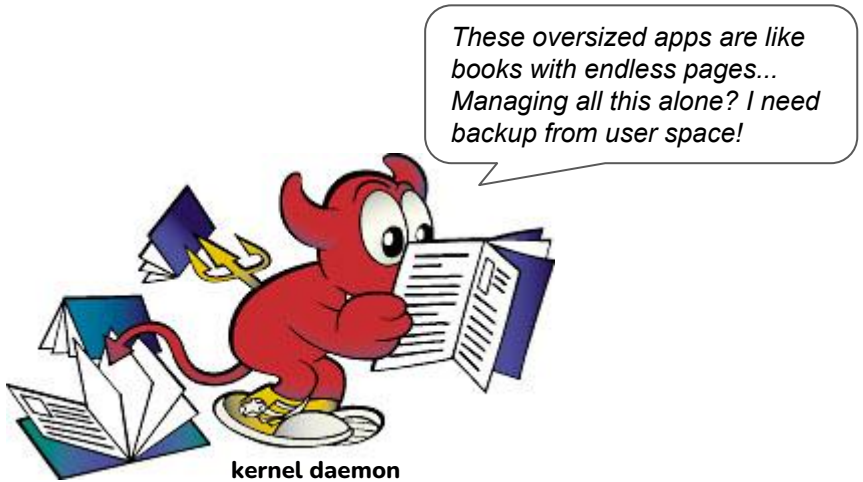
EXTMEM: Enabling Application-Aware Virtual Memory Management for Data-Intensive Applications *[USENIX ATC '24]*

https://www.usenix.org/system/files/atc24-jalalian.pdf

Presentation by Theodoros Moraitis, University of Athens



Brief Summary

Problem: Linux’s memory manager works well for the average application, but falls short for data-heavy ones with specialized memory access patterns.

Proposal: Let applications manage memory themselves — in user space — since modifying the kernel is complex and risky.

Results: This approach brings significant performance improvements with minimal system changes.

Some Background

- **Virtual Memory**, **Paging**, **Page Faults.**
  Familiar concepts – but **who actually handles them**?

- **The Linux Kernel**, through its Memory Manager module in kernel space.
  It handles page tables, page faults, and overall memory mapping.

- But…
  Can we **customize** or **replace it**? *It's hard*
  Is it **flexible** enough for every application's needs? *Not enough*

➡️ In this work, we **focus on page faults** – and how handling them in **user-space** can help.

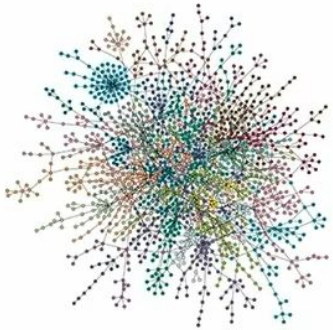
Motivation

Memory is expensive

"In recent years, DRAM has become a critical bottleneck for scaling the WSCs" – Google, 2019

"DRAM device scaling has slowed down and it accounts for over 30% of datacenter cost" – Meta, 2022

Applications are memory-hungry



Graph Processing

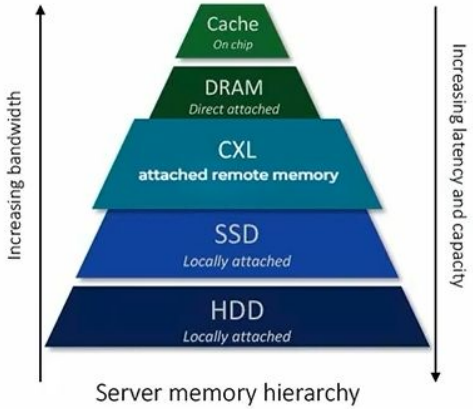
Machine Learning

Databases

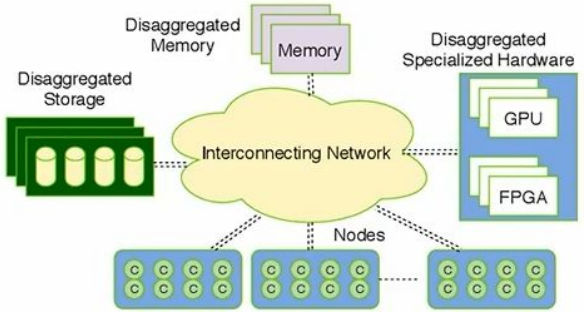
Diverse applications exhibit diverse memory patterns.

Systems are heterogenous

Memory Tiering



Disaggregation



Accelerators

Memory management policies need to be tailored to specific applications and hardware.

Key observation/insights

- The concept of a one-size-fits-all memory management policy is no longer tenable. Current kernel memory management policies are insufficient.
- Large overhead of existing userspace (userfaultfd) mechanisms.
- Rapidly changing environment, there is a need to support new scenarios but development inside the kernel is hard.

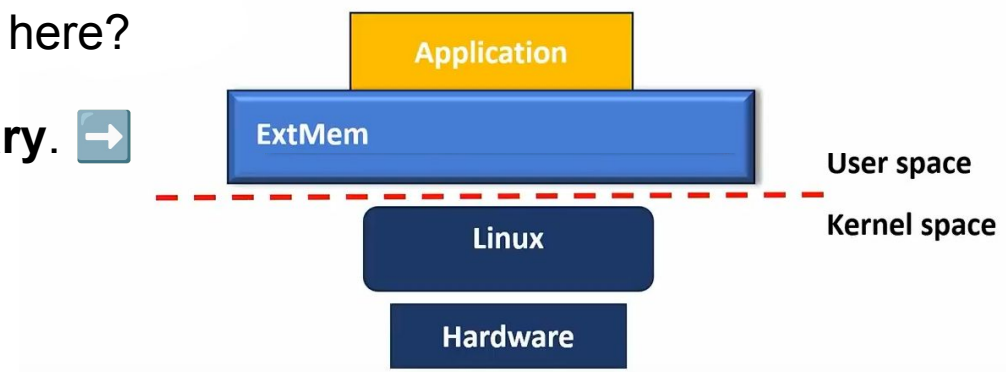
We need a solution that makes it **easy to develop new policies** for different systems/applications **while maintaining performance**!

Proposed idea

ExtMem:
a framework for application-specific memory management

What “framework” means here?

It is just a **library**. ➡️



ExtMem's User/Programmer Interface

ExtMem is a **dynamically linked library** that can be transparently loaded into the application address space via LD_PRELOAD.

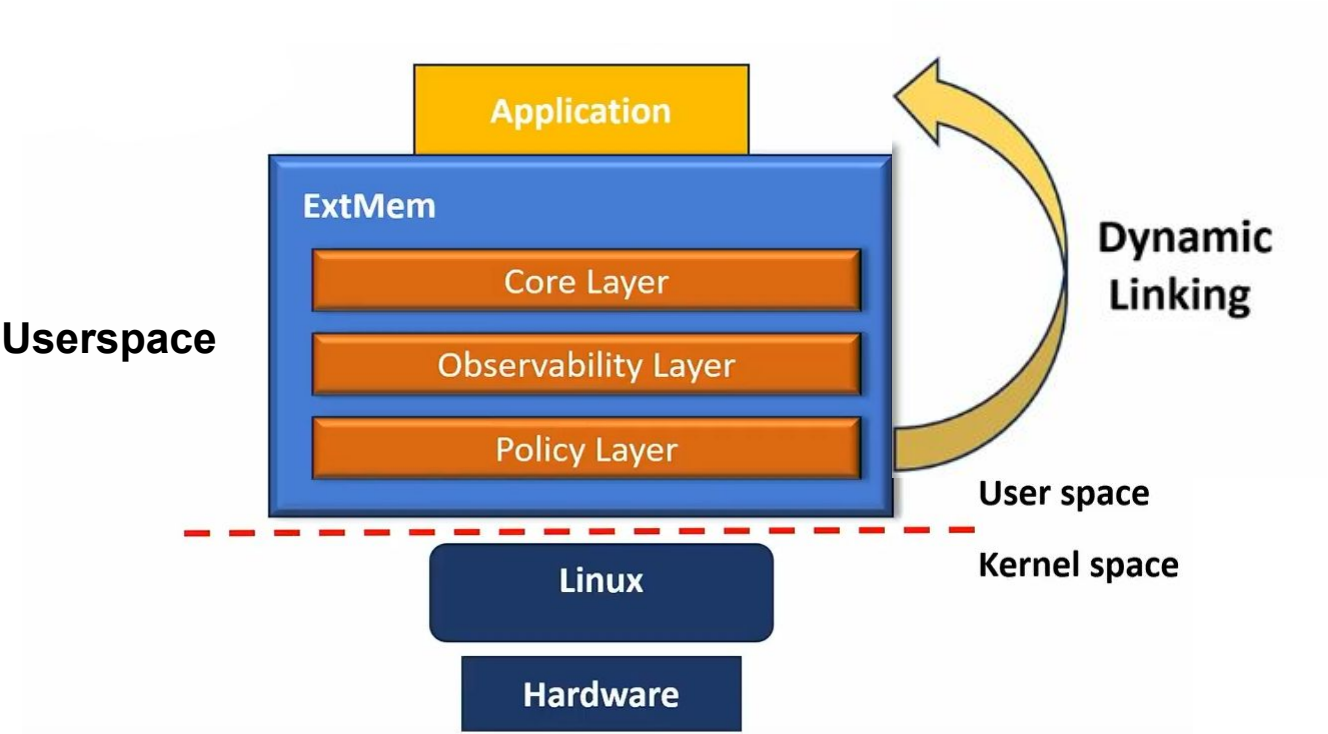
User code interacts with ExtMem in two ways:

- **Explicitly** via library API → *for developers creating custom memory managers*
- **Implicitly** via syscalls intercepted by the library → *for unmodified applications*

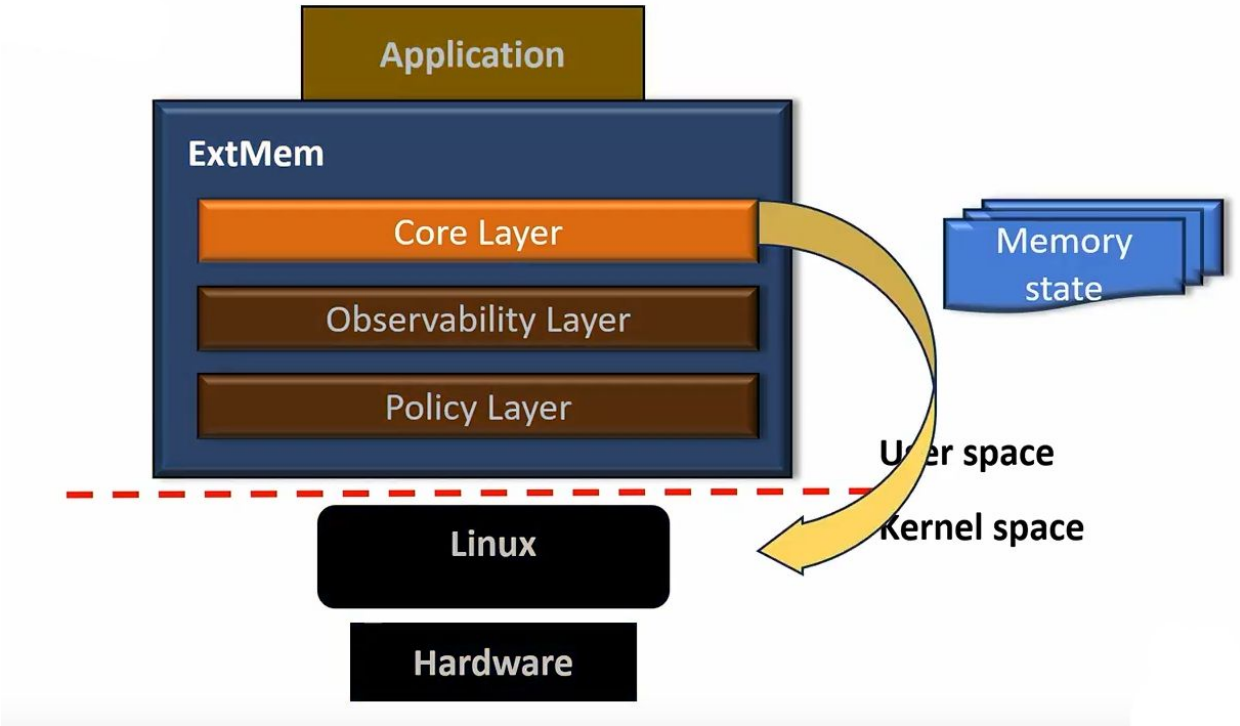
When an ExtMem core function is bound to an intercepted system call such as mmap, this function is executed upon each invocation of that system call. The bound function can itself invoke a kernel system call and/or other ExtMem functions.

