

COMP0008 CA&C

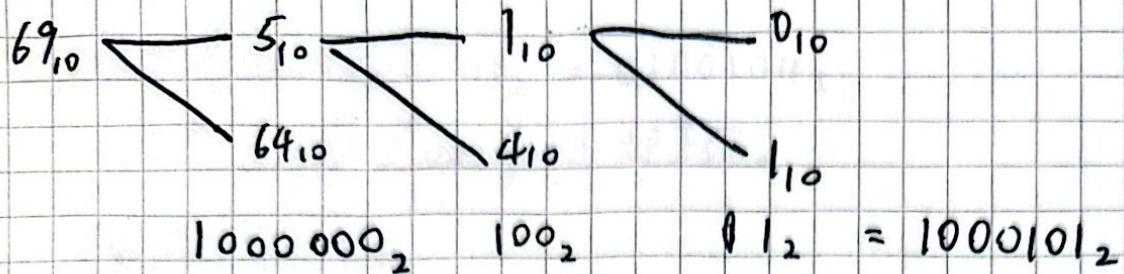
2024 Autumn

Based On

Ran Ben Basat

Stefano Vissicchio

D \rightarrow B



$$2^3 = 8 \quad 2^6 = 64 \quad 2^9 = 512$$

B \rightarrow D

$$1101_2 = 1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 = 13_{10}$$

B : 0b* 0b11 = 3

Octal : 0* 016 = 14

D : 无前缀

H 0x* 0x*1A = 26

Two's Complement :

$$\begin{array}{r} 0011 = 3 \\ + 1001 = -7 \\ \hline 1100 = -4 \end{array} \qquad \begin{array}{r} 0110 = 6 \\ - 1101 = -3 \\ \hline 1100 = 3 \end{array}$$

Signed: $-2^{n-1}, \dots, 2^{n-1}-1$ Unsigned: $0, \dots, 2^n-1$

二进制算法可能 overflows:

其中 $(pos) + (neg)$, $(pos) - (pos)$ 不会溢出
 $(neg) + (neg)$, $(pos) + (pos)$ 有可能。

那么为什么叫它二进制补码呢，因为

$$00000000\ 01001011_2 = 01001011_2$$

不论其是 signed / unsigned 都正确。

若最高位为 1, R:

unsigned: $00000000\ 11001011_2 = 11001011_2$

Signed : $11111111\ 11001011_2 = 11001011_2$

Byte = 8 bits Kilobyte (KB), Megabyte (MB) ...

Terminology for names in MIPS32:

8 bits : Byte

16 bits: Half word

32 bits : Word

64 bits: Double word

Address

4	.	.
3	-	-
2	:	..
1	:	..
0	← 8 bits →	

10	-	"
C	..	
8	:	..
4	:	..
0	← 32 bits →	

将 0xBABACAFE 和 9 放进 32 位内存，有 2 种顺序：

Big Endian

BA BA CA FE
0 0 0 9

Little Endian

FE CA BA BA
9 0 0 0

那么为什么叫它二进制补码呢，因为

$$00000000 \ 01001011_2 = 01001011_2$$

不论其是 signed / unsigned 都正确。

若最高位为1, R:

$$\text{@unsigned: } 00000000 \ 11001011_2 = 11001011_2$$

$$\text{Signed : } 11111111 \ 11001011_2 = 11001011_2$$

Byte = 8 bits Kilobyte (KB), Megabyte (MB) ...

Terminology for names in MIPS32:

8 bits: Byte 16 bits: Half word

32 bits: Word 64 bits: Double word

Address			
4	.	10	- -
3	- -	C	- -
2	- -	8	- -
1	- -	4	- -
0	← 8 bits →	0	← 32 bits →

将 0xBABACAFE 和 9 放进 32位内存，有2种顺序：

Big Endian

BA BA CA FE
0 0 0 9

Little Endian

FE CA BA BA
9 0 0 0

MIPS32 CPU has 32 registers: \$0, ..., \$31, each of size 32 bits.

