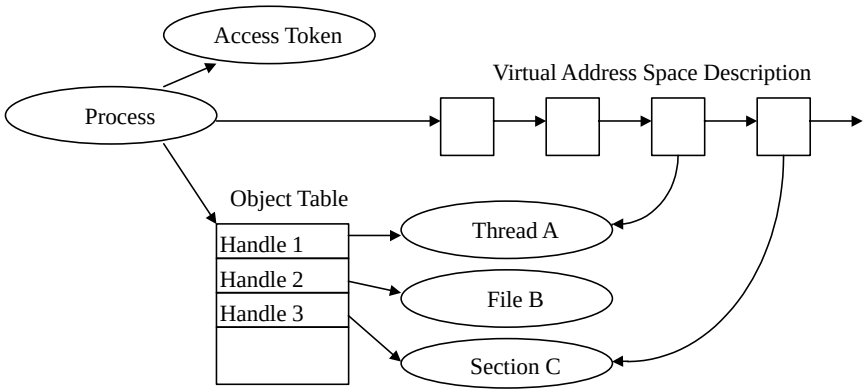
- NT的进程和线程作为对象(Object)，以句柄(handle)来引用。相应地有控制对象的服务(services)。
  - 进程对象的属性；PID, Access Token, Base Priority, 默认处理器集合等

NT的进程关系

- 对NT核心而言，进程之间没有任何关系（包括父子关系）。那么，如何表达UNIX进程之间的父子关系（以及其他关系）？由POSIX子系统来建立和维护



2. NT进程结构



3. 进程控制

- 创建：CreateProcess()函数用于创建新进程及其主线程，以执行指定的程序。
  - 新进程可以继承：打开文件的句柄、各种对象（如进程、线程、信号量、管道等）的句柄、环境变量、当前目录、原进程的控制终端、原进程的进程组（用于发送Ctrl+C或Ctrl+Break信号给多个进程）——每个句柄在创建或打开时能指定是否可继承；
  - 新进程不能继承：优先权类、内存句柄、DLL模块句柄
  - CREATE_NEW_CONSOLE表示新进程有一个新的控制台
  - CREATE_NEW_PROCESS_GROUP表示新进程是一个新的进程组的根。

- 退出：ExitProcess()或TerminateProcess()，则进程包含的线程全部终止；
  - ExitProcess()终止一个进程和它的所有线程；它的终止操作是完整的，包括关闭所有对象句柄、它的所有线程等；
  - TerminateProcess()终止指定的进程和它的所有线程；它的终止操作是不完整的（如：不向相关DLL通报关闭情况），通常只用于异常情况下对进程的终止。

4. 调试

- 进程对象的属性包括：调试时用于通知另一进程（调试器）的IPC channel
- 调试器在CreateProcess时指定DEBUG_PROCESS标志或利用DebugActiveProcess()函数，可在调用者（debugger）与被调用者（target）间建立调试关系，target会向debugger通报所有调试事件；
- 被调试进程向调试器进程发送调试事件，包括：创建新进程、新线程、加载DLL、执行断点等。调试器通过WaitForDebugEvent()和ContinueDebugEvent()构成事件循环。WaitForDebugEvent()可在指定的时间内等待可能的调试事件；ContinueDebugEvent()可使被调试事件暂停的进程继续运行。
- 通过ReadProcessMemory()和WriteProcessMemory()来读写被调试进程的存储空间。

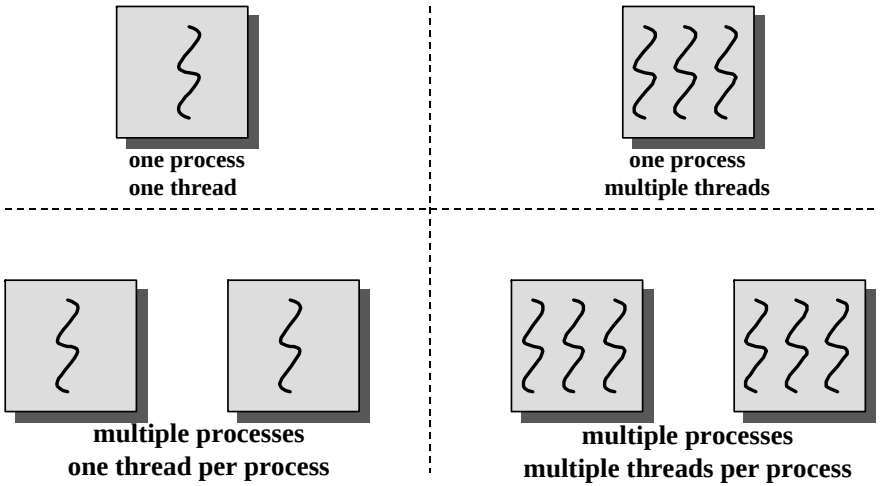
4.3 线程(THREAD)

引入线程的目的是简化线程间的通信，以小的开销来提高进程内的并发程度。

返回

4.3.1 线程的引入

- 进程：资源分配单位（存储器、文件）和CPU调度（分派）单位。又称为"任务(task)"
- 线程：作为CPU调度单位，而进程只作为其他资源分配单位。
  - 只拥有必不可少的资源，如：线程状态、寄存器上下文和栈
  - 同样具有就绪、阻塞和执行三种基本状态
- 线程的优点：减小并发执行的时间和空间开销（线程的创建、退出和调度），因此容许在系统中建立更多的线程来提高并发程度。
  - 线程的创建时间比进程短；
  - 线程的终止时间比进程短；
  - 同进程内的线程切换时间比进程短；
  - 由于同进程内线程间共享内存和文件资源，可直接进行不通过内核的通信；



进程与线程的关系

OS对线程的实现方式

内核线程(kernel-level thread)

依赖于OS核心，由内核的内部需求进行创建和撤销，用来执行一个指定的函数。Windows NT和OS/2支持内核线程；

- 内核维护进程和线程的上下文信息；
- 线程切换由内核完成；
- 一个线程发起系统调用而阻塞，不会影响其他线程的运行。
- 时间片分配给线程，所以多线程的进程获得更多CPU时间。

用户线程(user-level thread)

不依赖于OS核心，应用进程利用线程库提供创建、同步、调度和管理线程的函数来控制用户线程。如：数据库系统informix，图形处理Aldus PageMaker。调度由应用软件内部进行，通常采用非抢先式和更简单的规则，也无需用户态/核心态切换，所以速度特别快。一个线程发起系统调用而阻塞，则整个进程在等待。时间片分配给进程，多线程则每个线程就慢。

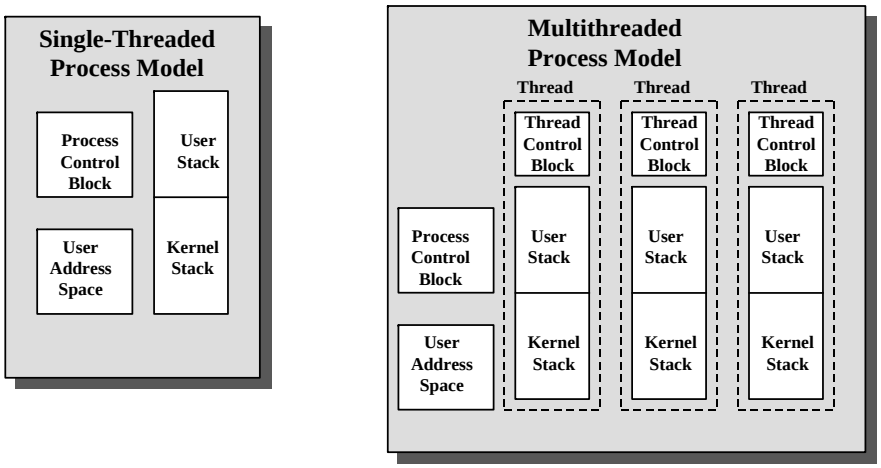
- 用户线程的维护由应用进程完成；
- 内核不了解用户线程的存在；
- 用户线程切换不需要内核特权；
- 用户线程调度算法可针对应用优化；

