操作系统

2.4 进程通信

朱小军

南京航空航天大学计算机科学与技术学院

2025年春

从程序员的角度回顾进程知识

- 系统调用
  - **fork()**创建新进程
    - 调用一次，返回**2**次
  - **exit()**终止进程自身
    - 调用一次，从不返回
    - 将进程放入“僵尸”状态（不在三状态中）
  - **wait()**等待并收割**(reap)**子进程
  - **exec()**在已经存在的进程中运行新的程序 //发音[‘eg zek]
    - 调用一次，从不返回
- **shell**
  - 内置命令，直接执行；
  - 外部命令，调用**fork()**，在子进程中**exec()**，父进程**wait**（）
    - 有时，父进程也不执行**wait**（），何时？（后台进程）

目录

看程序，写输出

```
int i=0;
void main(){
    i++;
    if(fork()==0){//child
        i++;
        printf("child: i=%d\n",i);
        exit();
    }else{//parent
        wait();
        printf("parent: i=%d\n",i);
    }
}
```

A
```
child: i=1
parent: i=1
```

B
```
child: i=2
parent: i=1
```

C
```
child: i=2
parent: i=2
```

投票 最多可选1项

设置

右侧程序的输出是?

```
int i=0;
void main(){
    i++;
    if(fork()==0){//child
        i++;
        printf("child: i=%d\n",i);
        exit();
    }else{//parent
        wait();
        printf("parent: i=%d\n",i);
    }
}
```

A
child: i=1
parent: i=1

B
child: i=2
parent: i=1

C
child: i=2
parent: i=2

提交

上面的例子有实际需求吗？

网页服务器的单进程与多进程实现

单进程版本：

```
while(true){
        conn=accept( );
        serve( conn );
}
```

缺点：服务一个客户端时，其他客户无法连接

多进程版本：

```
 while( true){
        conn=accept( );
        创建新进程执行serve( conn );
}
```

缺点：连接太多时服务器会奔溃！

限定进程数量

定义变量proc_num记录服务的客户数量

```
while( true){
        conn=accept( );
        while(proc_num≥给定值);
        proc_num++;
        创建新进程执行serve( conn );
}
```

新进程退出时应当执行proc_num--;

单选题 1分

设置

右侧程序长时间运行，第10个及以后的客户端连接时会出现哪种后果？

```
int proc_num=0;//全局变量，在数据区
void main(){
    while(true){
        while(proc_num>=10);//循环等
        conn=accept();//等待客户连接
        proc_num++;//进程数加1
        if(fork()==0){
            serve(conn);//服务客户
            proc_num--;//进程数减1
            exit();
        }
    }
}
```

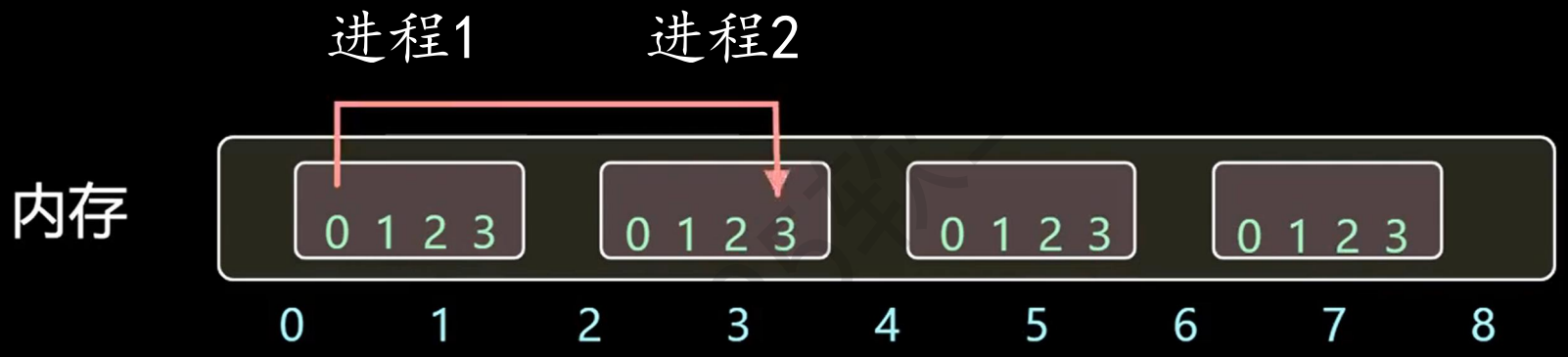
A 客户连不上，因为服务器不执行accept

B 客户能连上，但是无服务，因为服务器不执行serve

C 一切正常

提交

进程1如何访问进程2的虚拟内存？

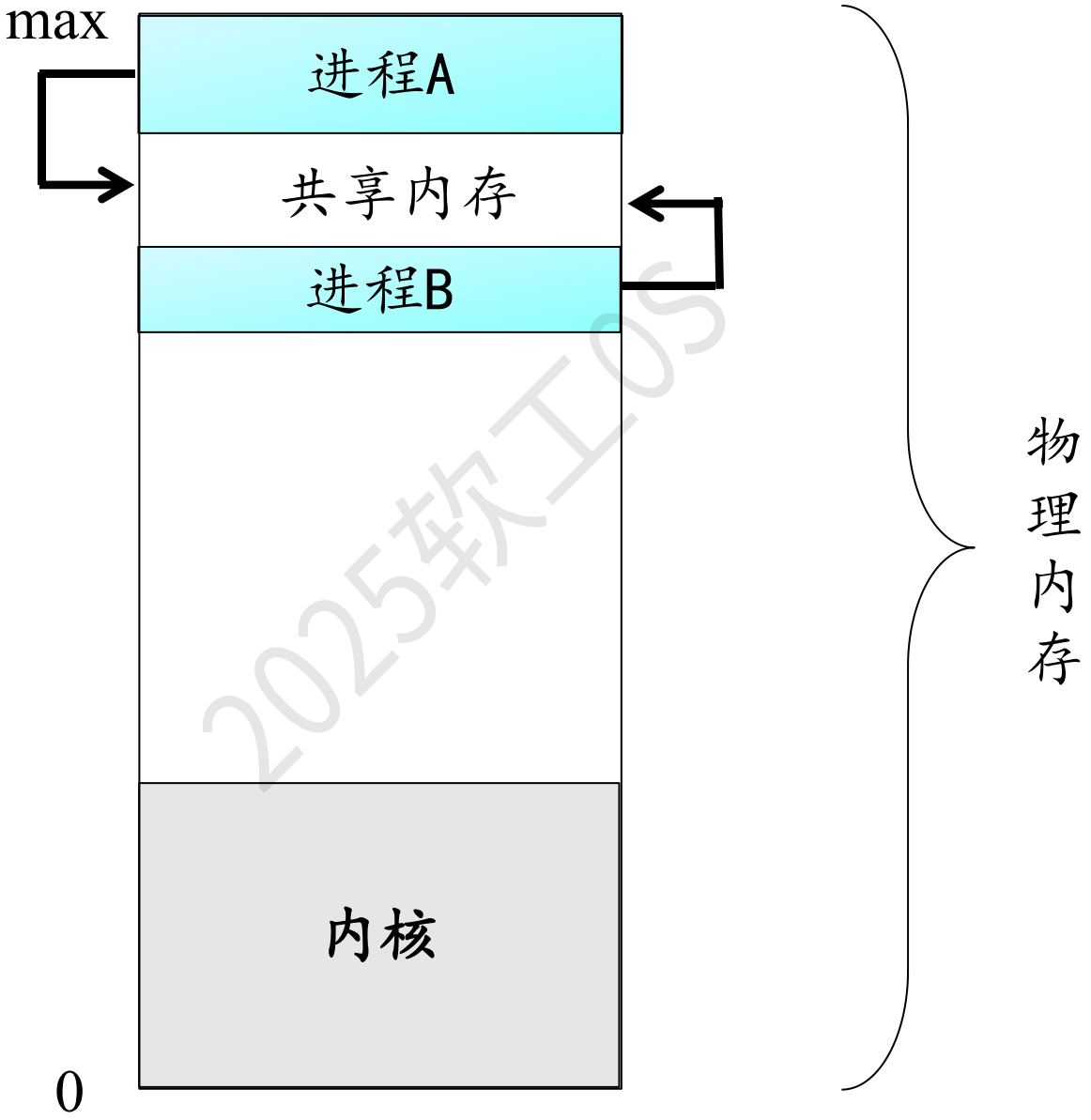


不同进程如何共享变量？

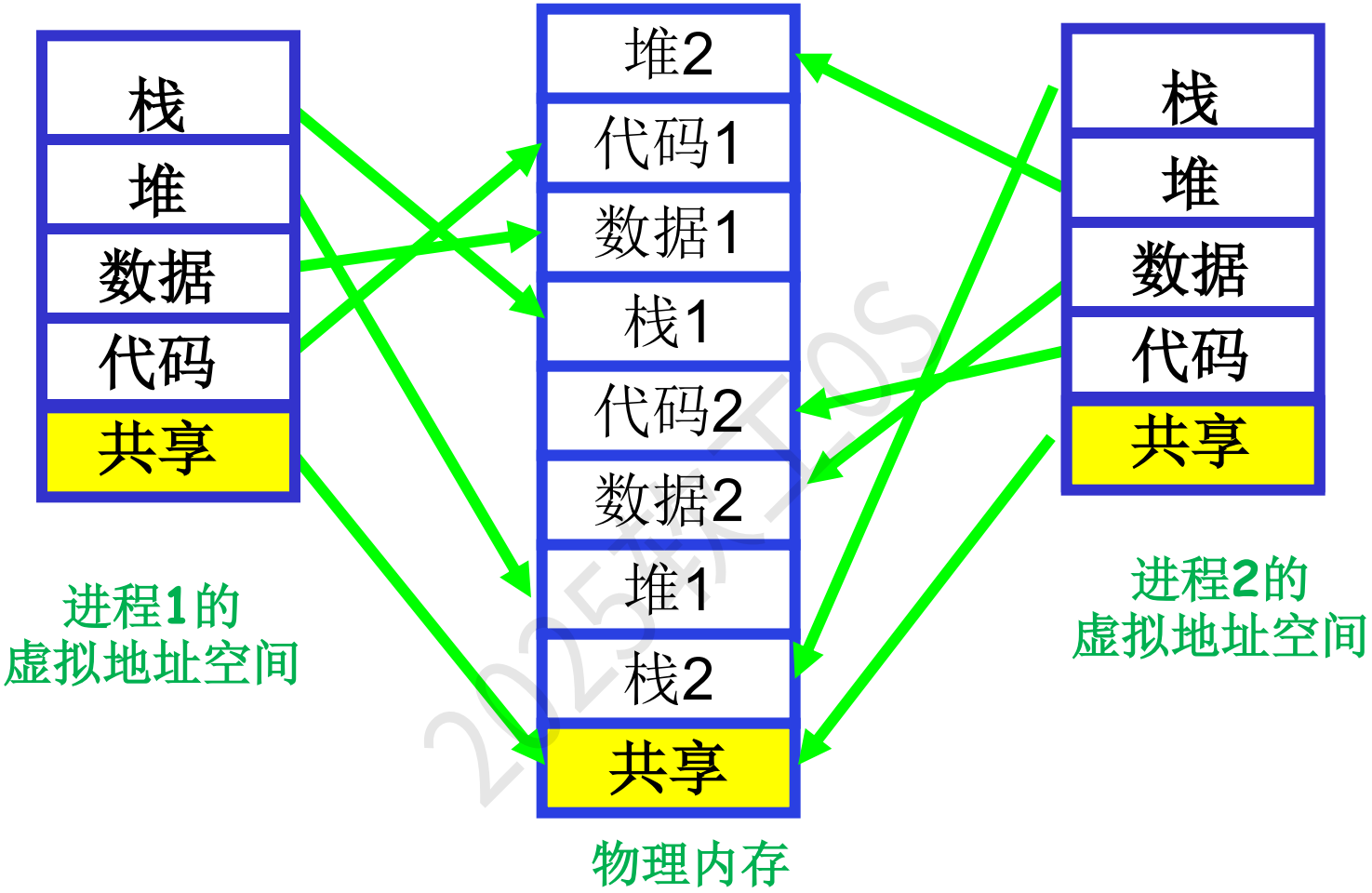
目录

