前置

完成一个简单的shell程序，总体的框架和辅助代码都已经提供好了，我们需要完成的函数主要以下几个:

- eval: 主要功能是解析 cmdline，并且运行. [70 lines]
- builtin_cmd: 辨识和解析出 bulidin 命令: quit, fg, bg, and jobs. [25lines]
- do_bgfg: 实现 bg 和 fg 命令. [50 lines]
- waitfg: 实现等待前台程序运行结束. [20 lines]
- sigchld_handler: 响应 SIGCHLD. 80 lines]
- sigint_handler: 响应 SIGINT (ctrl-c) 信号. [15 lines]
- sigtstp_handler: 响应 SIGTSTP (ctrl-z) 信号. [15 lines]

eval

源码

```
/*
 * eval - Evaluate the command line that the user has just typed in
 *
 * If the user has requested a built-in command (quit, jobs, bg or fg)
 * then execute it immediately. Otherwise, fork a child process and
 * run the job in the context of the child. If the job is running in
 * the foreground, wait for it to terminate and then return.  Note:
 * each child process must have a unique process group ID so that our
 * background children don't receive SIGINT (SIGTSTP) from the kernel
 * when we type ctrl-c (ctrl-z) at the keyboard.
*/
void eval(char *cmdline)
{
    return;
}
```

题解

IDA 学习法，这个 eval 函数书上第 525 页有简单示例，可以先试试书上的示例加深理解

1. 注意每个子进程必须拥有自己独一无二的进程组 id，要不然就没有前台后台区分
2. 在 `fork()` 新进程前后要阻塞 SIGCHLD 信号，防止出现竞争（race）这种经典的同步错误

```
void eval(char* cmdline){
    char* argv[MAXARGS];
    int bg;
    pid_t pid;
    sigset_t set;

    bg = parseline(cmdline, argv);  // 解析用户输入的字符串命令
    if(argv[0] == NULL) return;     // 如果嘛也没有，比如回车，就直接退出函数不做处理

    if(!builtin_cmd(argv)){         // 查找用户输入的命令是否为内置命令

```

```
        if(sigemptyset(&set) < 0)   // 初始化信号集合为空(执行成功则返回 0，如果有错误则返回 -1)
            unix_error("sigemptyset error");
        /* *
         * #define  SIGCHLD 17
         * #define  SIGINT  2
         * #define  SIGTSTP 20
         * */
        if(sigaddset(&set, SIGTSTP) || sigaddset(&set, SIGINT) || sigaddset(&set, SIGCHLD))
            unix_error("sigaddset error");

        /* #define  SIG_BLOCK 0 */
        if(sigprocmask(SIG_BLOCK, &set, NULL) < 0)
            unix_error("sigprocmask error");

        /* *
         * 把新建立的进程添加到新的进程组:
         * 当从 bash 运行 tsh 时，tsh 在 bash 前台进程组中运行。
         * 如果 tsh 随后创建了一个子进程，默认情况下，该子进程也将是 bash 前台进程组的成员。
         * 由于输入 ctrl-c 将向 bash 前台组中的每个进程发送一个 SIGINT，
         * 因此输入 ctrl-c 将向 tsh 以及 tsh 创建的每个进程发送一个 SIGINT，这显然是不正确的。
         * 这里有一个解决方案: 在 fork 之后，但在 execve 之前，子进程应该调用 setpgid(0,0)，
         * 这将把子进程放入一个新的进程组中，该进程组的 ID 与子进程的 PID 相同。
         * 这确保 bash 前台进程组中只有一个进程，即 tsh 进程。
         * 当您键入 ctrl-c 时，tsh 应该捕获结果 SIGINT，然后将其转发到适当的前台作业
         * */
        if((pid = fork()) < 0)
            unix_error("fork error");
        else if(!pid){
            /* #define  SIG_UNBLOCK 1 */
            sigprocmask(SIG_UNBLOCK, &set, NULL);  // 解除子进程的阻塞
            if(setpgid(0, 0) < 0)
                unix_error("setpgid error");
            if(execve(argv[0], argv, environ) < 0){
                printf("%s: command not found\n", argv[0]);
                exit(0);
            }
        }

        if(bg == FG){
            addjob(jobs, pid, BG, cmdline);
            sigprocmask(SIG_UNBLOCK, &set, NULL);  // 解除前台子进程的阻塞
        }
        else{
            addjob(jobs, pid, FG, cmdline);
            sigprocmask(SIG_UNBLOCK, &set, NULL);  // 解除(未定义、后台、已停止)子进程的阻塞
            if(!bg){
                if(waitpid(pid, &bg, 0) < 0){
                    unix_error("waiting: waitpid error");
                }
            }
        }

