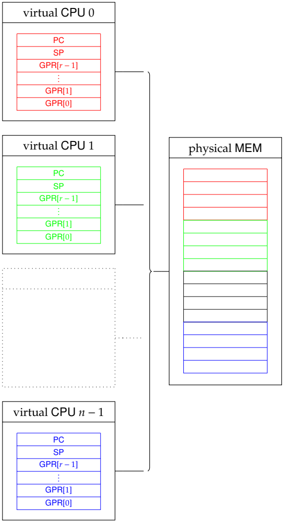
Concept: *virtualise* the memory

- 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 ...



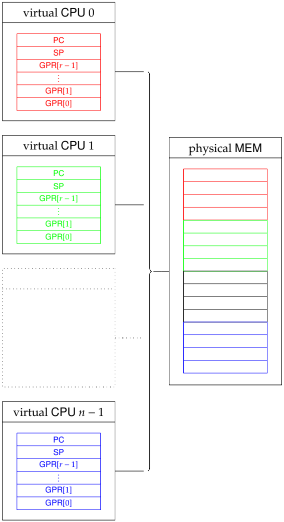
Concept: *virtualise* the memory

- 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,

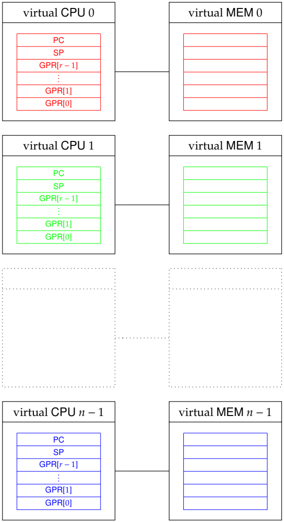
  warrant attention ...
- ... *but* we'll consider a narrower remit.



Concept: *virtualise* the memory

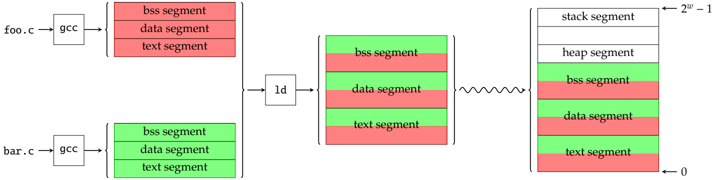
- Specifically, we want processes to
  1. *appear* to have dedicated access to the whole physical memory,
  2. have a larger footprint than physical memory *if* required,
  3. be protected wrt. access to their regions of the physical memory,
  4. to share regions of physical memory *if* required,
  5.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

  so, the question is, *how*?



Concept (1)

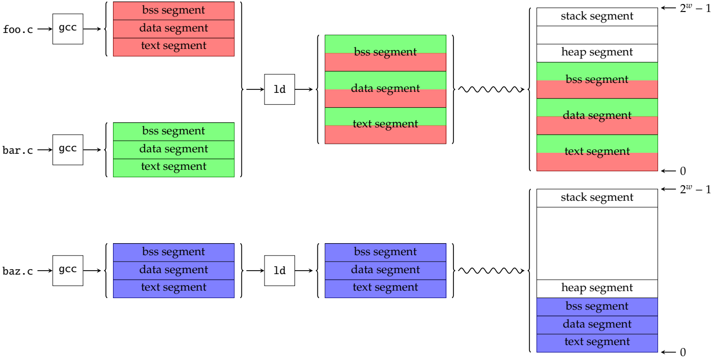
- Problem:



- the linker can combine object files, **relocating** addresses to suit,
- each program is compiled assuming it uses the *same* (i.e., a **uniform**) address space.

Concept (1)

- Problem:

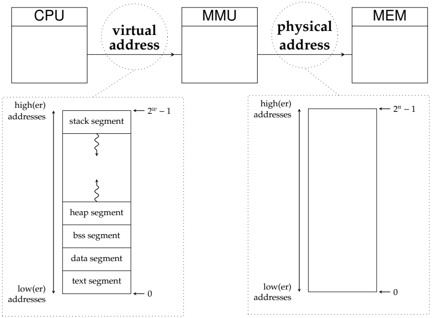


- the linker can combine object files, **relocating** addresses to suit,
- each program is compiled assuming it uses the *same* (i.e., a **uniform**) address space, *but*
- multi-programming means execution of *multiple* processes: clearly they cannot all occupy the same addresses in physical memory.

Concept (2)

Definition

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 (3)

Quote

Any problem in Computer Science can be solved by an extra level of indirection.

– *Wheeler* (http://en.wikiquote.org/wiki/Computer_science)

- Potential solution(s):

  1. **relocation** : at load-time
  2. **swapping** : at run-time
  3. **indirection** : at run-time

  with the ultimate aim to use an MMU to realise

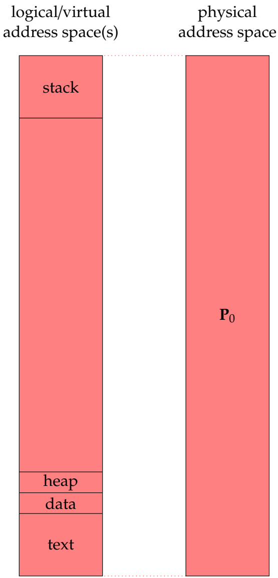
  1. **translation** of virtual to physical addresses,
  2. **protection** e.g., of the virtual address space associated with one process against access from another, and
  3. **sharing** i.e., controlled non-protection of (or overlap between) address spaces

  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	×

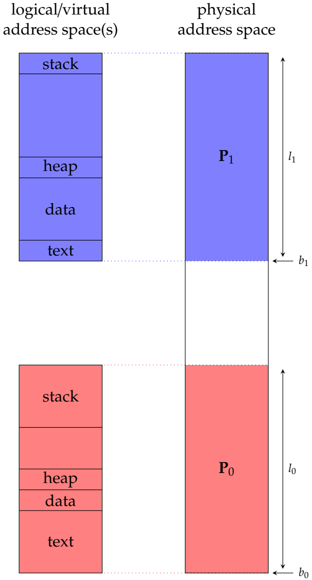


Mechanism: software (2)

- **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	✓	×

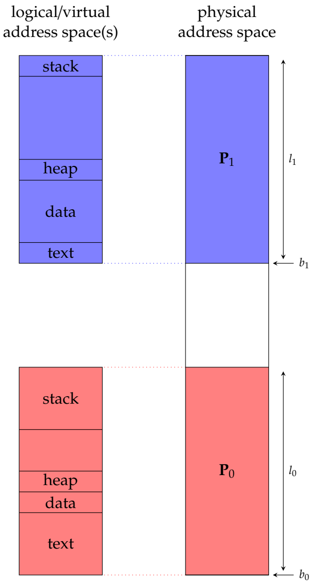


Mechanism: hardware-based per process segmentation (1)