Three instruction types:

1. Rithmetic

add \$8, \$1, \$2 sub \$12, \$6, \$3

2. Immediate

addi \$15, \$15, -1 beq \$10, \$14, 0x00000002

3. Jump

j 0x00400050 jal 0x0040007C

会有一个 Reference Card, 写了~~函数名~~, 类型, 函数功能, 和 OPCODE.

以 Add 为例

NAME, MNEMONIC FORMAT OPERATION

OPCODE/
FUNCT

Add add R $R[rd] = R[rs] + R[rt]$ 0/20_{hex}

顺序为表中顺序

add \$8, \$1, \$2 = 0x00224020

000000	00001	00010	01000	00000	100000
opcode	rs	rt	rd	shamt	funct

arguments

不同机器的位数不同, 若考试提供, 以卷面为准。反之以此为准。
(试卷结尾)

Pseudo Instructions:

move \$8 \$9 = add \$8, \$9, \$0

move. rd = register destination 注册目的地

rs = register operand (source) rt 是因为 t 是 S 的下一个字母。

I :

$$0x2108 + 11 = \text{addi } \$8, \$8, -1$$

001000 01000 01000 1111111111111111
Op code rs rt Immediate (16 bits)
 $= 0x08$

$$\text{addi} : R[rt] = R[rs] + \text{SignExtImm}$$

Memory addressing : $\begin{matrix} \text{memory} \\ \downarrow \\ \text{register} \end{matrix}$

$$\text{Store Word} \quad sw \quad I \quad M[R[rs] + \text{SignExtImm}] = R[rt] \quad 2b_{\text{hex}}$$

$$sw \quad \$9, \quad \underline{0x20(\$8)} = 0xad090020$$

101011 01000 01001 0000000000100000
sw rs rt

其中, \$8 中记录的是内存中的一个位置 (32 bits in MIPS32)

0x20 即偏移 32 位, 类似于 C 中的数组 [32]。

这条指令将 \$9 中的内容存到了这个 \$8 + 0x20 的位置。

$$\text{Load Word} \quad lw \quad I \quad R[rt] = M[R[rs] + \text{SignExtImm}] \quad 23_{\text{hex}}$$

$$lw \quad \$10, \quad 0x28(\$8) = 0x8d0a0028$$

找到 \$8 + 0x28 的位置, 读取一个 32 位的数据并写入到 \$10 中。

例如, \$8 中写着 0x10010000, 那 $0x20(\$8) = 0x10010020$

$$0x28(\$8) = 0x10010028$$

例：

Add Imm. Unsigned addiu I $R[rt] = R[rs] + \text{SignExtImm}$ 9 hex

① addiu \$0, \$0, 5

addiu \$8, \$0, 5

addiu \$8, \$8, 0xFFFF

求 \$8

\$0 = 0x0 (\$0 恒为 0)

\$8 = 0x5

\$8 = 0x4 (或 65540)

这是因为 MIPS 中的 addiu 是将 signed integers 相加并不关心溢出，其实操作是将 16 bit 的 0xFFFF 补成 0x11111111，并与 5 相加。教导地以机器行为要求学生并不道德，因为 $65535 + 5$ 符合表达式述。

注：用 lui (将其置 0) ori (可以扩展补 0) addu (加到 \$8) 的替代操作可以在 MIPS32 中实现 65540 的效果。

\$zero = \$0，其恒为 0，

\$t0 - \$t7 = \$8 - \$15

用于临时计算 (temporaries)

\$t8 - \$t9 = \$24 - \$25

\$s0 - \$s7 = \$16 - \$23 用于存储变量 (variables)

How high-level language constructs are converted into MIPS assembly.

```
int array[5];  
array[0] = array[0] * 2;  
array[1] = array[1] * 2;
```

