第 2 章 Web 服务器的多进程和多线程模型

2. 1 背景介绍

在上节实现了一个简单的单进程 Web 服务器。当此 Web 服务器正在处理一个用户请求的网页时，其他用户对网页的请求将被阻塞，直到处理完这个用户的请求后，才能响应其他用户的请求。这使得此 Web 服务器不能满足在短时间内满足大量用户同时请求网页的要求。本节将多进程（2.2 节）和多线程（2.3 节）模型引入到此 Web 服务器中，使得其能够并发处理大量的用户请求。

除此之外，为进一步提高 Web 服务器并发处理性能，还介绍了线程池模型（2.4 节）、业务分割模型（2.5 节）和混合模型（2.6 节）。其中，线程池模型利用“池”思想来缓存和复用线程，以避免因为线程的大量重复创建和销毁所带来的性能损失；业务分割模型通过将业务流程分解为更小粒度的操作单元，以减少阻塞等待时间，来提高系统的并发性能；混合模型利用进程和线程各自不同的特性，将多进程和多线程模型混合在一起，在提高系统性能基础上保证了系统的健壮性。

2. 2 进程模型

2.2.1 Linux 中进程创建相关函数

进程是程序的一次执行过程。程序在执行过程中，操作系统要为其分配内存空间、CPU 和 I/O 等计算机资源。操作系统为方便管理程序运行中所需的计算机资源，将与程序运行相关的计算机资源抽象为进程。因为程序一次执行对应着一个进程，而进程里面有程序运行所需的资源，因此操作系统可以通过对进程的管理，来掌握每个程序的执行状态和运行过程。例如程序 A 在运行时需要用内存 200MB，并且需要占用 socket 通信接口来进行通信，那么在程序 A 运行时，操作系统会为程序 A 分配这些资源，并将资源相关信息记入这次运行的进程中。当有多个程序需要并发运行时，操作系统为每个运行的程序分配计算机资源，并计入它们当前运行的进程中。

当操作系统中存在多个进程时，操作系统会为它们合理地安排计算机资源，以提高它们并发运行的效率。因此进程是操作系统资源分配和调度的基本单位。

Linux 为创建进程和使用进程提供了如下接口。

- fork 函数

程序在执行 fork 函数后，linux 会创建子进程，子进程和父进程共享程序 fork 函数后面执行的代码。而且在创建子进程时，linux 系统会将父进程内的几乎所有的资源复制给子进程，这样子进程就能够共享父进程已经获得的资源了。为提高效率，fork 函数采取了写时复制技术，当父子进程空间中的内容发生变化时，才将变化内容所在内存段复制一份给子进程，否则两个进程将会共享内容空间。写时复制技术极大地减少了不必要的数据复制过程。

既然在执行 fork 函数后，父子进程会共享后面的程序代码，那么该如何让这两个进程执行不同的运行代码呢？这就需要根据 fork 函数的返回值来确定。父进程中获得的此函数值为子进程的 PID，子进程获得的应该为 0。如果创建子进程失败，父进程得到的 PID 为-1。具体运行逻辑见如下代码。

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

int main(int argc,char *argv[]){
 pid_t pid=fork();
 if(pid==0)
        printf("This is a child process\n");     //child process executes the line
 else   //parent process executes the following line
        printf("This is a parent process, and its child process id is %d\n", pid);
 return 0;   //parent and child processes both execute the line
}
```

- **exec 系列函数**

exec 系列函数包括 execl、execlp、execle、execv 和 execvp。它们仅是调用参数不同，但具有相同的语义。它们都是用于将当前进程替换为一个新的进程，并且这个新的进程与被替换的进程具有相同的 PID。它们的具体函数参数格式请查阅相关的函数手册，在这里仅以 execl 函数来说明其用途。例如下面的代码，将子进程替换为一个“ls”程序进程。在子进程开始执行时将打印出“entering child process”，但是当执行 execl 函数后，“existing child process”将不会被打印，这是因为该子进程已经被执行“ls”的进程替换。

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

int main(int argc,char *argv[]){
 pid_t pid=fork();
 if(pid==0){
        printf("entering child process\n");
        execl("/bin/ls","ls","-l",NULL);
        printf("existing child process\n");
 }
 else
        printf("This is a parent process, and its child process id is %d\n", pid);
 return 0;
}
```

- **vfork 函数**

vfork 函数在创建子进程后，子进程与父进程共享空间，这样在子进程修改的变量数据，父进程也能够看得到。但是由于两个进程共享空间，很容易导致相互破坏运行堆栈。一般会在 vfork 子进程函数中使用 exec 函数来替换自己程中的内容。

2.2.2 Linux 中进程通信相关函数

进程间通信指的是两个或多个进程之间信息的共享，其主要方式包含管道、共享内存、消息队列、信号量、网络通信、文件等内容。具体函数如下。

- **管道**

管道是用于进程间通信的一种特殊文件，进程间通过对这个特殊文件的读写来完成信息的传递。管道一般是半双工的，即数据流只有一个方向。管道分为：匿名管道和命名管道。

两者的用法有所不同。其中匿名的管道主要用于亲缘性进程之间（比如，父子进程和兄弟进程之间）；而命名管道并无上述限制，可以在无关进程之间进行数据通信。

函数：int pipe(int fd[2])

用来创建匿名管道，其中参数 fd[2]为对此管道的读写操作描述符，其中 fd[0]为读描述符；fd[1]为写描述符。在用于父子进程通信时，因为通道为半双工的，所以在父子进程中要关闭相关的描述符，来获得不同的数据流向。比如，在父进程中关闭 fd[0]描述符，而在子进程中关闭 fd[1]描述符，则父进程负责向管道写数据、子进程从管道读数据；如果父子进程关闭的描述符相反，则数据流向也相反。具体父进程向子进程传递消息的例子代码如下所示。

```
#include <stdio.h>
#include <unistd.h>
int main()
{
 int fd[2];  // 两个文件描述符
 pid_t pid;
 char readbuf[20];
       char* writebuf="pipe message";
 if(pipe(fd) < 0)  // 创建管道
        printf("create pipe error!\n");

 if((pid = fork()) < 0)  // 创建子进程
        printf("fork error!\n");
 else if(pid > 0)  // 父进程
 {
        close(fd[0]); // 关闭读描述符
        write(fd[1], writebuf, strlen(writebuf));
 }
 else                //子进程
 {
        close(fd[1]); // 关闭写描述符
        read(fd[0], readbuf, 20);
        printf("%s", readbuf);
 }

 return 0;
}
```

函数：int mkfifo(const char *pathname, mode_t mode)

用来创建命名管道。此命名管道是文件系统中一个特殊设备文件。通过参数 pathname 来指定这个命名管道文件的名称；通过 mode 参数来制定这个文件的权限。当创建好这个文件后，可以使用文件操作函数 open 来打开这个文件，以进行读写操作。与普通文件的打开操作模式略有不同，其不包含 O_RDWR 模式（读写模式）。其中 open 函数中的 flag 参数可以读、写、阻塞和非阻塞。具体有以下四种模式：

O_RDONLY：open 将会调用阻塞，除非有另外一个进程以写的方式打开同一个 FIFO，否则一直等待。

O_WRONLY：open 将会调用阻塞，除非有另外一个进程以读的方式打开同一个 FIFO，否则一直等待。

O_RDONLY | O_NONBLOCK：非阻塞方式只读方式打开文件，无论是否有其它进程以写的方式打开此管道文件，open 均会成功返回，此时 FIFO 被读打开。

O_WRONLY|O_NONBLOCK：非阻塞方式只写方式打开文件，如果此时没有其它进程以读的方式打开，open 会失败打开，此时 FIFO 没有被打开，返回-1。

下面为应用命名管道来实现生产者和消费者问题的代码。

```
//生产者进程,produce.c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/stat.h>

int main(){
 int fd;
 int n, i;
 char* buf= "fifo example";
 time_t tp;
 // 以写打开一个 FIFO,如果此 FIFO 读进程不存在，则阻塞。
 if((fd = open("fifo-example", O_WRONLY)) < 0) {
     perror("open fifo-example failed");
     exit(1);
 }
 printf("send message: %s to fifo-example file", buf);
 if(write(fd, buf, strlen(buf)) < 0) { // 将 Buf 写入到 FIFO 中
     perror("write fifo failed");
     close(fd);
     exit(1);

 }
 close(fd);  // 关闭此 fifo-example 文件
 return 0;
}
```

```
//消费者进程 consumer.c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <sys/stat.h>

int main()
{
 int fd;
 int len;
 char buf[1024];

 if(mkfifo("fifo-example", 0777) < 0 ) {// 创建 FIFO 管道,
       printf("Please check the file must not exist in the directory\n");
       perror("Create FIFO Failed");
 }

 if((fd = open("fifo1", O_RDONLY)) < 0)  // 以读打开 FIFO
 {
       perror("Open FIFO Failed");
       exit(1);
 }

 while((len = read(fd, buf, 1024)) > 0) // 读取 FIFO 管道
       printf("Read message: %s from the fifo-example pipe", buf);

 close(fd);  // 关闭 FIFO 文件
 return 0;
}
```

- **消息队列**

消息队列就是操作系统内核中存放消息的队列。消息队列中的数据并不依赖于具体的进程而存在。消息队列可以很方便地作为多个进程之间交换数据的场所。消息队列与管道相比，其可提供格式化数据的存储，而管道只能提供流式文件存储。linux 支持 POSIX 和 System V 接口格式的消息队列，两种差别并不是太大。本书将以 POSIX 接口来介绍消息队列的使用。POSIX 小的队列的操作函数如下。

mqd_t mq_open(const char *name, int oflag);

mqd_t mq_open(const char *name, int oflag, mode_t mode, struct mq_attr *attr);

为创建或获取一个消息队列，其中 name 为消息队列名字；oflag 为打开队列的模式，包含 O_RDONLY、O_WRONLY、O_RWRD、O_CREATE、O_EXCL、O_NONBLOCK 等模式；在 oflag 指定为 O_CREATE 时，mode 参数指定消息队列的权限；attr 为消息队列的属性信息，如果其取值为 NULL，则会按默认值配置消息队列。其具体数据结构如下。

```
struct mq_attr {
    long mq_flags;       /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;      /* Max. # of messages on queue */
    long mq_msgsize;     /* Max. message size (bytes) */
    long mq_curmsgs;     /* # of messages currently in queue */
};
```

int mq_setattr(mqd_t mqdes, const struct mq_attr *newattr, struct mq_attr *oldattr)和

int mq_getattr(mqd_t mqdes, struct mq_attr *attr),

用于设置或者获取指定消息队列的属性。

int mq_send(mqd_t mqdes, const char *msg_ptr,size_t msg_len, unsigned int msg_prio);

int mq_timedsend(mqd_t mqdes, const char *msg_ptr,size_t msg_len, unsigned int msg_prio,const struct timespec *abs_timeout);