轻权进程(LightWeight Process)

它是内核支持的用户线程。一个进程可有一个或多个轻权进程，每个轻权进程由一个单独的内核线程来支持。

4.3.2 进程和线程的比较

- 地址空间和其他资源（如打开文件）：进程间相互独立，同一进程的各线程间共享——某进程内的线程在其他进程不可见
- 通信：进程间通信IPC，线程间可以直接读写进程数据段（如全局变量）来进行通信——需要进程同步和互斥手段的辅助，以保证数据的一致性
- 调度：线程上下文切换比进程上下文切换要快得多；

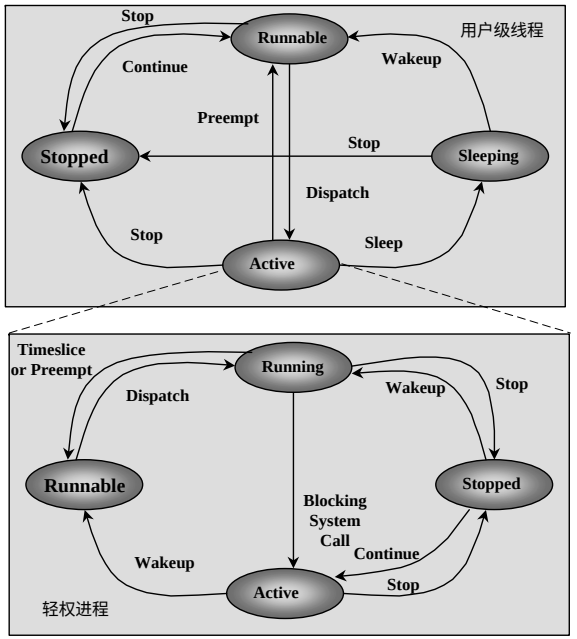


线程切换和进程切换

4.3.3 线程举例

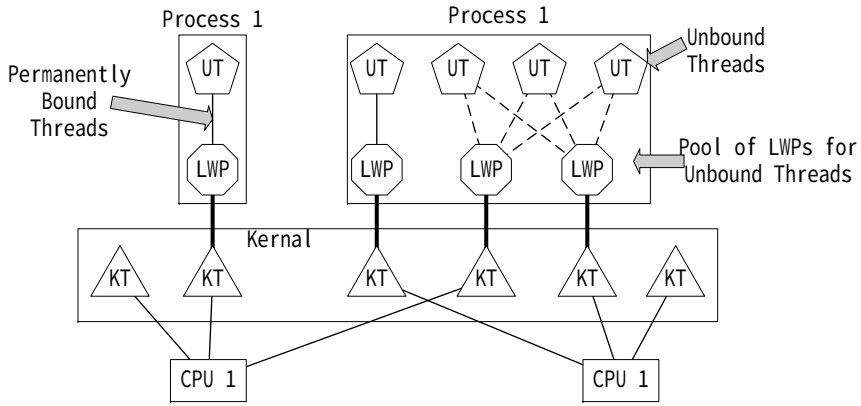
1. SUN Solaris 2.3

Solaris支持内核线程(Kernel threads)、轻权进程(Lightweight Processes)和用户线程(User Level Threads)。一个进程可有大量用户线程；大量用户线程复用少量的轻权进程，不同的轻权进程分别对应不同的内核线程。



Solaris用户线程和轻权进程

- 用户级线程在使用系统调用时（如文件读写），需要“捆绑(bound)”在一个LWP上。
  - 永久捆绑：一个LWP固定被一个用户级线程占用，该LWP移到LWP池之外
  - 临时捆绑：从LWP池中临时分配一个未被占用的LWP
- 在使用系统调用时，如果所有LWP已被其他用户级线程所占用（捆绑），则该线程阻塞直到有可用的LWP－－例如6个用户级线程，而LWP池中有4个LWP
- 如果LWP执行系统调用时阻塞（如read()调用），则当前捆绑在LWP上的用户级线程也阻塞。



用户线程、轻权进程和核心线程的关系

- 有关的C库函数

```
/* 创建用户级线程                                          */
 int thr_create(void *stack_base, size_t stack_size,
     void *(*start_routine)(void *), void *arg, long flags,
     thread_t *new_thread_id);
```

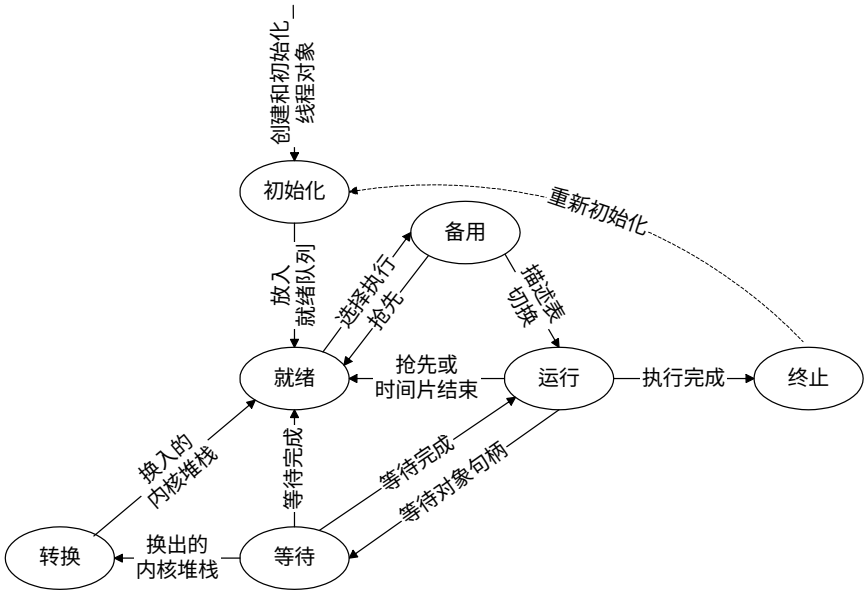
其中flags包括：THR_BOUND（永久捆绑）, THR_NEW_LWP（创建新LWP放入LWP池），若两者同时指定则创建两个新LWP，一个永久捆绑而另一个放入LWP池

- 有关的系统调用

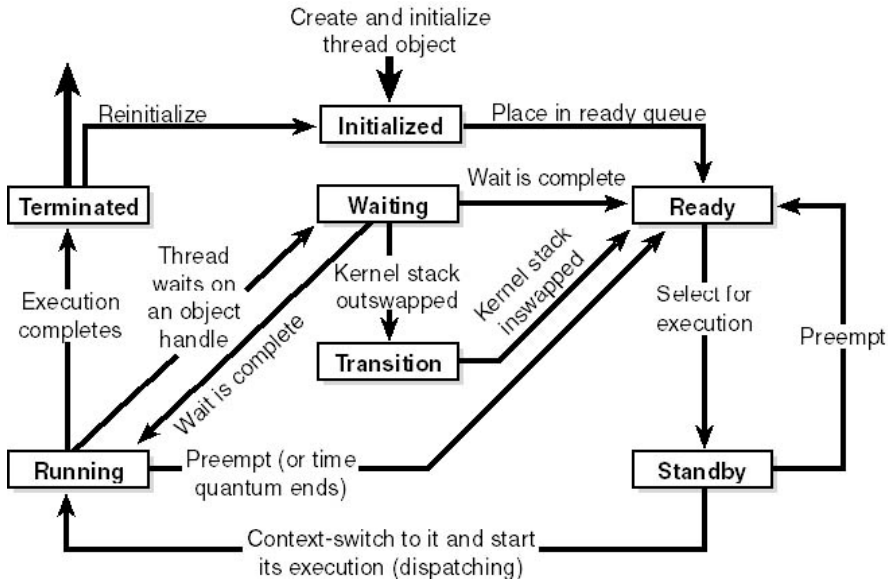
```
/* 在当前进程中创建LWP                          */
 int _lwp_create(ucontext_t *contextp, unsigned long flags,
     lwpid_t *new_lwp_id);
 /* 构造LWP上下文                                */
 void _lwp_makecontext(ucontext_t *ucp,
         void (*start_routine)( void *), void *arg,
         void *private, caddr_t stack_base, size_t stack_size);
 /* 注意：没有进行"捆绑"操作的系统调用 */
```

