Linux 环境多进程编程

信息

Aa 姓名	# 学号	编辑日期
梁鑫嵘	200110619	@September 18, 2021

多进程初试

Demo

Linux 系统提供了一系列内核函数供我们使用： `fork()` 、 `exec()` 、 `wait()` 等。

有这样一个小 Demo 可以让我们一览 Linux 下多进程程序的运行场景。

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

int glob = 6;
int main() {
  pid_t pid;
  int x = 1;

  pid = fork();
  if (pid < 0) {
    perror("Error: ");
  } else if (pid == 0) {
    sleep(1);
    glob = glob + 2;
    x = x + 2;
    printf("child : glob(%p)=%d, x(%p)=%d\n", &glob, glob, &x, x);
  } else {
    sleep(2);
    glob = glob + 1;
    x = x + 1;
    printf("parent: glob(%p)=%d, x(%p)=%d\n", &glob, glob, &x, x);
```

```
  }
}
```

可以看到，在调用 `fork()` 之后，程序瞬间分裂成为两个分支：主进程和子进程，判断主进程还是子进程的方法是查看 `fork()` 函数的返回值。

调整 `Sleep(x)` 的参数可以改变主进程和子进程的运行先后顺序。如果子进程先运行，输出是这样：

```
child : glob(0x555555558010)=8, x(0x7fffffffd580)=3
parent: glob(0x555555558010)=7, x(0x7fffffffd580)=2
```

如果主进程先运行，输出：

```
parent: glob(0x555555558010)=7, x(0x7fffffffd580)=2
child : glob(0x555555558010)=8, x(0x7fffffffd580)=3
```

可以看到，输出的这两行虽然行的顺序不同，但是行的内容是一样的，甚至变量的地址都是一样——但是变量在两个进程之间的值却不同。

“全局”变量竟然不再是“全局”。

fork()

`fork()` 函数是一个特殊的函数，它能返回两次。对于主进程，返回值为子进程的进程ID（不是PID）；对于子进程，它的返回值是 `0` 。

在这样一个“分裂”过程中，程序的堆栈被整体复制，所以子程序和主程序拥有相同的变量和函数；由于虚拟地址，所以变量的地址也是不变的，但是变量已经不是在同一个物理地址了。

在简单 HTTP Server 上使用多进程处理

这是一个基于 Socket 编程的简单 HTTP Server。

```
#include <arpa/inet.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>

#define PORT 8000  // 服务器监听端口
```

```
void send_handle(int client_socket) {
  char status[] = "HTTP/1.0 200 OK\r\n";
  char header[] =
      "Server: DWBServer\r\nContent-Type: text/html;charset=utf-8\r\n\r\n";
  char body[] =
      "<html><head><title>C语言构建小型Web服务器</title></head><body><h2>欢迎</"
      "h2><p>Hello，World</p></body></html>";

  printf("enter sleeping...\n");
  sleep(10);  //让进程进入睡眠状态，单位是秒。
  printf("finish sleeping...\n");

  write(client_socket, status, sizeof(status));
  write(client_socket, header, sizeof(header));
  write(client_socket, body, sizeof(body));

  close(client_socket);
}

int main() {
  int server_socket = socket(AF_INET, SOCK_STREAM, 0);  //初始化套接字
  struct sockaddr_in server_addr;

  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;  // 服务地址和端口配置
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  server_addr.sin_port = htons(PORT);

  bind(server_socket, (struct sockaddr *)&server_addr,
       sizeof(server_addr));  //将本地地址绑定到所创建的套接字上

  listen(server_socket, 5);  //开始监听是否有客户端连接，第二个参数是最大监听数

  char buf[1024];
  int client_socket;

  while (1) {
    printf("======waiting for client's request=====\n");
    if ((client_socket = accept(server_socket, (struct sockaddr *)NULL,
                                NULL)) == -1) {  //等待客户端（浏览器）连接
      printf("accept socket error :%s(errno:%d)\n", strerror(errno), errno);
      continue;
    }
    printf("======waiting for read request data=====\n");
    read(client_socket, buf, 1024);  //读取客户端内容，这里是HTTP的请求数据
    // printf("%s",buf);     // 打印读取的内容

    send_handle(client_socket);
  }
  close(server_socket);

  return 0;
}
```

编译成 `socket_server_demo` 可执行文件，我们运行它。 `./socket_server_demo` 。

然后对 `8000` 端口进行测试，观察请求返回时间。

