

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

#include<sys/wait.h>

int main()

{

pid\_t pid,pid1;

//fork a child process

printf("\n");

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

printf("child: pid = %d\n",pid);//A

printf("child: pid1 = %d\n",pid1);//B

}

else

{

//parent process

pid1 = getpid();

printf("parent: pid = %d\n",pid);//C

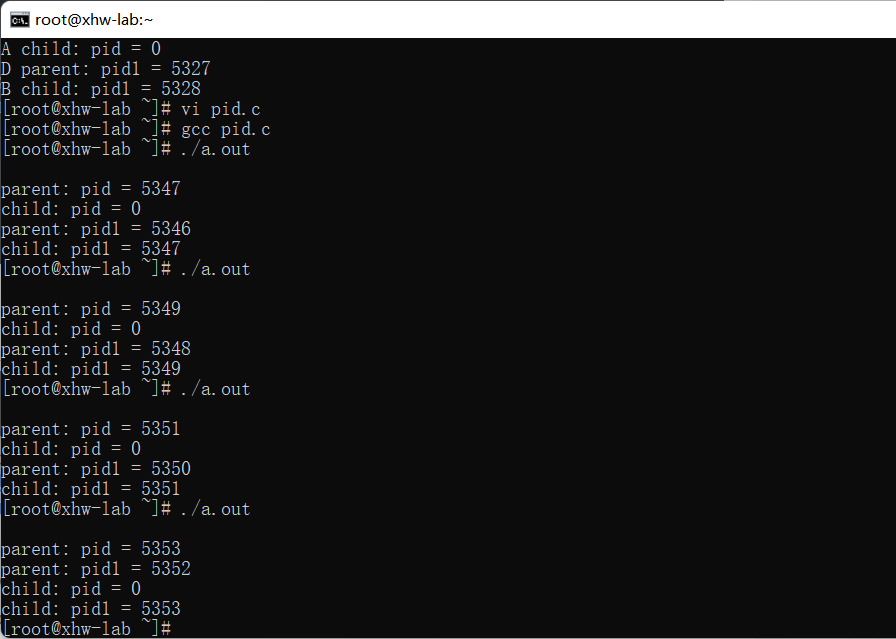
printf("parent: pid1 = %d\n",pid1);//D

wait(NULL);

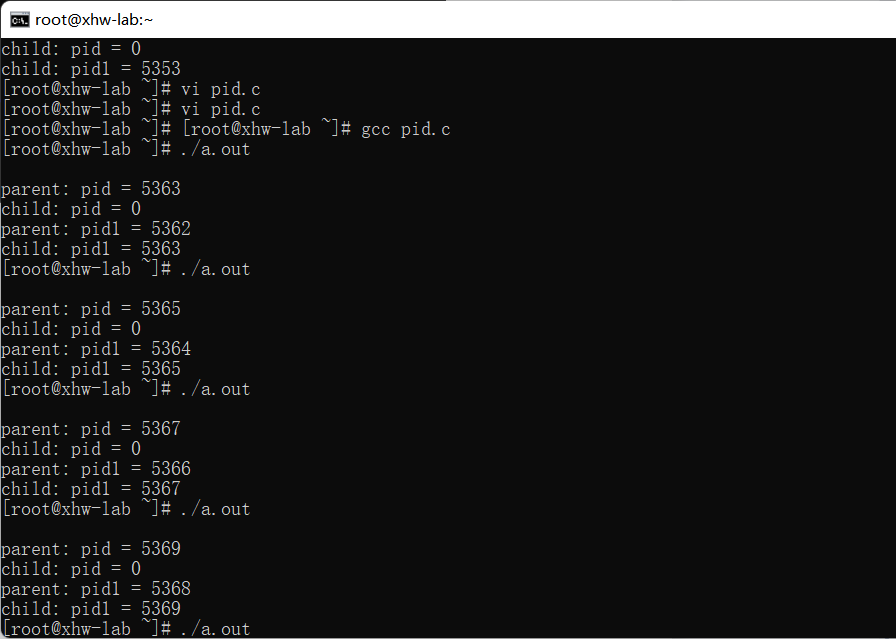
}

return 0;

}



去除掉wait后



同样也是既有可能child先执行，又有可能parent先执行

在去掉wait()前后，都是既有可能parent先输出，又有可能child先输出，这是因为父子进程的执行顺序是不确定的，既有可能父进程先执行，也有可能子进程先执行。wait()的作用是让父进程挂起等待子进程结束。所以只有将wait()放在父进程打印语句之前才能确保子进程先输出。

