

目录

1	实验概述	2
2	实验任务一	2
2.1	系统调用实现过程	2
2.2	实验结果展示	4
2.3	gdb 调试分析	5
3	实验任务二	6
3.1	fork 实现的基本思路	6
3.2	执行结果分析	7
3.3	gdb 调试分析	8
4	实验任务三	9
4.1	exit 的执行过程	9
4.2	进程退出后能够隐式地调用 exit 和此时的 exit 返回值是 0 的原因	9
4.3	wait 的执行过程	10
4.4	回收僵尸进程	11
5	实验总结	13

1 实验概述

- 实验任务一：编写一个系统调用，然后调用它，并用 gdb 分析系统调用后栈的变化情况以及说明 TSS 在系统调用执行过程中的作用
- 实验任务二：根据代码逻辑和执行结果来分析 fork 实现的基本思路，并用 gdb 跟踪并分析子进程的执行流程，并解释 fork 是如何保证子进程的返回值是 0，父进程的 fork 返回值是子进程的 pid
- 实验任务三：结合代码分析 exit 和 wait 的执行过程，并实现回收僵尸进程的有效方法

2 实验任务一

本任务中，实现了用户进程中输出函数的系统调用:user_printf，以在用户进程中实现输出功能。

2.1 系统调用实现过程

原先不能在用户进程中 printf 的原因是，printf 函数的调用中，会调用 print_add_to_buffer 函数，而在此函数中

```
1 counter = stdio.print(buffer);
```

这一句访问了显存，而用户进程特权级 3 的情况下是无法访问显存的，导致 qemu 闪退。

因此用户进程需要输出时，需要先通过系统调用进入内核态，再在内核态中将要输出的内容写入显存。

然而系统调用最多只支持五个参数，向 printf 这样的可变参数是不能直接传入系统调用函数的，是实现用户级的 printf 的一大难题。

我的实现则是，实现一个 vsprintf 函数，支持传入缓冲区，格式化字符串和参数列表，将要输出的内容写入缓冲区，从而避免直接访问显存：

```
1 int vsprintf(char *buffer, const char *fmt, va_list ap)
2 {
3     char number[33];
4     int idx = 0, counter = 0;
5
6     for (int i = 0; fmt[i]; ++i)
7     {
8         if (fmt[i] != '%')
9         {
10             buffer[idx++] = fmt[i];
11             counter++;
12         }
13         else
14         {
15             i++;
16             if (fmt[i] == '\\0') break;
17
18             switch (fmt[i])
19             {
20                 case '%':
21                     buffer[idx++] = fmt[i];
22                     counter++;
23                     break;
24
25                 case 'c':
```

```

26         buffer[idx++] = va_arg(ap, int); // char会被提升为int
27         counter++;
28         break;
29
30     case 's':
31     {
32         const char* str = va_arg(ap, const char*);
33         while (*str && idx < 1023) { // 防止缓冲区溢出
34             buffer[idx++] = *str++;
35             counter++;
36         }
37     }
38     break;
39
40     case 'd':
41     case 'x':
42     {
43         int temp = va_arg(ap, int);
44         if (temp < 0 && fmt[i] == 'd')
45         {
46             buffer[idx++] = '-';
47             counter++;
48             temp = -temp;
49         }
50
51         itos(number, temp, (fmt[i] == 'd' ? 10 : 16));
52         for (int j = 0; number[j] && idx < 1023; ++j)
53         {
54             buffer[idx++] = number[j];
55             counter++;
56         }
57     }
58     break;
59 }
60 }
61 }
62
63 buffer[idx] = '\0';
64 return counter;
65 }

```

接着在 user_printf 函数中调用 vsprintf 函数，得到要输出的字符串，再通过系统调用将字符串的内容写入显存：

```

1 int syscall_1(int first, int second, int third, int forth, int fifth)
2 {
3     return printf("%s", (char *)first);
4 }
5 int user_printf(const char *fmt, ...)
6 {
7     char buffer[1024];
8     va_list ap;
9     va_start(ap, fmt);
10
11     int len = vsprintf(buffer, fmt, ap);
12     va_end(ap);

