Shlab

2021-12-23T19:33:42+08:00

## Shell Lab

### CSAPP Chapter8 Exceptional Control Flow

**异常分类**:

Trap是用户态向内核请求服务的过程，提供syscall n接口，即系统调用。  
  
系统为异常分配了唯一的非负整数异常号，系统启动时，操作系统初始化一张异常表，起始地址存放在异常表基址寄存器中。异常处理流程如下:

**并发流、并行流:**

* 逻辑流在执行时间上有重叠即为并发流
* 运行在不同处理器核上的并发流为并行流

pid\_t fork(void);

* 创建父进程的副本，返回两次，若为0则在子进程中，反之在父进程中。
* 共享文件描述符

pid\_t waitpid(pid\_t pid, int \*statusup, int options);

* 等待进程号为pid的子进程执行结束，waitpid(pid, NULL, 0), statusup、options翻阅man文档
* status
  + WIFEXITED(status): 如果子进程通过调用exit 或者一个返回 (return) 正常终止，就返回真。
  + WEXITSTATUS (status): 返回一个正常终止的子进程的退出状态。只有在WIFEXITED() 返回为真时，才会定义这个状态。
  + WIFSIGNALED(status): 如果子进程是因为一个未被捕获的信号终止的，那么就返回真。
  + WTERMSIG(status): 返回导致子进程终止的信号的编号。只有在WIFSIGNALED() 返回为真时，才定义这个状态。
  + WIFSTOPPED(status): 如果引起返回的子进程当前是停止的，那么就返回真。
  + WSTOPSIGC(status): 返回引起子进程停止的信号的编号。只有在 WIFSTOPPED返回为真时，才定义这个状态。
  + WIFCONTINUED(status): 如果子进程收到 SIGCONT 信号重新启动，则返回真。
* return  
    
  如果调用进程没有子进程，那么 waitpid 返回 -1, 并且设置 errno 为 ECHILD 。如果 waitpid 函数被一个信号中断，那么它返回 -1,并设置 errno 为 EINTR 。

命令行参数、环境变量参数示意图:

全局变量 envrion 指向envp[0]

**信号**  
信号机制基于进程组，每个进程都属于一个进程组

pid\_t getpgrp(void);  
int setpgrp(pid\_t pid, pid\_t pgid);

* 获取进程组、修改进程组
  + pid = 0 -> pid = current pid
  + pgid = 0 -> pgid = pid
* setpgrp(0, 0) 将当前pid设置为pgid

发送信号:  
  
int kill(pid\_t pid, int sig);

* pid > 0, sig to pid
* pid = 0, sig to pids in group of pid
* pid < 0, sig to pids where |pid| in

unsigned int alarm(unsigned int secs);

* 向自己发送SIGALRM信号

接受信号:

* pending: 待处理信号向量
* blocked: 被阻塞的信号向量  
  当发送上下文切换或系统调用返回时，会检查(pending & ~blocked)，选择待处理信号最小的信号接收，如果为空则Inext

#include <signal.h>  
typedef void (\*sighandler\_t)(int);  
sighandler\_t signal(int signum, sighandler\_t handler);

* if handler == SIG\_IGN, ignore signum
* if handler == SIG\_DFL, restore signum
* else handler 为用户自定义信号处理函数

handler 可以被 handler中断，即在handler中处理handler

信号处理流程:

**阻塞、解除阻塞信号**

#include <signal.h>  
// success -> 0, error -> -1  
int sigprocmask(int how, const sigset\_t \*set, sigset\_t \*oldset);  
int sigemptyset(sigset\_t \*set);  
int sigfillset(sigset\_t \*set);  
int sigaddset(sigset\_t \*set, int signum);  
int sigdelset(sigset\_t \*set, int signum);  
  
// if signum in set -> 1, else -> 0, error -> -1  
int sigismember(const sigset\_t \*set, int signum);  
//

* how
  + SIG\_BLOCK: blocked = blocked | set
  + SIG\_UNBLOCK: blocked = blocked & ~set
  + SIG\_SETMASK: blocked = set

### Task1

完成trace01、trace02  
  
根据注释可知，输入的每条指令的过程为 eval -> parseline -> execve  
  
trace01为读取txt至eof结束，样例代码已经实现  
trace02为读取内置quit命令退出，实现如下

int builtin\_cmd(char \*\*argv) {  
 if (!strcmp(argv[0], "quit"))  
 exit(0);  
 return 0; /\* not a builtin command \*/  
}

