Notes on Computer Systems

LaTeX by Junfan Zhu

2024 年 9 月 29 日

目录

1 链接器 Linker

n 个源文件 code.c 就有 n 个目标文件 code.o，由链接器 Linker 程序把 n 个目标文件合并打包成可执行程序。就像把各章内容汇总合并成一本书，并确保章节间的互相引用成立。

1.1 符号决议 Symbol Resolution

让每个目标文件的外部符号都能在符号表里找到唯一定义。

1. 编译器 Compiler 把符号表放在目标文件中，符号表记录了一个源文件定义了那些符号（给其他模块使用）、用到哪些外部符号（给自己用）。
2. undefined reference to func：无编译错误，但装订书时发现 func 这一章没写，故本书不完整。

1.2 静态库 Static Library

把一堆源文件单独编译，在生成可执行文件时只要编译自己的代码，在链接过程中把静态库拼装进来，加快编译速度。

1.3 动态库 Dynamic Library/Shared Library

不用像静态库一样把（大量重复的）内容复制进可执行文件，节约磁盘空间。

1. 可执行文件加载时的动态链接。系统的加载器在加载可执行文件时检测它是否依赖动态库，若是则启动动态链接器。
2. 程序运行（程序被 CPU 执行）时的动态链接。

✓ 优点：方便维护、扩展、多语言混合编程和代码复用。如果修改动态库，只需重新编译动态库，不需编译依赖它的程序。

✗ 缺点：性能不如静态链接。如果动态库没有或与可执行文件不兼容，则程序无法启动。

1.4 重定位：确定符号运行时的地址

编译器不知道函数的内存地址（地址是 0），需要链接器去找到内存地址并修正指令（重定位）。链接器完成符号决议后，确定没有链接错误，就可以合并目标文件，然后所有指令的内存地址就确认了。就像书装订成型后就可以确定“参考第几页”。

1.5 虚拟内存：链接器怎么知道变量运行时的内存地址

CPU 执行程序 A 时，从 0x4000 内存地址获取的指令属于程序 A；执行程序 B 时，从从 0x4000 内存地址获取的指令属于程序 B。为什么同一个内存地址，两次获取的数据不同?

虚拟内存是假的、物理上不存在的内存，它让每个程序产生幻觉，每个程序在运行时都认为自己独占内存，而不管真实物理内存。因此，程序还没运行时，链接器就知道进程的内存布局，尽管内存地址是假的，链接器不关心程序运行后的物理内存。

每个进程都有自己的页表（虚拟内存到物理内存的映射），虚拟内存让我们像读写内存一样方便地操作文件（mmap 机制）。

2 进程线程

CPU 根据 Program Counter (PC) 寄存器，从内存读指令。寄存器就像内存，存放 CPU 要执行的指令所在内存的地址。

2.1 进程 Process

可以随意暂停和恢复进程的运行，让 CPU 随时切换各个进程。

2.2 线程 Thread

如果进程只有一个 main 函数，进程中的指令只能被一个 CPU 执行，怎样让多个 CPU 执行同一个进程中的指令呢？让进程有多个入口函数，于是多个 CPU 在一个共享进程的地址空间内，同时多个执行属于同一个进程的指令。

1. 是快速创建的轻量级进程，因为线程共享所属进程地址空间。
2. 消耗进程内存空间，因为每个线程都有自己的栈。

✓ 处理长时间任务好。一个请求创建一个线程 thread-per-request。

✗ 处理大量短任务不好，因为消耗时间、内存和线程间切换。

2.2.1 线程池

来一批订单，招一批工人，干完活就辞退工人，下次再招人，就很麻烦。不如有订单就处理工作，没工作就摸鱼。线程池的思想就是复用，而不用频繁创建销毁。生产者-消费者，队列数据结构。

2.2.2 线程资源

1. 堆区：线程间共享资源，指针知道变量地址。动态分配内存 malloc/new，全局变量。
2. 栈区：是线程私有的，无状态函数 Stateless 因为只用私有资源不用全局资源，因此是安全的。但虚拟内存确保了不同进程的地址空间相互隔离，因此线程 A 可以修改线程 B 栈区中的变量。
3. 线程安全 Thread Safe：区分线程的私有资源和共享资源。如果传入指针指向并修改全局变量，就需要加锁保护线程安全。

