

AVX512OS—基于 RISC-V 的操作系统

张俊逸，刘弈成，胡宇飞

2023 年 6 月 6 日

1 介绍

AVX512OS 操作系统是运行在 risc-v 架构上的简单的操作系统，其基本设计思想和框架来自于 MIT 的操作系统实验项目。本文档主要介绍 AVX512OS 操作系统的设计。

1.1 项目背景及意义

在华中科技大学的计算机系统教育中，我们注意到理论授课和实践操作的平衡尚待加强。操作系统作为计算机科学三大基石之一，是理解程序运行和硬件资源管理的关键。然而，由于其管理机制的复杂性，操作系统在理论教学和实践应用中都构成了一定的挑战。

华中科技大学的本科学生在自己关注的学术领域显示出了积极的探索精神，他们组建了一些研究小组，致力于各自感兴趣的领域的深入研究。这种积极的学习态度在培养学生的研究能力和解决实际问题的能力方面发挥了积极的作用。

然而，我们意识到，在计算机系统的专业培养中，华中科技大学还需要在以下三个方面进一步提升：首先，强化理论教学，以丰富学生的知识库并加深他们对专业理论的理解；其次，设计独特的实验，通过实践让学生更好地理解理论，增强他们的实践技能；最后，鼓励在相关领域的导师引领下进行前沿探索，让学生有机会深入研究，并从中获取新的知识和技能。

我们认为，设计独特的实验是目前需要优先解决的问题。为此，我们计划在实际硬件环境中进行操作系统的开发，这样就能为课程实验设计提供足够的自由度，摆脱现有的框架约束。我们相信，这不仅能够增加课程任务设计的可能性，也会为学生提供更多的空间去理解和探索新的知识。

但这不仅仅是解决现有问题的手段，我们更希望这种方式能够作为一个新的教学模式，通过实践和持续改进，发展成为一个可持续的教学模式。因此，我们计划通过 Github 项目组、课程实验以及计算机系统小组等形式来推广和实践这种模式。

我们深信，这对于人才培养至关重要，这不仅可以提高学生的理论知识和实践技能，还可以激发他们的创新思维，培养他们独立解决问题的能力。最终，我们希望这些努力能够为华中科技大学的计算机科学教育领域带来新的活力，推动其不断向前发展。

总的来说，我们希望通过这些行动，能够为华中科技大学的计算机科学教育领域带来新的活力，推动创新的进步，为未来的学生和教师提供更好的教学环境。

1.2 国内外研究状况

操作系统内核是操作系统的核心部分，它负责处理最基础的硬件与软件之间的交互，比如内存管理、进程调度和文件系统等。关于国内外操作系统内核的研究，可以从几个不同的角度进行介绍，包括商业系统、开源系统，以及相关的研究领域和趋势。

首先，从商业系统的角度来看，有两个主要的玩家：微软和苹果。

微软的 Windows 操作系统内核是 NT 内核，这是一种混合内核，结合了微内核和宏内核的特点。微软投入了大量的资源进行 NT 内核的研究和开发，以提供高效、安全和稳定的用户体验。苹果的 macOS 操作系统使用的是 XNU 内核，这是一种混合内核，包含了 Mach 微内核和 BSD 宏内核的部分。苹果的研究重点包括提高性能，增强安全性，以及支持新的硬件和技术。然后，从开源系统的角度来看，Linux 是最重要的项目之一。

Linux 内核是一个开源的宏内核，全球有数万名开发者参与到 Linux 内核的研究和开发中。Linux 内核的设计目标是提供一个通用、稳定和高效的操作系统内核。目前，Linux 内核已经被广

泛应用于服务器、超级计算机、嵌入式设备，甚至个人桌面环境。在国内，中科院软件研究所、哈工大等多家研究机构 and 高校也在进行操作系统内核相关的工作。

中科院软件研究所的可信鸿蒙操作系统，以 Linux 内核为基础，主要目标是提供一个具有高度可信性的操作系统，适应未来的网络安全环境。哈工大的麒麟操作系统，以 Linux 内核为基础，主要目标是提供一个具有高度国产化的操作系统，满足国内的自主可控需求。此外，未来的操作系统内核研究趋势也是一个值得关注的领域。

微内核研究：虽然目前的主流操作系统内核多为宏内核或混合内核，但微内核的研究仍在继续。微内核有助于提高系统的可靠性和安全性，因为它可以将大部分功能移到用户空间，减少内核空间的复杂性。目前，例如 Google 的 Fuchsia 操作系统就是使用的 Zircon 微内核。安全性研究：随着网络安全问题的日益突出，操作系统内核的安全性研究也越来越重要。例如，内核自我保护、内核隔离和内核加固等技术都是当前的研究重点。新硬件支持：随着硬件技术的进步，例如多核处理器、非易失性内存和量子计算机等，如何更好地利用这些新硬件，也是操作系统内核研究的重要方向。总的来说，操作系统内核的研究是一个既广泛又深入的领域，包括了商业系统、开源系统，以及各种各样的研究领域和趋势。

1.3 项目的主要工作

avx512OS 致力于开发一个基于 RISC-V64 的多核操作系统,并且能够运行在实体裸机 Visionfive2 平台上。我们使用了 MIT 教学操作系统 xv6 代码框架继续开发,并在其基础上逐步完善了 AVX512OS 系统功能,更加全方位地支持更多系统调用,改进了对硬件的抽象,提升了系统鲁棒性,并大幅增强了操作系统的性能,最终实现了 avx512OS 的开发。

首先,我们对硬件抽象化接口 (SBI) 进行了大量的工作。我们开发了一套全新的中断处理机制,它能够适应 Visionfive2 平台的特性,更有效地处理和分发中断。我们还开发了一套无效指令软处理机制,这样即使硬件支持有限,我们也可以在软件层面扩展其功能,这大大提高了我们操作系统的兼容性和扩展性。

然后,我们也对用户抽象化接口 (ABI) 进行了改进。我们增加了许多新的系统调用,使得用户程序可以更方便地使用操作系统提供的服务。这些新的系统调用包括了文件操作、进程管理、内存管理等多个领域,使得我们的操作系统可以支持更复杂的用户程序。

在鲁棒性方面,我们对内核进行了一系列的优化和改进。例如,我们改进了内核的错误处理机制,使其能够更好地处理硬件错误和软件错误。我们还增强了内核的内存保护,防止了内核内存被用户程序错误地访问。

最后,我们还进行了一系列的性能优化。我们优化了内核的调度算法,使得系统可以更公平、更高效地分配 CPU 时间。我们还改进了内核的内存管理,使得系统可以更快地分配和回收内存。

经过这一系列的改进和优化,我们最终成功地实现了 avx512OS 的开发。avx512OS 展示了极高的兼容性和扩展性。同时,由于我们在 SBI 和 ABI 上的改进,使得 avx512OS 在功能性、鲁棒性和性能上都达到了很高的水平。

2 设计目标

avx512OS 的实现在操作系统大赛的初赛阶段是为了实现 33 个系统调用,完成操作系统的基本功能。在实现测试的系统调用过程中,附带实现了一些其他的系统调用,以方便整个操作系统的实现。

1	#define SYS_fork	1
2	#define SYS_exit	93
3	#define SYS_wait	3
4	#define SYS_pipe	4
5	#define SYS_read	63
6	#define SYS_kill	6
7	#define SYS_exec	7
8	#define SYS_execve	221
9	#define SYS_fstat	80
10	#define SYS_chdir	49
11	#define SYS_dup	23
12	#define SYS_dup3	24
13	#define SYS_getpid	172
14	#define SYS_sbrk	12
15	#define SYS_brk	214
16	#define SYS_sleep	13
17	#define SYS_uptime	14
18	#define SYS_open	15
19	#define SYS_write	64
20	#define SYS_remove	25
21	#define SYS_trace	18
22	#define SYS_sysinfo	19
23	#define SYS_mkdir	1030
24	#define SYS_close	57
25	#define SYS_test_proc	22
26	#define SYS_dev	10
27	#define SYS_readdir	21
28	#define SYS_getcwd	17
29	#define SYS_rename	26
30	#define SYS_getppid	173
31	#define SYS_mkdirat	34
32	#define SYS_nanosleep	101
33	#define SYS_clone	220
34	#define SYS_pipe2	59
35	#define SYS_wait4	260
36	#define SYS_getdents64	61
37	#define SYS_openat	56
38	#define SYS_gettimeofday	169
39	#define SYS_mmap	222
40	#define SYS_munmap	215
41	#define SYS_sched_yield	124

```

42 #define SYS_uname      160
43 #define SYS_unlinkat   35
44 #define SYS_mount      40
45 #define SYS_umount     39
46 #define SYS_times      153

```

为了充分支持评测系统，我们需要实现以下功能：支持非在线库的项目依赖项，避免外部设备的自动启动依赖，并且需要实现自动运行评测程序，包括序列化和并行化的执行等功能。

3 系统设计与实现

3.1 进程管理

进程管理模块主要功能有：进程初始化、进程载入和解析、进程切换、进程状态构建模块。下面分别介绍。

3.1.1 进程控制块

进程是操作系统提供的一层抽象，它实际上是运行中的程序，为了方便管理进程，avx512OS使用进程控制块来进行操作系统进程的管理。进程控制块的结构如下所示。进程控制块中存储了进程的状态，这也是操作系统对进程进行管理的最基本单元。其中具体的状态我们将在后面一一进行解析。

```

1  struct proc {
2      struct spinlock lock;
3
4      // p->lock must be held when using these:
5      enum procstate state;           // Process state
6      struct proc *parent;           // Parent process
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     // these are private to the process, so p->lock need not be
        held.
13     uint64 kstack;                   // Virtual address of kernel
        stack
14     uint64 sz;                       // Size of process memory (bytes
        )
15     pagetable_t pagetable;           // User page table
16     pagetable_t kpagetable;          // Kernel page table

