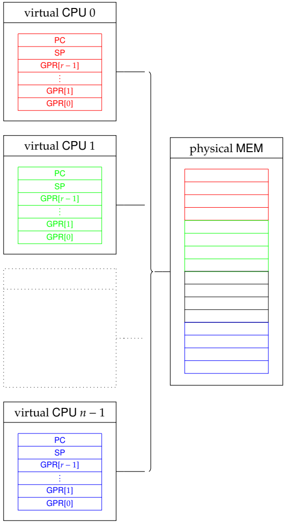
Concept: *virtualise* the memory

- We already virtualised the processor, *but*
  1. *how* do we segregate the processes in memory, and
  2. *why* put up with this restriction?!
- In general, several layers of memory management
  1. hardware (RAM, MMU, MPU),
  2. kernel (address space protection and virtualisation), and
  3. user (allocation, deallocation, garbage collection),

  supporting various use-cases, e.g.,

  1. process manages memory process uses,
  2. kernel manages memory kernel uses, and
  3. kernel manages memory process uses,

  warrant attention ...



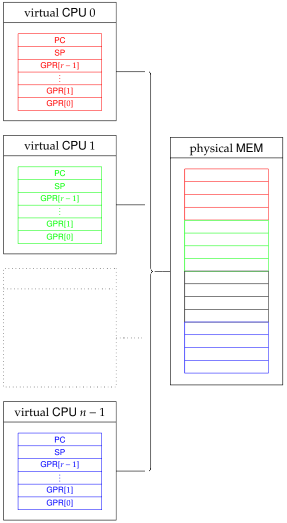
Concept: *virtualise* the memory

- We already virtualised the processor, *but*
  1. *how* do we segregate the processes in memory, and
  2. *why* put up with this restriction?!
- In general, several layers of memory management
  1. hardware (RAM, MMU, MPU), and
  2. kernel (address space protection and virtualisation),

  supporting various use-cases, e.g.,

  3. kernel manages memory process uses,

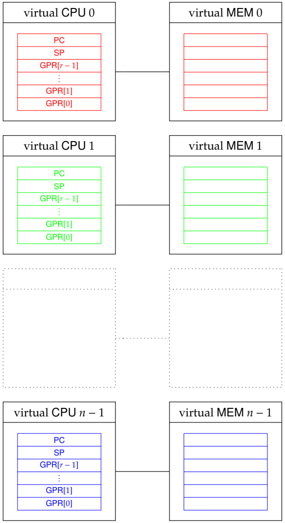
  warrant attention ...
- ... *but* we'll consider a narrower remit.



Concept: *virtualise* the memory

- Specifically, we want processes to
  1. *appear* to have dedicated access to the whole physical memory,
  2. have a larger footprint than physical memory *if* required,
  3. be protected wrt. access to their regions of the physical memory,
  4. to share regions of physical memory *if* required,
  5. ...

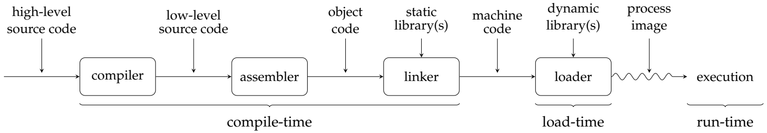
  so, the question is, *how*?



Concept (1)

Definition (resolution, relocation etc.)

A (sub-)sequence of standard steps, implemented by a user mode tool-chain plus the kernel, translates a high-level program into a form ready for execution



noting that

- at various steps we might use an
  - **abstract address**, e.g., a symbol per

    ```
    goto foo;
    ```

  - **concrete address**, e.g., a literal per

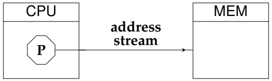
    ```
    uint32_t* bar = ( uint32_t* )( 0xDEADBEEF );
    ```

- abstract addresses are **resolved** into concrete addresses before execution, and
- addresses may be **relocated** (or moved, i.e., rewritten) before *or* during execution.

Concept (2)

Definition (**address stream** etc.)

Although executed instructions provoke memory accesses, e.g.,



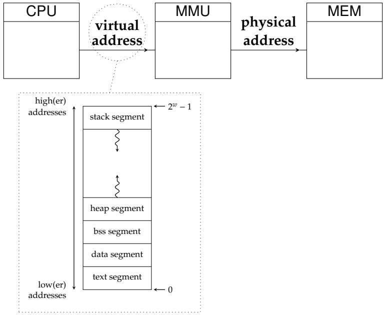
we can often ignore processes themselves, and instead focus on the resulting **address stream** (i.e., a sequence of addresses). An address stream will, *on average*, exhibit various properties:

- **access locality**, i.e., reuse of the same or "close" addresses, which implies a
- **working set**, denoted $\mathcal{W}(\mathbf{P}_i)$ for some process $\mathbf{P}_i$, which captures the set of addresses, or portion of the address space, currently in use.

Concept (3)

Definition (address space etc.)

Including a **Memory Management Unit (MMU)** per



allows transparent manipulation of the semantics of addresses and address spaces. Specifically,

- a **virtual address** relates to the processor view of *a* **virtual address space** (e.g., associated with a process), whereas
- a **physical address** relates to the memory view of *the* **physical address space** (i.e., the actual RAM)

noting there is one virtual address space per process, and one physical address space period.

Concept (4)

- Goal: *use* the MMU to realise

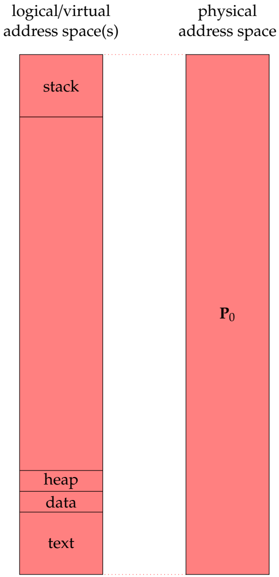
  1. **translation** of virtual to physical addresses,
  2. **protection** e.g., of the virtual address space associated with one process against access from another, and
  3. **sharing** i.e., controlled non-protection of (or overlap between) address spaces

  and hence *virtualise* the physical memory.

