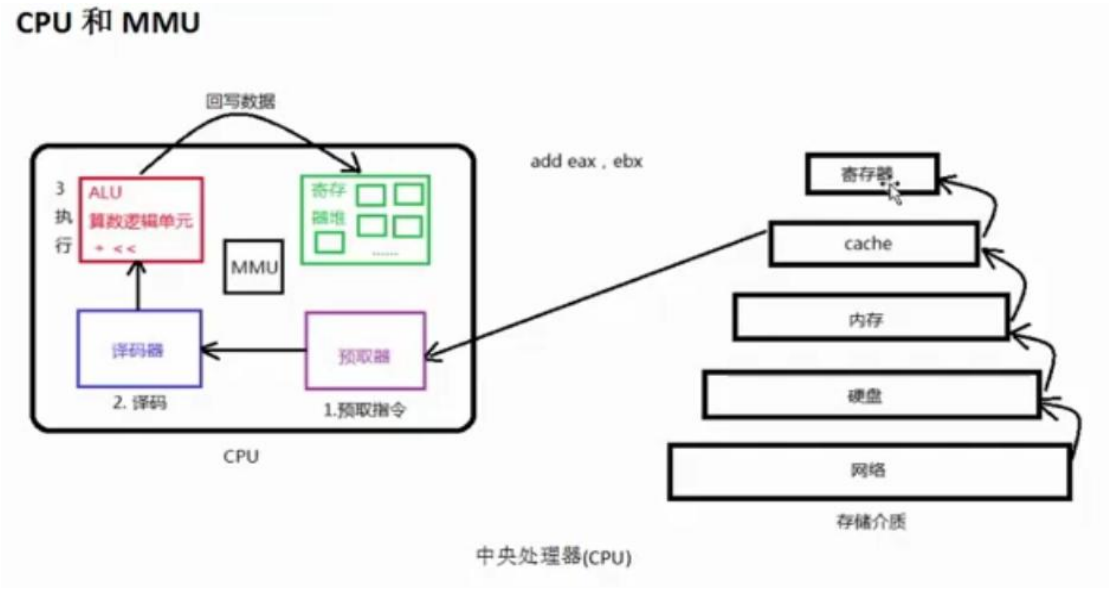
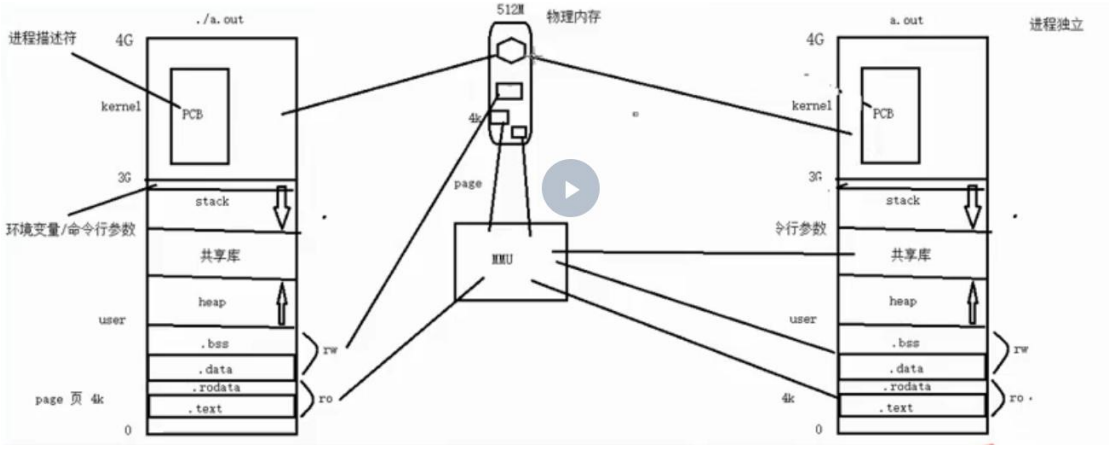
【进程 1】

CPU



MMU 存在于 CPU 中，负责虚拟内存于物理内存的映射与修改内存访问级别



不同进程（即使同一个程序，分别运行两次也属于不同进程，进程独立），父子进程映射时的内核空间一样，其他都不一样。每运行一个程序都会产生一个 0-4G 可寻址的虚拟内存空间，0-3G 相同，4G 不同。在 fork 之后两个进程用的是相同的物理空间（内存区），**两者的虚拟空间不同，但其对应的物理空间是同一个。（读时共享写时复制）**

PCB 进程控制块（进程描述符）

/usr/src/linux-headers-3.16.0-30/include/linux/sched.h 文件中可以查看 struct task_struct 结构体定义。其内部成员有很多，我们重点掌握以下部分即可：

- 进程 id。系统中每个进程有唯一的 id，在 C 语言中用 pid_t 类型表示，其实就是一个非负整数。
- 进程的状态，有就绪、运行、挂起、停止等状态。
- 进程切换时需要保存和恢复的一些 CPU 寄存器。
- 描述虚拟地址空间的信息。
- 描述控制终端的信息。
- 当前工作目录（Current Working Directory）。
- umask 掩码。
- 文件描述符表，包含很多指向 file 结构体的指针。
- 和信号相关的信息。
- 用户 id 和组 id。
- 会话（Session）和进程组。
- 进程可以使用的资源上限（Resource Limit）。

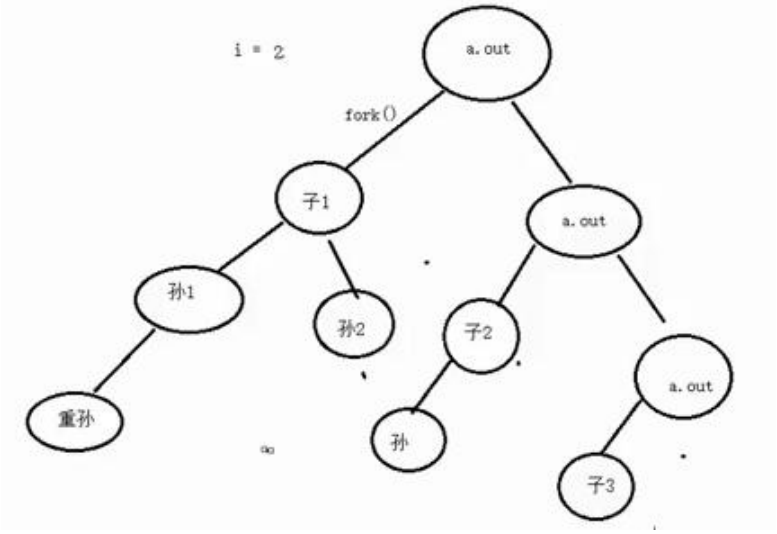
fork函数

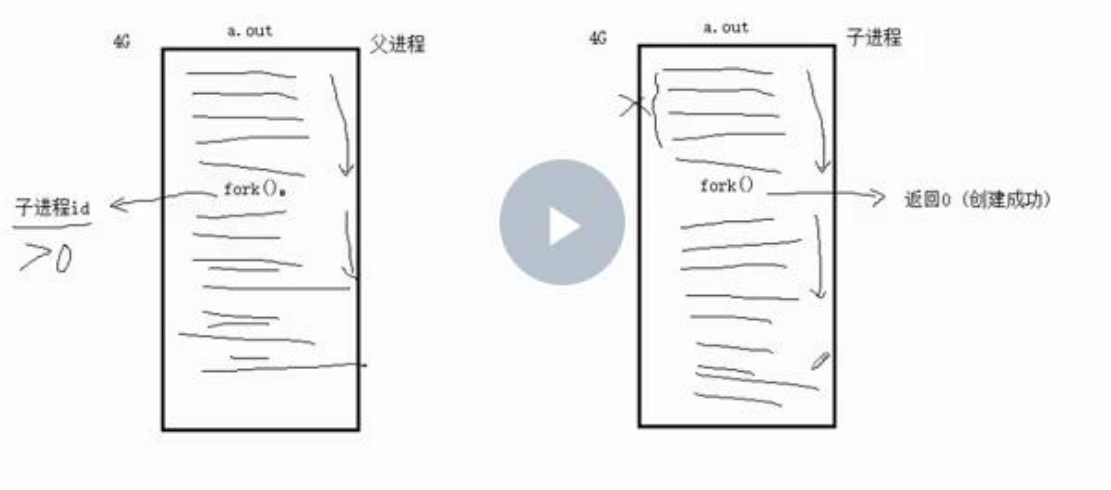
返回值有2个： 一个进程--> 2个进程 ---> 各自对fork做返回.

1. 返回子进程的pid（非负整数 > 0） （父进程）
2. 返回0 （子进程）

循环创建子进程的架构.

循环创建陷阱





子进程会接着执行在 fork 后面的代码，父进程也接着执行后面的代码

进程共享

父子进程之间在 fork 后。有哪些相同，那些相异之处呢？

刚 fork 之后：

父子相同处：全局变量、.data、.text、栈、堆、环境变量、用户 ID、宿主目录、进程工作目录、信号处理方式...

