Concurrent Computing (Operating Systems)

Daniel Page

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

February 16, 2016

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:

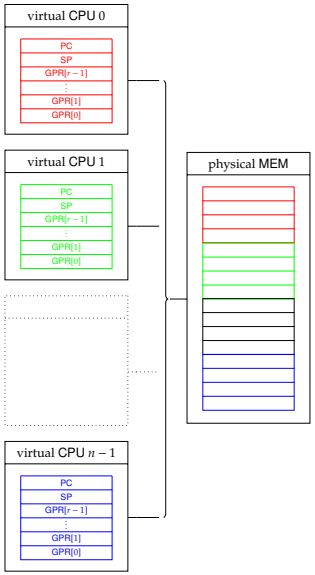
Notes:

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).

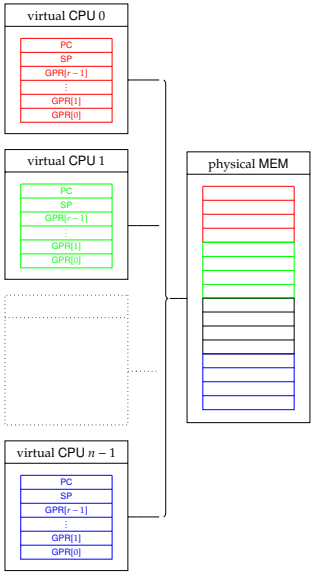
COMS20001 lecture: week #16

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.



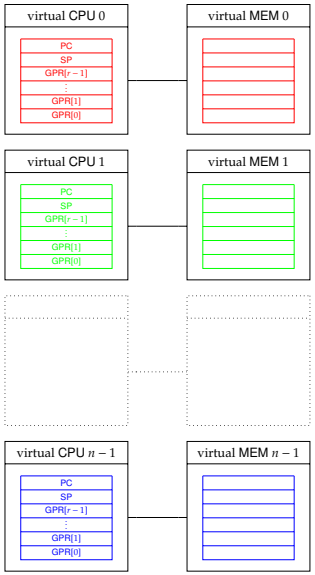
Notes:

- 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).

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*?



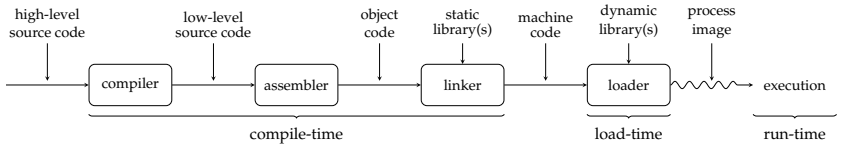
Notes:

- 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)

Definition (**resolution**, **relocation** etc.)

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



noting that

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

    ```
    goto foo;
    ```

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

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

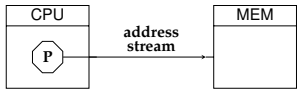
- abstract addresses are **resolved** into concrete addresses before execution, and
- addresses may be **relocated** (or moved, i.e., rewritten) before *or* during execution.

Notes:

Concept (2)

Definition (**address stream** etc.)

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



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

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

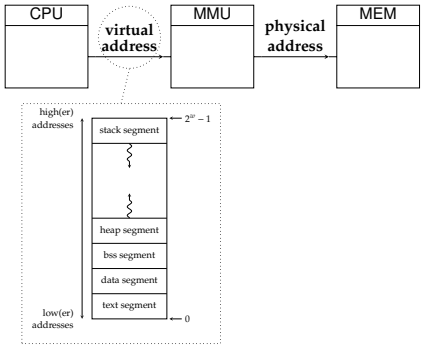
Notes:

- Recall there are two types of access locality, namely
  - **spatial locality**: if address x is accessed at time t, address x is likely to be accessed at time $t + 1$, and
  - **temporal locality**: if address x is accessed at time t, address $x \pm \delta$ is likely to be accessed at time $t + 1$.
- In the context of define what working set means, the "currently" part of "currently in use" can be tricky to capture. Typically it is approximated by defining the working set as those addresses accessed within some window of time, i.e., the last δ time units for an appropriate choice of δ; we know the principle of locality will apply, so the intuition is that those addresses used recently should approximate those "currently" useful, and so "in use".

Concept (3)

Definition (**address space** etc.)

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



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

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

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

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.
- 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 (4)

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

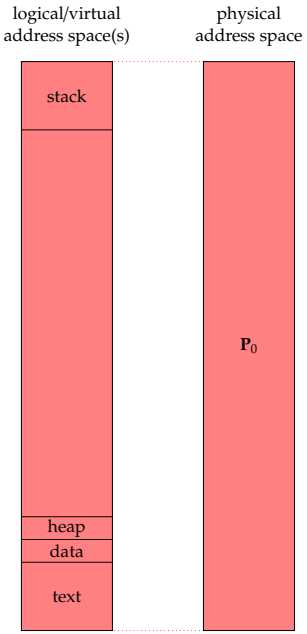
  and hence *virtualise* the physical memory.