\$S0，即\$16，存入一个array的位置，即 0x12348000。

```
lui $S0, 0x1234      # rt < imm < 16 , << 2 left shift.
```

```
ori $S0, $S0, 0x8000 # rt < rs | zero-ext(imm)
```

```
lw $t1, 0($S0)      # rt < Mem[rs + sign-ext(imm)]
```

```
sll $t1, $t1, 1      # rd < rt < shamt
```

```
sw $t1, 0($S0)      # Mem[rs+sign-ext(imm)] < rt
```

```
lw $t1, 4($S0)      # $t1 = array[1]
```

```
sll $t1, $t1, 1      # $t1 = $t1 * 2
```

```
sw $t1, 4($S0)      # array[1] = $t1
```

J:

Jump \rightarrow J $PC = \text{JumpAddr}$ 2hex

Jump And Link jal J $R[31] = PC + 8$; 3hex

\downarrow
\$ra $PC = \text{JumpAddr}$
return address

000010 0000010...0100 = 0x0100004
(26位)

为了最大化译码的效果，由于指令一定是字对齐的，这
26位不计末2位的零。

$(PC+4)[31:28]. \frac{00\ 00\ 01\ 0\ ... \ 01\ 00\ 0}{(28位)}$ = 0x0400010

将上一步结束后下一个word的高四位写入，可以在同256MB区域内直
接跳。

JumpRegister jr R $PC = R[rs]$ 0/08 hex

将地址写在寄存器中，RP可跳到任意位置。

If ($i == j$)

$t = g + h;$

else

$t = t - i;$

\$S0 = t, \$S1 = g \$S2 = h

\$S3 = i, \$S4 = j

bne \$S3, \$S4, L1

If ($rs != rt$) $PC \leftarrow PC + 4 + \text{offset} * 4$

addu \$S0, \$S1, \$S2 else

跳到t

if 完成 j done

后出 L1 : subu \$S0, \$S0, \$S3 ←

done :

while :

while: beq \$s0, \$t0, done

If ($rs == rt$) $PC \leftarrow PC + 4 + offset + 4$

这里用 beq 是因为 high-level code 写的是不等

④ ----- # while 内部语句

j while

done :

for 是 while 的语法糖，所以就不抄了。

jal 0x... # 跳到方程的位置, → Function Code

下一句 ← # RE[31] = PC + 8 是因为我抄的课本写的是 +8

课件写的是 +4, 19年试卷写的是 +0, jr \$ra

如果在一个函数中调用了另一个函数，会出现问题。因为跳回需要用到 \$ra 中的地址，但只能记一个。这样外部函数就回不去了。

故要将其放在内存中，而决定放置位置的机制与 stack 有关。

STACK	0x7fffffff
↓	← \$sp
↑	
DYNAMIC DATA	0x1001 0000
STATIC DATA	0x1000 0000
TEXT	0x00400000
RESERVED	0x0000 0000

```

addi $sp, -$sp, -8
sw $ra, 4($sp)
sw $fp, 0($sp)
addi $fp, $sp, 4
: # frame pointer 指向本节上界

```

```

lw $ra, 4($sp)
lw $fp, 0($sp)
addi $sp, $sp, 8
jr $ra

```

Syscall 是一个异常。通过抛出异常暂停处理器，切换到内核态并执行系统操作。

在调用指令之前，要在 \$V0 中写入调用号。在 \$A0-3 中放参数。

```
.data
str1: .asciiz "\n Enter a integer:"
str2: .asciiz " The number is:"

.text
.globl main

main: addi $V0, $0, 4    # 4: print_string
      la  $A0, str1    # 伪代码，将字符串地址写在 $A0
      syscall           # Print string.

      addi $V0, $0, 5    # 5: read_int
      syscall           # Read integer into $V0
      add  $T1, $0, $V0   # Store in temporary register.

      addi $V0, $0, 4
      la  $A0, str2
      syscall
      addi $V0, $0, 1    # 1: print_int
      add  $A0, $0, $T1   # 把要打印的整数直接写在 $A0
      syscall
      li  $V0, 10        # 10: exit
      syscall
```

UTF-32

32 bits = unicode characters

UTF8

0..... = ASCII (7位)

110.... 10..... = most of additional alphabets (11位)

1110... 1110... 10..... 10..... (16位)

11110... 10..... 10..... 10..... (21位)

IEEE 754 单精度

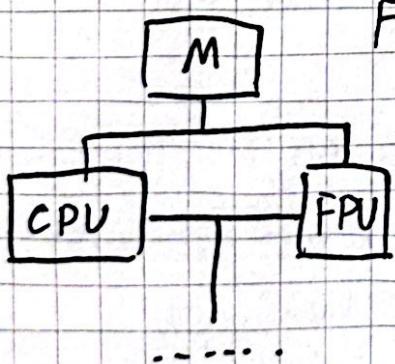
Sign(1bit) exponent argument (8bits) fraction (23 bits)

$$\text{Value} = (-1)^{\text{sign}} \cdot 2^{\text{EA}-127_{10}} \cdot (\text{1}. \text{fraction})$$

110000011 0100... (后面为0) \Rightarrow

$$(-1)^1 \cdot 2^{131-127} \cdot 1.01_2 = -16 \times 1.01_2 = -16 \times 1.25_{10} = -20_{10}$$

double \approx 1.11,52



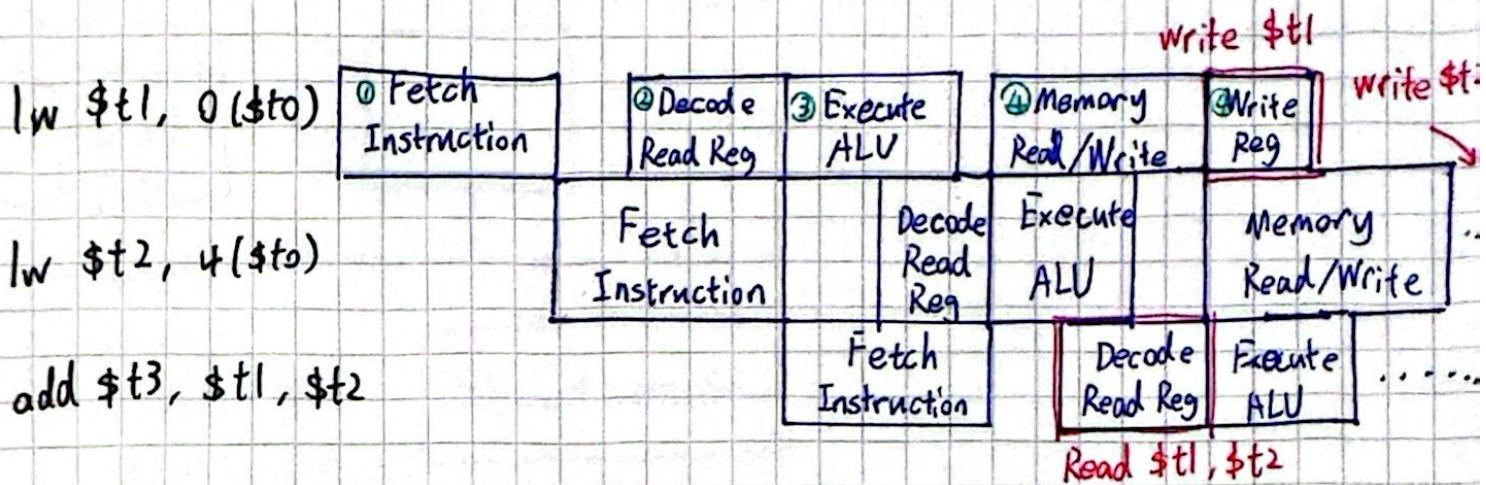
FPU 使用 arithmetic core instructions

lwc1 \$t1, 100(\$s2) 从内存中向 FPU 加载

swc1 \$t1, 100(\$s2) 从 FPU 向内存中写入

mtc1 \$t2, \$t2 从 CPU 中向 FPU 中加载

要进行以下三行指令，有一些内部部操作：



① Fetch Instruction: from memory

② Read register from the register file (读取rs等)

③ Execute : Sign_ext, compute memory address

④ Write to register file

对于红框框出的情况，说明对这种情况 add 还要再延后。可以记作：

lw ...

