Operating Systems

Memory Management

Sven Dziadek

October/November 2022

EPITA

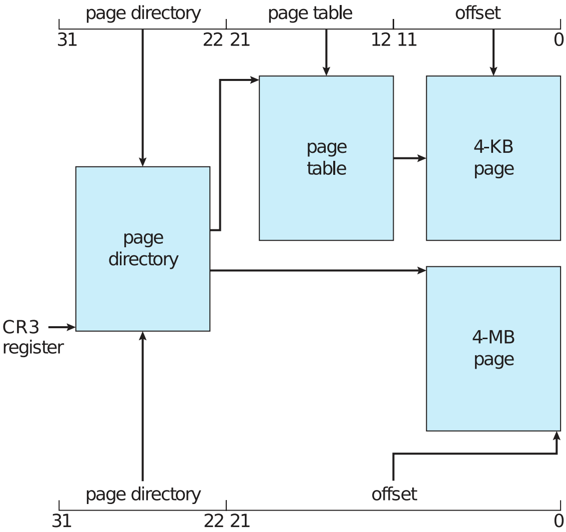
Recall: Paging – x86 (2-level)

Virtual address: 32 bit

Split in three parts:

page number		page offset
p_1	p_2	d
10	10	12

- Page size: 4 KiB $= 2^{12}$ B
- Page Table & Page Directory: 4 KiB
  they store $2^{10} = 1024$ entries
- 12 bits of each entry for metadata



Part II

Advanced Memory Management

Swapping

Page Swapping

If the system is low on memory, it can **swap out** frames to a backing store.

Backing store

File or disk partition in **secondary storage**

Pages are marked as `invalid` in metadata of page table
(lower 12 bits of addresses)
Kernel data structure (`vm_area` in Linux) has a flag for being `swapped out`.

History: Process Swapping

Historically, **swapping** moved entire processes to the backing store.
Modern swapping of pages is also called **paging**.

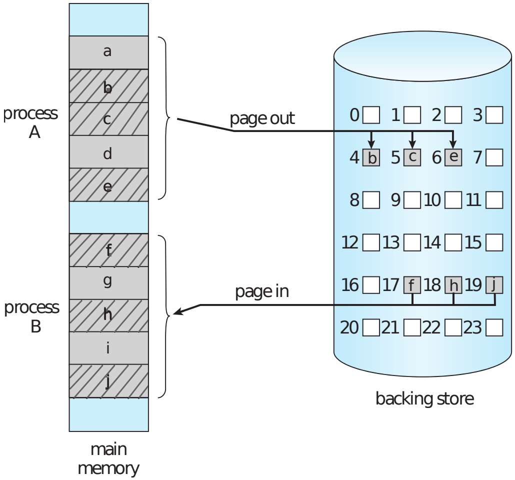
Page Swapping

Swap out/Page out

- Copy frames to backing store
- Unmap frames
  from physical memory
- Mark `vm_area` as `swapped out`
- Mark page as `invalid`

Swap in/Page in

- Load frame back to memory
- Mark page as `valid`



On-Demand Paging

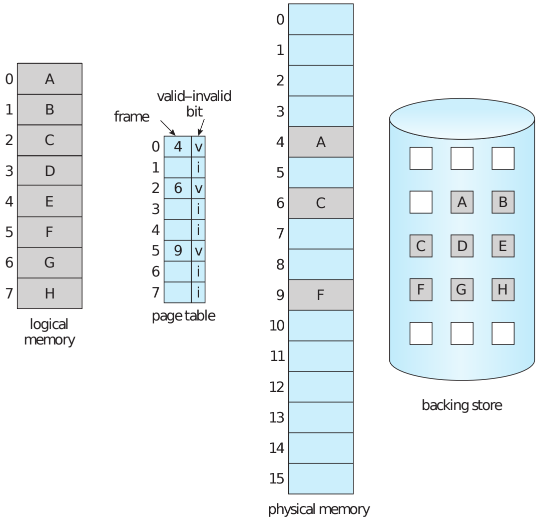
On-Demand Paging

Demanded memory is not given straight away.

- Kernel keeps promised pages in `vm_area`
- It stores what should be mapped
- Page is marked `invalid` in page table

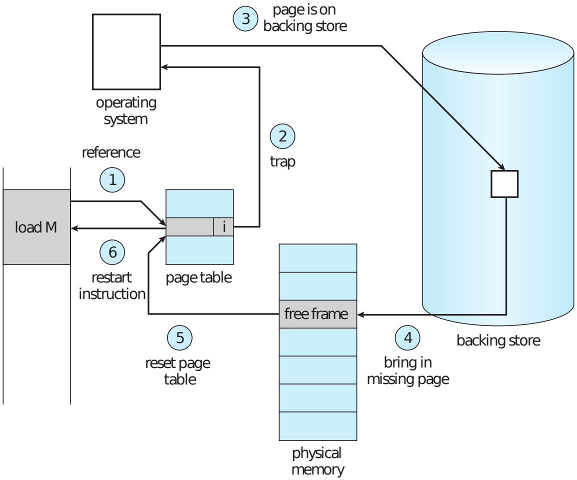
On access to page:

- Page fault is handled by the kernel
- Kernel does the mapping



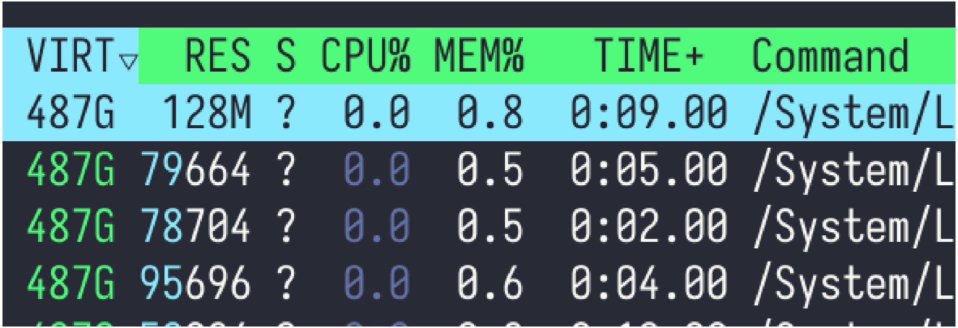
Handling of Page Faults

1. Invalid page is accessed
2. Page `fault` occurs
3. Kernel searches frame data
4. Frame is loaded to memory
5. Page table is updated
6. Instruction is restarted



On-Demand Paging - Demo

Run `top` or `htop`.



- `VIRT` is the amount asked to your kernel
- `RES` (for **resident**) is the amount actually mapped to the process

`calloc(3)` is faster than `malloc(3)+memset(3)` (1)

`calloc(3)`

Allocates an amount of memory that is zero-filled.

`memset(3)`

Fills memory with a constant byte, (e.g. zero).

Fun fact

`calloc(3)` is faster than `malloc(3)+memset(3)`.

`calloc(3)` is faster than `malloc(3)+memset(3)` (2)

The gap in speed increases with the size of the allocation.

- `calloc(3)` remains rather constant with large allocations.
- For `1GiB` allocations:
  `calloc(3)` is approx. 100 times faster than `malloc(3)+memset(3)`.

- For small allocations: it actually **is** a `malloc(3)+memset(3)`!
- For large allocations: it cheats with the kernel's help.

