第二讲 多进程程序设计

课程网站：http://spoc.buaa.edu.cn

主讲教师：李云春

办公室：新主楼G217

时间：周四上午1-2节（1-13周）

Email: lych@buaa.edu.cn

Phone: 82339268

Autumn 2023

本章内容

2.1 进程的基本概念和特点

2.2 进程的控制

2.3 信号（signal）

2.4 进程间通信（进程协作的桥梁）

2.1 进程的概念和特点

一、进程相关的概念

进程：执行中的程序的实例。系统中的每一个进程都运行在上下文(context)中。

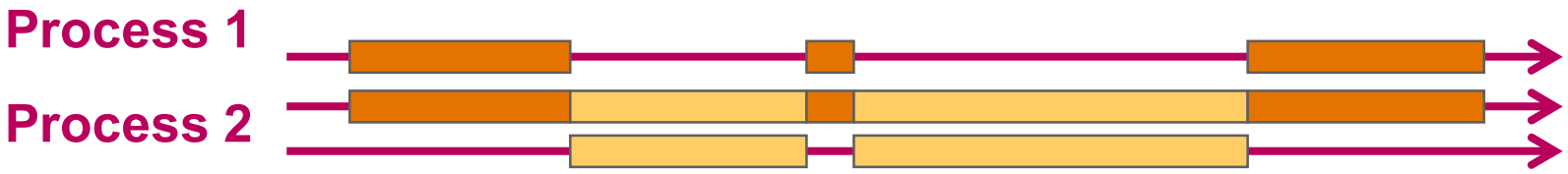
上下文：由程序正确运行所需要的状态组成，包括：

- **存放在内存中的代码和数据**
- **堆栈、寄存器内容、程序计数器、环境变量、文件描述符**

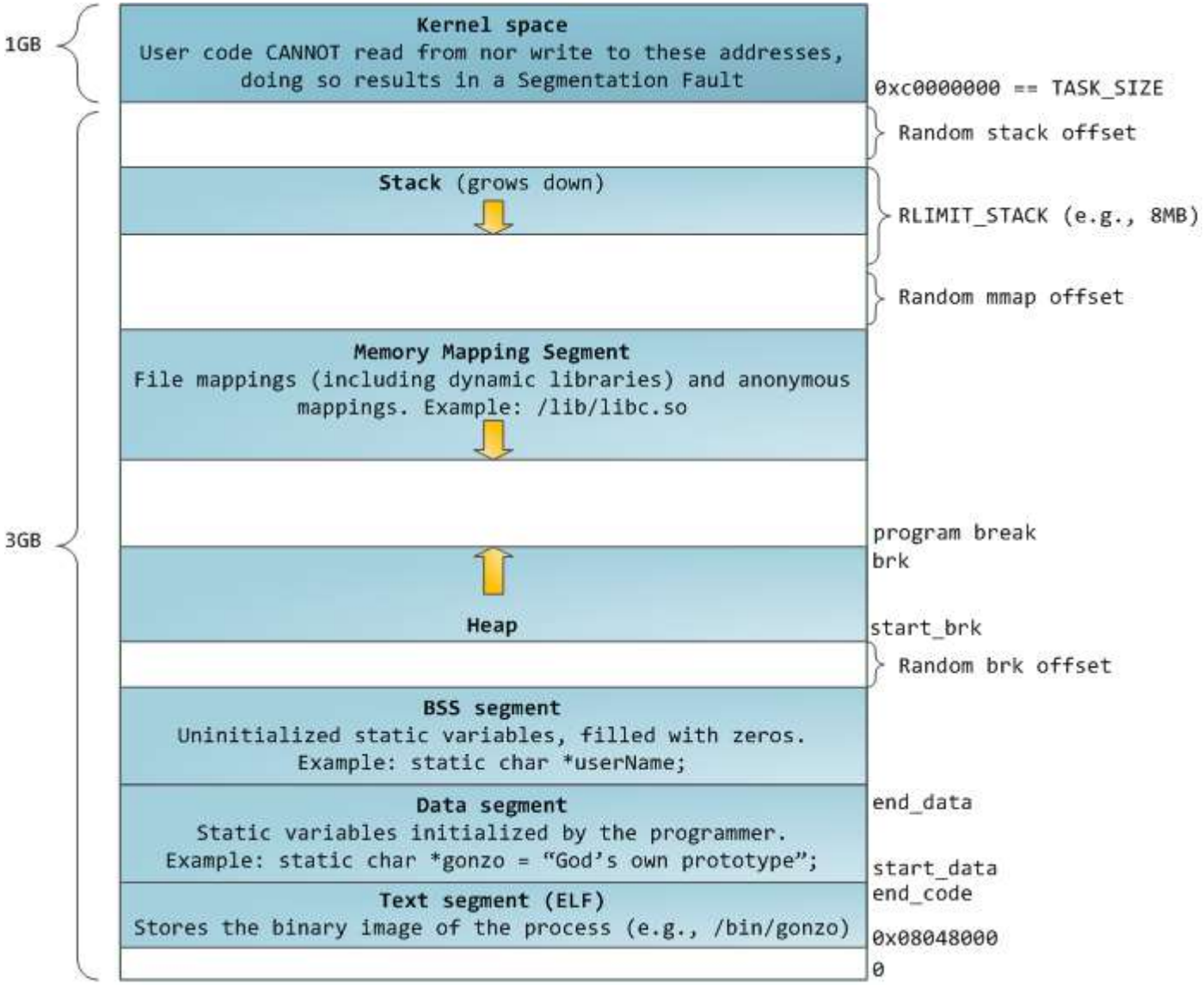
1.3 并发 vs. 并行

回忆

并发 一个逻辑流的执行在时间上与另一个流重叠，在某时间段内可能是交替执行，也可能是同时执行。



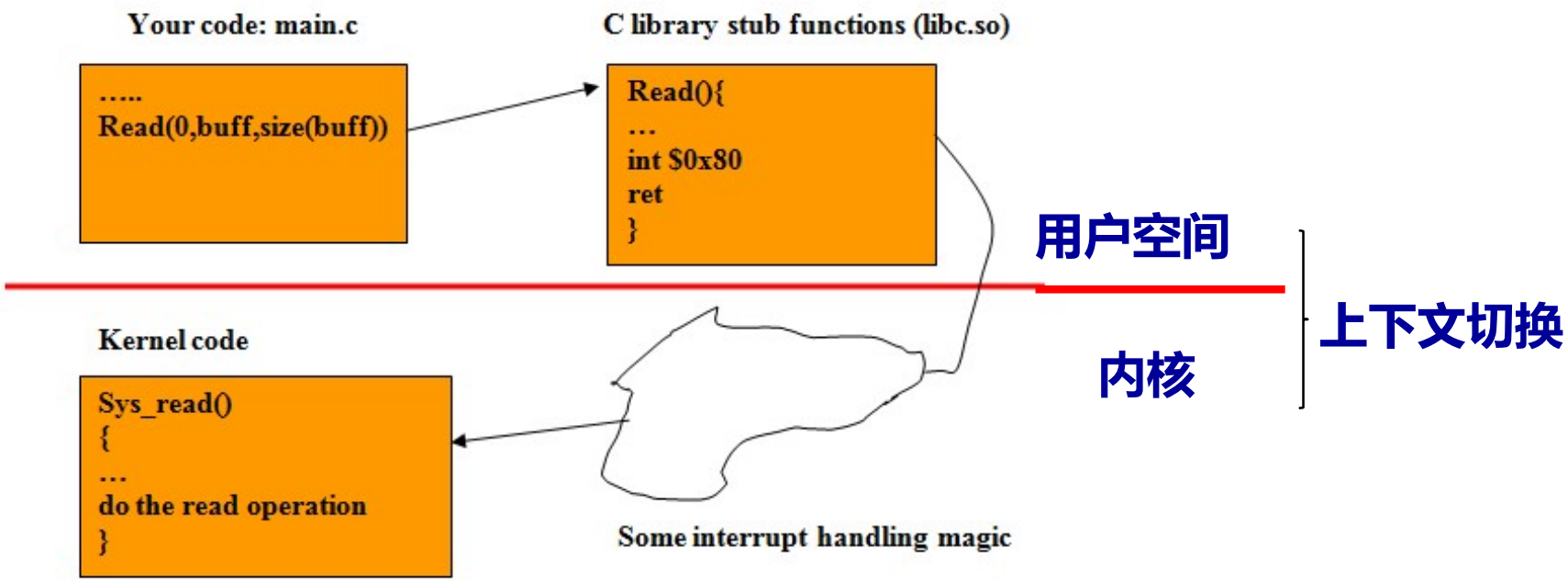
进程地址空间： 虚拟地址0~2^n-1，每个进程都有自己私有的地址空间，一个进程不可以直接访问另外一个进程的地址空间。



示例：

```
//main.cpp
  int  a  =  0;  //全局初始化区 DATA
  char  *p1;     //全局未初始化区 BSS
  main()
  {
  int  b;  //栈
  char  s[]  =  “abc”;  //栈
  char  *p2;  //栈
  char  *p3  =  “123456”;  //123456/0在常量区，p3在栈上
  static  int  c  =0;     //全局（静态）初始化区
  p1  =  (char  *)malloc(10);
  p2  =  (char  *)malloc(20);
  //分配得来的10和20字节的区域就在堆区
  strcpy(p1,  “123456”);  //123456/0放在常量区，编译器可能会
将它与p3所指向的“123456”优化到一个地方。
  }
```

系统调用： 由操作系统提供的应用程序编程接口（API）。



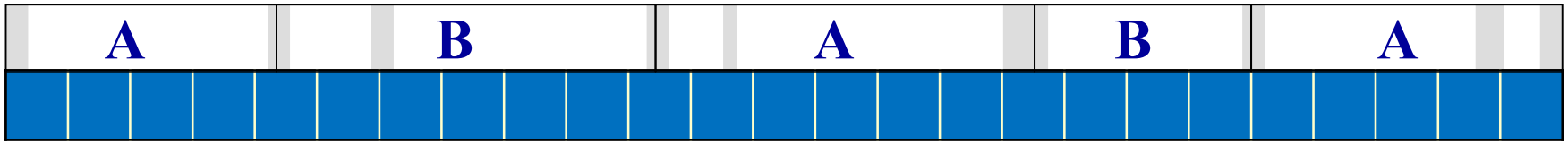
上下文切换：就是从一个可执行进程切换到另一个可执行进程。引发上下文切换的原因：

I/O request: **Process is placed on I/O queue for a device**

Spawn and wait: **Sometimes when a process creates a new process, it must wait until the child completes**

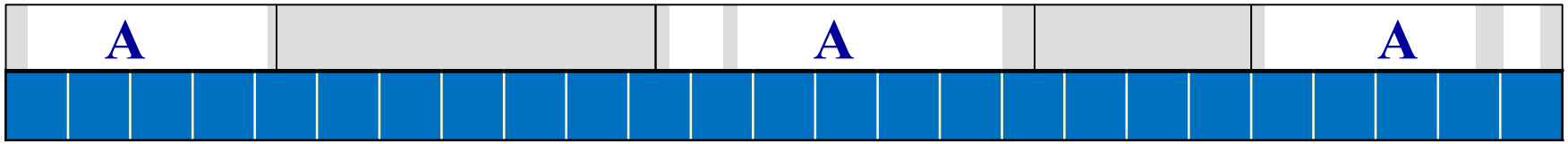
Time slice expiration

Sleep request



系统的角度

用户　内核



应用A的角度

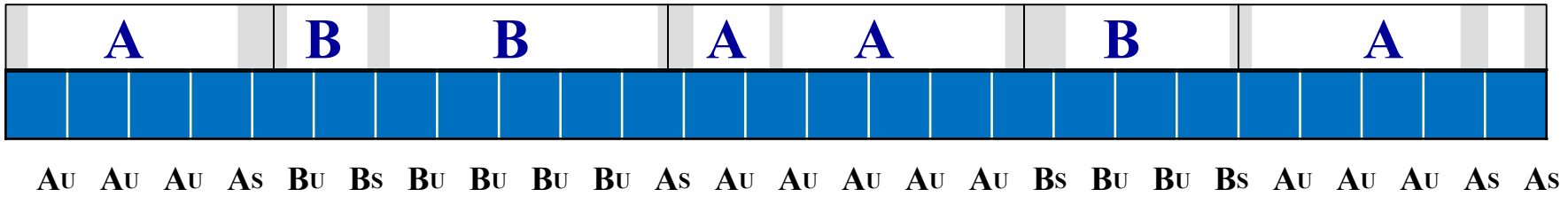
活动的　不活动的

间隔计数（Interval Counting）

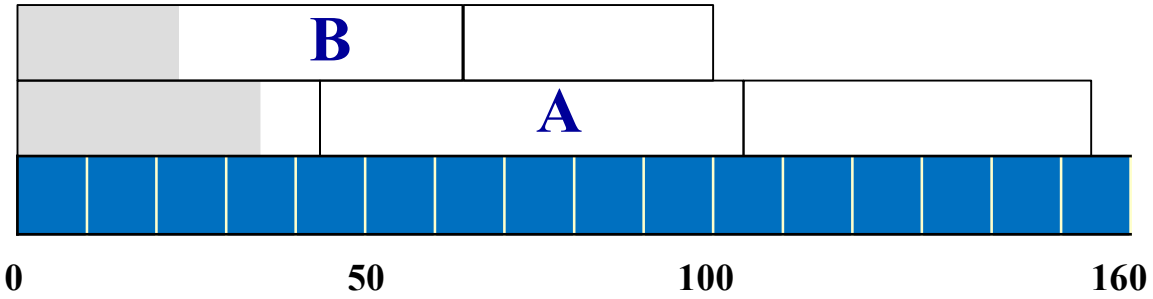
A 110u+40s

B 70u+30s

间隔计数



实际时间



B 73.3u+23.3s

A 120u+33.3s

Unix> time prog –n 17

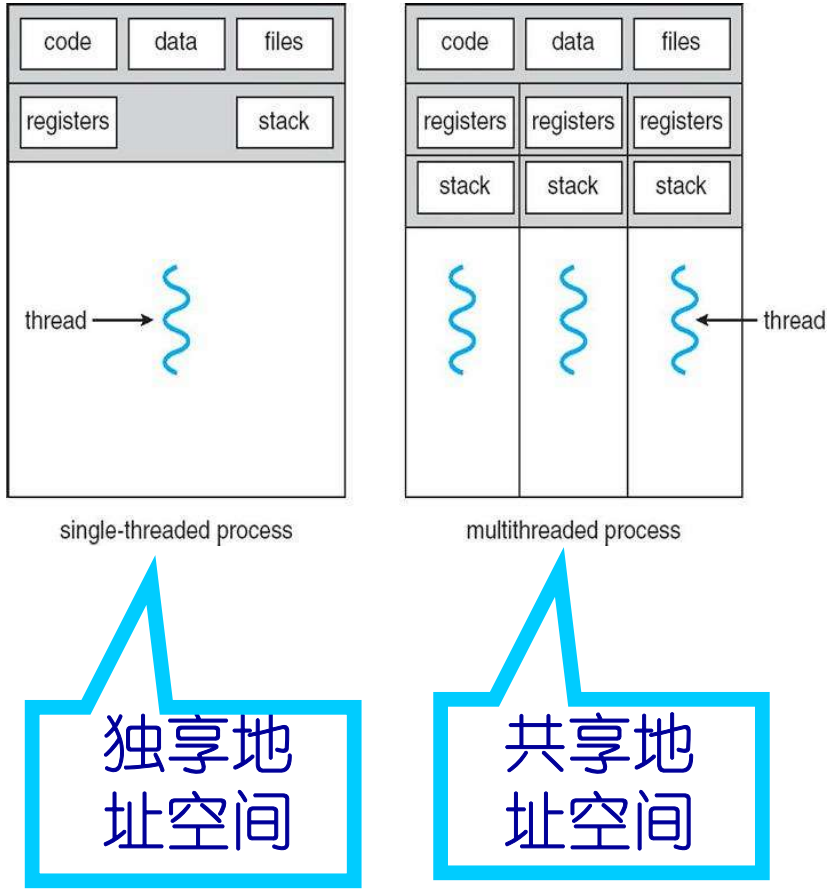
2.230u 0.260s 0:06.52 38.1% 0+0k 0+0io 80pf+0w

二、进程与线程区别

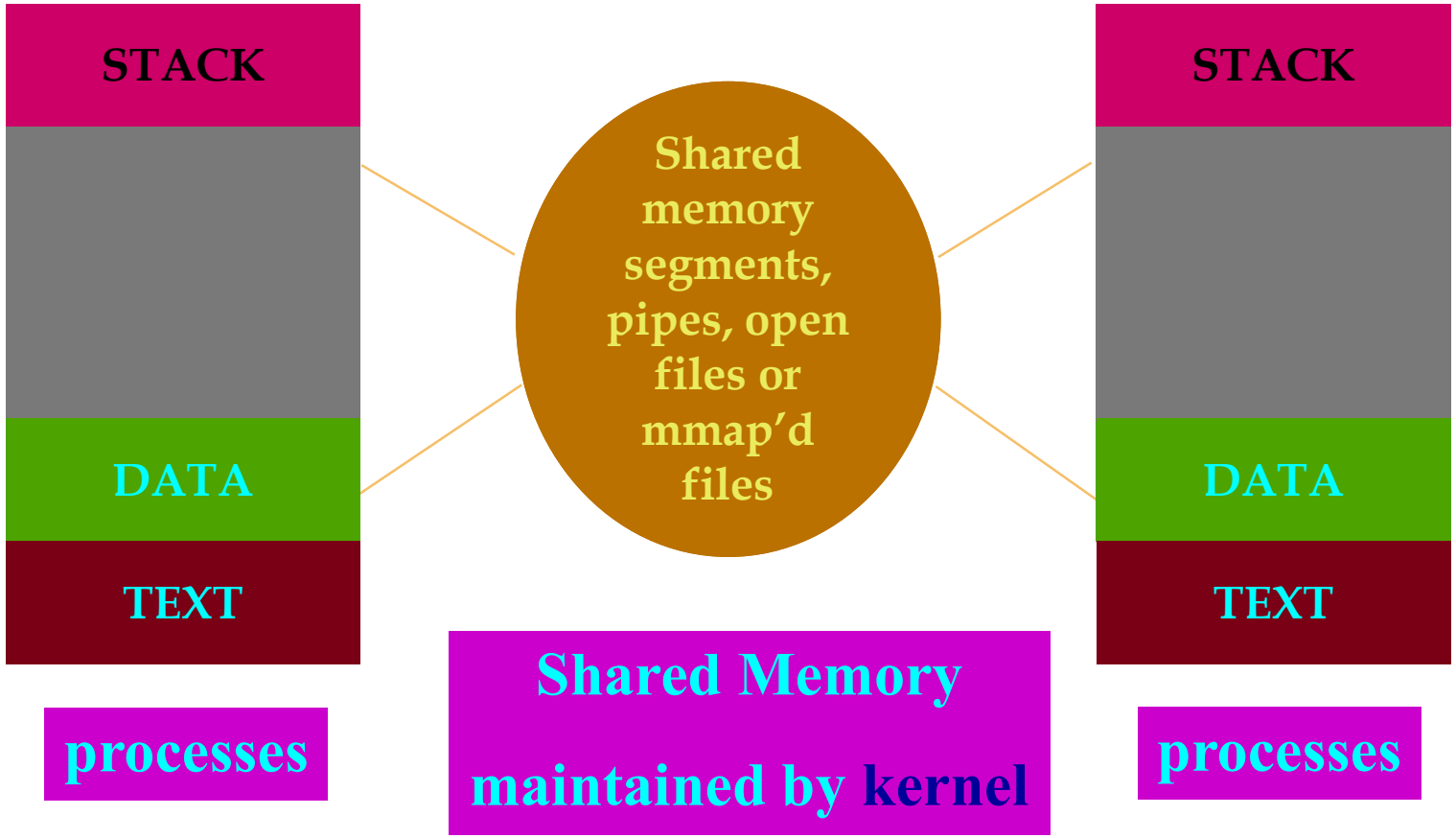
(1) 一个进程可以包括多个线程；

(2) 线程间共享地址空间，因此，一个线程修改一个变量（全局或静态），其它线程都能获知；

(3) 线程间通信直接通过共享变量，而进程间通信则需要通过管道、消息队列等；



三、进程间通信模型


STACK
DATA
TEXT
Shared memory segments, pipes, open files or mmap'd files
STACK
DATA
TEXT
processes
Shared Memory maintained by kernel
processes

2.2 进程控制

一、进程ID

每一个进程都有唯一的非零正数ID(PID)

获取进程ID

```
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

调用者进程ID

调用者父进程进程ID

二、进程创建

创建子进程

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

“fork”：把当前进程复制出一个新的进程，当前的进程就是新进程的父进程，新进程称为子进程。


```
int glob = 6;
int main()
{
        pid_t pid;
        int x=1;

        pid = fork();
        if(pid == 0) {
                glob++;
                printf("child : glob=%d, x=%d\n",
                                        glob, ++x);
                exit(0);
        }
        printf("parent: glob=%d, x=%d\n",
                                        glob, --x);
}
```

子进程

父进程

示例2

```
int glob = 6;
int main()
{
        pid_t pid;
        int x=1;

        pid = fork();
        if (pid < 0) {
                printf("%s\n", strerror(errno));
        } else if (pid == 0) {
                glob++;
                printf("child : glob=%d, x=%d\n",
                        glob, ++x);
        } else {
                printf("parent: glob=%d, x=%d\n",
                        glob, --x);
        }
}
```

子进程

父进程

调用一次，返回两次

(1) 返回到父进程；

(2) 返回到新创建的子进程；

并发执行

(1) 宏观上，父子进程并发执行；

(2) 微观上（多核系统），可能是交替执行或者并行执行；

相同但是独立的地址空间

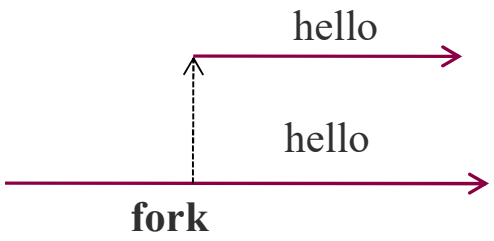
(1) 复制：相同的用户栈、相同的本地变量、相同的堆、相同的全局变量……；
(2) 独立：父子进程都有独立的私有地址空间，对变量的任何更改不会影响另外一个进程对应的变量；

文件共享

父进程打开的文件描述符，子进程可以继续使用；

多次调用 fork ?

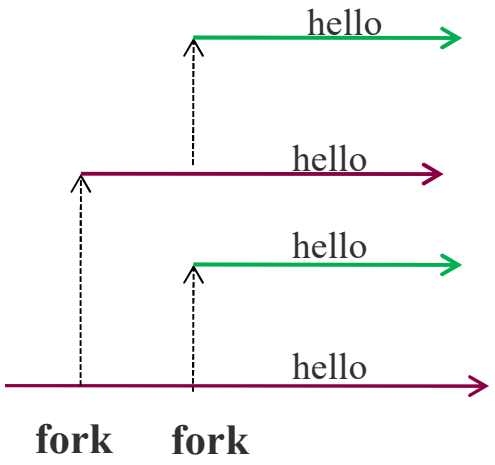

```
int main()
{
        fork();
        printf("hello\n");
        exit(0);
}
```



进程图

2次

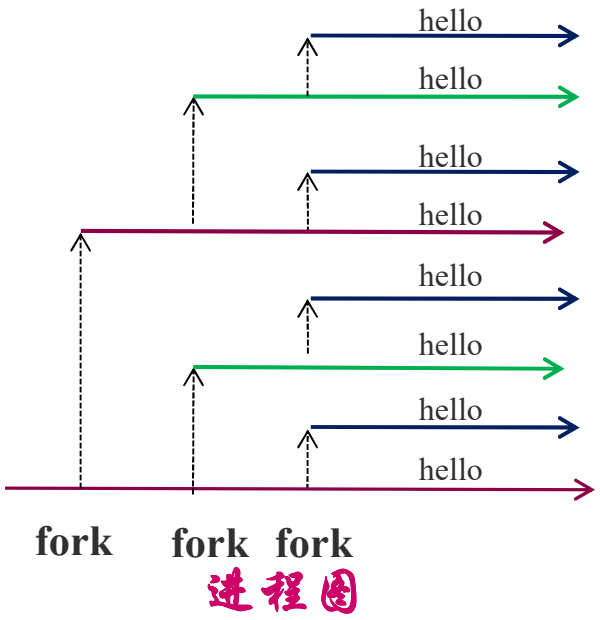
```
int main()
{
        fork();
        fork();
        printf("hello\n");
        exit(0);
}
```



进程图

3次

```
int main()
{
        fork();
        fork();
        fork();
        printf("hello\n");
        exit(0);
}
```

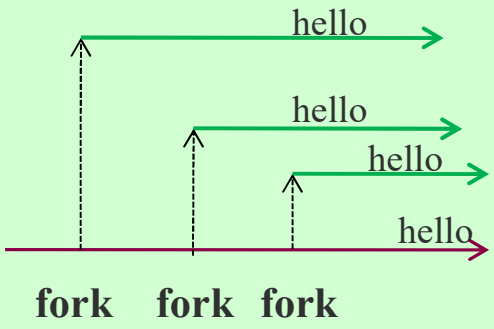


进程图

3次

```
int main()
{
        pid_t pid1, pid2;

        pid1 = fork();
        if(pid1>0) {
                pid2=fork();
                if(pid2>0) {
                        fork();
                }
        }
        printf("hello\n");
        exit(0);
}
```



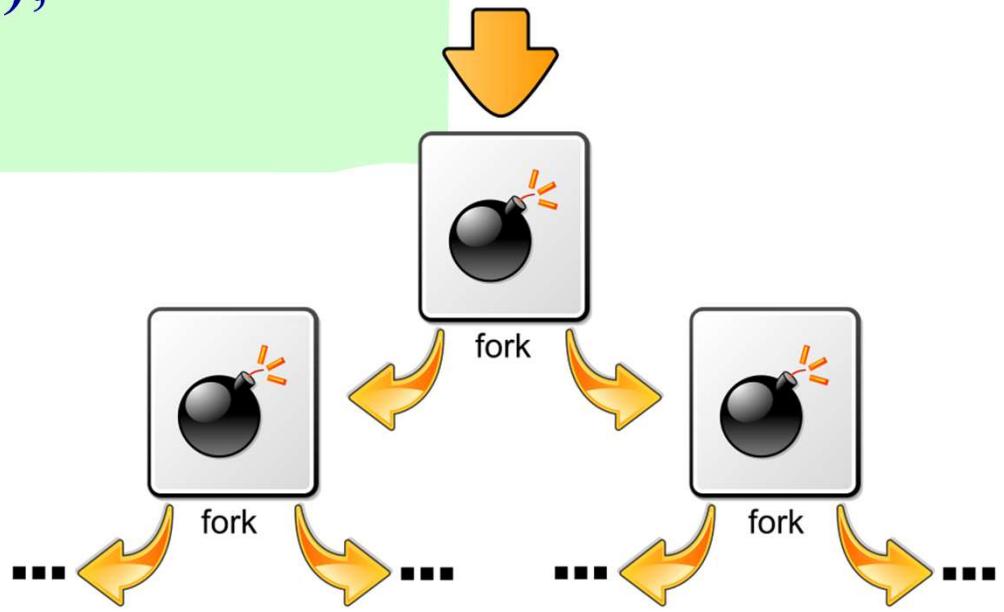
进程图

fork bomb

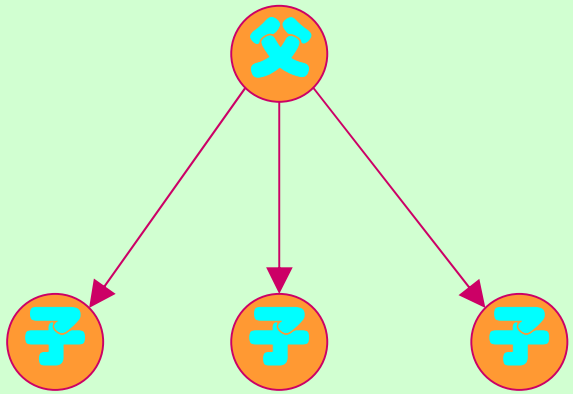
```
int main()  {

        while(1)
                fork();

        printf("hello\n");
        exit(0);
}
```




```
pid_t createChild() {
        pid_t pid = fork();
        if( pid == 0) {
                printf("child do something here!\n");
                exit(0);
        }
        return pid;
}
int main() {
        pid_t childPid;
        int i, procNum = 3;
        for ( i = 0; i < procNum ; i++) {
                childPid = createChild();
        }
        exit(0);
}
```



三、进程终止

进程终止的三种方式：

(1) 收到一个信号，信号的默认行为是终止运行；
(2) 从主程序返回；
(3) 调用exit函数；

exit函数

```
#include <stdlib.h>
void exit(int status);
```

调用exit函数，
主动退出运行

四、子进程回收

僵尸进程

子进程先于父进程终止运行，但是尚未被父进程回收，内核并未清除该进程，保持在一种已终止状态，该进程称为“僵尸进程（僵死进程）”

父进程正在运行

父进程长时间执行时，必须回收子进程

父进程先退出

?

回收子进程

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int * status);
pid_t waitpid(pid_t pid, int *status, int options);
```

wait:父进程阻塞，直到一个子进程运行结束。

退出状态

WIFEXITED (status)
WEXITSTATUS (status)

WIFSIGNALED (status)
WTERMSIG (status)
WCOREDUMP(status)

WIFSTOPPED(status)
WSTOPSIG(status)

WIFCONTINUED(status): since linux 2.6.10

POSIX.1规定终止状态用定义在<sys/wait.h>中的各个宏来查看

退出状态解析

```
void wait_exit(int status) {
        if (WIFEXITED(status))
                printf("normal termination, exit status = %d\n",
                        WEXITSTATUS(status));
        else if (WIFSIGNALED(status))
                printf("abnormal termination, signal number =
                        %d%s\n", WTERMSIG(status),
#ifdef  WCOREDUMP
                WCOREDUMP(status) ? " (core file generated)" : "");
#else
                "");
#endif
        else if (WIFSTOPPED(status))
                printf(“child stopped, signal number = %d\n”,
                        WSTOPSIG(status));
}
```

正常退出

非正常退出

暂停（挂起）

回收子进程

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int * status);
pid_t waitpid(pid_t pid, int *status, int options);
```

waitpid:比wait更加灵活，可以是非阻塞的。

参数:pid_t pid

pid > 0: 等待进程id为pid的子进程。

pid == 0: 等待与自己同组的任意子进程。

pid == -1: 等待任意一个子进程

pid < -1: 等待进程组号为-pid的任意子进程。

参数:int options

0：阻塞，即挂起调用进程；

WNOHANG：非阻塞，即如果指定的子进程都还没有终止运行，立即返回0；否则返回，终止运行进程ID；

WUNTRACED：如果子进程挂起（暂停），则立即返回；默认只返回终止运行的进程；

WUNTRACED | WNOHANG:

wait(&stat)等价于waitpid(-1, &stat, 0)

五、执行一个新程序

执行新程序

```
#include <unistd.h>

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

启动一个新的程序，新的地址空间完全覆盖当前进程的地址空间

参数

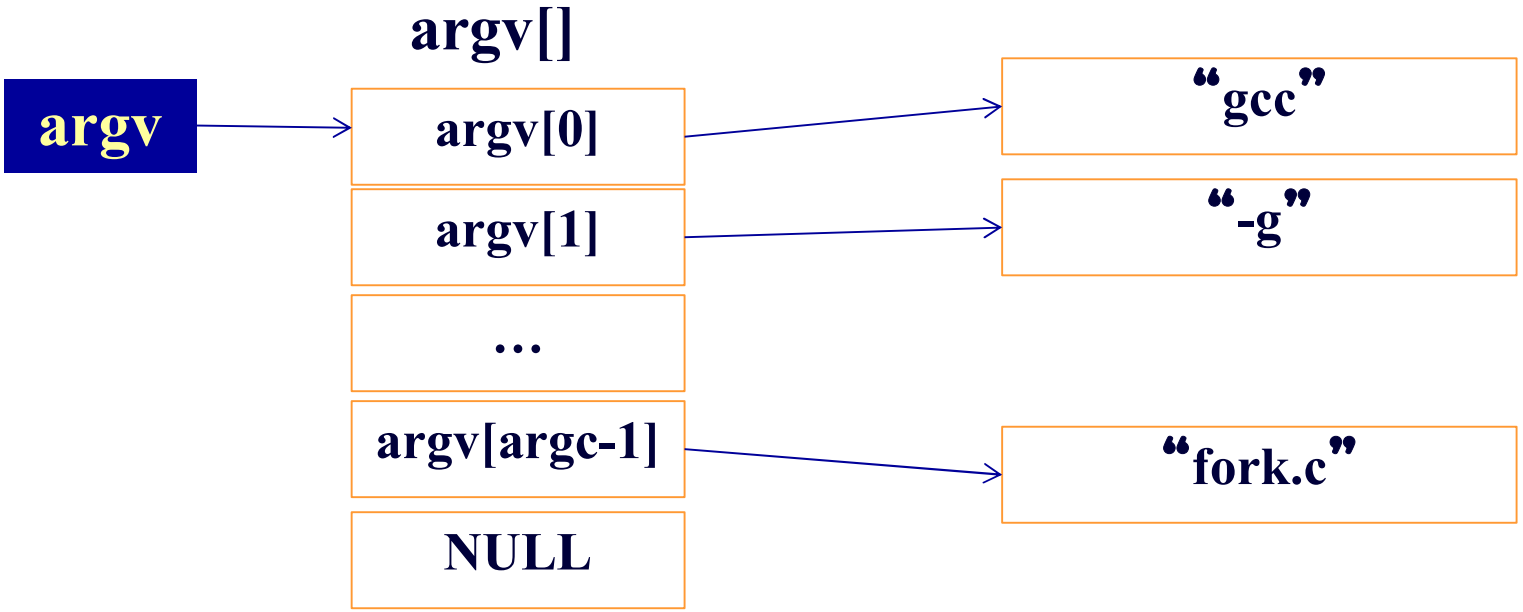
path：执行文件
argv：参数表
envp：环境变量表，一般直接用environ就行

