

# 操作系统实验报告 6

蔡日骏 12348003

2014 年 5 月 16 日

## 目录

<b>1 概述</b>	<b>1</b>
<b>2 构建指南</b>	<b>1</b>
2.1 编译	2
2.2 安装	2
<b>3 内核架构</b>	<b>3</b>
3.1 源文件目录	3
3.2 基本架构	4
3.3 定时器子系统	6
3.4 任务调度子系统	7
<b>4 运行演示</b>	<b>10</b>
4.1 前台任务	10
4.2 后台任务	10
4.3 多任务	10

## 1 概述

本实验中实现了一个保护模式多任务分页内核 AssignmentOS。其中多任务使用 x86 硬件多任务实现。由于文件系统部分尚未实现，目前用户程序只能与内核一同编译。另外，由于中断驱动程序尚未实现双缓冲输出，因此在用户态尚未支持多屏、分屏显示。目前版本只能作多任务演示，完整功能的内核将在以后版本中实现。

## 2 构建指南

AssignmentOS 编译需要一个默认目标为 i386-elf 的 GCC、GNU Make、gzip。安装时则需要 dd、mount/pmound 等程序进行软盘镜像操作。此外，AssignmentOS 还需要 GRUB Legacy 或 GRUB 2 等支持 Multiboot 标准的引导器进行启动。

## 2.1 编译

由于已有 `Makefile`，因此可以直接运行 `make` 命令进行编译。但是为了让 `make` 能够找到需要的构建工具链，一般需要先进行环境变量设置。可以设置的环境变量有两个，一个是 `CC`，用于指定需要使用的 C 编译器（目前仅在默认目标为 `i386-elf` 的 GCC 4.8.2 上进行过测试）；另一个是 `EXTRAFLAGS`，用于指定额外的编译参数（默认为空，可以使用 `-O2` 打开编译优化，`-g` 添加调试信息）。

编译完成后会自动进行内核压缩。

## 2.2 安装

编译出来的内核需要安装在软盘上，由 GRUB Legacy 或 GRUB 2 引导。下面以 FAT12 格式软盘、GRUB Legacy (0.97) 为例。

1. 准备 FAT 格式软盘镜像。

```
$ dd if=/dev/zero of=disk.img count=2880
$ mkfs -t fat -F 12 disk.img
```

2. 挂载软盘镜像，把 AssignmentOS 内核镜像和 GRUB 镜像文件复制进去，挂在软盘镜像可以使用直接使用 `mount` 程序，但是这样要处理权限问题。更简单的方法是使用用户态的挂载程序，下面以 `pmount` 为例。

```
$ pmount disk.img
$ mkdir -p /media/disk.img/boot/grub
$ cp kernel.gz /media/disk.img/boot
$ cp menu.lst /media/disk.img/boot/grub
$ cd /usr/lib/grub
$ cp stage1 fat_stage1_5 stage2 /media/disk.img/boot/grub
```

其中 GRUB 的配置文件 `menu.lst` 内容如下。

```
timeout 5
default 0
color light-blue/black light-cyan/blue

title AssignmentOS Version 1.0
root (fd0)
kernel /boot/kernel.gz
```

3. 安装 GRUB。安装前需要使用 `sync` 命令刷新系统的 IO 缓冲区保证前面的文件正确写入软盘镜像。

```
$ sync
$ grub
```

在 GRUB Shell 中运行以下命令。

```
grub> device (fd0) disk.img
grub> root (fd0)
grub> setup (fd0)
```

随后刷新缓冲区即可。

## 3 内核架构

下面对 AssignmentOS 内核的架构进行简单介绍。

### 3.1 源文件目录

```
.
|-- kernel
|   |-- boot
|   |   |-- boot.asm          内核头部
|   |   |-- Makefile
|   |-- drivers              驱动程序目录
|   |   |-- keyboard_map.c    键盘映射信息
|   |   |-- Makefile
|   |   |-- tty.c            终端驱动程序
|   |-- include
|   |   |-- constants.h      常数定义
|   |   |-- interrupts.h     中断子系统头文件
|   |   |-- kernel.h         内核全局头文件
|   |   |-- keyboard.h       键盘驱动程序头文件
|   |   |-- keyboard_map.h   键盘映射信息头文件
|   |   |-- mm.h             内存管理子系统头文件
|   |   |-- scheduler.h      任务调度子系统头文件
|   |   |-- timer.h          定时器子系统头文件
|   |   |-- tty.h            终端驱动程序头文件
|   |   |-- utils.h          工具函数头文件
|   |-- interrupts
|   |   |-- idt.c             中断描述符表操作
|   |   |-- int_handlers.asm  中断处理函数包装
|   |   |-- keyboard.c       键盘驱动程序
|   |   |-- Makefile
|   |   |-- pic.c            可编程中断控制器操作
|   |   |-- timer.c          定时器子系统
|   |-- mm
```

```

| | |-- Makefile
| | |-- mm.c          内存管理子系统
| | |-- paging.c      分页控制
| |-- scheduler
| | |-- Makefile
| | |-- scheduler.c    任务调度子系统
| |-- utils
| | |-- Makefile
| | |-- utils.asm      工具函数
| |-- link.ld          内核链接脚本
| |-- main.c          内核主函数
| |-- Makefile
| |-- programs.c      演示用户程序
| |-- shell.c         Shell程序
|-- Makefile

```

### 3.2 基本架构

AssignmentOS 是一个保护模式的内核，使用分页机制进行内存管理，任务调度使用 x86 硬件多任务方式进行上下文切换。内核格式为 ELF32 格式，根据 Multiboot 标准进行引导。

GRUB 把 AssignmentOS 内核加载到内存 0x100000 起始的位置并跳转到 `boot.asm` 中的内核入口点 `_start`，`boot.asm` 进行全局描述符表的初始化后跳转到 `main.c` 中的内核主函数 `kmain` 进行进一步的初始化。`kmain` 完成内核初始化后会加载 Shell 程序，等待调度器把控制权交给 Shell 程序，至此系统启动完成。

`boot.asm` 的文本段如下。

```

section .text
global _start
_start:
    mov esp, stack_top      ; 把 ESP 指向内核堆栈栈顶

    lgdt [gdtr]             ; 加载全局描述符表

    mov ax, 2<<3            ; 更新数据段寄存器
    mov ds, ax
    mov es, ax
    mov fs, ax
    mov gs, ax
    mov ss, ax

    jmp 0x8:after_lgdt      ; 更新数据段寄存器
after_lgdt:

```

```

push ebx                ; 向内核主函数传递 Multiboot 数据
extern kmain
call kmain              ; 调用内核主函数

.hang:
hlt                    ; 初始化完成后在这里挂起，等待调度器接管
jmp .hang

kmain 内核主函数如下。

void kmain(void *mb_info) /* mb_info 为 GRUB 传递过来的 Multiboot 数据 */
{
    cli();              /* 关闭中断 */
    init_variables();    /* 初始化某些内核全局变量 */
    enable_paging();     /* 启动内存分页 */
    init_mm(((uint32_t *)mb_info)[2]); /* 初始化内存管理子系统 */
    init_idt();          /* 初始化中断描述符表 */
    init_pic();          /* 初始化可编程中断控制器 */
    init_timer();        /* 初始化定时器子系统 */
    init_kb();           /* 初始化键盘驱动程序 */
    init_scheduler();     /* 初始化任务调度子系统 */

    init_task(&sh_task, 0x8, 0x10, 0x10); /* 初始化 Shell 进程 */
    tty_open(&sh_tty);                  /* 为 Shell 进程打开终端 */
    sh_task.tty = &sh_tty;
    sh_task.tss.eip = (uint32_t)sh_main; /* 更新指令指针 */
    add_task(&sh_task);                  /* 把 Shell 进程添加到调度队列 */

    start_scheduler();                  /* 启动任务调度器 */
    sti();                             /* 打开中断 */
}

```

系统中每个进程都拥有独立的分页结构，但其中 0x0-0x3FFFFFF，即第一个页表管辖的内存区域是共享且线性映射的。这个共享区域为内核态内存，存放系统调用和任务调度等所需的代码和数据。因此 AssignmentOS 至少需要 4M 物理内存。0x400000 开始的内存区域主要为用户态内存，但也有部分内存页为内核态内存（如 0 号特权环堆栈）。每个用户程序可以认为自己被加载到 0x400000 的地方，堆栈从 0xFFFFFFFF 地址处开始。

下面主要对 AssignmentOS 的定时器和任务调度两个子系统进程介绍。虚拟内存管理、中断处理键盘、中断等子系统并不是本实验的重点，在当前版本的 AssignmentOS 中也不是十分成熟，将在以后的实验中进行逐步介绍。

### 3.3 定时器子系统

定时器子系统用于高效、方便地处理系统中（包括内核和用户程序）大量的计时操作。AssignmentOS 的定时器子系统由可编程间隔计时器（Programmable Interval Timer）触发的 IRQ0 控制。

定时器子系统初始化时，把 PIT 的 0 号通道设置到 2 号工作模式（Rate Generator），并把 PIT 的计数寄存器设置为 0x2E9C，然后在中断描述符表中设置 IRQ0 的处理程序。此后，IRQ0 将会每 10 毫秒触发一次，对应的处理程序将会把“系统心跳”变量 CLOCK 增加 1。

为了高效、方便地处理多个定时请求，定时器子系统使用一个二叉最小堆保存所有定时器。新的定时请求将被插入到堆中。而每次 IRQ0 触发时处理程序将会把当前的系统心跳与堆根的定时器比较，并根据需要弹出堆中的定时器进行处理。

定时器子系统定义的数据结构如下。

```
typedef void (*timer_handler_t)(void *);    /* 定时器回调函数指针类型 */

struct Timer_t {                            /* 定时器结构 */
    timer_handler_t handler;                /* 定时器回调函数指针 */
    void *arg;                              /* 回调函数的参数指针 */
    uint32_t target_time;                   /* 定时器的目标时间 */
    uint32_t interval;                     /* 定时器间隔，用于自动重置计时器 */
};

struct TimerHeap_t {                        /* 定时器堆 */
    struct Timer_t heap[TIMER_HEAP_SIZE];
    size_t size;
};
```

定时器系统的核心部分是 IRQ0 处理程序。由于在中断处理子系统初始化时 Master PIC 被映射到 0x20-0x27 号中断，因此这里也是 0x20 号中断的处理程序。

```
void _inthandler20()
{
    pic_eoi();                             /* 向 PIC 输出 EOI 指令 */
    ++CLOCK;                               /* 更新系统心跳 */
    while(TIMER_HEAP.size && _W(0) <= CLOCK) { /* 检查计时器堆根节点 */
        struct Timer_t tmp = TIMER_HEAP.heap[0];
        timer_heap_pop();                  /* 弹出根节点 */
        if(tmp.interval) {                 /* 自动重置定时器 */
            tmp.target_time = CLOCK + tmp.interval;
            timer_heap_insert(tmp);
        }
        tmp.handler(tmp.arg);              /* 调用定时器的回调函数 */
    }
```

```

    }
}

```

此外，定时器子系统也提供了一个对用户程序很重要的功能——休眠指定时间。该功能的实现需要与3.4节介绍的任务调度子系统协同工作，大致的实现方式是根据需要的休眠时间创建定时器，并阻塞当前进程，当时间到后进行进程唤醒。

```

void timer_wakeup(uint16_t *pid_p)                /* 定时器回调函数 */
{
    wake_up_pid(*pid_p);                          /* 真正进行唤醒操作的函数 */
}

void sleep(uint32_t msec)                         /* 休眠 msec 毫秒 */
{
    struct Timer_t t;
    t.arg = &CURRENT_TASK->pid;                  /* 指定要被唤醒的进程 ID */
    t.handler = (timer_handler_t)timer_wakeup;    /* 唤醒函数 */
    t.interval = 0;                              /* 不进行自动重置 */
    t.target_time = CLOCK + msec / 10;           /* 设置目标时间 */
    timer_heap_insert(t);                        /* 把计时器添加到堆中 */
    wait();                                       /* 阻塞当前进程 */
}

```

二叉最小堆的实现比较简单，这里不作介绍。

### 3.4 任务调度子系统

AssignmentOS 中进程（任务的一种）的状态有 RUNNING、READY、BLOCKED 和 ZOMBIE 四种。同时，设置两个任务队列 `ready` 和 `blocked` 分别用于保存可以运行和被阻塞的任务。

AssignmentOS 中任务上下文切换通过 x86 硬件多任务机制进行，因此在大多数情况下并不需要手动进行状态保存和恢复，只需要把 TSS（任务的上下文数据，PCB 的一部分）的地址等信息写入描述符表中，并进行长跳转或长调用即可。为了方便起见，任务的 PID 被设置成该任务 TSS 的信息在全局描述符表中的索引（即段选择子除以 8）。全局描述符表前 6 项被其他段描述符占据，任务由 6 开始编号。目前 6 号任务被保留，7 号任务为任务调度器，进程的任务编号（相当于 PID）从 8 开始。任务调度子系统的相关数据结构如下。

```

struct Task_t                                     /* 任务结构 */
{
    struct TSS_t tss;                             /* 任务 TSS */
    struct TTY_t *tty;                            /* 任务的终端 */
    uint16_t pid, ppid;                          /* PID, 父进程 PID */
    enum TaskStatus status;                      /* 任务状态 */
    enum Event block_event;                      /* 阻塞原因 */
}

```

```

    int exit_code;                /* 退出码 */
    uint32_t *page_dir;           /* 任务的页目录表 */
    struct Task_t *next;           /* 任务队列链表中的下一项 */
};

struct TaskQueue_t                /* 任务队列结构 */
{
    struct Task_t *head, *tail;    /* 队头和队尾指针 */
    size_t size;                  /* 队列大小 */
};

```

其中 TSS 的数据结构定义与 Intel 的定义一致，此处不在赘述。调度器使用最简单的 Round-robin 调度算法，其核心部分如下。

```

void scheduler()
{
    while(true) {
        cli();
        while(ready.size == 0) {    /* 无可运行任务，挂起 */
            __asm__(
                "sti;"
                "hlt;"
            );
        }
        CURRENT_TASK = ready.head;    /* 设置当前任务指针 */
        CURRENT_TASK->status = RUNNING;
        ready.head = CURRENT_TASK->next; /* 删除队首元素 */
        if(--ready.size == 0)
            ready.tail = NULL;
        farjmp(0, CURRENT_TASK->pid << 3); /* 跳转到任务 */
        sti();
    }
}

```

任务调度子系统初始化时设置自动重置定时器，使每个任务最多拥有 100 毫秒的时间片。时间片用完后，强制进行任务切换。force\_switch 函数是该定时器的回调函数。

```

void force_switch()
{
    if(CURRENT_TASK) {
        add_to_task_queue(&ready, CURRENT_TASK);
        CURRENT_TASK->status = READY;
        TASK_SWITCH();
    }
}