父子不同处：1.进程 ID　2.fork 返回值　3.父进程 ID　4.进程运行时间　5.闹钟(定时器)　6.未决信号集

似乎，子进程复制了父进程 0-3G 用户空间内容，以及父进程的 PCB，但 pid 不同。真的每 fork 一个子进程都要将父进程的 0-3G 地址空间完全拷贝一份，然后在映射至物理内存吗？

当然不是!父子进程间遵循**读时共享写时复制**的原则。这样设计，无论子进程执行父进程的逻辑还是执行自己的逻辑都能节省内存开销。

练习：编写程序测试，父子进程是否共享全局变。　【fork_shared.c】

重点注意！躲避父子进程共享全局变量的知识误区！

【进程 2】

fork 创建子进程后执行的是和父进程相同的程序（但有可能执行不同的代码分支），子进程往往要调用一种 exec 函数以执行另一个程序。当进程调用一种 exec 函数时，该进程的用户空间代码和数据完全被新程序替换，从新程序的启动例程开始执行。调用 exec 并不创建新进程，所以调用 exec 前后该进程的 id 并未改变。

将当前进程的.text、.data 替换为所要加载的程序的.text、.data，然后让进程从新的.text 第一条指令开始执行，但进程 ID 不变，换核不换壳。

其实有六种以 exec 开头的函数，统称 exec 函数：

```
int execl(const char *path, const char *arg, ...);

int execlp(const char *file, const char *arg, ...);

int execle(const char *path, const char *arg, ..., char *const envp[]);

int execv(const char *path, char *const argv[]);

int execvp(const char *file, char *const argv[]);

int execve(const char *path, char *const argv[], char *const envp[]);
```

execlp 函数

加载一个进程，借助 PATH 环境变量

int execlp(const char *file, const char *arg, ...);　　成功：无返回；失败：-1

参数 1：要加载的程序的名字。该函数需要配合 PATH 环境变量来使用，当 PATH 中所有目录搜索后没有参数 1 则出错返回。

该函数通常用来调用系统程序。如：ls、date、cp、cat 等命令。

execl 函数

加载一个进程，　通过 路径+程序名 来加载。

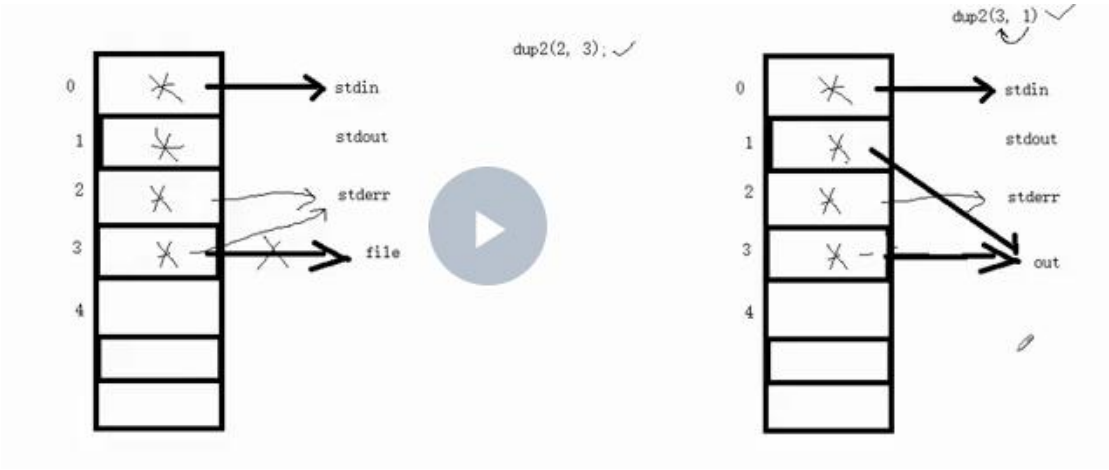
int execl(const char *path, const char *arg, ...);　　成功：无返回；失败：-1

对比 execlp，如加载"ls"命令带有-l，-F 参数

execlp("ls", "ls", "-l", "-F", NULL);　　使用程序名在 PATH 中搜索。

execl("/bin/ls", "ls", "-l", "-F", NULL);　　使用参数 1 给出的绝对路径搜索。

dup2 函数，重定向文件描述符：



文件描述符	通道名	描述	默认连接	用途
0	stdin	标准输入	键盘	read only
1	stdout	标准输出	终端	write only
2	stderr	标准错误	终端	write only
3以上	filename	其他文件	none	read and/or write

stdin，标准输入，默认设备是键盘，文件编号为 0

stdout，标准输出，默认设备是显示器，文件编号为 1，也可以重定向到文件

stderr，标准错误，默认设备是显示器，文件编号为 2，也可以重定向到文件

wait 函数

一个进程在终止时会关闭所有文件描述符，释放在用户空间分配的内存，但它的 PCB 还保留着，内核在其中保存了一些信息：如果是正常终止则保存着退出状态，如果是异常终止则保存着导致该进程终止的信号是哪个。这个进程的父进程可以调用 wait 或 waitpid 获取这些信息，然后彻底清除掉这个进程。我们知道一个进程的退出状态可以在 Shell 中用特殊变量$?查看，因为 Shell 是它的父进程，当它终止时 Shell 调用 wait 或 waitpid 得到它的退出状态同时彻底清除掉这个进程。

父进程调用 wait 函数可以回收 [宋体 (正文) 五号] 个功能：

① 阻塞等待子进程退出

② 回收子进程残留资源

③ 获取子进程结束状态(退出原因)。

pid_t wait(int *status); 成功：清理掉的子进程 ID；失败：-1 (没有子进程)

当进程终止时，操作系统的隐式回收机制会：1.关闭所有文件描述符 2. 释放用户空间分配的内存。内核的 PCB 仍存在。其中保存该进程的退出状态。(正常终止→退出值；异常终止→终止信号)

可使用 wait 函数传出参数 status 来保存进程的退出状态。借助宏函数来进一步判断进程终止的具体原因。宏函数可分为如下三组：

1. WIFEXITED(status) 为非 0 → 进程正常结束

   WEXITSTATUS(status) 如上宏为真，使用此宏 → 获取进程退出状态 (exit 的参数)

2. WIFSIGNALED(status) 为非 0 → 进程异常终止

   WTERMSIG(status) 如上宏为真，使用此宏 → 取得使进程终止的那个信号的编号。

所有进程异常终止，都是因为信号， wait 函数为阻塞态，wait 缺陷：不能指定某一死亡的子进程。

waitpid 函数

作用同 wait，但可指定 pid 进程清理，可以不阻塞。

pid_t waitpid(pid_t pid, int *status, in options); 成功：返回清理掉的子进程 ID；失败：-1(无子进程)

特殊参数和返回情况：

参数 pid：

>0 回收指定 ID 的子进程

-1 回收任意子进程（相当于 wait）

0 回收和当前调用 waitpid 一个组的所有子进程

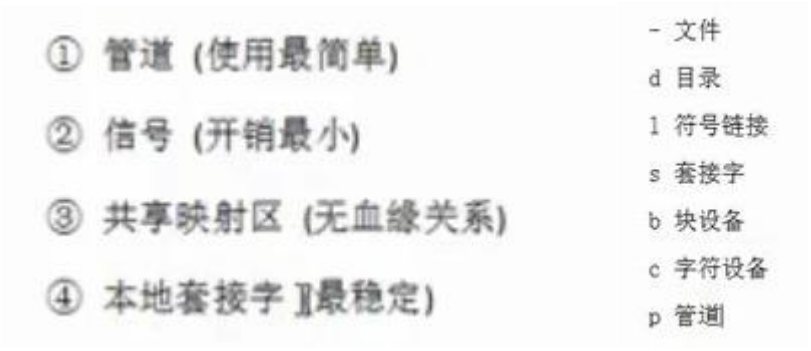
< -1 回收指定进程组内的任意子进程

返回 0：参 3 为 WNOHANG，且子进程正在运行。

注意：**一次 wait 或 waitpid 调用只能清理一个子进程**，清理多个子进程应使用循环。

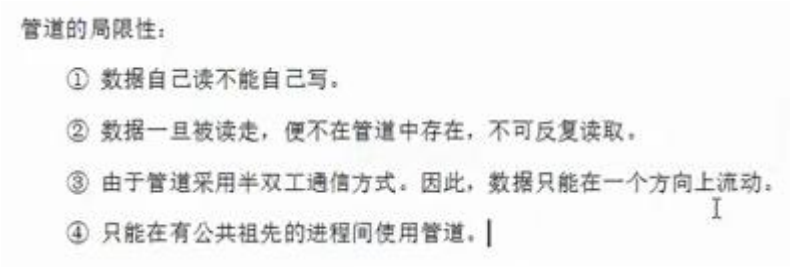
waitpid 非阻塞：第三个参数为 WNOHANG，ps –ajx，进程组 pgid