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	×



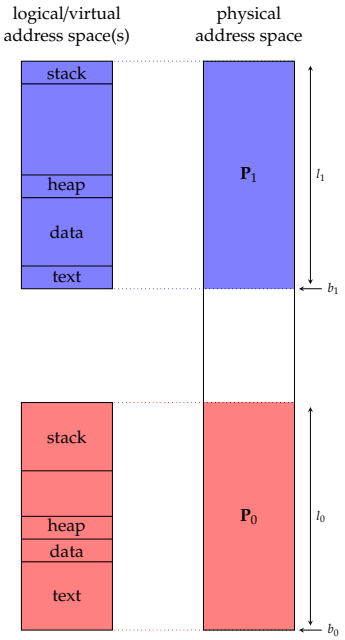
Notes:

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	✓	×



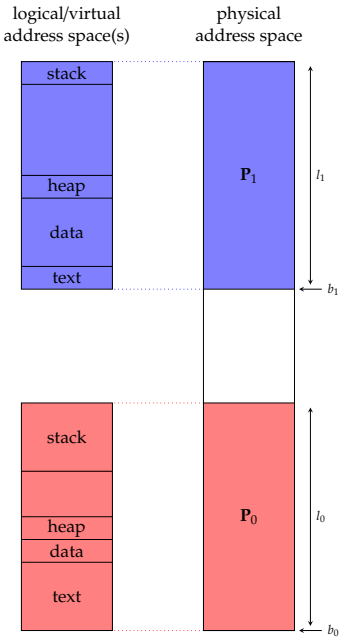
Notes:

- 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	✓	×

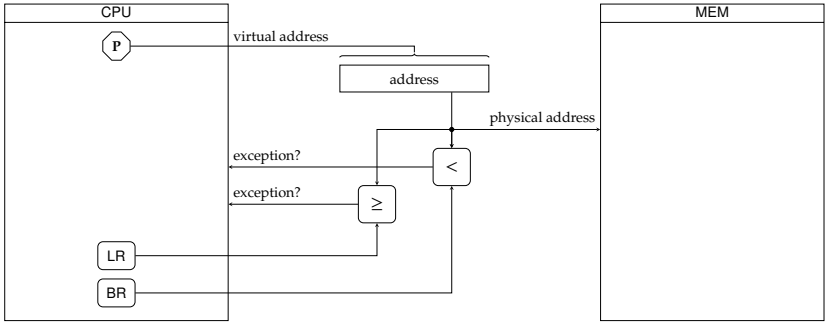


Notes:

- 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.,

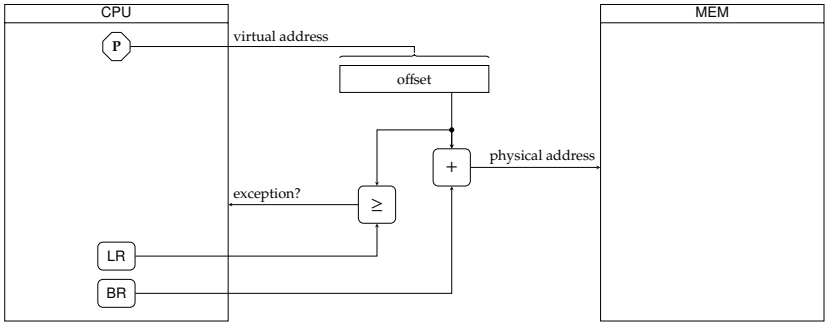


Notes:

- 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:

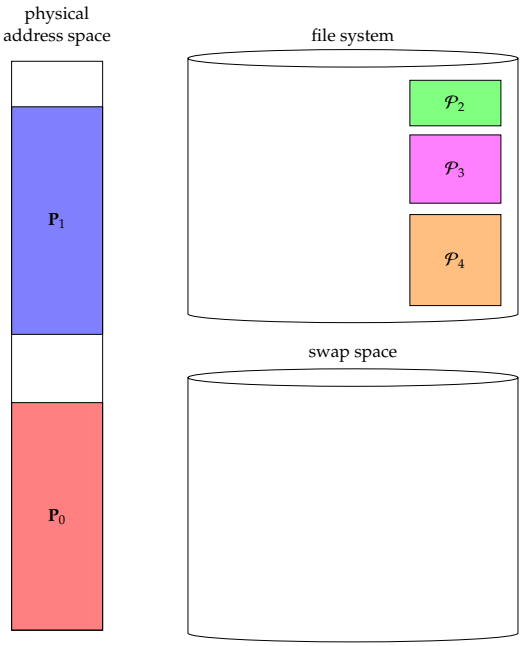
- This is the first time the term offset is used, but here, and here after, the same concept might be described as a displacement: what it represents, either way, is a value added to the base address to yield the target address.