### Task2 Task3

完成trace03验证quit命令、trace04验证后台作业。  
  
代码根据CSAPP 8.4.6修改  
  
遇到非built-in命令，fork一个子进程执行execve命令, environ为全局变量，存储系统环境变量。  
  
myspin是根据第一个参数秒数的睡眠程序。

运行foreground命令时，主程序等待其执行结束再return。

void eval(char \*cmdline) {  
 char \*argv[MAXARGS]; /\* Argument list execve() \*/  
 char buf[MAXLINE]; /\* Holds modified command line \*/  
 int bg; /\* Should the job run in bg or fg \*/  
 pid\_t pid; /\* Process id \*/  
  
 strcpy(buf, cmdline);  
 bg = parseline(buf, argv);  
 if (argv[0] == NULL)  
 return; /\* Ignore empty lines \*/  
  
 if (!builtin\_cmd(argv)) {  
 if ((pid = fork()) == 0) { /\* Child runs user job \*/  
 if (execve(argv[0], argv, environ) < 0) {  
 unix\_error("execve: ");  
 }  
 }  
  
 addjob(jobs, pid, bg ? BG : FG, cmdline);  
 /\* Parent waits for foreground job to terminate \*/  
 if (!bg) {  
 int status;  
 if (waitpid(pid, &status, 0) < 0)  
 unix\_error("waitfg: waitpid error");  
 } else  
 printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);  
 }  
 return;  
}

main函数中已经为我们注册好4个信号.

/\* These are the ones you will need to implement \*/  
Signal(SIGINT, sigint\_handler); /\* ctrl-c \*/  
Signal(SIGTSTP, sigtstp\_handler); /\* ctrl-z \*/  
Signal(SIGCHLD, sigchld\_handler); /\* Terminated or stopped child \*/  
  
/\* This one provides a clean way to kill the shell \*/  
Signal(SIGQUIT, sigquit\_handler);

测试过程中发现跟make rtest4/5有出入，根据pdf文档描述，需要解决jobs回收问题。

**hints5**  
  
One of the tricky parts of the assignment is deciding on the allocation of work between the waitfg and sigchld handler functions. We recommend the following approach:

* In waitfg, use a busy loop around the sleep function.
* In sigchld handler, use exactly one call to waitpid.

void waitfg(pid\_t pid) {  
 // hints5: busy loop  
 while (pid == fgpid(jobs))  
 sleep(1);  
 return;  
}  
  
  
void sigchld\_handler(int sig) {  
 pid\_t pid;  
 int status;  
 /\*\* csapp 8.4.3  
 WNOHANG | WUNTRACED  
 立即返回，如果等待集合中的子进程都没有被停 止或终止，则返回值为 0;  
 如果有一个停止或终止，则返回值为该子进程的 PID 。  
 \*/  
 while ((pid = waitpid(pid, &status, WNOHANG | WUNTRACED)) > 0) {  
 // WIFEXITED(status): 如果子进程通过调用 exit 或者一个返回(return ) 正常终  
 // 止，就返回真。   
 if (WIFEXITED(status))   
 deletejob(jobs, pid);  
 }  
 return;  
}

根据hints6, 父进程需要在创建子进程前阻塞SIGCHLD信号，防止子进程已经执行结束，在addjob后恢复信号。 由于fork后子进程拷贝了父进程的blocked信号，子进程中也需要恢复。

**hints6**  
  
In eval, the parent must use sigprocmask to block SIGCHLD signals before it forks the child, and then unblock these signals, again using sigprocmask after it adds the child to the job list by calling addjob. Since children inherit the blocked vectors of their parents, the child must be sure to then unblock SIGCHLD signals before it execs the new program.  
  
The parent needs to block the SIGCHLD signals in this way in order to avoid the race condition where the child is reaped by sigchld handler (and thus removed from the job list) before the parent calls addjob.