2.2.3 协程

1. 可暂停、恢复的函数，用 yield 来暂停，用 next() 调用该协程从上一个 yield 暂停点继续运行。
2. 和普通函数的 return 不同，return 后面的代码就不会被执行了，函数就是没有挂起点的协程。
3. 高性能、高并发领域。以同步的方式进行异步编程。

2.3 回调函数

只有我们才知道做什么，但我们不知道什么时候去做，只有其他模块知道，因此必须把我们知道的封装成回调函数告诉其他模块。

1. 调用函数不仅可以传变量，也可以传代码，将函数作为参数传递。从而不用根据定制化需求不断改代码。

```
handle(customer1);
something_important(); // 异步，执行它的时候，handle 可能还没开始

// f 是回调函数
void make(func f) {
    f();
}
void handle(func f) {
    // 调用 make 函数时，创建一个新线程，然后返回并立刻运行 something_important()
    // 线程启动后才真正开始执行 make
    thread t(make, f);
}
```

2. 函数内部可以创建线程，当线程启动后才执行重要功能，将调用方和被调用方在各自线程中并行运行，可以减少调用的等待时间。

   (a) 同步回调 Synchronous Callbacks/阻塞回调 Blocking Callbacks：整个任务在函数调用方线程中处理完成。

   (b) 异步回调 Asynchronous Callbacks/延迟回调 Deferred Callbacks：主程序和回调函数的执行位于不同线程或进程中。任务分成两部分：调用之前的部分在调用方线程中处理；第二部分只有调用方（其他进程、线程、机器）才知道做什么，不在我们掌控范围。

   (c) 异步回调比同步回调更充分利用多核资源，因为同步回调时主程序会偷懒一段时间，而异步回调时主程序一直运行，适用于 Web 服务等高并发场景。

例如，我调用第三方库中的函数，把回调函数传递给第三方库（因为第三方库不知道在什么情况执行什么操作，只有使用者我知道），第三方库中的函数在特定的节点（如文件传输完成时，事件驱动的 event handler）来调用我的回调函数（我的回调函数并不是由我调用的）。

2.3.1 同步异步

1. 同步就是 AB 相互依赖的紧耦合（如打电话，或者开会然后搬砖然后开会）。
2. 异步就是 AB 相互独立，各干各的（如邮件，或者一边开会一边搬砖，两件事同时进行效率高）。

```
void handle_B_after_A_query() {
    B;
}
// 回调函数：数据库线程查询后，调用 handle_B_after_A，回调函数做了什么，数据库不管
```

```
// 为什么不是数据库线程自己调用？因为这不是数据库线程的工作，它只要查询然后调用回调函数
// 高效，主线程处理用户请求、数据库处理查询同时进行，充分利用系统资源，系统响应迅速
A_query(request, handle_B_after_A_query);
```

3. 异步调用一般是 I/O 等高耗时任务，如网络数据收发、数据库、文件读写，从而调用方不会被阻塞，立即执行接下来的程序。

4. 同步调用不一定是阻塞的，但阻塞调用一定是同步的。非阻塞也不一定是异步的。

2.3.2 高并发、高性能

1. 多进程 Process-per-connection. 父进程 fork 出多个子进程，用子进程处理用户请求。充分利用多核资源，各进程隔离不易一起崩溃；创建进程有开销，进程间需要通信机制。

2. 多线程 Thread-per-connection. 线程就像螃蟹，房子（地址空间）都是进程的，自己只是一户租客。线程间通信不需通信机制，直接读取内存即可。但一个线程崩溃会导致整个进程崩溃，而且由于线程安全，多个线程不能同时读写它们共享的数据资源，用同步互斥机制又会带来死锁问题。

