

OpenEuler进程创建与变量独立性实验

获取PID实验

1.创建源代码文件

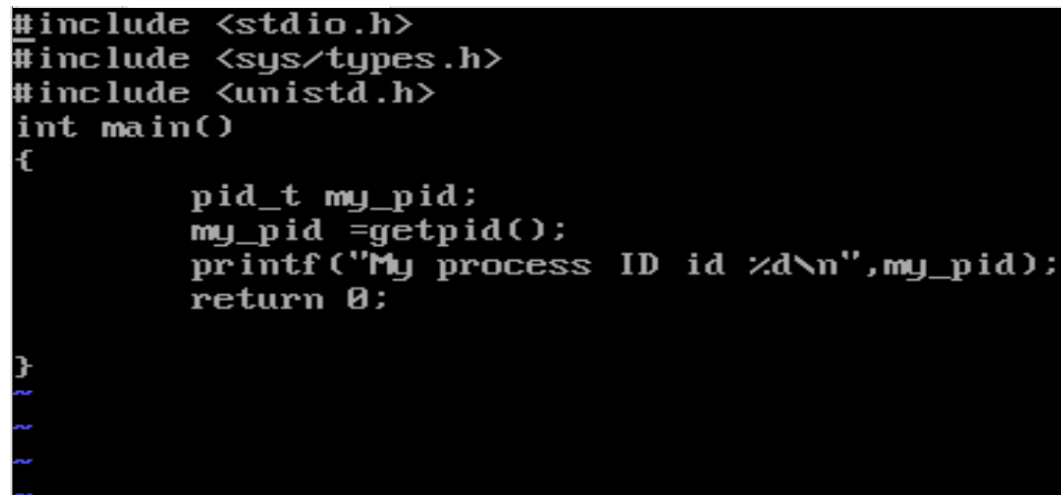
```
1 | vi yi.cpp
```

2.进入文件编写代码

进入文件以后按“a”键进入编辑模式
在yi.cpp中编写以下代码

```
1  #include<stdio.h>
2  #include<sys/types.h>
3  #include<unistd.h>
4
5  int main()
6  {
7      pid_t my_pid;
8      my_pid = getpid();
9      printf("My process ID is %d\n", my_pid);
10
11     return 0;
12 }
```

如图所示：



```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    pid_t my_pid;
    my_pid =getpid();
    printf("My process ID id %d\n",my_pid);
    return 0;
}
~
~
~
~
```

按 **ESC** 退出编辑模式

按住 **shift** + **:** 并输入wq

按下回车键退出文件

编译并运行代码

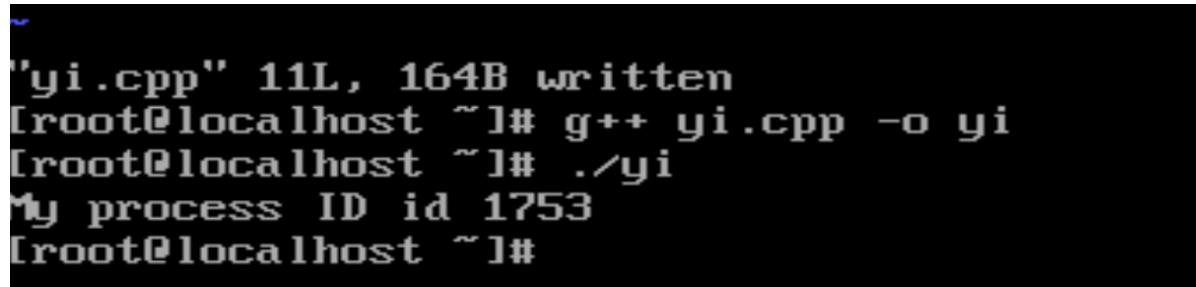
使用如下代码编译代码

```
1 | g++ yi.cpp -o yi
```

