Linux 进程与线程理解（fork()&clone()系统调用）

一、预备知识

1、基本概念

1）程序：一组指令序列的集合，也就是存储在磁盘上的一堆有序的二进制码，如 Windows 上的 exe 可执行文件（当然还包括该程序所使用的一些存储在磁盘上的相关文件），其本身并不占用资源，是一个没有生命的实体。
2）进程：一个正在执行程序的实例，包括程序计数器、寄存器和变量的当前值。将程序加载到内存，系统为它分配好所需要的资源，这样一个正在执行的有生命的程序就成为一个进程。每启动一个程序便启动了一个进程，它们有独立的内存空间（当然，可以通过编程实现一个程序在同一时刻只能启动一个进程）。
3）线程（轻量级进程）：是进程中的执行单元，进程中真正运行的是线程。一个进程可以看做线程的容器，可以包含多个线程，它们共享此进程的地址空间和其他资源，同样也包含各自的程序计数器、堆栈、私有的数据区寄存器等。可以这样理解，进程用于把资源集中到一起，而线程则是在 CPU 上被调度执行的实体。
4）信号量：是一个非负整形变量，可以对信号量进行 PV 操作（P 使信号量减一，V 使信号量加一）。在对信号量进行 P 操作时，若信号量此时为 0，则线程变为阻塞，直到信号量加一。当信号量只取 0/1 值时，可以实现互斥。信号量也可以实现同步，用来保证一组操作顺序发生。
5）互斥量：又叫锁，用来实现线程间的互斥。互斥量是信号量的一个简化版本，它不需要信号量的计数能力，也就是信号量只取 0/1 时的状态。它的访问是随机的，因此无法实现同步。互斥量有两种状态：加锁和解锁，当处于加锁状态时，其余线程被阻塞，直到处于临界区的线程解锁，然后随机选择一个被阻塞到该互斥量的线程并允许它获得锁。
6）!!!! 条件变量：用于临界区中，如果只有互斥量的话，有时会在临界区中造成死锁，而引入条件变量便可以解决这个问题。在临界区中，当满足某个条件，该线程无法继续运行时，可以通过对某个条件变量进行操作，该操作导致自身阻塞，并对互斥量进行解锁，当该线程被再次唤醒时，重新获得锁并继续运行。

2、多道程序设计

CPU 在很短的时间内实现将一个进程切换至另一个进程，使得每个进程各运行一段时间，这样来模拟并发过程，使人们认为计算机在同时做很多事情，这种快速的切换称作多道程序设计。采用多道程序设计可以提高 CPU 的利用率。当启动系统时，会秘密启动很多进程，例如等待接受电子邮件的进程、周期性检测是否有新的病毒的进程等等，它们一起运行互不冲突，但在某一给定的瞬间只有一个进程在真正的运行（我们现在只考虑一个 CPU 的情况）。

3、为什么要引入线程

1）线程是轻量级进程，它的创建和销毁比进程更加容易。
2）在多道程序设计中，CPU 通过快速切换进程并分配给进程一个时间片来模拟并发，进程切换时需要将当前进程程序计数器、程序状态字、寄存器值等必要的资源压入堆栈，耗时较长，而同一进程的不同线程间共享进程资源，所以线程在切换时需要保存的内容少，耗时自然就少。
3）不同线程间共享资源导致通信相当简单，而进程互相之间是独立的，通信更为困难（可以通过管道进行通信）。所以将一组使用相同资源、联系较为紧密的操作通过同一进程下的不同线程来实现更为方便、快捷，也更加合理，符合我们的正常思维模式。

4、杂记

1）一个进程至少包含一个线程。
2）进程开始运行时先从主线程开始（也就是从我们编程中的 main()函数开始执行）。
3）线程分为前台线程和后台线程（守护线程），一个进程只有当所有的前台线程都结束时才可以退出，而对于后台线程，它的作用是为前台线程提供服务，所以当进程中只有后台进程时，进程会直接退出。
4）!!!! 孤儿进程和僵尸进程。
5）在每个进程中必须将没用的读写数据端及时关掉，否则会造成严重后果。考虑以下一种情况：生产者已经退出，但管道写端未关闭（即写端引用计数大于 0），而管道中已经没有内容，此时消费者的 read()会一直阻塞等待写端写入内容，消费者便永远无法退出；若管道写端已关闭（即写端引用计数为 0），则消费者的 read()会返回 0，就想读到了文件末尾。

二、原理分析

1、实验设计

以生产者/消费者问题为例，通过实验理解 fork()和 clone()两个系统调用的区别。程序要求能够创建 4 个进程或线程，其中包括两个生产者和两个消费者，生产者和消费者之间能够传递数据。

2、函数介绍

1）pid_t fork(void);
（pid_t 是一个宏定义，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，父进程返回子进程 ID；否则，出错返回-1
作用：创建一个子进程，子进程是父进程的完整复制，将获得父进程数据空间、堆、栈等资源的副本。
2）int clone (int (*fn) (void *arg) , void *stack , int flag , void *arg) ;

其中，fn 是进程所执行的函数；stack 是进程所使用的栈；arg 是调用过程的对应参数。
clone()函数功能强大，它的关键是 flag 的设定，不同的组合可以创建进程、轻量进程（即线程）甚至可以为父子进程，在这里我们选择 flag 为 CLONE_VM，CLONE_FS， CLONE_FILES，CLONE_SIGNAND，CLONE_PID 的组合，CLONE_VM 表示子进程共享父进程内存，CLONE_FS 表示子进程共享父进程的文件系统，CLONE_SIGNAND 表示子进程共享父进程的消息处理机制，CLONE_PID 是指子进程继承父进程的 id 号。
3）int pipe(int filedes[2]);
作用：创建管道，用于进程间通信。
参数：filedes[0]——用于读数据端，filedes[1]——用于写数据端。
返回值：成功返回零，否则返回-1。
4）pid_t wait (int * status);
wait()函数会阻塞当前进程，直到某个子进程结束，wait()会收集该子进程的结束状态值（一些残余信息）保存到 status 中，并把它彻底销毁。如果在调用 wait()时子进程已经结束，则 wait()会立即返回子进程结束状态值。如果不在意结束状态值，则参数可设为 NULL。
返回值：执行成功则返回子进程 ID，如果有错误发生则返回-1。

3、主要思想

1）Linux 中用 fork()创建的子进程拥有自己独立的内存，是父进程内存的一份拷贝，父子进程通过管道进行通信，所以在调用 fork()之前必须先调用 pipe(int filedes[2])创建管道，这样创建的每个子进程才能继承管道的读写数据端；当调用 fork()后，父子进程同时运行到这个位置，父进程返回子进程 ID，子进程返回 0，出错返回-1，通过这个来区分父子进程；在每个进程中必须将没用的读写数据端关掉，否则进程会一直等待；并且在父进程中调用 wait(NULL)等待所有子进程结束在退出。
2）clone()在创建进程时，可通过 flag 参数设定两个进程的关系，当设置为共享存储空间时，可以创建真正意义上的线程。生产者和消费者线程共享内存，从而可以通过共享内存直接交换数据。但是多个进程共享内存需要互斥机制，程序中定义了临界区变量 mutex 和两个信号量 product，warehouse，临界区变量用于共享内存操作的互斥，信号量分别实现了生产者和消费者的顺序执行，实现了同步。

三、具体实现

1、fork()系统调用

1）代码如下：

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

```
int filedes[2];//读写数据端

void producer(int id) //生产者
{
	printf("producer %d is running!\n",id);
	close(filedes[0]); //关闭读数据端

	char w_buf[4]; //写缓存
	if (id == 1) //生产者 1
		strcpy(w_buf, "aaa\0");
	else		//生产者 2
		strcpy(w_buf, "bbb\0");

	int i=0;
	for (i = 0; i < 5; i++)
	{
		sleep(2);	//释放进程资源
		if (write(filedes[1], w_buf, 4) == -1)
			printf("write to pipe error\n");
	}

	close(filedes[1]); //关闭写数据端
	printf("producer %d is over!\n",id);
}

void consumer(int id) //消费者
{
	printf("consumer %d is running!\n",id);
	close(filedes[1]); //关闭写数据端

	char r_buf[4]; //读缓存
	while(1)
	{
		sleep(3);	//释放进程资源
		if(read(filedes[0], r_buf, 4) == 0)
			break;
		printf("consumer %d get %s from pipe.\n", id, r_buf);
	}

	close(filedes[0]); //关闭读数据端
	printf("consumer %d is over!\n", id);
}
```

```
int main()
{
	if (pipe(filedes) == -1) //为了用 fork 创建子进程，必须首先创建管道
		return -1;

	printf("pipe is created successfully!\n");
	pid_t pid;
	int i;
	for (i = 1; i < 3; i++)    //创建生产者
	{
		pid = fork();
		if (pid == -1)
			return -2;
		else if (pid == 0)
		{
			producer(i);
			return 0;
		}
		sleep(1);
	}
	for (i = 1; i < 3; i++)    //创建消费者
	{
		pid = fork();
		if (pid == -1)
			return -2;
		else if (pid == 0)
		{
			consumer(i);
			return 0;
		}
		sleep(1);
	}

	close(filedes[0]); //在主进程中必须关闭管道读写数据端，否则会一直等待
	close(filedes[1]);

	for (i = 0; i < 4; i++) //等待所有子进程结束后主进程再结束
		wait(NULL);

	return 0;
}
```

