

Operating System - P33

哲学家就餐

同步问题

五个哲学家围坐在一张圆桌周围

每个哲学家面前都有一盘通心粉

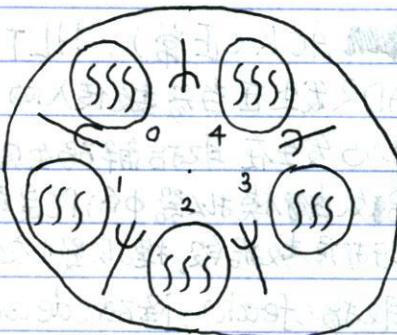
需要两把叉子才能吃到通心粉

相邻两个盘子之间放有一把叉子

哲学家有两种状态：吃饭和思考

只有当哲学家成功拿起两把叉子，才开始吃

为每一位哲学家写一段描述其行为的程序，且决不会出现死锁。



饥饿 (starvation) 表示所有的程序都在不停地运行，但都无法取得进展

方案一：是当哲学家拿不到右边叉子时等待一段随机时间，而非相同时间，则互锁的可能性降低

注意：虽然在实践中此方案工作良好，但依旧希望不能因为连串不可靠的随机数字而失败

方案二是用一个互斥量 mutex 来保证同一时间只有一个哲学家尝试拿起叉子

用一个信号量数组 state 来记录每个哲学家的状态 (思考, 吃饭, 饥饿)

#define N 5
用一个信号量数组来记录每个哲学家是否阻塞

#define LEFT (i+N-1)%N 邻居编号

#define RIGHT (i+1)%N

Void take_forks(int i){ 哲学家编号 i, [0, N)

#define THINKING 0 哲学家

#define HUNGRY 1 状态

#define EATING 2

typedef int semaphore;

int state[N]; i 是哲学家状态

semaphore mutex = 1; 互斥量

semaphore s[N]; 信号量

Void philosopher(int i){

while (TRUE){

}

}

}

}

此方案可以实现不仅没有死锁，

而且对于任意多位哲学家的情况

都可以获得最大的并行度

→ down(&mutex);
state[i] = HUNGRY; 将哲学家 i 状态置为饥饿
离开临界区 test(i); 哲学家 i 尝试拿起左右两把叉子
up(&mutex);
→ down(&s[i]);

Void put_forks(int i){
→ down(&mutex);
state[i] = THINKING; 哲学家 i 状态置为思考
test(LEFT); 其左右邻居 分别尝试左右
离开临界区 test(RIGHT); 两把叉子
up(&mutex);}

Void test(int i){
if (state[i] == HUNGRY &&
state[LEFT] != EATING &&
state[RIGHT] != EATING){
唤醒第 i 位哲学家
state[i] = EATING; 不处于吃饭
up(&s[i]);}}

Operating

System - P34

哲学家就餐问题用于建模互斥访问有限资源的竞争问题是通过共享一个资源来实现的。但如果有多个进程同时读数据库是可接受的，但如果有一个进程正在更新(写)数据库，则其他进程均无法访问数据库。

```
typedef int semaphore;
semaphore mutex = 1; // 互斥锁
semaphore db = 1; // 更新数据库的互斥锁
int rc = 0; // 正在读或即将读数据库的进程数目
```

```
Void reader (void){  
    while (TRUE){  
        if (rc == 0) down(&mutex); // 对rc的互斥锁  
        if (rc == 0) up(&db); // 第一个读者关闭写  
        read_data_base(); // 从数据库读数据  
        if (rc == 0) up(&mutex); // 退出时打开写  
    }  
}
```

```
Void writer (void){  
    while (TRUE){  
        if (rc == 0) down(&db); // 互斥地写数据库  
        write_data_base(); // 写入数据  
        if (rc == 0) up(&db); // 释放数据库  
    }  
}
```

在该方案中，第一个进入的读者对 db 进行 down 操作，其后进入的读者只是递增计数器 rc 而先退出的读者只是递减计数器 rc，最后一个退出的读者对 db 进行 up 操作

即允许一个被阻塞的写者访问数据库（如果存在的的话）

注意这里隐含了一个条件，即当读者使用数据库时，允许新来的读者进入，但会挂起新来的写者
即如果有一个稳定的读者流，则写者会被持续挂起，直到没有读者为止，其中其
如果读者流足够密集，则写者会被永久阻塞

