《操作系统》实验报告

实验题目	进程管理

一、实验目的

1. 了解第一个用户进程创建过程；

2. 了解系统调用框架的实现机制；

3. 了解 ucore 如何实现系统调用 sys_fork/sys_exec/sys_exit/sys_wait 来进行进程管理。

二、实验项目内容

1. 加载应用程序并执行。do_execve 函数调用 load_icode（位于 kern/process/proc.c 中）来加载并解析一个处于内存中的 ELF 执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好 proc_struct 结构中的成员变量 trapframe 中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。需设置正确的 trapframe 内容。

(1) 请在实验报告中简要说明你的设计实现过程；

(2) 请在实验报告中描述当创建一个用户态进程并加载了应用程序后，CPU 是如何让这个应用程序最终在用户态执行起来的。即这个用户态进程被 ucore 选择占用 CPU 执行（RUNNING 态）到具体执行应用程序第一条指令的整个经过。

2. 父进程复制自己的内存空间给子进程。创建子进程的函数 do_fork 在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。具体是通过 copy_range 函数（位于 kern/mm/pmm.c 中）实现的，请补充 copy_range 的实现，确保能够正确执行。

(1) 请在实验报告中简要说明如何设计实现“Copy on Write 机制”，给出概要设计，鼓励给出详细设计。并指出实现 COW 时我们还需要添加哪种异常类型的处理？

3. 阅读分析源代码，理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现。

(1)请在实验报告中简要说明你对 fork/exec/wait/exit 函数的分析：

(2)请分析 fork/exec/wait/exit 在实现中是如何影响进程的执行状态的？

(3)请给出 ucore 中一个用户态进程的执行状态生命周期图（包括执行状态，执行状态之间的变换关系，以及产生变换的事件或函数调用）

三、实验过程或算法（源程序）

实验过程：

1. 前期准备：连接到 WSL，进入 Ubuntu，启动 Docker 并进入容器，从仓库中克隆不含答案的项目；

2. 修改 Makefile：

```
LAB1    := -DLAB1_EX2 -DLAB1_EX3 #-D_SHOW_100_TICKS -D_SHOW_SERIAL_INPUT
LAB2    := -DLAB2_EX1 -DLAB2_EX2 -DLAB2_EX3
LAB3    := -DLAB3_EX1 -DLAB3_EX2
# LAB4  := -DLAB4_EX1 -DLAB4_EX2
```

需要将 LAB4 注释掉，因为本次实验只实现 LAB3；

需要注意的是，本实验**基于实验二**，因此 clone 不含答案的分支后需要先将实验二代码补全。

3. 加载应用程序并执行：

```
#ifdef LAB3_EX1
/* LAB3:EXERCISE1 YOUR CODE
 * should set tf_era,tf_regs.reg_r[LOONGARCH_REG_SP],tf->tf_prmd
 * NOTICE: If we set trapframe correctly, then the user level process can return to USER MODE fro
 *          tf->tf_prmd should be PLV_USER | CSR_CRMD_IE
 *          tf->tf_era should be the entry point of this binary program (elf->e_entry)
 *          tf->tf_regs.reg_r[LOONGARCH_REG_SP] should be the top addr of user stack (USTACKTOP)
 */
tf->tf_era = elf->e_entry;
tf->tf_regs.reg_r[LOONGARCH_REG_SP] = USTACKTOP;
uint32_t status = 0;
status |= PLV_USER; // set plv=3(User Mode)
status |= CSR_CRMD_IE;
tf->tf_prmd = status;
#endif
```

在 kern/process/proc.c 文件下，do_execve()函数调用 load_icode()函数来加载并解析一个处于内存中的 ELF 执行文件格式的应用程序，建立相应的用户内存空间来放置应用程序的代码段、数据段等，且要设置好 proc_struct 结构中的成员变量 trapframe 中的内容，确保在执行此进程后，能够从应用程序设定的起始执行地址开始执行。设计思路见练习 1。

