实验二　进程的控制

一．实验目的

1．熟悉和理解进程和进程树的概念，掌握有关进程的管理机制
2．通过进程的创建、撤销和运行加深对进程并发执行的理解
3．明确进程与程序、并行与串行执行的区别
4．掌握用 C 程序实现进程控制的方法

二．实验工具与设备

已安装 Linux 操作系统的计算机。

三．实验预备内容

1．阅读 Linux 的 sched.h 源代码文件，加深对进程管理概念的理解
2．阅读 Linux 的 fork.c 源代码文件，分析进程的创建过程

四．实验内容

1．了解系统调用 fork( ), exec 系列函数，exit( ), wait( )的功能和实现过程

（1）进程的创建

用户在使用 Linux 系统的时候，每次在终端下面输入一行命令，就由 shell 进程接收这个命令并创建一个新的进程，这个新的进程还可以通过 fork()系统调用，继续创建自己的子进程。系统中的多个进程构成一棵进程树。实际上，在 Linux 系统启动时，最早产生的进程是 idle 进程，其 pid 号为 0，该进程会创建一个内核线程，该线程进行一系列初始化动作后最终会执行/sbin/init 文件，执行该文件的结果是运行模式从核心态切换到用户态，该线程演变成了用户进程 init，pid 为 1。init 进程是一个非常重要的进程，一切用户态进程都是它的后代进程。

a. 派生进程

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

调用 fork 时，系统将创建一个与当前进程相同的新进程，其与原有进程具有相同的数据、连接关系和在程序同一处执行的连续性。原进程称为父进程，新生进程称为子进程。子进程是父进程的一个拷贝，子进程获得同父进程相同的数据，但是同父进程使用不同的数据段和堆栈段。子进程被创建以后，处于可运行状态，与父进程以及系统中的其他进程平等地参与系统调度。

fork 调用将执行两次返回，即分别从父进程和子进程分别返回，即子进程一旦创建就绪，就与父进程一样被平等地调度执行。因此，从 fork 返回以后，不能确切知道执行哪一个进程。从父进程返回时，返回值（>0）为子进程的进程标识号 PID；而从子进程返回时，返回值为 0，**并且返回都将执行 fork 之后的语句**。调用出错时返回值为-1，并将 errno 置为相应值。

调用 vfork 的作用与 fork 基本相同，但 vfork 并不完全拷贝父进程的数据段，而是和父进程共享数据段。调用执行 vfork 函数返回之前，父进程被阻塞，子进程先运行，直到从 vfork 调用返回。然后，子进程继续执行，可以调用 exec 执行新的进程，或调用 exit 结束其运行。此后，父进程才被唤醒，与子进程平等地被系统调度。因此，如果子进程在调用 exec 之前等待父进程，由于父进程因为执行 vfork 被阻塞，会造成死锁。

函数 vfork 的主要用途是创建子进程以后，由子进程调用 exec 函数启动其他进程，使新启动的其他进程以该子进程的进程标识号身份执行，但拥有自己的程序段和数据区。

例 2.1:

```
#include<sys/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
 pid_t pid;     char *message;     int n;
 printf("Fork program starting\n");
 pid=fork()
 switch(pid){
   case –1:
     printf("Fork error!\n"); exit(1);
   case 0:
    message= "Child process is printing.";
      n=5;
      for(;n>0;n--)
      { puts(message);   sleep(1);}
      break;
   default:
      message= "Parent process id printing.";
      n=3;
      for(;n>0;n--)
      { puts(message); sleep(1);}
      break;
   }
   exit(0);
}
```

b. 创建执行其他程序的进程

可以使用 exec 族的函数执行新的程序，以新的子进程来完全替代原有的进程。

```
#include<unistd.h>
int execl(const char *pathname, const char *arg, …, (char *)0);
int execlp(const char *filename, const char *arg, …, (char *)0);
int execle(const char *pathname, const char *arg, …, (char *)0, const char *envp[ ]);
int execv(const char *pathname, char *const argv[ ]);
int execvp(const char *filename, char *const argv[ ]);
int execve(const char *pathname, char *const argv[ ], char *const envp[ ]);
```