Zero page

Promise:

`mmap(2)` returns zero-filled pages

Reality:

- Zero page: read-only page full of zeros
- `mmap(2)` returns read-only pages mapping to the zero page
- On write access:
  - `Page fault` is emitted
  - Kernel checks `vm_area` and maps a new physical frame

Example: File Mapping

`mmap(2)` can map files into virtual memory
(use flag `MAP_FILE` and give file descriptor to file)

- Kernel represents file initially only in `vm_area`
- On reading: kernel maps accessed page into memory
- After write (depending on flags): kernel writes data back to disk

For some situation much faster than `read(2)`/`write(2)`/`seek(2)`

Copy-On-Write

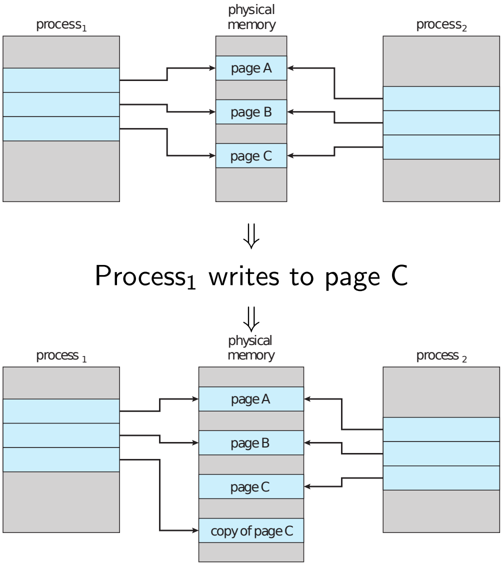
Copy-on-Write

Copy-on-Write (CoW)

Read-only page

that gets duplicated on write access

- One page mapped to multiple processes
- On write access:
  - kernel copies the page
  - copy is only for the writing process



CoW & fork(2)

```c
pid_t fork(void);
```

creates a new process by duplicating the calling process

- Copying all memory at once would be costly both in time and in space.
  $\Rightarrow$ set all pages `Copy-on-Write` and share them between both processes
- This makes `fork(2)` both fast & memory efficient

Example (In-memory database `Redis`)

For backup at runtime:

`Redis` calls `fork(2)` and uses child as free snapshot

Demo – Fork & Memory

```c
int main(void)
{
  int x = 0;

  pid_t pid = fork();

  if (pid < 0)
    return 1;  /* fork error */

  if (pid > 0) {
    x++;
    printf("[Parent] x: %d (%p)\n", x, &x);
  } else
    printf("[Child]  x: %d (%p)\n", x, &x);

  return 0;
}
```

What values of `x` are printed?

What addresses of `x` are printed?

OOM Killer

On-demand paging = optimistic memory allocation strategy = lazy allocation
$\Rightarrow$ memory allocation can succeed even if memory is full
$\Rightarrow$ access to pages can later lead to out-of-memory errors (OOM)
OOM errors can occur at any occasion: kernel has to call the OOM killer

OOM killer

Kills a process that consumes a lot of memory

You can configure what processes are critical and should be avoided.

More Techniques

Address Layout Randomization (ASLR)

To prevent attackers from reliably jumping to particular functions in memory:

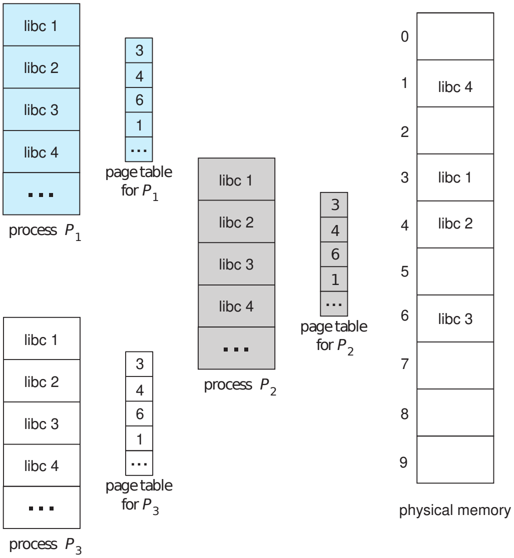
ASLR

Randomly arrange positions of code, stack, heap and libraries

Depending on implementation: randomness is limited

Shared Libraries

Recall:
Shared libraries are mapped into several address spaces



KSM

Kernel Samepage Merging (KSM)

Scans memory for duplicates and merges them as `Copy-on-Write` page

Mostly used for hypervisor (virtual machine monitors) to share between different virtual machines

Shared Memory

- You can have read/write shared pages between process
  (Not `Copy-on-Write` but both processes can read & write)
- Used for Inter-Process-Communication (IPC)

- Shared pages are mapped with the `MAP_SHARED` mmap flag.
  (You would usually not use that directly)
- Pages are still mapped in the child processes after using `fork(2)`
- See `shm_overview(7)`.

PAE

Page (or Physical) Address Extension (PAE)

Motivation: How to put more than 4 GiB of RAM into a 32 bits machine?

Extend physical addresses!

As virtual addresses stay unchanged
$\Rightarrow$ No process can address whole physical address space at once

Depends on OS, CPU and Mainboard

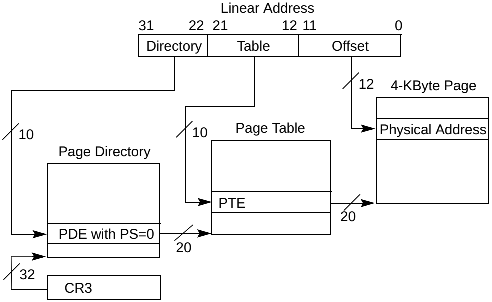
PAE – x86

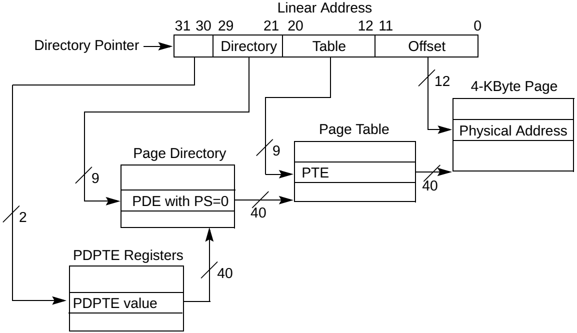
Without PAE:

- 32 bit virtual addresses
- 32 bit physical addresses
- entries are 32 bits ($\cong$ 1024 entries)

With PAE:

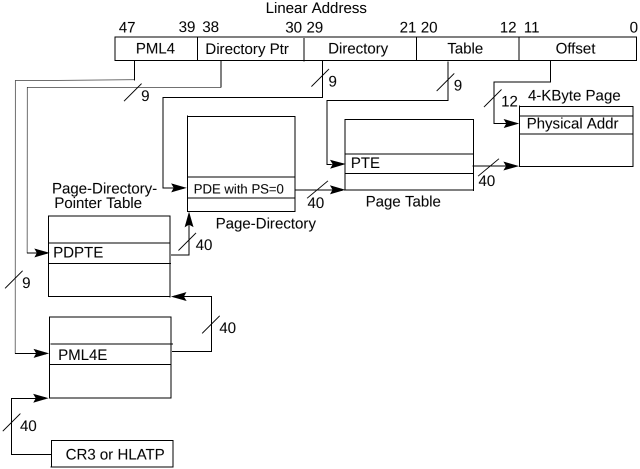
- 32 bit virtual addresses
- 52 bit physical addresses
- entries are 64 bits ($\cong$ 512 entries)





PAE – x86-64

Recall: uses 48 bit virtual addresses



Translates to 52 bit physical addresses

There is a second possibility:

Add layer (5-layer)

Translates 57 bits $\rightarrow$ 52 bits

This finally allows to address more than 256 TiB

References

- Linux sources:
  - Task struct: **include/linux/sched.h:task_struct**
  - Virtual memory: **include/linux/mm_types.h**
    - mm_struct: linked to your process
    - vm_area_struct: descriptor of a vm area
  - vm_operations_struct: **include/linux/mm.h**
    - Operations for a `vm_area`.
- Why does calloc exist?