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 94eafb3 @ 2016-02-15 University of BRISTOL

An Aside: allocation, swapping and fragmentation

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

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



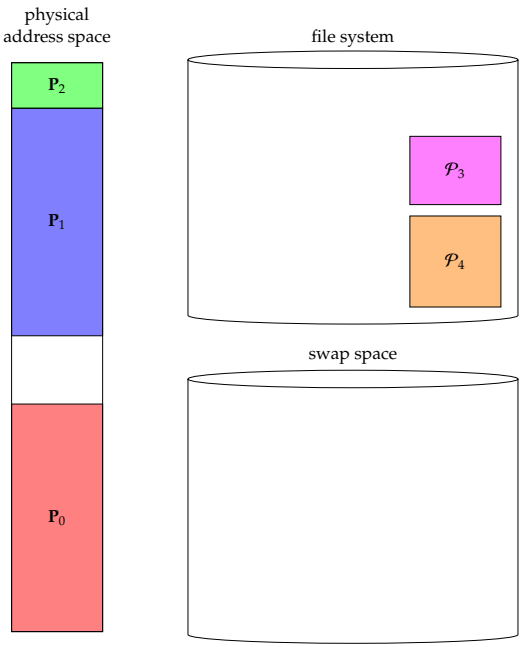
Notes:

- 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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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 should we load $\mathcal{P}_i$.
- Solution: we need
  1. an allocation algorithm, e.g.,
     - first-fit,
     - best-fit,
     - worst-fit,
     - ...

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



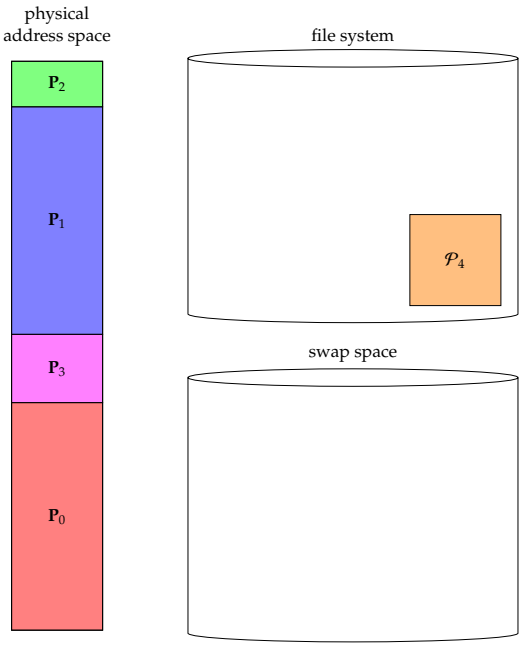
Notes:

- 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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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 should we load $\mathcal{P}_i$.
- Solution: we need
  1. an allocation algorithm, e.g.,
     - first-fit,
     - best-fit,
     - worst-fit,
     - ...

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

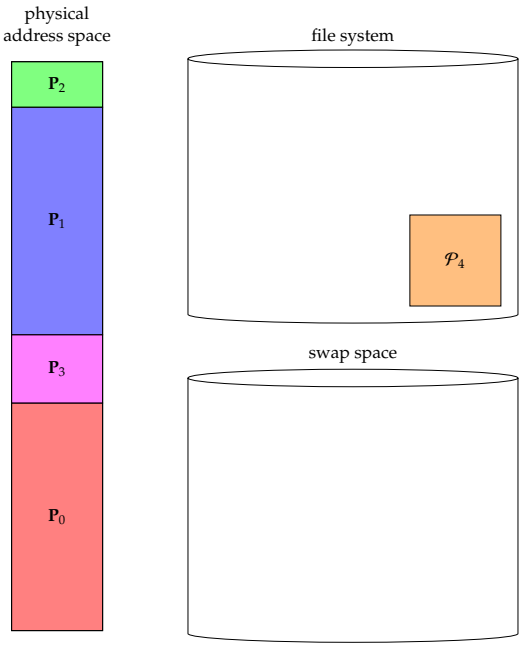


Notes:

- 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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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.

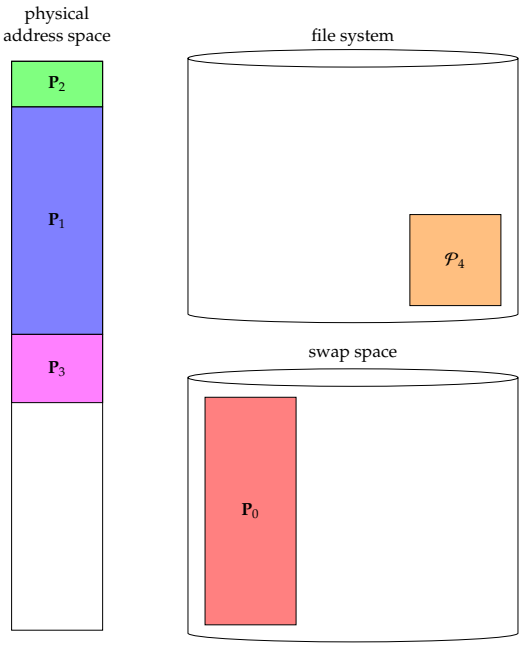


Notes:

- 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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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.

