Concurrent Programming

程羽

2019 年 12 月 19 日

Overview

并发编程的困难

- 运行过程是不确定的
- Data Race: 同时使用一个资源。例子：同时买最后一张机票。
- Deadlock: 死锁
  - printf 不是异步信号安全的函数
  - printf 需要一个共享的缓冲区，每次使用缓冲区的时候都要设置一个“锁”，使用完才消除“锁”
  - 如果在信号处理函数中使用 printf 就会导致出现“死锁”的现象
- Starvation: 资源分配不均，如设置不合理的优先级

并发编程的必要性

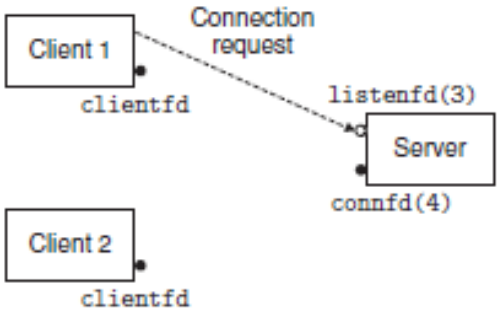
- 访问慢速 I/O 设备
- 人机交互
- 通过推迟工作以降低延迟: 使用 Malloc 合并空闲块时，推迟合并
- 在编写服务器时，需要使服务器可以为多个客户端同时服务。
- 多核机器上的并行计算

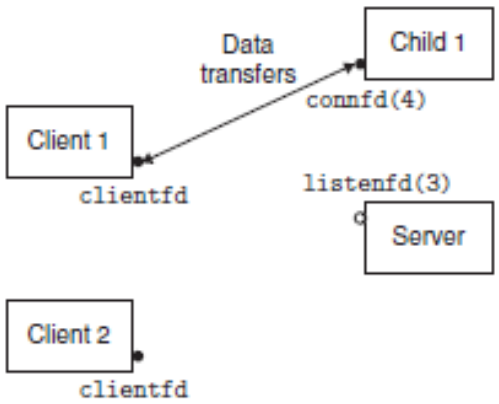
基于进程

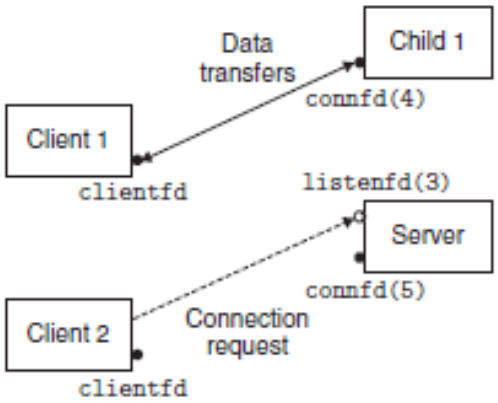
- 为每个逻辑控制流创建一个进程，由内核来维护和调度
- 每个逻辑控制流有独立的虚拟地址空间
- 控制流之间的通信是跨进程的，需要使用显示的进程间通信

基于进程

- step1: 服务器监听监听描述符 (3), 接受了客户端 1 的连接请求
- step2: 服务器 fork 一个子进程负责服务客户端 1(子进程中关闭监听描述符 (3), 父进程中关闭已连接描述符 (4))
- step3: 如果继续由客户端的连接请求，循环 step1,step2







基于进程的并发服务器

```c
void sigchld_handler(int sig)
{
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}
int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;

    Signal(SIGCHLD, sigchld_handler);
    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(struct sockaddr_storage);
        connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        if (Fork() == 0) {
            Close(listenfd); /* Child closes its listening socket */
            echo(connfd);    /* Child services client */
            Close(connfd);   /* Child closes connection with client */
            exit(0);         /* Child exits */
        }
        Close(connfd); /* Parent closes connected socket (important!) */
    }
}
```

基于进程的并发服务器

注意点：

- SIGCHLD 处理函数必须回收所有僵尸进程
- 调用 fork 后父进程必须关闭 connfd(父子进程指向同一个文件表表项，refcnt(connfd)=2, 只有父进程关闭 connfd 系统才可能释放文件表条目)
- 子进程需要关闭 listenfd

基于进程的优缺点

注意点：

- 优点：
  - 直接利用进程的机制，每个控制流拥有独立的空间，不会遭到修改。
  - 可以较为方便的利用多核
- 缺点：
  - 每次都要复制所有上下文信息，空间开销大
  - 进程间的通讯较为不便，开销较高

基于事件

- 事件：来自用户的 (如鼠标、键盘事件等)、来自硬件的 (如时钟事件等) 和来自软件的 (如操作系统、应用程序本身等)
- I/O 是阻塞的：无法处理多个独立的 I/O 事件
- I/O 是非阻塞的：需要不断循环询问来处理多个 I/O 事件，cpu 空转浪费资源
- I/O 多路复用：寻找一种方式可以同时监视多个 I/O 事件 (select,epoll)

基于事件

- 与信号处理相似，一个 I/O 事件就类似一个信号
- select 函数挂起进程，监视一个读集合，直到其中至少一个描述符准备好读，select 返回准备好集合. 通过 FD_ISSET 寻找哪一个描述符准备好读
- 根据准备好读的描述符类型进行相应的操作

基于事件驱动服务器

- pool 结构维护所有活动的客户端集合
- select 函数检测两种事件：
  - (1) 新客户端的连接请求到达: 调用 add_client 函数将客户端加入 pool
  - (2) 一个已存在的客户端的已连接描述符准备好读：调用 check_client 函数送回一个文本行

基于事件

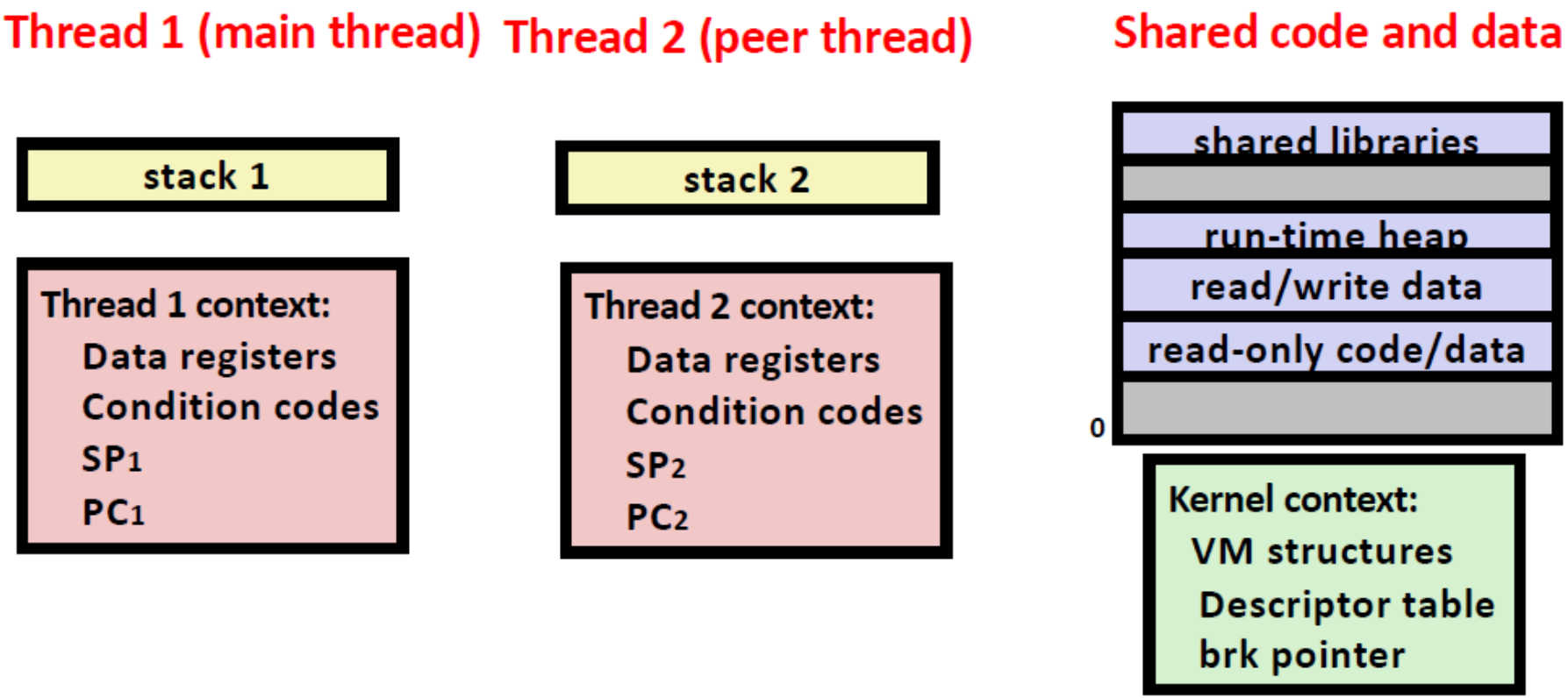
- 优点：
  - 全部都在单一上下文中进行，不需要切换上下文，流之间的数据共享简单高效
  - 没有多进程，便于 debug
- 缺点：
  - 编码复杂
  - 不能充分利用多核处理器
- 事实上高性能服务器都使用事件驱动.

