Ch12-Concurrent Porgramming

12.1-3 Concurrent Program

以并发的echo网络编程为例，原来的echo服务器一次只能处理一个客户端的请求，accept会滞留一段时间，现在的服务器可以并发地执行多个客户端读写

Concurrent Programming with processes

```
int main(int argc, char **argv)
{
   int listenfd, connfd;
   struct sockaddr_in clientaddr;
   socklen_t clientlen = sizeof(clientaddr);

   Signal(SIGCHLD, sigchld_handler);
   listenfd = Open_listenfd(argv[1]);
   while (1) {
      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!) */
   }
}
```

这里为每个请求都开一个新的子进程处理。为避免内存泄漏，父子进程应当关闭他们各自的connfd；需要使用SIGCHID处理程序回收多个僵死子进程

```c
void sigchld_handler(int sig)
{
   while (waitpid(-1, 0, WNOHANG) > 0)
    ;
   return;
}
```

Pros and cons of process

优点：进程的模型清晰，共享文件表而不共享用户地址空间，不会覆盖另一个进程的虚拟内存
缺点：进程间共享状态变得困难，必须使用显式的IPC(进程间通信)，而且因为开销大，会很慢

Concurrent Programming with threads

一个进程中可以运行多个线程，每个线程有自己的线程上下文，有单独的逻辑控制流，包括TID、栈、栈指针、程序计数器、通用目的寄存器和条件码；一个进程里的线程共享进程的虚拟地址空间，包括代码、数据、堆、共享库和打开的文件。
一个进程的所有线程构成一个对等池(peer thread)

Posix threads (Pthreads) interface

C程序处理线程的一个标准接口，主要有：

- 创建线程：

  ```c
  int pthread_create(pthread_t *tid, NULL, func *f, void *arg); // 获取tid，设置线程例程函数和参
  ```

- 获取tid：

  ```c
  pthread_t pthread_self(void);
  ```

- 回收进程：

  ```c
  int pthread_join(pthread_t tid, void** thread_return); // 获得返回值
  ```

- 终止进程：exit终止所有线程

  ```c
  void pthread_exit(void *thread_return);
  int pthread_cancel(pthread_t tid);
  ```

- 分离线程：默认线程可结合，即需要被其它线程显式回收，否则不会释放；修改为分离的，无法被其它线程杀死或回收，但会自动释放

```c
int pthread_detach(pthread_t tid);
```

- 初始化线程：

```c
/* hello.c - Pthreads "hello, world" program */
#include "csapp.h"
/* thread routine */
void *thread(void *vargp) {
  printf("Hello, world!\n");
  return NULL;
}
int main() {
  pthread_t tid;
  Pthread_create(&tid, NULL, thread, NULL);
  Pthread_join(tid, NULL);
  exit(0);
}
```

基于线程的并发服务器如下：

```c
int main(int argc, char **argv){
    int listenfd, *connfdp
    socklen_t clientlen;
    struct sockaddr_in clientaddr;
    pthread_t tid;
    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    listenfd = open_listenfd(argv[1]);
    while (1) {
        clientlen = sizeof(clientaddr);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}
```

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

这里的create不能直接给&connfd，因为会引发赋值与accept的竞争，可能拿成下一次的connfd

Pros and cons of thread

好处：易于在线程间分享数据结构；开销小于进程，更高效
缺点：容易出现无意识的分享，且难以发现

Concurrent Programming with I/O Multiplexing

```c
#include <sys/select.h>
int select (int maxfd, fd_set *readset, NULL, NULL, NULL);
void FD_ZERO(fd_set *fdset);/* clear all bits in fdset. */
void FD_CLR(int fd, fd_set *fdset);/* clear bit fd in fdset */
void FD_SET(int fd, fd_set *fdset);/* turn on bit fd in fdset */
int FD_ISSET(int fd, *fdset);/* Is bit fd in fdset on? */
```

