Linux socket 网络通信 I/O 模型深入分析

◆孔天欣（20188068）

（东北大学秦皇岛分校 计算机与通信工程学院　河北 066004）

摘要：本文介绍了 Linux 系统中 socket 网络通信的 I/O 模型分类。首先从传统的同步阻塞模型入手，介绍了它存在的优势和缺陷，再通过依次改进和优化的方式介绍 Linux 中的同步非阻塞 I/O 模型、多路复用 I/O 模型和信号驱动式 I/O 模型，分析了这些模型各自的优劣之处以及各自的应用。最后通过 epoll 函数部分源码的详细剖析以及其提供的接口功能阐述，揭示了 epoll 能够实现多路 I/O 复用模型的基本原理。

关键词：网络 I/O；socket 通信；同步 I/O 模型；阻塞 I/O 模型

0　引言

套接字（socket）最先起源于 Unix，在 Linux 系统中是跨进程通信的技术之一，相较于其他 IPC 技术而言，socket 既能做到同一主机跨进程通信，也可做到不同主机跨进程通信；就本质而言，socket 是对 TCP/IP 协议的封装，并提供了调用接口（API）供程序员方便使用 TCP/IP 协议栈。

区别于传统 I/O，网络 I/O 要求对 socket 进行读取和写入，在 Linux 系统中，socket 被抽象为流，因此网络 I/O 的本质可以看作是对流的操作[1]，以读取为例，当网络上一台主机的分组数据包通过层层路由发送到当前主机的网络适配器时，其中的数据会先从 DMA 拷贝到操作系统内核空间中，然后再从内核空间拷贝到用户空间，这其中又会产生阻塞和非阻塞、同步和异步 I/O 操作之间的区别。本文首先介绍 Linux 系统中目前使用的各类 I/O 模型，然后通过剖析支持多路 I/O 复用 epoll 函数的源代码，来指出不同模型之间的区别和优势。

1　内核空间和用户空间

在 Linux 系统中，如果程序运行在内核空间，对应的进程处于内核态；如果程序运行在用户空间，那么对应的进程则处于用户态[2]。

1.1　内核空间

内核空间表示运行在处理器最高级别下的代码或数据，它的地址空间由所有进程共享，但只有运行在内核态的进程才能访问，用户进程可以通过系统调用切换到内核态访问内核空间，进程运行在内核态时所产生的地址都属于内核空间。

1.2 用户空间

用户空间由每个进程独有，运行在用户态和内核态的进程都可以访问用户空间。用户态下的进程运行在用户地址空间中，被执行的代码要受到 CPU 严格检查，例如进程只能访问映射其地址空间的页表项中规定的在用户态下可访问页面的虚拟地址。Linux 用户地址空间从低位到高位的顺序可以分为：文本段、初始化数据段、未初始化数据段、堆、栈和环境变量区。

2　Linux 传统同步阻塞式网络 I/O 模型

在 Linux 系统中，传统的读写操作是通过系统调用 read()和 write()完成的[3]，以 socket 编程的读操作为例，当用户态下的程序调用 recv()后，Linux 系统发生的一系列操作如下如图 2.1 所示。

