搜狗实验室技术交流文档

Vol.1:1 C10K 问题

摘要

编写连接数巨大的高负载服务器程序时，经典的多线程模式和 select 模式都不再适用。应当抛弃它们，采用 epoll/kqueue/dev_poll 来捕获 I/O 事件。最后简要介绍了 AIO。

由来

网络服务在处理数以万计的客户端连接时，往往出现效率低下甚至完全瘫痪，这被称为 C10K 问题。随着互联网的迅速发展，越来越多的网络服务开始面临 C10K 问题，作为大型网站的开发人员有必要对 C10K 问题有一定的了解。本文的主要参考文献是 http://www.kegel.com/c10k.htmls。

C10K 问题的最大特点是：设计不够良好的程序，其性能和连接数及机器性能的关系往往是非线性的。举个例子：如果没有考虑过 C10K 问题，一个经典的基于 select 的程序能在旧服务器上很好处理 1000 并发的吞吐量，它在 2 倍性能新服务器上往往处理不了并发 2000 的吞吐量。

这是因为在策略不当时，大量操作的消耗和当前连接数 n 成线性相关。会导致单个任务的资源消耗和当前连接数的关系会是 O(n)。而服务程序需要同时对数以万计的 socket 进行 I/O 处理，积累下来的资源消耗会相当可观，这显然会导致系统吞吐量不能和机器性能匹配。为解决这个问题，必须改变对连接提供服务的策略。

基本策略

主要有两方面的策略：1.应用软件以何种方式和操作系统合作，获取 I/O 事件并调度多个 socket 上的 I/O 操作；2. 应用软件以何种方式处理任务和线程/进程的关系。前者主要有阻塞 I/O、非阻塞 I/O、异步 I/O 这 3 种方案，后者主要有每任务 1 进程、每任务 1 线程、单线程、多任务共享线程池以及一些更复杂的变种方案。常用的经典策略如下：

1. Serve one client with each thread/process, and use blocking I/O
   这是小程序和 java 常用的策略，对于交互式的长连接应用也是常见的选择(比如 BBS)。这种策略很能难足高性能程序的需求，好处是实现极其简单，容易嵌入复杂的交互逻辑。Apache、ftpd 等都是这种工作模式。
2. Serve many clients with single thread, and use nonblocking I/O and readiness notification
   这是经典模型，datapipe 等程序都是如此实现的。优点在于实现较简单，方便

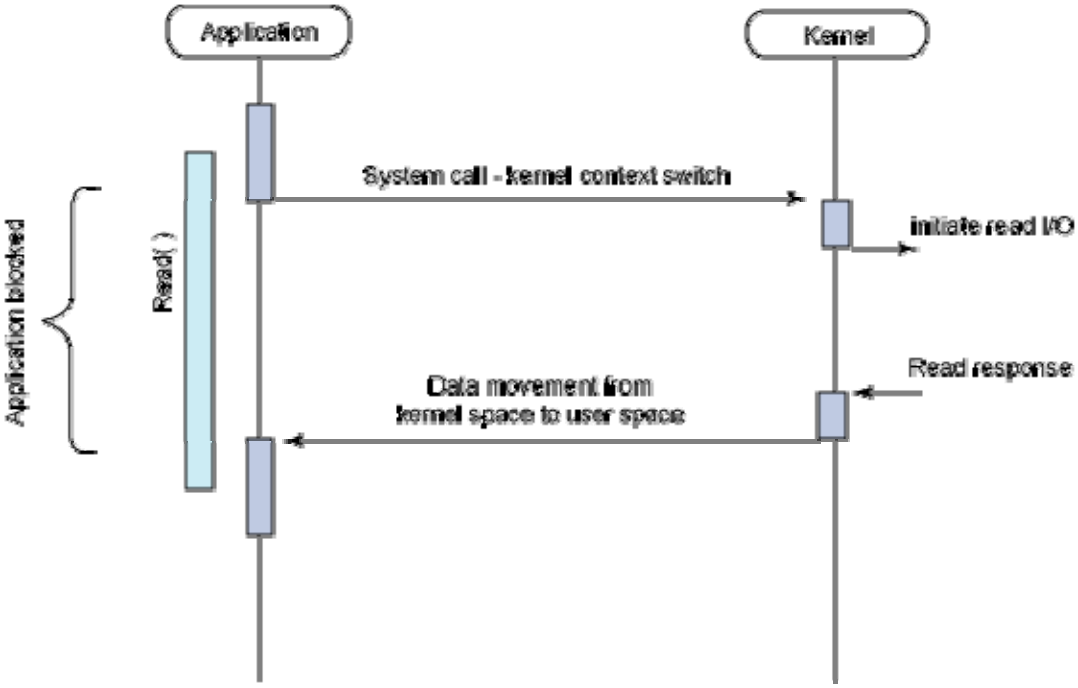
移植，也能提供足够的性能；缺点在于无法充分利用多 CPU 的机器。尤其是程序本身没有复杂的业务逻辑时。

