操作系统 第二次作业

姓名：朱首赫

学号：2023K8009906029

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。

解答：

(1) .data 段存放初始化过的全局变量和静态变量；.bss 段存放未初始化的全局变量和静态变量；堆 (heap) 为动态分配的内存。

下表中即为题目要求的 C 样例程序。其中 global_data 为初始化过的全局变量，在 data 段上；global_bss 为未初始化的全局变量，在 bss 段上；p 指向的内存是由 malloc 函数动态分配的，即 *p 在堆上。

Listing 1: 样例程序：包含 data 段、bss 段和堆的使用

```c
#include <stdlib.h>
int global_data = 100;
int global_bss;
int main(){
    int *p = (int *) malloc(sizeof(int));
    *p = 10;
}
```

(2) 图 1 使用 “objdump -t ./2.1 ” 命令输出了程序的符号表，列出程序中所有的函数名、全局变量名，其中标出的位置可以清楚看到 global_data 在 data 段上、global_bss 在 bss 段上。图 2 则是使用了 “objdump -h ./2.1 ” 命令输出了程序的段列表，显示文件中所有段的摘要信息，比如框选位置可以看到.data 段和.bss 段的大小、在文件中的位置等信息。

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw2$ objdump -t ./2.1

./2.1:     file format elf64-x86-64

SYMBOL TABLE:
0000000000000318 l    d  .interp        0000000000000000              .interp
0000000000000338 l    d  .note.gnu.property     0000000000000000              .note.gnu.property
0000000000000358 l    d  .note.gnu.build-id     0000000000000000              .note.gnu.build-id
000000000000037c l    d  .note.ABI-tag  0000000000000000              .note.ABI-tag
00000000000003a0 l    d  .gnu.hash      0000000000000000              .gnu.hash
00000000000003c8 l    d  .dynsym        0000000000000000              .dynsym
0000000000000470 l    d  .dynstr        0000000000000000              .dynstr
00000000000004f4 l    d  .gnu.version   0000000000000000              .gnu.version
0000000000000508 l    d  .gnu.version_r 0000000000000000              .gnu.version_r
0000000000000528 l    d  .rela.dyn      0000000000000000              .rela.dyn
00000000000005e8 l    d  .rela.plt      0000000000000000              .rela.plt
0000000000001000 l    d  .init  0000000000000000              .init
0000000000001020 l    d  .plt   0000000000000000              .plt
0000000000001040 l    d  .plt.got       0000000000000000              .plt.got
0000000000001050 l    d  .plt.sec       0000000000000000              .plt.sec
0000000000001060 l    d  .text  0000000000000000              .text
00000000000011f8 l    d  .fini  0000000000000000              .fini
0000000000002000 l    d  .rodata        0000000000000000              .rodata
0000000000002004 l    d  .eh_frame_hdr  0000000000000000              .eh_frame_hdr
0000000000002048 l    d  .eh_frame      0000000000000000              .eh_frame
0000000000003db8 l    d  .init_array    0000000000000000              .init_array
0000000000003dc0 l    d  .fini_array    0000000000000000              .fini_array
0000000000003dc8 l    d  .dynamic       0000000000000000              .dynamic
0000000000003fb8 l    d  .got   0000000000000000              .got
0000000000004000 l    d  .data  0000000000000000              .data
0000000000004014 l    d  .bss   0000000000000000              .bss
0000000000000000 l    d  .comment       0000000000000000              .comment
0000000000000000 l    df *ABS*  0000000000000000              crtstuff.c
0000000000001090 l     F .text  0000000000000000              deregister_tm_clones
00000000000010c0 l     F .text  0000000000000000              register_tm_clones
0000000000001100 l     F .text  0000000000000000              __do_global_dtors_aux
0000000000004014 l     O .bss   0000000000000001              completed.8061
0000000000003dc0 l     O .fini_array    0000000000000000              __do_global_dtors_aux_fini_array_entry
0000000000001140 l     F .text  0000000000000000              frame_dummy
0000000000003db8 l     O .init_array    0000000000000000              __frame_dummy_init_array_entry
0000000000000000 l    df *ABS*  0000000000000000              2.1.c
0000000000000000 l    df *ABS*  0000000000000000              crtstuff.c
000000000000214c l     O .eh_frame      0000000000000000              __FRAME_END__
0000000000000000 l    df *ABS*  0000000000000000
0000000000003dc0 l       .init_array    0000000000000000              __init_array_end
0000000000003dc8 l     O .dynamic       0000000000000000              _DYNAMIC
0000000000003db8 l       .init_array    0000000000000000              __init_array_start
0000000000002004 l       .eh_frame_hdr  0000000000000000              __GNU_EH_FRAME_HDR
0000000000003fb8 l     O .got   0000000000000000              _GLOBAL_OFFSET_TABLE_
0000000000001000 l     F .init  0000000000000000              _init
00000000000011f0 g     F .text  0000000000000005              __libc_csu_fini
0000000000000000  w      *UND*  0000000000000000              _ITM_deregisterTMCloneTable
0000000000004000  w       .data  0000000000000000              data_start
0000000000004014 g       .data  0000000000000000              _edata
0000000000004010 g     O .data  0000000000000004              global_data