使用select函数，其休眠到有一个或多个描述符准备好读，返回准备好的描述符数，并重新设置readset，指示准备情况；而maxfd指示检查readset的前maxfd个描述符；初始的readset则会设置需要监听的描述符

下面的例子既响应客户端连接，又响应标准输入

```
#include “csapp.h”

int main(int argc, argv)
{
    int listenfd, connfd;
    socklen_t clientlen=sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    fd_set read_set, ready_set;
    if (argc != 2) {
       fprintf(stderr, “usage: %s <port>\n”,argv[0]);
       exit(0);
    }
    listenfd = Open_listenfd(argv[1]);
    FD_ZERO(&read_set);
    FD_SET(STDIN_FILENO, &read_set);
    FD_SET(listenfd, &read_set);
    while(1) {
        ready_set = read_set;
        Select(listenfd+1, &ready_set,
               NULL, NULL, NULL);
        if (FD_ISSET(STDIN_FILENO, &ready_set)
            /*read command line from stdin */
            command();
        if (FD_ISSET(listenfd, &ready_set)){
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            echo(connfd);
        }
    }
}


void command(void)
{
    char buf[MAXLINE];
    if (!Fgets(buf, MAXLINE, stdin))
        exit(0);/* EOF */
    /*Process the input command */
    printf(“%s”, buf);
}
```

select函数沉睡直到：

- 新的用户端连接到达，其使用listenfd监听，那么建立连接，使用accept和echo

- 新的数据到达标准输入，使用STDIN_FILENO监听，调用command来进行读取响应

下面的例子给出了并发服务器：

```
#include “csapp.h”
typedef struct {/* represents a pool of connected descriptors */
     int maxfd;
     fd_set read_set;
     fd_set ready_set;
     int nready;
     int maxi;
     int clientfd[FD_SETSIZE];
     rio_t clientrio[FD_SETSIZE];
} pool;
int byte_cnt = 0; /*counts total bytes received by server */
int main()
{
  int listenfd, connfd;
  socketlen_t clientlen = sizeof(struct sockaddr_in);
  struct sockaddr_in clientaddr;
  static pool pool ;
  if (argc != 2) {
     fprintf(stderr, “usage” %s <port>\n”, argv[0]);
     exit(0);
  }

  listenfd = Open_listenfd(argv[1]);
  init_pool(listenfd, &pool);
  while (1) {
       /* wait for listening/connected descriptor(s) to become ready*/
       pool.ready_set = pool.read_set;
       pool.nready = Select(pool.maxfd+1,&pool.ready_set, NULL,NULL,NULL);
       /* If listening descriptor ready, add new client to pool*/
       if (FD_ISSET(listenfd, &pool.ready_set)) {
       connfd=Accept(listenfd, (SA *)&clientaddr, &clientlen);
       add_client(connfd, &pool);
       }
       /*Echo a text line form each ready connected descriptor */
       check_clients(&pool);
     }
}
```

```c
void init_pool(int listenfd, pool *p)
{
  /*Initially, there are no connected descriptors */
  int i;
  p->maxi = -1;
  for (i = 0; i < FD_SETSIZE; i++)
     p->clientfd[i] = -1;
  /* Initially, listenfd is only member of select read set */
  p->maxfd = listenfd ;
  FD_ZERO(&p->read_set);
  FD_SET(listenfd, &p->read_set);
}


void add_client(int connfd, pool *p)
{
  int i;
  p->nready--;
  for (i = 0; i < FD_SETSIZE; i++)  /* Find an available slot */
     if (p->clientfd[i] < 0) {
         /* Add connected descriptor to the pool */
         p->clientfd[i] = connfd;
         Rio_readinitb(&p->clientrio[i], connfd);
         /*Add the descriptor to descriptor set */
         FD_SET(connfd, &p->read_set);
         /* Update max descriptor and pool highwater mark */
         if (connfd > p->maxfd)
        p->maxfd = connfd ;
         if (i > p->maxi)
        p->maxi = i;
         break;
     }
  if (i == FD_SETSIZE)         /* Couldn’t find an empty slot */
     app_error(“add_client error: Too many clients”);
}
```

```c
void check_clients(pool *p)
{
  int i, connfd n;
  char buf[MAXLINE];
  rio_t rio;
  for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
     connfd = p->clientfd[i];
     rio = p->clientrio[i];

     /*If the descriptor is ready, echo a text line from it */
     if ((connfd>0) && (FD_ISSET(connfd, &p->ready_set))) {
        p->nready--;
        if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
           byte_cnt += n;
           printf("Server recerived %d (%d total)
               bytes on fd %d\n", n, byte_cnt, connfd);
           Rio_writen(connfd, buf, n);
        }
        /* EOF detected, remove descriptor from pool */
        else {
           Close(connfd);
           FD_CLR(connfd, &p->read_set);
           p->clientfd[i]=-1;
        }
     }
  }
}
```

