Virtual Memory (III)

ICS332 — Operating Systems

Henri Casanova (henric@hawaii.edu)

Spring 2018

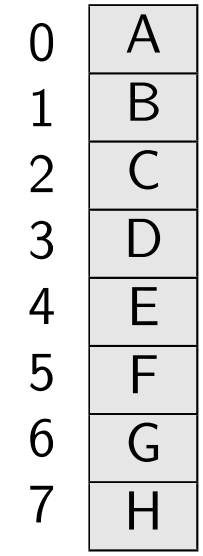
Demand Paging

- 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 loads all pages at once

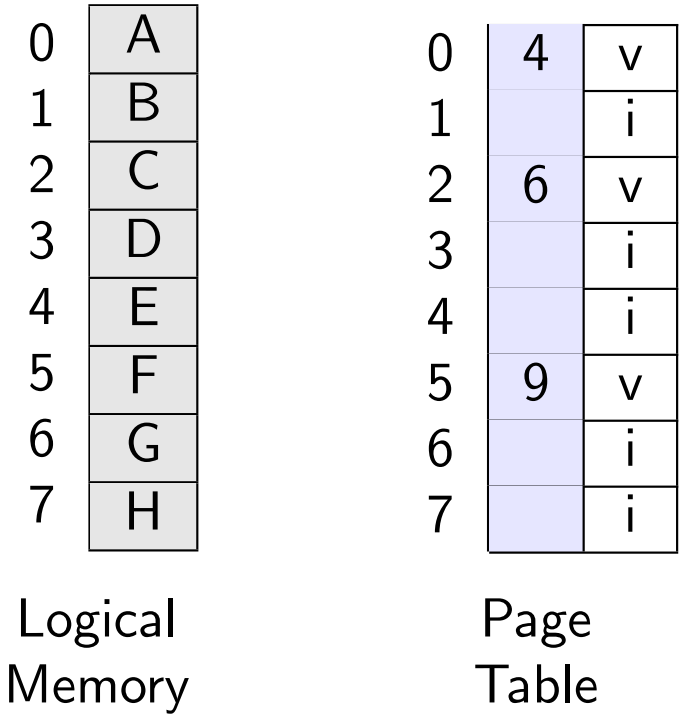
Demand Paging

- 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 loads 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

