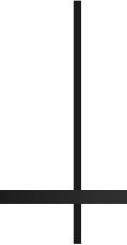


EECE7376: 操作系统 接口与实现



xv6 和 x86 概述



xv6 概述

- ❖ xv6是一个轻量级操作系统，第六版 Unix 的重新实现,用 ANSI C 编写,少数部分用汇编代码编写。
- ❖ xv6 针对 x86 和 RISC-V 计算机架构。 ❖ 它是为了教育目的在麻省理工学院的操作系统工程课程中创建的。编译它的时间非常短,因为它缺乏现代操作系统所期望的一些功能。

您可以在真实硬件上启动 xv6,但通常我们运行它使用QEMU x86 模拟器。



QEMU 概述

- ❖ QEMU (Quick Emulator 的缩写)是一个虚拟机 (VM) ,我们可以用它来模拟 x86 计算机架构。
- ❖ x86 最初是 Intel 指令集架构
为 8086 微处理器及其 8088 变体开发。
- ❖ 我们将使用这个 VM 来编译并运行 xv6 操作系统。

这使我们能够检查 xv6 系统调用和用户命令程序的实现,并可以选择通过修改这些程序来测试新想法。



在 COE Linux 网关中安装 xv6

QEMU 已安装在 CoE nu机器中。 登录 COE Linux 网关的流程: [https://](https://wiki.coe.neu.edu/xwiki/bin/view/Main/Linux_Machine_Help/)

wiki.coe.neu.edu/xwiki/bin/view/Main/Linux_Machine_Help/

❖如果您无权访问 COE Linux 帐户 ,请检查:

<https://coe.northeastern.edu/computer/general-services/request-a-coe-ece-computer-account/>

登录到 COE Linux 网关并在您的帐户中安装 xv6 后,运行以下命令: 1. `ssh -p27 nu`如果提示
输入密码,请输入您的 CoE 密码2. `git clone git://github.com/mit
-pdos/xv6-public.git`

如果步骤 2 出现连接超时错误,请尝试: `git clone http://github.com/
mit-pdos/xv6-public.git`



本地安装 QEMU 和 xv6

❖ 使用以下步骤在您的个人计算机 Linux 安装中安装 QEMU 和 xv6。

```
1.sudo apt-get update 2.sudo apt-
get install build-essential 3.sudo apt-get install gcc-multilib 4.sudo
apt install qemu qemu-utils qemu-kvm libvirt-clientsbridge-
utils 5.sudo apt- get install git-core 6.sudo git clone git://github.com/mit-pdos/xv6-public.git 7.cd xv6-
public 8.sudo make 9.sudo apt install qemu-system-
x86
```

如果步骤 6 出现连接超时错误,请尝试: sudo git clone
<http://github.com/mit-pdos/xv6-public.git>



运行xv6

从安装了 QEMU VM 和 xv6 的Linux shell

(对于 CoE 服务器,您需要首先运行`ssh -p27 nu`)使用以下命令开始在虚拟机上启动 xv6 操作系统:

```
$ cd xv6-公共
```

```
$ sudo make qemu-nox
```

注意:如果您使用的是`sudo` (对于超级用户而言) ,则不需要
CoE Linux 服务器。`nox`代表无图形

现在您正在运行带有 xv6 操作系统的 VM。我们将此 shell 称为VM/xv6 shell。

要关闭 VM,请按 `Ctrl+A`,然后按C。这将显示 QEMU 提示符。在这里,输入命令 “`q`” ,然后
按Enter

❖ 要重建 xv6,请转到 Linux shell 中的 `xv6-public` 目录并运行以下命令:

```
$ sudo 清理
```



Linux 的 Windows 子系统

您可以通过安装在 Windows 之上安装 Linux
适用于 Linux 的 Windows 子系统 (WSL), 然后是
Ubuntu Linux 发行版:

- <https://docs.microsoft.com/en-us/windows/wsl/install>
- <https://ubuntu.com/tutorials/ubuntu-on-windows#1-overview>



修改xv6用户命令

- ❖ xv6 用户命令之一是 ls。

当您从 VM/xv6 shell 运行 ls 时,将显示根目录的内容。每个条目包含以下四个字段:

1. 物品名称
2. 其类型 (1 = 目录、2 = 文件、3 = 设备)
3. inode 号,指包含文件元数据的数据结构 (将在持久性幻灯片中讨论)。
4. 物品尺寸。

例如,从 Linux shell (而不是 VM/xv6 shell)中,您可以通过修改 ls.c 来更改项目的显示方式

(位于 xv6-public 文件夹中)。

下次启动 VM/xv6 shell (使用 sudo make qemu-nox) 并运行 ls 时,将发生新的更改。



编辑 WSL Ubuntu 文件

您将需要 sudo 来编辑本地 WSL Ubuntu 中的文件设置。

❖ 从 Windows 编辑 WSL Ubuntu 文件（例如，使用

Notepad++）执行以下操作：

1. 在 Ubuntu 提示符下，向其他（网络上的用户）和组授予您需要编辑的文件的读/写权限： `sudo chmod og=rw <filename>`

2. 在 Windows 编辑器中打开以下文件夹中的文件： \\\wsl.localhost\Ubuntu\home\<user>

注意：需要更改文件权限的原因是上面的文件夹被视为 WSL 的网络文件夹。



添加新的 xv6 用户命令 (1 of 2)

❖ 向 xv6 添加新的用户命令并使其在 shell 中可用

提示,您需要添加新命令的 C 代码,并在 xv6 Makefile 中为其添加一个条目。

用户命令可以从任何地方运行,这与您在其中运行的常规程序不同。

必须从其文件夹中运行它们。

❖ 例如,在 Linux shell 的 xv6-public 文件夹中,创建一个

新文件名为 hello.c,内容如下:

```
#include “类型.h”
#include “用户.h”

int main(int argc, char **argv) printf(1,  Hello
world\n );
出口 () ;//不要使用返回 }
```

❖ 注意事项:

○ 标准C库在 xv6 中不可用。相反,一些实用程序源和标头提供了自己版本的常用函数,例如 printf 或 exit()

哦 在xv6中,所有用户空间应用程序都必须使用exit(),而不是简单地从main返回。



添加新的 xv6 用户命令 (2 of 2)

- ❖ 要将 hello 添加到当前用户命令列表中,请编辑文件 Makefile 并将名为 _hello 的条目添加到当前分配给 UPROGS 变量的值的末尾:

升级=\

.....
_你好\

- 要重新编译 xv6 以包含新命令,请使用 Linux shell 并从 xv6-public 文件夹运行:

```
$ sudo 清理  
$ sudo make qemu-nox
```

从 VM/xv6 shell 中,您可以通过以下方式测试新命令
从命令行运行 hello。



x86 通用寄存器

x86 是CISC ISA,具有用于非加载/存储指令的内存操作数

以及比 RISC 更复杂的寻址模式。

❖ x86 处理器有 8 个 32 位通用寄存器。 对于 EAX、EBX、ECX 和 EDX 寄存器,可

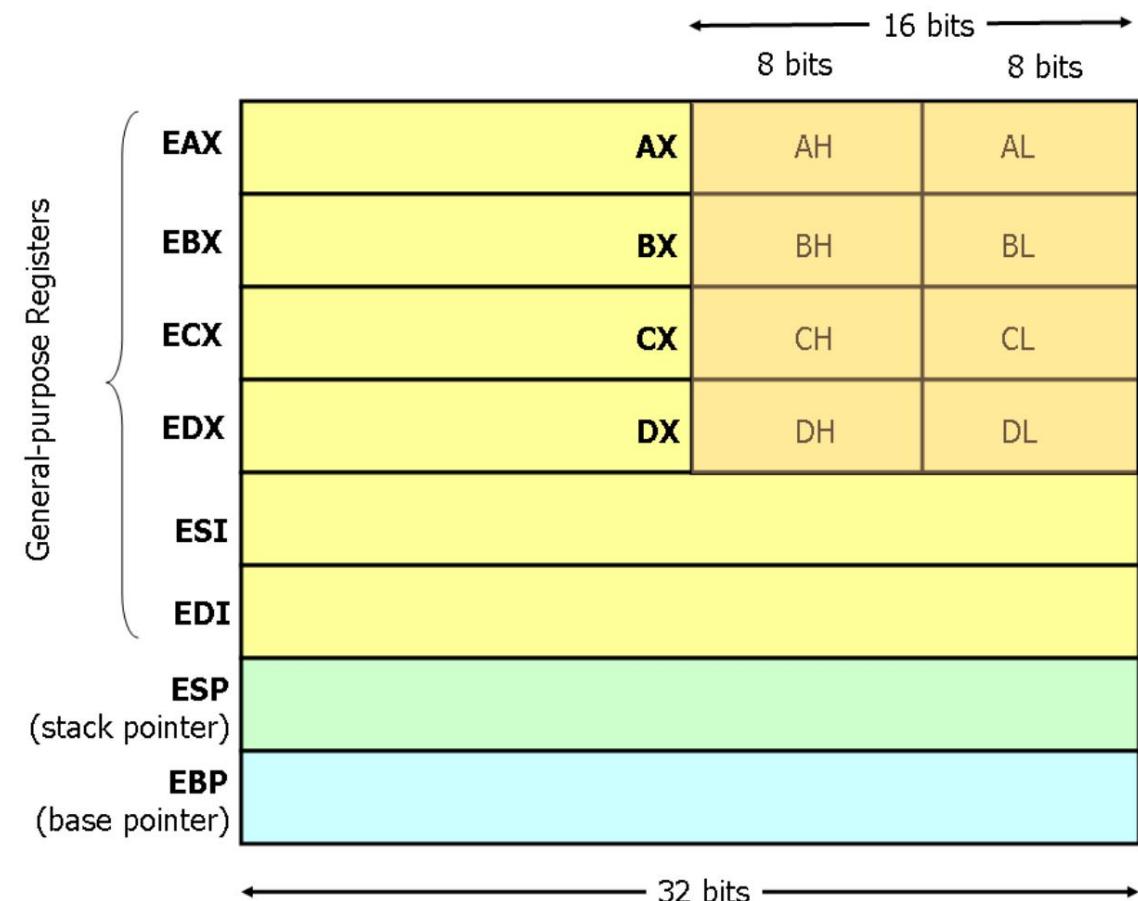
以使用分段。例如,EAX 的最低有效

2 个字节可以被视为一

个 16 位寄存器,称为 AX。

AX 的最低有效字节可用作称为

AL 的单个 8 位寄存器。





x86 数据传输

❖ 寄存器之间：

`movb %al, %ah ; %ah = %al` (第一个操作数是源,第二个操作数是目标) `movw %ax, %bx ; %bx = %ax`
`movl %eax, %ebx ; %ebx = %eax`

往返内存位置：

`movl %eax, (%ebp)` ;存储到内存 : $\text{mem}[\%ebp] = \%eax$
`movl %eax, 8(%ebp)` ;存储到内存 : $\text{mem}[\%ebp + 8] = \%eax$
`movl -4(%ebp), eax` ;从内存加载 : $\%eax = \text{mem}[\%ebp - 4]$

❖ 寄存器常量：

`movl $10, %eax` ; $\%eax = 10$
`movw $0x1000, %ax` ; $\%ax = 4096$ (0x1000 的十进制值)
`movb $ A , %ah` ; $\%ah = 65$ (A的ASCII码)



xv6 引导加载程序

引导加载程序是一个小型软件例程，在处理器启动后立即执行（在我们的例子中是在 QEMU VM 启动后立即执行）。

处理器被引导（例如，通过执行系统 BIOS 指令）从永久存储单元（例如硬盘驱动器或外部 USB 驱动器）的主引导记录运行引导加载程序软件。

❖ 引导加载程序负责加载代码

操作系统从永久存储单元到从地址 0x100000 开始的物理内存。

该地址加载的 xv6 代码最初是用 x86 汇编语言编写的，可以在文件 entry.S 中找到



xv6条目.S

❖ [entry.S](#)是一个x86汇编代码程序,它设置不同的处理器寄存器来配置xv6使用的内存分页结构和内核堆栈。

[entry.S](#) 末尾的代码跳转到函数 main() (最初编写在 main.c 中)并开始执行 xv6 内核的剩余代码。

大部分xv6 内核源代码是用 C 语言编写的,但也有一些代码是用 x86 汇编语言编写的。然而,加载到内存中执行的代码始终是这些源代码的可执行机器代码 (编译或汇编) 。



xv6 main.c

函数 main() 执行以下引导任务：

1. 调用函数 tvinit() (在 trap.c 中定义) 来初始化陷阱向量数组, 其中每个向量[i]包含处理中断 i 的代码的地址。向量数组本身在向量.S 中定义。一个特殊的条目是向量[64], 它处理系统调用中断。
2. 通过调用 proc.c 中定义的 userinit() 启动第一个用户进程, 该进程负责设置用户看到的环境, 包括 xv6 shell。
3. 调用函数 mpmain(), 通过调用函数 Scheduler() 来启动调度程序。函数 Scheduler() 永远不会返回 (它进入无限循环) 。

