# Linux高性能服务器编程

## 基础API

分为三类：socket地址API，socket基础API，网络信息API。

**创建socket**：`int socket(int domain, int type, int protocol)`，`int sock = socket(PF_INET, SOCK_STREAM, 0)`;type参数指定服务类型， 主要有SOCK_STREAM服务（流服务）和SOCK_DGRAM（数据报）服务，前者表示使用TCP，后者UDP。

**命名socket**：创建socket时，我们给它指定了地址族，但是并未指定使用该地址族中的具体哪个socket地址。将一个socket于socket地址绑定称为为其命名，在服务器程序中，通常要命名socket，因为只有命名后才能让客户端知道该如何连接它，客户端通常不需要命名socket，而是采用匿名方式，使用操作系统自动分配的socket地址。`int bind(int sockfd, const struct sockaddr* my_addr, socklen_t addrlen);`

**监听socket**：`int listen(int sockfd, int backlog)`

**接受连接**：`int accept(int sockfd, struct sockaddr* addr, socklen_t *addrlen)`

**发起连接**：`int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen)`

**关闭连接**：`int close (int fd)`

## 高级IO函数

**1. pipe函数**：`int pipe(int fd[2])`，从fd[0]读数据，往fd[1]写数据。

**2. dup和dup2**：有时希望把标准输入重定向到一个文件，或者把输出重定向到一个网络连接，用这俩函数。

**3. sendfile函数**：该函数在两个文件描述符之间直接传递数据（直接在内核中操作），从而避免了内核缓冲区和用户缓冲区之间的数据拷贝，效率很高，这称为**零拷贝**。`ssize_t sendfile(int out_fd,int in_fd,off_t* offset,size_t count);` in_fd参数是待读出内容的文件描述符，out_fd参数是待写入内容的文件描述符。offset参数指定从读入文件流的哪个位置开始读，如果为空，则使用读入文件流默认的起始位置。count参数指定文件描述符in_fd和out_fd之间传输的字节数。**in_fd必须是一个支持类似mmap函数的文件描述符，即它必须指向真实的文件，不能是socket和管道，~~而out_fd必须是一个socket.~~** 我看了下sendfile的man page, 里面有这句话: In Linux kernels before 2.6.33, out_fd must refer to a socket. Since Linux 2.6.33 it can be any file. 我的linux内核版本是3.10, 验证了下, 确实也可以进行两个文件的拷贝

首先我们来看看传统的read/write方式进行socket的传输。
当需要对一个文件进行传输的时候，具体流程细节如下：

1：调用read函数，文件数据copy到内核缓冲区
2：read函数返回，文件数据从内核缓冲区copy到用户缓冲区
3：write函数调用，将文件数据从用户缓冲区copy到内核与socket相关的缓冲区
4：数据从socket缓冲区copy到相关协议引擎。
在这个过程中发生了四次copy操作。

硬盘->内核->用户->socket缓冲区（内核）->协议引擎。

而sendfile的工作原理呢？？

1、系统调用 sendfile() 通过 DMA 把硬盘数据拷贝到 kernel buffer，然后数据被 kernel 直接拷贝到另外一个与 socket 相关的 kernel buffer。**这里没有 用户态和核心态 之间的切换，在内核中直接完成了从一个 buffer 到另一个 buffer 的拷贝。**
2、DMA 把数据从 kernel buffer 直接拷贝给协议栈，没有切换，也不需要数据从用户态和核心态，因为数据就在 kernel 里。

**4. mmap和munmap函数**：mmap函数用于申请一段内存空间，我们可以将这段内存作为进程间通信的共享内存，也可以将文件直接映射到其中。munmap则会释放掉创建的这段内存空间。

# 多进程编程

## fork系统调用

fork函数复制当前进程，在内核进程表上创建一个新的进程表项。新的进程表项有很多属性与原进程相同，比如堆指针、栈指针和标志寄存器的值，会复制父进程的数据（堆数据、栈数据和静态数据）。数据的复制是采用写时复制（copy on write）即只有在任一进程(父进程或子进程)对数据执行了写操作时，复制才会发生(先是缺页中断，然后操作系统给子进程分配内存并复制父进程的数据)。即便如此，如果我们在程序中分配了大量内存，那么使用fork时也应当十分谨慎，尽量避免没必要的内存分配和数据复制。
此外，创建子进程后，父进程中打开的文件描述符默认在子进程中也是**打开**的，且文件描述符的引用计数加1。不仅如此，父进程的用户根目录、当前工作目录等变量的引用计数均会加1。

