Memory Virtualization:
Swapping

OSTEP Chapters 21+22:
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys.pdf
http://pages.cs.wisc.edu/~remzi/OSTEP/vm-beyondphys-policy.pdf

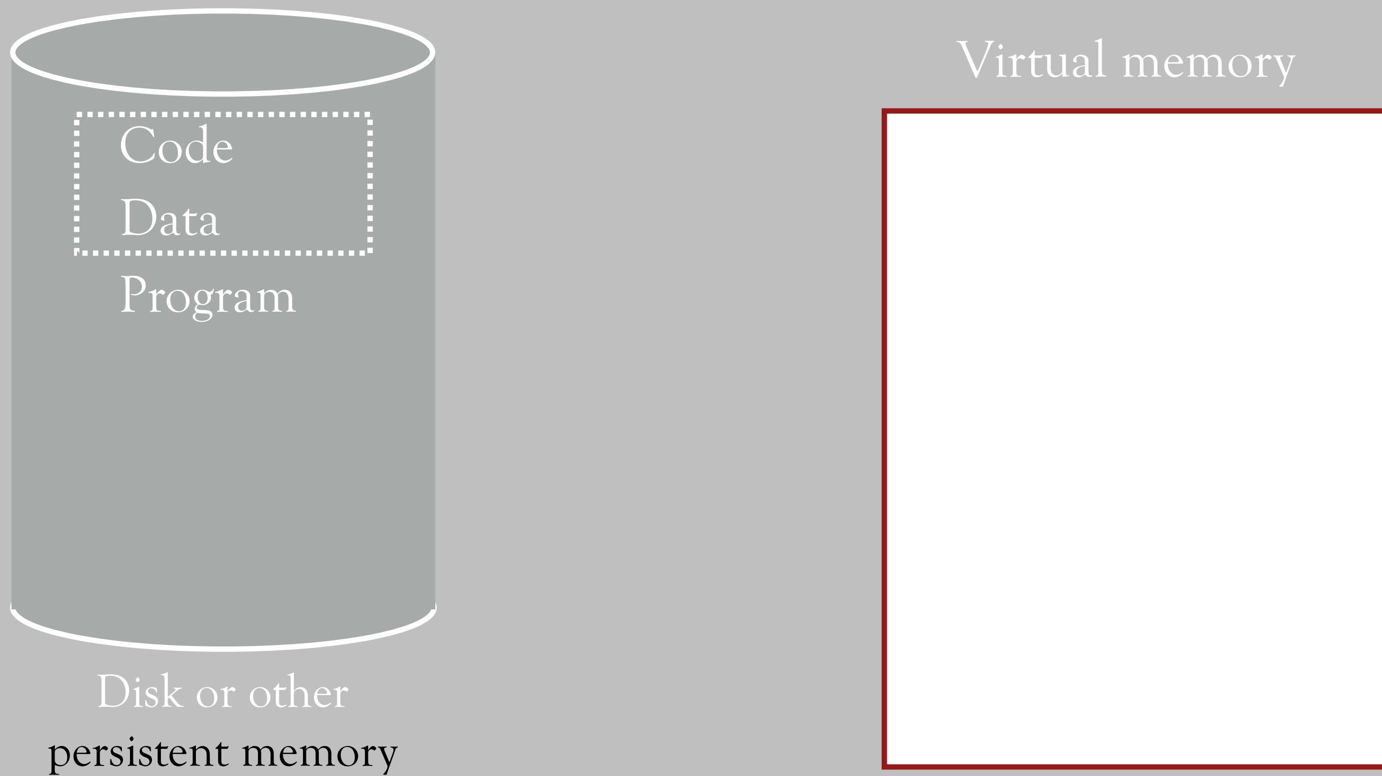
Jan Reineke
Universität des Saarlandes

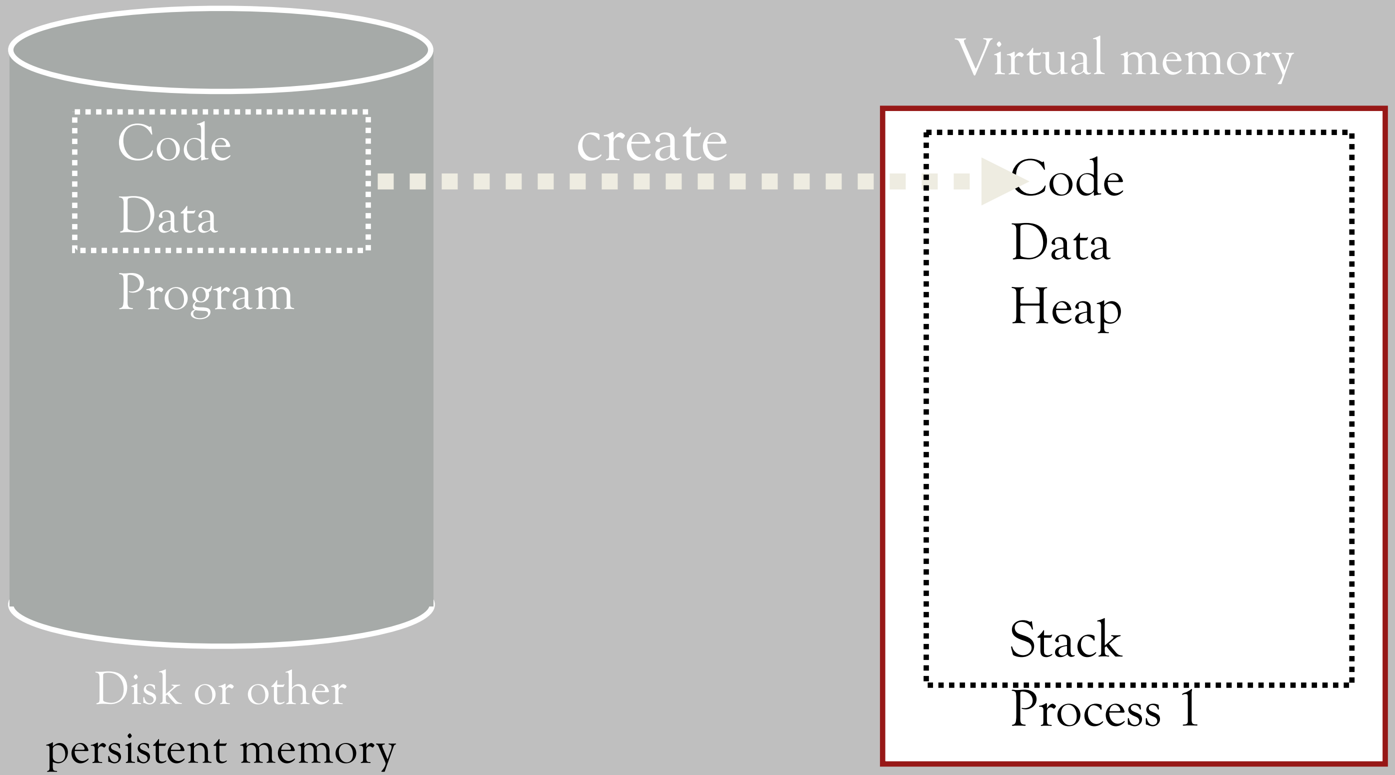
Motivation

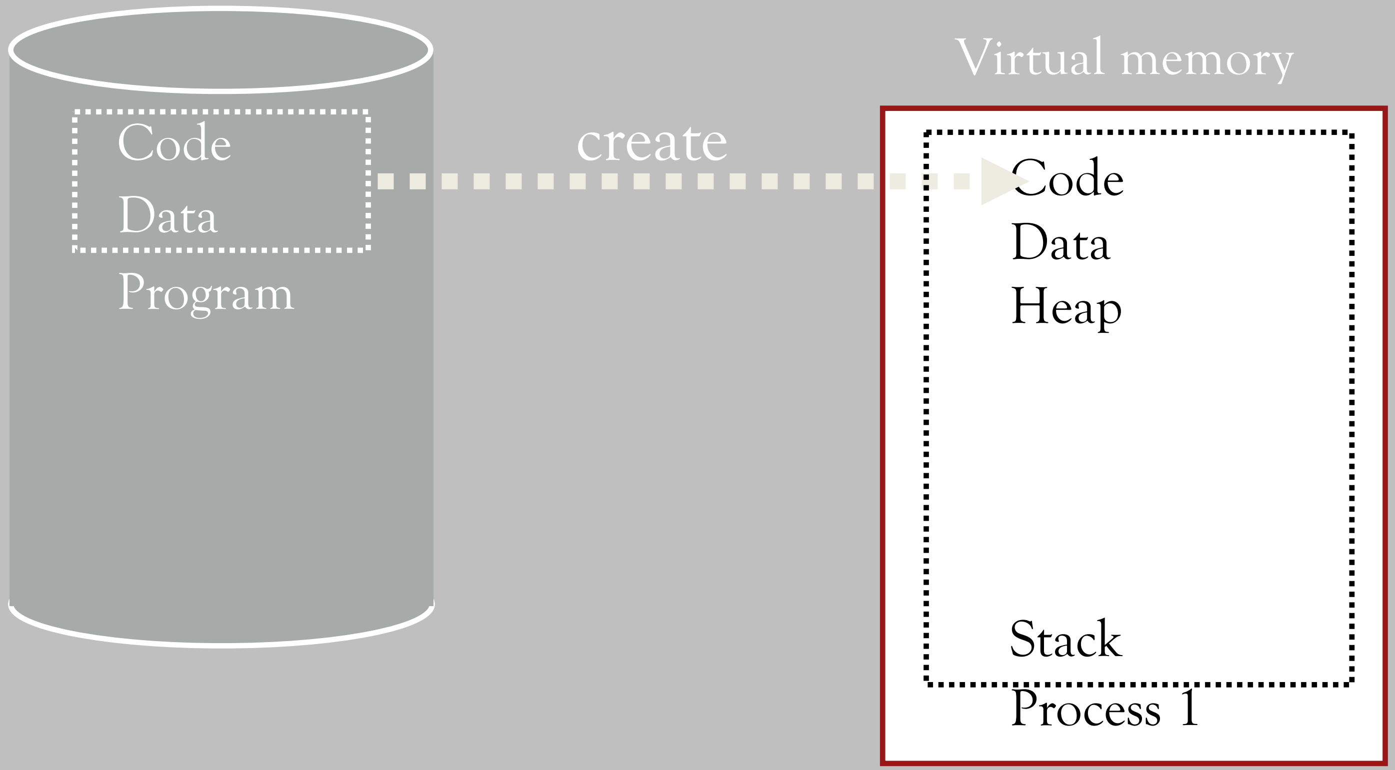
OS *goal*: Support processes when not enough physical memory:

