OS Practice

第2章　编程基础与代码调试

主讲教师：全红艳

计算机科学与技术学院

本次课程内容

1. 实践中编程语言
2. 程序编辑与编译
3. 程序的调试方法-gdb
4. 处理器硬件结构
5. 主要数据结构
6. QEMU仿真器及使用
7. 实验课程设计单元介绍

1. 实践中编程语言

- 实践教学的操作系统为Ubuntu
- 编程语言涉及C语言和汇编语言混合编程：
  - C语言编程主要基于GNU C的语法
  - 汇编常见格式：
    - GNU汇编采用AT&T汇编格式
    - Microsoft 汇编采用Intel格式

    项目实践中采用第一种GNU汇编进行。

AT&T汇编语法初步

- GCC，即Linux平台下的GNU C语言编译器，它使用AT&T汇编，与Intel汇编不同,基本语法：

1、源-目的 定序

AT&T语法和Intel语法在操作数的方向上是相反的。Intel语法的第一个操作数是目的地，第二个是来源地。然而AT&T语法的第一个操作数是来源地，第二的是目的地。也即：

Intel语法：Op-code dst src
AT&T语法: Op-code src dst

2、寄存器命名

寄存器名字要有前缀“%”。也即如果寄存器eax被使用，应写作%eax

5、内存操作数

AT&T语法里寄存器却改为“（”和“）”，例如：AT&T实例：section: disp(base, index, scale)。

3、立即操作数

AT&T立即操作数之前要有一个“$”符号。静态C语言变量也要有前缀“$”。在Intel语法里，十六进制常数 h作为后缀。在AT&T语法里用“0x”作为代替。所以，对于十六进制的数，我们看到一个“$”，然后一个“0x”，最后才是常数本身。

4、操作数大小

在AT&T语法里内存操作数的大小取决于操作码名字的最后一个字母。操作码后“b”，“w”和“l”分别指定byte（8字节长度），word（16字节长度）和long（32字节长度）的内存引用。

AT&T语法的“movb foo, %al”

AT&T汇编语法初步

- AT&T汇编，与Intel格式汇编的对比：

两种汇编基本指令的对比

区别点	AT&T	Intel
* 寄存器命名原则	%eax	eax
* 源/目的操作数顺序	movl %eax, %ebx	mov ebx, eax
* 常数/立即数的格式	movl $_value, %ebx	mov eax, _value
* 把value的地址放入eax寄存器	movl $0xd00d, %ebx	mov ebx, 0xd00d
* 操作数长度标识	movw %ax, %bx	mov bx, ax
* 寻址方式	immed32(basepointer, indexpointer, indexscale)	[basepointer + indexpointer × indexscale + imm32]

AT&T汇编语法初步

- AT&T汇编，与Intel格式汇编的对比：

```
* 直接寻址
        AT&T:  foo                          Intel: [foo]
        boo是一个全局变量。注意加上$是表示地址引用，不加是表示值引用。对于局部变量，可以通过堆栈指针引用

* 寄存器间接寻址
        AT&T: (%eax)                        Intel: [eax]

* 变址寻址
        AT&T: _variable(%eax)               Intel: [eax + _variable]
        AT&T: _array( ,%eax, 4)             Intel: [eax × 4 + _array]
        AT&T: _array(%ebx, %eax,8)          Intel: [ebx + eax × 8 + _array]
```

AT&T汇编实例

- AT&T汇编，与Intel格式汇编的对比：

Intel Code	AT&T Code
mov eax,1	movl $1,%eax
mov ebx,0ffh	movl $0xff,%ebx
int 80h	int $0x80
mov ebx, eax	movl %eax, %ebx
mov eax,[ecx]	movl (%ecx),%eax
mov eax,[ebx+3]	movl 3(%ebx),%eax
mov eax,[ebx+20h]	movl 0x20(%ebx),%eax
add eax,[ebx+ecx*2h]	addl (%ebx,%ecx,0x2),%eax
lea eax,[ebx+ecx]	leal (%ebx,%ecx),%eax
sub eax,[ebx+ecx*4h-20h]	subl -0x20(%ebx,%ecx,0x4),%eax

GCC内联汇编

- 指示编译器插入一个函数的代码到调用者的代码中，这样的函数就是内联函数
- 内联的方法减少了函数调用的额外开销
- gcc，即Linux平台下的GNU C语言编译器，使用AT&T语法
- 基本内联汇编的格式： **asm("assembly code");**

