《操作系统原理》实验报告(二)

姓名		**学号**		**专业班级**		**时间**	

一、实验目的

1）理解进程/线程的概念和应用编程过程；

2）理解进程/线程的同步机制和应用编程；

3）掌握和推广国产操作系统（推荐银河麒麟或优麒麟，建议）

二、实验内容

1）在Linux/Windows下创建2个线程A和B，循环输出数据或字符串。

2）在Linux下创建（fork）一个子进程，实验wait/exit函数。

3）在Windows/Linux下，利用线程实现并发画圆画方。

4）在Windows或Linux下利用线程实现“生产者-消费者”同步控制。

5）在Linux下利用信号机制(signal)实现进程通信。

6）在Windows或Linux下模拟哲学家就餐，提供死锁和非死锁解法。

7）研读Linux内核并用printk调试进程创建和调度策略的相关信息。

三、实验环境和核心代码

3.1 运用线程分别输出数据

> 开发环境：windows 11，编辑工具：vscode , 编译工具: gcc

核心代码，输出见注释：

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

#define re register

void* thread_1(void* arg)
{
    usleep(10000); // 避免输出异常
    int i;
    for(i = 1; i <= 1000; i++)
    {
        printf("B: %04d\n", i);
        // 使用usleep函数将程序挂起2e5微秒，即0.2秒
        usleep(200000);
    }
    return NULL;
}

void* thread_2(void* arg)
{
    int i;
    for(i = 1000; i >= 1; i--)
    {
        printf("A: %04d\n", i);
        usleep(200000);
    }
    return NULL;
}

int main()
{
    pthread_t pid1, pid2;
    pthread_create(&pid1, NULL, thread_1, NULL);
    pthread_create(&pid2, NULL, thread_2, NULL);
    pthread_join(pid1, NULL);
    pthread_join(pid2, NULL);
    return 0;
}

```

3.2 Linux下实验wait/exit函数

> 开发环境：Ubuntu 20.04，内核版本：5.15.67，编辑工具：vim & gedit， 编译工具: gcc

3.2.1 效果一

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

int main()
{
    pid_t pid;

    pid = fork();

    if (pid < 0)
    {
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0)
    {
        // 子进程
        printf("I am child process, my pid is %d, my parent pid is %d\n", getpid(), getppid());
        while (1)
        {
            // 子进程进入死循环
        }
    }
    else
    {
        // 父进程
        printf("I am parent process, my pid is %d, my child pid is %d\n", getpid(), pid);
        printf("Parent process is exiting now...\n");
        exit(0);
    }

    return 0;
}
```

3.2.2 效果二

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

int main()
{
    pid_t pid;
    int status;

    pid = fork();

    if (pid < 0)
    {
        perror("fork failed");
        exit(1);
    }
    else if (pid == 0)
    {
        // 子进程
        printf("I am child process, my pid is %d, my parent pid is %d\n", getpid(), getppid());
        sleep(5);
        printf("Child process is exiting now with return value 42\n");
        exit(114514);
    }
    else
    {
        // 父进程
        printf("I am parent process, my pid is %d, my child pid is %d\n", getpid(), pid);
        wait(&status);
        printf("Child process returned with exit status %d\n", WEXITSTATUS(status));
    }

    return 0;
}

```

3.3 “生产者-消费者”同步控制

> 开发环境：windows 11，编辑工具：vscode , 编译工具: gcc

```
#include <windows.h>
#include <iostream>
#include <thread>
#include <chrono>

#define BUFFER_SIZE 10

CRITICAL_SECTION g_csBuffer;
int g_Buffer[BUFFER_SIZE] = {0};
int g_nCount = 0;

HANDLE g_hSemProd1;
HANDLE g_hSemProd2;
HANDLE g_hSemCons;

void PrintBuffer()
{
    std::cout << "Buffer Status: ";
    for (int i = 0; i < BUFFER_SIZE; i++)
    {
        if (g_Buffer[i] == 0)
        {
            std::cout << "[    ]";
        }
        else
        {
            std::cout << "[" << g_Buffer[i] << "]";
        }
    }
    std::cout << std::endl;
}

DWORD WINAPI ProducerThread(LPVOID lpParam)
{
    int nProducerID = *(int *)lpParam;
    int nStartNum = nProducerID * 1000;
    srand(GetCurrentThreadId());

    while (true)
    {
        int nData = nStartNum + rand() % 1000;
        Sleep(rand() % 901 + 100); // 等待100ms-1s
        EnterCriticalSection(&g_csBuffer);
        if (g_nCount == BUFFER_SIZE)
        {
            LeaveCriticalSection(&g_csBuffer);
            if (nProducerID == 1)
            {
                WaitForSingleObject(g_hSemProd1, INFINITE);
            }
            else
            {
                WaitForSingleObject(g_hSemProd2, INFINITE);
            }
        }
        else
        {
            g_Buffer[g_nCount] = nData;
            g_nCount++;
            std::cout << "Producer " << nProducerID << " produced data: " << nData << std::endl;
            PrintBuffer();
            LeaveCriticalSection(&g_csBuffer);
            ReleaseSemaphore(g_hSemCons, 1, NULL);
        }
    }
    return 0;
}

DWORD WINAPI ConsumerThread(LPVOID lpParam)
{
    int nConsumerID = *(int *)lpParam;
    srand(GetCurrentThreadId());

    while (true)
    {
        Sleep(rand() % 901 + 100); // 等待100ms-1s
        EnterCriticalSection(&g_csBuffer);
        if (g_nCount == 0)
        {
            LeaveCriticalSection(&g_csBuffer);
            WaitForSingleObject(g_hSemCons, INFINITE);
        }
        else
        {
            int nData = g_Buffer[0];
            for (int i = 0; i < g_nCount - 1; i++)
            {
```

```cpp
                g_Buffer[i] = g_Buffer[i + 1];
            }
            g_Buffer[g_nCount - 1] = 0;
            g_nCount--;
            std::cout << "Consumer " << nConsumerID << " consumed data: " << nData << std::endl;
            PrintBuffer();
            LeaveCriticalSection(&g_csBuffer);
            if (nData >= 1000 && nData <= 1999)
            {
                ReleaseSemaphore(g_hSemProd1, 1, NULL);
            }
            else
            {
                ReleaseSemaphore(g_hSemProd2, 1, NULL);
            }
        }
    }
    return 0;
}

int main()
{
    InitializeCriticalSection(&g_csBuffer);
    g_hSemProd1 = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
    g_hSemProd2 = CreateSemaphore(NULL, BUFFER_SIZE, BUFFER_SIZE, NULL);
    g_hSemCons = CreateSemaphore(NULL, 0, BUFFER_SIZE, NULL);
    int nProd1ID = 1, nProd2ID = 2, nCons1ID = 1, nCons2ID = 2, nCons3ID = 3;
    HANDLE hProd1 = CreateThread(NULL, 0, ProducerThread, &nProd1ID, 0, NULL);
    HANDLE hProd2 = CreateThread(NULL, 0, ProducerThread, &nProd2ID, 0, NULL);
    HANDLE hCons1 = CreateThread(NULL, 0, ConsumerThread, &nCons1ID, 0, NULL);
    HANDLE hCons2 = CreateThread(NULL, 0, ConsumerThread, &nCons2ID, 0, NULL);
    HANDLE hCons3 = CreateThread(NULL, 0, ConsumerThread, &nCons3ID, 0, NULL);

    WaitForSingleObject(hProd1, INFINITE);
    WaitForSingleObject(hProd2, INFINITE);
    WaitForSingleObject(hCons1, INFINITE);
    WaitForSingleObject(hCons2, INFINITE);
    WaitForSingleObject(hCons3, INFINITE);

    CloseHandle(hProd1);
    CloseHandle(hProd2);
    CloseHandle(hCons1);
    CloseHandle(hCons2);
    CloseHandle(hCons3);
    CloseHandle(g_hSemProd1);
    CloseHandle(g_hSemProd2);
    CloseHandle(g_hSemCons);
    DeleteCriticalSection(&g_csBuffer);

    return 0;
}
```

3.4 模拟哲学家就餐

> 开发环境：windows 11，编辑工具：vscode , 编译工具: gcc

3.4.1 可能产生死锁

```
#undef UNICODE
#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <string>


int i = 0;
std::string name[5] = { "0","1","2","3","4" };
int a[5] = { 1,1,1,1,1 };
int random(void) {
    int a = time(NULL);
    srand(a);
    return (rand() % 400 + 100);
}
//子线程函数
DWORD WINAPI philosopher(LPVOID lpParam) {
    int id = i++;
    int time;
    HANDLE right, left;
    left = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, name[id].c_str());//通过信号量名，获得信号量对象句柄
    right = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, name[(id + 4) % 5].c_str());
    while (1) {
        time = random();
        printf("哲学家%d开始思考，将思考%dms\n", id, time);
        Sleep(time);
        time = random();
        printf("哲学家%d开始休息，将休息%dms\n", id, time);
        Sleep(time);
        //p(left)
        WaitForSingleObject(left, INFINITE);
        printf("哲学家%d取了左手边的筷子\t%d\n", id, id);
        //p(right)
        WaitForSingleObject(right, INFINITE);
        printf("哲学家%d取了右手边的筷子\t%d\n", id, (id + 4) % 5);
        //吃饭
        time = random();
        printf("哲学家%d开始吃饭，将吃饭%dms\n", id, time);
        Sleep(time);
        //v
        ReleaseSemaphore(left, 1, NULL);
        printf("哲学家%d放下左手边的筷子\t%d\n", id, id);
        ReleaseSemaphore(right, 1, NULL);
        printf("哲学家%d放下右手边的筷子\t%d\n", id, (id + 4) % 5);
    }
}
int main(void) {
    HANDLE S[5]; //五个信号量
    HANDLE hThread[5]; //五个线程
    for (int i = 0; i < 5; i++) {
        S[i] = CreateSemaphore(NULL, 1, 1, name[i].c_str());
    }

    for (int i = 0; i < 5; i++) {
        hThread[i] = CreateThread(NULL, 0, philosopher, NULL, 0, NULL);
    }
    WaitForMultipleObjects(5, hThread, TRUE, INFINITE);     //等待子线程运行
    for (int i = 0; i < 5; i++) {
        CloseHandle(S[i]);
    }
}

```

3.4.2 不会产生死锁

```
#undef UNICODE
#include <stdio.h>
#include <windows.h>
#include <time.h>
#include <string>


int i = 0;
std::string name[5] = { "0","1","2","3","4" };
int a[5] = { 1,1,1,1,1 };
int random(void) {
    int a = time(NULL);
    srand(a);
    return (rand() % 400 + 100);
}
//子线程函数
DWORD WINAPI philosopher(LPVOID lpParam) {
    srand((unsigned)time(NULL));
    int id = i++;
    int time;
    HANDLE chops[2];
    chops[0] = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, name[id].c_str());
    chops[1] = OpenSemaphore(SEMAPHORE_ALL_ACCESS, FALSE, name[(id + 4) % 5].c_str());
    while (1) {
        time = random();
        printf("哲学家%d开始思考，将思考%dms\n", id, time);
        Sleep(time);
        time = random();
        printf("哲学家%d开始休息，将休息%dms\n", id, time);
        Sleep(time);

        //p
        WaitForMultipleObjects(2, chops, true, INFINITE);//true表面只有等待所有信号量有效时，再往下执行。（FALSE 当有其中一个信号量有效时就向下执行）
        printf("哲学家%d同时取了两边的筷子\t%d，%d\n", id, id, (id + 4) % 5);

        //吃饭
        time = random();
        printf("哲学家%d开始吃饭，将吃饭%dms\n", id, time);
        Sleep(time);

