

# NoWaitOS

余奕俊，张宏斌，李明凯

2024 年 12 月 25 日

# 目录

<b>1. 介绍</b>	<b>4</b>
1.1 项目背景及意义	4
1.2 国内外研究状况	4
1.3 项目的主要工作	5
<b>2. 设计目标</b>	<b>6</b>
<b>3. 系统设计与实现</b>	<b>7</b>
3.1 进程管理	7
3.1.1 进程控制块	7
3.1.2 进程状态	7
3.1.3 分配进程	8
3.1.4 进程初始化	8
3.1.5 进程调度	10
3.1.6 进程释放	12
3.1.7 线程实现	13
3.2 内存管理	18
3.2.1 物理内存管理	18
3.2.2 虚拟内存管理	20
3.3 文件系统	27
3.3.1 总体架构与层次结构	27
3.3.2 关键功能实现	28
3.3.3 缓存管理与性能优化	29
3.4 net 功能	30
3.4.1 设计思想	30
3.4.2 模块协作说明	31
3.4.3 关键函数实现	32
3.4.4 socket 系统调用接口函数	33
<b>4. 系统调用的设计实现</b>	<b>36</b>
4.1 系统调用的流程	36

4.2 部分系统调用实现 .....	36
4.2.1 brk、sbrk 系统调用 .....	36
4.2.2 mmap 系统调用 .....	39
4.2.3 clone 系统调用 .....	43
4.2.4 wait4 系统调用 .....	48

# 1. 介绍

NoWaitOS 是一个基于 K210 开发板的 xv6 操作系统。它基于 MIT 开发的教学用简化版 UNIX 操作系统 xv6，并经过适配以支持 RISC-V 架构，特别是 Kendryte K210 芯片上的运行。

## 1.1 项目背景及意义

在合肥工业大学的计算机系统教育进程中，我们察觉到理论与实践的融合尚存在优化空间。操作系统课程理论性强且实践难度大，学生虽有探索热情，积极组建学习小组钻研兴趣领域，但在传统教学里，理论授课为主，实践环节不足，导致学生理解理论困难，实践技能锻炼有限。

为提升人才培养质量，我校计算机系统教育需在几方面改进。理论教学上，应优化课程与教学法，助学生构建扎实知识体系，深化理论理解；实践教学方面，需设计创新实验，让学生在实践中运用理论，增强实操与问题解决能力，培育创新思维；还要鼓励学生在导师引领下探索前沿，拓宽视野。

其中，设计创新实验是当务之急。我们打算在模拟器 QEMU 上开发操作系统，这能摆脱现有框架限制，为实验设计提供更多自由，加深学生对系统与硬件交互的理解，激发创新思维与探索精神。

我们期望此方式能发展成可持续教学模式，通过 Github 项目组、课程实验和计算机系统小组等推广实践。相信这对人才培养意义重大，可提升学生理论与实践水平，激发创新，培养解决问题能力，为我校计算机教育注入活力，推动其创新发展，培养更多优秀人才。

## 1.2 国内外研究状况

操作系统内核是操作系统的核心，负责硬件与软件的基础交互，涵盖内存管理、进程调度和文件系统等关键功能，其研究在国内外呈现出多维度、多层次的特点。

在商业系统领域，微软的 Windows 操作系统凭借其广泛的用户基础，成为全球个人电脑和服务器的市场的重要参与者。其内核 NT 采用混合内核架构，融

合了微内核和宏内核的优势，历经多年持续研发与优化，在性能、兼容性和安全性方面表现卓越，为用户提供了稳定高效的计算环境。苹果公司的 macOS 操作系统则基于 XNU 混合内核，通过不断改进 Mach 微内核和 BSD 宏内核的组合，为 Mac 系列产品打造了流畅、安全且功能丰富的用户体验，尤其在图形处理、多媒体应用和硬件整合方面独具特色。

开源系统方面，Linux 内核无疑是最为成功的案例之一。作为开源宏内核，Linux 吸引了全球众多开发者的积极参与，其代码贡献者遍布世界各地。Linux 内核以高度的可定制性、稳定性和安全性著称，广泛应用于服务器、超级计算机、嵌入式设备以及个人桌面等多个领域，成为服务器市场的主导力量，为互联网基础设施的稳定运行提供了坚实支撑。在国内，中科院软件研究所的可信鸿蒙操作系统和哈工大的麒麟操作系统均以 Linux 内核为基石，分别朝着高可信性和国产化方向发展。可信鸿蒙专注于网络安全环境下的应用，致力于为物联网设备提供安全可靠的操作系统解决方案；麒麟操作系统则紧密围绕国内自主可控需求，在政府、国防和关键基础设施等领域发挥着重要作用。

展望未来，操作系统内核研究呈现出多个引人注目的趋势。微内核研究虽在当前主流操作系统中占比相对较小，但因其有助于提升系统可靠性和安全性，通过将大部分功能移至用户空间，有效降低内核空间复杂性，依然吸引着众多研究者的目光。谷歌的 Fuchsia 操作系统采用 Zircon 微内核，便是这一趋势的有力例证。安全性研究随着网络安全形势日益严峻而愈发关键，内核自我保护、隔离和加固等技术成为研究热点，旨在确保操作系统免受各类恶意攻击。硬件技术的不断进步，如多核处理器、非易失性内存和量子计算机的出现，促使操作系统内核研究不断探索如何更好地利用这些新硬件特性，以实现更高的性能和效率提升。

总体而言，操作系统内核研究领域既涵盖了商业系统的成熟应用和持续创新，又包含开源系统的广泛协作与特色发展，同时紧跟未来技术趋势，为计算机系统的稳定、高效运行提供了坚实的核心支撑，推动着整个计算机科学领域不断向前发展。

## 1.3 项目的主要工作

NoWaitOS 项目主要致力于在多个关键方面对操作系统进行深度优化与功能

拓展，以适配 K210 芯片并满足多样化的应用需求。

在日志功能方面，我们深入剖析操作系统运行机制和数据流向后精心构建日志记录模块。该模块能够实时捕捉操作系统内核及应用程序运行时的关键事件和状态信息，像系统启动关键步骤、进程相关操作、内存使用情况以及设备驱动状态等。

对于系统调用，基于 K210 芯片硬件特性和操作系统架构设计，实现涵盖进程管理（如创建、销毁、挂起、恢复及进程间通信）、内存管理（分配、释放、映射、保护）、文件操作（创建、打开、读写、关闭、属性查询）、设备驱动（为各类外设开发接口以实现控制与通信）等多核心领域的丰富接口。通过这些系统调用，应用程序能便捷地与操作系统交互，实现高效资源利用和功能执行。

在完善操作系统基本功能上，进程调度引入多级反馈队列调度等先进算法，依据进程优先级、执行与等待时间等动态调度，保障进程公平高效获取 CPU 资源，提升系统整体性能。内存管理进一步优化分配策略，采用基于对象的技术减少内存碎片产生，提高内存利用率和系统稳定性。

此外，我们还积极尝试为操作系统添加网络功能，研究 K210 芯片网络相关硬件资源，探索合适的网络协议栈集成方案，期望实现如网络数据传输、通信连接建立与管理等基本网络功能，以拓展操作系统在网络应用场景中的适用性，为后续更复杂网络应用开发奠定基础。

## 2. 设计目标

在初赛阶段，我们计划 NoWaitOS 的实现主要集中在完成了 37 个系统调用，完成操作系统的基本功能。

除此之外，我们在原先 xv6-riscv 的基础上，增加对线程的支持。

另外，我们对 NoWaitOS 添加日志功能，以随时记录 x6 - k210OS 的各种操作及报错信息，以方便我们的调试。

并初步添加了网络代码，拟在后续完善网络部分代码以支持网络功能。

## 3. 系统设计与实现

### 3.1 进程管理

在 NoWaitOS 中，进程管理模块是操作系统的核心部分，负责进程的创建、初始化、调度以及资源回收。进程管理包括多个重要子模块，如进程控制块(PCB)、进程状态、进程分配、进程初始化、进程调度和进程释放等。下面我们逐一解析这些功能模块。

#### 3.1.1 进程控制块

在 NoWaitOS 中，进程控制块(PCB)是操作系统管理进程的基本单元。每个进程都对应一个进程控制块，PCB 中记录了进程的状态、程序计数器、堆栈指针、文件描述符、页表等信息，操作系统通过它来进行进程的调度、控制与资源管理。具体来说，进程控制块中的字段包括：进程状态、PID、父进程、虚拟内存地址(VMA)、CPU 上下文、内核时间、用户时间等。

#### 3.1.2 进程状态

进程的状态定义如下：

```
1. enum procs_state { UNUSED, SLEEPING, RUNNABLE, RUNNING, ZOMBIE };
```

- **UNUSED** 状态：表示进程控制块尚未关联任何进程。
- **SLEEPING** 状态：意味着进程因特定原因暂未运行。
- **RUNNABLE** 状态：表明进程正在等待调度器安排其投入运行。
- **RUNNING** 状态：说明进程当前正在运行。
- **ZOMBIE** 状态：指进程已被终止，但相关资源尚未回收。

### 3.1.3 分配进程

于 NoWaitOS 系统而言，新进程的分配工作由 `allocproc` 函数来完成。该函数的执行步骤如下：

寻觅未使用的进程控制块：系统会对预先设定好的进程控制块数组进行全面扫描，目的是找到一个状态为 `UNUSED` 的进程控制块。要是找到了，就跳转到名为 `found` 的标记处，接着进行下一步操作；要是没找到，那就返回 `NULL` 值，这表示当前系统中没有可用的进程控制块，无法创建新进程。

