Concurrent Computing (Operating Systems)

Daniel Page

Department of Computer Science,
University Of Bristol,
Merchant Venturers Building,
Woodland Road,
Bristol, BS8 1UB. UK.
⟨csdsp@bristol.ac.uk⟩

February 9, 2018

Keep in mind there are *two* PDFs available (of which this is the latter):

1. a PDF of examinable material used as lecture slides, and
2. a PDF of non-examinable, extra material:
   - the associated notes page may be pre-populated with extra, written explaination of material covered in lecture(s), plus
   - anything with a "grey'ed out" header/footer represents extra material which is useful and/or interesting but out of scope (and hence not covered).

Notes:

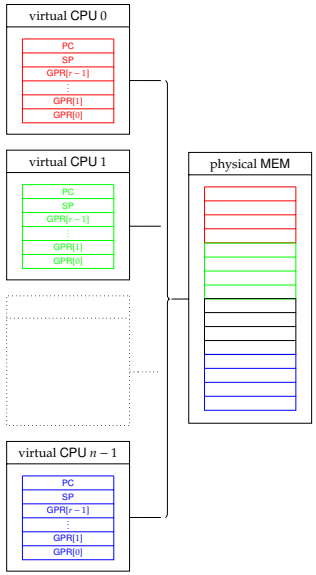
Notes:

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 ...



Notes:

- The management of memory the kernel uses is sort of a special-case. For example, it might be specialised to suit a need for contiguous allocation or specific physical addresses (e.g., for memory mapped I/O).

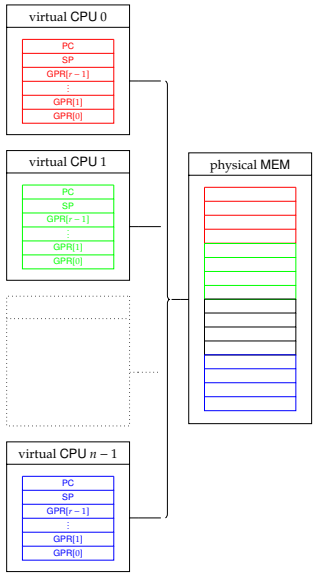
COMS20001 lecture: week #16 and #17

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.



Notes:

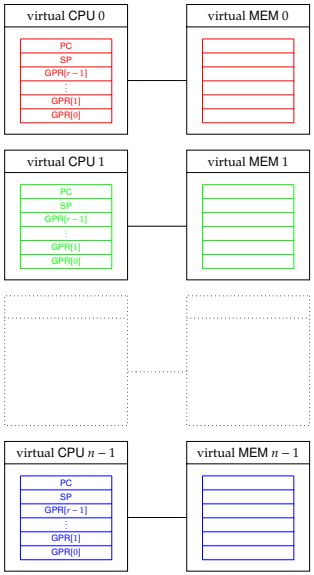
- The management of memory the kernel uses is sort of a special-case. For example, it might be specialised to suit a need for contiguous allocation or specific physical addresses (e.g., for memory mapped I/O).

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

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*?

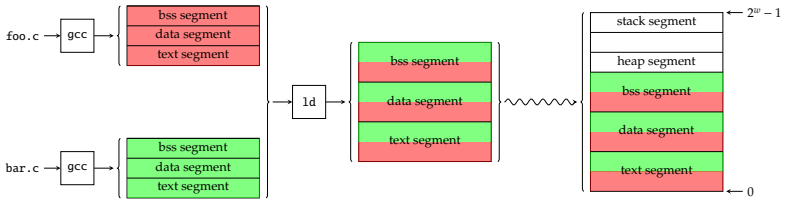


Notes:

- The management of memory the kernel uses is sort of a special-case. For example, it might be specialised to suit a need for contiguous allocation or specific physical addresses (e.g., for memory mapped I/O).

Concept (1)

- Problem:

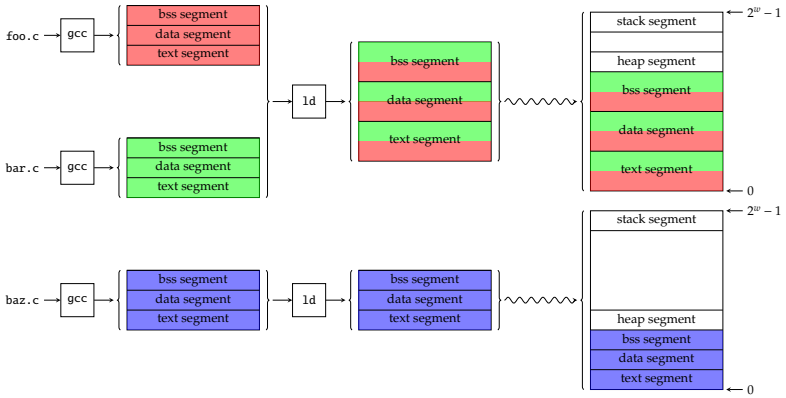


- the linker can combine object files, **relocating** addresses to suit,
- each program is compiled assuming it uses the *same* (i.e., a **uniform**) address space.

Notes:

Concept (1)

- Problem:



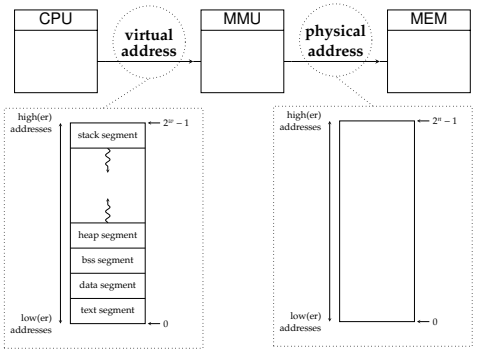
  - the linker can combine object files, **relocating** addresses to suit,
  - each program is compiled assuming it uses the *same* (i.e., a **uniform**) address space, *but*
  - multi-programming means execution of *multiple* processes: clearly they cannot all occupy the same addresses in physical memory.

Notes:

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.

Notes:

- In certain contexts, more specific terms than MMU might be used. For instance, ARM carefully distinguishes between a Memory *Protection* Unit (MPU) which supports protection *only*, and a Memory *Management* Unit (MMU) which supports protection *plus* translation.
- It is important to see that
  - A linear address space is just an ordered, contiguous set of addresses, e.g.,

    $$\{0, 1, \ldots\}$$

  - A physical address space is the set of addresses that are used to access elements in the physical, hardware-backed memory: given n-bit physical addresses, the associated address space is

    $$\{0, 1, \ldots, 2^n - 1\}$$

    meaning the memory (e.g, the RAM) has 2^n addressable elements (typically 8-bit bytes, cf. byte addressable).
  - A virtual address space is the set of addresses that a process can use, e.g., via load and store instructions; each access is satisfied by the MMU and, ultimately, the main memory. The virtual address space is often fixed by the processor word size w, because the process can naturally form and so use w-bit addresses. As such, we expect the set to be

    $$\{0, 1, \ldots, 2^w - 1\}$$

    noting that we do not *require* w to equal n.

  Note that the virtual address space will be sparse; this results from organisation of the process image into segments, meaing the sparsity has a structure (vs. being *randomly* sparse).
- Given the goals, you could view the physical address space of a process as a protection domain: it defines the set of (concrete) addresses that should be isolated wrt. *other* processes st. they cannot be read or written to.
  For example, imagine two processes each perform operations on a (separate) variable `x` via a pointer `p` which is equal to `&x`. The values of each `p` could legitimately be equal, since the two variables *could* have the same virtual address within the (separate) virtual address spaces of the respective processes. But when translated by the MMU into physical addresses, those must be different: the two variables cannot physically exist at the same address, because otherwise they couldn't have different values (which sort of means the processes aren't really different, or at least that their use of memory is not virtualised as intended).

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

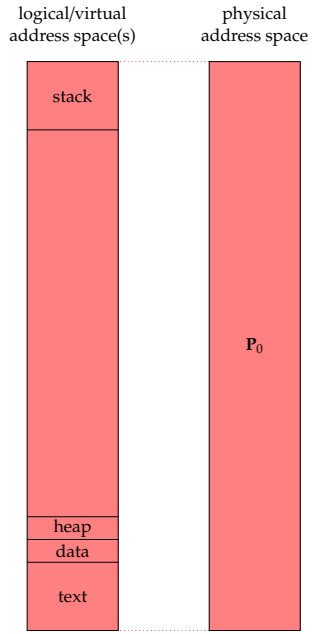
  and hence *virtualise* the physical memory.

Notes:

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	×



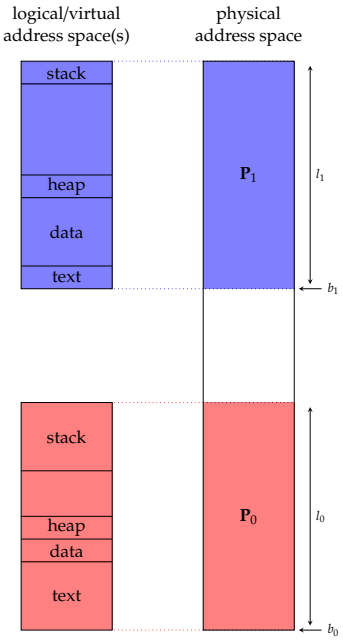
Notes:

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	✓	×



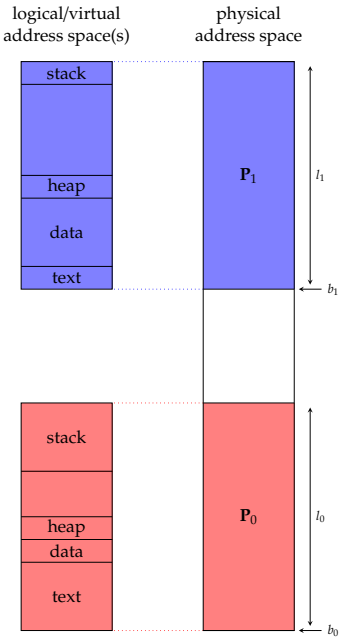
Notes:

- – As described, the address space of each process is captured in a single region with a notional base and limit; this is implemented at either compile- and/or load-time by a linker and/or loader. This is not *necessarily* a limitation, however. More specifically, the linker and/or loader could realise *any* layout provided appropriate relocation is performed: neither contiguous inter- *nor* intra-region allocation in physical memory is required for the processes to function correctly, even if one or other is preferable for ease of management.
  – There is no protection offered in this design. Although each resident process has a notional base and limit, this is implemented at compile- or load-time in software (via via linker or loader) so there is nothing to prevent one from accessing the address space of another (or in fact *any* address) at run-time.

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

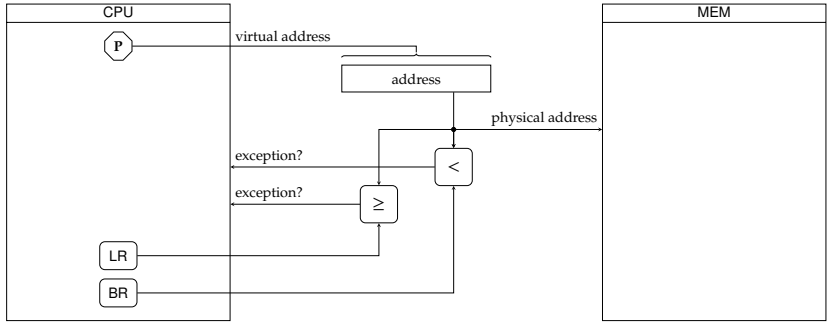


Notes:

- – The address space of each process is described by a single region, but now with an actual (vs. the previous, notional) base and limit. The region therefore must, by definition, be contiguous in physical memory so intra-region allocation need not be performed by the kernel. However, it *does* need to manage allocation of the regions themselves, which can be collectively non-contiguous. For example, it needs to ensure the allocation is st. regions do not unintentionally overlap.
  – A protected address space for each process is now enforced, rather than just assumed, using the additional hardware: an access to address x must be st.
    $$b \leq x < b + l$$
    or
    $$0 \leq x < l,$$
    depending on the scheme employed, where b and l are the base and limit registers that capture the region allocated to the (active) process performing said access. It *is* possible for regions to overlap with each other, however: we just need a case where
    $$b_0 < b_1 < b_0 + l_0,$$
    for example. However, such cases would not often produce a useful outcome. More specifically, note the granularity of regions is an entire process so the resulting overlap will only ever occur at the top- or bottom-end of the resulting address space: assuming use of a standard layout with stack and text segments located at the top- and bottom-end, such an overlap would be little use because their being shared makes little sense.
  – Since the address space for each process is captured by one region only, enlarging the address space boils down to one of two cases: we enlarge a) upward by increasing the limit, or b) downward by decreasing the base (*and* the limit: otherwise we just move the segment downward). Although in theory this amounts to simply manipulating the base and limit registers, various challenges can arise. As an example, consider enlarging upward: what happens if the region associated with *another* process is just above? Simply increasing the limit would cause unintended overlap; to legitimately do so, we need to relocate that process to make room. This is possible of course, but assumes there is a gap to relocate it into (cf. fragmentation) and that it is worth the associated overhead (the entire region will need to be copied).
  – Clearly *all* per process state maintained by the kernel, namely the base and limit registers, should *not* be accessible in user mode: if it were, this would void any protection offered (since a user mode process could simply rewrite the state). Likewise, this same state needs to be swapped to/from PCBs of the descheduled/scheduled process during a context switch.