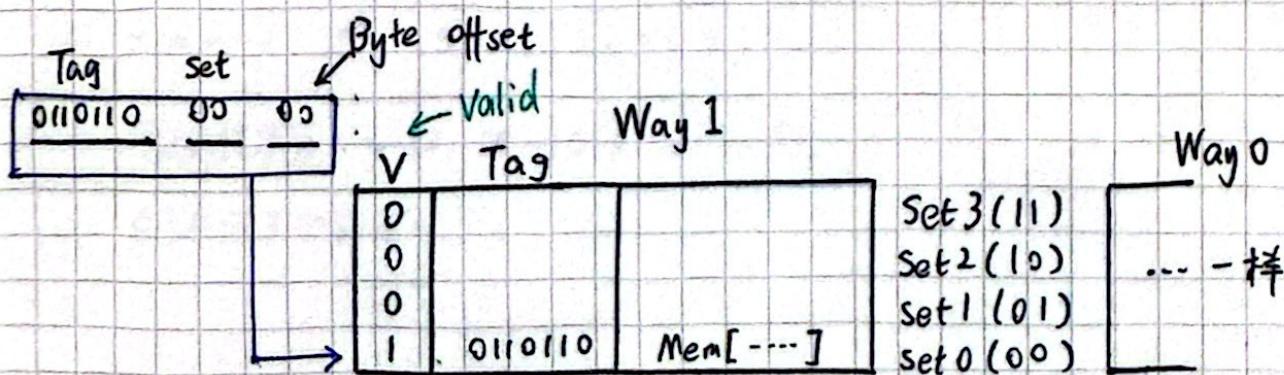
lw ...

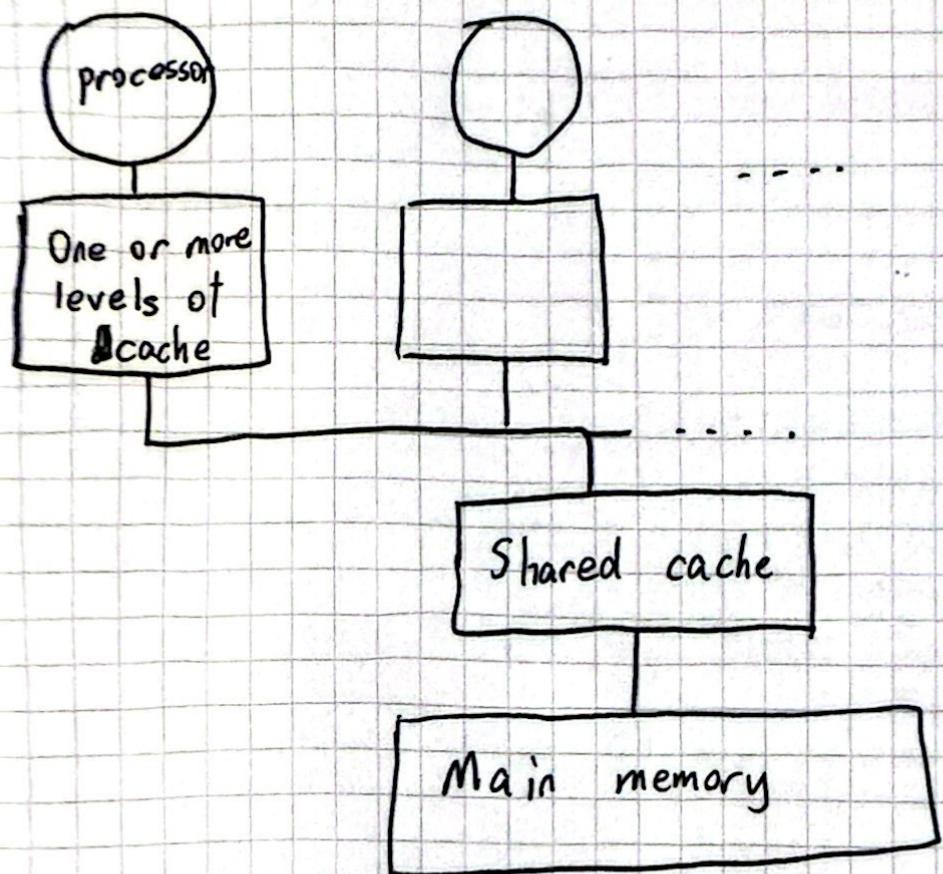
nop

add ...

