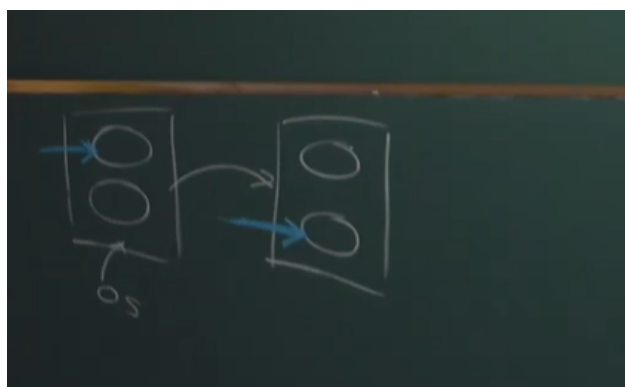


第5讲 多处理器编程：从入门到放弃（线程库，现代处理器架构，宽松内存模型）

1. 入门：线程库

概念：操作系统是状态机的管理者。每个程序都是一个状态机。操作系统也是一个状态机。



传统应该先讲进程；因为我们引入了状态机的概念，使得讲并发比较容易：

1. 操作系统是状态机的管理者，因此操作系统引入了一些可以共享的状态；
2. 比如操作系统中的文件，你在vscode里面编写完成，编译器却打不开，就见鬼了。
3. 所以操作系统是世界上最早的并发程序之一。

```
C hello.c
1  #include "thread.h"
2
3  void Thello(int id) {
4      while (1) {
5          // printf("%c", "_ABCDEFGHIJKLMNOPQRSTUVWXYZ[id]");
6      }
7  }
8
9  int main() {
10     for (int i = 0; i < 2; i++) {
11         spawn(Thello);
12     }
13 }
14
```

如果是2，我们发现cpu占用率是200%，如果是3，就是300%，我们就可以确认，我们的线程库是可以使用多处理器的！

```
tmux
top - 10:21:13 up 24 min, 3 users, load average: 2.92, 2.12, 1.35
Tasks: 394 total, 1 running, 393 sleeping, 0 stopped, 0 zombie
%Cpu(s): 17.1 us, 1.1 sy, 1.8 ni, 79.3 id, 0.1 wa, 0.0 hi, 0.6 si, 0.0 st
MiB Mem : 7725.2 total, 310.1 free, 4543.0 used, 2872.1 buff/cache
MiB Swap: 4096.0 total, 4095.2 free, 0.8 used, 2798.9 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 13065 jyy        20   0 19164   948  860  S 200.0   0.0   0:49.23 a.out
 2058 jyy        20   0 5145724 1.1g 256384 S  87.4  14.5 16:23.02 obs
 1009 root         0 -20    0     0     0  I   4.7   0.0   0:28.31 kworker/+
 2723 root         0 -20    0     0     0  I   4.0   0.0   0:16.65 kworker/+
 3530 jyy        39  19 2481560 984.9m 47404 S   4.0  12.7   0:28.19 tracker-+
 1358 jyy         9 -11 1645992 32320 23064 S   2.3   0.4   0:37.80 pulseaud+

$ ./a.out
```

```
tmux
top - 10:21:31 up 24 min, 3 users, load average: 2.66, 2.11, 1.36
Tasks: 397 total, 1 running, 396 sleeping, 0 stopped, 0 zombie
%Cpu(s): 23.2 us, 1.1 sy, 1.5 ni, 73.5 id, 0.0 wa, 0.0 hi, 0.7 si, 0.0 st
MiB Mem : 7725.2 total, 177.3 free, 4674.3 used, 2873.7 buff/cache
MiB Swap: 4096.0 total, 4095.2 free, 0.8 used, 2667.6 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 13482 jyy        20   0 27360   948  860  S 299.3   0.0   0:16.89 a.out
 2058 jyy        20   0 5145724 1.1g 256384 S  81.7  14.5 16:37.92 obs
 1009 root         0 -20    0     0     0  I   4.0   0.0   0:28.99 kworker/+
 1010 root         0 -20    0     0     0  I   3.3   0.0   0:24.76 kworker/+
 1358 jyy         9 -11 1645992 32320 23064 S   2.3   0.4   0:38.23 pulseaud+
 2723 root         0 -20    0     0     0  I   2.3   0.0   0:17.22 kworker/+

$ ./a.out
^C
$ gcc hello.c -o a.out
```