帕金森定律

不管存储器有多大，程序都可以把它填满

分区存储器体系

若干 MB 的高速缓存(Cache): 快速，昂贵，易失性

若干 GB 的内存：速度与价格适中，同样易失性

hierarchy

若干 TB 的磁盘存储：低速，廉价，非易失性

存储管理器 (memory manager) 用于有效地管理内存

记录哪些内存正在使用，哪些内存是空闲的

在进程需要时为其分配内存 (allocate)，在进程使用完成后释放内存 (free)

Operating System - P35

09 - 10.03.2023

无存储器抽象：最简单的存储器抽象是根本没有抽象，每一个程序都直接访问物理内存。

如指令 `MOV REGISTER1, 1000` 使计算机将物理内存 ^{位置} 1000 的数据移到 REG1 中。

存储器模型就是简单的物理内存：从0到一个上限的地址集合。

每个地址对应可容纳一定数目的存储单元 (byte)

在这种模型下，要在内存中同时运行多个程序是不可能的，但仍存在一些可选项：

过去的大型机和小型机

PDA / 嵌入式系统

早期个人计算机 (MS-DOS)

	0xFF...	OS位于内存顶端 的 ROM 中 (只读存储器)	位于 ROM 中的 操作系统	设备驱动程序 位于内存顶端 的 ROM 中	位于 ROM 中的 设备驱动程序 (Basic, Input)	位于 RAM 中的 用户程序	这部分称为 BIOS (Basic, Input Output System)
地址集合	0x0000 - 0xFFFF	位于 RAM 中 OS位于 RAM 底部 (随机访问存储器)	用户程序	OS和其他部分 位于 RAM 底部	用户程序	OS和其他部分 位于 RAM 底部	基本输入输出系统
	0x0000 - 0xFFFF	向操作系统 (随机访问存储器)				操作系统	

注意：这两种方案由于用户程序可以接触 OS，误操作可能摧毁操作系统。

注意：在这种方式中，通常同一时刻只能有一个进程在运行。

虽然可以使用多线程编程来实现并行，即假设进程所有线程对同一内存映像可见。

但通常是希望同一时间运行没有关联的程序。

交换概念：只要在没有存储器的系统中某一时间内内存中只有一个程序就不会发生冲突。同时运行的程序

所以 OS 可以将当前内存中的所有内容保存到磁盘，然后读入下一个程序的内容，即实现了

保护键：在没有交换功能时，用于实现并发运行多个程序的特殊硬件。

如 IBM 360 的方案：划分内存为 2KB 的块。

每个块分配一个 4 位的保护键。

保护键存储在 CPU 的特殊寄存器中

PSW (Program Status Word)

程序状态字中存有一个 4 位码。

如果程序访问保护键不同的内存，会被硬件捕获。

只有 OS 可以修改保护键，从而保护用户进程和 OS。

注意：引用绝对物理地址最需要避免的，因为可能导致错误和程序崩溃。

静态重定位：是 360 的解决方案，在装载程序时为每个绝对地址加上一个偏移量。

问题是装载器需要能够区分逻辑地址，而且会减慢装载速度。

暴露物理地址：一、用户程序可以很容易地破坏操作系统。

二、在这种模型下同时运行多个程序很困难。

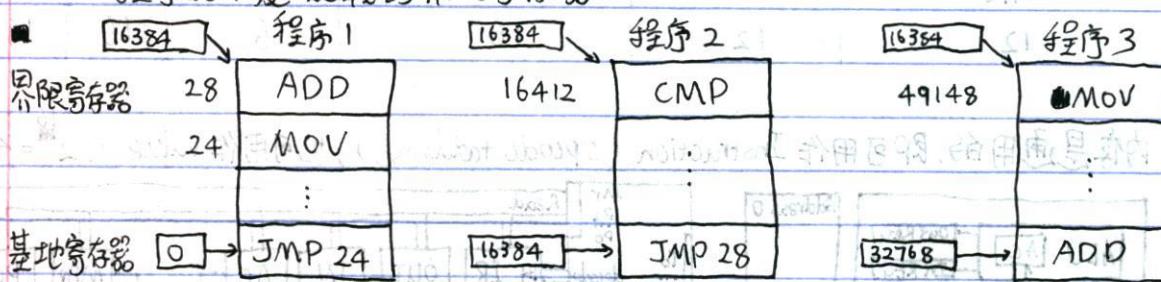
Operating System - P36

地址空间

是一个进程可用于寻址内存的一套地址集合，为程序创造了一种抽象的内存。每个进程都有自己的地址空间，并且会独立于其他进程的地址空间。存在一些特殊情况，需要进程间共享地址空间。