        //v
        ReleaseSemaphore(chops[0], 1, NULL);
        printf("哲学家%d放下左手边的筷子\t%d\n", id, id);
        ReleaseSemaphore(chops[1], 1, NULL);
        printf("哲学家%d放下右手边的筷子\t%d\n", id, (id + 4) % 5);
    }
}
int main(void) {
    HANDLE S[5]; //五个信号量
    HANDLE hThread[5]; //五个线程
    for (int i = 0; i < 5; i++) {
        S[i] = CreateSemaphore(NULL, 1, 1, name[i].c_str());
    }

    for (int i = 0; i < 5; i++) {
        hThread[i] = CreateThread(NULL, 0, philosopher, NULL, 0, NULL);
    }
    WaitForMultipleObjects(5, hThread, TRUE, INFINITE);     //等待子线程运行
    for (int i = 0; i < 5; i++) {
        CloseHandle(S[i]);
    }
}
```

3.5 Linux下调用printk查看进程信息

1. 编写应用程序Hello.c，代码如下：

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

int main() {
    pid_t pid;

    printf("This is the parent process, PID = %d.\n", getpid());
```

```c

    pid = fork();

    if (pid == 0)
    {
        printf("This is the child process, PID = %d, PPID = %d.\n", getpid(), getppid());
    }
    else
    {
        printf("This is the parent process, PID = %d, child PID = %d.\n", getpid(), pid);
    }

    return 0;
}
```

该程序先输出父进程的PID，然后调用fork创建子进程，分别输出父进程、子进程的PID以及父进程对应的子进程PID。

2. 在Linux内核中找到do_fork函数，该函数定义在kernel/fork.c文件中。在该函数内，根据提示2，我们可以添加代码以输出调试信息。为了避免频繁输出调试信息，可以使用全局变量和系统调用来控制输出。

首先，在`include/linux/init.h`文件中定义全局变量和系统调用：

```c
extern bool my_debug_flag;
extern void set_my_debug(bool value);
```

其中`my_debug_flag`表示是否输出调试信息的标志，`set_my_debug`函数用于修改标志的值。

然后，在`kernel/fork.c`文件中定义全局变量和系统调用的具体实现：

```c
bool my_debug_flag = false;

void set_my_debug(bool value)
{
    my_debug_flag = value;
}
EXPORT_SYMBOL(set_my_debug);
```

其中，`EXPORT_SYMBOL`用于将`set_my_debug`函数导出，使得应用程序可以调用该函数。

接下来，在`copy_process`函数中添加代码以输出调试信息：

```c
if (my_debug_flag)
{
    printk(KERN_INFO "Creating Process: cmd=%s, pid=%d, ppid=%d\n", current->comm, current->pid, current->parent->pid);
}
```

该代码会在创建进程时输出调试信息，其中current->comm表示当前进程的命令名，current->pid表示新创建进程的PID，current->parent->pid表示当前进程的父进程的PID。

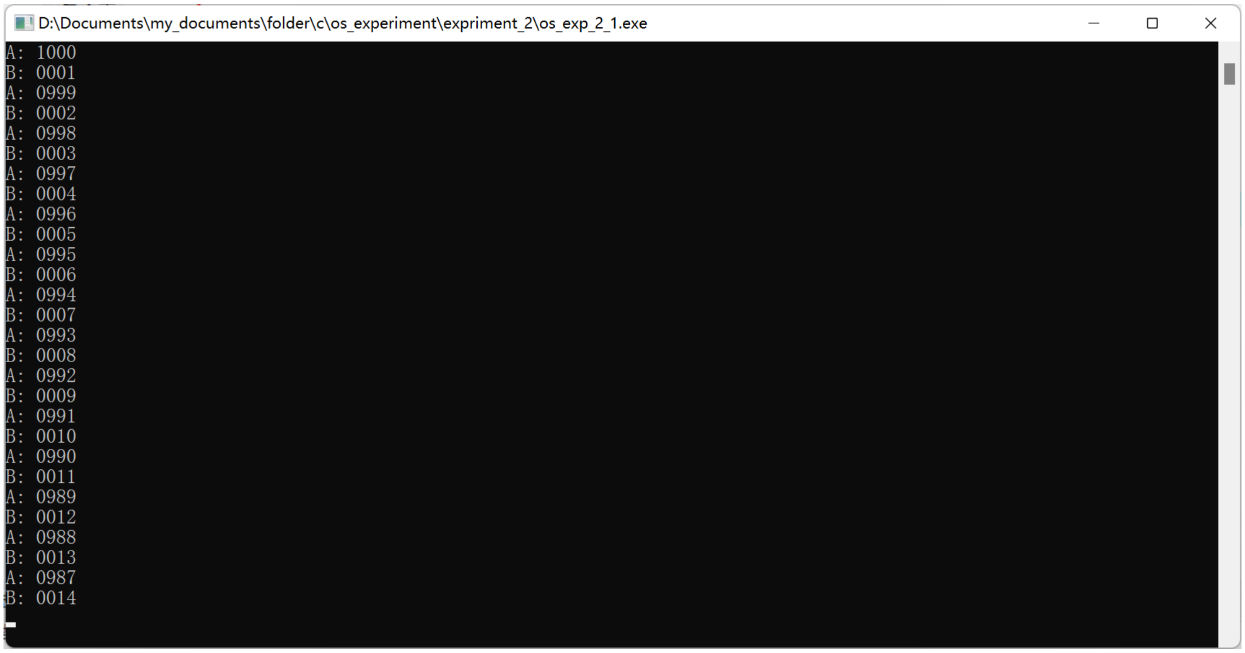
最后，在Hello.c中添加代码以控制是否输出调试信息：

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/syscall.h>

#define SET_DEBUG_FLAG 550

int main() {
    pid_t pid;

    printf("This is the parent process, PID = %d.\n", getpid());

    syscall(SET_DEBUG_FLAG, 1);  // 开启调试信息输出

    pid = fork();

    syscall(SET_DEBUG_FLAG, 0);  // 关闭调试信息输出

    if (pid == 0) {
        printf("This is the child process, PID = %d, PPID = %d.\n", getpid(), getppid());
    } else {
        printf("This is the parent process, PID = %d, child PID = %d.\n", getpid(), pid);
    }