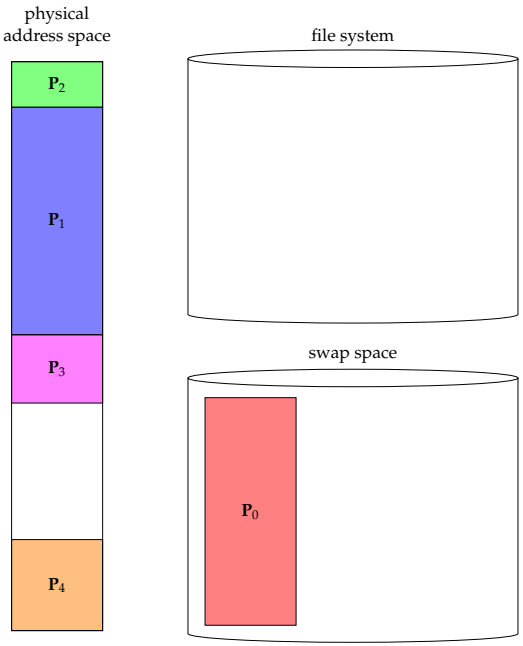


Notes:

- 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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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.

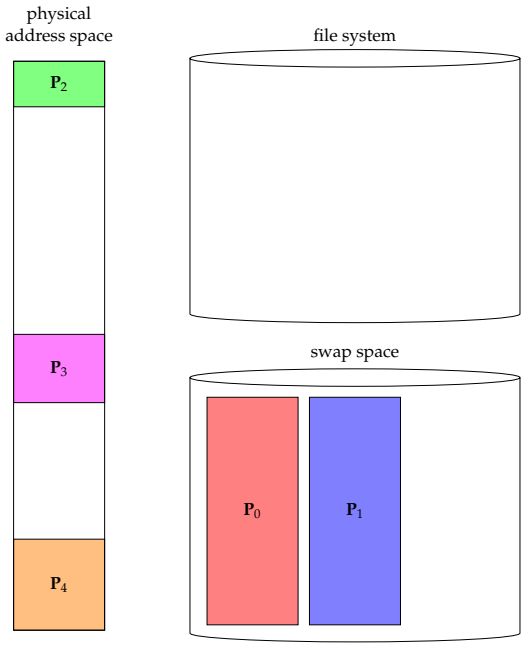


Notes:

- 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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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.

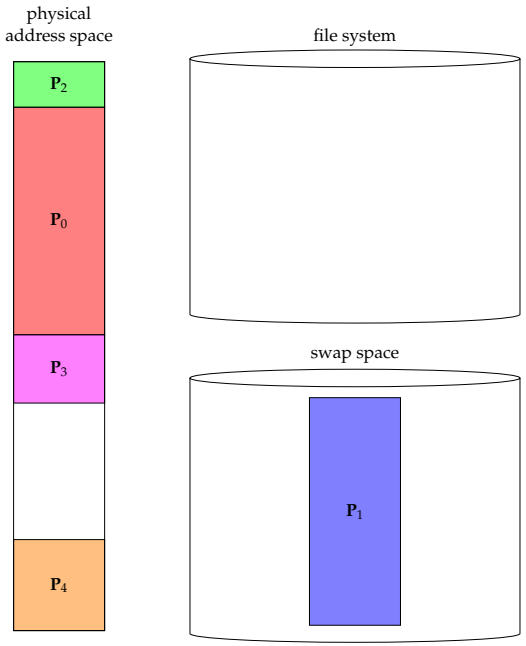


Notes:

- 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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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.



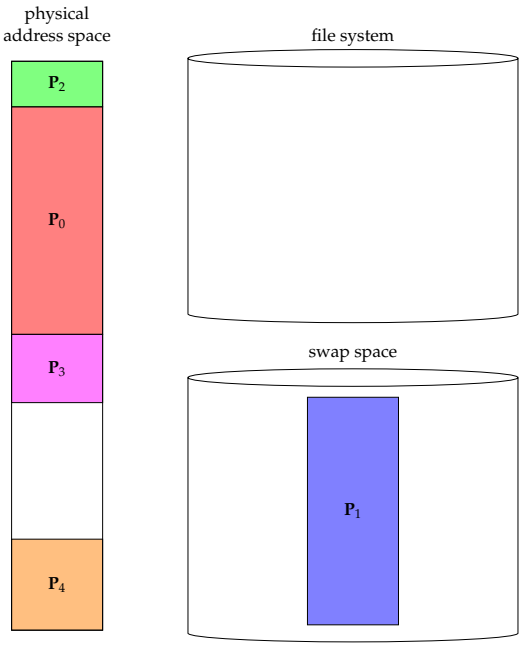
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 94eafb3 @ 2016-02-15 University of BRISTOL

