getpid/ wait/ waitpid

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

int main() {
    pid_t fpid; // fpid表示fork函数返回的值
    int count = 0;

    fpid = fork();
    if (fpid < 0) {
        printf("Error in fork!\n");
    } else if (fpid == 0) {
        printf("I am the child process, my process ID is %d\n", getpid());
        printf("I am child process\n");
        count++;
    } else {
        printf("I am the parent process, my process ID is %d\n", getpid());
        printf("I am parent process\n");
        count++;
    }

    printf("Count is: %d\n", count);

    return 0;
}
```

- 两个进程都会输出1
- 说明子进程的空间独立于父进程的，本质上是写时复制

```c
#include "stdio.h"
#include "sys/types.h"
#include "unistd.h"
int main()
{
pid_t pid1;
pid_t pid2;

pid1 = fork();
pid2 = fork();

printf("pid1:%d, pid2:%d\n", pid1, pid2);
}
```

- 总共会有一个父进程和三个子进程，一共四个进程。调用一次fork就会产生一个子进程，此外第一个子进程内部又会调用一次fork并再产生一个子进程
- 第一个fork只会执行一次；第二个fork会执行两次（主进程一次，第一个子进程一次）
- 都是子进程-pid1，pid2；第一个是子进程，第二个不是-pid1,0；第一个不是子进程，第二个也不是-0,0；第一个 不是，第二个是-0，pid3

- fork的本质：资源的拷贝的pc指针的拷贝，子进程从fork的下一句指令开始执行

```
#include "stdio.h"
#include "sys/types.h"
#include "unistd.h"
int main()
{
pid_t pid1;
pid_t pid2;

if ((pid1 = fork()) == 0) return 0;
if ((pid2 = fork()) == 0) return 0;

printf("pid1:%d, pid2:%d\n", pid1, pid2);
}
```

- 进程的创建只通过主线程进行
- 一共只有三个进程

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
    int i;
    for(i=0; i<3; i++){
        fork();
        printf("hello\n");
    }
    return 0;
}
```

- i=0：
  - main (fork后的父进程)
  - p1
- i=1
  - main+p2* （p2*是fork后的父进程）
  - main+p2 （p2是fork的子进程）
  - p1+p3*
  - p1+p3
- i=2
  - main+p2*+p4
  - main+p2*+p4*
  - main+p2+p5
  - main+p2+p5*

- p1+p3*+p6
- p1+p3*+p6*
- p1+p3+p7
- p1+p3+p7*

一共14次输出，8个进程

```c
/* wait2.c */

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

int main()
{
    int status;
    pid_t pc, pr;
    pc = fork();
    if (pc < 0)
    {
        printf("error ocurred!\n");
    }
    else if (pc == 0)
    {
        printf("This is child process with pid of %d.\n", getpid());
        exit(3);
    }
    else
    {
        pr = wait(&status);
        if (WIFEXITED(status))
        {
            printf("the child process %d exit normally.\n", pr);
            printf("the return code is %d.\n", WEXITSTATUS(status));
        }
        else
        {
            printf("the child process %d exit abnormally.\n", pr);
        }
    }
    return 0;
}
```

wait、WIFEXITED和exit是在Unix/Linux操作系统中用于管理进程的函数和关键字。

wait函数用于父进程等待子进程的状态改变，以便获取子进程的退出状态或终止信号，并在子进程退出后回收其资源。wait函数的语法为：

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

其中，pid_t是进程ID的数据类型，*status是一个指向整型变量的指针，用于存储子进程的状态。

WIFEXITED是一个宏定义，用于判断子进程是否正常退出。如果子进程正常退出，则WIFEXITED返回一个非零值，并且可以使用WEXITSTATUS宏获取子进程的退出状态。WIFEXITED的语法为：

```
#include <sys/wait.h>
int WIFEXITED(int status);
```

