COMP 4735: Operating Systems Concepts

Lecture 10: Memory Management

Rob Neilson

rneilson@bcit.ca

Administrative

- No quiz this week.
- No quiz next week.
- Next quiz (on Memory Management) is March 30th.

- Wednesday this week: no lecture. Go to career day, get a job, make money, do stuff like that.

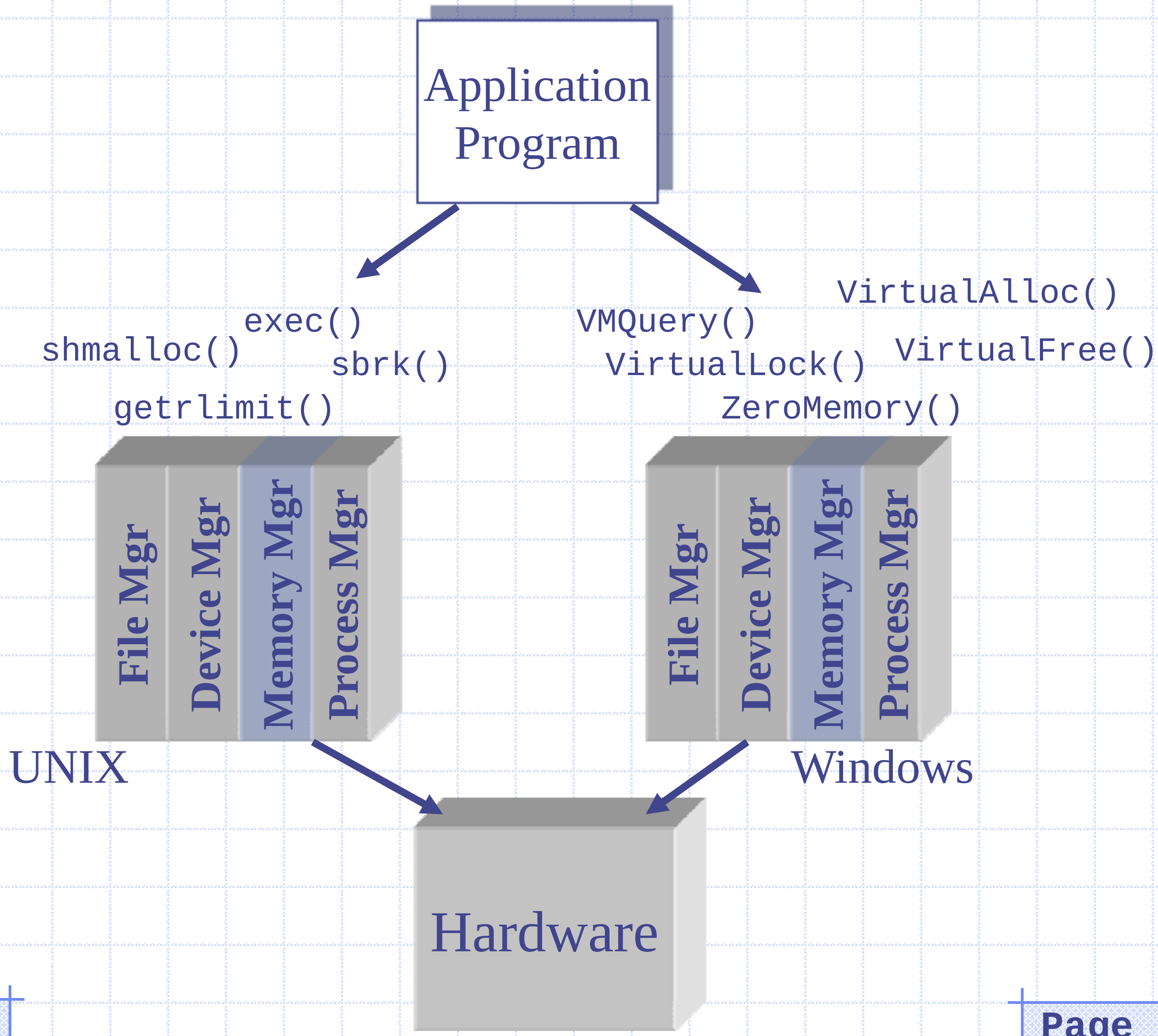
Acknowledgements ...

- *some of the slides in this lesson are courtesy of Jonathan Walpole, who teaches at Portland State University*

Today's Topics

- What is a Memory Manager?
- The Address Spaces Abstraction
- Static Address Binding
- Memory Allocation & Partitioning Strategies
- Dynamic Address Space Binding
- Swapping Systems

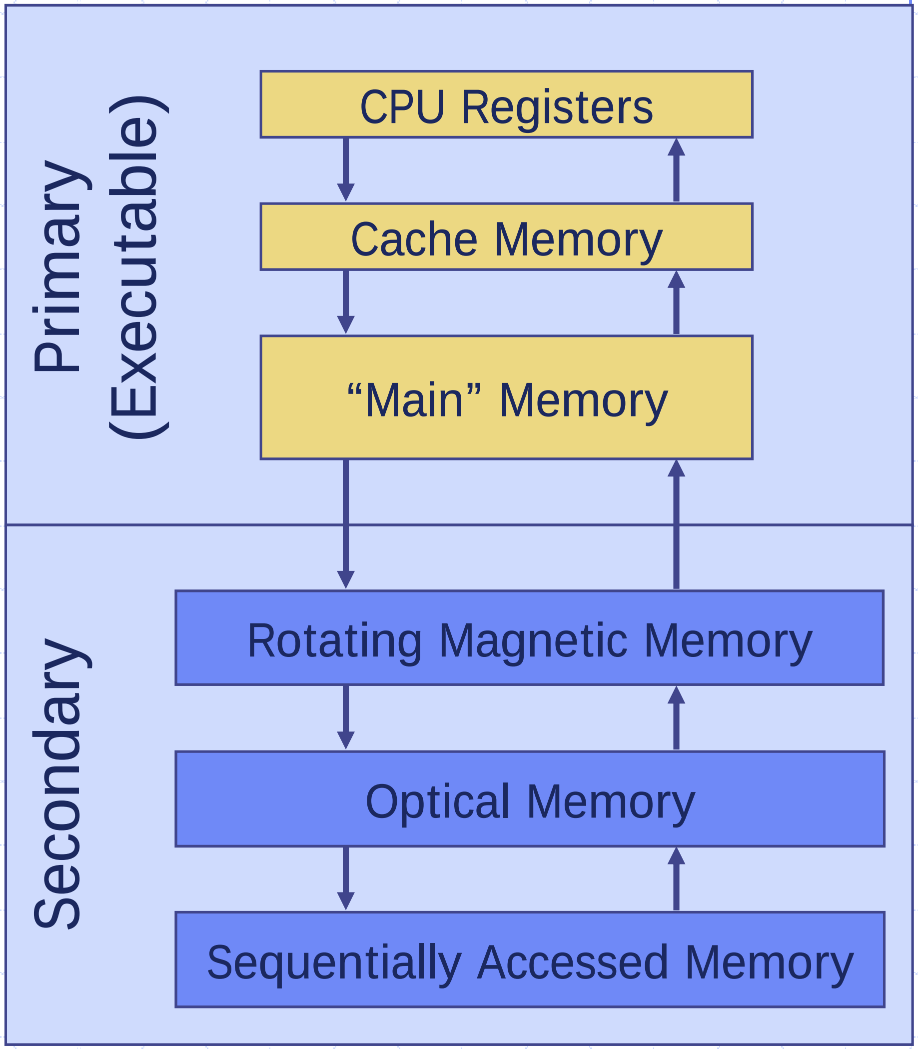
What is a Memory Manager?



Primary vs Secondary Memory

Memory Manager is responsible for

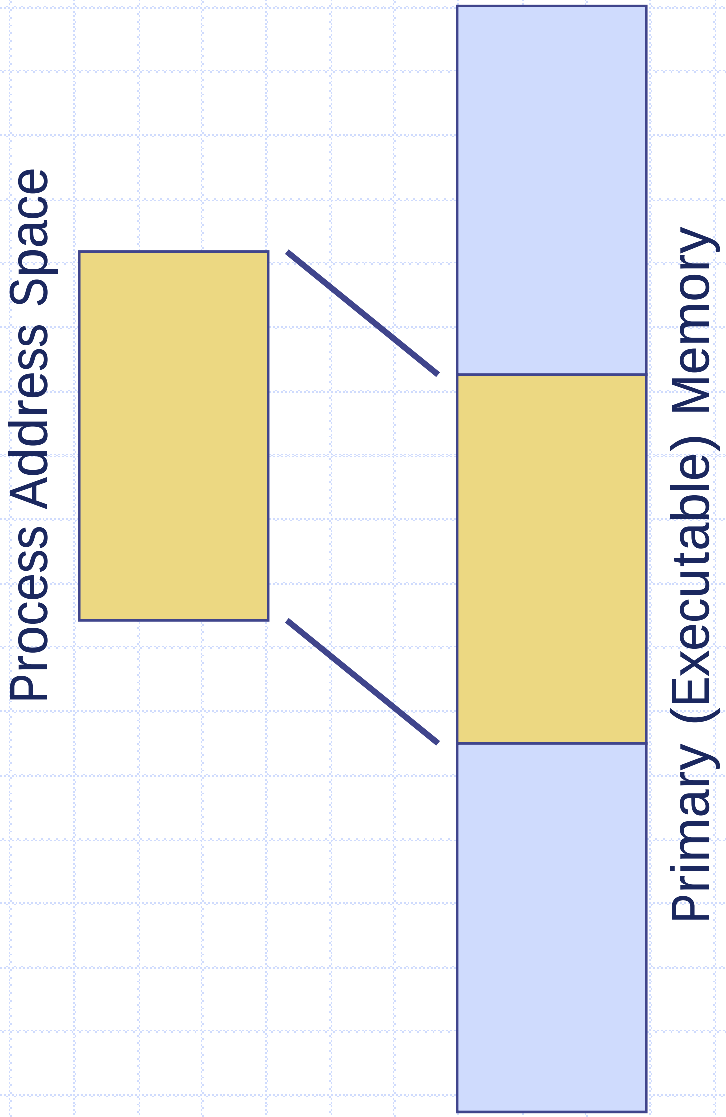
- automatically moving programs and data back and forth between *primary* and *secondary* memory



Address Space Abstraction

Memory Manager is responsible for

- automatically moving programs and data back and forth between *primary* and *secondary* memory
- managing the mapping of addresses between a processes address space and physical memory (binding)

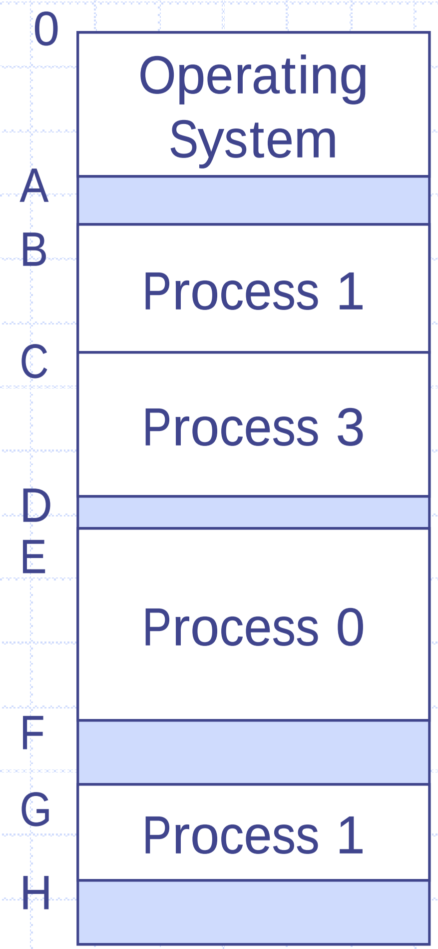


Memory Allocation

Memory Manager is responsible for

- automatically moving programs and data back and forth between *primary* and *secondary* memory
- managing the mapping of addresses between a processes address space and physical memory
- allocating/releasing blocks of memory as requested by processes

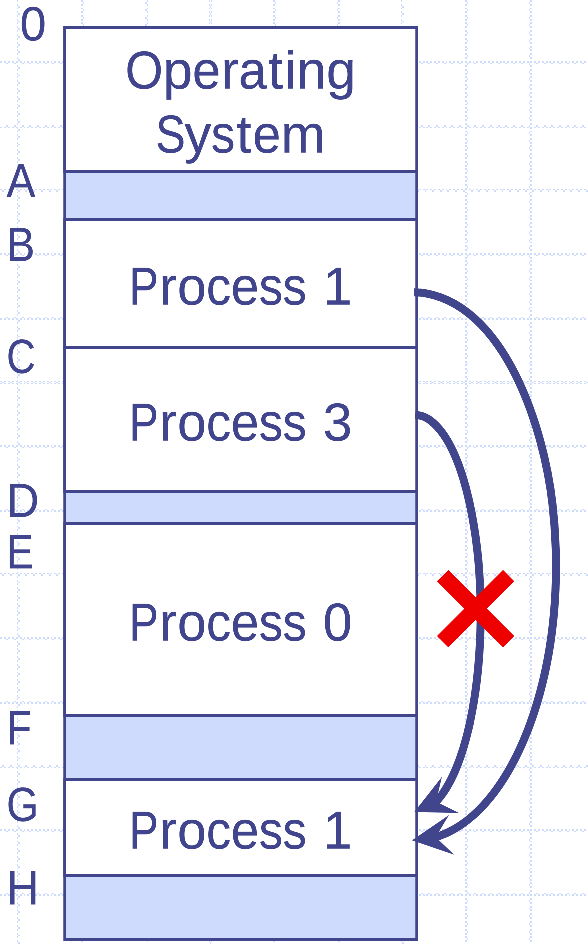
Physical Memory



Isolation

Memory Manager is responsible for

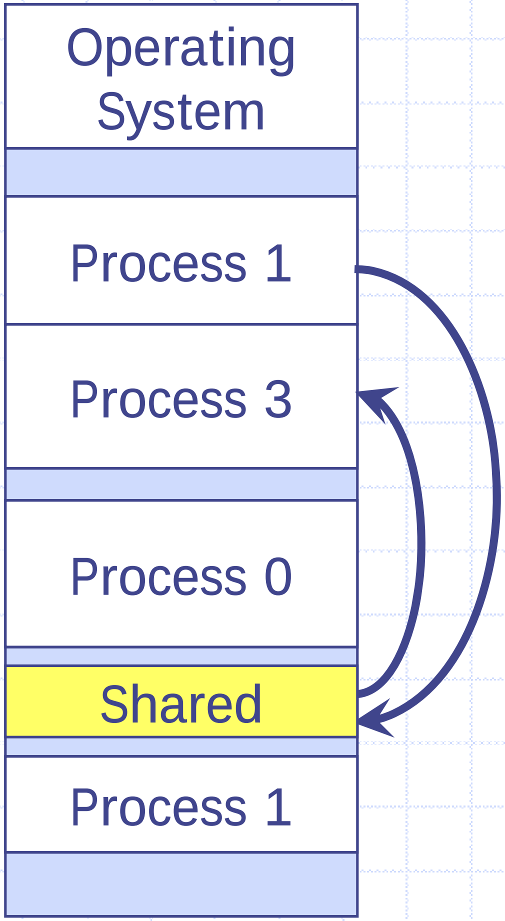
- automatically moving programs and data back and forth between *primary* and *secondary* memory
- managing the mapping of addresses between a processes address space and physical memory
- allocating/releasing blocks of memory as requested by processes
- ensuring that processes have exclusive use of the blocks of memory that are allocated to them



