Virtual Memory and Paging (3)

ICS332

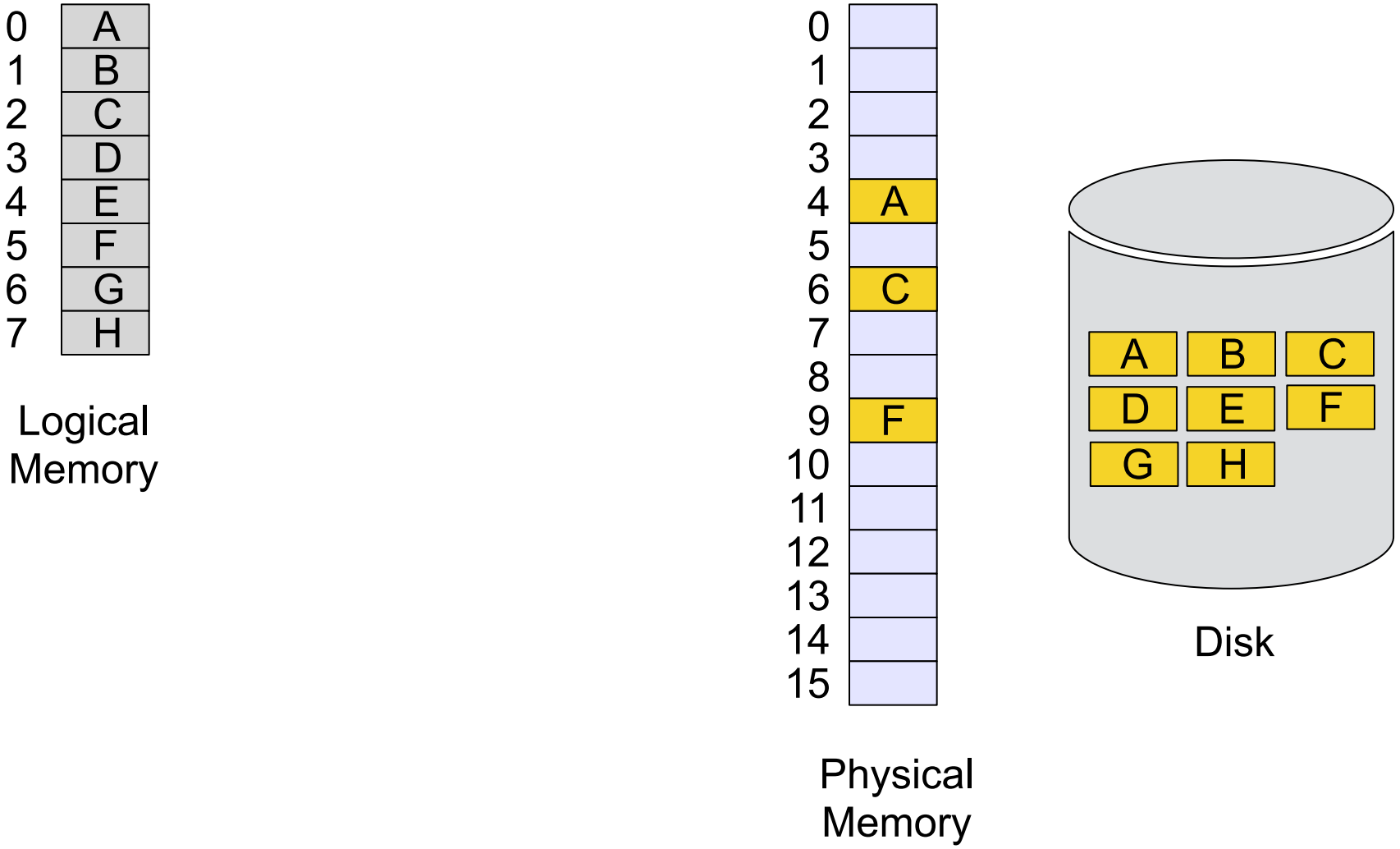
Operating Systems

Henri Casanova (henric@hawaii.edu)

