

Chapter 12 并发编程

- 我们有如下三种方式进行**并发编程**:

- 基于**进程**:

- **进程** 有独立的虚拟地址空间, 因此如果想和其它 **逻辑控制流** (进程)通信, 就必须使用某种显式的 **进程间通信(IPC)** 机制.

- 基于**I/O多路复用**:

- 应用程序在一个 **进程** 的上下文中, 显式地调度多个自己的 **逻辑流** .
- **逻辑流**被模型化为 **状态机** , 当数据到达 **文件描述符** 后, 主程序显式地从一个 **状态** 切换到另一个 **状态** .
- 由于是在同一个 **进程** 下, 所以所有的流共享同一个地址空间.

- 基于**线程**:

- 和 **I/O多路复用** 一样, 运行在 **单一进程** 的上下文中, 从而共享同一个 **虚拟地址空间** .
- 和 **进程** 一样, 由 **内核** 进行调度.
- 可以理解为是上述两种的"混合体".

- 基于**进程的 echo服务器** 编写思路:

- **父进程** 用于接受客户端的连接请求;
- 当接受到新的客户端时, **父进程** 创建 **子进程** 来为每个客户端提供服务.
- 注意, 由于fork()的子进程会获得父进程描述符的完整副本, 所以子进程需要关闭 **监听描述符** , 而父进程需要关闭 **已连接描述符** . 因为二者分别不需要相应的描述符.

应该强调, 如果父进程不关闭相应的 **已连接描述符** , 其后果是致命的.

回顾一下, 父子进程的 **已连接描述符** 都指向同一个 **文件表表项** , 如果父进程不 `close()` 这个 `connfd` , 那么即使子进程已经完成服务并关闭了这个描述符, 由于父进程没有 `close()` 这个 `connfd` , 所以文件表表项中的引用计数永远不会归0. 那么内核就永远不会终止到客户端的连接.

- 注意要(使用SIGCHLD信号处理程序)回收 **僵死子进程** . 因为服务器会长时间运行, 如果不回收这些资源会让系统崩溃.

- 基于**进程**并发编程的优缺点:

- **优点**: 由于父子进程不共享地址空间, 所以消除了很多令人迷惑的错误.
- **缺点**: 也是由于这种地址空间的私有, 两个进程之间为了共享信息, 必须使用IPC(进程间通信)机制, 它们的开销往往很高!

Unix IPC 机制:

- `wairpid` 和 **信号** 是最基本的IPC机制, 他们允许 **进程** 发送一些小消息到 **同一主机** 上的 **其它进程** .
- **套接字接口** 也是一种IPC机制. 不过我们通常所说的IPC, 主要是指在 **同一主机** 上的 **进程间** 通信.

- Unix IPC 包括: 管道, FIFO, 系统V共享内存, 系统V信号量(semaphore)等等. 这些均超出本书范围.

- 基于I/O多路复用(I/O Multiplexing)的 echo服务器 编写思路:

考虑对我们的 echo服务器 进行升级, 要求不仅要响应 客户端连接请求 , 还要能响应本地的 命令行输入 (从stdin).

这种情况下, 我们的程序有两个东西要进行 等待 : 1. 来自网络客户端的连接请求; 2. 来自键盘的输入.

事实上, 我们根本无法对任何一个进行等待! (比如如果等1, 那么2就无法被响应. 反之亦然)

我们引入基于 select函数 的 I/O多路复用 技术.

- select函数 : 要求内核挂起进程, 当指定的(一个或多个)I/O事件发生后, 再将控制返还给应用程序.

select() 有很多使用场景. 我们只讨论其中一种: 等待, 直到一组描述符中的至少一个 准备好被用于读 了.

```
1 int select(int n, fd_set* fdset, NULL, NULL, NULL);
```

- fdset 的类型为 fd_set , 叫做 描述符集合 . 是一个大小为 n 的位向量. 第 k 位对应描述符 k . 使用FD宏家族来修改和检查.
- select 函数会一直阻塞, 直到读集合中的某个描述符 准备好可以读 了.
- 所谓 准备好可以读 , 当且仅当 从该描述符 读取一个字节 的请求不会阻塞. (注意如果遇到EOF, 当然也算是能读取一个字节)
- select 会在返回时修改 fdset 指向的描述符集合, 用于传递返回值 准备好了的集合 . 因此每次调用 select 前应该重置 fdset .
- 具体到这个 echo服务器 , 我们要做的就是调用 select 函数, 参数 fdset 中包括 stdin 的描述符和 listenfd 的描述符.