```
wget 127.0.0.1:8000 &
wget 127.0.0.1:8000 &
```

我们执行了两次网络请求，但是服务器对于这两次请求是顺次处理的，先处理第一个，然后再处理第二个，所以最后两次请求用了足足 20 秒。

```
# 请求 Client 端
➜  data git:(master) wget 127.0.0.1:8000 &
wget 127.0.0.1:8000 &
[1] 34340
[2] 34341
➜  data git:(master)
Redirecting output to ‘wget-log’.
Redirecting output to ‘wget-log.1’.
[2]  + 34341 done       wget 127.0.0.1:8000
➜  data git:(master) ✗
[1]  + 34340 done       wget 127.0.0.1:8000
➜  data git:(master) ✗

# 服务 Server 端
➜  src git:(master) ./socket_server_demo
======waiting for client's request=====
======waiting for read request data=====
enter sleeping...
finish sleeping...
======waiting for client's request=====
======waiting for read request data=====
enter sleeping...
finish sleeping...
======waiting for client's request=====
^C
➜  src git:(master) ✗
```

我们使用多进程改善这一情况。

```
#include <arpa/inet.h>
#include <errno.h>
#include <stdio.h>
#include <string.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <sys/types.h>


#define PORT 8000  // 服务器监听端口

void send_handle(int client_socket) {
```

```
  char status[] = "HTTP/1.0 200 OK\r\n";
  char header[] =
      "Server: DWBServer\r\nContent-Type: text/html;charset=utf-8\r\n\r\n";
  char body[] =
      "<html><head><title>C语言构建小型Web服务器</title></head><body><h2>欢迎</"
      "h2><p>Hello，World</p></body></html>";

  printf("enter sleeping...\n");
  sleep(10);  //让进程进入睡眠状态，单位是秒。
  printf("finish sleeping...\n");

  write(client_socket, status, sizeof(status));
  write(client_socket, header, sizeof(header));
  write(client_socket, body, sizeof(body));

  close(client_socket);
}

int main() {
  int server_socket = socket(AF_INET, SOCK_STREAM, 0);  //初始化套接字
  struct sockaddr_in server_addr;

  memset(&server_addr, 0, sizeof(server_addr));
  server_addr.sin_family = AF_INET;  // 服务地址和端口配置
  server_addr.sin_addr.s_addr = htonl(INADDR_ANY);
  server_addr.sin_port = htons(PORT);

  bind(server_socket, (struct sockaddr *)&server_addr,
       sizeof(server_addr));  //将本地地址绑定到所创建的套接字上

  listen(server_socket, 5);  //开始监听是否有客户端连接，第二个参数是最大监听数

  char buf[1024];
  int client_socket;

  while (1) {
    printf("======waiting for client's request=====\n");
    if ((client_socket = accept(server_socket, (struct sockaddr *)NULL,
                                NULL)) == -1) {  //等待客户端（浏览器）连接
      printf("accept socket error :%s(errno:%d)\n", strerror(errno), errno);
      continue;
    }
    printf("======waiting for read request data=====\n");
    read(client_socket, buf, 1024);  //读取客户端内容，这里是HTTP的请求数据
    // printf("%s",buf);     // 打印读取的内容

    // Chiro: 改成多进程处理
    // 仅仅让子进程处理，主进程回到 lisent.
    pid_t pid = fork();
    if (pid == 0) {
      // 子进程
      send_handle(client_socket);
      return 0;
    } else {
      // 父进程
      // do nothing.
    }
  }
```

```
  close(server_socket);

  return 0;
}
```

这次仅仅用时 10s 左右，因为两个请求是几乎同时开始处理的。

```
# client
➜  data git:(master) ✗ wget 127.0.0.1:8000 &
wget 127.0.0.1:8000 &
[1] 34487
[2] 34488
Redirecting output to ‘wget-log.2’.
Redirecting output to ‘wget-log.3’.
➜  data git:(master) ✗
➜  data git:(master) ✗
# server
➜  src git:(master) ✗ ./socket_server_multiprocessing
======waiting for client's request=====
======waiting for read request data=====
======waiting for client's request=====
enter sleeping...
======waiting for read request data=====
======waiting for client's request=====
enter sleeping...
finish sleeping...
finish sleeping...
^C
➜  src git:(master) ✗
```