**暨南大学本科实验报告专用纸**

课程名称 操作系统原理实验 成绩评定

实验项目名称 进程的状态和进程调度 指导教师 郝振明

实验项目编号 0806015704 实验项目类型 验证、设计 实验地点

学生姓名 张印琪 学号 2018051948

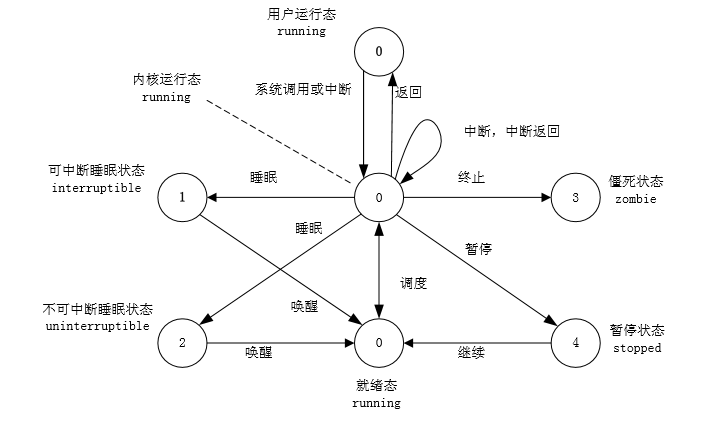
学院 信息科学技术学院 系 计算机科学系 专业 网络工程

实验时间 20XX 年 X 月 X日 上 午～ X 月 X 日 上 午

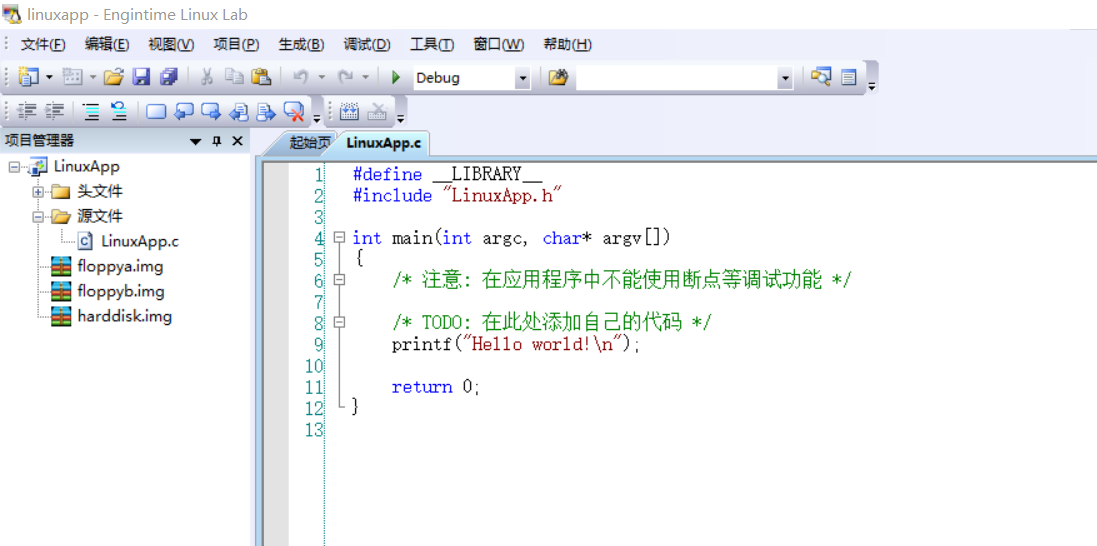
温度 ℃湿度

一、 实验目的  调试进程在各种状态间的转换过程，熟悉进程的状态和转换。  通过对进程运行轨迹的跟踪来形象化进程的状态和调度。  掌握 Linux 下的多进程编程技术。 二、 预备知识

一个进程在其整个生命周期内，不同阶段可能具有不同的进程状态（在 include/ linux/sched.h 文件的第 20 行定义）。一个进程的状态由其进程控制块中的 state 字段 指定。



 运行状态（TASK\_RUNNING）。当进程正在被 CPU 执行时，或已经准备就绪可由 调度程序调度时，则称该进程处于运行状态。若此时没有被 CPU 执行，则称其 处于就绪运行状态。进程可以在内核态运行，也可以在用户态运行。当一个进 程在内核代码中运行时，称其处于内核运行态，或简称内核态；当一个进程正 在执行用户自己的代码时，称其处于用户运行态，或简称用户态。当进程所需 的系统资源已经可用时，进程就会被唤醒，从而进入就绪运行状态。这些状态 在内核中表示方法相同，都被称为处于 TASK\_RUNNING 状态。例如，当一个新 进程刚刚被创建后就处于就绪运行态。

 可中断睡眠状态（TASK\_INTERRUPTIBLE）。当进程处于可中断睡眠状态时，系 统不会调度该进程执行。当系统产生一个中断或者释放了该进程正在等待的资 源，或者该进程收到一个信号时，该进程都会被唤醒，从而转换到就绪运行态。  不可中断睡眠状态（TASK\_UNINTERRUPTIBLE）。除了不会因为收到信号而被唤 醒外，该状态与可中断睡眠状态类似。但处于该状态的进程只有被 wake\_up 函数明确唤醒时才能转换到就绪运行状态。该状态常在进程需要不受干扰地等 待或者所等待事件很快就会发生时使用。  暂停状态（TASK\_STOPPED）。当进程收到信号 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 时就会进入暂停状态。可向处于暂停状态的进程发送 SIGCONT 信号， 让其转换到就绪运行状态。进程在调试期间接收到任何信号均会进入该状态。 在 Linux 0.11 中，还未实现对该状态的转换处理，处于该状态的进程将被作 为进程终止来处理。  僵死状态（TASK\_ZOMBIE）。当子进程已停止运行，但其父进程还没有调用 wait 函数询问其状态时，则称该进程处于僵死状态。因为，父进程需要获取子进程 停止运行的信息（例如获取子进程的退出码），所以，此时子进程的进程控制 块还需要保留。一旦父进程通过调用 wait 函数取得了子进程的信息，则处于 该状态的子进程的进程控制块就会被释放掉。 当一个进程的时间片用完时，操作系统会使用调度程序强制切换到其他的进程去执 行。另外，如果进程在内核态执行时需要等待系统的某个资源，此时该进程就会调用 sleep\_on 或 interruptible\_sleep\_on 函数自愿地放弃 CPU 的使用权，从而让调度程序 去执行其他进程，此进程则进入睡眠状态（TASK\_UNINTERRUPTIBLE 或 TASK\_INTERRUPTIBLE）。只有当进程从“内核运行态”转换到“睡眠状态”时，内核才 会进行进程切换操作。在内核态下运行的进程不能被其他进程抢占，而且一个进程不能 改变另一个进程的状态。为了避免进程切换时造成内核数据错误，内核在执行临界区代 码时会禁止一切中断。 内核中的调度程序 schedule（在文件 kernel/sched.c 中的第 135 行定义）用于选 择下一个要运行的进程。这种选择运行机制是多任务操作系统的基础。调度程序可以被 看作一个在所有处于运行状态的进程之间分配 CPU 运行时间的管理器。为了能有效地使 用系统资源，又能使各个进程有较快的响应时间，就需采用一定的调度策略，在 Linux 0.11 中采用了基于优先级排队的调度策略。 详细内容请读者阅读《Linux 内核完全注释》第 5 章第 7 节的内容。 三、 实验内容 3.1 准备实验 1. 启动 Engintime Linux Lab。 2. 新建一个 Linux011 应用程序项目。 

3.2 编写一个多进程程序 请读者按照下面的步骤使用 fork 函数编写一个多进程程序，一方面学习多进程编程技 术，另一方面，通过观察多个进程的运行过程，先从现象上对进程的调度过程有一个初步认 识。 1. 编辑 LinuxApp.c 文件中的 main 函数，让父进程新建三个子进程，并分别输出子进 程 id 以及其父进程 id。代码如下： int main( int argc, char \* argv[] ) if( 0 == fork() )

{

printf("child process pid=%d ppid=%d line=%d\n",

getpid(), getppid(), \_\_LINE\_\_);

}

else if( 0 == fork() )

{

printf("child process pid=%d ppid=%d line=%d\n",

getpid(), getppid(), \_\_LINE\_\_);

