

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 进程的类型	5
1.3.1 按使用资源的权限	5
1.3.2 按对 CPU 的依赖性	5
1.4 进程的状态	5
1.4.1 进程的三态模型 Running, Ready, Block	5
1.4.2 进程的五态模型 New, Exit	6
1.4.3 进程的七态模型 Suspend, Resume	6
1.4.4 进程状态的变迁	6
1.4.5 Linux 进程的状态	6
1.4.6 Linux 显示 process 状态	8
2 进程控制块 (Process Control Block, PCB)	8
2.1 PCB 的定义	8
2.2 PCB 中的基本成员	8
2.3 Linux 的进程控制块 PCB: task_struct	8
2.4 使用 SOURCE INSIGHT 阅读 Linux2.6 内核源码	9
2.4.1 进程家族关系相关的成员变量	10
2.4.2 和内存相关的成员变量	10
2.4.3 和文件相关的成员变量	10

2.4.4	和进程标识相关的成员变量	10
2.5	使用 SOURCE INSIGHT 阅读 Linux0.11 内核源码	11
2.6	进程的上下文 (约等于 PCB)	11
2.7	分时系统的进程切换过程	11
3	进程控制	12
3.1	创建进程	12
3.1.1	参数	12
3.1.2	过程	12
3.2	撤消进程	12
3.2.1	参数	12
3.2.2	过程	12
3.3	阻塞进程	13
3.3.1	引起阻塞的时机/事件	13
3.3.2	参数	13
3.3.3	过程	13
3.4	唤醒进程	13
3.4.1	引起唤醒的时机/事件	13
3.4.2	参数	13
3.5	进程控制原语	14
3.6	Windows 进程控制	14
3.6.1	WINDOWS 通过编程启动一个程序	14
3.6.2	CreateProcess	14
3.6.3	ExitProcess	15
3.6.4	TerminateProcess	15
3.7	Linux 进程控制	15
3.7.1	创建进程 fork()	15
3.7.2	fork() 执行流程	15
3.7.3	init 进程	16
3.7.4	fork 函数的实现, 以 Linux2.6 为例—sys_fork, do_fork()	16
3.7.5	fork 函数的实现, 以 Linux0.11 为例—_sys_fork	16
3.7.6	COW, Copy On Write, 写时复制原则	19
3.7.7	进程执行特定的功能 (不同于父进程的功能), exec 函数	19
3.7.8	fork 的两种常规用法	19

3.7.9 阻塞进程 wait()	19
3.7.10 终结进程 exit()	21
3.7.11 休眠进程 Sleep()	21
4 线程 Thread	22
4.1 线程的概念	22
4.2 Windows 创建一个线程: CreateThread()	22
4.2.1 创建远程线程: CreateRemoteThread()	23
4.3 Linux 线程	23
4.3.1 内核线程 (Kernel Thread)	23
4.3.2 轻量级进程 (Light Weight Process, LWP)	23
4.3.3 用户线程 (User Thread)	23
4.4 使用线程的一些问题	24
5 进程的相互关系	24
5.1 互斥关系	24
5.1.1 临界资源 (Critical Resource)	24
5.1.2 临界区 (Critical Section)	24
5.2 同步关系	24
6 进程的同步机制	24
6.1 硬件方法	25
6.1.1 中断屏蔽方法	25
6.1.2 测试并设置指令 (Test and Set)	25
6.1.3 交换指令	25
6.2 锁机制	25
6.2.1 上锁原语	25
6.2.2 开锁原语	25
6.2.3 设计临界区访问机制的四个原则	25
6.3 信号灯与 P-V 操作	25
6.3.1 信号灯数据结构 (S, q)	26
6.3.2 P 操作 通过	26
6.3.3 V 操作 释放	26
6.3.4 使用信号灯 P-V 操作的实现进程互斥	26

