**操作系统实验之进程调度**

小组人员：林子义、李搏、黄秋宇、胡亚龙

实验内容：

1. 实现作业的入队、出队操作。
2. 实现作业状态命令查看。
3. 实现多级反馈轮转调度算法。
4. 对程序的状态进行调试。

实验分工：

1. 黄秋宇（14061176）：BUG2、BUG3修复，总体测试。
2. 李搏（14061187）：BUG1修复，作业状态命令查看FIFO实现。
3. 林子义（14061194）：多级反馈轮转调度算法代码主要部分。
4. 胡亚龙（14061196）（主要负责人）：BUG4修复、会议记录，总体测试、实验报告撰写。

实验原理：

1. fork()之后父进程和子进程之间的通信是通过信号原语来实现的，通过设置信号goon来实现进程之间的同步，在子进程执行完毕之后设置goon，让循环等待的父进程执行。
2. 程序中的current和next两个工作被选出来之后就是一个独立的工作了，需要把其next属性置空，这样在执行完时间片放回链表之后，才不会形成环。
3. FIFO文件是一种高级的通信管道，存在于文件系统中，其存在依赖于进程的存活期，支持读写，读写的内容不能超过管道的缓冲区的大小，不然会造成数据混乱。
4. 多级轮转反馈调度算法:多级轮转反馈调度算法是一个优先级算法，根据优先级来确定执行任务的顺序，然后为了维持进程占用CPU的公平性，对每个优先级设置了不同的轮转时间片长度以及优先级动态变更规则。具体来说就是设置多个优先级，让一些比较重要的作业优先执行，优先级越低轮转的时间越长（因为低优先级的等待时间比较长，当然要占用比较长时间的CPU），对于当前的输入作业，如果优先级高于正在运行的作业，要进行抢占，被抢占的作业被放到最高优先级的第一个。执行完一个时间片的作业优先级恢复至初始优先级继续轮转等待。实际上这个调度算法一个比较折中的方案，如果操作系统采用这种调度算法，不会太快，因为隔一会就会有低优先级的算法升级了过去抢占CPU，也不会太慢，因为只要你初始优先级足够高，你不会一直等待。这个算法的问题也是非常明显的，首先有的任务根本就不需要多少CPU执行时间，他大多的时间都是在读写，存在时间长，那么分配给他的很多时间片会浪费，他可能根本就用不完时间片；其次无法处理优先级倒挂的问题，譬如一个纯计算的进程需要等待一个读取进程的数据，然而读取进程的优先级非常低，这样反而让优先级高的纯计算进程无法继续。再次时间片的选择也是一门艺术，如果选择过长，那么时间片轮转的加速意义就不明显，如果选择过短，进程切换的消耗就大大增加，更加不划算。最后，这个调度算法的实时性不高，无法完全做到及时响应。不过这个调度算法用来处理一般的应用程序来说是非常适合的，因为时间片一般在毫秒级别，这对于用户来说是感觉不到的，所以基本可以做到流畅操作。