2）实验结果：

```
runsir@ubuntu:~/OSEX/EX1$ ./fork.out
pipe is created successfully!
producer 1 is running!
producer 2 is running!
consumer 1 is running!
consumer 2 is running!
consumer 1 get aaa from pipe.
consumer 2 get bbb from pipe.
consumer 1 get aaa from pipe.
consumer 2 get bbb from pipe.
producer 1 is over!
producer 2 is over!
consumer 1 get aaa from pipe.
consumer 2 get bbb from pipe.
consumer 1 get aaa from pipe.
consumer 2 get bbb from pipe.
consumer 1 get aaa from pipe.
consumer 2 get bbb from pipe.
consumer 1 is over!
consumer 2 is over!
runsir@ubuntu:~/OSEX/EX1$
```

2、clone()系统调用

1）代码如下：

```
#define _GNU_SOURCE
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <semaphore.h>
#define WH_COUNT 8 //仓库数

pthread_mutex_t mutex;//互斥量
sem_t full_whs;    //满的仓库数
sem_t empty_whs; //空的仓库数

char warehouse[WH_COUNT][4]; //仓库
int fulled = 0;

int flag = 0; //判断主线程什么时候结束

//生产者
int producer(void* args)
{
    int id = *((int*)args);
    char buff[4];
```

```
    if (id == 0)
        strcpy(buff, "aaa\0");
    else
        strcpy(buff, "bbb\0");

    int i;
    for(i = 0; i < 5; i++)
    {
        sleep(i+1);    //表现线程速度差别
        sem_wait(&empty_whs); //判断是否还有空的仓库
        pthread_mutex_lock(&mutex);
        strcpy(warehouse[fulled],buff);
        fulled++;
        sem_post(&full_whs);
        printf("producer %d produce %s in %d\n",id,buff,fulled-1);
        pthread_mutex_unlock(&mutex);
    }
    printf("producer %d is over!\n",id);
    flag++;
}

//消费者
int consumer(void *args)
{
    int id = *((int*)args);

    int i;
    for(i = 0; i < 5; i++)
    {
        sleep(5-i);    //表现线程速度差别
        sem_wait(&full_whs); //判断是否还有满的仓库
        pthread_mutex_lock(&mutex);
        fulled--;
        sem_post(&empty_whs);
        printf("consumer %d get %s in %d\n",id,warehouse[fulled],fulled);
        pthread_mutex_unlock(&mutex);
    }
    printf("consumer %d is over!\n",id);
    flag++;
}

int main()
{
    //初始化互斥量和信号量
```

```
    pthread_mutex_init(&mutex, NULL);
    sem_init(&full_whs, 0, 0);
    sem_init(&empty_whs, 0, WH_COUNT);

    int clone_flag,arg;
    char *stack;
    clone_flag=CLONE_VM|CLONE_SIGHAND|CLONE_FS|CLONE_FILES;
    int i;
    //创建四个线程
    for(i = 0; i < 2; i++)
    {
        sleep(1);
        arg = i;
        stack = (char*)malloc(4096);
        if (0 == clone((void*)producer, &(stack[4095]), clone_flag, (void*)&arg))
            return 1;
        sleep(1);
        stack = (char*)malloc(4096);
        if (0 == clone((void*)consumer, &(stack[4095]), clone_flag, (void*)&arg))
            return 1;
    }

    while (flag != 4)
        sleep(1);
    return 0;
}
```

2）实验结果：

```
runsir@ubuntu:~/OSEX/EX1$ gcc clone.c -pthread -o clone.out
runsir@ubuntu:~/OSEX/EX1$ ./clone.out
producer 0 produce aaa in 0
producer 0 produce aaa in 1
producer 1 produce bbb in 2
producer 1 produce bbb in 3
consumer 0 get bbb in 3
producer 0 produce aaa in 3
consumer 1 get aaa in 3
producer 1 produce bbb in 3
producer 0 produce aaa in 4
consumer 0 get aaa in 4
consumer 1 get bbb in 3
producer 1 produce bbb in 3
consumer 0 get bbb in 3
producer 0 produce aaa in 3
producer 0 is over!
consumer 0 get aaa in 3
consumer 1 get bbb in 2
consumer 0 get aaa in 1
consumer 0 is over!
consumer 1 get aaa in 0
producer 1 produce bbb in 0
producer 1 is over!
consumer 1 get bbb in 0
consumer 1 is over!
runsir@ubuntu:~/OSEX/EX1$
```