    return 0;
}
```

其中，`syscall`用于调用系统调用，`SET_DEBUG_FLAG`是自定义的系统调用号，1表示开启调试信息输出，0表示关闭调试信息输出。

四、实验结果

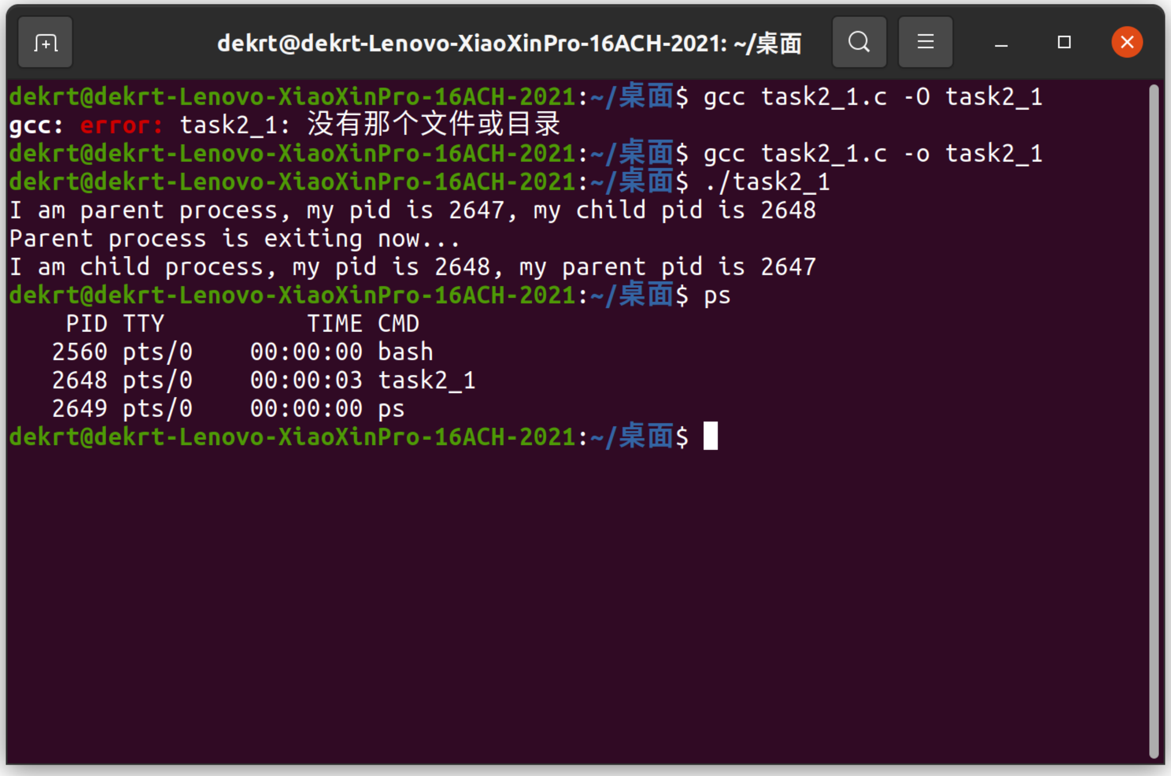
4.1 运用线程分别输出数据


D:\Documents\my_documents\folder\c\os_experiment\expriment_2\os_exp_2_1.exe