调试过程：

首先定义一个输出函数，打印当前所有作业的如下信息：

1.作业ID

2.进程ID

3.作业所有者

4.作业运行时间

5.作业等待时间

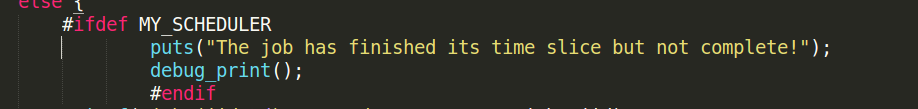
6.作业创建时间

7.作业状态

代码截图如下：

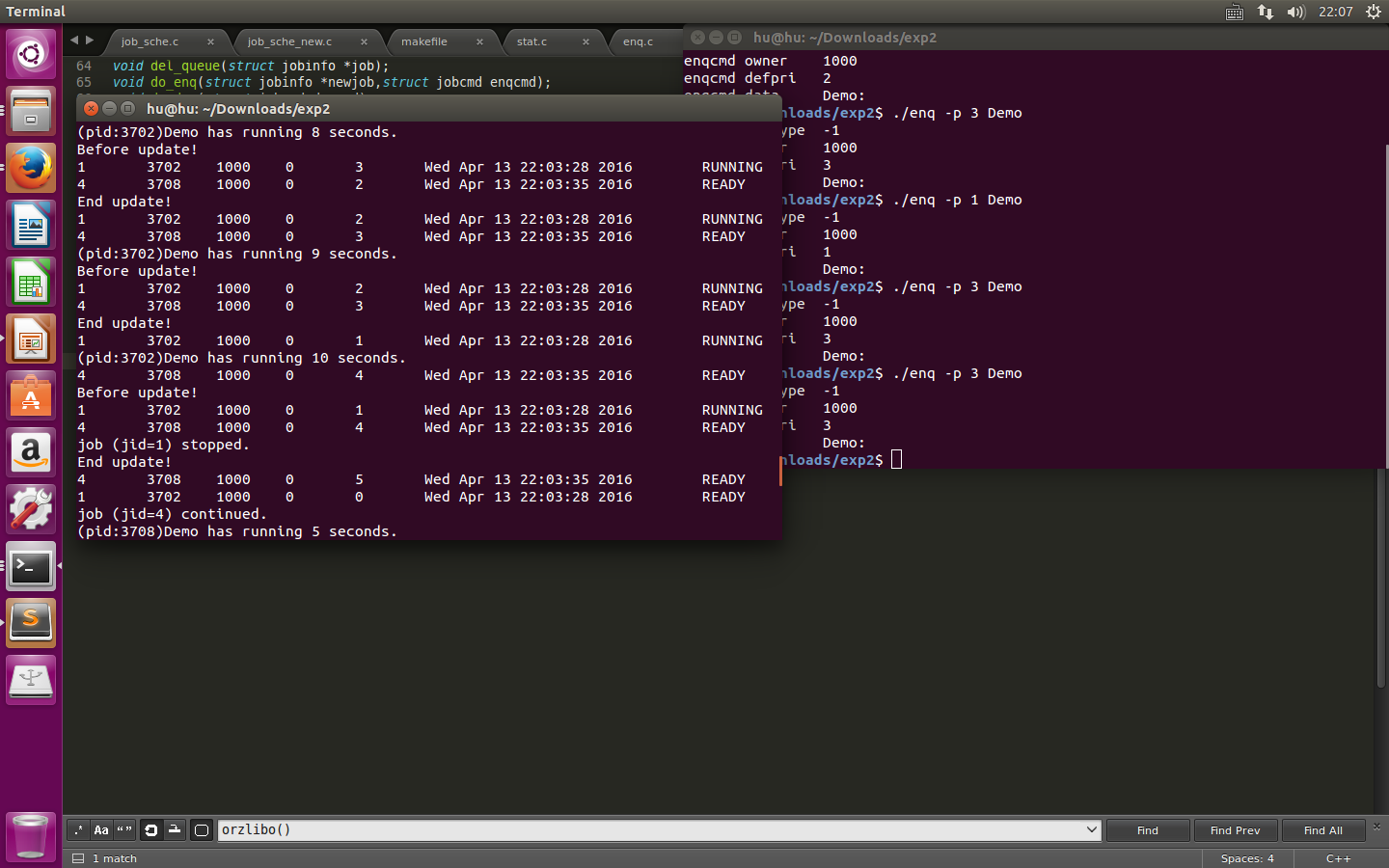


