Operating Systems

Memory over provisioning: mechanisms and policies

Debadatta Mishra, CSE, IITK

What is memory over provisioning?

- ➔ To commit/promise more memory than physical memory capacity
- ➔ Example:

  DRAM capacity = 8GB

  32 Applications, max. memory requirement 1GB each

  All applications promised maximum 4GB by the OS

Requirements

- Transparency
  - Design should be application transparent
  - View of memory related system call APIs remain unchanged
  - Virtual memory helps, how?

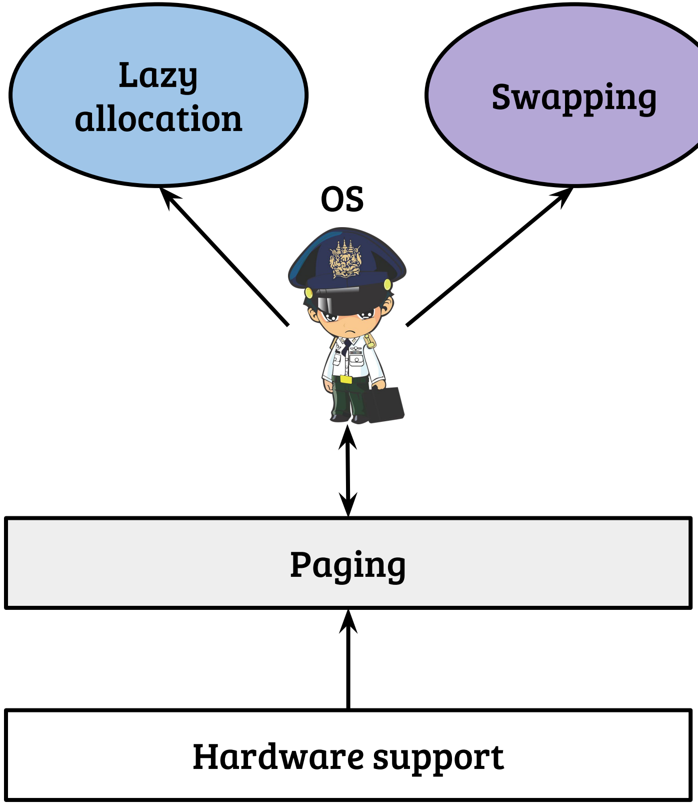
Requirements

- ➔ Transparency
  - ◆ Design should be application transparent
  - ◆ View of memory and the system call APIs remain unchanged
  - ◆ Virtual memory helps, how?
- ➔ Efficiency
  - ◆ Memory operations should be as efficient as non-overcommitted scenarios
  - ◆ The OS should manage the physical memory efficiently

Why overcommit?

- Applications are greedy!
  - Allocate long before they use
  - Overestimate memory requirements
  - Always reserve, even when not using
- Memory is a scarce resource, even today!
  - Memory footprint of applications are ever increasing
  - Multi-programming is a must, even for dedicated servers
  - OS should be prepared for worst case!

Memory overcommitment with paging



- ➔ Lazy allocation
  - ◆ Allocate VA on user request, allocate PA on first access
- ➔ Swapping
  - ◆ Physical memory full →backup physical memory to disk
  - ◆ Policy: when and what?
- ➔ Hardware support
  - ◆ Differentiate between valid, invalid and swapped out VA to PA mapping
  - ◆ OS trigger to handle invalid mapping

Hardware support

HW: If (P == 0)

Raise (PageFault);

Hardware support

Page table entry

HW: If (P == 0)

Raise (PageFault);

OS

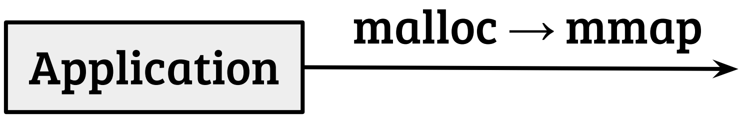
```
HandlePageFault( unsigned long vaddr )
{
    if ( vaddr belongs to current context){
        pfn = allocate_pfn ( );
        install_pt_mapping(vaddr, pfn);
    }
   else{
         inject_user_error (SEGFAULT);
   }
}
```

