the `page-fault` in our stars

the how and why your read most likely faults short (get it? ... sry)

2023-10-29 · 19 min · 4466 words

https://tramasys.mov/posts/page-faults/

> [!] **updated · august 2026**
>
> I revisited this post against current linux 7.1, glibc 2.44, Rust, jemalloc, memmap2, and QEMU sources. The fault entry, per-VMA locking, folio allocation, file-mapping, and userfaultfd paths now follow the current code.

what even is it?

A page fault is the CPU reporting that a virtual-memory access cannot be completed with the translation and permissions currently installed. The address may be perfectly valid. It may merely be late.

That last distinction matters. We tend to meet faults through `SIGSEGV`, so the name sounds terminal. Most page faults are routine kernel work. A program asks for address space, touches one page for the first time, the kernel fills in a page-table entry, and the same instruction runs again. Nothing reaches a signal handler and the program remains blissfully unaware of the paperwork.

An unresolved fault is the dramatic version. The address may belong to no mapping, a write may target a read-only mapping, an instruction fetch may hit non-executable memory, or the backing file may have disappeared underneath the mapping. The kernel then turns the hardware exception into `SIGSEGV` or `SIGBUS`, or fixes up a faulting kernel access instead.

The smallest useful experiment is one anonymous mapping and one load:

```
size_t page_size = (size_t)sysconf(_SC_PAGESIZE);
volatile unsigned char *memory = mmap(NULL, page_size,
    PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);

unsigned char value = memory[0];
```

`mmap()` has created a legal virtual range. It has not necessarily supplied a private physical page for `memory[0]`. The load is where the promise comes due.

```
virtual range exists              physical translation exists
        yes                                       no
         |                                         |
         +------------ first load -----------------+
                               |
                               v
                         page fault
                               |
                  kernel installs a mapping
                               |
                  the load runs successfully
```

This is demand paging. Virtual memory is cheap enough to promise in large amounts because physical memory is supplied when it is actually needed.

catching one without catching it

I put the load in a separate function and stopped immediately before it with `gdbgs`. Plain GDB accepts the same commands:

```
volatile unsigned char sink;

__attribute__((noinline))
static void first_read(uintptr_t address)
{
    sink = *(volatile unsigned char *)address;
}
```

With Intel disassembly enabled, the interesting instruction is not difficult to spot:

```
(gdb) set disassembly-flavor intel
(gdb) break first_read
(gdb) run
(gdb) disassemble /m first_read
   mov    rax,QWORD PTR [rbp-0x8]
=> movzx  eax,BYTE PTR [rax]
   mov    BYTE PTR [rip+0x2ec3],al
```

`info proc mappings` showed that `rax` pointed into a valid anonymous `rw-p` mapping. `info proc stat` reported 231 minor faults before the `movzx`. I used `stepi` once and asked again:

```
before movzx:  Minor faults (no memory page): 231
after movzx:   Minor faults (no memory page): 232
```

GDB did not stop in a kernel fault handler. From its point of view, `movzx` completed like any other instruction. The CPU entered the kernel, linux fixed the translation, returned to the saved instruction pointer, and the load ran again before the single-step trap was delivered.

There is a debugger-shaped rake on the floor here. Do not examine the target memory with `x`, and do not pass the address as a character pointer which GDB helpfully prints as a string. The debugger reads inferior memory through the kernel and can fault the page in for you. I did exactly that on the first run and then spent a while investigating the page fault I had already destroyed.

a larger first touch

One page is proof. Sixty-four MiB makes the shape visible. This program maps 16,384 normal 4 KiB pages, disables transparent huge pages for the range, and reads or writes one byte from each page. It records `ru_minflt` and `ru_majflt` with `getrusage()` around each pass:

```
for (size_t i = 0; i < pages; ++i)
    sum += memory[i * page_size];

for (size_t i = 0; i < pages; ++i)
    memory[i * page_size] = 1;
```

One run produced:

```
operation                  minor    major
mmap 64 MiB                    0        0
first read                 16387        0
first write                16384        0
second read                    0        0
MADV_DONTNEED                  0        0
read after discard         16384        0
write after discard        16384        0
```

The extra three faults in the first read are measurement and C-library noise. The interesting number is 16,384, once per base page. `mmap()` only edited the process's virtual-memory layout. The first read installed mappings to the kernel's shared zero page. The first write could not modify that shared page, so each page fault allocated private zeroed memory. The second read found all translations ready.

`MADV_DONTNEED` discarded the private anonymous contents and removed the useful translations. The virtual range remained. Reading and writing it paid the same first-touch costs again.

The complete run also looks pleasantly incriminating under `perf`:

```
$ perf stat -e page-faults,minor-faults,major-faults ./first-touch

             65596      page-faults:u
             65596      minor-faults:u
                 0      major-faults:u
```

There were four 16,384-fault passes plus process startup and measurement noise. `MADV_NOHUGEPAGE` is important for this particular demonstration. Current linux can satisfy anonymous faults with PMD-sized transparent huge pages or smaller multi-size THP folios. Without that hint, one fault may map far more than 4 KiB and flatten the graph for entirely legitimate reasons.

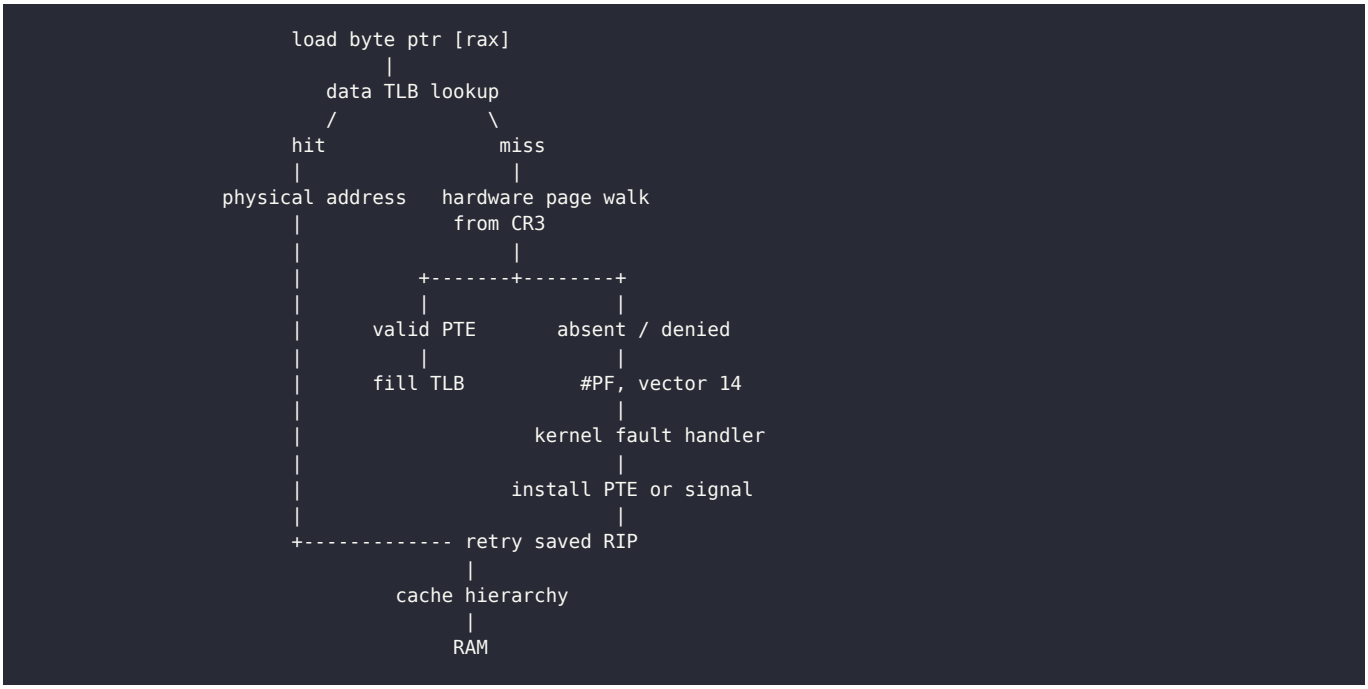
Minor does not mean unimportant, and major does not strictly mean "the disk moved its head." A successful fault is accounted as major when the memory manager returns `VM_FAULT_MAJOR` or had to retry. File-backed faults commonly become major when the folio was absent from the page cache and I/O was needed. A minor fault could still allocate and zero memory, copy a page, take locks, and invalidate a TLB entry. The adjective describes the resolution path, not how personally offended the latency-sensitive thread feels.