初始化进程控制块：一旦找到未被使用的进程控制块，函数会调用 `allocpid` 函数为其分配一个独一无二的进程 ID。同时，将虚拟内存地址（`vma`）初始化为空值 `NULL`，并且把内核时间（`ktime`）和用户时间（`utime`）都设定为 1。

分配 `trapframe` 页面：函数通过调用 `kalloc` 函数，为新进程的 `trapframe` 分配一块内存空间。要是内存分配操作未能成功，函数会释放进程锁，然后返回 `NULL`。

创建用户页表与内核页表：在成功分配 `trapframe` 页面之后，函数会分别调用 `proc_pagetable` 函数和 `proc_kpagetable` 函数，创建一个空的用户页表以及一个内核页表。只要其中任何一个页表创建失败，函数就会清理进程相关资源，随后返回 `NULL`。

设定进程内核堆栈：页表分配成功后，函数会为进程设定内核堆栈地址，这个地址是预先确定好的虚拟地址。

构建新上下文：函数会对进程的上下文（`context`）进行初始化操作，将返回地址寄存器（`ra`）设置为 `forkret` 函数的地址，而堆栈指针（`sp`）则设置在内核堆栈的顶部位置。如此一来，新进程就会从 `forkret` 函数处开始执行。

通过这一系列操作，`allocproc` 函数就完成了新进程的创建工作。每个新创建的进程在历经这些步骤后，都会拥有一个唯一的进程 ID、属于自己的页表和堆栈，为后续执行做好准备。

### 3.1.4 进程初始化

在 NoWaitOS 系统中，除了 `init` 进程之外，其余所有进程都是先通过 `fork`



系统调用进行复制，然后利用 `exec` 系统调用加载新程序来运行。

对于 `init` 进程，初始化过程包括：

进程分配：系统从预定义的进程控制块数组中寻找一个空闲的进程控制块，检查其状态是否为 `UNUSED`，表示该进程控制块未被使用。如果找到空闲块，系统为其分配唯一的进程 ID (`PID`)，并对其进行初始化，准备启动新的进程。

页表映射：系统为进程的各个段（如代码段、数据段、堆栈段等）创建虚拟内存映射。具体地，操作系统通过设置页表，将进程的虚拟地址空间映射到物理内存中。这一过程确保了进程能够安全、有效地访问其所需的内存区域，避免非法内存访问。

设置进程状态：在完成进程控制块的分配和页表映射之后，操作系统将进程的状态设置为 `RUNNABLE`，表示进程已经准备好被调度。进程进入可调度队列，等待操作系统的调度器分配 `CPU` 时间片，执行相应任务。

```
1. static struct proc *
2. allocproc(void)
3. {
4.     struct proc *p;
5.     for (p = proc; p < &proc[NPROC]; p++) {
6.         acquire(&p->lock);
7.         if (p->state == UNUSED) {
8.             goto found;
9.         } else {
10.            release(&p->lock);
11.        }
12.    }
13.    return NULL;
14. found:
15.    p->pid = allocpid();
16.    p->vma = NULL;
17.    p->ktime = 1;
```

```

18.     p->utime = 1;
19.     // 分配一个 trapframe 页面
20.     if ((p->trapframe = (struct trapframe *)kalloc()) == NULL) {
21.         release(&p->lock);
22.         return NULL;
23.     }
24.     // 为该进程创建一个空的用户页表和一个相同的内核页表
25.     if ((p->pagetable = proc_pagetable(p)) == NULL ||
26.         (p->kpagetable = proc_kpagetable()) == NULL) {
27.         freeproc(p);
28.         release(&p->lock);
29.         return NULL;
30.     }
31.     p->kstack = VKSTACK;
32.     // 设置新的上下文，从 forkret 开始执行，forkret 返回用户空间
33.     memset(&p->context, 0, sizeof(p->context));
34.     p->context.ra = (uint64)forkret;
35.     p->context.sp = p->kstack + PGSIZE;
36.     return p;
37. }

```

对于其他进程，fork 系统调用用于创建子进程，子进程会复制父进程的控制块和页表。随后，exec 系统调用会加载新程序并更新进程的虚拟内存映射，使新程序在原进程的环境中运行。

### 3.1.5 进程调度

在 NoWaitOS 操作系统中，进程调度采用的是 时间片轮转（Round Robin）算法，这是一种公平的调度策略，适用于多任务操作系统。在该算法中，调度器会周期性地中断当前正在运行的进程，随后从就绪队列中选择一个 RUNNABLE

状态的进程进行调度。所有进程的优先级均相同，因此调度顺序完全依赖于操作系统的时钟中断。在每个时间片内，进程将获得一个固定的执行时间，超时后将被强制中断，调度器会选择下一个进程来运行。

进程切换是操作系统核心功能之一，其目的是确保 CPU 能够在多个进程之间切换执行，实现多任务并发运行。进程切换的核心机制是上下文切换，即保存当前进程的执行状态，并加载下一个进程的执行状态。具体来说，在进程切换时，操作系统会进行以下几步操作：

保存当前进程的上下文：当前正在运行的进程会被挂起，操作系统首先保存该进程的上下文信息。这些上下文包括 CPU 寄存器的值、程序计数器（PC）、堆栈指针（SP）等关键数据，这些数据的保存确保了进程在下次被调度时能够从上次中断的地方继续执行。上下文保存通过调用 `swtch` 函数来实现，该函数会将当前进程的上下文数据保存到一个特定的内存区域。

恢复下一个进程的上下文：调度器从就绪队列中选择一个新的进程（通常是 `RUNNABLE` 状态的进程）来执行。操作系统随后恢复该进程的上下文，将相应的 CPU 寄存器值和程序计数器的值加载到 CPU 中。通过这些操作，CPU 能够无缝地切换到下一个进程，保证新进程从它暂停的地方继续执行。

切换进程的页表：每个进程都有独立的虚拟地址空间，操作系统会在进程切换时切换相应的页表，确保当前进程访问的是属于它自己的内存地址空间。这一切换是至关重要的，因为它能够避免不同进程之间的内存冲突和数据泄露。操作系统通过设置当前进程的页表为新进程的页表，从而隔离每个进程的内存空间，确保它们在运行时不会互相干扰。

通过这些步骤，操作系统能够实现平滑的进程切换，确保每个进程在得到 CPU 时间片后能够顺利执行。同时，操作系统也能够高效地管理多任务环境中的进程运行，避免某个进程长时间占用 CPU 导致系统响应迟缓。

在 NoWaitOS 中，CPU 由以下结构体进行管理：

```
1. struct cpu {  
2.     struct proc *proc; // 正在此 CPU 上运行的进程，若没有则为空  
3.     struct context context; // 执行 swtch() 在此处可进入调度器(scheduler)
```

```
4.     int noff; // push_off() 的嵌套深度
5.     int intena; // push_off() 之前中断是否启用};
6. };
```

其中，`proc` 和 `context` 字段尤为关键。`proc` 字段指向当前正在该 CPU 上执行的进程，而 `context` 字段则用于进程切换。`context` 的定义如下：

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

它的作用是保存当前进程的上下文，以便进程下次能够顺利恢复运行。NoWaitOS 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，该函数会遍历所有进程，查找状态为 `RUNNABLE` 的进程。

### 3.1.6 进程释放

在 NoWaitOS 操作系统中，当一个进程终止执行，或者在进程创建过程中发生错误时，操作系统需要对该进程所占用的各类资源进行有效的回收。为此，

系统通过 `freeproc` 函数来释放进程相关的资源。进程释放的主要步骤包括：

**释放 `trapframe` 页面：**`trapframe` 是一个结构体，负责保存进程的上下文信息（包括寄存器值、程序计数器等）。当进程结束时，操作系统首先会检查并释放与该进程相关联的 `trapframe` 页面。如果 `trapframe` 已经分配，操作系统会通过调用相应的内存释放函数将其回收，确保没有残留的上下文数据导致内存泄漏。

**释放内核页表 and 用户页表：**每个进程都有独立的页表，分别用于管理用户空间和内核空间的虚拟地址映射。为了防止内存泄漏，操作系统会在进程终止时释放进程的内核页表 and 用户页表。内核页表通常包含内核空间的虚拟地址映射，而用户页表则负责管理进程的用户空间。释放这些页表的内存是进程销毁过程中的关键一步。`proc_freepagetable` 函数会被调用来释放进程的用户页表，而内核页表则会通过其他机制进行回收。

**清理进程控制块（PCB）：**进程控制块（PCB）是操作系统管理进程的关键数据结构，包含了进程的各种元数据，例如进程 ID、父进程指针、虚拟内存区域、文件描述符等。在进程释放时，操作系统会逐一清空这些字段。例如，进程的进程 ID（`pid`）将被置为零，父进程指针（`parent`）将被清空，虚拟内存区域（`vma`）将被释放，进程的文件描述符（`fd`）也会被清理。这样可以确保进程控制块中的所有信息都被清除，避免对系统资源造成任何干扰。

**设置进程状态为 `UNUSED`：**在释放所有资源后，操作系统将进程的状态设置为 `UNUSED`，这意味着该进程控制块不再关联任何有效的进程，并且可以被系统重新分配给新的进程。设置进程为 `UNUSED` 状态是资源回收的最后一步，它标志着进程控制块已经被完全清理，并为新进程的创建腾出了空间。

通过 `freeproc` 函数，NoWaitOS 确保了在进程终止或出错时，所有相关资源能够被及时回收。这一过程不仅保证了操作系统的资源管理高效和清晰，还避免了内存泄漏和资源浪费，从而为新的进程创建提供了必要的资源支持。

