
阅码场
YOMOCODE

多线程编程(5)

麦当劳喜欢您来，喜欢您再来

扫描关注
Linux阅码场

多线程编程(5)

如何设计多线程程序
5.1 划分线程的典型原则
5.2 流水线
5.3 工作组
5.4 线程池
5.5 Amdahl定律
5.6 多线程与I/O, select/epoll

编程原则

EIGHT SIMPLE RULES FOR DESIGNING MULTITHREADED APPLICATIONS

Rule 1: Identify Truly Independent Computations

Rule 2: Implement Concurrency at the Highest Level Possible

Rule 3: Plan Early for Scalability to Take Advantage of Increasing Numbers of Cores

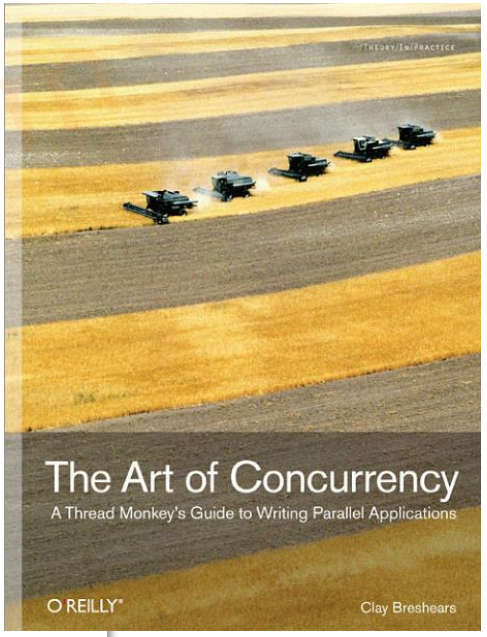
Rule 4: Make Use of Thread-Safe Libraries Wherever Possible

Rule 5: Use the Right Threading Model

Rule 6: Never Assume a Particular Order of Execution

Rule 7: Use Thread-Local Storage Whenever Possible or Associate Locks to Specific Data

Rule 8: Dare to Change the Algorithm for a Better Chance of Concurrency



+ 让I/O与计算型同时运行而不是相互等待

老板-工人(boss-worker)模型 w/o 线程池

一个boss线程负责分派工作(delegation)

```
main()
/* The boss */
{
forever {
            get a request
            switch request
            case X : pthread_create( ... taskX)
            case Y : pthread_create( ... taskY)
            .
            .
            .
}
}
taskX() /* Workers processing requests of type X */
{
perform the task, synchronize as needed if accessing shared
resources
done
}
taskY() /* Workers processing requests of type Y */
{
perform the task, synchronize as needed if accessing shared
resources
done
}
```

老板-工人(boss-worker)模型 w/线程池

一个boss线程负责分派工作(delegation)

```
main()
/* The boss */
{
for the number of workers
          pthread_create( ... pool_base )
forever {
          get a request
          place request in work queue
          signal sleeping threads that work is available
}
}
pool_base() /* All workers */
{
forever {
          sleep until awoken by boss
          dequeue a work request
          switch
            case request X: taskX()
            case request Y: taskY()
                    .
                    .
                    .
}
}
```

