1. 了解&使用开发环境

使用target=i686-elf作为gcc, as, ld交叉编译套件

以及QEMU（x86模拟器）

开发将运行在保护模式中（PMode）

- 使用更大的32位寄存器
- 使用更多的内存，到达4GB
- 能够执行C语言编译过来的机器码

具体环境配置

1. 使用自动环境安装脚本

sh gcc-build.sh(跑大概半个小时左右)

gcc-build.sh为下列代码

```
#! /usr/bin/bash
sudo apt update &&\
     sudo apt install -y \
        build-essential \
        bison\
        flex\
        libgmp3-dev\
        libmpc-dev\
        libmpfr-dev\
        texinfo

BINUTIL_VERSION=2.37
BINUTIL_URL=https://ftp.gnu.org/gnu/binutils/binutils-2.37.tar.xz

GCC_VERSION=11.2.0
GCC_URL=https://ftp.gnu.org/gnu/gcc/gcc-11.2.0/gcc-11.2.0.tar.xz

GCC_SRC="gcc-${GCC_VERSION}"
BINUTIL_SRC="binutils-${BINUTIL_VERSION}"

# download gcc & binutil src code

export PREFIX="$HOME/cross-compiler"
export TARGET=i686-elf
export PATH="$PREFIX/bin:$PATH"

mkdir -p "${PREFIX}"
mkdir -p "${HOME}/toolchain/binutils-build"
mkdir -p "${HOME}/toolchain/gcc-build"

cd "${HOME}/toolchain"

if [ ! -d "${HOME}/toolchain/${GCC_SRC}" ]
```

```
then
    (wget -O "${GCC_SRC}.tar" ${GCC_URL} \
        && tar -xf "${GCC_SRC}.tar") || exit
    rm -f "${GCC_SRC}.tar"
else
    echo "skip downloading gcc"
fi

if [ ! -d "${HOME}/toolchain/${BINUTIL_SRC}" ]
then
    (wget -O "${BINUTIL_SRC}.tar" ${BINUTIL_URL} \
        && tar -xf "${BINUTIL_SRC}.tar") || exit
    rm -f "${BINUTIL_SRC}.tar"
else
    echo "skip downloading binutils"
fi

echo "Building binutils"

cd "${HOME}/toolchain/binutils-build"

("${HOME}/toolchain/${BINUTIL_SRC}/configure" --target=$TARGET --
prefix="$PREFIX" \
    --with-sysroot --disable-nls --disable-werror) || exit

(make && make install) || exit

echo "Binutils build successfully!"

echo "Building GCC"

cd "${HOME}/toolchain/gcc-build"

which -- "$TARGET-as" || echo "$TARGET-as is not in the PATH"

("${HOME}/toolchain/${GCC_SRC}/configure" --target=$TARGET --prefix="$PREFIX" \
    --disable-nls --enable-languages=c,c++ --without-headers) || exit

(make all-gcc &&\
 make all-target-libgcc &&\
 make install-gcc &&\
 make install-target-libgcc) || exit

echo "done"
```

2. 添加环境变量

经测试完成上面的操作后,在root/cross-compiler中为编译完成的gcc交叉编译环境的二进制等文件将cross-compiler整个文件夹复制到用户的home文件夹中,打开etc/profile永久添加环境变量

```
sudo vi etc/profile
# 添加以下内容
export PATH="/home/<你的用户名>/cross-compiler/bin:$PATH"
```

3. 检测环境配置是否成功

执行

```
i686-elf-gcc -dumpmachine
```

输出结果应为

```
i686-elf
```

2. 多进程/任务

实现多进程需要知道：

1. 知道一个进程在哪里被切走，这样我们才能在稍后恢复执行。（保证连续性）
2. 知道进程被切走前一瞬间内的所有状态，并且能够正确的恢复这些状态！（保证正确性）
3. 知道什么时候应该进行切换，并且确保每个进程都会被执行。（保证公平性）