用于发送消息到指定的消息队列。其中参数 mqdes 为消息队列描述符，为 mq_open 所创建；msg_ptr 和 msg_len 分别是消息的指针和长度；msg_prio 用于指定消息的优先级。如果消息队列满了就阻塞，新的消息必须等到消息队列中有空间才能进入。如果在创建或打开消息队列时 oflag 中 O_NOBLOCK 选项，则会报错。如果想让消息队列满后，只等待有限的时间，则可以使用 abs_timeout 来进行设置等待的时间。

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio)和

ssize_t mq_timedreceive(mqd_t mqdes, char *msg_ptr, size_t msg_len, unsigned int *msg_prio, const struct timespec *abs_timeout)；

用于从消息队列里面接受消息。

int mq_notify(mqd_t mqdes, const struct sigevent *sevp)

向消息队列建立或删除消息通知事件。具体应用请查阅相关文献。

int mq_close(mqd_t mqdes)

关闭消息队列。

int mq_unlink(const char *name)

移除指定的消息队列。因为消息队列并不依附于进程而存在，因此在使用消息队列的最后一个进程关闭消息队列后，需使用 mq_unlink 函数以给操作系统内核明确的信号来删除此消息队列，否则将会造成系统资源的浪费。

使用消息队列的生产者和消费者问题的实现代码如下。

```
//头文件 book.h
struct book
{
    char name[36];
    int id;
};
//生产者进程 produce.c
#include <stdio.h>
#include <stdlib.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <sys/types.h>
#include <unistd.h>
#include <errno.h>
#include <mqueue.h>
#include <fcntl.h>
#include <sys/stat.h>
int main()
{
    //创建一个消息队列，并且为只写
    mqd_t mqid = mq_open("/examplemq", O_WRONLY|O_CREAT, 0777, NULL);
    if (mqid == -1)
        err_exit("mq_open error");

    struct book bk = {"webbook", 11};
    unsigned prio = 2;
    //向此消息队列发送 book 实例消息
    if (mq_send(mqid, (const char *)&bk, sizeof(bk), prio) == -1)
        err_exit("mq_send error");
    //关闭这个消息队列
    mq_close(mqid);
    return 0;
}
//消费者进程 consumer.c
int main()
{
 //以只读的方式打开一个消息队列
    mqd_t mqid = mq_open("/examplemq", O_RDONLY);
    if (mqid == -1)
        err_exit("mq_open error");

    struct book bk;
    int itcv;
    unsigned prio;
    struct mq_attr attr;
    //获得消息队列的属性信息
    if (mq_getattr(mqid, &attr) == -1)
        err_exit("mq_getattr error");
    //从消息队列里面接受 book 信息
    if ((itcv = mq_receive(mqid, (char *)&bk, attr.mq_msgsize, &prio)) == -1)
        err_exit("mq_receive error");
```

```
    printf("receive book message from mq %d\n", itcv);
    printf("The book's id is %d and its name is %s\n", bk.id,bk.name);
        //关闭消息队列
        mq_close(mqid);
        //删除此消息队列
        mq_unlink(mqid);
        return 0;
    }
```

- **信号量**

信号量表示可用资源的数量，其通过 P、V 操作来完成可用资源量的计数更新，从而达到进程间同步和互斥的目的。同样 linux 下的信号量函数的操作函数也对应着两种：POSIX 和 System V。本文将主要对 POSIX 接口的信号量进行介绍。在 POSIX 中将信号量分为无名信号量（unname semaphore）和有名信号量（named semaphore）两种。无名信号量是基于内存的信号量，如果其不放入进程共享内存区，则无法在进程间使用，仅能在同一进程的多线程中使用。而有名信号量可以提供进程间的操作。

无名信号量使用 sem_init 函数进行初始化，使用 sem_destory 函数进行销毁；有名信号量使用 sem_open 函数进行初始化，使用 sem_close 函数来关闭信号量资源，使用 sem_unlink 函数来销毁进行间信号量资源。其余信号量的操作函数为两种信息量的公用函数。

sem_t *sem_open(const char *name, int oflag);

sem_t *sem_open(const char *name, int oflag,mode_t mode, unsigned int value);

有名信号量的打开或创建函数。用于进程间同步和互斥操作。

int sem_close(sem_t *sem);

int sem_unlink(const char *name);

有名信号量的关闭和销毁函数。

int sem_init(sem_t *sem, int shared, unsigned int value);

int sem_destroy(sem_t *sem);

无名信号量的初始化和销毁函数，主要用在多线程环境中。

int sem_wait(sem_t *sem);

相当于信号量的 P 操作。其测试所指定信号量的值，大于 0，将它减 1 并返回，等于 0，调用进程或线程休眠，直到该值大于 0，将它减 1，函数随后返回。

int sem_trywait(sem_t *sem);

相当于信号量的 P 操作。但是其与上面的 sem_wait 不同的是，当所指定信号量值为 0 时，其上的进程并不休眠，而是返回一个 EAGAIN 错误。

int sem_post(sem_t *sem);

相当于信号量的 V 操作。

int sem_getvalue(sem_t *sem, int *valp);

此函数通过参数 valp 返回指定信号量中的当前数值

- **共享内存**

共享内存指多个进程之间共享的内存区域，这块内存区域可以被多个进程访问。在 POSIX 标准中，共享内存对象可以通过以下函数来实现。

int shm_open(const char *name, int oflag, mode_t mode)；

int shm_unlink(const char *name);

其中，shm_open 用于创建内存或打开一个共享内存区域对象；shm_unlink 为删除一个共享内存对象。要实现多个进程之间共享内存区域，还需要 mmap 函数来配合使用。

void *mmap(void *start, size_t len, int prot, int flags, int fd, off_t offset)

该函数的主要作用是将文件或设备映射到调用进程空间中。当文件被映射到进程空间后，可以通过该进程的虚拟地址读写来完成文件的读写操作，能够加快文件操作的 I/O 速度。该函数提供对共享内存对象进行进程内存映射。其中，

start 为被映射内容在进程空间内的起始地址，如果为 NULL，则让内核自动为其选择起始地址；

len 映射到进程地址空间的字节数；prot 为内存映射区域的读写操作保护标志位，由 PROT_READ、PROT_WRITE、PROT_EXEC 和 PROT_NONE 等值组合而成，可以分别表示该内存区域的读、写和执行权限；

flags 表示映射内存类型，如果为 MAP_SHARD，则表示在映射内存区域内修改的数据对所有能访问该内存区域的进程可见，如果为 MAP_PRIVATE，则该内存区域的修改数据仅能被修改该区域的进程所见，其它进程看不到被修改的数据；

fd 为文件、设备或共享内存区域对象描述符；

offset 为当前文件、设备或共享内存区域对象的偏移位置。

通过 munmap(void *start, size_t len)函数可以从进程地址空间中删除一个映射。

下面的例子代码通过使用共享内存和信号量来完成父子进程协作计数的功能。父子进程互斥地对共享内存中据进行修改以达到计数目的。

```
    //semaphoreposix.c 文件
    //使用下面的命令来编译下面代码
    // gcc -std=gnu99 -Wall -g -o semaphoreposix semaphoreposix.c -lrt -lpthread
    #include <sys/mman.h>
    #include <semaphore.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <unistd.h>
    #include <fcntl.h>
    #include <sys/stat.h>
    #include <sys/types.h>
    #include <signal.h>

    #define NUM 100
    #define SEM_NAME "sem_example"
    #define SHM_NAME "mmap_example"

    int main(){

     int count=0;
     sem_t* psem;

     //创建信号量,初始信号量为 1
     if((psem=sem_open(SEM_NAME, O_CREAT,0666, 1))==SEM_FAILED){
         perror("create semaphore error");
         exit(1);
     }
     int shm_fd;
     //创建共享内存对象
     if((shm_fd=shm_open(SHM_NAME,O_RDWR| O_CREAT,0666)) < 0){
         perror("create shared memory object error");
         exit(1);
     }
     /* 配置共享内存段大小*/
         ftruncate(shm_fd, sizeof(int));
     //将共享内存对象映射到进程
     void * memPtr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED,
shm_fd, 0);

     if(memPtr==MAP_FAILED){
```

```
        perror("create mmap error");
        exit(1);
    }
    //为此内存区域赋值
        * (int *) memPtr= count;
        //创建子进程
    pid_t pid=fork();
    if (pid==0)  //child process
    {

        for (int i = 0; i < NUM; ++i)
        {     //信号量实现的临界区
            sem_wait(psem);
            printf("Child Process count value: %d\n", (*(int *) memPtr)++);
            sem_post(psem);
        }
    }
    else if (pid > 0){ // parent process
        for (int i = 0; i < NUM; ++i)
        {    //信号量实现的临界区
            sem_wait(psem);
            printf("Parent Process count value: %d\n", (* (int *)memPtr)++);
            sem_post(psem);
        }
        sleep(1);
        //卸载各种资源
        if (munmap(memPtr, sizeof(int)) == -1) {
            perror("unmap failed");
            exit(1);
            }
            if (close(shm_fd) == -1) {
            perror("close shm failed");
            exit(1);
            }
            if (shm_unlink(SHM_NAME) == -1) {
            perror("shm_unlink error ");
            exit(1);
            }
            if(sem_close(psem)==-1){
                perror("close sem error");
                exit(1);
            }
        if (sem_unlink(SEM_NAME)==-1) {
                perror("sem_unlink error");
                exit(1);
            }
    }else{
        perror("create childProcess error");
        exit(1);
    }

    exit(0);
   }
```

- **网络通信**

可以通过前一章节中的 socket 接口实现两个进程之间通信，这两个进程可以在同一主机上，也可以在不同主机上（不同主机之间有网络连接）

2.2.3 多进程 Web 服务器模型

在实验 1 中，实现了一个基本的 Web 服务器。这个 Web 服务器是单进程模型的，当服务器接受到客户端请求，就建立一个网络连接，并从此连接解析请求的文件；然后从文件系统中读取这个文件到缓存，最后通过这个网络连接将缓存中的内容发送到客户端。在 Web 服务器处理这个客户端的上述步骤内容时，如果有其它客户端也请求连接 Web 服务器，则其它客户端的请求连接将被阻塞，直到 Web 服务器完成这个客户端的所有业务处理，才会从其它客户端连接中选择一个再进行上述步骤处理。如下图 2-1 所示，Client B 和其它的 Client 都被阻塞在 Web 服务器的 accept 函数，而当前 Web 服务器正在处理 Client A 的文件请求，也就是其它客户端要等待 Client A 的请求被处理完才能依次被处理。很明显，这样的设计使得 Web 服务器的并发处理客户请求的能力比较弱。