```

```

13
14 // 通过系统调用传递到内核
15 return asm_system_call(1, (int)buffer, 0, 0, 0);
16 }

```

最后再注册系统调用

```

1 systemService.setSystemCall(1, (int)syscall_1);

```

至此，用户进程的 user_printf 函数实现完成。

2.2 实验结果展示

```

1 void first_process()
2 {
3     //asm_system_call(0, 132, 324, 12, 124);
4     user_printf("hello %d %c %s %d %d %d\n",1,'a',"sysu",4,5,6);
5     asm_halt();
6 }

```

输出结果如下：

```

SeaBIOS (version 1.16.3-debian-1.16.3-2)

iPXE (https://ipxe.org) 00:03.0 CA00 PCI2.10 PnP PMM+06FCB050+06F0B050 CA00

Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
hello 1 a sysu 4 5 6

```

成功将内容输出到显存中

2.3 gdb 调试分析

用户进程刚开始执行时，查看栈的情况：



```
Terminal
File Edit View Search Terminal Help
../src/kernel/setup.cpp
43
44 // 通过系统调用传递到内核
45 return asm_system_call(1, (int)buffer, 0, 0, 0);
46 }
47 void first_process()
48 {
49     asm_system_call(0, 132, 324, 12, 124);
50     //user_printf("hello %d %c %s %d %d\n",1,'a',"sysu",4,5,6);
51     asm_halt();
52 }
53
54 void first_thread(void *arg)
55 {
remote Thread 1.1 (src) In: first_process L48 PC: 0xc0021367
(gdb) c
Continuing.

Breakpoint 1, first_process () at ../src/kernel/setup.cpp:48
warning: Source file is more recent than executable.
(gdb) info register esp
esp      0x8049000      0x8049000
(gdb)
```

此时栈指针的位置是处于用户进程的地址空间中，说明此时的栈为特权级 3 的栈。

进入系统调用处理函数后，查看当前栈的情况：



```
Terminal
File Edit View Search Terminal Help
../src/utls/asm_utils.asm
128 iret
129 asm_system_call:
130     push ebp
131     mov ebp, esp
132
133     push ebx
134     push ecx
135     push edx
136     push esi
137     push edi
138
139     mov eax, [ebp + 2 * 4]
140     mov ebx, [ebp + 3 * 4]
141     mov ecx, [ebp + 4 * 4]
142     mov edx, [ebp + 5 * 4]
143     mov esi, [ebp + 6 * 4]
144     mov edi, [ebp + 7 * 4]
145
146     int 0x80
147
148     pop edi
149     pop esi
150     pop edx
151     pop ecx
remote Thread 1.1 (src) In: asm_system_call L144 PC: 0xc002297a
(gdb) print tss
$2 = {backlink = 0, esp0 = -1073587712, ss0 = 16, esp1 = 0, ss1 = 0, esp2 = 0,
ss2 = 0, cr3 = 0, eip = 0, eflags = 0, eax = 0, ecx = 0, edx = 0, ebx = 0,
esp = 0, ebp = 0, esi = 0, edi = 0, es = 0, cs = 0, ss = 0, ds = 0, fs = 0,
gs = 0, ldt = 0, trace = 0, ioMap = -1073530100}
(gdb) print /x tss.esp0
$3 = 0xc0025a00
(gdb)
```