- single process with very large address space
- multiple processes with combined address spaces

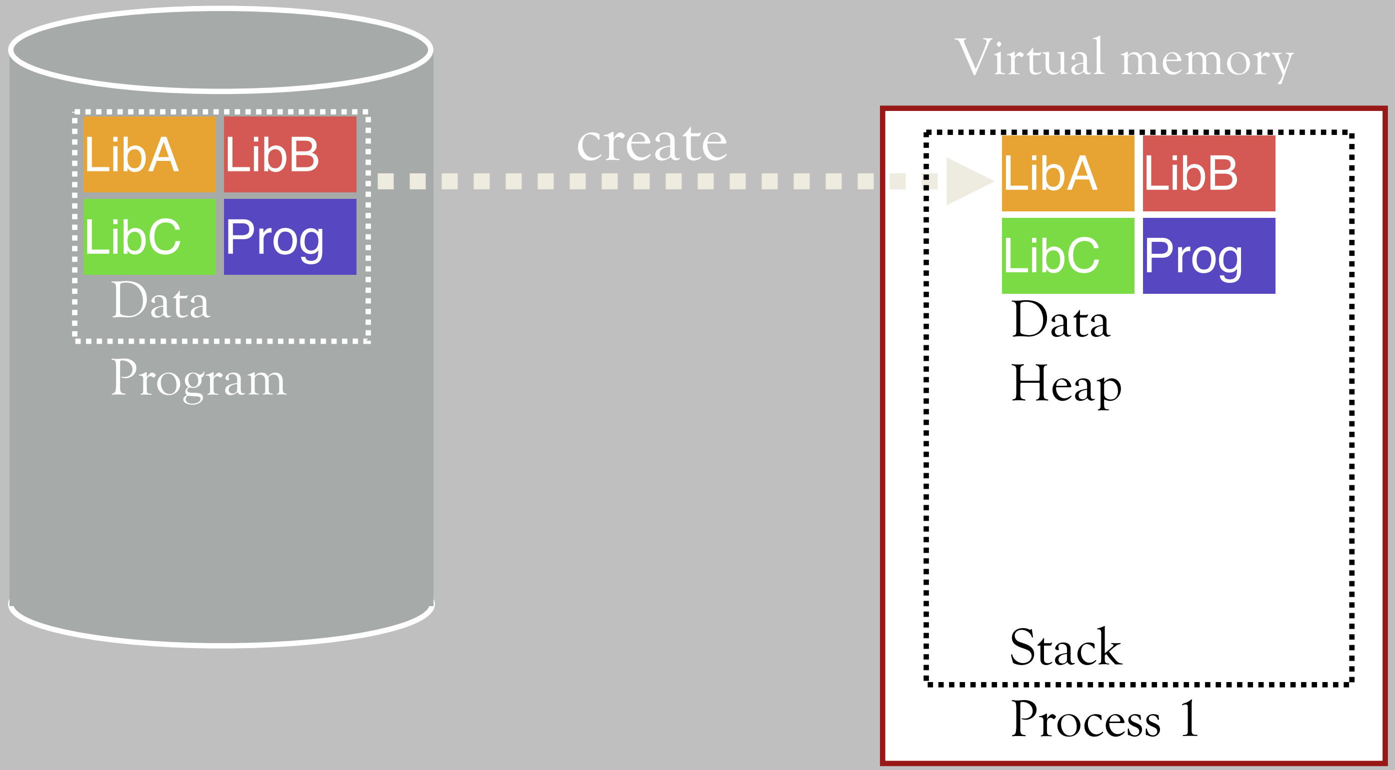
Programs should be **independent** of amount of physical memory


Code
Data
Program
Disk or other
persistent memory
Virtual memory


Code
Data
Program
Disk or other
persistent memory
create
Virtual memory
Code
Data
Heap
Stack
Process 1