### 3.1.7 线程实现

在 NoWaitOS 操作系统中，线程是进程内部的基本调度单位，每个进程可以拥有多个线程。操作系统提供了一些基本功能来管理线程的生命周期，包括线

程的创建、调度、同步以及销毁。

我们先看一下线程的数据结构。

```
1.     typedef struct thread {
2.         struct spinlock lock;
3.         // 当使用下面这些变量的时候，thread 的锁必须持有
4.         enum threadState state; // 线程的状态
5.         struct proc *p; // 这个线程属于哪一个进程
6.         void *chan; // 如果不为 NULL,则在 chan 地址上睡眠
7.         int tid; // 线程 ID
8.         uint64 awakeTime;
9.         // 使用下面这些变量的时候，thread 的锁不需要持有
10.        uint64 kstack; // 线程内核栈的地址,一个进程的不同线程所用的内核栈
            的地址应该不同
11.        uint64 vtf; // 线程的 trapframe 的虚拟地址
12.        uint64 sz; // 复制自进程的 sz
13.        struct trapframe *trapframe;
14.        context context; // 每个进程应该有自己的 context
15.        uint64 kstack_pa;
16.        uint64 clear_child_tid;
17.        struct thread *next_thread;
18.        struct thread *pre_thread;
19.        // TODO: signal
20.    }thread;
```

空闲线程和进程管理的线程都用双向链表来管理。对于每个进程，proc 结构体中有 thread\_queue 字段，表示管理线程中的第一个线程。每个线程有 pre\_thread 和 next\_thread 字段，分别用来指代链表中的前一个线程和后一个线程。在 thread.c 中定义有 free\_thread 作为空闲线程的链表头。

#### (1) 线程初始化 (threadInit)

线程系统的初始化通过 threadInit 函数完成。该函数主要负责初始化一个线

程控制块（TCB）数组，并为每个线程设置初始状态。具体步骤如下：

初始化每个线程的状态为 `t_UNUSED`，表示线程未被使用。

为线程链表设置前驱和后继指针，形成一个双向链表。这样能够更方便地管理线程的分配和回收。

设置 `free_thread` 为线程链表的头部，表示当前可用的线程。

```
1. void threadInit() {
2.     for (int i = 0; i < THREAD_NUM; i++) {
3.         threads[i].state = t_UNUSED;
4.         threads[i].pre_thread = NULL;
5.         threads[i].next_thread = NULL;
6.         if (i != 0) {
7.             threads[i].pre_thread = &threads[i - 1];
8.         }
9.         if (i + 1 < THREAD_NUM) {
10.            threads[i].next_thread = &threads[i + 1];
11.        }
12.    }
13.    free_thread = &threads[0];
14. }
```

## (2) 线程分配 (allocNewThread)

线程的创建和分配通过 `allocNewThread` 函数来完成。当操作系统需要创建一个新线程时，首先会检查 `free_thread` 是否为空。如果没有可用的线程控制块，则会触发异常（`panic`）。否则，操作系统会分配一个空闲的线程控制块，并为该线程分配一个 `trapframe`，这是线程上下文切换时需要保存的结构。具体步骤如下：

从链表中取出一个空闲线程控制块。

为线程分配一个 `trapframe` 页面，`trapframe` 用于保存线程的上下文。

将线程状态设置为 `t_RUNNABLE`，表示线程已准备好运行。

更新 `free_thread` 链表，移除已分配的线程控制块。

```

1.  thread *allocNewThread() {
2.      if (NULL == free_thread) {
3.          panic("allocNewThread: can not find unused thread");
4.      }
5.      if (NULL == (free_thread->trapframe = (struct trapframe
        *)kalloc())) {
6.          panic("allocNewThread: can not kalloc a page");
7.      }
8.      free_thread->awakeTime = 0;
9.      free_thread->state = t_RUNNABLE;
10.     free_thread->tid = nexttid++;
11.
12.     // 更新链表
13.     if (NULL != free_thread->next_thread) {
14.         free_thread->next_thread->pre_thread = NULL;
15.     }
16.     thread *tmp = free_thread;
17.     free_thread = free_thread->next_thread;
18.     tmp->next_thread = NULL;
19.     tmp->pre_thread = NULL;
20.
21.     return tmp;
22. }

```

### (3) 线程终止与回收

每个线程在完成任务后会进入终止状态，通常会由操作系统主动回收已结束的线程。线程的销毁过程涉及以下步骤：

更新线程的状态为 `t_ZOMBIE`，表示该线程已终止。

释放该线程占用的资源，包括内存和上下文。

重新加入空闲线程队列，以便将来复用。



#### (4) 线程调度

在 NoWaitOS 中，线程调度是通过进程调度器来进行的。当一个线程需要等待某个事件发生时（例如等待 I/O 操作），操作系统会将该线程的状态更新为 `t_SLEEPING`，并将其从可运行队列中移除。当该事件发生时，线程的状态会被更新为 `t_RUNNABLE`，重新加入可运行队列。

线程调度和进程调度类似，也是基于时间片轮转（Round Robin）算法。每当时间片用尽，操作系统会执行一次线程切换，保存当前线程的上下文并加载下一个可运行线程的上下文。

#### (5) 系统调用：sys\_tkill

`sys_tkill` 是一个系统调用，用于终止指定的线程。该系统调用接受两个参数：线程 ID (`tid`) 和信号号 (`sigum`)。目前，`sys_tkill` 函数的实现是一个简单的占位符，主要用于处理传入的参数并打印调试信息。

```
1. uint64 sys_tkill() {
2.     int tid;
3.     int sigum;
4.     if (argint(0, &tid) < 0 || argint(1, &sigum) < 0)
5.         return -1;
6.     debug_print("sys_tkill: tid = %d, sigum = %d\n", tid, sigum);
7.     return 0;
8. }
```

#### (6) 线程的同步与锁

为了确保多个线程在访问共享资源时的安全，操作系统为每个线程提供了 `lock` 锁。每个线程都有一个 `spinlock` 类型的锁，确保在修改线程的关键字段（例如状态、上下文）时，其他线程不能进行并发修改。操作系统在进行线程调度或资源释放时需要持有线程的锁，以避免数据竞争和不一致性。

## 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内存管理的基础，它负责计算机中的实际硬件内存的分配、回收和保护。由于物理内存是系统资源中的稀缺资源，高效且可靠的物理内存管理对于操作系统的性能至关重要。NoWaitOS 采用简单且高效的物理内存管理方案，通过页式管理和链表操作有效地管理物理内存。该系统的物理内存管理模块由 `kalloc.c` 负责，提供了初始化、分配和回收等基本功能。

#### (1) 物理内存分配和回收

在 NoWaitOS 中，物理内存的分配以页为单位，每个物理页面的大小为 4096 字节（即 4KB）。为了高效地管理这些页面，系统维护了一个空闲页面链表，链表中的每个节点代表一个空闲的物理页面。内存的分配和回收通过该链表来完成。具体来说，分配函数 `kalloc` 和回收函数 `kfree` 是物理内存管理的核心。

**kalloc:** 负责从空闲页面链表中分配一个页面。它首先尝试获取锁，确保分配操作的原子性。然后，从链表的头部取出一个空闲页面，并将该页面从链表中移除。最后，返回该物理页面的地址。为了提高调试效率，内核会将该页面内容填充为“垃圾值”，以帮助发现潜在的内存错误。如果链表为空，表示没有可用的内存，`kalloc` 返回 `NULL`。

**kfree:** 负责释放一个物理页面。它首先检查传入的地址是否合法（即是否是页对齐的地址），并且该地址是否在合法的物理内存范围内。然后，函数将该页面内容填充为“垃圾值”，以避免悬挂指针和数据泄露。接着，它将该页面添加回空闲链表中。为了确保在并发环境下的正确性，所有对空闲链表的操作都需要获取锁。

在系统启动时，`kinit` 函数会被调用，它负责初始化物理内存管理系统。`kinit` 首先初始化一个自旋锁，以保护内存分配和回收的原子性。然后，`freerange` 函数被调用，释放内核结束地址（`kernel_end`）到物理内存上限（`PHYSTOP`）之间的所有物理内存页，加入空闲页面链表中。这个过程确保了在系统启动时，所有可

用的物理内存页都能被有效地管理。

当进程终止时，其占用的物理内存页面会被回收。这些页面将被归还给空闲链表，以便其他进程可以使用。同样，如果内核栈溢出或发生其他异常，内核也会回收不再需要的物理内存。这种回收机制确保了内存不会泄漏，并且系统能够高效地使用物理内存。

## (2) 物理内存保护

物理内存保护是操作系统防止进程互相干扰的重要机制。在 NoWaitOS 中，物理内存的保护通过页表中的标志位实现。每个物理页面都对应一个页表项（PTE），PTE 中包含了一些标志位，用于控制该页面的访问权限。

这些标志位包括：

- PTE\_V (VALID)：指示该页面是否有效。如果该标志为 0，则该页面无效，访问该页面将引发页错误。
- PTE\_R (READ)：指示该页面是否可读。如果该标志为 0，则禁止读取该页面。
- PTE\_W (WRITE)：指示该页面是否可写。如果该标志为 0，则禁止写入该页面。
- PTE\_X (EXECUTE)：指示该页面是否可执行。如果该标志为 0，则禁止执行该页面中的代码。
- PTE\_U (USER)：指示该页面是否可以由用户模式程序访问。如果该标志为 0，则该页面仅对内核模式程序可访问。