栈指针位于内核的地址空间中，说明此时的栈为特权级 0 的栈。

在系统调用处理函数中，查看 TSS 中 esp0 的内容：

```

Terminal
File Edit View Search Terminal Help
~/src/utls/asm_utls.asm
77
78 add word[ASM_GDTR], 8
79 lgdt [ASM_GDTR]
80
81 pop esi
82 pop ebx
83 pop ebp
84
85 ret
86 ; int asm_system_call_handler();
87 asm_system_call_handler:
88 push cs
89 push es
90 push fs
91 push gs
92 pushad
93
94 push eax
95
96 ; 栈段会从tss中自动加载
97
98 mov eax, DATA_SELECTOR
99 mov ds, eax
100 mov es, eax

remote Thread 1.1 (src) In: asm_system_call_handler L88 PC: 0xc0022927
esp = 0, ebp = 0, esi = 0, edi = 0, es = 0, cs = 0, ss = 0, ds = 0, fs = 0,
gs = 0, ldt = 0, trace = 0, ioMap = -1073530100
(gdb) print /x tss.esp0
$3 = 0xc0025a00
(gdb) b asm_system_call_handler
Breakpoint 2 at 0xc0022927: file ../src/utls/asm_utls.asm, line 88.
(gdb) c
Continuing.

Breakpoint 2, asm_system_call_handler () at ../src/utls/asm_utls.asm:88
(gdb) info register esp
esp      0xc00259ec      0xc00259ec <PCB_SET+8172>
(gdb)

```

对比后可发现，TSS 的作用是为 CPU 提供 0 特权级栈所在的地址和段选择子。

在系统调用的 cpu 变态中，TSS 的 esp0 的内容将会被加载到 esp 寄存器中，完成栈的切换。

3 实验任务二

3.1 fork 实现的基本思路

fork 实现的基本思路如下：

- 父进程调用 fork 函数，执行系统调用，进入内核态
- 内核态中，创建子进程
- 复制父进程的 0 特权级栈到子进程的 0 特权级栈，0 特权级栈里面保存着父进程调用前的现场信息，以此来给子进程“伪造”一个中断现场
- 设置子进程的 0 特权级栈中的返回值 (eax 寄存器) 为 0，从而与父进程区分开
- 设置好子进程的栈，使得 asm_switch_thread 的返回地址正好是 asm_start_process 函数，并给此函数传入参数 (子进程的 0 特权级栈地址)，从而最后可以在 asm_start_process 中通过 iret 从“伪造的中断现场”返回到用户态，即父进程 fork 调用的“断点”
- 复制父进程的信息、页表、内存等情况到子进程，但子进程的页目录表与父进程区分开

代码中最关键的部分首先是子进程的栈设置，需要设置 asm_switch_thread 的返回地址、asm_start_process 的返回地址以及参数：

```

1 child->stack = (int *)childpss - 7;
2 child->stack[0] = 0; //栈顶
3 child->stack[1] = 0;
4 child->stack[2] = 0;
5 child->stack[3] = 0;
6 child->stack[4] = (int)asm_start_process; //asm_switch_thread的返回地址
7 child->stack[5] = 0; // asm_start_process 返回地址
8 child->stack[6] = (int)childpss; // asm_start_process 参数，就是写入栈指针的值

```

结合以下 asm_switch_thread 和 asm_start_process 的实现：

```

1  asm_switch_thread:
2  push ebp
3  push ebx
4  push edi
5  push esi
6  mov eax, [esp + 5 * 4]
7  mov [eax], esp ; 保存当前栈指针到PCB中, 以便日后恢复
8  mov eax, [esp + 6 * 4]
9  mov esp, [eax] ; 此时栈已经从cur栈切换到next栈
10 pop esi
11 pop edi
12 pop ebx
13 pop ebp
14 sti
15 ret
16 asm_start_process:
17     ; jmp $
18     mov eax, dword[esp+4]
19     mov esp, eax
20     popad
21     pop gs;
22     pop fs;
23     pop es;
24     pop ds;
25     iret

```

asm_switch_thread 弹出四个参数后, 栈顶此时就是即将返回的地址, 即 asm_start_process, 而在 asm_start_process 中, 栈顶的参数就是即将返回的地址, 而栈顶的第二个元素则是 asm_start_process 的参数, 即子进程的 0 特权级栈地址。因此这部分需要在 fork 中提前设置好

