7.3 Operating System Issues

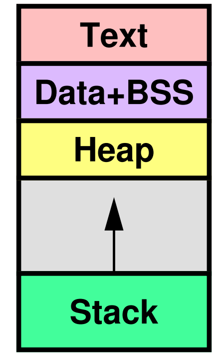
- *General Concerns*
- Representative Systems
- Copy on Write and Fork
- Backing Store Issues

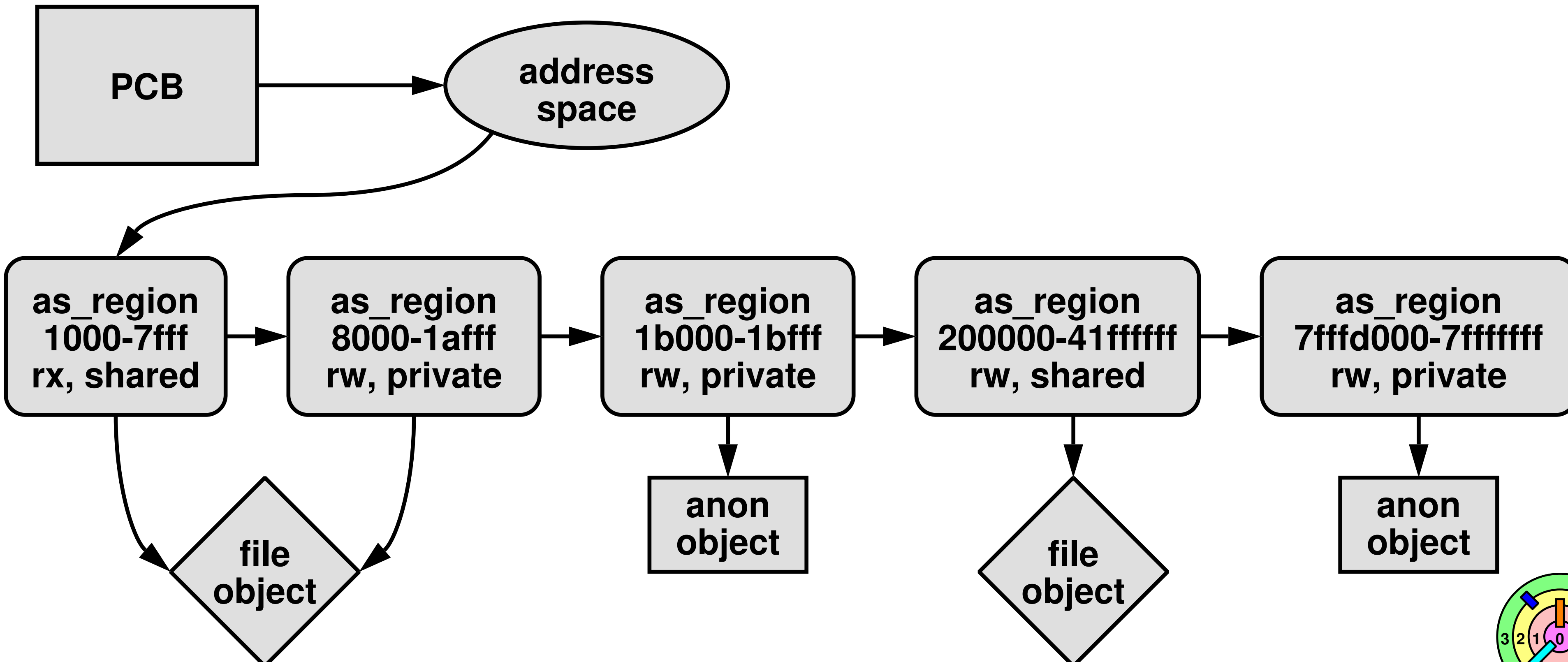
Virtual Memory: Traditional OS Issues

- Fetch policy
- Placement policy
- Replacement policy

Virtual Memory: Traditional OS Issues

- Fetch policy
- Placement policy
- Replacement policy





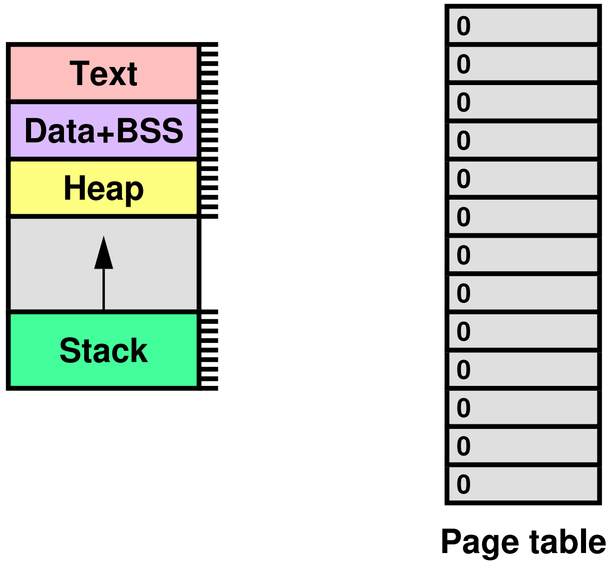
A Simple Paging Scheme

- ***Fetch*** **policy**
  - **start process off with no pages in primary storage**
  - **bring in pages** ***on demand*** **(and only on demand)**
    - **this is known as** ***demand paging***
      - **defer processing until you absolutely have to do it**
      - **why? because you may not have to process** ***at all***
      - ***demand paging*** **is an instance of** ***Lazy Evaluation*****, a powerful idea used in computer science**