实例：

```
asm("movl %ecx %eax");/*将ecx的值传给eax了/
__asm__("movb %bh (%eax)");/*将bh的值传到eax指向的内存处*/
```

asm和__asm__两个关键字都可以使用。

例如：

```
__asm__ ("movl %eax, %ebx/n/t"
        "movl $56, %esi/n/t"
        "movl %ecx, $label(%edx,%ebx,$4)/n/t"
        "movb %ah, (%ebx)");
```

扩展汇编内联

- 在基本汇编内联里，使用了指令,而在扩展内联汇编里，能够指定操作数、输入寄存器、输出寄存器和clobbered registers

基本的格式如下:

```
asm ( assembler template
        : output operands              /* optional */
        : input operands               /* optional */
        : list of clobbered registers  /* optional */
      );
```

例如:

```
int a=10, b;
asm ("movl %1, %%eax;
     movl %%eax, %0;"
    :"=r"(b)       /* output */
    :"r"(a)        /* input */
    :"%eax"        /* clobbered register */
    );
```

2. 程序编辑与编译

- 程序编辑---采用gedit等编辑工具
- 编译器采用gcc, gcc安装命令： sudo apt-get install build-essential
- 简单实例：

```
#include <stdio.h>
int
main(void)
{
    printf("Hello, world!\n");
    return 0;
}
```

- 假设存为文件‘hello.c’，编译命令为：$ gcc -Wall hello.c -o hello
  - ✓ 使用-o 选项，可以指定输出文件的名字，否则输出文件默认为 ‘a.out’
  - ✓ 选项 –Wall 能开启编译器几乎所有常用警告
- 运行命令：$ ./hello

3. 程序的调试方法–gdb

- 安装gdb, 输入如下命令行 运行：

  sudo apt-get install build-essential

  build-essential选项可以包含gcc和gdb等工具C语言的开发包

- 我们的开发环境（虚拟）中已经具有该功能，因此安装步骤可以省略

- gdb 功能强大包括:

  - 设置断点、监视程序变量的值、程序的单步(step in/step over)执行、显示/修改变量的值、显示/修改寄存器、查看程序的堆栈情况、远程调试、调试线程

gdb调试器使用

- 输入命令，开始调试程序： **gdb Cfilename**

Cfilename 是编译后的可执行文件

gdb一些常用命令:

l <n>	输出第n行到n+9行的源代码
break <n>	在第n行设置断点
info break	查看断点信息
r	运行
n	单步执行
c	继续执行
p varName	输出变量值
q	退出

调试实例

- 输入命令： gdb execl_test1 即可进入到gdb跟踪状态

- 设置断点(break命令)：break命令（可以简写为b）可以用来在调试的程序中设置断点，该命令有如下几种形式：

（1） 使程序恰好在执行给定行之前停止。

（2） 使程序恰好在进入指定的函数之前停止。

（3）程序到达指定行或函数时停止

调试实例

使程序恰好在执行给定行之前停止

- (gdb) break 79

  # 在line 79行设置断点

- (gdb) break thread_init

  # 在thread_init函数开始运行时，停止

```
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/user/pintos/src/threads/build/kernel.o...done.
(gdb) break 79
Breakpoint 1 at 0xc0100431: file ../../threads/init.c, line 79.
(gdb) break thread_init
Breakpoint 2 at 0xc010096a: file ../../threads/thread.c, line 89.
(gdb)
```

删除断点(clear命令)

- (gdb) clear main
  # 删除main函数上的断点
- (gdb) clear 26
  # 删除26行的断点
- (gdb) clear debug.c:main
  # 删除debug.c文件中main函数上的断点

其他断点相关命令:

- gdb)info program: 来查看程序的是否在运行，进程号，被暂停的原因。
- (gdb)delete 断点号n: 删除第n个断点
- (gdb)disable 断点号n: 暂停第n个断点
- (gdb)enable 断点号n: 开启第n个断点

观察变量

- (gdb) watch i

  # 监视变量i，当i的值变化之后，gdb将停止程序运行

查看监视变量的内容，当所监视的变量发生变化时，gdb将停止程序运行，如下：

```
Breakpoint 5, main () at debug.c:33
33 i = 10;
```

gdb调试运行命令

- 先设置断点，再让程序运行

1. (gdb) run (Pintos)

   #运行待调试程序

2. (gdb) continue

   如果run之后在断点处停止，使用continue命令继续程序的执行

3. (gdb) step

   #使用step进行单步调试

- gdb调试运行命令

(gdb) info break：查看断点信息

(gdb) r：运行程序

(gdb) n/s：单步执行

(gdb) c：继续运行

(gdb) p 变量：打印变量的值

(gdb) bt：查看函数堆栈

(gdb) finish：退出函数

gdb单步调试及跳转

- 单步执行命令：

  1、n/next：遇到函数不进入到函数体内(step over)

  2、s/step：遇到函数进入到函数体内(step in)

```
(gdb) step
Warning:
Could not insert hardware watchpoint 4.
Could not insert hardware breakpoints:
You may have requested too many hardware breakpoints/watchpoints.

0xc010096b in thread_init () at ../../threads/thread.c:89
89      {
(gdb)
```

- 跳转　　(gdb) jump 5

  跳转执行程序到第5行：

gdb过程输出变量值

- 在程序中如何查看程序变量的值，gdb中可以使用print命令

1. (gdb) print p　　# p为结构体类型

   $2 = {x = 0, y = 0}

2. (gdb) print arr　　# arr为int数组

   $3 = {1, 2}

3. (gdb) print a　　# a为int类型

   $4 = 10

4. (gdb) print *ptr@5　#输出ptr数组前5个值

   # 其中int* ptr = (int*)malloc(5 * sizeof(int));

   $5 = {0, 0, 0, 0, 0}

5. (gdb) print x=4

   修改运行时候的变量值，意为把变量x值改为4

gdb其他命令

- (gdb) shell 命令行：执行shell命令行
- (gdb) set args 参数:指定运行时的参数
- (gdb) show args：查看设置好的参数
- (gdb) show paths:查看程序运行路径;
- (gdb) cd 相当于shell的cd;
- (gdb) pwd：显示当前所在目录
- (gdb) [Enter]：执行上次执行的命令
- (gdb) backtrace：显示当前调用函数堆栈中的函数

4. 处理器硬件结构

- 需要了解ucore运行的硬件环境：
  - 即处理器体系结构
  - 机器指令集（ucore的汇编代码）

x86-32硬件运行模式

- 80386有四种运行模式：
  - 实模式
    - 80386启动后处于实模式运行状态，在这种状态下软件可访问的物理内存空间不能超过1MB，且处理器在32位Intel 80386以上时，CPU 内存4GB管理性能无法发挥
  - 保护模式
    - 支持内存分页机制，提供了对虚拟内存的良好支持。支持多任务，还支持优先级机制。优先级一共分35个级别（0～34），操作系统运行在最高的优先级0上，可以实现数据安全共享，也可以隔离各个任务
  - SMM模式
  - 虚拟8086模式

x86-32的内存结构

- 分为物理地址和逻辑地址
- 80386是32位的处理器，即可以寻址的物理内存地址空间为2^32=4G字节
- 三个地址概念：
  - 物理地址：处理器提交到总线上用于访问计算机系统内存和外设的地址
  - 线性地址：线性地址空间是80386处理器通过段（Segment）机制控制下的形成的地址空间。段的起始地址和长度属性，用于程序隔离，实现内存空间保护
  - 逻辑地址
- 三种地址的关系：

段机制启动、页机制未启动：逻辑地址->段机制处理->线性地址=物理地址

段机制和页机制都启动：逻辑地址->段机制处理->线性地址->页机制处理->物理地址

x86-32硬件寄存器

- 80386的寄存器可以分为8组：
  - 通用寄存器
  - 段寄存器
  - 指令指针寄存器
  - 标志寄存器
  - 控制寄存器
  - 系统地址寄存器
  - 调试寄存器
  - 测试寄存器

通用寄存器

- EAX：累加器
- EBX：基址寄存器
- ECX：计数器
- EDX：数据寄存器
- ESI：源地址指针寄存器
- EDI：目的地址指针寄存器
- EBP：基址指针寄存器
- ESP：堆栈指针寄存器

段寄存器

- CS：代码段(Code Segment)
- DS：数据段(Data Segment)
- ES：附加数据段(Extra Segment)
- SS：堆栈段(Stack Segment)
- FS：附加段
- GS：附加段

指令寄存器 EIP

EIP的低16位就是8086的IP，存储下一条指令的内存地址，在分段地址转换中，表示指令的段内偏移地址。

标志寄存器 EFLAGS：

IF(Interrupt Flag)：中断允许标志位,由CLI，STI两条指令来控制；设置 IF 使CPU可识别外部（可屏蔽）中断请求。复位 IF 则禁止中断。 IF 对不可屏蔽外部中断和故障中断的识别没有任何作用。