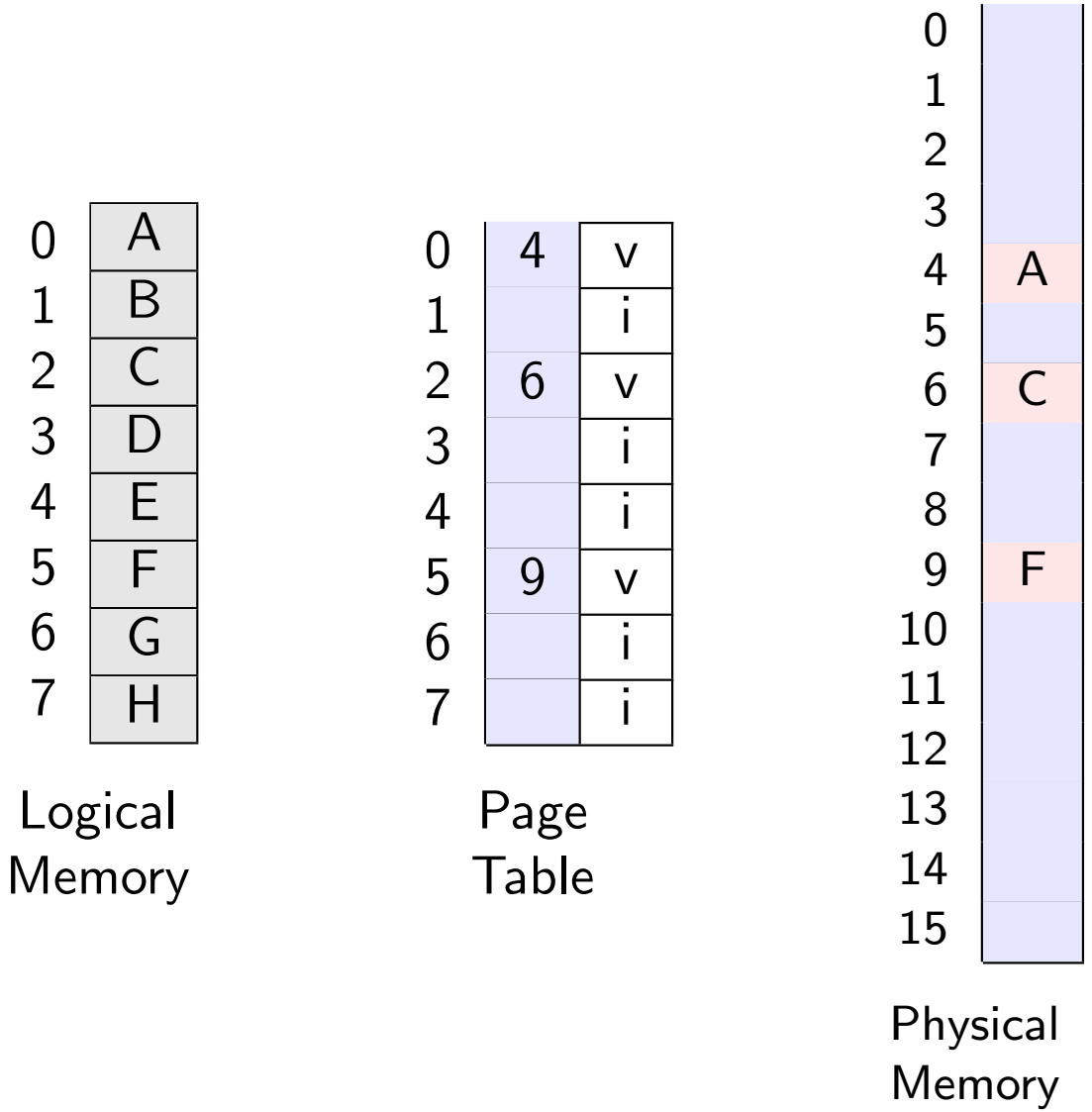
Demand Paging: Valid Bit Example



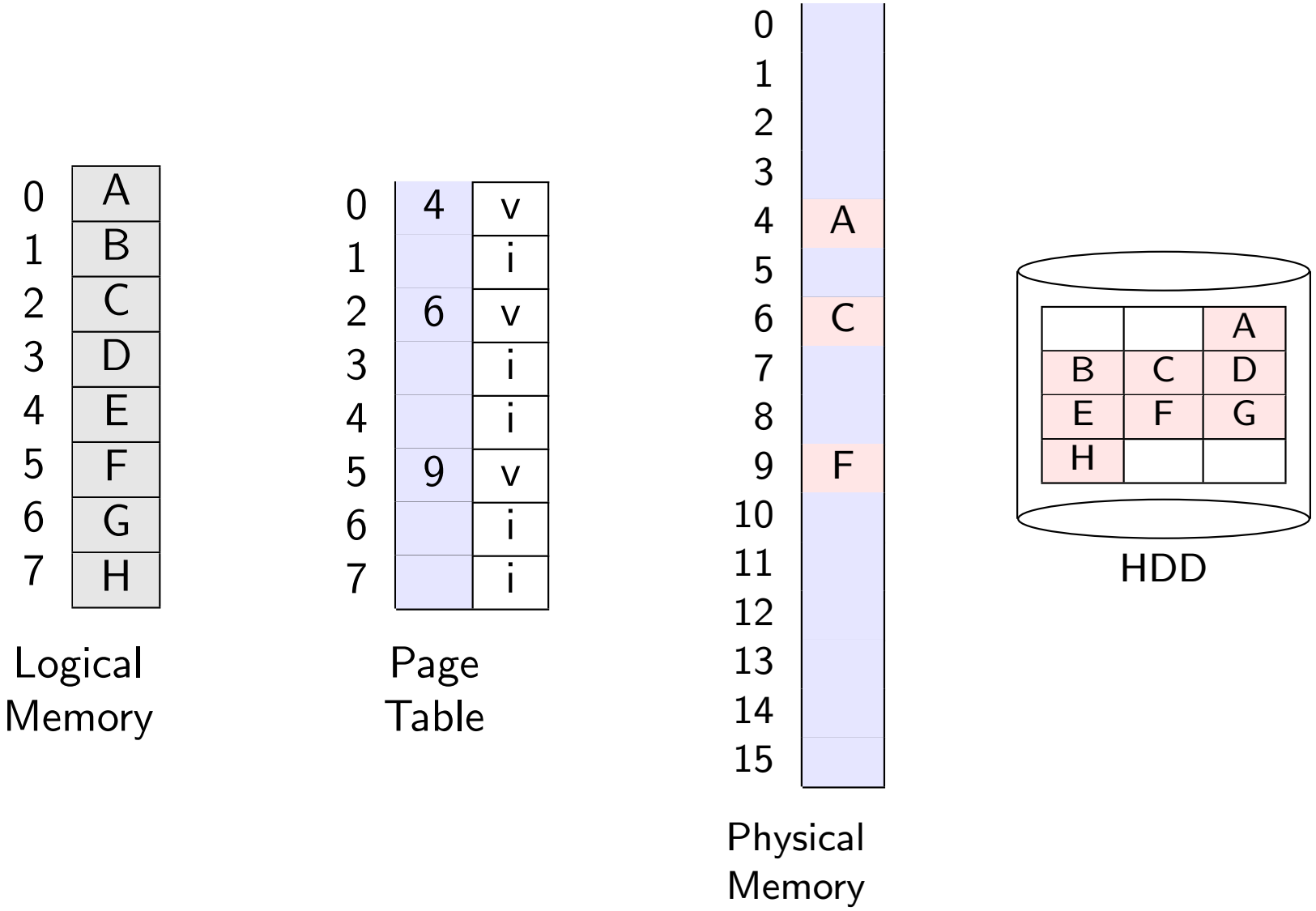
Demand Paging: Valid Bit Example



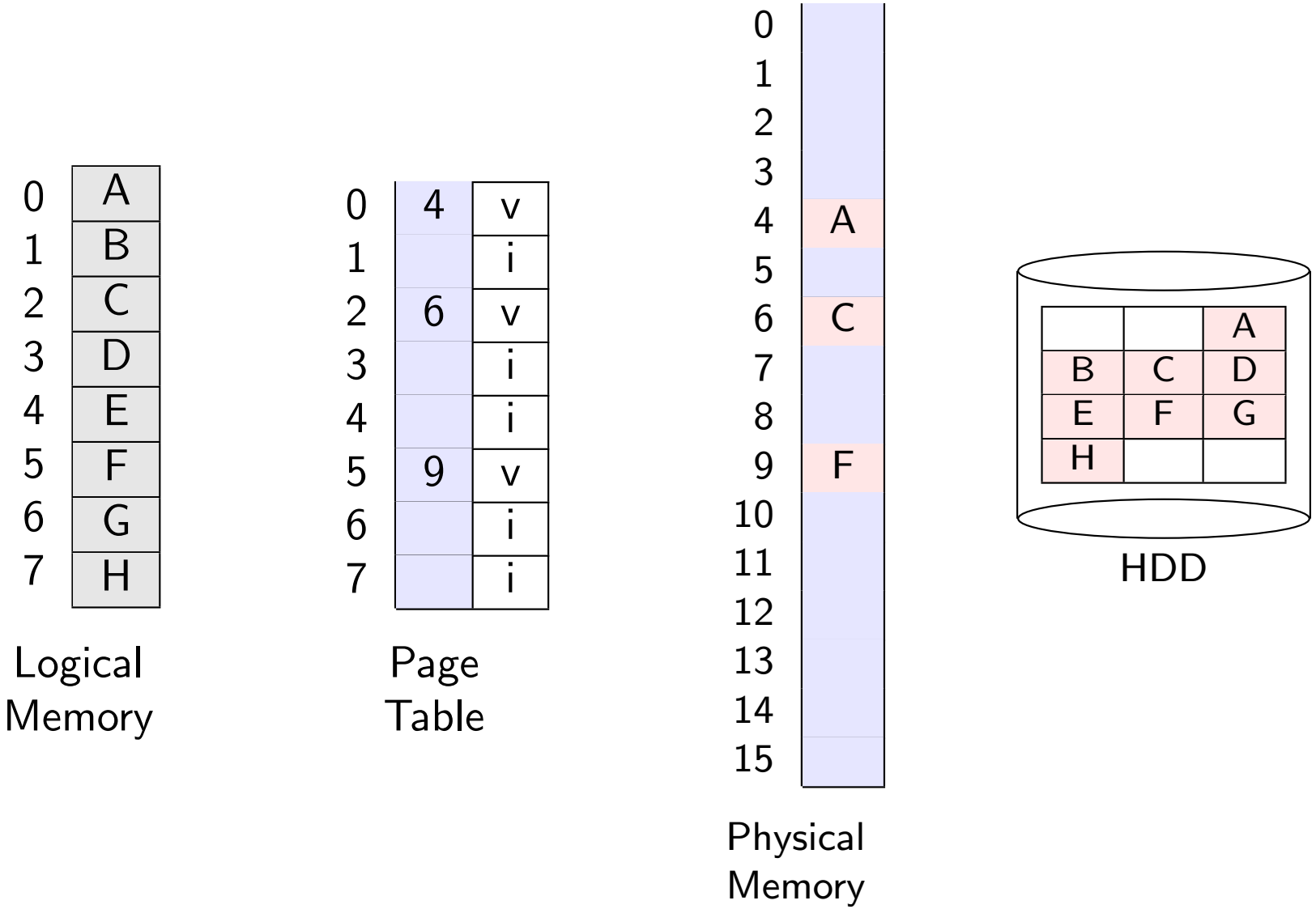
Demand Paging: Valid Bit Example



Demand Paging: Valid Bit Example

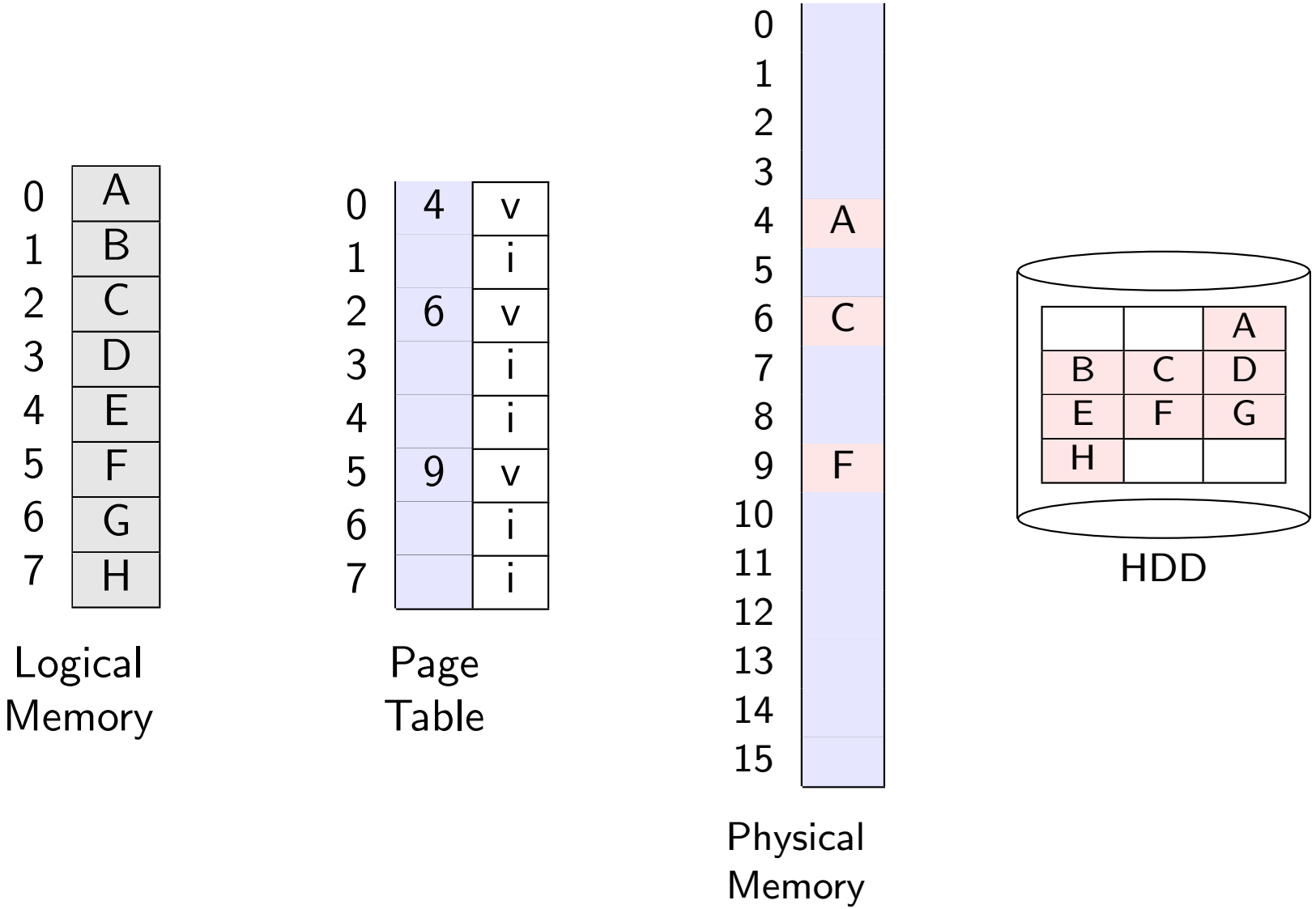


Demand Paging: Valid Bit Example



Access "Logical Page 2/C": No page fault

Demand Paging: Valid Bit Example



Access "Logical Page 2/C": No page fault

Access "Logical Page 3/D": Page fault

Page Faults

- When the CPU issues an address, first one determines whether it's 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