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

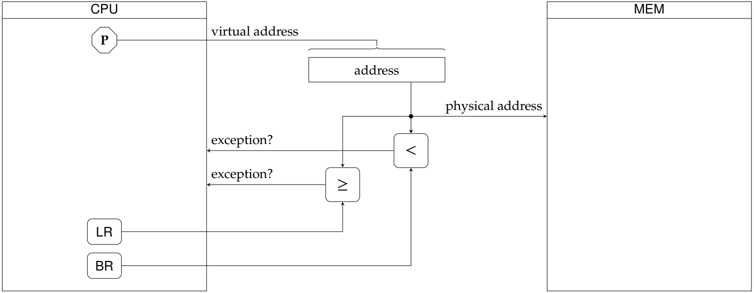
- Features:

address space(s) translated	$\times$	$\checkmark$
address space(s) protected	$\checkmark$	$\checkmark$
address space(s) virtualised	$\times$	$\checkmark$
address space(s) non-contiguous	$\times$	$\times$
req. hardware support	$\checkmark$	$\checkmark$
req. software (kernel) support	$\checkmark$	$\checkmark$
req. software (user) support	$\checkmark$	$\times$



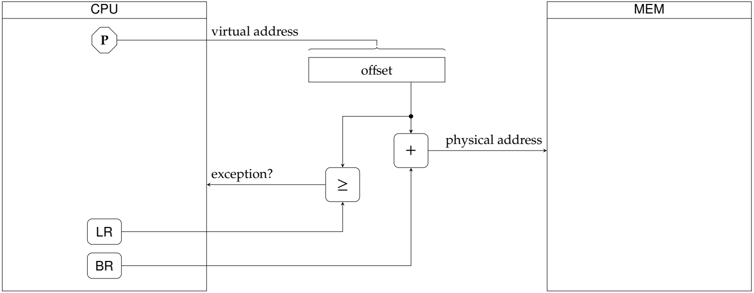
Mechanism: hardware-based per process segmentation (2)

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



Mechanism: hardware-based per process segmentation (2)

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

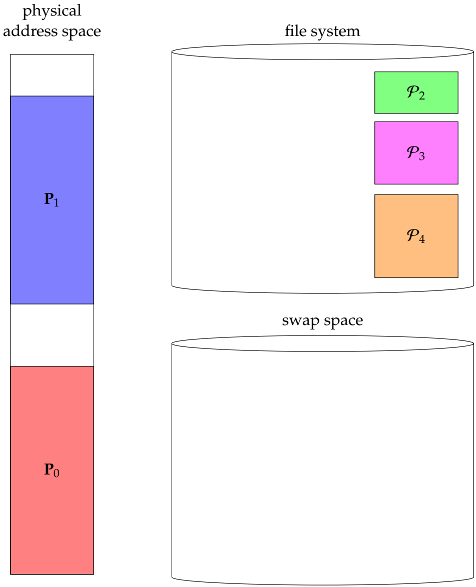


An Aside: allocation, swapping and fragmentation

- **Problem**: where do 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.

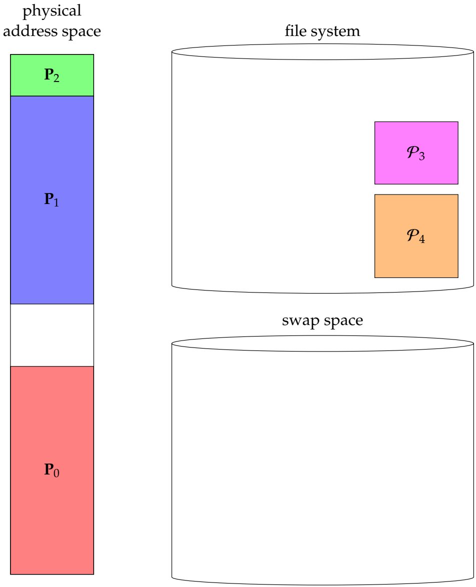


An Aside: allocation, swapping and fragmentation

- Problem: where do 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.

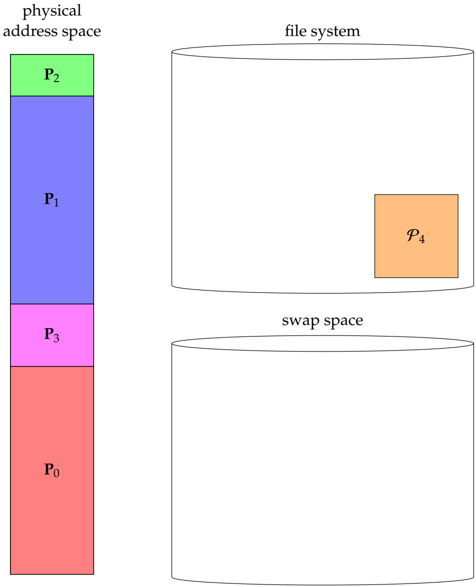


An Aside: allocation, swapping and fragmentation

- Problem: where do 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

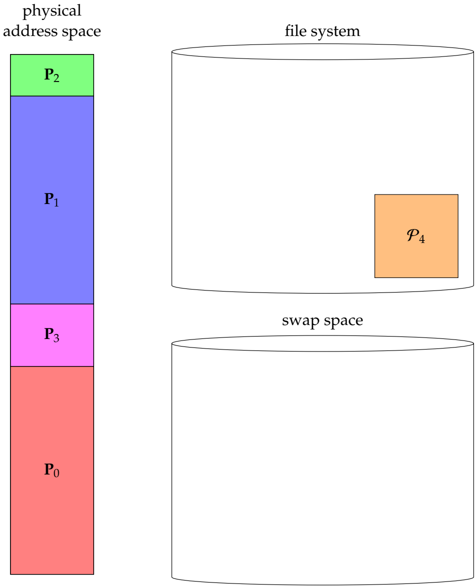
- **Problem**:
  1. cases st.

     $$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$

     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.

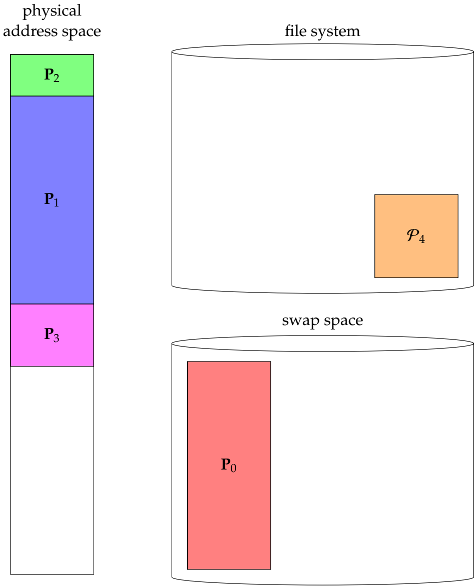
$$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$

  or

  2. cases st.