多线程共享内存并发

如何证实线程真的是共享内存的？

多线程共享内存并发：入门

多处理器编程：一个 API 搞定

```
#include "thread.h"

void Ta() { while (1) { printf("a"); } }
void Tb() { while (1) { printf("b"); } }

int main() {
    create(Ta);
    create(Tb);
}
```

- 这个程序可以利用系统中的多处理器
 - 操作系统会自动把线程放置在不同的处理器上
 - CPU 使用率超过了 100%

多线程共享内存并发

线程：共享内存的执行流

- 执行流拥有独立的堆栈/寄存器

简化的线程 API (thread.h)

- spawn(fn)
 - 创建一个入口函数是 fn 的线程，并立即开始执行
 - void fn(int tid) { ... }
 - 参数 tid 从 1 开始编号
 - 行为: sys_spawn(fn, tid)
- join()
 - 等待所有运行线程的返回 (也可以不调用)
 - 行为: while (done != T) sys_sched()

```
C hello.c
1  #include "thread.h"
2
3  int x = 0;
4
5  void Thello(int id) {
6      x++;
7      printf("%d\n", x);
8  }
9
10 int main() {
11     for (int i = 0; i < 10; i++) {
12         spawn(Thello);
13     }
14 }
15
```

```
tmux
top - 10:24:10 up 27 min, 3 users, load average: 0.99, 1.66, 1.31
Tasks: 395 total, 2 running, 393 sleeping, 0 stopped, 0 zombie
%Cpu(s): 4.7 us, 1.0 sy, 1.5 ni, 92.2 id, 0.1 wa, 0.0 hi, 0.6 si, 0.0 st
MiB Mem : 7725.2 total, 259.9 free, 4682.8 used, 2782.5 buff/cache
MiB Swap: 4096.0 total, 4092.7 free, 3.3 used. 2656.0 avail Mem

  PID USER      PR  NI  VIRT  RES  SHR S %CPU  %MEM    TIME+  COMMAND
 2058 jyy        20   0 5145724 1.1g 256384 S 80.4  14.5   18:48.33 obs
 1010 root         0  -20      0      0      0 I  3.3   0.0    0:29.26 kworker/+
 3530 jyy        39  19 2612632 1.0g 47244 S  3.3  13.7    0:32.77 tracker-+
 11490 root         0  -20      0      0      0 R  3.0   0.0    0:05.23 kworker/+
 8726 root         0  -20      0      0      0 I  2.7   0.0    0:17.28 kworker/+
 1358 jyy         9  -11 1645992 32320 23064 S  2.3   0.4    0:41.97 pulseaud+

$ gcc hello.c 66 ./a.out
1
4
5
2
3
7
8
6
9
10
$
```


There are many different attributes that must be handled when providing machine-level support for procedures. For discussion purposes, suppose procedure P calls procedure Q, and Q then executes and returns back to P. These actions involve one or more of the following mechanisms:

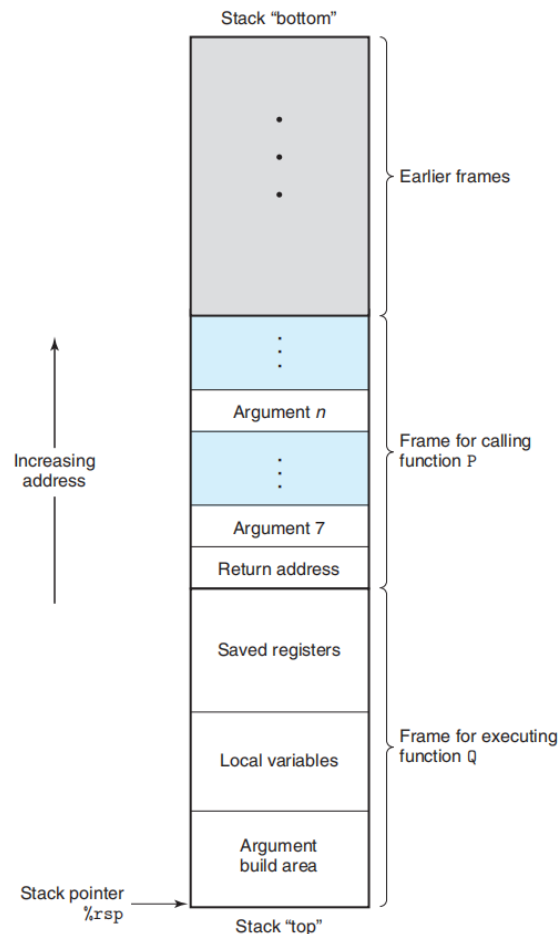
Passing control. The program counter must be set to the starting address of the code for Q upon entry and then set to the instruction in P following the call to Q upon return.

