Concurrent Programming

并发编程基本概念

- 竞争：程序的正确性依赖于调度的决策
- 死锁：如 `printf` 不能用在信号处理函数中
- 活锁：冲突碰撞
- 饥饿：如高优先度进程阻塞低优先度进程

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

long fork_cnt = 0;
long handler_cnt = 0;

void sigchld_handler(int sig) {
    while (waitpid(-1, NULL, 0) > 0) {
        printf("Handler reaped a child process, total:%ld\n", ++handler_cnt);
    }
}

int main() {
    signal(SIGCHLD, sigchld_handler);while (1) {
        if (fork() == 0) {
            exit(0);
        }
        printf("Parent created a child process, total:%ld\n", ++fork_cnt);
    }
    exit(0);
}
```

基于进程的并发编程

- 服务器接收客户端的连接请求后，服务器 `fork` 出一个子进程
- 子进程关闭 `listenfd`，父进程关闭 `connfd`，避免内存泄漏
- 子进程执行完后自动关闭连接，父进程继续监听

优劣：

- 优点：简单，地址空间独立，共享状态信息
- 缺点：难共享信息，进程间通信开销高

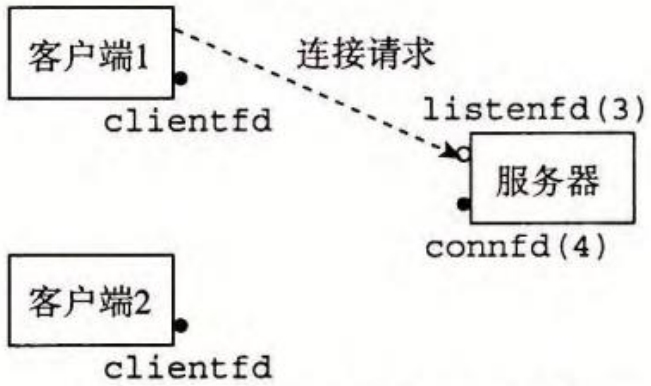

图 12-1　第一步：服务器接受客户端的连接请求

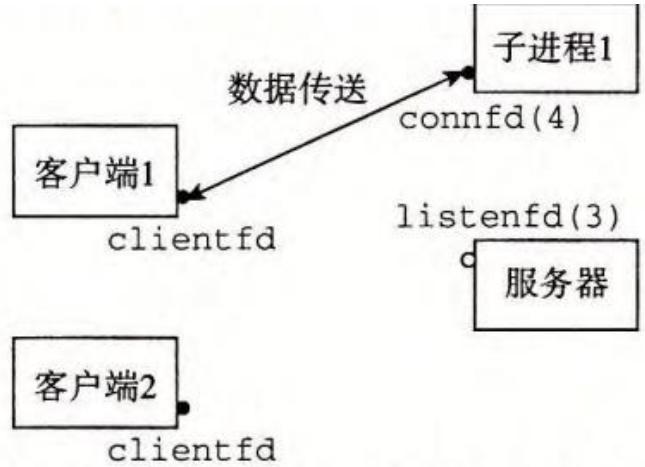

图 12-2　第二步：服务器派生一个子进程为这个客户端服务

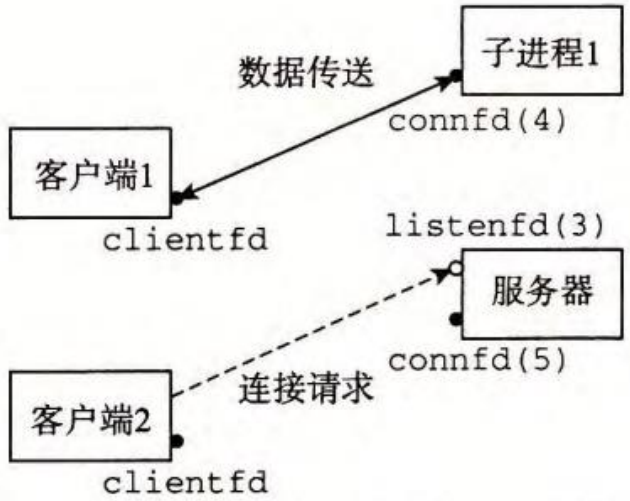

图 12-3　第三步：服务器接受另一个连接请求

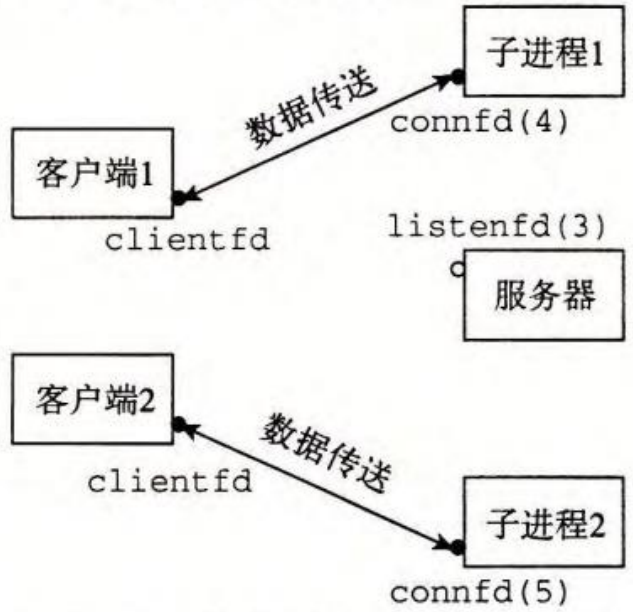

图 12-4　第四步：服务器派生另一个子进程为新的客户端服务

code/conc/echoserverp.c