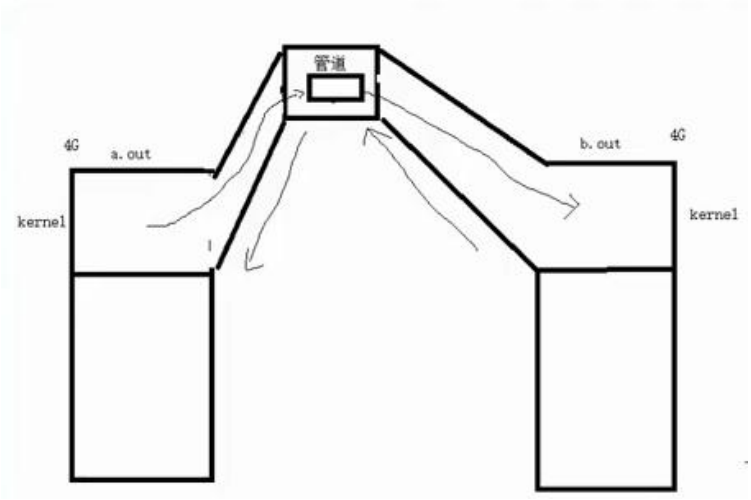
【进程 3：IPC 通信】

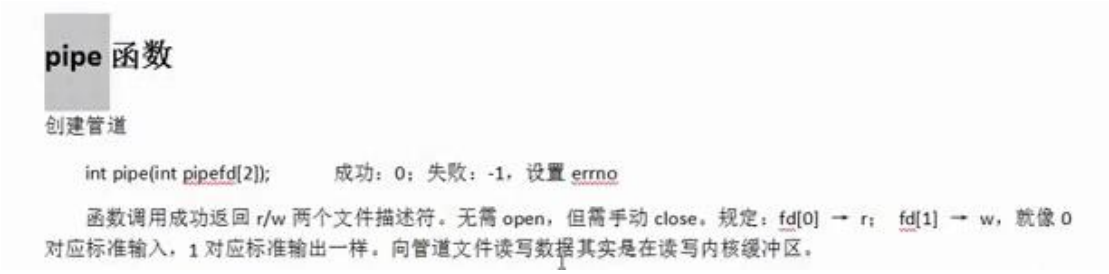


（文件类型）

fifo：又名命名管道，用于非血缘关系的进程件通信，pipe 管道：父子进程通信







【共享内存：mmap】

```
void *mmap(void *adrr, size_t length, int prot, int flags, int fd, off_t offset);
```

返回：成功：返回创建的映射区首地址；**失败：MAP_FAILED 宏**

参数：

addr：建立映射区的首地址，由 Linux 内核指定。使用时，直接传递 NULL

length：欲创建映射区的大小

prot：映射区权限 PROT_READ、PROT_WRITE、PROT_READ|PROT_WRITE

flags：标志位参数(常用于设定更新物理区域、设置共享、创建匿名映射区)

MAP_SHARED：会将映射区所做的操作反映到物理设备（磁盘）上。

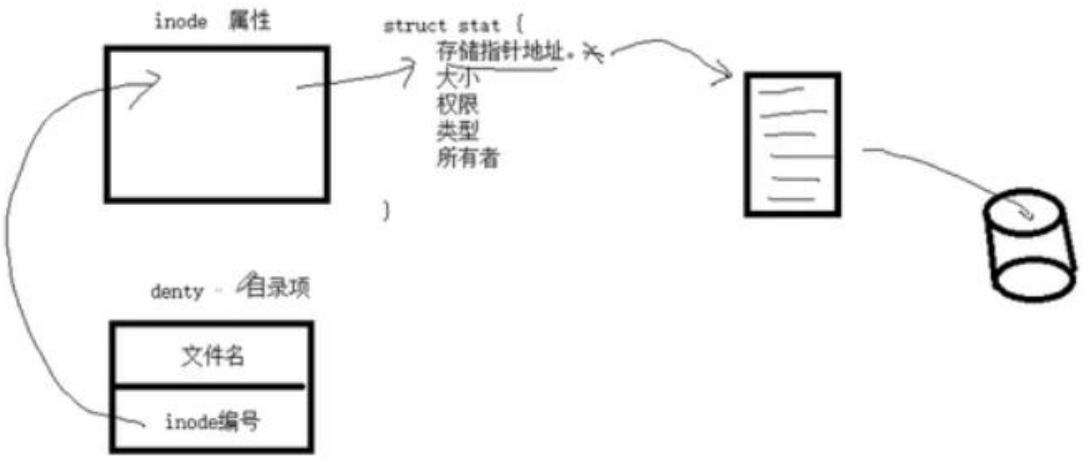
MAP_PRIVATE：映射区所做的修改不会反映到物理设备。

fd：用来建立映射区的文件描述符

offset：映射文件的偏移(4k 的整数倍)

offset 最好是 4k 的整数倍，因为 mmu 内存映射的最小单元就是 4k

一个文件包含：（inode-文件属性和 denty-文件目录项）



mmap 父子进程通信

父子等有血缘关系的进程之间也可以通过 mmap 建立的映射区来完成数据通信。但相应的要在创建映射区的时候指定对应的标志位参数 flags：

MAP_PRIVATE: (私有映射) 父子进程各自独占映射区；

MAP_SHARED: (共享映射) 父子进程共享映射区；

练习：父进程创建映射区，然后 fork 子进程，子进程修改映射区内容，而后，父进程读取映射区内容，查验是否共享。 【fork_mmap.c】

结论：**父子进程共享：1. 打开的文件 2. mmap 建立的映射区(但必须要使用 MAP_SHARED)**

非匿名映射必须借助 ftruncate 函数预置文件大小。

匿名映射

通过使用我们发现，使用映射区来完成文件读写操作十分方便，父子进程间通信也较容易。但缺陷是，每次创建映射区一定要依赖一个文件才能实现。通常为了建立映射区要 open 一个 temp 文件，创建好了再 unlink、close 掉，比较麻烦。 可以直接使用匿名映射来代替。其实 Linux 系统给我们提供了创建匿名映射区的方法，无需依赖一个文件即可创建映射区。同样需要借助标志位参数 flags 来指定。

使用 MAP_ANONYMOUS (或 MAP_ANON)， 如:

```
int *p = mmap(NULL, 4, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
```

"4"随意举例，该位置表大小，可依实际需要填写。

需注意的是，MAP_ANONYMOUS 和 MAP_ANON 这两个宏是 Linux 操作系统特有的宏。在类 Unix 系统中如无该宏定义，可使用如下两步来完成匿名映射区的建立。

① fd = open("/dev/zero", O_RDWR);

② p = mmap(NULL, size, PROT_READ|PROT_WRITE, MMAP_SHARED, fd, 0);

/dev/zero 吐无限大小的文件，/dev/null 吞无限多的文件

【信号】

信号的机制

A 给 B 发送信号，B 收到信号之前执行自己的代码，收到信号后，不管执行到程序的什么位置，都要暂停运行，去处理信号，处理完毕再继续执行。与硬件中断类似——异步模式。但信号是软件层面上实现的中断，早期常被称为“软中断”。

信号的特质：由于信号是通过软件方法实现，其实现手段导致信号有很强的延时性。但对于用户来说，这个延迟时间非常短，不易察觉。

每个进程收到的所有信号，都是由内核负责发送的，内核处理。

与信号相关的事件和状态

产生信号：

1. 按键产生，如：Ctrl+c、Ctrl+z、Ctrl+\
2. 系统调用产生，如：kill、raise、abort
3. 软件条件产生，如：定时器 alarm
4. 硬件异常产生，如：非法访问内存(段错误)、除 0(浮点数例外)、内存对齐出错(总线错误)
5. 命令产生，如：kill 命令

kill –l 系统支持的所有信号（1-31 普通信号（集），32-实时信号），man 7 signal（信号描述）

产生信号 5 种：

阻塞

解除 递达 处理： 1. 默认动作 --- 1） 终止进程

2） 终止进程 且 core文件

3） 忽略

4） 暂停 stop

5） 继续

2. 忽略处理

3. 捕捉

终端按键产生信号

Ctrl + c → 2) SIGINT（终止/中断） "INT" ----Interrupt

Ctrl + z → 20) SIGTSTP（暂停/停止） "T" ----Terminal 终端。

Ctrl + \ → 3) SIGQUIT（退出）

硬件异常产生信号

除 0 操作 → 8) SIGFPE (浮点数例外) "F" -----float 浮点数。

非法访问内存 → 11) SIGSEGV (段错误)

总线错误 → 7) SIGBUS

kill 命令产生信号：kill -SIGKILL pid

kill 函数：给指定进程发送指定信号(不一定杀死)

int kill(pid_t pid, int sig); 成功：0；失败：-1 (ID 非法，信号非法，普通用户杀 init 进程等权级问题)，设置 errno

sig：不推荐直接使用数字，应使用宏名，因为不同操作系统信号编号可能不同，但名称一致。

pid > 0: 发送信号给指定的进程。

pid = 0: 发送信号给 与调用 kill 函数进程属于同一进程组的所有进程。

pid < 0: 取|pid|发给对应进程组。

pid = -1：发送给进程有权限发送的系统中所有进程。

进程组：每个进程都属于一个进程组，进程组是一个或多个进程集合，他们相互关联，共同完成一个实体任务，每个进程组都有一个进程组长，默认进程组 ID 与进程组长 ID 相同。

