TCP/IP网络编程(十)

笔记本: 网络编程
创建时间: 2018/11/12 18:36 **更新时间:** 2018/11/12 20:47
作者: xiangkang94@outlook.com
标签: 第十章(多进程服务器端)

1. 并发服务器端实现模型和方法:
   a. 多进程服务器:通过创建多个进程提供服务
   b. 多路复用服务器:通过捆绑并统一管理IO对象提供服务(select和epoll)
   c. 多线程服务器:通过生成与客户端等量的线程提供服务

2. 通过fork函数创建进程:复制正在运行的调用fork函数的进程,复制相同的内存空间。两个进程都执行fork函数以后的语句

```
#include <unistd.h>
pid_t fork(void);  //成功时返回进程ID,失败时返回-1.
```

父进程Parent Prcess:fork函数返回子进程ID
子进程Child Process:fork函数返回0

图10-1 fork函数的调用

3. 僵死Zombie进程:进程已死,但仍旧占用系统资源
   a. 子进程终止方式:exit函数传递的参数和执行return并返回值。
   b. 僵死进程的产生:exit的传递值和return返回值会传递给操作系统,操作系统不会销毁子进程,直到这些值传递到父进程
   c. 如何传递值到父进程:操作系统不会主动把值传递到父进程,而是父进程主动请求,函数调用。如果父进程不主动请求,则子进程会一直处于僵死状态,所以父进程要负责收
   回子进程
4. 利用wait销毁僵死进程:父进程主动请求获取子进程的返回值

```
#include <sys/wait.h>
pid_t wait(int * statloc);//1. 成功时返回终止的子进程ID,失败返回-1
```

子进程终止参数保存在wait的参数中,里面还有其他信息,需要通过宏分离有效值
WIFEXITED:子进程注册终止,返回TRUE
WEXITSTATUS:返回子进程的返回值
调用wait函数时,如果没有已终止的子进程,那么程序blocking,直到有子进程终止,所以调用wait需谨慎

```
    pid = fork();
    if(pid == 0){
        exit(7);
    }
    else {
        printf("Child2 PID :%d \n",pid);
        wait(&status);
```

```
    if(WIFEXITED(status)){
        printf("Child one : %d \n",WEXITSTATUS(status));
    }
    wait(&status); //此处可能因为等不到终止的子进程而导致程序员阻塞
    if(WIFEXITED(status)){
        printf("Child two : %d \n",WEXITSTATUS(status));
    }
```

5. waitpid函数销毁僵死进程

```
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int * statloc, int options); //成功时返回终止子进程ID，失败返回-1
//pid：等待终止的目标子进程ID，若传递-1，则与wait函数相同，等待任意子进程
//statloc：与wait函数的statloc参数一致
//options：若传递头文件sys/wait.h中声明的常量WNOHANG，即使没有终止子进程也不会阻塞，而是返回0并退出函数
```

wait()和waitpid()两个函数都有一个缺点，就是没法确定子进程什么时候终止，难道要无休止的循环检测等待？
答案是向操作系统求助。

6. 在多进程间通信中

```
#include <signal.h>
//返回之前注册的函数指针。
void (*signal(int signo,void (*func)(int)))(int);

//sigaction函数类似于signal函数，完全可以替代后者，也更加稳定。
#include <signal.h>
struct sigaction{
    void (* sa_handler)(int);
    sigset_t sa_mask;
    int sa_flags;
}
int sigaction(int signo,const struct sigaction *act,struct sigaction *oldact);
//成功返回0，失败返回-1
//signo：传递的信号
//act：对应于第一个参数的信号处理函数
//oldact：获取之前注册的信号处理函数指针，若不需要则传递0
```

7.Signal函数

```
#include <signal.h>
void (*signal(int signo,void(*func)(int)))(int);
```

为了便于理解，做如下定义：
typedef void (*PF) (int)
PF signal(int signo, PF)
//当发生signo的情况时，调用PF指向的函数

```
SIGALRM: 已到通过调用alarm函数注册的时间
SIGINT:  输入了CTRL+C
SIGCHLD: 子进程终止
```

```
#include <unistd.h>
unsigned int alarm(unsigned int seconds)
//如果调用该函数同时向它传递一个正整数，相应的时间后将产生一个SIGALRM信号，若向该函数传递0，则取消之前对SIGALRM信号的预约，如果通过该函数的预约信号，但
//没有指定相应的响应函数，则通过调用signal函数终止进程，不做任何处理。
```

发生信号时将唤醒由于调用sleep()函数而进入阻塞状态的进程，而且进程一旦被唤醒不会再进入休眠状态，即使还没有到sleep参数给定的休眠时间。

8.sigaction函数进行信号处理
和signal函数很类似，但是由于unix系列的不同操作系统中可能存在区别，但sigaction函数完全相同，因此更适用

```
#include <signal.h>
struct sigaction{
```

```
    void (* sa_handler)(int);   //该成员保存信号处理函数的指针(地址值)
    sigset_t sa_mask;
    int sa_flags;
}
int sigaction(int signo,const struct sigaction *act,struct sigaction *oldact);
//成功返回0，失败返回-1
//signo：传递的信号
//act：对英语第一个参数的信号处理函数
//oldact：获取之前注册的信号处理函数指针，若不需要则传递0
```