Sharing

Memory Manager is responsible for

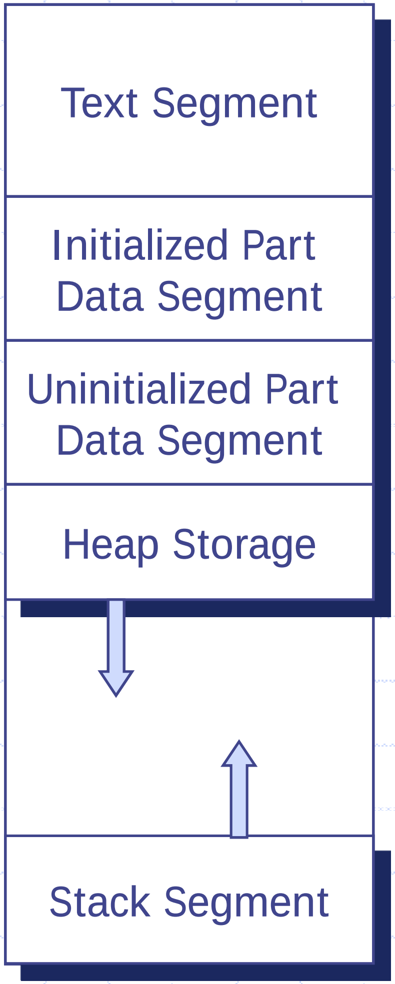
- automatically moving programs and data back and forth between *primary* and *secondary* memory
- managing the mapping of addresses between a processes address space and physical memory
- allocating/releasing blocks of memory as requested by processes
- ensuring that processes have exclusive use of the blocks of memory that are allocated to them
- facilitating shared access to blocks of memory by multiple processes (based on capabilities supported by the OS)



Address Space Abstraction

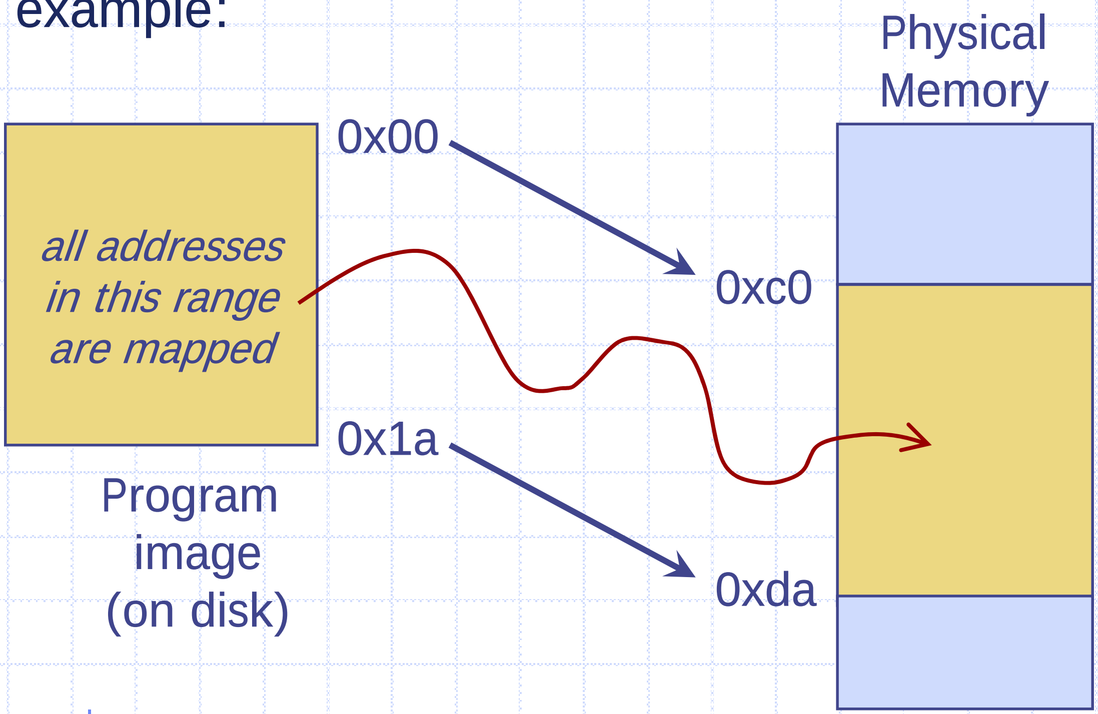
Memory Management

- Memory is presented as a linear array of bytes
  - Holds OS and programs (processes)
- Recall, processes are defined by an *address space,* consisting of text, data, and stack segments
- Process execution
  - CPU fetches instructions from the text region according to the value of the program counter (PC)
  - Each instruction may request additional operands from the data or stack region



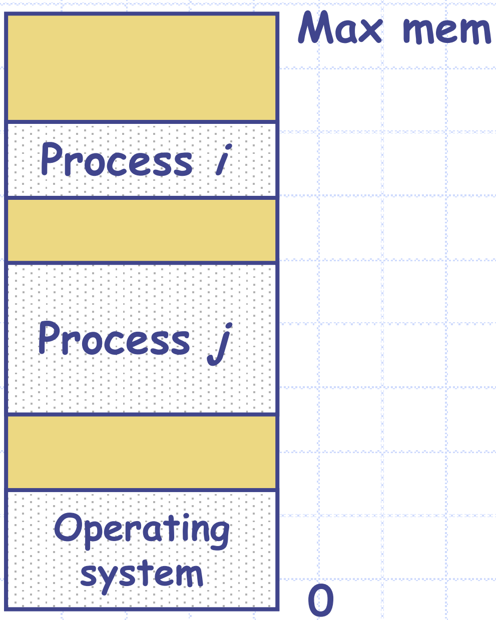
Addressing Memory

- We cannot know ahead of time where in memory a program will be loaded!
  - we don’t know what memory is available or allocated
- Memory Manager uses the notion of *address binding* to map programs (processes) into physical memory
- For example:



Multiple Processes

- in the simplest case, there are many processes mapped into different places in physical memory, and each process is mapped into one contiguous block of memory



Consider how a program gets organized / laid out within the process ...

Compiler

- Compiler produces code containing embedded addresses
  - these addresses can't be absolute (physical addresses)
  - compiler (and assembler) produces a *relocatable object module*

```
Prog P
   :
   :
  foo()
   :
   :
End P
```

```
P:
   :
  push ...
  jmp _foo
   :

foo: ...
```

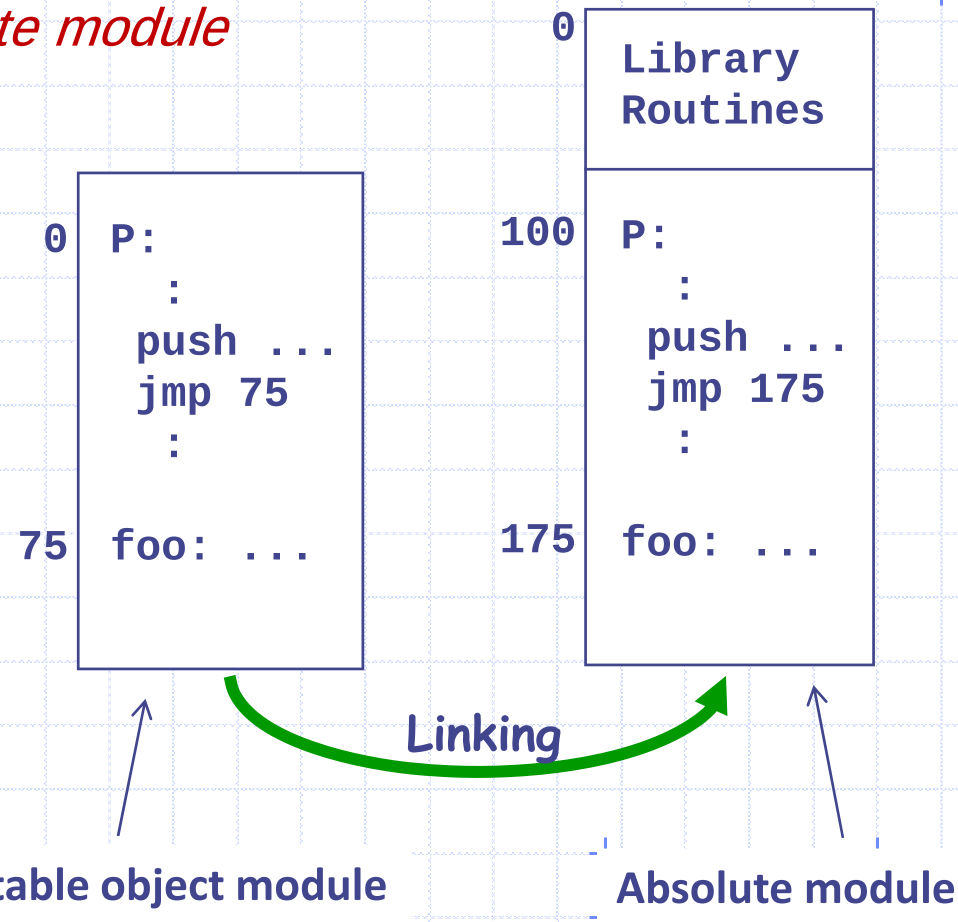
```
0   P:
       :
      push ...
      jmp 75
       :

75  foo: ...
```

Compilation

Assembly

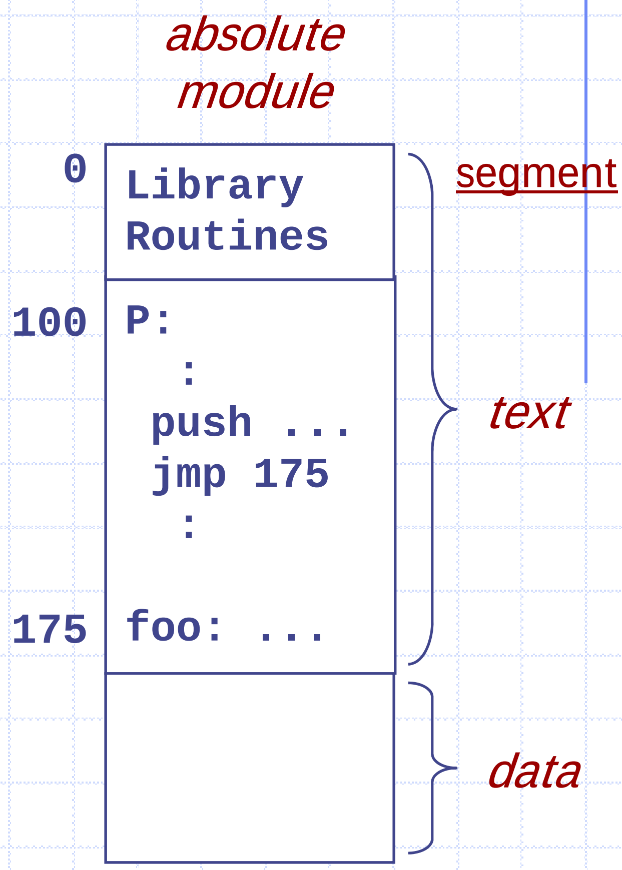
Linker

- Linker combines pieces of the program
  - combines the program with other object modules (eg: library calls)
  - linker produces an *absolute module*
  - linker assumes the program will be loaded at address 0
  - *the organization of the absolute module defines the address space*



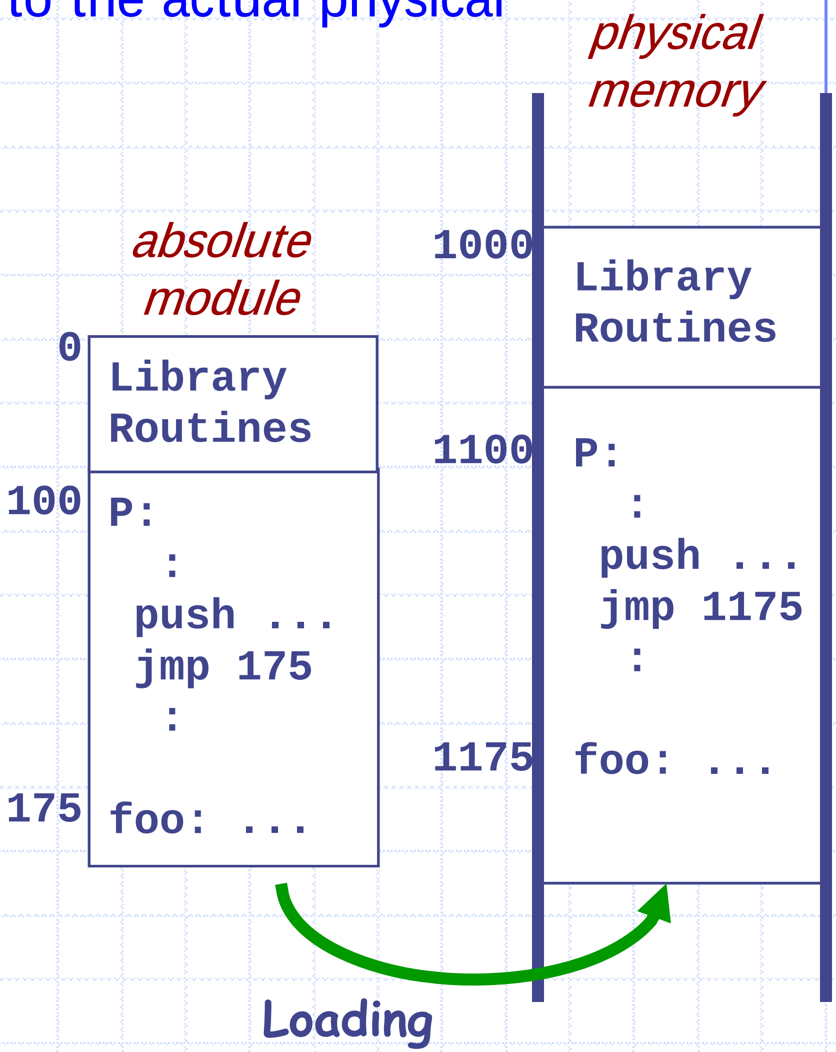
Linker (cont)

- In addition to resolving external references, linker also structures the program into segments, thereby defining the address space
- The absolute module is basically an image of the process, and includes all the non-empty segments
- The absolute module is of course much smaller than the entire address space



Loader

- Loader maps the absolute module to the actual physical memory
  - first it allocates a block of physical memory that can hold the process
  - next, all addresses in the absolute module are modified (mapped) to match the allocated physical memory addresses
  - finally, the image (code, data etc) is copied from secondary storage into the physical memory



Static Relocation

- we have been discussing a three step technique for mapping a program into memory:
  1. compile: produce a relocatable object
  2. link: produce an absolute module
  3. load: map the module into a range of allocated memory

- this technique is known as ***Static Relocation***
  - note: sometime referred to as *static address binding*

- in this technique, the address space is essentially defined by the structure of the absolute module