the hardware version

Before the kernel can handle anything, the CPU has to fail.

For a normal 4 KiB load, an x86-64 core first looks for the virtual-to-physical translation in its data TLB. A TLB miss is not itself a page fault. The hardware page walker can use the page-table root selected by `cr3`, walk the entries in memory, fill the TLB, and finish the load without an exception.

A page fault appears when that walk finds a non-present entry or an access which violates the page-table permissions:



The page tables translate addresses before ordinary cache lookup can identify the physical cache line. The caches do not solve a missing translation, and a page fault does not imply that the eventual data came from a disk. These are different layers which happen to meet on the slow path.

The useful architectural state is larger than just `cr2` and `cr3`:

- `cr3` selects the root of the current translation hierarchy and may carry a PCID
- `cr2` holds the faulting linear address on the traditional x86 exception path
- the page-fault error code describes what kind of access failed
- the saved `rip` identifies the instruction which must resume or receive a signal
- the saved privilege state tells the kernel whether execution came from ring 3 or ring 0

The error-code bits distinguish a non-present entry from a protection violation, a read from a write, supervisor from user access, and data from an instruction fetch. Newer bits cover reserved-bit violations, protection keys, shadow stacks, SGX, and AMD SEV-SNP reverse-map-table violations.

On the traditional IDT path, a privilege transition also moves execution onto the task's kernel stack. The CPU preserves the return state and supplies the error code. linux's entry assembly builds the register frame expected by C and dispatches vector 14 to

`exc_page_fault` .

Current x86 has a second entry mechanism called FRED. This is one place where the old `read_cr2()` -first description has aged. linux 7.1 selects the address like this in `arch/x86/mm/fault.c` , reduced slightly here:

```
address = cpu_feature_enabled(X86_FEATURE_FRED)
    ? fred_event_data(regs)
    : read_cr2();
```

Most existing machines use the latter branch. `read_cr2()` still becomes the Intel-syntax instruction `mov destination, cr2` . On a FRED-capable machine, the event data already carries the address. Either way, the handler captures it before doing work which might itself fault and overwrite the evidence.

Page faults are classed as faults rather than traps. The saved `rip` points at the instruction which did not complete. If the kernel repairs the cause, the exception return repeats that instruction. For the scalar load used here, the access has no half-completed architectural result. Restartable string instructions are the wrinkle. They can report partial progress in their register state so a restart continues with the remaining bytes.

entering current linux

`exc_page_fault` is emitted with `DEFINE_IDTENTRY_RAW_ERRORCODE` . It needs raw entry handling because page faults from kernel mode can occur in awkward contexts, and a fault on a userspace address may sleep while the kernel brings the page in.

After special handling for KVM asynchronous faults, the current path is:

```
asm_exc_page_fault
  -> exc_page_fault
  -> handle_page_fault
       -> do_kern_addr_fault       address belongs to kernel space
       -> do_user_addr_fault       address belongs to user space
            -> lock_vma_under_rcu  optimistic per-VMA route
            -> lock_mm_and_find_vma fallback under mmap_lock
            -> handle_mm_fault
                 -> __handle_mm_fault
                 -> handle_pte_fault
```

The dispatch is based on the faulting address, not merely the error code's user bit. Kernel code is allowed to access userspace through guarded helpers such as `copy_to_user()` . Such a store runs at ring 0 but still needs the user address-space fault machinery.

`fault_in_kernel_space()` treats addresses from `TASK_SIZE_MAX` upward as the kernel portion, except for the legacy vsyscall address. The current `task_size_max()` still chooses between these bounds:

```
4-level paging: (1 << 47) - PAGE_SIZE = 0x00007ffffffff000
5-level paging: (1 << 56) - PAGE_SIZE = 0x00fffffffffff000
```

The last canonical page is deliberately withheld due to old Intel `SYSRET` and AMD speculation hazards described in the source. The common page size is still `1 << PAGE_SHIFT` , with `PAGE_SHIFT == 12` on normal x86-64 builds.

Kernel-address faults have several legitimate fixups, including guarded userspace accesses and a few architecture quirks. An unexplained kernel fault eventually becomes an oops. User-address faults have more mundane checks for SMAP, reserved bits, instruction fetches, protection keys, and whether fault handling is currently disabled.

For an ordinary user-mode fault, linux enables interrupts, translates the x86 error code into `FAULT_FLAG_WRITE`, `FAULT_FLAG_INSTRUCTION`, and `FAULT_FLAG_USER`, then looks for the VMA which owns the address.

maple trees are not page tables

A virtual memory area, or VMA, describes a continuous region with one policy. It records bounds, read/write/execute permissions, private or shared behavior, the backing file and offset when one exists, and operations used to resolve faults. `/proc/<pid>/maps` is a userspace rendering of these regions.

A VMA does not say that every page is resident. Page tables answer the hardware question, "what physical frame and permissions apply right now?" The VMA answers the kernel question, "would this access be legal, and how should I create the translation?"

Current linux stores a process's VMAs in a Maple Tree. The old notes followed `find_vma()` directly. The common user fault now first tries `lock_vma_under_rcu()`, which walks the tree under RCU and acquires a read lock on only the matching VMA. This avoids bouncing every concurrent fault through the process-wide `mmap_lock`.

If that optimistic route cannot safely finish, the handler retries through `lock_mm_and_find_vma()`. That takes the `mmap_lock`, searches the Maple Tree, and can upgrade the lock to expand a `VM_GROWSDOWN` stack mapping. An address slightly below the current stack VMA can therefore be valid. An address in a real gap remains `SEGV_MAPERR` material.

The change is more than a renamed `find_vma`. Page faults from many threads in different VMAs can now proceed with less lock contention. The fallback still exists because file I/O, VMA mutation, stack growth, and retry handling have more complicated lifetime rules than one lock can wish away.

Once a VMA is locked, `access_error()` checks its permissions against the hardware error. A write into a genuinely read-only mapping ends here. A write into a private copy-on-write mapping is allowed by the VMA even though its current PTE is deliberately read-only. That disagreement is how COW asks for help.

descending the page tables

`handle_mm_fault()` sanitizes flags, checks architecture-specific VMA access, enters memory-cgroup fault handling, and separates explicit hugetlb mappings from the generic path. `__handle_mm_fault()` then builds a `vm_fault` and walks the software representation of the hardware hierarchy.

With four-level x86-64 paging, `p4d` is folded away. With LA57 enabled, it is a real fifth level. The generic names let most of the memory manager use the same code for both:

```
virtual address
  -> pgd_offset
  -> p4d_alloc
  -> pud_alloc
  -> pmd_alloc
  -> handle_pte_fault
```

Before falling all the way to a PTE, current linux may create or handle a PUD or PMD transparent huge mapping. Even the PTE path can allocate a multi-page anonymous folio when multi-size THP is enabled. "One fault, one 4 KiB page" is a useful first model and no longer a promise.

At the bottom, `handle_pte_fault()` is a compact switchboard:

```
PTE missing
  -> do_pte_missing
       -> do_anonymous_page     anonymous VMA
       -> do_fault              file or special VMA

PTE non-present but encoded
  -> do_swap_page               swap, migration, device-private state

PTE marked PROT_NONE
  -> do_numa_page               automatic NUMA placement hint

write to present read-only PTE
  -> do_wp_page                 COW or shared-write handling

present and permitted
  -> refresh accessed / dirty state, or repair a stale TLB case
```

There is a crucial correction to my old allocation path. `pte_alloc()` does not allocate the user's data page. It allocates a page which holds PTEs. That page is memory used to describe translations. The anonymous data allocation happens later in `alloc_anon_folio()`.

For an anonymous read fault, `do_anonymous_page()` can install a read-only PTE pointing at the shared zero page. Thousands of untouched zero-filled virtual pages can all read from the same physical frame. A first write then faults on the read-only PTE and takes the COW path.

For a write to a completely missing anonymous page, the route is roughly:

```
alloc_anon_folio
  -> choose an allowed folio order
  -> vma_alloc_folio
  -> apply NUMA memory policy
  -> __alloc_pages
       -> get_page_from_freelist
            -> rmqueue from per-CPU or buddy allocator state
       -> slow path: reclaim, compaction, kswapd, retry, or OOM
  -> zero the user-visible memory
  -> map_anon_folio_pte_pf
```

The allocator tries eligible zones and nodes, checks watermarks and memory policies, and prefers a fast free-list allocation. Under pressure it may wake reclaim, compact memory for a higher-order request, retry under different constraints, or eventually report `VM_FAULT_OOM`. Memory cgroups can impose a smaller world inside the machine's otherwise available RAM.

The `folio` is the memory-management object representing one or more pages. With `CONFIG_SPARSEMEM_VMEMMAP`, converting its first `struct page` to a PFN still reduces to its distance from `vmemmap`. `pfn_pte()` then shifts that PFN by `PAGE_SHIFT`, masks it into the physical-address bits, and combines it with the checked protection flags. A PTE stores a physical frame number and bits. It does not point at the `struct page` metadata.

The new PTE is installed while holding the page-table lock. Once ordering and MMU-cache hooks are satisfied, the handler can unwind to the exception return. The CPU repeats the original instruction, walks the repaired tables, and this time finds a translation.

six ways to arrive at the same exception

The entry machinery is shared. What happens below it depends on why the PTE was unhelpful.

anonymous first touch

An anonymous read normally maps the shared zero page. An anonymous write allocates a zeroed private folio. This is the route measured above.

copy-on-write after fork

`fork()` gives the child a new address space but initially lets parent and child PTEs refer to the same physical pages. Writable private mappings are made read-only. A later write reaches `do_wp_page()`.

If the old page is exclusively reusable, linux may make it writable in place. Otherwise it allocates a new folio, copies the contents, updates reverse mapping and accounting, replaces the PTE, and flushes the stale translation. The process that only reads never pays for a copy.

a file-backed mmap

A regular file VMA usually reaches its `vm_ops->fault` method and eventually `filemap_fault()`. The file offset is derived from the VMA offset and faulting address. linux first searches the mapping's page cache.

If an up-to-date folio is already cached, installing it is normally a minor fault. If it is absent, `filemap_fault()` marks the operation major, starts synchronous or asynchronous readahead, creates the cache folio, and calls the filesystem's `read_folio` operation when data must be fetched. It may drop the VMA lock while waiting for I/O and retry the entire lookup because another thread could change or unmap the region meanwhile.

Sequential access can therefore produce one I/O-producing fault followed by many cheaper faults into pages readahead already placed in the cache. Random access is less charming. Access beyond the current end of the mapped file is not an anonymous zero page. `filemap_fault()` returns `VM_FAULT_SIGBUS`.

Rust's `memmap2` is a thin layer over exactly this contract. Its Unix backend eventually calls `mmap` with `MAP_SHARED` for a normal mapping and exposes options for `MAP_POPULATE` and `MAP_NORESERVE`. Indexing the returned `Mmap` looks like a slice access in Rust. The first load can still travel through the page cache and filesystem before it returns.

```rust
use memmap2::MmapOptions;
use std::fs::File;

let file = File::open("stars.db")?;
let map = unsafe {
    MmapOptions::new().map(&file)?
};
println!("first byte: {}", map[0]);
```

The unsafe boundary is not there because `mmap()` immediately reads the file. It is there because another actor can truncate or mutate the file while Rust still presents the mapping as a slice. A truncation can turn that innocent index operation into `SIGBUS`.

swap and NUMA hinting

A non-present PTE is not necessarily empty. linux can encode a swap entry, migration entry, or device-private state in it. `do_swap_page()` finds or reads the swapped folio, validates that the PTE did not change while it slept, restores reverse mappings, and installs a present entry.

Automatic NUMA balancing intentionally marks sampled mappings so an access faults. `do_numa_page()` uses the fault to learn where the task is executing and may migrate memory closer to that node. Here the fault is not repairing an accident. It is an observation instrument designed into the policy.

an actually bad address

No suitable VMA becomes `SIGSEGV` with `SEGV_MAPERR`. A VMA exists but rejects the access becomes `SEGV_ACCERR`. A poisoned physical page or invalid file-backed access may become `SIGBUS`. Signal delivery changes the saved user context so a handler runs, or terminates the process when no handler saves it.

The faulting instruction is not blindly retried after the kernel has decided it can never succeed. That would be less an exception handler and more a very small denial-of-service loop.

a fault delegated back to userspace

`userfaultfd` allows a process to register ranges whose missing or write-protection faults should be reported through a file descriptor. The faulting task sleeps. A userspace manager reads a `UFFD_EVENT_PAGEFAULT` and resolves it with operations such as `UFFDIO_COPY`, `UFFDIO_ZEROPAGE`, or a wake. The original task then repeats its access.

QEMU uses this for postcopy live migration. The destination starts virtual CPUs before every guest RAM page has arrived. Its current C path registers RAM blocks with `UFFDIO_REGISTER_MODE_MISSING`, polls the userfaultfd in a fault thread, translates the reported host address back to a guest RAM block, requests that page from the source, and installs the received bytes with a userfaultfd operation.

```
destination vCPU touches guest RAM
             |
       host page is missing
             |
      linux queues userfaultfd event and sleeps vCPU thread
             |
      QEMU fault thread requests page over migration channel
             |
      QEMU resolve thread performs UFFDIO_COPY / UFFDIO_ZEROPAGE
             |
      kernel wakes vCPU thread
             |
      faulting memory access repeats
```

The kernel still catches the hardware exception. Userspace supplies the policy and contents. The process causing the fault and the process resolving it can even be different, which is a fairly long journey for one load.

malloc is mostly a promise

The C, C++, and Rust layers do not normally allocate a physical page for each language allocation.

Current libstdc++'s replaceable `operator new` calls `malloc()` in a loop and invokes the installed new-handler if allocation fails. Rust's Unix `System` allocator also calls `malloc`, `calloc`, or an aligned allocation routine. A `Box`, `Vec`, or `std::vector` usually reaches the same underlying allocator before any page fault enters the picture.

A C++ allocation followed by sparse writes makes the boundary visible without reading uninitialized objects:

```cpp
constexpr std::size_t length = 64 * 1024 * 1024;
auto *memory =
    static_cast<std::byte *>(::operator new(length));

for (std::size_t offset = 0; offset < length; offset += 4096)
    memory[offset] = std::byte{1};

::operator delete(memory);
```

`operator new` may only touch allocator metadata. The loop is what forces each selected page to become writable. Hard-coding 4096 is fine for this x86 experiment and should be replaced with the runtime page size in portable diagnostic code.

Rust exposes the same separation safely through a vector's capacity and length. `with_capacity` obtains storage, while `resize` initializes objects in that storage and therefore performs the writes:

```rust
let length = 64 * 1024 * 1024;
let mut memory = Vec::<u8>::with_capacity(length);

assert_eq!(memory.len(), 0);
memory.resize(length, 1);
```

glibc serves small requests from existing arena chunks. When an arena needs more virtual memory, `sysmalloc()` can extend the program break through `sbrk()` or create an anonymous mapping. Requests at or above the dynamic mmap threshold usually take the direct mapping route. Current `sysmalloc_mmap()` also gives eligible ranges a transparent-huge-page hint.

`brk()` is easy to over-credit. The kernel's `sys_brk` checks limits and neighboring VMAs, then creates or extends the heap VMA with `do_brk_flags()`. Unless the region is locked and must be populated, it does not walk through the range allocating every data page. Like `mmap()`, it establishes the right to fault later.

This makes three timings meaningfully different:

```
allocation call       allocator finds or obtains virtual range
first read             may install shared zero or file-cache mapping
first write            may allocate, zero, copy, and map private memory
```

`calloc()` and Rust's zeroed allocations can benefit particularly well. When glibc knows a fresh anonymous mapping is already logically zero, it need not write every byte merely to manufacture zeros the kernel already guarantees. The first reads can share the zero page, while pages that are actually mutated become private on demand.

allocators weaponize the bargain

General-purpose allocators know that virtual address space, committed memory, resident memory, and page-table presence are different currencies.

jemalloc 5.3.0's page backend maps extents with `mmap`. Where configured, it can include `MAP_NORESERVE`, avoiding an up-front swap reservation. Its purge paths use advice such as `MADV_FREE` for lazy purging and `MADV_DONTNEED` for forced purging. The latter lets linux discard anonymous contents and promises that a later access sees zeros again.

That trade moves work through time:

```
keep dirty page resident  -> more RSS, cheap reuse
purge with MADV_DONTNEED  -> lower RSS, future access faults and re-zeroes
reserve with mmap         -> cheap address space, pay on first touch
populate up front         -> slower setup, fewer faults in the hot path
```

There is no universally correct side. A batch process may prefer lazy commitment. A latency-sensitive service may prefault critical arenas with `MAP_POPULATE`, `MADV_POPULATE_READ`, `MADV_POPULATE_WRITE`, or `mlock`, then pay a predictable startup cost. A NUMA-aware program may deliberately let worker threads first-touch their own slices so physical pages follow the workers' memory policy.

The sharp edge of `MAP_NORESERVE` is that a successful mapping does not guarantee future writes can all be backed. Overcommit and memory-cgroup limits can turn a much later first touch into reclaim, the OOM killer, or process death. The address was successfully promised. The budget was not escrowed.

when read faults while writing

The description's bad joke hides a useful boundary case. Consider:

```c
char *buffer = malloc(16 * 1024 * 1024);
ssize_t received = read(fd, buffer, 16 * 1024 * 1024);
```

If the newly allocated buffer has not been touched, its pages may have no private writable translations. The user instruction stream does not touch them first. `read(2)` enters the kernel, the filesystem obtains data, and an eventual `copy_to_user()` writes it into `buffer` while executing at ring 0. That supervisor store can trigger the same hardware page fault on a user address.

The x86 handler deliberately sends it through `do_user_addr_fault()`. The current source warns against treating the error code's user bit as the whole story. Normal kernel accesses through `get_user()`, `put_user()`, and the copy helpers need to resolve valid user mappings too.

If the buffer is valid, the handler allocates or COWs the page and returns to the faulting copy instruction. If the pointer is invalid, x86 exception-table entries redirect execution to a fixup in the uaccess routine. The copy helper reports how many bytes could not be copied. Higher layers can return `EFAULT`, or a short count when some data was transferred first.

So a userspace `read()` can cause page faults while the CPU is in kernel mode, and a bad destination can make the read short. I regret nothing except the description.

This also explains why kernel code sometimes prefaults user buffers with helpers such as `fault_in_iov_iter_writeable()` before taking locks under which an unexpected sleeping fault would be dangerous. Prefaulting reduces the risk. Another thread can still unmap or reprotect the range, so guarded uaccess and exception tables remain necessary.

watching faults without changing them too much

For whole-process totals, start with tools which already consume the kernel's accounting:

```
$ /usr/bin/time -v ./program
    Minor (reclaiming a frame) page faults: 65596
    Major (requiring I/O) page faults: 0

$ perf stat -e page-faults,minor-faults,major-faults ./program

$ ps -o pid,min_flt,maj_flt,rss,vsz,comm -p "$pid"
```

`/proc/<pid>/stat` contains per-task minor and major counters. `/proc/<pid>/status` and `smaps` help compare virtual and resident size, while `mincore()` reports whether pages in a range are resident. `/proc/<pid>/maps` only proves that a VMA exists. It says nothing about which pages currently have useful PTEs.

`perf record -e page-faults:u` can sample faulting instruction pointers when permissions allow it. Kernel tracepoints such as `exceptions:page_fault_user` and memory-management tracepoints provide deeper paths on kernels which expose them through tracefs. eBPF and `bpftrace` can aggregate fault addresses or stacks, but probes on a hot fault path deserve restraint.

GDB is excellent for identifying the exact load or store, as the earlier single-step showed. It is poor evidence for page residency if you casually ask it to read the page. Observation is a memory access too.

Finally, counters require context. A high minor-fault rate during controlled startup can be healthy lazy population. A handful of major faults in a tail latency path can be disastrous. RSS without fault rate misses churn, and fault rate without I/O latency

misses why the thread waited.

fault resolved

A page fault is not the absence of memory. It is a request to reconcile a CPU access with the process's virtual-memory policy.

```
language allocation or mmap
  -> VMA says the address is legal
  -> CPU cannot translate or permit one access
  -> x86 raises #PF and preserves the faulting instruction
  -> linux finds and locks the VMA through the Maple Tree
  -> memory manager selects zero-page, allocation, COW, page cache,
     swap, NUMA migration, signal, or userfaultfd
  -> page tables and TLB state are repaired
  -> the original instruction runs again
```

Sometimes "memory allocation" only allocates an address. Sometimes a read allocates nothing and maps a communal page of zeros. Sometimes a write in the kernel allocates memory for a userspace buffer. Sometimes QEMU sends the fault over a network and asks another machine what the byte should have been.

The instruction at the center of it all is still just `movzx eax, BYTE PTR [rax]`. The stars around it are doing most of the work.

source trail

- linux 7.1 x86 page-fault handling
- linux 7.1 x86 page-fault error bits
- linux 7.1 x86 IDT setup
- linux 7.1 x86 control-register helpers
- linux 7.1 x86 user-address limits
- linux 7.1 VMA locking and lookup
- linux 7.1 Maple Tree VMA lookup
- linux 7.1 generic fault, anonymous, COW, and PTE paths
- linux 7.1 file-backed fault path
- linux 7.1 folio allocation under NUMA policy
- linux 7.1 physical page allocator
- linux 7.1 x86 PTE construction
- linux 7.1 memory-model PFN conversion
- linux 7.1 program-break implementation
- linux 7.1 uaccess helpers
- linux 7.1 user-buffer prefaulting
- linux userfaultfd documentation
- glibc 2.44 malloc system-allocation path
- glibc 2.44 brk wrapper
- libstdc++ operator new implementation
- Rust standard-library Unix system allocator

- memmap2 Unix mmap backend
- jemalloc 5.3.0 page backend
- QEMU 11.1 postcopy fault handling
- linux `mmap(2)` manual
- linux `madvise(2)` manual
- linux `userfaultfd(2)` manual
- Intel 64 and IA-32 software developer manuals
- Maple Tree design background