信号量 & POSIX线程编程

蒋炎岩

南京大学 | 计算机软件研究所 | 系统与软件分析研究组

复习：并发控制

- 互斥
  - lock(&lk) / unlock(&lk)
- 条件变量
  - wait(&cond, &mutex) / signal(&cond)

同步(2)：信号量

生产者/消费者的麻烦

- 我们用了三样东西
  - 互斥锁：保证操作的原子性
  - 条件变量：管理睡眠/唤醒
  - 计数器：维护资源的数量
- 信号量：合三为一
  - by E. W. Dijkstra
  - as a single primitive for all things related to synchronization

```
void producer_thread() {
  for (int i = 0; i < loops; i++) {
    lock(&mutex);
    while (count == N)
      wait(&empty, &mutex);
    printf("("); count++;
    signal(&fill);
    unlock(&mutex);
  }
}
void consumer_thread() {
  for (int i = 0; i < loops; i++) {
    lock(&mutex);
    while (count == 0)
      wait(&fill, &mutex);
    printf(")"); count--;
    signal(&empty);
    unlock(&mutex);
  }
}
```

信号量(Semaphore)

- P (prolaag = try + decrease), wait, down
- V (verhoog = increase), post, up

- 信号量自带计数器、互斥、同步vim

P/V操作(原子操作)

- 自带计数器(count)
- P(sem)
  - count -= 1
  - 如果余额不足，睡眠
- V(sem)
  - count += 1
  - 如果有人在睡眠，唤醒

```
void P(sem_t &sem) {
  sem->count--;
  if (sem->count < 0) {
    push(sem->queue, current);
    suspend();
  }
}

void V(sem_t &sem) {
  sem->count++;
  if (!empty(sem->queue)) {
    wakeup(pop(sem->queue));
  }
}
```

- 思考题：count代表什么？
- 思考题：如何用wait/signal实现P/V？

实现：兼容假唤醒

- The F**k Manual: a thread might be awoken from its waiting state even though no thread signaled the condition variable
  - spurious wakeup
  - 因为“不可告人的实现原因”
- 反过来实现则困难得多

```
void P() {
  lock(&mutex);
  while (sem->count == 0) {
    wait(&cond, &mutex);
  }
  sem->count--;
  unlock(&mutex);
}

void V() {
  lock(&mutex);
  count++;
  signal(&cond);
  unlock(&mutex);
}
```

信号量：可以当锁使用

- 思考题：SEM_INIT如何实现？
  - pthread里也有类似的技巧
  - PTHREAD_MUTEX_INITIALIZER
- 思考题：为什么它正确？

```
sem_t mutex = SEM_INIT(1);
void lock() {
  P(&mutex);
}

void unlock() {
  V(&mutex);
}
```

信号量：生产者-消费者问题

- 自带条件变量+计数器
  - Cool!
- count是资源的个数
  - 信号量适用于资源可以用整数表示的情况
  - 而这是绝绝绝绝大部分实际情况

```
sem_t empty = SEM_INIT(N);
sem_t fill = SEM_INIT(0);

void producer() {
  while (1) {
    P(&empty);
    printf("(");
    V(&fill);
  }
}

void consumer() {
  while (1) {
    P(&fill);
    printf(")");
    V(&empty);
  }
}
```

POSIX线程编程

线程API：创建、退出、等待

- 类似于进程的fork + execve; exit; waitpid
  - 大家已经做过实验了
  - 只是一切都发生在共享内存

```
int pthread_create(
  pthread_t *thread,
  const pthread_attr_t *attr,
  void *(*start_routine) (void *),
  void *arg);

void pthread_exit(void *retval);

int pthread_join(pthread_t thread, void **retval);
```