- the address binding occurs at when the module is loaded
  - ie: at load time

Another Example

- consider the following segment of a C program
  - it contains one static variable (with global scope)
  - it contains one function
  - it contains one function call to an external reference (put_record is extern)

```
...
static int gVar;
...
int proc_a(int arg){
        ...
        gVar = 7;
        put_record(gVar);
        ...
}
```

Another Example (after compilation)

Code Segment

Relative Address	***Generated Code***
0000	...
...	
0008	entry proc_a
...	
0220	load =7, R1
0224	store R1, 0036
0228	push 0036
0232	call 'put_record'
...	
0400	External reference table
...	
0404	'put_record' 0232
...	
0500	External definition table
...	
0540	'proc_a' 0008
...	
0600	(symbol table)
...	
0799	(end of code segment)

Data Segment

Relative Address	***Generated variable space***
...	
0036	[Space for gVar]
...	
0049	(end of data segment)

relocatable object module (includes two segments)

```
...
static int gVar;
...
int proc_a(int arg){
        ...
        gVar = 7;
        put_record(gVar);
        ...
}
```

Another Example (after linking)

Code Segment

```
Relative
Address   Generated Code

0000      (Other modules)
...
1008      entry   proc_a
...
1220      load    =7, R1
1224      store   R1, 0136
1228      push    0136
1232      call    2334
...
1399      (End of proc_a)
...
2334      entry   put_record
...
2670      (optional symbol table)
...
2999      (end of code segment)
```

Data Segment

```
Relative
Address   Generated variable space
...
0136      [Space for gVar]
...
1000      (end of data segment)
```

absolute program

- the program is relocated to 1000 to make room for other object modules being linked in
- the original data segment is relocated to 100 to make room for other data
- external entry points are resolved

Another Example (after loading)

- the memory manager allocates 4000-8000 to this process
- the absolute program is mapped into this address range

```
Physical
Address   Generated Code
0000      (Other process's programs)
...
4000      (Other modules)
...
5008      entry   proc_a
...
5220      load    =7, R1
5224      store   R1, 7136
5228      push    7136
5232      call    6334
...
5399      (End of proc_a)
...
6334      entry   put_record
...
6670      (optional symbol table)
...
6999      (end of code segment)
7000      (start of data segment)
...
7136      [Space for gVar variable]
...
8000      (Other process's programs)
```

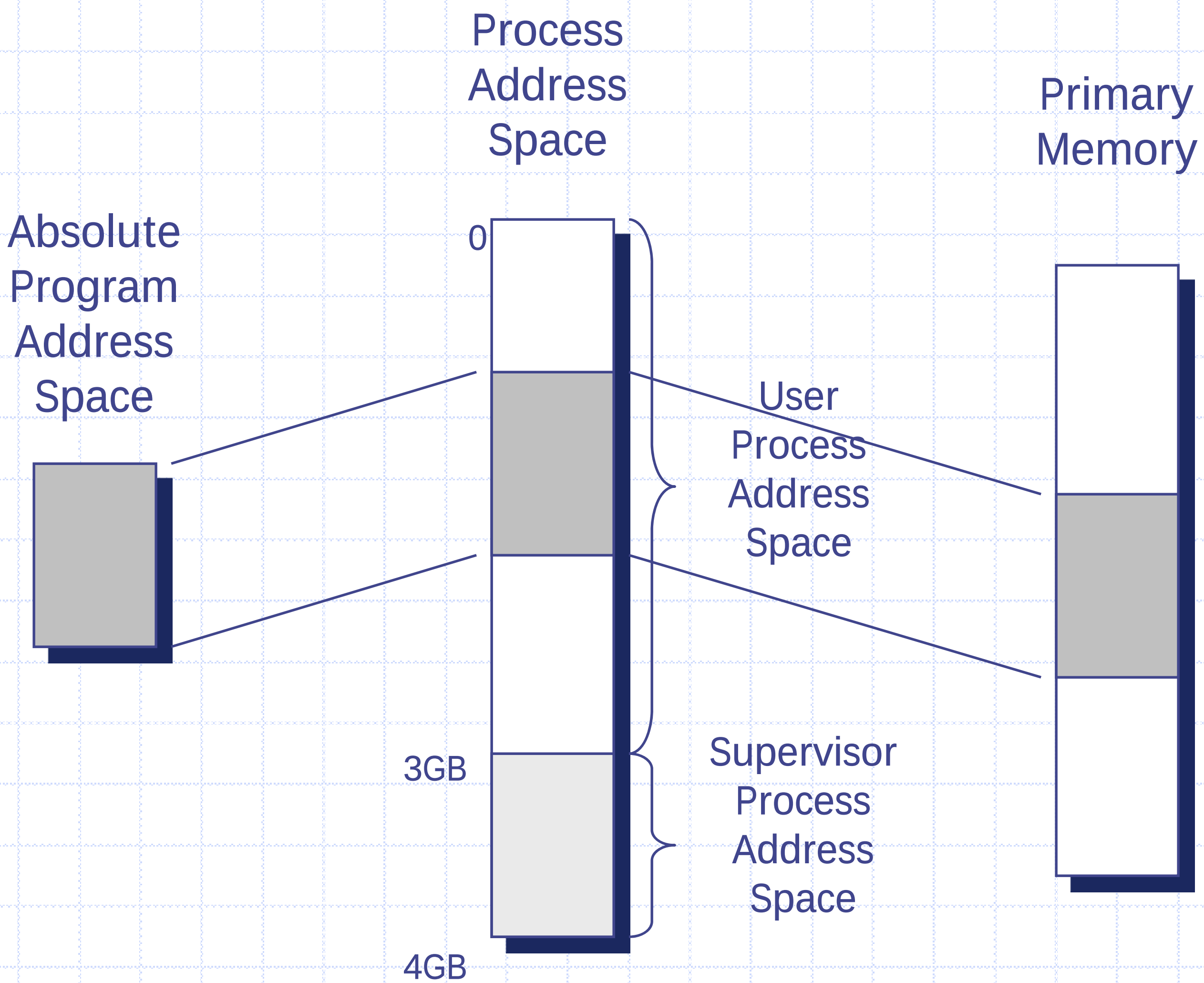
Modern Address Binding Mechanisms

- the modern process has a predefined address space
- for example linux reserves
  - 0 – 3GB for the user addressable locations
  - 3GB – 4GB for supervisor mode instructions

- with this process architecture predefined, the loader *binds the absolute module into the process address space* instead of into physical memory

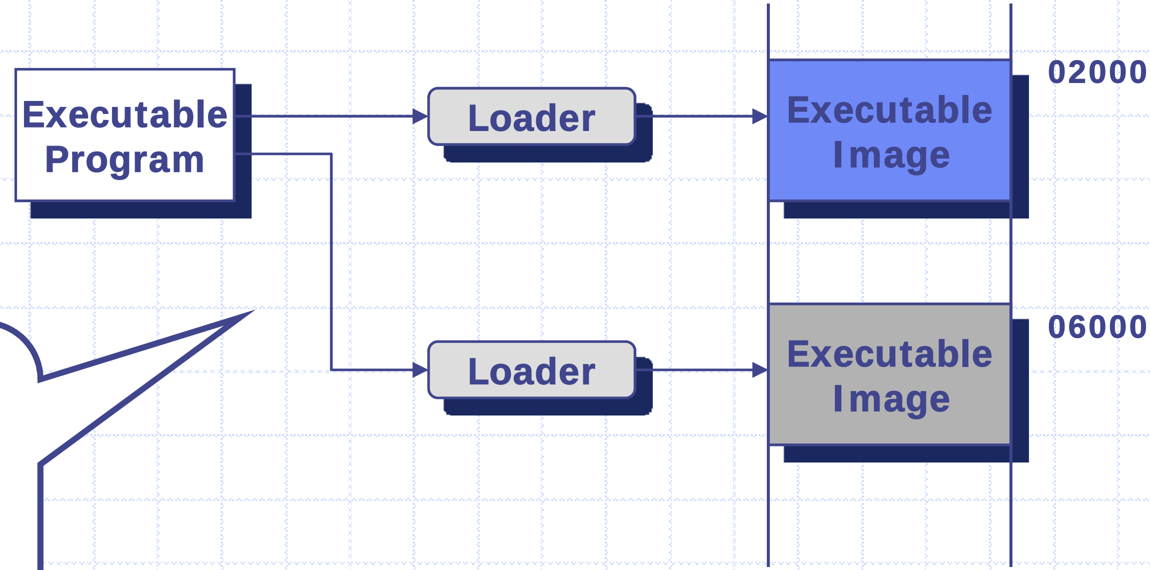
- the memory manager can then wait until *run-time* to *map the process address space into physical memory*
  - this allows the memory manager to make use of virtual memory and paging techniques (to be discussed in a subsequent lesson)
  - this also provides support for any “loadable module” mechanisms that the OS might provide

Program and Process Address Spaces



Relocating Programs in Memory

- in the previous discussion, a program always runs from the same memory location(s) after it has been loaded
- as we will see later, this can lead to memory fragmentation
- one method for dealing with fragmentation is to move programs around, to different locations
  - this requires us to re-run the loader on a program



we need a better mechanism than this to relocate programs in memory

Dynamic Relocation

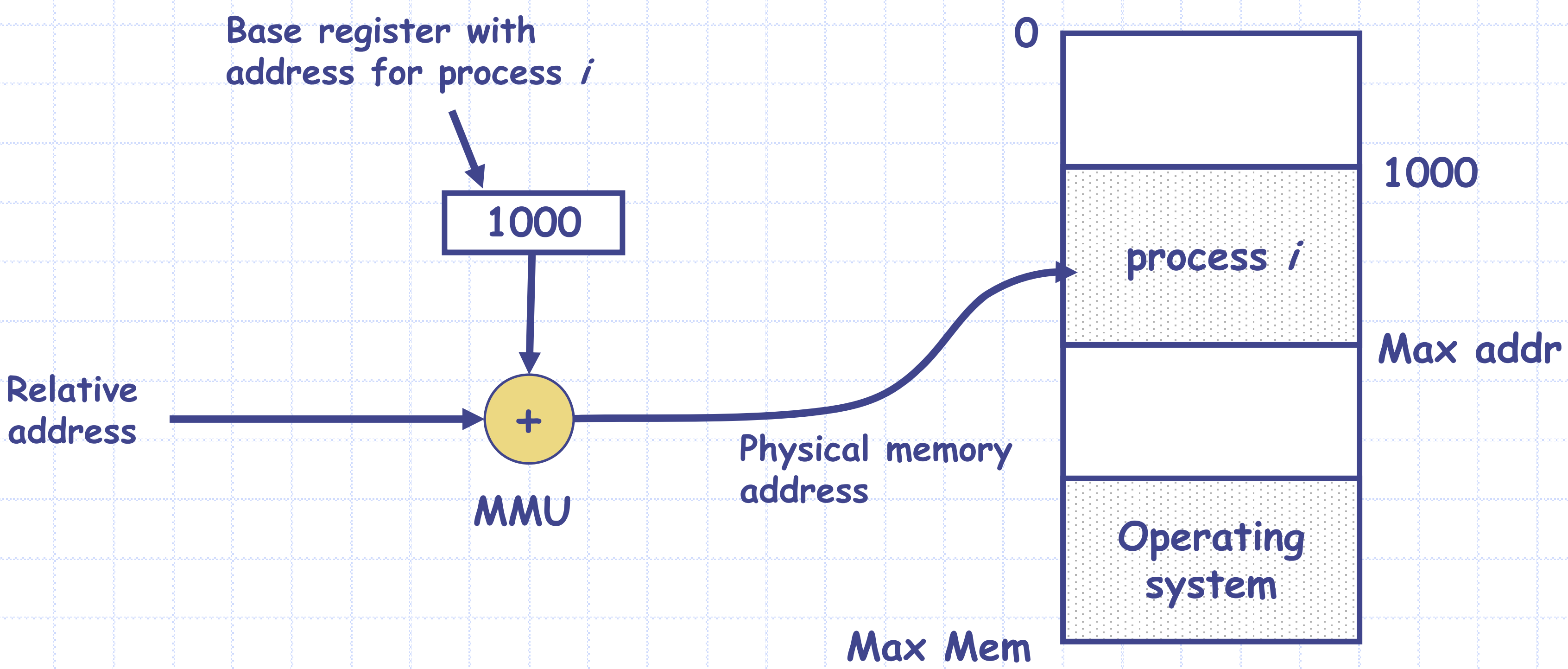
- Dynamic Relocation is:
  - mapping the physical memory to the program (or address space) at run-time
  - requires additional hardware support such as:
    - base & limit registers

- Simple runtime relocation scheme
  - use 2 registers to describe a program loaded into a partition in memory
    - base register defines the beginning of the program in memory
    - limit register is loaded with the length of the program in memory

- For every address generated, at runtime...
  - add address to the base register to give physical memory address
  - compare physical address to the limit register (& abort if larger)

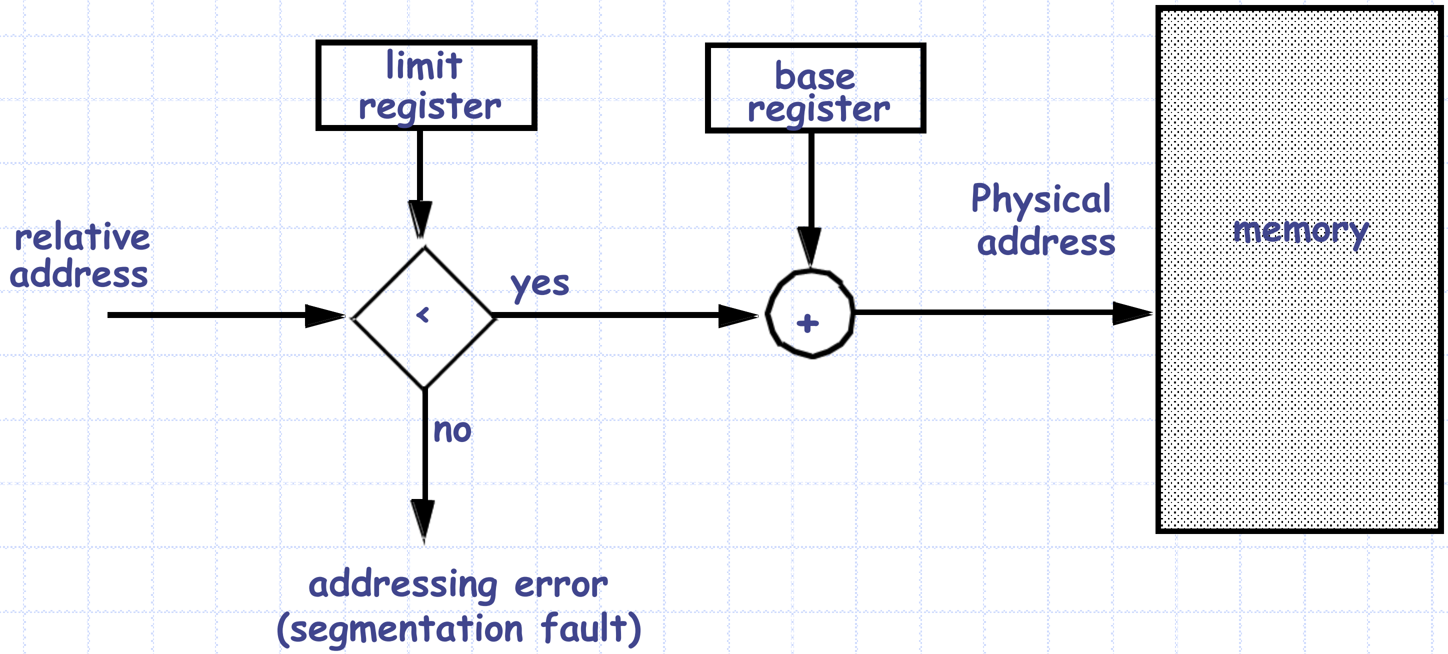
Dynamic relocation with a base register

- Memory Management Unit (MMU) - dynamically converts logical addresses into physical address
- MMU contains base address register for running process