4. 父进程复制自己的内存空间给子进程：

```
#ifdef LAB3_EX2
/* LAB3 EXERCISE2: YOUR CODE
 * replicate content of page to npage, build the map of phy addr of nage with the linear addr
 *
 * Some Useful MACROs and DEFINEs, you can use them in below implementation.
 * MACROs or Functions:
 *    page2kva(struct Page *page): return the kernel vritual addr of memory which page manage
 *    page_insert: build the map of phy addr of an Page with the linear addr la
 *    memcpy: typical memory copy function
 *
 * (1) find src_kvaddr: the kernel virtual address of page
 * (2) find dst_kvaddr: the kernel virtual address of npage
 * (3) memory copy from src_kvaddr to dst_kvaddr, size is PGSIZE
 * (4) build the map of phy addr of  nage with the linear addr start
 */
  memcpy(page2kva(npage), page2kva(page), PGSIZE);
  page_insert(to, npage, start,perm);
#endif
```

在 kern/mm/pmm.c 文件下，通过 copy_range 函数，创建子进程的函数 do_fork 在执行中将拷贝当前进程（即父进程）的用户内存地址空间中的合法内容到新进程中（子进程），完成内存资源的复制。设计思路见练习 2。

5. 完成以上代码编写后，编译运行该项目，运行结果如下：

```
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125# cd lab3/ucore-loongarch32
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab3/ucore-loongarch32# make clean
  rm -rf dep
  rm -rf boot/loader.o boot/loader boot/loader.bin
  rm -rf obj
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab3/ucore-loongarch32# make
  DEP kern/fs/devs/dev_stdout.c
  DEP kern/fs/devs/dev_stdin.c
  DEP kern/fs/devs/dev_disk0.c
  DEP kern/fs/devs/dev.c
```

```
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab3/ucore-loongarch32# make qemu -j 16
  loongson32_init: num_nodes 1
  loongson32_init: node 0 mem 0x2000000
  ++setup timer interrupts
  (THU.CST) os is loading ...

  Special kernel symbols:
    entry  0xA0000120 (phys)
    etext 0xA0022000 (phys)
    edata 0xA017C620 (phys)
    end    0xA017F900 (phys)
  Kernel executable memory footprint: 1399KB
  memory management: default_pmm_manager
  memory map:
      [A0000000, A2000000]

  freemem start at: A01C0000
  free pages: 00001E40
  ## 00000020
```

```
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_slab() succeeded!
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: stride_scheduler
proc_init succeeded
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
initproc exit.
Lab3 Check Pass!
```

可以看到父进程 fork 子进程后进程调度情况。

实验算法函数源代码：

练习 1. 加载应用程序并执行

```
#ifdef LAB3_EX1
    tf->tf_era = elf->e_entry;//set tf_era
    tf->tf_regs.reg_r[LOONGARCH_REG_SP] = USTACKTOP; //set
tf_regs.reg_r[LOONGARCH_REG_SP]
    uint32_t status = 0; //initialize status=0
    status = status | PLV_USER; //set plv=3(User Mode)
    status = status | CSR_CRMD_IE;
    tf->tf_prmd = status;//set tf->tf_prmd
#endif
```

（1）**设计实现过程**：此处完成一个优先级的转变，从内核态切换到用户态（即特权级从 0 到 3）。先解释一下代码中变量 tf 的含义：tf 是一个是中断帧的指针，指向内核栈的某个位置，当进程从用户空间跳到内核空间时，中断帧记录了进程在被中断前的状态。当内核需要跳回用户空间时，需要调整中断帧以恢复让进程继续执行的各寄

存器值。tf 的定义在 kern/trap/trap.h 中。优先级转变的具体做法如下：

① 将 tf_era 设置为用户态，这个定义在 kern/mm/memlayout.h 中，有一个宏定义已经定义了用户态和内核态；

② status 也需要设置为用户态；

③ 需要将 tf_regs.reg_r[LOONGARCH_REG_SP]设置为用户栈的栈顶，直接使用之前建立用户栈时的参数 USTACKTOP 就可以；

④ 最后打开中断。

（2）**创建一个用户态进程并加载了应用程序后，CPU 让这个应用程序最终在用户态执行起来的全过程**如下：

① 调用 mm_create 函数申请进程内存管理的数据结构 mm 所需的内存空间，并对 mm 初始化；

② 调用 setup_pgdir 函数，申请一个页目录表所需的一个页大小的内存空间，并把描述 ucore 内核虚空间映射的内核页表的内容拷贝到此新目录表中，最后 mm->pgdir 指向此页目录表，也就是进程新的页目录表了，且能够正确映射内核；