注意: 登录 Linux 后立即运行 ps 将显示 shell 进程 (例如 bash) 和 ps 进程本身。



系统调用

- ❖ 在 xv6 中运行进程涉及用户程序代码和系统代码的执行（通过系统（环境）调用/陷阱指令/中断）。

处理器在用户模式（低权限）下运行用户代码，而在内核模式（高权限）下运行系统代码。

- ❖ 处理器中的“模式位”区分处理器是运行用户代码还是内核代码。

在内核模式下，处理器允许执行更多特殊指令并访问受限内存区域。

内存页条目的保护位区分内存页是用于用户指令/数据还是用于内核指令/数据。

- ❖ 系统（环境）调用允许用户级进程向内核请求服务。一种控制如何在多个进程之间访问/共享关键资源的方法。



内核模式案例

❖ 在三种情况下,控制权必须从
用户程序进入内核态。

1. 系统调用:当用户程序请求操作系统服务时。

2. 软件异常:当程序执行非法操作时
行动。

非法操作的[示例](#)包括除以零、尝试访问内存以查找无效或不存在的页表条目等。

3. 硬件中断:当设备生成信号以指示它需要操作系统注意时。 ❖ 例如,时钟芯片可能每
100 毫秒产生一个中断,以允许内核通过其调度器实现 CPU 虚拟化。

另一个例子,当从磁盘读取一个数据块时,
生成一个中断以警告操作系统该块已准备好检索。



xv6/x86 中断处理

- ❖ x86 允许 256 个不同的中断来处理所有的前三个内核模式案例。
- ❖ xv6 将它们映射如下：
 中断 0-31 是为软件异常定义的。 中断 32-63 映射到 32 个硬件中断。
 ❖ 系统调用中断使用中断 64。

xv6 不允许用户进程使用 int 指令引发类似设备中断的中断。如果他们尝试，他们将遇到一般保护异常。



xv6 硬件中断

❖ xv6 是为具有多个处理器的系统而设计的。

例如,xv6 将键盘中断路由到第一个处理器（处理器 0）,将磁盘中断路由到系统上编号最高的处理器。

每个处理器都可以独立于定时器芯片接收定时器中断。xv6 在 lapic.c 中定义的 lapicinit() 函数中进行设置

指令 cli 通过清除 IF (中断标志)来禁用处理器上的中断,而 sti 则启用它们。

每个处理器上的调度程序都启用中断。然而,为了控制调度期间某些代码片段不被中断,xv6在这些代码片段期间禁用中断 (例如,switchuvm,它被调用来切换进程的地址空间)。



中断和设备驱动程序（3 中的 1）

驱动程序是操作系统中管理特定设备的代码 : 它告诉设备硬件执行操作 , 配置设备在完成时生成中断 , 并处理产生的中断。

在许多操作系统中 , 所有连接设备的驱动程序在操作系统中所占的代码量比核心内核还要多。

一些驱动程序在轮询和轮询之间动态切换

中断 , 因为使用中断可能会很昂贵 , 但使用轮询会导致延迟 , 直到驱动程序处理一个

事件。



中断和设备驱动程序 (3 中的 2)

轮询要求CPU (通过设备驱动程序代码)重复轮询设备以获取I/O 完成情况。

在许多计算机体系结构中,CPU 执行简单的指令来轮询设备:读取设备状态寄存器,进行逻辑与提取状态位,如果不为零则发出系统调用。

当重复尝试时,轮询会变得低效

但很少找到可供使用的设备。

在这种情况下,安排硬件控制器在设备准备好提供服务时通知 CPU 可能会更有效,这就是中断的思想。



中断和设备驱动程序 (3 of 3)

❖接收突发数据包的网络驱动程序可能会切换

从中断到轮询,因为它知道必须处理更多数据包,并且使用轮询处理它们的成本较低。一旦不再需要处理数据包,驱动程序代码可能会切换回中断,以便在新数据包到达时立即发出警报。

此外,网络驱动程序可能会将一个网络连接的数据包的中断传递到管理该连接的处理器,而将另一个连接的数据包的中断传递到另一个处理器。 ❖这种路由可能会变得相当复杂;例如,如果一些

网络连接的寿命很短,而其他连接的寿命很长,操作系统希望让所有处理器保持忙碌以实现高吞吐量。



x86 系统调用

- ◆ xv6 支持通过软件中断指令进行系统调用

综合 \$64

该中断指令在执行之前需要执行以下操作：

调用 : 寄存

器 %eax 包含系统调用号

寄存器 %ebx、%ecx、%edx、%esi、%edi、%ebp 包含

可选参数（例如，在屏幕上写入字符串的系统调用，其参数是要写入控制台的字符串的地址及其长度）。

中断指令可选择返回寄存器 %eax 中的值

示例：以下代码是系统调用退出，以退出代码 0 终止进程：

movl \$2, %eax ; 2 表示 SYS_exit

movl \$0, %ebx ; 代码 = 0

综合 \$64



一般系统调用时间线

❖ 中断指令停止用户的执行

处理指令并开始执行称为中断处理程序的新指令序列。

❖ 在启动中断处理程序之前,处理器会保存其中断处理程序寄存器,以便操作系统从处理程序返回时可以恢复它们。

在中断处理程序之间转换的一个挑战是处理器应该从用户模式切换到内核模式,然后再切换回来。

术语 “陷阱”和 “中断”通常可以互换使用。



系统调用与用户库函数

对于程序员来说,系统调用看起来就像对库函数的任何其他过程调用。

- ❖ 如果性能是一个问题,并且如果无需系统调用即可完成任务,则程序将运行得更快。为什么?
- ❖ 每个系统调用都涉及从用户模式切换到内核模式,然后再切换回来的开销时间。

操作系统可能会调度另一个进程来运行

当系统调用完成时,进一步减慢调用进程的执行速度。

另一个编程技巧是尽量减少系统调用的次数。

例如,您可以将输出合并到一个字符串中,然后在循环外打印该字符串,而不是在循环中使用 `printf`。



xv6 支持的系统调用 (1 of 2)

fork() 退	创建一个流程。
exit() 等待()	终止当前进程。
	等待子进程退出。
管道 (fds)	创建一个管道并向其返回文件描述符。
读取 (fd,buf,n)	从打开的文件中读取n个字节到buf中。
杀死(pid)	终止进程pid。
exec(filename, argv)	加载文件并执行它。
fstat(fd)	返回有关打开文件的信息。
chdir(目录名) dup(fd)	更改当前工作目录。
getpid()	重复的文件描述符。
sbrk(n) 睡眠	返回当前进程的 id。
(n)	将进程的内存增加n个字节。
	休眠n秒。



xv6 支持的系统调用 (2 of 2)

正常运行时间()	返回时钟节拍中断的数量自开始以来。
open(filename, flags) write(fd, buf, n) mknod(name,major,minor)	打开文件,标志表示读/写。 将n个字节写入打开的文件。
创建一个设备文件。	
取消链接 (文件名)链接 (src, dest)mkdir (目录名)关闭 (fd)	删除一个文件。 为文件创建硬链接。 创建一个目录。
	关闭一个文件。

- ❖ 上述每个系统调用都有一个唯一的系统调用号。
- ❖ xv6 需要使用寄存器 %eax 将这个数字传递给 int \$64。 ❖ 请注意,许多系统调用需要传递一个或多个参数。

xv6是否使用其他寄存器来传递这些参数?



xv6 系统调用号

❖ xv6 系统调用

数字在 syscall.h 中定义,如图所示。

❖ 这些是在调用 int \$64 之前需要加载到寄存器 %eax 中的数字

```
// 系统调用号
#define SYS_fork 1
#define SYS_exit2
#define SYS_wait 3
#define SYS_pipe 4
#define SYS_read 5
#define SYS_kill 6
#define SYS_exec 7
#define SYS_fstat 8
#define SYS_chdir 9
#define SYS_dup 10
#define SYS_getpid 11
#define SYS_sbrk 12
#define SYS_sleep 13
#define SYS_uptime 14
#define SYS_open 15
#define SYS_write 16
#define SYS_mknod 17
#define SYS_unlink 18
#define SYS_link 19
#define SYS_mkdir 20
#define SYS_close 21
```



xv6 系统调用定义

- ❖ xv6 使用 usys.S 中所示的宏生成 eax 初始化和 int \$64 指令。

- ❖ 要将其存储在寄存器eax 中，中断号使用SYS_ ## 名称从 syscall.h 中检索，其中## 是合并运算符。

系统调用是使用

`int $T_SYSCALL` (\$T_SYSCALL 在 traps.h 中定义为 64)

代码体中的每一行，例如`SYSCALL(fork)`，都将被宏扩展替换。xv6 不使用 int \$64 调用的参数，因为它将这些调用的参数保留在用户堆栈中并稍后

获取它们。

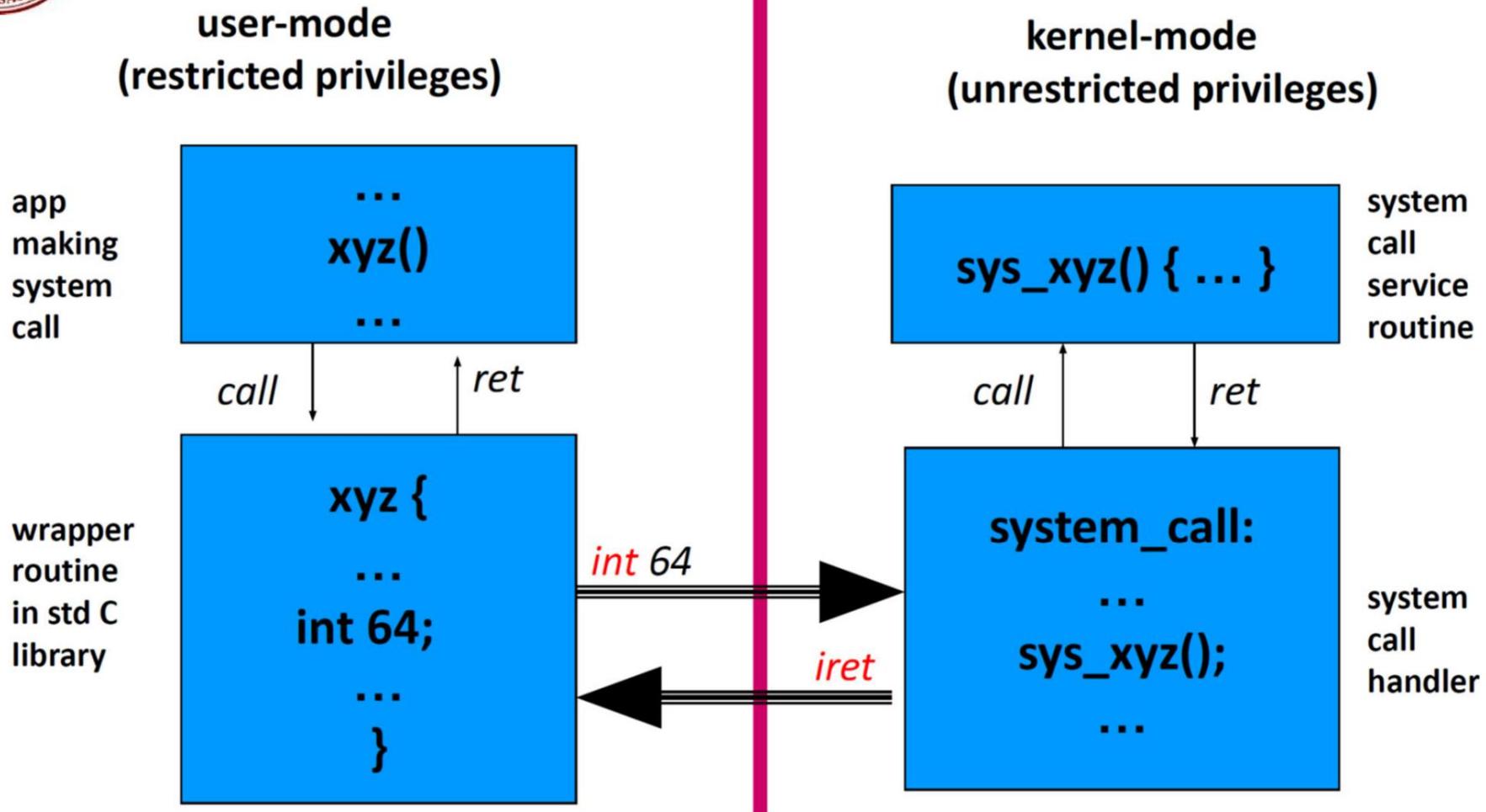
```
#include "system调用.h"
#include "traps.h"

#define SYSCALL(名称) \
    .全局名称; \
    姓名: \
        movl $SYS_ ## 名称, %eax; \
        int $T_SYSCALL; \
雷特
```

- 系统调用 (fork)
- 系统调用 (退出)
- 系统调用 (等待)
- 系统调用 (管道)
- 系统调用 (读)
- 系统调用 (写)
- 系统调用 (关闭)
- 系统调用 (杀死)
- 系统调用 (执行)
- 系统调用 (打开)
- 系统调用 (mknod)



xv6 中系统调用的用户调用



需要C语言的包装例程,以便用户程序员不需要处理系统调用 (`int $64`) 寄存器设置的细节。

`sys_xyz()` 服务例程通常调用另一个名为 `xyz()` 的函数来完成“服务”,但是这个内核模式 `xyz()` 例程与用户模式 `xyz()` 包装例程不同。



xv6进程结构

`struct proc` (在 `proc.h` 中)是每个进程的数据结构：

```
enum procstate // 所有可能的进程状态
    { 未使用、胚胎、睡眠、可运行、正在运行、僵尸 };
```

// 每个进程的数据结构

结构体过程{

<code>uint 大小;</code>	// 进程内存大小 (字节)
<code>pde_t* pgdir;字符</code>	// 页目录
<code>*kstack;枚举 prostate</code>	// 该进程的内核堆栈底部
<code>状态; PID;结构体过程*父级;结构陷阱</code>	// 进程状态
<code>框架 *tf;结构</code>	// 进程号
<code>上下文*上下文;无效*陈; int 被杀死;</code>	// 父进程
<code>结构文件 *ofile[NOFILE]; // 打开文件</code>	// 当前系统调用的陷阱帧
	// 在 <code>proc.h</code> 中定义,包含 <code>swtch()</code> 所需的寄存器
	// 如果非零,则在 <code>chan</code> 上休眠
	// 如果非零,则已被杀死

<code>结构 inode *cwd;</code>	// 当前目录
-----------------------------	---------

<code>字符串名[16];</code>	// 进程名称 (调试)
------------------------	--------------

};



xv6 进程表

进程表 ptable (在 proc.c 中)被实现为 struct proc 类型的元素数组。常量 NPROC (在 param.h 中)设置为 64,限制了 xv6 中活动进程的最大数量：

```
结构体{
    结构体自旋锁;
    struct proc proc [NPROC];
} ptable;
```

进程表中的条目通过将其状态字段设置为值 UNUSED (表示与活动进程不相关的记录)来初始化。

- ❖ ptable 中的锁字段是在多个的情况下需要的处理器,这样只有一个处理器可以获得锁,以避免对同一个表的并发访问。



C 中的函数指针

我们通常使用C指针来指向内存中存储的数据。 ✦ C 中的指针也可以用来指向一个程序的开头

内存中的函数指令。示例：

```
#include <stdio.h> float avg (int
x, int y) /* 求两个数平均值的函数 */ float a; a = s /2.0;
    整数; s
    = x + y; 返回
    一个; }

int main() { float
    (*avgP)(int, int); } /* 函数指针声明 */ /* 指针赋值 */ avgP = avg; 浮点 av =
    指针进行函数调用
                                avgP (10, 15); /* 使用其
    */ printf ( Average = %f\n , av); 返回 0; }
```



C 中的函数指针数组

函数指针数组可以起到 switch 语句的作用来做出决定,如下例所示:

```
#include <stdio.h>
float avg (int x, int y) { return (x + y)/2.0; }
float max (int x, int y) { return (x > y)? x:y; }
float min (int x, int y) { return (x < y)? x:y; }

int main() {
    // 函数指针数组声明和初始化float (*afp [ 3])(int, int )
    = {avg, max, min}; // 此处大小 3 是可选的for (int i = 0; i < 3; i++) printf(    Function %d result
    = %f\n    , i, afp [ i](10, 15));返回0; }
```

您可以通过指定要初始化的数组索引来乱序初始化上述代码中的数组元素。例子:

`float (*afp [3])(int, int) = {[2]平均值, [0]最大值, [1]最小值};`这里,上面的循环将按顺序调用 max、min 和 avg。



xv6 系统调用处理程序表

❖ 定位系统调用处理程序

xv6内核中的函数,使用了系统调用表数据结构。

❖ xv6 系统调用表是一个数组

函数指针,如图所示,在 `syscall.c` 中定义

❖ 该表的索引是系统调用号。我们可以省略索引的显式定义吗?

❖ 注意所有函数都没有

参数。像 `sys_kill` 这样的函数不需要一个参数 (`pid`) 吗?

系统调用处理程序的实际实现并不在此文件中。因此,需要

添加如下多条语句:

`extern int sys_exit(void);`

外部 `int sys_fork(void);`

EECE7376 - 艾马德·阿博埃拉博士

静态 `int (*syscalls[])(void) = {`

```
[SYS_fork] sys_fork,
[SYS_退出] system_exit,
[SYS_等待] system_wait,
[SYS_管道] system_pipe,
[SYS_read] sys_read,
[SYS_kill] sys_kill,
[SYS_exec] sys_exec,
[SYS_fstat] sys_fstat,
[SYS_chdir] sys_chdir,
[SYS_dup] sys_dup,
[SYS_getpid] sys_getpid,
[SYS_sbrk] sys_sbrk,
[SYS_睡眠] system_sleep,
[SYS_正常运行时间] system_normal_time,
[SYS_打开] system_open,
[SYS_write] sys_write,
[SYS_mknod] sys_mknod,
[SYS_unlink] sys_unlink,
[SYS_link] sys_link,
[SYS_mkdir] sys_mkdir,
[SYS_close] sys_close,
};
```



处理国际\$64

❖ int \$64 指令被处理

通过vector64中的vectors.S如图所示：

处理程序跳转到所有陷阱

trapasm.S,将所需的寄存器值压入堆栈并调用 trap.c 中的 trap 函数,其中包含堆栈中 trapframe 结构的地址 (struct trapframe 在 x86.h 中定义)

❖ 陷阱中显示的代码段

函数存储进程陷阱帧,然后调用 syscall()

syscall.c 中的函数

注意:函数 myproc() 定义在 proc.c 中
并返回指向当前CPU的struct proc (在proc.h中定义)的指针。

矢量64:

普什尔\$0

普什尔\$64

jmp所有陷阱

◦ ◦ ◦ ◦ ◦

.globl 所有陷阱

所有陷阱:

◦ ◦ ◦ ◦ ◦

调用trap(tf),其中tf=%esp

推%esp

呼叫陷阱

◦ ◦ ◦ ◦ ◦

无效陷阱 (struct trapframe *tf){

if (tf->trapno == T_SYSCALL){

if (myproc()->killed) exit();

myproc()->tf = tf;

系统调用 () ;

if (myproc()->killed) exit();

返回; }



xv6 syscall() 函数

❖下面是syscall.c中的syscall()函数

```
无效系统调用 (无效) {
    整数;
    struct proc *curproc = myproc();
    num = curproc->tf->eax;
    if(num > 0 && num < NELEM(系统调用) && 系统调用[num]) {
        curproc->tf->eax = syscalls[num]();
    } else
        cprintf( %d %s: 未知的系统调用 %d\n ,
                  curproc->pid, curproc->name, num);
    curproc->tf->eax = -1;
}
```

❖该函数从陷阱帧中加载系统调用号,其中包含
保存的%eax。

如果 num 是一个有效数字 (介于 0 和最大系统调用数之间)并且如果它
syscalls 表中的服务例程不为空,则调用该服务例程。

该函数的返回值存储回用户进程陷阱帧中的寄存器 eax 中。



系统调用参数

xv6 不使用 int \$64 系统调用的参数,因为处理程序表中的所有函数必须具有相同的签名。

相反,它将这些调用的参数保留在用户堆栈中,并使用 syscall.c 中定义的以下辅助函数之一获取它们。 所有函数都以参数n开头,以便函数获取第 n 个

来自用户堆栈的系统调用的参数。

- argint(int n, int *ip) 获取 32 位参数作为整数。 ○ argptr(int n, char **pp, int size) 获取参数并检查该参数是否是有效的用户空间指针。

- argstr(int n, char **pp) 将参数解释为指针
并确保指针指向以 NULL 结尾的字符串,并且完整的字符串位于地址空间的用户部分内。

- argfd(int n, int *pf, struct file **pf) 定义于 sysfile.c 并使用 argint 检索文件描述符编号,检查它是否是有效的文件描述符,并返回相应的 struct file。



xv6 系统调用处理程序实现

不同系统调用的实际处理在 [sysproc.c](#) 中实现,您可以
在其中找到诸如所示的 sys_kill 等函数的实现

```
int sys_kill (void) { int pid; } if
  (argint(0,
  &pid) < 0) return -1; return kill (pid);
```

}

[函数体](#)使用辅助函数 `argint` 从用户堆栈中读取其参数,而不是从当前活动的内核堆栈中读取。因
此,实现系统调用的函数在其标头中不使用显式参数。

[注意:](#) `sys_kill()` 函数调用了 `kill()` 函数,该函数是杀死进程的实现,定义在 `proc.c` 中



Kill() 函数的实现

该函数的主要任务是将进程pid的killed属性设置为1，
如果它正在休眠，则将其唤醒。

❖然而,被杀死的进程不会退出,直到它通过系统
调用或由于计时器 (或其他设备)中断进
入或离开内核。

回想一下,中断处理程序在trap.c中调用trap ,在
从syscall()调用返回后,它会检查killed属性,
如果它为1,则退出进程。

```
// 终止具有给定 pid 的进程。
// 进程在返回用户空间之前不会退出 // (请参阅 trap.c 中的
trap) 。 int Kill (int pid ) { struct proc * p;
获取(&ptable.lock); // 在 proc 表中搜
索 PID 匹配for (p =
ptable.proc; //proc 是进程数组p <
&ptable.proc [NPROC]; p++){ if ( p->pid ==
pid){ p->被杀= 1; // 如有必要,将进程从睡眠状态唤醒。 if ( p->state ==
SLEEPING ) p->state = RUNNABLE;释放(&ptable.lock);返
回0;
}
}
释放(&ptable.lock);返回- 1;
}
```



exit() 与 exit() 比较杀 ()

❖ exit() 允许进程自行终止,而kill()

让一个进程请求另一个进程 (受害者)终止。

要终止进程,exit() 对进程执行一些任务,包括关闭所有打开的文件、将其状态设置为 ZOMBIE 以及唤醒其父进程。

对于kill()来说,直接终止受害者进程会过于复杂,因为受害者可能正在另一个CPU上执行,也许正在对内核数据结构进行敏感的更新序列。



xv6 用户库

❖ 希望进行系统调用（例如 sleep()）的用户应用程序

调用所示 “user.h”文件中声明的相应 C 函数原型。

❖ 这些只是 C 编译器用来传递参数的原型（通过将它们压入堆栈）, 并且没有它们的C 实现。

而是系统会调用usys.S中宏生成的对应汇编代码

(其中显示的函数名称被定义为“全局”)

```
// 系统调用
int fork(void);
int exit(void) __attribute__((noreturn));
int 等待 (无效) ;
int 管道(int*);
int write(int, const void*, int);
int 读取(int, void*, int);
int 关闭 (int) ;
int 杀死 (int) ;
int exec(char*, char**);
int open(const char*, int);
int mknod(const char*, 短, 短);
int 取消连接(const char*);
int fstat(int fd, struct stat*);
int 链接(const char*, const char*);
int mkdir(const char*);
int chdir(const char*);
int dup(int);
int getpid(void);
字符* sbrk(int);
int 睡眠 (int) ;
int 正常运行时间 (无效) ;
```



修改系统调用()

在下面的示例中,我们将修改系统的陷阱处理程序
调用以使 xv6 在任何系统调用 fork、wait 或 exit 被调用时打印一条消息。

❖ 在 Linux shell 中,编辑 syscall.c 并修改其 syscall()

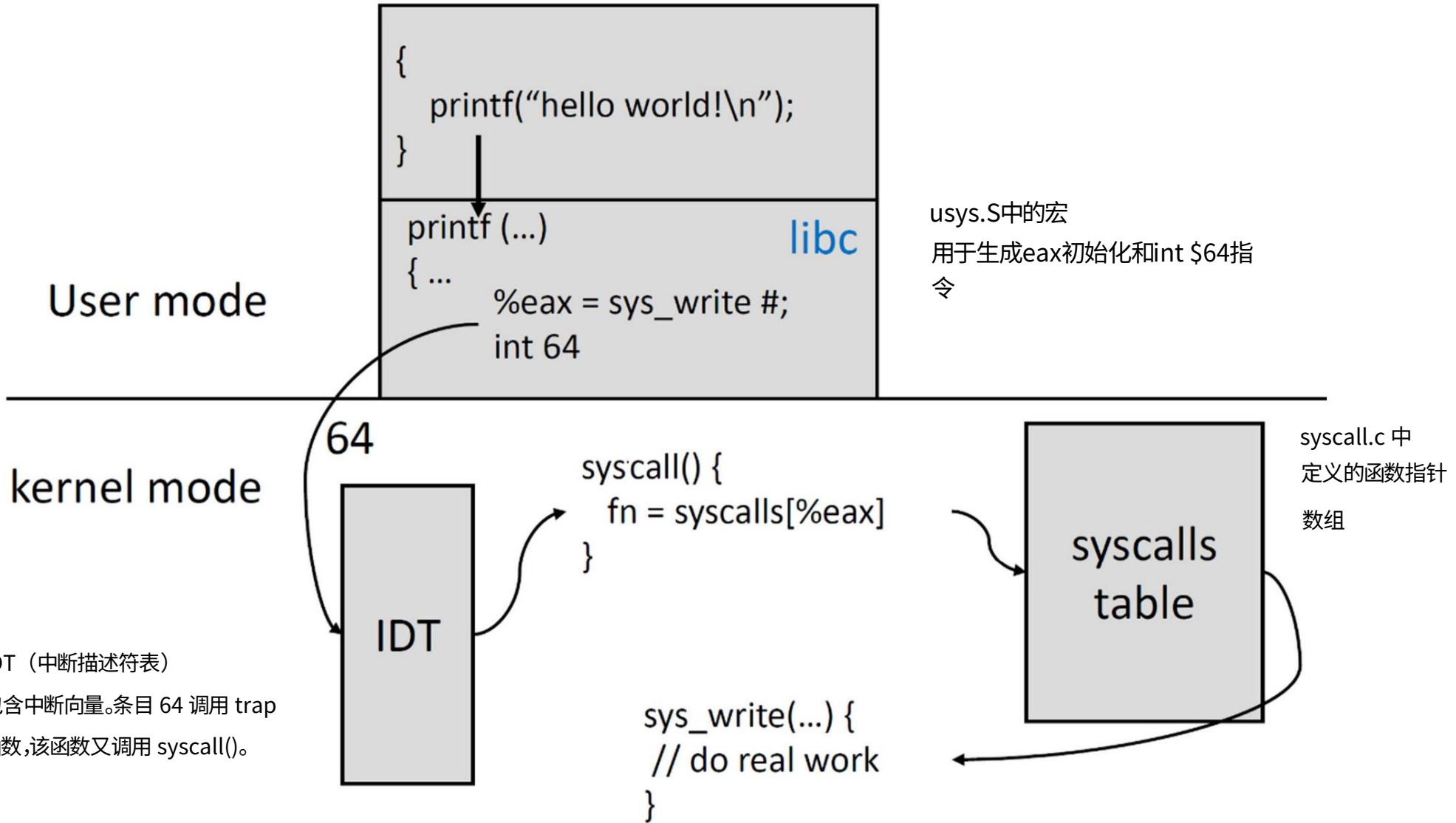
函数通过添加以下红行: (注意:在内核代码中,C 函数 printf 不可用。相反,内核代码应该调用它自己的 printf 版本,称为 cprintf (前缀 c 代表控制台))

```
if(num > 0 && num < NELEM(系统调用) && 系统调用[num]) {  
    if (num == SYS_fork) cprintf(    -----> 系统调用 fork\n    );  
    if (num == SYS_wait) cprintf(    -----> 系统调用等待\n    );  
    if (num == SYS_exit) cprintf(    -----> 系统调用退出\n    );  
    curproc->tf->eax = syscalls[num]();  
} 别的 {
```

❖ 从 VM/xv6 shell 中,您现在可以测试不同的命令并跟踪调用 fork、wait 和 exit 的时间顺序。



系统调用示例:printf()





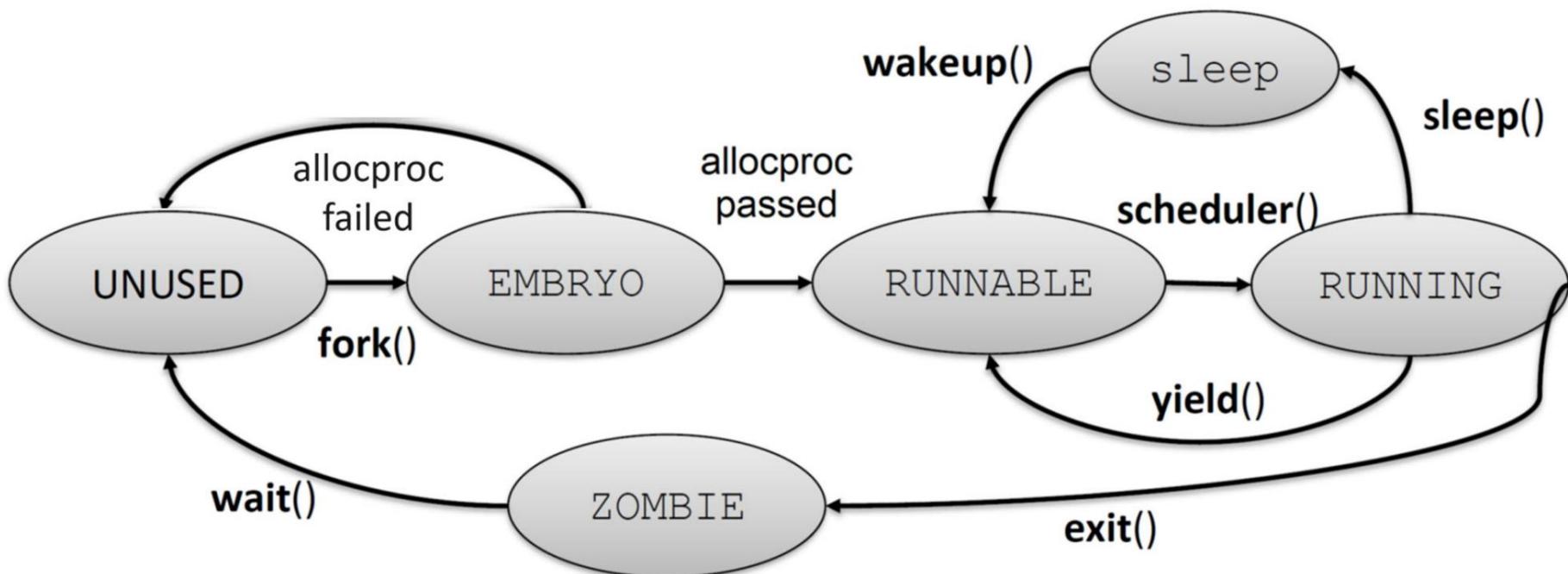
向 xv6 添加系统调用

- ❖ 根据到目前为止 xv6 系统调用的讨论,要向 xv6 添加新的系统调用,需要修改以下文件:
 1. **syscall.h** 为新的系统调用添加唯一的数字标识符。该数字稍后用于函数指针数组中的索引。是调用 int \$64 之前需要加载到寄存器 %eax 中的数字
 2. **usys.S** 为新的系统调用添加一行,以便由该文件中的宏生成所需的指令。
 3. **syscall.c** 将系统调用函数 (处理程序)的新条目添加到函数指针的系统调用表 (数组)中。还要在此文件中添加系统调用函数声明的 “extern”语句。
 4. **sysproc.c** → 添加该处理程序的实现。
 5. **user.h** → 添加新系统调用的用户级原型这里。



xv6 进程生命周期 (1 of 2)

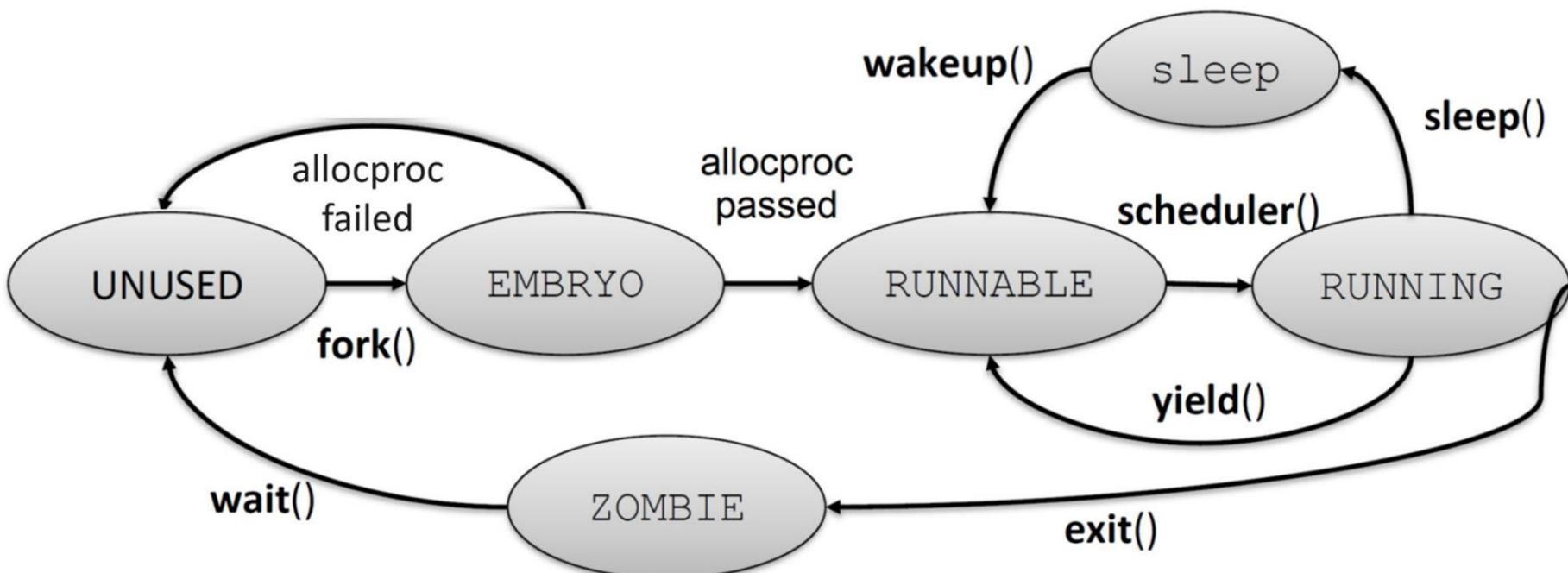
- 该图总结了 xv6 中触发进程状态转换的函数。函数在 proc.c 中
- 新分叉的进程处于 EMBRYO 状态。如果进程内存分配失败,allocproc 会将分配给该进程的记录状态更改回 UNUSED,否则将其设置为 RUNNABLE 状态。





xv6 进程生命周期 (2 of 2)

- ❖ 当一个进程退出时,它会切换到ZOMBIE状态,直到父进程调用 wait 来了解退出情况。然后,父进程负责释放与该进程关联的内存,并将其在ptable 中的条目更改为 UNUSED。
- ❖ 僵尸进程是指死亡的进程 (即完成其执行) ,但它的进程父级不通过 wait() 检查它。
孤儿进程是指其父进程死亡的进程。





xv6 第一个用户进程

❖第一个用户进程是 init.c 中定义的 “init” 进程。 ❖为了准备 init 进程, userinit() 负责

在进程表中为其分配一个未使用的条目。 它将该条目的 proc 结构体的状态字段设置为

RUNNABLE 以便 xv6 调度程序可以对其进行调度。

p->状态=可运行；

❖一旦 userinit() 完成执行, init 就准备好运行。 ❖一旦调度程序将其移至 RUNNING 状态, init

进程分叉一个子进程,该子进程转换为 xv6 “shell”。

❖从 xv6 “shell” 进程中, 用户可以 “创建” (fork + exec) 更多流程。

❖调度程序将处理这些多个任务的执行
流程。



xv6 调度程序 (1 / 2)

以下循环位于调度程序函数中,在 proc.c 中定义 调度程序永远不会返回,因为它无限循环,实现一个循环

罗宾调度策略

```
for(;;){ sti(); //
```

启用中断,因为 xv6 在持有锁时禁用 CPU 上的中断
`acquire(&ptable.lock); // 支持多处理器访问表` // 循环进程表寻找要运行的进程
`for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){ if(p->state != 可运行) 继续; c->proc = p; // 设置每个CPU当前进程变量`
`switchuvm(p); // 切换到进程的页表 (即地址空间) p->state = RUNNING; swtch(&(cpu->调度器), p->上下文); // 进程p恢复运行`
`switchkvm(); // 切换回内核页表 // 进程完成运行。它应该在返回之前更改其 p->state。 c->proc = 0; } 释放(&ptable.lock); }`



xv6 调度程序 (2 of 2)

xv6 调度程序通过检查 ptable 中的所有进程来查找处于 RUNNABLE 状态的进程来启动循环。

但在这样做之前并且为了安全起见（由于多处理器情况下的并发访问）,调度程序需要获取锁。

```
获取(&ptable.lock);
for(p = ptable.proc; p < &ptable.proc[NPROC]; p++){
    if(p->state != RUNNABLE) 继续;
    c->proc = p;
    ....
```

c 是指向 struct cpu 的指针类型,它记录了处理器 c 上当前运行的进程等信息。



进程切换 (1 of 3)

- ❖ 调度程序使用以下命令切换到所选进程：

```
c->proc = p;  
switchuvm(p); // 切换到进程地址空间  
p->状态=运行;  
swtch(&(cpu->scheduler), p->context); // 进程p恢复运行  
开关kvm(); // 切换回内核地址空间
```

- ❖ 调度器调用 swtch 函数 (定义在 swtch.S 中) 将当前 CPU 状态 (上下文) 切换到即将运行的进程的状态： `swtch(&(cpu->scheduler), p->context);`
- ❖ 注意，调度程序上下文不属于任何进程对象，因此在上述切换中使用了 CPU 上下文。



进程切换 (2 of 3)

- ❖ swtch 函数的原型 (定义在 swtch.S 中)为：

void swtch(结构上下文**旧, 结构上下文*新);

- ❖ 它将当前寄存器保存在堆栈上, 创建一个结构体 context, 并将其地址保存在*old中。将堆栈切换到新的并弹出以前保存的寄存器。

cpu->scheduler 和 p->context 都是 proc.h 中定义的 struct context 类型, 如下所示:

- ❖ 从一种环境切换到另一种环境
基本上涉及保存当前 (旧)
上下文寄存器并恢复新上下文之前保存的寄存器。

```
结构上下文{
    uint edi;
    uint esi;
    uint ebx;
    uint ebp;
    uint eip;
};
```

这包括寄存器 eip, 它具有要恢复的指令地址 (因此, 在 swtch 之后, CPU 运行所选进程)



进程切换 (3 of 3)

当一个进程正在运行时（通常在用户模式下），最终它将由于以下事件之一而进入内核模式：

- Ø 系统调用
- Ø 软件例外
- Ø 硬件中断（例如定时器中断）

❖ sched()函数处理完这些事件后

（在 proc.c 中定义）被调用，这又通过调用以下命令将上下文从当前进程切换到调度程序上下文： `swtch(&p->context, mycpu()->scheduler);`

这将从其 `switchkvm()` 代码行恢复调度程序。

❖ 当进程需要放弃CPU时，进程会执行以下操作：

- Ø 将其 `p->state` 更改为 SLEEPING 或 ZOMBIE。
- Ø 调用 `swtch`（与上面的方式相同）保存自己的上下文并返回到调度程序上下文以将控制权转移回调度程序。



x86 分页

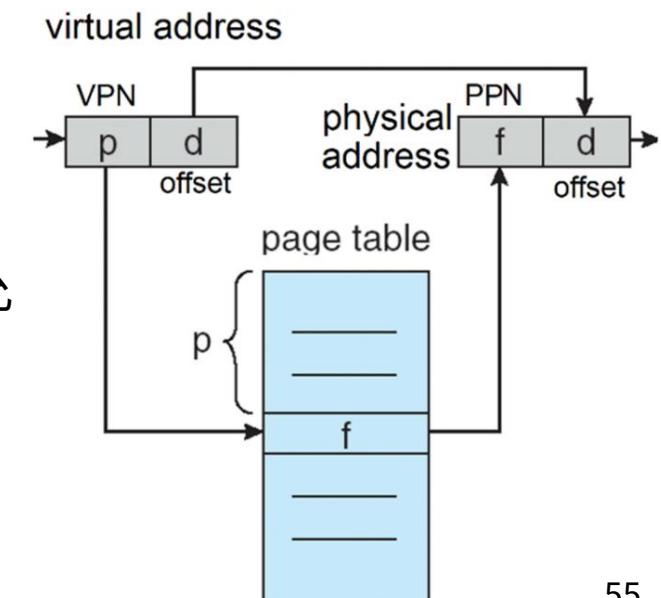
- ❖ x86 指令 (用户和内核)操作虚拟地址。
- ❖ x86 页表硬件将每个虚拟地址映射到物理地址。
- ❖ x86 页表在逻辑上是一个由 220 个页表条目(PTE) 组成的数组。每个 PTE 包含一个 20 位物理页号(PPN*)和一些标志。以下是 xv6 mmu.h 中定义的标志

PTE_P 指示 PTE 是否存在:如果未设置,则对该页面的引用会导致错误。

PTE_W 控制是否允许指令向页面发出写操作;如果未设置,则仅允许读取和取指令。

❖ PTE_U 控制是否允许用户程序使用该页面;如果清除,则仅允许内核使用该页面。

* 在内存虚拟化幻灯片中将其称为物理帧号 (PFN)。





x86 页表硬件

❖ 如图所示,页表

以两层树的形式存储在物理内存中。

❖ 树的根是一个 4096-

字节页目录,包含 1024 个类似
PTE 的页表页引用。

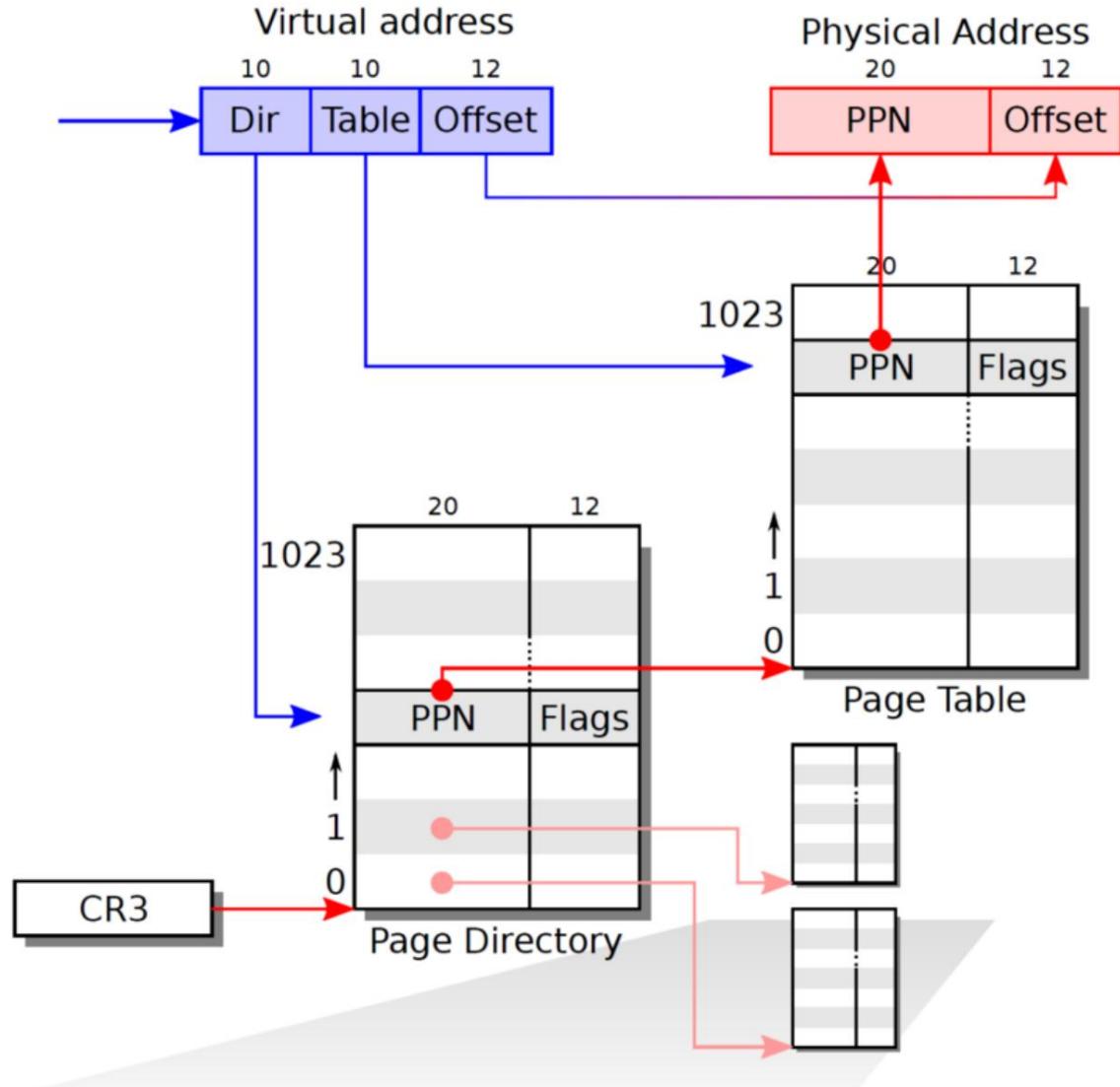
❖ 每个页表页都是一个 1024 个
32 位 PTE 的数组。

寄存器 cr3 的最高 20 位包含页目录基地址。

❖ 任意的物理地址

目录中的条目格式为： [cr3]20[Dir]10[00]

2



PPN:物理页码
PTE:页表项



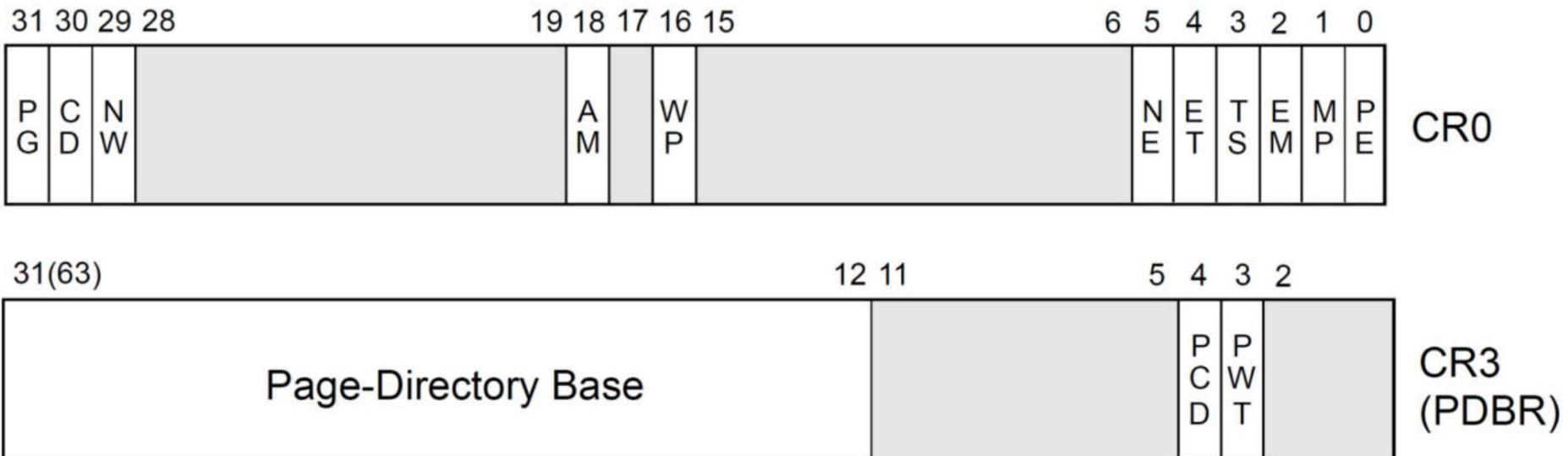
x86 分页支持寄存器 (1 of 2)

涉及多个寄存器和内存位置

x86中的地址转换过程:

寄存器 %cr0 包含位置 31 处的位 PG。当该位为 1 时,分页处于活动状态,并且 %cr3 包含页目录基地址位于其 20 个最高有效位中。

- ❖ 当 PG 位为 0 时,禁用分页,处理器仅使用物理分页地址。

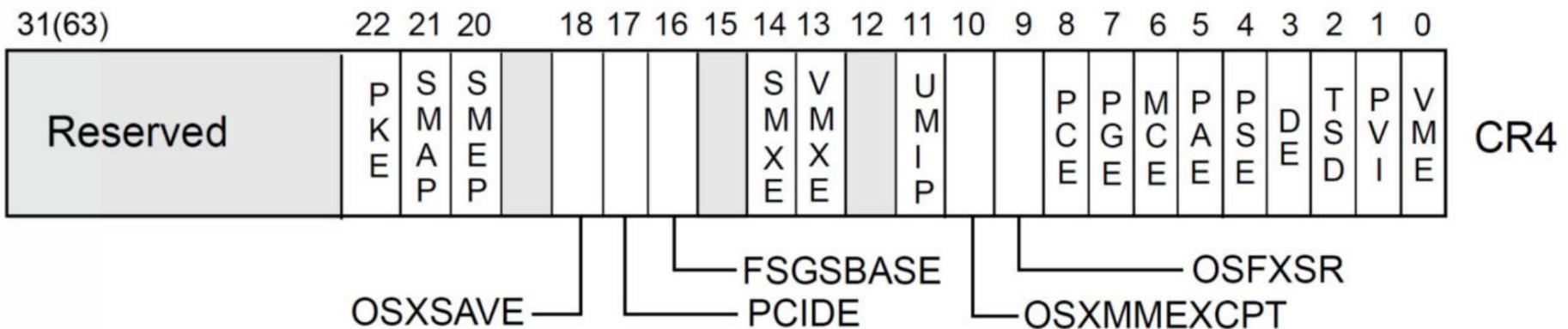




x86 分页支持寄存器 (2 of 2)

寄存器 %cr4 包含位 PSE (页面大小扩展)

位置 4。如果该位设置为 1，则会激活对 4MB 页面的支持，而不是默认的 4KB 页面大小。





xv6 和页表硬件。

❖ xv6 使用 32 位虚拟地址,虚拟地址空间为 4GB。 ❖ xv6 使用分页来管理其内存分配。

❖ xv6 使用 4KB 的页面大小,以及两级页表结构。

❖ 在“运行”一个进程之前,调度程序存储基址

控制寄存器 (%cr3) 中进程页表的地址。

[这](#)是在xv6调度器中通过调用switchuvm()来完成的,如下所示,switchuvm()在vm.c中定义

❖ vm.c 中的函数 walkpgdir() 接受一个虚拟地址并返回

它在页表中对应的PTE。

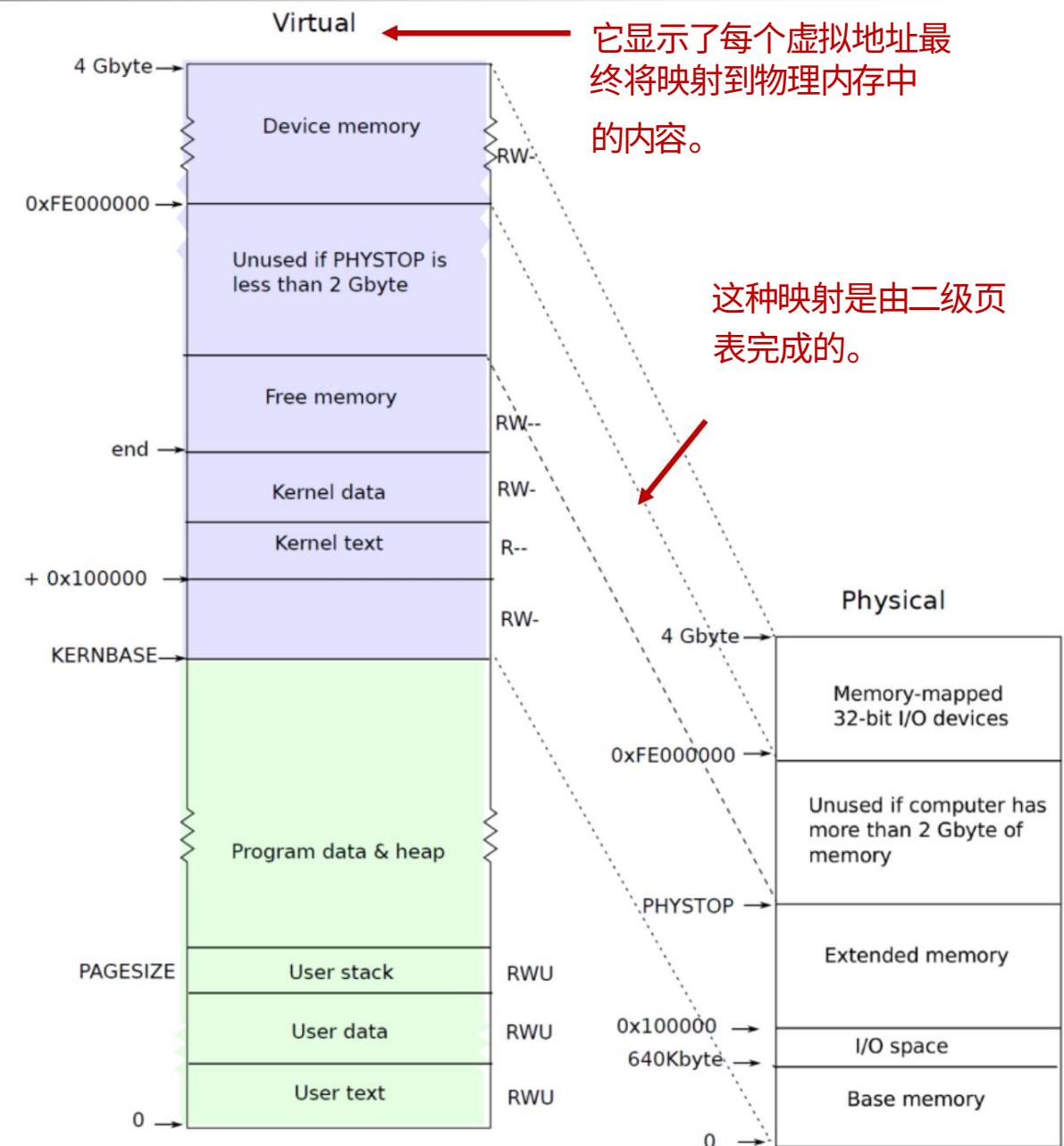
```
switchuvm(p); // 切换到进程的页表 (即地址空间)
p->状态=运行;
swtch(&(cpu->调度器), p->上下文); // 进程p恢复运行
开关kvm(); // 切换回内核页表
```



xv6 进程地址空间 (4 中的 1)

在xv6中,每个进程都有自己的页表,当xv6在进程之间切换时,xv6告诉页表硬件切换页表。

如图所示,进程的用户内存从虚拟地址 0 开始,可以增长到KERNBASE = 0x80000000,允许进程寻址最多2 GB 的内存 (大多数进程不使用整个用户空间)。



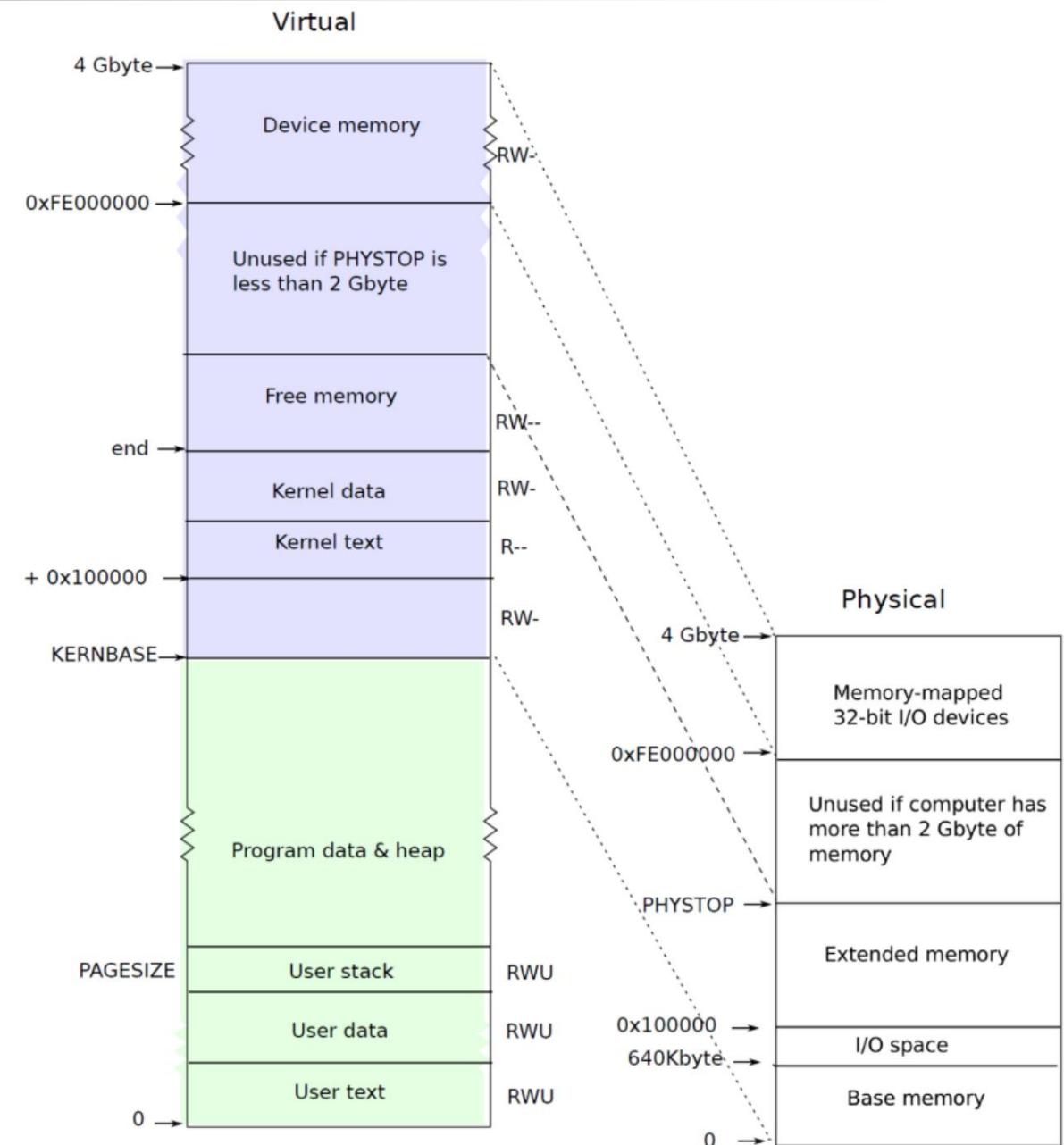


xv6 进程地址空间 (2 of 4)

[memlayout.h](#) 声明 xv6 内存布局的常量,以及在虚拟地址和物理地址之间转换的宏。

- ❖ 每个进程的页面表包含用户内存和整个内核的映射。

这很方便,因为在系统调用期间从用户代码切换到内核代码不需要页表切换。





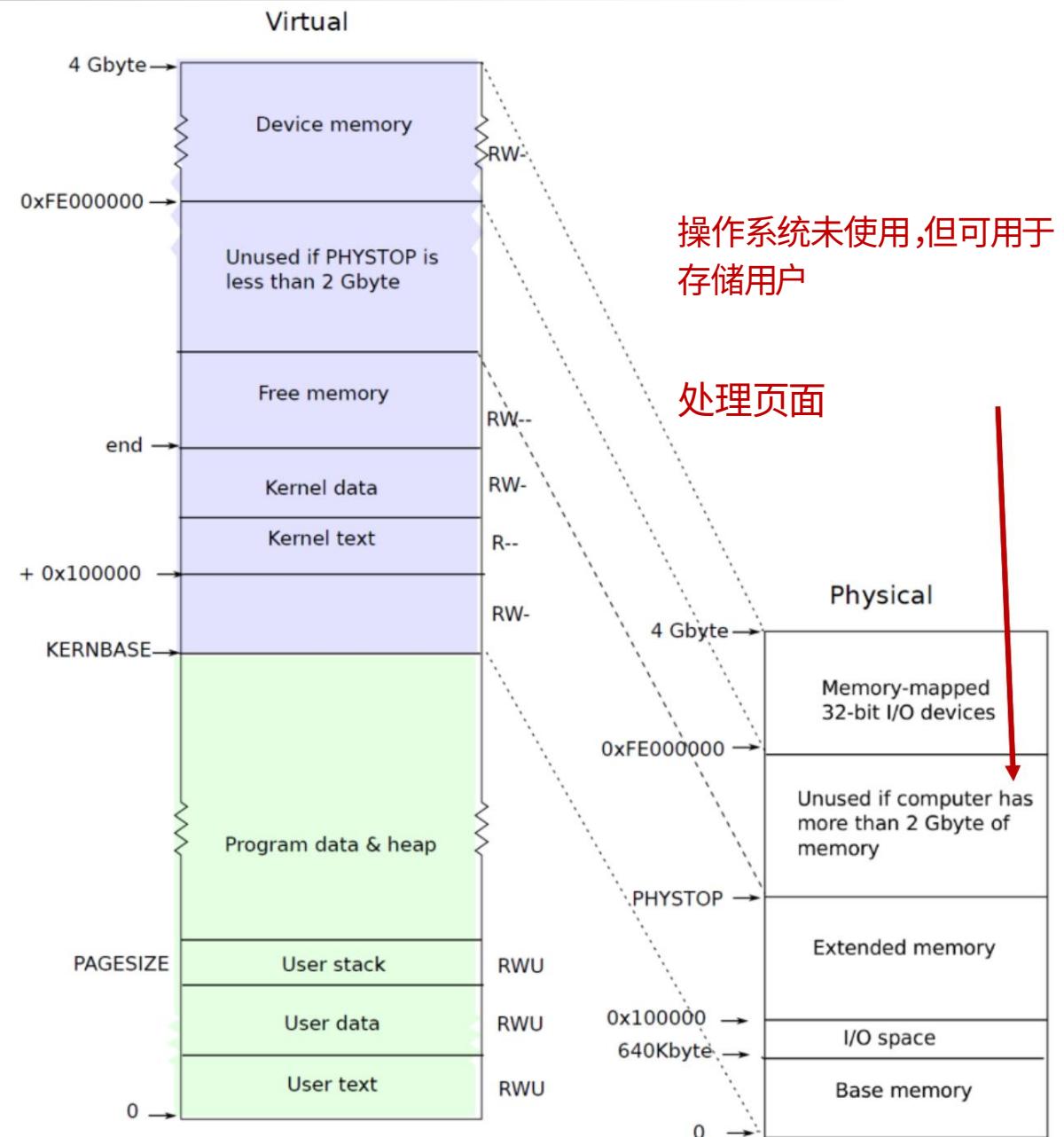
xv6 进程地址空间 (3 of 4)

❖ xv6 包括内核在每个进程的页表中运行所需的所有映射。 [页表中处理内核页的部分在所有内核页中都是相同的](#)

流程。

[vm.c”中的setupkvm\(\)和mappages\(\)函数将内核的物理内存地址空间 \(0到PHYSTOP\) 映射到新进程的虚拟地址空间 \(KERNBASE](#)

通过设置其页面目录来实现
KERNBASE+PHYSTOP)。





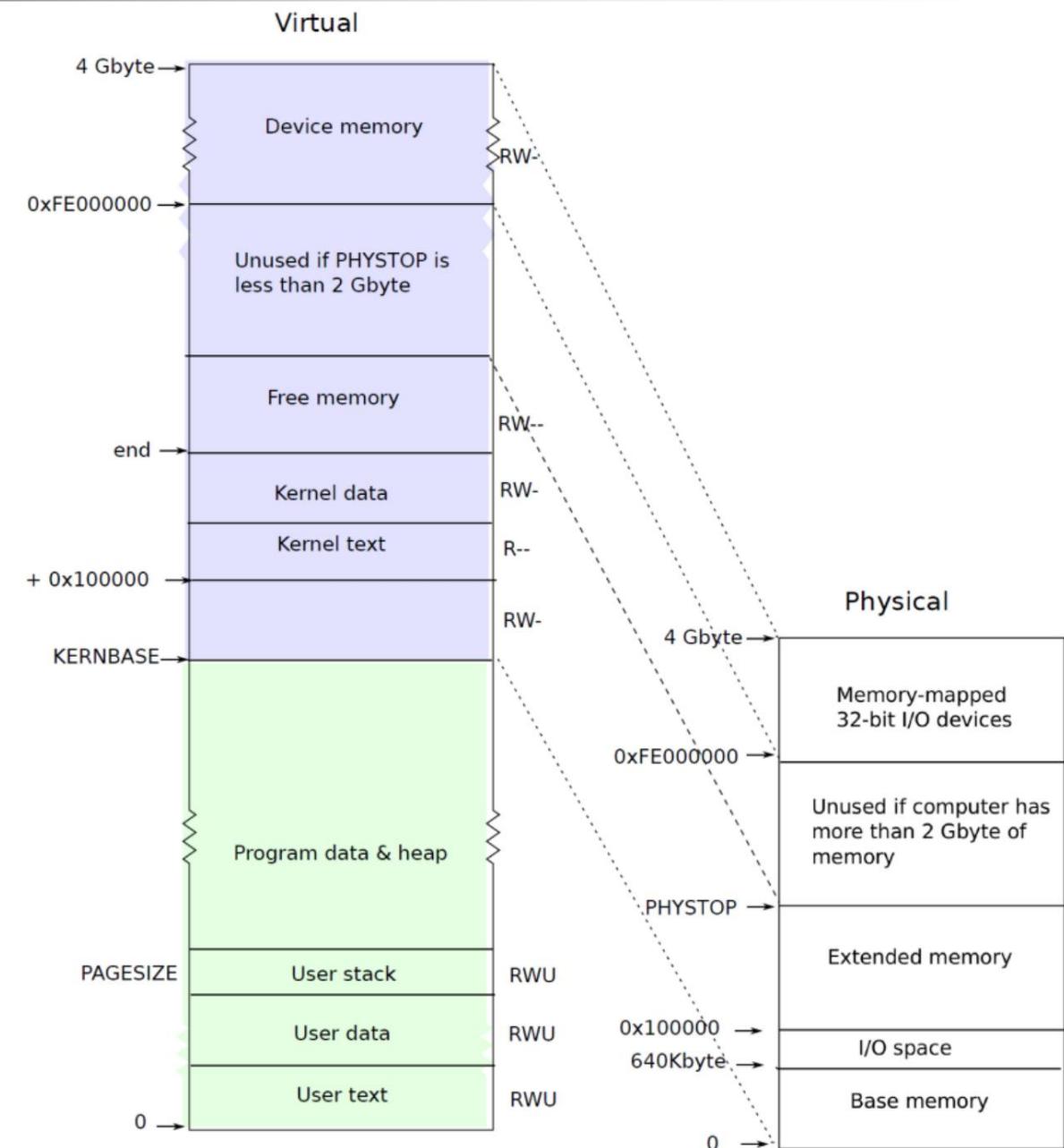
xv6 进程地址空间 (4 of 4)

❖ 用户布局

xv6 中执行进程的内存显示为绿色。

❖ xv6 只分配一个 每个进程的 PAGESIZE 堆栈 (即 4 KB)。

它在堆栈上方分配堆,当进程调用
sbrk 时,堆可以增长到
KERNBASE (2 GB) (稍后解
释)。





内核页表

每个进程有一个页表,加上一个 CPU 未运行任何进程时使用的页表 (kpgdir)。❖ kpgdir 是内核页表,在运行scheduler()时使用。调度程序通过运行 vm.c 中定义的 switchkvm()切换回它

正如所解释的,内核在系统调用和中断期间使用当前进程的页表;页保护位阻止用户代码使用内核的映射。

```
switchuvm(p); // 切换到进程的页表 (即地址空间)
p->状态=运行;
swtch(&(cpu->调度器), p->上下文); // 进程p恢复运行
开关kvm(); // 切换回内核页表
```



xv6 空闲内存的维护

启动后, RAM 包含 xv6 操作系统代码/数据和空闲页面。

- ❖ xv6 将所有空闲页面收集到一个空闲链表中。

- ❖ 所示函数 kalloc()

分配此列表头部的页面 (如果可用)。

空闲页最终分配给用户进程的代码/数据/堆栈/堆以及这些进程的页表。

空闲列表是一个链表, 内核在其中维护指向列表中第一页的指针。

结构运行{结构运行*

下一个;};

struct

{ struct spinlock 锁; int use_lock;
结构运行*freelist; }

公里管理;

```
// 分配一页 4096 字节的物理内存。
// 返回内核可以使用的指针。
// 如果内存无法分配则返回0。 char* kalloc
(void) { struct run * r; if
(kmem.use_lock )
acquire(&kmem.lock); r =
kmem.freelist; if ( r
kmem.freelist = r->next; if
(kmem.use_lock )释放(&kmem.lock);返回 (字符
*) r;
```

}



代码:执行

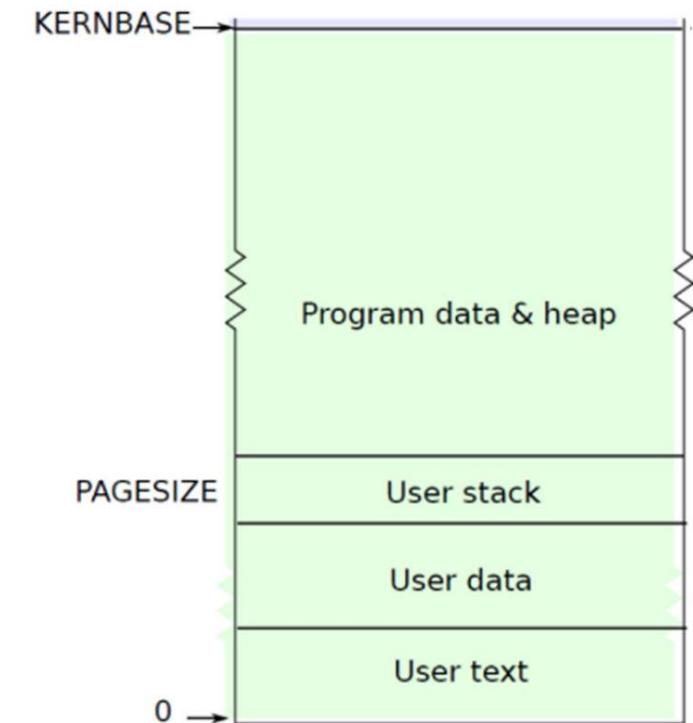
[exec](#) 是创建地址空间的用户部分的系统调用（在 exec.c 中）。

它通过首先读取文件系统中存储的文件的 ELF 标头来初始化该部分。

- ❖ ELF 代表可执行和可链接

格式，它是标准的二进制文件格式，xv6 在 elf.h 中定义它

第一步是快速检查该文件是否可能包含 ELF 二进制文件，该二进制文件必须以四字节“幻数”0x7F、‘E’、‘L’、‘F’开头。如果 ELF 标头具有正确的幻数，则 exec 会假定二进制文件格式正确。





在 exec 中构建页表

进程的页表是在 exec 中通过三个主要步骤构建的，如图所示。

♦ 我们已经谈到了步骤1。

② Load the program image and map it to the page table

③ Initialize the runtime stack and heap and map them into the page table

```

int
exec(char *path, char **argv)
{
    char *s, *last;
    int i, off;
    uint argc, sz, sp, ustack[3+MAXARG+1];
    struct elfhdr elf;
    struct inode *ip;
    struct proghdr ph;
    pde_t *pgdir, *oldpgdir;

if((pgdir = setupkvm()) == 0)
    goto bad;

// Load program into memory.
sz = 0;
for(i=0, off=elf.phoff; i<elf.phnum; i++, off+=sizeof(ph)){
    if(readi(ip, (char*)&ph, off, sizeof(ph)) != sizeof(ph))
        goto bad;
    if(ph.type != ELF_PROG_LOAD)
        continue;
    if(ph.memsz < ph.filesz)
        goto bad;
    if((sz = allocuvm(pgdir, sz, ph.vaddr + ph.memsz)) == 0)
        goto bad;
    if(loaduvm(pgdir, (char*)ph.vaddr, ip, ph.off, ph.filesz) < 0)
        goto bad;
}

// Allocate two pages at the next page boundary.
// Make the first inaccessible. Use the second as the user stack.
sz = PGROUNDUP(sz);
if((sz = allocuvm(pgdir, sz, sz + 2*PGSIZE)) == 0)
    goto bad;
clearpteu(pgdir, (char*)(sz - 2*PGSIZE));
sp = sz;

```

① map kernel to the page table



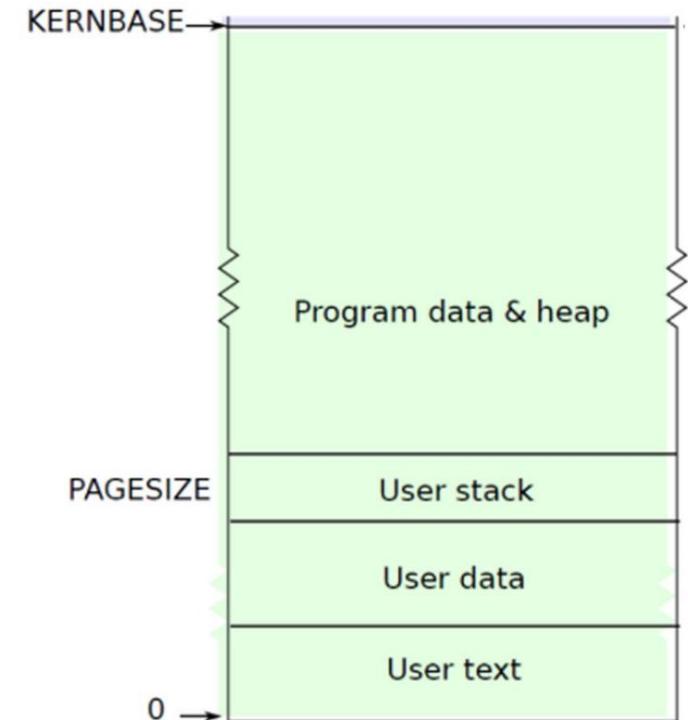
映射用户程序文本和数据

建立进程页表的第二步是将用户进程数据和文本（即其代码）段映射到进程页表。

这是通过逐段检查程序映像（ELF 文件）并加载标记为“`ELF_PROG_LOAD`”的段来完成的。

❖ 函数 `allocuvm()` 用于分配内存并将其映射到页表。

函数 `loaduvm()` 用于将段加载到分配的内存中。





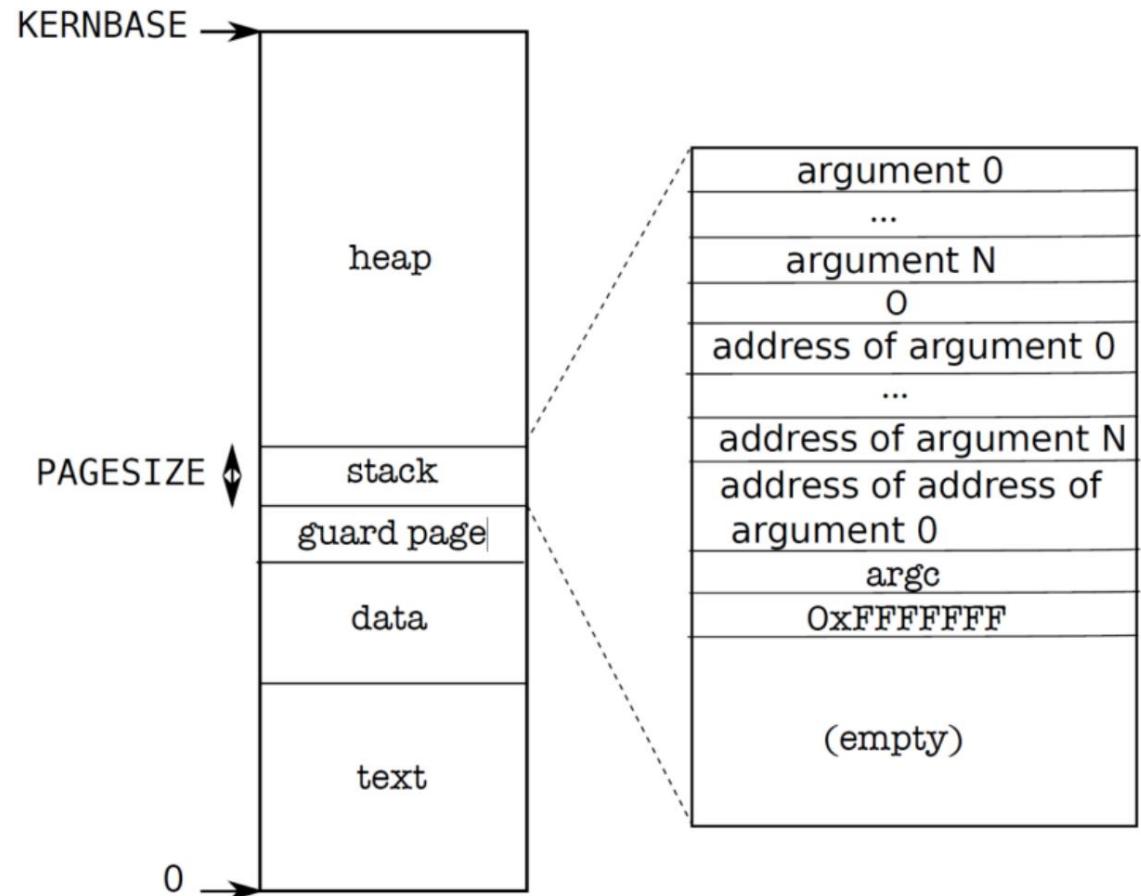
映射用户堆栈和堆 (1 of 2)

- 建立进程页表的第三步是将用户栈和堆映射到表中。

堆栈是单页,并用

由 exec 创建的初始内容。

- 包含以下内容的字符串
命令行参数、指向它们的指针数组和
argc 位于堆栈的最顶部。

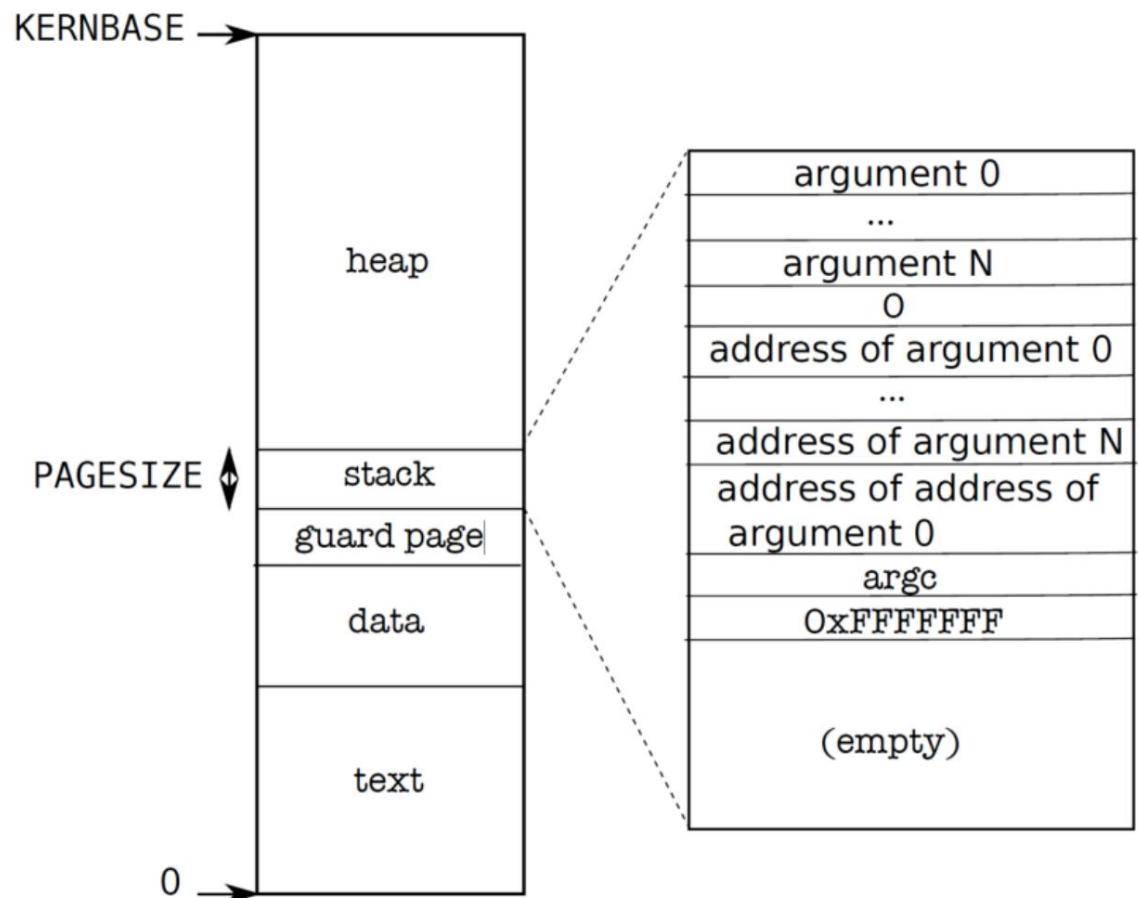




映射用户堆栈和堆 (2 of 2)

为了保护从堆栈页增长的堆栈,xv6 在堆栈的正下方放置了一个保护页。未映射保护页。如果堆栈超出堆栈页,硬件将生成异常。

返回地址“0xffffffff”是假的，没有分配。因此，在xv6的程序中，在main函数中使用“return”将会导致错误。因此，请使用exit()代替。





检查进程堆栈地址

在 exec.c 中添加所示的两个 “cprintf”语句,以检查 exec 执行的任何进程的堆栈起始地址和堆栈的最大限制。

当 exec 启动 “init”和 “shell”进程时,启动 xv6 时观察这些语句的输出。

```
56 // Allocate two pages at the next page boundary.  
57 // Make the first inaccessible. Use the second as the user stack.  
58 sz = PGROUNDUP(sz);  
59 if((sz = allocuvm(pgdир, sz, sz + 2*PGSIZE)) == 0)  
60     goto bad;  
61 clearpteу(pgdир, (char*)(sz - 2*PGSIZE));  
62 sp = sz;  
63 cprintf("pid:%d, base addr of the stack:0x%x\n", proc->pid, sp-1);  
64 cprintf("pid:%d, end addr of the stack:0x%x\n", proc->pid, sp+PGSIZE);  
65  
66 // Push argument strings, prepare rest of stack in ustack.  
67 for(argc = 0; argv[argc]; argc++) {  
68     if(argc >= MAXARG)  
69         goto bad;  
70     sp = (sp - (strlen(argv[argc]) + 1)) & ~3;  
71     if(copyout(pgdир, sp, argv[argc], strlen(argv[argc]) + 1) < 0)  
72         goto bad;  
73     ustack[3+argc] = sp;  
74 }  
75 ustack[3+argc] = 0;
```



代码:sbrk

❖ sbrk 是进程收缩或增长其堆内存空间的系统调用。系统调用

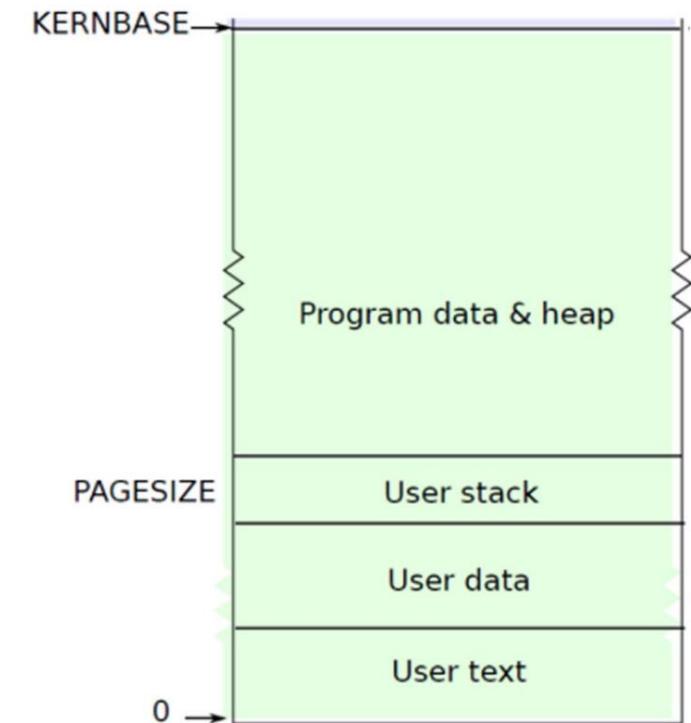
由proc.c中的growproc(int n)函数实现

❖ 如果 n 为正数,则一个或多个物理

页在进程地址空间的顶部分配和映射。 ❖ 如果 n 为

负数,growproc 将从进程的地址空间取消映射

一个或多个页面并释放相应的物理页面





检查进程堆大小 (1 of 2)

在 sysproc.c 中添加所示的 “cprintf”语句来检查

通过从进程调用 malloc() ,进程堆分配的大小会发生变化。当 exec 执行时启动 xv6 时观察这些语句的输出
贝壳。

```
int sys_sbrk (void) { int addr;  
    整数n;
```

```
if (argint (0, &n) < 0) 返回- 1; addr = myproc()-  
>sz; if (growproc (n) < 0) 返回-  
1; cprintf( pid:%d 将堆从  
%x 增长到 %x,  
大小为 %d 字节。\n , proc -->pid,addr, proc -->sz, proc --  
>sz addr); 返回地址;
```

```
}
```



检查进程堆大小 (2 of 2)

❖ 要测试堆大小如何变化,需要添加以下内容

test_heap.c 程序作为新的用户命令 (与我们在这些幻灯片开头添加 hello.c 的方式相同)。运行命令并观察堆大小的消息。 将 malloc 中的值 5 替换为 32768 并观察差异。

```
#include "types.h"
#include "stat.h"
#include "user.h"

int main (int argc, char *argv[]) { int i = 10; while ( i >
0){ char * addr =
malloc ( 5 *
sizeof (char)); printf ( 1,  返回地址:%p\n ,addr);我 - ;
}

出口 () ;
}
```



阅读材料和资源列表

- ❖ xv6 手册：<https://pdos.csail.mit.edu/6.S081/2020/xv6/book-riscv-rev1.pdf>
 - ❖ xv6 来源手册：<https://pdos.csail.mit.edu/6.828/2018/xv6/xv6-rev11.pdf>
 - ❖ xv6 源代码文件：<https://github.com/mit-pdos/xv6-public>
- 英特尔® IA-32 架构软件开发人员手册：
<https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html>