第二部分 操作系统算法实验

本部分的实验内容主要是为了加深理解和掌握操作系统原理中的经典算法而安排的。其中不涉及到要封装对象的实验全部用 C 语言编写。涉及到要封装对象的实验使用 C++语言编写。实验环境均为 Linux 操作系统，开发工具为 gcc 和 g++。

实验一、进程控制实验

1.1 实验目的

加深对于进程并发执行概念的理解。实践并发进程的创建和控制方法。观察和体验进程的动态特性。进一步理解进程生命期期间创建、变换、撤销状态变换的过程。掌握进程控制的方法，了解父子进程间的控制和协作关系。练习 Linux 系统中进程创建与控制有关的系统调用的编程和调试技术。

1.2 实验说明

1）与进程创建、执行有关的系统调用说明

进程可以通过系统调用 fork()创建子进程并和其子进程并发执行.子进程初始的执行映像是父进程的一个复本.子进程可以通过 exec()系统调用族装入一个新的执行程序。父进程可以使用 wait()或 waitpid()系统调用等待子进程的结束并负责收集和清理子进程的退出状态。

fork()系统调用语法**:**

```
#include <unistd.h>
pid_t fork(void);
```

fork 成功创建子进程后将返回子进程的进程号,不成功会返回-1.

exec 系统调用有一组 **6** 个函数**,**其中示例实验中引用了 **execve** 系统调用语法**:**

```
#include <unistd.h>
int execve(const char *path,    const char *argv[],    const char * envp[]);
```

path 要装入的新的执行文件的绝对路径名字符串.

argv[] 要传递给新执行程序的完整的命令参数列表(可以为空).

envp[] 要传递给新执行程序的完整的环境变量参数列表(可以为空).

Exec 执行成功后将用一个新的程序代替原进程，但进程号不变，它绝不会再返回到调用进程了。如果 exec 调用失败，它会返回-1。

wait() 系统调用语法**:**

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid,int *status,int option);
```

status 用于保留子进程的退出状态

pid 可以为以下可能值：

-1 等待所有 PGID 等于 PID 的绝对值的子进程

1 等待所有子进程

0 等待所有 PGID 等于调用进程的子进程

>0 等待 PID 等于 pid 的子进程

option 规定了调用 waitpid 进程的行为：

WNOHANG 没有子进程时立即返回

WUNTRACED 没有报告状态的进程时返回

wait 和 waitpid 执行成功将返回终止的子进程的进程号，不成功返回-1。

getpid()系统调用语法**:**

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

getpid 返回当前进程的进程号，getppid 返回当前进程父进程的进程号

2) 与进程控制有关的系统调用说明

可以通过信号向一个进程发送消息以控制进程的行为。信号是由中断或异常事件引发的，如：键盘中断、定时器中断、非法内存引用等。信号的名字都以 SIG 开头，例如 SIGTERM、SIGHUP。可以使用 kill -l 命令查看系统当前的信号集合。

信号可在任何时间发生，接收信号的进程可以对接收到的信号采取3种处理措施之一：

- 忽略这个信号
- 执行系统默认的处理
- 捕捉这个信号做自定义的处理

信号从产生到被处理所经过的过程：

产生(generate)-> 挂起(pending)-> 派送(deliver)-> 部署(disposition)或忽略(igore)

一个信号集合是一个 C 语言的 sigset_t 数据类型的对象，sigset_t 数据类型定义在<signal.h>中。被一个进程忽略的所有信号的集合称为一个信号掩码(mask)。

从程序中向一个进程发送信号有两种方法：调用 shell 的 kill 命令，调用 kill 系统调用函数。kill 能够发送除杀死一个进程(SIGKILL、SIGTERM、SIGQUIT)

之外的其他信号，例如键盘中断(Ctrl+C)信号 SIGINT,进程暂停(Ctrl+Z)信号 SIGTSTP 等等。

调用 Pause 函数会令调用进程的执行挂起直到一个任意信号到来后再继续运行。

调用 sleep 函数会令调用进程的执行挂起睡眠指定的秒数或一个它可以响应的信号到来后继续执行。

每个进程都能使用 signal 函数定义自己的信号处理函数，捕捉并自行处理接收的除 SIGSTOP 和 SIGKILL 之外的信号。以下是有关的系统调用的语法说明。

kill 系统调用语法：

#include <sys/types.h>
#include <signal.h>

int kill(pid_t pid, int sig);
pid 接收信号的进程号
signal 要发送的信号
kill 发送成功返回接收者的进程号，失败返回-1。

pause 系统调用语法：

＃include <unistd.h>
int pause(void);
pause 挂起调用它的进程直到有任何信号到达。调用进程不自定义处理方法，则进行信号的默认处理。只有进程自定义了信号处理方法捕获并处理了一个信号后，pause 才会返回调进程。pause 总是返回-1,并设置系统变量 errno 为 EINTR。

sleep 系统调用语法：

#include <unistd.h>
unsigned int sleep(unsigned int seconds);
seconds 指定进程睡眠的秒数
如果指定的秒数到，sleep 返回 0。

signal 系统调用语法为:

#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
signum 要捕捉的信号
handler 进程中自定义的信号处理函数名
signal 调用成功会返回信号处理函数的返回值，不成功返回-1,并设置系统变量 errno 为 SIG_ERR。

1.3 示例实验

以下实验示例程序应实现一个类似子 shell 子命令的功能,它可以从执行程序中启动另一个新的子进程并执行一个新的命令和其并发执行.

1) 打开一终端命令行窗体，新建一个文件夹，在该文件夹中建立以下名为**pctl.c**的**C**语言程序：

```
/*
   * Filename                    : pctl.c
   * copyright                 : (C) 2006 by zhanghonglie
   * Function                   :  父子进程的并发执行
*/
#include "pctl.h"
int main(int argc, char *argv[])
{
   int i;
   int pid;       //存放子进程号
   int status;     //存放子进程返回状态
```

```
  char *args[] = {"/bin/ls","-a",NULL};    //子进程要缺省执行的命令
  signal(SIGINT,(sighandler_t)sigcat); //注册一个本进程处理键盘中断的函数
  pid=fork() ; //建立子进程
  if(pid<0)    //  建立子进程失败?
  {
     printf("Create Process fail!\n");
     exit(EXIT_FAILURE);
  }
  if(pid == 0) //  子进程执行代码段
  {
     //报告父子进程进程号
       printf("I am Child process %d\nMy father is %d\n",getpid(),getppid());
       pause();      //暂停，等待键盘中断信号唤醒
       //子进程被键盘中断信号唤醒继续执行
       printf("%d child will Running:  \n",getpid()); //
       if(argv[1] != NULL){
          //如果在命令行上输入了子进程要执行的命令
          //则执行输入的命令
             for(i=1; argv[i] != NULL; i++)    printf("%s ",argv[i]);
               printf("\n");
             //装入并执行新的程序
            status = execve(argv[1],&argv[1],NULL);
         }
  else{
          //如果在命令行上没输入子进程要执行的命令
          //则执行缺省的命令
            for(i=0; args[i] != NULL; i++)    printf("%s ",args[i]);
            printf("\n");
              //装入并执行新的程序
            status = execve(args[0],args,NULL);
         }
  }
else //父进程执行代码段
{
    printf("\nI am Parent process    %d\n",getpid()); //报告父进程进程号
    if(argv[1] != NULL){
      //如果在命令行上输入了子进程要执行的命令
      //则父进程等待子进程执行结束
          printf("%d    Waiting for child done.\n\n");
            waitpid(pid,&status,0);     //等待子进程结束
          printf("\nMy child exit! status = %d\n\n",status);
     }
  else{
       //如果在命令行上没输入子进程要执行的命令
       //唤醒子进程，与子进程并发执行不等待子进程执行结束，
                  if(kill(pid,SIGINT) >= 0)
```

```
                printf("%d Wakeup %d child.\n",getpid(),pid) ;
            printf("%d don't Wait for child done.\n\n",getpid());
        }
    }
    return EXIT_SUCCESS;
}
```

2) 再建立以下名为 **pctl.h** 的 **C** 语言头文件：

```
#include <sys/types.h>
#include <wait.h>
#include <unistd.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
//进程自定义的键盘中断信号处理函数
typedef void (*sighandler_t) (int);
void sigcat(){
    printf("%d Process continue\n",getpid());
}
```

3) 建立以下项目管理文件 **Makefile**

```
head = pctl.h
srcs = pctl.c
objs = pctl.o
opts = -g -c
all:   pctl
pctl:       $(objs)
            gcc $(objs) -o    pctl
pctl.o:     $(srcs)    $(head)
            gcc $(opts) $(srcs)
clean:
            rm pctl    *.o
```

4) 输入 **make** 命令编译连接生成可执行的 **pctl** 程序

```
$ gmake
gcc -g -c pctl.c
gcc pctl.o -o    pctl
```

5) 执行 **pctl** 程序**(**注意进程号是动态产生的**,**每次执行都不相同**)**

```
$ ./pctl
I am Child process 4113
My father is 4112

I am Parent process    4112
Wakeup 4113 child.
4112 don't Wait for child done.
```

4113 Process continue
4113 child will Running: /bin/ls -a
. .. Makefile pctl pctl.c pctl.h pctl.o
$
以上程序的输出说明父进程 4112 创建了一个子进程 4113，子进程执行被暂停。父进程向子进程发出键盘中断信号唤醒子进程并与子进程并发执行。父进程并没有等待子进程的结束继续执行先行结束了（此时的子进程成为了孤儿进程，不会有父进程为它清理退出状态了）。而子进程继续执行，它变成了列出当前目录所有文件名的命令 ls -a。在完成了列出文件名命令之后，子进程的执行也结束了。此时子进程的退出状态将有初始化进程为它清理。

6) 再次执行带有子进程指定执行命令的 **pctl** 程序**:**

$ ./pctl /bin/ls -l
I am Child process 4223
My father is 4222

I am Parent process 4222
4222 Waiting for child done.
可以看到这一次子进程仍然被挂起，而父进程则在等待子进程的完成。为了检测父子进程是否都在并发执行，请输入 ctrl+z 将当前进程放入后台并输入 ps 命令查看当前系统进程信息，显示如下：
[1]+ Stopped ./pctl /bin/ls -l
$ *ps -l*

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
0	S	0	4085	4083	0	76	0	-	1413	wait	pts/1	00:00:00	bash
0	T	0	4222	4085	0	76	0	-	360	finish	pts/1	00:00:00	pctl
1	T	0	4223	4222	0	76	0	-	360	finish	pts/1	00:00:00	pctl
0	R	0	4231	4085	0	78	0	-	1302	-	pts/1	00:00:00	ps

可以看到当前系统中同时有两个叫 pctl 的进程，它们的进程号分别是 4222 和 4223。它们的状态都为¡ºT¡»说明当前都被挂起。4223 的父进程是 4222，而 4222 的父进程是 4085，也就是 bash-shell。为了让 pctl 父子进程继续执行，请输入 fg 命令让 pctl 再次返回前台，显示如下：
$ fg
./pctl /bin/ls -l
现在 pctl 父子进程从新返回前台。我们可以通过键盘发键盘中断信号来唤醒 pctl 父子进程继续执行，输入 ctrl+c，将会显示：
4222 Process continue
4223 Process continue
4223 child will Running: /bin/ls -l
total 1708
-rw-r--r-- 1 root root 176 May 8 11:11 Makefile
-rwxr-xr-x 1 root root 8095 May 8 14:08 pctl
-rw-r--r-- 1 root root 2171 May 8 14:08 pctl.c
-rw-r--r-- 1 root root 269 May 8 11:10 pctl.h
-rw-r--r-- 1 root root 4156 May 8 14:08 pctl.o

My child exit! status = 0
以上输出说明了子进程在捕捉到键盘中断信号后继续执行了指定的命令，按我们要求的长格式列出了当前目录中的文件名，父进程在接收到子进程执行结束的信号后将清理子进程的退出状态并继续执行，它报告了子进程的退出编码（0 表示子进程正常结束)最后父进程也结束执行。

1.4 独立实验

参考以上示例程序中建立并发进程的方法，编写一个多进程并发执行程序。父进程首先创建一个执行 ls 命令的子进程然后再创建一个执行 ps 命令的子进程，并控制 ps 命令总在 ls 命令之前执行。

1.5. 实验要求

根据实验中观察和记录的信息结合示例实验和独立实验程序，说明它们反映出操作系统教材中进程及处理机管理一节讲解的进程的哪些特征和功能？在真实的操作系统中它是怎样实现和反映出教材中讲解的进程的生命期、进程的实体和进程状态控制的。你对于进程概念和并发概念有哪些新的理解和认识？子进程是如何创建和执行新程序的？信号的机理是什么？怎样利用信号实现进程控制？根据实验程序、调试过程和结果分析写出实验报告。

实验二、进程通信实验

2.1 实验目的

通过 Linux 系统中管道通信机制，加深对于进程通信概念的理解，观察和体验并发进程间的通信和协作的效果 ，练习利用无名管道进行进程通信的编程和调试技术。

2.2 实验说明

管道 pipe 是进程间通信最基本的一种机制,两个进程可以通过管道一个在管道一端向管道发送其输出,给另一进程可以在管道的另一端从管道得到其输入.管道以半双工方式工作,即它的数据流是单方向的.因此使用一个管道一般的规则是读管道数据的进程关闭管道写入端,而写管道进程关闭其读出端.

⑴ **pipe** 系统调用的语法为**:**

#include <unistd.h>
int pipe(int pipe_id[2]);
如果 pipe 执行成功返回 0, pipe_id[0]中和 pipe_id[1]将放入管道两端的描述符.
出错返回-1.

2.3 示例实验

以下示例实验程序要实现并发的父子进程合作将整数 X 的值从 1 加到 10 的功能。它们通过管道相互将计算结果发给对方。

在新建文件夹中建立以下名为 **ppipe.c** 的 **C** 语言程序

```
/*
   * Filename                : ppipe.c
   * copyright               : (C) 2006 by zhanghonglie
   * Function                : 利用管道实现在父子进程间传递整数
  */
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   int pid;                //进程号
   int pipe1[2];     //存放第一个无名管道标号
   int pipe2[2];    //存放第二个无名管道标号
   int x;              //  存放要传递的整数
      //使用 pipe()系统调用建立两个无名管道。建立不成功程序退出，执行终止
    if(pipe(pipe1) < 0){
        perror("pipe not create");
      exit(EXIT_FAILURE);
```

```
  }
  if(pipe(pipe2) < 0){
     perror("pipe not create");
  exit(EXIT_FAILURE);
  }
  //使用 fork()系统调用建立子进程,建立不成功程序退出，执行终止
if((pid=fork()) <0){
      perror("process not create");
   exit(EXIT_FAILURE);
  }
//子进程号等于 0 表示子进程在执行,
else if(pid == 0){
  //子进程负责从管道 1 的 0 端读,管道 2 的 1 端写，
  //所以关掉管道 1 的 1 端和管道 2 的 0 端。
     close(pipe1[1]);
     close(pipe2[0]);
       //每次循环从管道 1 的 0 端读一个整数放入变量 X 中,
       //并对 X 加 1 后写入管道 2 的 1 端，直到 X 大于 10
       do{
            read(pipe1[0],&x,sizeof(int));
            printf("child %d read: %d\n",getpid(),x++);
            write(pipe2[1],&x,sizeof(int));
       }while( x<=9 );
     //读写完成后,关闭管道
       close(pipe1[0]);
       close(pipe2[1]);
       //子进程执行结束
        exit(EXIT_SUCCESS);
   }
  //子进程号大于 0 表示父进程在执行,
  else{
  //父进程负责从管道 2 的 0 端读,管道 1 的 1 端写，
  //所以关掉管道 1 的 0 端和管道 2 的 1 端。
        close(pipe1[0]);
        close(pipe2[1]);
        x=1;
       //每次循环向管道 1 的 1 端写入变量 X 的值,并从
       //管道 2 的 0 端读一整数写入 X 再对 X 加 1，直到 X 大于 10
        do{
              write(pipe1[1],&x,sizeof(int));
              read(pipe2[0],&x,sizeof(int));
              printf("parent %d read: %d\n",getpid(),x++);
        }while(x<=9);
  //读写完成后,关闭管道
  close(pipe1[1]);
  close(pipe2[0]);
   }
```

```
    //父进程执行结束
     return EXIT_SUCCESS;
}
```

2) 在当前目录中建立以下**Makefile**文件：

```
srcs =    ppipe.c
objs =    ppipe.o
opts =    -g -c
all:        ppipe
ppipe:    $(objs)
          gcc $(objs)    -o ppipe
ppipe.o: $(srcs)
          gcc    $(opts)    $(srcs)
clean:
          rm ppipe *.o
```

3) 使用**make**命令编译连接生成可执行文件**ppipe:**

$ gmake

```
gcc    -g -c    ppipe.c
 gcc ppipe.o    -o ppipe
```

4) 编译成功后执行**ppipe:**命令**:**

$ ./ppipe

```
child 8697 read: 1
parent 8696 read: 2
child 8697 read: 3
parent 8696 read: 4
child 8697 read: 5
parent 8696 read: 6
child 8697 read: 7
parent 8696 read: 8
child 8697 read: 9
parent 8696 read: 10
```

可以看到以上程序的执行中父子进程合作将整数 X 的值从 1 加到了 10。

2.4 独立实验

设有二元函数 f(x,y) = f(x) + f(y)

其中：

f(x) = f(x-1) * x	(x >1)
f(x)=1	(x=1)
f(y) = f(y-1) + f(y-2)	(y> 2)
f(y)=1	(y=1,2)

请编程建立 3 个并发协作进程，它们分别完成 f(x,y)、f(x)、f(y)

2.5 实验要求

根据示例实验程序和独立实验程序观察和记录的调试和运行的信息，说明它们反映出操作系统教材中讲解的进程协作和进程通信概念的哪些特征和功能？在真实的操作系统中它是怎样实现和反映出教材中进程通信概念的。你对于进程协作和进程通信的概念和实现有哪些新的理解和认识？管道机制的机理是什么？怎样利用管道完成进程间的协作和通信？根据实验程序、调试过程和结果分析写出实验报告。