函数名中含有字母"l"的函数，其参数个数不定。其参数由所调用程序的命令行参数列表组成，最后一个 NULL 表示结束。函数名中含有字母"v"的函数，则是使用一个字符串数组指针 argv 指向参数列表，这一字符串数组和含有"l"的函数中的参数列表完全相同，也同样以 NULL 结束。函数名中含有字母"p"的函数可以自动在环境变量 PATH 指定的路径中搜索要执行的程序。函数名中含有字母"e"的函数，比其他函数多含有一个参数 envp。该参数是字符串数组指针，用于指定环境变量。

例 2.2: 派生一个子进程后，在子进程中使用 execl 函数调用 shell 命令 sh。

```
#include<sy/types.h>
#include<stdio.h>
#include<unistd.h>
int main()
{
 pid_t pid;
 if(pid=vfork())<0) {printf("Fork error!\n"); exit(1);}
 else if(pid= =0)
 {
  printf("Child process PID:%d.\n",getpid());
  setenv("PS1","CHILD\\$",1);
  printf("Process %4d: calling exec.\n", getpid());
  if(execl("/bin/sh","bin/sh","arg2",NULL)<0)
  {
   printf("Process %4d:execle error!\n",getpid());
   exit(0);
  }
  printf("Process %4d: You should never see this
          because the child is already gone.\n",
          gepid());
  printf("Process %4d: The child process is
         exiting.\n", getpid());
 }
 else
 {
  printf("Parent process PID: %4d.\n",getpid());
  printf("Process %4d: The parent has fork process
          %d.\n", pid);
  printf("Process %4d: The Child had called exec or
          has exited.\n", getpid());
 }
 return 0;
}
```

（2）进程等待

当一个进程结束时，Linux 系统将产生一个 SIGCHLD 信号通知其父进程。在父进程未查询子进程结束的原因时，该子进程虽然停止了，但并未完全结束。此时这一子进程被称为僵尸进程（zombie process）。例如，在有些情况下父进程先于子进程退出，于是会看到在系统提示符"$"后子进程仍然在连续输出信息，这对用户是非常不友好的。我们可以使用系统调用 wait，来让父进程处于等待状

态，直到子进程退出后才继续执行下面的语句。

#include<sys/types.h>

#include<sys/wait.h>

pid_t wait(int *stat_loc);

参数 stat_loc 是一个整型指针，当子进程结束时，将子进程的结束状态字存放在该指针指向的缓冲区。当调用 wait 时，父进程将被挂起，直至该进程的某个子进程结束时，该调用返回。如果没有子进程，则错误返回。调用成功，返回值为子进程的进程号；调用失败时，返回值为-1。

例 2.3:

```
#include<sys/types.h>
#include<stdio.h>
#include<sys/wait.h>
#include<unistd.h>
int main()
{
 pid_t pid;     char *message;     int n,exit_code;
 printf("Fork program starting\n");
 pid=fork()
 switch(pid){
   case –1:
     printf("Fork error!\n"); exit(1);
   case 0:
     message= "Child process is printing.";
     n=5;   exit_code=37;
     break;
   default:
     message="Parent process id printing.";
     n=3; exit_code=0;
     break;
   }
 for(;n>0;n--)
 { puts(message); sleep(1);}
 if(pid)
 {
  int stat_val;
  pid_t child_pid;
  child_pid=wait(&stat_val);
  printf("Child has finished: PID=%d.\n",child_pid);
  if(WIFEXITED(stat_val))
     printf("Child exited with code %d.\n",
          WEXITSTATUS(stat_val));
  else
     printf("Child terminated abnormally.\n");
 }
 exit(0);
}
```

（3）进程的终止

进程结束可通过相应的函数实现：

#include<stdlib.h>

void exit(int status); //终止正在运行的程序，关闭所有被该文件打开的文件描述符

int atexit(void (*function)(void)); //用于注册一个不带参数也没有返回值的函数以供程序正常退出时被调用。参数 function 是指向所调用程序的文件指针。调用成功返回 0，否则返回-1，并将 errno 设置为相应值

int on_exit(void (*function)(int,void *),void *arg); //作用与 atexit 类似，不同是其注册的函数具有参数，退出状态和参数 arg 都是传递给该函数使用

void abort(void); //用来发送一个 SIGABRT 信号，该信号将使当前进程终止

例 2.4：

```
#include<sys/types.h>
#include<sys/wait.h>
#include<stdio.h>
void h_exit(int status);
static void forerror(void);
static void waiterror(void);
int main(void)
{
 pid_t pid;     int status;
 if(pid=fork())<0)    atexit(forkerror);
 else if(pid==0) abort();
 if(wait(&status)!=pid) atexit(waiterror);
 h_exit(status);
}

void h_exit(int status)
{
 if(WIFEXITED(status))
    printf("Normal termination, exit status=%d.\n",
          WEXITSTATUS(status));
 else if(WIFSIGNALED(status))
    printf("Abnormal termination, exit status=%d.\n",
          WEXITSTATUS(status));
}

void forkerror(void)
{   printf("Fork error!\n");   }

void waiterror(void)
{ printf("Wait error!\n");}
```

（4）system 函数

用户可以使用该函数来在自己的程序中调用系统提供的各种命令。

#include<stdlib.h>

int system(const char *cmdstring);

参数 cmdstring 是一个字符串指针，指向表示命令行的字符串。该函数的实现是通过调用 fork、exec 和 waitpid 函数来完成的，其中任意一个调用失败则 system 函数的调用失败，故返回值较复杂。例 2.5:

```
#include<stdlib.h>
#include<stdio.h>
int main()
{
 printf("running ps with system.\n");
 system("ps –ax");
 printf("Done.\n");
 exit(0);
}
```

（5）进程组

一个进程除了进程 ID 外，还有一个进程组 ID。进程组是一个或多个进程的集合，同一个进程组中进程都有一个统一的进程组 ID。

#include<sys/types.h>

#include<unistd.h>

pid_t getpgrp(void); //用于返回调用它的进程的进程组号

int setpgid(pid_t pid,pid_t pgid); //创建一个新的进程组或将一个进程加入一个已存在的进程组。当参数 pid 和 pgid 相等时，用于创建一个新的进程组；当 pgid 是一个已存在的进行组 ID 时，将 pid 代表的进程加入该进程组。返回值：0（成功），-1（失败）

（6）时间片的分配

a. **设置和获取进程的调度策略和参数**

#include<sched.h>

int shced_setscheduler(pid_t pid, int policy, const struct sched_param *param); //设置,0(成功),否则-1

int sched_getscheduler(pid_t pid); //获取，非负数（成功），-1（失败）

参数 param 用于保存进程的调度参数

policy 表示所设置的调度策略:SCHED_OTHER:缺省的调度策略,按通常方法分配时间片;

SCHED_FIFO：对应于先进先出的规则，实时分配时间片，可以抢占使用 SCHED_OTHER 的进程

SCHED_RR ：轮换规则，实时分配时间片

b. **优先级设定**

优先级的值越小，优先权越高

#include<unistd.h>

int nice(int inc); //改变进程的动态优先级，inc 为所设的值，返回值：0（成功），-1（失败）

#include<sys/time.h>

#include<sys/resource.h>

int setpriority(int which, int who, int prio); //设置进程、进程组或用户的动态优先级

int getpriority(int which, int who); //获取进程、进程组或用户的动态优先级，which 用于指定所操作的对象：PRIO_PROECSS(进程), PRIO_PGRP(进程组), PRIO_USER(用户)；who 用于指定函数所设置的进程；prio 用于指定进程优先级（-20~20）

（7）多线程库

- int pthread_create (pthread_t *thread,const pthread_attr_t *attr, void *(*routines)(void *), void *arg);

功能：创建以函数 routines 为线程体，以 arg 为参数，具有 attr 线程属性的线程。

参数表：thread：返回参数，新线程的句柄；attr：新生成线程的属性，如果值为 NULL，则具有默认的线程属性设置；routines：线程指定运行的参数，该参数必须具有 void *返回值；arg：该线程运行函数的参数。

返回值：如果成功创建该线程，则函数返回 0，否则返回一个非 0 的错误码。

头文件：<pthread.h>

- int pthread_join (pthread_t thread, void **status);

功能：等待一个线程结束，并将结束时的状态写入 status。

参数：thread: 被等待的线程；status: 输出参数，线程结束时的状态。

返回值：0（成功）；否则，返回非 0 的错误码。

头文件：<pthread.h>

- int pthread_mutex_lock (pthread_mutex_t *mutex);

功能：如果信号量 mutex 未加锁，则为其加锁，否则，该线程阻塞直到 mutex 解锁。

参数：mutex: 需要加锁的信号量。

返回值：0（成功）；否则，返回非 0 的错误码。

头文件：<pthread.h>

- int pthread_mutex_unlock (pthread_ t *mutex);

功能：为指定的信号量解锁。

参数：mutex: 需要加锁的信号量。

返回值：0（成功）；否则，返回非 0 的错误码。

头文件：<pthread.h>

例 2.6: 创建一个线程，进行运算和显示输出信息。主程序也对相同的变量进行运算，输出相应信息。

```
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>
#include<stdio.h>
int myglobal;
pthread_mutex_t
mytmutex=PTHREAD_MUTEX_INITIALIZER;
                                    //定义静态互斥
void *thread_function(void *arg)
{
 int i,j;
 for(i=0;i<20;i++)
 {
  pthread_mutex_lock(&mymutex); //加锁
  j=myglobal;
  j++;
  printf(“.”);
  fflush(stdout);
  sleep(1);
  myglobal=j;
  pthread_mutex_unlock(&mymutex); //解锁
 }
 Return NULL;
}
int main(void)
{
 pthread_t mythread;
 int i;
 if(pthread_create(&mythread, NULL,
                    thread_function, NULL))
 {
  printf(“error creating thread.”);
  abort();
 }
 for(i=0;i<20;i++)
 {
  pthread_mutex_lock(&mymutex); //加锁
  myglobal++;
  pthread_mutex_unlock(&mymutex); //解锁
  printf(“.”);
  fflush(stdout);
  sleep(1);
 }
 if(pthread_join(mythread, NULL))
 {
  printf(“error joining thread.”);
  abort();
 }
 printf(“\nmyglobal equals %d\n”,myglobal);
 exit(0);
}
```

2．程序设计

（1）编写一段程序，使用系统调用 fork( )创建两个子进程。当此程序运行时，在系统中有一个父进程和两个子进程活动。让每一个进程在屏幕上显示一个字符：父进程显示字符’a’；子进程分别显示字符’b’和’c’。观察屏幕上的显示结果，并分析原因。

（2）编写一段程序，使用系统调用 fork( )创建一个子进程。子进程通过系统调用 exec 系列函数调用命令 ls，调用 exit( )结束。而父进程则调用 waitpid( )等待子进程结束，并在子进程结束后显示子进程的标识符，然后正常结束。

五．思考题

1．怎样用 C 程序实现进程的控制？ 当首次调用新创建进程时，其入口在哪里？

2．系统调用 fork( )是如何创建进程的？系统调用 exit( )是如何终止一个进程的？

3．系统调用 exec 系列函数是如何更换进程的可执行代码的？