Pros and cons of I/O Multiplexing

优点：一个逻辑控制流，可单步debug
缺点：代码复杂，并发颗粒度小，不能充分利用多核

12.4 Shared variables in threaded C programs

Threads memory model

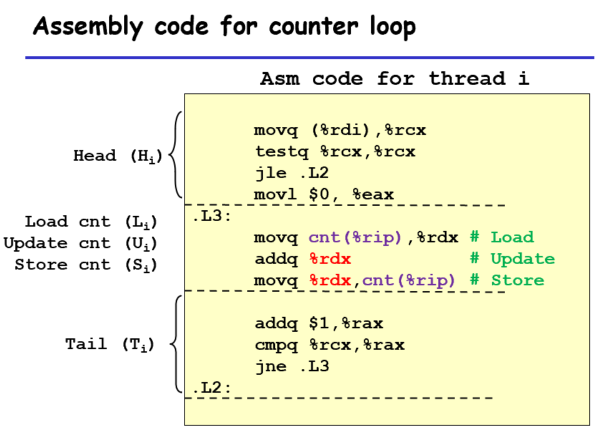
一组并发线程运行在一个进程的上下文中，每个线程都有自己独立的线程上下文；这些线程共享进程上下文的剩余部分
但不是绝对的，寄存器是绝对设防的，但栈不是

Mapping Variable Instances to Memory

- 全局变量：虚拟内存只有其一个实例，任何线程都能引用
- 本地自动变量：每个线程的栈都有自己的本地自动变量的实例
- 本地静态变量：虚拟内存也只有一个实例

TODO: Shared variable analysis

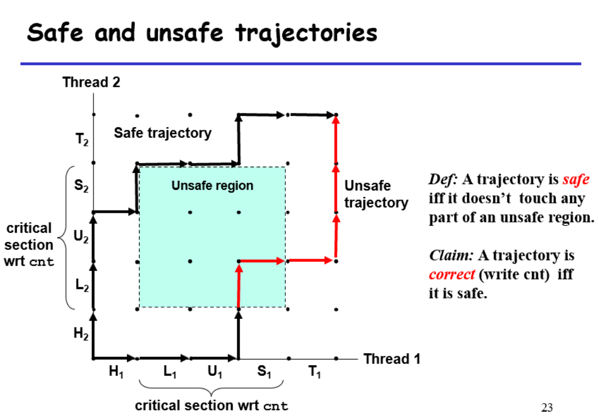
12.5 Synchronizing Threads with Semaphores



