第六章．系统调度，IPC 和页面失效控制（lab4）(v0.1)

6.1. 实验目标

MIT 这次实验是在 Lab3 进程和中断管理的基础上实现，目标是在他们的 JOS 操作系统中实现多进程管理和进程间消息通信的功能。

在实验三中，我们知道进程是一个执行中的程序实例。利用分时技术，操作系统上同时可以运行多个进程。分时技术的基本原理是把 CPU 的运行时间划分成一个个规定长度的时间片（实验中一个时间片为 100ms），让每个进程在一个时间片内运行。当进程的时间片用完时系统舅利用调度程序切换到另一个进程去运行。当一个进程在执行时，CPU 的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文。当内核需要切换至另一个进程时，它需要保存当前进程的所有状态，即保存当前进程的上下文，以便在再次执行该进程时，能够恢复到切换时的状态执行下去。

在操作系统的进程调度方式中有抢占式和非抢占式，本实验中采用抢占式进程调度，即现行进程在运行过程中，如果有重要或紧迫的进程到达（其状态必须为就绪），则现运行进程将被迫放弃处理机，系统较处理机立刻分配给新到达的进程，其需要时钟中断处理程序实现。其中如何产生多个进程以及如何进行进程切换是本实验的目标。

程序的代码比较零散，主要集中在 kern 和 lib 目录下。该实验可以分为 3 部分：实现调度算法、创建新的进程环境和进程间通信。第一部分通过循环轮转（Round-Robin）调度算法实现多用户进程；第二部分通过实现类似于 Unix 进程创建的 fork（）函数创建新的进程以及实现用户态下的缺页错误处理函数；最后通过时钟中断实现用户进程间的消息通信等。

本实验中的函数在 kern 下主要是完成以下文件：

kern/sched.c
kern/syscall.c
kern/trapentry.S
kern/trap.c
kern/env.c

在 lib 目录下需要完成的文件包括：

lib/pfentry.S
lib/pgfault.c
lib/fork.c
lib/ipc.c

在本次实验过程中没有检查函数，不过 JOS 给出了一些用户进程，在实验中运行这些进程和文档说明中的结果对照，如果出现问题可以利用 bochs 设置断点查看相关的错误，也可以用 cprintf 打印相关信息，来查看是否运行正确。只有每次运行正确之后才能继续下面

的实验，确保错误只出现在当前代码中。

6.2. 背景知识

通过实验二和实验三，JOS 系统在完成了内存分页管理以及进程结构初始化和中断初始化之后，已经可以简单的运行一个进程，并可以使用中断系统进行系统调用了。在主要函数 i386_init()中，对进程结构初始化和中断初始化之后，又增加了支持多进程环境的两个函数 pic_init();和 kclock_init();，前者是初始化 8259A 的中断控制器，为后面的时钟中断做准备，后者是初始化 8253 可编程定时器，用来产生时钟中断（系统定义每秒 100 次，即 8253 被设置成每隔 10ms 就发出一个时钟中断信号）。

在 JOS 系统内核启动之后，系统创建了一个用户 idle 进程，即传统意义上的守护进程（一个周期性运行或总是等待某个事件的后台进程），是系统中的后台服务进程，它是一个生存期较长的进程，通常独立于控制终端并且周期性地执行某种任务或等待处理某些发生的事件。守护进程常常在系统引导装入时启动，在系统关闭时终止。Linux 系统有很多守护进程，大多数服务都是通过守护进程实现的。在 JOS 系统中，idle 进程周期性的调用调度函数，调度执行就绪的进程。

在系统启动了用户进程之后，内核函数 i386_init();调用调度函数 sched_yield();开始执行第一个用户进程并周期性的执行就绪的用户进程。

我们知道，现代操作系统都是多用户多进程环境，因此，当前 JOS 只运行一个进程是远远不够的。在本实验中，我们让 JOS 支持多进程（像现代操作系统那样），通过创建新的进程以及实现简单的多进程调度算法来实现。

6.3. 多进程管理的实现

在这个实验中，主要是实现调度算法以及创建新的进程，并实现用户态下的 frok 函数以及用户态的页错误处理函数，对于前一部分，主要是在 NENV（1024）个进程中循环轮转实现多进程管理以及实现系统调用中的进程创建；而后一部分主要是讨论如何实现用户态进程创建新的进程。