3. Serve many clients with each thread, and use nonblocking I/O and readiness notification
   对经典模型 2 的简单改进，缺点是容易在多线程并发上出 bug，甚至某些 OS 不支持多线程操作 readiness notification。
4. Serve many clients with each thread, and use asynchronous I/O
   在有 AI/O 支持的 OS 上，能提供相当高的性能。不过 AI/O 编程模型和经典模型差别相当大，基本上很难写出一个框架同时支持 AI/O 和经典模型，降低了程序的可移植性。在 Windows 上，这基本上是唯一的可选方案。

本文主要讨论模型 2 的细节，也就是在模型 2 下应用软件如何处理 Socket I/O。

select 与 poll

最原始的同步阻塞 I/O 模型的典型流程如下：



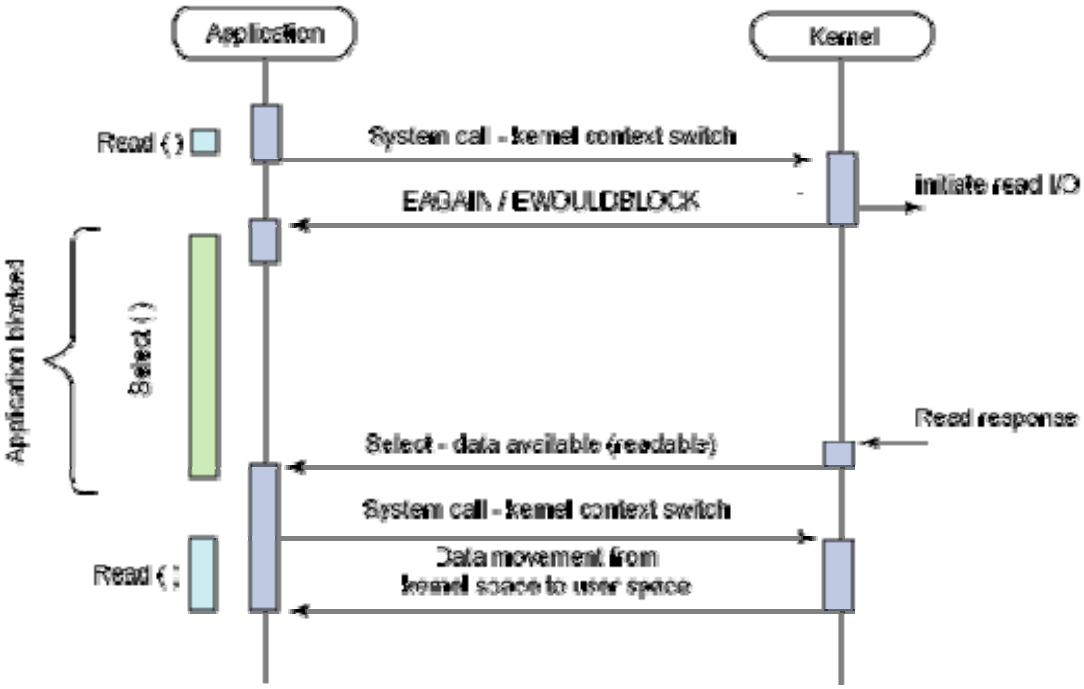
从应用程序的角度来说，read 调用会延续很长时间，应用程序需要相当多线程来解决并发访问问题。同步非阻塞 I/O 对此有所改进：

