南開大學
Nankai University

功能挑战赛 (初赛)

proj309：构建基于 **RDMA** 的多节点异质内存系统

开发设计文档

成员：曹骜天 朱奕翔 许宸

学校：南开大学

指导教师：宫晓利

2024 年 5 月 27 日

目录

1 目标描述

RDMA 是一种新型网络技术，CPU 可以通过 Infiniband RDMA 网卡设备对连接的另一个设备上的内存发起访问。我们希望通过 RDMA 网卡、RDMA 交换机构建一个 3-5 节点的原型系统。该系统应拥有 2-3 个远程内存节点和 1-2 个计算节点，我们希望可以在 Linux 内核层面，探索未来如何在更复杂 RDMA 拓扑连接上实现更高性能的内存池系统。

1. 基于 >6.1 版本的 linux 发行版开发
2. 组成实际的原型系统
3. 可运行在 86 架构上
4. 性能开销、稳定性高

基于以上目标实现的过程暂时分为三步：

1. 在内核上完成基于 Swap 子系统 frontswap 接口的 RDMA 页面置换机制。
2. 尝试适配 2 个远程内存设备
3. 在远程内存设备性能出现差异时使用调度手段优化

以上内容是题目要求，但是基于项目进度在实现了以上内容之后则需要对更多的设备进行测试，测试在更多的设备加入之后对性能的影响以及优化手段。

2 比赛题目分析与相关资料调研

在本章中，我们将对比赛题目进行分析，并结合我们查找的相关资料阐述我们的调研结果

2.1 比赛题目分析

题目要求开发一个基于 RDMA 技术的高性能内存池系统，具体包括以下几点：

- 使用 6.1 版本以上的 Linux 发行版进行开发。
- 构建一个由 2-3 个节点组成的原型系统，其中包括 1-2 个远程内存节点和 1 个计算节点。
- 系统需运行在 x86 架构上。
- 确保系统性能开销低且稳定性高。

2.1.1 实现过程分步

根据题目要求，系统实现过程可以分为以下三步：

1. 在内核上实现基于 Swap 子系统 frontswap 接口的 RDMA 页面置换机制。
2. 适配 2 个远程内存设备。
3. 在远程内存设备性能出现差异时使用调度手段进行优化。

2.2 相关资料调研

2.2.1 论文调研

Amaro 等人在 2020 年提出了 *Fastswap*[1]，*Fastswap* 是一种优化的 Linux 交换系统，通过以下方式实现了 RDMA 页面置换：

- **前置交换接口（Frontswap）**：用于页面粒度的交换，减少上下文切换，提高性能。
- **多队列机制**：使用两个队列，一个用于关键操作，一个用于预取操作，避免头阻塞。
- **轮询机制**：关键操作完成时使用轮询机制，减少延迟，提升页面获取速率。
- **内存回收**：将内存回收操作卸载到专用 CPU，减少页面故障处理路径上的延迟。

Memliner 由 Wang 等人在 2022 年提出。*Memliner* 基于 *fastswap* 实现，在 linux 5.4 内核上进行开发（*fastswap* 在 linux 4.1 内核上进行开发）[2]，其中，fastswap 基于 RDMA 实现了远程内存。

2.2.2 RDMA 的背景

RDMA 允许一台机器直接访问另一台机器的内存，而无需通过操作系统的干预，这可以显著减少延迟并提高数据传输效率。在分布式系统或云计算环境中，RDMA 常用于实现高性能和低延迟的内存访问。

在 Linux 系统中，可以通过修改内核，例如在内核中加入前面所述的 Fastswap 优化使用远程内存的方式。Fastswap 通过 RDMA 优化了页面交换（swapping）机制，提高了远程页面访问的速度，并通过多种技术减少了页面错误处理的延迟。这包括使用两个 RDMA 队列对不同类型的交换操作进行区分，以及优化页面预取和页面错误处理。

页面错误处理器是操作系统中处理内存页面错误的组件。当一个程序访问的内存页面不在物理内存中时，就会发生页面错误，此时页面错误处理器负责从磁盘或其他存储位置加载缺失的页面到物理内存中。

在使用 RDMA 和远程内存的环境下，页面错误处理器的工作也需要相应的调整。例如，在 Fastswap 系统中，页面错误处理器被优化以通过 RDMA 快速从远程内存中获取所需的页面。这包括对 RDMA 操作的轮询优化和对页面预取逻辑的改进，以减少对 CPU 的中断，并提高处理速度。

内存回收（Memory Reclaim）

内存回收是操作系统管理物理内存的一个重要方面。当系统的可用物理内存不足时，内存回收机制会被触发，其任务是回收那些不再被活跃使用的内存页面，以便这些页面可以被重新用于其他目的。

在 RDMA 和远程内存的使用场景中，内存回收同样需要特别的处理。例如，Fastswap 中引入了一种机制，将内存回收任务从页面错误处理的关键路径中分离出来，并通过一个专门的 CPU 进行处理。这种方法称为“卸载式回收”（offloaded reclaim），它可以显著降低页面错误处理的延迟，从而提高系统性能。

2.3 *Fastswap*

2.3.1 *Fastswap* 结构与调度策略

Fastswap 的重要结构如下：

1. RDMA 后端（RDMA Backend）

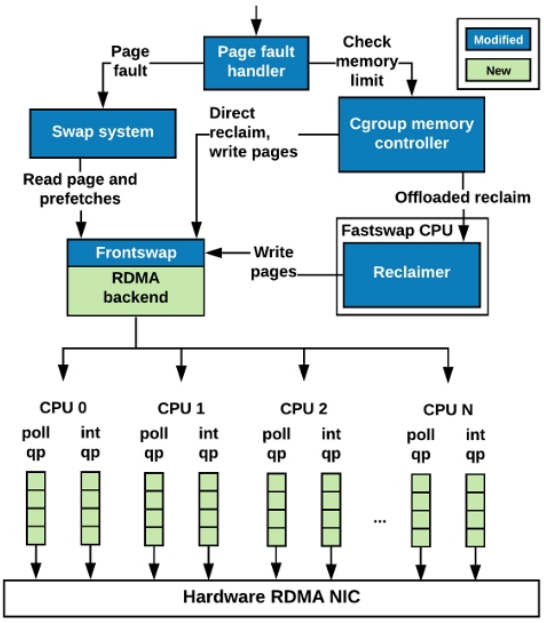


图 2.1: *Fastswap* 结构图

在 Fastswap 中，RDMA 后端是连接硬件 RDMA 网络接口卡（NIC）的主要组件。它主要负责处理来自 CPU 的所有页面交换操作，这些操作包括处理页面错误、页面预取和页面回写。Fastswap 使用两个 RDMA 队列来优化性能：

- 队列 1：处理页面错误读取的紧急 RDMA 操作。
- 队列 2：处理预取等非紧急 RDMA 操作。

这种分离确保了紧急操作可以快速完成，而不会因为其他较少紧急的操作而延迟。

2. 页面错误处理器（Page Fault Handler）

当程序访问的内存页不在物理内存中时，操作系统会通过页面错误处理器来处理这一情况。Fastswap 对传统的页面错误处理进行了优化，以减少处理时间和提高效率：

- 直接内存回收：当触发页面错误时，如果所需的页面不在物理内存中，则 Fastswap 会直接从远程内存通过 RDMA 读取该页面到本地内存中。
- 预读取和写回页：同时，Fastswap 会预读可能很快就会被访问的页面，并将修改过的页面写回到远程内存，以保证数据的一致性和减少未来的页面错误。

3. 内存回收（Memory Reclaimer）

内存回收是操作系统在物理内存不足时用来释放不再需要的内存页的机制。Fastswap 实现了以下几点优化：

- 卸载式回收（Offloaded Reclaim）：Fastswap 通过一个专门的 CPU（Reclaimer CPU）来处理内存回收任务，而不是在处理页面错误的关键路径上进行，这样可以显著降低页面错误处理的延迟。
- 检查内存限制（Check Memory-Limit）：此部分负责监控和管理内存使用，确保不会超过预设的内存限制，从而避免系统过载。

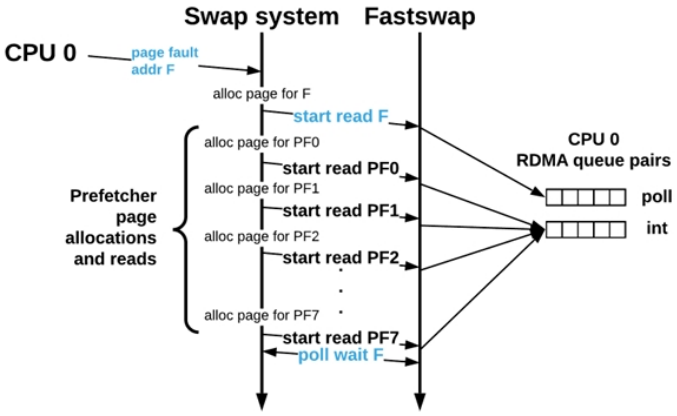


图 2.2: *Fastswap* 的页面置换策略

任务适应性调度（Far Memory-Aware Scheduling）：Fastswap 设计了一个远程内存意识的集群调度器，这个调度器可以根据可用的本地内存和远程内存来优化作业的放置和内存分配。当新作业到达时，调度器会评估当前服务器是否有足够的本地内存来支持这个作业。如果不足，调度器会尝试使用远程内存来满足需求。这种调度方式允许在内存资源受限但 CPU 资源充足的节点上运行更多的作业，从而提高资源利用率和作业吞吐量。

内存重平衡策略（Memory Rebalancing）：在作业运行期间，如果某个节点的本地内存需求发生变化，Fastswap 的调度器会动态调整作业的内存分配，优先考虑本地内存的使用，必要时使用远程内存。通过动态调整内存分配，Fastswap 可以在不同作业之间有效地分配内存资源，尽可能地减少远程内存访问的性能损耗。

使用多个队列来减少阻塞（Queue Pair Utilization）： Fastswap 利用两个 RDMA 队列来区分不同优先级的内存访问请求。一个队列用于处理关键路径上的访问（如页面错误处理），另一个队列处理预取等非关键路径的访问。这种方法可以减少关键请求的等待时间，提高页面交换的效率。

2.3.2 *Fastswap* 的局限性

对单一远程内存服务器的依赖：Fastswap 的实现目前只支持连接单个远程内存服务器，这在多远程内存服务器场景下限制了其扩展性和灵活性。当需要从多个远程内存服务器获取数据时，无法同时进行，可能会增加数据访问的复杂度和延迟。

调度算法的复杂性和开销：Fastswap 的调度算法需要实时监控作业的内存使用情况，这可能会增加调度器的运行开销。动态内存重平衡策略虽然可以提高内存利用率，但也需要更复杂的管理策略和算法支持，可能导致执行效率的下降。

不支持多内存服务器的灵活性：在大规模分布式系统中，可能需要从多个位置动态调配内存资源。Fastswap 目前的实现限制了这种灵活性，可能无法有效支持大规模的内存需求变化。

2.4 MemLiner 的基本思路

MemLiner 的基本思路是：在应用程序线程访问对象后，让 GC 立即并发地跟踪对象，并基于优先级算法确定 GC 跟踪对象的顺序，减少对 Application 的干扰。MemLiner 的实现方案是对对象进行先分类后处理。

对象分类

- **Local Objects**：正在被应用程序访问的对象。GC 线程应该立刻追踪这些对象，因为此时内存访问与应用程序一致。
- **Incoming Objects**：即将被应用程序访问的远程内存对象。GC 线程应该立刻追踪这些对象，相当于为应用程序预支了远程内存访问开销。
- **Distant Objects**：不会马上被应用程序使用的远程内存对象。GC 线程应该延后此类对象的追踪。

挑战和解决方案

MemLiner 在分类内存时需要解决以下挑战：

1. 如何通知 GC 对象正在被应用程序访问?
2. 如何判断对象是否即将被应用访问?
3. 如何判定对象是在本地还是远端?

为了应对这些挑战，MemLiner 采取了以下措施：

- MemLiner 维护一个 per-thread queue。当应用程序访问 local objects 后，将地址推入队列，GC 线程通过周期性检查并用原子指令读取队列中的对象地址，从而解决了如何通知 GC 对象正在被应用程序访问的问题。
- 高级语言中的对象一般通过引用访问，而即将被应用访问的对象一般能够在几个引用之内被定位到。MemLiner 基于这点识别 incoming objects，从而解决了如何判断对象是否即将被应用访问的问题。
- MemLiner 在对象引用中新增了 timestamp 字段，用于记录对象上次被访问的 epoch。通过将对象的 timestamp 字段和 OS 内核维护的全局 epoch 计数器比较，如果它们相差超过一定阈值，即可认为对象在远程，从而解决了如何判定对象是在本地还是远端的问题。

2.5 比赛题目思路

我们的工作基于 **linux 6.1 内核**和 [1] 对 Fastswap 相关的部分进行开发，并参考了 [2] 的实现过程。

基于 Fastswap 具有没有适配多个远内存的缺陷，我们将在后续进行创新开发，使其能够同时适配/连接多种远内存。

3 系统框架设计

我们最后的工作是尝试实现 3-5 节点之间的拓扑连接，以探索高性能内存池的构建。在此阶段并没有明确的框架，给出图3.3作为可能的参考：其中有三个内存节点，上方的两个仅作为内存节点使用；存在一个中间节点，两个计算节点通过 RDMA BACKEND 从中间节点中获取内存，而中间节点将自身的内存以及两个远程节点的内存提供给计算节点。在这个过程中我们可以限制两个远程节点的 RDMA 带宽不同来探索性能问题。

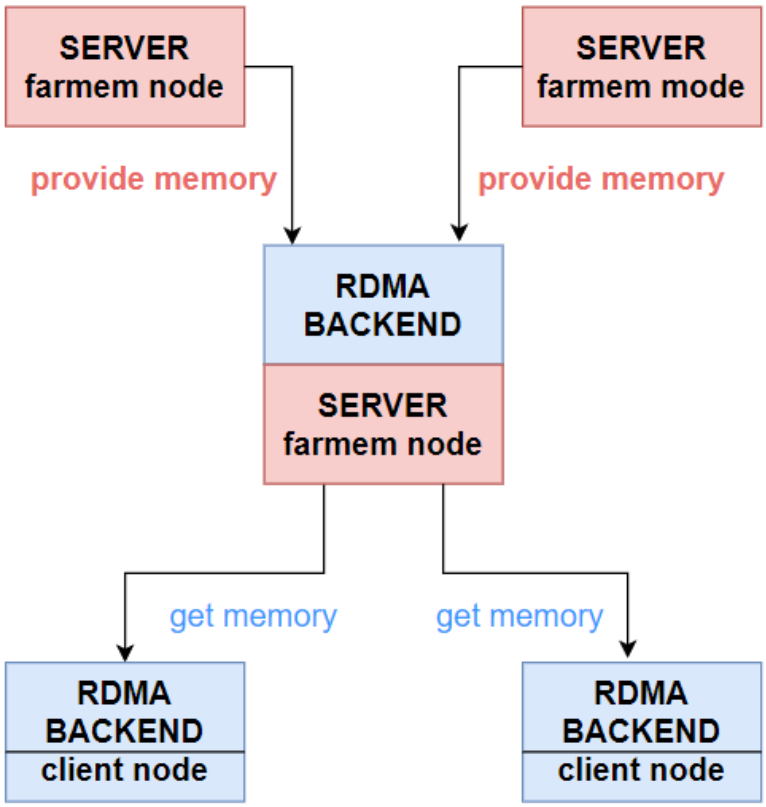


图 3.3: 系统框架

4 项目环境与开发计划

4.1 项目环境

我们对 server node(远程内存节点) 和 client node(计算节点) 采用不同的 Linux 版本进行开发，配置如下：

对于 client node: 采用 Linux6.1 的发行版内核以及 Ubuntu22.04 操作系统进行开发。采用的 MLNX OFED driver 版本为 MLNX_OFED_LINUX-24.04-0.6.6.0-ubuntu22.04-x86_64，运行架构为 x86。

对于 server node: 采用 Linux5.4 的发行版内核以及 Ubuntu20.04 操作系统进行开发。采用的 MLNX OFED driver 版本为 MLNX_OFED_LINUX-5.8-4.1.5.0-ubuntu20.04-x86_64, 运行架构为 x86。

对于所有的节点，使用的网卡均为：NVIDIA Mellanox ConnectX-5 Ethernet

4.2 开发计划

我们的开发计划大致如下：

1. 熟悉 RDMA 通信原理以及基本流程。(已实现)

2. 进行 MLNX OFED 网卡驱动的安装并给网卡分配相应 IP 地址，尝试在两台机器间能够通过 RDMA 网卡收发信息。(已实现)

3. 在 client node 修改内核并编写 module：在 Linux 的 frontswap 接口中短路拦截 swap_readpage 和 swap_writepage，采用 frontswap_load 和 frontswap_store 操作处理换入换出的页面，并用本机的 DRAM 来模拟所实现的 frontswap 的正确性。(已实现)

4. 在 client node 和 server node 之间建立 RDMA queue pairs，实现两个节点之间连接的建立。(已实现)

5. 在 client node 实现以页为颗粒度的与远程内存之间的换入换出：在本机测试通过的 DRAM 接口的基础上编写 RDMA 接口，能够通过 RDMA 独有的 read 和 write 操作实现远端 CPU 无感知的页面换入换出过程。(已实现)

6. 指定程序进行内存压力测试以强制调用 swap 操作，验证所实现接口的正确性，成功实现 client node 与 server node 一**对**一的基于 RDMA 的远程内存。(已实现)

7. 测试 server node 在本机上跑高内存带宽 workload 时，是否对 client node 访问 server 上的访存带宽有影响; 基于 a-b-c 连接方式 a 从 c 获取页面是否有性能损失, 其中 b 是 a 直连的远程内存，c 是 b 直连的远程内存。

8. 在一对一节点的基础上对 frontswap 接口进行修改，使 client node 能够适配多个 server node，并使得多个 far memory 之间负载均衡。

9. 对不同的 server node 限制不同带宽，探索在不同带宽下的异构 far memory 的访存性能影响。

5 项目实现

5.1 驱动安装与环境配置

在初赛阶段，我们先尝试配置两台机器，一台当作 server 端提供远程内存，一台当作 client 端。在 MLNX OFED 驱动安装的过程中，我们遇到了若干麻烦：某些驱动版本在 Ubuntu 系统上不支持对应版本的内核，经过测试选取了如第四节所描述的驱动版本，在官网下载压缩包后解压后进入目录，运行脚本并强制加入 DKMS (Dynamic Kernel Module Support)：

安装驱动

```
sudo ./mlnxofedinstall --force-dkms
/etc/init.d/openibd restart
```

这个第一条命令通常需要等待较长时间，在执行完后可以在终端中输入 ibstat 查看设备状态，我们显示结果如下图5.4，表示我们使用的是双口网卡，其中一个网口保持为启用并保持连接状态，HCAid 为 mlx5_1：

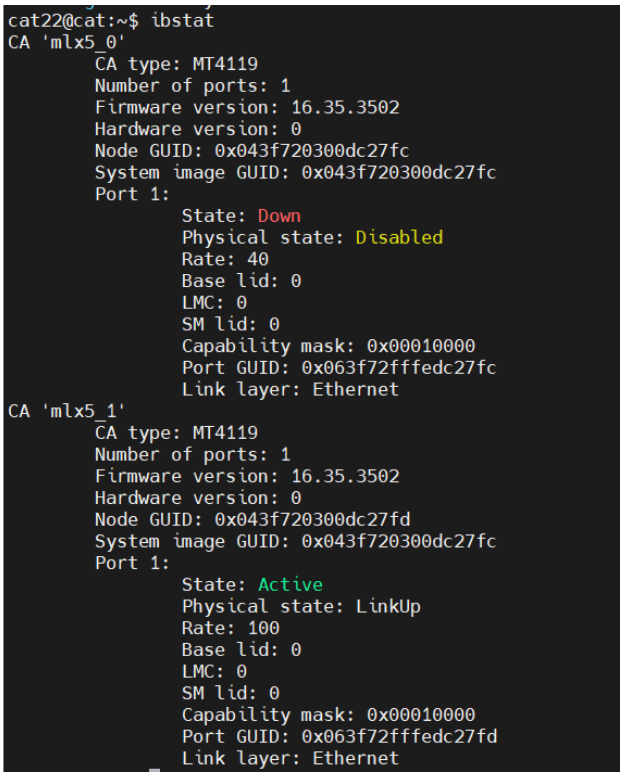


图 5.4: ibstat

接下来，为了使两台机器间建立连接，首先要给 ib 网卡配置 ip 地址，让其能够像其它的网卡一样工作。首先我们需要查看设备对应的网口：输入命令 **ibdev2netdev**，这是刚才安装的驱动所提供的脚本，可查看适配器端口与 ib 网卡的对应关系，我们找到对应网络接口为 enp1s0f1np1，如下图所示5.5

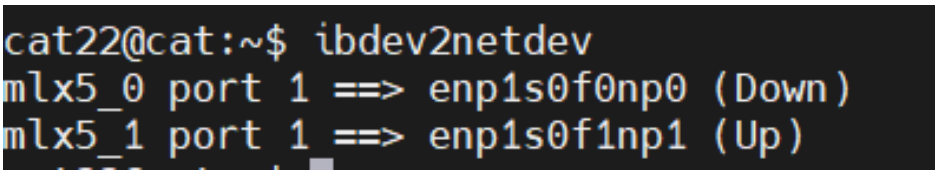


图 5.5: 查找 ib 设备对应网口

接着需要给该网络接口分配一个局域网的 ip 地址，选择分配一个私有 IP 地址范围（10.0.0.0 - 10.255.255.255）以供我们使用，在局域网内，我们给 client 配置 IP 为 10.10.10.1，给 server 配置 IP 为 10.10.10.2：

为 ib 网卡配置 ip

```
#client node
sudo ifconfig enp1s0f1np1 10.10.10.1 netmask 255.255.255.0
#server node
sudo ifconfig enp1s0f0np0 10.10.10.2 netmask 255.255.255.0
```

配置完毕后输入命令 **ifconfig** 查看是否配置成功，这里以 client node 作为示例，配置结果如下图5.6，显示配置成功。

```
cat22@cat:~$ ifconfig
enp1s0f0np0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        ether 04:3f:72:dc:27:fc  txqueuelen 1000  (以太网)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp1s0f1np1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 10.10.10.1  netmask 255.255.255.0  broadcast 10.10.10.255
        ether 04:3f:72:dc:27:fd  txqueuelen 1000  (以太网)
        RX packets 244  bytes 34856 (34.8 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 348  bytes 44343 (44.3 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

图 5.6: 配置 ip

接着我们需要测试两端之间是否能够通过 rdma 网卡相互通信，具体做法是使用 MLNX OFED 提供的 rping 命令建立一个 RC(Reliable Connection) 连接，它会使用 lib rdma_cm 在两个节点之间建立 RDMA 连接并传递消息，在终端中会输出连接建立等消息，在 server 端和 client 端分别输入以下命令:

RDMA 连接测试

```
# server node, 其中ip地址都填写为server的ip
rping -s -a 10.10.10.2
#client node
rping -c -a 10.10.10.2 -v
```

输入后，当 client node 连接时，会在终端中弹出一系列的 ping data，效果如下图5.7:

```
ping data: rdma-ping-29899: ^_`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQ
ping data: rdma-ping-29900: _`abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQR
ping data: rdma-ping-29901: `abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRS
ping data: rdma-ping-29902: abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRST
ping data: rdma-ping-29903: bcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTU
ping data: rdma-ping-29904: cdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
ping data: rdma-ping-29905: defghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVW
ping data: rdma-ping-29906: efghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWX
ping data: rdma-ping-29907: fghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXY
ping data: rdma-ping-29908: ghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ
ping data: rdma-ping-29909: hijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ[
ping data: rdma-ping-29910: ijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ[\
^Z
[2]+  已停止               rping -c -a 10.10.10.2 -v
cat22@cat:~$
```

图 5.7: rping data

至此，在两台机器中已经能够成功建立 RDMA 连接，支持我们工作的剩余部分。

5.2 实现用户态 RDMA 通信

RDMA 一共有两种通信方式：send/recv 和 read/write，其中 send 和 recv 和普通的网络通信一致，需要一方发送，一方接收，两端都需要在通信过程中主动参与，而 read/write 操作在连接建立成功之后，可以以远端 CPU 无感知的形式访问对方允许访问的内容。

在传统的网络通信中，数据需要在用户空间和内核空间之间进行多次复制，这会消耗大量的 CPU 资源。在这时 RDMA 可以通过其特有的 read/write 操作直接在应用程序的内存和网络设备之间传输数据，无需通过操作系统内核，从而避免了这些复制操作。

RDMA 支持零拷贝数据传输，即数据可以直接从源内存区域复制到目标内存区域，无需在源和目标之间进行中间复制。这可以进一步减少 CPU 的使用和内存的访问，从而提高数据传输的效率。

RDMA 通常由专门的硬件设备（如 InfiniBand 或 RoCE 网卡，此次比赛使用的设备为 RoCE 网卡）支持，这些设备可以在硬件级别进行数据传输，无需 CPU 的参与。这可以大大减少 CPU 的负载，从而让 CPU 有更多的资源来处理其他任务。

5.2.1 基于 rdma_cm API 和 InfiniBand Verbs API 的 RDMA 通信

为了在 frontswap 中实现内核态的连接建立，我们决定先尝试在用户态使两台机器使用 RDMA 通信，可以基于 rdma_cm API 和 IB_verbs API 进行通信，在连接建立成功时，我们会将 client node 中的通信代码移植进内核中以 modules 的方式安装。下面是关于用户态两端连接的建立。

RDMA 设备分为两种，一种是充当服务器（提供远内存的节点），一种是客户端（计算节点），服务器端需要运行的是一个用户态的程序，与客户端进行连接后，客户端就可以通过内核中的 RDMA 操作访问服务器端的内存。

在早期的 RDMA 测试过程中，没有在内核层面使用 RDMA 作为客户端，而是在客户端上运行和服务端相同 API 的用户态程序。在确保 rping 能够通的情况下，分别在服务端和客户端运行编写好的用户态程序，测试能否成功建立连接和发送接收可用的内存的区域 (memory region) 和 “钥匙 (remote key)”。

5.2.2 建立连接

RDMA 的建立连接过程如图5.8所示，在连接建立后可以进行发送接收消息等操作，下面结合代码详细解释建立连接的过程。

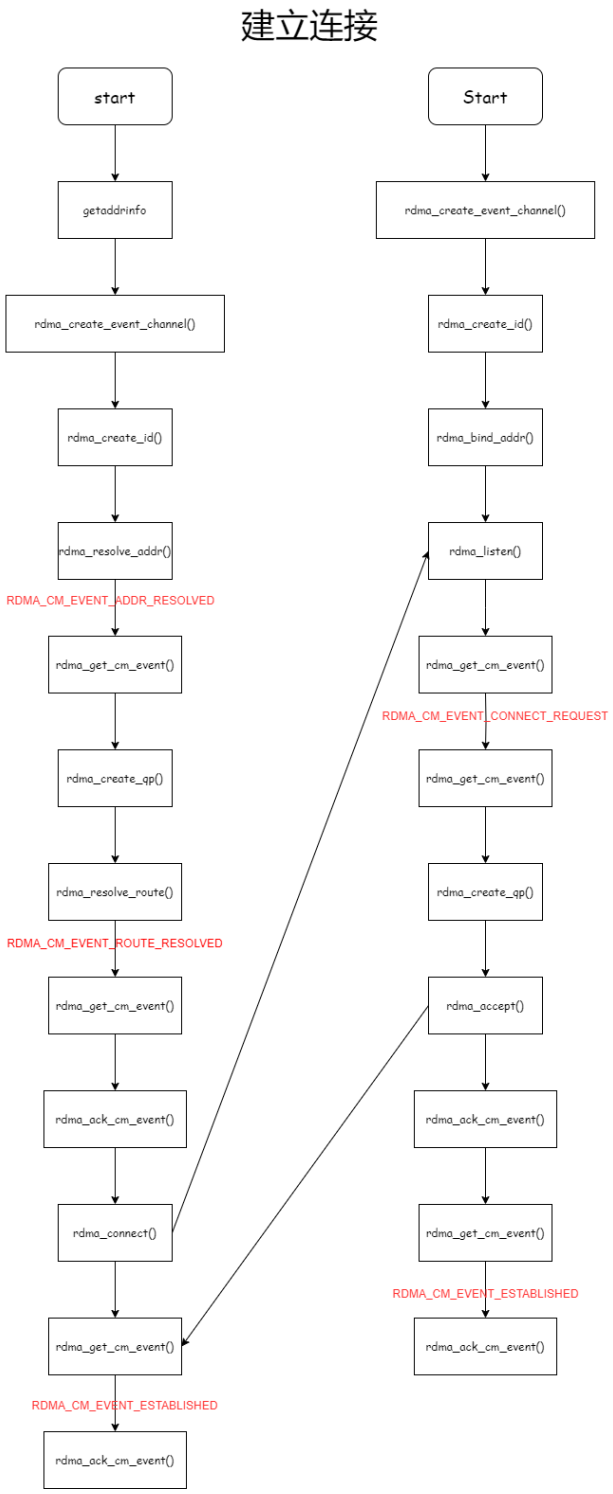


图 5.8: 建立连接

RDMA 的每一个连接是通过一个 event channel (下面简称 ec) 来管理的。rdma_create_event_channel() 创建一个 event channel。rdma_create_id() 根据该 event channel 创建一个 rdma_cm_id。接着使用 rdma_bind_addr() 为该 rdma_cm_id 绑定一个网络地址 (sockaddr)，最后用 rdma_listen() 用来监听连接请求。

服务端 RDMA 初始化及监听

```
TEST_Z(ec = rdma_create_event_channel());
TEST_NZ(rdma_create_id(ec, &listener, NULL, RDMA_PS_TCP));
TEST_NZ(rdma_bind_addr(listener, (struct sockaddr *)&addr));
TEST_NZ(rdma_listen(listener, NUM_QUEUES + 1));
```

当监听到其它设备有对此端口的连接请求时，cm_id 会接收到 RDMA_CM_EVENT_CONNECT_REQUEST 事件，阻塞函数 rdma_get_cm_event 会在监听到 CM 事件时返回，我们循环调用该函数，不断从 RDMA event channnel 中获取事件并处理。从 event channnel 获取一个事件并复制完相关资源后，需要调用 rdma_ack_cm_event 函数来确认收到这个事件。此后 RDMA event channnel 可以释放这个事件的资源。获取到的事件会被传递给 on_event 函数进行处理。如果 on_event 函数返回非零值，或者队列的状态变为 CONNECTED，则跳出循环。

处理 CM_EVENT

```
while (rdma_get_cm_event(ec, &event) == 0) {
  struct rdma_cm_event event_copy;

  memcpy(&event_copy, event, sizeof(*event));
  rdma_ack_cm_event(event);

  if (on_event(&event_copy) || q->state == queue::CONNECTED)
    break;
}
```

on_event 的函数用于处理 RDMA 事件。这个函数接收一个 rdma_cm_event 类型的参数，根据事件的类型，调用相应的处理函数。具体来说：

- 当事件类型为 RDMA_CM_EVENT_CONNECT_REQUEST 时，调用 on_connect_request 函数处理连接请求。
- 当事件类型为 RDMA_CM_EVENT_ESTABLISHED 时，调用 on_connection 函数处理连接已经建立事件。
- 当事件类型为 RDMA_CM_EVENT_DISCONNECTED 时，调用 on_disconnect 函数处理连接断开事件。

事件处理函数

```
int on_event(struct rdma_cm_event *event)
{
  printf("%s\n", __FUNCTION__);
  struct queue *q = (struct queue *) event->id->context;

  switch (event->event) {
    case RDMA_CM_EVENT_CONNECT_REQUEST:
      return on_connect_request(event->id, &event->param.conn);
    case RDMA_CM_EVENT_ESTABLISHED:
      return on_connection(q);
```

```
    case RDMA_CM_EVENT_DISCONNECTED:
      on_disconnect(q);
      return 1;
    default:
      printf("unknown event: %s\n", rdma_event_str(event->event));
      return 1;
  }
}
```

从 on_event 中可以看到，在触发相应的事件时，会调用相应的函数，而在建立连接的过程中会调用的函数有 on_connect_request() 和 on_connection()。on_connection() 在 Send/Recv 部分再详细介绍。

on_connect_request 在接收到 RDMA 连接请求时被调用。函数的主要作用是处理新的连接请求，包括设置连接参数、创建队列对（Queue Pair，QP）、尝试分配 protect domain、注册 memory region，接受连接请求。其中 create_qp() 会通过调用 rdma_create_qp() 创建队列对。其中的 rdma_accept() 也是所使用的 API，作用是接受 RDMA 连接请求。当服务器收到客户端的连接请求后，需要调用 rdma_accept 函数来接受这个请求，从而建立 RDMA 连接，并触发 RDMA_CM_EVENT_ESTABLISHED 事件，进而调用 on_connection 函数。

接收到连接请求

```
int on_connect_request(struct rdma_cm_id *id, struct rdma_conn_param *param)
{

  struct rdma_conn_param cm_params = {};
  struct ibv_device_attr attrs = {};
  struct queue *q = &gctrl->queues[queue_ctr++];

  TEST_Z(q->state == queue::INIT);
  printf("%s\n", __FUNCTION__);

  id->context = q;
  q->cm_id = id;

  struct device *dev = get_device(q);
  create_qp(q);

  TEST_NZ(ibv_query_device(dev->verbs, &attrs));

  printf("attrs: max_qp=%d, max_qp_wr=%d, max_cq=%d max_cqe=%d \
          max_qp_rd_atom=%d, max_qp_init_rd_atom=%d\n", attrs.max_qp,
          attrs.max_qp_wr, attrs.max_cq, attrs.max_cqe,
          attrs.max_qp_rd_atom, attrs.max_qp_init_rd_atom);

  printf("ctrl attrs: initiator_depth=%d responder_resources=%d\n",
      param->initiator_depth, param->responder_resources)
  cm_params.initiator_depth = param->initiator_depth;
  cm_params.responder_resources = param->responder_resources;
```

```
  cm_params.rnr_retry_count = param->rnr_retry_count;
  cm_params.flow_control = param->flow_control;

  TEST_NZ(rdma_accept(q->cm_id, &cm_params));

  return 0;
}
```

接下来介绍客户端的建立连接代码。

首先获取服务器的地址信息，getaddrinfo 将用户输入的 IP 地址和端口号保存在 &addr 中，然后使用 rdma_create_event_channel() 创建一个 event channnel，之后使用 rdma_create_id() 创建与之关联的 rdma_cm_id。最后，rdma_resolve_addr 解析服务器的地址，随后在解析完成后触发一个 RDMA_CM_EVENT_ADDR_RESOLVED 事件，事件的触发和事件的处理是异步的，处理流程在后文详细介绍。

客户端 RDMA 初始化及地址和路由解析

```
  TEST_NZ(getaddrinfo(argv[1], argv[2], NULL, &addr));
  TEST_Z(ec = rdma_create_event_channel());
  TEST_NZ(rdma_create_id(ec, &conn, NULL, RDMA_PS_TCP));
  TEST_NZ(rdma_resolve_addr(conn, NULL, addr->ai_addr, TIMEOUT_IN_MS));
```

而后的循环与前文介绍的循环类似，都是循环从 RDMA event channnel 中获取事件并处理。

对于 on_event() 进行详细说明,接上文所述,触发了 RDMA_CM_EVENT_ADDR_RESOLVED 事件后，on_event() 函数会对这个事件进行处理，调用 on_addr_resolved() 函数。

客户端事件处理函数

```
int on_event(struct rdma_cm_event *event)
{
  int r = 0;

  if (event->event == RDMA_CM_EVENT_ADDR_RESOLVED)
    r = on_addr_resolved(event->id);
  else if (event->event == RDMA_CM_EVENT_ROUTE_RESOLVED)
    r = on_route_resolved(event->id);
  else if (event->event == RDMA_CM_EVENT_ESTABLISHED)
    r = on_connection(event->id->context);
  else if (event->event == RDMA_CM_EVENT_DISCONNECTED)
    r = on_disconnect(event->id);
  else
    die("on_event: unknown event.");

  return r;
}
```

on_addr_resolved 在 RDMA 客户端解析服务器地址成功后被调用。

地址已解析

```
int on_addr_resolved(struct rdma_cm_id *id)
{
  struct ibv_qp_init_attr qp_attr;
  struct connection *conn;

  printf("address resolved.\n");

  build_context(id->verbs);
  build_qp_attr(&qp_attr);

  TEST_NZ(rdma_create_qp(id, s_ctx->pd, &qp_attr));

  id->context = conn = (struct connection *)malloc(sizeof(struct connection));

  conn->id = id;
  conn->qp = id->qp;
  conn->num_completions = 0;

  register_memory(conn);
  post_receives(conn);

  TEST_NZ(rdma_resolve_route(id, TIMEOUT_IN_MS));

  return 0;
}
```

在 on_addr_resolved() 中，首先 build_context() 根据 rdma_cm_id 中的 context（RDMA 设备的 context）来构建 context（会话特定的 context），构建初始化会话特定的 context 的过程包括设置好 RDMA 设备的 context，分配 protect domain，创建 Completion Channel 以及 Completion Queue（用于存放已完成的请求，后文简称为 CQ）。另外，还会请求在完成队列有新的完成事件时通知应用程序并创建一个线程来轮询 CQ。

构建会话特定 context

```
void build_context(struct ibv_context *verbs)
{
  if (s_ctx) {
    if (s_ctx->ctx != verbs)
      die("cannot handle events in more than one context.");

    return;
  }

  s_ctx = (struct context *)malloc(sizeof(struct context));

  s_ctx->ctx = verbs;

  TEST_Z(s_ctx->pd = ibv_alloc_pd(s_ctx->ctx));
  TEST_Z(s_ctx->comp_channel = ibv_create_comp_channel(s_ctx->ctx));
```

```
  TEST_Z(s_ctx->cq = ibv_create_cq(s_ctx->ctx, 10, NULL, s_ctx->comp_channel, 0)); /*
      cqe=10 is arbitrary */
  TEST_NZ(ibv_req_notify_cq(s_ctx->cq, 0));

  TEST_NZ(pthread_create(&s_ctx->cq_poller_thread, NULL, poll_cq, NULL));
}
```

接着 build_qp_attr() 构建和初始化队列对（Queue Pair 后文简称为 QP）的属性。发送和接收队列都使用 CQ，将 QP 的 type 设置为 IBV_QPT_RC 并且初始化可以存放的最大工作请求数量以及一次工作请求可以操作的数据块的数量。

初始化 QP

```
void build_qp_attr(struct ibv_qp_init_attr *qp_attr)
{
  memset(qp_attr, 0, sizeof(*qp_attr));

  qp_attr->send_cq = s_ctx->cq;
  qp_attr->recv_cq = s_ctx->cq;
  qp_attr->qp_type = IBV_QPT_RC;

  qp_attr->cap.max_send_wr = 10;
  qp_attr->cap.max_recv_wr = 10;
  qp_attr->cap.max_send_sge = 1;
  qp_attr->cap.max_recv_sge = 1;
}
```

connection 结构体包含了 RDMA 连接所需要的资源和信息，让 id->context 指向这个结构体，后续对资源进行操作时就比较方便，可以访问 rdma_cm_id 来找到这些资源。所以在 on_addr_resolved() 紧接着就将 conn（指向一个 connection 实例的指针）赋值给了 id->context，并完成了对 conn 的初始化，其中 memory region 的 register 是通过 register_memory(conn) 实现的。

连接所需资源信息

```
struct connection {
  struct rdma_cm_id *id;
  struct ibv_qp *qp;

  struct ibv_mr *recv_mr;
  struct ibv_mr *send_mr;

  char *recv_region;
  char *send_region;

  int num_completions;
};
```

register_memory(conn) 其实也是 conn 初始化，作用是对其中的 region 和 memory region 进行初始化，分配 BUFFER_SIZE 大小的内存区域，并通过 ibv_reg_mr() 将这个内存区域注册到 RDMA

设备，并将返回的内存区域对象存入 conn 中的 mr 中。

注册内存

```
void register_memory(struct connection *conn)
{
  conn->send_region = malloc(BUFFER_SIZE);
  conn->recv_region = malloc(BUFFER_SIZE);

  TEST_Z(conn->send_mr = ibv\_reg\_mr()(
    s_ctx->pd,
    conn->send_region,
    BUFFER_SIZE,
    0));

  TEST_Z(conn->recv_mr = ibv_reg_mr(
    s_ctx->pd,
    conn->recv_region,
    BUFFER_SIZE,
    IBV_ACCESS_LOCAL_WRITE));
}
```

紧接着是 post_receives() 函数，将一个接收工作请求 (Receive Work Request) 提交到 RDMA 设备的接收队列中，这样，当对端发送数据过来时，RDMA 设备就可以立即开始接收数据，而不需要等待接收工作请求被提交。

提交接收工作请求

```
void post_receives(struct connection *conn)
{
  struct ibv_recv_wr wr, *bad_wr = NULL;
  struct ibv_sge sge;

  wr.wr_id = (uintptr_t)conn;
  wr.next = NULL;
  wr.sg_list = &sge;
  wr.num_sge = 1;

  sge.addr = (uintptr_t)conn->recv_region;
  sge.length = BUFFER_SIZE;
  sge.lkey = conn->recv_mr->lkey;

  TEST_NZ(ibv_post_recv(conn->qp, &wr, &bad_wr));
}
```

在 on_addr_resolved() 的末尾调用了 rdma_resolve_route() 函数用来解析 RDMA 设备到目标地址的路由，在这个函数结束时会触发一个 RDMA_CM_EVENT_ROUTE_RESOLVED 事件，这个事件会被 on_event() 处理，调用 on_route_resolved() 函数。

on_route_resolved() 在 RDMA 客户端成功解析到服务器的路由后被调用。这个函

数先将 rdma_conn_param(包含建立连接过程所需要的参数) 结构体实例的所有字段设置为 0，随后调用 rdma_connect() 尝试建立 RDMA 连接，同样，也会触发一个事件 RDMA_CM_EVENT_ESTABLISHED。on_event() 会继续对这个事件进行处理，调用 on_connection() 函数。

路由已解析

```
int on_route_resolved(struct rdma_cm_id *id)
{
  struct rdma_conn_param cm_params;

  printf("route resolved.\n");

  memset(&cm_params, 0, sizeof(cm_params));
  TEST_NZ(rdma_connect(id, &cm_params));

  return 0;
}
```

on_connection() 只需要打印一下 connected 确认成功建立连接即可。

连接已建立

```
int on_connection(void *context)
{
  printf("connected\n");

  return 0;
}
```

5.2.3 RDMA 通信：Send/Recv

RDMA 的优势在于消息的发送接收过程 CPU 是不需要参与其中的，但却不是从一开始就是如此，客户端首先要知道服务端可用的内存的地址和 “钥匙”，也就是要获得这片远端内存的读写权限，这一定是要服务端的 CPU 允许的，因此最开始使用 Send/Recv 的通信方式，即一方发送，一方接收的形式。如图5.9所示，CPU 参与其中，使得客户端能够获得可用的内存地址与 key。而之后客户端对服务端的内存的读写操作通过 Read/Write 的通信方式来实现，远端 CPU 是不需要参与其中的，由此也可以看出 Send/Recv 和 Read/Write 通信方式的差别，在于前者为双端操作，而后者为单端操作。

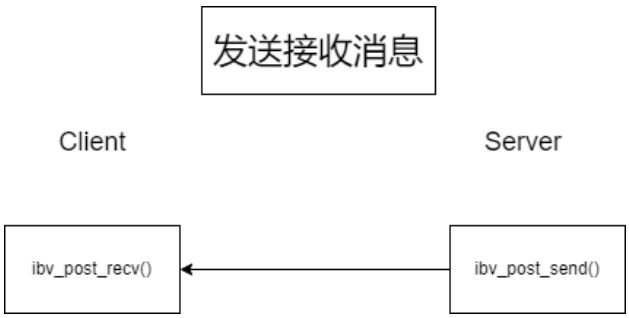


图 5.9: 发送接收消息

在连接建立之后，服务端会进行一次 Send 操作，该操作是在接收到 RDMA_CM_EVENT_ESTABLISHED 事件之后调用的 on_connection 中进行的，主要作用是向客户端发送服务器端的内存区域信息，包括内存区域的地址和键值。如果发送成功，它会将队列的状态设置为已连接。

服务端 Send

```
int on_connection(struct queue *q)
{
  printf("%s\n", __FUNCTION__);
  struct ctrl *ctrl = q->ctrl;

  TEST_Z(q->state == queue::INIT);

  if (q == &ctrl->queues[0]) {
    struct ibv_send_wr wr = {};
    struct ibv_send_wr *bad_wr = NULL;
    struct ibv_sge sge = {};
    struct memregion servermr = {};

    printf("connected. sending memory region info.\n");
    printf("MR key=%u base vaddr=%p\n", ctrl->mr_buffer->rkey, ctrl->mr_buffer->addr);

    servermr.baseaddr = (uint64_t) ctrl->mr_buffer->addr;
    servermr.key  = ctrl->mr_buffer->rkey;

    wr.opcode = IBV_WR_SEND;
    wr.sg_list = &sge;
    wr.num_sge = 1;
    wr.send_flags = IBV_SEND_SIGNALED | IBV_SEND_INLINE;

    sge.addr = (uint64_t) &servermr;
    sge.length = sizeof(servermr);

    TEST_NZ(ibv_post_send(q->qp, &wr, &bad_wr));

    // TODO: poll here
  }

  q->state = queue::CONNECTED;
  return 0;
}
```

客户端的 Recv 接收操作在建立连接时提到了，post_receives() 函数提交了一个接收的工作请求，所以当有消息 Send 到客户端时，能够正确接收。

5.2.4 断开连接

断开连接与建立连接一样重要，确保所有的数据都被正常接收了，同时也可以释放资源，断开连接的过程如图5.10所示。

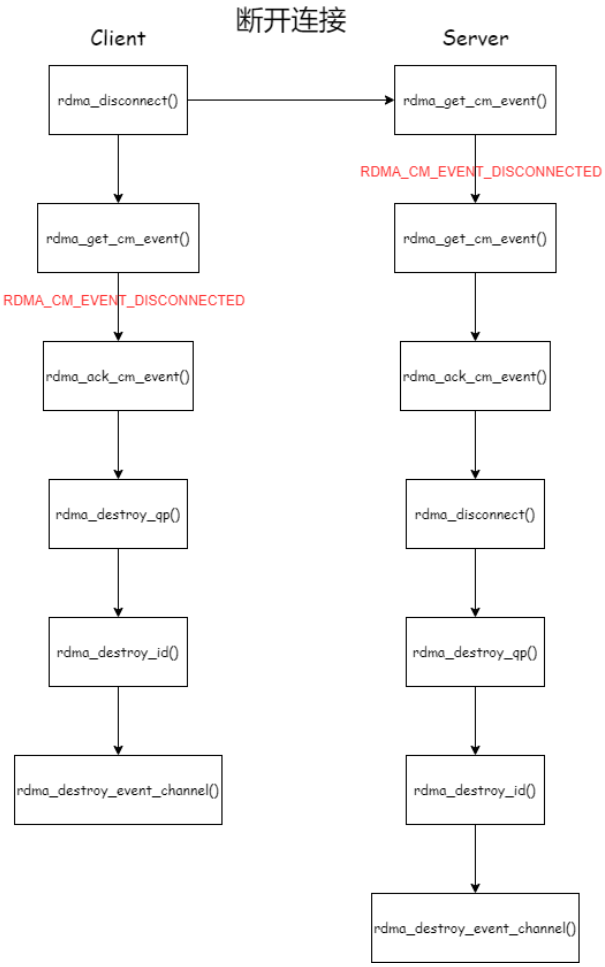


图 5.10: 断开连接

断开连接是由客户端发起的，客户端调用 rdma_disconnect() 来断开连接，在服务端会持续获取事件，当获取到事件为 RDMA_CM_EVENT_DISCONNECTED 事件时，对事件进行确认后，也会调用 rdma_disconnect()。同时客户端也会获取到 RDMA_CM_EVENT_DISCONNECTED 事件，然后各自进行断开连接后需要进行的销毁 RDMA 资源的操作。所使用的 API 为 rdma_destroy_qp(),rdma_destroy_id() 和 rdma_destroy_event_channel()，作用分别为销毁队列对 (Queue Pair)，销毁 rdma_cm_id 和销毁 event channnel。

运行结果 在搭建好的两台机器上，分别运行服务端程序和客户端程序，服务端的 IP 设置为 10.10.10.2，客户端的 IP 设置为 10.10.10.1，分别编写好 Makefile，make 之后分别输入以下命令：

```
# 服务器端
./rmserver 50000
# 客户端
./client 10.10.10.2 50000
```

如图5.11所示，服务端监听 50000 端口，客户端运行 client 程序与其进行通信后 server 端打印了连接成功，发送内存地址和 key 成功，断开连接等信息，初步证实了 server 端程序的正确性，进一步的验证需要在另一台机器上的内核层面运行 RDMA 客户端程序来进行。

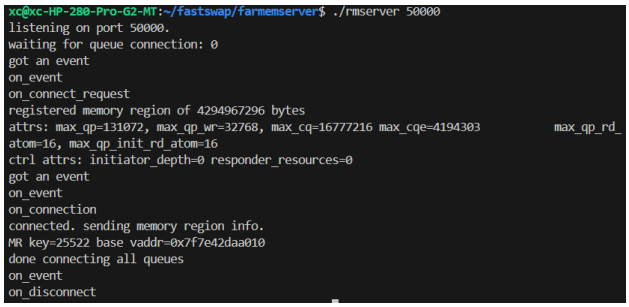


图 5.11: 运行结果

5.3 Frontswap 的接口介绍与使用

在内核中定义的部分被称为 Frontswap 的 frontend，我们需要完成的对该接口的 implementation 部分通常称为 frontswap 的 backend。

在开始之前需要说明的是，只有在 client 端需要实现内核态的 RDMA 连接建立，因为我们希望通过 RDMA 所进行的页面置换机制作为 mudules 安装在内核中，而服务端只需要在用户态运行一个程序用来监听 client 端的建立连接以及销毁连接的请求，并将 memory region 以及 rkey 发送给 client 即可，中间的 read 和 write 操作对于 server 来说是无感知的，不需要对内核进行修改；因此**我们对于内核的修改是基于 Linux6.1 进行的**。我们需要做的就是对 frontswap 接口进行一种实现，在/include/linux/frontswap.h 中定义的 frontswap_ops 描述了 frontswap 的接口行为：

frontswap_ops

```
struct frontswap_ops {
  void (*init)(unsigned); /* this swap type was just swapon'ed */
  int (*store)(unsigned, pgoff_t, struct page *); /* store a page */
  int (*load)(unsigned, pgoff_t, struct page *); /* load a page */
  void (*invalidate_page)(unsigned, pgoff_t); /* page no longer needed */
  void (*invalidate_area)(unsigned); /* swap type just swapoff'ed */
};
```

这是 Linux 内核中典型的**机制 (mechanism)-策略 (policy)** 的关系，我们本部分要实现的 DRAM 就是 frontswap 的一种策略，通过实现一种 frontswap_ops 并让内核采用此策略来完成本部分的工作，该机制可以通过调用函数 frontswap_register_ops 更换策略。frontswap 的核心操作在于 store 和 load，其中都包含了三个参数，分别代表了换出的 swap area(type)，换出的页偏移位置 (pgoffset) 以及换出的页 (page)，在/mm/page_io.c 中进行 swap_writepage 和 swap_readpage 时，以 write 操作为例，会先尝试对 frontswap 接口进行尝试，调用当 frongswap 接口被启用并成功调用将页面换入换出时，将会直接返回，否则才进行 writepage 的操作，如图5.12所示：

```
int swap_writepage(struct page *page, struct writeback_control *wbc)
     */
    ret = arch_prepare_to_swap(&folio->page);
    if (ret) {
        folio_mark_dirty(folio);
        folio_unlock(folio);
        goto out;
    }
    if (frontswap_store(&folio->page) == 0) {
        folio_start_writeback(folio);
        folio_unlock(folio);
        folio_end_writeback(folio);
        goto out;
    }
    ret = __swap_writepage(&folio->page, wbc);
out:
    return ret;
}
```

图 5.12: writepage

接下来，在 frontswap 中，frontswap_store 函数在 frontswap 启用时会进一步调用 __frontswap_store 函数，在此处，如果我们在先前注册 (register) 了一种 frontswap_ops，就会调用该 ops 所实现的 store 函数，将页换出到所实现的策略当中处理，其它操作同理。总体来说，通过 frontswap 接口进行页面的换入换出机制如下图5.13所示，以 store 操作为例：

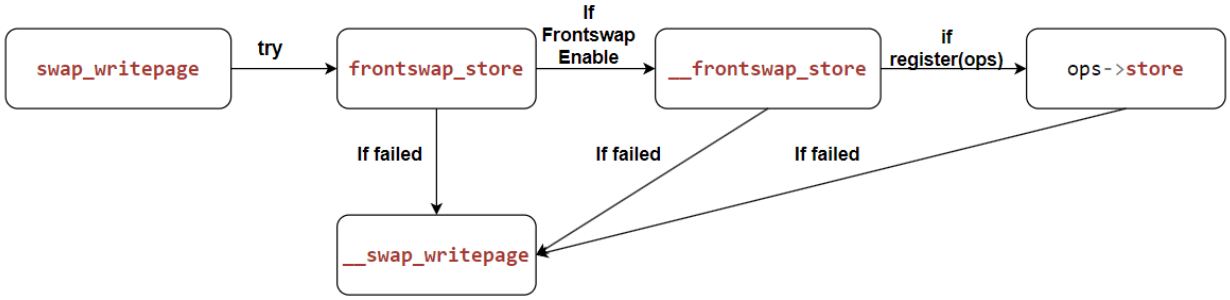


图 5.13: frontswap_store

至此，我们大概描述了 frontswap 接口的大致使用方式。

5.4 实现 frontswap 的策略：fastswap

下面开始实现 frontswap 的 backend。我们参考了 *FASTSWAP*[1] 这篇文章在 4.11 版本上的工作，在 6.1 内核上尝试实现基本的 read 和 write 功能，它将以 module 的方式安装在内核当中。首先定义 sswap_frontswap_ops 并放在模块文件 (fastswap.c) 中；对于安装在内核上的 module，需要编写模块的初始化以及退出函数，并在模块初始化时调用 frontswap_register_ops 函数向内核注册 sswap_frontswap_ops，代码实现如下：

fastswap 模块初始化