3. 事件驱动 Event-based concurrency. 通过事件循环 event loop (while 和 for 都可以) 接收源源不断到来的事件并处理。事件循环不能调用阻塞式接口，这会熄火。

```
// 创建 epoll，I/O 多路复用（事件循环的发动机）
epoll_fd = epoll_create();
//告诉 epoll 看管好一堆 socket 描述符
epoll_ctl(epoll_fd, fd1, fd2, ...);

while(1) {
    int n = epoll_wait(epoll_fd); // 相当于 getEvent()
    for (i = 0; i < n; i ++ ) {
        // 处理事件
    }
}
```

4. Reactor 模式。餐馆分前台服务员和后台大厨，服务员快速接待顾客点餐（事件循环），并告诉后台大厨（工作线程）煮面、烧菜，做好后服务顾客。事件处理函数和事件循环不在同一个线程中，而是独立线程。

5. 各服务器通过远程过程调用 RPC 进行通信，RPC 封装网络建立连接、数据传输、数据解析。

2.3.3 协程：同步编程、异步执行

1. 把 handle 函数（同步）放在协程里运行，当发起 RPC 通信后调用，yield 释放 CPU。协程与线程阻塞调用的区别：协程挂起后不好阻塞工作线程。

2. （内核态）线程是内核创建调度的；（用户态）协程对内核来说不可见，无论多少协程，内核依然按照线程来分配 CPU 时间片，在线程分配到的时间片内，我可以决定运行哪些协程，这是 CPU 时间片在用户态的二次分配。

2.4 计算机系统小结：回调、闭包、容器、虚拟机

1. 函数：代码复用。
2. 变量：内存不仅可以存放代码，也可以存放数据。
3. 指针：多个变量指代同一段数据。
4. 回调函数 Callback：多个变量指代同一段代码。代码可以像变量一样赋值、当参数传递、像普通变量一样返回函数。
5. 闭包 Closure，回调函数与一部分数据绑定后统一作为一个变量对待。回调函数是代码在 A 处定义，在 B 处调用，我们希望回调函数可以绑定只能在 A（不能在调用回调函数的地方 B）获得的数据。

```
def add():
    b = 1
    # add_inner 是闭包。依赖两个数据，一个在 add 函数中（运行时环境），一个在用户传入的参数
    def add_inner(x):
        return b + x
    return add_inner
# 调用 f，集齐 add_inner 所依赖的所有数据
f = add()
print(f(1))
```

6. 协程：函数让 CPU 暂停运行，并下次调用该函数时从暂停处继续。
7. 线程：函数的暂停与继续是在内核态。
8. 进程：线程 + 依赖的运行时资源（如地址空间）。
9. 容器 Container：程序 + 程序依赖的运行时环境（配置、库），是集装箱，是对操作系统的虚拟化。
10. 虚拟机：操作系统认为自己独占硬件资源（就像容器中的进程认为自己独占操作系统）。

3 内存

内存是个储物柜，每个柜子是 memory cell，只能存放 0 和 1。8 个储物柜 =1 字节，它们有一个编号就是内存地址。指针是内存地址的抽象，指针是个变量，这个变量保存的是地址。

3.1 虚拟内存

1. 虚拟内存是对内存的抽象，内存地址可以不是真实的物理内存地址。
2. 比如每个进程的代码区都从 0x4000 开始，如果两个进程调用 malloc 分配内存，可能返回同样的起始地址 0x7f64，而这个虚拟地址 0x7f64 是假的，在传送给内存之前会被修正为真实物理内存地址，通过页表来映射虚拟内存与物理内存。
3. 每个进程都有自己的页表。页 page 把进程地址空间分成大小相等的块，所以即使两个进程指向同一个内存地址写数据也没关系，因为内存地址所在的页数存放在不同的物理内存地址上。

3.2 栈区：函数调用

1. 函数调用栈 call stack 实现函数调用、跳转和返回。
2. 寄存器（CPU 内部资源）实现参数传递和返回，寄存器写入局部变量之前，要先把寄存器中的原始值保存在函数栈帧中。
3. 不能创建太大局部变量，函数调用层次不能过多，否则导致栈溢出。