③ 根据应用程序执行码的起始位置来解析此 ELF 格式的执行程序，并调用 mm_map 函数根据 ELF 格式的执行程序说明的各个段（代码段、数据段、BSS 段等）的起始位置和大小建立对应的 vma 结构，并把 vma 插入到 mm 结构中，从而表明了用户进程的合法用户态虚拟地址空间；

④ 调用根据执行程序各个段的大小分配物理内存空间，并根据执行程序各个段的起始位置确定虚拟地址，在页表中建立好物理地址和虚拟地址的映射关系。然后把执行程序各个段的内容拷贝到相应的内核虚拟地址中，至此应用程序执行码和数据已经根据编译时设定地址放置到虚拟内存中了；

⑤ 需要给用户进程设置用户栈，为此调用 mm_mmap 函数建立用户栈的 vma 结构，明确用户栈的位置在用户虚空间的顶端，大小为 256 个页，即 1MB，并分配一定数量的物理内存且建立好栈的虚地址-物理地址映射关系；

⑥ 至此，进程内的内存管理 vma 和 mm 数据结构已经建立完成，于是把 mm->pgdir 赋值到 cr3 寄存器中，即更新了用户进程的虚拟内存空间，此时的 initproc 已经被 hello 的代码和数据覆盖，成为了第一个用户进程，但此时这个用户进程的执行现场还没建立好；

⑦ 先清空进程的中断帧，再重新设置进程的中断帧，使得在执行中断返回指令“iret”后，能够让 CPU 转到用户态特权级，并回到用户态内存空间，使用用户态的代码段、数据段和堆栈，且能够跳转到用户进程的第一条指令执行，并确保在用户态能够响应中断。

至此，CPU 让这个应用程序最终以用户态执行起来。

练习 2. 父进程复制自己的内存空间给子进程

```
#ifdef LAB3_EX2
    memcpy(page2kva(npage), page2kva(page), PGSIZE);
    page_insert(to, npage, start, perm);
#endif
```

（1）**“Copy on Write 机制”概要设计：**

fork()之后，kernel 把父进程中所有的内存页的权限都设为 read-only，然后子进程的地址空间指向父进程。当父子进程都只读内存时，相安无事。当其中某个进程写内存时，CPU 硬件检测到内存页是 read-only 的，于是触发页异常中断，陷入 kernel

的一个中断例程。中断例程中，kernel 就会把触发的异常的页复制一份，于是父子进程各自持有独立的一份。

具体来说，实现时，在 fork 一个进程时，可以省去 load_icode 中创建新页目录的操作，而是直接将父进程页目录的地址赋给子进程，为了防止误操作以及辨别是否需要复制，应该将尚未完成复制的部分的访问权限设为只读。当执行读操作，父进程和子进程均不受影响。但当执行写操作时，会发生权限错误（因为此时的访问权限为只读）。这时候会进入到 page fault 的处理中去，在 page fault 的处理中，如果发现错误原因读/写权限问题，而访问的段的段描述符权限为可写，便可以知道是由于使用 COW 机制而导致的，这时再将父进程的数据段、代码段等复制到子进程内存空间上即可。