Page Faults

- When the CPU issues an address, first one determines whether it's 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

Page Faults

- When the CPU issues an address, first one determines whether it's 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
  - Rerun the instruction that caused the trap
  - Set the process state to Ready (it should then run soon)

Rerun the "offending" instruction

- If the page fault was because the instruction could not be fetched: (1) load the page, then (2) rerun the instruction from scratch (i.e., restart the fetch)
- If the page fault was because an operand value could not be read from memory: Do the same sequence
- If the trap been was issued because an operand value could not be written to memory: Do the same sequence

Rerun the "offending" instruction

- If the page fault was because the instruction could not be fetched: (1) load the page, then (2) rerun the instruction from scratch (i.e., restart the fetch)
- If the page fault was because an operand value could not be read from memory: Do the same sequence
- If the trap been was issued because an operand value could not be written to memory: Do the same sequence

- In all cases, it's pretty simple: just re-run the instruction from scratch
- 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?

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_d 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: 3-10 times faster)
- How much faster is the memory compared to the disk?

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_d 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: 3-10 times faster)
- How much faster is the memory compared to the disk?

Assume that $t_m = 10ns = 10^{-8}s$ and $t_p = 10ms = 10^{-2}s$

$$\frac{t_p}{t_m} = \frac{10^{-2}}{10^{-8}} = 10^6$$

The memory is 1 million time faster than the disk!

Virtual Memory 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:

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

Virtual Memory 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:

$$t = \frac{n_0 \times t_m + n_p \times t_p}{n} = (1 - \frac{n_p}{n}) \times t_m + \frac{n_p}{n} \times t_p$$

Virtual Memory 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:

$$t = \frac{n_0 \times t_m + n_p \times t_p}{n} = (1 - \frac{n_p}{n}) \times t_m + \frac{n_p}{n} \times t_p$$

- Let $p = \frac{n_p}{n}$ be the page fault probability, or page-fault rate ($0 \leq p \leq 1$)

Virtual Memory 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:

$$t = \frac{n_0 \times t_m + n_p \times t_p}{n} = (1 - \frac{n_p}{n}) \times t_m + \frac{n_p}{n} \times t_p$$

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

$$t = (1 - p)t_m + pt_p$$

Virtual Memory Performance

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

$$t \approx 10 + 10,000,000 \times p$$

- 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%? i.e. 10 ns + 10% = 11 ns

$$\begin{aligned} & 11 \geq & 10 + 10,000,000p \\ \Leftrightarrow \quad & p \leq & 10^{-8} \\ \Leftrightarrow \quad & p \leq & 0.000001\% \end{aligned}$$

- This is tiny!!!

Virtual Memory Performance

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

$$t \approx 10 + 10,000,000 \times p$$

- 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%? i.e. 10 ns + 10% = 11 ns

$$\begin{aligned} & 11 \geq & 10 + 10,000,000p \\ \Leftrightarrow \quad & p \leq & 10^{-8} \\ \Leftrightarrow \quad & p \leq & 0.000001\% \end{aligned}$$

- This is tiny!!!
- Just for kicks, what would a page fault rate of 1% cost?

$$t = 10 + 10,000,000 \times 0.01 \approx 100,000ns$$

Ouch! The memory would appear to be 10,000 times slower!

Virtual Memory Performance

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

Virtual Memory Performance

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

- What can be done?

Virtual Memory Performance

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

- What can be done?
  - Increase the memory size

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

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 $\Rightarrow$ That's your job! (every time you use more ram, you increase your page fault probability, and thus slow down your program)

Back to fork()-exec()

- We have said that `fork()` makes a copy of the parent process address space to create an identical child process

Back to 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", ...);
}
```

- Why is making a copy of the parent's address space wasteful?

Back to 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", ...);
}
```

- Why is making a copy of the parent's address space wasteful?
- The child address space is immediately overwritten with another (that of "/bin/ls")

Back to 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", ...);
}
```

- Why is making a copy of the parent's address space wasteful?
- The child address space is immediately overwritten with another (that of "/bin/ls")
  - If the parent has a 2GiB array, we just copied it (which takes time) and then immediately wiped it out!

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

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)

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!!

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

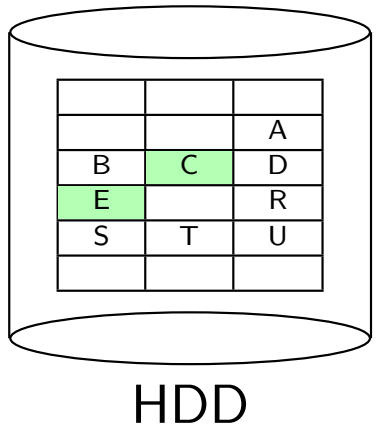
Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

Page table of process #1

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

Physical memory



HDD

0	R
1	S
2	T
3	U

Address space of process #2

P0	6	v
P1	1	v
P2	5	v
P3	4	v

Page table of process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

Page table of process #1

Process #1 needs to access Page 4 (E)

0	R
1	S
2	T
3	U

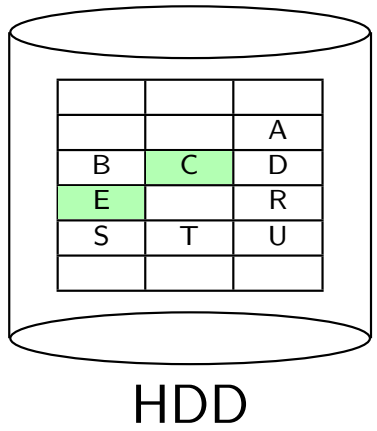
Address space of process #2

P0	6	v
P1	1	v
P2	5	v
P3	4	v

Page table of process #2

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

Physical memory



HDD

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

Page table of process #1

Process #1 needs to access Page 4 (E)

The kernel selects a victim frame

0	R
1	S
2	T
3	U

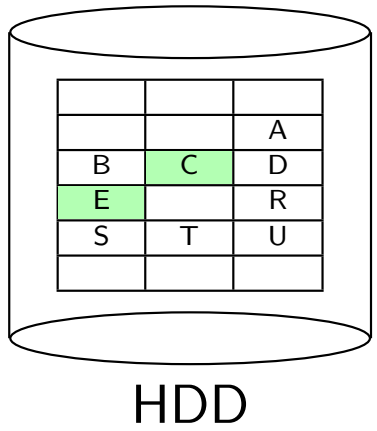
Address space of process #2

P0	6	v
P1	1	v
P2	5	v
P3	4	v

Page table of process #2

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

Physical memory



HDD

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

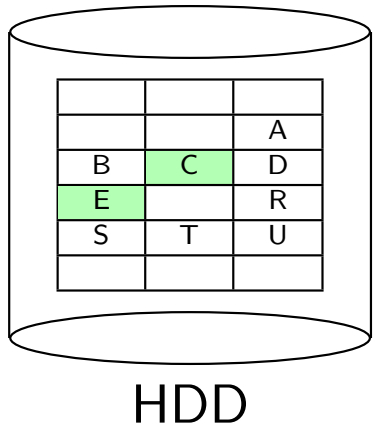
Page table of process #1

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

e.g. Frame 5

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

Physical memory



0	R
1	S
2	T
3	U

Address space of process #2

P0	6	v
P1	1	v
P2	5	v
P3	4	v

Page table of process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

Page table of process #1

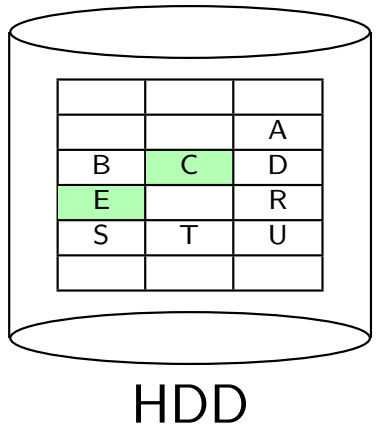
Process #1 needs to access Page 4 (E)
The kernel selects a victim frame

e.g. Frame 5

(which happens to belong to P#2 but it is "random")

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

Physical memory



0	R
1	S
2	T
3	U

Address space of process #2

P0	6	v
P1	1	v
P2	5	v
P3	4	v

Page table of process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space
of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

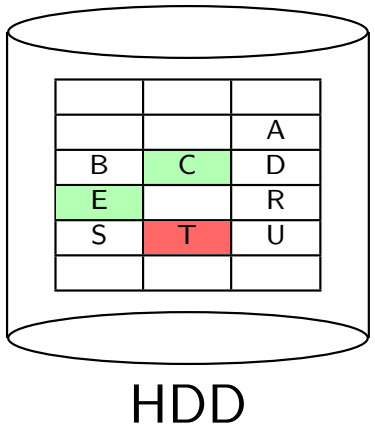
Page table
of process #1

Process #1 needs to access Page 4 (E)
The kernel selects a victim frame (e.g. frame 5)

The victim is written to disk

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

Physical
memory



0	R
1	S
2	T
3	U

Address space
of process #2

P0	6	v
P1	1	v
P2	5	v
P3	4	v

Page table
of process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

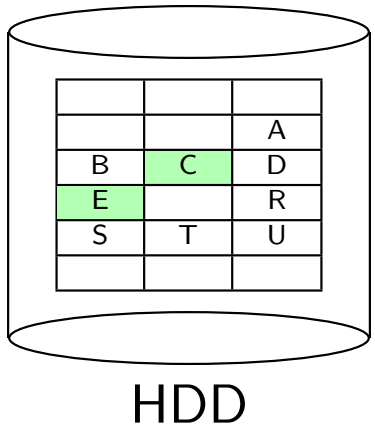
Page table of process #1

Process #1 needs to access Page 4 (E)
The kernel selects a victim frame (e.g. frame 5)
The victim is written to disk

The entry in P#2 page table is updated

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

Physical memory



0	R
1	S
2	T
3	U

Address space of process #2

P0	6	v
P1	1	v
P2		i
P3	4	v

Page table of process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4		i

Page table of process #1

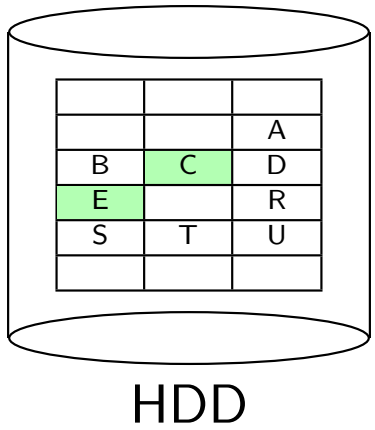
Process #1 needs to access Page 4 (E)
The kernel selects a victim frame (e.g. frame 5)
The victim is written to disk

The entry in P#2 page table is updated

The Free-Frame List is also updated

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

Physical memory



Free Frames: {5}

0	R
1	S
2	T
3	U

Address space of process #2

P0	6	v
P1	1	v
P2		i
P3	4	v

Page table of process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

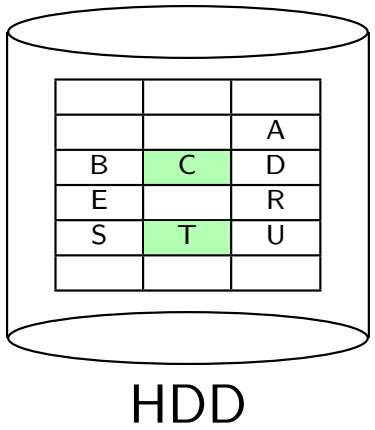
Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4	5	v

Page table of process #1

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

Physical memory



Process #1 needs to access Page 4 (E)
The kernel selects a victim frame (e.g. frame 5)
The victim is written to disk
The entry in P#2 page table is updated
The Free-Frame List is updated

E is loaded to frame 5

P1 page table is updated

Free Frames: {}

0	R
1	S
2	T
3	U

Address space of process #2

P0	6	v
P1	1	v
P2		i
P3	4	v

Page table of process #2

Page Replacement

0	A
1	B
2	C
3	D
4	E

Address space of process #1

P0	0	v
P1	3	v
P2		i
P3	2	v
P4	5	v

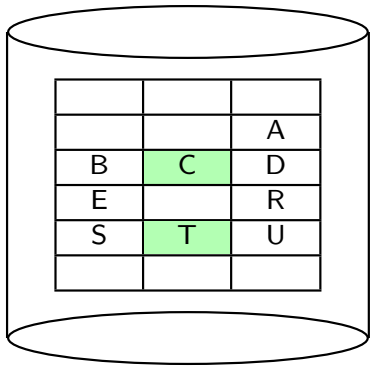
Page table of process #1

All is fine...

Until the next page fault

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

Physical memory



Process #1 needs to access Page 4 (E)
The kernel selects a victim frame (e.g. frame 5)
The victim is written to disk
The entry in P#2 page table is updated
The Free-Frame List is updated

E is loaded to frame 5

P1 page table is updated

Free Frames: {}

0	R
1	S
2	T
3	U

Address space of process #2

P0	6	v
P1	1	v
P2		i
P3	4	v

Page table of process #2

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

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

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

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? and 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...