2. Windows NT

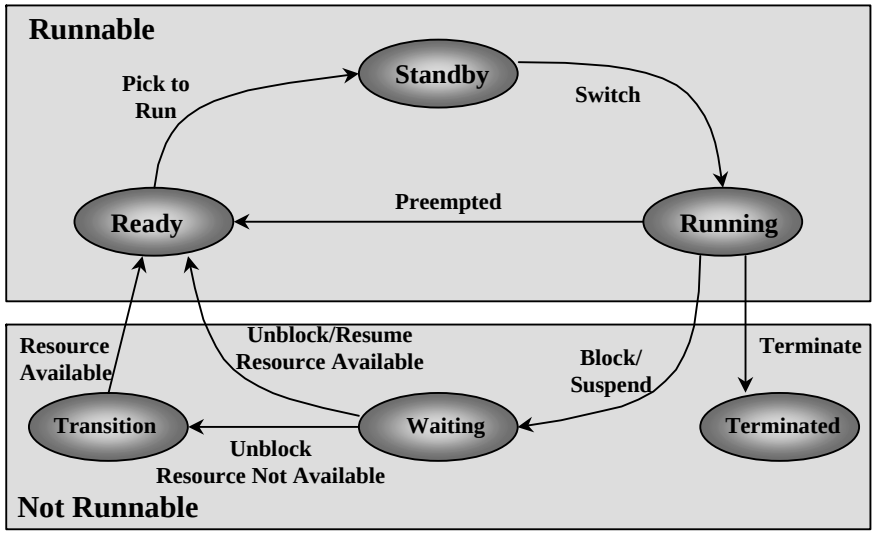
- 就绪状态(Ready)：进程已获得除处理机外的所需资源，等待执行。
- 备用状态(Standby)：特定处理器的执行对象，系统中每个处理器上只能有一个处于备用状态的线程。
- 运行状态(Running)：完成描述表切换，线程进入运行状态，直到内核抢先、时间片用完、线程终止或进行等待状态。
- 等待状态(Waiting)：线程等待对象句柄，以同步它的执行。等待结束时，根据优先级进入运行、就绪状态。
- 转换状态(Transition)：线程在准备执行而其内核堆栈处于外存时，线程进入转换状态；当其内核堆栈调回内存，线程进入就绪状态。
- 终止状态(Terminated)：线程执行完就进入终止状态；如执行体有一指向线程对象的指针，可将线程对象重新初始化，并再次使用。
- 初始化状态(Initialized)：线程创建过程中的线程状态；



Windows NT的线程状态



Windows 2000线程状态



Windows NT的线程状态

NT线程的有关API

- CreateThread()函数在调用进程的地址空间上创建一个线程，以执行指定的函数；返回值为所创建线程的句柄。
- ExitThread()函数用于结束本线程。
- SuspendThread()函数用于挂起指定的线程。
- ResumeThread()函数递减指定线程的挂起计数，挂起计数为0时，线程恢复执行。

4.4 进程互斥和同步

返回

4.4.1 互斥算法

4.4.1.1临界资源

多道程序环境－>进程之间存在资源共享，进程的运行时间受影响

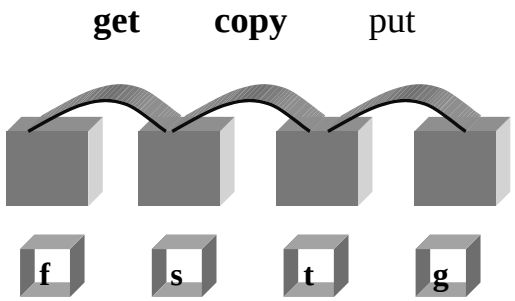
硬件或软件（如外设、共享代码段、共享数据结构），多个进程在对其进行访问时（关键是进行写入或修改），必须互斥地进行－－有些共享资源可以同时访问，如只读数据

- 进程间资源访问冲突
  - 共享变量的修改冲突
  - 操作顺序冲突
- 进程间的制约关系
  - 间接制约：进行竞争－－独占分配到的部分或全部共享资源，“互斥”
  - 直接制约：进行协作－－等待来自其他进程的信息，“同步”

一个飞机订票系统，两个终端，运行T1、T2进程

T1：	T2:
...	...
Read(x);	Read(x);
if x>=1 then	if x>=1 then
x:=x-1;	x:=x-1;
write(x);	write(x);
...	...

共享变量的修改冲突



有3个进程：get, copy和put，它们对4个存储区域f、s、t和g进行操作。

操作顺序冲突

	f	s	t	g	结果
初始状态	3,4,...,m	2	2	(1,2)	
g,c,p	4,5,...,m	3	3	(1,2,3)	正确
g,p,c	4,5,...,m	3	3	(1,2,2)	错误
c,g,p	4,5,...,m	3	2	(1,2,2)	错误
c,p,g	4,5,...,m	3	2	(1,2,2)	错误
p,c,g	4,5,...,m	3	2	(1,2,2)	错误
p,g,c	4,5,...,m	3	3	(1,2,2)	错误

有6种可能的操作顺序，只有一种结果是正确的。

进程的交互关系：可以按照相互感知的程度来分类

相互感知的程度	交互关系	一个进程对其他进程的影响	潜在的控制问题
相互不感知(完全不了解其它进程的存在)	竞争(competition)	一个进程的操作对其他进程的结果无影响	互斥，死锁（可释放的资源），饥饿
间接感知(双方都与第三方交互，如共享资源)	通过共享进行协作	一个进程的结果依赖于从其他进程获得的信息	互斥，死锁（可释放的资源），饥饿，数据一致性
直接感知(双方直接交互，如通信)	通过通信进行协作	一个进程的结果依赖于从其他进程获得的信息	死锁，饥饿

互斥，指多个进程不能同时使用同一个资源；
死锁，指多个进程互不相让，都得不到足够的资源；
饥饿，指一个进程一直得不到资源（其他进程可能轮流占用资源）

4.4.1.2临界区的访问过程

```
entry section
  critical section
exit section
  remainder section
```

临界区

- 临界区(critical section)：进程中访问临界资源的一段代码。
- 进入区(entry section)：在进入临界区之前，检查可否进入临界区的一段代码。如果可以进入临界区，通常设置相应"正在访问临界区"标志
- 退出区(exit section)：用于将"正在访问临界区"标志清除。
- 剩余区(remainder section)：代码中的其余部分。

4.4.1.3同步机制应遵循的准则

- 空闲则入：其他进程均不处于临界区；
- 忙则等待：已有进程处于其临界区；
- 有限等待：等待进入临界区的进程不能"死等"；
- 让权等待：不能进入临界区的进程，应释放CPU（如转换到阻塞状态）

4.4.1.4进程互斥的软件方法

算法1：单标志

- 有两个进程Pi, Pj，其中的Pi

```
while (turn != i);
  critical section
turn = j;
  remainder section
```

- 设立一个公用整型变量 turn：描述允许进入临界区的进程标识
  - 在进入区循环检查是否允许本进程进入：turn为i时，进程Pi可进入；
  - 在退出区修改允许进入进程标识：进程Pi退出时，改turn为进程Pj的标识j；