- 为什么OS要提供进程通信机制？
- 共享内存
- 消息传递
- 管道
- 信号

共享内存

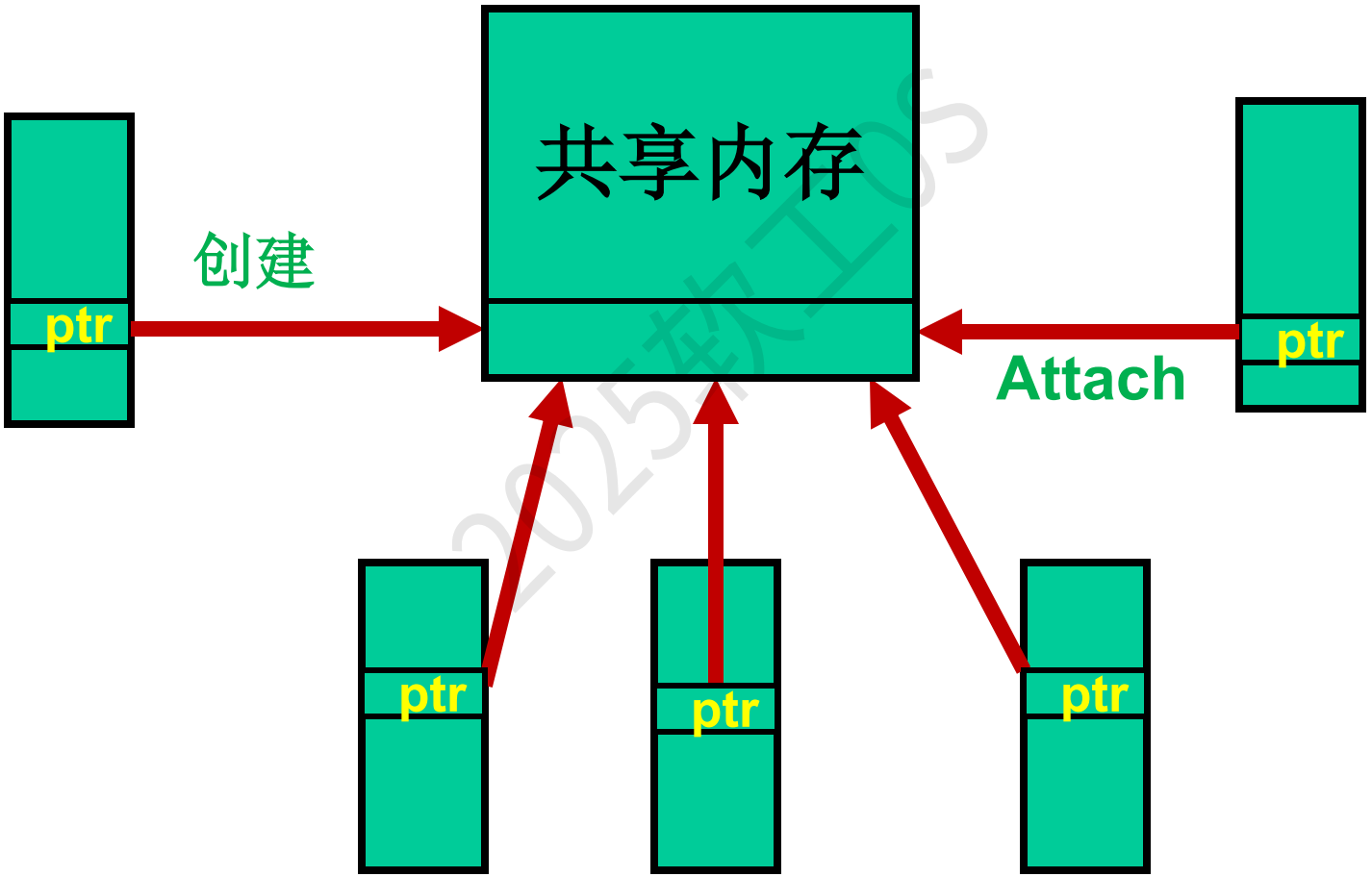


共享内存的实现



- 直接读写共享存储区即可
- 同步问题（？后面会讲）需要自行处理

共享内存的使用



POSIX共享内存

- **POSIX Shared Memory**
  - 首先创建共享内存

    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    - 同样用于打开已有的共享内存
  - 设置共享内存大小

    ftruncate(shm_fd, 4096);
  - 将共享内存映射到进程的地址空间

    ptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd,0);
  - 向共享内存写

    sprintf(shared memory, "Writing");