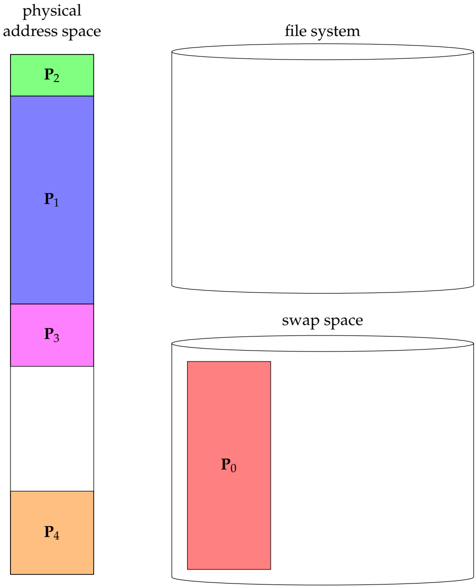
$$\exists i \text{ st. } |\mathbf{P}_i| > |\mathsf{MEM}|.$$

- **Solution(s)**:
  1. **swapping**,
  2. some improvement to per process segmentation.



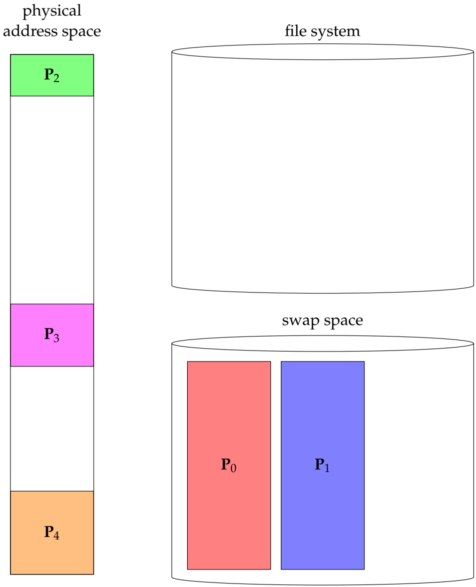
An Aside: allocation, swapping and fragmentation

- **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

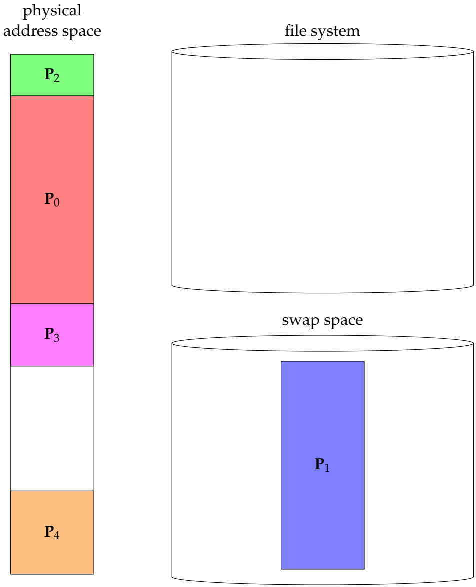
- **Problem**:
  1. cases st.

     $$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$

     or
  2. cases st.

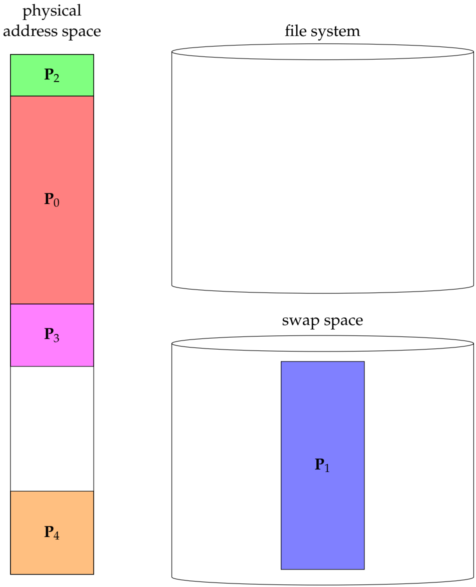
     $$\exists i \text{ st. } |\mathbf{P}_i| > |\mathsf{MEM}|.$$

- **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):
  - **de-fragmentation** (or **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

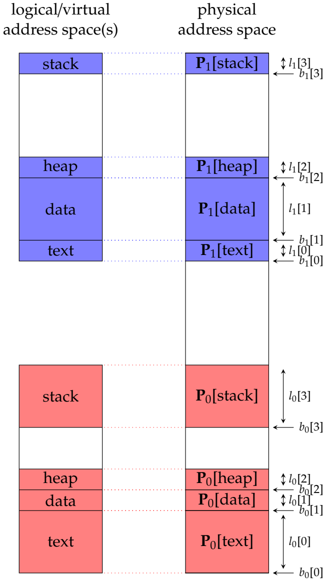
$$0 \le \mathrm{LSB}_{w-t}(x) < l[\mathrm{MSB}_t(x)],$$

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 (1)

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

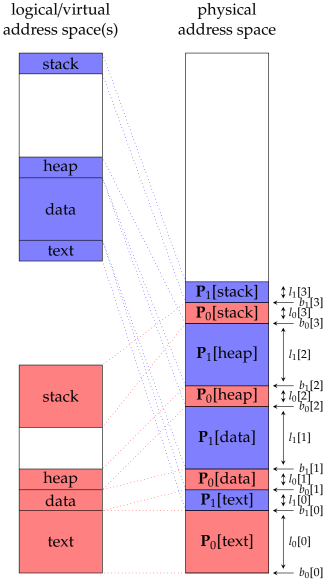
$$0 \leq \mathrm{LSB}_{w-t}(x) < l[\mathrm{MSB}_t(x)],$$

and translate st.

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

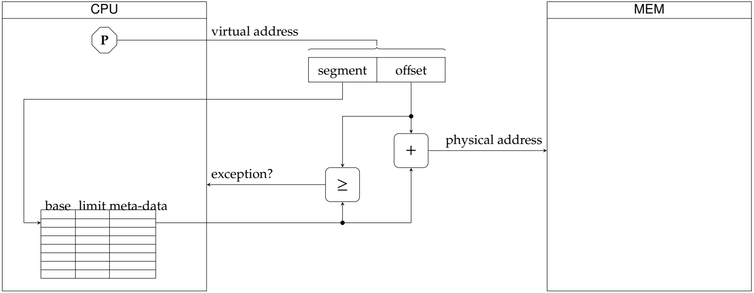
- Features:

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 (2)

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

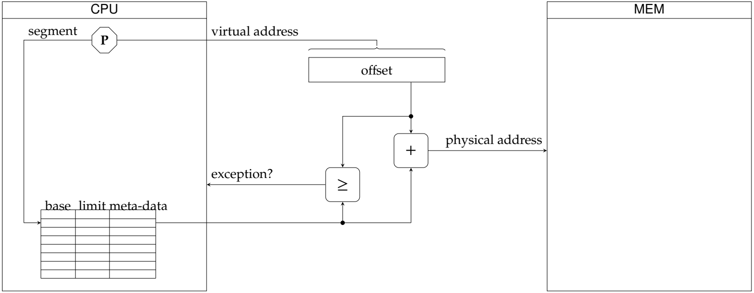


noting we *could* opt to index into the table via

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.