Mechanism: hardware-based per process segmentation (2)

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

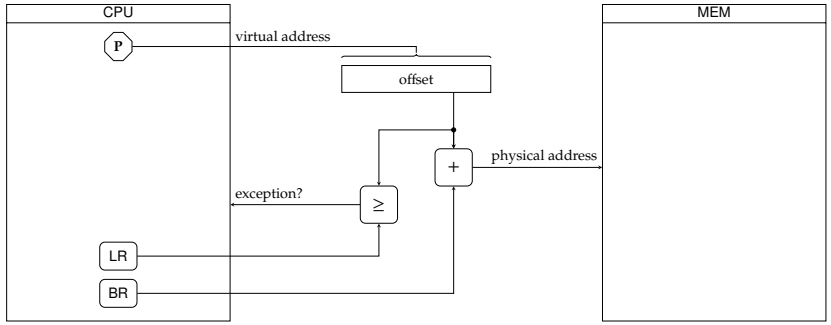


Notes:

- This is the first time the term offset is used, but here, and here after, the same concept might be described as a displacement: what it represents, either way, is a value added to the base address to yield the target address.

Mechanism: hardware-based per process segmentation (2)

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



Notes:

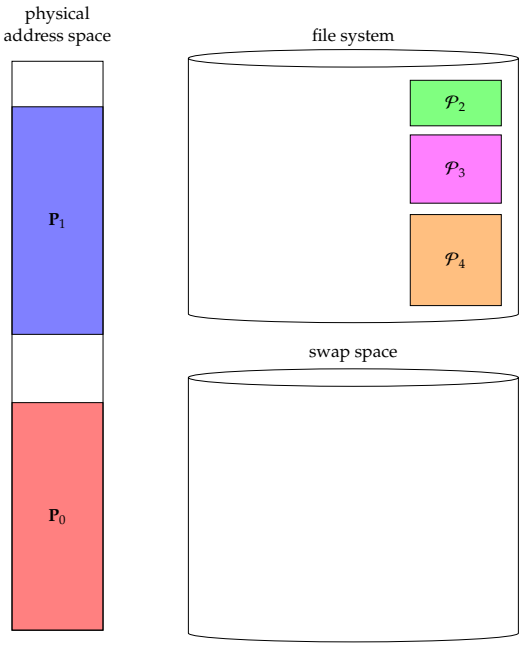
- This is the first time the term offset is used, but here, and here after, the same concept might be described as a displacement: what it represents, either way, is a value added to the base address to yield the target address.

© Daniel Page ⟨[redacted]⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

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.



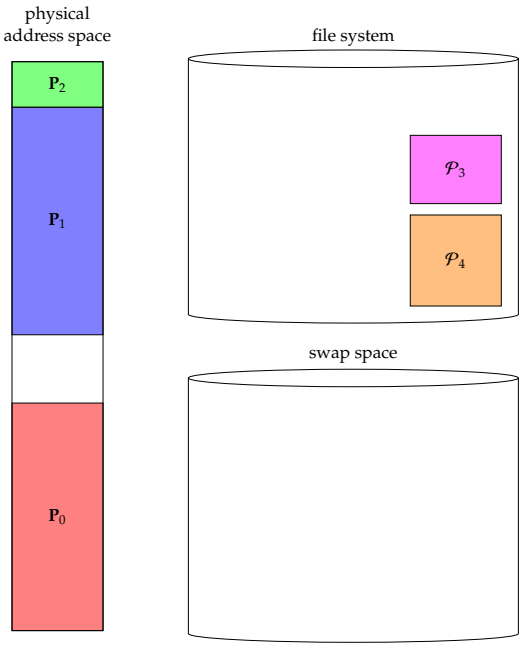
Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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.



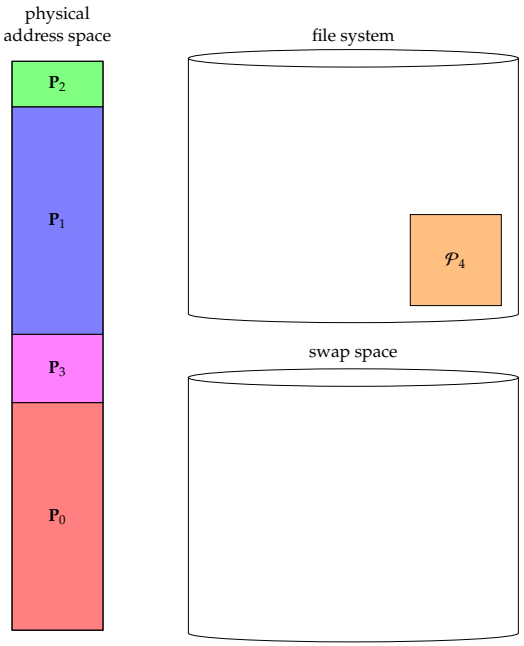
Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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.



Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

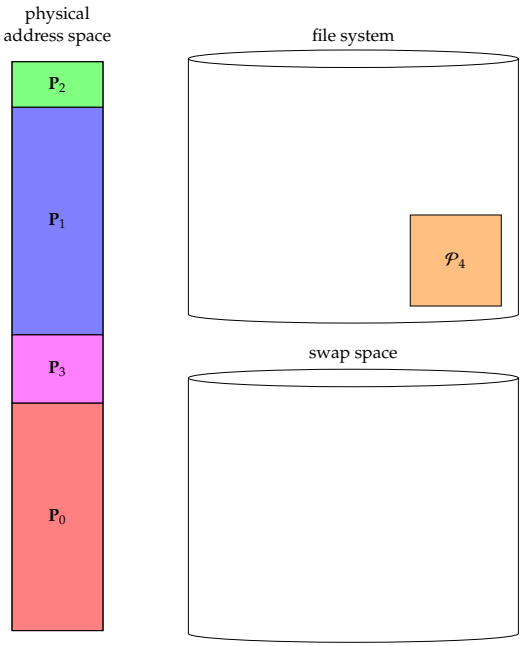
An Aside: allocation, swapping and fragmentation

- 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.

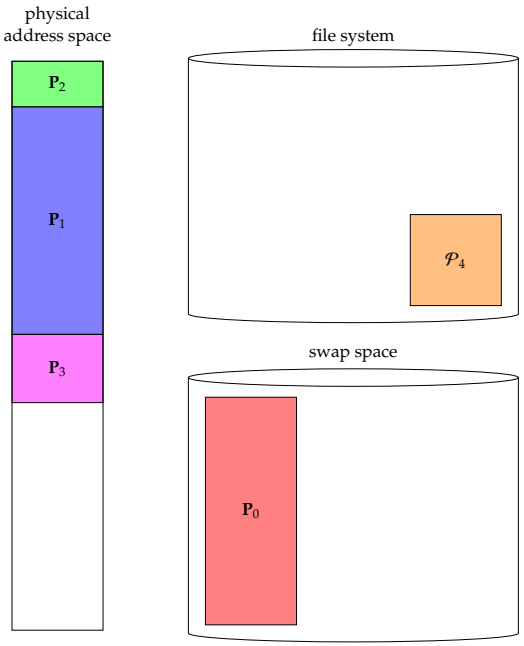


Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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.

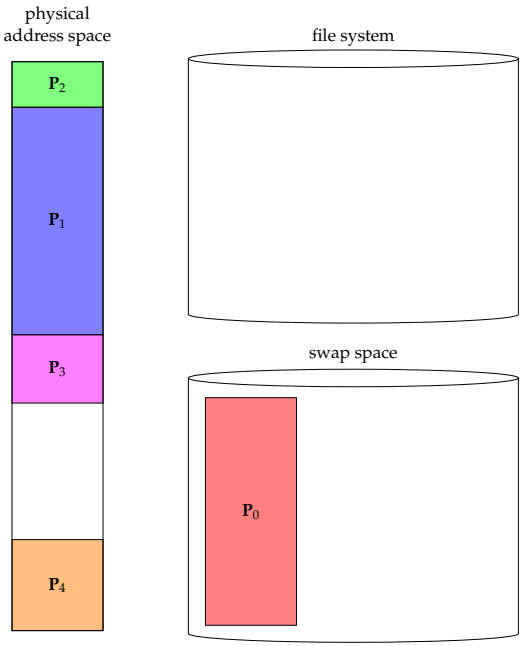


Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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.

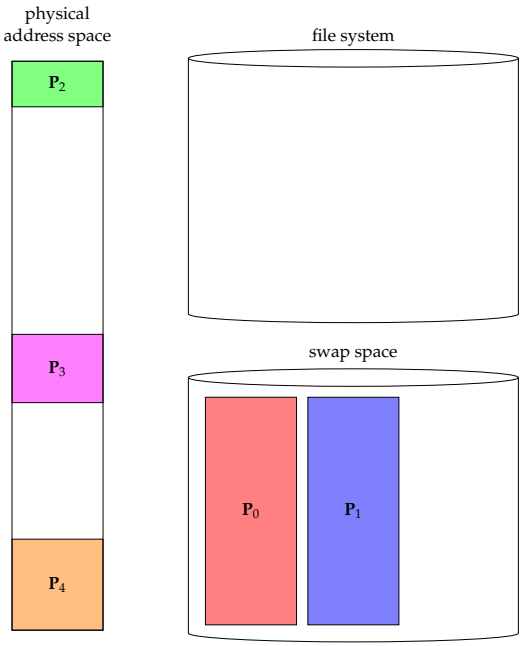


Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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.

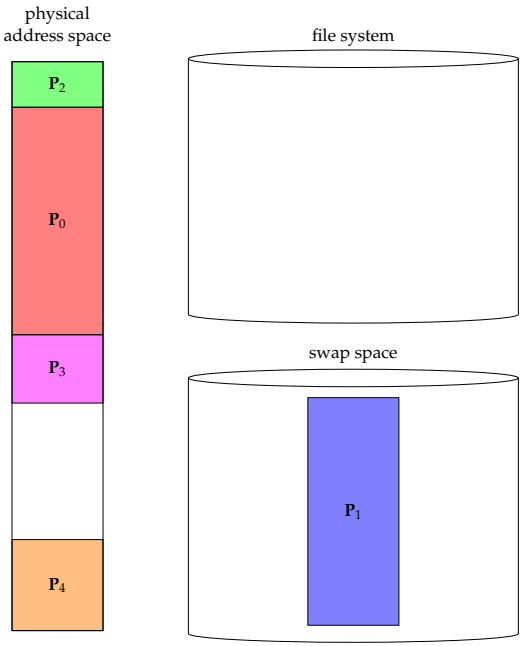


Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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.



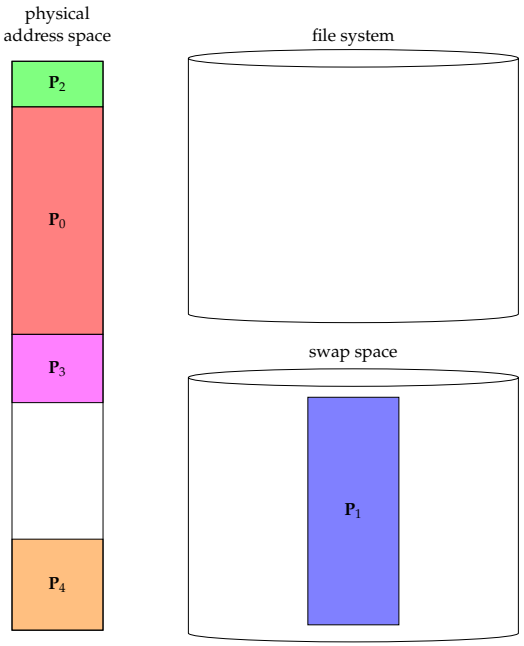
© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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.