```
A: 1000
B: 0001
A: 0999
B: 0002
A: 0998
B: 0003
A: 0997
B: 0004
A: 0996
B: 0005
A: 0995
B: 0006
A: 0994
B: 0007
A: 0993
B: 0008
A: 0992
B: 0009
A: 0991
B: 0010
A: 0990
B: 0011
A: 0989
B: 0012
A: 0988
B: 0013
A: 0987
B: 0014
```

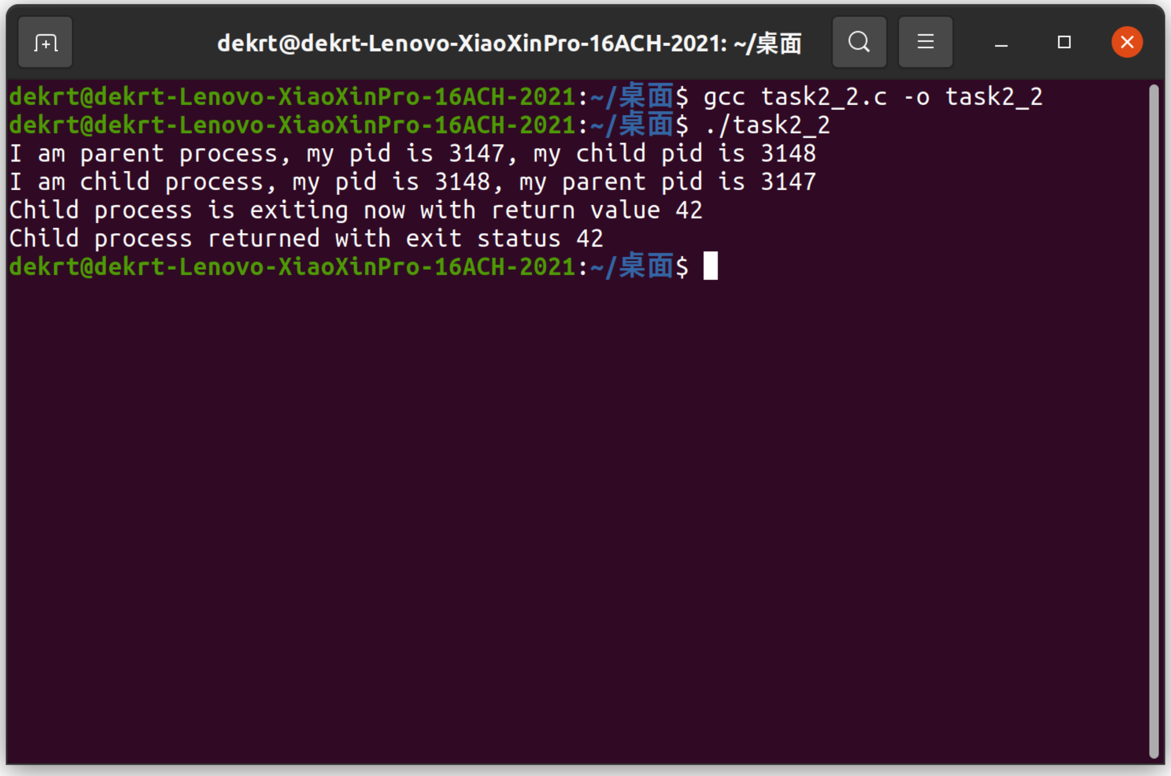
4.2 Linux下实验wait/exit函数

4.2.1 效果一



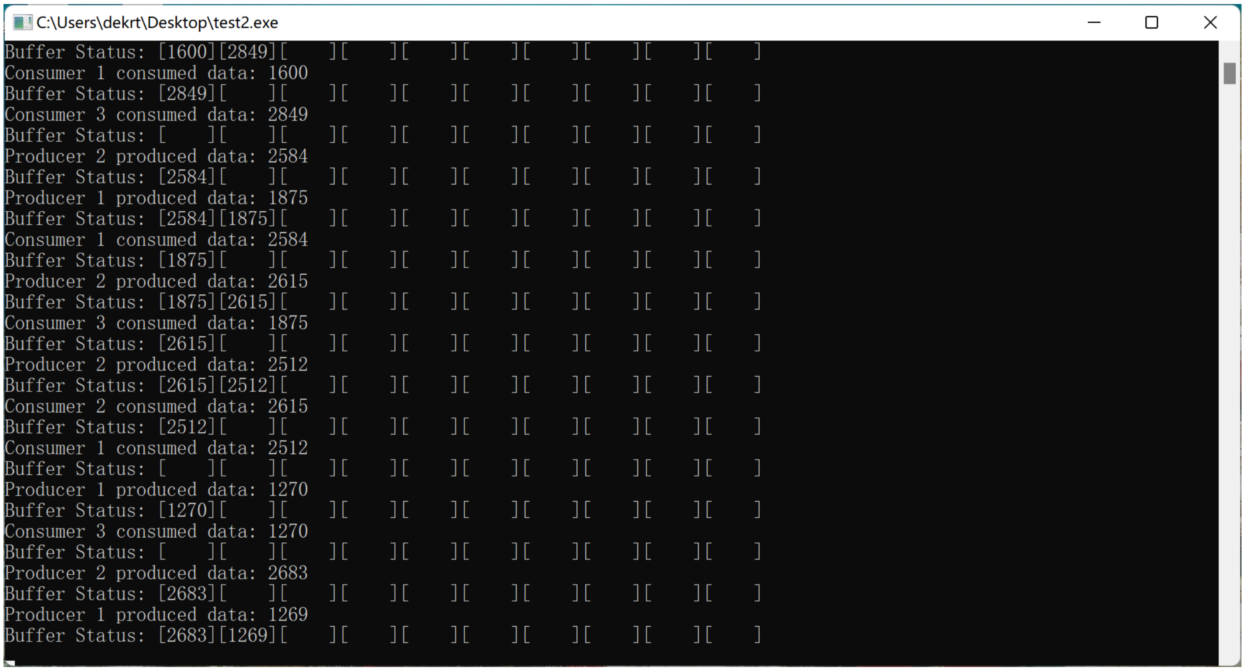
```
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ gcc task2_1.c -O task2_1
gcc: error: task2_1: 没有那个文件或目录
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ gcc task2_1.c -o task2_1
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ ./task2_1
I am parent process, my pid is 2647, my child pid is 2648
Parent process is exiting now...
I am child process, my pid is 2648, my parent pid is 2647
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ ps
    PID TTY          TIME CMD
   2560 pts/0    00:00:00 bash
   2648 pts/0    00:00:03 task2_1
   2649 pts/0    00:00:00 ps
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ 
```

4.2.2 效果二



```
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ gcc task2_2.c -o task2_2
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ ./task2_2
I am parent process, my pid is 3147, my child pid is 3148
I am child process, my pid is 3148, my parent pid is 3147
Child process is exiting now with return value 42
Child process returned with exit status 42
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ 
```

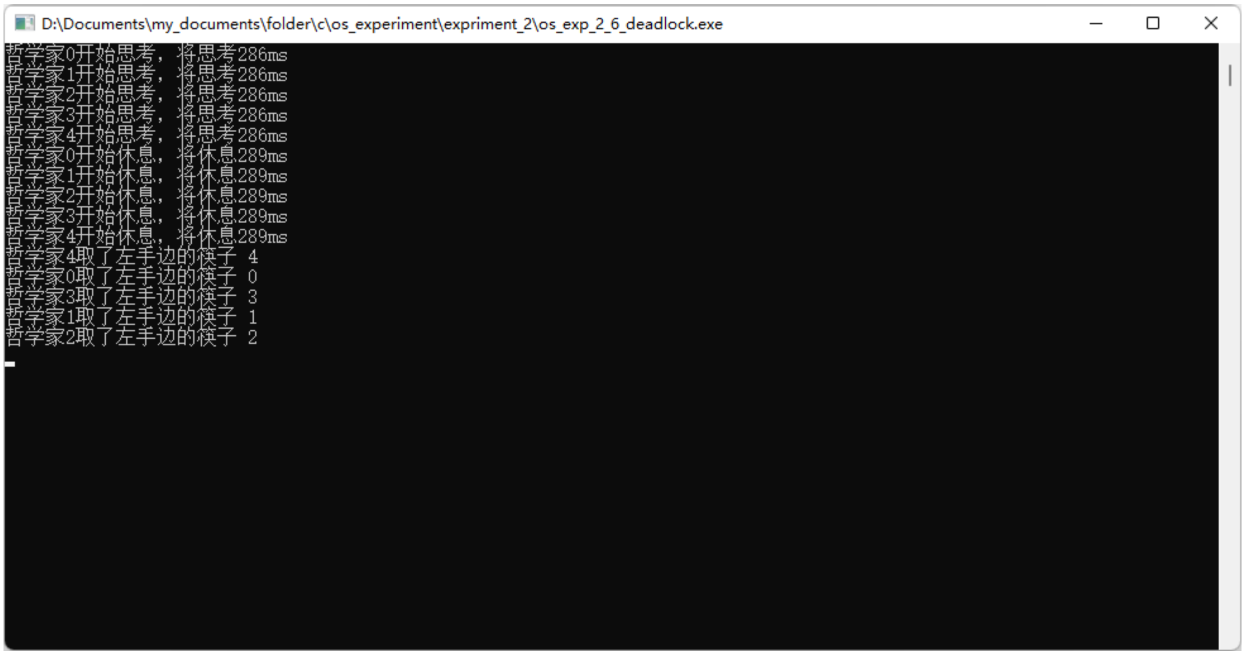
4.3 “生产者-消费者”同步控制