这些标志位使操作系统能够精确控制每个物理页面的访问权限，从而保护内核和用户进程之间的内存隔离，防止非法访问以及内存越界。

## (3) 内存映射与直接映射

在 NoWaitOS 中，内核通过“直接映射”技术简化了对物理内存的访问。直接映射是指将物理内存区域直接映射到内核虚拟地址空间中，这意味着内核可以使用相同的虚拟地址来访问物理内存。在这种映射下，内核不需要每次访问物理内

存时都进行虚拟地址到物理地址的转换，从而提高了内存访问的效率。

内核的直接映射区域包括内核代码、堆栈以及其他重要的内核数据结构。通过这种方式，内核能够直接在虚拟地址空间中操作物理内存，提高了内存访问效率和内核的运行效率。

### 3.2.2 虚拟内存管理

虚拟内存是现代操作系统中至关重要的特性之一，它使得每个进程都拥有一个独立的、隔离的地址空间，并且不需要关心物理内存的分布。虚拟内存的核心目标是通过虚拟地址映射到物理地址，使得程序能够像访问连续内存一样进行操作，即使这些内存并非物理上连续。

在 RISC-V 架构下，NoWaitOS 操作系统采用了 Sv39 虚拟地址方案，利用三级页表结构来完成虚拟地址到物理地址的映射。每个进程都有自己的页表来描述该进程的虚拟地址空间，页表通过页表项（PTE）映射虚拟页到物理页。

#### (1) 页表结构

在 NoWaitOS 中，每个进程有独立的页表，通过三个层级的页表来实现虚拟地址到物理地址的转换。每个页表项包含物理页号和一些控制标志，如访问权限等。

在 RISC-V 的 Sv39 虚拟地址模式下，虚拟地址分为 39 位，其中高 9 位用于索引第一级页表，中间的 9 位用于第二级页表，低 9 位则用于第三级页表。每个页表项（PTE）包含一个指向下一级页表的物理地址，最终通过三级页表来获得对应的物理地址。

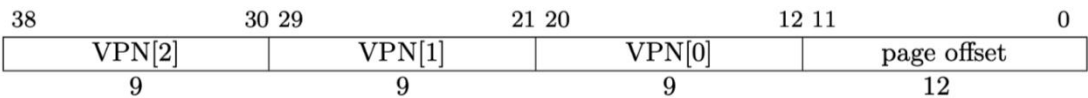


图 1 Sv39 虚拟地址，采用三级页表

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。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<sup>12</sup> 字节）的对齐块，也就是“页”，为单位的。

以下是一些与页表操作相关的常量和宏定义：

```
1. #define PTE_V (1L << 0) // valid
2. #define PTE_R (1L << 1) // read
3. #define PTE_W (1L << 2) // write
4. #define PTE_X (1L << 3) // execute
5. #define PTE_U (1L << 4) // user accessible
6. #define PTE_A (1L << 6) // accessed
7. #define PTE_D (1L << 7) // dirty
8. #define PA2PTE(pa) (((uint64_t)pa) >> 12) << 10)
9. #define PTE2PA(pte) (((pte) >> 10) << 12)
10 #define PTE_FLAGS(pte) ((pte) & 0x3FF)
```

这些宏和常量控制了页表项的权限和标志位。通过 PA2PTE 和 PTE2PA 宏，我们可以将物理地址和页表项互相转换，这对于页表的管理至关重要。

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<sup>39</sup> 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<sup>56</sup> 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。

## (2) 页表项 (PTE) 结构

一个页表项 (PTE) 包含多个字段，其中最关键的是物理页号和控制标志。物理页号 (PPN) 指向实际的物理内存页，而控制标志则用来限制对该页的访问。代码中通过定义 `pte_t` 和 `pagetable_t` 来表示页表项和页表：

```
1. typedef uint64 pte_t;  
2. typedef uint64 *pagetable_t; // 512 PTEs per page table
```

每个页表包含 512 个页表项，而每个页表项就是一个 `pte_t` 类型的值，包含虚拟地址映射到物理地址的相关信息。

## (3) 虚拟地址转换过程

虚拟地址转换为物理地址的过程通过三级页表进行。页表在物理内存中以一种三层的树状结构进行存储。这具体来说：

- 第一级页表：通过虚拟地址的高 9 位来索引，获取指向第二级页表的物理地址。
- 第二级页表：使用虚拟地址的中间 9 位来索引，获取指向第三级页表的物理地址。
- 第三级页表：通过虚拟地址的最低 9 位来获得物理页号 (PPN)，并将其与虚拟地址的低 12 位 (页内偏移) 组合成物理地址。
- 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)，NoWaitOS 将负责处理这一情况。

这段转换过程的代码片段如下：

```
1. #define PXMASK          0x1FF // 9 bits  
2. #define PXSHIFT(level) (PGSHIFT+(9*(level)))  
3. #define PX(level, va) (((uint64) (va)) >> PXSHIFT(level)) & PXMASK
```

在这段代码中，`PXSHIFT` 宏用于计算虚拟地址中不同级别页表的索引位置，`PX` 宏则根据虚拟地址和页表层级提取出对应的页表索引。

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，NoWaitOS 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  $3 \times 4096$ 。

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。

## (4) 内存管理函数

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。

例如，w\_satp 和 r\_satp 函数用于设置和读取 SATP 寄存器，SATP 寄存器用于保存当前的页表根地址：

```
1. static inline void w_satp(uint64 x)
2. {
3.     asm volatile("csrw satp, %0" : : "r" (x));
```

```
4. }
5. static inline uint64r_satp()
6. {
7.     uint64 x;
8.     asm volatile("csrr %0, satp" : "=r" (x) );
9.     return x;
10 }
```

`w_satp` 函数用于写入 SATP 寄存器，将当前的页表根地址写入该寄存器，而 `r_satp` 则用于读取该寄存器的值，获取当前使用的页表的根地址。

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。

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。`experm` 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。

## (5) 内核与用户地址空间的分离

NoWaitOS 采用了内核与用户地址空间分离的设计。内核的虚拟地址空间和用户的虚拟地址空间是独立的，通过页表映射来实现这种隔离。内核的虚拟地址空间通常是固定的，并且可以通过直接映射方式来访问物理内存。这种设计确保了进程无法访问内核空间，从而增强了系统的安全性。

内核空间的布局通常是预先定义的，并且通过特殊的映射方式来确保内核能够在虚拟地址空间中访问物理内存。例如，内核的虚拟地址空间从某一特定地址开始，并向上扩展，操作系统通过修改页表来保证内核空间的有效性。

## (6) 用户地址空间

在 NoWaitOS 中，每个用户进程都有独立的虚拟地址空间，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，如下图所示，进程的用户内存从虚拟地址 0 开始，可以增长到 MAXVA，从而允许进程在原则上寻址 256GB 的内存。

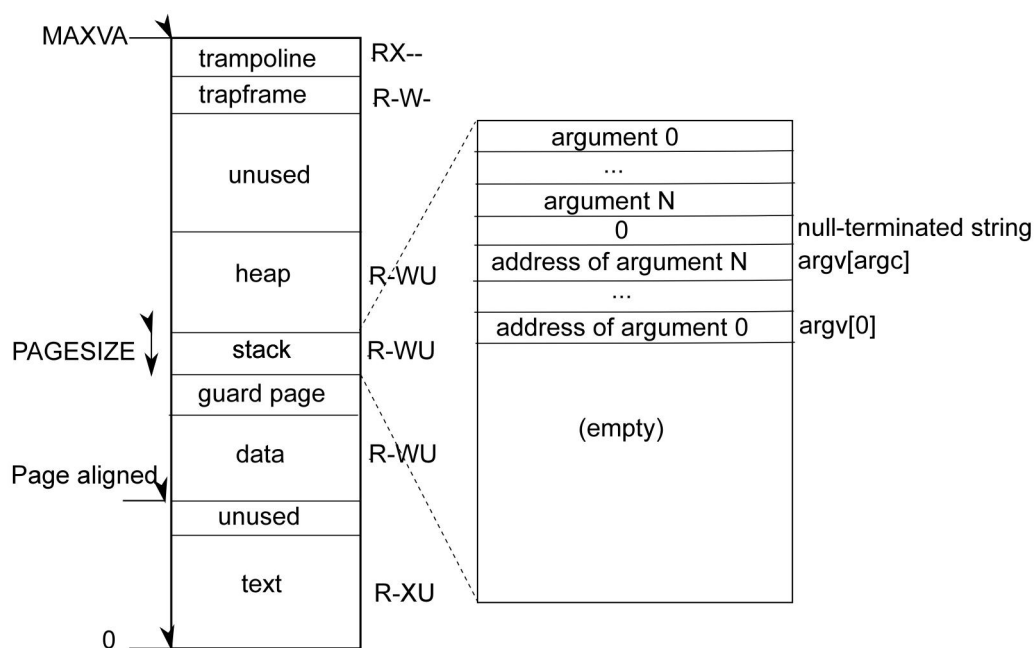


图 2 地址空间

进程的页表会映射虚拟地址到物理地址。当进程的虚拟地址空间需要更多内存时，NoWaitOS 首先使用 `kalloc` 分配物理页面。然后，它将页表项（PTE）添加到流程的页表中，该页表指向新的物理页。NoWaitOS 在这些页表项（PTE）中设置 `PTE_W`，`PTE_X`，`PTE_R`，`PTE_U` 和 `PTE_V` 标志位。大多数进程不会使用整个用户地址空间。对于未使用的页表项（PTE），NoWaitOS 将 `PTE_V`（VALID）标志位置 0。以下是与页表相关的函数：

```

1. static inline uint64r_sp()
2. {
3.     uint64 x;
4.     asm volatile("mv %0, sp" : "=r" (x) );
5.     return x;
6. }

