Virtual Memory — Paging I

ICS332 — Operating Systems

Henri Casanova (henric@hawaii.edu)

Spring 2018

Conclusion (of the previous module)

- Assumption: Each process is in a contiguous address space
- **Good**: Address virtualization is simple (base register)

Conclusion (of the previous module)

- Assumption: Each process is in a contiguous address space
- **Good**: Address virtualization is simple (base register)
- **Bad**: No "best" memory allocation strategies
  - First Fit Worst Fit, Best Fit, others??

Conclusion (of the previous module)

- Assumption: Each process is in a contiguous address space
- **Good**: Address virtualization is simple (base register)
- **Bad**: No "best" memory allocation strategies
  - First Fit Worst Fit, Best Fit, others??
- **Worse**: Fragmentation can be very large
  - RAM is wasted

Conclusion (of the previous module)

- Assumption: Each process is in a contiguous address space
- **Good**: Address virtualization is simple (base register)
- **Bad**: No "best" memory allocation strategies
  - First Fit Worst Fit, Best Fit, others??
- **Worse**: Fragmentation can be very large
  - RAM is wasted
- **Even Worse**: 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!

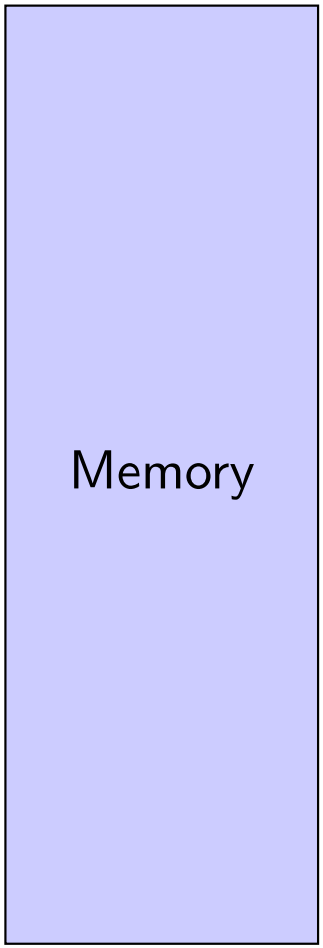
Conclusion (of the previous module)

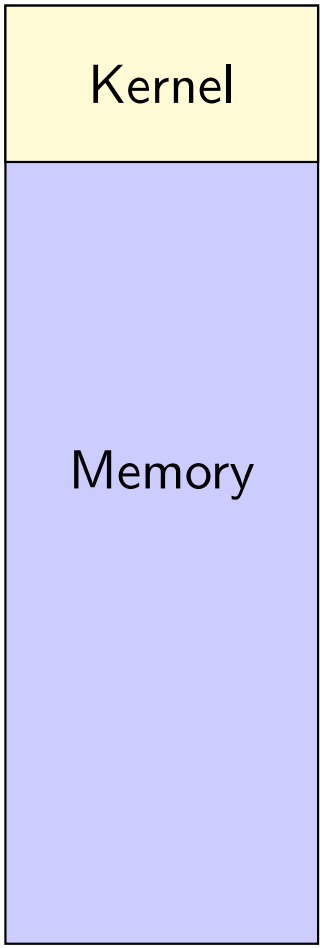
- Assumption: Each process is in a contiguous address space
- **Good**: Address virtualization is simple (base register)
- **Bad**: No "best" memory allocation strategies
  - First Fit Worst Fit, Best Fit, others??
- **Worse**: Fragmentation can be very large
  - RAM is wasted
- **Even Worse**: There 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!

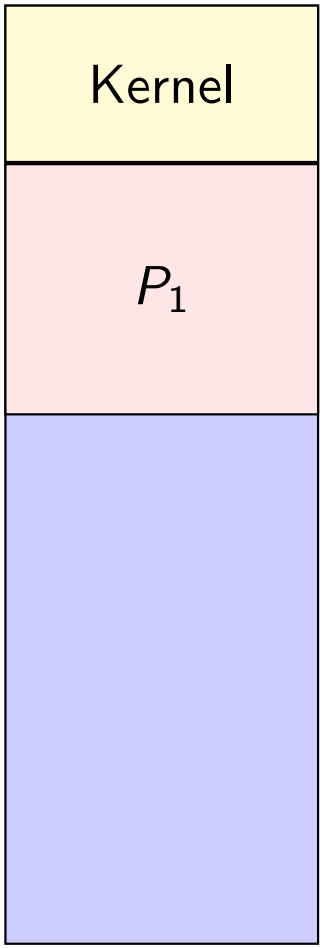
Conclusion (of the previous module)

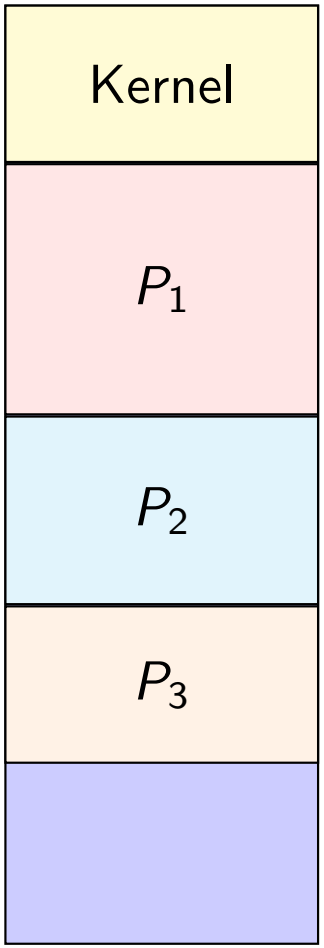
- Assumption: Each process is in a contiguous address space
- **Good**: Address virtualization is simple (base register)
- **Bad**: No "best" memory allocation strategies
  - First Fit Worst Fit, Best Fit, others??
- **Worse**: Fragmentation can be very large
  - RAM is wasted
- **Even Worse**: There 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!

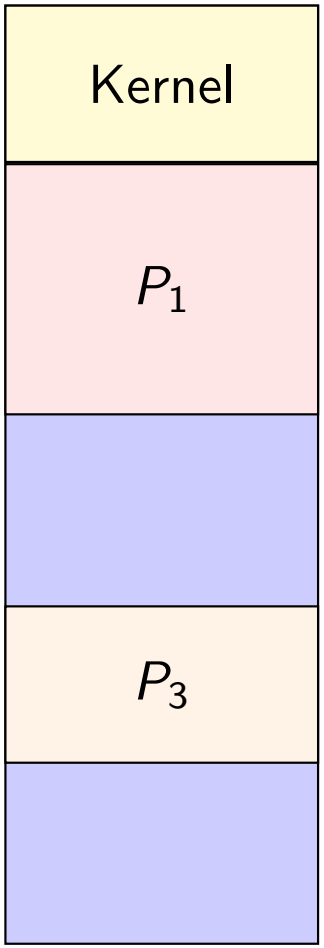
- So.... address spaces shouldn't be contiguous???

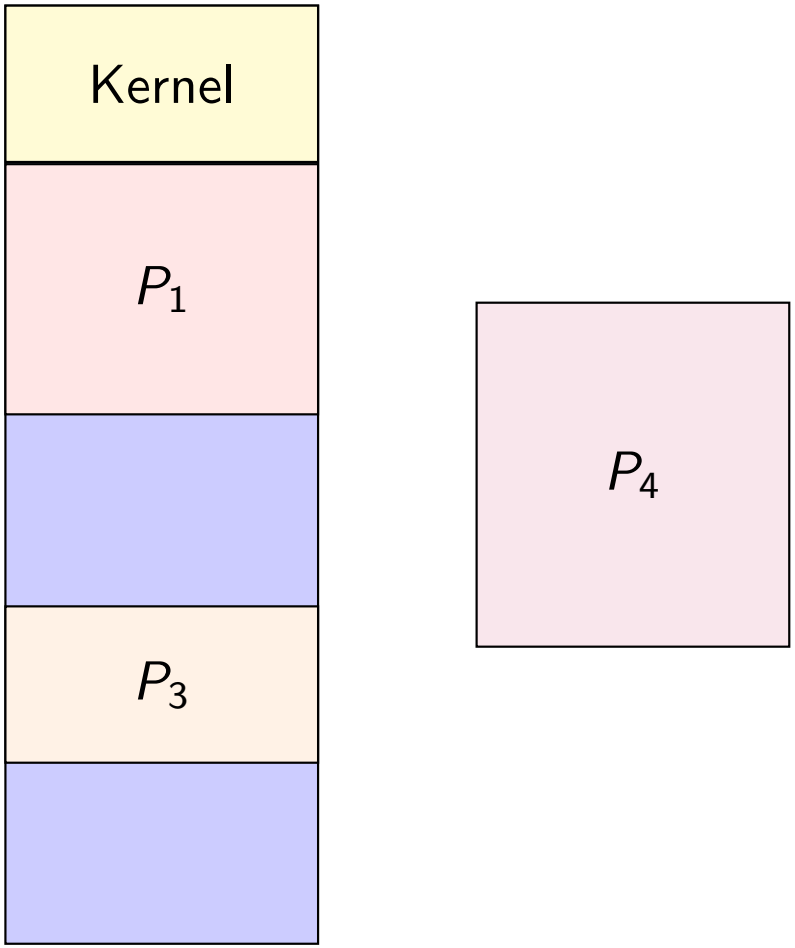

Memory

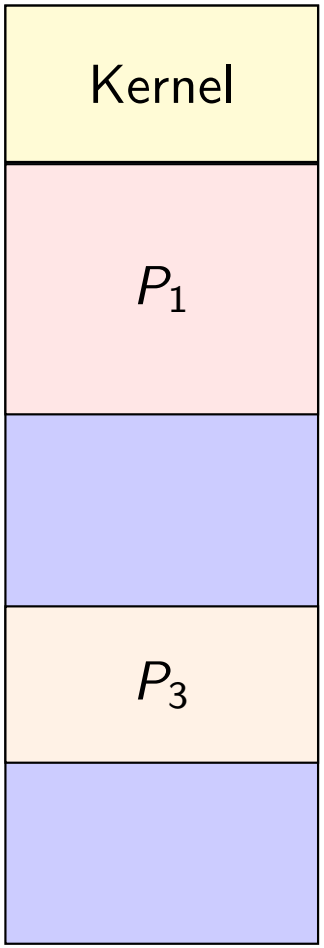

Kernel
Memory

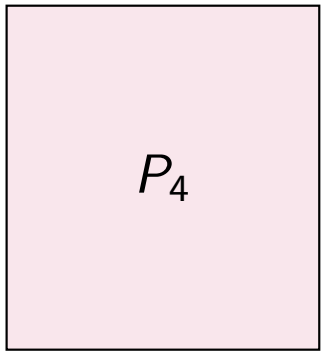

Kernel
P_1

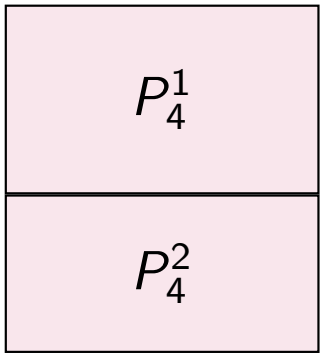

Kernel
P_1
P_2
P_3


Kernel
P_1
P_3

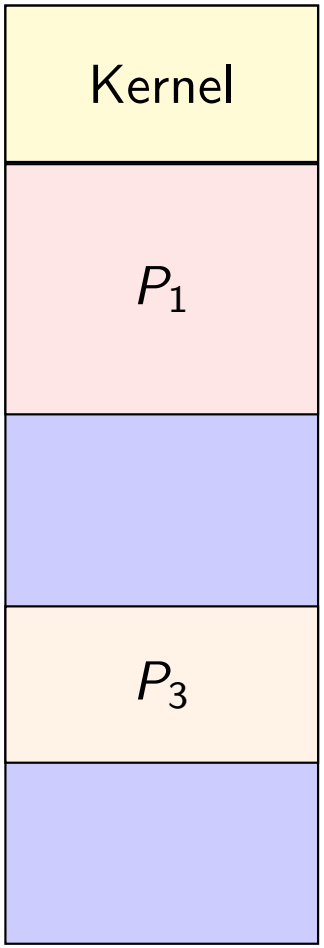

Kernel
P_1
P_4
P_3

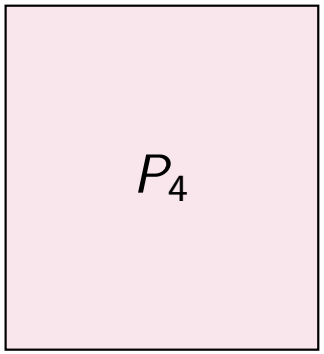


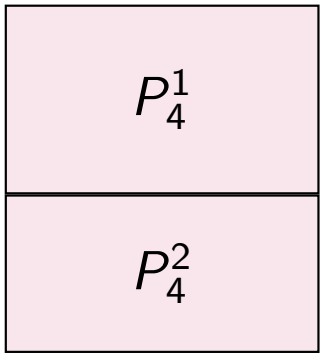




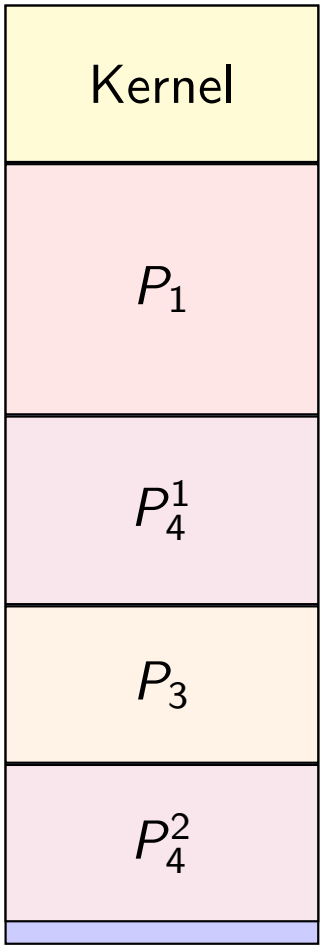
No room for P_4?

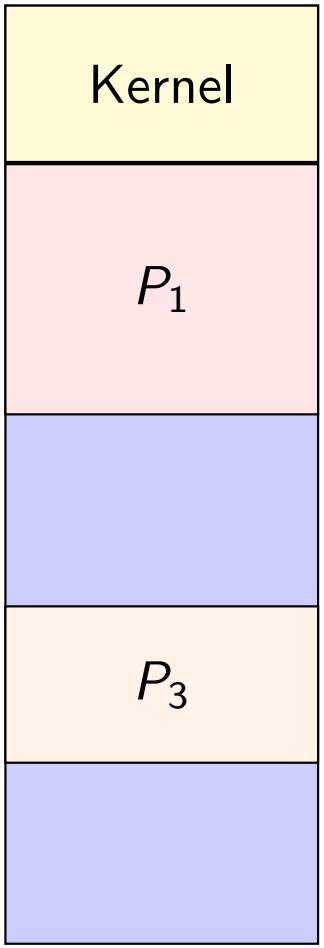


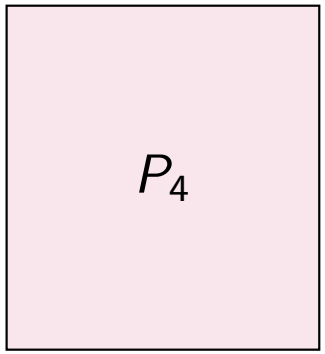


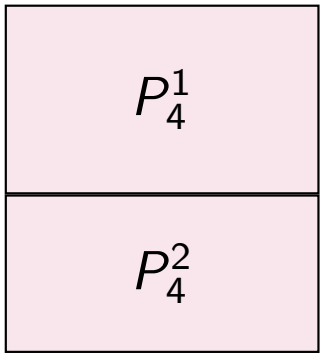


No room for P_4?

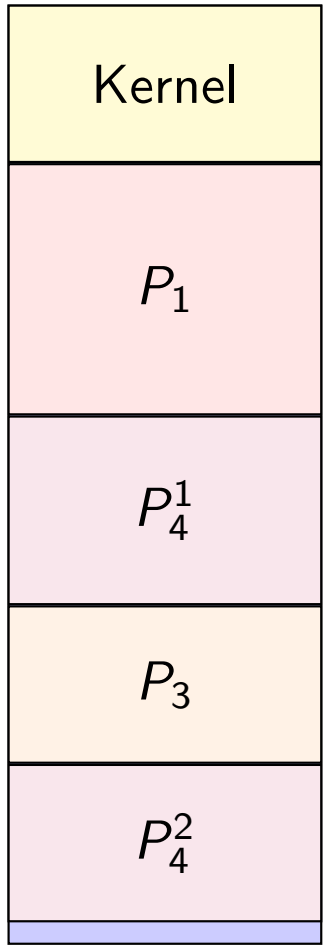








No room for P_4?



!

There **is** enough room for P_4 if we "chop it up"!

The solution

- The solution: Break up the process address spaces into smaller chunks!

The solution

- The solution: Break up the process address spaces into smaller chunks!
- Chunks of variable size?

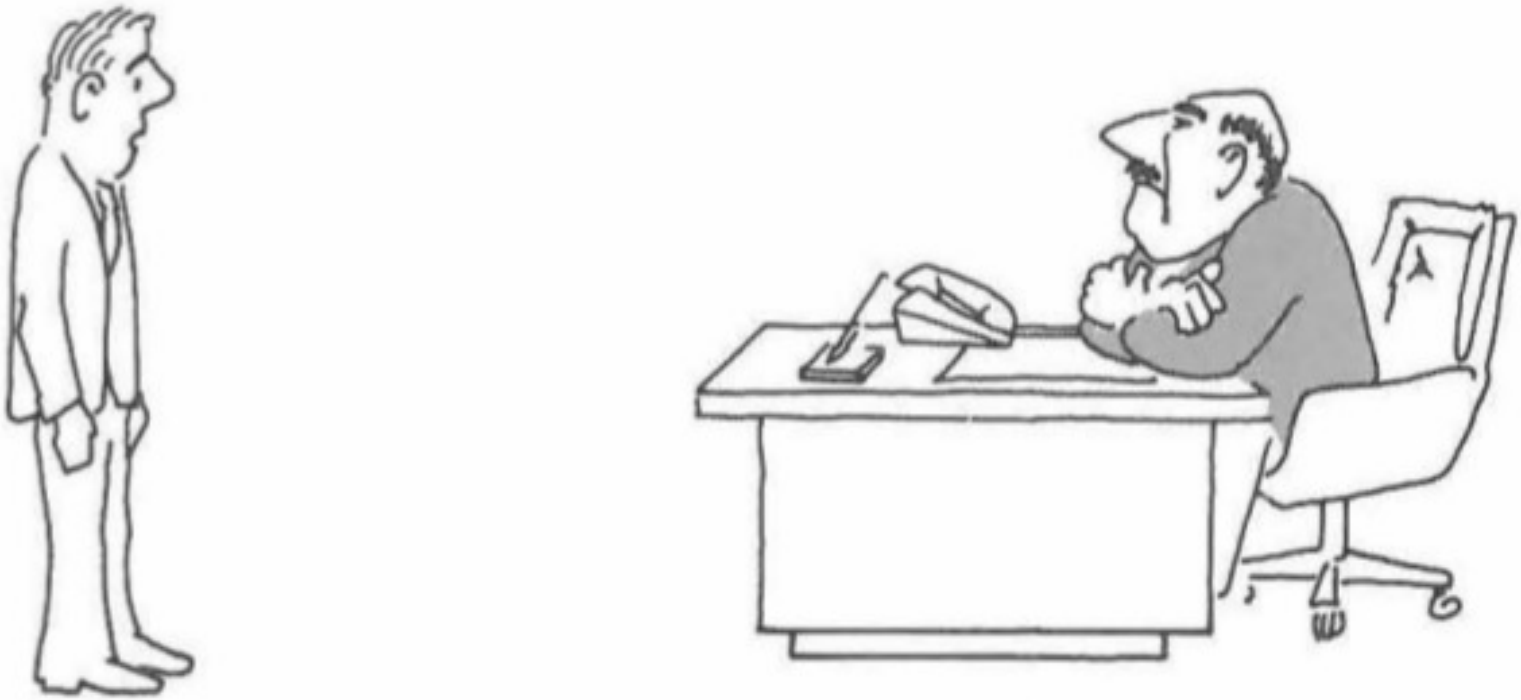
The solution

- The solution: Break up the process address spaces into smaller chunks!
- Chunks of variable size?
  - Well-known problem in Computer Science: Bin Packing

The solution

- The solution: Break up the process address spaces into smaller chunks!
- Chunks of variable size?
  - Well-known problem in Computer Science: Bin Packing
  - Known to be NP-Hard...

What you don't want to say



"I can't find an efficient algorithm, I guess I'm just too dumb."

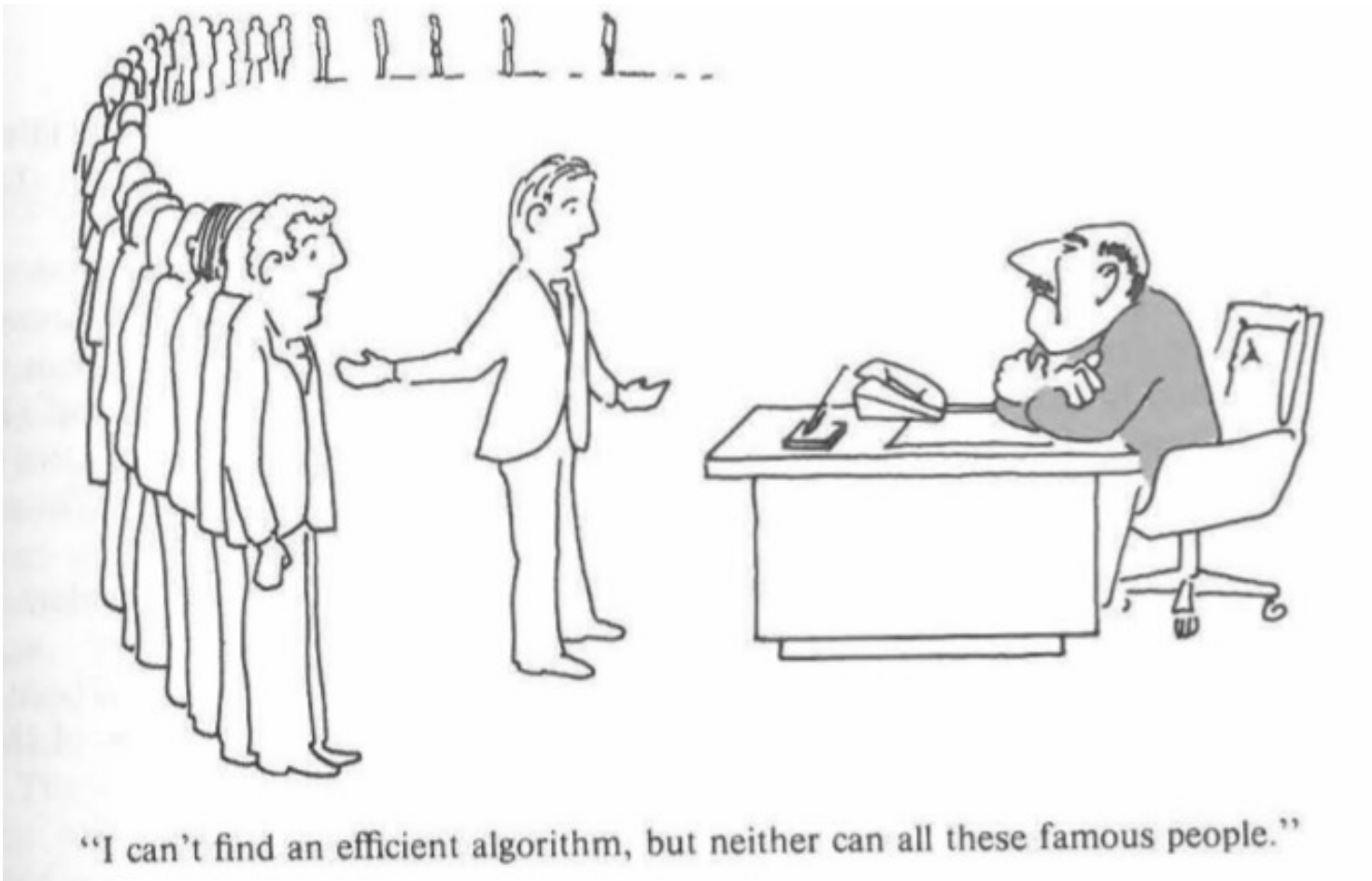
From "Computers and Intractability: : A Guide to the Theory of NP-Completeness", Garey M.R. and Johnson, D.S.; W.H. Freeman and Co Publisher, 1979. ISBN 0-7167-1045-5

What you wish you could say

"I can't find an efficient algorithm, because no such algorithm is possible!"

From "Computers and Intractability: : A Guide to the Theory of NP-Completeness", Garey M.R. and Johnson, D.S.; W.H. Freeman and Co Publisher, 1979. ISBN 0-7167-1045-5

What you can say for an NP-hard problem



From "Computers and Intractability: : A Guide to the Theory of NP-Completeness", Garey M.R. and Johnson, D.S.; W.H. Freeman and Co Publisher, 1979. ISBN 0-7167-1045-5

Computational Complexity in One Slide

- P problem: A decision problem where the "yes" or "no" answer can be decided in polynomial time (= polynomial number of operations relative to the input size), e.g., is 20 the maximum value in an array of n integers?

Computational Complexity in One Slide

- P problem: A decision problem where the "yes" or "no" answer can be decided in polynomial time (= polynomial number of operations relative to the input size), e.g., is 20 the maximum value in an array of n integers?
- NP problem: An optimization problem where a solution can be verified in polynomial time, e.g., traveling salesman (No known polynomial-time algorithm to compute the route, but easy to check whether a route is a solution)

Computational Complexity in One Slide

- P problem: A decision problem where the "yes" or "no" answer can be decided in polynomial time (= polynomial number of operations relative to the input size), e.g., is 20 the maximum value in an array of n integers?
- NP problem: An optimization problem where a solution can be verified in polynomial time, e.g., traveling salesman (No known polynomial-time algorithm to compute the route, but easy to check whether a route is a solution)
- NP-hard problem: problem which is at least as hard as the hardest NP problems.

Computational Complexity in One Slide

- P problem: A decision problem where the "yes" or "no" answer can be decided in polynomial time (= polynomial number of operations relative to the input size), e.g., is 20 the maximum value in an array of n integers?
- NP problem: An optimization problem where a solution can be verified in polynomial time, e.g., traveling salesman (No known polynomial-time algorithm to compute the route, but easy to check whether a route is a solution)
- NP-hard problem: problem which is at least as hard as the hardest NP problems. It is suspected that there are no polynomial-time algorithms that can solve NP-hard problems, but this has never been proven. It is not known if P!=NP or if P=NP: \$1M+posterity if you prove it (please wait after the final)

Computational Complexity in One Slide

- P problem: A decision problem where the "yes" or "no" answer can be decided in polynomial time (= polynomial number of operations relative to the input size), e.g., is 20 the maximum value in an array of n integers?
- NP problem: An optimization problem where a solution can be verified in polynomial time, e.g., traveling salesman (No known polynomial-time algorithm to compute the route, but easy to check whether a route is a solution)
- NP-hard problem: problem which is at least as hard as the hardest NP problems. It is suspected that there are no polynomial-time algorithms that can solve NP-hard problems, but this has never been proven. It is not known if P!=NP or if P=NP: \$1M+posterity if you prove it (please wait after the final)
- A typical NP-hard problem:

Computational Complexity in One Slide

- P problem: A decision problem where the "yes" or "no" answer can be decided in polynomial time (= polynomial number of operations relative to the input size), e.g., is 20 the maximum value in an array of n integers?
- NP problem: An optimization problem where a solution can be verified in polynomial time, e.g., traveling salesman (No known polynomial-time algorithm to compute the route, but easy to check whether a route is a solution)
- NP-hard problem: problem which is at least as hard as the hardest NP problems. It is suspected that there are no polynomial-time algorithms that can solve NP-hard problems, but this has never been proven. It is not known if P!=NP or if P=NP: \$1M+posterity if you prove it (please wait after the final)
- A typical NP-hard problem: Bin Packing

Computational Complexity in One Slide

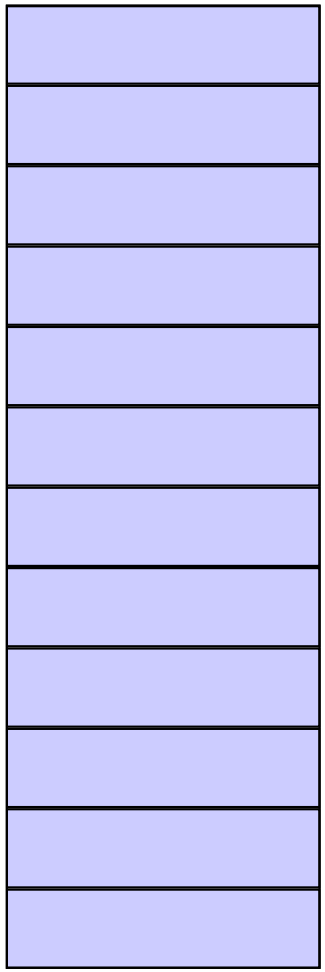
- P problem: A decision problem where the "yes" or "no" answer can be decided in polynomial time (= polynomial number of operations relative to the input size), e.g., is 20 the maximum value in an array of n integers?
- NP problem: An optimization problem where a solution can be verified in polynomial time, e.g., traveling salesman (No known polynomial-time algorithm to compute the route, but easy to check whether a route is a solution)
- NP-hard problem: problem which is at least as hard as the hardest NP problems. It is suspected that there are no polynomial-time algorithms that can solve NP-hard problems, but this has never been proven. It is not known if P!=NP or if P=NP: \$1M+posterity if you prove it (please wait after the final)
- A typical NP-hard problem: Bin Packing
- **Conclusion for OS design**: Variable-size chunks are a bad idea

Pages

- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging

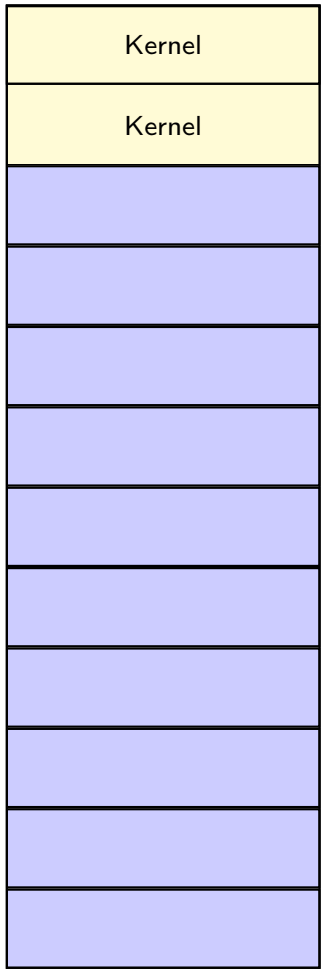
Pages

- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames (frame size = page size)



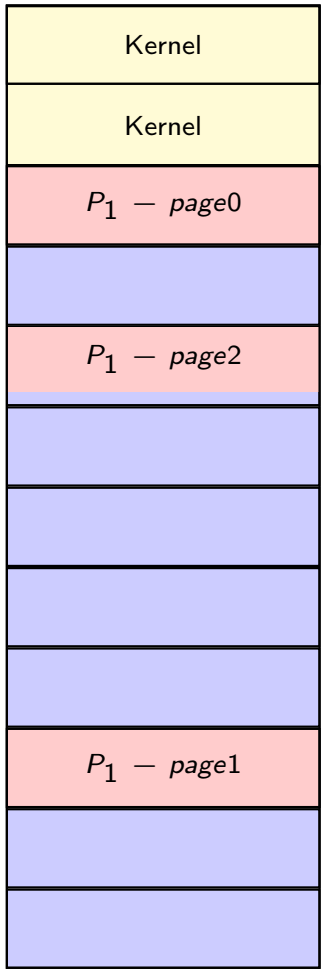
Pages

- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)
- A page can be placed in any free frame



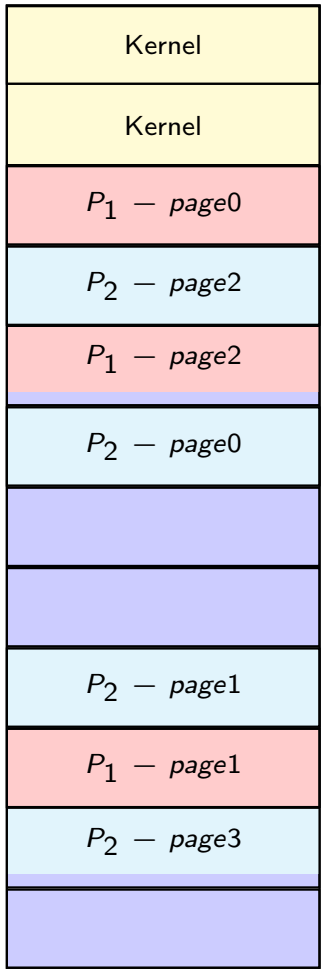
Pages

- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)
- A page can be placed in any free frame



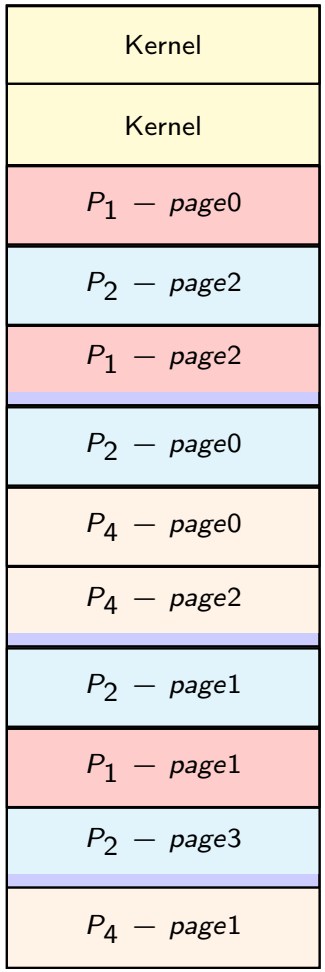
Pages

- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)
- A page can be placed in any free frame



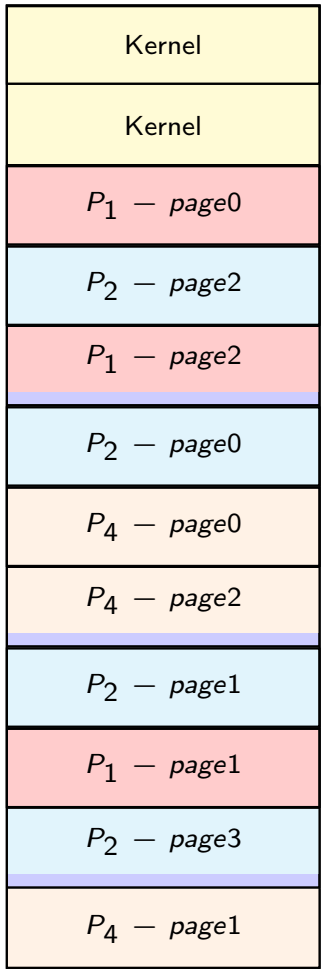
Pages

- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)
- A page can be placed in any free frame



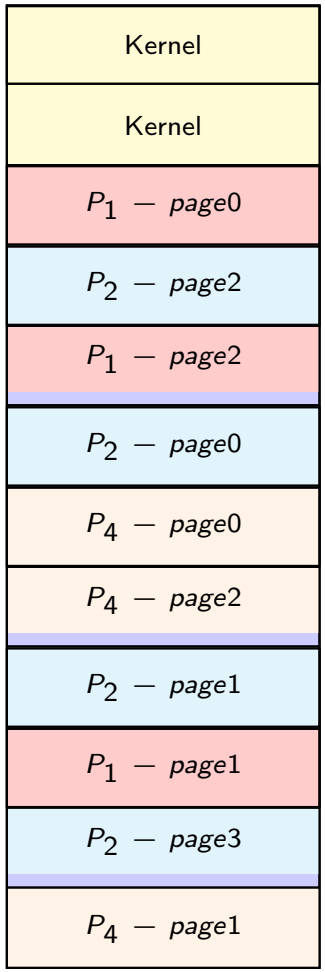
Pages

- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)
- A page can be placed in any free frame
- And just like that, we have non-contiguous memory allocation



Pages

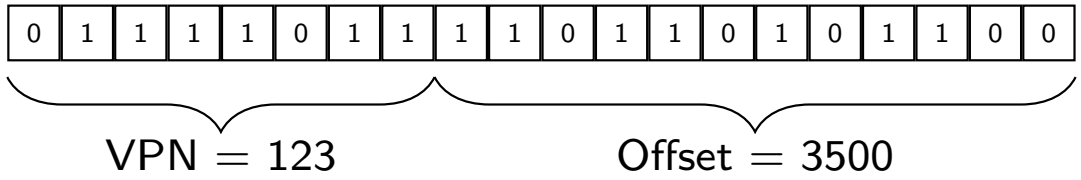
- Let's use same-size chunks
  - Easier to pack same-size boxes size into bins (not NP-hard!)
- We call these chunks the process' pages
- The process address space is split into fixed-size pages, a policy called paging
- The physical memory is split in fixed-size frames (frame size = page size)
- A page is "virtual" (or "logical"): Virtual Page Number (VPN)
- A frame is physical: Physical Frame Number (PFN)
- A page can be placed in any free frame
- And just like that, we have non-contiguous memory allocation



We still have internal fragmentation, but never external fragmentation!

(Virtual) Page Number

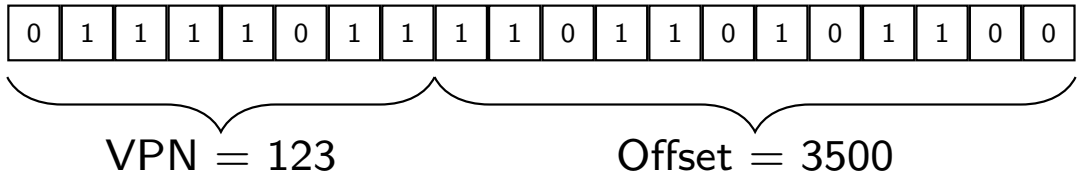
- When the CPU issues a virtual address, this address is split into two parts:



  - The virtual/logical page number: p
  - The offset within the page: d

(Virtual) Page Number

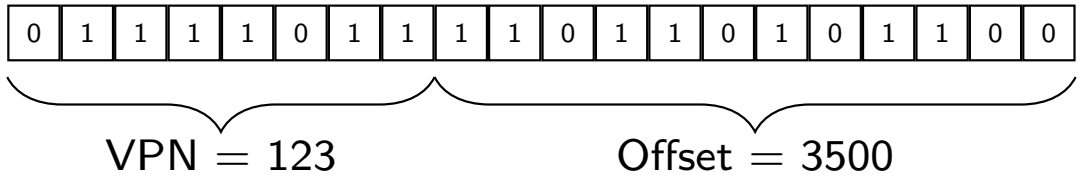
- When the CPU issues a virtual address, this address is split into two parts:



  - The virtual/logical page number: p
  - The offset within the page: d
  - i.e., instead of "read the value at address x" we think of it as "read the value at offset d in page p"

(Virtual) Page Number

- When the CPU issues a virtual address, this address is split into two parts:

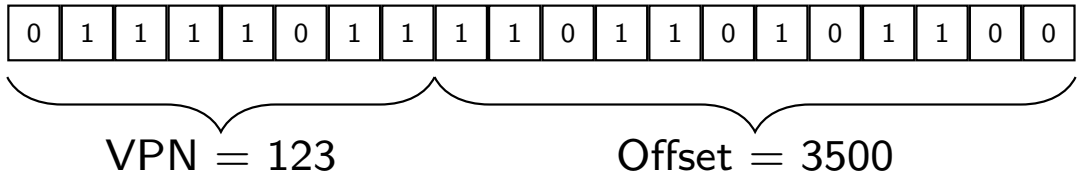


  - The virtual/logical page number: p
  - The offset within the page: d
  - i.e., instead of "read the value at address x" we think of it as "read the value at offset d in page p"

- 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

(Virtual) Page Number

- When the CPU issues a virtual address, this address is split into two parts:

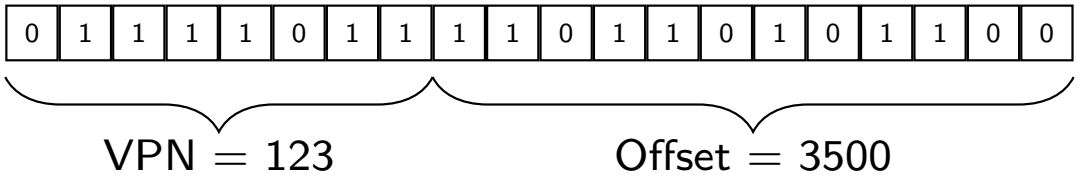


  - The virtual/logical page number: p
  - The offset within the page: d
  - i.e., instead of "read the value at address x" we think of it as "read the value at offset d in page p"

- 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

- When the CPU issues a virtual address, this address is split into two parts:



  - The virtual/logical page number: p
  - The offset within the page: d
  - i.e., instead of "read the value at address x" we think of it as "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

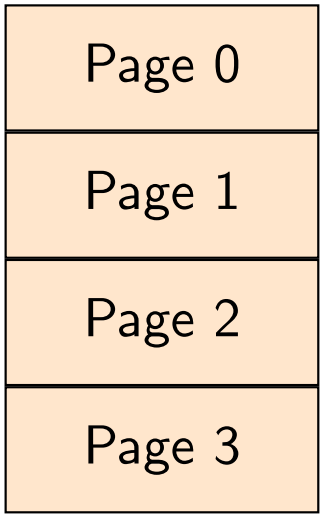
- The Virtual Page Number (VPN) has to be translated to the corresponding Physical Frame Number (PFN)
- This is more sophisticated address translation scheme than what we saw in the previous module for contiguous memory allocation
- Remember from the previous slide: instead of "read the value at address x", a program program does "read the value at offset d in page p"

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 one of its pages and the physical frame that page is in

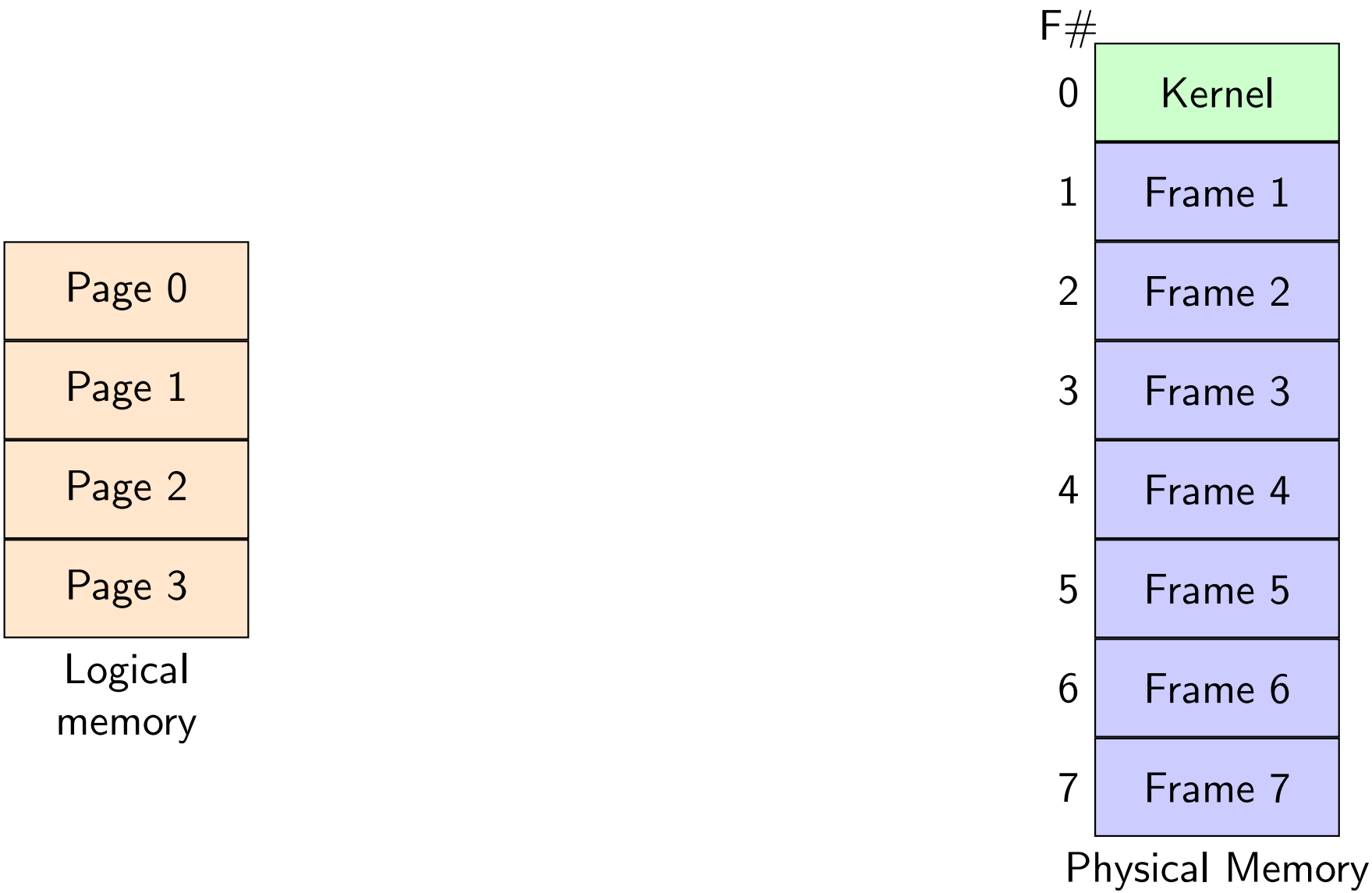
- Do this end, each process has a page table...

Page Table Example

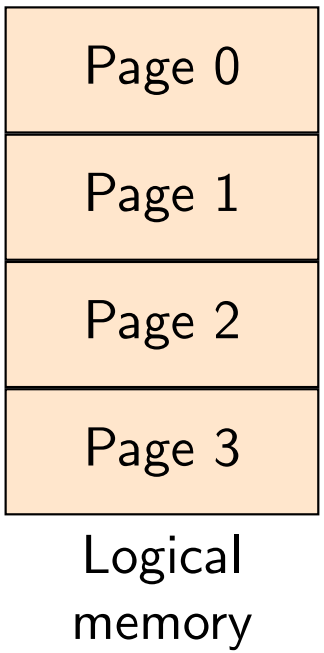


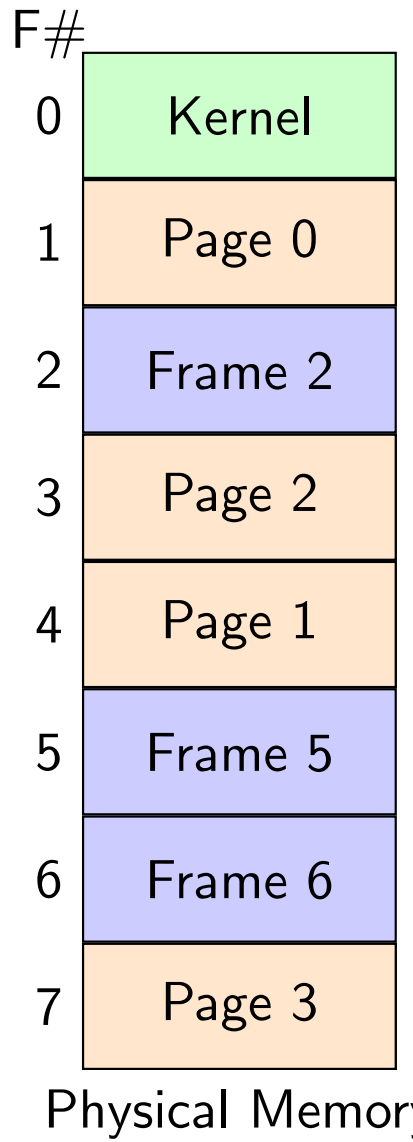
Logical memory

Page Table Example

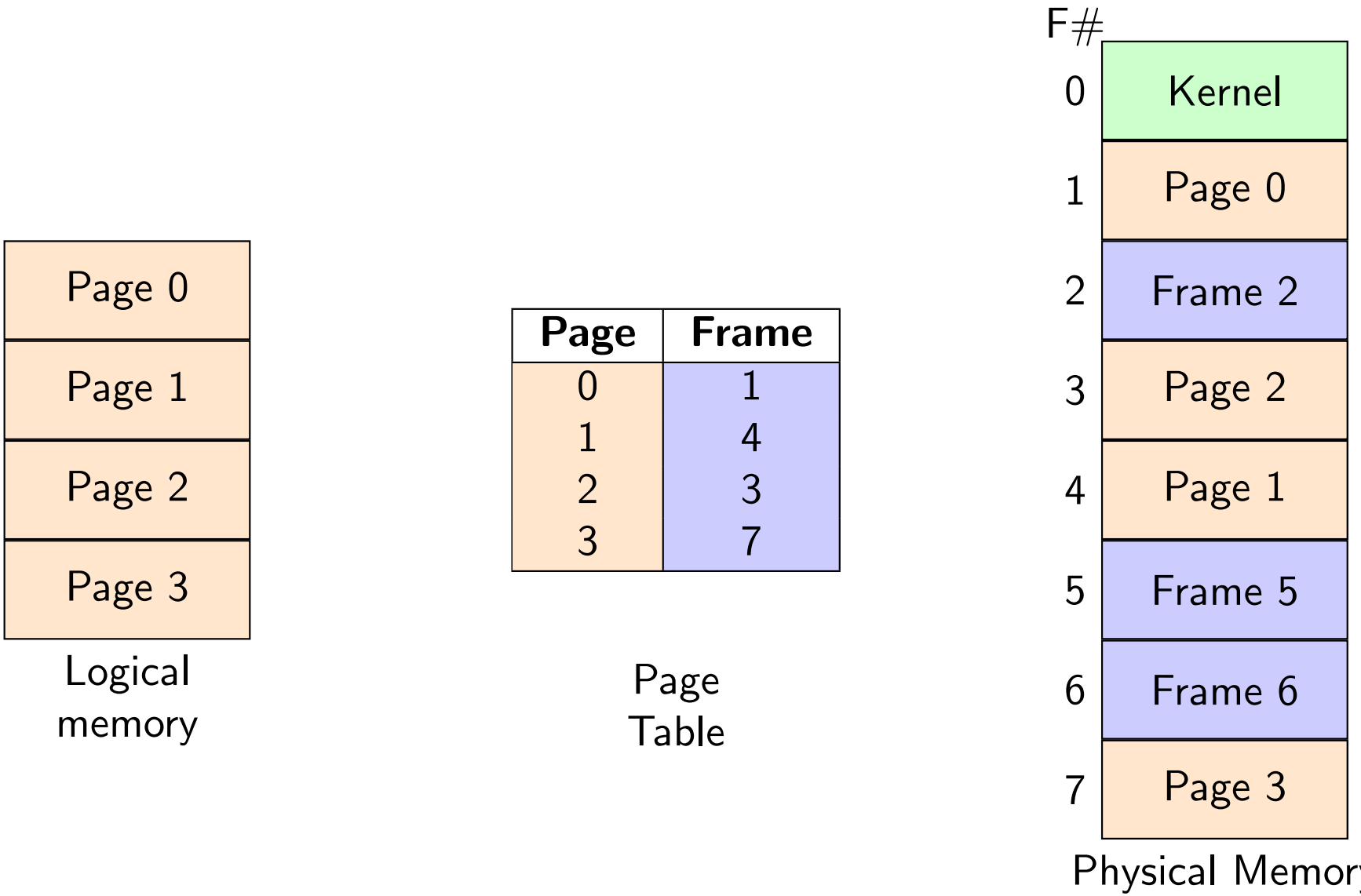


Page Table Example





Page Table Example



Page Size

- The page size is defined by the architecture

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

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: 4096

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: 4096

- 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...

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...

$\Longrightarrow$ 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)

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...

$\Longrightarrow$ 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)

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

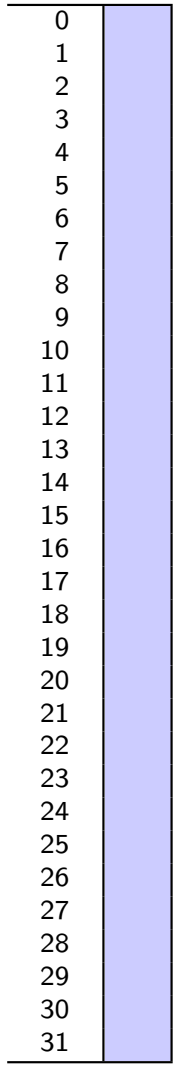
- Let's see this on an example...

Example

- Physical memory size $= 2^5 = 32$ bytes

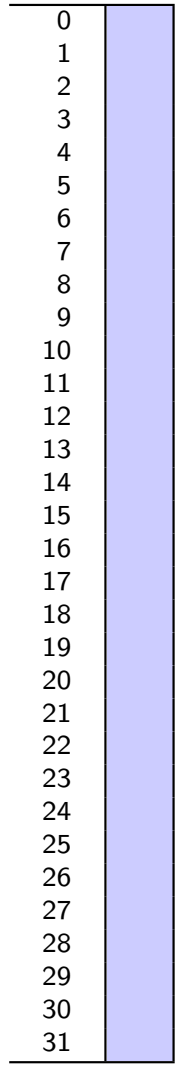
Example

- Physical memory size $= 2^5 = 32$ bytes



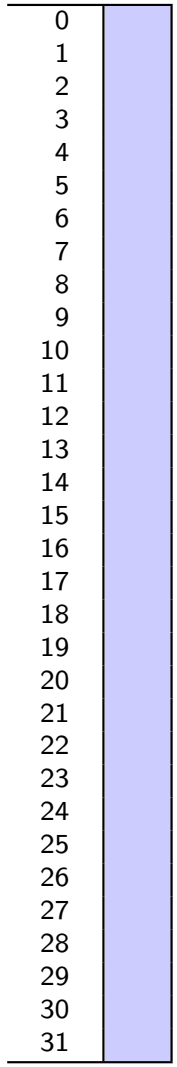
Example

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



Example

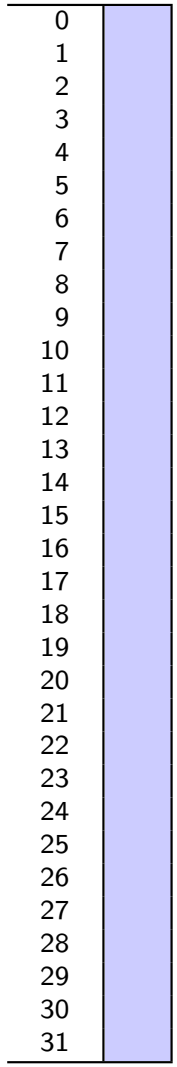
- Physical memory size $= 2^5 = 32$ bytes
- How many bits in a physical address?
  - How many bits are necessary to address 2^5 thingies?



Example

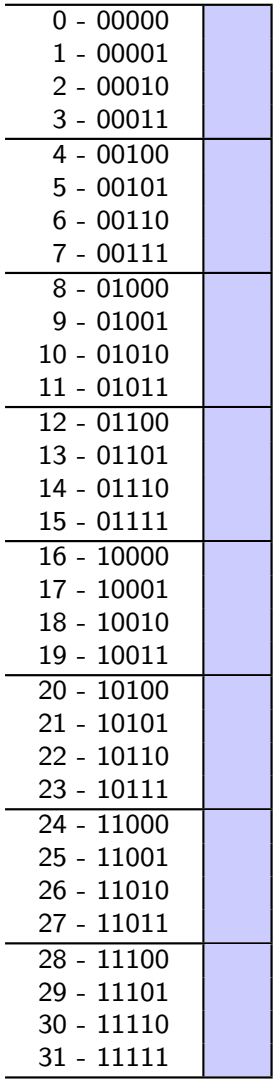
- Physical memory size $= 2^5 = 32$ bytes
- How many bits in a physical address?
  - How many bits are necessary to address 2^5 thingies?

5 bits



Example

- 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 all pick page size = 4 bytes



Example

- $2^5 = 32$ bytes of RAM
- 5-bit physical addresses
- 4-byte frames
- How many 4-byte frames are there?

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

Example

- $2^5 = 32$ bytes of RAM
- 5-bit physical addresses
- 4-byte frames
- How many 4-byte frames are there?

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

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

Example

- $2^5 = 32$ bytes of physical RAM
- 5-bit physical addresses
- 4-byte physical frames
- 8 frames in RAM
- 4-byte pages
- Let's say we have a process with a 16-byte address space
- How many pages it this address space?

Example

- $2^5 = 32$ bytes of physical RAM
- 5-bit physical addresses
- 4-byte physical frames
- 8 frames in RAM
- 4-byte pages
- Let's say we have a process with a 16-byte address space
- How many pages it this address space?

$$\frac{16(\texttt{bytes})}{4(\texttt{bytes/page})} = 4 \text{ pages}$$

Example

- $2^5 = 32$ bytes of physical RAM
- 5-bit physical addresses
- 4-byte physical frames
- 8 frames in RAM
- 4-byte pages
- Let's say we have a process with a 16-byte address space
- How many pages it this address space?

$$\frac{16(\texttt{bytes})}{4(\texttt{bytes/page})} = 4 \text{ pages}$$

  - Say the address space contains values a, b, . . ., p

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

Example

- $2^5 = 32$ bytes of physical RAM
- 5-bit physical addresses
- 4-byte physical frames
- 8 frames in RAM
- 4-byte pages
- Let's say we have a process with a 16-byte address space
- How many pages it this address space?

$$\frac{16(\texttt{bytes})}{4(\texttt{bytes/page})} = 4 \text{ pages}$$

  - Say the address space contains values a, b, . . ., p
- Say the OS has placed Page 0 into Frame 5, Page 1 into Frame 6, Page 2 into Frame 1, and Page 3 into Frame 2.

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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

Example

- $2^5 = 32$ bytes of physical RAM
- 5-bit physical addresses
- 4-byte physical frames
- 8 frames in RAM
- 4-byte pages
- Let's say we have a process with a 16-byte address space
- How many pages it this address space?

$$\frac{16(\texttt{bytes})}{4(\texttt{bytes/page})} = 4 \text{ pages}$$

  - Say the address space contains values a, b, . . ., p
- Say the OS has placed Page 0 into Frame 5, Page 1 into Frame 6, Page 2 into Frame 1, and Page 3 into Frame 2.
  - Therefore, the OS will have created a page table with 4 entries 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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

Example

- $2^5 = 32$ bytes of physical RAM
- 5-bit physical addresses
- 4-byte physical frames
- 8 frames in RAM
- 4-byte pages
- Let's say we have a process with a 16-byte address space
- How many pages it this address space?

$$\frac{16(\texttt{bytes})}{4(\texttt{bytes/page})} = 4 \text{ pages}$$

  - Say the address space contains values a, b, . . ., p
- Say the OS has placed Page 0 into Frame 5, Page 1 into Frame 6, Page 2 into Frame 1, and Page 3 into Frame 2.
  - Therefore, the OS will have created a page table with 4 entries for that process
- 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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

Example

- $2^5 = 32$ bytes of physical RAM
- 5-bit physical addresses
- 4-byte physical frames
- 8 frames in RAM
- 4-byte pages
- Let's say we have a process with a 16-byte address space
- How many pages it this address space?

$$\frac{16(\texttt{bytes})}{4(\texttt{bytes/page})} = 4 \text{ pages}$$

  - Say the address space contains values a, b, . . ., p
- Say the OS has placed Page 0 into Frame 5, Page 1 into Frame 6, Page 2 into Frame 1, and Page 3 into Frame 2.
  - Therefore, the OS will have created a page table with 4 entries for that process
- How many bits in a virtual address for that process?
  4 bits (because we have 2^4 bytes)
  2-bit page index
  2-bit offset in the page

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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

Example

- What is the logical address of g?
  logical @ = (page nbr)*(page size) + offset

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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

Example

- What is the logical address of g?
  logical @ = (page nbr)*(page size) + offset
  Page=1; Offset=2
  (often written 1:2)
  = 1×4+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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

Example

- What is the logical address of g?
  logical @ = (page nbr)*(page size) + offset
  Page=1; Offset=2
  (often written 1:2)
  = 1×4+2 = 6

- What is the physical address of g?
  physical @ = (frame nbr)*(frame size) + offset

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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

Example

- What is the logical address of g?
  logical @ = (page nbr)*(page size) + offset
  Page=1; Offset=2
  (often written 1:2)
  = 1×4+2 = 6

- What is the physical address of g?
  physical @ = (frame nbr)*(frame size) + offset
  Page 1 is in Frame 6,
  same offset: 2,
  therefore: 6×4+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

@		F#
0		0
1		
2		
3		
4	i	1
5	j	
6	k	
7	l	
8	m	2
9	n	
10	o	
11	p	
12		3
13		
14		
15		
16		4
17		
18		
19		
20	a	5
21	b	
22	c	
23	d	
24	e	6
25	f	
26	g	
27	h	
28		7
29		
30		
31		

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: 4GiB $= 2^{32}$ bytes
  $\Rightarrow$ 32-bit physical addresses
- How many bits are used for logical addresses?
  Logical Address space: 1GiB $= 2^{30}$ bytes
  $\Rightarrow$ 30-bit logical addresses
- How many bits are used for logical page numbers?
  Page size $= 2^{13}$ bytes
  Number of pages in logical address space: $2^{30}/2^{13} = 2^{17}$
  To address 2^{17} things, we need 17 bits
  $\Rightarrow$ 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 if 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 of 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?
- How many entries are there in the process' page table?

Another In-class Exercise (4)

A computer has 32-bit physical addresses. The logical page number of 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?
  How many 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 are there in the process' page table?
  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
  (one entry per 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?
- My process has 2^{22} pages, 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 bit 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

- My process has 2^{22} pages, 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 bites 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 $4 \times 0 = 0$ (the first byte in physical RAM)
- The first byte of physical frame 1 is at physical address $4 \times 1 = 4$ (the fifth byte in physical RAM)
- ...
- The first byte of physical frame 4 is at physical address $4 \times 4 = 16$
- 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)

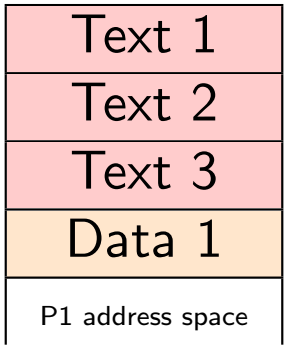
- Consider a system with 4-byte pages. A process has the following entries in its page table:

logical	physical
0	4
1	5
2	30

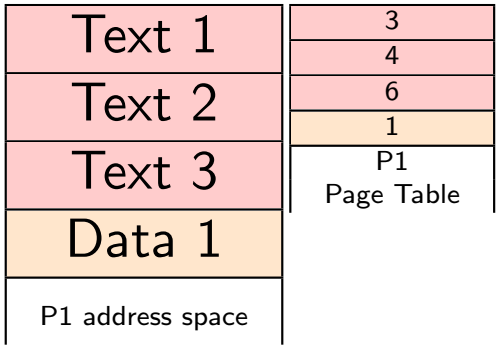
- What is the physical address of the byte with logical address 9?

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

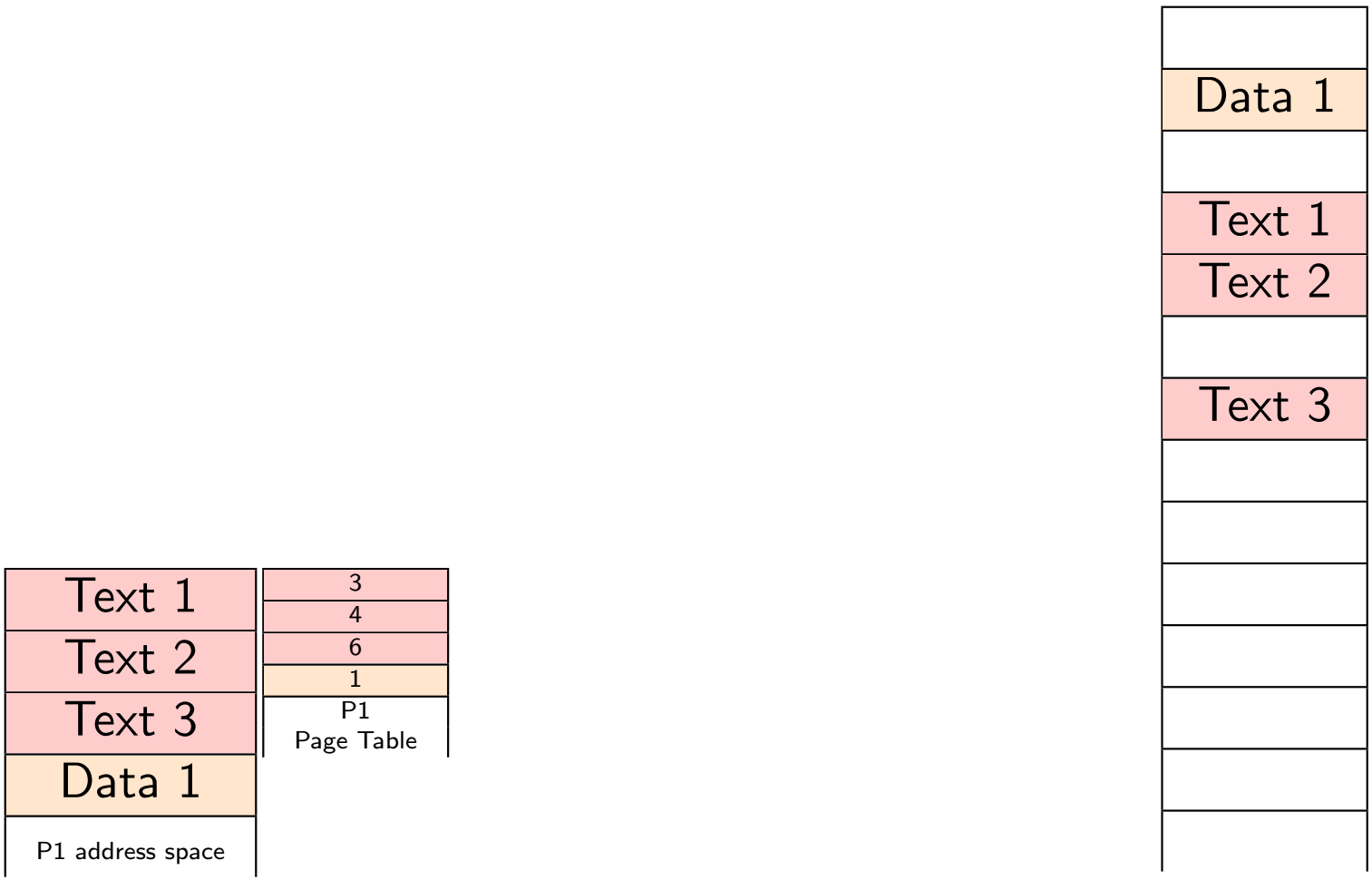
Sharing Memory Pages Across Processes? EASY!



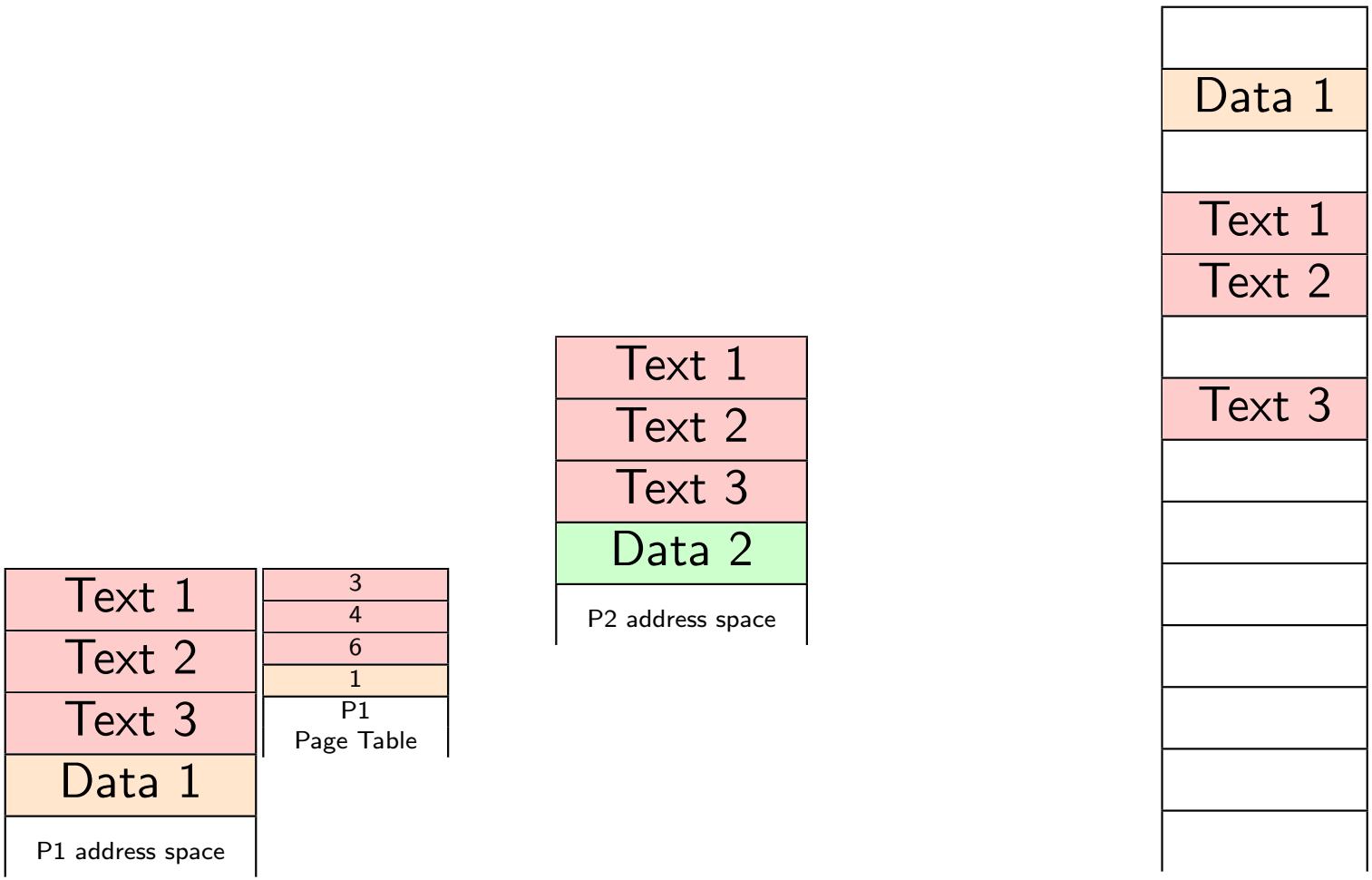
Sharing Memory Pages Across Processes? EASY!



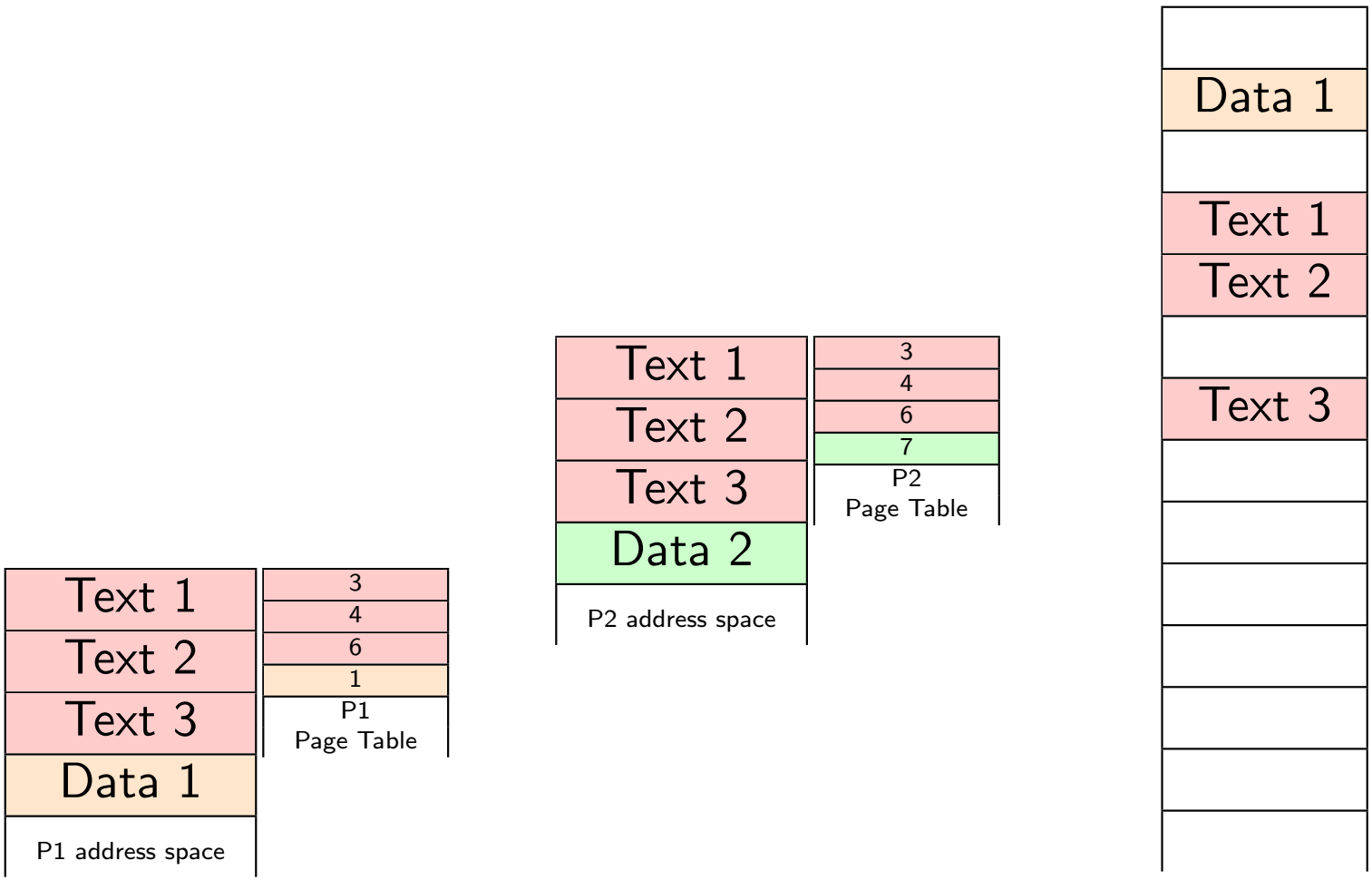
Sharing Memory Pages Across Processes? EASY!



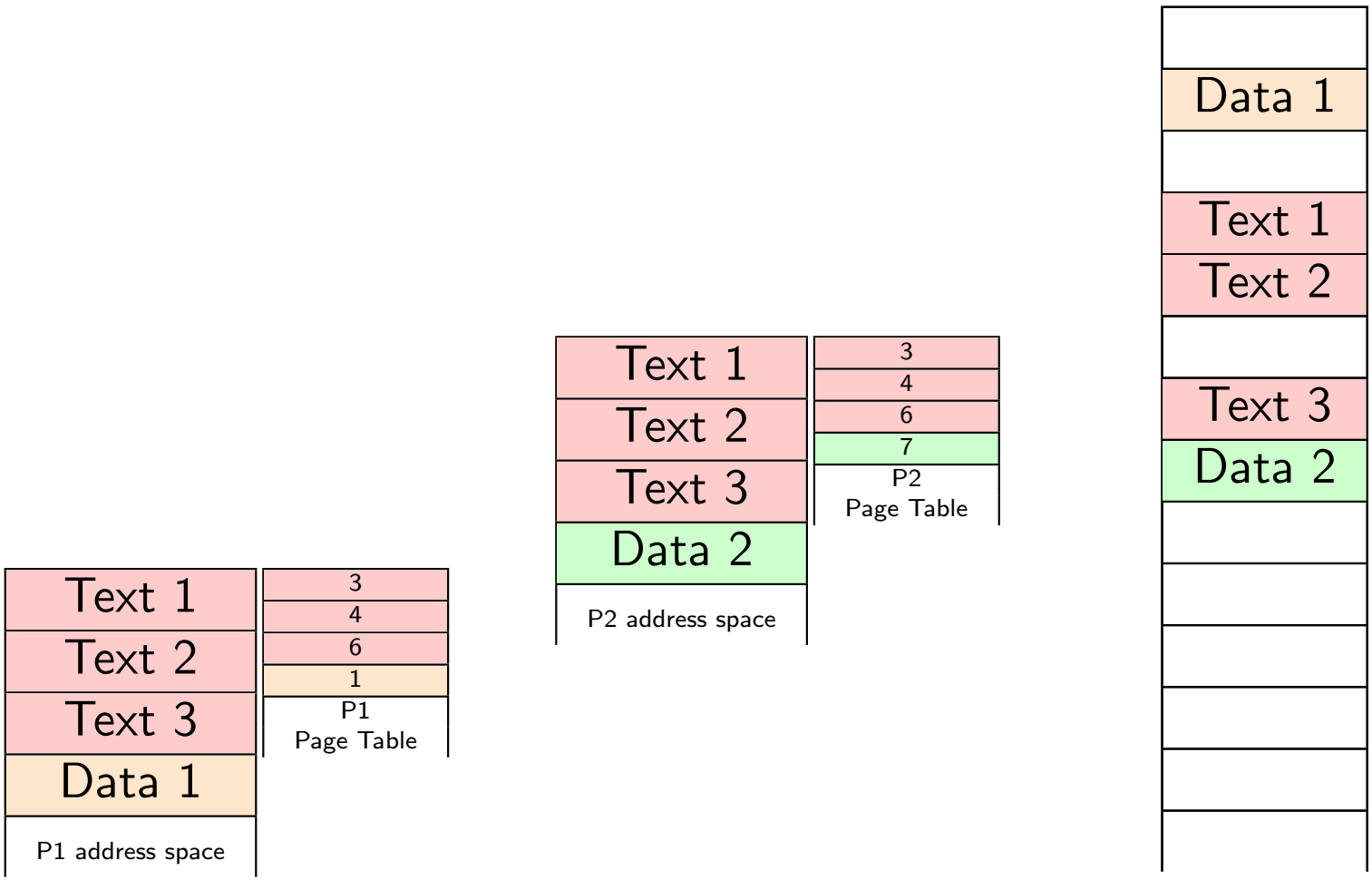
Sharing Memory Pages Across Processes? EASY!



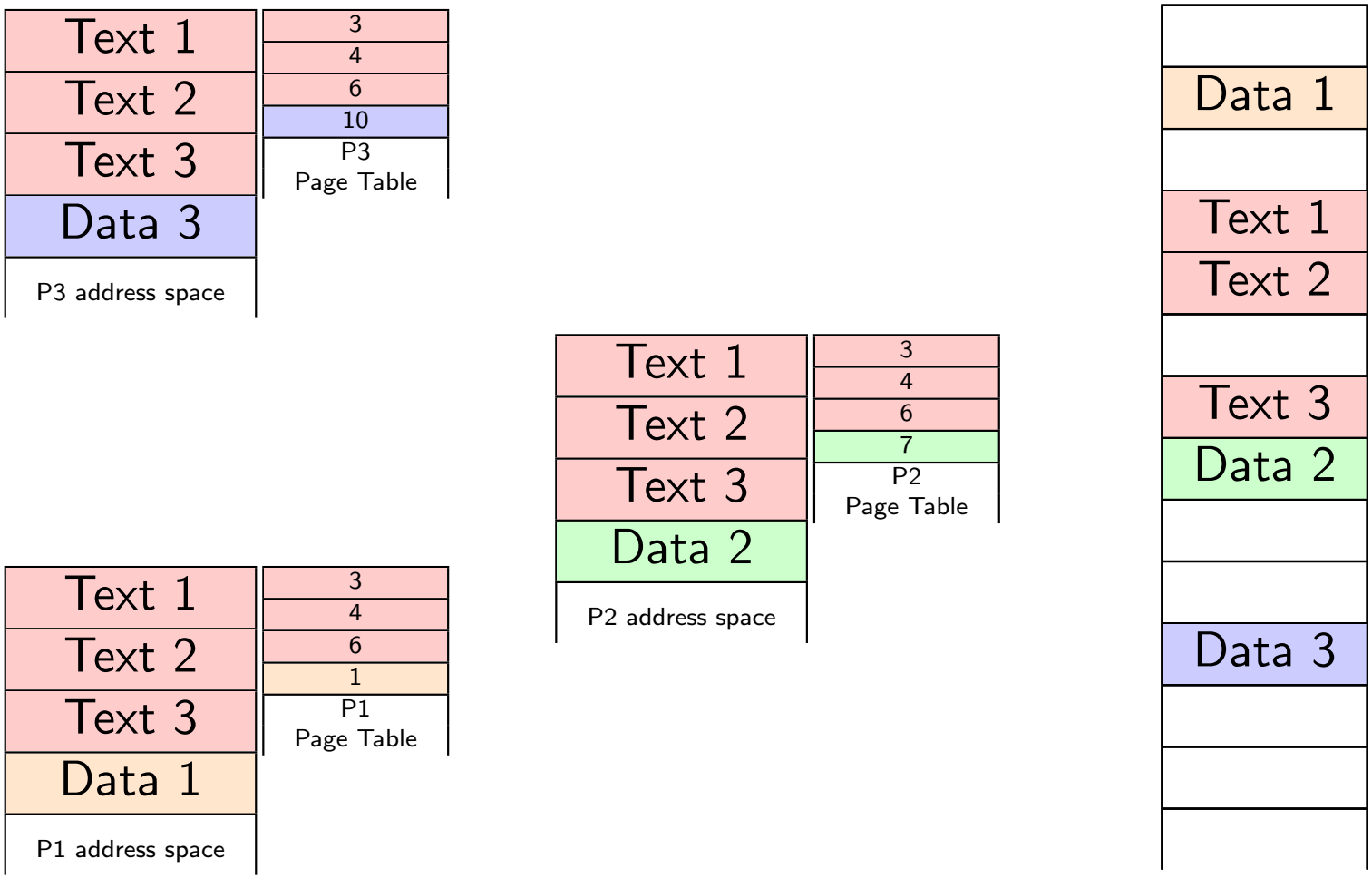
Sharing Memory Pages Across Processes? EASY!



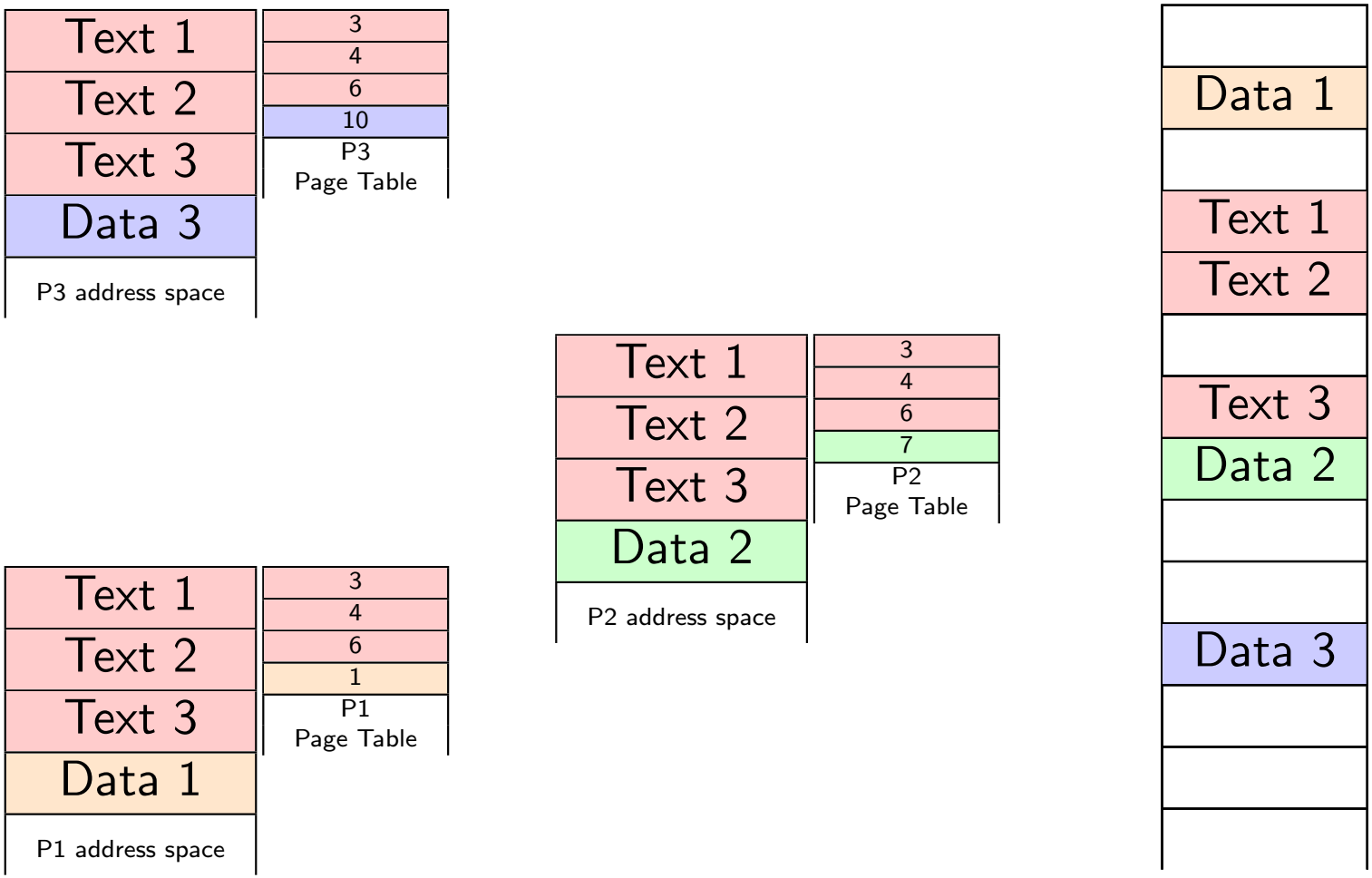
Sharing Memory Pages Across Processes? EASY!



Sharing Memory Pages Across Processes? EASY!



Sharing Memory Pages Across Processes? EASY!



Just insert page table entries that point to the same physical frames!

Valid bit

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

Page Table

P0	14
P1	13
P2	18
P3	20

- 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)
- So really, it looks like that:

Page Table

P0	14
P1	13
P2	18
P3	20
P4	not used (yet)
P5	not used (yet)
P6	not used (yet)
P7	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. is the page in the process address space)
- Set to invalid otherwise

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. is the page in the process address space)
- Set to invalid otherwise
- So page tables look like this:

Page Table

P0	14	✓
P1	13	✓
P2	18	✓
P3	20	✓
P4	xx	-
P5	xx	-
P6	xx	-
P7	xx	-

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. is the page in the process address space)
- Set to invalid otherwise
- So page tables look like this:

Page Table

P0	14	✓
P1	13	✓
P2	18	✓
P3	20	✓
P4	xx	-
P5	xx	-
P6	xx	-
P7	xx	-

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

What about Fragmentation?

- No external fragmentation!!
  - This is of course the HUGE advantage of paging

What about Fragmentation?

- No external fragmentation!!
  - This is of course the HUGE advantage of paging
- Only internal fragmentation

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

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

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 lookup overhead)
  - Loading one large page from disk takes less time than loading many small ones

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 lookup overhead)
  - Loading one large page from disk takes less time than loading many small ones
- Typical sizes: 4KiB, 8KiB (Linux: `pagesize`)
- Modern OSes: multiple page sizes support (Linux: Huge pages; Mac: Superpages; Windows: Large pages) through hardware

Frames Management

- The OS needs to keep track of the frames

Frames Management

- The OS needs to keep track of the frames
  - Which frames are used (and by which processes?)
  - Which frames are free?

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

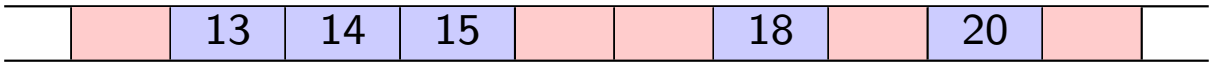
Frames Management

- 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 in the previous "Main Memory" module
- When a process needs a frame, then the OS takes a frame from the free frame list and allocate them to a process

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 in the previous "Main Memory" module
- When a process needs a frame, then the OS takes a frame from the free frame list and allocate them to a process

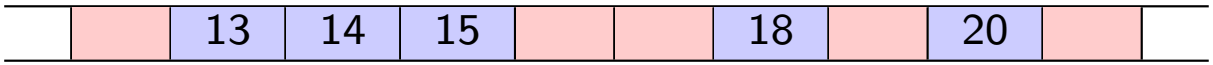
Free-frame list = {14, 13, 18, 20, 15}



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 in the previous "Main Memory" module
- When a process needs a frame, then the OS takes a frame from the free frame list and allocate them to a process

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

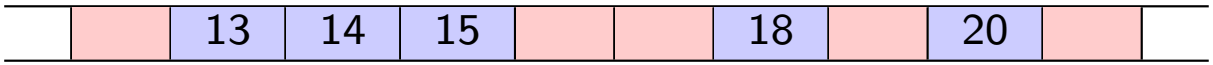


Process creation: Needs 4 pages: P0, P1, P2, P3

Frames Management

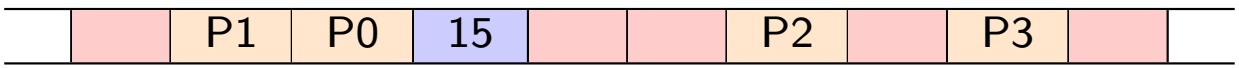
- The OS needs to keep track of the frames
  - Which frames are used (and by which processes?)
  - Which frames are free?
- The OS thus has a data structure: the free frame list
- Much simpler than a list of holes with different sizes
  - As done in the previous "Main Memory" module
- When a process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

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



Process creation: Needs 4 pages: P0, P1, P2, P3

Free-frame list $= \{15\}$



Page Table

P0	14
P1	13
P2	18
P3	20

Segmentation and Paging: e.g., IA 32/64

- The Intel architecture provides both segmentation and paging
- A logical/virtual address is transformed into a linear address via segmentation

  logical address = (segment selector, segment offset)

- A linear address is transformed into a physical address via paging

  linear address = (page number level-1, p-2, p-3, p-4, offset)

- See OSTEP: Advanced Page Tables for full details

Segmentation and Paging: e.g., IA 32/64

- The Intel architecture provides both segmentation and paging
- A logical/virtual address is transformed into a linear address via segmentation

  logical address = (segment selector, segment offset)

- A linear address is transformed into a physical address via paging

  linear address = (page number level-1, p-2, p-3, p-4, offset)

- See OSTEP: Advanced Page Tables for full details

CPU

Segmentation and Paging: e.g., IA 32/64

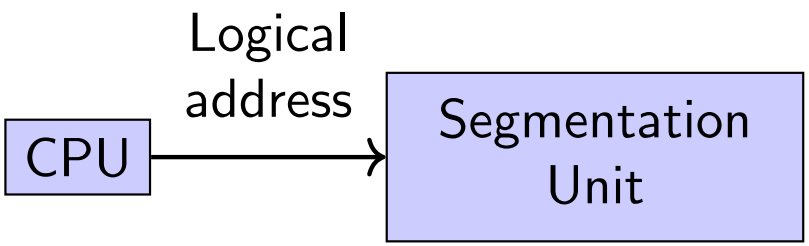
- The Intel architecture provides both segmentation and paging
- A logical/virtual address is transformed into a linear address via segmentation

  $$\text{logical address} = (\text{segment selector, segment offset})$$

- A linear address is transformed into a physical address via paging

  $$\text{linear address} = (\text{page number level-1, p-2, p-3, p-4, offset})$$

- See OSTEP: Advanced Page Tables for full details



Segmentation and Paging: e.g., IA 32/64

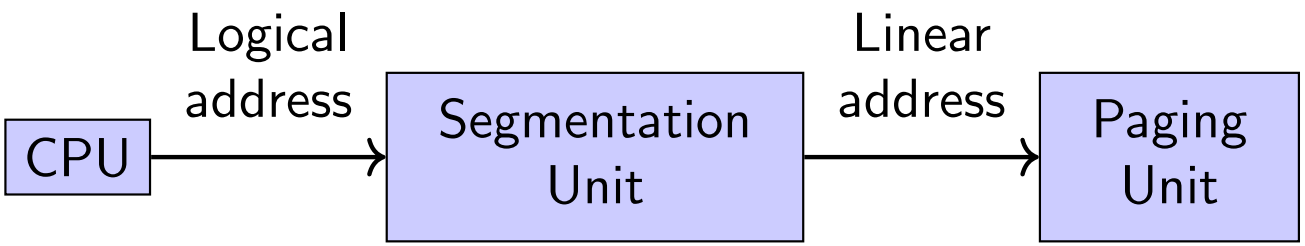
- The Intel architecture provides both segmentation and paging
- A logical/virtual address is transformed into a linear address via segmentation

  logical address = (segment selector, segment offset)

- A linear address is transformed into a physical address via paging

  linear address = (page number level-1, p-2, p-3, p-4, offset)

- See OSTEP: Advanced Page Tables for full details



Segmentation and Paging: e.g., IA 32/64

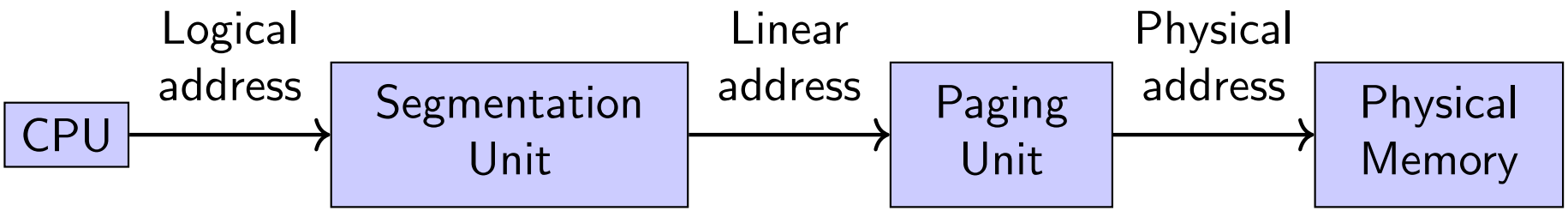
- The Intel architecture provides both segmentation and paging
- A logical/virtual address is transformed into a linear address via segmentation

  logical address = (segment selector, segment offset)

- A linear address is transformed into a physical address via paging

  linear address = (page number level-1, p-2, p-3, p-4, offset)

- See OSTEP: Advanced Page Tables for full details



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
- We can now do Question #2 of Homework #8...
- In the next set of lecture notes, we look at some challenges with paging and how we deal with them