至于每次的调试信息，只要找到对应的位置加入相应的调试信息并且调用输出函数就好。举一个显示收到子进程stop信号之后输出作业状态的例子，其他的情况大同小异：

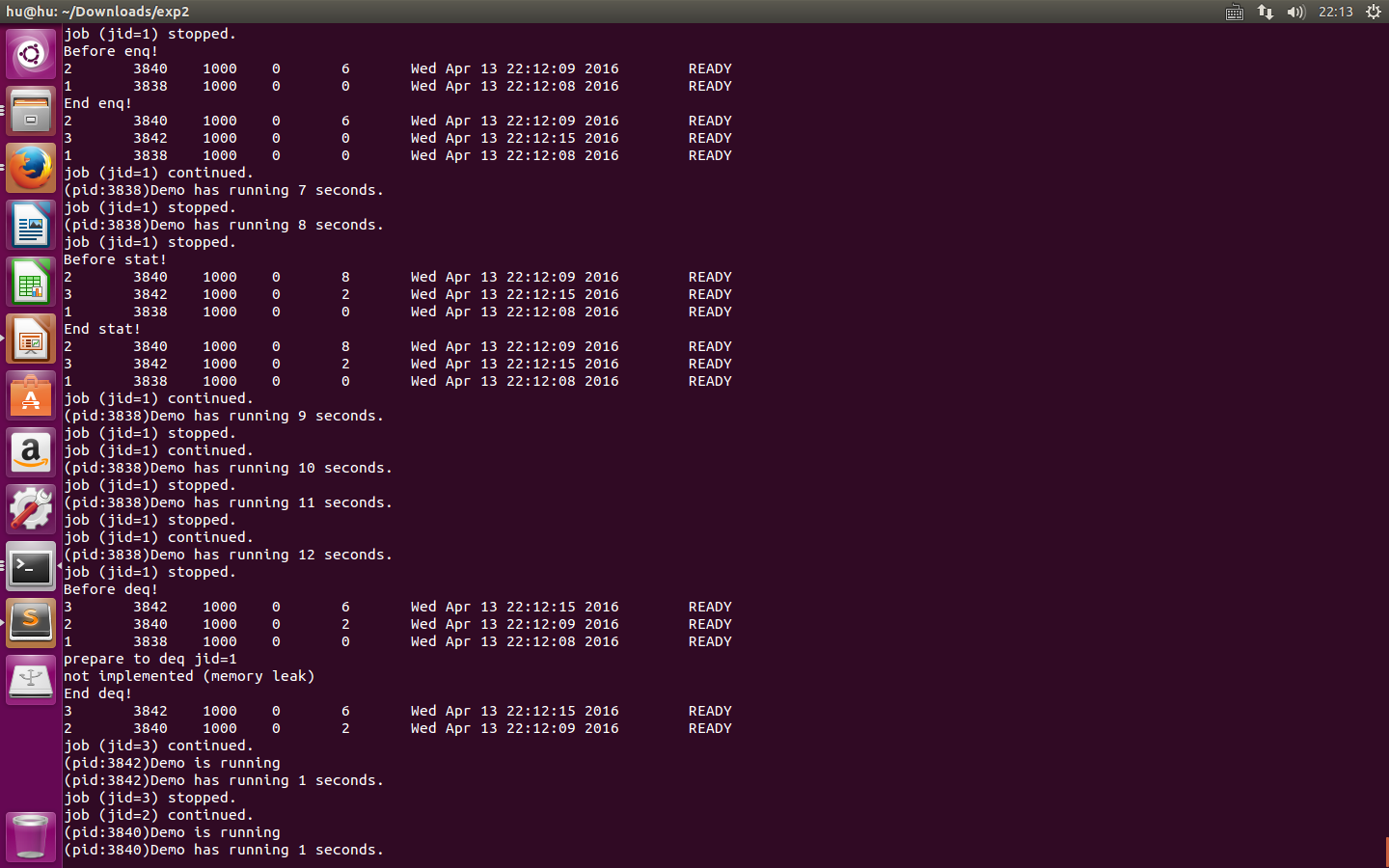


在添加调试的过程中遇到了一个很奇葩的问题，就是ctime()这个函数的调用在不添加头文件#include<time.h>的时候也可以编译通过，但是运行就报段错误。

