day5 下午

进程相关

进程：

程序：死的。只占用磁盘空间。 --剧本。

```
进程；活的。运行起来的程序。占用内存、cpu等系统资源。 --戏。
```

PCB进程控制块：

重点的：（其余的在课件）

```
进程id

文件描述符表

进程状态:  初始态、就绪态、运行态、挂起态、终止态。

进程工作目录位置

*umask掩码  不同的shell 进程是不同的 进程，改一个里面的,只是改那个进程的，内核空间那部分并没有改变
Shell 进程在运行时有各自独立的用户空间，每个进程拥有自己的环境变量、进程属性等。修改一个 Shell 进程中的某个环境变量（例如 umask），只会影响当前进程及其子进程，不会影响内核空间或其他 Shell 进程。

信号相关信息资源。

用户id和组id
```

环境变量

常见的

```
PATH   ---- 可执行文件的搜索路径  /bin/...
SHELL  ---- 通常是 /bin/bash
TERM   ---- 当前终端类型，在图形界面终端下它的值通常是xterm，显示方式不同：图形界面终端可以显示汉字，而字符终端一般不行
LANG   ---- 语言和locale，决定了字符编码以及时间、货币等信息的显示格式。
HOME   ---- 当前用户主目录的路径
加$，可以查看值

env  --- 快速查看所有环境变量

```

fork函数：（重点）

```
pid_t fork(void)

创建子进程。父子进程各自返回。父进程返回 子进程pid。 子进程 返回 0.
fork后，fork变为两个，父子各一个
一次fork， 两次返回
两个进程 都从 fork 后的位置 继续执行，
但不保证 同步进行

getpid(); 获取自己的 pid
getppid(); 获取 父亲的 pid

循环创建N个子进程模型。 每个子进程标识自己的身份。
```

父子进程相同：

```
刚fork后。 data段、text段、堆、栈、环境变量、全局变量、宿主目录位置、进程工作目录位置、信号处理方式
```

父子进程不同：

```
进程id、返回值、各自的父进程、进程创建时间、闹钟、未决信号集
```

父子进程共享：

```
读时共享、写时复制。--------- 不共享全局变量。
Copy-On-Write，COW  写时复制机制
父进程和子进程最初共享相同的物理内存页（直到某一方尝试修改数据时）。在它们开始修改内存时，操作系统才会为每个进程分配新的内存页，从而使父子进程的数据互不干扰。
父子 在写时 都会复制 一份新的

1. 文件描述符(特别注意) 2. mmap映射区(进程通信讲)。

在文件描述符上，凡是 父进程 共享的文件描述符，最后 都是被 父子 各自 使用，即使关闭，也是各自关闭自己的
而不是 父进程关闭，子进程就不能用，这是错误的

fork前的 文件描述符，会被 子进程 继承
但是 fork 后的  新的文件描述符， 是各自拥有的
```

进程 补充与实例

1.并发(gpt)

多核 是 并行，不是并发

实质上，并发是宏观并行，微观串行!

“并发”是计算机科学中的一个重要概念，指的是多个任务在同一时间段内执行。并发的实现不一定要求多个任务实际同时运行，而是通过合理调度和分时来让任务在逻辑上并行运行。

```
常见的并发模型
多线程
使用多个线程来实现并发，每个线程独立执行任务。线程共享同一进程的资源（如内存空间），但需要小心处理线程安全问题。

多进程
每个进程有独立的内存空间，通过进程间通信（IPC）实现数据共享。多进程通常更安全，但代价是较高的资源消耗。

协程
一种轻量级线程，通常在单线程中通过手动切换上下文来实现并发。协程避免了线程的切换开销，但需要编程者明确控制。

事件驱动模型
基于事件循环（如 Node.js 的事件驱动架构），通过非阻塞 I/O 和回调机制来高效处理并发任务。
```

```
并发相关的常见问题
资源竞争
多个任务同时访问共享资源（如变量或文件）时，可能导致数据不一致或死锁。

死锁
多个任务因相互等待资源释放而陷入无限等待。

线程安全
需要通过锁、信号量等机制，确保多个线程对共享资源的访问是安全的。

上下文切换
在多任务环境中，频繁的上下文切换可能导致性能下降。
```

2.单道程序设计， 多道程序设计(课件)

...

3.时钟中断(课件)

分时复用

...

4.虚拟地址

在 Linux 系统中，出现的 **0-4G 地址范围**是虚拟地址，而不是物理地址。Linux 操作系统使用**虚拟内存机制**，通过内存管理单元（MMU，Memory Management Unit）将虚拟地址映射到实际的物理内存地址。

虚拟地址的概念

虚拟地址是操作系统为每个进程提供的独立地址空间，这种机制使得进程之间互不干扰，并且每个进程都认为自己可以独占整个内存空间。通过 MMU 和页表，虚拟地址会动态映射到物理内存。

0-4G 地址范围的来源

在 Linux 的 32 位系统中，虚拟地址空间被划分为两部分：

1. **用户空间（0x00000000 - 0xBFFFFFFF，0-3G）：**
   分配给用户进程使用，程序代码、数据、堆、栈等都位于此范围内。
2. **内核空间（0xC0000000 - 0xFFFFFFFF，3G-4G）：**
   由操作系统内核保留，用于管理系统资源和提供服务，用户进程不能直接访问。

> 这种 3G/1G 的划分是 Linux 传统的内核设计模型，具体比例可以通过重新编译内核进行调整。

虚拟地址和物理地址的关系

- 虚拟地址是逻辑上的地址，由 CPU 执行指令时直接使用。
- 物理地址是实际内存中的地址，由 MMU 和页表将虚拟地址翻译成物理地址。
- 内核会维护页表，用于记录虚拟地址到物理地址的映射关系。

5.page

在大多数现代操作系统（如 Linux）中，内存被划分成固定大小的页（page），而每页的默认大小是 4KB（4096 字节）。

6.mmu分级

内核 与 用户空间 级别不同，一般是 内核是 0级，用户空间1级，这只是假设

mmu会实现 级别切换，因此 内核到用户空间 花销会大

7.环境变量(gpt)

环境变量 是操作系统中一组动态命名的值，用于影响程序的行为。它们存储在操作系统的环境中，通常由操作系统、Shell 或其他程序设置。环境变量通常用于传递配置信息、系统路径、用户信息等，供操作系统和程序在运行时使用。

环境变量的特点

1. **作用范围**：环境变量在进程中有效，且父进程会将其传递给子进程。
2. **继承机制**：当你启动一个新进程（如打开一个新的终端或运行一个程序）时，子进程会继承父进程的环境变量。子进程可以修改自己的环境变量，但不会影响父进程。
3. **存储位置**：环境变量存储在内存中，通常由 Shell 或操作系统设置。

8.fork 实例-1

可以看到 fork()前后的 输出 ， 父子进程的 先后顺序，以及 fork 后从哪开始

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
{
    printf("before fork---1\n");
    printf("before fork---2\n");
    printf("before fork---3\n");

    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        printf("child is created\n");
    }
    else if(pid > 0)
    {
        printf("parent : child pid is %d\n", pid);
    }

    printf("======================\n");

    return 0;
}
```

9.fork 实例-2

注意，不加sleep，会父进程id 不一样

`fork()` 后并不保证父子进程的同步完成，它们是并行执行的。`fork()` 只是创建一个新的子进程，然后父进程和子进程继续执行各自的代码，但它们并没有内建的同步机制。父进程和子进程将从 `fork()` 之后的代码行开始并行执行。

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
```

```c
{
    printf("before fork---1\n");
    printf("before fork---2\n");
    printf("before fork---3\n");

    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        printf("child is created\n");
        printf("child pid is %d\n", getpid());
        printf("parent pid is %d\n", getppid());
    }
    else if(pid > 0)
    {
        printf("parent: \nchild is created, pid is %d\n", pid);
        printf("my pid is %d\n", getpid());
        printf("my parent pid is %d\n", getppid());
        printf("====================\n");
    }

    printf("======================\n");
    sleep (30);
    return 0;
}
```

10.fork 实例-3

注意，这里 printf 加不加 \n ，决定是否立即输出

- C 标准库对输出流（例如 `stdout`）使用了 **缓冲机制**。默认情况下，标准输出是 **行缓冲** 的，也就是说，输出内容会先存储在缓冲区中，只有当遇到换行符（ `\n` ）、缓冲区满或者程序退出时，缓冲区的内容才会被实际输出到终端。
- 在你原来的代码中，如果没有加换行符 `\n`，`printf` 输出的内容会被先放入缓冲区，而不会立即显示在终端上。直到缓冲区被填满或程序结束时，才会把缓冲区的内容刷新出来。

`\n` **的作用**：

- 当你在字符串末尾加上换行符（ `\n` ）时，**会立即刷新缓冲区**，使得缓冲区中的内容被立刻输出到屏幕上。换行符告诉缓冲机制“内容已经完成，可以输出了”。
- 所以，在加了 `\n` 后，你的输出就不再延迟了。

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
{
```

```
    int n = 5;
    int i = 0;
    while (i<5)
    {
        pid_t pid = fork();
        if (pid < 0)
        {
            perror("fork error!");
            exit(1);
        }
        if(pid == 0)
        {
            sleep(3);  // 可以顺序 打印，不然子进程先后打印顺序不一样,但也仅是打印,产生还是无序
            printf("child %d \n", i);
            printf("child %d pid is %d \n", i, getpid());
            printf("child %d parent pid is %d \n", i, getppid());   // /n刷新缓冲区
            break;
        }
        else if(pid > 0)
        {
            i++;

            // printf("parent: my pid is %d \n",  getpid());
            continue;
        }
    }

    sleep(30);
    return 0;
}
```

11.读共享,写复制 实例-1

当调用 `fork()` 时，操作系统会为子进程创建一个 **独立的内存空间**，即使父子进程开始时有相同的数据（例如你定义的 `var = 100`），它们的内存地址是独立的。

写时复制机制 确保了父子进程初始时有相同的变量值，但修改之后的值不会互相影响。

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int var = 100;

int main(int argc, char *argv[])
{
```

```
    printf("before fork---1\n");
    printf("before fork---2\n");
    printf("before fork---3\n");

    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        printf("child var read = %d\n", var);
        var = 300;
        printf("child var write = %d\n", var);
    }
    else if(pid > 0)
    {
        printf("parent var read = %d\n", var);
        var = 200;
        printf("parent var write = %d\n", var);
    }

    printf("=======================\n");

    return 0;
}
// parent var read = 100
//parent var write = 200
//child var read = 100
//child var write = 300
```

---

day 6

day6 上 进程

gdb调试父子进程:

```
注意: gdb调试, 需要 gcc -g ... 生成exe, 再gdb exe
在进入fork前, 设置下面的这个,
设置父进程调试路径: set follow-fork-mode parent (默认)

设置子进程调试路径: set follow-fork-mode child
```

exec函数族:

所谓的族，表示 有一系列函数， 更多的见课件

pid 不变

注意:文件描述符 的共享 （gpt）

```
父子进程共享文件描述符:
文件描述符的继承:
当一个进程执行 exec 时，它会继承父进程的所有打开的文件描述符。也就是说，执行 exec 后，子进程会继承父进程的文件描述符表中的所有条目（即文件描述符号码）。例如，父进程打开了某个文件，执行 exec 后，子进程仍然可以访问这个文件。
文件描述符的指向:
文件描述符在父子进程之间是共享的，意味着它们指向相同的文件或资源。对文件的读写操作会影响文件的内容，这种影响在父进程和子进程之间是可见的。
文件描述符的关闭:
如果父进程在 exec 之前关闭了某个文件描述符，那么子进程也将无法访问这个文件描述符。
如果父进程在 exec 之后关闭了某个文件描述符（或者关闭某个文件描述符在 exec 之前关闭），子进程也不会受到影响，因为它会继承父进程在 exec 前打开的文件描述符。------------重点
关于文件描述符的控制:
exec 和文件描述符:
exec 本身不会关闭文件描述符。所有继承的文件描述符在 exec 后依然有效。
需要使用 fcntl() 函数或设置 close-on-exec 标志（FD_CLOEXEC）来明确控制哪些文件描述符应该在 exec 调用后被关闭。例如，可以通过 fcntl(fd, F_SETFD, FD_CLOEXEC) 来设置文件描述符在 exec 后关闭。
```

```
使进程执行某一程序。成功无返回值，失败返回 -1

int execlp(const char *file, const char *arg, ...);     借助 PATH 环境变量找寻待执行程序

    参1:  程序名

    参2:  argv0 --------这个是坑点,  从 0 开始的，不是通常的 1

    参3:  argv1

    ...:  argvN

    哨兵: NULL     //结束标记，必须手动加

int execl(const char *path, const char *arg, ...);      自己指定待执行程序路径。

int execvp();
```

ps ajx ⟶ pid ppid gid sid

孤儿进程：

```
父进程先于子进终止，子进程沦为“孤儿进程”，会被 init 进程领养。
ps ajx   可以看到 孤儿进程 父进程是是  init进程
```

僵尸进程：

```
子进程残留资源（PCB）存放于内核中

子进程终止，父进程尚未对子进程进行回收，在此期间，子进程为“僵尸进程”。   kill 对其无效。
[fork-2] <defunct>   带中括号，后面defunct ，这就是僵尸进程
状态 Z+
无法被  kill -9(强制终止) 无效，因为已经死了

需要杀  父亲进程，变成 孤儿进程，然后被 init 回收
```

wait函数：

回收子进程退出资源， 阻塞回收任意一个。

```
#include <sys/wait.h>
pid_t wait(int *status)

status 若为 NULL，则表示不关心进程结束原因!!!!!

参数：（传出） 回收进程的状态。

返回值：成功：  回收进程的pid

    失败：  -1，  errno

函数作用1：   阻塞等待子进程退出

函数作用2：   清理子进程残留在内核的 pcb 资源

函数作用3：   通过传出参数，得到子进程结束状态

注意，这个函数实际 得到两个 值：
返回之后：pid_t
传出参数：status，里面是 退出状态
是两个不一样的值
```

status 值的获取(还有更多,在man里)

```

获取子进程正常终止值：
•   WIFEXITED(status) --》 为真 --》调用 WEXITSTATUS(status) --》 得到 子进程退出值(子进程的 return %d，就是这个值)。
获取导致子进程异常终止信号：
•   WIFSIGNALED(status) --》 为真 --》调用 WTERMSIG(status) --》 得到 导致子进程异常终止的信号编号。

```

```
    // 下面三个 了解
WIFSTOPPED(status)
判断子进程是否被信号暂停（如收到 SIGSTOP）。
返回值：非零表示子进程暂停执行。
使用场景：检查子进程是否被暂停。
WSTOPSIG(status)
获取导致子进程暂停的信号编号（仅当 WIFSTOPPED(status) 为真时有效）。
返回值：导致子进程暂停的信号编号。
使用场景：分析是什么信号导致暂停。
WIFCONTINUED(status)（需要 _XOPEN_SOURCE ≥ 700）
判断子进程是否已继续运行（如收到 SIGCONT 信号）。

返回值：非零表示子进程已继续运行。
使用场景：检查子进程是否从暂停状态恢复。

waitpid函数:   指定某一个进程进行回收。可以设置非阻塞。            waitpid(-1,
&status, 0) == wait(&status);
```

快速记忆法：

```
正常退出（Exit）
WIFEXITED(status)   W----IF----EXIT---ED
判断是否“Exit”-ed（退出）。
记忆：IF（是否）+ EXITED（退出）。

WEXITSTATUS(status)  W----EXIT----STATUS
获取“Exit”-ed 的状态码。
记忆：EXIT + STATUS（退出状态）。

因信号终止（Signal）

WIFSIGNALED(status)   W---IF---SIGNAL---ED
判断是否因“Signal”-ed（信号）终止。
记忆：IF（是否）+ SIGNALED（信号终止）。

WTERMSIG(status)   W---TERM---SIG
获取“Terminate Signal”（终止信号）的编号。
记忆：TERM（终止）+ SIG（信号）。
```

waitpid:

```
<sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options)

参数:
    pid: 指定回收某一个子进程pid

        > 0: 待回收的子进程pid

        -1: 任意子进程

        0: 同组的子进程。

    status:  (传出)  回收进程的状态。
```

```
    options: WNOHANG 指定回收方式为，非阻塞。 如果没有子进程已经退出，则不等待子进程，

返回值:

    > 0 : 表成功回收的子进程 pid

    0 : 函数调用时， 参3 指定了WNOHANG， 并且，没有子进程结束。

    -1: 失败。errno

默认情况，同一个进程生成的多个子进程，是一个 进程组，但也可以进行 修改，分离到不同组
```

总结:

```
wait、waitpid    一次调用，回收一个子进程。  多个子进程，谁抢到是谁的

        想回收多个。while
```

==========

day6 上 补充与实例

1.exec(gpt)

完全替换当前进程:

- 调用 `exec` 后，当前进程的代码和数据会被替换为新程序的内容。
- 但进程号（`PID`）不变，文件描述符会根据继承规则保留。

不会返回:

- 如果 `exec` 调用成功，则不会返回到调用点；否则返回 `-1`，并设置 `errno`。

环境变量继承:

- 默认情况下，新程序继承父进程的环境变量。

pid 不变

> 更通俗的 理解，原本 fork 后 的子进程，会执行 父进程 fork 之后的 代码，但是exec后，整个子进程原本的代码会消失，而是去执行 另一个程序，

2.execlp 实例-1

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
```

```c
{
    printf("before fork---1\n");
    printf("before fork---2\n");
    printf("before fork---3\n");

    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        //execlp(ls, -l, -R, NULL);  // 错误写法
        execlp("ls","ls", "-l", "-R", NULL);
        perror("execlp error!");  // 这个 不用加判断, 因为如果成功, 所有代码就被丢弃, 换成新的 程序代码
        exit(1);
    }
    else if(pid > 0)
    {
        printf("parent pid is %d\n", getpid());
    }

    printf("========================\n");

    return 0;
}
```

3.execl 实例-1

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
{
    printf("before fork---1\n");
    printf("before fork---2\n");
    printf("before fork---3\n");

    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        //execlp(ls, -l, -R, NULL);  // 错误写法
```

```
        execl("/bin/ls","ls", "-l", "-R", NULL);
        perror("execl error!");  // 这个 不用加判断，因为如果成功，所有代码就被丢弃，换成新的 程序代码
        exit(1);
    }
    else if(pid > 0)
    {
        printf("parent pid is %d\n", getpid());
    }

    printf("======================\n");

    return 0;
}
```

4.execl 实例-2

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
{
    printf("before fork---1\n");
    printf("before fork---2\n");
    printf("before fork---3\n");

    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        //execlp(ls, -l, -R, NULL);  // 错误写法
        execl("./execlp-1","./execlp-1",  NULL);
        perror("execlp error!");  // 这个 不用加判断，因为如果成功，所有代码就被丢弃，换成新的 程序代码
        exit(1);
    }
    else if(pid > 0)
    {
        printf("parent pid is %d\n", getpid());
    }

    printf("======================\n");

    return 0;
```

```
}
```

5.excel 实例-3

使用子进程 实现 ps aux > .txt

这里面有个知识点： 文件描述符是 父子 共享的!!!!

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main(int argc, char *argv[])
{
    int fd;
    fd = open("ps-aux.txt", O_CREAT | O_RDWR | O_TRUNC, 0644);
    dup2(fd, 1);  // 子进程仍然有效的原因是 文件描述符表 是共享的, 还有一个是mmap
    if(fd < 0)
    {
        perror("open error!");
        exit(1);
    }
    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        execlp("ps","ps","aux",NULL);
        perror("execlp error!");  // 这个 不用加判断, 因为如果成功, 所有代码就被丢弃, 换成新的 程序代码
        exit(1);
    }
    close(fd);
    return 0;
}
```

6.wait 实例-1

包含 wait的 status 的获取

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>

```

```c
int main(int argc, char *argv[])
{
    printf("before fork---1\n");
    printf("before fork---2\n");
    printf("before fork---3\n");

    pid_t wpid;
    int status;
    pid_t pid = fork();
    if(pid < 0)
    {
        perror("fork error!");
        exit(1);
    }
    else if(pid == 0)
    {
        sleep(20);  // 用于 测试 不同的 status 获取
        printf("child is created\n");
        return 100;   // status 获取的正常退出值
    }
    else if(pid > 0)
    {
        printf("parent : child pid is %d\n", pid);
        wpid = wait(&status);  // wait 应用
        if(wpid < 0)
        {
            perror("wait error");
            exit(1);
        }

        // status获取
        if(WIFEXITED(status))
        {
            printf("child exit with %d\n", WEXITSTATUS(status));
        }

        if(WIFSIGNALED(status))
        {
            printf("child KILL with %d\n", WTERMSIG(status));
        }

        printf("parent wait child, wait pid is %d\n", wpid);
    }

    printf("======================\n");

    return 0;
}
```

7.kill -l 杀手表

```
1) SIGHUP    2) SIGINT    3) SIGQUIT   4) SIGILL    5) SIGTRAP
6) SIGABRT   7) SIGBUS    8) SIGFPE    9) SIGKILL   10) SIGUSR1
11) SIGSEGV   12) SIGUSR2   13) SIGPIPE   14) SIGALRM   15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD   18) SIGCONT   19) SIGSTOP   20) SIGTSTP
21) SIGTTIN   22) SIGTTOU   23) SIGURG    24) SIGXCPU   25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF   28) SIGWINCH  29) SIGIO     30) SIGPWR
31) SIGSYS

    man  7 kill
```

8.waitpid 实例-1

一定要注意，写时复制 ----- 父子进程 不共享 全局变量

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
int epid = 0;    // 父子进程不共享 全局变量，所以父进程里 这个 一直是0
int main(int argc, char *argv[])
{
    int n = 5;
    int i = 0;

    while (i<5)
    {
        pid_t pid = fork();
        if (pid < 0)
        {
            perror("fork error!");
            exit(1);
        }
        if(pid == 0)
        {
            // sleep(3);  // 可以顺序 打印，不然子进程先后打印顺序不一样,但也仅是打印,产生还是无序

            if(i==2)
            {
                epid = getpid();
                printf("----1-----%d", epid);
            }
            break;
        }
        else if(pid > 0)
        {
            i++;
            // printf("parent: my pid is %d \n",  getpid());
            continue;
```

```
        }
    }
    if(i==5)
            {
                sleep(5);
                pid_t wpid = waitpid(epid, NULL, WNOHANG); // epid 是0, 表示回收同组任意子进程, 这里最终返回的是 第一个子进程的 pid, 因为他快, 已经结束
                printf("-----%d",epid);    // 这里返回 0
                if(wpid == -1)
                {
                    perror("waitpid error");
                }
                printf("child pid %d is waitpid exit\n", wpid);

            }
    else
            {
                sleep(i);
                printf("child %d \n", i);
                printf("child %d pid is %d \n", i, getpid());
                printf("child %d parent pid is %d \n", i, getppid());
            }

    // sleep(30);
    return 0;
}
```

9.waitpid 实例-2

注意 : 子进程pid 在父进程的 获取

注意: WNOHANG 是 父进程 不等待指定子进程(未退出) 再往后执行, 而是直接往后执行, 和 非阻塞 比较像 , 实际上, 不用 waitpid 也是默认 非阻塞, 不阻塞等待

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    int n = 5;
    int i = 0;
    int epid = 0;

    while (i<5)
    {
        pid_t pid = fork();
         if(i == 2)
```

```
            {
                epid = pid;    // 父进程从这里 获得子进程 pid
            }
        if (pid < 0)
        {
            perror("fork error!");
            exit(1);
        }
        if(pid == 0)
        {
            break;
        }
        else if(pid > 0)
        {
            i++;
            // printf("parent: my pid is %d \n",  getpid());
            continue;
        }
    }

    if(i==5)
            {
                sleep(5);
                pid_t wpid = waitpid(epid, NULL, WNOHANG);  // 这里 同样 得到指定pid后, 设置里该pidsleep时间长,因此 子进程未结束, 直接返回, 返回值为 0
                printf("-----%d\n",epid);
                if(wpid == -1)
                {
                    perror("waitpid error");
                }
                printf("waitpid %d:child pid %d is waitpid exit\n", wpid, epid);

            }
    else
            {
                if(i==2)
                {
                    sleep(10);
                }
                sleep(i);
                printf("child %d \n", i);
                printf("child %d pid is %d \n", i, getpid());
                printf("child %d parent pid is %d \n", i, getppid());
            }

    // sleep(30);  // 即使是不等待子进程, 直接退出, 子进程会变为孤儿进程
    return 0;
}

```

10.wait和waitpid阻塞

`wait`：

- `wait` 会阻塞父进程，直到**任一子进程**退出。
- 如果父进程有多个子进程，`wait` 会阻塞并等待**任意一个子进程**的退出，然后返回退出的子进程的PID。
- 如果没有子进程，`wait` 会立即返回 `-1`。

`waitpid`：

- `waitpid` 也会阻塞父进程，但可以通过设置不同的参数来控制它的行为。
- 如果不使用 `WNOHANG` 标志，`waitpid` 会阻塞并等待指定的子进程退出，类似于 `wait`。
- 如果使用 `WNOHANG` 标志，`waitpid` 会立即返回，不会阻塞。如果子进程没有退出，返回值是 `0`，父进程可以继续执行。

11.waitpid循环结束 子进程

wait简单，没有那么多参数，直接外层加个循环就行 ，有几个子进程就循环几次

阻塞 与 非阻塞 都可以

非阻塞循环，一定要注意

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
    int n = 5;
    int i = 0;
    int epid = 0;
    pid_t wpid;

    while (i<5)
    {
        pid_t pid = fork();
         if(i == 2)
            {
                epid = pid;    // 父进程从这里 获得子进程 pid
            }
        if (pid < 0)
        {
            perror("fork error!");
            exit(1);
        }
        if(pid == 0)
        {
            break;
        }
        else if(pid > 0)
        {
```

```
            i++;
            // printf("parent: my pid is %d \n",  getpid());
            continue;
        }
    }

    if(i==5)
            {
                sleep(5);
                //while((wpid = waitpid(-1, NULL, 0)))  // 阻塞方式 waitpid 等待所有子进程退出
                //{
                  //  printf("child waitpid %d\n", wpid); // 循环返回-1,就是失败了, 也算是 子进程都退出了
                //}
                while((wpid = waitpid(-1, NULL, WNOHANG)) != -1)  // 非阻塞方式 如果想等, 就循环
                {
                    if(wpid > 0)    // 大于0 是正常退出的
                    {
                    printf("child waitpid %d\n", wpid);

                    }
                    else // =0 是未退出, 想继续等, 就循环
                    {
                        continue;
                    }
                }



            }
    else
            {
                if(i==2)
                {
                    sleep(5);
                }
                sleep(i);
                printf("child %d \n", i);
                printf("child %d pid is %d \n", i, getpid());
                printf("child %d parent pid is %d \n", i, getppid());
            }

    // sleep(30);  // 即使是不等待子进程, 直接退出, 子进程会变为孤儿进程
    return 0;
}

```

day6 下 进程通信

IPC（Inter-Process Communication，进程间通信）

进程间通信的常用方式，特征：

```
管道：简单，有血缘关系的

信号：开销小，数据量有限，快

mmap映射：非血缘关系进程间

socket（本地套接字）：稳定     实现复杂度高
```

管道：　（日常英文中：pipeline）

计算机中是 pipe

```
实现原理：  内核借助   环形队列机制，使用内核缓冲区实现。
pipe 系统函数即可实现
命令行 是 mkfifo 创建 伪文件，文件类型 是 p

特质；  1. 伪文件

    2. 管道中的数据只能一次读取。

    3. 数据在管道中，只能单向流动。

局限性：1. 自己写，不能自己读。

    2. 数据不可以反复读。

    3. 半双工通信。（只能 全部往左,或者全部往右    还有单工,全双工）

    4. 血缘关系进程间可用。
```

pipe函数实现管道：　　创建，并打开管道。

```
<unistd.h>
int pipe(int fd[2]);   这是个数组

参数：  fd[0]：读端。    man里有

    fd[1]：写端。

返回值：  成功：  0

     失败：  -1 errno
```

由于是 半双工，在man的例子里，一个读一个写 各自进行时，都需要关闭 另一端

管道的读写行为：

```
读管道：
    1. 管道有数据，read返回实际读到的字节数。

    2. 管道无数据：    1）无写端，read返回0  （类似读到文件尾）

            2）有写端，read阻塞等待。

写管道：
    1. 无读端(无读端的文件描述符)，  异常终止。  （SIGPIPE导致的）

    2. 有读端：  1）  管道已满(高版本内核,会自动扩容)，  阻塞等待

            2）  管道未满，  返回写出的字节个数。

管道的文件描述符：

pipefd[0]：这是管道的读端文件描述符，用来从管道中读取数据。
pipefd[1]：这是管道的写端文件描述符，用来向管道中写入数据。
```

// 下面 略 时作业

pipe管道： 用于有血缘关系的进程间通信。   ps aux | grep

ls | wc -l

```
父子进程间通信：
```

```
兄弟进程间通信：
```

da6 下 补充与实例

1.linux文件类型

普通文件（-）： 普通数据文件，如文本、程序等。

目录文件（d）： 用于组织文件和子目录。

符号链接文件（l）： 指向另一个文件或目录的快捷方式。

-------以上三种， 是真是需要 硬盘空间 存储----

-------以下4种， 一般称为 伪文件,由操作系统内核或其他进程动态生成或管理---

字符设备文件（c）： 面向字符流的设备文件（如 `/dev/null`）。

块设备文件（b）： 面向块的设备文件（如 `/dev/sda`）。

套接字文件（s）： 用于进程间通信的文件（如 `/var/run/docker.sock`）。

管道文件（p）： 用于进程间通信的 FIFO 文件。

2.pipe 实例-1

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int pipefd[2];
    int ret;
    pid_t pid;
    char *bufer = "hello world!";  //字符数组,要加/0,才是正规的字符串
    char buf[20];

    if(pipe(pipefd) == -1)   // pipe b必须初始化
    {
        perror("pipe error");
        exit(1);
    }
    pid = fork();
    if(pid < 0)
    {
        perror("fork error");
        exit(1);
    }
    else if(pid == 0)
    {
        close(pipefd[1]);
        ret = read(pipefd[0], buf, 20);
        write(1, buf, ret);
        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
    }
    else if(pid > 0)
    {
        close(pipefd[0]);
        printf("%ld\n",strlen(bufer));  //strlen 不读\0
        write(pipefd[1], bufer, strlen(bufer));
        close(pipefd[1]);
        wait(NULL);
    }

    return 0;
}
```

3.char * 和char []

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    char ch[] = "hello"; // 字符串字面量，会加\0结尾，若是['a',[c]] 这种，就没有0  ---- 可被修改值
    char *ptr = "hello";  //值不能被修改，只能重新指向 另一个字符串

    ch[2] = 'a';
    printf("ch is %s, sizeof is %ld, strlen is %ld\n",ch,sizeof(ch),strlen(ch) );
    printf("ptr is %s, sizeof is %ld, strlen is %ld\n",ptr,sizeof(ptr),strlen(ptr) );

    return 0;
}
ch is healo, sizeof is 6, strlen is 5
ptr is hello, sizeof is 8, strlen is 5

说明:
strlen  不会读 \0 , sizeof 在读 字符数组时，会读\0，读字符指针,则是指针的大小
    strlen 必须传入字符串，不管是什么方式 定义的，传入即可，不管是什么方式，都表示 指针
ch[] 中 ch 是起始地址
char *ptr ptr是起始地址

c++ 11及以后,认为char* 类型的字符串不安全，必须使用const char*
```

3.命令行的 管道 |

命令行中的 `|` 管道和进程间管道（IPC中的管道）在概念上是相关的，但它们的作用和实现有所不同

4.pipe注意点

管道是 有文件描述符的， 父子进程 共享

在使用 `pipe` 创建管道后，父进程和它的所有子进程 `都会继承管道的读端和写端文件描述符` ，除非这些描述符被显式关闭。这种行为是因为在 `fork` 时，子进程会复制父进程的文件描述符表，而管道描述符也是其中的一部分。

5.兄弟进程通信 实例-1

wc 这个命令，不关闭 所有的写端，会阻塞 具体看day7

课程讲的是错的，这里要注意

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
```

```c
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    int ret, pipefd[2];
    int i = 0 ;
    pid_t pid;
    if(pipe(pipefd) == -1)   // pipe b必须初始化
    {
        perror("pipe error");
        exit(1);
    }
    while(i<2)
    {
        pid = fork();
        if(pid < 0)
        {
            perror("fork error");
            exit(1);
        }
        if (pid > 0)
        {
            i++;
            continue;
        }
        else if(pid == 0)
        {
            break;
        }
    }

    if(i == 2)
    {
        close(pipefd[0]);
        close(pipefd[1]);    // 仅关闭 写端 就可以
        wait(NULL);  // 一次调用,只等待一个
        wait(NULL);
    }
    else if(i == 1) // 弟
    {
        close(pipefd[1]);
        dup2(pipefd[0],0); //标准输入 指向 读端
        execlp("wc","wc","-l",NULL);
        perror("execlp error");
        exit(1);
    }
    else if(i == 0)  // 兄
    {
        close(pipefd[0]);
        dup2(pipefd[1],1);  //写端 指向 标准输出    错的, 是标准输出指向写端
        /*标准输出 (stdout, 文件描述符 1) : 默认连接到终端 (屏幕) , 用于正常输出
        也就是说, ls 默认会使用标注输出 连到写到屏幕, 而当 标准输出 指向写端时, 就连到
到写端了*/
```

```
        execlp("ls","ls","-l",NULL);// 这个 不用加判断，因为如果成功，所有代码就被丢弃，换成新的 程序代码
        perror("execlp error");
        exit(1);
    }

    return 0;
}

// 仅输出 行数
```

6.兄弟进程通信 实例-2

```
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <string.h>
#include <stdlib.h>

int main(void)
{
    pid_t pid;
    int fd[2], i, n;
    char buf[1024];

    int ret = pipe(fd);
    if(ret == -1){
        perror("pipe error");
        exit(1);
    }

    for(i = 0; i < 2; i++){
        if((pid = fork()) == 0)
            break;
        else if(pid == -1){
            perror("pipe error");
            exit(1);
        }
    }

    if (i == 0) {
        close(fd[0]);
        write(fd[1], "1.hello\n", strlen("1.hello\n"));
    } else if(i == 1) {
        close(fd[0]);
        write(fd[1], "2.world\n", strlen("2.world\n"));
    } else {
        close(fd[1]);        //父进程关闭写端,留读端读取数据
        sleep(1);
        n = read(fd[0], buf, 1024);      //从管道中读数据
        write(STDOUT_FILENO, buf, n);

        for(i = 0; i < 2; i++)       //两个儿子wait两次
```

```
            wait(NULL);
    }

    return 0;
}

```

day 7

day7 上

pipe 补充与pipe 纠错

> 是否允许，一个pipe有一个写端，多个读端呢？ 允许
>
> 是否允许有一个读端多个写端呢？ 允许
>
> sleep 可以保证顺序，不然 可能 提前结束啥的

例子见 上面那两个 兄弟通信

这里有点乱

管道的工作原理简述

- **写端未关闭**：当你通过管道写数据时，数据会被存储在管道的缓冲区中。管道的读端会从缓冲区读取数据。**如果写端没有关闭，读端可能会继续等待更多数据的到来，直到所有写端都关闭**，以标记数据的结束（即 EOF）。
- **EOF（文件结束标志）**：在管道中，只有当**所有写端都关闭**时，读端才会接收到 EOF，标志着数据的结束。

关于 `wc` 的情况

- `wc` 是一个命令，用于统计文本的行数、字数、字符数等。当我们将数据通过管道传递给 `wc` 时，它会读取这些数据并进行计算。
- **如果写端没有关闭**，即使数据已经传递给管道，`wc` 会继续等待更多数据的到来。因为它无法知道管道的写端是否已经结束数据写入，直到所有写端都关闭，它才会输出结果。

关于父进程读取数据的情况

- 在你的例子中，父进程可以从管道中读取数据，**即使子进程没有关闭所有的写端**。这是因为父进程可以从管道的缓冲区中读取已写入的数据。如果子进程写入的数据已经存储在管道中，父进程可以读取这些数据，而不会因为写端没有关闭而阻塞。
- **管道的读端**可以在写端未关闭的情况下读取到缓冲区中的数据，但只有当**所有写端都关闭时**，管道才会向读端传递 EOF（标志着没有更多数据可供读取）。

总结对比：

- `wc` **程序行为**：如果没有关闭写端，`wc` 会一直阻塞等待更多数据的写入，直到所有写端关闭并且没有更多数据时，才会开始处理数据并输出结果。

- **父进程读取数据的行为**：父进程可以读取管道中的数据，甚至在写端未关闭的情况下，只要数据已经写入并保存在管道缓冲区中。读端并不会立即阻塞，除非管道中的数据已经完全读完，或者没有更多的数据待写入。

所以，**关键区别**是：

- `wc` **进程**：会阻塞，直到所有写端关闭并且数据传输完成后才输出结果。
- **父进程读取数据**：可以读取已经写入管道的数据，直到管道的缓冲区为空或者写端关闭。

管道缓冲区：

`ulimit -a` 命令行

和

`fpathconf函数` ----- 查 man

fifo管道：可以用于无血缘关系的进程间通信。

```
命名管道:   mkfifo

使用 mkfifo 命令或者 C 语言的 mkfifo() 系统调用。

fifo是一种 基础文件类型，按使用文件的方式，使用即可

无血缘关系进程间通信:

    读端，open fifo O_RDONLY

    写端，open fifo O_WRONLY
```

文件实现进程间通信：

已经被淘汰

```
打开的文件是内核中的一块缓冲区。多个无血缘关系的进程，可以同时访问该文件。
```

共享内存映射：（重点）

`Memory-mapped I/O`

把 硬盘上的东西 映射到 内存上：

```
内存映射 I/O (Memory-Mapped I/O, MMIO) 是一种通过将硬件设备的寄存器映射到系统内存地址空间来进行设备输入输出 (I/O) 操作的技术。它使得程序可以像访问内存一样访问设备寄存器，从而简化了I/O操作，并且提高了性能。
```

为什么呢?

在硬盘上，只能通过 open. read 等 去读取

在内存上，就有了 地址，通过地址 访问(指针)，效率更高

mmap函数：　　实现内存映射

memory map 内存映射

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);
创建共享内存映射

```
<sys/mman.h>   注意,不是map.h

参数:
    addr:    指定映射区的首地址。通常传NULL，表示让系统自动分配

    length: 共享内存映射区的大小。（≤ 文件的实际大小）

    prot:    共享内存映射区的读写属性。PROT_READ、PROT_WRITE、PROT_READ|PROT_WRITE(位或进行读写)    还有 PROT_EXEC,PROT_NONE
    PROT : 表示  protect, 保护

    flags:   标注共享内存的共享属性。MAP_SHARED、MAP_PRIVATE  表示 共享内存是否是共享的    私有表示 对内存的修改 不会反映到 磁盘上

    fd: 用于创建共享内存映射区的那个文件的 文件描述符。

    offset: 默认0，表示映射文件全部。偏移位置。需是 4k 的整数倍。

返回值:

    成功: 映射区的首地址。(注意 void*)

    失败: MAP_FAILED (void*(-1)),  errno     这么理解: 原本是 -1, 但受限于 void* 类型，将其定义为了一个 常量 宏
```

补充-1

1.void*

`void *` 是 C 语言中的一种指针类型，表示一个**通用指针**，可以指向任何类型的数据。它是一个**类型不确定的指针**，意味着它没有特定的类型限制。

解释:

- `void *` 是一个指向 **不确定类型** 数据的指针，你可以将它赋值为指向任何类型数据的指针，如 `int *`、`char *`、`float *` 等。
- 由于 `void *` 没有具体的类型信息，所以它不能直接解引用（即不能直接使用 `*` 操作符），必须先将其转换为具体类型的指针才能进行访问。

```c
void *ptr;
int num = 10;
ptr = &num;  // void pointer pointing to an int variable



// 类型转换
int *int_ptr = (int *)ptr;  // 将 void 指针转换为 int 指针
printf("%d\n", *int_ptr);   // 输出 10

```

2.od -tcx

`od` 是 Linux 和类 Unix 系统中的一个命令行工具，用于显示文件或标准输入的数据的八进制、十六进制、二进制等格式。它可以帮助用户查看原始数据，尤其是在调试和分析文件内容时非常有用。

3.实例-1

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>

void sys_err(const char *str)
{
    perror(str);
    exit(1);
}

int main(int argc, char *argv[])
{
    char *p = NULL;
    int fd;

    fd = open("testmap", O_RDWR|O_CREAT|O_TRUNC, 0644);      // 创建文件用于创建映射区
    if (fd == -1)
        sys_err("open error");
/*
    lseek(fd, 10, SEEK_END);            // 两个函数等价于 ftruncate()函数
    write(fd, "\0", 1);
*/
    ftruncate(fd, 20);                  // 需要借助写权限,才能够对文件进行拓展
    int len = lseek(fd, 0, SEEK_END);

    p = mmap(NULL, len, PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) {
        sys_err("mmap error");
    }
```

```
35
36      // 使用 p 对文件进行读写操作.
37      strcpy(p, "hello mmap");                // 写操作
38
39      printf("----%s\n", p);                  // 读操作
40
41      int ret = munmap(p, len);               // 释放映射区
42      if (ret == -1) {
43          sys_err("munmap error");
44      }
45
46      return 0;
47  }
```

解除映射： munmap

memory unmap　缩写

int munmap(void *addr, size_t length);　　释放映射区。

```
<sys/mman.h>

addr: mmap 的返回值

length: 大小

成功: 0
失败: -1 errno
```

mmap使用注意事项：

问题:

```
思考:
1. 可以open的时候O_CREAT一个新文件来创建映射区吗?
2. 如果open时O_RDONLY, mmap时PROT参数指定PROT_READ|PROT_WRITE会怎样?
3. 文件描述符先关闭, 对mmap映射有没有影响?　这个问题 是 mmap后直接关闭fd, 而不是mmap前关闭 fd
4. 如果文件偏移量为1000会怎样?
5. 对mem越界操作会怎样?
6. 如果mem++, munmap可否成功?
7. mmap什么情况下会调用失败?
8. 如果不检测mmap的返回值, 会怎样?
```

回答:

```
1. 用于创建映射区的文件大小为 0, 实际指定非0大小创建映射区, 出 “总线错误”。 ---SIGBUS

2. 用于创建映射区的文件大小为 0, 实际制定0大小创建映射区, 出 “无效参数”。 --invalid argument
```

```
注意：  如果是 新文件，  创建时只有 读属性，  则后续 映射时，  即使想用 写属性，也没法拿到
而且 ftruncate 无法拓展文件，  因为没有 写属性

会出现    -----   mmap error: Permission denied      “权限拒绝”

3. 用于创建映射区的文件读写属性为，只读。映射区属性为 读、写。 出 “无效参数”。

4. 创建映射区，需要read权限。当访问权限指定为 “共享”MAP_SHARED时，  mmap的读写权限，应
该 ≤文件的open权限。     open只写不行。    MAP_PRIVATE时，则不需≤

5. 文件描述符fd，在mmap创建映射区完成即可关闭。后续访问文件，用 地址访问。

6. offset 必须是 4096的整数倍。（MMU 映射的最小单位 4k    ---这是原因 ）  1k =
1024byte   4k = 4096byte    1byte = 8bit

7. 对申请的映射区内存，不能越界访问。

8. munmap用于释放的 地址，必须是mmap申请返回的地址。

9. 映射区访问权限为 “私有”MAP_PRIVATE，对内存所做的所有修改，只在内存有效，不会反应到
物理磁盘上。

10.  映射区访问权限为 “私有”MAP_PRIVATE，只需要open文件时，有读权限，用于创建映射区即
可。
```

补充-2

段错误是什么

段错误（Segmentation Fault，简称 Segfault） 是一种常见的运行时错误，通常发生在程序试图访问未经授权的内存区域时。它是由操作系统内存保护机制触发的，当程序尝试访问它没有权限访问的内存区域时，操作系统会终止程序并产生一个“段错误”。

Segmentation ： 分割， 分段 意思

段错误怎么看?

gdb 程序名   然后直接   run， 停止的地方 就是 段错误地方

mmap函数的保险调用方式：

```
1. fd = open ("文件名",  O_RDWR) ;

2. mmap(NULL, 有效文件大小,  PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
```

父子进程使用 mmap 进程间通信：

```
父进程 先 创建映射区。 open ( O_RDWR)  mmap( MAP_SHARED );

指定 MAP_SHARED 权限      若是私有，会执行 写时复制原则，copy一份，各自拥有一份

fork() 创建子进程。

一个进程读，  另外一个进程写。
```

无血缘关系进程间 mmap 通信：
【会写】

```
两个进程 打开同一个文件，创建映射区。

指定flags 为 MAP_SHARED。

一个进程写入，另外一个进程读出。

【注意】：无血缘关系进程间通信。mmap：数据可以重复读取。

                fifo：数据只能一次读取。
```

补充-3

无血缘关系进程间 mmap 通信 实例-1

```c
// 写
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <errno.h>

struct student {
    int id;
    char name[256];
    int age;
};

void sys_err(const char *str)
{
    perror(str);
    exit(1);
}

int main(int argc, char *argv[])
{
    struct student stu = {1, "xiaoming", 18};
    struct student *p;
```

```
    int fd;

//    fd = open("test_map", O_RDWR|O_CREAT|O_TRUNC, 0664);
    fd = open("test_map", O_RDWR | O_CREAT, 0664);
    if (fd == -1)
        sys_err("open error");

    ftruncate(fd, sizeof(stu));

    p = mmap(NULL, sizeof(stu), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED)
        sys_err("mmap error");

    close(fd);

    while (1) {
        memcpy(p, &stu, sizeof(stu));
        stu.id++;
        sleep(2);
    }

    munmap(p, sizeof(stu));

    return 0;
}

```

```
// 读
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <errno.h>

struct student {
    int id;
    char name[256];
    int age;
};

void sys_err(const char *str)
{
    perror(str);
    exit(1);
}

int main(int argc, char *argv[])
{
    struct student stu;
    struct student *p;
```

```
    int fd;

    fd = open("test_map", O_RDONLY);
    if (fd == -1)
        sys_err("open error");

    p = mmap(NULL, sizeof(stu), PROT_READ, MAP_SHARED, fd, 0);
    if (p == MAP_FAILED)
        sys_err("mmap error");

    close(fd);

    while (1) {
        printf("id= %d, name=%s, age=%d\n", p->id, p->name, p->age);
        usleep(10000);
    }

    munmap(p, sizeof(stu));

    return 0;
}

```

mmap和管道 区别--- 重点

mmap 是缓冲区机制， 可以反复读

管道是 队列 机制，读完就没了

匿名映射：只能用于 血缘关系进程间通信。

匿名 ： anonymous

```
p = (int *)mmap(NULL, 40, PROT_READ|PROT_WRITE, MAP_SHARED|MAP_ANONYMOUS, -1, 0);
```

优点：不需要 文件了，不需要fd 了

仅使用匿名映射，一般只能单个进程 或者 有血缘关系父子进程 可以使用

无血缘关系 进程，不能单独的 使用这个匿名映射

/dev/zero实现匿名映射

使用 `/dev/zero` 作为匿名映射的“替代”是一个常见的技术，实际上它提供了一种将映射区域初始化为零的方法。这种方法通常用于在没有文件内容的情况下，创建一块 **“初始化为零”的内存区域**。你可以将 `mmap()` 与 `/dev/zero` 结合起来，以便创建一块没有文件内容且初始化为零的内存。

> 注意： 将他的内容 映射到 内存，之后就和他没关系了，所以不要误解，该文件 确实是 写入丢弃，读是全0，但是 映射到 内存，只是 把他的 大小 拿来用
>
> 同样，不能用与 无血缘关系 通信， 因为写不进去源文件的

/dev/zero和/dev/null

`/dev/zero`：可以被称为 **“零源”** 或 **“零填充设备”**，因为它提供的是**无限的零**，用于填充或初始化内存等。

`/dev/null`：被称为 **“数据黑洞”** 或 **“空设备”**，因为它吸收一切输入的数据，读取时则什么也不会返回。

在程序员行话，尤其是Unix行话中，/dev/null 被称为位桶(bit bucket)或者黑洞(black hole)。空设备通常被用于丢弃不需要的输出流，或作为用于输入流的空文件。这些操作通常由重定向完成。

1. `/dev/zero`：
   - **作用**：提供**无限的零值**。
   - **读取**：读取时返回零。
   - **写入**：写入时会丢弃数据。
   - 常见用途：
     - **初始化内存**（如通过 `mmap()` 映射来创建零初始化的内存）。
     - **生成空文件**（通过重定向写入生成全零的文件）。
   - **示例用途**：创建匿名映射、初始化为零。
2. `/dev/null`：
   - **作用**：充当**数据的黑洞**，所有写入的数据都会被丢弃。
   - **读取**：读取时返回“空”内容（EOF）。
   - **写入**：写入的数据会被丢弃。
   - 常见用途：
     - **丢弃不需要的输出**（例如，将输出重定向到 `/dev/null`，实现“静默”运行）。
     - **作为空设备**：可以用来清空或丢弃数据。
   - **示例用途**：丢弃不需要的命令输出，模拟空输入等。

简而言之，`/dev/zero` 提供的是“零填充”，而 `/dev/null` 则是“丢弃一切”。

vim快速跳转 man对应 函数

在光标对准函数名处， shift+k 即可

day7 补充

自这里开始,简化例子，仅实现重要的例子

1.usleep

微秒， sleep 是秒

2.普通文件通信和管道通信

普通文件 通信，不会阻塞，可能出现 什么都没读到

3.父子进程文件偏移量

如果 父子进程 使用 一个相同的 文件描述符，将会共享其 文件偏移量

而 如果 各自分开，分别 open 相同的文件，得到不同的 文件描述符，就不会 共享文件偏移量，可以 直接读取，而不受 偏移量影响

实例对比：

```c
// 这是 使用同一个文件,同一个文件描述符
/*
 *父子进程共享打开的文件描述符------使用文件完成进程间通信.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>


int main(void)
{
    int fd1, fd2; pid_t pid;
    char buf[1024];
    char *str = "---------test for shared fd in parent child process-----\n";
    fd1 = open("test.txt", O_RDWR | O_CREAT | O_TRUNC, 0664);
        if (fd1 < 0) {
            perror("open error");
            exit(1);
        }
    pid = fork();
    if (pid < 0) {
        perror("fork error");
        exit(1);
    } else if (pid == 0) {

        // sleep(3);
        write(fd1, str, strlen(str));
        printf("child wrote over...\n");

    } else {
        sleep(1);
        // fd2 = open("test.txt", O_RDWR);
        // if (fd2 < 0) {
        //     perror("open error");
```

```
        //      exit(1);
        // }
//        sleep(1);                    //保证子进程写入数据

        int len = read(fd1, buf, sizeof(buf));
        printf("------parent read len = %d\n", len);
        len = write(STDOUT_FILENO, buf, len);
        printf("------parent write len = %d\n", len);

        wait(NULL);
    }

    return 0;
}
// 将什么读不到
```

对比

```
// 使用同一个文件，不同的文件描述符
/*
 *父子进程共享打开的文件描述符------使用文件完成进程间通信.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>


int main(void)
{
    int fd1, fd2; pid_t pid;
    char buf[1024];
    char *str = "---------test for shared fd in parent child process-----
\n";

    pid = fork();
    if (pid < 0) {
        perror("fork error");
        exit(1);
    } else if (pid == 0) {
        fd1 = open("test.txt", O_RDWR | O_CREAT | O_TRUNC, 0664);
        if (fd1 < 0) {
            perror("open error");
            exit(1);
        }
        // sleep(3);
        write(fd1, str, strlen(str));
        printf("child wrote over...\n");

    } else {
        sleep(1);
        fd2 = open("test.txt", O_RDWR);
```

```
        if (fd2 < 0) {
            perror("open error");
            exit(1);
        }
//        sleep(1);                    //保证子进程写入数据

        int len = read(fd2, buf, sizeof(buf));
        printf("------parent read len = %d\n", len);
        len = write(STDOUT_FILENO, buf, len);
        printf("------parent write len = %d\n", len);

        wait(NULL);
    }

    return 0;
}
```

4.父子进程fork后

fork后，是各自使用各自的 pcb进程控制块，因此,下面这个程序， fd2和fd3 都是 4

```
/*
 *父子进程共享打开的文件描述符------使用文件完成进程间通信.
 */
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/wait.h>


int main(void)
{
    int fd1, fd2, fd3; pid_t pid;
    char buf[1024];
    char *str = "---------test for shared fd in parent child process-----
\n";
    fd1 = open("test.txt", O_RDWR | O_CREAT | O_TRUNC, 0664);
        if (fd1 < 0) {
            perror("open error");
            exit(1);
        }
    printf("fd2 initial value is %d...\n", fd2);
    pid = fork();
    if (pid < 0) {
        perror("fork error");
        exit(1);
    } else if (pid == 0) {

        printf("child and parent fd1 is %d...\n", fd1);
        fd2 = open("test.txt", O_RDWR | O_CREAT | O_TRUNC, 0664);
```

```c
        if (fd2 < 0) {
            perror("open error");
            exit(1);
        }
        printf("child fd2  is %d...\n", fd2);
        sleep(5);
    } else {
        // sleep(1);
        // fd2 = open("test.txt", O_RDWR);
        // if (fd2 < 0) {
        //     perror("open error");
        //     exit(1);
        // }
//         sleep(1);                    //保证子进程写入数据
        printf("child and parent fd1 is %d...\n", fd1);
        fd3 = open("test.txt", O_RDWR | O_CREAT | O_TRUNC, 0664);
        if (fd3 < 0) {
            perror("open error");
            exit(1);
        }
        printf("parent  fd3  is %d...\n", fd3);
        printf("parent: child  fd2  is %d...\n", fd2); // 拿不到子进程的 fd2, 还是初始值

        wait(NULL);
    }

    return 0;
}

```