其中，status是wait函数返回的子进程状态。

exit函数用于终止当前进程，并返回一个退出状态。exit的语法为：

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

其中，status是一个整型值，用于表示进程的退出状态。

总结起来，wait和WIFEXITED是用于获取子进程状态的函数和宏定义，而exit是用于终止当前进程并返回退出状态的函数。

```
/* 包含头文件 */
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

/* 主函数 */
int main()
{
    pid_t pc, pr;

    /* 创建子进程 */
    pc = fork();
    if (pc < 0) // 处理fork失败的情况
        printf("Error occured on forking.\n");
    else if (pc == 0) // 子进程执行的代码
    {
        sleep(10); // 延迟10秒后退出
        exit(0);
    }
    else // 父进程执行的代码
        do
        {
            /* 检查子进程状态 */
            pr = waitpid(pc, NULL, WNOHANG);
            if (pr == 0) // 子进程还未退出
            {
```

```
                printf("No child exited\n");
                sleep(1); // 等待1秒
            }
        }
        while (pr == 0); // 循环直到子进程退出

    /* 根据waitpid返回值判断子进程是否成功退出 */
    if (pr == pc)
        printf("successfully get child %d\n", pr);
    else
        printf("some error occured\n");
}
```

其中，waitpid函数的语法为：

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

其中，pid表示需要等待的子进程ID，status表示用于存储子进程状态信息的指针，options表示等待子进程的选项。在本程序中，options使用了WNOHANG选项，表示如果子进程还未退出，则waitpid函数立即返回0，而不是等待子进程退出。循环检查子进程状态的代码使用了do-while循环，因为第一次检查子进程状态时，子进程可能还未退出，因此需要先执行一次检查。在检查子进程状态时，使用了sleep函数等待1秒，以免父进程过于频繁地检查子进程状态，浪费系统资源。最后，根据waitpid函数的返回值判断子进程是否成功退出，如果成功退出则输出子进程的ID，否则输出错误信息。

kill

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/wait.h>

int main()
{
    pid_t pid;
    int status;

    if (!(pid = fork()))
    {
        printf("Hi I am child process!\n");
        sleep(100);
        exit(0); // 使用exit函数退出子进程
    }
    else
    {
        printf("send signal to child process (%d)\n", pid);
        sleep(1);
        kill(pid, SIGKILL);
        wait(&status);
        if (WIFSIGNALED(status))
            printf("child process receive signal %d\n", WTERMSIG(status));
    }
```

```
    return 0;
}
```

- WIFSIGNALED宏用于判断子进程是否因为接收到一个信号而终止。当子进程因为接收到一个信号而终止时，WIFSIGNALED宏返回一个非零值，否则返回0。
- WTERMSIG宏用于获取子进程终止时接收到的信号的编号。当子进程因为接收到一个信号而终止时，WTERMSIG宏返回该信号的编号，否则返回0。

信号量

```
// 导入sys/sem.h头文件，包含信号量相关的函数和数据类型
#include <sys/sem.h>

// 定义一个联合体semun，用于设置信号量的值
union semun {
    int val;    // 信号量的值
    struct semid_ds *buf;   // 用于IPC_STAT和IPC_SET命令的缓冲区
    unsigned short *array;  // 数组指针，指向一组信号量值
};

// 定义静态变量sem_id表示信号量的ID，用于标识信号量
static int sem_id = 0;

// 定义struct sembuf结构体，用于对信号量进行操作
struct sembuf sem_b;

// 定义静态函数set_semvalue，用于初始化信号量的值
static int set_semvalue() {
    union semun sem_union;  // 定义一个semun类型的联合体变量sem_union
    sem_union.val = 1;  // 将信号量的值初始化为1
    // 调用semctl函数设置信号量的值，如果失败则返回0
    if (semctl(sem_id, 0, SETVAL, sem_union) == -1) {
        return 0;
    }
    return 1;  // 设置信号量的值成功，返回1
}