Mechanism: software (1)

- **Idea**: *no* MMU!
  - no translation,
  - no protection.
- **Features**:

address space(s) translated	×
address space(s) protected	×
address space(s) virtualised	×
address space(s) non-contiguous	×
req. hardware support	×
req. software (kernel) support	×
req. software (user) support	×

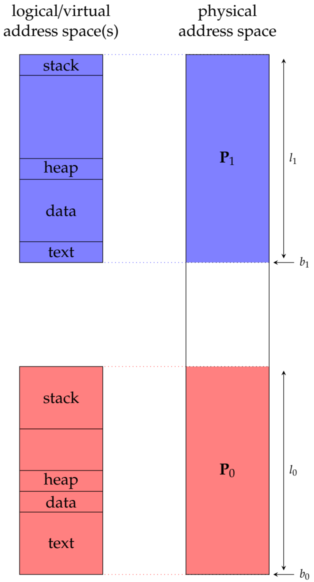


Mechanism: software (2)

- Idea: *no* MMU!
  - still no translation: either
    1. linker or
    2. loader

    relocates each address x wrt. some base b,
  - still no protection: assumes process will be "honest" st. $b < x < b + l$.
- Features:

address space(s) translated	×	×
address space(s) protected	×	×
address space(s) virtualised	×	×
address space(s) non-contiguous	×	×
req. hardware support	×	×
req. software (kernel) support	×	✓
req. software (user) support	✓	×

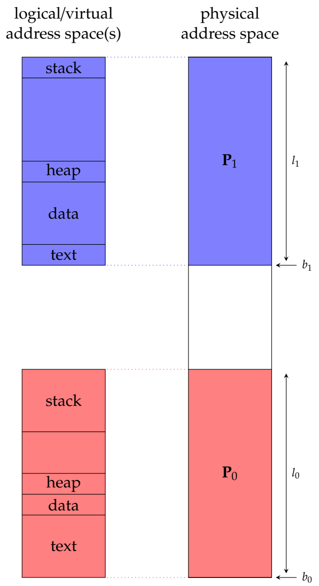


Mechanism: hardware-based per process segmentation (1)

- **Idea**: per process **segmented memory**.
  1. maintain a base and limit register per process,
  2. enforce
     $$b \le x < b + l$$
     and relocate as before, or
  3. enforce
     $$0 \le x < l$$
     and translate st.
     $$x \mapsto b + x.$$

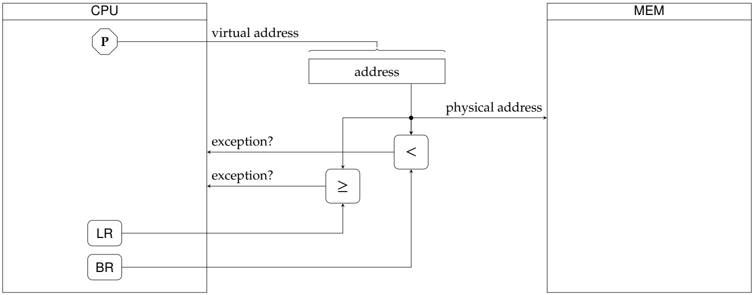
- **Features**:

address space(s) translated	×	✓
address space(s) protected	✓	✓
address space(s) virtualised	×	✓
address space(s) non-contiguous	×	×
req. hardware support	✓	✓
req. software (kernel) support	✓	✓
req. software (user) support	✓	×



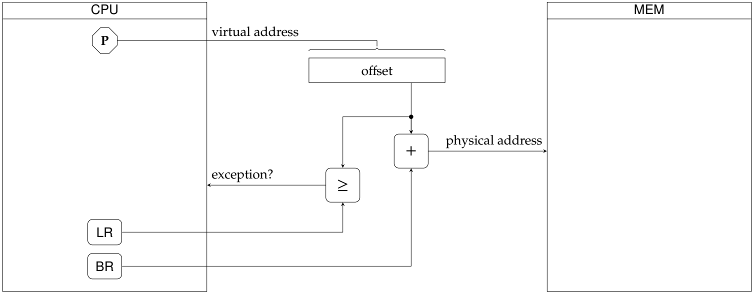
Mechanism: hardware-based per process segmentation (2)

- An implementation requires MMU-like hardware, e.g.,



Mechanism: hardware-based per process segmentation (2)

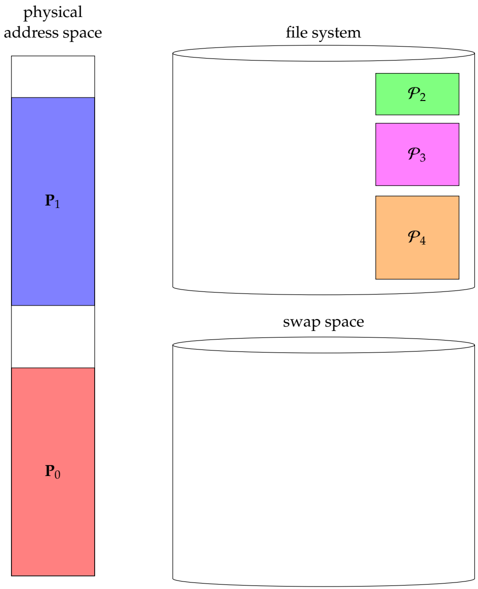
- An implementation requires MMU-like hardware, e.g.,



An Aside: allocation, swapping and fragmentation

- Problem: where should we load $\mathcal{P}_i$.
- Solution: we need
  1. an allocation algorithm, e.g.,
     - first-fit,
     - best-fit,
     - worst-fit,
     - ...

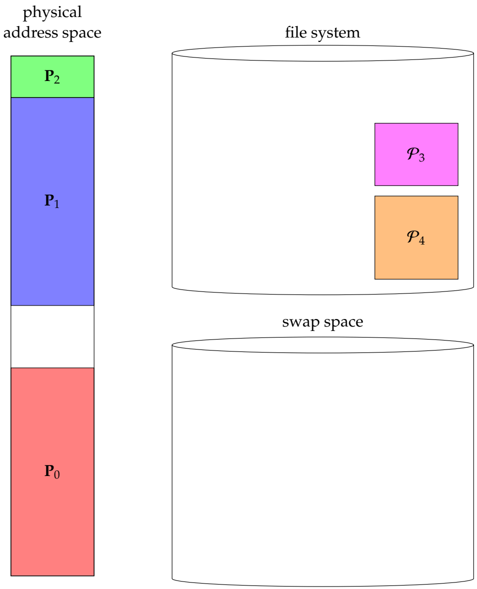
     and
  2. a data structure to capture the current allocation state.