权限保护：super 用户(root)可以发送信号给任意用户，普通用户是不能向系统用户发送信号的。 kill -9 (root 用户的 pid) 是不可以的。同样，普通用户也不能向其他普通用户发送信号，终止其进程。 只能向自己创建的进程发送信号。普通用户基本规则是：发送者实际或有效用户 ID == 接收者实际或有效用户 ID

kill -9 -pid //杀死进程组，出现提示已杀死，ps ajx 查看进程组

raise 和 abort 函数

raise 函数：给当前进程发送指定信号(自己给自己发) raise(signo) =kill(getpid(), signo);

int raise(int sig); 成功：0，失败非 0 值

abort 函数：给自己发送异常终止信号 6) SIGABRT 信号，终止并产生 core 文件

void abort(void); 该函数无返回

从调用 abort 函数处结束进程；raise 函数发送任意信号给本进程。

alarm 函数：发送 SIGLARM 信号

函数的返回值，之前闹钟定时剩余时间，alarm（0）取消定时器，在哪个进程设置的定时器，信号就发给哪个进程。

setitimer 函数

设置定时器(闹钟)。 可代替 alarm 函数。精度微秒 us，可以实现周期定时。

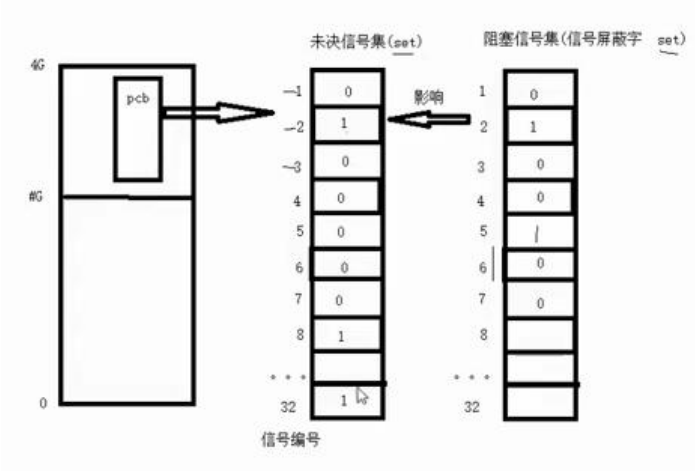
int setitimer(int which, const struct itimerval *new_value, struct itimerval *old_value); 成功：0；失败：-1，设置 errno

参数：which：指定定时方式

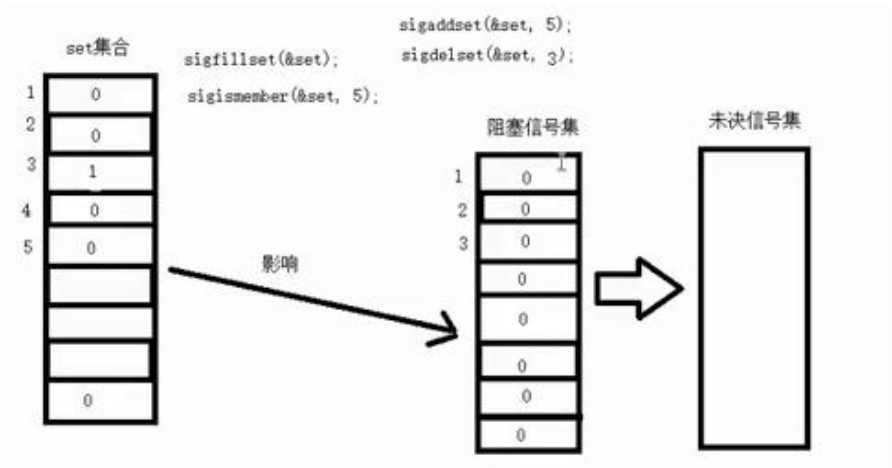
① 自然定时：ITIMER_REAL → 14）SIGLARM 计算自然时间

② 虚拟空间计时(用户空间)：ITIMER_VIRTUAL → 26）SIGVTALRM 只计算进程占用 cpu 的时间

③ 运行时计时(用户+内核)：ITIMER_PROF → 27）SIGPROF 计算占用 cpu 及执行系统调用的时间



未决信号集：待处理的信号；阻塞信号集：写 1 可以屏蔽住未决信号集。



用自定义的信号集取影响阻塞信号集，进而影响未决信号

信号集设定

sigset_t set; // typedef unsigned long sigset_t;

int sigemptyset(sigset_t *set); 将某个信号集清 0 成功：0；失败：-1

int sigfillset(sigset_t *set); 将某个信号集置 1 成功：0；失败：-1

int sigaddset(sigset_t *set, int signum); 将某个信号加入信号集 成功：0；失败：-1

int sigdelset(sigset_t *set, int signum); 将某个信号清出信号集 成功：0；失败：-1

int sigismember(const sigset_t *set, int signum);判断某个信号是否在信号集中 返回值：在集合：1；不在：0；出错：-1

sigset_t 类型的本质是位图。但不应该直接使用位操作，而应该使用上述函数，保证跨系统操作有效。

对比认知 select 函数。

sigprocmask 函数

用来屏蔽信号、解除屏蔽也使用该函数。其本质，读取或修改进程的信号屏蔽字(PCB 中)

严格注意，屏蔽信号：只是将信号处理延后执行(延至解除屏蔽)；而忽略表示将信号丢处理。

int sigprocmask(int how, const sigset_t *set, sigset_t *oldset); 成功：0；失败：-1，设置 errno

参数：

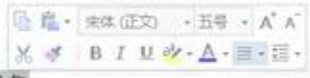
set：传入参数，是一个位图，set 中哪位置 1，就表示当前进程屏蔽哪个信号。

oldset：传出参数，保存旧的信号屏蔽集。

how 参数取值： 假设当前的信号屏蔽字为 mask

1. SIG_BLOCK: 当 how 设置为此值，set 表示需要屏蔽的信号。相当于 mask = mask|set
2. SIG_UNBLOCK: 当 how 设置为此，set 表示需要解除屏蔽的信号。相当于 mask = mask & ~set
3. SIG_SETMASK: 当 how 设置为此，set 表示用于替代原始屏蔽及的新屏蔽集。相当于 mask = set 若，调用 sigprocmask 解除了对当前若干个信号的阻塞，则在 sigprocmask 返回前，至少将其中一个信号递达。

sigpending 函数



读取当前进程的未决信号集

int sigpending(sigset_t *set); set 传出参数。 返回值：成功：0；失败：-1，设置 errno

SIGKILL 信号无法被屏蔽的！

sigaction 函数捕捉信号：

struct sigaction 结构体

```
struct sigaction {
    void     (*sa_handler)(int);
    void     (*sa_sigaction)(int, siginfo_t *, void *);
    sigset_t   sa_mask;
    int        sa_flags;
    void     (*sa_restorer)(void);
};
```

sa_restorer：该元素是过时的，不应该使用，POSIX.1 标准将不指定该元素。(弃用)

sa_sigaction：当 sa_flags 被指定为 SA_SIGINFO 标志时，使用该信号处理程序。(很少使用)

重点掌握：

① sa_handler：指定信号捕捉后的处理函数名(即注册函数)。也可赋值为 SIG_IGN 表忽略 或 SIG_DFL 表执行默认动作

② sa_mask：调用信号处理函数时，所要屏蔽的信号集合(信号屏蔽字)。注意：仅在处理函数被调用期间屏蔽生效，是临时性设置。

③ sa_flags：通常设置为 0，表使用默认属性。

sa_mask：表示在执行处理函数时，不被哪些信号中断。只在处理函数执行期间有效!!!

sa_flags：=0 表示在此信号的处理函数执行期间自动屏蔽再一次产生的本信号!!!，即使产生多次该信号，也只执行一次（下面 2、3 点）

信号捕捉特性

1. 进程正常运行时，默认 PCB 中有一个信号屏蔽字，假定为☆，它决定了进程自动屏蔽哪些信号。当注册了某个信号捕捉函数，捕捉到该信号以后，要调用该函数。而该函数有可能执行很长时间，在这期间所屏蔽的信号不由☆来指定。而是用 sa_mask 来指定。调用完信号处理函数，再恢复为☆。
2. XXX 信号捕捉函数执行期间，XXX 信号自动被屏蔽。
3. 阻塞的常规信号不支持排队，产生多次只记录一次。（后 32 个实时信号支持排队）

竞态条件(时序竞态)：

pause 函数

调用该函数可以造成进程主动挂起，等待信号唤醒。调用该系统调用的进程将处于阻塞状态(主动放弃 cpu) 直到有信号递达将其唤醒。

int pause(void);　　返回值：-1 并设置 errno 为 EINTR

返回值：

① 如果信号的默认处理动作是终止进程，则进程终止，pause 函数么有机会返回。

② 如果信号的默认处理动作是忽略，进程继续处于挂起状态，pause 函数不返回。

③ 如果信号的处理动作是捕捉，则【调用完信号处理函数之后，pause 返回-1】

errno 设置为 EINTR，表示“被信号中断”。想想我们还有哪个函数只有出错返回值。

④ pause 收到的信号不能被屏蔽，如果被屏蔽，那么 pause 就不能被唤醒。

解决时序问题

可以通过设置屏蔽 SIGALRM 的方法来控制程序执行逻辑，但无论如何设置，程序都有可能在“解除信号屏蔽”与“挂起等待信号”这个两个操作间隙失去 cpu 资源。除非将这两步骤合并成一个“原子操作”。sigsuspend 函数具备这个功能。在对时序要求严格的场合下都应该使用 sigsuspend 替换 pause。

int sigsuspend(const sigset_t *mask); 挂起等待信号。

sigsuspend 函数调用期间，进程信号屏蔽字由其参数 mask 指定。

可将某个信号（如 SIGALRM）从临时信号屏蔽字 mask 中删除，这样在调用 sigsuspend 时将解除对该信号的屏蔽，然后挂起等待，当 sigsuspend 返回时，进程的信号屏蔽字恢复为原来的值。如果原来对该信号是屏蔽态，sigsuspend 函数返回后仍然屏蔽该信号。

原子操作函数：cpu 一旦调用该函数，就会直到函数调用完才释放 cpu 使用权。
可以避免在 alarm 与挂起函数之间失去 cpu，从而造成 alarm 定时结束了，挂起函数才获得 cpu 开始执行，从而无法等到 alarm 定时结束产生的信号。

信号传参

发送信号传参

sigqueue 函数对应 kill 函数，但可在向指定进程发送信号的同时携带参数

int sigqueue(pid_t pid, int sig, const union sigval value); 成功：0；失败：-1，设置 errno

```
union sigval {
    int     sival_int;
    void *sival_ptr;
};
```

向指定进程发送指定信号的同时，携带数据。但，如传地址，需注意，不同进程之间虚拟地址空间各自独立，将当前进程地址传递给另一进程没有实际意义。

极少使用，因为信号不允许携带大量信息。（了解）

中断系统调用

系统调用可分为两类：慢速系统调用和其他系统调用。

1. 慢速系统调用：可能会使进程永远阻塞的一类。如果在阻塞期间收到一个信号，该系统调用就被中断,不再继续执行(早期)；也可以设定系统调用是否重启。如，read、write、pause、wait...
2. 其他系统调用：getpid、getppid、fork...