6.3.5	使用信号灯 P-V 操作的实现进程同步	27
6.4	经典的同步问题	27
6.4.1	生产者和消费者问题	27

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；

1.3 进程的类型

1.3.1 按使用资源的权限

系统进程：指系统内核相关的进程。

用户进程：运行于用户态的进程。

1.3.2 按对 CPU 的依赖性

偏 CPU 进程：计算型进程

偏 I/O 进程：侧重于 I/O 的进程

1.4 进程的状态

1.4.1 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。

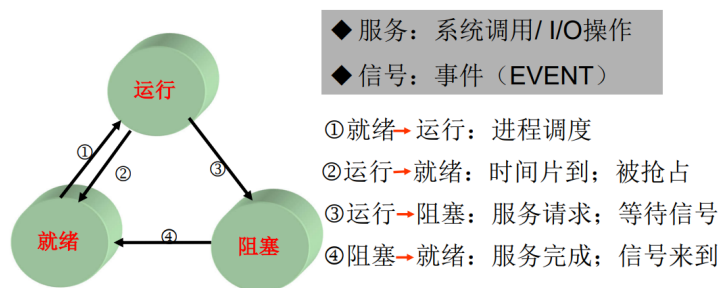


图 1: 三态模型的变化

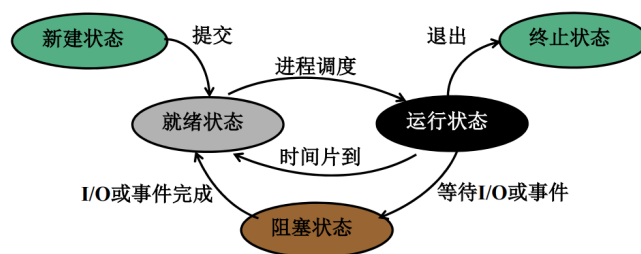


图 2: 具有新建 (new) 和终止 (terminate) 状态的进程状态

1.4.2 进程的五态模型 New, Exit

新建状态 (New)：对应于进程被创建时的状态，尚未进入就绪队列。

终止状态 (Exit)：处于终止态的进程不再被调度执行，下一步将被系统撤销，最终从系统中消失。

1.4.3 进程的七态模型 Suspend, Resume

挂起与解挂 Suspend, Resume

1.4.4 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。

1.4.5 Linux 进程的状态

可运行态 (TASK_RUNNING)：就绪与运行。

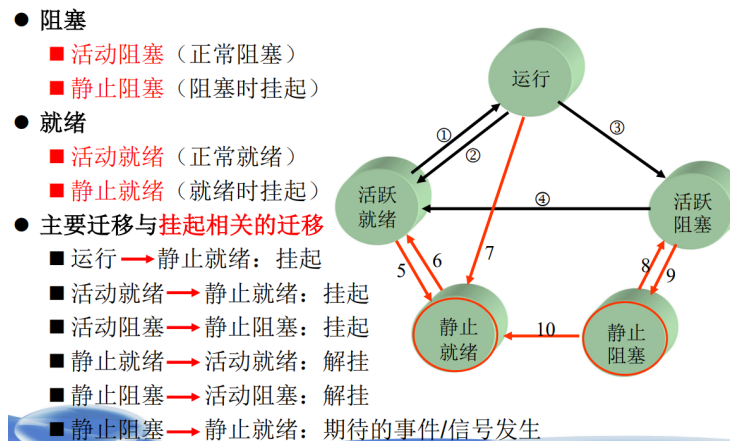


图 3: 支持挂起 (suspend) 和解挂 (resume) 操作的进程状态

睡眠态/阻塞态/等待态:

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。

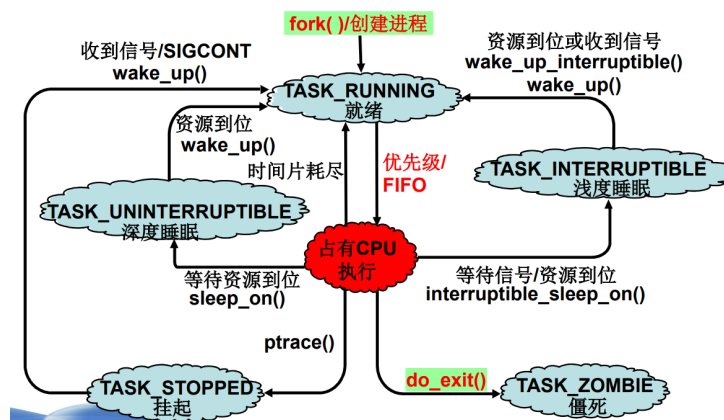


图 4: Linux 进程状态的转换

1.4.6 Linux 显示 process 状态

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. 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