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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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):
  - **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).
- [9, Sections 8.3.2 and 8.8.3] or [15, 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

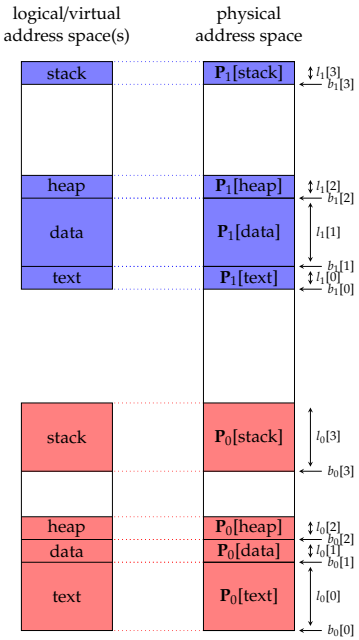
$$0 \leq \text{LSB}_{w-t}(x) < l[\text{MSB}_t(x)],$$

  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

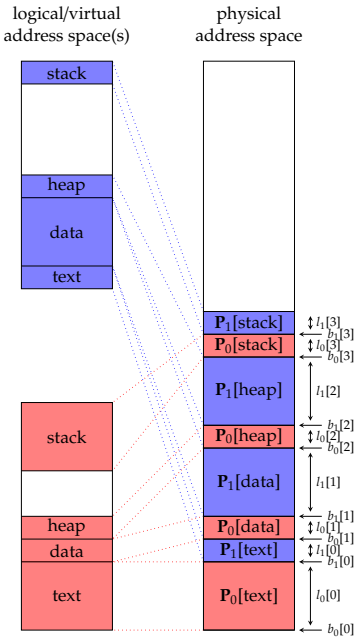
$$0 \leq \text{LSB}_{w-t}(x) < l[\text{MSB}_t(x)],$$

  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	✓

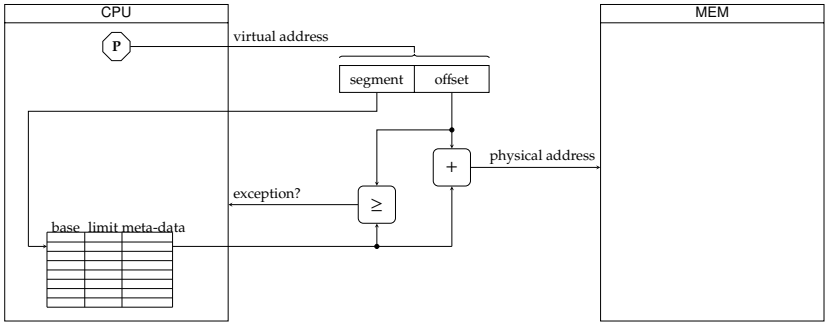


Notes:

- – 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.,



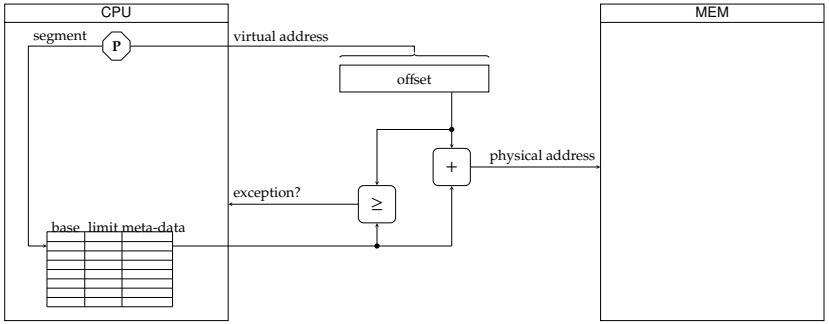
noting we *could* opt to index into the table via

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**

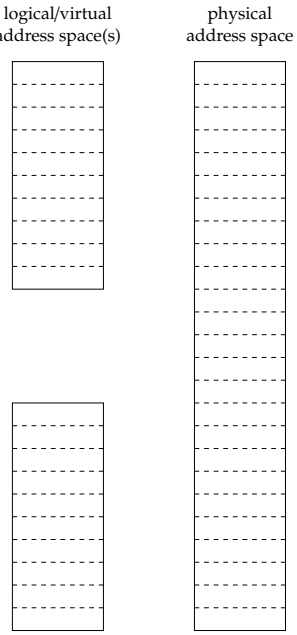
    of l bytes in each case,
  - maintain a **page table** T per process,
  - let $t = \log_2(|T|)$, and translate st.

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

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 unmapping, 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.