Passing data. P must be able to provide one or more parameters to Q, and Q must be able to return a value back to P.

Allocating and deallocating memory. Q may need to allocate space for local variables when it begins and then free that storage before it returns.

Figure 3.25

General stack frame structure. The stack can be used for passing arguments, for storing return information, for saving registers, and for local storage. Portions may be omitted when not needed.



控制转移

Passing control from function P to function Q involves simply setting the program counter (PC) to the starting address of the code for Q. However, when it later comes time for Q to return, the processor must have some record of the code location where it should resume the execution of P. This information is recorded in x86-64 machines by invoking procedure Q with the instruction `call Q`. This instruction pushes an address A onto the stack and sets the PC to the beginning of Q. The pushed address A is referred to as the *return address* and is computed as the address of the instruction immediately following the `call` instruction. The counterpart instruction `ret` pops an address A off the stack and sets the PC to A.

`call = push + jmp`

`ret = pop + jmp`

这里也可以看到，每个procedure都有自己的stackframe，上面存放了自己的local variable。

2. 放弃：原子性

例子：求和

分两个线程，计算 $1 + 1 + 1 + \dots + 1$ (共计 $2n$ 个 1)

```
#define N 100000000
long sum = 0;

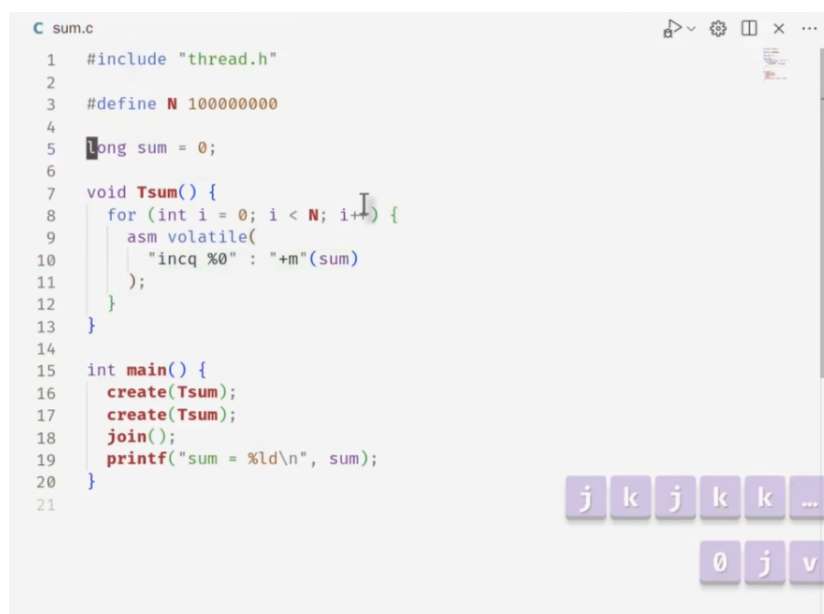
void Tsum() { for (int i = 0; i < N; i++) sum++; }

int main() {
    create(Tsum);
    create(Tsum);
    join();
    printf("sum = %ld\n", sum);
}
```

可能的结果

- 119790390, 99872322 (结果可以比 N 还要小), ...
- 直接使用汇编指令也不行

我们就算是把 `sum++` 改为一条 x86-64 的指令，我们都没有办法做到让结果是 $2n$ ：



```
C sum.c
1  #include "thread.h"
2
3  #define N 100000000
4
5  long sum = 0;
6
7  void Tsum() {
8      for (int i = 0; i < N; i++) {
9          asm volatile(
10             "incq %0 : '+m'(sum)"
11             );
12      }
13  }
14
15  int main() {
16      create(Tsum);
17      create(Tsum);
18      join();
19      printf("sum = %ld\n", sum);
20  }
21
```