3.3 堆区：内存分配

1. 动态内存分配与释放能将数据保存在我管理的内存区域，从而让数据跨越多个函数。就像在停车场分配停车位一样，用 malloc/new 在堆区申请内存，不用时 free/delete 释放内存。
2. 遍历每个内存块的信息头 header 的最后一个比特位，就知道它空不空。遍历信息尾 footer，释放内存时可以快速合并相邻空闲内存块。header+footer 将内存块组成隐式双向链表。

3.3.1 CPU 内核态、用户态

1. CPU 在内核态无所不能，而在用户态没有特权。系统调用 System Call 是两界传送门，可以让操作系统替我完成读写、通信等任务。进程就像客户端，操作系统就像服务器端，系统调用就像网络请求。
2. 标准库（如 malloc）可以屏蔽 Linux 和 Windows 的底层差异，我调用标准库进行读写、通信，标准库根据具体操作系统选择对应的系统调用。有了系统调用，如果堆区内存不足，可以通过 brk 系统调用（内核态，操作系统的一部分），向操作系统申请扩大堆区。
3. 虚拟内存才是终极 boss。进程看到 malloc 申请到的内存是假的内存、空头支票，真正分配到物理内存是在使用内存时，此时产生缺页错误 page fault，因为虚拟内存没有关联到物理内存，操作系统捕捉到错误时开始分配真正物理内存，通过修改页表来建立虚拟内存映射真实内存。
4. malloc 是内存的二次分配，分配的是虚拟内存，发生在用户态；程序使用虚拟内存时映射到真实的物理内存，这时操作系统真正分配物理内存，发生在内核态。

3.3.2 malloc 内存分配全过程

就像我的工作态度，老板分配我任务，我没空并先给他一个虚拟空头支票，我说好的，但我压根不急，等实在不行需要交差了，这时候我出现了一个缺页中断，我再真正抽空开始弄。

1. malloc 搜索空闲内存块，如果找不到，就调用 brk 系统调用扩大堆区。
2. 调用 brk 后转入内核态，操作系统用虚拟内存扩大进程堆区，但并没有分配真正物理内存。
3. brk 结束后返回 malloc，CPU 从内核态切换到用户态，malloc 找到空闲内存块并返回。
4. 内存获得，程序继续。
5. 代码读写新内存时，系统出现缺页中断，CPU 再次从用户态切换到内核态，操作系统分配物理内存，在页表建立虚拟内存与物理内存的映射，CPU 再从内核态切换回用户态，程序继续。

3.4 内存经典 bug 类型

1. 返回指向局部变量的指针
2. 指针运算。移动指针不需考虑数据类型大小，int 的指针 ++ 是移动 4 字节，1024 字节的结构体的指针 ++ 是移动 1024 字节。
3. 有问题的指针

```
int a;
scanf("%d", a);
```

如果 a 的值被解释成指针后指向

(a) 代码区或不可写区：操作系统会 kill 该进程，程序中止问题不大。

(b) 栈区：其他函数的栈帧就被破坏，程序行为脱离掌控，bug 无法定位。

(c) 堆区或数据区：程序动态分配的内存被破坏，程序行为脱离掌控，bug 无法定位。

4. 读取未被初始化的内存。记得手动清空，malloc 返回的内存未必初始化为 0。
5. 引用已被释放的内存。如果这个内存已被分配出去，则指针指向的内存已被覆盖，解引用得到了被覆盖的数据。
6. 数组下标从 0 开始。数组越界可能破坏 malloc 工作状态。
7. 栈溢出。比堆溢出更严重，因为栈帧保存函数返回地址。黑客会让溢出部分覆盖栈帧返回地址，修改为保存黑客恶意代码的特定地址。
8. 内存泄露。程序不断申请内存却不释放，导致堆区太大被操作系统杀掉（Linux 的 OOM 机制 Out of Memory Killer）