修改后的eval:

void eval(char \*cmdline) {  
 char \*argv[MAXARGS]; /\* Argument list execve() \*/  
 char buf[MAXLINE]; /\* Holds modified command line \*/  
 int bg; /\* Should the job run in bg or fg \*/  
 pid\_t pid; /\* Process id \*/  
 sigset\_t mask;  
  
 strcpy(buf, cmdline);  
 bg = parseline(buf, argv);  
 if (argv[0] == NULL)  
 return; /\* Ignore empty lines \*/  
  
 if (!builtin\_cmd(argv)) {  
 // 初始化  
 sigemptyset(&mask);  
 // 添加SIGCHLD到mask集合中  
 sigaddset(&mask, SIGCHLD);  
 // 屏蔽mask集合中的信号  
 sigprocmask(SIG\_BLOCK, &mask, NULL);  
 if ((pid = fork()) == 0) { /\* Child runs user job \*/  
 // 子进程中unblock mask集合  
 sigprocmask(SIG\_UNBLOCK, &mask, NULL);  
 if (execve(argv[0], argv, environ) < 0) {  
 unix\_error("execve: ");  
 }  
 }  
  
 addjob(jobs, pid, (bg ? BG : FG), cmdline);  
 // 父进程unblock mask集合  
 sigprocmask(SIG\_UNBLOCK, &mask, NULL);  
 /\* Parent waits for foreground job to terminate \*/  
 if (!bg) {  
 waitfg(pid);  
 } else {  
 printf("[%d] (%d) %s", pid2jid(pid), pid, cmdline);  
 }  
 }  
 return;  
}

**hint8**  
  
在linux下运行tsh时，tsh运行在一个foreground group中，ctrl c会将SIGINT信号发送个整个group，根据上面笔记可知，setpgid(0, 0)可以将当前进程号设置为gid并切换进程组。  
所以需要在子进程中添一行setpgid(0, 0)。

### task6

trace06、trace07  
  
Forward SIGINT to foreground job, 即将crtl + c信号转发给tsh的**前台进程**，fgpid获取前台进程pid。同时需要在sigchld\_handler中处理因信号退出产生的status。

-void sigint\_handler(int sig) { return; }  
+void sigint\_handler(int sig) {  
+ pid\_t pid = fgpid(jobs);  
+ if (pid) {  
+ if (kill(-pid, SIGINT)) {  
+ unix\_error("kill sigint: ");  
+ }  
+ }  
+ return;  
+}  
  
@@ -311,6 +312,12 @@ void sigchld\_handler(int sig) {  
 if (WIFEXITED(status)) { /\* process terminated normaly \*/  
 deletejob(jobs, pid);  
 }  
+  
+ if (WIFSIGNALED(status)) {  
+ printf("Job [%d] (%d) terminated by signal %d\n", pid2jid(pid), pid,  
+ WTERMSIG(status));  
+ deletejob(jobs, pid);  
+ }

### task7

trace08  
  
Forward SIGTSTP only to foreground job.  
将SIGTSTP信号转发给前台作业，中断当前作业，流程与task6相似。

@@ -318,6 +318,12 @@ void sigchld\_handler(int sig) {  
 WTERMSIG(status));  
 deletejob(jobs, pid);  
 }  
+  
+ if (WIFSTOPPED(status)) {  
+ printf("Job [%d] (%d) stopped by signal %d\n", pid2jid(pid), pid,  
+ WSTOPSIG(status));  
+ getjobpid(jobs, pid)->state = ST;  
+ }  
  
-void sigtstp\_handler(int sig) { return; }  
+void sigtstp\_handler(int sig) {  
+ pid\_t pid = fgpid(jobs);  
+ if (pid) {  
+ if (kill(-pid, SIGTSTP) < 0)  
+ unix\_error("kill sigtstp: ");  
+ }  
+ return;  
+}

### task7

* trace09: 实现内建bg命令  
    
  The bg <job> command restarts <job> by sending it a SIGCONT signal, and then runs it in  
  the background. The <job> argument can be either a PID or a JID.

bg参数可以为pid / jid

* 第一步：在builtin\_cmd中加入bg指令
* 第二步：填充do\_bgfg()
  + 解析参数以及参数是否符合bg pid / %jid要求
  + 根据pid、jid获取job结构体指针
  + 将SIGCONT信号发送给job所在的进程组
  + 修改job的state为BG

@@ -266,6 +266,9 @@ int builtin\_cmd(char \*\*argv) {  
 } else if (!strcmp(argv[0], "jobs")) {  
 listjobs(jobs);  
 return 1;  
+ } else if (!strcmp(argv[0], "bg")) {  
+ do\_bgfg(argv);  
+ return 1;  
  
