

操作系统上机

实 验 报 告

成绩

教 师：

2023 年 月 日

班 级：操作系统 3 班

学 号：21009200574

姓 名：赵真卿

实验地点：E-203

实验时间：2023. 3. 17—2023. 5. 25

实验一：进程的建立

实验目的

- 创建进程及子进程；
- 在父子进程间实现通信

实验软硬件环境

- Linux 操作系统

实验内容

- 创建进程并显示标识等进程控制块的属性信息；
- 显示父子进程的通信信息和相应的应答信息（进程间通信机制任选）

实验要求

- 显示创建的进程和控制参数；
- 显示进程间关系参数。

实验设计与实现

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

int main() {
    int n=100;
    pid_t pid;

    pid = fork(); // 创建子进程

    if (pid < 0) {
        printf("进程创建失败\n");
        return 1;
    } else if (pid == 0) {
        // 子进程代码
        printf("子进程开始执行,子进程 pid=%hd,父进程 pid=%hd\n",getpid(),getppid());
        // 执行子进程任务
        n++;
        printf("n=%d\n",n);
        printf("子进程执行完毕\n");
        exit(0);
    } else {
        // 父进程代码
        printf("父进程等待子进程执行完毕\n");
        wait(NULL); // 等待子进程结束
        printf("父进程继续执行,父进程 pid=%hd,子进程 pid=%hd\n",getpid(),pid);
        printf("n=%d\n",n);
    }

    return 0;
}
```

在 Linux 操作系统中，可以使用 `fork()` 函数来复制一个当前进程，并作为当前进程的子进程。`fork()` 函数在头文件中，没有参数，返回值为 `pid_t` 类型。在父进程中，该函数返回值是子进程的 `pid`，而在子进程中，该函数返回值始终为 0，可以通过返回值的特点来区别父子进程。进程创建失败时，返回值为负数。

`fork` 出的子进程会复制父进程的资源（实际上用的是一种类似“懒标记”法的方法，子进程会继承父进程的资源，但是当子进程修改资源的值时，会复制一份资源，不会对父进程修改），并继续执行剩余代码。

实验结果分析

```
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/1$ gcc 1.c -o 1
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/1$ ./1
父进程等待子进程执行完毕
子进程开始执行,子进程pid=10532,父进程pid=10531
n=101
子进程执行完毕
子进程已经结束,返回值为 0
父进程继续执行,父进程pid=10531,子进程pid=10532
n=100
```

可以看到，父子进程可以互相显示进程控制块中的信息，且子进程中修改值并不会影响父进程。

实验二：线程共享进程数据

实验目的

- 了解线程与进程之间的数据共享关系。创建一个线程，在线程中更改进程中的数据。

实验软硬件环境

- Linux 操作系统

实验内容

- 在进程中定义全局共享数据，在线程中直接引用该数据进行更改 并输出该数据。

实验要求

- 显示和输出共享数据。

实验设计与实现

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

int sharedData = 0; // 全局共享数据

void* threadFunc(void* arg) {
    // 在线程中更改共享数据
    sharedData = 100;

    // 输出共享数据
    printf("线程中的共享数据: %d\n", sharedData);

    pthread_exit(NULL);
}

int main() {
    pthread_t thread;
    int ret;

    // 创建线程
    ret = pthread_create(&thread, NULL, threadFunc, NULL);
    if (ret != 0) {
        printf(stderr, "无法创建线程\n");
        return 1;
    }

    // 等待线程结束
    ret = pthread_join(thread, NULL);
    if (ret != 0) {
        printf(stderr, "无法等待线程\n");
        return 1;
    }

    // 输出共享数据
    printf("进程中的共享数据: %d\n", sharedData);

    return 0;
}
```

在这个示例中，定义了一个名为 **sharedData** 的全局变量作为进程中的共享数据。然后，创

建了一个线程 **thread**，线程函数为 **threadFunc**。在线程函数 **threadFunc** 中，我们将共享数据 **sharedData** 更改为 100，并输出该数据。在 **main** 函数中，使用 **pthread_create** 函数创建线程，并使用 **pthread_join** 函数等待线程结束。最后，在主线程中输出共享数据 **sharedData**。