经典的单线程服务器程序结构往往如下：

```
do {
    Get Readiness Notification of all sockets
    Dispatch ready handles to corresponding handlers
        If (readable) {
            read the socket
            If (read done)
                Handler process the request
```

```
        }
        if (writable)
            write response
        if (nothing to do)
            close socket
} while(True)
```

非阻塞 I/O 模型的典型流程:



其中关键的部分是 readiness notification，找出哪一个 socket 上面发生了 I/O 事件。一般从教科书和例子程序中首先学到的是用 select 来实现。Select 定义如下：

```
int select(int n, fd_set *rd_fds, fd_set *wr_fds, fd_set *ex_fds, struct timeval *timeout);
```

Select 用到了 fd_set 结构，从 man page 里可以知道 fd_set 能容纳的句柄和 FD_SETSIZE 相关。实际上 fd_set 在*nix 下是一个 bit 标志数组，每个 bit 表示对应下标的 fd 是不是在 fd_set 中。fd_set 只能容纳编号小于 FD_SETSIZE 的那些句柄。

FD_SETSIZE 默认是 1024，如果向 fd_set 里放入过大的句柄，数组越界以后程序就会垮掉。系统默认限制了一个进程最大的句柄号不超过 1024，但是可以通过 ulimit -n 命令/setrlimit 函数来扩大这一限制。如果不幸一个程序在 FD_SETSIZE=1024 的环境下编译，运行时又遇到 ulimit –n > 1024 的，那就只有祈求上帝保佑不会垮掉了。

在 ACE 环境中，ACE_Select_Reactor 针对这一点特别作了保护措施，但是还是有 recv_n 这样的函数间接的使用了 select，这需要大家注意。

针对 fd_set 的问题，*nix 提供了 poll 函数作为 select 的一个替代品。Poll 的接口如下：

```
int poll(struct pollfd *ufds, unsigned int nfds, int timeout);
```

第 1 个参数 ufds 是用户提供的一个 pollfd 数组，数组大小由用户自行决定，因此避免了 FD_SETSIZE 带来的麻烦。Ufds 是 fd_set 的一个完全替代品，从 select 到 poll 的移植很方便。到此为止，至少我们面对 C10K，可以写出一个能 work 的程序了。

然而 Select 和 Poll 在连接数增加时，性能急剧下降。这有两方面的原因：首先操作系统面对每次的 select/poll 操作，都需要重新建立一个当前线程的关心事件列表，并把

线程挂在这个复杂的等待队列上，这是相当耗时的。其次，应用软件在 select/poll 返回后也需要对传入的句柄列表做一次扫描来 dispatch，这也是很耗时的。这两件事都是和并发数相关，而 I/O 事件的密度也和并发数相关，导致 CPU 占用率和并发数近似成 $O(n^2)$ 的关系。

epoll, kqueue, /dev/poll

因为以上的原因，*nix 的 hacker 们开发了 epoll, kqueue, /dev/poll 这 3 套利器来帮助大家，让我们跪拜三分钟来感谢这些大神。其中 epoll 是 linux 的方案，kqueue 是 freebsd 的方案，/dev/poll 是最古老的 Solaris 的方案，使用难度依次递增。

简单的说，这些 api 做了两件事：1.避免了每次调用 select/poll 时 kernel 分析参数建立事件等待结构的开销，kernel 维护一个长期的事件关注列表，应用程序通过句柄修改这个列表和捕获 I/O 事件。2.避免了 select/poll 返回后，应用程序扫描整个句柄表的开销，Kernel 直接返回具体的事件列表给应用程序。