如何让两个进程共享一个变量？

```
int i=0;
void main(){
        i++;
        if(fork()==0){//child
                i++;
                printf(“child: i=%d\n”,i);
                exit();
        }else{//parent
                wait();
                printf(“parent: i=%d\n”,i);
        }
}
```

原始无法共享变量的程序

```
int i=0;
void main(){
        i++;
        int shm_fd; void * ptr;
        shm_fd=shm_open("oscourse",O_CREAT|O_RDWR,0666);
        ftruncate(shm_fd,4096);
        ptr=mmap(0,4096,PROT_READ|PROT_WRITE,MAP_SHARED,shm_fd,0);
        if(fork()==0){//child
                i++;
                *(int *)ptr=i;
                printf("child: i=%d\n",i);
                exit();
        }else{//parent
                wait();
                i=*(int *)ptr;
                printf("parent: i=%d\n",i);
        }
}
```

改造后的程序

运行结果

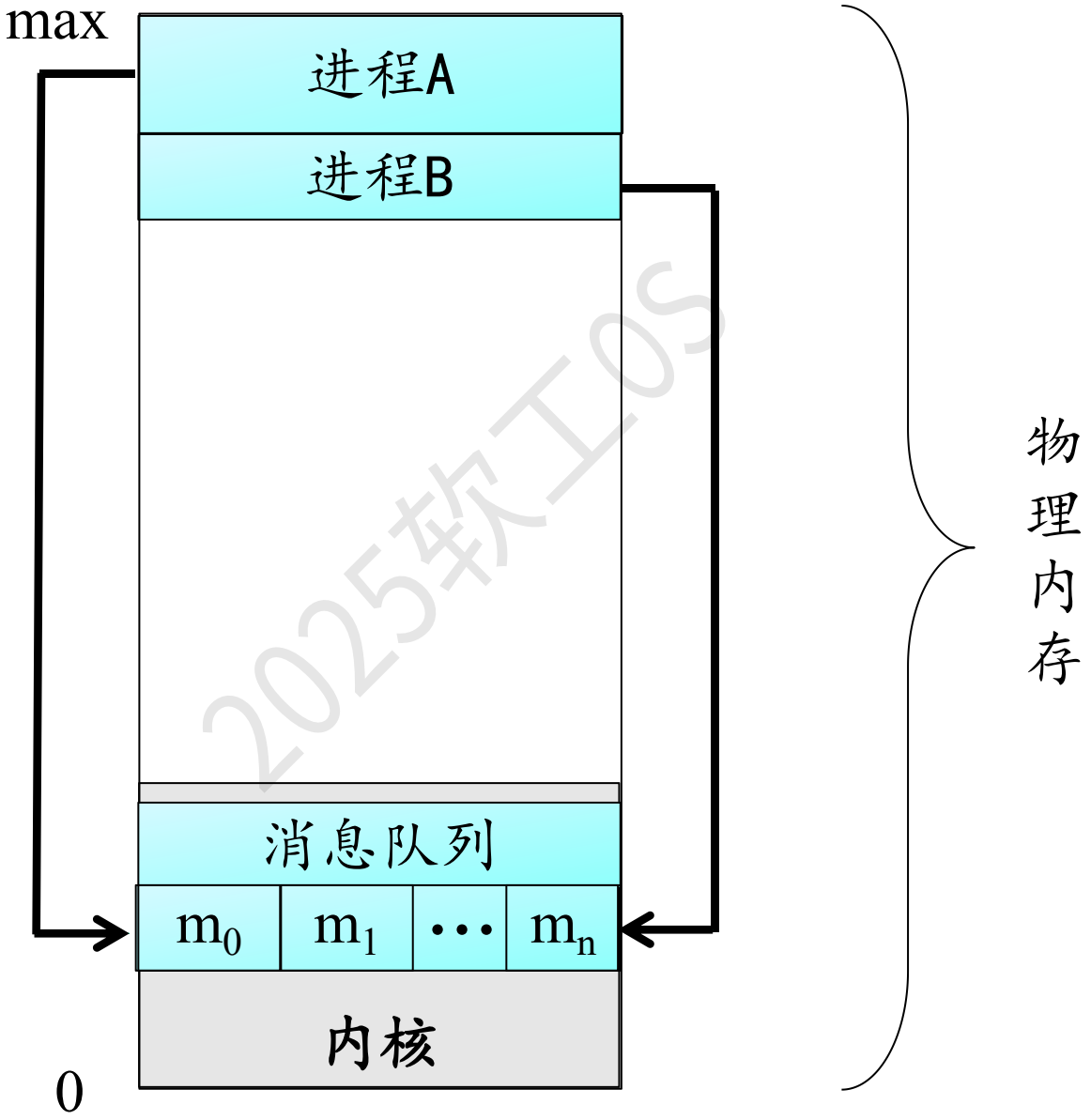
- 注意
  - 加头文件"fcntl.h"、"sys/mman.h"
  - gcc *.c -lrt //与rt库链接

```
ubuntu@VM-16-15-ubuntu:~$ gcc -m32 share_variable_via_shared_memory.c -o share -lrt
share_variable_via_shared_memory.c: In function ‘main’:
share_variable_via_shared_memory.c:26:3: warning: implicit declaration of function ‘wait’
tion]
   26 |   wait();
      |   ^~~~
ubuntu@VM-16-15-ubuntu:~$ ./share
child: i=2
parent: i=2
```