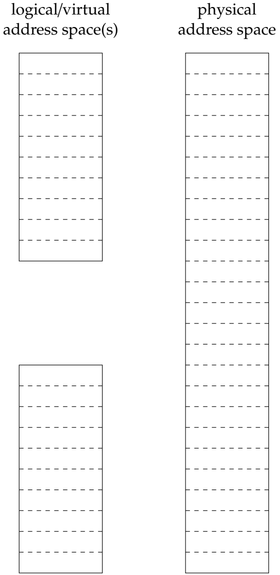
    $$x \mapsto b[\mathrm{MSB}_t(x)] \cdot l + \mathrm{LSB}_{w-t}(x).$$

    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.

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

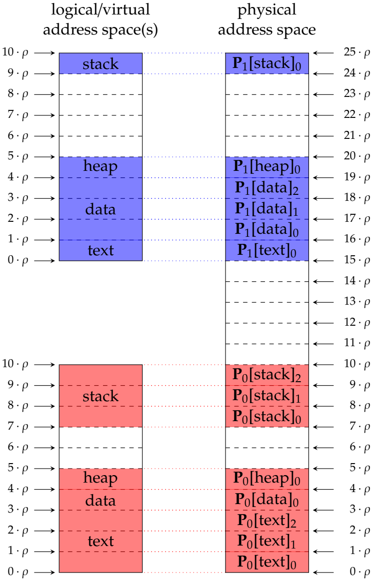
    noting no check is required since

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

    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.

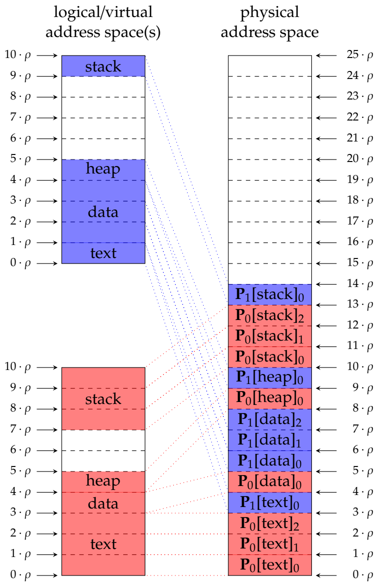
    $$x \mapsto b[\mathrm{MSB}_t(x)] \cdot l + \mathrm{LSB}_{w-t}(x).$$

    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.

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

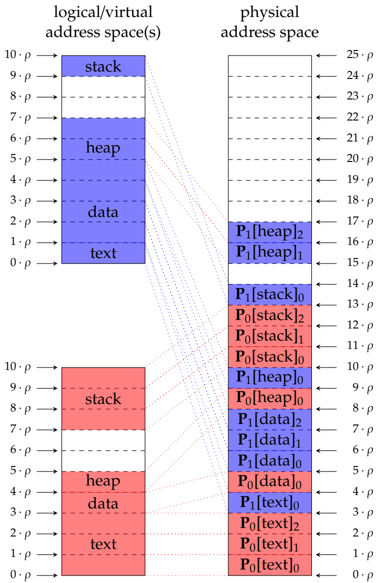
    noting no check is required since

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

    by definition.

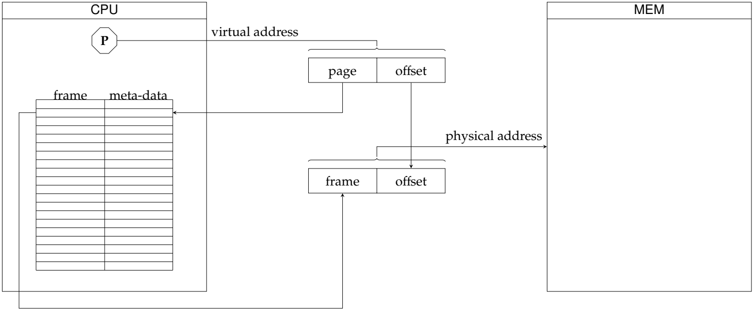
- Features:

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 (2)

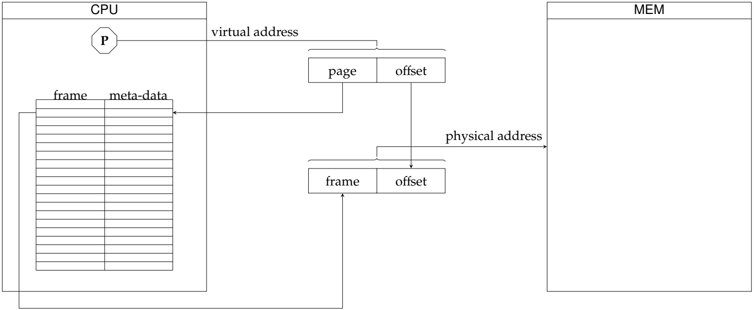
- An implementation requires MMU-like hardware, e.g.,



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

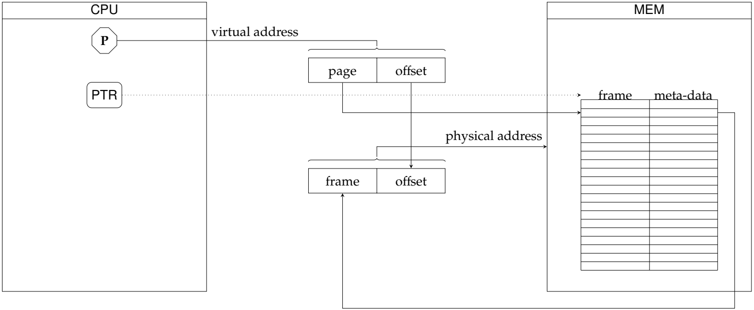


1. store the page table in memory,
2. point at the page table with a **Page Table Register (PTR)**, and
3. use a τ-entry **Translation Look-aside Buffer (TLB)** to cache the page table, noting
   - flush the TLB during a context switch, or
   - include a process identifier as a disambiguation tag,

   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

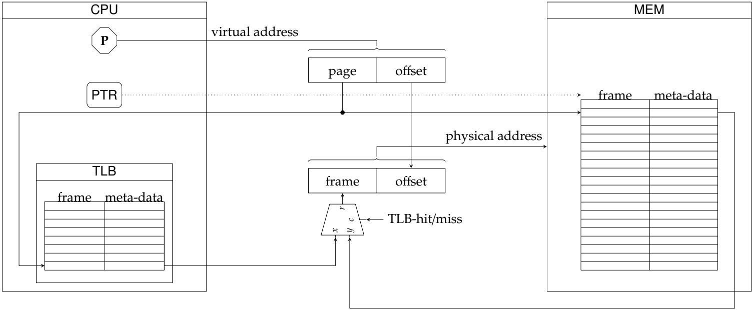


1. store the page table in memory,
2. point at the page table with a **Page Table Register (PTR)**, and
3. use a τ-entry **Translation Look-aside Buffer (TLB)** to cache the page table, noting
   - flush the TLB during a context switch, or
   - include a process identifier as a disambiguation tag,

