

MaQueOS 实验指导书

dslab

2024 年 12 月 13 日

目录

实验一 利用定时器控制飞机移动速度	1	实验四 根据用户输入显示不同颜色的飞机	10
1.1 实验目的	1	4.1 实验目的	10
1.2 实验内容	1	4.2 实验内容	10
1.3 参考资料	1	4.3 参考资料	10
1.4 实验环境	1	4.4 实验环境	10
1.5 实验原理	1	4.5 实验原理	11
1.6 实验步骤	2	4.6 实验步骤	11
1.7 实验结果	2	4.7 实验结果	12
实验二 通过键盘控制飞机移动	4	实验五 顺序读写硬盘扇区	13
2.1 实验目的	4	5.1 实验目的	13
2.2 实验内容	4	5.2 实验内容	13
2.3 参考资料	4	5.3 参考资料	13
2.4 实验环境	4	5.4 实验环境	13
2.5 实验原理	4	5.5 实验原理	14
2.6 实验步骤	5	5.6 实验步骤	14
2.7 实验结果	5	5.7 实验结果	14
实验三 利用系统调用绘制飞机	7	实验六 以行为单位读写硬盘扇区	16
3.1 实验目的	7	6.1 实验目的	16
3.2 实验内容	7	6.2 实验内容	16
3.3 参考资料	7	6.3 参考资料	16
3.4 实验环境	7	6.4 实验环境	16
3.5 实验原理	8	6.5 实验原理	16
3.6 实验步骤	8	6.6 实验步骤	17
3.7 实验结果	9		

6.7 实验结果	17	10.5 实验原理	32
实验七 实时监控硬盘扇区的使用信息	19	10.6 实验步骤	32
7.1 实验目的	19	10.7 实验结果	33
7.2 实验内容	19	实验十一 大小写字母切换	35
7.3 参考资料	19	11.1 实验目的	35
7.4 实验环境	19	11.2 实验内容	35
7.5 实验原理	20	11.3 参考资料	35
7.6 实验步骤	20	11.4 实验环境	35
7.7 实验结果	21	11.5 实验原理	36
实验八 实时监控缓冲块的使用信息	23	11.6 实验步骤	36
8.1 实验目的	23	11.7 实验结果	36
8.2 实验内容	23	实验十二 实现显示器分屏	38
8.3 参考资料	23	12.1 实验目的	38
8.4 实验环境	24	12.2 实验内容	38
8.5 实验原理	24	12.3 参考资料	38
8.6 实验步骤	24	12.4 实验环境	38
8.7 实验结果	25	12.5 实验原理	39
实验九 在显示屏上显示彩色图片	28	12.6 实验步骤	39
9.1 实验目的	28	12.7 实验结果	41
9.2 实验内容	28	实验十三 配置定时器实现飞机变速移	42
9.3 参考资料	28	动	
9.4 实验环境	28	13.1 实验目的	42
9.5 实验原理	28	13.2 实验内容	42
9.6 实验步骤	29	13.3 参考资料	42
9.7 实验结果	29	13.4 实验环境	42
实验十 在进程描述符中添加字段	31	13.5 实验原理	43
10.1 实验目的	31	13.6 实验步骤	43
10.2 实验内容	31	13.7 实验结果	43
10.3 参考资料	31	实验十四 支持三级页表结构	45
10.4 实验环境	31	14.1 实验目的	45

14.2 实验内容	45	实验十八 多扇区文件读	57
14.3 参考资料	45	18.1 实验目的	57
14.4 实验环境	45	18.2 实验内容	57
14.5 实验原理	46	18.3 参考资料	57
14.6 实验步骤	46	18.4 实验环境	57
14.7 实验结果	47	18.5 实验原理	57
		18.6 实验步骤	58
		18.7 实验结果	58
实验十五 基于剩余时间片的进程切换策略	48		
15.1 实验目的	48	实验十九 多扇区文件写	60
15.2 实验内容	48	19.1 实验目的	60
15.3 参考资料	48	19.2 实验内容	60
15.4 实验环境	48	19.3 参考资料	60
15.5 实验原理	49	19.4 实验环境	60
15.6 实验步骤	49	19.5 实验原理	60
15.7 实验结果	49	19.6 实验步骤	61
		19.7 实验结果	62
实验十六 从文件中读指定长度的数据	51		
16.1 实验目的	51	实验二十 增加文件读写属性	63
16.2 实验内容	51	20.1 实验目的	63
16.3 参考资料	51	20.2 实验内容	63
16.4 实验环境	51	20.3 参考资料	63
16.5 实验原理	52	20.4 实验环境	63
16.6 实验步骤	52	20.5 实验原理	63
16.7 实验结果	53	20.6 实验步骤	64
		20.7 实验结果	65
实验十七 向文件中写指定长度的数据	54		
17.1 实验目的	54	实验二十一 实现返回文件大小的系统调用	66
17.2 实验内容	54	21.1 实验目的	66
17.3 参考资料	54	21.2 实验内容	66
17.4 实验环境	54	21.3 参考资料	66
17.5 实验原理	54	21.4 实验环境	66
17.6 实验步骤	55	21.5 实验原理	67
17.7 实验结果	55		

21.6 实验步骤	67	25.1 实验目的	77
21.7 实验结果	67	25.2 实验内容	77
实验二十二 实现返回文件属性的系统调用	69	25.3 参考资料	77
22.1 实验目的	69	25.4 实验环境	77
22.2 实验内容	69	25.5 实验原理	78
22.3 参考资料	69	25.6 实验步骤	78
22.4 实验环境	69	25.7 实验结果	78
22.5 实验原理	70	实验二十六 从硬盘镜像文件中读取指定文件	80
22.6 实验步骤	70	26.1 实验目的	80
22.7 实验结果	70	26.2 实验内容	80
实验二十三 实现 ls 应用程序	72	26.3 参考资料	80
23.1 实验目的	72	26.4 实验环境	80
23.2 实验内容	72	26.5 实验原理	81
23.3 参考资料	72	26.6 实验步骤	81
23.4 实验环境	72	26.7 实验结果	82
23.5 实验原理	72	实验二十七 自定义字模并显示	83
23.6 实验步骤	73	27.1 实验目的	83
23.7 实验结果	73	27.2 实验内容	83
实验二十四 实现可以打印文件详细信息的 ls 应用程序	74	27.3 参考资料	83
24.1 实验目的	74	27.4 实验环境	83
24.2 实验内容	74	27.5 实验原理	83
24.3 参考资料	74	27.6 实验步骤	84
24.4 实验环境	74	27.7 实验结果	84
24.5 实验原理	75	实验二十八 显示一架红色飞机	85
24.6 实验步骤	75	28.1 实验目的	85
24.7 实验结果	75	28.2 实验内容	85
实验二十五 实现打印当前进程打开文件的系统调用	77	28.3 参考资料	85
		28.4 实验环境	85
		28.5 实验原理	86
		28.6 实验步骤	86

28.7 实验结果	86	32.4 实验环境	96
实验二十九 实现光标闪烁的效果	87	32.5 实验原理	97
29.1 实验目的	87	32.6 实验步骤	97
29.2 实验内容	87	32.7 实验结果	97
29.3 参考资料	87	实验三十三 实现遍历页表的函数	99
29.4 实验环境	87	33.1 实验目的	99
29.5 实验原理	87	33.2 实验内容	99
29.6 实验步骤	88	33.3 参考资料	99
29.7 实验结果	88	33.4 实验环境	99
实验三十 支持光标在显示器上上下左		33.5 实验原理	100
右移动	90	33.6 实验步骤	100
30.1 实验目的	90	33.7 实验结果	100
30.2 实验内容	90	实验三十四 实现 wait 系统调用	102
30.3 参考资料	90	34.1 实验目的	102
30.4 实验环境	90	34.2 实验内容	102
30.5 实验原理	91	34.3 参考资料	102
30.6 实验步骤	91	34.4 实验环境	102
30.7 实验结果	92	34.5 实验原理	103
实验三十一 支持退格键删除字符	93	34.6 实验步骤	103
31.1 实验目的	93	34.7 实验结果	103
31.2 实验内容	93	实验三十五 实现 kill 系统调用	105
31.3 参考资料	93	35.1 实验目的	105
31.4 实验环境	93	35.2 实验内容	105
31.5 实验原理	94	35.3 参考资料	105
31.6 实验步骤	94	35.4 实验环境	105
31.7 实验结果	94	35.5 实验原理	106
实验三十二 实现对大写字母的识别	96	35.6 实验步骤	106
32.1 实验目的	96	35.7 实验结果	106
32.2 实验内容	96	实验三十六 统计进程运行过程中时钟	
32.3 参考资料	96	中断的次数	108

36.1 实验目的	108	39.7 实验结果	117
36.2 实验内容	108		
36.3 参考资料	108	实验四十 支持任意大小的共享内存	118
36.4 实验环境	108	40.1 实验目的	118
36.5 实验原理	109	40.2 实验内容	118
36.6 实验步骤	109	40.3 参考资料	118
36.7 实验结果	109	40.4 实验环境	118
		40.5 实验原理	119
实验三十七 创建硬盘镜像文件	111	40.6 实验步骤	119
37.1 实验目的	111	40.7 实验结果	120
37.2 实验内容	111		
37.3 参考资料	111	实验四十一 飞机大战之屏幕刷新	121
37.4 实验环境	111	41.1 实验目的	121
37.5 实验原理	111	41.2 实验内容	121
37.6 实验步骤	112	41.3 参考资料	121
37.7 实验结果	112	41.4 实验环境	121
		41.5 实验原理	121
实验三十八 设置字符背景颜色	113	41.6 实验步骤	122
38.1 实验目的	113	41.7 实验结果	122
38.2 实验内容	113		
38.3 参考资料	113	实验四十二 飞机大战之进程间通信	124
38.4 实验环境	113	42.1 实验目的	124
38.5 实验原理	113	42.2 实验内容	124
38.6 实验步骤	114	42.3 参考资料	124
38.7 实验结果	114	42.4 实验环境	124
		42.5 实验原理	124
实验三十九 基于位图的物理页分配算法优化	115	42.6 实验步骤	125
39.1 实验目的	115	42.7 实验结果	125
39.2 实验内容	115		
39.3 参考资料	115	实验四十三 飞机大战之战机篇	127
39.4 实验环境	115	43.1 实验目的	127
39.5 实验原理	116	43.2 实验内容	127
39.6 实验步骤	116	43.3 参考资料	127
		43.4 实验环境	127

43.5 实验原理	127	44.7 实验结果	132
43.6 实验步骤	128		
43.7 实验结果	128	实验四十五 飞机大战之敌机篇	133
实验四十四 飞机大战之子弹篇	130	45.1 实验目的	133
44.1 实验目的	130	45.2 实验内容	133
44.2 实验内容	130	45.3 参考资料	133
44.3 参考资料	130	45.4 实验环境	133
44.4 实验环境	130	45.5 实验原理	133
44.5 实验原理	130	45.6 实验步骤	134
44.6 实验步骤	131	45.7 实验结果	135

实验一 利用定时器控制飞机移动速度

1.1 实验目的

1. 理解和控制定时器循环模式对系统或对象运动速度的影响。
2. 通过设置或取消定时器的循环模式，观察系统定时事件的触发频率对对象运动的影响，加深对定时器的周期性及非周期性工作模式的理解。

1.2 实验内容

首先在屏幕上画出一架飞机，使其在屏幕中间左右移动，并且移动速度越来越慢。

1.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 2 章。
- MaQueOS 代码 code2（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

1.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

1.5 实验原理

code2 的运行结果是在屏幕上每隔 1 秒显示一次字符串“hello,world.”，这是因为 code2 将恒定频率计时器设置为每隔 1 秒产生一次时钟中断，并且在时钟中断中断处

理程序中实现在显示器上显示字符串。

本实验基于 code2，每触发一次时钟中断就擦除上一架飞机，更新坐标后画出新的飞机，飞机的移动速度由触发中断的时间决定，比如一开始以 0.01 秒触发中断的话飞机就移动的很快，因此改变飞机的移动速度可以通过修改计时器数来实现。

1.6 实验步骤

- 修改 code2/kernel/excp/exception.c 中的 timer_interrupt() 函数：
 1. 关闭循环模式，CSR_TCFG_PER 寄存器是控制时钟计数器的，将其值置为 1 即可关闭循环模式。
 2. 硬件的晶振频率是 0x5f5e100，设置定时器初始值为 val1=0xF4240，即每隔 0.01 秒发生一次中断，将 val1 设置为全局变量。
 3. 在每次处理完时钟中断后，val1 自增 0x10000，然后调用 write_csr_64() 函数将 val1 写入寄存器。
- 在 code2/kernel/exep/exception.c 中添加 clock_interrupt() 函数：
 1. 获得飞机中心的坐标，以该中心位置为依据，得到飞机其他位置的坐标，然后依次清除该位置图标。
 2. 飞机中心的 x 坐标加一，注意检查边界环境。
 3. 以飞机中心新坐标为依据，得到飞机其他位置的坐标，然后重新绘制飞机。

1.7 实验结果

实验结果如动画1.1所示，飞机从左到右移动，随着时间增长，速度越来越慢。

图 1.1: 飞机移动越来越慢动画

实验二 通过键盘控制飞机移动

2.1 实验目的

1. 深入理解按键中断的处理过程。
2. 通过屏幕上飞机的绘制与清除，深入理解显示原理。

2.2 实验内容

程序一开始在屏幕中心画出一架飞机，然后用 wasd 键控制飞机的上下左右移动。

2.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 3 章。
- MaQueOS 代码 code3（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

2.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

2.5 实验原理

键盘按下按键后会由中断处理函数处理该事件，本实验做的主要内容就是修改键盘中断处理函数，针对不同的字符做出不同的操作。

2.6 实验步骤

- 定义全局变量：
 1. plane_x: 飞机在显示器中的横坐标。
 2. plane_y: 飞机在显示器中的纵坐标。
- 在 code3/kernel/drv/console.c 文件中增加 draw_plane 函数:
 1. 参数 x 和参数 y 表示飞机中心, 根据飞机中心得出飞机其他位置坐标。
 2. 在每个位置上打印'*'。
- 在 code3/kernel/drv/console.c 文件中增加 erase_plane 函数:
 1. 增加一个 erase_plane(int x,int y) 函数, 参数 x 和参数 y 表示飞机中心, 根据飞机中心得出飞机其他位置坐标。
 2. 以飞机坐标为参数调用 erase_char 函数, 擦除每个位置的字符。
- 修改 code3/kernel/drv/console.c 文件中的 do_keyboard 函数:
 1. 将传入的字符分别与字符'd'、'a'、'w'、's' 做比较。'd' 表示向右移动, 'a' 表示向左移动, 'w' 表示向上移动, 's' 表示向下移动。
 2. 飞机坐标更改完后, 注意更改完的坐标是否越过了屏幕上下左右边界, 如果没有越过边界的话则先调用 erase_plane 函数对上一个飞机进行擦除, 然后坐标更新调用 draw_plane 函数画下一个飞机。

2.7 实验结果

程序运行效果如动画2.1所示。先按'd' 键将位于中心的飞机往右移动, 再按'a' 键将飞机往左移动, 最后按'w' 键将飞机往上移动。

图 2.1: 按键移动飞机动画

实验三 利用系统调用绘制飞机

3.1 实验目的

1. 理解多进程工作的原理，掌握简单汇编代码的编写。
2. 理解系统调用的工作流程，掌握添加系统调用的方法。

3.2 实验内容

程序运行三个进程，进程 0 创建进程 1，进程 1 创建进程 2，进程 2 在用户态下调用 `input` 系统调用获取键盘输入字符，然后调用 `output` 系统调用将字符显示到屏幕，如果输入的字符是 `p` 则在屏幕上画出一架飞机。

3.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 6 章。
- MaQueOS 代码 code6（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

3.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

3.5 实验原理

code6 中进程 2 负责在用户态下调用 input 系统调用获取键盘输入字符，然后调用 output 系统调用将字符显示到屏幕上，本实验基于 code6，在进程 2 获取字符后，比较一下获取字符是否是字符'p'，如果是则调用 draw_plane 系统调用，在屏幕画出一架飞机；如果不是字符'p' 则调用 output 系统调用，将该字符显示到屏幕上。具体而言主要内容主要有两个方面，首先是修改进程 0、进程 1 和进程 2 运行的二进制可执行代码对应的汇编程序 proc0.s，其次是添加系统调用函数。

3.6 实验步骤

- 修改 code6/kernel/proc/proc0/proc0.s 文件
 1. 在 proc0.s 头部声明 NR_drew_plane 的系统调用号为 5。
 2. 将 NR_output 系统调用拿到的字符与 112(字符'p' 的系统调用) 比较，如果相等则跳转至 drew_plane 处。
 3. drew_plane 标记处正式调用 NRNR_drewPlane 系统调用。
- 实现系统调用 NR_drew_plane
 1. 在 code6/kernel/excp/exception.c 中的系统调用函数指针数组 syscalls 中添加 sys_drewPlane 。
 2. 在 code6/kernel/include/xtos.h 中添加 sys_drewPlane 函数的声明。
 3. 在 code6/kernel/drv/console.c 中完成对 sys_drewPlane() 函数的实现。sys_drewPlane 函数直接使用 printk 函数完成对飞机的绘制即可。
- 修改 code6/kernel/proc/process.c 文件中 proc0_code 数据
 1. 运行 code6/kernel/proc/proc0/下的 get_proc_code.sh 脚本，该脚本编译 proc0.s 文件，生成一个纯二进制文件，然后将其二进制文件转成十六进制输出到控制台。
 2. 复制控制台数据替换 code6/kernel/proc/process.c 中的 proc0_code 数据。

3.7 实验结果

如图3.1所示，输入字符'p'，通过系统调用在屏幕上绘制飞机，如果是其他字符则直接在屏幕上输出。



图 3.1: 系统调用绘制飞机

实验四 根据用户输入显示不同颜色的飞机

4.1 实验目的

1. 熟练掌握添加系统调用的方法。
2. 深入了解 `write_char()` 函数的实现。

4.2 实验内容

程序首先在屏幕上输出提示语句，提醒用户输入飞机的颜色。字符 `r` 表示红色，字符 `b` 表示蓝色，字符 `g` 表示绿色，字符 `y` 表示黄色。用户输入这几个字符中的其中之一就会在屏幕上画出对应颜色的飞机，如果是其他字符则不做任何处理。

4.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 6 章。
- MaQueOS 代码 code6（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

4.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

4.5 实验原理

lab6 中的进程 2 在拿到一个字符后, 检查该字符是否是 p, 是的话调用 drew_plane 系统调用。而在本实验中拿到一个字符后, 需要检查该字符是否是”r,g,b,y”中的一个, 是的话则调用相对应的系统调用函数。因此还需添加四个系统调用, 分别画出红色、蓝色、绿色、黄色的飞机。在四个系统调用函数中设置对应颜色的 rgb 值就可以完成对颜色的控制。

4.6 实验步骤

- 修改 lab6/kernel/proc/proc0/proc0.s 文件
 1. 在比较完 p 字符后, 继续比较是否是”r,g,b,y”, 然后跳转到相对于的段执行具体的系统调用。
 2. 在 proc0.s 头部声明 NR_drewGplane,NR_drewBplane,NR_drewYplane 这三个系统调用号分别为 6, 7, 8。
- 修改 lab6/kernel/excp/exception.c
 1. 在系统调用函数指针数组 syscalls 中添加 sys_drewGplane。
 2. 在系统调用函数指针数组 syscalls 中添加 sys_drewBplane。
 3. 在系统调用函数指针数组 syscalls 中添加 sys_drewYplane 。
- 修改 lab6/kernel/include/xtos.h
 1. 添加 sys_drewGplane 函数的声明。
 2. 添加 sys_drewBplane 函数的声明。
 3. 添加 sys_drewYplane 函数的声明。
- 修改 lab6/kernel/drv/console.c
 1. 添加一个全局变量 color 数组, 数组长度为三, 存放 RGB 值。
 2. 修改 write_char 函数, 在设置字符的 rgb 值时通过 color 数组进行赋值。
 3. 修改 sys_drewPlane 函数, 先设置全局变量 color 数组, 使其三个元素值分别为 0, 0, 255。然后用 printk 函数打印飞机。

4. 添加 `sys_drewGplane` 函数，先设置全局变量 `color` 数组，使其三个元素值分别为 0, 255, 0。然后用 `printk` 函数打印飞机。
 5. 添加 `sys_drewBplane` 函数，先设置全局变量 `color` 数组，使其三个元素值分别为 255, 0, 0。然后用 `printk` 函数打印飞机。
 6. 添加 `sys_drewYplane` 函数，先设置全局变量 `color` 数组，使其三个元素值分别为 0, 255, 255。然后用 `printk` 函数打印飞机。
- 修改 `lab6/kernel/proc/process.c` 中的 `proc0_code` 数据。
 1. 运行 `lab6/kernel/proc/proc0/` 下的 `get_proc_code.sh` 脚本，该脚本编译 `proc0.s` 文件，生成一个纯二进制文件，然后将其二进制文件转成十六进制输出到控制台。
 2. 复制控制台数据替换 `lab6/kernel/proc/process.c` 中的 `proc0_code` 数据。