}

else if( 0 == fork() )

{

printf("child process pid=%d ppid=%d line=%d\n",

getpid(), getppid(), \_\_LINE\_\_);

}

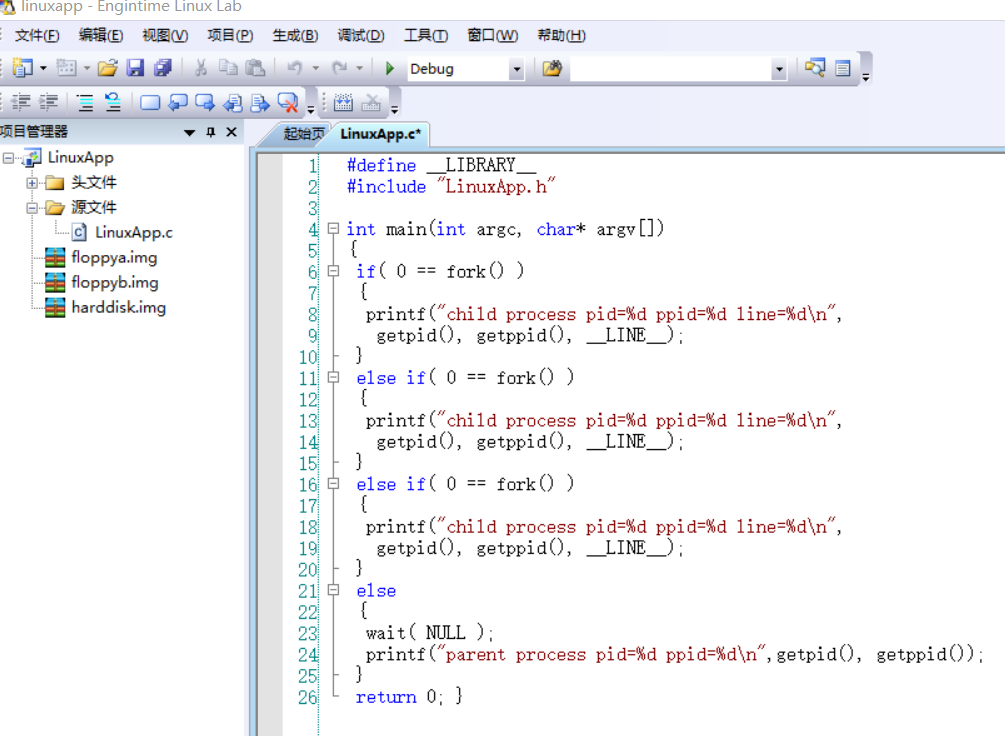
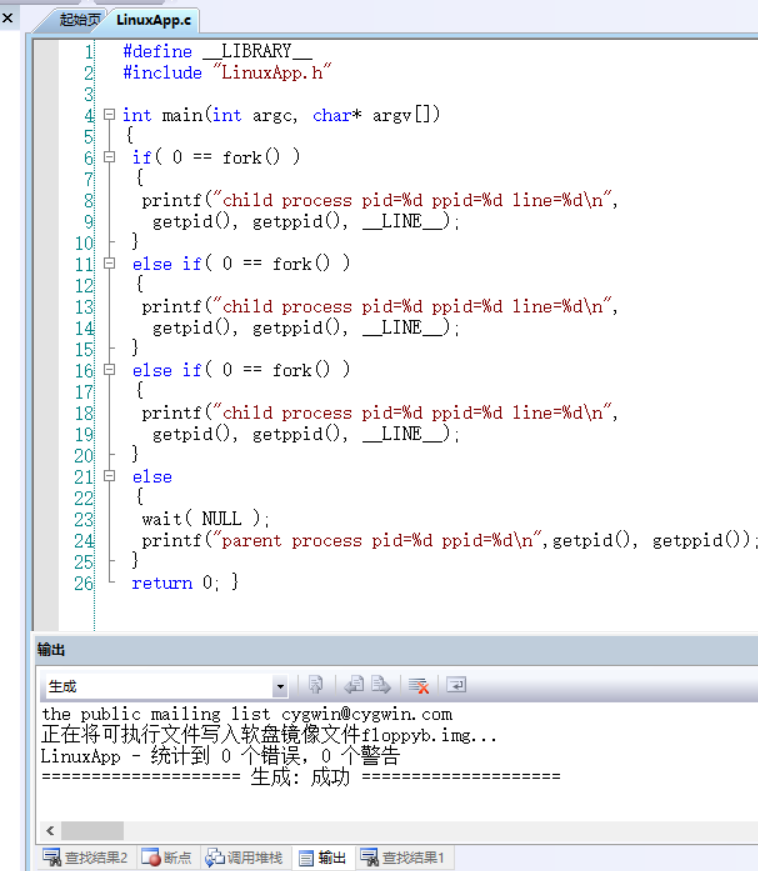
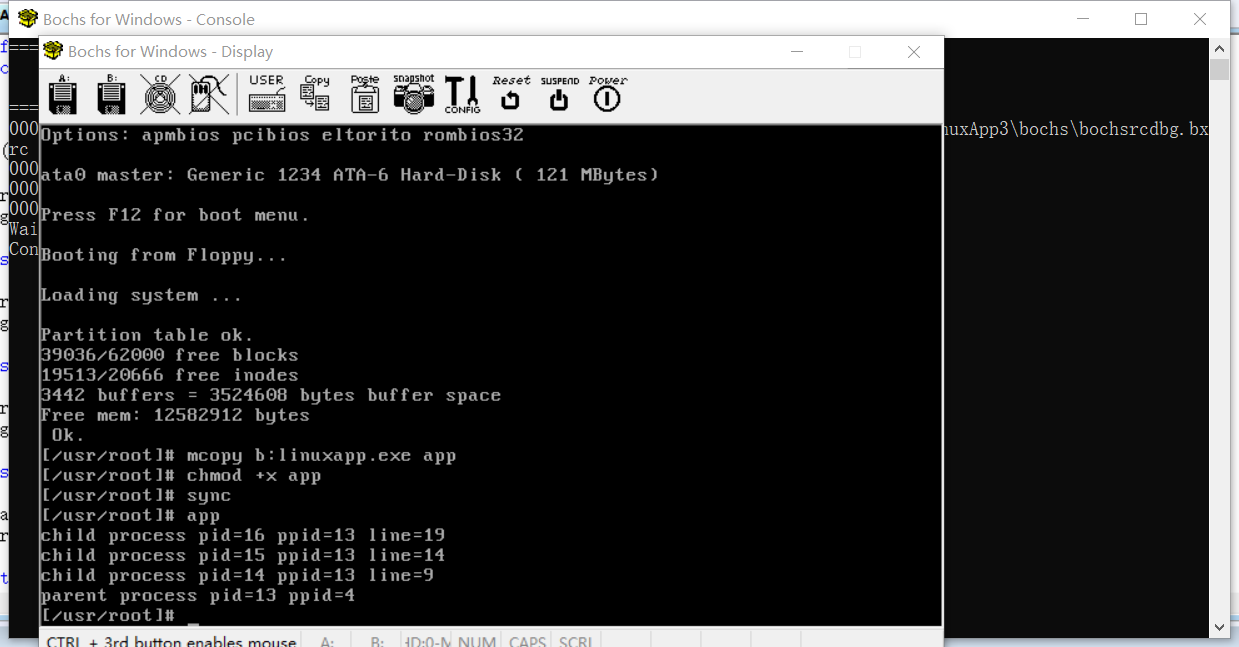
else

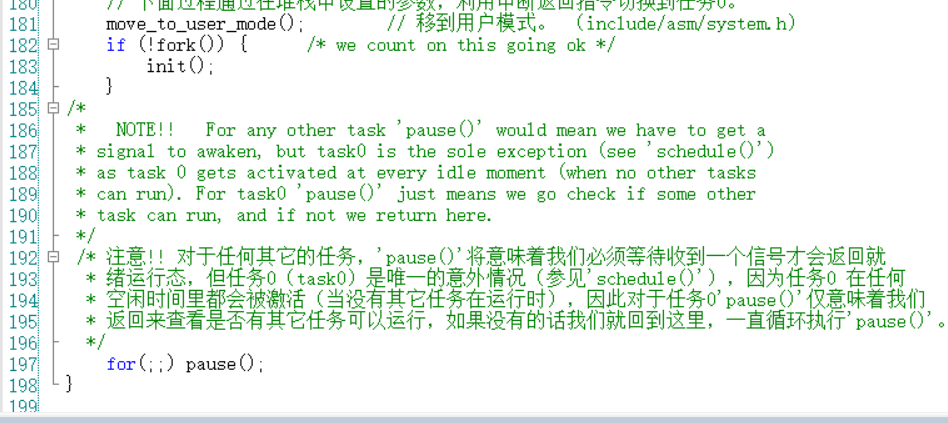
{

wait( NULL );

