CSCE 313-200
Introduction to Computer Systems
Spring 2025

Memory II

Dmitri Loguinov
Texas A&M University

April 15, 2025

Chapter 7: Roadmap

7.1 Requirements

7.2 Partitioning

7.3 Paging

7.4 Segmentation

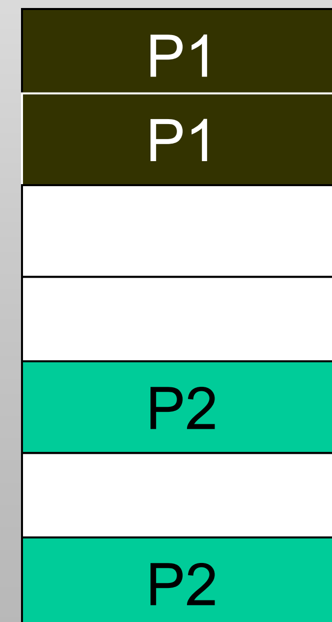
7.5 Security

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

char *ptr = $\overbrace{\boxed{\text{0x33}}}^{\text{20 bits}}\overbrace{\boxed{\text{0x567}}}^{\text{12 bits}} = \boxed{\text{0x33567}}$

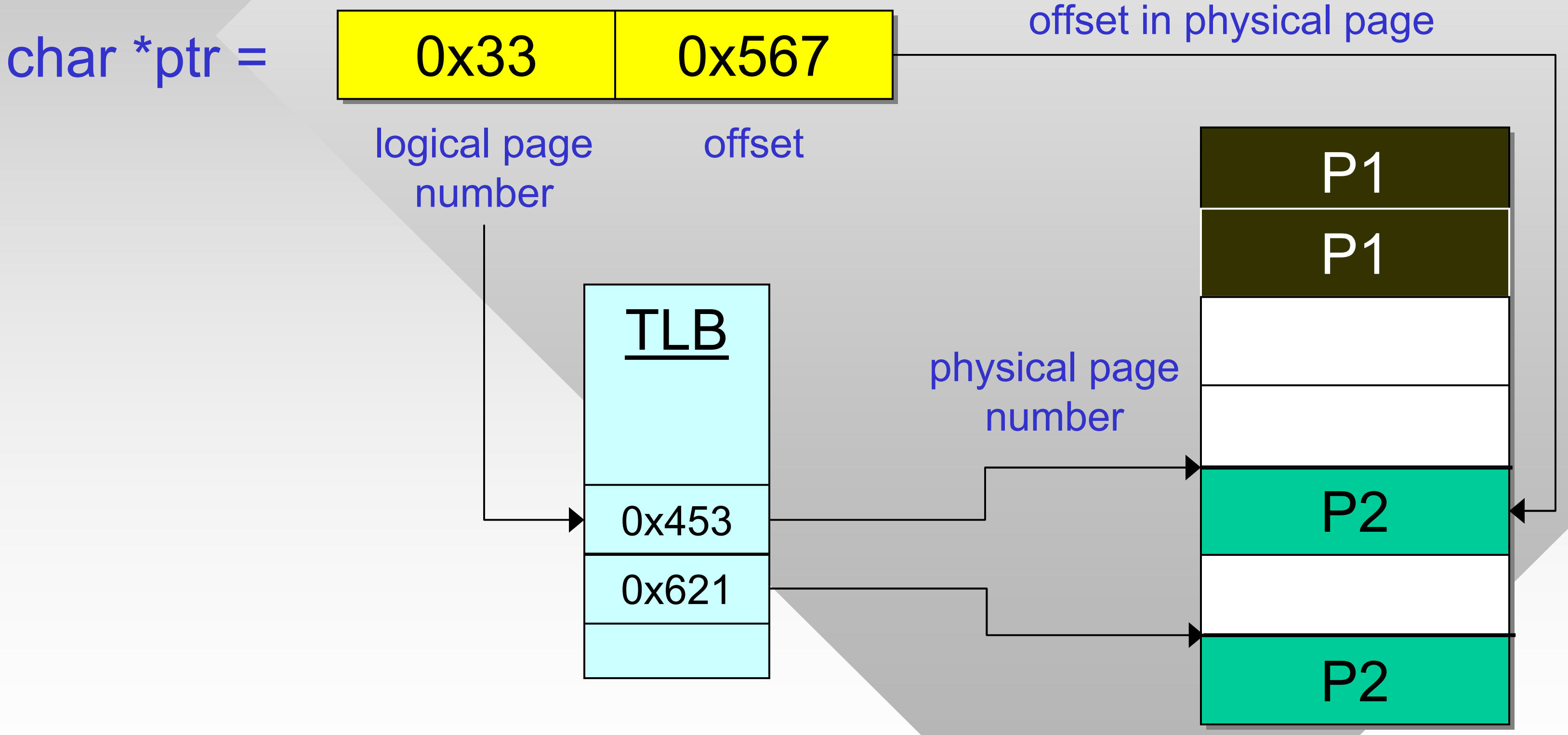
logical page number | offset



RAM

Paging

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



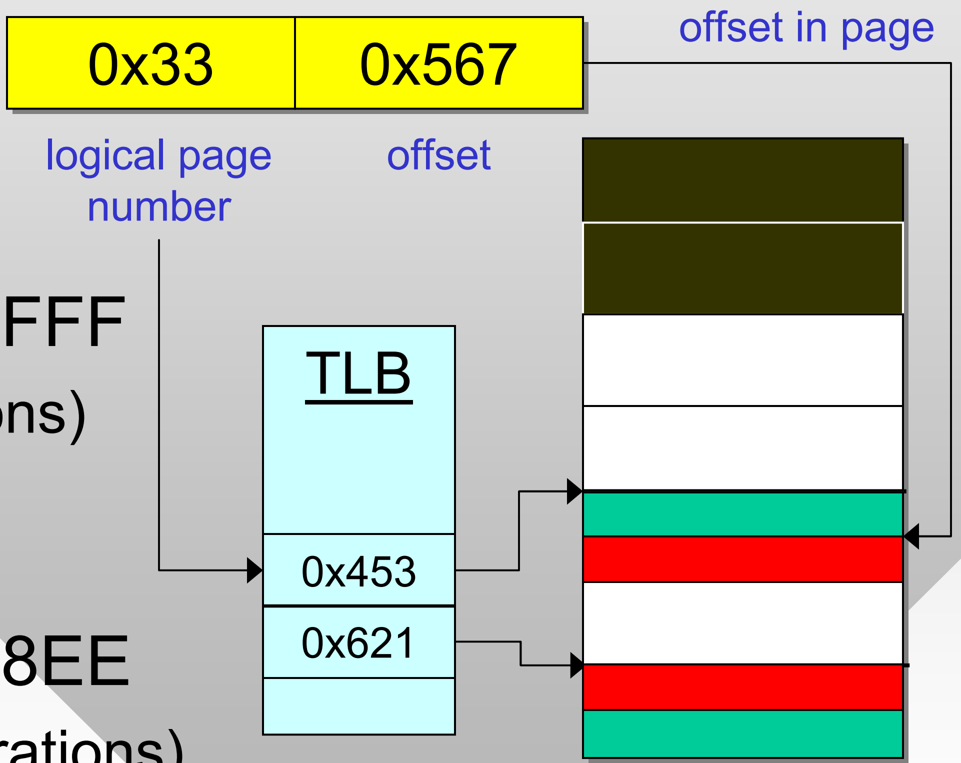
- Each process owns a page table controlled by OS

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 concepts 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

- Rarely used pages 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 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:

Windows Task Manager

File Options View Help

Applications | Processes | Services | Performance | Networking | Users

Image Name	PID	User Name	CPU	Working Set (...	Memory (Priv...	Commit Size	Paged Pool	Page Faults	Threads	Description
hw4.exe	5428	dmitri	00	3,440 K	1,500 K	1,052,260 K	75 K	874	1	hw4.exe