动态重定位

简单地把每个进程的地址空间映射到物理内存的不同部分。世界上最早的超级计算机CDC 6600 和原始 IBM PC 的心脏 (Intel 8088) 采用的经典方案：给每个 CPU 配置两个特殊寄存器，基址寄存器和界限寄存器。程序的起始物理地址装载到基址寄存器，程序的长度装载到界限寄存器。



基址寄存器和界限寄存器使程序可装载到内存的连续空间的位置

且装载期间无须重定位

而当进程访问内存时，CPU 硬件会自动把基址值加到进程发出的地址，再送到内存总线

同时检查程序提供的地址是否大于等于界限值

如果访问地址超过了界限，则产生错误并终止访问

如 JMP 28 指令会被硬件解释为 JMP 16412

缺点是每次访问内存都需要进行加法和比较运算，其中比较运算速度更快

但没有特殊电路支持时，加法会由于进位传递时间而显得缓慢

所有的进程所需的 RAM 数量总和通常要远远超出存储器能向多支持的范围；有两种方法处理内存超载

交换 (swapping) 把一个进程完整地调入内存，使该进程运行一段时间后再存回磁盘

空闲进程存放在磁盘上，所以当它们不运行时就不再占用内存

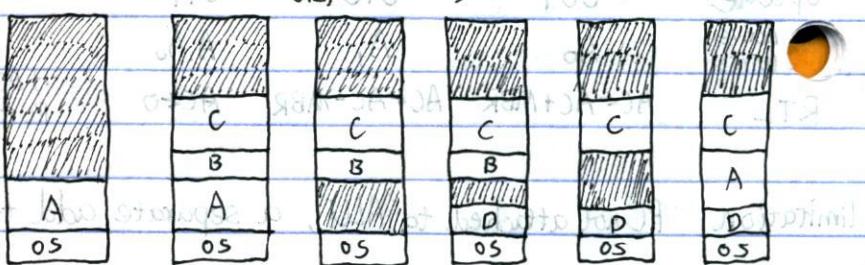
虚拟内存 (Virtual memory)，使程序能在只有一部分调入内存的情况下运行

交换系统中，当进程再次调入，其位置可能发生改变。

则换入时通过软件或程序运行其间

通过硬件对其地址进行重定位

如基址寄存器和界限寄存器



Operating

System - P37

内存紧缩 (memory compaction)