目录

- 为什么OS要提供进程通信机制？
- 共享内存
- 消息传递
- 管道
- 信号

消息传递

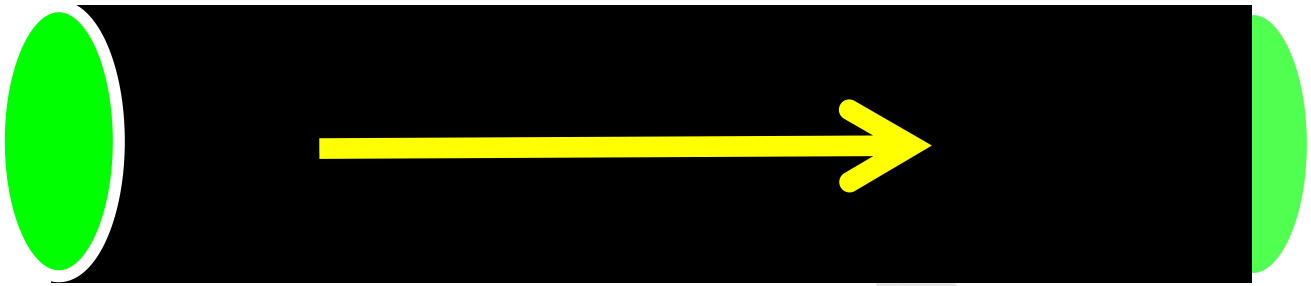


消息传递的实现

- 操作系统提供两个系统调用
  - **send(P, msg):** 发送**msg**到进程**P**
  - **receive(Q, msg)**：从进程**Q**接收消息存到**msg**
- 阻塞与非阻塞（或：同步与异步）
  - 阻塞：发出系统调用的进程变为阻塞状态，直到消息被对方接收或者接收到对方的消息
  - 非阻塞：消息发送或接收成功与否均立即返回
- 消息定长或变长
  - 定长易实现，不易使用；变长相反
- 信箱/邮箱 **mailbox**

