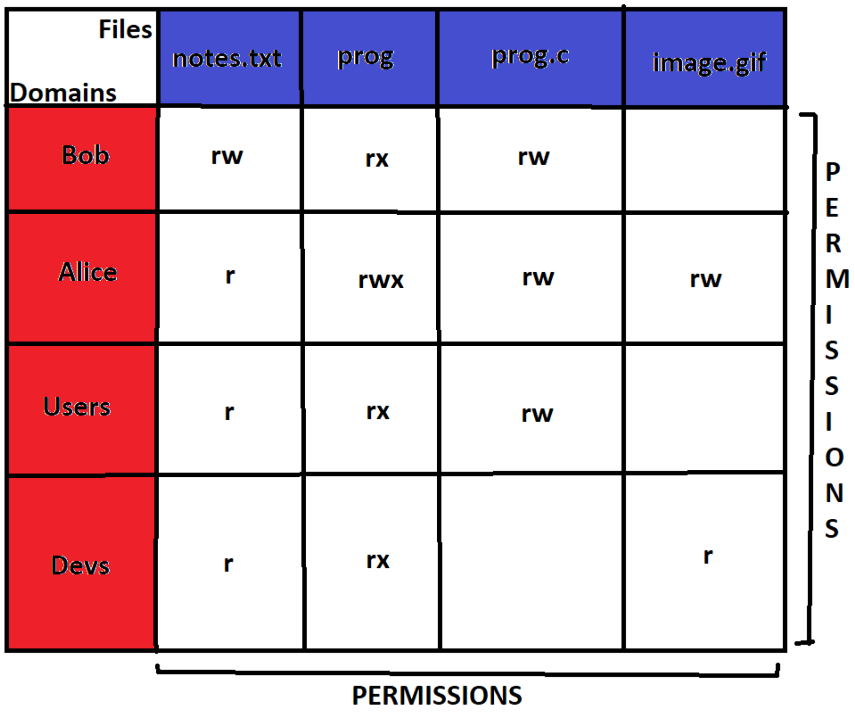
Stephen Woodbury - 1429496 - swoodbur
HW4 - CMPS 111 - Spring 2018 - 5/16/2018 - Wednesday

1) *For the following Unix directory listing, express the file permissions shown as (a) a protection matrix, (b) access control lists, and (c) capability lists. In this case, the user alice is in two groups, users and devs.*

-rw-r--r-- 2 bob users 4096 Apr 11 08:08 notes.txt -rwxr-xr-x 1 alice devs 3264 Apr 11 08:08 prog -rw-rw---- 1 alice users 128 Apr 11 08:08 prog.c -rw-r----- 1 alice devs 8122 Apr 11 08:08 image.gif

A: First, a couple notes. I kept track of permissions for our groups too. Permission labels: r=read, w=write, x=execute

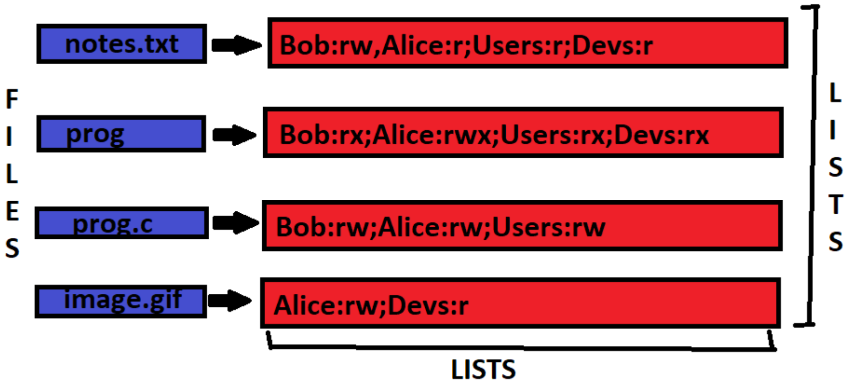
a) Here is our protection Matrix of file permissions:



Files Domains	notes.txt	prog	prog.c	image.gif
Bob	rw	rx	rw	
Alice	r	rwx	rw	rw
Users	r	rx	rw	
Devs	r	rx		r

PERMISSIONS
PERMISSIONS

b) Here are our access control lists of file permissions:



notes.txt → Bob:rw,Alice:r;Users:r;Devs:r

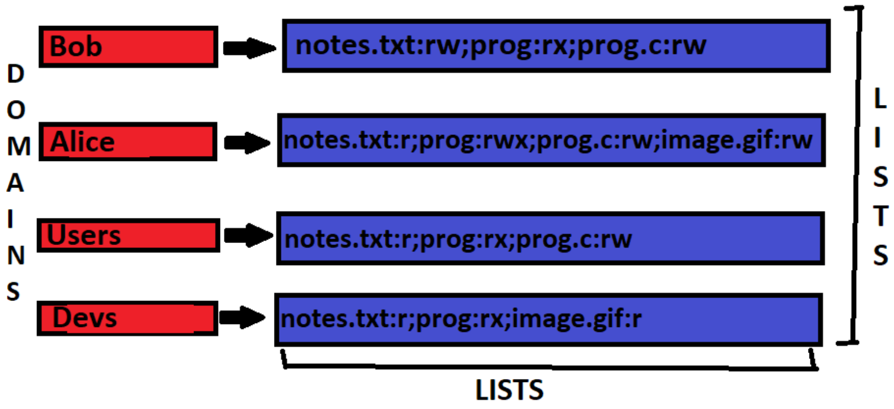
prog → Bob:rx;Alice:rwx;Users:rx;Devs:rx

prog.c → Bob:rw;Alice:rw;Users:rw

image.gif → Alice:rw;Devs:r

Stephen Woodbury - 1429496 - swoodbur
HW4 - CMPS 111 - Spring 2018 - 5/16/2018 - Wednesday

c) Here are our Capability Lists of File Permissions:


DOMAINS
Bob
notes.txt:rw;prog:rx;prog.c:rw
Alice
notes.txt:r;prog:rwx;prog.c:rw;image.gif:rw
Users
notes.txt:r;prog:rx;prog.c:rw
Devs
notes.txt:r;prog:rx;image.gif:r
LISTS
LISTS

Notes: control lists are lists belonging to each object/file that hold what domains have what permissions with said object. Capability lists are lists belonging to each Domain that hold what objects/files that domain has access to as well as what permissions it has with said object.

Stephen Woodbury - 1429496 - swoodbur
HW4 - CMPS 111 - Spring 2018 - 5/16/2018 - Wednesday

2) *Processor A has separate memory caches for data and executable code; processor B has a single cache where it stores both data and executable code. (a) Which processor will leave an operating system running on it more susceptible to a buffer overflow attack? Explain your answer. (b) List and describe two ways an operating system might try to protect itself from buffer overflow attacks regardless of the processor on which it runs.*

A:

a) Processor B is more susceptible to buffer overflow attacks as it's easier for hackers to run malicious code. Why is it easier? Because executable code can be injected into memory, even in places where there's supposed to be data (Because we aren't separating our executable and data memory). If we did have our memory separated as in Processor A, it's easier to lock access to executable code. Hackers would have to usually find already implemented executable code to stitch together to get what they want done, which is much harder than writing the executable code from scratch somewhere in memory meant for data.

b) Two ways in which an Operating system might protect itself from buffer overflow attacks:

