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:

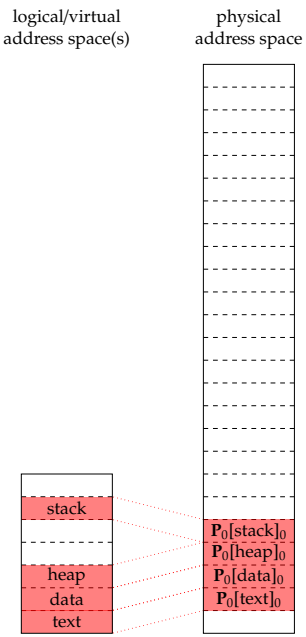
Continued from last lecture ...

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.



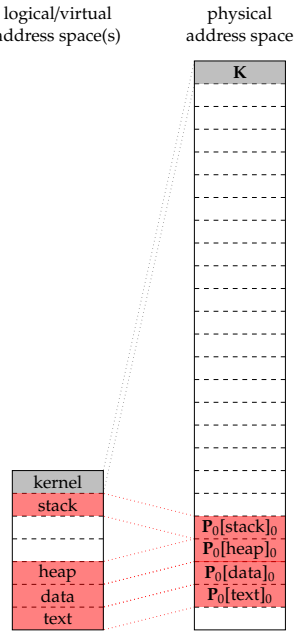
Notes:

- 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.

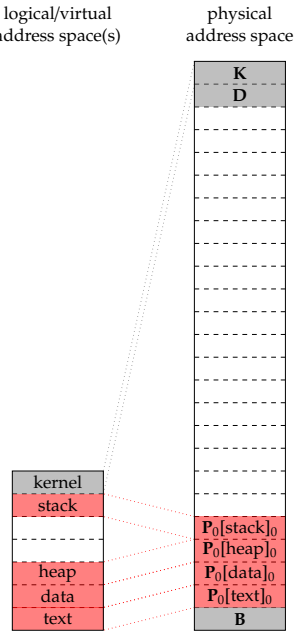


Notes:

- 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.



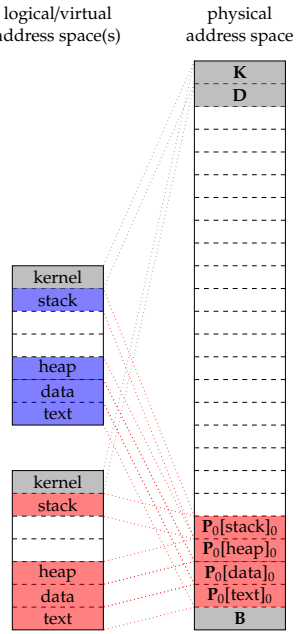
Notes:

- 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.



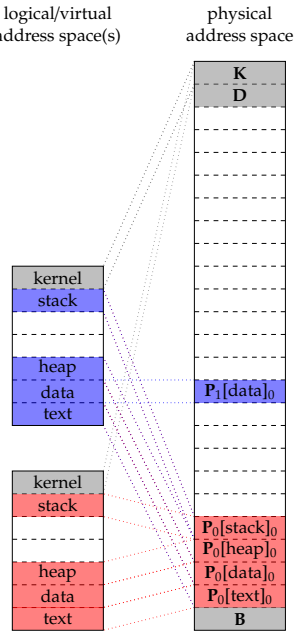
Notes:

- 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.



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

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"

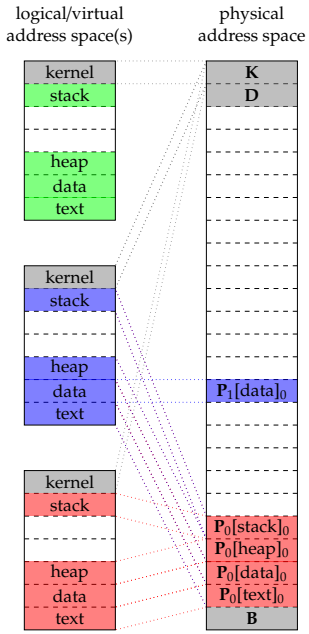
  i.e.,

  - 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"