接着在复制父进程的页目录表和页表到子进程时, 需要注意区分子进程和父进程的地址空间, 适时更新 CR3 寄存器。

3.2 执行结果分析

```

1  void first_process()
2  {
3      int pid = fork();
4
5      if (pid == -1)
6      {
7          printf("can not fork\n");
8      }
9      else
10     {
11         if (pid)
12         {
13             printf("I am father, fork reutrn: %d\n", pid);
14         }
15         else

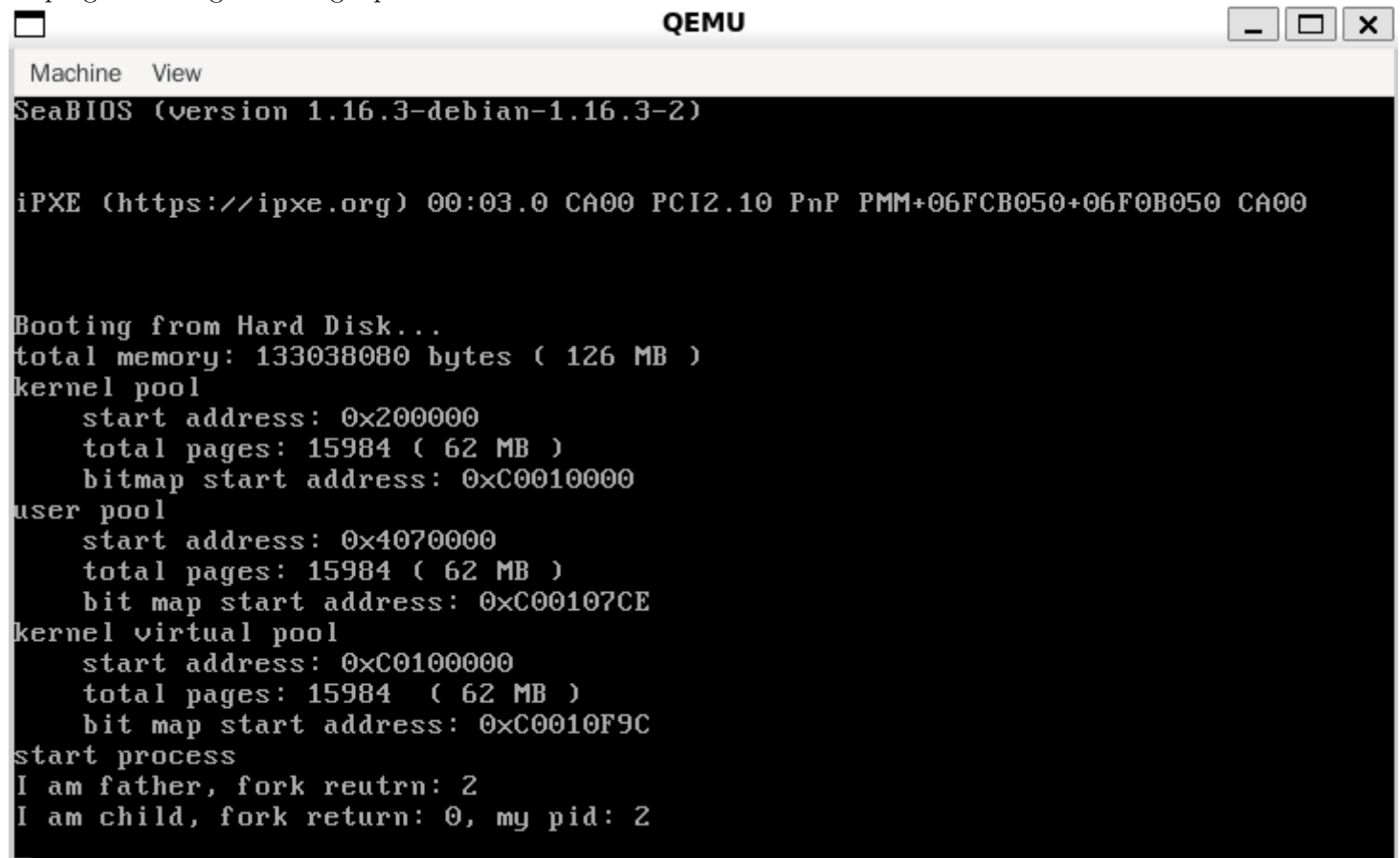
```

```

16     {
17         printf("I am child, fork return: %d, my pid: %d\n", pid, programManager.running->pid);
18     }
19 }
20 asm_halt();
21 }