Lazy allocation: example

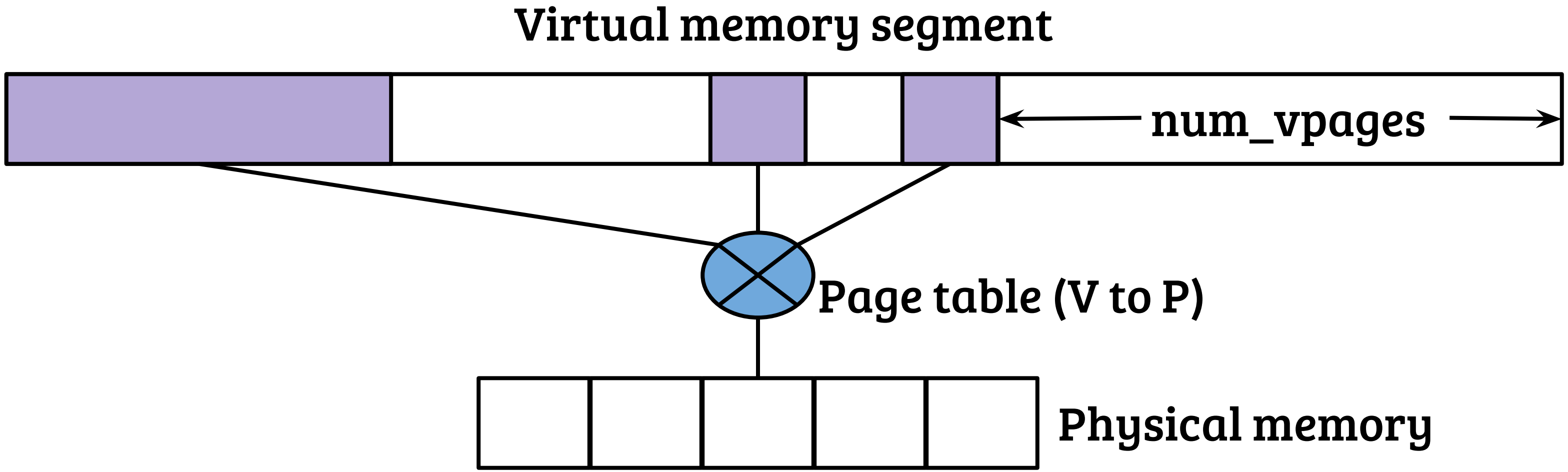
Application $\xrightarrow{\text{malloc} \rightarrow \text{mmap}}$

```
Syscall_mmap ( num_vpages,  ac_flags )
{
  vm_segment *vs = find_segment(ac_flags);
  expand_vm_segment (vs, num_vpages);
}
```

Lazy allocation: example




```
Syscall_mmap ( num_vpages,  ac_flags )
{
  vm_segment *vs = find_segment(ac_flags);
  expand_vm_segment (vs, num_vpages);
}
```

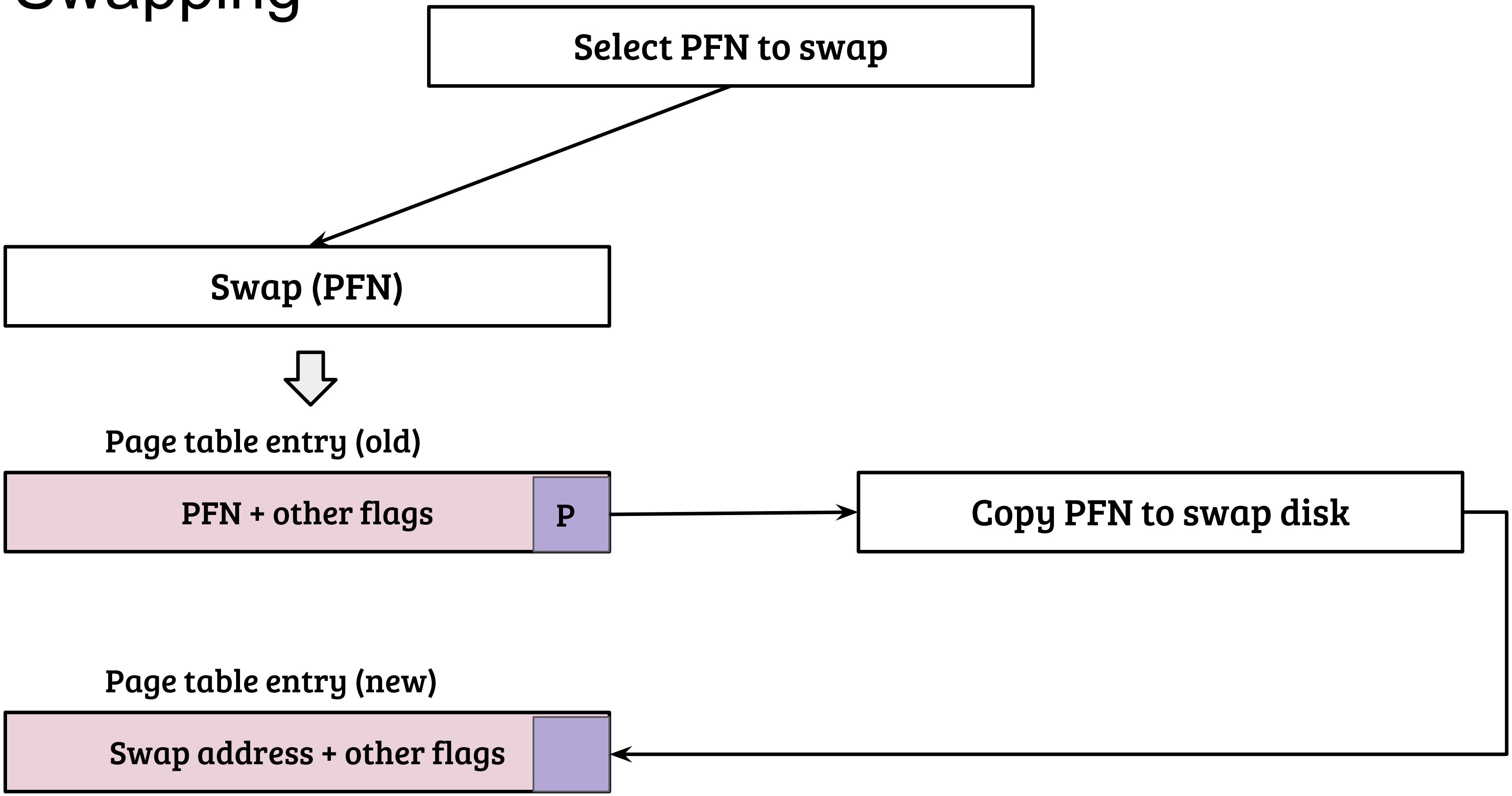


Swapping

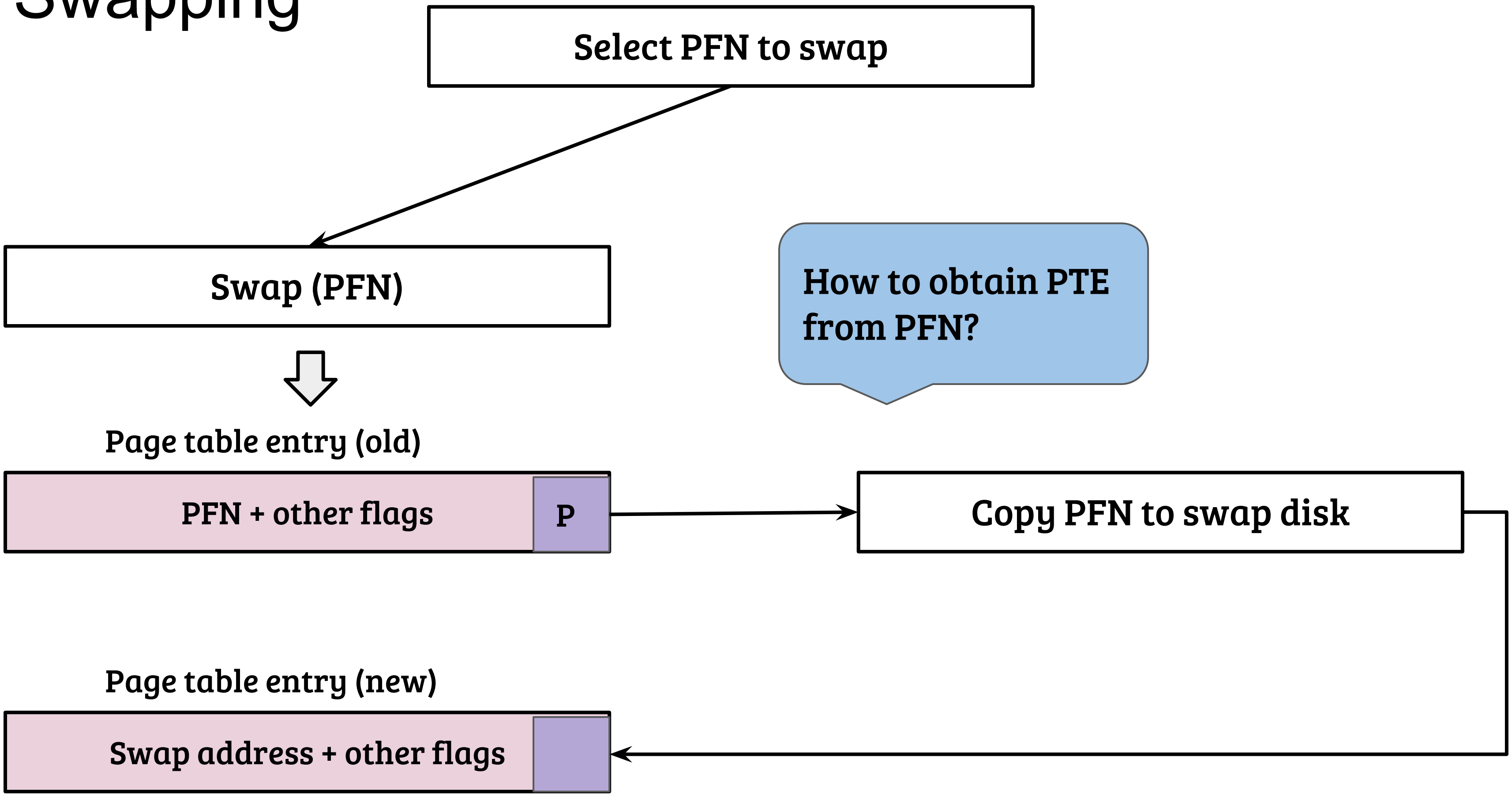
```
pfn_t allocate_pfn ( )
{
    if( isempty(pfn_free_list)){
        do_swap( );
    }
    return  dequeue (pfn_free_list);
}
```

- ➔ Swapping can be invoked earlier also
    - Example: Low free memory threshold violated
- ➔ Number of pages swapped can be more than one
- ➔ Swap implementation can be on a separate context!

Swapping

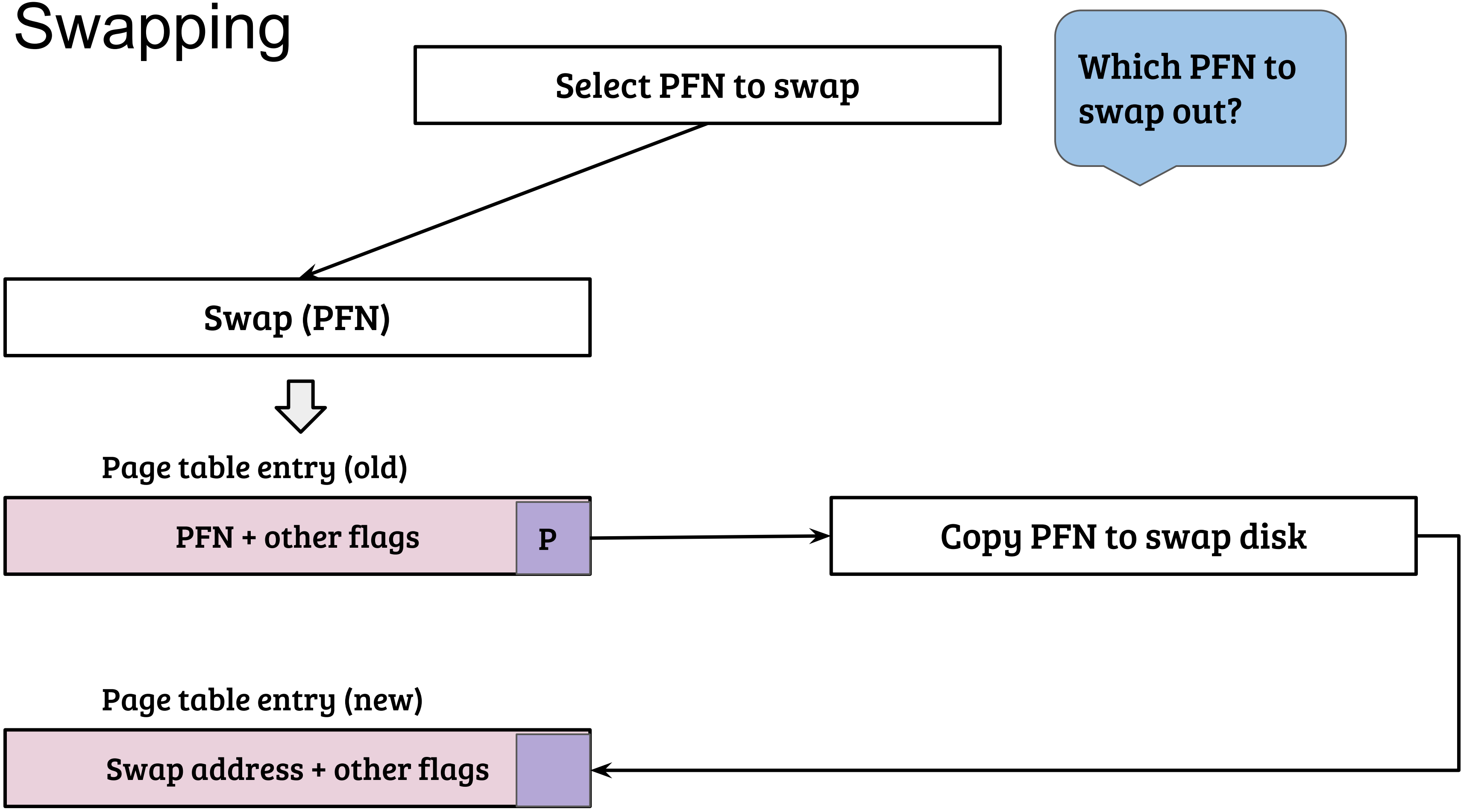

Select PFN to swap
Swap (PFN)
Page table entry (old)
PFN + other flags
P
Copy PFN to swap disk
Page table entry (new)
Swap address + other flags

Swapping

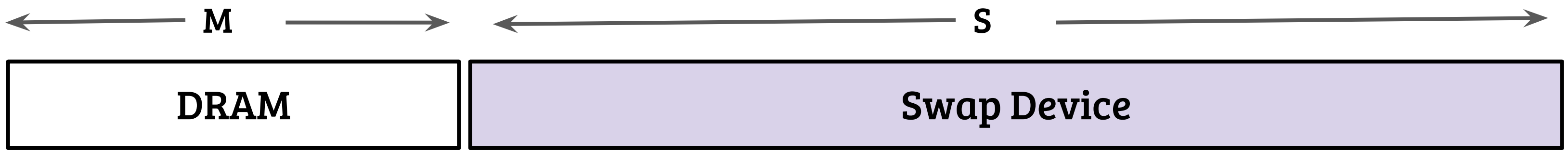


Select PFN to swap
Swap (PFN)
How to obtain PTE from PFN?
Page table entry (old)
PFN + other flags
P
Copy PFN to swap disk
Page table entry (new)
Swap address + other flags

Swapping


Select PFN to swap
Which PFN to swap out?
Swap (PFN)
Page table entry (old)
PFN + other flags
P
Copy PFN to swap disk
Page table entry (new)
Swap address + other flags

Page replacement



- Why needed?
  - Combined application memory usage > DRAM capacity
  - Instruction operands can not be disk addresses
- Solution approach
  - Keep “important” data in memory, rest in swap device
  - Anything not in RAM accessed → select a page in RAM to swap out
  - Qn: Which page to replace?

Page replacement

- ➔ Objective
  - ◆ Minimize #of page faults
- ➔ Model this problem with three parameters
  - ◆ A given sequence of access to virtual pages
  - ◆ # of memory pages (Frames)
  - ◆ Page replacement policy
- ➔ Metric → # of page faults, page fault rate, avg. memory access time

Example

of frames = 4

Reference sequence (in temporal order)

1, 3, 5, 0, 4, 1, 7, 5, 7, 5, 1, 2, 7, 3, 7, 0

Qn: What is the best strategy?

Belady’s optimal algorithm (MIN)

IDEA: Replace the page that will be referenced after the longest time

#of frames = 4

Reference sequence (in temporal order)

1, 3, 5, 0, 4, 1, 7, 5, 7, 5, 1, 2, 7, 3, 7, 0

#of page faults = ?

Optimality of Belady

- ➔ Why MIN is optimal? Is there a proof?

- ➔ Impossible to implement → If only we know the future :-)
- ➔ Useful to know the best, for comparison

First In First Out (FIFO)

IDEA: Replace the page that is in memory for the longest time

#of frames = 4

Reference sequence (in temporal order)

1, 3, 5, 0, 4, 1, 7, 5, 7, 5, 1, 2, 7, 3, 7, 0

#of page faults = ?

Least Recently Used (LRU)

IDEA: Replace the page that is not referenced for the longest time

#of frames = 4

Reference sequence (in temporal order)

1, 3, 5, 0, 4, 1, 7, 5, 7, 5, 1, 2, 7, 3, 7, 0

#of page faults = ?

LRU implementation issues

- How to track least recently accessed?
- Alternate 1: Access timestamp
  - Sorted list based on access timestamps
  - Data structure?
- Alternate 2: Stack
  - Accessed virtual page moves to TOS
  - Element @stack bottom evicted
- X86 provides accessed bit in PTE
- Approximate LRU: CLOCK, CLOCK-Pro

Belady’s anomaly

Consider the following access sequence

0, 1, 2, 3, 0, 1, 4, 0, 1, 2, 3, 4

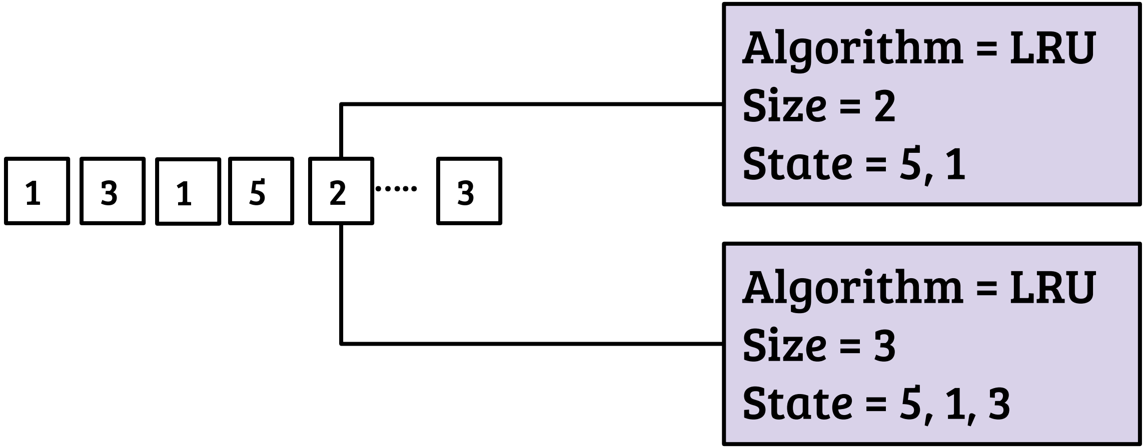
With FIFO,

Page faults with 3 frames = ?

Page faults with 4 frames = ?

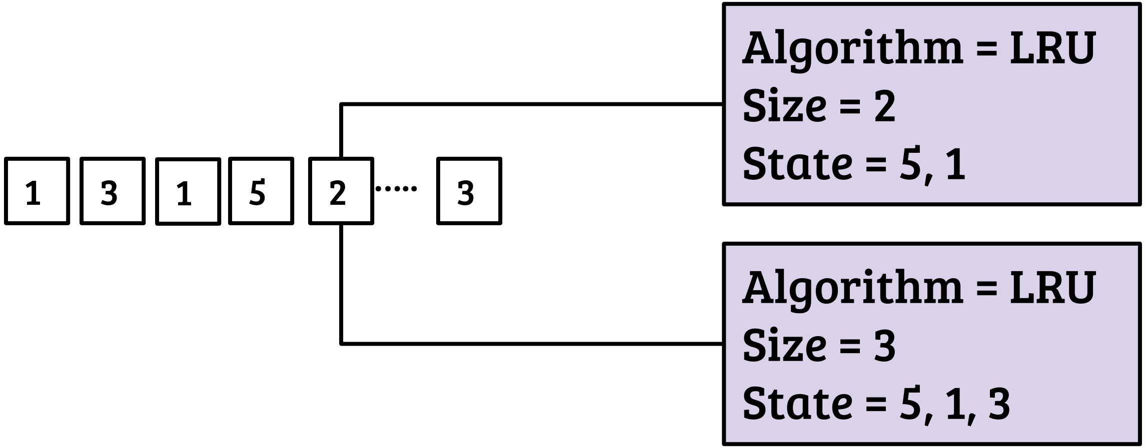
Stack algorithms

- Simply put, eviction algorithms not suffering from Belady's anomaly
- More formally, for each access in the reference string, set of pages in memory of size K will always a subset of pages in memory of size K+1



Stack algorithms

- ➔ Simply put, eviction algorithms not suffering from Belady's anomaly
- ➔ Formally, for each access in the reference string, set of pages in memory of size K frames will always a subset of pages in memory of size K+1 frames



MIN and LRU are stack algorithms. Proof left as exercise.

For interested readers

- ➔ CLOCK
- ➔ CLOCK-Pro
- ➔ Adaptive Replacement Cache (ARC)
- ➔ Miss ratio curve (MRC)