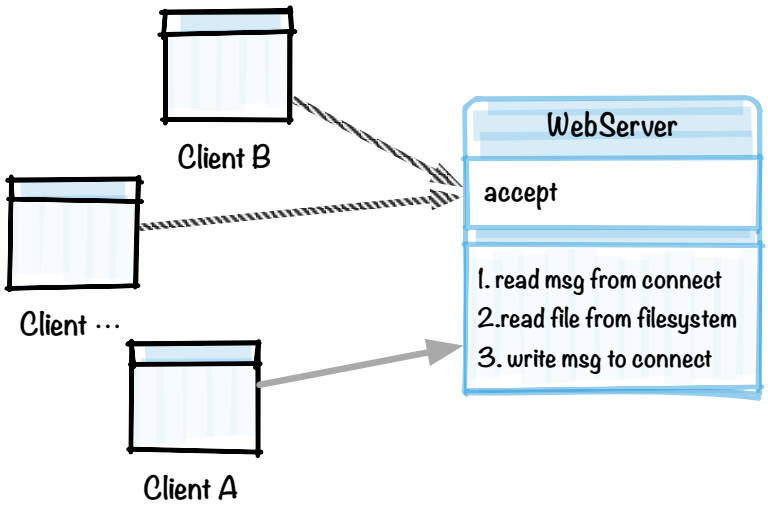

图 2-1 Web 服务器串行处理多用户的同时请求

如果 WebServer 在处理客户端连接中请求的文件同时，也能够接受其它客户端的连接请求处理，则将会提高 Web 服务器的并发处理能力。如果采用多进程模型，则 WebServer 在接收到客户端连接请求后，就创建一个子进程，在这个子进程中进行客户端的文件请求处理。如果多个客户端同时请求连接，则会创建多个子进程。其中，每个子进程都会处理一个客户端的请求。这样就使得 WebServer 能够在一个时间段内能够同时处理多个客户端请求，大大增加了其并发处理能力。

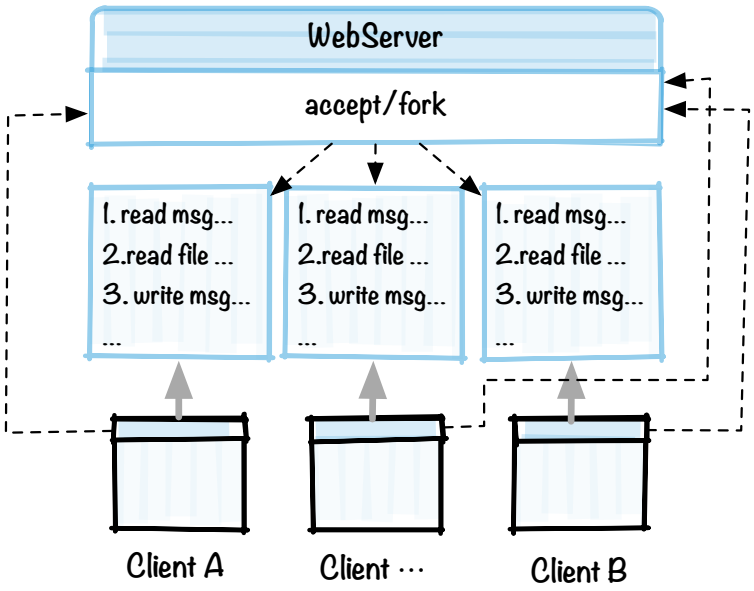

图 2-2 Web 服务器多进程模型

具体过程如下图 2-2 所示，每个客户端请求连接 WebServer 指定的侦听接口（虚线箭头），在 WebServer 侦听到连接请求后，将会与客户端建立连接通道（实线箭头），同时使用 fork 创建子进程。在这个子进程中处理这个连接通道，而父进程会马上返回到 accept 函数，继续等待新的客户端的连接请求。这样就使得 WebServer 为每个客户端创建一个进程，来处理其对文件的请求。

2.2.4 实验 2 Web 服务器的多进程模型实现

根据上节对 WebServer 多进程模型的描述，多进程相关函数，来完成如下题目。

题目 1：使用 fork 函数，设计并实现 WebServer 以支持多进程并发处理众多客户端的请求。

题目 2：使用信号量、共享内存等系统接口函数，来统计每个子进程的消耗时间以及所有子进程消耗时间之和。

题目 3：使用 http_load 来测试当前设计的多进程 WebServer 服务性能，根据测试结果来分析其比单进程 Web 服务性能提高的原因。同时结合题目 2，来分析当前多进程 WebServer 的性能瓶颈在何处？是否还能够继续提高此 WebServer 服务的性能？

2. 3 线程模型

2.3.1 Linux 线程模型

线程负责具体程序逻辑的执行，是处理器调度的基本单位。与进程相比，线程不具有独立的地址空间，可以与进程内的其它线程共享进程的资源。因此线程具有容易共享信息、调度切换开销小等特点。在现代操作系统中将线程分为用户线程、内核线程和 LWP(Light Weight Process-轻量级进程)三种类型。

- **用户线程**

用户线程由线程库在用户空间内创建、调度、同步和管理。由于用户线程由线程库来管理，它们之间调度切换并不需要进行系统调用，因此切换开销小。

但是由于操作系统内核并不知道用户线程的存在，内核仅是以用户线程所在的进程为单位来进行处理器调度，导致此线程内的所有用户线程只能共享一个处理器资源，不能充分地利用多处理器；当一个用户线程进行系统调用而导致阻塞时，操作系统内核将阻塞其所在的进程，因此此进程的其它用户线程也不能运行。

- **内核线程**

内核线程是由操作系统内核来创建、调度和管理的线程。内核线程是操作系统调度的基本单位，这些内核线程在操作系统进程内竞争系统资源，如果有一个内核线程处于阻塞状态，并不影响其它内核线程的调度和运行。由于内核线程之间切换需要进行系统调用（用户态到系统态之间相互转换），因此切换开销较大。

- **LWP**

LWP（轻量级进程）是一种由内核支持的用户线程。它是基于内核线程的抽象，是用户线程与内核线程之间的桥梁。一个 LWP 与一个内核线程相对应，因此操作系统内核是能够识别和调度 LWP。将用户线程绑定到 LWP 后，LWP 可以被看为用户线程的虚拟处理器。

如果一个用户线程与一个内核线程相对应，则为“一对一”模型；如果多个用户线程与一个内核线程相对应，则为“多对一”模型；如果多个用户线程与多个内核线程对应，则为“多对多”模型。每个操作系统提供了不同的线程对应模型。

在目前的 Linux 中默认的 POSIX 线程模型采用的是“一对一”模型，也就是一个用户线程对应一个内核线程，其通过线程创建函数创建的线程是 LWP 类型的。

2.3.2 POSIX 线程库接口

Linux 提供了兼容 POSIX 标准的线程操作 API，其主要的函数如下所示。

- **int pthread_create(thread_t* thread, pthread_attr_t* attr, void* (start_routine)(void*), void* arg)**

线程创建函数，其中参数 thread 是创建好线程的指针，用于后续的线程操作；atrr 为线程属性指针，如果为 NULL，将按默认属性来创建线程；start_routine 为完成线程逻辑功能的函数指针；arg 为向线程传递参数的指针。如果成功返回 0，失败返回-1。

- **void pthread_exit(void* retval)**

退出当前线程函数，参数 retval 用来返回当前函数的退出值。

- **int pthread_cancel(pthread_t thread)**

向目标线程发送请求终止（cancel）信号，其中参数 thread 为要被取消运行的线程 id。当然调用 pthread_cancel 并不意味着目标的线程一定要被终止，而是目标线程接收到 cancel 信息后，它自己决定如何来响应这个信号：忽略这个信号、立即退出、运行至取消点（cancelation-point）后再退出。

在此函数发出 cancel 信号后，目标线程的 cancel state 来决定是否接受此 cancel 信号，如果 cancel state 是 PTHREAD_CANCEL_ENABLE（默认）则接收信号；如果是 PTHREAD_CANCEL_DISABLE，则不接收此信号。对 cancel state 的设置使用“int pthread_setcancelstate(int state, int *oldstate)”函数，其中参数 state 可以设置为上述两种状态之一。

当目标线程接收到 cancel state 信号，目标线程的 cancel type 来决定何时取消。如果 cancel type 是 PTHREAD_CANCEL_DEFERRED（默认），目标线程并不会马上取消，而是在执行下一条 cancellation point 的时候才会取消；如果 cancel type 是

PTHREAD_CANCEL_ASYNCHRONOUS，目标线程会立即取消。对 cancel type 的设置使用“int pthread_setcanceltype(int type, int *oldtype) ”函数，其中参数 type 为上述两种数值之一。

而 cancellation point 为调用 POSIX 库中 pthread_join、pthread_testcancel、pthread_cond_wait、pthread_cond_timedwait、sem_wait、sigwait 以及 read、write 等函数的位置。

- **int pthread_join(pthread_t* tid, void** thread_return)**

等待线程结束函数，其中参数 tid 为被等待的线程 id 指针；thread_return 为被等待线程的返回值，也就是 pthread_exit 里面的参数值。如果当前线程调用此函数，将会阻塞，直到被等待线程运行结束后或者被其它线程取消运行，当前线程才会继续运行。另外需要注意的是，一个线程不能被多个线程等待，否则除第一个等待线程外，其它等待线程均会返回错误值 ESRCH。

下面为 linux 手册中关于 pthread 的例子代码，演示了上述函数的使用。

```
#include <pthread.h>
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <unistd.h>

#define handle_error_en(en, msg) \
        do { errno = en; perror(msg); exit(EXIT_FAILURE); } while (0)
//进程函数
static void * thread_func(void *ignored_argument)
{
    int s;

    /* 将线程中 cancel state 暂时改为不接收 cancel 信号，以不响应其它进程向它发出的 cancel 信号
*/
        s = pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_setcancelstate");
    printf("thread_func(): started; cancellation disabled\n");
    sleep(5);
    printf("thread_func(): about to enable cancellation\n");
    /* 恢复线程中接收 cancel 信号的状态*/
    s = pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_setcancelstate");

    /* sleep 函数是一个 cancellation point */
    sleep(1000);        /* Should get canceled while we sleep */
    /* 下面的代码在正常情况下应该不会被执行*/
    printf("thread_func(): not canceled!\n");
    return NULL;
}
int main(void)
{
    pthread_t thr;
    void *res;
    int s;

    /* 创建一个线程，并向它发生 cancel 信号 */
    s = pthread_create(&thr, NULL, &thread_func, NULL);
    if (s != 0)
        handle_error_en(s, "pthread_create");

    sleep(2);           /* 给这个线程留出足够多的时间启动 */

    printf("main(): sending cancellation request\n");
```

```
    s = pthread_cancel(thr);
    if (s != 0)
        handle_error_en(s, "pthread_cancel");

    /* 调用 pthread_join 函数，并查看目标线程的退出状态*/
    s = pthread_join(thr, &res);
    if (s != 0)
        handle_error_en(s, "pthread_cancel");

    /* 调用 pthread_join 函数，并查看目标线程的退出状态*/
    s = pthread_join(thr, &res);
    if (s != 0)
        handle_error_en(s, "pthread_join");

    if (res == PTHREAD_CANCELED)
        printf("main(): thread was canceled\n");
    else
        printf("main(): thread wasn't canceled (shouldn't happen!)\n");
    exit(EXIT_SUCCESS);
}
```