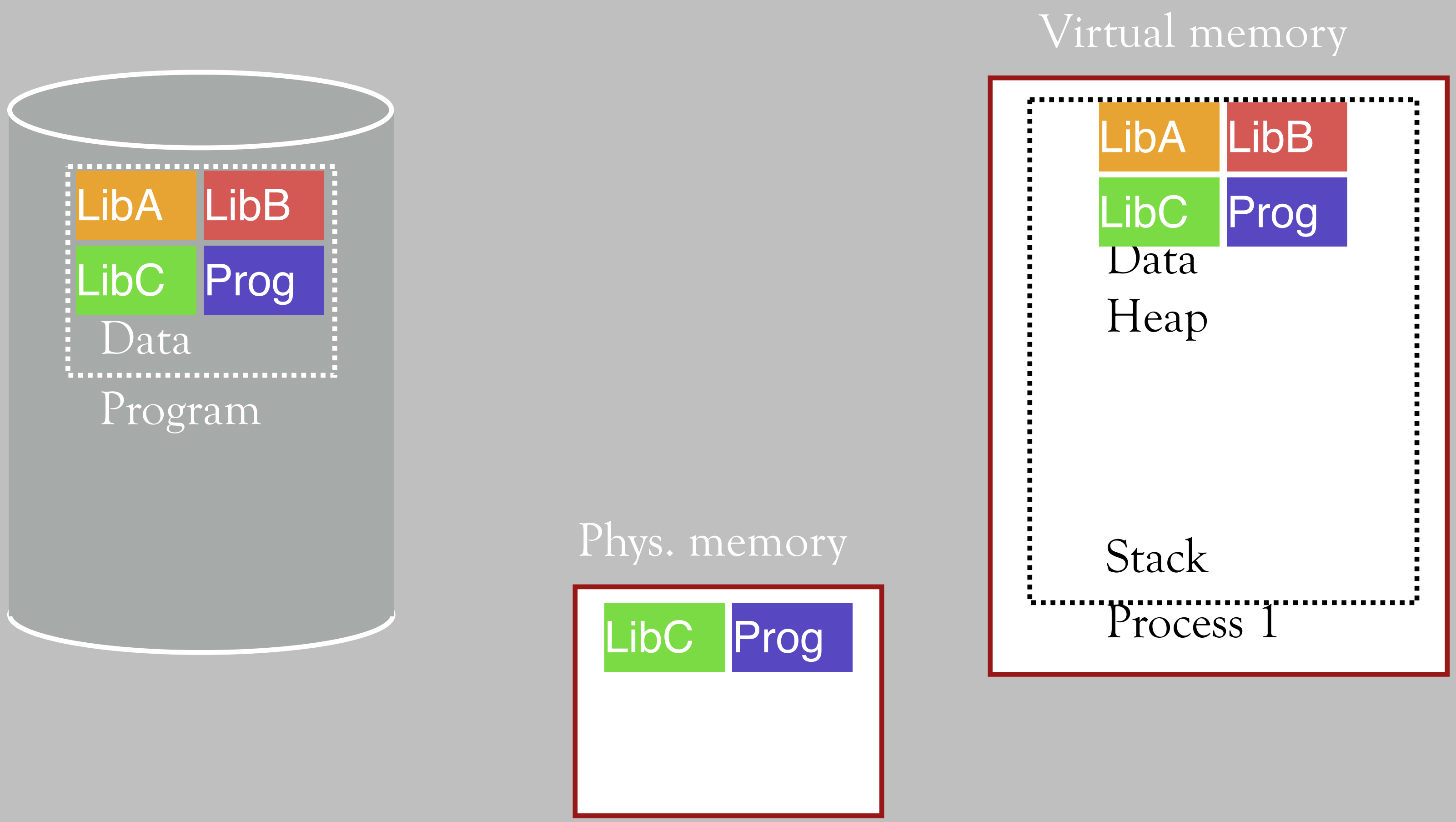


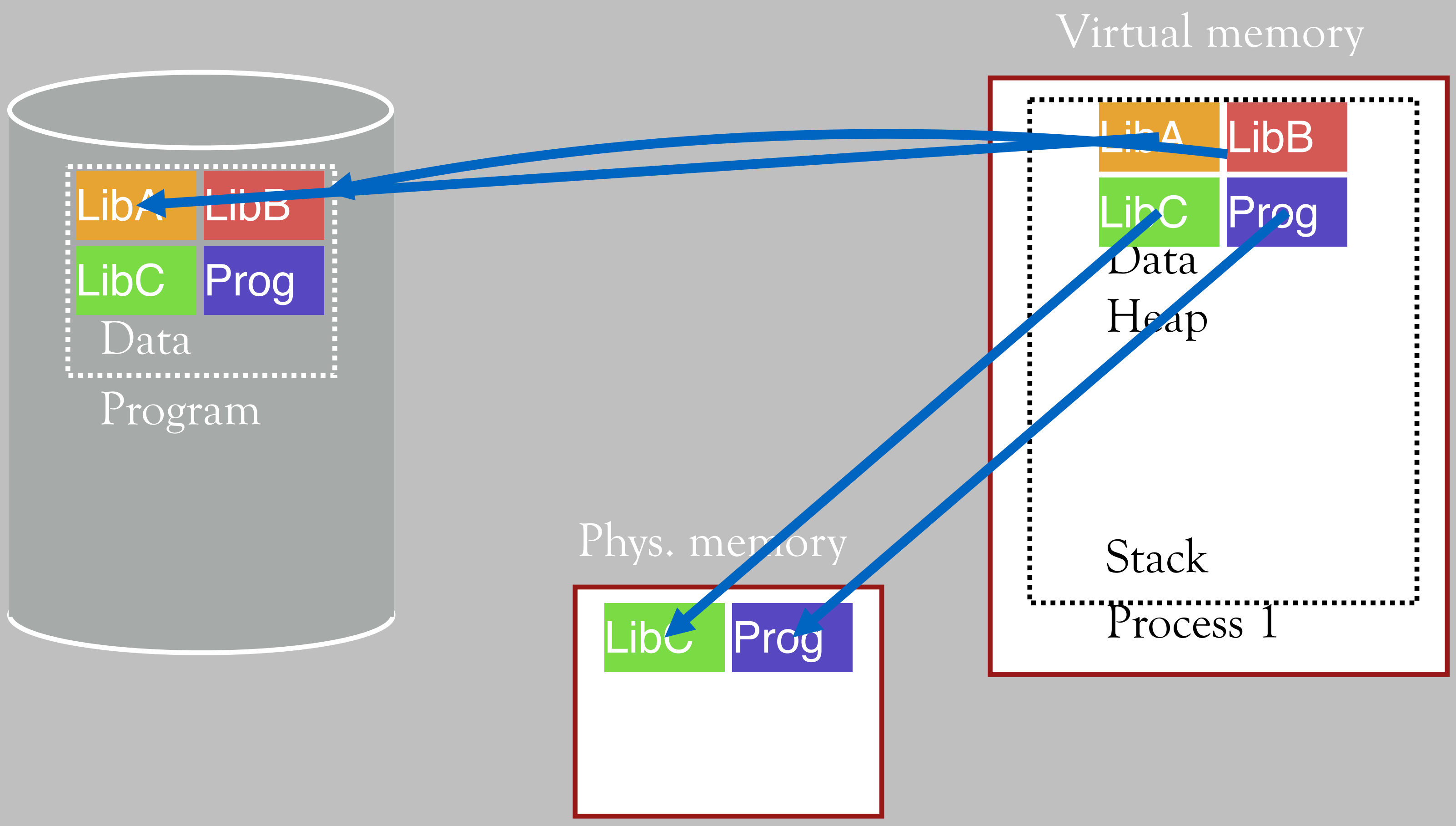
What all is in code?