2.4 使用 SOURCE INSIGHT 阅读 Linux2.6 内核源码

```

struct task_struct {
    volatile long state; /* -1 unrunnable, 0 runnable, >0 stopped */
    void *stack;
    atomic_t usage; /* 有几个进程正在使用该结构 */
    unsigned int flags;
    unsigned int ptrace;

    int lock_depth; /* BKL lock depth */

    int oncpu; /* 在哪个CPU上运行 */

    int prio, static_prio, normal_prio; /* 静态优先级, 动态优先级 */
    struct list_head run_list;
    const struct sched_class *sched_class; /* 与调度相关的函数 */
    struct sched_entity se; /* 调度实体 */

    unsigned int policy; /* 调度策略 */
    cpumask_t cpus_allowed;
    unsigned int time_slice;
    struct sched_info sched_info; /* 调度相关的信息, 如在CPU上运行的时间/在队列中等待的时间等。 */

    struct list_head tasks; /* 任务队列 */
    struct list_head ptrace_children;
    struct list_head ptrace_list;

    struct mm_struct *mm, *active_mm;
    /* mm是进程的内存管理信息, active_mm指向结构进程当前活动的地址空间 */

    /* task state */
    struct linux_binfmt *binfmt;
    int exit_state; /* 进程退出时的状态 */
    int exit_code, exit_signal; /* 进程退出时发出的信号 */
    unsigned int personality;
    unsigned did_exec:1;
    pid_t pid; /* 进程ID */
    pid_t tgid; /* 进程组ID */
    /* 进程家族关系相关的成员变量 */
    struct task_struct *real_parent; /* real parent process (when being debugged) */
    struct task_struct *parent; /* parent process */
    struct list_head children; /* list of my children */
    struct list_head sibling; /* linkage in my parent's children list */
    struct task_struct *group_leader; /* threadgroup leader */

    /* PID/PID hash table linkage. */
    struct pid_link pids[PIDTYPE_MAX];
    struct list_head thread_group;

    unsigned int rt_priority;
    cputime_t utime, stime, utimescaled, stimescaled;
    cputime_t gtime;
    cputime_t prev_utime, prev_stime;

    /* process credentials */
    uid_t uid, euid, suid, fsuid;
    gid_t gid, egid, sgid, fsgid;
    struct group_info *group_info;

    /* file system info */
    int link_count, total_link_count;
    struct sysv_sem sysvsem;
    /* CPU-specific state of this task */
    struct thread_struct thread;
    /* filesystem information */
    struct fs_struct *fs; /* 文件系统信息 */
    /* open file information */
    struct files_struct *files; /* 使用的文件 */
    /* namespaces */
    struct nsproxy *nsproxy;

    /* signal handlers */
    struct signal_struct *signal; /* pending sigs */
    struct sighand_struct *sighand;

    sigset_t blocked, real_blocked; /* max=sked sigs */
    sigset_t saved_sigmask;
    struct sigpending pending;

    tty_struct *tty; /* 终端 */