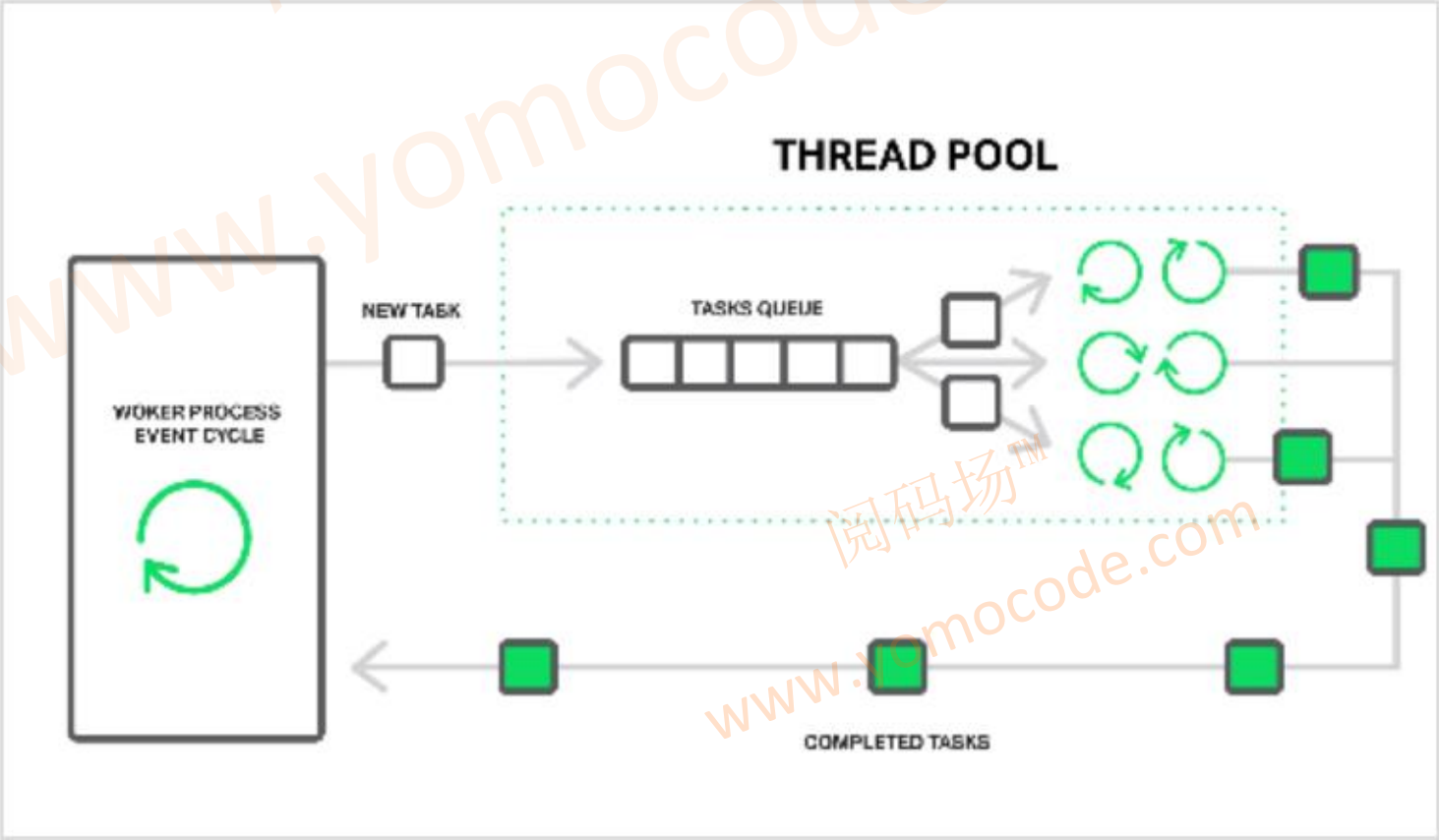
Peer模型

没有boss分派工作的过程，每个线程自己有输入(workcrew)

```
main()
{
    pthread_create( ... thread1 ... task1 )
    pthread_create( ... thread2 ... task2 )
    .
    .
    .
    signal all workers to start
    wait for all workers to finish
    do any clean up
}
task1()
{
    wait for start
    perform task, synchronize as needed if accessing shared
resources
    done
}
task2()
{
    wait for start
    perform task, synchronize as needed if accessing shared
resources
    done
}
```

线程池模型

避免频繁创建撤销，或者不确定数量的创建；
解耦作用：线程的创建与执行完全分开，方便维护；
放入一个池子中，可以给其他任务进行复用。



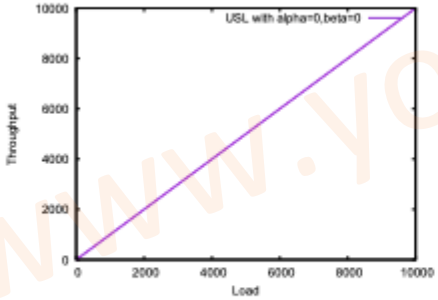
流水线模型

流水线的每一个阶段应尽可能相等时间

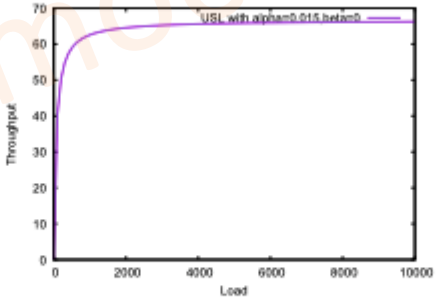
```
//...

   pthread_t Thread[N]
   Queues[N]

   // initial thread
   {
       place all input into stage1's queue
       pthread_create(&(Thread[1]...stage1...);
       pthread_create(&(Thread[2]...stage2...);
       pthread_create(&(Thread[3]...stage3...);
       //...
   }

void *stageX(void *X)
{
   loop
     suspend until input unit is in queue
     loop while XQueue is not empty
         dequeue input unit
         process input unit
         enqueue input unit into next stage's queue
      end loop
   until done
   return(NULL)
}

//...
```

Universal Scalability Law (USL)

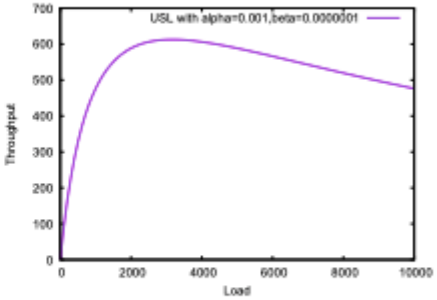
$$C(N) = \frac{N}{1+\alpha(N-1)+\beta N(N-1)}$$