在接触具体 api 之前，先了解一下边缘触发(edge trigger)和条件触发(level trigger)的概念。边缘触发是指每当状态变化时发生一个 io 事件，条件触发是只要满足条件就发生一个 io 事件。举个读 socket 的例子，假定经过长时间的沉默后，现在来了 100 个字节，这时无论边缘触发和条件触发都会产生一个 read ready notification 通知应用程序可读。应用程序读了 50 个字节，然后重新调用 api 等待 io 事件。这时条件触发的 api 会因为还有 50 个字节可读从而立即返回用户一个 read ready notification。而边缘触发的 api 会因为可读这个状态没有发生变化而陷入长期等待。

因此在使用边缘触发的 api 时，要注意每次都要读到 socket 返回 EWOULDBLOCK 为止，否则这个 socket 就算废了。而使用条件触发的 api 时，如果应用程序不需要写就不要关注 socket 可写的事件，否则就会无限次的立即返回一个 write ready notification。大家常用的 select 就是属于条件触发这一类，以前本人就犯过长期关注 socket 写事件从而 CPU 100%的毛病。

epoll 的相关调用如下：

```
int epoll_create(int size)
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event)
int epoll_wait(int epfd, struct epoll_event * events, int maxevents, int timeout)
```

epoll_create 创建 kernel 中的关注事件表，相当于创建 fd_set。

epoll_ctl 修改这个表，相当于 FD_SET 等操作

epoll_wait 等待 I/O 事件发生，相当于 select/poll 函数

epoll 完全是 select/poll 的升级版，支持的事件完全一致。并且 epoll 同时支持边缘触发和条件触发，一般来讲边缘触发的性能要好一些。这里有个简单的例子：

```
struct epoll_event ev, *events;
int kdpfd = epoll_create(100);
ev.events = EPOLLIN | EPOLLET;   // 注意这个 EPOLLET，指定了边缘触发
ev.data.fd =listener;
epoll_ctl(kdpfd, EPOLL_CTL_ADD, listener, &ev);
for(;;) {
    nfds = epoll_wait(kdpfd, events, maxevents, -1);
```

```
    for(n = 0; n < nfds; ++n) {
        if(events[n].data.fd == listener) {
            client = accept(listener, (struct sockaddr *) &local,
                                &addrlen);
            if(client < 0){
                perror("accept");
                continue;
            }
            setnonblocking(client);
            ev.events = EPOLLIN | EPOLLET;
            ev.data.fd = client;
            if (epoll_ctl(kdpfd, EPOLL_CTL_ADD, client, &ev) < 0) {
                fprintf(stderr, "epoll set insertion error: fd=%d0,
                        client);
                return -1;
            }
        }
        else
            do_use_fd(events[n].data.fd);
    }
}
```

简单介绍一下 kqueue 和/dev/poll

kqueue 是 freebsd 的宠儿，kqueue 实际上是一个功能相当丰富的 kernel 事件队列，它不仅仅是 select/poll 的升级，而且可以处理 signal、目录结构变化、进程等多种事件。Kqueue 是边缘触发的

/dev/poll 是 Solaris 的产物，是这一系列高性能 API 中最早出现的。Kernel 提供一个特殊的设备文件/dev/poll。应用程序打开这个文件得到操纵 fd_set 的句柄，通过写入 pollfd 来修改它，一个特殊 ioctl 调用用来替换 select。由于出现的年代比较早，所以/dev/poll 的接口现在看上去比较笨拙可笑。