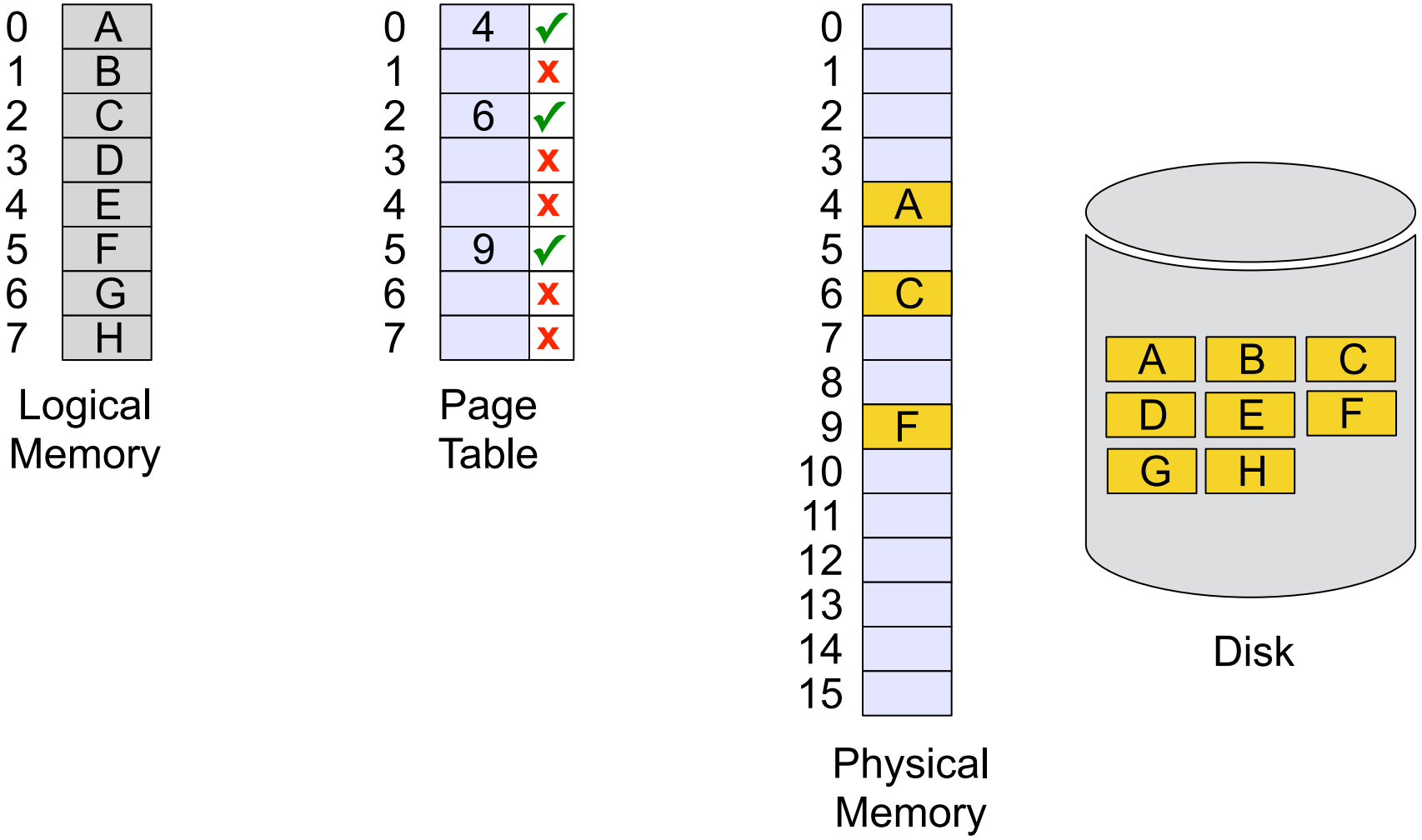
Demand Paging

- The way in which the OS allocates pages to a process is called Demand Paging
- “Don’t load a page before the process references it”
  - Initially just load one page, the one with the first instruction of the program
  - Each time the program issues an address, load the corresponding page if not already loaded
- This is a “lazy” scheme, as opposed to the “eager” scheme that would load all pages at once
- For each page, the OS keeps track of whether it is in RAM or not
- This is done using the valid bit of the page table entries
  - A page is marked as valid if it is legal and in memory
  - A page is marked as invalid if it is illegal or on disk
  - Initially all pages are marked invalid
- During address translation, if the bit is invalid, a trap is generated: a page fault

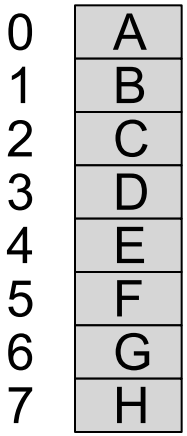
Demand Paging: Valid Bit Example



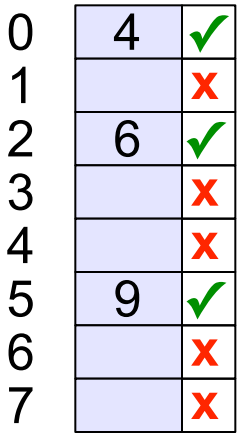
Demand Paging: Valid Bit Example



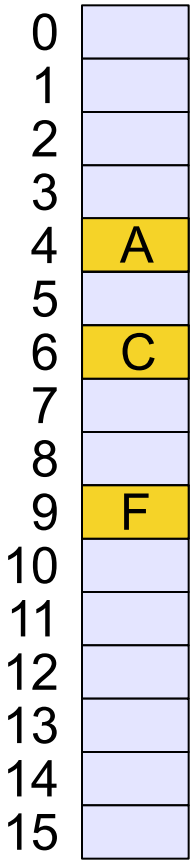
Demand Paging: Valid Bit Example



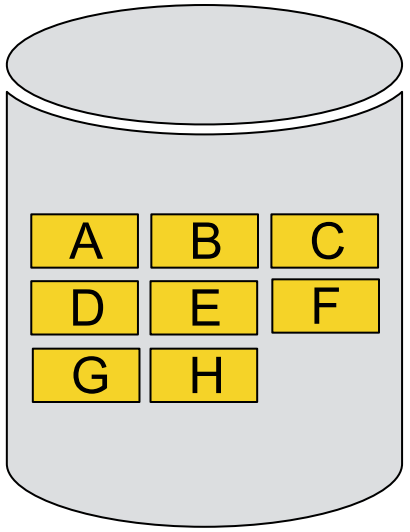
Logical Memory



Page Table



Physical Memory



Disk

Access Page 2: no page fault
Access Page 3: page fault

Page Faults

- When the CPU issues an address, first one determines whether it's potentially legal or not
  - i.e., does it correspond to a page number that's not beyond the number of pages allowed for a process?
  - If it is illegal, then the process is aborted with some message
- Lookup the valid bit in the page table entry
- If the valid bit is set, do the address translation as usual