   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

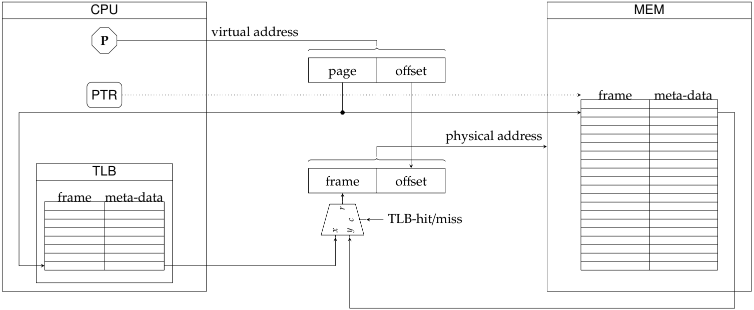


1. store the page table in memory,
2. point at the page table with a **Page Table Register (PTR)**, and
3. use a τ-entry **Translation Look-aside Buffer (TLB)** to cache the page table, noting
    - flush the TLB during a context switch, or
    - include a process identifier as a disambiguation tag,

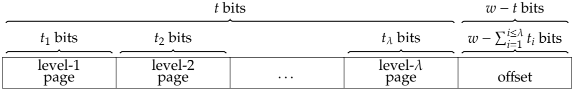
   st. cached PTEs for one process cannot be used by another.

Mechanism: hardware-based paging (4)

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



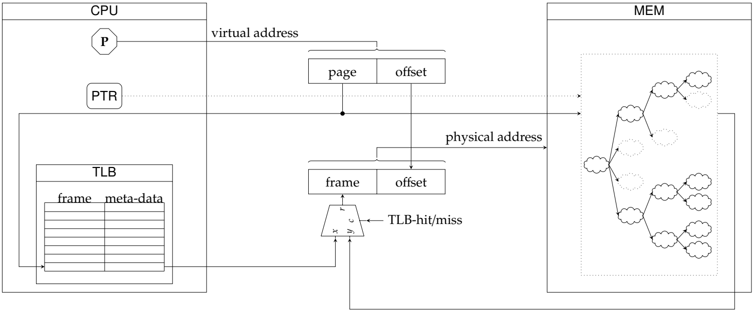
1. store the page table as a λ-level tree (vs. a list),
2. decompose original page number to index into each level, i.e.,



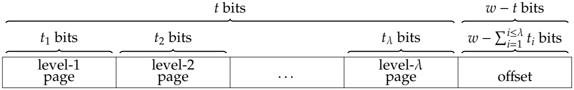
3. use a valid flag to indicate whether or not a sub-tree exists.

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

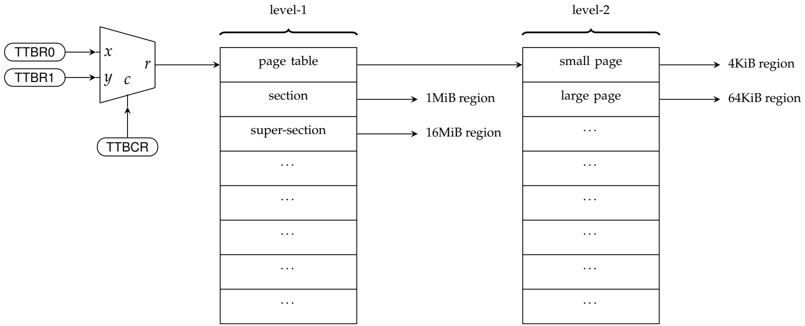
  1. the **Protected Memory System Architecture (PMSA)** [9, Chapter B5] and
  2. the **Virtual Memory System Architecture (VMSA)** [9, Chapter B3]

  both of which are controlled via the co-processor interface [9, Chapters B4 and B6].

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



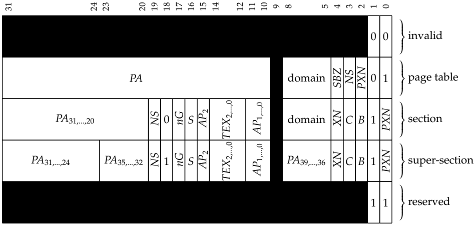
although in this (specific) example, all level-1 PTEs will point to a level-2 page table, and all level-2 PTEs will point to a small page by definition.

