> 最后修改: 2020-3-30

1. 实验要求

本实验主要完成自制简单操作系统的进程管理功能，通过实现一个简单的任务调度，介绍**基于时间中断进行进程切换**完成任务调度的全过程，主要涉及到`fork`、`exec`、`sleep`、`exit`等库函数和对应的处理例程实现

1.1. 面向测试用例编程

先看一看这次用户程序代码`lab3/app/main.c`是什么

```
#include "lib.h"
#include "types.h"
int data = 0;
int uEntry(void) {
  int ret = fork();
  int i = 8;
  if (ret == 0) {
    data = 2;
    while(i != 0) {
      i --;
      printf("Child Process: Pong %d, %d;\n", data, i);
      sleep(128);
    }
    exit();
  }
  else if (ret != -1) {
    data = 1;
    while(i != 0) {
      i --;
      printf("Father Process: Ping %d, %d;\n", data, i);
      sleep(128);
    }
    exec("/usr/print\0", 0);
    exit();
  }
  while(1);
  return 0;
}
```

其中`fork`、`exec`、`sleep`、`exit`都是以前没有涉及的函数，所以，第一步就是在`lab3/lib/syscall.c`中补上相应的库函数

1.2. 实现对应的系统调用

通过实验2`printf`的锻炼，相信大家都对系统调用的流程有了一定的了解，所以接下来为了让这些库函数起作用，需要实现对应的系统调用的处理例程

2. 相关资料

2.1. 实验2回顾

实验2的内容中，kernel启动之后会按`lab2/kernel/main.c`写到的进行一系列的初始化操作，然后调用`load_UMain`加载用户程序，并通过`iret`进入用户态，执行用户程序

详细看一下

```
uint32_t loadUMain(void) {
  int i = 0;
  int phoff = 0x34; // program header offset
  int offset = 0x1000; // .text section offset
  uint32_t elf = 0x200000; // physical memory addr to load
  uint32_t uMainEntry = 0x200000;
  Inode inode;
  int inodeOffset = 0;
  readInode(&sBlock, &inode, &inodeOffset, "/boot/initrd");
  for (i = 0; i < inode.blockCount; i++) {
    readBlock(&sBlock, &inode, i, (uint8_t *)(elf + i * sBlock.blockSize));
  }
  uMainEntry = ((struct ELFHeader *)elf)->entry; // entry address of the
program
  phoff = ((struct ELFHeader *)elf)->phoff;
  offset = ((struct ProgramHeader *)(elf + phoff))->off;
  for (i = 0; i < 200 * 512; i++) {
    *(uint8_t *)(elf + i) = *(uint8_t *)(elf + i + offset);
  }
  enterUserSpace(uMainEntry);
}
```

实际上了解elf文件结构的同学很容易能发现上面这一段代码有很大的问题，这一段代码应该对应elf文件的加载，实际上现代操作系统的加载过程更加复杂

2.1.1. 加载ELF文件

ELF(Executable and Linking Format)是一种对象文件格式(Object files), 而这里的对象文件格式又包括三种

- 可重定位的对象文件(Relocatable file)——.o 文件
- 可执行的对象文件(Executable file)
- 可被共享的对象文件(Shared object file)——.so 动态库文件

在本课程的实验中不涉及.so动态库文件，我们就拿lab2 `make` 所产生的文件举例，在app目录下，mian.o和uMain.elf都是 ELF 对象文件, 如果想要知道都属于哪类文件, 可以使用 file 命令查看

```
$file main.o uMain.elf
main.o:    ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not
stripped
uMain.elf: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV),
statically linked, not stripped
```

其实以上内容都是我瞎编的, 你可以自己亲自尝试一下, 到底一个 ELF 对象文件哪类.

ELF头结构

ELF 文件由4部分组成, 分别为ELF 头(ELF Header), 程序头(Program Header Table), 节(Section)和节头(Section Header Table). 虽然不是每个 ELF 文件都包含这4部分内容, 但是 ELF 头的位置是固定的, 也是由 ELF 头中的数据来决定其他部分的组成. 下面是一张通用的 ELF 文件结构图

```
+----------------------+
|      ELF Header      |
+----------------------+
|   Program Header 0   |
+----------------------+
|   Program Header 1   |
+----------------------+
|         ...          |
+----------------------+
|      Section 0       |
+----------------------+
|      Section 1       |
+----------------------+
|         ...          |
+----------------------+
|   Section Header 0   |
+----------------------+
|   Section Header 1   |
+----------------------+
|         ...          |
+----------------------+
```

可以看到 ELF 头出现在最前面, 而其他部分的长度都是可变的, 只有 ELF 是固定的. 我们给出 ELF 头的结构

```
struct ELFHeader {
  unsigned int   magic;
  unsigned char  elf[12];
  unsigned short type;
  unsigned short machine;
  unsigned int   version;
  unsigned int   entry;       //程序入口
```

```
  unsigned int   phoff;       //Program Header 偏移
  unsigned int   shoff;       //Section Header 偏移
  unsigned int   flags;
  unsigned short ehsize;      //此头长度
  unsigned short phentsize;   //程序头长度
  unsigned short phnum;       //程序头数
  unsigned short shentsize;   //节头长度
  unsigned short shnum;       //节头数
  unsigned short shstrndx;
};
```

我们通过一个例子来理解一个结构的内容, 我们用 GNU binutils 的readelf 工具查看一个 .o 文件的ELF头内容

```
$readelf -h uMain.elf
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x0
  Start of program headers:          52 (bytes into file)
  Start of section headers:          8884 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         3
  Size of section headers:           40 (bytes)
  Number of section headers:         10
  Section header string table index: 9
```

ELF 的开头四个字节(也就是结构中的 magic 和 12比特的 elf), 是机器无关的固定内容, 包括 `magic=` 0x7f 以及三个EFL 字符.