Compilers 可能可以将其它语句在 nop 处执行来
进一步加速，但

向主内存请求数据实际上更慢，故在 Processor (即CPU) 附近会有一个 Cache，
它更小，但快很多。若可以在其中找到需求数据，就无须去 Memory 里找。





Thread : 线程

$A_1 \rightarrow A_2 \rightarrow A_3$ → Possible interleaving (对编程者来说, 不要预测
 硬件行为)

例, 对 $\{A_1, A_2\} \{B_1\}$, 有3种可能。 $\{A_1, A_2\} \{B_1, B_2\}$ 有6种可能,

对 $\{x=5 ; x=2*x\} \{x=x+2\}$, 要注意, 其中的表达式并不是 atomic.

$$x = 2 * x \Rightarrow t = x ; x = 2 * t ;$$

$$x = x + 2 \Rightarrow s = x ; x = s + 2 ;$$

可能的结果有 $x = 12, 14, 10, 2, 4, 7$.

自己在运算中的要分开

现在的设备可以有真正的并行性(true parallelism)了，即线程可以在不同的核上运行。

Thread safety implies no race condition

想象一个服务器是用来分解质因数的 (如果0008考写java代码，我建议你直接走出考场以示抗议。但一定要能看得懂)

题目代码：100行左右
Public class SimpleFactorizer implements Servelet {

public synchronized void service (ServletRequest req ,
ServletResponse resp) {

BigInteger i = extract (req);

BigInteger[] factors = null;

if (i.equals (lastNumber)) { #当用户请求分解的数与上一个相同

factors = lastFactors . clone(); [A] lastNumber . equals (X)

}

if (factors == null) {

factors = factorize (i);

lastNumber = i; [B] IN = Y

lastFactors = factors; [B] IF = f (Y)

}

encodeInResponse (resp , factors); } }

synchronized 是给当前对象 this 锁定，确保是同一个 servelet 实例在并发调用 service。若消除这个描述，会出现两种情况，如上。

但是众所周知(或0017你会学)分解算法可能很耗时。若禁止整个方法被同时进入，会排很长的队。要缩小锁的粒度。代码见下一页。

```

public void service(.....)
    BigInteger ..... 不被 synchronized.
    BigInteger[]
        synchronized(this) {
            if(i.equals(lastNumber)){ factors = lastFactors.clone(); }
        }
        if(factors == null){
            factors = factorize(i);
            synchronized(this) {
                lastNumber = i; lastFactors = factors;
            }
        }
    .....
}

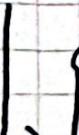
```

若两段方法均被 synchronized，就无法 run concurrently on the same instance.

上述问题可以叫 interference. 而并发程序中另一类问题是 visibility.
方法依然是 synchronization.

- ① No thread sees a partially constructed object.
- ② All threads see changes to the object's state.

Java Memory Model (JMM) guarantees visibility of one thread's writes.


 published: An object is available to code outside its current context
 safe published: Both reference and state of object are visible to other threads at the same time.