5. 主要数据结构

- ucore主要基于C语言设计，采用了面向对象编程方法
- uCore采用了类似C++接口（interface）概念，让内核子系统（比如物理内存分配器、调度器，文件系统等）实现细节不同，但是具有共同的操作方式

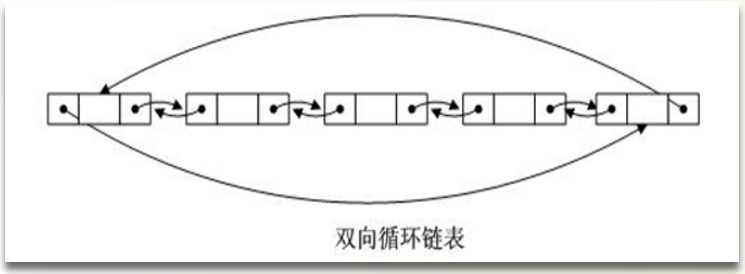
/lab2/kern/mm/pmm.h

```
---------------------------------
struct pmm_manager {
    const char *name;
    void (*init)(void);
    void (*init_memmap)(struct Page *base, size_t n);
    struct Page *(*alloc_pages)(size_t n);
    void (*free_pages)(struct Page *base, size_t n);
    size_t (*nr_free_pages)(void);
    void (*check)(void);
};
```

数据结构

- 双向循环链表

```
typedef struct foo {
    ElemType data;
    struct foo *prev;
    struct foo *next;
} foo_t;
```



- 需要定义特定的链表插入、删除等各种操作，会导致代码冗余。

数据结构

- 双向链表结构定义

```
struct list_entry {
    struct list_entry *prev, *next;
};
```

```
typedef struct {
    list_entry_t free_list;
    unsigned int nr_free;
} free_area_t;
```

```
struct Page {
    atomic_t ref;
     ……
    list_entry_t page_link;
};
```

数据结构

- 链表操作函数

```
list_init(list_entry_t *elm)
list_add_after
list_add_before
list_del(list_entry_t *listelm)
```

- 访问链表节点所在的宿主数据结构

6. QEMU仿真器及使用

- QEMU用于模拟一台x86计算机，ucore在QEMU上运行
- 我们的实验环境的虚拟计算机中，QEMU已经安装成功
- 命令格式：

```
qemu [options] [disk_image]
```

- QEMU运行常用参数：

```
`-hda file'        `-hdb file' `-hdc file' `-hdd file'
    使用 file  作为硬盘0、1、2、3镜像。
`-fda file'  `-fdb file'
    使用 file  作为软盘镜像，可以使用 /dev/fd0 作为 file 来使用主机软盘。
`-cdrom file'
    使用 file  作为光盘镜像，可以使用 /dev/cdrom 作为 file 来使用主机 cd-rom。
`-boot [a|c|d]'
    从软盘(a)、光盘(c)、硬盘启动(d)，默认硬盘启动。
`-snapshot'
    写入临时文件而不写回磁盘镜像，可以使用 C-a s 来强制写回。
`-m megs'
    设置虚拟内存为 msg M字节，默认为 128M 字节。
`-smp n'
    设置为有 n 个 CPU 的 SMP 系统。以 PC 为目标机，最多支持 255 个 CPU。
```

QEMU常用调试命令

q|quit|exit	退出 qemu。
stop	停止 qemu。
c|cont|continue	连续执行。
x /fmt addr xp /fmt addr	显示内存内容，其中 'x' 为虚地址，'xp' 为实地址。 参数 /fmt i 表示反汇编，缺省参数为前一次参数。
p|print'	计算表达式值并显示，例如 $reg 表示寄存器结果。
memsave addr size file pmemsave addr size file	将内存保存到文件，memsave 为虚地址，pmemsave 为实地址。
breakpoint 相关：	设置、查看以及删除 breakpoint，pc执行到 breakpoint，qemu 停止。（暂时没有此功能）
watchpoint 相关：	设置、查看以及删除 watchpoint, 当 watchpoint 地址内容被修改，停止。（暂时没有此功能）
s|step	单步一条指令，能够跳过断点执行。
r|registers	显示全部寄存器内容。
info 相关操作	查询 qemu 支持的关于系统状态信息的操作。

结合gdb和qemu调试ucore

- 需要在使用gcc编译源文件的时候加参数："-g"
- ucore 代码编译
  - 对ucore 源码使用make编译，例如 lab1中：

```
stu@laptop: ~/lab1$  make
```

然后，在lab1目录下的bin目录中，生成一系列的目标文件。

远程调试ucore

- 步骤1： 启动qemu，需要使用参数-S –s这两个参数:

  qemu -S -s -hda ./bin/ucore.img -monitor stdio

- 步骤2：运行 gdb并与qemu进行连接

  (gdb) target remote 127.0.0.1:1234

  (gdb) file ./bin/kernel

- 步骤3：设置断点并执行

- 步骤4： qemu 单步调试

窗口一	窗口二
stu@laptop: ~/lab1$ qemu -S -s -hda ./bin/ucore.img	chy@laptop: ~/lab1$ gdb ./bin/kernel (gdb) target remote:1234 Remote debugging using :1234 0x0000fff0 in ?? () (gdb) break memset Breakpoint 1, memset (s=0xc029b000, c=0x0, n=0x1000) at libs/string.c:271 (gdb) continue Continuing. Breakpoint 1, memset (s=0xc029b000, c=0x0, n=0x1000) at libs/string.c:271 271 memset(void *s, char c, size_t n) { (gdb)

使用gdb配置文件

- 步骤1：以lab1为例，在lab1/tools目录下，执行 make
- 步骤2：创建文件 gdbinit文件：

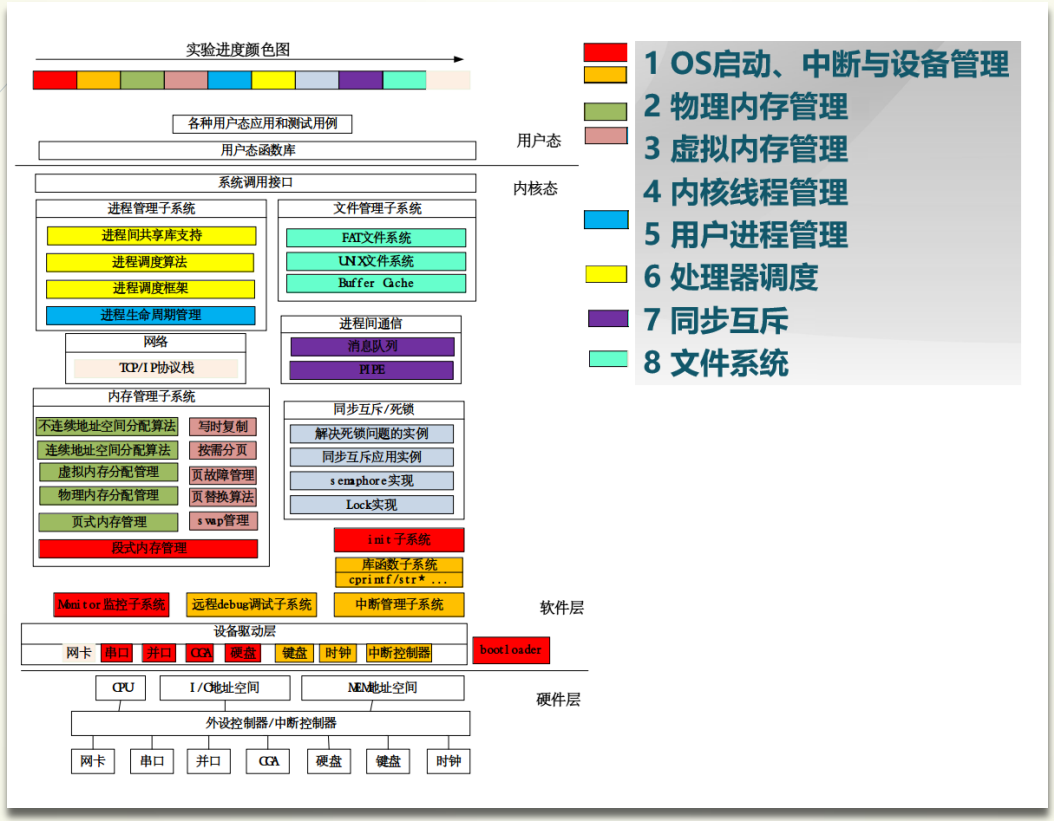
```
target remote 127.0.0.1:1234
file bin/kernel
```