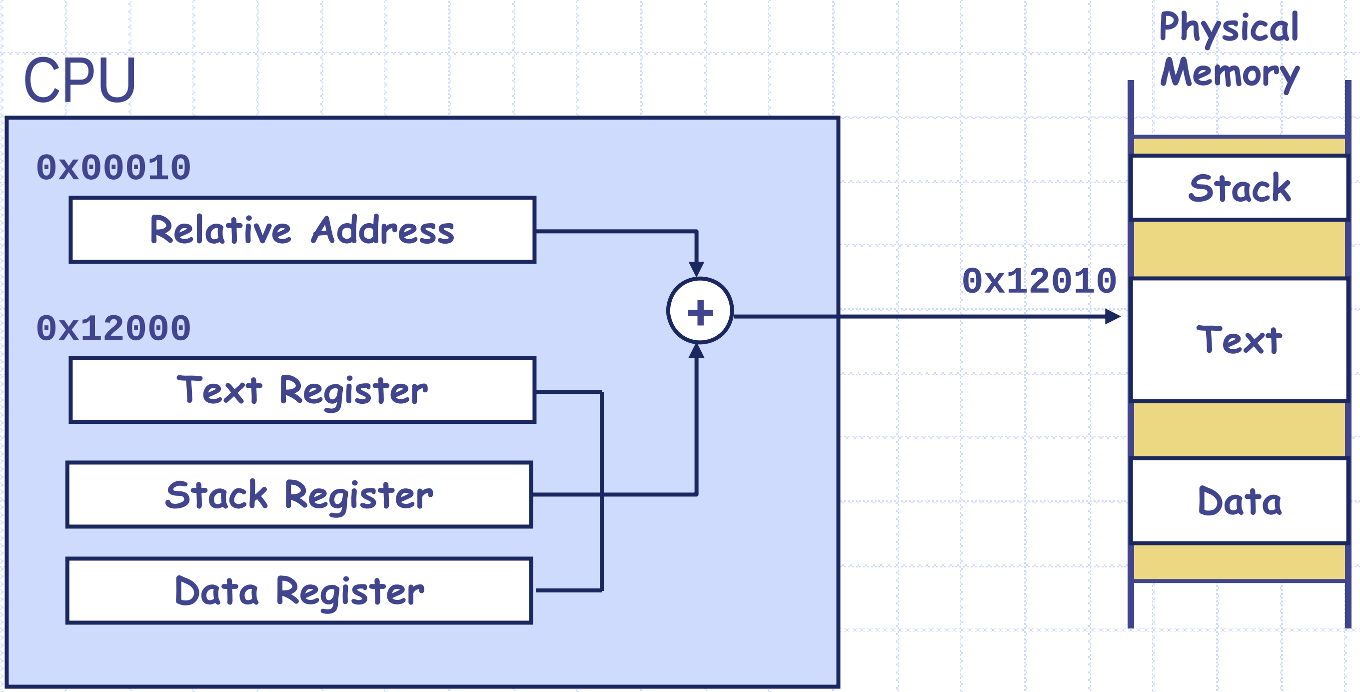
Protection using base & limit registers

- Memory protection
  - Base register gives starting address for process
  - Limit register limits the offset accessible from the relocation register



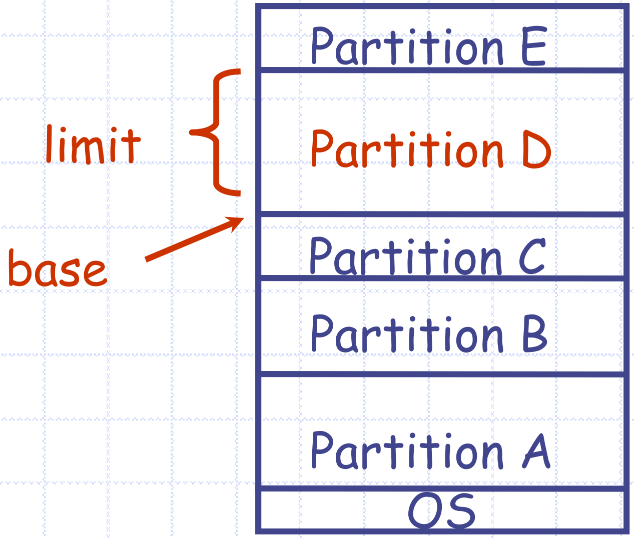
Multiple Base Registers

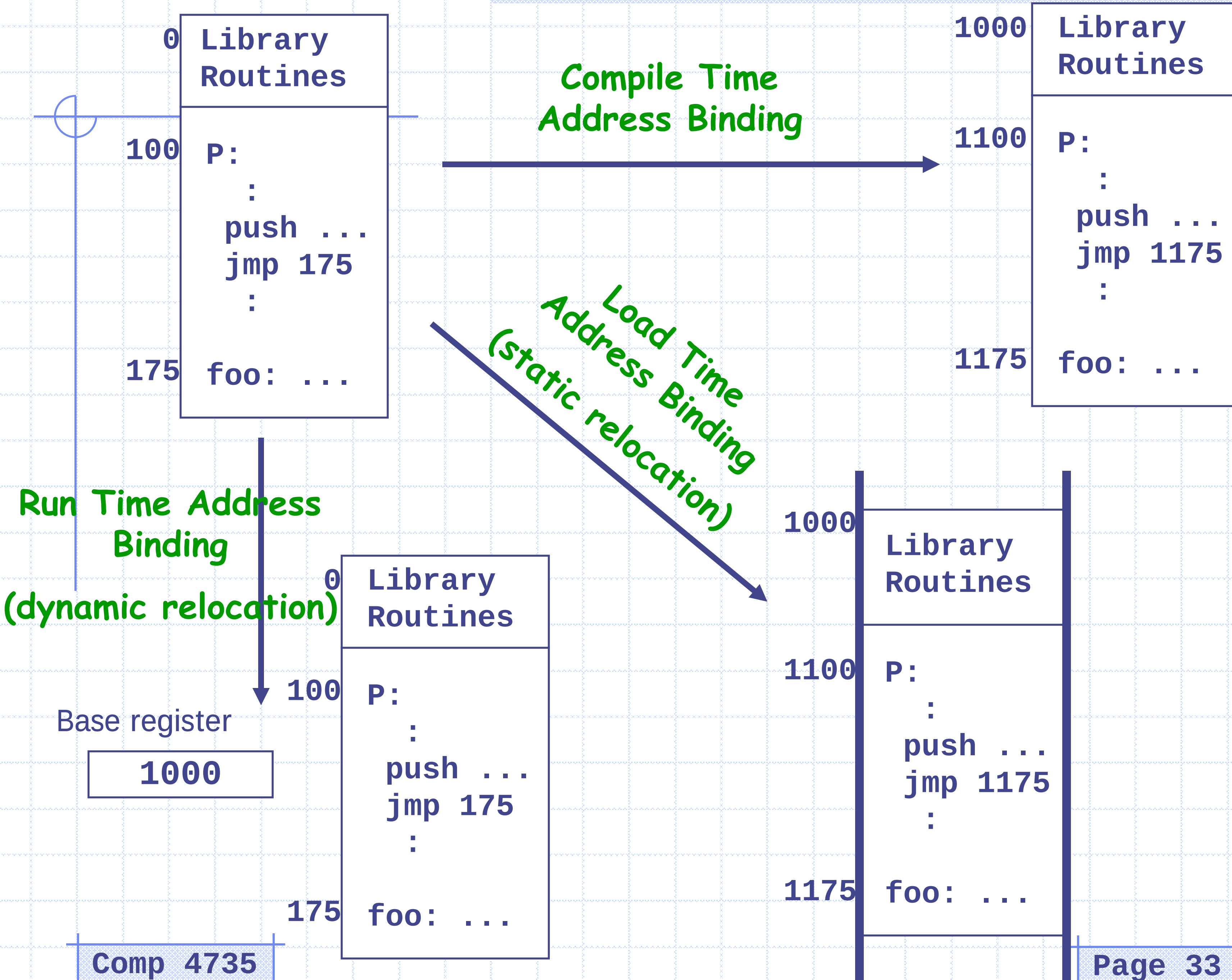
- we can use a different base register for each segment in the address space, for example:



Multiprogramming with base and limit registers

- Multiprogramming: a separate memory block (partition) for each process
- What happens on a context switch?
  - Store process A's base and limit register values
  - Load new values into base and limit registers for process B

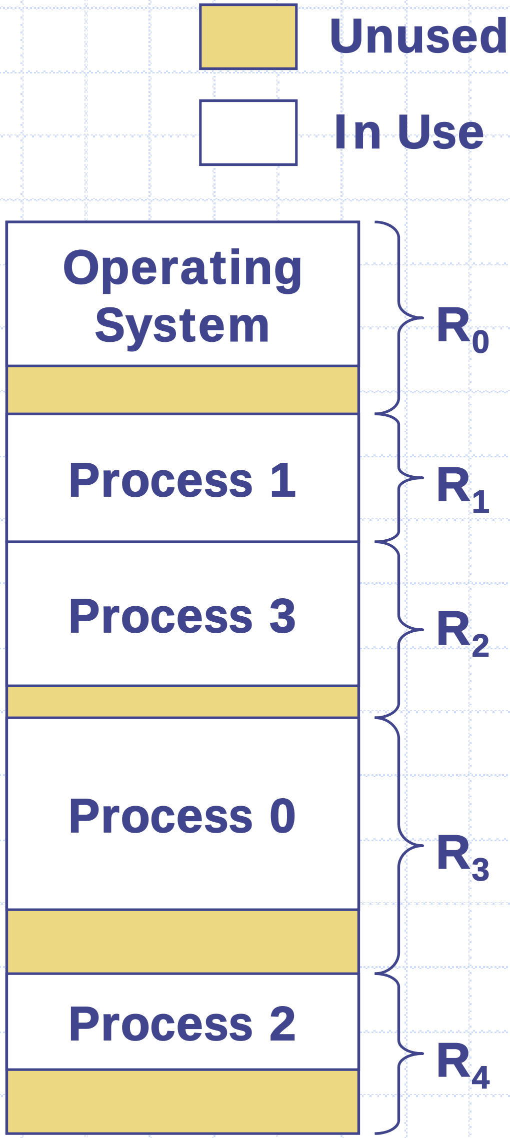



0
Library Routines
100
P:
:
push ...
jmp 175
:
175
foo: ...
Compile Time Address Binding
1000
Library Routines
1100
P:
:
push ...
jmp 1175
:
1175
foo: ...
Load Time Address Binding (static relocation)
1000
Library Routines
1100
P:
:
push ...
jmp 1175
:
1175
foo: ...
Run Time Address Binding
(dynamic relocation)
Base register
1000
0
Library Routines
100
P:
:
push ...
jmp 175
:
175
foo: ...

Memory Allocation

- to support multiprogramming, we need to allocate separate blocks (partitions) of memory to each process
- there are three basic strategies for doing this:

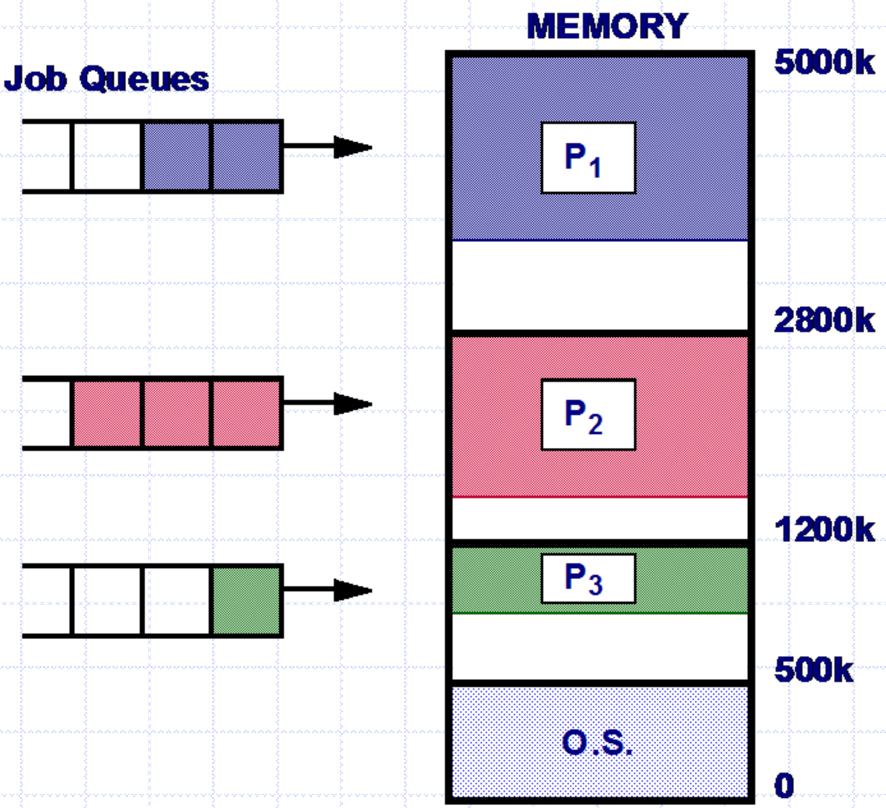
  1. use Fixed Size Partitions
  2. use Variable Size Partitions
  3. dynamically allocate memory as needed (pages … aka virtual memory)



Fixed Partition

- Memory is divided into fixed size partitions
- Processes loaded into partitions of equal or greater size

traditionally used in batch systems



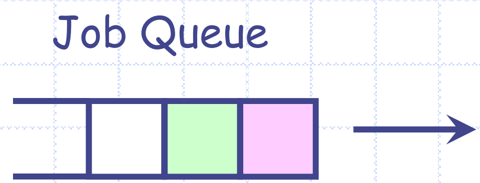
Internal Fragmentation

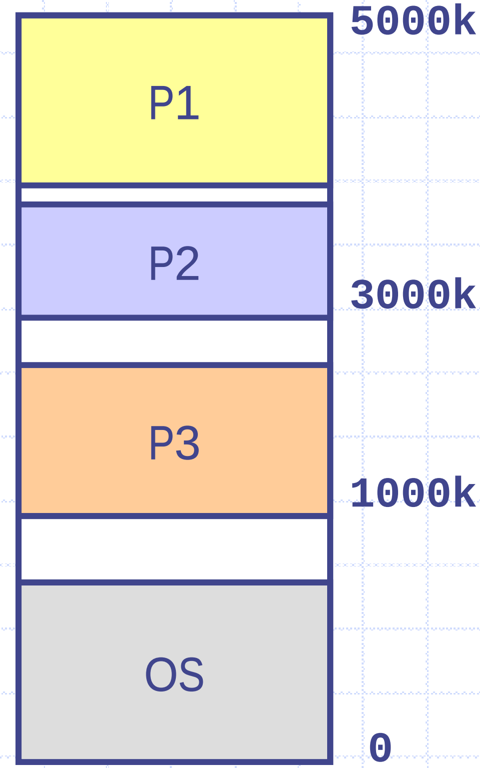
- can lead to wasted memory inside each allocated partition