ELF文件的加载主要和Program Header有关，所以还要看一下其结构

```
struct ProgramHeader {
  unsigned int type;
  unsigned int off;       //这个段第一个字节在文件中的偏移
  unsigned int vaddr;     //这个段第一个字节在内存中的虚拟地址
  unsigned int paddr;     //应当加载到的物理内存地址
  unsigned int filesz;    //这个段在 elf 文件中的长度
  unsigned int memsz;     //这个段在内存中的长度
  unsigned int flags;
  unsigned int align;
};
```

程序头描述的是这个段在文件中的位置和大小以及在内存中的位置和大小，同样可以用readelf查看

```
$readelf -l uMain.elf

Elf file type is EXEC (Executable file)
Entry point 0x0
There are 3 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  LOAD           0x001000 0x00000000 0x00000000 0x008f0 0x008f0 R E 0x1000
  LOAD           0x002000 0x00002000 0x00002000 0x0000c 0x00010 RW  0x1000
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x10

 Section to Segment mapping:
  Segment Sections...
   00     .text .rodata .eh_frame
   01     .got.plt .bss
   02
```

实际操作系统需要把Program Headers中type为LOAD即`0x1`的段加载到对应的物理内存

实际操作时需要**找出每一个需要加载的段的off, vaddr, filesz和memsz这些参数. 其中相对文件偏移off指出相应段的内容从ELF文件的第off字节开始, 在文件中的大小为filesz, 它需要被分配到以vaddr为首地址的虚拟内存位置, 在内存中它占用大小为memsz. 也就是说, 这个段使用的内存就是[vaddr, vaddr + memsz)这一连续区间, 然后将段的内容从ELF文件中读入到这一内存区间, 并将[vaddr + filesz, vaddr + memsz)对应的物理区间清零**.

2.1.2. 进入用户态

实验2通过enterUserSpace跳转到entry处的代码进行用户态执行

```
void enterUserSpace(uint32_t entry) {
  uint32_t EFLAGS = 0;
  asm volatile("pushl %0":: "r" (USEL(SEG_UDATA))); // push ss
  asm volatile("pushl %0":: "r" (0x200000)); //TODO push esp, further
modification
```

```
  asm volatile("pushfl"); //push eflags, sti
  asm volatile("popl %0":"=r" (EFLAGS));
  asm volatile("pushl %0"::"r"(EFLAGS|0x200));
  asm volatile("pushl %0":: "r" (USEL(SEG_UCODE))); // push cs
  asm volatile("pushl %0":: "r" (entry)); //TODO push eip, further
modification
  asm volatile("pushl %0":: "r" (USEL(SEG_UDATA)));
  asm volatile("pushl %0":: "r" (USEL(SEG_UDATA)));
  asm volatile("pushl %0":: "r" (USEL(SEG_UDATA)));
  asm volatile("pushl %0":: "r" (USEL(SEG_UDATA)));
  asm volatile("popl %gs");
  asm volatile("popl %fs");
  asm volatile("popl %es");
  asm volatile("popl %ds");
  asm volatile("iret");
}
```

这一段是先在栈中设置好`iret`返回需要的eip，cs，eflags，esp，ss，然后通过栈设置gs，fs，es，ds。因为实验2默认只有一个用户进程，所以可以省略多进程利用`iret`切换时的额外操作，这个在后面**2.4.2.**节详细说明

2.2. 启动时钟源

实验2已经使用以下代码对8253可编程计时器进行设置，使得8253以频率`HZ`产生时间中断信号发送给8259A可编程中断控制器；若依照代码框架`lab2/kernel/kernel/i8259.c`中给出的配置示例对8259A进行设置，时间中断的中断向量为`0x20`

```
#define TIMER_PORT 0x40
#define FREQ_8253 1193182
#define HZ 100

void initTimer() {
  int counter = FREQ_8253 / HZ;
  outByte(TIMER_PORT + 3, 0x34);
  outByte(TIMER_PORT + 0, counter % 256);
  outByte(TIMER_PORT + 0, counter / 256);
}
```

在实验2中，`timerHandle`为空，到实验3就需要正式进行处理，时钟中断处理例程能直接管理进程切换

2.3. 分段内存管理

实验2中，默认操作系统一个内核进程，一个用户进程。内核在链接时通过`-Ttext 0x100000`指定程序的起始地址，同时在加载时将内核进程加载到内存的`0x100000`处，设置GDT中内核代码段和数据段的基址为`0x0`，值得一提的是内核`ss`也指向数据段，设置`esp`为`0x200000`；用户程序则不同，链接时指定程序起始地址为`0x0`，却加载到内存的`0x200000`，这时虚拟地址到物理地址就只能通过段选择子