4.7 实验结果

如图4.1所示，依次输入 y,b,g,g,r，在屏幕上分别画出黄色、蓝色、绿色、绿色和红色的飞机。

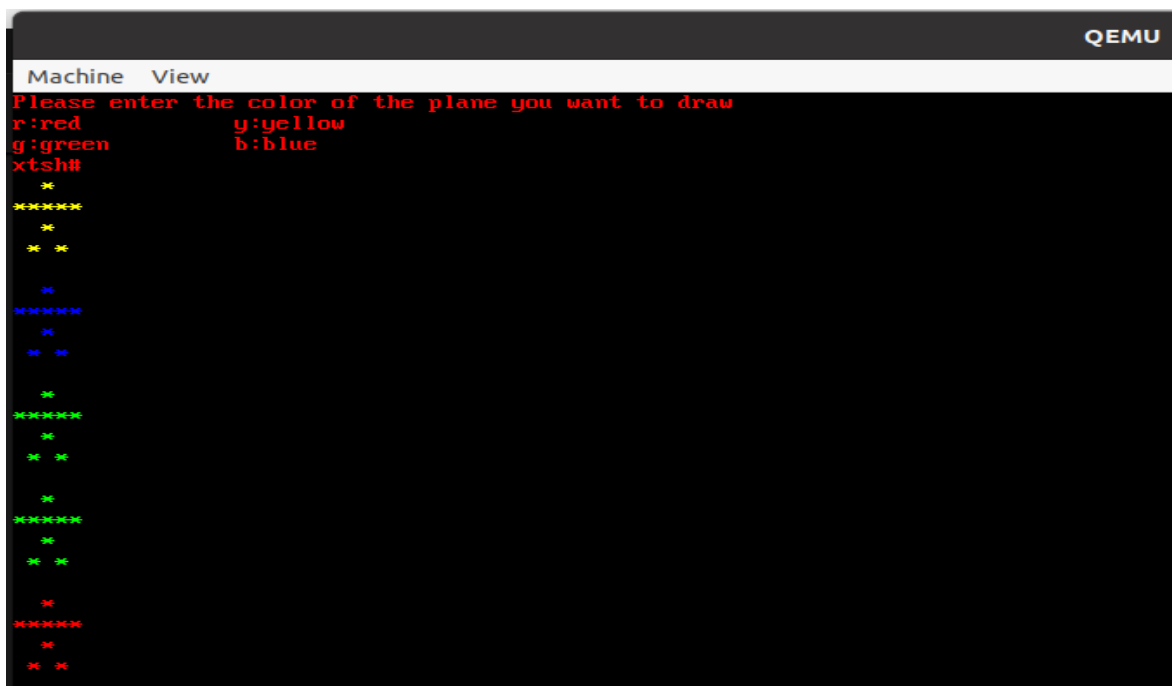


图 4.1: 不同颜色飞机图

实验五 顺序读写硬盘扇区

5.1 实验目的

1. 加深对硬盘驱动的理解。
2. 深入了解写硬盘的详细过程。

5.2 实验内容

本实验一共有两个进程，进程 0 创建进程 1，进程 1 通过 `read_disk` 系统调用从硬盘的 0 号扇区读取字符串“xtsh#”后显示到显示器上，然后通过 `write_disk` 系统调用把从键盘输入的字符依次写到硬盘的 1,2,3... 扇区，然后从磁盘中读出该字符，并显示到显示器。

5.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 7 章。
- MaQueOS 代码 code7（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

5.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

5.5 实验原理

code7 中进程 1 首先从硬盘 0 号扇区中读出"xtsh#" 字符串并将其显示到显示器,接着将从键盘获取的字符写到 0 号扇区,再读 0 号扇区,将读出的字符显示到显示器。本实验基于 code7,和 code7 不同的是需要把从键盘获取的字符按顺序写到硬盘的后续空闲扇区(0 号扇区之后),写完扇区后接着读该扇区将读到的内容显示到显示器。

5.6 实验步骤

- 定义全局变量
 1. readNum: 读硬盘的扇区号。
 2. writeNum: 写硬盘的扇区号。
- 修改 code7/kernel/drv/disk.c 文件
 1. readNum 初始值为 0,表示要读 0 号扇区,
 2. WriteNum 初始为 1,表示要写 1 号扇区。
 3. 修改 sys_read_disk 函数,在调用 write_block 函数时,第一个参数设为 readNum 。
 4. 修改 sys_read_disk 函数,函数里面每次读完数据,readNum 更新。
 5. 修改 sys_write_disk 函数,在调用 write_block 函数时,第一个参数设为 writeNum。
 6. 修改 sys_write_disk 函数,在函数里面每次写完数据后,writeNum 更新。

5.7 实验结果

如图5.1所示,程序首先从硬盘 0 号扇区读出"xtsh#" 字符串并显示到显示器,然后将键入的字符先写入空闲硬盘再读取硬盘,读完后将字符显示到显示器。

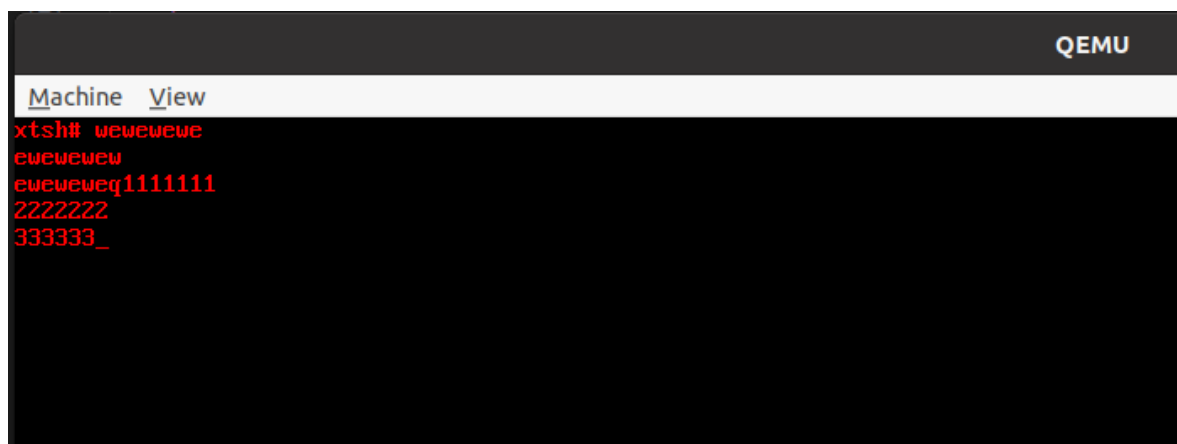


图 5.1: 写硬盘和读硬盘协同截图

实验六 以行为单位读写硬盘扇区

6.1 实验目的

1. 加深对硬盘驱动的理解。
2. 深入了解读写硬盘的详细过程。

6.2 实验内容

程序首先读出数据块 0 的数据显示到屏幕上，然后将键盘输入的字符以换行符为单位依次写进数据块，写进一个数据块后马上读出一个数据块显示到屏幕上。

6.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 7 章。
- MaQueOS 代码 code7（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

6.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

6.5 实验原理

本实验有两个进程，进程 1 首先从硬盘 0 号扇区中读出”xtsh#”字符串并将其显示到显示器，然后将键盘输入的字符以换行符为单位将多个字符依次写入到磁盘的空

闲数据块中，之后再按顺序读取硬盘，将读取的字符串输出到显示器。

6.6 实验步骤

- 定义全局变量。
 1. readNum: 读硬盘的扇区号。
 2. writeNum: 写硬盘的扇区号。
- 修改 code7/kernel/drv/disk.c 文件
 1. readNum 初始值为 0，表示要读 0 号扇区，
 2. WriteNum 初始为 1，表示要写 1 号扇区。
 3. 修改 sys_read_disk 函数，在调用 write_block 函数时，第一个参数设为 readNum。
 4. 修改 sys_read_disk 函数，函数里面每次读完数据，readNum 更新。
 5. 修改 sys_write_disk 函数，在调用 write_block 函数时，第一个参数设为 writeNum。
 6. 修改 sys_write_disk 函数，在函数里面每次写完数据后，writeNum 更新。
- 修改 lab8/kernel/drv/console.c 文件
 1. 修改 put_queue 函数，在 lab8 中该函数将字符 c 放进队列后，马上唤醒阻塞的读队列进程。在该实验中唤醒队列语句之前要检查一下字符 'c' 是否是换行符，如果是换行符则执行唤醒语句。
 2. 修改 sys_input 函数，在 lab8 中该函数只是从队列中读取一个字符。在该实验中要把队列中所有字符都出队。出队完后更新队列尾指针的值。

6.7 实验结果

程序首先从硬盘 0 号扇区读出 "xtsh#" 字符串并显示到显示器。之后用户输入 123，显示器上并不会立即输出 123，直到用户输入回车才会在屏幕上显示出 123，显示效果如图6.1所示。

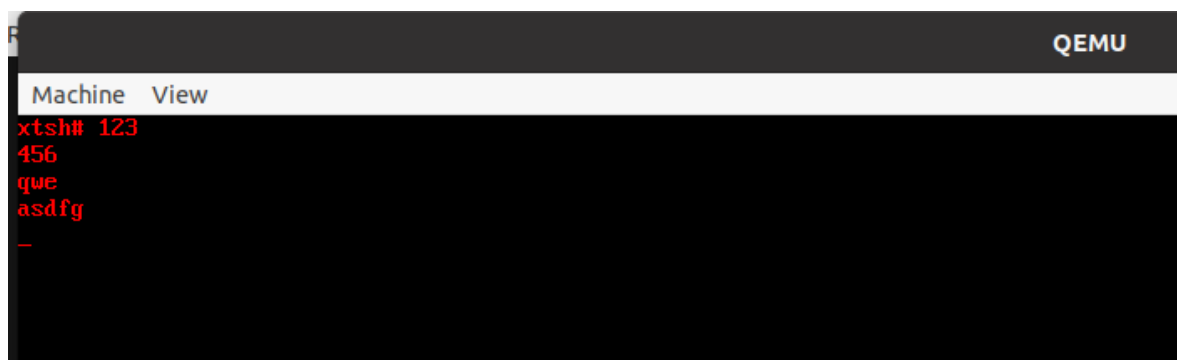


图 6.1: 以换行符为单位读写硬盘扇区实验结果截图

实验七 实时监控硬盘扇区的使用信息

7.1 实验目的

1. 掌握数据分上下两部分打印的原理。
2. 理解读写操作中缓冲块与数据块之间的联系。

7.2 实验内容

本实验基于 code7,code7 中进程 1 首先从硬盘 0 号扇区中读出"xtsh#"字符串并将其显示到显示器，接着将从键盘获取的字符写到 0 号扇区，再读 0 号扇区，将读出的字符显示到显示器。可见 code7 中进程一直在对数据块 0 进行读写操作，本实验在此基础上实时监控硬盘数据块 0 的使用信息，将其实时显示在屏幕下半部分。数据块 0 的使用信息包括 bufferseq(数据块 0 使用的缓冲号)、type(读写操作，0 表示读操作，1 表示写操作)、blocknum(数据块号)、used(使用数据块的进程号)。

7.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 7 章。
- MaQueOS 代码 code7（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

7.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

7.5 实验原理

实时监控数据块 0 的使用信息，即要求每次对数据块 0 的读写操作都会在屏幕上更新。read_block 和 write_block 函数是读写磁盘块的上层函数，只需在该函数内调用打印数据块信息的函数就可以完成实时的要求。

7.6 实验步骤

- 修改 code7/kernel/drv/disk.c 文件
 1. 修改 buffer 数据结构，增加三个成员，seq(buffer 块号)，type(读写类型)，used(使用进程号)。
 2. 用 extern 语句引入 process.c 中的 current 变量。current 表示当前正在执行的进程。
 3. 修改 find_buffer 函数，从 buffer_table 中找到可用缓冲块后，将该缓冲块的 seq 成员进行赋值。
 4. 修改 get_buffer，在返回合适的 buffer 块之前，对 buffer 的 seq 成员进行赋值。
 5. 修改 write_block 函数，将 buffer 的 type 字段设置为 1，表示写操作，然后将 buffer 的 used 字段设置为当前进程 id，然后调用 write_detail 函数进行详细数据的打印。
 6. 修改 read_block 函数，将 buffer 的 type 字段设置为 0，表示读操作。其余修改跟 write_block 函数一样。
- 修改 code7/kernel/drv/console.c 文件
 1. 实现 write_detail 函数，如果是第一次调用该函数则函数首先调用 print_block_title 函数，完成对表头的打印。接下来就是在固定的位置打印各个值。
 2. 实现 print_block_title 函数，要在屏幕的下半部分打印，首先设置 dx,dy 坐标，屏幕的下半部分起始位置设置为屏幕整体宽度的一半，该行打印表头。数据块 0 的使用信息从 dy 行的下两行开始打印。

7.7 实验结果

如图7.1所示是程序运行后的初始情况，code7 中进程 1 从硬盘 0 号扇区中读出”xtsh#” 字符串，并显示到控制台，在屏幕的下半部分打印出了一条记录，type 为 0，used 为 1，bufferseq 为 0，blocknum 为 0，表示进程 1 从数据块 0 中读出数据，使用的缓冲块是 0 号 buffer 块。

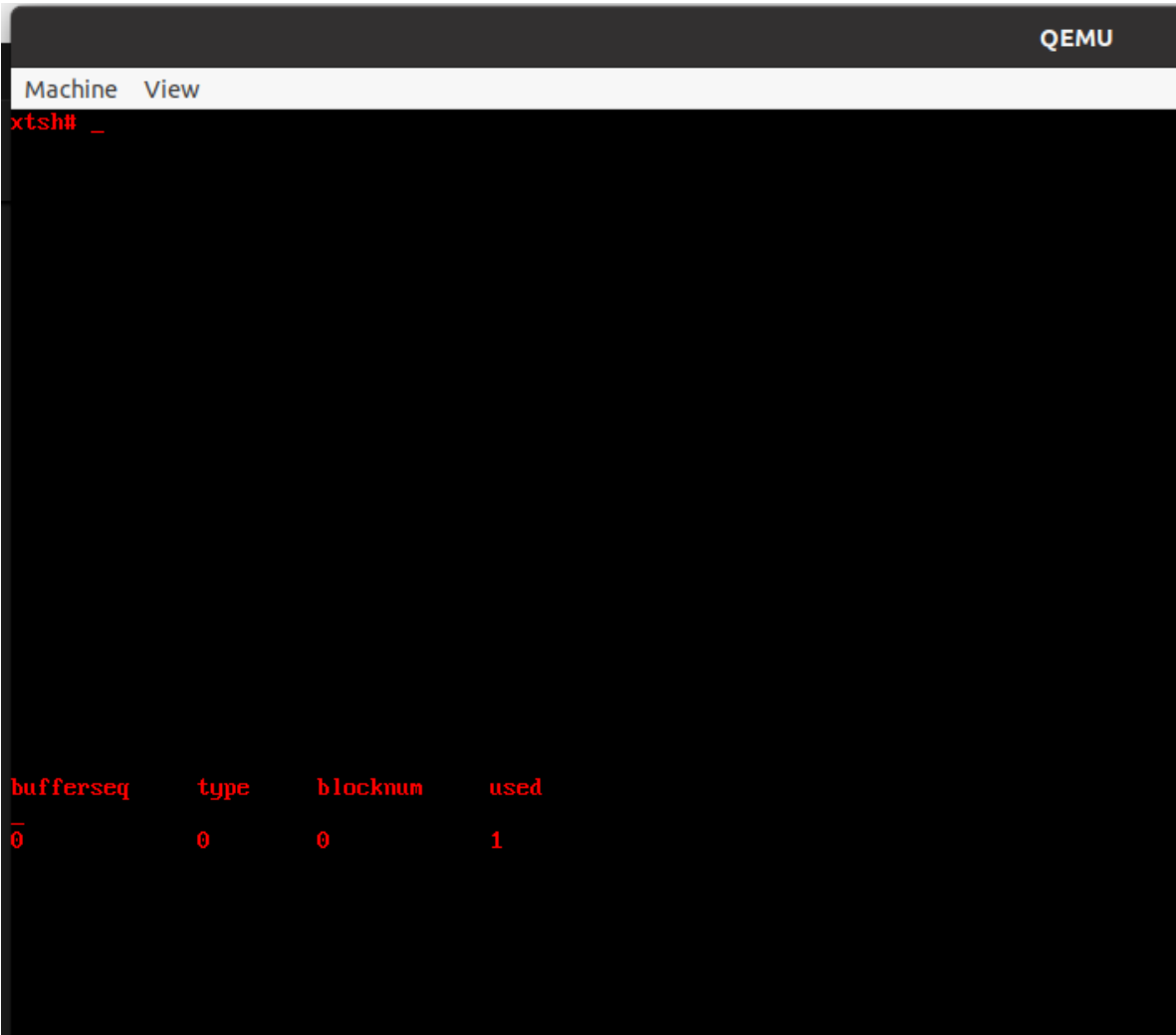
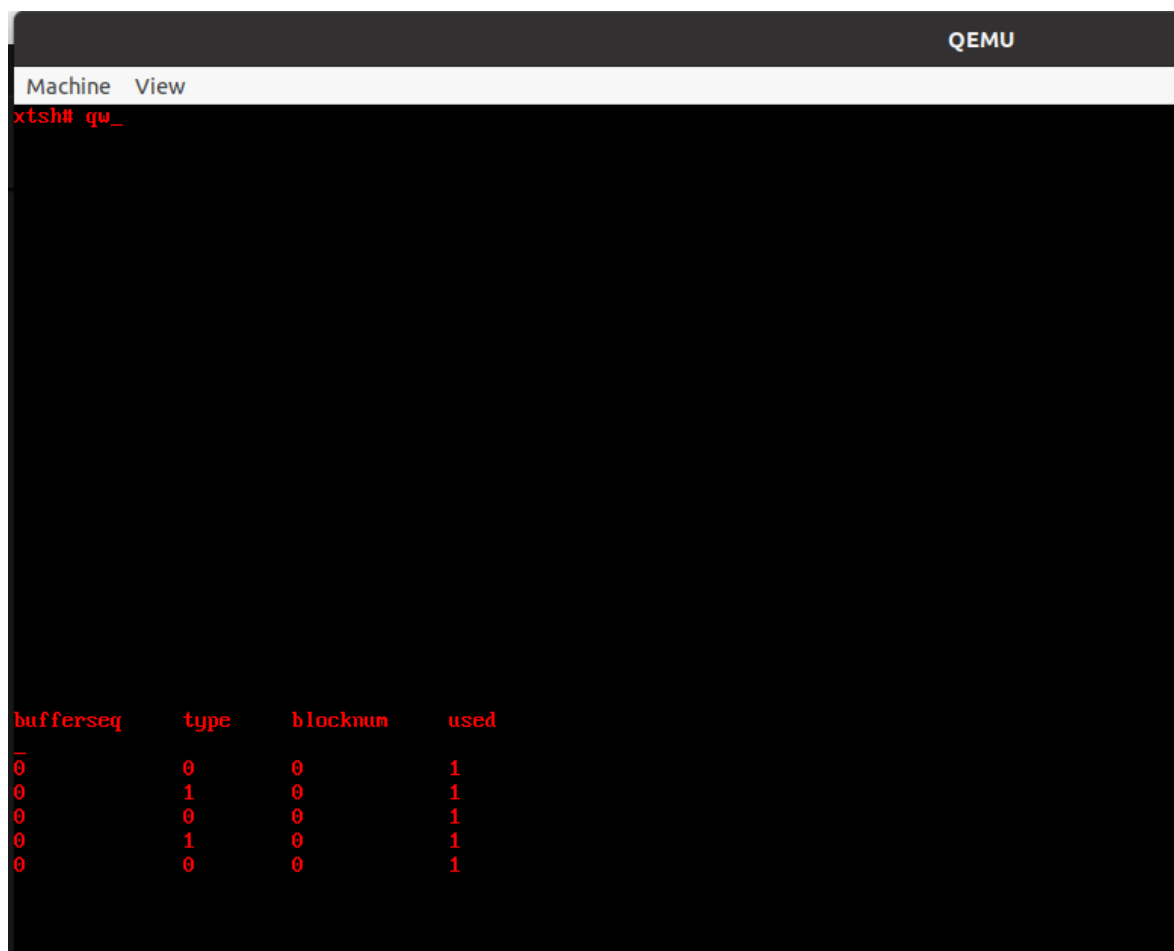


图 7.1: 初始情况截图

然后键盘输入 q 和 w，可以看到在屏幕下方连续又输出 4 条记录，如图7.2所示，因为每个字符有两条记录，分别是将字符写入到 0 号数据块和从 0 号数据块中读出字符。由此可见，程序完成了对数据块 0 的实时监控。



The image shows a QEMU terminal window with a dark background. At the top, there is a title bar with 'QEMU' on the right. Below it, a menu bar shows 'Machine' and 'View'. The terminal prompt is 'xtsh# qw_'. The output is a table with four columns: 'bufferseq', 'type', 'blocknum', and 'used'. The data rows are as follows:

bufferseq	type	blocknum	used
0	0	0	1
0	1	0	1
0	0	0	1
0	1	0	1
0	0	0	1

图 7.2: 键盘输入 qw 后输出截图

实验八 实时监控缓冲块的使用信息

8.1 实验目的

1. 加深对缓冲块的理解。
2. 理解磁盘读写的详细过程。
3. 掌握下半部分屏幕左右分屏的方法。

8.2 实验内容

该实验的主程序逻辑和 lab8 一样，一共有两个进程，进程 0 创建进程 1，进程 1 首先从磁盘 0 号扇区读出字符串“xtsh#”并将其显示到显示器，接着等待键盘输入，将键盘输入字符依次写入磁盘空闲扇区，再读出扇区内容将其显示到显示器。在此基础上本实验要实时监控每个缓冲块的使用信息，即要在屏幕的下方要打印 16 个缓冲块的实时信息，这些信息包括 bufferseq(缓冲块号)、type(读写操作，0 表示读操作，1 表示写操作)、blocknum(数据块号)、used(本次读写操作是由哪个进程发起的)。需要注意的是由于缓冲块的存在，对磁盘的写入有可能没有进行实时的写入，由缓冲块暂存，在 16 个缓冲块都被用完后再一次性写入磁盘。

8.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 7 章。
- MaQueOS 代码 code7（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

8.4 实验环境

- 操作系统: ubuntu20.04
- 虚拟机: qemu-system-loongarch64

8.5 实验原理

键盘每输入一个字符会触发一个磁盘写操作和磁盘读操作，而磁盘的读写操作都要经过一个缓冲块，所以一个字符的输入要在屏幕上显示两遍缓冲块的使用信息。由于写完磁盘后紧接着就读磁盘，读磁盘导致的显示信息会覆盖写操作的显示信息，因此本实验在屏幕下方分开两个区域，读操作导致的显示信息打印在屏幕下方的左边，写操作导致的显示信息打印在屏幕下方的右边。

8.6 实验步骤

- 修改 lab10，将 lab8 的修改更新在 lab10 代码中，保证字符的读写磁盘都是顺序访问。
- 修改 lab10/kernel/include/xtos.h 文件
 1. 将 lab10/kernel/drv/disk.c 下 buffer 数据结构的声明移到 xtos.h 文件中。
- 修改 lab10/kernel/drv/disk.c 文件
 1. 修改 disk_init 函数。该函数是对缓冲块进行初始化，由于起初并没有读写操作，因此所有缓冲块的 type 都是-1，blocknr(对应的磁盘扇区号)也是-1，used 设置为当前进程 id。
 2. 修改 find_buffer 函数，增加一个 type 参数，找到合适的 buffer 块后，对 buffer 块的各个成员进行赋值。
 3. 修改 get_buffer 函数，增加一个 type 参数，找到合适的 buffer 块后，对 buffer 块的各个成员进行赋值。
 4. 修改 write_block 函数，在调用 find_buffer 和 get_buffer 函数时，type 参数设置为 1，然后调用 write_detail 函数打印缓冲块的使用信息。

5. 修改 `read_block` 函数，在调用 `find_buffer` 和 `get_buffer` 函数时，`type` 参数设置为 0，然后调用 `write_detail` 函数打印缓冲块的使用信息。
- 修改 `lab10/kernel/drv/console.c` 文件
 1. 用 `extern` 语句引入外部变量 `buffer_table` 数组。该数组长度为 16，每个成员都是 `buffer` 块，每次触发打印操作后把 `buffer_table` 里每个成员的信息都打印一遍即可。这也是为什么把 `write_detail` 函数修改为无参函数的原因。
 2. `write_detail` 函数的实现。如果是第一次进入该函数的话需要先调用 `printk_block_title` 函数来打印标题。打印 16 个 `buffer` 块的区域是固定的，更新时只需先擦除固定位置的数据，然后再打印。打印时打印整数时调用 `write_int_to_screen` 函数，因为没有实现 `printf("%d",int)` 函数，具体实现方法见下文。另外 `dx` 控制下半部分屏幕左右分屏的操作，由 `wite` 操作引起的打印显示的区域为下半屏幕右半部分，所以 `dx` 要左右移动。
 3. `write_int_to_screen` 函数的实现。该函数实现了将整数转换为字符串，并按字符顺序显示在屏幕上指定的坐标处。
 4. `printk_block_title` 函数的实现。该实验需要打印四个表头，分别是读操作触发下 `buffer` 块的信息和写操作触发下 `buffer` 块的信息。在打印表头的时候注意坐标设置，打印完后还需重置全局坐标变量 `x` 和 `y`。

8.7 实验结果

如图8.1是程序读取完 0 号数据块取出字符后显示的结果，0 号 `buffer` 块的 `type` 为 0，`blocknum` 为 0，`used` 为 1，说明进程 1 从 0 号磁盘数据块中取出数据放到 0 号 `buffer` 中，其余 15 个 `buffer` 块都处于未使用状态，它们的 `used` 为 0，表示是主进程在管理它们。

之后输入 `e`，进程 1 会先将 `e` 写入 1 号磁盘块中，再读 1 号磁盘；更新后的信息如图8.2所示。写入磁盘导致的更新操作打印在屏幕下方右半部分，1 号缓冲块被进程 1 使用，用于写硬盘的数据块 1。读磁盘导致的更新操作在屏幕下方左半部分，1 号缓冲块被进程 1 使用，用于读硬盘的数据块 1。其他缓冲块也是处于空闲状态。股份热狗热狗热狗隔热隔热隔热个

如果之后再输入其他字符，程序会将字符按顺序写进空闲磁盘块号，写完后读出字符，用户可以在屏幕下方看到数据块号在依次递增。写完 15 个字符后，16 个 `buffer`

块都已使用，这时程序会将暂存在 buffer 块中的字符一次性全部写进磁盘中，然后又初始化各个缓冲块表示未使用。接着再输入字符，对应的磁盘数据块号还是再递增，而对应的 buffer 块又从 0 号开始。

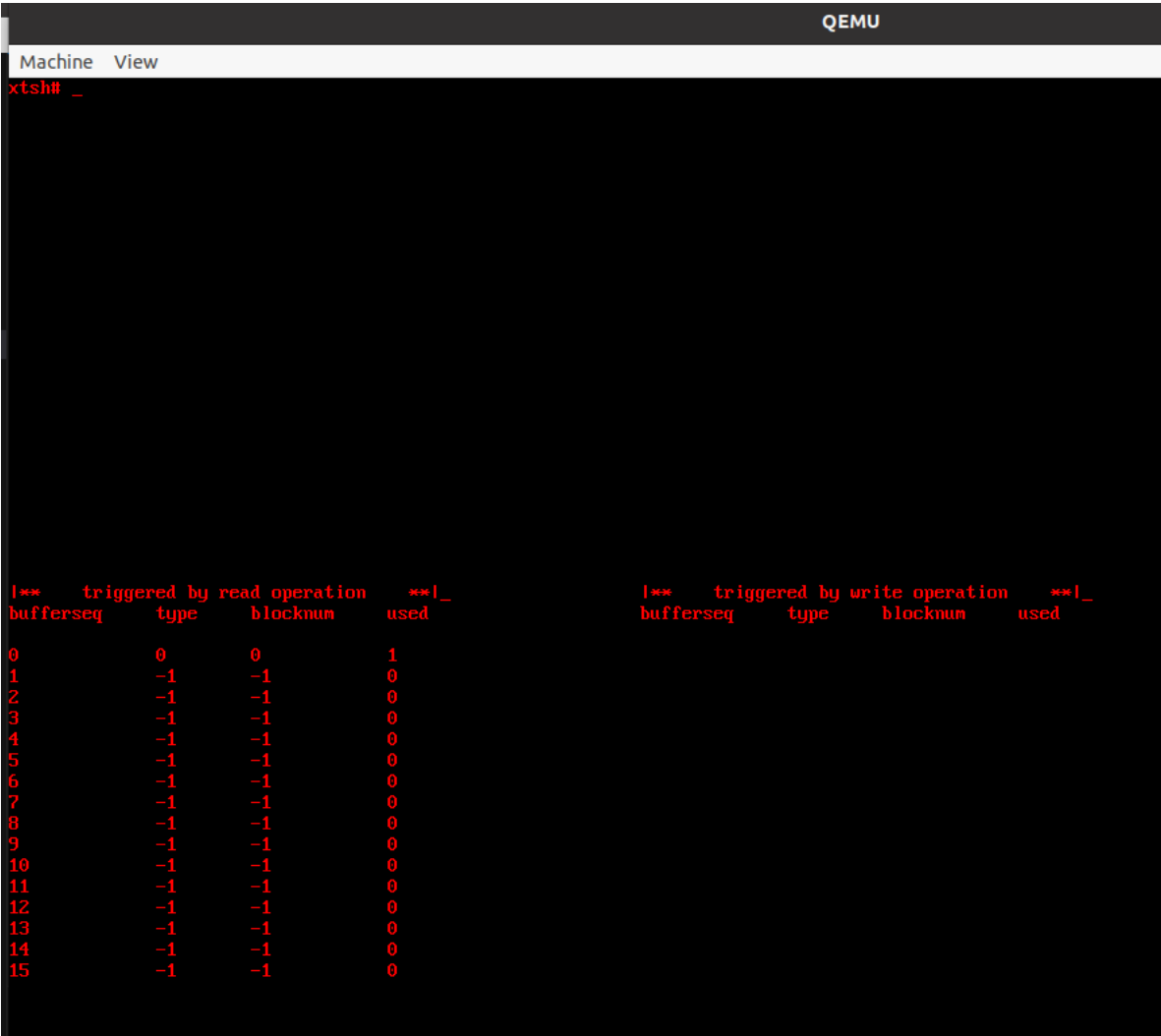


图 8.1: 初始情况截图



The image shows a QEMU terminal window with a dark background and red text. At the top, there is a header bar with 'QEMU' on the right and 'Machine View' on the left. Below the header, the prompt 'xtsh# e_' is visible. The main content of the terminal is two tables of buffer block information, one for read operations and one for write operations. Each table has four columns: 'bufferseq', 'type', 'blocknum', and 'used'. The 'bufferseq' column ranges from 0 to 15. The 'type' column has values 0, -1, or -1. The 'blocknum' column has values 0, 1, or -1. The 'used' column has values 1 or 0.

triggered by read operation				triggered by write operation			
bufferseq	type	blocknum	used	bufferseq	type	blocknum	used
0	0	0	1	0	0	0	1
1	0	1	1	1	1	1	1
2	-1	-1	0	2	-1	-1	0
3	-1	-1	0	3	-1	-1	0
4	-1	-1	0	4	-1	-1	0
5	-1	-1	0	5	-1	-1	0
6	-1	-1	0	6	-1	-1	0
7	-1	-1	0	7	-1	-1	0
8	-1	-1	0	8	-1	-1	0
9	-1	-1	0	9	-1	-1	0
10	-1	-1	0	10	-1	-1	0
11	-1	-1	0	11	-1	-1	0
12	-1	-1	0	12	-1	-1	0
13	-1	-1	0	13	-1	-1	0
14	-1	-1	0	14	-1	-1	0
15	-1	-1	0	15	-1	-1	0

图 8.2: 输入 e 后 buffer 块信息截图

实验九 在显示屏上显示彩色图片

9.1 实验目的

1. 熟练掌握显示彩色图片的方法。
2. 深入了解像素点打印函数的实现。

9.2 实验内容

实验需要提前找好图片，获取该图片的 rgb 文件并存储到数组中，然后在屏幕上一个一个地输出相应的像素点，使得这些像素点组成彩色的图片。

9.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 1 章。
- MaQueOS 代码 code1（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

9.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

9.5 实验原理

在本实验中，要先获取显存的起始地址，即屏幕左上角（0，0）的位置，然后设置两层循环，对于每行每列的每个像素点，分别用提前准备好的 rgb 数组设置当前像

素的红色分量，绿色分量，蓝色分量和透明度，最后将所有像素点都设置完毕形成了彩色的图片。

9.6 实验步骤

- 定义全局变量：
 1. buffer：储存了 3280 个像素点的二维数组。
- 在 code1/kernel/drv/console.c 文件中实现 print_picture 函数：
 1. 用 pos 获取当前的光标位置。
 2. 设置外层为 56 次，内层为 58 次的两层循环。
 3. 在内层循环中将当前光标位置的像素点改为 buffer 数组存储的像素点，需要修改的包括当前像素的红色分量，绿色分量，蓝色分量和透明度。
 4. 内层循环结束后，在外层循环中将当前光标位置移动到下一行。
- 修改 code1/kernel/init/main.c 文件中的 main 函数：
 1. 添加 print_picture 函数的调用。

9.7 实验结果

如图9.1所示，在屏幕上显示出一张彩色图片。



图 9.1: 彩色照片图

实验十 在进程描述符中添加字段

10.1 实验目的

1. 理解进程数据结构的基本组成：通过修改现有的进程数据结构，深入理解操作系统中如何管理进程信息。
2. 掌握进程信息的显示机制：通过向进程数据结构中添加 name 字段，实现对进程详细信息的显示。

10.2 实验内容

本实验的目标是向 process 结构体中添加一个 name 字段，用于存储进程的名称信息，并在系统的显示器下半部分显示进程 0 和进程 1 的详细信息，包括其名称以及其他进程信息。

10.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 5 章。
- MaQueOS 代码 code5（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

10.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

10.5 实验原理

该实验需要在进程结构体的定义中加入一个 name 字段用于设置进程的名字，然后设计一个函数在显示器的下半部分打印内容，再在时钟中断函数中打印进程 0 和进程 1 的详细信息，包括名称及其他进程信息。

其他进程信息中有页目录表地址和进程可执行文件在虚拟地址空间中的结束地址，需要用十六进制表示，所以还需要修改打印十六进制内容的函数。

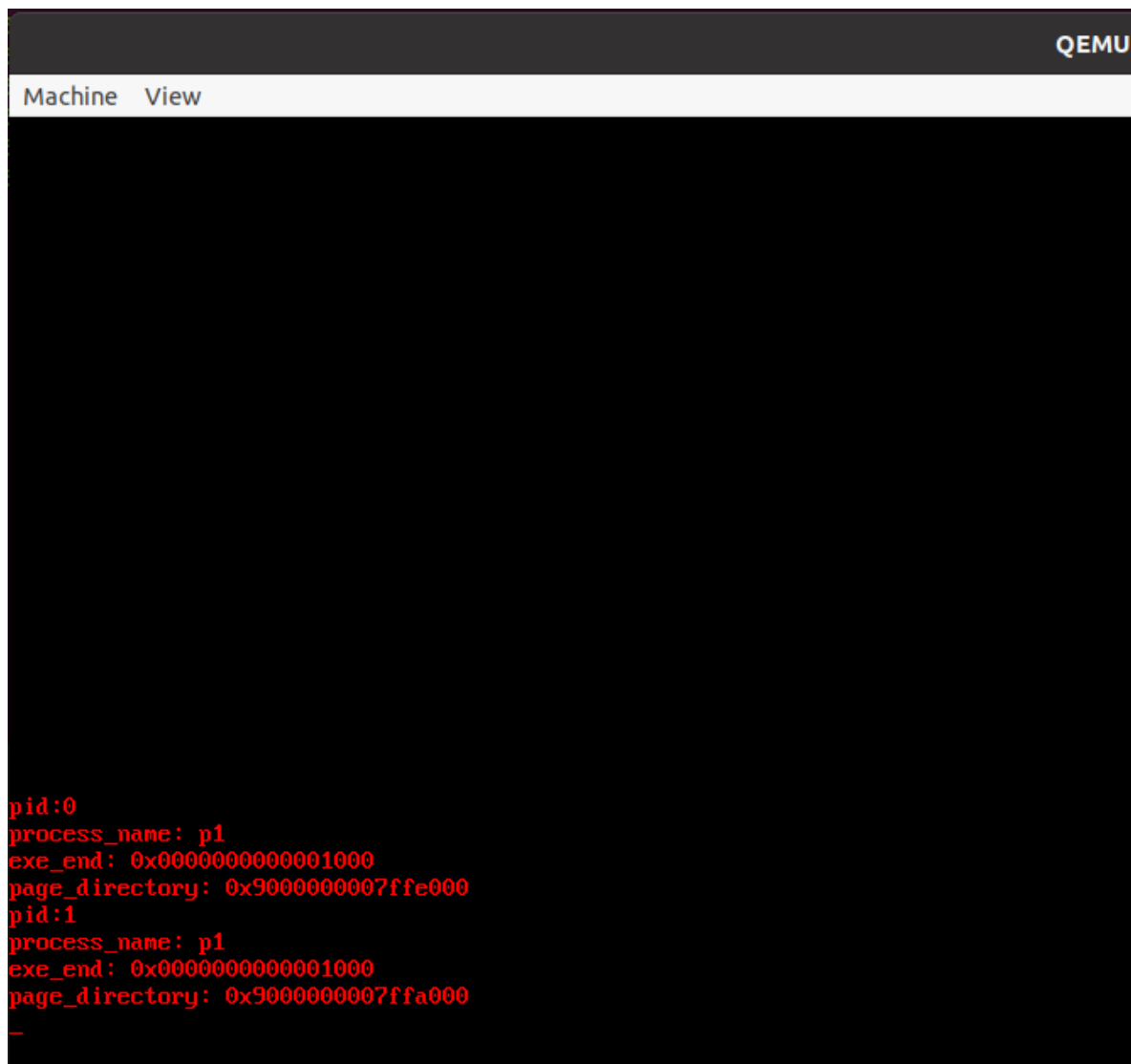
10.6 实验步骤

- 定义全局变量：
 1. x1: 显示器下半部分显示的横坐标。
 2. y1: 显示器下半部分显示的纵坐标。
 3. count: 用于控制时钟中断函数输出内容的次数。
- 在 code5/kernel/drv/console.c 文件中实现 printk_kernel 函数：
 1. 将光标的位置 x, y 变为显示器下半部分的光标 x1, y1。
 2. 调用 printk 函数打印传入的字符串。
 3. 将打印后的光标位置 x, y 赋值给 x1, y1。
- 修改 code5/kernel/drv/console.c 文件中的 printk_debug 函数：
 1. 将所有的 printk 函数调用部分改为 printk_kernel 函数调用。
- 修改 code5/kernel/include/xtos.h 文件中的 process 结构体：
 1. 在结构体中加入一个字符型指针 name 用于储存进程名字。
- 在 code5/kernel/proc/process.c 文件中实现 memcpy 函数：
 1. 将传入的一个字符串部分内容复制到另一个传入的字符串中。
- 修改 code5/kernel/proc/process.c 文件中的 process_init 函数：

1. 增加进程 0 的 name 字段的初始化。
- 修改 code5/kernel/proc/process.c 文件中的 sys_fork 函数：
 1. 增加 fork 创建进程的 name 字段的初始化。
 - 修改 code5/kernel/excp/exception.c 文件中的 do_timer 函数：
 1. 利用 count 判断是否要输出进程信息，仅当 count 为 0 时输出进程信息。
 2. 用字符数组存储好相应的进程显示信息，调用 printk_kernel 函数输出进程 0 的进程号和进程名。
 3. 调用 printk_debug 函数输出进程 0 的页目录表地址和进程可执行文件在虚拟地址空间中的结束地址。
 4. 进程 1 也和进程 0 执行一样的调用输出进程信息。

10.7 实验结果

如图10.1所示，最后在屏幕的下半部分显示出所有进程的详细信息，包括进程名，进程号，页目录表地址和进程可执行文件在虚拟地址空间中的结束地址。



The image shows a QEMU Machine View window. The title bar is dark gray with the QEMU logo in the top right corner. Below the title bar is a light gray bar with the text "Machine View". The main area is black and contains red text showing process information for two processes, pid:0 and pid:1. The text is as follows:

```
pid:0
process_name: p1
exe_end: 0x00000000000001000
page_directory: 0x9000000007ffe000
pid:1
process_name: p1
exe_end: 0x00000000000001000
page_directory: 0x9000000007ffa000
—
```

图 10.1: 所有进程信息图

实验十一 大小写字母切换

11.1 实验目的

1. 通过编写操作系统内核代码，学会如何识别大写字母的输入。
2. 掌握 MaQueOS 的键盘中断处理过程。

11.2 实验内容

在本实验中，需要输入一些包含大写字母和小写字母的字符串，需要实现大小写字母的输入识别，未按下 caps 键时，显示小写字母，当按下 caps 键时，显示大写字母。

11.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 3 章。
- MaQueOS 代码 code3（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

11.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

11.5 实验原理

首先需要在代码中实现对大写字母输入的支持，然后实现 caps 键的功能，按下 caps 键可以打开或关闭大写，同时要对大写字母和小写字母做相应的处理，从而实现大小写字母的输出功能。

11.6 实验步骤

- 定义全局变量：
 1. def: 用于表示 caps 键是否按下。
 2. keys_map_1: 将从键盘读取到的键盘扫描码转换为相应的字符的一维数组。
- 在 code3/kernel/drv/console.c 文件中实现 memcpy 函数：
 1. 将传入的一个字符串部分内容复制到另一个传入的字符串中。
- 修改 code3/kernel/drv/console.c 文件中修改 keyboard_interrupt 函数：
 1. 无论按键是否松开，若键盘扫描码为 0x58，对 def 取反，表示大小写转换。
- 修改 code3/kernel/drv/console.c 文件中修改 do_keyboard 函数：
 1. 若 def 为 1，将键盘扫描码转换为包含大写字母的相应字符，否则将键盘扫描码转换为包含小写字母的相应字符。
 2. 按键是 caps 则根据 def 的值判断当前输入的字符是大写还是小写状态并输出提示，然后将 def 的值取反。

11.7 实验结果

如图11.1所示，实验成功在显示屏上显示了六行输入字符的效果，当按下 caps 键使得大写打开时，提示当前打开了大写，同时输出大写字母，当按下 caps 键关闭大写时，提示当前关闭了大写，同时输出小写字母。



图 11.1: 大小写字输出

实验十二 实现显示器分屏

12.1 实验目的

1. 通过编写操作系统内核代码，学会如何在特定位置进行内容的显示，并将在显示器下半部分显示实现为一个接口。
2. 了解显示器的显示原理，学会如何使用系统调用将缓冲区的内容刷新到显示器上。
3. 了解显示器卷屏等功能的实现，将显示器分为上下两个屏。

12.2 实验内容

在本实验中，需要重做原有显示器的显示功能，使得显示器能够分屏，最终通过两个函数接口来在上下部分显示屏上输出相应的内容，这两个函数的功能分别是在显示器上部分和下部分显示需要打印的东西。

12.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 2 章。
- MaQueOS 代码 code2（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

12.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

12.5 实验原理

本实验需要终于需要实现两个函数接口，这两个函数接口都需要完成一个工作：打印字符串到显示器的指定位置上，从而实现显示器分屏。为了使得这两块屏幕都有原屏幕的功能，需要重新实现并多实现一个卷屏，回车换行，删除字符的函数。同时在坐标初始化时还要多初始化下半部分显示屏的坐标。

12.6 实验步骤

- 定义全局变量：
 1. count1: 输出在“hello, world”的后面，用于区分不同的“hello, world”。
 2. X: 下半屏的横坐标。
 3. Y: 下半屏的纵坐标。
 4. NR_PIX_Y_1: 上半屏的高度。
 5. NR_PIX_Y_2: 总屏幕的高度。
 6. NR_CHAR_Y_1: 上半屏可以显示的字符行数。
 7. NR_CHAR_Y_2: 整个屏幕可以显示的字符行数。
- 在 code2/kernel/drv/console.c 文件中实现 scrup_1 函数：
 1. 将 scrup 函数中的 NR_PIX_Y, NR_CHAR_Y 分别改为 NR_PIX_Y_1, NR_CHAR_Y_1, 负责上半屏幕的卷屏。
- 在 code2/kernel/drv/console.c 文件中实现 scrup_2 函数：
 1. 获取第 25 行的内容放在 to 中，获取第 26 行的内容放在 from 中。
 2. 从第 26 行开始，逐行向前滚动。
 3. 滚动结束后清除最后一行的内容。
 4. 更新 sum_char_x 数组。
 5. 最后实现下半屏幕的卷屏。
- 在 code2/kernel/drv/console.c 文件中实现 cr_lf_1 函数：

1. 将 `cr_lf` 函数中的 `scrup` 函数调用, `NR_CHAR_Y` 分别改为 `scrup_1` 函数调用, `NR_CHAR_Y_1`, 负责上半屏幕的回车换行。
- 在 `code2/kernel/drv/console.c` 文件中实现 `cr_lf_2` 函数:
 1. 将 `cr_lf` 函数中的 `scrup` 函数调用, `NR_CHAR_Y`, `x`, `y` 分别改为 `scrup_2` 函数调用, `NR_CHAR_Y_2`, `X`, `Y`, 负责下半屏幕的回车换行。
 - 在 `code2/kernel/drv/console.c` 文件中实现 `del_2` 函数:
 1. 将 `del` 函数中 `x`, `y` 分别改为 `X`, `Y`, 负责下半屏幕的字符删除。
 - 修改 `code2/kernel/drv/console.c` 文件中的 `printk` 函数:
 1. 将 `del` 函数调用, `cr_lf` 函数调用分别改为 `del_1` 函数, `cr_lf_1` 函数的调用, 实现上半屏幕的字符打印。
 - 在 `code2/kernel/drv/console.c` 文件中实现 `print_kernel` 函数:
 1. 将 `printk` 函数中的 `del` 函数调用, `cr_lf` 函数调用, `x`, `y` 分别改为 `del_2` 函数调用, `cr_lf_2` 函数调用, `X`, `Y`, 实现下半屏幕的字符打印。
 - 在 `code2/kernel/drv/console.c` 文件中实现 `con2_init` 函数:
 1. 初始化 `X`, `Y` 的坐标分别为 0, 25。
 - 修改 `code2/kernel/init/main.c` 文件中的 `main` 函数:
 1. 添加 `con2_init` 函数的调用。
 - 修改 `code2/kernel/excp/exception.c` 文件中的 `timer_interrupt` 函数:
 1. 将 `count1` 加一。
 2. 调用 `printk` 函数在上半屏幕不断输出带数字标志的 “hello, world.”。
 3. 调用 `printk_kernel` 函数在下半屏幕不断输出带数字标志的 “hello world in the low part.”。

12.7 实验结果

实验结果如动画12.1所示，显示屏上下部分分别显示相应字符串，上下屏幕不断显示带数字标识的字符串，屏幕满之后就开始不断卷屏显示字符串。

图 12.1: 上下屏卷屏显示字符串动画

实验十三 配置定时器实现飞机变速移动

13.1 实验目的

1. 了解定时器对于飞机移动和一些定时操作的影响。
2. 学会如何设置定时器，使得飞机的移动速度能够根据定时器的配置而变快或者变慢。

13.2 实验内容

首先在屏幕上画出一架飞机，使其在屏幕往下移动，移动速度随时可以改变，或比正常的移动速度慢，或比正常的移动速度快。

13.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 2 章。
- MaQueOS 代码 code2（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

13.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

13.5 实验原理

第二章 code 通过调用 `read_cpucfg` 库函数，获取定时器所用时钟对应的晶振频率 (`0x5f5e100`)，即定时器每秒自减 `0x5f5e100` 次，通过将该频率值设置为定时器计时自减计数的初始值，达到将定时器中断产生的时间间隔设置为 1 秒的目的，所以若自减计数初始值设计越大，定时器的中断就产生越快，自减计数初始值设计越小，定时器的中断就产生越慢，再在定时器中断函数中擦除原飞机输出新飞机并换行，就可以达到改变飞机移动速度，使得飞机移动变快或变慢的目的。

13.6 实验步骤

- 修改 `code2/kernel/excp/exception.c` 文件中的 `timer_interrupt` 函数：
 1. 调用 `erase_char` 函数擦除显示屏上的内容。
 2. 调用 `printk` 函数输出一架飞机。
- 修改 `code2/kernel/excp/exception.c` 文件中的 `excp_init()` 函数：
 1. 修改 `val` 的值可以实现飞机移动变速的效果。
 2. 此处将 `val` 的值设置为函数调用 `read_cpucfg(CC_FREQ)` 返回值的四倍，使得飞机的移动速度比原来慢四倍。

13.7 实验结果

实验结果如动画13.1所示，显示屏上显示了飞机用原来的两倍速度从上到下移动的结果。

图 13.1: 飞机两倍速移动动画

实验十四 支持三级页表结构

14.1 实验目的

1. 了解三级页表的结构及其在虚拟内存管理中的应用。
2. 熟悉页表的层次结构、页面映射的方式及其实现原理。
3. 深入了解操作系统如何管理虚拟地址到物理地址的转换过程。

14.2 实验内容

将 maqueOS 的二级页表结构升级为三级页表结构，进程地址空间从 1GB 增大到 512GB。

14.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第四章。
- MaQueOS 代码 code4（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

14.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

14.5 实验原理

在操作系统中，虚拟地址需要通过页表映射到物理地址。多级页表是为了解决大内存空间下的内存管理效率问题。三级页表将虚拟地址划分为三个部分，分别由三级页表进行管理，从而减小页表占用的内存和提高效率。

要使 MaQueOS 支持三级页表，需要进行如下操作：初始化 PWC 寄存器，修改 `get_pte` 函数，以及修改 `tlb_handle` 函数。

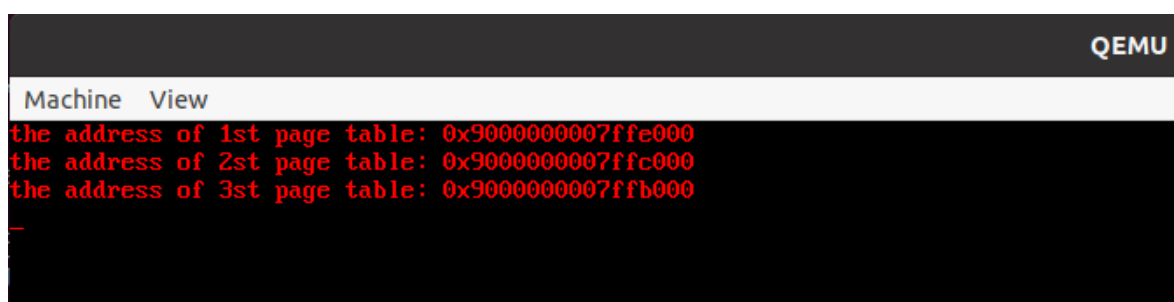
14.6 实验步骤

- 修改 `code4/kernel/excp/exception_handler.S` 中的 `tlb_handler` 函数：
 1. 在 `tlb_handler` 的原代码中，通过 `lddir` 和 `ldpte` 指令加载了第一级页表的奇偶页。
 2. 在 `tlb_handler` 中，使用同样的指令新增第二级页表与第三级页表的加载逻辑。
- 修改 `code4/kernel/mm/memory.c` 中的 `get_pte` 函数：
 1. 增加了新的变量 `pd2`，这个变量用于存储第一级页表的地址
 2. 修改了 `pde` 的计算方式，原来的代码只使用了 `u_vaddr` 的高 21 位来计算页目录项的偏移。修改后的代码首先使用 `u_vaddr` 的高 30 位来计算第一级页表的偏移，然后再加上 `u_vaddr` 的高 21 位（减去前 30 位的偏移）来计算第二级页表项的偏移。
- 修改 `code4/kernel/mm/memory.c` 中的 `mem_init` 函数：
 1. 增加了两个新的参数：`PWCL_PD2WIDTH` 和 `PWCL_PD2BASE`，分别代表第二级页目录的宽度和基地址。
 2. 将包括新增参数的完整配置值写入 `CSR_PWCL` 寄存器。
- 为了在屏幕上显示三级页表修改后的结果，可以访问并打印进程 0 的页表内容，具体步骤如下：
 1. 从进程 0 的控制块中获取其页目录的基址。

2. 计算进程 0 的第一级页表的地址，第二级页表的地址以及第三级页表的地址。
3. 依次打印进程 0 的第一级页表的地址，第二级页表的地址以及第三级页表的地址。

14.7 实验结果

如图14.1所示，修改页表结构后，打印进程 0 的第一级，第二级，第三级页表对应的地址。



```
QEMU
Machine View
the address of 1st page table: 0x9000000007ffe000
the address of 2st page table: 0x9000000007ffc000
the address of 3st page table: 0x9000000007ffb000
-
```

图 14.1: 打印进程 0 的页表地址

实验十五 基于剩余时间片的进程切换策略

15.1 实验目的

1. 深入理解操作系统如何管理和调度进程。
2. 掌握进程调度算法的实现。

15.2 实验内容

实现基于剩余时间片的进程切换策略，剩余时间片越多，进程的优先级越高。

15.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 5 章。
- MaQueOS 代码 code5（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

15.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

15.5 实验原理

在本实验中，增加新的变量来记录所有进程中最大的剩余时间片，在进程调度时，比较当前进程的剩余时间片与最大的剩余时间片，选择剩余时间片最多的进程进行调度。

15.6 实验步骤

修改/code5/kernel/proc/process.c 文件中的 schedule 函数。

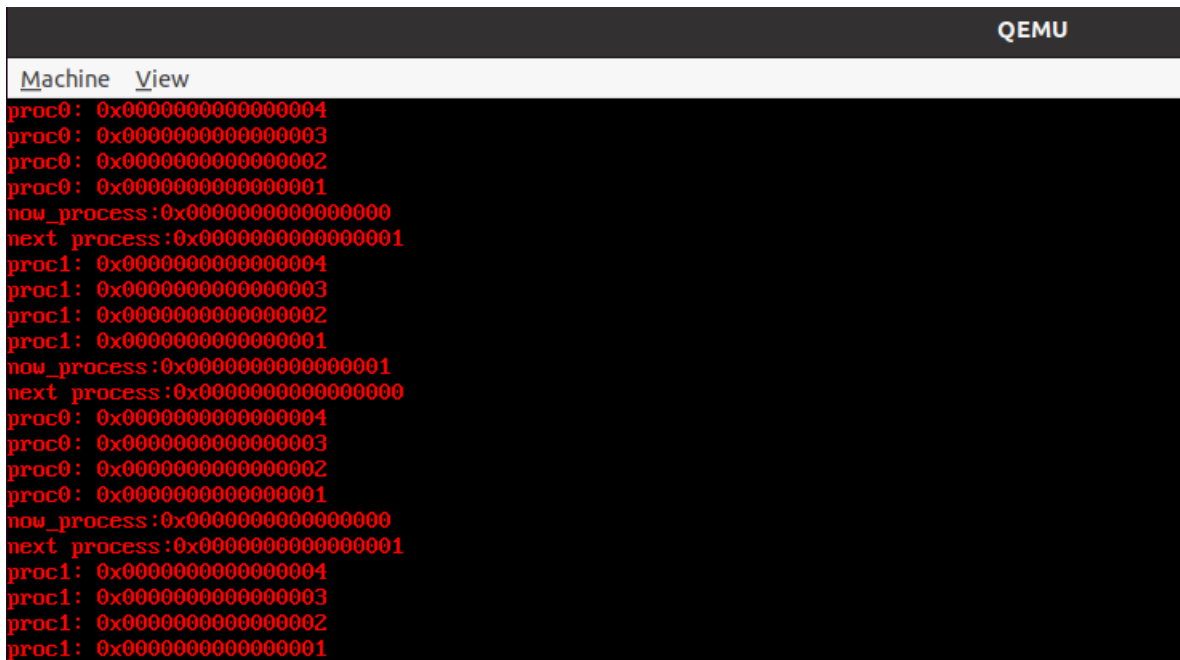
- 在 schedule 函数中添加 max_mem 变量：
 1. max_mem 变量初始值为 0。
 2. max_mem 变量是用来跟踪当前调度过程中已选择进程的剩余时间片的最大值。
- 比较当前进程的剩余时间片与 max_mem 变量的大小：
 1. 如果当前进程的剩余时间片大于 max_mem，则将当前进程的剩余时间片的大小赋值给 max_mem。
 2. 如果当前进程的剩余时间片小于 max_mem，则跳过该进程，继续比较。
 3. 需要注意循环退出的条件 break 是否合适。
- 在进程调度的过程中，使用 print_debug 函数，打印当前进程的 pid 以及下一个需调度进程的 pid。
- 当所有进程的时间片都耗尽后，要给所有的进程重新分配剩余时间片，则要修改 schedule 函数相关的判定条件，并将 max_mem 重新赋初值为 0，重新将当前进程的剩余时间片与 max_mem 变量进行比较。

15.7 实验结果

如图15.1所示，总共有两个进程正在运行，分别是进程 0 与进程 1。进程 0 先给 max_mem 进行赋值，由于两个进程分配的时间片是相同，所以在后续的剩余时间片的比较中，max_mem 不会再次赋值。因此进程 0 先调度，随后调度进程 1。

当两个进程的时间片都耗尽后，系统会重新分配剩余时间片，并将 `max_mem` 重新赋初值为 0，重新将当前进程的剩余时间片与 `max_mem` 变量进行比较。同上，因为进程 0 先给 `max_mem` 进行赋值，因此先调度进程 0，再调度进程 1。

综上，进程的调度顺序是进程 0，进程 1，重新分配剩余时间片后，又重复进程 0，进程 1 的顺序进行调度。



```
QEMU
Machine  View
proc0: 0x0000000000000004
proc0: 0x0000000000000003
proc0: 0x0000000000000002
proc0: 0x0000000000000001
now_process:0x0000000000000000
next_process:0x0000000000000001
proc1: 0x0000000000000004
proc1: 0x0000000000000003
proc1: 0x0000000000000002
proc1: 0x0000000000000001
now_process:0x0000000000000001
next_process:0x0000000000000000
proc0: 0x0000000000000004
proc0: 0x0000000000000003
proc0: 0x0000000000000002
proc0: 0x0000000000000001
now_process:0x0000000000000000
next_process:0x0000000000000001
proc1: 0x0000000000000004
proc1: 0x0000000000000003
proc1: 0x0000000000000002
proc1: 0x0000000000000001
```

图 15.1: 进程调度

实验十六 从文件中读指定长度的数据

16.1 实验目的

1. 熟悉系统调用宏的实现原理。
2. 掌握 MaQueOS 写文件操作过程。

16.2 实验内容

给 `sys_read` 增加一个参数，表示读取的字符数，使得 `sys_read` 可以读取指定数量的字符。具体涉及，修改内核文件读部分以及系统调用部分代码。以及修改原有的可执行文件 `read.S`（修改后的为 `read1.S`），在 `xtsh` 的命令行下运行，进行读写测试。

16.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

16.4 实验环境

- 操作系统：Ubuntu 20.04。
- 虚拟机：qemu-system-loongarch64

16.5 实验原理

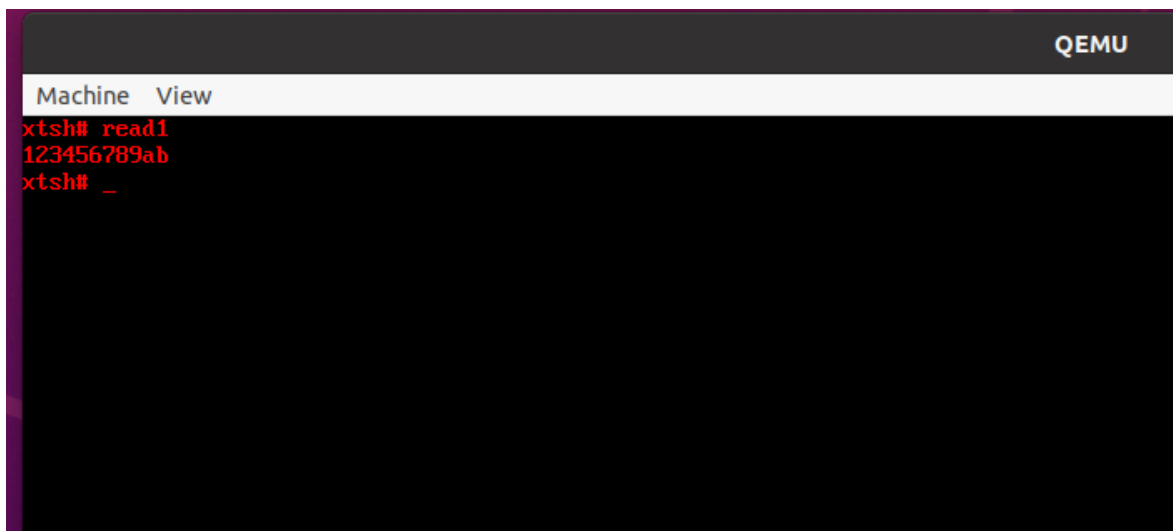
首先，修改 `copy_mem` 函数，使其能够复制指定数量的字节到缓冲区 (buf)。然后，给 `sys_read` 系统调用增加一个参数，表示需要读取的字符数。最后，修改内核中的相关函数（如 `read_inode_block`），使其根据新的参数值来读取指定长度的数据。

16.6 实验步骤

- 修改 `sys_read()` 函数：
 1. 打开并修改 `xtfs.c` 文件，找到 `sys_read()` 函数的定义。
 2. 在 `sys_read()` 函数的参数列表中增加一个名为 `size` 的参数，用于指定读取的字符数。
 3. 根据新增加的参数 `size`，修改函数内部的逻辑，确保只读取指定数量的字符。
- 更新其他相关函数：
 1. 打开并修改 `read_inode_block` 函数，将新的 `size` 参数传递给需要它的地方。
 2. 修改 `copy_mem` 函数，使其能够根据指定的字节数进行复制。
- 添加新的系统调用接口：
 1. 在 `xtfs/bin/asm.h` 文件中，添加一个宏定义，用于新的三个参数的系统调用。
- 测试修改后的系统调用：
 1. 修改原有的可执行文件 `read.S` 为 `read1.S`，并更新其中的系统调用代码以使用新的接口。
 2. 在 `xtsh` 命令行下运行修改后的可执行文件，进行读写测试，验证新的 `sys_read` 系统调用是否正确工作。

16.7 实验结果

如图16.1所示，执行修改后的 read1 应用程序，成功读取了对应字节数的数据。



```
Machine View
xtsh# read1
123456789ab
xtsh# _
```

图 16.1: 执行修改后的 sys_read 系统调用

实验十七 向文件中写指定长度的数据

17.1 实验目的

1. 熟悉系统调用宏的实现原理。
2. 掌握 MaQueOS 写文件操作过程。

17.2 实验内容

修改内核文件写部分以及系统调用部分的代码。修改原有的可执行文件 `write.S`，在 `xtsh` 的命令行下运行，进行写文件测试。

17.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

17.4 实验环境

- 操作系统：Ubuntu 20.04。
- 虚拟机：qemu-system-loongarch64

17.5 实验原理

原内核写入文件是直接写入 512 字节（即一个磁盘块大小）。现修改 `copy_mem` 函数，使其能够复制指定数量的字节到缓冲区。同时，为 `sys_write` 系统调用增加一

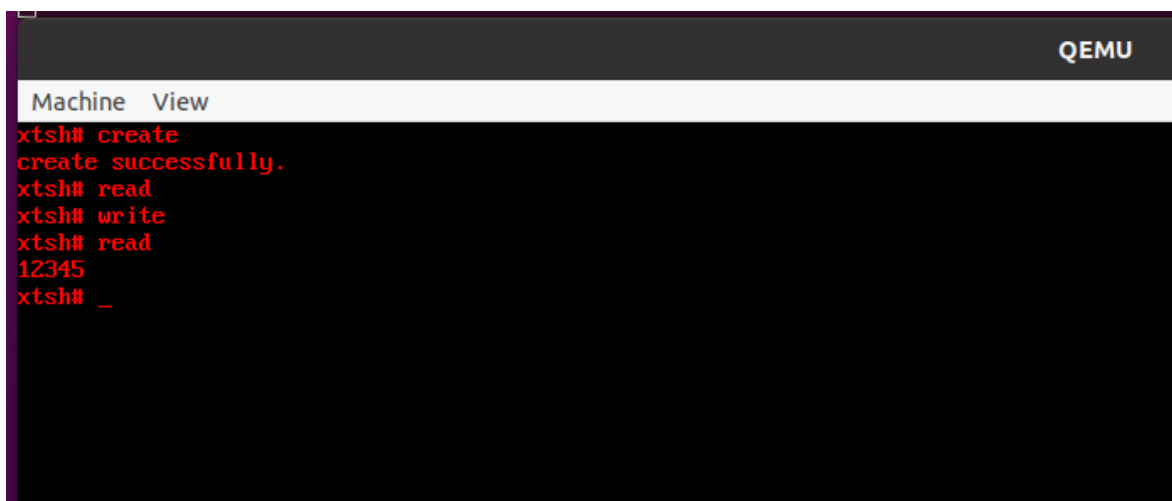
个参数，用于指定写入的字符数，并增加一个三个参数的系统调用接口。

17.6 实验步骤

- 修改 `xtfs.c` 中的 `sys_write()` 函数：
 1. 在函数中增加一个名为 `size` 的参数，用于指定要写入的字符数。
 2. 在函数体内，根据 `size` 参数的值来控制要写入的数据量，并调用相应的底层写函数。
- 修改 `write_block` 函数并更新系统调用接口：
 1. 将 `write_block` 函数重命名为 `write_block1`。
 2. 在 `write_block1` 函数中，根据传入的 `size` 参数来复制指定数量的字节到磁盘缓冲区。
- 修改可执行程序 `write.S` 并测试：
 1. 打开 `write.S` 文件，更新系统调用接口的使用，确保传入正确的参数值。
 2. 编译 `write.S` 程序。

17.7 实验结果

如图17.1所示，执行修改后的 `write` 应用程序，成功写入了对应字节数的数据。随后执行读取操作进行验证。



A screenshot of a QEMU terminal window. The title bar at the top right says "QEMU". Below the title bar is a tab labeled "Machine View". The terminal content shows a series of commands and their outputs in red text on a black background:

```
xtsh# create
create successfully.
xtsh# read
xtsh# write
xtsh# read
12345
xtsh# _
```

图 17.1: 验证修改后的 sys_write

实验十八 多扇区文件读

18.1 实验目的

1. 熟悉磁盘文件结构。
2. 熟悉读文件过程。

18.2 实验内容

修改内核文件读部分代码，以支持读取超过 512 字节的文件。创建一个超过 512 字节的文件，用于测试修改后的文件读取功能。

18.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

18.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

18.5 实验原理

xtfs 文件储存的形式是数据块表（一级块），实际上可存储 $512/\text{sizeof}(\text{short}) = 256$ 个扇区。原始 code12 代码默认只读了第一个扇区，为了读取超过 512 字节的文件，需

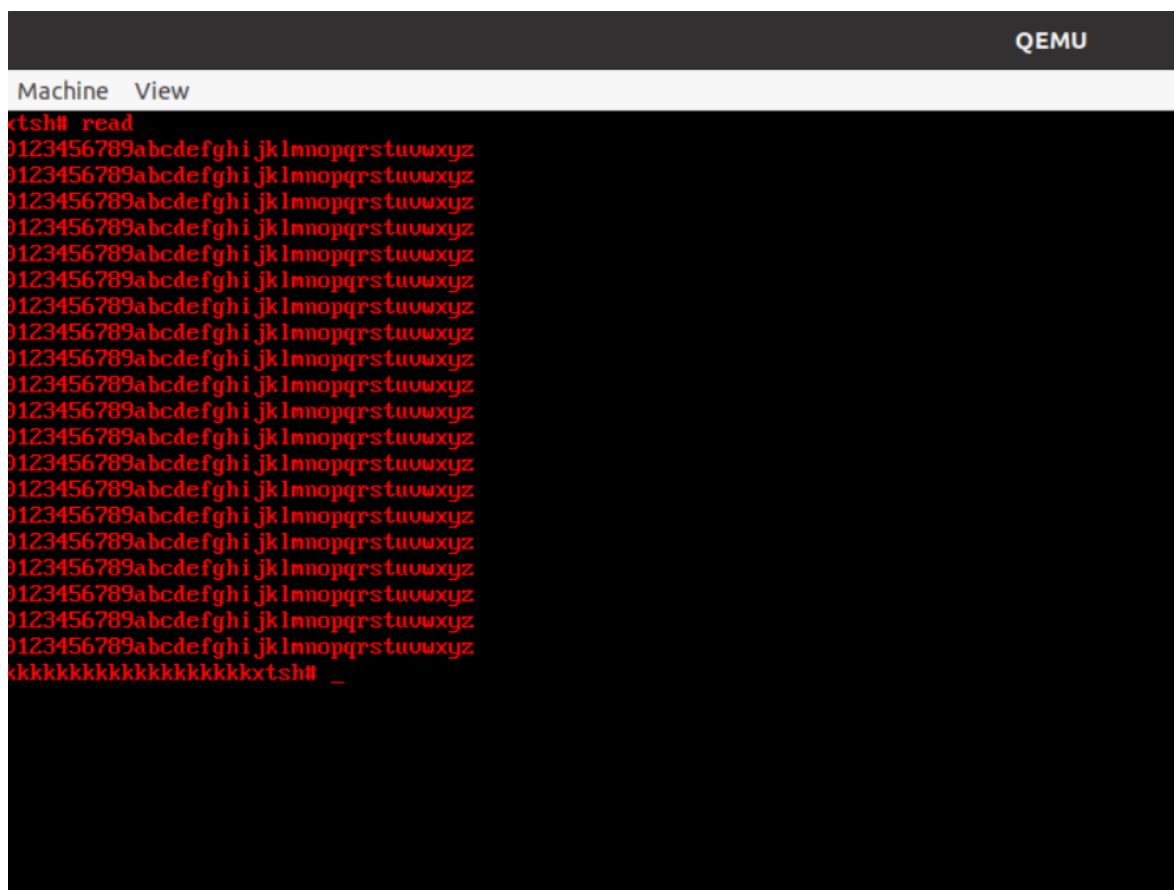
要遍历数据块表，读取所有块。

18.6 实验步骤

- 打开 `xtfs.c` 文件，找到 `sys_read()` 函数。
 1. 根据 `inode->size` 计算出文件包含的扇区数。
 2. 修改 `sys_read()` 函数，将原有读取第一个扇区的操作改为循环读取每个扇区。
 3. 确保在循环读取过程中，不会超出文件的实际大小。
- 编译并测试修改后的内核代码，确保系统能够正常启动。
- 创建一个超过 512 字节的文件：
 1. 在虚拟机中，使用文本编辑器或命令行工具创建一个包含多行文本的文本文件，确保文件大小超过 512 字节。
 2. 保存文件，并记下文件名和路径。
- 创建一个 `read1.S` 可执行文件或修改 `read` 文件：
 1. 根据需要，可以选择创建一个新的汇编文件 `read1.S`，或修改现有的 `read` 文件。
 2. 在文件中编写代码，调用 `sys_read()` 函数读取之前创建的文件。
 3. 编译并运行该文件，验证是否能够正确读取并显示文件的所有内容。

18.7 实验结果

如图18.1所示，在 `xtsh` 中执行 `read` 命令，读取文件内容。`xtsh` 中显示出文件的所有内容，验证了修改后的文件读取功能。



The image shows a terminal window titled "Machine View" with a "QEMU" logo in the top right corner. The terminal displays a series of red text lines representing a file read operation. The first line is "xtsh# read". This is followed by 20 lines of data, each starting with a hexadecimal address "0123456789" and containing the string "abcdefghijklmnopqrstuvwxy". The final line of the sequence is "xxxxxxxxxxxxxxxxxxxxxxxxxtsh# _", where the underscores represent the continuation of the data stream.

```
Machine View
QEMU
xtsh# read
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
0123456789abcdefghijklmnopqrstuvwxy
xxxxxxxxxxxxxxxxxxxxxxxxxtsh# _
```

图 18.1: 验证读写超过 512 字节的文件

实验十九 多扇区文件写

19.1 实验目的

1. 熟悉磁盘文件结构。
2. 熟悉写文件过程。

19.2 实验内容

修改内核文件写部分代码，以支持写入超过 512 字节的文件。写入文件一个超过 512 字节的字符串，并读出文件所有内容，验证写入是否成功。

19.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

19.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

19.5 实验原理

xtfs 文件储存的形式是数据块表（一级块），实际上可存储 $512/\text{sizeof}(\text{short}) = 256$ 个扇区。原始 code12 代码写文件默认只写入一个扇区，即 512 字节以内。为了写入

超过一个扇区（512 字节）的字符串，需要修改写文件函数，以支持多扇区写入。

19.6 实验步骤

- 修改 `sys_write()` 函数
 1. 打开 `xtfs.c` 文件，找到 `sys_write()` 函数。
 2. 修改 `sys_write()` 函数，遍历字符串 `buf` 计算字符串长度，并根据字符串长度计算出所占扇区数。
 3. 在 `sys_write()` 函数中，循环调用 `write_block` 函数，将字符串内容写入各个扇区。
- 创建一个空文件 `t1.txt`，并修改 `init_img.sh` 脚本，将 `t1.txt` 拷贝到镜像中。
 1. 使用文本编辑器创建一个名为 `t1.txt` 的空文件。
 2. 打开 `init_img.sh` 脚本，添加将 `t1.txt` 拷贝到镜像中的命令。
 3. 保存并关闭 `init_img.sh` 脚本。
- 修改 `write.S` 文件。
 1. 打开 `write.S` 文件，找到写文件的代码部分。
 2. 将写入的文件名改为 `t1.txt`。
 3. 将原有写入的字符串改成超过 512 字节的字符串。
 4. 保存并关闭 `write.S` 文件。
- 修改 `read.S` 可执行文件，读取 `t1.txt` 文件内容（在 `lab30` 的基础上可读取超过 512 字节的文件）。
 1. 打开 `read.S` 文件，找到读文件的代码部分。
 2. 确保读文件代码能够支持读取超过 512 字节的文件。
 3. 保存并关闭 `read.S` 文件。
 4. 编译 `read.S` 文件，生成可执行文件。

实验二十 增加文件读写属性

20.1 实验目的

1. 熟悉 inode 结构中各个字段的含义和作用。
2. 熟悉文件系统打开、读写相关代码的实现原理。

20.2 实验内容

在 `sys_write` 函数中增加检查文件是否只读的逻辑。创建相关的可执行文件，用于验证文件的读写属性是否按预期工作。

20.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

20.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

20.5 实验原理

在现有的 inode 结构中，`type` 字段用于表示文件的类型。目前，`type` 字段的值为 0 表示空文件，1 表示可执行文件，2 表示可读文件。为了增加文件只读属性，给

inode->type 增加两个新的值：3 表示只读文件，4 表示只写文件。因此，type 字段的新含义为：2 表示可读可写文件，3 表示只读文件，4 表示只写文件。

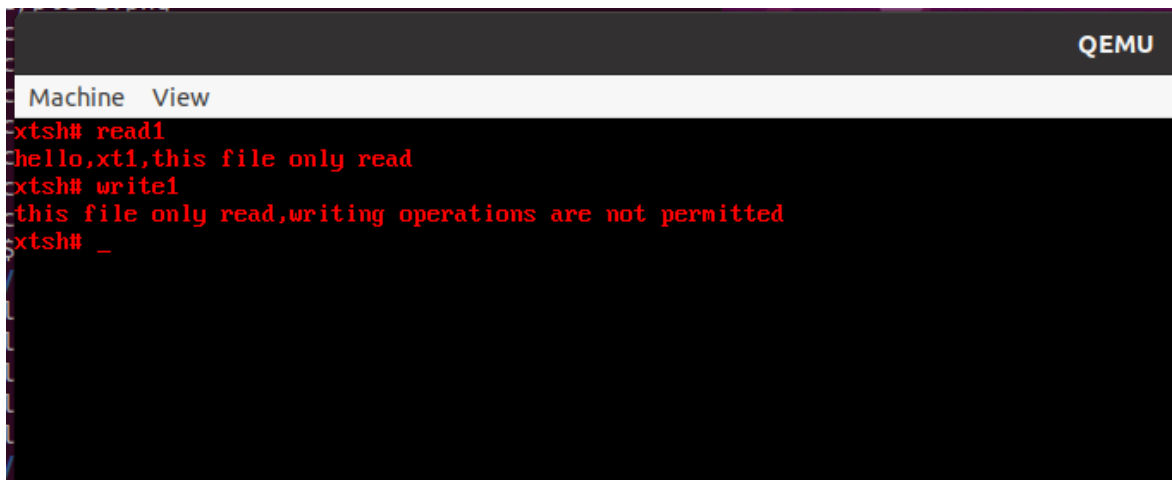
20.6 实验步骤

- 准备阶段：
 1. 下载并解压 MaQueOS 代码仓库中的相关代码。
 2. 熟悉 xfs 文件系统的源代码，特别是 inode 结构和 sys_open、sys_write 等函数的实现。
- 修改 sys_open() 函数：
 1. 在 xfs.c 文件中找到 sys_open() 函数的实现。
 2. 修改 sys_open() 函数，使其能够正确处理 type 值为 2（可读可写文件）、3（只读文件）和 4（只写文件）的情况，确保这些文件都能被正常打开。
- 修改 sys_write() 函数：
 1. 在 xfs.c 文件中找到 sys_write() 函数的实现。
 2. 在 sys_write() 函数中添加逻辑，当文件的 type 值为 3（只读文件）时，不允许写入数据，并打印出提示信息。
- 修改 init_img.sh 脚本：
 1. 找到 init_img.sh 脚本文件。
 2. 修改 init_img.sh 脚本，使其能够向模拟磁盘镜像中拷贝一个 type 值为 3 的只读文件。
- 创建测试文件：
 1. 编写 read1.S 和 write1.s 两个汇编语言文件。
 2. read1.S 文件用于测试读取文件内容的功能，write1.s 文件用于测试写入文件内容的功能。
 3. 编译 read1.S 和 write1.s 文件，生成可执行文件 read1 和 write1。

- 测试阶段：
 1. 在 qemu 虚拟机中启动 MaQueOS 操作系统。
 2. 使用 xtsh 命令行工具，执行 read1 和 write1 命令，测试文件的读写属性是否按预期工作。
 3. 记录实验结果，包括命令的输出信息和任何异常情况。
- 调试与优化：
 1. 如果测试结果不符合预期，使用调试工具（如 gdb）对代码进行调试，查找并修复问题。
 2. 优化代码性能，确保文件系统的读写操作高效且稳定。

20.7 实验结果

如图20.1所示，在 xtsh 中执行 read1 命令和 write1 命令。read1 可正常读取文件内容，而 write1 在尝试写入只读文件时失败，并打印出提示信息。



```
Machine View
xtsh# read1
hello,xt1,this file only read
xtsh# write1
this file only read,writing operations are not permitted
xtsh# _
```

图 20.1: 文件读写属性验证

实验二十一 实现返回文件大小的系统调用

21.1 实验目的

1. 掌握增加新的系统调用。
2. 熟悉 inode 结构体字段含义，能自己增加文件系统的新功能。

21.2 实验内容

增加一个系统调用，`sys_file_size`，接受一个参数文件名，返回该文件大小。编写一个测试汇编文件 `filesize.S`，用于返回 `hello_xt` 文件的大小。

21.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

21.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

21.5 实验原理

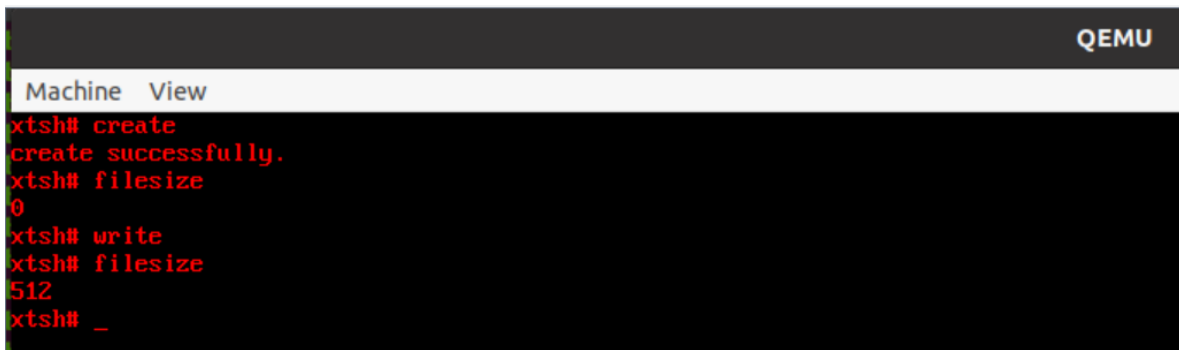
原来内核写入文件时，是直接写入 512 字节（即一个磁盘块大小）。现在，`copy_mem` 函数已修改为可以复制指定数量的字节到 `buf`。同时，为了支持新功能，增加一个接受三个参数的系统调用。

21.6 实验步骤

- 编写 `sys_file_size` 函数：
 1. 首先找到文件名对应的 `inode` 结构体。
 2. 在找到 `inode` 结构体后，通过访问 `inode->size` 字段即可获取文件的大小。
 3. 为了在 `maqueos` 中打印文件大小（因为 `maqueos` 无法直接打印整数），需要增加一个函数，用于将整数 `int` 转换成字符串数组 `char[]`。
 4. 编写测试汇编文件 `filesize.S`，通过调用 `sys_file_size` 系统调用来获取 `hello_xt` 文件的大小，并打印出来。
- 在命令行里依次执行以下命令进行测试：
 1. 执行 `create` 命令，创建 `hello_xt` 文件。
 2. 执行 `filesize` 命令，此时应显示文件大小为 0，因为文件刚创建还未写入内容。
 3. 执行 `write` 命令，向 `hello_xt` 文件写入内容。
 4. 再次执行 `filesize` 命令，此时应显示文件的大小为实际写入的字节数（例如 512 字节，如果写入了 512 字节的内容）。

21.7 实验结果

如图21.1所示，先执行 `create` 命令创建 `hello_xt` 文件，再执行 `filesize` 命令发现文件大小为 0；然后执行 `write` 命令向 `hello_xt` 文件写入内容，再次执行 `filesize` 命令时，发现文件大小已变为 512 字节。



```
Machine View
xtsh# create
create successfully.
xtsh# filesize
0
xtsh# write
xtsh# filesize
512
xtsh# _
```

图 21.1: 调用验证 sys_file_size

实验二十二 实现返回文件属性的系统调用

22.1 实验目的

1. 掌握文件属性获取操作。
2. 增加一个系统调用 `sys_filetype`，参数为文件名，返回该文件的读、写、执行属性。

22.2 实验内容

编写一个 `sys_filetype` 系统调用，用于返回文件的属性。编写几个汇编可执行文件，向磁盘镜像中拷贝几个文件，并使用 `sys_filetype` 系统调用输出这几个文件的读、写、执行属性。

22.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

22.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

22.5 实验原理

目前 inode 的 type 字段定义如下：0 表示空，1 表示可执行文件，2 表示可读文件。为了支持更多文件类型，给 inode->type 增加 3 和 4 两个值，使 type 字段的含义变为：2 表示可读可写文件，3 表示只读文件，4 表示只写文件。通过读取 inode->type 的值，可以判断文件的类型。

22.6 实验步骤

- 编写 sys_filetype 函数：
 1. 根据文件名在文件系统中查找，找到对应名的 inode 结构体。
 2. 根据找到的 inode 的 type 字段值，判断文件的类型，并准备返回相应的文件属性。
 3. 修改 init_img.sh 脚本，向磁盘镜像中拷贝各种类型的文件，包括可读文件、可写文件、只读文件、只写文件以及可执行文件等。
 4. 编写汇编测试文件 ftype1.S, ftype2.S, ftype3.S 等，这些文件将使用 sys_filetype 系统调用来测试不同文件的类型，并输出测试结果。
- 在命令行中依次执行以下步骤进行测试：
 1. 运行 init_img.sh 脚本，初始化磁盘镜像并拷贝文件。
 2. 编译并运行 ftype1.S, ftype2.S, ftype3.S 等汇编测试文件，观察并记录输出结果。

22.7 实验结果

如图22.1所示，执行 ftype1, ftype2, ftype3, ftype4 等汇编测试文件后，成功输出了各种文件的类型属性。



A screenshot of the QEMU Machine View terminal window. The window has a dark background with red text. The title bar at the top right says 'QEMU'. Below the title bar, the text 'Machine View' is displayed. The terminal shows a series of commands and their outputs:

```
xtsh# ftype1
executable file
xtsh# ftype2
read and write file
xtsh# ftype3
only read file
xtsh# ftype4
only write file
xtsh# _
```

图 22.1: 调用 `sys_filetype` 查看文件属性

实验二十三 实现 ls 应用程序

23.1 实验目的

1. 理解 inode 文件名的存储方式。
2. 掌握添加一个新的系统调用。

23.2 实验内容

在 xtsh 命令行中，实现一个类似 Linux 的 ls 命令，用于列出系统内的所有文件。

23.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

23.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

23.5 实验原理

在挂载文件系统时，所有 inode 的信息已被读取并存储在 inode_table 中，因此，通过遍历 inode_table 即可获取所有文件的信息。

23.6 实验步骤

- 实现 `sys_ls()` 系统调用：
 1. 遍历 `inode_table` 的逻辑，并打印出每个 `inode` 结构体中的文件名（即 `inode->file_name`）。
 2. 确保 `sys_ls()` 函数的返回值和错误处理机制正确无误。
- 编写 `ls.S` 汇编语言文件：
 1. 创建一个新的汇编语言文件 `ls.S`，该文件将使用 `sys_ls()` 系统调用来列出系统内的所有文件。
 2. 在 `ls.S` 文件中，编写汇编代码，包括设置系统调用号、传递参数（如果有的话）、调用系统调用以及处理返回值等。

23.7 实验结果

如图23.1所示，通过 `ls` 命令成功列出了硬盘中的所有文件。



```
QEMU
Machine  View
xtsh# ls
xtsh
print
share
shmem
hello
read
write
create
destroy
sync
ls
xtsh# _
```

图 23.1: 验证 `ls` 命令

实验二十四 实现可以打印文件详细信息的 ls 应用程序

24.1 实验目的

1. 理解 inode 各个字段的意义，包括文件类型、大小、权限等。
2. 熟悉磁盘结构和文件系统的工作原理。
3. 学会在操作系统中添加一个新的系统调用，并实现其功能。

24.2 实验内容

实现一个类似 Linux 的 `ls -l` 命令的应用程序，列出系统所有文件以及文件的详细内容，包括文件名、文件类型、文件大小等。

24.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

24.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

24.5 实验原理

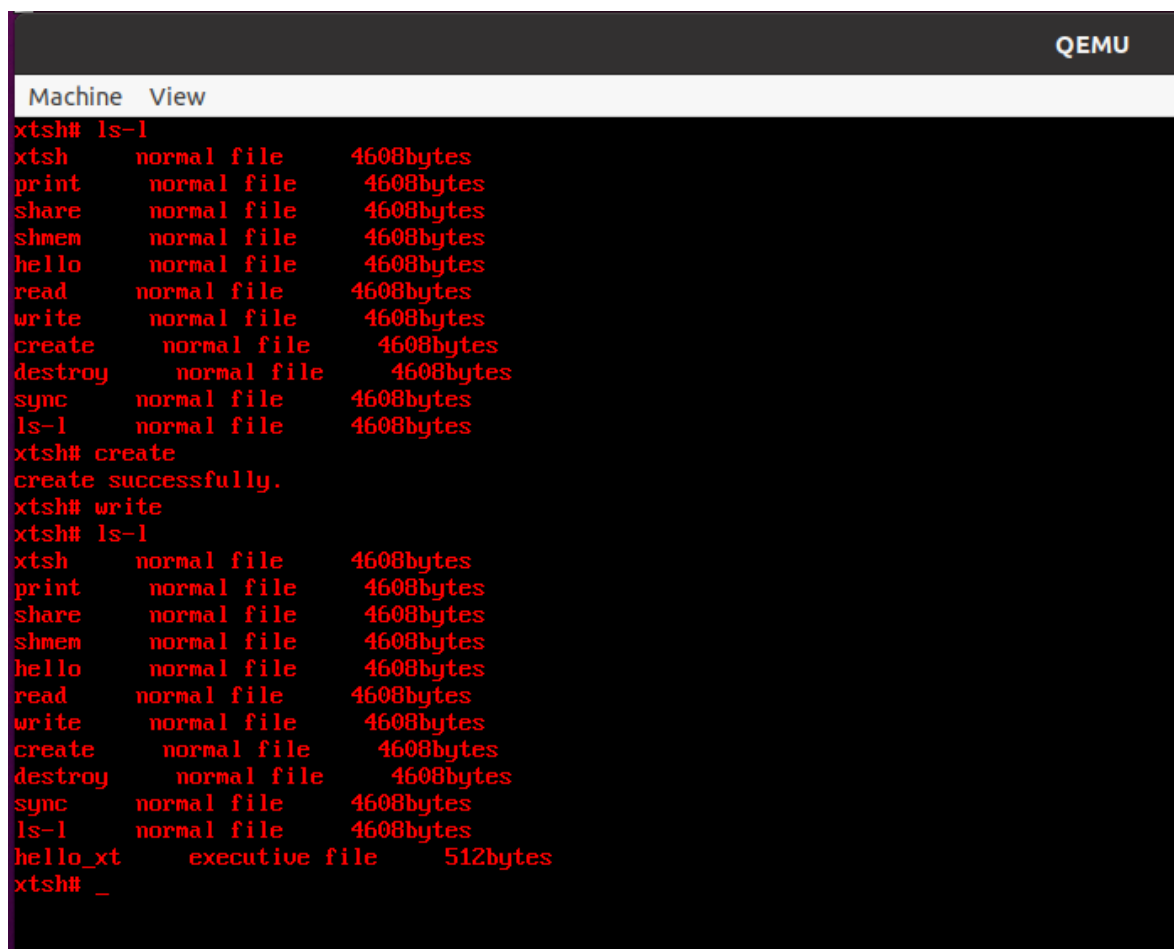
挂载时，操作系统已将 inode 的所有信息读取到 `inode_table` 中。通过遍历 `inode_table`，可以访问每个 inode 的 `type`、`size` 等字段，从而获取文件的类型和大小等信息。

24.6 实验步骤

- 实现系统调用：
 1. 在操作系统源码中定义一个新的系统调用 `sys_ls_l()`。
 2. 实现 `sys_ls_l()` 函数的逻辑，遍历 `inode_table` 并获取每个 inode 的信息。
 3. 编写辅助函数，用于将整数转换为字符串，以便在 `printk` 中使用。
- 打印文件信息：
 1. 在 `sys_ls_l()` 函数中，打印每个 inode 的文件名 (`inode->file_name`)。
 2. 根据 `inode->type` 的值，判断文件类型并打印相应的字符串（如可执行文件、普通文件等）。
 3. 使用辅助函数将 `inode->size` 转换为字符串，并打印文件大小。
- 编写测试代码：
 1. 编写一个汇编语言文件 `ls-l.S`，该文件在运行时调用 `sys_ls_l()` 系统调用。
 2. 确保 `ls-l.S` 文件能够正确编译并链接成可执行文件。

24.7 实验结果

如图24.1所示，`ls-l` 命令成功列出了系统所有文件，以及每个文件的详细信息，包括文件名、文件类型和文件大小等。



The image shows a QEMU Machine View window with a terminal. The terminal output is as follows:

```
Machine View
xtsh# ls-l
xtsh  normal file      4608bytes
print  normal file      4608bytes
share  normal file      4608bytes
shmem  normal file      4608bytes
hello  normal file      4608bytes
read   normal file      4608bytes
write  normal file      4608bytes
create normal file      4608bytes
destroy normal file      4608bytes
sync   normal file      4608bytes
ls-l   normal file      4608bytes
xtsh# create
create successfully.
xtsh# write
xtsh# ls-l
xtsh  normal file      4608bytes
print  normal file      4608bytes
share  normal file      4608bytes
shmem  normal file      4608bytes
hello  normal file      4608bytes
read   normal file      4608bytes
write  normal file      4608bytes
create normal file      4608bytes
destroy normal file      4608bytes
sync   normal file      4608bytes
ls-l   normal file      4608bytes
hello_xt executive file  512bytes
xtsh# _
```

图 24.1: 验证 `ls-l` 命令

实验二十五 实现打印当前进程打开文件的系统调用

25.1 实验目的

1. 熟悉进程打开文件列表数据的数据结构。
2. 理解 file 和 inode 的关系。

25.2 实验内容

实现一个名为 `sys_current_open_files` 的系统调用，用于打印当前进程打开的文件。在可执行文件中调用 `sys_current_open_files` 系统调用，查看打开的文件。可以选择自己编写新的汇编文件，也可以修改原有的汇编文件。

25.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

25.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

25.5 实验原理

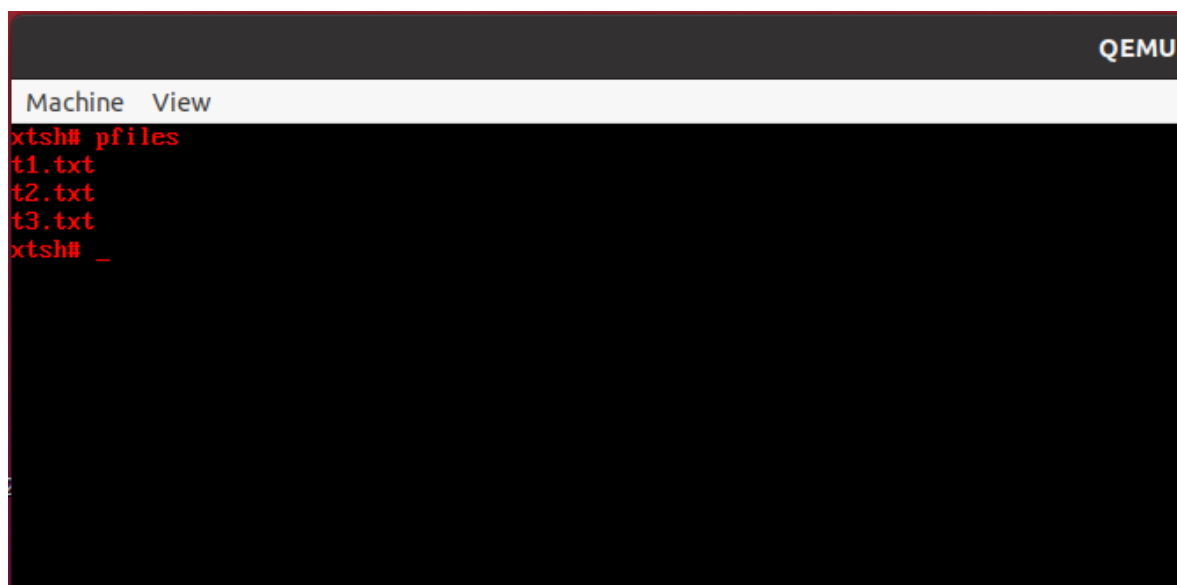
进程结构体 `struct process` 中, 有一个字段名为 `file_table`, 表示当前进程打开的文件列表。全局变量 `struct process *current` 表示当前进程。通过遍历 `current->file_table`, 可以获取当前进程打开的所有文件信息。如果 `file` 的 `inode` 字段不为 0 (`null`), 则表示该文件项被占用。

25.6 实验步骤

- 实现 `sys_current_open_files` 系统调用:
 1. 遍历当前进程的 `file_table`, 根据 `inode` 的 `file_name` 字段打印文件名。
 2. 在系统调用数组中, 添加该系统调用。
- 创建 `pfiles.S` 汇编文件:
 1. 打开 `t1.txt`、`t2.txt`、`t3.txt` 三个文件后, 调用 `sys_current_open_files` 系统调用。
 2. 确保在 `sys_open` 和 `sys_close` 之间调用该系统调用, 以验证文件是否处于打开状态。

25.7 实验结果

如图25.1结果显示, 执行 `pfiles` 命令, `pfiles.S` 文件中先打开了 `t1.txt`、`t2.txt`、`t3.txt` 三个文件, 然后调用 `sys_current_open_files` 打印出当前进程打开的文件名。



A screenshot of a QEMU terminal window. The title bar at the top right says 'QEMU'. Below the title bar is a menu bar with 'Machine' and 'View'. The terminal content shows a red prompt 'xtsh#' followed by the command 'pfiles'. The output of the command is three lines of red text: 't1.txt', 't2.txt', and 't3.txt'. Below the output, there is another red prompt 'xtsh#' followed by an underscore character '_'. The background of the terminal is black.

```
Machine View
xtsh# pfiles
t1.txt
t2.txt
t3.txt
xtsh# _
```

图 25.1: 验证 sys_current_open_files 命令

实验二十六 从硬盘镜像文件中读取指定文件

26.1 实验目的

1. 理解 xtrfs 磁盘结构以及文件存储方式，包括 inode 表、块表和数据块等组成部分。
2. 能够编写程序，将指定文件从硬盘镜像文件（如 qemu 使用的 xtrfs.img）中正确读取出来。

26.2 实验内容

编写 read.c 程序，从硬盘镜像文件中读取指定文件的内容。

26.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 12 章。
- MaQueOS 代码 code8、12（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

26.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

26.5 实验原理

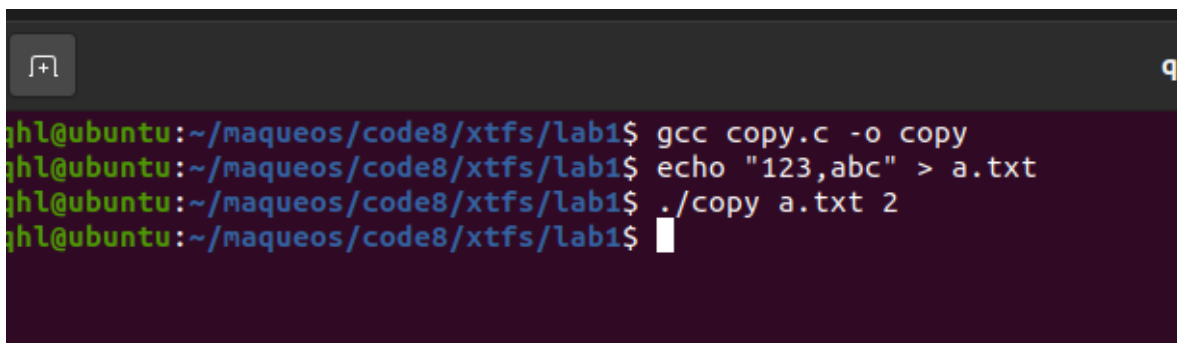
使用 C 语言中的二进制文件读取函数，根据 xfs 磁盘块结构，依次读取 inode 表、块表和数据块，从而获取指定文件的内容。

26.6 实验步骤

- 编写 read.c 程序：
 1. 包含必要的头文件，如 `stdio.h`、`stdlib.h` 等。
 2. 定义磁盘块大小、inode 表大小、块表大小等常量。
 3. 编写函数读取磁盘镜像文件，将其加载到内存中。
 4. 编写函数遍历 inode 表，找到指定文件的 inode。
 5. 编写函数根据 inode 中的块表信息，读取文件内容并打印到屏幕上。
- 初始化磁盘块：
 1. 编译 `copy.c` 和 `format.c` 文件，生成可执行文件。
 2. 执行 `copy.c` 生成的可执行文件，将 `xfs.img` 复制到实验目录下（如果尚未复制）。
 3. 执行 `format.c` 生成的可执行文件，对 `xfs.img` 进行格式化，初始化 inode 表和块表等。
- 测试 read 程序：
 1. 编写一个测试脚本或手动在 ubuntu 命令行中执行相关命令，将一个小文件（如 `a.txt`）写入到硬盘镜像文件中。
 2. 编译 `read.c` 文件，生成可执行文件。
 3. 执行 `read.c` 生成的可执行文件，读取指定文件（如 `a.txt`）的内容，并打印到屏幕上进行验证。

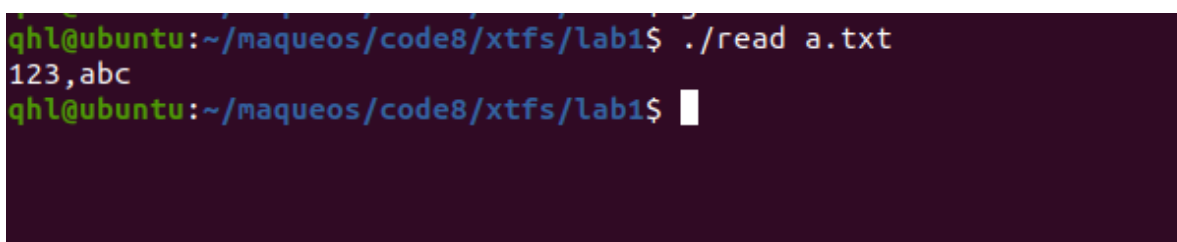
26.7 实验结果

如图26.1所示，首先向硬盘镜像文件中拷贝 a.txt 文件。之后如图26.2所示，运行 read 应用程序，从硬盘镜像文件中读取 a.txt 文件的内容。

A terminal window with a dark background and light green text. The prompt is 'qhl@ubuntu:~/maqueos/code8/xtfs/lab1\$'. The user enters four commands: 'gcc copy.c -o copy', 'echo "123,abc" > a.txt', './copy a.txt 2', and then a blank line.

```
qhl@ubuntu:~/maqueos/code8/xtfs/lab1$ gcc copy.c -o copy
qhl@ubuntu:~/maqueos/code8/xtfs/lab1$ echo "123,abc" > a.txt
qhl@ubuntu:~/maqueos/code8/xtfs/lab1$ ./copy a.txt 2
qhl@ubuntu:~/maqueos/code8/xtfs/lab1$
```

图 26.1: 向磁盘拷贝文件

A terminal window with a dark background and light green text. The prompt is 'qhl@ubuntu:~/maqueos/code8/xtfs/lab1\$'. The user enters './read a.txt', and the output '123,abc' is displayed on the next line.

```
qhl@ubuntu:~/maqueos/code8/xtfs/lab1$ ./read a.txt
123,abc
qhl@ubuntu:~/maqueos/code8/xtfs/lab1$
```

图 26.2: 读文件

实验二十七 自定义字模并显示

27.1 实验目的

1. 通过编写操作系统内核代码，实现在显示器上显示自定义的字符。
2. 了解显示器的显示原理，通过向 font 数组中添加元素，对单个字符的显示进行熟悉。

27.2 实验内容

定义一个字模，并把字模显示在显示屏上。

27.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 1 章。
- MaQueOS 代码 code1（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

27.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

27.5 实验原理

字模通常是一个二维数组，每个数组元素代表显示屏上对应像素的亮灭状态。每个字符可以用一个矩阵表示，矩阵中的每个元素对应显示屏上的一个像素点。通过设

计这样的字模，可以灵活地控制每个字符的显示效果。

在程序中，通过将字模数据映射到相应的显示屏像素，控制屏幕上对应位置的亮度。显示屏上的每个字符通常是由多个像素点组成，实验中通过逐个处理字模的像素，确保字符在显示屏上的正确显示。

27.6 实验步骤

- 修改 kernel/drv/font.c 文件，添加自定义字模：
 1. font.c 文件中只包含一个名字为 font 的字符串数组，数组中由 16 组八位二进制数构成一个字模。
 2. 在数组最后添加一个全部由 1 组成的 16 组八位二进制数定义的字模，显示是一个与字符长宽相同的亮块。
- 修改 kernel/init/main.c 文件的主函数：
 1. 添加的字模是数组中的第 127 个元素，所以在调用 write_char 函数时，第一个输入的参数是 127。
 2. 调用 write_char 函数，把字模显示在屏幕的第一行第一列。

27.7 实验结果

实验中定义的字模内容全部为 1，显示在显示屏上的结果是一个与字符长宽相同的亮块，验证了字模的设计和显示控制正确性，结果如图27.1所示：



图 27.1: 字符显示图片

实验二十八 显示一架红色飞机

28.1 实验目的

1. 通过编写操作系统内核代码，实现在显示器上显示想要的形式。
2. 了解显示器的显示原理，以及如何编写简单的显示器驱动程序来显示图案，辨别与显示单个字符的区别。

28.2 实验内容

在本实验中，需要设计一个飞机的样式，并通过控制显示屏的颜色输出功能，将飞机图案以红色显示在屏幕上。

28.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 1 章。
- MaQueOS 代码 code1（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

28.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

28.5 实验原理

对于颜色的指定可以通过修改像素的 RGB 值来指定颜色。红色可以通过设置 RGB 中的红色通道为最大值，其它通道为 0 来实现。

28.6 实验步骤

- 在 kernel/drv/console.c 文件中添加一个 plane 函数：
 1. 在 console.c 中添加 plane 函数。
 2. 使用 printk 对 ‘*’ 进行打印，使出现一个飞机的形状。
- 在 kernel/init/main.c 文件中调用 plane 函数。

28.7 实验结果

实验成功在显示屏上显示了一架清晰的红色飞机图案，验证了自定义图形显示与颜色控制的实现效果，结果如图28.1所示：



图 28.1: 红色飞机图案

实验二十九 实现光标闪烁的效果

29.1 实验目的

1. 通过编写操作系统内核代码，对时钟中断进行控制，实现自己想要的效果。
2. 通过使用定时器来周期性切换光标的状态，掌握定时器的配置与使用方法，理解定时事件的触发机制以及如何利用定时器实现周期性任务。

29.2 实验内容

通过配置时钟中断，在每次中断发生时切换光标的显示状态。

29.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 2 章。
- MaQueOS 代码 code2（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

29.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

29.5 实验原理

本实验的原理是通过结合时钟中断和标志位的动态更新，实现光标的闪烁功能。具体而言，程序首先设置一个标志位，用于表示光标的当前显示状态。当标志位为 1

时，光标被隐藏，不在屏幕上显示；当标志位为 0 时，光标被显示在指定位置。时钟中断每隔一定时间触发一次，触发时通过修改标志位的值来切换光标的显示状态，从而在屏幕上呈现光标的闪烁效果。通过这一机制，程序能够以固定频率控制光标的显隐变化，模拟出光标的闪烁行为，同时确保对其他任务的实时响应。

29.6 实验步骤

- 将 kernel/excp/exception.c 文件中的 timer_interrupt 函数移动到 kernel/drv/console.c 文件。
- 并对 timer_interrupt 函数进行修改：
 1. 在代码中设置 add 为一个初始值，作为计时变量，用于控制光标闪烁的速度。设定 flag 为 0，表示光标的初始状态为不可见。
 2. 每次中断时，add 值递减。
 3. 在每次中断时，判断 add 是否等于 0。
 - (a) add 等于 0，则执行光标闪烁逻辑；
 - i. flag 等于 0 时，显示光标；
 - ii. flag 等于 1 时，擦除光标；
 - (b) add 不等于 0，继续等待下次中断。

29.7 实验结果

实验成功实现了光标闪烁功能，在等待几秒后，光标会出现在屏幕上。光标在时钟中断的控制下，每隔一定时间自动切换显示状态，形成稳定的闪烁效果，结果如动画29.1所示：

图 29.1: 光标闪烁动图

实验三十 支持光标在显示器上上下下左右移动

30.1 实验目的

1. 通过编写操作系统内核代码，对键盘中断进行控制，实现自己想要的效果。
2. 了解键盘中断的中断原理，以及如何编写中断控制代码来实现控制中断。在编写键盘中断代码的过程中，加深对于键盘中断原理的熟悉

30.2 实验内容

本实验的目标是实现光标在显示器上上下下左右移动。通过编程控制光标的位置，每次用户输入方向键（上、下、左、右）时，光标会根据按键指令移动到对应的屏幕位置。

30.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 3 章。
- MaQueOS 代码 code3（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

30.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

30.5 实验原理

本实验的原理是通过捕获键盘的输入信号，并识别键盘上“上”、“下”、“左”、“右”四个方向键的扫描码来实现光标在显示器上的上下左右移动功能。程序首先通过键盘中断机制实时获取按键输入信号，解析方向键的扫描码，判断具体的按键动作。每次捕获到方向键的输入时，程序根据当前光标的位置和对应方向的按键逻辑，动态调整光标在显示器上的坐标位置。

为确保光标移动的准确性，程序会对光标的新位置进行边界检测，防止光标超出显示区域。通过这种方式，实现了光标与方向键的交互，使用户能够直观地控制光标在显示器上的移动，从而提升系统的可操作性和交互体验。

30.6 实验步骤

- 读取键盘扫描码，调用系统中断处理函数 `keyboard_interrupt` 函数。
- 修改 `kernel/drv/console.c` 文件中的 `do_keyboard` 函数：
 1. 如果键盘中断码识别到“上”、“下”、“左”、“右”四个方向键。
 - 如果是第一次写入（即 `isFirst` 等于 1），不用保存数据，并且把 `isFirst` 置 0。
 - * 0x75 上箭头：向上移动光标，前提是当前行（`y`）大于 0。
 - * 0x72 下箭头：向下移动光标，前提是当前行小于屏幕最大行数。
 - * 0x6b 左箭头：向左移动光标，如果是行首（`x` 等于 0），移动到上一行的行尾。
 - * 0x74 右箭头：向右移动光标，如果是行尾（`x` 等于 `NR_CHAR_X - 1`），移动到下一行的行首。
 - 如果不是第一次写入（即 `isFirst` 不等于 1），需要保存数据。
 - * 把 `pos` 指向显存中光标当前位置对应的显存地址。
 - * 恢复显存数据。
 - * 把 `temp` 指针重新指向缓冲区的起始地址。
 2. 将光标处的显存数据存储到缓冲区，保存数据。

30.7 实验结果

实验成功实现了光标的上下左右移动功能，光标能够根据用户输入的方向键准确地显示器上移动，移动过程流畅无误，符合预期的实验目标，结果如动画30.1所示：

图 30.1: 光标移动动图

实验三十一 支持退格键删除字符

31.1 实验目的

1. 通过编写操作系统内核代码，对键盘中断进行控制
2. 学习如何使用缓冲区管理输入内容，尤其是处理退格操作时字符的移除和光标位置的更新。

31.2 实验内容

本实验的目标是实现退格键删除字符的功能。通过监听键盘输入，检测是否按下退格键（Backspace）。当按下退格键时，程序会删除光标前一个位置的字符，并更新显示内容。

31.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 3 章。
- MaQueOS 代码 code3（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

31.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

31.5 实验原理

系统维护一个字符缓冲区，存储当前输入的所有字符。每当用户输入一个字符时，该字符会被添加到缓冲区的末尾。一旦检测到退格键，程序会从缓冲区中删除最后一个字符。更新后的字符缓冲区会被重新渲染到显示屏上，删除后的字符被从屏幕上移除，保持与用户输入的状态同步。

31.6 实验步骤

- 将 `kernel/excp/exception.c` 中的 `keyboard_interrupt` 函数移动到 `kernel/drv/console.c` 文件中。
- 对 `keyboard_interrupt` 函数进行修改：
 1. 读取键盘扫描码。
 2. 识别键盘扫描码，调用 `keyboard_interrupt` 函数。
 - (a) 擦除显示在坐标 (x, y) 的光标。
 - (b) 判断 x 坐标是否为 0，如果不为 0，x 减 1。
 - (c) 若 x 坐标为 0，且 y 坐标不为 0，y 减 1。
 - (d) 擦除 (x,y) 处的字符，并写入光标。

31.7 实验结果

实验成功实现了退格键删除字符功能。当用户按下退格键时，光标前的字符会被正确删除，并且显示内容实时更新，字符删除后的效果准确反映在屏幕上，光标位置也正确回退，符合预期的实验目标，结果如动画31.1所示：

图 31.1: 退格删除动图

实验三十二 实现对大写字母的识别

32.1 实验目的

1. 通过编写操作系统内核代码，对键盘中断进行控制，实现对大写字母的识别。
2. 通过实现对大写字母的识别功能，学习键盘输入的处理流程，包括按键扫描码的捕获和映射，以及如何将输入信号解析为实际字符。

32.2 实验内容

把书中 code3 对于小写字母 a 的判断，修改为对于大写字母 A 的判断，大小写切换方式为同时按下 shift 切换。

32.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 3 章。
- MaQueOS 代码 code3（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

32.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

32.5 实验原理

本实验的原理是通过设置一个标记变量来记录 Shift 键的状态，从而实现对键盘输入的大写字母识别功能。

具体来说，当 Shift 键被按下时，键盘中断程序会检测到对应的扫描码，并将标记变量设置为 1；当 Shift 键松开时，程序再次检测到松开事件的扫描码，并将标记变量复原为 0。根据标记变量的值，程序动态调整对键盘输入的处理逻辑：当标记变量为 1 时，程序将输入识别为大写字母；当标记变量为 0 时，程序识别为小写字母。

32.6 实验步骤

- 对 kernel/drv/console.c 文件中的 keyboard_interrupt 函数进行修改：
 1. 并定义一个标志变量。
 2. 如果读取的扫描码为 0xf0，表示按键松开。
 - (a) 检查松开键是否为 Shift 键，如果是，则将标志位 def 设置为 0，表示 Shift 键已松开。
 - (b) 检查松开键是否为 Shift 键，如果不是，则将标志位 def 设置为 0，表示 Shift 键已松开。
 3. 如果当前扫描码为 Shift 键（扫描码 0x12 或 0x59），设置标志位 def 为 1，表示 Shift 键按下。
 4. 将当前扫描码传递给 do_keyboard 函数
- 对 kernel/drv/console.c 文件中的 do_keyboard 函数进行修改：
 1. 准备的大写字母键盘映射和的小写字母键盘映射，分别用于处理普通按键和按下 Shift 键时的按键映射。
 2. 当按下大写字母 A，分配页面。
 3. 当按下小写字母 s，释放页面。

32.7 实验结果

在本次实验中把对小写字母'a'的识别转化为对大写字母'A'的识别。只有在识别为大写字母时，才会分配页面，符合预期的实验目标，结果如图32.1所示：

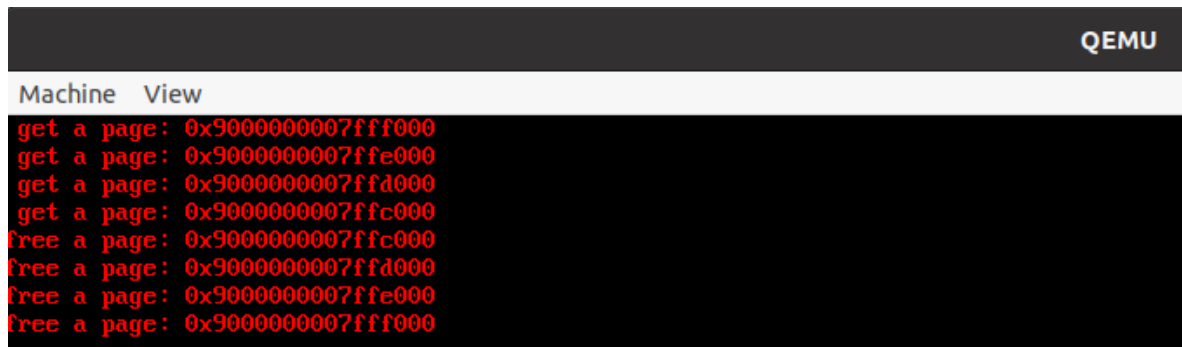


图 32.1: 大写字母识别分配页面图片

实验三十三 实现遍历页表的函数

33.1 实验目的

1. 通过编写操作系统内核代码，对页表进行读取
2. 了解进程 0 的创建过程，以及虚拟地址与物理地址之间的映射

33.2 实验内容

本实验的目标是实现一个遍历页表的函数 walk，用于解析虚拟地址到物理地址的映射。通过该函数，能够遍历页表的多级结构，计算并输出虚拟地址对应的物理地址。

33.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 4 章。
- MaQueOS 代码 code4（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

33.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

33.5 实验原理

MaQueOS 中的页表映射使用二级页表结构。在进行虚拟地址到物理地址的映射时,需要将 64 位的虚拟地址分为 3 部分:页目录项索引、页表项索引和页内偏移。其中,页目录项索引用于在页目录中定位页目录项,页表项索引用于在页表中定位页表项,页内偏移用于在物理页中定位虚拟地址转换到的物理地址。

33.6 实验步骤

- 在 kernel/proc/process.c 文件中添加 walk 函数:
 1. 定义遍历的虚拟地址范围,从 0x800 到 0x1200,步长为 0x100。这是遍历的虚拟地址范围,用于逐步获取对应的物理地址。
 2. 在循环内,计算当前虚拟地址的偏移量(低 12 位),用于后续物理地址计算。
 3. 获取一级页表的基地址,并用 DMW_MASK 处理后存入 base 变量。
 4. 根据虚拟地址的高位,计算一级页表的索引。然后计算获取一级页表项的地址。
 5. 利用一级页表项的值和虚拟地址的中间部分,计算二级页表的索引,再计算获取二级页表项地址。
 6. 计算最终物理地址,将偏移量和二级页表项地址相加,并清除 DMW_MASK 的影响。
 7. 使用 print_debug 函数,分别输出当前虚拟地址和对应的物理地址,验证地址映射关系是否正确。
- 在 kernel/init/main.c 文件中调用 walk 函数。

33.7 实验结果

通过 walk 函数遍历页表,成功输出了指定虚拟地址范围内的虚拟地址 0x0000000000000800 与对应的物理地址 0x0000000007ffd80f,显示 walk 函数能够正确计算页表项,获取二级页表的映射关系,结果如图33.1所示。

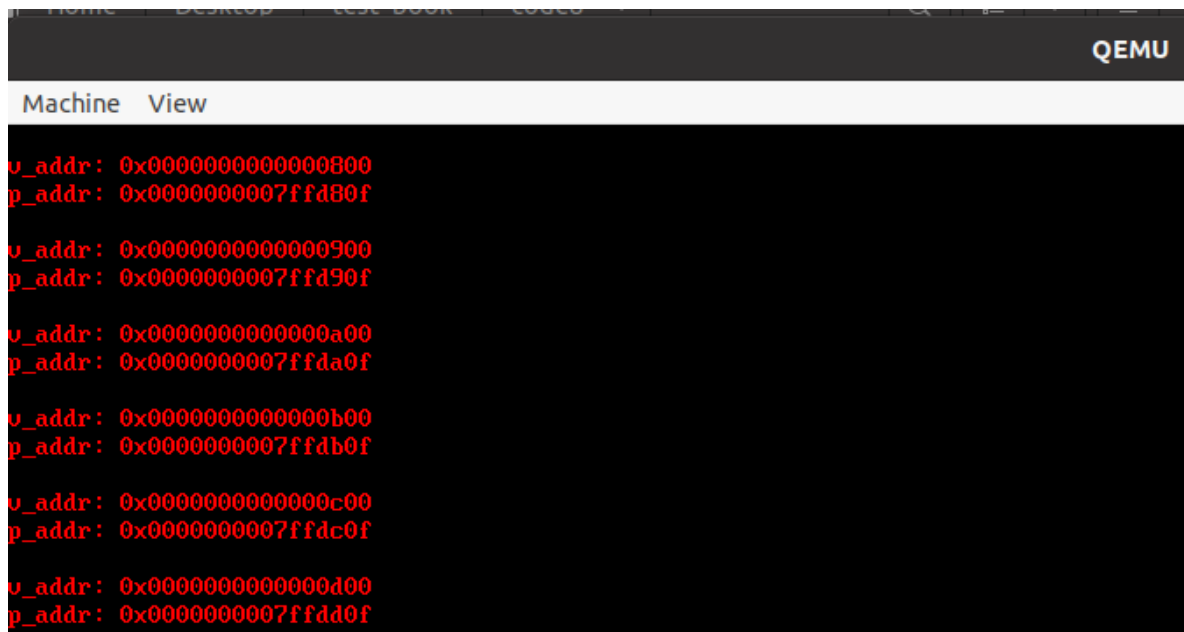


图 33.1: 页表遍历图片

实验三十四 实现 wait 系统调用

34.1 实验目的

1. 通过编写操作系统内核代码，对系统调用进行了解。
2. 了解进程挂起、唤醒和终止过程，以及三个状态之间的切换，掌握如何在操作系统中实现进程的状态监控和管理。

34.2 实验内容

本实验的目标是实现一个系统调用 `wait`，用于检测并等待子进程的终止状态。通过该系统调用，父进程能够挂起自身，直到子进程结束并返回其退出状态。

34.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 6 章。
- MaQueOS 代码 code6（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

34.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

34.5 实验原理

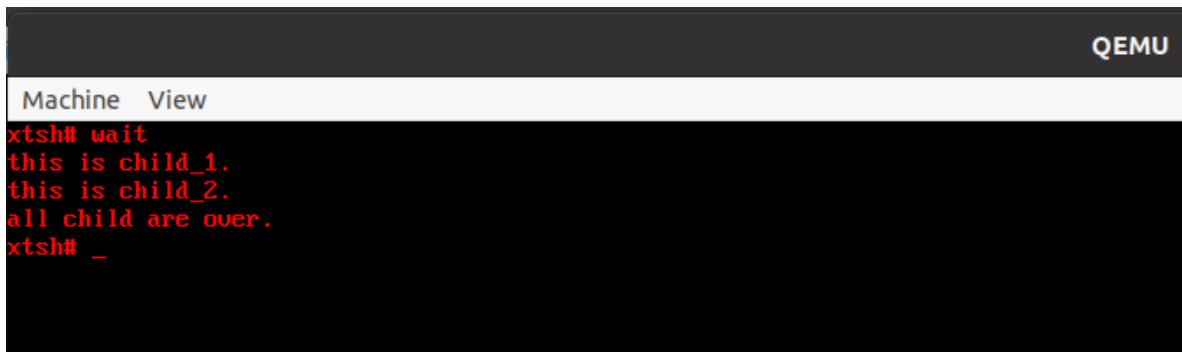
为了实现进程的状态切换，在 MaQueOS 中的 process 数据结构中添加了 state 字段，用于存放进程的当前状态。MaQueOS 支持四种进程状态：可运行状态、不可中断挂起状态、可中断挂起状态和终止状态，通过读取 state 字段，来判断进程处于什么状态。

34.6 实验步骤

- 在 code12/kernel/proc/process.c 文件中添加了 sys_wait 函数：
 1. 定义了一个循环变量 i，用于遍历进程表中的条目。
 2. 遍历所有进程表项。
 3. 如果 process[i] 的父进程等于当前进程 current，说明当前进程存在一个子进程。
 - (a) 如果有子进程，返回 1。
 - (b) 如果没有子进程，返回 0。
- 并在 code12/kernel/xtfs/bin 目录下添加了 wait.S 文件进行验证。
 1. 父进程创建子进程 child1，输出 “this is child_1”。
 2. 父进程创建子进程 child2，输出 “this is child_2”。
 3. 父进程调用 wait 系统调用等待一个子进程结束。
 4. 如果返回值不为 0（有子进程尚未结束），继续等待。
 5. 如果返回值为 0，说明所有子进程都已结束，终结进程，输出 “all child are over”。

34.7 实验结果

在命令行中输入了 wait 命令，调用了 wait 系统调用，先输出了 “this is child_1” 和 “this is child_2”，然后输出了 “all child are over” 表示没有子进程存在，结果如图34.1所示：



```
Machine View
xtsh# wait
this is child_1.
this is child_2.
all child are over.
xtsh# _
```

图 34.1: wait 系统调用结果图片

实验三十五 实现 kill 系统调用

35.1 实验目的

1. 通过编写操作系统内核代码，对系统调用进行了解。
2. 通过编写和调试 kill 系统调用，掌握如何通过系统调用控制进程的生命周期以及在进程退出时释放系统资源

35.2 实验内容

本实验旨在实现一个自定义的 kill 系统调用，功能是使父进程能够终结其所有后代进程。通过遍历父进程的所有子进程，并递归终结每一个子进程及其后代，最终实现终结整个进程树。

35.3 参考资料

- 《操作系统设计与实现——基于 LoongArch 架构》第 6 章。
- MaQueOS 代码 code6（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

35.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

35.5 实验原理

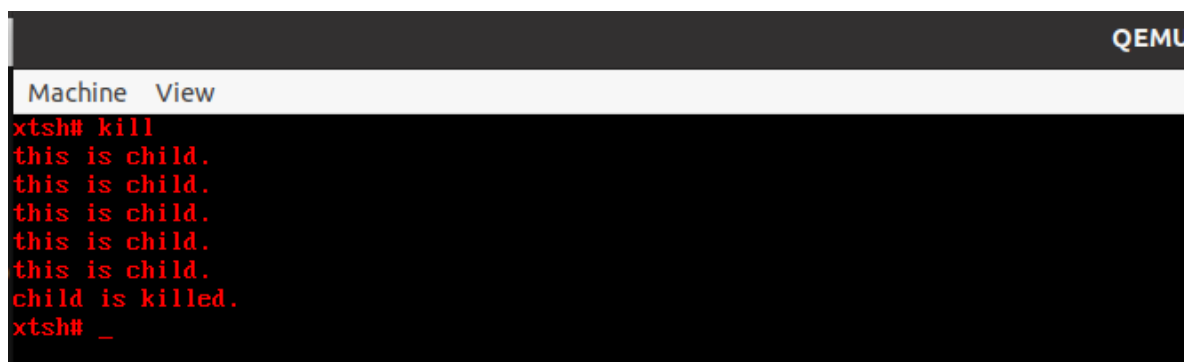
本实验实现了 kill 系统调用的核心功能，用于终止当前进程的所有子进程并释放相关资源。通过遍历进程表，找到当前进程的子进程，并清理其文件表中的资源。然后调用函数将子进程从进程队列中移除，最终实现对子进程的彻底终止，确保系统资源不被泄漏。

35.6 实验步骤

- 在 code12/kernel/proc/process.c 文件中添加了 sys_kill 函数：
 1. 定义两个循环变量 i 和 j。
 2. 遍历所有进程表项。
 3. 如果 process[i] 的父进程等于当前进程 current，说明当前进程存在一个子进程。
 - (a) 如果找到子进程，进一步清理其文件表。
 - (b) 调用 free_process 函数释放子进程的资源。
 4. 返回 0 表示所有子进程已被成功终止。
- 并在 code12/kernel/xtfs/bin 目录下添加了 kill.S 文件进行验证。
 1. 父进程创建一个子进程 child，循环输出 “this is child”。
 2. 父进程等待任意输入，终结子进程。
 3. 父进程终结，输出 “child is killed”。

35.7 实验结果

在命令行中输入了 kill 命令，调用了 kill 系统调用，循环输出 “this is child”，在接收到任意输入后，终结子进程，输出 “child is killed”，结果如图35.1所示：



A screenshot of a QEMU terminal window. The title bar at the top right says "QEMU". Below the title bar is a tab labeled "Machine View". The terminal content shows a series of red text lines on a black background. It starts with "xtsh# kill", followed by five lines of "this is child.", then "child is killed.", and finally "xtsh# _".

```
xtsh# kill
this is child.
this is child.
this is child.
this is child.
this is child.
child is killed.
xtsh# _
```

图 35.1: kill 系统调用结果图

实验三十六 统计进程运行过程中时钟中断的次数

36.1 实验目的

1. 熟练掌握 MaQueOS 进程管理，特别是对进程描述符的修改。
2. 加深对 MaQueOS 中断处理流程的理解。

36.2 实验内容

系统产生时钟中断后，在时钟中断处理程序中，统计进程在用户态或者内核态下发生时钟中断的次数。

36.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 5 章。
- MaQueOS 代码 code5（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

36.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

36.5 实验原理

在单个处理器中运行多个进程时，需要在内核态下进程切换操作。进程在运行过程中，会运行在两个处理器状态中：用户态和内核态。

本实验首先在进程描述符中增加两个计数器：用户态时钟中断次数计数器和内核态时钟中断次数计数器。然后在进程 0 或进程 1 运行期间发生时钟中断时，中断次数计数器会通过累加操作进行计数。

36.6 实验步骤

- 在 `code5/kernel/include/xtos.h` 中的进程描述符中增加两个时钟中断次数计数器字段：
 1. `count_user`：用户态时钟中断次数计数器。
 2. `count_kernel`：内核态时钟中断次数计数器。
- 修改 `code5/kernel/excp/exception.c` 文件中负责时钟中断处理的 `do_timer` 函数：
 1. 查看 `PRMD` 寄存器的值，判断时钟中断发生在用户态还是内核态。若发生在用户态，则对用户态时钟中断次数计数器 `count_user` 进行加 1 操作；若发生在内核态，则对内核态时钟中断次数计数器 `count_kernel` 进行加 1 操作。
 2. 根据进程 `pid` 的值，在显示器上打印进程 0 和进程 1 在用户态和内核态下发生时钟中断次数的数量。

36.7 实验结果

如图36.1所示，在显示器中显示进程 0 和进程 1 在用户态和内核态下发生时钟中断次数的数量。



The image shows a QEMU Machine View window with a black background and red text. It displays the results of clock interrupt statistics for two processes, proc0 and proc1. Each process has two counters: count_user and count_kernel. The values for count_user are consistently 0x0000000000000001 for all entries. The values for count_kernel increase sequentially for each process, starting from 0x0000000000000000 and ending at 0x0000000000000004 for proc0 and 0x0000000000000007 for proc1.

```
Machine View
proc0:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000000
proc0:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000001
proc0:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000002
proc0:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000003
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000001
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000002
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000003
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000004
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000005
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000006
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000007
proc1:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000008
proc0:
count_user: 0x0000000000000001
count_kernel: 0x0000000000000004
proc0:
```

图 36.1: 时钟中断次数统计结果

实验三十七 创建硬盘镜像文件

37.1 实验目的

1. 掌握硬盘镜像文件创建过程。
2. 熟悉有硬盘镜像文件的 MaQueOS 实验环境。

37.2 实验内容

创建大小为 1MB 的硬盘镜像文件存放在 run 文件夹下，将 I LOVE LoongArch# 内容写入该镜像文件的 0 号数据块中，并在屏幕显示。

37.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 7 章。
- MaQueOS 代码 code7（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

37.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

37.5 实验原理

硬盘镜像文件是将硬盘或者分区的数据完整地写入到一个文件中，这个文件不仅包含用户数据，还包含操作的各种设置。在 MaQueOS 中通过调用读/写硬盘的临时

系统调用，实现对硬盘中数据块的读写操作。

37.6 实验步骤

- 创建 LoongArch_Img.sh 文件：
 1. 使用 echo 命令输出字符串 “I Love LoongArch#”，将该输出写入 temp 临时文件中。
 2. 使用 dd 命令，通过从/dev/zero 读取数据填充，创建一个名为 xtfs.img 镜像文件，大小为 1MB。
 3. 使用 dd 命令，将 temp 文件中的内容追加到 xtfs.img 文件的开始部分（0 号数据块）。
 4. 使用 mv 命令，将 xtfs.img 移动到上一级目录中的 run 目录下。
 5. 使用 rm 命令，删除临时文件 temp。
- 修改 run.sh 文件的代码。
 1. 使用 cd 命令，进入上一级目录下的 xtfs 目录下。
 2. 在 xtfs 目录下运行 LoongArch_img.sh。
- 运行 run.sh，实现硬盘镜像创建与 0 号数据块内容显示。

37.7 实验结果

如图37.1所示，在运行 run.sh 脚本文件后，在显示器中显示硬盘镜像文件中 0 号数据块中的内容 “I Love LoongArch#”。



图 37.1: 硬盘镜像文件 0 号数据块中的内容

实验三十八 设置字符背景颜色

38.1 实验目的

1. 通过编写操作系统内核代码，实现在显示器上显示字符时可以设置背景颜色的功能。
2. 了解显示器的显示原理，以及如何编写简单的显示器驱动程序来显示字符。

38.2 实验内容

为显示器上显示的字符设置背景颜色。

38.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 1 章。
- MaQueOS 代码 code1（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

38.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

38.5 实验原理

在循环遍历字模时，根据字模中的当前位的值，通过设置对应像素在显存中的 (B, G, R) 对应的字节的值，分别设置字符显示和背景颜色。

38.6 实验步骤

- 修改 `code1/kernel/drv/console.c` 文件中的 `write_char` 函数：
 1. 获取字符的起始像素在显存中的起始地址。
 2. 按行循环遍历字模：
 - (a) 若字模中的当前位为 1，则将对应该像素在显存中的 (B, G, R) 对应的字节分别设置为 (255, 0, 0)，表示将该像素绘制为蓝色。
 - (b) 若字模中的当前位为 0，则将对应该像素在显存中的 (B, G, R) 对应的字节分别设置为 (255, 255, 255)，表示将该像素绘制为白色。

38.7 实验结果

实验结果如图38.1所示，字符串的背景颜色被修改为白色。



图 38.1: 实验结果

实验三十九 基于位图的物理页分配算法优化

39.1 实验目的

1. 探究如何提高操作系统的内存管理性能，并且比较不同的物理页分配算法在性能上的优劣。
2. 深入地理解内存管理的相关原理，加深对操作系统内核的理解。
3. 研究不同的内存分配策略，提高内存分配速度。

39.2 实验内容

优化 MaQue 的物理页分配算法，并测试优化后的分配时间比优化前减少了多少。

39.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 3.1 节。
- MaQueOS 代码 code3（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

39.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

39.5 实验原理

首先，对 `get_page` 进行优化，具体地，将 `mem_map` 数组改为一个位图，用一个 `unsigned long` 类型的变量表示 32 个物理内存页的占用情况。这样可以节省内存空间，并提高查找空闲页的效率。然后，记录空闲页的起始位置，下一次分配时从该位置开始扫描。最后，通过多次调用 `get_page` 函数，计算申请内存的时间，对比时间优化的程度。

39.6 实验步骤

- 定义全局变量：
 1. `timer`：通过时钟中断不断累加值来记录时间。
 2. `mem_map`：表示物理内存页的占用情况的位图。
 3. `next_free_page`：用于记录下一个空闲页的起始位置。
- 修改 `code3/kernel/init/main.c` 文件中的 `main` 函数：
 1. 在 `int_on` 函数返回后，打印全局变量 `timer` 的值。
 2. 循环调用 10000 次 `get_page` 函数后，再次打印全局变量 `timer` 的值。
- 修改 `code3/kernel/mm/memory.c` 文件中的 `get_page` 函数：
 1. 从 `next_free_page` 开始扫描 `mem_map` 位图。
 2. 查找未被占用的页后，标记该页已被占用。
 3. 清零该空闲页的内容。
 4. 更新下一个空闲页的起始位置 `next_free_page`。
- 修改 `code3/kernel/excp/exception.c` 文件中的 `timer_interrupt` 函数：
 1. 对全局变量 `timer` 进行加 1 操作。
- 修改 `code3/kernel/excp/exception.c` 文件中的 `excp_init` 函数：
 1. 修改定时器每隔 1ms 中断一次。

39.7 实验结果

图39.1和图39.2分别显示了对内存分配算法进行优化前后，物理页分配时间的对比情况。如图所示，基于位图的物理页分配算法所用的时间明显少于 MaQueOS 的物理页分配算法所用的时间。



图 39.1: MaQueOS 的物理页分配算法所用的时间



图 39.2: 基于位图的物理页分配算法所用的时间

实验四十 支持任意大小的共享内存

40.1 实验目的

1. 深入地理解基于共享内存的进程间通信机制。
2. 掌握系统调用的多参数调用原理。
3. 熟悉 MaQueOS 进程地址空间的布局。

40.2 实验内容

突破 MaQueOS 仅支持 4KB 大小的共享内存的限制。在 MaQueOS 中实现对任意大小的共享内存的支持。

40.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 11 章。
- MaQueOS 代码 code11（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

40.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

40.5 实验原理

首先，修改 `shmem` 数据结构，使共享内存的大小支持多个物理页面。然后，为 `sys_shmem` 系统调用增加一个用于指定物理页面数量的参数。最后，在应用程序中调用支持多个物理页面的 `sys_shmem` 系统调用，验证结果的正确性。

40.6 实验步骤

- 修改 `shmem` 数据结构：
 1. 增加表示共享内存大小的字段 `nr_pages`。
 2. 将 `mem` 字段扩展为数组，用于存放所有物理页面的起始地址。
- 修改 `code11/kernel/proc/ipc.c` 文件中的 `sys_shmem` 函数：
 1. 遍历 `shmem_table` 数组中的每个共享页，对共享页的名字和 `name` 参数进行匹配，若匹配成功，则将该共享页的使用计数加 1 后结束循环。
 2. 若匹配失败，则再次遍历 `shmem_table` 数组中的每个共享页，找到空闲共享页后，初始化该共享页对应 `shmem` 数据结构的字段：
 - `count`：将使用计数设置为 1。
 - `nr_pages`：初始化为共享内存占用的物理页面的数量。
 - `mem`：调用 `get_page` 函数，申请指定数量的空闲物理页面。
 - `name`：初始化为该共享页的名字。
 3. 将该虚拟页 `vpage` 的起始地址存放到用户态下的 `u_vaddr` 变量中。
 4. 调用 `share_page` 函数，将所有共享页设置为共享状态。
 5. 调用 `put_page` 函数，为所有共享页建立映射。
- 修改 `code11/xtfs/bin/shmem.S` 文件中 `shmem` 程序：
 1. 在父进程中，调用 `shmem` 系统调用，申请两个物理页面作为共享内存。
 2. 获取共享内存存在父进程虚拟地址。
 3. 将字符写入共享内存的第二页的起始位置。

4. 在子进程中，进行同样的操作
5. 最后在父进程中，前后调用两次 output 系统调用，对父子进程分别写入共享内存的内容打印到显示器，以验证共享内存的有效性。

40.7 实验结果

如图40.1所示，输出结果中的‘31’分别表示：父进程往 plane 共享内存中写入的字符‘3’，和子进程重覆盖写的字符‘1’。



```
Machine View
xtsh# shmem
31
xtsh# _
```

图 40.1: 实验结果

实验四十一 飞机大战之屏幕刷新

41.1 实验目的

1. 掌握系统调用创建过程。
2. 加深对 MaQueOS 显示器驱动的理解。

41.2 实验内容

实现“飞机大战”应用程序的主程序；通过创建显存刷新系统调用，实现屏幕刷新功能。

41.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 1 章。
- MaQueOS 代码 code1（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

41.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

41.5 实验原理

首先，在“飞机大战”应用程序的主进程 pwar 通过调用 fork 系统调用创建 refresh 子进程。然后，在主进程 pwar 调用 pause 系统调用将其挂起。在 refresh 子进程中，

通过调用 refresh 系统调用将缓冲区的内容刷新到显示器的显存中。

41.6 实验步骤

- 创建 refresh 系统调用：
 1. 在 kernel/drv/console.c 文件中添加 ys_refresh 函数：
 - 使用双层循环遍历显示器中所有 8000 (160*50) 个字符。
 - 针对每个字符, 首先调用 erase_char 函数, 然后调用 write_char 函数, 将需要显示的内容刷新到显存中。
 2. 将 sys_refresh 添加到 syscalls 数组中。
- 在 pwar.S 汇编文件中, 编写子程序 refresh:
 1. 调用 refresh 系统调用, 将缓冲区 buffer 中的内容刷新到显示器的显存中。
 2. 调用 timer 系统调用, 实现周期刷新显示器的功能。
 3. 初始化缓冲区 buffer 的内容。
- 在 pwar.S 汇编文件中, 编写主程序:
 1. 调用 fork 系统调用创建 refresh 子进程。
 2. refresh 子进程运行子程序 refresh。
 3. 主进程 pwar 调用 pause 系统调用。

41.7 实验结果

在 shell 中运行“飞机大战”应用程序后, 显示如图41.1所示界面。



图 41.1: 飞机大战

实验四十二 飞机大战之进程间通信

42.1 实验目的

1. 熟练运用 MaQueOS 的进程间通信机制。
2. 掌握多个汇编文件的编译和链接过程。

42.2 实验内容

通过调用创建共享内存系统调用，为两个进程申请共享内存，并实现“飞机大战”应用程序各个进程间的通信。

42.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 11 章。
- MaQueOS 代码 code11（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

42.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

42.5 实验原理

本实验在实验“飞机大战之屏幕刷新”的基础上进行修改。首先，在“飞机大战”应用程序的主进程 pwar 中，通过调用 shmem 系统调用，申请创建名为 plane 的共享

内存空间，作为显存缓冲区。然后，在主进程 pwar 中将缓冲区写满字符 ‘*’ 后，调用 fork 系统调用，创建 refresh 子进程。最后，在 refresh 子进程中，再次调用 shmem 系统调用，与主进程 pwar 共享 plane 内存空间，并通过调用 refresh 系统调用，将写满字符 ‘*’ 的缓冲区的内容刷新到显示器的显存中。

42.6 实验步骤

- 修改主程序 pwar.S:
 1. 调用 shmem 系统调用，创建名为 plane 的共享内存空间。
 2. 循环向共享内存空间写满字符 ‘*’。
 3. 调用 fork 系统调用，创建 refresh 子进程。
- 修改子程序 refresh.S:
 1. 调用 shmem 系统调用，与主进程 pwar 共享 plane 内存空间。
 2. 周期调用 refresh 系统调用，将共享内存空间中的内容，定期刷新到显示器的显存中。
- 编译链接：
 1. 在 xtfs 目录下的 init_img.sh 脚本文件中，首先单独编译 pwar.S 和 refresh.S 两个汇编文件，然后进行链接操作。

42.7 实验结果

在 shell 中运行 “飞机大战” 应用程序后，显示如图42.1所示界面。



图 42.1: 飞机大战

实验四十三 飞机大战之战机篇

43.1 实验目的

1. 掌握在汇编场景中，在调用函数时对寄存器的处理。
2. 加深对 MaQueOS 键盘驱动的理解。

43.2 实验内容

实现“飞机大战”应用程序中的 plane 子程序，实现可以通过用户的键盘输入，控制飞机的移动。

43.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 3 章。
- MaQueOS 代码 code3（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

43.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

43.5 实验原理

本实验在实验“飞机大战之进程间通信”的基础上进行修改。首先，在“飞机大战”应用程序的主进程 pwar 中，调用 fork 系统调用，创建 plane 子进程。然后，在

plane 子进程中，完成在显存缓冲区中飞机的移动功能。

43.6 实验步骤

- 修改主程序 pwar.S:
 1. 调用 fork 系统调用，创建 plane 子进程。
- 创建子程序 plane.S:
 1. 调用 shmem 系统调用，与主进程 pwar 和子进程 refresh 共享 plane 内存空间。
 2. 定义用于存放飞机位置的变量 plane_index，并将其设置为显示器最下方的中间位置。
 3. 循环调用 input 系统调用，检测按键输入信息。若是 ‘a’，则修改变量 plane_index 的值，使飞机向左移动；若是 ‘s’，则修改变量 plane_index 的值，使飞机向右移动。
 4. 飞机的移动需要飞机显示和清除函数的支持，这两个函数都根据变量 plane_index 中保存的飞机位置，在显示器中显示和清除飞机。
 5. 在编写飞机显示和清除函数的时候，需要注意的是，在函数开始和结束位置处，需要对函数运行过程中使用到的寄存器进行入栈保存和出栈恢复操作。

43.7 实验结果

如动画43.7所示。在 shell 中运行“飞机大战”应用程序后，首先在显示器的底部显示红色飞机，然后在分别按下 ‘a’ 和 ‘b’ 键后，飞机左右移动。

图 43.1: 飞机大战

实验四十四 飞机大战之子弹篇

44.1 实验目的

1. 熟练掌握 MaQueOS 共享内存机制的应用。
2. 加深对 MaQueOS 进程的创建和终止流程的理解。

44.2 实验内容

实现“飞机大战”应用程序中的 bullet 子程序，随着飞机移动，完成子弹的创建和移动功能。

44.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 6 章。
- MaQueOS 代码 code6（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

44.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

44.5 实验原理

本实验在实验“飞机大战之战机篇”的基础上进行修改。首先，在子进程 plane 中，调用 fork 系统调用，创建 bullet_create 子进程。然后，在 bullet_create 子进程

中，定时创建一个 bullet 子进程。最后，在 bullet 子进程中，实现子弹的创建和移动功能。

44.6 实验步骤

- 修改子程序 plane.S:
 1. 在飞机显示函数中，当飞机位置发生移动后，在共享内存中更新子弹发射位置。
 2. 调用 fork 系统调用，创建子弹创建子进程 bullet_create。
- 创建子程序 bullet_create.S:
 1. 调用 timer 系统调用，实现每隔固定时间，创建 1 个 bullet 子进程。
 2. 调用 pause 系统调用，将自己挂起，当 bullet 进程终止运行后，释放它们占用的资源。
- 创建子程序 bullet.S:
 1. 调用 shmем 系统调用，与主进程 pwar、子进程 plane 和子进程 refresh 共享 plane 内存空间。
 2. 定义用于存放子弹位置的变量 bullet_index，并将其设置为在 plane 子进程中更新的子弹发射位置。
 3. 在子弹移动过程中，变量 bullet_index 保存子弹的实时位置。
 4. 调用 timer 系统调用，每隔固定时间，使子弹向上移动一行。
 5. 子弹的向上移动需要子弹显示和清除函数的支持，这两个函数都根据变量 bullet_index 中保存的子弹位置，在显示器中显示和清除子弹。
 6. 在子弹向上移动的过程中，需要判断子弹是否移动出显示器：若未移出，则将子弹在显示器中向上移动一行；若移出，则在显示器中擦除子弹后，进程终止 bullet 子进程的运行。

44.7 实验结果

如动画44.7所示。在 shell 中运行“飞机大战”应用程序后，显示在显示器的底部的红色飞机会自动发送子弹，当按下‘a’和‘b’键飞机发生移动时，子弹发送位置也同步发生变化。

图 44.1: 飞机大战

实验四十五 飞机大战之敌机篇

45.1 实验目的

1. 熟练掌握 MaQueOS 共享内存机制的应用。
2. 加深对 MaQueOS 进程的创建和终止流程的理解。

45.2 实验内容

实现“飞机大战”应用程序中的 enemy 子程序，完成敌机的创建和移动功能。并实现子弹和敌机相遇后，两者同时消失的功能。

45.3 参考资料

- 《操作系统设计与实现：基于 LoongArch 架构》第 6 章。
- MaQueOS 代码 code6（仓库地址：<https://gitee.com/dslab-lzu/maqueos>）。

45.4 实验环境

- 操作系统：ubuntu20.04
- 虚拟机：qemu-system-loongarch64

45.5 实验原理

本实验在实验“飞机大战之子弹篇”的基础上进行修改。首先，在子进程 plane 中，调用 fork 系统调用，创建 enemy_create 子进程。然后，在 enemy_create 子进

程中，定时创建一个 enemy 子进程。最后，在 enemy 子进程中，实现敌机的创建和移动功能。在子弹向上和敌机向下移动的过程中，若两者相遇，则同时消失。

45.6 实验步骤

- 修改子程序 plane.S:

1. 调用 fork 系统调用，创建敌机创建子进程 enemy_create。

- 创建子程序 enemy_create.S:

1. 调用 timer 系统调用，实现每隔固定时间，创建 1 个 enemy 子进程。
2. 调用 pause 系统调用，将自己挂起，当 enemy 进程终止运行后，释放它们占用的资源。

- 创建子程序 bullet.S:

1. 调用 shmем 系统调用，与主进程 pwar、子进程 plane、子进程 bullet 和子进程 refresh 共享 plane 内存空间。
2. 定义用于存放敌机位置的变量 enemy_index，并将其设置为 160 以内的随机数。
3. 在敌机移动过程中，变量 enemy_index 保存敌机的实时位置。
4. 调用 timer 系统调用，每隔固定时间，使敌机向下移动一行。
5. 敌机的向下移动需要敌机显示和清除函数的支持，这两个函数都根据变量 enemy_index 中保存的敌机位置，在显示器中显示和清除敌机。
6. 在敌机向下移动的过程中，需要判断敌机的位置：
 - 若敌机未与子弹或者飞机相遇，或者未移动出显示器，则将敌机在显示器中向下移动一行。
 - 若敌机与子弹或者飞机相遇，或者移动出显示器，则在显示器中擦除敌机后，进程终止运行。

- 修改子程序 bullet.S:

1. 在子弹向上移动的过程中，需要判断子弹是否与敌机相遇：若未相遇，则将子弹在显示器中向上移动一行；若相遇，则在显示器中擦除子弹后，进程终止 bullet 子进程的运行。

45.7 实验结果

如动画45.7所示。在 shell 中运行“飞机大战”应用程序后，显示在显示器的底部的红色飞机会自动发送子弹，当按下‘a’和‘b’键飞机发生移动时，子弹发送位置也同步发生变化。与此同时，在最上方的随机位置出现一架敌机，在子弹向上和敌机向下移动的过程中，若两者相遇，则同时消失。

图 45.1: 飞机大战