An Aside: allocation, swapping and fragmentation

- Problem: where should we load $\mathcal{P}_i$.
- Solution: we need
  1. an allocation algorithm, e.g.,
     - first-fit,
     - best-fit,
     - worst-fit,
     - ...

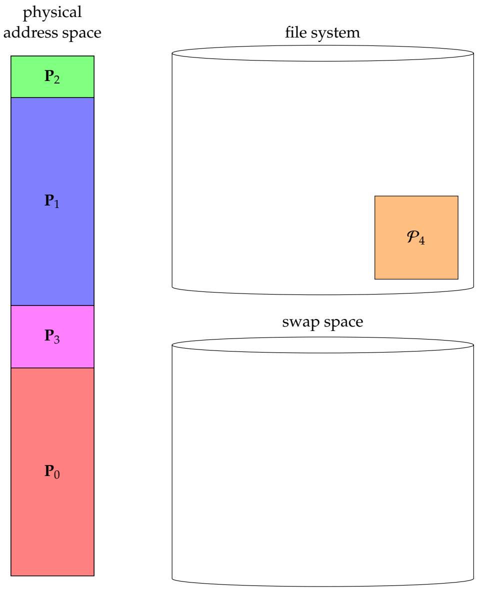
     and
  2. a data structure to capture the current allocation state.



An Aside: allocation, swapping and fragmentation

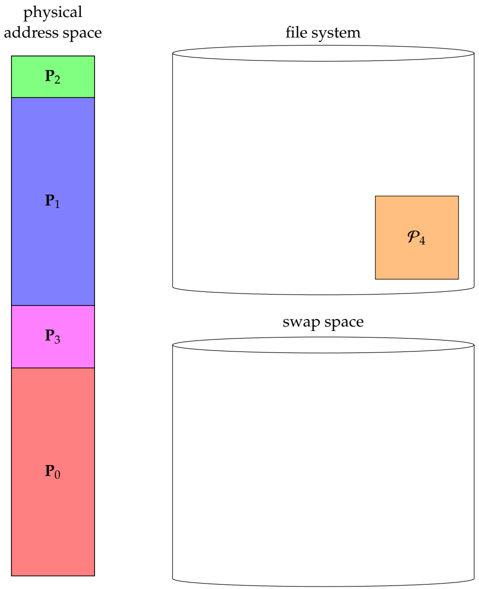
- Problem: where should we load $\mathcal{P}_i$.
- Solution: we need
  1. an allocation algorithm, e.g.,
     - first-fit,
     - best-fit,
     - worst-fit,
     - ...

     and
  2. a data structure to capture the current allocation state.



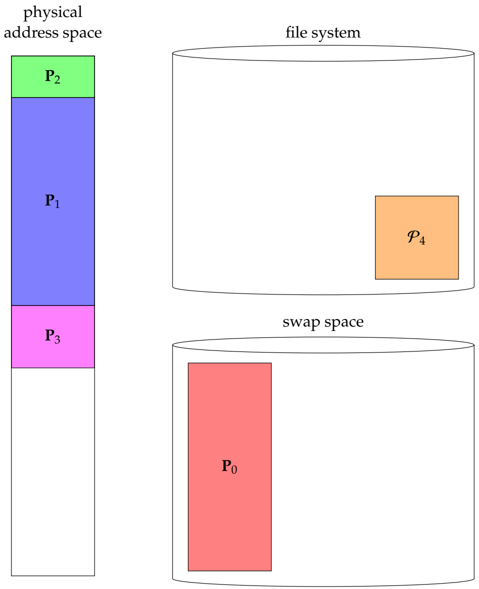
An Aside: allocation, swapping and fragmentation

- **Problem**:
  1. cases st.
     $$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$
     or
  2. cases st.
     $$\exists i \text{ st. } |\mathbf{P}_i| > |\mathsf{MEM}|.$$
- **Solution(s)**:
  1. **swapping**,
  2. some improvement to per process segmentation.



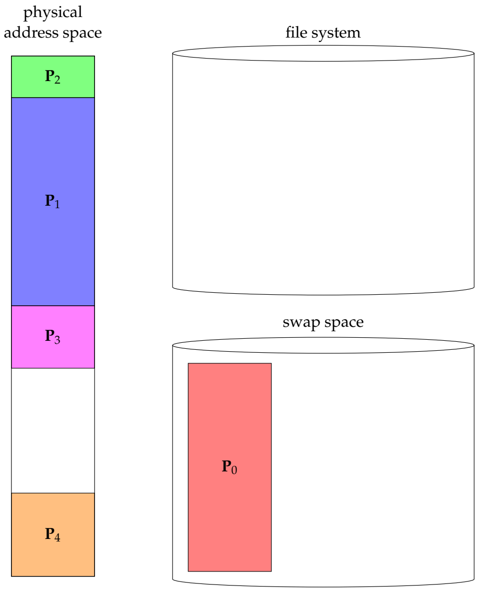
An Aside: allocation, swapping and fragmentation

- Problem:
  1. cases st.
     $$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\text{MEM}|$$
     or
  2. cases st.
     $$\exists i \text{ st. } |\mathbf{P}_i| > |\text{MEM}|.$$
- Solution(s):
  1. **swapping**,
  2. some improvement to per process segmentation.



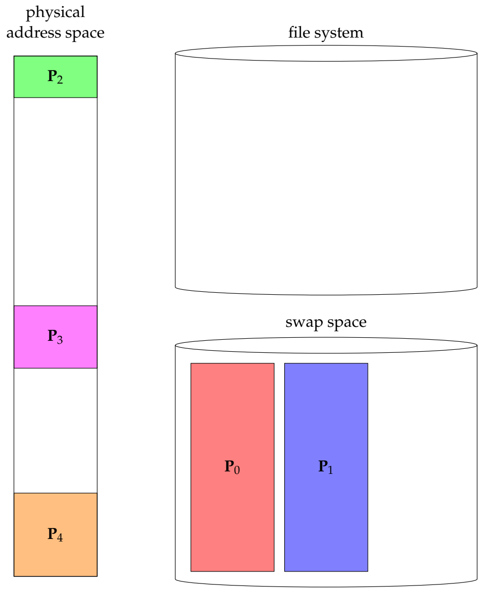
An Aside: allocation, swapping and fragmentation

- **Problem**:
  1. cases st.
  $$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$
  or
  2. cases st.
  $$\exists i \text{ st. } |\mathbf{P}_i| > |\mathsf{MEM}|.$$
- **Solution(s)**:
  1. **swapping**,
  2. some improvement to per process segmentation.