A Simple Paging Scheme

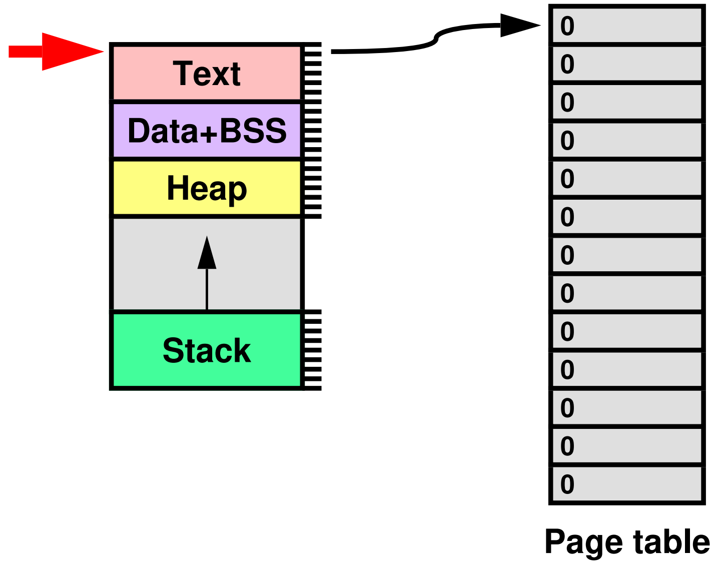
- ***Placement*** **policy**
  - **unlike disk pages, it doesn't matter here - put the incoming page (from disk) in the first available physical page**
    - **page *frames* are used to keep track of physical pages**
- ***Replacement*** **policy**
  - **required if there is not enough *resource* to go around**
  - **e.g., replace the page that has been in primary storage the longest (FIFO policy, which can be bad)**

Demand Paging



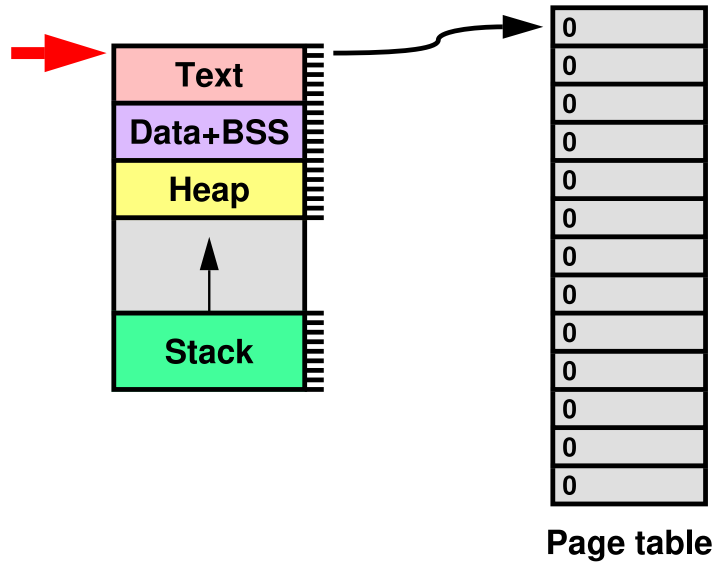
- In `exec()`, address space is created and page table is cleared with all entries having V=0

Demand Paging



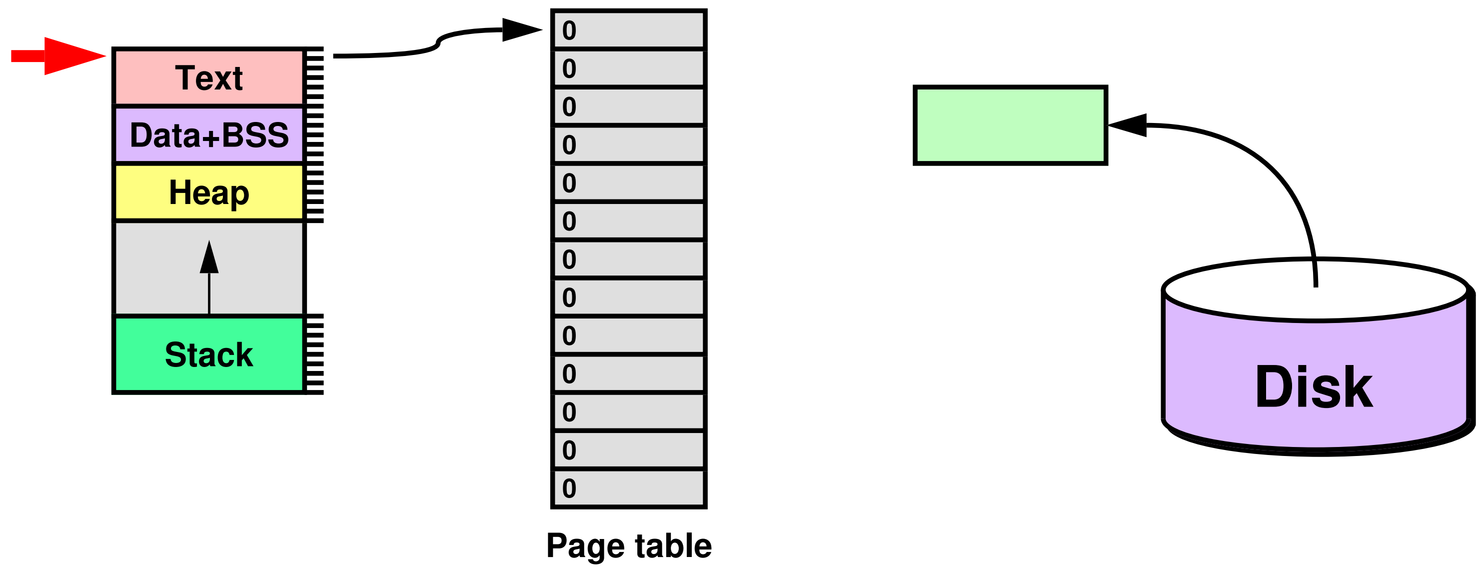
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the ***first instruction executes***