通过把所有进程尽可能向下移动，有可能将这些小的空闲区合成一大块
但因为需要耗费大量的CPU时间，所以通常不进行这个操作

注意问题：当进程被创建或被换入内存时，应该为它分配多大的内存

如果进程创建时其大小固定且不会改变，则OS可按其需要大小进行分配

由于程序语言允许从堆中动态地分配内存，使得进程的数据段可以增长

如果进程与空闲区相邻，可把空闲区分配给该进程

但如果与其他进程相邻，则无法自然增长

要么移动该进程到内存中一个足够大的区域，等同于重新分配内存

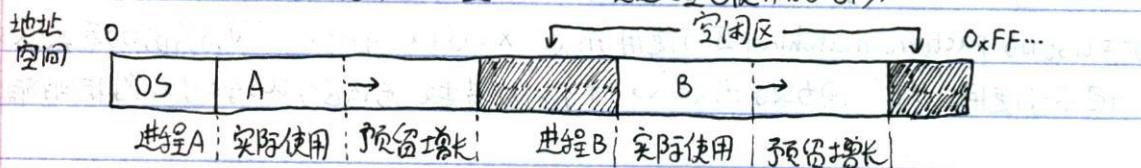
要么将若干进程交换出去，以便生成一个足够大的空闲区

如果进程在内存中无法增长，且磁盘交换区已满，那么只能阻塞或结束该进程

为了减少进程移动和交换所产生的开销

方案一是换入或移动进程时为其分配额外内存

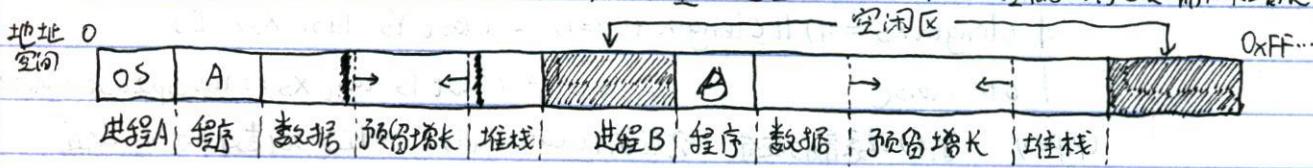
但当换出到磁盘时，只交换进程已使用的部分



方案二是进程拥有两个可增长的段

数据段 供变量动态分配和释放，作为堆使用，向上增长

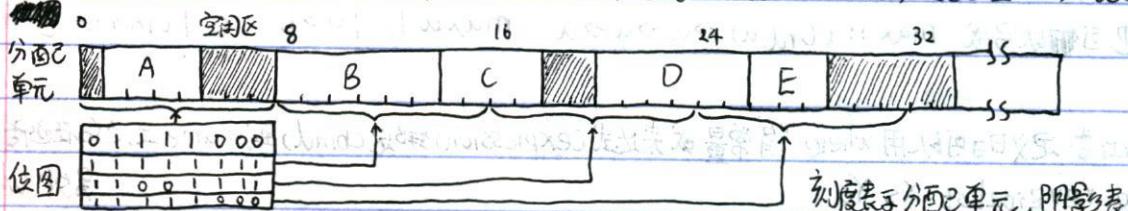
堆栈段 用于存放普通局部变量与返回地址，从进程的内存顶端向下增长



动态分配内存跟踪使用情况通常有两种方法：位图和空闲区链表

位图

将内存划分为大小固定的分配单元，每个分配单元对应于图中的一位，0表示空闲，1表示占用



注意：分配单元的大小决定了位图的大小，过小的分配单元需要大的位图，而过大的分配单元容易导致浪费

优点是提供了一种简单地利用一块固定大小的内存区就能对内存使用情况进行记录的方法，

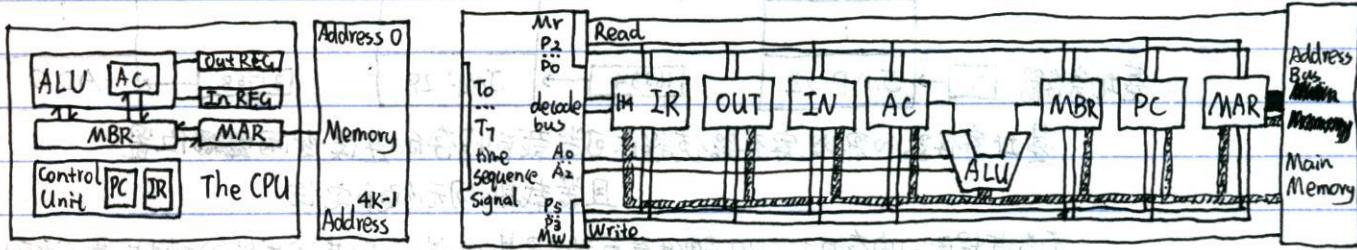
缺点是在位图中查找指定长度的连续0串（指定长度的空闲区）的操作很耗时

Machine
Architecture that is
Really
Intuitive and
Easy

code table	16-bits for instruction, higher 4-bits \rightarrow op-code, lower 12-bits \rightarrow address
Instruction Set	I/O Input 0x5 Input value Output 0x6 \ll AC [C \leftarrow 5th, 6th bits] [M[X] \leftarrow PC] PC \leftarrow X address pointer
Instruction	Add X Subt X AddI X Clear Load X Store X Jump X Skip cond C JnS X [†] Jump I X Store I Load I Hal
Hex code	0x3 0x4 0xB 0xA 0x1 0x2 0x9 0x8 0x0 0xC 0x7
summary	AC \leftarrow AC + X AC \leftarrow AC - X AC \leftarrow AC + M[X] AC \leftarrow 0 AC \leftarrow M[X] M[X] \leftarrow AC PC \leftarrow X -000 : AC < 0 -400 : AC = 0 -800 : AC > 0 PC \leftarrow M[X] M[PC] \leftarrow AC AC \leftarrow M[PC] End
Type	Arithmetic Data Transfer Branch Subroutine Indirect Addressing