这段代码正常执行将打印出如下信息。

```
thread_func(): started; cancellation disabled
main(): sending cancellation request
thread_func(): about to enable cancellation
main(): thread was canceled
```

对与 pthread 的属性设置，提供了如下的接口函数。

- int pthread_attr_init (pthread_attr_t* attr)

属性初始化函数，创建一个线程属性结构，并通过 attr 指向此结构。

- `int pthread_attr_destroy(pthread_attr_t *attr);`

销毁一个线程属性结构。

- int pthread_attr_setscope (pthread_attr_t* attr, int scope);

设置线程的作用域，在 POSIX 标准中参数 scope 可以取 PTHREAD_SCOPE_SYSTEM 和 PTHREAD_SCOPE_PROCESS。这两个值分别表示线程调度范围是在系统中还是在进程中。但是在 Linux 中仅支持 PTHREAD_SCOPE_SYSTEM（也就是“一对一”模型），如果 scope 设置 PTHREAD_SCOPE_PROCESS，则会报错，返回 ENOTSUP。

- int pthread_attr_setdetachstate (pthread_attr_t* attr, int detachstate)

设置分离属性。创建的线程分为分离和非分离状态。如果线程为分离状态(detachstate 取值为 PTHREAD_CREATE_DETACHED)，则线程在运行完就自行结束并释放资源；如果线程为非分离状态（detachstate 取值为 PTHREAD_CREATE_JOINABLE），则此线程需要等待它的线程中 pthread_joint 函数返回后，才终止并释放资源。线程默认为非分离状态。如果将线程状态设置为分离状态，需要注意的是创建的线程运行可能非常快，在 pthread_create 函数没有返回时已经运行结束，这时 pthread_create 中可能会得到操作的线程号。为了避免这个问题，可以在创建线程里调用 pthread_cond_timewait 函数，让线程等待一会。

- int pthread_attr_setaffinity_np(pthread_attr_t *attr, size_t cpusetsize, const

cpu_set_t *cpuset)

设置线程的 CPU 亲缘性。在多 CPU 环境下，如果设置一个线程在一个指定 CPU 运行，则需要调用此函数。如下代码将指定线程运行在 0 号 CPU。

```
pthread_attr_t attr1;
pthread_attr_init(&attr1);
cpu_set_t cpu_info;
__CPU_ZERO(&cpu_info);
__CPU_SET(0, &cpu_info);
pthread_attr_setaffinity_np(&attr1, sizeof(cpu_set_t), &cpu_info)
```

- int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy)

设置线程的调度策略。policy 可以取值 SCHED_FIFO、SCHED_RR 和 SCHED_OTHER。其中 SCHED_OTHER 为默认的分时调度策略，表示线程一旦开始运行，直到时间片运行完或者阻塞或者运行结束才让出 CPU 控制权，此状态下不支持线程的优先级。SCHED_FIFO 为实时调度，执行先来先服务的调度策略，一个线程一旦占有 CPU，则运行到阻塞或者有更高优先级的线程到来。SCHED_RR 为实时调度，执行时间片轮转调度。

- int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param)

设置线程的优先级。其中 sched_param 结构中仅有属性 sched_priority，其用来设置线程的优先级。线程优先级可以取 1-99 中任意个数字，数值越大优先级越高。

```
pthread_attr_t attr;
struct sched_param param;
pthread_attr_init(&attr);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
param.sched_priority = 50;
pthread_attr_setschedparam(&attr,&param);
```

2.3.3 Linux 线程间同步与互斥

线程除了可以使用进程间通信函数来实现同步与互斥外，在 Linux 系统中 POSIX 库中还提供一些列接口函数用于线程间的同步。

- **信号量**

线程间同步和互斥还可以使用无名信号量来实现。有关无名信号量的创建、操作和销毁函数见进程模型。

- **互斥量操作**

互斥量可以用来实现临近区，让线程互斥地使用临界资源。在 linux 的 POSIX 库中，提供 pthread_mutex_init、pthread_mutex_lock、pthread_mutex_trylock、pthread_mutex_unlock 和 pthread_mutex_destroy 等函数来完成互斥量的初始化、加锁、释放、摧毁等操作。其中 pthread_mutex_trylock 为非阻塞函数，如果互斥量没有被锁住，其对互斥量加锁，并进入临界区；如果互斥量已经加锁，则返回 EBUSY，而不会阻塞。pthread_mutex_lock 为阻塞函数，如果已经有其它线程占有互斥量，则阻塞直到获得这个互斥量为止。

- **读写锁**

对于读写者问题（多个读线程能同时读取数据，只有写线程写入数据时才会阻塞其它线程），POSIX 库提供了 pthread_rwlock_init 和 pthread_rwlock_destroy 函数用来创建和销毁读写锁；pthread_rwlock_rdlock、pthread_rwlock_wrlock 和 pthread_rwlock_timedrdlock 函数使用阻塞的方式来获得读锁或者写锁；pthread_rwlock_tryrdlock 和 pthread_rwlock_trywrlock

函数使用非阻塞方式来获得读锁或写锁； pthread_rwlock_unlock 函数释放读写锁。具体用法，如下代码所示。

```
# 编译命令：gcc -std=gnu99  -o readerwriter readerwriter.c  -lpthread
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>

pthread_rwlock_t rwlock;  //读写锁
int num=0;

//读线程函数
void * reader1(){
 for (int i = 0; i < 10; ++i)
 {
        pthread_rwlock_rdlock(&rwlock);
        printf("reader1 reads %d times num = %d\n", i,num);
        pthread_rwlock_unlock(&rwlock);
        sleep(1);
 }
}

void * reader2(){
 for (int i = 0; i < 10; ++i)
 {
        pthread_rwlock_rdlock(&rwlock);
        printf("reader2 reads %d times num = %d\n", i,num);
        pthread_rwlock_unlock(&rwlock);
        sleep(1);
 }
}

//写线程
void * writer1(){
 for (int i = 0; i < 10; ++i)
 {
        pthread_rwlock_wrlock(&rwlock);
        num++;
        printf("writer1 writes %d times num=%d\n",i,num);
        pthread_rwlock_unlock(&rwlock);
        sleep(1);
 }

}
int main(int argc, char const *argv[])
{
 pthread_t thr1,thr2,thw1;  //读线程、写线程

 pthread_rwlock_init(&rwlock,NULL); //初始化读写锁
 //rwlock=PTHREAD_RWLOCK_INITIALIZER; //使用宏来初始化读写锁
 //创建读写线程
 pthread_create(&thr1,NULL,reader1,NULL);
 pthread_create(&thr2,NULL,reader2,NULL);
 pthread_create(&thw1,NULL,writer1,NULL);
 //等待线程结束回收资源
 pthread_join(thr1,NULL);
 pthread_join(thr2,NULL);
 pthread_join(thw1,NULL);

 //销毁读写锁
 pthread_rwlock_destroy(&rwlock);
 return 0;
```

```
}
```

- **条件变量**

条件变量用于某个进程或线程等待某个信号条件到来时才继续运行的场景。POSIX 库中提供了 pthread_cond_init 和 pthread_cond_destroy 函数用来完成条件变量的创建和销毁；提供 pthread_cond_wait 和 pthread_cond_timewait 函数用来完成线程等待或限时等待在某个条件量上，函数中的参数为条件变量和互斥量，以上函数将利用互斥量来完成对条件变量状态的修改，以保证多线程状态下条件变量一致性，因此在调用这两个函数之前，一定要获得这个互斥量的资源，即在此函数调用前一定要有互斥量的加锁操作；提供 pthread_cond_signal 和 pthread_cond_broadcast 函数用来唤醒等待在条件变量上的一个线程或所有线程。

2.3.4 Web 服务器的多线程模型

与 Web 服务器的多进程模型类似，在主线程接收到客户端连接请求信号后（accept 函数返回与客户端的连接），通过 pthread_create 函数来创建一个线程来处理这个客户端的请求信号。Web 服务器将为每个连接客户端创建一个线程来单独处理这个客户端的请求信息，如下图 2-3 所示。与 Web 服务器多进程模型不同的是，这里每个线程与主线程共存于 Web 服务器进程空间中，共享 Web 服务器进程资源。

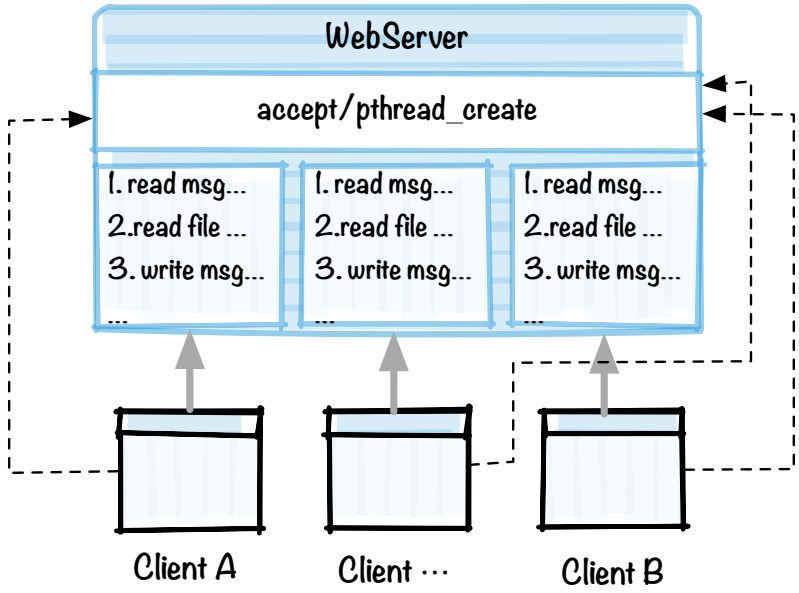

图 2-3Web 服务器的多线程模型

下面代码在 nweb 项目基础上，利用 POSIX 线程函数，实现了多线程模型的 Web 服务器。

```
//编译代码指令 gcc -std=gnu99 -g -o multithread_webserver multithread_webserver.c -lphtread
#include <stdio.h>
#include <stdlib.h>
```

```
#include <unistd.h>
#include <errno.h>
#include <string.h>
#include <fcntl.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <pthread.h>
#include <sys/stat.h>

#define VERSION 23
#define BUFSIZE 8096
#define ERROR      42
#define LOG        44
#define FORBIDDEN 403
#define NOTFOUND  404

#ifndef SIGCLD
#   define SIGCLD SIGCHLD
#endif

struct {
    char *ext;
    char *filetype;
} extensions [] = {
    {"gif", "image/gif" },
    {"jpg", "image/jpg" },
    {"jpeg","image/jpeg"},
    {"png", "image/png" },
    {"ico", "image/ico" },
    {"zip", "image/zip" },
    {"gz",  "image/gz"  },
    {"tar", "image/tar" },
    {"htm", "text/html" },
    {"html","text/html" },
    {0,0} };

typedef struct {
    int hit;
    int fd;
} webparam;


unsigned long get_file_size(const char *path)
{
    unsigned long filesize = -1;
    struct stat statbuff;
    if(stat(path, &statbuff) < 0){
        return filesize;
    }else{
        filesize = statbuff.st_size;
    }
    return filesize;
}

void logger(int type, char *s1, char *s2, int socket_fd)
{
    int fd ;
    char logbuffer[BUFSIZE*2];

    switch (type) {
```

```
                case ERROR: (void)sprintf(logbuffer,"ERROR: %s:%s Errno=%d exiting pid=%d",s1, s2, errno,getpid());
                    break;
                case FORBIDDEN:
                    (void)write(socket_fd, "HTTP/1.1 403 Forbidden\nContent-Length: 185\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>403 Forbidden</title>\n</head><body>\n<h1>Forbidden</h1>\nThe requested URL, file type or operation is not allowed on this simple static file webserver.\n</body></html>\n",271);
                    (void)sprintf(logbuffer,"FORBIDDEN: %s:%s",s1, s2);
                    break;
                case NOTFOUND:
                    (void)write(socket_fd, "HTTP/1.1 404 Not Found\nContent-Length: 136\nConnection: close\nContent-Type: text/html\n\n<html><head>\n<title>404 Not Found</title>\n</head><body>\n<h1>Not Found</h1>\nThe requested URL was not found on this server.\n</body></html>\n",224);
                    (void)sprintf(logbuffer,"NOT FOUND: %s:%s",s1, s2);
                    break;
                case LOG: (void)sprintf(logbuffer," INFO: %s:%s:%d",s1, s2,socket_fd); break;
            }
            /* No checks here, nothing can be done with a failure anyway */
            if((fd = open("nweb.log", O_CREAT| O_WRONLY | O_APPEND,0644)) >= 0) {
                (void)write(fd,logbuffer,strlen(logbuffer));
                (void)write(fd,"\n",1);
                (void)close(fd);
            }
            //if(type == ERROR || type == NOTFOUND || type == FORBIDDEN) exit(3);
        }

        /* this is a web thread, so we can exit on errors */
        void *  web(void * data)
        {
            int fd;
            int hit;

            int j, file_fd, buflen;
            long i, ret, len;
            char * fstr;
            char buffer[bufsize+1]; /* static so zero filled */
            webparam *param=(webparam*) data;
            fd=param->fd;
            hit=param->hit;

            ret =read(fd,buffer,bufsize);    /* read web request in one go */
            if(ret == 0 || ret == -1) {  /* read failure stop now */
                logger(forbidden,"failed to read browser request","",fd);

            }else{
                if(ret > 0 && ret < bufsize)  /* return code is valid chars */
                    buffer[ret]=0;     /* terminate the buffer */
                else buffer[0]=0;
                for(i=0;i<ret;i++)  /* remove cf and lf characters */
                    if(buffer[i] == '\r' || buffer[i] == '\n')
                        buffer[i]='*';
                logger(log,"request",buffer,hit);
                if( strncmp(buffer,"get ",4) && strncmp(buffer,"get ",4) ) {
                    logger(forbidden,"only simple get operation supported",buffer,fd);
                }
                for(i=4;i<bufsize;i++) { /* null terminate after the second space to ignore extra stuff */
                    if(buffer[i] == ' ') { /* string is "get url " +lots of other stuff */
                        buffer[i] = 0;
                        break;
                    }
                }
```

```
            for(j=0;j<i-1;j++)     /* check for illegal parent directory use .. */
                if(buffer[j] == '.' && buffer[j+1] == '.') {
                    logger(forbidden,"parent directory (..) path names not supported",buffer,fd);
                }
            if( !strncmp(&buffer[0],"get /\0",6) || !strncmp(&buffer[0],"get /\0",6) ) /* convert no filename to
index file */
                (void)strcpy(buffer,"get /index.html");

            /* work out the file type and check we support it */
            buflen=strlen(buffer);
            fstr = (char *)0;
            for(i=0;extensions[i].ext != 0;i++) {
                len = strlen(extensions[i].ext);
                if( !strncmp(&buffer[buflen-len], extensions[i].ext, len)) {
                    fstr =extensions[i].filetype;
                    break;
                }
            }
            if(fstr == 0) logger(forbidden,"file extension type not supported",buffer,fd);

            if(( file_fd = open(&buffer[5],o_rdonly)) == -1) {   /* open the file for reading */
                logger(notfound, "failed to open file",&buffer[5],fd);
            }
            logger(log,"send",&buffer[5],hit);
            len = (long)lseek(file_fd, (off_t)0, seek_end); /* 使用 lseek 来获得文件长度，比较低效*/
            (void)lseek(file_fd, (off_t)0, seek_set);          /* 想想还有什么方法来获取*/
              (void)sprintf(buffer,"http/1.1  200  ok\nserver:  nweb/%d.0\ncontent-length:  %ld\nconnection:
close\ncontent-type: %s\n\n", version, len, fstr); /* header + a blank line */
            logger(log,"header",buffer,hit);
            (void)write(fd,buffer,strlen(buffer));

            /* send file in 8kb block - last block may be smaller */
            while (   (ret = read(file_fd, buffer, bufsize)) > 0 ) {
                (void)write(fd,buffer,ret);
            }
            usleep(10000);/*在 socket 通道关闭前，留出一段信息发送的时间*/
            close(file_fd);
        }
        close(fd);
        //释放内存
        free(param);
    }

    int main(int argc, char **argv)
    {
        int i, port, pid, listenfd, socketfd, hit;
        socklen_t length;
        static struct sockaddr_in cli_addr; /* static = initialised to zeros */
        static struct sockaddr_in serv_addr; /* static = initialised to zeros */

        if( argc < 3   || argc > 3 || !strcmp(argv[1], "-?") ) {
            (void)printf("hint: nweb Port-Number Top-Directory\t\tversion %d\n\n"
                        "\tnweb is a small and very safe mini web server\n"
                        "\tnweb only servers out file/web pages with extensions named below\n"
                        "\t and only from the named directory or its sub-directories.\n"
                        "\tThere is no fancy features = safe and secure.\n\n"
                        "\tExample: nweb 8181 /home/nwebdir &\n\n"
                        "\tOnly Supports:", VERSION);
            for(i=0;extensions[i].ext != 0;i++)
                (void)printf(" %s",extensions[i].ext);

            (void)printf("\n\tNot Supported: URLs including \"..\", Java, Javascript, CGI\n"
                        "\tNot Supported: directories / /etc /bin /lib /tmp /usr /dev /sbin \n"
```

```
                    "\tNo warranty given or implied\n\tNigel Griffiths nag@uk.ibm.com\n"   );
            exit(0);
        }
        if( !strncmp(argv[2],"/"     ,2 ) || !strncmp(argv[2],"/etc", 5 ) ||
            !strncmp(argv[2],"/bin",5 ) || !strncmp(argv[2],"/lib", 5 ) ||
            !strncmp(argv[2],"/tmp",5 ) || !strncmp(argv[2],"/usr", 5 ) ||
            !strncmp(argv[2],"/dev",5 ) || !strncmp(argv[2],"/sbin",6) ){
            (void)printf("ERROR: Bad top directory %s, see nweb -?\n",argv[2]);
            exit(3);
        }
        if(chdir(argv[2]) == -1){
            (void)printf("ERROR: Can't Change to directory %s\n",argv[2]);
            exit(4);
        }
        /* Become deamon + unstopable and no zombies children (= no wait()) */
        if(fork() != 0)
            return 0; /* parent returns OK to shell */
        (void)signal(SIGCLD, SIG_IGN); /* ignore child death */
        (void)signal(SIGHUP, SIG_IGN); /* ignore terminal hangups */
        for(i=0;i<32;i++)
            (void)close(i);     /* close open files */
        (void)setpgrp();    /* break away from process group */
        logger(LOG,"nweb starting",argv[1],getpid());
        /* setup the network socket */
        if((listenfd = socket(AF_INET, SOCK_STREAM,0)) <0)
            logger(ERROR, "system call","socket",0);
        port = atoi(argv[1]);
        if(port < 0 || port >60000)
            logger(ERROR,"Invalid port number (try 1->60000)",argv[1],0);

        //初始化线程属性，为分离状态
        pthread_attr_t attr;
        pthread_attr_init(&attr);
        pthread_attr_setdetachstate(&attr,PTHREAD_CREATE_DETACHED);
        //
        pthread_t pth;
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
        serv_addr.sin_port = htons(port);
        if(bind(listenfd, (struct sockaddr *)&serv_addr,sizeof(serv_addr)) <0)
            logger(ERROR,"system call","bind",0);
        if( listen(listenfd,64) <0)
            logger(ERROR,"system call","listen",0);
        for(hit=1; ;hit++) {
            length = sizeof(cli_addr);
            if((socketfd = accept(listenfd, (struct sockaddr *)&cli_addr, &length)) < 0)
                logger(ERROR,"system call","accept",0);
            webparam *param=malloc(sizeof(webparam));
            param->hit=hit;
            param->fd=socketfd;
            if(pthread_create(&pth, &attr, &web, (void*)param)<0){
                logger(ERROR,"system call","pthread_create",0);
            }
        }
    }
```

2.3.5 Web 服务器的多线程模型

题目 1. 将上述多线程 Web 服务器与实验 2 中多进程模型的 Web 服务器性能进行对比，说明它俩各具有什么优缺点。具体对比的指标包括如下：

- 使用 http_load 命令来测试这两个模型下 Web 服务器的性能指标。并根据这些测试指标对比，分析为什么这两种模型会产生不同的性能结果?
- 对这两个模型中的 socket 数据读取、发送、网页文件读取和日志文件写入四个 I/O 操作分别计时，并打印出每个进程或线程处理各项 I/O 计时的平均时间。例如，编写的程序应该打印出如下结果。

```
共用 10000ms 成功处理 100 个客户端请求，其中
    平均每个客户端完成请求处理时间为 5100ms。
    平均每个客户端完成读 socket 时间为 500ms。
    平均每个客户端完成写 socket 时间为 1000ms。
    平均每个客户端完成读网页数据时间为 110ms。
    平均每个客户端完成写日志数据时间为 50ms。
```

- 根据上面的计时数据结果，分析并说明哪些多进程模型和多线程模型中哪些 I/O 操作是最消耗时间的？。
- 思考一下，怎么修改线程模型，才能提高线程的并发性能?