```

这些函数用于读取和修改当前的堆栈指针，堆栈指针对于进程的虚拟地址空间管理至关重要。

## (7) 页表的更新与换页机制

当系统内存不足时，操作系统可能需要将某些页面从物理内存中移除并保存到磁盘上，这通常被称为换页机制。此时，操作系统需要确保页表中的页面状态得到更新，并且确保虚拟地址和物理地址之间的映射正确无误。NoWaitOS 使用 `sfence_vma` 函数来刷新 TLB（Translation Lookaside Buffer），确保页表变更后能够生效：

```
1. static inline void sfence_vma()
2. {
3.     asm volatile("sfence.vma");
4. }
```

`sfence_vma` 指令用于刷新 TLB 缓存，确保新的地址映射生效。

## 3.3 文件系统

### 3.3.1 总体架构与层次结构

基于 FAT32 文件系统，我们构建了一个层次化的文件系统架构，以实现高效的文件存储、检索和管理。该架构主要包括以下几个层次：

**磁盘驱动层：**负责与底层存储设备（如磁盘、SD 卡等）进行直接交互，实现物理扇区的读写操作。它提供了基本的磁盘初始化、扇区读取和写入功能，为上层文件系统提供了对存储介质的访问接口。

**缓存层（buf 管理）：**位于磁盘驱动层之上，引入了缓冲区缓存机制。通过 `struct buf` 结构体来管理缓存块，每个缓存块对应磁盘上的一个扇区。缓存层的主要作用是减少对磁盘的频繁读写操作，提高文件系统的性能。当上层文件系统需要读取或写入磁盘数据时，首先在缓存中查找是否已有相应的缓存块。如果命中缓存，则直接从缓存中获取数据，避免了磁盘 I/O 操作；如果未命中，则从磁盘读取数据到缓存块，并在缓存中管理这些数据块的状态（如有效位 `valid`、引用计数 `refcnt` 等）。缓存的替换策略采用常见的 LRU（最近最少使用）算法，通过双向链表结构（`prev` 和 `next` 指针）来维护缓存块的使用顺序，确保最近使

用过的缓存块能够被优先保留在缓存中，而最久未使用的缓存块在缓存空间不足时可以被替换出去。

**FAT32 文件系统核心层：**这是文件系统的核心部分，负责解析和管理 FAT32 文件系统的结构和数据。它基于磁盘扇区数据和缓存层提供的缓存块，实现了对 FAT 表（文件分配表）、根目录区和文件数据区的操作。通过 FAT 表来跟踪文件在磁盘上的存储位置（簇链），管理文件的分配和释放。在这个层次中，涉及到对 `struct dirent` 结构体的操作，用于表示目录项，记录文件或目录的各种属性（如文件名 `filename`、属性 `attribute`、创建时间 `ctime`、修改时间 `mtime`、访问时间 `atime`、文件大小 `file_size` 等）以及在磁盘上的存储位置信息（如起始簇 `first_clus`、簇数量 `clus_cnt` 等）。同时，通过对 FAT 表的操作来维护文件的簇链，实现文件数据的读写和存储管理。

**文件操作接口层：**为上层应用程序提供了一组统一的文件操作接口，如文件的打开、关闭、读写、创建、删除、目录操作等。这些接口基于 FAT32 文件系统核心层的功能实现，将底层的复杂操作封装起来，使得应用程序能够方便地进行文件系统相关的操作。在这个层次中，会涉及到对 `struct stat` 结构体的使用，用于获取文件的状态信息（如文件类型 `type`、大小 `size`、设备号 `dev` 等），以便应用程序能够了解文件的基本属性。

## 3.3.2 关键功能实现

### 文件和目录管理

**文件创建与删除：**当应用程序调用文件创建接口时，文件系统首先在目录区中查找是否存在同名文件。如果不存在，则在目录区中分配一个新的目录项（`struct dirent` 结构体），填写文件名、属性等相关信息，并在 FAT 表中为文件分配足够的簇空间来存储文件数据。对于文件删除操作，文件系统将释放文件占用的簇（通过在 FAT 表中标记簇为空闲），并从目录区中删除对应的目录项。

**目录操作：**支持目录的创建、删除和遍历功能。创建目录时，在目录区中创建一个特殊的目录项，标记其为目录属性，并在 FAT 表中为目录分配一定的簇空间来存储目录中的文件和子目录信息。目录遍历操作通过解析目录项结构，依

次获取目录中的文件和子目录信息，返回给应用程序进行处理。

## 文件读写操作

**读文件：**当应用程序调用读文件操作时，文件系统首先根据文件路径找到对应的目录项，获取文件的起始簇和大小等信息。然后，通过 FAT 表依次读取文件数据所在的簇内容，并将数据返回给应用程序。在读取过程中，利用缓存层来提高数据读取效率，如果文件数据已经在缓存中，则直接从缓存中获取；否则，从磁盘读取数据到缓存块，并更新缓存相关信息。

**写文件：**写文件操作相对复杂一些，涉及到文件数据的更新和文件大小的变化。如果是对已有文件进行写入操作，文件系统需要先根据写入位置和写入数据量确定需要修改的簇范围，然后在 FAT 表中找到相应的簇，将数据写入缓存块（如果缓存未命中，则先从磁盘读取相应簇到缓存），并标记缓存块为脏（dirty），表示需要在合适的时候写回磁盘。如果写入操作导致文件大小增加，文件系统还需要在 FAT 表中为文件分配额外的簇空间，并更新目录项中的文件大小信息。当缓存中的脏块达到一定数量或者系统执行同步操作时，缓存层会将脏块中的数据写回磁盘，确保数据的持久化存储。

## 文件属性操作

通过文件操作接口层，应用程序可以获取和修改文件的属性信息，如文件的创建时间、修改时间、访问时间、文件大小等。文件系统在内部维护这些属性信息，并在相应的系统调用（如 `stat`、`utime` 等）中更新和返回这些信息。例如，当文件被读取或写入时，文件系统会自动更新文件的访问时间和修改时间；当应用程序调用文件属性修改接口时，文件系统会相应地更新 `struct dirent` 和 `struct stat` 结构体中的相关字段。

### 3.3.3 缓存管理与性能优化

缓存层在文件系统性能优化中起着关键作用。除了采用 LRU 算法进行缓存替换外，还可以考虑以下优化措施：

**预读策略：**根据应用程序的文件访问模式，预测下一次可能读取的数据，并提前将其读取到缓存中。例如，当读取一个文件的某个簇时，可以预读相邻的几个簇到缓存中，因为在很多情况下，文件的连续簇内容可能会被顺序读取，这样可以减少后续读取操作的磁盘 I/O 延迟。

**延迟写回：**为了减少磁盘写操作的频率，缓存层可以采用延迟写回策略。即当缓存块被标记为脏后，并不立即将其写回磁盘，而是等待一段时间或者缓存中的脏块数量达到一定阈值时，再批量将脏块写回磁盘。这样可以将多次小的写操作合并为一次较大的写操作，提高磁盘 I/O 效率。

**缓存一致性维护：**在多线程或多进程环境下，需要确保缓存数据的一致性。当一个进程修改了缓存中的数据时，要确保其他进程能够正确感知到这些变化。可以采用锁机制（如 `struct sleeplock`）来保护缓存数据的并发访问，确保在同一时刻只有一个进程能够修改缓存中的特定数据块，避免数据冲突和不一致问题。

## 3.4 net 功能

### 3.4.1 设计思想

在操作系统内核 net 功能的设计中，融合了 `socket` 接口、`ringbuffer` 以及 `lwip` 等技术，构建了一个层次分明、高效可靠的网络通信架构。各部分紧密协作，共同实现了丰富的网络功能，同时满足了性能和灵活性的需求。

整体架构层次如下：

**应用层：**通过 `socket` 系统调用接口与应用程序进行交互，为应用提供创建套接字、绑定地址、监听连接、收发数据等功能。应用程序通过调用这些接口，能够方便地实现网络通信，而无需关心底层的实现细节。

**中间层：**包含 `socket` 核心实现以及与 `lwip` 的交互部分。`socket` 核心实现负责管理 `socket` 套接字的状态、连接、数据收发等操作，通过一系列的结构体和函数来维护和操作 `socket` 资源。同时，`lwip`（轻型 TCP/IP 协议栈）被引入到这个层次，提供了完整的 TCP/IP 协议栈功能，使得系统能够支持多种网络协议，如 IPv4、IPv6 等，以及不同类型的套接字通信（如 TCP、UDP）。

**底层：**由 `ringbuffer` 实现，负责在本地回环的情况下进行高效的信息传输。

当数据在本地进程间进行通信时，绕过了 qemu 的网卡，直接通过 ringbuffer 进行数据的读写，极大地提升了传输效率。

### 3.4.2 模块协作说明

**socket 接口：**作为应用层与系统网络功能之间的桥梁，socket 接口以一系列标准化的系统调用形式呈现，包括 `sys_socket`、`sys_bind`、`sys_listen` 等。这些接口函数犹如一个个精密的工具，将底层复杂的网络操作细节进行了高度封装，使得应用程序能够以一种统一、简洁且高效的方式开展网络编程工作。以 `sys_socket` 函数为例，它允许应用程序根据自身需求指定协议簇、socket 类型以及协议等关键参数，系统依据这些参数在底层创建对应的 socket 资源，并返回一个可供后续操作使用的文件描述符。通过这种方式，应用程序能够轻松地开启网络通信的大门，而无需关心底层网络协议栈的具体实现细节以及资源管理的复杂性。