```

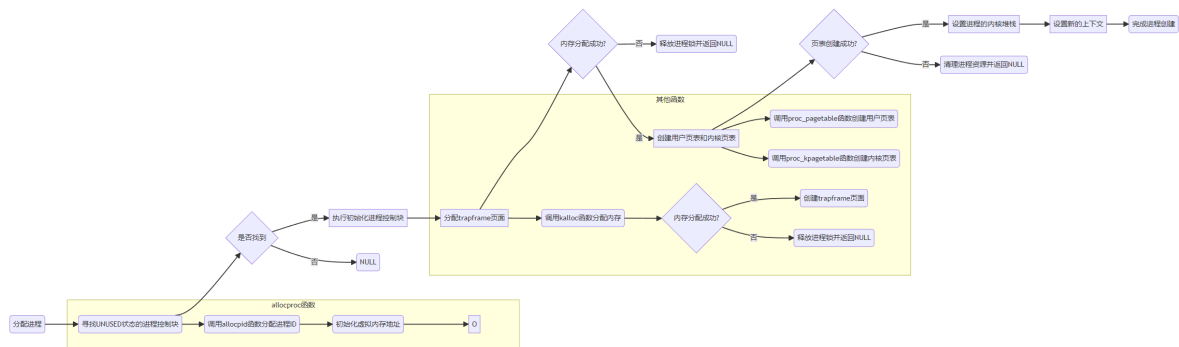


图 1: 分配进程流程图

```

17  struct trapframe *trapframe; // data page for trampoline.S
18  struct context context;      // swtch() here to run process
19  struct file *ofile[NOFILE]; // Open files
20  struct dirent *cwd;          // Current directory
21  char name[16];               // Process name (debugging)
22  int tmask;                   // trace mask
23  struct vma *vma;
24  int ktime;
25  int utime;
26  };

```

3.1.2 进程状态

进程的状态定义如下

```

1  enum procstate { UNUSED, SLEEPING, RUNNABLE, RUNNING,
    ZOMBIE };

```

UNUSED 状态表明进程控制块没有对应任何进程。

SLEEPING 表明进程正因为某种原因而没有运行。

RUNNABLE 表明进程正在等待调度器投入运行。

RUNNING 表明进程正在运行。

ZOMBIE 表明进程已经被杀死，但是资源还没有被回收。

3.1.3 分配进程

在 avx512OS 中，分配新进程的过程是通过 allocproc 函数实现的。该函数主要执行以下步骤：

寻找一个状态为 UNUSED 的进程控制块：系统首先扫描预设的进程控制块数组，寻找一个状态为 UNUSED 的进程控制块。如果找到，便跳转到 found 标签进行下一步操作；如果未找到，则返回 NULL，表示当前没有可用的进程控制块，无法创建新的进程。

初始化进程控制块：一旦找到一个未使用的进程控制块，函数将为该进程控制块分配一个新的进程 ID（通过调用 allocpid 函数），初始化其虚拟内存地址（vma）为 NULL，并设置其内核时间（ktime）和用户时间（utime）为 1。

分配 trapframe 页面：该函数会为新进程的 trapframe 分配一块内存（通过调用 kalloc 函数）。如果内存分配失败，函数将释放进程锁并返回 NULL。

创建用户页表和内核页表：在成功分配 trapframe 页面后，函数将创建一个空的用户页表（通过调用 proc_pagetable 函数）和一个内核页表（通过调用 proc_kpagetable 函数）。如果任一页表创建失败，函数会清理进程资源并返回 NULL。

设置进程的内核堆栈（kstack）：在分配页表成功后，函数将设置进程的内核堆栈地址。这个地址是预设的虚拟地址。

设置新的上下文：该函数会初始化进程的上下文（context），并设置返回地址寄存器（ra）为 forkret 函数的地址，堆栈指针（sp）设置为内核堆栈的顶部。这样设置后，新的进程将从 forkret 函数开始执行。

通过这个过程，allocproc 函数完成了一个新进程的创建。每个新创建的进程在完成这个步骤后，就被分配了一个唯一的进程 ID，有了自己的页表和堆栈，准备好了开始执行。

3.1.4 进程初始化

在 avx512OS 中，除了 init 进程，所有的进程都是通过 fork 系统调用首先复制，然后使用 exec 系统调用加载新程序来执行的。

对于 init 进程，初始化过程基本如下：

进程分配：avx512OS 预设了固定的进程数量，因此，每次创建新进程时，系统需要从进程控制块（Process Control Block, PCB）数组中寻找一个空闲的块。

页表映射：avx512OS 采用了 RISC-V 的 sv39 虚拟内存模型。在此步骤中，操作系统会为程序的各个段（如代码段，数据段等）进行页表映射。为了方便处理，我们预编译了一段二进制代码，这段代码的任务就是执行 init 程序。init 程序将使用 fork 系统调用创建一个新的进程，然后通过 exec 系统调用执行 shell 程序。而原始的 init 进程则将进入无限的进程调度循环。

设置进程状态：在完成以上步骤后，操作系统将进程状态设置为可运行。

代码如下：

```
1 static struct proc*
2 allocproc(void)
3 {
4     struct proc *p;
5
6     for(p = proc; p < &proc[NPROC]; p++) {
7         acquire(&p->lock);
8         if(p->state == UNUSED) {
9             goto found;
```

```

10     } else {
11         release(&p->lock);
12     }
13 }
14 return NULL;
15
16 found:
17 p->pid = allocpid();
18 p->vma = NULL;
19 p->ktime = 1;
20 p->utime = 1;
21 // Allocate a trapframe page.
22 if((p->trapframe = (struct trapframe *)kalloc()) == NULL){
23     release(&p->lock);
24     return NULL;
25 }
26
27 // An empty user page table.
28 // And an identical kernel page table for this proc.
29 if ((p->pagetable = proc_pagetable(p)) == NULL ||
30     (p->kpagetable = proc_kpagetable()) == NULL) {
31     freeproc(p);
32     release(&p->lock);
33     return NULL;
34 }
35
36 p->kstack = VKSTACK;
37
38 // Set up new context to start executing at forkret,
39 // which returns to user space.
40 memset(&p->context, 0, sizeof(p->context));
41 p->context.ra = (uint64)forkret;
42 p->context.sp = p->kstack + PGSIZE;
43
44 return p;
45 }

```

对于其他进程，其初始化过程大致如下：1. 使用 fork 系统调用：fork 的主要作用是创建一个与父进程完全相同的子进程。这个过程包括为新进程创建一个进程控制块，复制父进程的页表，并在进程控制块中记录父子进程关系，这将在 PCB 的 parent 字段中完成。

2. 执行 exec 系统调用：exec 的主要功能是解析 ELF 文件，并为进程重新映射虚拟内存。通过这种方式，新的程序就可以在原来进程的环境中执行，而不是仅仅复制父进程的行为。整个过

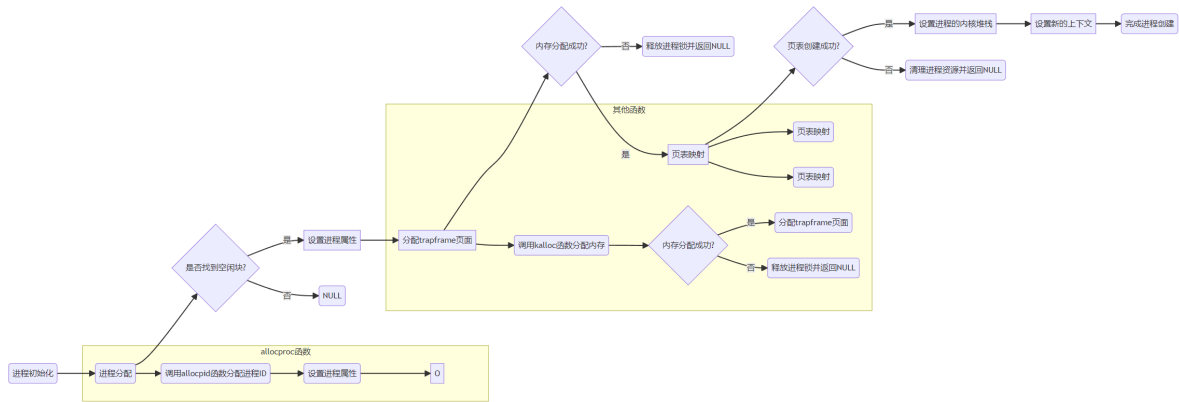


图 2: 进程初始化

程保证了进程的正确创建和程序的正确加载，保障了 avx512OS 的多进程环境能够正常运作。

3.1.5 进程调度

目前，avx512OS 操作系统的进程调度采用时间片均分的策略，因此，操作系统将会在时钟中断时，打断当前进程的运行，并调度下一进程进行执行。所有进程的优先级不分，统一放在一个队列进行调度。下面我们将介绍进程的切换过程，但在此之前，我们需要先介绍 avx512OS 对 cpu 的一层抽象。

在 avx512OS 中，cpu 是由下面的结构体进行管理的。

```

1 struct cpu {
2     struct proc *proc;           // The process running on this
        cpu, or null.
3     struct context context;      // swtch() here to enter
        scheduler().
4     int noff;                    // Depth of push_off() nesting.
5     int intena;                  // Were interrupts enabled before
        push_off()?
6 };

```

其中的重点为 proc 和 context 字段。proc 字段指向的是在当前 cpu 上执行的进程，而 context 字段的作用正是进程切换。context 的定义如下：

```

1 struct context {
2     uint64 ra;
3     uint64 sp;
4     uint64 s0;
5     uint64 s1;

```

<https://www.overleaf.com/project/64703095929650>

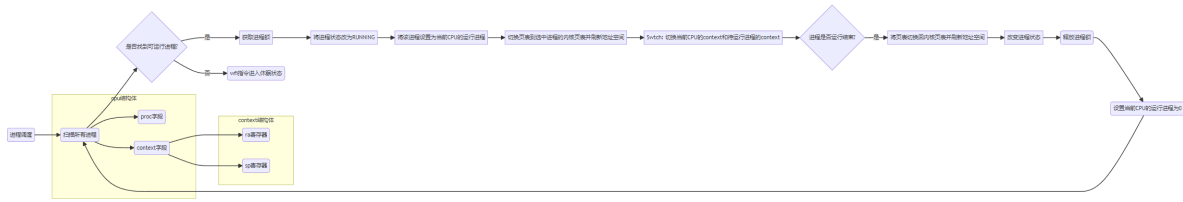


图 3: 进程调度

```

ecbc5428ad
6  uint64 s2;
7  uint64 s3;
8  uint64 s4;
9  uint64 s5;
10 uint64 s6;
11 uint64 s7;
12 uint64 s8;
13 uint64 s9;
14 uint64 s10;
15 uint64 s11;
16 };

```

它用于保存当前进程的上下文,以便于进程下一次投入运行。avx512OS 的进程调度是通过 scheduler 函数实现的,当系统需要进行进程切换时,它会扫描所有的进程,寻找状态为 RUNNABLE 的进程。

首先,该函数从进程列表中选取一个状态为 RUNNABLE 的进程。获取进程后,会获得该进程的锁以防止其他进程同时修改进程的状态。

然后,这个被选中的 RUNNABLE 进程状态被改为 RUNNING,并且把它设置为当前 CPU 的正在运行的进程。这时,该函数会切换页表到选中进程的内核页表,并刷新地址空间。

接着,函数会执行 swtch 操作,将当前 CPU 的 context 和待运行进程的 context 进行切换。这个切换操作涉及到改变处理器的 ra 寄存器和 sp 寄存器。这就意味着处理器会跳转到新进程指定的地址去执行,即待运行进程的 ra 寄存器的地址。

当进程运行结束后,处理器会跳回 scheduler 函数,并且将页表切换回内核页表,同时刷新地址空间。这个进程此时应该已经改变了自己的状态。

最后,当前 CPU 的正在运行的进程将被设置为 0,表示 CPU 暂时没有正在运行的进程。然后释放进程的锁,进行下一轮的进程选择。

如果在一轮循环中没有找到可运行的进程,那么将执行 wfi 指令使处理器进入休眠状态,等待中断唤醒。

通过这样的过程,avx512OS 实现了进程调度,保证了多个进程可以公平、有序地在 CPU 上运行。

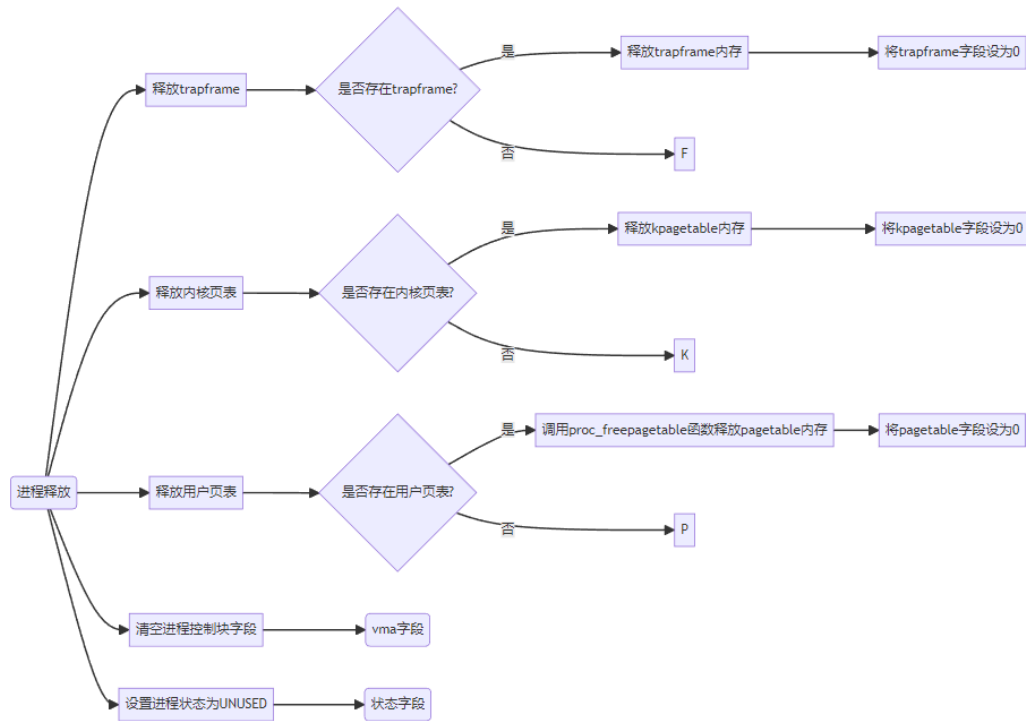


图 4: 进程释放

3.1.6 进程释放

在 avx512OS 中，当一个进程结束后或者在进程创建过程中发生错误时，需要释放进程所占用的资源，这个过程通过 `freeproc` 函数来完成。函数的主要步骤如下：

释放 trapframe: 首先,如果 trapframe 存在,则释放 trapframe 所占用的内存,然后将 trapframe 字段设为 0。

释放内核页表: 然后，如果内核页表（kpagetable）存在，则释放其所占用的内存，接着将 kpagetable 字段设为 0。

释放用户页表: 之后，如果用户页表（pagetable）存在，利用 `proc_freepagetable` 函数释放用户页表所占用的内存，同时将 pagetable 字段设为 0。

清空进程控制块的其他字段: 接下来，清空进程的虚拟内存地址（vma），将进程大小（sz）设为 0，进程 ID（pid）设为 0，父进程（parent）设为 0，清空进程的名称（name），清空等待的条件变量（chan），设置进程未被杀死（killed 设置为 0），设置进程的扩展状态为 0。

设置进程状态为 UNUSED: 最后，将进程状态设为 UNUSED，表示这个进程控制块可以被重新分配用于新的进程。通过这样的步骤，`freeproc` 函数实现了对进程资源的回收，这些资源包括内存资源（例如 trapframe、页表等）和进程控制块中的各种字段。回收完毕后，这个进程控制块就可以被系统重新利用，用来创建新的进程。

3.2 内存管理

3.2.1 物理内存管理

物理内存管理是操作系统的核心部分之一。avx512OS 采用简单且高效的物理内存分配和回收算法。在 avx512OS 中，物理内存管理是由 `kalloc.c` 模块负责的。这个模块包含了初始化物理内存、分配物理内存、回收物理内存等功能。

在 avx512OS 中，物理内存是以页为单位进行管理的，每页的大小为 4096 字节。在系统中，所有的空闲物理内存页通过一个单向链表进行组织，链表中的每个节点就代表一个物理内存页。

数据结构包括：`struct run`：这是一个简单的链表节点结构，它表示一个空闲的物理内存页，其中 `next` 指针指向下一个空闲的内存页。

`kmem`：这是一个全局变量，用于管理所有的空闲物理内存页。其中，`freelist` 指针指向空闲内存页链表的头部，`npage` 记录当前系统中空闲物理内存页的数量。`kmem` 中还包含一个名为 `lock` 的自旋锁，用于在多核环境下保护内存分配和回收操作的原子性。

函数包括：`kinit`：这个函数在系统启动时调用，用于初始化物理内存管理系统。它首先初始化 `kmem` 中的自旋锁，然后调用 `freerange` 函数，将内核结束地址 `kernel_end` 到物理内存上限 `PHYSTOP` 之间的所有物理内存页添加到空闲列表中。

`freerange`：这个函数用于将一段物理内存地址范围内的所有物理内存页释放，加入到空闲内存页链表中。它首先将起始地址向上取整到页的边界，然后从起始地址开始，每次增加一个页的大小（4096 字节），依次释放每个物理内存页。

`kfree`：这个函数用于释放一个物理内存页，它接收一个指向物理内存页的指针。函数首先检查传入的物理地址是否合法，然后将该物理内存页的内容清空，并将其添加到空闲内存页链表的头部。

`kalloc`：这个函数用于分配一个物理内存页。它首先尝试从空闲内存页链表的头部取出一个节点，如果链表为空，则返回 `NULL` 表示内存分配失败。如果成功取出一个节点，它将该节点从链表中移除，并清空该内存页的内容后返回。

`freemem_amount`：这个函数返回当前系统中的空闲物理内存量。由于内存页的大小是固定的，所以直接将空闲内存页的数量左移 12 位即可得到空闲内存的字节数。

总的来说，avx512OS 的物理内存管理采用了非常简洁和高效的设计，通过一个链表管理所有的空闲内存页，通过简单的链表操作实现物理内存页的分配和回收，满足了操作系统对物理内存管理的需求。

未来的改进方向：1. 改进锁机制：当前的实现中，所有对物理内存操作都需要获取全局的 `kmem` 锁，这在多核环境下可能会成为性能瓶颈。可以考虑引入更细粒度的锁机制，如每个 CPU 拥有自己的内存池和锁，或者使用无锁数据结构来提高并发性。

2. 改进内存分配策略：在当前的实现中，分配内存时总是从空闲列表的头部取出一个内存页，这实际上是一个简单的“先进先出”策略。虽然在页式内存管理中不会产生传统意义上的内存碎片，但是这种策略可能会导致物理内存的使用分布不均。可以考虑引入一些更复杂的内存分配策略，如伙伴系统（Buddy System）或者 slab 分配器。

3. 支持大页分配：当前的实现中，只支持分配和回收固定大小的内存页。如果需要分配大块的连续物理内存，会需要进行多次分配操作，这可能会影响性能。可以考虑支持大页（如 2M、1G）的分配，以提高内存访问的性能。

3.2.2 虚拟内存管理

页表是虚拟内存管理的重要工具，avx512OS 运行在 RISC-V 的 Sv39 配置下，这表示在 64 位的虚拟地址中，只有低 39 位被使用，而高 25 位则是闲置的。在 Sv39 模式下，RISC-V 的页表在逻辑层面上看起来就是一个包含了 2^{27} 个页表条目（Page Table Entries，简称 PTE）的大数组。每一个 PTE 都由 44 位的物理页码（Physical Page Number，简称 PPN）和一些标志位组成。在寻找虚拟地址所对应的 PTE 时，分页硬件会利用虚拟地址的前 27 位去索引页表。生成的物理地址为 56 位，其前 44 位来源于 PTE 中的 PPN，而后 12 位则直接来自原始的虚拟地址。页表在逻辑上看来就是一个简单的 PTE 数组（更详细的描述可以参考图 3.2）。页表为操作系统提供了控制虚拟地址到物理地址转换粒度的能力，这种粒度是以 4096 字节（ 2^{12} 字节）的对齐块，也就是“页”，为单位的。

在 RISC-V 的 Sv39 配置下，虚拟地址的前 25 位并未被应用于地址转换过程；在未来的 RISC-V 设计中，这些位可能会被利用来实现更多级别的地址转换。此外，物理地址也拥有进一步扩展的空间：PTE 的格式预留了 10 位，可以用于扩展物理地址的长度。这些参数的选择基于 RISC-V 设计者们技术预测。以 2^{39} 字节，即 512GB 的虚拟内存空间，应能满足 RISC-V 计算机上应用程序的运行需求。至于物理内存空间， 2^{56} 字节的范围在不久的将来，将足以支撑可能出现的 I/O 设备和 DRAM 芯片的内存需求。