Demand Paging



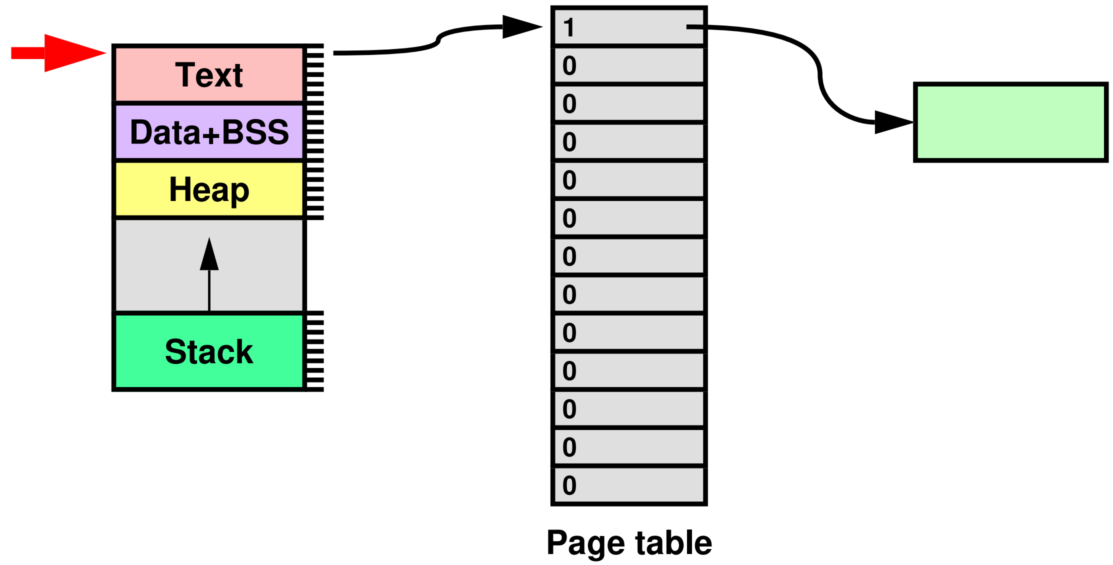
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the *first instruction executes*
    - since V=0, the hardware traps into the kernel

Demand Paging



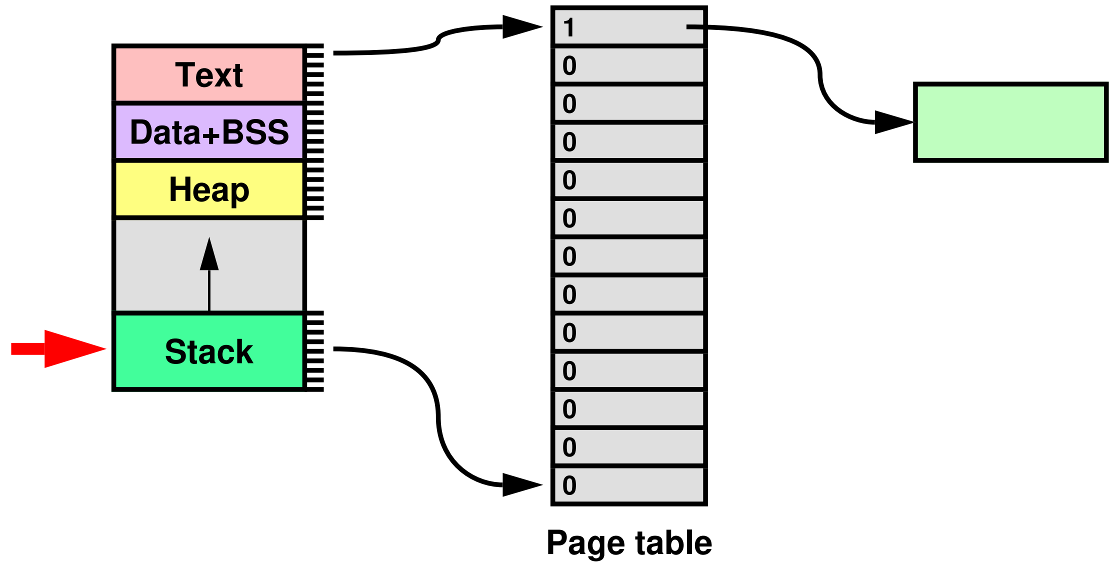
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the *first instruction executes*
    - since V=0, the hardware traps into the kernel
    - the kernel *allocates a physical page* and copy the first 4KB of code into this page (allocate from where?)
      - point the corresponding page table entry to this page
      - update all necessary data structures

Demand Paging



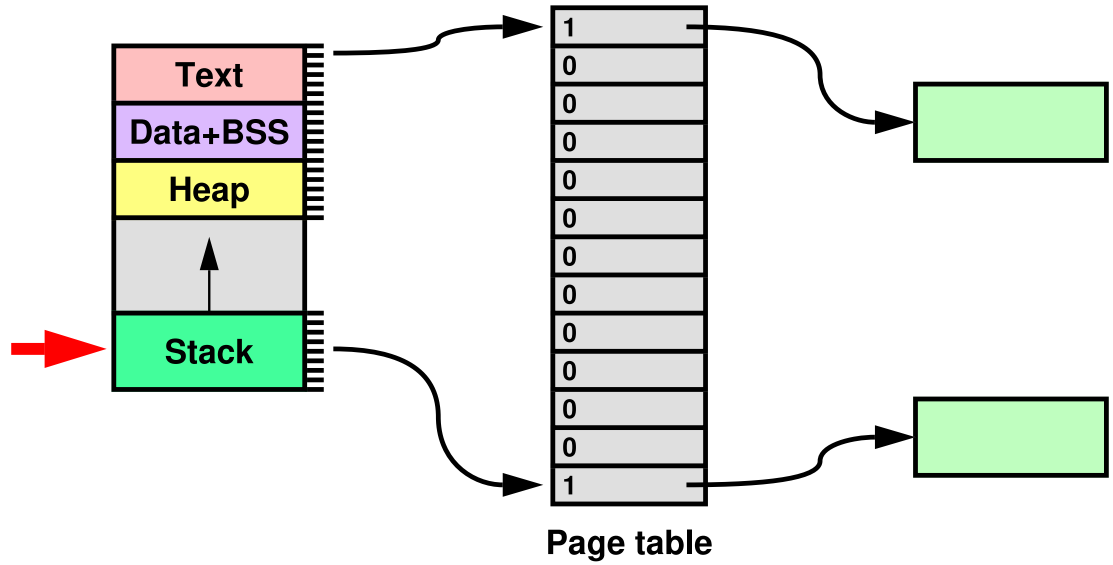
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the *first instruction executes*
    - since V=0, the hardware traps into the kernel
    - the kernel *allocates a physical page* and copy the first 4KB of code into this page (allocate from where?)
      - point the corresponding page table entry to this page
      - update all necessary data structures
    - set V=1 and return from the trap

