

TCSS 422: OPERATING SYSTEMS

Free Space Management,  
Introduction to Paging,  
Translation Lookaside Buffer



Wes J. Lloyd  
Institute of Technology  
University of Washington - Tacoma

February 26, 2018

TCSS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

OBJECTIVES

- Ungraded Quiz 3– Synchronized Array
- Homework 2 Questions
- Homework 3 Questions
- Ch. 17
  - Free Space Management
- Ch. 18
  - Introduction to Paging
- Ch. 19
  - Translation Lookaside Buffer (TLB)

February 26, 2018

TCSS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.2

FEEDBACK FROM 2/21

- Is this how my output will be tested (by running 'cat /proc/proc\_report' command)? I'm able to get the output printed to the proc file.

```
PROCESS REPORTER
Runnable:9
Runnable:4
Stopped:174
Process ID=1 Name=systd number_of_children=29 first_child_pid=234 first_child_name: systd-journal
Process ID=2 Name=kworker/0/0 *No Children
Process ID=3 Name=kworker/0/1 *No Children
Process ID=4 Name=kworker/0/2 *No Children
Process ID=5 Name=kworker/0/3 *No Children
Process ID=6 Name=kworker/0/4 *No Children
Process ID=7 Name=kworker/0/5 *No Children
Process ID=8 Name=kworker/0/6 *No Children
Process ID=9 Name=kworker/0/7 *No Children
Process ID=10 Name=kworker/0/8 *No Children
Process ID=11 Name=kworker/0/9 *No Children
Process ID=12 Name=kworker/0/10 *No Children
Process ID=13 Name=kworker/0/11 *No Children
Process ID=14 Name=kworker/0/12 *No Children
Process ID=15 Name=kworker/0/13 *No Children
Process ID=16 Name=kworker/0/14 *No Children
Process ID=17 Name=kworker/0/15 *No Children
Process ID=18 Name=kworker/0/16 *No Children
Process ID=19 Name=kworker/0/17 *No Children
Process ID=20 Name=kworker/0/18 *No Children
Process ID=21 Name=kworker/0/19 *No Children
Process ID=22 Name=kworker/0/20 *No Children
```

- YES

February 26, 2018

TCSS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.3

FEEDBACK - 2

- 2. Do I also need to print output to kernel log messages? (when I type 'dmesg') or only print the proc file?
- NO, this is optional... The kernel log output is only used to grade the program if the procfile is not implemented.
- 3. I notice process ID skips numbers, is that okay? See 'Process ID =' below.

```
Process ID=896 Name=VBoxService *No children
Process ID=921 Name=Xorg *No Children
Process ID=925 Name=dhclient *No Children
Process ID=938 Name=dnsnag *No Children
Process ID=1089 Name=lightdm number_of_children=1 first_child_pid=1098 first_child_name: upstart
Process ID=1094 Name=systd number_of_children=1 first_child_pid=1095 first_child_name: (sd-pam)
```

- YES, the process that formerly had the PID has likely terminated.

February 26, 2018

TCSS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.4

FEEDBACK - 3

- I am confused about trying to get this assignment going. I understand the importance of process tree traversal being separate from output being placed in proc file, however I don't know how to go about completely separating these two.
- I am thinking I need to make a method that traverses the process tree, and I store that information in some sort of data structure?
- This is a good approach. Store the information in a data structure so it can be accessed later. Note that malloc is called kcalloc for kernel module programming.
- Or make a temporary output file?
- This is a less optimal solution.

February 26, 2018

TCSS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.5

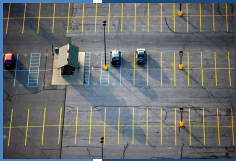
FEEDBACK - 4

- The assignment description says: "Report generation and process list computation cannot occur in the /proc output routines."
- I am confused as to what this report generation is supposed to look like it's not the /proc output.
- Points are deducted if report generation is done in the proc file output routine. (e.g. the event handler that is called when someone tries to read /proc/proc\_report.)
- Page 2 of Assignment 2 shows example output of the proc file.

February 26, 2018

TCSS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.6



# CHAPTER 17: FREE SPACE MANAGEMENT

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.7

## 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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.8

## 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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.9

## 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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.10

## ALLOCATION STRATEGY: SPLITTING

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

30-byte heap:

free

used

free

0

10

20

30

free list: head →

addr:0  
len:10

addr:20  
len:10

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

30-byte heap:

free

used

free

0

10

21

30

free list: head →

addr:0  
len:10

addr:21  
len:9

→ NULL

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.11

## ALLOCATION STRATEGY: COALESCING

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

head →

addr:0  
len:10

addr:10  
len:10

addr:20  
len:10

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

head →

addr:0  
len:30

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

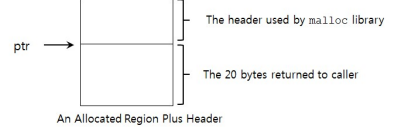
February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.12

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



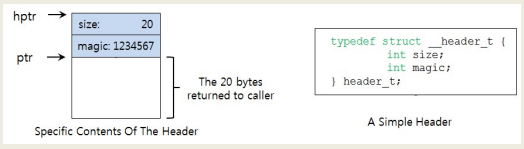
An Allocated Region Plus Header

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.13

MEMORY HEADERS - 2



Specific Contents Of The Header

A Simple Header

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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.14