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

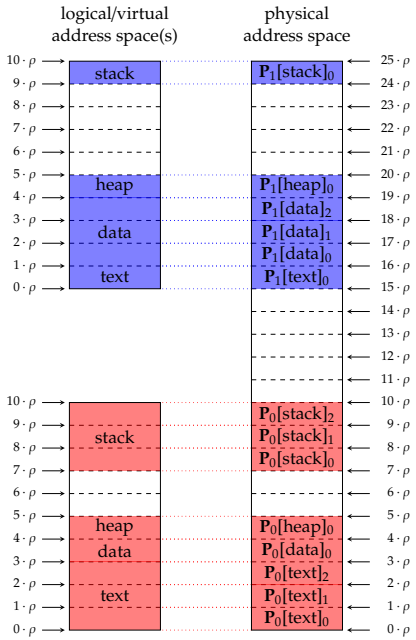
noting no check is required since

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

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 unmapping, 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.

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

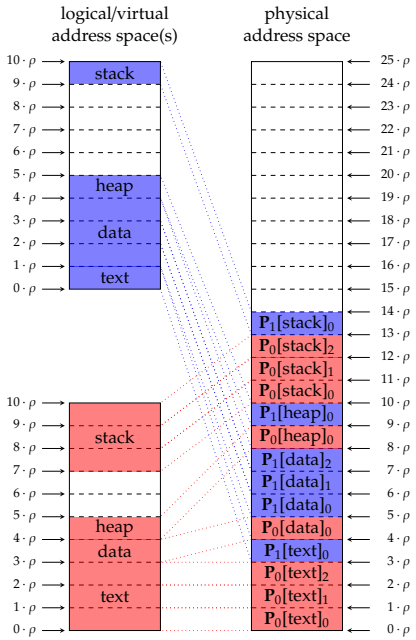
noting no check is required since

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

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	✓



Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 94eafb3 @ 2016-02-15 University of BRISTOL

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 unmapping, 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.

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

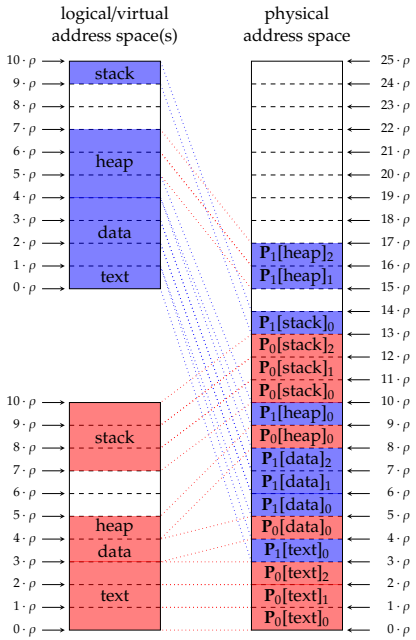
    noting no check is required since

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

    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	✓



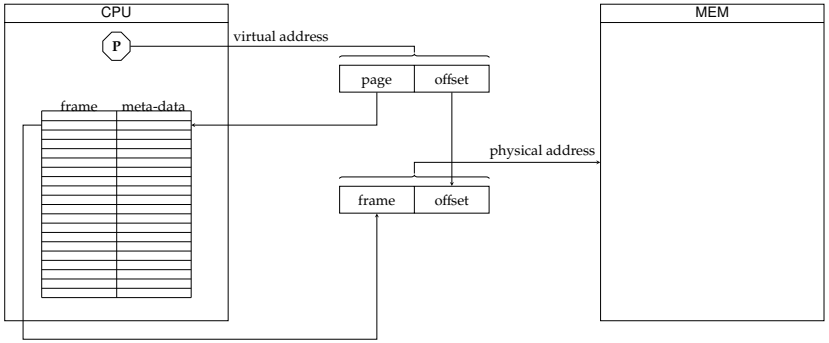
Notes:

- – 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 unmapping, 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.,



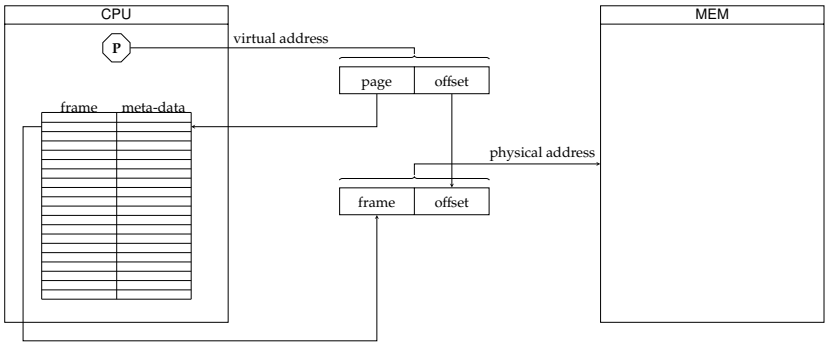
noting the page table consists of **Page Table Entries (PTEs)**.

Notes:

- Versus the segment table used in the previous approach, a page table will be large. As described (so far), we need one PTE per page in the virtual address space. So, 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 (cf. say 4 or so segments).