Demand Paging



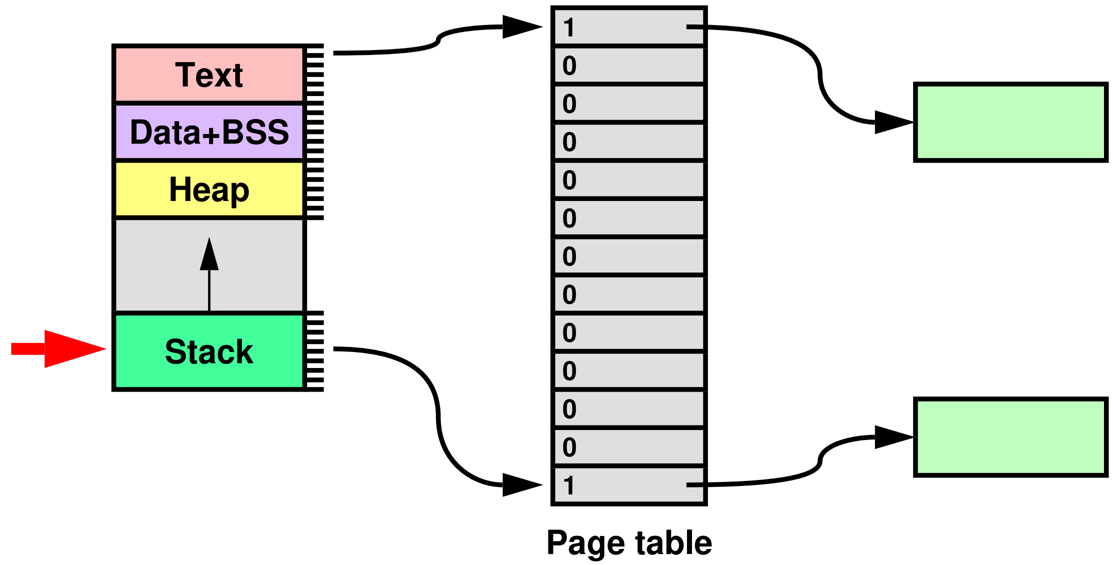
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the program access the *stack*

Demand Paging



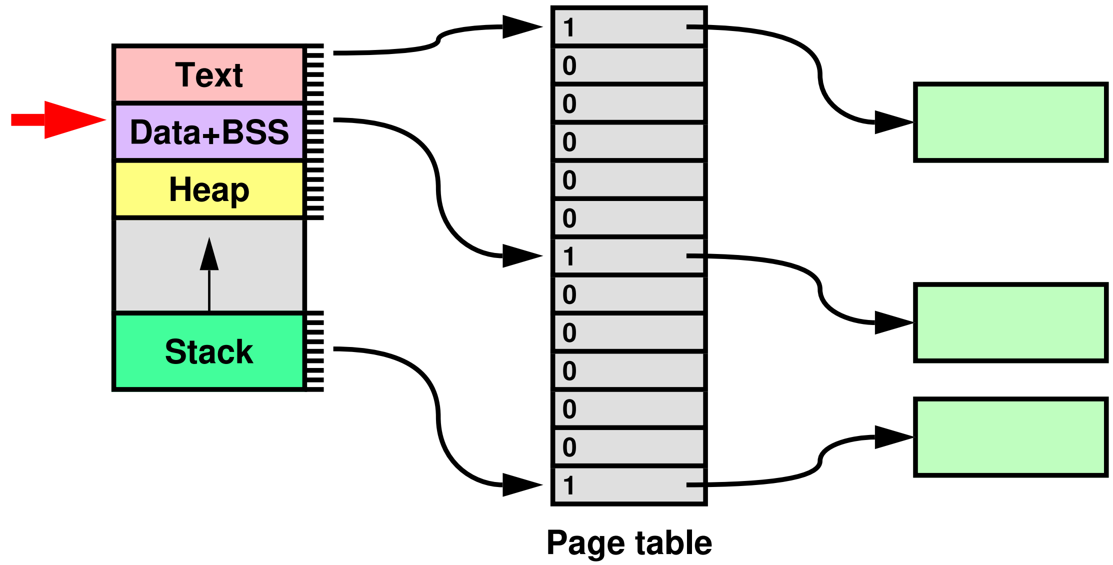
- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the program access the ***stack***
    - similar things happen

Demand Paging



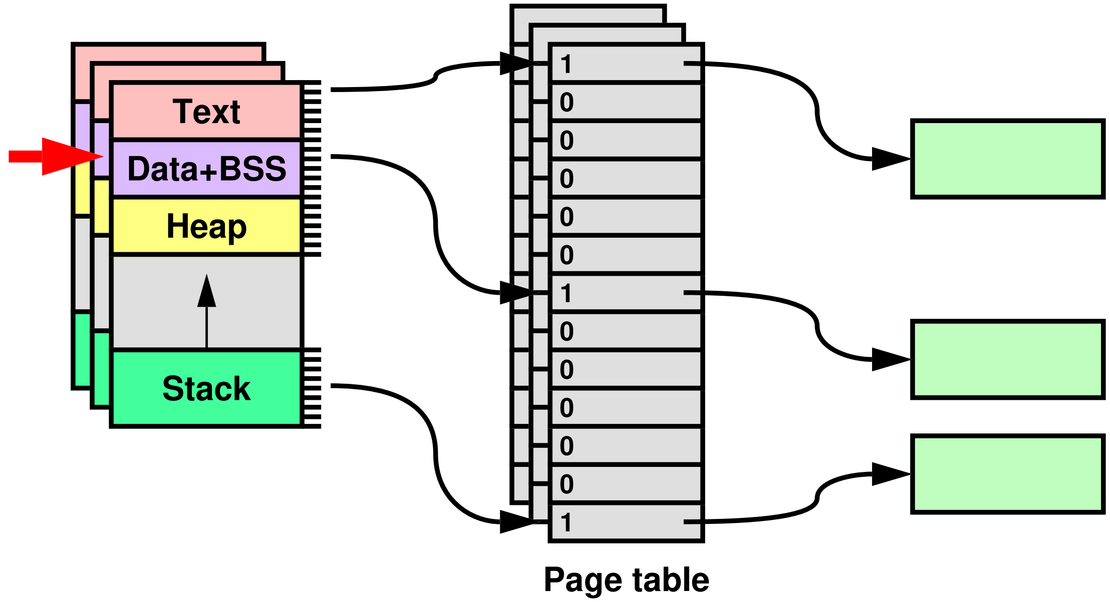
- In **exec()**, address space is created and page table is cleared with all entries having V=0
  - as the program access the ***stack***
    - similar things happen
    - although stack is a little different since it needs a ***backing store*** and need to set up for ***copy-on-write***