这就是多处理器！

```
fish
sum = 109506671
$ ./a.out
sum = 110871248
$ ./a.out
sum = 110554137
$ ./a.out
sum = 102387548
$ ./a.out
sum = 128193726
$ ./a.out
sum = 122332556
$ ./a.out
sum = 105269485
$ gcc sum.c
$ gcc sum.c -O2
$ objdump -d a.out | less
$ ./a.out
sum = 100534922
$ ./a.out
sum = 100310216
$ ./a.out
sum = 103468336
$ ./a.out
sum = 115257063
$ ./a.out
sum = 101418193
$
```

所以我们最重要的基本假设被打破了，我们最基本的世界观被摧毁了。（当然这个行为只有在多处理器行为会碰到，如果只有一个cpu，中断一定是发生在指令的边界上的，但是这里我们必须放弃代码甚至指令的原子性）

这里其实我觉得我学到的就是多处理器上的，以前我ics的时候没想到写一条汇编指令也会挂掉。

放弃 (1): 指令/代码执行原子性假设

“处理器一次执行一条指令”的基本假设在今天的计算机系统上不再成立(我们的模型作出了简化的假设)。

单处理器多线程

- 线程在运行时可能被中断，切换到另一个线程执行

多处理器多线程

- 线程根本就是并行执行的

(历史) 1960s, 大家争先在共享内存上实现原子性(互斥)

- 但几乎所有的实现都是**错的**，直到 [Dekker's Algorithm](#)，还只能保证两个线程的互斥

放弃原子性假设的后果

printf 还能在多线程程序里调用吗？

```
void thread1() { while (1) { printf("a"); } }
void thread2() { while (1) { printf("b"); } }
```

我们都知道 printf 是有缓冲区的(为什么?)

- 如果执行 `buf[pos++] = ch` (pos 共享) 不就 **爆了** 了吗？

RTFM!

文档告诉我们，printf其实并没有爆栈：

```
man

truncated due to this limit, then the return value is the number of
characters (excluding the terminating null byte) which would have been
written to the final string if enough space had been available. Thus,
a return value of size or more means that the output was truncated.
(See also below under NOTES.)

If an output error is encountered, a negative value is returned.

ATTRIBUTES
For an explanation of the terms used in this section, see at-
tributes(7).



| Interface                                                                                                                                                          | Attribute     | Value          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|----------------|
| <b>printf()</b> , <b>fprintf()</b> ,<br><b>sprintf()</b> , <b>snprintf()</b> ,<br><b>vprintf()</b> , <b>vfprintf()</b> ,<br><b>vsprintf()</b> , <b>vsnprintf()</b> | Thread safety | MT-Safe locale |



CONFORMING TO
fprintf(), printf(), sprintf(), vprintf(), vfprintf(), vsprintf():
POSIX.1-2001, POSIX.1-2008, C89, C99.

snprintf(), vsnprintf(): POSIX.1-2001, POSIX.1-2008, C99.
Manual page printf(3) line 373/530 83% (press h for help or q to quit)
```

3. 放弃：顺序

例子：求和 (再次出现)

分两个线程，计算 $1 + 1 + 1 + \dots + 1$ (共计 $2n$ 个 1)

