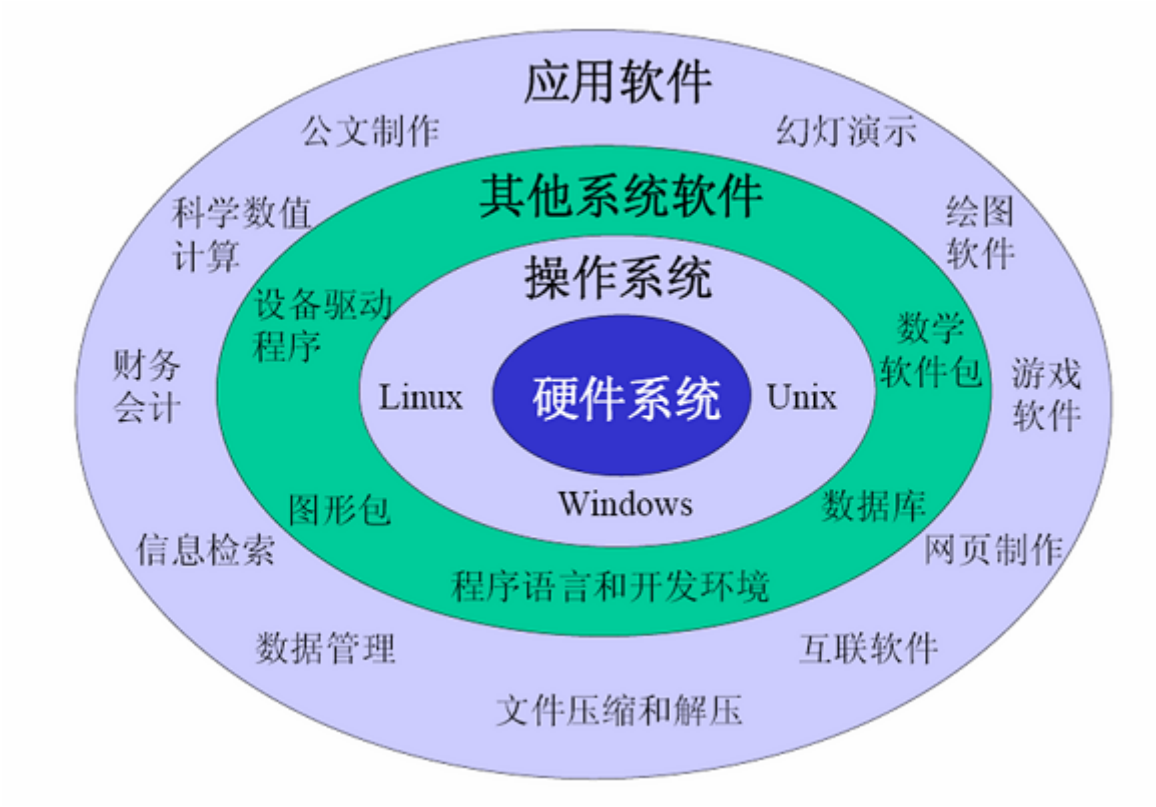



#深入理解计算机系统



# 1.实时系统概念

**前后台系统**

后台是各种面向硬件的程序，如中断，定时器，gpio等。
前台是：

```
main()
{
    while(1)
    {
        ;
    }
}
```

循环中不断调用各种函数实现功能。

**代码临界段**

指处理时不可分割的代码，一旦这部分代码运行就不可以打断、 为了确保代码能正常运行，进入临界段代码钱需要关中断，执行完后再开中断。

**任务**

一个任务也就是一个线程，是一个简单的程序。 任务间通信最简单的办法是使用数据共享结构。 任务间通信途径：1）全局变量；2）发消息给另一个任务。 任务切换（context switch）

**基于优先级的内核**

不可剥夺型内核，允许使用不可重入函数。 可剥夺型内核，，最高优先级的任务一就绪，总能得到CPU的使用权。，不能直接使用不可重入函数。

**互斥条件**

处理共享数据时保证互斥，最简单的办法是关中断和开中断。

**信号量**

一种约定机制。 就好像一把钥匙。任务要运行下去需要获得信号量，且信号量没有被占用。

**死锁**

两个任务相互等待对方释放资源。

**同步**

一个中断或者任务触发另一个任务。

**邮箱**

一种内存共享方式。

**时钟节拍**

特定的周期性中断，如同系统的心脏。

# 2.编译连接

**目标文件的格式**

可重定位文件：

```
包含代码和数据
可被用来链接成执行文件或者共享目标文件
linux (.o)  windows (.obj)
```

可执行文件：

```
包含可以执行的程序
系统可以直接执行的文件
linux (ELF文件,无后缀) windows (.exe)
```

共享目标文件：

```
包含代码和数据
跟可重定位文件和共享目标文件链接，产生新的目标文件
动态连接器将共享目标文件与可执行文件结合，作为进程映像的一部分来运行
linux (.so)  windows (.DLL)
```