地址转换是一个三步的过程。页表在物理内存中以一种三层的树状结构进行存储。这棵树的根节点是一个占据 4096 字节的页表页面，其中包含了 512 个 PTE，每个 PTE 都记录了树的下一层级页表页的物理地址。这些下一层的每个页表页中的每个 PTE 都引用了树的最后一层中 512 个 PTE。在根页表页面中，分页硬件利用 27 位中的前 9 位选择 PTE，然后在树的次一级页表页中用中间的 9 位选择 PTE，最后用剩下的 9 位选择最终的 PTE。在进行地址转换过程中，如果缺少必要的三个 PTE 中的任何一个，分页硬件将会抛出一个页面错误异常（page-fault exception），avx512OS 将负责处理这一情况。

三级设计为存储 PTE 提供了一种更为节省内存的方式。在许多虚拟地址范围并未被映射的场景下，这种三级结构能够省略完整的页目录。例如，如果一个应用程序仅使用一个页面，那么最顶级的页目录仅使用第 0 个条目，而忽略从第 1 个到第 511 个的所有条目，这样，avx512OS 就无需为这 511 个条目对应的中级页目录分配页面，更不用为这 511 个中级页目录分配最低级的页目录页面。因此，在这个情况下，这个三级设计仅需占用三个页面，总共占用的字节数为 3×4096 。

由于在执行转换过程中，CPU 需要在硬件中遍历这种三级结构，这种方法的一个缺点是，CPU 必须从内存中提取三个 PTE 以将虚拟地址转换为物理地址。为了降低从物理内存中加载 PTE 的成本，RISC-V CPU 将页表条目缓存在了 Translation Look-aside Buffer (TLB) 中。

每个 PTE 都包含一些标志位，这些标志位向分页硬件说明如何处理相应的虚拟地址。PTE_V 标志位指示 PTE 是否存在：如果未设置，对页面的引用将引发异常（即不被允许）。PTE_R 标志位决定是否允许读取页面的指令。PTE_W 标志位决定是否允许写入页面的指令。PTE_X 标志位决定 CPU 是否能将页面内容解释为指令并执行。PTE_U 标志位决定用户模式下的指令是否能访问页面；如果未设置 PTE_U，那么 PTE 只能在管理模式中使用。所有与页面硬件相关的标志和结构在（kernel/riscv.h）中被定义。

为了让硬件知道应使用哪个页表，avx512OS 需要将根页表页的物理地址写入 satp 寄存器（satp 寄存器的作用是存储根页表页在物理内存中的地址）。每个 CPU 都有自己的 satp，每个 CPU 将使用自己的 satp 指向的页表来转换其后续指令生成的所有地址。由于每个 CPU 都有自己的 satp，所以不同的 CPU 可以运行不同的进程，每个进程都有自己的页表描述的独立地址空

间。

通常，avx512OS 会将所有的物理内存映射到其页表中，这样它就能够使用加载/存储指令来读取和写入物理内存中的任何位置。由于页目录位于物理内存中，所以 avx512OS 可以通过使用标准的存储指令来写入 PTE 的虚拟地址，从而对页目录中的 PTE 内容进行编程。

下面对 avx512OS 虚拟内存管理相关代码进行介绍：虚拟内存管理系统，主要用于分页和内存映射。代码 (vm.c) 中定义的函数主要执行内存分页、页表创建和管理、虚拟地址与物理地址之间的映射和反映射等操作。kvminit(): 这个函数创建一个直接映射的页表 (page table) 用于内核。在函数中，首先会为页表分配空间，然后将页表清零。接着，它会为不同的设备和内核区域设置映射，例如 UART 寄存器，CLINT，PLIC 等。这些映射使得内核可以通过这些虚拟地址访问这些设备。

kvminithart(): 这个函数将硬件的页表寄存器设置为内核的页表，并启用内存分页。walk(): 这个函数返回对应于给定虚拟地址的页表条目 (PTE) 的地址。如果 alloc 为非零，函数将创建任何必需的页表页。

walkaddr(): 这个函数返回与给定虚拟地址对应的物理地址，或者如果该地址没有映射，就返回 0。这个函数只能用来查找用户页。

kvmmap(): 这个函数向内核页表添加一个映射。这个函数只在启动时使用，不刷新 TLB 或启用内存分页。

kvmmap(): 这个函数将内核虚拟地址转换为物理地址。只在栈地址上需要。

mappages(): 这个函数为从 va 开始的虚拟地址创建页表条目，这些页表条目指向从 pa 开始的物理地址。va 和大小可能没有页对齐。

vmunmap(): 这个函数移除从 va 开始的 npages 页的映射。va 必须是页对齐的，映射必须存在。如果 do_free 参数为真，就释放物理内存。

uvmcreate(): 这个函数创建一个空的用户页表。如果内存不足，则返回 0。

uvmminit(): 这个函数将用户的初始代码加载到页表的地址 0，对于第一个进程。大小必须小于一页。

uvmalloc1(): 这个函数为从 start 到 end 的虚拟地址分配 PTE 和物理内存。如果成功，则返回 0，如果失败，则返回-1。

freewalk: 这个函数遍历页表，并递归地释放所有子页表。

uvmfree: 先释放用户内存页，然后释放页表页。

uvmcopy: 复制父进程的页表及其物理内存到子进程的页表。

uvmclear: 将指定虚拟地址的页表条目标记为用户无法访问。

copyout: 从内核复制数据到用户空间。

copyout2: 这是 copyout 的一个简化版本，它直接在虚拟地址空间中操作。

copyin: 从用户空间复制数据到内核。

copyin2: 这是 copyin 的一个简化版本，它直接在虚拟地址空间中操作。

copyinstr: 从用户空间复制一个以 null 结尾的字符串到内核空间。

copyinstr2: 这是 copyinstr 的一个简化版本，它直接在虚拟地址空间中操作。

proc_kpagetable: 初始化每个进程的内核页表。

kfreewalk: 释放页表的内核空间。

kvmfreeusr: 释放用户页表的内核空间。

kvmfree: 释放整个内核页表。

vmprint: 打印页表的详细信息。

experim: 改变指定虚拟地址的页表条目的权限。

以下是 avx512OS 虚拟内存管理设计的主要元素: 分页和页表: 这个操作系统使用分页, 也就是将物理内存和虚拟内存划分为固定大小的“页”, 并用页表进行映射。页表条目中包含了相关页的权限标志位和物理地址。

页表遍历和修改: 代码中有多个函数用于遍历和修改页表。例如, walk 函数遍历页表来查找一个特定的页表条目。mappages 函数则修改页表以映射一段虚拟地址到一段物理地址。

内存释放: freewalk 函数递归地遍历页表并释放已经不再使用的页表页。uvmfree 函数则释放用户内存页和页表页。

内存拷贝: uvmcopy 函数负责从一个进程的页表复制内存到另一个进程的页表, 用于实现进程的创建。copyin, copyout 和 copyinstr 函数则负责从用户空间到内核空间 (或者反过来) 的内存拷贝。

权限管理: 页表条目中的权限标志位用于控制访问该页的权限。例如, uvmclear 函数清除了一个页表条目的用户访问权限。experim 函数则修改一个页表条目的权限。

地址空间管理: vmunmap 函数用于从页表中取消一段虚拟地址的映射。

分页内存映射: mappages 函数将一段虚拟内存映射到物理内存。

复制内核到用户和用户到内核的内存: copyin 和 copyout 函数用于在内核和用户空间之间复制数据。

用户内存的复制: uvmcopy 函数在父进程和子进程之间复制用户内存。

虚拟地址空间的检查: copyin2, copyout2 和 copyinstr2 函数在复制数据之前检查目标虚拟地址是否在合理的地址空间范围内。

每个 AVX512OS 进程都具有独立的页表, 用于映射其虚拟地址空间到物理内存。当 AVX512OS 在不同进程之间进行切换时, 也会相应地切换页表以确保正确的地址映射。下图展示了这一过程: 每个 AVX512OS 进程的用户内存从虚拟地址 0 开始, 可以逐渐增长直到 MAXVA (在 riscv.h 中定义)。这意味着每个进程的内存寻址空间原则上可以达到 256GB。

通过使用独立的页表和虚拟地址空间的切换, AVX512OS 实现了进程之间的隔离和内存地址的映射。这种设计确保了每个进程在其虚拟地址空间内拥有独立的内存空间, 同时提供了灵活性和可扩展性, 使得进程可以使用大内存空间来满足各种应用程序的需求。

当 AVX512OS 进程向系统请求更多的用户内存时, AVX512OS 会采取以下步骤。首先, 它使用 kalloc 函数来分配物理页面, 为进程分配新的物理内存页。然后, AVX512OS 将相应的页表项 (Page Table Entry, PTE) 添加到进程的页表中, 将其指向新分配的物理页面。在这些页表项中, AVX512OS 设置了 PTE_W、PTE_X、PTE_R、PTE_U 和 PTE_V 标志, 以指示页面的读写、执行和用户访问权限。值得注意的是, 大多数进程并不使用整个用户地址空间, 因此 AVX512OS 在未使用的页表项中将 PTE_V 标志设置为空闲状态。

这里展示了一些使用页表的典型示例。首先, 不同进程的页表将用户地址转换为不同的物理内存页面, 这样每个进程都拥有独立的私有内存空间。其次, 每个进程所看到的用户内存空间都以虚拟地址 0 开始, 并可以是非连续的物理内存。第三, 内核在用户地址空间的顶部映射一个包含蹦床 (trampoline) 代码的页面, 使得在所有进程的地址空间中都可以访问到同一个物理内存页面。

栈被分配为一个单独的页面, 页面的内容在 exec 系统调用后被初始化。栈的顶部包含命令行参数的字符串以及指向它们的指针数组。在栈的下方是一些用于启动程序的值, 例如 main 函数

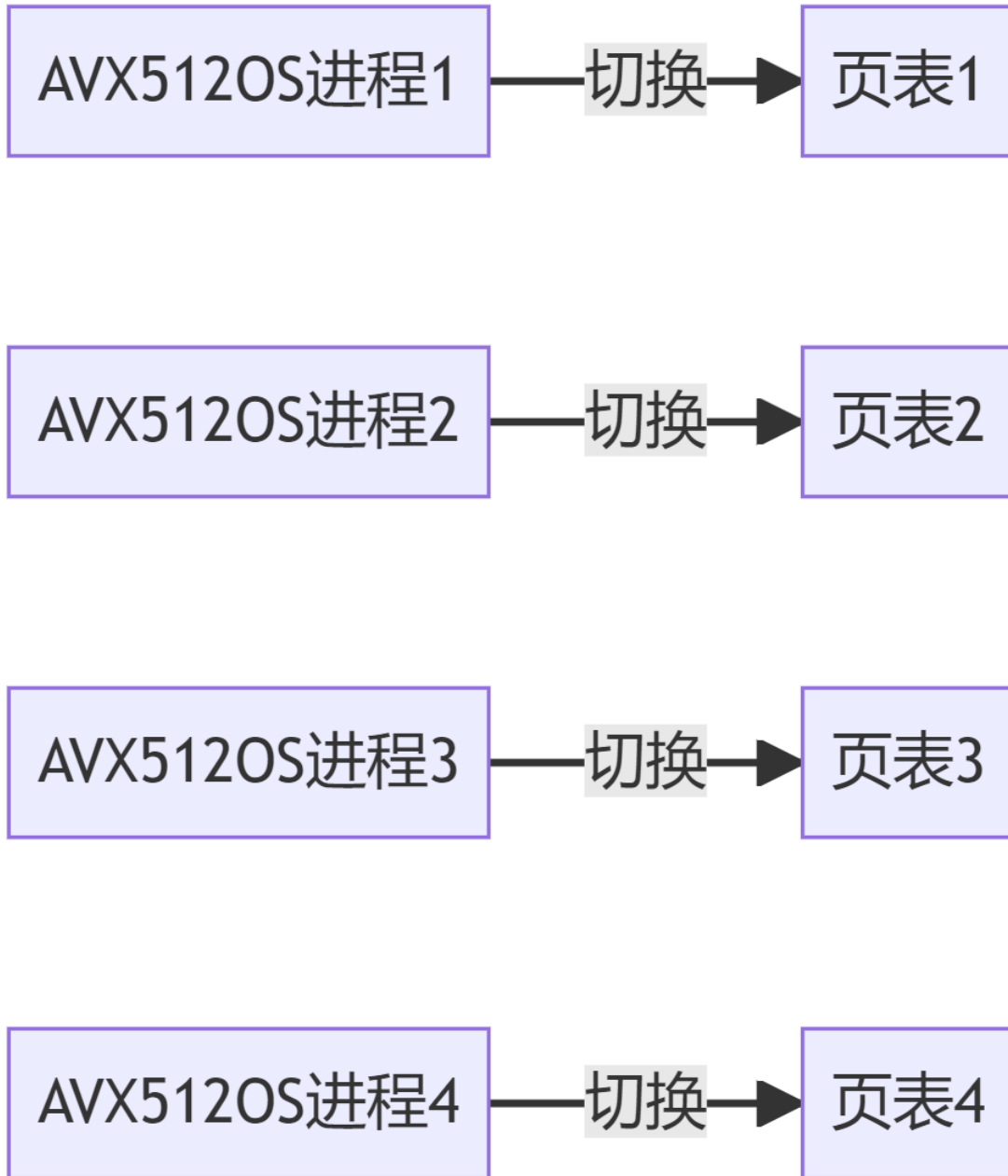


图 5: 进程页表

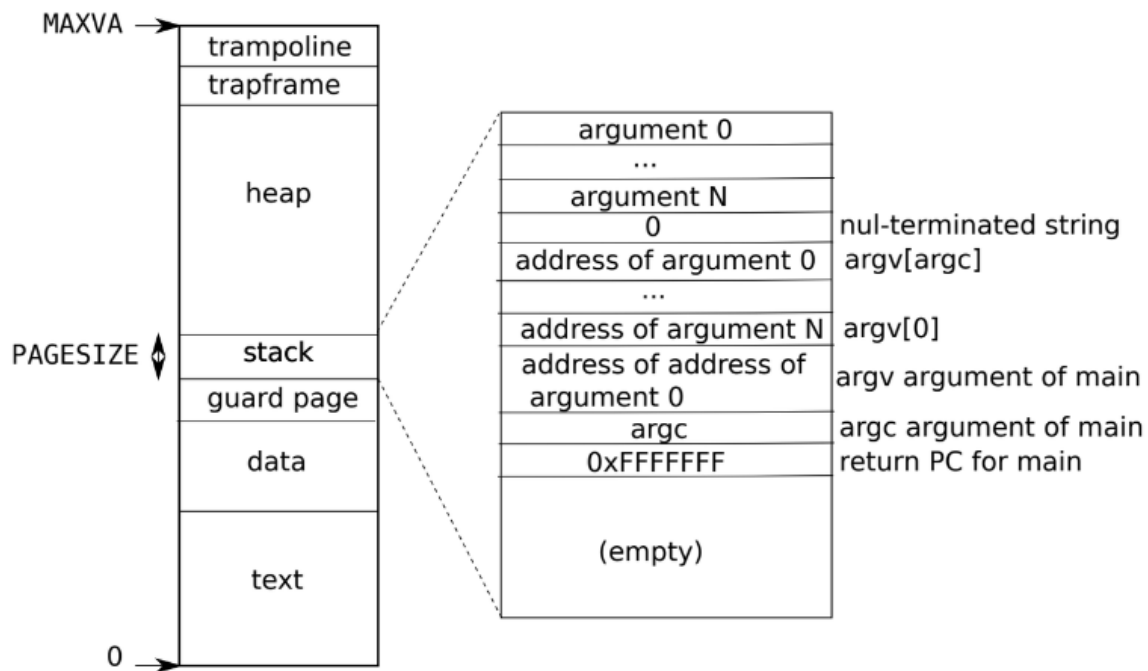


图 6: 地址空间

的地址、`argc` 和 `argv` 参数，这些值使得程序看起来像刚刚调用了 `main(argc, argv)` 一样开始执行。为了检测用户栈是否溢出，AVX512OS 在栈的下方放置了一个无效的保护页（guard page）。如果用户栈溢出并且进程尝试使用栈下方的地址，由于页表项的映射无效（`PTE_V` 为 0），硬件将产生一个页面故障异常。当用户栈溢出时，实际的操作系统可能会自动为其分配更多的内存来避免出现问题。

未来的改进方向有：引入高级内存管理功能：可以考虑引入内存压缩、NUMA 感知、内存池预分配等技术，以进一步优化内存使用。

增强错误处理：设计更详细的错误处理机制，比如当映射失败或者访问权限违规时，应该有明确的错误码和处理逻辑。

增加缓存或预取机制：通过预读取和缓存，可以显著提高内存的访问速度。

内存保护和安全性增强：可以考虑引入更多的内存保护机制，例如内存隔离、内存清零等，来提高系统的安全性。

3.3 文件系统

3.3.1 总览

AVX512OS 操作系统支持的文件系统为 `fat32` 文件系统，这一文件系统具有设计简单，容易实现的好处。在本节中，我们不会具体的介绍 `fat32` 文件系统，这一文件系统的资料在互联网上非常多。我们将重点介绍 AVX512OS 文件系统的设计。

3.3.2 文件系统分层

文件系统采取的是分层的结构，这样的实现清晰易懂，容易了解。从低到上，AVX512OS 文件系统的分层如下：

1. 磁盘读写层，这一层主要负责与底层的磁盘进行交互，当 AVX512OS 运行在 qemu 上，是通过 virtio 与虚拟磁盘进行交互，当 AVX512OS 运行在 visionfive 上，是通过 sdio 与 sd 卡进行交互。其基本操作包括磁盘的初始化，磁盘的读和磁盘的写。

2. 抽象层，分流层。这一层的主要作用是对底层读写 api 进行包装，主要通过宏定义进行选择，选择执行 sd 卡函数还是虚拟磁盘的函数。这一层抽象出对 sd 卡和虚拟磁盘的操作不同，对上提供磁盘的统一读，写 api。他们均是以扇区为单位进行读写。

3. 磁盘扇区缓冲层。文件系统以块 (扇区) 为基本读写单位，因此缓存的基本单位也是块。这一层的主要结构为对磁盘的缓冲 buf，一个 buf 是对一个扇区的缓冲，一个缓冲结构体可以记录一个对应扇区号的对应扇区数据。其中，buf 的结构如下：

```
1 struct buf {
2     int valid;
3     int disk;
4     uint dev;
5     uint sectorno;
6     struct sleeplock lock;
7     uint refcnt;
8     struct buf *prev;
9     struct buf *next;
10    uchar data[BSIZE];
11 };
```

sectorno 记录了缓冲结构对应的扇区号，data 字段则对应存储了底层磁盘上的数据。在这里，我们对 buf 的驱逐算法采用了 lru 算法。通过双向链表来实现这一算法。

通过缓存，我们屏蔽了扇区读写的具体细节。我们定义了 bget, brelease, bread, bwrite 四个接口，通过这些接口，上层模块不必考虑分区的起始偏移等问题，只需向缓存寻求需要的逻辑块号，缓存就会帮其获取。

4. fat32 文件系统层。这一层依靠上一层的 buf 结构，开始构造，实现，解析 fat32 文件系统，这一层实现了从单纯的扇区数据读写，到有组织有结构的 fat32 文件系统的转变。

5. 文件层。在这一层，操作系统提供了文件抽象，基于 unix “万物皆文件”的设计思想，将设备，管道，文件都统一为文件，这一层也会顺利为进程提供进程描述符的功能。文件的主要设计结构为：

```
1 struct file {
2     enum { FD_NONE, FD_PIPE, FD_ENTRY, FD_DEVICE } type;
3     int ref; // reference count
4     char readable;
5     char writable;
6     struct pipe *pipe; // FD_PIPE
```

```

7   struct dirent *ep;
8   uint off;           // FD_ENTRY
9   short major;        // FD_DEVICE
10  };