Notes:

- The terms swapping-in and swapping-out are respectively used to describe storing content into or retrieving content from the swap space.
  The diagram here depicts the swap space as a separate disk; in reality, it could equally be a separate partition on the same disk as the file system. Here we assume (following normal terminology) the granularity swap-in and swap-out operations is the *entire* address space of a process. This is not a restriction per se, as we will see later, but, either way, the kernel needs to maintain a data structure that captures the mapping of swapped-out content to an address (i.e., location) on the disk.
- The requirement for an allocation algorithm (or strategy) and associated data structure is a fairly general one: *anywhere* we need to make a decision about where a contiguous region is placed, similar requirements will likely exist. We will see other granularities of region, but they have similar requirements as here (where the granularity is an entire process).
- [5, Sections 8.3.2 and 8.8.3] or [9, Sections 3.2.2 and 3.2.3] are good starting points to find out detail about allocation algorithms, or data structures, or the issue of fragmentation.
- Using memory compaction (or defragmentation) as a way to reduce external fragmentation is unattractive, since the overhead is large: relocation of a region implies a need to copy it from one place to another, so if that region is large (which is possible, or even likely: currently we have regions that represent an entire process) this will take a tangible amount of time.
  Added to this, and in a similar way to defragmentation in other contexts (in file systems, for example) the need to shuffle regions around rather than simply copy them into place can become problematic. For example, imagine we need to copy fragments of one region into a contiguous region at some address: if that region is allocated (i.e., in use), we need to temporarily copy it into *another* region first so defragmentation will not simply overwrite it. This is possible of course, but if there is limited space in physical memory, finding a region for the temporary copy can be challenging.

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

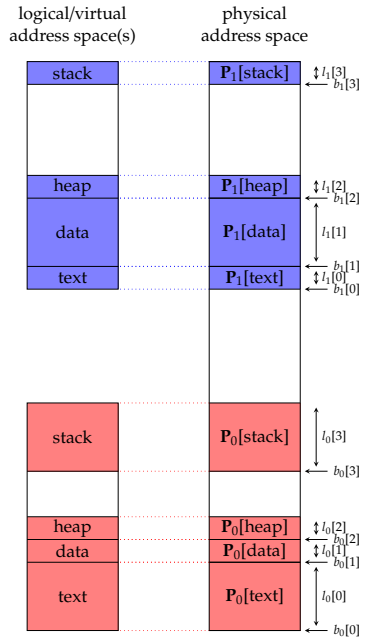
$$0 \leq \text{LSB}_{w-t}(x) < l[\text{MSB}_t(x)],$$

  and translate st.

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

- 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	✓



Notes:

- – Now, the address space of each process will be described by multiple regions: iff. one region is used for each segment the terms can be used anonymously, but this is often more a policy implemented by a potentially more general mechanism. x86 has an "extra" segmentation register `es`, for example, which can be used as required by the programmer alongside other registers for standard segments.
  Either way, per segment segmentation is somewhat similar to per process segmentation wrt. address space contiguousness. That is, the kernel must perform inter- but not intra-region allocation. Doing so is simultaneously easier and harder, in the sense that regions are now finer grained so smaller (meaning more flexibility: a smaller segment will typically fit into more gaps than larger ones) but there are also more of them.
  As an aside, shifting from a single (contiguous) to multiple (non-contiguous) regions can be viewed as transforming the associated address space from a 1D region into a 2D region: the former needs one coordinate (i.e., address) whereas the latter needs two (i.e., the segment identifier and address).
  – The now multiple regions which capture the address space of a process offer protection for the same reason as a single region in per process segmentation. Additionally, sharing regions of physical memory is more useful due to the finer grained control possible. For example, we could decide to overlap and hence share the text segments (e.g., in order to support execution of identical programs).
  – The challenges of region growth in per segment segmentation are similar to per process segmentation. However, resolving them is arguably easier because only some regions are likely to be enlarged. For example, the stack and heap segments are relatively more likely to be enlarged than the text segment.
  – Clearly *all* per process state maintained by the kernel, namely the segment table should *not* be accessible in user mode: if it were, this would void any protection offered (since a user mode process could simply rewrite the state). Likewise, this same state needs to be swapped to/from PCBs of the descheduled/scheduled process during a context switch.
- Typically we assume there are (relatively) few segments, meaning the segment table is (relatively) small. Rather than a table per se, you *could* think of it as a collection of registers.

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

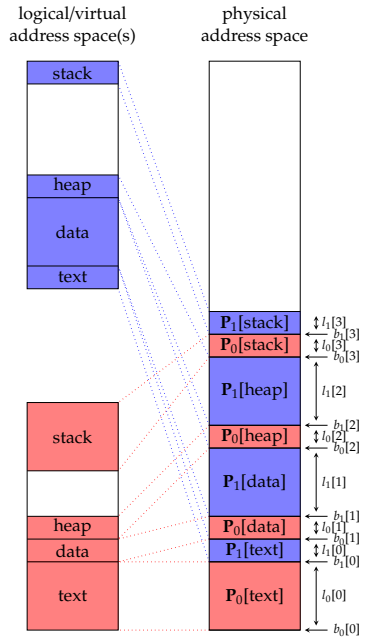
$$0 \leq \text{LSB}_{w-t}(x) < l[\text{MSB}_t(x)],$$

  and translate st.

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

- 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	✓

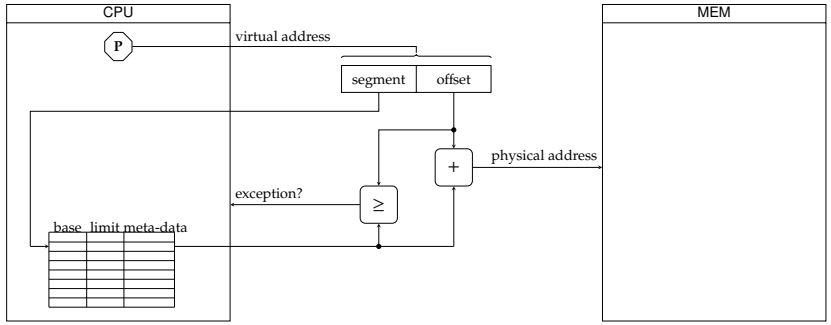


Notes:

- – Now, the address space of each process will be described by multiple regions: iff. one region is used for each segment the terms can be used anonymously, but this is often more a policy implemented by a potentially more general mechanism. x86 has an "extra" segmentation register `es`, for example, which can be used as required by the programmer alongside other registers for standard segments.
  Either way, per segment segmentation is somewhat similar to per process segmentation wrt. address space contiguousness. That is, the kernel must perform inter- but not intra-region allocation. Doing so is simultaneously easier and harder, in the sense that regions are now finer grained so smaller (meaning more flexibility: a smaller segment will typically fit into more gaps than larger ones) but there are also more of them.
  As an aside, shifting from a single (contiguous) to multiple (non-contiguous) regions can be viewed as transforming the associated address space from a 1D region into a 2D region: the former needs one coordinate (i.e., address) whereas the latter needs two (i.e., the segment identifier and address).
  – The now multiple regions which capture the address space of a process offer protection for the same reason as a single region in per process segmentation. Additionally, sharing regions of physical memory is more useful due to the finer grained control possible. For example, we could decide to overlap and hence share the text segments (e.g., in order to support execution of identical programs).
  – The challenges of region growth in per segment segmentation are similar to per process segmentation. However, resolving them is arguably easier because only some regions are likely to be enlarged. For example, the stack and heap segments are relatively more likely to be enlarged than the text segment.
  – Clearly *all* per process state maintained by the kernel, namely the segment table should *not* be accessible in user mode: if it were, this would void any protection offered (since a user mode process could simply rewrite the state). Likewise, this same state needs to be swapped to/from PCBs of the descheduled/scheduled process during a context switch.
- Typically we assume there are (relatively) few segments, meaning the segment table is (relatively) small. Rather than a table per se, you *could* think of it as a collection of registers.

Mechanism: hardware-based per segment segmentation (2)

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



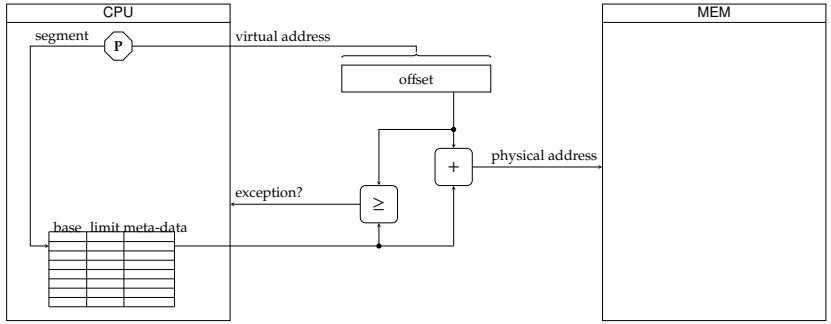
noting we *could* opt to index into the table via

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

Notes:

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.

Notes:

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**

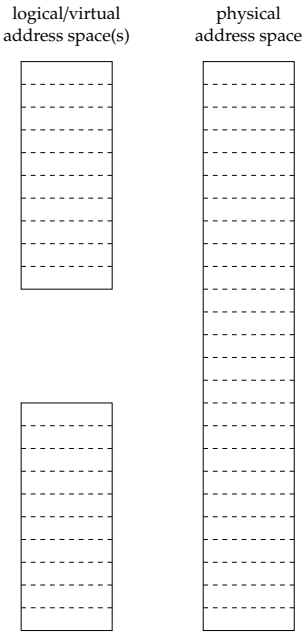
    of l bytes in each case,
  - maintain a **page table** T per process,
  - let $t = \log_2(|T|)$, and translate st.

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

  noting no check is required since

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

  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	✓

Notes:

- – Introduction of paging adds further flexibility, basically by further decreasing the granularity at which regions can be described. Previously, each region would describe the address space for a process or segments of it; now a region is a page, far smaller than either.
  Although each page is always contiguous, this is not true in either intra- or inter-region cases. For example, the text segment for a process could be comprised of non-contiguous pages, and, itself, be non-contiguous wrt. pages associated with the stack segment.
  – Maintaining the page table requires some effort and care, but also provides an opportunity: one can easily associate other attributes, or meta-data, with each page. Examples include
    - a valid bit(s) to support a sparse address space by allowing a page to be unmapped, meaning there is no associated page frame for a given page,
    - a dirty bit(s) to mark pages that be altered so require writing to the swap space when swapped-out,
    - access control bit(s) to support various forms of protection or sharing.

    The (relatively) fine grained nature of pages compared to per process or segment segment gives even more flexibility about how this meta-data can be used (e.g., to implement protection and sharing policies).
  – A segment previously had to be contiguous, which was the crux of various challenges. Now a segment can span multiple pages, which can be non-contiguous and even unallocated in physical memory; this means enlarging a segment simply means mapping pages to unused page frames.
  – By using paging, the virtual and physical address spaces are totally decoupled wrt. a) the mapping of one to the other (e.g., supporting a uniform address space ranging from address 0 to $2^w - 1$ for each of n processes), and b) their capacity (e.g., one could, and it is common to have less physical memory than required to support one full virtual address space, let alone n). This suggests virtualisation of the underlying, physical memory in the sense paging makes it *seem* as if said memory has the idealised properties required (even when it does not).
  – Clearly *all* per process state maintained by the kernel, namely the page table, should *not* be accessible in user mode: if it were, this would void any protection offered (since a user mode process could simply rewrite the state). Likewise, this same state needs to be swapped-to/from PCBs of the descheduled/scheduled process during a context switch.
- Strictly speaking there is no need for all pages (resp. page frames) to have an identical size, although we *do* need need each page to map to a page frame whose size matches. Note that when the pages are the same size, external fragmentation is a non-issue since any page can fit exactly into any page frame.

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.

