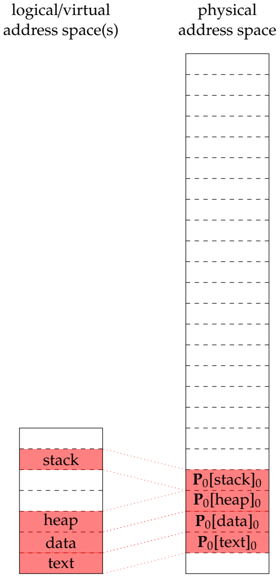
Continued from last lecture ...

Policy (1)

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

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

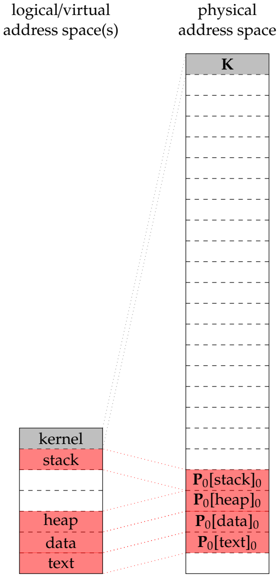


Policy (1)

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

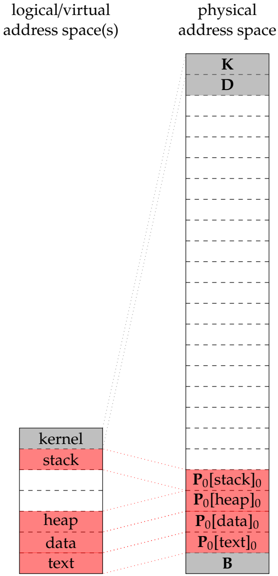
plus it suggests a more general ability to

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



Policy (1)

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

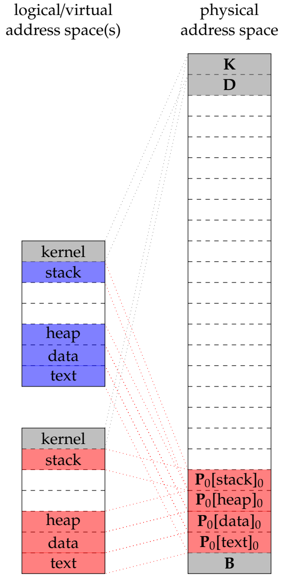


Policy (1)

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

  plus it suggests a more general ability to

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

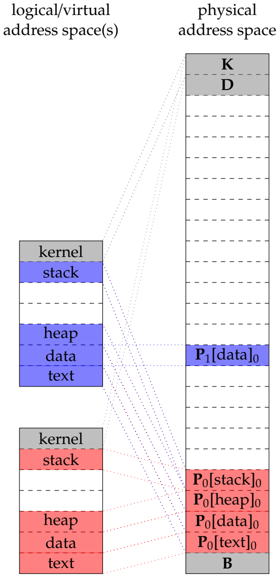


Policy (1)

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

  plus it suggests a more general ability to

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



Policy (1)

- Idea: implement

  demand paging ≃ "lazy swapping"

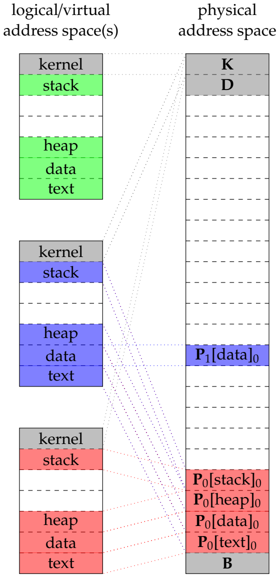
  i.e.,

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

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

  plus it suggests a more general ability to

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



Policy (1)

- **Idea**: implement

  demand paging $\simeq$ "lazy swapping"

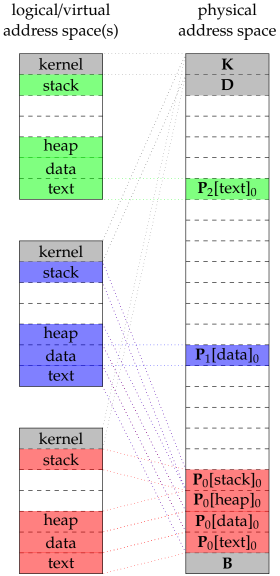
  i.e.,

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

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

  plus it suggests a more general ability to

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



Policy (1)

- Idea: implement

  demand paging ≃ "lazy swapping"