Sample：利用信号销毁僵尸进程(**子进程终止时将产生SIGCHLD信号**)：

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <sys/wait.h>

void read_childproc(int sig){
    int status;
    pid_t id = waitpid(-1, &status, WNOHANG); //传递-1则与wait()函数相同，等待任意子进程结束
    if(WIFEXITED(status)){
        printf("Removed proc id : %d \n",id);
        printf("child send : %d \n",WEXITSTATUS(status));
    }
}

int main(int argc, char *argv[]){
    pid_t pid;
    struct sigaction act;
    act.sa_handler = read_childproc;
    sigemptyset(&act.sa_mask);
    act.sa_flags = 0;
    sigaction(SIGCHLD,&act,0);  //注册信号，触发SIGCHILD信号后将会调用act函数
    pid = fork();
    if(pid == 0){
        //child
        puts("Hi, I'm child process");
        sleep(5);
        return 12;  //五秒后将会触发SIGCHILD信号
    }
    else
    {
        printf("Child proc1 id : %d \n",pid);
        pid = fork();
        if(pid ==0){
            puts("Hi, sec child");
            sleep(5);
            exit(24);
        }
        else{
            int i;
            printf("child proc2 id : %d \n",pid);
            for(i=0;i<10;i++){
            puts("wait...");
            sleep(1);
            }
        }
    }
    return 0;
}
```

9.通过fork函数复制的子进程

问题是，进程中的变量是否由父进程和子进程共享?

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

```
#include<unistd.h>
int main(){
    pid_t pid;
    int num=0;
    pid=fork();
    if(pid==0){
        num+=10;
        printf("child %d\n",num);
        printf("child %d\n",&num);
    }
    if(pid>0){
        num+=20;
        printf("parent %d\n",num);
        printf("parent %d\n",&num);
    }
    return 0;
}
-------------------------------------------------
parent 20
parent -948260088
child 10
child -948260088
```

由于num在子进程中加上了10，结果为10，父进程中加上了20，结果为20.所以num这个变量对于父进程和子进程来说应该是独立的。
但是看两个进程中的num的地址，发现居然是一样的。

一个程序中代码中的变量要么是事先声明好的，要么是动态分配的，在这里num显然是事先分配好的变量，也就是说，num在声明时其地址就已经确定就好了。我们的num变量是在fork调用前定义的。所以无论如何num变量的地址是不会改变的。可是想一想我们打印的num地址是不是真正的地址呢？答案是否定的，这个地址是给我们程序员使用的虚拟内存地址，最后都会通过内核映射到实际的物理内存中去，所以，如果要出现我们程序运行的这种情况，只有一种可能，我们打印的是num的虚拟内存地址，它是无论如何不会改变的，可是num变量的值是独立的，那是因为通过num的地址映射的物理地址是不一样的，所以在物理内存中num应该是由两个的，只是他们两都映射到一个虚拟内存地址。

首先，子进程是完全复制父进程的，所以num的地址是一样的，可是**子进程复制的是虚拟地址空间，而非物理空间**。如果，子进程和父进程对变量只读，也就是说变量不会被改变，这时候，变量表现为共享的,此时物理空间只有一份。如果说父进程或者子进程需要改变变量，那么进程将会对物理内存进行复制，这个时候变量是独立的，也就是说，物理内存中存在两份空间。

同时，fork之后文件描述符也会被复制，导致父子进程的两个文件描述符指向同一个套接字，所以需要将无关的套接字文件描述符关闭。

10.分割TCP的I/O程序

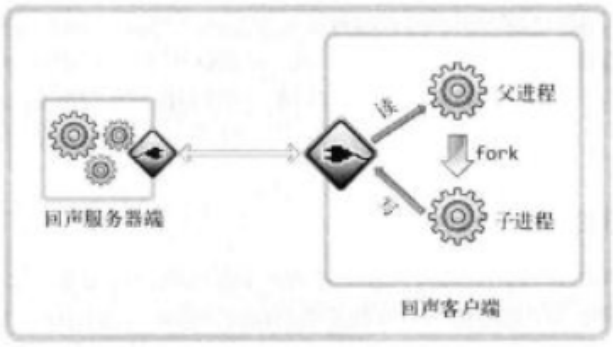

图10-5　回声客户端I/O分割模型

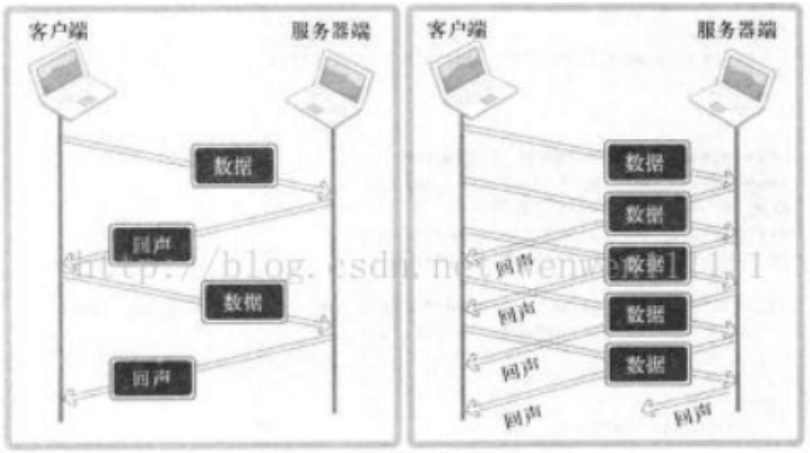

图10-6　数据交换方法比较

以回声客户端为例，分割后的客户端发送数据时不必考虑接受数据的情况，因此可以连续发送数据，来提高同一时间内传输的数据量