```

在这一层，对上提供了统一的读写 api，对下则根据文件的种类，分别调取不同的 api 函数对文件进行读写操作。

4 系统调用的设计实现

4.1 系统调用的流程

对于 riscv 体系结构而言，系统调用通过专用指令 ecall 进行。在用户态，我们将系统调用的参数保存在 a0、a1 等寄存器中，而 a7 寄存器则用于保存系统调用号。当执行 ecall 指令时，会主动触发一次异常，并进入到异常处理流程中。在系统初始化的起始阶段，我们已经设置好了 stvec 寄存器，因此当异常发生时，操作系统将执行 stvec 寄存器指向的函数。

在异常触发后，首先需要将用户态的寄存器以及用户态进程需要保留的数据结构保存到一个指定的 trapframe 结构中，然后进行内核态的操作。我们会读取 scause 寄存器的值，如果发现其值为 8，即异常是由 ecall 指令主动触发的，接下来将进入系统调用的处理流程。

为了处理系统调用，我们设计了一个函数指针数组，将对应系统调用号所要执行的函数保存在相应的数组位置。通过根据系统调用号调用对应的函数，我们能够执行相应的系统调用操作。这样的设计使得系统能够根据不同的系统调用号来调用相应的功能函数，实现了系统调用的灵活性和扩展性。

4.2 一些系统调用的实现

4.2.1 brk、sbrk 系统调用

brk 是一个用于修改指定数据段大小的系统调用。该系统调用只有一个参数，该参数为指定待修改的数据段地址。用户程序通过 brk 系统调用，来向内核申请空间，系统将把用户进程空间扩大到指定的数据段地址。

同时实现的 sbrk 系统调用的功能类似，该系统调用的参数为需要申请的地址空间大小。该参数若为正，则会给用户程序分配新的地址空间。若为负，则会释放指定大小的地址空间。

brk 与 sbrk 系统调用主要依靠 growproc 函数来实现，该函数根据传入的参数的正负，分别调用 uvmalloc 函数和 uvmdealloc 函数来分配/释放物理内存。下面主要介绍一下 uvmalloc 函数和 uvmdealloc 函数的实现逻辑。

uvmalloc 函数核心代码如下：

```

1  for(a = oldsz; a < newsz; a += PGSIZE){
2      mem = kalloc();
3      if(mem == NULL){
4          uvmdealloc(pagetable, kpagetable, a, oldsz);
5          return 0;
6      }

```

```

7     memset(mem, 0, PGSIZE);
8     if (mappages(pagetable, a, PGSIZE, (uint64)mem, PTE_W|PTE_X
9         |PTE_R|PTE_U) != 0) {
10         kfree(mem);
11         uvmdealloc(pagetable, kpagetable, a, oldsz);
12         return 0;
13     }
14     if (mappages(kpagetable, a, PGSIZE, (uint64)mem, PTE_W|
15         PTE_X|PTE_R) != 0){
16         int npages = (a - oldsz) / PGSIZE;
17         vmunmap(pagetable, oldsz, npages + 1, 1);    // plus the
18             page allocated above.
19         vmunmap(kpagetable, oldsz, npages, 0);
20         return 0;
21     }
22 }

```

uvmalloc 函数主体由一个循环构成，循环体从调用以前的用户空间大小到新的用户空间大小之间的区域，遍历每一个页面（大小为 PGSIZE）。通过 kalloc 函数申请一块物理内存，申请成功后通过 mappages 函数创建 PTE（页表项，从虚拟内存映射到物理内存），将 PTEs 添加到用户页表中。以上环节若有任何步骤分配出现问题，则立刻释放此前所有以分配的空间以及 PTE，并退出该函数。

接下来是 uvmdealloc 函数。该函数用于释放物理内存，主要实现方式就是通过 uvmunmap 函数释放对应区域的用户页表的所有页表项。核心代码如下：

```

1 int npages = (PGROUNDUP(oldsz) - PGROUNDUP(newsz)) / PGSIZE;
2 vmunmap(kpagetable, PGROUNDUP(newsz), npages, 0);
3 vmunmap(pagetable, PGROUNDUP(newsz), npages, 1);

```

4.2.2 yield 系统调用

yield 系统调用用于线程调度操作，主要功能是让出内核线程调度器。当中断处理程序结束时，usertrap 调用了 yield。依次地：Yield 调用 sched，sched 调用 swtch 将当前上下文保存在 p->context 中，并切换到先前保存在 cpu->scheduler 中的调度程序上下文。

yield 函数只做了几件事情，它首先获取了进程的锁。实际上，在锁释放之前，进程的状态会变得不一致，例如，yield 将要进程的状态改为 RUNNABLE，表明进程并没有在运行，但是实际上这个进程还在运行，代码正在当前进程的线程中运行。所以这里加锁的目的之一就是：即使我们将进程的状态改为了 RUNNABLE，其他的 CPU 核的调度器线程也不可能看到进程的状态为 RUNNABLE 并尝试运行它。否则的话，进程就会在两个 CPU 核上运行了，而一个进程只有一个栈，这意味着两个 CPU 核在同一个栈上运行代码（注，因为 AVX512OS 中一个用户进程只有一个用户线程）。

接下来 yield 函数中将进程的状态改为 RUNNABLE。这里的意思是，当前进程要出让 CPU，

并切换到调度器线程。当前进程的状态是 RUNNABLE 意味着它还会再次运行，因为毕竟现在是一个定时器中断打断了当前正在运行的进程。yield 代码如下：

```
1 struct proc *p = myproc();
2 acquire(&p->lock);
3 p->state = RUNNABLE;
4 sched();
5 release(&p->lock);
```

yield 函数中调用了 sched 函数，sched 函数在 AVX512OS 进程调度中是一个比较重要的函数。sched 函数主要是进行一些条件的判断，防止异常情况的出现。随后调用 swtch 函数进行程序上下文的切换（该内容存储在 struct context 中）。如图 4.1 所示概述了从一个用户进程（旧进程）切换到另一个用户进程（新进程）所涉及的步骤。一个到旧进程内核线程的用户-内核转换（系统调用或中断），一个到当前 CPU 调度程序线程的上下文切换，一个到新进程内核线程的上下文切换，以及一个返回到用户级进程的 trap 程序。

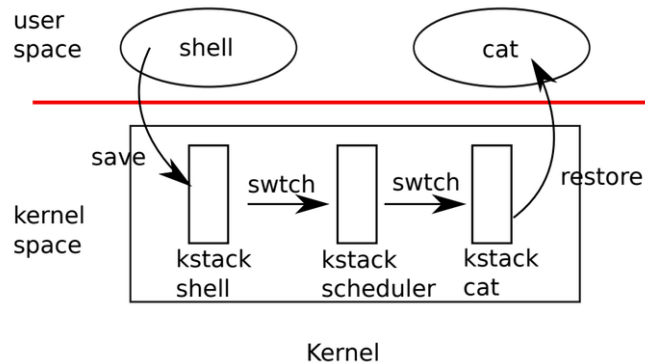


图 7: AVX512OS 进程调度流程

函数 swtch 为内核线程切换执行保存和恢复操作。swtch 对线程没有直接的了解；它只是保存和恢复寄存器集，称为上下文（contexts）。当某个进程要放弃 CPU 时，该进程的内核线程调用 swtch 来保存自己的上下文并返回到调度程序的上下文。每个上下文都包含在一个 struct context 中，这个结构体本身包含在一个进程的 struct proc 或一个 CPU 的 struct cpu 中。Swtch 接受两个参数：struct context *old 和 struct context *new。它将当前寄存器保存在 old 中，从 new 中加载寄存器，然后返回。以上就是 yield 函数的函数调用全流程。

4.2.3 wait 系统调用

wait 系统调用，用于等待子进程运行结束。在 AVX512OS 中，主要用于内核需要杀死某父进程时，需要调用 wait 函数等待所有子进程都已结束，才能杀死父进程。传入参数为一个指针类型，将子进程中为 ZOMBIE 状态的进程的对应信息存到该指针指向的地址中。

wait 函数主要实现方法如下。首先看看遍历看看有没有孩子，没有孩子或者当前进程被杀死就直接退出。有孩子进程就一定会 wait 到一个孩子变为僵尸进程为止，每次扫描一轮没有发生有子进程为僵尸进程就陷入睡眠，子进程调用 exit 后会唤醒父进程。如果找到状态为 ZOMBIE 的子进程，回收他的页表、内核栈、进程控制块。

4.2.4 read 系统调用

read 系统调用用于操作系统的输入功能，该系统调用将从指定的文件描述符读取内容。以下为 read 系统调用的传入参数：

fd：要读取文件的文件描述符。

buf：一个缓存区，用于存放读取的内容。

count：要读取的字节数。

根据文件描述符的不同，将会处理不同类型的输入流：PIPE,DEVICE,ENTRY。这里主要介绍以下 DEVICE 类型的输入流实现。AVX512OS 对于每一个设备都由一个结构体来维护，同时，AVX512OS 还有一个 devsw 结构体数组用于维护各设备的 IO 情况。以标准控制台输入为例，当 read 系统调用的文件描述符表示为控制台输入的情况时，将会调用 consoleread 函数。

在 consoleread 函数中，程序将等待输入到达（通过中断）并在 cons.buf 中缓冲，将输入复制到用户空间，然后（在整行到达后）返回给用户进程。如果用户还没有键入整行，任何读取进程都将在 sleep 系统调用中等待。而输入流是如何到达 cons.buf 缓冲的，具体将在设备中断部分介绍。

4.2.5 mmap 系统调用

mmap 系统调用用于将文件或设备映射到内存中。以下为 mmap 系统调用传入的参数：

start：映射起始位置（Start Address）

它指定了映射区域在进程地址空间中的起始位置。可以将文件或设备映射到指定的虚拟内存地址，或者通过传递 NULL 来由系统自动选择一个可用的地址。

len：长度（Length）

它指定了映射区域的长度，即需要映射的文件或设备的大小。长度以字节为单位，并且通常是页大小的整数倍。

prot：映射的内存保护方式（Protection）

它定义了映射区域的内存保护方式，指定了进程对映射区域的访问权限。常见的保护方式包括：PROT_READ：可读取映射区域的内容。PROT_WRITE：可写入映射区域的内容。PROT_EXEC：可执行映射区域的内容。

flags：映射是否与其他进程共享的标志（Flags）

它用于指定映射的共享方式和其他标志。常见的标志包括：MAP_SHARED：映射区域与其他进程共享，对映射的修改会影响到其他进程。MAP_PRIVATE：映射区域私有，对映射的修改不会影响到其他进程。MAP_FIXED：强制使用指定的起始地址进行映射。

fd：文件句柄（File Descriptor）

它是要映射的文件的文件描述符，通过该文件描述符可以确定要映射的文件。

off：文件偏移量（File Offset）

它指定了要映射的文件的起始位置偏移量，即从文件的哪个位置开始映射。通常用于指定映射文件的某个特定部分。

首先，函数获取当前进程的结构体指针，并进行参数检查。如果文件描述符（fd）小于 0、偏移量（offset）小于 0 或者起始地址（start）不是页面大小（PGSIZE）的整数倍，则返回-1 表示参数错误。

接下来，定义了一个权限变量（perm）并初始化为 PTE_U，表示用户级别的权限。注释部分代码表示如果保护方式（prot）包含 PROT_READ，则将 perm 中加入 PTE_R 和 PTE_A 标志，如果包含 PROT_WRITE，则将 perm 中加入 PTE_W 和 PTE_D 标志。

接着，根据文件描述符（fd）获取当前进程打开的文件结构体指针（struct file *f），如果 fd 为-1 则表示不关联任何文件，否则需要确保文件指针 f 不为空。

调用 alloc_mmap_vma 函数为当前进程分配一个 vma（Virtual Memory Area）结构体，并传递相关参数。该函数将在进程的虚拟内存区域中创建一个新的映射区域（vma），并返回 vma 结构体的指针。如果无法成功分配 vma，返回-1 表示失败。

更新起始地址（start）为分配的 vma 的起始地址（vma->addr）。

如果文件描述符（fd）不等于-1，表示关联了一个文件，那么计算 mmap 的大小（mmap_size），即映射文件的大小减去偏移量。如果 len 小于 mmap_size，则将 mmap_size 设置为 len，以确保不超过 len 的范围。然后将文件的偏移量（f->off）设置为给定的偏移量（offset）。

接下来，计算 mmap_size 对页面大小（PGSIZE）求余的结果（end_pagespace），以及需要映射的页面数目（page_n）。同时定义一个虚拟地址变量（va）并初始化为起始地址（start）。

进入循环，循环次数为页面数目（page_n）。在每次循环中，通过调用 experm 函数将虚拟地址（va）映射到物理地址（pa），并传递权限变量（perm）。如果映射失败（pa 为 NULL），返回-1 表示失败。

在循环中，根据当前循环的页是否为最后一页进行不同的操作。如果不是最后一页，调用 fileread 函数从文件中读取 PGSIZE 大小的数据，并写入虚拟地址（va）指向的内存。如果是最后一页，调用 fileread 函数从文件中读取 end_pagespace 大小的数据，并写入虚拟地址（va）指向的内存。然后使用 memset 函数将剩余部分（PGSIZE-end_pagespace）的内存清零。

循环结束后，调用 filedup 函数对文件结构体进行引用计数的增加，以确保在 mmap 映射期间文件不会被关闭。

最后，返回起始地址（start），表示 mmap 映射成功。

接下来介绍一下 vma 结构：

```
1      struct vma {
2          enum segtype type;      // VMA类型
3          uint64 addr;            // VMA起始地址
4          uint64 sz;              // VMA大小
5          int perm;               // VMA的访问权限
6          uint64 end;             // VMA结束地址
7          int fd;                 // VMA关联的文件描述符
8          uint64 f_off;           // VMA在文件中的偏移量
9          struct vma *prev;       // 上一个VMA的指针
10         struct vma *next;       // 下一个VMA的指针
11     };
```

struct vma 结构体表示一个虚拟内存区域。它包含了该区域的类型、起始地址、大小、访问权限、结束地址、关联的文件描述符和文件偏移量等信息。同时，通过指针 prev 和 next，可以将多个 VMA 连接成链表。

函数 vma_init 该函数用于初始化进程的 VMA，并返回指向 VMA 结构体的指针。它接受一个指向进程结构体的指针作为参数。在函数内部，它会为 VMA 结构体分配内存，并将各个字段进行初始化。然后，它会将 VMA 结构体与进程关联，并创建一个初始的 MMAP 类型的 VMA，起始地址为 USER_MMAP_START，大小为 0。若分配和初始化过程中出现错误，则返回 NULL。

函数 `alloc_vma` 该函数用于分配一个 VMA 并将其插入进程的 VMA 链表中。它接受进程指针 `p`、VMA 类型 `type`、VMA 的起始地址 `addr`、大小 `sz`、访问权限 `perm`、是否进行内存分配 `alloc`、物理地址 `pa` 等参数。函数首先根据地址和大小检查是否存在冲突的 VMA，然后分配一个 VMA 结构体，并根据 `alloc` 的值进行内存分配或者页面映射。最后，将 VMA 结构体的各个字段进行赋值，并插入进程的 VMA 链表中。如果分配或者映射过程中出现错误，则返回 `NULL`。

函数 `find_mmap_vma` 该函数用于在给定的 VMA 链表中查找类型为 `MMAP` 的 VMA。它接受一个指向 VMA 链表头结点的指针，并通过遍历链表来查找类型为 `MMAP` 的 VMA。如果找到，则返回该 VMA 的指针；否则返回 `NULL`。

函数 `alloc_mmap_vma` 该函数用于为进程分配一个 `MMAP` 类型的 VMA。它接受进程指针 `p`、标志 `flags`、起始地址 `addr`、大小 `sz`、访问权限 `perm`、文件描述符 `fd` 和文件偏移量 `f_off` 等参数。函数首先通过调用 `find_mmap_vma` 函数找到 `MMAP` 类型的 VMA，然后根据给定的参数分配一个新的 VMA，并将其插入进程的 VMA 链表中。最后，设置新的 VMA 的文件描述符和文件偏移量，并返回该 VMA 的指针。如果分配过程中出现错误，则返回 `NULL`。

以上是对 `vma.c` 中的结构体和函数的详细介绍。这些函数和结构体用于管理进程的虚拟内存区域，包括初始化 VMA、分配 VMA、查找 `MMAP` 类型的 VMA 等操作。

```
1      struct vma *vma_init(struct proc *p)
2  {
3      if (NULL == p) {
4          printf("p is not existing\n");
5          return NULL;
6      }
7      struct vma *vma = (struct vma*)kalloc();
8      if (NULL == vma) {
9          printf("vma kalloc failed\n");
10         return NULL;
11     }
12
13     vma->type = NONE;
14     vma->prev = vma->next = vma;
15     p->vma = vma;
16
17     if (NULL == alloc_mmap_vma(p,0,USER_MMAP_START,0,0,0,0)) {
18         //free_vma_list(p);
19         return NULL;
20     }
21
22     return vma;
23 }
24
25 struct vma* alloc_vma(struct proc *p,enum segtype type,uint64
    addr, uint64 sz,int perm,int alloc,uint64 pa) {
```



```

26     uint64 start = PGROUNDDOWN(addr);
27     uint64 end = addr + sz;
28     end = PGROUNDUP(end);
29
30     struct vma* find_vma = p->vma->next;
31     while (find_vma != p->vma) {
32         if (end <= find_vma->addr)
33             break;
34         else if (start >= find_vma->end)
35             find_vma = find_vma->next;
36         else {
37             printf("vma address overflow\n");
38             return NULL;
39         }
40     }
41     struct vma* vma = (struct vma*)kalloc();
42     if (NULL == vma) {
43         printf("vma kalloc failed\n");
44         return NULL;
45     }
46     if (0 != sz) {
47         if (alloc) {
48             if (0 != uvmmalloc(p->pagetable, start, end, perm)) {
49                 printf("uvmmalloc failed\n");
50                 kfree(vma);
51                 return NULL;
52             }
53         } else if (pa != 0) {
54             if (0 != mappages(p->pagetable, start, sz, pa, perm)) {
55                 printf("mappages failed\n");
56                 kfree(vma);
57                 return NULL;
58             }
59         }
60     }
61     vma->addr = start;
62     vma->sz = sz;
63     vma->perm = perm;
64     vma->end = end;
65     vma->fd = -1;
66     vma->f_off = 0;

```

```

67     vma->type = type;
68     vma->prev = find_vma->prev;
69     vma->next = find_vma;
70     find_vma->prev->next = vma;
71     find_vma->prev = vma;
72
73     return vma;
74 }
75
76 struct vma* find_mmap_vma(struct vma* head)
77 {
78     struct vma* vma = head->next;
79     while (vma != head) {
80         if (MMAP == vma->type)
81             return vma;
82         vma = vma->next;
83     }
84     return NULL;
85 }
86
87 struct vma* alloc_mmap_vma(struct proc *p, int flags, uint64
88     addr, uint64 sz, int perm, int fd, uint64 f_off)
89 {
90     struct vma* vma = NULL;
91     struct vma* find_vma = find_mmap_vma(p->vma);
92     if (0 == addr)
93         addr = PGROUNDDOWN(find_vma->addr - sz);
94     vma = alloc_vma(p, MMAP, addr, sz, perm, 1, NULL);
95     if (NULL == vma)
96     {
97         printf("alloc_mmap_vma: alloc_vma failed\n");
98         return NULL;
99     }
100     vma->fd = fd;
101     vma->f_off = f_off;
102
103     return vma;
104 }

```

4.2.6 clone 系统调用

clone 系统调用用于创建一个子进程。以下为 clone 系统调用传入的参数

flags: 创建的标志, 如 SIGCHLD; stack: 指定新进程的栈, 可为 0; ptid: 父线程 ID; tls: TLS 线程本地存储描述符; ctid: 子线程 ID;

clone 函数是相对于 UNIX 中原有的 fork 函数的一个更加完善的版本。在 clone 系统调用中, 可以指定一个新的用户栈空间。clone 函数接收一个预先分配的用户空间的虚拟地址作为用户栈, 将子进程的 proc 结构体的 sp 属性改为对应的用户栈地址。