首先，在全局范围内定义了一个名为 **sharedData** 的整数变量，它被用作进程中的全局共享数据。

```
int sharedData = 0; // 全局共享数据
```

接下来，定义了一个名为 **threadFunc** 的函数，它作为线程的入口点。在这个函数中，将共享数据 **sharedData** 的值修改为 100，并在控制台上输出该数据。

```
void* threadFunc(void* arg) {
    // 在线程中更改共享数据
    sharedData = 100;

    // 输出共享数据
    printf("线程中的共享数据: %d\n", sharedData);

    pthread_exit(NULL);
}
```

然后，在主函数 **main** 中，声明了一个 **pthread_t** 类型的变量 **thread**，用于表示线程的标识符。还定义了一个整数变量 **ret**，用于保存函数调用的返回值。

```
pthread_t thread;
int ret;
```

接下来，使用 **pthread_create** 函数创建一个新线程，并传递给它线程函数 **threadFunc** 作为入口点。如果成功创建线程，**pthread_create** 函数将返回 0，否则返回一个非零值。

```
ret = pthread_create(&thread, NULL, threadFunc, NULL);
if (ret != 0) {
    printf(stderr, "无法创建线程\n");
    return 1;
}
```

在创建线程后，使用 **pthread_join** 函数等待线程的结束。这样做是为了确保主线程在子线程完成后才继续执行。**pthread_join** 函数将阻塞主线程，直到指定的线程终止。

```
ret = pthread_join(thread, NULL);
if (ret != 0) {
    fprintf(stderr, "无法等待线程\n");
    return 1;
}
```

最后，在主线程中输出共享数据 **sharedData** 的值。

```
printf("进程中的共享数据: %d\n", sharedData);
```

当运行这段代码时，它将创建一个新线程，在线程中将共享数据更改为 100，并在控制台输出线程中的共享数据。然后，主线程将继续执行，并在控制台输出进程中的共享数据。

实验结果分析

A terminal window with a dark background and light green text. The window title is 'zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/2'. The prompt is 'zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/2\$'. The user enters 'gcc 2.c -o 2', followed by './2'. The output shows '线程中的共享数据: 100' and '进程中的共享数据: 100'. The prompt returns to 'zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/2\$'.

```
zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/2$ gcc 2.c -o 2
zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/2$ ./2
线程中的共享数据: 100
进程中的共享数据: 100
zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/2$
```

可以看到，子线程修改了主进程中的数据，并将结果输出。

实验三：信号通信

实验目的

- 利用信号通信机制在父子进程及兄弟进程间进行通信。

实验软硬件环境

- Linux 操作系统

实验内容

- 父进程创建一个有名事件，由子进程发送事件信号，父进程获取 事件信号后进行相应的处理。

实验要求

- 显示控制参数；

实验设计与实现

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

void signalHandler(int signum) {
    printf("接收到信号 %d\n", signum);
}

int main() {
    pid_t pid;

    // 创建有名事件（信号）
    if (signal(SIGUSR1, signalHandler) == SIG_ERR) {
        fprintf(stderr, "无法注册信号处理程序\n");
        return 1;
    }

    // 创建子进程
    pid = fork();

    if (pid < 0) {
        fprintf(stderr, "无法创建子进程\n");
        return 1;
    } else if (pid == 0) {
        // 子进程发送信号给父进程
        kill(getppid(), SIGUSR1);
        exit(0);
    } else {
        // 父进程等待信号
        pause();
        printf("父进程收到信号，处理完成\n");
    }

    return 0;
}
```

在这段代码中，父进程创建了一个有名事件（信号）**SIGUSR1**，使用 **signal** 函数注册了一个信号处理程序 **signalHandler**。**signalHandler** 函数中在接收到信号后进行处理。然后，父进程通过 **fork** 函数创建了一个子进程。子进程使用 **kill** 函数向父进程发送 **SIGUSR1** 信号。父进程使用 **pause** 函数等待信号的到来。当收到信号时，将触发信号处理程序

signalHandler，并输出接收到的信号编号。

当执行这段代码时，父进程将创建一个有名事件（信号）SIGUSR1，并注册一个信号处理程序 **signalHandler**。

```
if (signal(SIGUSR1, signalHandler) == SIG_ERR) {  
    fprintf(stderr, "无法注册信号处理程序\n");  
    return 1;  
}
```

接下来，父进程使用 **fork** 函数创建一个子进程。如果 **fork** 函数返回负值，则表示创建子进程失败，程序将打印错误消息并退出。如果返回值为 0，则表示当前代码正在子进程中执行，子进程使用 **kill** 函数向父进程发送 SIGUSR1 信号，然后通过 **exit(0)**退出子进程。

```
pid = fork();  
  
if (pid < 0) {  
    fprintf(stderr, "无法创建子进程\n");  
    return 1;  
} else if (pid == 0) {  
    // 子进程发送信号给父进程  
    kill(getppid(), SIGUSR1);  
    exit(0);  
}
```

在父进程中，通过调用 **pause** 函数等待信号的到来。**pause** 函数会使进程挂起，直到收到一个信号才会返回。当父进程收到 SIGUSR1 信号时，信号处理程序 **signalHandler** 将被调用。

```
pause();  
printf("父进程收到信号，处理完成\n");
```

信号处理程序 **signalHandler** 将输出接收到的信号编号。

```
void signalHandler(int signum) {  
    printf("接收到信号 %d\n", signum);  
}
```

实验结果分析



```
zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/3
zzq@zzq-VMware-Virtual-Platform:~$ cd 桌面/os/3
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/3$ gedit 3.c
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/3$ gcc 3.c -o 3
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/3$ ./3
接收到信号 10
父进程收到信号，处理完成
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/3$
```

可见父进程收到了由子进程传来的信号

实验四：匿名管道通信

实验目的

- 学习使用匿名管道在两个进程间建立通信。

实验软硬件环境

- Linux 操作系统

实验内容

- 分别建立名为 **Parent** 的单文档应用程序和 **Child** 的单文档应用程序作为父子进程；
- 由父进程创建一个匿名管道，实现父子进程向匿名管道写入和读取数据。

实验要求

- 显示父子进程的通信过程；

实验设计与实现

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

int main() {
    int pipefd[2];
    pid_t pid;
    char buffer[50];
    int bytesRead;

    // 创建匿名管道
    if (pipe(pipefd) == -1) {
        fprintf(stderr, "无法创建管道\n");
        return 1;
    }

    // 创建子进程
    pid = fork();
```

```

if (pid < 0) {
    fprintf(stderr, "无法创建子进程\n");
    return 1;
} else if (pid == 0) {
    // 子进程关闭写入端
    close(pipefd[1]);

    // 从管道中读取数据
    bytesRead = read(pipefd[0], buffer, sizeof(buffer));
    printf("子进程收到消息: %s\n", buffer);

    // 关闭读取端
    close(pipefd[0]);
    exit(0);
} else {
    // 父进程关闭读取端
    close(pipefd[0]);

    // 向管道中写入数据
    write(pipefd[1], "Hello, child!", 14);

    // 关闭写入端
    close(pipefd[1]);
}

return 0;
}

```

在这段代码中，父进程创建了一个匿名管道，通过 **pipe** 函数创建了一个包含两个文件描述符的数组 **pipefd**。其中，**pipefd[0]**用于从管道读取数据，**pipefd[1]**用于向管道写入数据。然后，父进程使用 **fork** 函数创建了一个子进程。如果 **fork** 函数返回负值，则表示创建子进程失败，程序将打印错误消息并退出。如果返回值为 0，则表示当前代码正在子进程中执行。在子进程中，子进程关闭了写入端（**pipefd[1]**），然后使用 **read** 函数从管道中读取数据，并将读取到的数据打印出来。最后，子进程关闭了读取端（**pipefd[0]**）并通过 **exit(0)** 退出子进程。

在父进程中，父进程关闭了读取端（**pipefd[0]**），然后使用 **write** 函数向管道中写入数据。写入的数据是"Hello, child!"。最后，父进程关闭了写入端（**pipefd[1]**）。

当运行这段代码时，父进程向管道中写入数据，子进程从管道中读取数据，并将读取到的数据打印出来。

当执行这段代码时，父进程将创建一个匿名管道，使用 **pipe** 函数创建了一个包含两个文件描述符的数组 **pipefd**。

```

if (pipe(pipefd) == -1) {
    fprintf(stderr, "无法创建管道\n");
    return 1;
}

```

接下来，父进程使用 **fork** 函数创建一个子进程。如果 **fork** 函数返回负值，则表示创建子进程失败，程序将打印错误消息并退出。如果返回值为 0，则表示当前代码正在子进程中执行。

```

pid = fork();

if (pid < 0) {
    fprintf(stderr, "无法创建子进程\n");
    return 1;
} else if (pid == 0) {
    // 子进程关闭写入端
    close(pipefd[1]);

    // 从管道中读取数据
    bytesRead = read(pipefd[0], buffer, sizeof(buffer));
    printf("子进程收到消息: %s\n", buffer);

    // 关闭读取端
    close(pipefd[0]);
    exit(0);
}

```

在子进程中，首先关闭了管道的写入端（**pipefd[1]**），因为子进程将从管道中读取数据。然后，子进程使用 **read** 函数从管道中读取数据。**read** 函数会阻塞进程，直到有数据可读取。读取的数据存储在 **buffer** 中，并返回读取的字节数。在代码中，假设读取的数据不超过 50 个字节。子进程将读取到的数据打印出来。

最后，子进程关闭了管道的读取端（**pipefd[0]**），并通过 **exit(0)** 退出子进程。

在父进程中，首先关闭了管道的读取端（**pipefd[0]**），因为父进程将向管道中写入数据。然后，父进程使用 **write** 函数向管道中写入数据。**write** 函数将数据写入管道，使得可读进程能够读取到数据。在本示例中，我们向管道中写入了 "Hello, child!"。最后，父进程关闭了管道的写入端（**pipefd[1]**）。

实验结果分析



```
zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/4
zzq@zzq-VMware-Virtual-Platform:~$ cd 桌面/os/4
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/4$ gedit 4.c
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/4$ gcc 4.c -o 4
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/4$ ./4
子进程收到消息: Hello, child!
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/4$
```

实验五：命名匿名管道通信

实验目的

- 学习使用命名匿名管道在多进程间建立通信

实验软硬件环境

- Linux 操作系统

实验内容

- 建立父子进程，由父进程创建一个命名匿名管道；
- 由子进程向命名管道写入数据，由父进程从命名管道读取数据。

实验要求

实验设计与实现

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

#define FIFO_NAME "/tmp/myfifo"
#define BUFFER_SIZE 1024

int main() {
    int fd;
    char buffer[BUFFER_SIZE];

    // 创建命名管道
    mkfifo(FIFO_NAME, 0666);

    // 创建子进程
    pid_t pid = fork();

    if (pid < 0) {
        fprintf(stderr, "无法创建子进程\n");
        return 1;
    } else if (pid == 0) {
        // 子进程
        // 打开命名管道以进行写入
        fd = open(FIFO_NAME, O_WRONLY);
        if (fd == -1) {
            fprintf(stderr, "无法打开命名管道\n");
            return 1;
        }
    }
```

```

    }

    // 写入数据到命名管道
    char* message = "Hello, parent!";
    write(fd, message, strlen(message) + 1);

    // 关闭文件描述符
    close(fd);
    exit(0);
} else {
    // 父进程
    // 打开命名管道以进行读取
    fd = open(FIFO_NAME, O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "无法打开命名管道\n");
        return 1;
    }

    // 从命名管道中读取数据
    read(fd, buffer, BUFFER_SIZE);
    printf("父进程收到消息: %s\n", buffer);

    // 关闭文件描述符
    close(fd);