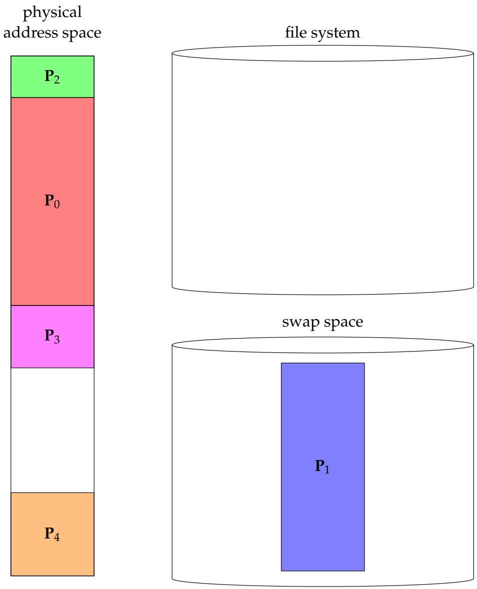
An Aside: allocation, swapping and fragmentation

- **Problem**:
  1. cases st.
     $$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$
     or
  2. cases st.
     $$\exists i \text{ st. } |\mathbf{P}_i| > |\mathsf{MEM}|.$$
- **Solution(s)**:
  1. **swapping**,
  2. some improvement to per process segmentation.



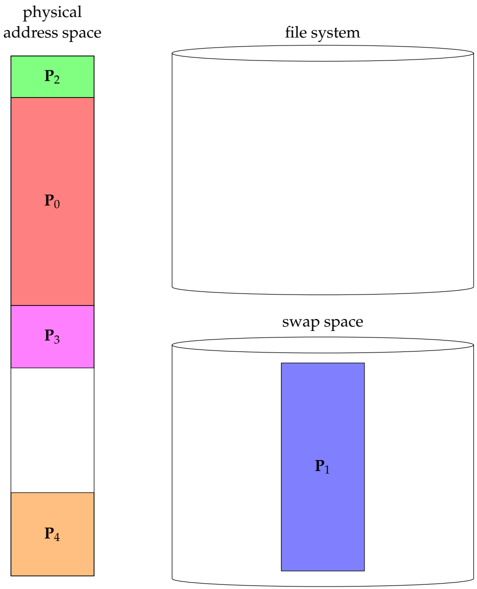
An Aside: allocation, swapping and fragmentation

- Problem:
  1. cases st.
     $$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$
     or
  2. cases st.
     $$\exists i \text{ st. } |\mathbf{P}_i| > |\mathsf{MEM}|.$$
- Solution(s):
  1. **swapping**,
  2. some improvement to per process segmentation.



An Aside: allocation, swapping and fragmentation

- Problem: **fragmentation**, namely
  1. **internal** (i.e., *within* allocations), or
  2. **external** (i.e., *between* allocations).
- Solution(s):
  - **compaction**,
  - some improvement to per process segmentation.



Mechanism: hardware-based per segment segmentation (1)

- Idea: per segment **segmented memory**.
  - maintain a **segment table** T per process,
  - let $t = \log_2(|T|)$, check

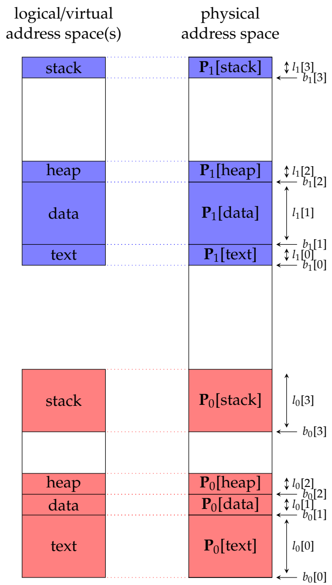
    $$0 \leq \mathrm{LSB}_{w-t}(x) < l[\mathrm{MSB}_t(x)],$$

    and translate st.

    $$x \mapsto b[\mathrm{MSB}_t(x)] + \mathrm{LSB}_{w-t}(x).$$

- Features:

address space(s) translated	✓
address space(s) protected	✓
address space(s) virtualised	✓
address space(s) non-contiguous	✓
req. hardware support	✓
req. software (kernel) support	✓
req. software (user) support	✓



Mechanism: hardware-based per segment segmentation (1)

- Idea: per segment **segmented memory**.
  - maintain a **segment table** T per process,
  - let $t = \log_2(|T|)$, check

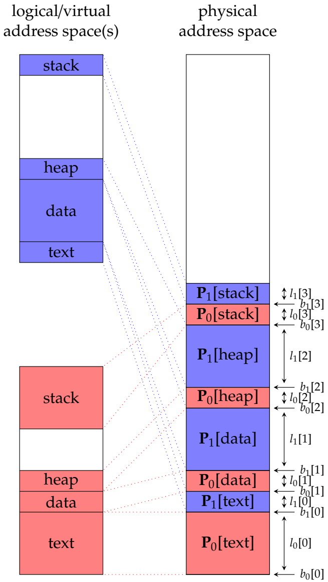
$$0 \le \mathrm{LSB}_{w-t}(x) < l[\mathrm{MSB}_t(x)],$$

and translate st.