- 缺点：强制轮流进入临界区，没有考虑进程的实际需要。容易造成资源利用不充分：在Pi出让临界区之后，Pj使用临界区之前，Pi不可能再次使用临界区；

算法2：双标志、先检查

```
while (flag[j]);          <a>
flag[i] = TRUE;           <b>
  critical section
flag[i] = FALSE;
  remainder section
```

- 设立一个标志数组flag[]：描述进程是否在临界区，初值均为FALSE。
  - 先检查，后修改：在进入区检查另一个进程是否在临界区，不在时修改本进程在临界区的标志；
  - 在退出区修改本进程在临界区的标志；

- 优点：不用交替进入，可连续使用；
- 缺点：Pi和Pj可能同时进入临界区。按下面序列执行时，会同时进入："Pi<a> Pj<a> Pi<b> Pj<b>"。即在检查对方flag之后和切换自己flag之前有一段时间，结果都检查通过。这里的问题出在检查和修改操作不能连续进行。

算法3：双标志、后检查

```
flag[i] = TRUE;           <b>
while (flag[j]);          <a>
  critical section
flag[i] = FALSE;
  remainder section
```

- 类似于算法2，与互斥算法2的区别在于先修改后检查。可防止两个进程同时进入临界区。

- 缺点：Pi和Pj可能都进入不了临界区。按下面序列执行时，会都进不了临界区："Pi<a> Pj<a> Pi<b> Pj<b>"。即在切换自己flag之后和检查对方flag之前有一段时间，结果都切换flag，都检查不通过。

算法4(Peterson's Algorithm)：
先修改、后检查、后修改者等待

```
flag[i] = TRUE; turn = j;
while (flag[j] && turn == j);
  critical section
flag[i] = FALSE;
  remainder section
```

- 结合算法1和算法3，是正确的算法
- turn=j;描述可进入的进程（同时修改标志时）
- 在进入区先修改后检查，并检查并发修改的先后：
  - 检查对方flag，如果不在临界区则自己进入－－空闲则入
  - 否则再检查turn：保存的是较晚的一次赋值，则较晚的进程等待，较早的进程进入－－先到先入，后到等待

4.4.1.5进程互斥的硬件方法

- 完全利用软件方法，有很大局限性（如不适于多进程），现在已很少采用。
- 可以利用某些硬件指令－－其读写操作由一条指令完成，因而保证读操作与写操作不被打断；

Test-and-Set指令

该指令读出标志后设置为TRUE

```
boolean TS(boolean *lock) {
  boolean old;
  old = *lock;  *lock = TRUE;
  return old;
}
```

lock表示资源的两种状态：TRUE表示正被占用，FALSE表示空闲

互斥算法（TS指令）

```
while TS(&lock);
  critical section
lock = FALSE;
  remainder section
```

- 利用TS实现进程互斥：每个临界资源设置一个公共布尔变量lock，初值为FALSE
- 在进入区利用TS进行检查：有进程在临界区时，重复检查；直到其它进程退出时，检查通过；

Swap指令（或Exchange指令）

交换两个字（字节）的内容

```
void SWAP(int *a, int *b) {
  int temp;
  temp = *a;  *a = *b;  *b = temp;
}
```

互斥算法（Swap指令）

- 利用Swap实现进程互斥：每个临界资源设置一个公共布尔变量lock，初值为FALSE。每个进程设置一个私有布尔变量key

```
key = TRUE;
do
{
  SWAP(&lock, &key);
} while (key);
  critical section
lock = FALSE;
  remainder section
```

- 硬件方法的优点
  - 适用于任意数目的进程，在单处理器或多处理器上
  - 简单，容易验证其正确性
  - 可以支持进程内存在多个临界区，只需为每个临界区设立一个布尔变量
- 硬件方法的缺点
  - 等待要耗费CPU时间，不能实现"让权等待"
  - 可能"饥饿"：从等待进程中随机选择一个进入临界区，有的进程可能一直选不上
  - 可能死锁

4.4.2 信号量(semaphore)

前面的互斥算法都存在问题，它们是平等进程间的一种协商机制，需要一个地位高于进程的管理者来解决公有资源的使用问题。OS可从进程管理者的角度来处理互斥的问题，信号量就是OS提供的管理公有资源的有效手段。

信号量代表可用资源实体的数量。

4.4.2.1 信号量和P、V原语

4.4.2.2 信号量集

4.4.2.1 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。
- 每个信号量s除一个整数值s.count（计数）外，还有一个进程等待队列s.queue，其中是阻塞在该信号量的各个进程的标识
  - 信号量只能通过初始化和两个标准的原语来访问－－作为OS核心代码执行，不受进程调度的打断
  - 初始化指定一个非负整数值，表示空闲资源总数（又称为"资源信号量"）－－若为非负值表示当前的空闲资源数，若为负值其绝对值表示当前等待临界区的进程数
- "二进制信号量(binary semaphore)"：只允许信号量取0或1值

1. P原语wait(s)

```
--s.count;          //表示申请一个资源;
if (s.count <0)    //表示没有空闲资源;
{
   调用进程进入等待队列s.queue;
   阻塞调用进程;
}
```

2. V原语signal(s)

V原语通常唤醒进程等待队列中的头一个进程

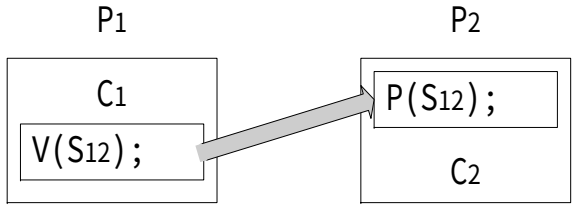
```
++s.count;          //表示释放一个资源；
if (s.count <= 0)//表示有进程处于阻塞状
  态；
{
   从等待队列s.queue中取出一个进程P;
   进程P进入就绪队列;
}
```

3. 利用信号量实现互斥

```
P(mutex);
  critical section
V(mutex);
  remainder section
```

- 为临界资源设置一个互斥信号量mutex(MUTual Exclusion)，其初值为1；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

4. 利用信号量来描述前趋关系



- 前趋关系：并发执行的进程P1和P2中，分别有代码C1和C2，要求C1在C2开始前完成；
- 为每个前趋关系设置一个互斥信号量S12，其初值为0

4.4.2.2 信号量集

信号量集用于同时需要多个资源时的信号量操作；

1. AND型信号量集

AND型信号量集用于同时需要多种资源且每种占用一个时的信号量操作；

- 一段处理代码需要同时获取两个或多个临界资源——可能死锁：各进程分别获得部分临界资源，然后等待其余的临界资源，"各不相让"
- 基本思想：在一个原语中，将一段代码同时需要的多个临界资源，要么全部分配给它，要么一个都不分配。称为Swait(Simultaneous Wait)。在Swait时，各个信号量的次序并不重要，虽然会影响进程归入哪个阻塞队列，但是由于是对资源全部分配或不分配，所以总有进程获得全部资源并在推进之后释放资源，因此不会死锁。

```
Swait(S1, S2, …, Sn) //P原语；
{
while (TRUE)
{
    if (S1 >=1 && S2 >= 1 && … && Sn >= 1)
 {                  //满足资源要求时的处理；
      for (i = 1; i <= n; ++i) --Si;
        //注：与wait的处理不同，这里是在确信可满足
        //资源要求时，才进行减1操作；
      break;
    }
 else
 {                  //某些资源不够时的处理；
     调用进程进入第一个小于1信号量的等待队列Sj.queue;
     阻塞调用进程;
    }
}
}
```

```
Ssignal(S1, S2, …, Sn)
{
  for (i = 1; i <= n; ++i)
  {
    ++Si;                    //释放占用的资源；
    for (each process P waiting in Si.queue)
         //检查每种资源的等待队列的所有进程；
    {
      从等待队列Si.queue中取出进程P;
      if (判断进程P是否通过Swait中的测试)
          //注：与signal不同，这里要进行重新判断；
          {         //通过检查（资源够用）时的处理；
          进程P进入就绪队列;
          }
      else
          {         //未通过检查（资源不够用）时的处理；
          进程P进入某等待队列；
          }
    }
  }
}
```