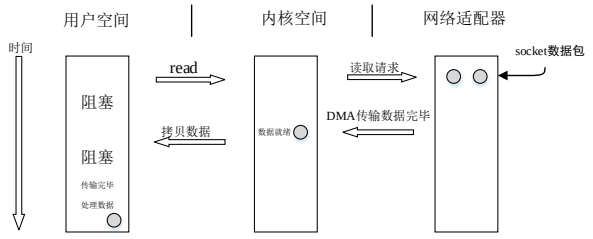

图 2.1 同步阻塞 I/O 模型

图示操作解释如下。

(1) 用户态进程发起系统调用，操作系统将上下文从用户态切换到内核态。

(2) CPU 向网络适配器的 DMA 控制器发出读取请求，DMA 控制器将网络适配器中的分组数据包拷贝到内核空间中。此后 CPU 继续执行其他任务。

(3) DMA 控制器传输数据到内核空间完毕后，向 CPU 发出中断，CPU 再次切换上下文，将内核空间的数据拷贝到该用户进程的用户空间当中。

(4) 在步骤（2）（3）期间，执行 recv()函数的用户进程一直处于阻塞状态，直到数据拷贝到用户空间为止，才能继续执行后续的代码。

可以看到，在这些操作中，除 DMA 拷贝是必需的以外，还至少需要上下文需要切换 2 次，CPU 数据拷贝 1 次，这些操作浪费了 CPU 资源，也增加了时间开销，在低延时高性能的网络环境中，如果并发量增大，对整个程序的性能产生较大的影响。

3　改进后的 Linux 同步非阻塞式网络 I/O 复用模型

3.1 同步非阻塞 I/O 模型

不同于传统 I/O 模型，同步非阻塞的 recv()操作优化成如图 3.1 所示步骤。

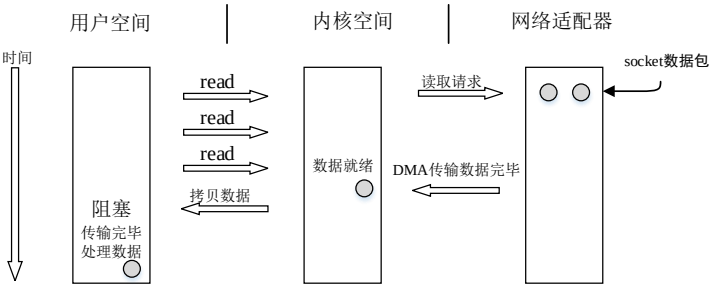

图 3.1 同步非阻塞 I/O 模型

图示操作解释如下。

(1) 用户态进程发起系统调用，操作系统切换上下文，从用户态切换到内核态。

(2) 操作系统向网络适配器的 DMA 控制器发出读取请求，DMA 控制器将网络适配器中的分组数据包拷贝到内核空间中。此后操作系统向用户进程返回一个“未就绪”标识。

(3) 用户进程收到“未就绪”标识后，继续执行后续代码，而不是阻塞于当前指令，每隔一段时间再次进行系统调用询问操作系统数据是否就绪。

(4) 如果数据已经从内核空间拷贝到用户空间，那么操作系统就返回给用户进程“就绪”标识，然后用户进程执行 I/O 相关任务；否则用户进程继续询问操作系统，直到数据就绪为止。

这种操作可以使得执行 recv()调用的用户进程不必一直阻塞于当前操作，可显著提高代码执行效率。然而询问间隔时间的设计需要多加考量：倘若间隔过大，那么系统任务完成的整体延迟会增加；倘若间隔过小，过于频繁的询问造成过多的上下文切换，造成的开销反而浪费更多 CPU 时间。

3.2 多路复用 I/O 模型

复用技术是一种设计思想，在网络通信和硬件设计中存在频分复用、时分复用等，多路 I/O 复用就是通过协调多个 socket 通道共享同一线程，实现对多个 I/O 处理效率最大化。

Linux 系统使用了 select 和 poll 以及 epoll 这三个系统调用函数实现了多路 I/O 复用模型，这种模型相较于 3.1 所述，设计更加精妙，现以 select()和 recv()操作为例，假设有多台主机发起了 socket 通信至本主机，本主机的用户进程（单进程单线程）建立了多个 socket 连接通道，相关流程见图 3.2。

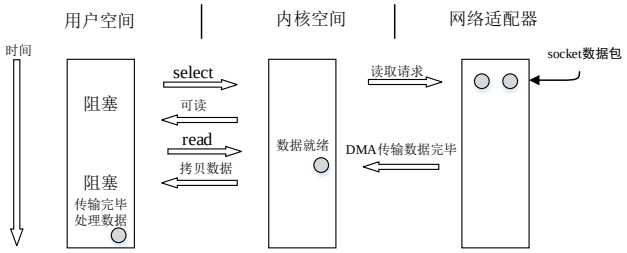

图 3.2 多路复用 I/O 模型

图示操作解释如下。

(1) 用户态进程调用 select()，发起系统调用，操作系统进行上下文切换到内核态。

(2) 操作系统监控所有用户态进程负责的多个 socket 通道对应的文件描述符，并通知 DMA 从网络适配器拷贝数据到内核空间中。

(3) 当其中一个 socket 的数据成功拷贝到了内核空间，该 socket 的文件描述符就绪，操作系统通知用户进程该 socket 的数据已就绪。

(4) 在步骤(3)未完成之前，该用户进程一直处于阻塞状态；完成后，select()会返回就绪信息，然后用户进程执行 recv()，再次发起系统调用，此后操作系统将内核空间的数据拷贝到用户空间当中去，然后用户进程开始处理数据。

这种模型相较于 3.2 中的模型而言，避免了用户进程高频询问内核空间数据是否准备就绪导致大量 CPU 上下文切换的时间开销，直接转由内核态监视所有 socket 通道情况并通知，在应对大量 socket 连接的情况会更有优势。

然而，这种模型又使得用户进程回归了调用时阻塞模式，降低了代码的整体执行效率。

3.3 信号驱动式 I/O

尽管本文 3.2 中指出的模型改善了许多问题，但用户进程调用时回归阻塞的行为显得不合时宜。借鉴计算机硬件中的 CPU 中断演进历史，DMA 在数据传输完毕后才发出中断通知 CPU 处理，否则 CPU 继续执行其他任务。在 socket 的网络 I/O 模型软件层面，也可以实现类似设计。信号驱动式 I/O 便能够使得用户进程不进入阻塞，见图 3.3。

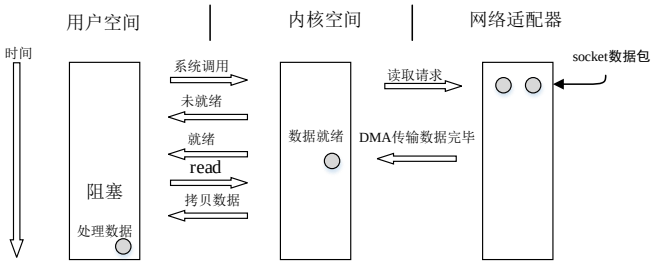

图 3.3 信号驱动式 I/O 模型

图示操作解释如下。

(1) 用户态进程首先编写信号处理程序，然后进行 sigaction 系统调用，切换上下文到内核态。

(2) 内核态开始监控一系列该进程负责的 socket 通道文件描述符集，然后立刻返回。

(3) 用户进程继续执行后续代码。

(4) 当有 socket 通道文件描述符就绪时，内核发出信号通知用户进程。

(5) 用户进程会在信号处理程序中调用 recv()，发起系统调用，操作系统将数据从内核空间拷贝到用户空间，并处理数据。

这种模型可以使得用户进程不必等待数据就绪，转而去处理其他任务，提高了代码整体的执行效率。同时也能应对大量 socket 通道的连接。

4 I/O 复用模型具体实现——以 epoll 源码为例

select、poll、epoll 等函数都可以实现 I/O 复用，其中 select 函数存在监控的 socket 文件描述符数量限制太小、同时查看 socket 集是否就绪采用 O(n)的遍历方式、以及消耗内存大等问题，已经被后继者 epoll 函数所超越，因此本文介绍 epoll 部分源码实现 I/O 多路复用的机制。

epoll 有三个供用户进程调用的接口，见代码 4.1。

代码 4.1 epoll 接口

```
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 的作用就是创建管理 size 个 socket 文件描述符的管理器 eventpoll。

epoll_ctl 的作用是对 eventpoll 文件描述符进行增删查改。它的参数 epfd 还是 epoll_create 的返回值，fd 是文件描述符，op 是对 fd 的操作类型，epoll_event 用于存放用户感兴趣的监听事件。

epoll_wait 的作用是阻塞等待内核返回的就绪 socket 事件，以供用户进程进行后续的数据处理，当就绪事件为空时它会被挂起，否则会被唤醒。

epoll 底层实现有两个关键的数据结构，分别名为 epitem 和 eventpoll，见代码 4.2 和代码 4.3。

代码 4.2 eventpoll