6.3.1 多进程管理

在 JOS 系统中，主要通过实现一些新的系统调用函数来允许用户进程创建新的进程，并实现简单的 Round-Robin 调度算法允许内核当进程自愿放弃 CPU 或者退出时切换到另一个进程，在后面我们还会实现时钟中断控制下的可剥夺调度。下面我们分别对调度算法和新的系统调用进行讨论。

6.3.1.1 调度算法

在 JOS 中，Round-Robin 调度算法把全局变量 envs（见实验 3 报告中 3.1.2）当作循环数组进行搜索，跳过 idle 进程，只有当没有进程可运行时，运行 idle 进程；调度的时机即当进程自愿放弃 CPU 或者由于某件事件阻塞时也即系统进行系统切换，后一种涉及到了抢占式调度，比如当前进程在 I/O 时并没有用到处理器，此时内核需要强制暂停当前进程，执行另一个就绪进程。

在 JOS 系统中，我们主要通过一个函数实现进程调度，即 kern/sched.c 中的 sched_yield() 函数，下面对此函数进行说明。

void **sched_yield**(void)

首先，实验 4 开始的第一个进程 envs[0]就是特殊的 idle 进程。在本函数中，通过搜索 envs 数组寻找一个新的进程去执行，从前一个执行的进程开始，直到找到一个就绪的进程；当没有就绪进程时，才重新调度 idle 进程。

在完成 sched_yield（）之后一定要将此函数加入到系统调用中，允许用户进程调用调度函数。

6.3.1.2 创建进程相关的系统调用

实验 3 中，我们只创建了一个进程运行，并没有允许用户进程产生新的子进程，我们需要在 JOS 系统允许用户进程通过系统调用创建新的用户进程。

传统上的 Unix 提供 fork（）系统调用创建进程，创建出来的子进程复制了父进程的整个地址空间，除了进程号不同，在父进程中 fork（）返回子进程的进程号，在子进程中返回 0，以此来区分父子进程。我们所要讨论的进程创建与 Unix 的类似。

由于进程创建往往涉及到内核操作，如果用户态允许创建进程的话，往往涉及到一些特权指令，比如进程的状态设置和内存分配；还有就是容易产生恶意代码对系统进行破坏；，因此在创建进程时，我们需要用户进程陷入到内核态执行。以下即是我们需要实现 kern/syscall.c 中的函数：

static envid_t **sys_exofork**(void)

static int **sys_env_set_status**(envid_t envid, int status)

static int **sys_page_alloc**(envid_t envid, void *va, int perm)

static int **sys_page_map**(envid_t srcenvid, void *srcva,envid_t dstenvid, void *dstva, int perm)

static int **sys_page_unmap**(envid_t envid, void *va)

这里，我们首先对 JOS 系统中要使用的函数进行说明。

int envid2env(envid_t, struct Env **, bool)

//将一个 envid 转换成指向 env 的一个指针，在 fork（）中调用本函数一定要把 bool 设为 1，以此来说明设置的进程要么是父进程，要么是当前进程的子进程。

下面对这些函数进行说明。

static envid_t **sys_exofork**(void)

这是 JOS 实现进程创建中的最重要的一个函数，创建一个新的进程，其寄存器状态和当前父进程寄存器状态一样，并标记为 ENV_NOT_RUNNABLE，在父进程中，sys_exofork 返回创建子进程的 envid_t,而在子进程中返回 0.

注意：当进程调用 sys_exofork 创建子进程时所有的寄存器的值都被复制到了子进程里，包括 eip 的值，这样当子进程继续执行的时候如果它的内容与父进程完全相同，它将从父进程执行系统调用的下一条指令处开始执行，而此处的汇编指令应该是将 eax 的值复制给一个变量留待后续作为返回值返回，这样如果在系统调用 sys_exofork 中将新创建的子进程的 eax 寄存器置为 0 就将使得子进程继续执行时认为其系统调用的返回值为 0。

static int **sys_env_set_status**(envid_t envid, int status)

一旦进程的地址空间和寄存器状态都初始化之后，用来标示一个新的进程的执行状态信

息。设置 envid_t 进程状态为 status（ENV_RUNNABLE 或者 ENV_NOT_RUNNABLE）。

static int **sys_page_alloc**(envid_t envid, void *va, int perm)

分配一页物理内存，并将它以 perm 属性映射到 envid 进程 va 所对应的地址空间中。

注意：在分配物理内存时，需要将页内容初始化为 0，以防止脏数据产生异常（系统有时在释放内存时并没有清除内存信息，一些内存中还保留有以前的信息，对内存清零增强了系统的正确性），对后面实验造成影响。

static int **sys_page_map**(envid_t srcenvid, void *srcva,envid_t dstenvid, void *dstva, int perm)

将 srcenvid 进程地址空间中的线性地址 srcva 的页映射到 dstenvid 进程地址空间中的 dstva 地址处，并设置页属性为 perm。此处的操作并非数据拷贝而是页表操作，即两进程共享同一个页的地址。如果使用数据拷贝，当一个进程有大量的数据时，而子进程只是用到了一小部分的话就会浪费内存空间，而且降低了系统的效率。使用页表操作则不仅提高了效率还增强了系统的灵活性。

static int **sys_page_unmap**(envid_t envid, void *va)

取消进程 envid 地址空间中线性地址 va 所对应页的映射，即删除线性地址 va 所对应的物理页面。

6.3.2 用户态 fork 和 page fault

在本节中，主要讨论 JOS 系统中用户态进程创建子进程。

在此，我们需要了解 fork（）函数中用到的 COW 技术。在 fork（）系统调用中需要复制父进程的地址空间到子进程中，所谓复制，只是进程的基本资源的复制，如 task_struct 数据结构、系统空间堆栈、页面表等等。传统的 fork（）系统调用直接把所有的资源复制给新创建的进程。这种实现效率比较低下，因为它拷贝的数据也许并不共享，而且，如果新进程打算执行一个新的映像，那么所有的拷贝都将前功尽弃。JOS 系统中 fork（）使用写时拷贝（copy-on-write）页实现。

写时拷贝是一种可以推迟甚至免除拷贝数据的技术，即内核此时并不复制整个进程地址空间，而是让父进程和子进程共享同一个拷贝。只有在需要写入的时候，数据才会被复制，从而使各个进程拥有各自的拷贝。也就是说，资源的复制只有在需要写入的时候才进行，在此之前，只是以只读方式共享。这种技术使地址空间上的页的拷贝被推迟到实际发生写入的时候。在页根本不会被写入的情况下，它们就无需复制了。fork（）的实际开销就是复制父进程的页表以及给子进程创建惟一的进程描述符。在系统中，进程创建后都会马上运行一个可执行的文件，这种优化可以避免拷贝大量根本就不会被使用的数据。

由于用户态 fork 涉及到页表出错的处理，即当创建的进程引用一个不存在页面中的内存地址时，就会触发 CPU 产生页错误异常中断，并把引起中断的线性地址放到 CR2 控制寄存器中。因此处理该中断的过程就可以知道发生页异常的确切地址，从而可以把进程要求的页面从二级存储空间加载到物理内存中。我们首先要讨论的就是关于用户级别的页错误处理。

6.3.2.1 用户级别的页错误处理

在处理用户级的页错误时，由于进程运行在用户级别，即 Ring3，而内存操作需要在特权级别，因此，我们在处理用户级别的页错误时需要用到系统调用陷入到特权级别。

本实验中，由于进程增加了页错误处理，相应的进程数据结构增加了一项（可参看实验3报告中的5.3.1.1原数据结构）：

```
// Exception handling
void *env_pgfault_upcall;    // page fault upcall entry point
//用于记录页错误处理函数的入口
```

为了完成用户级页错误处理我们需要完成以下几件事情，让内核知道进程页错误信息并在处理完成后直接返回用户态进程：

1.用户注册 user level 的 page fault 处理函数到 env 环境中供内核调用。

2.内核处理部分为 user level 的 page fault 处理函数设置 exception stack 使得 user level 处理函数处理完后直接返回用户态出错代码执行，并使得内核返回到 user level 的 page fault 处理函数处执行。

3.user level 的 page fault 处理函数具体处理 page fault。

首先，为了处理用户态的页错误，需要在 JOS 内核中注册一个页错误处理函数，用 env 结构中新增加的 **env_pgfault_upcall** 记录此信息。具体我们要实现 kern/syscall.c 下面的 ses_env_set_pgfault_upcall 函数，此函数比较简单，将用户态自己的处理页错误函数注册到进程结构中。