2. 一般“信号量集”

一般信号量集用于同时需要多种资源、每种占用的数目不同、且可分配的资源还存在一个临界值时的处理；

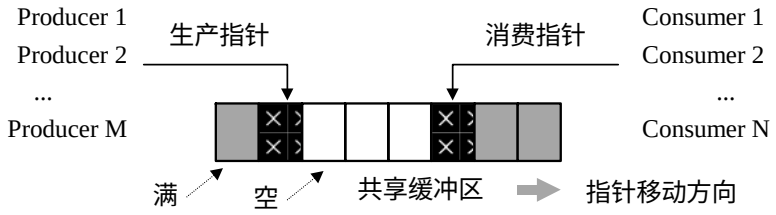
- 一次需要N个某类临界资源时，就要进行N次wait操作——低效又可能死锁
- 基本思想：在AND型信号量集的基础上进行扩充：进程对信号量Si的测试值为ti（用于信号量的判断，即Si >= ti，表示资源数量低于ti时，便不予分配），占用值为di（用于信号量的增减，即Si = Si - di和Si = Si + di）
- Swait(S1, t1, d1; ...; Sn, tn, dn);
- Ssignal(S1, d1; ...; Sn, dn);

- 一般"信号量集"的几种特定情况：
  - Swait(S, d, d)表示每次申请d个资源，当少于d个时，便不分配；
  - Swait(S, 1, 1)表示互斥信号量；
  - Swait(S, 1, 0)作为一个可控开关
    - 当S>=1时，允许多个进程进入临界区；
    - 当S=0时，禁止任何进程进入临界区；
- 一般"信号量集"未必成对使用Swait和Ssignal：如：一起申请，但不一起释放；

4.4.3 经典进程同步问题

1. 生产者－消费者问题(the producer-consumer problem)

问题描述：若干进程通过有限的共享缓冲区交换数据。其中，"生产者"进程不断写入，而"消费者"进程不断读出；共享缓冲区共有N个；任何时刻只能有一个进程可对共享缓冲区进行操作。



- 采用信号量机制：
  - full是"满"数目，初值为0，empty是"空"数目，初值为N。实际上，full和empty是同一个含义：full + empty == N
  - mutex用于访问缓冲区时的互斥，初值是1
- 每个进程中各个P操作的次序是重要的：先检查资源数目，再检查是否互斥——否则可能死锁(为什么？)
- 采用AND信号量集：Swait(empty, mutex), Ssignal(full, mutex), …

Producer

```
P(empty);
P(mutex);    //进入区
  one unit --> buffer;
V(mutex);
V(full);     //退出区
```

Consumer

```
P(full);
P(mutex);    //进入区
  one unit <-- buffer;
V(mutex);
V(empty);    //退出区
```

2. 读者－写者问题(the readers-writers problem)

- 问题描述：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

- 采用信号量机制：
  - Wmutex表示"允许写"，初值是1。
  - 公共变量Rcount表示“正在读”的进程数，初值是0；
  - Rmutex表示对Rcount的互斥操作，初值是1。

Writer

```
P(Wmutex);
  write;
V(Wmutex);
```

Reader

```
P(Rmutex);
  if (Rcount == 0)
P(Wmutex);
  ++Rcount;
V(Rmutex);
  read;
P(Rmutex);
  --Rcount;
  if (Rcount == 0)
V(Wmutex);
V(Rmutex);
```

- 采用一般"信号量集"机制：问题增加一个限制条件：同时读的"读者"最多R个
  - Wmutex表示"允许写"，初值是1
  - Rcount表示"允许读者数目"，初值为R

Writer

```
Swait(Wmutex, 1, 1; Rcount, R, 0);
  write;
Ssignal(Wmutex, 1);
```

Reader

```
Swait(Rcount, 1, 1; Wmutex, 1, 0);
  read;
Ssignal(Rcount, 1);
```

3. 哲学家进餐问题 (the dining philosophers problem)

- 问题描述：（由Dijkstra首先提出并解决）5个哲学家围绕一张圆桌而坐，桌子上放着5支筷子，每两个哲学家之间放一支；哲学家的动作包括思考和进餐，进餐时需要同时拿起他左边和右边的两支筷子，思考时则同时将两支筷子放回原处。如何保证哲学家们的动作有序进行？如：不出现相邻者同时要求进餐；不出现有人永远拿不到筷子；

4.4.4 管程(monitor)

用信号量可实现进程间的同步，但由于信号量的控制分布在整个程序中，其正确性分析很困难。管程是管理进程间同步的机制，它保证进程互斥地访问共享变量，并方便地阻塞和唤醒进程。管程可以函数库的形式实现。相比之下，管程比信号量好控制。

1. 信号量同步的缺点

- 同步操作分散：信号量机制中，同步操作分散在各个进程中，使用不当就可能导致各进程死锁（如P、V操作的次序错误、重复或遗漏）
- 易读性差：要了解对于一组共享变量及信号量的操作是否正确，必须通读整个系统或者并发程序；
- 不利于修改和维护：各模块的独立性差，任一组变量或一段代码的修改都可能影响全局；
- 正确性难以保证：操作系统或并发程序通常很大，很难保证这样一个复杂的系统没有逻辑错误；

2. 管程的引入

- 1973年，Hoare和Hanson所提出；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。
- 管程的定义：管程是关于共享资源的数据结构及一组针对该资源的操作过程所构成的软件模块。
- 管程可增强模块的独立性：系统按资源管理的观点分解成若干模块，用数据表示抽象系统资源，同时分析了共享资源和专用资源在管理上的差别，按不同的管理方式定义模块的类型和结构，使同步操作相对集中，从而增加了模块的相对独立性
- 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：采用集中式同步机制。一个操作系统或并发程序由若干个这样的模块所构成，一个模块通常较短，模块之间关系清晰。

3. 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，管程中的外部过程（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；

4. 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外部过程（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；

5. 管程中的多个进程进入

- 当一个进入管程的进程执行等待操作时，它应当释放管程的互斥权；当一个进入管程的进程执行唤醒操作时（如Ｐ唤醒Ｑ），管程中便存在两个同时处于活动状态的进程。
- 管程中的唤醒切换方法：
  - P等待Q继续，直到Q等待或退出；
  - Q等待P继续，直到P等待或退出；
  - 规定唤醒为管程中最后一个可执行的操作；

- 入口等待队列：因为管程是互斥进入的，所以当一个进程试图进入一个已被占用的管程时它应当在管程的入口处等待，因而在管程的入口处应当有一个进程等待队列，称作入口等待队列。
- 紧急等待队列：如果进程Ｐ唤醒进程Ｑ，则Ｐ等待Ｑ继续，如果进程Ｑ在执行又唤醒进程Ｒ，则Ｑ等待Ｒ继续，...，如此，在管程内部，由于执行唤醒操作，可能会出现多个等待进程（已被唤醒，但由于管程的互斥进入而等待），因而还需要有一个进程等待队列，这个等待队列被称为紧急等待队列。它的优先级应当高于入口等待队列的优先级。

6. 条件变量(condition)

- 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，并不取具体数值－－相当于每个原因对应一个队列。

- 同步操作原语cwait和csignal：针对条件变量c，cwait(c)将自己阻塞在c队列中，csignal(c)将c队列中的一个进程唤醒。
  - cwait（c）：如果紧急等待队列非空，则唤醒第一个等待者；否则释放管程的互斥权，执行此操作的进程排入c队列尾部（紧急等待队列与c队列的关系：紧急等待队列是由于管程的互斥进入而等待的队列，而c队列是因资源被占用而等待的队列）
  - csignal（c）：如果c队列为空，则相当于空操作，执行此操作的进程继续；否则唤醒第一个等待者，执行此操作的进程排入紧急等待队列的尾部
- 若进程P唤醒进程Q，则随后可有两种执行方式（进程P、Q都是管程中的进程）
  - P等待，直到执行Q离开管程或下一次等待。Hoare采用。
  - Q送入Ready队列，直到执行P离开管程或下一次等待。1980年，Lampson和Redell采用。原语由csignal(x)改为cnotify(x)。