结合 pause，回顾慢速系统调用：

慢速系统调用被中断的相关行为，实际上就是 pause 的行为： 如，read

① 想中断 pause，信号不能被屏蔽。

② 信号的处理方式必须是捕捉 (默认、忽略都不可以)

③ 中断后返回-1， 设置 errno 为 EINTR(表“被信号中断”)

可修改 sa_flags 参数来设置被信号中断后系统调用是否重启。SA_INTERRURT 不重启， SA_RESTART 重启。

sa_flags 还有很多可选参数，适用于不同情况。如：捕捉到信号后，在执行捕捉函数期间，不希望自动阻塞该信号，可将 sa_flags 设置为 SA_NODEFER，除非 sa_mask 中包含该信号。

进程组：

概念和特性

进程组，也称之为作业。BSD 于 1980 年前后向 Unix 中增加的一个新特性。代表一个或多个进程的集合。每个进程都属于一个进程组。在 waitpid 函数和 kill 函数的参数中都曾使用到。操作系统设计的进程组的概念，是为了简化对多个进程的管理。

当父进程，创建子进程的时候，默认子进程与父进程属于同一进程组。进程组 ID==第一个进程 ID(组长进程)。所以，组长进程标识：其进程组 ID==其进程 ID

可以使用 kill -SIGKILL -进程组 ID(负的)来将整个进程组内的进程全部杀死。 【kill_multprocess.c】

组长进程可以创建一个进程组，创建该进程组中的进程，然后终止。只要进程组中有一个进程存在，进程组就存在，与组长进程是否终止无关。

进程组生存期：进程组创建到最后一个进程离开(终止或转移到另一个进程组)。

一个进程可以为自己或子进程设置进程组 ID

进程组操作函数

getpgrp 函数

获取当前进程的进程组 ID

pid_t getpgrp(void); 总是返回调用者的进程组 ID

getpgid 函数

获取指定进程的进程组 ID

pid_t getpgid(pid_t pid); 成功：0；失败：-1，设置 errno

如果 pid = 0，那么该函数作用和 getpgrp 一样。

setpgid 函数

改变进程默认所属的进程组。通常可用来加入一个现有的进程组或创建一个新进程组。

int setpgid(pid_t pid, pid_t pgid); 成功：0；失败：-1，设置 errno

将参 1 对应的进程，加入参 2 对应的进程组中。

注意：

1. 如改变子进程为新的组，应 fork 后，exec 前。
2. 权级问题。非 root 进程只能改变自己创建的子进程，或有权限操作的进程

进程编号---》进程组再编号---》会话：

会话

创建会话

创建一个会话需要注意以下 6 点注意事项：

1. 调用进程不能是进程组组长，该进程变成新会话首进程(session header)
2. 该进程成为一个新进程组的组长进程。
3. 需有 root 权限(ubuntu 不需要)
4. 新会话丢弃原有的控制终端，该会话没有控制终端
5. 该调用进程是组长进程，则出错返回
6. 建立新会话时，先调用 fork，父进程终止，子进程调用 setsid

父进程不能作会长（1 点）

getsid 函数

获取进程所属的会话 ID

pid_t getsid(pid_t pid)；成功：返回调用进程的会话 ID；失败：-1，设置 errno

pid 为 0 表示察看当前进程 session ID

ps ajx 命令查看系统中的进程。参数 a 表示不仅列当前用户的进程，也列出所有其他用户的进程，参数 x 表示不仅列有控制终端的进程，也列出所有无控制终端的进程，参数 j 表示列出与作业控制相关的信息。

组长进程不能成为新会话首进程，新会话首进程必定会成为组长进程。

setsid 函数

创建一个会话，并以自己的 ID 设置进程组 ID，同时也是新会话的 ID。

pid_t setsid(void)； 成功：返回调用进程的会话 ID；失败：-1，设置 errno

调用了 setsid 函数的进程，既是新的会长，也是新的组长。

ps ajx ：PPID（父进程 id） PID（本进程 id） PGID（所在进程组） SID（所在会议）

僵尸进程

> 当一个子进程结束运行（一般是调用exit、运行时发生致命错误或收到终止信号所导致）时，子进程的退出状态（返回值）会回报给操作系统，系统则以SIGCHLD信号将子进程被结束的事件告知父进程，此时子进程的进程控制块（PCB）仍驻留在内存中。一般来说，收到SIGCHLD后，父进程会使用wait系统调用以获取子进程的退出状态，然后内核就可以从内存中释放已结束的子进程的PCB；而如若父进程没有这么做的话，子进程的PCB就会一直驻留在内存中，也即成为僵尸进程

孤儿进程

孤儿进程则是指当一个父进程退出，而它的一个或多个子进程还在运行，那么那些子进程将成为孤儿进程。孤儿进程将被init进程(进程号为1)所收养，并由init进程对它们完成状态收集工作。

守护进程

Daemon(精灵)进程，是 Linux 中的后台服务进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。一般采用以 d 结尾的名字。

Linux 后台的一些系统服务进程，没有控制终端，不能直接和用户交互。不受用户登录、注销的影响，一直在运行着，他们都是守护进程。如：预读入缓输出机制的实现；ftp 服务器；nfs 服务器等。

守护进程

编写一个守护进程。

1. 创建子进程 fork
2. 子进程创建新会话 setsid()
3. 改变进程的工作目录 chdir()
4. 指定文件掩码 umask()
5. 将0/1/2重定向 /dev/null dup2()
6. 守护进程主逻辑。

2：为什么要选会话作为守护进程，因为会话会丢弃控制终端，而守护进程正是没有控制终端的。

3:改变进程的工作目录,因为要保证该进程不会由于文件系统原因出现不能执行的情况(如，u 盘执行，突然拔掉 u 盘）

5：守护进程会脱离终端，文件描述符（012），标准输入、标准输出、标准错误都无用（无法输出到终端显示）

【线程】

什么是线程

LWP：light weight process 轻量级的进程，本质仍是进程(在 Linux 环境下)

进程：独立地址空间，拥有 PCB

线程：也有 PCB，但没有独立的地址空间(共享)

区别：在于是否共享地址空间。 独居(进程)；合租(线程)。

Linux 下： 线程：最小的执行单位

进程：最小分配资源单位，可看成是只有一个线程的进程。

1. 轻量级进程(light-weight process)，也有 PCB，创建线程使用的底层函数和进程一样，都是 clone

LWP: 线程号，ps –Lf pid 可查看某个进程的线程；

线程 id：在进程内区分线程，而不使用 LWP 来区分；pthread_self() 获取

尽量避开线程与信号的混合使用；

线程同步的方法：互斥锁、读写锁、条件变量、信号量

线程共享资源

1.文件描述符表

2.每种信号的处理方式

3.当前工作目录

4.用户 ID 和组 ID

5.内存地址空间 (.text/.data/.bss/heap/共享库)

共享信号处理方式，独享信号屏蔽字，栈空间，函数调用的空间
线程可以用全局变量通信，而进程必须 IPC 通信

线程非共享资源

1.线程 id

2.处理器现场和栈指针(内核栈)

3.独立的栈空间(用户空间栈)