```
struct eventpoll {
    spin_lock_t          lock;
```

```
    struct mutex           mtx;
    wait_queue_head_t      wq;
    wait_queue_head_t      poll_wait;
    struct list_head       rdllist;
    struct rb_root         rbr;
    struct epitem          *ovflist;
}
```

eventpoll 是用来管理所有被监控的 socket 文件描述符集的对象，它通过调用 epoll_create 函数创建并返回给用户进程作为一个句柄。eventpoll 对象的属性 rbr 的类型是 rb_root，是红黑树的根节点，这个红黑树存储着一系列 epitem 对象，此外，eventpoll 还有个属性名为 rdllist，它是一个双链表的头结点，这个双链表连接了一系列就绪的 epitem 对象。ovfilst 是链表，用于暂存所有当前就绪的 epitem 对象，在未来它们会加入到 rdllist 当中。关于 epitem 的具体结构见代码 4.3。

代码 4.3　epitem

```
struct epitem {
    struct rb_node    rbn;
    struct list_head    rdllink;
    struct epitem    *next;
    struct epoll_filefd    ffd;
    int    nwait;
    struct list_head    pwqlist;
    struct eventpoll    *ep;
    struct list_head    fllink;
    struct epoll_event    event;
}
```

epitem 对象用于封装用户进程需要监听的 socket 文件描述符并扩展一些功能，ffd 就是被监听的文件描述符信息。在 epitem 中，属性 rb_node 表明它加入了红黑树的一个结点，rdllink 是事件就绪链表，pwqlist 是存放了被监视文件的等待队列。

用户进程通过调用 epoll_ctl 函数来创建 epitem 对象，创建完毕后，epitem 会被插入到 eventpoll 的红黑树中进行管理，然后它又会被插入到对应监控事件的目标文件等待列表，并且注册事件就绪之后的回调函数。回调函数会在事件就绪后被执行。

此外，epoll 内部存在 ep_send_events 函数，它会在被唤醒时扫描就绪的链表并通过链表结点 epitem 上的文件描述符将数据从内核空间拷贝到用户空间，然后返回给用户。此外，epoll 还设置了 ET 模式和 RT 模式，如果是 ET 模式下，那么 epitem 就会退出就绪队列直到文件描述符状态改变为止；如果是 RT 模式下，那么 epitem 会保留在就绪队列中。

5　结束语

本文比较了 Linux socket 通信中传统同步阻塞网络 I/O 模型和改进后的同步非阻塞 I/O 模型，并分析了各个模型的优缺点，最后将 epoll 的工作机制结合源码进行解释，可以看到，多路复用 I/O 在有大量 socket 连接时具备性能更加优异。

此外，根据近期资料[4]得知，Linux 5.1 已经出现异步非阻塞的网络 I/O 模型（AIO）工具 io_uring，甚至无需通知用户进程即可完成数据从内核空间到用户空间的拷贝。这可以更进一步提升 I/O 性能，或许能够在未来的网络 socket 编程中大发异彩。

参考文献：

[1] 常正超. 高并发访问量下网络 I/O 模型选择的研究[J]. 电脑知识与技术, 2016(7):28-29.

[2] RobertLove, 洛夫, 陈莉君,等. Linux 内核设计与实现[M]. 机械工业出版社, 2006(20-46).

[3] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff. UNIX 网络编程[M]. 人民邮电出版社, 2010(122-148).

[4] Jonathan Corbet.Ringing in a new asynchronous I/O API[EB/OL].https://lwn.net/Articles/776703/,2019