+void do\_bgfg(char \*\*argv) {  
+ struct job\_t \*job;  
+ // 解析参数  
+ if (!argv[1]) {  
+ app\_error("lack jid\n");  
+ }  
+  
+ if (!isdigit(argv[1][0]) && argv[1][0] != '%') {  
+ app\_error("error usage: bg pid/%(jid) ");  
+ }  
+  
+ // 判断为pid / jid  
+ int flag = isdigit(argv[1][0]) ? 1 : 0;  
+ if (flag) {  
+ int pid = atoi(argv[1]);  
+ job = getjobpid(jobs, pid);  
+ if (!job)  
+ app\_error("no such pid");  
+ } else {  
+ int jid = atoi(argv[1] + 1);  
+ job = getjobjid(jobs, jid);  
+ if (!job)  
+ app\_error("no such jid");  
+ }  
+ // 转发信号  
+ kill(-job->pid, SIGCONT);  
+  
+ if (!strcmp(argv[0], "bg")) {  
+ job->state = BG;  
+ printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);  
+ }  
+ return;  
+}

* trace10: 实现内建fg命令  
  fg命令实现方式与bg相似，在builtin\_cmd中添加fg指令，只是需要waitfg()等待前台进程结束。

@@ -269,6 +269,9 @@ int builtin\_cmd(char \*\*argv) {  
 } else if (!strcmp(argv[0], "bg")) {  
 do\_bgfg(argv);  
 return 1;  
+ } else if (!strcmp(argv[0], "fg")) {  
+ do\_bgfg(argv);  
+ return 1;  
 } else if (!strcmp(argv[0], "&"))  
 return 1;  
 return 0; /\* not a builtin command \*/  
@@ -308,6 +311,11 @@ void do\_bgfg(char \*\*argv) {  
 job->state = BG;  
 printf("[%d] (%d) %s", job->jid, job->pid, job->cmdline);  
 }  
+  
+ if (!strcmp(argv[0], "fg")) {  
+ job->state = FG;  
+ waitfg(job->pid);  
+ }  
 return;  
 }

### task8

trace11-16验证

* trace11: 测试SIGINT是否对前台进程组的进程有效
* trace12: 测试SIGTSTP是否对前台进程组有效
* trace13: 测试restart被SIGTSTP暂停的进程

unix下执行ps命令中进程STAT状态格式如下:

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:  
 D uninterruptible sleep (usually IO)  
 R running or runnable (on run queue)  
 S interruptible sleep (waiting for an event to complete)  
 T stopped, either by a job control signal or because it is being traced.  
 W paging (not valid since the 2.6.xx kernel)  
 X dead (should never be seen)  
 Z defunct ("zombie") process, terminated but not reaped by its parent.  
  
 For BSD formats and when the stat keyword is used, additional characters may be displayed:  
 < high-priority (not nice to other users)  
 N low-priority (nice to other users)  
 L has pages locked into memory (for real-time and custom IO)  
 s is a session leader  
 l is multi-threaded (using CLONE\_THREAD, like NPTL pthreads do)  
 + is in the foreground process group.

trace14: 错误处理  
  
自定义用宏打印错误并退出

#define ERROR\_RETURN(msg, ...) \  
 do { \  
 printf(msg, ##\_\_VA\_ARGS\_\_); \  
 return; \  
 } while (0)

void do\_bgfg(char \*\*argv) {  
 struct job\_t \*job;  
 if (argv[1] == NULL) {  
- app\_error("lack jid\n");  
+ ERROR\_RETURN("%s command requires PID or %%jobid argument\n", argv[0]);  
 }  
   
 if (!isdigit(argv[1][0]) && argv[1][0] != '%') {  
- app\_error("error usage: bg pid/%(jid) ");  
+ ERROR\_RETURN("%s: argument must be a PID or %%jobid\n", argv[0]);  
 }  
   
