

第 4 章 进程管理

月出皓兮 苏曙光老师的课堂笔记

2022 年 3 月 19 日

目录

1 进程 (process) 概念	5
1.1 进程的定义	5
1.2 进程的特征	5
1.3 进程的状态	5
1.3.1 进程的三态模型 Running, Ready, Block	5
1.3.2 进程状态的变迁	5
1.3.3 Linux 进程的状态	5
1.3.4 Linux 显示 process 状态	6
2 进程控制块 (Process Control Block, PCB)	7
2.1 PCB 的定义	7
2.2 PCB 中的基本成员	7
2.3 Linux 的进程控制块 PCB: task_struct	7
2.3.1 进程家族关系相关的成员变量	8
2.3.2 和内存相关的成员变量	8
2.3.3 和文件相关的成员变量	8
2.3.4 和进程标识相关的成员变量	8
2.4 使用 SOURCE INSIGHT 阅读 Linux0.11 内核源码	9
2.5 进程的上下文 (约等于 PCB)	9
2.6 分时系统的进程切换过程	9

3 进程控制	10
3.1 创建进程	10
3.1.1 参数	10
3.1.2 过程	10
3.2 撤消进程	10
3.2.1 参数	10
3.2.2 过程	10
3.3 阻塞进程	11
3.3.1 引起阻塞的时机/事件	11
3.3.2 参数	11
3.3.3 过程	11
3.4 唤醒进程	11
3.4.1 引起唤醒的时机/事件	11
3.4.2 参数	12
3.5 进程控制原语	12
3.6 Windows 进程控制	12
3.6.1 WINDOWS 通过编程启动一个程序	12
3.6.2 CreateProcess	12
3.6.3 ExitProcess	13
3.6.4 TerminateProcess	13
3.7 Linux 进程控制	13
3.7.1 创建进程 fork()	13
3.7.2 fork() 执行流程	14
3.7.3 init 进程	14
3.7.4 fork 函数的实现, 以 Linux2.6 为例—do_fork()	14
3.7.5 COW, Copy On Write, 写时复制原则	14
3.7.6 进程执行特定的功能 (不同于父进程的功能), exec 函数	14
3.7.7 fork 的两种常规用法	15
3.7.8 阻塞进程 wait()	15
3.7.9 终结进程 exit()	16
3.7.10 休眠进程 Sleep()	16

4 线程 Thread	16
4.1 线程的概念	16
4.2 Windows 创建一个线程: CreateThread()	17
5 进程的相互关系	17
5.1 互斥关系	17
5.1.1 临界资源 (Critical Resource)	17
5.1.2 临界区 (Critical Section)	17
5.2 同步关系	18
6 进程的同步机制	18
6.1 同步机制的功能	18
6.1.1 设计临界区访问机制的四个原则	18
6.2 硬件方法 (访问临界区的硬件方法)	18
6.2.1 中断屏蔽方法	18
6.2.2 测试并设置指令 (Test and Set)	18
6.2.3 交换指令	19
6.3 锁机制 (访问临界区的软件方法)	19
6.3.1 上锁原语	19
6.3.2 开锁原语	19
6.4 信号灯与 P-V 操作	19
6.4.1 信号灯数据结构 (S, q)	19
6.4.2 P 操作 通过	19
6.4.3 V 操作 释放	19
6.4.4 使用信号灯 P-V 操作的实现进程互斥	21
6.4.5 使用信号灯 P-V 操作的实现进程同步	21
6.4.6 进程流图	21
6.5 经典的同步问题	21
6.5.1 生产者和消费者问题	21
6.5.2 读者 (Reader) 和编者 (Editor) 问题	21
6.6 Linux 同步机制	23
7 进程通信	24
7.1 Linux 的信号 (Signal) 机制	24

7.1.1	信号的来源	24
7.1.2	Linux 定义了 64 种信号，信号用整数 1 ~ 64 表示 . . .	24
7.1.3	注册信号 <code>signal()</code>	24
7.1.4	发送信号 <code>kill()</code>	25

1 进程 (process) 概念

1.1 进程的定义

进程是程序在某个**数据集合**上的一次**运行活动**。

数据集合：软/硬件环境，多个进程共存/共享的环境

1.2 进程的特征

1. 动态性：进程是程序的一次执行过程，动态产生/消亡
2. 并发性：进程可以同其他进程一起向前推进
3. **异步性**：进程按各自速度向前推进（必要的时候需要进行同步）
4. 独立性：进程是系统分配资源和调度 CPU 的单位；

