

第二次作业

2.1

所写 C 程序 2_1.c 如下:

```
#include <stdio.h>
#include <stdlib.h>
int a = 1;
int b;
int main(){
    int *array = malloc(sizeof(int) * 2);
    array[0] = a;
    array[1] = b;
    printf("array[0] = %d\n", array[0]);
    printf("array[1] = %d\n", array[1]);
    free(array);
    return 0;
}
```

其中,已初始化的全局变量 `a` 在 `data` 段,未初始化的全局变量 `b` 在 `bss` 段,使用 `malloc` 函数分配的内存空间在堆区,即 `array` 指向的内存空间在堆区。

`readelf` 是一个用于显示 ELF (Executable and Linkable Format) 文件信息的工具。它通常用于分析和调试 ELF 文件。`objdump` 是另一个用于分析和调试二进制文件的工具。它可以显示二进制文件的各种信息,类似于 `readelf`,但功能更为广泛。

在 Linux 终端输入命令:

```
readelf -S 2_1.o
```

输出了各段的信息,其中 `data` 段和 `bss` 段如下:

[3]	.data	PROGBITS	0000000000000000	000000c8
	0000000000000004	0000000000000000	WA 0 0	4
[4]	.bss	NOBITS	0000000000000000	000000cc
	0000000000000004	0000000000000000	WA 0 0	4

图 1. `data` 段和 `bss` 段

再输入命令:

```
objdump -D 2_1.o
```

输出了各段的汇编代码,其中 `data` 段和 `bss` 段如下:

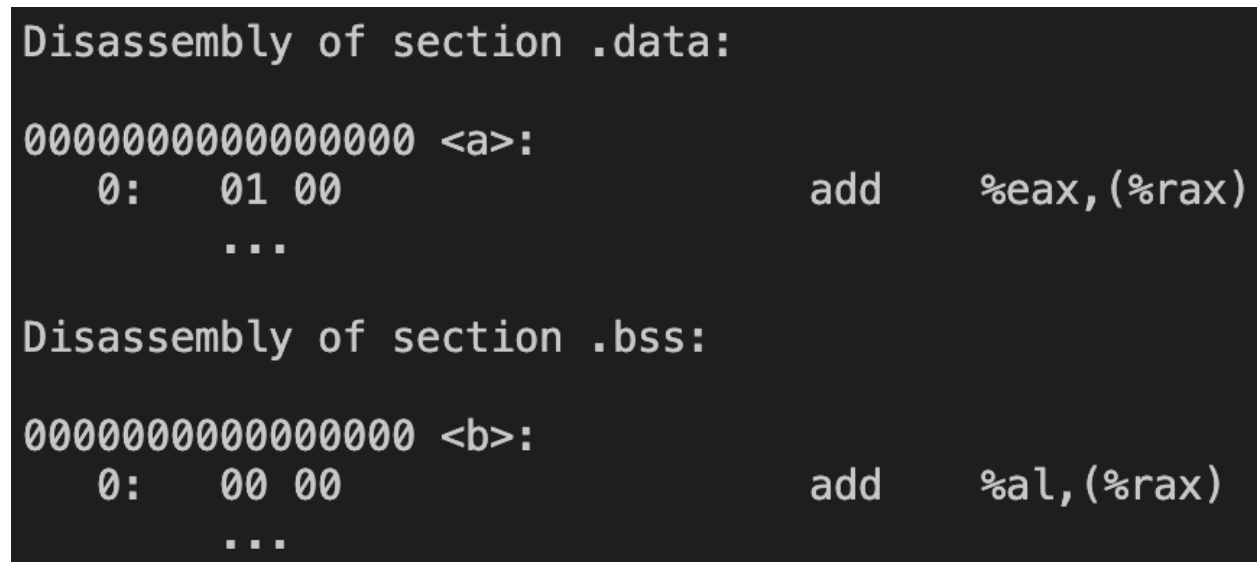


图 2. data 段和 bss 段

该 C 程序用到了栈。在调用 `printf` 函数时,程序会将 `array[0]` 和 `array[1]` 的值压入栈中,然后调用 `printf` 函数。具体来说,`printf` 函数的参数会被依次压入栈中,函数调用时会创建一个新的栈帧来保存返回地址和局部变量。函数执行过程中,`printf` 函数会从栈中取出这些参数值进行打印。函数执行完毕后,栈帧会被销毁,栈指针恢复到调用前的状态。

