课程目标
理解IO? 阻塞? epoll?
流? I/O操作? 阻塞?
流
可以进行I/O操作的内核对象
文件、管道、套接字……
流的入口：文件描述符(fd)
I/O操作
所有对流的读写操作，我们都可以称之为IO操作。
阻塞
阻塞等待 不占用CPU宝贵的时间片
非阻塞忙轮询 占用CPU，系统资源
在处理意见数据的接收场景时， 我们建议优先选择阻塞等待的方式， 不浪费性能资源
阻塞等待缺点
不能够很好的处理 多个(I/O)请求的问题
同一个阻塞，同一时刻只能处理一个流的阻塞监听
多路IO复用
既能够阻塞等待，不浪费资源
也能够同一时刻监听多个IO请求的状态
解决阻塞死等待的缺点(如何解决大量IO请求读写的问题)
方法一 阻塞等待+多进程/多线程 需要开辟线程浪费资源
方法二 非阻塞+忙轮询
while true {
for i in 流[] {
if i has 数据 {
读 或者 其他处理
}
}
}
CPU在大量的做判断，while和for
cpu的利用率不高
方法三、方法四 多路IO复用机制
select(与平台无关)
监听的IO数量有限，默认是1024个
不会精准的告诉开发者，哪些IO是可读可写的，需要遍历
while true {
select(流[]); //阻塞
//有消息抵达
for i in 流[] {
if i has 数据 {
读 或者 其他处理
}
}
}
epoll(Linux)
与select，poll一样，对I/O多路复用的技术
只关心“活跃”的链接，无需遍历全部描述符集合
能够处理大量的链接请求(系统可以打开的文件数目 可以通过 /proc/sys/fd/file-max查看)
while true {
可处理的流[] = epoll_wait(epoll_fd); //阻塞
//有消息抵达，全部放在“可处理的流[]”中
for i in 可处理的流[] {
读 或者 其他处理
}
}
<编程设计及通用理论>
第1讲:epoll的理论与IO复用及阻塞机制
作者:刘丹冰AceId
epoll 的API
(1) 创建EPOLL
/**
* @param size 告诉内核监听的数目
*
* @returns 返回一个epoll句柄（即一个文件描述符）
*/
int epoll_create(int size);
int epfd = epoll_create(1000);
Kernel
root
epfd
创建一个epoll句柄。实际上是在内核空间，建立一个root根节点，这个根节点的关系与epfd相对应。
(2) 控制EPOLL
/**
* @param epfd 用epoll_create所创建的epoll句柄
* @param op 表示对epoll监控描述符控制的动作
*
* EPOLL_CTL_ADD(注册新的fd到epfd)
* EPOLL_CTL_MOD(修改已经注册的fd的监听事件)
* EPOLL_CTL_DEL(epfd删除一个fd)
*
* @param fd 需要监听的文件描述符
* @param event 告诉内核需要监听的事件
*
* @returns 成功返回0，失败返回-1, errno查看错误信息
*/
int epoll_ctl(int epfd, int op, int fd,
struct epoll_event *event);
struct epoll_event {
__uint32_t events; /* epoll 事件 */
epoll_data_t data; /* 用户传递的数据 */
}
/*
* events : {EPOLLIN, EPOLLOUT, EPOLLPRI,
EPOLLHUP, EPOLLET, EPOLLONESHOT}
*/
typedef union epoll_data {
void *ptr;
int fd;
uint32_t u32;
uint64_t u64;
} epoll_data_t;
struct epoll_event new_event;
new_event.events = EPOLLIN | EPOLLOUT;
new_event.data.fd = 5;
epoll_ctl(epfd, EPOLL_CTL_ADD, 5, &new_event);
new_event
events:
EPOLLIN|EPOLLOUT
5
event_data:
fd=5
5
root
epfd
new_event
User
Kernel
(3) 等待EPOLL
/**
*
* @param epfd 用epoll_create所创建的epoll句柄
* @param event 从内核得到的事件集合
* @param maxevents 告知内核这个events有多大,
* 注意: 值 不能大于创建epoll_create()时的size.
* @param timeout 超时时间
* -1: 永久阻塞
* 0: 立即返回，非阻塞
* >0: 指定微秒
*
* @returns 成功: 有多少文件描述符就绪,时间到时返回0
* 失败: -1, errno 查看错误
*/
int epoll_wait(int epfd, struct epoll_event *event, int maxevents, int timeout);
struct epoll_event my_event[1000];
int event_cnt = epoll_wait(epfd, my_event, 1000, -1);
my_event
0
new_event
1
event1
2
EPOLLIN|EPOLLOUT
fd=5
root
epfd
new_event
event1
...
(4) 使用epoll编程主流程骨架
创建 epoll
int epfd = epoll_crete(1000);
将 listen_fd 添加进 epoll 中
epoll_ctl(epfd, EPOLL_CTL_ADD, listen_fd, &listen_event);
while (1) {
阻塞等待 epoll 中 的fd 触发
int active_cnt = epoll_wait(epfd, events, 1000, -1);
for (i = 0 ; i < active_cnt; i++) {
if (evnets[i].data.fd == listen_fd) {
accept. 并且将新accept 的fd 加进epoll中.
}
else if (events[i].events & EPOLLIN) {
对此fd 进行读操作
}
else if (events[i].events & EPOLLOUT) {
对此fd 进行写操作
}
}
}
触发模式
水平触发
水平触发的主要特点是，如果用户在监听epoll事件，当内核有事件的时候，会拷贝给用户态事件，但是如果用户只处理了一次，那么剩下没有处理的会在下一次epoll_wait再次返回该事件。
默认就是水平触发模式
边缘触发
边缘触发，相对跟水平触发相反，当内核有事件到达， 只会通知用户一次，至于用户处理还是不处理，以后将不会再通知。这样减少了拷贝过程，增加了性能，但是相对来说，如果用户马虎忘记处理，将会产生事件丢的情况。
在给事件进行绑定的时候，通过EPOLLET来设置