ExtMem design

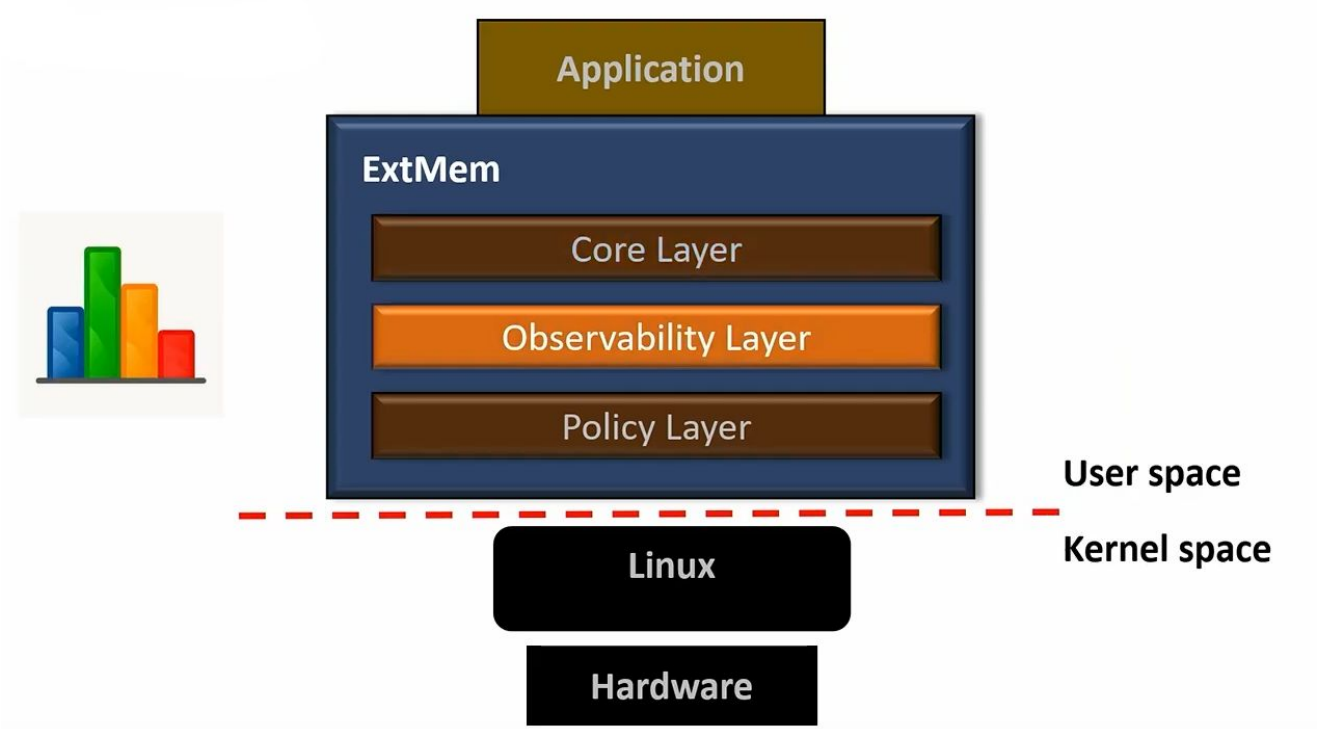
Runs in Userspace



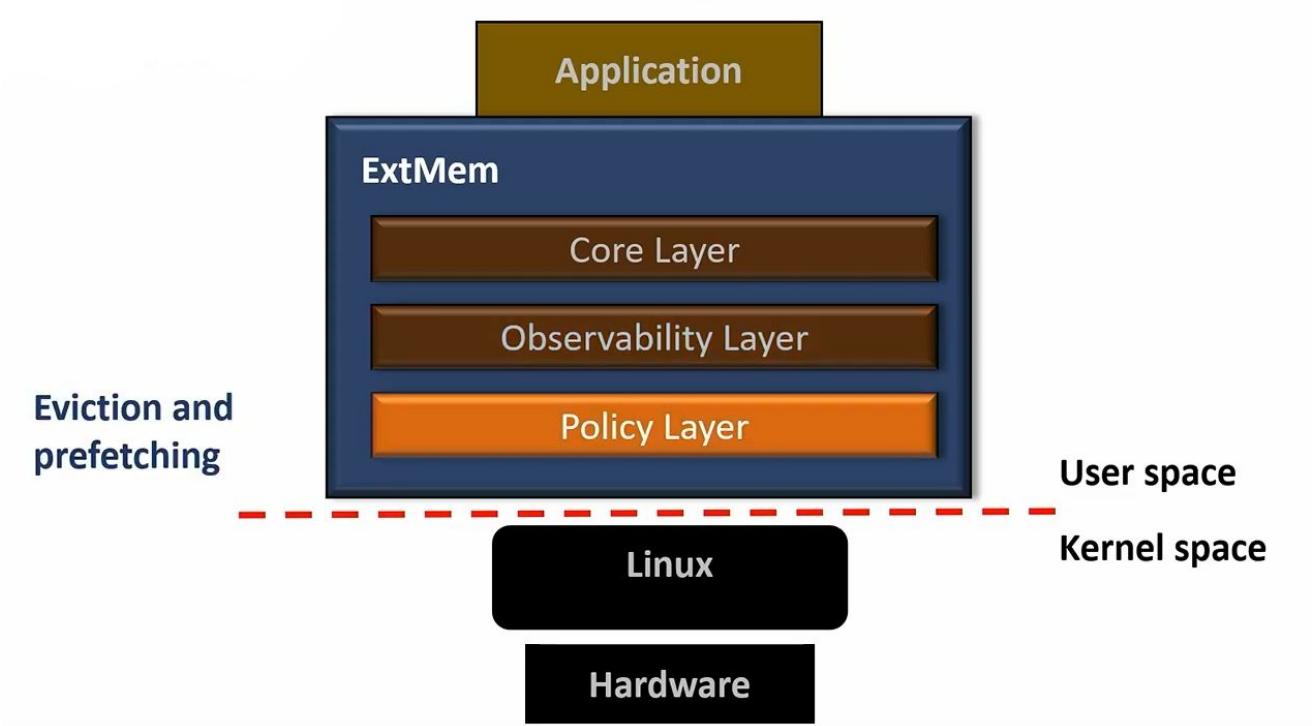
ExtMem design



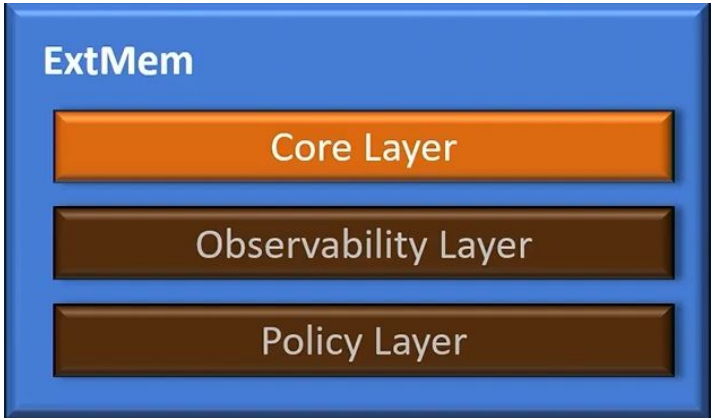
ExtMem design



ExtMem design



Core Layer



*The core layer is responsible for interacting with the kernel, and has **two** key **responsibilities**:*

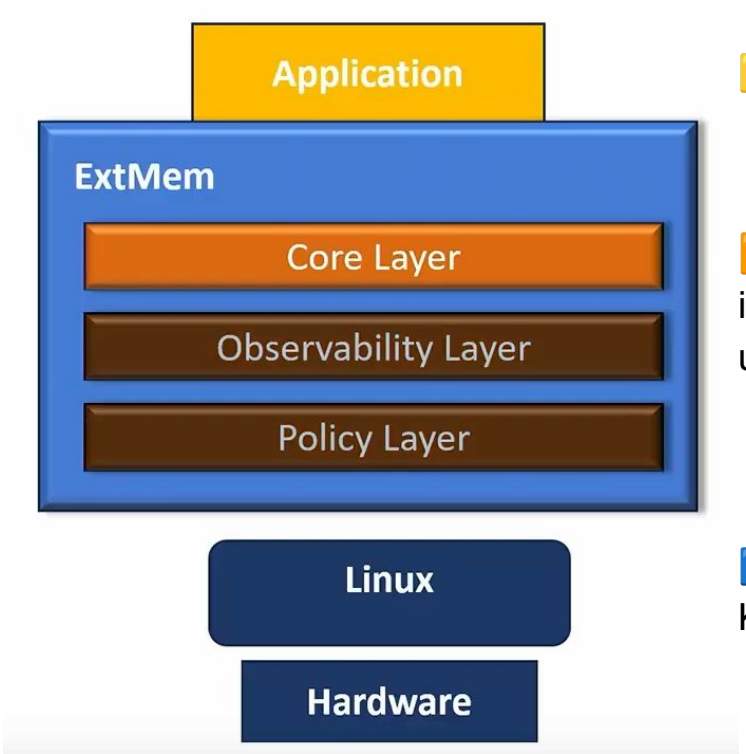
1) Register application **memory areas** for user level page fault handling
[userfaultfd is still used here]

2) Do the actual **handling** of a generated **page fault**
[userfaultfd is NOT used here, upcall instead]

Core Layer: Memory Area Registration

- Uses `LD_PRELOAD` and `libsyscall_intercept` to hook application syscalls.
- Intercepts memory-related syscalls: `mmap`, `munmap`, `madvise`, etc.
- Allocates memory as usual, then registers the region with the kernel using `userfaultfd`. Also, uses `ioctl()` to mark regions for user-space fault handling.
- Sets up ExtMem to later receive page faults via **upcalls** instead of traditional IPC.

Core Layer: Memory Area Registration



🟨 Application syscalls mmap().

🟧 ExtMem transparently intercepts the syscall, and calls userfaultfd() and ioctl().

🟦 Execution forwarded to Kernel, who maps the memory.

Handling Page Faults?

Before we see how ExtMem's Core Layer manages the page fault handling:

Which is the state-of-the-art way to do this in user-space until now?

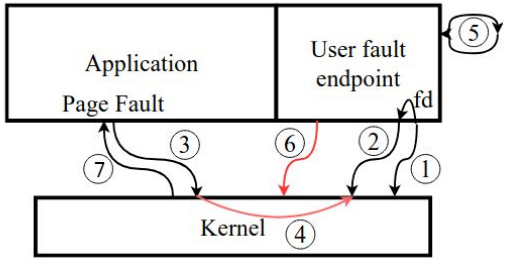


Figure 1: Userfaultfd method: 1. Application registers a memory area with the kernel and receives a file descriptor. 2. A handler thread calls a select/poll system call on the file descriptor. 3. A thread triggers a page fault and context switches into the kernel. 4. After verifying the faulting address, kernel submits the fault information to the file descriptor and blocks the faulting thread. 5. The user-level handler receives the fault, and performs the necessary IO and data preparation. 6. A user level makes a system call to map the new page and unblock the faulting thread. 7. Faulting thread goes back to continue the execution. The red arrows are inter-process-communication.

Answer: **userfaultfd**→ the problem: **not** scalable!

→It adds IPC and serialization!

Core Layer: Handling Page Faults

ExtMem's Core Layer uses an **upcall** method → very **scalable**! ✅

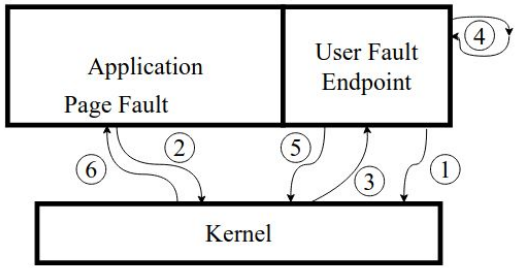
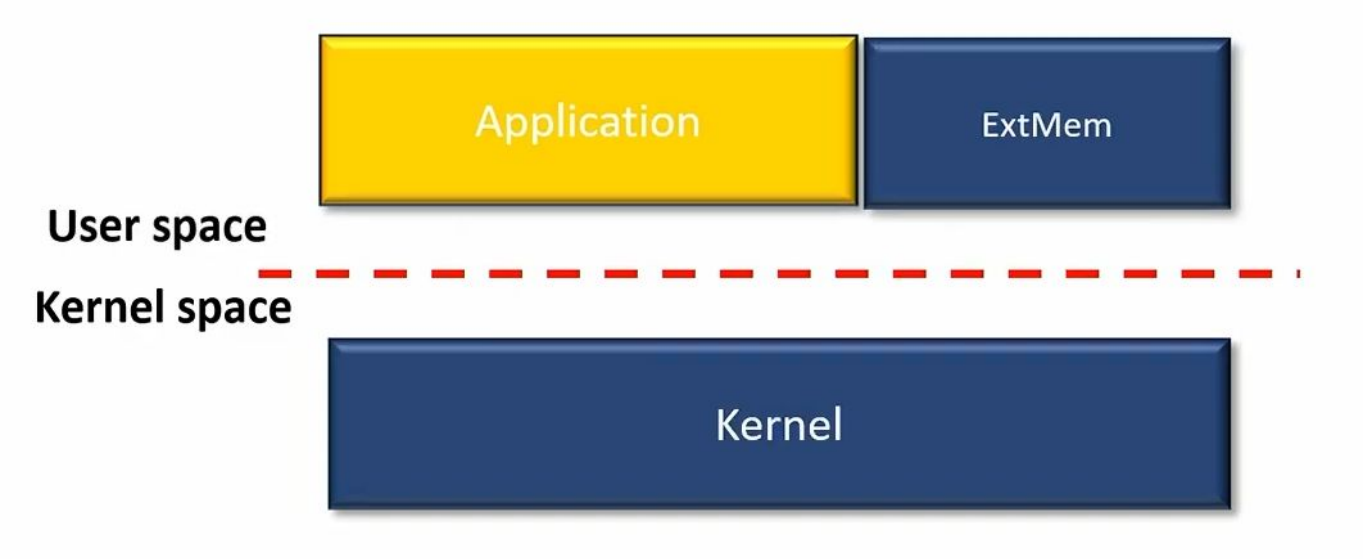
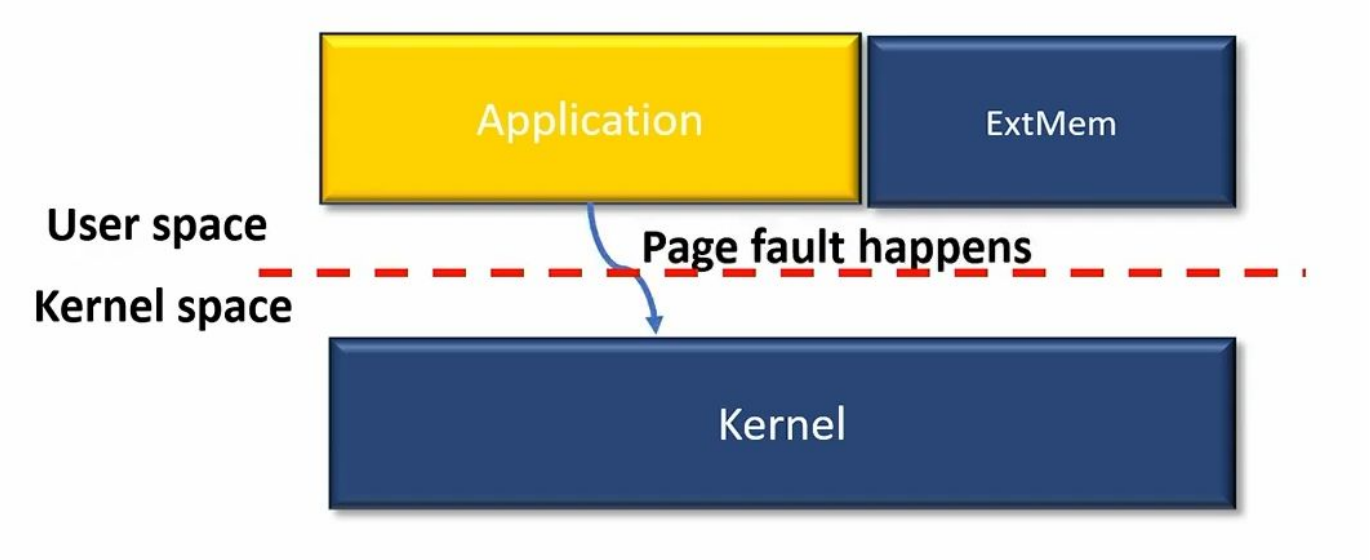


Figure 2: Upcall method based on Linux signal path: 1. Application registers a memory area with the kernel and registers an upcall handler function. 2. A thread triggers a page fault and context switches into the kernel. 3. Kernel makes an upcall by registering the upcall handler context on the faulting thread's stack 4. The faulting thread is back in userland, and executes handler code to perform the necessary IO and data preparation. 5. Faulting thread makes a system call to map the new page and return from the upcall. 6. Faulting thread goes back to continue the execution. No IPC is necessary.

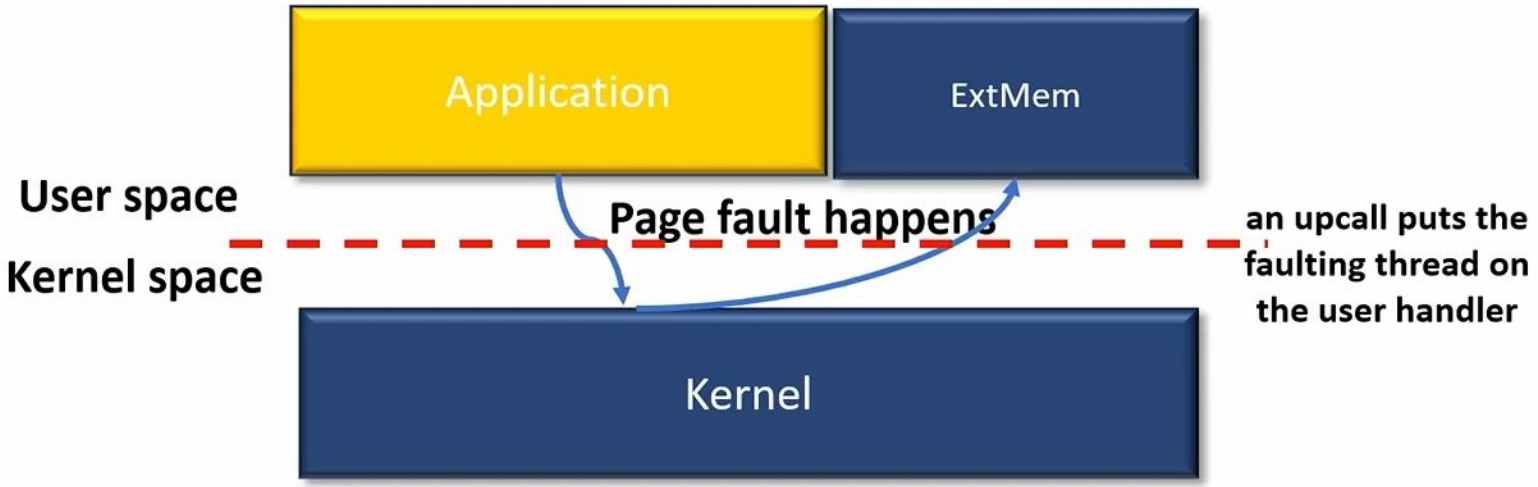
Core Layer: Handling Page Faults



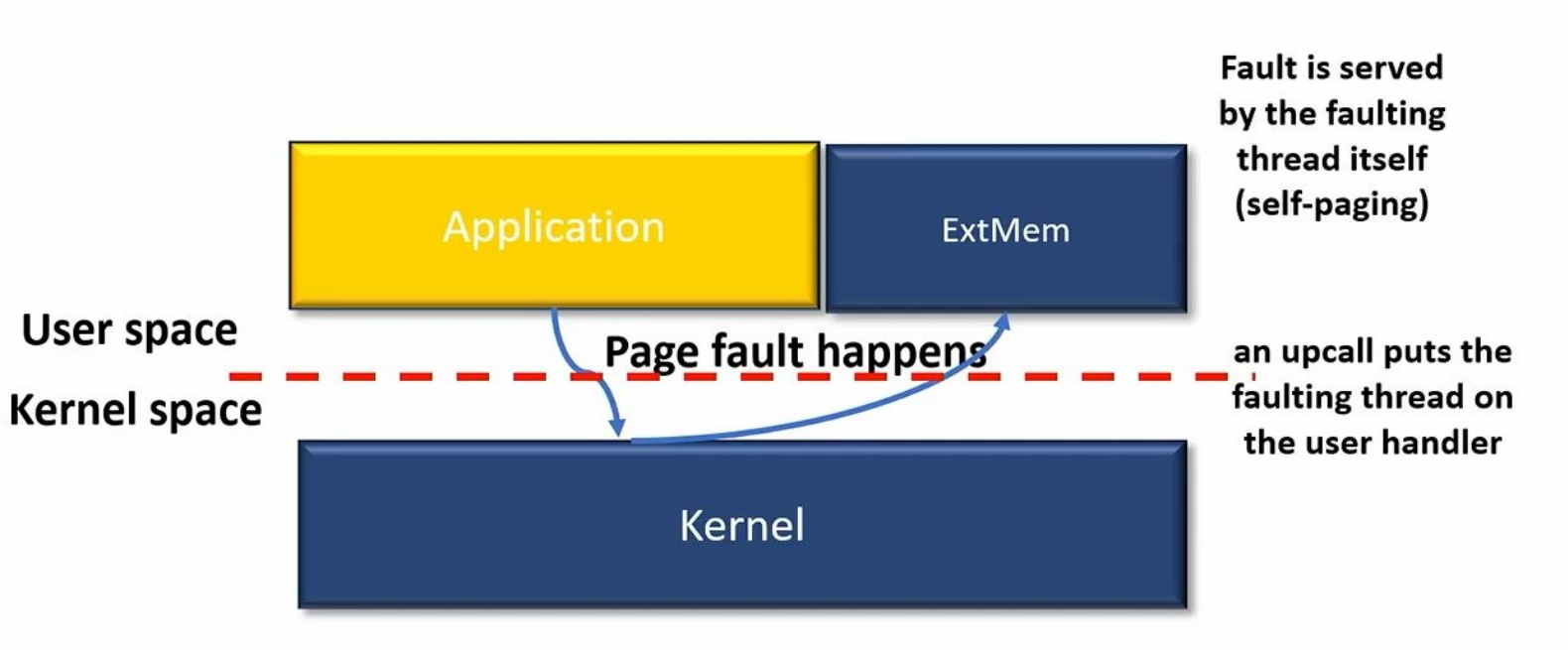
Core Layer: Handling Page Faults



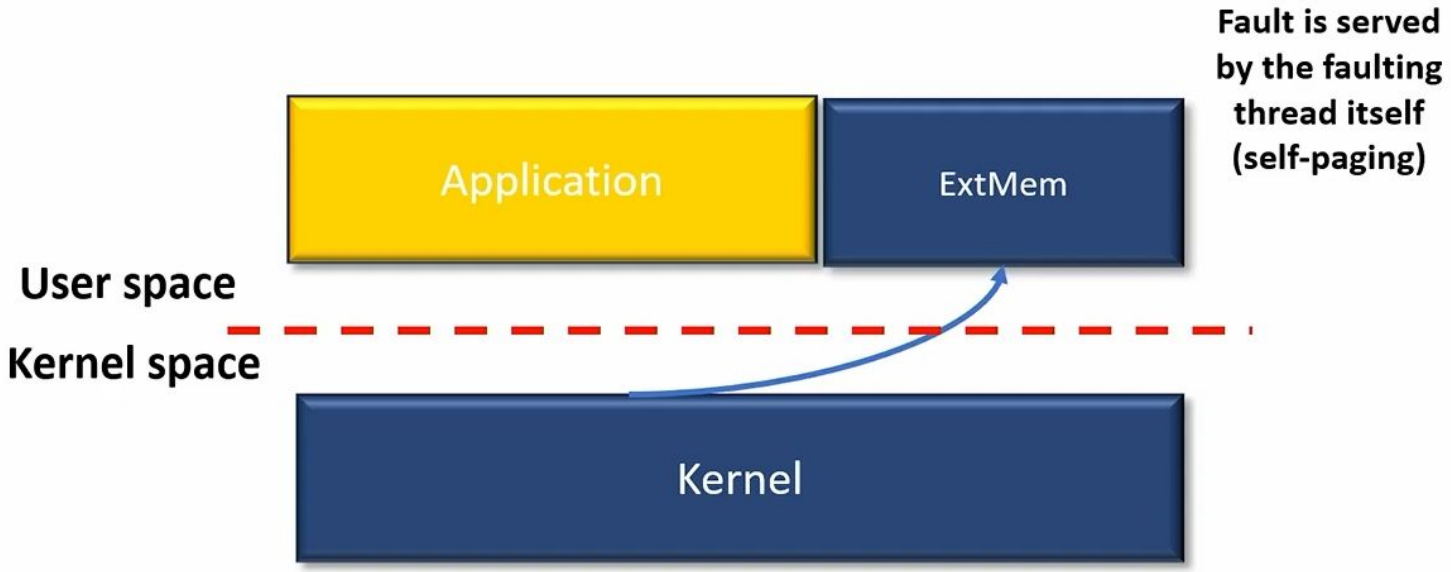
Core Layer: Handling Page Faults



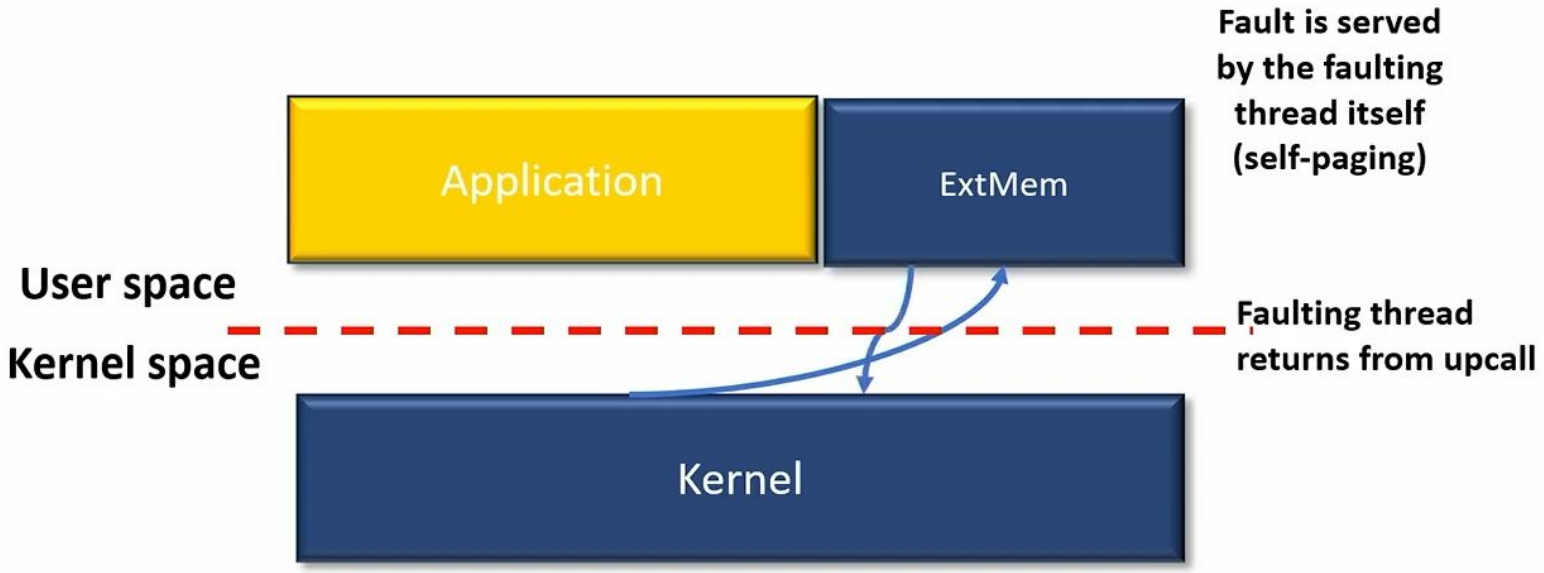
Core Layer: Handling Page Faults



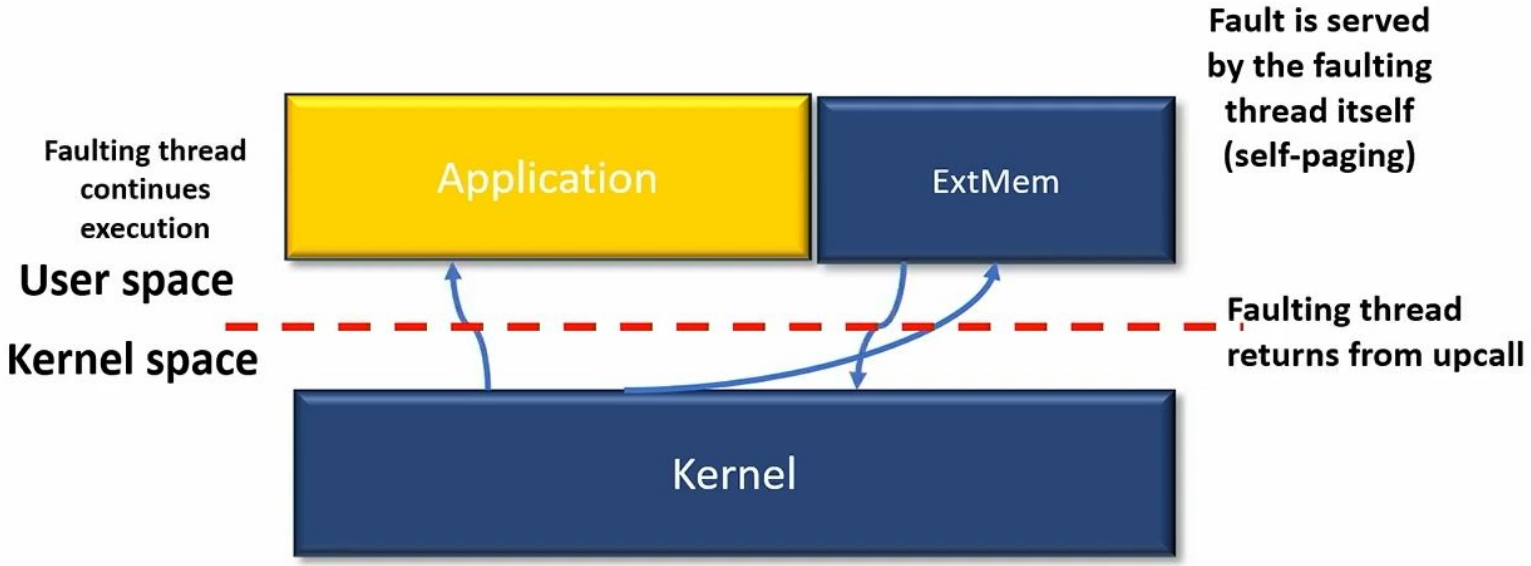
Core Layer: Handling Page Faults



Core Layer: Handling Page Faults



Core Layer: Handling Page Faults

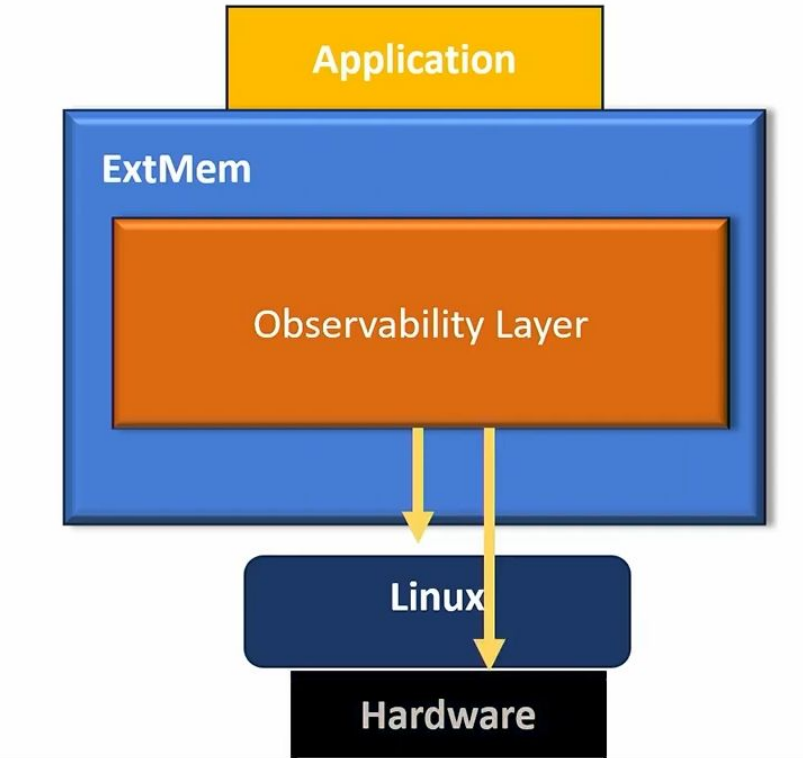


No IPC and serialization!

Observability Layer

The observability layer provides the functionality commonly used by memory managers such as access to page table access bits and to hardware counter sampling.

Observability Layer

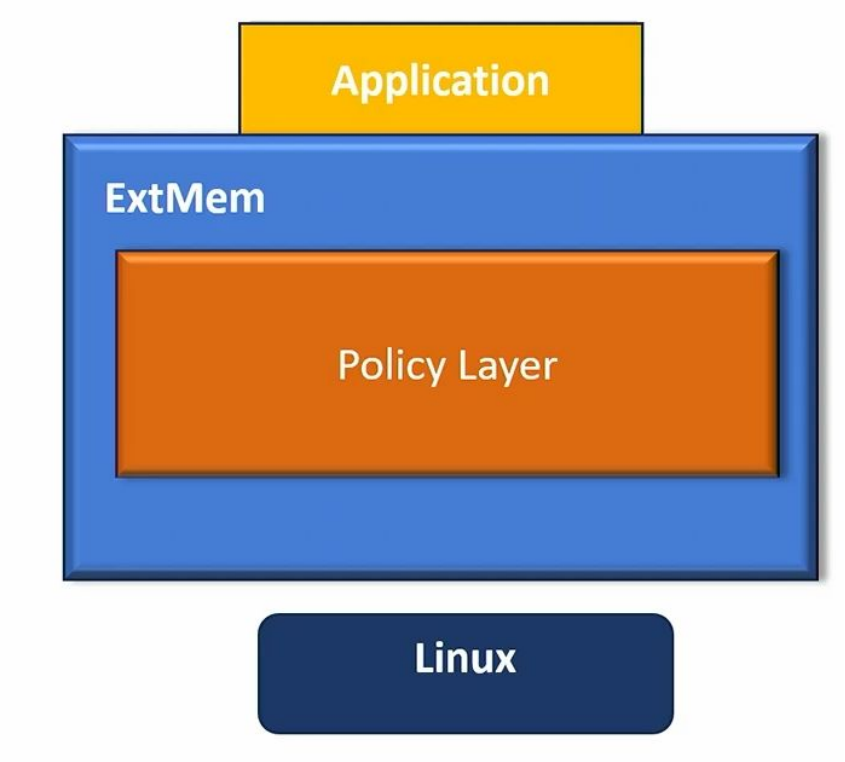


Info about:
Page fault addresses
Accessed and dirty bits
Hardware counters
…*(additional data collection features can be added, e.g. for CXL)****...***

Policy Layer

The policy layer contains implementations of policies, e.g., deciding which pages to evict, prefetch, etc.

Policy Layer



Explicit **API** for the programmer, example:
Swal_pages(...)
Swap_async(...)
Try_prefetch(...)
…

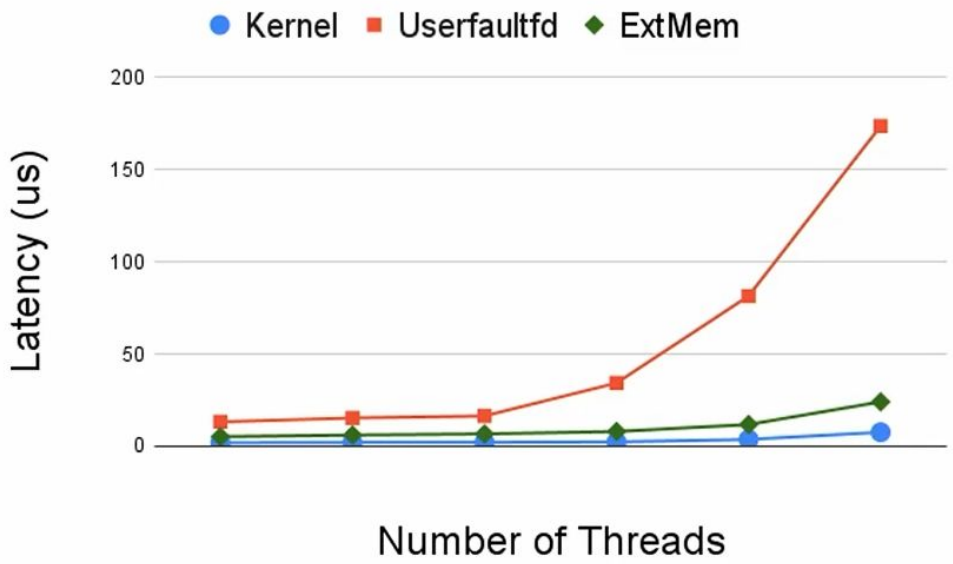
Implementing eviction & prefetching policies:
2QLRU *- 300 loc vs Linux - 500 loc*
Sequential Prefetcher *- 200 loc*
ExtMem-PR
…

Evaluation

Scalability with concurrent faulting threads

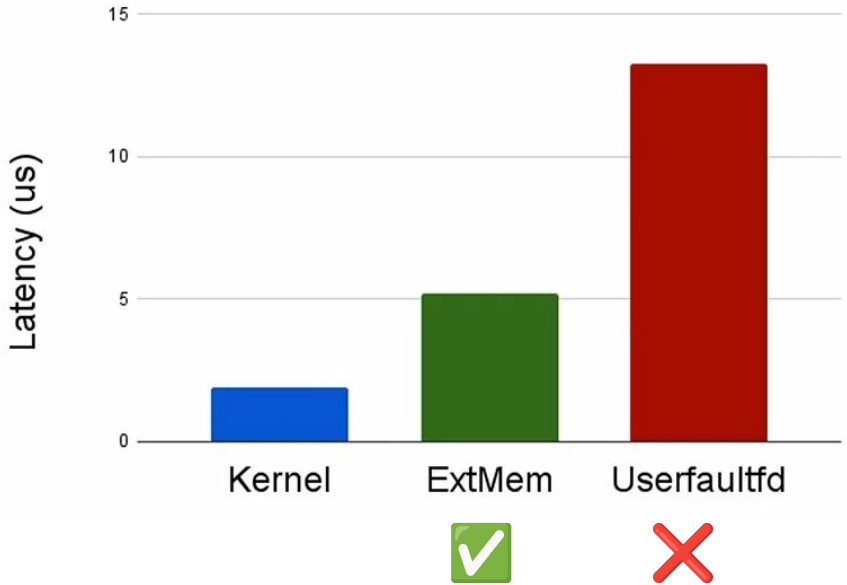
ExtMem's upcall has better scalability than userfaultfd by eliminating serialization

❌ ✅



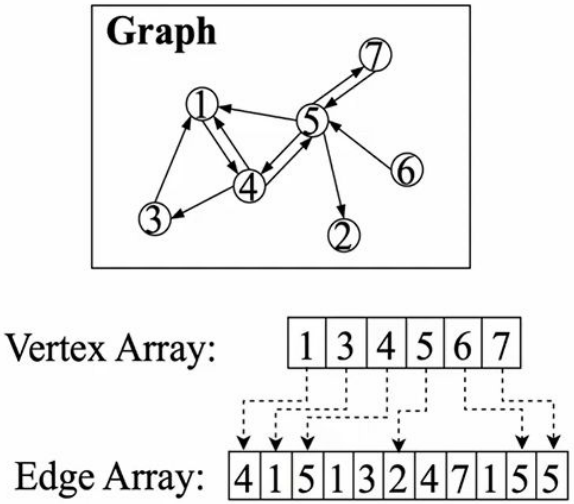
Upcall **minor** page fault **latency**

ExtMem's upcall has better latency than userfaultfd by eliminating expensive IPC's



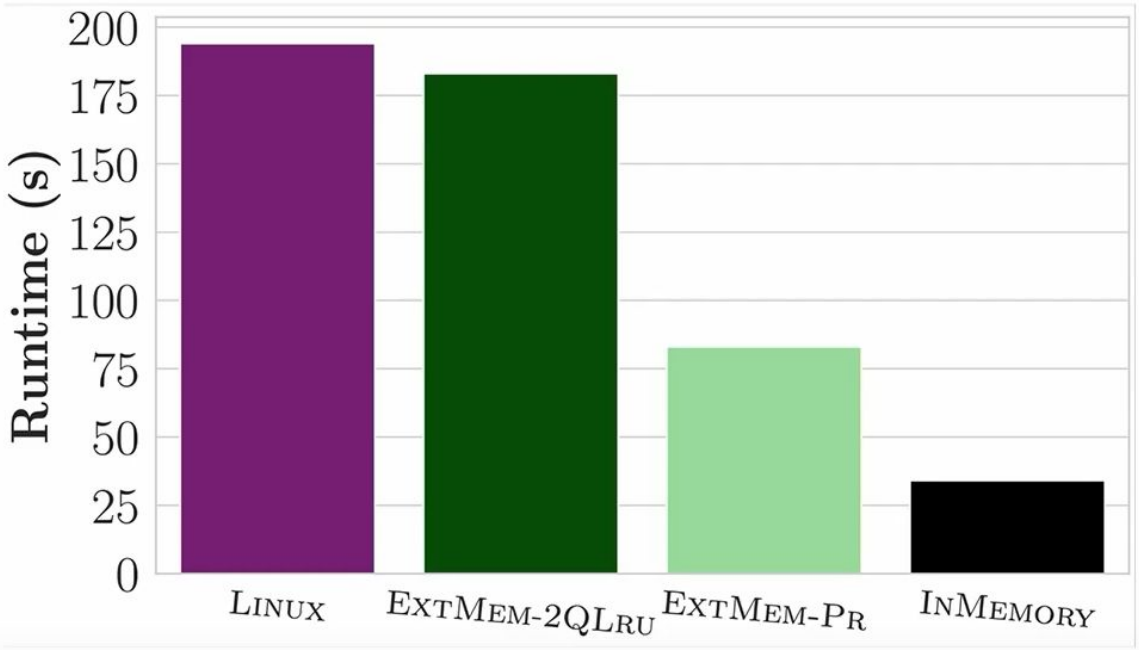
Example ExtMem usage: Graph Processing

- A large scale graph processing: PageRank
- Intution: Vertex array stays hot and edge array is cold after being used once.
- Custom Policies:
  - Pin vertex array
  - Discard + prefetch window on edge array

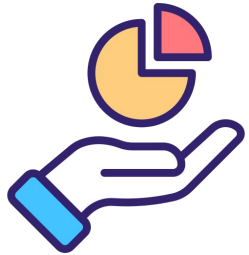


Example Results

ExtMem's special policy (ExtMem-PR) has better performance than the Linux default policies



Contributions



- **Flexible Framework Design**: Introduced ExtMem, a library that provides application-specific memory management without requiring significant in-kernel changes.
- **Application-Specific Optimization**: Enabled customized memory management policies tailored to specific application requirements rather than one-size-fits-all approaches.
- ExtMem builds on well-known ideas and mechanisms, such as user-level page fault handling, kernel signals and upcalls, but it **brings them together in a novel way**.

Strengths

- **Non-intrusive** – ensuring any kernel modifications are small and easily portable as Linux evolves.
- **Extensible** – allowing rapid development of new memory managers.
- **Transparent** – requiring no changes to applications.
- **Safe** – preventing application-specific memory managers from compromising the kernel or unaffected applications.
- **Efficient** – with low overhead and high performance gain.

Limitations

- ExtMem relies on SIGBUS-based upcalls to the faulting thread, which imposes **restrictions on signal handling**. Applications using custom signal handlers (or certain libraries) and rely on the default or custom SIGBUS behaviour, may conflict with ExtMem.

- ExtMem's upcall handler runs in an async signal context (via SIGBUS). **Must only use async-signal-safe functions** (no malloc(), printf(), etc…).

- Upcall-based handling scales better than blocking userfaultfd, but there is still **overhead from signal delivery and handler execution**.

- The paper does not explore use within Docker/Kubernetes. Thus, **deployment in containerized environments** would require additional integration and testing.

Ideas for Improvements

- **Kernel support for upcalls:** Make ExtMem’s upcall mechanism (based on SIGBUS) part of official Linux – reduces need for custom patched kernel and minimizes conflicts.
- **Integration with lockless paging:** Remove the bottleneck of mmap lock contention by using newer "lockless paging" techniques, making user-level page fault handling faster and more scalable.
- **Container and cloud compatibility:** Extend ExtMem to work cleanly inside Docker/Kubernetes environments to support modern deployment models.

What I learned/enjoyed in the paper

- Exploring low-level system programming and Linux internals.
- Understanding how system calls can be intercepted and extended.
- Seeing how custom memory managers and policies can be implemented.
- Realizing the flexibility to modify OS behavior from user space.
- Observing measurable performance improvements on modern workloads (e.g., ML, graph processing).

The Artifact

Artifact: **Available**

The artifact is available at: https://github.com/SepehrDV2/ExtMem

In the repo the below are included:

`src/` – *EXTMEM user-space library source code*
`linux/` – *Modified Linux kernel (based on v5.15)*
`run-scripts/` – *Scripts to reproduce experiments*
`swap-benchmarks/` – *Graph benchmarks (e.g., PageRank)*
`swap-microbenchmarks/` – *Microbenchmark tests*

Instructions on how to install and run it:

`README.md` – *Setup instructions and dependencies*

Artifact: **Functional**

Minimum System Requirements:

Given a dedicated swap file ExtMem can run on any multi-core X86 machine that can run the Linux 5.15>= kernel.

Recommended System Requirements:

For the best performance we recommend a dedicated NVMe SSD partition with async IO and at least 16 cores.

How I ran it:

1. *QEMU/KVM Virtual Machine, given 2 Cores and 8GB of RAM*
2. *Ubuntu 22.04 → minimal install*
3. *Installed required dependencies*
4. *Cloned the EXTMEM repository*
5. *Initialized submodules*
6. *Built the custom Linux kernel*
7. *Booted into the new kernel*
8. *Ran EXTMEM benchmarks using provided scripts → no results yet*

Artifact: Reproduced (TBD)

Coming Soon…

- Reproduction of paper results is pending.
- Recommended system requirements are not met on my current setup.
- May attempt reproduction via:
  - Cloud VM (e.g. AWS).
  - Modifying benchmarks to fit current setup.
- Plan to also run **XSBench** / **MyMicrobenchmark**, from the experimental handout, using ExtMem.

References

The Paper: https://www.usenix.org/system/files/atc24-jalalian.pdf

The Artifact: https://github.com/SepehrDV2/ExtMem

Video Presentation of the Paper: https://www.youtube.com/watch?v=_p8TRkpWgD0

Linux Kernel Interactive Map: https://makelinux.github.io/kernel/map/

userfaultfd(2): https://man7.org/linux/man-pages/man2/userfaultfd.2.html

LD_PRELOAD: https://stackoverflow.com/questions/426230/what-is-the-ld-preload-trick

SIGBUS: https://man7.org/linux/man-pages/man7/signal.7.html

libsyscall_intercept: https://github.com/pmem/syscall_intercept

mmap: https://man7.org/linux/man-pages/man2/mmap.2.html

Thank You!

Check this interactive ExtMem visualization*

https://cgi.di.uoa.gr/~sdi2000150/personal-webpage/static-files/cheatsheets/extmem.html

[Under construction & mostly AI generated for now]