Example

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

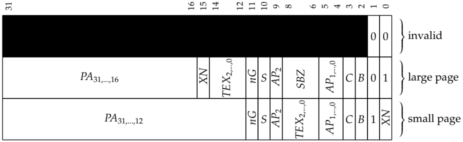
The format of (short) PTEs

- at level-1 [9, Figure B3-4] is



and

- at level-2 [9, Figure B3-5] is



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 MEM[E[PA] + x_{11,...,0}]
    else
        request exception
    end
else
    if MSB_n(x) = 0 then
        load level-1 entry E_1 from MEM[TTBR0 + x_{31,...,20}]
    else
        load level-1 entry E_1 from MEM[TTBR1 + x_{31,...,20}]
    end
    if E_1 is invalid or access control check fails then request exception
    load level-2 entry E_2 from MEM[E_1[PA] + x_{19,...,12}]
    if E_2 is invalid or access control check fails then request exception
    load from MEM[E_2[PA] + x_{11,...,0}]
    update TLB(s)
end
```

An Aside: a rough summary and/or interlude

1. no MMU – relocation

no multi-programming | add relocation

2. no MMU + relocation

no protection
no virtualisation

add single base/limit register
add comparison
add translation (i.e., base plus offset)

3. per process segmented memory

contiguous allocation only | add multiple base/limit registers

4. per segment segmented memory

limited flexibility

generalise from segments to pages
add in-memory page table
add TLB

5. paged memory

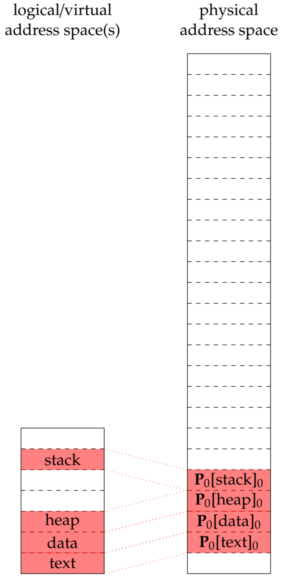
select concrete ρ, λ, and τ
select concrete PTE format
implement access algorithm

6. ARM VMSA

Policy (1)

- **Recall**:
  - paged memory divides
    - virtual address space(s) into **pages**,
    - physical address space into **page frames**

    of a fixed size,
  - a **page table** captures the mapping between pages and page frames,
  - the MMU (efficiently) uses page table entries to
    - translate between virtual address space(s) and physical address space, and
    - enforce protection.

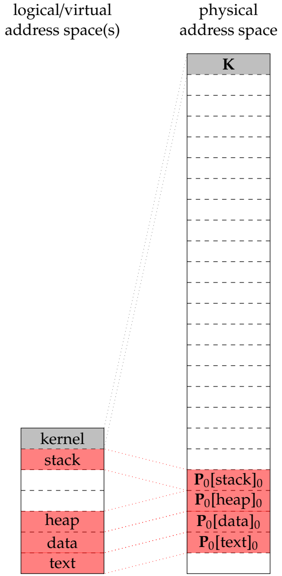


Policy (1)

- Idea: map the kernel address space into *every* user mode address space.
- Why?
  - clearly the kernel *requires* a protected address space, *but*
  - address space switches are pure overhead, and
  - this mapping avoids said overhead: the kernel address space is always resident

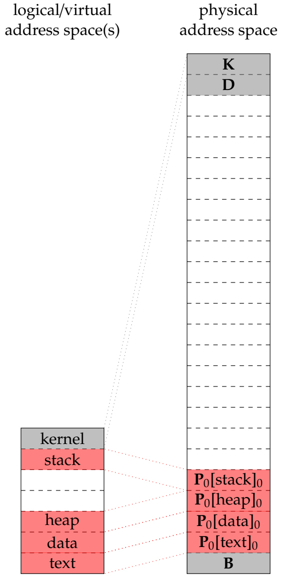
plus it suggests a more general ability to

- lock pages in physical memory, and
- protect pages.



Policy (1)

- **Idea**: avoid special-purpose regions in physical memory, e.g.,
  - regions relating to ROM-backed content such as the **Basic Input/Output System (BIOS)**, or
  - regions used for memory-mapped I/O, relating to device communication.

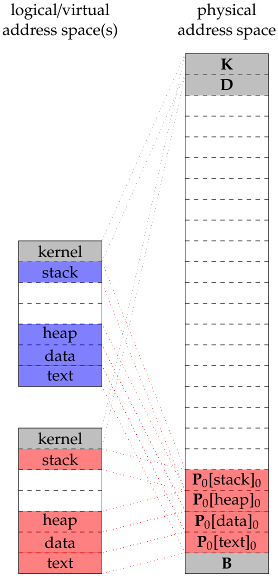


Policy (1)

- **Idea**: optimise `fork` via **copy-on-write**.
- **Why**?
  - naive `fork` must replicate address space of parent,
  - overhead introduced for unaltered pages, *so*
  - share address space of parent; allocate and copy shared page *only* when written to.

  plus it suggests a more general ability to

  - share pages between address spaces, and
  - optimise allocation of zero-filled regions.

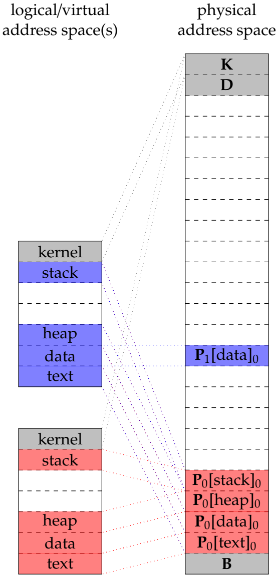


Policy (1)

- **Idea**: optimise `fork` via **copy-on-write**.
- **Why**?
  - naive `fork` must replicate address space of parent,
  - overhead introduced for unaltered pages, *so*
  - share address space of parent; allocate and copy shared page *only* when written to.

plus it suggests a more general ability to

- share pages between address spaces, and
- optimise allocation of zero-filled regions.



Policy (1)

- **Idea**: implement

  demand paging ≃ "lazy swapping"

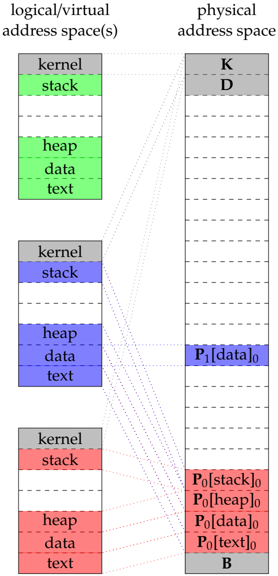
i.e.,

- naive program execution means
  1. initialise virtual address space,
  2. map pages to page frames,
  3. populate page frames, then
  4. start execution

  whereas
- demand paged program execution means
  1. initialise virtual address space,
  2. start execution, then