再使用下面的命令启动gdb

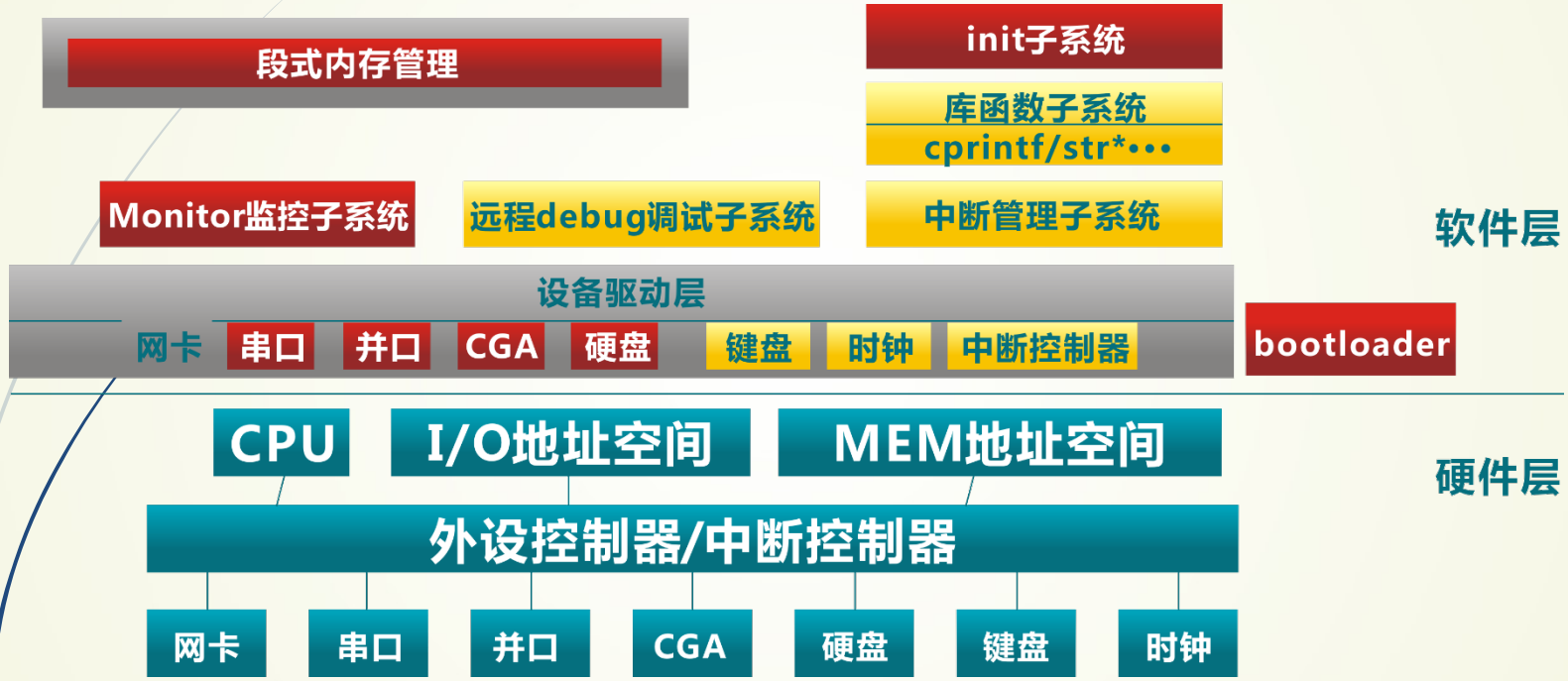
```
$ gdb -x tools/gdbinit
```

- 步骤3：设置断点并执行
- 步骤4： qemu 单步调试

7. 实验课程设计单元介绍



7. 实验课程设计单元介绍



7. 实验课程设计单元介绍

7. 实验单元内容介绍

- **Lab1: Bootloader**

  Bootloader /Interrupt/Device Driver

- 了解bootloader启动操作系统的程过的
- 行运译编bootloader
- 试调学会bootloader法方的

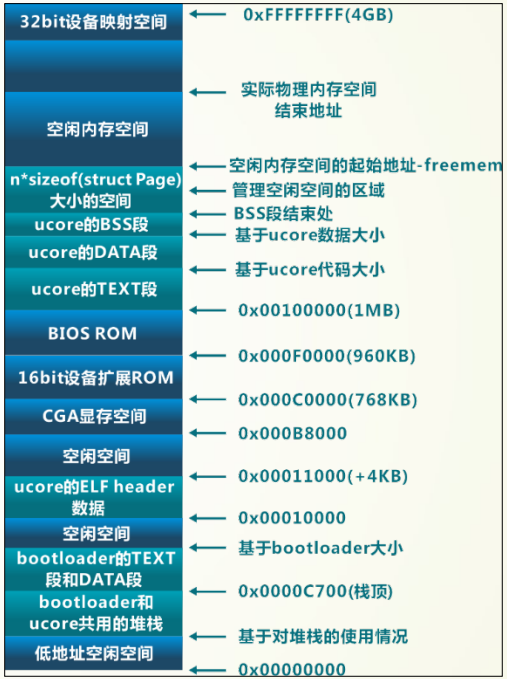
```
proj1 /
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- libs
|   |-- types.h
|   `-- x86.h
|-- Makefile
`-- tools
    |-- function.mk
    `-- sign.c

3 directories, 8 files
```