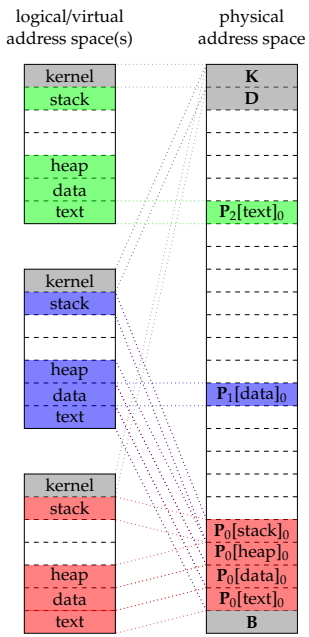
  i.e.,

  - 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`.



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

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"

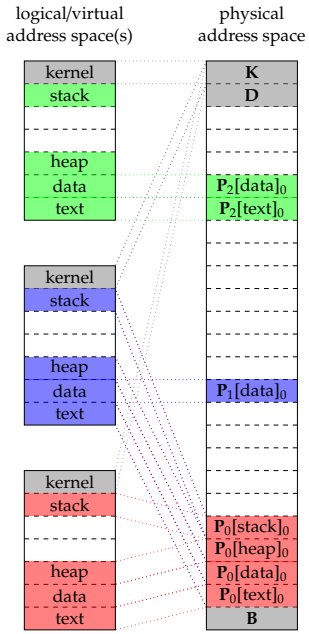
  i.e.,

  - 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)

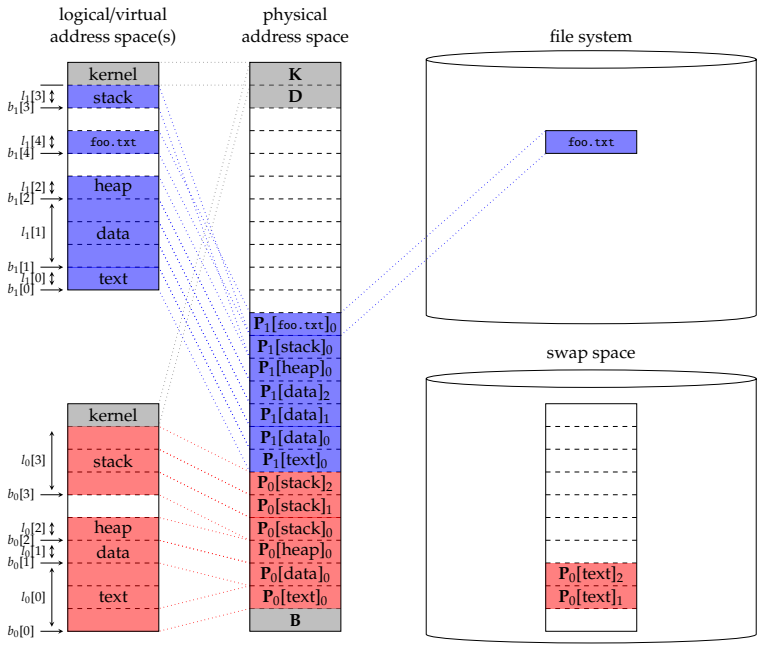
- To implement demand paging, the kernel needs (at least):
  1. a per process
     1.1 allocation table,
     1.2 page table, and
     1.3 swap table (or disk map)
  2. a global page frame table,
  3. a page frame allocation policy, and
  4. a page allocation policy

  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 (**demand paging**)

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

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

noting that

- the red cases cause an invalid page fault, whereas the green case might complete as is ...
- ... modulo special-cases such as copy-on-write,
- the allocation policy could fail, meaning it decides the right action is to raise 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 raise 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