4.errno 变量

5.信号屏蔽字

6.调度优先级

pthread_self 函数

获取线程 ID。其作用对应进程中 getpid() 函数。

pthread_t pthread_self(void); 返回值：成功：0； 失败：无！

线程 ID：pthread_t 类型，本质：在 Linux 下为无符号整数(%lu)，其他系统中可能是结构体实现

线程 ID 是进程内部，识别标志。(两个进程间，线程 ID 允许相同)

pthread_create 函数

创建一个新线程。 其作用，对应进程中 fork() 函数。

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
```

返回值：成功：0； 失败：错误号 -----Linux 环境下，所有线程特点，失败均直接返回错误号。

参数：

pthread_t：当前 Linux 中可理解为：typedef unsigned long int pthread_t;

参数 1：传出参数，保存系统为我们分配好的线程 ID

参数 2：通常传 NULL，表示使用线程默认属性。若想使用具体属性也可以修改该参数。

参数 3：函数指针，指向线程主函数(线程体)，该函数运行结束，则线程结束。

参数 4：线程主函数执行期间所使用的参数。

主控线程：调用 pthread_create 函数的进程会沦陷为线程，但是进程仍不会消失。刚创建完的线程也会与主控线程竞争 cpu，谁先挣到谁先执行!!!

线程与共享

线程间共享全局变量！

【牢记】：线程默认共享数据段、代码段等地址空间，常用的是全局变量。而进程不共享全局变量，只能借助 mmap。

pthread_exit 函数

将单个线程退出

void pthread_exit(void *retval); 参数：retval 表示线程退出状态，通常传 NULL

思考：使用 exit 将指定线程退出，可以吗？ 【pthrd_exit.c】

结论：线程中，禁止使用 exit 函数，会导致进程内所有线程全部退出。

在不添加 sleep 控制输出顺序的情况下。pthread_create 在循环中，几乎瞬间创建 5 个线程，但只有第 1 个线程有机会输出（或者第 2 个也有，也可能没有，取决于内核调度）如果第 3 个线程执行了 exit，将整个进程退出了，所以全部线程退出了。

所以，多线程环境中，应尽量少用，或者不使用 exit 函数，取而代之使用 pthread_exit 函数，将单个线程退出。任何线程里 exit 导致进程退出，其他线程未工作结束，主控线程退出时不能 return 或 exit。

pthread((void*) 1)指定退出状态为 1

pthread_join 函数

阻塞等待线程退出，获取线程退出状态 其作用，对应进程中 waitpid() 函数。

int pthread_join(pthread_t thread, void **retval); 成功：0；失败：错误号

参数：thread：线程 ID （【注意】：不是指针）；retval：存储线程结束状态。

对比记忆：

进程中：main 返回值、exit 参数-->int；等待子进程结束 wait 函数参数-->int *

线程中：线程主函数返回值、pthread_exit-->void *；等待线程结束 pthread_join 函数参数-->void **

pthread_detach 函数

实现线程分离

int pthread_detach(pthread_t thread); 成功：0；失败：错误号

线程分离状态：指定该状态，线程主动与主控线程断开关系。线程结束后，其退出状态不由其他线程获取，而直接自己自动释放。网络、多线程服务器常用。

进程若有该机制，将不会产生僵尸进程。僵尸进程的产生主要由于进程死后，大部分资源被释放，一点残留资源仍存于系统中，导致内核认为该进程仍存在。

也可使用 pthread_create 函数参 2(线程属性)来设置线程分离。

一般情况下，线程终止后，其终止状态一直保留到其它线程调用 pthread_join 获取它的状态为止，但是线程也可以被置为 detach 状态，**这样的线程一旦终止就立刻回收它占用的所有资源，而不保留终止状态。**不能对一个已经处于 detach 状态的线程调用 pthread_join，这样的调用将返回 EINVAL 错误。也就是说，如果已经对一个线程调用了 pthread_detach 就不能再调用 pthread_join 了。

pthread_cancel 函数

杀死(取消)线程　　其作用，对应进程中 kill() 函数。

int pthread_cancel(pthread_t thread); 成功：0；失败：错误号

【注意】：线程的取消并不是实时的，而有一定的延时，需要等待线程到达某个取消点(检查点)。

类似于玩游戏存档，必须到达指定的场所(存档点，如：客栈、仓库、城里等)才能存储进度。杀死线程也不是立刻就能完成，必须要到达取消点。

取消点：是线程检查是否被取消，并按请求进行动作的一个位置。通常是一些系统调用 creat，open，pause，close，read，write..... 执行命令 man 7 pthreads 可以查看具备这些取消点的系统调用列表。也可参阅 APUE.12.7 取消选项小节。

可粗略认为一个系统调用(进入内核)即为一个取消点。如线程中没有取消点，可以通过调用 pthread_testcancel 函数自行设置一个取消点。

被取消的线程，　退出值定义在 Linux 的 pthread 库中。常数 PTHREAD_CANCELED 的值是-1。可在头文件 pthread.h 中找到它的定义：**#define PTHREAD_CANCELED ((void *) -1)**。因此当我们对一个已经被取消的线程使用 pthread_join 回收时，得到的返回值为-1。

线程可以相互之间回收，进程必须父进程进程回收。

线程控制原语	进程控制原语
pthread_create()	fork();
pthread_self()	getpid();
pthread_exit()	exit(); / return
pthread_join()	wait()/waitpid()
pthread_cancel()	kill()
pthread_detach()	

线程属性

本节作为指引性介绍，linux 下线程的属性是可以根据实际项目需要，进行设置，之前我们讨论的线程都是采用线程的默认属性，默认属性已经可以解决绝大多数开发时遇到的问题。如我们对程序的性能提出更高的要求那么需要设置线程属性，比如可以通过设置线程栈的大小来降低内存的使用，增加最大线程个数。

```
typedef struct
{
    int             etachstate;    //线程的分离状态
    int             schedpolicy;   //线程调度策略
    struct sched_param  schedparam;  //线程的调度参数
    int             inheritsched;  //线程的继承性
    int             scope;         //线程的作用域
    size_t          guardsize;     //线程栈末尾的警戒缓冲区大小
    int             stackaddr_set; //线程的栈设置
    void*           stackaddr;     //线程栈的位置
    size_t          stacksize;     //线程栈的大小
} pthread_attr_t;
```

线程属性初始化

注意：应先初始化线程属性，再 pthread_create 创建线程

初始化线程属性

int pthread_attr_init(pthread_attr_t *attr); 成功：0；失败：错误号

销毁线程属性所占用的资源

int pthread_attr_destroy(pthread_attr_t *attr); 成功：0；失败：错误号

ulimt –a 查看栈大小，设置方式：（以设置分离属性为例）：

```
pthread_attr_t attr;                    /*通过线程属性来设置游离态*/
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr,   PTHREAD_CREATE_DETACHED);
pthread_create(&tid, &attr, tfn, NULL);
```

线程库版本查看：getconf GNU_LIBPTHREAD_VERSION

线程使用注意事项

1. 主线程退出其他线程不退出，主线程应调用 pthread_exit
2. 避免僵尸线程

   pthread_join

   pthread_detach

   pthread_create 指定分离属性

   被 join 线程可能在 join 函数返回前就释放完自己的所有内存资源，所以不应当返回被回收线程栈中的值；
3. malloc 和 mmap 申请的内存可以被其他线程释放
4. 应避免在多线程模型中调用 fork 除非，马上 exec，子进程中只有调用 fork 的线程存在，其他线程在子进程中均 pthread_exit
5. 信号的复杂语义很难和多线程共存，应避免在多线程引入信号机制

//----------

线程同步

同步即协同步调，按预定的先后次序运行。

线程同步，指一个线程发出某一功能调用时，在没有得到结果之前，该调用不返回。同时其它线程为保证数据一致性，不能调用该功能。

互斥量 mutex

Linux 中提供一把互斥锁 mutex（也称之为互斥量）。

每个线程在对资源操作前都尝试先加锁，成功加锁才能操作，操作结束解锁。

资源还是共享的，线程间也还是竞争的，

但通过“锁”就将资源的访问变成互斥操作，而后与时间有关的错误也不会再产生了。

但，应注意：同一时刻，只能有一个线程持有该锁。

当 A 线程对某个全局变量加锁访问，B 在访问前尝试加锁，拿不到锁，B 阻塞。C 线程不去加锁，而直接访问该全局变量，依然能够访问，但会出现数据混乱。

所以，互斥锁实质上是操作系统提供的一把“建议锁”（又称“协同锁”），建议程序中有多线程访问共享资源的时候使用该机制。但，并没有强制限定。

因此，即使有了 mutex，如果有线程不按规则来访问数据，依然会造成数据混乱。

主要应用函数：

pthread_mutex_init 函数

pthread_mutex_destroy 函数

pthread_mutex_lock 函数

pthread_mutex_trylock 函数

pthread_mutex_unlock 函数

以上 5 个函数的返回值都是：成功返回 0， 失败返回错误号。

pthread_mutex_t 类型，其本质是一个结构体。为简化理解，应用时可忽略其实现细节，简单当成整数看待。

pthread_mutex_t mutex; 变量 mutex 只有两种取值 1、0。

pthread_mutex_init 函数

初始化一个互斥锁(互斥量) ---> 初值可看作 1

int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr);

参 1：传出参数，调用时应传 &mutex

restrict 关键字：只用于限制指针，告诉编译器，所有修改该指针指向内存中内容的操作，只能通过本指针完成。不能通过除本指针以外的其他变量或指针修改

参 2：互斥量属性。是一个传入参数，通常传 NULL，选用默认属性(线程间共享)。 参 APUE.12.4 同步属性

1. 静态初始化：如果互斥锁 mutex 是静态分配的（定义在全局，或加了 static 关键字修饰），可以直接使用宏进行初始化。e.g. pthead_mutex_t muetx = PTHREAD_MUTEX_INITIALIZER;
2. 动态初始化：局部变量应采用动态初始化。e.g. pthread_mutex_init(&mutex, NULL)

死锁

1. 线程试图对同一个互斥量 A 加锁两次。
2. 线程 1 拥有 A 锁，请求获得 B 锁；线程 2 拥有 B 锁，请求获得 A 锁

读写锁

与互斥量类似，但读写锁允许更高的并行性。其特性为：写独占，读共享。

读写锁状态：

一把读写锁具备三种状态：

1. 读模式下加锁状态 (读锁)
2. 写模式下加锁状态 (写锁)
3. 不加锁状态

读写锁特性：

1. 读写锁是“写模式加锁”时， 解锁前，所有对该锁加锁的线程都会被阻塞。
2. 读写锁是“读模式加锁”时， 如果线程以读模式对其加锁会成功；如果线程以写模式加锁会阻塞。
3. 读写锁是“读模式加锁”时， 既有试图以写模式加锁的线程，也有试图以读模式加锁的线程。那么读写锁会阻塞随后的读模式锁请求。优先满足写模式锁。**读锁、写锁并行阻塞，写锁优先级高**

读写锁也叫共享-独占锁。当读写锁以读模式锁住时，它是以共享模式锁住的；当它以写模式锁住时，它是以独占模式锁住的。**写独占、读共享。**

主要应用函数：

pthread_rwlock_init 函数

pthread_rwlock_destroy 函数

pthread_rwlock_rdlock 函数

pthread_rwlock_wrlock 函数

pthread_rwlock_tryrdlock 函数

pthread_rwlock_trywrlock 函数

pthread_rwlock_unlock 函数

以上 7 个函数的返回值都是：成功返回 0， 失败直接返回错误号。

pthread_rwlock_t 类型 用于定义一个读写锁变量。

pthread_rwlock_t rwlock;

条件变量：

条件变量本身不是锁！但它也可以造成线程阻塞。通常与互斥锁配合使用。给多线程提供一个会合的场所。

主要应用函数：

pthread_cond_init 函数

pthread_cond_destroy 函数

pthread_cond_wait 函数

pthread_cond_timedwait 函数

pthread_cond_signal 函数

pthread_cond_broadcast 函数

以上 6 个函数的返回值都是：成功返回 0， 失败直接返回错误号。

pthread_cond_t 类型 用于定义条件变量

pthread_cond_t cond;

pthread_cond_wait 函数

阻塞等待一个条件变量

int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex);

函数作用：

1. 阻塞等待条件变量 cond（参 1）满足
2. 释放已掌握的互斥锁（解锁互斥量）相当于 pthread_mutex_unlock(&mutex);

   1.2.两步为一个原子操作。
3. 当被唤醒，pthread_cond_wait 函数返回时，解除阻塞并重新申请获取互斥锁 pthread_mutex_lock(&mutex);

pthread_cond_signal 函数

唤醒至少一个阻塞在条件变量上的线程

```
int pthread_cond_signal(pthread_cond_t *cond);
```

pthread_cond_broadcast 函数

唤醒全部阻塞在条件变量上的线程

```
int pthread_cond_broadcast(pthread_cond_t *cond);
```

pthread_cond_timedwait 函数

限时等待一个条件变量

```
int pthread_cond_timedwait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex, const struct timespec
*restrict abstime);
```

参 3： 参看 man sem_timedwait 函数，查看 struct timespec 结构体。

```
struct timespec {
    time_t tv_sec;      /* seconds */ 秒
    long   tv_nsec;     /* nanosecondes*/ 纳秒
}
```

形参 abstime：绝对时间。

如：time(NULL)返回的就是绝对时间。而 alarm(1)是相对时间，相对当前时间定时 1 秒钟。

```
struct timespec t = {1, 0};
pthread_cond_timedwait (&cond, &mutex, &t); 只能定时到 1970 年 1 月 1 日 00:00:01 秒(早已经过去)
```

正确用法：

```
time_t cur = time(NULL); 获取当前时间。
struct timespec t;  定义 timespec 结构体变量 t
t.tv_sec = cur+1;  定时 1 秒
pthread_cond_timedwait (&cond, &mutex, &t); 传参
```

参 APUE.11.6 线程同步条件变量小节

信号量

进化版的互斥锁（1 --> N）

由于互斥锁的粒度比较大，如果我们希望在多个线程间对某一对象的部分数据进行共享，使用互斥锁是没有办法实现的，只能将整个数据对象锁住。这样虽然达到了多线程操作共享数据时保证数据正确性的目的，却无形中导致线程的并发性下降，线程从并行执行，变成了串行执行。与直接使用单进程无异。

信号量，是相对折中的一种处理方式，既能保证同步，数据不混乱，又能提高线程并发。

主要应用函数：

sem_init 函数

sem_destroy 函数

sem_wait 函数

sem_trywait 函数

sem_timedwait 函数

sem_post 函数

类似加锁函数，post 类似解锁函数

以上 6 个函数的返回值都是：成功返回 0， 失败返回-1，同时设置 errno。(注意，它们没有 pthread 前缀)

sem_t 类型，本质仍是结构体。但应用期间可简单看作为整数，忽略实现细节（类似于使用文件描述符）。

sem_t sem; 规定信号量 sem 不能 < 0。头文件 <semaphore.h>

以上信号量函数也可用于进程同步。注意信号（signal）与信号量（sem）是完全独立的东西！

信号量基本操作：

sem_wait: 1. 信号量大于 0，则信号量--　　（类比 pthread_mutex_lock）
　　| 2. 信号量等于 0，造成线程阻塞
对应
　　|
sem_post: 将信号量++，同时唤醒阻塞在信号量上的线程　　（类比 pthread_mutex_unlock）

但，由于 sem_t 的实现对用户隐藏，所以所谓的++、--操作只能通过函数来实现，而不能直接++、--符号。

信号量的初值，决定了占用信号量的线程的个数。

rand()%10：产生 0~（10-1）

【进程 4：同步】

进程间同步

互斥量 mutex

进程间也可以使用互斥锁，来达到同步的目的。但应在 pthread_mutex_init 初始化之前，修改其属性为进程间共享。mutex 的属性修改函数主要有以下几个。

主要应用函数：

pthread_mutexattr_t mattr 类型：　　用于定义 mutex 锁的【属性】

pthread_mutexattr_init 函数：　　初始化一个 mutex 属性对象

int pthread_mutexattr_init(pthread_mutexattr_t *attr);

pthread_mutexattr_destroy 函数：　　销毁 mutex 属性对象（而非销毁锁）

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);

pthread_mutexattr_setpshared 函数：　　修改 mutex 属性。

int pthread_mutexattr_setpshared(pthread_mutexattr_t *attr, int pshared);

参 2：pshared 取值：

线程锁：PTHREAD_PROCESS_PRIVATE (mutex 的默认属性即为线程锁，进程间私有)

进程锁：PTHREAD_PROCESS_SHARED

attribute 属性

文件锁

借助 fcntl 函数来实现锁机制。 操作文件的进程没有获得锁时，可以打开，但无法执行 read、write 操作。

fcntl 函数：　获取、设置文件访问控制属性。

int fcntl(int fd, int cmd, ... /* arg */);

参 2：

F_SETLK (struct flock *) 设置文件锁（trylock）

F_SETLKW (struct flock *) 设置文件锁（lock）W --> wait

F_GETLK (struct flock *) 获取文件锁

参 3：

```
struct flock {
        ...
        short l_type;       锁的类型：F_RDLCK 、F_WRLCK 、F_UNLCK
        short l_whence;     偏移位置：SEEK_SET、SEEK_CUR、SEEK_END
        off_t l_start;      起始偏移：1000
        off_t l_len;        长度：0 表示整个文件加锁
        pid_t l_pid;        持有该锁的进程 ID：(F_GETLK only)
        ...
};
```

pid 获取，只能 fcntl 第二个参数必须是 F_GETLK

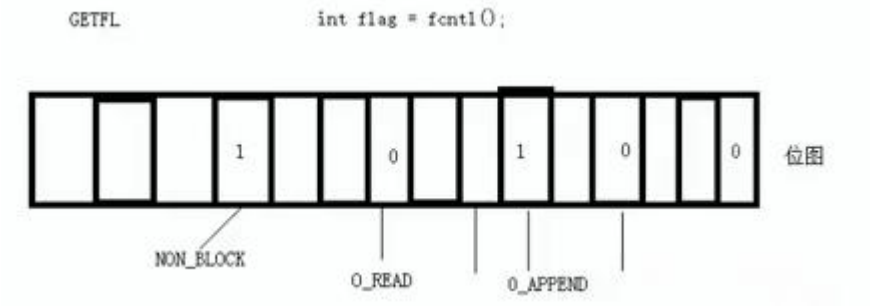
解释：从 l_whence 位置偏移 l_start 字节，开始锁 l_len 字节

读锁共享，写锁独占!!! 多线程无法使用文件锁，可以使用自己的读写锁!!!

位图：

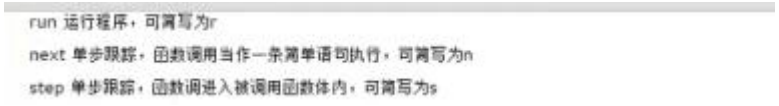
给某个文件描述符添加非阻塞属性：

```
int flag   =   fcntl(fd,   F_GETFL);
flag   |=   O_NONBLOCK;
fcntl(fd,   F_SETFL, flag);
```

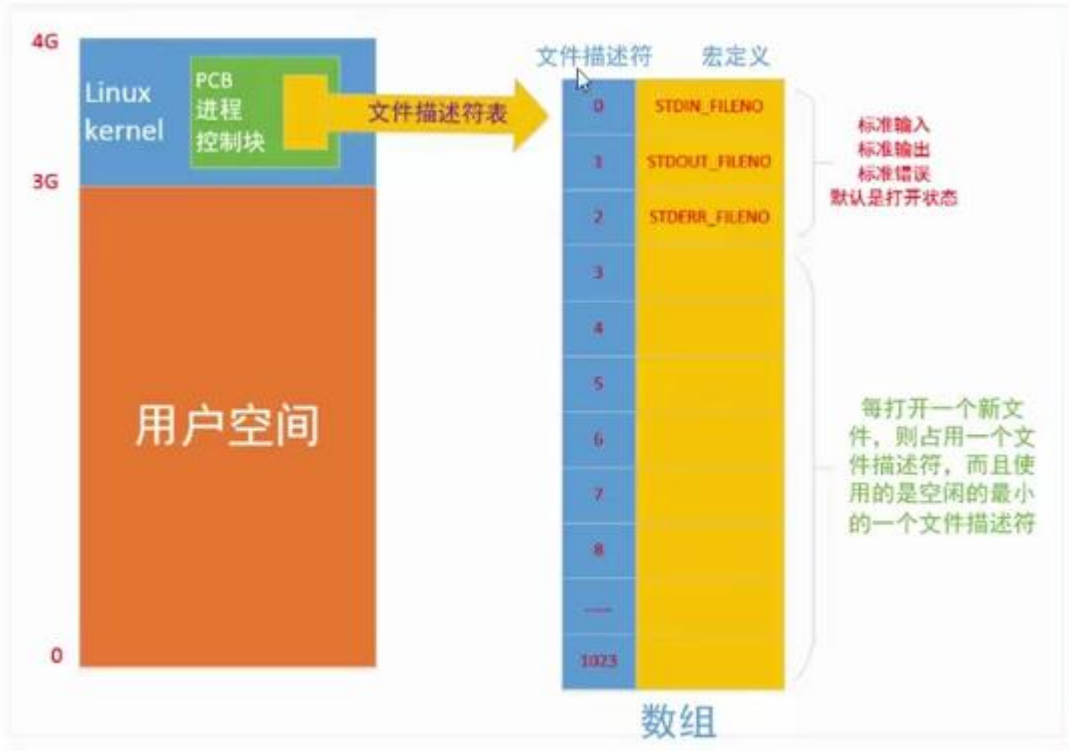


【gdb 调试】

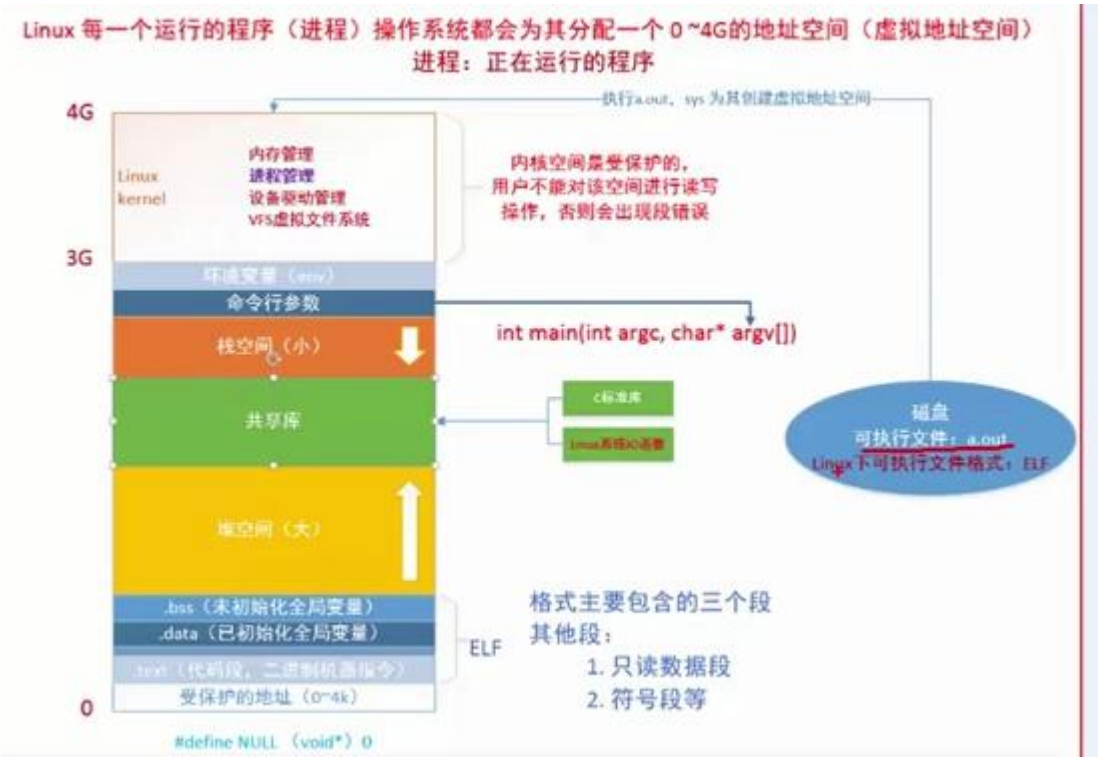
gcc main.c –g　　、gdb　a.out　、　list（显示代码）、 l（展开）、指定某一行断点、run 运行完/start 只执行一步程序停止、n 单步运行、q 退出

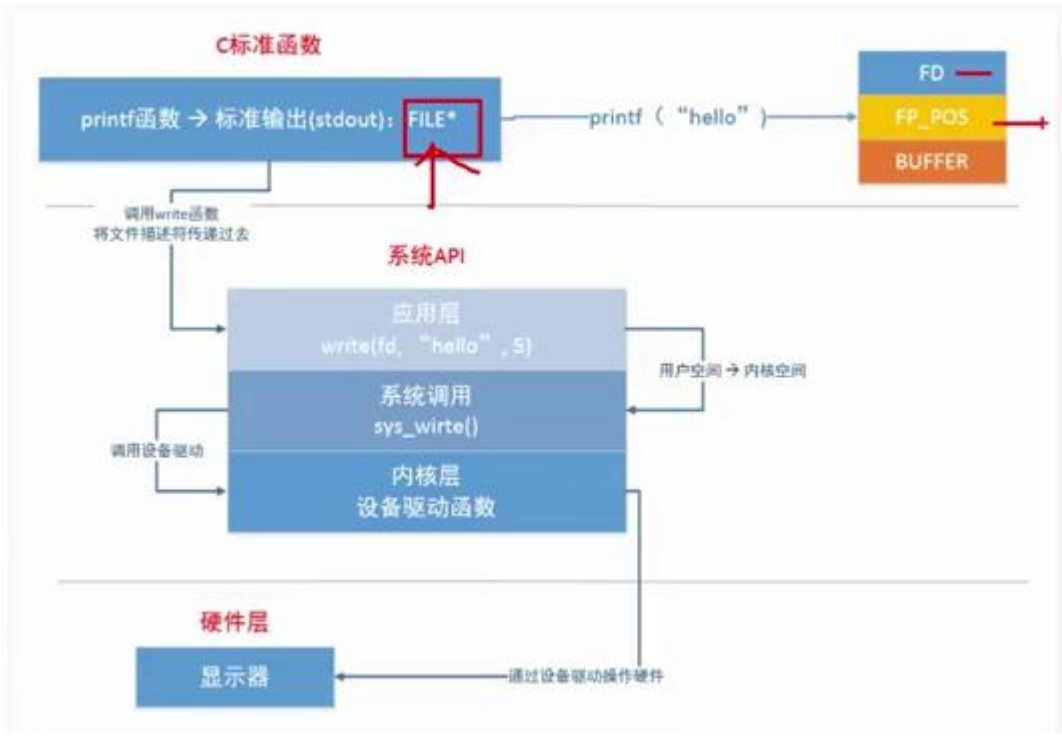


【虚拟地址空间】



32 位机分配 4G，64 位机分配 8 位（2 的 32 次方）


Linux 每一个运行的程序（进程）操作系统都会为其分配一个 0 ~4G的地址空间（虚拟地址空间）
进程：正在运行的程序
4G
3G
0
执行a.out，sys 为其创建虚拟地址空间
Linux kernel
内存管理
进程管理
设备驱动管理
VFS虚拟文件系统
内核空间是受保护的，
用户不能对该空间进行读写
操作，否则会出现段错误
环境变量（env）
命令行参数
int main(int argc, char* argv[])
栈空间（小）
共享库
C标准库
Linux系统IO函数
堆空间（大）
磁盘
可执行文件：a.out
Linux下可执行文件格式：ELF
.bss（未初始化全局变量）
.data（已初始化全局变量）
.text（代码段，二进制机器指令）
受保护的地址（0~4k）
ELF
格式主要包含的三个段
其他段：
1. 只读数据段
2. 符号段等
#define NULL （void*）0


C标准函数
printf函数 → 标准输出(stdout)：FILE*
printf（ “hello” ）
FD
FP_POS
BUFFER
调用write函数
将文件描述符传递过去
系统API
应用层
write(fd, “hello” , 5)
用户空间 → 内核空间
系统调用
sys_wirte()
调用设备驱动
内核层
设备驱动函数
硬件层
显示器
通过设备驱动操作硬件