$$x \mapsto b[\mathrm{MSB}_t(x)] + \mathrm{LSB}_{w-t}(x).$$

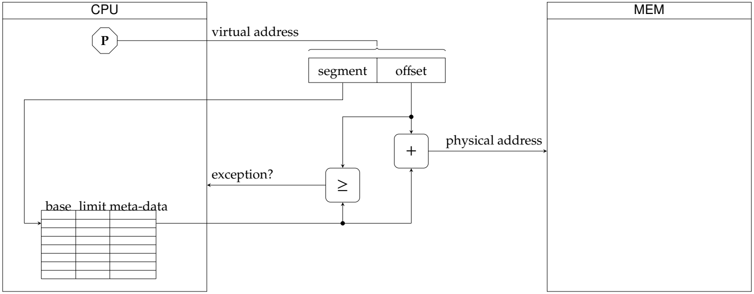
- Features:

address space(s) translated	✓
address space(s) protected	✓
address space(s) virtualised	✓
address space(s) non-contiguous	✓
req. hardware support	✓
req. software (kernel) support	✓
req. software (user) support	✓



Mechanism: hardware-based per segment segmentation (2)

- An implementation requires MMU-like hardware, e.g.,

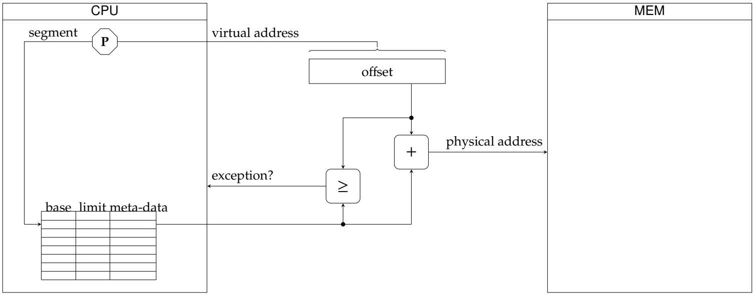


noting we *could* opt to index into the table via

1. one address, i.e., split address into a segment identifier and offset.

Mechanism: hardware-based per segment segmentation (2)

- An implementation requires MMU-like hardware, e.g.,



noting we *could* opt to index into the table via

1. one address, i.e., split address into a segment identifier and offset, or
2. two address, i.e., a dedicated segment identifier and offset.

Mechanism: hardware-based paging (1)

- Idea: **paged memory**.
  - fix $l = \rho$, and divide
    - virtual address space(s) into **pages**,
    - physical address space into **page frames**

    of l bytes in each case,
  - maintain a **page table** T per process,
  - let $t = \log_2(|T|)$, and translate st.

    $$x \mapsto b[\mathrm{MSB}_t(x)] \cdot l + \mathrm{LSB}_{w-t}(x).$$

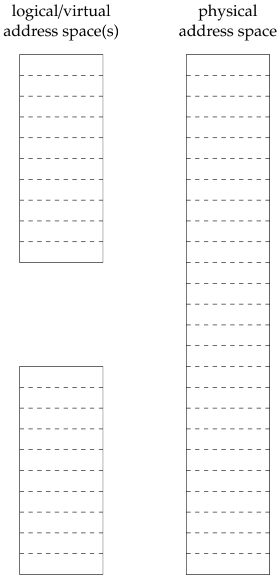
    noting no check is required since

    $$0 \leq \mathrm{LSB}_{w-t}(x) < l$$

    by definition.

- Features:

address space(s) translated	✓
address space(s) protected	✓
address space(s) virtualised	✓
address space(s) non-contiguous	✓
req. hardware support	✓
req. software (kernel) support	✓
req. software (user) support	✓



Mechanism: hardware-based paging (1)

- Idea: **paged memory**.
  - fix $l = \rho$, and divide
    - virtual address space(s) into **pages**,
    - physical address space into **page frames**

    of l bytes in each case,
  - maintain a **page table** T per process,
  - let $t = \log_2(|T|)$, and translate st.

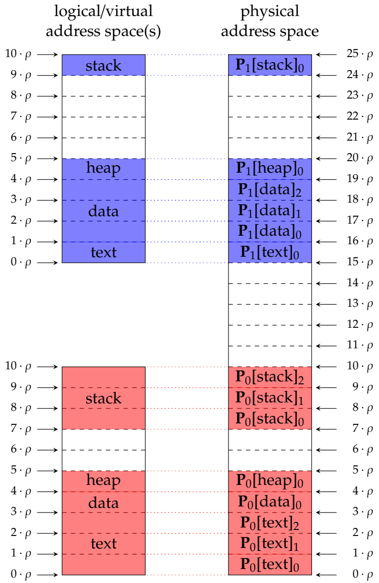
    $$x \mapsto b[\mathrm{MSB}_t(x)] \cdot l + \mathrm{LSB}_{w-t}(x).$$

    noting no check is required since

    $$0 \leq \mathrm{LSB}_{w-t}(x) < l$$

    by definition.
- Features:

address space(s) translated	✓
address space(s) protected	✓
address space(s) virtualised	✓
address space(s) non-contiguous	✓
req. hardware support	✓
req. software (kernel) support	✓
req. software (user) support	✓



Mechanism: hardware-based paging (1)

- Idea: **paged memory**.
  - fix $l = \rho$, and divide
    - virtual address space(s) into **pages**,
    - physical address space into **page frames**

    of l bytes in each case,
  - maintain a **page table** T per process,
  - let $t = \log_2(|T|)$, and translate st.

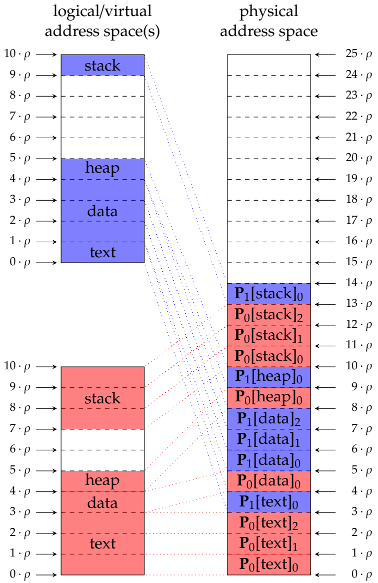
    $$x \mapsto b[\mathsf{MSB}_t(x)] \cdot l + \mathsf{LSB}_{w-t}(x).$$

    noting no check is required since

    $$0 \le \mathsf{LSB}_{w-t}(x) < l$$

    by definition.
- Features:

address space(s) translated	✓
address space(s) protected	✓
address space(s) virtualised	✓
address space(s) non-contiguous	✓
req. hardware support	✓
req. software (kernel) support	✓
req. software (user) support	✓



Mechanism: hardware-based paging (1)

- Idea: **paged memory**.
  - fix $l = \rho$, and divide
    - virtual address space(s) into **pages**,
    - physical address space into **page frames**

    of l bytes in each case,
  - maintain a **page table** T per process,
  - let $t = \log_2(|T|)$, and translate st.