4 CPU

4.1 CPU 与进程线程

1. 空闲进程与 CPU 低功耗。为了让设计没有异常，我们让队列永不为空，调度器总能从队列找到空闲进程运行，链表存在“哨兵”节点也可以避免判空。CPU 执行 halt 指令时，系统中已经没有可运行的就绪进程了，halt 是特权指令，只在内核态 CPU 才执行（用户态不行）。空闲进程被定时器中断后，中断处理函数会判断是否有就绪进程，若没有就继续运行空闲进程。
2. 分支预测。CPU 执行 if 跳转指令还没做完时，后面的指令就要进入流水线，但 CPU 不知道哪些指令进入流水线，就靠猜，这就是分支预测。猜对了则流水线继续，猜错了则流水线已执行错误分支指令作废，产生性能损耗。因此高性能代码最好把 if 语句写得让 CPU 猜对，比如用 likely/unlikely 宏告诉编译器哪些分支有可能为真，编译器就可性能优化。
3. CPU 与线程。CPU 核心数 = 厨师数，线程数 = 上菜数，二者没有直接关系。如果线程只是计算，不 I/O、同步互斥，则每个核心一个线程。如果线程需要 I/O、同步互斥，则增加线程数确保操作系统有足够线程分配给 CPU，但也不能太多，因为线程间切换开销增加。
4. 精简指令集 RISC 思想。Relegate Interesting Stuff to Compiler，发挥流水线优势，用更长但更简单的指令高效执行、提高 CPU 吞吐量，编译器对 CPU 控制力比复杂指令集更强。

5. 超线程 Hyper-threading/硬件线程 Hardware Threads. 具有超线程的 CPU 核心让操作系统产生幻觉，以为一个 CPU 核心存在多个 CPU 核心，因为指令间依赖关系让流水线并不满载运行，因此可以让指令流见缝插针填满流水线。

4.2 CPU 机制：函数调用、系统调用、线程切换、中断处理

1. 寄存器。CPU 要寄存器，因为寄存器访问内存速度快。寄存器就是内存，信息的临时存放站。
2. 栈寄存器。栈顶信息保存在栈寄存器 Stack Pointer，能跟踪函数的调用栈。
3. 指令地址寄存器。指令地址寄存器 Program Counter (PC) 能记录“正在执行哪一条指令”。
4. 状态寄存器 Status Register。保存两种状态（内核态、用户态），能记录 CPU 正切换于哪种状态。
5. 上下文 Context。CPU 执行顺序被打断时，用栈的嵌套结构处理以下情况
   (a) 函数调用：CPU 从函数 A 跳转到 B
   (b) 系统调用：CPU 从用户态切换到内核态。操作系统完成系统调用所需的“运行时栈”，在内核态栈 Kernel Mode Stack 中，每个用户态线程在内核态都有一个对应的内核态栈。CPU 状态切换后进入内核态，找到对应的内核态栈，此时用户态线程的运行上下文都在这里，完成系统调用后返回用户态。
   (c) 中断处理：CPU 被打断而去处理中断。若中断处理函数没有自己的运行时栈，则依赖内核态栈完成中断处理。如有中断处理函数栈 Interrupt Service Routine (ISR)，和系统调用的切换内核态并返回的过程类似。
   (d) 线程切换：CPU 从机器指令 A 切换到执行机器指令 B。先切换地址空间，再把线程 A 切换到线程 B，保存 A 的上下文，恢复 B 的上下文。就像 CPU 实施换颅手术，把 A 的记忆封存在 A 结构体，换上 B 的记忆，此时 B 的记忆刚刚处理完定时器中断，但线程 B 是在其时间片用尽后被暂停执行的，但线程 B 对自己被暂停这件事一无所知，线程 B 只记得接下来要切换回用户态，好像什么也没发生。

5 缓存

cache 本无必要，但因为 CPU 与内存速度差异巨大，所以工程角度有意义。CPU 是永远吃不饱的，内存是慢吞吞的厨师，永远喂不饱 CPU，所以有 cache 保存近期内存数据，访问速度和 CPU 一样快。

5.1 cache 一致性与性能杀手

1. cache 更新。如果 cache 值被更新了，内存中还是旧的，就是 inconsistent。解决方法：
   (a) write-through. 更新 cache 同时更新内存，但更新 cache 就不得不访问内存，CPU 要等待内存更新完毕，这是同步。
   (b) write-back. CPU 写内存时直接更新 cache，需要把容量不足的 cache 中剔除的数据更新到内存中（如果被修改过），这样更新 cache 与更新内存就解耦了，这是异步。