```
1 static struct frontswap_ops sswap_frontswap_ops = {
2   .init = sswap_init,
```

```
    .store = sswap_store,
    .load = sswap_load,
    .invalidate_page = sswap_invalidate_page,
    .invalidate_area = sswap_invalidate_area,

};
static int __init init_sswap(void)
{
   pr_info("begin init sswap\n");
  if(frontswap_register_ops(&sswap_frontswap_ops))
  {
     pr_err("sswap register ops failed!\n");
     return -1;
  }
   pr_info("sswap module loaded\n");
   return 0;
}

static void __exit exit_sswap(void)
{
   pr_info("unloading sswap\n");
}

module_init(init_sswap);
module_exit(exit_sswap);
MODULE_LICENSE("GPL v2");
MODULE_DESCRIPTION("Fastswap");
```

接下来需要实现策略中的各个函数，其中查阅 Linux 内核手册可知，invalidate_page 将会从远内存中移除该页，invalidate_area 将会移除所有和此交换设备相关的页，类似于 swapoff 的操作。我们暂时没有使用到这两个接口，于是直接简单在函数中返回即可，同理 init 操作也暂时没有什么需要做的工作，它将会在 frontswap_init 中被调用，这里只让它输出相关信息告知初始化成功即可。在关键函数 load 和 store 的实现上，再一次使用了策略和机制的思想，进一步抽象为 sswap_rdma_write 和 sswap_rdma_read 两个接口，将页偏移转换成地址的偏移，进一步处理。可根据接口来完成不同的实现，为后续要完成的 dram(模拟远程内存) 以及 rdma 的实现提供便捷，这部分的代码实现如下：

sswap 的 read 和 write 操作

```
static int sswap_store(unsigned type, pgoff_t pageid,
        struct page *page)
{
  if (sswap_rdma_write(page, pageid << PAGE_SHIFT)) {
    pr_err("could not store page remotely\n");
    return -1;
  }
  return 0;
}

static int sswap_load(unsigned type, pgoff_t pageid, struct page *page)
```

```
{
  if (unlikely(sswap_rdma_read_sync(page, pageid << PAGE_SHIFT))) {
    pr_err("could not read page remotely\n");
    return -1;
  }
  return 0;
}
```

注：使用同步读 (read_sync) 接口是为后续可能需要扩展的异步读 (read_async) 做准备。

5.5 用 DRAM 实现页的换入换出

在前面提到，对 fastswap 进行了抽象，下面将使用本机的 DRAM 进行实现 fastswap_dram。fastswap_dram 同样也作为一个模块安装在内核上，是 fastswap 的一种实现方式。**在模块的 init 函数中，需要分配一块内存区域作为换出页的存放区域，对这块区域的大小有相关要求，在后续我们将说明**，在模块的 exit 函数中，只需要释放这一块内存区域即可。

fastswap_dram 的初始化与退出

```
static void __exit sswap_dram_cleanup_module(void)
{
  vfree(drambuf);
  pr_info("DRAM backend is cleaned up\n");
}

static int __init sswap_dram_init_module(void)
{
  pr_info("start: %s\n", __FUNCTION__);
  pr_info("will use new DRAM backend");

  drambuf = vzalloc(REMOTE_BUF_SIZE);
  pr_info("vzalloc'ed %lu bytes for dram backend\n", REMOTE_BUF_SIZE);

  if(SWAPFILE_SIZE > REMOTE_BUF_SIZE) {
    pr_info("warning: swapfile size is larger than remote buffer size\n");
    pr_info("this may cause a part of pages are not use frontswap\n");
  }

  pr_info("DRAM backend is ready for reqs\n");
  return 0;
}

module_init(sswap_dram_init_module);
module_exit(sswap_dram_cleanup_module);
```

关键部分为对页面的 read 和 write 操作，在 dram 中的操作很简单：进行 write 操作时，从 page 结构体中使用 kmap 映射到对应的 vaddr，将其内容 copy 到提供的地址偏移 roffset 后使用 kunmap 解除该映射即可；read 操作与 write 进行复制的方向相反，需要更加小心地判断该页是否为可以从远端读取的状态，最后在读取完毕后更新 Page 的状态。

5.5.1 对 far memory 大小的要求

对于充当远程内存的大小也有一定的要求。在 Linux 系统中为了回收 Anonymous pages，会在磁盘上开辟专门的 swap space 作为这些页面内容的 backing store，这些 swap space 由若干个 swap area 组成，在本项目中使用的 swap area 是 swapfile。每个 swap area 被分割为若干个 slot(swap entry)。内核在回收页面之前，会将它们的内容复制到一个 swap area 的某一个 slot 中保存起来，为页面分配对应的 type（swap_info 数组的索引，用来索引 swap area）以及 swap entry。分配的 swap entry 相当于是对某个 swap area 的偏移量，最基本的要求就是在相对应的 swapfile 的可分配范围之内，或者说最大只能分配到该 swapfile 大小。等到调用 swap_writepage()，才会执行真正的 I/O 操作，将页面的内容写入外部的 swap area，然后清除 swap cache 对页面的指向，释放页面所占的内存。而我们要实现的工作。就是在这个过程中劫持换出到 swap area(磁盘区域) 的过程，而是换出到准备的远程内存中去。

在 ___frontswap_store 中所获取的 offset，就是根据 swap entry 计算出的相对于整个 swap area 的页偏移，如下图5.14所示：

```
int __frontswap_store(struct page *page)
{
    int ret = -1;
    swp_entry_t entry = { .val = page_private(page), };
    int type = swp_type(entry);
    struct swap_info_struct *sis = swap_info[type];
    pgoff_t offset = swp_offset(entry);
```

图 5.14: pgoffset 的含义

由先前的讨论可知，此参数 offset 后续我们会传入 frontswap 接口中，作为远程内存的相对地址使用。所以，**远程内存的大小必须要大于等于 swap area**，否则会导致需要换出的页面相对于基地址（远程内存起始地址）的偏移量大于远程内存大小的情况。

采用 swapon 命令查看本地 swap 交换区的大小，发现为 15.8GB，而本地内存总大小为 16GB，显然不能够全部分配给 DRAM 用来换出页面，接下来需要重新添加 swap 分区，使得其大小为 4GB：

修改 swap 分区

```
# 删除旧的swap分区：
sudo swapoff /dev/nvme0n1p3
sudo swapoff /dev/nvme0n1p6
sudo rm /dev/nvme0n1p3
sudo rm /dev/nvme0n1p6
# 注释在/etc/fstab文件中这两个swap分区的自动挂载

#添加新的swap分区，大小为4GB
sudo dd if=/dev/zero of=/opt/swapfile/4GB.swap bs=1024 count=4194304
ll -h /opt/swapfile/4GB.swap
#在/etc/fstab中设置开机自动挂载，具体格式参考文件中的描述
/opt/swapfile/4GB.swap none  swap sw 0 0
```

执行完毕后，再次执行命令 swapon，可以看到 swap 分区已经修改为 4GB，如下图5.15所示：

```
cat22@cat:~$ swapon
NAME                      TYPE SIZE USED PRIO
/opt/swapfile/4GB.swap file   4G 1.5M   -2
```

图 5.15: 修改 swap 分区

除此之外，为了不引起不必要的麻烦，我们在 fastswap_dram.c 中添加了宏 **SWAPFILE_SIZE**，在 write 操作中进行了判别：即使需要换出的 offset 超过了我们分配出的 REMOTE_BUFFER 的大小，也能够直接返回-1，跳出 frontswap 接口在内核中进行后续的正常处理，而不会在 copy_page 中因为索引超出大小而报错，对于后续的 RDMA BACKEND 的实现，也采用了此方法。实现代码如下：

fastswap_dram read and write

```
int sswap_rdma_write(struct page *page, u64 roffset)
{
  //this part is not use frontswap
  if(roffset >= REMOTE_BUF_SIZE)
    return -1;
  void *page_vaddr;
  page_vaddr = kmap_atomic(page);
  copy_page((void *) (drambuf + roffset), page_vaddr);
  kunmap_atomic(page_vaddr);
  DEBUG_PRINT("write over\n");
  return 0;
}
EXPORT_SYMBOL(sswap_rdma_write);

int sswap_rdma_read_sync(struct page *page, u64 roffset)
{
  //this part is not use frontswap
  if(roffset >= REMOTE_BUF_SIZE)
    return -1;
  void *page_vaddr;

  VM_BUG_ON_PAGE(!PageSwapCache(page), page);
  VM_BUG_ON_PAGE(!PageLocked(page), page);
  VM_BUG_ON_PAGE(PageUptodate(page), page);

  page_vaddr = kmap_atomic(page);
  copy_page(page_vaddr, (void *) (drambuf + roffset));
  kunmap_atomic(page_vaddr);

  SetPageUptodate(page);
  unlock_page(page);
  DEBUG_PRINT("read over\n");
  return 0;
}
EXPORT_SYMBOL(sswap_rdma_read_sync);
```

5.6 修改 6.1 内核以适配 frontswap 接口

在 Linux5.4 的内核中，经尝试已经能够成功安装模块并通过内存压力测试 (测试将在下一节中详细说明)，但是在 6.1 的内核中，模块编译发生报错，大致信息为 Unknow symbol(未知的符号)，导致 Module.symvers 文件出错，查阅资料可知，Linux 内核中，一个模块的编译如果需要调用内核中的函数，需要依赖内核编译时生成的 Module.symvers 文件，在这个文件中保存了大量通过 EXPORT_SYMBOL 导出的全局符号。于是定位到错误在于在 **6.1 内核中没有 EXPORT frontswap 的相关接口**到全局符号中，而在自定义模块中却使用了 register_ops 等函数导致出错。在 5.4 的内核中这是默认操作，所以编译成功。于是在内核的 frontswap.c 中，相关函数的最后做类似的 EXPORT_SYMBOL 操作即可。同理，在 fastswap.c 中，使用到的 fastswap_dram.c 中实现的函数也需要进行 EXPORT 处理 (上部分代码已经做了展示)。

除此之外，Linux5.4 内核与 6.1 内核对于注册的 frontswap_ops 的管理也大有不同。我们在测试时发现，注册 frontswap_ops(设备) 总是失败，这正是由于两者对于已注册的设备的不同处理方式导致的，5.4 的 frontswap_ops 结构体相比于 6.1 多增加了一个 next 指针，当一个设备进行 register 操作时，会把其 next 指针指向前一个加入的设备，在此之后自身充当这个链表的头指针；在进行 load 和 store 操作时，将会从该链表头开始进行尝试，只要有一个设备成功进行 load/store，则直接返回结果并退出，正如官方文档中描述的那样：

“frontswap_ops are added by frontswap_register_ops, and provide the frontswap ”backend” implementation functions. Multiple implementations may be registered, but implementations can never deregister.”

而在 6.1 的内核中，摈弃了这种设计，直接以一种简单的方式来实现：定义一个全局的 frontswap_ops 变量，在定义时被初始化为空。进行注册时如果该全局变量为空，则将要加入的 ops 赋值给它，否则直接退出注册流程并返回注册失败，后续进行 load/store 操作时也是对它操作。但是我们手动注册的 ops 并非总是第一个注册的 ops，在编译内核前的.config 文件中包含 ZSWAP 相关选项，zswap 一种 frontswap 的 backend 实现，frontswap_ops 被抢先注册导致注册流程失败。于是我们需要进行修改，注释掉前面的条件判断语句，直接将新注册的 ops 赋值给 frontswap_ops 全局变量即可，实现如下：

修改内核 register 函数

```
int frontswap_register_ops(const struct frontswap_ops *ops)
{
  //this only enable one ops! diff from kernel 5.4
  // if (frontswap_ops)
  //  return -EINVAL;

  frontswap_ops = ops;
  static_branch_inc(&frontswap_enabled_key);
  return 0;
}
EXPORT_SYMBOL(frontswap_register_ops);
```

最后对内核进行重新编译安装，重启后安装模块：

重装内核与模块

```
# 重装内核
```

```
make -j12
sudo make modules_install -j12
sudo make install

# 编译安装模块
make BACKEND=DRAM
sudo insmod fastswap_dram.ko
sudo insmod fastswap.ko
```

重新编译内核并安装后，再执行内存压力测试程序，限制程序内存的使用以触发 swap 操作，发现成功调用了 frontswap 接口的 DRAM BACKEND 并实现了拦截 swap 的功能，此部分工作完全成功!

5.7 在内核态实现 RDMA 连接的建立和销毁

基于以上实现，只需要复用 fastswap.c 的 sswap 接口并重写 sswap_rdma_write 和 read 函数即可，但首先，我们需要进行内核态 RDMA 连接的建立。此部分的内容我们参考 6.1kernel 中/drivers/nvme/host/rdma.c 目录下的实现。

5.7.1 内核态 RDMA 连接的建立

首先，我们需要调用 ib_register_client 函数向 InfiniBand 中间层注册 ib_client，在这之后，开始相关的初始化操作。

首先定义了一个结构体 ctrl 用来管理连接相关的内容，其结构体定义如下所示：

ctrl 结构体

```
struct sswap_rdma_ctrl {
  struct sswap_rdma_dev *rdev;
  struct rdma_queue *queues;
  struct sswap_rdma_memregion servermr;

  union {
    struct sockaddr addr;
    struct sockaddr_in addr_in;
  };

  union {
    struct sockaddr srcaddr;
    struct sockaddr_in srcaddr_in;
  };
};
```

在相关结构 rdma_dev 中，包含了 protect domain 以及 rdma device 的相关信息，queues 指针用来管理在连接中所创建的所有 RDMA 队列，rdma_queue 结构体是对底层的各种类型的 ib_queue 的封装，其中包含了 queue pair 以及 complete queue 等信息，还包含了可用于同步队列[1]操作的一些变量，此处使用的是 completion 机制。除此之外，此控制器包含了从远端获取到的内存区域以及两端的网络地址信息。

[1]注：在此后提到的队列，如果没有特别说明，都指的是封装后的结构体

在给 ctrl 分配空间后，把网络地址信息存入其中；接着进行队列的初始化。此部分内容在注册完 ib_client 后通过 sswap_rdma_create_ctrl 函数调用。

初始化队列并解析地址

在 Fastswap[1] 的队列里，一共有三种类型，包括写队列、同步读队列以及异步读队列，我们在此项目中暂时没有实现异步读相关的接口，于是只考虑两种队列类型：写队列和同步读队列。**我们为每一个 CPU 核心都对应了一个同步读队列与一个同步写队列**。初始化队列工作在 sswap_rdma_init_queue 中进行，分为如下几步：

1. 首先需要确定队列的类型，如上所述分别将队列类型设置为写队列和读队列，接着初始化 completion，在后续中用于同步操作。

2. 然后，调用 rdma_create_id 为这些队列分配 rdma_cm_id(类似 socket 结构体) 用来监听 RDMA_CM_EVENT。在用户态的程序中我们也提到了这个函数，不同的是我们需要**将事件处理句柄作为参数传入函数**中，当触发对应事件时，和用户态类似，通过获取事件的类型来调用不同的处理函数。另外还有两个参数需要说明:port space 参数指定准备使用的 RDMA 端口空间，我们选用 RDMA_PS_TCP 提供可靠、面向连接的 QP 通信其与 TCP 不同，RDMA 端口空间提供基于消息的通信，而不是基于流的通信。qp type 指定使用的 RDMA 队列类型，我们选用 RC(reliable connection) 类型。

3. 分配得到 cm_id 后，和用户态 client 一样，调用 rdma_resolve_addr 函数解析网络地址，解析完毕后会调用事件处理函数处理 RDMA_CM_EVENT_ADDR_RESOLVED 事件（在后续详细说明）。

4. 做完上述一系列工作后，使用 wait_for_completion 等待 queue->cm_done 这个 completion 变量的完成。在 Linux 内核中，completion 是通过 wait_for_completion 和 complete 两种相对的操作来进行同步操作，我们**在此将一直等待，直到连接建立成功后返回**，为防止死锁，我们在此处设置了 timeout，超过时间则默认返回失败。

根据上述的执行逻辑，所实现的代码如下：

初始化队列结构体

```
static int sswap_rdma_init_queue(struct sswap_rdma_ctrl *ctrl, int idx)
{
  struct rdma_queue *queue;
  int ret;

  pr_info("start: %s\n", __FUNCTION__);

  queue = &ctrl->queues[idx];
  queue->ctrl = ctrl;
  init_completion(&queue->cm_done);//done=0
  atomic_set(&queue->pending, 0);
  spin_lock_init(&queue->cq_lock);//初始化成unlocked状态
  queue->qp_type = get_queue_type(idx);

  //创建一个cm_id, 类似于socket,event_handler用于处理事件
```

```
  queue->cm_id = rdma_create_id(&init_net, sswap_rdma_cm_handler, queue,
      RDMA_PS_TCP, IB_QPT_RC);
  if (IS_ERR(queue->cm_id)) {
    pr_err("failed to create cm id: %ld\n", PTR_ERR(queue->cm_id));
    return -ENODEV;
  }
  queue->cm_error = -ETIMEDOUT;
  //在resolve后会产生ADDR_RESOLVED事件
  ret = rdma_resolve_addr(queue->cm_id, &ctrl->srcaddr, &ctrl->addr,
      CONNECTION_TIMEOUT_MS);
  if (ret) {
    pr_err("rdma_resolve_addr failed: %d\n", ret);
    goto out_destroy_cm_id;
  }

  //等待completion完成
  ret = sswap_rdma_wait_for_cm(queue);
  if (ret) {
    pr_err("sswap_rdma_wait_for_cm failed\n");
    goto out_destroy_cm_id;
  }
  return 0;

out_destroy_cm_id:
  rdma_destroy_id(queue->cm_id);
  return ret;
}
```

分配 ib 队列并解析路由信息

和用户态 client 程序一样，在解析到 ADDR_RESOLVED 后，需要调用相应的事件处理函数 sswap_rdma_addr_resolved：函数执行流程如下：

1. 为队列结构体分配 RDMA device，该 device 是从 cm_id 中获取的；除此之外，还需要为队列分配 protect domain。这部分的工作在函数 sswap_rdma_get_device 中完成。

2. 创建并分配 ib 队列内容 (sswap_rdma_create_queue_ib 中完成)：在先前的初始化队列工作中，我们只是为 rdma_queue 结构体的 queue pair(qp，由 send queue 和 receive queue 组成) 以及 complete queue(cq) 分配了空间，接下来需要真正创建它们，在此之前先介绍 cq：

   rdma 操作和远端进行通信的过程是异步的，也就是说在 read/write 函数返回后，并不代表读写操作已经完成了。当操作真正完成后，会产生 cqe(complete queue element) 通过 cq 返回给上层；通过轮询 cq 获取相应的完成事件，才可确定某项操作是否已经完成。

   在此时我们调用 ib_alloc_cq 函数分配 cq 并绑定到 RDMA 设备上，轮询 cq 的常用操作也分为两种：IB_POLL_DIRECT 和 IB_POLL_SOFTIRQ，分别代表直接轮询 (不切换上下文) 和软中断轮询 (推迟到软中断上下文中进行轮询) 两种模式，可用来进行同步读取于异步读取，在此时我们选择以直接轮询方式初始化 cq。

创建 ib 队列

```
static int sswap_rdma_create_queue_ib(struct rdma_queue *q)
{
  struct ib_device *ibdev = q->ctrl->rdev->dev;
  int ret;
  int comp_vector = 0;
  pr_info("start: %s\n", __FUNCTION__);

  q->cq = ib_alloc_cq(ibdev, q, CQ_NUM_CQES,
  comp_vector, IB_POLL_DIRECT);

  if (IS_ERR(q->cq)) {
    ret = PTR_ERR(q->cq);
    goto out_err;
  }

  ret = sswap_rdma_create_qp(q);
  if (ret)
    goto out_destroy_ib_cq;
  return 0;
    //省略释放操作。。。
}
```

在创建完成队列之后，调用 sswap_rdma_create_qp 进而调用 rdma_create_qp 来进行 qp 的创建，该操作也会将 qp 绑定到特定的 cm_id 上。

3. 接着与用户态相似，传入 cm_id 调用 rdma_resolve_route 函数解析路由，这是在连接之前的必要操作。

建立 RDMA 连接

解析路由成功并触发 RDMA_CM_EVENT_ROUTE_RESOLVED 事件后，设置好连接的参数，就可在 sswap_rdma_route_resolved 进行连接了，建立连接过程也是参考内核中 nvme 相关的 rdma 连接。初次尝试使用与用户态一致的 rdma_connect 函数，但是陷入了死等待的状态且无法卸载驱动，但是在 5.4 版本的内核中执行该函数却能连接成功。在 6.1 版本下进入内核/drivers/infiniband/core/cma.c 查看该函数的实现如下：

rdma_connect 函数的实现

```
int rdma_connect(struct rdma_cm_id *id, struct rdma_conn_param *conn_param)
{
  struct rdma_id_private *id_priv =
    container_of(id, struct rdma_id_private, id);
  int ret;

  mutex_lock(&id_priv->handler_mutex);
  ret = rdma_connect_locked(id, conn_param);
  mutex_unlock(&id_priv->handler_mutex);
  return ret;
```

```
}
```

该函数首先会根据 cm_id 在结构体 rdma_id_private 中的位置来获取 priv 结构体，再为其加上锁，接着调用函数 rdma_connect_locked 来真正进入连接。通过在内核中加入打印信息确认了问题在于此处 id_priv 一直无法上锁。我们对比了 5.4 与 6.1 的内核，发现在 5.4 的内核版本中并没有 rdma_connect_locked 函数，仅有 rdma_conncet 函数。再详细查看内部实现，其与 6.1 版本中 rdma_connect_locked 函数的实现一致，于是判断此处上锁这是 6.1 内核的新特性。id_priv 是在连接时封装 rdma_conn_param 结构体所传递的，我们将其中的 private_data 设置为了 NULL，如下所示：

建立连接

```
static int sswap_rdma_route_resolved(struct rdma_queue *q,
    struct rdma_conn_param *conn_params)
{
  struct rdma_conn_param param = {};
  int ret;

  param.qp_num = q->qp->qp_num;
  param.flow_control = 1;
  param.responder_resources = 16;
  param.initiator_depth = 16;
  param.retry_count = 7;
  param.rnr_retry_count = 7;
  param.private_data = NULL;
  param.private_data_len = 0;

  //进行addr_resolved和route_resolved后，进行connect，
  ret = rdma_connect_locked(q->cm_id, &param);
  if (ret) {
    pr_err("rdma_connect failed (%d)\n", ret);
    sswap_rdma_destroy_queue_ib(q);
  }
  return 0;
}
```

未对其进行初始化，自然无法上锁，于是直接**采用 rdma_connect_locked 来替代**，忽略了加锁的过程，经此修改过后连接建立成功。

在建立连接后。捕捉到 RDMA_CM_EVENT_ESTABLISHED 事件，complete 当时进行等待的完成量，进行等待的 sswap_rdma_init_queue 函数成功返回。整个内核态 RDMA 连接建立的流程如下图5.16所示：

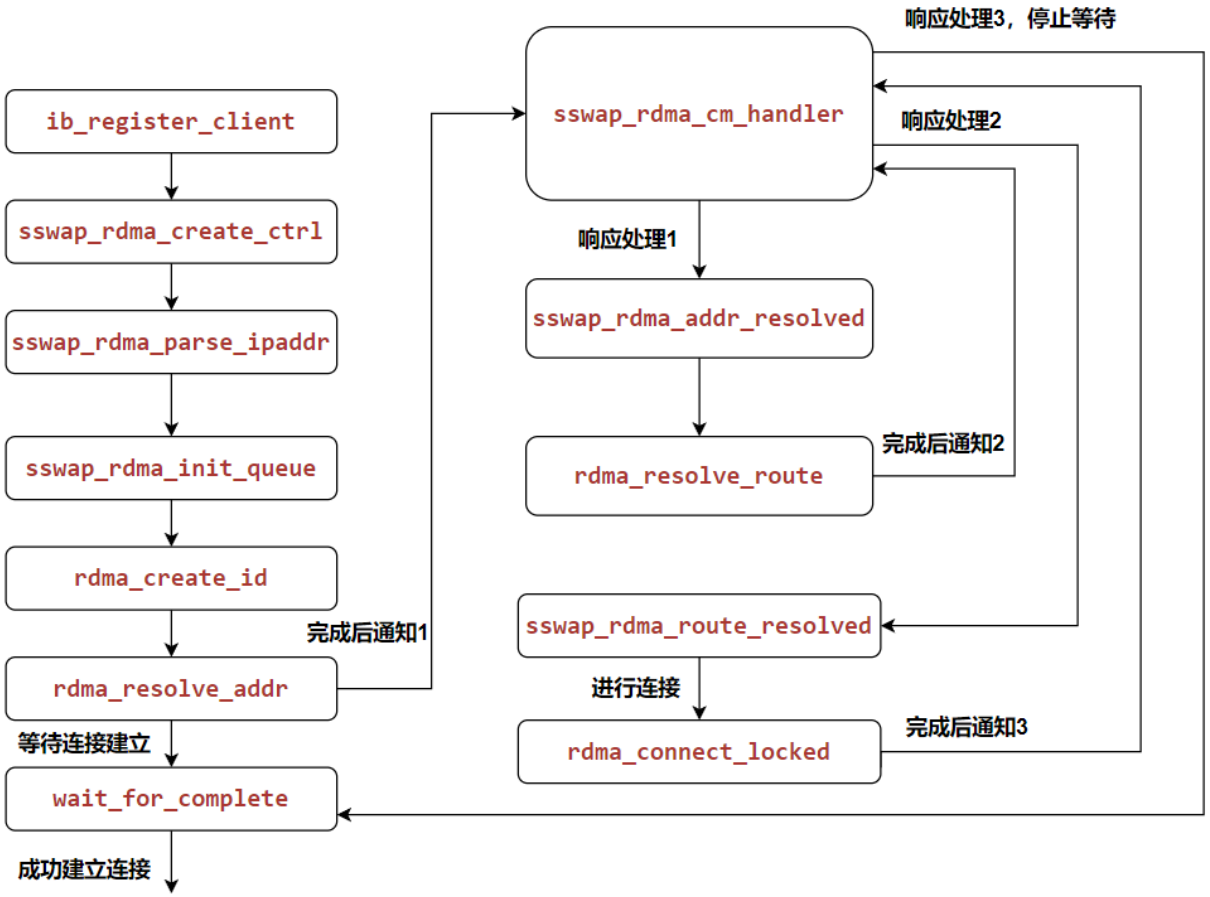


图 5.16: 内核态连接建立流程图

5.7.2 内核态 RDMA 连接的销毁

销毁连接的流程相对固定：首先调用 rdma_disconnect 函数关闭连接，由于我们在本地没有注册相关 memory region，不需要对其资源进行释放，只需要依次释放队列的相关资源即可，使用的为内核态 (以 ib 开头的) 相关接口。

销毁连接

```
static void sswap_rdma_free_queue(struct rdma_queue *q)
{
  rdma_destroy_qp(q->cm_id);
  ib_free_cq(q->cq);
  rdma_destroy_id(q->cm_id);
}
```

5.7.3 接收 memory region 以及 rkey

在建立连接之后，尝试从远程接收 memory region 以及 rkey(被封装在 ctrl->servermr 中)，在后续使用 read 和 write 操作时将会使用 rkey 在远端进行认证，将 page 以远端 cpu 无感知的方式换出到对应的 memory region 中。接收这些信息采用的时 Recieve 的通信方式主动接收。在接收前，**需要为 RDMA device 创建一块 DMA 区域，并将此区域与 servermr 地址之间添加映射关系**，这个过

程只需要由 RDMA device->kernel memory 单向传输即可。一旦从远端收到了 queue element (req), dma 就会将其从 RDMA 设备中复制到对应的内存地址中。这一部分的内容在函数 get_req_for_buf 中实现。

映射 dma 区域到 memorymr

```
inline static int get_req_for_buf(struct rdma_req **req, struct ib_device *dev,
        void *buf, size_t size,
        enum dma_data_direction dir)
{
  int ret;
  ret = 0;
  *req = kmem_cache_alloc(req_cache, GFP_ATOMIC);
  if (unlikely(!req)) {
    pr_err("no memory for req\n");
    ret = -ENOMEM;
    goto out;
  }
  init_completion(&(*req)->done);

  (*req)->dma = ib_dma_map_single(dev, buf, size, dir);
  if (unlikely(ib_dma_mapping_error(dev, (*req)->dma))) {
    pr_err("ib_dma_mapping_error\n");
    ret = -ENOMEM;
    kmem_cache_free(req_cache, req);
    goto out;
  }
  ib_dma_sync_single_for_device(dev, (*req)->dma, size, dir);
out:
  return ret;
}
```

接着就可以通过 ib_post_recv 操作，发送请求到接收队列中 (由对方 Send 的内容会存放在 receive queue 中，通过 ib_post_recv 操作提取)。创建 work request 来向 RDMA 设备发起接收请求，sge(scatter gather element) 用来存放接收内容的位置: 我们将其设置为先前创建的 dma 区域，local key 用来提供访问对应区域的许可。具体实现如下:

接收远程信息

```
static int sswap_rdma_post_recv(struct rdma_queue *q, struct rdma_req *qe,
  size_t bufsize)
{

  struct ib_recv_wr wr = {};
  struct ib_sge sge;//sge实际上是一段内存区域的描述符
  int ret;

  sge.addr = qe->dma;
  sge.length = bufsize;
  sge.lkey = q->ctrl->rdev->pd->local_dma_lkey;
```

```
12
13    wr.next     = NULL;
14    wr.wr_cqe   = &qe->cqe;
15    wr.sg_list  = &sge;
16    wr.num_sge  = 1;
17
18    //首先要通过cpu感知的recv操作获取远程的mr（远端主动发送本端主动接收）
19    ret = ib_post_recv(q->qp, &wr, NULL);
20    if (ret) {
21      pr_err("ib_post_recv failed: %d\n", ret);
22    }
23    return ret;
24  }
```

发送请求成功并不代表着接收到了对应的数据，这是个异步的过程。实际上处理完该请求后，会在完成队列 (cq) 中产生 cqe，我们可以通过调用 ib_process_cq_direct 以轮询 cq 来确认是否完成了接收过程。当轮询 cq 时，会执行每一个 cqe 预先设置好的 cqe.done 函数。我们可以在发送请求前设置完成量 completion 用于同步。done 函数中触发 complete 操作，发送请求后调用 wait_for_complete 后不断从 cq 中轮询，若存在 ceq 则执行 done 函数。当请求被完成并在 done 函数里释放完成量后，结束轮询操作。我们特地为轮询的线程设置了延时滞后，以便于请求的执行与 dma 的传输。结束轮询后 ctrl->memorymr 中已经得到了远端的数据。从此以后，我们不再需要主动从 RDMA driver 里获取内容了，于是可以顺带在 done 函数中释放之前分配的 DMA 区域。上述描述的调用流程如下图5.17:

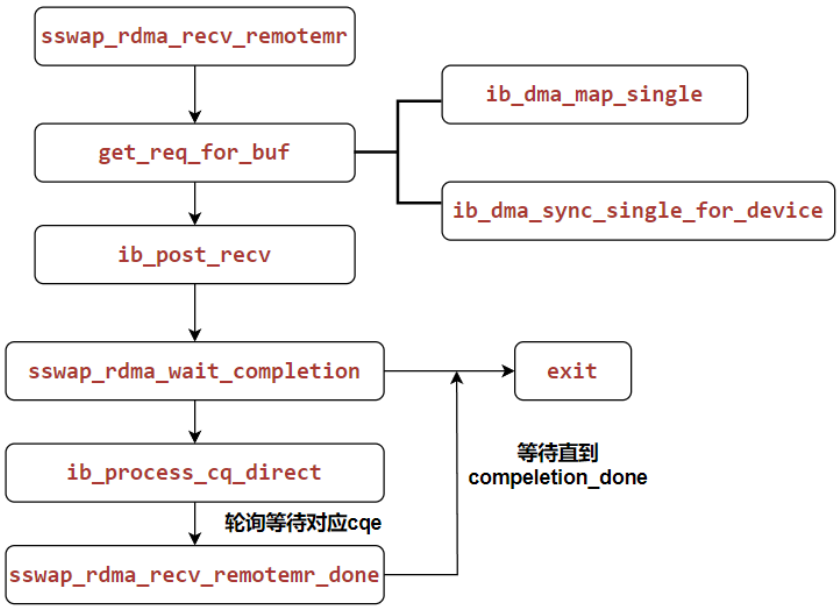


图 5.17: 接收远端数据流程

至此，完成了内核态连接建立和销毁的全过程。

5.8 RDMA 通信: 利用 read 和 write 实现换页

本节中的内容为基于 RDMA 以页为粒度和远程内存之间的页面交换过程。和上面我们提及的操作都不同，read/write 只需要本地 cpu 发送请求即可，远端是无感知的，这部分的内容我们通过重写 sswap_rdma_read_sync 和 sswap_rdma_write 来实现。读和写操作总体的流程相差不大，我们以先以写操作为例（页面总是要被先换出再换入）。

5.8.1 RDMA:write 操作实现页换出

首先，和 DRAM 中一致，需要做越界大小检查，保证换出页的偏移地址在远程内存所提供的范围内，接着找到当初为 CPU 核心所对应的写队列结构体，后续将使用其向 send queue 发送请求，流程分为以下几步：

1. 查看此 queue 的状态，如果当前队列中未完成的 request 较多 (通过 q->pending 来记录)，将会先调用 ib_process_cq_direct 处理一些 cqe 以防止队列阻塞过久。

2. 与 recv 操作一样，需要先给换出的 page 映射一块 RDMA device 能访问的 dma 地址，和 recv 相比传输方向相反（DMA_TO_DEVICE），在处理换出请求后能够绕过 CPU 直接复制页到 RDMA 设备中。

3. 设置好 cqe.done，在处理完请求并轮询 cq 时调用该函数：根据 work request 的 status 来判断是否换出成功，若成功则减少 q->pending，并释放先前创建的 DMA 映射。

4. 接下来可以向 send queue 队列发送请求，同样以 work request 的方式发送，选择其 opcode 为 IB_WR_RDMA_WRITE 表示使用 RDMA 的写操作，remoteaddr 为从 memorymr 中得到的 baseaddr 加上页的偏移量 roffset，接着就可以调用 ib_post_send 发送请求，将 q->pending 增加，实现如下：

发送请求到 send queue

```
1  inline static int sswap_rdma_post_rdma(struct rdma_queue *q, struct rdma_req *qe,
2    struct ib_sge *sge, u64 roffset, enum ib_wr_opcode op)
3  {
4    struct ib_rdma_wr rdma_wr = {};
5    int ret;
6
7    BUG_ON(qe->dma == 0);
8
9    sge->addr = qe->dma;
10   sge->length = PAGE_SIZE;
11   sge->lkey = q->ctrl->rdev->pd->local_dma_lkey;
12
13   rdma_wr.wr.next     = NULL;
14   rdma_wr.wr.wr_cqe   = &qe->cqe;
15   rdma_wr.wr.sg_list = sge;
16   rdma_wr.wr.num_sge = 1;
17   rdma_wr.wr.opcode   = op;
18   rdma_wr.wr.send_flags = IB_SEND_SIGNALED;
19   rdma_wr.remote_addr = q->ctrl->servermr.baseaddr + roffset;
20   rdma_wr.rkey = q->ctrl->servermr.key;
```

```
21
22        atomic_inc(&q->pending);
23        //第三个参数在linux6.1下应设置为NULL
24        ret = ib_post_send(q->qp, &rdma_wr.wr, NULL);
25        if (unlikely(ret)) {
26          pr_err("ib_post_send failed: %d\n", ret);
27        }
28        return ret;
29      }
```

5. 在发送完请求后，轮询 cq，处理其中的 cqe，并调用相应的 cqe.done 函数，直到 q->pending 为 0。

上述流程可表示为下图5.18

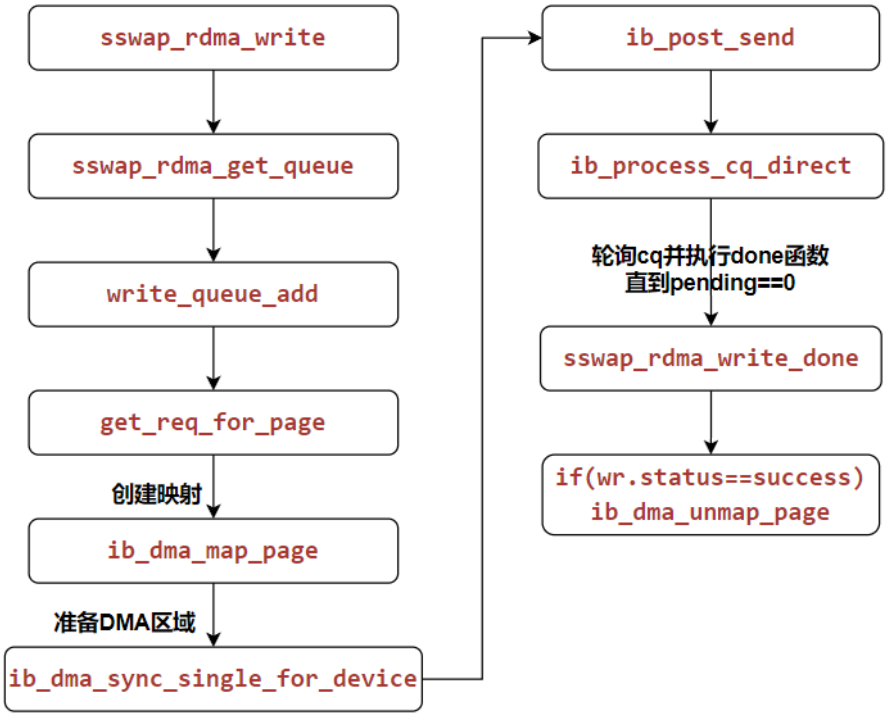


图 5.18: write 操作函数执行路径

5.8.2 RDMA:read 操作实现页换入

初始阶段，实现的 read 操作是同步的，也就是说，直到查询到目标页面被换入，函数才会返回。它和 write 的过程相差不大，只是需要另外增加一些条件判断：在读取页面之前，需要做一些检查：需要保证该页在 SwapCache 且被锁定，并且不是 uptodate 的状态。接下来的步骤和 write 相差不大，有以下几点不同：

1. 在映射 dma 区域以及解除映射时，由于是从 RDMA driver 读入到内存中，传入的 direction 参数为 DMA_FROM_DEVICE。
2. 在发送请求时，仍然调用的是 ib_post_send 函数，只是其中 wr.opcode 参数需要设置为 IB_WR_RDMA_READ。

3. 轮询 cq 并执行 sswap_rdma_read_done，在 wr 执行成功后，需要解锁该页面，并将该页面的状态设置为 uptodate。这部分的操作原本是在 frontswap.c 中 frontswap_load 函数调用完 ops->load 函数中执行的，在同步读的情况下这么做没问题，但若是在异步读的情况下，ops->load 函数返回并不代表页面真正读取完成，所以为了后续扩展方便，将这两个操作统一放在 sswap_rdma_read_done 中执行。

程序执行流程与 write 操作基本一致，在此处不再赘述。至此，RDMA BACKEND 实现，可以进行相应的测试工作。

6 系统测试情况

6.1 测试说明

在初赛阶段，我们只进行了**系统实现的正确性测试**。我们模拟内存访问的负载，通过随机和周期性地访问内存页面，以测试内存的性能和行为。采用 cgroup 限制测试程序的内存使用以强制触发 swap 操作将页换出，进而检验实现的 frontswap BACKEND 的正确性。测试程序为 pagewalker.c，测试的原理为初始化随机种子，对给定的页面范围进行随机读写操作，在此期间，程序被允许使用的内存显著小于所需使用的大小，若 frontswap 实现正确，我们将通过 dmesg 在内核缓冲区中看到关键路径上的输出 (打开预先定义好的 DEBUG_MODE 宏)。

简单说明 test 文件夹下的测试脚本 memory_limit_test.sh：我们在 cgroup 路径下新建一个 group，在其中 memory.max 文件里添加程序的内存限制范围，我们设置为 128M，将当前进程 id 写入 cgroup.procs 中，表明当前进程受此 group 管理，接着编译并运行测试程序，程序执行完毕后输出这段时间内 cgroup 内存的峰值，用来判断是否触发了 swap：

测试脚本

```
#!/bin/bash
echo $$ >> /sys/fs/cgroup/cgroup.procs
sleep 1
cgdelete -r -g cpuset,cpu,io,memory,hugetlb,pids,rdma,misc:/yuri
echo $$
if [ ! -d "/sys/fs/cgroup/yuri/" ];then
  mkdir /sys/fs/cgroup/yuri
else
  echo "cgroup yuri already exists"
fi
echo "+memory" >> /sys/fs/cgroup/yuri/cgroup.subtree_control

if [ ! -d "/sys/fs/cgroup/yuri/pagerank_150M/" ];then
  mkdir /sys/fs/cgroup/yuri/pagerank_150M
else
  echo "cgroup yuri/pagerank_150M already exists"
fi

echo 134217728 > /sys/fs/cgroup/yuri/pagerank_150M/memory.max
echo "set memory.max to"
cat /sys/fs/cgroup/yuri/pagerank_150M/memory.max
```

```
echo "adding current shell to pagerank_150M"
echo $$ | sudo tee /sys/fs/cgroup/yuri/pagerank_150M/cgroup.procs
gcc pagewalker.c -lm -O0 -o pagewalker
./pagewalker
echo "memory.peak is:"
cat /sys/fs/cgroup/yuri/pagerank_150M/memory.peak
```

6.2 DRAM BACKEND 正确性测试

执行命令安装模块到内核中并进行测试，进入项目文件夹下：

安装 DRAM BACKEND

```
#dev-rdma文件夹下
make BACKEND=DRAM
sudo insmod fastswap_dram.ko
sudo insmod fastswap.ko
#test文件夹下
sudo ./memory_limit_test.sh
```

在执行完模块安装后，dmesg 在内核缓冲区中可看到如下输出6.19，表明安装成功。

```
[  228.033238] fastswap_dram: start: sswap_dram_init_module
[  228.033240] fastswap_dram: will use new DRAM backend
[  228.257122] fastswap_dram: vzalloc'ed 4294967296 bytes for dram backend
[  228.257125] fastswap_dram: DRAM backend is ready for reqs
[  232.801122] fastswap: begin init sswap
[  232.801129] fastswap: sswap module loaded
```

图 6.19: DRAM 安装信息

运行完测试程序后，可以看到每一轮的迭代时间以及 cgroup 中的 memory.peak，达到了我们预期的 128M。进入内核缓冲区中也能发现有关键路径上的输出，如下图6.20，可以说明确实发生了 swap 操作。

```
epoch 65
epoch 66
epoch 67
epoch 68
epoch 69
epoch 70
epoch 71
epoch 72
epoch 73
epoch 74
Time measured: 10.950 seconds.
memory.peak is:
134217728
```

(a) 测试程序输出

```
[  384.622721] in sswap_store
[  384.622724] in sswap_store
[  384.622726] in sswap_store
[  384.622729] in sswap_store
[  384.622731] in sswap_store
[  384.622734] in sswap_store
[  384.622736] in sswap_store
[  384.622739] in sswap_store
[  384.622741] in sswap_store
[  384.622744] in sswap_store
[  384.622749] in sswap_store
[  384.622752] in sswap_store
[  384.622754] in sswap_store
[  384.622757] in sswap_store
[  384.622759] in sswap_store
[  384.622762] in sswap_store
```

(b) 内核缓冲区信息

图 6.20: DRAM BACKEND 测试结果

6.3 RDMA BACKEND 正确性测试

6.3.1 server 端建链操作

在进行实验之前，我们需要保证 server 的队列个数大于或等于 client 端，一组队列之间建立一个连接。我们将访问 server 端的 50000 号端口，在 server 端执行命令：

开启 server

```
#farmemserver文件夹下
make rmserver
./rmserver 50000
```

此时 server 端会出现监听信息，表示等待连接状态。注意，若此时手动 ctrl+C 终止程序，由于端口 50000 正在等待连接，再次重新运行程序会报如图6.21的错误：

```
xc@xc-HP-280-Pro-G2-MT:~/fastswap/farmemserver$ ./rmserver 50000
error: rdma_bind_addr(listener, (struct sockaddr *)&addr) failed (returned non-zero). - errno: 99
```

图 6.21: server-error

表示 rdma_bind_addr 出现错误，因为 50000 号端口在先前已经被绑定并仍然在等待连接，之前的程序并没有完全被杀死。我们查看先前程序的进程 id：

```
ps -aux | grep rmserver
```

搜索结果如图6.22，表明确实存在该进程,id 为 7923：

```
xc@xc-HP-280-Pro-G2-MT:~/fastswap/farmemserver$ ps -aux | grep rmserver
xc          7923  0.0  0.0   4084  2428 pts/2    T    20:34   0:00 ./rmserver 50000
xc         10381  0.0  0.0  12108   656 pts/2    S+   20:42   0:00 grep --color=auto rmserver
```

图 6.22: 查找 rmserver 进程 id

我们需要手动 kill 该进程：

```
sudo kill -9 7923
```

而后就可以重新使用相同端口监听。

6.3.2 client 端建链操作

在 client 端项目文件夹下执行命令：

安装 RDMA BACKEND

```
#dev-rdma文件夹下
make BACKEND=DRAM
sudo ./insmod.sh
```

执行成功后，内核缓冲区也会输出相关信息，如图6.23所示：

```
[ 3550.248821] fastswap_rdma: cm_handler msg: address resolved (0) status 0 id 0000000076806e47
[ 3550.248832] fastswap_rdma: RDMA_CM_EVENT_ADDR_RESOLVED
[ 3550.248835] fastswap_rdma: start: sswap_rdma_addr_resolved
[ 3550.248838] fastswap_rdma: start: sswap_rdma_create_queue_ib
[ 3550.249630] fastswap_rdma: start: sswap_rdma_create_qp
[ 3550.250993] fastswap_rdma: cm_handler msg: route resolved  (2) status 0 id 0000000076806e47
[ 3550.251004] fastswap_rdma: RDMA_CM_EVENT_ROUTE_RESOLVED
[ 3550.251007] fastswap_rdma: max_qp_rd_atom=16 max_qp_init_rd_atom=16
[ 3550.251010] fastswap_rdma: begin RDMA connect
[ 3550.255378] fastswap_rdma: cm_handler msg: established (9) status 0 id 0000000076806e47
[ 3550.255390] fastswap_rdma: RDMA_CM_EVENT_ESTABLISHED
[ 3550.255393] fastswap_rdma: connection established
[ 3550.255457] fastswap_rdma: start: sswap_rdma_recv_remotemr
[ 3551.269791] fastswap_rdma: servermr baseaddr=7f96689e2010, key=2093741
[ 3551.269793] fastswap_rdma: recv_remotemr_done success
[ 3551.269794] fastswap_rdma: ctrl is ready for reqs
[ 3551.279277] fastswap: begin init sswap
[ 3551.279279] fastswap: sswap_module loaded
```

图 6.23: client 端内核缓冲区输出

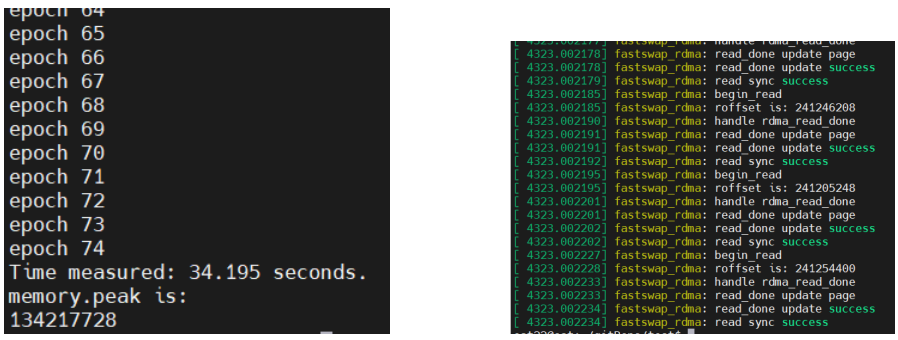
server 端程序也会在终端中输出连接相关信息，在图6.24中，可以看到对于 client 端的 6 核 CPU，每个 CPU 分配三个队列 (写队列，同步异步读队列)，共计匹配了 18 个队列并全部连接：

```
waiting for queue connection: 17
got an event
on_event
on_connect_request
attrs: max_qp=131072, max_qp_wr=32768, max_cq=16777216 max_cqe=4194303
ctrl attrs: initiator_depth=16 responder_resources=16
got an event
on_event
on_connection
done connecting all queues
```

图 6.24: server 端连接信息

6.3.3 测试程序

和 DRAM 使用相同的测试程序，在 test 文件夹下运行对应脚本，得到对应结果如图6.25，达到了所设置的内存峰值，成功触发了 swap 操作，并在 RDMA BACKEND 进行页面交换的关键路径上打印了信息，RDMA BACKEND 测试成功。



(a) 测试程序输出 (b) 内核缓冲区信息

图 6.25: DRAM BACKEND 测试结果

7 遇到的主要问题以及解决方法

在项目实现环节中，我们已经详细说明了所遇到的问题以及相应的解决方式，在此只做概述，不做详细说明。

1. 安装驱动过程中，一直无法找到适配 6.3 版本内核的 MLNX OFED 版本，在 kernel.org 的 long term support 中选择 Linux6.1 版本，找到特定版本驱动并增加选项–force-dkms 安装成功。
2. 在编写 DRAM BACKEND 的过程中，编译模块报错缺少对应符号，对相应函数添加 export_symbol 解决。
3. 安装模块后，无法使用我们编写的 DRAM 后端，未进入 frontswap 过程：修改内核 frontswap_register_ops 以适配安装。
4. 修改内核后 MLNX OFED 驱动加载失败：卸载后重新安装解决。
5. 内核态建立 RDMA 连接，调用 rdma_connect 函数出现死锁问题：查阅内核后，改用 rdma_connect_locked 函数。
6. server 端利用特定端口监听时，使用 ctrl+C 终止程序，会出现无法再次使用此端口的情况：查询所有进程发现此端口仍然被 server 程序占用且并未终止，通过 kill 指令来终止进程解决。
7. 在进行换页的过程中，发现部分 page 的 roffset 偏移量超过了远程的内存范围：roffset 对应的大小最多为 swapfile 的大小，swapfile 过大导致 roffset 可为很大 (在实验机器中为 32G)，修改 swap 交换区的大小 (4GB) 以适应远程内存的大小。

8 项目目录

```
├─ 6.1kernel          # 增加patch后的Linux6.1内核
├─ dev-rdma           # RDMA驱动
│   ├─ fastswap.c     # frontswap_ops接口实现
│   ├─ fastswap_dram.c    # DRAM BACKEND
│   ├─ fastswap_dram.h    # DRAM BACKEND
│   ├─ fastswap_rdma.c
│   ├─ fastswap_rdma.h
│   ├─ insmod.sh      # 安装脚本
│   ├─ Makefile
│   └─ rmmod.sh       # 卸载脚本
├─ farmemserver       # 远程内存服务端
│   ├─ client.c
│   ├─ Makefile
│   └─ rmserver.c
├─ README.md
└─ test
    ├─ memory_limit_test.sh  # 测试脚本
    └─ pagewalker.c          # 测试正确性程序
```

图 8.26: index

9 分工与合作

任务	分工
资料调研	许宸，曹骜天
驱动安装	许宸
环境配置	朱奕翔
用户态 RDMA 连接的建立：client/server	朱奕翔
适配 frontswap 的内核补丁修改	曹骜天
DRAM BACKEND 的实现	曹骜天
RDMA BACKEND 的实现	曹骜天

表 1: 初赛阶段的任务与分工

10 比赛收获

在初赛阶段，我们基于 FASTSWAP[1] 和 [2] 的工作，在 Linux 6.1 版本内核上的 swap 子系统实现了基于 RDMA 的远程内存，学习到了 Linux 内存管理的知识以及 RDMA 通信的相关知识，亲自动手实践开发内核相关代码，增强了内核代码编写、调试、修改等能力。

参考文献

[1] Emmanuel Amaro, Christopher Branner-Augmon, Zhihong Luo, Amy Ousterhout, Marcos K Aguilera, Aurojit Panda, Sylvia Ratnasamy, and Scott Shenker. Can far memory improve job throughput? In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[2] Chenxi Wang, Haoran Ma, Shi Liu, Yifan Qiao, Jonathan Eyolfson, Christian Navasca, Shan Lu, and Guoqing Harry Xu. {MemLiner}: Lining up tracing and application for a {Far-Memory-Friendly} runtime. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 35–53, 2022.