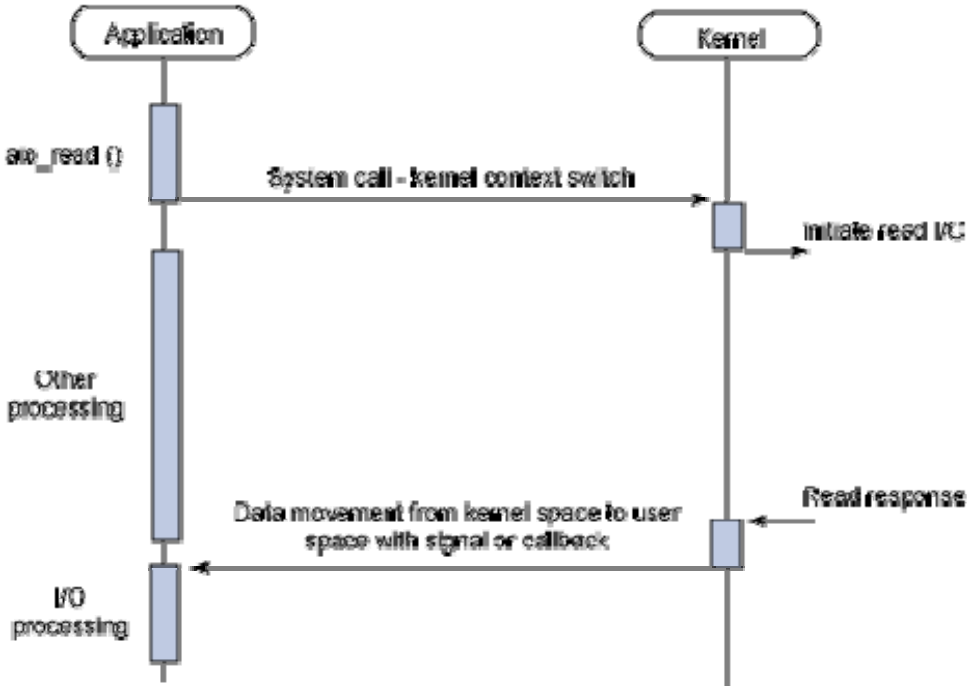
C++开发： ACE 5.5 以上版本提供了 ACE_Dev_Poll_Reactor 封装了 epoll 和/dev/poll 两种 api，需要分别在 config.h 中定义 ACE_HAS_EPOLL 和 ACE_HAS_DEV_POLL 来启用。

Java 开发： JDK 1.6 的 Selector 提供了对 epoll 的支持，JDK1.4 提供了对/dev/poll 的支持。只要选择足够高的 JDK 版本就行了。

异步 I/O 以及 Windows

和经典模型不同，异步 I/O 提供了另一种思路。和传统的同步 I/O 不同，异步 I/O 允许进程发起很多 I/O 操作，而不用阻塞或等待任何操作完成。稍后或在接收到 I/O 操作完成的通知时，进程就可以检索 I/O 操作的结果。

异步非阻塞 I/O 模型是一种处理与 I/O 重叠进行的模型。读请求会立即返回，说明 read 请求已经成功发起了。在后台完成读操作时，应用程序然后会执行其他处理操作。当 read 的响应到达时，就会产生一个信号或执行一个基于线程的回调函数来完成这次 I/O 处理过程。异步 I/O 模型的典型流程：



对于文件操作而言，AIO 有一个附带的好处：应用程序将多个细碎的磁盘请求并发的提交给操作系统后，操作系统有机会对这些请求进行合并和重新排序，这对同步调用而言是不可能的——除非创建和请求数目同样多的线程。

Linux Kernel 2.6 提供了对 AIO 的有限支持——仅支持文件系统。libc 也许能通过来线程来模拟 socket 的 AIO，不过这对性能没意义。总的来说 Linux 的 aio 还不成熟

Windows 对 AIO 的支持很好，有 IOCP 队列和 IPCP 回调两种方式，甚至提供了用户级异步调用 APC 功能。Windows 下 AIO 是唯一可用的高性能方案，详情请参考 MSDN。

待续

Thread Pool 与 PipeLine
Sendfile 等技巧节省用户空间和系统空间的来回拷贝
Timer 技巧
alloca 与内存池
Lock 技巧