```

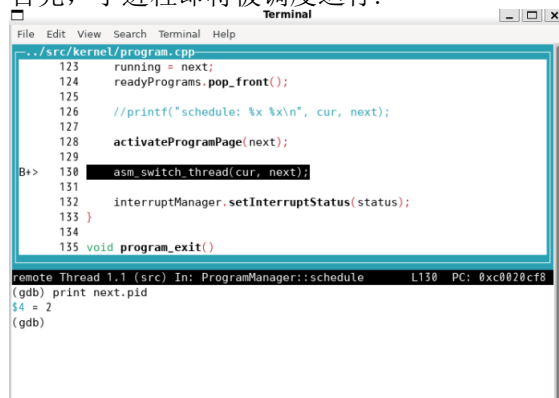
父进程中，fork 的返回值 pid 是子进程的 pid，子进程的返回值是 pid=0，因此输出可以看到父进程 pid 的值等于子进程 programManager.running->pid 的值。



3.3 gdb 调试分析

在本示例中，父进程的 pid 是 1，子进程的 pid 是 2

首先，子进程即将被调度运行:



asm_switch_thread 的返回地址是 asm_start_process，进入 asm_start_process 函数执行，接着通过 popad 弹出 0 特权级栈中存的通用寄存器的内容，其中 fork 的返回值在 eax 寄存器中，此时查看 eax 寄存器:


```

../src/utlis/asm_utils.asm
37  pop eax
38  ret
39  asm_start_process:
40  ; jmp $
41  mov eax, dword[esp+4]
42  mov esp, eax
43  popad
44  pop ebx
45  pop fs;
46  pop es;
47  pop ds;
48
49  iret

remote Thread 1.1 (src) In: asm_start_process L44 PC: 0xc0022c07
(gdb) info register eax
eax          0x0          0
(gdb)

```

eax 寄存器的值为 0，说明子进程的返回值是 0。这正是我们在 fork 函数中为子进程特意设置的随后“中断返回”，进入用户态进程，查看当前正在运行的进程 pid 和 fork 的返回值

```

../src/kernel/setup.cpp
38
39 void first_process()
40 {
41     int pid = fork();
42
43     if (pid == -1)
44     {
45         printf("can not fork\n");
46     }
47     else
48     {
49         if (pid)
50         {
51             printf("I am father, fork return: %d\n", pid);
52         }
53         else
54         {
55             printf("I am child, fork return: %d, my pid: %d\n", pid, programManager.running->pid);
56         }
57     }
58     asm_halt();
59 }
60
remote Thread 1.1 (src) In: first_process L55 PC: 0xc0021000
(gdb) s
(gdb) s
fork () at ../src/kernel/syscall.cpp:37
fork () at ../src/kernel/syscall.cpp:37
first_process () at ../src/kernel/setup.cpp:39
(gdb) print pid
pid = 2
(gdb) print programManager.running->pid
pid = 2
(gdb)

```

此时正在运行的进程 pid 是 2，fork 的返回值是 0，说明子进程的返回值是 0。

4 实验任务三

4.1 exit 的执行过程

exit 的执行过程如下：

- 用户进程调用 exit 函数，通过系统调用进入内核态
- 设置进程状态为 DEAD，并将返回值放入 PCB 中
- 释放进程的物理页、页表、页目录表和虚拟地址池 bitmap 的空间
- 立即调度