Demand Paging



- In `exec()`, address space is created and page table is cleared with all entries having V=0
  - as the program access the ***stack***
    - similar things happen
    - although stack is a little different since it needs a ***backing store*** and need to set up for ***copy-on-write***
  - accessing the ***data segment*** is similar to stack (but different)
    - original (read-only) backing store is the executable file
    - after copy-on-write, backing store is the swap space

Demand Paging



- Remember, there are multiple processes and multiple page tables that the OS is servicing
  - complicated by the fact that page frames can be shared
- In kernel 3, you need to make sure that every time when you return back into user space, all kernel data structures are in a consistent state

Page Fault

- ***Page Fault*** *(*accessing a page with ***V=0****)*
  1) Trap occurs (due to a page fault)
  2) Find free physical page
  3) Write page out if no free physical page
  4) Fetch page
  5) Return from trap

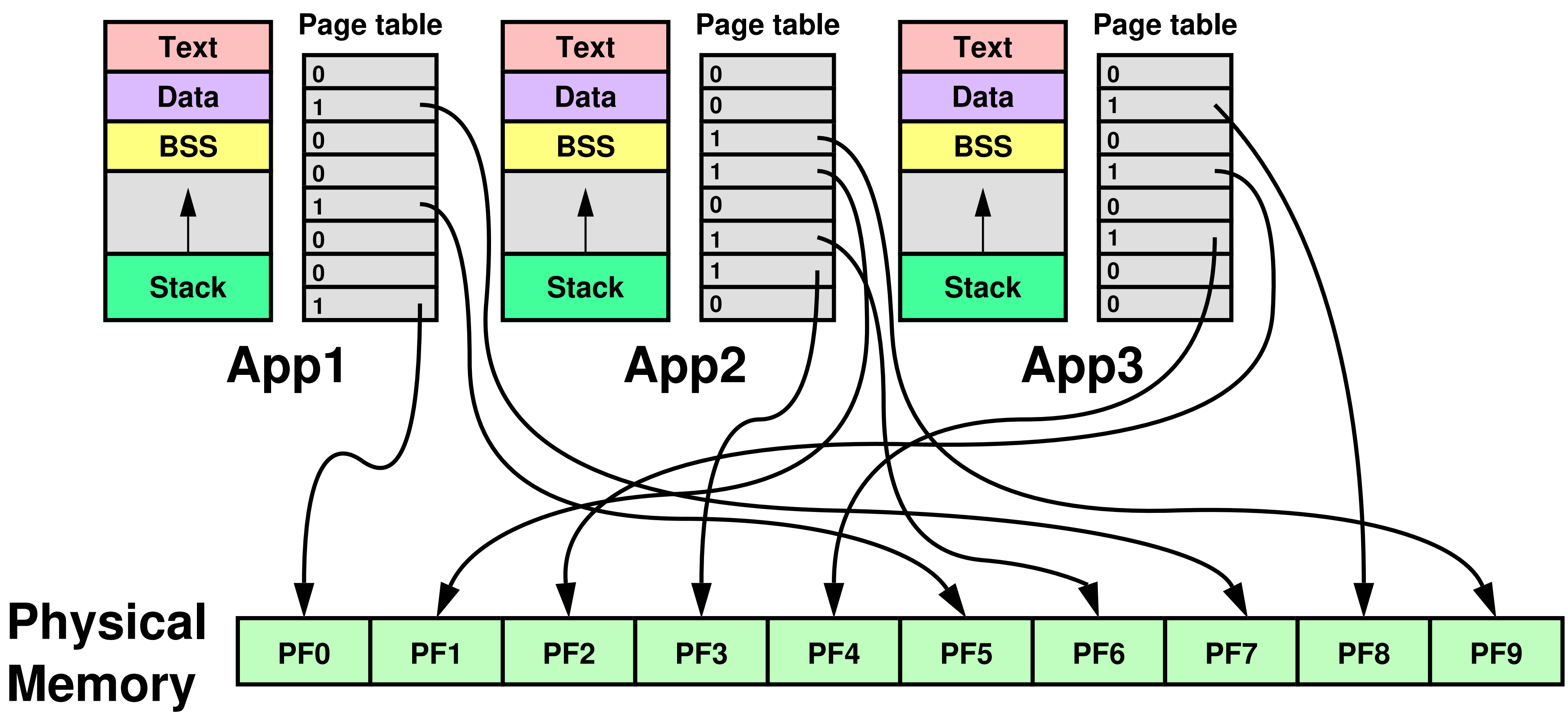
- Issues
  - in step (2), where and how do we find such a ***free physical page****?*

Page Fault

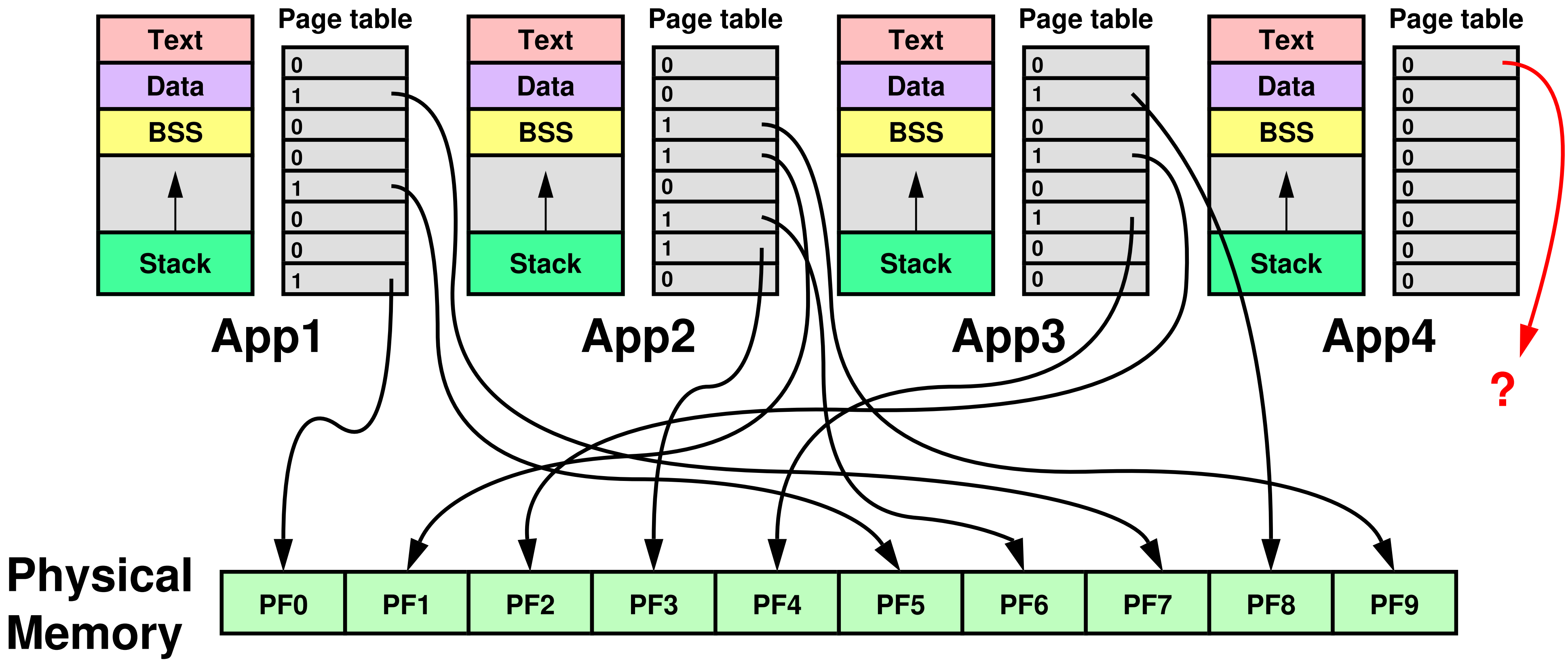
- ***Page Fault*** (accessing a page with ***V=0***)
  1) Trap occurs (due to a page fault)
  2) Find free physical page
  3) Write page out if no free physical page
  4) Fetch page
  5) Return from trap

- Issues
  - in step (2), where and how do we find such a ***free physical page***?
    - the Buddy System is used
      - return NULL if no free physical page is available
  - in step (3), where and how do we find an ***in-use physical page*** to write out to disk?

Example

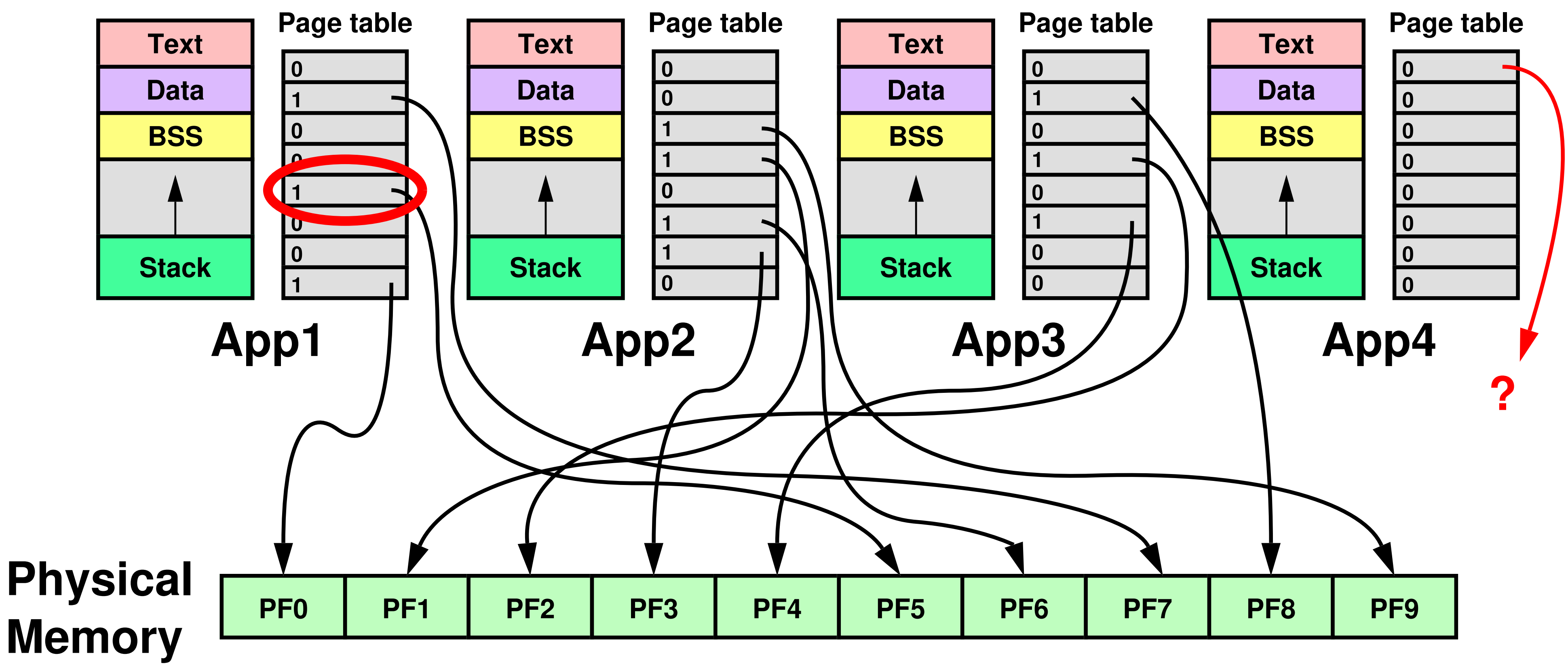

Text
Data
BSS
Stack
Page table
0
1
0
0
1
0
0
1
App1
Text
Data
BSS
Stack
Page table
0
0
1
1
0
1
1
0
App2
Text
Data
BSS
Stack
Page table
0
1
0
1
0
1
0
0
App3
Physical Memory
PF0
PF1
PF2
PF3
PF4
PF5
PF6
PF7
PF8
PF9