// 定义静态函数del_semvalue，用于删除信号量
static void del_semvalue() {
    union semun sem_union;  // 定义一个semun类型的联合体变量sem_union
    // 调用semctl函数删除信号量，如果失败则输出错误信息
    if (semctl(sem_id, 0, IPC_RMID, sem_union) == -1) {
        fprintf(stderr, "Failed to delete semaphore\n");
    }
}

// 定义静态函数semaphore_p，用于对信号量做减1操作，即等待P(sv)
static int semaphore_p() {
    sem_b.sem_num = 0;  // 信号量数组中的位置，一般为0
    sem_b.sem_op = -1;  // 操作类型，-1表示P操作
    sem_b.sem_flg = SEM_UNDO;  // 操作标识符，表示如果进程结束时没有释放该信号量，则系统自动释放
```

```
    // 调用semop函数对信号量做减1操作，如果失败则输出错误信息并返回0
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_p failed\n");
        return 0;
    }
    return 1;  // 减1操作成功，返回1
}

// 定义静态函数semaphore_v，用于释放对共享资源的访问控制，即发送信号V(sv)
static int semaphore_v() {
    sem_b.sem_num = 0;  // 信号量数组中的位置，一般为0
    sem_b.sem_op = 1;  // 操作类型，1表示V操作
    sem_b.sem_flg = SEM_UNDO;  // 操作标识符，表示如果进程结束时没有释放该信号量，则系统自动释放
    // 调用semop函数释放对共享资源的访问控制，如果失败则输出错误信息并返回0
    if (semop(sem_id, &sem_b, 1) == -1) {
        fprintf(stderr, "semaphore_v failed\n");
        return 0;
    }
    return 1;  // 释放操作成功，返回1
}

// 主函数
int main() {
    // 创建信号量
    sem_id = semget((key_t)1234, 1, 0666 | IPC_CREAT);
    // 如果创建信号量失败，则输出错误信息并退出程序
    if (sem_id == -1) {
        fprintf(stderr, "Failed to create semaphore\n");
        exit(EXIT_FAILURE);
    }
    // 初始化信号量的值
    if (!set_semvalue()) {
        fprintf(stderr, "Failed to initialize semaphore\n");
        exit(EXIT_FAILURE);
    }
    // 在临界区内执行操作
    if (!semaphore_p()) {
        fprintf(stderr, "Failed to perform semaphore_p operation\n");
        exit(EXIT_FAILURE);
    }

    // 执行临界区操作...

    // 退出临界区
    if (!semaphore_v()) {
        fprintf(stderr, "Failed to perform semaphore_v operation\n");
        exit(EXIT_FAILURE);
    }

    // 删除信号量
    del_semvalue();

    return 0;
}
```

无名管道

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

int main()
{
   int d1[2]; // 定义管道d1
   int d2[2]; // 定义管道d2
   int d3[2]; // 定义管道d3
   int r,j,k; // 定义变量r、j、k
   char buff[200]; // 定义字符数组buff，用于存储读取的字符串

   printf("please input a string:");
   scanf("%s",buff); // 从标准输入读取一个字符串

   // 创建管道d1
   r=pipe(d1);
   if(r==-1)
   {
     printf("chuangjianguandaoshibai 1\n");
     exit(1);
   }

   // 创建管道d2
   r=pipe(d2);
   if(r=-1)
   {
     printf("chuangjianguandaoshibai 2\n");
     exit(1);
   }

   // 创建管道d3
   r=pipe(d3);
   if(r==-1)
   {
     printf("chuangjianguandaoshibai 3\n");
     exit(1);
   }

   // 创建子进程P1
   r=fork();
   if(r)
   {
     // 父进程P2
     close(d1[1]); // 关闭管道d1的写端
     read(d1[0],buff,sizeof(buff)); // 从管道d1中读取数据
     if(strlen(buff)%2==1) // 判断字符串长度是否为奇数
     {
       // 如果是奇数
```

```
        j=fork(); // 创建子进程P3
        if(j)
         {
           // 父进程P2
           close(d2[1]); // 关闭管道d2的写端
           read(d2[0],buff,sizeof(buff)); // 从管道d2中读取数据
           printf("p3 pipe2 odd length string: %s\n",buff); // 输出读取的字符串
           close(d2[0]); // 关闭管道d2的读端
           exit(0); // 退出进程P2
         }
        else
        {
           // 子进程P3
           close(d2[0]); // 关闭管道d2的读端
           write(d2[1],buff,strlen(buff)); // 将从管道d1中读取的字符串写入管道d2中
           printf("P2 finishes writing to pipe2.\n"); // 输出提示信息
           close(d2[1]); // 关闭管道d2的写端
           exit(0); // 退出进程P3
        }
      }
      else
      {
        // 如果是偶数
        k=fork(); // 创建子进程P4
        if(k)
        {
           // 父进程P2
           close(d3[1]); // 关闭管道d3的写端
           read(d3[0],buff,sizeof(buff)); // 从管道d3中读取数据
           printf("P4 pipe3 even length string:%s\n",buff); // 输出读取的字符串
           close(d3[0]); // 关闭管道d3的读端
           exit(0); // 退出进程P2
        }
        else
        {
          // 子进程P4
          close(d3[0]); // 关闭管道d3的读端
          write(d3[1],buff,strlen(buff)); // 将从管道d1中读取的字符串写入管道d3中
          printf("P2 finishes writing to pipe3.\n"); // 输出提示信息
          close(d3[1]); // 关闭管道d3的写端
          exit(0); // 退出进程P4
        }
      }
    }
    else
   {
      // 子进程P1
      close(d1[0]); // 关闭管道d1的读端
      write(d1[1],buff,strlen(buff)); // 将从标准输入读取的字符串写入管道d1中
      close(d1[1]); // 关闭管道d1的写端
      exit(0); // 退出进程P1
    }
}
```

消息队列

```
# include <sys/types.h>
# include <sys/ipc.h>
# include <sys/msg.h>
# include <stdio.h>
# include <unistd.h>
# define MSGKEY 75 // 定义消息队列的键值

struct msgform
{
   long mtype; // 消息类型
   char mtext[256]; // 消息内容
};

Int main()
{
   struct msgform msg; // 定义消息结构体
   int msgqid,pid,*pint; // 定义消息队列ID、进程ID以及指向消息内容的整型指针

   // 获取消息队列ID
   msgqid=msgget(MSGKEY,0777);

   // 获取当前进程的ID
   pid=getpid();

   printf("client:pid=%d\n",pid);

   // 将当前进程的ID写入消息内容中
   pint=(int*)msg.mtext;
   *pint=pid;

   // 设置消息类型为1，并将消息发送到消息队列中
   msg.mtype=1;
   msgsnd(msgqid,&msg,sizeof(int),0);

   // 从消息队列中接收消息
   msgrcv(msgqid,&msg,256,pid,0);

   // 从消息内容中读取进程ID并输出
   printf("client:receive from pid%d\n",*pint);
}
```

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

#define MSGKEY 75 // 定义消息队列的键值
```

```c
struct msgform
{
    long mtype; // 消息类型
    char mtext[256]; // 消息内容
};

int msgqid; // 消息队列ID

void cleanup()
{
    msgctl(msgqid, IPC_RMID, 0); /*删除队列*/
    exit(0);
}

int main()
{
    struct msgform msg; // 定义消息结构体
    int pid, *pint, i;

    for (i = 0; i < 23; i++)
        signal(i, cleanup); // 注册信号处理函数

    // 获取消息队列ID
    msgqid = msgget(MSGKEY, 0777 | IPC_CREAT);

    printf("server : pid = % d\n", getpid());

    for (;;)
    {
        // 从消息队列中接收消息
        msgrcv(msgqid, &msg, 256, 1, 0);