00000000000011f8 g     F .fini  0000000000000000              .hidden _fini
0000000000004018 g     O .bss   0000000000000004              global_bss
0000000000000000       F *UND*  0000000000000000              __libc_start_main@@GLIBC_2.2.5
0000000000004000 g       .data  0000000000000000              __data_start
0000000000000000  w      *UND*  0000000000000000              __gmon_start__
0000000000004008 g     O .data  0000000000000000              .hidden __dso_handle
0000000000002000 g     O .rodata        0000000000000004              _IO_stdin_used
0000000000001180 g     F .text  0000000000000065              __libc_csu_init
0000000000000000       F *UND*  0000000000000000              malloc@@GLIBC_2.2.5
0000000000004020 g       .bss   0000000000000000              _end
0000000000001060 g     F .text  000000000000002f              _start
0000000000004014 g       .bss   0000000000000000              __bss_start
0000000000001149 g     F .text  000000000000002b              main
0000000000004018 g     O .data  0000000000000000              .hidden __TMC_END__
0000000000000000  w      *UND*  0000000000000000              _ITM_registerTMCloneTable
0000000000000000  w    F *UND*  0000000000000000              __cxa_finalize@@GLIBC_2.2.5
```

图 1: 样例程序符号表

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw2$ objdump -h ./2.1

./2.1:     file format elf64-x86-64

Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000000318  0000000000000318  00000318  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.gnu.property 00000020  0000000000000338  0000000000000338  00000338  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .note.gnu.build-id 00000024  0000000000000358  0000000000000358  00000358  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  3 .note.ABI-tag 00000020  000000000000037c  000000000000037c  0000037c  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  4 .gnu.hash     00000024  00000000000003a0  00000000000003a0  000003a0  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  5 .dynsym       000000a8  00000000000003c8  00000000000003c8  000003c8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  6 .dynstr       00000084  0000000000000470  0000000000000470  00000470  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  7 .gnu.version  0000000e  00000000000004f4  00000000000004f4  000004f4  2**1
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  8 .gnu.version_r 00000020  0000000000000508  0000000000000508  00000508  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  9 .rela.dyn     000000c0  0000000000000528  0000000000000528  00000528  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 10 .rela.plt     00000018  00000000000005e8  00000000000005e8  000005e8  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 11 .init         0000001b  0000000000001000  0000000000001000  00001000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 12 .plt          00000020  0000000000001020  0000000000001020  00001020  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 13 .plt.got      00000010  0000000000001040  0000000000001040  00001040  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 14 .plt.sec      00000010  0000000000001050  0000000000001050  00001050  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 15 .text         00000195  0000000000001060  0000000000001060  00001060  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 16 .fini         0000000d  00000000000011f8  00000000000011f8  000011f8  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
 17 .rodata       00000004  0000000000002000  0000000000002000  00002000  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 18 .eh_frame_hdr 00000044  0000000000002004  0000000000002004  00002004  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 19 .eh_frame     00000108  0000000000002048  0000000000002048  00002048  2**3
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
 20 .init_array   00000008  0000000000003db8  0000000000003db8  00002db8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 21 .fini_array   00000008  0000000000003dc0  0000000000003dc0  00002dc0  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 22 .dynamic      000001f0  0000000000003dc8  0000000000003dc8  00002dc8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 23 .got          00000048  0000000000003fb8  0000000000003fb8  00002fb8  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 24 .data         00000014  0000000000004000  0000000000004000  00003000  2**3
                  CONTENTS, ALLOC, LOAD, DATA
 25 .bss          0000000c  0000000000004014  0000000000004014  00003014  2**2
                  ALLOC
 26 .comment      0000002b  0000000000000000  0000000000000000  00003014  2**0
                  CONTENTS, READONLY
```

图 2: 样例程序段列表

(3) （1）中所写程序用到了栈。栈主要存放局部变量和函数调用的栈帧（包括函数参数、返回地址、旧的帧指针、保存的寄存器、局部变量）。在表 1 的程序中，main 函数中的指针变量 p 是一个局部变量，它被分配在栈内存上；且程序运行时先调用 main 函数，又在 main 函数里调用了 malloc 函数，每次调用时系统都会使用栈来保存返回地址等信息。

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/6.828/2025/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

解答:

(1) 表 2 即为题目要求的 C 程序代码。为了让父进程等待子进程完成后再执行打印语句，需要调用 wait() 函数暂停父进程的执行，直到它的一个子进程结束为止。

运行结果如下图 3 所示，可以看到父进程的输出在子进程完成之后。在此之后尝试去掉 wait() 语句，发现父进程打印语句总是在子进程之前。经查阅资料后发现：在单核系统中，父子进程执行的先后顺序时不确定的，这基于操作系统的调度算法；在多核系统中，两个进程是可以同时执行的。无论是上面哪种情况，由于父进程任务非常简单（只需要打印一条语句），所以它大概率会先执行完成。

Listing 2: 符合 2.2(1) 要求的 C 程序

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

int main(int argc, char** argv) {
    int a[10];
    for (int i = 0; i < 10; i++)
```

```
    {
        a[i] = i + 1;
    }

    int rc = fork();

    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
    }
    else if (rc == 0) {
        printf("Child process:\n");
        int sum = 0;
        for (int j = 0; j < 10; j++)
        {
            sum += a[j];
        }
        printf("sum: %d\n", sum);
    }
    else {
        wait(NULL); // 这里的 NULL 表示我们不关心子进程的退出状态
        printf("parent process finishes\n");
    }
    return 0;
}
```

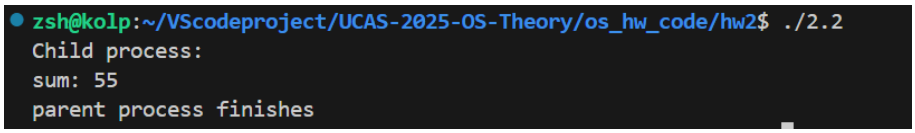

图 3: 2.2(1) 运行结果

(2) exec 函数的作用：加载一个新的程序到当前进程的内存空间，并开始执行它。一旦 exec 调用成功，原来进程的代码、数据、BSS、堆栈等所有内容都会被完全替换掉，从而导致 exec 后面的代码永远不会被执行。exec 函数的种类有很多，在表 3 的程序中，选择调用了 execpl() 函数执行 ls -l 命令，函数名中的 “l” 代表可以直接把命令参数作为字符串一个一个写在代码里，可读性很高；“p” 则表示该函数会自动在 PATH 环境变量设定的所有目录里帮我们找到 ls 程序，无需我们手动指定确切位置。

运行结果如下图 4 所示，由于我的环境中 usr/bin 目录下的文件太多不便于展示，所以改为显示程序所在目录的信息。

Listing 3: 符合 2.2(2) 要求的 C 程序

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