和GDT进行转换，所以设置代码段和数据段的段基址为`0x200000`。

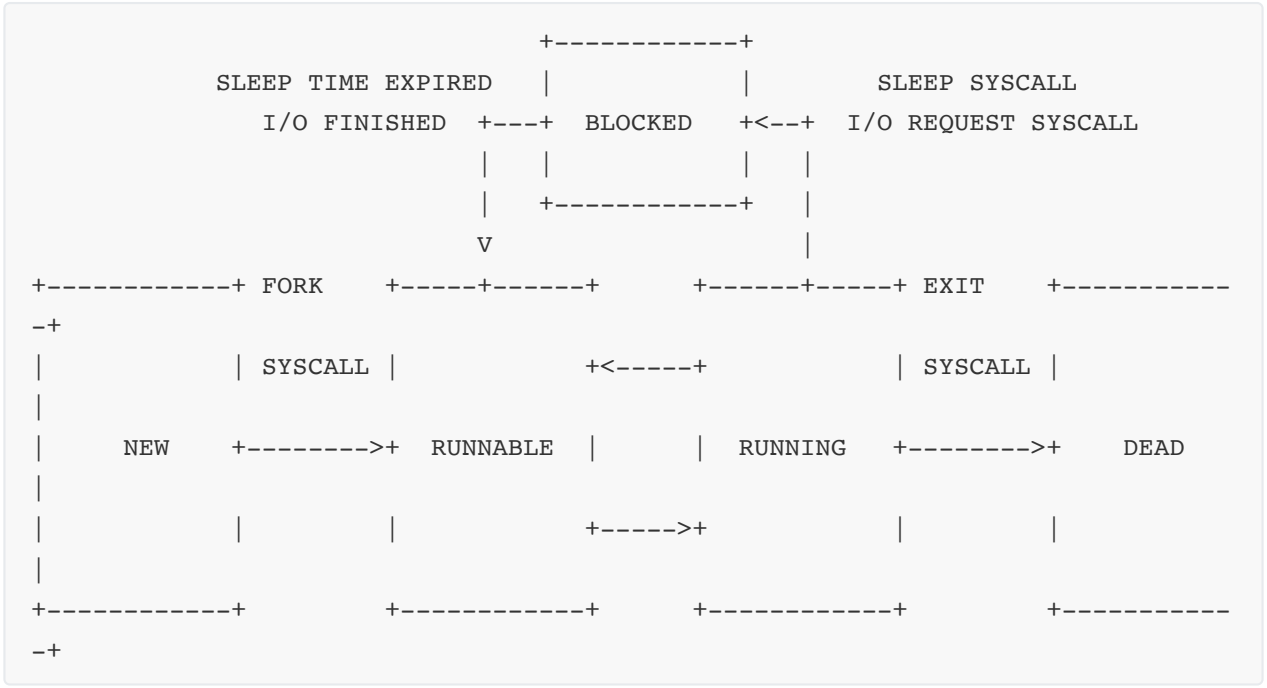
后续实验用户多进程寻址应该采用实验2用户进程的方式，也就是说不管有多少用户进程，我们默认链接时指定程序起始地址都是`0x0`，但是加载到不同的内存地址。在实验3中默认用户进程起始地址为`0x200000`，每个进程占用`0x100000`大小的内存。通过设置不同的段基址来隔离用户进程在内存中的位置，这样理论上可以通过切换段选择子的值进行用户进程的切换

```
+-------------------------+ 0xffffffff
|           ...           |
+-------------------------+ 0x00400000
|      app 1 stack        |
+-------------------------+
|          app 1          |
+-------------------------+ 0x00300000
|      app 0 stack        |
+-------------------------+
|          app 0          |
+-------------------------+ 0x00200000
|      kernel stack       |
+-------------------------+
|         kernel          |
+-------------------------+ 0x00100000
|           ...           |
+-------------------------+ 0x00000000
```

2.4. 进程

进程为操作系统资源分配的单位，每个进程都有独立的地址空间（代码段、数据段），独立的堆栈，独立的进程控制块；以下为一个广义的进程生命周期中的状态转换图

- 进程由其父进程利用`FORK`系统调用创建，则该进程进入`RUNNABLE`状态
- 时间中断到来，`RUNNABLE`状态的进程被切换到，则该进程进入`RUNNING`状态
- 时间中断到来，`RUNNING`状态的进程处理时间片耗尽，则该进程进入`RUNNABLE`状态
- `RUNNING`状态的进程利用`SLEEP`系统调用主动阻塞；或利用系统调用等待硬件I/O，则该进程进入`BLOCKED`状态
- 时间中断到来，`BLOCKED`状态的进程的`SLEEP`时间片耗尽；或外部硬件中断表明I/O完成，则该进程进入`RUNNABLE`状态
- `RUNNING`状态的进程利用`EXIT`系统调用主动销毁，则该进程进入`DEAD`状态



`EXEC`因为不涉及进程生命周期的状态转换，所以不做说明

2.4.1. 进程控制块

在本课程和其它课程中，大家应该对进程控制块有了一定的了解，实验3开始操作系统提供对多进程的支持，简单介绍一下lab3中的进程控制块

```
struct ProcessTable {
  uint32_t stack[MAX_STACK_SIZE];   // 内核堆栈
  struct TrapFrame regs;            // 陷阱帧，保存上下文
  uint32_t stackTop;                // 保存内核栈顶信息
  uint32_t prevStackTop;            // 中断嵌套时保存待恢复的栈顶信息
  int state;      // 进程状态：STATE_RUNNABLE、STATE_RUNNING、STATE_BLOCKED、
STATE_DEAD
  int timeCount;                    // 当前进程占用的时间片
  int sleepTime;                    // 当前进程需要阻塞的时间片
  uint32_t pid;                     // 进程的唯一标识
  char name[32];                    // not used
};
```

其中`TrapFrame`相对实验2也稍微有点不同

```
struct TrapFrame {
  uint32_t gs, fs, es, ds;
  uint32_t edi, esi, ebp, xxx, ebx, edx, ecx, eax;
  uint32_t irq, error;
  uint32_t eip, cs, eflags, esp, ss;
};
```

简单说就是将`int`指令会保存的ss、esp、eflags、cs、eip、error number都做为陷阱帧的部分，也就是说陷阱帧保存了陷入中断时**硬件**保存的内容