1. 定义全局变量并观察结果

定义一个a，初始为0，子进程自增1 ，父进程自减1

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int a = 0;

int main()

{

pid\_t pid,pid1;

//fork a child process

printf("\n");

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

a++;

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

printf("child: pid1 = %d\n",pid1);

printf("a: %d\n",a);

}

else

{

//parent process

pid1 = getpid();

a--;

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

printf("parent: pid1 = %d\n",pid1);

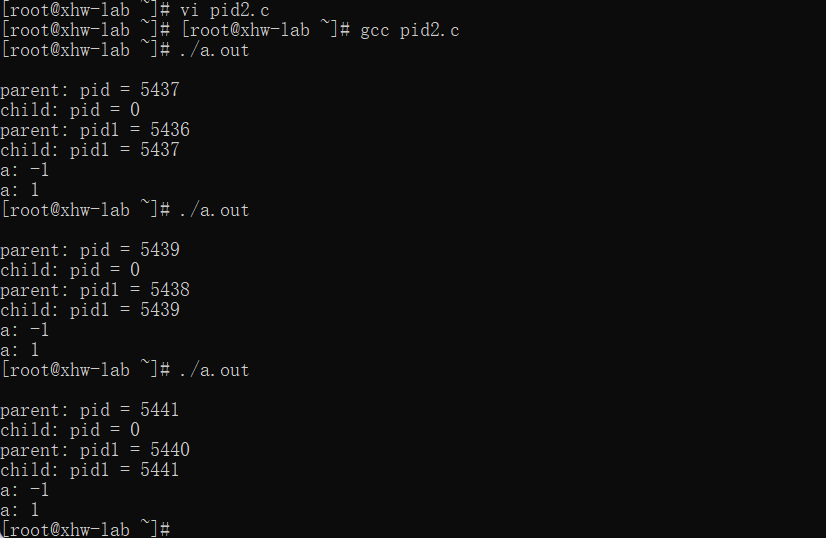
printf("a: %d\n",a);

wait(NULL);

}

return 0;

}



子进程在创建时，继承了全局变量，虽然这两个变量都是a，但是它们其实已经变成了两个，执行时在各自的物理空间内执行。子进程对该全局变量的操作是相互独立的，因此会分别对a进行操作，打印出的a值不同

1. 在return前加一个操作使a自增2

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int a = 0;

int main()

{

pid\_t pid,pid1;

//fork a child process

printf("\n");

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

a++;

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

printf("child: pid1 = %d\n",pid1);

printf("a: %d\n",a);

}

else

{

//parent process

pid1 = getpid();

a--;

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

printf("parent: pid1 = %d\n",pid1);

printf("a: %d\n",a);

wait(NULL);