```

```c
int main(int argc, char** argv) {
    int a[10];
    for (int i = 0; i < 10; i++)
    {
        a[i] = i + 1;
    }

    int rc = fork();

    if (rc < 0) {
        fprintf(stderr, "fork failed\n");
    }
    else if (rc == 0) {
        printf("Child process:\n");
        int sum = 0;
        for (int j = 0; j < 10; j++)
        {
            sum += a[j];
        }
        printf("sum: %d\n", sum);
        printf("list details about the current dir:\n");
        execlp("ls", "ls", "-l", "/home/zsh/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw2", NULL);
    }
    else {
        wait(NULL); // 这里的 NULL 表示我们不关心子进程的退出状态
        printf("parent process finishes\n");
    }
    return 0;
}
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw2$ ./2.2_2
Child process:
sum: 55
list details about the current dir:
total 100
-rwxr-xr-x 1 zsh zsh 16768 Sep 30 23:01 2.1
-rw-r--r-- 1 zsh zsh   127 Sep 30 22:57 2.1.c
-rwxr-xr-x 1 zsh zsh 16960 Oct  1 10:03 2.2
-rw-r--r-- 1 zsh zsh   652 Oct  1 09:54 2.2_1.c
-rwxr-xr-x 1 zsh zsh 17008 Oct  1 10:08 2.2_2
-rw-r--r-- 1 zsh zsh   811 Oct  1 10:08 2.2_2.c
-rwxr-xr-x 1 zsh zsh 16768 Sep 30 22:57 a.out
-rw-r--r-- 1 zsh zsh   456 Oct  1 09:45 fork1.c
-rw-r--r-- 1 zsh zsh   760 Oct  1 09:45 fork2.c
parent process finishes
```

图 4: 2.2(2) 运行结果

(3) 对 PCB 定义的代码位于头文件 xv6-riscv/kernel/proc.h 中，内容如表 4 。kfork() 函数的定义则位于 xv6-riscv/kernel/proc.c 中，内容如表 5 。当 kfork() 执行时，对 PCB 的操作如下：

1. 先分配一个新的 PCB
2. 复制父进程的整个用户内存空间，包括代码、全局变量、堆和栈
3. 复制父进程的所有保存寄存器（trapframe）
4. 将子进程 a0 寄存器的值改为 0（即 fork 的返回值），表示自己是子进程
5. 复制打开的文件描述符、当前工作目录、进程名
6. 将子进程的 parent 字段指向父进程的 proc 结构体
7. 将新进程的 state 字段设置为 RUNNABLE，表示该进程可运行

Listing 4: PCB 定义

```
// 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)
};
```

Listing 5: kfork() 定义

```
// Create a new process, copying the parent.
// Sets up child kernel stack to return as if from fork() system call.
int
kfork(void)
{
  int i, pid;
  struct proc *np;
  struct proc *p = myproc();

  // Allocate process.
  if((np = allocproc()) == 0){
    return -1;
```

```
  }

  // Copy user memory from parent to child.
  if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0){
    freeproc(np);
    release(&np->lock);
    return -1;
  }
  np->sz = p->sz;

  // copy saved user registers.
  *(np->trapframe) = *(p->trapframe);

  // Cause fork to return 0 in the child.
  np->trapframe->a0 = 0;

  // increment reference counts on open file descriptors.
  for(i = 0; i < NOFILE; i++)
    if(p->ofile[i])
      np->ofile[i] = filedup(p->ofile[i]);
  np->cwd = idup(p->cwd);

  safestrcpy(np->name, p->name, sizeof(p->name));

  pid = np->pid;

  release(&np->lock);

  acquire(&wait_lock);
  np->parent = p;
  release(&wait_lock);

  acquire(&np->lock);
  np->state = RUNNABLE;
  release(&np->lock);

  return pid;
}
```

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

Listing 6: 2.3 题目示例代码

```c
#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;
}
```