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,

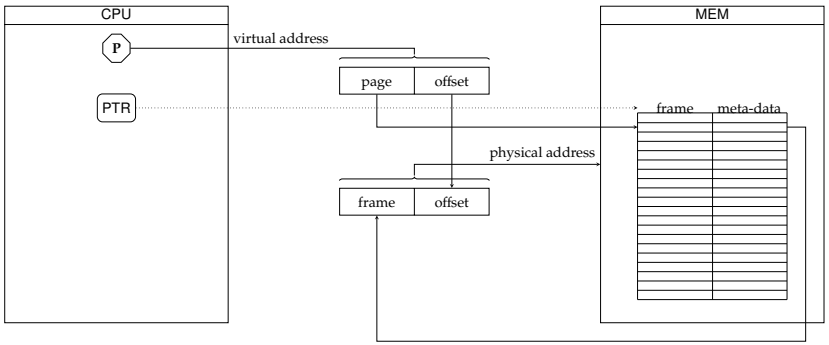
   st. cached PTEs for one process cannot be used by another.

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.

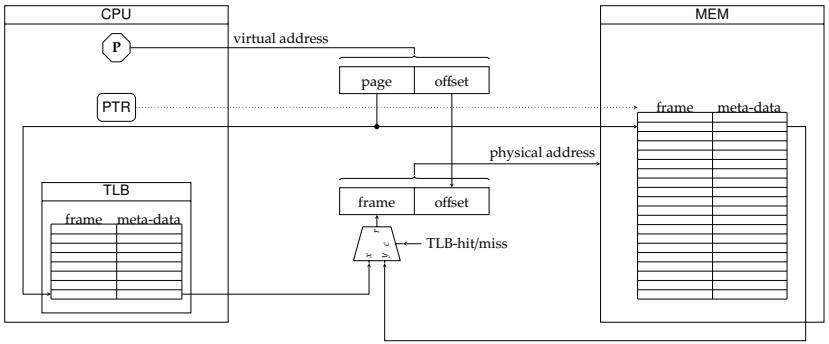
Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 94eafb3 @ 2016-02-15 University of BRISTOL

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,

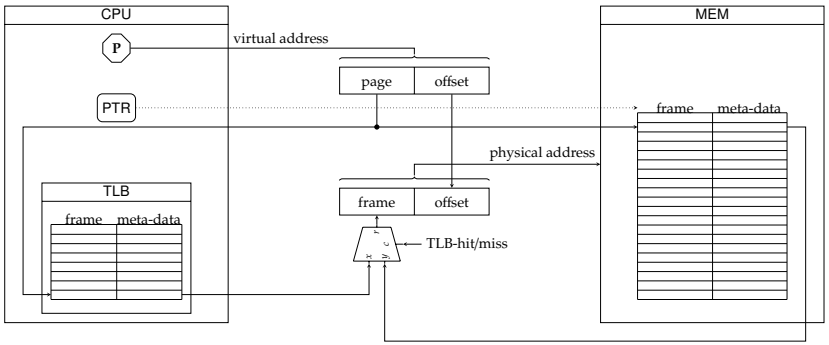
   st. cached PTEs for one process cannot be used by another.

Notes:

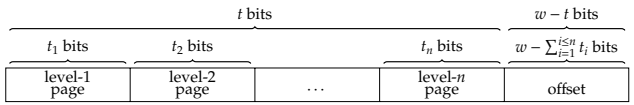
- An important thing to keep in mind is that the TLB caches PTEs, *not* the pages themselves: is simply accelerates the task of address translation.

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.,



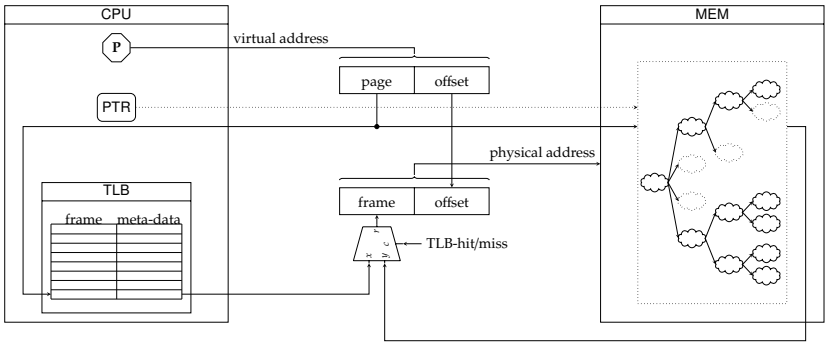
3. use a valid flag to indicate whether or not a sub-tree exists.

