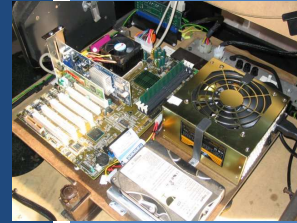


TCSS 422: OPERATING SYSTEMS

Memory Virtualization, Segmentation, Memory Paging



Wes J. Lloyd
School of Engineering and Technology,
University of Washington - Tacoma

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

FEEDBACK FROM 11/19

- **Assignment #2:**
 - Can we have another example of optimizing TLP by moving locks around?
- **Assignment #3:**
 - What is the purpose of a Linux “proc” file?
- **Memory Virtualization:**
 - When would you need to use `brk()`, `sbrk()`?
 - **Legacy:** *The `brk()` and `sbrk()` functions are historical curiosities left over from earlier days before virtual memory management.*
 - Called internally by `malloc()`, `realloc()`, to adjust heap location

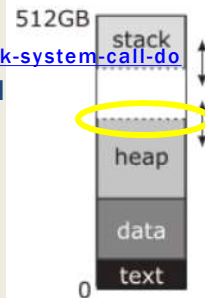
November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.2

BRK(), SBRK()

- From [stackoverflow.com](https://stackoverflow.com/questions/6988487/what-does-the-brk-system-call-do):
- <https://stackoverflow.com/questions/6988487/what-does-the-brk-system-call-do>
- The "break", the address manipulated by `brk()` and `sbrk()`, is the dotted line at the top of the heap
- In traditional Unix (before shared libraries) the data segment was continuous with the heap.
- Before the program starts, the kernel would load the "text" and "data" blocks into RAM starting at address zero and set the break address to the end of the data segment.
- The first call to `malloc()` would use `sbrk()` to move the break up and create the heap in between the top of the data segment and the new, higher break address, as shown in the diagram, and subsequent use of `malloc()` would use it to make the heap bigger as necessary.



November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.3

FEEDBACK - 2

- Hard to track the details relevant for final exam
- All is of course important, but my notes weren't substantial enough for the midterm

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.4

OBJECTIVES

- Quiz 4
- Program 2
- Program 3
- Segments
- Chapter 17 – Free Space Management
- Paging
- Chapter 18 – Introduction to Paging
- Chapter 19 – Translation Lookaside Buffer
- Chapter 20 – Paging Smaller Tables
- Chapter 21/22 – Beyond Physical Memory

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.5



CHAPTER 17: FREE SPACE MANAGEMENT

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.6

FREE SPACE MANAGEMENT

- Management of memory using

- Only fixed-sized units
 - Easy: keep a list
 - Memory request → return first free entry
 - Simple search

- With variable sized units
 - More challenging
 - Results from variable sized malloc requests
 - Leads to fragmentation

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.7
-------------------	---	-------

FRAGMENTATION

- Consider a 30-byte heap

30-byte heap:

free	used	free
------	------	------

0 10 20 30

- Request for 15-bytes

free list: head →

addr:0 len:10

 →

addr:20 len:10

 → NULL

- Free space: 20 bytes

- No available contiguous chunk → return NULL

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.8
-------------------	---	-------

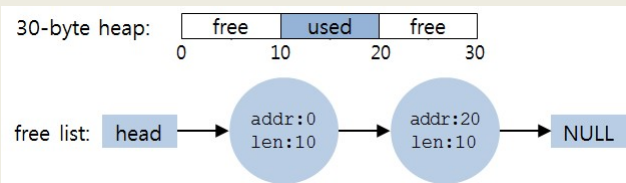
FRAGMENTATION - 2

- **External:** OS can compact
 - Example: Client asks for 100 bytes: malloc(100)
 - OS: No 100 byte contiguous chunk is available: returns NULL
 - Memory is externally fragmented - - Compaction can fix!
- **Internal:** lost space - OS can't compact
 - OS returns memory units that are too large
 - Example: Client asks for 100 bytes: malloc(100)
 - OS: Returns 125 byte chunk
 - Fragmentation is *in* the allocated chunk
 - Memory is lost, and unaccounted for - can't compact

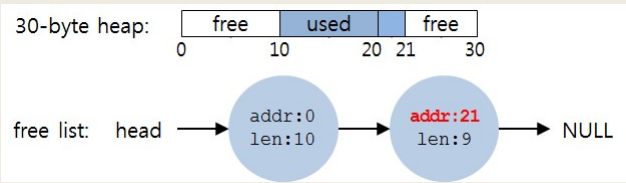
November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.9
-------------------	---	-------

ALLOCATION STRATEGY: SPLITTING

- Request for 1 byte of memory: malloc(1)



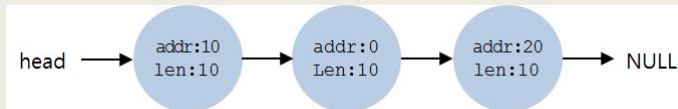
- OS locates a free chunk to satisfy request
- Splits chunk into two, returns first chunk



November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.10
-------------------	---	--------

ALLOCATION STRATEGY: COALESCING

- Consider 30-byte heap
- Free() frees all 10 bytes segments (*list of 3-free 10-byte chunks*)



- Request arrives: malloc(30)
- ***SPLIT DOES NOT WORK*** - no contiguous 30-byte chunk exists!
- Coalescing regroups chunks into contiguous chunk



- Allocation can now proceed
- Coalescing is defragmentation of the free space list

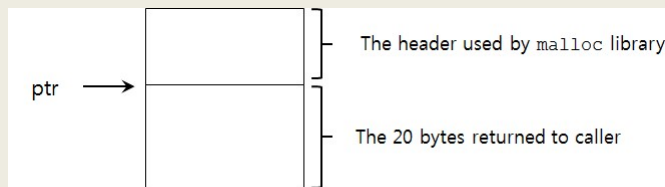
November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.11

MEMORY HEADERS

- free(void *ptr): Does not require a size parameter
- *How does the OS know how much memory to free?*
- Header block
 - Small descriptive block of memory at start of chunk



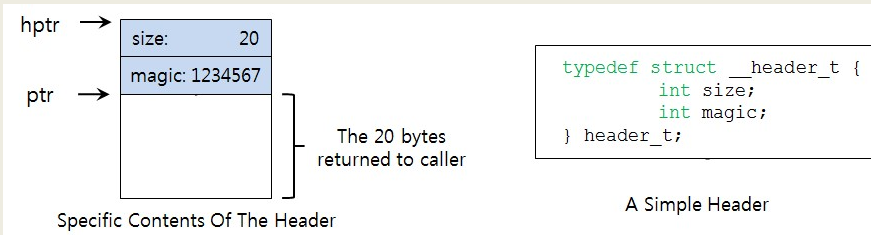
An Allocated Region Plus Header

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.12

MEMORY HEADERS - 2



- Contains size
- Pointers: for faster memory access
- Magic number: integrity checking

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.13

MEMORY HEADERS - 3

- Size of memory chunk is:
 - Header size + user malloc size
 - N bytes + sizeof(header)
- Easy to determine address of header

```
void free(void *ptr) {  
    header_t *hptr = (void *)ptr - sizeof(header_t);  
}
```

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.14

THE FREE LIST

- Simple free list struct

```
typedef struct __node_t {  
    int size;  
    struct __node_t *next;  
} node_t;
```

- Use mmap to create free list
- 4kb heap, 4 byte header, one contiguous free chunk

```
// mmap() returns a pointer to a chunk of free space  
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,  
                    MAP_ANON|MAP_PRIVATE, -1, 0);  
head->size = 4096 - sizeof(node_t);  
head->next = NULL;
```

November 20, 2018	TCCS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.15
-------------------	---	--------

FREE LIST - 2

- Create and initialize free-list “heap”

```
// mmap() returns a pointer to a chunk of free space  
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,  
                    MAP_ANON|MAP_PRIVATE, -1, 0);  
head->size = 4096 - sizeof(node_t);  
head->next = NULL;
```

- Heap layout:

head →

size:	4088	[virtual address: 16KB] header: size field
next:	0	header: next field(NULL is 0)
...		the rest of the 4KB chunk

November 20, 2018	TCCS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.16
-------------------	---	--------

FREE LIST: MALLOC() CALL

- Consider a request for a 100 bytes: `malloc(100)`
- Header block requires 8 bytes
 - 4 bytes for size, 4 bytes for magic number
- Split the heap - header goes with each block

A 4KB Heap With One Free Chunk

head →

size:	4088
next:	0
...	

the rest of the 4KB chunk

A Heap : After One Allocation

ptr →

size:	100
magic:	1234567
First block is used	

the 100 bytes now allocated

head →

size:	3980
next:	0
...	

the free 3980 byte chunk

November 20, 2018
TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma
L15.17

FREE LIST: FREE() CALL

- Addresses of chunks
- Start=16384
 - + 108 (end of 1st chunk)
 - + 108 (end of 2nd chunk)
 - + 108 (end of 3rd chunk)
 - = 16708

8 bytes header

sptr →

head →

size:	100	[virtual address: 16KB]
magic:	1234567	
...		
size:	100	100 bytes still allocated
magic:	1234567	
Free this block		
size:	100	100 bytes still allocated (but about to be freed)
magic:	1234567	
...		
size:	3764	100 bytes still allocated
next:	0	
...		

The free 3764-byte chunk

Free Space With Three Chunks Allocated

November 20, 2018
TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma
L15.18

FREE LIST: FREE() CHUNK #2

- Free(sptr)
- Our 3 chunks start at 16 KB (@ 16,384 bytes)
- Free chunk #2 - sptr
- Sptr = 16500
 - addr - sizeof(node_t)
- Actual start of chunk #2
 - 16492

November 20, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.19

FREE LIST- FREE ALL CHUNKS

- Now free remaining chunks:
- Free(16392)
- Free(16608)
- Walk back 8 bytes for actual start of chunk
- External fragmentation
- Free chunk pointers out of order
- Coalescing of next pointers is needed

November 20, 2018

TCCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.20

GROWING THE HEAP

- Start with small sized heap
- Request more memory when full
- `sbrk()`, `brk()`

Segmented heap

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.21
-------------------	---	--------

MEMORY ALLOCATION STRATEGIES

- **Best fit**
 - Traverse free list
 - Identify all candidate free chunks
 - Note which is smallest (has best fit)
 - When splitting, “leftover” pieces are small (and potentially less useful -- fragmented)

- **Worst fit**
 - Traverse free list
 - Identify largest free chunk
 - Split largest free chunk, leaving a still large free chunk

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.21
-------------------	---	--------

EXAMPLES

- Allocation request for 15 bytes



- Result of Best Fit



- Result of Worst Fit



November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.23

MEMORY ALLOCATION STRATEGIES - 2

- First fit

- Start search at beginning of free list
- Find first chunk large enough for request
- Split chunk, returning a “fit” chunk, saving the remainder
- Avoids full free list traversal of best and worst fit

- Next fit

- Similar to first fit, but start search at last search location
- Maintain a pointer that “cycles” through the list
- Helps balance chunk distribution vs. first fit
- Find first chunk, that is large enough for the request, and split
- Avoids full free list traversal

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.24

SEGREGATED LISTS

- For popular sized requests
e.g. for kernel objects such as locks, inodes, etc.
- Manage as segregated free lists
- Provide object caches: stores pre-initialized objects

- How much memory should be dedicated for specialized requests (object caches)?

- If a given cache is low in memory, can request “*slabs*” of memory from the general allocator for caches.
- General allocator will reclaim slabs when not used

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.25
-------------------	---	--------

BUDDY ALLOCATION

- Binary buddy allocation
 - Divides free space by two to find a block that is big enough to accommodate the request; the next split is too small...
- Consider a 7KB request

64KB free space for 7KB request


November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.26
-------------------	---	--------

BUDDY ALLOCATION - 2

- Buddy allocation: suffers from internal fragmentation
- Allocated fragments, typically too large
- Coalescing is simple
 - Two adjacent blocks are promoted up

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.27
-------------------	---	--------

CHAPTER 18: INTRODUCTION TO PAGING



November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.28
-------------------	---	--------

PAGING

- Split up address space of process into *fixed sized pieces* called **pages**
- Alternative to *variable sized pieces* (Segmentation) which suffers from significant fragmentation
- Physical memory is split up into an array of fixed-size slots called **page frames**.
- Each process has a **page table** which translates virtual addresses to physical addresses

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.29

ADVANTAGES OF PAGING

- **Flexibility**
 - Abstracts the process address space into pages
 - No need to track direction of HEAP / STACK growth
 - *Just add more pages...*
 - No need to store unused space
 - *As with segments...*
- **Simplicity**
 - Pages and page frames are the same size
 - Easy to allocate and keep a free list of pages

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.30

PAGING: EXAMPLE

Page Table:
 VP0 → PF3
 VP1 → PF7
 VP2 → PF5
 VP3 → PF2

- Consider a 128 byte address space with 16-byte pages
- Consider a 64-byte program address space

A Simple 64-byte Address Space

0		(page 0 of the address space)
16		(page 1)
32		(page 2)
48		(page 3)
64		

64-Byte Address Space Placed In Physical Memory

0	reserved for OS	page frame 0 of physical memory
16	(unused)	page frame 1
32	page 3 of AS	page frame 2
48	page 0 of AS	page frame 3
64	(unused)	page frame 4
80	page 2 of AS	page frame 5
96	(unused)	page frame 6
112	page 1 of AS	page frame 7
128		

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
 School of Engineering and Technology, University of Washington - Tacoma

L15.31

PAGING: ADDRESS TRANSLATION

- **PAGE:** Has two address components
 - **VPN:** Virtual Page Number
 - **Offset:** Offset within a Page

VPN		offset			
Va5	Va4	Va3	Va2	Va1	Va0

- **Example:**
 Page Size: 16-bytes, Address Space: 64-bytes

VPN		offset			
0	1	0	1	0	1

Here there are just four pages...

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
 School of Engineering and Technology, University of Washington - Tacoma

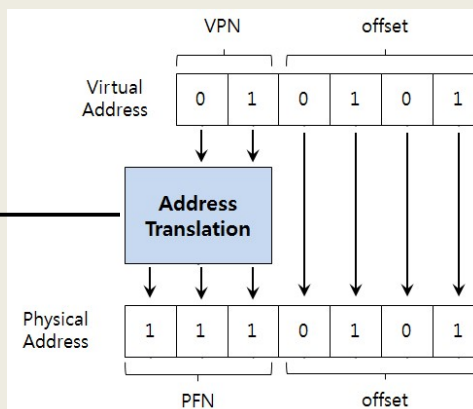
L15.32

EXAMPLE: PAGING ADDRESS TRANSLATION

- Consider a 64-byte program address space (4 pages)
- Stored in 128-byte physical memory (8 frames)
- Offset is preserved
- VPN is looked up

Page Table:

VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2



November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.33

PAGING DESIGN QUESTIONS

- (1) Where are page tables stored?
- (2) What are the typical contents of the page table?
- (3) How big are page tables?
- (4) Does paging make the system too slow?

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.34

(1) WHERE ARE PAGE TABLES STORED?