实验单元内容介绍

- **Lab2:物理内存管理**

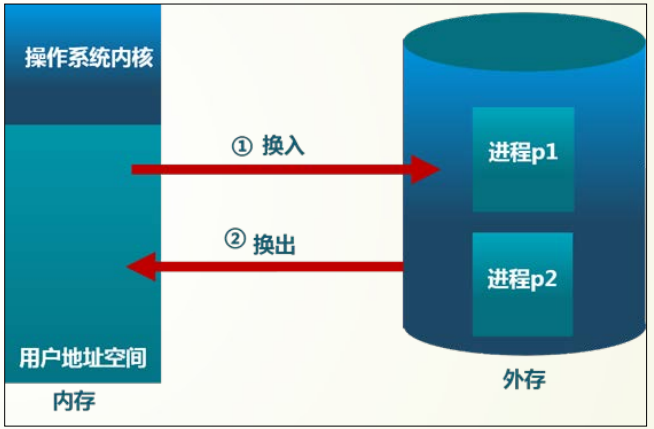
理解x86分段/分页模式，了解操作系统如何管理连续空间的物理内存



实验单元内容介绍

- **Lab3: 虚拟内存管理**

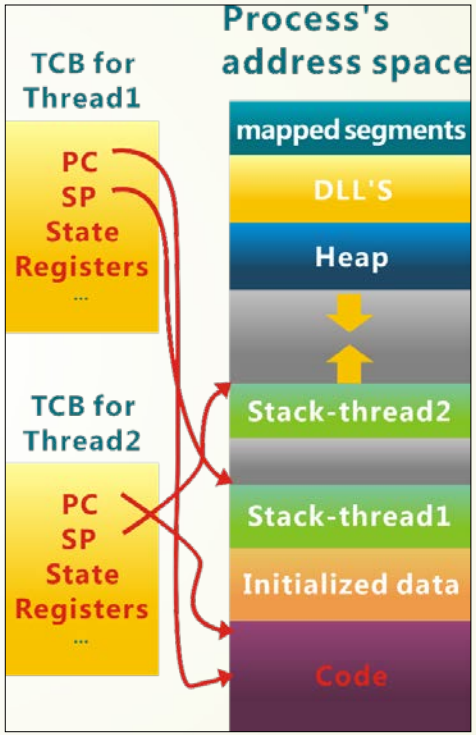
了解页表机制和换出（swap）机制，以及中断-“故障中断”、缺页故障处理等，基于页的内存替换算法



实验单元内容介绍

- **Lab4: 虚拟内存管理**

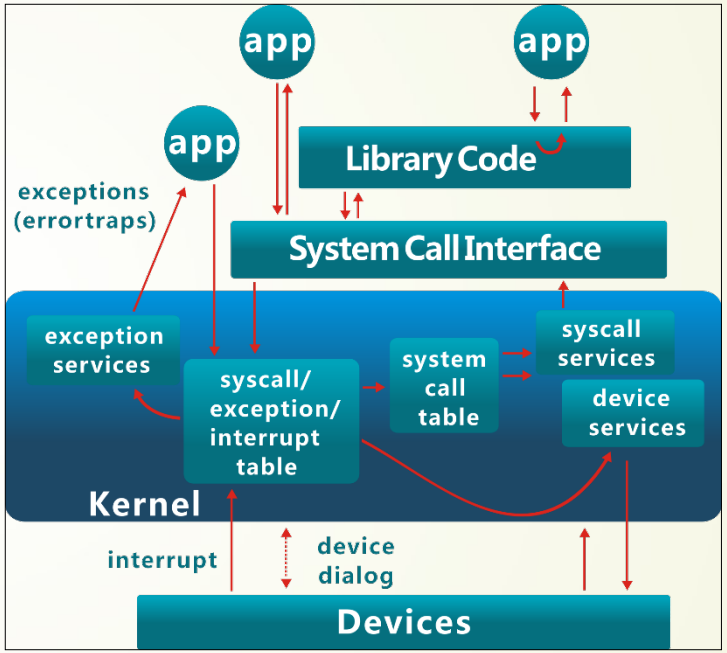
了解如果利用CPU来高效地完成各种工作的设计与实现基础，如何创建相对与用户进程更加简单的内核态线程，如果对内核线程进行动态管理等



