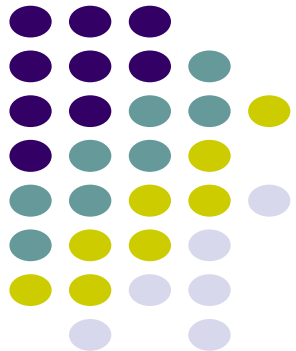
操作系统1实验

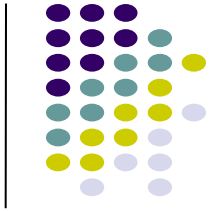
进程控制与通信



进程操作一：LINUX的进程控制

- LINUX的进程控制的系统调用：
  - fork()： 创建一个新进程
  - exec(): 执行一个可执行程序
  - exit()： 终止
  - sleep()：暂停一段时间
  - pause()：暂停并等待信号
  - wait()：等待子进程暂停或终止
  - signal：发送信号
  - kill()　：　发送信号到某个或一组进程
  - ptrace()：设置执行断点(breakpoint)，允许父进程控制子进程的运行

fork()—创建一个新进程



- fork()调用格式： pid = fork()
- 在调用fork（）之后，父进程和子进程均在下一条语句上继续运行。可以检查fork（）返回值，同时进程可以决定下面要执行的代码。
- 父、子进程的fork()返回值不同
  - 在子进程中返回时，pid为0；
  - 在父进程中返回时，pid为所创建的子进程的标识。

fork()—创建一个新进程

- fork()创建子进程之后，执行返回父进程，或调度切换到子进程或其他进程。
- fork创建一个新进程（子进程），除了子进程标识符和其PCB结构中的某些特性参数不同之外，子进程是父进程的精确复制。
- 父、子进程的运行是无关的，所以运行顺序也不固定。若要求父子进程运行顺序一定，则要用到进程间的通信。

父进程

```
main()
{
   pid_t pid;
   printf("PID…);
   pid=fork();
  if(pid!=0)
     printf("parent…");
  else
     printf("child…);
}
```

执行分裂

父进程

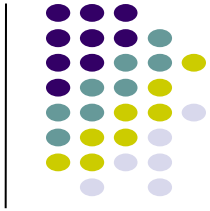
```
main( )
{
   pid_t pid;
   printf("PID…);
   pid=fork();
  if(pid!=0)
     printf("parent…");
  else
     printf("child…);
}
```

继续执行

子进程

```
main( )
{
   pid_t pid;
   printf("PID…);
   pid=fork();
  if(pid!=0)
     printf("parent…");
  else
     printf("child…);
}
```

继续执行



fork()调用例子（1）

```
void main(void) {
  printf(“Hello \n”);
  fork();
  printf(“Bye \n”）;
}
```

运行结果

Hello

Bye

Bye

fork()调用例子（2）

```
void main(void){
  if (fork()==0)
     printf("In the CHILD process\n");
 else
    printf("In the PARENT process\n");
}
```

程序的运行无法保证输出顺序，如果多次运行该程序，有时语句“In the CHILD process” 会在In the PARENT process”之前，有时却相反。