$\alpha = 0, \beta = 0$



$\alpha > 0, \beta = 0$



$\alpha > 0, \beta > 0$

Amdahl's law

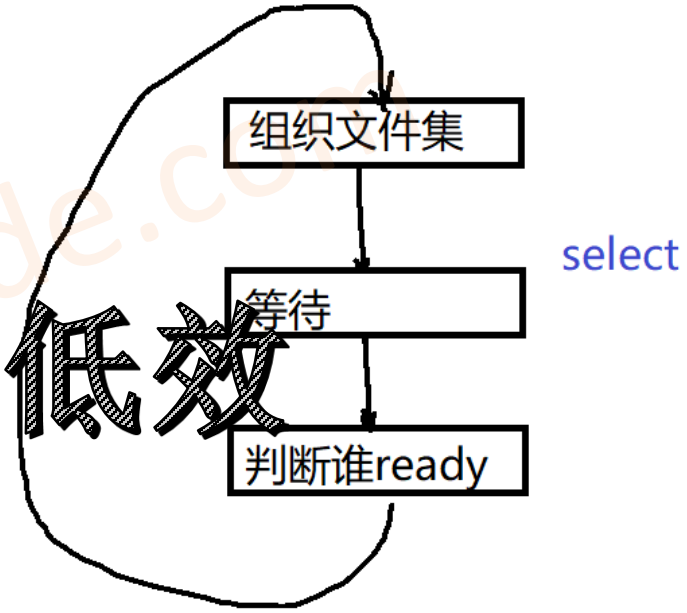
异步I/O

```
#include <aio.h>

int aio_read(struct aiocb *aiocbp);
int aio_write(struct aiocb *aiocbp);
```

```
#2  __pthread_cond_timedwait (cond=0x7ffff7fb32a0 <__aio_new_request_notification>,
    mutex=0x7ffff7fb3200 <__aio_requests_mutex>, abstime=0x7ffff7fcbeb0) at pthread_cond_wait.c:667
--Type <RET> for more, q to quit, c to continue without paging--
#3  0x00007ffff7facbf0 in handle_fildes_io (arg=<optimized out>) at ../sysdeps/pthread/aio_misc.c:629
#4  0x00007ffff7da7164 in start_thread (arg=<optimized out>) at pthread_create.c:486
#5  0x00007ffff7edadef in clone () at ../sysdeps/unix/sysv/linux/x86_64/clone.S:95
```

多路复用

```
int select(int nfds, fd_set
*readfds, fd_set *writefds,
fd_set *exceptfds, struct
timeval *timeout);
```



有时候很低效

```
int epoll_ctl(int epfd, int op, int fd, struct epoll_event *event);
                    EPOLL_CTL_ADD
                    EPOLL_CTL_MOD
                    EPOLL_CTL_DEL
 int epoll_wait(int epfd, struct epoll_event *events,
                    int maxevents, int timeout);
```

select

while(1) 死循环内添加fd到fdset，等待select，检查谁ready-FD_ISSET()

```
while(1){
        FD_ZERO(&rset);
        for (i = 0; i< 5; i++ ) {
                FD_SET(fds[i],&rset);
        }

        puts("round again");
        select(max+1, &rset, NULL, NULL, NULL);

        for(i=0;i<5;i++) {
                if (FD_ISSET(fds[i], &rset)){
                        memset(buffer,0,MAXBUF);
                        read(fds[i], buffer, MAXBUF);
                        puts(buffer);
                }
        }
}
```

epoll

epoll在循环体外添加FD，循环体内epoll_wait，返回值知道ready的文件数量

```
for (i=0;i<5;i++)
{
  static struct epoll_event ev;
  memset(&client, 0, sizeof (client));
  addrlen = sizeof(client);
  ev.data.fd = accept(sockfd,(struct sockaddr*)&client, &addrlen);
  ev.events = EPOLLIN;
  epoll_ctl(epfd, EPOLL_CTL_ADD, ev.data.fd, &ev);
}

while(1){
      puts("round again");
      nfds = epoll_wait(epfd, events, 5, 10000);

      for(i=0;i<nfds;i++) {
                  memset(buffer,0,MAXBUF);
                  read(events[i].data.fd, buffer, MAXBUF);
                  puts(buffer);
      }
}
```

多线程文件冲突

指定偏移offset位置开始读取/写入count个字节

man page

The `pread()` and `pwrite()` system calls are especially useful in multi-threaded applications. They allow multiple threads to perform I/O on the same file descriptor without being affected by changes to the file offset by other threads.

```
#include <sys/types.h>
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbyte, off_t offset);

ssize_t pwrite(int filedes, void *buf, size_t nbyte,
    off_t offset);
```

谢谢!

阅码场出品