**lwip：**lwip 作为一个轻量级的 TCP/IP 协议栈，在整个网络通信架构中扮演着核心协议支持的关键角色。它全面实现了网络层（如 IP 协议）、传输层（如 TCP、UDP 协议）以及部分应用层协议的功能。在系统运行过程中，lwip 与 socket 核心实现紧密协作，相互配合。当应用程序通过 socket 接口发起各种网络操作时，socket 核心实现会依据具体的操作类型，精准地调用 lwip 所提供的相应功能模块，以此来完成实际的网络通信任务。例如，在进行 TCP 连接的建立过程中，socket 核心首先会对应用程序传递的参数进行初步处理与校验，然后调用 lwip 实现的 TCP 协议栈功能，按照 TCP 协议的规范与流程，逐步完成三次握手等连接建立步骤，并在后续的数据传输过程中，协同 lwip 进行数据的有序收发与流量控制，直至连接的安全关闭。

**ring buffer：**在本地回环通信场景中，ring buffer 成为了数据传输的高效通道。当应用程序通过 socket 接口进行数据发送操作时，如果目标是本地进程，数据将会被直接写入到目的 socket 对应的 ring buffer 中；而在接收数据时，应用程序则从自身 socket 所关联的 ring buffer 中读取信息。这种基于 ring buffer 的本地数据传输方式，有效规避了传统网络通信中因涉及网卡驱动、网络协议栈多层处理等环节而带来的额外开销，显著提高了本地通信的效率与速度。同时，

ring buffer 的环形队列结构设计赋予了其独特的优势，通过合理的读写指针管理与缓冲区空间利用策略，有效避免了因缓冲区写满判断不准确而导致的假写满问题，只要写入的数据量在缓冲区空闲空间允许的范围内，就能够确保数据的安全写入与读取，不会出现越界错误，从而为本地回环通信提供了稳定可靠的数据传输保障。

### 3.4.3 关键函数实现

**in\_addr 结构体：**用于存储 socket 套接字的 IP 地址信息，其结构成员 s\_addr 承担着存储具体 IP 地址值的重要职责。

```
1. struct in_addr { in_addr_t s_addr; };
```

**sockaddr 结构体：**此结构体全面涵盖了 socket 套接字的关键信息，包括地址族、端口号、IP 地址以及填充字段等内容。其中，sin\_family 字段明确标识了地址族信息，sin\_port 字段用于存储端口号，sin\_addr 则是专门用于存储 IP 地址的结构体成员，而 sin\_zero 为填充字段，用于保证结构体的字节对齐等要求。

```
1. struct sockaddr {  
2.     sa_family_t sin_family;  
3.     in_port_t sin_port;  
4.     struct in_addr sin_addr;  
5.     uint8 sin_zero[8];  
6. };
```

**socket 结构体：**该结构体对一个 socket 套接字的完整信息进行了详尽描述，涵盖了协议簇、类型、所用协议、最大等待连接数、套接字号、监听等待名单、使用状态、地址信息以及用于收发信息的环形队列 ring\_buffer 等多个重要方面。例如，domain 成员记录了 socket 所遵循的协议簇信息，type 成员明确了 socket 的类型（如 SOCK\_STREAM 或 SOCK\_DGRAM），protocol 成员指定了具体使用的协议（如 IPPROTO\_TCP 或 IPPROTO\_UDP），backlog 成员设定了最大等待连接数，socknum 成员为套接字号，wait\_list 数组用于存储监听等待名单，used 成员表示是否被某个文件使用，status 成员记录了 socket 的当前状态（如



SOCK\_CLOSED、SOCK\_LISTEN 等），addr 成员存储了 IP 地址和端口信息，data 成员则是与数据收发紧密相关的环形队列 ring\_buffer。

```
1.      struct socket {
2.          int domain;                // 协议簇
3.          int type;                  // socket 类型
4.          int protocol;              // socket 所用协议
5.          int backlog;               // 最大等待连接数
6.          int socknum;
7.          uint8 wait_list[MAX_WAIT_LIST]; // 监听等待名单
8.          uint8 used;                 // 是否被某个文件使用
9.          uint8 status;               // socket 当前状态
10.         struct sockaddr addr;        // IP 地址和端口信息
11.         struct ring_buffer data;      // 用于收发信息的环形队列
        ring_buffer
12.     };
```

### 3.4.4 socket 系统调用接口函数

**sys\_socket (对应 do\_socket)：**此函数用于创建一个未绑定的 socket 套接字。应用程序在调用时需要指定协议簇 domain、socket 类型 type 和协议 protocol 等关键参数，系统根据这些参数在底层进行一系列复杂的资源分配与初始化操作，成功创建后将返回一个 socket 文件描述符，该描述符作为后续对该 socket 进行操作的唯一标识；若创建过程中出现错误，则返回相应的错误信息，以便应用程序进行错误处理。

```
1.  int do_socket(int domain, int type, int protocol);
```

**sys\_bind(对应 do\_bind)：**主要功能是将地址和端口绑定到指定的 socket 上。应用程序需要传入待操作 socket 的文件描述符 sockfd、包含绑定信息的 sockaddr 结构体地址 addr 以及该结构体的长度 addrlen。函数内部会对这些参

数进行合法性校验，并根据校验结果执行相应的绑定操作。若绑定成功，则返回 0；若出现错误，如地址无效、端口已被占用等情况，则返回对应的错误信息。

```
1. int do_bind(int sockfd, struct sockaddr *addr, socklen_t addrlen);
```

**sys\_listen（对应 do\_listen）：**用于使 socket 进入监听状态，等待客户端的连接请求。调用该函数时需要传入 socket 文件描述符 sockfd 和指明套接字侦听队列中半连接状态请求数最大值的 backlog 参数。函数会根据这些参数设置 socket 的监听参数，并启动监听机制。若监听设置成功，则返回 0；若出现错误，如参数非法、系统资源不足等情况，则返回错误信息。

```
1. int do_accept(int sockfd, struct sockaddr *addr, socklen_t
    *addrlen);
```

**sys\_accept（对应 do\_accept）：**从监听队列中取出第一个连接请求，创建一个与指定套接字具有相同套接字类型的地址族的新套接字，新套接字专门用于后续的数据传递工作，而原套接字则继续在监听队列中处理其他连接请求。如果侦听队列中当前无连接请求，该函数将阻塞等待，直到有连接请求到来。函数参数包括 socket 文件描述符 sockfd、用于保存客户端信息的 sockaddr 结构体地址 addr 以及保存客户端信息长度的地址 addrlen。若成功接受连接并创建新套接字，则返回新套接字的文件描述符；若出现错误，如监听队列已满、系统资源不足等情况，则返回错误信息。

```
1. int do_accept(int sockfd, struct sockaddr *addr, socklen_t
    *addrlen);
```

**sys\_connect（对应 do\_connect）：**客户端通过此函数在一个套接字上向服务器发起连接请求。传入参数包括 socket 文件描述符 sockfd、包含服务器地址和端口号的 sockaddr 结构体地址 addr 以及该结构体的长度 addrlen。函数内部会根据这些参数构建连接请求数据包，并按照网络协议规范向服务器发送连接请求，然后等待服务器响应。若连接成功建立，则返回 0；若出现错误，如服务器不可达、连接超时等情况，则返回错误信息。

```
1. int do_connect(int sockfd, struct sockaddr *addr, socklen_t
    addrlen);
```

**sys\_recvfrom (对应 do\_recvfrom)：**用于从 socket 接收消息。传入参数包括 socket 文件描述符 sockfd、待接收信息的首地址 buf、信息最大长度 length、信息接收类型 flags、发送者 socket 的 sockaddr 结构体指针 src\_addr 以及该结构体长度的地址 addrlen。函数会根据这些参数设置接收缓冲区和接收条件，然后等待接收数据。若接收成功，返回接收消息的字节数；若出现错误，如连接中断、接收超时等情况，则返回错误信息。

```
1.  ssize_t do_recvfrom(int sockfd, void *buf, size_t len, int
    flags, struct sockaddr *src_addr, socklen_t *addrlen);
```

**关闭 socket (对应 close\_socket)：**用于关闭指定的 socket 套接字，释放与之相关的所有系统资源，包括内存分配、文件描述符回收等操作。应用程序传入要关闭的 socket 套接字的编号 sock\_num，函数根据该编号找到对应的 socket 资源，并执行关闭和资源释放操作。若关闭成功，则返回 0；若出现错误，如参数非法、资源释放失败等情况，则返回错误信息。

```
1.      int close_socket(uint32 sock_num);
```

**sys\_setsockopt (对应 do\_setsockopt)：**该函数用于设置 socket 的各种选项，如设置超时时间、调整缓冲区大小等。应用程序需要传入 socket 文件描述符 sockfd、选项级别 level、选项名称 optname、指向选项值的指针 optval 以及选项值的长度 optlen。函数会根据这些参数对指定 socket 的相应选项进行设置操作。若设置成功，则返回 0；若出现错误，如参数非法、不支持的选项等情况，则返回错误信息。

```
1.  int do_setsockopt(int sockfd, int level, int optname, void *optval,
    socklen_t optlen);
```

**do\_lwip\_select：**此函数可能用于处理 lwip 相关的多路复用操作，例如同时监听多个 socket 的状态变化，当其中任何一个 socket 满足特定条件（如有数据可读、可写或出现异常情况）时，函数能够及时返回并通知应用程序进行相应处理。函数参数包括要监听的 socket 数量 socknum 以及超时时间结构体指针 timeout，通过合理设置超时时间，可以控制函数的阻塞等待时间。若在超时时间