执行上述的两个线程，这五步的交错顺序是无法确定的，可能导致计算错误

Progress graphs

轨迹线和不安全区，不安全区的边缘是安全的，想办法同步(Synchronizing)线程，保证安全轨迹线



Semaphores

信号量是具有非负整数值的全局变量，只能进行两种操作：

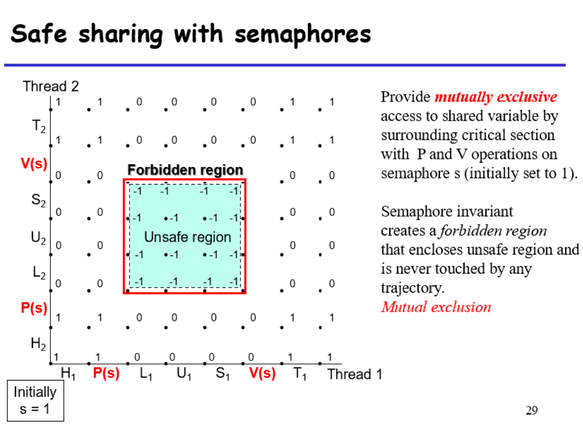
- P(s)：`while (s == 0) wait(); s--;`
- V(s)：`s++;` 如果被P操作阻塞，重启某个在阻塞的线程并额外完成P操作

保证s非负，且P与s--，Q与s++是不可分割的

```
#include <semaphore.h>
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);/* Wrapper function for sem_wait */
void V(sem_t *s);/* Wrapper function for sem_wait */
```

使用信号量包裹不安全区的操作，就可以实现同步操作，防止进入不安全区，这是因为不安全区的信号量是负数

```
/* thread routine */
void *count(void *arg)
{
    int i;
    for (i=0; i<NITERS; i++) {
        P(&sem);
        cnt++;
        V(&sem);
    }
    return NULL;
}
```



Using Semaphores to Schedule Access to Shared Resources

两种信号量：counting semaphores和binary semaphores(mutex)；两个经典例子：The Producer-Consumer Problem和The Readers-Writers Problem

Producer-Consumer on an n-element Buffer

使用sbuf包

```c
struct {
  int *buf;     /* Buffer array */
  int n;    /* Maximum number of slots */
  int front;    /* buf[(front+1)%n] is the first item */
  int rear;   /* buf[rear%n] is the last item */
  sem_t mutex;  /* protects accesses to buf */
  sem_t slots;  /* Counts available slots */
  sem_t items;  /* Counts available items */
} sbuf_t;


void sbuf_init(sbuf_t *sp, int n)
{
    sp->buf = Calloc(n, sizeof(int));
    /* Buffer holds max of n items */
    sp->n = n;
    /* Empty buffer iff front == rear */
    sp->front = sp->rear = 0;
    /* Binary semaphore for locking */
    Sem_init(&sp->mutex, 0, 1);
    /* Initially, buf has n empty slots */
    Sem_init(&sp->slots, 0, n);
    /* Initially, buf has zero data items */
    Sem_init(&sp->items, 0, 0);
}
```

```c
void sbuf_insert(sbuf_t *sp, int item)
{
    /* Wait for available slot */
    P(&sp->slots);
    /*Lock the buffer */
    P(&sp->mutex);
    /*Insert the item */
    sp->buf[(++sp->rear)%(sp->n)] = item;
    /* Unlock the buffer */
    V(&sp->mutex);
    /* Announce available items*/
    V(&sp->items);
}


void sbuf_remove(sbuf_t *sp)
{
    int item;
    /* Wait for available item */
    P(&sp->items);
    /*Lock the buffer */
    P(&sp->mutex);
    /*Remove the item */
    item = sp->buf[(++sp->front)%(sp->n)];
    /* Unlock the buffer */
    V(&sp->mutex);
    /* Announce available slot*/
    V(&sp->slots);
    return item;
}
```