```

```
        if(pid){
            printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);
        }
    }
}
```

builtin_cmd

这个函数没写之前我的 eval 函数的反汇编里没有显示这个函数，应该是被优化掉了

还有就是我的程序没有加上 canary 保护，剩下的反汇编代码基本都长得一样

源码

```
/*
 * builtin_cmd - If the user has typed a built-in command then execute
 *    it immediately.
 */
int builtin_cmd(char **argv)
{
    return 0;     /* not a builtin command */
}
```

题解

在 eval 函数有记录，该函数的作用是判断用户输入的是否是内置函数：是就返回 1，不是就返回 0

```
int builtin_cmd(char** argv){
    // just handle by argv
    if(!strcmp(argv[0], "quit"))
        exit(0);
    else if(!strcmp(argv[0], "jobs"))
        listjobs(jobs);
    else if(!strcmp(argv[0], "bg") || !strcmp(argv[0], "fg"))
        do_bgfg(argv);
    else
        return 0;     /* not a builtin command */
    return 1;
}
```

这里从 IDA 看，对于 bg 和 fg，作者貌似是先匹配 b 和 f 再匹配是否第二个字符是 g 的，这里干脆就全匹配好了

do_bgfg

源码

```c
/*
 * do_bgfg - Execute the builtin bg and fg commands
 */
void do_bgfg(char **argv)
{
    return;
}
```

题解

提前理解一下 kill 函数的用法：向任何进程组或进程发送信号

> int kill(pid_t pid, int sig);
> 参数 pid 的可能选择：
>
> 1. pid 大于零时，pid 是信号欲送往的进程的标识
> 2. pid 等于零时，信号将送往所有与调用 kill() 的那个进程属同一个使用组的进程
> 3. pid 等于 -1 时，信号将送往所有 调用进程 有权给其发送信号 的进程，除了进程 1(init)
> 4. pid 小于 -1 时，信号将送往以 -pid 为组标识的进程。

```c
void do_bgfg(char** argv){
    int id;
    struct job_t* job;
    /* 判断 bg fg 命令后有无参数 */
    if(argv[1] == NULL){
        printf("%s command requires PID or %%jobid argument\n", argv[0]);
        return;
    }
    /* % 纯粹就是可加可不加，加了就过滤掉罢了 */
    if(argv[1][0] == '%'){
        if(argv[1][1] >= '0' && argv[1][1] <= '9'){
            id = atoi(argv[1] + 1);
            job = getjobjid(jobs, id);
            if(job == NULL){
                /* 第一个参数是 % 的情况下，printf 里的 %s 不需要被括号括起来 */
                printf("%s: No such job\n", argv[1]);
                return;
            }
        }
        else{
            printf("%s: argument must be a PID or %%jobid\n", argv[0]);
            return;
        }
    }
    else{
        if(argv[1][0] >= '0' && argv[1][0] <= '9'){
            id = atoi(argv[1]);
            job = getjobjid(jobs, id);
            if(job == NULL){
                printf("(%s): No such process\n", argv[1]);
                return;
            }
        }
        else{
            printf("%s: argument must be a PID or %%jobid\n", argv[0]);
            return;
```

```
        }
    }

    if(!strcmp(argv[0], "bg")){
        /* #define  SIGCONT 18 */
        if(kill(-(job->pid), SIGCONT) < 0)  // 将后台任务唤醒，在后台运行
            puts("kill (bg) error");
        job->state = BG;
        printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);
    }
    else if(!strcmp(argv[0], "fg")){
        if(kill(-(job->pid), SIGCONT) < 0)  // 将后台任务唤醒，在前台运行
            puts("kill (fg) error");
        job->state = FG;
        waitfg(job->pid);  // 等待前台程序运行结束
    }
    else{
        puts("do_bgfg: Internal error");
        exit(0);
    }
}
```

IDA 中显示的源码里写的是 `__printf_chk(1LL, "(%d): No such process\n", job);`

感觉直接用 %s 输出就完了，IDA 里的 job 是通过 `job = strtol(v1, 0LL, 10);` 来的

waitfg

源码

```
/*
 * waitfg - Block until process pid is no longer the foreground process
 */
void waitfg(pid_t pid)
{
    return;
}
```

题解

这个还算简单，具体就是一直睡眠，直到这个进程的标志位不再是前台标志（FG）或者是进程已经被杀死就行

```
void waitfg(pid_t pid){
    struct job_t* job;
    if(pid > 0){
        job = getjobpid(jobs, pid);
        /* Check if any job is there in foreground state */
        while(job != NULL && (job->state == FG)){
            sleep(1);
        }
        if(verbose)
            printf("waitfg: Process (%d) no longer the fg process\n", pid);
    }
}
```

sigint_handler

源码

```
/*
 * sigint_handler - The kernel sends a SIGINT to the shell whenver the
 *    user types ctrl-c at the keyboard.  Catch it and send it along
 *    to the foreground job.
 */
void sigint_handler(int sig)
{
    return;
}
```