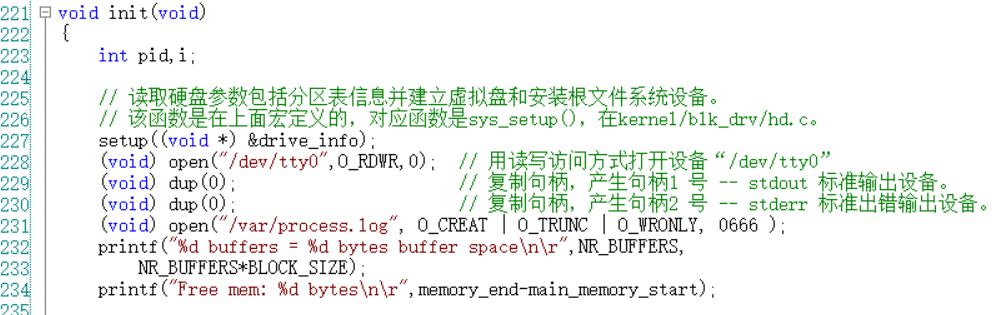
printf("parent process pid=%d ppid=%d\n",getpid(), getppid());

}

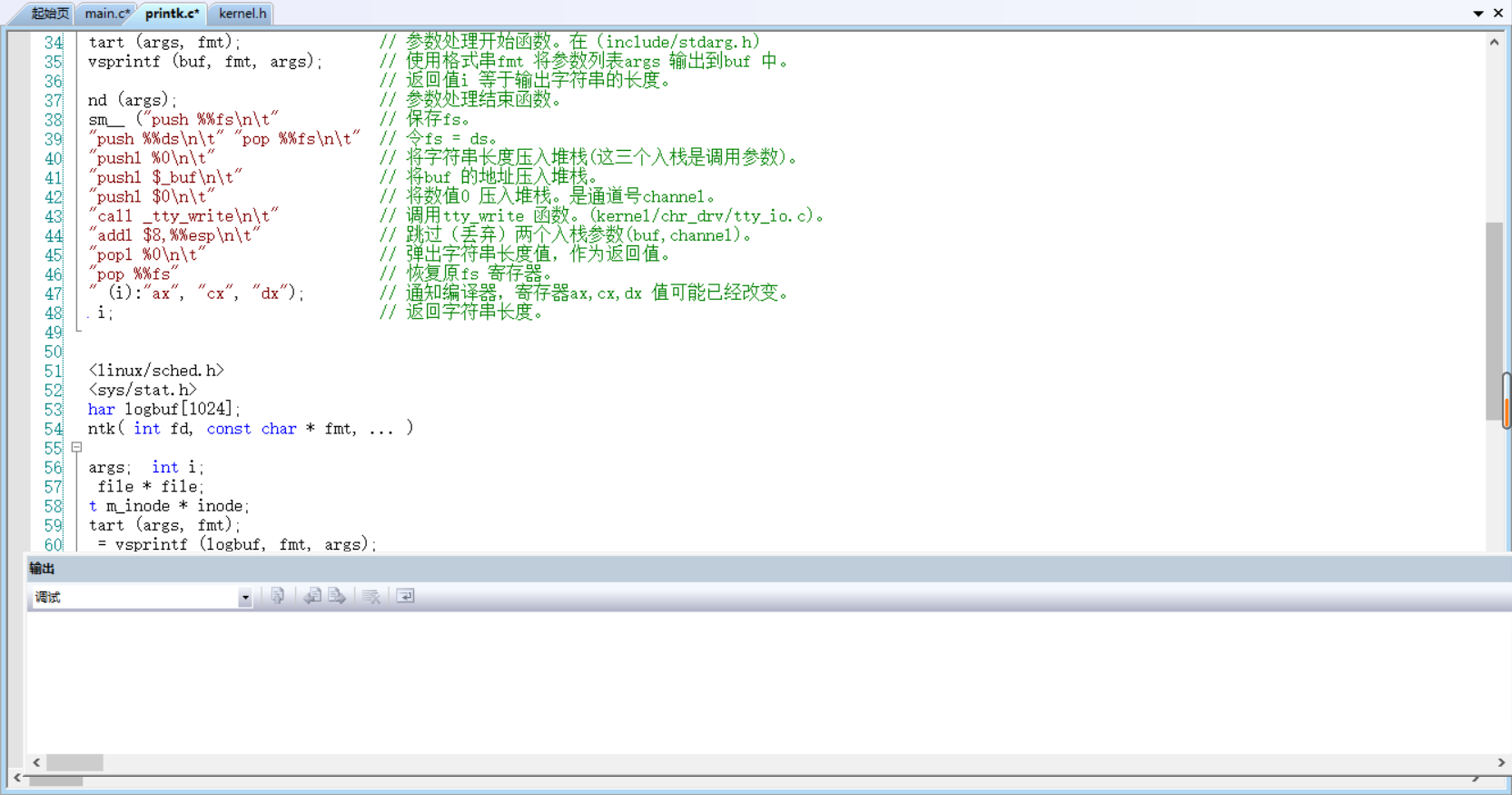
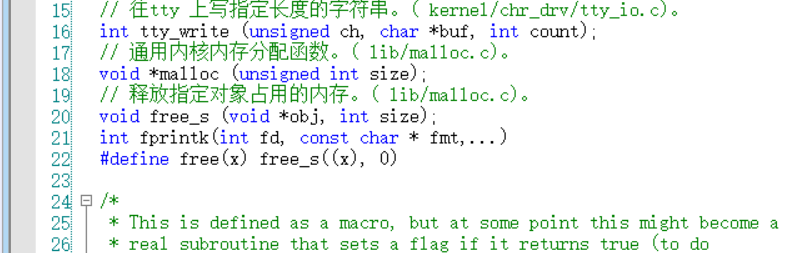
return 0; }  getppid 函数可以在子进程中获取父进程的 id。“\_\_LINE\_\_”是 GCC 编译器提供的 一个预定义宏，它的值是其所在的行号。 2. 按 F7 生成项目，按 F5 启动调试。 3. 将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下： mcopy b:linuxapp.exe app 4. 为 app 文件添加可执行权限，命令如下： chmod +x app 5. 执行“sync”命令。 6. 执行可执行文件 app，观察各个进程执行的顺序并说明原因，理解进程在生命周期 中状态的转换过程和进程调度过程。 7. 结束此次调试。

3.3 跟踪进程的运行轨迹 跟踪进程的运行轨迹，本质上就是跟踪并记录进程在其生命周期中的状态转换过程 和调度过程。进程在生命周期中状态的转换是由操作系统的调度程序实现的。所以要实 现对进程运行轨迹的跟踪，不仅要对进程在其生命周期的各个状态有一个全面的了解， 而且还要对进程的调度有一个透彻的理解。  在开始记录进程的运行轨迹之前，需要先在 Linux 0.11 内核中添加一个写日志文 件的功能，然后使用此功能将进程状态转换的日志信息写入一个文本文件。 3.3.1 在系统初始化时打开日志文件 process.log