## exec系列系统调用

有时我们需要在于进程中执行其他程序，即替换当前进程映像，这就需要使用exec系列函数之一：

```
#include <unistd.h>
extern char** environ;

int execl( const char* path, const chart arg, ... );
int execlp( const char* file, const char* arg, ...);
int execle( const char* path, const char* arg, ..., char* const envp[] );
int execv( const char* path, char* const argv[] );
int execvp( const char* file, char* const argv[]);
int execve( const char* path, char* const argv[], char* const envp[] );
```

path参数指定可执行文件的完整路径，file 参数可以接受文件名，该文件的具体位置则在环境变量PATH中搜寻。arg 接受可变参数，argv 则接受参数数组，它们都会被传递给新程序(path或file 指定的程序)的main函数。envp 参数用于设置新程序的环境变量。如果未设置它，则新程序将使用由全局变量environ指定的环境变量。
　一般情况下，exec函数是不返回的，除非出错。它出错时返回-1,并设置ermo。如果没出错，则原程序中exec调用之后的代码都不会执行，因为此时原程序已经被exec的参数指定的程序完全替换(包括代码和数据)。exec函数不会关闭原程序打开的文件描述符，除非该文件描述符被设置了类似SOCK_CLOEXEC的属性(见5.2节)。

---

字母p表示该函数取filename作为参数，并且用PATH环境变量寻找可执行文件。字母l表示该函数取一个参数表，它与字母v互斥。v表示该函数取一个argv[]矢量。最后，字母e表示该函数取envp[]数组，而不使用当前环境。　——UNIX环境高级编程。

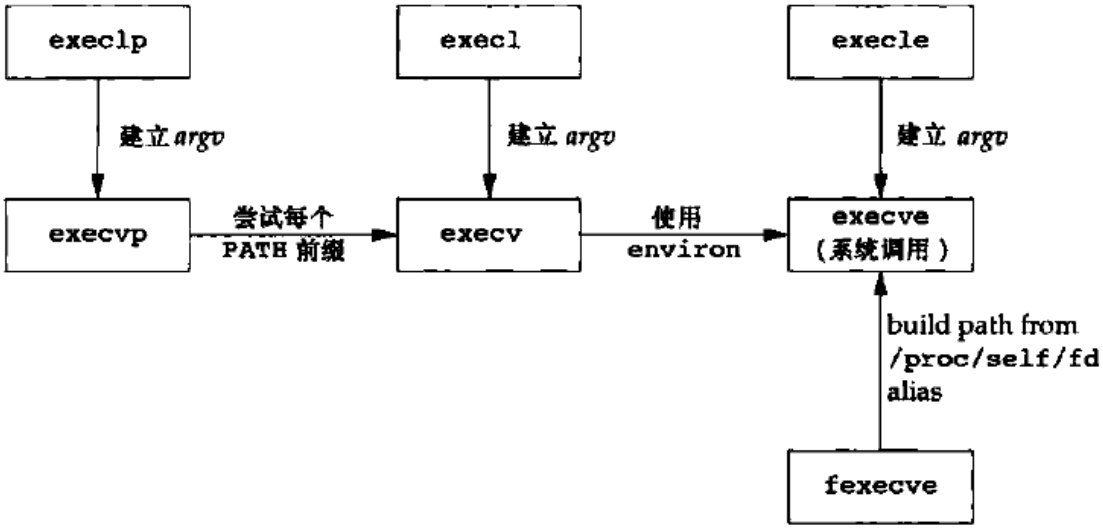


图 8-15　7 个 exec 函数之间的关系

## 处理僵尸进程

对于多进程程序而言，父进程一般需要跟踪子进程的退出状态。因此，当子进程结束运行时，**内核不会立即释放该进程的进程表表项，以满足父进程后续对该子进程退出信息的查询(如果父进程还在运行)。**在于进程结束运行之后，父进程读取其退出状态之前，我们称该子进程处于僵尸态。另外一种使子进程进人僵尸态的情况是：**父进程结束或者异常终止**，而子进程继续运行。此时子进程的PPID将被操作系统设置为1, 即init进程。init进程接管了该子进程，并等待它结束。在父进程退出之后，子进程退出之前，该子进程处于僵尸态。？

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int* stad_loc);
pid_t waitpid(pid_t pid, int* stat_loc, int options);
```

wait函数将阻塞进程，直到该进程的某个子进程结束运行为止。它返回结束运行的子进程的PID，并将该子进程的退出状态信息存储于stat_loc参数指向的内存中。
waitpid只等待由pid参数指定的子进程，如果pid取值为-1，那么它就和wait函数相同，即等待任意一个子进程结束。stat_loc参数含义与wait中的相同。waitpid调用将是非阻塞的，如果pid指定的目标子进程还没有结束或意外终止，则立即返回0；如果目标子进程确实正确退出了，那么会返回该子进程的PID。

# 事件处理模式

服务器通常需要处理三类事件：I/O事件、信号及定时器事件。

同步IO模型通常用于实现Reactor模式，异步IO则用于实现Proactor模式。

小林：https://mp.weixin.qq.com/s/GRkZ1IEfTalQSkErWe1SAg

## Reactor

它要求主线程（IO处理单元）只负责监听文件描述上是否有事件发生，有的话就立即将该事件通知工作线程（逻辑单元）。除此之外，主线程不做任何其他实质性的工作。读写数据，接受新的连接，以及处理客户请求均在工作线程中完成。

事实上，Reactor 模式也叫 `Dispatcher` 模式，我觉得这个名字更贴合该模式的含义，即 **I/O 多路复用监听事件，收到事件后，根据事件类型分配（Dispatch）给某个进程 / 线程。**

# Proactor

Proactor模式将所有IO操作都交给主线程和内核来处理，工作线程仅仅负责业务逻辑。 | Proactor 正是采用了异步 I/O 技术，所以被称为异步网络模型。

# 小结

- **Reactor 是非阻塞同步网络模式，感知的是就绪可读写事件。**在每次感知到有事件发生（比如可读就绪事件）后，就需要应用进程主动调用 read 方法来完成数据的读取，也就是要应用进程主动将 socket 接收缓存中的数据读到应用进程内存中，这个过程是同步的，读取完数据后应用进程才能处理数据。
- **Proactor 是异步网络模式， 感知的是已完成的读写事件。**在发起异步读写请求时，需要传入数据缓冲区的地址（用来存放结果数据）等信息，这样系统内核才可以自动帮我们把数据的读写工作完成，这里的读写工作全程由操作系统来做，并不需要像 Reactor 那样还需要应用进程主动发起 read/write 来读写数据，操作系统完成读写工作后，就会通知应用进程直接处理数据。

常见的 Reactor 实现方案有三种。

第一种方案**单 Reactor 单进程 / 线程**，不用考虑进程间通信以及数据同步的问题，因此实现起来比较简单，这种方案的缺陷在于无法充分利用多核 CPU，而且处理业务逻辑的时间不能太长，否则会延迟响应，所以不适用于计算机密集型的场景，适用于业务处理快速的场景，比如 **Redis** 采用的是单 Reactor 单进程的方案。

第二种方案单 **Reactor 多线程**，通过多线程的方式解决了方案一的缺陷，但它离高并发还差一点距离，差在只有一个 Reactor 对象来承担所有事件的监听和响应，而且只在主线程中运行，在面对瞬间高并发的场景时，容易成为性能的瓶颈的地方。

第三种方案多 **Reactor 多进程 / 线程**，通过多个 Reactor 来解决了方案二的缺陷，**主 Reactor** 只负责监听事件，响应事件的工作交给了**从 Reactor**，Netty 和 Memcache 都采用了「多 Reactor 多线程」的方案，Nginx 则采用了类似于 「多 Reactor 多进程」的方案。

Reactor 可以理解为「来了事件操作系统通知应用进程，让应用进程来处理」，而 Proactor 可以理解为「来了事件操作系统来处理，处理完再通知应用进程」。

因此，真正的大杀器还是 Proactor，它是采用异步 I/O 实现的异步网络模型，感知的是已完成的读写事件，而不需要像 Reactor 感知到事件后，还需要调用 read 来从内核中获取数据。

不过，无论是 Reactor，还是 Proactor，都是一种基于「事件分发」的网络编程模式，区别在于 Reactor 模式是基于「待完成」的 I/O 事件，而 Proactor 模式则是基于「已完成」的 I/O 事件。