内有 `socket` 满足条件，则返回相应的状态信息；若超时时间内无任何事件发生，则返回表示超时的信息。

```
1. int do_lwip_select(int socknum, struct timeval * timeout);
```

## 4. 系统调用的设计实现

### 4.1 系统调用的流程

在 NoWaitOS 操作系统里，系统调用是用户程序获取系统服务的关键方式。用户程序若需执行特权操作，如获取进程信息，会先将对应的系统调用号（如 `SYS_getpid`）存入 `a7` 寄存器，再执行 `ecall` 指令。

执行 `ecall` 后，RISC-V 硬件开始处理。它先检查是否为设备中断且中断关闭，若不是则关闭外部设备中断，接着拷贝 `pc` 值到 `sepc`，保存特权态信息到 `sstatus` 的 `SPP` 字段，设置 `scause` 为系统调用原因代码，将特权态设为 `S` 模式并把 `stvec` 值存入 `pc`，从而进入内核。

进入内核后，系统根据 `scause` 判断为系统调用，从 `a7` 获取调用号，再依此在系统调用表中找到对应函数指针（如 `sys_getpid`）并调用。函数执行完，内核存放返回值，然后执行 `sret` 指令。`sret` 会把 `sepc` 值写回 `pc`，更新 `sstatus` 的 `SIE` 字段并恢复特权态为 `U` 模式，之后用户程序就可从 `ecall` 下一条指令继续执行，获取系统调用结果。

### 4.2 部分系统调用实现

#### 4.2.1 `brk`、`sbrk` 系统调用

`brk` 和 `sbrk` 系统调用主要用于进程内存空间的动态调整。`brk` 系统调用允许用户程序指定一个新的数据段地址，从而扩展或收缩进程的数据段空间。`sbrk` 系统调用则依据传入的参数值，当参数为正时，分配新的地址空间给用户程序；当参数为负时，释放指定大小的地址空间。这两个系统调用为进程在运行时灵活管理内存提供了重要手段，使得进程能够根据实际需求动态调整其内存占用。

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 uvmalloc2 函数和 uvmdealloc2 函数来分配/释放物理内存。下面主要介绍一下 uvmalloc2 函数和 uvmdealloc2 函数的实现逻辑。

uvmalloc2 函数核心代码如下：

```
2.     uint64
3.     uvmalloc2(pagetable_t pagetable, uint64 start, uint64 end, int
    perm)
4.     {
5.         char *mem;
6.         uint64 a;
7.         if(start>=end)return -1;
8.         for(a = start; a < end; a += PGSIZE){
9.             mem = kalloc();
10.            if(mem == NULL){
11.                uvmdealloc2(pagetable, start, a);
12.                printf("uvmalloc kalloc failed\n");
13.                return -1;
14.            }
15.            memset(mem, 0, PGSIZE);
16.            if (mappages(pagetable, a, PGSIZE, (uint64)mem, perm) != 0)
17.            {
18.                kfree(mem);
19.                uvmdealloc2(pagetable, start, a);
20.                printf("[uvmalloc]map user page failed\n");
21.                return -1;
22.            }
23.        }
24.        return 0;
25.    }
```

该函数是实现内存分配的核心部分。首先，它检查起始地址 start 是否大于等于结束地址 end，若满足此条件，则直接返回 -1，表示参数错误。然后，通过一

个循环从起始地址 `start` 到结束地址 `end`, 以页大小 `PGSIZE` 为步长遍历地址空间。在每次循环中, 调用 `kalloc` 函数尝试从物理内存中分配一页内存。若分配失败(即 `mem == NULL`), 则调用 `uvmdealloc2` 函数释放此前已分配的内存(如果有), 并打印错误信息后返回 `-1`。若分配成功, 使用 `memset` 函数将新分配的内存页清零, 然后调用 `mappages` 函数创建页表项, 将虚拟地址 `a` 映射到物理地址 `mem`, 并设置相应的权限 `perm`。如果 `mappages` 函数执行失败, 释放已分配的内存页(通过 `kfree` 函数), 再次调用 `uvmdealloc2` 函数释放相关资源, 并打印错误信息后返回 `-1`。若整个循环过程顺利完成, 说明内存分配和映射成功, 返回 `0`。

`uvmdealloc2` 函数核心代码如下:

```
1.      uint64
2.      uvmdealloc2(pagetable_t pagetable, uint64 start, uint64 end)
3.      {
4.          if( start >= end )return -1;
5.          if(PGROUNDUP(start) < PGROUNDUP(end)){
6.              int npages = (PGROUNDUP(end) - PGROUNDUP(start)) / PGSIZE;
7.              vmunmap(pagetable, PGROUNDUP(start), npages, 1);
8.          }
9.          return 0;
10.     }
```

此函数用于释放内存空间。首先, 同样检查起始地址 `start` 是否大于等于结束地址 `end`, 若满足则返回 `-1`。接着, 判断 `PGROUNDUP(start)` 是否小于 `PGROUNDUP(end)`, 若满足, 则计算需要释放的页数 `npages`, 通过 `vmunmap` 函数释放从 `PGROUNDUP(start)` 开始的 `npages` 页内存, 其中参数 `1` 表示释放用户页表中的映射。最后, 若释放操作成功, 返回 `0`。

在 `brk` 和 `sbrk` 系统调用中, 通过调用 `uvmalloc2` 和 `uvmdealloc2` 函数, 与内存管理模块紧密协作。当进程调用 `brk` 或 `sbrk` 时, 系统根据传入的参数决定是扩展还是收缩内存空间。在扩展内存时, `uvmalloc2` 函数负责从物理内存分配页面、建立页表映射并设置权限; 在收缩内存时, `uvmdealloc2` 函数负责释放相应的页表映射和物理内存页面。这种交互确保了进程内存空间的动态调整能够正确、高效地进行, 同时与系统的内存管理机制保持一致, 维护了整个系统内存状态的正确性和稳定性。例如, 在一个多进程环境中, 当一个进程通过 `brk` 扩

展内存时，uvmalloc2 函数的操作不会影响其他进程的内存空间，保证了进程间内存的隔离性；而当进程通过 sbrk 释放内存时，释放的内存能够及时被系统回收，可供其他进程后续使用，提高了内存的利用率。

## 4.2.2 mmap 系统调用

mmap 系统调用在 NoWaitOS 操作系统中是一个非常重要的功能，它允许进程将文件或设备映射到其地址空间中，实现了高效的数据共享和内存管理。通过 mmap，进程可以像访问内存一样直接读写文件内容，无需进行繁琐的文件 I/O 操作，大大提高了数据访问效率。同时，它还支持多种映射方式，如私有映射和共享映射，以及不同的内存保护权限设置，为进程提供了灵活的内存管理手段。

以下是 mmap 的具体代码实现：

```
1.      uint64 mmap(uint64 start, uint64 len, int prot, int flags,
      int fd, long int offset) {
2.          struct proc *p = myproc();
3.          if (offset < 0) {
4.              return -1;
5.          }
6.          int perm = PTE_U | PTE_A | PTE_D | PTE_R | PTE_W | PTE_X;
7.
8.          if (prot & PROT_READ)
9.              perm |= PTE_R;
10.         if (prot & PROT_WRITE)
11.             perm |= PTE_W;
12.         if (prot & PROT_EXEC)
13.             perm |= (PTE_X | PTE_A);
14.
15.         struct file *f = fd == -1 ? NULL : p->ofile[fd];
16.         if (fd != -1 && f == NULL)
17.             return -1;
```

```
18.         struct vma *vma = allocate_mmap_vma(p, flags, start, len,
perm, fd, offset);
19.         if (!(flags & MAP_FIXED))
20.             start = vma->addr;
21.         if (NULL == vma) {
22.             return -1;
23.         }
24.         uint64 mmap_size = 0;
25.         if (-1 != fd) {
26.             mmap_size = f->ep->file_size - offset;
27.             if (len < mmap_size)
28.                 mmap_size = len;
29.             f->off = offset;
30.         } else {
31.             return start;
32.         }
33.         uint64 end_pagespace = mmap_size % PGSIZE;
34.         int page_n = PGROUNDUP(mmap_size) >> PGSHIFT;
35.         uint64 va = start;
36.         for (int i = 0; i < page_n; i++) {
37.             uint64 pa = experm(p->pagetable, va, perm);
38.             if (NULL == pa) {
39.                 return -1;
40.             }
41.             if (i != page_n - 1)
42.                 fileread(f, va, PGSIZE);
43.             else {
44.                 fileread(f, va, end_pagespace);
45.                 memset((void *) (pa + end_pagespace), 0, PGSIZE -
```



```
        end_pagespace);  
46.        }  
47.        va += PGSIZE;  
48.    }
```

**初始化：**函数首先获取当前进程的结构体指针 `p`，然后对传入的参数进行一系列严格检查。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_addr` 不是页面大小 `PGSIZE` 的整数倍，函数将直接返回 -1，表示参数错误。接着，初始化权限变量 `perm`，这里将 `perm` 初始化为 `PTE_U`，表示用户级别的权限，并根据注释部分代码（虽然注释部分可能存在一些问题，但从现有代码逻辑看是直接设置了读写等权限），将 `perm` 设置为包含读写等权限（`PTE_W | PTE_R | PTE_A | PTE_D`）。然后，根据文件描述符 `fd` 获取当前进程打开的文件结构体指针 `f`，如果 `fd` 不为 -1 且 `f` 为 `NULL`，则表示文件打开失败，函数返回 -1。

**分配虚拟内存区域（VMA）：**调用 `allocate_mmap_vma` 函数为当前进程分配一个 VMA 结构体，并传递相关参数（包括进程指针 `p`、标志 `flags`、起始地址 `start_addr`、长度 `len`、权限 `perm`、文件描述符 `fd` 和偏移量 `offset`）。如果分配失败（即 `vma == NULL`），函数返回 -1。分配成功后，更新起始地址 `start_addr` 为分配的 VMA 的起始地址 `vma->addr`。

**计算映射相关参数并进行内存映射：**如果文件描述符 `fd` 不等于 -1，表示关联了一个文件。此时计算 `mmap` 的大小 `mmap_size`，即映射文件的大小减去偏移量。如果传入的长度 `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`。接着计算 `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_addr`。通过一个循环，循环次数为页面数目 `page_n`，在每次循环中，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 == NULL`），函数返回 -1。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。

**文件引用计数增加与返回结果：**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\_addr，表示 mmap 映射成功。

接下来介绍一下 vma 结构：

```
1.      struct vma {
2.          enum segtype type;
3.          int perm;
4.          uint64 addr;
5.          uint64 sz;
6.          uint64 end;
7.          int flags;
8.          int fd;
9.          uint64 f_off;
10.         struct vma *prev;
11.         struct vma *next;
12.     };
```

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。

## 4.2.3 clone 系统调用

clone 系统调用用于创建新进程或线程，扩展了传统进程创建机制，为多线程编程和多任务处理提供灵活支持，能依据参数实现类似 fork 的进程创建或线程创建，提升系统资源利用率与程序执行效率。

以下是 sys\_clone 具体的代码实现：

```
1.      uint64 sys_clone(void) {
2.          uint64 new_stack, new_fn;
3.          uint64 ptid, tls, ctid;
4.          argaddr(1, &new_stack);
5.          if (argaddr(0, &new_fn) < 0) {
6.              return -1;
```

```

7.         }
8.         if (argaddr(2, &ptid) < 0) {
9.             return -1;
10.        }
11.        if (argaddr(3, &tls) < 0) {
12.            return -1;
13.        }
14.        if (argaddr(4, &ctid) < 0) {
15.            return -1;
16.        }
17.        if (new_stack == 0) {
18.            return fork();
19.        }
20.        if (new_fn & CLONE_VM)
21.            return thread_clone(new_stack, ptid, tls, ctid);
22.        else
23.            return clone(new_stack, new_fn);
24.    }

```

在 `sys_clone` 中，通过 `argaddr` 获取参数，若获取失败返回 -1。若 `new_stack` 为 0，调用 `fork` 创建进程；否则，根据 `new_fn` 是否设置 `CLONE_VM` 标志决定创建线程（调用 `thread_clone`）还是进程（调用 `clone`）。`thread_clone` 中，分配线程资源，进行栈空间映射与内存分配，复制栈参数，设置线程属性与上下文，处理 `ptid`，设置相关属性后返回线程 ID。`clone` 函数则分配进程资源，复制父进程内存，设置进程属性、执行上下文，最后返回进程 PID。

以下是 `thread_clone` 代码的实现：

```

1.    uint64 thread_clone(uint64 stackVa, uint64 ptid, uint64 tls,
2.        uint64 ctid) {
3.        struct proc *p = myproc();
4.        thread *t = allocNewThread();

```

```
4.         t->p = p;
5.         if (mappages(p->kpagetable, p->kstack - PGSIZE * p->thread_num
    * 2, PGSIZE,
6.             (uint64)(t->trapframe), PTE_R | PTE_W) < 0)
7.             panic("thread_clone: mappages");
8.         t->vtf = p->kstack - PGSIZE * p->thread_num * 2;
9.         void *kstack_pa = kalloc();
10.        if (NULL == kstack_pa)
11.            panic("thread_clone: kalloc kstack failed");
12.        if (mappages(p->kpagetable, p->kstack - PGSIZE * (1 +
    p->thread_num * 2),
13.            PGSIZE, (uint64)kstack_pa, PTE_R | PTE_W) < 0)
14.            panic("thread_clone: mappages");
15.        thread_stack_param tmp;
16.        if (copyin(p->pagetable, (char *)(&tmp), stackVa,
17.            sizeof(thread_stack_param)) < 0) {
18.            panic("copy in thread_stack_param failed");
19.        }
20.        t->kstack_pa = (uint64)kstack_pa;
21.        t->kstack = p->kstack - PGSIZE * (1 + p->thread_num * 2);
22.        t->next_thread = p->thread_queue;
23.        if (NULL != p->thread_queue)
24.            p->thread_queue->pre_thread = t;
25.        p->thread_queue = t;
26.
27.        copytrapframe(t->trapframe, p->trapframe);
28.        t->trapframe->a0 = tmp.func_point;
29.        t->trapframe->tp = tls;
30.        t->trapframe->kernel_sp =
```

```

31.         p->kstack = PGSIZE * (1 + p->thread_num * 2) + PGSIZE;
32.         t->trapframe->sp = stackVa;
33.         t->trapframe->epc = tmp.func_point;
34.         copycontext_from_trapframe(&t->context, t->trapframe);
35.         t->context.ra = (uint64)forkret;
36.         t->context.sp = t->trapframe->kernel_sp;
37.         if (ptid != 0) {
38.             if (either_copyout(1, ptid, (void *)&t->tid, sizeof(int)) <
0)
39.                 panic("thread_clone: either_copyout");
40.         }
41.         t->clear_child_tid = ctid;
42.         p->thread_num++;
43.         return t->tid;
44.     }

```

以下是 clone 代码实现:

```

1.     uint64 clone(uint64 new_stack, uint64 new_fn) {
2.         int i, pid;
3.         struct proc *np;
4.         struct proc *p = myproc();
5.         // Allocate process.
6.         if ((np = allocproc()) == NULL) {
7.             return -1;
8.         }
9.         // Copy user memory from parent to child.
10.        if (uvmcopy(p->pagetable, np->pagetable, np->kpagetable, p->sz)
< 0) {
11.            freeproc(np);

```

```
12.     release(&np->lock);
13.     return -1;
14. }
15. np->sz = p->sz;
16.
17. np->parent = p;
18.
19. // copy tracing mask from parent.
20. np->tmask = p->tmask;
21.
22. // copy saved user registers.
23. *(np->trapframe) = *(p->trapframe);
24.
25. // Cause fork to return 0 in the child.
26. np->trapframe->a0 = 0;
27.
28. // increment reference counts on open file descriptors.
29. for (i = 0; i < NOFILE; i++)
30.     if (p->ofile[i])
31.         np->ofile[i] = filedup(p->ofile[i]);
32. np->cwd = edup(p->cwd);
33.
34. safestrcpy(np->name, p->name, sizeof(p->name));
35.
36. pid = np->pid;
37.
38. np->state = RUNNABLE;
39.
40. np->trapframe->epc = new_fn;
```

```
41.     np->trapframe->sp = new_stack;
42.
43.     release(&np->lock);
44.
45.     return pid;
46. }
```

## 4.2.4 wait4 系统调用

`wait4` 用于父进程等待指定子进程终止并获取其状态。相比 `wait` 更灵活，可指定子进程 ID、获取状态地址及多种等待选项（当前部分未实现），适用于多进程协作场景下的同步需求。

以下是 `wait4` 功能实现：

```
1.     int wait4pid(int pid, uint64 addr, int options) {
2.         struct proc *p = myproc(), *child = NULL, *tchild = NULL;
3.         int kidpid;
4.         acquire(&p->lock);
5.         while (1) {
6.             kidpid = pid;
7.             child = findchild(p, pid, &tchild);
8.             if (NULL != child) {
9.                 kidpid = child->pid;
10.                child->xstate <= 8;
11.                if (addr != 0 && copyout(p->pagetable, addr, (char
                *)&child->xstate,
12.                    sizeof(child->xstate)) < 0) {
13.                    release(&child->lock);
14.                    release(&p->lock);
15.                    return -1;
16.                }
```



```

17.         freeproc(child);
18.         release(&child->lock);
19.         release(&p->lock);
20.         return kidpid;
21.     }
22.     if (!tchild) {
23.         release(&p->lock);
24.         return -1;
25.     }
26.     if (pid == -1) {
27.         sleep(p, &p->lock);
28.     } else {
29.         // printf("arrive here: tchild: %d\n", tchild->pid);
30.         sleep(tchild, &p->lock);
31.         // pay attention!
32.     }
33. }
34. release(&p->lock);
35. return 0;
36.
37. // TODO: deal with options
38. }

```

先获取父进程指针及锁，进入循环。确定要等待的子进程 ID，调用 `findchild` 查找。若找到，更新 `kidpid`，若需获取状态信息且复制成功，回收子进程资源并返回其 ID；若未找到，根据等待选项和条件处理，如 `WNOHANG` 选项（未完全实现）、是否有兄弟进程等，或睡眠等待子进程结束唤醒。

以下是 `findchild` 函数的实现：

```

1.     struct proc *findchild(struct proc *p, int pid, struct proc
    **child) {

```

```
2.      *child = NULL;
3.      // iterator all process
4.      for (struct proc *np = proc; np < &proc[NPROC]; np++) {
5.          if ((pid == -1 || np->pid == pid) && np->parent == p) {
6.              acquire(&np->lock);
7.              *child = np;
8.              if (np->state == ZOMBIE) {
9.                  return np;
10.             }
11.             release(&np->lock);
12.         }
13.     }
14.     return NULL;
15. }
```

依赖 `findchild` 与进程管理模块交互查找子进程；通过 `copyout` 与内存管理模块交互获取子进程状态信息；调用 `sleep` 与进程调度模块交互实现睡眠等待，涉及多模块协作。