对于连续性
只需要记录程序指针

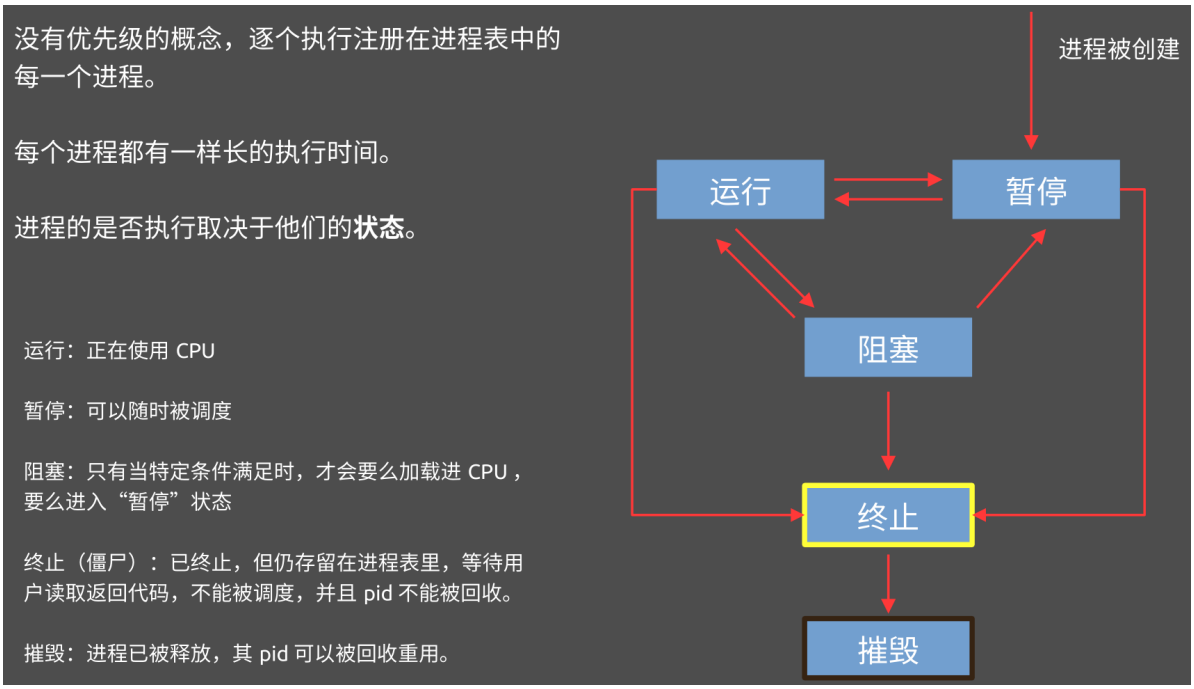
对于正确性
中断上下文环境以及虚拟内存

虚拟内存已经实现进程内存隔离
那么每个进程都有自己的一套页表，切换页表 = 切换内存 = 切换进程的所有运行时状态

中断的触发为信号，进行上下文切换

轮询式调度器

为保证公平，使用**轮询式调度器（Round-Robin Scheduler）**



```
55  void schedule() {
56      if (!sched_ctx.ptable_len) {
57          return;
58      }
59
60      struct proc_info* next;
61      int prev_ptr = sched_ctx.procs_index;
62      int ptr = prev_ptr;
63      // round-robin scheduler
64      do {
65          ptr = (ptr + 1) % sched_ctx.ptable_len;
66          next = &sched_ctx._procs[ptr];
67      } while(next->state != PROC_STOPPED && ptr != prev_ptr);
68
69      sched_ctx.procs_index = ptr;
70
71      run(next);
72  }
```

利用iret实现上下文切换:

```
37
38  void run(struct proc_info* proc) {
39      if (!(__current->state & ~PROC_RUNNING)) {
40          __current->state = PROC_STOPPED;
41      }
42      proc->state = PROC_RUNNING;
43
44      __current = proc;
45
46      cpu_lcr3(__current->page_table);
47
48      apic_done_servicing();
49
50      asm volatile (
51          "pushl %0\n"
52          "jmp soft_iret\n"::"r"(&__current->intr_ctx): "memory");
53  }
54
```

PCB结构：

并发物理内存访问

内存共享与引用计数

```
You, 5 days ago | 1 author (You)
struct pp_struct {
    pid_t owner;
    uint32_t ref_counts;
    pp_attr_t attr;
};
```

记录物理页的所有者

引用次数（为 0 则被视为释放）

页属性（预留，暂时没用）

释放一个页就递减引用次数。

同理，分配一个页则递增。

```
int
pmm_free_page(pid_t owner, void* page)
{
    struct pp_struct* pm = &pm_table[(intptr_t)page >> 12];

    // Is this a MMIO mapping or double free?
    if (((intptr_t)page >> 12) ≥ max_pg || !(pm->ref_counts)) {
        return 0;
    }

    pm->ref_counts--;
    return 1;
}
```

记录权限

```
You, 3 days ago | 1 author (You)
struct proc_mm {
    heap_context_t u_heap;
    struct mm_region* regions;
};
```

用户栈

内存分区链表

```
You, 3 days ago | 1 author (You)
struct mm_region
{
    struct llist_header head;
    unsigned long start;
    unsigned long end;
    unsigned int attr;
};
```

这里记录了权限

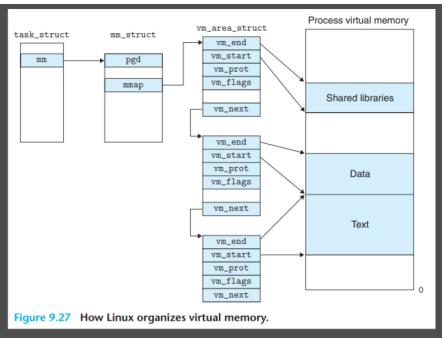


Figure 9.27 How Linux organizes virtual memory.

这和 Linux 的 vm_area_struct 有着异曲同工之妙

```
/** ...
#define REGION_PRIVATE      0x0

/** ...
#define REGION_RSHARED      0x1

/** ...
#define REGION_WSHARED      0x2
```

Fork

fork() 的功能就是将进程一分为二。
fork() 是仅有的能返回两次的函数

把父进程的一切复制过来
注意:

- pid 等进程唯一的属性不能直接复制过来
- 任何栈空间需要进行完整拷贝
- 对于任何读共享的内存区域，需要同时将父进程和子进程的对应映射标记为只读，从而保证 COW 的应用

```
void* __dup_pagetable(pid_t pid, uintptr_t mount_point) {
    void* ptd_pp = pmm_alloc_page(pid, PP_FGPERSIST);
    x86_page_table* ptd = vmm_fmap_page(pid, PG_MOUNT_1, ptd_pp, PG_PREM_RW);
    x86_page_table* pptd = (x86_page_table*) (mount_point | (0x3FF << 12));

    for (size_t i = 0; i < PG_MAX_ENTRIES - 1; i++)
    {
        x86_pte_t ptde = pptd->entry[i];
        if (!ptde || !(ptde & PG_PRESENT)) {
            ptd->entry[i] = ptde;
            continue;
        }

        x86_page_table* ppt = (x86_page_table*) (mount_point | (i << 12));
        void* pt_pp = pmm_alloc_page(pid, PP_FGPERSIST);
        x86_page_table* pt = vmm_fmap_page(pid, PG_MOUNT_2, pt_pp, PG_PREM_RW);

        for (size_t j = 0; j < PG_MAX_ENTRIES; j++)
        {
            x86_pte_t pte = ppt->entry[j];
            pmm_ref_page(pid, pte & ~0xfff);
            pt->entry[j] = pte;
        }

        ptd->entry[i] = (uintptr_t)pt_pp | PG_PREM_RW;
    }

    ptd->entry[PG_MAX_ENTRIES - 1] = NEW_L1_ENTRY(T_SELF_REF_PERM, ptd_pp);

    return ptd_pp;
}
```

子进程PCB创建：

```
pid_t dup_proc() {
    pid_t pid = alloc_pid();

    struct proc_info pcb = (struct proc_info) {
        .created = clock_systime(),
        .pid = pid,
        .mm = __current->mm,
        .intr_ctx = __current->intr_ctx,
        .parent = __current
    };

#ifdef USE_KERNEL_PG…
#else
    setup_proc_mem(&pcb, PD_REFERENCED);
#endif

    // 根据 mm_region 进一步配置页表
    if (!__current->mm.regions) {
        goto not_copy;
    }
```

复制与初始化地址空间：

```
void setup_proc_mem(struct proc_info* proc, uintptr_t usedMnt) {
    // copy the entire kernel page table
    pid_t pid = proc->pid;
    void* pt_copy = __dup_pagetable(pid, usedMnt);

    vmm_mount_pd(PD_MOUNT_2, pt_copy);    // 将新进程的页表挂载到挂载点#2

    // copy the kernel stack
    for (size_t i = KSTACK_START >> 12; i ≤ KSTACK_TOP >> 12; i++)
    {
        volatile x86_pte_t *ppte = &PTE_MOUNTED(PD_MOUNT_2, i);

        /*…
        cpu_invplg(ppte);

        x86_pte_t p = *ppte;
        void* ppa = vmm_dup_page(pid, PG_ENTRY_ADDR(p));
        *ppte = (p & 0xfff) | (uintptr_t)ppa;
    }
```

注意：重写PTE时需要刷新TLB缓存

mm_region复制：

```
    struct mm_region *pos, *n;
    llist_for_each(pos, n, &__current->mm.regions->head, head) {
        region_add(&pcb, pos->start, pos->end, pos->attr);

        // 如果写共享，则不作处理。
        if ((pos->attr & REGION_WSHARED)) {
            continue;
        }

        uintptr_t start_vpn = PG_ALIGN(pos->start) >> 12;
        uintptr_t end_vpn = PG_ALIGN(pos->end) >> 12;
        for (size_t i = start_vpn; i < end_vpn; i++)
        {
            x86_pte_t *curproc =
                &((x86_page_table*)(PD_MOUNT_1 | ((i & 0xffc00) << 2)))->entry[i & 0x3ff];
            x86_pte_t *newproc =
                &((x86_page_table*)(PD_MOUNT_2 | ((i & 0xffc00) << 2)))->entry[i & 0x3ff];

            if (pos->attr == REGION_RSHARED) {
                // 如果读共享，则将两者的都标注为只读，那么任何写入都将会应用COW策略。
                *curproc = *curproc & ~PG_WRITE;
                *newproc = *newproc & ~PG_WRITE;      You, 1 second ago • Uncommitted chang
            }
            else {
                // 如果是私有页，则将该页从新进程中移除。
                *newproc = 0;
            }
        }
    }

not_copy:
    vmm_unmount_pd(PD_MOUNT_2);

    // 正如同fork，返回两次。
    pcb.intr_ctx.registers.eax = 0;

    push_process(&pcb);

    return pid;
```

3. 内存管理

物理内存管理

分配可用叶(用于映射)
对合适的页进行Swap操作

记录所有物理页的可用性

4GiB大小 = 2^20 个记录

位图bitmap
0 - 可用, 1 - 已占用
需要 128Ki的空间

虚拟内存管理

管理映射 - 增删改查

- 增： VA <---> PA
- 删：删除映射（删除对应表项，释放占用的物理页，刷新 TLB ）
- 改：修改映射
- 查： VA ----> PA

虚拟地址解决方法:递归映射

malloc

堆空间

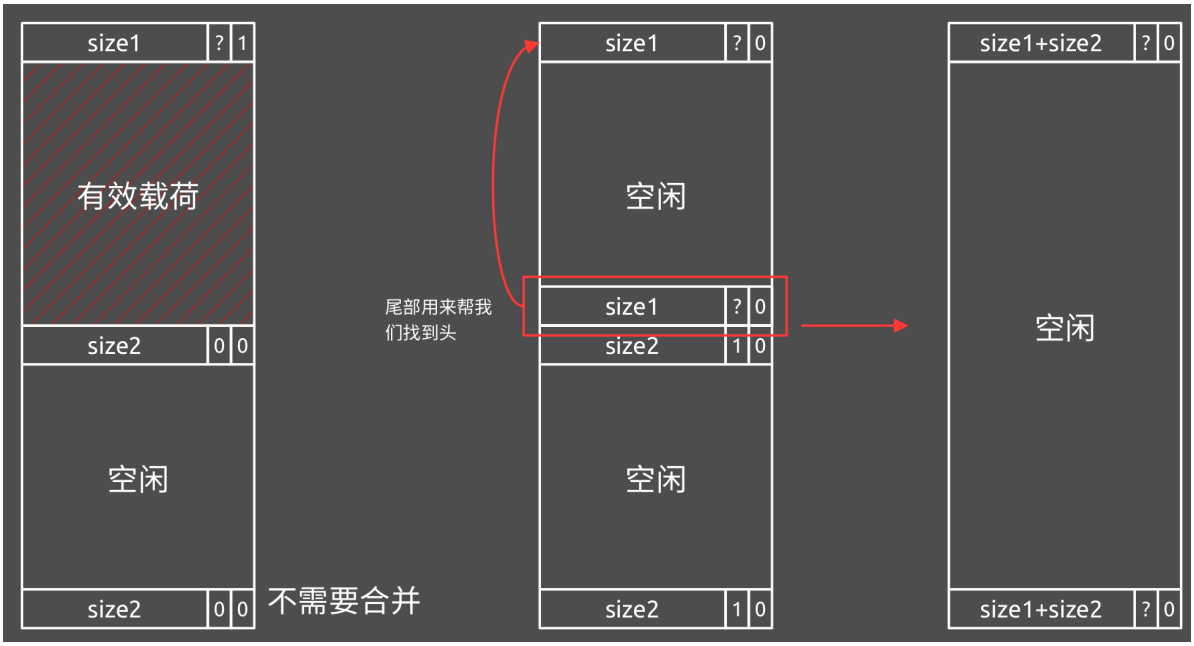
动态地,按需创建的空间

规定，所有分配的空间大小必须为 4 的倍数（地址 4 字节对齐）

如何处理虚碎片

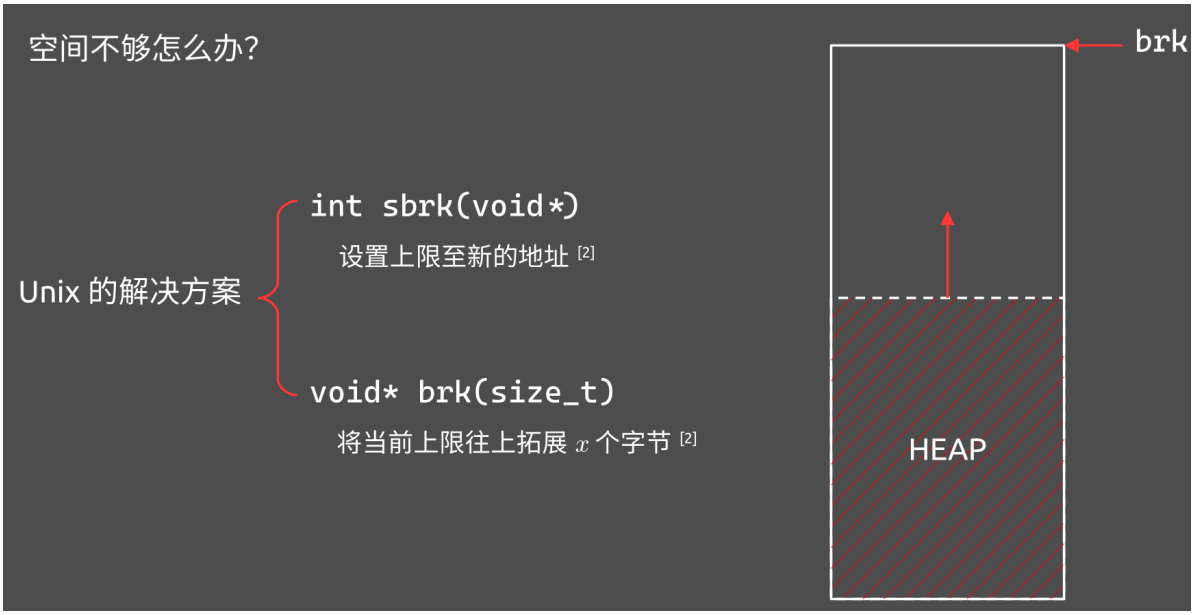
Implicit Free List

边界标签法（ Boundary Tag ）——在头尾加上标签，写上一些必要元数据

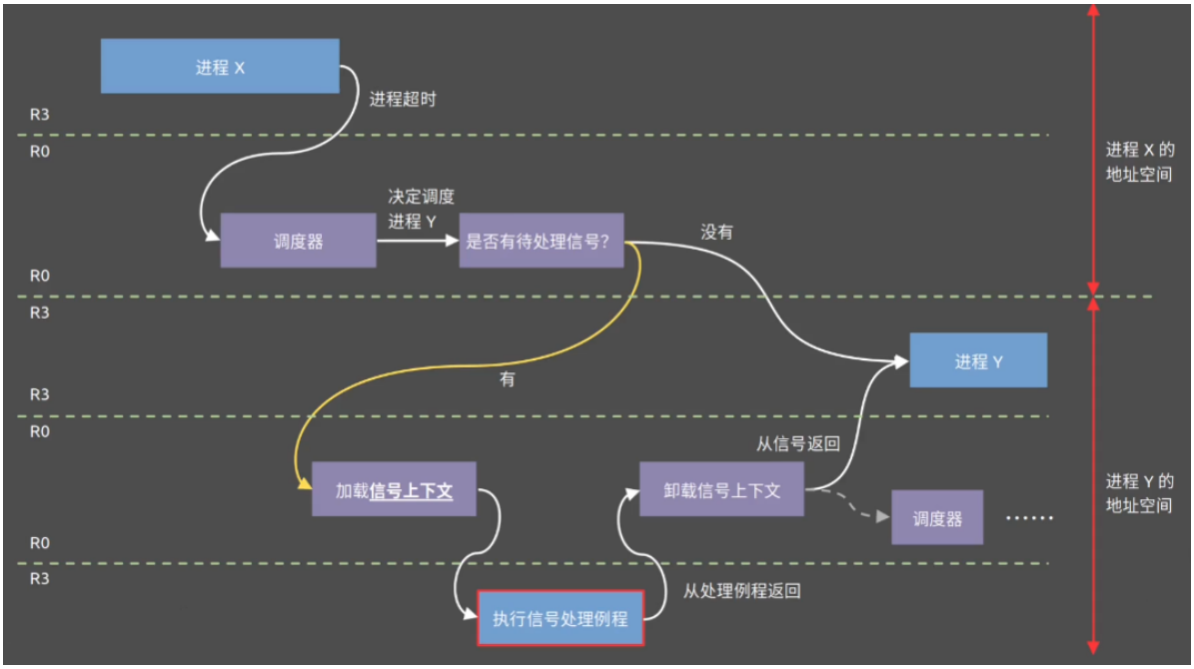


malloc具体实现

类似Unix的操作



4. 同步的仿真



表达已送达的信号: 位图法

检测并选中需要运行的信号处理例程: 用户自定义例程或默认例程的地址

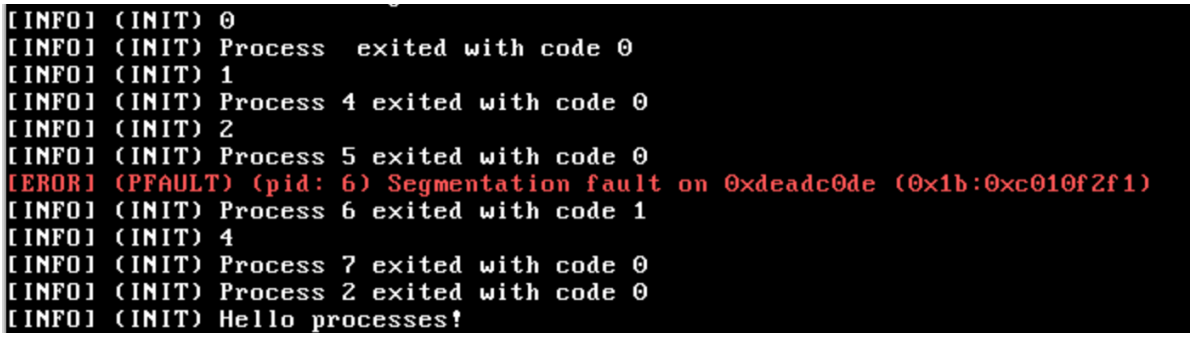
每次切换前更新TSS

仿真demo

```
#include <lunaix/lunistd.h>
#include <lunaix/proc.h>
#include <lunaix/signal.h>
#include <lunaix/spike.h>
#include <lunaix/syslog.h>
#include <lunaix/types.h>

LOG_MODULE("SIGDEMO")

void __USER__
sigchild_handler(int signum)
{
    kprintf(KINFO "SIGCHLD received\n");
}

void __USER__
sigsegv_handler(int signum)
{
    pid_t pid = getpid();
    kprintf(KWARN "SIGSEGV received on process %d\n", pid);
    _exit(signum);
}

void __USER__
sigalrm_handler(int signum)
{
    pid_t pid = getpid();