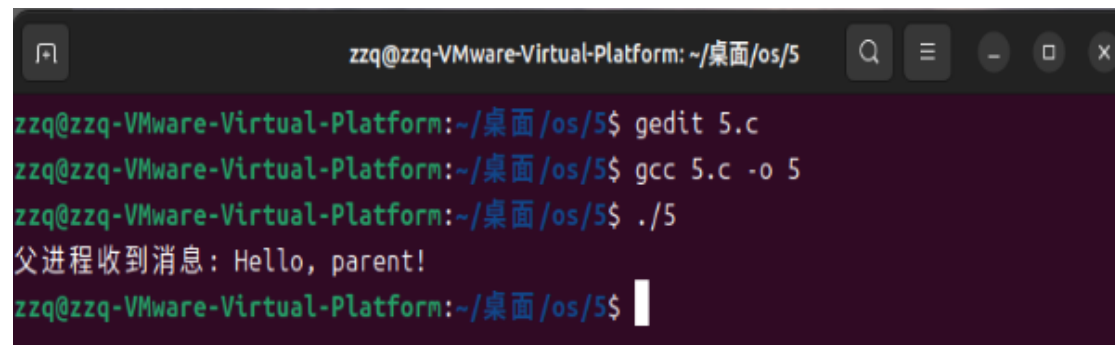
    // 删除命名管道
    unlink(FIFO_NAME);
}

return 0;
}

```

首先，使用 **mkfifo** 函数创建了一个命名管道，即 **/tmp/myfifo**。然后，使用 **fork** 函数创建了一个子进程。如果 **fork** 函数返回负值，则表示创建子进程失败，程序将打印错误消息并退出。如果返回值为 0，则表示当前代码正在子进程中执行。在子进程中，使用 **open** 函数以只写模式打开命名管道，并检查打开是否成功。然后，使用 **write** 函数向命名管道写入数据。在这段代码中，向命名管道写入了 "Hello, parent!"。在父进程中，使用 **open** 函数以只读模式打开命名管道，并检查打开是否成功。然后，使用 **read** 函数从命名管道中读取数据，并将读取到的数据存储在 **buffer** 中。最后，打印出父进程收到的消息。最后，父进程关闭文件描述符，而子进程在写入数据后关闭文件描述符。父进程还使用 **unlink** 函数删除了命名管道。

实验结果分析



```
zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/5
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/5$ gedit 5.c
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/5$ gcc 5.c -o 5
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/5$ ./5
父进程收到消息: Hello, parent!
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/5$
```

The image shows a terminal window with a dark background. The title bar at the top reads "zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/5". The terminal content shows the user editing a file named "5.c" with "gedit", compiling it with "gcc 5.c -o 5", and then running it with "./5". The output of the program is "父进程收到消息: Hello, parent!". The prompt "zzq@zzq-VMware-Virtual-Platform:~/桌面/os/5\$" is visible at the end of the last line.

实验六：信号量实现进程同步

实验目的

- 进程同步是操作系统多进程/多线程并发执行的关键之一，进程同步是并发进程为了完成共同任务采用某个条件来协调他们的活动，这是进程之间发生的一种直接制约关系。本次试验是利用信号量进行进程同步。

实验软硬件环境

- Linux 操作系统

实验内容

- 生产者进程生产产品，消费者进程消费产品；
- 当生产者进程生产产品时，如果没有空缓冲区可用，那么生产者进程必须等待消费者进程释放出一个缓冲区；
- 当消费者进程消费产品时，如果缓冲区中没有产品，那么消费者进程将被阻塞，直到新的产品被生产出来；

实验要求

- 显示生产和消费过程

实验设计与实现

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

#define KEY 1234
#define BUFFER_SIZE 5

void P(int sem_id) {
```

```

    struct sembuf buf;
    buf.sem_num = 0;
    buf.sem_op = -1;
    buf.sem_flg = SEM_UNDO;
    semop(sem_id, &buf, 1);
}

void V(int sem_id) {
    struct sembuf buf;
    buf.sem_num = 0;
    buf.sem_op = 1;
    buf.sem_flg = SEM_UNDO;
    semop(sem_id, &buf, 1);
}

int main() {
    int sem_id;
    pid_t pid;

    // 创建信号量
    sem_id = semget(KEY, 1, IPC_CREAT | 0666);
    if (sem_id == -1) {
        perror("无法创建信号量");
        exit(1);
    }

    // 初始化信号量为 1，表示可用
    semctl(sem_id, 0, SETVAL, 1);

    // 创建子进程
    pid = fork();
    if (pid == -1) {
        perror("无法创建子进程");
        exit(1);
    } else if (pid == 0) {
        // 子进程为生产者进程
        int i;
        for (i = 1; i <= 10; i++) {
            sleep(1);
            P(sem_id);
            printf("生产者生产产品: %d\n", i);
            V(sem_id);
        }
        exit(0);
    }
}

```

```

    } else {
        // 父进程为消费者进程
        int i;
        for (i = 1; i <= 10; i++) {
            sleep(2);
            P(sem_id);
            printf("消费者消费产品: %d\n", i);
            V(sem_id);
        }
        wait(NULL);
    }

    // 删除信号量
    semctl(sem_id, 0, IPC_RMID);

    return 0;
}

```

这段代码实现了使用信号量实现进程同步的功能。它包含了以下主要步骤：
引入所需的头文件：

- `<stdio.h>`：用于输入输出操作
- `<stdlib.h>`：包含了`exit`函数
- `<unistd.h>`：包含了`fork`函数和`sleep`函数
- `<sys/types.h>`：包含了进程相关的数据类型
- `<sys/ipc.h>`：包含了使用 IPC 机制的函数和数据结构
- `<sys/sem.h>`：包含了信号量相关的函数和数据结构
- `<sys/wait.h>`：包含了`wait`函数（用于等待子进程结束）

定义了两个信号量操作函数：`P`和`V`。`P`函数用于对信号量做减 1 操作（等待），`V`函数用于对信号量做加 1 操作（释放）。

在`main`函数中：

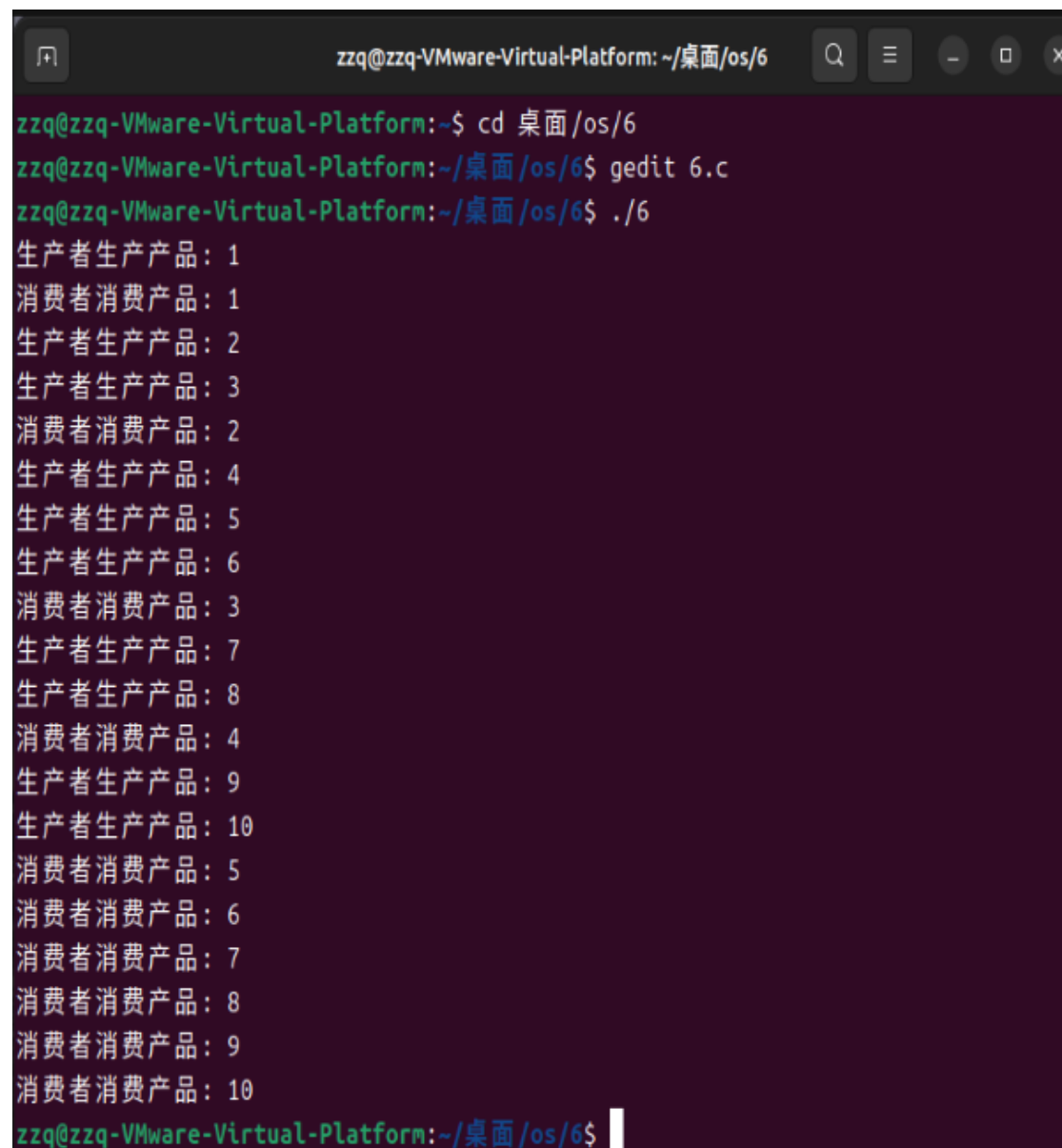
- 创建一个信号量，使用`semget`函数，并传入一个键值和标志。
- 初始化信号量的值为 1，表示可用，使用`semctl`函数。
- 创建子进程，使用`fork`函数。
- 在子进程中，使用一个循环模拟生产者进程，生产 10 个产品。在每次生产之前，使用`P`函数等待信号量，确保有空缓冲区可用，然后生产产品并使用`V`函数释放信号量。
- 在父进程中，使用一个循环模拟消费者进程，消费 10 个产品。在每次消费之前，使用`P`函数等待信号量，确保缓冲区中有产品可供消费，然后消费产品并使用`V`函数释放信号量。
- 在父进程中，使用`wait`函数等待子进程结束。
- 最后，使用`semctl`函数删除信号量。