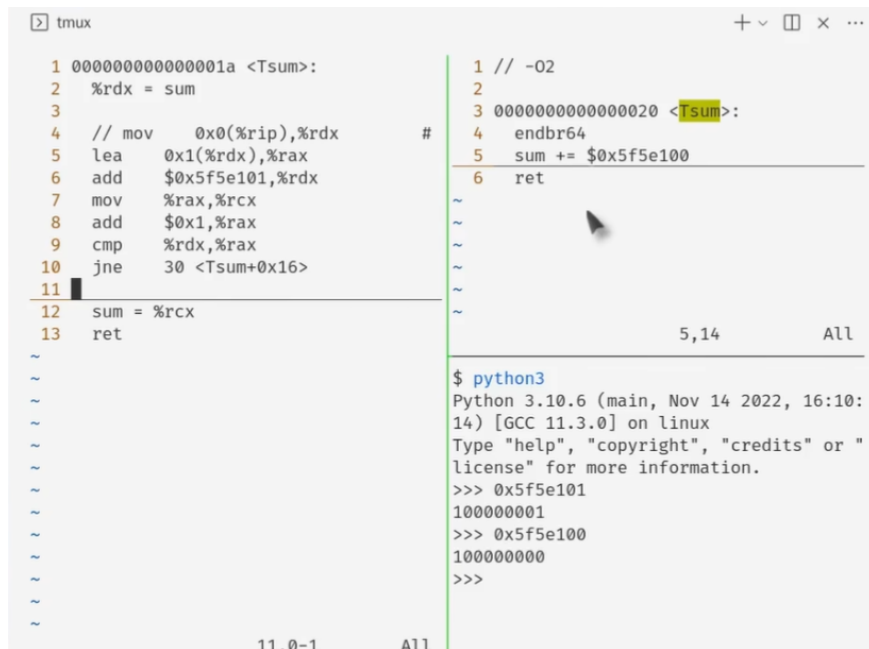
```
#define N 100000000
long sum = 0;

void Tsum() { for (int i = 0; i < N; i++) sum++; }

int main() {
    create(Tsum);
    create(Tsum);
    join();
    printf("sum = %ld\n", sum);
}
```

如果添加编译优化？

- -O1: 1000000000 🤖🤖
- -O2: 2000000000 🤖🤖🤖



```
tmux
1 0000000000000001a <Tsum>:
2  %rdx = sum
3
4  // mov    0x0(%rip),%rdx    #
5  lea     0x1(%rdx),%rax
6  add     $0x5f5e101,%rdx
7  mov     %rax,%rcx
8  add     $0x1,%rax
9  cmp     %rdx,%rax
10 jne     30 <Tsum+0x16>
11
12 sum = %rcx
13 ret
~
~
~
~
~
~
~
~
~
~
11,0-1    All

1 // -O2
2
3 00000000000000020 <Tsum>:
4  endbr64
5  sum += $0x5f5e100
6  ret
~
~
~
~
~
~
~
~
~
~
5,14      All

$ python3
Python 3.10.6 (main, Nov 14 2022, 16:10:
14) [GCC 11.3.0] on linux
Type "help", "copyright", "credits" or "
license" for more information.
>>> 0x5f5e101
100000001
>>> 0x5f5e100
100000000
>>>
```

左边是 `-o1` 优化，右边是 `-o2` 优化。

放弃 (2)：程序的顺序执行假设

编译器对内存访问“eventually consistent”的处理导致共享内存作为线程同步工具的失效。

刚才的例子

- `-O1: R[eax] = sum; R[eax] += N; sum = R[eax]`
- `-O2: sum += N;`
- (你的编译器也许是不同的结果)

另一个例子

```
while (!done);
// would be optimized to
if (!done) while (1);
```

解释：

如果编译器假设我们的程序是顺序程序，它可以优化成，把循环的判断条件提到外面来。因为对于独占的顺序程序来说，while的循环体里面没有改过我们的变量。所以变量会一直不变。

4. 放弃：处理器间的可见性

```
1 int x = 0, y = 0;
2
3 void T1() {
4   x = 1; int t = y; // Store(x); Load(y)
5   printf("%d", t);
6 }
```

```

7
8 void T2() {
9     y = 1; int t = x; // Store(y); Load(x)
10    printf("%d", t);
11 }

```

遍历模型告诉我们：01, 10, 11

- 机器永远是对的
- Model checker 的结果和实际的结果不同 → 假设错了

但是事实是：

```

fish
$ ./a.out | head -n 10000 | sort | uniq -c
 9700 0 0
   278 0 1
    22 1 0
$ ./a.out | head -n 100000 | sort | uniq -c
 90272 0 0
  9408 0 1
   320 1 0
$ ./a.out | head -n 1000000 | sort | uniq -c
961895 0 0
 34272 0 1
  3829 1 0
     4 1 1
$

```

为什么程序（我们天真地认为“程序每次执行一步”）得到了00这种东西？？？

实现处理器

我们自己都实现过简单的处理器：fetch, decode, execute...