- Example:
 - Consider a 32-bit process address space (up to 4GB)
 - With 4 KB pages
 - 20 bits for VPN (2^{20} pages)
 - 12 bits for the page offset (2^{12} unique bytes in a page)
- Page tables for each process are stored in RAM
 - Support potential storage of 2^{20} translations = 1,048,576 pages per process
 - Each page has a page table entry size of 4 bytes

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.35

PAGE TABLE EXAMPLE

- With 2^{20} slots in our page table for a single process
- Each slot dereferences a VPN
- Provides physical frame number
- Each slot requires 4 bytes (32 bits)
 - 20 for the PFN on a 4GB system with 4KB pages
 - 12 for the offset which is preserved
 - (note we have no status bits, so this is unrealistically small)
- How much memory to store page table for 1 process?
 - 4,194,304 bytes (or 4MB) to index one process

VPN ₀
VPN ₁
VPN ₂
...
...
VPN ₁₀₄₈₅₇₆

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.36

NOW FOR AN ENTIRE OS

- If 4 MB is required to store one process
- Consider how much memory is required for an entire OS?
 - With for example 100 processes...
- Page table memory requirement is now $4\text{MB} \times 100 = 400\text{MB}$
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

$400\text{ MB} / 4000\text{ GB}$

- Is this efficient?

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.37
-------------------	---	--------

(2) WHAT'S ACTUALLY IN THE PAGE TABLE

- Page table is data structure used to map virtual page numbers (VPN) to the physical address (Physical Frame Number PFN)
 - Linear page table → simple array
- Page-table entry
 - 32 bits for capturing state

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN												G	PAT	D	A	PCD	PWT	U/S	R/W	P											

An x86 Page Table Entry(PTE)

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.38
-------------------	---	--------

PAGE TABLE ENTRY

- **P: present**
- **R/W: read/write bit**
- **U/S: supervisor**
- **A: accessed bit**
- **D: dirty bit**
- **PFN: the page frame number**

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PFN																						G	PAT	D	A	PCD	PWT	U/S	R/W	P	

An x86 Page Table Entry(PTE)

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.39
-------------------	---	--------

PAGE TABLE ENTRY - 2

- **Common flags:**
 - **Valid Bit:** Indicating whether the particular translation is valid.
 - **Protection Bit:** Indicating whether the page could be read from, written to, or executed from
 - **Present Bit:** Indicating whether this page is in physical memory or on disk(swapped out)
 - **Dirty Bit:** Indicating whether the page has been modified since it was brought into memory
 - **Reference Bit(Accessed Bit):** Indicating that a page has been accessed

November 20, 2018	TCSS422: Operating Systems [Fall 2018] School of Engineering and Technology, University of Washington - Tacoma	L15.40
-------------------	---	--------

(3) HOW BIG ARE PAGE TABLES?

- Page tables are too big to store on the CPU
- Page tables are stored using physical memory
- Paging supports efficiently storing a sparsely populated address space
 - Reduced memory requirement
Compared to base and bounds, and segments

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.41

(4) DOES PAGING MAKE THE SYSTEM TOO SLOW?

- Translation
- **Issue #1:** Starting location of the page table is needed
 - HW Support: Page-table base register
 - stores active process
 - Facilitates translation
- **Issue #2:** Each memory address translation for paging requires an extra memory reference
 - HW Support: TLBs (Chapter 19)

Page Table:

VP0 → PF3
VP1 → PF7
VP2 → PF5
VP3 → PF2

Stored in RAM →

November 20, 2018

TCSS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.42

PAGING MEMORY ACCESS

```
1. // Extract the VPN from the virtual address
2. VPN = (VirtualAddress & VPN_MASK) >> SHIFT
3.
4. // Form the address of the page-table entry (PTE)
5. PTEAddr = PTBR + (VPN * sizeof(PTE))
6.
7. // Fetch the PTE
8. PTE = AccessMemory(PTEAddr)
9.
10. // Check if process can access the page
11. if (PTE.Valid == False)
12.     RaiseException(SEGMENTATION_FAULT)
13. else if (CanAccess(PTE.ProtectBits) == False)
14.     RaiseException(PROTECTION_FAULT)
15. else
16.     // Access is OK: form physical address and fetch it
17.     offset = VirtualAddress & OFFSET_MASK
18.     PhysAddr = (PTE.PFN << PFN_SHIFT) | offset
19.     Register = AccessMemory(PhysAddr)
```

November 20, 2018

TCS422: Operating Systems [Fall 2018]
School of Engineering and Technology, University of Washington - Tacoma

L15.43

QUESTIONS