进程与程序的区别：

进程是动态的：程序的一次执行过程。程序是静态的：一组指令的有序集合。

进程是暂存的：在内存驻留。程序是长存的：在介质上长期保存。

一个程序可能有多个进程。

1.3 进程的状态

1.3.1 进程的三态模型 Running, Ready, Block

运行状态 (Running)：进程已经占有 CPU，在 CPU 上运行。

就绪状态 (Ready)：具备运行条件但由于无 CPU，暂时不能运行。

阻塞状态 (Block) (**等待状态 (Wait)**)：因为等待某项**服务完成**或**信号来到**而不能运行的状态，例如等待：系统调用，I/O 操作，合作进程的服务或信号。

1.3.2 进程状态的变迁

进程的状态可以依据一定的条件相互转化。

1.3.3 Linux 进程的状态

可运行态 (TASK_RUNNING)：就绪与运行。

睡眠态/阻塞态/等待态：

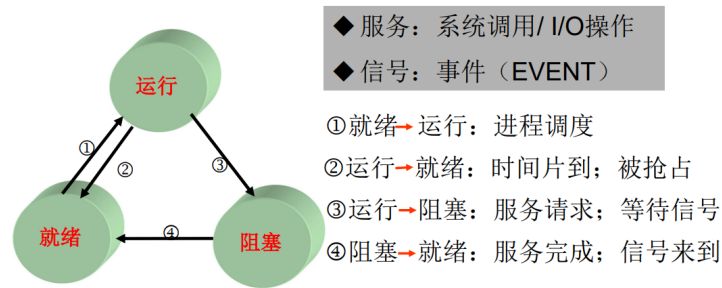


图 1: 三态模型的变化

深度睡眠（TASK_UNINTERRUPTIBLE）**不能被**其他进程通过信号和时钟中断唤醒。

浅度睡眠（TASK_INTERRUPTIBLE）**可被**其他进程的信号或时钟中断唤醒。

僵死态（TASK_ZOMBIE）：进程终止执行，释放大部分资源。

挂起态（TASK_STOPPED）：进程被挂起。

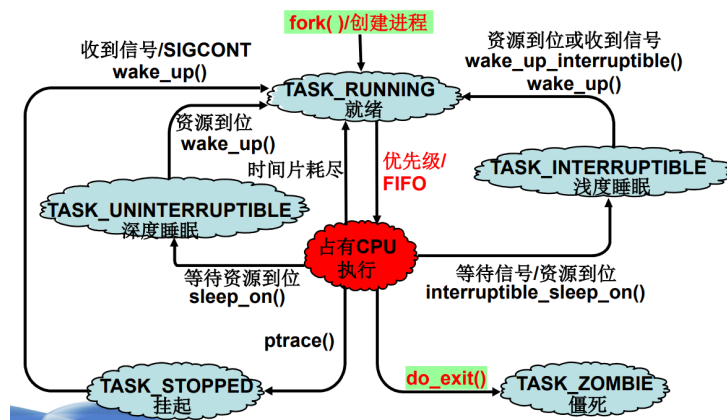


图 2: Linux 进程状态的转换

1.3.4 Linux 显示 process 状态

PID: 进程 ID

TTY: 终端名称

TIME: 进程执行时间

COMMAND: 进程对应命令/程序名

ps 命令。ps aux

2 进程控制块 (Process Control Block, PCB)

2.1 PCB 的定义

描述进程的状态、资源、和相关进程的关系的一种数据结构。

PCB 是进程的标志：创建进程时创建 PCB；进程撤销后 PCB 同时撤销。

进程 = 程序 + PCB。每当程序运行一次，创建一次 PCB，即进程运行一次。

2.2 PCB 中的基本成员

1. name (ID): 进程名称 (标识符)
2. status: 状态
3. next: 指向下一个 PCB 的指针
4. start_addr: 程序地址
5. priority: 优先级
6. cpu_status: 现场保留区 (堆栈)
7. comm_info: 进程通信机制
8. process_family: 家族
9. own_resource: 资源清单

2.3 Linux 的进程控制块 PCB: task_struct

基本内容包括

1. 进程状态
2. 调度信息
3. 进程标识符
4. 内部进程通信信息
5. 链接信息
6. 时间和计时器
7. 文件系统

8. 虚拟内存信息

9. 处理器信息/现场保留区

2.3.1 进程家族关系相关的成员变量

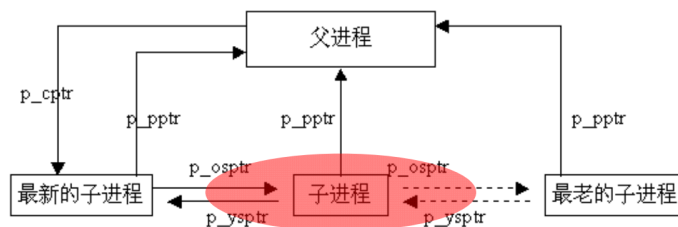


图 3: 进程家族关系

2.3.2 和内存相关的成员变量

`p->mm` 指向 `mm_struct` 结构进程的地址空间。

`p->active_mm` 指向 `mm_struct` 结构进程的当前活动地址空间。

2.3.3 和文件相关的成员变量

`p->fs`, 文件系统信息: `root` 目录和挂载点; 当前工作目录和挂载点。

`p->files` 字段包含了文件句柄表

2.3.4 和进程标识相关的成员变量

LINUX 进程的标识:

PID: 进程 ID, 每个进程有唯一编号: PID

PPID: 父进程 ID

PGID: 进程组 ID

LINUX 进程的用户标识:

UID: 用户 ID

GID: 用户组 ID

除 `init` 进程外, 每个进程都可用 `kill` 命令杀死。

2.4 使用 SOURCE INSIGHT 阅读 Linux0.11 内核源码

当读者使用盗版 SOURCE INSIGHT 时，注意断网使用。

```

struct task_struct {
/* these are hardcoded - don't touch */
    long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    long counter; /* 信号通信相关 */
    long priority;
    long signal; /* 信号通信相关 */
    struct sigaction sigaction[32]; /* 信号安装函数 */
    long blocked; /* bitmap of masked signals */
/* various fields */
    int exit_code;
    unsigned long start_code, end_code, end_data, brk, start_stack;
    long pid, father, pgrp, session, leader;
    unsigned short uid, euid, suid; /* ID */
    unsigned short gid, egid, sgid; /* ID */
    long alarm;
    long utime, stime, cutime, cstime, start_time;
    unsigned short used_math;
/* file system info */
    int tty; /* -1 if no tty, so it must be signed */ // 终端
    unsigned short umask;
    struct m_inode * pwd; /* 与文件目录相关 */
    struct m_inode * root; /* 与文件目录相关 */
    struct m_inode * executable; /* 与文件目录相关 */
    unsigned long close_on_exec;
    struct file * filp[NR_OPEN]; /* 打开的文件列表 */
/* ldt for this task 0 - zero 1 - cs 2 - ds&ss */
    struct desc_struct ldt[3]; /* 保护模式下，当前状态代码运行的空间 */
/* tss for this task */
    struct tss_struct tss; /* 上下文 */
} /* end task_struct */

```

2.5 进程的上下文（约等于 PCB）

Context，进程运行环境，约等于 PCB。

2.6 分时系统的进程切换过程

进程的上下文在 CPU 中交换

换出进程的上下文离开 CPU（到栈 +PCB 上去）

换入进程的上下文进入 CPU（从栈 +PCB 上来）

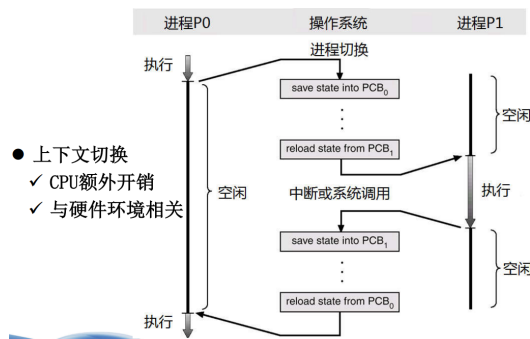


图 4: 进程切换的过程

3 进程控制

在进程生存全期间，对其全部行为的控制。主要包括创建进程，撤消进程，阻塞进程，唤醒进程。

3.1 创建进程

创建一个具有指定标识（ID）的进程。

3.1.1 参数

进程标识、优先级、进程起始地址、CPU 初始状态、资源清单等。

3.1.2 过程

1. 创建一个空白 PCB
2. 赋予进程标识符 ID
3. 为进程分配空间
4. 初始化 PCB(CPU 的状态，内存，优先级，进程状态等)。
5. 插入相应的进程队列（**就绪队列**）

3.2 撤消进程

撤消一个指定的进程，收回进程所占有的资源，撤消该进程的 PCB。
大概率出现于正常结束，异常结束，外界干预这三种情况。

3.2.1 参数

被撤消的进程名（ID）

3.2.2 过程

1. 在 PCB 队列中检索出该 PCB
2. 获取该进程的状态。
3. 若该进程处在运行态，立即终止该进程。
4. 是否需要撤销其子进程？若是，则递归地撤销其子进程。若不是，将子进程挂接到 init 进程下。

5. 释放进程占有的资源
6. 将进程从 PCB 队列中移除

3.3 阻塞进程

停止进程执行，变为阻塞。

3.3.1 引起阻塞的时机/事件

请求系统服务（由于某种原因，OS 不能立即满足进程的要求）
启动某种操作（进程启动某操作，阻塞等待该操作完成）
新数据尚未到达（A 进程要获得 B 进程的中间结果，A 进程等待）
无新工作可作（进入 idle 进程/pause()，等待新任务到达）

3.3.2 参数

阻塞原因
不同原因构建有不同的阻塞队列。

3.3.3 过程

停止运行
将 PCB “运行态”改“阻塞态”
插入对应的阻塞队列
转调度程序

3.4 唤醒进程

唤醒处于阻塞队列当中的某个进程。

3.4.1 引起唤醒的时机/事件

系统服务由不满足到满足
I/O 完成
新数据到达
进程提出新请求（服务）

3.4.2 参数

被唤醒进程的标识

3.5 进程控制原语

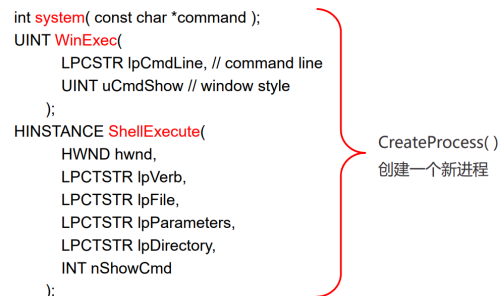
由若干指令构成的具有特定功能的函数，具有**原子性**，其操作不可分割。

进程控制，要不然全部完成，要不然就不完成。

创建原语，撤消原语，阻塞原语，唤醒原语。

3.6 Windows 进程控制

3.6.1 WINDOWS 通过编程启动一个程序



```

int system( const char *command );
UINT WinExec(
    LPCSTR lpCmdLine, // command line
    UINT uCmdShow // window style
);
HINSTANCE ShellExecute(
    HWND hwnd,
    LPCTSTR lpVerb,
    LPCTSTR lpFile,
    LPCTSTR lpParameters,
    LPCTSTR lpDirectory,
    INT nShowCmd
);
    
```

CreateProcess()
创建一个新进程

图 5: WINDOWS 通过编程启动一个程序

应注意的是，CreateProcess 只能创建 32 位进程，在 Win10 中运行存在兼容性问题。

3.6.2 CreateProcess

```

BOOL CreateProcess(
    LPCTSTR lpApplicationName, // 可执行程序名
    LPTSTR lpCommandLine, // [可执行程序名] 程序参数, 例如打开的文件等
    LPSECURITY_ATTRIBUTES lpProcessAttributes,
    LPSECURITY_ATTRIBUTES lpThreadAttributes,
    
```

```

    BOOL bInheritHandles,
    DWORD dwCreationFlags, //创建标志
    LPVOID lpEnvironment,
    LPCTSTR lpCurrentDirectory,
    LPSTARTUPINFO lpStartupInfo,
    LPPROCESS_INFORMATION lpProcessInformation
); // lpProcessInformation : 接收新进程的识别信息
创建进程内核对象，创建虚拟地址空间
装载 EXE 和/或 DLL 的代码和数据到地址空间中
创建主线程和线程内核对象
启动主线程，进入主函数 (main)

```

3.6.3 ExitProcess

VOID ExitProcess(UINT uExitCode) 结束自身进程

3.6.4 TerminateProcess

VOID TerminateProcess (HANDLE hProcess, UINT uExitCode)
结束目标进程。

3.7 Linux 进程控制

3.7.1 创建进程 fork()

pid_t fork(void); 子进程：新建的进程
父进程：fork() 的调用者
子进程是父进程的**复制**。
父进程和子进程并发运行。
应注意的是，在子进程中 PID = 0，在父进程中 PID > 0 (等于子进程 ID)，若进程创建失败 PID=-1。
fork 为什么在子进程中没有再建立新的进程呢？新的进程的 CS:IP 指向在 fork 函数后的第一个指令。

3.7.2 fork() 执行流程

分配 task_struct 结构

拷贝父进程, 复制正文段、用户数据段及系统数据段, 复制 task_struct 的大部分内容, 修改 task_struct 的小部分内容

把新进程的 task_struct 保存在 task 队列中。

新进程置于**就绪状态**

3.7.3 init 进程

在 Linux 系统初启时, 构建进程 0。进程 0 用 fork() 创建 init 进程 (进程 1)。

其他进程由当前进程通过 fork() 创建。父进程 → 子进程

3.7.4 fork 函数的实现, 以 Linux2.6 为例—do_fork()

1. 分配物理页面存放 task_struct 结构和内核空间的堆栈
2. 把当前进程 task_struct 结构中所有内容都拷贝到新进程中
3. 判断用户进程数量是否超过了最大限制, 否则不许 fork
4. 子进程初始状态设 TASK_UNINTERRUPTIBLE
5. 拷贝进程的所有信息
6. 进程创建后与父进程链接起来形成一个进程组
7. 唤醒进程, 将其挂入可执行队列等待被调度

3.7.5 COW, Copy On Write, 写时复制原则

父进程的资源被设置为只读, 当父进程或子进程试图修改某些内容时, 内核才在修改前将该部分进行拷贝——**写时复制**。

fork() 的实际开销只是复制父进程的页表以及给予进程创建唯一的 PCB

3.7.6 进程执行特定的功能 (不同于父进程的功能), exec 函数

功能: 在子进程空间运行指定的**可执行程序**。

步骤:

1. 根据**文件名**找到相应的**可执行文件**。
2. 可执行文件的内容填入子进程的地址空间。

3. `exec` 调用成功就会进入新进程执行且不再返回。`exec` 调用失败：继续从调用点向下执行。

Unix/Linux 可以通过执行用户命令建立新的进程

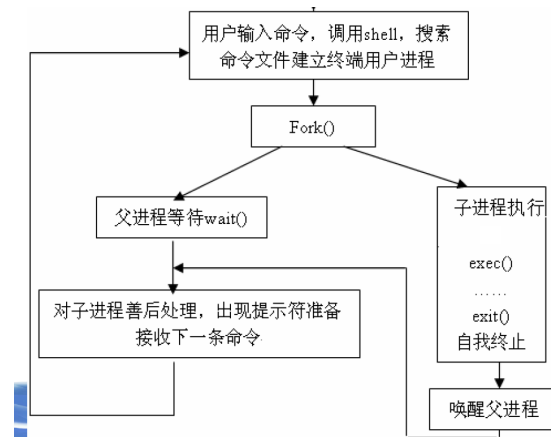


图 6: 执行用户命令建立新的进程

3.7.7 fork 的两种常规用法

(1) 子进程复制父进程

```

if(pid == 0)
    {子进程}
else if(pid > 0)
    {父进程}
  
```

(2) 子进程执行不同的程序

```

if(pid == 0)
    {exec("Exc file name")}
else if(pid > 0)
    {父进程}
  
```

3.7.8 阻塞进程 wait()

进程调用 `wait(int &status)` 阻塞自己。若没有子进程结束，等待子进程结束：继续阻塞。若已经结束，`wait` 收集该子进程信息并彻底销毁它后返

回。返回值为被收集的子进程的进程 ID。Status 接收子进程退出时的退出代码（按位处理）。若忽略子进程的退出信息。pid = wait(NULL)。

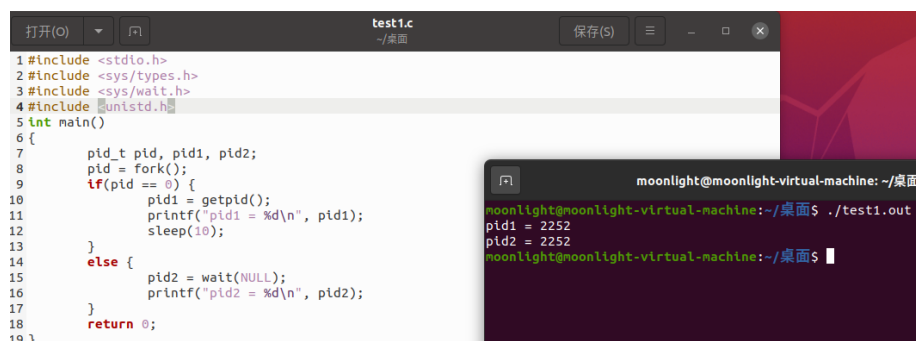
3.7.9 终结进程 exit()

调用 void exit(int status) 终结进程。

进程终结时要释放资源并向父进程报告，利用 status 向父进程报告结束时的退出代码，进程变为僵尸状态，保留部分 PCB 信息供 wait 收集（正常结束还是异常结束，占用总系统 cpu 时间，缺页中断次数）。最后调用 schedule() 函数，重新调度进程运行。

3.7.10 休眠进程 Sleep()

Sleep(int nSecond)，进程暂停执行 nSeconds，系统暂停调度该进程，相当于 windows 挂起操作 resume()，挂起指定秒。



```

1 #include <stdio.h>
2 #include <sys/types.h>
3 #include <sys/wait.h>
4 #include <unistd.h>
5 int main()
6 {
7     pid_t pid, pid1, pid2;
8     pid = fork();
9     if(pid == 0) {
10         pid1 = getpid();
11         printf("pid1 = %d\n", pid1);
12         sleep(10);
13     }
14     else {
15         pid2 = wait(NULL);
16         printf("pid2 = %d\n", pid2);
17     }
18     return 0;
19 }

```

```

moonlight@moonlight-virtual-machine: ~/桌面
moonlight@moonlight-virtual-machine:~/桌面$ ./test1.out
pid1 = 2252
pid2 = 2252
moonlight@moonlight-virtual-machine:~/桌面$

```

图 7: 一个小例子

4 线程 Thread

4.1 线程的概念

线程是进程内的一个执行路径；一个进程可以创建和包含多个线程；线程之间共享 CPU 可以实现并发运行；创建线程比创建进程开销要小；线程间通信十分方便。

线程的应用：如果把程序中某些函数创建为线程，那么这些函数将可以并发运行

现代操作系统的进程/线程计算模型：进程 = 资源集 + 线程组，进程为线程组提供资源。

线程也有着创建，就绪，运行，等待，中止等状态与状态变化。

程序可分为单线程程序和多线程程序。Windows 程序缺省创建一个线程（主线程，main 线程）。

多线程技术广泛应用于：多个功能需要并发的地方，改善窗口交互性的地方，需要改善程序结构的地方，多核 CPU 上的应用，充分发挥多核性能。

4.2 Windows 创建一个线程: *CreateThread()*

功能：把一个函数创建为一个线程。

```
HANDLE CreateThread(  
    LPSECURITY_ATTRIBUTES lpThreadAttributes,  
    SIZE_T dwStackSize,  
    LPTHREAD_START_ROUTINE lpStartAddress, //线程函数  
    __drv_aliasesMem LPVOID lpParameter, //线程函数的参数  
    DWORD dwCreationFlags,  
    LPDWORD lpThreadId  
);
```

5 进程的相互关系

5.1 互斥关系

多个进程由于共享具有**独占性的资源**，必须确保各进程**互斥地**存取资源，即确保没有任何两个进程**同时存取资源**。

目的：保证并发结果的正确性。

5.1.1 临界资源 (Critical Resource)

一次只允许一个进程独占访问（使用）的资源

临界资源的访问具有排他性；

5.1.2 临界区 (Critical Section)

进程中访问临界资源的程序段。

并发进程不能同时进入“临界区”；

5.2 同步关系

若干合作进程为了共同完成一个任务，需要相互协调运行步伐：一个进程 A 开始某个操作之前要求另一个进程 B 必须已经完成另一个操作，否则进程 A 只能等待。

互斥关系属于特殊的同步的关系。

6 进程的同步机制

6.1 同步机制的功能

(1) 当进程不能执行即将要执行的某个操作（运行条件不满足时）能让该进程立即暂停执行该操作；

(2) 当被暂停的操作在运行条件一旦满足时，相应进程能被尽快唤醒以便继续运行。

另外，同步进程在实现上也需要满足原子性。

6.1.1 设计临界区访问机制的四个原则

忙则等待，空闲让进（当无进程处于临界区时，任何有权进程可进入临界区），有限等待，让权等待（等待进程放弃 CPU）。

6.2 硬件方法（访问临界区的硬件方法）

6.2.1 中断屏蔽方法

进入临界区前，执行“关中断”指令。离开临界区后，执行“开中断”指令。

让系统不再进行进程切换。

6.2.2 测试并设置指令 (Test and Set)

TS(boolean *lock)

6.2.3 交换指令

SWAP(int *a, int *b)

6.3 锁机制 (访问临界区的软件方法)

初始化锁的状态 $S = 1$ (可用)

进入临界区之前执行上锁 Lock(s) 操作;

离开临界区之后执行开锁 unlock(s) 操作。

锁机制只满足了 3 个原则, 未满足“让权等待”原则。

6.3.1 上锁原语

第 1 步: 检测锁 S 的状态 (0 或 1?)

第 2 步: 如果 $S=0$, 则返回第 1 步

第 3 步: 如果 $S=1$, 则置其为 0

6.3.2 开锁原语

第 1 步: 把锁 S 的状态置 1

6.4 信号灯与 P-V 操作

进程在运行过程中受信号灯的**控制** (进程因信号灯的状态被阻塞或被唤醒), 并能**改变信号灯**。

6.4.1 信号灯数据结构 (S, q)

一个二元矢量 (S, q)。

S: 信号量, 整数, 初值非负。

q: 队列 (进程 PCB 集合), 初值为空集。

6.4.2 P 操作 通过

P 操作可能使进程在调用处阻塞。(S<0 时)

6.4.3 V 操作 释放

P 操作可能使进程在调用处唤醒。(S≤0 时)

```
P(S,q)
{
    S = S - 1;
    if (S < 0 )
    {
        Insert( Caller , q );
        Block( Caller );
        转调度函数( );
    }
}
```

图 8: P 操作的伪代码

```
V(S,q)
{
    S = S + 1;
    if ( S ≤ 0 )
    {
        Remove( q , PID ); // PID: 进程ID
        Wakeup( PID );
    }
}
```

图 9: V 操作的伪代码

6.4.4 使用信号灯 P-V 操作的实现进程互斥

M 个临界资源：允许最多 M 个进程同时处于临界区。

进入临界区之前先执行 P 操作；离开临界区之后再执行 V 操作；S 的初值设置为临界资源的数量。

6.4.5 使用信号灯 P-V 操作的实现进程同步

同步机制实质：运行条件不满足时，能让进程**暂停**；运行条件满足时，能让进程立即**继续**。

在某个需要特定条件的关键操作之前，执行 P 操作。

在某个影响别的操作的关键操作之后，执行 V 操作。

定义有意义的**信号量**（可能多个）S，并设置合适的初值信号量 S 能明确地表示“**运行条件**”。不合理的初值不仅达不到同步的目的，还会发生**死锁**。

6.4.6 进程流图

6.5 经典的同步问题

6.5.1 生产者和消费者问题

一群生产者（Producer）通过缓冲区向一群消费者（Consumer）提供产品（数据）。共享**缓冲区**。

1. 缓冲区满时不能存；缓冲区空时不能取（消费）
2. 每个时刻生产者或消费者**只能有 1 个存或取缓冲区**

6.5.2 读者（Reader）和编者（Editor）问题

读者和编者问题描述的是一群读者和一群编者共同读写同一本书的问题。

- (1) 允许多个读者同时来读；
- (2) 不允许读者、编者同时读和编；
- (3) 不允许多个编者同时编。

解决方案：记录一个目前读者的数目。

```

1  int DATA = 0; //信号量: 缓冲区中新数据的个数, 初值0
2  int SPACE = 5; //信号量: 缓冲区中空位置的个数, 初值5
3  int MUTEX = 1; //信号量: 缓冲区互斥使用, 初值1
4  //生产者进程 i = 1 .. m
5  producer_i ( )
6  {
7      while( TRUE )
8      {
9          生产1个产品/数据;
10         P(SPACE);
11         P(MUTEX);
12         存1个产品/数据到缓冲区;
13         V(MUTEX);
14         V(DATA);
15     }
16 }
17 //消费者进程 j = 1 .. k
18 consumer_j ( )
19 {
20     while( TRUE )
21     {
22         P(DATA);
23         P(MUTEX);
24         从缓冲区取1个产品/数据;
25         V(MUTEX);
26         V(SPACE);
27         消费一个产品/数据;
28     }
29 }

```

图 10: 生产者和消费者问题的解决

```

int ReadCount = 0; /* 读者计数 */
int mutex = 1; /* 互斥量: ReadCount是临界资源 */
int editor = 1; /* 互斥量: 编者对编者和读者的互斥 */

```

```

Reader_i ( )
while (true)
{
    P(mutex);
    ReadCount ++;
    if (ReadCount ==1)
        P(editor);
    V(mutex);
    读书;
    P(mutex);
    ReadCount --;
    if (ReadCount ==0)
        V(editor);
    V(mutex);
};

```

```

Editor_i ( )
while (true)
{
    P(editor);
    编书;
    V(editor);
};

```

如何实现:

- 1.编者之间的互斥?
- 2.编者和读者之间的互斥?
- 3.读者之间不互斥?

图 11: 读者 (Reader) 和编者 (Editor) 问题的解决

Question: 编者优先和读者优先与公平竞争的实现？读者优先：当存在读操作时，编操作有可能将被无限延迟。

编者优先：当有读操作时，如果有编者请求访问，这时应禁止后续的读操作请求，应让当前读操作执行完毕后立即让编者进程执行。只有在无编者操作的情况下才允许读进程再次运行。

增加一个信号量，并增加一对 P-V 操作，实现写者优先。

1. 通过添加信号量 read 实现写者到来时能够打断读者进程。
2. 设置信号量 fileSrc 实现读写者对临界资源的访问。
3. 设置计数器 writeCount 来统计当前阻塞的写者进程的数目，设置信号量 writeCountSignal 完成对 writeCount 计数器资源的互斥访问。
4. 设置计数器 readCount 来统计访问临界资源的读者数目，设置信号量 readCountSignal 完成对 readCount 计数器资源的互斥访问。

6.6 Linux 同步机制

使用 fork() 函数进行同步的例子，但要注意到的是父子进程对于变量是“即时复制”的副本关系，对于文件是使用了同一个文件指针。

父子进程共享普通变量

```
int main(int argc, char *argv[])
{
    pid_t pid;
    int i = 1;
    pid = fork(); // 创建新进程
    if (pid == 0) // 子进程
    {
        printf("In child i=%d\n", i); // 打印 i 值
        exit(0);
    }
    else // 父进程
    {
        sleep(10); // 10秒休眠，让子进程先执行
        printf("In parent i=%d\n", i); // 打印 i 值
        exit(0);
    }
}
```

思考：父子进程输出i是多少？

[root@michael root]# ./test

In child i = 2
In parent i = 1

结论：对于普通变量，父子进程各自操作变量副本，互相不影响。

图 12: 父子进程共享普通变量

父子进程共享文件资源

```

int main(int argc, char *argv[ ])
{
    int file;
    char *chA = "A";
    char *chB = "B";
    char *chC = "C";
    //打开（创建）一个文件
    file = open("test.txt");
    write(file, chA, strlen(chA));
    pid_t pid = fork(); //创建新进程
    if(pid == 0) //子进程
    {
        write(file, chB, strlen(chB)); //写test.txt
        exit(0);
    }
    else //父进程
    {
        sleep(10); //休眠10秒，让子进程先运行完
        write(file, chC, strlen(chC)); //写test.txt
        exit(0);
    }
}

```

思考：程序运行后test.txt内容是什么？

test.txt内容是：
ABC或ACB或A或B或C或AC或AB或.....?

结论：对于文件，父子进程
共享同一文件和读写指针。

图 13: 父子进程共享文件资源

7 进程通信

7.1 Linux 的信号 (Signal) 机制

Linux 软中断机制，低级通信原语，模拟硬件中断。

7.1.1 信号的来源

键盘输入特殊组合键产生信号，例：“Ctrl+C”

执行终端命令产生信号，例：kill 系列命令

程序中调用函数产生信号，例：kill(), abort()

硬件异常或内核产生相应信号。例：内存访问错

7.1.2 Linux 定义了 64 种信号，信号用整数 1~64 表示

SIGINT 信号编号为 2，进程结束信号。

SIGUSR1 信号编号为 11，用户自定义信号 1

7.1.3 注册信号 signal()

sighandler_t signal(int signum, sighandler_t handler);

为指定信号注册信号处理函数。当进程收到 signum 信号时，立即自动调用 Handle 函数执行。一般在进程初始化时使用该函数注册信号处理函数。

7.1.4 发送信号 kill()

```
int kill(pid_t pid, int sig);
```

向目标进程 PID 发送 SigNal 信号。