我们希望我们跑得更快一点！

我们有两条语句：

```

1 mov $1,%eax
2 mov $1,%ebx

```

原则上你应该先执行1再执行2，但是没有任何人阻止你在一个时钟周期里面执行多条指令！

汇编指令其实还会被翻译成更小的指令！这解释了它为什么也不是原子的！

🍌 现代处理器也是 (动态) 编译器！

错误 (简化) 的假设

- 一个 CPU 执行一条指令到达下一状态

实际的实现

- 电路将连续的指令“编译”成更小的 μops
 - $\text{RF}[9] = \text{load}(\text{RF}[7] + 400)$
 - $\text{store}(\text{RF}[12], \text{RF}[13])$
 - $\text{RF}[3] = \text{RF}[4] + \text{RF}[5]$



在任何时刻，处理器都维护一个 μop 的“池子”

- 与编译器一样，做“顺序执行”假设：没有**其他处理器“干扰”**
- 每一周期执行尽可能多的 μop - 多路发射、乱序执行、按序提交

所以我们来解释一下前面为什么会出现 0 0：

1. Store x
2. Load y

编译器知道 $x \neq y$ ，所以这俩从顺序执行的意义上面不会影响最终的 eventual consistency。所以编译器可以做乱序，或者当 cpu 支持的时候，甚至可以同时做两条指令。

放弃 (3)：多处理器间内存访问的即时可见性

满足单处理器 eventual memory consistency 的执行，在多处理器系统上可能无法序列化！

当 $x \neq y$ 时，对 x, y 的内存读写可以交换顺序

- 它们甚至可以在同一个周期里完成 (只要 load/store unit 支持)
- 如果写 x 发生 cache miss，可以让读 y 先执行
 - 满足“尽可能执行 μop ”的原则，最大化处理器性能

```
# <-----+
movl $1, (x) # |
movl (y), %eax # --+
```

- 在多处理器上的表现
 - 两个处理器分别看到 $y = 0$ 和 $x = 0$

示例: store-load

store-load.c

thread.h

代码示例：线程间的内存可见性

CFLAGS

为了提高共享内存系统的性能，系统中并非只有一个“全局共享内存”。每个处理器都有自己的缓存，并且通过硬件实现的协议维护一致性。在 x86 多处理器系统中，允许 store 时暂时写入处理器本地的 store buffer，从而延迟对其他处理器的可见性。

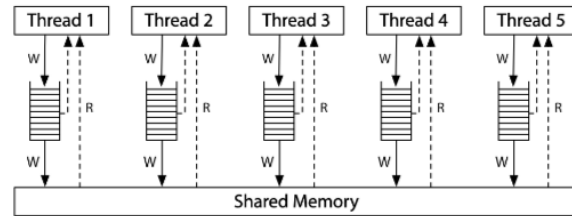
所以，把 x86 的程序移到 arm 上，会出阴间的事情。

宽松内存模型 (Relaxed/Weak Memory Model)

宽松内存模型的目的是使单处理器的执行更高效。

x86 已经是市面上能买到的“最强”的内存模型了 😊

- 这也是 Intel 自己给自己加的包袱
- 看看 [ARM/RISC-V](#) 吧，根本就是个分布式系统



(x86-TSO in [Hardware memory models](#) by Russ Cox)

总结

在一个简化的模型中，多线程/多进程程序就是“状态机的集合”，每一步选一个状态机执行一步。然而，真实的系统可能带来一些复杂性：

- 指令/代码执行原子性假设不再成立
- 程序的顺序执行假设不再成立
- 多处理器间内存访问无法即时可见

人类本质上是物理世界 (宏观时间) 中的“sequential creature”：时间是我们潜移默化的概念。我们发明了顺序的程序设计语言：分支、循环……这意味着我们程序员不要和这三件事情硬刚。“放弃”的意思是，我们总是要回到原点：simple seq exec。所以，我们的意思就是，不要试图以为你自己对编译器/内存模型理解得很好，老老实实用lock/unlock。

当我们掌握了并发控制技术，我们就回到了简单的simple seq exec，即使我们确实是在多处理器上跑的代码。