

作业2

[姓名:蔡合森] [学号:2022K8009909004]

2.1 一个C程序可以编译成目标文件或可执行文件。目标文件和可执行文件通常包含text、data、bss、rodata段，程序执行时也会用到堆(heap)和栈(stack)。

(1) 请写一个C程序，使其包含data段和bss段，并在运行时包含堆的使用。请说明所写程序中哪些变量在data段、bss段和堆上。

(2) 请了解 readelf、objdump 命令的使用，用这些命令查看(1)中所写程序的data和bss段，截图展示。

(3) 请说明(1)中所写程序是否用到了栈。

提交内容：所写C程序、问题解答、截图等。

解：

(1)

```

#include<stdio.h>

//rodata
#define MAX 100
char *p= "12345"

//bss
int var_global;
int var_global_init = 64;

//text

void Print(int i){
    printf("%d\n",i);
}

int main(){
    int x =1;
    int y;

    //heap
    char arr[MAX];
    arr=(char*)malloc(sizeof(char)*MAX);
    free(arr);

    return 0;

}

```

data段：通常是指用来存放程序中已初始化的全局变量和已初始化的静态变量的一块内存区域

```

int var_global_init = 64;
int x =1;

```

bbs:通常是指用来存放程序中未初始化的全局变量和未初始化的局部静态变量

```

int var_global;
int y;

```

堆：堆是用于存放进程运行中被动态分配的内存段

```
char arr[MAX];
arr=(char*)malloc(sizeof(char)*MAX);
free(arr);
```

(2) objdump 命令截图如下:

		CONTENTS, ALLOC, LOAD, DATA			
24	.data	00000020	00000000000004000	00000000000004000	00003000 2**3
		CONTENTS, ALLOC, LOAD, DATA			
25	.bss	00000008	00000000000004020	00000000000004020	00003020 2**2
		ALLOC			

readelf 命令截图如下:

[25]	.data	PROGBITS	00000000000004000	00003000	
		00000000000000020	00000000000000000	WA	0 0 8
[26]	.bss	NOBITS	00000000000004020	00003020	
		00000000000000008	00000000000000000	WA	0 0 4

(3)使用到了栈，用户存放程序临时创建的局部变量，也就是说我们函数括弧{}中定义的变量（但不包括static声明的变量，static意味着在数据段中存放变量）。除此以外，在函数被调用时，其参数也会被压入发起调用的进程栈中，并且待到调用结束后，函数的返回值也会被存放回栈中。由于栈的先进后出特点，所以栈特别方便用来保存/恢复调用现场

2.2 fork 、 exec 、 wait 等是进程操作的常用API，请调研了解这些API的使用方法。

(1) 请写一个C程序，该程序首先创建一个1到10的整数数组，然后创建一个子进程，并让子进程对前述数组所有元素求和，并打印求和结果。等子进程完成求和后，父进程打印“parent process finishes”，再退出。

(2) 在(1)所写的程序基础上，当子进程完成数组求和后，让其执行 `ls -l` 命令(注：该命令用于显示某个目录下文件和子目录的详细信息)，显示你运行程序所用操作系统的某个目录详情。例如，让子进程执行 `ls -l /usr/bin` 目录，显示 `/usr/bin` 目录下的详情。父进程仍然需要等待子进程执行完后打印“parent process finishes”，再退出。

(3) 请阅读XV6代码(<https://pdos.csail.mit.edu/6.828/2024/xv6.html>)，找出XV6代码中对进程控制块(PCB)的定义代码，说明其所在的文件，以及当 `fork` 执行时，对PCB做了哪些操作？

提交内容

1. 所写C程序，打印结果截图，说明等
2. 所写C程序，打印结果截图，说明等
3. 代码分析介绍

解:

(1)

```

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

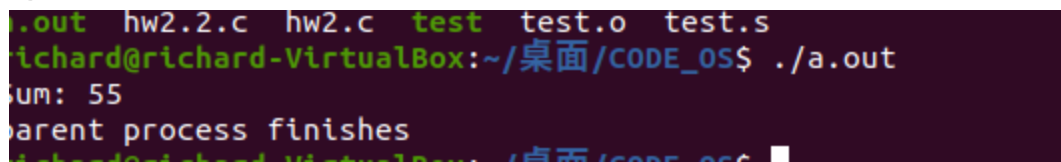
int main() {
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    } else if (pid == 0) {
        // Child process
        int sum = 0;
        for (int i = 0; i < 10; i++) {
            sum += arr[i];
        }
        printf("Sum: %d\n", sum);
        exit(0);
    } else {
        // Parent process
        wait(NULL);
        printf("parent process finishes\n");
    }

    return 0;
}

```

打印结果:



```

hw2.2.c hw2.c test test.o test.s
richard@richard-VirtualBox:~/桌面/CODE_OS$ ./a.out
Sum: 55
parent process finishes
richard@richard-VirtualBox:~/桌面/CODE_OS$

```

(2)

```

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

int main() {
    int arr[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        return 1;
    } else if (pid == 0) {

        int sum = 0;
        for (int i = 0; i < 10; i++) {
            sum += arr[i];
        }
        printf("Sum: %d\n", sum);

        execlp("ls", "ls", "-l", "/usr/bin", (char *)NULL);
        perror("execlp failed");
        exit(1);
    } else {
        // Parent process
        wait(NULL);
        printf("parent process finishes\n");
    }

    return 0;
}

