

Lecture 11.1

Virtual Memory pages, page tables and page faults

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
./vmphys (real)
parent  &x = 0x55f9037b0010  x = 1
        frame 0x120727
child   &x = 0x55f9037b0010  x = 2
        frame 0x12de1e
# same address, different memory
```

What you will learn today



Explain virtual and physical addresses, and read a process's map in `/proc/self/maps`.



Translate an address through x86-64's four page-table levels.



Explain page faults: demand paging, minor and major faults, copy-on-write, SIGSEGV.



Use mmap for files and for anonymous memory.

Today's plan (60 min)

0-5

Recap of 10.2

5-12

Why virtual memory

12-20

The address space

20-32

Pages, page tables

32-48

Page faults

48-55

mmap

55-60

Checkpoint

Lab 11.1: vtop.c, virtual to physical.

Quick recap of Lecture 10.2

Q1

Where does the kernel put the frame for a signal handler?

On the user stack; then rip = handler, and rt_sigreturn restores everything.

Q2

A SIGSEGV handler simply returns. What happens next?

The faulting instruction runs again and faults again, for ever (Homework 10.2). Today: what the fault is.

Q3

What survives exec: a handler, or SIG_IGN?

SIG_IGN. Handlers are reset to SIG_DFL because their code is gone.

PART 1

Why virtual memory?

Every address your program uses is a fiction, translated by the hardware.

Five problems one idea solves

Isolation: A process cannot even name another process's memory: its addresses lead only to its own frames.

Same addresses for everyone: Every program can be linked to start at the same place; the stack is always near 0x7ff...

More than RAM: malloc(4 GiB) succeeded on our 8 GiB machine: memory is promised first, given page by page.

Sharing: One copy of libc's code in RAM, mapped into every process (vtop: "only mine 0").

Protection: Each page has rights: code r-x, constants r--, data rw-. Breaking them: SIGSEGV.

DIFFERENT FROM C

Not the RAM address

Two processes can hold the same pointer value and reach different bytes.

Analogy: a city with private maps. Every family gets its own map, and every map has a "House No. 10". The post office (the MMU) looks at whose letter it is, and delivers it to the real plot of land written in that family's map.

SAME AS C

A pointer is a number

In C, `int *p` holds an address, and you can print it with `%p`. That number is a VIRTUAL address: it means something only inside your process.

Book: CS:APP §9.1–9.2; Love, LKD Ch. 15

Proof: same virtual address, different physical memory

```
./vmphys (our machine, real, as root)
```

```
parent      &x = 0x55f9037b0010 x = 1 only mine 1 frame 0x120727
child, before &x = 0x55f9037b0010 x = 1 only mine 0 frame 0x120727
child, after  &x = 0x55f9037b0010 x = 2 only mine 1 frame 0x12de1e
parent, end   &x = 0x55f9037b0010 x = 1 only mine 1 frame 0x120727
```

```
QEMU guest, Ubuntu 6.8 (static build; real)
```

```
parent      &x = 0x4ab0d0 x = 1 only mine 1 frame 0x1b1ea
child, before &x = 0x4ab0d0 x = 1 only mine 0 frame 0x1b1ea
child, after  &x = 0x4ab0d0 x = 2 only mine 1 frame 0x1b1df
```

Before the write

child and parent share ONE frame (only mine = 0): fork copied the page tables, not the page.

After the write

the child got its own frame: copy-on-write (Part 4). The parent's x is still 1.

How we know

/proc/self/pagemap: one 8-byte entry per virtual page with the frame number. Lab 11.1.

Homework 8.1 answered: the same &x, different values, because the same virtual address leads to two different frames.

PART 2

One process's address space

Code, data, heap, libraries, stack: all in `/proc/self/maps`.

Where each kind of C variable lives (layout.c)

./layout (real)