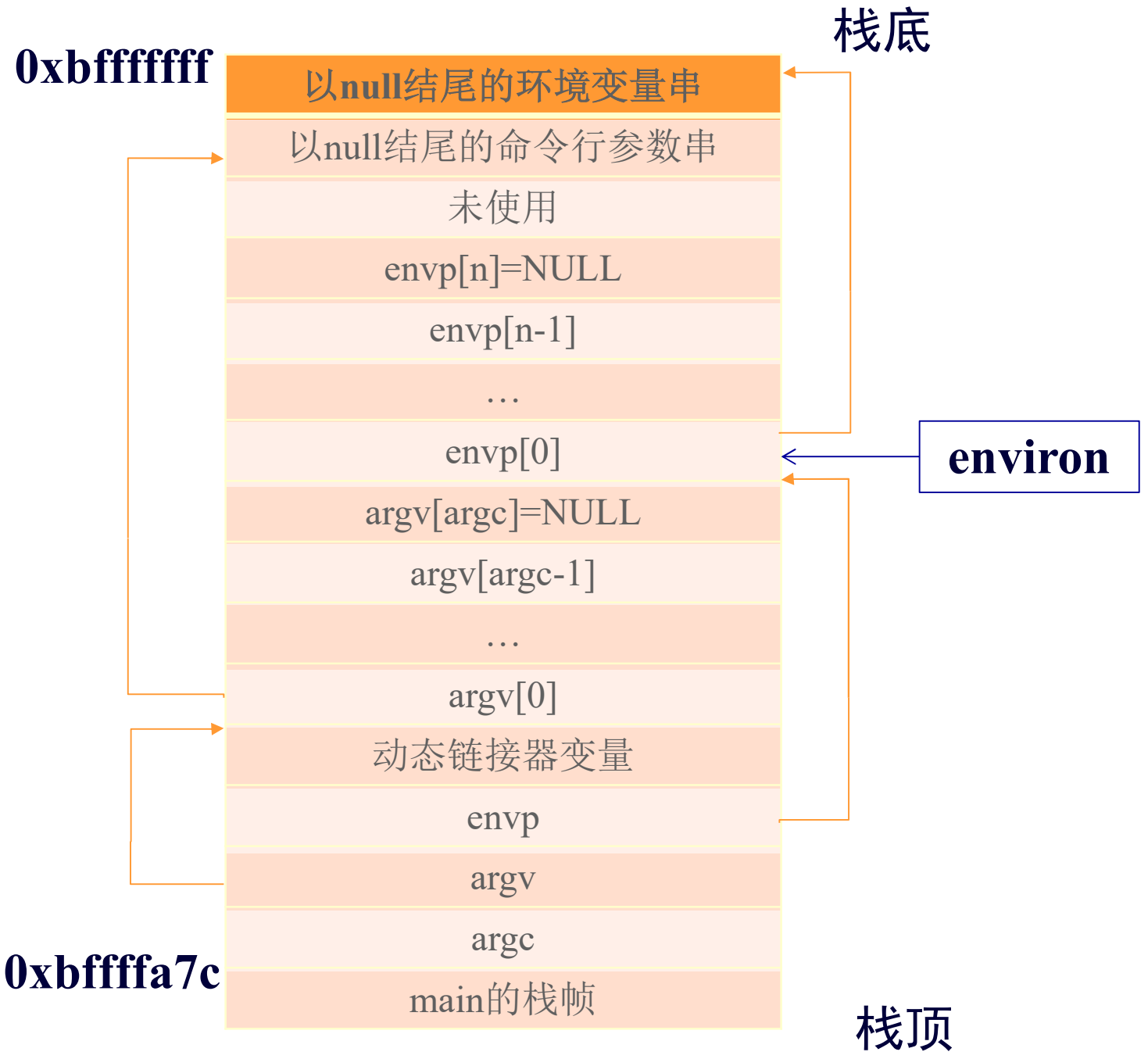
环境变量:在操作系统中用来指定操作系统运行环境的一些参数


```
char *argv[] = {“gcc”, “-g”, “-c”, “fork.c”, NULL};
execve(“/usr/bin/gcc”, argv, environ);
```

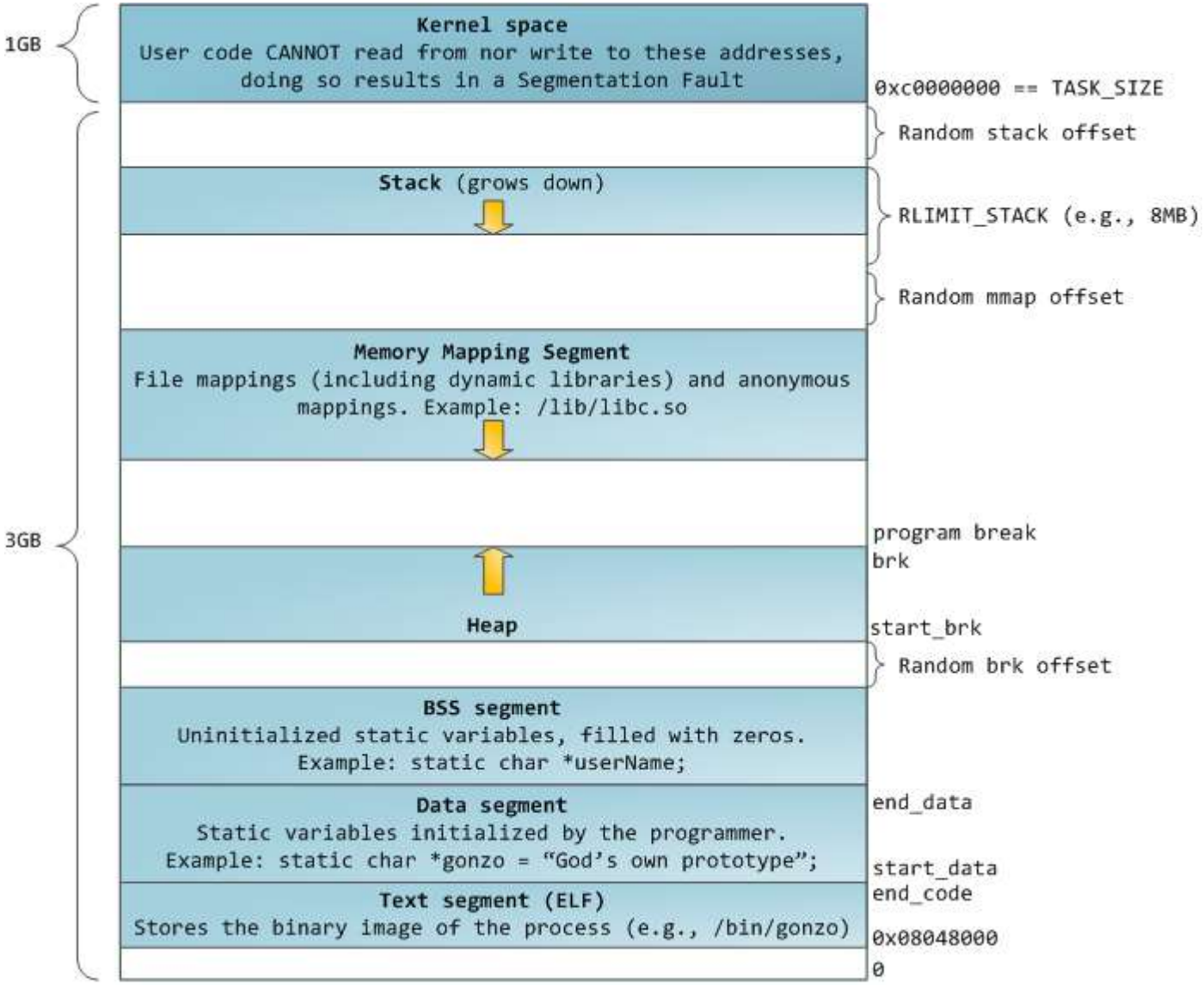
char *argv[] = {“gcc”, “-g”, “-c”, “fork.c”, NULL};

execve(“/usr/bin/gcc”, argv, environ);




栈底
0xbfffffff
以null结尾的环境变量串
以null结尾的命令行参数串
未使用
envp[n]=NULL
envp[n-1]
…
envp[0]
environ
argv[argc]=NULL
argv[argc-1]
…
argv[0]
动态链接器变量
envp
argv
argc
0xbffffa7c
main的栈帧
栈顶

进程地址空间： 虚拟地址0~2^n-1，每个进程都有自己私有的地址空间，一个进程不可以直接访问另外一个进程的地址空间。



例子：

1. 源代码上传

Web编程：

通过HTTP协议上传源代码文件至服务器

2. 编译源代码

1）守护进程fork出子进程；

2）子进程内调execve，执行gcc

```
char *argv[] = {“gcc”, “-o”, “3906”, “3906.c”, “-O2” NULL};
execve(“gcc”, argv, environ);
```

3. 执行

1）守护进程fork出子进程；

2）子进程内调execve，执行3906

3）调用waitpid：

a)等待子进程结束

i)父进程计时，如果子进程在2秒内不结束，父进程 kill -9 pid

b)获取结束状态

i)正常终止

ii)非正常终止： SIGSEGV、 SIGFPE

3. 执行伪代码

```
pid_t pid = fork();
if(pid == 0) {
        execv(“./3906”, argv, environ);
        exit(0);
}
开始计时
int status;
while(waitpid(pid, &status, WNOHANG)){
        if(超过规定时间)
                kill(pid);
}
```

status，检查退出状态，错误原因

2.3 信号(signals)

信号：也称作软中断，是Linux操作系统用来通知进程发生了某种异步事件的一种手段。

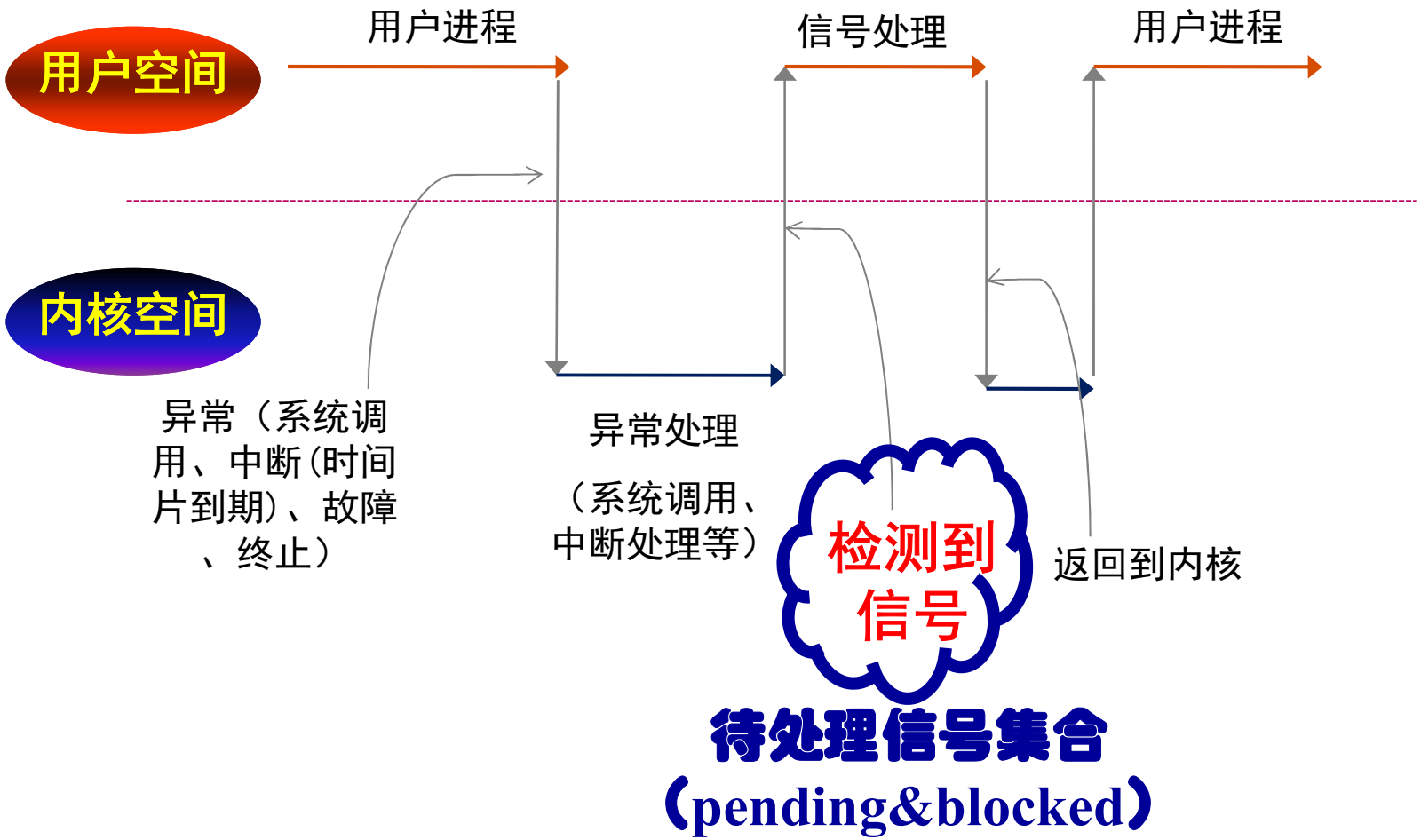
例

```
zch@ZCH:~/PP/lecture02$ ./loopforever &
[1] 2227
zch@ZCH:~/PP/lecture02$ kill 2227
zch@ZCH:~/PP/lecture02$
[1]+  Terminated                 ./loopforever
zch@ZCH:~/PP/lecture02$ ./loopforever &
[1] 2231
zch@ZCH:~/PP/lecture02$ kill -9 2231
zch@ZCH:~/PP/lecture02$
[1]+  Killed                     ./loopforever
```

常见信号

SIGINT	前台程序执行过程中按下Ctrl-c就会向它发出SIGINT信号，默认终止进程
SIGKILL	立即中止进程，不能被捕获或忽略
SIGTERM	kill命令默认的中止程序信号
SIGALRM	定时器到期，可用alarm函数来设置定时器。默认动作终止进程
SIGCHLD	子进程终止或停止，默认动作为忽略（waitpid）
SIGSEGV	无效的内存引用

信号检测与处理流程


用户空间
内核空间
用户进程
信号处理
用户进程
异常（系统调用、中断(时间片到期)、故障、终止）
异常处理
（系统调用、中断处理等）
检测到信号
返回到内核
待处理信号集合
（pending&blocked）

一、发送信号

内核通过更新目的进程上下文的某个状态（pending位向量,blocked位向量），来实现信号的发送。

信号发送起因

内核检测到一个系统事件：除零、子进程终止、非法内存引用

一个进程调用了kill函数：显式要求内核向目的进程发送信号，目的进程可以是调用进程自身

信号发送函数

kill函数

向其它进程（包括自己）发送信号

```
#include <sys/types.h>
#include <signal.h>

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

alarm函数

给自己发送时钟信号(SIGALRM)

```
#include <unistd.h>

unsigned int alarm(unsigned int secs);
```

例1

```
int main() {
        pid_t pid;
        /* child sleeps until SIGKILL signal received, then dies */
        if ((pid = fork()) == 0) {
            pause(); /* wait for a signal to arrive */
            printf("control should never reach here!\n");
            exit(0);
        }
        /* parent sends a SIGKILL signal to a child */
        kill(pid, SIGKILL);
        exit(0);
}
```

子进程sleep,只有信号才能将其唤醒

父进程向子进程发信号


```
int main() {
        alarm(5); /* next SIGALRM will be delivered in 5s */


        while (1) {
                ; /* signal handler returns control here each time */
        }
        exit(0);
}
```

二、接收信号

当进程接收到一个信号（可能是自己发出，也可能是别的有权限的进程发出），它可以采取的动作可以是下面任意一种：

信号处理

忽略信号：SIGSTOP与SIGKILL除外

捕获信号：通过系统调用，自定义接收到信号后采取的行动。SIGSTOP与SIGKILL除外

执行信号的默认动作

信号捕捉

函数指针

回调函数
(callback function)

```
#include <signal.h>

typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

=

```
typedef void (sighandler_t)(int);
sighandler_t* signal(int signum, sighandler_t* handler);
```

SIG_IGN
SIG_DFL
函数地址

信号捕捉

```
#include <signal.h>

int sigaction(int signum, struct sigaction *act, struct
sigaction *oldact);
```

安装信号处理函数

```
int main() {
        signal(SIGALRM, handl
        alarm(5); /* next SIGALRM


        while (1) {
            ; /* signal handler returns c
        }
        exit(0);
}
```

```
void handler(int sig) {
        static int beeps = 0;
        printf("BEEP\n");
        if (++beeps < 5)
            alarm(1);
        /* 1秒后再次发送信号*/
        else {
printf("BOOM!\n");
            exit(0);
        }
}
```

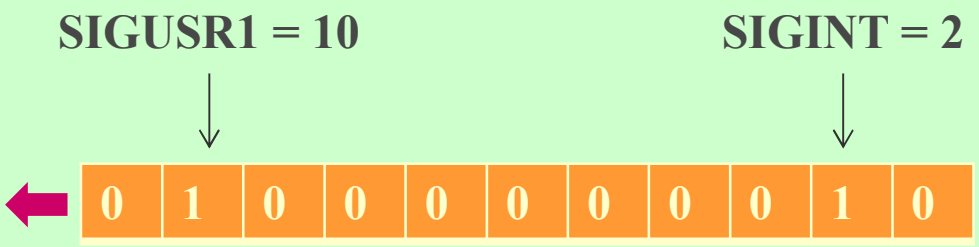

```
int main() {
        pid_t pid;
        /* 子进程内安装信号处理函数
        if ((pid = fork()) == 0) {
             signal(SIGTERM, han
             signal(SIGINT, handle
              while(1)
                   pause();
        }
        /*父进程向子进程发送信号 */
        sleep(2);
        kill(pid, SIGINT);
        sleep(2);
        kill(pid, SIGTERM);
}
```

```
void handler(int sig) {
    /*只有收到SIGTERM信号才终止运行*/
     if(sig == SIGTERM) {
          printf("Catch signal
                  SIGTERM\n");
          exit(0);
     } else {
          printf("Sorry! you are signal
          %d, i will not exit!\n", sig);
     }
}
```

三、同时接收多个信号

如果进程同时收到多个同类型的信号，
如何处理？

信号存储



位向量pending

以信号SIGUSR1为例

多个信号处理

情况1：如果SIGUSR1的信号处理函数正在执行，第二个SIGUSR1到达

SIGUSR1存入pending位向量，针对该信号的处理函数随后执行

情况2：pending位向量内已经存在一个SIGUSR1，一个新的SIGUSR1到达？

新的SIGUSR1信号被丢弃

信号阻塞

可以显示的阻塞
或者取消阻塞

信号阻塞函数

```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t
*oldset);

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

SIG_BLOCK
SIG_UNBLOCK
SIG_SETMASK

信号阻塞：目的进程接收但是不处理该信号，直到取消阻塞。实现：blocked位向量存放被阻塞信号


```
int sign = 0;
int main() {
    pid_t pid;
    if ((pid = fork()) == 0){
        signal (SIGUSR1, receive); //注册SIGUSR1的响应器
        while ( sign==0 ) ;//原地踏步程序;
        sign = 0; //还原sign值
        kill (getppid (), SIGUSR1); //给父进程发信号
        exit(0);
    }
    signal (SIGUSR1, receive); //注册SIGUSR1的响应器
    //here, do something
    kill (pid, SIGUSR1);
    while (sign==0) ; //原地踏步程序,等待子进程的接受确认消息
    sign = 0; //还原sign值
}
```

```
void receive (int signo) {
        printf ("pid=%d,received\n", getpid());
        sign = 1;
}
```

子进程

父进程

```
int sign = 0;
int main() {
    pid_t pid;
    if ((pid = fork()) == 0){
        signal (SIGUSR1, receive); //注册SIGUSR1的响应器
        while ( sign==0 ) ;//原地踏步程序;
        sign = 0; //还原sign值
        kill (getppid (), SIGUSR1); //给父进程发信号
        exit(0);
    }
    signal (SIGUSR1, receive); //注册SIGUSR1的响应器
    //here, do something
    kill (pid, SIGUSR1);
    while (sign==0) ; //原地踏步程序,等待子进程的接受确认消息
    sign = 0; //还原sign值
}
```

执行结果：父进程永远“忙等”

子进程在signal之前，收到了父进程发来的SIGUSR1信号
SIGUSR1的默认行为是终止运行

子进程

父进程

正确的程序

```
int sign = 0;
int main() {
    pid_t pid;
    if ((pid = fork()) == 0){
        signal (SIGUSR1, receive); //注册SIGUSR1的响应器
        while ( sign==0 ) ;//原地踏步程
        sign = 0; //还原sign值
        kill (getppid (), SIGUSR1); //给
        exit(0);
    }
    signal (SIGUSR1, receive); //注册SIGUSR1的响应器
    //here, do something
    kill (pid, SIGUSR1);
    while (sign==0) ; //原地踏步
    sign = 0; //还原sign值
}
```

```
sigemptyset(&chldmask);
sigaddset(&chldmask, SIGUSR1);
sigprocmask(SIG_BLOCK, &chldmask, &savemask)
```

阻塞SIGUSR1信号，即，如果收到SIGUSR1信号，不做处理，放到block位向量内

```
sigprocmask(SIG_SETMASK, &savemask, NULL);
```

取消SIGUSR1信号阻塞，即，将block位向量的信号，放到pending向量，并调用相应的信号处理函数

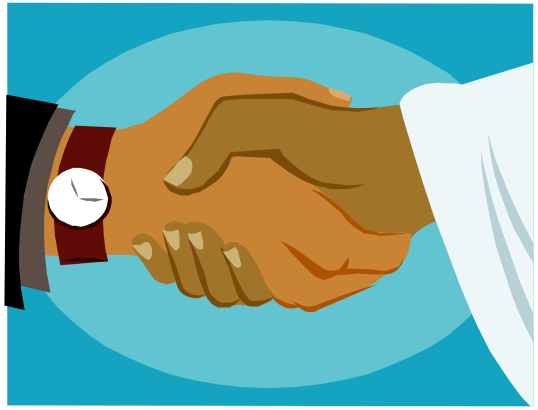
子进程

```
sigprocmask(SIG_SETMASK, &savemask, NULL);
```

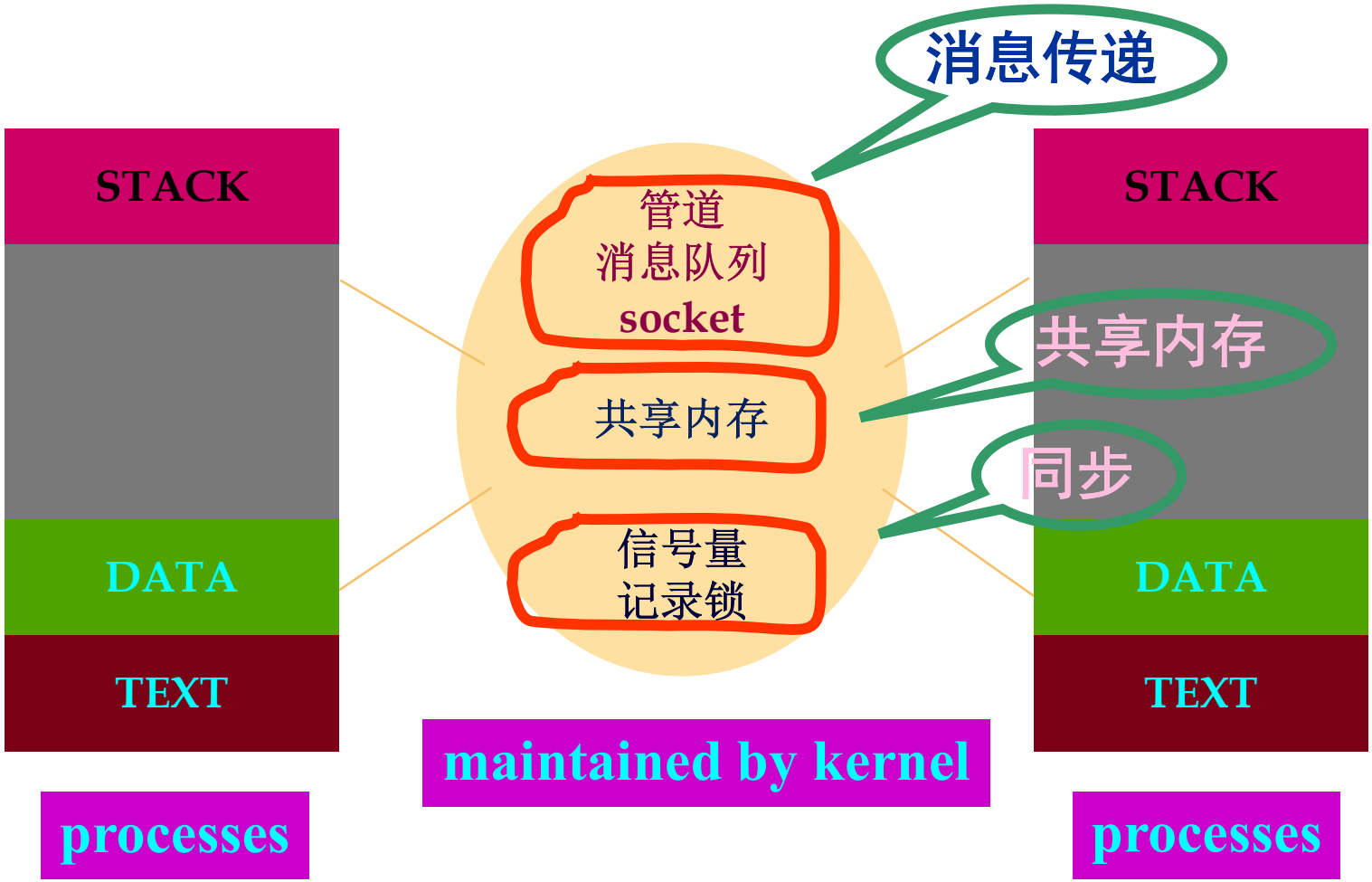
取消SIGUSR1信号阻塞，即，将block位向量的信号，放到pending向量，并调用相应的信号处理函数

2.4 进程间通信(interprocess communication)

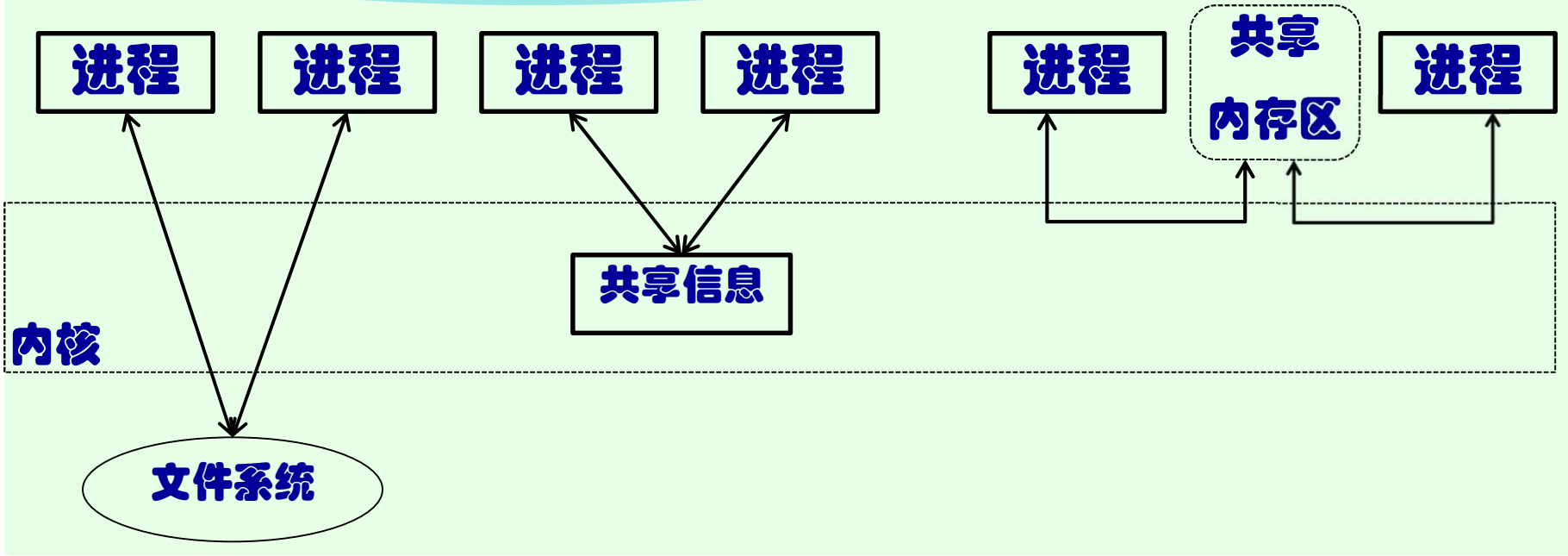
进程间通信（IPC）：多个进程间协作或沟通的桥梁。



进程间通信方式


消息传递
STACK
管道
消息队列
socket
共享内存
信号量
记录锁
共享内存
同步
STACK
DATA
TEXT
DATA
TEXT
maintained by kernel
processes
processes

进程间共享三种方式

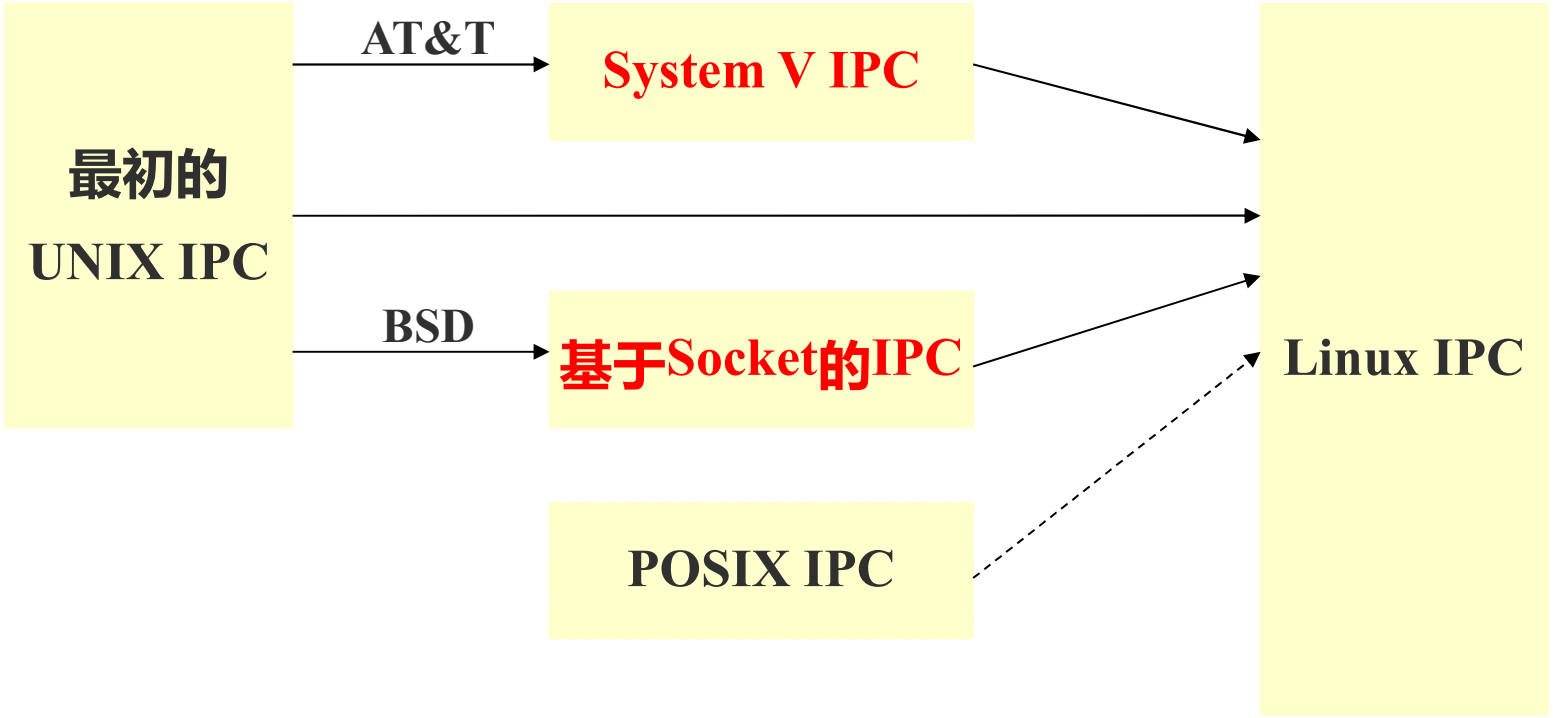


- 随进程持续
- 随内核持续
- 随文件系统持续

IPC类型	持续性
管道	随进程
FIFO	随进程
Posix互斥锁	随进程
Posix条件变量	随进程
Posix读写锁	随进程
fcntl记录上锁	随进程
Posix消息队列	随内核
Posix有名信号量	随内核
Posix基于内存的信号量	随进程
Posix共享内存区	随内核
System V消息队列	随内核
System V信号量	随内核
System V共享内存区	随内核
TCP套接字	随进程
UDP套接字	随进程
Unix域套接字	随进程

IPC进化史

POSIX： Portable Operating System Interface [for Unix]



最初的Unix IPC：信号(signal)、管道、FIFO；

System V IPC：System V 消息队列、 System V信号量、 System V共享内存；

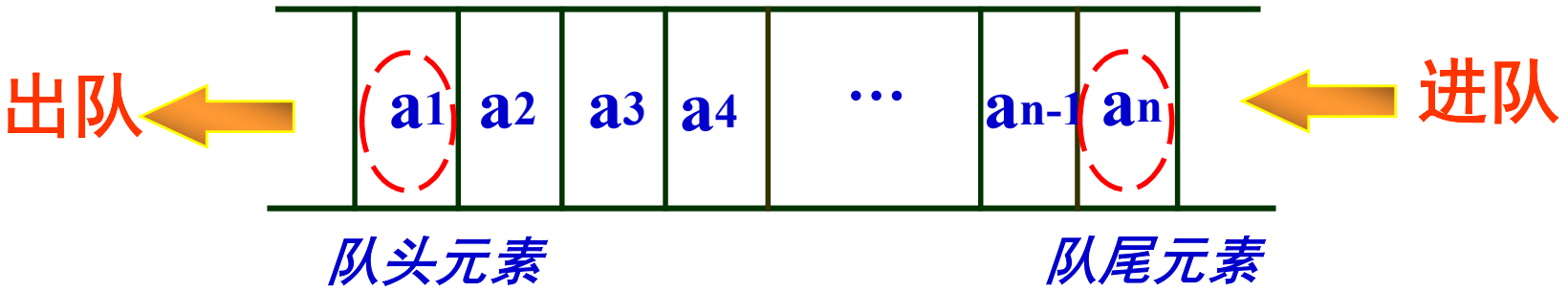
POSIX IPC：POSIX消息队列、 POSIX信号量、POSIX共享内存 √

主要介绍

1. POSIX message queue √
2. POSIX Semphore √
3. POSIX共享内存

一、消息队列

普通队列示意图



先进先出(FIFO)

消息队列（优先级队列）示意图

优先级

先进先出

消息队列（优先级队列）示意图

队列满

1 阻塞

2 失败返回（非阻塞）

发送进程

消息队列（优先级队列）示意图

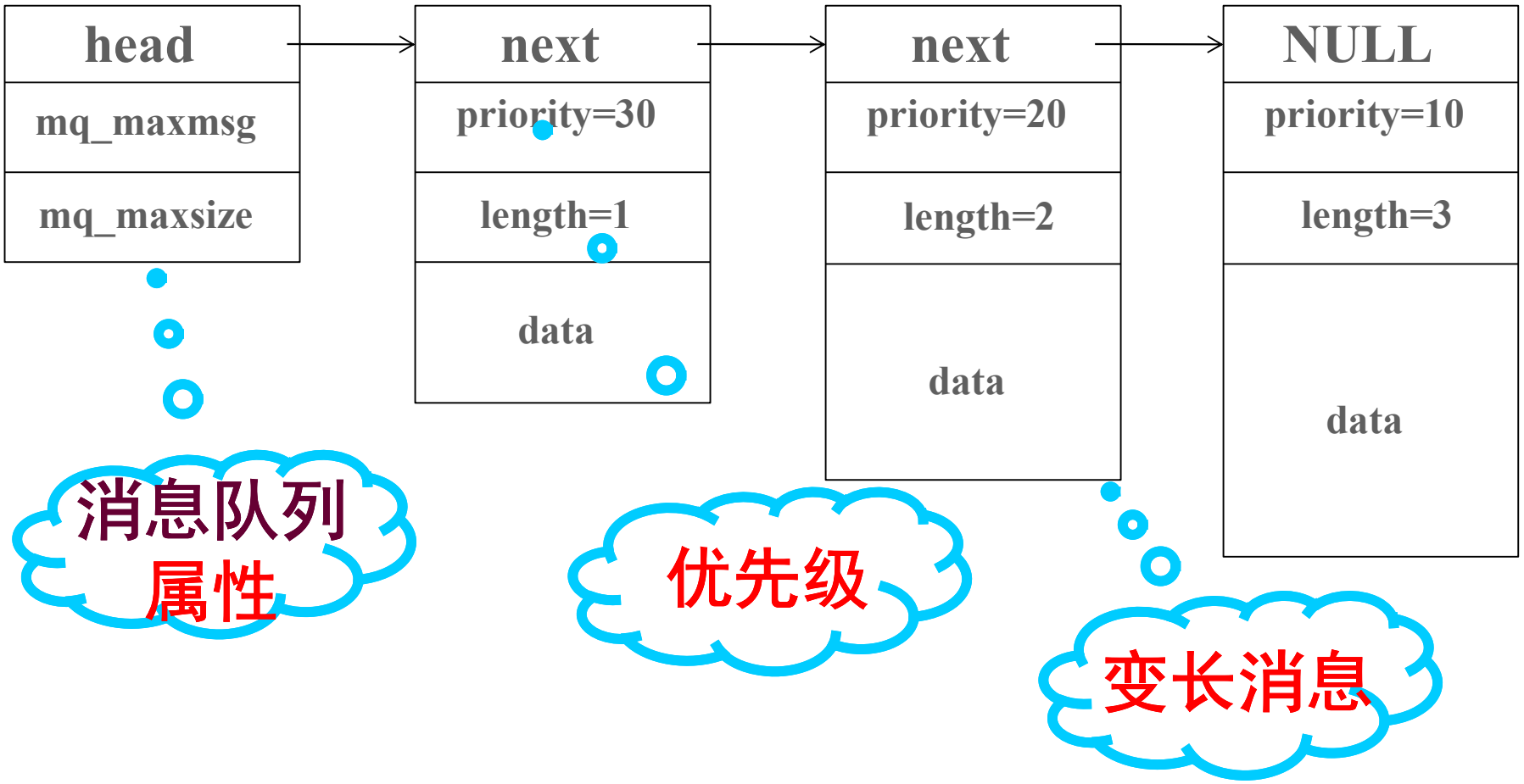
接收进程

1 阻塞

2 失败返回（非阻塞）

队列空

POSIX消息队列可能实现



消息队列操作

打开消息队列

```
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>

mqd_t mq_open(const char *name, int oflag);

mqd_t mq_open(const char *name, int oflag,
              mode_t mode,  struct mq_attr *attr);
```

打开已经存在的队列

创建新的队列

参数：int oflag

O_RDONLY	O_NONBLOCK
O_WRONLY	O_CREAT
O_RDWR	O_EXCL

参数：const char *name

以/开头的字符串。除了开头的/，字符串内不能包含/

参数：struct mq_attr * attr

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

attr可以是NULL

创建的时候只有这两个成员起作用

文件访问权限

参数：mod_t mod

mode 宏	值（8进制）	涵义
S_IRUSR	00400	用户读【文件owner具有读权限】
S_IWUSR	00200	用户写
S_IXUSR	00100	用户执行
S_IRGRP	00040	组读【与owner同组的用户具有读权限】
S_IWGRP	00020	组写
S_IXGRP	00010	组执行
S_IROTH	00004	其他读【其他用户具有读权限】
S_IWOTH	00002	其他写
S_IXOTH	00001	其他执行

常用权限设置

普通文件：　　0644

可执行文件：　　0755

文件夹：　　0751


```
struct mq_attr attr;
attr.mq_maxmsg = 10;
attr.mq_msgsize = 100;
attr.mq_flags = 0;
mqd_t mqdes = mq_open(“/lyc”, O_RDWR|O_CREAT,
                                        0664, &attr);
if(mqdes < 0) {
        printf(“%s\n", strerror(errno));
        exit(-1);
}
```

创建队列


```
mqd_t mqdes = mq_open("/lyc", O_RDONLY);
if(mqdes < 0) {
        printf("%s\n", strerror(errno));
        exit(-1);
}
```

打开队列

（用于接收消息）

队列属性

获取队列属性

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *attr);

int mq_setattr(mqd_t mqdes, struct mq_attr *newattr,
               struct mq_attr *oldattr);
```

设置队列属性

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

发送消息

```
#include <mqueue.h>

int mq_send(mqd_t  mqdes,  const char *msg_ptr,
                    size_t msg_len, unsigned int msg_prio);
```

msg_ptr:　　发送的消息缓冲区（传入）

msg_len:　　消息实际长度（传入）

msg_prio:　　消息优先级（传入）

例如:

```
int msg;
mq_send(mqdes, &msg, sizeof(int), 0);
```

接收消息

返回消息实际大小

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t  mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);
```

msg_ptr: 接收到的消息缓冲区（传出）

msg_len: 消息缓冲区大小（传入）

msg_prio: 接收消息的优先级（传出）

例如:

```
char buffer[128];
ssize_t len = mq_receive(mqdes, buffer, 128, NULL);
```

关闭队列

```
#include <mqueue.h>

int mq_close(mqd_t  mqdes);
```

删除队列

```
#include <mqueue.h>

int mq_unlink(const char *name);
```

计算机学习

求深不求广

理解运行机理、重视基础学习

简单的问题（项目）深入去做

有深必有广

程序设计

复杂问题简单化

分而治之：分层次、分步骤、分阶段、分任务

利用你所掌握的最成熟的技术

非最新、最热门的技术

不要追求一步到位

程序设计是一个不断反复、重构的过程

已知N，计算：

$$\sum_{i=1}^{N}\frac{\sqrt{i}}{i+1}$$

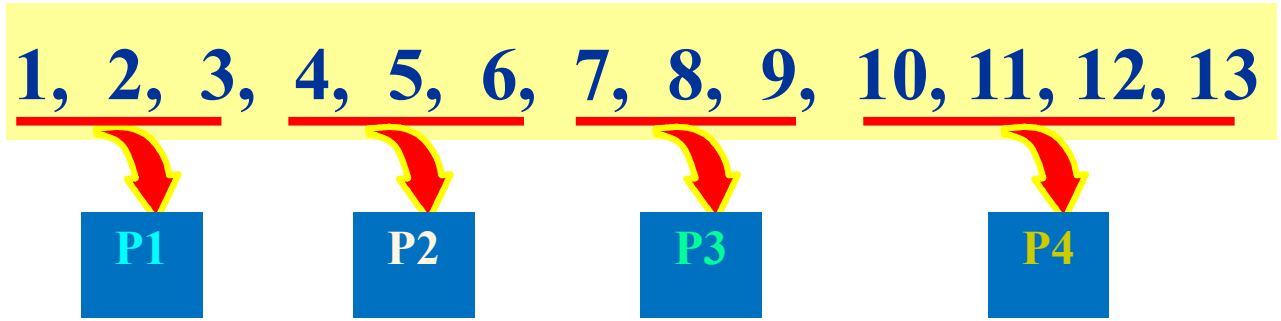
需要做的工作

1. 划分任务：任务之间依赖性最小。

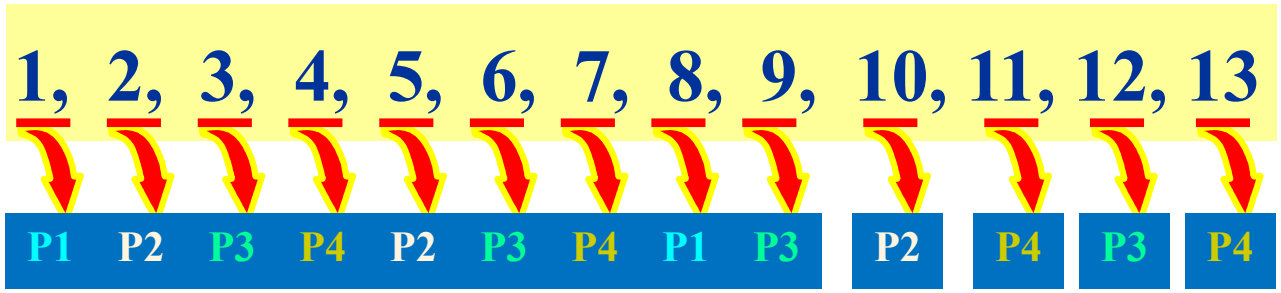
$$\sum_{i=1}^{N}\frac{\sqrt{i}}{i+1}$$

每一个i的计算都是独立的

静态任务分配



动态任务分配（能者多劳）



需要做的工作

1. 划分任务：任务之间依赖性最小。
2. 选择技术（动态任务分配）：

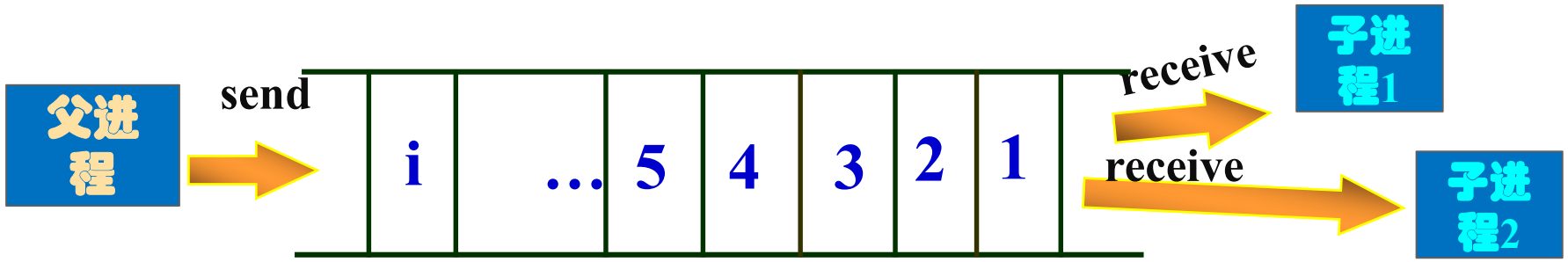
生产者

父进程利用消息队列分任务，队列满，阻塞；

消费者

子进程从消息队列取任务，队列空，阻塞；

回收子进程-waitpid;



父进程例