运行程序

```
1 | ./yi
```

输出结果如图所示：



```
'yi.cpp' 11L, 164B written
[root@localhost ~]# g++ yi.cpp -o yi
[root@localhost ~]# ./yi
My process ID id 1753
[root@localhost ~]#
```

获取到的当前进程号为1753

进程创建与父子进程关系实验

1.创建源代码文件

创建文件 `er`

```
1 | vi er.cpp
```

2.输入代码

```
1 | #include <stdio.h>
2 | #include <sys/types.h>
3 | #include <unistd.h>
4 | #include <sys/wait.h>
5 | int main ()
6 | {
7 |     pid_t child_pid;
8 |     child_pid fork();
9 |     if( child_pid < 0 )
10 |    {
11 |        perror("Fork failed");
12 |        return 1;
13 |    }
14 |     else if( child_pid == 0 )
15 |     printf("Child process:My PID is %d \n",getpid() );
16 |     else
17 |     {
18 |         printf ("Parent process:Child Process ID is %d \n ",child_pid);
19 |         int status;
20 |         waitpid(child_pid,&status,0);
21 |         if (WIFEXITED(status))
22 |             printf ('Parent process:Child exited with status %d
\n",WEXITSTATUS(status));
```

```
23     }
24
25     return 0;
26 }
```

如图所示:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    pid_t child_pid;
    child_pid = fork();
    if(child_pid < 0)
    {
        perror("Fork failed");
        return 1;
    }
    else if(child_pid == 0)
    {
        printf("Child process: My PID is %d \n",getpid());
    }
    else
    {
        printf("Parent process: My PID is %d \n ",getpid());
        printf("Parent process: Child process ID is %d \n",child_pid);
    }
    return 0;
}
```

3.编译并运行代码

编译代码

```
1 g++ er.cpp -o er
```

运行程序

```
1 ./er
```

输出结果如图所示:

```
"er.cpp" 23L, 406B written
[root@localhost ~]# g++ er.cpp -o er
[root@localhost ~]# ./er
Parent process: My PID is 1669
Child process: My PID is 1670
Parent process: Child process ID is 1670
[root@localhost ~]#
```

`fork()` 执行成功以后父进程会产生一个子进程

父进程会输出自己的进程号和子进程号, 而子进程只输出自己进程号

父进程等待子进程退出测试

1.修改 er.cpp 的代码

```
1 | vi er.cpp
```

修改为以下代码

```
1  #include <stdio.h>
2  #include <sys/types.h>
3  #include <unistd.h>
4  #include <sys/wait.h>
5
6  int main()
7  {
8      pid_t child_pid;
9      child_pid = fork();
10     if (child_pid < 0)
11     {
12         perror("Fork failed");
13         return 1;
14     }
15     else if (child_pid == 0)
16     {
17         printf("Child process:My PID is %d \n", getpid());
18     }
19     else
20     {
21         printf("Parent process: Child process ID is %d \n", child_pid);
22         int status;
23         waitpid(child_pid, &status, 0);
24         if (WIFEXITED(status))
25         {
26             printf("Parent process: Child exited with status %d\n",
WEXITSTATUS(status));
27         }
28     }
29     return 0;
30 }
```

如图所示:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <sys/wait.h>
int main()
{
    pid_t child_pid;
    child_pid = fork();
    if(child_pid < 0)
    {
        perror("Fork failed");
        return 1;
    }
    else if(child_pid == 0)
    {
        printf("Child process:My PID is %d \n",getpid());
    }
    else
    {
        printf("Parent process: Child Process ID is %d \n ",child_pid);
        int status;
        waitpid(child_pid,&status,0);
        if (WIFEXITED(status))
        {
            printf("Parent process: Child exited with status %d \n",WEXITSTATUS(status));
        }
    }
    return 0;
}
```