```
main (code)      0x55c131d23140
"hello" rodata   0x55c131d240e1
global_init      0x55c131d26010
global_zero      0x55c131d2602c
malloc(100)     0x55c13284c2a0
malloc(1 MiB)   0x7fd2e42ff010
printf (libc)    0x7fd2e4460100
local (stack)   0x7ffd57af587c
```

Each line is a **VMA** (virtual memory area): a range with one set of rights and one source (a file, or nothing = zeros). The heap grows up with `brk`; big mallocs get their own mapping; the stack grows down from near `0x7fff...`

/proc/self/maps of the same run (real, shortened)

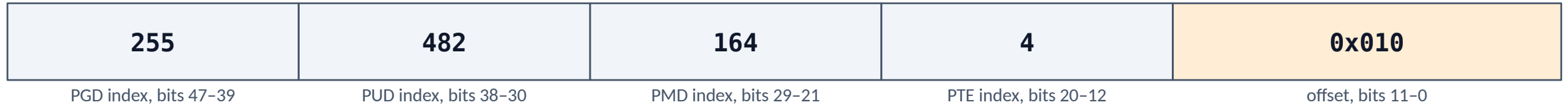
```
55c131d22000-55c131d23000 r--p layout # ELF headers
55c131d23000-55c131d24000 r-xp layout # code
55c131d24000-55c131d25000 r--p layout # rodata
55c131d26000-55c131d27000 rw-p layout # data, bss
55c13284c000-55c13286d000 rw-p [heap]
7fd2e42ff000-7fd2e4400000 rw-p # 1 MiB malloc
7fd2e4400000-7fd2e4428000 r--p libc.so.6
7fd2e4428000-7fd2e45b1000 r-xp libc.so.6
7fd2e4604000-7fd2e4606000 rw-p libc.so.6
7fd2e46b5000-7fd2e46b7000 r-xp [vdso]
7fd2e46b7000-7fd2e46b8000 r--p ld-linux-x86-64.so.2
7ffd57ad8000-7ffd57afe000 rw-p [stack]
ffffffffffff600000-...601000 --xp [syscall]
```

PART 3

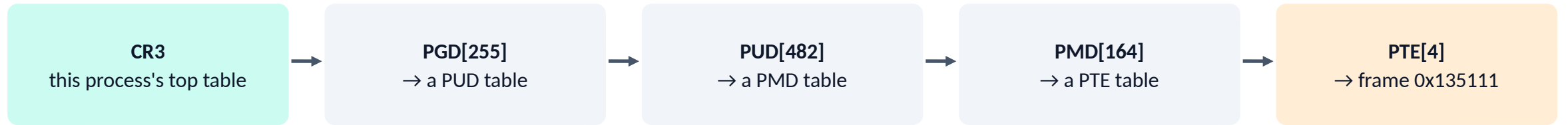
Pages and page tables

4 KiB pages, and a four-level tree that maps them.

Translating an address: four levels of 512 entries



virtual address 0x7ff894804010 (vtop.c, real); the CPU uses its lower 48 bits



```
arch/x86/include/asm, Linux 6.8 (real lines)
```

```
page_types.h      10  #define PAGE_SHIFT      12
pgtable_64_types.h 75  #define PGDIR_SHIFT     39
                  84  #define PUD_SHIFT      30
                  91  #define PMD_SHIFT      21
                  97  #define PTRS_PER_PTE  512
```

C link: like `pgd[255][482][164][4]`, but each inner array exists only if the process uses that part of its address space: a tree, not a full 4-D array.

Why not one flat table? 2^{36} pages $\times$ 8 bytes = 512 GiB per process. With the tree, 100 MiB of touched memory needs about 200 KiB of tables.

Book: CS:APP §9.6.3 (multi-level page tables), §9.7.1

What a page-table entry says, and what pagemap tells us

arch/x86/include/asm/pgtable_types.h (real)