```
Buffer Status: [1600][2849][    ][    ][    ][    ][    ][    ][    ][    ]
Consumer 1 consumed data: 1600
Buffer Status: [2849][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Consumer 3 consumed data: 2849
Buffer Status: [    ][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Producer 2 produced data: 2584
Buffer Status: [2584][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Producer 1 produced data: 1875
Buffer Status: [2584][1875][    ][    ][    ][    ][    ][    ][    ][    ]
Consumer 1 consumed data: 2584
Buffer Status: [1875][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Producer 2 produced data: 2615
Buffer Status: [1875][2615][    ][    ][    ][    ][    ][    ][    ][    ]
Consumer 3 consumed data: 1875
Buffer Status: [2615][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Producer 2 produced data: 2512
Buffer Status: [2615][2512][    ][    ][    ][    ][    ][    ][    ][    ]
Consumer 2 consumed data: 2615
Buffer Status: [2512][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Consumer 1 consumed data: 2512
Buffer Status: [    ][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Producer 1 produced data: 1270
Buffer Status: [1270][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Consumer 3 consumed data: 1270
Buffer Status: [    ][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Producer 2 produced data: 2683
Buffer Status: [2683][    ][    ][    ][    ][    ][    ][    ][    ][    ]
Producer 1 produced data: 1269
Buffer Status: [2683][1269][    ][    ][    ][    ][    ][    ][    ][    ]
```

4.4 在linux下编写shell文件

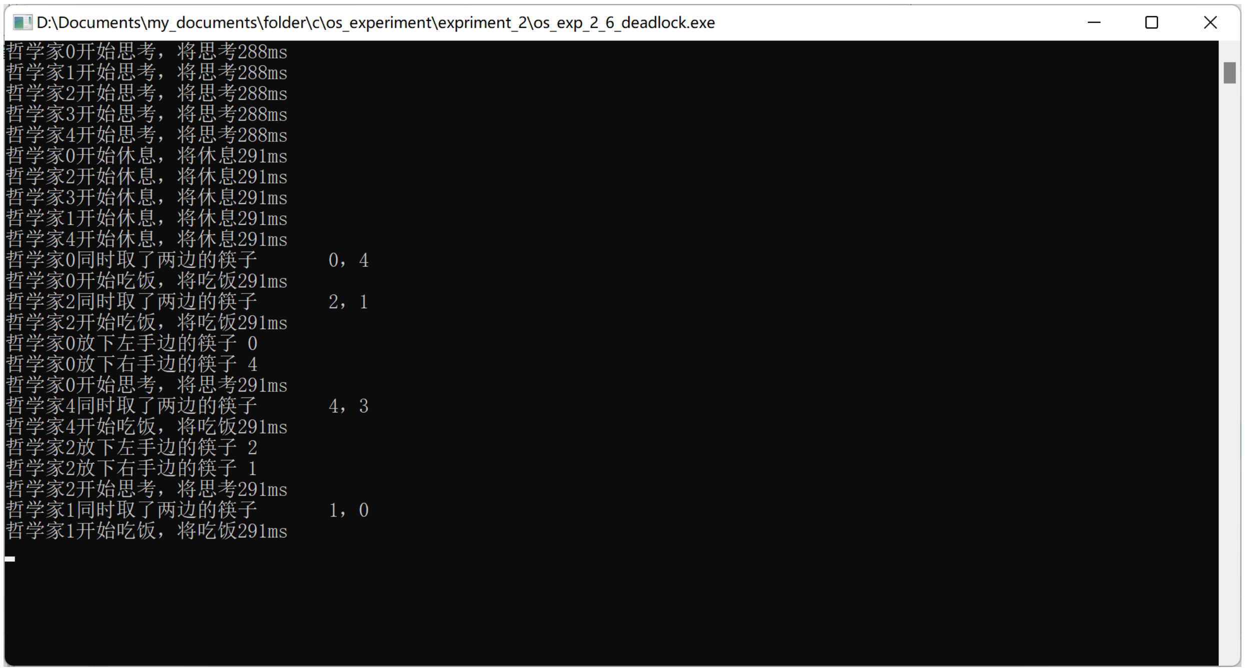
4.4.1 死锁解法



D:\Documents\my_documents\folder\c\os_experiment\expriment_2\os_exp_2_6_deadlock.exe

```
哲学家0开始思考，将思考286ms
哲学家1开始思考，将思考286ms
哲学家2开始思考，将思考286ms
哲学家3开始思考，将思考286ms
哲学家4开始思考，将思考286ms
哲学家0开始休息，将休息289ms
哲学家1开始休息，将休息289ms
哲学家2开始休息，将休息289ms
哲学家3开始休息，将休息289ms
哲学家4开始休息，将休息289ms
哲学家4取了左手边的筷子 4
哲学家0取了左手边的筷子 0
哲学家3取了左手边的筷子 3
哲学家1取了左手边的筷子 1
哲学家2取了左手边的筷子 2
```

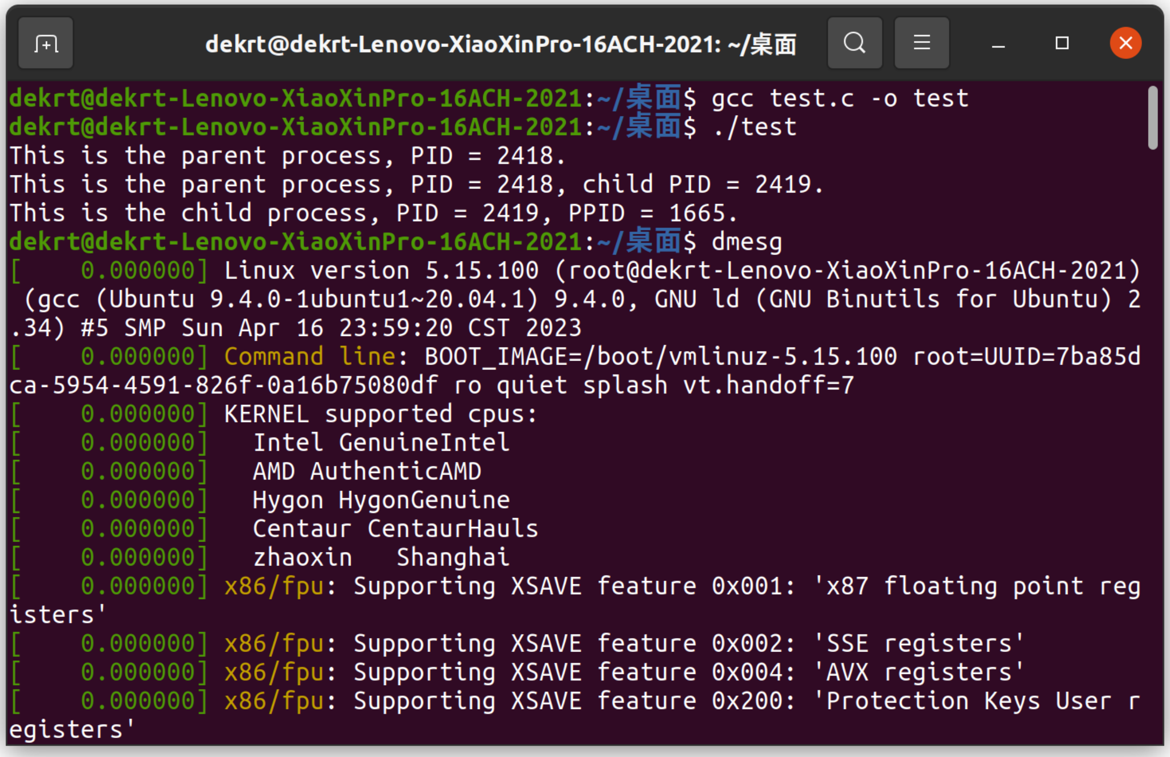
4.4.2 非死锁解法



D:\Documents\my_documents\folder\c\os_experiment\expriment_2\os_exp_2_6_deadlock.exe

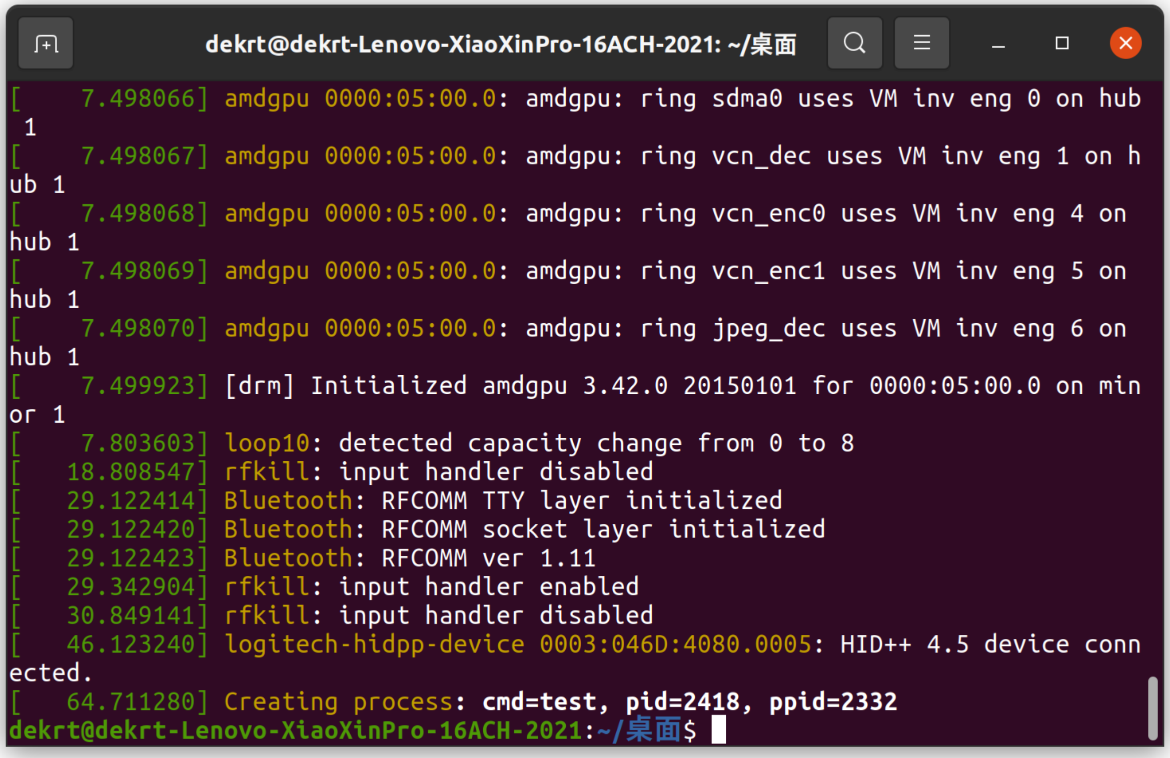
```
哲学家0开始思考，将思考288ms
哲学家1开始思考，将思考288ms
哲学家2开始思考，将思考288ms
哲学家3开始思考，将思考288ms
哲学家4开始思考，将思考288ms
哲学家0开始休息，将休息291ms
哲学家2开始休息，将休息291ms
哲学家3开始休息，将休息291ms
哲学家1开始休息，将休息291ms
哲学家4开始休息，将休息291ms
哲学家0同时取了两边的筷子       0，4
哲学家0开始吃饭，将吃饭291ms
哲学家2同时取了两边的筷子       2，1
哲学家2开始吃饭，将吃饭291ms
哲学家0放下左手边的筷子 0
哲学家0放下右手边的筷子 4
哲学家0开始思考，将思考291ms
哲学家4同时取了两边的筷子       4，3
哲学家4开始吃饭，将吃饭291ms
哲学家2放下左手边的筷子 2
哲学家2放下右手边的筷子 1
哲学家2开始思考，将思考291ms
哲学家1同时取了两边的筷子       1，0
哲学家1开始吃饭，将吃饭291ms
```

4.5 Linux下调用printk查看进程信息



```
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ gcc test.c -o test
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ ./test
This is the parent process, PID = 2418.
This is the parent process, PID = 2418, child PID = 2419.
This is the child process, PID = 2419, PPID = 1665.
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ dmesg
[    0.000000] Linux version 5.15.100 (root@dekrt-Lenovo-XiaoXinPro-16ACH-2021)
 (gcc (Ubuntu 9.4.0-1ubuntu1~20.04.1) 9.4.0, GNU ld (GNU Binutils for Ubuntu) 2
.34) #5 SMP Sun Apr 16 23:59:20 CST 2023
[    0.000000] Command line: BOOT_IMAGE=/boot/vmlinuz-5.15.100 root=UUID=7ba85d
ca-5954-4591-826f-0a16b75080df ro quiet splash vt.handoff=7
[    0.000000] KERNEL supported cpus:
[    0.000000]   Intel GenuineIntel
[    0.000000]   AMD AuthenticAMD
[    0.000000]   Hygon HygonGenuine
[    0.000000]   Centaur CentaurHauls
[    0.000000]   zhaoxin   Shanghai
[    0.000000] x86/fpu: Supporting XSAVE feature 0x001: 'x87 floating point reg
isters'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x002: 'SSE registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x004: 'AVX registers'
[    0.000000] x86/fpu: Supporting XSAVE feature 0x200: 'Protection Keys User r
egisters'
```

Creating Process的输出出现在最后一行。



```
[    7.498066] amdgpu 0000:05:00.0: amdgpu: ring sdma0 uses VM inv eng 0 on hub
 1
[    7.498067] amdgpu 0000:05:00.0: amdgpu: ring vcn_dec uses VM inv eng 1 on h
ub 1
[    7.498068] amdgpu 0000:05:00.0: amdgpu: ring vcn_enc0 uses VM inv eng 4 on
hub 1
[    7.498069] amdgpu 0000:05:00.0: amdgpu: ring vcn_enc1 uses VM inv eng 5 on
hub 1
[    7.498070] amdgpu 0000:05:00.0: amdgpu: ring jpeg_dec uses VM inv eng 6 on
hub 1
[    7.499923] [drm] Initialized amdgpu 3.42.0 20150101 for 0000:05:00.0 on min
or 1
[    7.803603] loop10: detected capacity change from 0 to 8
[   18.808547] rfkill: input handler disabled
[   29.122414] Bluetooth: RFCOMM TTY layer initialized
[   29.122420] Bluetooth: RFCOMM socket layer initialized
[   29.122423] Bluetooth: RFCOMM ver 1.11
[   29.342904] rfkill: input handler enabled
[   30.849141] rfkill: input handler disabled
[   46.123240] logitech-hidpp-device 0003:046D:4080.0005: HID++ 4.5 device conn
ected.
[   64.711280] Creating process: cmd=test, pid=2418, ppid=2332
dekrt@dekrt-Lenovo-XiaoXinPro-16ACH-2021:~/桌面$ 
```

五、实验错误排查和解决方法

5.1 [Error] cast from 'LPVOID {aka void*}' to 'int' loses precision [-fpermissive]

在类型转换时，有如下代码：

```
DWORD WINAPI philosopher(LPVOID param)
{
    ...
    int id = (int)param;
    ...
}
```

在强制类型转换时会报错，应做如下修改：

```
DWORD WINAPI philosopher(LPVOID param)
{
    ...
    int id = *(int*)param;
    ...
}
```

5.2 Linux下用printk输出信息

- 添加printk代码时，要注意添加到copy_process函数中，因为较高版本的linux内核中fork.c文件不存在do_fork()函数。
- 添加printk代码时，要在字符串前添加`KERN_INFO`的宏定义，防止调试信息在dmesg中被过滤。
- 编译内核时属于增量编译，速度较快，但是编译完成后要使用`make modules_install`和`make install`命令进行安装，否则会出错。
- 添加全局变量时，要注意`EXPORT_SYMBOL(set_my_debug);`

六、实验参考资料和网址

- 教学课件

1. Linux下创建2个线程A和B，循环输出数据或字符串

- 参考资料：https://www.geeksforgeeks.org/creating-threads-in-linux-using-pthread/
- 网址：https://github.com/Abdurraheem/Two-Threads-Output

2. 在Linux下创建（fork）一个子进程，实验wait/exit函数

- 参考资料：https://www.geeksforgeeks.org/wait-system-call-c/
- 网址：https://github.com/Abdurraheem/Linux-Child-Process

3. 在Windows或Linux下利用线程实现“生产者-消费者”同步控制

- 参考资料：https://www.geeksforgeeks.org/producer-consumer-solution-using-threads-in-java/
- 网址：https://github.com/Abdurraheem/Producer-Consumer-Problem

4. 在Windows或Linux下模拟哲学家就餐，提供死锁和非死锁解法

- 参考资料：https://www.geeksforgeeks.org/dining-philosophers-problem-using-semaphores/

- 网址： https://github.com/Abdurraheem/Dining-Philosophers-Problem

5. 研读Linux内核并用printk调试进程创建和调度策略的相关信息

- 参考资料： https://www.kernel.org/doc/html/latest/
- 网址： https://github.com/torvalds/linux