Example



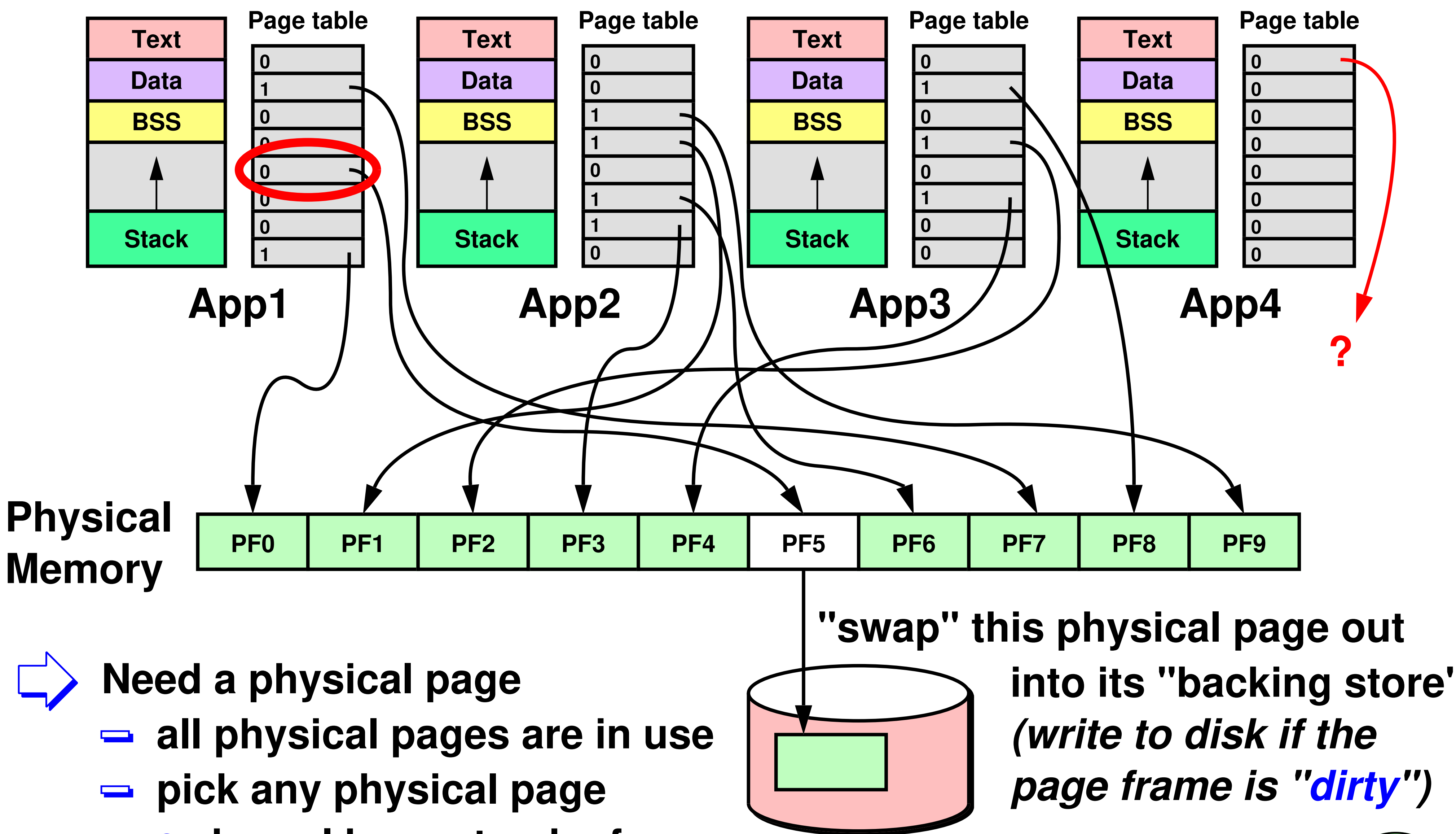
- Need a physical page
  - all physical pages are in use

Example



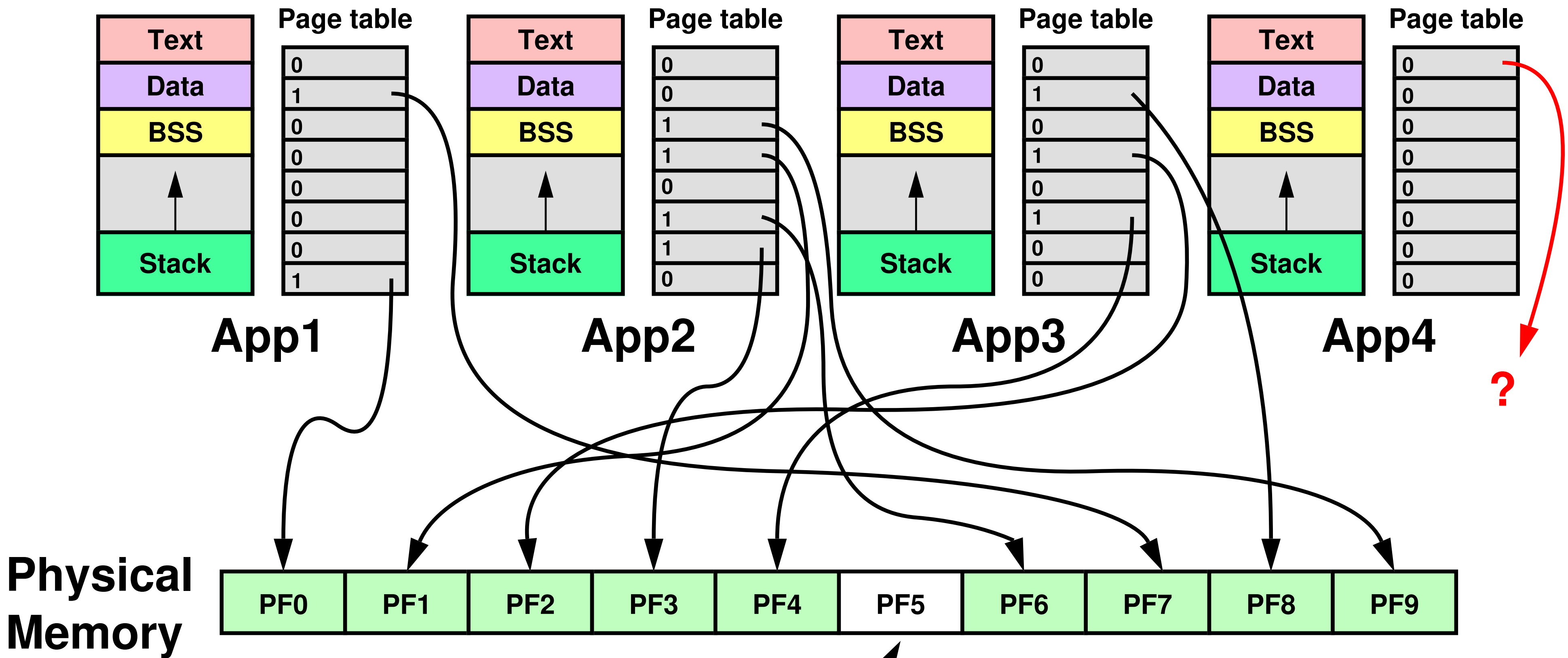
- Need a physical page
  - all physical pages are in use
  - pick any physical page
    - well, according to the *page replacement policy*

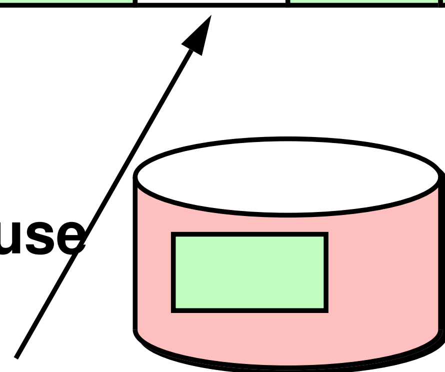
Example



- Need a physical page
  - all physical pages are in use
  - pick any physical page
    - kernel keeps track of where the physical page is copied to

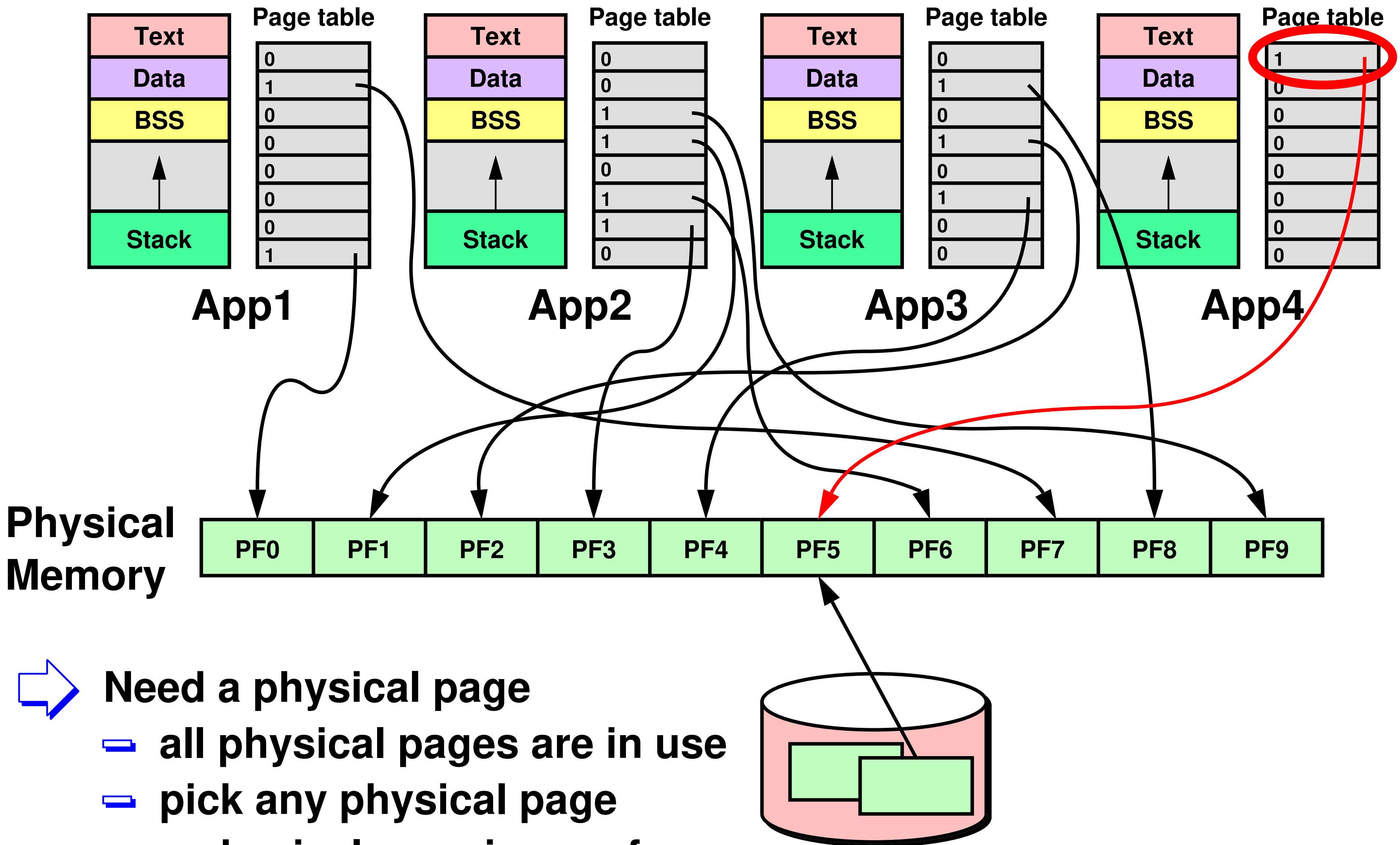
Example





- Need a physical page
  - all physical pages are in use
  - pick any physical page
  - a physical page is now free

Example



- Need a physical page
  - all physical pages are in use
  - pick any physical page
  - a physical page is now free
  - fetch page from disk and fix up page table