题解

SIGINT 就是截获 CTRL+C 然后发给前台程序

```
void sigint_handler(int sig){
    if(verbose)
        puts("sigint_handler: entering");

    pid_t pid = fgpid(jobs);

    if(pid > 0){
        if(kill(-pid, SIGINT) < 0)
            unix_error("kill (sigint) error");
        if(verbose){
            printf("sigint_handler: Job (%d) killed\n", pid);
        }
    }
    if(verbose){
        puts("sigint_handler: exiting");
    }
}
```

这段代码反编译后跟 IDA 中的结果一模一样

sigtstp_handler

源码

```
/*
 * sigtstp_handler - The kernel sends a SIGTSTP to the shell whenever
 *     the user types ctrl-z at the keyboard. Catch it and suspend the
 *     foreground job by sending it a SIGTSTP.
 */
void sigtstp_handler(int sig){
    return;
}
```

题解

SIGTSTP 就是截获 CTRL+Z 然后发给前台程序

```
void sigtstp_handler(int sig){
    if(verbose)
        puts("sigtstp_handler: entering");

    pid_t pid = fgpid(jobs);

    if(pid > 0){
        if(kill(-pid, SIGTSTP) < 0)
            unix_error("kill (tstp) error");
        if(verbose){
            printf("sigtstp_handler: Job [%d] (%d) stopped\n",
pid2jid(pid), pid);
        }
    }
    if(verbose){
        puts("sigtstp_handler: exiting");
    }
}
```

sigchld_handler

源码

```
/*
 * sigchld_handler - The kernel sends a SIGCHLD to the shell whenever
 *     a child job terminates (becomes a zombie), or stops because it
 *     received a SIGSTOP or SIGTSTP signal. The handler reaps all
 *     available zombie children, but doesn't wait for any other
 *     currently running children to terminate.
 */
void sigchld_handler(int sig)
{
    return;
}
```

题解

```
void sigchld_handler(int sig)
{
    int status, jid;
    pid_t pid;
    struct job_t *job;

    if(verbose)
        puts("sigchld_handler: entering");

    /* *
     * 以非阻塞方式等待所有子进程
     * waitpid 参数3:
     *    1.     0     : 执行waitpid时,  只有在子进程 **终止** 时才会返回。
     *    2. WNOHANG   : 若子进程仍然在运行，则返回0
     *                   只有设置了这个标志，waitpid 才有可能返回 0
```

```
     *    3. WUNTRACED : 如果子进程由于传递信号而停止，则马上返回。
     *                   只有设置了这个标志，waitpid 返回时其 WIFSTOPPED(status) 才有可能返回 true
     * */
    while((pid = waitpid(-1, &status, WNOHANG | WUNTRACED)) > 0){

        // 如果当前这个子进程的 job 已经删除了，则表示有错误发生
        if((job = getjobpid(jobs, pid)) == NULL){
            printf("Lost track of (%d)\n", pid);
            return;
        }

        jid = job->jid;
        // 如果这个子进程收到了一个暂停信号（还没退出
        if(WIFSTOPPED(status)){
            printf("Job [%d] (%d) stopped by signal %d\n", jid, job->pid, WSTOPSIG(status));
            job->state = ST;
        }
        // 如果这个子进程正常退出
        else if(WIFEXITED(status)){
            if(deletejob(jobs, pid))
                if(verbose){
                    printf("sigchld_handler: Job [%d] (%d) deleted\n", jid, pid);
                    printf("sigchld_handler: Job [%d] (%d) terminates OK (status %d)\n", jid, pid, WEXITSTATUS(status));
                }
        }
        // 如果这个子进程因为其他的信号而异常退出，例如 SIGKILL
        else {
            if(deletejob(jobs, pid)){
                if(verbose)
                    printf("sigchld_handler: Job [%d] (%d) deleted\n", jid, pid);
            }
            printf("Job [%d] (%d) terminated by signal %d\n", jid, pid, WTERMSIG(status));
        }
    }

    if(verbose)
        puts("sigchld_handler: exiting");
}
```

问题汇总

问题一（已解决）

> ctrl + c 后会出现如下错误提示：`tsh> ^Ckill (sigint) error: No such process`
>
> 但是 `sigint_handler` 函数没有问题，暂且不知道问题在哪

最后发现是因为还没有写 sigchld_handler 函数

参考链接

https://www.jianshu.com/p/f7054d98c6b8