    kprintf(KWARN "I, pid %d, have received an alarm!\n", pid);
}
```

```
void __USER__
_signal_demo_main()
{
    signal(_SIGCHLD, sigchild_handler);
    signal(_SIGSEGV, sigsegv_handler);
    signal(_SIGALRM, sigalrm_handler);

    alarm(5);

    int status;
    pid_t p = 0;

    kprintf(KINFO "Child sleep 3s, parent pause.\n");
    if (!fork()) {
        sleep(3);
        _exit(0);
    }

    pause();

    kprintf("Parent resumed on SIGCHILD\n");

    for (int i = 0; i < 5; i++) {
        pid_t pid = 0;
        if (!(pid = fork())) {
            signal(_SIGSEGV, sigsegv_handler);
            sleep(i);
            if (i == 3) {
                i = *(int*)0xdeadc0de; // seg fault!
            }
            kprintf(KINFO "%d\n", i);
            _exit(0);
        }
        kprintf(KINFO "Forked %d\n", pid);
    }

    while ((p = wait(&status)) >= 0) {
        short code = WEXITSTATUS(status);
        if (WIFSIGNALED(status)) {
            kprintf(KINFO "Process %d terminated by signal, exit_code: %d\n",
                    p,
                    code);
        } else if (WIFEXITED(status)) {
            kprintf(KINFO "Process %d exited with code %d\n", p, code);
        } else {
            kprintf(KWARN "Process %d aborted with code %d\n", p, code);
        }
    }

    kprintf("done\n");

    spin();
}
```

仿真输出



5. 文件系统

暂时还没有实现，说一下 **想法思路**

引导扇区

引导扇区包含引导程序和FAT12文件系统的整个组成结构信息，这些信息描述了FAT文件系统对磁盘扇区的管理情况

FAT表

FAT12中每个簇就是512个字节。另一方面，如果文件比较大，需要占据多个簇时，文件不一定在连续的簇内，这时就需要一种方法寻找到完整的文件，这个功能由FAT表完成。

FAT12对应的表项位宽就是12

表项0和1的值是无关紧要的。其他表项每个对应数据区的一个簇，而表里保存的数据是下一个簇的簇号，这样，就形成了一个链表一样的结构了。具体的，表项保存的数据有以下的取值：

000h：可用簇
002h-FEFh：已用簇，标识下一个簇的簇号
FF0h-FF6h：保留簇
FF7h：坏簇
FF8h-FFFh：文件最后一个簇

根目录区与数据区

根目录的开始扇区号是19，它由最多BPB_RootEntCnt个目录项。这些目录项可能存储了指向文件和目录的信息。目录项是一个32B的结构体。它的结构如下：

偏移量	长度	描述
0	8	文件名
8	3	文件扩展名
11	1	文件属性
12	10	保留位
22	2	创建时间
24	2	创建日期
26	2	首簇号
28	4	文件大小

数据区不仅可以保存目录项信息，也可以保存文件内的数据。对于树状的目录结构，树的层级结构可以通过目录的目录项简历起来，从根目录开始，经过目录项的逐层嵌套，形成了树状结构。

实现思路

1. 预先实现磁盘的读写功能(DMA, SATA:FIS, AHCI, SLAB分配器)
2. DEFINE引导扇区
3. 初始化FAT表结构并对FAT表的增删改查操作进行编写
4. 对根目录区与数据区使用树状结构（文件目录树？）进行编写，同样需要有对应增删改查的方法
5. 首先加载引导扇区与FAT表，再初始化根目录区与数据区
6. 查找文件时需要先查看根目录区是否有匹配的目录，有则通过对应目录项的首段簇获取其目录文件的首簇号；接着通过FAT表获得改目录文件的全部内容，遍历该文件，一次偏移32字节继续查找目录项，匹配查询路径中对应的项。如果查到则类似1，2查找对应的目录文件及目录项，否则说明找不到，结束；如果在倒数第一层目录文件中找到了被查文件的目录项，从中获取首簇号，即可通过fat表访问该文件整个相关簇。