7. 管程的格式

```
TYPE monitor_name = MONITOR;
共享变量说明
define 本管程内所定义、本管程外可调用的过程（函数）名字表
use    本管程外所定义、本管程内将调用的过程（函数）名字表
PROCEDURE 过程名（形参表）；
                过程局部变量说明；
                BEGIN
                        语句序列；
                END;
.....
```

```
FUNCTION 函数名（形参表）：值类型；
        函数局部变量说明；
                BEGIN
                        语句序列；
                END;
.....
BEGIN
        共享变量初始化语句序列；
END;
```

8. 管程的的组成

- 名称：为每个共享资源设立一个管程
- 数据结构说明：一组局部于管程的控制变量
- 操作原语：对控制变量和临界资源进行操作的一组原语过程（程序代码），是访问该管程的唯一途径。这些原语本身是互斥的，任一时刻只允许一个进程去调用，其余需要访问的进程就等待。
- 初始化代码：对控制变量进行初始化的代码

```
TYPE one_instance=RECORD
   mutex:semaphore;（入口互斥队列类型，初值1）
   urgent:semaphore;（紧急等待队列类型，初值0）
   urgent_count:integer;（紧急等待队列计数类型，初值0）
            END;
TYPE
 monitor_elements=MODULE;
define enter,leave,wait,signal;

mutex（入口互斥队列）
   urgent（紧急等待队列）
   urgent_count（紧急等待计数）
```

```
PROCEDURE enter(VAR instance:one_instance);
BEGIN
  P(instance.mutex)
END;

PROCEDURE leave(VAR instance:one_instance);
BEGIN
  IF instance.urgent_count >0 THEN
  BEGIN   instance.urgent--;
    V(instance.urgent)
  END
  ELSE
    V(instance.mutex)
END;
```

```
PROCEDURE wait(VAR instance:one_instance;
                    VAR s:semephore;VAR count:integer);
BEGIN
  count++;
  IF instance.urgent_count>0 THEN
  BEGIN
    instance.urgent_count--;
    V(instance.urgent)
  END
  ELSE
    V(instance. mutex);
  P(s);
END;
```

```
PROCEDURE signal(VAR instance:one_instance;
                        VAR s:semaphore;VAR count:integer);
BEGIN
  IF count>0 THEN
  BEGIN
    count--;
    instance.urgent_count++;
    V(s);
    P(instance.urgent)
  END
END;
```

9. 例子：读者-写者问题

```
TYPE r_and_w=MODULE;
VAR instance:one_instance;
rq,wq:semaphore;                  //读写等待队列
r_count,w_count:integer;          //读写等待计数
reading_count,write_count:integer;   //正在读写进程计数
define
 start_r,finish_r,start_w,finish_w;
use monitor_elements.enter,
     monitor_elements.leave,
    monitor_elements.wait,
    monitor_elements.signal;
```

```
PROCEDURE start_r;
BEGIN
    monitor_elements.enter(instance);
    IF write_count>0 THEN
        monitor_elements.wait
                (instance,rq,r_count);
    reading_count++;
    monitor_elements.signal
            (instance,rq,r_count);
    monitor_elements.leave(instance);
END;
```

```
PROCEDURE finish_r;
BEGIN
    monitor_elements.enter(instance);
    reading_count--;
    IF reading_count=0 THEN
       monitor_elements.signal
                (instance,wq,w_count);
    monitor_elements.leave(instance);
END;
```

```
PROCEDURE start_w;
BEGIN
    monitor_elements.enter(instance);
    write_count++;
    IF (write_count>1)
       OR(reading_count>0) THEN   monitor_elements.wait
           (instance,wq,w_count);
 monitor_elements.leave(instance);
END;
```

```
BEGIN
     reading_count::=0;
     write_count::=0;
     r_count:=0;
     w_count::=0;
END;
```

读者的活动：
r_and_w.start_r;
读操作；
r_and_w.finish_r;

写者的活动：
r_and_w.start_w;
写操作；
r_and_w.finish_w;

10. 管程和进程的异同点

- 设置进程和管程的目的不同
- 系统管理数据结构
  - 进程：PCB
  - 管程：等待队列
- 管程被进程调用
- 管程是操作系统的固有成分，无创建和撤消

4.4.5 进程互斥和同步举例

4.4.5.1 Solaris 2.3

4.4.5.2 Windows NT

4.4.5.1 Solaris 2.3

- 支持信号量和信号量集，通过semaphore ID来标识
- 有关的系统调用：
  - 获取信号量集semget（依据用户给出的整数值key，创建新信号量或打开现有信号量，返回一个信号量ID）
  - 信号量控制操作semctl
  - semop（对信号量的原子操作）

4.4.5.2 Windows NT

对象可用(signaled state)表示该对象不被任何线程使用或所有；

1. NT支持的三种同步对象

对象名称是由用户给出的字符串。不同进程中用同样的名称来创建或打开对象，从而获得该对象在本进程的句柄。

- Mutex对象：互斥对象，相当于互斥信号量，在一个时刻只能被一个线程使用。有关的API：
  - CreateMutex创建一个互斥对象，返回对象句柄；
  - OpenMutex返回一个已存在的互斥对象的句柄，用于后续访问；
  - ReleaseMutex释放对互斥对象的占用，使之成为可用；

- Semaphore对象：相当于资源信号量，取值在0到指定最大值之间，用于限制并发访问的线程数。有关的API：
  - CreateSemaphore创建一个信号量对象，指定最大值和初值，返回对象句柄；
  - OpenSemaphore返回一个已存在的信号量对象的句柄，用于后续访问；
  - ReleaseSemaphore释放对信号量对象的占用；

- Event对象：事件对象，相当于"触发器"，可通知一个或多个线程某事件的出现。有关的API：
  - CreateEvent创建一个事件对象，返回对象句柄；
  - OpenEvent返回一个已存在的事件对象的句柄，用于后续访问；
  - SetEvent和PulseEvent设置指定事件对象为可用状态；
  - ResetEvent设置指定事件对象为不可用状态（nonsignaled）；手工复位，并唤醒所有等待线程；

2. 同步对象等待

(1) WaitForSingleObject在指定的时间内等待指定对象为可用状态(signaled state)；

```
DWORD WaitForSingleObject( HANDLE hHandle,
// handle of object to wait for
DWORD dwMilliseconds      // time-out interval in milliseconds
);
```

(2) WaitForMultipleObjects在指定的时间内等待多个对象为可用状态；

```
DWORD WaitForMultipleObjects( DWORD nCount,
//对象句柄数组中的句柄数；
CONST HANDLE *lpHandles,
// 指向对象句柄数组的指针，数组中可包括多种对象句柄；
BOOL bWaitAll,
// 等待标志：TRUE表示所有对象同时可用，FALSE表示至少一个对象可用；
DWORD dwMilliseconds // 等待超时时限；
);
```

3. 子进程对同步对象的继承

对象在创建时指定可否被子进程继承，另外还要把对象的句柄通过命令行参数传递给子进程（才能引用该对象）。DuplicateHandle可以将对象句柄复制给指定的另一个进程；

```
BOOL DuplicateHandle( HANDLE hSourceProcessHandle,
        //被复制对象句柄所在进程的进程对象句柄；
        HANDLE hSourceHandle,//被复制对象句柄；
        HANDLE hTargetProcessHandle,
        //复制后对象句柄所在进程的进程对象句柄；
        LPHANDLE lpTargetHandle,        //指向复制后对象句柄的指针；
        DWORD dwDesiredAccess,          //复制后对象句柄的访问类型，不同类型对象的访问类型会不同；
        BOOL bInheritHandle,//复制后对象句柄在子进程中的继承方式；
        DWORD dwOptions         //选项；
);
```