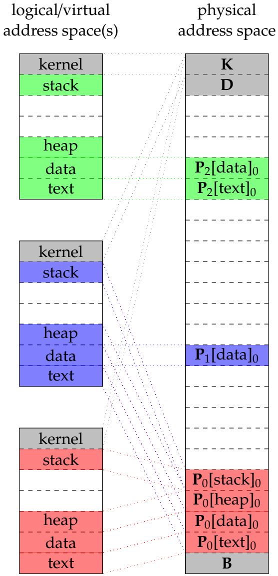
  i.e.,

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

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

  plus it suggests a more general ability to

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



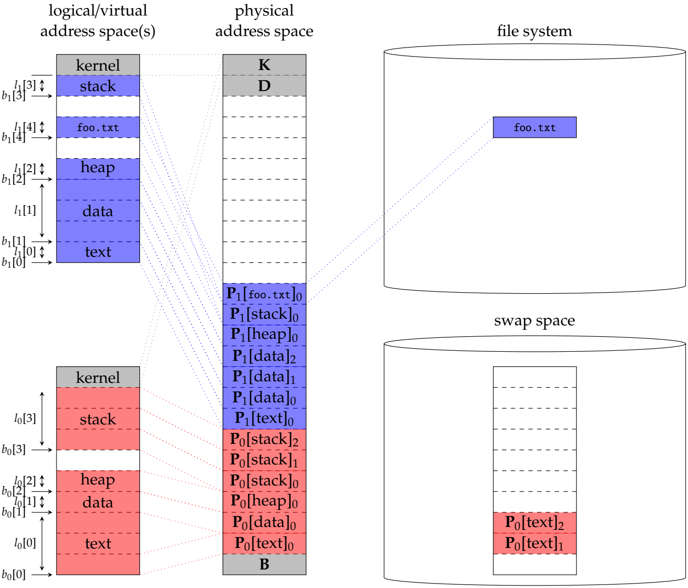
Implementation: demand paging (1)

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

  noting that

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

Implementation: demand paging (2)



Algorithm (**demand paging**)

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

		address (allocation table)	
		valid (allocated)	invalid (unallocated)
page (page table)	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.

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.

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 ...

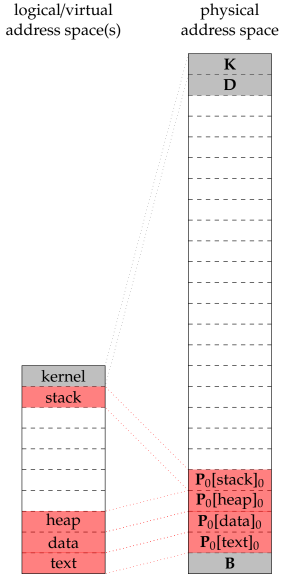
Implementation: demand paging (6) – page frame allocation

Algorithm (allocate page frame)

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

Implementation: demand paging (7) – page allocation

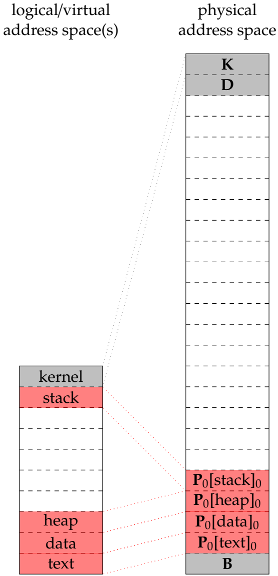
- Question: an initial allocation is fixed at load-time, but how are new pages allocated?



Implementation: demand paging (7) – page allocation

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

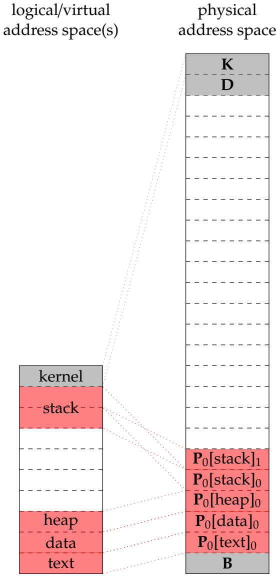
     and
  2. explicit or manual cases, e.g.,
     - enlargement of heap via `brk`, and
     - mapping a file via `mmap`.