4.2 进程退出后能够隐式地调用 exit 和此时的 exit 返回值是 0 的原因

在 load_process 函数中，提前为进程的栈规划好，即进程这个“函数”的返回地址设置为 exit 函数的地址，并将 exit 的参数 (即返回值) 设置为 0，通过设置栈来完成。

```

1 int *userStack = (int *)interruptStack->esp;
2 userStack -= 3;
3 userStack[0] = (int)exit;
4 userStack[1] = 0;
5 userStack[2] = 0;

```

在这里 userStack[0] 是进程这个“函数”的返回地址，userStack[1] 是 exit 这个“函数”的返回地址，userStack[2] 是 exit 的参数 (即返回值)。

通过这样的设置，进程在执行完后就能隐式地调用 exit，且此时的 exit 返回值是 0。

4.3 wait 的执行过程

wait 的实现逻辑是，子进程在 exit 时不释放 PCB，而是等待父进程释放子进程的 PCB。父进程在调用 wait 时，找到第一个已经 exit 了但是还没释放 PCB 的子进程，将其 PCB 释放掉从而 wait 返回，并将子进程的返回值返回给父进程。

若子进程未结束，则父进程会一直等待，不会继续运行。

关键代码如下

```

1 while (true)
2 {
3     interrupt = interruptManager.getInterruptStatus();
4     interruptManager.disableInterrupt();
5
6     item = this->allPrograms.head.next;
7
8     // 查找子进程
9     flag = true;
10    while (item)
11    {
12        child = ListItem2PCB(item, tagInAllList);
13        if (child->parentPid == this->running->pid)
14        {
15            flag = false;
16            if (child->status == ProgramStatus::DEAD)
17            {
18                break;
19            }
20        }
21        item = item->next;
22    }
23
24    if (item) // 找到一个可返回的子进程
25    {
26        if (retval)
27        {
28            *retval = child->retValue;
29        }
30
31        int pid = child->pid;
32        printf("parent: %d ,wait for child: %d\n", child->parentPid, pid);
33        releasePCB(child);
34        interruptManager.setInterruptStatus(interrupt);
35        return pid;
36    }
37    else
38    {
39        if (flag) // 子进程已经返回
40        {
41
42            interruptManager.setInterruptStatus(interrupt);
43            return -1;
44        }
45        else // 存在子进程，但子进程的状态不是 DEAD
46        {

```

```

47         interruptManager.setInterruptStatus(interrupt);
48         schedule();
49     }
50 }
51 }

```

由于 while 循环的存在，若子进程未结束，则父进程会一直等待，不会继续运行。

4.4 回收僵尸进程

wait 进程的实现中存在以下问题:

```

1 else if (running->status == ProgramStatus::DEAD)
2 {
3     // 回收线程，子进程留到父进程回收
4     if(!running->pageDirectoryAddress) {
5         printf("release PCB, pid: %d\n", running->pid);
6         releasePCB(running);
7     }
8 }

```

在 schedule 函数中，没有回收任何进程，而父进程的 wait 只会回收第一个子进程，因此会造成:

1. 父进程的其余子进程无法被回收，造成僵尸进程。
2. 父进程的 PCB 无法被回收 (由于此父进程的父进程为 init 线程)，此父进程也成了僵尸进程。

因此实现僵尸进程回收的主要思路如下:

- 每隔一段时间 (时钟中断时)，自动调用回收僵尸进程的函数
- 在回收僵尸进程的函数中，遍历所有进程
- 若此进程是 DEAD 状态且它的父进程是 DEAD 状态或者已经被释放掉，则释放掉此进程
- 若此进程是 DEAD 状态且它的父进程是 init 线程，则释放掉此进程