- If not:
  - Find a free frame (from the list of free frames in the kernel)
  - Schedule the disk access to load the page into the frame
  - Kick the process off the CPU and put it the blocked/waiting state
  - Once the disk access is complete, update the process' page table with the new logical/physical memory mapping
  - Update the valid bit
  - Put the process back into the Ready Queue (it should then run soon)
  - That process will rerun the instruction that caused the trap

Rerun the “offending” instruction

- After a page fault is resolved and the OS has loaded the required page in RAM, one simply **reruns the instruction from scratch** (i.e., restart the fetch, decode, execute cycle at that instruction)

- This is only possible because our instructions don’t modify more than one memory location
  - Which avoids a difficult “the instruction did half its work in RAM, but then page faulted, so when you restart it be careful that the first half of the work was already done” situation
- In other terms, load/store ISAs are perfectly designed for page faults

Virtual Memory Performance

- We know that loading from disk is very slow. What are the limits of this on-demand mechanism? Is it worth using?
- Let t_m be the memory access time (10ns to 200 ns; typically: 70 ns), i.e., the time to access a byte in memory;
- Let t_p be the page fault time, i.e., the time required to load the page from the disk, place it in memory, and rerun the instruction. Typically: 5-50 ms (SSD: 3 to10 times faster)
- How much faster is the memory compared to the disk?

