转 协程 及 Libco 介绍

2015年07月21日 22:13:48 kobejayandy 阅读数：6189

libco 是腾讯开源的一个协程库，主要应用于微信后台RPC框架，下面我们从为什么使用协程、如何实现协程、libco使用等方面了解协程和libco。

why协程

为什么使用协程，我们先从server框架的实现说起，对于client-server的架构，server最简单的实现：

while(1) {accept();recv();do();send();}

串行地接收连接、读取请求、处理、应答，该实现弊端显而易见，server同一时间只能为一个客户端服务。

为充分利用好多核cpu进行任务处理，我们有了多进程/多线程的server框架，这也是server最常用的实现方式：

accept进程 - n个epoll进程 - n个worker进程

1. accpet进程处理到来的连接，并将fd交给各个epoll进程
2. epoll进程对各fd设置监控事件，当事件触发时通过共享内存等方式，将请求传给各个worker进程
3. worker进程负责具体的业务逻辑处理并回包应答

以上框架以事件监听、进程池的方式，解决了多任务处理问题，但我们还可以对其作进一步的优化。

进程/线程是Linux内核最小的调度单位，一个进程在进行io操作时 (常见于分布式系统中RPC远程调用)，其所在的cpu也处于iowait状态。直到后端svr进程的时间片用完、进程被切换到就绪态。是否可以把原本用于iowait的cpu时间片利用起来，发生io操作时让cpu处理新的请求，以提高单核cpu的使

协程在用户态下完成切换，由程序员完成调度，结合对socket类/io操作类函数挂钩子、添加事件监听，为以上问题提供了解决方法。

用户态下上下文切换

Linux提供了接口用于用户态下保存进程上下文信息，这也是实现协程的基础：

- getcontext(ucontext_t *ucp): 获取当前进程/线程上下文信息，存储到ucp中
- makecontext(ucontext_t *ucp, void (*func)(), int argc, ...): 将func关联到上下文ucp
- setcontext(const ucontext_t *ucp): 将上下文设置为ucp
- swapcontext(ucontext_t *oucp, ucontext_t *ucp): 进行上下文切换，将当前上下文保存到oucp中，切换到ucp

以上函数与保存上下文的 ucontext_t 结构都在 ucontext.h 中定义，ucontext_t 结构中，我们主要关心两个字段：

- struct ucontext *uc_link: 协程后继上下文
- stack_t uc_stack: 保存协程数据的栈空间

stack_t 结构用于保存协程数据，该空间需要事先分配，我们主要关注该结构中的以下两个字段：

- void __user *ss_sp: 栈头指针
- size_t ss_size: 栈大小

获取进程上下文并切换的方法，总结有以下几步：

1. 调用 getcontext()，获取当前上下文
2. 预分配栈空间，设置 xxx.uc_stack.ss_sp 和 xxx.uc_stack.ss_size 的值
3. 设置后继上下文环境，即设置 xxx.uc_link 的值

4. 调用 makecontext()，变更上下文环境
5. 调用 swapcontext()，完成跳转

Socket族函数/io异步处理

当进程使用socket族函数 (connect/send/recv等)、io函数 (read/write等)，我们使用协程切换任务前，需对相应的fd设置监听事件，以便io完成后原行。

对io函数，我们可以事先设置钩子，在真正调用接口前，对相应fd设置事件监听。同样，Linux为我们设置钩子提供了接口，以read()函数为例：

1. 编写名字为 read() 的函数，该函数先对fd调用epoll函数设置事件监听
2. read() 中使用dlsym()，调用真正的 read()
3. 将编写好的文件打包，编译成库文件：gcc -shared -ldl -fPIC prog2.c -o libprog2.so
4. 执行程序时引用以上库文件：LD_PRELOAD=/home/qspace/lib/libprog2.so ./prog

当在prog程序中调用 read() 时，使用的就是我们实现的 read() 函数。

对于glibc函数设置钩子的方法，可参考：Let's Hook a Librarg Function

libco

有了以上准备工作，我们可以构建这样的server框架：

accept进程 - epoll进程(n个epoll协程) - n个worker进程(每个worker进程n个worker协程)

该框架下，接收请求、业务逻辑处理、应答都可以看做单独的任务，相应的epoll、worker协程事先分配，服务流程如下：

1. mainloop主循环，负责 i/监听请求事件，有请求则拉起一个worker协程处理；ii/如果timeout时间内没有请求，则处理就绪协程(即io操作已返回
2. worker协程，如果遇到io操作则挂起，对fd加监听事件，让出cpu

libco 提供了以下接口：

- co_create: 创建协程，可在程序启动时创建各任务协程
- co_yield: 协程主动让出cpu，调io操作函数后调用
- co_resume: io操作完成后(触发相应监听事件)调用，使协程继续往下执行

socket族函数(socket/connect/sendto/recv/recvfrom等)、io函数(read/write) 在libco的co_hook_sys_call.cpp中已经重写，以read为例：

复制代码

```
ssize_t read( int fd, void *buf, size_t nbyte )

{
   struct pollfd pf = { 0 };
   pf.fd = fd;
   pf.events = ( POLLIN | POLLERR | POLLHUP );

   int pollret = poll( &pf,1,timeout );  /*对相应fd设置监听事件*/
   ssize_t readret = g_sys_read_func( fd,(char*)buf ,nbyte );   /*真正调用read()*/
   return readret;
}
```

复制代码

小结

由最简单的单任务处理，到多进程/多线程(并行)，再到协程(异步)，server在不断地往极致方向优化，以更好地利用硬件性能的提升(多核cpu的出现、不断提升)。

对程序员而言，可时常检视自己的程序，是否做好并行与异步，在硬件性能提升时，程序服务能力可不可以有相应比例的提升。