  3. whenever a page fault occurs, map page to page frame and populate

plus it suggests a more general ability to

- map files into an address space, e.g., via `mmap`.



Policy (1)

- **Idea**: implement

  demand paging ≃ "lazy swapping"

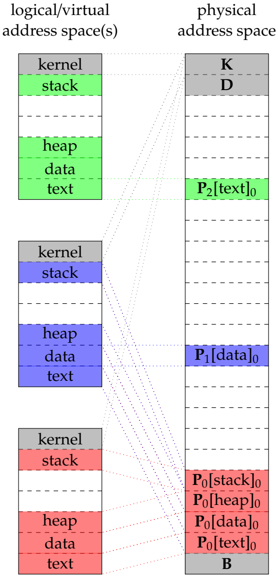
  i.e.,

  - naive program execution means
    1. initialise virtual address space,
    2. map pages to page frames,
    3. populate page frames, then
    4. start execution

    whereas
  - demand paged program execution means
    1. initialise virtual address space,
    2. start execution, then
    3. whenever a page fault occurs, map page to page frame and populate

  plus it suggests a more general ability to

  - map files into an address space, e.g., via `mmap`.



Policy (1)

- **Idea**: implement

  demand paging ≃ "lazy swapping"

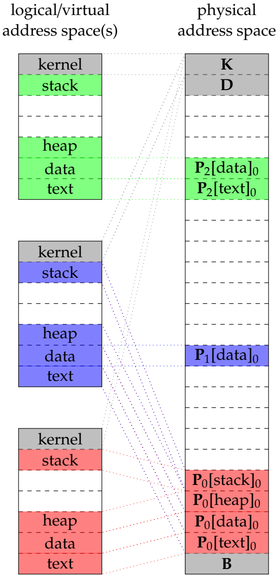
  i.e.,

  - naive program execution means
    1. initialise virtual address space,
    2. map pages to page frames,
    3. populate page frames, then
    4. start execution

    whereas
  - demand paged program execution means
    1. initialise virtual address space,
    2. start execution, then
    3. whenever a page fault occurs, map page to page frame and populate

  plus it suggests a more general ability to

  - map files into an address space, e.g., via `mmap`.



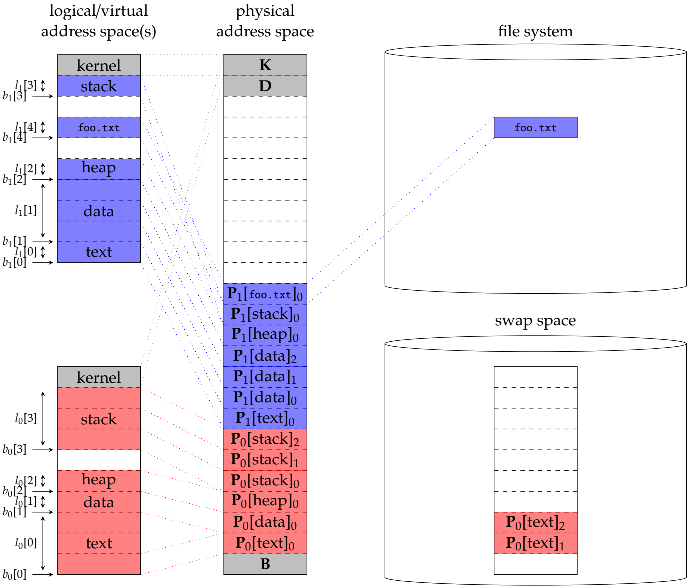
Implementation: demand paging (1)

- To implement demand paging, the kernel needs (at least):
  1. a per process
     1.1 allocation table,
     1.2 page table, and
     1.3 swap table (or disk map)
  2. a global page frame table,
  3. a page frame allocation policy, and
  4. a page allocation policy

  noting that

  - there are various options re. data structures, and
  - we're assuming management of the swap space is a separate problem.

Implementation: demand paging (2)



Implementation: demand paging (3)

Algorithm

Imagine we've attempted to load from some virtual address x:

		address (allocation table): valid (allocated)	address (allocation table): invalid (unallocated)
page (page table)	valid (mapped)		
page (page table)	invalid (unmapped)	allocate or swap-in	allocate

noting that

- the red cases cause an invalid page fault, whereas the green case might complete as is ...
- ... modulo special-cases such as copy-on-write,
- the allocation policy could fail, meaning it decides the right action is to request an exception, and
- the red cases demand we
  - allocate a page frame,
  - populate page frame with content (e.g., swap-in page) if need be,
  - update PTE to map page frame into virtual address space

  then restart the instruction.

Implementation: demand paging (4) – page frame allocation

- Problem:
  1. cases st.

$$\sum_{i=0}^{i<n} |\mathbf{P}_i| > |\mathsf{MEM}|$$

  and
  2. cases st.

$$\exists i \;\text{ st. }\; |\mathbf{P}_i| > |\mathsf{MEM}|$$

  remain problematic if we exhaust the number of page frames available.

- **Solution**: we allocate page frames via two dependant mechanisms, namely
  1. a page frame allocation algorithm, e.g., given m page frames
     - equal allocation: allocate m/n page frames, or
     - proportional allocation: allocate

       $$m \cdot \left(\frac{|\mathbf{P}_i|}{\sum_{i=0}^{i<n} |\mathbf{P}_i|} \right)$$

       page frames

     to each i-th of n processes, and
  2. a page frame replacement algorithm, e.g.,

     $$\begin{array}{rcl} \text{FIFO} & \Rightarrow & \text{select then replace oldest page} \\ \text{LRU} & \Rightarrow & \text{select then replace least-recently used page} \\ \text{LFU} & \Rightarrow & \text{select then replace least-frequently used page} \end{array}$$

     plus various LRU-approximations

  using them as follows ...

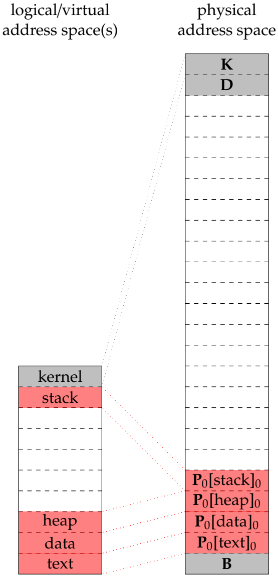
Implementation: demand paging (6) – page frame allocation

Algorithm

```
if ( page frame allocation algorithm allows new allocation ) ∧ ( an unallocated page frame exists ) then
    select unallocated page frame f
else
    select allocated page frame f using page frame replacement algorithm
    if page p resident in page frame f is dirty then
        store p in swap space
    else
        discard p
    end
end
return f
```

Implementation: demand paging (7) – page allocation

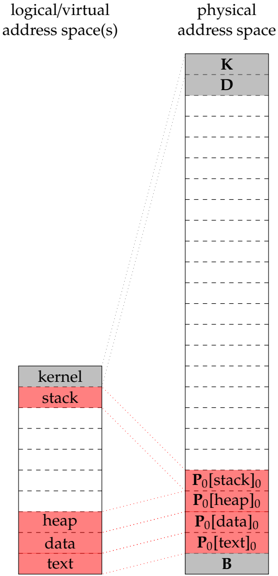
- Question: an initial allocation is fixed at load-time, but how are new pages allocated?



Implementation: demand paging (7) – page allocation

- **Solution**:
  1. implicit or automatic cases, e.g.,
     - enlargement of stack,

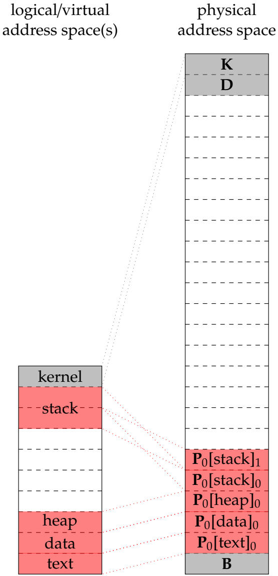
     and
  2. explicit or manual cases, e.g.,
     - enlargement of heap via `brk`, and
     - mapping a file via `mmap`.



Implementation: demand paging (7) – page allocation

- **Solution**:
  1. implicit or automatic cases, e.g.,
     - enlargement of stack,

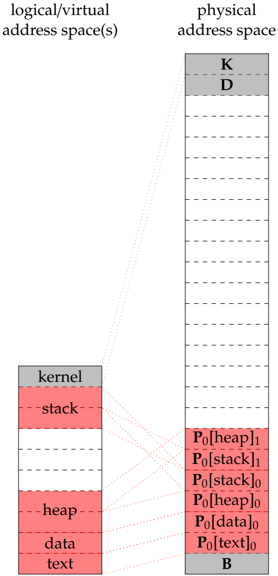
     and
  2. explicit or manual cases, e.g.,
     - enlargement of heap via `brk`, and
     - mapping a file via `mmap`.



Implementation: demand paging (7) – page allocation

- Solution:
  1. implicit or automatic cases, e.g.,
     - enlargement of stack,

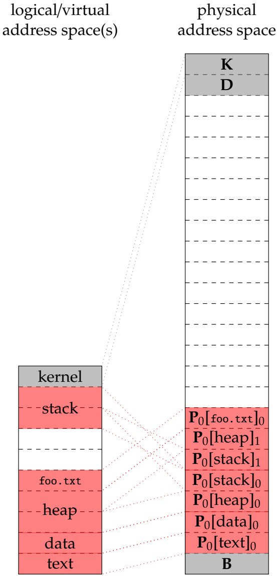
     and
  2. explicit or manual cases, e.g.,
     - enlargement of heap via `brk`, and
     - mapping a file via `mmap`.



Implementation: demand paging (7) – page allocation

- Solution:
  1. implicit or automatic cases, e.g.,
     - enlargement of stack,

     and
  2. explicit or manual cases, e.g.,
     - enlargement of heap via `brk`, and
     - mapping a file via `mmap`.



Implementation: demand paging (8) – performance

- Fact(s):
  - each process has a working set, $\mathcal{W}(\mathbf{P}_i)$, of pages.

Implementation: demand paging (8) – performance

- Fact(s):
  - each process has a working set, $\mathcal{W}(\mathbf{P}_i)$, of pages,
  - swapping-in or -out a page is pure, and significant overhead

$$\begin{array}{rcr} \text{memory access} & \simeq & 100\text{ns} \\ \text{disk access} & \simeq & 1000000\text{ns} \end{array}$$

    so ideally we minimise such events.

Implementation: demand paging (8) – performance

- Fact(s):
  - each process has a working set, $\mathcal{W}(\mathbf{P}_i)$, of pages,
  - swapping-in or -out a page is pure, and significant overhead

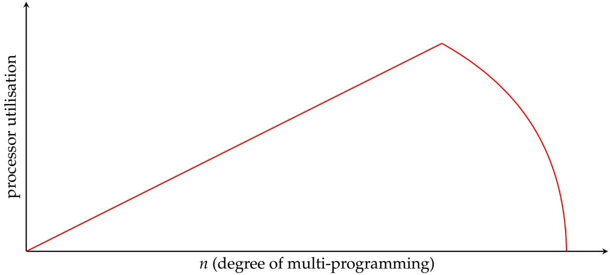
$$\begin{array}{rcr} \text{memory access} & \simeq & 100\text{ns} \\ \text{disk access} & \simeq & 1000000\text{ns} \end{array}$$

    so ideally we minimise such events, *but*
  - under a multi-programmed kernel with n resident processes,

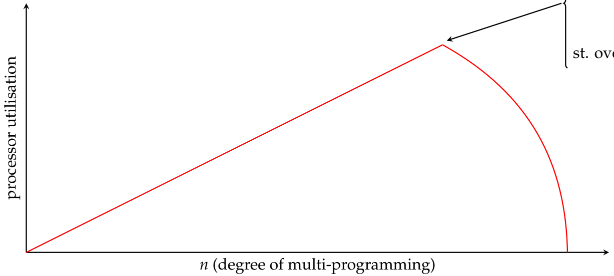
$$\begin{array}{lcl} \text{larger } n & \Rightarrow & \text{higher processor utilisation} \\ \text{larger } n & \Rightarrow & \text{higher contention wrt. page frames} \end{array}$$

  so, we find ...

Implementation: demand paging (9) – performance



Implementation: demand paging (9) – performance



thrashing, i.e.,

$$\sum_{i=0}^{i<n} \mathcal{W}(\mathbf{P}_i) > |\mathsf{MEM}|$$

st. overhead of swaping dominates.

- Potential mitigations against thrashing include
  1. keep track of **page fault frequency**, noting that

     $$\begin{array}{lcll} \text{too high} & \Rightarrow & \text{too few} & \text{page frames allocated} \\ \text{too low} & \Rightarrow & \text{too many} & \text{page frames allocated} \end{array}$$

     and tune parameters (e.g., page frame allocation algorithm) to suit,
  2. suspend process, i.e., set $\mathcal{W}(\mathbf{P}_i) = 0$ because it will not access memory until resumed,
  3. swap-out entire process, or
  4. terminate process.

Conclusions

- Take away points:
  - This is a broad and complex topic: it involves (at least)
    1. a hardware aspect:
       - the MMU
    2. a low(er)-level software aspect:
       - some data structures (e.g., page table),
       - a page fault handler,
       - a TLB fault handler
    3. a high(er)-level software aspect:
       - some data structures (e.g., allocation table),
       - a page allocation policy,
       - a page frame allocation policy,
       - any relevant POSIX system calls (e.g., `brk`)
  - Keep in mind that, even then,
    - we've excluded and/or simplified various (sub-)topics,
    - there are numerous trade-offs involved, meaning it is often hard to identify one ideal solution.
  - Focus on *understanding* demand paging: the performance of your software is strongly influenced by it, but remember

$$\text{demand paging} \subset \text{memory management}$$

    and, in some cases, *full* memory virtualisation isn't required since *protection* is often enough.

Additional Reading

- *Wikipedia: Memory management*. URL: http://en.wikipedia.org/wiki/Memory_management.
- *Wikipedia: Virtual memory*. URL: http://en.wikipedia.org/wiki/Virtual_memory.
- M. Gorman. *Understanding the Linux Virtual Memory Manager*. http://www.kernel.org/doc/gorman/. Prentice Hall, 2004.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 8: Memory management strategies". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 9: Virtual-memory management". In: *Operating System Concepts*. 9th ed. Wiley, 2014.
- A.S. Tanenbaum and H. Bos. "Chapter 3: Memory managament". In: *Modern Operating Systems*. 4th ed. Pearson, 2015.
- A. N. Sloss, D. Symes, and C. Wright. "Chapter 13: Memory protection units". In: *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004.
- A. N. Sloss, D. Symes, and C. Wright. "Chapter 14: Memory management units". In: *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004.

References

[1] *Wikipedia: Memory management.* URL: http://en.wikipedia.org/wiki/Memory_management (see p. 69).

[2] *Wikipedia: Virtual memory.* URL: http://en.wikipedia.org/wiki/Virtual_memory (see p. 69).

[3] M. Gorman. *Understanding the Linux Virtual Memory Manager.* http://www.kernel.org/doc/gorman/. Prentice Hall, 2004 (see p. 69).

[4] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 8: Memory management strategies". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 69).

[5] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 9: Virtual-memory management". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see p. 69).

[6] A. N. Sloss, D. Symes, and C. Wright. "Chapter 13: Memory protection units". In: *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004 (see p. 69).

[7] A. N. Sloss, D. Symes, and C. Wright. "Chapter 14: Memory management units". In: *ARM System Developer's Guide: Designing and Optimizing System Software*. Elsevier, 2004 (see p. 69).

[8] A.S. Tanenbaum and H. Bos. "Chapter 3: Memory managament". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see p. 69).

[9] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R edition.* Tech. rep. DDI-0406C. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406c/index.html. 2014 (see pp. 39, 41).