

深圳大学

操作系统原型 ——xv6 分析与实验

(V 1.0)

罗秋明 林沛昭

第1章 xv6 安装使用

xv6 是一个教学操作系统，它是对 Dennis Ritchie 和 Ken Thompson 的 UNIX Version 6 (v6) 的简单实现，但并不严格遵循 v6 的结构和风格。xv6 用 ANSI C 实现，现有 x86 和 RISC-V 版本，麻省理工大学的网站 (<http://pdos.csail.mit.edu/6.828/2011/xv6.html>) 上有 xv6 来龙去脉的详细介绍。

在我们后面的操作中，将使用 x86 仿真器来观察 xv6 的运行过程，因此我们先安装 QEMU（当然也可以用 bochs），然后再安装 xv6 代码。需要注意的是 xv6 所生成的代码是 32 位代码。

1.1. 运行于 QEMU 的 xv6

我们以 Ubuntu20 64 位 Linux 为例说明 xv6 的安装过程。在该操作系统环境中，以 QEMU 仿真系统来运行 xv6 的。QEMU 是一套由法布里斯贝拉 (Fabrice Bellard) 所编写的处理器仿真软件，是在 GNU/Linux 平台上使用广泛的 GPL 开源软件。xv6 操作系统可以运行于该仿真系统上。

1.1.1. Ubuntu20+QEMU+xv6

本节我们使用 Ubuntu 20.04.5 的 ISO 镜像 (ubuntu-20.04.5-desktop-amd64) 来安装开发环境的操作系统，安装期间选择不更新其他软件包也不安装第三方软件。Ubuntu 安装结束后，输入用户名和密码后即可完成登陆。如果要修改 root 账户的密码，此时可以执行 `sudo passwd root` 然后输入新密码就可以了。如果需要使用终端，则在桌面空白处按下 `Ctrl+ALT+T` 可以弹出终端。使用 `Ctrl+ALT+F1~F6` 可以在多个终端之间切换。

下面展示如何在 Ubuntu 上安装 xv6 及相关软件。

QEMU 安装

前面提到过，QEMU 是一套由法布里斯贝拉 (Fabrice Bellard) 所编写的处理器仿真软件，是在 GNU/Linux 平台上使用广泛的 GPL 开源软件。以管理员身份，在终端窗口直接执行 `apt-get install qemu-system` 命令即可完成安装。

配置交叉编译工具链

为了能编译在 LoongArch 下运行的 xv6 内核，需要下载交叉编译工具链，在 shell 中执行 `wget https://github.com/loongson/build-tools/releases/download/2022.05.29/loongarch64-clfs-5.0-cross-tools-gcc-full.tar.xz` 下载工具链包 `loongarch64-clfs-5.0-cross-tools-gcc-full.tar.xz`，然后用 `sudo tar -vxf loongarch64-clfs-5.0-cross-tools-gcc-full.tar.xz -C /opt` 命令解压缩。

下载完工具链需要配置交叉编译工具的环境变量，可通过修改 `/etc/profile` 文件实现，在该文件中添加如下几条语句：

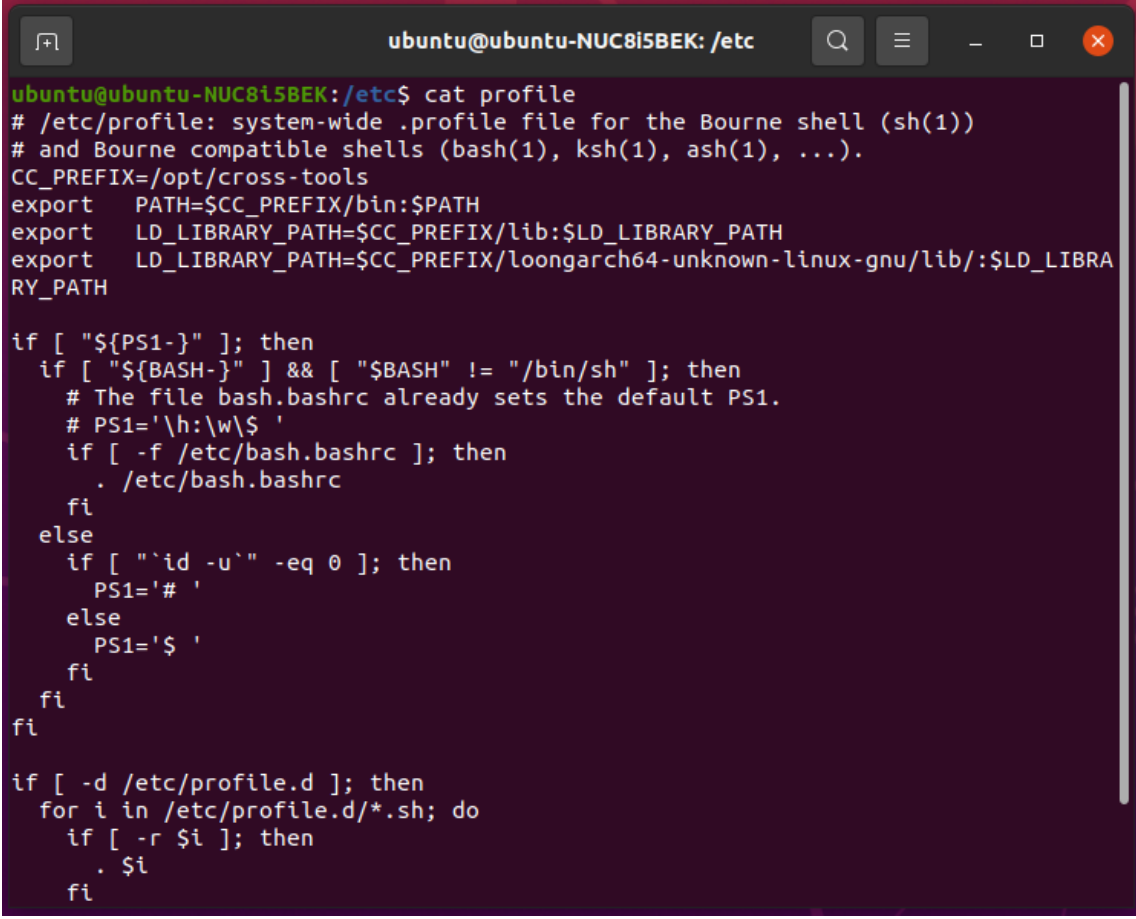
```
CC_PREFIX=/opt/cross-tools
```

```
export PATH=$CC_PREFIX/bin:$PATH
```

```
export LD_LIBRARY_PATH=$CC_PREFIX/lib:$LD_LIBRARY_PATH
```

```
export LD_LIBRARY_PATH=$CC_PREFIX/loongarch64-unknown-linux-gnu/lib/:$LD_LIBRARY_PATH
```

添加完后可以用 cat 命令查看是否添加成功，如图 1-1 所示。

A terminal window titled 'ubuntu@ubuntu-NUC8i5BEK: /etc' showing the output of the 'cat profile' command. The output displays the system-wide .profile file for the Bourne shell, including environment variable exports for CC_PREFIX, PATH, and LD_LIBRARY_PATH, and shell initialization logic for PS1 and sourcing /etc/bash.bashrc.

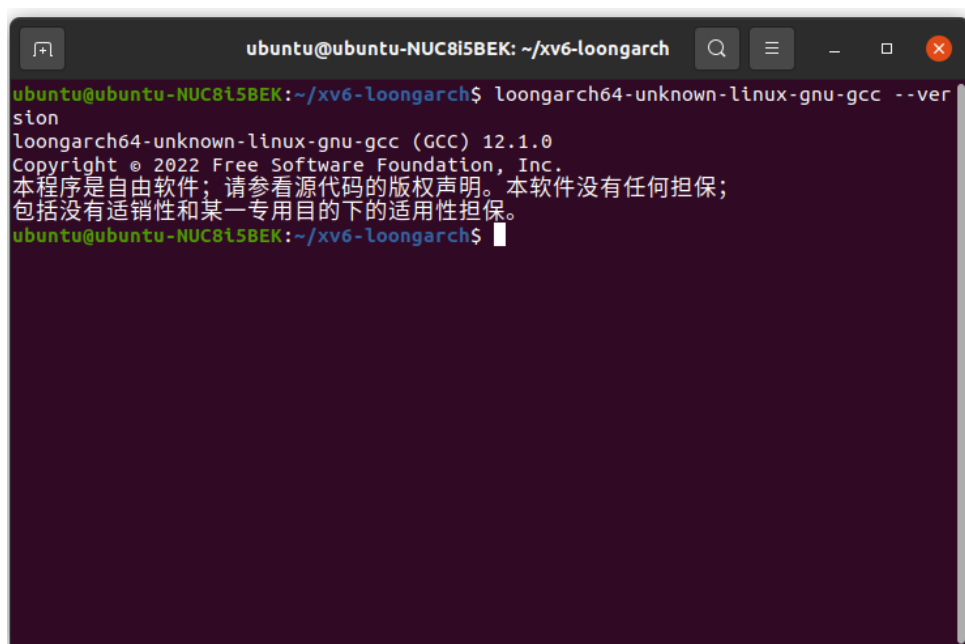
```
ubuntu@ubuntu-NUC8i5BEK:/etc$ cat profile
# /etc/profile: system-wide .profile file for the Bourne shell (sh(1))
# and Bourne compatible shells (bash(1), ksh(1), ash(1), ...).
CC_PREFIX=/opt/cross-tools
export  PATH=$CC_PREFIX/bin:$PATH
export  LD_LIBRARY_PATH=$CC_PREFIX/lib:$LD_LIBRARY_PATH
export  LD_LIBRARY_PATH=$CC_PREFIX/loongarch64-unknown-linux-gnu/lib/:$LD_LIBRARY_PATH

if [ "${PS1-}" ]; then
  if [ "${BASH-}" ] && [ "$BASH" != "/bin/sh" ]; then
    # The file bash.bashrc already sets the default PS1.
    # PS1='\h:\w\$ '
    if [ -f /etc/bash.bashrc ]; then
      . /etc/bash.bashrc
    fi
  else
    if [ "`id -u`" -eq 0 ]; then
      PS1='# '
    else
      PS1='$ '
    fi
  fi
fi

if [ -d /etc/profile.d ]; then
  for i in /etc/profile.d/*.sh; do
    if [ -r $i ]; then
      . $i
    fi
  done
fi
```

图 1-1 用 cat 命令查看/etc/profile 文件

通过命令 `loongarch64-unknown-linux-gnu-gcc --version` 可以检验是否设置成功，看到如图 1-2 的提示则说明已正确设置。



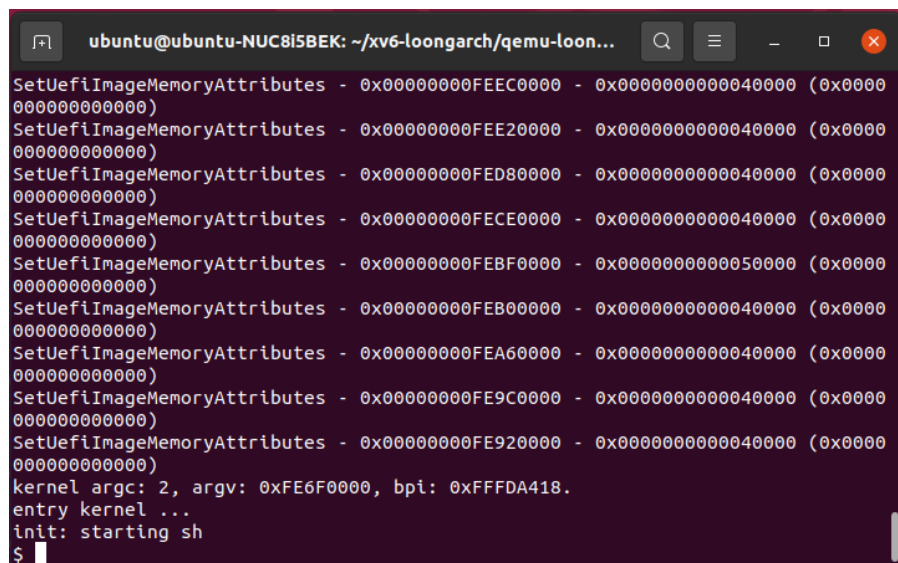
```
ubuntu@ubuntu-NUC8i5BEK: ~/xv6-loongarch
ubuntu@ubuntu-NUC8i5BEK:~/xv6-loongarch$ loongarch64-unknown-linux-gnu-gcc --version
loongarch64-unknown-linux-gnu-gcc (GCC) 12.1.0
Copyright © 2022 Free Software Foundation, Inc.
本程序是自由软件；请参看源代码的版权声明。本软件没有任何担保；
包括没有适销性和某一专用目的下的适用性担保。
ubuntu@ubuntu-NUC8i5BEK:~/xv6-loongarch$
```

图 1-2 检验工具链环境变量设置

xv6 安装

之后在 github 网站 <https://github.com/SKT-CPUOS/xv6-loongarch-exp> 下载 xv6 源码包 xv6-loongarch-exp-main.zip，然后用 `unzip xv6-loongarch-exp-main.zip` 命令解压缩，最后进入 xv6 目录中执行 `make all` 即可完成编译。如果系统还没有安装 `make` 则需要用 `sudo apt install make`。类似地，如果 `make` 是提示还未安装 `gcc` 则需要执行 `sudo apt install gcc` 完成相应的安装。

此时，在 xv6 目录的 `qemu-loongarch-runenv` 下执行 `./run_loongarch.sh -k ../kernel/kernel` 即可启动仿真运行，如图 1-所示（在执行这一步时可能会报错显示缺少 `libSDL2-2.0.so.0` 和 `libfuse3.so.3` 文件，可以通过执行 `sudo apt install libSDL2-2.0-0 libfuse3-3` 命令安装软件包解决，若仍缺少其他文件请根据具体情况安装缺少的软件包）。需要退出仿真时可以先按下“Ctrl+a”松开后再按“x”来结束仿真。



```
ubuntu@ubuntu-NUC8i5BEK: ~/xv6-loongarch/qemu-loon...
SetUefiImageMemoryAttributes - 0x00000000FEEC0000 - 0x0000000000040000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEE20000 - 0x0000000000040000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FED80000 - 0x0000000000040000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FECE0000 - 0x0000000000040000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEBF0000 - 0x0000000000050000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEB00000 - 0x0000000000040000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEA60000 - 0x0000000000040000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FE9C0000 - 0x0000000000040000 (0x0000
000000000000)
SetUefiImageMemoryAttributes - 0x00000000FE920000 - 0x0000000000040000 (0x0000
000000000000)
kernel argc: 2, argv: 0xFE6F0000, bpi: 0xFFFFDA418.
entry kernel ...
init: starting sh
$
```

图 1-3 Ubuntu20+QEMU+xv6 运行示意图

1.2. 调试观察

在学习 xv6 代码的过程中，我们需要不断地用 gdb 调试器观察代码的行为，因此读者需要有一点 gdb 常用命令的知识。如果读者已经学习过本系列的《Linux GNU C 程序观察》一书，已具备足够的背景知识。否则，可能需要自行了解 gdb 调试器基本命令的使用。

1.2.1. xv6 shell 命令

在 xv6 目录的 qemu-loongarch-runenv 下执行 ./run_loongarch.sh -k ../kernel/kernel 将启动 QEMU 仿真器并运行 xv6 操作系统，此时将在原来的 shell 文本窗口显示 xv6 的输出。如图 1-所示。

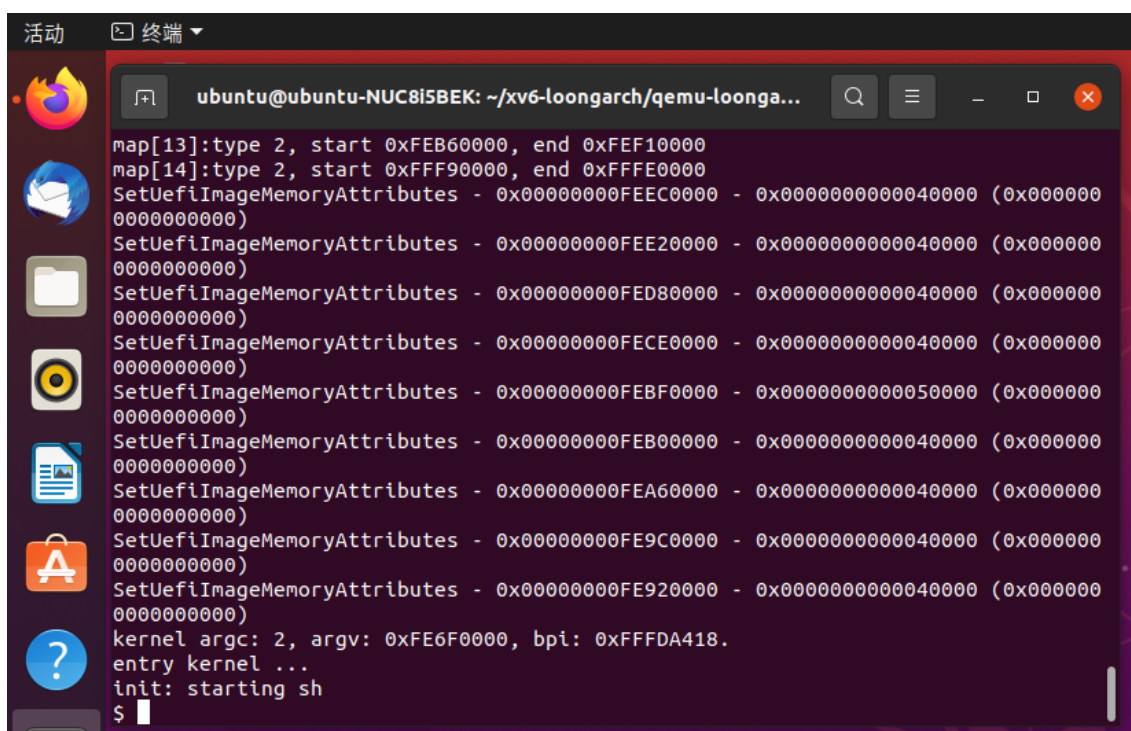


图 1-4 启动 QEMU 仿真器运行 xv6

此时在 xv6 shell 中执行 ls 命令，可以看到 xv6 文件系统中所有的可执行文件，如屏显 1-1 所示。

屏显 1-1 ls 列出所有磁盘文件

```

$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
cat        2 3 20120
echo       2 4 19056
forktest  2 5 11312
grep       2 6 22568
init       2 7 19808
kill       2 8 18944
ln         2 9 18832
ls         2 10 21928
mkdir     2 11 19048

```

```
rm          2 12 19032
sh          2 13 34688
stressfs    2 14 19920
usertests   2 15 127776
console     3 16 0
$
```

读者可以尝试用 `cat README` 查看 `README` 内容，或者用 `echo` 显示某一行消息等。甚至还可以尝试其他 Linux 常用功能，例如用管道将两个命令连接执行 `cat README | grep xv6`。Xv6 对目录操作的几个命令和 Linux 的命令差不多：`mkdir`、`rm` 和内部命令 `cd` 等。

如果在 shell 中按下 `Ctrl+P` 则会显示当前运行的进程信息，如屏显 1-2 所示。此时有两个进程，进程号为 1 的是 `init` 进程，当前处于 `sleep` 阻塞状态；编号为 2 的进程是 `sh` 进程，当前也是处于 `sleep` 阻塞状态。后面我们会看到，上面的信息实际上是通过内核函数 `procdump()` 打印出来的。

屏显 1-2 用 `Ctrl+P` 查看运行进程信息

```
$
$ 1 sleep  init
2 sleep  sh
```

1.2.2. QEMU+gdb 调试

GDB 工具安装

首先安装 `gdb` 工具，先在 shell 中执行 `sudo apt install texinfo bison flex libgmp-dev` 安装一些必要依赖，若编译时仍报错则再根据编译时的报错情况自行安装所需依赖。然后编译安装 `gdb` 工具，具体步骤为：执行 `git clone https://github.com/foxsen/binutils-gdb` 命令下载 `gdb` 源码包，进入 `binutils-gdb` 目录，执行 `git checkout loongarch-v2022-03-10`，然后用 `mkdir build` 命令创建 `build` 目录，在 `build` 目录下执行 `../configure --target=loongarch64-unknown-linux-gnu --prefix=/opt/gdb` 命令。最后用 `make` 命令编译，如果系统还没有安装 `g++` 则需要先用 `sudo apt install g++` 安装 `g++`，编译完成后再执行 `make install` 即可完成安装。

GDB 调试观察

用 `GDB` 调试 `QEMU` 时可以将调试目标分为两种，一种是用 `GDB` 调试由 `QEMU` 启动的虚拟机，即远程调试虚拟机系统内核，可以从虚拟机的 `bootloader` 开始调试虚拟机启动过程，另一种是调试 `QEMU` 本身的代码而不是虚拟机要运行的代码。我们这里需要调试的是 `xv6` 代码，而不是 `QEMU` 仿真器的代码。

当用 `gdb` 调试时涉及两个窗口，如图 1-所示。一个是启动 `QEMU` 的 `shell` 窗口（左上角），一个是运行 `gdb` 的 `shell` 窗口（右上角）。下面将详细讨论这两个窗口中运行的命令。

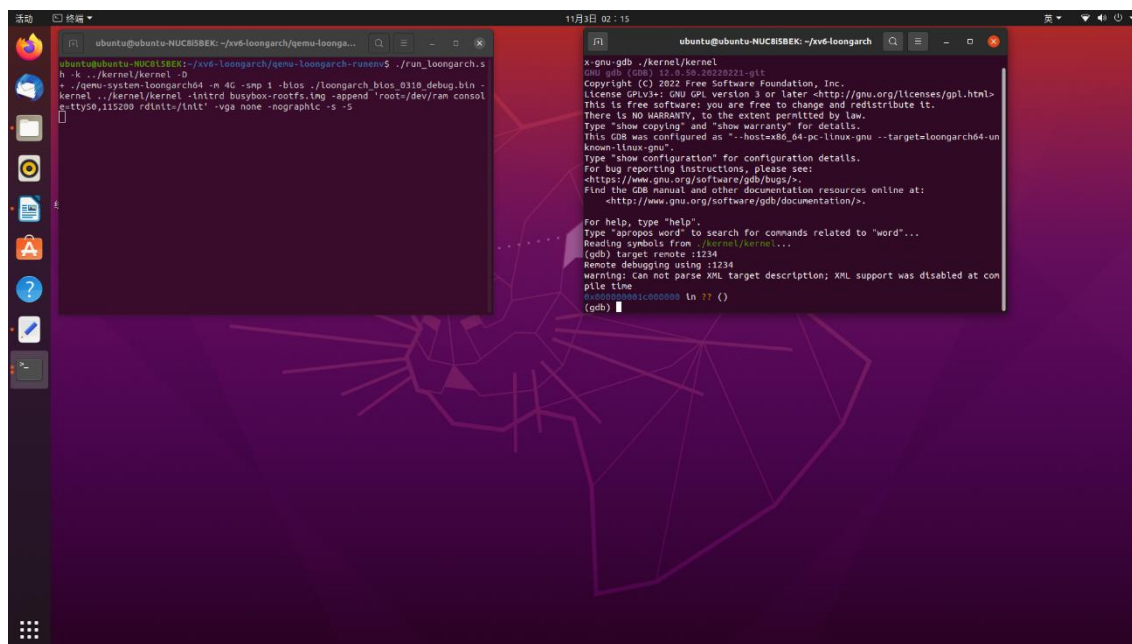


图 1-5 用 gdb 调试运行于 QEMU 环境的 xv6

首先，我们在一个终端 shell 中进入 xv6 源码目录下的 `qemu-loongarch-runenv` 并执行 `./run_loongarch.sh -k ../kernel/kernel -D` 启动调试模式，实际上是执行了调试服务器 `gdbserver` 的角色，默认监听 TCP::1234 端口，等待 gdb 客户端的接入，如屏显 1-所示。

屏显 1-3 启动 QEMU 调试模式

```
lqm@lqm-VirtualBox:~/xv6-loongarch-exp/qemu-loongarch-runenv$ ./run_loongarch.sh -k ../kernel/kernel -D
+ ./qemu-system-loongarch64 -m 4G -smp 1 -bios ./loongarch_bios_0310_debug.bin - kernel ../kernel/kernel -
initrd busybox-rootfs.img -append 'root=/dev/ram console=ttyS0,115200 rdinit=/init' -vgaS
```

我们在另外一个终端上启动 `gdb` 并连接到该主机的 1234 上。在 `xv6` 目录执行 `/opt/gdb/bin/loongarch64-unknown-linux-gnu-gdb ./kernel/kernel` 启动 `gdb` 对 `xv6` 内核 `kernel` 的调试，然后在 `gdb` 命令提示符下执行 `target remote :1234` 连接到 `xv6` 目标系统上，如屏显 1-所示。注意其中的被调试对象（`kernel` 文件）要在当前目录，如果不在，则需要用路径指出其位置。

屏显 1-4 gdb 客户端接入

```
lqm@lqm-VirtualBox:~/xv6-loongarch-exp$ /opt/gdb/bin/loongarch64-unknown-linux-gnu-gdb ./kernel/kernel
GNU gdb (GDB) 12.0.50.20220221-git
Copyright (C) 2022 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from ./kernel/kernel...
(gdb) target remote :1234
Remote debugging using :1234
warning: Can not parse XML target description; XML support was disabled at compile time
```

```
0x000000001c000000 in ?? ()  
(gdb)
```

此时读者用 `gdb` 的 `c` 命令，将开始执行 `xv6` 内核代码，并在 `xv6` 终端上看到系统启动并执行 `shell` 的界面，如屏显 1-所示。

屏显 1-5 `xv6` 启动运行

```
Entering C environment  
SecCoreStartupWithStack (0x1C010000, 0x90020000)  
(0x90010000, 0x10000)  
&SecCoreData.BootFirmwareVolumeBase=9001FF88 SecCoreData.BootFirmwareVolumeBase=1C010000  
&SecCoreData.BootFirmwareVolumeSize=9001FF90 SecCoreData.BootFirmwareVolumeSize=40000  
Find Pei EntryPoint=1C011240  
  
...省略，以节省篇幅  
  
SetUefiImageMemoryAttributes - 0x00000000FE920000 - 0x0000000000040000 (0x0000000000000000)  
kernel argc: 2, argv: 0xFE6F0000, bpi: 0xFFFDA418.  
entry kernel ...  
init: starting sh  
$
```

本书其余部分将使用 `QEMU+gdb` 的调试方式来观察 `xv6` 的运行。

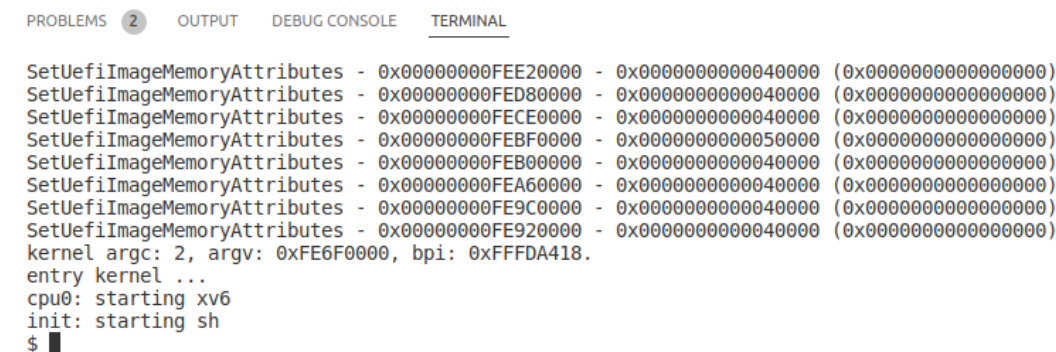
1.3. 小结

本章完成了 `xv6` 实验系统的建立，给出了 `Ubuntu20` 上的 `QEMU` 安装和 `xv6` 的安装过程。出于方便考虑，`QEMU` 的宿主操作系统 `Ubuntu20` 可以安装在 `VirtualBox` 或其他虚拟机上。然后简单展示了 `xv6 shell` 中执行磁盘上的外部命令，如何用 `gdb` 调试 `xv6` 代码的运行。

第2章 入门实验

在深入学习和修改内核代码之前，我们先来在一些外围的编程操作。学会如何添加新的应用程序、如何增加新的系统调用，以后才能做一些更加复杂的实验修改。本章实验的完整代码可以从 <https://github.com/SKT-CPUOS/xv6-loongarch-exp> 下载。

首先我们先来实施一个最简单的热身动作，修改 xv6 启动时的提示信息。打开 `kernel/main.c` 程序，在第 17 行开始的 `if` 语句块中任意位置添加一句 `printf("cpu0: starting xv6\n");`；修改启动提示信息，或者修改成其他你所希望的样子，如将它修改成图 2-1 所示的样子——标志着自己开始动手修改 xv6 操作系统了。



```

PROBLEMS 2 OUTPUT DEBUG CONSOLE TERMINAL
SetUefiImageMemoryAttributes - 0x00000000FEE20000 - 0x0000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FED80000 - 0x0000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FECE0000 - 0x0000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEBF0000 - 0x0000000000050000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEB00000 - 0x0000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEA60000 - 0x0000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FE9C0000 - 0x0000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FE920000 - 0x0000000000040000 (0x0000000000000000)
kernel argc: 2, argv: 0xFE6F0000, bpi: 0xFFFFDA418.
entry kernel ...
cpu0: starting xv6
init: starting sh
$ █

```

图 2-1 修改后的启动提示信息

2.1. 新增可执行程序

为了给 xv6 添加新的可执行文件，需要了解 xv6 磁盘文件系统是如何生成的，然后才能编写应用程序并出现在 xv6 的磁盘文件系统中。

2.1.1. 磁盘映像的生成

现在我们先来了解 xv6 磁盘文件系统上的可执行文件是怎么生成的，包括两步骤：

- (1) 生成各个应用程序。
- (2) 将应用程序构成文件系统映像。

首先在 `Makefile` 中有一个默认规则，那就是所有的 `*.c` 文件都需要通过默认的编译命令生成 `*.o` 文件。另外在 `Makefile` 中有一个规则用于指出可执行文件的生成，如代码 2-1 所示。该模式规则说明，对于所有的 `%o` 文件将结合 `$(ULIB)` 一起生成可执行文件 `_%`，比如说 `ls.o`（即依赖文件 `$^`）将链接生成 `_ls`（即输出目标 `$@`）。`-Ttext 0` 用于指出代码存放在 0 地址开始的地方，`-e main` 参数指出以 `main` 函数代码作为运行时的第一条指令，`-N` 参数用于指出 `data` 节与 `text` 节都是可读可写、不需要在页边界对齐。

代码 2-1 Makefile 中可执行文件的生成规则

```

1.      ULIB = ulib.o usys.o printf.o umalloc.o
2.
3.      _%: %.o $(ULIB)
4.          $(LD) $(LDFLAGS) -N -e main -Ttext 0 -o $@ $^
5.          $(OBJDUMP) -S $@ > $*.asm
6.          $(OBJDUMP) -t $@ | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$$/d' > $*.sym

```

代码 2-2 是 Makefile 中有关创建磁盘文件系统的部分。其中变量 UPROGS 包含了所有相关的可执行文件名。磁盘文件系统 fs.img 目标依赖于 UPROGS 变量，并且将它们和 README 文件一起通过 mkfs 程序转换成文件系统映像 fs.img。

代码 2-2 Makefile 中创建磁盘文件系统的部分

```

1.      UPROGS=\
2.          $U/_cat\
3.          $U/_echo\
4.          $U/_forktest\
5.          $U/_grep\
6.          $U/_init\
7.          $U/_kill\
8.          $U/_ln\
9.          $U/_ls\
10.         $U/_mkdir\
11.         $U/_rm\
12.         $U/_sh\
13.         $U/_stressfs\
14.         $U/_usertests\
15.         # $U/_grind\
16.         $U/_wc\
17.         $U/_zombie\
18.
19.      fs.img: mkfs/mkfs README $(UPROGS)
20.          mkfs/mkfs fs.img README $(UPROGS)
21.          xxd -i fs.img > kernel/ramdisk.h

```

2.1.2. 添加简单程序

因此我们在 xv6 源码的 user 目录下，编写一个程序作为我们为 xv6 增加的一个应用程序，如代码 2-3 所示。其中 types.h、stat.h 和 user.h 分别是 kernel 和 user 本目录中的头文件。程序运行结果是打印一行信息 “This is my own app!\n”。

代码 2-3 my-app.c

```

1.      #include "kernel/types.h"
2.      #include "kernel/stat.h"
3.      #include "user/user.h"
4.
5.      int
6.      main(int argc, char *argv[])
7.      {
8.          printf("This is my own app!\n");
9.          exit(0);
10.     }

```

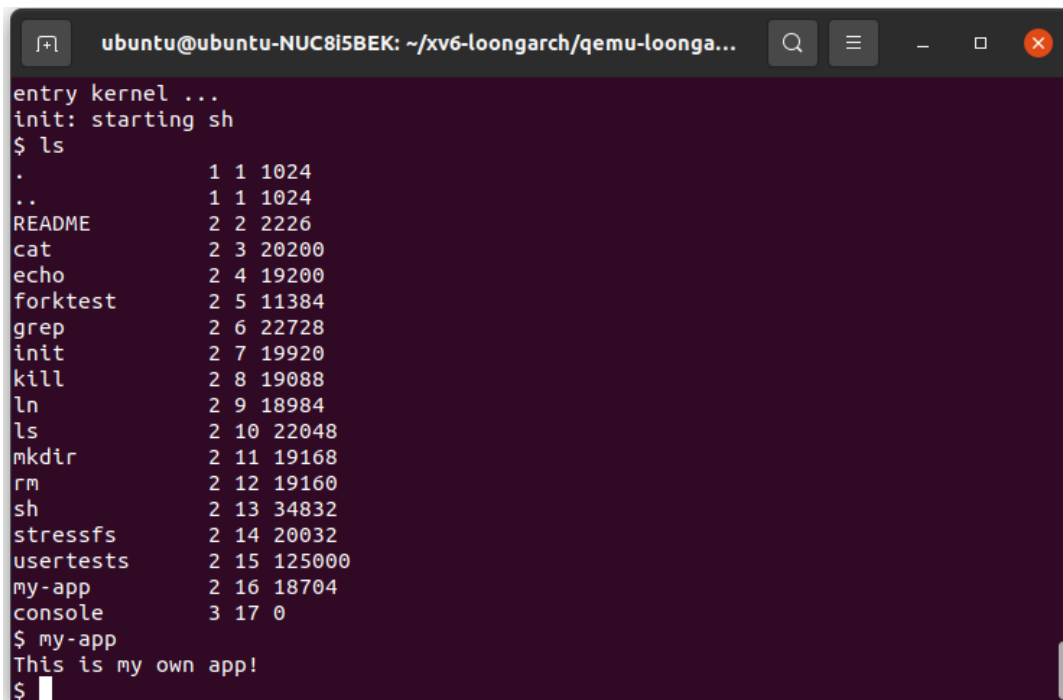
然后我们修改 Makefile 中的 UPROGS 变量，添加一个 \$U/_my-app。然后可以执行 make all，此时可以看到输出的 _my-app 文件，如屏显 2-1 所示。其中第二行 gcc 编译命令是由默认规则触发的，生成 my-app.o。第三、四、五行的命令是代码 2-1 的规则触发的，其中 ld 链接命令生成 _my-app 可执行文件。由于可执行文件发生了更新，因此触发了磁盘文件系统 fs.img 目标

的规则——第六行显示的命令使用 `mkfs` 程序生成 `fs.img` 磁盘映像文件，其中可执行文件列表的最后一个就是我们刚生成的 `_my-app`。

屏显 2-1 编译 `my-app.c`

1. `lqm@ubuntu:~/project/xv6-loongarch-exp$ make all`
2. `loongarch64-unknown-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -MD -march=loongarch64 -mabi=lp64s -ffreestanding -fno-common -nostdlib -l. -fno-stack-protector -fno-pie -no-pie -c -o user/my-app.o user/my-app.c`
3. `loongarch64-unknown-linux-gnu-ld -z max-page-size=4096 -N -e main -Ttext 0 -o user/_my-app user/my-app.o user/ulib.o user/usys.o user/printf.o user/umalloc.o`
4. `loongarch64-unknown-linux-gnu-objdump -S user/_my-app > user/my-app.asm`
5. `loongarch64-unknown-linux-gnu-objdump -t user/_my-app | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/my-app.sym`
6. `mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_my-app`
7. `nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 954 total 1000`
8. `balloc: first 493 blocks have been allocated`
9. `balloc: write bitmap block at sector 45`
10. `xxd -i fs.img > kernel/ramdisk.h`

启动 `xv6` 系统后，执行 `my-app` 程序，正确地输出了我们期待的 “This is my own app!” 字符串，如图 2-2 所示。



```

entry kernel ...
init: starting sh
$ ls
.          1 1 1024
..         1 1 1024
README    2 2 2226
cat        2 3 20200
echo       2 4 19200
forktest  2 5 11384
grep       2 6 22728
init       2 7 19920
kill       2 8 19088
ln         2 9 18984
ls         2 10 22048
mkdir      2 11 19168
rm         2 12 19160
sh         2 13 34832
stressfs   2 14 20032
usertests  2 15 125000
my-app     2 16 18704
console    3 17 0
$ my-app
This is my own app!
$

```

图 2-2 在 `xv6` 中运行新增的 `my-app` 程序

2.2. 新增系统调用

如果我们修改了 `xv6` 的代码，例如增加了调度优先级，那么就需要有设置优先级的系统调用，并且通过应用程序调用该系统调用进行优先级设置。因此我们需要学习如何增加新的系统调用，以及如何在应用程序中进行系统调用，后面才能验证 `xv6` 修改的功能。

比如，如果进程希望知道自身所在的处理器编号，这可以通过一个新的系统调用来实现。下面我们先学习如何在应用程序中调用现成的系统调用，然后再学习如何实现上述的新的系统调用。

2.2.1. 系统调用示例

可用的系统调用都在 `user/user.h` 中定义，我们在程序中直接使用即可。我们以获取进程号的 `getpid()` 系统调用为例，编写如代码 2-4 所示的 `print-pid.c` 代码。

代码 2-4 `print-pid.c`

```
1.  #include "kernel/types.h"
2.  #include "kernel/stat.h"
3.  #include "user/user.h"
4.
5.  int
6.  main(int argc, char *argv[])
7.  {
8.
9.      printf("My PID is: %d\n", getpid());
10.     exit(0);
11. }
```

按照前面的方法修改 `Makefile` 并重新生成 `xv6`，启动后在 `shell` 中执行 `print-pid` 并成功打印进程号，如图 2-3 所示。

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL

SetUefiImageMemoryAttributes - 0x00000000FEBF0000 - 0x00000000000050000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEB00000 - 0x00000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FEA60000 - 0x00000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FE9C0000 - 0x00000000000040000 (0x0000000000000000)
SetUefiImageMemoryAttributes - 0x00000000FE920000 - 0x00000000000040000 (0x0000000000000000)
kernel argc: 2, argv: 0xFE6F0000, bpi: 0xFFFFDA418.
entry kernel ...
cpu0: starting xv6
init: starting sh
$ print-pid
My PID is: 3
```

图 2-3 执行 `print-pid` 执行 `getpid()` 系统调用

2.2.2. 添加系统调用

由于系统调用涉及较多内容，分散在多个文件中——这包括系统调用号的分配、系统调用的分发代码（依据系统调用号）修改、系统调用功能的编码实现、用户库头文件修改等。另外还涉及验证用的样例程序——用于检验该系统调用的功能。

增加系统调用号

`xv6` 的系统调用都有一个唯一编号，定义在 `kernel/syscall.h` 中。我们可以在 `SYS_close` 的后面，新加入一行“`#define SYS_getcpuid 22`”即可，这里的编号 22 可以是其他值——只要不是前面使用过的就好。

增加用户态入口

为了让用户态代码能进行系统调用，需要提供用户态入口函数 `getcpuid()` 和相应的头文件。

■ 修改 user.h

为了让应用程序能调用用户态入口函数 `getcpuid()`，需要在 `user/user.h` 中加入一行函数原型声明 `“int getcpuid(void);”`。该头文件应该被应用程序段源代码所使用，因为它声明了所有用户态函数的原型。除此之外所有标准 C 语言库的函数都不能使用，因为 `Makefile` 用参数 `“-nostdinc”` 禁止使用 Linux 系统的头文件，而且用 `“-I.”` 指出在当前目录中搜索头文件。也就是说 xv6 系统中，并没有实现标准的 C 语言库。

■ usys.S 中定义用户态入口

定义了 `getcpuid()` 原型之后，还需要实现 `getcpuid()` 函数。我们在 `user/usys.pl` 中加入一行 `“entry(“getcpuid”);”`，例如可以插入到 `usys.pl` 第 38 行后面。`entry` 定义于 `usys.pl` 的第九行。`entry(“getcpuid”)` 经过宏展开，将把 `“getcpuid”` 定义为入口函数名，然后把 `SYS_getcpuid=22` 作为系统调用号保存到 `a7` 寄存器中，最后发出 `uint64` 指令进行系统调用 `“uint64 $T_SYSCALL”`。这样，经过用户态函数 `getcpuid()`，借助 `uint64` 指令进入到系统调用公共入口后，以 `a7` 作为下标在系统调用表 `syscalls[]` 中就可以找到需要执行的对应系统调用的具体代码。

这里定义的 `getcpuid()` 函数，就是在需要执行系统调用时所调用的用户态函数，使得用户代码无需编写汇编指令来执行 `uint64` 指令。

修改 syscall.c 中的跳转表

在系统调用公共入口 `syscall()` 中，xv6 将根据系统调用号进行分发处理。负责分发处理的函数 `syscall()`（定义于 `kernel/syscall.c`），分发依据是一个跳转表。我们需要这个修改跳转表，首先要在 `syscall.c` 第 107 行中的分发函数表 `syscalls[]` 中加入 `“[SYS_getcpuid] sys_getcpuid,”`，也就是下标 22 对应的是 `sys_getcpuid()` 函数地址（后面我们会实现该函数）。其次，由于 `sys_getcpuid` 未声明，因此要在它前面（例如第 106 行后面的位置）加入一行 `“extern uint64 sys_getcpuid(void);”` 用于指出该函数是外部符号。

前面提到：当用户发出 22 号系统调用是通过用户态函数 `getcpuid()` 完成的，其中系统调用号 22 是保存在 `a7` 的。因此 `syscall()` 系统调用入口代码可以通过 `p->trapframe->a7` 获得该系统调用号，并保存在 `num` 变量中，于是 `syscalls[num]` 就是 `syscalls[22]` 也就是 `sys_getcpuid()`。代码 2-5 是从 `syscall.c` 中截取的 `syscall()` 部分。

代码 2-5 系统调用分发代码 `syscall()`

```

1. void
2. syscall(void)
3. {
4.     int num;
5.     struct proc *p = myproc();
6.
7.     num = p->trapframe->a7;
8.     if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
9.         p->trapframe->a0 = syscalls[num]();
10.    } else {
11.        printf("%d %s: unknown sys call %d\n",
12.            p->pid, p->name, num);
13.        p->trapframe->a0 = -1;
14.    }
15. }
```

实现 sys_getcpuid()

前面的工作使得用户可以用 `getcpuid()` 作为系统调用户态的入口，而且进入系统调用的分发例程 `syscall()` 中也能正确地转入到 `sys_getcpuid()` 函数里，但是我们还未实现 `sys_getcpuid()` 函数。如代码 2-6 所示，在 `kernel/sysproc.c` 中加入系统调用处理函数 `sys_getcpuid()`——注意要和同名的用户态函数区分开来。

代码 2-6 `sysproc.c` 中添加 `sys_getcpuid()`

```
1.  uint64
2.  sys_getcpuid()
3.  {
4.      return getcpuid ();
5.  }
```

然后在 `kernel/proc.c` 中实现内核态的 `getcpuid()` 函数，如代码 2-7 所示。

代码 2-7 `proc.c` 中添加 `getcpuid()`

```
1.  int getcpuid()
2.  {
3.      int id = r_tp();
4.      return id;
5.  }
```

为了让 `sysproc.c` 中的 `sys_getcpuid()` 能调用 `proc.c` 中的 `getcpuid()`，还需要在 `kernel/defs.h` 加入一行 “`int getcpuid(void);`”，用作内核态代码调用 `getcpuid()` 时的函数原型。`defs.h` 定义了 `xv6` 几乎所有的内核数据结构和函数，也被几乎所有内核代码所包含。

```
1.  // proc.c
2.  int      cpid(void);
3.  void     exit(int);
4.  int      fork(void);
5.  int      growproc(int);
6.  void     proc_mapstacks(pagetable_t);
7.  pagetable_t proc_pagetable(struct proc *p);
8.  void     proc_freepagetable(pagetable_t, uint64);
9.  int      kill(int);
10. struct cpu* mycpu(void);
11. struct cpu* getmycpu(void);
12. struct proc* myproc();
13. void     procinit(void);
14. void     scheduler(void) __attribute__((noreturn));
15. void     sched(void);
16. void     sleep(void*, struct spinlock*);
17. void     userinit(void);
18. int      wait(uint64);
19. void     wakeup(void*);
20. void     yield(void);
21. int      either_copyout(int user_dst, uint64 dst, void *src, uint64 len);
22. int      either_copyin(void *dst, int user_src, uint64 src, uint64 len);
23. void     procdump(void);
24. int      getcpuid(void);
```

2.2.3. 验证新系统调用

最后，我们需要验证新增系统调用是否能被应用程序所正常使用。由于前面已经在 `user.h` 中声明了 `getcpuid()` 用户态函数原型，因此可以在应用程序中进行调用。编写如代码 2-8 所示的程序，打印本进程所在的 CPU 编号。参照前面 `my-app.c` 实验，完成其编译过程、加入到磁盘文件系统（记得要修改 `Makefile` 的 `UPROGS` 目标加上 `_pcpuid`）。

代码 2-8 pcpuid.c

```

1.  #include"kernel/types.h"
2.  #include"kernel/stat.h"
3.  #include"user/user.h"
4.
5.  int
6.  main(int argc, char *argv[])
7.  {
8.      printf("My CPU id is :%d\n",getcpuid());
9.      exit(0);
10. }
```

执行 pcpuid 程序，将打印出本进程所在的处理器编号，如屏显 2-2 所示。

屏显 2-2 pcpuid 运行结果

```

$ pcpuid
My CPU id is:0
$
```

2.3. 观察调度过程

在本章结束之前，我们再编写一个程序，使得它可以创建多个进程并发运行，用于观察多进程分时运行的现象。我们将 my-app.c 稍作修改，调用两次 fork() 来创建（实际上是复制自己）新的进程——两次 fork 调用将一共建 4 个进程，如代码 2-9 所示。

代码 2-9 my-app-fork.c

```

1.  #include"kernel/types.h"
2.  #include"kernel/stat.h"
3.  #include"user/user.h"
4.
5.
6.  int
7.  main(int argc, char *argv[])
8.  {
9.      int a;
10.     printf("This is my own app!\n");
11.     a=fork();
12.     a=fork();
13.     while(1)
14.         a++;
15.     exit(0);
16. }
```

按前面的 my-app 的方法，我们重新在磁盘文件系统中增加 my-app-fork 程序，运行后间断性地键入 Ctrl+P 用于显示当时的进程状态，如屏显 2-3 所示。在四次查看进程的状态中，发现第一次进程 3 在运行，第二次和第三次时进程 5 在运行，第四次进程 6 在运行。也就是说最多有一个进程能拥有 CPU。

屏显 2-3 查看 my-app-fork 运行时的进程状态

```

$ my-app-fork
This is my own app!
1 sleep  init
2 sleep  sh
3 run    my-app-fork
4 runble my-app-fork
5 runble my-app-fork
6 runble my-app-fork
```

```
1 sleep  init
2 sleep  sh
3 runble my-app-fork
4 runble my-app-fork
5 run    my-app-fork
6 runble my-app-fork
```

```
1 sleep  init
2 sleep  sh
3 runble my-app-fork
4 runble my-app-fork
5 run    my-app-fork
6 runble my-app-fork
```

```
1 sleep  init
2 sleep  sh
3 runble my-app-fork
4 runble my-app-fork
5 runble my-app-fork
6 run    my-app-fork
```

2.4. 小结

在未阅读 xv6 内核代码之前，读者先完成两个 xv6 的入门小实验，可以近距离体验到内核修改所带来的成就感。除了本书组织的初级、中级和高级实验外，感兴趣的读者可以进一步完成各章的后面的练习或者直接学习美国大学的操作系统课程的实验内容（例如华盛顿大学的实验内容¹）。

练习

1. 请为 xv6 增加一个新的应用程序，读者自行选定其功能。
2. 定义一个内核全局变量，用于进程间共享的目的。设计并实现两个系统调用 `read_sh_var()` 和 `write_sh_var()` 用于读取和修改该全局变量的值。编写应用程序，检验是否能在进程间完成数值的共享。

¹ <https://courses.cs.washington.edu/courses/cse451/15au/index.html>

第3章 中级实验

本章我们来一起完成 xv6 的进程和内存相关实验，从易到难逐步展开。本章实验的完整代码可以从 <https://github.com/SKT-CPUOS/xv6-loongarch-exp> 下载。

3.1. 调度实验

调度方面的我们安排两个实验：一个是热身性质的，只是将时间片延长到多个时钟中断，所需修改的代码比较少，也不涉及添加系统调用；另一个是优先级调度实验，由于还涉及系统调用以及用于验证的用户可执行程序，因此略微复杂一点。

调度对操作系统而言很关键，但是代码量并不大。相对于内存管理、文件系统而言，调度代码是相对简单的。

3.1.1. 调整时间片长度

首先我们来尝试将一个进程运行的时间片扩展为 N 个时钟周期。具体思路也很简单，为每个进程的 PCB 中添加时钟计数值，当前进程的时间片未用完则不切换。

增加时间片信息

在 xv6 的进程控制块 `kernel/proc.h` 中修改 `proc` 结构体，增加成员 `slot` 并定义时间片长度为 8 个 tick，如代码 3-1 所示。

代码 3-1 `proc.h` 中的 `proc` 结构体

```

1 // Per-process state
2 struct proc {
3     struct spinlock lock;
4
5     // p->lock must be held when using these:
6     enum procstate state; // Process state
7     void *chan;           // If non-zero, sleeping on chan
8     int killed;           // If non-zero, have been killed
9     int xstate;           // Exit status to be returned to parent's wait
10    int pid;               // Process ID
11
12    // wait_lock must be held when using this:
13    struct proc *parent;   // Parent process
14
15    // these are private to the process, so p->lock need not be held.
16    uint64 kstack;         // Virtual address of kernel stack
17    uint64 sz;             // Size of process memory (bytes)
18    pagetable_t pagetable; // User lower half address page table
19    struct trapframe *trapframe; // data page for uservec.S, use DMW address
20    struct context context; // swtch() here to run process
21    struct file *ofile[NOFILE]; // Open files
22    struct inode *cwd;       // Current directory

```

```

23     char name[16];           // Process name (debugging)
24     int slot;                // time slot (ticks)
25 };
26 #define SLOT 8              //one slot contains 8 ticks

```

然后在创建进程时分配 `proc` 结构体的 `allocproc()` 函数中设置新进程的 `slot` 成员，并它设置为 `SLOT`，如代码 3-2 所示。

代码 3-2 `proc.c` 的 `allocproc()` 中设置时间片

```

1     ...
1     found:
2         p->state = EMBRYO;
3         p->pid = nextpid++;
4         p->slot = SLOT;
5     ...

```

为了能查看到进程时间片信息，还需要在 `kernel/proc.c` 中的 `procdump()` 函数中将输出信息增加一项时间片剩余量，如代码 3-3 所示。

代码 3-3 `proc.c` 中 `procdump()` 显示时间片信息

```

1     ...
2     printf("slice left:%d ticks,%d %s %s", p->slot, p->pid, state, p->name);
3     ...

```

时间片控制

`xv6` 原本是在每次时钟中断时就调用 `yield()` 让出 CPU 并引发一次调度。现在修改后的代码需要对时间片剩余量进行递减，以及判定当前进程时间片是否用完——决定是否需要进行调度。修改后的代码在 `trap.c` 的 `usertrap()` 函数中完成上述检查，如代码 3-4 所示。

代码 3-4 `trap.c` 中检查时间片是否用完

```

1     ...
2     // give up the CPU if this is a timer interrupt.
3     // if(which_dev == 2)
4     //     yield();
5
6     if(p && p->state == RUNNING) {
7         p->slot--;
8         if(p->slot == 0) {
9             p->slot=8;
10            yield();
11        }
12    }
13    ...

```

查看时间片信息

我们编写了 `loop.c` 作为示例应用程序，用于查看进程时间片信息。`loop.c` 有父子两个进程，分别进行长时间的循环计算，如代码 3-5 所示。编译前，不要忘记在 `Makefile` 的 `UPROGS` 目标中增加一项 “`_loop\`”。

代码 3-5 `loop.c`

```

1     #include "kernel/types.h"
2     #include "kernel/stat.h"
3     #include "user/user.h"
4
5     int
6     main(int argc, char *argv[])
7     {
8         int pid;

```

```

8      int data[8];
9      int i, j, k;
10
11     pid=fork();
12     for( i=0; i<2; i++)
13     {
14         for( j=0; j<1024*100; j++)
15             for( k=0; k<1024*1024; k++)
16                 data[k%8]=pid*k;
17     }
18     printf("%d ", data[0]);
19     exit();
20 }

```

最后在 loop 运行时，就可以用 Ctrl+P 检查当前进程剩余的时间片。从屏显 3-1 屏显 3-2 可以看出各个进程在两次 Ctrl+P 检查时，所剩时间片的 tick 计数

屏显 3-1 屏显 3-2 查看 loop 运行时进程的时间片剩余量

```

$ loop
slice left:7 ticks,1 sleep init
slice left:5 ticks,2 sleep sh
slice left:8 ticks,3 runble loop
slice left:3 ticks,4 run loop
slice left:7 ticks,1 sleep init
slice left:5 ticks,2 sleep sh
slice left:4 ticks,3 run loop
slice left:8 ticks,4 runble loop

```

3.1.2. 优先级调度

接下来再做一个调度相关的简单实验，为原来 RR 调度增加优先级——只调度最高优先级进程，如果有多个最高优先级的进程，则这些进程间按照 RR 方式调度。为了实现优先级调度，首先需要在进程控制块中加入优先级成员，然后需要修改调度器的调度算法。当然还需要提供修改和设置优先级的系统调用，如果是动态优先级还需要优先级调整算法等。我们下面来实现 xv6 系统上的静态优先级调度。

增加优先级属性

在 xv6 的进程控制块 kernel/proc.h 中修改 proc 结构体，增加成员，如代码 3-6 所示。

代码 3-6 proc.h 中的 proc 结构体

```

1  // Per-process state
2  struct proc {
3      struct spinlock lock;
4
5      // p->lock must be held when using these:
6      enum procstate state;          // Process state
7      void *chan;                    // If non-zero, sleeping on chan
8      int killed;                    // If non-zero, have been killed
9      int xstate;                    // Exit status to be returned to parent's wait
10     int pid;                        // Process ID
11
12     // wait_lock must be held when using this:
13     struct proc *parent;            // Parent process
14
15     // these are private to the process, so p->lock need not be held.

```

```

16  uint64 kstack;           // Virtual address of kernel stack
17  uint64 sz;               // Size of process memory (bytes)
18  pagetable_t pagetable;   // User lower half address page table
19  struct trapframe *trapframe; // data page for uservec.S, use DMW address
20  struct context context;   // switch() here to run process
21  struct file *ofile[NOFILE]; // Open files
22  struct inode *cwd;        // Current directory
23  char name[16];           // Process name (debugging)
24  int priority;            // Process priority (0-20)=(highest-lowest)
25  };

```

既然有了优先级，那么在创建进程的时候就需要指定一个优先级或设置一个默认优先级。我们选择创建时使用默认优先级，后续需要的时候再调整优先级的方案。因此创建进程时分配 `proc` 结构体的 `allocproc()` 需要设置新进程的 `priority` 成员，并将 10 作为默认优先级，如代码 3-7 所示。

代码 3-7 `proc.c` 的 `allocproc()` 中设置默认优先级

```

1  ...
2  found:
3      p->pid = allocpid();
4      p->state = USED;
5      p->priority=10;           //default priority
6
7      // Allocate kernel stack.
8      if((p->trapframe = (struct trapframe *)kalloc()) == 0) {
9  ...

```

为了能查看进程的优先级，我们需要修改 `proc.c` 中的 `procdump()` 函数，使之能打印优先级信息，如代码 3-8 所示。除了打印进程优先级 `p->priority` 之外，在格式上也有一点小修改，使得打印输出的效果更好一点。

代码 3-8 `procdump()` 中打印优先级信息

```

1  void
2  procdump(void)
3  {
4      static char *states[] = {
5          [UNUSED]    "unused",
6          [SLEEPING]  "sleep ",
7          [RUNNABLE]  "runble",
8          [RUNNING]   "run   ",
9          [ZOMBIE]    "zombie"
10     };
11     struct proc *p;
12     char *state;
13
14     printf("\n");
15     for(p = proc; p < &proc[NPROC]; p++) {
16         if(p->state == UNUSED)
17             continue;
18         if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
19             state = states[p->state];
20         else
21             state = "???";
22         // printf("%d %s %s", p->pid, state, p->name);
23         printf("PID=%d state=%s priority=%d %s:", p->pid, state, p->priority,
24             p->name);
25     }

```

26 }

设置优先级

既然有基于优先级的调度，那么就需提供设置优先级的系统调用。由于前面第 2 章讨论过如何添加新的系统调用，这里只给出简要说明和具体代码。这个新的系统调用命名为 `chpri`（change priority）。首先在 `kernel/syscall.h` 中为新的系统调用定义其编号（必须和其他系统调用编号不同）：

```
1 #define SYS_chpri 22
```

在 `user.h` 中增加用户态函数原型 `int chpri(int pid, int priority)` 函数，第一个参数用于指出进程号，第二个参数指出新的优先级：

```
1 int chpri( int, int );
```

然后在 `usys.pl` 中添加 `chpri()` 函数的汇编实现代码（宏展开后对应于 `sys_chpri` 函数）：

```
1 entry("chpri")
```

再接下来修改系统调用的跳转表，在 `syscall.c` 中的 `syscalls[]` 数组中添加一项：

```
1 [SYS_chpri] sys_chpri,
```

由于 `kernel/syscall.c` 中未定义 `sys_chpri` 函数，因此需要在上面这个 `syscalls[]` 数组前面增加一个外部函数声明：

```
1 extern uint64 sys_chpri(void);
```

最后我们在 `kernel/sysproc.c` 中实现 `sys_chpri()`，如代码 3-9 所示。简单地检查进程号和优先级不能为负数，然后调用 `chpri(pid,pr)` 编号为 `pid` 的进程优先级设置为 `pr`。读者可以增加一个约束，实现优先级不大于 20——正如代码 3-6 最后一行注释里期待的那样。

代码 3-9 `sysproc.c` 中添加 `sys_chpri()`

```
1
2 int
3 sys_chpri ( void )
4 {
5     int pid, pr;
6     if ( argint(0, &pid) < 0 )
7         return -1;
8     if ( argint(1, &pr) < 0 )
9         return -1;
10    return chpri ( pid, pr );
11 }
```

上面的 `chpri()` 函数实现放到 `proc.c` 中，如代码 3-10 所示。

代码 3-10 `proc.c` 中的 `chpri`

```
1 uint64
2 chpri(int pid, int priority)
3 {
4     struct proc *p;
5
6     for(p = proc; p < &proc[NPROC]; p++) {
7         acquire(&p->lock);
8         if(p->pid == pid) {
9             p->priority = priority;
10            release(&p->lock);
11            break;
12        }
13        release(&p->lock);
14    }
15 }
```

```

16     return (uint64)pid;
17 }

```

最后在 `defs.h` 的 `proc.c` 部分将添加函数原型 “`uint64 chpri(int,int);`”，以便内核代码访问该函数。

修改调度器

为进程添加优先级的信息后，还需要在调度器中修改调度行为。我们并没有对进程控制块数组进行改动。每次调度时，先检查所有的可运行程序的最高优先级，最先找到的就绪进程就是优先级最高的进程。由于我们在调度器里只会向后寻找优先级相等或者更高的可运行程序，或者完成一轮循环后选取目前优先级最高的程序。具体实现如代码 3-11 所示。

代码 3-11 `proc.c` 中修改后的 `scheduler()` 调度器

```

1  void
2  scheduler(void)
3  {
4      struct proc *p;
5      struct proc *temp;
6      struct cpu *c = mycpu();
7      int priority;
8      int needed = 1;    // 是否需要重新搜索最高优先级
9
10     c->proc = 0;
11
12     for(;;) {
13         // Avoid deadlock by ensuring that devices can interrupt.
14         intr_on();
15
16         for(p = proc; p < &proc[NPROC]; p++) {
17             if(needed) {
18                 priority = 19;
19                 for(temp = proc; temp < &proc[NPROC]; temp++) // 获取当前可运行的最高当前优先级
20                     {
21                         if(temp->state == RUNNABLE && temp->priority < priority)
22                             {
23                                 priority = temp->priority;
24                             }
25                     }
26
27             }
28
29             needed = 0;
30             if(p->state != RUNNABLE)
31                 continue;
32             if(p->priority > priority)
33                 continue;
34
35
36             acquire(&p->lock);
37             if(p->state == RUNNABLE) {
38                 // Switch to chosen process. It is the process's job
39                 // to release its lock and then reacquire it
40                 // before jumping back to us.
41                 p->state = RUNNING;
42                 c->proc = p;

```

```

43         swtch(&c->context, &p->context);
44
45         // Process is done running for now.
46         // It should have changed its p->state before coming back.
47         c->proc = 0;
48     }
49     release(&p->lock);
50     needed = 1;
51 }
52 }
53 }

```

验证优先级调度

我们创建一个子进程，其优先级为 5，然后查看它们被调度的情况。编辑如代码 3-12 所示的程序，并在 Makefile 中的 UPROGS 中添加 “_prio-sched\”，最后 make all 完成编译。

代码 3-12 prio-sched.c

```

1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int
6  main(int argc, char * argv[])
7  {
8      int pid;
9      printf("This is a demo for prio-schedule!\n");
10     pid = getpid();
11     chpri(pid, 19); //系统默认优先级是 10
12
13     int i = 0;
14
15     pid = fork();
16     if(pid == 0) //子进程
17     {
18         chpri(getpid(), 5);
19         i = 1;
20         while(i <= 10000)
21         {
22             if(i / 1000 == 0)
23             {
24                 printf("p2 is running\n");
25             }
26             i++;
27         }
28         printf("p2 sleeping\n");
29         sleep(100);
30         i=1;
31         while(i <= 1000000)
32         {
33             if(i/100000 == 0)
34             {
35                 printf("p2 is running again\n");
36             }
37             i++;
38         }
39         printf("p2 finished\n");
40     }else //父进程
41     {

```

```

42         i = 1;
43         while (i > 0)
44         {
45             if(i/1000000 == 0)
46                 printf("p1 is running\n");
47             i++;
48             if(i > 1000000)
49                 break;
50         }
51
52     }
53     exit(0);
54 }

```

运行 xv6 系统，并且在 shell 中键入 `prio-sched` 命令，执行测试程序结果如屏显 3-2 所示。我们看到 `pid=3` 是 `prio-sched` 一开始的进程号，该进程被我们设置为优先级 19，因此子进程优先级高于父进程。后面 `fork` 出来的有 `pid=4` 的进程优先级为 5。我们使用单核 CPU 的配置，此时执行次序是子进程先执行，然后子进程 `sleep`，父进程执行，最后子进程又抢占执行，最后子进程结束，父进程继续执行。为了便于区分，屏显 3-2 中父进程的输出用灰色背景标示。

屏显 3-2 `prio-sched` 优先级调度执行结果

```

init: starting sh
$ prio-sched
This is a demo for prio-schedule!
p2 is running *n
p2 sleeping
p1 is running *n
p2 is running again *n
p2 finshed
p1 is running *n
$

```

在运行过程中，通过 `Ctrl+P` 也可以看到调度的实时状态，例如屏显 3-3 给出了当时 `pid=4` 的进程在运行，而 `pid=3` 的进程因优先级较低（`priority` 数值较大）未能获得 CPU。

屏显 3-3 `prio-sched` 中 `pid=3` 和 `pid=4` 的进程在运行

```

PID=1 state=sleep prio=10 init
PID=2 state=sleep prio=10 sh
PID=3 state=runble prio=19 prio-sched
PID=4 state=run prio=5 prio-sched

```

当显示 `pid=4` 的进程结束后，再用 `Ctrl+P` 查看时将获得如屏显 3-4 所示的输出，可以看到此时 `pid=4` 的进程已经处于 `zombie` 状态（父进程没有执行 `wait()` 系统调用的缘故），而 `pid=3` 的进程已经成为系统中优先级最高的进程，因此正在运行处于 `run` 调度状态。

屏显 3-4 `prio-sched` 中 `pid=6` 和 `pid=7` 的进程已经结束，`pid=3` 和 `pid=4` 的进程正在运行

```

PID=1 state=sleep prio=10 init
PID=2 state=sleep prio=10 sh
PID=3 state=run prio=19 prio-sched
PID=4 state=zombie prio=5 prio-sched

```


上述执行过程可以基本确定我们的调度算法功能正常。如果程序执行时间过快或过慢而不利于观察，读者可以自行调整里面的循环次数以获得最佳的观测效果。

3.2. 实现信号量

虽然 xv6 提供了自旋锁用于内核代码的并发同步，但是用户态并没有提供同步手段。我们这里尝试实现一个简单的信号量机制来为用户进程提供同步，内部实现上仍是建立在内核自旋锁之上——加上了阻塞睡眠的能力从而避免“忙等”的问题。

由于 xv6 没有提供类似共享内存这样的共享资源，我们就在系统中定义一个共享整数变量 `sh_var_for_sem_demo`，通过 `sh_var_read()` 和 `sh_var_write()` 进行读写操作。以此作为验证信号量工作正常的功能展示。

3.2.1. 共享变量及其访问

共享变量

验证信号量的时候需要提供临界资源，因此我们定义了 `sh_var_for_sem_demo` 全局变量，在 `kernel` 中新建 `sem.h` 文件并添加：“`uint sh_var_for_sem_demo`”。由于内核空间为所有进程所共享，因此该变量可以被所有进程所感知。

访问共享变量

为了访问这个共享变量，需要提供两个系统调用来完成读写操作。添加系统调用的方法在第 2 章讨论过，这里就简单给出添加过程而不过多解释。在 `syscall.h` 末尾插入两行“`#define SYS_sh_var_read 22`”和“`#define SYS_sh_var_write 23`”，为读写操作的系统调用进行编号。接着在 `user.h` 中声明“`int sh_var_read(void);`”和“`int sh_var_write(int);`”两个用户态函数原型，并在 `usys.pl` 末尾插入两行“`entry("sh_var_read");`”和“`entry("sh_var_write");`”以实现上述两个函数。下面还要修改系统调用跳转表，即 `syscall.c` 中的 `syscalls[]` 数组——添加两个元素“`[SYS_sh_var_read] sys_sh_var_read,`”和“`[SYS_sh_var_write] sys_sh_var_write,`”。同时还要在 `syscall.c` 的 `syscalls[]` 数组前面声明上述两个函数是外部函数“`extern uint64 sys_sh_var_read(void);`”和“`extern uint64 sys_sh_var_write(void);`”。

最后则是在 `sysproc.c` 中实现 `sh_var_read()` 和 `sh_var_write()`，如代码 3-13 所示。与前面第 2 章增加无参数的系统调用不同，这里需要解决参数的获取的问题。参数获取的函数在 `syscall.c` 中定义，本例子可以用其中的 `argint()` 来获取整数参数。为了能访问到 `sh_var_for_sem_demo` 变量，还需要在 `sysproc.c` 中包含 `sem.h` 头文件。

代码 3-13 `sysproc.c` 中插入 `sys_sh_var_read()` 和 `sys_sh_var_write()`

```

1  uint64
2  sys_sh_var_read()
3  {
4      return (uint64)sh_var_for_sem_demo;
5  }
6  uint64
7  sys_sh_var_write()
8  {
```

```

9     int n;
10    if(argint(0, &n) < 0)
11        return (uint64)-1;
12    sh_var_for_sem_demo = n;
13    return (uint64)sh_var_for_sem_demo;
14 }

```

无互斥的并发访问

定义了共享变量以及访问的系统调用之后，我们可以在应用程序中尝试并发访问它们。编写 `sh_rw_nolock.c`，如所示。同时还需要修改 `Makefile` 的 `$UPROGS`，添加一个“`_sh_rw_nolock\`”。

代码 3-14 `sh_rw_nolock.c`

```

1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int
6  main(int argc, char * argv[])
7  {
8      int pid = fork();
9      int i, n;
10     for(i = 0; i < 100000; i++)
11     {
12         n = sh_var_read();
13         sh_var_write(n+1);
14     }
15
16     printf("sum = %d\n", sh_var_read());
17     if(pid > 0)
18     {
19         wait(0);
20     }
21
22     exit(0);
23 }

```

编译后运行，可以发现计数结果并不正确（正确值为 200000），如屏显 3-5 所示。

屏显 3-5 无护持保护下 `_sh_rw_nolock` 错误的计数结果

```

$ sh_rw_nolock
sum = 122709
sum = 133382
$

```

3.2.2. 信号量数据结构

为了实现信号量，除了创建、撤销、P、V 操作外，还需要添加新的数据结构、初始化函数、调整 `wakeup` 唤醒操作等。

为了管理信号量我们声明 `struct sem` 结构体，其中 `resource_count` 成员用于记录信号量中资源的数量，`lock` 内核自旋锁是为了让信号量的操作保持原子性，`allocated` 用于表示该信号量是否已经分配使用。

整个系统内部声明一个信号量数组 `sems[128]`，也就是说用户进程申请的信号量总数不超过 128 个。我们把这些代码放到 `kernel/spinlock.h` 中，相应的数据定义如代码 3-15 所示。

代码 3-15 新增的信号量声明代码

```

1.  #define SEM_MAX_NUM    128           //信号量总数
1.  extern int      sem_used_count;       //当前在用信号量数目
2.  struct sem{
3.      struct spinlock lock;           //内核自旋锁
4.      int      resource_count;        //资源计数
5.
6.      int      allocated;             //是否被分配使用: 1 已分配, 0 未分配
7.  };
8.  extern struct sem  sems[SEM_MAX_NUM]; //系统可有 SEM_MAX_NUM 个信号量

```

3.2.3. 信号量操作的系统调用

为了实现信号量，我们需要增加四个系统调用，分别是创建信号量 `sem_create()`，其参数是信号量的初值（例如互斥量则用 1 作为初值），返回值是信号量的编号，即内核变量 `sems[]` 数组的下标。`sem_p()`则是对指定编号的信号量进行 `p` 操作（减一操作、`down` 操作），反之 `sem_v()`则是对指定 `id` 的信号量进行 `v` 操作（增一操作、`up` 操作），这两个操作和操作系统理论课堂上讨论的行为一致，都会涉及到进程的睡眠或唤醒操作。`sem_free()`是撤销一个信号量。

（1）`int sem_create(int n_sem)` 参数 `n_sem` 是初值，返回的是信号量的编号，-1 为出错

（2）`int sem_p(int sem_id)` 减一操作，减为 0 时阻塞睡眠，记录到 `sem.procs[]` 中。返回值 0 表示正常，返回值 -1 则出错。

（3）`int sem_v(int sem_id)` 增一操作，增加到 0 时唤醒队列中的进程，清除 `sems[id].procs[]` 对应的进程号。返回值为 0 表示成功，-1 表示出错。

（4）`int sem_free(int sem_id)` 释放指定 `id` 的信号量。返回值为 0 表示成功，-1 表示出错。

信号量的核心代码

我们将信号量的核心实现代码放在 `kernel/spinlock.c` 中，而不是用独立的 C 文件，从而避免增加 `Makefile` 上的修改工作。

■ `seminit()`

系统启动时要调用 `seminit()` 对信号量进行初始化。`seminit()` 完成的工作很简单，就是完成信号量数组的自旋锁的初始化。我们把该函数的代码插入到 `kernel/spinlock.c` 中，具体如代码 3-16 所示。

代码 3-16 `seminit()`

```

1  int      sem_used_count =0;
2  struct sem  sems[SEM_MAX_NUM];
3
4  void seminit() {
5      int i;
6      for(i=0; i<SEM_MAX_NUM; i++) {
7          initlock(&(sems[i].lock), "semaphore");
8          sems[i].allocated=0;
9      }
10 }

```

然后我们在 kernel/main.c 的 main() 中插入一行 “seminit(); //semaphor”（插在 userinit() 之前）。为了让 main.c 能调用 seminit(), 还需要在 defs.h 中插入 seminit() 函数原型。

■ sys_sem_create()

sys_sem_create() 扫描 sems[] 数组，查看里面 allocated 标志，发现未用的则将其 allocated 置 1，即可返回其编号。如果扫描一次后未发现，则返回错误代码。注意每次操作时需要 sems[i] 进行加锁操作，检查完成后进行解锁操作。

代码 3-17 sys_sem_create()

```

1  int
2  sys_sem_create() {
3      int n_sem, i;
4      if(argint(0, &n_sem) < 0)
5          return -1;
6      for(i = 0; i < SEM_MAX_NUM; i++) {
7          acquire(&sems[i].lock);
8          if(sems[i].allocated == 0) {
9              sems[i].allocated = 1;
10             sems[i].resource_count = n_sem;
11             printf("create %d sem\n", i);
12             release(&sems[i].lock);
13             return i;
14         }
15         release(&sems[i].lock);
16     }
17     return -1;
18 }
```

■ sys_sem_free()

sys_sem_free() 将指定 id 作为下标访问 sems[id] 获得当前信号量 sems[id]，然后对 sems[id].lock 加锁，判定该信号量上没有睡眠阻塞的进程，则将 sems[id].allocated 标志设置为未使用，从而释放信号量，最后对 sems[id].lock 解锁。

代码 3-18 sys_sem_free()

```

1  int
2  sys_sem_free() {
3      int id;
4      if(argint(0, &id) < 0)
5          return -1;
6      acquire(&sems[id].lock);
7      if(sems[id].allocated == 1 && sems[id].resource_count > 0) {
8          sems[id].allocated = 0;
9          printf("free %d sem\n", id);
10     }
11     release(&sems[id].lock);
12     return 0;
13 }
```

■ sys_sem_p()

sys_sem_p() 将指定 id 作为下标访问 sems[id] 获得当前信号量 sem，然后用 acquire() 对 sems[id].lock 加锁，加锁成功后对 sems[id].resource_count --，接着用 release() 解锁退出临界区。如果发现 sems[id].resource_count < 0 则睡眠。其他情况下则直接返回表示完成 p 操作。

注意在 `sleep()` 的时候，会释放 `sems[id].lock` 才执行 `sched()` 切换——允许其他进程继续执行 P 操作或 V 操作。而 `sleep()` 返回前，会再次持有 `sems[id].lock`——即使有多个等待进程被唤醒，也只有一个进程能被唤醒并退出睡眠阻塞状态。

代码 3-19 `sys_sem_p()`

```

1  int sys_sem_p()
2  {  int id;
3      if(argint(0, &id) < 0)
4          return -1;
5      acquire(&sems[id].lock);
6      sems[id].resource_count--;
7      if(sems[id].resource_count < 0)           //首次进入、或被唤醒时，资源不足
8          sleep(&sems[id], &sems[id].lock); //睡眠（会释放 sems[id].lock 才阻塞）
9      release(&sems[id].lock);                 //解锁（唤醒到此处时，重新持有 sems[id].lock）
10     return 0;                                //此时获得信号量资源
11 }
```

■ `sys_sem_v()`

`sys_sem_v()` 将指定 `id` 作为下标访问 `sems[id]` 获得当前信号量 `sem`，然后对 `sem.lock` 加锁，加锁成功后对 `sem.resource_count+=1`，如果发现 `sem.resource_count>=0`，则解锁 `sem.lock`，并唤醒该信号量上阻塞的睡眠进程。否则直接返回。

代码 3-20 `sys_sem_v()`

```

1.  int sys_sem_v(int sem_id)
2.  {  int id;
3.      if(argint(0, &id) < 0)
4.          return -1;
5.      acquire(&sems[id].lock);
6.      sems[id].resource_count+=1;           //增 1
7.      if(sems[id].resource_count < 1)       //有阻塞等待该资源的进程
8.          wakeup1p(&sems[id]);             //唤醒等待该资源的 1 个进程
9.      release(&sems[id].lock);              //释放锁
10.     return 0;
11. }
```

修正 `wakeup` 操作

由于 `xv6` 系统自带的 `wakup` 操作会将所有等待相同事件的进程唤醒，因此也可以重写一个新的 `wakeup` 操作函数 `wakup1p()`，仅唤醒等待指定信号量的一个进程，从而避免“群惊”效应。我们将 `wakup1p()` 函数放在 `proc.c` 中，具体如代码 3-21 所示。另外，还需要在 `defs.h` 中声明该函数原型“`void wakeup1p(void*);`”。

代码 3-21 `wakup1p()`

```

1  void wakeup1p(void *chan) {
2      struct proc *p;
3
4      for (p = proc; p < &proc[NPROC]; p++)
5      {
6          if(p != myproc()) {
7              acquire(&p->lock);
8              if(p->state == SLEEPING && p->chan == chan) {
9                  p->state = RUNNABLE;
10                 release(&p->lock);
11                 break;
12             }
13             release(&p->lock);

```

```

14     }
15
16 }
17

```

系统调用的辅助代码

除了上述四个系统调用的核心实现代码外，还有系统调用号的设定、用户入口函数、系统调用跳转表的修改等工作，一并在此给出，以便读者操作时对照检查。

在 `syscall.h` 末尾插入四行“`#define SYS_sem_create 24`”“`#define SYS_sem_free 25`”“`#define SYS_sem_p 26`”和“`#define SYS_sem_v 27`”，为新添的四个系统调用进行编号。接着在 `user.h` 中声明“`int sem_create (int);`”“`int sem_free (int);`”“`int sem_p (int);`”和“`int sem_v (int);`”四个用户态函数原型，并在 `usys.pl` 末尾插入四行“`entry (sem_create)`”“`entry (sem_free)`”“`entry (sem_p)`”和“`entry (sem_v)`”以实现上述四个函数。下面还要修改系统调用跳转表，即 `syscall.c` 中的 `syscalls[]` 数组——添加四个元素“`[SYS_sem_create] sys_sem_create,`”“`[SYS_sem_free] sys_sem_free,`”“`[SYS_sem_p] sys_sem_p,`”和“`[SYS_sem_v] sys_sem_v,`”。同时还要再 `syscall.c` 的 `syscalls[]` 数组前面声明上述两个函数是外部函数“`extern uint64 sys_sem_create (void);`”“`extern uint64 sys_sem_free (void);`”“`extern uint64 sys_sem_p (void);`”和“`extern uint64 sys_sem_v(void);`”。

3.2.4. 用户测试代码

我们重新编写一个访问共享变量的应用程序，并且加上信号量的互斥控制，如代码 3-22 所示。修改 `Makefile` 为 `UPROGS` 添加一个“`_sh_rw_lock`”，重新编译生成系统。

代码 3-22 `sh_rw_lock.c`

```

1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5  int main() {
6      int id=sem_create(1);
7      int pid = fork();
8      int i;
9      for(i=0;i<100000;i++){
10         sem_p(id);
11         sh_var_write(sh_var_read()+1);
12         sem_v(id);
13     }
14     if(pid >0) {
15         wait(0);
16         sem_free(id);
17     }
18     printf("sum=%d\n", sh_var_read());
19     exit(0);
20 }

```

系统启动后，运行 `sh_rw_lock` 程序，可以看到此次两个并发进程对计数值的累加可以获得正确的计数结果 200000，如屏显 3-6 所示。

屏显 3-6 `sh_rw_lock` 输出的正确计数结果

```
$ sh_rw_lock
```

```

create 0 sem
sum = 200000
free 0 sem
sum = 200000
$

```

当子进程先退出的时候，第一个输出的值可能会比 200000 略小一点。

3.3. 实现进程间通信的实验

我们选取共享内存、消息队列两种进程间通信作为本节的实验。

3.3.1. 共享内存的实现

我们设计一个简化版的共享内存，远达不到 Linux 共享内存的通用程度，但也能将共享内存的核心思想体现出来。简化后的限制包括：整个系统只有固定的若干个共享内存区；进程不允许将一个共享内存区反复映射到自己的虚存空间；进程退出时自动解除共享映射（不提供显式解除映射的系统调用）；共享内存区作为进程最高端的空间从而避免与 xv6 原来的 `sbrk` 操作相冲突。此时系统上进程映射共享内存的形态如图 3-1 所示。

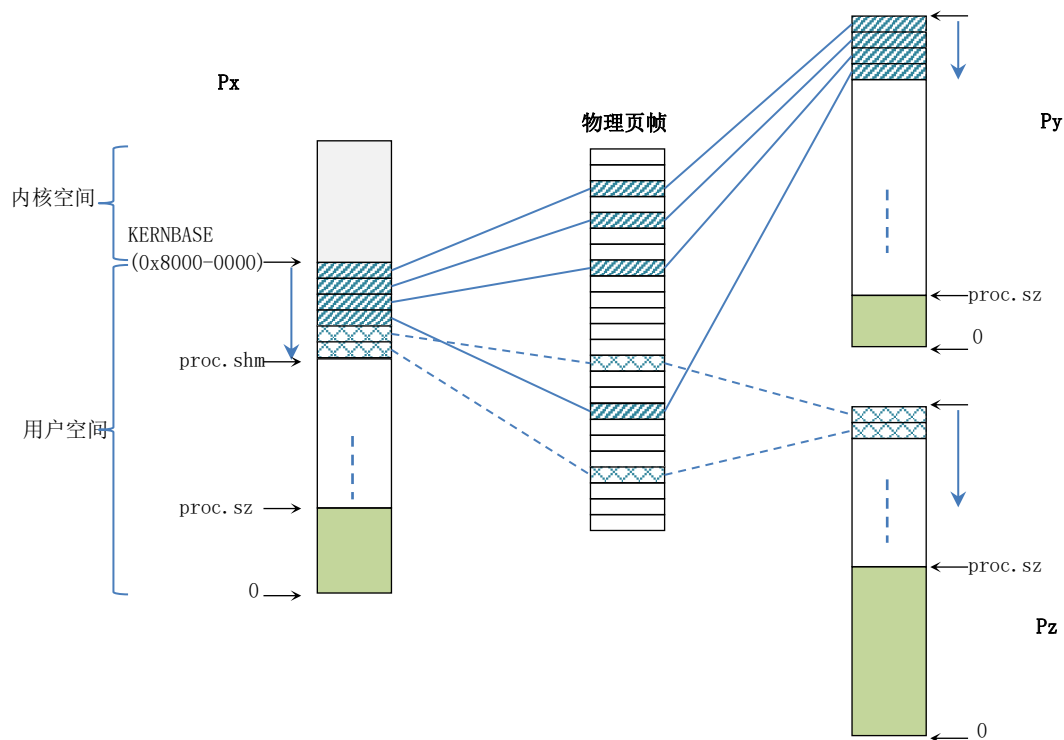


图 3-1 一种 xv6 共享内存方案示意图

图 3-1 展示了 Px 和 Py 共享四个页帧、Px 和 Pz 共享两个页帧的情况。由于系统中一共有 8 个共享内存区，因此系统中每个共享内存使用一个引用计数来表明是否启用。每个进程则使用一个 8 位的掩码（`shmkeymask`）或者一个编号（`key`、`i` 等）来指示其中一个区域，其中 `shmkeymask` 用于记录本进程对这些共享内存区的启用情况，而 `key` 用来指出一个共享内存区的编号，而 `i` 往往用作遍历这些内存区时的循环变量。

共享内存需要实现不同进程访问到同一块物理内存，其核心机制是在不同进程分配 `pte` 指向相同物理页帧（物理内存地址），从而使得该页帧在不同进程中都是可见的。

核心数据结构

简化方案中只允许系统拥有最多 8 个共享内存区，使用 `shmtab[8]` 记录，每个成员是一个 `sharemem` 结构体，里面的成员包括引用计数（被进程映射的次数）、本共享内存区间大小（即页帧数量，且不超过 4）、本区间所映射的物理页帧。由于是一个简化系统，上述的限制都显得并不合理，读者可以自行扩展突破其限制。我们新建一个 `sharemem.c` 文件（记得要在 `Makefile` 中将该文件编译链接到 `kernel`），并定义相关数据结构，如代码 3-23 所示。

代码 3-23 新建的 `kernel/sharemem.c`

```
1  #include "types.h"
2  #include "param.h"
3  // #include "mmu.h"
4  #include "loongarch.h"
5  #include "spinlock.h"
6  #include "proc.h"
7  #include "defs.h"
8  #include "memlayout.h"
9  #define MAX_SHM_PGNUM (4) //每个共享内存最带 4 页内存
11 struct sharemem
12 {
13     int refcount;           //当前共享内存引用数，当引用数为 0 时才会真正回收
14     int pagenum;           //占用的页数 (0-4)
15     void* physaddr[MAX_SHM_PGNUM]; //对应每页的物理地址
16 };
17 struct spinlock shmlock;   //用于互斥访问的锁
18 struct sharemem shmtab[8]; //整个系统最多 8 个共享内存
```

除此之外，还需要在 `proc` 结构（`kernel/proc.h`）中添加 3 个变量：`shm`，用来标记进程用户空间的上限，原本是 `KERNELBASE`，但共享内存分配在用户空间顶部，记录这个上限就能正常使用 `sbrk()`；`shmkeymask` 是一个掩码，使用 8 位标记，使用了第几个共享内存就在第几位置 1；`shmva[8]` 记录进程使用共享内存的地址。

```
1  uint shm;           //本进程共享内存区域的下边界
2  uint shmkeymask;    //本进程的 8 个共享内存区与使用掩码（位图）
3  void* shmva[8];     //本进程共享内存起始地址（虚地址）列表
```

实现细节

参考 linux 相关设计，我们新增一个系统调用 `void *shmgetat(id, size)`。其参数 `id` 代表 `shmtab[8]` 的下标，用于指定获取 8 个共享内存区中的哪一块；`size` 代表共享内存大小。该系统调用返回对应共享内存的虚拟地址。

进程调用 `shmgetat(id, size)` 时，根据共享内存是否已经存在而分成两种情况来处理：

（1）如果 `shmtab[id]` 对应的共享内存区还未创建，则根据 `size` 初始化 `shmtab[id]`，分配对应的物理内存建立页表映射后返回该内存的虚拟地址。

（2）如果 `shmtab[id]` 共享内存区已经存在，则新建页表项 `pte` 绑定 `shmtab[id].pagenum` 个页帧的物理地址，返回对应的虚拟地址，此时忽略传入的 `size` 参数。

类似地，我们还需要用获得共享内存引用数的 `shmrefcount(uint id)`，需要共享内存区的初始化操作函数 `sharememinit()` 等配套的系列函数。这些新添加的函数的实现在 `sharemem.c` 文件中，同时需要 `defs.h` 中声明，以便其他代码调用，如代码 3-24 所示，我们将逐个分析。

代码 3-24 在 `defs.h` 中增加共享内存相关的函数声明

```

1  .....
2  struct sharemem;
3  .....
4  // sharemem.c
5  void      sharememinit();
6  void*     shmgetat(uint, uint);
7  int       shmrefcount(uint);
8  int       shmrelease(pde_t*, uint64, uint);
9  void      shmaddcount(uint);
10 int       shmkeyused(uint, uint);

```

■ 初始化

在 `sharemem.c` 中添加一个 `sharememinit` 方法，对共享变量描述符表 `shmtab[]` 进行初始化。

代码 3-25 `sharememinit()`

```

1  void
2  sharememinit()
3  {
4      initlock(&shmlock, "shmlock"); //初始化锁
5      for (int i = 0; i < 8; i++)      //初始化 shmtab
6      {
7          shmtab[i].refcount = 0;
8      }
9
10     printf("shm init finished.\n");
11 }

```

然后在 `kernel/main.c` 中的 `main()` 方法中调用该初始化函数，使得系统启动时能完成共享变量的准备工作。

代码 3-26 `main.c` 的 `main()` 中调用 `sharememinit()`

```

1  void
2  main()
3  {
4      if(cpuid() == 0) {
5          consoleinit();
6          printfinit();
7
8          kinit();          // physical page allocator
9          //printf("kinit\n");
10         vminit();          // create kernel page table
11         //printf("vminit\n");
12         procinit();        // process table
13         //printf("procinit\n");
14         trapinit();        // trap vectors
15         //printf("trapinit\n");
16         apic_init();        // set up LS7A1000 interrupt controller
17         //printf("apicinit\n");
18         extioi_init();      // extended I/O interrupt controller
19         //printf("extioi_init\n");
20         binit();           // buffer cache

```

```

21 //printf("binit\n");
22     iinit();           // inode table
23 //printf("iinit\n");
24     fileinit();        // file table
25 //printf("fileinit\n");
26     ramdiskinit();     // emulated hard disk
27
28     sharememinit();    //这里添加初始化
29 //printf("ramdiskinit\n");
30     userinit();        // first user process
31 //printf("userinit\n");
32     __sync_synchronize();
33     started = 1;
34 } else {
35     while(started == 0)
36         ;
37     __sync_synchronize();
38     printf("hart %d starting\n", cpuid());
39 }
40     scheduler();
41 }

```

除了系统层面的初始化工作外，每个进程的 PCB 中也有相关信息需要准备。该工作分成两步完成，第一步是在分配 PCB 的 `allocproc()` 时将 `p->shm` 初始化和 `p->shmkeymask` 清零；其次是在 `fork()` 时将子进程的共享内存信息从父进程那里拷贝过来。下面我们先在 `proc.c` 的 `allocproc()` 中对在 `proc.h` 新添加的属性的初始化，以便 `fork()` 时可以进一步设置。

代码 3-27 为 `proc.c` 中的 `allocproc()` 增加共享内存初始化功能

```

1  static struct proc*
2  allocproc(void)
3  {
4      struct proc *p;
5
6      ...
7
8      found:
9      p->pid = allocpid();
10     p->state = USED;
11
12     p->shm = TRAPFRAME - 64 * 2 * PGSIZE;           //初始化 shm
13
14     p->shmkeymask = 0;                               // 初始化 shmkeymask
15
16
17
18     ...
19
20     return p;
21 }

```

`proc.c` 的 `fork()` 中，我们将子进程的共享内存区从父进程复制而来，修改后的 `fork()` 如代码 3-28 所示。

代码 3-28 在 `fork()` 中复制共享内存区的信息

```

1  int
2  fork(void)
3  {
4      int i, pid;

```

```

5     struct proc *np;
6
7     ...
8
9     // Copy user memory from parent to child.
10    if(uvmcopy(p->pagetable, np->pagetable, p->sz) < 0) {
11        freeproc(np);
12        release(&np->lock);
13        return -1;
14    }
15    np->sz = p->sz;
16
17    shmaddcount(p->shmkeymask);           //fork 出新进程, 所以引用数加 1
18    np->shm = p->shm;                      //复制父进程的 shm
19    np->shmkeymask = p->shmkeymask;        //复制父进程的 shmkeymask
20
21    for(i=0; i<8; ++i) {                  //复制父进程的 shmva 数组
22        if(shmkeyused(i, np->shmkeymask)) { //只拷贝已启用的共享内存区
23            np->shmva[i] = p->shmva[i];
24        }
25    }
26
27    ...
28
29
30    return pid;
31 }
32

```

虽然我们最多只有 8 个内存区，但我们的代码仅拷贝“已使用”那几个而不是全盘拷贝，因此还需要判定这些内存区是否已启用——`shmkeyused()`，它用于判断当前进程是否持有某个共享内存。

```

1     int
2     shmkeyused(uint key, uint mask)
3     {
4         if(key<0 || 8<=key) {
5             return 0;
6         }
7         return (mask >> key) & 0x1; //这里判断对应的系统共享内存区是否已经启用
8     }

```

■ 将共享内存映射到进程空间

接下来需要在 `sharemem.c` 实现将共享内存区映射到进程空间的 `shmgetat()`。前面简单提到过，其参数 `key` 为共享内存一个下标（0~7），`num` 为需要分配页数，返回共享内存的地址。具体执行时需要分成三种情况来讨论：

（1）如果共享内存区已经在本进程空间中映射过，则直接返回该地址。也就是说我们不支持同一个共享内存区在同一进程的多次映射；

（2）如果共享内存区还未创建，也就是说系统的 `shmtab[id]` 的引用计数仍为 0，则需要创建该内存区（包括分配页帧和建立页表映射）——`allocshm()` 和 `allocuvm()` 工作原理相同，区别在于我们的内存使用是从高地址往低地址方向分配。

（3）如果系统已经有对应的共享内存，则直接映射到进程空间即可。

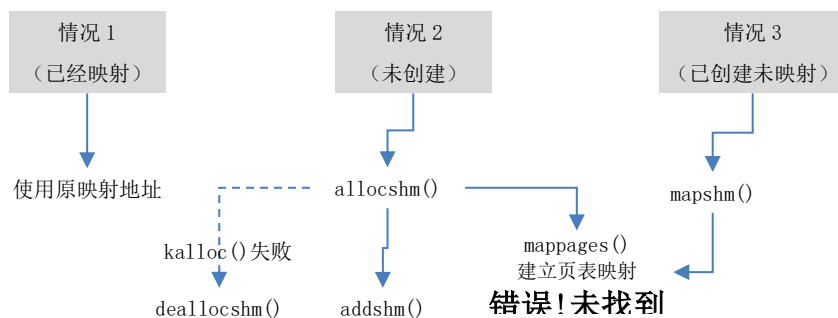


图 3-2 映射共享内存的三种情况

```

1  void*
2  shmgetat(uint key, uint num)
3  {
4      pde_t *pgdir;
5      struct proc* p;
6      void *phyaddr[MAX_SHM_PGNUM];
7      uint64 shm = 0;
8      if(key<0||8<=key||num<0||MAX_SHM_PGNUM<num) //校验参数
9          return (void*)-1;
10     acquire(&shmlock);
11     p = myproc();
12     pgdir = p->pagetable;
13     shm = p->shm;
14
15
16     // 情况 1. 如果当前进程已经映射了该 key 的共享内存，直接返回地址
17     if(p->shmkeymask>>key & 1) {
18         release(&shmlock);
19         return p->shmva[key];
20     }
21
22     // 情况 2. 如果系统还未创建此 key 对应的共享内存，则分配内存并映射
23     if(shmtab[key].refcount == 0) {
24         shm = allocshm(pgdir, shm, shm - num * PGSIZE, p->sz, phyaddr);
25         // 新增的 allocshm() 分配内存并映射，其原理和 allocuvm() 相同
26         if(shm == 0) {
27             release(&shmlock);
28             return (void*)-1;
29         }
30         p->shmva[key] = (void*)shm;
31         shmadd(key, num, phyaddr); // 将新内存区信息填入 shmtab[8] 数组
32     } else {
33
34         // 情况 3. 如果未持有且已经系统中分配此 key 对应的共享内存，则直接映射
35         for(int i = 0; i<num; i++)
36         {
37             phyaddr[i] = shmtab[key].physaddr[i];
38         }
39         num = shmtab[key].pagenum;
40         // mapshm 方法新建映射
41         if((shm = mapshm(pgdir, shm, shm-num*PGSIZE, p->sz, phyaddr))==0) {
42             release(&shmlock);
43             return (void*)-1;
44         }
45         p->shmva[key] = (void*)shm;
46         shmtab[key].refcount++; // 引用计数+1

```

```

47     }
48     p->shm = shm;
49     p->shmkeymask |= 1<<key;
50     release(&shmlock);
51     return (void*)shm;
52 }

```

其中上面的 `allocshm()` 用于创建一个共享内存区，由于 xv6 的虚存空间分配和物理页帧的分配是合并完成的，也就是说不存在未映射进程空间，因此和 `allocvm()` 相同地需要同时进行 `kalloc()` 和 `mappages()` 两个操作，具体代码如代码 3-29 所示。

代码 3-29 `allocshm()` 的实现

```

1  // 这个方法和 allocvm 实现基本一样
2  uint64
3  allocshm(pde_t *pgdir, uint64 oldshm, uint64 newshm, uint64 sz, void
    *phyaddr[MAX_SHM_PGNUM])
4  {
5      char *mem;
6      uint64 a;
7
8      if(oldshm & 0xFFF || newshm & 0xFFF || oldshm > (TRAPFRAME - 64 * 2 * PGSIZE)
        || newshm < sz)
9          return 0;
10     a = newshm;
11     // int count = 0;
12     for (int i = 0; a < oldshm; a+=PGSIZE, i++)
13     {
14         // count++;
15         mem = kalloc(); //分配物理页帧
16         if(mem == 0) {
17             printf("allocshm out of memory\n");
18             deallocshm(pgdir, newshm, oldshm);
19             return 0;
20         }
21         memset(mem, 0, PGSIZE);
22
23         mappages(pgdir, a, PGSIZE, (uint64) (mem), PTE_P|PTE_W|PTE_PLV|PTE_MAT|PTE_D);
24         //页表映射
25         phyaddr[i] = (void *) (mem);
26         printf("allocshm : %x\n", a);
27     }
28     // printf("count:%d\n", count);
29     return newshm;
30 }

```

`mapshm()` 用于支撑 `allocshm()` 函数的映射功能，其本质是对 `kernel/vm.c` 中的 `mappages()` 的封装，具体实现如代码 3-30 所示。

代码 3-30 `mapshm()`

```

1  uint64
2  mapshm(pde_t *pgdir, uint64 oldshm, uint64 newshm, uint sz, void **physaddr)
3  {
4      uint64 a;
5      if(oldshm & 0xFFF || newshm & 0xFFF || oldshm > (TRAPFRAME - 64 * 2 * PGSIZE)
        || newshm < sz)
6          return 0; // 验证参数
7      a=newshm;
8      for (int i = 0; a<oldshm;a+=PGSIZE, i++) //逐页映射

```

```

9      {
10
11      mappages(pgdir, a, PGSIZE, (uint64)physaddr[i], PTE_P|PTE_W|PTE_PLV|PTE_MAT|PTE_D);
12      }
13      return newshm;
14  }

```

shmadd()函数实现将新内存区信息填入 shmtab[8]数组的功能，其实现如代码 3-31 所示。

代码 3-31 shmadd()

```

1  int
2  shmadd(uint key, uint pagenum, void* physaddr[MAX_SHM_PGNUM])
3  {
4      if(key<0 || 8<=key || pagenum<0 || MAX_SHM_PGNUM < pagenum) {
5          return -1;
6      }
7      shmtab[key].refcount = 1;
8      shmtab[key].pagenum = pagenum;
9      for(int i = 0; i<pagenum; ++i) {
10         shmtab[key].physaddr[i] = physaddr[i];
11     }
12     return 0;
13 }

```

由于 fork()的时候，由于子进程对共享内存区进行了一次映射，因此需要对其引用计数增 1，这是通过 shmaddcount()完成的，其实现如代码 3-32 所示。

代码 3-32 shmaddcount()

```

1  void
2  shmaddcount(uint mask)
3  {
4      acquire(&shmlock);
5      for (int key = 0; key < 8; key++)
6      {
7          if(shmkeyused(key, mask)) { //对目前进程所有引用的共享内存的引用数加 1
8              shmtab[key].refcount++;
9          }
10     }
11     release(&shmlock);
12 }

```

在 allocshm()创建共享内存区时如果因为未能分配到物理页帧而失败时，需要调用 deallocshm()将已经分配的物理页帧归还。deallocshm()工作原理和 dealloctvm()相同，具体代码如代码 3-33 所示。

代码 3-33 deallocshm()

```

1  uint64
2  deallocshm(pde_t *pgdir, uint64 oldshm, uint64 newshm)
3  {
4      if(newshm <= oldshm)
5          return oldshm;
6
7      if(PGROUNDUP(newshm) > PGROUNDUP(oldshm)) {
8          int npages = (PGROUNDUP(newshm) - PGROUNDUP(oldshm)) / PGSIZE;
9          uvmunmap(pgdir, PGROUNDUP(oldshm), npages, 1);
10     }
11
12     return oldshm;
13 }

```

■ 解除共享内存区的映射

`shmrelease()`用于解除共享内存的映射，可以是主动调用，也可以是在每个进程退出时调用。`shmrelease()`把当前进程中引用的共享内存的 `refcount` 减 1，同时调用 `deallocshm()`解除页表映射；若有共享内存区的 `refcount` 变为 0，还要调用 `shrmr()`释放物理内存从而彻底销毁该共享内存。具体代码实现如代码 3-34 所示，其中的 `deallocshm()`的实现请参见前面的代码 3-33。

代码 3-34 `shmrelease()`

```

1  int
2  shmrelease(pde_t *pgdir, uint64 shm, uint keymask)
3  {
4      //cprintf("shmrelease: shm is %x, keymask is %x. \n", shm, keymask);
5      acquire(&shmlock);
6      deallocshm(pgdir, shm, TRAPFRAME - 64 * 2 * PGSIZE);           //释放用户
                                空间的内存
7      for (int k=0; k<8; k++)
8      {
9          if(shmkeyused(k, keymask)) {
10             shmtab[k].refcount--;                                //引用数目减 1
11             if(shmtab[k].refcount==0) {                          //若为 0，即可以回收物理内存
12                 shrmr(k);
13             }
14         }
15     }
16     release(&shmlock);
17     return 0;
18 }
```

`shrmr()`用于将共享内存的物理页帧回收，从而彻底清除共享内存区——利用 `kfree()`逐个释放共享内存对应的物理页帧，如代码 3-35 所示。

代码 3-35 `shrmr()`

```

1  int
2  shrmr(int key)
3  {
4      if(key<0 || 8<=key) {
5          return -1;
6      }
7      //cprintf("shrmr: key is %d\n", key);
8      struct sharemem* shmem = &shmtab[key];
9      for(int i=0; i<shmem->pagenum; i++) {
10         kfree((char*)(shmem->physaddr[i]));                      //逐个页帧回收
11     }
12     shmem->refcount = 0;
13     return 0;
14 }
```

如果进程没有主动撤销共享内存的映射，那么在进程结束时父进程 `wait()`操作中解除映射以及可能回收页帧，如代码 3-36 所示。

代码 3-36 修改后的 `kernel/proc.c wait()`

```

1  int
2  wait(uint64 addr)
3  {
4      struct proc *np;
5      int havekids, pid;
6      struct proc *p = myproc();
7
8      acquire(&wait_lock);
```

```

9
10     for(;;) {
11         // Scan through table looking for exited children.
12         havekids = 0;
13         for(np = proc; np < &proc[NPROC]; np++) {
14             if(np->parent == p) {
15                 // make sure the child isn't still in exit() or swtch().
16                 acquire(&np->lock);
17
18                 havekids = 1;
19                 if(np->state == ZOMBIE) {
20                     // Found one.
21                     pid = np->pid;
22                     if(addr != 0 && copyout(p->pagetable, addr, (char *)&np->xstate,
23                                             sizeof(np->xstate)) < 0) {
24                         release(&np->lock);
25                         release(&wait_lock);
26                         return -1;
27                     }
28                     shmrelease(np->pagetable, np->shm, np->shmkeymask); // 解除共享内存映
射
29                     np->shm = TRAPFRAME - 64 * 2 * PGSIZE;           //把 shm 重置
30                     np->shmkeymask = 0; //把 shmkeymask 重置
31                     freeproc(np);
32                     release(&np->lock);
33                     release(&wait_lock);
34                     return pid;
35                 }
36                 release(&np->lock);
37             }
38         } ...
39     }

```

同理，在 `exec()` 更换进程映像时，在清理原有进程映像时也需要解除共享内存的映射。修改后的 `exec()` 如代码 3-37 所示。

代码 3-37 修改后的 `kernel/exec.c` `exec()`

```

1  int
2  exec(char *path, char **argv)
3  {
4      ...
5      safestrcpy(p->name, last, sizeof(p->name));
6
7      // Commit to the user image.
8      oldpagetable = p->pagetable;
9      p->pagetable = pagetable;
10     p->sz = sz;
11     p->trapframe->era = elf.entry; // initial program counter = main
12     p->trapframe->sp = sp; // initial stack pointer
13
14     shmrelease(oldpagetable, p->shm, p->shmkeymask); //回收共享内存
15
16     proc_freepagetable(oldpagetable, oldsz);
17
18     p->shm = TRAPFRAME - 64 * 2 * PGSIZE; //重置 shm
19     p->shmkeymask = 0; //重置 shmkeymask
20     return 0;
21     ...
22 }

```


■ shmrefcount 系统调用

shmrefcount 系统调用返回共享内存的引用数。首先在 sharemem.c 中添加 shmrefcount() 代码:

```
1  int
2  shmrefcount(uint key)
3  {
4      acquire(&shmlock);
5      int count;
6      count = (key<0 || 8<=key)? -1:shmtab[key].refcount;
7      release(&shmlock);
8      return count;
9  }
```

还需要在 sysproc.c 中添加 sys_shmrefcount() 代码, 与上文的 sys_shmgetat() 一起加入。

```
1  uint64
2  sys_shmgetat (void)
3  {
4      int key, num;
5      if(argint(0, &key) < 0 || argint(1, &num) < 0)
6          return -1;
7      return (uint64) shmgetat(key, num);
8  }
9
10 int
11 sys_shmrefcount(void)
12 {
13     int key;
14     if(argint(0, &key) < 0)
15         return -1;
16     return shmrefcount(key);
17 }
```

此外与第 2 章所描述的添加系统调用方法一样, 需要在多个文件中添加相关代码, 此处不一一展开。

测试验证

测试代码将创建一个子进程, 父子进程都映射编号为 1 的共享内存块, 父、子进程先后往共享内存写入消息字符串, 再将共享内存中的字符串打印出来, 并打印进程 pid 和共享内存块 1 的引用数, 从而验证两个进程访问到同一内存区域这一个功能。该测试代码的关键部分如代码 3-38 所示。

代码 3-38 进程间使用共享内存通信的示例代码

```
1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4  #include "kernel/fs.h"
5  int main(void)
6  {
7      char *shm;
8      int pid = fork();
9      if(pid == 0) {
10         sleep(1);
```

```

11     shm = (char*)shmgetat(1,3);//key 为 1, 大小为 3 页的共享内存
12     printf("child process pid:%d shm is %s refcount of 1 is:%d\n", getpid(),
        shm, shmrefcount(1));
13     strcpy(shm, "hello_world!");
14     printf("child process pid:%d write %s into the shm\n", getpid(), shm);
15 } else if (pid > 0) {
16     shm = (char*)shmgetat(1,3);
17     printf("parent process pid:%d before wait() shm is %s refcount of 1 is:%d\n",
        getpid(), shm, shmrefcount(1));
18     strcpy(shm, "share_memory!");
19     printf("parent process pid:%d write %s into the shm\n", getpid(), shm);
20     wait(0);
21     printf("parent process pid:%d after wait() shm is %s refcount of 1 is:%d\n",
        getpid(), shm, shmrefcount(1));
22 }
23 exit(0);
24 }

```

编译后运行, 父进程成功读出子进程写入的数据, 从而展示了父子进程正常使用共享内存提供的通信功能。

屏显 3-7 父子进程通过共享内存进行通信

```

$ test
parent process pid:3 before wait() shm is refcount of 1 is:1
parent process pid:3 write share_memory! into the shm
child process pid:4 shm is share_memory! refcount of 1 is:2
child process pid:4 write hello_world! into the shm
parent process pid:3 after wait() shm is hello_world! refcount of 1 is:1

```

3.3.2. 消息队列的实现

本科操作系统教材中通常都有讨论消息队列的实现原理, 也就是在各个进程内部实现一个邮箱, 可以接受其他进程发过来的消息。这里实现了一个简单的消息队列, 在进程 `proc` 中添加一个成员用于记录消息队列。出于简化编程的目的, 一个消息队列不管长度为多少, 都用一个页帧大小来管理。由于涉及多进程并发操作, 还需要注意同步关系。

关键数据结构

为了给 `xv6` 进程增加消息队列的通信能力, 需要描述消息和消息队列自身形成如图 3-3 所示的系统。消息队列使用 `msg` 结构体描述一个消息, 其定义如代码 3-39 所示, 所有消息构成一个链表。其中第一个消息, 仅作为表头一直存在, 并不保存消息。

由于是一种简化的实现, 整个系统只有 8 个消息队列 `mqs[MQMAX]`, 每个消息队列使用 `key` 成员作为身份标识——不同的进程只要指定相同的 `key` 值就使用相同的消息队列²。

² 这里的简易实现并没有检查 `key` 值是否唯一, Linux 通常使用文件的索引节点号来产生唯一标识。

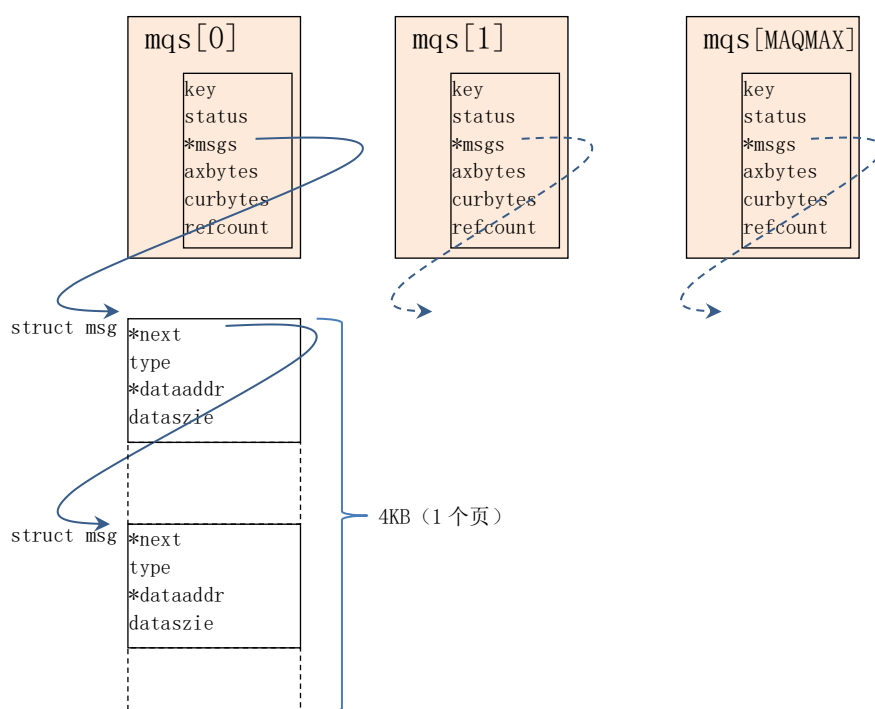


图 3-3 一种 xv6 的消息队列方案

由于是共享资源，因此使用 `mqlock` 锁保护消息队列上的操作，另有 `rqueue[NPROC]` 和 `wqueue[NPROC]` 分别对应读写阻塞队列。在 `kernel` 目录新建一个 `messagequeue.c` 文件用于描述上述数据对象，如代码 3-39 所示。

代码 3-39 消息 msg 结构体

```

1  struct msg {           //消息结构体
2      struct msg *next;   //指向下一个消息
3      long type;          // 消息类型
4      char *dataaddr;     //数据地址，实际上等与 msg+16
5      int datasize;       //消息长度
6  };
7
8  struct mq {            //消息队列
9      int key;            //对应的 key
10     int status;         //0 代表未使用，1 代表已使用
11     struct msg *msgs;   //指向 msg 链表
12     int maxbytes;       //一个消息队列最大为 4k
13     int curbytes;       //当前已使用字节数
14     int refcount;       //引用数（进程数）
15 };
16
17 struct spinlock mqlock; //消息队列 锁
18 struct mq mqs[MAQMAX]; //默认系统最多 8 个消息队列
19 struct proc* wqueue[NPROC]; //写阻塞队列
20 int wstart=0;              //写阻塞队列指示下标
21
22 struct proc* rqueue[NPROC]; //读阻塞队列
23 int rstart=0;              //读阻塞队列指示下标

```

同时在 `defs.h` 添加消息队列的操作函数，如代码 3-40 所示。其中 `mqinit()` 用于系统启动时对消息队列初始化，`mqget()` 用于申请使用消息队列，`msgsnd` 用于向消息队列发出消息，`msgrcv()` 用于从消息队列接收消息。

代码 3-40 消息队列的操作函数

```

1 //messagequeue.c
2 void    mqinit();                //初始化系统的消息队列
3 int      mqget(uint);            //申请使用某个消息队列
4 int      msgsnd(uint, int, int, char *); //发送消息
5 int      msgrcv(uint, int, int, uint64); //接收消息
6 void     releasemq(uint);
7 void     releasemq2(int);        //释放消息队列
8 void     addmqcount(uint);       //增加消息队列的引用计数

```

然后在 `proc.h` 的进程 PCB `proc` 结构体中添加一个 `mqmask` 成员，以掩码形式记录本进程所使用的消息队列。

```
1 uint mqmask;
```

最后再添加 `int mqget(uint); int msgsnd(uint, int, int, char*); int msgrcv(uint, int, int, uint64);` 这三个系统调用，系统调用的添加已经在第 2 章中讨论过，不再赘述。

实现细节

消息队列的实现涉及初始化、创建和撤销消息队列、消息的发送和接收等操作，下面逐个进行分析。

■ 初始化

首先需要在 `messagequeue.c` 实现消息队列的初始化函数 `mqinit()`，主要是将各个消息队列设置为空闲未用 `msq[i].status=0`，以及锁的初始化。

```

1 void
2 mqinit()
3 {
4     cprintf("mqinit.\n");
5     initlock(&mqlock, "mqlock");
6     for(int i=0; i<MQMAX; ++i) {
7         msq[i].status = 0;
8     }
9 }

```

消息队列的初始化将在系统启动时，由 `kernel/main.c` 中的 `main()` 函数调用，修改后的 `main()` 如所示。

```

1 void
2 main()
3 {
4     if(cpuid() == 0) {
5         consoleinit();
6         printfinit();
7
8         kinit(); // physical page allocator
9         //printf("kinit\n");
10        vminit(); // create kernel page table
11        //printf("vminit\n");
12        procinit(); // process table
13        //printf("procinit\n");
14        trapinit(); // trap vectors

```

```

15 //printf("trapinit\n");
16     apic_init();      // set up LS7A1000 interrupt controller
17 //printf("apicinit\n");
18     extioi_init();    // extended I/O interrupt controller
19 //printf("extioi_init\n");
20     binit();          // buffer cache
21 //printf("binit\n");
22     iinit();          // inode table
23 //printf("iinit\n");
24     fileinit();       // file table
25 //printf("fileinit\n");
26     ramdiskinit();    // emulated hard disk
27 //printf("ramdiskinit\n");
28     mqinit();         // mqinit 初始化消息队列
29     userinit();       // first user process
30 //printf("userinit\n");
31     __sync_synchronize();
32     started = 1;
33 } else {
34     while(started == 0)
35         ;
36     __sync_synchronize();
37     printf("hart %d starting\n", cpuid());
38 }
39 scheduler();
40 }

```

新进程创建过程中执行 `allocproc()` 时，需要对 `proc` 中的消息队列成员设置初值，即 `p->mqmask=0`，表示还未使用消息队列，如代码 3-41 所示。

代码 3-41 修改后的 `kernel/proc.c allocproc()`

```

1  static struct proc*
2  allocproc(void)
3  {
4      ...
5
6  found:
7      p->pid = allocpid();
8      p->state = USED;
9
10     // Allocate a trapframe page.
11     if((p->trapframe = (struct trapframe *)kalloc()) == 0) {
12         freeproc(p);
13         release(&p->lock);
14         return 0;
15     }
16
17     // An empty user page table.
18     p->pagetable = proc_pagetable(p);
19     if(p->pagetable == 0) {
20         freeproc(p);
21         release(&p->lock);
22         return 0;
23     }
24
25     p->mqmask = 0;    //初始化 mqmask
26     ...
27
28     return p;

```

29 }

当使用 `fork()` 创建进程时, 子进程的消息队列将继承自父进程, 因此也需要作相应的修改, 如代码 3-42 所示。

代码 3-42 修改后的 `kernel/proc.c fork()`

```

1  int
2  fork(int back)
3  {
4  ...
5  // copy saved user registers.
6  *(np->trapframe) = *(p->trapframe);
7
8  // Cause fork to return 0 in the child.
9  np->trapframe->a0 = 0;
10
11  addmqcount(p->mqmask);    //父进程持有的消息队列引用数全部加 1
12  np->mqmask = p->mqmask;   //复制父进程的 mqmask
13
14  // increment reference counts on open file descriptors.
15  for(i = 0; i < NOFILE; i++)
16      if(p->ofile[i])
17          np->ofile[i] = filedup(p->ofile[i]);
18  np->cwd = idup(p->cwd);
19
20  ...
21  }
```

前面 `fork()` 复制父进程的消息队列时使用了 `addmqcount()`, 它用于父进程 `mqmask` 标示的消息队列引用计数加 1。其中 `MQMAX` 宏需要在 `kernel/param.h` 中定义。

```

1  void
2  addmqcount(uint mask)
3  {
4  //cprintf("addcount: %x. \n", mask);
5  acquire(&mqlock);
6  for (int key = 0; key < MQMAX; key++)
7  {
8      if(mask >> key & 1) {
9          mqs[key].refcount++;
10     }
11 }
12 release(&mqlock);
13 }
```

■ 创建消息队列

进程在使用消息队列之前, 需要先调用 `mqget()` 创建消息队列。由于消息队列是共享的, 因此有可能进程发出创建操作时, 别的进程已经提前创建好了对应的消息队列, 这时直接使用即可。前面提到过, 消息队列使用 `key` 作为标识, 因此创建的时候就需要指出 `key` 值。`mqget()` 将根据 `key` 值在 `mqs[]` 数组中查找, 如果找到匹配的, 说明该消息队列已经有其他进程创建, 本进程直接使用即可。否则, 通过 `newmq()` 在 `mqs[]` 中找一个空闲项并使用之。

```

1  int
2  mqget(uint key)
3  {
4      struct proc *proc = myproc();
5
6      acquire(&mqlock);
```

```

7     int idx = findkey(key);
8     if(idx != -1) {
9         if(!(proc->mqmask >> idx & 1)) {
10             proc->mqmask |= 1 << idx;
11             mqs[idx].refcount++;
12         }
13         release(&mqlock);
14         return idx;
15     }
16     // 对应 key 消息队列未创建则 newmq
17     // 创建
18     idx = newmq(key);
19     // 创建消息队列
20     release(&mqlock);
21     return idx;
22     // 返回该消息队列在 mqs [] 中的下标
23 }
24

```

mqget()中用到的 findkey()函数实现非常简单,就是遍历 mqs[]数组,比对是否有相同的 key 值,从而把传入的 key 映射到一个 mqs[]数组的下标。

```

1  int findkey(int key)
2  {
3      int idx=-1;
4      for(int i=0; i<MQMAX; ++i) {
5          if(mqs[i].status != 0 && mqs[i].key == key) {
6              idx = i;
7              break;
8          }
9      }
10     return idx;
11 }

```

另外,当没有找到与 key 匹配的消息队列时,需要调用 newmq()创建该消息队列,其代码如代码 3-43 所示。从代码中可以看出,只分配了一个页的空间来存储本消息队列的消息,系统的共有 MQMAX 个消息队列,因此一共用了 MQMAX 个页来存储整个系统的全部消息。

代码 3-43 newmq()

```

1  int newmq(int key)
2  {
3      struct proc *proc = myproc();
4      int idx=-1;
5      for(int i=0; i<MQMAX; ++i) {
6          if(mqs[i].status == 0) {
7              idx = i;
8              break;
9          }
10     }
11     if(idx == -1) {
12         //消息队列全部用满, 创建失败
13         printf("newmq failed: can not get idx. \n");
14         return -1;
15     }
16     mqs[idx].msgs = (struct msg*)kalloc();
17     //为消息池分配空间 (1 个页)
18     if(mqs[idx].msgs == 0) {
19         //消息的存储空间不能为 NULL
20         printf("newmq failed: can not alloc page. \n");
21         return -1;
22     }
23     mqs[idx].key = key;
24     //为该消息队列设置 key 值
25     mqs[idx].status = 1;
26     //标示为已启用
27     memset(mqs[idx].msgs, 0, PGSIZE);
28     //清空消息池
29     mqs[idx].msgs->next = 0;
30     //接下来都是初始化消息队列

```

```

24     mqs[idx].msgs->datasize = 0;
25     mqs[idx].maxbytes = PGSIZE;
26     mqs[idx].curbytes = 16;
27     mqs[idx].refcount = 1;
28     proc->mqmask |= 1 << idx;    //修改当前进程的mqmask，表示使用中
29     return idx;
30 }

```

■ 消息存储空间管理

每个消息队列只使用一个页的内存空间，各个消息都存储于该页的 4kB 空间内。消息队列中消息储存方式如图 3-4 所示，结合代码 3-39 中的 struct msg 来分析其存储管理。一个消息结构体 msg 由成员 next、type、dataaddr 和 datasize，以及紧随其后的数据区组成，数据区大小即为 datasize。所以，一个消息总共占用 datasize+16 字节的空间。

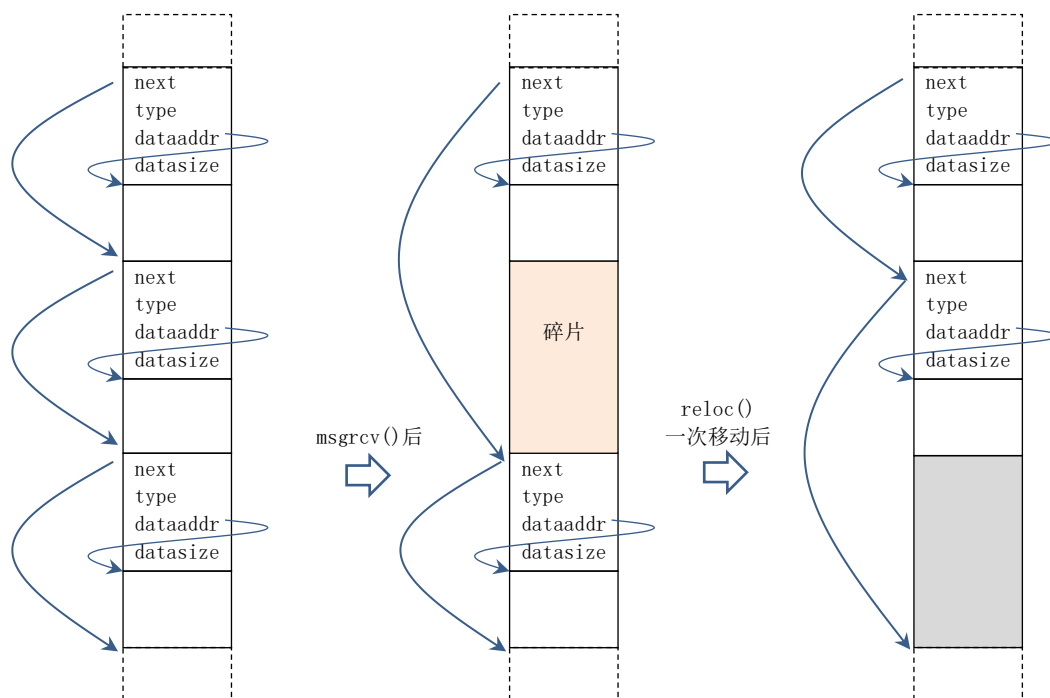


图 3-4 消息的存储示意图

如果简单地采用这种上述布局有个弊端：当队列中间有消息被取出时，会造成内存碎片，如图 3-4 中间所示。如果在每次取出消息时，都调用一次 reloc() 方法，把后面消息往前移动填补碎片（类似 jvm 的 GC 中的标记-整理），那么可以在这 4kB 的空间内容纳更多的消息。reloc() 整理函数的实现如代码 3-44 所示。全空的消息队列，是不会发出 reloc() 调用的。

代码 3-44 reloc()

```

1  int reloc(int mqid)
2  {
3      struct msg *pages = mqs[mqid].msgs;    //移动消息目标地址
4      struct msg *m = pages;                  //待移动消息
5      struct msg *t;
6      struct msg *pre = pages;
7      while (m != 0)
8      {
9          t = m->next;                          //提前保存下一个待移动消息的地址
10
11         memmove(pages, m, m->datasize+32);    //移动消息（包括原地拷贝）

```



```

12     pages->next = (struct msg *)((char *)pages + pages->datasize + 32);
    // 修改下个消息指针
13     pages->dataaddr = ((char *)pages + 32);    //修改数据指针
14     pre = pages;                                //记录当前消息指针
15     pages = pages->next;                        //下个消息的目标位置 (目的)
16     m = t;                                      //下一个待移动消息 (源)
17 }
18 pre->next = 0;                                //最后一个消息的 next 指针置 0
19 return 0;
20 }
21

```

■ 发送和接收

发送消息使用 `msgsnd()`，需要指出使用哪个消息队列、消息类型、消息缓冲区首地址以及消息长度四个参数，如代码 3-45 所示。

代码 3-45 `msgsnd()`

```

1  int
2  msgsnd(uint mqid, int type, int sz, char *msg)
3  {
4      struct proc *proc = myproc();
5      if(mqid<0 || MQMAX<=mqid || mqs[mqid].status == 0) {
6          return -1;
7      }
8
9      // char *data = msg;
10     // printf("data:%s\n", data);
11
12     if(mqs[mqid].msgs == 0) {
13         printf("msgsnd failed: msgs == 0.\n");
14         return -1;
15     }
16
17     acquire(&mqlock);
18
19     while(1) {                                //一直循环直到发送成功
20         if(mqs[mqid].curbytes + sz + 16 <= mqs[mqid].maxbytes) { //如果剩余空间
充裕
21             struct msg *m = mqs[mqid].msgs;
22             while(m->next != 0) {              //找到队尾最后一个空闲消息区
23                 m = m->next;
24             }                                  //退出循环时，m->next==0 标示空闲消
息区
25             m->next = (void *)m + m->datasize + 32; //计算用于存储消息的起始位
置
26             m = m->next;                        //m 为本消息存储空间起点
27             m->type = type;                      //填写本消息 type
28             m->next = 0;                        //本消息暂无后续消息
29             m->dataaddr = (char *)m + 32;      //数据区
30             m->datasize = sz;                  //数据长度
31
32             memmove(m->dataaddr, msg, sz);    //拷贝消息数据
33
34
35             mqs[mqid].curbytes += (sz+32);    //可用空间缩减
36
37             for(int i=0; i<rstart; i++)        //唤醒所有读阻塞进程
38             {

```

```

39         wakeup(rqueue[i]);
40     }
41     rstart = 0;                //读阻塞队列置空
42
43     release(&mqlock);
44     return 0;
45 } else {                      //空间不足, 进程睡眠在 wqueue 阻塞队
列
46     printf("msgsnd: can not alloc: pthread: %d sleep.\n", proc->pid);
47     wqueue[wstart++] = proc;    //环形队列
48
49     sleep(proc, &mqlock);
50 }
51
52 }
53
54 return -1;
55 }

```

接收消息使用 `msgrcv()` 函数, 需要指出消息队列的 `id`、消息类型、消息缓冲区和消息大小。其工作原理和发送过程类似, 但是多了一个消息紧凑的内存操作——前面刚讨论过的 `reloc()`。

代码 3-46 `msgrcv()`

```

1  int
2  msgrcv(uint mqid, int type, int sz, uint64 addr)
3  {
4      struct proc *proc = myproc();
5      if(mqid<0 || MQMAX<=mqid || mqs[mqid].status ==0) {
6          return -1;
7      }
8
9
10
11     acquire(&mqlock);
12
13     while(1) {
14         struct msg *m = mqs[mqid].msgs->next;
15         struct msg *pre = mqs[mqid].msgs;
16         while (m != 0)
17         {
18             if(m->type == type) {          //找到要读取的消息类型
19                 copyoutstr(proc->pagetable, addr, m->dataaddr, sz);
20
21                 pre->next = m->next;        //将已读取的消息从消息队列中删除
22                 mqs[mqid].curbytes -= (m->datasize + 32); //释放消息空间
23                 reloc(mqid);                //重新整理内存
24
25                 for(int i=0; i<wstart; i++) //唤醒写阻塞进程
26                 {
27                     wakeup(wqueue[i]);
28                 }
29                 wstart = 0;                //写阻塞队列置空
30
31                 release(&mqlock);
32                 return 0;
33             }
34             pre = m;

```

```

35         m = m->next;
36     }
37     printf("msgrcv: can not read: pthread: %d sleep.\n", proc->pid);
38     rqueue[rstart++] = proc;
39     sleep(proc, &mqlock);
40 }
41 return -1;
42 }

```

msgrcv()中使用到的 copyoutstr()函数实现需要在 kernel/vm.c 中添加,同时还需要在 defs.h 中声明该函数原型 “int copyoutstr(pagetable_t, uint64, char *, uint64);”

```

1  int
2  copyoutstr(pagetable_t pagetable, uint64 dstva, char *src, uint64 len)
3  {
4      uint64 n, va0, pa0;
5
6      int got_null = 0;
7      while(len > 0) {
8          va0 = PGROUNDDOWN(dstva);
9          pa0 = walkaddr(pagetable, va0);
10         if(pa0 == 0)
11             return -1;
12         n = PGSIZE - (dstva - va0);
13         if(n > len)
14             n = len;
15         char *p = (char *)((pa0 + (dstva - va0)) | DMWIN_MASK);
16         while(n > 0) {
17             if(*src == '\0') {
18                 *p = '\0';
19                 got_null = 1;
20                 break;
21             } else {
22                 *p = *src;
23             }
24             --n;
25             --len;
26             p++;
27             src++;
28         }
29
30
31
32         len -= n;
33         src += n;
34         dstva = va0 + PGSIZE;
35     }
36
37     if(got_null) {
38         return 0;
39     } else {
40         return -1;
41     }
42 } //todo

```

可以看出这个简化实现的消息队列,在接收消息也需要指定消息长度,读者可以将该约束去除,读入消息后再返回消息长度给调用者。另外,读者也可以将等待队列的实现方式修改为链表方式,比现在的用 rstart/rend 的首尾指针方式要合理得多。

读者还需要注意到，上面的代码直接将内核态数据拷贝到用户态空间，而没有检查用户空间是否合法可用，读者可以自行增加上述检查。

■ 撤销消息队列

当不再使用消息队列时，可以撤销消息队列。需要借助 `refcount` 来管理引用，引用为 0 时回收内存，否则仍将保留消息队列的内存空间。撤销操作由 `releasemq()` 负责，具体如代码 3-47 所示，如果没有进程使用该消息队列，则进一步通过 `rmmq()` 撤销其内存。另有 `releasemq2()` 用于将一个进程的所用的全部消息队列撤销。

代码 3-47 `releasemq()/rmmq()/releasemq2()`

```

1  void
2  rmmq(int mqid)
3  {
4      //cprintf("rmmq: %d.\n", mqid);
5      kfree((char *)mqs[mqid].msgs); //回收物理内存
6      mqs[mqid].status = 0;
7  }
8
9  void
10 releasemq(uint key)
11 {
12     //cprintf("releasemq: %d.\n", key);
13     int idx = findkey(key);
14     if (idx != -1) {
15         acquire(&mqlock);
16         mqs[idx].refcount--; //引用数目减 1
17         if (mqs[idx].refcount == 0) //引用数目为 0 时候需要回收物理内存
18             rmmq(idx);
19         release(&mqlock);
20     }
21 }
22
23 void
24 releasemq2(int mask)
25 {
26     //cprintf("releasemq: %x.\n", mask);
27     acquire(&mqlock);
28     for (int id = 0; id < MQMAX; ++id) {
29         if (mask >> id & 0x1) {
30             mqs[id].refcount--; //引用数目减 1
31             if (mqs[id].refcount == 0) { //引用数目为 0 时候需要回收物理内存
32                 rmmq(id);
33             }
34         }
35     }
36     release(&mqlock);
37 }

```

除了进程主动调用外，其他两个调用时机是进程结束时和 `exec` 更换进程映像时。对于进程结束后，父进程执行 `wait()` 操作时，需要将子进程所使用消息队列撤销。

代码 3-48 修改后的 `kernel/proc.c wait()`

```

1  int
2  wait(void)
3  {
4      ...

```

```

5   for(;;) {
6       // Scan through table looking for exited children.
7       havekids = 0;
8       for(np = proc; np < &proc[NPROC]; np++) {
9           if(np->parent == p) {
10              // make sure the child isn't still in exit() or swtch().
11              acquire(&np->lock);
12
13              havekids = 1;
14              if(np->state == ZOMBIE) {
15                  // Found one.
16                  pid = np->pid;
17                  if(addr != 0 && copyout(p->pagetable, addr, (char *)&np->xstate,
18                                          sizeof(np->xstate)) < 0) {
19                      release(&np->lock);
20                      release(&wait_lock);
21                      return -1;
22                  }
23                  releasemq2(np->mqmask); //回收消息队列
24                  np->mqmask = 0;         //重置进程的 mqmask
25                  freeproc(np);
26                  release(&np->lock);
27                  release(&wait_lock);
28                  return pid;
29              }
30              release(&np->lock);
31          }
32      } ...
33  }
34  }

```

另一处需要撤销消息队列的使用的地方是 `exec()` 更换进程映像的时候，如代码 3-49 所示。

代码 3-49 修改后的 `kernel/exec.c` `exec()`

```

1   int
2   exec(char *path, char **argv)
3   {
4       ...
5
6       oldpagetable = p->pagetable;
7       p->pagetable = pagetable;
8       p->sz = sz;
9       p->trapframe->era = elf.entry; // initial program counter = main
10      p->trapframe->sp = sp; // initial stack pointer
11
12
13      proc_freepagetable(oldpagetable, oldsz);
14
15      releasemq(p->mqmask); //回收消息队列
16      p->mqmask = 0;        //重置进程的 mqmask
17
18
19      return argc; // this ends up in a0, the first argument to main(argc, argv)
20      return 0;
21      ...
22  }

```

测试验证

下面代码 3-50 给出了测试发送和接受消息的核心代码，读者自行尝试编写完成的代码进行测试。

代码 3-50 测试消息队列发送和接受功能的代码

```

1  #include "kernel/param.h"
2  #include "kernel/types.h"
3  #include "kernel/stat.h"
4  #include "user.h"
5  #include "kernel/fs.h"
6  #include "kernel/fcntl.h"
7  #include "kernel/syscall.h"
8  // #include "traps.h"
9  #include "kernel/memlayout.h"
10
11
12  struct msg{
13      int type;
14      char *dataaddr;
15  } s1, s2, g;
16
17  void msg_test()
18  {
19      int mqid = mqget(123);          //使用消息队列
20      // int msg_len = 48;
21      // s1.dataaddr = "total number:47 : hello, this is child process.";
22      int pid = fork();
23      if(pid == 0) {                  //子进程
24          s1.type = 1;
25          s1.dataaddr = "This is the first message!\n";
26
27
28
29          msgsnd(mqid, s1.type, 28, s1.dataaddr); //发送消息 1
30
31          s1.type = 2;
32          s1.dataaddr = "Hello, another message comes!\n";
33          msgsnd(mqid, s1.type, 31, s1.dataaddr); //发送消息 2
34          s1.type = 3;
35          s1.dataaddr = "This is the third message, and this message has great many
characters!\n";
36          msgsnd(mqid, s1.type, 72, s1.dataaddr); //发送消息 3
37
38          printf("all messages have been sent.\n");
39      } else if (pid > 0)              //以下是父进程
40      {
41
42          sleep(10);                  // sleep 保证子进程消息写入
43          g.dataaddr = malloc(70);
44          g.type = 2;
45          msgrcv(mqid, g.type, 31, (uint64)g.dataaddr); //读入消息 2
46          printf("receive the %dth message: %s\n", 2, g.dataaddr);
47          g.type = 1;
48          msgrcv(mqid, g.type, 28, (uint64)g.dataaddr); //读入消息 1
49          printf("receive the %dth message: %s\n", 1, g.dataaddr);
50          g.type = 3;
51          msgrcv(mqid, g.type, 70, (uint64)g.dataaddr); //读入消息 3

```

```

52     printf("receive the %dth message: %s\n", 3, g.dataaddr);
53
54     wait(0);
55
56 }
57 exit(0);
58 }
59
60
61 int
62 main(int argc, char *argv[])
63 {
64     // printf(1, "消息队列测试\n");
65     msg_test();
66     exit(0);
67 }

```

启动系统后运行 `msg_test` 测试程序，获得如屏显 3-8 所示的输出。

屏显 3-8 父进程睡眠等待子进程发出消息后读取的消息

```

init: starting sh
$ msg_test
all messages have been sent.
receive the 2th message: Hello, another message comes!

receive the 1th message: This is the first message!

receive the 3th message: This is the third message, and this message has great many characters!

```

注释掉 `sleep` 后很大概率父进程在子进程还没有写之前读，所以会阻塞，等子进程写后再唤醒输出。

屏显 3-9 父进程读取子进程消息（有睡眠等待）

```

init: starting sh
$ msg_test
msgrcv: can not read: pthread: 3 sleep.
all messages have been sent.
receive the 2th message: Hello, another message comes!

receive the 1th message: This is the first message!

receive the 3th message: This is the third message, and this message has great many characters!

```

3.4. 内存管理实验

我们这里安排两个实验，第一个实现进程用户空间的非连续的分区分配（实际上以页为最小分配尺度），第二个实验将进程的内存布局分成代码和数据两个属性不同的段，从而保护代码不会受到意外修改。由于虚拟内存还需要磁盘文件系统的支持，因此虚存的实验将放到高级实验部分讨论。

3.4.1. 实现 `myfree()` 和 `myalloc()` 系统调用

前面学习过 `xv6` 进程分配和释放内存的方式后，发现 `xv6` 只允许扩展和收缩进程空间。具体来说，`xv6` 进程空间的内存管理方法：

（1）从 `kernel/sysproc.c` 中的 `sys_sbrk()` 系统调用入手，可以看到 `sys_sbrk()` 将进一步调用 `growproc(n)` 进行内存空间的调整。`kernel/proc.c` 中的 `growproc(n)` 根据 `n` 的正负不同，分别利用 `allocvm()` 进行扩展或 `deallocvm()` 进行收缩。

(2) kernel/vm.c 中的 `allocvm()` 和 `deallocvm()` 则是进程空间管理的核心代码。学习进程分配内存的 `allocvm()` 和 `deallocvm()` 时, 注意要明确区分: 物理页帧、页表、虚存地址范围, 以及三者之间的关系。其中 `kalloc()` 将分配一个物理页帧, `kfree()` 将释放一个页帧; `mappages()` 用于将建立虚存地址和物理页帧之间的页表映射。

对 xv6 的进程空间的扩展和收所有了解之后, 可以思考如何实现 Linux 方式的 `alloc()` 和 `free()`, 因为它们可能造成内存空间中的孔洞, 而 xv6 当前并不支持这样的内存布局。

内存空间描述

我们对内存管理作了一些限制, 以避免 `myalloc()/myfree()` 机制和 `sbrk` 机制相冲突——启用 `vma` 内存区域之后, 不再允许 `sbrk` 发出内存扩展的请求。也就是说, `myalloc()/myfree()` 紧接在原来的进程映像结束位置, 安排在 `proc->sz` 之上的空间, 如图 3-5 所示。

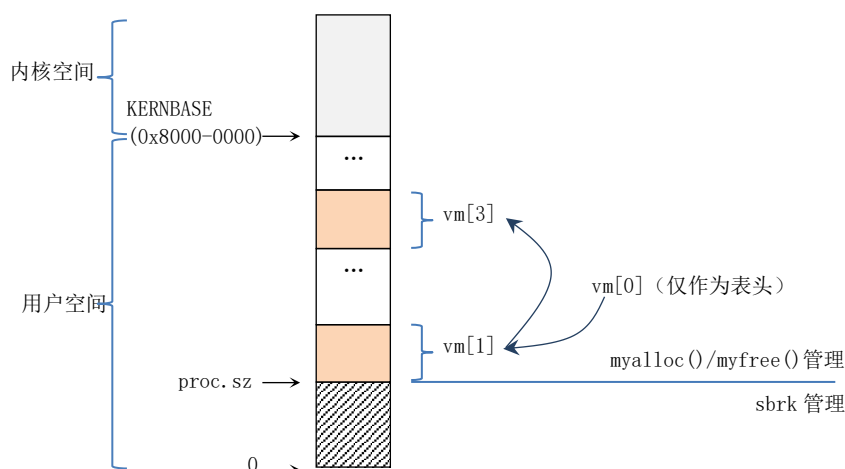


图 3-5 一种 xv6 的 vma 管理方案示意图

■ 修改 PCB

xv6 的进程空间只有一个连续区间, 只允许扩展和收缩两个操作, 因此只需要一个 `sz` 成员就可以记录。如果实现类似 Linux 操作系统的内存分配 `alloc()` 和释放 `free()` 操作, 那么就可能在进程空间上造成空洞, 这种不连续的空间需要其他额外信息来描述。

我们定义一个连续内存空间的描述符 `vma` (virtual memory area) 结构体, 进程的各个 `vma` 构成一个链表并由 `vm` 结构体描述。`vma` 结构体的定义如代码 3-51 所示。这里为了便于实现, 我们用数组链表来表示。

代码 3-51 vma 结构体

```
1. struct vma {
2.     uint64 address; // 内存块起始地址
3.     int length; // 内存块大小
4.     int next; // 下一块内存索引, -1 表示未分配, 0 表示没有下一个
5. };
```

然后在 PCB 中加入成员 `vm []` 数组, 如代码 3-52 所示。需要注意的是, 我们将 `vm[0]` 作为头指针使用, 它并不管理内存区。`vm[0]` 作为链表头一直存在 (记录进程创建 `vma` 之前的内存映像区域), 当 `vm[0].next=0` 表示没有创建 `vma` 内存区。当创建 `vma` 内存区域后, `vma` 结构

体成员 `next` 用于按地址从低到高排序构成链表，并且用 `vm[x].next=0` 表示 `vm[x]` 是链上最后一个区域。`vm[x].next=-1` 表示 `vm[x]` 为空闲未用。

代码 3-52 为 `proc` 添加 `vm` 成员

```

1  struct proc {
2      struct spinlock lock;
3
4      // p->lock must be held when using these:
5      enum procstate state;      // Process state
6      void *chan;                // If non-zero, sleeping on chan
7      int killed;                // If non-zero, have been killed
8      int xstate;                // Exit status to be returned to parent's wait
9      int pid;                   // Process ID
10
11     // wait_lock must be held when using this:
12     struct proc *parent;        // Parent process
13
14     // these are private to the process, so p->lock need not be held.
15     uint64 kstack;              // Virtual address of kernel stack
16     uint64 sz;                  // Size of process memory (bytes)
17     pagetable_t pagetable;      // User lower half address page table
18     struct trapframe *trapframe; // data page for uservec.S, use DMW address
19     struct context context;      // swtch() here to run process
20     struct file *ofile[NOFILE]; // Open files
21     struct inode *cwd;           // Current directory
22     char name[16];               // Process name (debugging)
23     struct vma vm[10];           // vm[0] 为指针头，剩余 9 个可用
24 };

```

新增的 `vm[]` 变量在 `kernel/proc.c` 的 `allocproc()` 中初始化，添加如下代码。

```

1  for(int i = 1; i < 10; i++) { // vma 的初始化
2      p->vm[i].next = -1;
3      p->vm[i].length=0;
4  }
5  p->vm[0].next = 0;

```

■ 查看 `vma` 信息

修改 `procdump()`，增加内存映像的输出，这样在 `Ctrl+P` 的时候就可以将每个进程的各 `vma` 起始地址和长度显示出来。（主要修改输出循环）

```

1.  for(p = proc; p < &proc[NPROC]; p++) {
2.      if(p->state == UNUSED)
3.          continue;
4.      if(p->state >= 0 && p->state < NELEM(states) && states[p->state])
5.          state = states[p->state];
6.      else
7.          state = "???";
8.      printf("%d %s %s", p->pid, state, p->name);
9.      printf("\n");
10.     for(int i = p->vm[0].next; i != 0; i = p->vm[i].next) {
11.         printf("start: %d, length: %d\n", p->vm[i].address, p->vm[i].length);
12.     }
13.     printf("\n");
14. }

```

分配与回收操作

添加 `myfree()` 的 `myalloc()` 系统调用, 在分配空间时, 需要 (1) 查找合适的地址范围, 并创建 `vma` 进行描述; (2) 要用 `kalloc()` 分配足够的物理页帧, 并用 `mappages()` 将这些页帧映射到所指定的虚存地址上。在释放空间时, (1) 需要将所涉及的页帧, 逐个解除页表映射; (2) 删除 `vma`。我们这里分配和释放内存空间, 都以页 (4KB) 的整数倍为大小, 以减少编程细节、减轻大家的编程工作量。

其中 `alloc()` 调用了 `mygrowproc()` 来完成进程空间分配, 并用相应的 `vma` 来记录空洞信息, `free()` 调用 `myreduceproc()` 来释放进程空间, 并释放对应的 `vma`。将这两个函数的实现放在 `proc.c` 中, 如代码 3-53 所示。

代码 3-53 `mygrowproc()` 和 `myreduceproc()`

```

1  uint64
2  mygrowproc(int n) {                // 实现首次最佳适应算法
3      struct proc *proc = myproc();
4      struct vma *vm = proc->vm;    // 遍历寻找合适的空间
5      uint64 start = proc->sz;      // 寻找合适的分配起点
6      int index;
7      int prev = 0;
8      int i;
9
10     for(index = vm[0].next; index != 0; index = vm[index].next) {
11         if(start + n < vm[index].address)
12             break;
13         start = vm[index].address + vm[index].length;
14         prev = index;
15     }
16
17     for(i = 1; i < 10; i++) {      // 寻找一块没有用的 vma 记录新的内存块
18         if(vm[i].next == -1) {
19             vm[i].next = index;
20             vm[i].address = start;
21             vm[i].length = n;
22
23             vm[prev].next = i;      // 将 vm[i] 挂入链表尾部
24
25             myallocvm(proc->pagetable, start, start + n); // 为 vm[i] 分配内存
26             return start;          // 返回分配的地址
27         }
28     }
29     return 0;
30 }
31 int
32 myreduceproc(uint64 address) { // 释放 address 开头的内存块
33     int prev = 0;
34     int index;
35     struct proc *proc = myproc();
36
37     for(index = proc->vm[0].next; index != 0; index = proc->vm[index].next) {
38         if(proc->vm[index].address == address && proc->vm[index].length > 0)
39             { // 找到对应内存块
40                 mydeallocvm(proc->pagetable, proc->vm[index].address,
proc->vm[index].address + proc->vm[index].length); // 释放内存
proc->vm[prev].next = proc->vm[index].next; // 从链上摘除

```

```

41         proc->vm[index].next = -1;           //标记为未用
42         proc->vm[index].length = 0;
43         break;
44     }
45     prev = index;
46 }
47 return 0;
48 }

```

其中还用到了 `myallocvm()` 完成虚拟空间到物理页帧的映射，而 `mydeallocvm()` 完成虚拟空间到物理页帧的解绑。这两个代码实现在 `vm.c` 中，如代码 3-54 所示。

代码 3-54 `myallocvm()`和 `mydeallocvm()`

```

1  // start 和 end 都是虚拟地址
2  uint64
3  myallocvm(pagetable_t pgdir, uint64 start, uint64 end) {
4      char* mem;
5      uint64 a;
6
7      a = PGROUNDUP(start);
8      for(; a < end; a += PGSIZE) {
9          mem = kalloc();
10         memset(mem, 0, PGSIZE);
11         mappages(pgdir, a, PGSIZE, (uint64)mem, PTE_P|PTE_W|PTE_PLV|PTE_MAT|PTE_D);
12     }
13     return (end-start);
14 }
15 // start 和 end 都是虚拟地址
16 uint64
17 mydeallocvm(pagetable_t pgdir, uint64 start, uint64 end) {
18     if(PGROUNDUP(start) < PGROUNDUP(end)) {
19         int npages = (PGROUNDUP(end) - PGROUNDUP(start)) / PGSIZE;
20         uvmunmap(pgdir, PGROUNDUP(start), npages, 1);
21     }
22
23     return start;
24 }

```

上述方案对分配的空间大小限制为页的整数倍，实际应用上并不应该有这样的限制。

测试代码

编写应用程序如代码 3-55 所示，分配连续 5 个空间，然后释放其中的 2/4，（1）察看内存空间是否有空洞；（2）发出指向内存空洞的区间，是否会引发进程非法操作而撤销。关于如何将 `mygrowproc()`和 `myreduceproc()`封装成系统调用的代码，请读者自行补充完整。

代码 3-55 `myalloc.c` 测试代码

```

1  #include "kernel/types.h"
2  #include "kernel/stat.h"
3  #include "user/user.h"
4
5
6  int
7  main(int argc, char *argv[]) {
8      // int pid = getpid();
9      // map(pid);
10
11     char* m1 = (char*)myalloc(2 * 4096);
12     char* m2 = (char*)myalloc(3 * 4096);

```


```

13  char* m3 = (char*)myalloc(1 * 4096);
14  char* m4 = (char*)myalloc(7 * 4096);
15  char* m5 = (char*)myalloc(9 * 4096);
16
17  m1[0] = 'a';
18  m1[1] = '\0';
19  m2[2] = 'b';
20  m3[2] = 'b';
21  m4[2] = 'b';
22  m5[2] = 'b';
23
24  printf("m1:%s\n", m1);
25  myfree((uint64)m2);
26
27
28  // 尝试往空洞写数据
29  //  m2[1] = 'p';
30
31  myfree((uint64)m4);
32
33  // map(pid);
34  sleep(5);
35  myfree((uint64)m1);
36  myfree((uint64)m3);
37  myfree((uint64)m5);
38
39
40
41  exit(0);
42  }

```

进入 xv6 系统后运行 myalloc, 先输出 m1 内存区中的字符, 然后程序会睡眠一段时间, 等待用户查看其内存区间的情况, 如屏显 3-10 所示。

屏显 3-10 myalloc 的输出

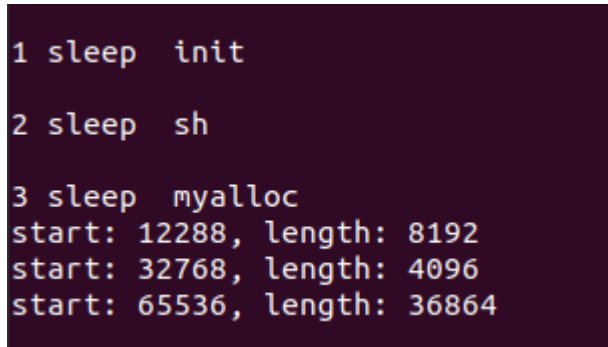


```

$ myalloc
m1:a

```

这时按 Ctrl+P 可以看到如下的具体空间。



```

1 sleep  init
2 sleep  sh
3 sleep  myalloc
start: 12288, length: 8192
start: 32768, length: 4096
start: 65536, length: 36864

```

如果将代码中注释掉的代码恢复, 试图在内存空洞里面写数据, 会出现如下结果

```
$ myalloc
m1:a
usertrap(): unexpected trapcause 20000 pid=3
          era=0x000000000000009c badi=29c06061
panic: freewalk: leaf
```

以上测试结果表明所设计功能基本正常，读者可以对上述内存区进行读写操作或其他更详尽的测试。

3.5. 小结

本章的几个实验将前面所学的进程管理、进程通信和内存管理知识进行了检验。3.1 节和 3.2 节的实验代码几乎给出了全部细节，后面的实验也给出了绝大部分的主体代码，仅简化了在系统调用的封装和应用程序的编写上的少量细节。通过学习实验操作和代码的阅读，读者对 xv6 核心机制的编码实现将会有较大收获。

书中给出的只是实例代码，读者可以自行审视其缺陷和不足，并进行扩展。实验的设计介绍中，也提到很多简化约束，读者可以突破这些限制将相应的功能拓展到更合理的状态。

练习

1. 请自行设计并实现时间片长度可调的调度，提供一个系统调用用于设置指定进程的时间片长度——`setslot(int pid, int slots)`，其中参数 `pid` 是待设置的进程号，`slots` 是该进程的时间片长度。
2. 共享内存实验中虽然设置了 `shm` 边界，但是并没有在 `sbrk()` 中检测越界问题。请尝试通过 `sbrk()` 申请内存并观测捕捉越界行为的表现，并给出改正后的代码。然后将共享内存的锁的粒度细化，各个内存区使用独立的锁而不是示例中共用一个粒度的锁。
3. 将数据代码分离实验进一步完善，利用 `vma` 描述两段属性不同的区间，并在页表映射时的填写合适的访问属性，使得代码段不可写。
4. 请实现进程间的无名管道通信，注意读写同步问题的实现。
5. 思考：（1）如果允许用户分配任意尺寸的空间，例如分配了 16 个字节和 32 字节的两个空间，那么你使用两个页帧来映射它们，还是用一个页帧来映射它们？（2）如果用一个页来映射这 48 个字节，那么释放时又如何处理，你能提供一个比较完整的解决方案使得页帧的空间利用率较高？

第4章 高级实验

在学习了 xv6 的全部源代码以及完成了初级和中级实验之后，本章组织了几个综合性实验，需要涉及多个模块的知识。本章实验的完整代码可以从 <https://github.com/SKT-CPUOS/xv6-loongarch-exp> 下载。

4.1. 实现 xv6 内核线程

创建线程的开销比创建进程要小，线程可以共享进程的主要资源——例如内存映像、打开的文件等等。在 xv6 上实现线程所涉及的主要工作包括如下：

- (1) 实现 `clone()` 系统调用，用于创建一个内核线程。
- (2) 实现 `join()` 系统调用，用于回收一个内核线程。
- (3) 实现用户线程库，封装对线程的管理，而用户只需要知道接口的使用即可。
- (4) 提供测试样例，包括共享进程空间、多线程并行。

由于这里的实验代码需要 `alloc()` 和 `free()` 的内存管理机制，因此需要在前面中级实验的基础之上进行。

4.1.1. 修改 PCB

为了支持内核线程，我们将进程控制块 PCB 改造成线程控制块 TCB——改造为线程和进程共用的控制块，在 `proc` 结构体中添加两个变量如下：

```
1. struct proc *pthread;           // Parent thread
2. void *ustack;                   // User thread stack
```

进程创建过程也需要做相应的修改。在 `proc.c` 的 `allocproc()` 中将 `pthread` 初始化为 0。

新增的两个成员变量用于分别记录父线程和本线程自己的用户栈，作为线程私有资源。在内核态中，线程的初始用户态栈被设置了线程执行的起点（伪造返回的 PC 指针）、传递的参数值。

在这里的简化线程方案中，我们确定几个称谓：进程在未创建新的线程时，就只有一个执行流，称为主线程或父线程；新创建的执行流称为子线程。主线程随着进程创建而存在，子线程创建后共享主线程的进程映像，并且在线程库登记线程的资源（线程栈）。

4.1.2. `sys_clone` 和 `sys_join` 系统调用

`sys_clone()`

线程的创建由 `clone()` 系统调用实现。`clone()` 的实现和 `fork()` 的实现及其相似，不过子线程共享了父线程/主线程的进程映像和大多数其他资源。此外 `clone()` 还要负责初始化用户栈，使得线程回到用户态后能找到对应的入口。用户栈的设置可以参考 `exec()` 函数。`clone()` 的实现放在了 `kernel/proc.c`，如代码 4-1 所示。

由于新创建线程的内核栈 `trapframe` 里的 `era` 被设置成传入的参数 `fcn`（即线程函数），因此当 `clone()` 返回到用户态后，将执行 `fcn` 所指定的函数。用户态的线程栈则填写了对应的线程函数的参数以及返回地址。这里的简化实现中，要求线程结束处必须显式调用 `exit()`。当然也可以将返回地址填上 `exit()` 的地址，从而不必显式调用。

这里的简化实现方案中，并没有对页表的引用进行计数，如果主线程提前结束释放内存空间后会造成子线程的异常。

代码 4-1 clone()

```

1. //调用 clone()前需要分配好线程栈的内存空间，并通过 stack 参数传入
2. int clone(void (*fcn)(void *), void *stack, void *arg) {
3.
4.     struct proc *curproc = myproc(); //记录发出 clone 的进程
5.     struct proc *np;
6.     if ((np = allocproc()) == 0) //为新线程分配 PCB/TCB
7.         return -1;
8.
9.     np->pagetable = curproc->pagetable; //线程间共用同一个页表
10.
11.     if((np->trapframe = (struct trapframe *)kalloc()) == 0){ //分配线程内核栈空间
12.         freeproc(np);
13.         release(&np->lock);
14.         return 0;
15.     }
16.
17.
18.     np->sz = curproc->sz;
19.     np->pthread = curproc; // exit 时用于找到父线程并唤醒
20.     np->ustack = stack; // 设置自己的线程栈
21.     np->parent = 0;
22.     *(np->trapframe) = *(curproc->trapframe); //继承 trapframe
23.
24.     // 设置 trapframe 映射
25.     if(mappages(np->pagetable, TRAPFRAME - PGSIZE, PGSIZE,
26.         (uint64)(np->trapframe), PTE_NX | PTE_P | PTE_W | PTE_MAT | PTE_D) < 0){
27.         uvmfree(np->pagetable, 0);
28.         return 0;
29.     }
30.
31.     // 设置栈指针
32.     np->trapframe->sp = (e)(stack + 4096 - 8);
33.     // 修改返回值 a0
34.     np->trapframe->a0 = (uint64)arg;
35.
36.     // 修改返回地址
37.     np->trapframe->era = (uint64)fcn;
38.
39.     // 复制文件描述符
40.     for (int i = 0; i < NOFILE; i++)
41.         if (curproc->ofile[i])
42.             np->ofile[i] = filedup(curproc->ofile[i]);
43.     np->cwd = idup(curproc->cwd);
44.
45.     safestrcpy(np->name, curproc->name, sizeof(curproc->name));
46.     int pid = np->pid;
47.
48.     release(&np->lock);
49.

```

```

50.     acquire(&np->lock);
51.     np->state = RUNNABLE;
52.     release(&np->lock);
53.
54.     // 返回新线程的 pid
55.     return pid;
56. }

```

将上述核心函数 `clone()` 封装成系统调用 `sys_clone()`，如代码 4-2 所示。

代码 4-2 `sys_clone()`

```

1.  uint64 sys_clone(void){
2.      uint64 a;
3.      uint64 b;
4.      uint64 c;
5.      argaddr(0,&a);
6.      argaddr(1,&b);
7.      argaddr(2, &c);
8.
9.      return (uint64)clone((void (*)(void *))a,(void*)b, (void *)c);
10. }

```

sys_join

在这里的简化方案中，子进程和子线程的结束时的资源回收过程有所不同：

- （1）子进程的 PCB 由父进程调用 `wait()` 回收；
- （2）子线程的 PCB/TCB 由父进程/线程调用 `join()` 回收。

由于两者差异很小，因此 `join()` 的实现和 `wait()` 极其相似。`join()` 可看作 `clone()` 的逆过程，由当前线程回收状态为 ZOMBIE 的子线程。子线程需要调用 `exit()` 才会变成 ZOMBIE 线程。

`join()` 的实现如代码 4-3 所示。由于线程退出并不需要撤销进程映像，因此不需要像 `wait()` 里面那样执行 `freeproc()`。

代码 4-3 `join()`

```

1.  int join() {
2.      struct proc *curproc = myproc();
3.      struct proc *p;
4.      int havekids;
5.      for (;;) {
6.          havekids = 0;
7.          for (p = proc; p < &proc[NPROC]; p++) {
8.              if (p->pthread != curproc)    // 判断是不是自己的子线程
9.                  continue;
10.
11.             havekids = 1;
12.             if (p->state == ZOMBIE) {
13.                 acquire(&p->lock);
14.                 if(p->trapframe)
15.                     kfree((void*)p->trapframe);
16.                 p->trapframe = 0;
17.                 if(p->pagetable)
18.                     uvmunmap(p->pagetable, TRAPFRAME - PGSIZE, 1, 0); // 释放内核栈
19.                 p->pagetable = 0;
20.                 p->sz = 0;
21.                 p->pid = 0;
22.                 p->parent = 0;
23.                 p->pthread = 0;

```



```

24.         p->name[0] = 0;
25.         p->chan = 0;
26.         p->killed = 0;
27.         p->xstate = 0;
28.         p->state = UNUSED;
29.
30.         int pid = p->pid;
31.
32.         release(&p->lock);
33.         return pid;
34.     }
35. }
36. if (!havekids || curproc->killed) {
37.     return -1;
38. }
39. sleep(curproc, &wait_lock);
40. }
41. return 0;
42. }

```

将上述核心函数 `clone()` 封装成系统调用 `sys_join()`，如代码 4-4 所示。

代码 4-4 `sys_join()`

```

1.  uint64 sys_join(void){
2.      return (uint64)join();
3.  }

```

可以看出，如果主线程调用 `join()` 的时候，子线程还未执行 `exit()`，那就找不到状态为 ZOMBIE 子线程，主线程将会进入睡眠状态。因此子线程退出的时候需要唤醒对应的主线程。

我们需要在 `proc.c` 的 `exit()` 中增加唤醒主线程的功能，具体代码如下。

```

1.  // Parent might be sleeping in join().
2.  if(p->parent==0 && p->pthread!=0)
3.      wakeup(p->pthread);
4.  else wakeup(p->parent);

```

用户态函数

`sys_clone()` 和 `sys_join()` 两个系统调用还需要封装成用户态可调用的函数，例如可以是同名的 `clone()` 和 `join()`，经过编译链接形成库供应用程序调用。系统调用如何封装成用户态可调用函数的内容在初级实验和系统调用章节已经讨论过，此处不再赘述。

4.1.3. 线程库的实现

`clone()` 和 `join()` 只实现了内核线程，为了方便用户程序调用。我们还需要实现相应的用户线程库，帮忙管理用户栈和线程的协调。用户库为 `uthread.c`，需要将其添加到 `Makefile` 的 `ULIB` 变量中。其中 `thread_create()` 和 `thread_join()` 是暴露给用户程序使用的。

`thread_create()` 用于创建线程，需要提供待执行的线程函数和运行参数。`thread_create()` 通过 `clone()` 创建线程，需要提前用 `malloc()` 分配一个线程栈，最后借助于 `add_thread()` 将线程记录在本进程的 TCB 数组 `threads[NTHREAD]` 中。对应地有一个 `thread_join()` 用于等待线程结束，它通过 `join()` 系统调用回收已经停止的线程，然后通过 `remove_thread()` 从本进程的线程数组 `threads[NTHREAD]` 中删除。

代码 4-5 `uthread.c`

```

1.  #include "kernel/types.h"
2.  #include "user/user.h"

```

```

3.
4.     #define NTHREAD 4      // 一个进程的最大线程数 (不包括主线程)
5.     #define PGSIZE 4096
6.     struct {
7.         int pid;
8.         void* ustack;
9.         int used;
10.    } threads[NTHREAD];      // TCB 表
11.
12.    // add a TCB to thread table
13.    void add_thread(int* pid, void* ustack) {
14.        for(int i = 0; i < NTHREAD; i++) {
15.            if(threads[i].used == 0) {
16.                threads[i].pid = *pid;
17.                threads[i].ustack = ustack;
18.                threads[i].used = 1;
19.                break;
20.            }
21.        }
22.    }
23.    void remove_thread(int* pid) {
24.        for(int i = 0; i < NTHREAD; i++) {
25.            if(threads[i].used && threads[i].pid == *pid) {
26.                free(threads[i].ustack);      // 释放用户栈
27.                threads[i].pid = 0;
28.                threads[i].ustack = 0;
29.                threads[i].used = 0;
30.                break;
31.            }
32.        }
33.    }
34.
35.    int thread_create(void (*start_routine)(void*), void* arg) {
36.        // If first time running any threads, initialize thread table with zeros
37.        static int first = 1;
38.        if(first) {
39.            first = 0;
40.            for(int i = 0; i < NTHREAD; i++) {
41.                threads[i].pid = 0;
42.                threads[i].ustack = 0;
43.                threads[i].used = 0;
44.            }
45.        }
46.        void* stack = malloc(PGSIZE);      // allocate one page for user stack
47.        int pid = clone(start_routine, stack, (void*)arg); // system call for kernel thread
48.        add_thread(&pid, stack); // save new thread to thread table
49.        return pid;
50.    }
51.
52.
53.    int thread_join(void) {
54.        for(int i = 0; i < NTHREAD; i++) {
55.            if(threads[i].used == 1) {
56.                int pid = join();      // 回收子线程
57.
58.                if(pid > 0) {
59.                    remove_thread(&pid);
60.                    return pid;
61.                }
62.            }
63.        }

```

```

64.     return 0;
65. }
66.
67. void printTCB(void) {
68.     for(int i = 0; i < NTHREAD; i++)
69.         printf("TCB %d: %d\n", i, threads[i].used);
70. }

```

由于创建线程的时候并没有检查子线程数量是否超过 NTHREAD 上限，因此需要用户代码自行遵循这个约定。读者也可以为 `thread_create()` 增加这个检查。

需要注意的是在从内核态回到用户态时，需要判断 `proc->pthread` 保存对应的 `trapframe`。具体需要修改 `kernel/trap.c` 的 `usertrapret()` 函数。

```

1.     // jump to uservec.S at the top of memory, which
2.     // switches to the user page table, restores user registers,
3.     // and switches to user mode with ertn.
4.     // userret(TRAPFRAME, pgdl);
5.     if(p->pthread == 0){
6.         userret(TRAPFRAME, pgdl);
7.     }else{
8.         userret(TRAPFRAME - PGSIZE, pgdl);
9.     }

```

4.1.4. 测试样例

测试样例为 `test.c`，内容如代码 4-6，它将调用用户态库中的 `clone()` 和 `join()`。这个测试程序用于进行如下测试：测试 `join()` 和 `clone()` 的参数是否传递成功；线程间是否共享进程空间；子线程是否继承了主线程的文件描述符（但不共享）；测试用户栈是否设置正确（可以将代码 4-1 第 4 行的注释去掉从而可以打印线程栈的起始地址）；测试 `join()` 得到的 `pid` 是否等于 `clone()` 得到的 `pid`（同个线程）；调用 `wait()` 的时候处理 `fork()`，调用 `join()` 的时候处理 `clone()`；保证线程库不会产生内存泄漏问题（用户栈的分配与回收）；测试线程的递归（斐波那契数列）：若线程不调用 `exit()`，应该发生非法地址中断；测试子线程和主线程的进程空间大小是否一致。

代码 4-6 线程测试代码 `test.c`

```

1  #include "kernel/fcntl.h"
2  #include "kernel/types.h"
3  #include "user/uthread.h"
4  #include "user/user.h"
5  volatile int global = 1;
6  int F(int n) {
7      if (n < 0)
8          printf("input a positive integer\n");
9      else if (n == 1 || n == 2)
10         return 1;
11     else {
12         return F(n - 1) + F(n - 2);
13     }
14     return 0;
15 }
16 void worker(void* arg) {
17     global = 100;
18     printf("thread %d is worker.\n", *(int *)arg);
19 }

```

```

20     global = F(15);
21     write(3, "hello\n", 6);
22     exit(0);
23 }
24
25 int main() {
26     int t = 1;
27     open("tmp", O_RDWR | O_CREATE);
28     int pid = thread_create(worker, &t);
29     sleep(10);
30     thread_join();
31     printf("thread id = %d\n", pid);
32     printf("global = %d\n", global);
33
34     exit(0);
35 }

```

运行如屏显 4-1，可以看出成功创建了线程并将参数传入，线程通过递归计算得到的斐波那契数列也正确，`global` 变量的使用结果也验证了线程间共享内存的事实，检查 `tmp` 文件也可以看到相应的字符串。读者可以尝试其他更全面的测试，例如使主线程不执行 `thread_join()` 而提前结束并观察运行结果。

屏显 4-1 线程测试输出

```

$ test
thread 1 is worker.
thread id = 4      # 线程号
global = 610      # 线程共享变量 global 记录斐波那契数列递归值

```

4.2. 文件系统实验

文件系统方面我们安排了三个实验，一个是为 `xv6` 文件系统增加文件读写权限控制，第二个实现恢复刚被删除的文件内容，第三个是和设备有关的磁盘裸设备读写。

4.2.1. 文件权限

本实验将在原来的 `xv6` 文件系统上，增加文件读写权限控制，具体步骤如下：

- (1) 在 `inode` 上添加读写权限标志。
- (2) 提供修改读写权限的系统调用。
- (3) 修改相应的读写系统调用，添加读写权限检查。
- (4) 提供测试样例。

添加访问权限位

`xv6` 的 `inode` 结构体定义在 `kernel/file.h`，其中一部分信息是要存储在硬盘上的，这部分内容用 `dinode`（定义在 `kernel/fs.h`）描述。由于磁盘布局是很严格的，所以结构体的大小都是设计好的。`xv6` 中用来描述文件类型的变量是 `short` 类型，为了增加文件权限功能，且不影响文件系统布局，我们将 `short` 类型拆成两个 `char` 类型，一个用来当作 `mode` 来描述文件权限，此时新的 `dinode` 定义如代码 4-7。同时 `kernel/file.h` 中的 `inode` 结构体也要做同样的修改。

代码 4-7 增加访问权限的 `dinode`

```

1.  // On-disk inode structure
2.  struct dinode {
3.      char  mode;           // 文件权限
4.      char  type;          // File type
5.      short major;         // Major device number (T_DEV only)
6.      short minor;         // Minor device number (T_DEV only)
7.      short nlink;         // Number of links to inode in file system
8.      uint size;           // Size of file (bytes)
9.      uint addrs[NDIRECT+1]; // Data block addresses
10. };

```

这样就完美地将文件的权限管理添加到了文件系统，由于 `type` 的类型变化，我们要适配所有涉及 `type` 类型的函数或变量（使用 `grep -rn "short type" *.c *.h` 可以快速定位）。

（1）在 `mkfs/mkfs.c` 中，对函数 `ialloc()` 进行修改，将参数中的 `type` 全部改为 `uchar` 类型，并在函数中初始化 `mode` 为 3。

（2）修改 `kernel/fs.c` 中的 `ialloc()` 中参数 `type` 改为 `char` 类型，并在函数中初始化 `mode` 为 3。同时修改在 `defs.h` 中的声明。

（3）修改 `kernel/sysfile.c` 中的 `create()` 函数，将其参数 `type` 类型改为 `char`。

（4）在 `kernel/fs.c` 中的 `ilock()` 中，将 `dinode->mode` 传递给 `inode->mode`

（5）在 `kernel/fs.c` 中的 `iupdate()` 中，将 `inode->mode` 传递给 `dinode->mode`

（6）在 `kernel/stat.h` 中，修改 `stat` 结构体，添加 `char` 类型的 `mode`，并把 `type` 修改为 `char` 类型。

（7）在 `kernel/fs.c` 中的 `stat()` 中，将 `inode->mode` 传递给 `stat->mode`

（8）在 `user/ls.c` 中，修改其中的几处 `printf` 输出语句，使得能打印文件的访问权限 `mode`。

做完以上修改后，运行 `xv6`，执行 `ls` 命令，结果如下：

```

$ ls
.          3 1 1 1024
..         3 1 1 1024
README    3 2 2 2226
cat        3 2 3 20272
echo       3 2 4 19272
forktest  3 2 5 11464
grep       3 2 6 22792
init       3 2 7 19984
kill       3 2 8 19160
ln         3 2 9 19056

```

从左到右分别是：文件名、访问权限、文件类型、索引节点、文件大小。

设置权限的系统调用

`xv6` 的文件有 3 种类型，分别是目录、文件、设备。我们只限制普通文件的读写权限，即 `T_FILE` 类型的文件读写才受到控制。而且将 `mode` 的最低位作为读位，次低位作为写位，则有

- ✧ 3 表示可读可写。
- ✧ 2 表示可写。
- ✧ 1 表示可读。
- ✧ 0 表示不可读不可写。

为了支持文件权限，且不改变 xv6 的接口，我们需要实现专门的系统调用来改变文件的读写权限。用户接口定义如下：

```
1. int chmod(const char* pathname, char mode);
```

参考 link() 系统调用过程，我们新增 sys_chmod() 来修改 inode->mode，该函数可以放到 kernel/sysfile.c 中实现，如代码 4-8 所示。

代码 4-8 sys_chmod()

```
1. uint64
2. sys_chmod(void) {
3.     char *pathname = "";
4.     int mode;
5.     struct inode *ip;
6.     if(argstr(0, pathname, 20) < 0) {
7.         return -1;
8.     }
9.     if(argint(1, &mode) < 0) {
10.        return -1;
11.    }
12.
13.    begin_op();
14.
15.    if((ip = namei((char *)pathname)) == 0) {
16.        end_op();
17.        return -1;
18.    }
19.
20.    ilock(ip);
21.
22.    ip->mode = mode;
23.
24.    iupdate(ip);
25.    iunlock(ip);
26.
27.    end_op();
28.
29.    return 0;
30. }
```

读写前判断

接下来可以使用新增的权限来限制读写操作了。文件的读写函数分别是 fileread() 和 filewrite()，它们位于 kernel/file.c 中。

其中 fileread() 中的修改部分如代码 4-9，执行读操作之前进行 mode 权限检查。

代码 4-9 具有权限检查的 fileread()

```
1. if(f->type == FD_INODE){           // 文件类型
2.     ilock(f->ip);                   // 索引节点为临界资源
3.     if((f->ip->mode & 1) == 0) {     // 不可读
4.         iunlock(f->ip);
5.         return -1;                 //表示读操作失败
6.     }
7.
8.     ...
9. }
```

filewrite() 中的修改部分如代码 4-10，多了写之前的访问权限发判断。

代码 4-10 具有权限检查的 filewrite()

```
1. if(f->type == FD_INODE){
2.     begin_op();
```

```

3.      ilock(f->ip);
4.      if((f->ip->mode & 2) == 0) {           // 判断是否可写
5.          iunlock(f->ip);
6.          end_op();
7.          return -1;                         //表示写操作失败
8.      }
9.      iunlock(f->ip);
10.     end_op();
11.
12.     ...
13. }

```

这样，就完成了读写的权限判断。

测试样例

首先，我们得实现一个用户程序 `chmod`，可以在 `shell` 中直接修改某个文件的访问权限，具体如代码 4-11。

代码 4-11 `chmod` 程序

```

1.  #include "kernel/types.h"
2.  #include "kernel/stat.h"
3.  #include "user/user.h"
4.
5.  int
6.  main(int argc, char *argv[]) {
7.      if(argc <= 2) {
8.          printf("format: chmod pathname mode\n");
9.      }
10.
11.
12.      chmod(argv[1], atoi(argv[2]));
13.
14.      exit(0);
15.  }

```

接下来开始读写操作权限控制的测试。首先在默认读写权限（可读+可写）情况下，用 `echo hello > content` 将数据写入到 `content` 文件中，此时用 `ls` 查看 `content` 文件为访问权限为 3（表示可读+可写）。

```

$ echo hello > content # 利用重定向生成 content 文件
$ ls                  # 显示 content 文件属性
content              3 2 18 6 # 3 表示可读可写
$ cat content         # 读 content 的内容
hello                # 成功读取
$ echo world > content # 修改 content 内容
$ cat content         # 查看内容，发现修改成功
world

```

然后重复上面的操作，但是第一次写入 `hello` 后用 `chmod` 命令将 `content` 修改为可读不可写的权限，然后执行第二次写入 `world`。由于第二次写操作前关闭了写入权限，因此后面重新读入文件后，发现第二次写操作并没有成功，数据仍是原来的旧数据 `hello`。

```

$ echo hello > content # 创建 content
$ cat content         # 查看 content 内容
hello
$ ls                  # 查看 content 文件属性
content              3 2 18 6
$ chmod content 1     # 修改 content 的访问权限

```

```
$ ls                # 查看修改后的文件属性
content            1 2 18 6
$ echo world > content # 修改 content 内容
$ cat content        # 查看内容, 发现修改失败
hello
```

最后, 我们先将 `hello` 写入到 `content`, 然后用 `chmod` 命令将 `content` 设置为可写不可读的权限, 再用 `cat` 命令读取 `content` 文件内容时提示出错失败。

```
$ echo hello > content # 生成 content 文件
$ cat content          # 读取文件
hello
$ ls                  # 查看文件属性
content              3 2 21 6
$ chmod content 2      # 修改访问权限
$ ls                  # 查看修改后的文件属性
content              2 2 21 6
$ cat content          # 读取内容失败
cat: read error
```

如果想设置为不可读不可写模式, 只需执行 `chmod pathname 0` 即可。

4.2.2. 恢复被删除的文件

我们尝试删除一个文件后, 根据原有文件留存的数据盘块, 将其恢复。由于 `xv6` 在删除文件时, `inode` 内容被虽然清除掉了, 但是数据盘块没有清除, 因此重建 `inode` 信息就可以恢复文件。我们做一个实验演示, 提前将索引信息保存下来, 然后删除文件, 接着再恢复文件内容。本实验展示如下的删除和恢复过程:

(1) 在用 `rm` 命令删除一个文件前, 用一个新的系统调用来记录其 `inode` 索引表到一个临时文件中。

(2) 调用 `rm` 删除文件后, 利用临时文件中保存的索引表找回文件内容。

记录索引表

由于 `inode` 记录着盘块的组织, 如果 `inode` 内容被清除了, 那么即使盘块数据还在, 也无法知道具体些盘块上, 也就访问不了文件。

早期的 `EXT2` 文件系统, 删除文件操作只是将磁盘索引节点标志设置为无效, 而没有清空其 `inode` 内容, 因此通过扫描索引节点表, 将那些无效的但其索引表仍有内容的索引节点重新生效, 就能找回被删除的文件——前提是它们的数据盘块还没有被改写。但是 `xv6` 的文件系统以及 `Linux` 的 `EXT3/4` 等文件系统在删除文件的时候都会将 `inode` 内容清空, 因此也就无法凭空恢复这些信息——所以本实验中先要将文件 `inode` 信息保存下来。

为了简化实验, 我们先记录我们将要删除的文件索引表内容, 然后直接按照索引表重建一个文件。所以我们得实现相应的代码来记录所指定文件的 `inode`, 以便删除后还能根据 `inode` 的索引表找到所有的盘块并恢复文件。

我们还需要添加一个 `shell` 命令程序，实现将文件（例如 `README.md`）的 `inode` 的索引表保持到 `temp` 文件中。例如，当需要保存 `README.md` 文件的 `inode` 索引表时，使用的命令格式如下：

```
savei README
```

程序源代码为 `savei.c`，其内容如代码 4-12。由于我们在前面已经讨论过 `xv6` 文件系统的知识，代码工作原理就不再细述。

代码 4-12 `savei.c`

```
1.  #include "kernel/types.h"
2.  #include "kernel/stat.h"
3.  #include "user/user.h"
4.  #include "kernel/fcntl.h"
5.
6.  int main(int argc, char *argv[]) {
7.
8.      if(argc < 2) {
9.          printf("format: savei filename temp\n");
10.         exit(0);
11.     }
12.     uint addrs[13];                // 存储索引信息的数组
13.     geti(argv[1], addrs);          // 新增系统调用，用于获取索引信息，详见下文
14.     int fd = open("temp", O_CREATE | O_RDWR); // 讲索引信息写到 temp 文件上
15.
16.     for(int i = 0; i < 13; i++) {
17.         printf("a:%x\n", addrs[i]);
18.     }
19.     write(fd, addrs, sizeof(addrs));
20.     close(fd);
21.     exit(0);
22. }
```

其中 `geti()` 用于读取索引表，属于内核功能需要以一个新增的系统调用方式实现。作用是拷贝索引表到一个临时文件 `temp` 中，其核心代码如代码 4-13。该系统调用还需要在用户态库中以 `geti()` 函数的形式出现，从而可以被 `savei` 应用程序所调用。索引表记录在 `addrs[]` 数组中，其中 `addrs[13]` 用来记录文件的大小。

代码 4-13 `sys_geti()`

```
1.  uint64
2.  sys_geti(void) {
3.      char *filename = "";        // 文件名
4.      uint64 addrs = 0;           // 将索引表拷贝到内存地址 addr 处
5.      struct inode *ip;
6.      if(argstr(0, filename, 100) < 0)
7.          return -1;
8.      if(argaddr(1, &addrs) < 0)
9.          return -1;
10.     begin_op();
11.     if((ip = namei(filename)) == 0) // 得不到对应索引节点
12.     {
13.         end_op();
14.         return -1;
15.     }
16.
17.     ilock(ip);                    // 同步 inode 和 dinode
18.     int i;
19.     printf("dev:%d\n", ip->dev);
20.     for(i = 0; i < 13; i++){
```

```

21.     printf("%x\n", ip->addrs[i]);
22.
23.     copyout(myproc()->pagetable, addrs + i * 4, (char *)&ip->addrs[i], sizeof(uint)); // 拷贝索引
24. }
25. iunlock(ip);
26. end_op();
27. return 0;
28. }

```

删除文件

记录完索引节点后，使用 xv6 自带的 rm 命令将文件删除：

```
rm README
```

文件删除后，以保证该文件的数据盘块内容不被修改。删除后数据盘块内容还在，但需要避免进行任何文件写操作，否则将这些数据盘块重新分配给其他文件而使得盘块内容被改写。所以如果想恢复被删文件，必须在删除文件后尽快快地恢复文件。

恢复数据

由于我们直接保存了文件的索引表，因此可以直接进行恢复——按文件的逻辑顺序读出盘块即可。恢复程序为 `recoveri.c`，内容如代码 4-14。

代码 4-14 `recoveri.c`

```

1.  #include "kernel/types.h"
2.  #include "kernel/stat.h"
3.  #include "user/user.h"
4.  #include "kernel/fcntl.h"
5.
6.  char buf[1024 * 256];           // 盘块缓冲区
7.
8.  int main(int argc, char *argv[])
9.  {
10.     if(argc < 2)
11.     {
12.         printf("format: savei filename temp\n"); exit(0);
13.     }
14.     uint addrs[13];
15.     int fd;
16.     fd = open("temp", O_RDONLY);    // 从 temp 文件读取索引表
17.     read(fd, addrs, sizeof(addrs));
18.     close(fd);
19.     fd = open(argv[1], O_CREATE | O_RDWR);    // 根据索引信息读取对应盘块
20.     int i;
21.
22.     for(i = 0; i < 12 && addrs[i] != 0; i++) {
23.         recoverb(addrs[i], buf, 0);    // 读取数据到缓存块 buf 中
24.         write(fd, buf, 1024);         // 写到新文件
25.     }
26.
27.     i = 0;
28.
29.     if(addrs[12] != 0) {
30.
31.         int ret = recoverb(addrs[12], buf, 1);
32.
33.         write(fd, buf, ret);
34.     }
35.

```

```

36.     close(fd);
37.     exit(0);
38.
39. }

```

其中 `recoverb()` 是新增的系统调用，作用是获取盘块数据。核心代码如下：

```

1.  uint64
2.  sys_recoverb(void)
3.  {
4.      uint blockno;
5.      // char* buf = 0;
6.      uint64 addr = 0;
7.      int indirect = 0;
8.      if(argint(0, (int *)&blockno) < 0 || argaddr(1, &addr) < 0 || argint(2, (int *)&indirect) < 0) {
9.          return -1;
10.     }
11.     struct buf *b = 0;
12.     if(indirect == 0) {          // 直接索引
13.         b = bread(1, blockno);
14.
15.         copyout(myproc()->pagetable, addr, (char *)b->data, 1024);
16.
17.
18.
19.         return 0;
20.     }
21.
22.     struct buf *a = 0;
23.     int i = 0;
24.     a = bread(1, blockno);
25.     uint* addrList = (uint *)a->data;
26.     while (addrList[i] != 0)      // 间接索引读取
27.     {
28.         b = bread(1, addrList[i]);
29.         copyout(myproc()->pagetable, addr, (char *)b->data, 1024);
30.         addr += 1024;
31.         i++;
32.     }
33.
34.     return i * 1024;
35.
36.
37.
38. }

```

样例演示

实现完上面的功能后，我们可以测试一下恢复功能了。测试过程如下，全操作都是在 `shell` 中完成。

```

$ ls                      # 查看待删除文件的属性
README      2 2 2226
$ cat README              # 查看待删除文件的内容
...

You will need a RISC-V "newlib" tool chain from
https://github.com/riscv/riscv-gnu-toolchain, and qemu compiled for
riscv64-softmmu. Once they are installed, and in your shell
search path, you can run "make qemu".
...

$ savei README            # 存储待删除文件的索引信息
$ ls                      # 查看索引信息文件属性

```

```
temp          2 19 52
$ rm README      # 删除 README.md
$ recoveri newfile # 利用 temp 中的索引信息读取盘块恢复文件, 保证在 newfile
$ ls
newfile        2 20 3072 # 可以看到新文件占用了索引节点号 2
$ cat newfile    # 查看内容, 恢复成功
...
You will need a RISC-V "newlib" tool chain from
https://github.com/riscv/riscv-gnu-toolchain, and qemu compiled for
riscv64-softmmu. Once they are installed, and in your shell
search path, you can run "make qemu".
...
```

读者可以尝试修改 xv6 的 `itrunc()`, 使得删除文件用特殊的 `type` 标识, 但保留索引节点的大部分信息。这样, 在删除文件后可以通过扫描索引节点表, 将被删除的索引节点列出来给用户, 用户根据需要进行其中一个进行恢复。由于上述情况下被删除的索引节点中还存在有文件名, 方便作为用户挑选的依据。

4.3. 虚拟内存

我们仅对进程空间中堆栈后面（高地址）的 `sbrk()` 分配的空间, 允许其换出, 堆栈前面的内容不换出（`exec()` 中生成的进程空间）。虚拟内存需要和请求式分页结合, 所以在使用申请的物理页帧之前, 系统不会进行分配操作, 直到产生缺页中断, 调用相应的中断程序进行分配。

为了复用 xv6 的代码, 我们依旧使用 xv6 的文件系统作为交换磁盘。修改 `kernel/memlayout.h` 中的参数 `RAMSTOP` 为 `RAMBASE + 300*4*1024`。实验中, 观察到上述配置下系统的物理页数大概在 300 个左右。

本实验总共实现了 7 个核心函数。3 个用于 `kernel/vm.c` 中实现缺页异常和换入换出, 2 个用于 `kernel/fs.c` 中实现盘块的分配, 分别是 `ballo4()` 和 `bfree4()`, 分配 4 块连续盘块。最后实现物理页帧和盘块的数据交换, 在 `kernel/bio.c` 中实现 `read_page_from_disk()` 和 `write_page_to_disk()`

1. `pgfault()` // 缺页异常函数, 会调用 `swapout()` 和 `swapin()`
2. `void swapout()` // 换出一个物理页帧
3. `void swapin(char* mem, pte_t* pte)` // 根据缺页 `pte` 读入盘块到 `mem` 指向的页帧
4. `uint ballo4(int dev)` // 连续 4 个盘块用于存放一个页帧
5. `void bfree4(int dev, uint blockno)` // 释放 4 个连续的盘块
6. `void write_page_to_disk(int dev, char* va, uint blockno)` // 从 `va` 开始的内存数据写到 `blockno` 连续 4 个磁盘块
7. `void read_page_from_disk(int dev, char* va, uint blockno)` // 读 `blockno` 开始的连续 4 个磁盘块

为了记录剩余页帧数量, 需要修改 `kernel/kalloc.c` 中的 `kmem` 结构体, 加上一个 `count` 计数值（`=freelist` 长度）, 每次 `kalloc()` 和 `kfree()` 后以做相应修改。

4.3.1. 虚存交换机制

本实验实现的虚存交换机制比较简陋, 换出的页帧内容保存到磁盘文件系统的普通文件数据区, 而不是像 Linux 那样使用独立的交换分区或交换文件。换出的页帧所在的盘块号直接保存在其 `pte` 的高位, 其 `pte` 低 12 位仍用做标志用途。

交换机制的作用范围仅限于本进程的 `sbrk()` 之上的物理页帧，而且一个进程并不会去抢占其他进程的物理页帧。我们为此在 `proc.h` 的 `proc` 结构体中添加一个变量 `uint64 swap_start`，用来记录 `sbrk()` 的起始地址。`swap_start` 以下的空间是进程创建时建立的空间，它们在本实验中不涉及交换的内存区间。修改 `xv6` 代码，在 `exec()` 和 `fork()` 中对 `swap_start` 进行初始化。

本实验只是演示一个进程启动后，分配和使用的内存总量超过系统剩余的内存总量时，呈现的本进程内部的交换过程。本实验并没有实现一个完整全面的虚存交换功能。

换出方案

换出某个进程的一个页帧的方法是：

(1) 在交换区起始地址 `proc->swap_start` 到 `proc->sz` 区间，找到有映射的页帧（walk），解除页表映射；为了知道某个虚页是否有映射，可以用 `walk()` 查找对应的 `pte` 项，然后用 `(*pte) & PTE_P != 0` 检验是否有映射。如果一个进程没有可换出的页帧，查找下一个进程。

(2) 将页帧内容写入到文件系统中。由于一块物理页帧的大小相当于 4 块盘块，我们需要分配连续的 4 块盘块，调用 4 次 `bwrite()` 写入即可。

(3) 记录换出页帧的去向，以便将来可以换入。将换出的盘块号记录在页表项 `PTE` 中。此时 `PTE` 的 `SWAPPED` 位 = 0 (`(*pte) ^ PTE_SWAPPED`)，其他高 20 位记录该页帧在对应磁盘上的盘块号。`SWAPPED` 是我们定义为 `0x200` 的一个值，供换入换出标志。从未使用过的物理页帧的 `PTE` 为 0；调出磁盘的状态 `PTE_SWAPPED` 位为 1，高 20 位记录 `blockno`；有映射的 `PTE_P` 为 1，所以这三种区别正好对应三种情况。

有映射的页表项 `PTE` 内容

Page physical address	Page flags (PTE_P=1)
-----------------------	-------------------------------

换出的页表项 `PTE` 内容

Block number in swapdisk	Page flags (PTE_SWAPPED=1)
--------------------------	-------------------------------------

由于我们简化了交换机制，没有设置独立的交换区或者交换文件，而是直接在文件系统的数据区内的盘块用于换出。也就是说，换出的内容将离散地分布在文件系统的数据盘块区，每个换出的页以连续 4 个盘块的形式出现。

编码实现

接下来讨论交换功能的具体实现。`bio.c` 文件主要负责磁盘的读写，在这里我们实现 `write_page_to_disk()` 和 `read_page_from_disk()` 操作用于承担换进换出时的磁盘读写操作，如代码 4-15 所示。在 `write_page_to_disk()` 中用到的 `DMWIN_MASK` 宏定义在 `kernel/memlayout.h` 中，所以还要在 `bio.c` 中包含 `memlayout.h`。

代码 4-15 `write_page_to_disk()` 和 `read_page_from_disk()`

```

1. // blockno 起始的连续 4 块盘块数据拷贝到 va 起始的物理页帧中
2. void read_page_from_disk(int dev, char *pa, uint blockno)
3. {
4.     // printf("从磁盘换入\n");

```

```

5.     struct buf* b; // xv6 的读写必须经过缓存块
6.     for(int i = 0; i < 4; i++) { // 物理页帧分 4 片存入 4 个盘块
7.         b = bread(dev, blockno + i); // 将磁盘数据读到缓存块
8.         // cprintf("读出来吗? :%d", b->data[1]);
9.         memmove((void *)(pa + i * 1024), b->data, 1024); // 将缓存块数据写入物理页帧
10.        brelse(b); // 释放缓存块
11.    }
12. }
13. // 将 4096 字节的物理页帧写到 blockno 起始的连续 4 块盘块中
14. void write_page_to_disk(int dev, char *pa, uint blockno)
15. {
16.     begin_op();
17.     // begin_op();
18.     struct buf* b;
19.     for(int i = 0; i < 4; i++)
20.     {
21.         // printf("写出到磁盘\n");
22.         b = bread(dev, blockno + i); // 获取设备 1 上第 blockno+i 个盘块
23.         memmove(b->data, (void *)((uint64)(pa + i * 1024) | DMWIN_MASK), 1024); // 将数据移动到物理
内存上
24.         // memset(b->data, 0, PGSIZE);
25.         bwrite(b);
26.         // log_write(b); // 写磁盘
27.         // log_write(b);
28.         brelse(b); // 释放缓存块
29.     }
30.     end_op();
31. }

```

由于是简化实现，并没有专门建立交换区，而是直接在普通文件区找到连续 4 个盘块（对应一个页，共 4KB）来存储一个换出的页帧。xv6 磁盘读写是建立在缓存块已经分配的基础上，所以我们还要负责缓存块的分配和释放，借用 `bfree()` 和 `balloc()` 实现两个函数 `bfree4()` 和 `balloc4()`，为数据交换提供容量为 4KB 的缓存块，具体如代码 4-21 所示。

代码 4-16 `balloc4()`和 `bfree4()`

```

1.     uint
2.     balloc4(uint dev)
3.     {
4.         for(uint b = 0; b < sb.size; b += BPB) { // 逐块遍历位图
5.             struct buf* bp = bread(dev, BBLOCK(b, sb)); // 读取某一块位图
6.             for(uint bi = 0; bi < BPB && b + bi < sb.size; bi += 8) {
7.                 if(bp->data[bi/8] == 0) { // 连续 4 块未分配（1 字节）
8.                     bp->data[bi/8] = 0xff; // 按位取反，全 1
9.                     bwrite(bp); // 更新位图
10.                    // log_write(bp);
11.                    brelse(bp); // 释放缓存位图的缓存块
12.                }
13.                return b + bi; // 返回连续 4 块的第一块
14.            }
15.        }
16.        brelse(bp);
17.    }
18.    panic("balloc: out of blocks");
19. }
20. void
21. bfree4(int dev, uint b)
22. {
23.     for(uint i = 0; i < 4; i++) // 逐个调用 bfree 即可

```

```

24.         bfree(dev, b + i);
25.     }
26.

```

这样一来，物理页帧和磁盘盘块的交互接口算是完成了。

4.3.2. 缺页异常

由于我们将页面调出到磁盘后，进程再去访问就会发生缺页中断，我们首先要测试和验证一下 xv6 的缺页中断号。我们知道在 `sys_sbrk()` 函数中会调用 `growproc(n)` 来申请物理页帧，我们将其注释掉，重新运行 xv6，执行某个可执行文件（例如 `ls`）后会出现如下语句。

```

$ ls
usertrap(): unexpected trapcause 20000 pid=3
             era=0x00000000000018b0 badi=29c06061
             vm=16392

```

这是因为 `sh` (shell) 在执行外部命令 `ls` 的时候会调用 `sbrk()` 函数分配内存并使用，但由于我们注释掉了 `growproc()` 并没有为之分配内存，因此导致缺页异常。上面的提示说明缺页中断号是 2，`ls` 进程因缺页异常而被撤销。

既然要实现换入换出机制，我们索性实现延迟分配物理页帧——`sys_sbrk()` 只修改进程空间大小而不进行物理页帧的分配和页表的映射，也就是将请求式分页和交换结合。需要注意，替换子进程执行的 `exec`，`exec` 并没有使用 `sys_sbrk()`，也就是说装入新进程的 ELF 可执行文件过程不会和我们的交换机制相关，只有调用 `sys_sbrk()` 函数才会“假装”分配内存（只修改 `proc->sz`），才涉及我们实现的交换机制。对于 `sys_sbrk()` 分配的内存，直到真正访问对应内存区的时候才因 `pte` 没有有效映射而引起缺页，再由缺页中断程序执行分配操作。

修改 `kernel/trap.c` 的 `usertrap()`，添加缺页中断处理。

```

uint32 rcause = 0x10000;
uint32 wcause = 0x20000;
if(rcause == r_csr_estat() || wcause == r_csr_estat()) {
    pgfault();
}else{
    printf("usertrap(): unexpected trapcause %x pid=%d\n", r_csr_estat(), p->pid);
    printf("             era=%p badi=%x\n", r_csr_era(), r_csr_badi());
    printf("             vm=%d\n", r_csr_badv());
    p->killed = 1;
}

```

将中断处理 `pgfault()` 函数定义在 `kernel/vm.c`（因为涉及 `mappages()`），其工作步骤如下：

（1）如果缺页地址大于 `sz` 则表示非法地址（未分配），终止程序；

（2）如果地址小于 `proc->sz` 且大于 `swap_start`，则合法的可交换地址，需要进行处理。首先检查引起缺页的 `PTE` 中 `SWAPPED` 位，判定该页是否被调出内存。如果是被换出则启动交换机制；如果不是，则调用 `kalloc()` 生成一个物理页帧以供使用。如果 `kalloc()` 没有分配到物理页帧，则从进程空间中 `swap_start~sz` 之间找一个物理页帧调出磁盘。如果没有找到可供换出的页帧，则撤销本进程。

缺页时候需要找到一块已经有映射的页换出去，由于对应功能函数 `pagfault()` 和进程空间有关，我们将 `pgfault()` 代码放到 `vm.c` 中。当需要执行页帧交换功能时，从 `swap_start` 开始遍历，找到一个直接返回，中断程序 `pgfault()` 实现如代码 4-17 所示。

代码 4-17 `pgfault()`

```

1. // 缺页中断处理
2. void
3. pgfault()
4. {
5.     uint64 addr = PGROUNDDOWN(r_csr_badv()); // 获取导致中断的线性地址,要取整
6.
7.
8.     struct proc *proc = myproc();
9.
10.    char* mem;
11.
12.
13.
14.    if(r_csr_badv() > proc->sz) { // 越界
15.        printf("kalloc out of memory! %p %d\n",addr,proc->pid);
16.        proc->killed = 1;
17.        return;
18.    }
19.
20.
21.
22.    mem = kalloc(); // 分配一块物理页帧
23.    while(mem == 0) { // 没有页帧了
24.        printf("执行换出操作\n");
25.
26.        swapout(myproc()); // 进程空间找个页扔出去
27.        mem = kalloc(); // 再分配一次
28.    }
29.
30.    pte_t* pte = walk(proc->pagetable, addr, 1); // 查看 va 的页表项
31.
32.
33.    if((*pte & PTE_SWAPPED) == 0) { // 第一次分配
34.        memset(mem, 0, PGSIZE);
35.    }
36.    else { // 已分配但被换出的页
37.        printf("将物理页从磁盘调入内存\n");
38.        swapin(mem, pte); // 根据 pte 的盘块号将数据取回来
39.    }
40.
41.    *pte = (*pte & (~PTE_V));
42.    // 最后建立映射
43.    mappages(proc->pagetable, addr, PGSIZE, (uint64)mem, PTE_PLV | PTE_P | PTE_W | PTE_MAT | PTE_D);
44. }

```

`pgfault()` 中用到的 `r_csr_badv()` 函数需要在 `kernel/loongarch.h` 中添加实现。

```

1. static inline uint32
2. r_csr_badv()
3. {
4.     uint32 x;
5.     asm volatile("csrrd %0, 0x7" : "=r" (x) );
6.     return x;
7. }

```


其中的换出和换入功能由 `swapout()` 和 `swapin()` 负责，我们将它们实现在 `vm.c` 中，具体如代码 4-18 所示，同时需要在 `vm.c` 中包含 `spinlock.h` 和 `proc.h`。此处 `swapout()` 换出的时候我们从进程地址最高端开始往低地址扫描，直到 `proc->swap_start`。后面进行验证的时候，我们会反过来扫描体现不同交换算法的差异。

代码 4-18 `swapout()` 和 `swapin()`

```

1.  void
2.  swapout(struct proc *p) { // 换出一个物理页帧
3.      pte_t *pte;
4.      pagetable_t pgdir = p->pagetable;
5.      uint64 a = p->sz - 1; // 起始地址
6.      a = PGROUNDDOWN(a); // 向下取整
7.
8.
9.      for(; a >= p->swap_start; a -= PGSIZE) {
10.         pte = walk(pgdir, a, 0);
11.         if(*pte & PTE_P && ((*pte & PTE_SWAPPED) == 0)) { // 找到一个映射页,并且没有被换出
12.
13.             uint64 pa = walkaddr(pgdir, a);
14.
15.             uint blockno = balloc4(1); // 申请 4 个盘块
16.
17.             write_page_to_disk(1, (void *)pa, blockno); // 写入磁盘
18.
19.             kfree((void *) (pa | DMWIN_MASK)); // 释放对应的页帧
20.
21.             *pte = (blockno << 12); // 记录盘块号
22.             *pte = (*pte | PTE_SWAPPED); // 将其 swapped 位置 1
23.
24.             return;
25.         }
26.     }
27. }
28.
29. void
30. swapin(char* mem, pte_t *pte) { // 线性地址、页表项
31.     uint blockno = ((uint)*pte >> 12); // 取磁盘号
32.     read_page_from_disk(1, mem, blockno);
33. }
```

`swapout()` 函数中用到的 `PTE_SWAPPED` 宏需要在 `kernel/loongarch.h` 中定义。

```
#define PTE_SWAPPED (1L << 5) //swapped to disk
```

由于前面使用了延迟分配内存以及 `swapout()` 函数中修改了换出的 `pte` 的 `PTE_V` 标志，所以需要在 `uvmunmap()` 中做适当的修改，避免一些系统报错。

```

1.  void
2.  uvmunmap(pagetable_t pagetable, uint64 va, uint64 npages, int do_free)
3.  {
4.      uint64 a;
5.      pte_t *pte;
6.
7.      if((va % PGSIZE) != 0)
8.          panic("uvmunmap: not aligned");
9.
10.     for(a = va; a < va + npages*PGSIZE; a += PGSIZE){
11.         if((pte = walk(pagetable, a, 0)) == 0) {
12.             continue;
13.         }
14.         // panic("uvmunmap: walk");
```

```

15.     if(*pte == 0) {
16.         continue;
17.     }
18.     // if((*pte & PTE_V) == 0)
19.     //     panic("uvmunmap: not mapped");
20.     if(PTE_FLAGS(*pte) == PTE_V)
21.         panic("uvmunmap: not a leaf");
22.     if(do_free){
23.         if((*pte & PTE_SWAPPED) == 0) {
24.             uint64 pa = PTE2PA(*pte);
25.             kfree((void*)(pa | DMWIN_MASK));
26.         }
27.
28.     }
29.     *pte = 0;
30. }
31. }

```

4.3.3. 功能验证

接着我们进行简单的验证工作，编写代码 4-19 所示的应用程序 `swap-demo`，其内部操作安排如下：

（1）为了便于实验观察，我们先分配并使用掉系统的大部分物理页帧，直到剩下 1 个物理页帧。

（2）然后分配 4 个页，用于验证延时分配。

（3）分别给这 4 个页的第一字节写入数值 `a~d`。访问第 4 个页的第一字节并打印输出，然后访问第 1 到第 4 个页的第一字并输出，验证缺页功能和交换功能。

其中的 `bstat()` 用与打印系统剩余物理页帧数量，也就是前面提到的 `kmem` 结构体新增的 `count`。

代码 4-19 `swap-demo.c`

```

1.  #include "kernel/types.h"
2.  #include "kernel/stat.h"
3.  #include "user/user.h"
4.
5.  void
6.  mem(void){
7.      bstat();
8.
9.      char *addrTable[11];
10.
11.     for(int i = 0; i < 4; i++) {
12.         addrTable[i] = sbrk(4096);
13.     }
14.
15.     printf("-----\n");
16.
17.     for(int i = 0; i < 4; i++) {
18.         addrTable[i][1] = 'a' + i;
19.         bstat();
20.     }
21.
22.     printf("-----\n");
23.
24.     printf("访问第四个页，其内容是%c，不发生缺页异常\n", addrTable[3][1]);
25.

```

```

26.     printf("下面展示缺页异常的交换功能: \n");
27.
28.     for(int i = 0, cnt = 1; i < 4; i ++, cnt++) {
29.         bstat();
30.         printf("第%d 个页的内容是: %c\n", cnt, addrTable[i][1]);
31.         sleep(3);
32.     }
33.
34. }
35.
36. int
37. main(void)
38. {
39.     bstat();
40.     char *s = 0;
41.
42.     //先把页帧数量减少, 便于观察实验
43.     for(int i = 1; i <= 238; i++) {
44.         s = sbrk(4096);
45.         s[1] = 1;
46.
47.     }
48.
49.     bstat();
50.     mem();
51.     exit(0);
52.     return 0;
53. }

```

代码 4-19 的运行结果如屏显 4-2 所示。系统中物理页帧数量 239, 我们先分配了 238 个页帧并访问使用, 因此显示只剩下 1 个页帧。用 `sbrk(4096)` 分配 4 个页, 此时只是把本进程 `proc->sz` 的值增大, 并没有实际的分配。接着访问 4 个页的首字节会出现 4 次缺页异常, 由于还有 1 个空闲物理页帧, 所以第一次访问直接分配空闲的物理页, 但是后面 3 次访问由于系统空闲物理页为 0, 所以就需要 3 次换出操作。接着访问第 4 个页, 因为刚刚最后访问的是第 4 个页, 所以第 4 个页对应的数据还在内存, 并没有换出去磁盘, 所以并不会发生缺页。接着访问输出第 1~4 页的首字节, 都会发生缺页异常, 并依次出现换出、换入操作。

屏显 4-2 虚拟内存功能验证

```

物理页帧总数: 236
物理页帧总数: 0
-----
物理页帧总数: 0
-----

执行换出操作
物理页帧总数: 0
执行换出操作
物理页帧总数: 0
执行换出操作
物理页帧总数: 0
执行换出操作
物理页帧总数: 0
-----

访问第四个页, 其内容是 d, 不发生缺页异常
下面展示缺页异常的交换功能:
物理页帧总数: 0

```

```

执行换出操作
将物理页从磁盘调入内存
第 1 个页的内容是 : a
物理页帧总数: 0
执行换出操作
将物理页从磁盘调入内存
第 2 个页的内容是 : b
物理页帧总数: 0
执行换出操作
将物理页从磁盘调入内存
第 3 个页的内容是 : c
物理页帧总数: 0
执行换出操作
将物理页从磁盘调入内存
第 4 个页的内容是 : d

```

我们反转 `swapout()` 的扫描过程，从低地址 `proc->proc->swap_start` 到高地址 `proc->sz` 方向查找换出页，则发现最后分配的 4 个页帧第一次访问时有四次缺页，但第二次访问时不会引起缺页——因为换出的时前面分配的那些映射到低地址处页帧。此时结果如屏显 4-3 所示。

屏显 4-3 修改 `swapout()` 换出算法后的 `virtualMem` 输出

```

$ virtualMem
物理页帧总数: 236
执行换出操作
执行换出操作
物理页帧总数: 0
-----
物理页帧总数: 0
-----

执行换出操作
物理页帧总数: 0
执行换出操作
物理页帧总数: 0
执行换出操作
物理页帧总数: 0
执行换出操作
物理页帧总数: 0
-----

访问第四个页，其内容是 d，不发生缺页异常
下面展示缺页异常的交换功能：
物理页帧总数: 0
第 1 个页的内容是 : a
物理页帧总数: 0
第 2 个页的内容是 : b
物理页帧总数: 0
第 3 个页的内容是 : c
物理页帧总数: 0
第 4 个页的内容是 : d

```

4.4. 小结

本章的几个实验需要综合多方面的知识点才能完成，当读者完成上述的实验操作后，经历了一次高强度的综合训练。由于实验是基于简化的思路进行设计的，考虑得并不周全。读者可以基于自己的理解，发现其不足并自行进行完善。

在此恭喜读者完成了学习任务，经历了 xv6 实验课程锻炼并获得操作系统核心编程能力的提升！

练习

1. 为线程实验中的示例代码与“信号量”实验结合，使得多线程可以正确地访问临界资源。
2. 结合文件系统的知识将中级实验中的无名管道，修改为命令管道。
3. 修改调度器，使得 `scheduler()` 运行于当前进程的上下文，类似于 Linux 那样无需单独的一个调度器执行流。
4. 将用户 `id` 和文件访问权限联系起来，实现类似 Linux 下的文件访问权限管理功能。
5. 结合磁盘裸设备读写功能，将本章示例中的虚存交换从普通文件数据区转移到独立的磁盘分区上，从而不再和文件系统共用一个磁盘。