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

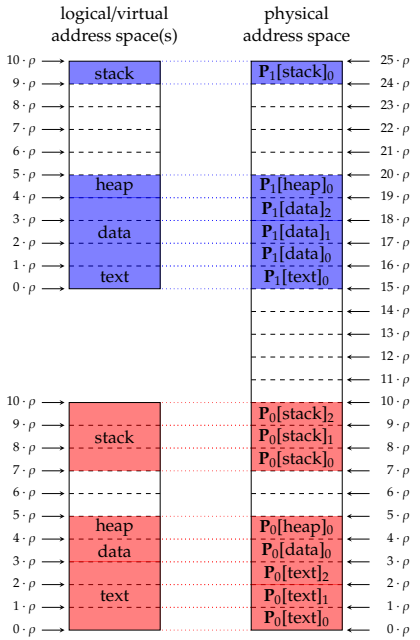
noting no check is required since

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

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	✓



Notes:

- – Introduction of paging adds further flexibility, basically by further decreasing the granularity at which regions can be described. Previously, each region would describe the address space for a process or segments of it; now a region is a page, far smaller than either.
  Although each page is always contiguous, this is not true in either intra- or inter-region cases. For example, the text segment for a process could be comprised of non-contiguous pages, and, itself, be non-contiguous wrt. pages associated with the stack segment.
  – Maintaining the page table requires some effort and care, but also provides an opportunity: one can easily associate other attributes, or meta-data, with each page. Examples include
    - a valid bit(s) to support a sparse address space by allowing a page to be unmapped, meaning there is no associated page frame for a given page,
    - a dirty bit(s) to mark pages that be altered so require writing to the swap space when swapped-out,
    - access control bit(s) to support various forms of protection or sharing.

    The (relatively) fine grained nature of pages compared to per process or segment segment gives even more flexibility about how this meta-data can be used (e.g., to implement protection and sharing policies).
  – A segment previously had to be contiguous, which was the crux of various challenges. Now a segment can span multiple pages, which can be non-contiguous and even unallocated in physical memory; this means enlarging a segment simply means mapping pages to unused page frames.
  – By using paging, the virtual and physical address spaces are totally decoupled wrt. a) the mapping of one to the other (e.g., supporting a uniform address space ranging from address 0 to $2^w - 1$ for each of n processes), and b) their capacity (e.g., one could, and it is common to have less physical memory than required to support one full virtual address space, let alone n). This suggests virtualisation of the underlying, physical memory in the sense paging makes it *seem* as if said memory has the idealised properties required (even when it does not).
  – Clearly *all* per process state maintained by the kernel, namely the page table, should *not* be accessible in user mode: if it were, this would void any protection offered (since a user mode process could simply rewrite the state). Likewise, this same state needs to be swapped-to/from PCBs of the descheduled/scheduled process during a context switch.
- Strictly speaking there is no need for all pages (resp. page frames) to have an identical size, although we *do* need need each page to map to a page frame whose size matches. Note that when the pages are the same size, external fragmentation is a non-issue since any page can fit exactly into any page frame.

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.

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

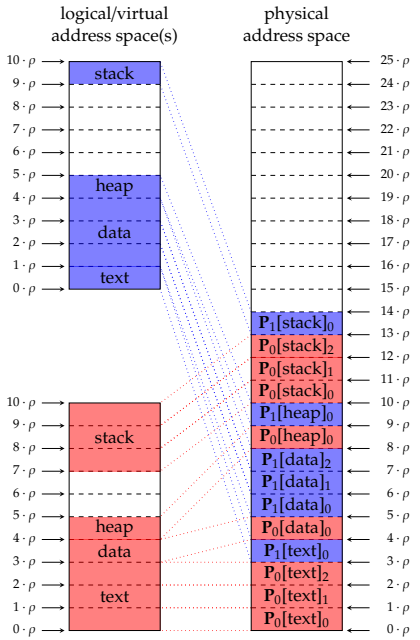
noting no check is required since

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

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	✓



Notes:

- – Introduction of paging adds further flexibility, basically by further decreasing the granularity at which regions can be described. Previously, each region would describe the address space for a process or segments of it; now a region is a page, far smaller than either.
  Although each page is always contiguous, this is not true in either intra- or inter-region cases. For example, the text segment for a process could be comprised of non-contiguous pages, and, itself, be non-contiguous wrt. pages associated with the stack segment.
  – Maintaining the page table requires some effort and care, but also provides an opportunity: one can easily associate other attributes, or meta-data, with each page. Examples include
    - a valid bit(s) to support a sparse address space by allowing a page to be unmapped, meaning there is no associated page frame for a given page,
    - a dirty bit(s) to mark pages that be altered so require writing to the swap space when swapped-out,
    - access control bit(s) to support various forms of protection or sharing.

    The (relatively) fine grained nature of pages compared to per process or segment segment gives even more flexibility about how this meta-data can be used (e.g., to implement protection and sharing policies).
  – A segment previously had to be contiguous, which was the crux of various challenges. Now a segment can span multiple pages, which can be non-contiguous and even unallocated in physical memory; this means enlarging a segment simply means mapping pages to unused page frames.
  – By using paging, the virtual and physical address spaces are totally decoupled wrt. a) the mapping of one to the other (e.g., supporting a uniform address space ranging from address 0 to $2^w - 1$ for each of n processes), and b) their capacity (e.g., one could, and it is common to have less physical memory than required to support one full virtual address space, let alone n). This suggests virtualisation of the underlying, physical memory in the sense paging makes it *seem* as if said memory has the idealised properties required (even when it does not).
  – Clearly *all* per process state maintained by the kernel, namely the page table, should *not* be accessible in user mode: if it were, this would void any protection offered (since a user mode process could simply rewrite the state). Likewise, this same state needs to be swapped-to/from PCBs of the descheduled/scheduled process during a context switch.
- Strictly speaking there is no need for all pages (resp. page frames) to have an identical size, although we *do* need need each page to map to a page frame whose size matches. Note that when the pages are the same size, external fragmentation is a non-issue since any page can fit exactly into any page frame.

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.

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

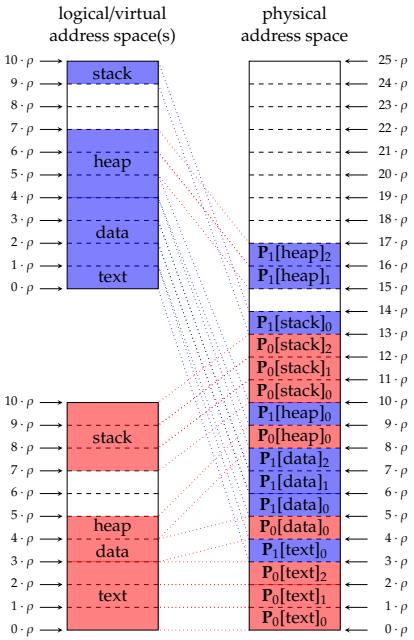
    noting no check is required since

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

    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	✓



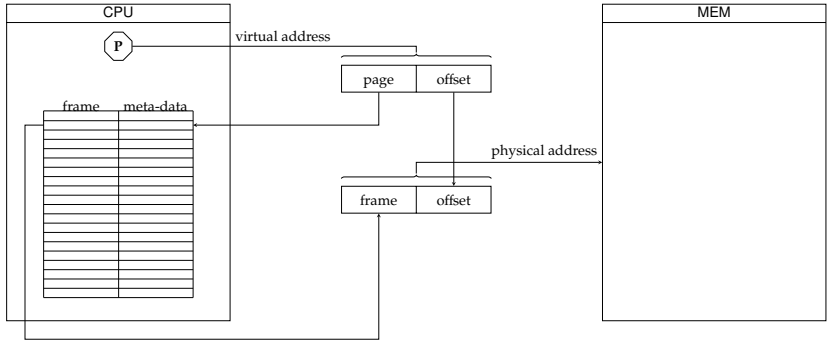
Notes:

- – Introduction of paging adds further flexibility, basically by further decreasing the granularity at which regions can be described. Previously, each region would describe the address space for a process or segments of it; now a region is a page, far smaller than either.
  Although each page is always contiguous, this is not true in either intra- or inter-region cases. For example, the text segment for a process could be comprised of non-contiguous pages, and, itself, be non-contiguous wrt. pages associated with the stack segment.
  – Maintaining the page table requires some effort and care, but also provides an opportunity: one can easily associate other attributes, or meta-data, with each page. Examples include
    - a valid bit(s) to support a sparse address space by allowing a page to be unmapped, meaning there is no associated page frame for a given page,
    - a dirty bit(s) to mark pages that be altered so require writing to the swap space when swapped-out,
    - access control bit(s) to support various forms of protection or sharing.

    The (relatively) fine grained nature of pages compared to per process or segment segment gives even more flexibility about how this meta-data can be used (e.g., to implement protection and sharing policies).
  – A segment previously had to be contiguous, which was the crux of various challenges. Now a segment can span multiple pages, which can be non-contiguous and even unallocated in physical memory; this means enlarging a segment simply means mapping pages to unused page frames.
  – By using paging, the virtual and physical address spaces are totally decoupled wrt. a) the mapping of one to the other (e.g., supporting a uniform address space ranging from address 0 to $2^w - 1$ for each of n processes), and b) their capacity (e.g., one could, and it is common to have less physical memory than required to support one full virtual address space, let alone n). This suggests virtualisation of the underlying, physical memory in the sense paging makes it *seem* as if said memory has the idealised properties required (even when it does not).
  – Clearly *all* per process state maintained by the kernel, namely the page table, should *not* be accessible in user mode: if it were, this would void any protection offered (since a user mode process could simply rewrite the state). Likewise, this same state needs to be swapped-to/from PCBs of the descheduled/scheduled process during a context switch.
- Strictly speaking there is no need for all pages (resp. page frames) to have an identical size, although we *do* need need each page to map to a page frame whose size matches. Note that when the pages are the same size, external fragmentation is a non-issue since any page can fit exactly into any page frame.

Mechanism: hardware-based paging (2)

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



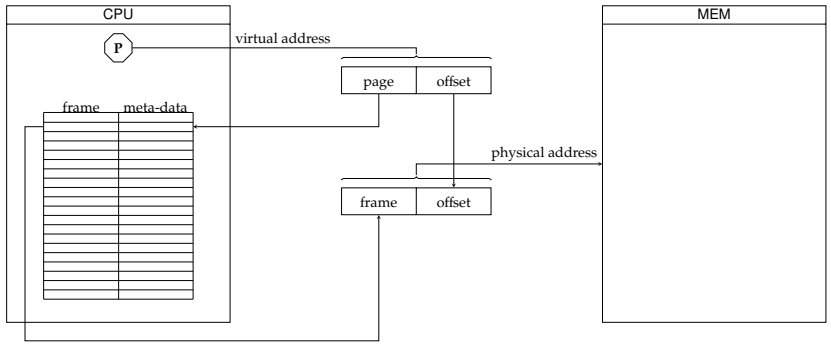
noting the page table consists of **Page Table Entries (PTEs)**.

Notes:

- This diagram mirrors historical architectures such as the PDP-11: it had 16-bit addresses, and a 64KiB virtual address space with 8 pages each of 8KiB. With this few entries, the page table could be kept on-chip by using a small set of registers (in the same way as the segment table used in the previous approach).
  However, modern architectures allow much larger virtual and physical address spaces, and require much larger page tables. For example, with 32-bit virtual addresses we can address 4GiB of virtual memory; a page size of say ~ 4KiB will therefore mean upto 4GiB/ ~ 4KiB = 1048576 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,

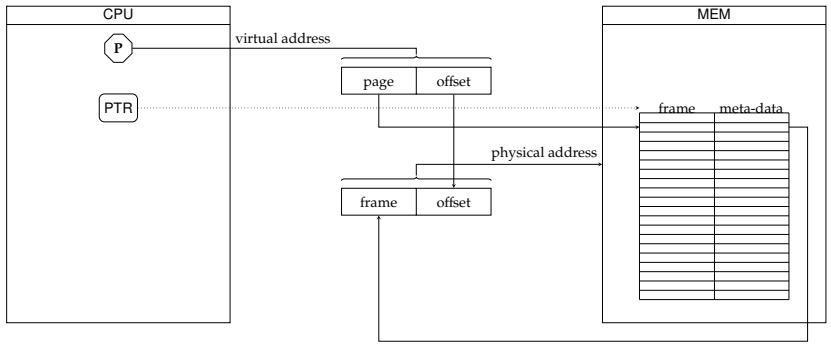
   st. cached PTEs for one process cannot be used by another.