- 状态机: 不严谨地说, 就是一组状态(State), 输入事件(Input Event)和转移(Transition).

- 每个 转移 是将一对 (输入状态, 输入事件) 映射到一个 输出状态 .
- 一个状态机从某个 初始状态 开始运行, 而每个 输入事件 都会引发一个 状态转移 .
- 基于 I/O多路复用 的并发 事件驱动 服务器.

- 模型构建: 我们把每个客户端 k 都抽象为一个状态机 s_k .

- 每个 s_k 有一个 状态 : 等待描述符 d_k 准备好可以读.
- 每个 s_k 有一个 输入事件 : 描述符 d_k 准备好可以读了.
- 每个 s_k 有一个 转移 : 从描述符 d_k 读一个文本行.

换言之, 就是借助 select 函数检测 输入事件 的发生,

当某个已连接描述符 准备好可读 时, 服务器就为相应的状态机执行 转移 ,

即从这个 已连接描述符 读一个文本行, 并写回一个文本行.

- 具体实现: 使用一个 pool 池结构, 维护全体 活动客户端 的集合.

服务器在一个无限循环中:

1. 调用 `select` 函数来 检测 (共两种可能) 输入事件 :
 - 一个新客户端的连接请求到达, 对应于 `listenfd` "可读"
 - 一个已存在的客户端的已连接描 `connfd` 述符 准备好可以读 了.
 2. 调用 `add_clients` 函数 创建 新的 状态机 .
 3. 调用 `check_clients` 函数执行状态转移: 回送输入行.
- 基于 I/O多路复用 的 事件驱动 服务器的优缺点:
 - 优点:
 - 运行在 同一进程 中, 逻辑流之间方便数据共享.
 - 运行在单个进程中, 可以用GDB等调试工具方便地调试
 - 不需要进行开销巨大的上下文切换, 运行效率更高
 - 缺点:
 - 程序写起来比较复杂, 如果 并发粒度 减小, 复杂性会进一步上升.
 - 所谓 并发粒度 即每个逻辑流在每个时间片中执行的指令数量.
比如在我们的例子中, 并发粒度 就是读一个完整的文本行所需要的指令数量.
 - 此外, 如果某个逻辑流正在忙于读某个文本行, 其他的逻辑流就不可能有进展!
比如想象一个 故意只发送一部分文本行然后就停止 的恶意攻击客户端, 我们的服务器就直接卡死了.
-

- 线程(Thread): 运行在 同一进程 的上下文中的逻辑流, 由 内核 自动调度.
 - 每个 线程 都有它自己的**线程上下文(Thread Context)**,
包括: 唯一的整数**线程ID(Thread ID, TID)**, 栈 , 栈指针 , 程序计数器 , 通用寄存器 , 条件码寄存器 .
 - 内核 通过 TID 唯一地识别 线程 .
 - 同一进程 中的所有 线程 共享该进程的整个 虚拟地址空间 .
 - 线程模型 的特点:
 - 每个进程最开始只有一个线程, 称为**主线程(Main Thread)**.
 - 在某一时刻, 主线程可以创建一个**对等线程(Peer Thread)**, 这样两个线程就会并发地运行:
例如: 当主线程执行了一个慢速系统调用时, 内核就会通过上下文切换, 切换到对等线程. 过了一段时间, 控制又会传递回主线程.
 - 线程的上下文切换要比进程快得多, 因为线程上下文的内容少得多.
 - (不同于进程是按照父子层次组织的), 和一个线程相关的线程组成**对等线程池**,
一个线程可以 杀死 它的任何对等线程, 也可以 等待 它的任何对等线程终止.
-