- Assume that t_m = 10ns = 10^{-8} s and t_p = 10ms = 10^{-2} s
- Then t_p / t_m =10^{-2} / 10^{-8} = 10^6
- The memory is 1 million time faster than the disk!
  - This is just one example, but regardless it's orders of magnitude

Performance: Effective Access Time

- Consider a process that access memory n times. n_0 of these times there is no page faults, and n_p of these times there is a page fault ($n = n_0 + n_p$). The total memory access time T is:

$$T = n_0 \times t_m + n_p \times t_p$$

- The average access time for one memory access, t, is:

$$t = (n_0 \times t_m + n_p \times t_p) / n = (1 - n_p / n) \times t_m + (n_p / n) \times t_p$$

- Let $p = n_p / n$ be the page fault probability, or page-fault rate ($0 \le p \le 1$)
- The average access time is then:

$$t = (1 - p)\, t_m + p\, t_p$$

Performance: Effective Access Time

- With the numbers given previously (rescaling to nanoseconds and assuming that p is small):

  $$t \sim 10 + 10{,}000{,}000 \times p$$

- Ideally (p = 0) there is no page fault and the access time would be 10 ns
- Say we do not want a performance degradation of more than 10% on average?
- That means $10 + 10{,}000{,}000 \times p < 11$
- This gives us: $p < 10^{-7} = 0.00001\%$
  - This is absolutely tiny….
- If our page fault rate is 1%, then $t = 10 + 10{,}000{,}000 \times 0.01 \sim 100{,}000$ns
  - The memory would appear 10,000x slower!!!

Virtual Memory Performance

- Conclusion: The page fault rate must be kept as small as possible

- What can be done?
  - Increase the memory size
  - Limit the size of the process address space
  - Tell programmers to develop programs with small address spaces ⇒ That's your job!
    - Every time you use more RAM, you increase your page fault probability

Aside: fork()-exec()

- We have said that fork() makes a copy of the parent process address space to create an identical child process
- But most of the time exec() is used in the child to run another program

Some code

```
if (!fork()) {
  exec("/bin/ls", …);
}
```

- Making a copy of the parent's address space is wasteful
- The child address space is immediately overwritten with another (that of "/bin/ls")
- So the copy is completely unecessary

Aside: Copy-on-Write

- Copy-on-Write: During a **`fork()`**, don't copy the address space and initially share all pages
  - Save for some heap and stack pages, that are necessary for any new process
- Whenever the parent **or** the child modifies a page, then copy it

- This "lazy" scheme is used in all OSes (Windows, Mac, Linux)
- In the fork-exec classical example, no page is copied!

Page Replacement

- Virtual Memory increases multi-programming and provides the illusion of large address spaces
- What if we run out of memory?
  - A page fault occurs
  - Oh no, the free-frame list is empty!!
- **We need to kick a page out of RAM**
- This is called page replacement
  - Evict a victim page from a frame (write it to the disk if necessary)
  - Put the newly needed page into that frame
- Page replacement may thus require two page transfers

- When the physical memory is full and all processes try to access it, everything just gets slooooow...

