第12章 并发编程

基于Process并发编程

- 共享文件表，不共享地址空间
- 共享信息必须使用显式的IPC(interprocess communications)
- 进程控制和IPC的开销都很高

基于I/O Multiplexing并发编程

- 解决多个独立I/O事件的响应
- 单进程，共用地址空间
- 编码复杂，容易被部分发送的情况卡断
- 不能利用多核处理器

基于Thread并发编程

- 是前面两种方法的混合
  - 与Process一样，线程由内核自动调度，并且内核用一个ID识别线程
  - 与I/O Multiplexing一样，多个线程运行在单一进程上下文中，共享虚拟地址空间
- 线程与进程的不同：
  - 线程的切换开销小得多
  - 线程不像进程那样按照父子层次来组织，与一个进程相关的线程组成一个pool of peers，独立于其他线程创建的线程
  - 主线程总是进程中第一个运行的线程
  - pool of peers意味着，一个线程可以杀死任何peers或者等待peers终止
  - 每个peer可以读写相同的共享数据
- Posix线程(Pthreads)是C程序中处理线程的一个标准接口
- 函数

```
#include <pthread.h>
typedef void *(func)(void *);

//成功返回0，出错非零
//创建新的线程，带着输入变量arg，在新线程上下文中运行f，用attr参数改变新线程的默认
属性(通常为NULL)
//参数tid返回新线程ID
int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void
*arg);

//返回自己的线程ID
pthread_t pthread_self(void);

//线程return时隐式终止
//调用该函数显式终止
//主线程调用会等待所有其他peer threads终止，再终止主线程和整个进程，返回值为
thread_return
//从不返回
```

```
void pthread_exit(void *thread_return);

//peer thread调用exit终止进程和所有相关的线程
//peer thread调用pthread_cancel终止tid线程
//成功返回0，出错非零
int pthread_cancel(pthread_t tid);

//阻塞直到线程tid终止，将线程返回的void*赋值为thread_return指向的位置，并回收其内存资源
//成功返回0，出错非零
int pthread_join(pthread_t tid, void **thread_return);

//每个joinable的线程都必须被其他线程显式收回，否则都应该detached
//成功返回0，出错非零
int pthread_detach(pthread_t tid);

pthread_once_t once_control = PTHREAD_ONCE_INIT;
//总是返回0
//只初始化一次，用于动态初始化多个线程共享的全局变量
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

- 需要注意的问题
  1. 传递线程参数时，将参数动态分配到自己的内存块，否则可能引入race
  2. 如果不显式收回线程，必须分离每个线程，及其分配的内存块，避免内存泄漏
- Shared Variables
  - 寄存器是从不共享的，而虚拟内存总是共享的
  - Global variables: 在虚存的读写区域仅包含一个实例，任何线程都可引用
  - Local automatic variables: 每个线程的栈都包含自己的实例
  - Local static variables: 在虚存的读写区域只包含一个实例，每个peer thread都读写这个实例
  - 只要能够获取到其他线程的变量的地址，那么也都是共享的
- Semaphore
  - 用于同步线程
  - 典型错误：两个线程为同一共享变量加1，重复1亿次，结果不是2亿
  - Progress Graphs
    - critical section不应该与其他进程交替执行，即拥有对共享变量的mutually exclusive access，称为mutual exclusive(互斥)
    - 两个critical section交集形成unsafe region，不应该进入
  - semaphore(信号量)
    - 信号量s是具有非负整数值的全局变量，只能由P和V两种操作处理
    - P(s): 如果s非零，s减1，立即返回；如果s为零，挂起线程直到s非零(V会重启这个线程)，重启后再将s减1，将控制返回给调用者
    - V(s): 将s加1，如果由线程阻塞在P，那么重启其中的一个(不可预测重启哪一个)
    - P中的测试和减1不可分割，V中的加1也不可分割

```
#include <semaphore.h>

//成功返回0，出错返回-1
//将信号量sem初始化为value
```

```
int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *s); /* P(s) */
int sem_post(sem_t *s); /* V(s) */

#include "csapp.h"
void P(sem_t *s);
void V(sem_t *s);
```

  - 使用初始为1的信号量称为binary semaphore, 也称为mutex(互斥锁), P操作称为对互斥锁加锁，V操作称为对互斥锁解锁，加锁但没有解锁的线程称为占用这个互斥锁。

```
volatile long cnt = 0; /* Counter */
sem_t mutex;           /* Semaphore */

Sem_init(&mutex, 0 ,1);
for(i = 0; i < niters; i++){
    P(&mutex);
    cnt++;
    V(&mutex);
}
```

  - 生产者-消费者问题：也即提供一个缓冲槽
  - 读者-写者问题
    - 第一类问题：读者优先，即读者不会因有一个写者在等待而等待
    - 第二类问题：写者优先，即在一个写者后到达的读者必须等待
    - 这两种都可能导致starvation
  - 基于prethreading的并发服务器：利用的是生产者-消费者模型，类似连接池的概念
- 使用线程提高Parallelism
  - 并行程序是一个运行在多个处理器上的并发程序
  - 同步操作的开销很大
  - 使用局部变量减少内存引用可以提高效率
  - 并行程序通常写为每个核上只运行一个线程

其他并发问题

- Thread Safety
  - 一个函数称为thread-safe, 当且仅当被多个并发线程反复调用时总是产生正确的结果。
  - thread-unsafe的类型

类型	举例
1. 不保护共享变量的函数	P&V保护计数器
2. 保持跨越多个调用状态的函数	伪随机数生成器rand
3. 返回指向静态变量的指针的函数	ctime，gethostbyname等
4. 调用线程不安全函数的函数	用互斥锁保护一下

- Reentrancy

    - 可重入函数是一种线程安全函数，与一般线程安全函数相比更高效，因为不需要同步操作
    - 例如把rand函数重新为可重入的版本，用一个调用者传递进来的指针取代了静态的next变量
- 在线程化程序中使用库函数
    - 多数库函数(malloc,free,realloc,printf,scanf)都是线程安全的，只有一小部分是例外，例如：rand,strtok,asctime,ctime,gethostbyaddr,gethostbyname,inet_ntoa,localtime...
    - Linux系统提供了可重入版本，总是以_r后缀结尾
- Race：当一个程序的正确性因不同的执行流轨迹线而受影响时，就发生了竞争
- Deadlock:
    - 使用Process graph
    - Mutex lock ordering rule: 如果每个线程都是以一种顺序获得互斥锁并以相反的顺序释放，那么这个程序就是无死锁的