代码中使用`P`函数和`V`函数保证了生产者和消费者的互斥访问共享资源（缓冲区），确保生产者只在有空缓冲区可用时才生产产品，消费者只在有产品可供消费时才进行消费，实

现了进程之间的同步。

在运行代码时，父进程和子进程并发执行，通过信号量实现了正确的生产者和消费者模型，确保了生产和消费的顺序和互斥性。

实验结果分析

A terminal window titled 'zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/6'. The user enters 'cd 桌面/os/6', then 'gedit 6.c', and finally './6'. The program output shows a sequence of production and consumption messages: '生产者生产产品: 1' through '生产者生产产品: 10' and '消费者消费产品: 1' through '消费者消费产品: 10'. The messages are interleaved, demonstrating the synchronization between the producer and consumer processes.

```
zzq@zzq-VMware-Virtual-Platform: ~/桌面/os/6
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/6$ cd 桌面/os/6
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/6$ gedit 6.c
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/6$ ./6
生产者生产产品: 1
消费者消费产品: 1
生产者生产产品: 2
生产者生产产品: 3
消费者消费产品: 2
生产者生产产品: 4
生产者生产产品: 5
生产者生产产品: 6
消费者消费产品: 3
生产者生产产品: 7
生产者生产产品: 8
消费者消费产品: 4
生产者生产产品: 9
生产者生产产品: 10
消费者消费产品: 5
消费者消费产品: 6
消费者消费产品: 7
消费者消费产品: 8
消费者消费产品: 9
消费者消费产品: 10
zzq@zzq-VMware-Virtual-Platform:~/桌面/os/6$
```