Virtual Memory and Paging (1)

ICS332
Operating Systems

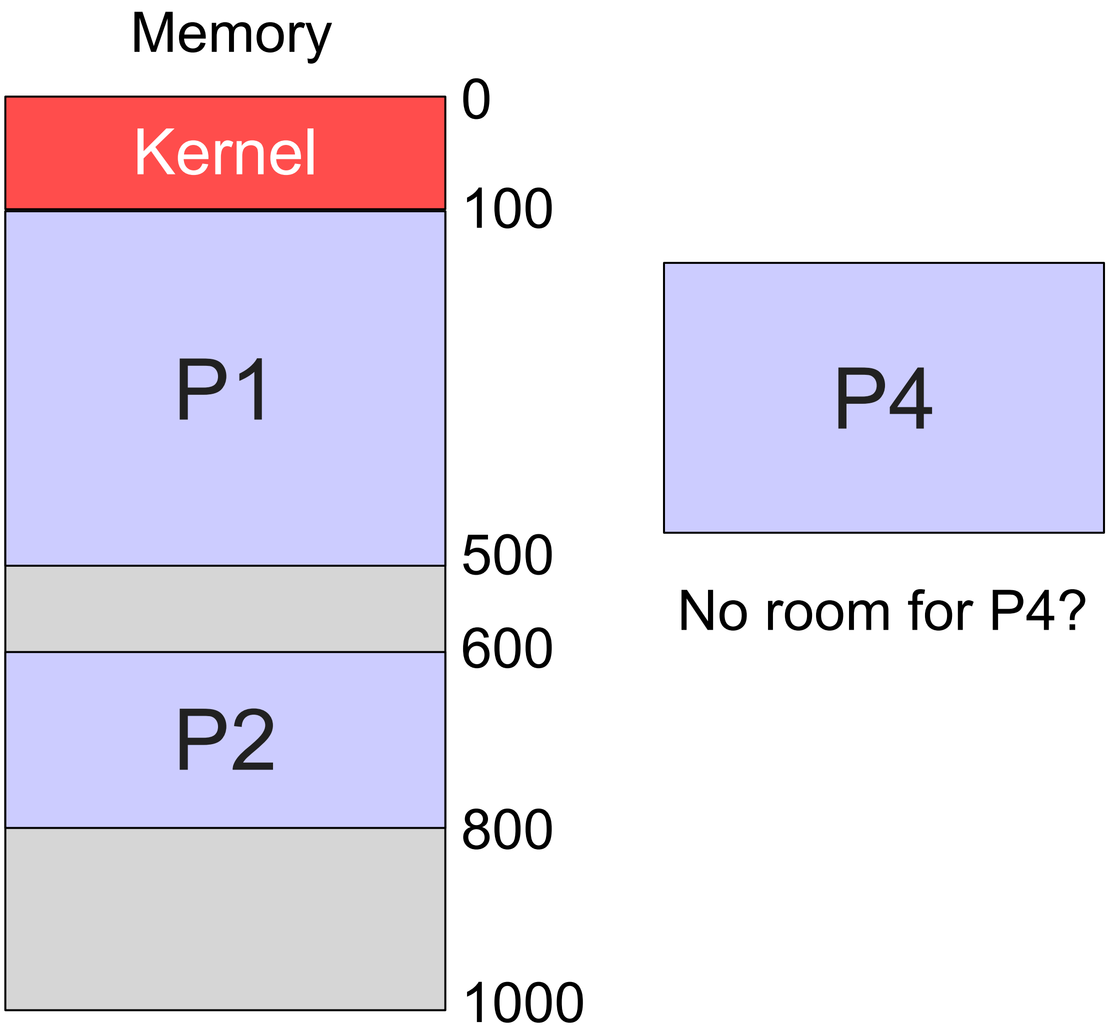
Henri Casanova (henric@hawaii.edu)

Conclusion (Previous Module)

- **Assumption so far**: Each process is in a **contiguous** address space
  - I'll assume a single segment, for simplicity ("address space" = "segment" in these lecture notes)
- 😀 Address virtualization is simple
  - Just a base register and a limit register, a comparison, an addition
- 😳 No "best" memory allocation strategies
  - First Fit, Worst Fit, Best Fit, others??
- 😡 Fragmentation can be very large
  - RAM is wasted
- 🤬 There can be process starvation in spite of sufficient available RAM due to fragmentation
  - 100 1MiB holes don't allow a 100MiB process to run!
- **Conclusion:** Our base assumption is flawed!

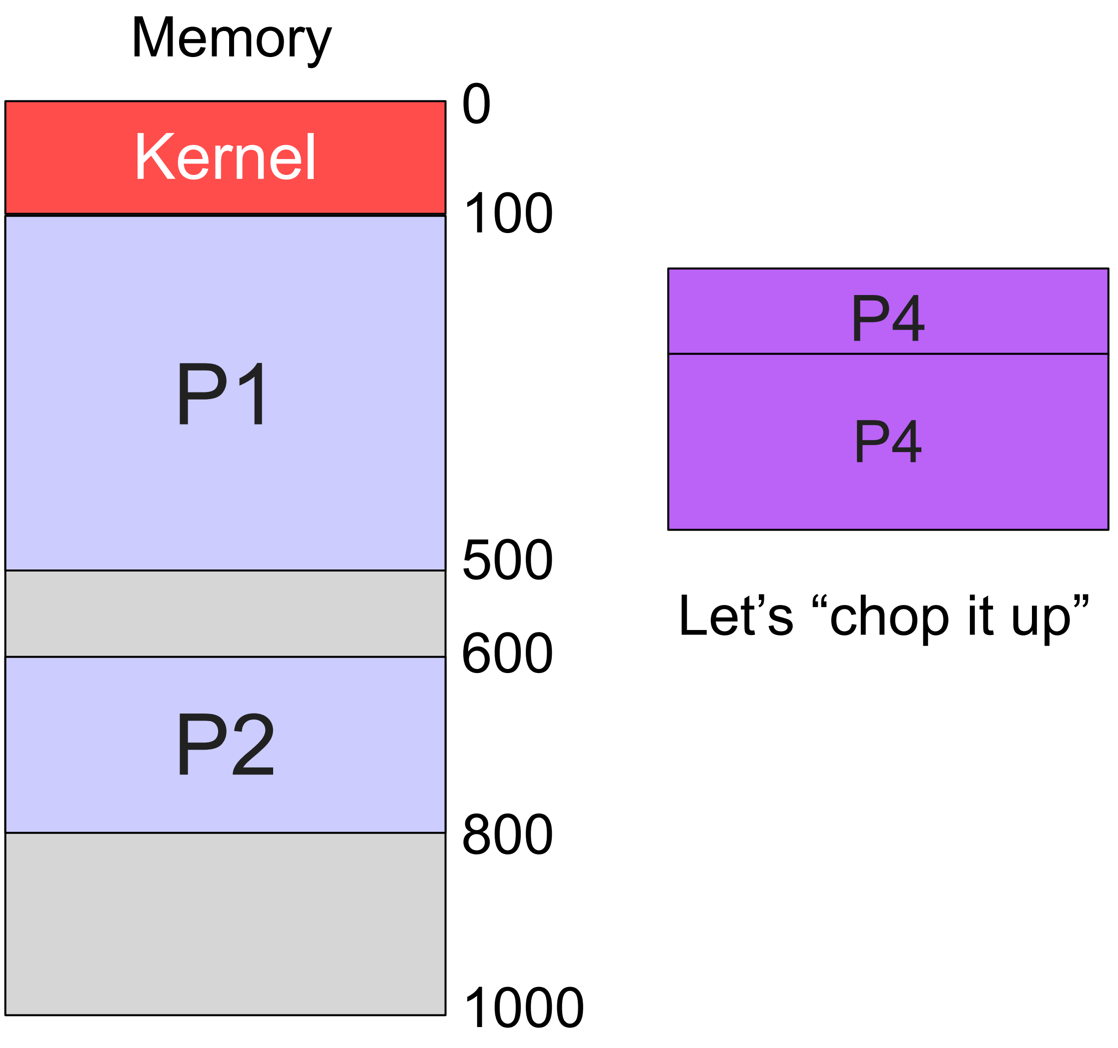
- So.... address spaces shouldn't be contiguous!?!

Non-Contiguous Address Space



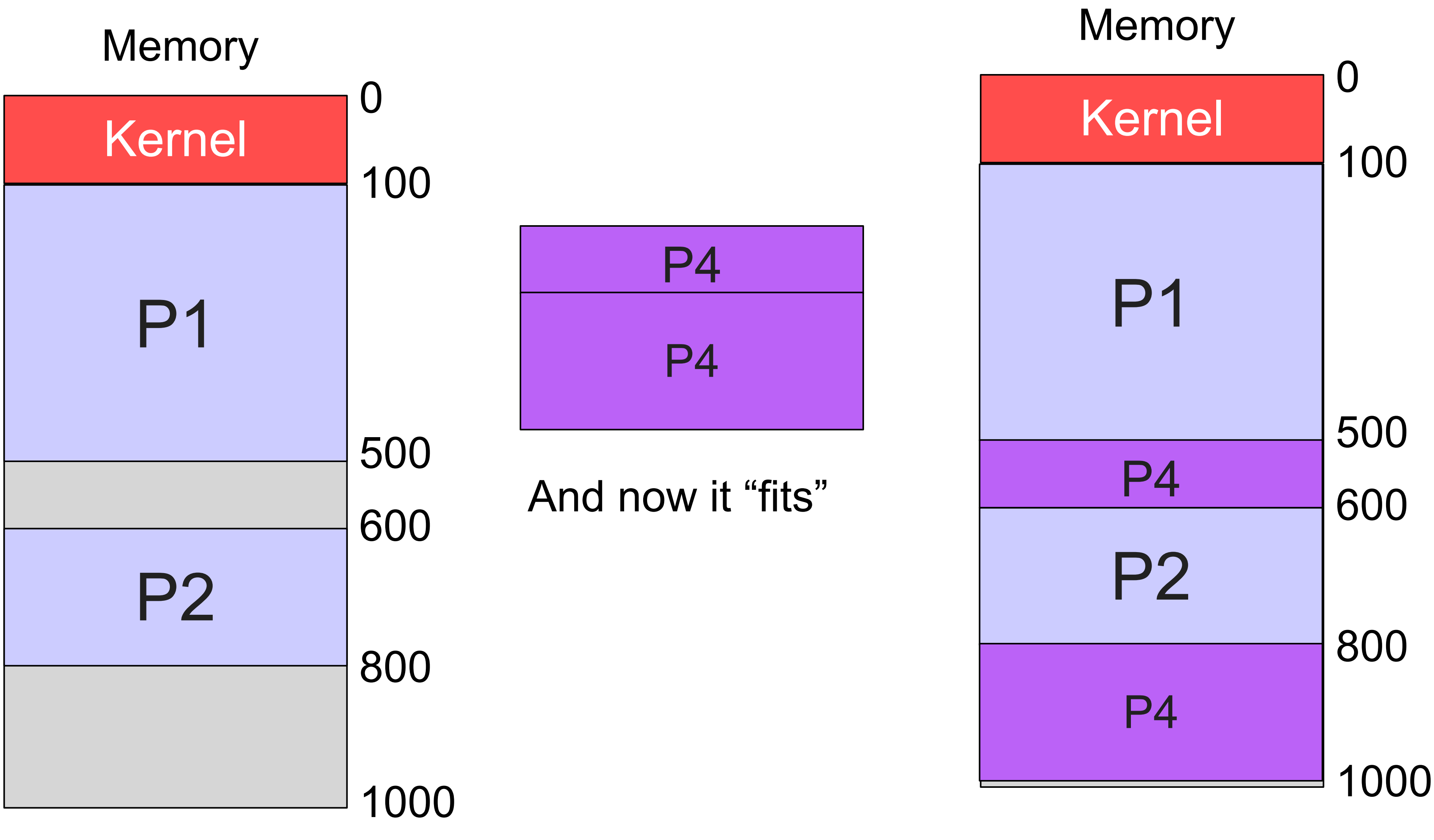
No room for P4?

Non-Contiguous Address Space



Let’s “chop it up”

Non-Contiguous Address Space

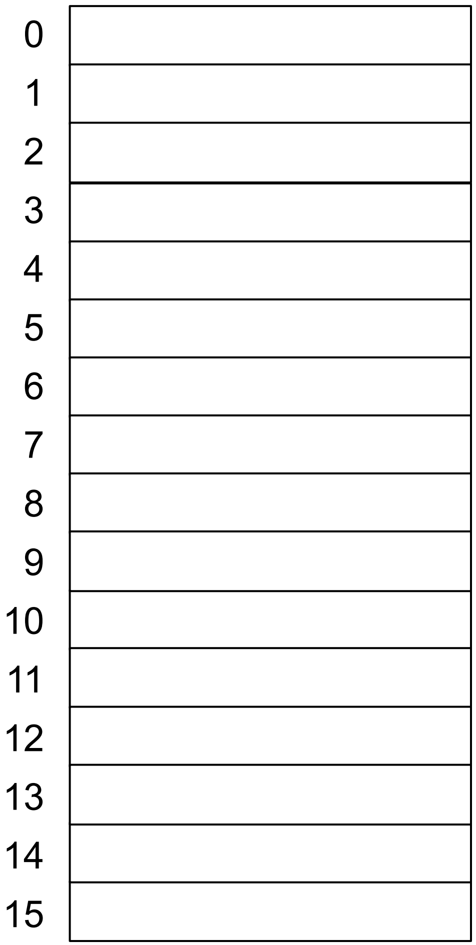


The Solution: “Paging”

- Our solution: break up address spaces into smaller chunks
- Should we have chunks of variable size like we just did on the previous example?
- Not a good idea as this is a well-known difficult problem algorithmically: Bin Packing
  - Known to be NP-hard
- We really don’t want for the OS to have to solve some NP-hard problem
- **But if chunk sizes are fixed, it all becomes easy!**
  - No longer NP-hard
- So that’s what we do!
- Each process’ address spaces in split into same-size **“pages”**
- This is called **Paging**

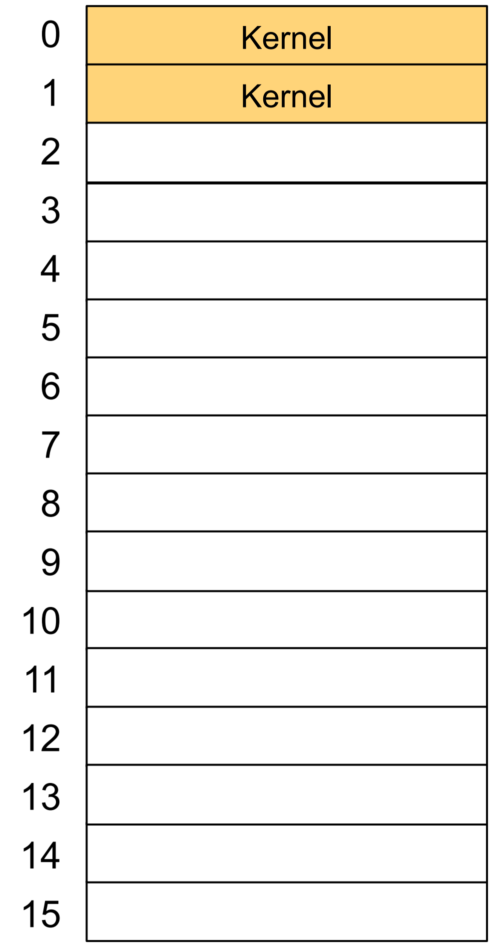
Paging

- The physical memory is split in fixed-size frames, and each frame can hold a page (frame size = page size)
- A page is “virtual” (or “logical”): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)



Paging

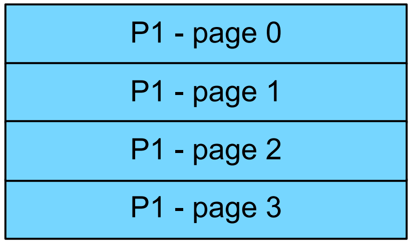
- The physical memory is split in fixed-size frames, and each frame can hold a page (frame size = page size)
- A page is “virtual” (or “logical”): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)

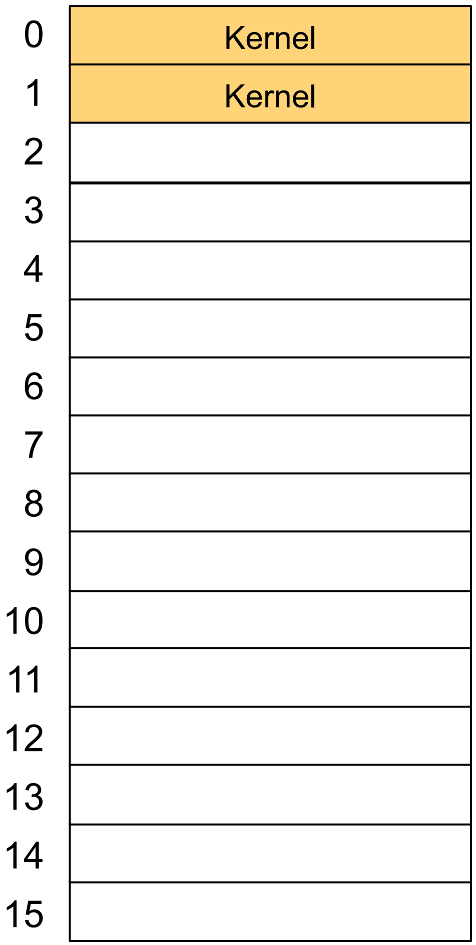


Paging

- The physical memory is split in fixed-size frames, and each frame can hold a page (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)

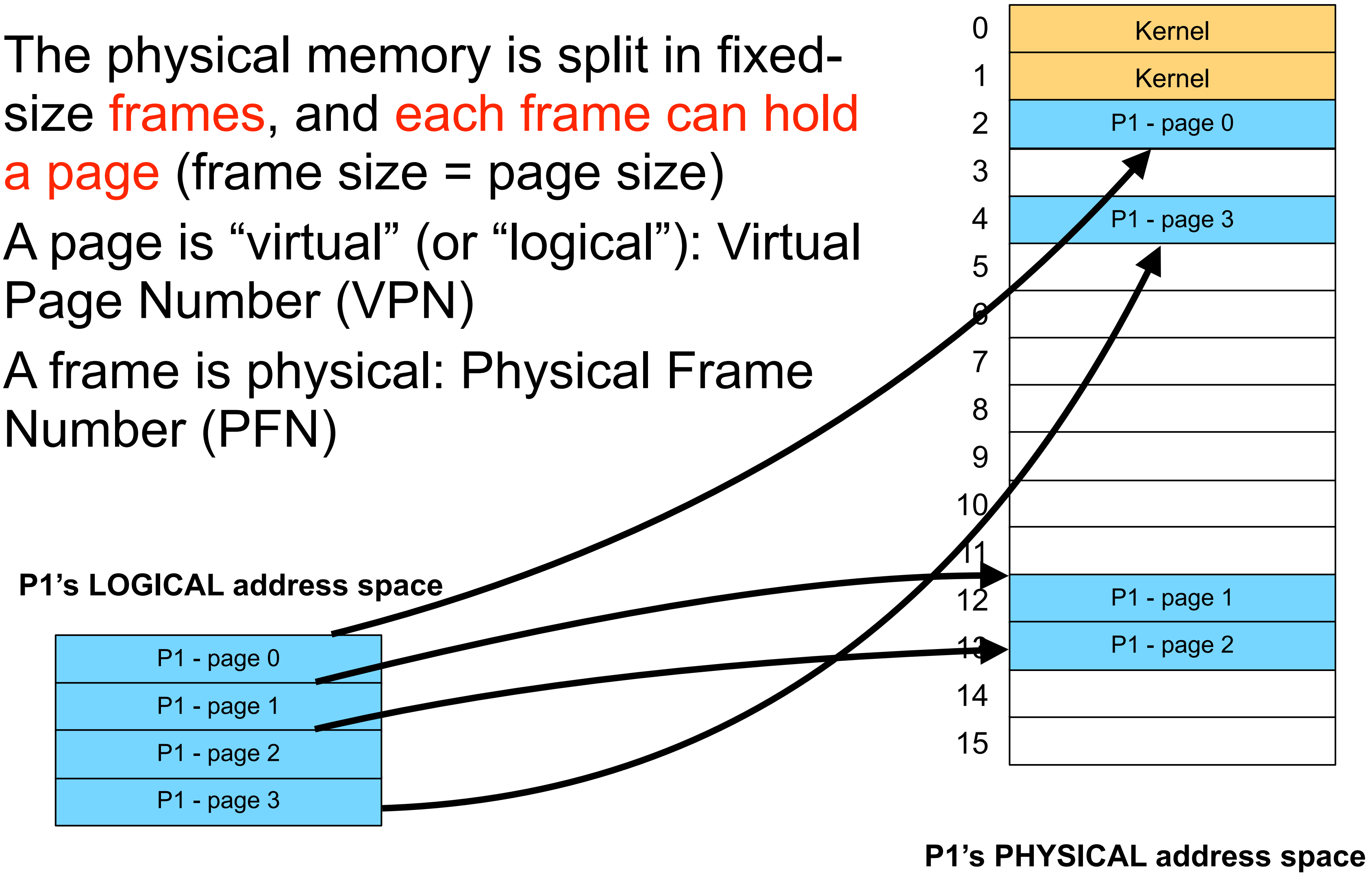
P1's LOGICAL address space





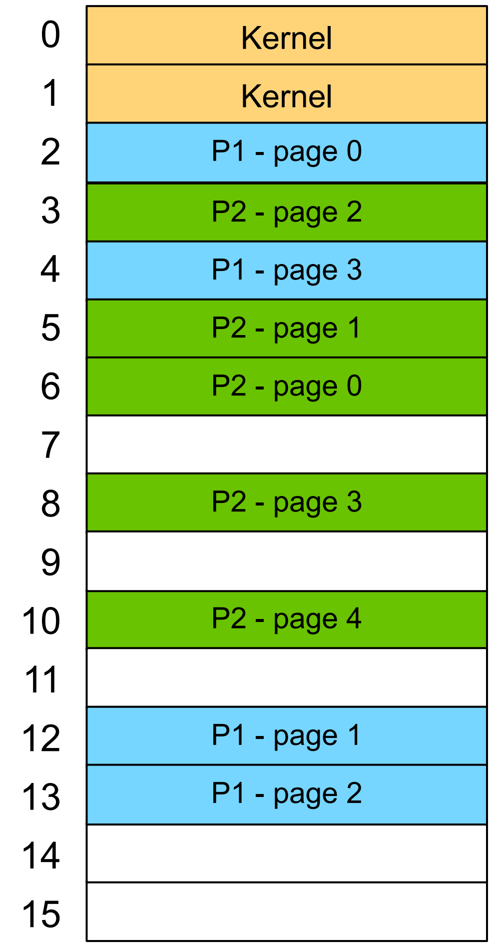
Paging

- The physical memory is split in fixed-size frames, and each frame can hold a page (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)



Paging

- The physical memory is split in fixed-size frames, and each frame can hold a page (frame size = page size)
- A page is “virtual” (or “logical”): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)

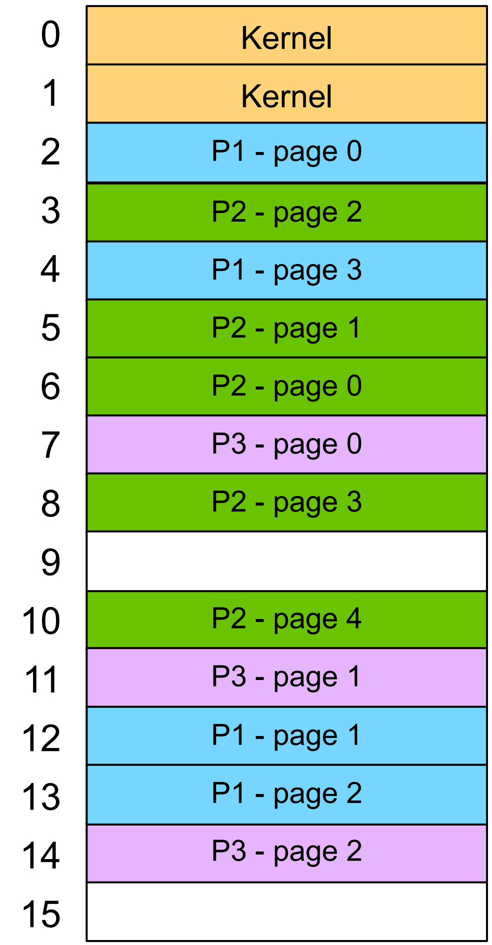


Paging

- The physical memory is split in fixed-size frames, and each frame can hold a page (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)

- And just like that, we have **non-contiguous memory allocation**

- We still have internal fragmentation, but never external fragmentation!



Paging and Addressing

- In the previous picture you see that a process' address space is non-contiguous and pages are not even in the "right order"
- When we used to say "some byte is at offset X from the beginning of the address space", now we have to say "some byte is at offset Z from the beginning of the Y-th page of the address space"
- So when we're given a logical address, we have to compute: the virtual page number and the offset within that page
- For instance, if the page/frame size is 1000 bytes, and we're talking about the 1200-th byte in the address space, then we say that the virtual page number is 1 and the offset is 200!
  - Now you see why we talked about parking lots in the Counting and Addressing module (spots are bytes, blocks of spots are pages)

Virtual Page number

- Virtual addresses issued by the CPU are split into two parts

$$\underbrace{0\;1\;1\;1\;1\;0\;1\;1}_{p\,=\,123}\;\underbrace{1\;1\;0\;1\;0\;1\;1\;0\;0\;0\;0}_{d\,=\,1712}$$

  - The virtual/logical page number: p
  - The offset within the page: d
  - “Read the value at address x” becomes “read the value at offset d in page p”

Virtual Page number

- Virtual addresses issued by the CPU are split into two parts

0	1	1	1	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0

p = 123 **d = 1712**

  - The virtual/logical page number: p
  - The offset within the page: d
  - "Read the value at address x" becomes "read the value at offset d in page p"

In the above example, how many pages can the process have?

Virtual Page number

- Virtual addresses issued by the CPU are split into two parts

0	1	1	1	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0

$\underbrace{\text{first 8 bits}}_{p\,=\,123}\;\underbrace{\text{last 11 bits}}_{d\,=\,1712}$

  - The virtual/logical page number: p
  - The offset within the page: d
  - "Read the value at address x" becomes "read the value at offset d in page p"

In the above example, how many pages can the process have?

8 bits → 2^8 = 256 pages

Virtual Page number

- Virtual addresses issued by the CPU are split into two parts

$$\underbrace{0\ 1\ 1\ 1\ 1\ 0\ 1\ 1}_{p\ =\ 123}\ \underbrace{1\ 1\ 0\ 1\ 0\ 1\ 1\ 0\ 0\ 0\ 0}_{d\ =\ 1712}$$

  - The virtual/logical page number: p
  - The offset within the page: d
  - “Read the value at address x” becomes “read the value at offset d in page p”

In the above example, how big is each page?

Virtual Page number

- Virtual addresses issued by the CPU are split into two parts

0	1	1	1	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0

p = 123 d = 1712

  - The virtual/logical page number: p
  - The offset within the page: d
  - "Read the value at address x" becomes "read the value at offset d in page p"

In the above example, how big is each page?

11 bits → 2^{11} = 2KiB in a page

Virtual Page number

- Virtual addresses issued by the CPU are split into two parts

0	1	1	1	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0

p = 123 d = 1712

  - The virtual/logical page number: p
  - The offset within the page: d
  - “Read the value at address x” becomes “read the value at offset d in page p”

- The process still has the illusion of a contiguous address space starting at page 0, continuing at page 1, etc.
- But in reality (i.e., in the physical RAM), each page is in a memory frame anywhere: We say “page p is in frame f ”

Virtual Page number

- Virtual addresses issued by the CPU are split into two parts

0	1	1	1	1	0	1	1	1	1	0	1	0	1	1	0	0	0	0

p = 123 **d = 1712**

  - The virtual/logical page number: p
  - The offset within the page: d
  - "Read the value at address x" becomes "read the value at offset d in page p"

- The process still has the illusion of a contiguous address space starting at page 0, continuing at page 1, etc.
- But in reality (i.e., in the physical RAM), each page is in a memory frame anywhere: We say "page p is in frame f "

- **Obvious Question:** how do we know in which frame a page is??

Page-to-Frame Translation

- The Virtual Page Number (VPN) has to be translated to the corresponding Physical Frame Number (PFN)
- This is more sophisticated address translation scheme than what we saw in the previous module for contiguous memory allocation
- Remember from the previous slide: instead of “read the value at address x”, a program program does “read the value at offset d in page p”
- Therefore we need to keep track for each process of the mapping between each of its pages and the physical frame that page is in

- To this end, each process has a **page table**...

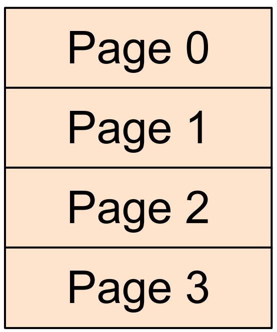
Page Table Example

- Let's consider a system where the physical memory consists of 8 frames
  - The physical memory has some size, and the OS defines the frame/page size
- Let's say the Kernel fits in one frame

F#	
0	Kernel
1	free
2	free
3	free
4	free
5	free
6	free
7	free

Physical Memory

Page Table Example

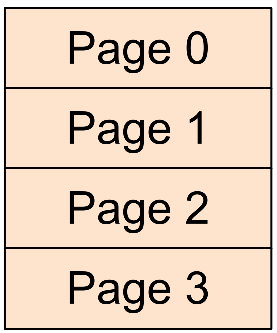


Logical Address Space

- Let's consider a process whose address space fits in 4 pages
- The OS will place these pages in some of the frames…

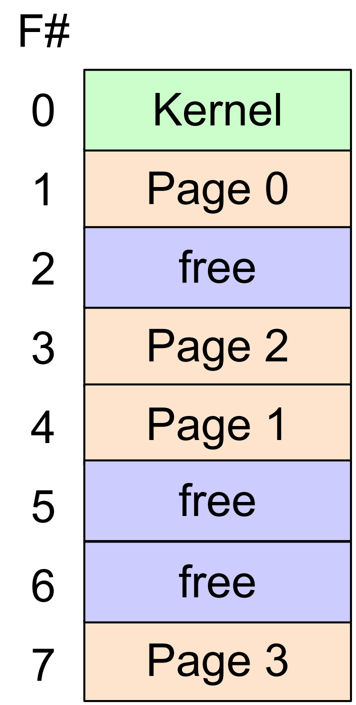
Physical Memory

Page Table Example



Logical Address Space

- Let's consider a process whose address space fits in 4 pages
- The OS will place these pages in some of the frames…
- For instance, as shown on the right
- The OS will maintain a table that maps each page # to a frame #…



Physical Memory

Page Table Example

Page 0
Page 1
Page 2
Page 3

Logical Address Space

Page	Frame
0	1
1	4
2	3
3	7

Page Table

F#	
0	Kernel
1	Page 0
2	free
3	Page 2
4	Page 1
5	free
6	free
7	Page 3

Physical Memory

Page Table Example

This entry means that page 1 is in frame 4

Page 0
Page 1
Page 2
Page 3

Logical Address Space

Page	Frame
0	1
1	4
2	3
3	7

Page Table

F#	
0	Kernel
1	Page 0
2	free
3	Page 2
4	Page 1
5	free
6	free
7	Page 3

Physical Memory

Page Size

- The page size is defined by the architecture
  - x86-64: 4 KiB, 2 MiB, and 1 GiB
  - ARM: 4 KiB, 64 KiB, and 1 MiB
- The page size in bytes is always a power of 2
- The `pagesize` command gives you the page size on UNIX-like systems
- For instance, on my laptop: 16KiB

- You can easily reconfigure your OS to use a different page size
  - But that page size has to be supported by the hardware
- We'll understand why you may want smaller/bigger pages later...

Page Size: Address Decomposition

- Say the size of the logical address space is 2^m bytes
- Say a page is 2^n bytes ($n < m$), then...

- The n low-order bits of a logical address are the offset into the page
  - offset ranges between 0 and 2^{n-1}, each one corresponding to a byte in the page

- The remaining $m - n$ high-order bits are the logical page number

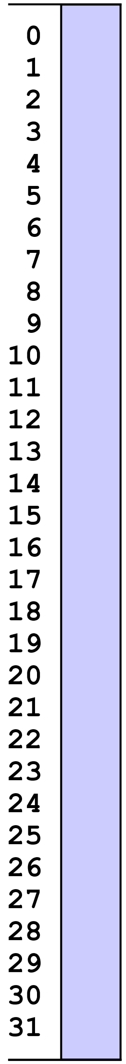
- We already saw this on an example! let's see it on another example again…

Example

- Physical memory size = 2^5 = 32 bytes

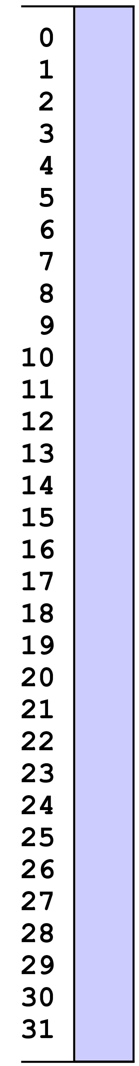
Example

- Physical memory size = 2^5 = 32 bytes
- How many bits in a physical address?



Example

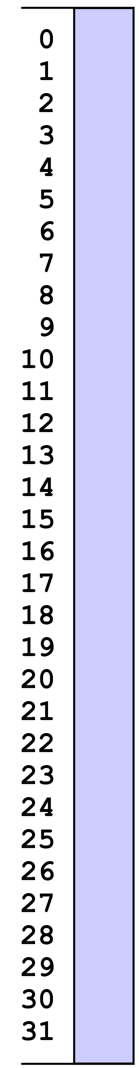
- Physical memory size = 2^5 = 32 bytes
- How many bits in a physical address?
  - How many bits are necessary to address 2^5 thingies?



Example

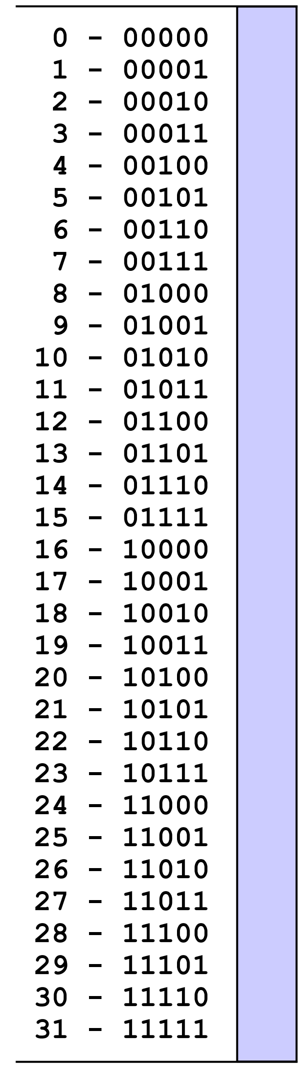
- Physical memory size = 2^5 = 32 bytes
- How many bits in a physical address?
  - How many bits are necessary to address 2^5 thingies?

    5 bits



Example

- Physical memory size = 2^5 = 32 bytes
- 5-bit physical addresses



Example

- Physical memory size = 2^5 = 32 bytes
- 5-bit physical addresses
- Say we pick frame size = 4 bytes
  - e.g., Frame #2 contains values at physical addresses 8, 9, 10, 11
- Therefore we also pick page size = 4 bytes

```
 0 - 00000
 1 - 00001
 2 - 00010
 3 - 00011
 4 - 00100
 5 - 00101
 6 - 00110
 7 - 00111
 8 - 01000
 9 - 01001
10 - 01010
11 - 01011
12 - 01100
13 - 01101
14 - 01110
15 - 01111
16 - 10000
17 - 10001
18 - 10010
19 - 10011
20 - 10100
21 - 10101
22 - 10110
23 - 10111
24 - 11000
25 - 11001
26 - 11010
27 - 11011
28 - 11100
29 - 11101
30 - 11110
31 - 11111
```

Example

- Physical memory size = 2^5 = 32 bytes
- 5-bit physical addresses
- Say we pick frame size = 4 bytes
  - e.g., Frame #2 contains values at physical addresses 8, 9, 10, 11
- Therefore we also pick page size = 4 bytes

0 - 00000		Frame 0
1 - 00001		
2 - 00010		
3 - 00011		
4 - 00100		Frame 1
5 - 00101		
6 - 00110		
7 - 00111		
8 - 01000		Frame 2
9 - 01001		
10 - 01010		
11 - 01011		
12 - 01100		Frame 3
13 - 01101		
14 - 01110		
15 - 01111		
16 - 10000		Frame 4
17 - 10001		
18 - 10010		
19 - 10011		
20 - 10100		Frame 5
21 - 10101		
22 - 10110		
23 - 10111		
24 - 11000		Frame 6
25 - 11001		
26 - 11010		
27 - 11011		
28 - 11100		Frame 7
29 - 11101		
30 - 11110		
31 - 11111		

Example

- Physical memory size = 2^5 = 32 bytes
- 5-bit physical addresses
- Say we pick frame size = 4 bytes
  - e.g., Frame #2 contains values at physical addresses 8, 9, 10, 11
- Therefore we also pick page size = 4 bytes
- How many 4-byte frames are there?

$$\frac{2^5 \text{ (bytes)}}{2^2 \text{ (bytes / frame)}} = 2^3 = 8 \text{ frames}$$

0	00000	Frame 0
1	00001	
2	00010	
3	00011	
4	00100	Frame 1
5	00101	
6	00110	
7	00111	
8	01000	Frame 2
9	01001	
10	01010	
11	01011	
12	01100	Frame 3
13	01101	
14	01110	
15	01111	
16	10000	Frame 4
17	10001	
18	10010	
19	10011	
20	10100	Frame 5
21	10101	
22	10110	
23	10111	
24	11000	Frame 6
25	11001	
26	11010	
27	11011	
28	11100	Frame 7
29	11101	
30	11110	
31	11111	

Example

- Physical memory size = 2^5 = 32 bytes
- 5-bit physical addresses
- frame / page size = 4 bytes
- Say we have a process with a 16-byte address space
  - Therefore is has 16/4 = 4 pages
- Say its bytes have values a, b, c, …

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

0 - 00000		Frame 0
1 - 00001		
2 - 00010		
3 - 00011		
4 - 00100		Frame 1
5 - 00101		
6 - 00110		
7 - 00111		
8 - 01000		Frame 2
9 - 01001		
10 - 01010		
11 - 01011		
12 - 01100		Frame 3
13 - 01101		
14 - 01110		
15 - 01111		
16 - 10000		Frame 4
17 - 10001		
18 - 10010		
19 - 10011		
20 - 10100		Frame 5
21 - 10101		
22 - 10110		
23 - 10111		
24 - 11000		Frame 6
25 - 11001		
26 - 11010		
27 - 11011		
28 - 11100		Frame 7
29 - 11101		
30 - 11110		
31 - 11111		

Example

- Physical memory size = 2^5 = 32 bytes
- 5-bit physical addresses
- frame / page size = 4 bytes
- How many bits in a virtual address for that process?

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

p #	f #
0	5
1	6
2	1
3	2

0 - 00000		Frame 0
1 - 00001		
2 - 00010		
3 - 00011		
4 - 00100	i	Frame 1
5 - 00101	j	
6 - 00110	k	
7 - 00111	l	
8 - 01000	m	Frame 2
9 - 01001	n	
10 - 01010	o	
11 - 01011	p	
12 - 01100		Frame 3
13 - 01101		
14 - 01110		
15 - 01111		
16 - 10000		Frame 4
17 - 10001		
18 - 10010		
19 - 10011		
20 - 10100	a	Frame 5
21 - 10101	b	
22 - 10110	c	
23 - 10111	d	
24 - 11000	e	Frame 6
25 - 11001	f	
26 - 11010	g	
27 - 11011	h	
28 - 11100		Frame 7
29 - 11101		
30 - 11110		
31 - 11111		

Example

- Physical memory size = 2^5 = 32 bytes
- 5-bit physical addresses
- frame / page size = 4 bytes
- How many bits in a virtual address for that process?
  - 2-bit page index (2^2 pages)
  - 2-bit offset (2^2 bytes in a page)
  - 4-bit addresses

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

p #	f #
0	5
1	6
2	1
3	2

0 - 00000		Frame 0
1 - 00001		
2 - 00010		
3 - 00011		
4 - 00100	i	Frame 1
5 - 00101	j	
6 - 00110	k	
7 - 00111	l	
8 - 01000	m	Frame 2
9 - 01001	n	
10 - 01010	o	
11 - 01011	p	
12 - 01100		Frame 3
13 - 01101		
14 - 01110		
15 - 01111		
16 - 10000		Frame 4
17 - 10001		
18 - 10010		
19 - 10011		
20 - 10100	a	Frame 5
21 - 10101	b	
22 - 10110	c	
23 - 10111	d	
24 - 11000	e	Frame 6
25 - 11001	f	
26 - 11010	g	
27 - 11011	h	
28 - 11100		Frame 7
29 - 11101		
30 - 11110		
31 - 11111		

Example

- What is the **logical** address of byte “g”?
- Logical @ = (page #) * (page size) + offset
- Page = 1, Offset = 2 (often written 1:2)
- Logical @ = 1x4 + 2 = **6**

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

p #	f #
0	5
1	6
2	1
3	2

0 - 00000		Frame 0
1 - 00001		
2 - 00010		
3 - 00011		
4 - 00100	i	Frame 1
5 - 00101	j	
6 - 00110	k	
7 - 00111	l	
8 - 01000	m	Frame 2
9 - 01001	n	
10 - 01010	o	
11 - 01011	p	
12 - 01100		Frame 3
13 - 01101		
14 - 01110		
15 - 01111		
16 - 10000		Frame 4
17 - 10001		
18 - 10010		
19 - 10011		
20 - 10100	a	Frame 5
21 - 10101	b	
22 - 10110	c	
23 - 10111	d	
24 - 11000	e	Frame 6
25 - 11001	f	
26 - 11010	g	
27 - 11011	h	
28 - 11100		Frame 7
29 - 11101		
30 - 11110		
31 - 11111		

Example

- What is the **physical** address of byte “g”?
- Physical @ = (**frame #**) * (page size) + offset
- Page = 1 is in Frame 6
- Same Offset = 2
- Physical @ = 6x4 + 2 = **26**

0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

p #	f #
0	5
1	6
2	1
3	2

0 - 00000		Frame 0
1 - 00001		
2 - 00010		
3 - 00011		
4 - 00100	i	Frame 1
5 - 00101	j	
6 - 00110	k	
7 - 00111	l	
8 - 01000	m	Frame 2
9 - 01001	n	
10 - 01010	o	
11 - 01011	p	
12 - 01100		Frame 3
13 - 01101		
14 - 01110		
15 - 01111		
16 - 10000		Frame 4
17 - 10001		
18 - 10010		
19 - 10011		
20 - 10100	a	Frame 5
21 - 10101	b	
22 - 10110	c	
23 - 10111	d	
24 - 11000	e	Frame 6
25 - 11001	f	
26 - 11010	g	
27 - 11011	h	
28 - 11100		Frame 7
29 - 11101		
30 - 11110		
31 - 11111		

In-class Exercise (1)

- A computer has 4 GiB of RAM with a page size of 8KiB. Processes have 1 GiB address spaces.
  - How many bits are used for physical addresses
  - How many bits are used for logical addresses
  - How many bits are used for logical page numbers?

In-class Exercise (1)

- A computer has 4 GiB of RAM with a page size of 8KiB. Processes have 1 GiB address spaces.
  - How many bits are used for physical addresses

    Physical RAM: 4 GiB = 2^{32} bytes

    → 32-bit physical addresses
  - How many bits are used for logical addresses

    Logical address space: 1 GiB = 2^{30} bytes

    → 30-bit physical addresses
  - How many bits are used for logical page numbers?

    Page size = 2^{13} bytes

    Number of pages in logical address space: $2^{30}/2^{13} = 2^{17}$

    → 17-bit logical page numbers
    (and 13-bit offsets)

In-class Exercise (2)

- Logical addresses are 44-bit, and a process can have up to 2^{27} pages. What is the page size?

In-class Exercise (2)

- Logical addresses are 44-bit, and a process can have up to 2^{27} pages. What is the page size?

The address space can have up to 2^{44} bytes

There are up to 2^{27} pages

Therefore, a page is $2^{44} / 2^{27} = 2^{17}$ bytes

In-class Exercise (3)

- On my computer the page size is 16 KiB, and my process' address space is 4GiB.
- How many bits are used for the page number in a logical address?

In-class Exercise (3)

- On my computer the page size is 16 KiB, and my process' address space is 4GiB.
- How many bits are used for the page number in a logical address?

The address space contains 2^{32} bytes

A page is 2^{14} bytes

Therefore, my address space has $2^{32}/2^{14} = 2^{18}$ pages

Therefore, we need 18 bits for the page number in a logical address (and we have 14 bits in the offset)

In-class Exercise (4)

- A computer has 32-bit physical addresses. The logical page number in a logical address is 14-bit. A process can have up to a 2GiB address space
- Let's consider a process with currently a 1GiB address space (i.e., it could get up to another 1GiB during execution).
- What is the page size?
- How many entries in the process' page table currently point to pages that are part of the address space?

In-class Exercise (4)

- A computer has 32-bit physical addresses. The logical page number in a logical address is 14-bit. A process can have up to a 2GiB address space
- Let's consider a process with currently a 1GiB address space (i.e., it can get up to another 1GiB during execution).
- What is the page size?

Bytes in 2GiB (the max address space): 2^{31}
Therefore: 31-bit logical addresses
Therefore: 31 - 14 = 17-bit offsets
Therefore: 2^{17} bytes in a page
Therefore: 128KiB pages

- How many entries in the process' page table currently point to pages that are part of the address space?

The process has a 1GiB = 2^{30}-byte address space
Number of pages in the address space: $2^{30}/2^{17} = 2^{13}$
Therefore: there are 2^{13} entries in the page table, each pointing to one page

In-class Exercise (5)

- Logical addresses are 40-bit, and a process can use at most 1/4 of the physical RAM.
  - How big is the RAM?
  - A process has at most 2^{22} pages on this system. How many bits are used for the “offset” part of logical addresses?

In-class Exercise (5)

- Logical addresses are 40-bit, and a process can use at most 1/4 of the physical RAM.
  - How big is the RAM?

With 40-bit logical addresses, an address space is at most 2^{40} bytes

So the RAM is 4 times as big: 2^{42} bytes, which is 4TiB

  - A process has at most 2^{22} pages on this system. How many bits are used for the "offset" part of logical addresses?

Since we have 2^{22} pages, 22 bits are used for the page number

Therefore 40 - 22 = 18 bits are used for the offset

In-class Exercise (6)

- Consider a system with 4-byte pages. A process has the following entries in its page table:

logical	physical
0	4
1	5
2	30

- What is the physical address of the byte with logical address 2?
- What is the physical address of the byte with logical address 9?

In-class Exercise (6)

- Consider a system with 4-byte pages. A process has the following entries in its page table:

logical	physical
0	4
1	5
2	30

- What is the physical address of the byte with logical address 2?
- The byte with logical address 2 is the 3rd byte in page 0 (because that page contains the bytes at addresses 0, 1, 2, and 3)
- Page 0, according to the page table is in physical frame 4
- The first byte of physical frame 0 is at physical address 4 × 0 = 0 (the first byte in physical RAM)
- The first byte of physical frame 1 is at physical address 4 × 1 = 4 (the fifth byte in physical RAM)
- …
- The first byte of physical frame 4 is at physical address 4 × 4 = 16
- The 3rd byte of physical frame is thus at address 16 + 2
- Therefore, the byte at logical address 2 is at physical address 18

In-class Exercise (6)

- Consider a system with 4-byte pages. A process has the following entries in its page table:

logical	physical
0	4
1	5
2	30

- What is the physical address of the byte with logical address 9?
- The byte with logical address 9 is in page 9 / 4 = 2 (integer division)
- Its offset on that page is 9 % 4 = 1
- Page 2 is in frame 30
- Therefore, the byte at logical address 9 is at physical address 30 x 4 + 1 = 121

Generalization

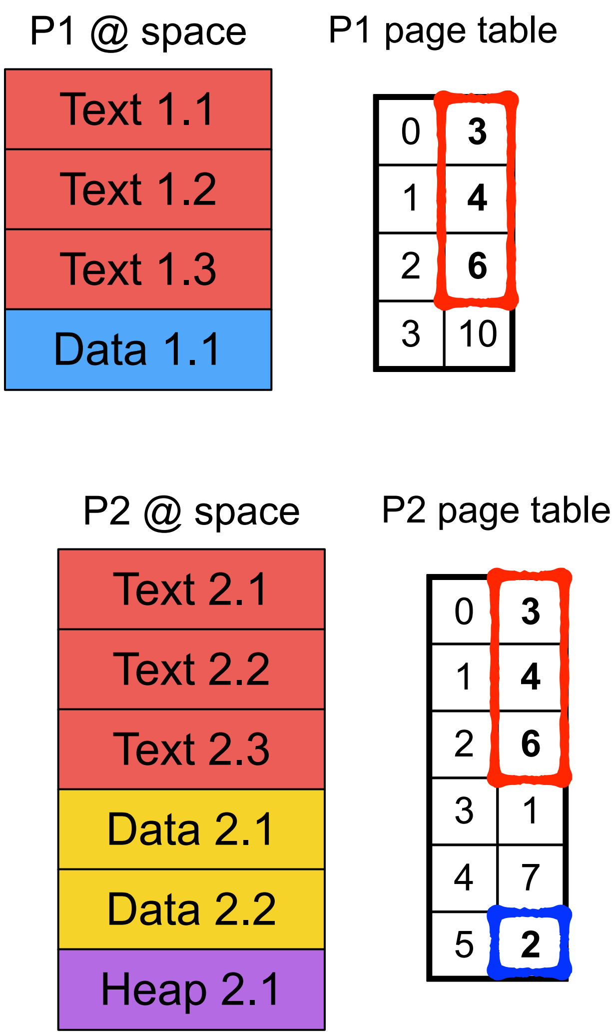
- If the page size is s
- If the logical address is x
- Then:
  - the logical page number is p = floor(x / s)
  - the offset is o = x mod s

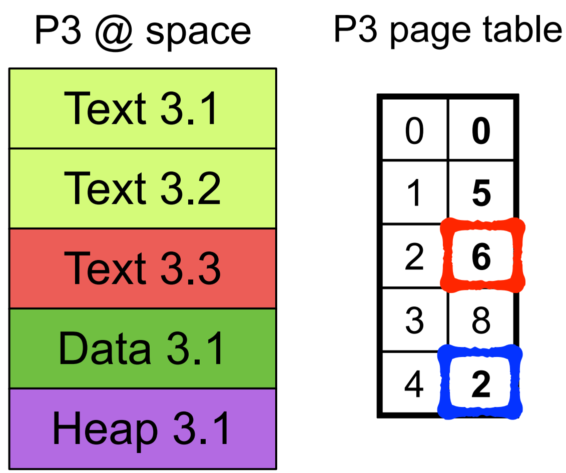
- If page p is in frame f
- Then:
  - logical address x translates to physical address y = f * s + o

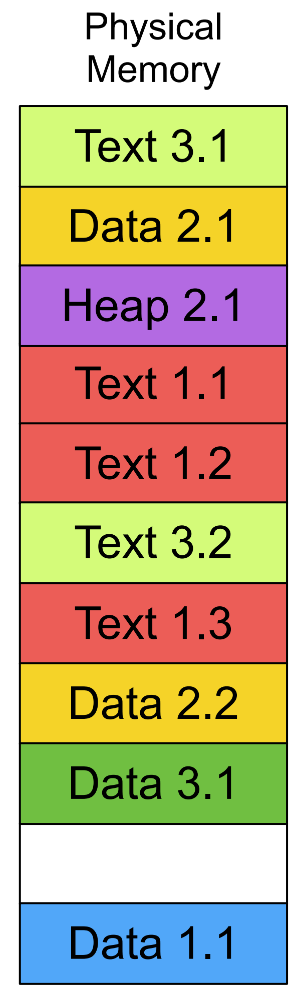
Sharing Memory Pages

- Time and again we've talked about processes sharing memory
  - Using shared memory IPC
  - With dynamic linking
- It breaks the memory protection abstraction, but it's useful
- Now that we have paging, and that each process has a page table, there is a very simple mechanism to share memory!
- Just create page table entries that point to the same physical frame in different page tables
- Let's see it on a picture…

Sharing Memory Pages - EASY!







- P1 and P2 share all their text pages (likely invocations of the same program)
- P3 shares one page of its text with P1 and P2 (likely a dynamically linked library, e.g., the code of printf)
- P2 and P3 share one page of heap (likely a shared memory segment)

Pages Not Allocated (yet)

- So far, we've shown page tables like this:

Page	Frame
0	1
1	4
2	3
3	7

- But in fact, a page table contains entries for all possible pages (up to the maximum allowed number of pages for a process, as defined by the OS

Page	Frame
0	1
1	4
2	3
3	7
4	Not used (yet)
5	Not used (yet)
6	Not used (yet)
7	Not used (yet)

Valid Bit

- Each page entry is augmented by a valid bit
- Set to valid if the process is allowed to access the page (i.e., if the page in the process address space)
- Set to invalid otherwise
- So page tables look like this:

Page	Frame	Valid
0	1	✓
1	4	✓
2	3	✓
3	7	✓
4	xx	x
5	xx	x
6	xx	x
7	xx	x

- If the process references a page whose entry's valid bit is not set, then a trap is generated (and the OS will likely terminate the process)

What about Fragmentation?

- No external fragmentation!!
  - This is of course the HUGE advantage of paging
- Only internal fragmentation
  - Worst case: A process address space is n pages plus 1 byte
    - In this case, we waste 1 page minus 1 byte
  - Average case: Uniform distribution of address space sizes: 50%
    - i.e., on average we waste 1/2 page per process
- Using smaller pages reduces internal fragmentation
- But large pages have advantages:
  - Smaller page tables (and less frequent page table lookups)
  - Loading one large page from disk takes less time than loading many small ones
- Typical sizes: 4KiB, 8KiB, 16KiB
- Modern OSes: multiple page sizes supported (Linux: Huge pages; Mac: Superpages; Windows: Large pages) through hardware

Frames Management

- The OS needs to keep track of the frames
  - Which frames are used (and by which processes?)
  - Which frames are free?
- The OS thus has a data structure: the free frame list
- Much simpler than a list of holes with different sizes
  - As done for contiguous memory allocation in the previous “Main Memory” module
- When a process needs a frame, then the OS takes a frame from the free frame list and allocates them to a process

Free frame list = {13, 14, 15, 18, 20}

	13	14	15			18		20	

Process creation: P1 needs 4 pages

Free frame list = {15}

	P1.0	P1.1	15			P1.3		P1.2	

P1’s page table

Page	Frame
0	13
1	14
2	20
3	18

Aside: Memory-Mapped Files

- I/O is very expensive
  - Each access to a file requires a disk seek and a disk access
  - Out of question to read/write bytes one by one to a file
- On-disk address spaces are brought into RAM and virtualized
- Data files can be virtualized **the same way**, i.e., by mapping them to memory
- Memory mapping
  - Map disk block(s) to a memory frame(s)
  - Initial access is expensive (and generates page faults)
  - Subsequent access is made in memory (and cheap-er)
  - The on-disk file may be updated at a convenient time, upon closing…
  - Memory mapping is performed by dedicated system calls (mmap)
  - Potential concurrency issues: multiple processes can map the same file concurrently
- Let's look at the man page for **`mmap`**

Conclusion

- Paging is great!
  - No external fragmentation
  - Easy to share pages among processes
- Mechanisms:
  - Each process as a page table
  - Each page table entry maps a logical page to a physical frame
  - Each page table entry has a valid bit
  - Address translation is based on the page table
  - The OS manages the list of free frames, and gives out frames to processes
- In the next set of lecture notes, we look at some challenges with paging and how we deal with them...