函数实现如下

```

1 void ProgramManager::releaseZombie()
2 {
3     PCB *child;
4     ListItem *item;
5     bool interrupt;
6     item=this->allPrograms.head.next;
7     interrupt = interruptManager.getInterruptStatus();
8     interruptManager.disableInterrupt();
9     while (item)
10    {
11        child = ListItem2PCB(item, tagInAllList);
12        item = item->next;
13        if(child->status!=ProgramStatus::DEAD)
14            continue;
15        int parentpid=child->parentPid;
16        PCB *parent=get_pcb_from_pid(parentpid);
17        if(parent==nullptr||parent->pid==0||parent->status==ProgramStatus::DEAD)
18        {
19            printf("release zombie process, pid: %d\n", child->pid);
20            releasePCB(child);

```

```

21     }
22 }
23 }
24 interruptManager.setInterruptStatus(interrupt);
25 }

```

在时钟中断处理函数中调用此函数

```

1 extern "C" void c_time_interrupt_handler()
2 {
3     PCB *cur = programManager.running;
4     programManager.releaseZombie();
5     if (cur->ticks)
6     {
7         --cur->ticks;
8         ++cur->ticksPassedBy;
9     }
10    else
11    {
12        programManager.schedule();
13    }
14 }

```

在 setup.cpp 中设置的测试样例如下:

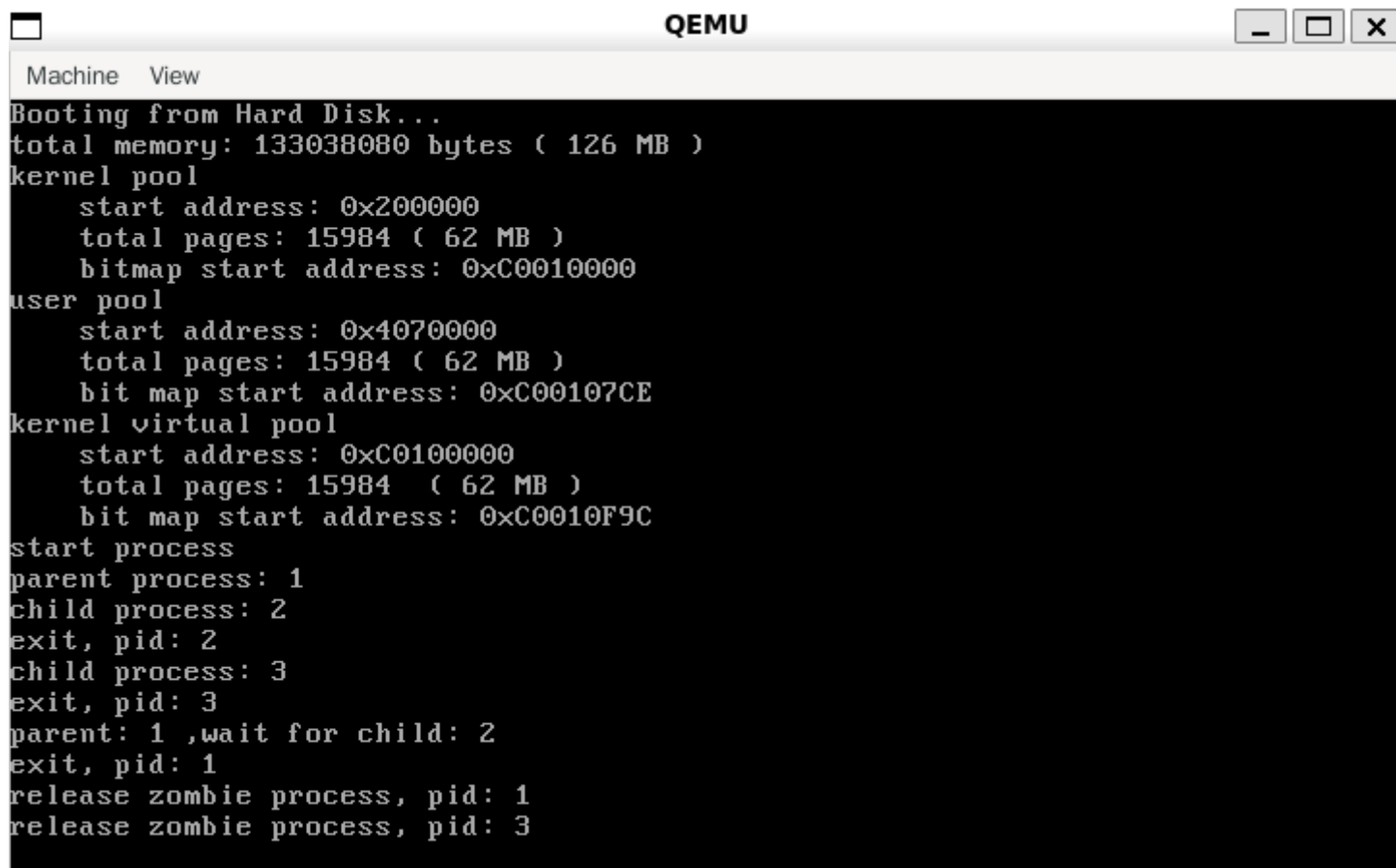
```

1 void first_process()
2 {
3     printf("parent process: %d\n",programManager.running->pid);
4     int pid1 = fork();
5     int retwal;
6     if (pid1) {
7         int pid2=fork();
8         if (pid2) {
9
10        } else {
11            printf("child process: %d\n",programManager.running->pid);
12        }
13        wait(&retwal);
14    } else {
15        printf("child process: %d\n",programManager.running->pid);
16    }
17 }