Implementation: demand paging (7) – page allocation

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

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

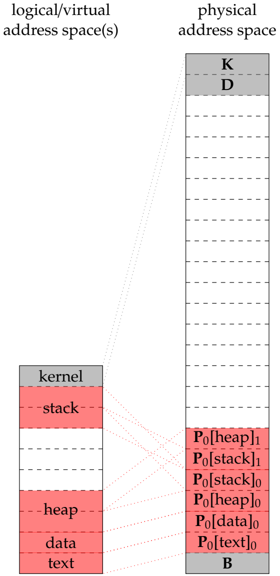


Daniel Page ⟨Daniel.Page@bristol.ac.uk⟩

Implementation: demand paging (7) – page allocation

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

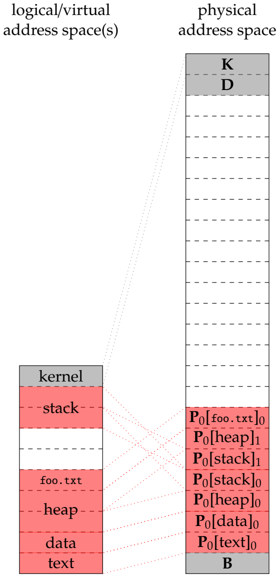
     and
  2. explicit or manual cases, e.g.,
     - enlargement of heap via `brk`, and
     - mapping a file via `mmap`.



Implementation: demand paging (7) – page allocation

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

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



Implementation: demand paging (8) – performance

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

Implementation: demand paging (8) – performance

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

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

so ideally we minimise such events.

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

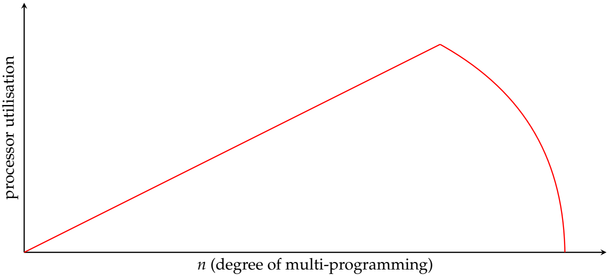
$$\begin{array}{rcr} \text{memory access} & \simeq & 100\text{ns} \\ \text{disk access} & \simeq & 1000000\text{ns} \end{array}$$

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

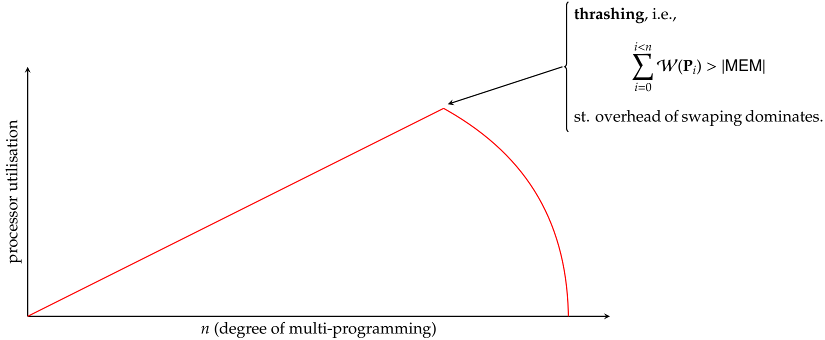
$$\begin{array}{lcl} \text{larger } n & \Rightarrow & \text{higher processor utilisation} \\ \text{larger } n & \Rightarrow & \text{higher contention wrt. page frames} \end{array}$$

  so, we find ...

Implementation: demand paging (9) – performance



Implementation: demand paging (9) – performance



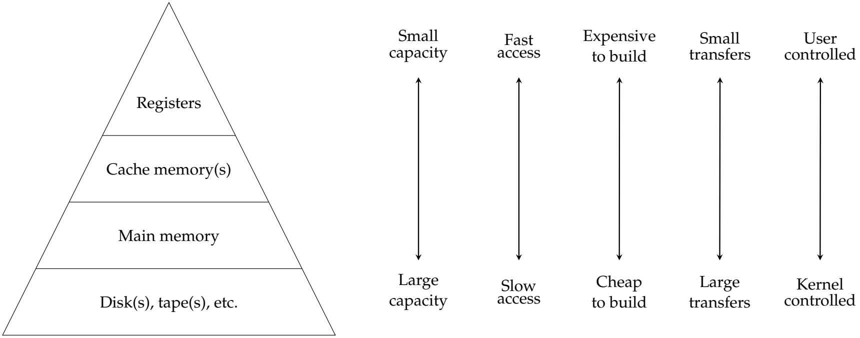
- Potential mitigations against thrashing include
  1. keep track of **page fault frequency**, noting that

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

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

Conclusions

Definition (**memory hierarchy**)



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.

References

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

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

[3] A. Silberschatz, P.B. Galvin, and G. Gagne.
Chapter 8: Memory management strategies.
In *Operating System Concepts* [5].

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

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

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

References

[7] A. N. Sloss, D. Symes, and C. Wright.
Chapter 13: Memory protection units.
In *ARM System Developer's Guide: Designing and Optimizing System Software* [6].

[8] A. N. Sloss, D. Symes, and C. Wright.
Chapter 14: Memory management units.
In *ARM System Developer's Guide: Designing and Optimizing System Software* [6].

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