- 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

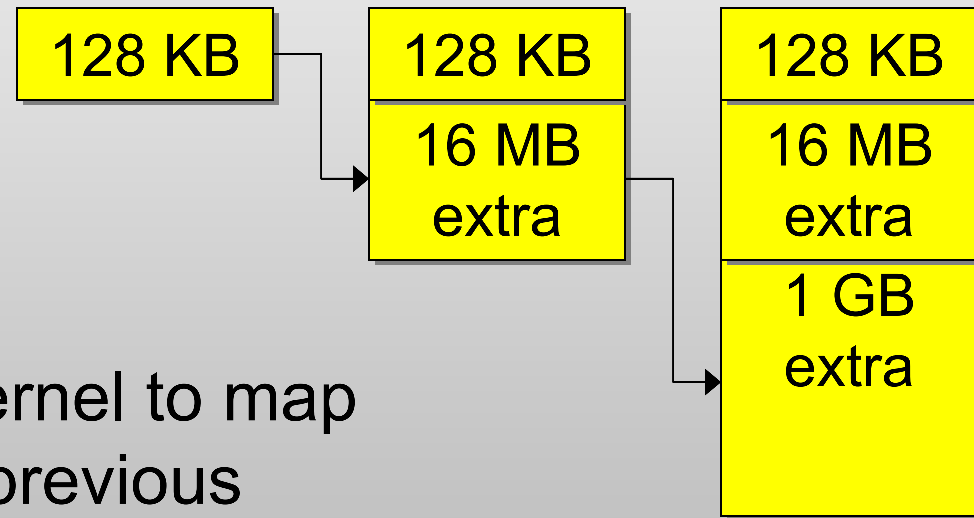
260K soft page faults

Applications | Processes | Services | Performance | Networking | Users

Image Name	PID	User Name	CPU	Working Set (...	Memory (Priv...	Commit Size	Paged Pool	Page Faults	Threads	Description
hw4.exe	5204	dmitri	00	1,054,100 K	1,052,132 K	1,052,264 K	75 K	263,539	1	hw4.exe

Working with Buffers

- Suppose we intend to dynamically expand the region of allocated memory
  - Similar to HeapReAlloc
  - But don’t want to copy data over to the new area each time