Notes:

- An important thing to keep in mind is that the TLB caches PTEs, *not* the pages themselves: is simply accelerates the task of address translation.

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.

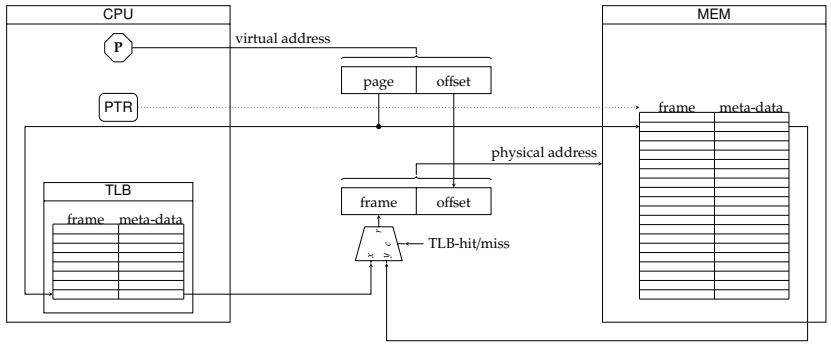
© Daniel Page (<redacted>)
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- An important thing to keep in mind is that the TLB caches PTEs, *not* the pages themselves: is simply accelerates the task of address translation.

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,

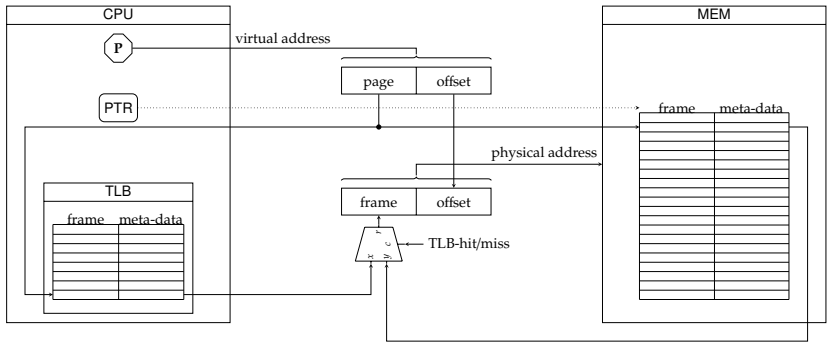
   st. cached PTEs for one process cannot be used by another.

Notes:

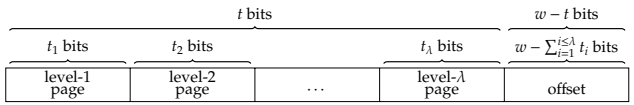
- An important thing to keep in mind is that the TLB caches PTEs, *not* the pages themselves: is simply accelerates the task of address translation.

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.,



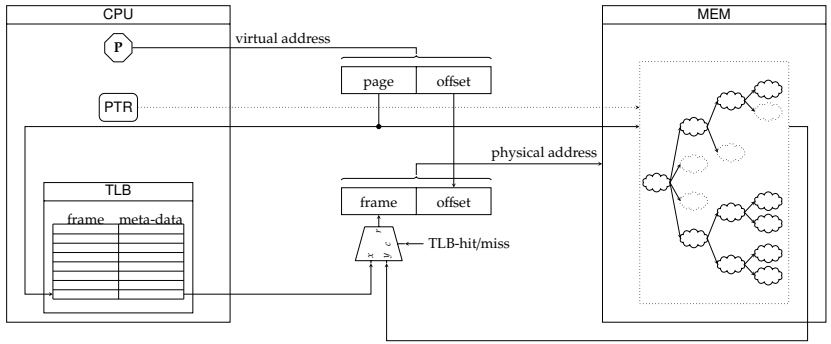
3. use a valid flag to indicate whether or not a sub-tree exists.

Notes:

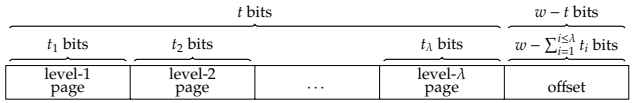
- The way this is realised is for the page number to be split into more than one component: each i-th component is used as an index into the level-i page table.
- One disadvantage is that to get from the initial PTR to the actual page we want, assuming a TLB miss we need to walk the page table; this places even higher value on the TLB, because *many* accesses (vs. two, which is already bad enough) could be required for a multi-level page table. Note that it is possible to walk the page table in hardware *or* software, but the associated cost is st. software-controlled walks are less common.

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.

Notes:

- The way this is realised is for the page number to be split into more than one component: each i-th component is used as an index into the level-i page table.
- One disadvantage is that to get from the initial PTR to the actual page we want, assuming a TLB miss we need to walk the page table; this places even higher value on the TLB, because *many* accesses (vs. two, which is already bad enough) could be required for a multi-level page table. Note that it is possible to walk the page table in hardware *or* software, but the associated cost is st. software-controlled walks are less common.

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

Notes:

- There is a rich literature exploring the design trade-offs for paged memory implementation; see, for example, [10] wrt. the TLB.
- The terms soft and hard are often used interchangeable with minor and major when used to describe these exceptions.
- In [11, Section B3.13], ARM outlines a more specific set of exceptions that can be communicated by the MMU via a dedicated Fault Status Register (FSR); these decompose the case labelled very generically here as "access fault" using more specific terms.
- The concept of a soft page fault may seem odd. Such an exception can occur, for example, if the page is being shared between two processes: then it is reasonable for it to be in physical memory, but only in the page table for one process st. access by the other causes a fault.
- A subtle but important thing to keep in mind is that handling a fault is often a sort of 2-step process: first the kernel has to perform any actions required for the fault type (e.g., allocate a previously unused page frame and map some previously invalid page to it), plus then restart the instruction which caused the fault (to preserve the impression of memory being virtualised: if the process "knew" the fault had occurred, the virtualisation would be weak).

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.

Notes:

- There is a rich literature exploring the design trade-offs for paged memory implementation; see, for example, [10] wrt. the TLB.
- The terms soft and hard are often used interchangeable with minor and major when used to describe these exceptions.
- In [11, Section B3.13], ARM outlines a more specific set of exceptions that can be communicated by the MMU via a dedicated Fault Status Register (FSR); these decompose the case labelled very generically here as "access fault" using more specific terms.
- The concept of a soft page fault may seem odd. Such an exception can occur, for example, if the page is being shared between two processes: then it is reasonable for it to be in physical memory, but only in the page table for one process st. access by the other causes a fault.
- A subtle but important thing to keep in mind is that handling a fault is often a sort of 2-step process: first the kernel has to perform any actions required for the fault type (e.g., allocate a previously unused page frame and map some previously invalid page to it), plus then restart the instruction which caused the fault (to preserve the impression of memory being virtualised: if the process "knew" the fault had occurred, the virtualisation would be weak).

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

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

Notes:

- There is a rich literature exploring the design trade-offs for paged memory implementation; see, for example, [10] wrt. the TLB.
- The terms soft and hard are often used interchangeable with minor and major when used to describe these exceptions.
- In [11, Section B3.13], ARM outlines a more specific set of exceptions that can be communicated by the MMU via a dedicated Fault Status Register (FSR); these decompose the case labelled very generically here as "access fault" using more specific terms.
- The concept of a soft page fault may seem odd. Such an exception can occur, for example, if the page is being shared between two processes: then it is reasonable for it to be in physical memory, but only in the page table for one process st. access by the other causes a fault.
- A subtle but important thing to keep in mind is that handling a fault is often a sort of 2-step process: first the kernel has to perform any actions required for the fault type (e.g., allocate a previously unused page frame and map some previously invalid page to it), plus then restart the instruction which caused the fault (to preserve the impression of memory being virtualised: if the process "knew" the fault had occurred, the virtualisation would be weak).

Implementation: ARMv7-A (1)

- ARMv7-A supports *two* (very flexible) mechanisms via
  1. the **Protected Memory System Architecture (PMSA)** [11, Chapter B5] and
  2. the **Virtual Memory System Architecture (VMSA)** [11, Chapter B3]

  both of which are controlled via the co-processor interface [11, Chapters B4 and B6].

Notes:

- PMSA requires simpler MPU-style hardware but offers protection *only*, whereas VMSA requires more complex MMU-style hardware but offers protection *plus* translation. It is important to note that for embedded processors, PMSA might actually be enough: not *every* context demands fully virtualised memory.

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 [11, Section B3.3], namely

$$\text{long} \Rightarrow \text{64-bit PTE} \begin{cases} \text{upto } \lambda = 3 \text{ levels} \\ \text{translates 32-bit to 40-bit address spaces at 4KiB granularity} \end{cases}$$

$$\text{short} \Rightarrow \text{32-bit PTE} \begin{cases} \text{upto } \lambda = 2 \text{ levels} \\ \text{translates 32-bit to 32-bit address spaces at 4KiB granularity} \end{cases}$$

$$\text{short} \Rightarrow \text{32-bit PTE} \begin{cases} \text{upto } \lambda = 2 \text{ levels} \\ \text{translates 32-bit to 40-bit address spaces at 16MiB granularity} \end{cases}$$

  plus per-level variants of each.

Notes:

- A 40-bit physical address space implies 2^{40}B = 1TiB of physical memory. Which is a *lot*!
- One rationale for the inclusion of *two* PTRs is so that one can be dedicated to the kernel (i.e., never changed), while the other is used for the active (user mode) address space.
- As a result of the terms used for what are essentially just different page sizes, the ARM documentation uses the term translation table to describe (e.g., [11, Section B3.3]) what we term a page table: they are the same thing, but ARM (presumably) uses the more general term to stress the more general parameterisations possible in VMSA.
- Although ARMv7-A specifies some [11, Section B3.9] architectural requirements of the TLBs (e.g., what operations should be allowed) the micro-architectural implementation can differ, e.g.,
  1. Cortex-A8 [12, Section 6.1] has
     - a level-1 instruction TLB with 32 fully-associative, lockable entries,
     - a level-1 data TLB with 32 fully-associative, lockable entries
  2. Cortex-A9 [13, Section 6.2] has
     - a level-1 (or micro) instruction TLB with 32 fully-associative, lockable entries,
     - a level-1 (or micro) data TLB with 32 fully-associative, lockable entries,
     - a level-2 (or main) unified TLB with 64/8 low/fully-associative, non-/lockable entries

  which support the **Address Space IDentifier (ASID)** to avoid a TLB flush during a context switch.

- Some details:

3. It supports four page sizes [11, 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.

Notes:

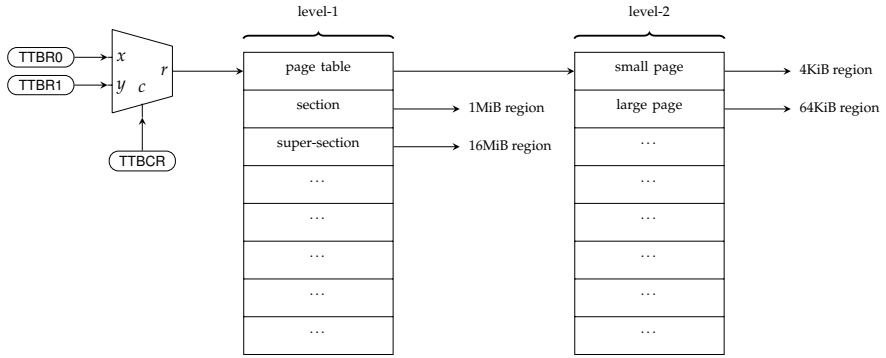
- A 40-bit physical address space implies 2^{40}B = 1TiB of physical memory. Which is a *lot*!
- One rationale for the inclusion of *two* PTRs is so that one can be dedicated to the kernel (i.e., never changed), while the other is used for the active (user mode) address space.
- As a result of the terms used for what are essentially just different page sizes, the ARM documentation uses the term translation table to describe (e.g., [11, Section B3.3]) what we term a page table: they are the same thing, but ARM (presumably) uses the more general term to stress the more general parameterisations possible in VMSA.
- Although ARMv7-A specifies some [11, Section B3.9] architectural requirements of the TLBs (e.g., what operations should be allowed) the micro-architectural implementation can differ, e.g.,
  1. Cortex-A8 [12, Section 6.1] has
     - a level-1 instruction TLB with 32 fully-associative, lockable entries,
     - a level-1 data TLB with 32 fully-associative, lockable entries
  2. Cortex-A9 [13, Section 6.2] has
     - a level-1 (or micro) instruction TLB with 32 fully-associative, lockable entries,
     - a level-1 (or micro) data TLB with 32 fully-associative, lockable entries,
     - a level-2 (or main) unified TLB with 64/8 low/fully-associative, non-/lockable entries

  which support the **Address Space IDentifier (ASID)** to avoid a TLB flush during a context switch.

