CSCE 313-200
Introduction to Computer Systems
Spring 2024

Memory III

Dmitri Loguinov
Texas A&M University

April 24, 2024

Homework #3

- Tested Rabin-Karp performance on enwiki-all.txt
  - FILE_FLAG_NO_BUFFERING, B = 2 MB, 50 slots
  - 8-core Skylake-X server w/RAID @ 4 GB/s

	keywords-B			keywords-D	
	Time	Speed	Found	Speed	Found
A	12.5	2.4 GB/s	318,798,734	66 MB/s	3,374,677,735
B	22.1	1.4 GB/s	319,017,279	49 MB/s	3,374,677,735
C	40.9	0.7 GB/s	319,017,279	--	--
Ref	7.50	4.0 GB/s	319,017,279	125 MB/s	3,374,677,735

Chapter 7: Roadmap

7.1 Requirements

7.2 Partitioning

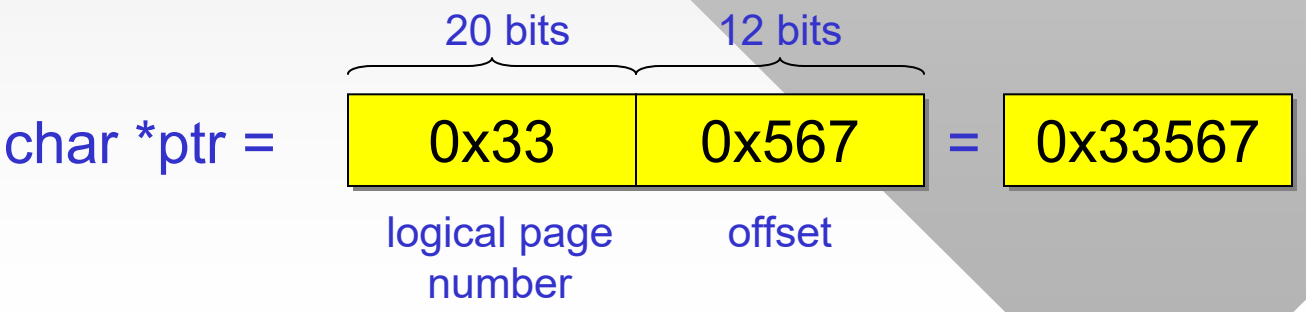