解答：

(1) 运行结果如图 5 ，该程序一共会生成 3 个子进程。

进程之间的关系如图 6 所示，P0 为原始的父进程，C1, C2, C3 均为 fork 生成的子进程，其中 C3 的父进程是 C1。

整个程序运行的过程如图 7 所示，原始进程 P0 在 loop=0 循环中生成了子进程 C1，然后 P0 和 C1 根据不同的分支条件分别执行 sleep(5) 和 print 语句；接着进入 loop=1 循环，P0 和 C1 各自生成了一个子进程 C2、C3，此时 C1 变成了父进程 (fork() 返回 C1 的 pid 变为正) 根据分支条件，P0、C1 执行 sleep(5)，C2、C3 执行 print 语句。

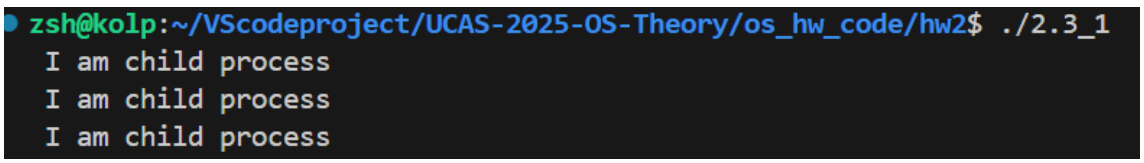

图 5: 2.3(1) 题示代码运行结果

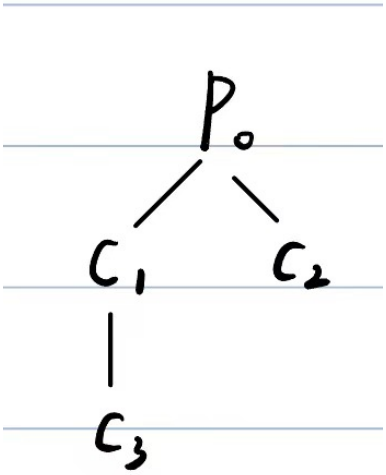

图 6: 进程间关系

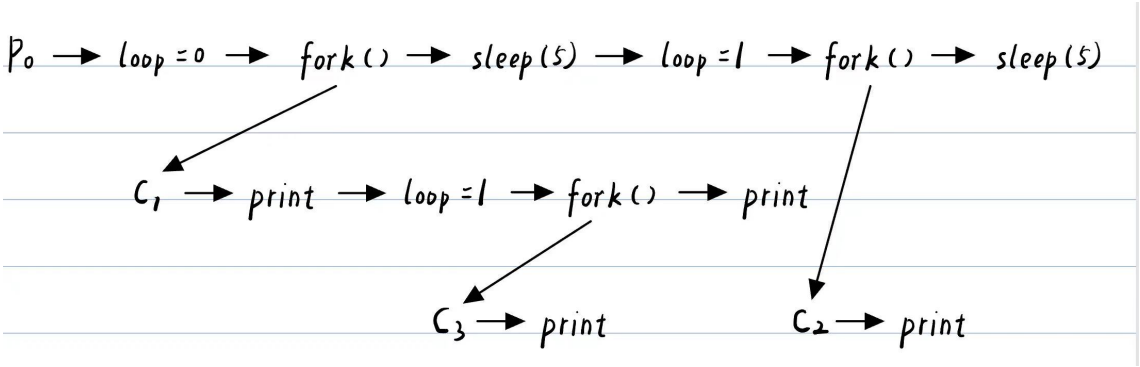

图 7: 程序运行过程示意图

(2) 只需在子进程分支中添加 loop=LOOP; 语句，即可让子进程直接结束循环，这样只有原始的父进程才能创建子进程，由此只会创建 LOOP 个子进程。代码如表 7 所示，运行结果如图 8 。

Listing 7: 创建 LOOP 个子进程

```
#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)
```

```
14          fprintf(stderr, "fork failed\n");
15      else if(pid == 0) {
16          printf(" I am child process\n");
17          loop = LOOP;
18      }
19      else {
20          sleep(5);
21      }
22    }
23    return 0;
24 }
```

```
zsh@kolp:~/VScodeproject/UCAS-2025-OS-Theory/os_hw_code/hw2$ ./2.3_1
I am child process
I am child process
```

图 8: 2.3(2) 修改后代码运行结果

注：本作业使用了 Gemini2.5 Pro 模型，用于查阅并分析 API 用法。