目录

[实验1 进程管理 2](#_Toc22520)

[1. 函数使用 2](#_Toc25636)

[（1） fork() 2](#_Toc14753)

[（2） execve() 3](#_Toc18585)

[（3） getpid() 4](#_Toc13591)

[（4） wait() 4](#_Toc3164)

[（5） waitpid() 5](#_Toc13695)

[2. 实验结果 6](#_Toc23987)

[实验2 处理器调度 10](#_Toc10482)

# 实验1 进程管理

## 函数使用

### fork()

**函数原型**

pid\_t fork( void);（pid\_t 是一个[宏定义](https://baike.baidu.com/item/%E5%AE%8F%E5%AE%9A%E4%B9%89" \t "https://baike.baidu.com/item/fork/_blank)，其实质是int 被定义在#include<sys/types.h>中）

返回值： 若成功调用一次则返回两个值，子进程返回0，[父进程](https://baike.baidu.com/item/%E7%88%B6%E8%BF%9B%E7%A8%8B" \t "https://baike.baidu.com/item/fork/_blank)返回子进程ID；否则，出错返回-1

**函数说明**

一个现有进程可以调用fork函数创建一个新进程。由fork创建的新进程被称为子进程（child process）。fork函数被调用一次但返回两次。两次返回的唯一区别是子进程中返回0值而[父进程](https://baike.baidu.com/item/%E7%88%B6%E8%BF%9B%E7%A8%8B" \t "https://baike.baidu.com/item/fork/_blank)中返回子进程ID。

子进程是父进程的副本，它将获得父进程数据空间、堆、栈等资源的副本。注意，子进程持有的是上述存储空间的“副本”，这意味着**父子进程间不共享这些存储空间**。

**函数使用示例**

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid = fork();

if(pid < 0){

printf("error!!");

} else if(pid == 0){

printf("子进程空间");

exit(0);

} else{

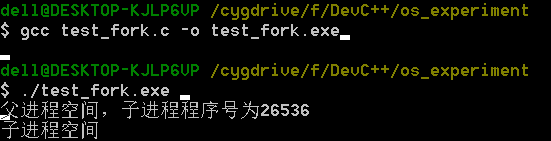
printf("父进程空间，子进程程序号为%d\n", pid);

}

exit(0);

}

结果：



### execve()

**函数定义**

int execve(const char \* filename,char \* const argv[ ],char \* const envp[ ]);

**函数说明**

filename：程序所在的路径

argv：传递给程序的参数，数组指针argv必须以程序(filename)开头，NULL结尾

envp：传递给程序的新环境变量，无论是shell脚本，还是可执行文件都可以使用此环境变量，必须以NULL结尾

**返回值**

如果执行成功则函数不会返回，执行失败则直接返回-1，失败原因存于errno 中。

**函数使用示例**

**test\_execve\_testfile.c**

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <unistd.h>

extern char \*\*environ;

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

{

int i = 0;

//打印envp，新环境变量

for( i = 0; environ[i] != NULL; i++ )

{

printf("environ[%d]:%s\r\n",i,environ[i]);

}

printf("\r\n\r\n");

//打印argv，参数

for( i = 0; i < argc; i++ )

{

printf("argv[%d]:%s\r\n",i,argv[i]);

}

return 0;

}

**test\_execve.c**

#include<unistd.h>

main()

{

char \*envp[] = {"T1=222","T2=333",NULL};

char \*argv\_send[] = {"./test\_execve\_testfile","1","2",NULL};

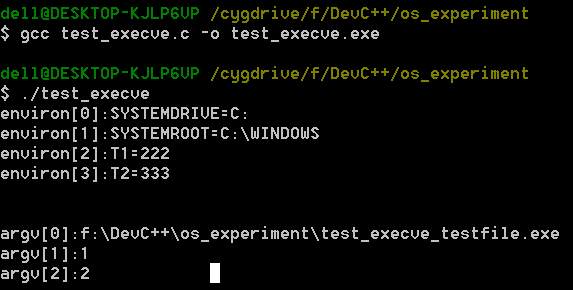
int a = execve("./test\_execve\_testfile",argv\_send,envp);

printf("%d", a);

return 0;

}

结果：



### getpid()

**函数原型**

旧的原型为pid\_t getpid(void);，推荐使用int \_getpid( void );这种形式。注意，函数名第一个字符是下划线。

**函数说明**

getpid函数用来取得目前进程的进程[ID](https://baike.baidu.com/item/ID" \t "https://baike.baidu.com/item/getpid/_blank)，许多程序利用取到的此值来建立[临时文件](https://baike.baidu.com/item/%E4%B8%B4%E6%97%B6%E6%96%87%E4%BB%B6" \t "https://baike.baidu.com/item/getpid/_blank)，以避免临时文件相同带来的问题。

**返回值**

目前进程的进程ID

**函数使用示例**

#include <stdio.h>

int main( void )

{

printf( "Process id: %d\n", \_getpid() );

return 0;

}

结果



### wait()

**函数原型**

pid\_t wait (int \* status);

**函数说明**

等待子进程中断或结束，参数 status 是一个整形指针。如果status不是一个空指针,则终止进程的终止状态将存储在该指针所指向的内存单元中。如果不关心终止状态,可以将 status参数设置为NULL。

**status 不是NULL时子进程的结束状态值会由参数 status 返回，**而**子进程的进程识别码作为函数返回值返回。**

调用 wait 函数时,调用进程将会出现下面的情况:

· 如果其所有子进程都还在运行,则阻塞。

· 如果一个子进程已经终止,正等待父进程获取其终止状态,则获取该子进程的终止状态然后立即返回。

· 如果没有任何子进程,则立即出错返回。

**返回值**

如果执行成功则返回子进程识别码（PID），如果有错误发生则返回-1。失败原因存于errno 中。

**函数使用示例**

#include <unistd.h>

#include <stdlib.h>

int main()

{

pid\_t pc,pr;

pc=fork();

if(pc<0) /\* 如果出错 \*/

printf("error ocurred!/n");

else if(pc==0){ /\* 如果是子进程 \*/

printf("This is child process with pid of %d\n",getpid());

sleep(10); /\* 睡眠10秒钟 \*/

}

else{ /\* 如果是父进程 \*/

pr=wait(NULL); /\* 在这里等待 \*/

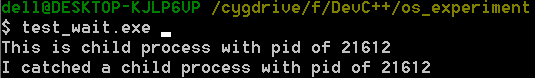
printf("I catched a child process with pid of %d\n",pr);

}

exit(0);

}

结果



### waitpid()

**函数原型**

pid\_t waitpid(pid\_t pid,int \*status,int options)

**函数说明**

如果在调用 waitpid()时子进程已经结束,则 waitpid()会立即返回子进程结束状态值。 子进程的结束状态值会由参数 status 返回, 而子进程的进程识别码也会一起返回。如果不在意结束状态值,则参数 status 可以设成 NULL。参数 pid 为欲等待的子进程识别码,

其他数值意义如下:

pid<-1 等待[进程组](https://baike.baidu.com/item/%E8%BF%9B%E7%A8%8B%E7%BB%84" \t "https://baike.baidu.com/item/waitpid/_blank)识别码为 pid 绝对值的任何子进程。

pid=-1 等待任何子进程,相当于 wait()。

pid=0 等待进程组识别码与目前进程相同的任何子进程。

pid>0 等待任何子进程识别码为 pid 的子进程。

参数options提供了一些额外的选项来控制waitpid，最有用的一个选项是WNOHANG，它的作用是**防止waitpid把调用者的执行挂起**，WNOHANG：若pid指定的子进程没有结束，则waitpid()函数返回0，不予以等待。若结束，则返回该子进程的ID。

**返回值**

成功返回等待子进程的pid，失败返回-1

**函数使用示例**

#include<stdio.h>

#include<unistd.h>

#include<sys/wait.h>

int main()

{

pid\_t pc,pr;

pc=fork();

if(pc<0) /\* 如果出错 \*/

printf("error ocurred!/n");

else if(pc==0){ /\* 如果是子进程 \*/

printf("This is child process with pid of %d\n",getpid());

sleep(10); /\* 睡眠10秒钟 \*/

}

else{ /\* 如果是父进程 \*/

while(pr == 0)

pr=waitpid(pc, NULL, WNOHANG); /\* 在这里等待 \*/

printf("process id %d do not release \n", pc); /\*并不会在上一行等待\*/

printf("I catched a child process with pid of %d\n",pr);

}

exit(0);

}

结果



## 实验结果

1. **每个进程都执行自己独立的程序，打印自己的pid，每个父进程打印其子进程的pid;**

**expriment1\_1.c代码**

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid1, pid2;

pid1 = fork(); // <--第一次fork

if(pid1 < 0){

printf("error!!");

} else if(pid1 == 0){

pid2 = fork(); // <--第二次fork

if(pid2 < 0){

printf("error!!");

} else if(pid2 == 0){

printf("\n子进程2pid为%d,子进程2无子进程pid\n", \_getpid());

} else{

printf("子进程1pid为%d,子进程1的子进程pid为%d\n", \_getpid(), pid2);

}

} else{

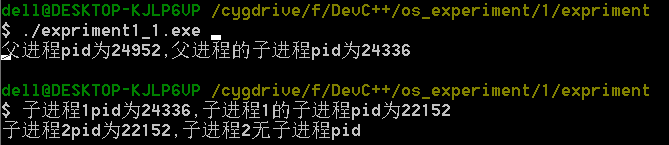
printf("父进程pid为%d,父进程的子进程pid为%d\n", \_getpid(), pid1);

}

exit(0);

}

**结果**



1. **每个进程都执行自己独立的程序，打印自己的pid，父进程打印其子进程的pid;**

**expriment1\_2.c代码**

#include<stdio.h>

#include<unistd.h>

int main()

{

pid\_t pid0, pid1, pid2;

pid0 = \_getpid();

pid1 = fork(); // <--第一次fork

if(pid1 < 0){

printf("error!!");

} else if(pid1 == 0){

printf("子进程1pid为%d\n", \_getpid());

} else{

pid2 = fork(); // <--第二次fork

if(pid2 < 0){

printf("error!!");

} else if(pid2 == 0){

printf("子进程2pid为%d\n", \_getpid());

} else{

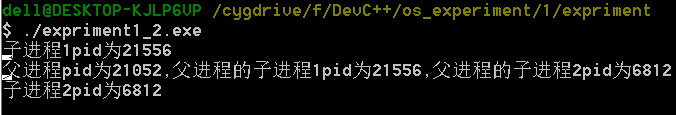
printf("父进程pid为%d,父进程的子进程1pid为%d,父进程的子进程2pid为%d\n", pid0, pid1, pid2);

}

}

}

**结果**



**（3）编写一个命令处理程序，能处理max(m,n), min(m,n)和average(m,n,l)这几个命令。（使用exec函数族）**

**expriment1\_3\_main.c代码**

#include<unistd.h>

#include<string.h>

#include<sys/wait.h>

#define N 10

main()

{

char function\_name[N];

char \*argv\_send[] = {NULL};

char \*envp[] = {NULL};

printf("pid %d(max,min,avg):", \_getpid());

scanf("%s", function\_name);

if(strcmp(function\_name, "min") == 0){

execl("./expriment1\_3\_min", NULL);

} else if(strcmp(function\_name, "max") == 0){

execve("./expriment1\_3\_max", argv\_send, envp);

} else if(strcmp(function\_name, "avg") == 0){

execve("./expriment1\_3\_avg", argv\_send, envp);

}

}

**expriment1\_3\_max.c代码**

**#include <stdio.h>**

**#include <unistd.h>**

**int main()**

**{**

**float m,n;**

**printf("pid %d请输入要求最大值的2个数：", \_getpid());**

**scanf("%f%f", &m, &n);**

**if(m > n)**

**printf("%f", m);**

**else if(m < n)**

**printf("%f", n);**

**else**

**printf("一样大");**

**}**

**expriment1\_3\_min.c代码**

#include <stdio.h>

#include <unistd.h>

int main()

{

float m,n;

printf("pid %d请输入要求最小值的2个数：", \_getpid());

scanf("%f%f", &m, &n);

if(m > n)

printf("%f", n);

else if(m < n)

printf("%f", m);

else

printf("一样大");

}

**expriment1\_3\_avg.c代码**

#include <stdio.h>

#include <unistd.h>

int main()

{

float m, n, l;

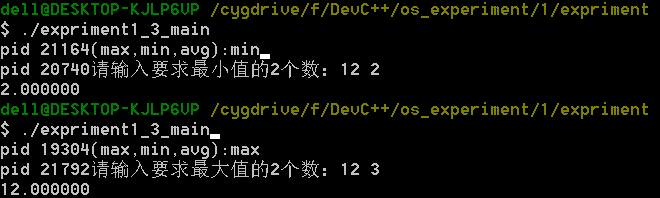
printf("pid %d请输入要求平均值的3个数：", \_getpid());

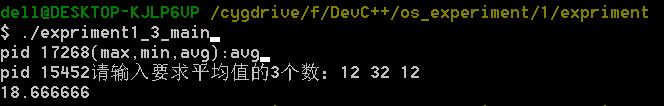
scanf("%f%f%f", &m, &n, &l);

printf("%f", (m + n + l) / 3);

}

**结果**





（在我的电脑上，并没有像“2020实验课题目.docx”中所说的“Exec执行成功后将用一个新的程序代替原进程，但进程号不变，它绝不会再返回到调用进程了。”，如上图所示，使用execve和execl函数调用的新程序的进程号都变了）。

# 实验2 处理器调度