Register Bank	Memory Address Register	Program Counter	Memory Buffer Register	Accumulator	Input	Output	Instruction Register
Number	1	2	3	4	5	6	7
Abbreviation	MAR	PC	MBR	AC	IN	OUT	IR
bits stored	12	12	16	16	16	16	16

内存是通用的，即可用作 Instruction (opcode + address)，也可用作 value ($2^{16} = 4^K \times 16\text{ bits}$)



• Read Control Bus

Abbreviation	M [MAR]	MAR	PC	MBR	AC	IN	OUT	IR
Register Num	N/A	001	010	011	100	101	110	111
Activated Wires	Mr.)	P ₀	P ₁	P ₁ P ₀	P ₂	P ₂ P ₀	P ₂ P ₁	P ₂ P ₁ P ₀

~~Write Control Bus~~ X read or write any register, a separate memory read wire and a write wire.

Abbreviation	MEMARJ	MAR	PC	MBR	AC	IN	OUT	IR
Register Num	N/A	001	010	011	100	101	110	111
Activated Wires	Mw P3	P4	P4P3	P5	P5P3	P5P4	P5P4	

ALU opcode Signals

operation	Addition	Subtraction	Clear	AC negative	AC zero	AC positive	default
opcode	001	010	011	100	101	110	111
Active Wires	A ₀	A ₁	A ₁ , A ₀	A ₂	A ₂ , A ₀	A ₂ , A ₁	A ₂ , A ₁ , A ₀
RTL	AC \leftarrow AC + MBR	AC \leftarrow AC - MBR	AC \leftarrow 0	if AC < 0 PC \leftarrow PC + 1	if AC = 0 PC \leftarrow PC + 1	if AC > 0 PC \leftarrow PC + 1	PC \leftarrow PC + 1

limitation PC not attached to ALU, a separate add to increment PC (hidden behind the PC)

CPU 指令 - Instruction

RegisterBank	%eax	%ecx	%edx	%ebx	%esp(堆栈)	%ebp(保护,寻址)	%esi	%edi
index	0	1	2	3	4	5	6	7

Instruction Formats	Byte 5	Byte 4	Byte 3	Byte 2	Byte 1	Byte 0	每个 instruction 由 8 个字节组成
No-oper	icode / fcode	Unused					无 operands 的简单指令
R-data	icode / fcode	rA rB	Unused				使用 1-2 个寄存器作为 operands
I-data	icode / fcode	rA rB	Immediate Value				使用 1-2 个寄存器和 1 个不超过 32-bit 的立即数作为 operands
J-data	icode / fcode	Immediate Value		Unused			使用 1 个不超过 32-bit 的立即数作为 operands

Instruction Set Byte 0 | 1 | 2 | 3 | 4 | 5 注意此处与上表相反, 因为此处为内存中实际地址的位移量

halt	00	-					No	Stop Processor.
nop	10	-					No	No-operation
rrmovl rA, rB	20	rA rB	-				R	Reg[rB] = Reg[rA]
cmovlxx rA, rB	2f	rA rB	-				R	If (xx → CC), Reg[rB] = Reg[rA]
irmovl V, rB	30	8 rB	Value				I	Reg[rB] = Value, 注意此处 rA 位置为 8, 是保护性占位
rmmovl rA, DcrB	40	rA rB	Dest				I	Mem[rB + Value] = Reg[rA] [注意 Value 为偏移量, 以及参数存放数据流]
mrmovl DcrB, rA	50	rA rB	Dest				I	Reg[rA] = Mem[rB + Value]
OPL rA, rB	6f	rA rB	-				R	Reg[rB] = Reg[rA] op Reg[rB]
jxx Dest	7f	Dest	-				J	If (xx → CC), PC = Value
jmp Dest	70	Dest	-				J	PC = Value
call Dest	80	Dest	-				J	%esp = %esp - 4, Mem[%esp] = PC, PC = Value
ret	90	-					No	PC = Mem[%esp], %esp = %esp + 4, 所以需要先将 %esp 存入 %ebp
pushl rA	A0	rA F	-				R	%esp = %esp - 4, Mem[%esp] = Reg[rA]
popl rA	B0	rA F	-				R	Reg[rA] = Mem[%esp], %esp = %esp + 4
(brkxx) brk	FF	-					No	扩展数据段