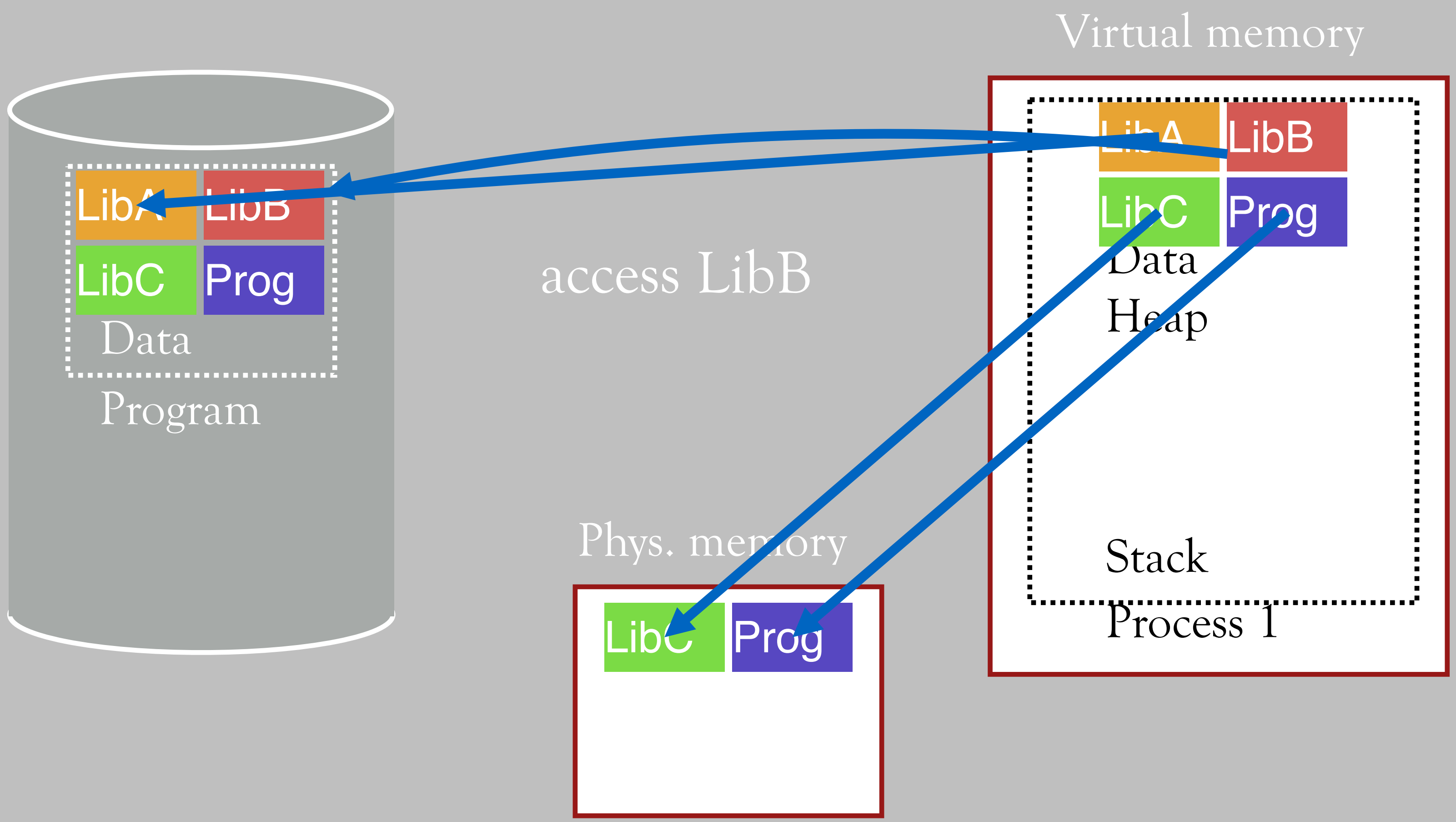


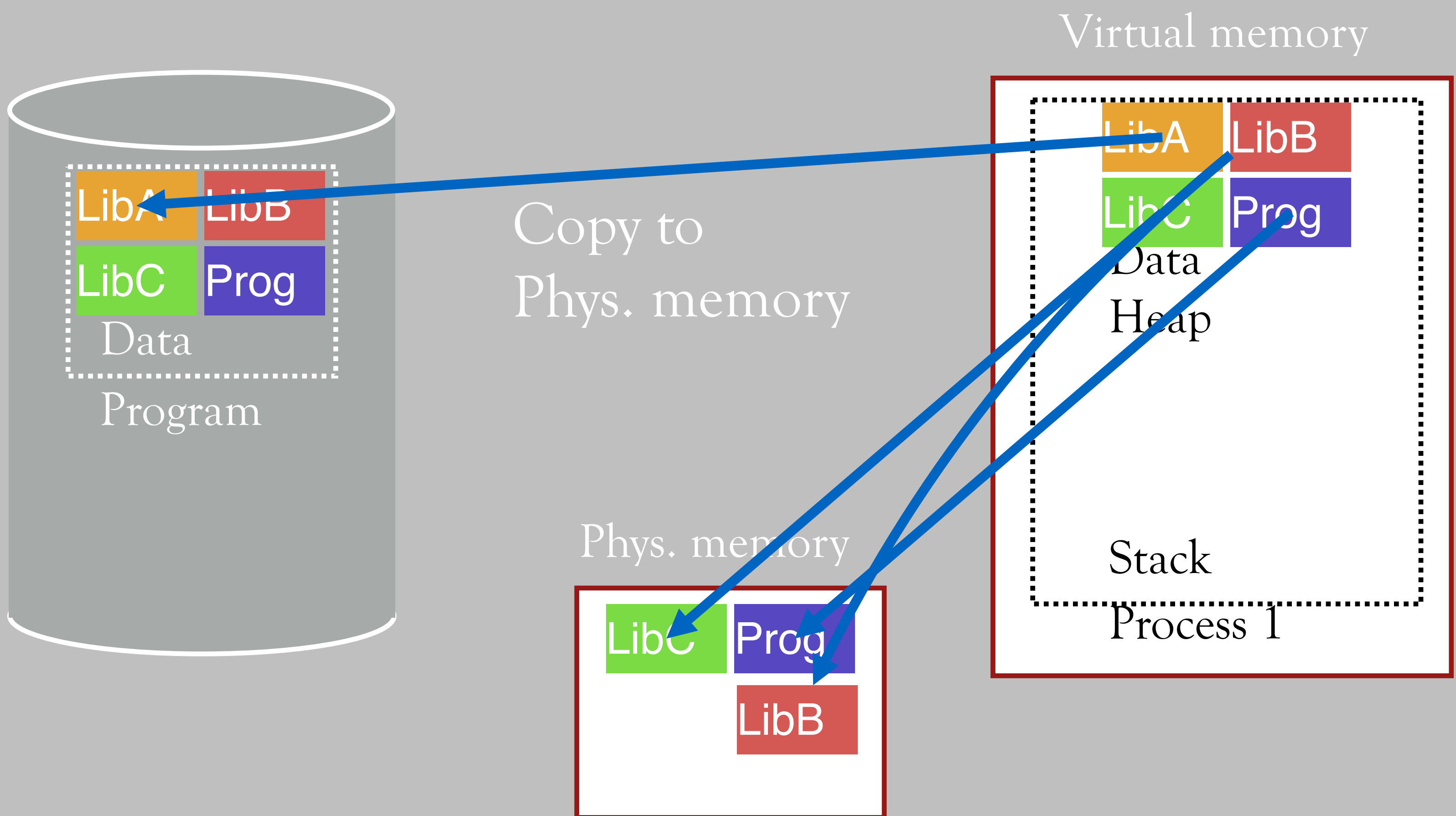
Many large libraries,
some of which are rarely/never used

How to avoid wasting **physical pages**
for rarely used **virtual pages**?


Virtual memory
LibA
LibB
LibC
Prog
Data
Program
Phys. memory
LibC
Prog
LibA
LibB
LibC
Prog
Data
Heap
Stack
Process 1


Virtual memory
LibA
LibB
LibC
Prog
Data
Heap
Stack
Process 1
Phys. memory
LibC
Prog
LibA
LibB
LibC
Prog
Data
Program


Virtual memory
LibA
LibB
LibC
Prog
Data
Program
access LibB
Data
Heap
Stack
Process 1
Phys. memory
LibC
Prog


Virtual memory
LibA
LibB
LibC
Prog
Data
Heap
Stack
Process 1
LibA
LibB
LibC
Prog
Data
Program
Copy to
Phys. memory
Phys. memory
LibC
Prog
LibB

Once more: Locality

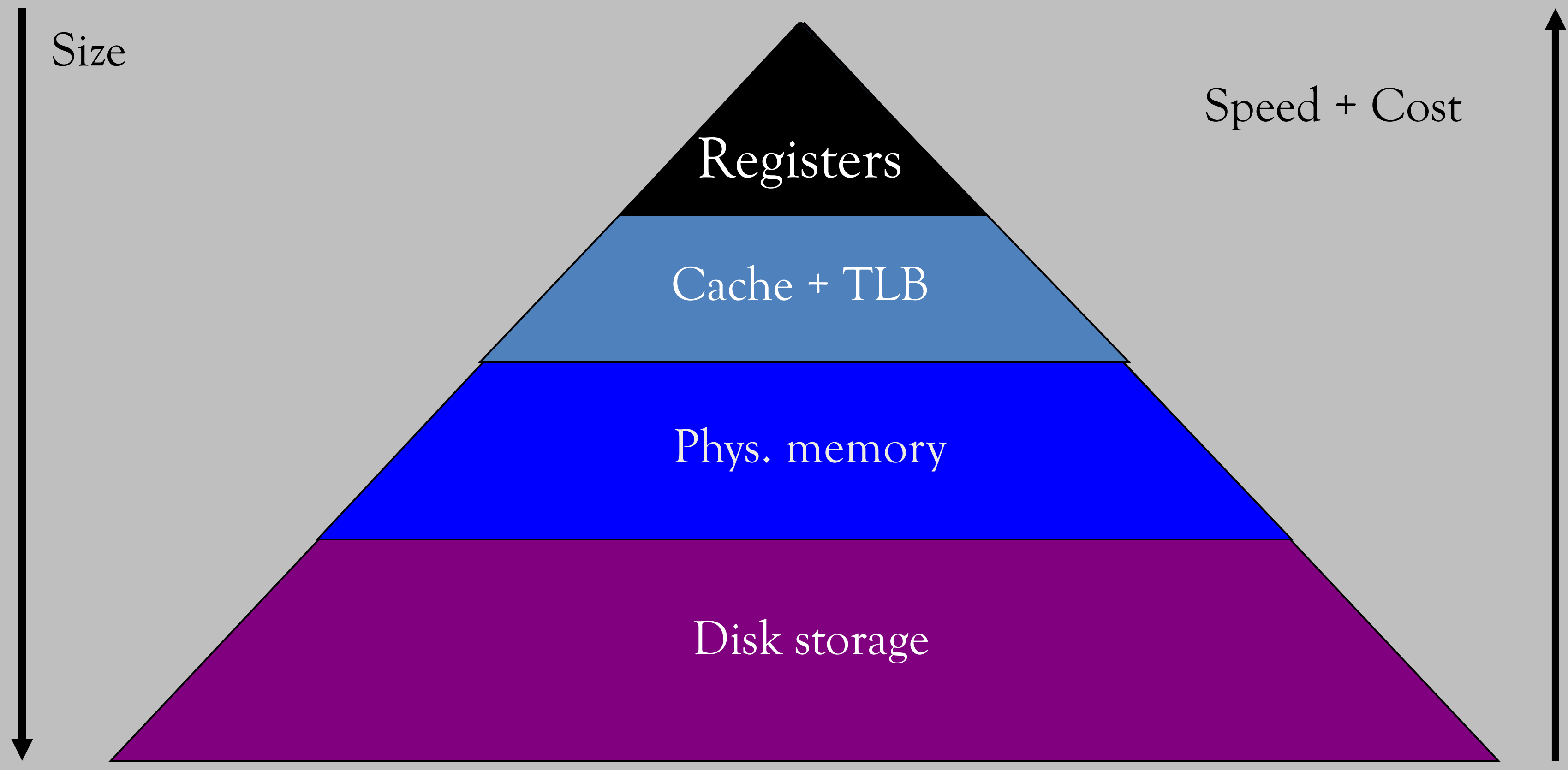
Leverage **locality** within processes:

- Spatial and temporal
- Processes spend majority of time in small portion of code
  - Estimate: 90% of time in 10% of code

Implication:

- Only small amount of address space must be resident in physical memory

Memory hierarchy



Virtual memory: Intuition

- *Goal*:
  - OS keeps **unreferenced** pages on disk
  - Process can run when not all pages are loaded into physical memory
  - OS and HW cooperate to provide illusion of large disk as fast as main memory
- *Requirements*:
  - **Mechanism** to manage location of each page: in memory or on disk
  - **Policy** to determine *which* pages to keep in memory