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.

Notes:

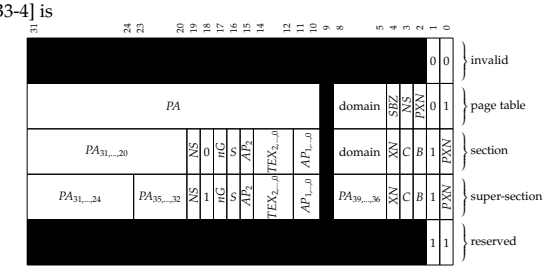
- The net result of this page table organisation is that when larger page sizes (i.e., sections) are used,
  - there's no need to walk through both levels of the hierarchy to get to a page, and
  - large regions captured by such a page will occupy only one TLB entry.
- It might not be obvious, but, to be clear, each process will have one level-1 page table and a set of level-2 page tables (that depend on the validity of entries in the level-1 page table).
- [11, Section B3.5.2] offers a definitive guide to all the acronym'ised PTE field names, but a (very) brief overview is as follows:
  - *SBZ* stands for Should Be Zero, i.e., equal to 0,
  - *PA* is basically a pointer to something,
  - *XN* is the execute never bit,
  - *PXN* is the privileged execute never bit,
  - *NS* is the non-secure bit,
  - *nG* is the not-global bit,
  - *S* is the sharable bit,
  - *TEX*, *C* and *B* represent attribute bits for a region; they can be used, for example, to specify whether it can be cached or not,
  - *AP* is a set of access permission bits.
- Clearly the input of translation is a Virtual Address (VA) and the output a Physical Address (PA); after the intermediate step when a walk involves both the level-1 and level-2 page tables, there is what is termed an Intermediate Physical Address (IPA).

Example

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

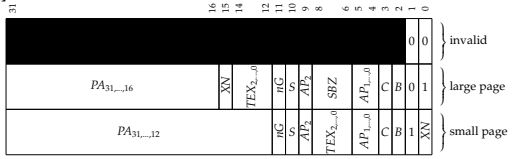
The format of (short) PTEs

- at level-1 [11, Figure B3-4] is



and

- at level-2 [11, Figure B3-5] is



Notes:

- The net result of this page table organisation is that when larger page sizes (i.e., sections) are used,
  - there's no need to walk through both levels of the hierarchy to get to a page, and
  - large regions captured by such a page will occupy only one TLB entry.
- It might not be obvious, but, to be clear, each process will have one level-1 page table and a set of level-2 page tables (that depend on the validity of entries in the level-1 page table).
- [11, Section B3.5.2] offers a definitive guide to all the acronym'ised PTE field names, but a (very) brief overview is as follows:
  - *SBZ* stands for Should Be Zero, i.e., equal to 0,
  - *PA* is basically a pointer to something,
  - *XN* is the execute never bit,
  - *PXN* is the privileged execute never bit,
  - *NS* is the non-secure bit,
  - *nG* is the not-global bit,
  - *S* is the sharable bit,
  - *TEX*, *C* and *B* represent attribute bits for a region; they can be used, for example, to specify whether it can be cached or not,
  - *AP* is a set of access permission bits.
- Clearly the input of translation is a Virtual Address (VA) and the output a Physical Address (PA); after the intermediate step when a walk involves both the level-1 and level-2 page tables, there is what is termed an Intermediate Physical Address (IPA).

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:

1 **if** *PTE E for x is resident in the appropriate TLB(s)* **then**
2 | **if** *access control check for x and E passes* **then**
3 | | load from MEM[$E[PA] + x_{11,...,0}$]
4 | **else**
5 | | request exception
6 | **end**
7 **else**
8 | **if** $\text{MSB}_n(x) = 0$ **then**
9 | | load level-1 entry E_1 from MEM[TTBR0 + $x_{31,...,20}$]
10 | **else**
11 | | load level-1 entry E_1 from MEM[TTBR1 + $x_{31,...,20}$]
12 | **end**
13 | **if** E_1 *is invalid or access control check fails* **then** request exception
14 | load level-2 entry E_2 from MEM[$E_1[PA] + x_{19,...,12}$]
15 | **if** E_2 *is invalid or access control check fails* **then** request exception
16 | load from MEM[$E_2[PA] + x_{11,...,0}$]
17 | update TLB(s)
18 **end**

Notes:

- The net result of this page table organisation is that when larger page sizes (i.e., sections) are used,
  - there's no need to walk through both levels of the hierarchy to get to a page, and
  - large regions captured by such a page will occupy only one TLB entry.
- It might not be obvious, but, to be clear, each process will have one level-1 page table and a set of level-2 page tables (that depend on the validity of entries in the level-1 page table).
- [11, Section B3.5.2] offers a definitive guide to all the acronym'ised PTE field names, but a (very) brief overview is as follows:
  - *SBZ* stands for Should Be Zero, i.e., equal to 0,
  - *PA* is basically a pointer to something,
  - *XN* is the execute never bit,
  - *PXN* is the privileged execute never bit,
  - *NS* is the non-secure bit,
  - *nG* is the not-global bit,
  - *S* is the sharable bit,
  - *TEX*, *C* and *B* represent attribute bits for a region; they can be used, for example, to specify whether it can be cached or not,
  - *AP* is a set of access permission bits.
- Clearly the input of translation is a Virtual Address (VA) and the output a Physical Address (PA); after the intermediate step when a walk involves both the level-1 and level-2 page tables, there is what is termed an Intermediate Physical Address (IPA).

An Aside: a rough summary and/or interlude

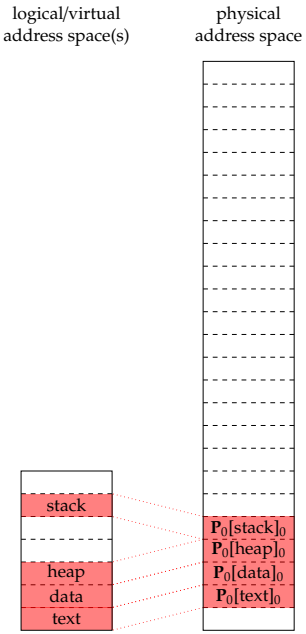
1. no MMU − relocation
   - no multi-programming
   - add relocation
2. no MMU + relocation
   - no protection
   - no virtualisation
   - add single base/limit register
   - add comparison
   - add translation (i.e., base plus offset)
3. per process segmented memory
   - contiguous allocation only
   - 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

Notes:

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.



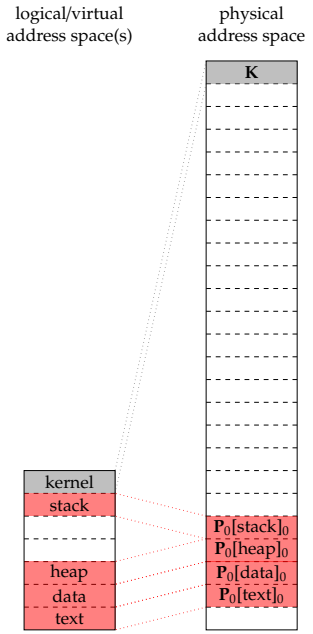
Notes:

- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

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.

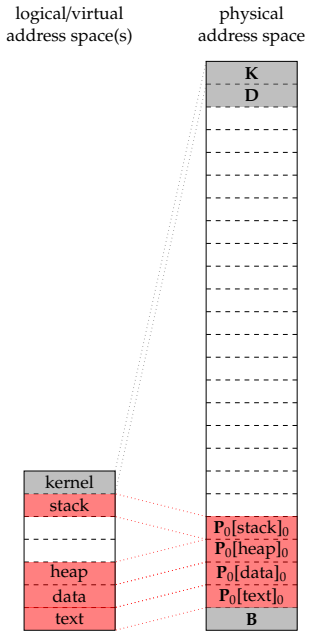


Notes:

- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

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.



Notes:

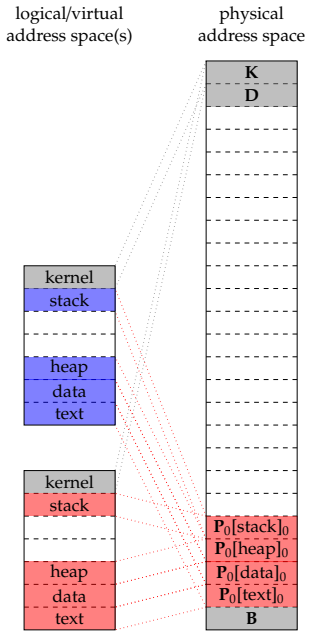
- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

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.



Notes:

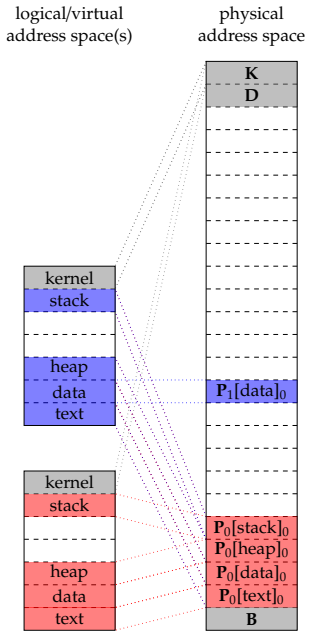
- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

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.



Notes:

- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

Policy (1)

- Idea: implement

  demand paging ≃ "lazy swapping"

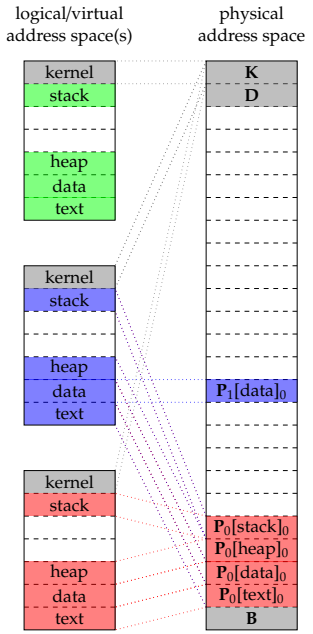
  i.e.,

  - 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`.



Notes:

- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

Policy (1)

- Idea: implement

  demand paging ≃ "lazy swapping"

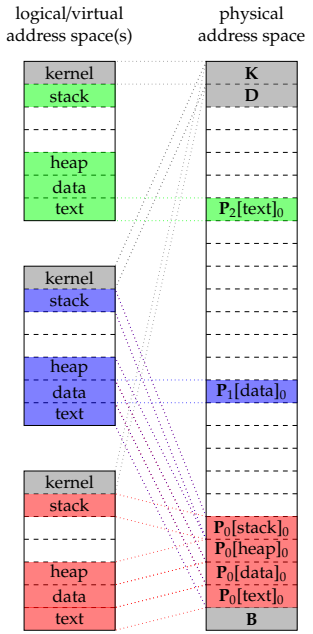
  i.e.,

  - 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`.



Notes:

- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

Policy (1)

- Idea: implement

  demand paging ≃ "lazy swapping"

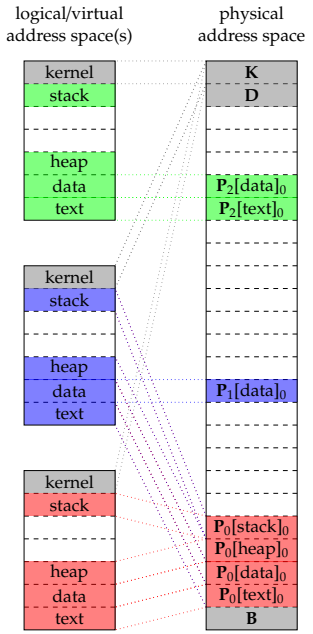
  i.e.,

  - 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`.