题目 2. 调整 http_load 命令参数，增加其并发访问线程数量，会发现随着并发访问数量在达到一定数量后，再增多会导致多线程 Web 服务进程的性能出现下降的现象。试分析产生上述现象的原因是什么？

2.4 线程池模型

与进程相比，虽然线程创建、销毁的代价较小，但还是需要系统内核为其分配运行堆栈。由于在多线程模型中每个线程完成的业务逻辑基本一样，因此如果线程在完成一次客户端请求处理后并不退出，而是等待运行客户端的下一次请求，那么将节省线程的创建、销毁所耗费的时间。

同时如果有大量客户端同时请求 Web 服务器时，将造成 Web 服务器同时创建大量的线程，这些线程将相互竞争 CPU 资源、I/O、进程内临界资源等计算机资源，从而导致 Web 服务进程并发吞吐量降低。对于 CPU 利用率来说，如果线程数量增多，则因为线程的读写 I/O 阻塞而导致的线程上下文切换次数增多，则 CPU 的利用率会下降。对于外存 I/O，如果多个线程竞争对外存的读写权，由于外存存储数据的特性以及 I/O 传输数据带宽限制，会导致大多数线程存在阻塞状态。如果多个线程之间存在临界资源、数据同步等使用问题，随着线程数量的增多，也会使得线程的并发性下降。总之，通过以上分析，会发现 Web 服务进程的并发、吞吐性能并不是随着线程数量的增多而增强。一般情况下，在初始状态，Web 服务进程的性能随线程数量增多而增强，但是线程到达一定数量后，其性能会随着线程进一步增多而下降。

基于以上分析，会得到两个结论：1）进程中并非线程数量越多，I/O 处理能力越强；2）在 Web 服务器中，每个线程处理的业务逻辑是相同的，而每个线程的创建、销毁都要销毁时间。

根据以上两点，如果设计一种结构，能够在初始化时就创建一定数量的线程，并且这些线程在处理完任务后并不退出，而是等待下一次任务的到来。这样就能够同时克服以上两个问题。目前实行这种思想的结构被称为线程池。

在设计和实现线程池时，需要考虑两个问题：1）要完成的任务如何进行封装，才能让已经创建的线程来运行它？；2）已经创建的线程如何能够知道任务的到来，并且在运行完后并不销毁？

要解决第一个问题，首先考虑 pthread_create 函数中的参数 void* (start_routine)(void*) 和 void* arg 分别为要运行任务逻辑的函数指针和这个函数的参数指针。此函数在创建线程堆栈等运行体后，一定通过执行“start_routine(arg)”代码来执行业务代码（即封装运行任务逻辑的函数）。虽然在线程池中，表示运行任务逻辑的这两个参数不能通过创建线程来进行传递，但是在线程执行函数内，如果能够得到这两个参数，那么可以通过执行“start_routine(arg)”代码来完成。因此，运行任务中如果封装了这两个参数，则就能够让线程运行它。

要解决第二问题，首先分析一下线程池中线程运行状态和场景。既然在初始化线程池时，需要创建一定数量的线程，那么这些线程的运行函数在创建后一定要阻塞，直到有信号通知它们，它们才能够执行任务。在执行任务完成后，线程的运行函数还要阻塞，等待下一次信号到来。

根据上面分析，可以描绘出一幅线程池运行状态图，如图 2-4 所示。其中，task queue 为存放任务的队列；thread array 为线程数组。在初始化时，由于任务队列中没有任务，所有的线程全部阻塞在条件变量上。当向任务队列里面添加任务时，可以恢复阻塞在条件变量上的线程运行。线程运行时从任务队列头部里面取一个任务，然后执行这个任务，当执行任务结束后，会判断目前是否还有任务在任务队列，如果有则继续执行前面的步骤；如果没有则阻塞在这个条件变量。

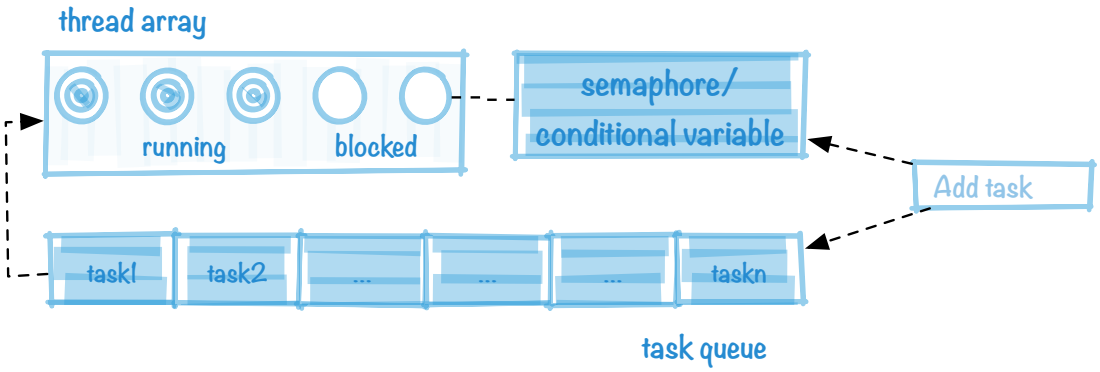

图 2-4 线程池运行状态图

通过整理上面线程运行描述，可以得到以下的线程之间同步/互斥操作。

- 如果任务队列里面有任务，则线程不会阻塞；没有任务线程才会阻塞。

- 线程从任务队列消费任务，增加任务函数（Add task）向任务队列生成任务。因此任务队列是生产者和消费者问题中的临界资源。
- 线程池在销毁时，需要等待线程池内所有的线程运行完毕，才能释放线程池所占资源。

下面为线程池的数据结构和线程池接口函数。其中在线程池接口函数中，使用“…”，表示忽略的代码，但是这些忽略代码的逻辑在相关位置都有描述。除此之外，代码中并没有给出任务队列操作的相关函数，push_taskqueue，take_taskqueue，init_taskqueue 和 destory_taskqueue。

```
/* queue status and conditional variable*/
typedef struct staconv {
 pthread_mutex_t mutex;
 pthread_cond_t    cond;    /*用于阻塞和唤醒线程池中线程*/
 int status;               /*表示任务队列状态：false 为无任务；true 为有任务*/
} staconv;

/*Task*/
typedef struct task{
 struct task*  next;                  /* 指向下一任务    */
 void    (*function)(void* arg);      /*  函数指针       */
 void*  arg;                          /*  函数参数指针  */
} task;

/*Task Queue*/
typedef struct taskqueue{
 pthread_mutex_t mutex;               /* 用于互斥读写任务队列 */
 task  *front;                        /* 指向队首             */
 task  *rear;                         /* 指向队尾             */
 staconv *has_jobs;                   /* 根据状态，阻塞线程  */
 int    len;                          /* 队列中任务个数      */
} taskqueue;

/* Thread */
typedef struct thread{
 int         id;                      /* 线程 id             */
 pthread_t pthread;                   /* 封装的 POSIX 线程      */
 struct threadpool* pool;             /* 与线程池绑定         */
} thread;

/*Thread Pool*/
typedef struct threadpool{
 thread**    threads;                 /* 线程指针数组              */
 volatile int num_threads;            /* 线程池中线程数量          */
 volatile int num_working;            /* 目前正在工作的线程个数      */
 pthread_mutex_t  thcount_lock;       /* 线程池锁用于修改上面两个变量 */
 pthread_cond_t  threads_all_idle;    /* 用于销毁线程的条件变量      */
 taskqueue  queue;                    /* 任务队列                  */
 volatile bool is_alive;              /* 表示线程池是否还存活        */
}threadpool;

/*线程池初始化函数*/
struct threadpool* initTheadPool(int num_threads){
 //创建线程池空间
 threadpool* pool;
 pool=(threadpool*)malloc(sizeof(struct threadpool));
     pool->num_threads=0;
```

```
    pool->num_working=0;
    //初始化互斥量和条件变量
    pthread_mutex_init(&(thpool_p->thcount_lock), NULL);
 pthread_cond_init(&thpool_p->threads_all_idle, NULL);
 //初始化任务队列
 //****需实现*****
 init_taskqueue(&pool->queue);
 //创建线程数组
 pool->threads=(struct thread **)malloc(num_threads*sizeof(struct* thread));
 //创建线程
 for (int i = 0; i < num_threads; ++i)
 {
       create_thread(pool,pool->thread[i],i);   //i 为线程 id,
 }
 //等等所有的线程创建完毕,在每个线程运行函数中将进行 pool->num_threads++ 操作
 //因此，此处为忙等待，直到所有的线程创建完毕，并马上运行阻塞代码时才返回。
 while(pool->num_threads!=num_threads) {}

 return pool;
}

/*向线程池中添加任务*/
void addTask2ThreadPool(threadpool* pool,task* curtask){
 //将任务加入队列
 //****需实现*****
 push_taskqueue(&pool->queue,curtask);
}
/*等待当前任务全部运行完*/
void waitThreadPool(threadpool* pool){
 pthread_mutex_lock(&pool->thcount_lock);
 while (pool->jobqueue.len || pool->num_threads_working) {
       pthread_cond_wait(&pool->threads_all_idle, &pool->thcount_lock);
 }
 pthread_mutex_unlock(&thpool_p->thcount_lock);
}
/*销毁线程池*/
void destoryThreadPool(threadpool* pool){
 //如果当前任务队列中有任务，需等待任务队列为空，并且运行线程执行完任务后
 ....
 ....
 ....
 //销毁任务队列
 //****需实现*****
 destory_taskqueue(&pool->queue);
 //销毁线程指针数组,并释放所有为线程池分配的内存
 ....
 ....
 ....
}
/*获得当前线程池中正在运行线程的数量*/
int getNumofThreadWorking(threadpool* pool){
 return pool->num_working;
}

/*创建线程*/
int create_thread（struct threadpool* pool, struct thread** pthread, int id){
    //为 thread 分配内存空间
 *pthread = (struct thread*)malloc(sizeof(struct thread));
 if (pthread == NULL){
       error("creat_thread(): Could not allocate memory for thread\n");
       return -1;
 }
 //设置这个 thread 的属性
```

```
        (*pthread)->pool = pool;
        (*pthread)->id     = id;
            //创建线程
        pthread_create(&(*pthread)->pthread, NULL, (void *)thread_do, (*pthread));
        pthread_detach((*pthread)->pthread);
        return 0;
       }
       /*线程运行的逻辑函数*/
       void* thread_do(struct thread* pthread){

        /* 设置线程名字 */
        char thread_name[128] = {0};
        sprintf(thread_name, "thread-pool-%d", thread_p->id);

        prctl(PR_SET_NAME, thread_name);

        /* 获得线程池*/
        threadpool* pool = pthread->pool;

        /* 在线程池初始化时，用于已经创建线程的计数，执行 pool->num_threads++ */


        ....
        ....
        ....
        /*线程一直循环往复运行，直到 pool->is_alive 变为 false*/
        while(pool->is_alive){

              /*如果任务队列中还要任务，则继续运行，否则阻塞*/
               ....
               ....
               ....

              if (pool->is_alive){
                    /*执行到此位置，表明线程在工作，需要对工作线程数量进行计数*/
                    //pool->num_working++
                     ....
                     ....
                     ....
                     /* 从任务队列的队首提取任务，并执行*/
                     void (*func)(void*);
                     void*   arg;
                     //take_taskqueue 从任务队列头部提取任务，并在队列中删除此任务
                     //****需实现 take_taskqueue*****
                     task* curtask = take_taskqueue(&pool->queue);
                     if (curtask) {
                           func = curtask->function;
                           arg   = curtask->arg;
                           //执行任务
                           func(arg);
                           //释放任务
                           free(curtask);
                     }
                           /*执行到此位置，表明线程已经将任务执行完成，需更改工作线程数量*/
                     //此处还需注意，当工作线程数量为 0，表示任务全部完成，要让阻塞在 waitThreadPool 函
数上的线程继续运行
                     ....
                     ....
                     ....

              }
        }
        /*运行到此位置表明，线程将要退出，需更改当前线程池中的线程数量*/
        //pool->num_threads--
```

```
    ....
    ....
    ....
    return NULL;
}
```