7.3 Paging

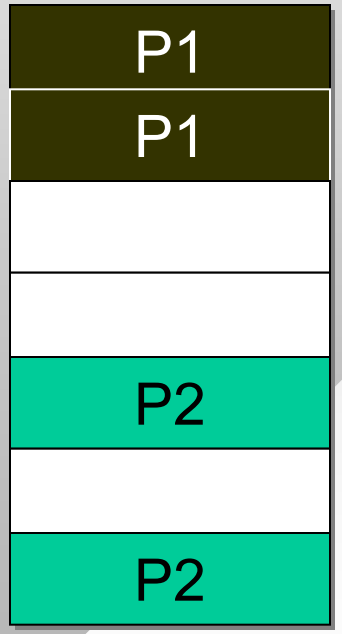
7.4 Segmentation

7.5 Security

Paging

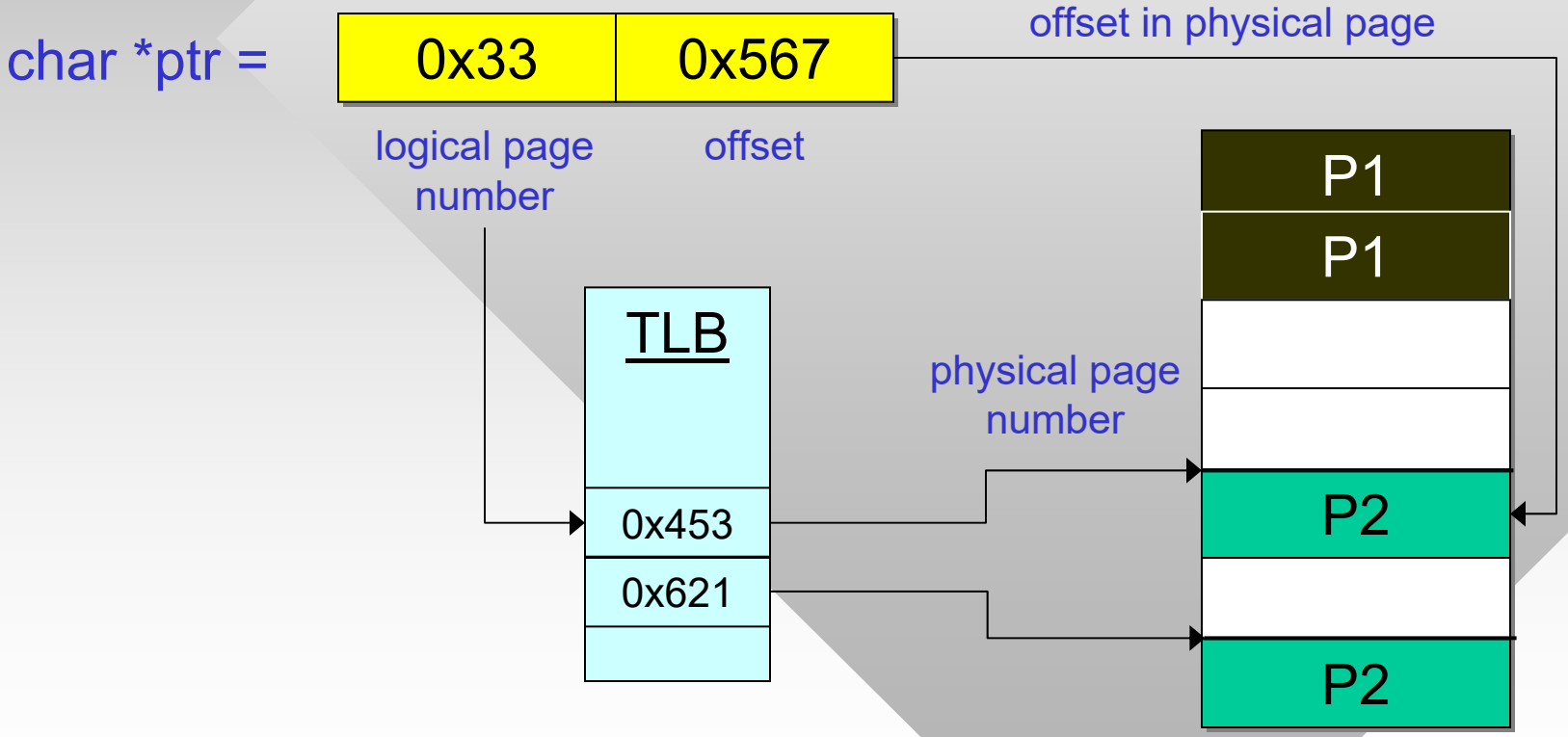
- Paging allows the OS to allocate non-contiguous chunks of space to application requests
  - Hardware finds the page in RAM by transparently mapping from logical to physical addresses
- Logical address consists of two parts
  - Page number
  - Offset within that page
- Example: 32 bit address, 4 KB pages





Paging

- Conversion of page numbers is done using the TLB (Translation Lookaside Buffer):



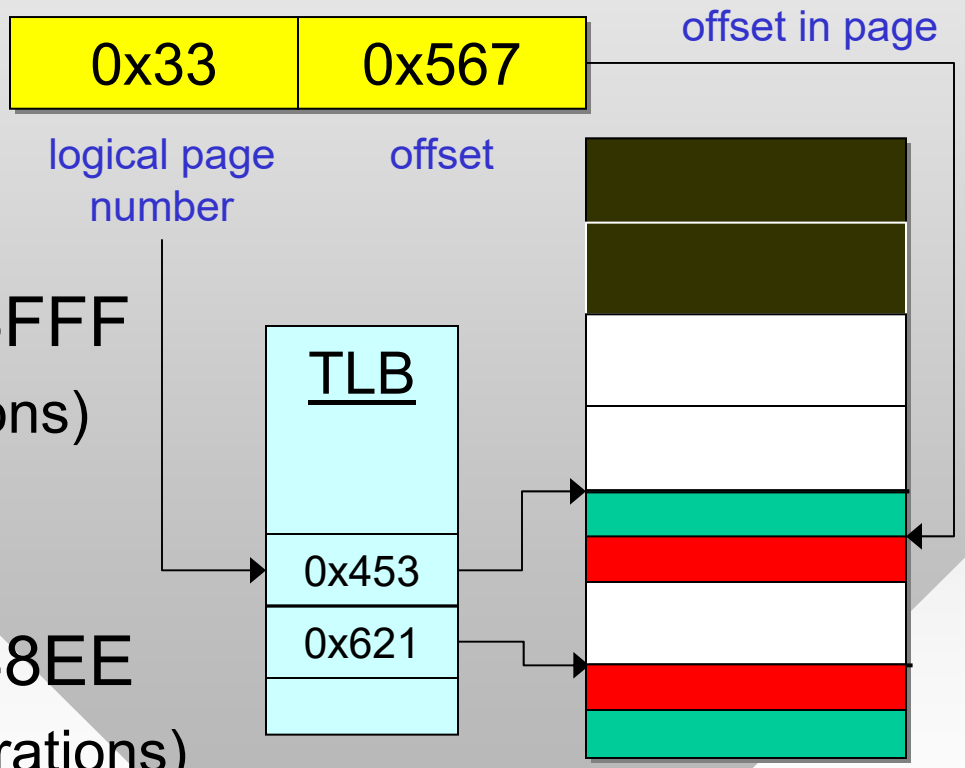
- Each process owns a page table controlled by OS

