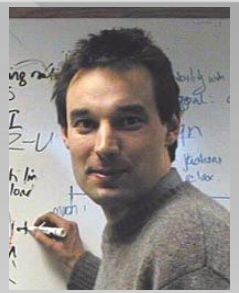
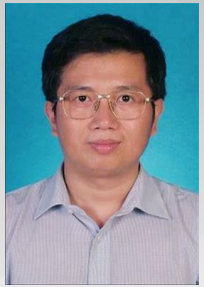
操作系统
Operating System

2018 春季

清华大学计算机系

1. 了解OS
2. 了解硬件
3. 了解编程
4. 了解labs



建立在

计算机科学导论/计算机组成原理/编译原理/汇编语言程序设计

……

课程基础上

了解OS

- OS的功能
  - **干啥？**
- OS的特征
  - **与应用soft的区别？**
- OS的组成
  - **包含那些部分？**
- OS的需求
  - **对硬件的需求？**

了解OS

- OS的功能
  - 干啥？
- OS的特征
  - 与应用s
- OS的组成
  - 包含那些
- OS的需求
  - 对硬件的

Linux Syscall Reference

Show 10 entries

#	Name	Registers			
		eax	ebx	ecx	edx
0	sys_restart_syscall	0x00	-	-	-
1	sys_exit	0x01	int error_code	-	-
2	sys_fork	0x02	struct pt_regs *	-	-
3	sys_read	0x03	unsigned int fd	char __user *buf	size_t count
4	sys_write	0x04	unsigned int fd	const char __user *buf	size_t count
5	sys_open	0x05	const char __user *filename	int flags	int mode
6	sys_close	0x06	unsigned int fd	-	-
7	sys_waitpid	0x07	pid_t pid	int __user *stat_addr	int options
8	sys_creat	0x08	const char __user *pathname	int mode	-
9	sys_link	0x09	const char __user *oldname	const char __user *newname	-

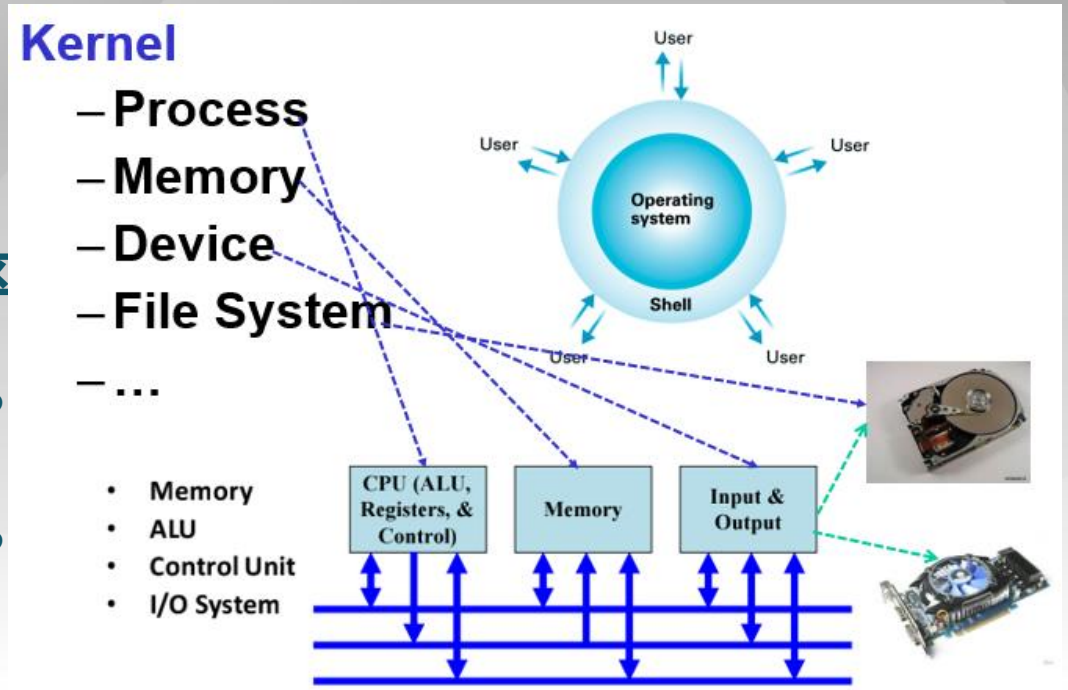
了解OS

- OS的功能
  - **干啥？**
- OS的特征
  - **与应用soft的区别？**

执行特权
Exceptions/Interrupt
Segmentation/Paging
Virtual Memory
Privilege Modes

了解OS

- OS的功能
  - 干啥？
- OS的特征
  - 与应用soft的区
- OS的组成
  - 包含那些部分？
- OS的需求
  - 对硬件的需求？



了解OS

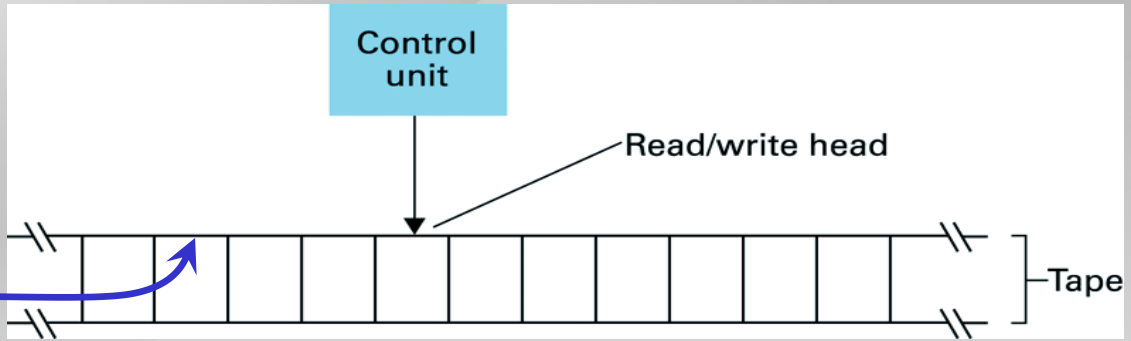
- OS的功能
  - **干啥？**
- OS的特征
  - **与应用soft的区别？**
- OS的组成
  - **包含那些部分？**
- OS的需求
  - **对硬件的需求？**