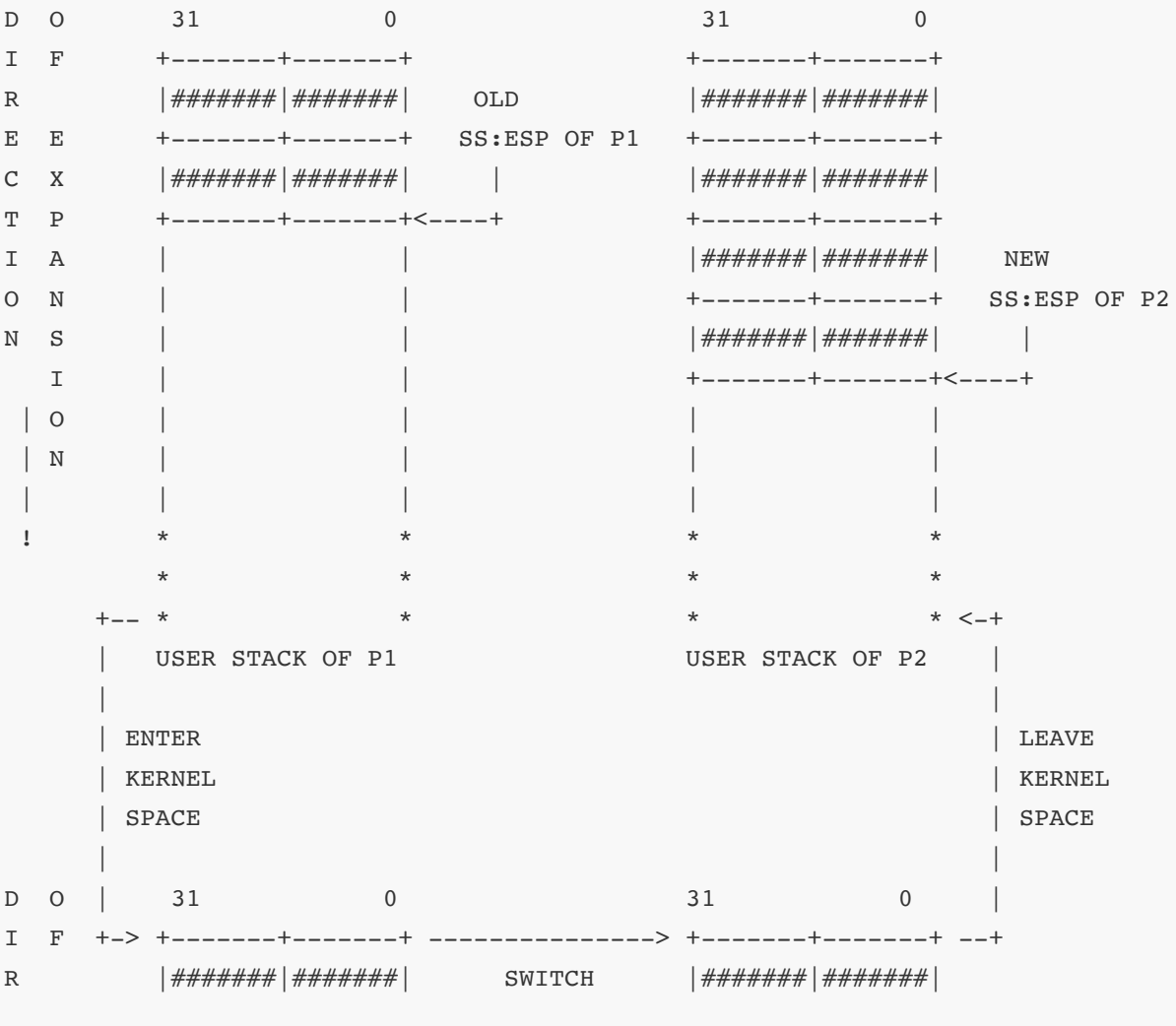
在实验中，采用线性表的方式组织pcb，也就是将pcb以数组形式连续存放，为了简单起见，可以将pcb的pid设为其索引。内核进程会占据0号pcb，剩下的分配给用户进程。同样为了简单，我们默认每个pcb对应进程的内存空间固定，`pcb[i]`对应的内存起始地址为`(i + 1) * 0x100000`，大小为`0x100000`