Paging

- Example: write 5000 bytes to array ptr[]

```
char *ptr = 0x33567;

for (int i = 0; i < 5000; i++)
    ptr [i] = i;
```



- Ptr + i = 0x33567-0x33FFF
  - i = 0-2712 (2713 iterations)
  - Physical address range 0x453567-0x453FFF
- Ptr + i = 0x34000-0x348EE
  - i = 2713-4999 (2287 iterations)
  - Physical address range 0x621000-0x6218EE

Paging

- To avoid doubling RAM latency on random access, TLB is kept in dedicated cache memory
  - CPU performs a lookup before sending address to RAM
- Within a given page, no control of address validity
  - However, if a process goes far enough to hit next page, the TLB must have an entry for that page with correct permissions
  - If not, a page fault is thrown and the process is killed
- These concept allow allocation of pages beyond physical RAM, swapping to disk, loading to new addr
- Example: computer with 8 GB of RAM
  - Process requests 7 GB, but all other resident software and kernel occupy 2.5 GB

Paging

- Whatever pages aren't being used are swapped to disk
  - Special pagefile provides space for this operation
  - Usually, pagefile.sys is twice the size of RAM
- Memory classification
  - Non-pageable memory: special types of pages that cannot be swapped to disk (e.g., parts of OS, locked pages, AWE segments, large-page allocations)
  - Commit set: all pageable memory of the process (i.e., allocated in the page file)
  - Working set: touched (accessed) pages in RAM
  - Private working set: a subset of the working set (e.g., heap-allocated) that is not shared with other processes
- The last three can be seen in Task Manager

Paging

- Access to page outside working set causes a page fault
- Types of page faults
  - Hard: requires the page to be read from disk
  - Soft: can be resolved with remapping (e.g., pages exists in working set of another process or first-time access)
  - Violation: access outside virtual space of this process or using incompatible permissions (e.g., writing to read-only page)
- Hard/soft faults are handled transparently by OS
- Example: allocate 1 GB of committed memory

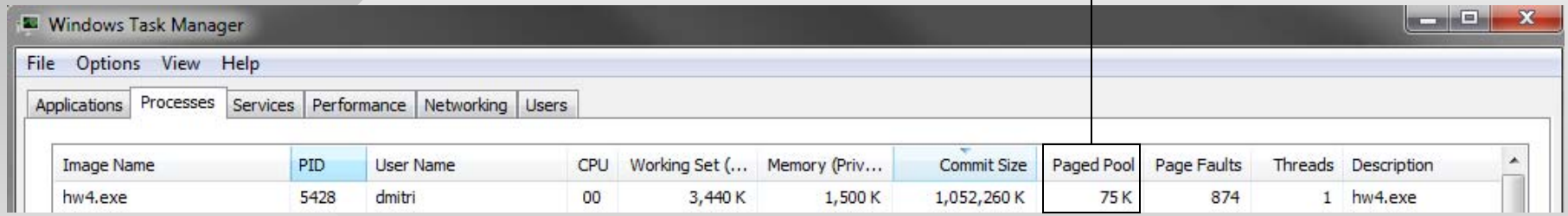
```
char *buf = (char *) VirtualAlloc (NULL, 1 << 30, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
```

- Commit size, working set size, and private set size?

Paging

paged pool contains kernel objects (e.g., handles) suitable for paging

- Examine Task Manager:

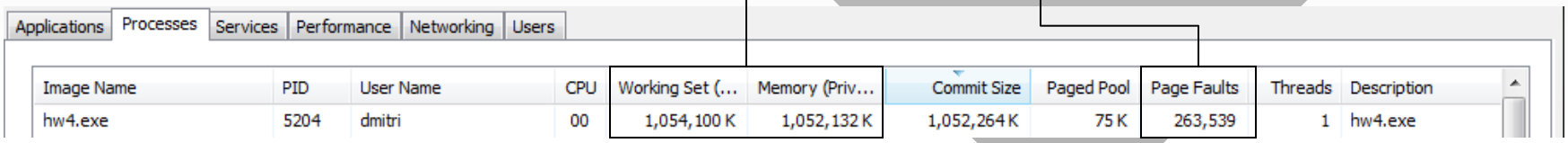


- Commit size is 1 GB as expected, but none of that memory has been allocated in physical RAM yet
  - OS doesn't know which pages we'll need and in what order
  - Conserves physical RAM as much as possible
- Write something into each page:

```
memset (buf, 0x55, 1 << 30);
```

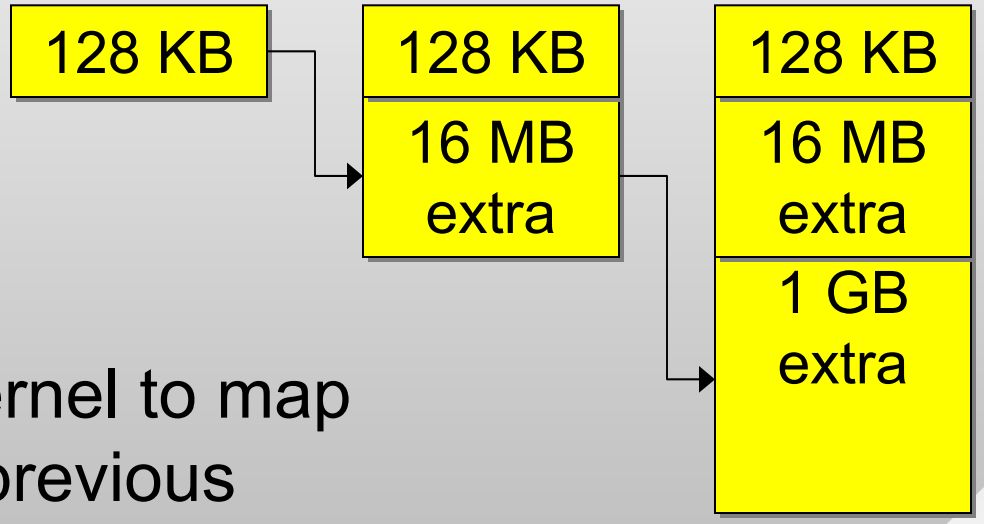
both working sets change

260K soft page faults



Working with Buffers

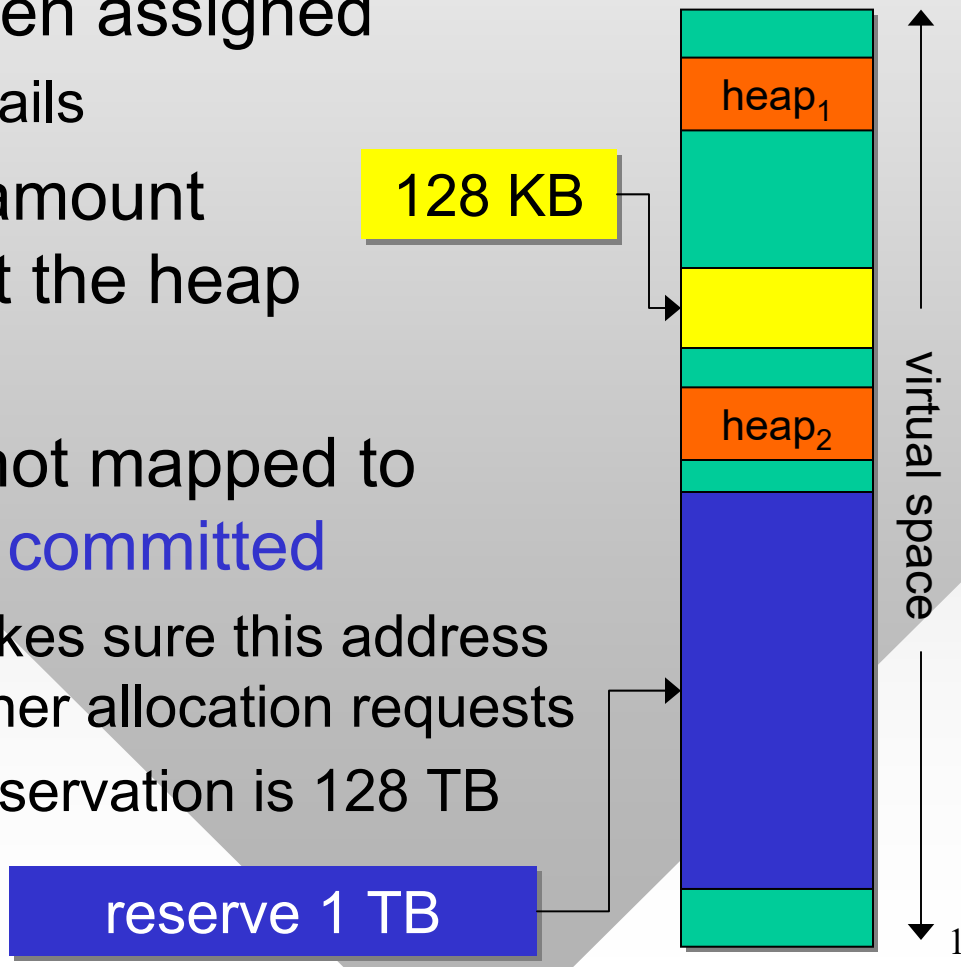
- Suppose we intend to dynamically expand the region of allocated memory
  - But don't want to copy data over to the new area each time
  - Similar to HeapReAlloc
- Would like to ask the kernel to map the continuation of the previous buffer to some additional physical pages:




```
// allocation of initial 128 KB succeeds
int size = 1 << 17;
char *buf = (char *) VirtualAlloc (NULL, size, MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
// attempt to add 16 MB to this buffer may fail
char *result = (char *) VirtualAlloc (buf + size, 1 << 24,
          MEM_COMMIT|MEM_RESERVE, PAGE_READWRITE);
```

Working with Buffers