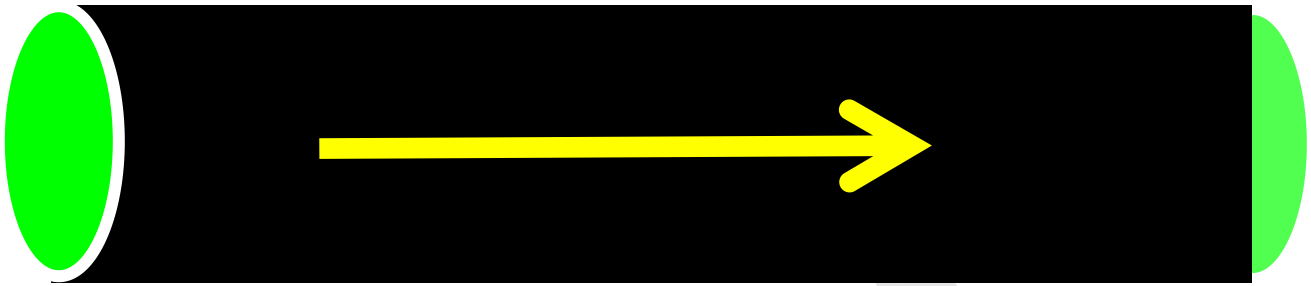
目录

管道 pipe



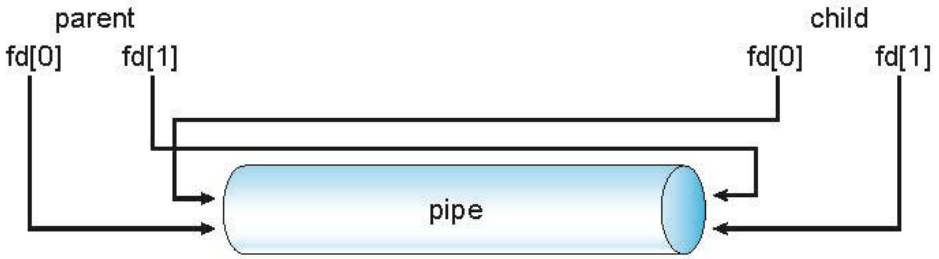
- 简单、高效、被大量操作系统采用
- 特殊的文件或缓冲区，大小一般固定
  - 初始化时指定
- 先入先出，即，FIFO
- 被大量操作系统用于将一个程序的输出重定向为另一个程序的输入

管道 pipe



- 读写特征
  - 若管道已满，则试图写的进程被阻塞，直到有进程将部分数据取走
  - 若管道为空，则试图读的进程被阻塞，直到有进程写入新的数据
  - 若无人使用管道的另一侧，则阻塞的进程被唤醒

两类管道



- 无名管道（unnamed pipe）
  - 仅用于有相互关系的进程之间
  - 父进程/子进程，子进程/子进程
  - 使用管道的所有进程死亡时自动消亡
- 命名管道（named pipe）
  - 作为一个文件存在
  - 有文件的各种操作属性
  - 可以被不相关的进程所使用

管道案例

- Linux下，“|”，比如cat filename | less
- 利用管道，子进程计算、父进程读结果

```
void main() {
    int fd[2]; // 子进程写，父进程读
    pipe(fd);// 创建管道
    if (fork() > 0) { // 父进程
        close(fd[1]); // 关闭写端
        int result;
        read(fd[0], &result, sizeof(result));
        printf("Parent received result: %d\n", result);
        close(pipe_child_to_parent[0]);
        wait(); // 等待子进程退出
    } else { // 子进程
        close(fd[0]); // 关闭读端
        int square = 5 * 5;
        write(fd[1], &square, sizeof(square));// 子进程计算，结果写入管道
        close(fd[1]); // 关闭写端
        exit(0);
    }
}
```

如何实现管道？

- xv6的做法
  - 从内核中开辟一块区域，4KB大小，除去存储数据的部分，还需要记录
    - 用于同步的锁
    - 读写位置
    - 读写权限

```
#define PIPESIZE 512

struct pipe {
  struct spinlock lock;
  char data[PIPESIZE];
  uint nread;     // number of bytes read
  uint nwrite;    // number of bytes written
  int readopen;   // read fd is still open
  int writeopen;  // write fd is still open
};
```

目录

信号signal

- shell上运行的后台进程怎么处理？
  - 终止时变成“僵尸”状态
  - 永不会被收割，因为父进程shell不终止（能杀死shell吗？）
  - 造成内存泄漏，导致内核崩溃