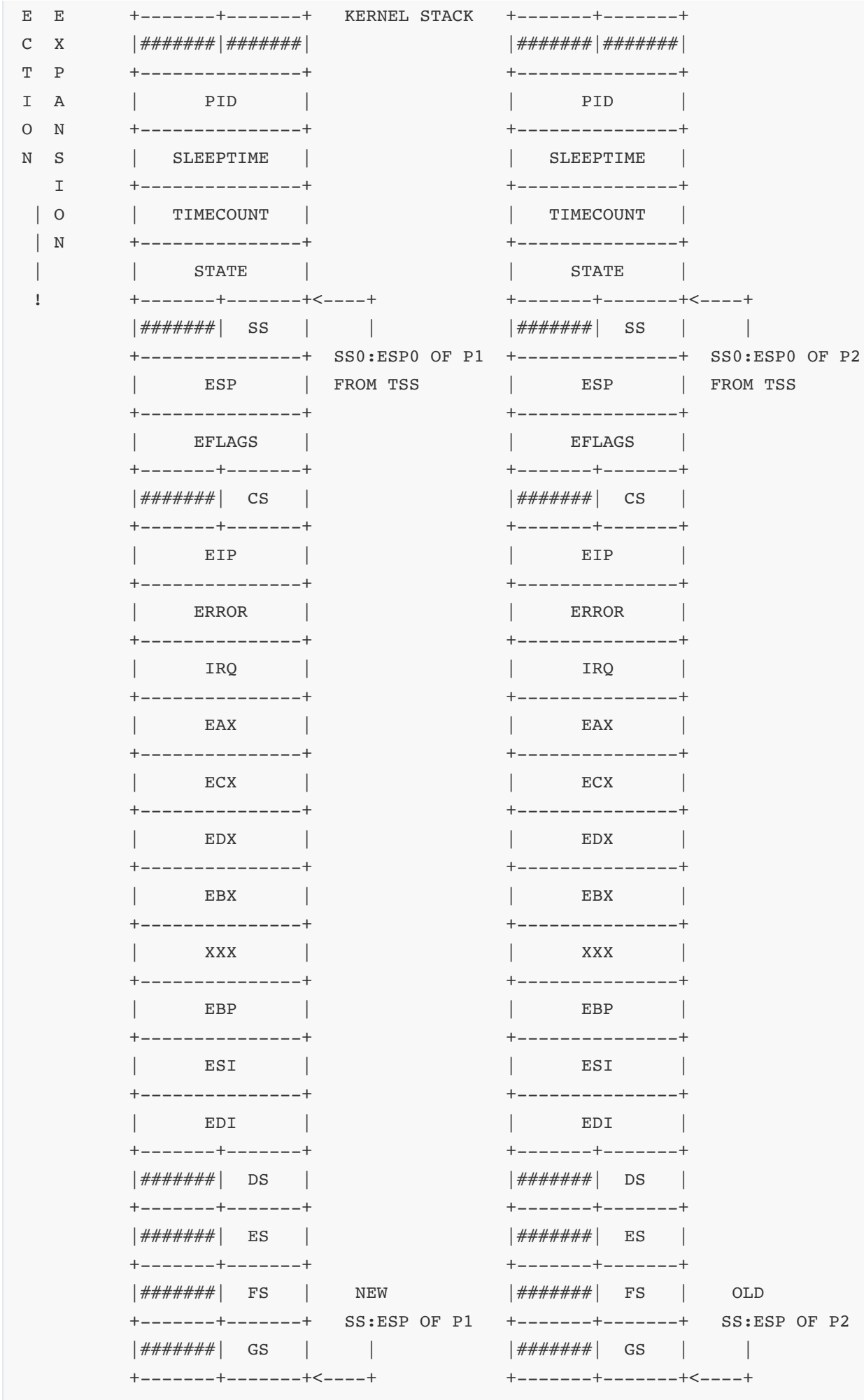
```
struct ProcessTable pcb[MAX_PCB_NUM];
```

2.4.2. 进程切换与堆栈切换

下图为产生时间中断后，时间中断处理程序为两个用户态进程P1、P2进行进程切换的过程中，堆栈切换的简单图示

1. 进程P1在用户态执行，8253可编程计时器产生时间中断
2. 依据TSS中记录的进程P1的`SS0:EPS0`，从P1的用户态堆栈切换至P1的内核堆栈，并将P1的现场信息压入内核堆栈中，跳转执行时间中断处理程序
3. 进程P1的处理时间片耗尽，切换至就绪状态的进程P2，并从当前P1的内核堆栈切换至P2的内核堆栈
4. 从进程P2的内核堆栈中弹出P2的现场信息，切换至P2的用户态堆栈，从时间中断处理程序返回执行P2




E E
C X
T P
I A
O N
N S
I
O
N
KERNEL STACK
PID
SLEEPTIME
TIMECOUNT
STATE
SS
ESP
EFLAGS
CS
EIP
ERROR
IRQ
EAX
ECX
EDX
EBX
XXX
EBP
ESI
EDI
DS
ES
FS
GS
SS0:ESP0 OF P1
FROM TSS
NEW
SS:ESP OF P1
SS0:ESP0 OF P2
FROM TSS
OLD
SS:ESP OF P2

```
          |               |                   |               |
          *               *                   *               *
          *               *                   *               *
          *               *                   *               *
          KERNEL STACK OF P1                  KERNEL STACK OF P2
```

在 lab2里，只有一个用户进程，我们将tss中的ss0:esp0设成了一个固定的值，这样用户进程的内核栈固定。但是在 lab3里，你将面临堆栈切换的问题，也就是每个用户进程的内核堆栈也是不一样的，所以**每次切换进程时需要将tss的esp0设置对应用户进程的内核堆栈位置**

说了那么多进程切换的细节，那要如何选择下一个就绪进程呢？这就涉及到调度策略，目前只需要采用轮转调度(Round Robin)的策略即可，即依次调度进程1, 进程2, ..., 进程n, 进程1...

Linux采用多级队列调度的策略, 为每一个进程设定一个优先级, 调度时选择所有就绪进程中优先级最高的进程进行调度, 这样可以保证一些请求能够尽快得到响应. 有兴趣的同学可以在以后实现这个调度策略

2.4.3. 内核IDLE进程

若没有处于`RUNNABLE`状态的进程可供切换，则需要切换至以下内核IDLE进程，该进程调用`waitForInterrupt()`执行`hlt`指令，`hlt`会使得CPU进入暂停状态，直到外部硬件中断产生，在实验中不需要手动创建IDLE进程，还记得系统启动的执行流吗？执行流在bootloader中加载并跳转到内核，然后执行一系列的初始化工作，等到初始化结束后将会打开中断，此时执行流摇身一变，成为了实验中的IDLE进程，等待中断的到来

```
static inline void waitForInterrupt() {
  asm volatile("hlt");
}
...
while(1) {
  waitForInterrupt();
}
```