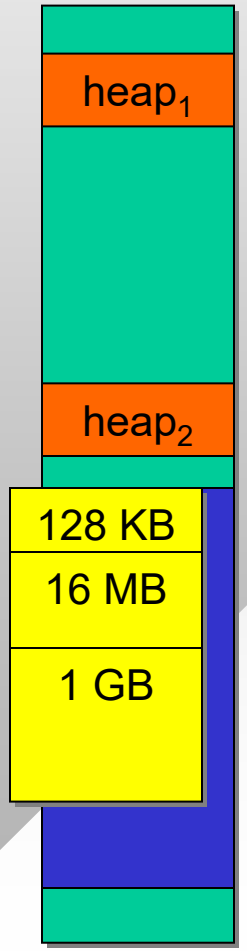
- The problem is that the virtual space beyond buf + size might have already been assigned
  - Allocation in this case fails
- Idea: reserve a huge amount of virtual space so that the heap can't use it
- Reserved memory is not mapped to pagefile until explicitly committed
  - Reservation simply makes sure this address space is not used in other allocation requests
  - In Server 2016, max reservation is 128 TB



Working with Buffers

- Can now commit memory in our reserved space

```
// reserve 1 TB
char *bufMain = (char *) VirtualAlloc (NULL, (uint64) 1<<40,
          MEM_RESERVE, PAGE_READWRITE);
// allocate 128 KB
int size0 = 1 << 17;
char *buf0 = (char *) VirtualAlloc (bufMain, size0,
          MEM_COMMIT, PAGE_READWRITE);
// now add 16 MB to this buffer
int size1 = 1 << 24;
char *buf1 = (char *) VirtualAlloc (buf0 + size0, size1,
          MEM_COMMIT, PAGE_READWRITE);
// now add 1 GB
int size2 = 1 << 30;
char *buf2 = (char *) VirtualAlloc (buf1 + size1, size2,
          MEM_COMMIT, PAGE_READWRITE);
```



- Memory may be decommitted as needed

```
// decommit 4KB from the middle of committed space
char *result = (char*) VirtualFree (buf1, 1 << 12, MEM_DECOMMIT);
```