- 解决方案：用信号机制
- 典型的（从键盘发出的）信号
  - ctrl+c　//SIGINT　中断
  - ctrl+z　//SIGSTP　挂起

信号signal

- 信号：操作系统通知进程的机制
  - 回忆进程通知操作系统的机制是什么？
  - 从内核发出（可能源于进程请求）到一个进程
  - 信号用小的整数标识
  - 信号中的信息只包括：ID和信号到达

ID	名称	默认行为	对应的事件
2	SIGINT	Terminate	Interrupt from keyboard (ctrl-c)
9	SIGKILL	Terminate	Kill program (cannot override or ignore)
11	SIGSEGV	Terminate & Dump	Segmentation violation
14	SIGALRM	Terminate	Timer signal
17	SIGCHLD	Ignore	Child stopped or terminated
20	SIGSTP	Suspend/Stop	Interrupt from keyboard (ctrl-z)

信号发送

- 内核向进程发送信号的方式
  - 修改进程的内核数据结构，如PCB中的某些域

- 发送原因
  - 检测到了系统事件，如除以0（SIGFPE），子进程结束（SIGCHLD），键盘发出了特殊中断（SIGINT ctrl-c，SIGSTP ctrl-z，SIGQUIT ctrl-\）
  - 其他进程通过kill系统调用要求内核向指定的进程发送信号

信号接收

- 进程被迫接收信号，必须处理信号
- 三种可能的处理方式
  - 忽略信号
  - 终止进程
  - 捕获信号，执行一个用户态的信号处理程序
- 每种信号有默认的处理方式，有的可以被覆盖，有的不可以
- 有的信号可以被忽略，有的不可以被忽略，如**SIGKILL**不可以被忽略、不可以被覆盖
  - 如何杀死一个进程？**kill -9 <pid>**

信号何时被处理？

- 从内核态返回用户态前，检查是否有信号，如果有，执行相应的程序

```
96
97    // Force process exit if it has been killed and is in user space.
98    // (If it is still executing in the kernel, let it keep running
99    // until it gets to the regular system call return.)
100   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
101     exit();
102
103   // Force process to give up CPU on clock tick.
104   // If interrupts were on while locks held, would need to check nlock.
105   if(myproc() && myproc()->state == RUNNING &&
106      tf->trapno == T_IRQ0+IRQ_TIMER)
107     yield();
108
109   // Check if the process has been killed since we yielded
110   if(myproc() && myproc()->killed && (tf->cs&3) == DPL_USER)
111     exit();
112 }
113
```

xv6中的trap函数，在结尾处理了kill

示例：屏蔽ctrl-c信号

```c
// 自定义信号处理函数
void handle_signal(int sig) {
    const char* msg = "Don't kill me!\n";
    write(STDOUT_FILENO, msg, 14);  // 直接使用 write 避免 printf 的可重入性问题
}

int main() {
    // 定义信号处理结构体
    struct sigaction sa;
    sa.sa_handler = handle_signal;  // 指定处理函数
    sigemptyset(&sa.sa_mask);       // 清空信号屏蔽集（不阻塞其他信号）
    sa.sa_flags = 0;                // 无特殊标志

    // 注册 SIGINT (Ctrl+C) 的处理方式
    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("sigaction");
        return 1;
    }

    // 保持程序运行
    printf("Press Ctrl+C to test. Use Ctrl+\\ to quit.\n");
    while (1) {
        pause();  // 等待信号
    }

    return 0;
}
```

回答前面的问题

- shell如何处理后台僵尸进程？
- 通过SIGCHLD信号！
  - shell在启动时，注册SIGCHLD信号
  - 后台进程变成僵尸状态（即exit），内核向其父进程，即shell，发送SIGCHLD信号
  - shell接收到信号后，在信号处理函数中，判断SIGCHLD的原因，如果是exit（还有可能是被kill -SIGSTP，即ctrl-z），则调用wait收割

你还知道哪些进程之间的通信方式？

本部分内容的要求（2025考研大纲）

- 进程间通信
  - 共享内存
  - 消息传递
  - 管道
  - 信号