2.4.1 实验 4 Web 服务器的线程池模型

题目 1. 添补相应的程序代码到上面函数中“……”位置处。

题目 2. 完成函数 push_taskqueue，take_taskqueue，init_taskqueue 和 destory_taskqueue。

题目 3. 添加必要的程序代码，以最终完成线程池。

题目 4. 利用实现的线程池，替换实验 3 中 Web 服务的多线程模型。

题目 5. 调整线程池中线程个数参数，以达到 Web 服务并发性能最优。利用 http_load 及其它性能参数，分析和对比多线程模型与线程池模型在 Web 服务进程中的优点和缺点。

2.5 业务分割模型

在多进程和多线程模型中每个进程或线程完成相同的任务（由于多进程与多线程模型在本节的描述中具有相同的含义，下面主要使用多线程来进行问题描述）。针对客户端的文件请求处理，每个任务包含五个步骤：网络读取数据、解析数据、读取文件、向网络发送数据和写日志文件。

这些步骤主要涉及两类 I/O 设备：外部存储器和网络。在多线程模型中，每个线程都要竞争使用这两类 I/O 设备，并且这两类设备 I/O 速度慢，从而导致在一个线程执行任务过程中大量时间阻塞在这两类设备上。同时，每个设备都是在处理完一个线程的 I/O 数据请求后，由操作系统恢复另一个阻塞在此设备上的线程来使用此设备。从设备的角度上看，该设备也并没有一直在处理数据，而是处理完一块数据后，需要等待操作系统调度另外线程后，才能再处理数据。具体设备工作时间如下图 2-5 所示，其中 W 表示设备在工作，I 表示设备在等待。

图 2-5 设备工作时间状态图

除了设备的等待时间外，由于每个线程中与 I/O 设备相关的数据区在内存独立，从而造成 I/O 设备操作数据的离散化和碎片化，尤其是每次读写的数据内容过少，会严重影响 I/O 设备性能。

例如，可以设计以下实验来进行验证：一次性从文件中读取 1MB 数据所消耗时间；分 1000 次，每次从文件读取 1KB 数据，读 1MB 数据所消耗的时间；分 1000000 次，每次从文件读 1 字节数据，读 1MB 数据所消耗的时间。具体实验代码如下。

```
// 编译指令 gcc -std=gnu99 -g -o readfiletimedemo readfiletimedemo.c
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
```

```
#include <sys/time.h>
#include <fcntl.h>

#define BUFSIZE 1024*1024
#define KB      1024

int main(int argc, char const *argv[])
{
 int fd ;
    char buffer[BUFSIZE];

    struct  timeval  start;
    struct  timeval  end;
    unsigned long timer;

    if((fd = open("nweb.log", O_CREAT| O_RDONLY,0644)) >= 0) {
     gettimeofday(&start,NULL);
     read(fd,buffer,BUFSIZE);
     gettimeofday(&end,NULL);
          timer   =   1000000   *   (end.tv_sec-start.tv_sec)+   end.tv_usec-
start.tv_usec;
      printf("read all 1MB date timer = %ld us\n",timer);

      gettimeofday(&start,NULL);
     for (int i = 0; i < 1024; ++i)
     {
         read(fd,buffer,KB);
     }
     gettimeofday(&end,NULL);
          timer   =   1000000   *   (end.tv_sec-start.tv_sec)+   end.tv_usec-
start.tv_usec;
        printf("read  1024  times,  each  1KB,  total  1MB  date  timer  =  %ld
us\n",timer);

     gettimeofday(&start,NULL);
     for (int i = 0; i < BUFSIZE; ++i)
     {
         read(fd,buffer,1);
     }
     gettimeofday(&end,NULL);
         timer   =   1000000   *   (end.tv_sec-start.tv_sec)+   end.tv_usec-
start.tv_usec;
      printf("read  1024*1024  times,  each  1B,  total  1MB  date  timer  =  %ld
us\n",timer);

     close(fd);

   }else{
     printf("cannot open the file\n");
   }
   return 0;
}
```

这个程序执行的结果如下所示。

```
read all 1MB date timer = 1043 us
read 1024 times, each 1KB, total 1MB date timer = 1523 us
read 1024*1024 times, each 1B, total 1MB date timer = 1081759 us
```

通过上面分析，可以通过以下两个方面来提高 I/O 设备利用率和读写速度。

- 尽量减少多个线程同时互斥使用设备的情况。
- 在每次操作 I/O 设备时，尽量向 I/O 设备读取或写入更多的数据。

为达到这两方面目标，可以考虑将原来任务按逻辑步骤进行分割，每个逻辑步骤为一个线程，步骤与步骤之间通过缓冲区进行数据传递，这就是业务分割模型，如下图 2-6 所示。

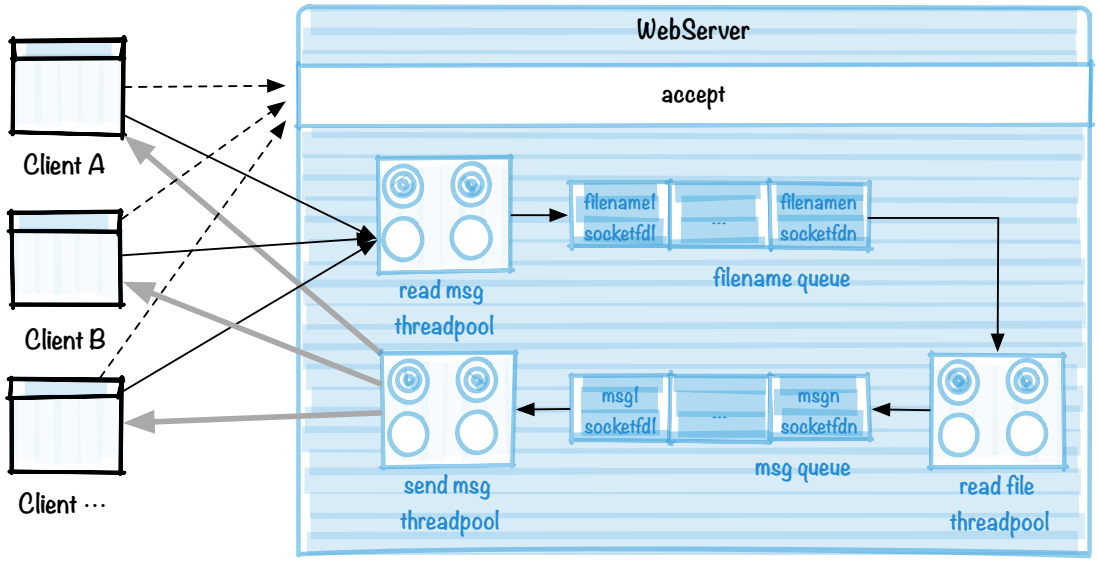

图 2-6 Web 服务器的业务分割模型

在 WebSever 服务进程中，按照业务步骤分别创建三个线程池（readmsg threadpool，readfile threadpool 和 sendmsg threadpool）和两个消息队列（filename queue 和 msg queue）。其中，readmsg threadpool 中的线程主要完成从客户端 socket 通道中读取消息并进行解析，然后将请求的文件名和 socket 通道加入到 filename queue；readfile threadpool 中的线程从 filename queue 中提取文件名，读取文件，并将文件内容和 socket 通道发送到 msg queue；sendmsg threadpool 中的线程从 msg queue 中提取文件内容和 socket 通道，并将文件内容通过 socket 通道发送到客户端。

在上述模型中，可以将 filename queue 作为 readfile pool 中的任务队列，将 msg queue 作为 sendmsg threadpool 中的任务队列；也可以将这两个消息队列作为单独的队列，然后通过增加消息队列上的线程，完成消息的读取和加入到下一个业务步骤线程池中。

建立上述的 WebServer 业务分割模型后，可以简单分析一下，其与多线程模型的不同。首先，在多线程模型中，每个步骤之间有严格的时序关系；而在业务分割模型中，由于每个步骤之间并没有直接的联系，而是通过消息队列而发生间接联系，从而达到步骤之间的解耦。并且通过消息队列作为数据缓冲区，可以减少和避免因为业务处理速度不一致而导致的性能下降现象。例如，如果消息队列长度仅为 1，并且在某个时间段，从 readmsg threadpool 中线程接收消息速度大于 readfile pool 中线程读取文件的速度，那么 readmsg threadpool 中的线程将会频繁阻塞于 filename queue 队列，等待队列有空余空间；而如果消息队列足够长，则在一定程度上可以避免此现象发生，因为这个时间段内 readmsg threadpool 中线程处理的所有数据都可以缓存在这个队列上。

其次，通过分割任务，使得每个线程执行的业务单元粒度变小。对于 I/O 密集型业务单元，可以通过极少量的线程就能够达到充分利用 I/O 设备的效果。例如，在上图的 readfile pool 中如果仅有一个线程，其几乎可以不断地读取磁盘中每个请求文件的内容，因为与 I/O

设备传递数据相比，与消息队列之间传递极为快速（内存数据传输速度为外存数据传输速度的几倍到数百倍，一般内存速度为 SSD 速度的几倍~几十倍，为磁盘的几十倍~数百倍）。即使考虑到 I/O 设备的并行性，也仅需几个线程就可以充分利用 I/O 设备。

最后，通过分割任务，每个线程执行的业务逻辑变得简单，这样更有利于分析、调试和优化多任务环境的程序性能。

2.5.1 实验 5 Web 服务器的业务分割模型

题目 1. 实现上述的业务分割模型的 Web 服务程序。

题目 2. 在程序里面设置性能监测代码，通过定时打印这些性能参数，能够分析此 Web 服务程序的运行状态。例如，线程池中线程平均活跃时间及其阻塞时间，线程最高活跃数量、最低活跃数量、平均活跃数量；消息队列中消息的长度等。除此以外还可以利用相关系统命令来监测系统的 I/O、内存、CPU 等设备性能。

题目 3. 通过上述的性能参数和系统命令，对 Web 服务程序进行逻辑分析，发现当前程序存在性能瓶颈的原因。进而通过控制各个线程池中的线程数量和消息队列长度，来改善此程序的性能。

2.6 混合模型

前面讲解的模型可以分为进程模型和线程模型两大类。进程模型主要依赖创建子进程来处理任务；线程模型主要依赖线程来处理任务。这两个模型各有优缺点。从性能角度考虑，多线程模型占据优势。这是因为，与进程相比，线程的创建、销毁和维护所需系统资源较少，并且线程之间数据共享方便。从安全角度考虑，多进程模型占据优势。这是因为，与线程相比，进程独立占有系统资源（内存、I/O 设备），一个进程运行崩溃不会或很少干扰另一个进程的运行。