线程

- 线程是运行在进程上下文中的逻辑流
- 每个线程有自己的线程上下文 (TID, 栈，栈指针，PC, 寄存器，条件码)
- 多个线程运行在一个进程的上下文中，所有运行在一个进程里的线程共享整个虚拟地址空间 (代码，数据，堆，共享库，打开文件)
- 每个进程开始时都是单线程的 (称为主线程), 之后创建的线程成为对等线程
- 线程中没有父子关系，所有线程的关系都是对等的 (线程池)

线程

- 由于所有线程是在同一个虚拟内存空间中的，线程的私有内容可能会遭到其他线程的修改



线程

- 线程是操作系统可以调度的最小单位，可以并行
- 线程的上下文比进程的上下文小得多，切换开销较小 (大约为进程的 50%)

Posix 线程

- Posix: 可移植操作系统接口 (Portable Operating System Interface)
- Posix threads 是 C 程序中处理线程的一个接口
  - pthread_creat()
  - pthread_join()
  - pthread_self()
  - pthread_cancel()
  - pthread_exit()

Posix 线程

创建一个线程，代码和本地数据封存在线程例程中

- 以指向一个结构的指针 arg 作为传入参数
- f 为线程运行函数的起始地址
- 如果成功返回 0，如果失败返回错误编号

```
1 #include<pthread.h>
2 typedef void* (func)(void *);
3 int pthread_creat(pthread_t *tid, pthread_attr_t *attr, func *f, void arg);
```

Posix 线程

终止线程

- 隐式终止：线程例程返回
- 显式终止：调用 pthread_exit()
  - 如果主线程调用 pthread_exit()，会等待所有其他对等线程终止再终止主线程和整个进程
- 某个线程调用 exit 函数，终止进程和其中的所有线程
- 一个对等线程利用 pthread_cancel(pthread_t tid) 终止 tid 对应的线程
- 注意：线程终止后资源仍未被回收

Posix 线程

回收已终止线程资源

- 线程是可结合 (joinable) 的或分离的 (detached)
- 可结合的线程可被其他线程回收和杀死，但不会自动释放内存资源
- pthread_join(pthread_t tid)) 函数等待 tid 对应的线程终止，回收该线程占据的资源.
- pthread_join() 只能回收指定 tid 的线程，而无法像回收进程那样回收任意一个终止的线程
- 分离的线程不能被其他线程回收或杀死，终止时系统自动释放占据的内存空间
- 线程默认状态是可结合的，但是实际应用时，应将每个线程设置为分离的

基于线程的并发服务器

```
int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_t clientlen;
    struct sockaddr_storage clientaddr;
    pthread_t tid;

    listenfd = Open_listenfd(argv[1]);
    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
    return 0;
}
```

基于线程的并发服务器

- 如何将已连接描述符 connfd 传递给新创建的线程?
  - 如果使用普通变量，会造成竞争！(在多核情况下更加严重)
  - 应当使用 Malloc
- 在对等线程的例程中，应当释放主线程 Malloc 的内存空间
- 每次创建新的线程，应进行分离
- 应当使用线程安全的函数

```c
/* Thread routine */
void *thread(void *vargp)
{
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
    Free(vargp);
    echo(connfd);
    Close(connfd);
    return NULL;
}
```

echoservert.c

基于进程的并发服务器

- 优点：
  - 线程间通讯较为方便
  - 线程的切换效率高于进程切换
- 缺点：
  - 线程的数据可能遭到其他线程的修改，引起难以察觉的错误
  - 可能会发生不良竞争

Thanks!