2. 多核 cache 一致性。CPU 的两个核心的 cache 中有两个副本，两个核心间需要用 MESI 协议来同时修改 cache 中的值。

3. 性能杀手

   (a) cache 乒乓问题。多线程为什么比单线程慢？因为 cache 一致性导致两个 CPU 核心从内存读值、修改，又必须将另一个 cache 中的值置为无效，反复乒乒乓乓拖累了性能，使得从内存中读值消耗了多线程的性能。

   (b) 伪共享问题 False Sharing。多线程且无共享变量为什么比单线程慢？因为两个变量尽管不共享数据，但可能共享同一个 cache line，cache 和内存以 cache line 伪单位交互，当变量 a 未命中 cache 时，会把变量所在 cache line 一并加载到 cache 中，所以变量 b 也可能被加载进来。改进方法是在两个变量之间填充其他变量，只要其他变量大于 cache line，a 和 b 就不会共享同一个 cache line。

5.2 内存屏障 memory barrier：周幽王烽火戏诸侯

线程间同步问题。周幽王 = 线程，诸侯 = 线程，烽火 = 两个线程间同步信号。内存屏障是机器指令，让 CPU 按顺序执行，确保某核心在其他核心看来言行一致。

```
bool is_enemy_coming = False
int enemy_num = 0

// 周幽王线程
void thread_zhouyouwang() {
    enemy_num = 10000;
    is_enemy_coming = true;
}

// 诸侯线程
void thread_zhuhou() {
    int n;
    if (is_enemy_coming)
        n = enemy_num;
}
```

1. CPU 并不严格按顺序执行机器指令，因为

   (a) 编译器在把我的代码转换到 CPU 机器指令时会动手脚（指令重排序）

   (b) CPU 执行指令时也会动手脚（指令乱序执行 Out of Order Execution, OoOE），因为 CPU 与内存速度差异巨大，如果必须顺序执行，流水线会出现空隙 slots，OoOE 是为了充分利用流水线。

2. CPU 必须停止等待维护 cache 一致性，为此，系统增加一个 store buffer 队列，让写操作记录在队列而非立即更新到 cache 中，然后 CPU 继续执行后续指令而不用等待。

   对于 CPU 执行指令来说，这是异步，因为 CPU 不等写操作真正更新到 cache 才执行后续指令，为了更好性能，效果就像“抢跑式提前执行”：先执行第二行再执行第一行。但这种“言行不一致”的乱序只在除自身外的其他核心观察该核心时才出现。如果单线程，就不用操心。

3. 无锁编程 lock-free programming：可以在不用锁保护的情况下，在多线程操作共享资源，原理是利用原子操作，如 CAS (Compare And Swap)。周幽王与诸侯无锁，只有无锁编程者才需要关心指令重排序。

   (a) 互斥锁：锁占用线程后，其他请求该锁的线程被挂起等待，直到占用该锁的线程释放。

   (b) 回旋锁：锁占用线程后，其他请求该锁的线程反复来检测锁是否释放，请求锁的线程不会被操作系统挂起。

   (c) 无锁编程：用原子操作检测到共享资源被使用时，去处理其他事情。用于处理资源竞争、ABA 问题。

四种内存屏障。

1. LoadLoad. 诸侯“抢跑”读取 *enemy_num*，此时为 0，但读取时敌人已经来了，所以要 LoadLoad 屏障，确保 *is_enemy_coming* 为真时读取最新的（而非旧的）*enemy_num*。

```
// 诸侯线程
void thread_zhuhou() {
    int n;
    int important;
    if (is_enemy_coming) {
        LoadLoad_FENCE(); // LoadLoad 内存屏障
        n = enemy_num;
        important = 1; // 假如某个写操作必须看到烽火后才执行
    }
}
```

2. StoreStore 阻止 CPU 在执行 store 时“抢跑”执行后面的 store。周幽王线程要两次设置 *is_enemy_coming* 和 *enemy_num*，周幽王 CPU 可能“抢跑”设置 *is_enemy_coming*，虽然诸侯检测到烽火但敌人没来，为防止周幽王戏诸侯，在两次设置变量中加入 StoreStore 内存屏障防止先设置烽火信号。由于系统的 cache，是异步的，但 StoreStore 内存屏障可以保证核心看到的变量更新顺序与代码顺序一致，故读取的肯定是最新值。StoreStore 不保证更新立即对其他核心可见，只保证更新顺序与代码顺序一致。