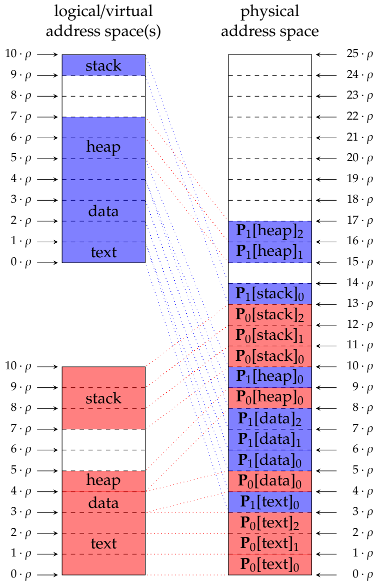
    $$x \mapsto b[\mathrm{MSB}_t(x)] \cdot l + \mathrm{LSB}_{w-t}(x).$$

    noting no check is required since

    $$0 \leq \mathrm{LSB}_{w-t}(x) < l$$

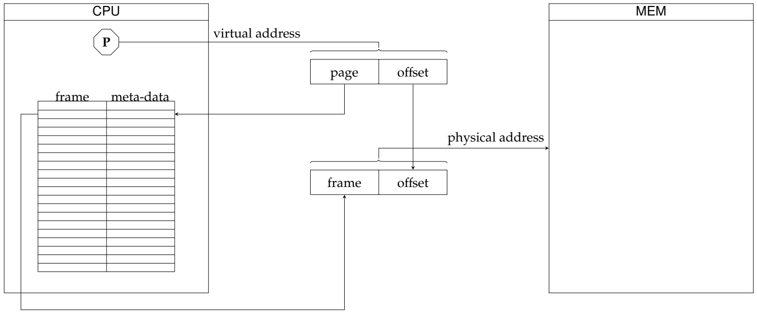
    by definition.
- Features:

address space(s) translated	✓
address space(s) protected	✓
address space(s) virtualised	✓
address space(s) non-contiguous	✓
req. hardware support	✓
req. software (kernel) support	✓
req. software (user) support	✓



Mechanism: hardware-based paging (2)

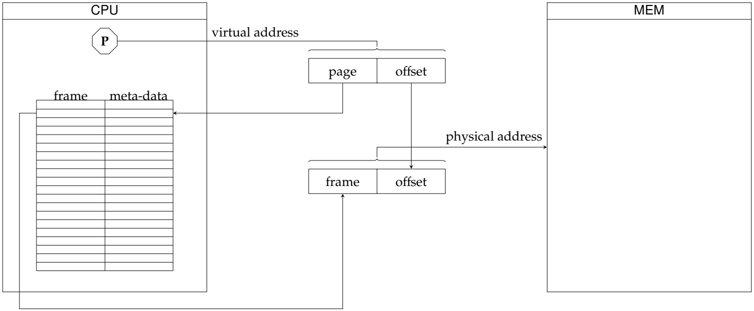
- An implementation requires MMU-like hardware, e.g.,



noting the page table consists of **Page Table Entries (PTEs)**.

Mechanism: hardware-based paging (3)

- Improvement #1: since the page table is *large*, we could

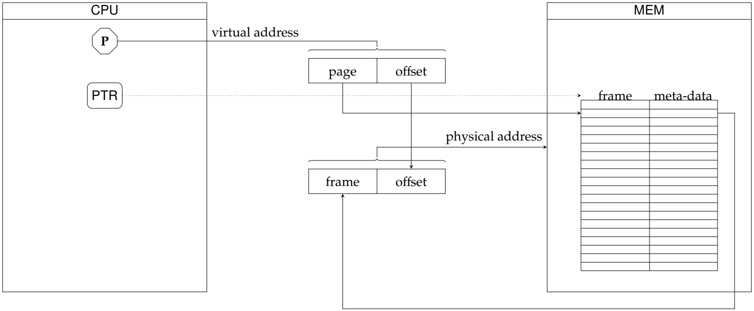


1. store the page table in memory,
2. point at the page table with a **Page Table Register (PTR)**, and
3. use a τ-entry **Translation Look-aside Buffer (TLB)** to cache the page table, noting
    - flush the TLB during a context switch, or
    - include a process identifier as a disambiguation tag,

   st. cached PTEs for one process cannot be used by another.

Mechanism: hardware-based paging (3)

- Improvement #1: since the page table is *large*, we could

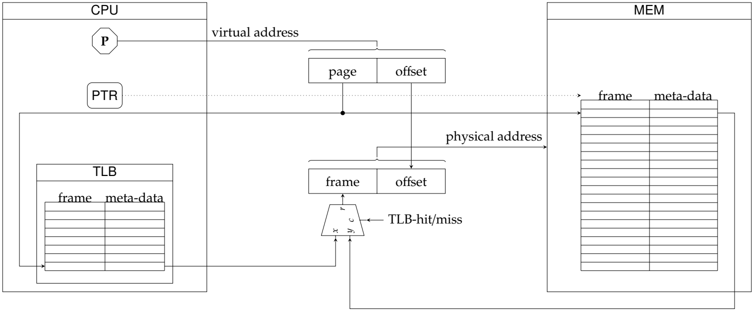


1. store the page table in memory,
2. point at the page table with a **Page Table Register (PTR)**, and
3. use a τ-entry **Translation Look-aside Buffer (TLB)** to cache the page table, noting
    - flush the TLB during a context switch, or
    - include a process identifier as a disambiguation tag,

   st. cached PTEs for one process cannot be used by another.