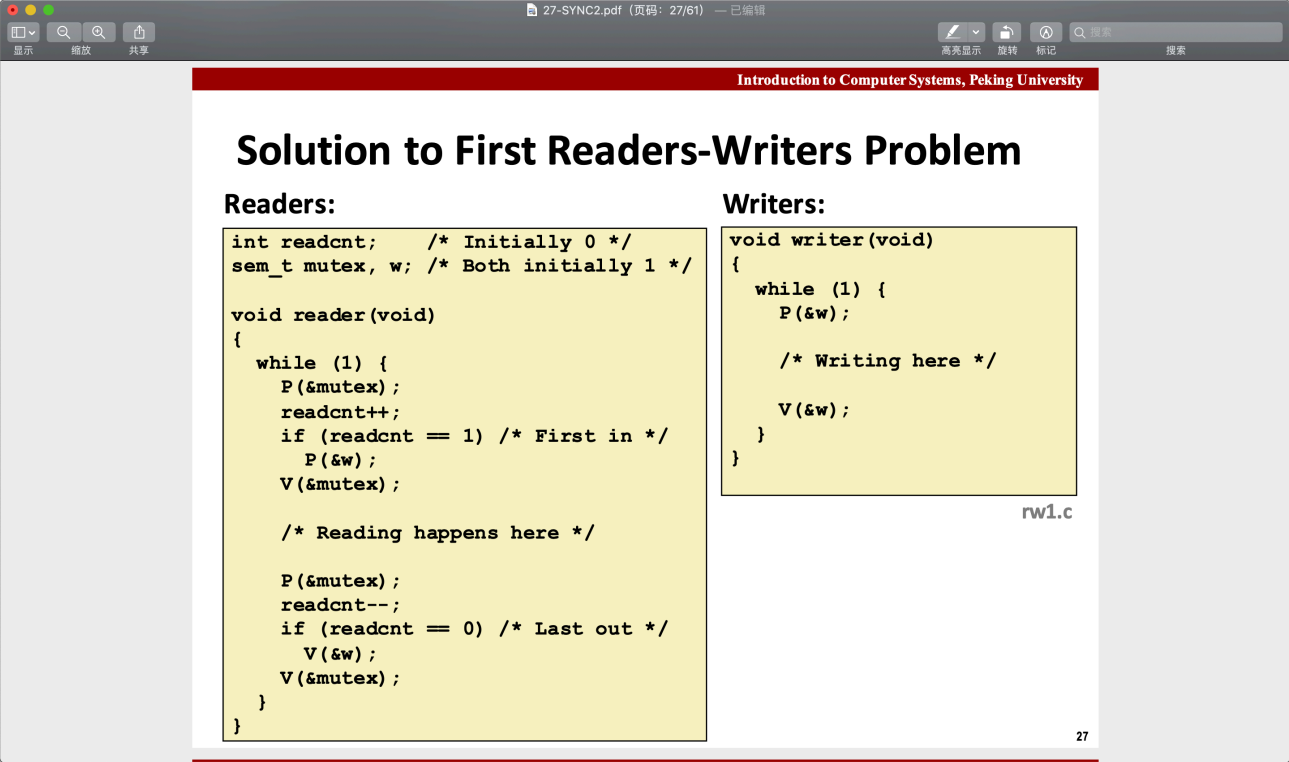
后面的笔记就摸掉了, 因为期末这两周在写别的论文, 整理这个笔记还是蛮耗时的.

下面是某节课的课堂笔记.

mutex: mutual exclusion

第一类读写者问题 和 第二类读写者问题

首先是信号量的命名方式: `mutex` 是用来保护 `cnt` 变量的访问的; `r` 和 `w` 是用来控制reader和writer的进入的.



The screenshot shows a presentation slide with a red header bar containing the text "Introduction to Computer Systems, Peking University". The slide title is "Solution to First Readers-Writers Problem". It is divided into two columns: "Readers:" and "Writers:". The "Readers:" column contains C code for a reader function that uses a semaphore `mutex` and a counter `readcnt`. The "Writers:" column contains C code for a writer function that uses a semaphore `w`. The code is highlighted in yellow. The slide number "27" is in the bottom right corner.

Readers:

```
int readcnt; /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void)
{
    while (1) {
        P(&mutex);
        readcnt++;
        if (readcnt == 1) /* First in */
            P(&w);
        V(&mutex);

        /* Reading happens here */

        P(&mutex);
        readcnt--;
        if (readcnt == 0) /* Last out */
            V(&w);
        V(&mutex);
    }
}
```

Writers:

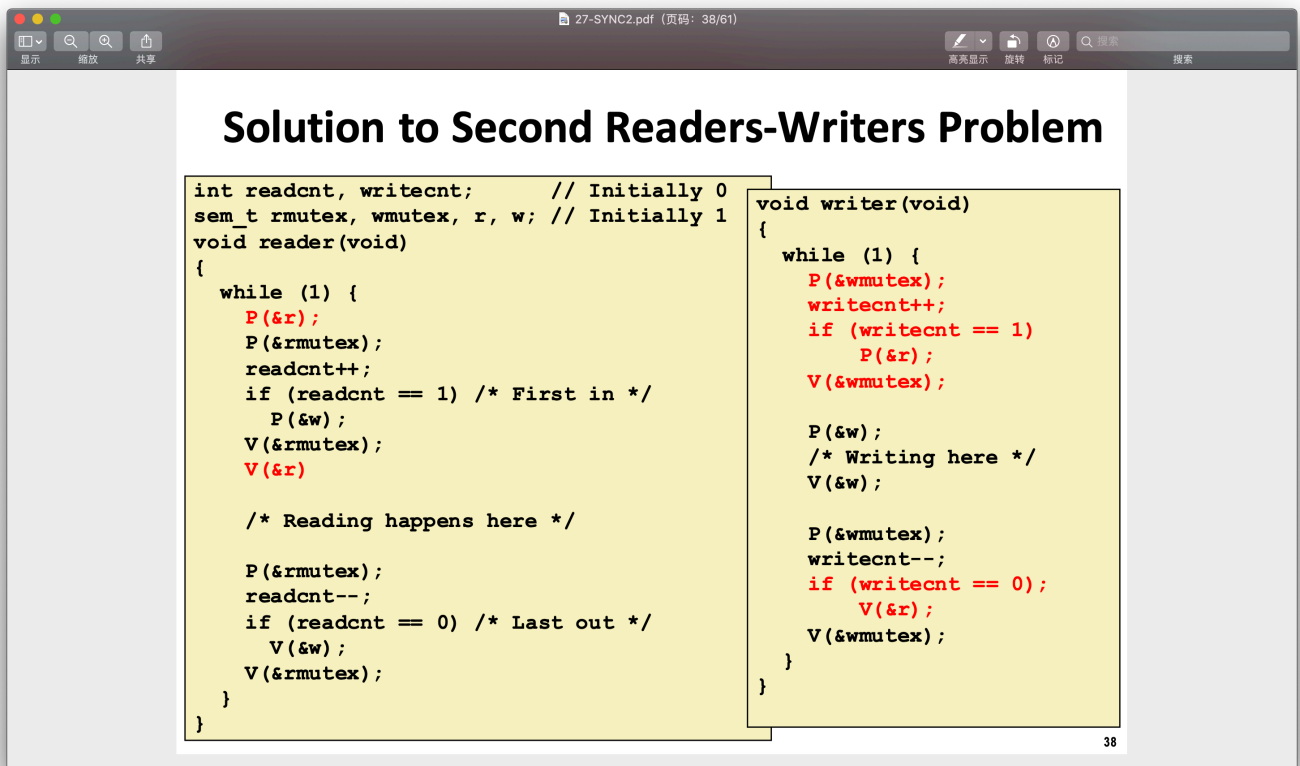
```
void writer(void)
{
    while (1) {
        P(&w);

        /* Writing here */

        V(&w);
    }
}
```

rw1.c

27



cpy: 第二类我好像get到了

就是首先mutex信号量只是为了保护balabala_cnt的访问, 所以大可以暂时忽略(只要记住mutex必须紧贴地包围cnt前后即可~)

然后相比于第一类, 第二类只是多了一个 `&r` 信号量.

在第一类里, writer是非常简单的, 啥权力都没有; 与此同时, reader的"尝试进入"也是不受限制的.

而在第二类里, reader的"尝试进入"要受到 `&r` 的限制; 与此同时, writer也获得了"锁住 `&r` "的权力, 换句话说就是writer可以通过P(&r)来阻止reader进入, 从而达到"插队"的目的.

?疑问:writer难道不还是随机地获得&r?

并不会, 因为读者前7行那个P(r)V(r)是非常快的, 这些代码几乎不耗时.