2.4.4. 系统调用

fork系统调用用于创建子进程，内核需要为子进程分配一块独立的内存，将父进程的地址空间、用户态堆栈完全拷贝至子进程的内存中，并为子进程分配独立的进程控制块，完成对子进程的进程控制块的设置

若子进程创建成功，则对于父进程，该系统调用的返回值为子进程的`pid`，对于子进程，其返回值为`0`；若子进程创建失败，该系统调用的返回值为`-1`

```
pid_t fork();
```

sleep系统调用用于进程主动阻塞自身，内核需要将该进程由`RUNNING`状态转换为`BLOCKED`状态，设置该进程的`SLEEP`时间片，并切换运行其他`RUNNABLE`状态的进程

```
int sleep(uint32_t time);
```

exit系统调用用于进程主动销毁自身，内核需要将该进程由`RUNNING`状态转换为`DEAD`状态，回收分配给该进程的内存、进程控制块等资源，并切换运行其他`RUNNABLE`状态的进程

```
int exit();
```

exec系统调用用于以新的进程去代替原来的进程，但进程的PID保持不变。因此，可以这样认为，exec系统调用并没有创建新的进程，只是替换了原来进程上下文的内容。原进程的代码段，数据段，堆栈段被新的进程所代替

```
int exec(const char *filename, char * const argv[]);
```

因为我们的系统没有环境变量，所以exec的参数也很简单，第一个参数就是可执行文件的路径，第二个参数是给可执行文件传递的参数，注意当需要传递参数时，`argv[0]`是`filename`，`argv[1]`才是第一个参数，`argv`列表最后一个必须是`NULL`。与一般情况不同，exec函数执行成功后不会返回，只有调用失败了，它们才会返回一个`-1`，从原程序的调用点接着往下执行

2.5. 中断嵌套与临界区

由于系统调用的处理时间往往很长，为保证进程调度的公平性，需要在系统调用中开启外部硬件中断，以便当前进程的处理时间片耗尽时，进行进程切换；由于可以在系统调用中进行进程切换，因此可能会出现多个进程并发地处理系统调用，对共享资源（例如内核的数据结构，视频显存等等）进行竞争，例如以下场景

1. 进程P1在内核态处理系统调用，处理视频显存，此时外部硬件中断开启
2. 8253可编程计时器产生一个时间中断
3. 在内核态处理系统调用的进程P1将现场信息压入P1的内核堆栈中，跳转执行时间中断处理程序
4. 进程P1的处理时间片耗尽，切换至就绪状态的进程P2，并从当前P1的内核堆栈切换至P2的内核堆栈
5. 从进程P2的内核堆栈中弹出P2的现场信息，从时间中断处理程序返回执行P2
6. 进程P2在内核态处理系统调用，处理视频显存，与进程P1形成竞争

在以下系统调用内核处理函数中利用`int $0x20`指令主动陷入时间中断来模拟以上场景

```
void syscallPrint(struct StackFrame *sf) {
  ...
  for (i = 0; i < size; i++) {
    asm volatile("movb %%es:(%1), %0":"=r"(character):"r"(str+i));
    if(character == '\n') {
      displayRow ++;
      displayCol = 0;
      if(displayRow == 25) {
        displayRow = 24;
        displayCol = 0;
        scrollScreen();
      }
    }
    else {
```

```
      data = character | (0x0c << 8);
      pos = (80*displayRow + displayCol) * 2;
      asm volatile("movw %0, (%1)"::"r"(data),"r"(pos+0xb8000));
      displayCol ++;
      if(displayCol == 80) {
        displayRow ++;
        displayCol = 0;
        if(displayRow == 25){
          displayRow = 24;
          displayCol = 0;
          scrollScreen();
        }
      }
    }
    asm volatile("int $0x20"); // 测试系统调用嵌套时间中断
  }
  ...
}
```

对编译生成的内核ELF文件进行反汇编，得到以下代码

```
001005dc <syscallPrint>:
  ...
  100606: or  $0xc,%ah
  100609: lea (%ecx,%ecx,4),%edx
  10060c: shl $0x4,%edx
  10060f: add 0x102404,%edx
  100615: lea 0xb8000(%edx,%edx,1),%edx
  10061c: mov %ax,(%edx)
  10061f: mov 0x102404,%eax
  100624: inc %eax
  100625: mov %eax,0x102404
  10062a: cmp $0x50,%eax
  10062d: je  10063d
  10062f: int $0x20
  100631: inc %ebx
  100632: cmp %esi,%ebx
  100634: je  10066c
  100636: mov %es:(%ebx),%al
  100639: cmp $0xa,%al
  10063b: jne 100606
  10063d: inc %ecx
  10063e: mov %ecx,0x102408
  100644: mov $0x0,0x102404
  10064e: cmp $0x19,%ecx
  100651: jne 10062f
  ...
00102404 <displayCol>:
  102404: 00 00
```

```
  ...
00102408 <displayRow>:
  102408: 00 00
  ...
```

考虑以下场景

- P1从时钟中断返回，顺序执行
  `0x100631`、`0x100632`、`0x100634`、`0x100636`、`0x100639`、`0x10063b`、
  `0x10063d`、`0x10063e`、`0x100644`、`0x10064e`、`0x100651`、`0x10062f`，
  再次陷入时间中断，切换至P2
- P2从时间中断返回，顺序执行
  `0x100631`、`0x100632`、`0x100634`、`0x100636`、`0x100639`、`0x10063b`、
  `0x100606`、`0x100609`、`0x10060c`、`0x10060f`、`0x100615`、`0x10061c`
- 全局变量`displayRow`的更新产生一致性问题

多个进程并发地进行系统调用，对共享资源进行竞争可能会产生一致性问题，带来未知的BUG；因此，在系统调用过程中，对于临界区的代码不宜开启外部硬件中断，而对于非临界区的代码，则可以开启外部硬件中断，允许中断嵌套

3. 解决思路

3.1. 完成库函数

这一部分算是对实验2的复习，我们在`lab3/lib/syscall.c`中留下了四个库函数待完成，你需要调用`syscall`完善库函数，**难度0**

3.2. 时钟中断处理

实验3的主要内容是进程管理，在现阶段的操作系统中，进程切换有两种情况，一个是sleep相关（阻塞或恢复）；一个是进程时间片用完切换到下一个进程，其中sleep又可以转化为时间片用完，所以时钟中断处理`timerHandle`处是一个绝佳的进行进程切换的场所。时钟中断功能：

1. 遍历pcb，将状态为STATE_BLOCKED的进程的sleepTime减一，如果进程的sleepTime变为0，重新设为STATE_RUNNABLE
2. 将当前进程的timeCount加一，如果**时间片用完（timeCount==MAX_TIME_COUNT）且有其它状态为STATE_RUNNABLE的进程**，切换，否则继续执行当前进程

内核IDLE进程需不需要特殊考虑都随便你，这里提供一份进程切换的代码

```
    tmpStackTop = pcb[current].stackTop;
    pcb[current].stackTop = pcb[current].prevStackTop;
    tss.esp0 = (uint32_t)&(pcb[current].stackTop);
    asm volatile("movl %0, %%esp"::"m"(tmpStackTop)); // switch kernel stack
    asm volatile("popl %gs");
    asm volatile("popl %fs");
    asm volatile("popl %es");
    asm volatile("popl %ds");
    asm volatile("popal");
    asm volatile("addl $8, %esp");
    asm volatile("iret");
```

请**自行理解**代码含义，并用在合适的地方，当然这个**仅供参考**，有自己的想法也行

说到这里得提一下`irqHandle`对比lab2也增加了部分保存与恢复的内容，同学们自行理解

```
 void irqHandle(struct TrapFrame *tf) { // pointer tf = esp
  asm volatile("movw %%ax, %%ds"::"a"(KSEL(SEG_KDATA)));

+ uint32_t tmpStackTop = pcb[current].stackTop;
+ pcb[current].prevStackTop = pcb[current].stackTop;
+ pcb[current].stackTop = (uint32_t)tf;

  switch(tf->irq) {
    case -1:
      break;
    case 0xd:
      GProtectFaultHandle(tf); // return
      break;
    case 0x20:
      timerHandle(tf);         // return or iret
      break;
    case 0x21:
      keyboardHandle(tf);      // return
      break;
    case 0x80:
      syscallHandle(tf);       // return
      break;
    default:assert(0);
  }

+ pcb[current].stackTop = tmpStackTop;
 }
```

`timerHandle`实现，**难度2**

3.3. 系统调用例程

3.3.1. syscallFork

syscallFork要做的是在寻找一个空闲的pcb做为子进程的进程控制块，将父进程的资源复制给子进程。如果没有空闲pcb，则fork失败，父进程返回-1，成功则子进程返回0，父进程返回子进程pid

在处理fork时有以下几点注意事项：

1. 代码段和数据段可以按照**2.4.1.**节最后的说明进行完全拷贝
2. pcb的复制时，需要考虑哪些内容可以直接复制，哪些内容通过计算得到，哪些内容和父进程无关
3. 返回值放在哪

提示：`initProc`中有初始化`pcb[0]`和`pcb[1]`的经验可供参考，**难度4**

3.3.2. syscallSleep

将当前的进程的sleepTime设置为传入的参数，将当前进程的状态设置为STATE_BLOCKED，然后利用

```
asm volatile("int $0x20");
```

模拟时钟中断，利用`timerHandle`进行进程切换

需要注意的是判断传入参数的合法性，**难度0**

3.3.3. syscallExit

将当前进程的状态设置为STATE_DEAD，然后模拟时钟中断进行进程切换，**难度0**

3.3.4. syscallExec

exec分三步走：

1. 读取传入的文件名（可以参考syscallPrint）
2. 加载对应文件到内存（完善loadElf）
3. 第2步成功，设置eip，执行加载的用户进程；如果第2步失败，设置返回值，继续执行原进程

loadElf在`lab3/kernel/kernel/kvm.c`中，定义如下，需要完成，至于loadElf需要做什么，参看**2.1.1.**节

```
int loadElf(const char *filename, uint32_t physAddr, uint32_t *entry);
```

`filename`是待加载的文件，`physAddr`是加载文件的物理内存起始地址，`entry`是用来返回加载文件的入口的，所以是传址。如果加载不成功（文件格式不对，读取文件出错等）loadElf会返回-1，成功返回0。注意虚拟内存地址到物理内存地址的转换，**难度4**

选做：**2.4.4.**节说过exec还可以传递参数，有兴趣的同学可以尝试实现

3.4. 选做：中断嵌套

pcb中提供的数据是支持嵌套中断的，但是在前面的实验中，并没有要求支持嵌套中断，如果你觉得你完成了嵌套中断，这里提供一个可用的方式测试，下面这段代码是在syscallFork中进行内存拷贝的代码

```
    for (j = 0; j < 0x100000; j++) {
      *(uint8_t *)(j + (i + 1) * 0x100000) = *(uint8_t *)(j + (current + 1) *
0x100000);
    }
```