Variable Partition

- Memory allocated to fit processes exactly
  - processes are loaded into regions of memory that are equal or greater size
  - there is really only one queue of jobs in this case

traditionally used in swapping systems



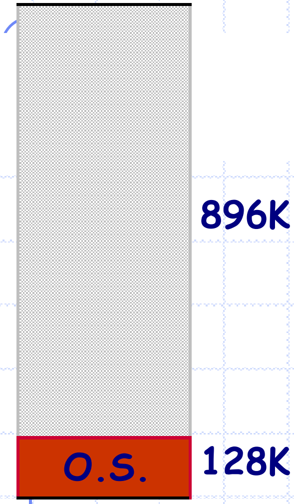


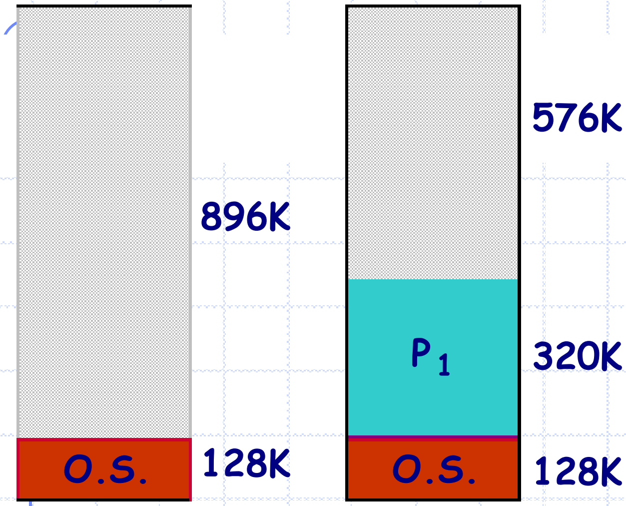
External Fragmentation

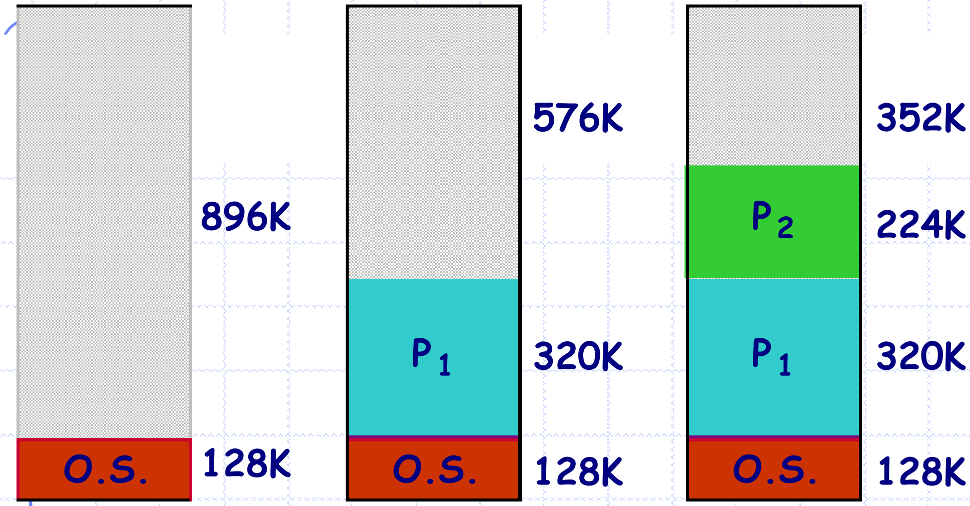
- can lead to wasted memory outside each allocated partition

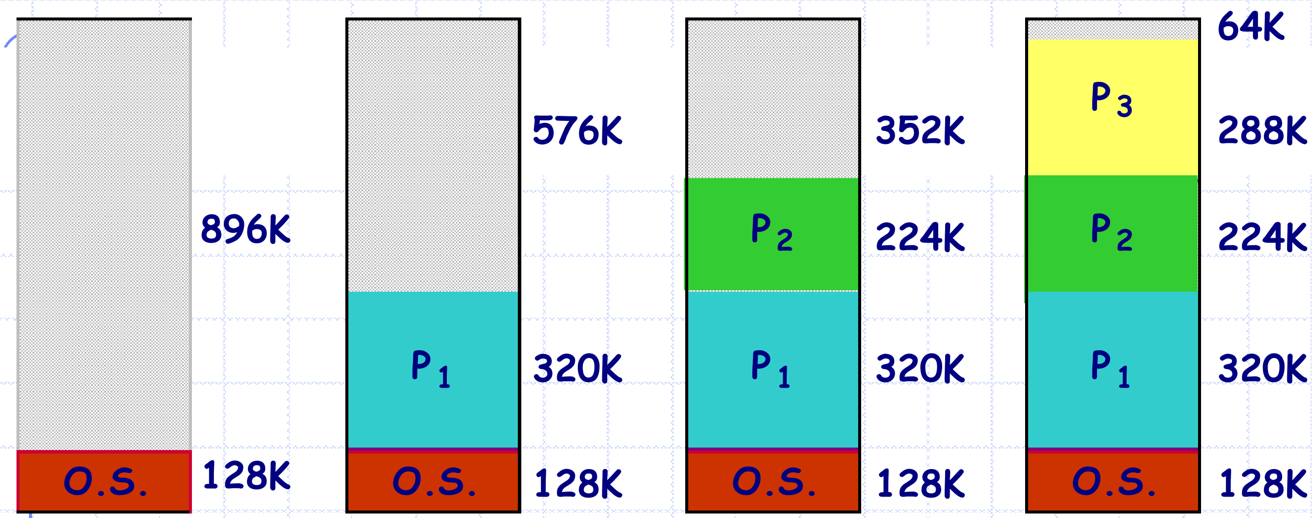
Fragmentation in Variable Partition Strategies

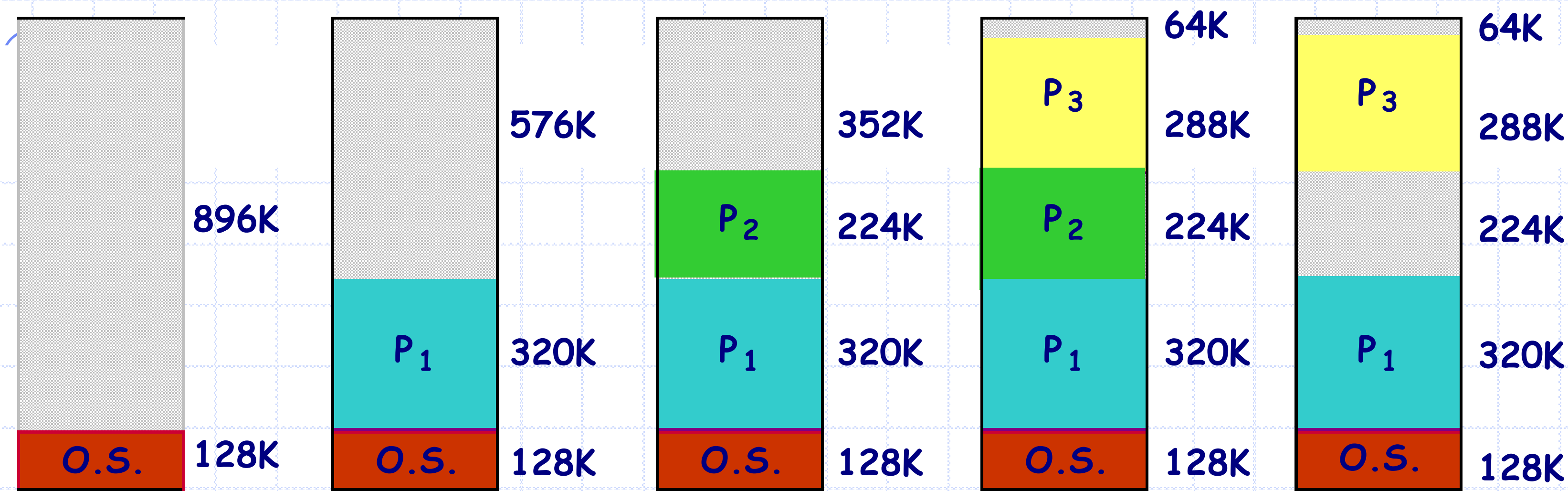
- in most types of systems we are continually loading and unloading processes
- this leads to fragmentation, for example …

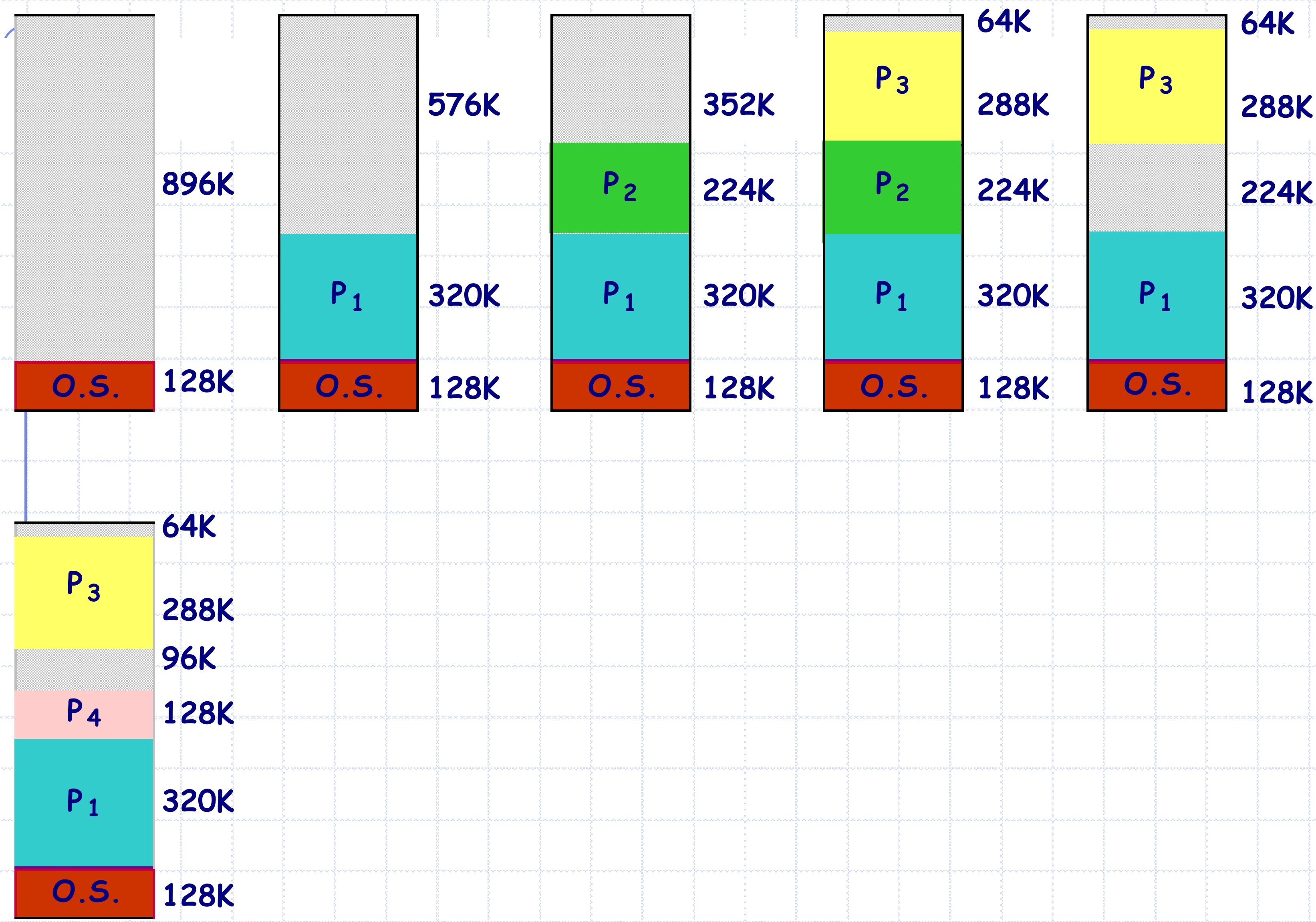

896K
O.S.
128K

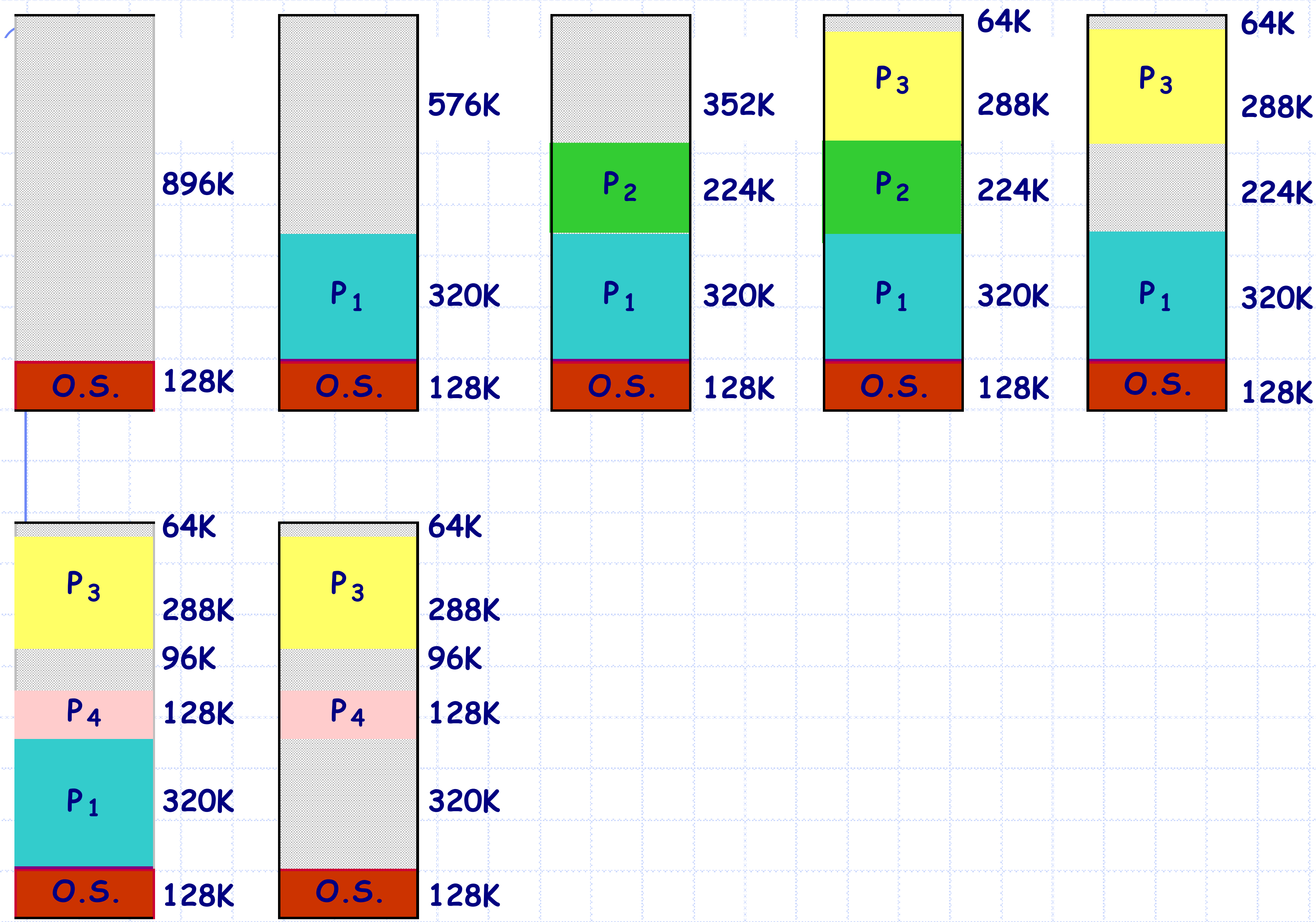

896K
O.S.
128K
576K
P 1
320K
O.S.
128K

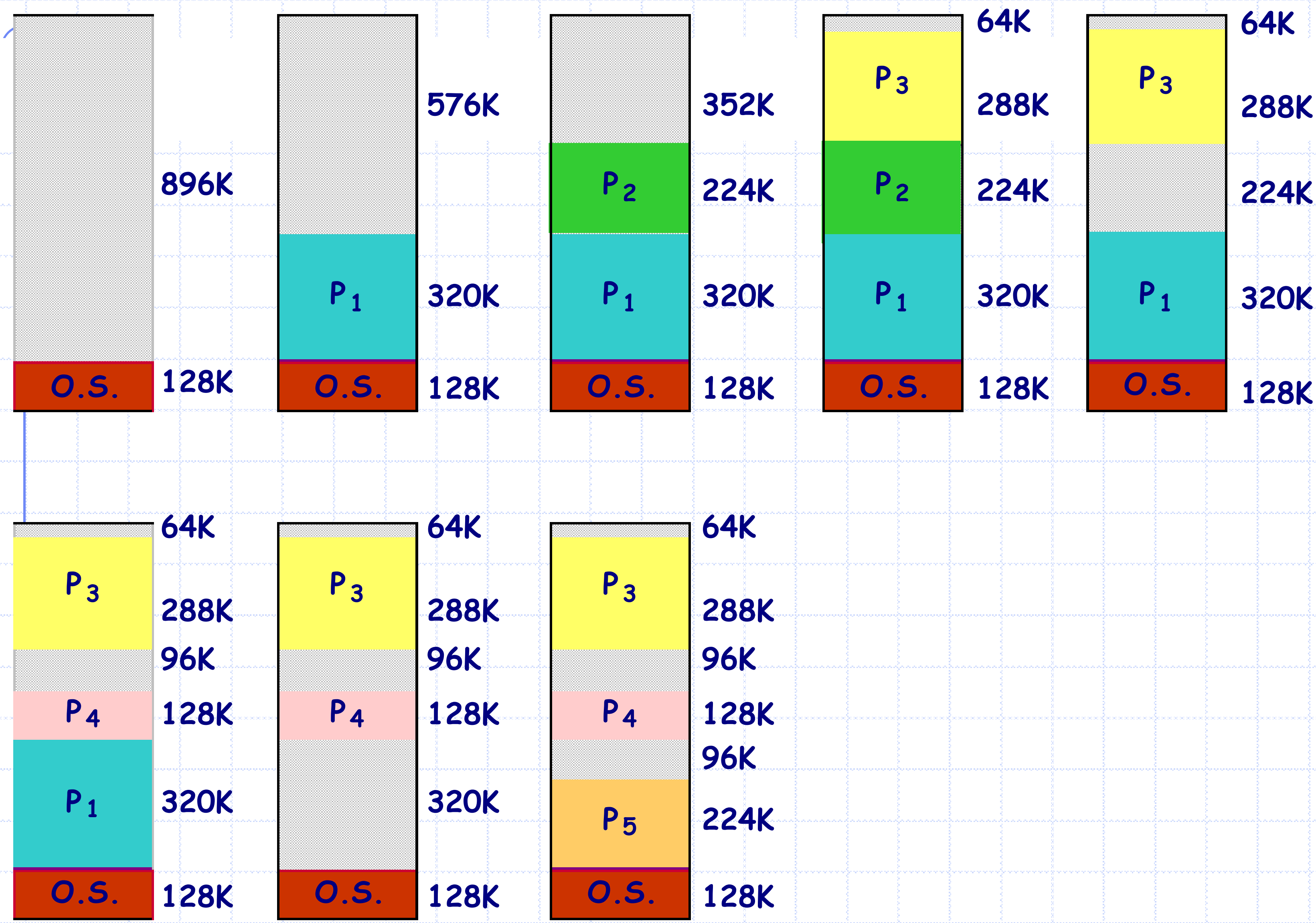

896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K


896K
O.S.
128K
576K
P 1
320K
O.S.
128K
352K
P 2
224K
P 1
320K
O.S.
128K
64K
P 3
288K
P 2
224K
P 1
320K
O.S.
128K


O.S.
128K
896K
P1
320K
576K
P2
224K
352K
P3
288K
64K


896K
O.S.
128K
576K
P 1
320K
O.S.
128K
352K
P 2
224K
P 1
320K
O.S.
128K
64K
P 3
288K
P 2
224K
P 1
320K
O.S.
128K
64K
P 3
288K
224K
P 1
320K
O.S.
128K
64K
P 3
288K
96K
P 4
128K
P 1
320K
O.S.
128K

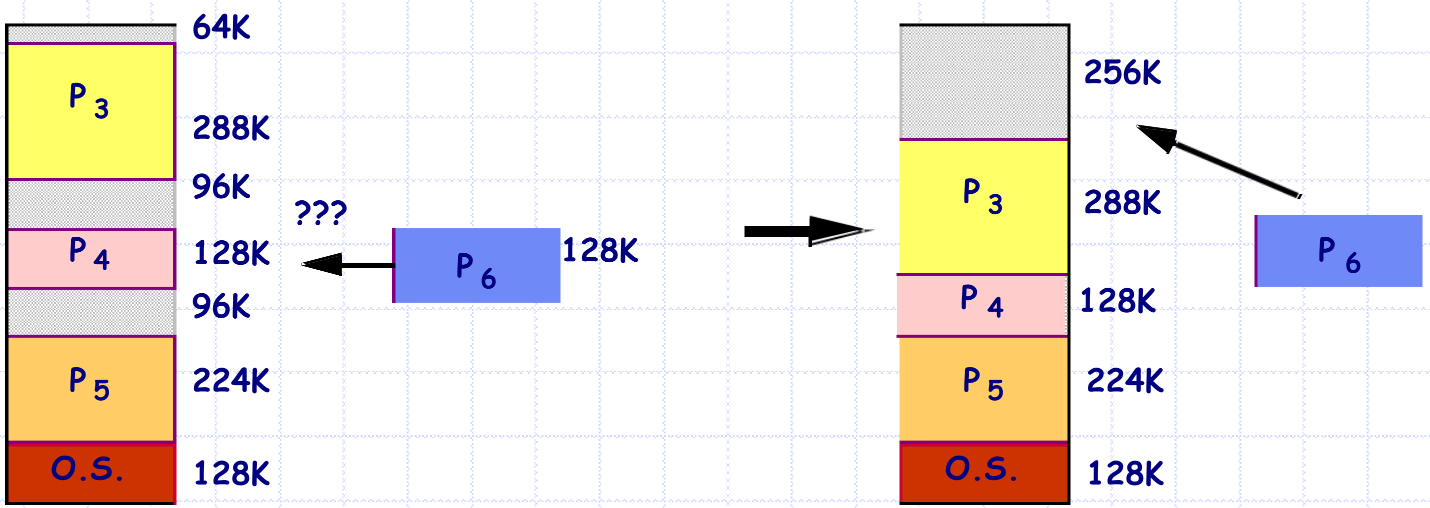

O.S.
128K
896K
P1
576K
320K
P2
352K
224K
P3
64K
288K
96K
P4


896K
O.S.
128K
576K
P1
320K
O.S.
128K
352K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
P2
224K
P1
320K
O.S.
128K
64K
P3
288K
224K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
P1
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
320K
O.S.
128K
64K
P3
288K
96K
P4
128K
96K
P5
224K
O.S.
128K

O.S.
128K
896K
P1
320K
576K
P2
224K
352K
P3
288K
64K
P4
96K
P5
96K
???
P6
128K

Dealing with fragmentation

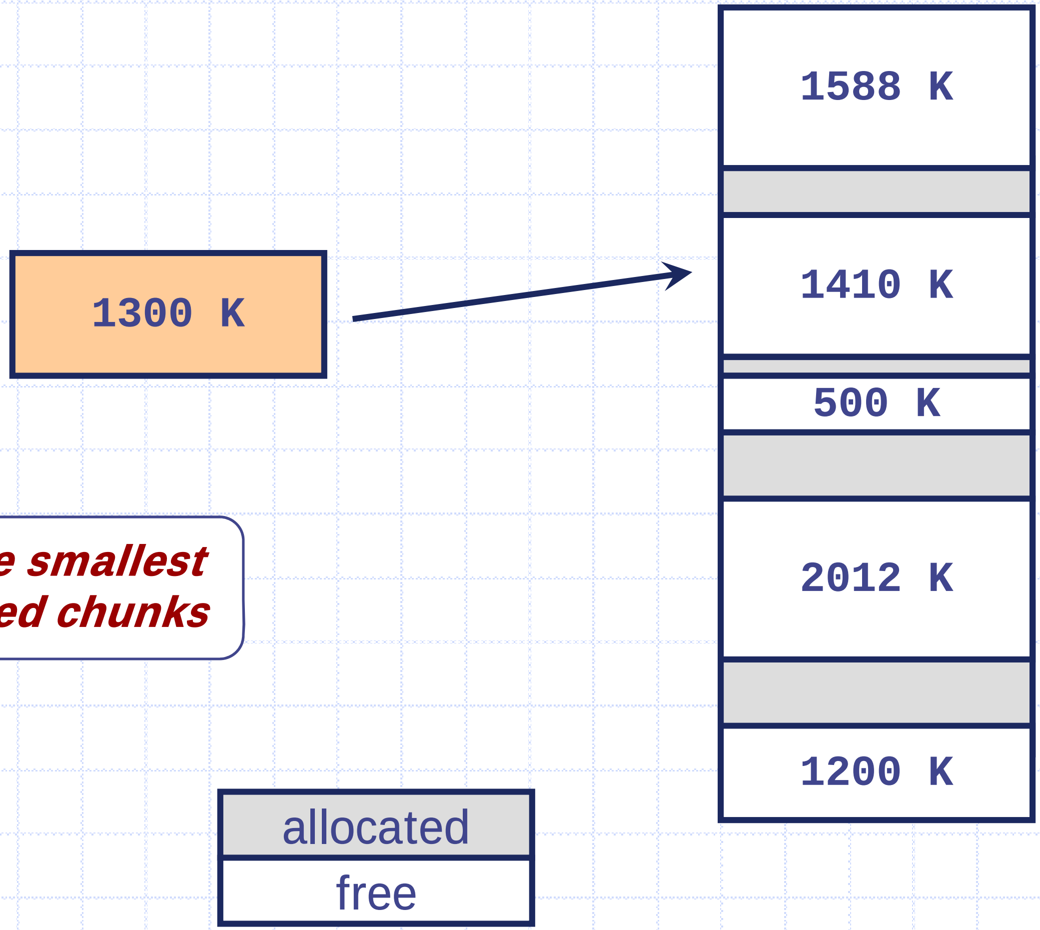
- *Compaction* – from time to time we move the processes around to collect all free space into one contiguous block



- Allocation Strategies: First-fit, best-fit, worst-fit
  - these affect the degree of fragmentation

Best Fit Memory Allocation

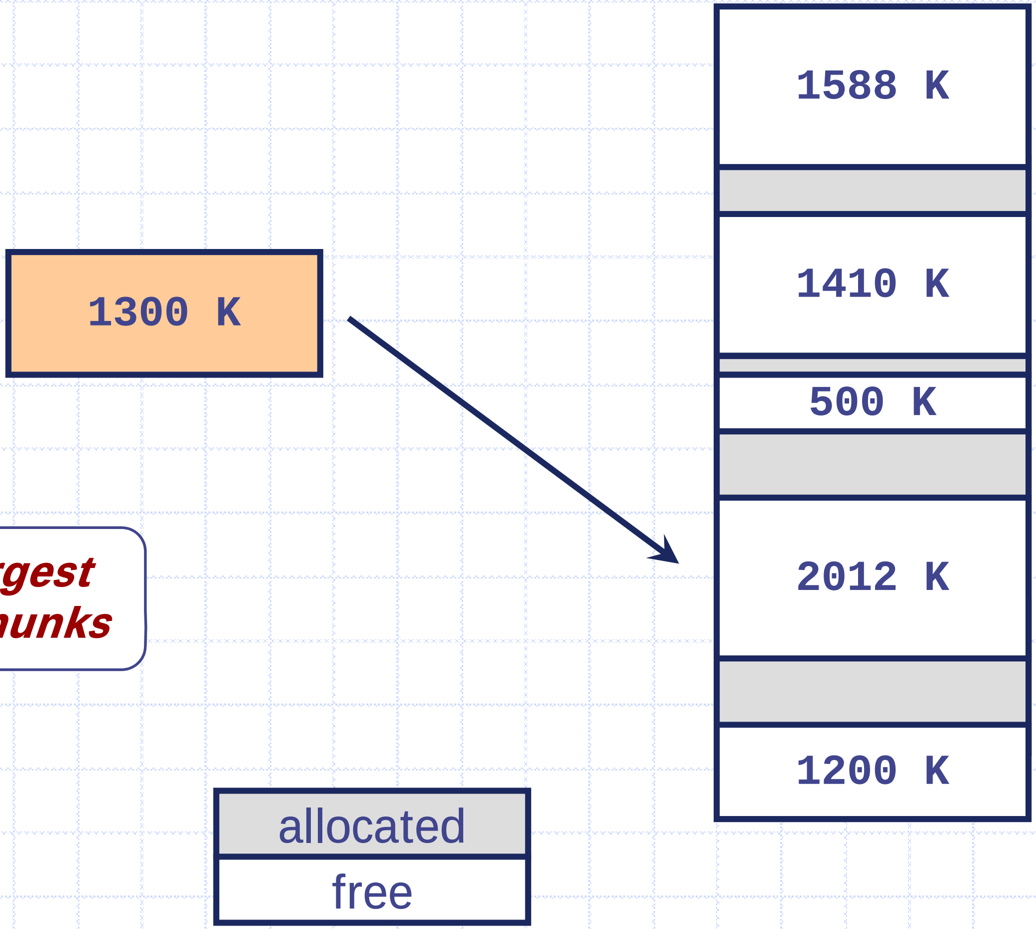
- allocate the smallest block of free memory in which the process will fit



leaves the smallest unallocated chunks

Worst Fit Memory Allocation

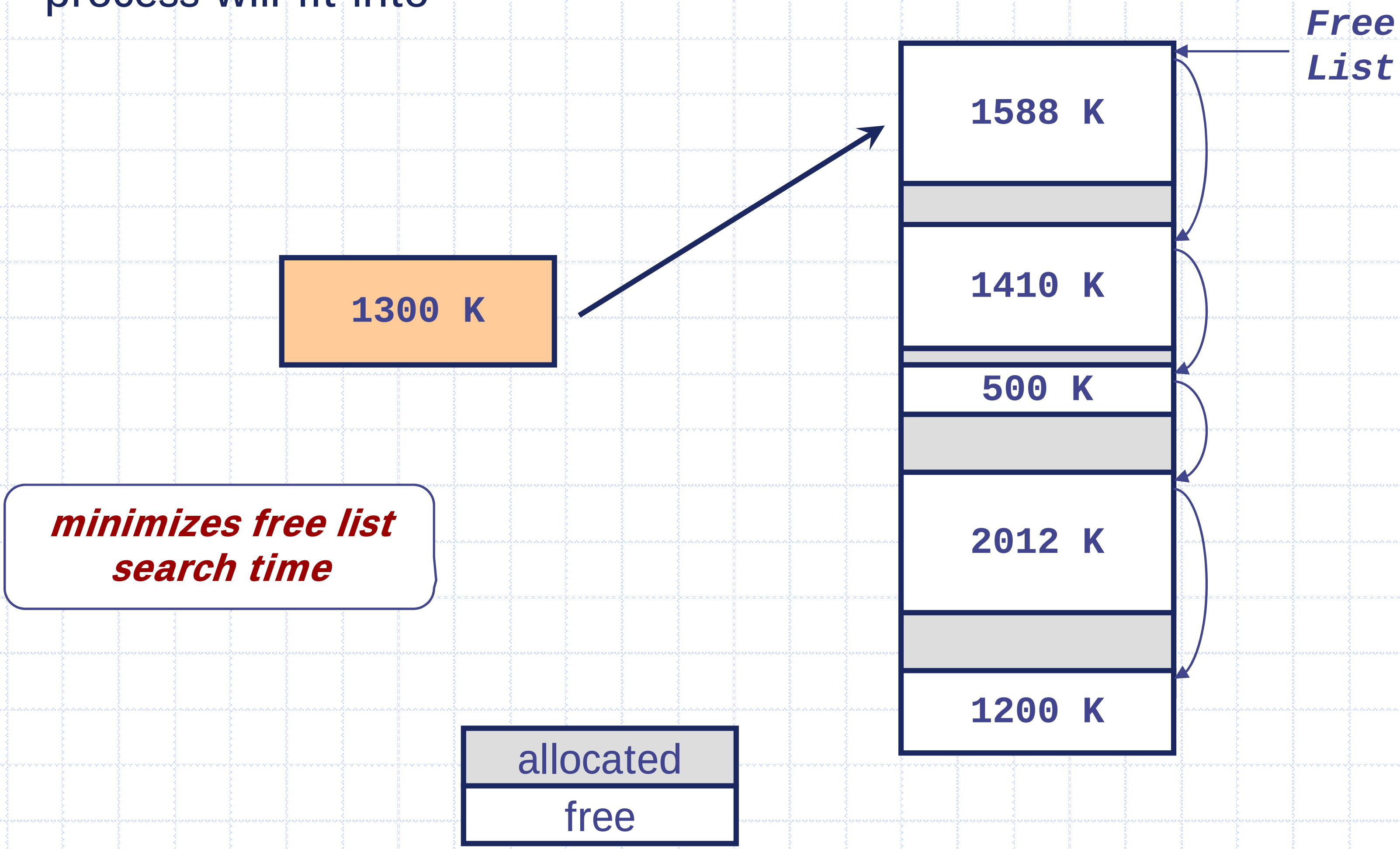
- allocate the largest block of free memory
  - as long as the process will fit



leaves the largest unallocated chunks

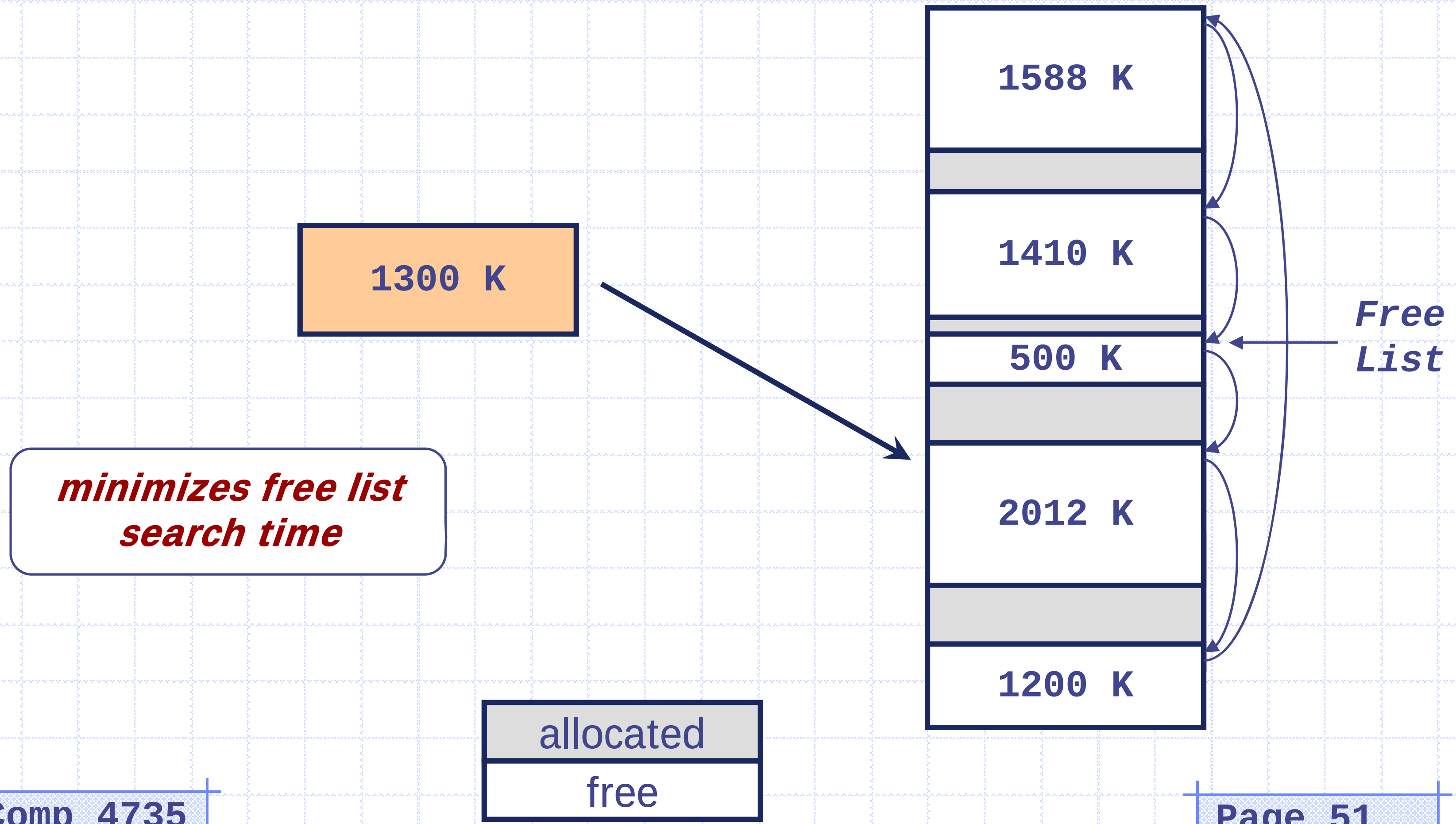
First Fit Memory Allocation

- allocate the first block of memory (from the free-list) that the process will fit into

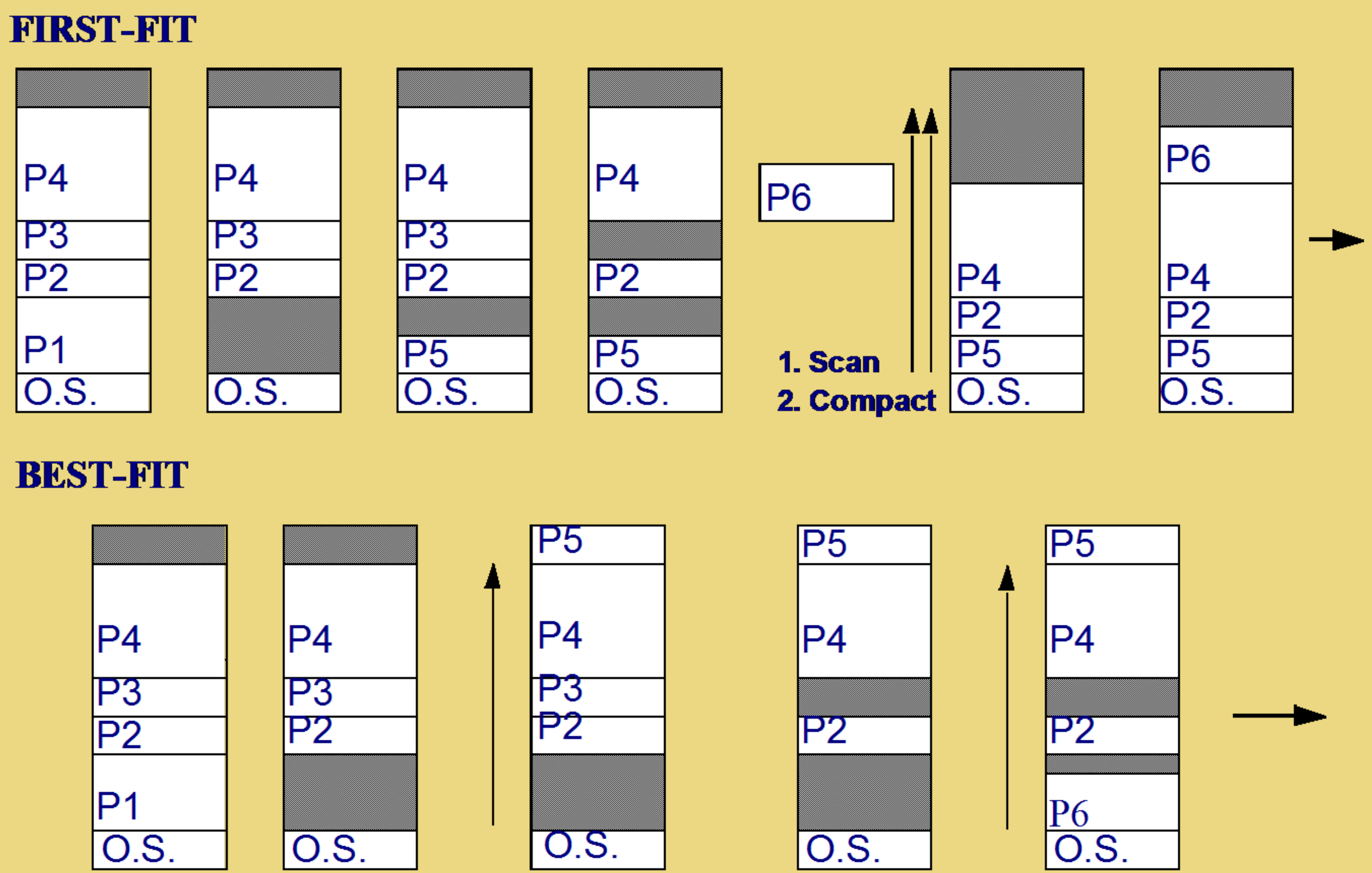


Next Fit Memory Allocation

- similar to first fit, but free-list is a circular list
  - start search at the first free block following the last one allocated



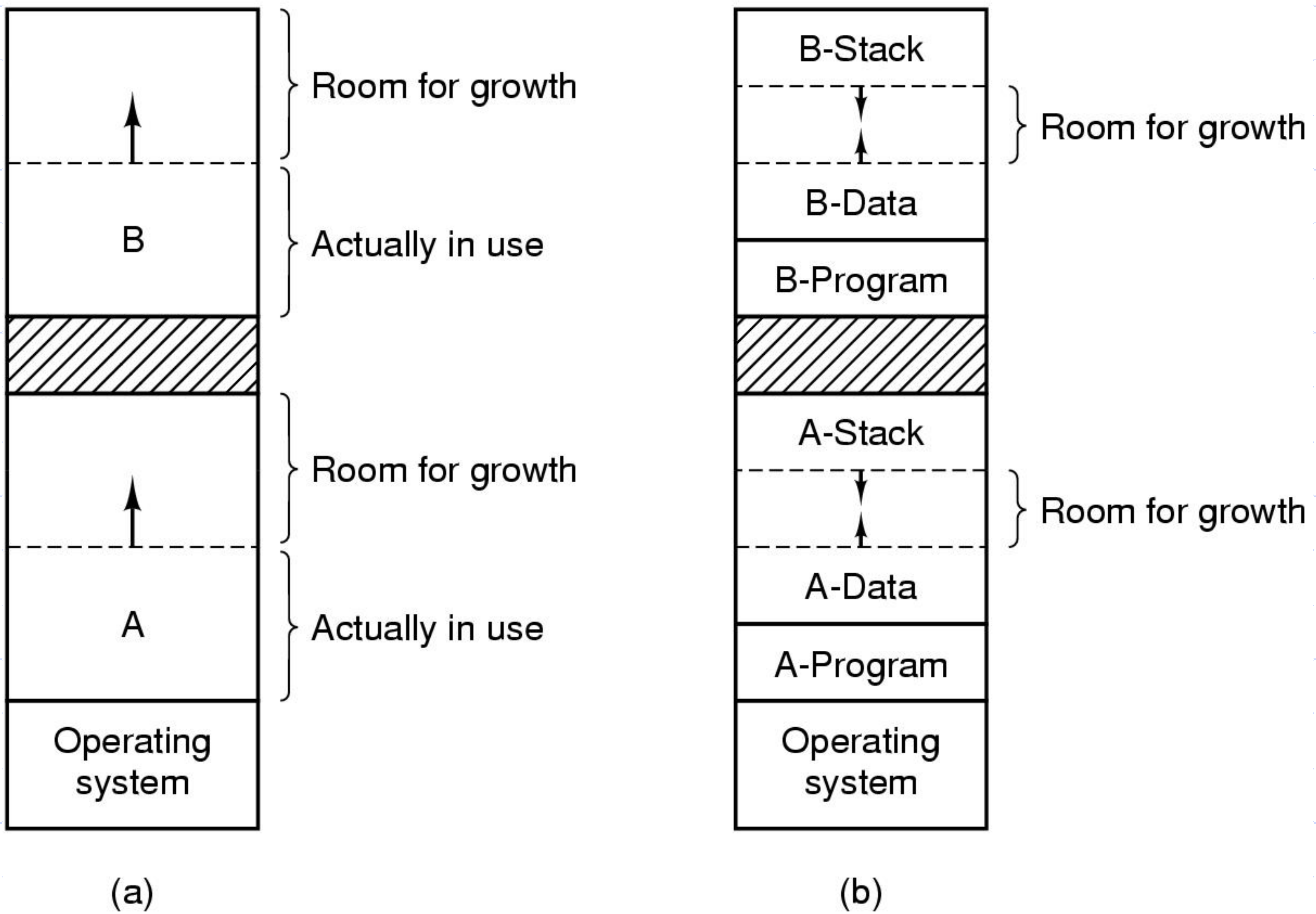
Influence of allocation policy



How big should partitions be?

- Programs may want to grow during execution
  - More room for stack, heap allocation, etc
- Problem:
  - If the partition is too small programs must be moved
  - Requires modification of base and limit regs
  - Why not make the partitions a little larger than necessary to accommodate “some” growth?
- Fragmentation:
  - External fragmentation = unused space between partitions
  - Internal fragmentation = unused space within partitions

Allocating extra space within partitions



Managing free memory

- Each chunk of memory is either
  - Used by some process or unused (“free”)

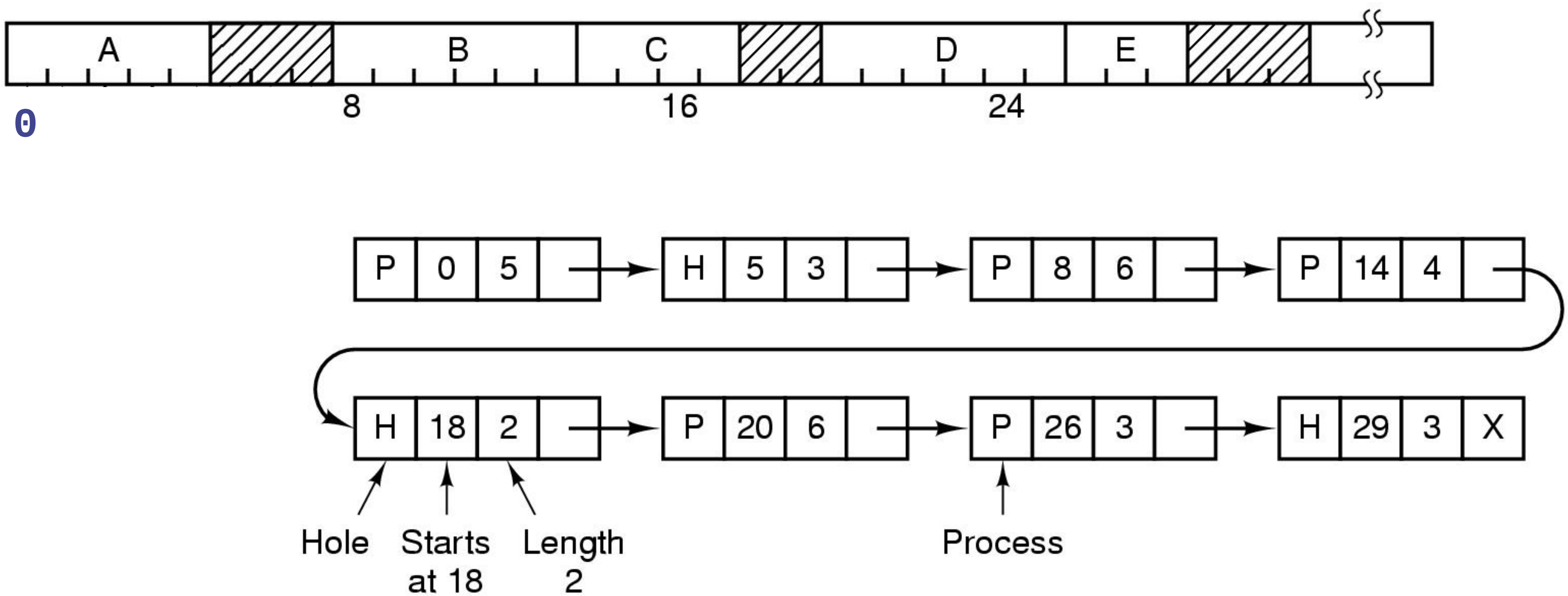
- Operations
  - Allocate a chunk of unused memory big enough to hold a new process
  - Free a chunk of memory by returning it to the free pool after a process terminates or is swapped out

Managing memory with linked lists

- Problem - we need to keep track of used and unused memory
- Technique: use a Linked List
- Keep a list of elements
- Each element describes one unit of memory
  - Free / in-use Bit ("P=process, H=hole")
  - Starting address
  - Length
  - Pointer to next element

Managing memory with linked lists

- the free-list structures indicate where processes and holes start
- each block of memory (free or otherwise) is on the list

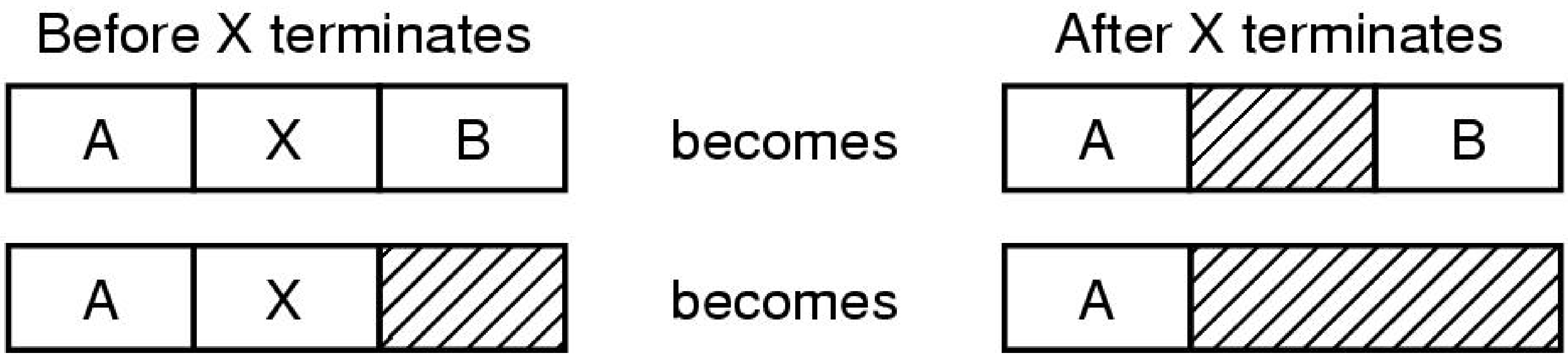


Merging holes

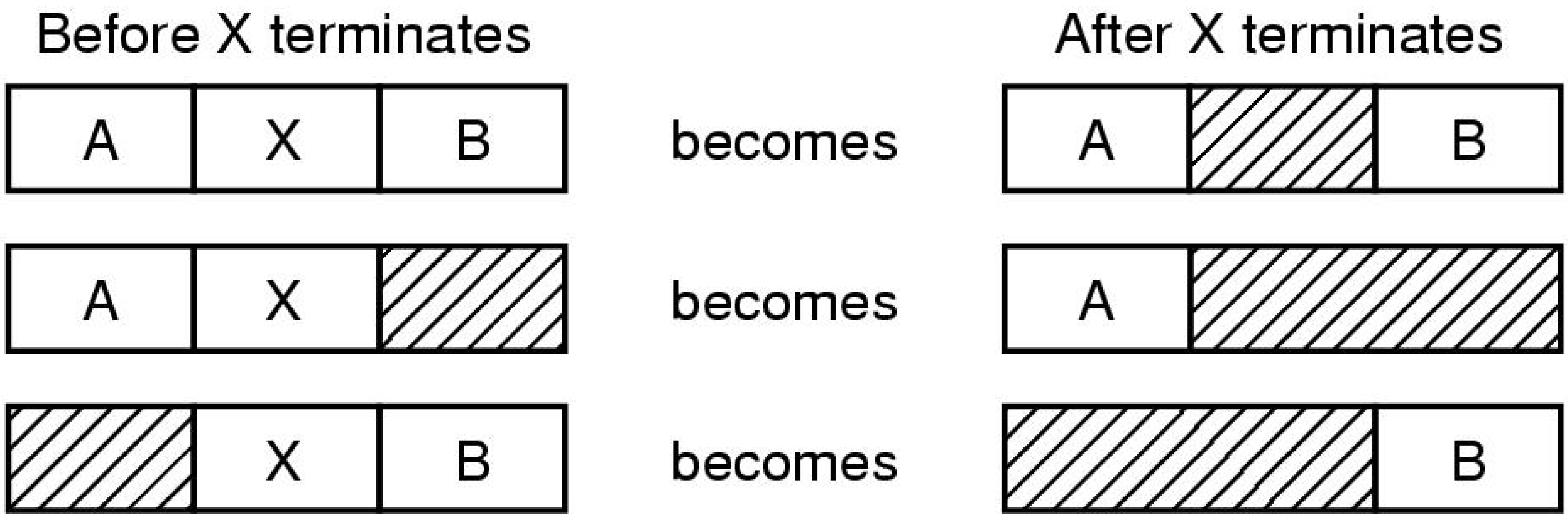
- Whenever a unit of memory is freed we want to merge adjacent holes …

Merging holes

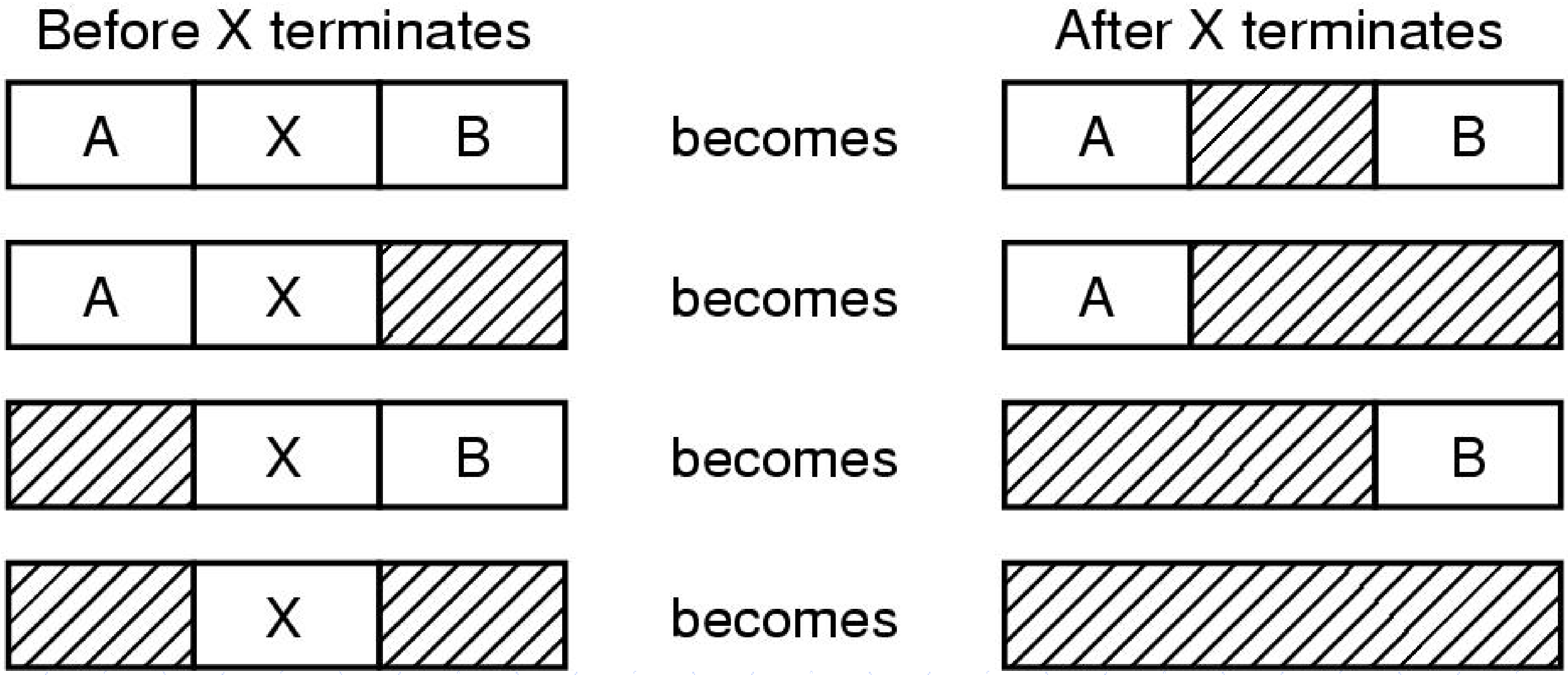
Merging holes



Merging holes



Merging holes

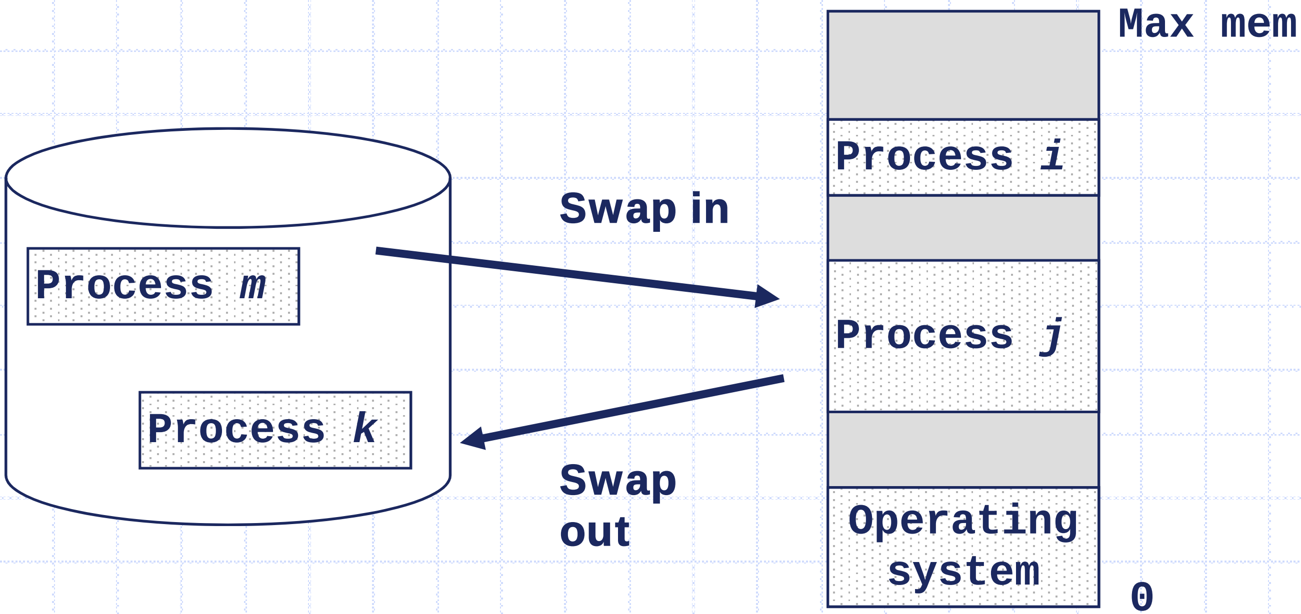


Swapping

- When a program is running…
  - The entire program must be in memory
  - Each program is put into a single partition
- When the program is not running…
  - May remain resident in memory
  - May get "*swapped*" out to disk
- If we consider the life of a program over time…
  - Programs come into memory when they get swapped in
  - Programs leave memory when they get swapped out
  - A program may execute out of many partitions as it is swapped into and out of memory

Basics - swapping

- Benefits of swapping:
  - Allows multiple programs to be run concurrently
  - ... more than will fit in memory at once



The End