Virtual memory: Mechanisms

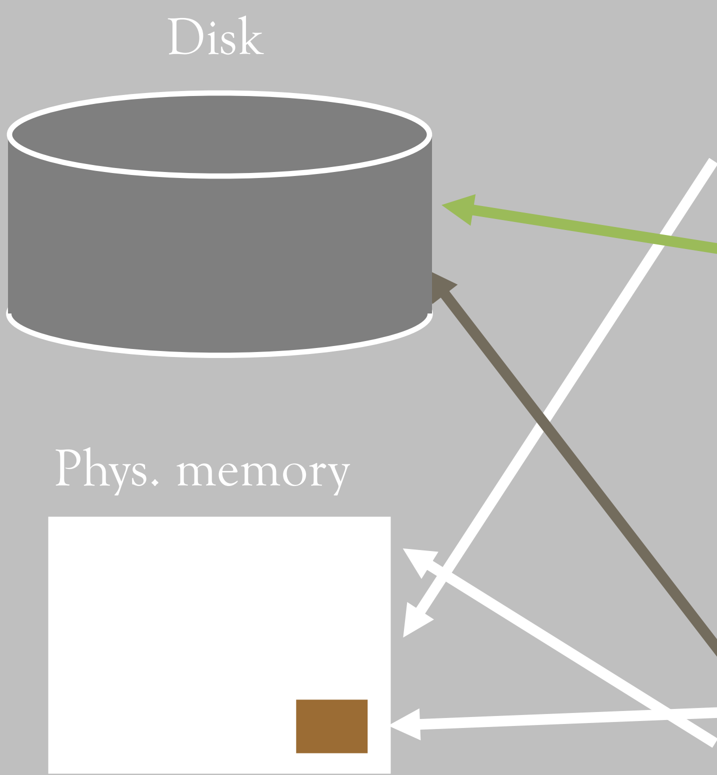
Each page in virtual address space maps to one of three locations:

- Phys. memory: Small, fast, expensive
- Disk: Large, slow, cheap
- nowhere (not allocated)

Extend page tables with an extra bit: `present`

- Permissions (`r/w`), `valid`, `present`
- Page in memory → `present = 1`
- Page on disk → `present = 0`
  - PTE points to block on disk
  - Causes trap into OS when page is referenced: "**page fault**"

Present Bit



PFN	valid	prot	present
10	1	r-x	1
-	0	-	-
23	1	rw-	0
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
-	0	-	-
28 / 16	1	rw-	0 / 1
4	1	rw-	1

What if we access VPN 0xb?

Virtual memory: Mechanisms

HW and OS cooperate to translate addresses:

1. Hardware checks TLB for virtual address
   - if TLB-Hit → address translation is done; page in physical memory
2. If **TLB miss**
   - HW (*or* OS) "walk" page tables
   - If `present = 1`, then page in physical memory, add entry in TLB
3. If **page fault** `(present = 0)`
   - HW generates exception (also "trap") → OS takes over
   - OS selects victim page *and*
     writes victim page out to disk if modified (add `dirty bit` to PTE)
   - OS reads referenced page from disk into memory
   - OS updates page table and sets `present := 1`
   - Process continues execution

What should scheduler do?

Mechanism: Precise interrupts

Page fault may occur in middle of instruction:

- At instruction fetch
- At load or store

Requires hardware support for **precise interrupts**:

- All instructions "before" interrupt generating instruction are completed; all others are discarded
- Possible difficulties?
  - Due to pipelining, later instructions may have already taken some effect → Needs to be reverted

Virtual memory: Policies

G*oal*: Minimize number of page faults

- Page faults require milliseconds to handle (reading from disk)
- Implication: OS has plenty of time to make good decision

OS has two decisions:

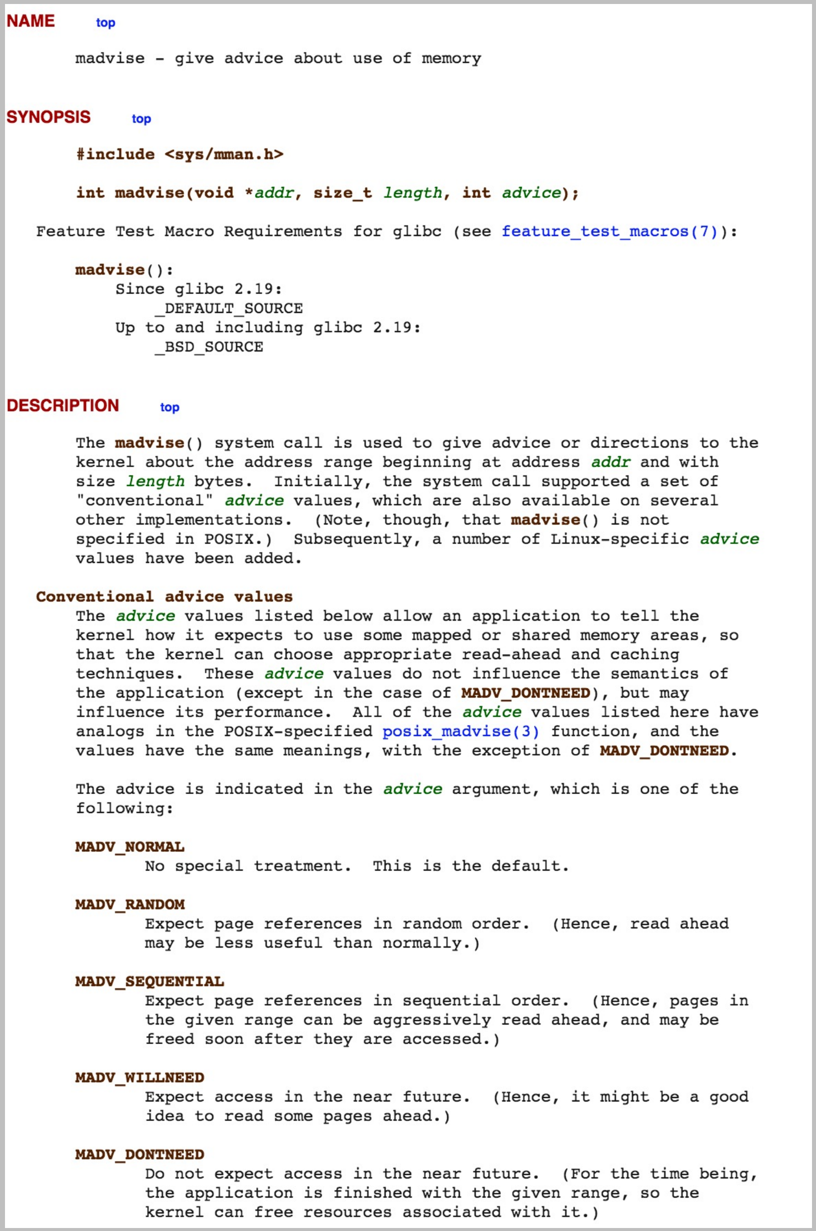
- **Page selection**:
  When should a page (or pages) on disk be **brought into** memory?
- **Page replacement**:
  Which resident page (or pages) in memory should be **thrown out** to disk?

Page selection

When should a page be brought from disk into memory?

1. **Demand paging**: Load pages only upon page faults
   - When process starts: No pages are loaded in memory
   - Disadvantage: Pay cost of page fault for every newly accessed page
2. **Prefetching**: Load page before referenced
   - OS predicts future accesses and brings pages into memory early
   - Works well for some access patterns (e.g. sequential)
   - Problems?
3. **Hints**: Program informs OS about future behavior
   - "need page soon", "don't need page anymore", "sequential access pattern"
   - *Example*: `madvise()` in Unix

madvise()



NAME top

madvise - give advice about use of memory

SYNOPSIS top

```
#include <sys/mman.h>

int madvise(void *addr, size_t length, int advice);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