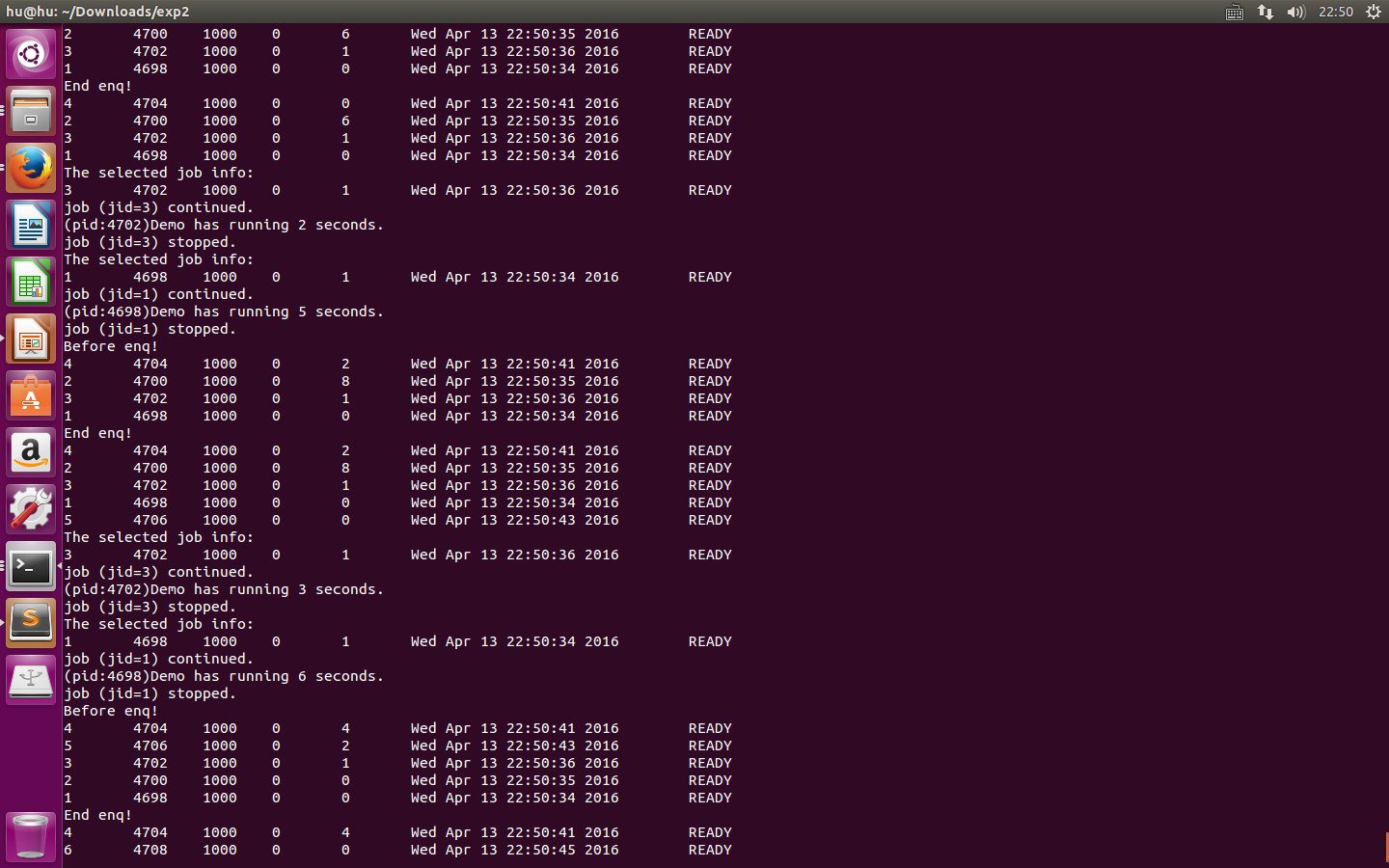
Update()函数的调试：



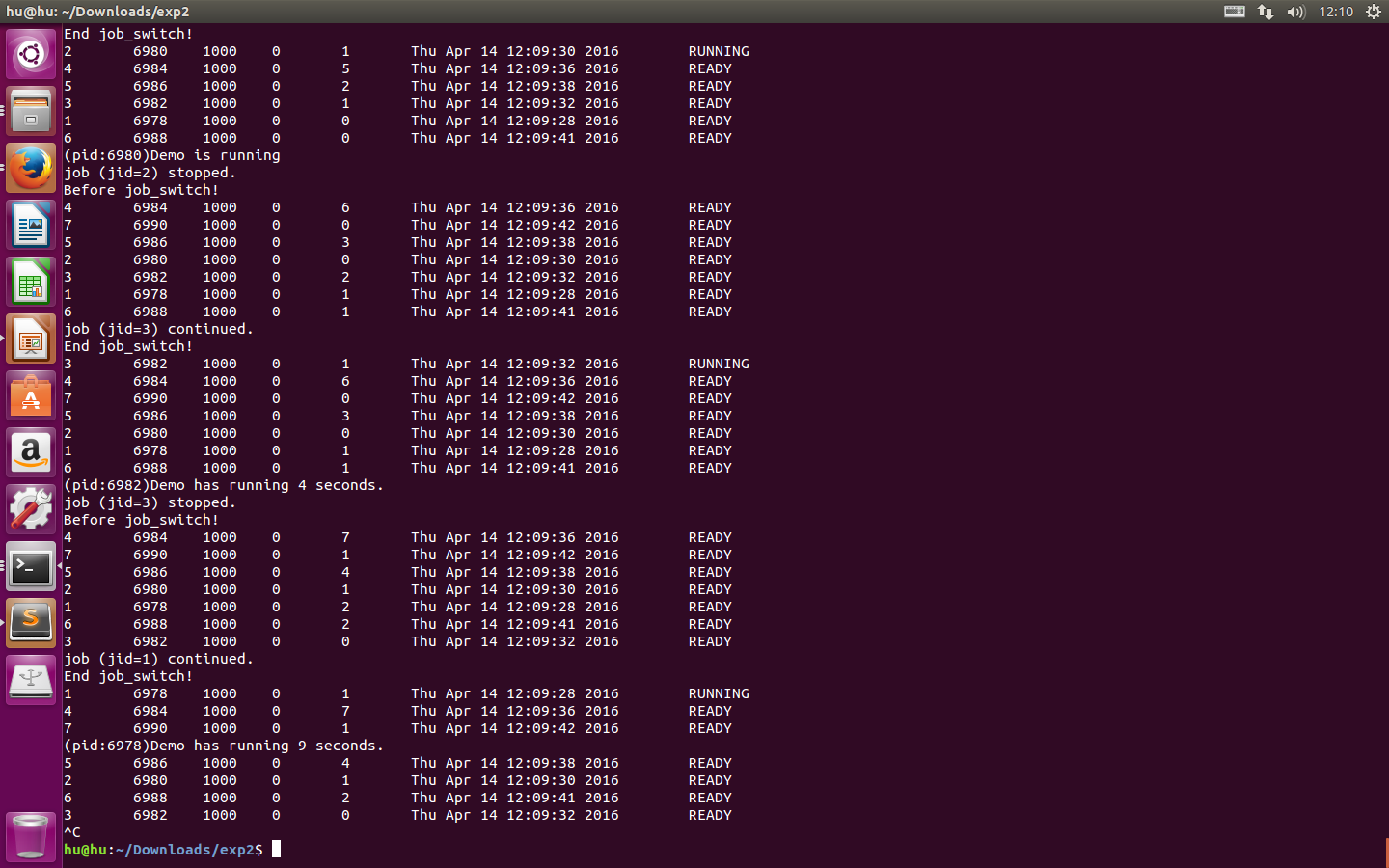
实现在enq,deq,stat前后显示队列里所有作业的信息。



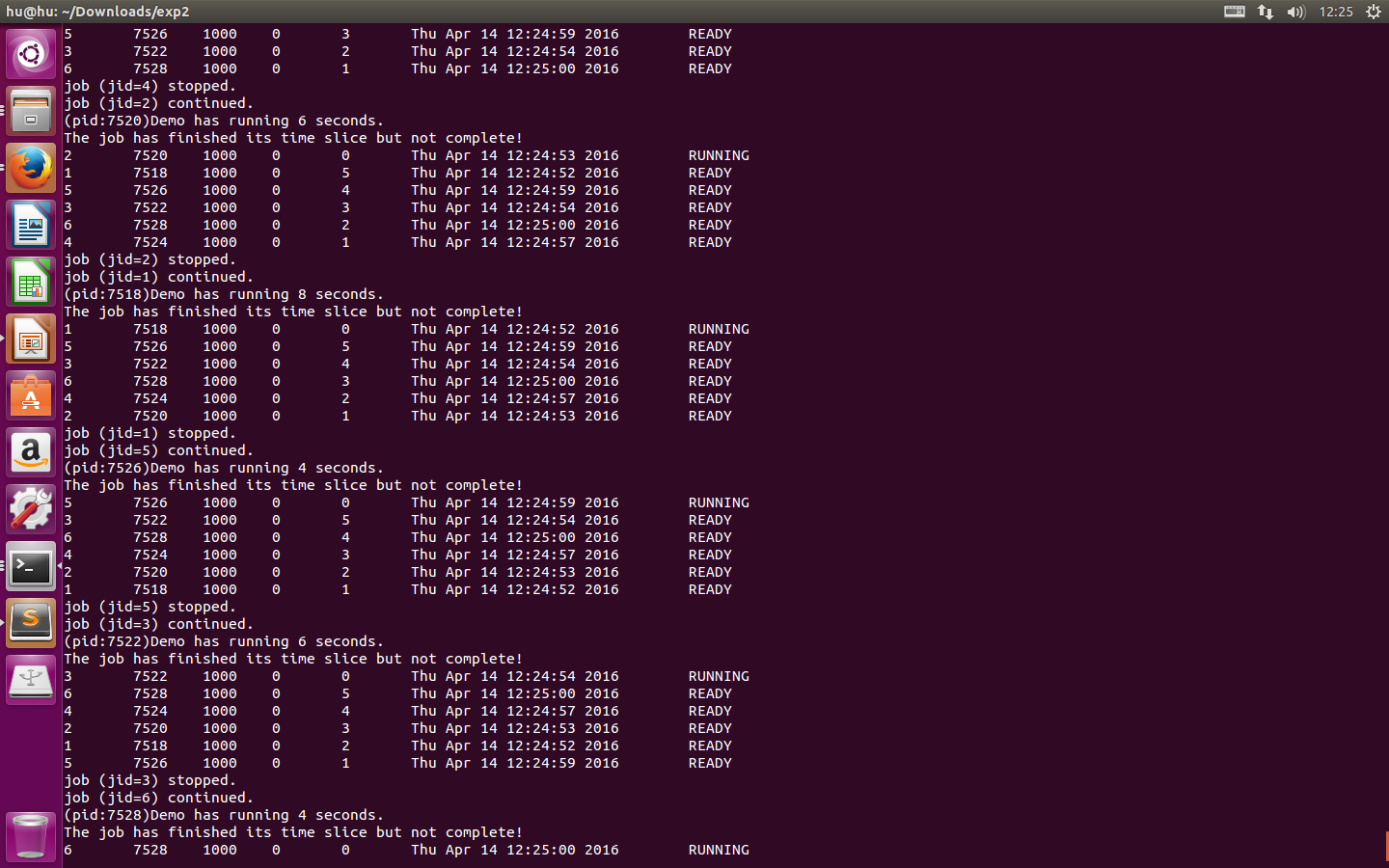
实现显示job\_select()函数选择的作业信息的调试：



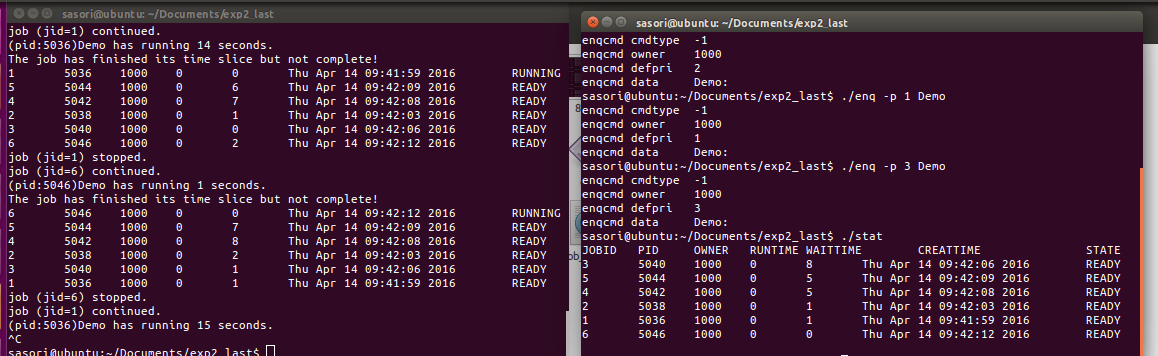
实现job\_switch()函数执行前后作业状态的变化信息：