```



```

    }
}

```

`TASK_SWITCH` 是一个宏，用于跳转到 7 号任务（即调度器）。

当进程调用 `sleep` 等函数，或者尝试读取键盘缓冲区但是缓冲区为空时，会导致当前进程被阻塞。在处理任务阻塞和恢复方面，主要由 `wait`<sup>1</sup>和 `wake_up_pid` 两个函数完成。任务被唤醒后并不一定马上被执行，还需要等待调度器处理。

```

void wait()                                /* 阻塞当前进程 */
{
    CURRENT_TASK->status = BLOCKED;
    add_to_task_queue(&blocked, CURRENT_TASK);
    CURRENT_TASK = NULL;
    TASK_SWITCH();
}

void wake_up_pid(uint16_t pid)             /* 唤醒指定 PID 的进程 */
{
    cli();
    struct Task_t *curr = blocked.head, *prev = NULL;
    while(curr) {
        if(curr->pid == pid) {              /* 下面是一些链表操作 */
            if(prev)
                prev->next = curr->next;
            else
                blocked.head = curr->next;
            if(curr->next == NULL)
                blocked.tail = prev;
            if(--blocked.size <= 1)
                blocked.tail = blocked.head;
            curr->block_event = NONBLOCKED;
            curr->status = READY;           /* 设定目标进程的状态 */
            add_to_task_queue(&ready, curr);
            break;
        } else {
            prev = curr;
            curr = curr->next;
        }
    }
    sti();
}

```

---

<sup>1</sup>不同于 POSIX 中定义的 `wait` 系统调用

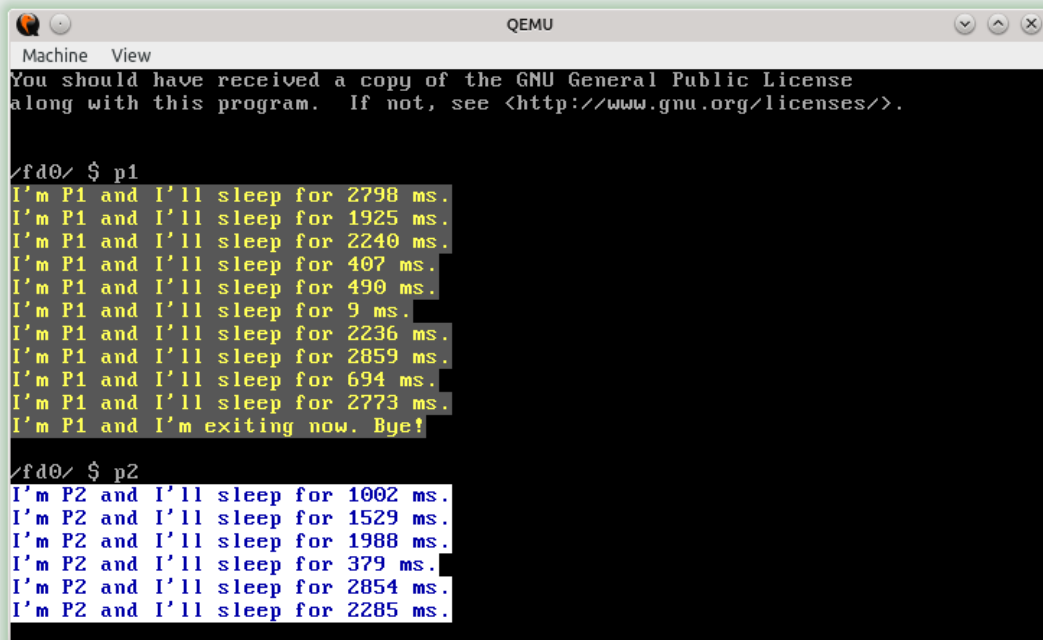
## 4 运行演示

由于文件系统尚未实现，像过去一样直接在指定位置写入数据的方法又可能破坏 FAT12 文件系统结构，因此这个版本的 AssignmentOS 只能直接把几个演示程序的代码跟内核一起编译。当文件系统完成后，将能加载任意用户程序。

目前内置的演示程序有 4 个，分别命名为 p1、p2、p3、p4。每个程序运行时会以不同颜色打印一些信息，然后休眠随机一段时间，重复 10 次，比较适合用来做多任务演示。

### 4.1 前台任务

在控制台中直接输入程序名称并按回车可在前台运行程序。此时 Shell 程序将被阻塞，在该程序退出前不能再输入命令运行其他程序。见图 1。



```
Machine View
You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

/fd0/ $ p1
I'm P1 and I'll sleep for 2798 ms.
I'm P1 and I'll sleep for 1925 ms.
I'm P1 and I'll sleep for 2240 ms.
I'm P1 and I'll sleep for 407 ms.
I'm P1 and I'll sleep for 490 ms.
I'm P1 and I'll sleep for 9 ms.
I'm P1 and I'll sleep for 2236 ms.
I'm P1 and I'll sleep for 2859 ms.
I'm P1 and I'll sleep for 694 ms.
I'm P1 and I'll sleep for 2773 ms.
I'm P1 and I'm exiting now. Bye!

/fd0/ $ p2
I'm P2 and I'll sleep for 1002 ms.
I'm P2 and I'll sleep for 1529 ms.
I'm P2 and I'll sleep for 1988 ms.
I'm P2 and I'll sleep for 379 ms.
I'm P2 and I'll sleep for 2854 ms.
I'm P2 and I'll sleep for 2285 ms.
```

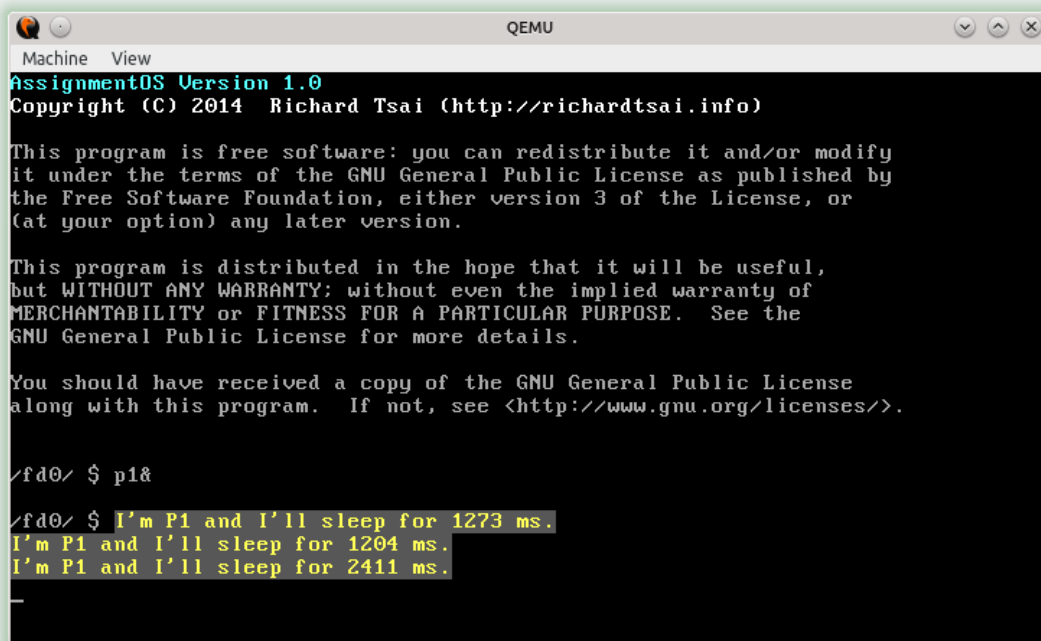
图 1: 前台任务

### 4.2 后台任务

在控制台中输入程序名称 +& 可以在后台运行程序。程序运行后 Shell 程序将可以继续读取用户输入的命令。当然，像 UNIX 的大多数 Shell 一样，在后台运行的程序的输出将会与 Shell 程序的提示符混在一起，但是由于有键盘缓冲区，用户仍然可以继续输入命令。见图 2。

### 4.3 多任务

用户可以输入 p1&-> 回车 ->p2&-> 回车 ->p3&-> 回车p4& 回车（不用管其他进程的输出）同时在后台运行 4 个程序观察多任务运行效果。全部进程退出后可以按回车调出命令提示符。见图 3。



```
Machine View
AssignmentOS Version 1.0
Copyright (C) 2014 Richard Tsai (http://richardtsai.info)

This program is free software: you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation, either version 3 of the License, or
(at your option) any later version.

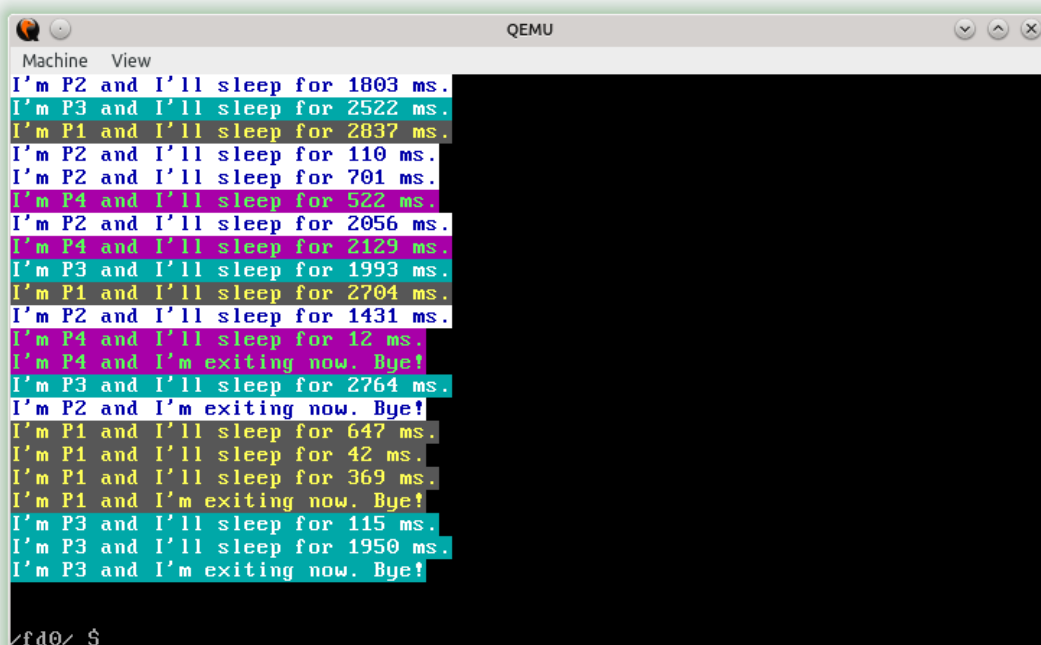
This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program. If not, see <http://www.gnu.org/licenses/>.

/fd0/ $ p1&

/fd0/ $ I'm P1 and I'll sleep for 1273 ms.
I'm P1 and I'll sleep for 1204 ms.
I'm P1 and I'll sleep for 2411 ms.
-
```

图 2: 后台任务



```
Machine View
I'm P2 and I'll sleep for 1803 ms.
I'm P3 and I'll sleep for 2522 ms.
I'm P1 and I'll sleep for 2837 ms.
I'm P2 and I'll sleep for 110 ms.
I'm P2 and I'll sleep for 701 ms.
I'm P4 and I'll sleep for 522 ms.
I'm P2 and I'll sleep for 2056 ms.
I'm P4 and I'll sleep for 2129 ms.
I'm P3 and I'll sleep for 1993 ms.
I'm P1 and I'll sleep for 2704 ms.
I'm P2 and I'll sleep for 1431 ms.
I'm P4 and I'll sleep for 12 ms.
I'm P4 and I'm exiting now. Bye!
I'm P3 and I'll sleep for 2764 ms.
I'm P2 and I'm exiting now. Bye!
I'm P1 and I'll sleep for 647 ms.
I'm P1 and I'll sleep for 42 ms.
I'm P1 and I'll sleep for 369 ms.
I'm P1 and I'm exiting now. Bye!
I'm P3 and I'll sleep for 115 ms.
I'm P3 and I'll sleep for 1950 ms.
I'm P3 and I'm exiting now. Bye!

/fd0/ $
```

图 3: 多任务