实现 COW 时，我们还需要添加分页错误，即页异常中断的处理。

练习 3.理解进程执行 fork/exec/wait/exit 的实现，以及系统调用的实现

（1）fork/exec/wait/exit 影响进程的执行状态如下：

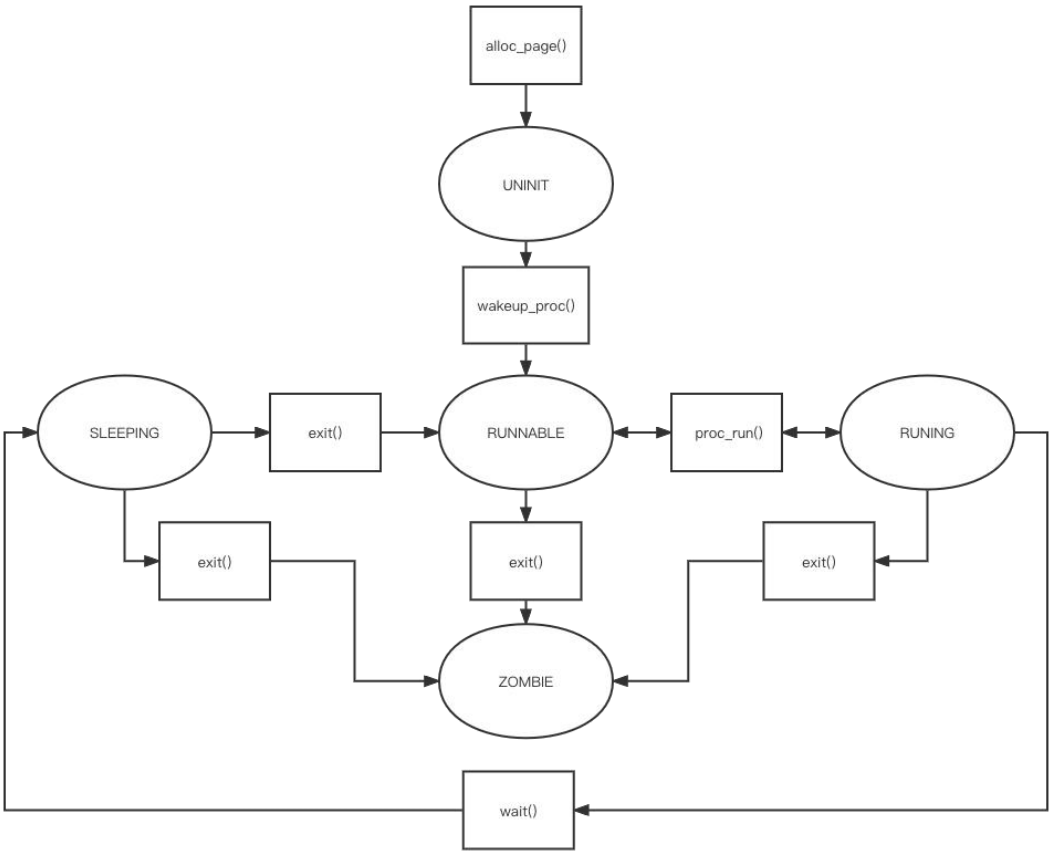
• 父进程 fork 将创建新的子线程，将子线程的状态由 UNINIT 态变为 RUNNABLE 态，不改变父进程的状态；

• exec 完成用户进程的创建工作，同时使用户进程进入执行，不改变进程状态；

• wait 完成子进程资源回收，如果有已经结束的子进程或者没有子进程，那么调用会立刻结束，不影响进程状态；否则，进程需要等待子进程结束，进程从 RUNNIG 态变为 SLEEPING 态；

• exit 完成对资源的回收，进程从 RUNNIG 态变为 ZOMBIE 态。

（2）ucore 中一个用户态进程的执行状态生命周期图如下：



四、实验结果及分析

实验结果截图：

```
root@DESKTOP-ELVOHAA:/mnt/c/Users/Jackson1125/lab3/ucore-loongarch32# make qemu -j 16
loongson32_init: num_nodes 1
loongson32_init: node 0 mem 0x2000000
++setup timer interrupts
(THU.CST) os is loading ...

Special kernel symbols:
  entry  0xA0000120 (phys)
  etext 0xA0022000 (phys)
  edata 0xA017C620 (phys)
  end   0xA017F900 (phys)
Kernel executable memory footprint: 1399KB
memory management: default_pmm_manager
memory map:
    [A0000000, A2000000]

freemem start at: A01C0000
free pages: 00001E40
## 00000020
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
check_slab() succeeded!
kmalloc_init() succeeded!
check_vma_struct() succeeded!
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: stride_scheduler
proc_init succeeded
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 3
I am the parent, waiting now..
I am the child.
waitpid 3 ok.
exit pass.
all user-mode processes have quit.
init check memory pass.
initproc exit.
Lab3 Check Pass!
QEMU: Terminated
```