```

在此测试样例中，父进程创建了两个子进程，父进程会等待其中一个子进程结束后再结束，然后父进程和另外一个子进程在结束后都成了僵尸进程，通过定期调用回收函数回收。

输出结果如下



The image shows a QEMU window titled "QEMU" with standard window controls. The main area displays the boot output of a virtual machine. The text is as follows:

```

Machine  View
Booting from Hard Disk...
total memory: 133038080 bytes ( 126 MB )
kernel pool
  start address: 0x200000
  total pages: 15984 ( 62 MB )
  bitmap start address: 0xC0010000
user pool
  start address: 0x4070000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC00107CE
kernel virtual pool
  start address: 0xC0100000
  total pages: 15984 ( 62 MB )
  bit map start address: 0xC0010F9C
start process
parent process: 1
child process: 2
exit, pid: 2
child process: 3
exit, pid: 3
parent: 1 ,wait for child: 2
exit, pid: 1
release zombie process, pid: 1
release zombie process, pid: 3

```

可以看到，父进程 `pid=1` 创建了两个子进程 `pid=2`、`3`，父进程在 `wait` 中回收了子进程 `pid=2`，父进程和 `pid=3` 都成为了僵尸进程，随后通过定期调用回收函数回收了僵尸进程。

5 实验总结

- 在文档提供的 `wait`、`exit` 的代码示例中，用户级别的 `printf` 也实现完成，且通过将 `stdio.print` 改变为封装好的 `write` 函数，避免了用户态的直接显存访问，同时也保留了 `printf` 函数在用户态和内核态的一致性，效果比我自己实现的 `user_printf` 要好，这一点是需要改进的地方。
- `fork` 的构造非常巧妙，在创建子进程时是在函数调用中创建，且要把父进程的断点复制给子进程，通过还要通过栈对“中断现场”进行伪造，从而子进程调度时可以直接进入 `load_process` 函数，最终进入用户态。这个过程需要仔细使用 `gdb` 进行 `debug` 来加深理解
- 有了用户态的概念，需要知道在先前使用的许多函数如 `stdio.print`，`schedule` 等函数都是内核态的专有函数，因此这些函数是不得在用户态随意调用的。