4. 其他同步方法

- Critical Section对象：只能在同一进程内使用的临界区，同一进程内各线程对它的访问是互斥进行的。把变量说明为CRITICAL_SECTION类型，就可作为临界区使用。有关的API：
  - InitializeCriticalSection对临界区对象进行初始化；
  - EnterCriticalSection等待占用临界区的使用权，得到使用权时返回；
  - TryEnterCriticalSection非等待方式申请临界区的使用权；申请失败时，返回0；
  - LeaveCriticalSection释放临界区的使用权；
  - DeleteCriticalSection释放与临界区对象相关的所有系统资源；

- 互锁变量访问：相当于硬件指令，对一个整数（进程内的变量或进程间的共享变量）进行操作。其目的是避免线程间切换的影响。有关的API：
  - InterlockedExchange进行32位数据的先读后写原子操作；
  - InterlockedCompareExchange依据比较结果进行赋值的原子操作；
  - InterlockedExchangeAdd先加后存结果的原子操作；
  - InterlockedDecrement先减1后存结果的原子操作；
  - InterlockedIncrement先加1后存结果的原子操作；

作业

设有5个哲学家，共享一张放有5把椅子的桌子，每人分得1把椅子。但是，桌子上总共只有5支筷子，在每人两边分开各放一支。哲学家们在肚子饥饿时才试图分两次从两边拾起筷子就餐。
条件：
1）只有拿到两支筷子时，哲学家才能吃饭。
2）如果筷子已在他人手上，则该哲学家必须等待到他人吃完之后才能拿到筷子。
3）任劳任怨哲学家在自己未拿到两支筷子吃饭之前，决不放下自己手中的筷子。
要求：
1）有什么情况下5个哲学家全部吃不上饭？
2）描述一种没有人饿死（永远拿不到筷子）算法。

实验二

用信号灯来实现读者-写者问题

4.5 进程间通信
(IPC, INTER-PROCESS COMMUNICATION)

返回

4.5.0 进程间通信的类型

1. 低级通信和高级通信

- 低级通信：只能传递状态和整数值（控制信息），包括进程互斥和同步所采用的信号量和管程机制。优点的速度快。缺点是：
  - 传送信息量小：效率低，每次通信传递的信息量固定，若传递较多信息则需要进行多次通信。
  - 编程复杂：用户直接实现通信的细节，编程复杂，容易出错。
- 高级通信：能够传送任意数量的数据，包括三类：共享存储区、管道、消息。

返回

2. 直接通信和间接通信

- 直接通信：信息直接传递给接收方，如管道。
  - 在发送时，指定接收方的地址或标识，也可以指定多个接收方或广播式地址；
  - 在接收时，允许接收来自任意发送方的消息，并在读出消息的同时获取发送方的地址。
- 间接通信：借助于收发双方进程之外的共享数据结构作为通信中转，如消息队列。通常收方和发方的数目可以是任意的。

3. 高级通信的特征

- 通信链路(communication link)：
  - 点对点/多点/广播
  - 单向/双向
  - 有容量（链路带缓冲区）/无容量（发送方和接收方需自备缓冲区）
- 数据格式：
  - 字节流(byte stream)：各次发送之间的分界，在接收时不被保留，没有格式；
  - 报文(datagram/message)：各次发送之间的分界，在接收时被保留，通常有格式（如表示类型），定长/不定长报文，可靠报文/不可靠报文。
- 收发操作的同步方式
  - 发送阻塞（直到被链路容量或接收方所接受）和不阻塞（失败时立即返回）
  - 接收阻塞（直到有数据可读）和不阻塞（无数据时立即返回）
  - 由事件驱动收发：在允许发送或有数据可读时，才做发送和接收操作

4.5.1 信号(signal)

信号相当于给进程的“软件”中断；进程可发送信号，指定信号处理例程；它是单向和异步的。

返回

4.5.1.1 UNIX信号

1. 信号类型

- 一个进程向另一个进程或进程组（或自己）发送（kill系统调用）：发送者必须具有接收者同样的有效用户ID，或者发送者是超级用户身份
- 某些键盘按键，如：中断字符（通常是Ctrl+C或Del）、暂停字符（如Ctrl+Z）
- 硬件条件，如：除数为零、浮点运算错、访问非法地址等异常条件
- 软件条件，如：Socket中有加急数据到达

2. 对信号的处理

- 进程可以设置信号处理例程（signal系统调用），在接收到信号时就被调用，称为"捕获"该信号。信号处理例程的参数是接收到信号的编号。
- 进程也可以忽略指定的信号(SIG_IGN)。
  - 只有SIGKILL信号（无条件终止进程）和SIGSTOP（使进程暂停）不能被忽略。
  - 在库函数system()的实现中，通过fork和exec加载新程序之后，在父进程中对SIGINT和SIGQUIT都要忽略，然后wait直到子进程终止，才恢复对SIGINT和SIGQUIT的原有处理例程。
- 进程创建后为信号设立了默认处理例程(SIG_DFL)，如：终止并留映象文件(core)

4.5.1.2 Windows NT信号

NT的信号处理有两组系统调用，分别用于不同的信号；

1. SetConsoleCtrlHandler和GenerateConsoleCtrlEvent

- SetConsoleCtrlHandler在本进程的处理例程(HandlerRoutine)列表中定义或取销用户定义的处理例程；如：缺省时，它有一个CTRL+C输入的处理例程，我们可利用本调用来忽视或恢复CTRL+C输入的处理；
- GenerateConsoleCtrlEvent发送信号到与本进程共享同一控制台的控制台进程组；

处理信号列表(5种)

CTRL_C_EVENT	A CTRL+C signal was received.
CTRL_BREAK_EVENT	A CTRL+BREAK signal was received.
CTRL_CLOSE_EVENT	A signal that the system sends to all processes attached to a console when the user closes the console.
CTRL_LOGOFF_EVENT	A signal that the system sends to all console processes when a user is logging off. This signal does not indicate which user is logging off.
CTRL_SHUTDOWN_EVENT	A signal that the system sends to all console processes when the system is shutting down.

2. signal和raise

- signal设置中断信号处理例程；如：SIGINT（CTRL+C）、SIGABRT异常中止等信号的处理；
- raise给本进程发送一个信号；
- 处理信号列表（6种）

SIGABRT	Abnormal termination
SIGFPE	Floating-point error
SIGILL	Illegal instruction
SIGINT	CTRL+C signal (对 Win32 无效)
SIGSEGV	Illegal storage access
SIGTERM	Termination request

4.5.2 共享存储区(shared memory)

相当于内存，可以任意读写和使用任意数据结构（当然，对指针要注意），需要进程互斥和同步的辅助来确保数据一致性

1. UNIX的共享存储区

- 创建或打开共享存储区(shmget)：依据用户给出的整数值key，创建新区或打开现有区，返回一个共享存储区ID。
- 连接共享存储区(shmat)：连接共享存储区到本进程的地址空间，可以指定虚拟地址或由系统分配，返回共享存储区首地址。父进程已连接的共享存储区可被fork创建的子进程继承。
- 拆除共享存储区连接(shmdt)：拆除共享存储区与本进程地址空间的连接。
- 共享存储区控制(shmctl)：对共享存储区进行控制。如：共享存储区的删除需要显式调用shmctl(shmid, IPC_RMID, 0)；

返回

2. UNIX的文件映射

mmap：建立进程地址空间到文件或共享存储区对象的映射，将文件偏移off起的len字节映射到进程地址addr处：

```
caddr_t mmap(caddr_t addr, size_t len, int prot, int flags,
             int fildes, off_t off);
```

munmap：拆除映射

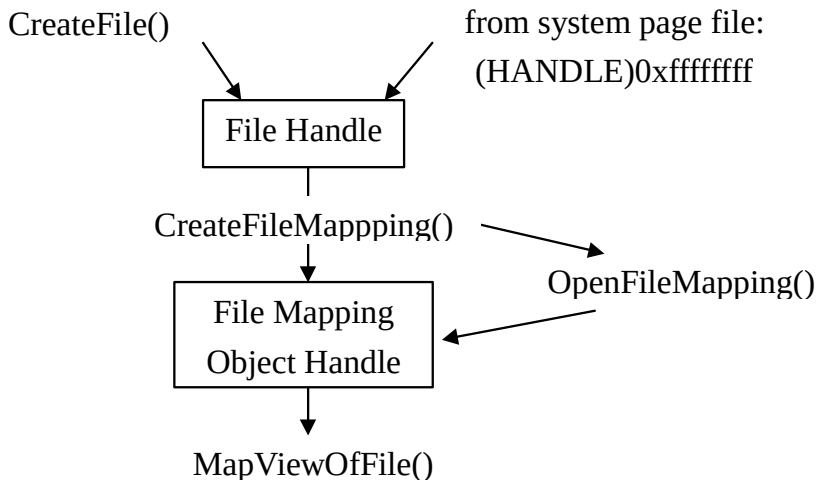
```
int munmap(void *addr, size_t len);
```

优点：

提高效率（消除不必要的数据拷贝）、
降低复杂性（直接读写操作而不必管理缓冲区）、
可直接将内存的数据类型记录在文件上（如指针）

3. Windows NT的文件映射

- 采用文件映射(file mapping)机制：可以将整个文件映射为进程虚拟地址空间的一部分来加以访问。在CreateFileMapping和OpenFileMapping时可以指定对象名称。
  - CreateFileMapping为指定文件创建一个文件映射对象，返回对象指针；
  - OpenFileMapping打开一个命名的文件映射对象，返回对象指针；
  - MapViewOfFile把文件映射到本进程的地址空间，返回映射地址空间的首地址；
- 这时可利用首地址进行读写；
  - FlushViewOfFile可把映射地址空间的内容写到物理文件中；
  - UnmapViewOfFile拆除文件映射与本进程地址空间间映射关系；
- 随后，可利用CloseHandle关闭文件映射对象；



4.5.3 管道(pipe)

管道是一条在进程间以字节流方式传送的通信通道。它由OS核心的缓冲区（通常几十KB）来实现，是单向的；常用于命令行所指定的输入输出重定向和管道命令。在使用管道前要建立相应的管道，然后才可使用。

返回

1. UNIX管道

- 通过pipe系统调用创建无名管道，得到两个文件描述符，分别用于写和读。
  - int pipe(int fildes[2]);
  - 文件描述符fildes[0]为读端，fildes[1]为写端；
  - 通过系统调用write和read进行管道的写和读；
  - 进程间双向通信，通常需要两个管道；
  - 只适用于父子进程之间或父进程安排的各个子进程之间；
- UNIX中的命名管道，可通过mknod系统调用建立：指定mode为S_IFIFO
  - int mknod(const char *path, mode_t mode, dev_t dev);

2. Windows NT管道

无名管道：类似于UNIX管道，CreatePipe可创建无名管道，得到两个读写句柄；利用ReadFile和WriteFile可进行无名管道的读写；

```
BOOL CreatePipe( PHANDLE hReadPipe,
          // address of variable for read handle
         PHANDLE hWritePipe,
          // address of variable for write handle
         LPSECURITY_ATTRIBUTES lpPipeAttributes,
          // pointer to security attributes
         DWORD nSize          // number of bytes reserved for pipe
);
```

- 命名管道：一个服务器端与一个客户进程间的通信通道；可用于不同机器上进程通信；
  - 类型分为：字节流，消息流（报文）；
  - 访问分为：单向读，单向写，双向；
  - 还有关于操作阻塞(wait/nowait)的设置。相应有一个文件句柄
  - 通常采用client-server模式，连接本机或网络中的两个进程
  - 管道名字：作为客户方（连接到一个命名管道实例的一方）时，可以是"\\serverName\pipe\pipename"；作为服务器方(创建命名管道的一方)时，只能取serverName为\\.\pipe\PipeName，不能在其它机器上创建管道；

- 命名管道服务器支持多客户：为每个管道实例建立单独线程或进程。
  - CreateNamedPipe在服务器端创建并返回一个命名管道句柄；
  - ConnectNamedPipe在服务器端等待客户进程的请求；
  - CallNamedPipe从管道客户进程建立与服务器的管道连接；
  - ReadFile、WriteFile（用于阻塞方式）、ReadFileEx、WriteFileEx（用于非阻塞方式）用于命名管道的读写；

4.5.4 消息(message)

与窗口系统中的“消息”不同。通常是不定长数据块。消息的发送不需要接收方准备好，随时可发送。

返回

1. UNIX消息

- 消息队列(message queue)：每个message不定长，由类型(type)和正文(text)组成
- UNIX消息队列API：
  - msgget依据用户给出的整数值key，创建新消息队列或打开现有消息队列，返回一个消息队列ID；
  - msgsnd发送消息；
  - msgrcv接收消息，可以指定消息类型；没有消息时，返回-1；
  - msgctl对消息队列进行控制，如删除消息队列；
- 通过指定多种消息类型，可以在一个消息队列中建立多个虚拟信道
- 注意：消息队列不随创建它的进程的终止而自动撤销，必须用msgctl(msgqid, IPC_RMID, 0)。另外，msgget获得消息队列ID之后，fork创建子进程，在子进程中能否继承该消息队列ID而不必再一次msgget。

2. Windows NT邮件槽

- 邮件槽(mailslot)：驻留在OS核心，不定长数据块（报文），不可靠传递
- 通常采用client-server模式：单向的，从client发往server；server负责创建邮件槽，它可从邮件槽中读消息；client可利用邮件槽的名字向它发送消息；
- 邮件槽名字：作为client（发送方）打开邮件槽时，可以是"\\range\mailslot\[path]name"，这里range可以是 .(local computer), computerName, domainName, *(primary domain)；作为server（接收方）只能在本机建立邮件槽，名字可为"\\.\mailslot\[path]name"；

- 有关的API如：
  - CreateMailslot服务器方创建邮件槽，返回其句柄；
  - GetMailslotInfo服务器查询邮件槽的信息，如：消息长度、消息数目、读操作等待时限等；
  - SetMailslotInfo服务器设置读操作等待时限；
  - ReadFile服务器读邮件槽；
  - CreateFile客户方打开邮件槽；
  - WriteFile客户方发送消息；
- 在邮件槽的所有服务器句柄关闭后，邮件槽被关闭。这时，未读出的报文被丢弃，所有客户句柄都被关闭。

4.5.5 套接字(socket)

- 双向的，数据格式为字节流（一对一）或报文（多对一，一对多）；主要用于网络通信；
- 支持client-server模式和peer-to-peer模式，本机或网络中的两个或多个进程进行交互。提供TCP/IP协议支持
- UNIX套接字(基于TCP/IP)：send, sendto, recv, recvfrom;
- 在Windows NT中的规范称为"Winsock"(与协议独立，或支持多种协议)：WSASend, WSASendto, WSARecv, WSARecvfrom;

返回

作业

在Windows2000或Linux下用共享存储区和管道机制分别实现5个进程间的数据区复制，分析数据量对性能的影响。

4.6 死锁问题(DEADLOCK)

4.6.1 概述
4.6.2 死锁的预防
4.6.2 死锁的检测
4.6.3 死锁的避免
4.6.4 解决死锁问题的综合方法

返回

4.7 进程其他方面的举例

4.7.1 UNIX
4.7.2 Windows NT

返回