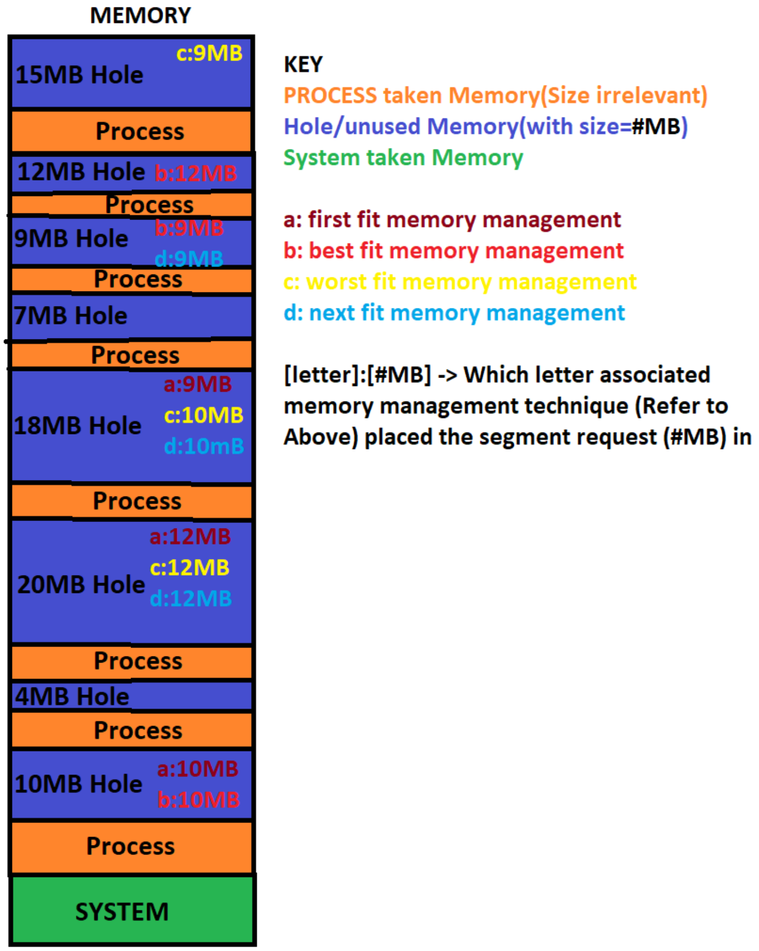
I) Stack Canaries: Placing a random number 'canary' just below a function's return address (In the memory stack). Place code after last instruction to run in a function to check the 'canary'. If the number has changed, something went awry, it may be that someone tried overflowing into the return address to change it to reference malicious code(In doing so, they overrode the canary's data/random number).

II) Address Randomization: Given our memory is separable, hackers may try and use already written executable code to create a frankenstein like program that does malicious things. Address Randomization randomizes addresses of functions and data every time a program runs. This prevents hackers from finding/referencing these already written bits of executable code.

Stephen Woodbury - 1429496 - swoodbur
HW4 - CMPS 111 - Spring 2018 - 5/16/2018 - Wednesday

3) *Consider a simple, non-paged memory management system in which memory consists of the following hole sizes in memory order: 10 MB, 4 MB, 20 MB, 18 MB, 7 MB, 9 MB, 12 MB, and 15 MB. Which hole is taken for successive segment requests of 12 MB, 10 MB, and 9 MB for (a) first fit (b) best fit (c) worst fit (d) next fit? Show all your work.*

A: Reference below picture for answer explained below:


MEMORY
c:9MB
15MB Hole
Process
12MB Hole b:12MB
Process
9MB Hole b:9MB
d:9MB
Process
7MB Hole
Process
a:9MB
18MB Hole c:10MB
d:10mB
Process
a:12MB
20MB Hole c:12MB
d:12MB
Process
4MB Hole
Process
10MB Hole a:10MB
b:10MB
Process
SYSTEM
KEY
PROCESS taken Memory(Size irrelevant)
Hole/unused Memory(with size=#MB)
System taken Memory
a: first fit memory management
b: best fit memory management
c: worst fit memory management
d: next fit memory management
[letter]:[#MB] -> Which letter associated memory management technique (Refer to Above) placed the segment request (#MB) in

a) First Fit Memory Management finds the first hole that fits a segment request. Every subsequent call/search done by First Fit starts from the start of memory range(Memory order or bottom->Top). With this we find that the 12MB segment request is placed in the 20MB hole, the 10MB segment request is placed in the 10MB hole, and the 9MB segment request is placed in the 18MB hole.

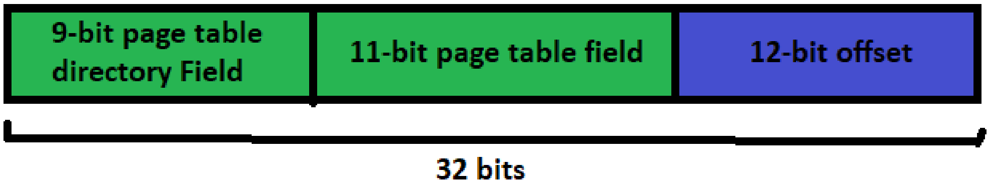
b) Best Fit Memory Management finds the smallest hole that can hold our segment request. With this we find the 12MB segment request goes into the 12MB hole, the 10MB segment request goes into the 10MB hole, and the 9MB segment request goes into the 9MB hole.

c) Worst Fit Memory Management finds the biggest hole that the segment request can go into. With this we find the 12MB segment request goes into the 20MB hole, the 10MB segment request goes into the 18MB hole, and the 9MB segment request goes into the 15MB hole.

d) Next Fit Memory Management works like First Fit. The only difference is that instead of starting from the beginning of the memory range every time we search for a hole, we start where we left off with the last search. With this, we find that the 12MB segment request goes into the 20MB hole, the 10MB segment request goes into the 18MB hole, and the 9MB segment request goes into the 9MB hole.

4) *A computer with a 32-bit processor uses a two-level page lookup mechanism consisting of page tables and a page table directory. Virtual Addresses consist of a 9-bit page table directory field, an 11-bit page table field, and an offset. How large are the pages and how many are there in the address space? Show your work.*

A:

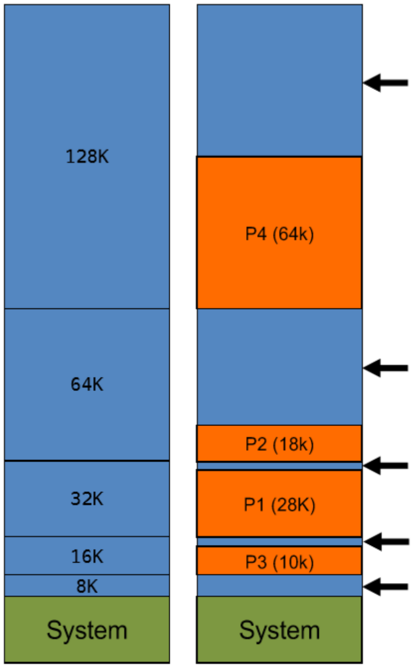


We calculated our offset as such:
32 bit address - 9-bit page table directory field - 11-bit page table field = 12-bit offset.
Our page size is 2^12 (as our offset is 12 bits) which is: 4KB
Since there are 20 bits dedicated to defining virtual pages, the total number of virtual pages we can have is 2^20 pages.
That's 1048576 pages.
Page Size: 4KB; Page Count: 1,048,576

5) *(a) Explain the difference between the terms internal fragmentation and external fragmentation as they relate to memory allocation. (b) Briefly outline the concept of paging. (c) Explain why paging is considered a better memory management technique than compaction. In all cases, draw diagrams if you feel this will make your answers clearer.*

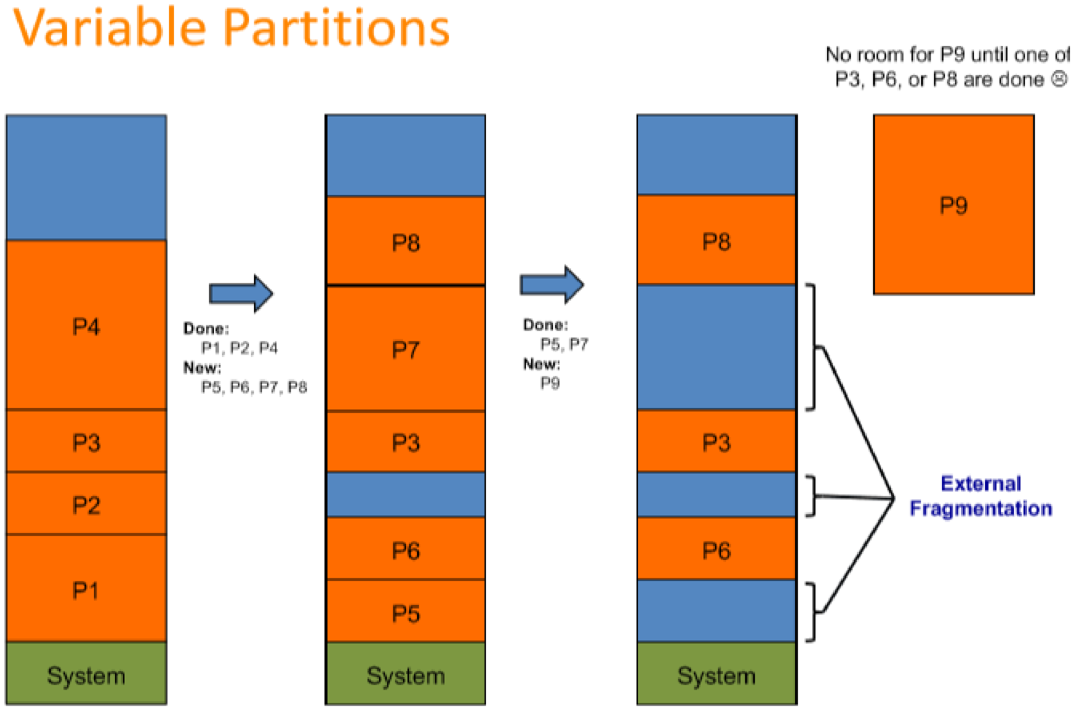
A:

a) Internal fragmentation is referencing how there is unused memory in supposedly 'occupied' partitions of memory in a fixed partitioned memory system. Refer to the picture below, taken from notes:



On the left, we've our partitioned memory system prior to process allocation. Only one process can occupy a partition. On the right, we've our process occupied memory system. The orange blocks represent the process's and their associated memory they need. The blue blocks are the leftover bits of our partitions that aren't being used by our processes (despite our processes occupying the partitions).

External fragmentation is referencing how there can be leftover memory that is unused in a variably partitioned memory system. Reference the picture below, from notes:



On the left, we see our variable partitioned memory system being occupied by processes. The partitions are made to fit around our processes so that initially, there are no holes between our processes. In the middle, we see that Process 1, 2, and 4 have completed while process 3 is still running. Processes 5,6,7, and 8 are new, and are given memory wherever they can fit. Processes 5 and 6 are placed before Process 3. Process 7 is placed after process 3 as the gap between process 6 and process 3 isn't enough for process 7. Likewise with process 8. On the right, we see that processes 5 and 7 have finished up. We also see that process 9 wants to run. Though there exists enough unused memory in the system to run Process 9, it's not continuous, ie, process 9 must wait for enough continuous memory to be freed up.

b) Paging is used as a memory management technique that allows processes to be stored in non-continuous memory. It also allows for each process to reference its own supposedly large real continuous address space (In reality, it's an illusion) even if the actual physical memory is small. The way it works is that it separates our theoretical/virtual address space and our actual/physical address space into equally sized pages. These pages are whats allocated to processes to use. Those pages that are active/being used by a process/are on the CPU, are mapped to a physical page. The result is that we can reference/use more memory than we actually have.

c) Paging is better than compaction in that compaction forces processes to wait for compacting to happen before the process can run. In paging, the process doesn't need one large continuous physical address space. This means compaction doesn't need to happen and the process can run immediately.