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address Space of Process #1

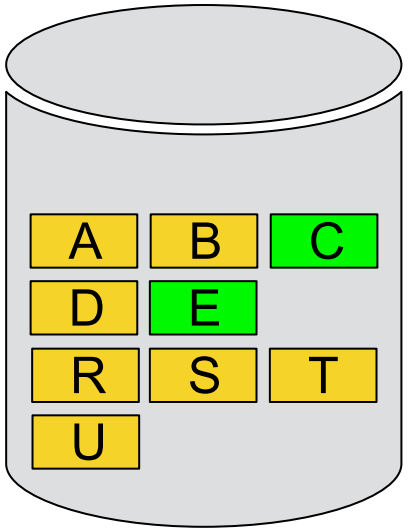
0	0	✓
1	3	✓
2		x
3	2	✓
4		x

Page Table of Process #1

Process #1 needs to access Page 4 (E)
The kernel selects a victim frame

0	A
1	S
2	D
3	B
4	U
5	T
6	R

Physical Memory



Disk

0	R
1	S
2	T
3	U

Address Space of Process #2

0	6	✓
1	1	✓
2	5	✓
3	4	✓

Page Table of Process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address Space of Process #1

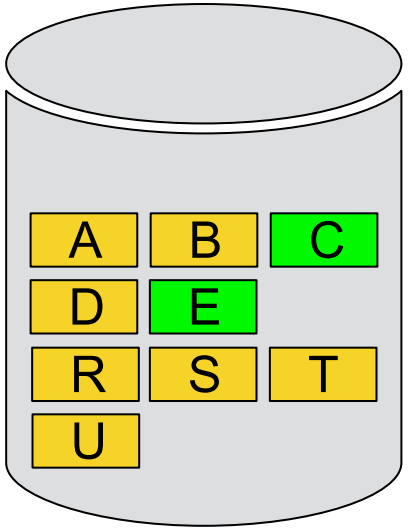
0	0	✓
1	3	✓
2		x
3	2	✓
4		x

Page Table of Process #2

Process #1 needs to access Page 4 (E)
The kernel selects a victim frame
Say frame 5 (which happens to belong to Process #2)

0	A
1	S
2	D
3	B
4	U
5	T
6	R

Physical Memory



Disk

0	R
1	S
2	T
3	U

Address Space of Process #2

0	6	✓
1	1	✓
2	5	✓
3	4	✓

Page Table of Process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address Space of Process #1

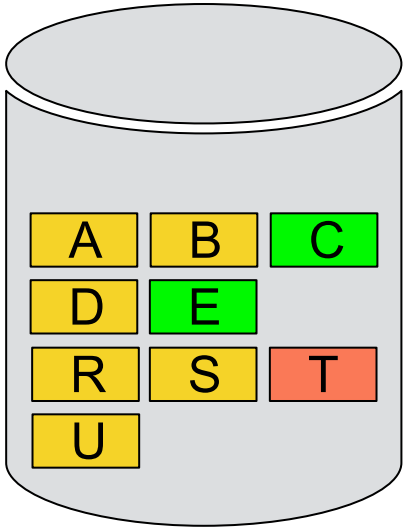
0	0	✓
1	3	✓
2		x
3	2	✓
4		x

Page Table of Process #2

Process #1 needs to access Page 4 (E)
The victim is written to disk

0	A
1	S
2	D
3	B
4	U
5	T
6	R

Physical Memory



Disk

0	R
1	S
2	T
3	U

Address Space of Process #2

0	6	✓
1	1	✓
2	5	✓
3	4	✓

Page Table of Process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address Space of Process #1

0	0	✓
1	3	✓
2		x
3	2	✓
4		x

Page Table of Process #2

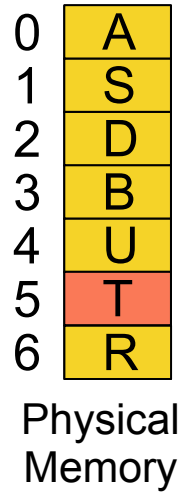
Process #1 needs to access Page 4 (E)
The page table of Process #2 is updated

0	R
1	S
2	T
3	U

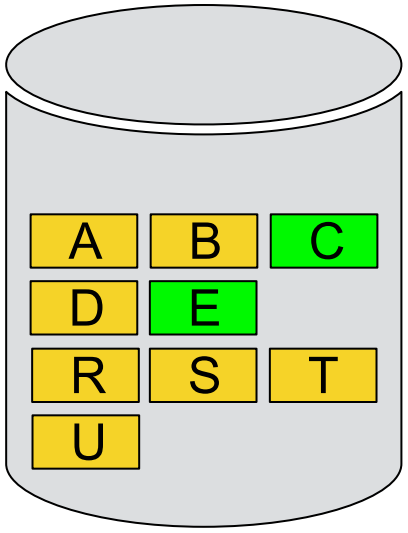
Address Space of Process #2

0	6	✓
1	1	✓
2	5	x
3	4	✓

Page Table of Process #2



Physical Memory



Disk

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address Space of Process #1

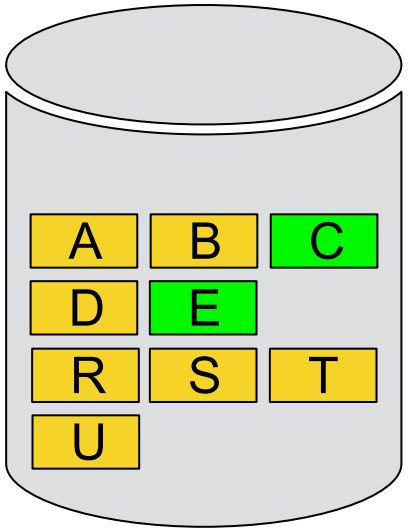
0	0	✓
1	3	✓
2		x
3	2	✓
4		x

Page Table of Process #2

Process #1 needs to access Page 4 (E)
The free-frame list is updated

0	A
1	S
2	D
3	B
4	U
5	T
6	R

Physical Memory



Disk

0	R
1	S
2	T
3	U

Address Space of Process #2

0	6	✓
1	1	✓
2	5	x
3	4	✓

Page Table of Process #2

Free Frames: {5}

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address Space of Process #1

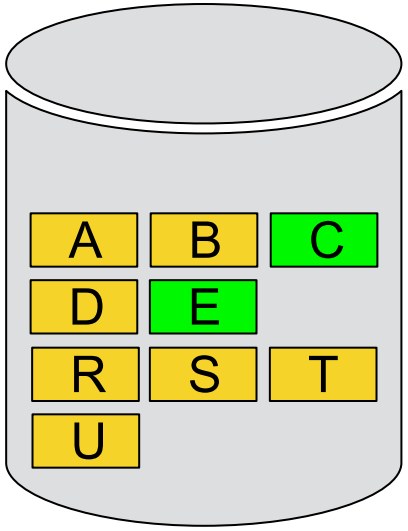
0	0	✓
1	3	✓
2		x
3	2	✓
4		x

Page Table of Process #2

Process #1 needs to access Page 4 (E)
The page is loaded into frame 5

0	A
1	S
2	D
3	B
4	U
5	E
6	R

Physical Memory



Disk

0	R
1	S
2	T
3	U

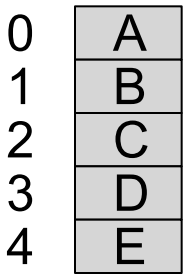
Address Space of Process #2

0	6	✓
1	1	✓
2	5	x
3	4	✓

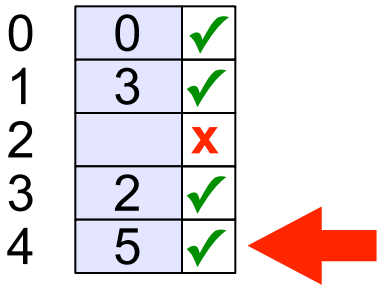
Page Table of Process #2

Free Frames: {5}

Page Replacement

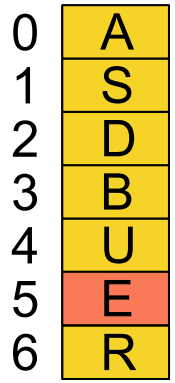


Address Space of Process #1

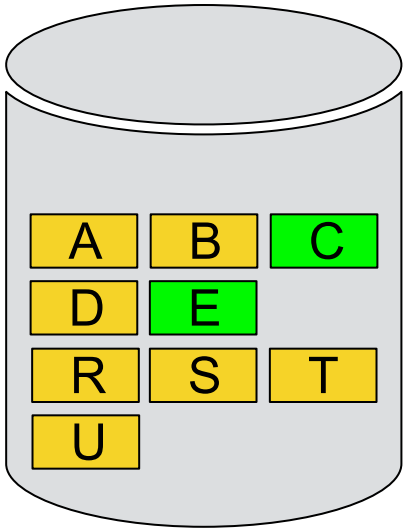


Page Table of Process #2

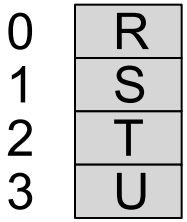
Process #1 needs to access Page 4 (E)
Process #1's Page Table is updated
Update the free-frame list



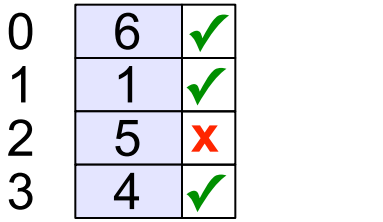
Physical Memory



Disk



Address Space of Process #2



Page Table of Process #2

Free Frames: {}

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address Space of Process #1

0	0	✓
1	3	✓
2		x
3	2	✓
4	5	✓

Page Table of Process #2

All is well until the next page fault

0	R
1	S
2	T
3	U

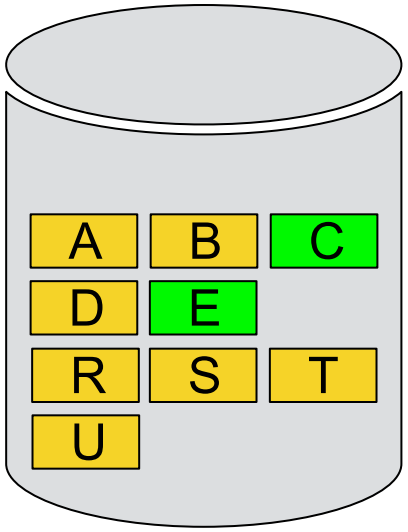
Address Space of Process #2

0	6	✓
1	1	✓
2	5	x
3	4	✓

Page Table of Process #2

0	A
1	S
2	D
3	B
4	U
5	E
6	R

Physical Memory



Disk

Free Frames: {}

Dirty Bit

- In the previous example, why write T back to disk if it had not been modified?
  - Perhaps T contains read-only code or data
  - Or Process #2 just hasn't had time to modify its bytes
- No need to write a victim back to disk if that victim has never been modified
- For this reason, each page table entry has a dirty bit
- This dirty bit is initially set to 0
- Whenever the process writes to the page, that dirty bit is set to 1
- If a page is evicted, it's written to disk only if its dirty
  - One speaks of "clean" and "dirty" pages
- Most OSes do opportunistic un-dirtying: If the disk is idle pick a dirty page, write it out and clear its dirty bit
  - The more clean pages in RAM, the faster page-faults will be when RAM is full

Conclusion

- At this point we have mechanisms
  - We can bring pages in from disk on demand (when page fault)
  - We can write pages to disk when needed (RAM is full)
  - The dirty bit is used to avoid doing redundant writes to disk
- What we need are policies
- The main questions are: Which pages do we kick back to disk? How many frames do we let a process have?
  - If we make good decisions we can lower the page-fault rate
  - The page-fault rate has to be super low (see the calculations a few slides back)
- So it's the usual story: first the mechanisms, and now the algorithms...