{

为了记录操作系统从启动到关机过程中所有进程的运行轨迹，需要在硬盘上维护一 个日志文件/var/process.log。为了能尽早开始记录日志，应当在内核初始化时就打开 process.log 文件。 首先使用 Linux011 Kernel 模板新建一个项目，按照下面的内容修改其源代码： 内核的入口函数是 init/main.c 文件中的 start 函数，其中从第 181 行开始的一段 代码是： …… move\_to\_user\_mode(); if( !fork() ) { /\*we count on this going ok\*/ init(); } …… 这段代码是在进程 0 中运行的，先切换到用户模式，然后第一次调用 fork 函数创建子 进程 1，子进程 1 接着调用了本文件中的 init 函数继续进行初始化工作。 在 init 函数中从第 227 行开始的一段代码是： …… setup( ( void \* )&drive\_info ); ( void ) open("dev/tty0", O\_RDWR, 0 ); ( void ) dup( 0 ); ( void ) dup( 0 ); …… 这段代码建立了文件描述符 0、1 和 2，它们分别是 stdin、stdout 和 stderr，即标准 输入、标准输出和标准错误。可以在这里把 process.log 文件关联到文件描述符 3。 为了能尽早访问 process.log 文件，可以让上述建立文件描述符的工作在进程 0 中完成。所以，将这一段代码从 init 函数中移动(是剪切不是复制！ )到 start 函数中， 而且要放在调用move\_to\_user\_mode之后（不能再靠前了！），同时加上打开process.log 文件的代码。修改后的 start 函数如下： …… move\_to\_user\_mode(); /\*\*\*\*\*\*\*\*\*\*\*\* your code begin \*\*\*\*\*\*\*\*\*\*/ setup( ( void \* )&drive\_info ); ( void ) open("dev/tty0", O\_RDWR, 0 ); ( void ) dup( 0 ); ( void ) dup( 0 ); ( void ) open("/var/process.log", O\_CREAT | O\_TRUNC | O\_WRONLY, 0666 ); /\*\*\*\*\*\*\*\*\*\*\*\* your code end \*\*\*\*\*\*\*\*\*\*/ if( !fork() ) { /\*we count on this going ok\*/ init(); } …… 调用 open 函数打开 process.log 文件。第二个参数的含义是建立只写文件，并且 如果文件已存在则清空已有内容。第三个参数设置文件权限为所有人可读可写。这样，文件描述符 0、1、2 和 3 就在进程 0 中建立了，根据 fork 函数的原理，进 程 1 就会从进程 0 中继承这些文件描述符，而且，由于 此后所有的进程都是进程 1 的子 进程，当然也就会继承这些文件描述符了。 3.3.2 编写 fprintk 函数用于向 process.log 文件写入数据 process.log 文件将会被用来记录进程状态切换的轨迹，但是，所有进程的状态切 换工作都是在内核状态下进行的，此时 write 系统调用函数失效（其原理等同于不能在 内核状态下调用 printf 函数，而只能调用 printk 函数）。编写可以在内核状态下被调 用的 fprintk 函数的难度较大，所以这里直接给出源代码，主要是参考了 printk 函数 和 sys\_write 函数而写成的： #include<linux/sched.h> #include<sys/stat.h>

static char logbuf[1024];

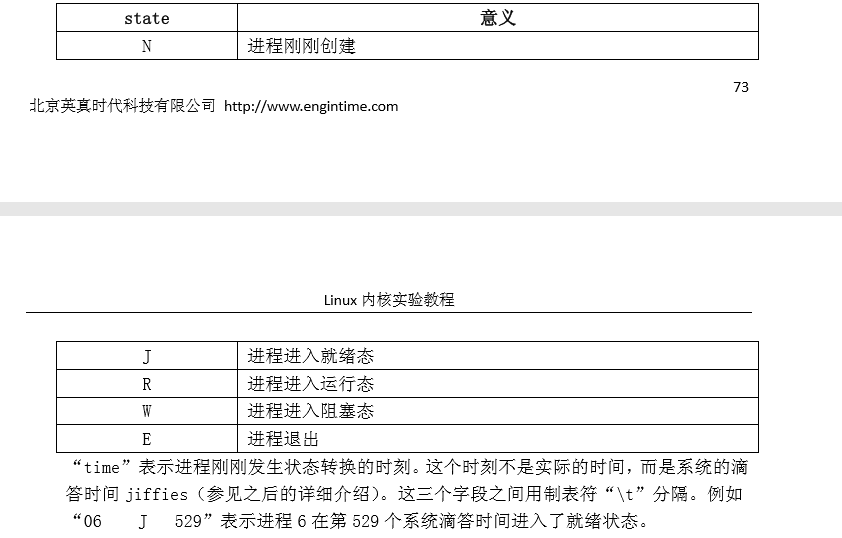
int fprintk( int fd, const char \* fmt, ... ) { va\_list args; int i; struct file \* file; struct m\_inode \* inode; va\_start (args, fmt); i = vsprintf (logbuf, fmt, args); va\_end (args); if( fd<3 ) { \_\_asm\_\_ ("push %%fs\n\t" "push %%ds\n\t" "pop %%fs\n\t" "pushl %0\n\t" "pushl $\_logbuf\n\t" "pushl %1\n\t" "call \_sys\_write\n\t" "addl $8,%%esp\n\t" "popl %0\n\t" "pop %%fs" ::"r" (i),"r" (fd):"ax", "dx"); } else { if( !( file=task[0]->filp[fd] ) ) return 0; inode=file->f\_inode; \_\_asm\_\_ ("push %%fs\n\t" "push %%ds\n\t" "pop %%fs\n\t" "pushl %0\n\t" "pushl $\_logbuf\n\t" "pushl %1\n\t" "pushl %2\n\t" "call \_file\_write\n\t" "addl $12,%%esp\n\t" "popl %0\n\t" "pop %%fs" ::"r" (i), "r" (file), "r" (inode) ); } return i;// 返回字符串长度 }  可以将该函数定义在 kernel/printk.c 文件中，并且在 include/linux/kernel.h 文件中添加该函数的声明。第一个参数 fd 是文件描述符，为 1 时将信息写入标准输出 stdout，为 3 时将信息写入 process.log 文件；第二个参数 fmt 是格式化字符串，类似 于 printf 的第一个参数；后面的是可变参数列表，类似于 printf 的可变参数列表。

3.3.3 记录进程的运行轨迹 之前已经打开了 process.log 文件并解决了向 process.log 文件写入数据的问题， 接下来需要解决在什么时刻将进程的运行轨迹信息写入 process.log 文件的问题。这就 要求读者对进程的状态转换以及进程调度有一个全面的了解。

Linux 0.11 支持四种经典的进程状态的转换过程：  就绪到运行：通过 schedule 函数（在文件 kernel/sehed.c 的第 135 行）完成。  运行到就绪：通过 schedule 函数完成。  运行到睡眠：通过 sleep\_on 函数（在文件 kernel/sehed.c 的第 202 行）和 interruptible\_sleep\_on 函数（在文件 kernel/sehed.c 的第 225 行）完成。或者 通过进程主动睡眠的系统调用内核函数 sys\_pause（在文件 kernel/sehed.c 的第 192 行）和 sys\_waitpid（在文件 kernel/exit.c 的第 183 行）完成。  睡眠到就绪：通过 wake\_up 函数（在文件 kernel/sehed.c 的第 251 行）完成。

此外还有进程的创建和退出两种情况：  进程的创建：通过 copy\_process 函数（在文件 kernel/fork.c 的第 89 行）完成。

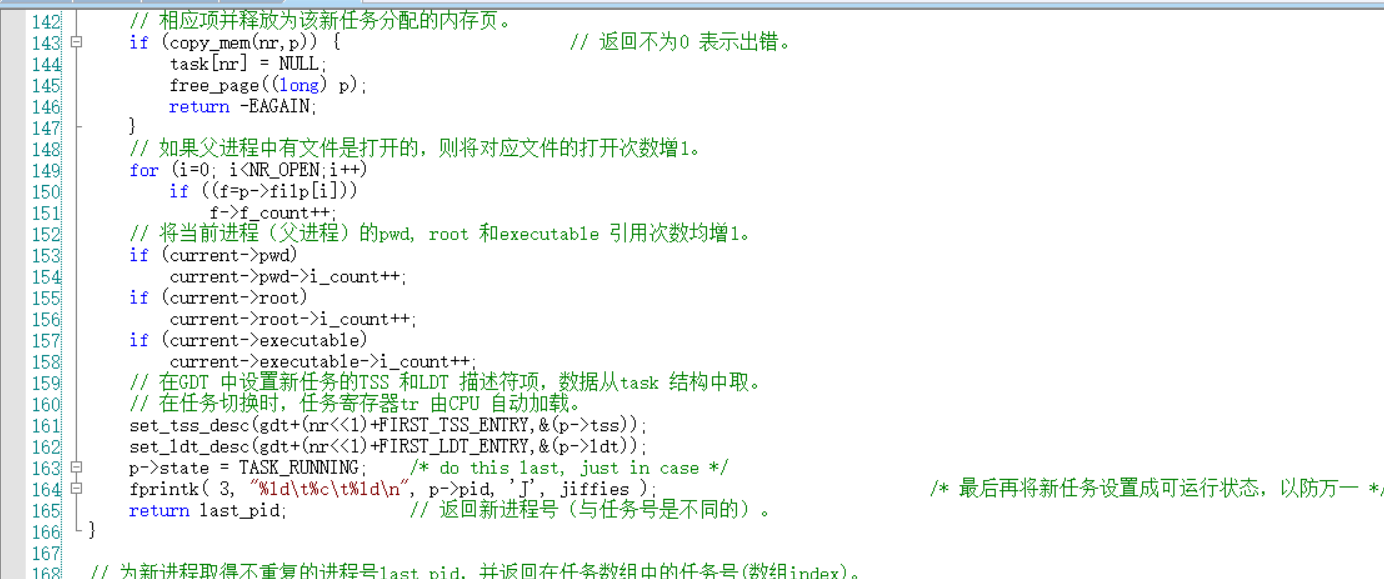
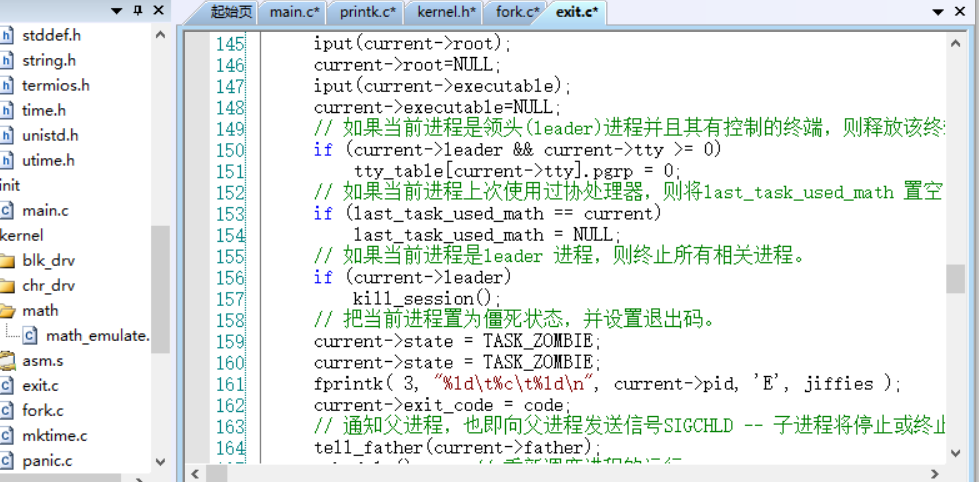
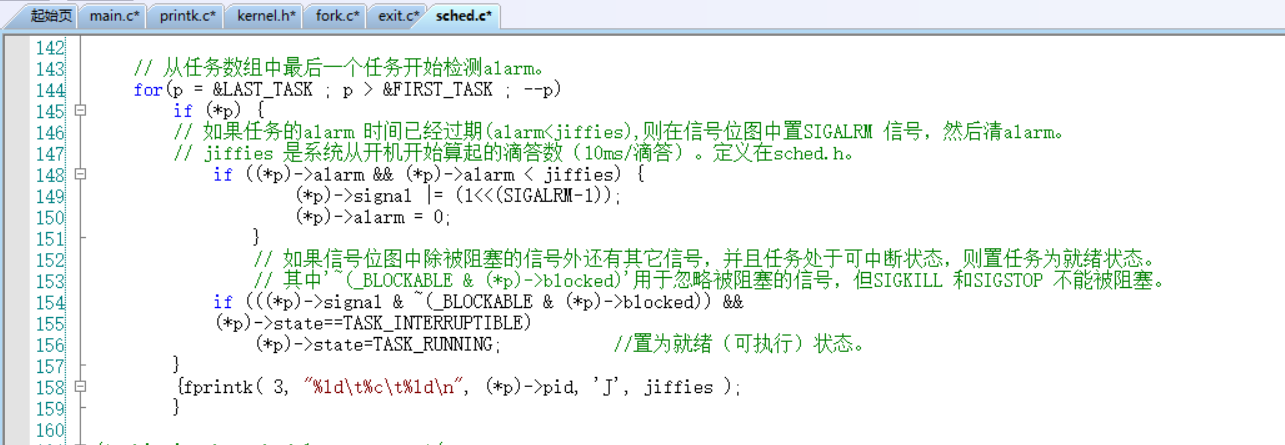
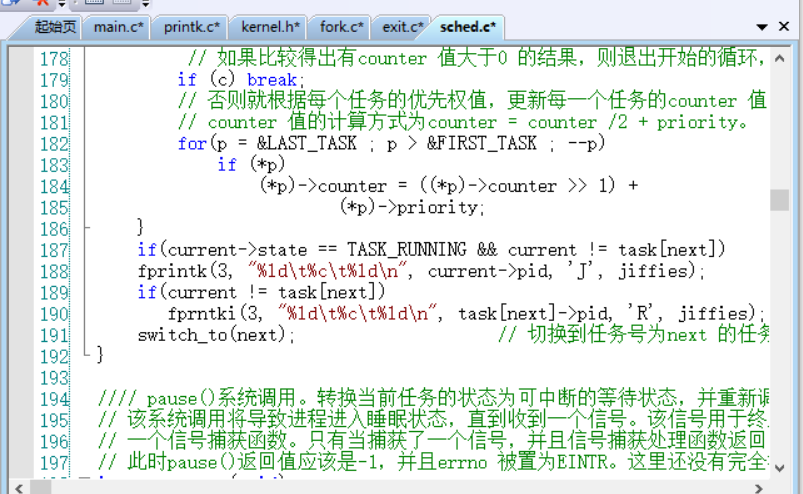
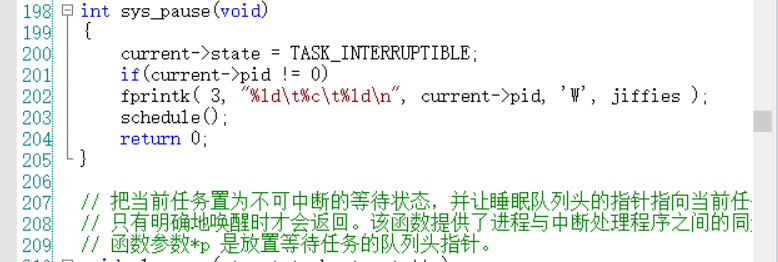
 进程的退出：通过 do\_exit 函数（在文件 kernel/exit.c 的第 122 行）完成。

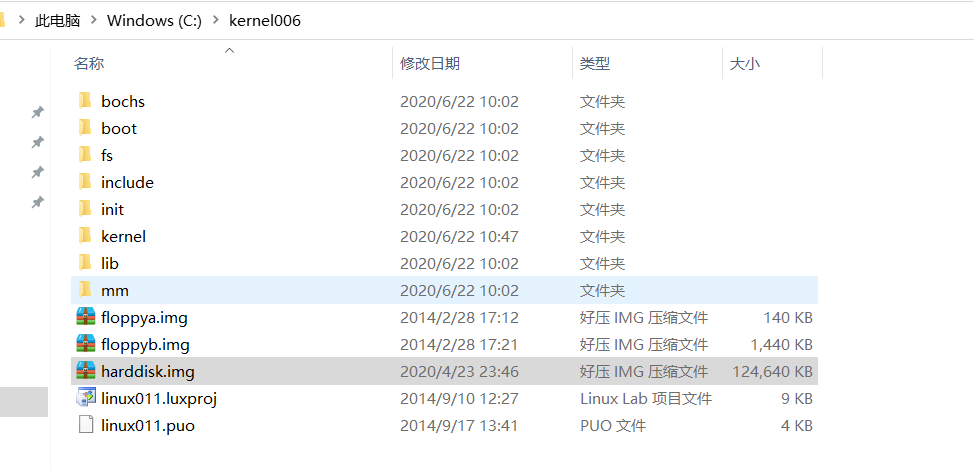
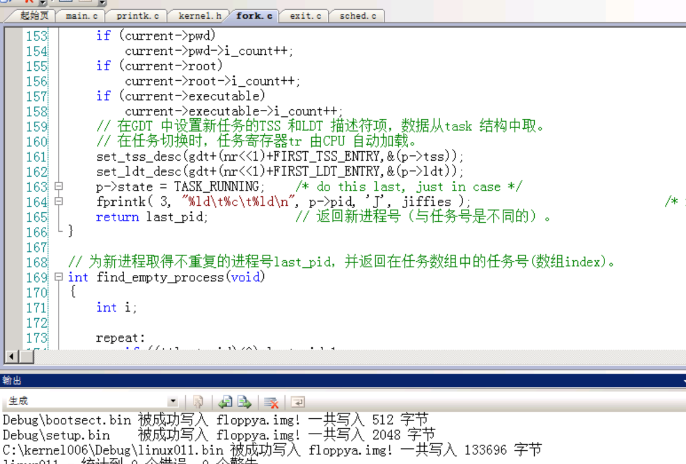
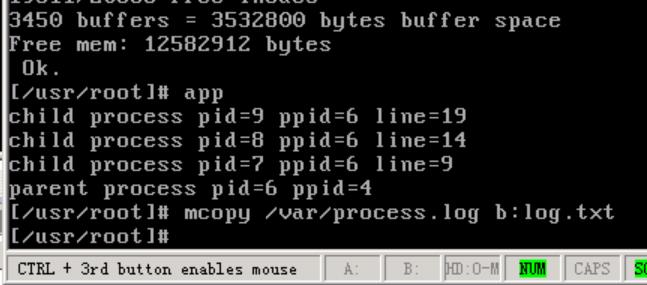
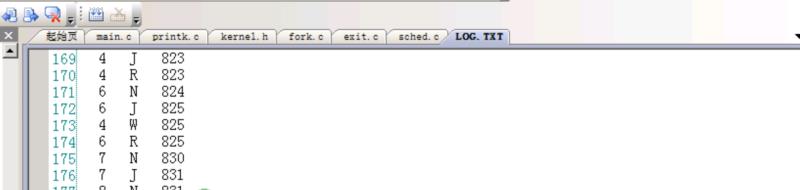
所以，只要在以上提到的这些函数的适当位置调用 fprintk 函数输出日志信息到 process.log 文件，就能完成进程轨迹的全面跟踪了。在调用 fprintk 函数前，先定义 process.log 文件中每行日志的格式为： pid state time 其中“pid”是进程的 id。“ state”表示进程刚刚进入的状态，其取值及意义如下表：

jiffies 滴答 jiffies是一个全局变量（在文件kernel/sched.c的第72行定义），它记录了Linux 0.11 操作系统从开机到当前时刻的 8253 定时计数器发生的中断次数。在 sched\_init 函数中（在 kernel/sched.c 文件的第 528 行），定时计数器的中断处理程序被设置为 timer\_interrupt 函数，在此函数响应定时计数器中断的过程中，每次都会将 jiffies 的值增加 1（在文件 kernel/system\_call.s 的第 243 行）。 此外，在 sched\_init 函数中有如下代码（在 kernel/sched.c 文件的第 524 行）： …… outb\_p(0x36,0x43); outb\_p(LATCH & 0xff , 0x40); outb(LATCH >> 8 , 0x40); …… 这段代码用来初始化 8253 定时计数器，设置中断时间间隔为 LATCH，而 LATCH 是一个 宏定义，在文件 kernel/sched.c 的第 57 行定义如下： #define LATCH (1193180/HZ) 宏 HZ 在文件 include/linux/sched.h 的第 5 行定义如下： #define HZ 100 同时，由于 8253 定时计数器输入时钟频率为 1.193180MHz（即 1193180/每秒），所以， LATCH=1193180/100 就是将定时计数器设置为每跳 11931.80 下产生一次时钟中断，即 每 1/100 秒（10 毫秒）产生一次时钟中断。所以，jiffies 实际上记录了从开机以来共 经历了多少个 10 毫秒。

下面的表格列出了在 Linux 0.11 的内核源代码中定义的进程状态的名称，在 Linux 0.11 内核项目中，使用“查找和替换”功能，以这些状态名称为关键字进行搜索，就 可以找到进程状态发生改变的所有源代码。读者也可以按照下面的步骤逐步添加调用 fprintk 函数的语句： 状态定义 含义 TASK\_RUNNING 可运行 TASK\_INTERRUPTIBLE 可中断的等待状态 TASK\_UNINTERRUPTIBLE 不可中断的等待状态 TASK\_ZOMBIE 僵死 TASK\_STOPPED 暂停 TASK\_SWAPPING 换入/换出

1. 为了跟踪进程的创建，可以修改 kernel/fork.c 文件中的 copy\_process 函数。在 第 114 行后面增加语句（一定在子进程控制块的 pid 和 start\_time 被赋值之后）: p->start\_time = jiffies;

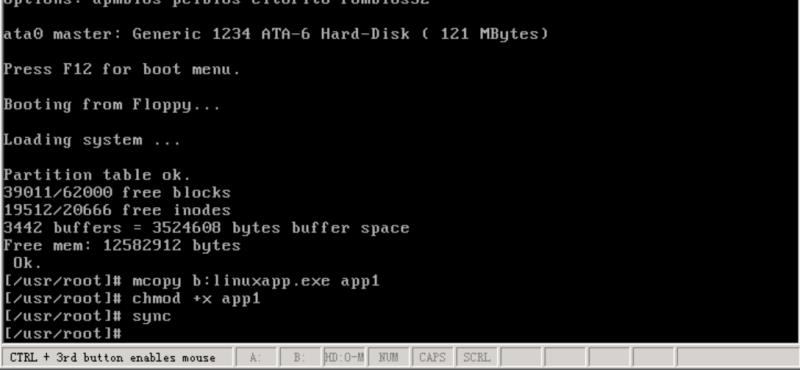
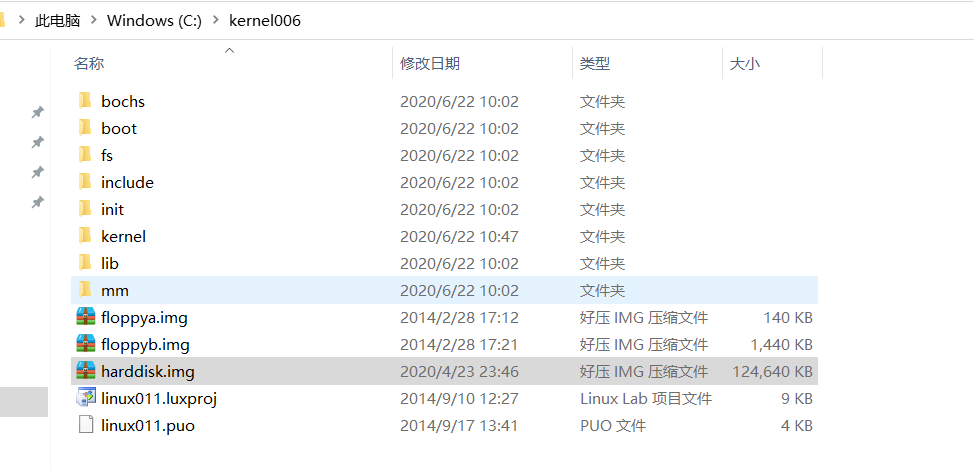
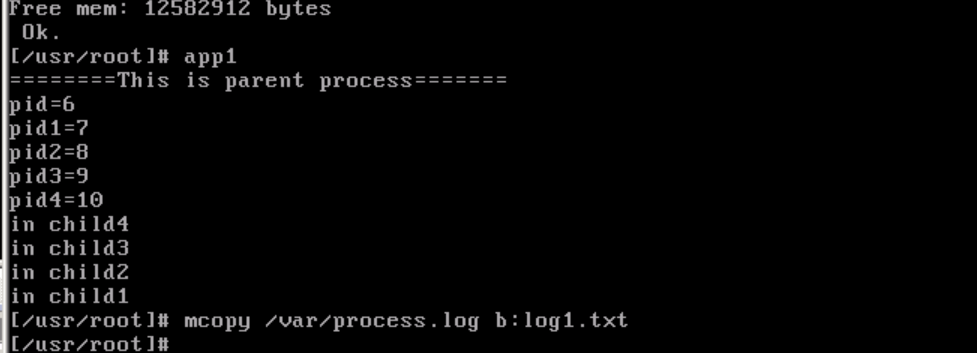
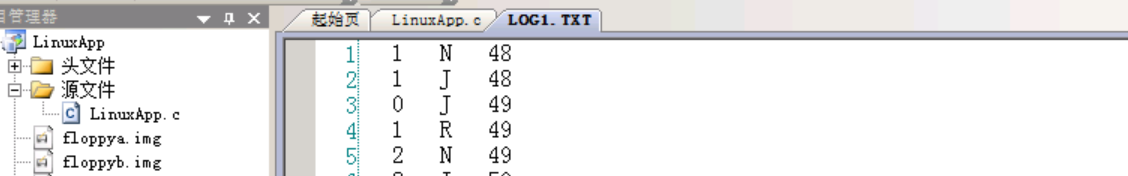
fprintk( 3, "%ld\t%c\t%ld\n", p->pid, 'N', jiffies ); 记录进程刚刚创建并处于不可中断睡眠状态。在第 164 行后面增加语句： fprintk( 3, "%ld\t%c\t%ld\n", p->pid, 'J', jiffies ); 记录刚刚创建的进程进入就绪状态。 2. 当进程的状态被设置为 TASK\_ZOMBIE，并且已经获取了退出码时，表示进程已经完 成了退出操作，虽然此时进程的控制块还没有被释放（要留待父进程获取子进程的 信息）。子进程退出的最后一步是发送信号通知父进程，目的是唤醒正在等待此事 件的父进程，由父进程来释放子进程的控制块。从时序上来说，应该是子进程先退 出，父进程才被唤醒。所以，为了跟踪进程的退出，可以修改 kernel/exit.c 文件 中的 do\_exit 函数。在第 159 行将进程状态设置为 TASK\_ZOMBIE 的后面插入一条语 句，如下： current->state = TASK\_ZOMBIE; fprintk( 3, "%ld\t%c\t%ld\n", current->pid, 'E', jiffies );  3. 接下来，修改 kernel/sched.c 文件中的 Schedule 函数，跟踪进程进入就绪状态或 运行状态。注意，Schedule 函数找到的 next 进程是接下来将要运行的进程。如果 next 恰好是当前正处于运行态的进程，switch\_to(next)也会被调用，这种情况相 当于当前进程的状态没有改变。在第 157 行进程由于收到信号而进入就绪状态，所 以需要在此行的后面添加一条语句： fprintk( 3, "%ld\t%c\t%ld\n", (\*p)->pid, 'J', jiffies ); 注意此句要跟上面一句代码共同属于 if 语句，需加{} 在第 188 行 switch\_to(next)语句前添加下面的代码： if(current->state == TASK\_RUNNING && current != task[next]) fprintk(3, "%ld\t%c\t%ld\n", current->pid, 'J', jiffies); if(current != task[next]) fprintk(3, "%ld\t%c\t%ld\n", task[next]->pid, 'R', jiffies);  4. 另外，只要操作系统处于空闲状态，就会让进程 0 运行，进程 0 会不停的调用 sys\_pause 函数，以激活调度程序，使得调度程序可以随时选择处于就绪态的进程 来运行。此时，可以认为进程 0 处于等待状态（等待有其它可运行的进程），也可 以认为进程 0 处于运行态，因为它是唯一在 CPU 上运行的进程，只不过运行的效果 是等待。所以，在修改 sys\_pause 函数时，添加的跟踪语句应该为：(在第 202 行 schedule 函数的前面) if(current->pid != 0) fprintk( 3, "%ld\t%c\t%ld\n", current->pid, 'W', jiffies );  5. 注意，当进程被唤醒时虽然会将进程设置为 TASK\_RUNNING 状态，但是进程实际是 进入了就绪状态，所以此时在日志记录中应使用标志‘J’ 。只有在 Schedule 函数 的最后，被选中为 next 的进程才会进入实际的运行状态，才能在日志记录中使用 标志‘R’。也就是说，在读者添加的所有调用 fprintk 函数的语句中，只有在 Schedule 函数的最后这一个地方使用‘R’标志。余下的跟踪语句请读者在适当的 位置自行添加。（提示，包括前面提到的几处，总共需要添加 15 处，可以将 “ fprintk” 作为“查找”功能的关键字进行查找，确保数量正确）。

待读者添加完所有的跟踪语句之后，按照步骤完成下面的操作：1. 使用在第3.2节创建的Linux011应用程序项目的硬盘镜像文件harddisk.img覆盖 当前 Linux011 Kernel 项目的 harddisk.img 文件，这样就可以继续使用之前生成 的应用程序可执行文件 app 了。 2. 按 F5 启动调试 Linux011 Kernel 项目。 3. 执行 app 可执行文件（在纸上记录下打印输出的父进程和子进程的 id，以便下面 分析数据）。 4. 将 process.log 文件复制到软盘 B，命令如下： mcopy /var/process.log b:log.txt 5. 结束此次调试。 6. 在“项目管理器”窗口中双击 floppyb.img 文件，使用软盘编辑器工具打开，将软 盘 B 中的 log.txt 文件复制到 Windows 的 C 盘根目录下。 7. 打开 log.txt 文件（可以将 log.txt 文件拖动到 Linux Lab 窗口中释放）。要求读 者在日志文件中能够找到类似于下面日志的 app 应用程序的运行轨迹信息。其中， 04 是 Shell 程序的进程 id，06 是 app 的主进程 id，07,08 和 09 是三个子进程 id。 结合应用程序 app 的源代码及其运行时打印输出的信息分析这些日志，理解进程在 生命周期中状态的转换过程和进程调度过程。 …… 4 J 528 4 R 528 06 N 529 06 J 530 04 W 530 06 R 530 07 N 534 07 J 534 08 N 535 08 J 535 09 N 535 09 J 536 06 W 536 09 R 536 09 E 537 06 J 537 08 R 537 08 E 538 07 R 538 07 E 539 06 R 539 06 E 539 ……

在之前的应用程序 app 中还无法控制各个子进程运行的时间，特别是无法控制子进 程实际占用 CPU 的时间和等待 I/O 操作的时间，也就无法对进程的调度过程进行量化分 析。但是，如果在内核中对各种设备的 I/O 操作时间进行统计，并在应用程序中对各种I/O 设备进行访问也不现实。所以，接下来采用在应用程序中模拟这样一种折中的方式， 从而实现对进程调度过程的量化分析。 3.4 编写用于模拟的应用程序 修改在第 3.2 节中编写的多进程程序作为测试程序。首先在 LinuxApp.c 文件中添 加一个函数 cpuio\_bound，用来模拟进程在生命周期中占用 CPU（运行态）与 I/O 操作 （阻塞态）的情景，代码为: #include<sys/wait.h> #include<linux/sched.h> #include<time.h> void cpuio\_bound( int last, int cpu\_time, int io\_time ) { struct tms start\_time, current\_time; clock\_t utime, stime; int sleep\_time; while( last>0 ) { times( &start\_time ); do { times( &current\_time ); utime=current\_time.tms\_utime-start\_time.tms\_utime; stime=current\_time.tms\_stime-start\_time.tms\_stime; }while( ( ( utime+stime )/HZ )< cpu\_time ); last-=cpu\_time; if( last<=0 ) break; sleep\_time=0; while( sleep\_time<io\_time ) { sleep( 1 ); sleep\_time++; } last-=sleep\_time; } } 参数 last 表示占用 CPU 以及 I/O 操作的总时间，不包括在就绪队列中的时间。参数 cpu\_time 表示一次连续占用 CPU 的时间，必须大于等于 0。参数 io\_time 表示一次 I/O 操作占用的时间，必须大于等于 0。如果 last > cpu\_time + io\_time,则往复多次占用 CPU 和 I/O 操作，且时间单位均为秒。其中 struct tms 在 include/sys/times.h 中定 义，clock\_t 在 include/time.h 中定义。将 main 函数修改为如下的代码： int main( int argc, char \* argv[] ) { pid\_t p1, p2, p3, p4; if( ( p1=fork() )==0 )

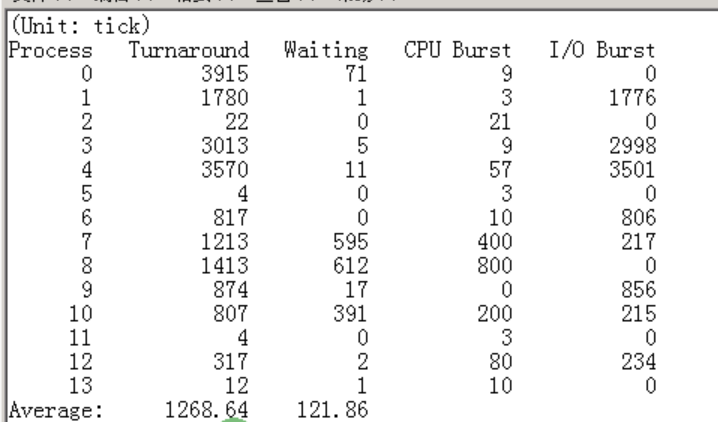
Linux 内核实验教程

{ printf( "in child1\n" ); cpuio\_bound( 5, 2, 2 );} else if( ( p2=fork() )==0 ) { printf( "in child2\n" ); cpuio\_bound( 5, 4, 0 );} else if( ( p3=fork() )==0 ) { printf( "in child3\n" ); cpuio\_bound( 5, 0, 4 );} else if( ( p4=fork() )==0 ) { printf( "in child4\n" ); cpuio\_bound( 4, 2, 2 );} else { printf( "========This is parent process=======\n" ); printf( "pid=%d\n", getpid() ); printf( "pid1=%d\n", p1 ); printf( "pid2=%d\n", p2 ); printf( "pid3=%d\n", p3 ); printf( "pid4=%d\n", p4 ); } wait( NULL ); return 0; }

按照下面步骤操作，得到最新的日志文件: 1. 按 F7 生成 Linux011 应用程序项目，F5 启动调试。 2. 将生成的可执行文件从软盘 B 拷贝到硬盘，命令如下 mcopy b:linuxapp.exe app 3. 给 app 文件添加可执行权限，命令如下： chmod +x app 4. 执行“sync”命令。 5. 结束调试，用 Linux011 应用程序项目中的硬盘镜像文件 harddisk.img 覆盖 Linux011 Kernel 项目中的 harddisk.img 文件，这样就可以在内核项目中继续使 用刚刚生成的 app 可执行文件了。 5. 按 F5 启动调试 Linux011 Kernel 项目。 6. 执行 app 可执行文件（记录父进程和子进程的 id）。 7. 将 process.log 文件复制到软盘 B，命令如下： mcopy /var/process.log b:log.txt 8. 结束此次调试。 8. 在“项目管理器”窗口中双击 floppyb.img 文件，使用软盘编辑器工具打开，将软 盘 B 中的 log.txt 文件复制到 Windows 的 C 盘根目录下。 9. 打开 log.txt 文件（可以将 log.txt 文件拖动到 Linux Lab 窗口中释放）。查看日 志文件中记录的信息，根据之前记录的进程 id 找出 app 应用程序运行时产生的进 程调度轨迹信息。

根据信息以及app的源文件思考app运行时父进程及子进程在生命周期状态转换的 过程，分析进程调度的过程及其原因,进一步理解进程在生命周期中进程的状态转换和 进程调度。

Linux 内核实验教程

3.5 统计并分析数据 前面已经得到 log.txt 日志文件，为了加深对进程调度和调度算法的理解；接下来 统计分析 log.txt 文件的数据，计算平均周转时间、平均等待时间和吞吐率，量化分析 进程调度和调度算法。分析数据步骤如下： 1. 为了从 log 文件读取数据，然后计算平均周转时间、平均等待时间和吞吐率，需要 编写一个程序，用什么语言编写都行，读者可自行设计。为了方便，给读者提供一 个 Python 语言编写的程序。打开“学生包”找到本实验对应的文件夹，将其中的 “stat\_log.py”文件复制到 C 盘根目录下。（在“学生包”本实验对应文件夹下提 供了 Python 的安装包，有需要的读者可自行安装到自己的电脑上） 2. 检查 log.txt 日志文件，保证文件的最后一条记录是完整的，若不完整则删除。 3. 启动 Windows 的控制台，将当前目录设置为 C 盘根目录，然后输入命令： C:\>stat\_log.py log.txt -g > out.txt 如果对 C 盘根目录没有写权限，可以将这些文件移动到其他盘的根目录或文件夹中。

4. 查看在 C 盘根目录下生成的 out.txt 文件中的内容，其中一部分内容如下： (Unit: tick) Process Turnaround Waiting CPU Burst I/O Burst 0 3501 64 8 0 1 1680 0 1 1679 2 24 4 20 0 3 3003 0 4 2999 4 2948 43 45 2860 5 3 1 2 0 6 868 0 7 860 7 1008 398 400 210 8 1208 407 800 0 9 861 0 0 860 10 4 0 3 0 11 520 2 66 452 12 10 0 9 0 Average: 1202.92 70.69 Throughout: 0.37/s

意义如下表： 名称 意义 Process 进程 id Turnaround 周转时间 Waiting 等待时间 CPU Burst 占用 CPU 时间 I/O Burst I/O 时间 Average 平均周转时间和平均等待时间 Throughout 吞吐率 进程 0 比较特殊，因为其运行轨迹没有被完全记录下来，所以得到的信息误差比较 大，可以忽略进程 0；其它进程的信息可能也存在误差，但在误差允许范围之内。

Linux 内核实验教程

* 1. 调度算法、时间片和优先级 根据 Linux 0.11 的进程调度函数 schedule 的代码（在文件 kernel/sched.c 的第 135 行），可知 Linux 0.11 的调度算法是选取 counter（时间片）最大的就绪进程占用 CPU。运行态进程（即 current）的 counter 每当发生时钟中断时就会减 1（在 do\_timer 函数中完成，在文件 kernel/sched.c 的第 416 行），所以是一种比较典型的时间片轮转 调度算法。另外，由 schedule 函数可以看出，当没有 counter 大于 0 的就绪进程时， 要对所有进程做“(\*p)->counter = ((\*p)->counter >> 1) +(\*p)->priority;”操作 （kernel/sched.c 第 182 行），效果是对所有的进程（包括阻塞进程）进行 counter 的 衰减（除 2），并累加 priority 的值，这样对已经阻塞的进程来说，一个进程在阻塞队 列中停留的时间越长，其优先级越大，被分配的时间片就越大（不会大于优先级的 2 倍），而对于时间片已经为 0 的进程来说，其时间片的值会被重置为其优先级的值。所 以总的来说，Linux 0.11 的进程调度是一种综合考虑进程优先级，并能动态反馈调整 时间片的轮转调度算法。 再来看一下进程的时间片是如何被初始化的。进程在被创建时，在 copy\_process 函数（kernel/fork.c 第 89 行）中有如下代码： …… \*p = \*current; …… p->counter = p->priority; …… 虽然父进程的时间片会发生改变，但是优先级不会改变，上面的第二句代码将 p->counter 设置成 p->priority，说明在创建进程时，分配的时间片是一个固定值。同 时，由于每个进程的优先级都是从父进程继承的，除非自己通过调用 nice 系统调用函 数（在文件 kernel/sched.c 的第 506 行）修改优先级。结合以上的因素，如果没有调 用 nice 系统调用，进程时间片的初值就是进程 0 的优先级。进程 0 的优先级由宏 INIT\_TASK（在文件 include/linux/sched.h 的第 176 行）定义如下： #define INIT\_TASK \ { 0,15,15,\ //state, counter, priority …… 所以，如果修改了进程 0 的优先级，就会修改所有进程的初始时间片。 现在请读者尝试将进程的初始时间片改大一些（大于 15），重新 按照 3.4 的内容跟 踪进程运行的轨迹，得到新的日志文件，然后按照 3.5 的内容得到统计数据。再请读者 尝试将进程的初始时间片改小一些（小于 15），同样得到统计数据。最后，请读者将得 到的统计数据填入下面的表格，通过对比这些数据体会时间片大小对平均周转时间、平 均等待时间和吞吐率的影响，并分析其中的原因。
  2. **实验总结**

本次实验让我对调试进程在各种状态间的转换过程，熟悉进程的状态和转换。通过对进程运行轨迹的跟踪来形象化进程的状态和调度。掌握 Linux 下的多进程编程技术以上知识点有了更深的掌握，使我对实验的流程有了更深的理解！