 // 判断为pid / jid  
@@ -295,17 +301,19 @@ void do\_bgfg(char \*\*argv) {  
 if (flag) {  
 int pid = atoi(argv[1]);  
 job = getjobpid(jobs, pid);  
- if (!job)  
- app\_error("no such pid");  
+ if (job == NULL) {  
+ ERROR\_RETURN("(%s): no such process\n", argv[1]);  
+ }  
 } else {  
 int jid = atoi(argv[1] + 1);  
 job = getjobjid(jobs, jid);  
- if (!job)  
- app\_error("no such jid");  
+ if (job == NULL) {  
+ ERROR\_RETURN("%s: No such job\n", argv[1]);  
+ }  
 }

trace15: 整合测试  
  
trace16: 测试从其它进程发出sigint / sigtstp信号对tsh是否有效(见mystop.c)。  
由mytop.c代码可见，其发送SIGTSTP给当前进程组。

pid = getpid();  
if (kill(-pid, SIGTSTP) < 0)  
 fprintf(stderr, "kill (tstp) error");

### 拓展题目

#### A2

使用vi编辑器编写一段shell文件，取名为mycal实现与Linux中cal命令相当的功能：当输入$mycal[月份名]时，屏幕输出指定年月的日历。  
  
使用内置cal命令，并且传递两个参数月份、年份。

# mycal\_A2.sh  
  
#!/bin/bash  
cal $1 $2

#### B3

使用vim编辑脚本实现把当前文件目录的信息输出到filedir.txt中。  
  
使用内置ls命令，通过重定向输出到filedir.txt文件中。

# filedir\_B3.sh  
  
#!/bin/bash  
ls -la > filedir.txt

#### C1

使用vim编写shell脚本实现随机产生10个100以内的整数，并输出其中能被7整除的数。并为其编写一个Makefile文件。

# Makefile  
rand:  
 ./random\_C1.sh  
  
# random\_C1.sh  
#!/bin/bash  
for i in {1..10}  
do  
 echo $(expr $RANDOM % 100);  
done

#### D2

利用互斥锁pthread\_mutex\_t编写互斥访问临界资源的多线程程序。程序完成的功能要求：主函数初始化共享内存变量mv(初值为10)，创建互斥锁，创建两个线程并等待两个线程执行完毕，之后输出mv的值并释放互斥锁；两个线程分别实现通过获取互斥锁的方式完成对内存变量mv自加10次和自减5次的功能。

注意返回值检查， mutex通过静态初始化不需要调用pthread\_mutex\_init();

#include <pthread.h>  
#include <stdio.h>   
#include <stdlib.h>   
  
  
static pthread\_mutex\_t mtx = PTHREAD\_MUTEX\_INITIALIZER;  
static int mv = 10;  
  
void errExit(const char\* msg) {  
 printf("%s", msg);  
 exit(EXIT\_FAILURE);  
}  
void \*thread\_increment(void \*args) {  
 for (int i = 0; i < 10; i++) {  
 if (pthread\_mutex\_lock(&mtx) < 0) errExit("lock error.\n");  
 mv++;  
 if (pthread\_mutex\_unlock(&mtx)) errExit("unlock error.\n");  
 }  
 return NULL;  
}  
  
  
void \*thread\_decrement(void \*args) {  
 for (int i = 0; i < 5; i++) {  
 if (pthread\_mutex\_lock(&mtx) < 0) errExit("lock error.\n");  
 mv--;  
 if (pthread\_mutex\_unlock(&mtx)) errExit("unlock error.\n");  
 }  
 return NULL;  
}  
  
int main(int argc, char \*\*argv) {  
 pthread\_t t1, t2;  
 if (pthread\_create(&t1, NULL, thread\_increment, NULL) < 0) errExit("create error.\n");  
 if (pthread\_create(&t2, NULL, thread\_decrement, NULL) < 0) errExit("create error.\n");  
 if (pthread\_join(t1, NULL) < 0) errExit("join error.\n");  
 if (pthread\_join(t2, NULL) < 0) errExit("join error.\n");  
 printf("mv: %d\n", mv);  
 if (pthread\_mutex\_destroy(&mtx) < 0) errExit("mutex destroy error.\n");  
 return 0;  
}

#### E2

要求编写函数实现以下功能:

/\* Return 1 when any odd bit of x equals 1; 0 otherwise.  
Assume W=32 \*/  
int any \_odd\_one (unsigned x);

函数应该遵循位级整数编码规则，不过你可以假设数据类型int有w= 32位.

int any \_odd\_one(unsigned x) {
  
 int odd = (0xaa << 24) + (0xaa << 16) + (0xaa << 8) + 0xaa;
  
 return !((x & odd) ^ odd);
  
}