```

此处仅展示了部分信息。

2.4.1 进程家族关系相关的成员变量

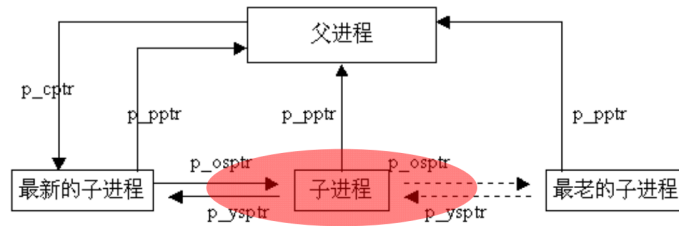


图 5: 进程家族关系

2.4.2 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。

2.4.3 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

2.4.4 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

INUX 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。

2.5 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 » ;

```

2.6 进程的上下文（约等于 PCB）

Context，进程运行环境，约等于 PCB。

2.7 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）

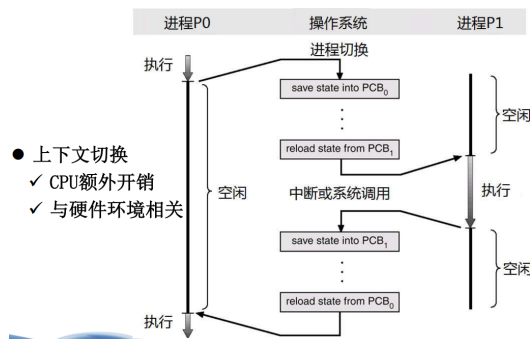


图 6: 进程切原的过程

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 的 CPU 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。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。
4. 释放进程占有的资源
5. 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

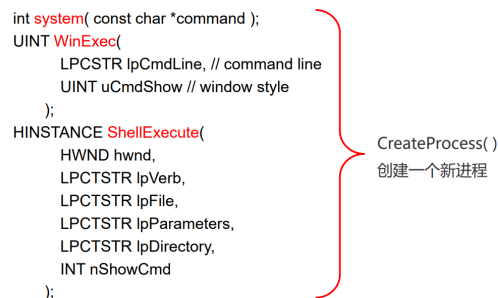
由若干指令构成的具有特定功能的函数，具有原子性，其操作不可分割。

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 7: 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

1. 分配 task_struct 结构
2. 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)
3. 把新进程的 task_struct 结构地址保存在 task 队列。
4. 新进程置于就绪状态。

3.7.3 init 进程

在 Linux 系统初启时，生成 init 进程 (1 号进程)。其他进程由当前进程通过系统调用 `fork()` 建立。

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

`fork` 的系统调用的接口的入口地址为 `sys_fork`，系统调用从寄存器中提取用户空间提供的信息，调用 `do_fork` 函数。

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 fork 函数的实现，以 Linux0.11 为例—`__sys_fork`

首先 `fork` 函数本身是个系统调用，在 Linux0.11 中使用 INT 80H 中断，实现系统调用。

```
#define __syscall0(type,name) \
type name(void) \
{ \
    long __res; \
    __asm__ volatile ("int $0x80" \
        : "=a" (_res) \
        : "0" (__NR_#name)); \
    if (_res >= 0) \
        return (type) __res; \
    errno = -_res; \
    return -1; \
}
```

系统调用编号的声明。(include/unistd.h)

```
#define __NR_fork 2
```

系统调用函数的声明(与系统调用处理函数指针表)(include/linux/sys.h)

```
rnell | sys.h (D:\Linux0.11\core\...\linux) % system_calls (D:\Linux0.11\core\...\kernel) | unistd.h (include) |
extern int sys_setup();
extern int sys_exit();
extern int sys_fork();
```


kernel)	sys.h (D:\Linux0.11core\...\linux) %	system_calls (D:\Linux0.11core\...\kernel)	unistd.h (include)
	<pre> fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read, sys_write, sys_open, sys_close, sys_waitpid, sys_creat, sys_link, sys_unlink, sys_execve, sys_chdir, sys_time, sys_mknod, sys_chmod, sys_chown, sys_break, sys_stat, sys_lseek, sys_getpid, sys_mount, sys_umount, sys_setuid, sys_getuid, sys_stime, sys_ptrace, sys_alarm, sys_fstat, sys_pause, sys_ftime, sys_kill, sys_rename, sys_access, sys_nice, sys_ftime, sys_sync, sys_kill, sys_rename, sys_mkdir, sys_rmdir, sys_dup, sys_pipe, sys_times, sys_prof, sys_brk, sys_setgid, sys_getgid, sys_signal, sys_geteuid, sys_getegid, sys_acct, sys_phys, sys_lock, sys_ioctl, sys_fcntl, sys_mpx, sys_setpgid, sys_ulimit, sys_uname, sys_umask, sys_chroot, sys_ustat, sys_dup2, sys_getppid, sys_getpgrp, sys_setsid, sys_sigaction, sys_getmask, sys_ssetmask, sys_setreuid, sys_setregid }; </pre>		

系统调用函数的定义。(kernel/system_calls)

```

### sys_fork()调用，用于创建子进程，是system_call功能2。
# 首先调用C函数find_empty_process()，取得一个进程号PID。
# 然后调用copy_process()复制进程。
.align 2
sys_fork:
    call find_empty_process
    testl %eax,%eax          # 在eax中返回进程号任务数组中的任务号(数组项)
    js 1f
    pushl %gs
    pushl %esi
    pushl %edi
    pushl %ebp
    pushl %eax
    call copy_process
    addl $20,%esp            # 丢弃这里所有压栈内容。
1: ret

```

find_empty_process 函数的编写

```

// 为新进程取得不重复的进程号last_pid.函数返回在任务数组中的任务号(数组项)。
int find_empty_process(void)
{
    int i;

    // 首先获取新的进程号。如果last_pid增1后超出进程号的整数表示范围，则重新从1开始
    // 使用pid号。然后在任务数组中搜索刚设置的pid号是否已经被任何任务使用。如果是则
    // 跳转到函数开始处重新获得一个pid号。接着在任务数组中为新任务寻找一个空闲项，并
    // 返回项号，last_pid是一个全局变量，不用返回。如果此时任务数组中64个项已经被全部
    // 占用，则返回出错码。
    repeat:
        if ((++last_pid)<0) last_pid=1;
        for(i=0 ; i<NR_TASKS ; i++)
            if (task[i] && task[i]->pid == last_pid) goto repeat;
        for(i=1 ; i<NR_TASKS ; i++) // 任务0项被排除在外
            if (!task[i])
                return i;
        return -EAGAIN;
}

```

copy_process 函数的实现

```

// 3. 调用sys_call_table中sys_fork函数时压入栈的返回地址(用参数none表示);
// 4. 在调用copy_process()分配任务数组项号。
int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,
                long ebx, long ecx, long edx,
                long fs, long es, long ds,
                long eip, long cs, long eflags, long esp, long ss)
{
    struct task_struct *p;
    int i;
    struct file *f;

    // 首先为新任务数据结构分配内存。如果内存分配出错, 则返回出错码并退出。
    // 然后将新任务结构指针放入任务数组的nr项中。其中nr为任务号, 由前面
    // find_empty_process()返回。接着把当前进程任务结构内容复制到刚申请到
    // 的内存页面p开始处。
    p = (struct task_struct *) get_free_page();
    if (!p)
        return -EAGAIN;
    task[nr] = p;
    *p = *current; // 把当前进程任务结构内容复制到刚申请到的内存页面p开始处。

    // 随后对复制来的进程结构内容进行一些修改, 作为新进程的任务结构。先将
    // 进程的状态置为不可中断等待状态, 以防止内核调度其执行。然后设置新进程
    // 的进程号pid和父进程号father。
    p->state = TASK_UNINTERRUPTIBLE;
    p->pid = last_pid; // 新进程号, 也由find_empty_process()得到。
    p->father = current->pid; // 设置父进程
    p->counter = p->priority;
    p->signal = 0;
    p->alarm = 0;
    p->leader = 0;
    p->utime = p->stime = 0;
    p->cutime = p->cstime = 0;
    p->start_time = jiffies;
    // 再修改任务状态段TSS数据(一个进程的上下文)。由于系统给任务结构p分配了1页新内存, 所以(PAGE_SIZE+
    // (long)p)让esp0正好指向该页顶端。ss0:esp0用作程序在内核态执行时的栈。另外,
    // 每个任务在GDT表中都有两个段描述符, 一个是任务的TSS段描述符, 另一个是任务的LDT
    // 表描述符。下面语句就是把GDT中本任务LDT段描述符和选择符保存在本任务的TSS段中。
    // 当CPU执行切换任务时, 会自动从TSS中把LDT段描述符的选择符加载到ldtr寄存器中。
    p->tss.back_link = 0;
    p->tss.esp0 = PAGE_SIZE + (long) p;
    p->tss.ss0 = 0x10;
    p->tss.eip = eip;
    p->tss.eflags = eflags;
    p->tss.eax = 0; // 这是当fork()返回时新进程会返回0的原因所在
    p->tss.ecx = ecx;
    p->tss.edx = edx;
    p->tss.ebx = ebx;
    p->tss.esp = esp;
    p->tss.ebp = ebp;
    p->tss.esi = esi;
    p->tss.edi = edi;
    p->tss.es = es & 0xffff;
    p->tss.cs = cs & 0xffff;
    p->tss.ss = ss & 0xffff;
    p->tss.ds = ds & 0xffff;
    p->tss.fs = fs & 0xffff;
    p->tss.gs = gs & 0xffff;
    p->tss.ldt = LDT(nr); // 任务局部表描述符的选择符(LDT描述符在GDT中)实现内存共享
    p->tss.trace_bitmap = 0x80000000; // 高16位有效

    if (last_task_used_math == current)
        asm ("clts ; fnsave %0::"m" (p->tss.i387));
    // 接下来复制进程页表。即在线性地址空间中设置新任务代码段和数据段描述符中的基址和限长。
    // 并复制页表。如果出错(返回值不是0), 则复位任务数组中相应项并释放为该新任务分配的用于
    // 任务结构的内存页。
    if (copy_mem(nr, p)) {
        task[nr] = NULL;
        free_page((long) p);
        return -EAGAIN;
    }
    // 如果父进程中有文件是打开的, 则将对对应文件的打开次数增1, 因为这里创建的子进程会与父
    // 进程共享这些打开的文件。实现共享文件
    for (i=0; i<NR_OPEN; i++)
        if ((f=p->filp[i]))
            f->f_count++;
    if (current->puid)
        current->puid->i_count++;
    if (current->root)
        current->root->i_count++;
    if (current->executable)
        current->executable->i_count++;
    // 随后GDT表中设置新任务TSS段和LDT段描述符项。把新进程设置成就绪态。另外在任务切换时, 任务寄存器tr由
    // CPU自动加载。最后返回新进程号。
    set_tss_desc(gdt+(nr<<1)+FIRST_TSS_ENTRY, &(p->tss));
    set_ldt_desc(gdt+(nr<<1)+FIRST_LDT_ENTRY, &(p->ldt));
    p->state = TASK_RUNNING; /* do this last, just in case */
    return last_pid;
} // end copy_process »

// 为新进程取得不重复的进程号last_pid。函数返回在任务数组中的任务号(数组项)。

```

3.7.6 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子进程创建唯一的 PCB

3.7.7 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

3.7.8 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}
```

3.7.9 阻塞进程 wait()

进程调用 wait(int &status) 阻塞自己。Status 接收子进程退出时的退出代码（按位处理）。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。

若忽略子进程的退出信息。pid = wait(NULL)。

```

// 如果pid > 0, 表示等待进程号等于pid的子进程。
// 如果pid = 0, 表示等待进程组号等于当前进程组号的任何子进程。
// 如果pid < -1,表示等待进程组号等于pid绝对值的任何子进程。
// 如果pid = -1,表示等待任何子进程。
// 如 options = WUNTRACED,表示如果子进程是停止的,也马上返回(无须跟踪)
// 若 options = WNOHANG,表示如果没有子进程退出或终止就马上返回。
// 如果返回状态指针 stat_addr不为空,则就将状态信息保存到那里。
// 参数pid是进程号,*stat_addr是保存状态信息位置的指针,options是waitpid选项。
int sys_waitpid(pid_t pid,unsigned long * stat_addr, int options)
{
    int flag, code;           // flag标志用于后面表示所选出的子进程处于就绪或睡眠态。
    struct task_struct ** p;

    verify_area(stat_addr,4);
repeat:
    flag=0;
    // 从任务数组末端开始扫描所有任务,跳过空项、本进程项以及非当前进程的子进程项。
    for(p = &LAST_TASK ; p > &FIRST_TASK ; --p) {
        if (!*p || *p == current)
            continue;
        if ((*p)->father != current->pid)
            continue;
        // 此时扫描选择到的进程p肯定是当前进程的子进程。
        // 如果等待的子进程号pid>0,但与扫描子进程p的pid不相等,说明它是当前进程另外的
        // 子进程,于是跳过该进程,接着扫描下一个进程。
        if (pid>0) {
            if ((*p)->pid != pid)
                continue;
            // 否则,如果指定等待进程的pid=0,表示正在等待进程组号等于当前进程组号的任何子进程。
            // 如果此时扫描子进程p的进程组号与当前进程的组号不等,则跳过。
        } else if (!pid) {
            if ((*p)->pgrp != current->pgrp)
                continue;
            // 否则,如果指定的pid < -1,表示正在等待进程组号等于pid绝对值的任何子进程。如果此时
            // 扫描子进程p的组号与pid的绝对值不等,则跳过。
        } else if (pid != -1) {
            if ((*p)->pgrp != -pid)
                continue;
        }
        // 如果前3个对pid的判断都不符合,则表示当前进程正在等待其任何子进程,也即pid=-1的情况,
        // 此时所选择到的进程p或者是其进程号等于指定pid,或者是当前进程组中的任何子进程,或者
        // 是进程号等于指定pid绝对值的子进程,或者是任何子进程(此时指定的pid等于-1)。接下来根据
        // 这个子进程p所处的状态来处理。
        switch ((*p)->state) {
            // 子进程p处于停止状态时,如果此时WUNTRACED标志没有置位,表示程序无须立刻返回,于是
            // 继续扫描处理其他进程。如果WUNTRACED置位,则把状态信息0x7f放入*stat_addr,并立刻
            // 返回子进程号pid,这里0x7f表示的返回状态是wifstopped()宏为真。
            case TASK_STOPPED:
                if (!(options & WUNTRACED))
                    continue;
                put_fs_long(0x7f,stat_addr);
                return (*p)->pid;
            // 如果子进程p处于僵死状态,则首先把它在用户态和内核态运行的时间分别累计到当前进程
            // (父进程)中,然后取出子进程的pid和退出码,并释放该子进程。最后返回子进程的退出码和pid。
            case TASK_ZOMBIE:
                current->cutime += (*p)->utime;
                current->cstime += (*p)->stime;
                flag = (*p)->pid;           // 临时保存子进程pid
                code = (*p)->exit_code;     // 取子进程的退出码
                release(*p);               // 释放该子进程
                put_fs_long(code,stat_addr); // 置状态信息为退出码值
                return flag;               // 返回子进程的pid
            // 如果这个子进程p的状态既不是停止也不是僵死,那么就置flag=1,表示找到过一个符合
            // 要求的子进程,但是它处于运行态或睡眠态。
            default:
                flag=1;
                continue;
        }
    }
    // 上面扫描任务数组结束后,如果flag被置位,说明有符合等待要求的子进程并没有处于退出或
    // 僵死状态。如果此时已设置WNOHANG选项(表示若没有子进程处于退出或终止状态就立刻返回),就
    // 立刻返回0,退出。否则把当前进程置为可中断等待状态并重新执行调度。当又开始执行本进程时,
    // 如果本进程没有收到除SIGCHLD以外的信号,则还是重复处理。否则,返回出错码‘中断系统调用’
    // 并退出。针对这个出错码用户程序应该再继续调用本函数等待子进程。
    if (flag) {
        if (options & WNOHANG)           // options = WNOHANG,则立刻返回。
            return 0;
        current->state=TASK_INTERRUPTIBLE; // 置当前进程为可中断等待态
        schedule();                       // 重新调度。
        if (!(current->signal &= ~(1<<(SIGCHLD-1))))
            goto repeat;
        else
            return -EINTR;                // 返回出错码(中断的系统调用)
    }
    // 若没有找到符合要求的子进程,则返回出错码(子进程不存在)。
    return -ECHILD;
}
// end sys_waitpid

```

3.7.10 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。

```

int i;
// 首先释放当前进程代码段和数据段所占的内存页。函数free_page_tables()的第一个参数
// (get_base()返回值)指明在CPU线性地址空间中起始基地址，第2个(get_limit()返回值)
// 说明欲释放的字节长度值。get_base()宏中的current->ldt[1]给出进程代码段描述符的
// 位置(current->ldt[2]给出进程代码段描述符的位置)；get_limit()中0x0f是进程代码段
// 的选择符(0x17是进程数据段的选择符)。即在取段基地址时使用该段的描述符所地址作为
// 参数，取段长度时使用该段的选择符作为参数。free_page_tables()函数位于mm/memory.c
// 文件中。
free_page_tables(get_base(current->ldt[1]),get_limit(0x0f));
free_page_tables(get_base(current->ldt[2]),get_limit(0x17));
// 如果当前进程有子进程，就将子进程的father置为1(其父进程改为进程1，即init进程)。
// 如果该子进程已经处于僵死(ZOMBIE)状态，则向进程1发送子进程中止信号SIGCHLD。
for (i=0 ; i<NR_TASKS ; i++)
    if (task[i] && task[i]->father == current->pid) {
        task[i]->father = 1;
        if (task[i]->state == TASK_ZOMBIE)
            /* assumption task[i] is always init */
            (void) send_sig(SIGCHLD, task[i], 1);
    }
// 关闭当前进程打开着的所有文件。
for (i=0 ; i<NR_OPEN ; i++)
    if (current->filp[i])
        sys_close(i);
// 对当前进程的工作目录pwd，根目录root以及执行程序文件的i节点进行同步操作，放回
// 各个i节点并分别置空(释放)。
input(current->pwd);
current->pwd=NULL;
input(current->root);
current->root=NULL;
input(current->executetable);
current->executable=NULL;
// 如果当前进程是会话头领(leader)进程并且其有控制终端，则释放该终端。
if (current->leader && current->tty >= 0)
    tty_table[current->tty].pgpr = 0;
// 如果当前进程上次使用过协处理器，则将last_task_used_math置空。
if (last_task_used_math == current)
    last_task_used_math = NULL;
// 如果当前进程是leader进程，则终止该会话的所有相关进程。
if (current->leader)
    kill_session();
// 把当前进程置为僵死状态，表明当前进程已经释放了资源，并保存将由父进程读取的退出码。
current->state = TASK_ZOMBIE;
current->exit_code = code;
// 通知父进程，也即向父进程发送信号SIGCHLD - 子进程将停止或终止。
tell_father(current->father);
schedule(); // 重新调度进程运行，以让父进程处理僵死其他的善后事宜。
// 下面的return语句仅用于去掉警告信息，因为这个函数不返回，所以在函数名前加关键字
// volatile，就可以告诉gcc编译器本函数不会返回的特殊情况。这样可让gcc产生更好一些的代码，
// 并且可以不用再写return语句也不会产生警告信息。
return (-1); /* just to suppress warnings */
} // end do_exit »

//// 系统调用exit()，终止进程。
} // end do_exit »

```

3.7.11 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 8: 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。

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。

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。

4.2 Windows 创建一个线程：CreateThread()

功能：把一个函数创建一个线程。

```

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
);
```

4.2.1 创建远程线程: CreateRemoteThread()

在另外一个指定的进程空间创建线程。起到监视，木马等作用。

4.3 Linux 线程

Linux 中实现现成的即为 task_struct 结构体，Linux 中线程实际上是进程。

当我们使用 ps 命令时，COMMAND 一列被方括号括住的实际上表达的是线程。可以通过查看其父进程，判断其主进程。

4.3.1 内核线程 (Kernel Thread)

创建，运行，撤销均在内核中实现，由内核驱动。

mm = NULL。其没有独立的地址空间，因为其旨在内核控件中运行，不切换到用户空间。

由 kthread_create() 函数创建。

在编写驱动程序时有着重要的作用。

4.3.2 轻量级进程 (Light Weight Process, LWP)

代表着进程的一个执行路径，其共享进程中的地址空间和其他资源，但可以被独立调度。

与线程相比，LWP 拥有进程标志符，程序计数器，优先级，状态，栈和局部存储区。

由 clone() 函数创建。(fork(), vfork(), clone() 函数均是调用了 do_fork() 函数，其中比较关键的参数为 clone_flags)

4.3.3 用户线程 (User Thread)

通过 pthread 线程库创建和管理，线程库提供同步和调度的方法。

用户线程不是一个真正的调度实体，内核对他们一无所知。当一个进程被抢占时，它的所有用户进程都被抢占了。

通过 `pthread_create()` 函数创建。

4.4 使用线程的一些问题

1. 程序难以调试。
2. 线程安全问题。
3. 并发过程难以控制。

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**，必须协调各进程对资源的存取顺序：确保没有任何两个或以上的进程同时进行资源存取。

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

5.1.2 临界区 (Critical Section)

进程中访问临界资源的程序段。

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

功能：有条件的继续或停止进程。

6.1 硬件方法

6.1.1 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。

6.1.2 测试并设置指令 (Test and Set)

6.1.3 交换指令

6.2 锁机制

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。

6.2.1 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

6.2.2 开锁原语

第 1 步：把锁 S 的状态置 1

6.2.3 设计临界区访问机制的四个原则

忙则等待，空闲让进，有限等待，让权等待（等待进程放弃 CPU）。

6.3 信号灯与 P-V 操作

进程在运行过程中收信号灯的控制，并能改变信号灯。这是一个双向的影响。

6.3.1 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。

6.3.2 P 操作 通过

P 操作可能使进程在调用处阻塞。

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

图 9: P 操作的伪代码

6.3.3 V 操作 释放

P 操作可能使进程在调用处唤醒。

6.3.4 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。

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

图 10: V 操作的伪代码

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。

6.3.5 使用信号灯 P-V 操作的实现进程同步

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（一般是 0）。不合理的初值不仅达不到同步的目的，还会发生死锁。

6.4 经典的同步问题

6.4.1 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