2.2

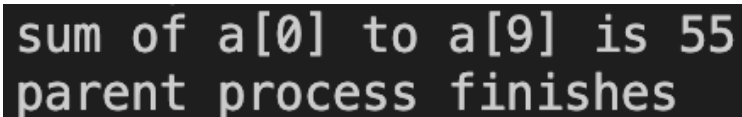
(1) 所写 C 程序 `2_2_1.c` 如下:

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

int main(){
    int a[10] = {1,2,3,4,5,6,7,8,9,10};
    pid_t pid = fork();
    if (pid < 0){
        printf("fork failed\n");
        return 1;
    }
    if (pid == 0)
        printf("sum of a[0] to a[9] is %d\n",
            a[0]+a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7]+a[8]+a[9]);
    else{
        wait(NULL);
        printf("parent process finishes\n");
    }
}
```

```
    return 0;  
}
```

输出如下:



```
sum of a[0] to a[9] is 55  
parent process finishes
```

图 3.2_2_1.c 输出

代码解释: 先用 `fork()` 函数创建一个子进程, 子进程中计算数组 `a` 的和, 父进程使用 `wait()` 函数等待子进程结束, 然后输出 `parent process finishes`。

(2) 所写 C 程序 2_2_2.c 如下:

```
#include <stdio.h>  
#include <unistd.h>  
#include <sys/wait.h>  
  
int main(){  
    int a[10] = {1,2,3,4,5,6,7,8,9,10};  
    pid_t pid = fork();  
    if (pid < 0)  
    {  
        printf("fork failed\n");  
        return 1;  
    }  
    if (pid == 0)  
    {  
        printf("sum of a[0] to a[9] is %d\n",  
            a[0]+a[1]+a[2]+a[3]+a[4]+a[5]+a[6]+a[7]+a[8]+a[9]);  
        execlp("ls", "ls", "-l", "/usr/lib", NULL);  
    }  
    else  
    {  
        wait(NULL);  
        printf("parent process finishes\n");  
    }  
    return 0;  
}
```

输出如下:

```

sum of a[0] to a[9] is 55
total 8
drwxr-xr-x 1 root root 98 Mar 17 15:44 apt
drwxr-xr-x 1 root root 32 Mar 20 16:58 bfd-plugins
drwxr-xr-x 1 root root 30 Mar 17 15:44 binfmt.d
drwxr-xr-x 1 root root 4 Mar 20 16:38 compat-ld
drwxr-xr-x 1 root root 66 Mar 17 15:43 console-setup
lrwxrwxrwx 1 root root 21 Feb 17 2023 cpp -> /etc/alternatives/cpp
drwxr-xr-x 1 root root 50 Mar 17 15:43 dbus-1.0
drwxr-xr-x 1 root root 14 Mar 17 15:43 dpkg
drwxr-xr-x 1 root root 38 Mar 17 15:44 environment.d
drwxr-xr-x 1 root root 18 Sep 2 22:45 file
drwxr-xr-x 1 root root 32 Mar 20 16:58 gcc
drwxr-xr-x 1 root root 4468 Mar 20 16:55 git-core
drwxr-xr-x 1 root root 160 Sep 2 22:41 gnupg
drwxr-xr-x 1 root root 74 Sep 2 22:41 gnupg2
drwxr-xr-x 1 root root 4 Mar 20 16:38 gold-ld
drwxr-xr-x 1 root root 40 Mar 17 15:43 init
drwxr-xr-x 1 root root 6 Mar 17 15:43 initramfs-tools
drwxr-xr-x 1 root root 42 Mar 17 15:44 kernel
drwxr-xr-x 1 root root 40 Mar 17 15:44 locale
drwxr-xr-x 1 root root 60 Mar 17 15:43 lsb
drwxr-xr-x 1 root root 16 Jan 15 2023 mime
drwxr-xr-x 1 root root 88 Mar 17 15:44 modprobe.d
drwxr-xr-x 1 root root 624 Aug 28 22:54 modules
drwxr-xr-x 1 root root 0 Mar 20 2023 modules-load.d
drwxr-xr-x 1 root root 16 Mar 17 15:44 netplan
drwxr-xr-x 1 root root 110 Mar 17 15:43 networkd-dispatcher
drwxr-xr-x 1 root root 106 Mar 17 15:44 openssh
-rw-r--r-- 1 root root 389 Jan 3 2024 os-release
drwxr-xr-x 1 root root 24 Mar 17 15:44 pam.d
drwxr-xr-x 1 root root 0 Jan 26 2023 pkgconfig
drwxr-xr-x 1 root root 26 Mar 17 15:43 python3
drwxr-xr-x 1 root root 4196 Mar 20 16:59 python3.11
drwxr-xr-x 1 root root 74 Mar 17 15:43 rsyslog
drwxr-xr-x 1 root root 0 Nov 23 2022 sasl2
drwxr-xr-x 1 root root 70 Mar 17 15:44 ssl
drwxr-xr-x 1 root root 72 Mar 17 15:44 sysctl.d
drwxr-xr-x 1 root root 2064 Mar 17 15:44 systemd
drwxr-xr-x 1 root root 200 Mar 17 15:44 sysusers.d
drwxr-xr-x 1 root root 28 Jan 26 2023 terminfo
drwxr-xr-x 1 root root 532 Mar 17 15:44 tmpfiles.d
drwxr-xr-x 1 root root 332 Mar 17 15:44 ubuntu-advantage
drwxr-xr-x 1 root root 176 Mar 17 15:44 udev
drwxr-xr-x 1 root root 74 Mar 17 15:43 usrmerge
drwxr-xr-x 1 root root 24 Mar 17 15:43 valgrind
drwxr-xr-x 1 root root 13226 Sep 2 22:45 x86_64-linux-gnu
parent process finishes
zhangjiawei@ubuntu1: /Users/zhangjiawei/Desktop/Operating System/hw2$

```

图 4.2_2.c 输出

代码解释：先用 `fork()` 函数创建一个子进程，子进程中计算数组 `a` 的和，然后使用 `execlp()` 函数调用 `ls -l /usr/lib` 命令，父进程使用 `wait()` 函数等待子进程结束，然后输出 `parent process finishes`。

(3) 对 PCB 的定义代码在/xv6-riscv/kernel/proc.h 中,如下:

```
// Per-process state
struct proc {
    struct spinlock lock;

    // p->lock must be held when using these:
    enum procstate state; // Process state
    void *chan;           // If non-zero, sleeping on chan
    int killed;           // If non-zero, have been killed
    int xstate;           // Exit status to be returned to parent's wait
    int pid;              // Process ID

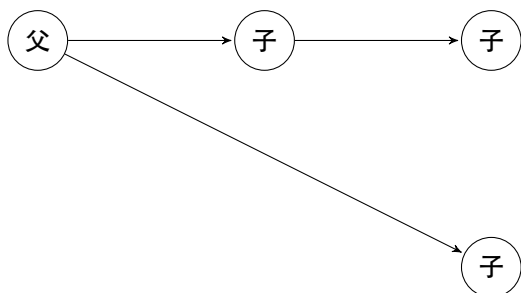
    // wait_lock must be held when using this:
    struct proc *parent; // Parent process

    // these are private to the process, so p->lock need not be held.
    uint64 kstack;        // Virtual address of kernel stack
    uint64 sz;            // Size of process memory (bytes)
    pagetable_t pagetable; // User page table
    struct trapframe *trapframe; // data page for trampoline.S
    struct context context; // swtch() here to run process
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;      // Current directory
    char name[16];         // Process name (debugging)
};
```

在执行 fork 函数时,使用 np 指向子进程的 PCB, 首先为子进程分配一个新的 PCB, 然后将父进程的用户内存复制到子进程的用户内存, 接着将父进程的用户寄存器状态复制到子进程, 即将父进程的陷阱帧的内容复制到子进程的陷阱帧中, 然后确保 fork 系统调用在子进程中返回 0, 即设置子进程的陷阱帧中的寄存器 a0 为 0, 最后复制父进程的文件描述符、当前工作目录、进程名称, 设置子进程 pid, 释放子进程锁, 设置指向父进程的指针, 设置子进程状态为 RUNNABLE, 最后返回子进程的 pid。

2.3

(1) 一共会生成 3 个子进程。关系图如下:



第一次循环中,父进程生成第一个子进程,第二次循环中,父进程再生成一个子进程,第一个子进程又生成一个子进程,共生成 3 个子进程。

(2) 只需要使得生成的子进程不再调用 `fork()` 函数即可。修改后的代码如下:

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#define LOOP 2

int main(int argc, char *argv[])
{
    pid_t pid;
    int loop;
    for (loop = 0; loop < LOOP; loop++)
    {
        if ((pid = fork()) < 0)
            fprintf(stderr, "fork failed\n");
        else if (pid == 0)
        {
            printf(" I am child process\n");
            break;
        }
        else
        {
            sleep(5);
        }
    }
    return 0;
}
```

这样修改使得子进程在生成后直接退出循环,不再调用 `fork()` 函数,从而所有的子进程都由父进程生成,循环次数即为生成的子进程数。