# 第二次操作系统实验总结

## 一、用到的系统函数：

#### 1．stat

定义：

int stat(const char \* file\_name,struct stat \*buf);

表头文件：

#include<sys/stat.h>

#include<unistd.h>

说明：

stat()用来将参数file\_name所指的文件状态, 复制到参数buf所指的结构中。

返回值：

执行成功（即文件确实存在）则返回0, 失败返回-1, 错误代码存于errno

#### 2．remove

定义：

int remove(const char \* pathname);

表头文件：

#include<stdio.h>

说明：

remove()会删除参数pathname指定的文件。如果参数pathname为一文件, 则调用unlink()处理, 若参数pathname为一目录, 则调用rmdir()来处理。请参考unlink()与rmdir()。

返回值：

成功则返回0, 失败则返回-1, 错误原因存于errno。

#### 3．mkfifo

定义：

int mkfifo(const char \* pathname,mode\_t mode);

表头文件：

#include<sys/types.h>

#include<sys/stat.h>

说明：

mkfifo()会依参数pathname建立特殊的FIFO文件, 该文件必须不存在, 而参数mode为该文件的权限(mode%~umask), 因此umask值也会影响到FIFO文件的权限。Mkfifo()建立的FIFO文件其他进程都可以用读写一般文件的方式存取。当使用open()来打开FIFO文件时, O\_NONBLOCK旗标会有影响

1、当使用O\_NONBLOCK 旗标时, 打开FIFO 文件来读取的操作会立刻返回, 但是若还没有其他进程打开FIFO 文件来读取, 则写入的操作会返回ENXIO 错误代码。

2、没有使用O\_NONBLOCK 旗标时, 打开FIFO 来读取的操作会等到其他进程打开FIFO文件来写入才正常返回。同样地, 打开FIFO文件来写入的操作会等到其他进程打开FIFO 文件来读取后才正常返回。

返回值：

若成功则返回0, 否则返回-1, 错误原因存于errno中。

#### 4．open，close，read，write，creat等文件处理函数

略（见上次实验）

#### 5．Kill，signal，sigaction等信号处理函数

略（见上次实验）

#### 6．waitpid

定义：

pid\_t waitpid(pid\_t pid,int \* status,int options);

表头文件：

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

说明：

waitpid()会暂时停止目前进程的执行, 直到有信号来到或子进程结束。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。子进程的结束状态值会由参数status返回, 而子进程的进程识别码也会一快返回。如果不在意结束状态值, 则参数status可以设成NULL。参数pid为欲等待的子进程识别码, 其他数值意义如下:  
pid<-1 等待进程组识别码为pid绝对值的任何子进程。  
pid=-1 等待任何子进程, 相当于wait()。  
pid=0 等待进程组识别码与目前进程相同的任何子进程。  
pid>0 等待任何子进程识别码为pid的子进程。  
参数option可以为0 或下面的OR 组合  
WNOHANG **如果没有任何已经结束的子进程则马上返回, 不予以等待。**  
WUNTRACED 如果子进程进入暂停执行情况则马上返回, 但结束状态不予以理会。

子进程的结束状态返回后存于status, 底下有几个宏可判别结束情况  
WIFEXITED(status)如果子进程正常结束则为非0值。  
WEXITSTATUS(status)取得子进程exit()返回的结束代码, 一般会先WIFEXITED 来判断是否正常结束才能使用此宏。  
WIFSIGNALED(status)如果子进程是因为信号而结束（如CTRL+C）则此宏值为真。WTERMSIG(status)取得子进程因信号而中止的信号代码,一般会先WIFSIGNALED 判断后才使用此宏。  
WIFSTOPPED(status)如果子进程处于暂停执行情况则此宏值为真。一般只有使用WUNTRACED 时才会有此情况。  
WSTOPSIG(status)取得引发子进程暂停的信号代码, 一般会先用WIFSTOPPED 来判断后才使用此宏。

返回值：

如果执行成功则返回子进程识别码(PID), 如果有错误发生则返回-1。失败原因存于errno中。

#### 7．bz**ero**

定义：

void bzero(void \*s,int n);

表头文件：

#include <string.h>

说明：

bzero()会将参数s所指的内存区域前n个字节, 全部设为零值。相当于调用memset((void\*)s,0,size\_tn);

#### 8.strcpy(),strcat()等字符串处理函数

略

#### 9.getpid

定义：

pid\_t getpid(void);

表头文件：

#include<unistd.h>

说明：

getpid()用来取得目前进程的进程识别码, 许多程序利用取到的此值来建立临时文件, 以避免临时文件相同带来的问题。

返回值：

目前进程的进程识别码

#### 10.getupid

定义：

uid\_t getuid(void);

表头文件：

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

说明：

getuid()用来取得执行目前进程的用户识别码。

返回值：

用户识别码

#### 11.raise

kill和raise都是用来发送信号的:

kill把信号发送给进程或进程组;

raise把信号发送给(进程)自身.

#### 12.atoi

定义：

int atoi(const char \*nptr);

表头文件：

#include <stdlib.h>

说明：

atoi()会扫描参数nptr字符串, 跳过前面的空格字符, 直到遇上数字或正负符号才开始做转换, 而再遇到非数字或字符串结束时('\0')才结束转换, 并将结果返回。

返回值：

返回转换后的整型数。

## 二、源码的关键语句及行为分析

#### 1．SIGCHLD信号

简单的说，子进程退出时父进程会收到一个SIGCHLD信号，默认的处理是忽略这个信号，而常规的做法是在这个信号处理函数中调用wait函数族获取子进程的退出状态。本程序在收到SIGCHLD信号后，会执行waitpid(-1,&status,WNOHANG)获取子进程的退出状态。程序对退出状态status的判断和处理分析可以参考上面对waitpid函数的分析。

注意：本程序不同于上次的shell程序，所有作业的父进程都是job.c，而job.c本身是shell的子进程。这个作业调度应该是在模拟调度，对shell而言，job.c是一个用户程序，它本身也会被切换运行。

#### 2．newjob->pid=pid

do\_enq函数中，子进程和父进程都执行了newjob->pid=pid（pid是子进程的进程号），这样重复做的原因是：

由于进程执行顺序的不可测性，父进程fork之后不一定立刻切换到子进程，也就是没法确定子进程和父进程哪个先执行，这样在两个里面都进行同样的操作，可以保证无论谁先执行，这个操作都会正确的完成。

#### 3．Main函数FIFO的打开方式

语句：fifo=open("/tmp/server",O\_RDONLY|O\_NONBLOCK)；

O\_NONBLOCK表示以非阻塞模式打开管道，这个选择在对管道的读写时非常重要。

#### 4．定时处理函数sig\_handler分析

计时器所设的时间间隔是1s，这个1s调度的时间间隔，也就是每隔1s执行一次调度程序。这里要区别调度间隔和作业时间片这两个概念：调度间隔是作业调度内核执行的间隔，内核需要以较高的频率监控整个执行环境，处理发生的信号，更新队列等等，调度间隔是内核监视作业调度情况的间隔；而时间片是作业每次运行的时间，这个长度一般都是调度间隔的整数倍。

当子进程结束发送给父进程SIGCHLD信号时，sig\_handler函数只进行了一些必要的信息修正，并没有再次调用scheduler函数进行作业调度，原因就是因为作业调度间隔已经固定为了1s中，即使某个时间点子进程结束了，作业切换也是在下次SIGVTALRM信号到来之后才进行的。

## 三、源码中的问题

1．调度程序每次都只能从fifo中读出一个cmd，当入队作业数比较多时，会出现无法响应的情况。

解决：我们假定作业每秒发生一次，所以这个细节可以不进行处理的。

其实，只需要在schedule函数中用循环语句来读取fifo即可，这个工作可以在以后进行完善，不会有什么大的问题。

2．即使enq.c立刻入队，调度程序也是在SIGVTALRM信号到来之后才会进行处理。

解决：主调度程序是每隔1s中运行一次的，也就是每隔1s中处理一次。在这次模拟作业调度的程序中，是可以接受的。（这是助教说的~）

3．在一个运行job.c的终端，ctrl+Z再fg回来之后，调度行为异常。

解决：是的，这是助教唯一承认的一个bug。我没有想出很好的解决办法，毕竟job.c本身也是一个子进程，没有办法真正的来模拟作业调度程序。

4．如果只输入一个enq命令，该作业并不进行。而只有再输入一个enq命令，即再加入一个作业，作业调度才开始，第一个作业也才开始执行。

解决：这个问题困扰了我很久，最后在助教老师的帮助下顺利解决了。

在do\_enq函数体里pid=fork()语句前后添加调试信息，我们可以发现自fork之后父进程进行得快于子进程。父进程在执行完jobswitch()里的kill(current->job->pid,SIGCONT)之后，子进程才执行到raise(SIGSTOP)语句，这就导致了顺序的错误。因此第一个作业并不执行。而当加入第二个作业后，父进程执行了两次切换作业的语句，此时第一个作业才开始执行（在此之前第二个作业也不执行）。

有上面的分析，bug已经很明确了。实际上就是在创建子进程后，父进程没有等待子进程。在本程序里，父进程的等待是必须的。当子进程执行raise(SIGSTOP)后，父进程再继续进行，就解决了这个bug。即只要在父进程的位置加上wait(NULL);即可。

这里wait(NULL);的意思是只需要父进程等待子进程退出，但对退出状态不关心。

## 四、提高要求

采用多级轮转反馈调度算法。

三个优先级：1，2，3，分别对应3个队列，运行时间分别为5s，2s，1s。

高优先级队列没有执行完毕的话，不会去轮转低优先级队列。

低优先级队列等待时间超过10s会升级，从而得到运行。

新加入的作业优先级高于当前作业的话，会进行抢占。

## 五、一些感悟

1．并发程序执行时序的不确定性

第一个地方就是多进程运行的时候，进行执行顺序的重要性。而且这个错误通常很难被发现，因为进程的执行顺序是不可预测的，这就要求我们在写这样的并发程序的时候，尤其需要关注这种时序问题，并且充分利用号信号量来使得顺序可控。

2．指针虽好，却容易滥用

这个调度程序当中使用了大量的指针，尤其是提高要求的时候，设置了三个链表队列，整个程序到处都充斥了指针操作。

刚开始的时候，感觉C语言的指针真是太好用了，方便简单，还省事。

是的，写程序的时候确实省事啊，但是写完之后出错也是so easy啊！可以这么说吧，我们组用了大约6个小时的时间完成了提高要求的代码，然后，至少花了两倍的时间进行调试才使得程序运行正常。在这近乎让人崩溃的调试，所有找到的错误，全部都与指针有关。感觉，这真是一种讽刺！

通过函数调用传递一个指针，改变指针指向的内容，原函数也会跟着变化，但如果改变的是指针本身的内容，那原函数是肯定不会变化的。这应该是指针最常犯的错误吧。

所以，如果传指针的话，能不用子函数就不要用子函数了，调用的层数越多，指针越不可控。