此外, 与 fork 函数类似, 需要将父进程的用户空间复制一份, 即子线程将共享了父进程的进程页表。此外, 父进程使用的文件描述符也需要复制给子进程。clone 核心代码如下:

```
1 // Copy user memory from parent to child.
2 if (uvmcopy(p->pagetable, np->pagetable, np->kpagetable, p->sz
3         ) < 0){
4     freeproc(np);
5     release(&np->lock);
6     return -1;
7 }
8 np->sz = p->sz;
9 np->parent = p;
10
11 // copy tracing mask from parent.
12 np->tmask = p->tmask;
13
14 // copy saved user registers.
15 *(np->trapframe) = *(p->trapframe);
16
17 // Cause fork to return 0 in the child.
18 np->trapframe->a0 = 0;
19
20 // increment reference counts on open file descriptors.
21 for (i = 0; i < NOFILE; i++)
22     if (p->ofile[i])
23         np->ofile[i] = filedup(p->ofile[i]);
```

可以看到, clone 函数和 fork 函数的主体部分近乎相同, 主要完成的任务是: 分配任务结构体, 初始化任务结构体; 分配内核栈, 模拟上下文填充内核栈; 复制父进程数据、创建“新”页表; 复制文件描述符表; 修改进程结构体属性。

4.2.7 times 系统调用

times 系统调用用于返回进程时间，输入参数为 tms 结构体指针，用于获取保存当前进程的运行时间数据。该系统调用需要成功返回已经过去的滴答数，失败则返回-1。

需要注意的是，对于一个进程来说，它的进程时间包括其所有子进程的 CPU 时间，因此在 times 系统调用中，需要将所有子进程的 CPU 时间求和结果同样返回。

CPU 时间包括内核态时间与用户态时间，分别表示一个进程运行的生命周期中，分别处于内核态的时间以及用户态的时间。times 系统调用代码如下：

```
1 struct tms ptms;
2     uint64 utms;
3     argaddr(0, &utms);
4     ptms.tms_utime = myproc()->utime;
5     ptms.tms_stime = myproc()->ktime;
6     ptms.tms_cstime = 1;
7     ptms.tms_cutime = 1;
8     struct proc *p;
9     for(p = proc; p < proc + NPROC; p++){
10         acquire(&p->lock);
11         if(p->parent == myproc()){
12             ptms.tms_cutime += p->utime;
13             ptms.tms_cstime += p->ktime;
14         }
15         release(&p->lock);
16     }
17     copyout2(utms, (char*)&ptms, sizeof(ptms));
```

以上便是通过 times 系统调用求进程 CPU 时间的方法。那么，最后的问题就是，记录内核时间和用户态时间的 utime 和 ktime 变量是如何维护的呢？答案很简单，利用时钟中断。对于 utime 来说，可以在每次 usertrap 引发的时钟中断的时候将 utime 的值加一即可。对于 ktime，则在 kerneltrap 中处理的时钟中断时将 ktime 的值加一即可。以上便是实现 times 系统调用的所有逻辑。

5 系统测试

5.1 测试方法

5.1.1 思路

初赛提测的思路是考虑 AVX512OS 初始化完成后，将会运行 initcode.S 中的代码，这个程序的主要作用是执行系统调用 exec(/init)，故而，我们可以在 initcode.S 中直接运行测试程序，注意这里我们是可以直接拿到测试程序的名称的。

其实最基本的思路应该是检测根目录下所有的文件，但是考虑到用户程序根本无法带到测评环境，如何使项目中包含用户代码便有些困难。故而我出此下策，使用纯汇编代码运行所有的程

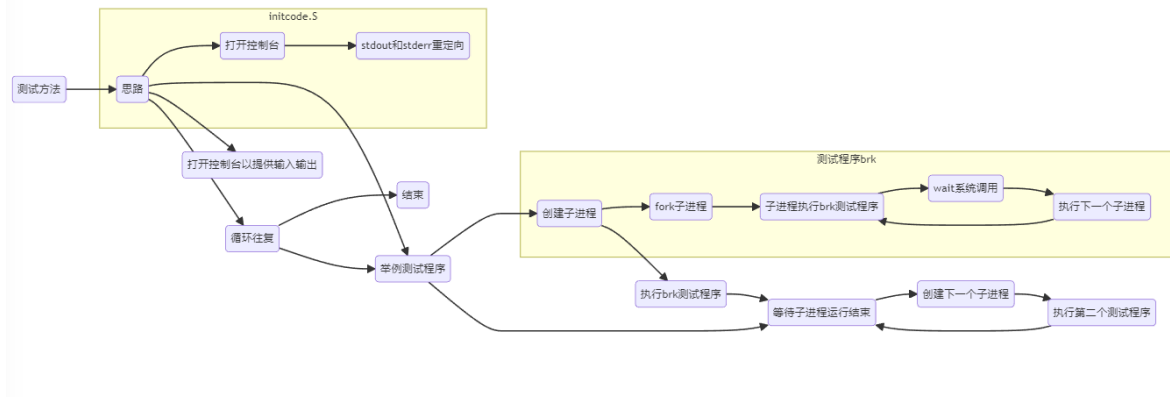


图 8: 系统测试

序。

提测代码为了便于区别，增加文件 `init-for-test.S`。

首先，注意打开控制台以给出 `stdin`，同时使用 `dup` 系统调用给出 `stdout` 和 `stderr`。

```

1  .globl start
2  start:
3      li a0, 2    //O_RDWR
4      la a1, 1    //console
5      li a2, 0
6      li a7, SYS_dev
7      ecall      # dev(O_RDWR, CONSOLE, 0);
8
9      li a0, 0
10     li a7, SYS_dup
11     ecall      # dup(0);  // stdout
12
13     li a0, 0
14     li a7, SYS_dup
15     ecall      # dup(0);  // stderr

```

接下来举一个测试程序 `brk` 的例子。先使用 `fork` 以创建一个子进程，接着子进程执行 `brk` 测试程序，父进程跳到下一段代码中去，使用 `wait` 系统调用，等待子进程运行结束，再去创建下一个子进程，下一个子进程执行第二个测试程序，循环往复。

```

1      li a7, SYS_fork
2      ecall
3      bne a0, zero, after_brk

```

```

4          la a0, brk
5          la a1, argv_brk
6          li a7, SYS_exec
7          ecall
8  after_brk:
9          li a0, 0
10         li a7, SYS_wait
11         ecall

```

其中 brk 和 argv_brk 的定义如下:

```

1  brk:
2          .string "/brk\0"
3  argv_brk:
4          .long brk
5          .zero 12 //不知道为什么要填充12个字节，实际上才能让编译
                   出来的代码填充了8个0字节。

```

brk 是待运行程序的字符串, argv_brk 是程序的运行参数 argv 的指针数组, 第一个指针指向/brk 字符串, 第二个应该为空指针。理论上应该为 8 字节 long long 类型的 0, 也就是空指针, 但是鉴于一些编译器的神奇操作, 在这里我们使用.zero 12 才可以是编译出来的二进制代码在这里填写 8 字节的 0。

后续程序按照上述示例照着写即可。

makefile 中已经写好了相关的编译命令, 只需执行

make all

然后利用 tools/cmd.txt 中的命令, 将二进制代码按照我们指定的格式提取出来。随后将其加入到我们创建的 uchar init_for_test[] 数组中, 直接初始化。

考虑到这个 uchar 数组的大小没有超过 4096Bytes, 也就是一页, 我们直接修改第一个用户程序的映射函数即可。(在 void userinit(void))

```

1  //uvminit(p->pagetable , p->kpagetable , initcode , sizeof(
        initcode));
2  uvminit(p->pagetable , p->kpagetable , init_for_test , sizeof(
        init_for_test));

```

如果要进入 shell, 运行上面的代码, 如果要运行测试程序, 运行下面的代码。

5.2 测试结果

测试结果全部通过了大赛的系统调用要求。

点击查看/隐藏评测结果		
得分102.00 最后一次提交时间:2023-05-22 23:21:08		
Accept		
开始评测时间: 2023-05-22 23:16:52.686482+08:00		
结束评测时间: 2023-05-22 23:19:00.878252+08:00		
开发板编号: QEMU		
通过测试点: 102/102		
得分: 102		
测试结果		
测试样例名	通过测试点	全部测试点
test_mount	5	5
test_read	3	3
test_uname	2	2
test_fstat	3	3
test_mmap	3	3
test_getdents	5	5
test_munmap	4	4
test_getcwd	2	2
test_open	3	3
test_getpid	3	3
test_wait	4	4
test_umount	5	5
test_unlink	2	2
test_openat	4	4
test_gettimeofday	3	3
	-	-

图 9: 测试结果

6 总结与展望

在 avx512OS 项目中, 一个基于 RISC-V64 架构的多核操作系统, 通过在 MIT 教学操作系统 xv6 的基础上进行改进, 旨在强化华中科技大学计算机系统教育中的理论与实践平衡。该项目的目标是提供学生一个自由度较大的操作系统开发环境, 促进他们对操作系统的理解和探索。

avx512OS 的开发主要集中在改进硬件抽象和提升系统性能上。其中, 对硬件抽象化接口(SBI)进行了改进, 包括中断处理机制和无效指令软处理机制。同时, 对用户抽象化接口(ABI)进行了改进, 增加了新的系统调用。为了提升系统的鲁棒性, 进行了错误处理机制和内存保护的改进。此外, 还进行了性能优化, 包括调度算法和内存管理的改进。

在实现阶段, avx512OS 成功地实现了 33 个系统调用以完成操作系统的基本功能, 并在实现过程中附带实现了其他系统调用, 进一步完善了整个操作系统。为了支持评测系统, 还实现了非在线库的项目依赖、避免外部设备的自动启动依赖, 并实现了自动运行评测程序的功能。

未来, avx512OS 项目可以继续发展和扩展。可能的方向包括进一步改进硬件抽象和系统性能, 探索新的调度算法和内存管理策略, 加强系统的安全性和可靠性。此外, 可以考虑支持更多的系统调用和功能, 以满足不同应用场景的需求。同时, 可以将 avx512OS 作为教学操作系统的基础, 进一步丰富计算机系统教育的内容和实践环节。

总之, avx512OS 项目在华中科技大学计算机系统教育中具有重要的意义, 通过开发一个自由度较大的操作系统开发环境, 促进学生对操作系统的深入理解和实践。该项目在硬件抽象、系统性能、鲁棒性和功能方面取得了显著的进展, 并展示了其潜力和扩展性。未来的发展可以继续探索新的改进和功能扩展, 为学生提供更好的教学平台, 并为计算机系统领域的研究做出贡献。