```
10 #define _PAGE_BIT_PRESENT 0 /* is present */
11 #define _PAGE_BIT_RW 1 /* writeable */
12 #define _PAGE_BIT_USER 2 /* userspace ... */
15 #define _PAGE_BIT_ACCESSED 5 /* raised by CPU */
16 #define _PAGE_BIT_DIRTY 6 /* raised by CPU */
17 #define _PAGE_BIT_PSE 7 /* 2 MB page */
30 #define _PAGE_BIT_NX 63 /* No execute */
```

fs/proc/task_mmu.c: /proc/PID/pagemap (real)

```
1339 #define PM_PFRAME_BITS 55
1342 #define PM_MMAP_EXCLUSIVE BIT_ULL(56)
1344 #define PM_FILE BIT_ULL(61)
1345 #define PM_SWAP BIT_ULL(62)
1346 #define PM_PRESENT BIT_ULL(63)
/* one 8-byte entry per virtual page */
```

sudo ./vtop (Lab 11.1 reference; real)

main (code)	0x5591411f6388	[171 69 8 502]	+0x388	present	file 1	only mine 1	frame 0x12cbd5
printf (libc)	0x7ff895260100	[255 482 169 96]	+0x100	present	file 1	only mine 0	frame 0x10476e
global_var	0x5591411f9010	[171 69 8 505]	+0x010	present	file 0	only mine 1	frame 0x137623
local	0x7ffffbf112e4	[255 511 479 273]	+0x2e4	present	file 0	only mine 1	frame 0x1345bd
big, untouched	0x7ff894804010	[255 482 164 4]	+0x010	not present			
big, touched	0x7ff894804010	[255 482 164 4]	+0x010	present	file 0	only mine 1	frame 0x135111

The TLB: so that every access is not four extra reads

Without help

Every load or store would need 4 table reads first: memory five times slower.

TLB

Translation Lookaside Buffer: a small cache in the CPU of recent page → frame translations. A hit costs about nothing.

Context switch

A new process means new page tables (CR3 changes); old TLB entries are useless. PCID tags let the CPU keep them apart.

Huge pages

One 2 MiB page = 512 small pages in ONE TLB entry, and 512× fewer page faults (Homework 3).

```
cat /sys/kernel/mm/transparent_hugepage/enabled (real)
our machine:    always [madvise] never
QEMU guest:     always madvise [never]
```

C link: the TLB is a cache, like the CPU caches from Lecture 1: it pays off because programs touch nearby addresses again and again.

Book: CS:APP §9.6.2 (TLB); Love, LKD Ch. 15

PART 4

Page faults

Not an error: the normal way memory arrives.

Demand paging: malloc promises, the first touch delivers

● ● ● ./rss (our machine, real)

start	VmRSS	1412 kB	page faults:	89 minor, 0 major
after malloc(100 MiB)	VmRSS	1792 kB	page faults:	92 minor, 0 major
after touching 50 MiB	VmRSS	52988 kB	page faults:	12891 minor, 0 major
after touching 100 MiB	VmRSS	104188 kB	page faults:	25691 minor, 0 major
touching it all again	VmRSS	104188 kB	page faults:	25691 minor, 0 major
after free	VmRSS	1788 kB	page faults:	25691 minor, 0 major

malloc(100 MiB): VmRSS (RAM really used) hardly moves: only the VMA was created.

Touch 50 MiB: 12 800 more faults: one per 4 KiB page, the first time it is touched.

Touch again: No faults: the pages are present now.

free: The mapping is removed (munmap): RAM goes back at once.

● ● ● QEMU guest, Ubuntu 6.8, 512 MiB (real)

after malloc(100 MiB)	VmRSS	860 kB	page faults:	55 minor
after touching 100 MiB	VmRSS	103256 kB	page faults:	25654 minor
after free	VmRSS	856 kB	page faults:	25654 minor

25 600 faults = 100 MiB / 4 KiB. Each one is the path in the next video: about a microsecond each, invisible to the program.

Book: CS:APP §9.3.4 (demand paging)

Watch one page fault, from the MMU to the kernel and back

A page fault: the first write to a new page

SPLIT

WALK

FAULT

FIX

RETRY

```
user: big[5 * 4096] = 1;
movb $1, (%rax)    rax = 0x7ff894804010
```

255

bits 47-39

482

bits 38-30

164

bits 29-21

4

bits 20-12

0x010

bits 11-0

4 indices of 9 bits + a 12-bit offset (inside the 4 KiB page)

PGD[255] → the next table

PUD[482] → the next table

PMD[164] → the next table

PTE[4] **NOT PRESENT**

CR3 → PGD → PUD → PMD → PTE: four memory reads (the TLB usually saves them)

kernel (Linux 6.8)

exc_page_fault
fault.c 1510: address from CR2

do_user_addr_fault
fault.c 1230

lock_vma_under_rcu
1347: a VMA rw-p covers it

handle_mm_fault
1355 → memory.c 5438

do_anonymous_page
memory.c 4242: zeroed page

RAM (frames)

free frame

handle_mm_fault finds the missing PTE in an anonymous mapping: do_anonymous_page takes a free frame and fills it with zeros, so you never see another program's old data.

Watch for

1. The address is **cut into 5 parts**.
2. The walk ends at a PTE that is **not present**.
3. The kernel checks the **VMA** first.
4. A **zeroed** frame is given.
5. The **same instruction** runs again.

Click the video to play or pause. About 1.2 minutes, no sound.

Md Mahbub E Noor | Lecture 11.1: Virtual Memory | University of Barishal

15 / 27

The fault path in the kernel (Linux 6.8)

● ● ● arch/x86/mm/fault.c (real lines)

```
1510 DEFINE_IDENTRY_RAW_ERRORCODE(exc_page_fault)
1512     unsigned long address = read_cr2();
...
1230 void do_user_addr_fault(regs, error_code, address)
1347     vma = lock_vma_under_rcu(mm, address);
1351     if (unlikely(access_error(error_code, vma)))
1355     fault = handle_mm_fault(vma, address, ...);
... /* no VMA at all: */
1380     bad_area_nosemaphore(regs, error_code, address);
...
848     force_sig_fault(SIGSEGV, si_code, address);
```

● ● ● mm/memory.c (real lines)

```
5438 vm_fault_t handle_mm_fault(...)
5130 static vm_fault_t handle_pte_fault(...)
5164     if (!vmf->pte) return do_pte_missing(vmf);
5167     if (!pte_present(...)) return do_swap_page(vmf);
5180     if (!pte_write(entry)) return do_wp_page(vmf);
3742 if (vma_is_anonymous(vmf->vma))
3743     return do_anonymous_page(vmf); /* 4242 */
3745 else return do_fault(vmf); /* file */
```

First question: is it a bug? No VMA, or the rights forbid it: SIGSEGV.
Otherwise the kernel decides which kind of fault it is, fixes the page tables and lets the instruction run again.

Book: Bovet & Cesati Ch. 9 (page fault exception handler)

Kinds of page fault, measured

Kind	When	Measured (our machine)	Kernel
minor, anonymous	first touch of malloc'ed memory	25 600 faults for 100 MiB (rss.c)	do_anonymous_page
major	file page not in RAM: read from disk	2178 for /usr/bin/gdb, 2178 pages (major.c)	do_fault + disk read
minor, file	file page already in RAM (page cache)	only 137 for the same 2178 pages	do_fault, fault-around
copy-on-write	write to a shared page after fork	25 600 faults, 90.6 ms (cow.c)	do_wp_page
invalid	no VMA, or not allowed	SIGSEGV (segv2.c)	bad_area → SIGSEGV

```
./major (real)
/usr/bin/gdb: 8920528 bytes = 2178 pages
first pass (from disk)      0 minor  2178 major
second pass (page cache)   137 minor   0 major
```

Why 137, not 2178? On a file fault the kernel also maps up to 15 neighbouring pages that are already in RAM: "fault-around", 64 KiB (memory.c 4633). $2178 / 16 \approx 137$.

Copy-on-write, counted (cow.c)

./cow (our machine, real)

```
fork took 1.54 ms
child read 100 MiB:      2 page faults    11.5 ms
child wrote 100 MiB: 25600 page faults  90.6 ms
child wrote it again:    0 page faults     0.4 ms
parent still sees p[0] = 1
```

QEMU guest (emulated CPU: slower; real)

```
fork took 6.63 ms
child read 100 MiB:      2 page faults     8.6 ms
child wrote 100 MiB: 25600 page faults 394.1 ms
```

1

fork copies the page tables and marks every page **read-only** in both: fast.

2

Reading needs nothing: the shared frame is there.

3

Writing faults (read-only PTE): `do_wp_page` copies 4 KiB into a new frame, makes it writable.

4

Writing again: the page is the child's own now: no fault.

fork + exec (Lecture 8.1) almost never copies anything: `exec` throws the shared pages away before anyone writes them.

When the fault is a bug: SIGSEGV, with the reason

```
./segv2 (SA_SIGINFO handler; real)
```

```
read *NULL          SIGSEGV at (nil)          si_code 1 = SEGV_MAPERR (nothing mapped there)
read 0x123456789000  SIGSEGV at 0x123456789000  si_code 1 = SEGV_MAPERR (nothing mapped there)
write into "hello" (rodata) SIGSEGV at 0x55e575cb608e si_code 2 = SEGV_ACCERR (mapped, but not allowed)
write into the code of try() SIGSEGV at 0x55e575cb5230 si_code 2 = SEGV_ACCERR (mapped, but not allowed)
```

```
segv2.c (the handler)
```

```
static void on_segv(int sig, siginfo_t *si, void *c)
{
    addr = si->si_addr;    /* where */
    code = si->si_code;     /* why */
    siglongjmp(back, 1);    /* not back to the fault */
}
```

SEGV_MAPERR: no VMA contains the address: NULL, a wild pointer, a freed and unmapped block.

SEGV_ACCERR: a VMA exists, but its rights say no: writing to code or string constants, executing data.

DIFFERENT FROM C

Not every C bug faults

Writing past an array into memory that is still mapped gives no fault at all: silent corruption (Lecture 11.2).

PART 5

mmap: map a file, or fresh memory

One system call behind libraries, big mallocs and fast file access.

mmap: a file as an array

● ● ● mmapfile.c (core lines)

```
p = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
for (i = 0; i < st.st_size; i++)
    if (p[i] == '\n') lines++;    /* no read() at all */
...
q = mmap(NULL, 12, PROT_READ | PROT_WRITE, MAP_SHARED, out, 0);
q[0] = 'J';                      /* writes the FILE */
```

● ● ● ./mmapfile ; wc -l ; cat (real)

```
/usr/include/stdio.h: 34649 bytes, 985 lines, mapped at 0x7fc79eef4000
985 /usr/include/stdio.h
Jello, file          # was "hello, file"
```

	MAP_PRIVATE	MAP_SHARED
a file	read it as memory; your writes are copy-on-write, never saved	writes go to the file and to every other process mapping it
no file (ANONYMOUS)	fresh zero pages: big malloc, stacks	shared memory after fork (Lecture 10.1)

Who uses mmap? ld.so maps every shared library (the r-xp lines of libc). glibc's malloc uses it for blocks ≥ 128 KiB (Lecture 11.2). Databases map their files. The kernel fills the pages on first touch: page faults again.

Book: CS:APP §9.8 (memory mapping); Kerrisk, TLPI Ch. 49

LAB 11.1

vtop.c: virtual to physical

Split addresses into page-table indices and ask pagemap where each page lives.

Lab 11.1 tasks, step by step

1 **TODO 1** in vtop_starter.c: split v into the four 9-bit indices and the 12-bit offset

2 **TODO 2**: read the 8-byte pagemap entry at offset $(v / 4096) \times 8$ with pread

3 **TODO 3**: decode bits 63 (present), 61 (file), 56 (only mine), 0-54 (frame)

4 **TODO 4**: physical address = frame $\times$ 4096 + offset

5 Run with sudo. Then: add a fork() and look at global_var before and after the child writes it

Without root every frame reads as 0: the kernel hides physical addresses from normal users. The present, file and "only mine" bits still work.

```
sudo ./vtop (real, shortened)

name      [pgd pud pmd pte] offset
main (code) [171 69 8 502] +0x388
  present  file 1  only mine 1
printf (libc) [255 482 169 96] +0x100
  present  file 1  only mine 0
global_var [171 69 8 505] +0x010
  present  file 0  only mine 1
malloc(64) [171 69 459 503] +0x2a0
  present  file 0  only mine 1
big, untouched [255 482 164 4] +0x010
  not present
big, touched [255 482 164 4] +0x010
  present  frame 0x135111
  physical 0x135111010
```

Checkpoint: test yourself

- Q1** Two processes print the same pointer value. Same memory? → not necessarily: each has its own page tables
- Q2** How many bits does each page-table index have on x86-64, and why? → 9: a table is one 4 KiB page of 512 entries of 8 bytes
- Q3** malloc(100 MiB) returned. How much RAM is used now? → almost none: frames come on the first touch (page faults)
- Q4** Minor or major: the page is in the page cache, not yet mapped? → minor: no disk read is needed
- Q5** Write to a string literal: which signal and which si_code? → SIGSEGV, SEGV_ACCERR (mapped r--, write not allowed)

Key ideas from today



Every pointer is virtual; the MMU translates it with the process's own page tables.



`/proc/self/maps` lists the VMAs: code, data, heap, mappings, stack, each with its rights.



x86-64: 9 + 9 + 9 + 9 bits of index, 12 bits of offset; 4 KiB pages; the TLB caches translations.



A page fault is normal: first touch, file pages, copy-on-write. Only bad ones become SIGSEGV.



fork shares frames read-only; the first write copies one page (copy-on-write).



mmap maps files or zero pages; private or shared. Libraries and big mallocs use it.

Homework 11.1 (due before Lecture 11.2)

1. COW, seen

Add a `fork()` to your `vtop`. Print `global_var`'s entry in the child before and after it writes the variable, and in the parent. Explain each frame number and the "only mine" bit.

2. How big?

How many bytes would ONE flat page table for 48-bit addresses need? How much do 4-level tables need for 100 MiB of touched memory? Check VmPTE in `/proc/self/status`.

3. Huge pages

`thp.c` touches 100 MiB twice: normal, then with `madvise(MADV_HUGEPAGE)`. Predict the page-fault counts; run it; explain.

4. How deep?

`deep.c` recurses with about 1 KiB per call until SIGSEGV. Predict the depth from `ulimit -s`; run it; then try `ulimit -s 16384`. Why does its handler need `sigaltstack`?

5. 137

`major.c` read 2178 pages from the page cache with only 137 faults. Find the reason in `mm/memory.c` (search `fault_around`).

Next lecture and reading

NEXT: LECTURE 11.2

Memory allocation

What malloc really does: brk and mmap, chunks and free lists, heap bugs, overcommit and the OOM killer, and the kernel's own allocators. Lab: your own malloc.

Main text

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., Ch. 9 (§9.1–9.8 today; §9.9 dynamic memory allocation for 11.2).

Kernel side

R. Love, **LKD**, Ch. 15 (address space) today; Ch. 12 (memory management: pages, slab) for 11.2. Bovet & Cesati Ch. 2, 9.

Source

Linux 6.8: arch/x86/mm/fault.c, mm/memory.c, fs/proc/task_mmu.c, include/linux/mm_types.h (Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

Lab files

`lab11_files.zip`: `vtop_starter.c` and the demos; `mymalloc_starter.c` for Lab 11.2.