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);  
}
```

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.15

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;
```

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

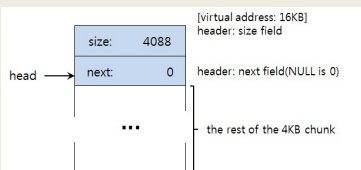
L13.16

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:



(virtual address: 16KB) header: size field

header: next field(NULL is 0)

the rest of the 4KB chunk

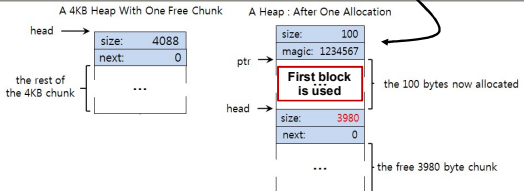
February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.17

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

A Heap : After One Allocation

the 100 bytes now allocated

the free 3980 byte chunk

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.18

FREE LIST: FREE() CALL

■ Addresses of chunks

■ Start=16384  
+ 108 (end of 1<sup>st</sup> chunk)  
+ 108 (end of 2<sup>nd</sup> chunk)  
+ 108 (end of 3<sup>rd</sup> chunk)  
= 16708

8 bytes header

size: 100  
magic: 1234567

...

size: 100  
magic: 1234567

Free this block

size: 100  
magic: 1234567

...

size: 3764  
next: 0

...

The free 3764-byte chunk

Free Space With Three Chunks Allocated

virtual address: 16KB

100 bytes still allocated

100 bytes still allocated (but about to be freed)

100 bytes still allocated

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.19

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

head

size: 100  
magic: 1234567

...

size: 100  
next: 16708

Block Now Free

size: 100  
magic: 1234567

...

size: 3764  
next: 0

...

The free 3764-byte chunk

virtual address: 16KB

100 bytes still allocated

(now a free chunk of memory)

100 bytes still allocated

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.20

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

size: 100  
next: 16492

...

size: 100  
next: 16708

...

size: 100  
next: 16384

...

size: 3764  
next: 0

...

The free 3764-byte chunk

virtual address: 16KB

(now free)

(now free)

(now free)

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.21

GROWING THE HEAP

■ Start with small sized heap

■ Request more memory when full

■ sbrk(), brk()

Segmented heap

Heap

break

sbrk()

break

Address Space

Physical Memory

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.22

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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.23

EXAMPLES

■ Allocation request for 15 bytes

head → 10 → 30 → 20 → NULL

■ Result of Best Fit

head → 10 → 30 → 5 → NULL

■ Result of Worst Fit

head → 10 → 15 → 20 → NULL

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.24

## 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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.25

## 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

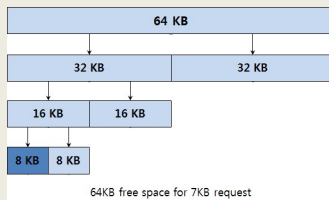
February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.26

## 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



February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.27

## BUDDY ALLOCATION - 2

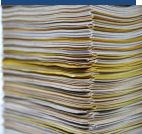
- Buddy allocation: suffers from internal fragmentation
- Allocated fragments, typically too large
- Coalescing is simple
  - Two adjacent blocks are promoted up

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.28

## CHAPTER 18: INTRODUCTION TO PAGING



February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.29

## 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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.30

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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.31

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

64-Byte Address Space Placed In Physical Memory

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.32

PAGING: ADDRESS TRANSLATION

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

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

Here there are just four pages...

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.33

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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.34

PAGING DESIGN QUESTIONS

- Where are page tables stored?
- What are the typical contents of the page table?
- How big are page tables?
- Does paging make the system too slow?

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.35

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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.36

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 <sub>0</sub>
VPN <sub>1</sub>
VPN <sub>2</sub>
...
...
VPN <sub>1048576</sub>

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.37

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 4MB x 100 = 400MB
- If computer has 4GB memory (maximum for 32-bits), the page table consumes 10% of memory

400 MB / 4000 GB
- Is this efficient?

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.38

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																				U		R/W		D		A		PWT		PWT		R/W		R	

An x86 Page Table Entry(PTE)

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.39

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																				U		R/W		D		A		PWT		PWT		R/W		R	

An x86 Page Table Entry(PTE)

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.40

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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.41

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

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.42

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 →

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.43

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)
```

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.44

COUNTING MEMORY ACCESSES

■ Example: Use this Array Initialization Code

```
int array[1000];
...
for (i = 0; i < 1000; i++)
    array[i] = 0;
```

■ Assembly equivalent:

```
0x1024 movl $0x0, (%edi,%eax,4)
0x1028 incl %eax
0x102c cmpl $0x03e8,%eax
0x1030 jne 0x1024
```

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.45

VISUALIZING MEMORY ACCESSES:  
FOR THE FIRST 5 LOOP ITERATIONS

■ Locations:

■ Page table

■ Array

■ Code

■ 50 accesses for 5 loop iterations

Page Table[39]

Page Table[1]

Array(PA)

Code(PA)

Memory Access

February 26, 2018

TCCS422: Operating Systems [Winter 2018]  
Institute of Technology, University of Washington - Tacoma

L13.46

QUESTIONS