3. LoadStore. 假如某个写操作 $important = 1$ 必须看到烽火后才执行，那么 LoadLoad 还不够，需要 LoadStore 内存屏障保证 CPU 不抢跑执行。

4. StoreLoad. 同步的，最重的内存屏障，阻止 CPU 在写的时候“抢跑”执行读，只要其他核心在 StoreLoad 之后再读屏障之前的变量，一定是最新值。

5.3 acquire-release 语义：周幽王无法戏诸侯

不需要 StoreLoad 这种很重的内存屏障，只需要其他三种内存屏障。

1. acquire = LoadLoad + LoadStore，Load 之后所有内存操作不能放到 Load 之前执行，确保诸侯线程内存读写不会在检测到烽火信号前。

2. release = StoreStore + LoadStore，Store 之前所有内存操作不能放到 Store 之后执行，确保周幽王线程内存读写不会在设置烽火信号后。

```
// 原子变量
std::atomic<bool> is_enemy_coming(false);
int enemy_num = 0;

// 周幽王线程
void thread_zhouyouwang() {
    enemy_num = 10000;
    // release 屏障
    // 原子变量的读写 std::memory_order_release 确保原子性即可，不需指令重排序
    std::atomic_thread_fence(std::memory_order_release);
    is_enemy_coming.store(true, std::memory_order_relaxed);
}

// 诸侯线程
void thread_zhuhou() {
    int n;
    if (is_enmy_coming.load(std::memory_order_relaxed)) {
        // acquire 屏障
        std::atomic_thread_fence(std::memory_order_acquire);
        n = enemy_num;
    }
}
```

6 I/O

6.1 高并发的秘诀：I/O 多路复用三剑客

I/O 多路复用 multiplexing: 餐馆排队要叫号，使用文件要文件描述符 File Descriptions，进行文件操作时就把文件描述符告诉内核。如果服务器要处理成千上万客户端请求，则不能用 read 这种阻塞式 I/O，而多线程也不能应付高并发场景因为开销太大。我们用“别打电话给我，有必要我会打电话通知你”的策略。

调用函数告诉内核：“这个函数先别返回，帮我盯着文件描述符，有可以进行读写操作的文件描述符时你再返回。”

I/O 多路复用三剑客

1. select. 把进程线程放到等待队列，此时进程线程因 select 而阻塞运行，当出现可读可写事件时，唤醒该进程线程。由于要从头到尾检查文件描述符是否就绪，所以低效。

2. poll. 类似 select

3. epoll. 在内核创建数据结构，重要的字段是一个就绪文件描述符列表。任何被监听文件描述符有事了，就唤醒进程并加载就绪文件到列表中，这样进程唤醒就能直接获取就绪文件描述符，而不用从头到尾遍历文件描述符。高效、高并发，适用网络的框架。

6.2 mmap：像读写内存一样操作文件

1. 虚拟内存让进程以为自己独占内存，那么机器指令中携带的是虚拟地址，但在虚拟地址到达内存前会被转为物理内存地址。于是我们把文件映射到了进程地址空间，做到读写内存就是直接操作磁盘文件。

2. 开销 tradeoff：read/write 需要系统调用，读写把数据从内核态和用户态相互拷贝，有开销。mmap 则不需要拷贝开销，但需要维护地址空间与文件的映射关系的开销、缺页 page fault 中断开销。

3. mmap 优势是处理大文件（> 物理内存的文件）。

4. 很多进程依赖同一个动态链接库，可以用 mmap 把它映射到各个依赖此库的进程地址空间，物理内存仅有一份的库就被各进程都认为自己地址空间拥有该库。

Disclaimer: 笔记内容来源于 [1] ，感谢作者的这本好书，受益匪浅。

参考文献

[1] 陆小风. 计算机底层的秘密. 电子工业出版社, 2023.