Condition codes (CC) 标志位, 分别是符号标志 (Sign), 零标志 (Zero), 盈出标志 (Overflow), 简称 SF, ZF, OF

CC 只有在 OPL 指令执行时才会改变, 根据当次计算的结果进行设置, 在下一次 OPL 指令前保持不变

XX 与 CC 的关系 对于 cmovxx 和 jxx 指令, 其判断依据为上一次 OPL 指令后设置的 CC 值

XX le (<=) l- (<) e- (=) ne (!=) ge (>=) g- (>)

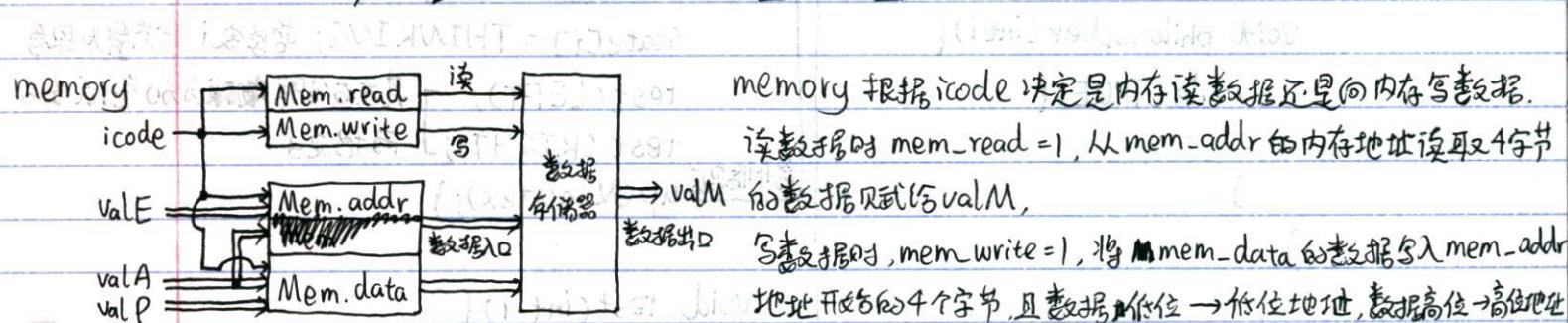
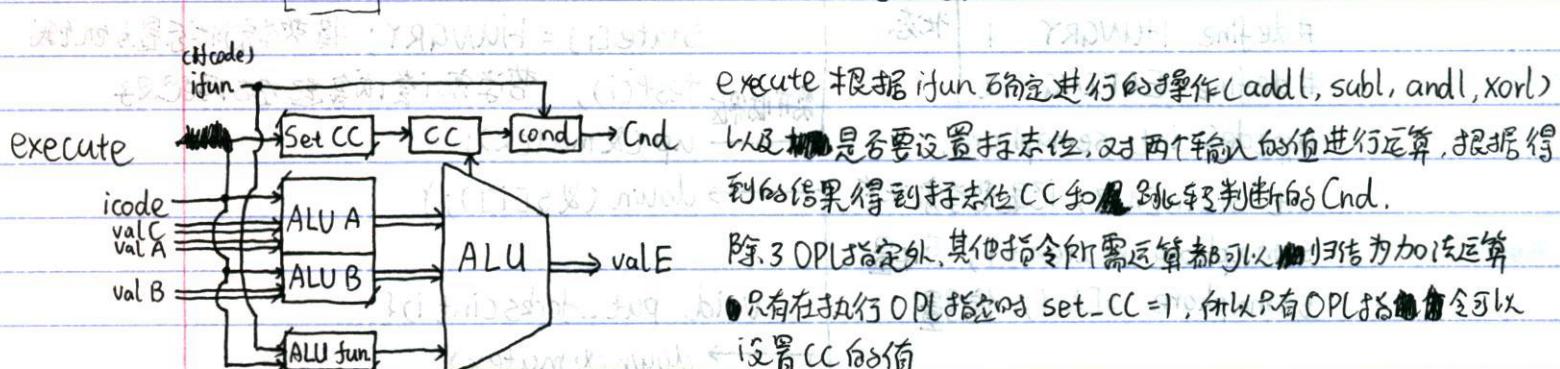
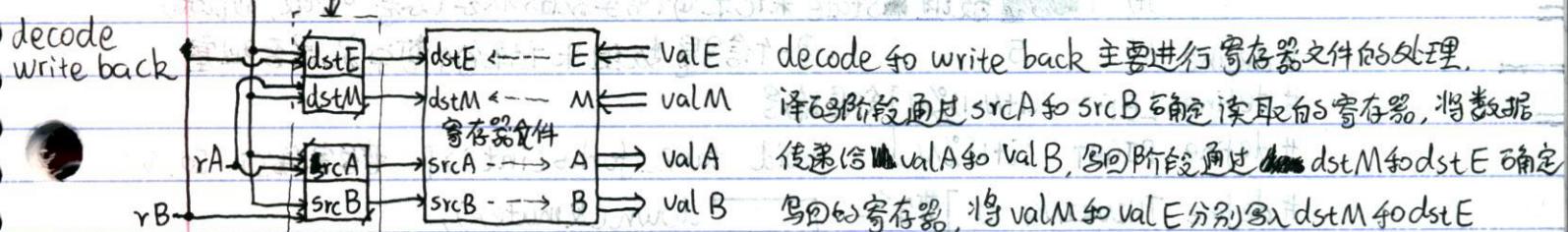
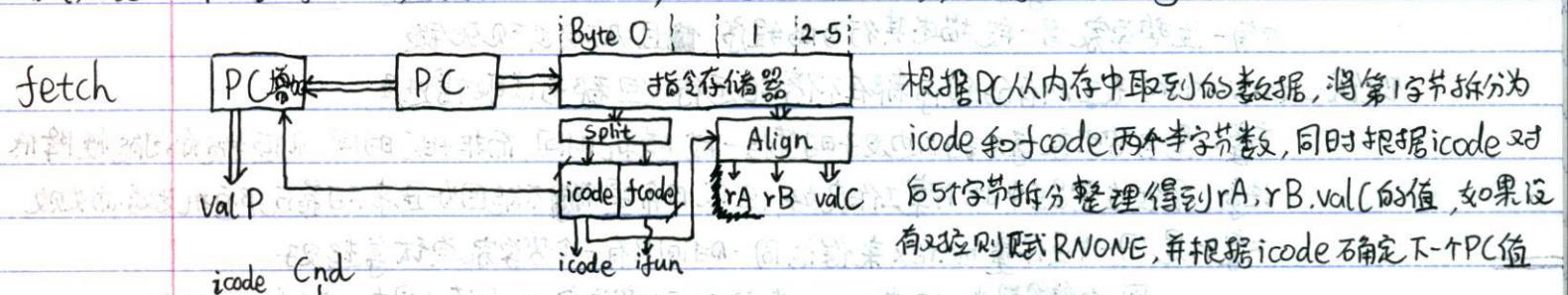
CC SF || ZF SF !SF ZF !ZF !SF !SF & !ZF