代码的作用就是将current进程的内存空间拷贝到进程i的空间，我们可以做如下操作：

```
    enableInterrupt();
    for (j = 0; j < 0x100000; j++) {
      *(uint8_t *)(j + (i + 1) * 0x100000) = *(uint8_t *)(j + (current + 1) *
0x100000);
    }
    disableInterrupt();
```

这样拷贝内存空间时就开启了嵌套中断，可惜由于cpu速度过快，可能来不及嵌套中断，代码就执行完了，为了测试充分，需要手动模拟时钟中断

```
    enableInterrupt();
    for (j = 0; j < 0x100000; j++) {
      *(uint8_t *)(j + (i + 1) * 0x100000) = *(uint8_t *)(j + (current + 1) *
0x100000);
      asm volatile("int $0x20"); //XXX Testing irqTimer during syscall
    }
    disableInterrupt();
```

如果这样你还能完成通过测试，就完成了实验3对嵌套中断的要求

3.5. 选做：临界区

自行编写测试用例，自行设计输出，验证**2.5.**节全局变量`displayRow`的更新一致性问题

4. 代码框架

本次实验为了配合exec调用，我们还在文件系统中加入了一个新的用户程序`app_print`将实验2的测试用例单独分出来：

```
lab3
├── Makefile
├── app
├── app_print        // 新的用户程序
│   ├── Makefile
│   └── main.c
├── bootloader
├── kernel
├── lib
└── utils
```

看一下如何在文件系统加入新的用户程序，从app_print中的Makefile开始

```
CC = gcc
LD = ld

CFLAGS = -m32 -march=i386 -static \
   -fno-builtin -fno-stack-protector -fno-omit-frame-pointer \
   -Wall -Werror -O2 -I../lib
LDFLAGS = -m elf_i386

UCFILES = $(shell find ./ -name "*.c")
LCFILES = $(shell find ../lib -name "*.c")
UOBJS = $(UCFILES:.c=.o) $(LCFILES:.c=.o)

app_print.bin: $(UOBJS)
  $(LD) $(LDFLAGS) -e main -Ttext 0x00000000 -o app_print.elf $(UOBJS)

clean:
  rm -rf $(UOBJS) app_print.elf
```

这一部分只负责将app_print编译并链接

然后需要修改genFS，使得其拷贝两个文件到文件系统

```
    stringCpy(argv[2], srcFilePath, NAME_LENGTH - 1);
    stringCpy("/usr/print", destFilePath, NAME_LENGTH - 1);
    cp(driver, srcFilePath, destFilePath);
```

简单来说就是在`lab3/utils/genFS/main.c`中加入以上代码

在调用genFS的地方，将编译生成的新用户程序也传进去

```
--- lab2/app/Makefile 2020-03-05 22:43:10.000000000 +0800
+++ lab3/app/Makefile 2020-03-07 21:00:26.000000000 +0800
@@ -16,8 +16,9 @@
  $(LD) $(LDFLAGS) -e uEntry -Ttext 0x00000000 -o uMain.elf $(UOBJS)
  @#objcopy -S -j .text -j .rodata -j .eh_frame -j .data -j .bss -O binary
uMain.elf uMain.bin
  @#objcopy -O binary uMain.elf uMain.bin
- @../utils/genFS/genFS uMain.elf
+ @cp ../app_print/app_print.elf ./app_print.elf
+ @../utils/genFS/genFS uMain.elf app_print.elf

 clean:
  @#rm -rf $(UOBJS) uMain.elf uMain.bin
- rm -rf $(UOBJS) uMain.elf fs.bin
+ rm -rf $(UOBJS) uMain.elf app_print.elf fs.bin
```

最后就是在整个项目的Makefile中添加对新用户程序的支持

```
--- lab2/Makefile 2020-03-05 22:43:10.000000000 +0800
+++ lab3/Makefile 2020-03-07 21:00:26.000000000 +0800
@@ -4,6 +4,7 @@
  @cd utils/genFS; make
  @cd bootloader; make
  @cd kernel; make
+ @cd app_print; make
  @cd app; make
  @#cat bootloader/bootloader.bin kernel/kMain.bin app/uMain.bin > os.img
  @#cat bootloader/bootloader.bin kernel/kMain.bin app/uMain.elf > os.img
@@ -20,5 +21,6 @@
  @cd utils/genFS; make clean
  @cd bootloader; make clean
  @cd kernel; make clean
+ @cd app_print; make clean
  @cd app; make clean
  rm -f os.img
```

如果同学们想再增加其它可执行程序，可以按照上述流程添加，如果觉得上述过程过于复杂，欢迎大家修改框架，提供更好的项目组织形式

5. 作业提交

- 本次作业需提交可通过编译的实验相关源码与报告,提交前请确认`make clean`过.
- **请大家在提交的实验报告中注明你的邮箱, 方便我们及时给你一些反馈信息.**
- **学术诚信**: 如果你确实无法完成实验, 你可以选择不提交, 作为学术诚信的奖励, 你将会获得10%的分数; 但若发现抄袭现象, 抄袭双方(或团体)在本次实验中得0分.
- 请你在实验截止前务必确认你提交的内容符合要求(格式, 相关内容等), 你可以下载你提交的内容进行确认. 如果由于你的原因给我们造成了不必要的麻烦, 视情况而定, 在本次实验中你将会被扣除一定的分数, 最高可达50%.
- 实验不接受迟交，一旦迟交按**学术诚信**给分.
- 其他问题参看`index.pdf`中的**作业规范与提交**一章
- **本实验最终解释权归助教所有**

截止时间：2020-4-27 23:55:00