Remark:

Thread safety implies no race condition

Safe publication implies no partially construction

For a mutable object, to make sure no thread sees a partially constructed object, it must be safely published; to make sure all threads see changes to the object's state, it must be thread-safe or guarded by lock.

```
③ volatile (weak synchronization) ④ final ⑤ public private
public class ConfigSettings {
    public Holder holder;
    public ConfigSettings() {
        holder = new Holder(42);
    }
}
② public static Holder
holder = new Holder(42);
#使holder成为ConfigSetting的
静态属性。
⑤ public synchronized Holder
GetHolder() { return holder; }

public class Holder {
    private final int n; ① final #使之immutable, 完全不能改变。
    public Holder(int n) { this.n = n; } ② effectively immutable
                                         (发布后不变, 经 safely published)
    public void MyTest() {
        if (n != n) throw new AssertionError("error!");
    }
}
```

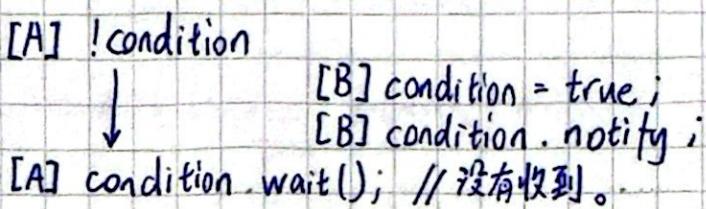
对于③中的volatile，其确保当这种变量被某个线程改变，其他线程可以立即看到此结果，于是不再读取自己工作缓存中过期的值。同时它禁止编译器对指令重新排序。

与synchronized不同，其只保证可见性，不保证原子性。

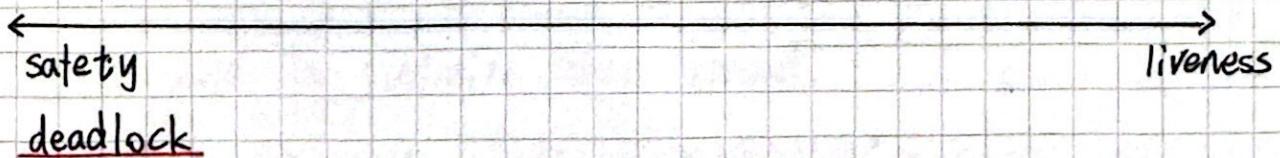
Condition Queues :

```
try {  
    ↗ condition predicate  
    while (!condition){  
        condition.wait(); // 会被 condition.notify() 取醒  
    }  
    // do something  
} finally {  
    lock.unlock();  
}
```

可能会有 missed signals 的情况, 如图



Correctness : safety + liveness.



- ① conflicting methods: Left Right Deadlock [A] getleftLock [B] getrightLock
Multiple Databases [A] recordSuccess [B] recordFail
- ② conflicting external calls: transferMoney [A] lockA [B] lockB; requireA
Dining Philosophers
- ③ Cross objects : Taxi dispatcher [A] taxi → dispatcher [B] dispatcher → taxi.

Open calls: Call the external method with no lock held

But loss of atomicity.

可以解决 Taxi dispatcher 问题，若可以确保内部是 thread safe 的。

Order resource if (fromId < toId) {
 synchronized (fromAcc, toAcc) { }

 }
 else synchronized (toAcc, fromAcc) { }

Acquire with timeout

Starvation 饿死

- ① Low-priority threads starved by high-priority ones.
- ② Lock held by infinite loop thread.
- ③ Readers & Writers (写入时不能有其它操作，可以单方优先，或信号控制)

Livelock

- ① 那个两个机器人来回走的 meme.
- ② Poison message problem (反复重试一条不可能成功的输入)
- ③ WiFi Link (收不到 confirmation 的 client 继续尝试连接占用频段)

Incorrect conditional sync

主要是 missed signals, 写在最开头了。

最后一个简单的例子。

```
public class SecretHolder {  
    private int secret = 0;  
  
    public synchronized void down() {  
        if (secret == 0) { wait(); } // 错在这种结构，应是 while.  
        secret --;  
        notify(); // 叫醒随机一个  
    }  
  
    public synchronized void up() {  
        secret++;  
        notifyAll();  
    }  
}
```

有 [A] down() [B] down() [C] up()。开始只有 [C] 能执行，但 [A][B] 可能先进入 wait()。[C] 结束时把 [A][B] 全部叫醒，且计语句已被跳出，故结果为 -1。
也有可能是 0 或 1，若 [A][B] 开始地晚就收不到 notify() 了。

哦咩得多，下课。