Notes:

- Note that the diagram here is illustrative only. Although one page is shown here and although it typically *will* be smaller than a user mode address space, the kernel address space is more general: it might, for example, have text, data, heap and stack segments within different pages. It is also important to note that this is where the set of page tables which describe both the kernel and user mode address spaces are stored. As shown later, we can swap-out pages: if the pages used to store the page tables are not locked, it is possible (and, as a result, totally mind-bending) to swap-out the page tables!
  Either way, the fact the kernel has a dedicated address space implies overhead wrt. switching between address spaces. For example, for each interrupt the kernel would have to first switch to use of the kernel mode address space, handle the interrupt, then switch back to use of a user mode address space: each switch means we (at least) need to switch the PTR(s) and potentially flush the TLB(s), the latter of which will have a knock-on effect in the form of increased miss-rate until the working set of pages is resident again.
  This motivates mapping the kernel mode address space into each user mode address space. The pages themselves can be protected st. a user mode instruction cannot access them, but by having them mapped means we no longer need to switch address spaces apart from when a context switch (of user mode processes) is required. In Linux, the dividing line between kernel and user mode pages is set by `TASK_SIZE` (e.g., 3GiB for a 4GiB, 32-bit address space meaning 1GiB for the kernel) which effectively limits how large a process can be.
- Various types of content are good candidates for sharing, e.g., text segments of processes stemming from the same program, or dynamically linked libraries used by multiple processes. In addition, sharing is a *requirement* for some functionality: to realise mechanisms for IPC based on shared memory, we need some shared (physical) memory!
  Likewise, various types of situation either necessitate or suggest locking (or pinning) pages so they cannot be swapped-out of physical memory; one example is where a region is involved in (asynchronous) DMA transfer.
- Strictly speaking, the term swapping is used when the granularity is an entire address space: we previously introduced it as such. Once we moved to paged memory specifically, swapping occurs at a per page granularity instead. This means the swap space could be termed a page space (or page file); this is easier to manage than in a more general context, since only fixed-size pages will ever be stored into or retrieved from it.
  For consistency we continue to use swap related terminology, but keep in mind that we swap-in or swap-out one page at a time now (which is just part of the overall address space for some process).
- General use of `mmap` may seem odd, but is the same mechanism a demand paged implementation would use to load an executable (e.g., a file `a.out`) into the text segment of a process. In fact, this suggests we can categorise pages held on-disk as being either
  - swapped-out pages (which are not associated with a file, so termed anonymous) that exist in the special-purpose swap file system (or swap space), or
  - memory-mapped pages (which are associated with a file) that exist in a general-purpose file system.

Implementation: demand paging (1)

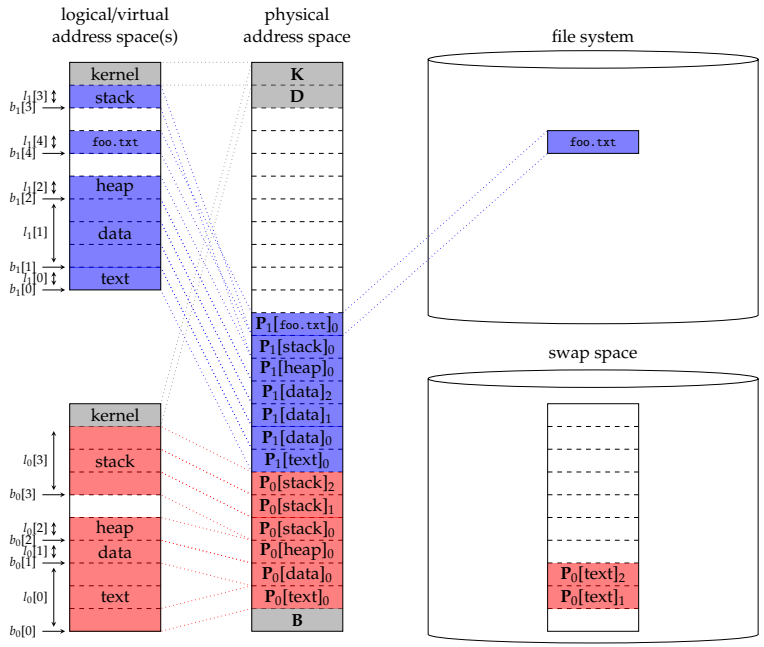
- To implement demand paging, the kernel needs (at least):
  1. a per process
     1.1 allocation table,
     1.2 page table, and
     1.3 swap table (or disk map)
  2. a global page frame table,
  3. a page frame allocation policy, and
  4. a page allocation policy

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

Notes:

- In some more detail:
  1. The allocation table basically just tracks the regions allocated to a process.
  2. Each page table describes the mapping of pages in that processes virtual address space to page frames in physical memory.
  3. Each swap table describes where in the swap space a given page is, iff. it is has been swapped-out to disk.
  4. The page frame table describes the allocation of page frames to pages in the virtual address spaces of processes. Depending on the data structure used there could be upto one entry per page frame, which roughly act as a reverse-map vs. the associated page tables. Searching for a free, unallocated page frame (via the table) can be accelerated by maintaining an associated and list of free page frames.
  5. The replacement policy basically performs *de*allocation of page frames. That is, it selects a page currently resident in a page frame; that page is evicted, thus allowing the page frame to be reused to house some *other* page.
  6. The allocation policy basically decides how to allocate memory to processes: this can be implicit (or fully automatic) in some cases, but it makes sense for it to be explicit in others.
- As an example of the options available wrt. data structures, Linux opts to use the page table (i.e., the PTEs) to index into a global swap table, rather than maintain an additional, dedicated swap table for each process.

Implementation: demand paging (2)

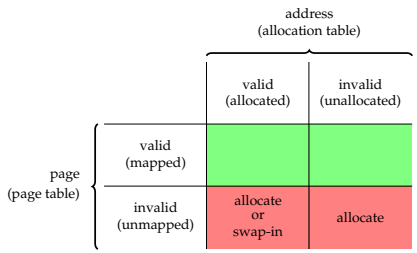


Notes:

Implementation: demand paging (3)

Algorithm

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



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.

Notes:

- Given the allocated address space

$$S = \bigcup_{\forall j} \{b_i[j] + 0, b_i[j] + 1, \ldots, b_i[j] + l_i[j] - 1\}$$

for some i-th process, then for an address x we can define

$$x \in S = \begin{cases} \textbf{true} & \Rightarrow \text{address is allocated} \\ \textbf{false} & \Rightarrow \text{address isn't unallocated} \end{cases}$$

and can therefore disambiguate

$$\text{PTE valid bit} = \begin{cases} \textbf{true} & \Rightarrow \text{page is mapped} \wedge \text{page is in memory} \\ \textbf{false} & \Rightarrow \text{page isn't mapped} \vee \text{page isn't in memory} \end{cases}$$

This is basically what allows the demand aspect of demand paging to work. The first part basically formalises the idea that the kernel maintains an allocation table for each process, it can determine if an address x is valid (i.e., within a previously allocated range) or not. The second part gives a more involved semantics for a PTE valid bit. *Previously* the valid bit indicated whether a page was valid (i.e., mapped to a page frame) or not, but now there are two cases. When a page fault occurs there is no valid mapping, but this could now be because either a) no mapping exists, *or* b) there is a mapping but it is currently invalid because the page has been swapped-out (so is not in memory); the allocation table allows the kernel to disambiguate these cases, and thus decide on the appropriate action to take.

- The top-right cell may seem odd: why is this green? The intuition is that demand paging is a policy realised by the kernel, meaning the allocation table is managed by the kernel. Put another way, even though it represents an odd case, no page fault occurs since the MMU correctly retrieves a valid PTE an so performs the access.
- This algorithm only attempts to capture the actions required when an invalid page fault occurs. Other, special-cases also exist of course (examples include writing to a page marked read-only or as copy-on-write): these will typically request a different exception type, and stem from the checks made when accessing each PTE.

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.

Notes:

- A process can execute without the entire associated virtual address space resident in physical memory: demand paging allows this, because if/when a page of the virtual address space *is* demanded it will be swapped-in. However, even though we can at least swap on a per page rather than per segment or per process basis, the problem remains of what to do if/when a free, unallocated page does not exist.
  Put another way, imagine we need to to allocate a new page or swap-in a previously swapped-out page; this implies a need to accommodate the page content in an unallocated page frame, and causes a problem when no such page frame exists (because they are all used). We could swap-out a page to create an unallocated page frame, but then the question is which one?

Implementation: demand paging (5) – page frame allocation

- 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.,

FIFO	$\Rightarrow$	select then replace oldest page
LRU	$\Rightarrow$	select then replace least-recently used page
LFU	$\Rightarrow$	select then replace least-frequently used page

     plus various LRU-approximations

  using them as follows ...

Notes:

- The page frame allocation algorithm can be viewed as making a difficult choice between acting locally or globally.
  The latter is more flexible, in that page frames can be allocated to a given process from the set of *all* those available rather than a fixed subset; this may allow a higher-priority process to be allocated more page frames, for example, and therefore incur less overhead wrt. swapping (since it will be more likely a given page is resident in page frame, rather than swapped-out). However, subtle disadvantages include the fact that processes cannot control their page fault rate: a process may incur overhead due to swapping because of the behaviour of another process. This makes it harder to predict the performance of any given process, since it will depend on behaviour of all other processes (and so may change over time, or wrt. the context they are executed in).
  Likewise, the former, local strategy makes no attempt to and so cannot cater for the size of working set associated with a given process: the allocation of page frames is limited, even if unallocated page frames exist that it could make effective use of (e.g., to reduce the amount of swapping required).
- Some more advanced replacement algorithms are supported by the MMU, in the sense it automatically maintains some form of access history; this might be fairly simple (e.g., a reference bit, which tracks whether or not a page has been accessed), but reduces the burden on the kernel. Keeping in mind that such a history potentially needs to be updated on a per access basis, this is reduction is important: without it, the overhead of using said algorithms would often be prohibitive.

Implementation: demand paging (6) – page frame allocation

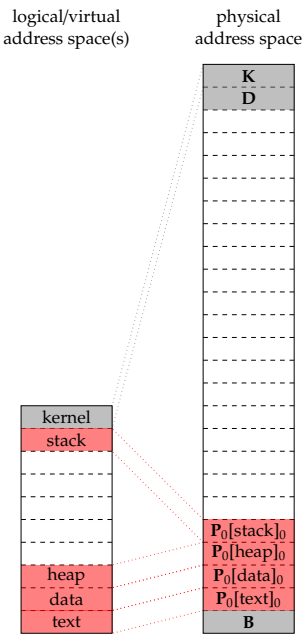
Algorithm

```
1  if ( page frame allocation algorithm allows new allocation ) ∧ ( an unallocated page frame exists ) then
2  |  select unallocated page frame f
3  else
4  |  select allocated page frame f using page frame replacement algorithm
5  |  if page p resident in page frame f is dirty then
6  |  |  store p in swap space
7  |  else
8  |  |  discard p
9  |  end
10 end
11 return f
```

Notes:

Implementation: demand paging (7) – page allocation

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



Notes:

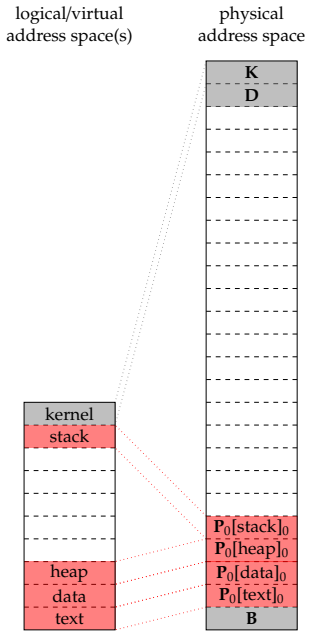
- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [4, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point it is deemed to have overflowed.
  - Management of the heap segment is mainly performed in user mode, e.g., by functions such as `malloc`. Cases can obviously occur when `malloc` cannot satisfy a request because the heap segment is too small. In such cases, it is enlarged to cope as the result of an explicit request (via `brk`).
- It might seem that a simple policy suffices for both the stack and heap: if x is in an adjacent page to a enlargeable segment, then enlarge that segment and restart the instruction. Differences between the two exist, however: for example
  - enlargement of the stack would typically be in small(er) increments, be more frequent and triggered by user-instructions, and
  - enlargement of the heap would typically be in large(r) increments, be less frequent and triggered by library-instructions

  motivate differing mechanisms, and, in particular, a manual approach to growth for the heap segment.

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`.