```

打印结果:

```

Sum: 55
total 205156
-rwxr-xr-x 1 root root      59736 9月 5 2019 '['
-rwxr-xr-x 1 root root        96 2月 1 2024 2to3-2.7
-rwxr-xr-x 1 root root     31248 8月 28 05:51 aa-enabled
-rwxr-xr-x 1 root root     35344 8月 28 05:51 aa-exec
-rwxr-xr-x 1 root root     22912 4月 14 2021 aconnect
-rwxr-xr-x 1 root root     19016 11月 28 2019 acpi_listen
-rwxr-xr-x 1 root root      7415 3月 21 2023 add-apt-repository
-rwxr-xr-x 1 root root     30952 4月 9 23:34 addpart
lrwxrwxrwx 1 root root        26 1月 23 2024 addr2line -> x86_64-linux-gnu-addr2line
-rwxr-xr-x 1 root root     47552 4月 14 2021 alsabat
-rwxr-xr-x 1 root root     85296 4月 14 2021 alsaloop
-rwxr-xr-x 1 root root     72432 4月 14 2021 alsamixer
-rwxr-xr-x 1 root root     14720 4月 14 2021 alsatplg
-rwxr-xr-x 1 root root     31528 4月 14 2021 alsaucm
-rwxr-xr-x 1 root root     31112 4月 14 2021 amidi
-rwxr-xr-x 1 root root     63952 4月 14 2021 amixer
-rwxr-xr-x 1 root root      2668 3月 22 2020 amuFormat.sh
-rwxr-xr-x 1 root root       274 10月 2 2017 apg
-rwxr-xr-x 1 root root     26696 10月 2 2017 apgbfm
-rwxr-xr-x 1 root root     84408 4月 14 2021 aplay
-rwxr-xr-x 1 root root     93816 4月 22 2017 zipcloak
-rwxr-xr-x 1 root root     50718 11月 23 2023 zipdetails
-rwxr-xr-x 1 root root      2953 2月 1 2024 zipgrep
-rwxr-xr-x 2 root root    186664 2月 1 2024 zipinfo
-rwxr-xr-x 1 root root     89488 4月 22 2017 zipnote
-rwxr-xr-x 1 root root     93584 4月 22 2017 zipsplit
-rwxr-xr-x 1 root root     26952 3月 20 2023 zjsdecode
-rwxr-xr-x 1 root root      2206 4月 8 2022 zless
-rwxr-xr-x 1 root root      1842 4月 8 2022 zmore
-rwxr-xr-x 1 root root      4577 4月 8 2022 znew
parent process finishes
richard@richard-VirtualBox:~/桌面/CODE_OS$ S

```

(3)

对应代码截图：

```

// 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)
};

```

所在文件: kernel/proc.h

操作: 复制父进程的 PCB 以创建新的子进程 PCB。包括设置进程状态、复制地址空间以及将新进程添加到进程表中。

2.3 请阅读以下程序代码，回答下列问题

- (1) 该程序一共会生成几个子进程？请你画出生成的进程之间的关系(即谁是父进程谁是子进程)，并对进程关系进行适当说明。
- (2) 如果生成的子进程数量和宏定义 LOOP 不符，在不改变 for 循环的前提下，你能用少量代码修改，使该程序生成 LOOP 个子进程么？

提交内容

1. 问题解答，关系图和说明等
2. 修改后的代码，结果截图，对代码的说明等

```

#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");
        }
        else {
            sleep(5);
        }
    }
    return 0;
}

```

解:

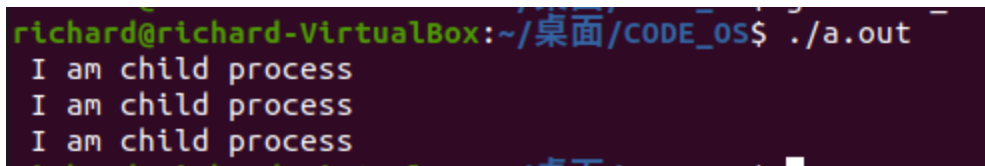
(1)包含两个子进程

```

Parent Process (P)
├─ Child Process 1 (C1)
└─ Child Process 2 (C2)

```

输出截图:



```

richard@richard-VirtualBox: ~/桌面/CODE_OS$ ./a.out
I am child process
I am child process
I am child process

```

P是父进程，c1是第一次循环的子进程，c2是第二次循环的子进程。每次调用fork都会创建一个子进程

(2)

为了保证子进程被创建，只需确保只有父进程持续调用fork，添加exit(0)即可实现，代码如下：

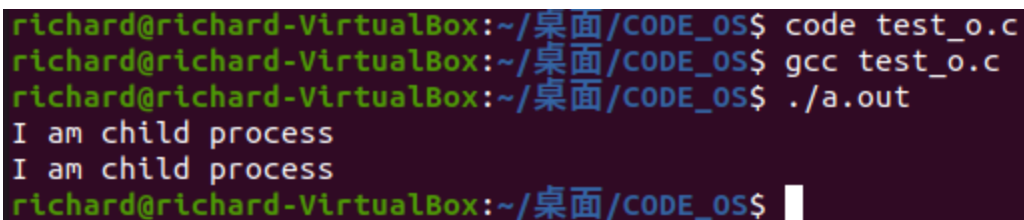

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#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) {
            // Child process
            printf("I am child process\n");
            exit(0); // Ensure child process exits
        } else {
            // Parent process
            sleep(5);
        }
    }
    return 0;
}
```

输出截图:



```
richard@richard-VirtualBox:~/桌面/CODE_OS$ code test_o.c
richard@richard-VirtualBox:~/桌面/CODE_OS$ gcc test_o.c
richard@richard-VirtualBox:~/桌面/CODE_OS$ ./a.out
I am child process
I am child process
richard@richard-VirtualBox:~/桌面/CODE_OS$
```