2.运行代码

编译代码

```
1 | g++ er.cpp -o er
```

运行代码

```
1 | ./er
```

得到结果如下：

```
[root@localhost ~]# ./er
Parent process: Child Process ID is 1732
Child process:My PID is 1732
Parent process: Child exited with status 0
```

父进程在调用 `waitpid()` 后进入等待状态，知道子进程正常退出以后继续执行代码

多次fork()进程创建实验

1.编写代码

```
1 | #include<stdio.h>
2 | #include<sys/types.h>
3 | #include<unistd.h>
4 |
5 | int main()
6 | {
7 |     fork();
8 |     fork();
9 |     fork();
10 |    printf("laicai\n");
11 |    return 0;
12 | }
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();
    fork();
    fork();
    printf("laicai\n");
    return 0;_
}
```



```

10     if (p < 0)
11     {
12         perror("fork fail");
13         exit(1);
14     }
15     else if (p == 0)
16         printf("Child has x = %d \n", ++x);
17     else
18         printf("Parent has x = %d\n", --x);
19
20     return 0;
21 }

```

```

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

int main()
{
    int x=1;
    pid_t p = fork();
    if(p<0)
    {
        perror("fork failed");
        exit(1);
    }
    else if(p==0)
        printf("Child has x = %d \n",++x);
    else
        printf("Parent has x=%d \n",--x);
    return 0;
}

```

"demo320.cpp" 21L, 280B

2.运行代码

```

[root@localhost ~]# g++ demo320.cpp -o demo320
[root@localhost ~]# ./demo320
Parent has x=0
Child has x = 2

```

这表明父子进程拥有独立的内存空间

思考题

- 若父进程不调用 `waitpid()`，子进程退出后会成为僵尸进程，如何避免？

方法：使用信号处理函数（SIGCHLD）

当子进程退出时，内核会向父进程发送 SIGCHLD 信号。父进程可通过捕获该信号并调用 waitpid() 回收子进程资源。代码示例：

```
1  C
2  #include <signal.h>
3  #include <sys/wait.h>
4
5  void sigchld_handler(int signo) {
6      while (waitpid(-1, NULL, WNOHANG) > 0); // 非阻塞回收所有子进程
7  }
8
9  int main() {
10     signal(SIGCHLD, sigchld_handler); // 注册SIGCHLD信号处理函数
11     pid_t pid = fork();
12     if (pid == 0) {
13         // 子进程逻辑
14         exit(0);
15     } else {
16         // 父进程逻辑（无需调用waitpid）
17         while (1); // 父进程持续运行
18     }
19     return 0;
20 }
```

- 多次 fork() 可能导致系统资源耗尽，应如何限制进程数量？

使用信号量限制并发进程数

通过信号量（Semaphore）控制同时存在的进程数量。示例代码：

```
1  C#include <stdio.h>
2  #include <unistd.h>
3  #include <semaphore.h>
4  #include <sys/wait.h>
5  #include <sys/mman.h>
6
7  #define MAX_PROCESSES 4 // 最大并发进程数
8
9  int main() {
10     sem_t *sem = sem_open("/proc_sem", O_CREAT, 0666, MAX_PROCESSES);
11     for (int i = 0; i < 10; i++) { // 模拟多次任务
12         sem_wait(sem); // 等待可用资源
13         pid_t pid = fork();
14         if (pid == 0) { // 子进程执行任务
15             printf("Child %d: laicai\n", getpid());
16             sleep(1); // 模拟任务执行
17             sem_post(sem); // 释放信号量
18             _exit(0);
19         } else if (pid < 0) {
20             sem_post(sem); // 出错时释放信号量
21             perror("fork failed");
22         }
23     }
24     sem_close(sem);
```



```
25     sem_unlink("/proc_sem");
26     while (wait(NULL) > 0); // 等待所有子进程结束
27     return 0;
28 }
```

效果：

- 通过信号量确保最多 `MAX_PROCESSES` 个进程同时运行。
- 父进程动态创建子进程，但受信号量严格限制。