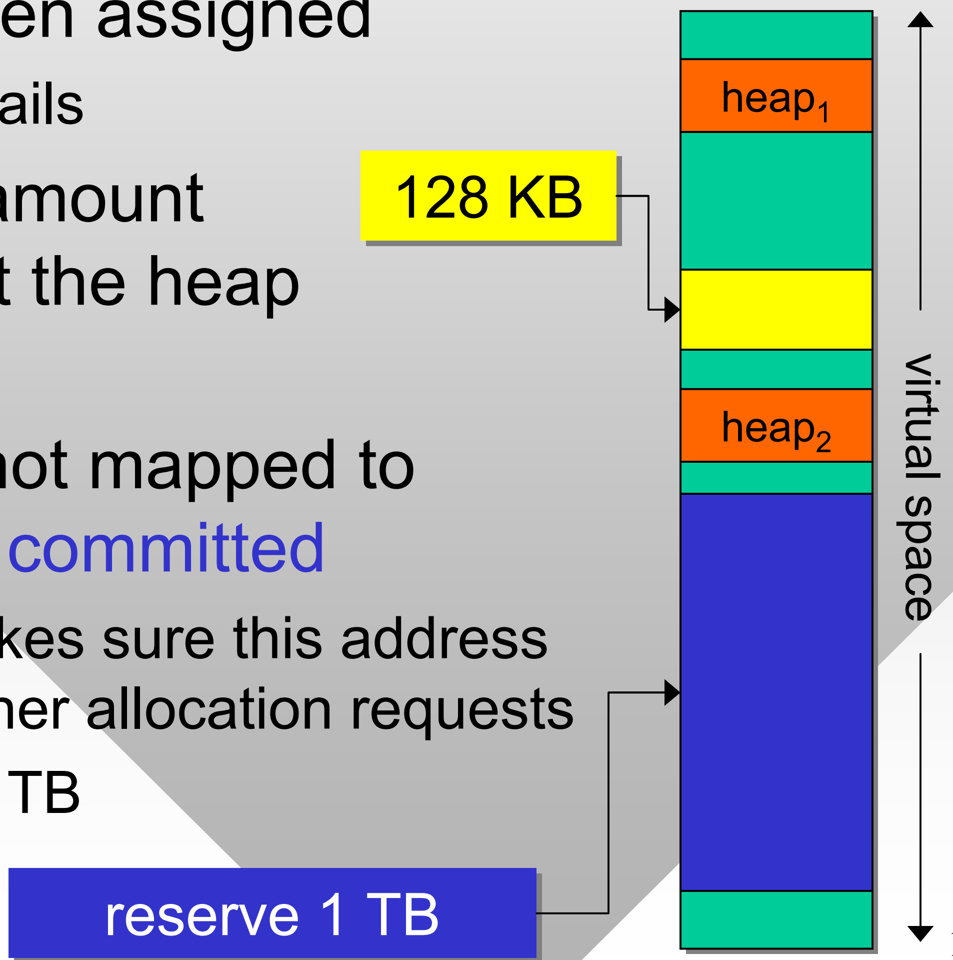


- Would like to ask the kernel to map the continuation of the previous buffer to some additional physical pages:

```
// 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

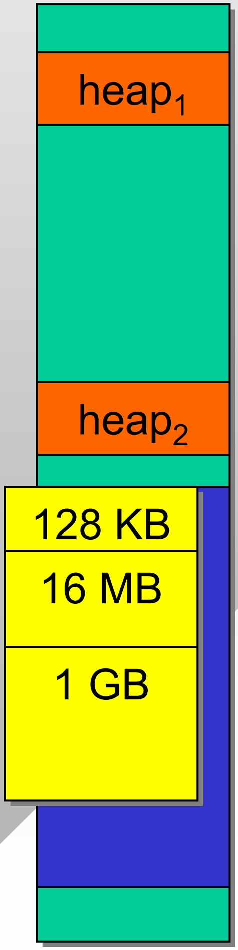
- The problem is that the virtual space beyond buf + size might have already been assigned
  - Allocation in this case fails
- Idea: reserve a huge amount of virtual space so that the heap can't use it
- Reserved memory is not mapped to pagefile until explicitly committed
  - Reservation simply makes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



Working with Buffers

- Can now commit memory in our reserved space

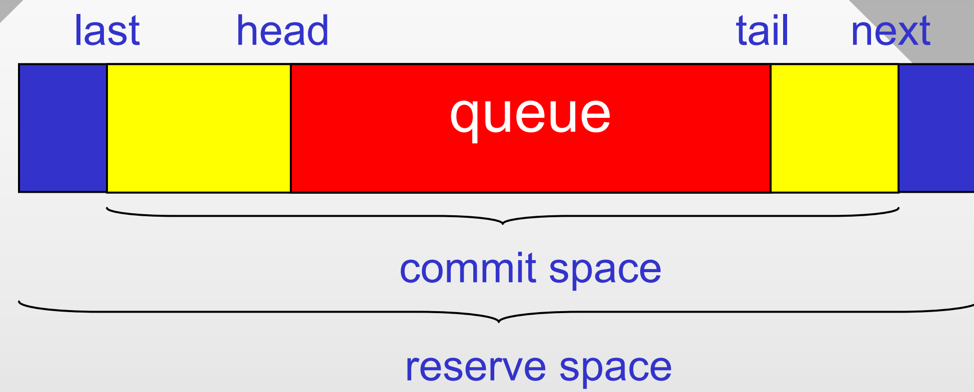
```
// reserve 1 TB
char *bufMain = (char *) VirtualAlloc (NULL, (uint64) 1<<40,
          MEM_RESERVE, PAGE_READWRITE);
// allocate 128 KB
int size0 = 1 << 17;
char *buf0 = (char *) VirtualAlloc (bufMain, size0,
          MEM_COMMIT, PAGE_READWRITE);
// now add 16 MB to this buffer
int size1 = 1 <<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

Queue Example



- Design self-resizing Q that keeps data contiguous and never has to memcpy
  - Code below does not handle errors, nor does it compute how much to expand or shrink by

```
Q::Q () {
    reserveSize = (uint64) 1<<40;
    char *bufMain = (char *) VirtualAlloc (NULL, reserveSize,
                                    MEM_RESERVE, PAGE_READWRITE);
    head = tail = (Item*) (next = last = bufMain);
}

void Q::push (Item x) {
    // overflow of current commit section?
    if (tail + sizeof(x) > next) {
          // add some commit space in front of the tail
          VirtualAlloc (next, expandSize, MEM_COMMIT, PAGE_READWRITE);
          next += expandSize;
    }

    *tail++ = item;
}
```

```
class Q {
    char *next, *last;
    char *bufMain;
    Item *head, *tail;
};
```

Queue Example

- Shrink the committed region during pop

```
Item Q::pop (void) {
    if (head > last + shrinkSize) {
          // decommit old memory behind the head
          VirtualFree (last, shrinkSize, MEM_DECOMMIT);
          last += shrinkSize;
    }

    return *head++;
}
```

- Problem #1: cannot commit/decommit too fast
  - Keep expandSize and shrinkSize around 1 MB
- Problem #2: queue eventually overflows when reserveSize is exceeded
  - If 128 TB of virtual space is not enough, memcpy or linked lists of buffers cannot be avoided

