修改一：

去掉用户态和内核态之分，每次运行用户程序时可以执行kernal部分的代码，自由添加用户程序。直接去掉变量isUserMode即可。

修改二：

之前实验6,7,8在载入用户程序的时候,偶尔会卡住，之前一直找不到问题所在，在做实验九的时候，这个问题也是有发生，但是这一次我发现了问题所在。

之前载入用户程序的函数是这样子的(实验6,7,8该函数具体实现不同，但是逻辑是一样的)，由内核执行：

/\*运行新程序\*/

int runProg(int numOfSectors, int startOfSectors)

{

1. int x,y,z;
2. int pid = createProcess();
3. x = startOfSectors;
4. y = x / 18;
5. z = x % 18 + 1;
6. LoadsubProgram(numOfSectors, y & 1, y >> 1, z); /\*加载进程到内存\*/
7. return pid;

}

/\*创建新的进程\*/

int createProcess()

{

int i;

for(i = 1; i < MAX\_NrPCB; i++)

{

if(pcb\_list[i].used == 0)

{

currentSeg = SegLength\*(i-1) + baseSeg;

initPCB( &pcb\_list[i] ,i, currentSeg, SP\_OFF); /\*初始化pcb块\*/

processNum++;

isUserMode = 1;

return i; /\*创建进程成功\*/

}

}

return 0; /\*创建进程失败\*/

}

即先createProcess 再加载程序进内存

考虑到createProcess之后，OS会认为内存中已经有了一个process，是内核态，所以若恰好在2执行完毕后发生了时间中断，那么此时就会跳转到用户程序，但此时用户程序还没有载入，所以会发生错误，程序卡死在这个地方。

修改之后的如下：

/\*运行新程序\*/

int runProg(char \*filename)

{

int pid;

int tmp;

pid = requirePCB();/\*申请pcb块，失败返回-1\*/

if(pid != -1)

{

currentSeg = baseSeg + (pid - 1)\*SegLength;/\*确定程序的段地址\*/

tmp = loadProg(filename); /\*加载程序,失败返回-1\*/

if(tmp != -1)

{

createProcess(pid); /\*创建进程\*/

return pid;

}

else

{

pcb\_list[pid].used = 0; /\*失败，释放pcb\*/

return tmp;

}

}

else

return pid;

}

/\*申请PCB块\*/

int requirePCB()

{

int i;

for(i = 1; i < MAX\_NrPCB; i++)

{

if(pcb\_list[i].used == 0)

{

pcb\_list[i].used = 1;

return i;

}

}

return -1;

}

/\*创建新的进程\*/

int createProcess(int process\_id)

{

initPCB( &pcb\_list[process\_id] ,process\_id, currentSeg, SP\_OFF); /\*初始化pcb块\*/

}

逻辑如下：

先申请pcb块，申请成功后再加载用户程序，加载完毕无错误后，再createProcess

这样子保证了即使该过程被时间中断打断，也不会发生错误

由于修改了createProcess，相应的实验7中的fork函数也要修改成下面，逻辑是这样子的：先申请pcb块，成功了就申请栈段，申请栈段成功了就直接复制父进程的状态信息给子进程，返回：

/\*复制父进程的状态信息给子进程\*/

void copyFatherToSon(int father, int son)

{

pcb\_list[son].regImg.SP = pcb\_list[father].regImg.SP;

pcb\_list[son].regImg.GS = pcb\_list[father].regImg.GS;

pcb\_list[son].regImg.FS = pcb\_list[father].regImg.FS;

pcb\_list[son].regImg.ES = pcb\_list[father].regImg.ES;

pcb\_list[son].regImg.DS = pcb\_list[father].regImg.DS;

pcb\_list[son].regImg.DI = pcb\_list[father].regImg.DI;

pcb\_list[son].regImg.SI = pcb\_list[father].regImg.SI;

pcb\_list[son].regImg.BP = pcb\_list[father].regImg.BP;

pcb\_list[son].regImg.BX = pcb\_list[father].regImg.BX;

pcb\_list[son].regImg.DX = pcb\_list[father].regImg.DX;

pcb\_list[son].regImg.CX = pcb\_list[father].regImg.CX;

pcb\_list[son].regImg.IP = pcb\_list[father].regImg.IP;

pcb\_list[son].regImg.CS = pcb\_list[father].regImg.CS;

pcb\_list[son].regImg.Flags = pcb\_list[father].regImg.Flags;

pcb\_list[son].fpid = father;

pcb\_list[son].pid = son;

}

/\*

1）在父进程中，fork返回新创建子进程的进程ID;

2）在子进程中，fork返回0;

3）如果出现错误，fork返回一个负值;

\*/

int fork() /\*-1表示分配子程序空间失败， -2表示分配栈空间失败 \*/

{

int stack\_id;

pid = requirePCB();/\*申请pcb块\*/

if(pid)

{

/\*申请 分配栈内存空间\*/

stack\_id = requrireStackMemory();

if(stack\_id == -1)

{

/\*分配栈空间失败\*/

pcb\_list[pid].used = 0;

pcb\_list[CurrentPCBno].regImg.AX = -1;

}

else

{

copyFatherToSon(CurrentPCBno, pid);/\*复制父进程的状态信息给子进程\*/

pcb\_list[pid].regImg.SS = SS\_baseSeg + stack\_id \* SS\_Len; /\*确定SS\*/

memcopy(pcb\_list[pid].regImg.SS,pcb\_list[CurrentPCBno].regImg.SS, SP\_OFF);/\*复制父进程的栈区给子进程\*/

pcb\_list[CurrentPCBno].regImg.AX = pid; /\*父进程返回子进程id\*/

pcb\_list[pid].regImg.AX = 0; /\*子进程返回0\*/

pcb\_list[pid].status = READY;

pcb\_list[pid].next = 0;

}

}

else

{

/\*表示分配子程序空间失败\*/

pcb\_list[CurrentPCBno].regImg.AX = -1;

}

}

现在可以在其他用户程序执行的时候添加其他程序，而且程序加载的时候没有出现问题。

实验九是在上面修改的基础上做的。所以可以直接在实验九的报告中查看结果。

心得体会:

之前遇到上述修改二中的问题时，有点不知所措，后来随着实验的深入，对于发现这些问题的敏感度提高了，特别是在对于进程的执行顺序不可控的情况下，会有很多很多线性执行单一程序不会遇到的问题。这一次更正了这一个问题，更加深刻的理解了os的互斥和同步问题，而且深刻知道了这些问题控制和发现的难度。