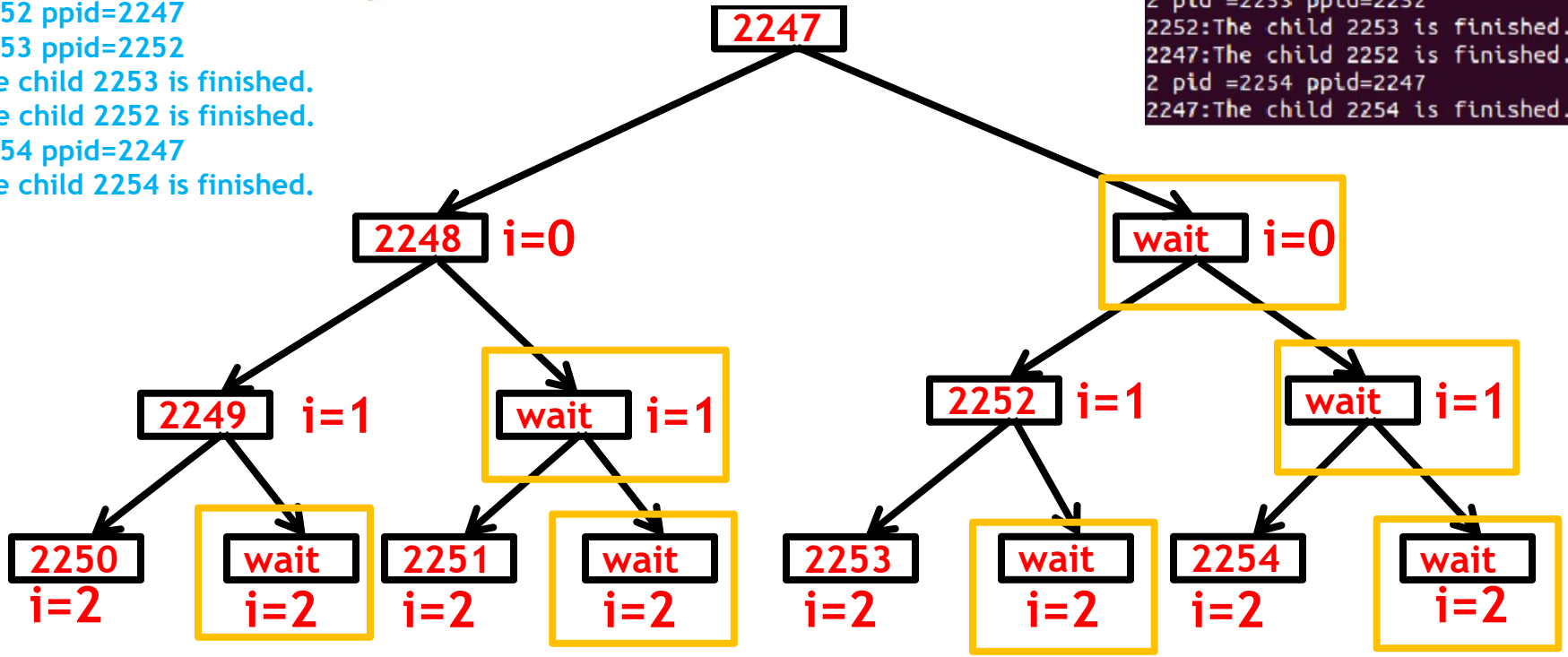
例题

```
#include<unistd.h>
main()
{
int i;
printf(“My pid is %d, my father’s pid is %d\n”,getpid(),getppid());
for(i=0; i<3; i++)
  if(fork()==0)
    printf(“%d pid=%d ppid=%d\n”, i,getpid(),getppid());
  else
  {
    j=wait(0);
    Printf(“ %d:The chile %d is finished.\n” ,getpid(),j);
  }
}
```

```
0 pid =2248 ppid=2247
1 pid =2249 ppid=2248
2 pid =2250 ppid=2249
2249: The child 2250 is fintshed.
2248: The child 2249 is finished.
2 pid =2251 ppid-2248
2248: The child 2251 is finished.
2247: The child 2248 ts finished.
1 pid =2252 ppid=2247
2 pid =2253 ppid=2252
2252: The child 2253 is finished.
2247: The child 2252 is finished.
2 pid =2254 ppid=2247
2247: The child 2254 is finished.
```

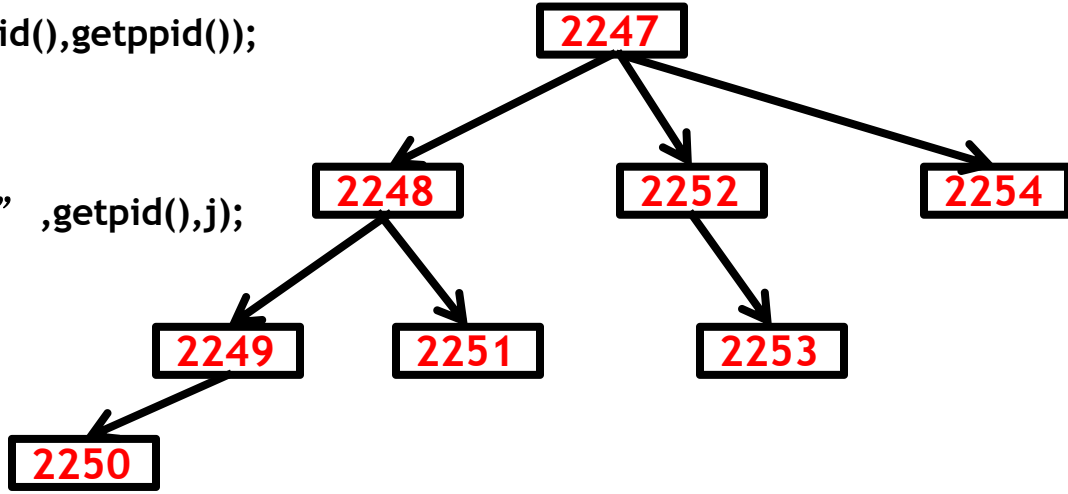
```
for(i=0; i<3; i++)
  if(fork()==0)
    printf( "%d pid=%d ppid=%d\n" , i,getpid(),getppid());
  else
  {
    j=wait(0);
    Printf( " %d:The chile %d is finished.\n" ,getpid(),j);
  }
}
```

```
april@april-VirtualBox:~/homewor
My pid is 2247,my father's pid i
0 pid =2248 ppid=2247
1 pid =2249 ppid=2248
2 pid =2250 ppid=2249
2249:The child 2250 is finished.
2248:The child 2249 is finished.
2 pid =2251 ppid=2248
2248:The child 2251 is finished.
2247:The child 2248 is finished.
1 pid =2252 ppid=2247
2 pid =2253 ppid=2252
2252:The child 2253 is finished.
2247:The child 2252 is finished.
2 pid =2254 ppid=2247
2247:The child 2254 is finished.
```