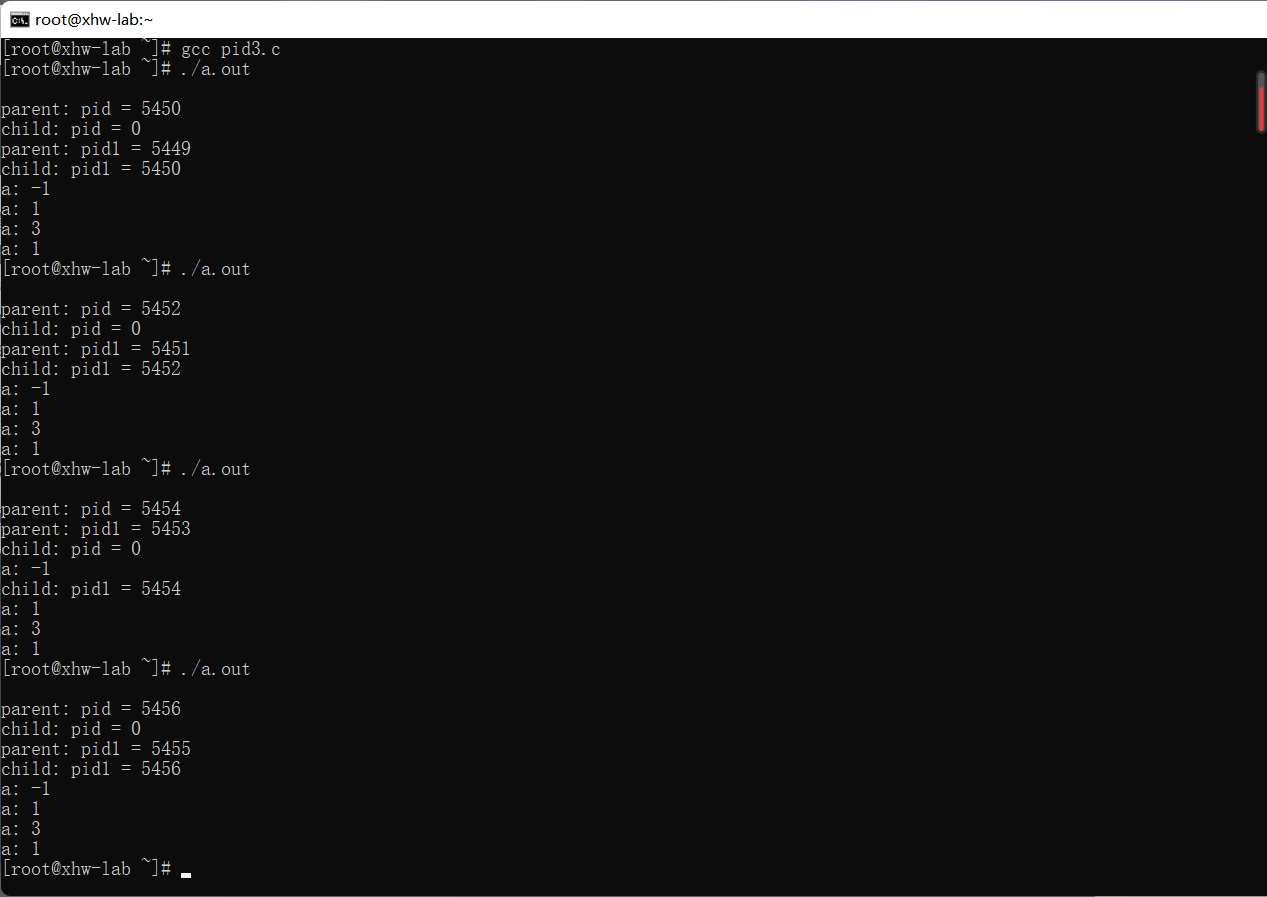
}

a+=2;

printf("a: %d\n",a);

return 0;

}



3.在子进程中调用system函数

在子进程中通过system()函数调用事先编译好的文件

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int main()

{

pid\_t pid,pid1;

//fork a child process

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

pid1 = getpid();

printf("child: pid1 = %d\n",pid1);

system("./call");

printf("子进程不输出\n");

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

}

else

{

//parent process

pid1 = getpid();

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

printf("parent: pid1 = %d\n",pid1);

wait(NULL);

}

return 0;

}

Call.c:

#include<unistd.h>

# include<stdio.h>

# include <sys/types.h>

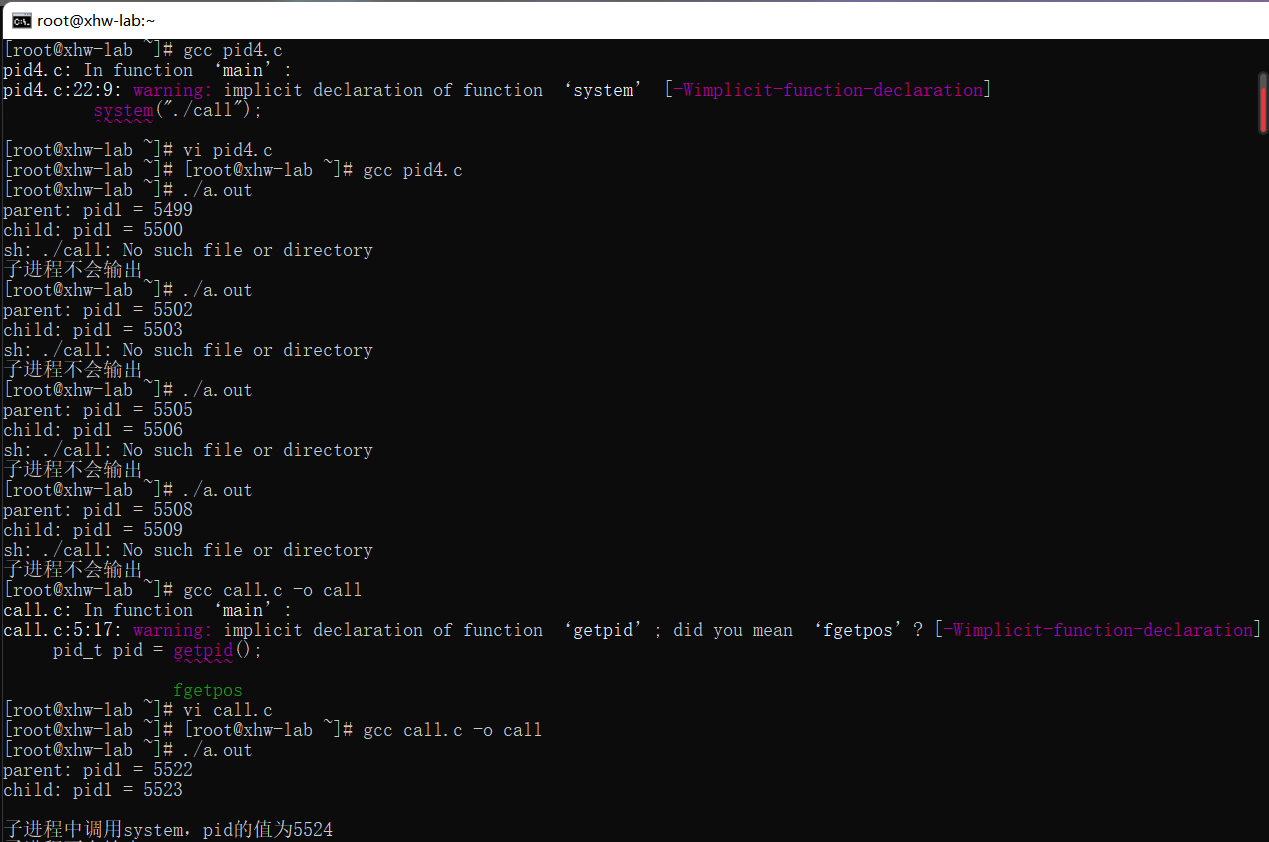
int main()

{

pid\_t pid = getpid();

printf("\n子进程中调用system，pid的值为%d\n",pid);

}



调用system()之后pid和之前有所不同，因为system()函数先执行了fork()函数，新产生的子进程立刻执行了exec()函数，产生了一个新进程，新进程的pid等与原进程不同。

4.在子进程中调用exec函数

# include <sys/types.h>

# include <stdio.h>

# include <unistd.h>

# include <sys/wait.h>

int main()

{

pid\_t pid,pid1;

//fork a child process

pid = fork();

if(pid < 0)

{

//error occured

fprintf(stderr, "Fork Failed");

return 1;

}

else if (pid == 0)

{

//child process

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

pid1 = getpid();

printf("child: pid1 = %d\n",pid1);

execl("./call","");

printf("子进程不输出\n");

}

else

{

//parent process

pid1 = getpid();

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

printf("parent: pid1 = %d\n",pid1);

wait(NULL);

}

return 0;

}

1. 使用pthread实现多线程

使用pthread\_t创建线程号，pthread\_create()创建线程，pthread\_join()来等待线程执行完成

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void\* threadfunc(void\* tid);

int main(void){

pthread\_t threads[2];

int status,i;

for(i=0;i<2;i++){//循环创建10个现场

printf("主函数中创建第%d个线程\n",i+1);

status=pthread\_create(&threads[i],NULL,threadfunc,(void\*)i);

if(status!=0){

printf("线程创建失败 %d\n",status);

exit(-1);

}

}

for(i=0;i<2;i++){

pthread\_join(threads[i],NULL);

}

exit(0);