- 特权指令
  - 只有CPU在特权态时才可执行的指令
  - 如果CPU不在特权态而执行它们，那么会引起异常

Exceptions: 中断/异常/系统服务等管理指令
Virtual Memory: TLB/MMU等管理指令
Privilege Modes: 调整特权级管理指令
Segmentation/Paging: 分段分页管理指令

1. 了解OS
2. 了解硬件
3. 了解编程
4. 了解labs

了解硬件 Turing Machines



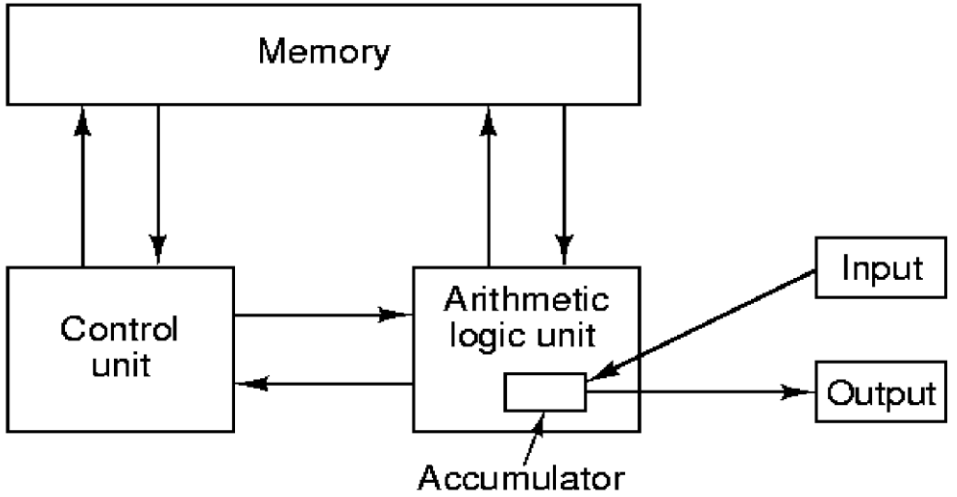
symbols from machine′s finite alphabet represented in machine′s cells

finite number of conditions called states – START, HALT, ADD, etc.

each step executed by control unit – *observe*s symbol, *write*s a symbol, and maybe *move* the read/write left or right, *change*s states

了解硬件- VonNeuman Arch

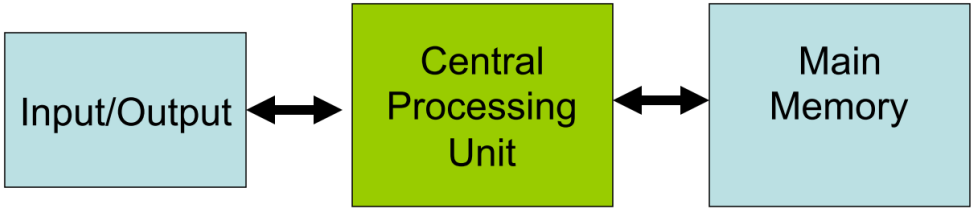
Von Neumann计算机



存储程序、二进制、体系结构

了解硬件

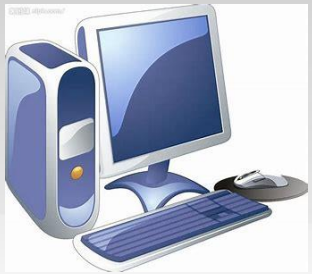
Abstract model

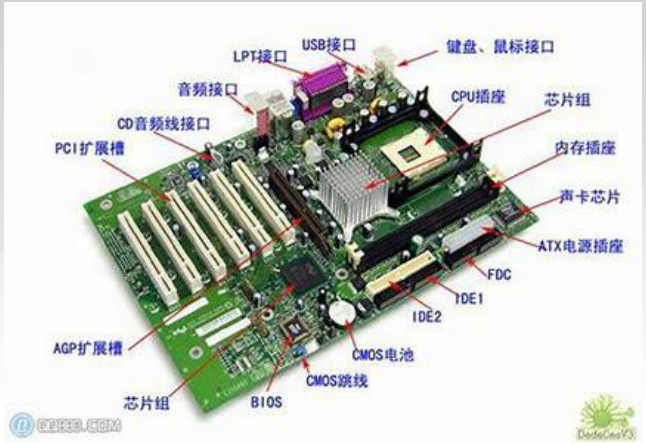


- I/O: communicating data to and from devices
- CPU: digital logic for performing computation
- Memory: *N* words of *B* bits

了解硬件

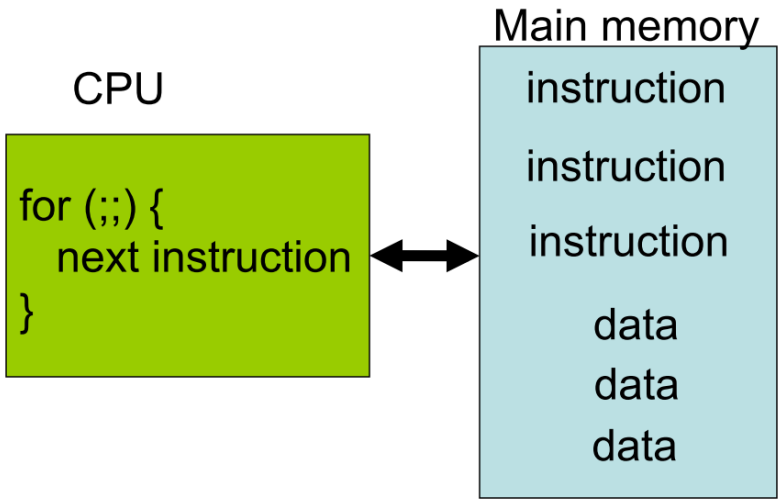
- - A full PC has:
  - an x86 CPU with registers, execution unit, and memory management
  - CPU chip pins include address and data signals
  - memory
  - disk
  - keyboard
  - display
  - other resources: BIOS ROM, clock, ...





了解硬件

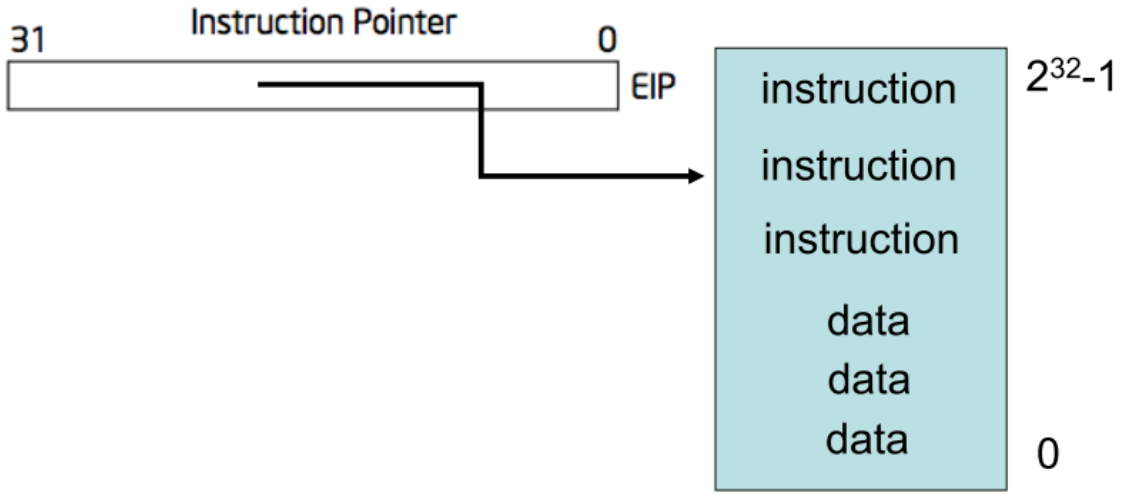
The stored program computer



- Memory holds *instructions* and *data*
- CPU *interpreter* of instructions

了解硬件

x86 implementation



- EIP is incremented after each instruction
- Instructions are different length
- EIP modified by CALL, RET, JMP, and conditional JMP

了解硬件

Registers for work space

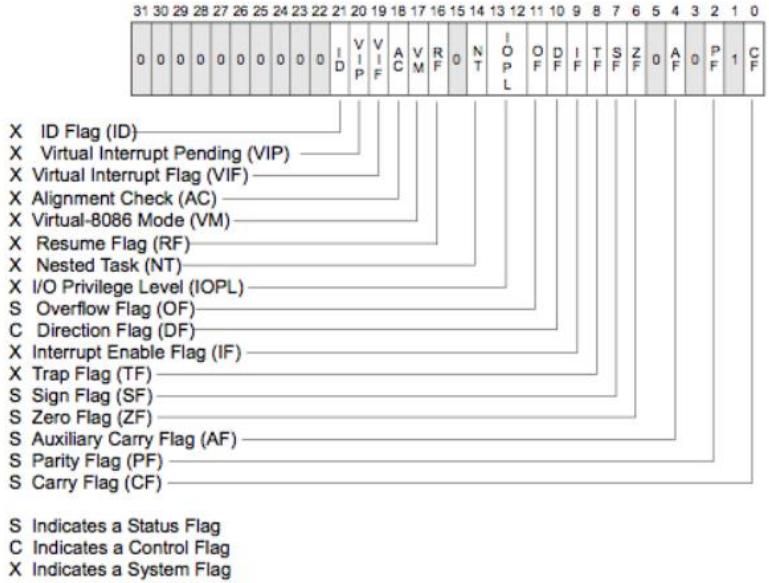
General-Purpose Registers

31 16	15 8	7 0	16-bit	32-bit
	AH	AL	AX	EAX
	BH	BL	BX	EBX
	CH	CL	CX	ECX
	DH	DL	DX	EDX
	BP			EBP
	SI			ESI
	DI			EDI
	SP			ESP

- 8, 16, and 32 bit versions
- By convention some registers for special purposes
- Example: ADD EAX, 10
- Other instructions: SUB, AND, etc.

了解硬件

EFLAGS register



- Test instructions: TEST EAX, 0
- Conditional JMP instructions: JNZ address

了解硬件

Memory: more work space

```
movl %eax, %edx      edx = eax;                    register mode
movl $0x123, %edx    edx = 0x123;                  immediate
movl 0x123, %edx     edx = *(int32_t*)0x123;       direct
movl (%ebx), %edx    edx = *(int32_t*)ebx;         indirect
movl 4(%ebx), %edx   edx = *(int32_t*)(ebx+4);     displaced
```

- Memory instructions: MOV, PUSH, POP, etc
- Most instructions can take a memory address

了解硬件

Stack memory + operations

Example instruction	What it does
pushl %eax	subl $4, %esp movl %eax, (%esp)
popl %eax	movl (%esp), %eax addl $4, %esp
call 0x12345	pushl %eip (*) movl $0x12345, %eip (*)
ret	popl %eip (*)

- Stack grows down
- Use to implement procedure calls

了解硬件

More memory

- 8086 16 registers and 20-bit bus addresses
- The extra 4 bits come *segment registers*
  - CS: code segment, for EIP
  - SS: stack segment, for SP and BP
  - DS: data segment for load/store via other registers
  - ES: another data segment, destination for string ops
  - For example: CS=4096 to start executing at 65536
- Makes life more complicated
  - Cannot use 16 bit address of stack variable as pointer
  - Pointer arithmetic and array indexing across segment boundaries
  - For a far pointer programmer must include segment reg

了解硬件

And more memory

- 80386: 32 bit data and bus addresses
- Now: the transition to 64 bit addresses
- Backwards compatibility:
  - Boots in 16-bit mode, and boot.S switches to protected mode with 32-bit addresses
  - Prefix 0x66 gets you 32-bit:
    - MOVW = 0x66 MOVW
  - .code32 in boot.S tells assembler to insert 0x66
- 80386 also added virtual memory addresses
  - Segment registers are indices into a table
  - Page table hardware

了解硬件

I/O space and instructions

```
#define DATA_PORT    0x378
#define STATUS_PORT  0x379
#define   BUSY 0x80
#define CONTROL_PORT 0x37A
#define   STROBE 0x01
void
lpt_putc(int c)
{
  /* wait for printer to consume previous byte */
  while((inb(STATUS_PORT) & BUSY) == 0)
    ;

  /* put the byte on the parallel lines */
  outb(DATA_PORT, c);

  /* tell the printer to look at the data */
  outb(CONTROL_PORT, STROBE);
  outb(CONTROL_PORT, 0);
}
```

- 8086: Only 1024 I/O addresses

了解硬件

Memory-mapped I/O

- Use normal addresses
  - No need for special instructions
  - No 1024 limit
  - System controller routes to device
- Works like “magic” memory
  - Addressed and accessed like memory
  - But does not behave like memory
  - Reads and writes have “side effects”
  - Read result can change due to external events

了解硬件

指令功能分类

- 数据运算指令
  - 算术运算、逻辑运算
- 数据传输指令
  - 寄存器之间、主存/寄存器之间
- 输入/输出指令
  - 与输入/输出端口的数据传输
- 控制指令
  - 转移指令、子程序调用/返回
- 其它指令
  - 停机、开/关中断、空操作、特权、置条件码

了解硬件

- 特权指令
  - 是指保护方式下只有当前特权级CPL=0时，才可执行的指令
  - 如果CPL不等于0而执行它们，那么会引起通用保护异常。
  - 从上面介绍的操作系统类指令可归纳出如下表所示的80386特权指令

Exceptions: LIDT, LTR, IRET, STI, CLI

Virtual Memory: MOV CRn, INVLPG, INVPCID

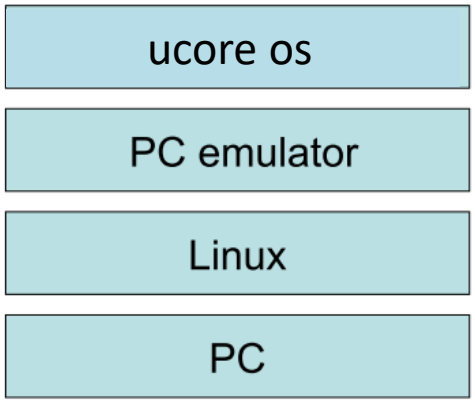
Privilege Modes: SYSRET, SYSEXIT, IRET

Segmentation/Paging: LGDT, LLDT CRx: CR0,CR3....

了解硬件

Development using PC emulator

- QEMU PC emulator
  - does what a real PC does
  - Only implemented in software!
- Runs like a normal program on “host” operating system



了解硬件

Emulation of memory

```
int32_t regs[8];
#define REG_EAX 1;
#define REG_EBX 2;
#define REG_ECX 3;
...
int32_t eip;
int16_t segregs[4];
...

char mem[256*1024*1024];
```

了解硬件

Emulation of CPU

```
for (;;) {
        read_instruction();
        switch (decode_instruction_opcode()) {
        case OPCODE_ADD:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] + regs[src];
                break;
        case OPCODE_SUB:
                int src = decode_src_reg();
                int dst = decode_dst_reg();
                regs[dst] = regs[dst] - regs[src];
                break;
        ...
        }
        eip += instruction_length;
}
```

了解硬件

Emulation x86 memory

```
uint8_t read_byte(uint32_t phys_addr) {
        if (phys_addr < LOW_MEMORY)
                return low_mem[phys_addr];
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                return rom_bios[phys_addr - 960*KB];
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                return ext_mem[phys_addr-1*MB];
        else ...
}

void write_byte(uint32_t phys_addr, uint8_t val) {
        if (phys_addr < LOW_MEMORY)
                low_mem[phys_addr] = val;
        else if (phys_addr >= 960*KB && phys_addr < 1*MB)
                ; /* ignore attempted write to ROM! */
        else if (phys_addr >= 1*MB && phys_addr < 1*MB+EXT_MEMORY) {
                ext_mem[phys_addr-1*MB] = val;
        else ...
}
```

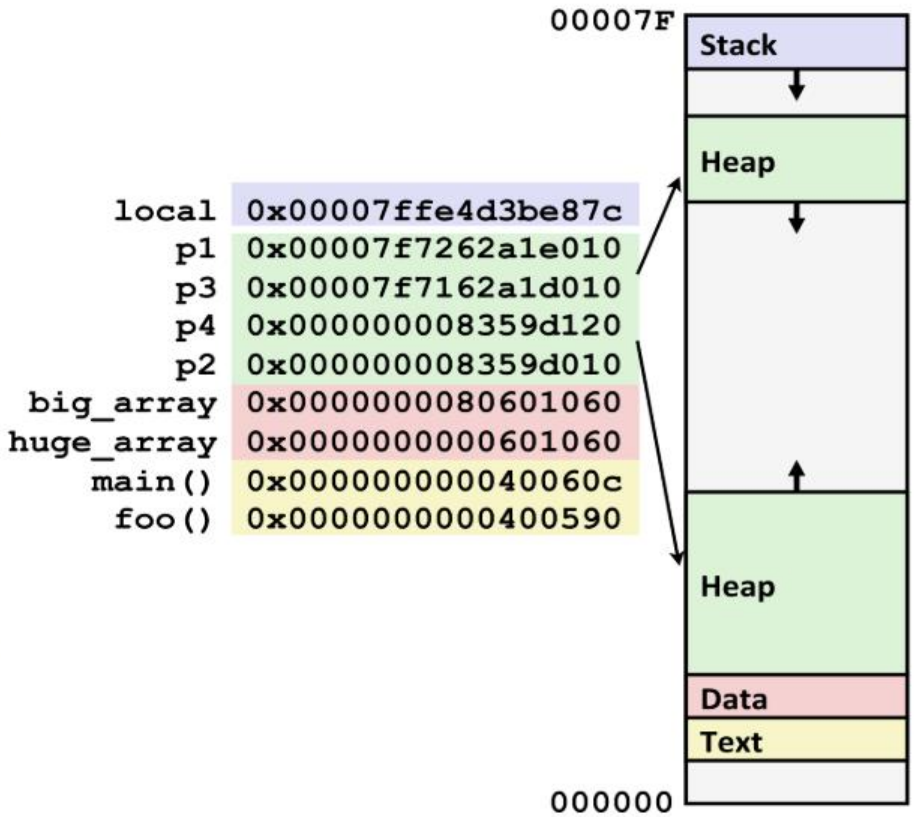
了解硬件

Emulating devices

- Hard disk: using a file of the host
- VGA display: draw in a host window
- Keyboard: hosts’s keyboard API
- Clock chip: host’s clock
- Etc.

了解硬件

```
char big_array[1L<<24];
char huge_array[1L<<31];

int foo() { return 0; }

int main() {
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28);
    p2 = malloc(1L << 8);
    p3 = malloc(1L << 32);
    p4 = malloc(1L << 8);
 /* Some print statements ... */
}
```



1. 了解OS
2. 了解硬件
3. 了解编程
4. 了解labs

了解ucore编程

- 代码编译生成　　（From 程序设计/编译原理课）
- 汇编码与机器码　　（From 计算机组成原理/汇编语言程序设计课）
- 函数调用　　（From 编译原理/汇编语言程序设计课）
- 指令集/特权指令　　（From计算机组成原理课）
- 程序运行　　（From ALL）

了解ucore编程方法和通用数据结构

- ucore主要基于C语言设计，采用了一定的面向对象编程方法。

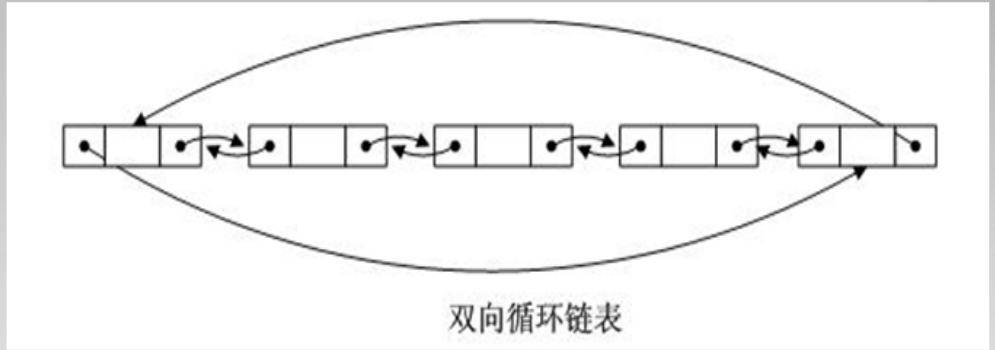
/lab2/kern/mm/pmm.h

```
------------------------------
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);
};
```

了解ucore编程方法和通用数据结构

- **双向循环链表**

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

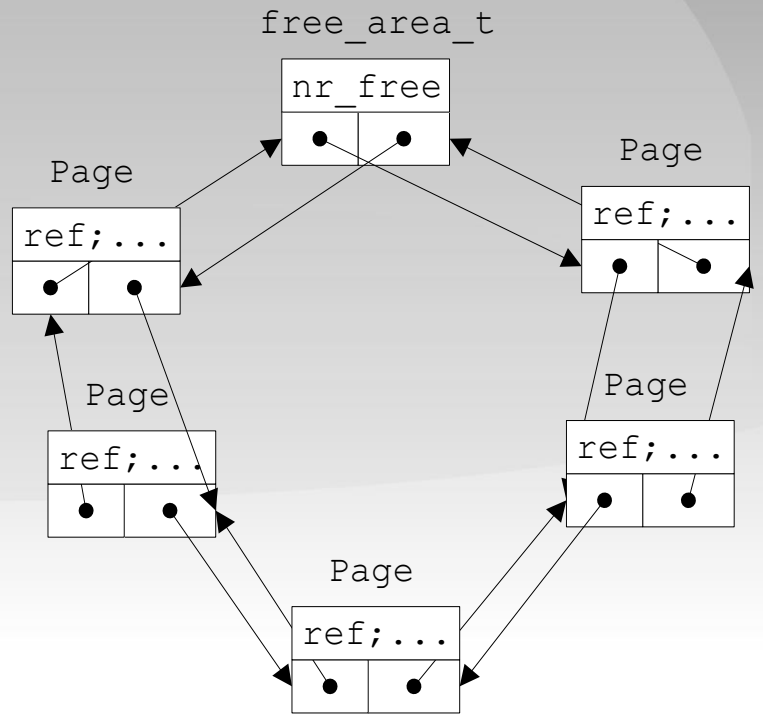


需要为每种特定数据结构类型定义针对这个数据结构的特定链表插入、删除等各种操作，会导致代码冗余。

了解ucore编程方法和通用数据结构

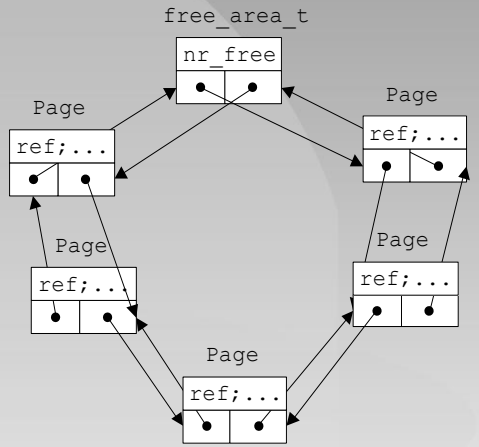
- uCore的双向链表结构定义

```
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;
};
```



了解ucore编程方法和通用数据结构

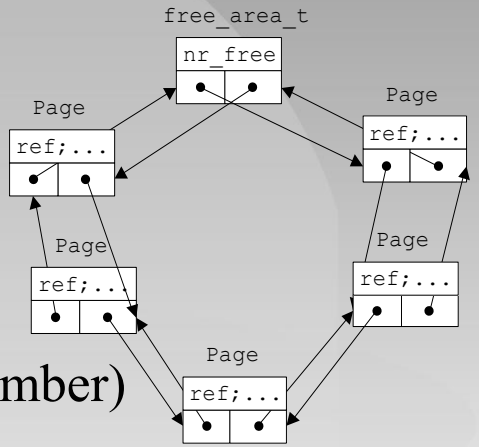
- **链表操作函数**
  - list_init(list_entry_t *elm)
  - list_add_after和list_add_before
  - list_del(list_entry_t *listelm)
- **访问链表节点所在的宿主数据结构**




```
free_area_t free_area;
list_entry_t * le = &free_area.free_list;
while((le=list_next(le)) != &free_area.free_list) {
      struct Page *p = le2page(le, page_link);        ……
}
```

了解ucore编程方法和通用数据结构

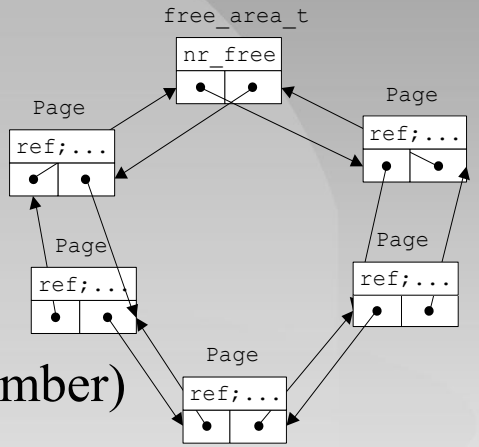
- **链表操作函数**
  - list_init(list_entry_t *elm)
  - list_add_after和list_add_before
  - list_del(list_entry_t *listelm)
- **访问链表节点所在的宿主数据结构**



#define le2page(le, member) to_struct((le), struct Page, member)

了解ucore编程方法和通用数据结构

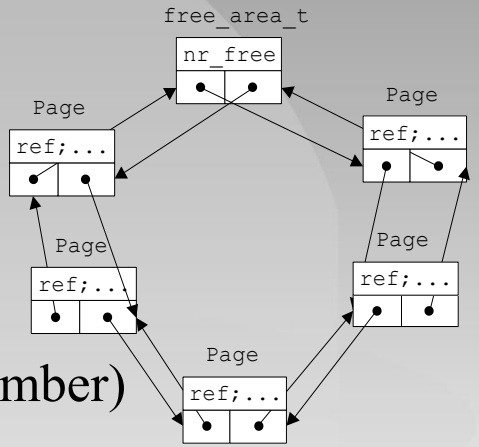
- **链表操作函数**
  - list_init(list_entry_t *elm)
  - list_add_after和list_add_before
  - list_del(list_entry_t *listelm)
- **访问链表节点所在的宿主数据结构**




```
#define le2page(le, member)      to_struct((le), struct Page, member)

#define to_struct(ptr, type, member)                                \
    ((type *)((char *)(ptr) - offsetof(type, member)))
```

了解ucore编程方法和通用数据结构

- **链表操作函数**
  - list_init(list_entry_t *elm)
  - list_add_after和list_add_before
  - list_del(list_entry_t *listelm)
- **访问链表节点所在的宿主数据结构**




```
#define le2page(le, member)      to_struct((le), struct Page, member)

#define to_struct(ptr, type, member)                        \
    ((type *)((char *)(ptr) - offsetof(type, member)))

#define offsetof(type, member)                              \
    ((size_t)(&((type *)0)->member))
```

1. 了解OS
2. 了解硬件
3. 了解编程
4. 了解labs

实验课程设计

- 设计思路
  - 采用小巧全面的操作系统ucore并进行改进，需要覆盖操作系统的关键点，为此增加：
    - 外设：I/O管理/中断管理
    - 内存：虚存管理/页表/缺页处理/页替换算法
    - CPU：进程管理/调度器算法
    - 并发：信号量实现和同步互斥应用
    - 存储：基于链表/FAT的文件系统
  - 完整代码量控制在10000行以内
  - 提供实验讲义和源码分析文档

实验课程设计


各种用户态应用和测试用例
用户态函数库
用户态
系统调用接口
内核态
进程管理子系统
进程间共享库支持
进程调度算法
进程调度框架
进程生命周期管理
文件管理子系统
FAT文件系统
UNIX文件系统
Buffer Cache
网络
TCP/IP协议栈
进程间通信
消息队列
PIPE
内存管理子系统
不连续地址空间分配算法
连续地址空间分配算法
虚拟内存分配管理
物理内存分配管理
页式内存管理
写时复制
按需分页
页故障管理
页替换算法
swap管理
同步互斥/死锁
解决死锁问题的实例
同步互斥应用实例
semaphore实现
Lock实现

实验课程设计

- 实验内容
  - 1 OS启动、中断与设备管理： 0200~1800行
  - 2 物理内存管理： 1800~2500行
  - 3 虚拟内存管理： 2500~3200行
  - 4 内核线程管理： 3200~3600行
  - 5 用户进程管理： 3600~4300行
  - 6 处理器调度： 4300~5100行
  - 7 同步互斥： 5100~6400行
  - 8 文件系统： 6400~9999行

实验课程设计

Lab1: Bootloader/Interrupt/Device Driver

- 启动操作系统的bootloader，了解操作系统启动前的状态和要做的准备工作，了解运行操作系统的硬件支持，操作系统如何加载到内存中，理解两类中断--“外设中断”，“陷阱中断”等；
  - 理管储存的制机段分于基
  - 念概本基的理管备设
  - PC动启bootloader程过的
  - bootloader成组件文的
  - 行运译编bootloader程过的
  - 试调bootloader法方的
  - 程过理处和构结的栈解了级编汇在
  - 制机理处断中
  - 口串过通/口并/CGA法方的符字出输

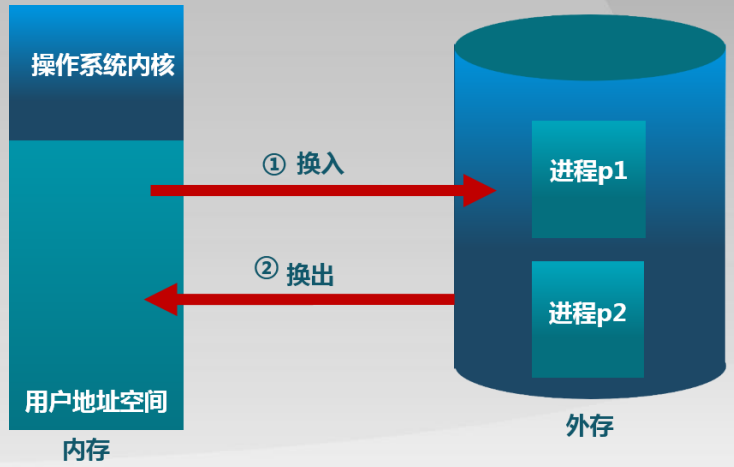
```
proj1 /
|-- boot
|   |-- asm.h
|   |-- bootasm.S
|   `-- bootmain.c
|-- libs
|   |-- types.h
|   `-- x86.h
|-- Makefile
`-- tools
    |-- function.mk
    `-- sign.c

3 directories, 8 files
```

实验课程设计

- 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
    - 理解内存地址的转换和保护
    - 实现页表的建立和使用方法
    - 实现物理内存的管理方法
    - 了解常用的减少碎片的方法

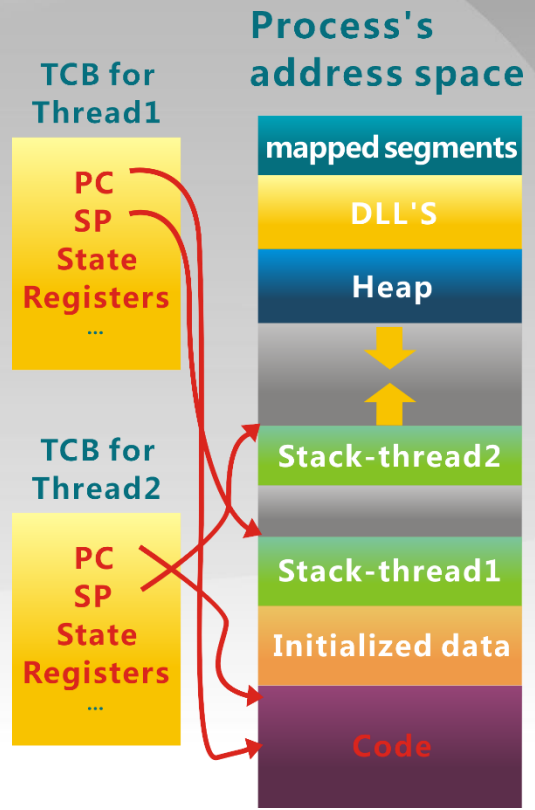
实验课程设计

- 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；
    - 理解换页的软硬件协同机制
    - 实现虚拟内存的Page Fault异常处理
    - 实现页替换算法



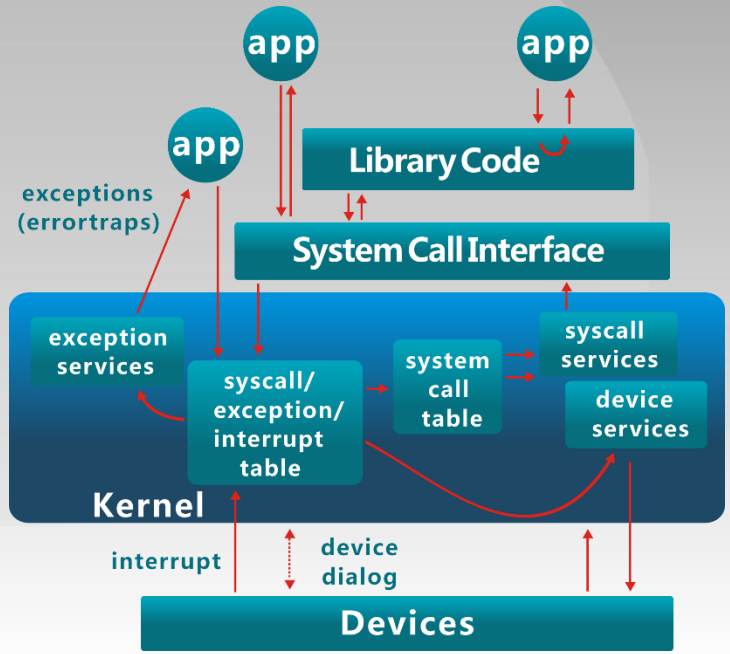
实验课程设计

- Lab4: 内核线程管理
  - 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；
    - 建立内核线程的关键息信
    - 实现内核线程的管理方法



实验课程设计

- 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，了解在用户态通过系统调用得到内核态的内核服务的过程
    - 建立用户进程的关键信息
    - 实现用户进程管理
    - 分析进程和内存管理的关系
    - 实现系统调用的处理过程

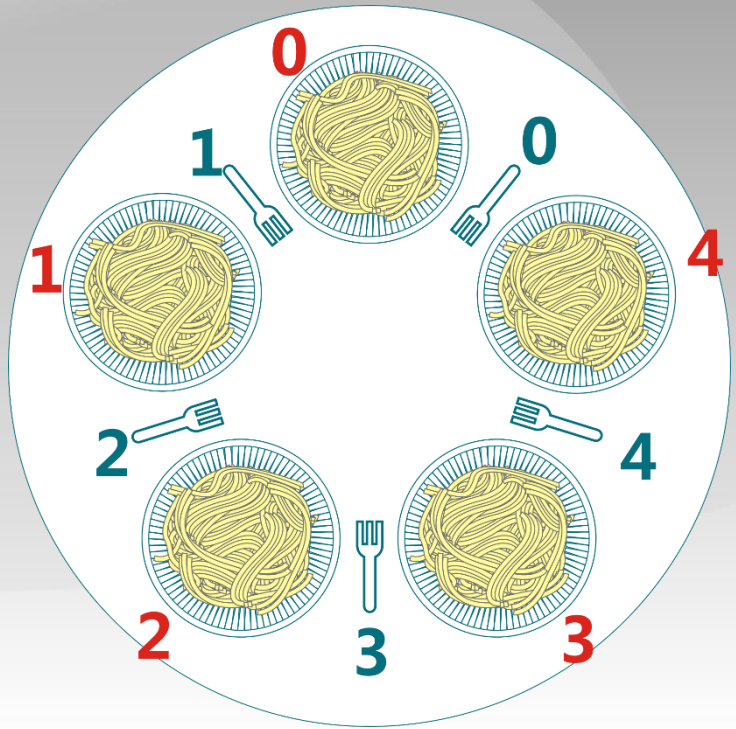


实验课程设计

- Lab6：进程调度
  - 用于理解操作系统的调度过程和调度算法
    - 熟悉 ucore 的系统调度器框架，以及内置的 Round-Robin 调度算法。
    - 基于调度器框架实现一个调度器算法

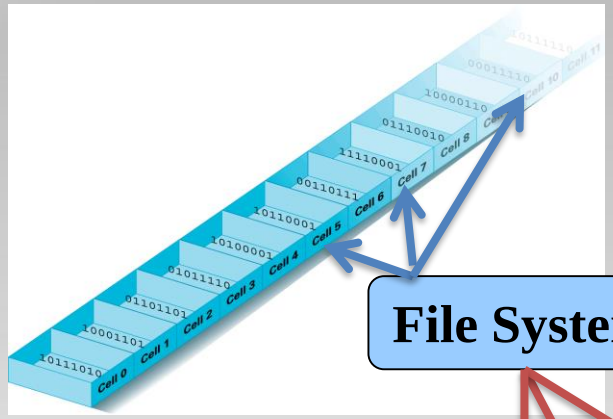
实验课程设计

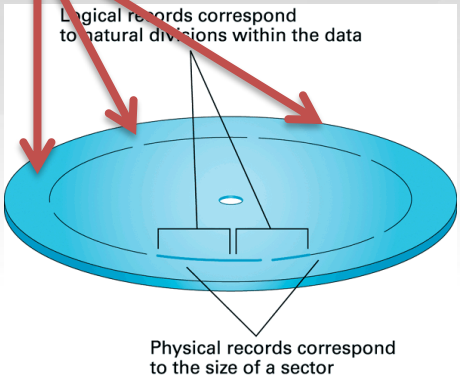
- Lab7：同步互斥
  - 了解进程间如何进行信息交换和共享，并了解同步互斥的具体实现以及对系统性能的影响，研究死锁产生的原因，以及如何避免死锁；
    - 熟悉 ucore 的同步互斥机制
    - 理解基本的spinlock、semphpore、condition variable的实现
    - 用各种同步机制解决同步问题



实验课程设计

- Lab8：文件系统
  - 了解文件系统的具体实现，与进程管理等的关系，了解缓存对操作系统IO访问的性能改进，了解虚拟文件系统（VFS）、buffer cache和disk driver之间的关系。
    - 掌握基本的文件系统系统调用的实现方法；
    - 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；





实验课程设计

- **扩展实验**

U0: ucore porting on x86-64
Status: 100%, ucorer: wnz

U1: local page replacement framework with different algorithms of local page replacement
status: 100%, ucorer: yxh

U2: ucore支持ARM CPU(with mmu)
Status: 100 %, ucorer: wjf, ykl, xb
…
U9: ucore文件系统框架：支持在VFS下同时支持FAT32等文件系统，实现更加简化的VFS、FAT和SFS，并能够实现高性能的基于DMA方式的磁盘访问；
status: 100%, ucorer: qz,rsw
…
U12:ucore支持GO programming
Status: 100 %, ucorer: cr,fjy

效果

- 好的方面
  - 理论和实验能够较好地结合起来，不再感到OS课是一个只要死记硬背的课程了
  - 理解了一个OS的全局设计实现，而不是一个一个分离的知识点
  - 掌握了许多OS原理上没有涉及或涉及不够的东西，比如中断/系统调用的实现，X86的段页机制，进程上下文如何切换的，内核态和用户态的具体区别是什么
  - 这是大学期碰到的最复杂的软件，学习了分析和设计大型系统软件的方法