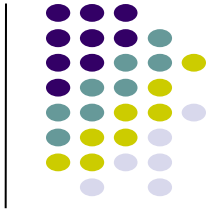


例题

```
#include<unistd.h>
main()
{
int i;
printf( “My pid is %d, my father’ s pid is %d\n” ,getpid(),getppid());
for(i=0; i<3; i++)
  if(fork()==0)
    printf( “%d pid=%d ppid=%d\n” , i,getpid(),getppid());
  else
  {
    j=wait(0);
    Printf( “ %d:The chile %d is finished.\n” ,getpid(),j);
  }
}
```



exec()-执行一个文件的调用



- 子进程如何执行一个新的程序文本？
  - 通过exec()函数簇，加载新的程序文本
- 通过一个系统调用exec，子进程可以拥有自己的可执行代码。即exec用一个新进程覆盖调用进程，但进程的PID保持不变
- 它的参数包括新进程对应的文件和命令行参数。成功调用时，不再返回；否则，返回出错原因。
- 六种exec调用格式：各种调用的区别在于参数的处理方法不同，常用的格式有：
  - execvp(filename, arg[ ])；
  - execlp(filename,arg0,arg1,...,(char *)0)；

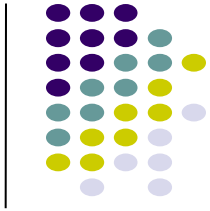
exec()说明

所需头文件	#include <unistd.h>
函数说明	执行文件
函数原型	int execl(const char *pathname, const char *arg, ...)
	int execv(const char *pathname, char *const argv[])
	int execle(const char *pathname, const char *arg, ..., char *const envp[])
	int execve(const char *pathname, char *const argv[], char *const envp[])
	int execlp(const char *filename, const char *arg, ...)
	int execvp(const char *filename, char *const argv[])
函数返回值	成功：函数不会返回
	出错：返回-1，失败原因记录在error中

exec()查找方式

- 查找方式：上表其中前4个函数execl,execv,execle,execve的查找方式都是完整的文件目录路径（pathname），而最后2个execlp,execvp函数（也就是以p结尾的两个函数）可以只给出文件名，系统就会自动从环境变量“$PATH”所指出的路径中进行查找。

exec()参数传递



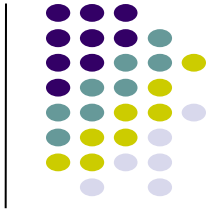
- 参数传递方式：exec函数族的参数传递有两种方式，一种是逐个列举(l)的方式，而另一种则是将所有参数整体构造成指针数组(v)进行传递。在这里参数传递方式是以函数名的第5位字母来区分的，字母为“l”（list）的表示逐个列举的方式，字母为“v”（vertor）的表示将所有参数整体构造成指针数组传递，然后将该数组的首地址当做参数传给它，数组中的最后一个指针要求是NULL。

exec()查找方式

- execl("/bin/ls","ls",NULL)
- execvp("ls",arg)
  - char *arg[] = {"ls",NULL};
- execlp("ls","ls",NULL)

第4位	统一为：exec	
第5位	l：参数传递为逐个列举方式	execl、execle、execlp
	v：参数传递为构造指针数组方式	execv、execve、execvp
第6位	e：可传递新进程环境变量	execle、execve
	p：可执行文件查找方式为文件名	execlp、execvp

僵尸进程

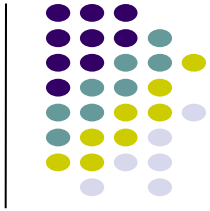


- 一个进程使用 fork 创建子进程，如果子进程退出而父进程并没有调用 wait() 或者 waitpid() 获取子进程信息，那么子进程的描述符仍然保存在系统中。这种进程就被称为僵尸进程。
- 原因：在每个进程退出的时候, 内核仍然为其保留一定的信息(包括进程号,退出状态,运行时间等)。直到父进程通过wait / waitpid来取时才释放。
  - 僵尸进程会占用资源，导致其他资源无法使用
  - 通过kill发送SIGTERM或者SIGKILL信号杀掉父进程

孤儿进程

- 如果父进程退出而它的一个或多个子进程还在运行，那么这些子进程就被称为孤儿进程。孤儿进程最终将被 init 进程 (进程号为 1 的 init进程) 所收养并由 init 进程完成对它们的状态收集工作。
- 孤儿进程不会产生危害

wait()



- 进程一旦调用了wait，就立即阻塞自己，由wait自动分析是否当前进程的某个子进程已经退出，如果让它找到了这样一个已经变成僵尸的子进程， wait就会收集这个子进程的信息，并把它彻底销毁后返回；如果没有找到这样一个子进程，wait就会一直阻塞在这里，直到有一个出现为止。
  - #include <sys/types.h> /* 提供类型pid_t的定义 */
  - #include <sys/wait.h>
  - pid_t wait(int *status);
- 参数status用来保存被收集进程退出时的一些状态

wait()

- 如果我们对这个子进程是如何死掉的毫不在意，只想把这个僵尸进程消灭掉，（事实上绝大多数情况下，我们都会这样想），我们就可以设定这个参数为NULL
  - pid = wait(NULL);
- 如果成功，wait会返回被收集的子进程的进程ID，如果调用进程没有子进程，调用就会失败，此时wait返回-1。

exit()—进程终止

- 进程终止：Unix/Linux中系统调用exit()
- 进程终止时：返回数据到其父进程，它相关所有资源：内存、打开文件和I/O缓冲会被释放掉。
- 父进程可以通过wait（）等待子进程的结束，wait（）可以将子进程的标识符返回给父进程

软中断信号

- 软中断是利用硬件中断的概念，用软件方式进行模拟，实现宏观上的异步执行效果。
- 利用signal和kill实现软中断通信
- kill(pid,signal)：向进程pid发送信号signal

```
1  #include<sys/types.h>
2  #include<signal.h>      //头文件
3  int kill(pid_t pid, int sig)        //函数原型
```

- signal(sig,func)：设置sig号软中断信号的处理方式

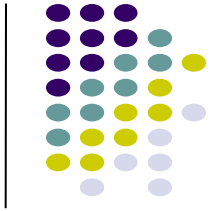
```
1  #include<signal.h>      //头文件
2  void (*signal(int sig, void (*func)(int)))(int)      //函数原型
```

signal处理方式

```
1  #include<signal.h>        //头文件
2  void (*signal(int sig, void (*func)(int)))(int)      //函数原型
```

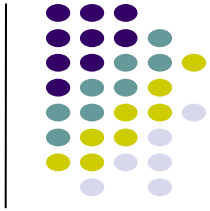
- ① SIG_IGN：忽略参数所指的信号
- ② 一个自定义的处理信号的函数，信号的编号为这个自定义函数的参数。
- ③ SIG_DFL：恢复为默认值，一般为终止进程。

常用linux命令



- ls [-alrtAFR] [name...]
- ps、pstree
- cat
- chmod
- ln
- pwd
- echo
- cd
- 等等，参考实验指导书

进程操作二：管道通信

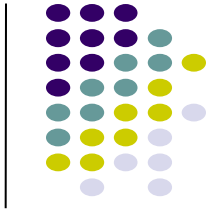


- 管道的基本定义
  - 管道就是传输信息或数据的工具
  - 某一时刻只能单一方向传递数据，不能双向传递数据（半双工模式）
- 管道创建和管道关闭
  - 管道由Linux系统提供的pipe()函数创建

```
1 #include <unistd.h>
2 int pipe(int filedes[2]);
```

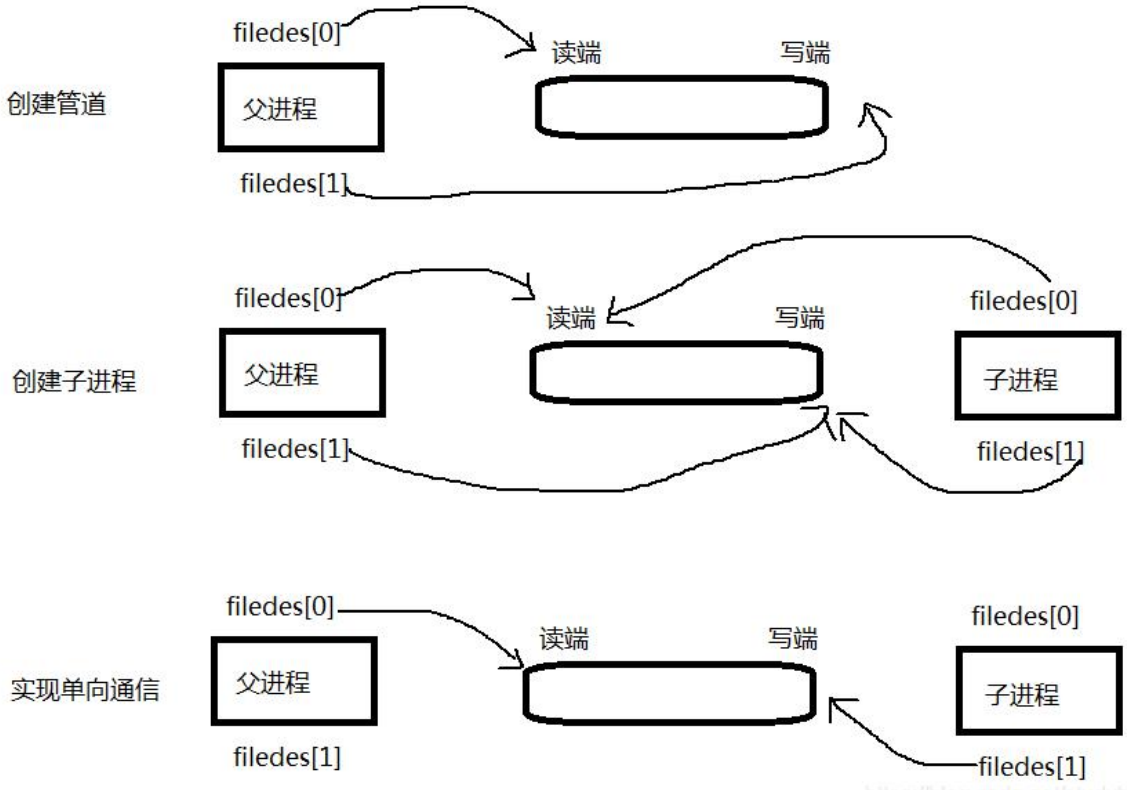
  - pipe()函数用于在内核中创建一个管道，该管道一端用于读取管道中的数据，另一端用于将数据写入管道
  - filedes[0]指向管道的读端，filedes[1]指向写端

管道通信



- pipe()函数实现管道通信
  - （1）在父进程中调用pipe()函数创建一个管道，产生一个文件描述符filedes[0]指向管道的读端和另一个文件描述符filedes[1]指向管道的写端。
  - （2）在父进程中调用fork()函数创建一个一模一样的新进程，也就是所谓的子进程。父进程的文件描述符指向读端，子进程的文件描述符指向写端。。
  - （3）此时，就可以将子进程中的某个数据写入到管道，然后在父进程中，将此数据读出来。

管道通信



管道特点

- 管道通常指的是无名管道，只能用于具有共同祖先的进程（具有亲缘关系的进程）之间进行通信；通常，一个管道由一个进程创建，然后该进程调用fork()，此后父子进程之间就可以应用该管道。
- 一般而言，进程退出，管道释放，所以管道的生命周期跟随进程。
- 一般而言，内核会对管道操作进行同步与互斥
- 管道是半双工的，数据只能向一个方向流动；需要双方通信时，需要建立起两个管道。