实验单元内容介绍

- **Lab5:** 用户进程管理

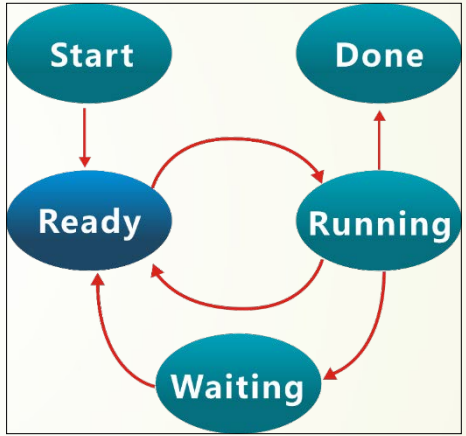
了解用户态进程创建、执行、切换和结束的动态管理过程，实现系统调用的处理过程



实验单元内容介绍

- **Lab6: 进程调度**

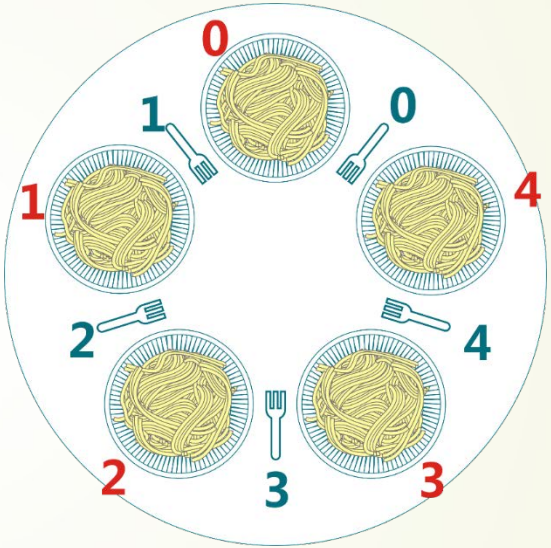
用于理解操作系统的调度过程和调度算法，基于调度器框架实现一个调度器算法



实验单元内容介绍

- **Lab7:** 同步互斥

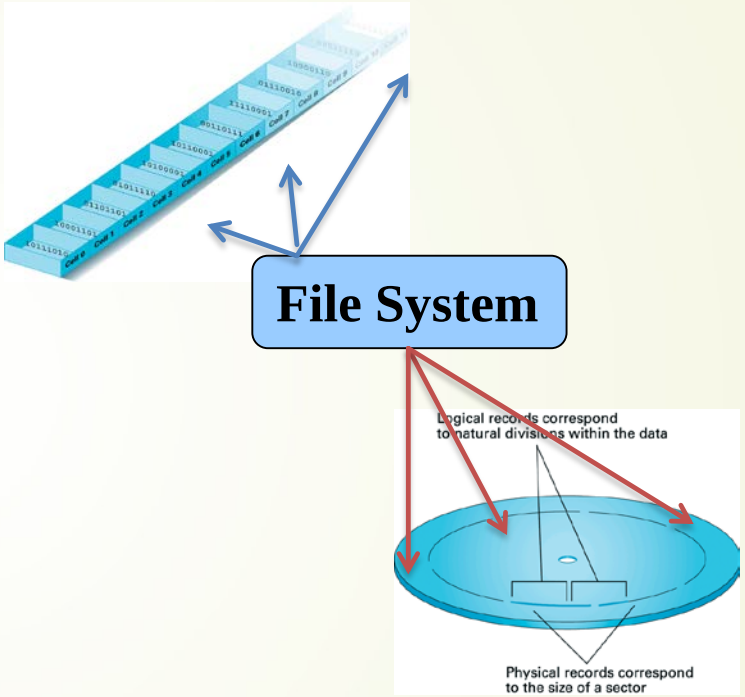
了解进程间如何进行信息交换和共享，用各种同步机制解决同步问题



实验单元内容介绍

- **Lab8: 文件系统**

  了解文件系统的具体实现原理，了解一个基于索引节点组织方式的Simple FS文件系统的设计与实现；了解文件系统抽象层-VFS的设计与实现



本章作业

- 熟悉实验环境设置
- 学会使用gdb跟踪命令
- 学会程序的编辑、编译和调试方法
- 巩固使用Linux的常用命令
- 撰写实验报告report2