```
madvise():
    Since glibc 2.19:
        _DEFAULT_SOURCE
    Up to and including glibc 2.19:
        _BSD_SOURCE
```

DESCRIPTION top

The **madvise**() system call is used to give advice or directions to the kernel about the address range beginning at address *addr* and with size *length* bytes. Initially, the system call supported a set of "conventional" *advice* values, which are also available on several other implementations. (Note, though, that **madvise**() is not specified in POSIX.) Subsequently, a number of Linux-specific *advice* values have been added.

Conventional advice values

The *advice* values listed below allow an application to tell the kernel how it expects to use some mapped or shared memory areas, so that the kernel can choose appropriate read-ahead and caching techniques. These *advice* values do not influence the semantics of the application (except in the case of **MADV_DONTNEED**), but may influence its performance. All of the *advice* values listed here have analogs in the POSIX-specified posix_madvise(3) function, and the values have the same meanings, with the exception of **MADV_DONTNEED**.

The advice is indicated in the *advice* argument, which is one of the following:

MADV_NORMAL
No special treatment. This is the default.

MADV_RANDOM
Expect page references in random order. (Hence, read ahead may be less useful than normally.)

MADV_SEQUENTIAL
Expect page references in sequential order. (Hence, pages in the given range can be aggressively read ahead, and may be freed soon after they are accessed.)

MADV_WILLNEED
Expect access in the near future. (Hence, it might be a good idea to read some pages ahead.)

MADV_DONTNEED