Disk I/O Example

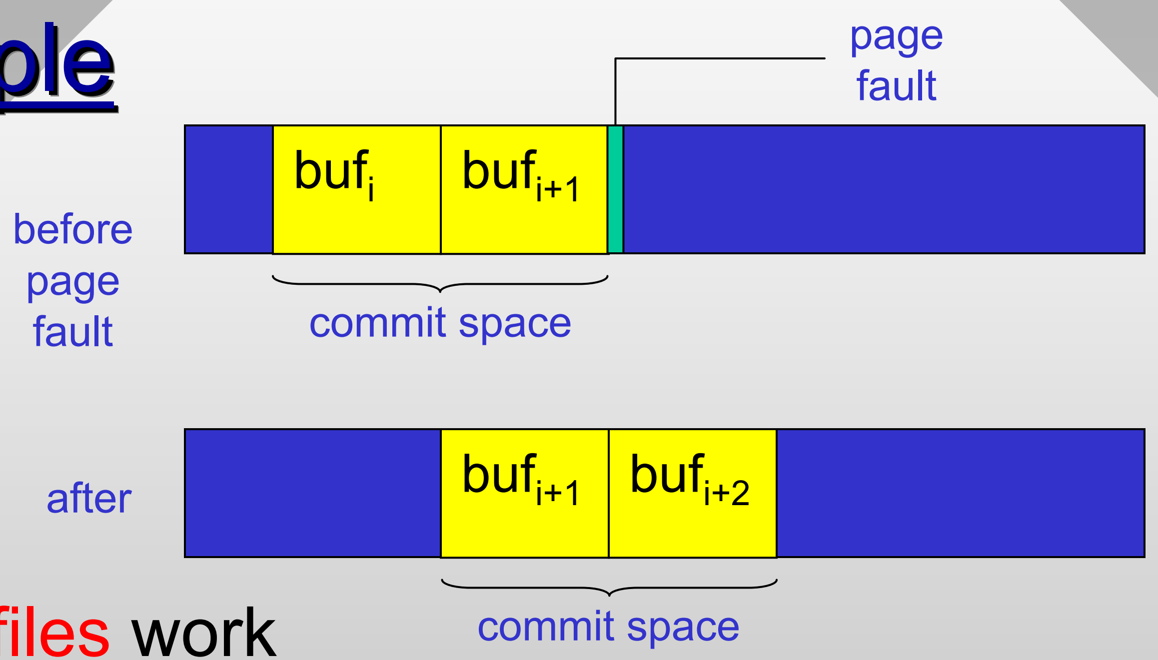
single-threaded application that reads a file larger than RAM

- Assume there exists some complex data processing library whose APIs only work with contiguous buffers
  - Can the library be hacked to work with shadow buffers?
- If so, what if some records do not fit in shadow buffer?
  - Recall that shadow buffers must be at least the size of the longest record (e.g., word) in the file
- Some files may have extremely long records
  - E.g., each record in a graph contains a node ID and a list of its neighbors; for 300M neighbors, 2.4 GB per record
- Worse yet, what if individual records do not fit in RAM?
  - E.g., search engine index contains a keyword hash and a list of pages where the keyword appears; for a popular keyword found in 5B pages, this requires 40 GB

Disk I/O Example

- Suppose the library is a streaming data processor
  - Operates on data only sequentially and going forward
  - Never returns by more than X bytes, where X is small
- Goal: use virtual memory to create an illusion of a continuous file in RAM for this library
- Idea: let the library run into page faults
  - Which we catch, commit the next chunk of virtual memory, read the next file block into it, and return control to the API
  - Blocks of memory that are 2 buffers behind are decommitted assuming buffer size is no smaller than X
- Performance (AMD Phenom II): page-fault rate is ~900K/sec

