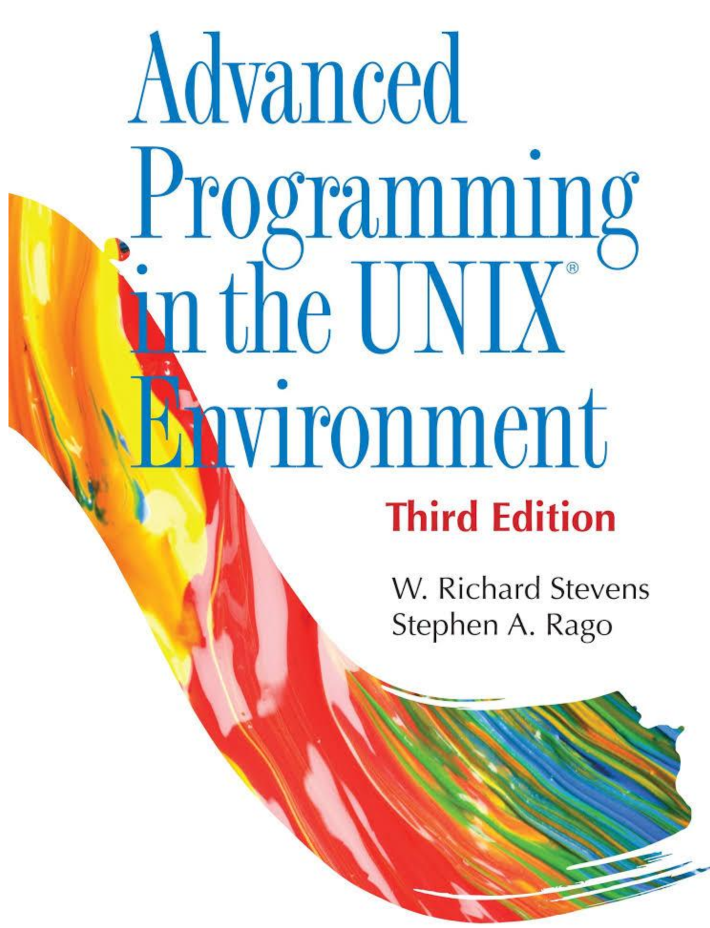
分布式系统编程

系统/网络编程



ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

▸ Updated Classic!

UNIX®
Network
Programming

The Sockets Networking API

THIRD EDITION
VOLUME 1

W. RICHARD STEVENS
BILL FENNER
ANDREW M. RUDOFF

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Unix设计哲学

- Everything is a file
- Small tools
- 组合性

用户态 vs 内核态

- **系统调用边界**
- **上下文切换成本**

fork/exec模型

- **基本概念**
- **fork()**
  - 创建子进程（复制当前进程）
  - 返回值:
    - 子进程：0
    - 父进程：子进程PID
- **exec()**
  - 用新程序**替换当前进程映像**
  - 成功后**不返回**

父进程

先复制，再替换

```
父进程
 |
fork()
 |
 +---- 父进程（继续执行）
 |
 +---- 子进程
        |
      exec()
        |
     新程序运行
```

fork/exec编程

```c
#include <unistd.h>
#include <stdio.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        // 子进程
        execl("/bin/ls", "ls", "-l", NULL);
        perror("exec failed");
    } else {
        // 父进程
        printf("child pid = %d\n", pid);
    }
    return 0;
}
```

- **地址空间fork：**
  - 写时复制（Copy-On-Write）
- **父子进程初始共享物理页**
- **文件描述符fork后共享FD**

线程

- **Pthread（POSIX Thread）**
- **Unix/Linux 标准线程库（POSIX 标准）**
- **提供**
  - 用户级并发控制接口
- **支持：**
  - 线程创建 / 终止
  - 同步（mutex / condition variable）
  - 线程通信

- **Thread（线程）**
  - 共享进程地址空间
- **Race Condition（竞态条件）**
  - 多线程访问共享数据导致不确定行为
- **Critical Section（临界区）**
  - 需要加锁保护的代码区域

pthread编程

```c
#include <pthread.h>

void* worker(void* arg) {
    printf("Hello from thread\n");
    return NULL;
}

int main() {
    pthread_t tid;
    pthread_create(&tid, NULL, worker, NULL);
    pthread_join(tid, NULL);
    return 0;
}
```

```c
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&lock);
// 临界区
pthread_mutex_unlock(&lock);
```

```c
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

pthread_cond_wait(&cond, &lock);
pthread_cond_signal(&cond);
```