线程执行需要：代码、数据、堆栈

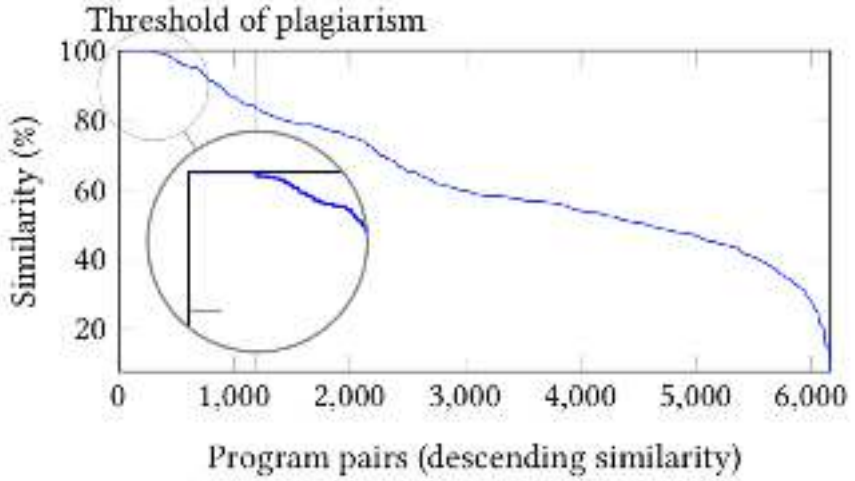
- 代码和数据是共享的，但需要有独立的堆栈
  - 堆栈分配在同一地址空间：局部变量可以被取地址并共享

- 思考题：怎样知道每个线程的堆栈被映射到了虚拟地址空间的何处?
  - Native POSIX Thread Library (NPTL), The F**k Manual: On Linux/x86-32, the default stack size for a new thread is 2 megabytes (好少).
  - 改变`pthread_attr_t`设置得到更大的堆栈

线程有三宝：创建、退出、等你好

- pthread: create, exit, join
  - 有了这三个API，从今天开始你就可以写并行程序啦！

- 例子：代码查重
  - 要跑很多类似的实例
  - 就多线程并发跑
  - 24C机器 = 24X速度提升



同步有三宝：互斥、CV、信号量

- 互斥：pthread_mutex_t
  - init, lock, trylock, unlock, ...
- 读/写锁：pthread_rwlock_t
  - 同mutex，但允许多个读者同时进入临界区
  - 思考题：在什么条件下有用?
- 条件变量：pthread_cond_t
  - init, wait, timedwait, signal, broadcast, ...
- 信号量：sem_t
  - init, post, wait, getvalue, destroy

POSIX线程信号量

```
int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_post(sem_t *sem);
int sem_wait(sem_t *sem);
int sem_getvalue(sem_t *sem, int *sval);
int sem_destroy(sem_t *sem);
```

- OSLab1里你要在你自己的操作系统里实现信号量(Cool!)
  - 而且能正确运行生产者-消费者问题

真实世界的例子：sqlite3

- 一把大锁完事
  - SQLite3是跨平台兼容的项目，所以用自己的sqlite3_mutex进行了包装
  - 更多开源项目假设有POSIX线程

```
148 sqlite3_mutex_enter(db->mutex); in sqlite3_blob_open()
361 sqlite3_mutex_enter(db->mutex); in sqlite3_blob_close()
388 sqlite3_mutex_enter(db->mutex); in blobReadWrite()
486 sqlite3_mutex_enter(db->mutex); in sqlite3_blob_reopen()
```

真实世界的例子：pbzip2 (parallel bzip2)

- 没什么可怕的：又见producer/consumer

```
164 int producer(int, int, off_t, int, queue *);
165 void *consumer(void *);
```

- 就是你见过的
  - producer: while (1) printf(“(“);
  - consumer: while (1) printf(“)”);

Producer/Consumer in pbzip2

```
762 int producer(int hInfile, off_t fileSize,
                int blockSize, queue *fifo) {
774   while (bytesLeft > 0) {
786     pthread_mutex_lock(MemMutex);
790     pthread_mutex_unlock(MemMutex);
801     ret = read(hInfile, (char *) FileData, inSize);
826     bytesLeft -= ret;
836     pthread_mutex_lock(fifo->mut);
837     while (fifo->full) {
842       pret = pthread_cond_wait(fifo->notFull, fifo->mut);
845     }
850     queueAdd(fifo, FileData, inSize, blockNum);
851     pthread_mutex_unlock(fifo->mut);
852     pthread_cond_signal(fifo->notEmpty);
854     blockNum++;
855   } // while
859   allDone = 1;
860   return 0;
861 }
```

所以……

- Don't panic
  - 并不是人人都想把事情搞砸
  - 绝大部分时候用简单的并发模式就能解决问题了

- 当然也有panic的地方
  - 比如Linux Kernel (你的OS Kernel会比应用程序复杂一些)