Memory

指令的存储是从指定的第一个字节开始, 顺序地从低位地址到高位地址, 指令之间连续存放, 由于指令长度不等, 所以由硬件解码时确定 PC 的偏移量, 数据的存储以字节为单位, 高位字节 → 低位地址 由于 word size 为 32-bit, 即 4 Bytes, 所以 Dest 通常为 4k, k ∈ Z

Y86

STAT	AOK(正常), HLT(执行 halt 指令后), ADR(非法地址), INS(非法指令)
状态码	ADR发生在当参数传入的 DcrB) 或 Dest 不在地址空间时, 注意模拟器中为 0x1fff 0x0000
brk	INS发生在取指解码失败时, 注意模拟器中由于忽略无效指令, 仅在寄存器错误时触发 另外, 模拟器中的 PC 是由上一条指令生成偏移量, 所以依旧可以继续运行 用于扩展数据段, 模拟器中效果为将 STAT 置于 DBG, 并将显存的地址上限 0xf0 提高到 0xffff,
指令分段	取指(fetch), 解码(decode), 执行(execute), 读存(memory), 写回(write back)



流水线:

数据冒险: 写后读(CRAW), 读后写(WAR), 写后写(WAW), 对于 RAW 通过数据前推实现, 不行就加入气泡

控制冒险: 指令流水线, 遇到分支指令无法在流水开始阶段判断出分支结果, 通过分支预测或数据前推解决

结构冒险: 一个存储单元被一条指令取操作数同时另一条指令要写入结果, 划分代码段和数据段, 保证程序不修改代码