文件描述符

- 统一抽象：文件/socket/pipe

进程间通信（IPC）机制

- 管道
- 共享内存
- 信号量

Socket API

- **socket/bind/listen/accept**

I/O模型

- blocking
- non-blocking
- multiplexing
- async

select vs epoll

- O(n) vs O(1)
- 扩展性

epoll示例代码

```
int epfd = epoll_create1(0);
epoll_ctl(epfd, EPOLL_CTL_ADD, sockfd, &ev);
while (1) {
  int n = epoll_wait(epfd, events, MAX, -1);
  for (int i = 0; i < n; i++) {
    handle(events[i]);
  }
}
```

TCP服务器示例

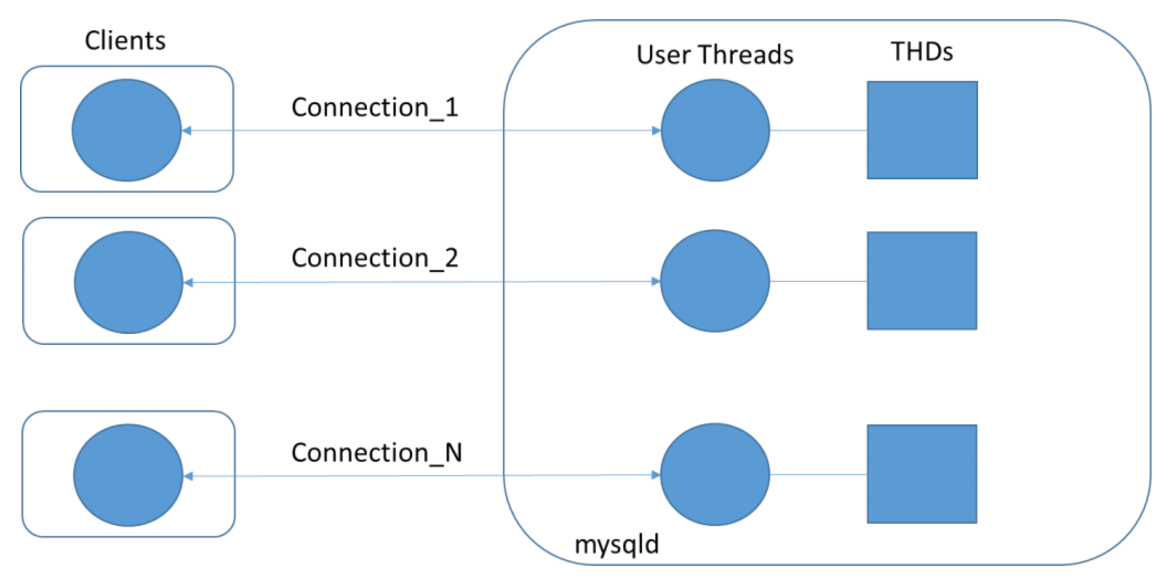
```
int listenfd = socket(...);
bind(listenfd,...);
listen(listenfd,...);
while (1) {
int conn = accept(listenfd,...);
read(conn,...);
write(conn,...);
}
```

- **Redis TCP server**
  - redis/src/server.c
  - redis/src/networking.c
  - redis/src/ae.c （事件循环）

```
int listenToPort(int port, int *fds, int *count) {
    int s = socket(AF_INET, SOCK_STREAM, 0);

    int yes = 1;
    setsockopt(s, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(yes));

    struct sockaddr_in sa;
    sa.sin_family = AF_INET;
    sa.sin_port = htons(port);
    sa.sin_addr.s_addr = htonl(INADDR_ANY);

    bind(s, (struct sockaddr*)&sa, sizeof(sa));
    listen(s, 511);   // backlog

    fds[*count] = s;
    (*count)++;
    return C_OK;
}
```

传统线程/进程模型

- 每个请求创建一个线程/进程
- 顺序代码逻辑简单
- 长期被服务器程序采用



MySQL Connection Handling

Thread-per-request 架构

Client Request

Thread

Application Logic

Response

Why Threads Are A Bad Idea (for most purposes)

John Ousterhout
Sun Microsystems Laboratories

john.ousterhout@eng.sun.com
http://www.sunlabs.com/~ouster

Thread

- **线程（Thread）起源于操作系统**
- **后来发展为用户级编程工具**
- **被广泛认为是解决并发问题的通用方案**

- **问题：**
  - 线程编程非常困难
- **核心观点：**
  - 大多数场景下，事件驱动优于线程
  - 线程只在真正需要CPU并行时使用

什么是线程

- 一种通用并发管理机制
- 多个执行流（execution streams）
- 共享状态（内存、文件等）
- 抢占式调度（preemption）
- 需要同步机制（锁、条件变量）

线程的应用场景

- **操作系统（每个进程一个线程）**
- **科学计算（多核并行）**
- **分布式系统（并发处理请求）**
- **GUI：**
  - 响应用户操作
  - 多媒体/动画