        // 从消息内容中读取客户端的进程ID
        pint = (int *)msg.mtext;
        pid = *pint;

        printf("server: receive from pid %d\n", pid);

        // 将服务端的进程ID写入消息内容中，并将消息发送回客户端
        msg.mtype = pid;
        *pint = getpid();
        msgsnd(msgqid, &msg, sizeof(int), 0);
    }
}
```

共享存储

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <unistd.h>

#define SHM_SIZE sizeof(int) // 定义共享内存的大小

int main(void)
{
    int shmid, *shmptr;
    pid_t pid;

    // 创建共享内存段
    if((shmid = shmget(IPC_PRIVATE, SHM_SIZE, IPC_CREAT | 0666)) == -1) {
        perror("shmget error");
        exit(EXIT_FAILURE);
    }

    // 将共享内存段附加到当前进程的地址空间中
    if((shmptr = (int *)shmat(shmid, 0, 0)) == (int *)-1) {
        perror("shmat error");
        exit(EXIT_FAILURE);
    }

    // 读取共享内存段的初始值
    printf("Input an initial value for *shmptr: ");
    scanf("%d", shmptr);

    // 创建子进程
    pid = fork();
    if(pid == -1) {
        perror("fork error");
        exit(EXIT_FAILURE);
    }

    if(pid == 0) {
        // 子进程从共享内存段中读取值并修改
        printf("When child runs, *shmptr=%d\n", *shmptr);
        printf("Input a value in child: ");
        scanf("%d", shmptr);
        printf("*shmptr=%d\n", *shmptr);
    } else {
        // 父进程等待子进程结束，并输出共享内存段的当前值
        wait(NULL);
        printf("After child runs, in parent, *shmptr=%d\n", *shmptr);

        // 删除共享内存段
        if(shmctl(shmid, IPC_RMID, NULL) == -1) {
            perror("shmctl error");
            exit(EXIT_FAILURE);
        }
    }

    return 0;
}
```

线程

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

int num = 100;

// 子线程函数
void *myfunc(void *arg)
{
    printf("child pthread id = %ld\n", pthread_self());
    for (int i = 0; i < 5; i++)
    {
        printf("child pthread i = %d\n", i);
        if (i == 2)
        {
            num = 666; // 验证不同线程可以利用全局变量通信
            // pthread_exit(NULL);  // 不携带数据的退出
            pthread_exit(&num); // 携带数据的退出
        }
    }
    return NULL;
}

int main()
{
    int ret;
    int i = 0;
    pthread_t pthid;

    // 创建子线程
    ret = pthread_create(&pthid, NULL, myfunc, NULL);
    if (ret != 0) // 创建失败判断
    {
        printf("error number is %d\n", ret);
        printf("%s\n", strerror(ret));
        exit(EXIT_FAILURE);
    }

    printf("parent pthread id = %ld\n", pthread_self());

    // 动态申请内存
    void *ptr = malloc(sizeof(int));
    if (ptr == NULL)
    {
        perror("malloc failed");
        exit(EXIT_FAILURE);
    }

    void *tmp = ptr; // 用 tmp 指向申请的内存来操作内存，以防改变 ptr 的指向导致 free 时产生段错误
```

```
    // 等待子线程结束，并获取子线程的退出参数
    ret = pthread_join(pthid, &tmp);
    if (ret != 0)
    {
        printf("pthread_join failed: %s\n", strerror(ret));
        exit(EXIT_FAILURE);
    }

    printf("num = %d\n", *(int *)tmp);

    // 释放动态申请的内存
    free(ptr);
    ptr = NULL; // 指针指向 NULL 以防后续误操作

    while (i < 5)
    {
        i++;
        printf("parent pthread i = %d\n", i);
    }

    sleep(2);

    return 0;
}
```

1. `pthread_create` 函数示例：

```
#include <stdio.h>
#include <pthread.h>

void *myfunc(void *arg)
{
    printf("Hello, world!\n");
    return NULL;
}

int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid, NULL, myfunc, NULL);
    if (ret != 0)
    {
        printf("pthread_create error\n");
        return -1;
    }
    pthread_join(tid, NULL);
    return 0;
}
```

该示例中，使用 `pthread_create` 函数创建一个新线程，并执行 `myfunc` 函数。在 `main` 函数中，使用 `pthread_join` 函数等待新线程执行完毕，以保证程序正确执行。

2. `pthread_join` 函数示例：

```
#include <stdio.h>
#include <pthread.h>

void *myfunc(void *arg)
{
    int i;
    for (i = 0; i < 5; i++)
    {
        printf("child thread: %d\n", i);
        sleep(1);
    }
    return NULL;
}

int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid, NULL, myfunc, NULL);
    if (ret != 0)
    {
        printf("pthread_create error\n");
        return -1;
    }
    printf("parent thread waiting for child thread...\n");
    pthread_join(tid, NULL);
    printf("parent thread exit\n");
    return 0;
}
```

该示例中，使用 `pthread_join` 函数等待新线程执行完毕，并打印出相应的提示信息，以保证程序正确执行。

3. `pthread_exit` 函数示例：

```
#include <stdio.h>
#include <pthread.h>

void *myfunc(void *arg)
{
    printf("child thread exit\n");
    pthread_exit(NULL);
}

int main()
{
    pthread_t tid;
    int ret = pthread_create(&tid, NULL, myfunc, NULL);
    if (ret != 0)
    {
        printf("pthread_create error\n");
        return -1;
    }
    printf("parent thread waiting for child thread...\n");
```

```
    pthread_join(tid, NULL);
    printf("parent thread exit\n");
    return 0;
}
```

该示例中，使用 `pthread_exit` 函数退出子线程，并在主线程中使用 `pthread_join` 函数等待子线程执行完毕，以保证程序正确执行。

4. `pthread_mutex_init` 函数示例：

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex;

void *myfunc(void *arg)
{
    pthread_mutex_lock(&mutex);
    printf("Hello, world!\n");
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main()
{
    int ret = pthread_mutex_init(&mutex, NULL);
    if (ret != 0)
    {
        printf("pthread_mutex_init error\n");
        return -1;
    }
    pthread_t tid;
    ret = pthread_create(&tid, NULL, myfunc, NULL);
    if (ret != 0)
    {
        printf("pthread_create error\n");
        return -1;
    }
    pthread_join(tid, NULL);
    pthread_mutex_destroy(&mutex);
    return 0;
}
```

该示例中，使用 `pthread_mutex_init` 函数初始化互斥锁，并使用 `pthread_mutex_lock` 和 `pthread_mutex_unlock` 函数保证线程同步，以保证程序正确执行。

5. `pthread_cond_init` 函数示例：

```
#include <stdio.h>
#include <pthread.h>

pthread_mutex_t mutex;
pthread_cond_t cond;

void *myfunc(void *arg)
```

```
{
    pthread_mutex_lock(&mutex);
    printf("child thread waiting...\n");
    pthread_cond_wait(&cond, &mutex);
    printf("child thread wake up!\n");
    pthread_mutex_unlock(&mutex);
    return NULL;
}

int main()
{
    pthread_mutex_init(&mutex, NULL);
    pthread_cond_init(&cond, NULL);
    pthread_t tid;
    int ret = pthread_create(&tid, NULL, myfunc, NULL);
    if (ret != 0)
    {
        printf("pthread_create error\n");
        return -1;
    }
    sleep(3);
    pthread_mutex_lock(&mutex);
    printf("parent thread wake up child thread...\n");
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
    pthread_join(tid, NULL);
    pthread_mutex_destroy(&mutex);
    pthread_cond_destroy(&cond);
    return 0;
}
```

该示例中，使用 `pthread_cond_init` 函数初始化条件变量和互斥锁，并使用 `pthread_cond_wait` 和 `pthread_cond_signal` 函数实现线程间的通信，以保证程序正确执行。在主线程中使用 `sleep` 函数暂停一段时间，以便在子线程执行 `pthread_cond_wait` 函数时等待一段时间。+

实践

编写一段 C 语言程序使其完成：父进程创建两个子进程，父子进程都在屏幕上显示自己的进程 ID 号。要求先显示子进程的 ID 号， 后显示父进程的 ID 号。（6分）

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

int main(){
    pid_t p1 = fork();
    if (p1 == 0) {
        printf("sub process1: %d\n", getpid());
        exit(0);
    } else {
        wait(NULL);
```

```
        pid_t p2 = fork();
        if (p2 == 0) {
            printf("sub process2: %d\n", getpid());
            exit(0);
        } else {
            wait(NULL);
            printf("main process: %d\n", getpid());
        }
    }
    return 0;
}
```

编写利用 IPC 实现进程通信的 C 程序。该程序主要模拟根据帐号查询余额 的过程。包括三方面：

1)请求进程从标准输入读入帐号，并将该帐号通过消息队列发送给服务进程；

2)服务进程接收该帐号后，按照请求的先后顺序在标准输入上输入该帐户的姓名和余额，并将结果返回给请求进程；

3)请求进程接收返回的信息，并将结果输出在标准输出上。服务进程先于请求进程启动， 请求进程启动时要携带请求编号，可同时启动多个请求进程。（7 分)

请求进程

```
#define MSG_SIZE 50
typedef struct message {
  long type;
  int pid;
  char text[MSG_SIZE];
} Message;

void UsingMessageQueue(){
  key_t key;
  int msgid;
  Message message;
  char data_buffer[MSG_SIZE];
  // 生成key
  // 将当前目录和一个字符 'a' 转换成一个唯一的键值，该键值将作为消息队列的标识符
  if ((key = ftok(".", 'a')) < 0) {
    perror("ftok error");
    exit(1);
 }
  // 创建消息队列
  if ((msgid = msgget(key, IPC_CREAT | 0666)) < 0) {
    perror("msgget error");
    exit(1);
 }
  message.pid = getpid();
  // 循环读取用户输入
  while (1) {
    printf("%d input a account\n", getpid());
    fgets(data_buffer, MSG_SIZE, stdin);
    if (strcmp(data_buffer, "q!\n") == 0) {
```

```
        break;
    }
     // 发送消息
     message.type = 1;
     strncpy(message.text, data_buffer, MSG_SIZE);
     if (msgsnd(msgid, &message, MSG_SIZE, 0) < 0) {
       perror("msgsnd error");
       exit(1);
    }
  }
}

int main() {
     // 消息队列初始化
     UsingMessageQueue();
}
```

服务进程

```
#define MSG_SIZE 50
typedef struct message {
  long type;
  int pid;
  char text[MSG_SIZE];
} Message;


void UsingMessageQueue(){
  key_t key;
  int msgid;
  Message message;
  // 生成key
  // 将当前目录和一个字符 'a' 转换成一个唯一的键值，该键值将作为消息队列的标识符
  if ((key = ftok(".", 'a')) < 0) {
    perror("ftok error");
    exit(1);
  }
  // 获取消息队列
  if ((msgid = msgget(key, 0666)) < 0) {
    perror("msgget error");
    exit(1);
  }
  // 循环接收消息
  while (1) {
    if (msgrcv(msgid, &message, MSG_SIZE, 1, 0) < 0) {
      perror("msgrcv error");
      exit(1);
    }
     printf("Received account from pid %d: %s", message.pid, message.text);
     if (!strcmp(message.text, "end")) {
       break;
    }
     cin
```

```
 }
  // 删除消息队列
   if (msgctl(msgid, IPC_RMID, NULL) < 0) {
    perror("msgctl error");
    exit(1);
 }
}

int main() {
    // 消息队列初始化
    UsingMessageQueue();
}
```