Mechanism: hardware-based paging (3)

- Improvement #1: since the page table is *large*, we could

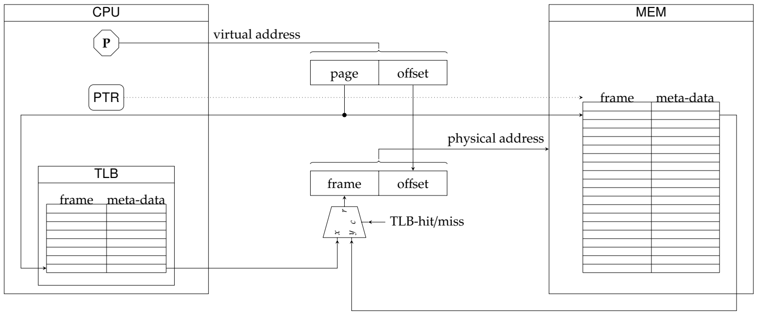


1. store the page table in memory,
2. point at the page table with a **Page Table Register (PTR)**, and
3. use a τ-entry **Translation Look-aside Buffer (TLB)** to cache the page table, noting
    - flush the TLB during a context switch, or
    - include a process identifier as a disambiguation tag,

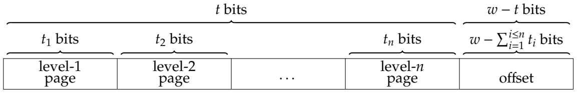
   st. cached PTEs for one process cannot be used by another.

Mechanism: hardware-based paging (4)

- Improvement #2: since the page table is *sparse*, we could



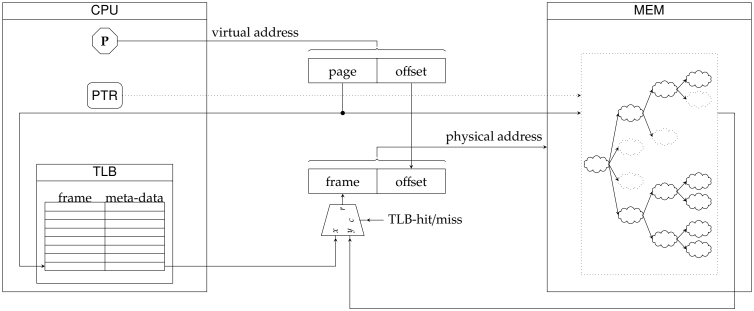
1. store the page table as a λ-level tree (vs. a list),
2. decompose original page number to index into each level, i.e.,



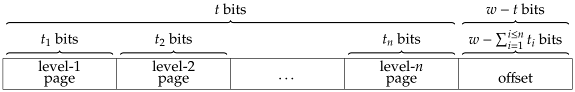
3. use a valid flag to indicate whether or not a sub-tree exists.

Mechanism: hardware-based paging (4)

- Improvement #2: since the page table is *sparse*, we could



1. store the page table as a λ-level tree (vs. a list),
2. decompose original page number to index into each level, i.e.,



3. use a valid flag to indicate whether or not a sub-tree exists.

Mechanism: hardware-based paging (5)

- ... *but*, we need to
  1. select various (non-independent) parameters,
  2. consider how to interface with the wider memory hierarchy, and
  3. handle various exceptions appropriately.

larger ρ	smaller ρ
less pages smaller page table more internal fragmentation	more pages larger page table less internal fragmentation
larger λ	smaller λ
longer walk finer grained sparsity	shorter walk coarse grained sparsity
larger τ	smaller τ
more overhead wrt. hardware less contention	less overhead wrt. hardware more contention

Mechanism: hardware-based paging (5)

- ... *but*, we need to
  1. select various (non-independent) parameters,
  2. consider how to interface with the wider memory hierarchy, and
  3. handle various exceptions appropriately.

- any given cache could potentially be placed before (i.e., deal with virtual addresses) *or* after (i.e., deal with physical addresses) the MMU,
- it can make sense to align the page size with the swap space (i.e., disk) transfer size.

Mechanism: hardware-based paging (5)

- ... *but*, we need to
  1. select various (non-independent) parameters,
  2. consider how to interface with the wider memory hierarchy, and
  3. handle various exceptions appropriately.

soft TLB miss	page is in memory
	page table entry isn't in TLB
hard TLB miss	page isn't in memory
	page table entry isn't in TLB
invalid page fault	page isn't in memory
	page isn't valid in page table
soft page fault	page is in memory
	page isn't valid in page table
hard page fault	page isn't in memory
	page is valid in page table
access fault	fails some check wrt. meta-data

Implementation: ARMv7-A (1)

- ARMv7-A supports *two* (very flexible) mechanisms via
  1. the **Protected Memory System Architecture (PMSA)** [5, Chapter B5] and
  2. the **Virtual Memory System Architecture (VMSA)** [5, Chapter B3]

  both of which are controlled via the co-processor interface [5, Chapters B4 and B6].

Implementation: ARMv7-A (2) VMSA

- Some details:
  1. It supports
     - a 32-bit virtual address space, and
     - *upto* a 40-bit physical address space

     with the latter realised via the **Large Physical Address Extension (LPAE)** ...
  2. ... and so two PTE formats [5, Section B3.3], namely

$$\begin{array}{lcll} \text{long} & \Rightarrow & \text{64-bit PTE} & \left\{ \begin{array}{l} \text{upto } \lambda = 3 \text{ levels} \\ \text{translates 32-bit to 40-bit address spaces at 4KiB granularity} \end{array} \right. \\ \text{short} & \Rightarrow & \text{32-bit PTE} & \left\{ \begin{array}{l} \text{upto } \lambda = 2 \text{ levels} \\ \text{translates 32-bit to 32-bit address spaces at 4KiB granularity} \end{array} \right. \\ \text{short} & \Rightarrow & \text{32-bit PTE} & \left\{ \begin{array}{l} \text{upto } \lambda = 2 \text{ levels} \\ \text{translates 32-bit to 40-bit address spaces at 16MiB granularity} \end{array} \right. \end{array}$$

     plus per-level variants of each.

Implementation: ARMv7-A (2) VMSA

- Some details:

3. It supports four page sizes [5, Section B3.3]

small page	$\Rightarrow$	$\rho =$ 4KiB	$\rightsquigarrow$	12-bit offsets
large page	$\Rightarrow$	$\rho =$ 64KiB	$\rightsquigarrow$	16-bit offsets
section	$\Rightarrow$	$\rho =$ 1MiB	$\rightsquigarrow$	20-bit offsets
super-section	$\Rightarrow$	$\rho =$ 16MiB	$\rightsquigarrow$	24-bit offsets

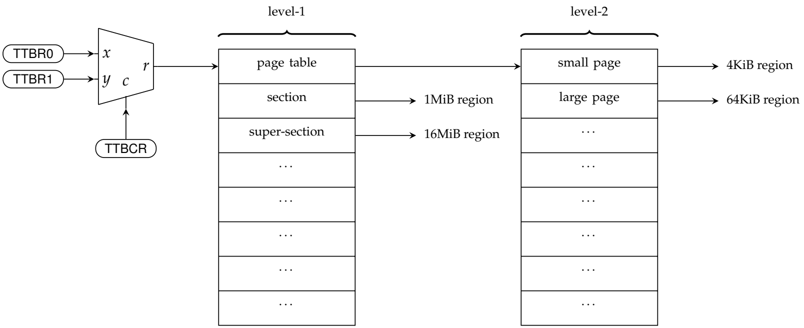
4. It uses two PTRs named TTBR0 and TTBR1, selecting one via TTBCR.

Implementation: ARMv7-A (3) VMSA

Example

Consider a (simple) example where we set $\lambda = 2$, utilise short PTEs only, utilise small pages only, and ignore functionality such as ASID.

The (general) 2-level page table organisation can be described as follows



although in this (specific) example, all level-1 PTEs will point to a level-2 page table, and all level-2 PTEs will point to a small page by definition.

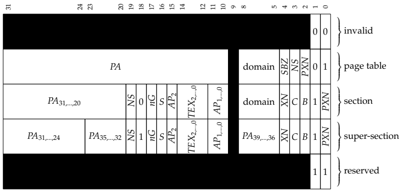
Implementation: ARMv7-A (3) VMSA

Example

Consider a (simple) example where we set $\lambda = 2$, utilise short PTEs only, utilise small pages only, and ignore functionality such as ASID.

The format of (short) PTEs

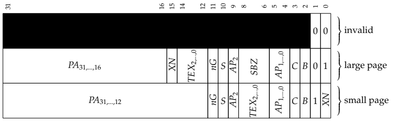
- at level-1 [5, Figure B3-4] is



31 24 23 20 19 18 17 16 15 14 12 11 10 9 8 5 4 3 2 1 0
0 0 invalid
PA domain SBZ NS PXN 0 1 page table
$PA_{31,\ldots,20}$ NS 0 nG S AP_2 $TEX_{2,\ldots,0}$ $AP_{1,\ldots,0}$ domain XN C B 1 PXN section
$PA_{31,\ldots,24}$ $PA_{35,\ldots,32}$ NS 1 nG S AP_2 $TEX_{2,\ldots,0}$ $AP_{1,\ldots,0}$ $PA_{39,\ldots,36}$ XN C B 1 PXN super-section
1 1 reserved

and

- at level-2 [5, Figure B3-5] is



31 16 15 14 12 11 10 9 8 6 5 4 3 2 1 0
0 0 invalid
$PA_{31,\ldots,16}$ XN $TEX_{2,\ldots,0}$ nG S AP_2 SBZ $AP_{1,\ldots,0}$ C B 0 1 large page
$PA_{31,\ldots,12}$ nG S AP_2 $TEX_{2,\ldots,0}$ $AP_{1,\ldots,0}$ C B 1 XN small page

Implementation: ARMv7-A (3) VMSA

Example

Consider a (simple) example where we set $\lambda = 2$, utilise short PTEs only, utilise small pages only, and ignore functionality such as ASID.

To load from some virtual address x, we proceed as follows:

```
if *PTE E for x is resident in the appropriate TLB(s)* then
    if *access control check for x and E passes* then
        load from $\mathsf{MEM}[E[PA] + x_{11,\ldots,0}]$
    else
        raise exception
    end
else
    if *if* $\mathrm{MSB}_n(x) = 0$ then
        load level-1 entry $E_1$ from $\mathsf{MEM}[\mathsf{TTBR0} + x_{31,\ldots,20}]$
    else
        load level-1 entry $E_1$ from $\mathsf{MEM}[\mathsf{TTBR1} + x_{31,\ldots,20}]$
    end
    if *$E_1$ is invalid or access control check fails* then raise exception
    load level-2 entry $E_2$ from $\mathsf{MEM}[E_1[PA] + x_{19,\ldots,12}]$
    if *$E_2$ is invalid or access control check fails* then raise exception
    load from $\mathsf{MEM}[E_2[PA] + x_{11,\ldots,0}]$
    update TLB(s)
end
```

Conclusions

Continued in next lecture ...

References

[1] Wikipedia: Memory management.
https://en.wikipedia.org/wiki/Memory_management.

[2] Wikipedia: Memory segmentation.
https://en.wikipedia.org/wiki/Memory_segmentation.

[3] Wikipedia: Translation Look-aside Buffer (TLB).
https://en.wikipedia.org/wiki/Translation_lookaside_buffer.

[4] Wikipedia: Virtual memory.
https://en.wikipedia.org/wiki/Virtual_memory.

[5] ARM Limited.
ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition.
Technical Report DDI-0406C, 2014.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html.

[6] M. Gorman.
Understanding the Linux Virtual Memory Manager.
Prentice Hall, 2004.
http://www.kernel.org/doc/gorman/.

[7] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 8: Memory management strategies.
In *Operating System Concepts* [9].

References

[8] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 9: Virtual-memory management.
In *Operating System Concepts* [9].

[9] A. Silberschatz, P.B. Galvin, and G. Gagne.
Operating System Concepts.
Wiley, 9th edition, 2014.

[10] A. N. Sloss, D. Symes, and C. Wright.
ARM System Developer's Guide: Designing and Optimizing System Software.
Elsevier, 2004.

[11] A. N. Sloss, D. Symes, and C. Wright.
Chapter 13: Memory protection units.
In *ARM System Developer's Guide: Designing and Optimizing System Software* [10].

[12] A. N. Sloss, D. Symes, and C. Wright.
Chapter 14: Memory management units.
In *ARM System Developer's Guide: Designing and Optimizing System Software* [10].

[13] A.S. Tanenbaum and H. Bos.
Chapter 3: Memory managament.
In *Modern Operating Systems*. Pearson, 4th edition, 2015.