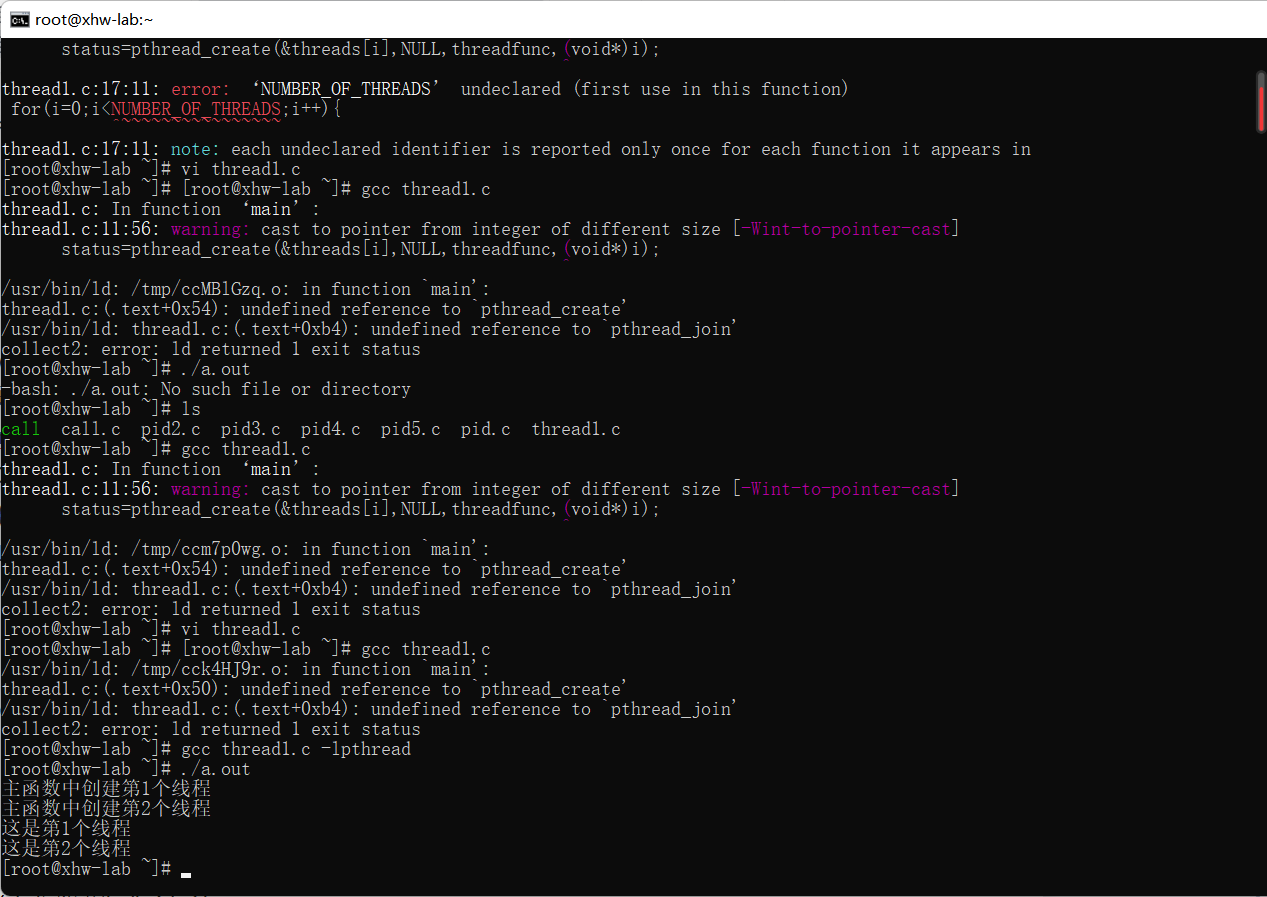
}

void\* threadfunc(void\* tid){

printf("这是第%d个线程\n",tid+1);

pthread\_exit(0);

}



#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

int a = 1; // 全局变量

void run() { int i=0; while(i<5002){ a=a+1;i++;printf("%d\n",a);} }

void run2() { int i=0; while(i<5002){ a=a-1;i++;printf("%d\n",a);} }

int main() {

int tmp1, tmp2;

pthread\_t thread1, thread2;

int ret\_thrd1, ret\_thrd2;

ret\_thrd1 = pthread\_create(&thread1, NULL, (void \*)&run, NULL);

if (ret\_thrd1 != 0) {

printf("thread1 create error\n");

}

else {

printf("thread1 create success\n");

}

pthread\_join(thread1,NULL);

ret\_thrd2 = pthread\_create(&thread2, NULL, (void \*)&run2, NULL);

if (ret\_thrd2 != 0) {

printf("thread2 create error\n");

} else {

printf("thread2 create success\n");

}

pthread\_join(thread2,NULL);

}

去除pthread\_join()函数,线程之间的同步关系被打破了，线程thread1还未执行完，就被thread2强占了资源

代码中如果没有pthread\_join主线程会很快结束从而使整个进程结束，从而使创建的线程没有机会开始执行就结束了

加入pthread\_join后，主线程会一直等待直到等待的线程结束自己才结束，而在今天的例子中，主线程会等待线程的结束才继续执行下面的语句（创建thread2），使创建的线程有机会执行。