```
1   #include "csapp.h"
2   void echo(int connfd);
3
4   void sigchld_handler(int sig)
5   {
6       while (waitpid(-1, 0, WNOHANG) > 0)
7           ;
8       return;
9   }
10
11  int main(int argc, char **argv)
12  {
13      int listenfd, connfd;
14      socklen_t clientlen;
15.     struct sockaddr_storage clientaddr;
16
17      if (argc != 2) {
18          fprintf(stderr, "usage: %s <port>\n", argv[0]);
19          exit(0);
20      }
21
22      Signal(SIGCHLD, sigchld_handler);
23      listenfd = Open_listenfd(argv[1]);
24      while (1) {
25          clientlen = sizeof(struct sockaddr_storage);
26          connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
27          if (Fork() == 0) {
28              Close(listenfd); /* Child closes its listening socket */
29              echo(connfd);    /* Child services client */
30              Close(connfd);   /* Child closes connection with client */
31              exit(0);         /* Child exits */
32          }
33          Close(connfd); /* Parent closes connected socket (important!) */
34      }
35  }
```

code/conc/echoserverp.c

图 12-5　基于进程的并发 echo 服务器。父进程派生一个子进程来处理每个新的连接请求

基于事件的并发编程

- I/O 多路复用的思想：同时监测若干个文件描述符是否可以执行IO操作

  当描述符准备好IO操作时再去操作
- `select` 确定要等待的描述符：*读集合*
- 状态机：等待描述符准备好、描述符准备好可以读、从描述符读一个文本行

优劣：

- 优点：一个逻辑控制流，一个地址空间，没有进程/线程管理
- 缺点：编码复杂，只能在一个核上跑

code/conc/select.c

```
1   #include "csapp.h"
2   void echo(int connfd);
3   void command(void);
4
5   int main(int argc, char **argv)
6   {
7       int listenfd, connfd;
8       socklen_t clientlen;
9       struct sockaddr_storage clientaddr;
10      fd_set read_set, ready_set;
11
12      if (argc != 2) {
13          fprintf(stderr, "usage: %s <port>\n", argv[0]);
14          exit(0);
15      }
16      listenfd = Open_listenfd(argv[1]);
17
18      FD_ZERO(&read_set);              /* Clear read set */
19      FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
20      FD_SET(listenfd, &read_set);     /* Add listenfd to read set */
21
22      while (1) {
23          ready_set = read_set;
24          Select(listenfd+1, &ready_set, NULL, NULL, NULL);
25          if (FD_ISSET(STDIN_FILENO, &ready_set))
26              command(); /* Read command line from stdin */
27          if (FD_ISSET(listenfd, &ready_set)) {
28              clientlen = sizeof(struct sockaddr_storage);
29              connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
30              echo(connfd); /* Echo client input until EOF */
31              Close(connfd);
32          }
33      }
34  }
35
36  void command(void) {
37      char buf[MAXLINE];
38      if (!Fgets(buf, MAXLINE, stdin))
39          exit(0); /* EOF */
40      printf("%s", buf); /* Process the input command */
41  }
```

code/conc/select.c

图 12-6　使用 I/O 多路复用的迭代 echo 服务器。服务器使用 select 等待监听描述符上的连接请求和标准输入上的命令

```c
#include "csapp.h"

typedef struct { /* Represents a pool of connected descriptors */
    int maxfd;        /* Largest descriptor in read_set */
    fd_set read_set;  /* Set of all active descriptors */
    fd_set ready_set; /* Subset of descriptors ready for reading  */
    int nready;       /* Number of ready descriptors from select */
    int maxi;         /* High water index into client array */
    int clientfd[FD_SETSIZE];    /* Set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* Set of active read buffers */
} pool;

int byte_cnt = 0; /* Counts total bytes received by server */

int main(int argc, char **argv)
{
    int listenfd, connfd;
    socklen_t clientlen;
    struct sockaddr_storage 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)) {
            clientlen = sizeof(struct sockaddr_storage);
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
            add_client(connfd, &pool);
        }

        /* Echo a text line from each ready connected descriptor */
        check_clients(&pool);
    }
}
```

code/conc/echoservers.c

code/conc/echoservers.c

```c
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 high water 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 received %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;
            }
        }
    }
}
```

code/conc/echoservers.c

基于线程的并发编程

线程：进程上下文中的逻辑流

- 共享代码、数据、堆、共享库和打开的文件
- 私有的线程ID、栈、栈指针、寄存器
- 和一个进程相关的线程组成一个对等线程池
- 上下文切换、创建和终止比进程快

Posix线程：

```
typedef void *func(void *)
int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg);
pthread_t pthread_self(void);
void pthread_exit(void *thread_return);
int pthread_cancel(pthread_t tid);
int pthread_join(pthread_t tid, void **thread_return);
int pthread_detach(pthread_t tid);
int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));
```

```c
#include "csapp.h"

void echo(int connfd);
void *thread(void *vargp);

int main(int argc, char **argv)
{
    int listenfd, *connfdp;
    socklen_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]);

    while (1) {
        clientlen=sizeof(struct sockaddr_storage);
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}

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