线程的问题

- 对大多数程序员来说太难
- 即使专家也很痛苦
- 👉 编程难度阶梯：
- 普通程序员 → C → C++ → 线程专家

线程为什么难

- 同步问题
- 必须用锁保护共享数据
- 忘记加锁 → 数据损坏

- 死锁问题
- 锁循环依赖
- 系统卡死

线程为什么难

- **性能问题**
  - 粗粒度锁 → 并发低
  - 细粒度锁 → 复杂度高
- **系统限制**
  - 上下文切换开销
  - 调度成本
- **工具支持差**
  - 不易移植
  - 库不线程安全
  - 调试工具少

Event Loop 架构

Event Source

Event Queue

Event Loop

Handlers

事件驱动模型

- 单执行流（无多线程并发）
- 基于事件回调（callbacks）
- 事件循环（event loop）
- handler 不被抢占
- handler 通常很短

事件的应用

- GUI：
- 每个操作一个 handler
- 分布式系统
- socket / I/O 事件处理
- 请求 → 响应

事件循环伪代码

```
while True:
events = epoll_wait()
for e in events:
    handle(e)
```

事件驱动优势

- 减少锁
- 减少上下文切换
- 可扩展到大量连接

典型系统

- nginx
- redis
- Node.js

事件的缺点

- 长任务 → 阻塞界面
- 需要拆分任务
- 不能保持局部状态
- 无CPU并行
- I/O支持不完善

事件 vs 线程

- 事件驱动
- 避免并发
- 无锁、无死锁
- 易上手

- 实现复杂性高
- 需要从一开始处理完整的复杂性

调试与性能对比

- **调试**
  - 事件：简单（与事件相关）
  - 线程：复杂（调度+时序）

- **性能**
- **单CPU：事件更快**
  - 无锁
  - 无上下文切换

线程的价值

- 线程的价值
- 真正的CPU并行
- 支持长时间任务
- 多核扩展能力

- 建议：
- 使用线程
  - 并行（parallelism）✅
- 绝大多数情形下，不使用线程
  - 并发（concurrency）❌

第15页：结论

- **线程概念上简单，但用起来很难**
- **并发本质上很难**
- **线程强大但很少必要**
- **线程难以编程（专家级）**
- 👉 **建议：**
  - GUI → 用事件
  - 分布式系统 → 用事件
  - 普通服务器 → 用事件
  - 高性能计算 → 用线程（并行）

类似的观点

Making reliable
distributed systems
in the presence of
software errors

Final version (with corrections) — last update 20 November 2003

Joe Armstrong

A Dissertation submitted to
the Royal Institute of Technology
in partial fulfilment of the requirements for
the degree of Doctor of Technology
The Royal Institute of Technology
Stockholm, Sweden

December 2003

Erlang的一些特性
被现代编程语言所继承

A Note on Distributed Computing

Jim Waldo
Geoff Wyant
Ann Wollrath
Sam Kendall

SMLI TR-94-29 November 1994

1994

- **当时流行：**
  - RPC（Remote Procedure Call）
  - CORBA
  - 分布式对象
  - 这些系统试图让开发者“感觉不到网络存在”
- **文章说一件事：**
  - 让远程调用看起来像本地调用”是**错误抽象**

分布式系统通信

- **两个重要模型**
- **Message Passing**
- **Remote Procedure Call (RPC)**

Message Passing 模型

Process A

send()

Network

receive()

Process B

Message Passing 特点

- 通信显式
- 模块解耦
- 系统结构清晰

RPC 模型

- 远程过程调用
- 看起来像本地函数

RPC 调用流程

Client Call

Serialize

Network

Server Execute

Return

Message Passing vs RPC —— 论文观点

- RPC 提供更高抽象
- Message Passing 提供更大控制力
- 两者适用于不同系统

RPC 优点

- 简单易用
- 隐藏通信细节
- 开发效率高

RPC 问题

- 隐藏网络延迟
- 隐藏网络失败
- 容易产生错误假设

RPC 的经典谬误

- 网络是可靠的
- 延迟可以忽略
- 带宽是无限的

Message Passing 优势

- 明确网络边界
- 更适合分布式系统
- 错误处理更清晰

论文结论

- RPC 适合简单服务调用
- Message Passing 更适合复杂分布式系统
- 系统设计需要权衡

现代服务器架构

- I/O 使用事件驱动
- CPU 使用线程池
- 避免阻塞 event loop

总结

- **理解常用系统/网络编程API、模型**
- **Thread/Event**
  - 线程模型（理解）简单，但（使用）复杂度高
  - 事件驱动，更适合高并发系统
- **Message Passing/RPC**
  - RPC，（理解）简单，但（使用）复杂度高
  - Message Passing在复杂异步系统上更合适