Readers-Writers Problem

读者只读取对象，写者只修改对象。第一类问题为读者优先，不让读者因写者等待；第二类问题为写者优先。

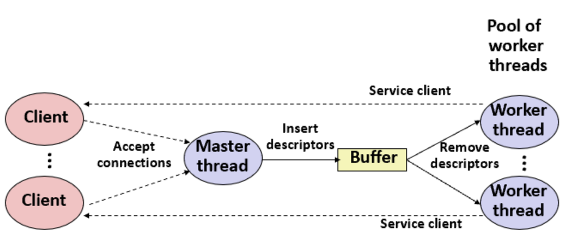
对于第一类问题，如下：

```c
int readcnt;    /* Initially 0 */
sem_t mutex, w; /* Both initially 1 */

void reader(void) {
  while (1) {
    P(&mutex);
    readcnt++;
    if (readcnt == 1) /* First in */
      P(&w);
    V(&mutex);
    /* Reading happens here */
    P(&mutex);
    readcnt--;
    if (readcnt == 0) /* Last out */
      V(&w);
    V(&mutex);
  }
}


void writer(void) {
  while (1) {
    P(&w);
    /* Writing here */
    V(&w);
  }
}
```

可能产生饥饿问题，即线程无限期阻塞。在这个问题中，读者不断过来，就会让写者无限期等待

Case Study: Prethreaded Concurrent Server

基于预线程化的并发服务器



```
#include “csapp.h”
#include “sbuf.h”
#define NTHREADS  4
#define SBUFSIZE  16

sbuf_t sbuf ; /* shared buffer of connected descriptors */

int main(int argc, char **argv)
{
    int i, listenfd, connfd;
    sockelen_t clientlen ;
    struct sockaddr_storage clientaddr;
    pthread_t tid;
    if (argc != 2) {
        fprintf(stderr, “usage: %s <port>\n”, argv[0]) ;
        exit(0);
    }
    listenfd = open_listenfd(argv[1]);
    sbuf_init(&sbuf, SBUFSIZE);
    for (i = 0; i < NTHREADS; i++)   /* Create worker threads */
        Pthread_create(&tid, NULL, thread, NULL);
    while (1) {
        clientlen =  sizeof(struct sockaddr_storage);
        connfd = Accept (listenfd, (SA *)&clientaddr, &clientlen);
        sbuf_insert(&sbuf, connfd);  /* Insert connfd in buffer */
    }
}
void *thread(void *vargp)
{
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf);  /* Remove connfd from buffer */
        echo_cnt(connfd);/* Service client */
        Close(connfd);
    }
}
```

```c
#include "csapp.h"

static int byte_cnt;/* byte counter */
static sem_t mutex;/* and the mutex that protects it */

static void init_echo_cnt(void)
{
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}
void echo_cnt(int connfd)
{
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;

    Pthread_once(&once, init_echo_cnt);
    Rio_readinitb(&rio, connfd);
    while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
        P(&mutex);
        byte_cnt += n;
        printf("server received %d(%d) byte on fd %d\n", n,byte_cnt, connfd);
        V(&mutex);
        Rio_writen(confd, buf, n);
    }
}
```

12.6 Using Threads for Parallelism

并行程序(parallel programs)时运行在多核处理器上的并发程序(concurrent programs)

以0~n-1的求和举例，划分成多个线程进行计算

```c
/* Thread routine for psum-mutex.c */
void *sum_mutex(void *vargp)
{
   int myid = *((int *)vargp);         /* Extract the thread ID */
   long start = myid * nelems_per_thread;         /* Start element index    */
   long end = start + nelems_per_thread;           /* End element index     */
   long i;
   for (i = start; i < end; i++) {
      p(&mutex) ;
      gsum += i;
      V(&mutex);
   }
   return NULL;
}
```

但效果很差，因为同步开销非常大，尽量避免或减小

```c
/* Thread routine for psum-array.c*/
void *sum_array(void *vargp)
{
   int myid = *((int *)vargp);   /* Extract the thread ID */
   long start = myid * nelems_per_thread;   /* Start element index */
   long end = start + nelems_per_thread;  /* End element index */
   long i;
   for (i = start; i < end; i++) {
      psum[myid] + = i ;
   }
   return NULL;
}
```

比psum_mutex快了几个数量级，使用私有变量来计算部分和，最后再加起来，因此规避了使用互斥锁
还可以进一步优化，即使用第五章的所学知识优化

12.7 Other Concurrency Issues

TODO