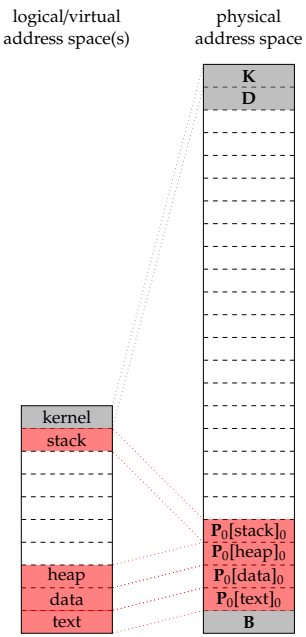
Algorithm (allocate page frame)

```
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:

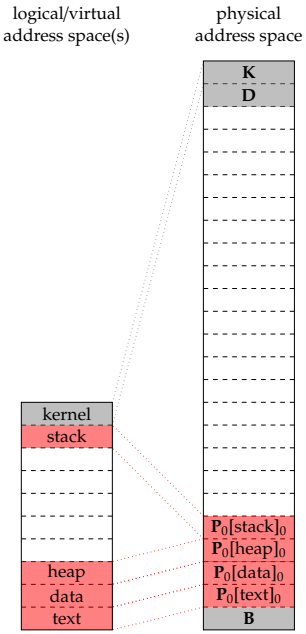
- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [3, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point 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:

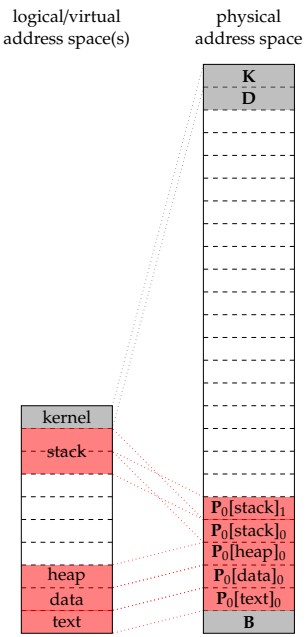
- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [3, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point 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 [3, 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

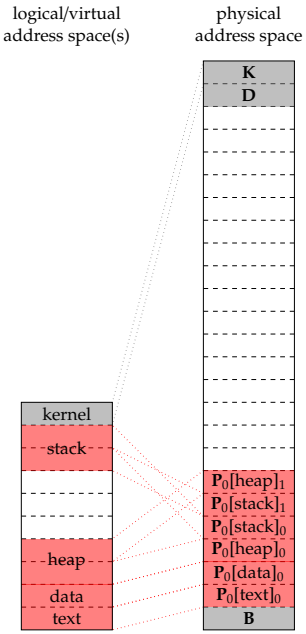
  motivate differing mechanisms, and, in particular, a manual approach to growth for the heap segment.

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

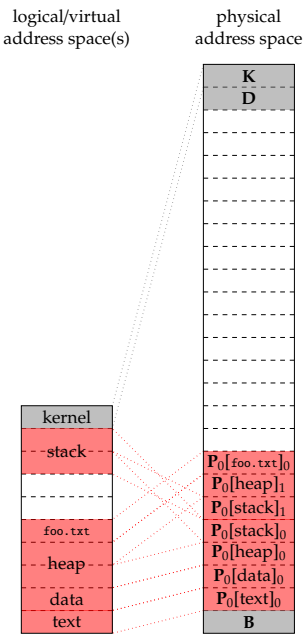
- In Linux, for example, various specific mechanisms exist to enlarge specific segments: see [3, Section 4.6.1]:
  - The stack segment may need to be enlarged as the result of deeply nested function calls. It is enlarged to automatically, upto the point it exceeds a limit `RLIMIT_STACK`: at this point 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 [3, 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.

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,

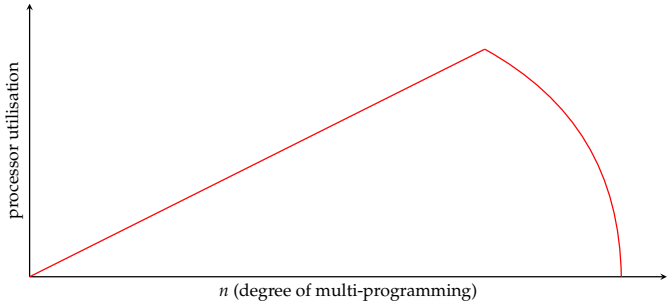
$$\begin{array}{lcl} \text{larger } n & \Rightarrow & \text{higher processor utilisation} \\ \text{larger } n & \Rightarrow & \text{higher contention wrt. page frames} \end{array}$$

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

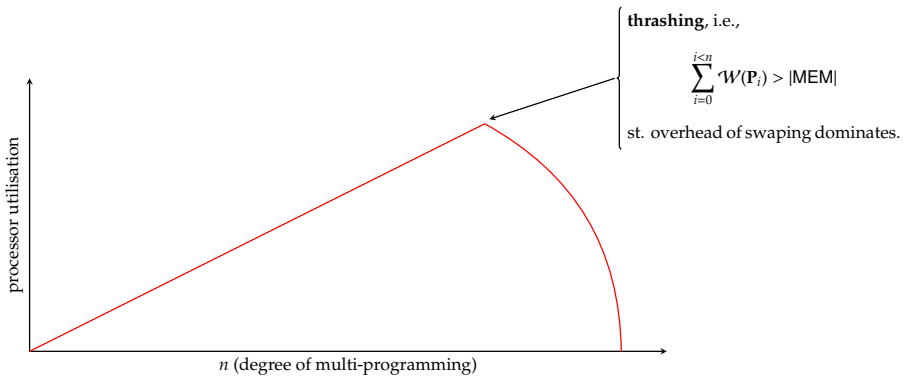


Notes:

- 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 [2], 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.

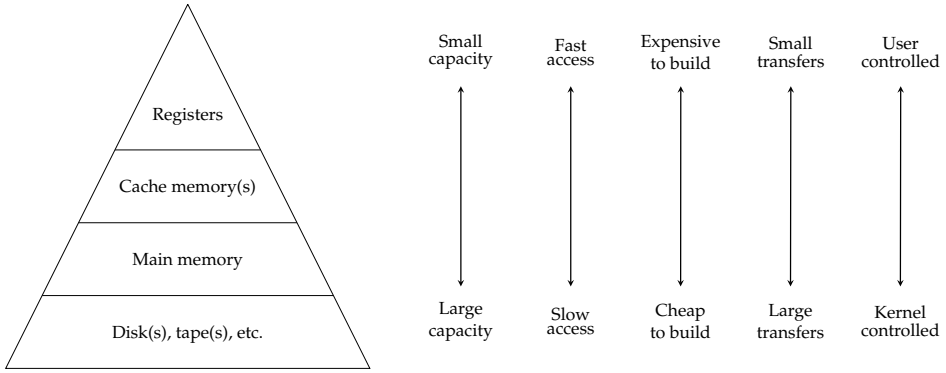
Notes:

- 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 [2], 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

Definition (memory hierarchy)



Notes:

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 that

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

and, in some cases, *full* memory virtualisation isn't required: *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 Anomoly),
  - techniques for pre-fetching or buffering pages,
  - swap space management.

  Some of these *are* covered in the recommended reading: see in particular [10, 4, 5].

References

[1] Wikipedia: Paging.
https://en.wikipedia.org/wiki/Paging.

[2] M. Gorman.
Chapter 13: Out of memory management.
In *Understanding the Linux Virtual Memory Manager* [3].
http://www.kernel.org/doc/gorman/.

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

[4] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 8: Memory management strategies.
In *Operating System Concepts* [6].

[5] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 9: Virtual-memory management.
In *Operating System Concepts* [6].

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

Notes:

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

[8] A. N. Sloss, D. Symes, and C. Wright.
Chapter 13: Memory protection units.
In *ARM System Developer's Guide: Designing and Optimizing System Software* [7].

[9] A. N. Sloss, D. Symes, and C. Wright.
Chapter 14: Memory management units.
In *ARM System Developer's Guide: Designing and Optimizing System Software* [7].

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

Notes: