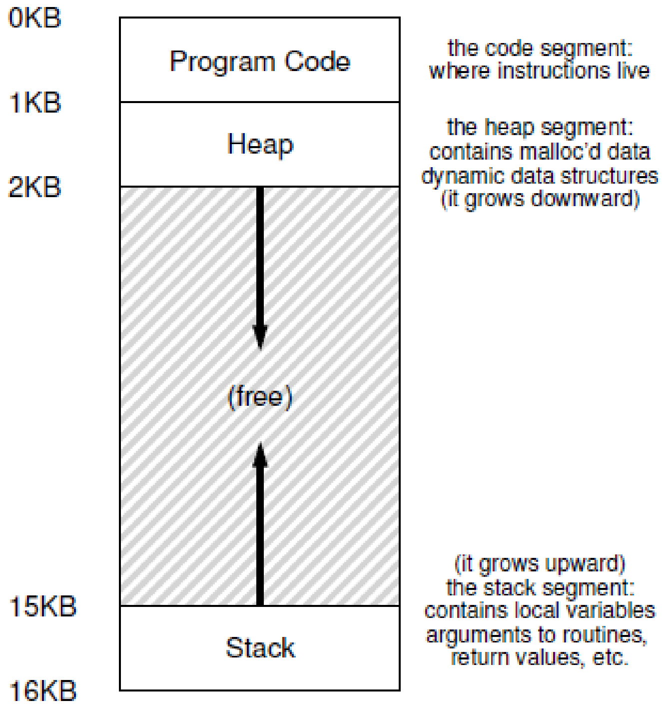
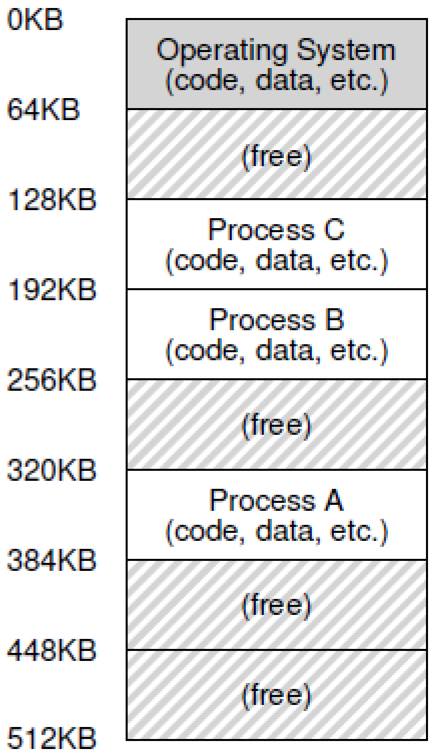
Memory
Chap 13, 15, 16, 17

Address Spaces

- Easy to use abstraction of physical memory
- Running program's view of memory
- OS build this abstraction of a private, potentially large address space for multiple running processes (all sharing memory) on top of a single, physical memory



Virtual view



Physical memory in multiprogramming

Address Translation

- **Logical address (virtual address)**
  - An address generated by the CPU
- **Physical address**
  - An address seen by the memory unit
  - An address loaded into MAR

Every Address You See is Virtual

```
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[]) {
        printf("location of code : %p□n", (void *) main);
        printf("location of heap : %p□n", (void *) malloc(1));
        int x = 3;
        printf("location of stack : %p□n", (void *) &x);
        return x;
}
```

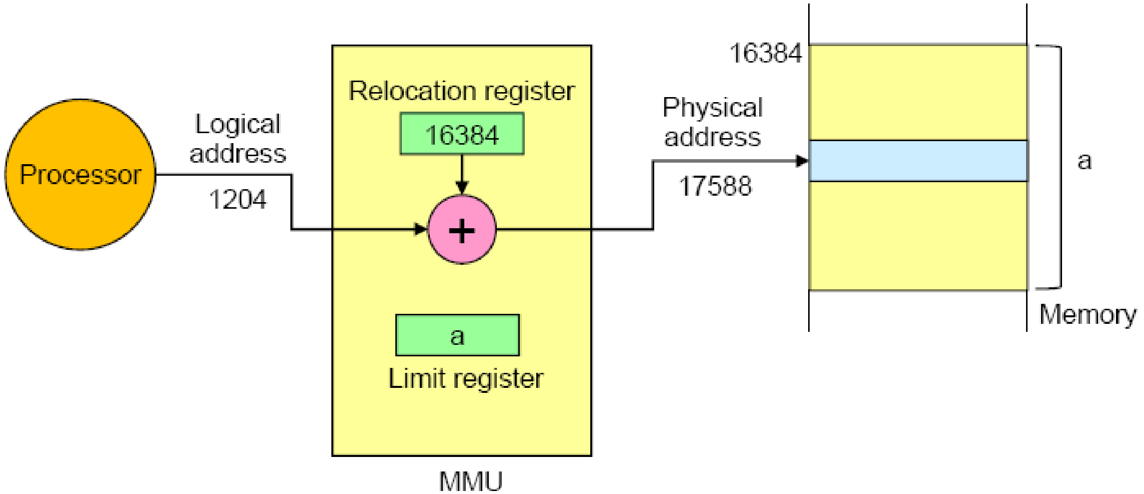
```
location of code : 0x1095afe50
location of heap : 0x1096008c0
location of stack : 0x7fff691aea64
```

Goals of Memory Virtualization

- **Transparency**
  - Program shouldn't be aware of the fact that memory is virtualized
  - Program behaves as if it has its own private physical memory
  - OS (and hardware) does all the work to multiplex memory among many different jobs, and hence implements the illusion
- **Efficiency**
  - Time: not making programs run much more slowly
    - Need H/W support (TLB)
  - Space: not using too much memory for structures needed to support virtualization
- **Protection**
  - protect processes from one another as well as the OS itself from processes
  - deliver the property of isolation among processes

Dynamic (Hardware-based) Relocation

- **Runtime Binding**
  - two hardware registers, base and bounds (limit), in MMU
- each program is written and compiled as if it is loaded at address zero.
- When a program starts running, the OS decides where in physical memory it should be loaded and sets the base register to that value.
- When any memory reference is generated by the process, it is **translated** by the processor in the following manner:
  - physical address = virtual address + base
- A **bounds** (or **limit**) register ensures that such addresses are within the confines of the address space.



Dynamic (Hardware-based) Relocation

- The hardware should provide special **privileged** instructions to modify the base and bounds registers, allowing the OS to change them when different processes run.
- CPU must be able to generate **exceptions** in situations where a user program tries to access memory illegally (with an address that is "out of bounds")

Hardware Requirements	Notes
Privileged mode	*Needed to prevent user-mode processes from executing privileged operations*
Base/bounds registers	*Need pair of registers per CPU to support address translation and bounds checks*
Ability to translate virtual addresses and check if within bounds	*Circuitry to do translations and check limits; in this case, quite simple*
Privileged instruction(s) to update base/bounds	*OS must be able to set these values before letting a user program run*
Privileged instruction(s) to register exception handlers	*OS must be able to tell hardware what code to run if exception occurs*
Ability to raise exceptions	*When processes try to access privileged instructions or out-of-bounds memory*

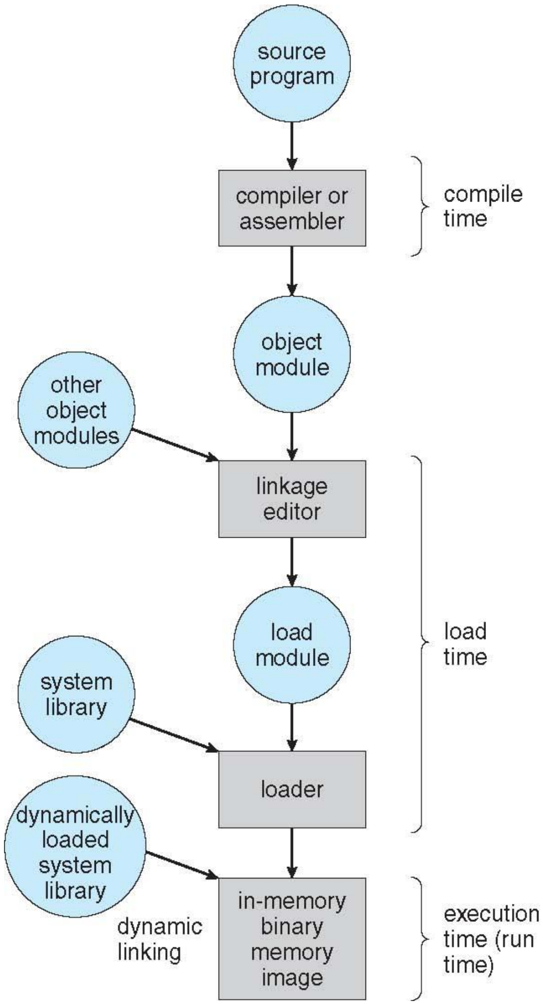
Dynamic (Hardware-based) Relocation

- OS must save and restore the base-and-bounds pair when it switches between processes

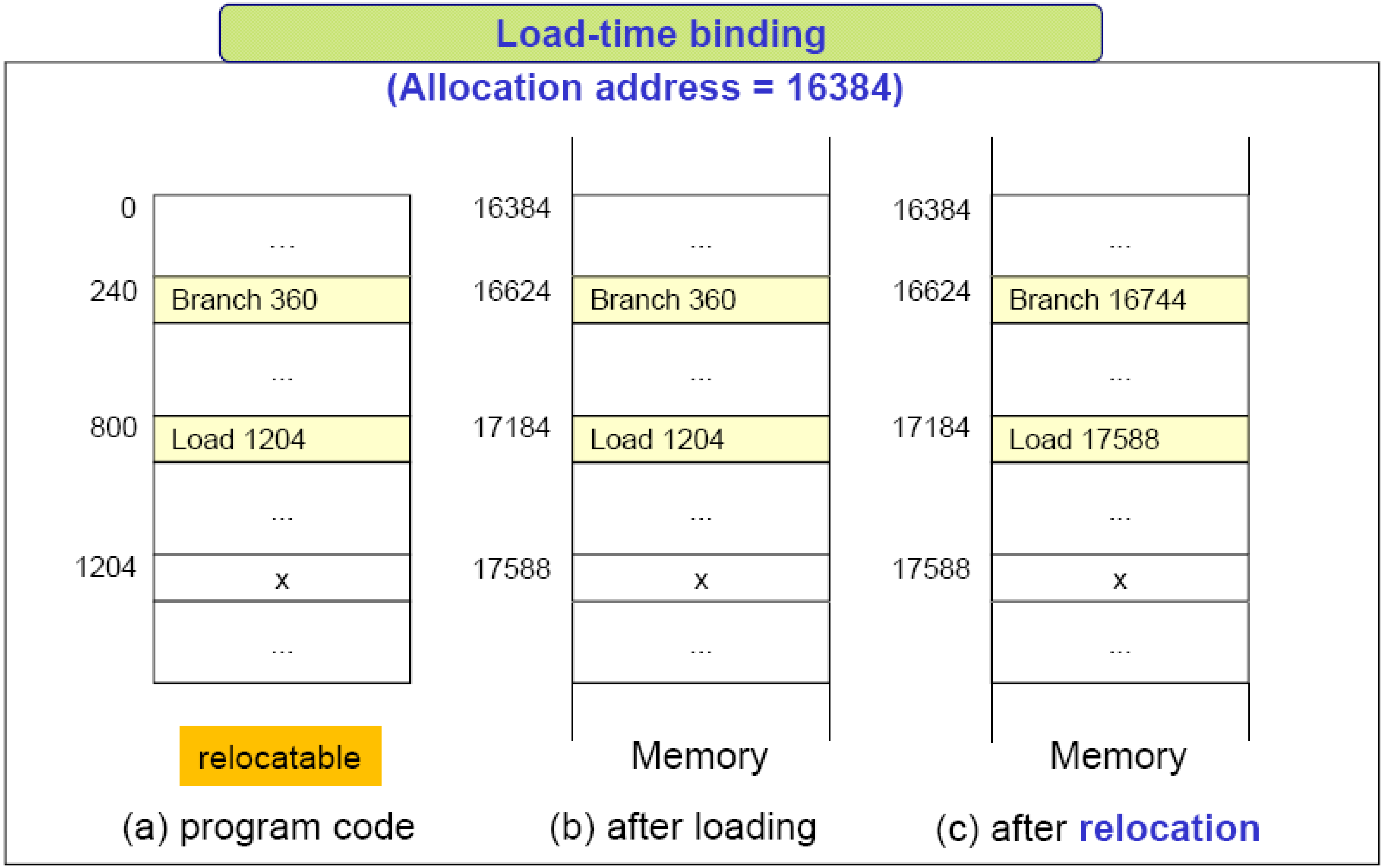
OS Requirements	Notes
Memory management	*Need to allocate memory for new processes;* *Reclaim memory from terminated processes;* *Generally manage memory via free list*
Base/bounds management	*Must set base/bounds properly upon context switch*
Exception handling	*Code to run when exceptions arise;* *likely action is to terminate offending process*

Static (Software-based) Relocation

- **Compile time binding**
  - When it is known at compile time where the process will reside in memory, then absolute code can be generated
  - Changing the starting location requires recompilation
  - MS-DOS .COM-format programs
- **Load time binding**
  - When it is not known at compile time where the process will reside in memory, then the compiler must generate relocatable code
  - Final binding is delayed until load time
  - Changing the starting location requires reloading or relocation of the user code
  - No protection
  - difficult to later relocate an address space to another location

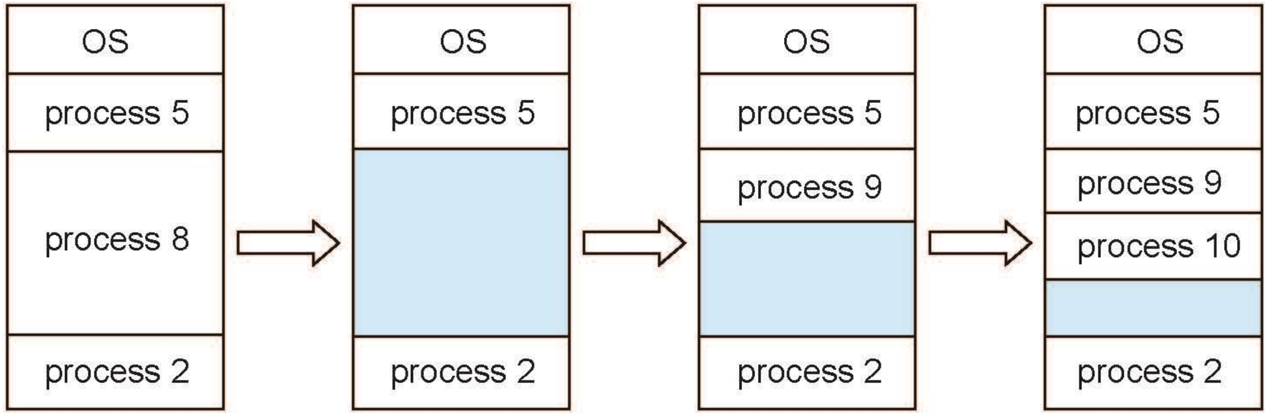


Address Binding



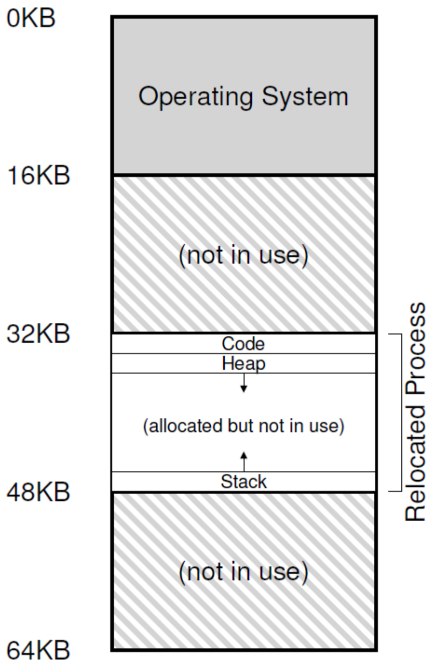
Contiguous Memory Allocation

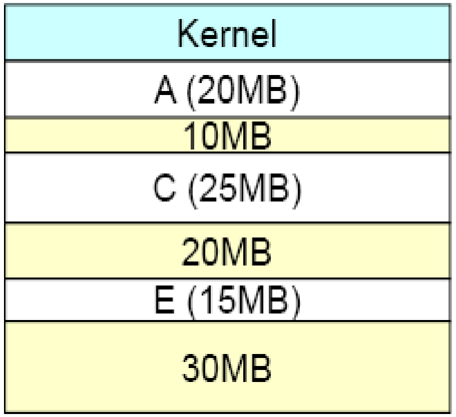
- Initially, all memory is available as a single large block of available memory (hole, partition)
- When a process arrives, it is allocated memory from a hole large enough to accommodate it
- Process exiting frees its partition, adjacent free partitions combined
- Memory partition state dynamically changes as a process enters (or exits) the system
- Contiguous allocation



Contiguous Memory Allocation

- **Fragmentation**
  - Internal fragmentation
    - When the process stack and heap are not too big, all of the space between the two is simply wasted
  - External fragmentation
    - Exists when enough total memory space exists to satisfy the request, but it is not contiguous





Contiguous Memory Allocation

- **Placement strategies**
  - First-fit
    - Start searching at the beginning of the state table
    - Allocate the first partition that is big enough
    - Simple and low overhead
  - Best-fit
    - Search the entire state table
    - Allocate the smallest partition that is big enough
    - Long search time
    - Can reserve large size partitions
    - May produce many small size partitions
    - External fragmentation

Contiguous Memory Allocation

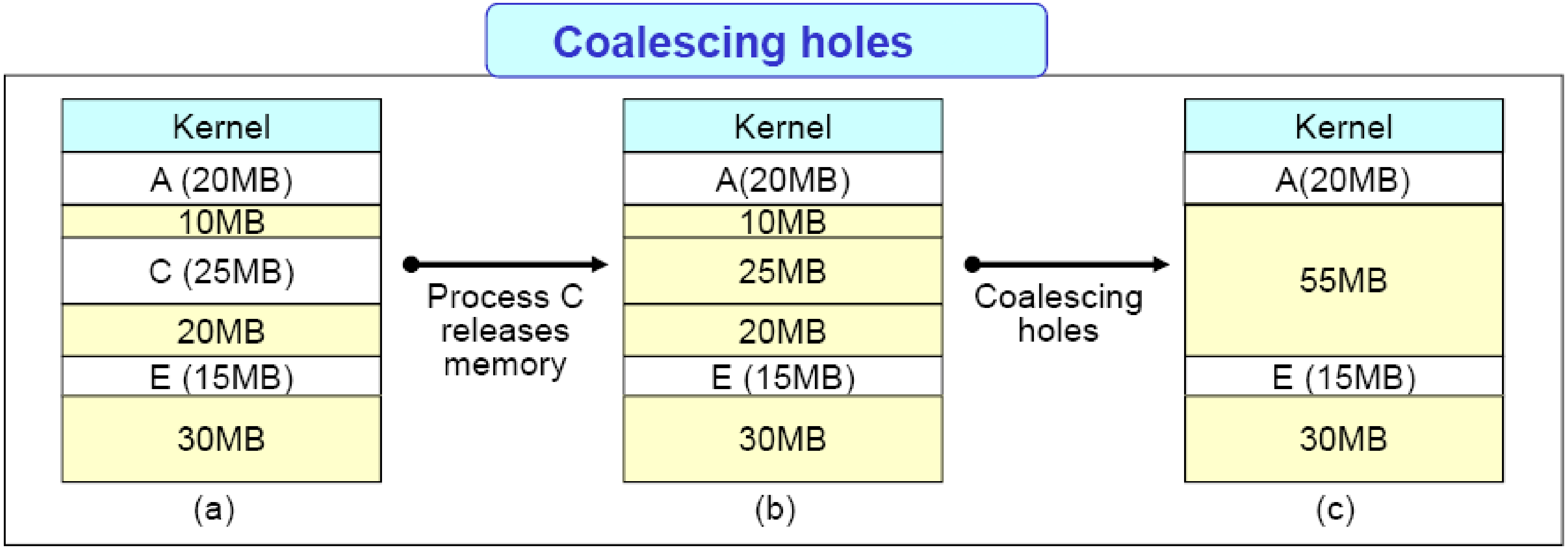
- **Placement strategies**
  - Worst-fit
    - Search the entire state table
    - Allocate the largest partition available
    - May reduce the number of small size partitions
  - Next-fit
    - Similar to first-fit method
    - Start searching from where the previous search ended
    - Circular search the state table
    - Uniform use for the memory partitions
    - Low overhead

Contiguous Memory Allocation

- **Coalescing holes**
  - Merge adjacent free partitions into one large partition

- **Compaction**
  - Shuffle the memory contents to place all free memory together in one large block (partition)
  - Done at execution time
    - Can be done only if relocation is dynamically possible
  - Consumes so much system resources
    - Consumes long CPU time

Contiguous Memory Allocation



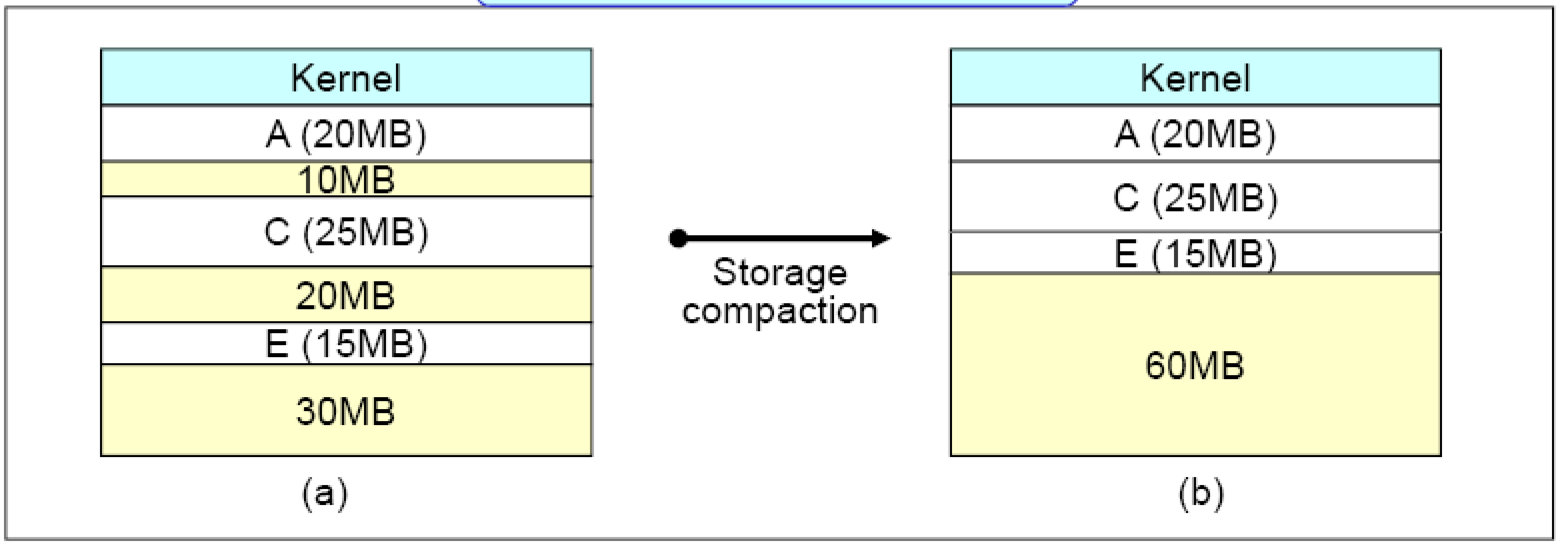
partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	C
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	none
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

partition	start address	size	current process ID
1	u	20	A
2	u+20	55	none
3	u+75	15	E
4	u+90	30	none

Contiguous Memory Allocation

Storage compaction



partition	start address	size	current process ID
1	u	20	A
2	u+20	10	none
3	u+30	25	C
4	u+55	20	none
5	u+75	15	E
6	u+90	30	none

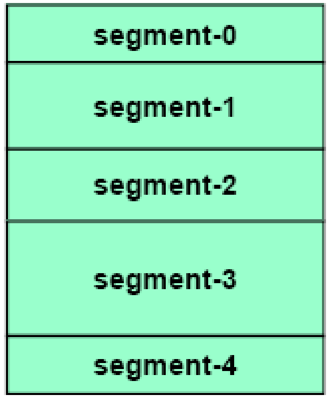
partition	start address	size	current process ID
1	u	20	A
2	u+20	25	C
3	u+45	15	E
4	u+60	60	none

Discontiguous Memory Allocation

- **Segmentation**
- **Paging**

Segmentation

- **Basic concept**
  - View memory as a collection of variable-sized segments
  - View program as a collection of various objects
    - Functions, methods, procedures, objects, arrays, stacks, variables, and so on
  - Logical address
    - v = (s, d)
      - s: segment number
      - d: offset (displacement)

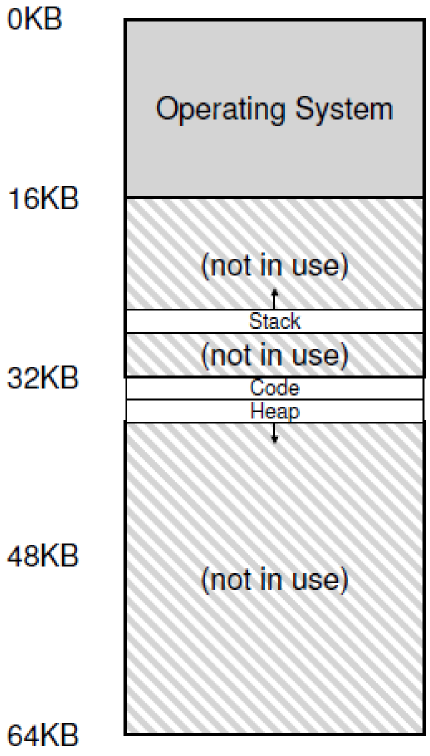


Segmentation

- **Basic concept**
  - Normally, when the user program is compiled, the compiler automatically constructs segments reflecting the input program
    - Code
    - Global variables
    - Heap
    - Stacks
    - Standard library

Segmentation

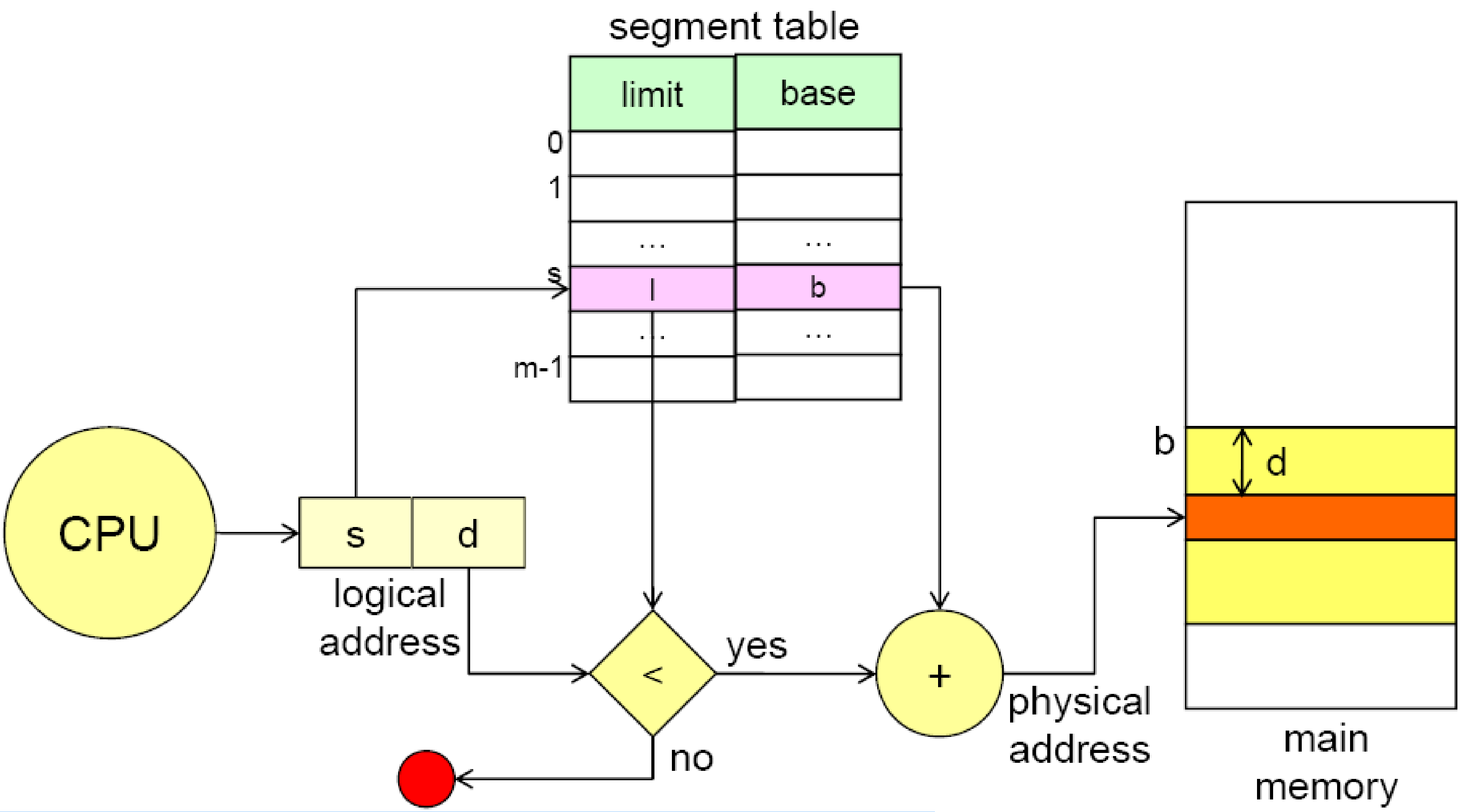
- All the unused space between the stack and the heap need not be allocated in physical memory,
- allowing us to fit more address spaces into physical memory.



Segment	Base	Size
Code	32K	2K
Heap	34K	2K
Stack	28K	2K

Segmentation

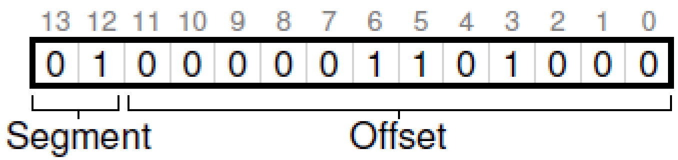
- **Address mapping**



segmentation violation or **segmentation fault**

Which Segment Are We Referring To?

- **Explicit** approach
  – the address space into segments is determined based on the top few bits of the virtual address



- **Implicit** approach
  – hardware determines the segment by noticing how the address was formed.
  – If the address was generated from the program counter (i.e., it was an instruction fetch), then the address is within the code segment
  – If the address is based off of the stack or base pointer, it must be in the stack segment
  – any other address must be in the heap.

What About The Stack?

- Stack grows backwards

Segment	Base	Size	Grows Positive?
Code	32K	2K	1
Heap	34K	2K	1
Stack	28K	2K	0

Stack range: 26KB~28KB

Support for Sharing

- Hardware also has to check whether a particular access is permissible

Segment	Base	Size	Grows Positive?	Protection
Code	32K	2K	1	Read-Execute
Heap	34K	2K	1	Read-Write
Stack	28K	2K	0	Read-Write

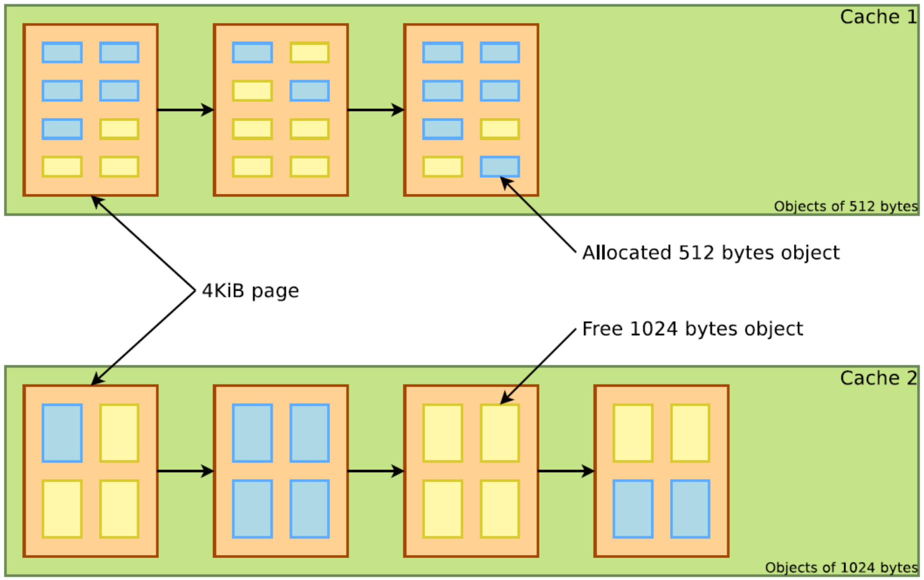
OS Support

- What should the OS do on a context switch?
  - the segment registers must be saved and restored.
  - each process has its own virtual address space, and the OS must make sure to set up these registers correctly before letting the process run again.

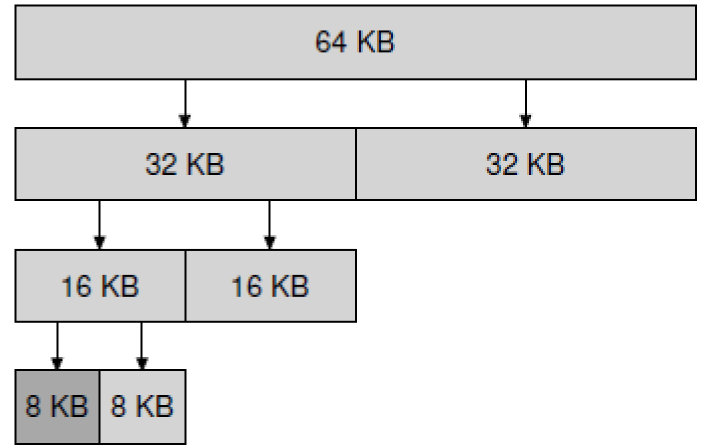
OS Support

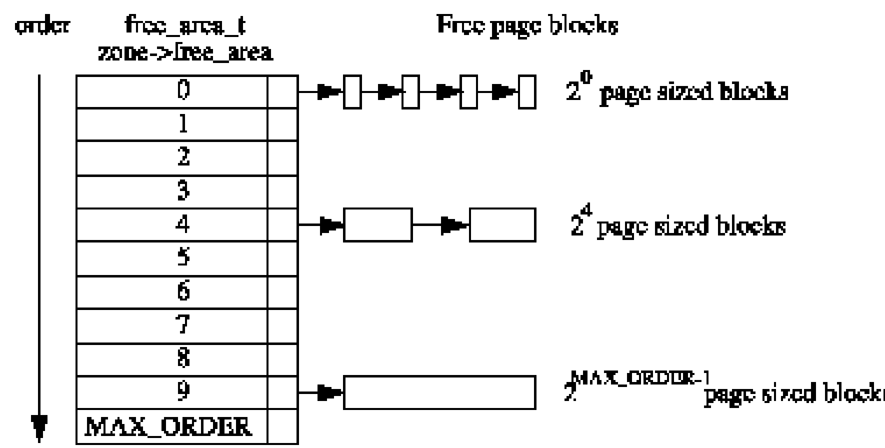
- Managing free space in physical memory
  - OS has to be able to find space in physical memory for its segments
  - best-fit, worst-fit, first-fit, … ➔ External fragmentation, need compaction
  - segregated list
    - manage objects of popular size; all other requests are forwarded to a more general memory allocator
    - No fragmentation, no complicated search
    - how much memory should one dedicate to the pool of memory that serves specialized requests of a given size ?
    - slab allocator: allocates a number of **object caches** for kernel objects that are likely to be requested frequently (such as locks, file-system inodes, etc.)
      - when a given cache is running low on free space, it requests some **slabs** of memory from a more general memory allocator
      - when the reference counts of the objects within a given slab all go to zero, the general allocator can reclaim them
  - Binary buddy allocator

Slab Allocator & Buddy Allocator




```
% cat /proc/slabinfo
slabinfo - version: 1.1
kmem_cache          60     78   100    2    2    1
blkdev_requests   5120   5120    96  128  128    1
mnt_cache           20     40    96    1    1    1
inode_cache       7005  14792   480 1598 1849    1
dentry_cache      5469   5880   128  183  196    1
filp               726    760    96   19   19    1
buffer_head      67131  71240    96 1776 1781    1
vm_area_struct    1204   1652    64   23   28    1
...
size-8192            1     17  8192    1   17    2
size-4096           41     73  4096   41   73    1
...
```





Homework

- Homework in Chap 13 (Debugging Memory Allocation w/ *valgrind*)