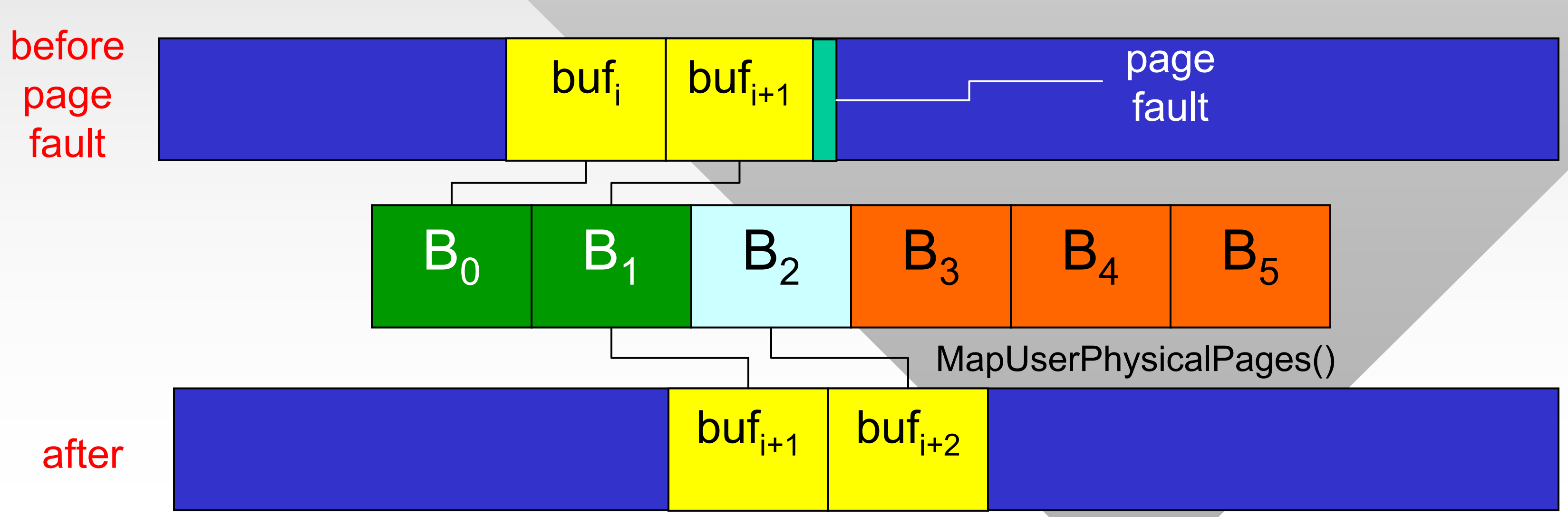
Disk I/O Example



- What's a good reserve size?
  - Length of file
- This is how memory-mapped files work
  - Slightly more general as they allow random access
  - Read small buffer surrounding the page fault
  - Decommit old pages using LRU or some other technique
  - See CreateFileMapping and MapViewOfFile
- <u>Problem</u>: this method can only do single-buffering
  - Stalls processing while the next buffer is being read
  - Only solution is to read ahead into other RAM locations, then memcpy into buf_{i+2} during page faults

Disk I/O Example

- Using AWE (Address Windowing Extensions)
  - Six physical buffers allocated by disk thread, into which it reads the file, wrapping back to B_0 after B_5
  - Two green buffers are mapped to virtual addresses currently being processed by the library; B_2-B_5 are used for read-ahead
  - On page fault, the oldest buffer B_0 is unmapped, the next buffer B_2 is mapped where the page fault occurred



Disk I/O Example

- Writing-to-buffer benchmark
  - 1) No remapping or page-fault processing

```
char *buf = VirtualAlloc (NULL, 1e9, MEM_COMMIT|MEM_RESERVE, ...);
```

  - 2) Reserve virtual memory, catch page faults, commit new chunks of size 1 MB, decommit old chunks

```
char *buf = VirtualAlloc (NULL, 1e9, MEM_RESERVE, ...);
__try {
          writeToPtr (buf, 1e9);
}
__except ( ... ) {

}
```

  - 3) Reserve physical memory (AWE), catch page faults, remap chunks of size 1MB, unmap old chunks

Disk I/O Example

- Two versions of writeToPtr():

```
writeToPtrA (char *buf, int size) {
    for (int i=0; i < size; i++)
          buf[i] = 55;
}
```

```
writeToPtrB (char *buf, int size) {
    memset (buf, 55, size);
}
```

- Benchmark results:

Mapping	writeToPtr	Working set	Page faults	Time
1) None	loop	1 GB	245,493	3.4 sec
	memset	same		343 ms
2) Commit	loop	5.3 MB	245,327	3.2 sec
	memset	same		499 ms
3) Physical	loop	5.3 MB	1,361	3.1 sec
	memset	same		156 ms