```c
int main() {
        pid_t childPid;
        int cpuNum = 1, waitStatus, i, exitMsg=-1;

        /*parent open message queue*/
        struct mq_attr attr;
        attr.mq_maxmsg = MY_MQ_MAXMSG;
        attr.mq_msgsize = MY_MQ_MSGSIZE;

        mqd_t p2cmqdes = mq_open(MY_MQ_NAME, O_RDWR|O_CREAT, 0664, &attr);
        if(p2cmqdes < 0) {
                printf("parent(%d):%s\n", getpid(), strerror(errno));
                return -1;
        }

        cpuNum = get_nprocs();  /*the computer's cpu number*/;
        for ( i = 0; i < cpuNum; i++) {
                childPid = createChild(p2cmqdes);
        }

        for( i = 1; i <= 100000; i++) {
                mq_send(p2cmqdes, &i, sizeof(int), 0);
        }

        /*tell the children to exit*/
        for ( i = 0; i < cpuNum; i++) {
                mq_send(p2cmqdes, &exitMsg, sizeof(int), 0);
        }

        /*wait all children to exit*/
        while( (childPid = wait(&waitStatus) ) > 0 ) {
                printf("child %d exit\n", childPid);
        }
}
```

设置消息队列属性

创建消息队列

创建子进程
(=CPU数)

分配任务
(N=100000)

发送-1代表任务结束

等待子进程结束

例

子进程

从父进程传入的队列描述符

子进程接收任务

接收到-1，退出循环

执行计算任务

```
#define MY_MQ_NAME "/mymqP2cc"
#define MY_MQ_MAXMSG 10
#define MY_MQ_MSGSIZE 64

char msg_buf[MY_MQ_MSGSIZE];

pid_t createChild(mqd_t rmqdes) {
        int recvValue;
        uint prio;

        pid_t pid = fork();

        float sum = 0;
        if( pid == 0) {
                while(1) {
                        ssize_t msgSize = mq_receive(rmqdes, msg_buf, MY_MQ_MSGSIZE, &prio);
                        if(msgSize < 0) {
                                printf("child(%d):receive error: %s\n", getpid(), strerror(errno));
                                exit(1);
                        }
                        memcpy(&recvValue, msg_buf, msgSize);

                        if(recvValue < 0)
                                break;

                        sum += sqrt(recvValue) / (recvValue + 1) ;
                }
                printf("child(%d): sum = %f\n", getpid(), sum);
                exit(0);
        }

        return pid;
}
```

当多个子进程并行执行，同时receive消息队列中的队首消息时，怎么保证每个消息只被一个进程读取，而不会被多个读取?

1.消息队列receive函数内部保证对队首消息的互斥访问!
2. 队首消息拷贝到msg_ptr所指的地址，而不是队首消息的指针赋给msg_ptr

```
ssize_t mq_receive(mqd_t  mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio);
```

memcpy系统调用从缓冲区msg_buf读取msgSize长度的消息到recvValue。memcpy是不是一个原子操作？如何保证缓冲区msg_buf中的内容不会被多个进程同时读取?

1. memcpy不是原子操作

2. 由于进程的地址空间是独立的，每一个进程都有一个msg_buf，并不是被多个进程共享的，所以msg_buf的内容不会被多个进程同时读取。

父进程如何获得子进程计算结果并汇总

用两个消息队列

一个队列向子进程发送任务

新增一个队列接收子进程的计算结果

父进程

改进版

```
int waitStatus;

int i, j;

/*parent open message queue*/
mq_unlink(MY_MQ_P2C_NAME);
mq_unlink(MY_MQ_C2P_NAME);
struct mq_attr attr;
attr.mq_maxmsg = MY_MQ_MAXMSG;
attr.mq_msgsize = MY_MQ_MSGSIZE;
attr.mq_flags = 0;

mqd_t p2cmqdes = mq_open(MY_MQ_P2C_NAME, O_RDWR|O_CREAT, 0664, &attr);
if(p2cmqdes < 0) {
        printf("parent(%d):%s\n", getpid(), strerror(errno));
        return -1;
}

mqd_t c2pmqdes = mq_open(MY_MQ_C2P_NAME, O_RDWR|O_CREAT, 0664, &attr);
if(c2pmqdes < 0) {
        printf("parent(%d):%s\n", getpid(), strerror(errno));
        return -1;
}

cpuNum = get_nprocs();  /*the computer's cpu number*/;
for ( i = 0; i < cpuNum; i++) {
        childPid = createChild(p2cmqdes, c2pmqdes);
}
for( i = 1; i < 10000; i++) {
        memcpy(msg_buf, &i, sizeof(int));
        mq_send(p2cmqdes, msg_buf, sizeof(int), 0);
}
```

先删除 两个队列
（边界处理）

创建任务发送队列

创建任务接收队列

创建子进程

发送任务

父进程

改进版

```
/*tell the children to exit*/
for ( i = 0; i < cpuNum; i++) {
        j = -1;
        memcpy(msg_buf, &j, sizeof(int));
        mq_send(p2cmqdes, &j, sizeof(int), 0);
}
```

任务结束，告知子进程

```
/*receive the result from children*/
float total = 0;
for ( i = 0; i < cpuNum; i++) {
        uint prio;
        mq_receive(c2pmqdes, msg_buf, MY_MQ_MSGSIZE, &prio);
        float sum;
        int childPid;
        memcpy(&sum, msg_buf, sizeof(float));
        memcpy(&childPid, msg_buf + sizeof(float), sizeof(int));
        printf("child(%d): sum = %f\n", childPid, sum);
        total += sum;
}
printf("final result: %f\n", total);
```

接收子进程计算结果

等待子进程结束

```
/*wait all children to exit*/
while( (childPid = wait(&waitStatus) ) > 0 ) {
        printf("child %d exit\n", childPid);
}

mq_close(p2cmqdes);
mq_close(c2pmqdes);

mq_unlink(MY_MQ_P2C_NAME);
mq_unlink(MY_MQ_C2P_NAME);
```

```
pid_t createChild(mqd_t r_mqdes, mqd_t s_mqdes) {
        int recvValue;
        uint prio;

        pid_t pid = fork();

        float sum = 0;
        if( pid == 0) {
                while(1) {
                        ssize_t msgSize = mq_receive(r_mqdes, msg_buf, MY_MQ_MSGSIZE, &prio);
                        if(msgSize < 0) {
                                printf("child(%d):receive error: %s\n", getpid(), strerror(errno));
                                exit(1);
                        }
                        memcpy(&recvValue, msg_buf, msgSize);
                        if(recvValue < 0)
                                break;
                        sum += sqrt(recvValue) / (recvValue + 1) ;
                }

                /*send result to the parent*/
                memcpy(msg_buf, &sum, sizeof(float));
                int childPid = getpid();
                memcpy(msg_buf + sizeof(float), &childPid, sizeof(int));
                mq_send(s_mqdes, msg_buf, sizeof(float) + sizeof(int), 1);

                exit(0);
        }

        return pid;
}
```

子进程

改进版

从父进程传入的队列描述符

子进程接收任务

接收到-1，退出循环

执行计算任务

计算结果发给父进程

二、信号量（named semaphores）

信号量(semaphore)：用来对共享资源进行并发访问控制。


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

    FILE* fp;
    long i, seqno = 1;

    fp = fopen("seqno", "r+");

    for (i = 0; i < 20; i++) {
        rewind(fp);     /*rewind before read*/
        fscanf(fp, "%ld\n", &seqno);

        printf("pid = %d, seq# = %ld\n",  getpid(),  seqno);

        seqno++;

        rewind(fp);     /*rewind before write*/
        fprintf(fp, "%ld\n", seqno);
    }

    fclose(fp);

}
```

序号生成器，序号要保证唯一

应用：作业调度、打印池

从文件内读序号

临界区

将当前序号写入文件

用信号量保护临界区

进程1	进程2	进程n
sem_wait	sem_wait	sem_wait
critical regions	*critical regions*	*critical regions*
sem_post	sem_post	sem_post

同时只有1个或几个进程进入临界区

信号量操作

打开信号量

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag);

sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);
```

打开已经存在的信号量

创建新的信号量

参数：int oflag

0	打开已存在的
O_CREAT	创建如果不存在
O_EXCL| O_CREAT	创建如果不存在，若存在，返回错误

参数：int value

最多允许并发访问的进程数

== 1：互斥访问

> 1：有限的并发访问

关闭信号量

```
#include <mqueue.h>

int sem_close(sem_t *sem);
```

删除信号量

```
#include <mqueue.h>

int sem_unlink(const char *name);
```

test and decrement

```
#include <mqueue.h>

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

sem_wait：(1) 若信号量的值等于0，阻塞，直到值大于0；
(2) 若信号量的值大于0，值减1；

sem_trywait：不会阻塞，返回-1，errno == EAGAIN；

increment

```
#include <mqueue.h>

int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int*valp);
```

sem_post：(1) 信号量的值加1，唤醒等待进程；

取得等待的进程数目

```
#include <mqueue.h>

int sem_getvalue(sem_t *sem, int*valp);
```

参数：int *valp

*valp == 0：没有进程正在等待；

*valp < 0：abs(*valp)个进程正在等待；

test and decrement

```
#include <mqueue.h>
```

P

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
```

sem_wait：(1) 若信号量的值等于0，阻塞，直到值大于0；
(2) 若信号量的值大于0，值减1；

sem_trywait：不会阻塞，返回-1，errno == EAGAIN；

increment

```
#include <mqueue.h>
```

V

```
int sem_post(sem_t *sem);
int sem_getvalue(sem_t *sem, int*valp);
```

sem_post：(1) 信号量的值加1，唤醒等待进程；
(2) 若信号量的值大于0，值减1；


```
int main(int argc, char ** argv) {
        FILE* fp;
        long i, seqno = 1;

        fp = fopen("seqno", "r+");
        sem_t *sem = sem_open("/semlock", O_CREAT, 0644, 1);
        for (i = 0; i < 20; i++) {
            sem_wait(sem);
            rewind(fp);     /*rewind before read*/
            fscanf(fp, "%ld\n", &seqno);
            printf("pid = %d, seq# = %ld\n",  getpid(),  seqno);

            seqno++;

            rewind(fp);     /*rewind before write*/
            fprintf(fp, "%ld\n", seqno);
            sem_post(sem);
        }
        fclose(fp);
    }
```

打开信号量

进入临界区

出临界区

本章内容小结

多进程编程

进程的基本概念

- 上下文、上下文切换、与线程的区别
- 系统调用

  运行于内核空间

进程控制

- 创建子进程，**fork**的特点
- 子进程回收（子进程结束释放**SIGCHLD**信号）

  调用**wait:**等待子进程结束、获得子进程结束状态

信号

- 信号在操作系统中的实现
- 信号的发送、信号捕获与处理

进程间通信

- **POSIX**消息队列
- **POSIX**信号量

如何考核?

1. 考勤 （10%）
2. 作业 （75%）
   - 3次小作业（45%）
   - 1次大作业（30%）
3. **Presentations** （15%）

读论文：（ppt 12分钟，3分钟问答）

1）3人一组，确定一个主题

2）每人选择一篇论文（从CCF A或者B的期刊或者会议挑选），同一团队的系列论文

3）进行论文分析、总结