核心转储文件

```
Linux (core dump)
```

目标文件的具体内容 file header

```
目标文件头
```

code section

```
程序指令(.code /.text)
存放程序代码程序
```

data section

```
程序数据(.data /.bss)
.data段 初始化的全局和局部静态变量
.bss段  未初始化的全局和局部静态变量
.bss (block started by symbol) 符号预留块,没有内容不占据空间
```

othe section

```
还有可能包含的其他段，例 bank data .ect
```

程序指令和数据分开存放的优点？3点。

# 3.AT&T汇编指令学习(GCC)

1. 寄存器命名原则: 相比inter语法,AT&T语法格式要求所有的寄存器都必须加上取值符"%".

2. 操作码命令格式:

    1. 源/目的操作数顺序:
    Intel语法格式中命令表示格式为:"opcode dest, src"; "操作码 目标, 源"
    AT&T语法格式表示为:"opcode src, dest"; "操作码 源, 目标"

    2. 操作数长度标识:
    在AT&T语法中,通过在指令后添加后缀来指明该指令运算对象的尺寸.
    后缀 'b' 指明运算对象是一个字节(byte)
    后缀 'w' 指明运算对象是一个字(word)
    后缀 'l' 指明运算对象是一个双字(long)
    Intel语法中指令'mov'在AT&T语法必须根据运算对象的实际情况写成:'movb','movw'或'movl'。
    注:若在AT&T中省略这些后缀,GAS将通过使用的寄存器大小来猜测指令的操作数长度.

    3. 另外,
    'FAR'不是GAS的关键字,因此对far的call或jmp指令须加前缀 'l', 'far call'要写成 'lcall' , 'far jmp' 要写成 'ljmp' , 'ret far' 写成 'lret'。

3. 常数/立即数的格式:
在AT&T语法中对立即数,须在其前加前缀 $ 来指明,而Inter语法则不需要。
另外, 在常数前也必须加一个前缀字符 * ,而Inter语法则也是不需要的。

4. 内存寻址方式:
在Intel语法中,使用下面格式来表示存储器寻址方式:
`SECTION:[BASE + INDEX*SCALE + DISP];`段:[基地址+变址*比例因子+偏移量]
BASE是基地址索引寄存器(可以是任一通用寄存器),
INDEX是变址寄存器(除ESP外的任一通用寄存器),
SCALE是变址寄存器的比例常数,
DISP是基址/变址寄存器的位移量。
AT&T语法则使用不同的格式来表示寻址方式:
`SECTION:DISP(BASE, INDEX, SCALE);`段:偏移量(基地址,变址,比例因子)

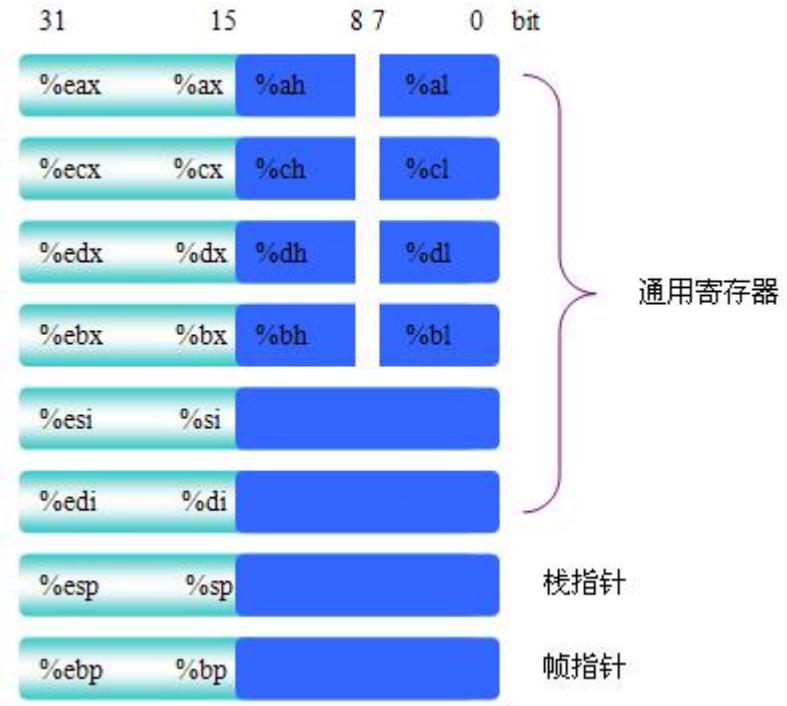
5. 标号 & 标识符:
所有的标号必须以一个字母,点或下划线开始,标号后加一个冒号表示标号的结束。
局部标号使用数字0-9后跟一个冒号,使用局部标号时要在数字后跟一个字符'b'(向后引用)或字符'f'(向前引用)。因为只能使用数字0-9作为局部标号名,所以最多只能定义10个局部标号.一个标识符能给它赋于一个值。(如:'TRUE=1', 或者使用 .set 或 .equ 指令)。

6. 基本的行内汇编格式:
`asm("statements");`
例如： asm("nop"); asm("movl %eax,%ebx");
asm 和 __asm__是完全一样的.
如果有多行汇编，则每一行都要加上 "\n\t"

7. 扩展的行内汇编格式:
`asm ( "statements" : output_regs : input_regs : clobbered_regs);`

冒号后的语句指明输入，输出和被改变的寄存器.

8. IA32整数寄存器：




9. 常用指令：

    1. 数据传送指令：move,push,pop;
    2. 加载有效地址指令：leal;
    3. 一元操作指令：inc(加1),dec(减1)，neg(取负)，not(取补);
    4. 二元操作指令：add,sub,imul,idivl(有符号除法),xor,or,and;
    5. 移位指令：sal(左移)，shl,sar(算数右移)，shr(逻辑右移)；
    6. 跳转指令：jmp,je,jne,js,jns,jg,jl,ja,jb,jbe...

10. 条件码寄存器(单个bit)：
cf(进位标志),zf(零标志),sf(符号标志),of(溢出标志)... 访问条件码指令：cmp,test,set...

```
t = a + b;
cf: (unsigned) t < (unsigned) a;//无符号溢出
zf: t == 0;//零
sf: t < 0;//负数
of: (a < 0 == b < 0) && (t < 0 != a < 0)//有符号溢出
```

# 4.内存对齐

1. 为何要内存对齐

    1. **平台原因(移植原因)**：不是所有的硬件平台都能访问任意地址上的任意数据的；某些硬件平台只能在某些地址处取某些特定类型的数据，否则抛出硬件异常。
    2. **性能原因**：经过内存对齐后，CPU的内存访问速度大大提升。

2. 内存对齐的规则

> 许多实际的计算机系统对基本类型数据在内存中存放的位置有限制，它们会要求这些数据的首地址的值是某个数k(通常它为4或8)的倍数，这就是所谓的内存对齐，而这个k则被称为该数据类型的对齐模数(alignment modulus)。当一种类型S的对齐模数与另一种类型T的对齐模数的比值是大于1的整数，我们就称类型S的对齐要求比T强(严格)，而称T比S弱(宽松)。这种强制的要求一来简化了处理器与内存之间传输系统的设计，二来可以提升读取数据的速度。
>
> > 比如这么一种处理器，它每次读写内存的时候都从某个8倍数的地址开始，一次读出或写入8个字节的数据，假如软件能保证double类型的数据都从8倍数地址开始，那么读或写一个double类型数据就只需要一次内存操作。否则，我们就可能需要两次内存操作才能完成这个动作，因为数据或许恰好横跨在两个符合对齐要求的8字节内存块上。某些处理器在数据不满足对齐要求的情况下可能会出错.
> >
> > > 但是Intel的IA32架构的处理器则不管数据是否对齐都能正确工作。不过Intel奉劝大家，如果想提升性能，那么所有的程序数据都应该尽可能地对齐。

1. Win32平台下的微软C编译器(cl.exe for 80x86)在默认情况下采用如下的对齐规则:
   **任何基本数据类型T的对齐模数就是T的大小，即sizeof(T)。比如对于double类型8字节)，就要求该类型数据的地址总是8的倍数，而char类型数据(1字节)则可以从任何一个地址开始。**
2. Linux下的GCC对齐规则:
   **char类型数据(1字节)起始位置任意,任何2字节大小的数据类型(比如short)的对齐模数是2，而其它所有超过2字节的数据类型(比如long,double)都以4为对齐模数。也就是说2字节数据类型（如short）的地址必须是2的倍数，而较大的数据类型（如int,double等）的地址必须是4的倍数，这意味着short类型的队形的地址最低位必须等于0，任何int类型的对象或指针的最低两位必须都是0.**

## 5.Big-Endian大端模式和Little-Endian小端模式

定义

> 1. Little-Endian就是低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

2. Big-Endian就是高位字节排放在内存的低地址端，低位字节排放在内存的高地址端。
3. 网络字节序：TCP/IP各层协议将字节序定义为Big-Endian，因此TCP/IP协议中使用的字节序通常称之为网络字节序.
4. 高/低字节定义:在十进制中我们都说靠左边的是高位，靠右边的是低位，在其他进制也是如此。就拿0x12345678来说，从高位到低位的字节依次是0x12、0x34、0x56和0x78.

例子分析:

```
unsigned int value = 0x12345678
```

1.Big-Endian: 低地址存放高位

| **栈底 （高地址）** | | |
|---|---|---|
| buf[3] | (0x78) | 低位 |
| buf[2] | (0x56) | |

| **栈底 （高地址）** | | |
|---|---|---|
| buf[1] | (0x34) | |
| buf[0] | (0x12) | 高位 |
| **栈顶 （低地址）** | | |

2.Little-Endian: 低地址存放低位

| **栈底 （高地址）** | | |
|---|---|---|
| buf[3] | (0x12) | 高位 |
| buf[2] | (0x34) | |
| buf[1] | (0x56) | |
| buf[0] | (0x78) | 低位 |
| **栈 顶 （低地址）** | | |

在Little-endian模式CPU内存中的存放方式(假设从地址0x4000开始存放)

| **内存地址** | **0x4000** | **0x4001** | **0x4002** | **0x4003** |
|---|---|---|---|---|
| 存放内容 | 0x78 | 0x56 | 0x34 | 0x12 |

在Big- endian模式CPU内存中的存放方式则为

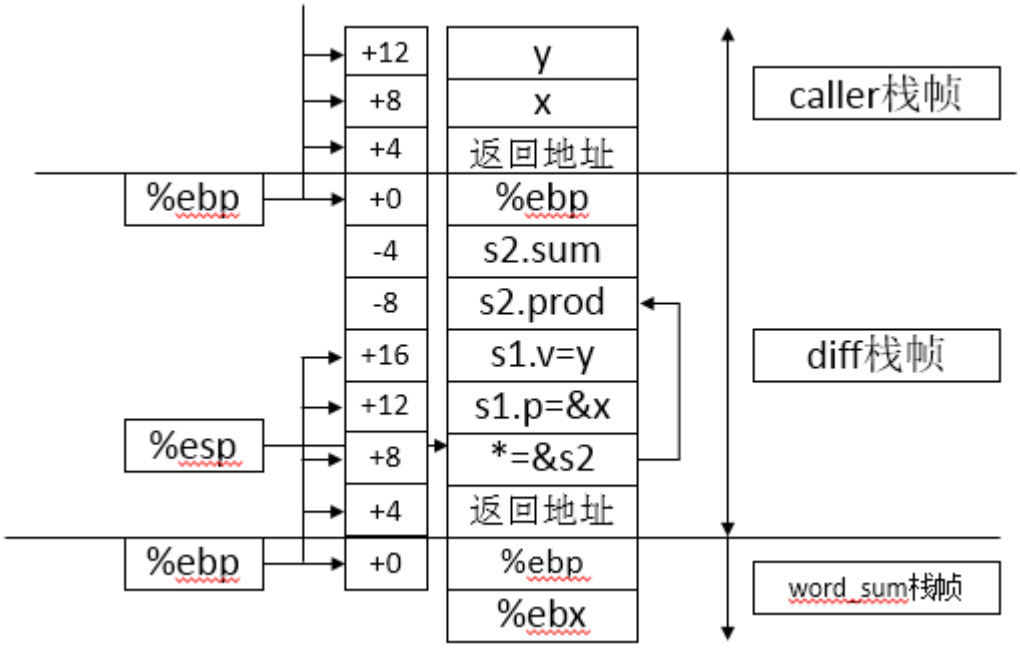
| **内存地址** | **0x4000** | **0x4001** | **0x4002** | **0x4003** |
|---|---|---|---|---|
| 存放内容 | 0x12 | 0x34 | 0x56 | 0x78 |

注意：通常我们说的主机序（Host Order）就是遵循Little-Endian规则。所以当两台主机之间要通过TCP/IP协议进行通信的时候就需要调用相应的函数进行主机序 （Little-Endian）和网络序（Big-Endian）的转换。
检查CPU是大端还是小端:

```
int checkCPU(void)
{
    union
    {
        int a;
        char b;
    }c;
    c.a = 1;
    return (c.b == 1);
}
```

# 6.过程调用

**1. 栈帧结构**




> 说明：
> 返回值在相对%ebp偏移量为4的位置；
> 第一个参数放在相对于%ebp偏移量为8的位置；

支持过程调用和返回的指令：

| 指令 | 描述 |
| --- | --- |
| call *Label* | 过程调用 |
| call **Operand* | 过程调用 |
| leave | 为返回准备栈 |
| ret | 从过程调用中返回 |

一个过程调用的整个汇编流程示意：

```
push ebp
mov ebp,esp
[sub esp,xxx]
[push xxx] ;寄存器压栈
... 其中利用eax edx传递函数返回值
[pop xxx]  ;寄存器出栈
mov esp,ebp
pop ebp
ret      ;从栈中取得返回地址并跳转
```