Do not expect access in the near future. (For the time being, the application is finished with the given range, so the kernel can free resources associated with it.)

Page replacement

Which page in memory should be selected as victim?

1. *OPT/BEL*: Optimal strategy, requires knowledge about the future
2. *LRU*: Replace page not used for longest time in past
3. *FIFO*: Replace page that has been in memory the longest
   – Advantage: easy to implement

Write page back to disk if it has been modified (`dirty = 1`)

LRU: Implementation alternatives

In Software:

- OS maintains list of pages, ordered by the time of their last access
- Upon page access: Move page to front of list
- "Victim selection": Select last page on list
- Trade off:
  - slow upon every memory access,
  - fast upon replacement.

Does that make sense?

→ Rather **not**, because (hopefully)
Number of memory accesses >> Number of replacements

LRU: Implementation alternatives

In Hardware:

- Store time of last access for each page
- Upon page access: Store current time in page table
- "Victim selection": Search page table for oldest timestamp

- Trade off:
  - relatively fast upon every memory access,
  - slow upon replacement

Better, but also not great.

LRU: Implementation alternatives

In practice: **approximate** LRU

- LRU approximates optimal replacement anyway, so why not approximate more
- *Goal*: Find "old" page,
  but not necessarily the oldest

Clock algorithm

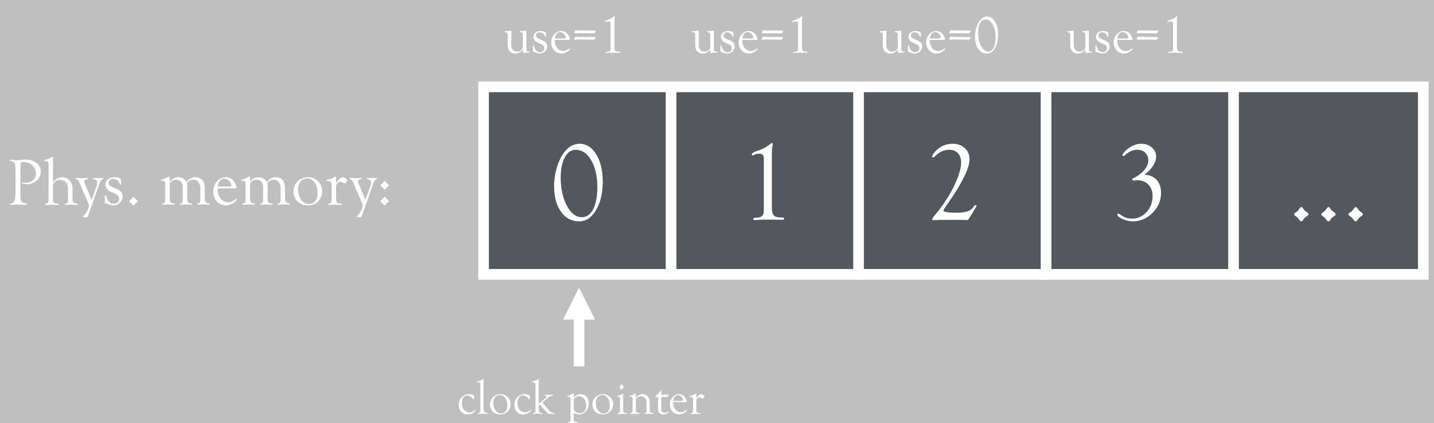
Hardware:

- Keep `use` bit for each page frame
- Upon page access: Set `use` bit to 1

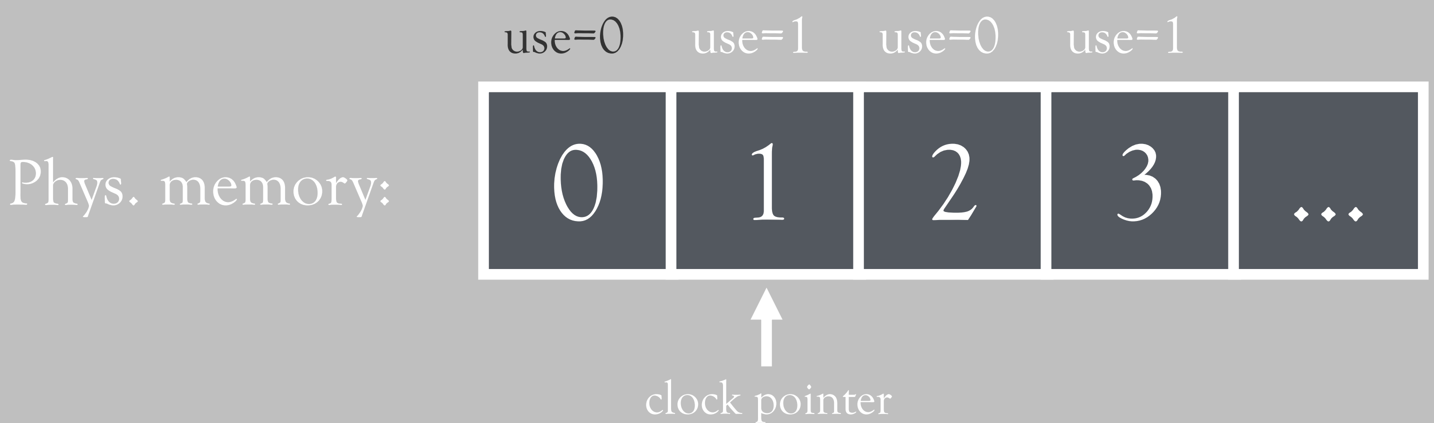
Operating system:

- Page replacement: Look for page with `use = 0`