Notes:

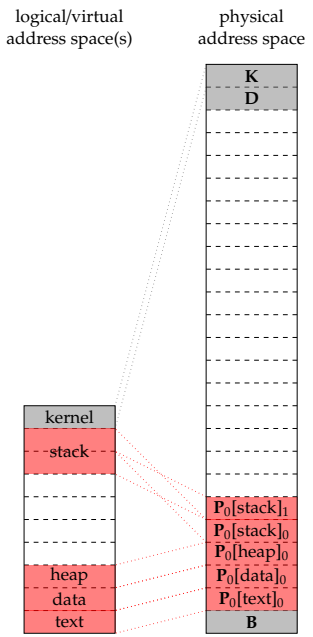
- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [4, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point it is deemed to have overflowed.
  - Management of the heap segment is mainly performed in user mode, e.g., by functions such as `malloc`. Cases can obviously occur when `malloc` cannot satisfy a request because the heap segment is too small. In such cases, it is enlarged to cope as the result of an explicit request (via `brk`).
- It might seem that a simple policy suffices for both the stack and heap: if x is in an adjacent page to a enlargeable segment, then enlarge that segment and restart the instruction. Differences between the two exist, however: for example
  - enlargement of the stack would typically be in small(er) increments, be more frequent and triggered by user-instructions, and
  - enlargement of the heap would typically be in large(r) increments, be less frequent and triggered by library-instructions

  motivate differing mechanisms, and, in particular, a manual approach to growth for the heap segment.

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`.



Notes:

- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [4, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point it is deemed to have overflowed.
  - Management of the heap segment is mainly performed in user mode, e.g., by functions such as `malloc`. Cases can obviously occur when `malloc` cannot satisfy a request because the heap segment is too small. In such cases, it is enlarged to cope as the result of an explicit request (via `brk`).
- It might seem that a simple policy suffices for both the stack and heap: if x is in an adjacent page to a enlargeable segment, then enlarge that segment and restart the instruction. Differences between the two exist, however: for example
  - enlargement of the stack would typically be in small(er) increments, be more frequent and triggered by user-instructions, and
  - enlargement of the heap would typically be in large(r) increments, be less frequent and triggered by library-instructions

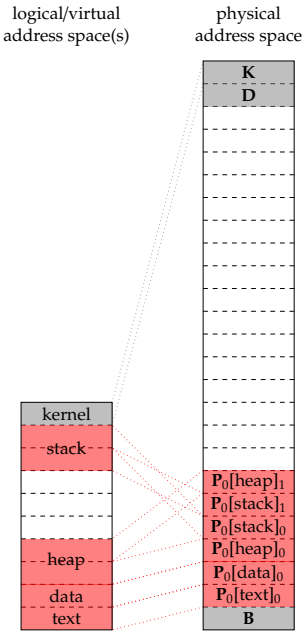
  motivate differing mechanisms, and, in particular, a manual approach to growth for the heap segment.

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

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`.



Notes:

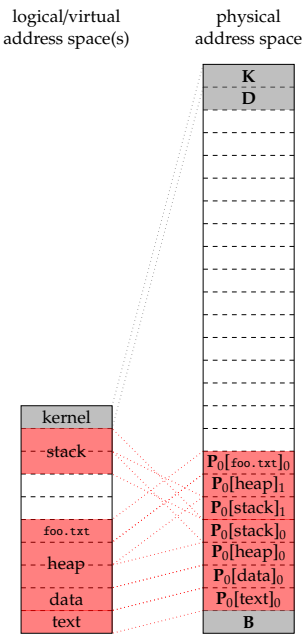
- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [4, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point it is deemed to have overflowed.
  - Management of the heap segment is mainly performed in user mode, e.g., by functions such as `malloc`. Cases can obviously occur when `malloc` cannot satisfy a request because the heap segment is too small. In such cases, it is enlarged to cope as the result of an explicit request (via `brk`).
- It might seem that a simple policy suffices for both the stack and heap: if x is in an adjacent page to a enlargeable segment, then enlarge that segment and restart the instruction. Differences between the two exist, however: for example
  - enlargement of the stack would typically be in small(er) increments, be more frequent and triggered by user-instructions, and
  - enlargement of the heap would typically be in large(r) increments, be less frequent and triggered by library-instructions

  motivate differing mechanisms, and, in particular, a manual approach to growth for the heap segment.

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`.



Notes:

- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [4, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point it is deemed to have overflowed.
  - Management of the heap segment is mainly performed in user mode, e.g., by functions such as `malloc`. Cases can obviously occur when `malloc` cannot satisfy a request because the heap segment is too small. In such cases, it is enlarged to cope as the result of an explicit request (via `brk`).
- It might seem that a simple policy suffices for both the stack and heap: if x is in an adjacent page to a enlargeable segment, then enlarge that segment and restart the instruction. Differences between the two exist, however: for example
  - enlargement of the stack would typically be in small(er) increments, be more frequent and triggered by user-instructions, and
  - enlargement of the heap would typically be in large(r) increments, be less frequent and triggered by library-instructions

  motivate differing mechanisms, and, in particular, a manual approach to growth for the heap segment.

Implementation: demand paging (8) – performance

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

Notes:

- The access latency quoted here will clearly depend strongly on the underlying technology used. For example, a memory access satisfied by the L1 cache rather than main memory will have a *much* lower latency (perhaps a factor of 100 or so less); an SSD-based disk will have a *much* lower latency than a traditional alternative (perhaps a factor of 10 or so less). Likewise, for disks (more so than but also memory when caches are taken into account) there is likely to be constant factors related to seek time and so whether the access is to sequential or random data.

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

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

  so ideally we minimise such events.

Notes:

- The access latency quoted here will clearly depend strongly on the underlying technology used. For example, a memory access satisfied by the L1 cache rather than main memory will have a *much* lower latency (perhaps a factor of 100 or so less); an SSD-based disk will have a *much* lower latency than a traditional alternative (perhaps a factor of 10 or so less). Likewise, for disks (more so than but also memory when caches are taken into account) there is likely to be constant factors related to seek time and so whether the access is to sequential or random data.

© Daniel Page ⟨cdp@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 1d9e4fa @ 2018-02-08 University of BRISTOL

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,

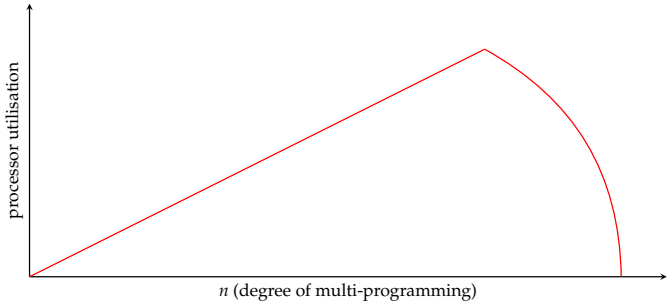
$$\begin{array}{lcl} \text{larger } n & \Rightarrow & \text{higher processor utilisation} \\ \text{larger } n & \Rightarrow & \text{higher contention wrt. page frames} \end{array}$$

so, we find ...

Notes:

- The access latency quoted here will clearly depend strongly on the underlying technology used. For example, a memory access satisfied by the L1 cache rather than main memory will have a *much* lower latency (perhaps a factor of 100 or so less); an SSD-based disk will have a *much* lower latency than a traditional alternative (perhaps a factor of 10 or so less). Likewise, for disks (more so than but also memory when caches are taken into account) there is likely to be constant factors related to seek time and so whether the access is to sequential or random data.

Implementation: demand paging (9) – performance

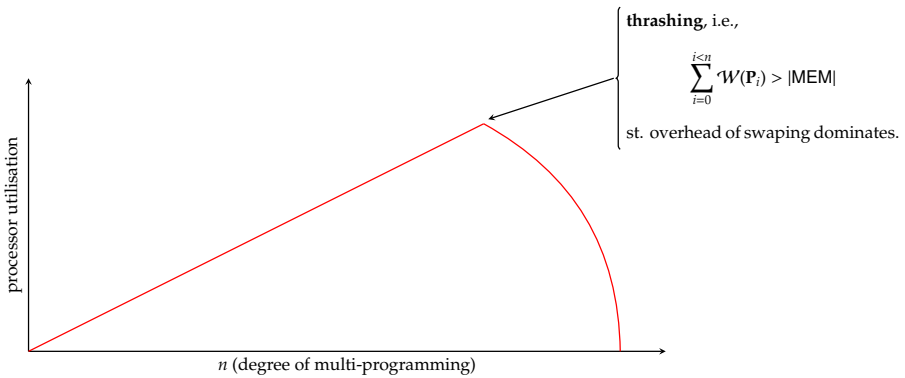


Notes:

- Linux uses the so-called kernel swap daemon `kswapd`, which is basically a high(er)-level process tasked with a) observing then b) fine-tuning the memory management sub-system: invoked intermittently, `kswapd` can detect and attempt recover from any low-memory situations (i.e., few or no unallocated page frames) which will lead to thrashing. This topic is described in detail by [3], which covers aspects such as the Out-Of-Memory (OOM) "killer" (the policy for recovering from low-memory situations by selecting then terminating processes) also overviewed by

  `http://linux-mm.org/OOM_Killer`

Implementation: demand paging (9) – performance



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

     too high ⇒ too few page frames allocated
     too low ⇒ too many page frames allocated

     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.

Notes:

- Linux uses the so-called kernel swap daemon `kswapd`, which is basically a high(er)-level process tasked with a) observing then b) fine-tuning the memory management sub-system: invoked intermittently, `kswapd` can detect and attempt recover from any low-memory situations (i.e., few or no unallocated page frames) which will lead to thrashing. This topic is described in detail by [3], which covers aspects such as the Out-Of-Memory (OOM) "killer" (the policy for recovering from low-memory situations by selecting then terminating processes) also overviewed by

  `http://linux-mm.org/OOM_Killer`

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

    demand paging ⊂ memory management

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

Notes:

- A non-exhaustive list of topics *not* covered, or covered in a more superficial level than ideal, include
  - other page table organisations (e.g., inverted page tables),
  - other page frame replacement algorithms (e.g., the so-called clock algorithm)
  - performance analyses and features (e.g., Belady's Anomaly),
  - techniques for pre-fetching or buffering pages,
  - swap space management.

  Some of these *are* covered in the recommended reading: see in particular [9, 5, 6].
- Keep in mind that despite our covering this topic in some depth, it remains possible to develop a kernel that supports multi-programming *without* using an MMU: the project at

  `http://nommu.org`

  does exactly this, albeit caveated wrt. efficiency, protection etc.

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.

Notes:

References

[1] *Wikipedia: Memory management*. URL: http://en.wikipedia.org/wiki/Memory_management (see p. 145).

[2] *Wikipedia: Virtual memory*. URL: http://en.wikipedia.org/wiki/Virtual_memory (see p. 145).

[3] M. Gorman. "Chapter 13: Out of memory management". In: *Understanding the Linux Virtual Memory Manager*. http://www.kernel.org/doc/gorman/. Prentice Hall, 2004 (see pp. 140, 142).

[4] M. Gorman. *Understanding the Linux Virtual Memory Manager*. http://www.kernel.org/doc/gorman/. Prentice Hall, 2004 (see pp. 124, 126, 128, 130, 132, 145).

[5] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 8: Memory management strategies". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see pp. 30, 32, 34, 36, 38, 40, 42, 44, 46, 144, 145).

[6] A. Silberschatz, P.B. Galvin, and G. Gagne. "Chapter 9: Virtual-memory management". In: *Operating System Concepts*. 9th ed. Wiley, 2014 (see pp. 144, 145).

[7] 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. 145).

[8] 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. 145).

[9] A.S. Tanenbaum and H. Bos. "Chapter 3: Memory managament". In: *Modern Operating Systems*. 4th ed. Pearson, 2015 (see pp. 30, 32, 34, 36, 38, 40, 42, 44, 46, 144, 145).

[10] R. Uhlig et al. "Design Tradeoffs for Software-managed TLBs". In: *ACM Transactions on Computer Systems (TOCS)* 12.3 (1994), pp. 175–205 (see pp. 76, 78, 80).

[11] 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. 76, 78, 80, 81, 83–86, 88–90, 92).

[12] ARM Limited. *Cortex-A8 Technical Reference Manual*. Tech. rep. DDI-0344K. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html. 2010 (see pp. 84, 86).

[13] ARM Limited. *Cortex-A9 Technical Reference Manual*. Tech. rep. DDI-0388E. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388e/index.html. 2012 (see pp. 84, 86).

Notes: