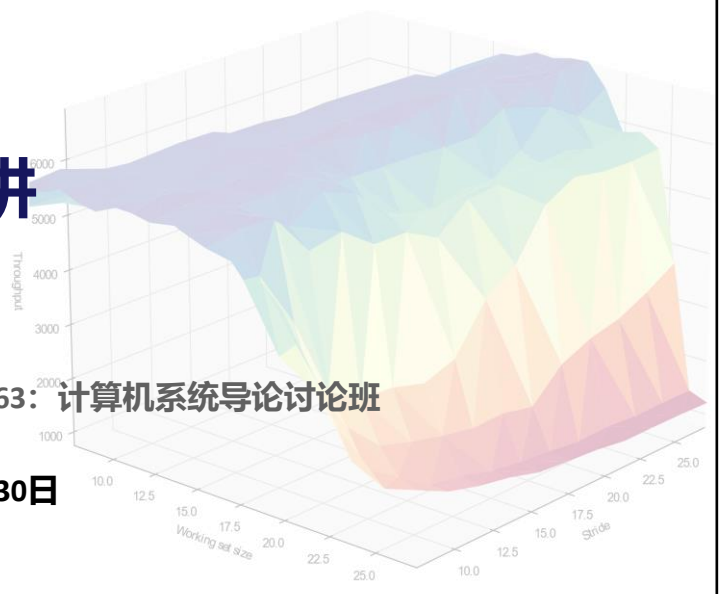
第八讲



PKU 04832363：计算机系统导论讨论班

王畅

2021年11月30日

Activity 0

复习讲座课+动态链接

PIC和延迟绑定的过程（可能要考！）

- **基本原理**
  - 相关全局函数和全局变量的位置都需要从GOT表中查。GOT一个条目8字节，前三个条目是动态链接器需要占用的。
- **延迟绑定**
  - 为了减少动态链接的开销，函数第一次用到时才进行重定位。
  - 用到PLT表，PLT表要能被执行，所以在代码段。PLT表一个条目16字节（实际上是几条指令），前三个条目是动态链接器需要占用的（课本声称为2个）。
  - 过程：先跳入PLT表（foo@plt），第一条指令会根据foo对应的GOT条目，查出目标并跳过去。然而，还没重定位时GOT表会填这条指令的下一条指令，这条指令会触发进入PLT的开头，调用动态链接器填写真正的GOT条目。接下来执行foo，foo@plt的第一条指令就间接跳转到foo去执行。
- **感兴趣的同学可以读下《程序员的自我修养》，7.3节**

3

具体地说，每个动态链接的函数foo都对应一个GOT表条目和一个PLT条目，不论何时调用foo，都是先call foo@plt（就是到foo的PLT条目里去）。

这个PLT条目的第一句指令是jmp *foo@GOT（就是到foo的GOT条目中查出foo真正的位置，然后跳过去）。

在第一次调用foo时，foo@GOT填写的是foo@PLT的第二条指令的地址，所以第一次绕了一大圈还是顺序执行。第二条指令开始会涉及一系列指令，将foo的GOT表真正填好（即将其填成其真正位置，一般会先把一些相关数据推栈，然后跳到PLT表的开头进入动态链接器），此后再调用foo@PLT，第一句就会直接跳到真正的foo位置了。

Activity 1

轮流回答问题

书本内容梳理

- **信号是软件形式的异常**
- **读图8-26，了解常见的信号，默认行为和触发原因**
  - SIGHUP
  - SIGINT
  - SIGKILL
  - SIGSEGV
  - SEGALRAM
  - SIGTSTP
  - SIGCHLD
  - SIGFPE
  - SIGSTOP
  - SIGUSR1
  - SIGUSR2

5

注意SIGTSTP和SIGSTOP的区别，前者是终端发的stop，后者是非终端发的stop。以上列出的信号都是很常见的，应当熟悉含义。

考点1：信号发送、接收、处理、阻塞

- **发送：可以发送给自己（kill函数通用，alarm特别）**
- **接收：可以忽略，终止或者执行一个信号处理程序**
  - 惟SIGKILL、SIGSTOP（不要和SIGTSTP弄混了）不能修改默认行为
  - 什么时候处理？内核模式返回用户模式的时候
- **待处理：等待接收。一种类型至多有一个，后来的被丢弃；在pending位向量中维护，接收时被清除**
  - 推论：信号不能用来计数
- **阻塞：选择性地不接收，仍可以发送；在blocked位向量中维护**
  - 隐式阻塞：处理某个信号时，相同类型信号会被阻塞
  - 显式阻塞：sigprocmask函数（熟悉用法）
- **Section 15 练习4、9、10、11、2**

6

问题：一般是否可能出现程序从不进入内核模式，因而就不会处理信号的问题？回答：不会。从理论上说，如果信号来自于另一个进程，那么之后执行本进程必然发生上下文切换，所以一定处理信号；如果信号是发给自己的，由于发送信号是系统调用，所以返回后也会处理信号。而且从实际上讲，系统调度进程，当其CPU时间用完时，会给进程发时钟中断，进入内核模式，并切换上下文。由于时钟一般很短，所以不会出现不处理信号的问题。

另外注意隐式阻塞的概念。

轮流回答问题

- **设一段程序中阻塞了SIGCHLD、SIGUSR1信号。接下来，向它按顺序发送SIGCHLD、SIGUSR1、SIGCHLD信号，当程序取消阻塞继续执行时，将处理这三个信号中的哪几个?**

前两个，因为信号是不排队的，最后一个SIGCHLD被丢弃。

轮流回答问题

- **判断下列说法正确性：**
  - SIGTSTP信号既不能被捕获，也不能被忽略
  - 存在信号的默认处理行为是进程停止直到被SIGCONT信号重启
  - 系统调用不能被中断，因为那是操作系统的工作
  - 在任何时刻，一种类型至多只会有一个待处理信号
  - 信号既可以发送给一个进程，也可以发送给一个进程组
  - SIGTERM和SIGKILL信号既不能被捕获，也不能被忽略
  - 当进程在前台运行时键入Ctrl-C，内核就会发一个SIGINT信号给这个前台进程
  - 子进程能给父进程发送信号，但不能发送给兄弟进程

8

错，混淆了SIGTSTP和SIGSTOP

对，有很多停止性质的信号都是这样，比如SIGSTOP、SIGTSTP

错，有一些系统调用不是原子的

对，信号不排队

对，例如kill中，pid写成负数，就是表示给|pid|所在的进程组发信号

错，SIGTERM可以自定义，是SIGSTOP不能改变行为（例如不能忽略）

对

错，只要知道了PID或相关信息就可以发信号

考点2：结合其他API的fork puzzles

- **几个技巧**
  - 需要`wait`时，要在关系图中，子进程结束前的最后一个输出向父进程`wait`后的第一个输出连一条有向边，然后再拓扑排序。
  - `signal`打断时间随机（除了屏蔽的时候），因此本质上是一个“半悬空”的结点。如果有打印的话可以先排列再插`signal`（根据`signal`本身的性质，插可以插的地方）。`signal`中有操作的，可以先正常排列，再插`signal`并更新`signal`的影响。
  - 条件判断中有`fork`：改写成`if+goto`的语句，比较复杂的就不要直接“瞪眼”分析。
- **Section 14 6、7**
- **Section 15 7、15、18、28**

9

一般只有遇到比较麻烦的题目才用一些技巧，不必滥用。

注意：一些代码看上去比较复杂，实际上很容易分析，不要被吓到了！

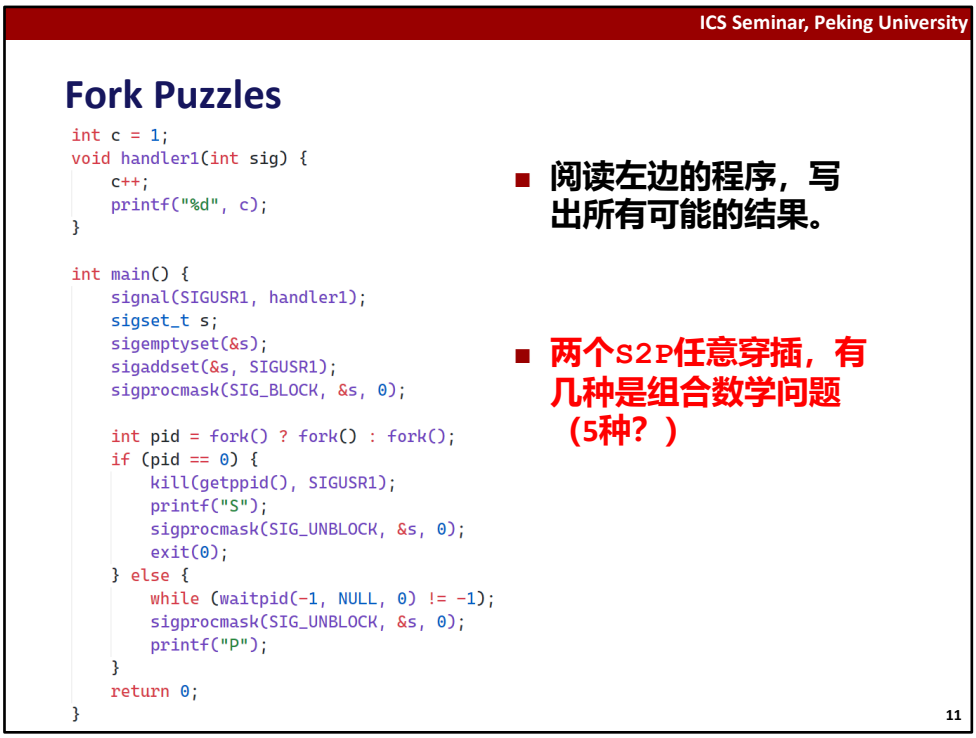
Fork Puzzles

```c
void handler() {
    printf("D\n");
    return;
}
int main() {
    signal(SIGCHLD, handler);
    if (fork() > 0) {
        printf("A\n");
    } else {
        printf("B\n");
    }
    printf("C\n");
    exit(0);
}
```

- **阅读左边的程序，下列哪些输出是可能的?**

- **ACBC** **Y**
- **ABCCD** **Y**
- **ACBDC** **N**
- **ABDCC** **N**
- **BCDAC** **Y**
- **ABCC** **Y**

主进程打印AC，子进程打印BC，而D必须在BC之后出现，但可以在A之前出现，可以最后出现，也可能不出现（可能父进程在子进程结束之前就结束了）。具体分析D是一个“悬空”结点，有一条边的连法有多种可能。



fork():fork()?fork()实际上非常简单，就是产生了4个进程，进程树两边是对称的。每边打印的字符顺序是严格的（注意sigprocmask有系统调用，结束后是从内核模式回到用户模式，信号都会被正常接收，不会错过）：父进程需要等子进程，是S2P，所以穿插即可。枚举知道至少有5个答案：SS22PP、S2PS2P、SS2P2P、S2S2PP、S2SP2P。

并发安全

- **请仔细阅读课本的shell框架，学习优良的代码范式**
- **阅读课本P533-546，通过shell lab学习**

- **内容较多，但主要原理（包括今后的基本原理）就两个**
  - 一、信号处理程序是一种“随时打断”式的程序，其内部最好不要被中断，而其他普通代码很多不能保证时刻正确响应信号处理程序的存在。（有关阻塞信号、全局共享数据的处理、volatile关键字、可重入函数或原子的的概念）
  - 二、进程调度顺序是不确定的，不能有任何假设，否则可能有问题。因此必须用阻塞、同步等方式防止出现racing（冒险/竞争）。（有关sigsuspend、阻塞等）

12

第一个原理举例：假设编译器将全局变量x安排在寄存器中，而信号处理程序需要修改x，那么在进入信号处理程序时，需要写回x；从信号处理程序返回时，必须有一条指令从内存中读回x，但因为信号处理程序打断位置不确定，编译器无法保证x的值正确。

第二个原理举例：假设某两个程序同时需要给某个共享全局变量+1，+1的步骤分为：读出x，给x+1，写回x。结果，第一个程序读出x之后被打断，另一个程序也读出x，最后导致虽然加了两次，但是只加了1，结果错误。需要同步。再举一个例子：假如父进程要给子进程发信号，子进程在信号处理程序中做有意义的事情，那么子进程需要**等待信号**，因此也必须同步（sigsuspend），否则可能过早退出。通常这种同步问题彻底解决至少需要一条原子语句的介入，否则理论上会发生无限打断的情况。

书本内容梳理

- **longjmp从env中恢复调用环境，然后从最近的一次setjmp中返回，它的第二个参数会成为setjmp此次的返回值**
- **setjmp的返回值不能赋值给变量，但是可以用于switch**
  - 哪些函数调用一次，返回多次?
  - longjmp的“返回值”（它实际永不返回，这里指回到setjmp后的返回值）永远不会是0
- **思考题：用非本地跳转设计try catch块**

13

longjmp的第二个参数如果填0，会被改成1，确保setjmp的这种返回能和初次设置的返回能够相互区分。setjump的值也可以赋给if、while等语句。

非本地跳转=>异常处理

```
#define TRY do{ jmp_buf ex_buf__; if( !setjmp(ex_buf__) ){
#define CATCH } else {
#define ETRY } }while(0)
#define THROW longjmp(ex_buf__, 1)

int main(int argc, char** argv) {
    TRY {
        printf("In Try Statement\n");
        THROW;
        printf("I do not appear\n");
    }
    CATCH
    {
      printf("Got Exception!\n");
    }
    ETRY;
    return 0;
}
```


注意C++中的try-catch并不是这样实现的。现时setjmp-longjmp用得并不是非常多了。

为什么setjmp不能随意赋值？（了解）

- **setjmp/longjmp是非常危险的（和丑陋的）**
  - C标准要求setjmp以宏实现（主要是内嵌汇编），但POSIX标准没有确定是函数还是宏
  - 它们的实现基于保存部分上下文
    - 即返回到setjmp时，会恢复上一次setjmp后的寄存器状态，特别地，包括**栈指针和程序计数器**，但不修改栈的具体内容
  - longjmp后返回到setjmp的位置，因此setjmp和longjmp之间（含setjmp本身）不能访问任何没有volatile限定的局部变量，否则为未定义行为
  - 如果setjmp返回赋值给了局部变量，那这个局部变量既要被restore，又要被“返回”，怎么做？未定义。总之是要返回后的表达式足够简单。
- **具体可参看The Linux Programming Interface, 6.8节**

15

为什么会为未定义行为？因为中间访问的局部变量如果修改，后来longjmp回去会被恢复。然而考虑到这一点的“特别聪明的”编译器可能就不会覆盖这个寄存器。

These restrictions are specified because an implementation of setjmp() as a conventional function can't be guaranteed to have enough information to be able to save the values of all registers and temporary stack locations used in an enclosing expression so that they could then be correctly restored after a longjmp(). Thus, it is permitted to call setjmp() only inside expressions simple enough not to require temporary storage.（大致意思是将setjmp用于特别复杂的表达式或者返回给局部变量，这个过程需要用到额外的寄存器，但是又要恢复上下文，而且setjmp保存的信息不足，所以无法做这种事情。）

轮流回答问题

- **下面关于非局部跳转的描述，正确的是**
  - A. setjmp可以和siglongjmp使用同一个jmp_buf变量
  - B. setjmp必须放在main()函数中调用
  - C. 虽然longjmp通常不会出错，但仍然需要对其返回值进行出错判断
  - D. 在同一个函数中既可以出现setjmp，也可以出现longjmp

16

D。A不行，信号安全和不安全的函数不能共用全局变量。B没有这种说法。C longjmp不会返回。D正确，当然setjmp-longjmp的主要目的之一确实是在不同函数之间直接“跳转”（类似于一个“耍赖”的goto）。

系统级I/O的注意事项

- 刚刚助教回课已经讲过了

考点3：I/O函数的选择和使用

- 注意课本上的使用说明
- Section 15 2、6、13、20

轮流回答问题

- **假设某进程有且仅有五个已打开的文件描述符0~4，分别引用了五个不同的文件，尝试运行以下代码：**
  `dup2(3, 2); dup2(0, 3); dup2(1, 10); dup2(10, 4); dup2(4, 0);`。关于得到的结果，说法正确的是：

  A. 运行正常完成，现在有四个描述符引用同一个文件。
  B. 运行正常完成，现在进程共引用四个不同的文件。
  C. 由于试图从一个未打开的描述符进行复制，发生错误。
  D. 由于试图向一个未打开的描述符进行复制，发生错误。

19

A。注意API的顺序`dup2(old, new)`，所以`old`的指针会复制给`new`，`new`的指向的打开文件表条目会变成指向`old`的条目。

轮流回答问题

- **判断下列说法的正确性。**
  - 由于RIO包的健壮性，所以RIO中的函数都可以交叉调用。
  - 成功调用open函数后，一定返回一个不小于3的文件描述符。
  - 调用Unix I/O开销较大，标准I/O库使用缓冲区来加快I/O的速度。
  - 和描述符表一样，每个进程拥有独立的打开文件表。
  - 从网络套接字（socket）读取内容时，可以通过反复读的方式处理不足值问题，直到读完所需要的数量或遇到EOF为止。
  - 以O_RDWR方式打开文件后，文件会有两个指针，分别记录读文件的当前位置和写文件的当前位置。
  - 用read函数直接读取控制台输入的文本行，会自动在行末追加\0字符。
  - 使用dup2(4, 1)成功进行重定向后执行close(4)，会导致1号文件描述符也不可用。

错，是否有缓冲区的不能交叉调用，不然输入输出的顺序会不对

错，如果提前关闭了stdin、stdout、stderr，就会返回0、1、2等描述符

对，顺便提请注意缓冲区的各种概念

错，描述符表是独立的，但打开文件表系统只有一张

对，具体怎样“反复读”涉及网络协议的规定

错，无论用什么方法都只有一个指针，指针对应一个打开文件表条目

错，不会追加，需要自己在buf中添加\0

错，虽然1、4都指向打开文件表的同一条目，但是refcnt=2，因此关闭4之后refcnt=1，不会释放这个条目，1还是能继续用的，直到1被关闭，这个条目才被销毁

Activity 2

问题求解

考点4：结合fork、dup和打开文件的分析

- **主要考点：父子进程变量独立、文件读写指针位置、同步安全问题**
  - 画图模拟，理解打开文件表和独立文件描述符的指向关系
  - 标明读写指针的位置
    - 技巧：互相独立的指针读写序列独立，写的结果是最后写的那次得到的
  - 理解`fork`理解子进程是父进程的一个独立副本
  - 小心缓冲区的问题（有可能题目会不严谨）
  - 如有必要，结合拓扑排序
- **Section 14 8、9、10、11**
- **Section 15 3、5、14、19、21、26、27**

注意读写指针这个技巧。对于`dup2`处理过或者`fork`前后得到的描述符，它们的读写指针是同一个，可以认为是一回事；而不同的打开文件表条目，其指针相互独立，面对**比较复杂**的问题，可以先确定这些独立的指针都分别写了什么，然后再决定哪一次是最后一次写，最后一次写决定文件内容。

any questions?

Thanks & 感谢观看