- Implementation:
  - Keep pointer to last examined page frame
  - Traverse pages in circular buffer
  - Clear `use` bits upon traversal
  - Stop when find page with already cleared `use` bit; replace this page; increment pointer

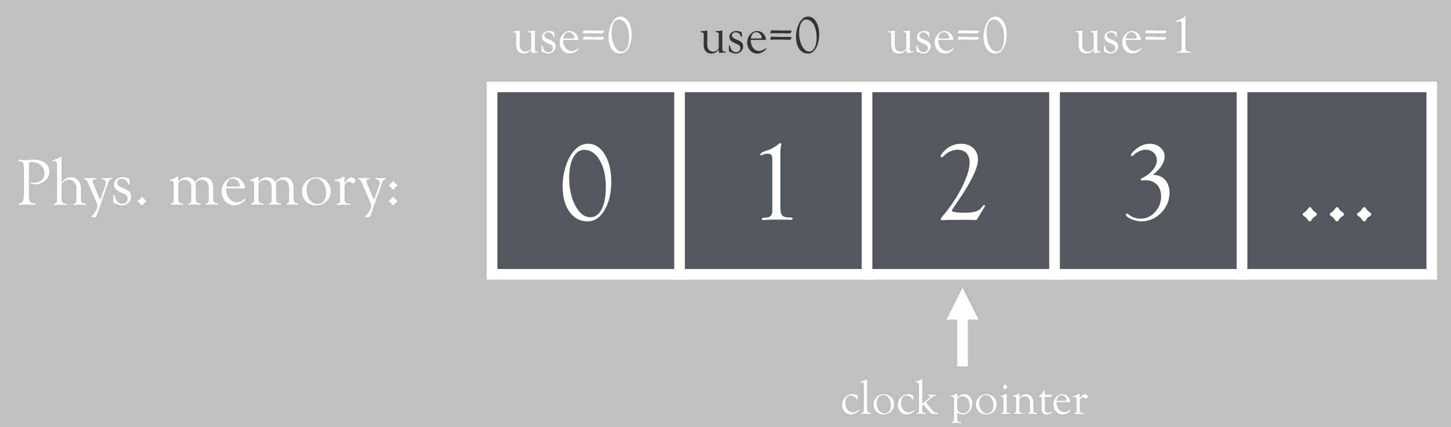
Clock: Look for a victim page



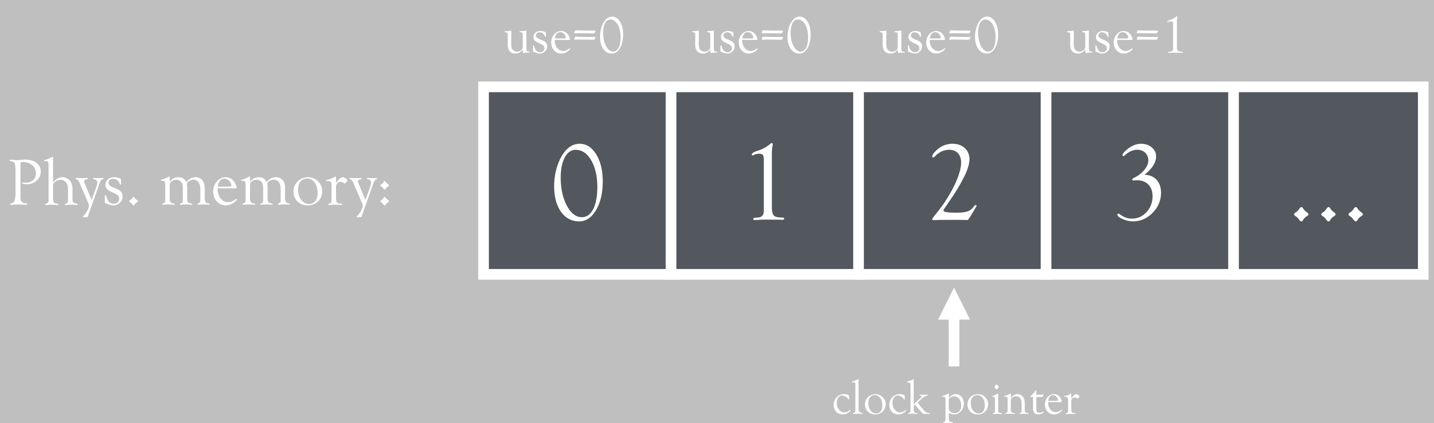
Clock: Look for a victim page



Clock: Look for a victim page



Clock: Look for a victim page

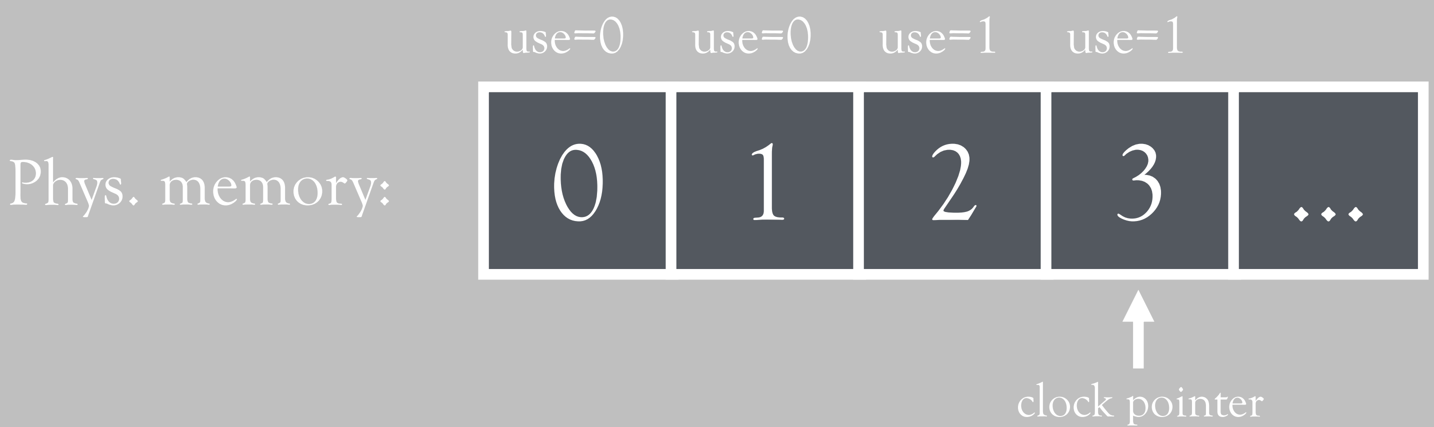


Evict **page 2** because it has **not** been used recently.

Clock: Look for a victim page

Evict **page 2** because it has **not** been used recently.

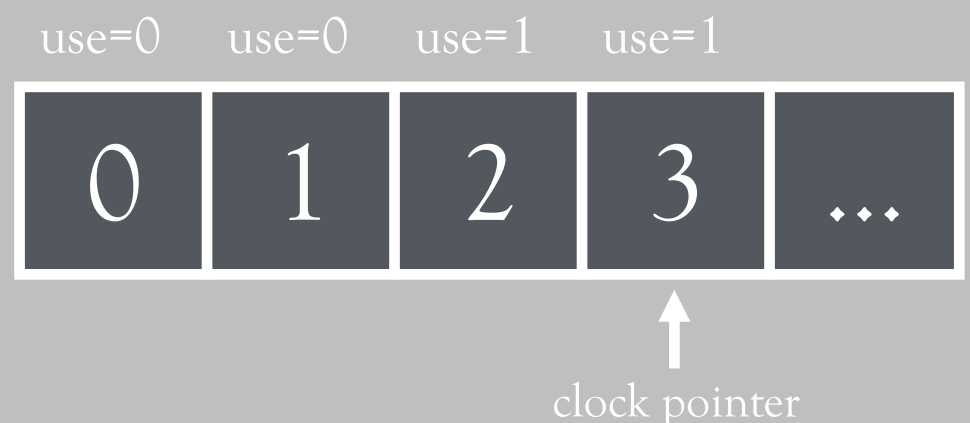
Clock: Look for a victim page



Evict **page 2** because it has **not** been used recently.

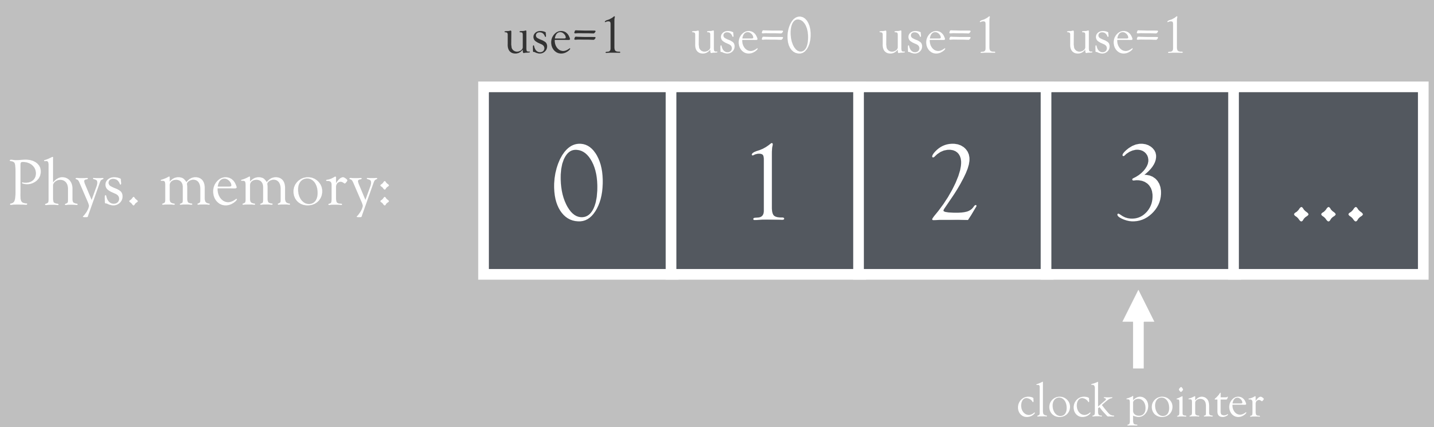
Clock: Look for a victim page

Phys. memory:



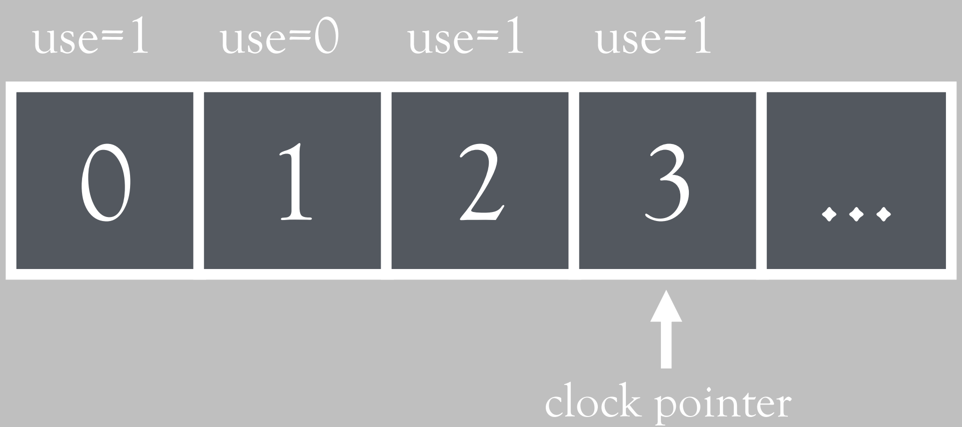
Page 0 is accessed.

Clock: Look for a victim page

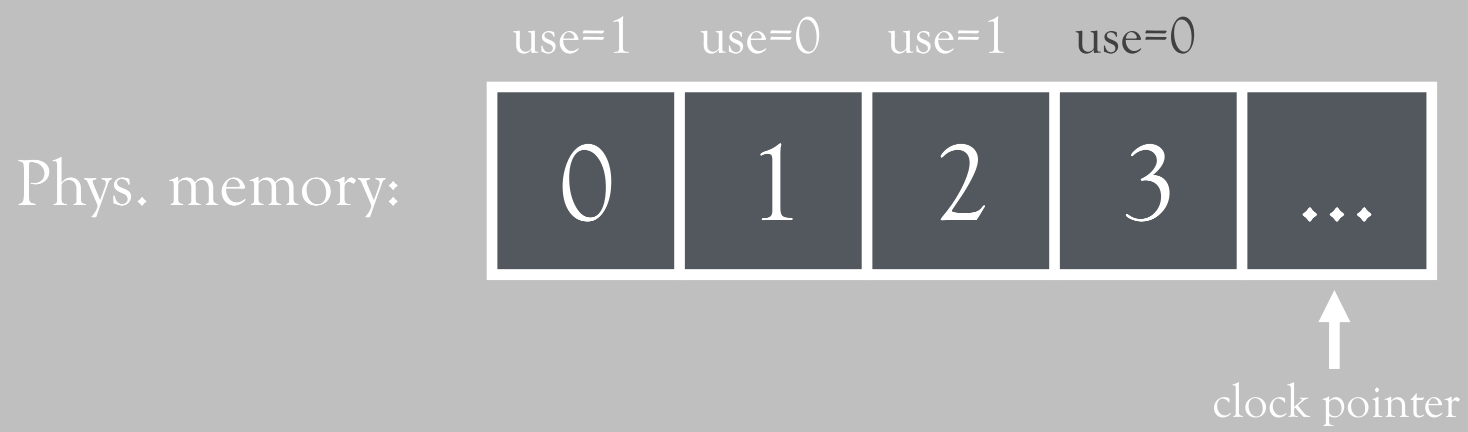


Clock: Look for a victim page

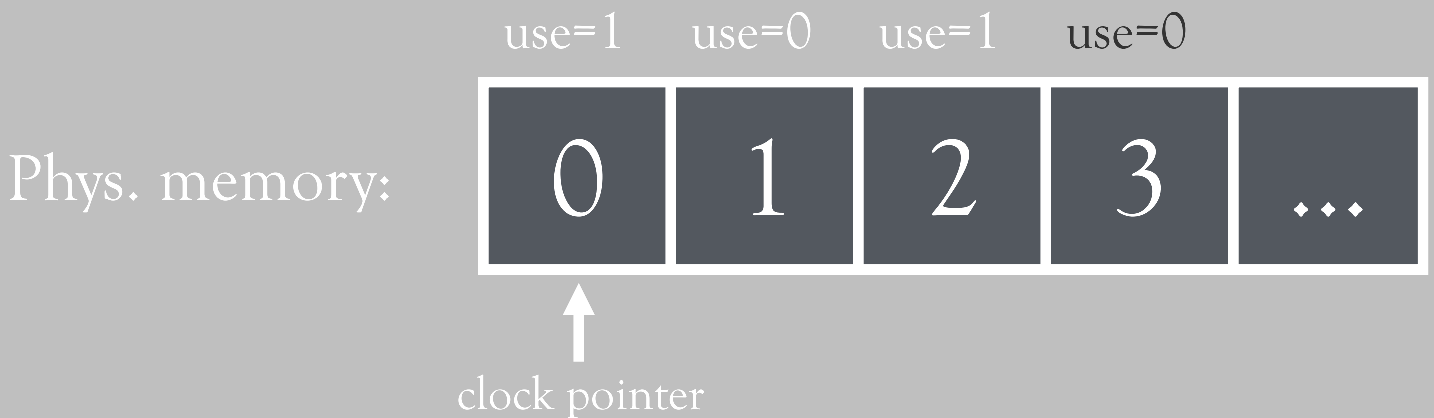
Phys. memory:



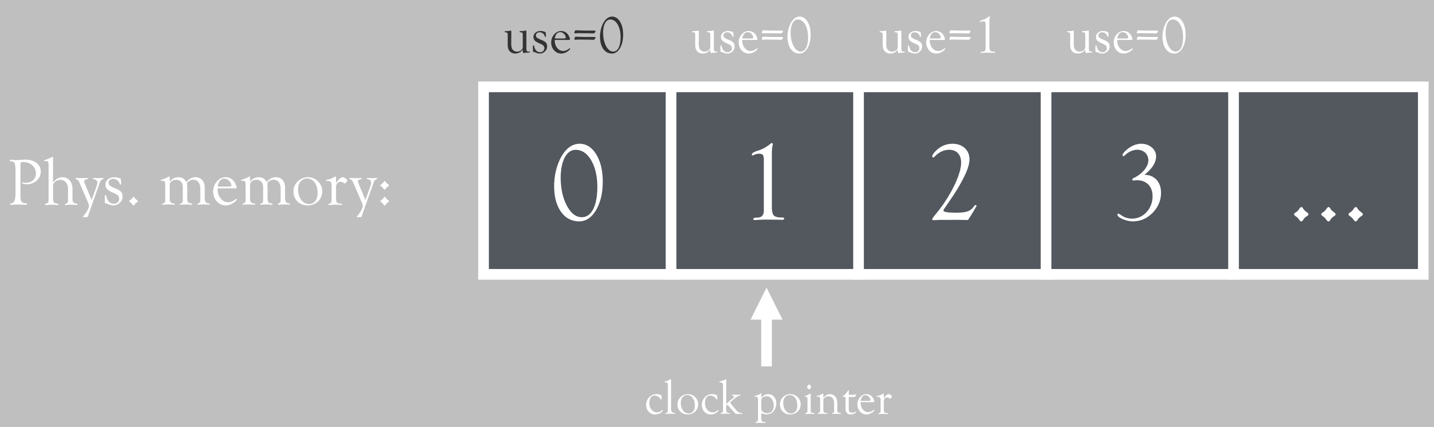
Clock: Look for a victim page



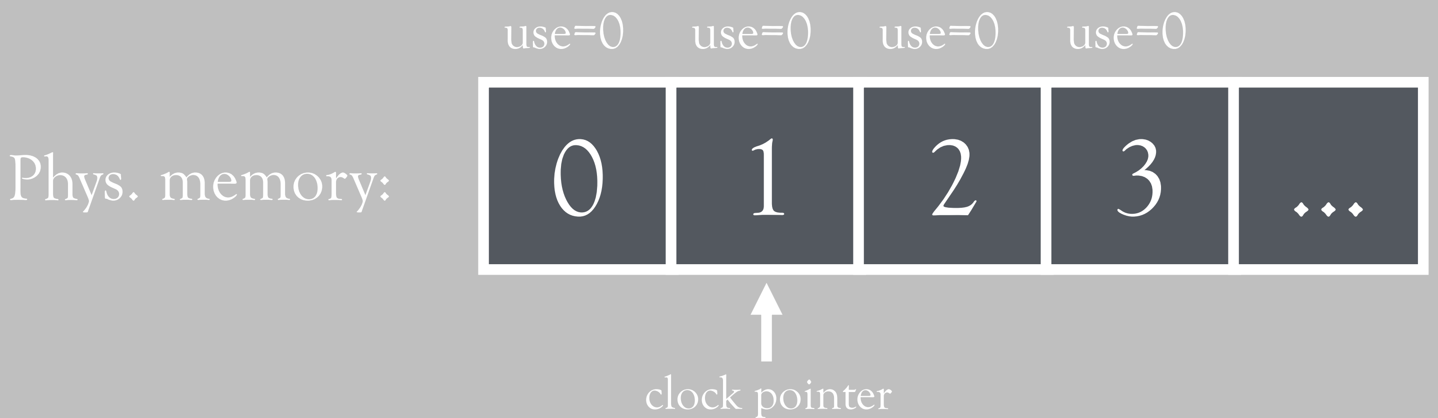
Clock: Look for a victim page



Clock: Look for a victim page



Clock: Look for a victim page



Evict **page 1** because it has **not** been used recently.

Summary

Processes can run when
sum of virtual address spaces > amount of physical memory

Mechanism:

- Extend page table entry with `present` bit
- OS handles page faults by reading in desired page from disk

Policy:

- Page selection: demand paging, prefetching, hints
- Page replacement: Clock as cheap approximation of LRU