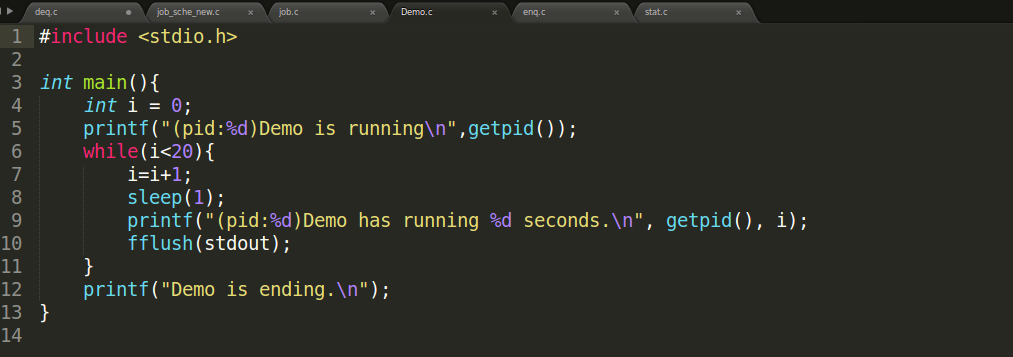
实现显示当前作业完成后发送信号给父进程时显示所有作业状态和信息：



执行./stat在本终端显示作业状态和信息：



修改了Demo.c，让其不再死循环，执行到一段时间后自动终止。



实验收获&感想:

经过本次实验，加深了对于Linux系统内部进程之间通信的理解，同时较为深入的思考了操作系统的调度算法，虽然并没有量化的计算每个调度算法的平均等待时间、CPU有效利用率等数据，但是对于每个调度算法都有过思考，比较深刻的理解了设计各种调度算法的初衷以及算法适用的任务，设想过并且计算了一些特殊情况。我们设计调度算法的初衷是因为我们要完成一些特定的任务，在综合了硬件、软件、成本之后所设计出来的折中算法，而多级轮转反馈调度算法无疑是其中的佼佼者，应用非常广泛，而且具有适应性。在查找资料文献的过程中，我们也稍微了解了一些别的算法，并且就其优劣性进行了分析。如适用于批处理系统和实时系统的FPF算法，总是执行优先级最高的进程，这样就能保证紧急任务总能得到执行；SJ(P)F算法，总是将短作业进程优先调度的算法；实时性比较强的EDF或者RMS算法。但是很多算法都需要预估进程所需的运行时间，一般来讲是根据进程已经运行的时间长度和代码的完成量，以及进程的结构来大概的估计一下时间，然后选择合适的调度来实现。其实不管是什么调度，最终的目标都是快、省、准，如果是批处理系统，直接优先级调度就好，不需要轮转，直接一个做完做下一个，这样减少了逻辑和进程切换的花销，对于纯计算的CPU，直接傻瓜调度+抢占式运行就好，因为进程都差不多，轮转会消耗比重比较大的CPU时间。就本次进程调度实验来看，完成本次实验的内容只是一个基本要求，更重要的是对操作系统的调度算法有个比较明确的认识，对于调度的过程需要有比较清晰的认识，并且学会去分析对于一个比较有特点的操作系统，究竟什么样的算法才是最适合的算法。