其次，我们知道用户进程运行在用户栈，其 esp 指针指向 USTACKTOP，其堆栈数据保存在 USTACKTOP-PGSIZE 和 USTACKTOP-1 之间；而当发生页错误时，内核将指定异常处理程序运行在指定的页错误处理堆栈上，即 user exception stack.其有效的地址是 UXSTACKTOP-PGSIZE 和 UXSTACKTOP-1 之间，在此堆栈上运行时，用户级页处理函数可以使用 JOS 的系统调用解决引起页错误的问题，然后直接返回用户态出错代码继续执行。

在 JOS 系统中，我们将处于页错误处理的用户进程状态称为 trap-time。在本实验中，为此状态增加了一个新的数据结构 Utrapframe，以记录进程页错误之前的信息，利用此结构返回到出错进程：

```
struct UTrapframe {
    /* information about the fault */
    uint32_t utf_fault_va;    /* va for T_PGFLT, 0 otherwise */
    uint32_t utf_err;
    /* trap-time return state */
    struct PushRegs utf_regs;
    uintptr_t utf_eip;
    uint32_t utf_eflags;
    /* the trap-time stack to return to */
    uintptr_t utf_esp;
};
```

此结构与 Trapframe 基本上类似，就是为了保存 Trapframe 中相关的信息。通过程序我们知道此结构用来保存用户态出错代码的寄存器信息，当页错误程序处理完后直接返回用户态出错代码继续执行（此处设计的比较灵活），后续我们会详细介绍。

我们需要修改 kern/trap.c 中的 page_fault_handler(struct Trapframe *tf)来处理用户级别的页错误，使得其把 page faults 分发给用户态的处理函数。设置 exception stack 时应如下图所示，与上文提到的 Utrapframe 对应起来：

```
                    <-- UXSTACKTOP
trap-time esp
trap-time eflags
trap-time eip
trap-time eax       <-- start of struct PushRegs
trap-time ecx
trap-time edx
trap-time ebx
trap-time esp
trap-time ebp
trap-time esi
trap-time edi       <-- end of struct PushRegs
tf_err (error code)
fault_va            <-- %esp when handler is run
```

图 6-1. 进入页错误异常时的 UXSTACKTOP

图 6-1 即为 UXSTACKTOP 中异常堆栈的设置，一个 Utrapframe 结构，内核使用这个 stack frame 从页错误处理返回到用户进程，其中 fault_va 为引起页错误的虚拟地址。

在 page_fault_handler()函数中，我们用 UTrapframe 结构保存用户态出错程序相关寄存器信息，然后将用户堆栈换为异常堆栈，并将 UTrapframe 结构压入异常堆栈（用于返回用户进程），并使当前页错误处理堆栈指向 UTrapframe，即异常堆栈栈顶，在具体页错误处理函数里使用。设置完堆栈信息就开始执行具体的页错误处理函数了。

注意：在页错误处理时也有可能发生嵌套的页错误，如果发生，我们只需要在异常堆栈中压入一个空字（32 位），然后再是一个 UTrapframe 结构，判断是否发生嵌套页错误，只需要检查当前堆栈是否已经是在 UXSTACKTOP-PGSIZE 和 UXSTACKTOP-1 之间了。

最后，我们要实现 user level 的页错误处理函数，需要完成 lib/pfentry.S 以及 lib/pgfault.c 文件。下面对文件中的函数进行说明：

在 lib/pfentry.S 中的_pgfault_upcall 汇编函数，首先是调用具体的页错误处理函数，然后利用在 trap.c 中压入的 UTrapframe 结构变换到调用缺页处理进程的上下文。首先变换到调用缺页处理的堆栈，将保存的缺页处理进程的 eip 压入堆栈，然后空一个字出来保存后面的堆栈信息以免覆盖堆栈信息。剩下的就是换回到原来的堆栈，返回到调用缺页处理进程了。

注意：该函数是页错误处理的关键，是用户态页错误处理函数的总入口，负责调用具体的 page fault 处理函数（该函数由 lib/pgfault.c 中的代码注册到进程的 env 结构中），并返回到引起 page fault 的代码处执行，而不必经过内核，因为在上文中的 Utrapframe 结构中保存了出错前进程的状态，因此可以直接返回。难点是同时变换堆栈和 EIP。

文件 lib/pgfault.c 实现了用户程序可调用的页错误处理函数的设置函数，包含用户异常栈的初始化和页错误处理函数的设置，利用前面实现的 sys_env_set_pgfault_upcall（）函数注册用户页错误处理函数。

由以上分析，我们知道了用户进程中的页错误处理过程，其流程如下图 6-2 所示：

图 6-2.页错误处理流程

在页错误实验过程中，为了保证大家自己写的程序段的正确性，JOS 系统安排了一些用户函数，以进行页错误处理检查，只有这些用户函数执行结果与程序相同才能继续下面的实验。

6.3.2.2Copy-on-Write Fork

通过 6.2 关于 COW 的讨论以及上面页错误处理函数的完成，接下来我们需要完成 JOS 系统中用户进程创建子进程的 Fork（）函数，其中包括：

static void **pgfault**(struct UTrapframe *utf)

static int **duppage**(envid_t envid, unsigned pn)

envid_t **fork**(void)

下面对这些函数进行说明。

static void **pgfault**(struct UTrapframe *utf)

具体的 page fault 处理函数，根据情况分配新页或者进行其他工作。当出现页错误时，主要是对标为可写的或者 COW 的页面分配新的页面，复制旧页的数据到新页并映射到旧页的地址处。

static int **duppage**(envid_t envid, unsigned pn)

将父进程的页表空间映射到子进程中，即共享数据，并都标记为 COW，为了以后任一进程写数据时产生页错误，为其分配新的一页。

envid_t **fork**(void)

这里的 fork 是创建新进程的总入口，使用 **env_alloc()**创建一个新的进程，然后扫描父进程的整个地址空间将其映射到子进程相关的页表中，对于父进程，返回子进程的进程号，对于子进程，返回 0。fork 的流程如下：

1. 首先调用 set_pgfault_handler 函数对 pgfault 处理函数进行注册。
2. 调用 sys_exofork 创建一个新的进程。
3. 映射可写或者 COW 的页都为 COW 的页。
4. 为子进程分配 exception stack。
5. 为子进程设置用户级的页错误处理句柄。//此处为容易遗忘的地方
6. 标记子进程为 runnable。

在完成以上函数之后，我们可以用 user/forktree 来测试我们的 fork（）函数，每一个子进程又创建子进程，直到三层结束……

6.4. 可剥夺调度与 IPC

完成了前面的实验后，我们通过用户进程 user/spin 发现其创建的子进程一旦运行，一直死循环，父进程和内核都不能得到 CPU 继续运行了，这显然不是操作系统应该有的情况。为了允许内核抢占一个正在运行的进程，在实验 4 的最后部分，我们需要修改内核，使其在时钟中断中调用调度函数，实现可剥夺的调度，并完成进程间的消息通信。

6.4.1 可剥夺调度

可剥夺调度也就是抢占调度方式，在这种调度方式中，进程调度程序可根据某种原则停止正在执行的进程，将已分配给当前进程的处理机收回，重新分配给另一个处于就绪状态的进程。在 JOS 中我们的抢占原则是时间片原则：即各进程按系统分配给的一个时间片运行，当该时间片用完或由于该进程等待某事件发生而被阻塞时，系统就停止该进程的执行而重新进行调度。此处需要用到外部中断，时钟中断。

外部中断（如：设备中断）一般是指 IRQs，在每台 PC 的系统中，是由一个中断控制器 8259 或是 8259A 的芯片来控制系统中每个硬件的中断。目前共有 16 组 IRQ，去掉其中用来作桥接的一组 IRQ，实际上只有 15 组 IRQ 可供硬件调用。在 picirq.c 中 JOS 系统将 IRQs0-15 映射到了 IDT 表中的 IRQ_OFFSET 到 IRQ_OFFSET+15（即 32-47）。

在 JOS 系统中，在内核态时一直是禁止外部中断的，外部中断是由 eflags 标志寄存器的 FL_IF 位控制的，在 JOS 中我们只在进入和离开用户模式时保存和恢复 eflags 寄存器。并确保 FL_IF 标志在用户进程运行时置位，这样当时钟中断时，内核可以得到处理器继续调度其他进程。

我们需要修改 kern/trapentry.S 和 kern/trap.c 来初始化 IDT 中的外部中断，所有的外部中断统一调用调度函数，系统中每隔 10HZ 发生一次时钟中断，这样，内核可以获得处理器来调度其他进程。其初始化请参看实验 3 报告中的 5.4.2.1 节。

6.4.2 IPC

在前面，JOS 系统主要是考虑操作系统隔离性，即每一个进程都好像独占一台机器一样，而操作系统另一个重要的服务，进程之间相互通信即 IPC（inter-process communication）：进程间通信。允许进程通信，进程间可以合作完成一个整体的任务。

传统上有许多进程间通信的方式，比如信号量，管道等，我们在 JOS 系统中使用简单

的消息传递来实现进程间通信。在 JOS 的 IPC 机制中传送和几首的消息包括两类：一个是单字和一个页的映射；后者可以高效传送大量的数据。

本实验中，为了完成进程间通信在进程数据结构中增加了相应的 IPC 数据结构：

```
// Lab 4 IPC
bool env_ipc_recving;		// env is blocked receiving
void *env_ipc_dstva;		// va at which to map received page
uint32_t env_ipc_value;		// data value sent to us
envid_t env_ipc_from;		// envid of the sender
int env_ipc_perm;		// perm of page mapping received
//以上数据结构用来保存 IPC 时的相关信息，如发送者进程 id，发送的数据等。
```

实现 IPC 我们首先需要完成 kern/syscall.c 中的 sys_ipc_recv 和 sys_ipc_can_send 系统调用，其说明如下：

static int **sys_ipc_recv**(void *dstva)

阻塞调用进程直到接收到一个消息，然后设置 env 中相应项，调用调度函数。

注意：此处要设置返回值 eax 为 0，即永远不执行 return 语句，直接调用调度函数即可。

static int **sys_ipc_try_send**(envid_t envid, uint32_t value, void *srcva, unsigned perm)

发送一个消息到 envid 进程，当 srcva 为 0 时，传送一个单字，否则传送一页，即将当前进程 srcva 地址处的页映射到接收进程同一地址处。

注意：在以上函数中当调用 envid2env 时设置 checkperm 为 0，这样任何进程都可以发送消息给任何其他进程，内核仅仅检查目标进程是否存在。

一个进程可以调用 sys_ipc_recv 接收一个消息，这个系统调用将会阻塞当前进程，除非它接收到一个消息。单一个进程等待接收一个消息时，其他任何进程都可以发送消息给它，在 JOS 系统中没有特殊的要求。与接收一个消息类似，一个进程调用 sys_ipc_can_send 发送一个消息给指定的进程，如果指定的进程正在阻塞等待消息，则发送消息并返回 0。

在完成了系统调用中的 IPC 之后，为了使用户进程可以相互通信，我们需要完成用户态下的 IPC，即 lib/ipc.c 文件

int32_t **ipc_recv**(envid_t *from_env_store, void *pg, int *perm_store)

使用 sys_ipc_revc 系统调用接收消息，并返回接收的 value。

void **ipc_send**(envid_t to_env, uint32_t val, void *pg, int perm)

使用 sys_ipc_try_send 系统调用发送 val 到进程 to_env，直到发送成功。

至此，我们可以运行用户进程 user/pingpong 和 user/primes 来测试进程之间传送消息。

6.5. 常见错误以及调试经验

在完成实验 4 的过程中，我们遇见了一些错误，总结一下调试经验，希望以后做这个实验的人注意一下!!

1. Lab4 是整个实验中比较关键的部分，包含了 os 内核的核心概念和实现，并且承接了 2 和 3 的内容。在实验中，要确保前一阶段的工作完全正确，再继续往下做，

不要累计错误一起调试。

2. 设置调试点时，往往用到 printf 输出信息，因为涉及文件较多，可以在 printf 语句中增加该调试点所处的文件和函数。
3. 在完成 sys_exofork 函数时，一定要细心，在创建子进程时一定也要把页错误处理句柄复制给子进程，否则，后面用户态 fork 时错误很难发现，因为当用户态的页错误时其找不到页错误处理句柄，会出现异常。
4. 在本次实验中，每次编译时经常会报用户进程错误，而每次加一条语句比如：printf 语句，或者 exit()语句就通过了，这个错误很诡异，后面发现是编译器的问题，换成其他 Linux 系统就 ok 了。