如果综合考虑上述两种模型的特点，可以将多进程模型与多线程模型进行融合。其主要实现思想是在多进程模型中，每个进程运行环境内建立多线程模型。这样做的好处就是在为 Web 服务器带来安全的同时，也使得其具有较高的并发性。

实际上，这种将多进程与多线程进行混合的模型也出现在日常使用的程序内部。例如，如下图 2-7 和图 2-8 所示，chrome 浏览器会为每个标签页面建立一个进程，当在这个标签页面输入 URL 地址来进行文件请求时，chrome 浏览器会在这个进程内默认启动三个线程来并发获取其渲染页面所需要的资源。这样做除了能提高页面的渲染显示速度外，还具有很好的安全性，使得标签页面之间的运行并不相互干扰。当一个标签页面所对应的进程因为某种原因崩溃，并不会影响到其它标签页面的工作。

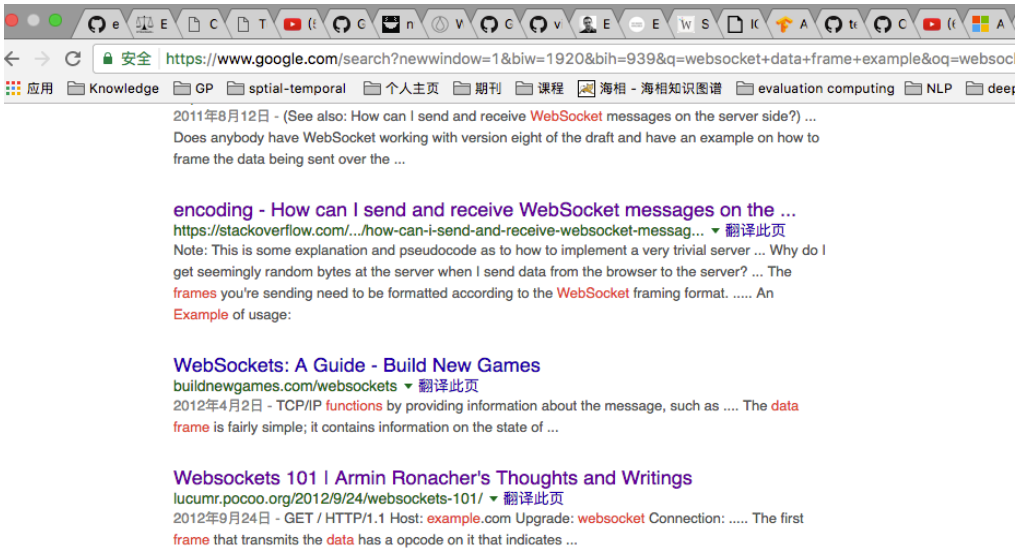

图 2-7 Chrome 浏览器多标签运行界面

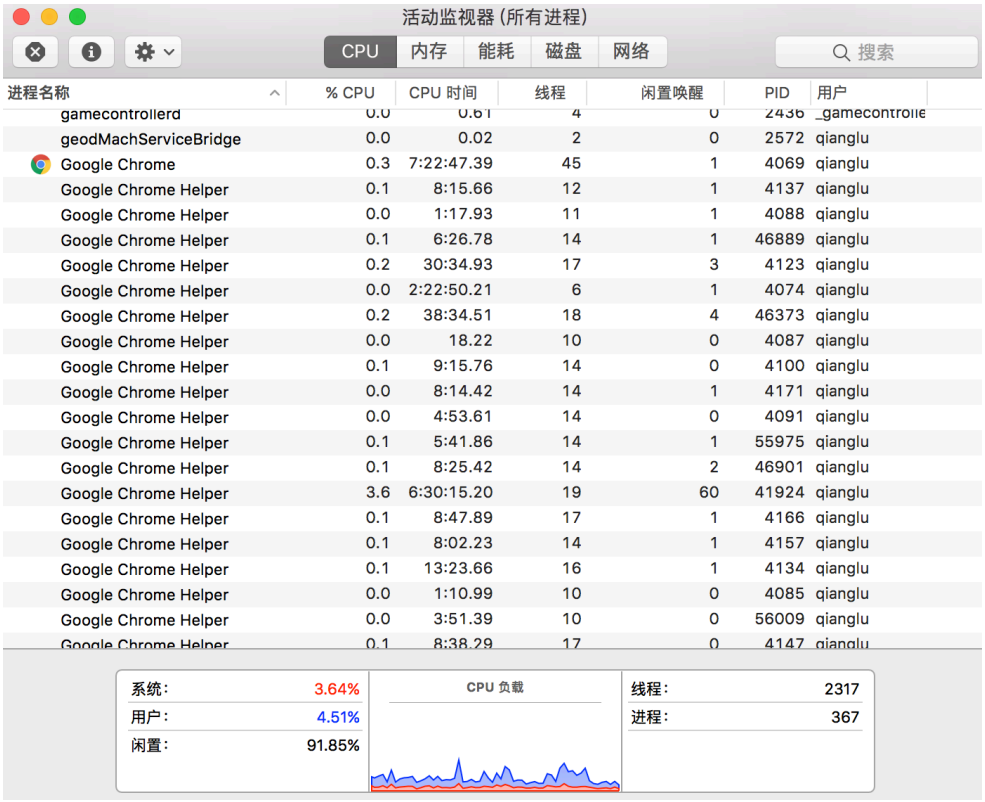

图 2-8 Chrome 浏览器启动的进程视图

与 chrome 浏览器中简单的多进程多线程混合模型相比，Web 服务器混合模型的设计会更加复杂。它需要综合考虑性能和安全前提下，动态调整此模型下进程和线程的数量。其主要涉及以下问题：

1. 在每个进程中，是使用多线程模型还是线程池模型？

2.在客户端并发请求数量增多后，是新建一个子进程及其多线程模型来处理新增请求，还是在原有的进程内部增加线程数量？如果在原有的进程内部增加线程数量，应该在哪个进程中增加线程？

3. 在客户端并发请求数量减少后，是减少原来进程中的线程数量？还是关闭进程？

这些问题并没有统一标准的答案，需要设计人员根据业务逻辑规则以及系统运行环境来进行权衡。设计人员在设计系统时，不能忽略的一个因素就是让系统逻辑和结构尽可能地保持简单（chrome 浏览器中多进程多线程使用理念）。因为复杂系统会带来设计、开发和维护的难度增加，会导致系统 bug 增多、系统容易崩溃和不安全。

在下图 2-9 中，给出了一个混合模型的 Web 服务器逻辑架构设计方案，以供读者以后进行相关服务设计时参考。此系统架构可以很容地扩展为分布式系统。

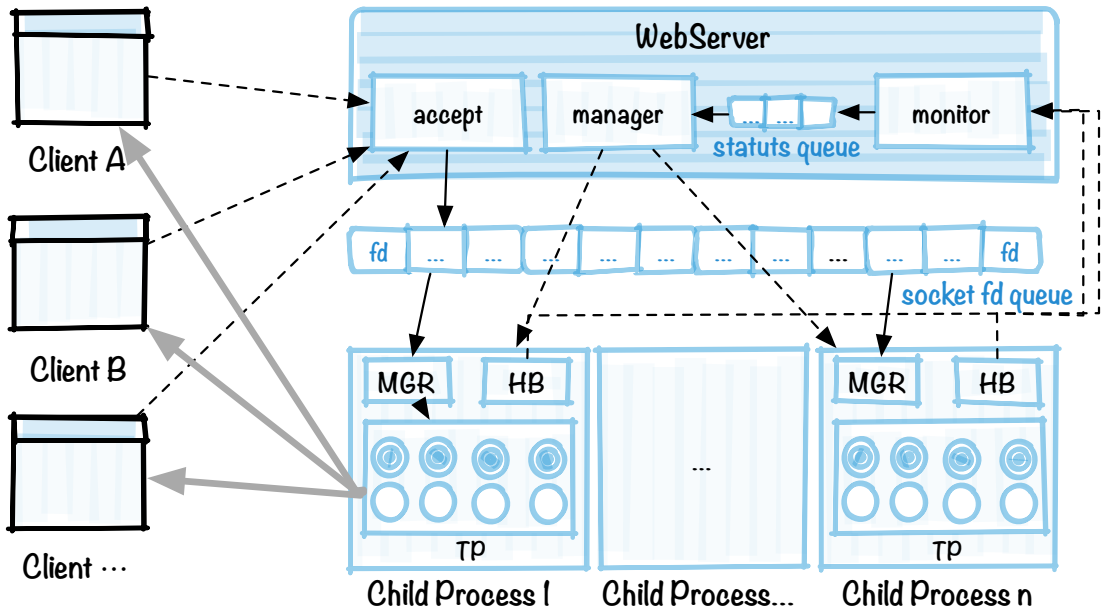

图 2-9 Web 服务器的混合模型

在 WebServer 主进程中包含三个线程，其中 accept 线程主要用于针对服务端口，当建立好客户连接通道后，就把这个通道描述符发送到内存共享队列 socketfd queue 中；manager 线程负责维护子进程，其根据状态队列（status queue）里面每个子进程运行状态、socketfd queue 的长度和当前系统性能等参数来决定是否创建或关闭任务子进程；monitor 线程为系统信息收集线程，其接收子进程心跳线程发送的心跳信号以及子进程运行状态信息，并将这些信息保存在 status queue 中。

在每个子进程中包含三个运行组件：MGR、HB 和 TP。其中，MGR 为子进程的管理线程，其负责从 socketfd queue 中提取客户端连接通道，然后根据本进程的执行状态，来决定是启动新的任务线程还是利用已经创建好的任务线程来处理此通道的信息；HB 为心跳线程（heartbeat），其定时将本进程的状态信息发送到主进程的 monitor 线程侦听端口；TP 为任务线程池或任务线程，其内部包含要执行具体业务逻辑的线程，此部分也可以进一步分解为

业务分割模型。除此以外，子进程还要包含信号处理函数，以响应主进程发送过来的管理信号。例如，主进程中 manager 线程通过检测发现目前并发请求的客户端少，系统中存在大量子进程，并且这些子进程中的任务线程未工作，这时可以向这些子进程发送关闭信号以关闭子进程。而这些子进程中关闭信号处理函数，将执行子进程退出时的处理工作，如等待执行完成、释放内存等。

2.6.1 实验 6 Web 服务器的混合模型

题目 1. 参考本节给出的基于混合模型的 Web 服务器，请尝试设计并实现它。在考虑心跳信息基础上，请仔细设计 manager 内部的子进程运行及控制的调度算法，使得系统具有优良的自适应能力。

题目 2. 考虑一下，是否能将本混合模型进行扩展分布式模型，即 Web 服务的主进程部署在一台主机，能够将客户端请求信号转发到后置系统，并具有负载平衡能力；而每个子进程部署在独立的主机上，作为后置系统来处理客户端的请求。这时，Web 服务的主进程被称为反代理服务器；而后置系统的子进程被称为 Web 服务器。