Notes:

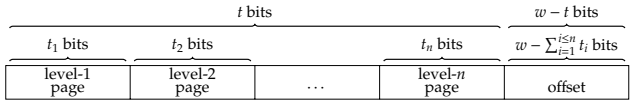
- The way this is realised is for the page number to be split into more than one component: each i-th component is used as an index into the level-i page table.
- One disadvantage is that to get from the initial PTR to the actual page we want, assuming a TLB miss we need to walk the page table. Although it is possible to do this in hardware *or* software, 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. Although it is possible to do this in hardware *or* software, 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:

- The terms soft and hard are often used interchangeable with minor and major when used to describe these exceptions.
- In [7, 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:

- The terms soft and hard are often used interchangeable with minor and major when used to describe these exceptions.
- In [7, 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

Notes:

- The terms soft and hard are often used interchangeable with minor and major when used to describe these exceptions.
- In [7, 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)** [7, Chapter B5] and
  2. the **Virtual Memory System Architecture (VMSA)** [7, Chapter B3]

  both of which are controlled via the co-processor interface [7, 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 [7, 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.
- Although ARMv7-A specifies some [7, 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 [5, 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 [6, 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.

Implementation: ARMv7-A (2) VMSA

- Some details:

3. It supports four page sizes [7, Section B3.3]

$$\begin{array}{rcrcl} \text{small page} & \Rightarrow & \rho = & 4\text{KiB} & \rightsquigarrow \text{12-bit offsets} \\ \text{large page} & \Rightarrow & \rho = & 64\text{KiB} & \rightsquigarrow \text{16-bit offsets} \\ \text{section} & \Rightarrow & \rho = & 1\text{MiB} & \rightsquigarrow \text{20-bit offsets} \\ \text{super-section} & \Rightarrow & \rho = & 16\text{MiB} & \rightsquigarrow \text{24-bit offsets} \end{array}$$

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.
- Although ARMv7-A specifies some [7, 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 [5, 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 [6, 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

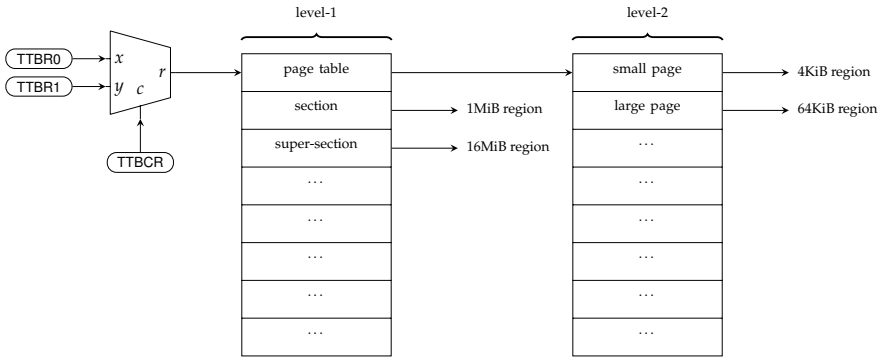
  which support the **Address Space IDentifier (ASID)** to avoid a TLB flush during a context switch.

Implementation: ARMv7-A (3) VMSA

Example

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

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



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

Notes:

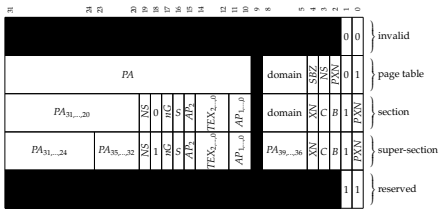
- The net result of this page table organisation is that when larger page sizes (i.e., sections) are used,
  - there's no need to walk through both levels of the hierarchy to get to a page, and
  - large regions captured by such a page will occupy only one TLB entry.
- It might not be obvious, but, to 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).
- [7, 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.

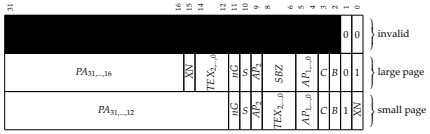
The format of (short) PTEs

- at level-1 [7, Figure B3-4] is



and

- at level-2 [7, 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).
- [7, 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 | | raise exception
6 | **end**
7 **else**
8 | **if** *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** raise 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** raise exception
16 | load from MEM[$E_2[PA] + x_{11,...,0}$]
17 | update TLB(s)
18 **end**

Daniel Page ⟨daniel.page@bristol.ac.uk⟩
Concurrent Computing (Operating Systems) git # 94eafb3 @ 2016-02-15 University of BRISTOL

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).
- [7, 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).

Continued in next lecture ...

Notes:

References

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

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

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

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

[5] ARM Limited.
Cortex-A8 Technical Reference Manual.
Technical Report DDI-0344K, 2010.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0344k/index.html.

[6] ARM Limited.
Cortex-A9 Technical Reference Manual.
Technical Report DDI-0388E, 2012.
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388e/index.html.

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

Notes:

References

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

[9] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 8: Memory management strategies.
In *Operating System Concepts* [11].

[10] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 9: Virtual-memory management.
In *Operating System Concepts* [11].

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

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

[13] A. N. Sloss, D. Symes, and C. Wright.
Chapter 13: Memory protection units.
In *ARM System Developer's Guide: Designing and Optimizing System Software* [12].

Notes:

References

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

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

Notes: