Virtual Memory II: Thrashing & Dynamic Memory Management

Note: Some slides and/or pictures in the following are adapted from the text books on OS by Silberschatz, Galvin, and Gagne AND Andrew S. Tanenbaum and Albert S.Woodhull. Slides courtesy of Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, David Wagner, etc.

Outline

PART-I

- What is thrashing?
- Solutions to thrashing
  - Approach 1: Working Set
  - Approach 2: Page fault frequency

PART-II

- Dynamic Memory Management
  - Memory allocation and deallocation and goals
  - Memory allocator - impossibility result
    - Buddy-block scheme
    - Slab allocation scheme

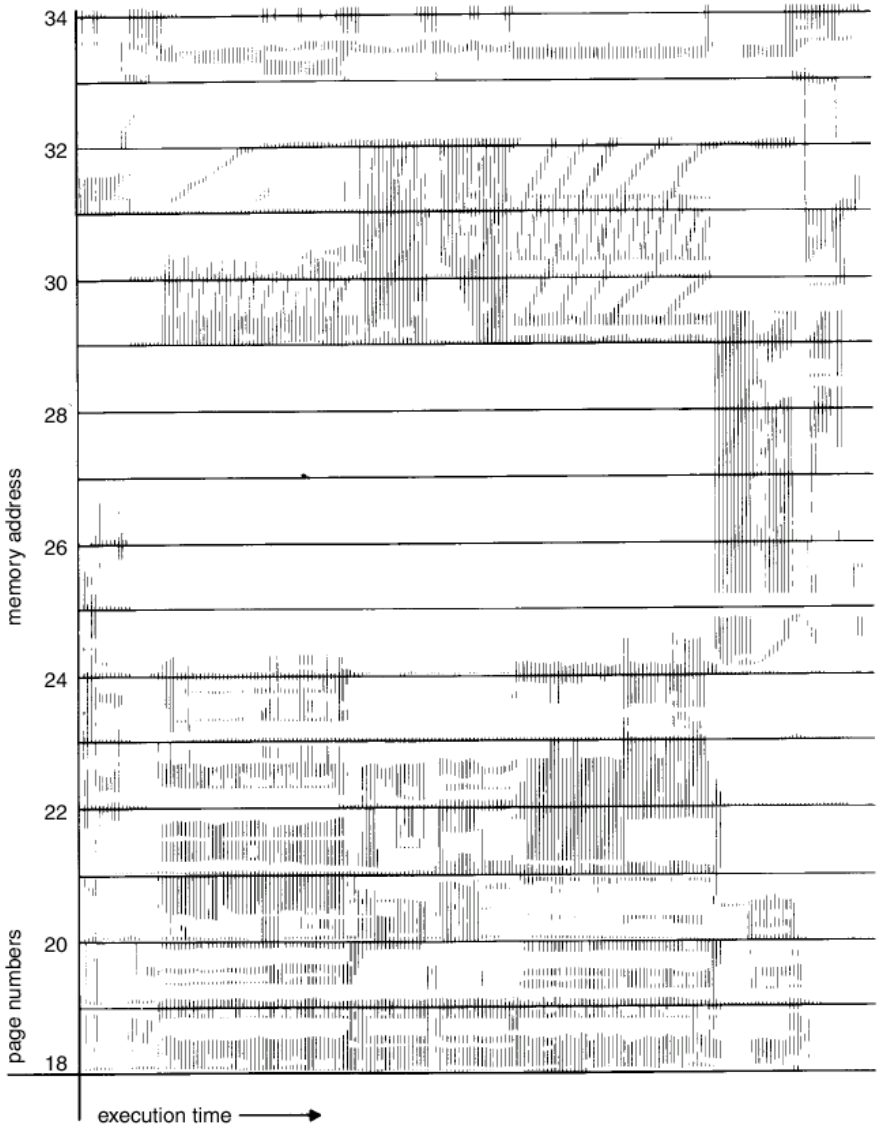
PART-III

- Other Considerations

Locality in a Memory-Reference Pattern

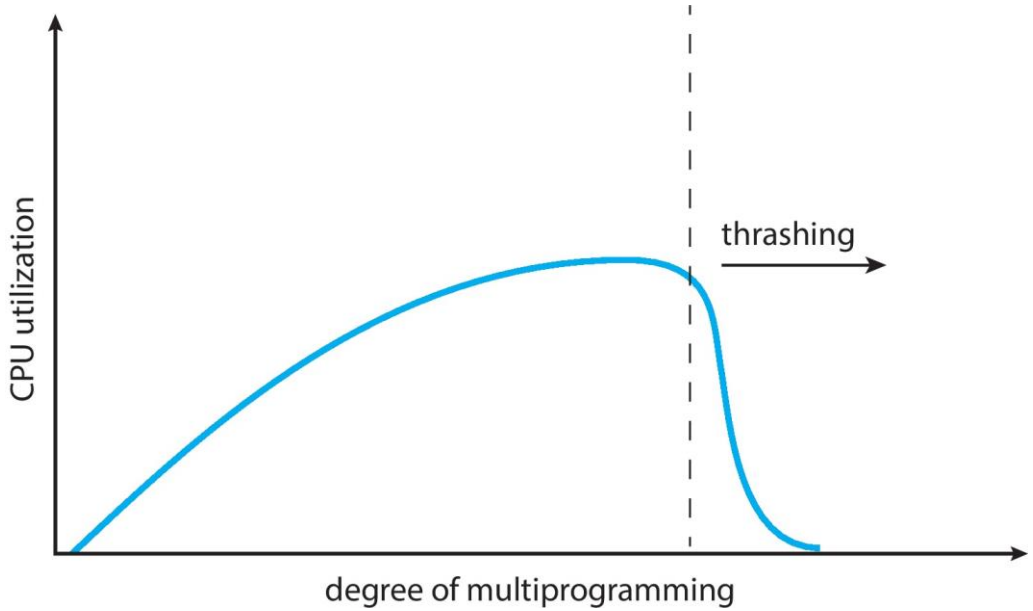
- Program Memory Access Patterns have both temporal and spatial localities
  - Group of Pages accessed along a given time window is called the “Working Set”
  - Working Set defines minimum number of pages needed for process to behave well

- Not enough memory for Working Set ⇒ Thrashing
  - Better to swap out process?



Thrashing

- **Definition:** High page fault rate (swapping pages in and out continuously) that occurs because processes in system require more memory than what had been allocated to them
  - They keep accessing memory addresses that are not there in main memory
  - And keep throwing out page that will be referenced soon



- Why does it occur?
  - Poor locality, past != future
  - There is reuse, but all active pages of the process do not fit in the allocated main memory
  - Too many processes in the system

Approach 1: Working Set

- Peter Denning, 1968
  - He used this term to denote memory locality of a program

- **Definition:** pages referenced by process in last Δ time-units comprise its working set (WS)

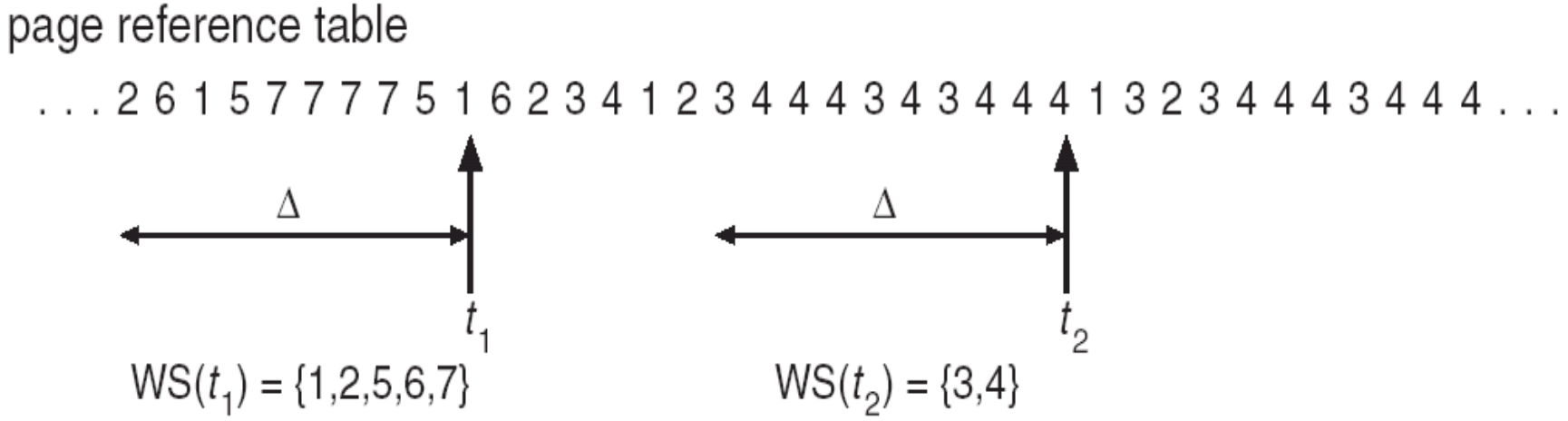
- For our examples, we usually discuss WS in terms of Δ, a "window" *in the page reference string*

  So, Δ is the size of working set window

  But while this is easier on paper it makes less sense in practice!

- In real systems, the window should probably be a period of time, perhaps a second or two.

Working Sets



- The working set size is *no. of* pages in the working set
  - the number of pages touched in the interval [t-Δ+1..t].
- The working set size changes with program locality.
  - during periods of poor locality, you reference more pages.
  - Within that period of time, you will have a larger working set size.
- Goal: keep WS for each process in memory
  - E.g. If Σ WS_i for all *i runnable* processes > physical memory, then suspend a process to avoid thrashing

Working Set Approximation

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and set the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in the working set
- Why is this not completely accurate?
  - Cannot tell (within interval of 5000) where reference occurred
- Improvement: 10 bits and interrupt every 1000 time units
- It can then apply LRU for page replacement at page faults

Using the Working Set

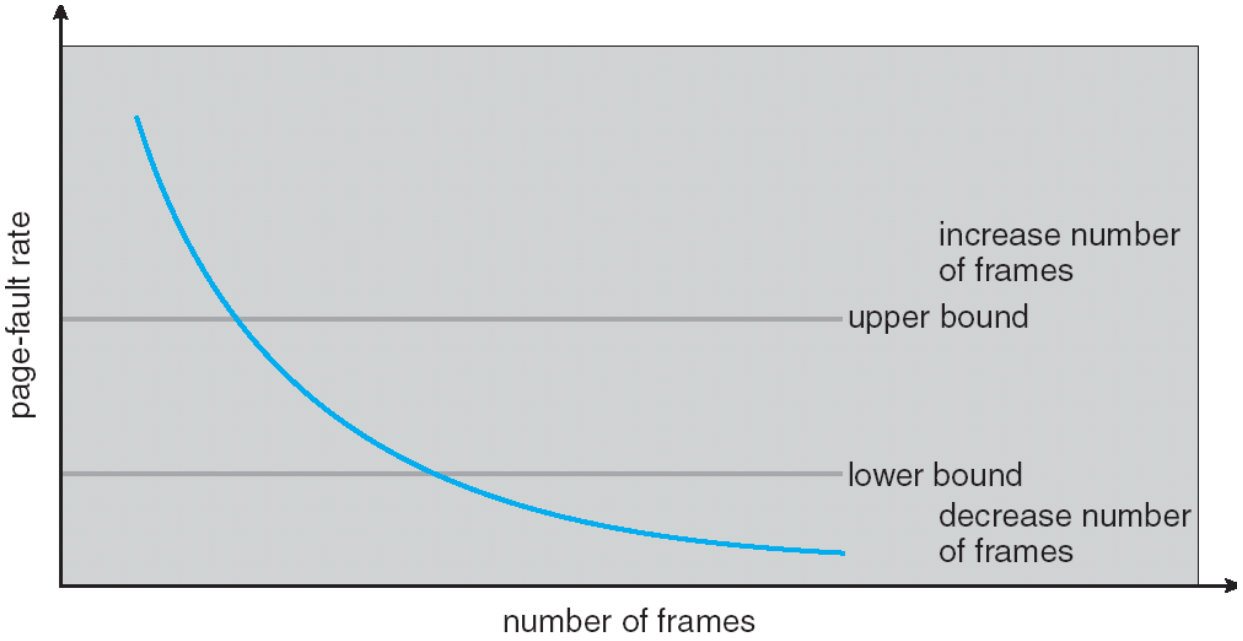
- Used mainly for prepaging
  - Demand paging suffers from large no of page faults when process is initiated or resumed after Swap in
  - Pages in working set are a good approximation
  - During Swap Out, save working set and Swap in all pages in working set when resuming the process
- In Windows, processes have a *max* and *min* WS size
  - At least *min* pages of the process are in memory
  - The *max* WS can be specified by the application
  - If > *max* pages in memory, on page fault a page is replaced
  - Else if memory is available, then WS is increased on page fault

How do WS and LRU compare?

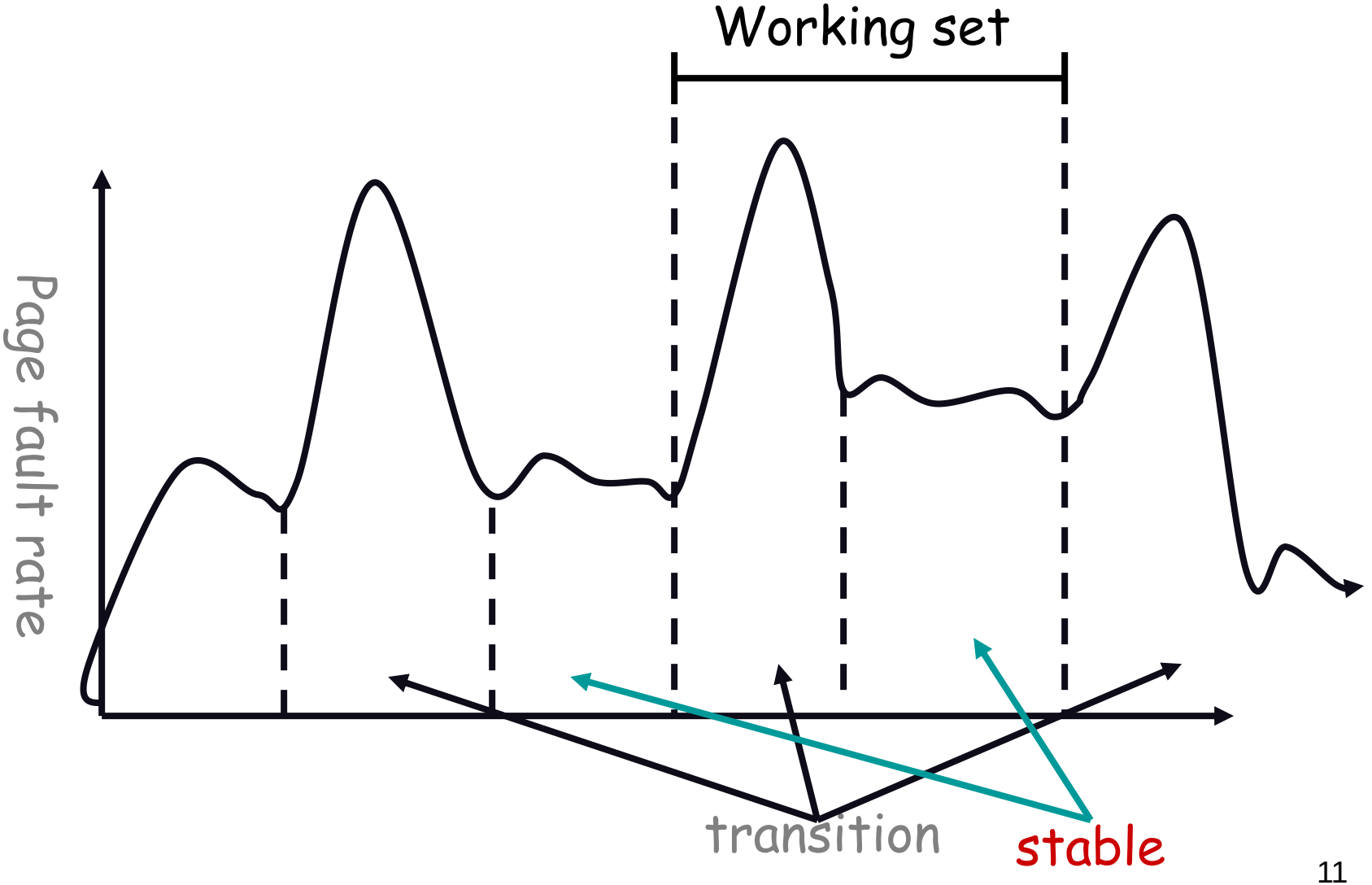
- Suppose we use the same value of Δ page references
  - WS removes pages if they aren't referenced and hence keeps less pages in main memory
  - When there is a page fault, it is using an LRU policy!
  - LRU will keep all Δ pages in memory, whether referenced or not
- Thus LRU often has a lower miss rate, but needs more memory than WS strategy!

Approach 2: Page Fault Frequency

- Thrashing viewed as poor ratio of fetch to work
- PFF = page faults / memory references
- if PFF rises above threshold, process needs more memory
  - not enough memory on the system? Swap out.
- if PFF sinks below threshold, memory can be taken away



Working Sets and Page Fault Rates



END of PART-I

What is thrashing?

Solutions to thrashing

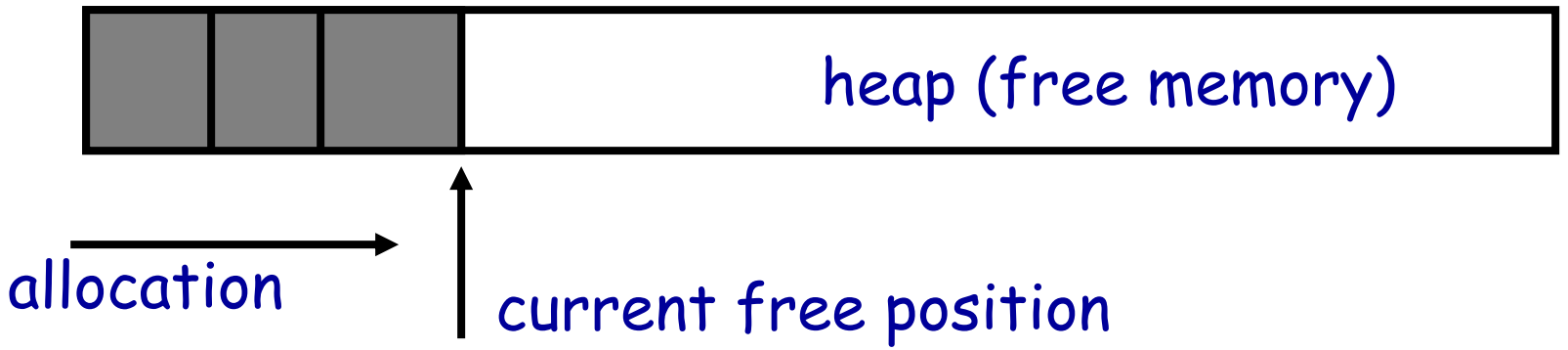
PART-II
Dynamic Memory Management

Dynamic Memory Management: Contiguous Memory Allocation - Part II

- Notice that the O/S kernel can manage memory of user processes in a fairly trivial way:
  - All memory allocations are in units of “pages”
  - And pages can be anywhere in memory… so a simple free list or bitmap is the only data structure needed
  - But suffers from Internal fragmentation on last page frame allocated
- But for kernel processes which have variable-sized objects, we need an efficient kernel heap:
  - Is a very large array (a group of page frames) allocated by OS, managed by kernel program
  - It should be used conservatively to minimize wastage due to fragmentation
  - Used for all dynamic memory allocations
    - malloc/free in C, new/delete in C++, new/garbage collection in Java
  - Some I/O devices need contiguous page frames

Allocation and deallocation

- What happens when you call:
  - `int *p = (int *) malloc(2500*sizeof(int));`
    - Allocator slices a chunk of the heap and gives it to the program
  - `free(p);`
    - Deallocator will put back the allocated space to a free list
- Simplest implementation:
  - Allocation: increment pointer on every allocation
  - Deallocation: no-op
  - Problems: lots of fragmentation
  - This is essential FIFO or First-Fit



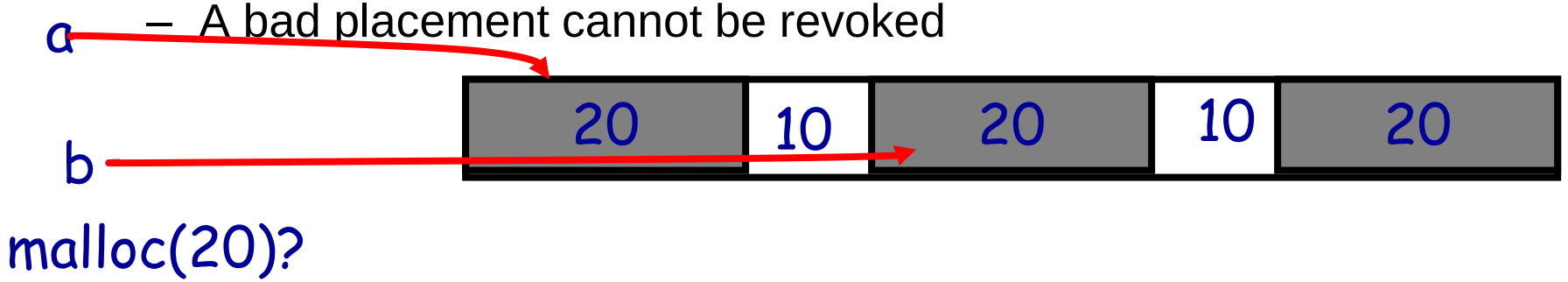
Memory allocation goals

- Minimize space
  - Should not waste space, minimize fragmentation
- Minimize time
  - As fast as possible, minimize system calls
- Maximizing locality
  - Minimize page faults, cache misses
- And many more

- Proven: impossible to construct “always good” memory allocator

Memory Allocator

- Applicable for heaps of User processes and Kernel heap
- What allocator has to do:
  - Maintain free list, and grant memory to requests
  - Ideal: no fragmentation and no wasted time
- What allocator cannot do:
  - Control order of memory requests and frees
  - A bad placement cannot be revoked



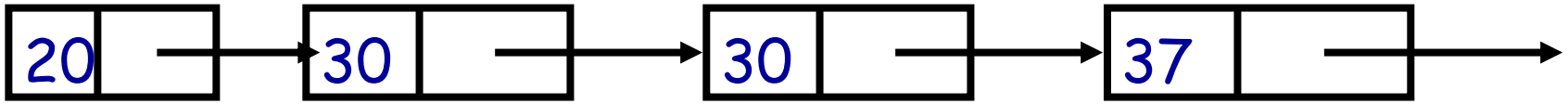
- Main challenge: avoid fragmentation

Impossibility Results

- Optimal memory allocation is NP-complete
- Given any allocation algorithm, ∃ streams of allocation and deallocation requests that defeat the allocator and cause extreme fragmentation

Best Fit Allocation

- Minimum size free block that can satisfy request
- Data structure:
  - List of free blocks
  - Each block has size, and pointer to next free block

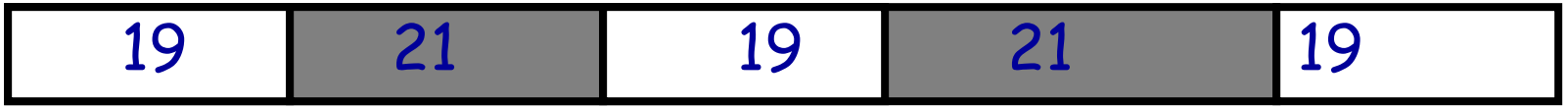


- Algorithm:
  - Scan list for the best fit

Best Fit gone wrong

- Simple bad case: allocate n, m (m<n) in alternating orders, free all the m's, then try to allocate an m+1.
- Example:
  - If we have 100 bytes of free memory
  - Request sequence: 19, 21, 19, 21, 19

  - Free sequence: 19, 19, 19



  - Wasted space: 57!

Design features

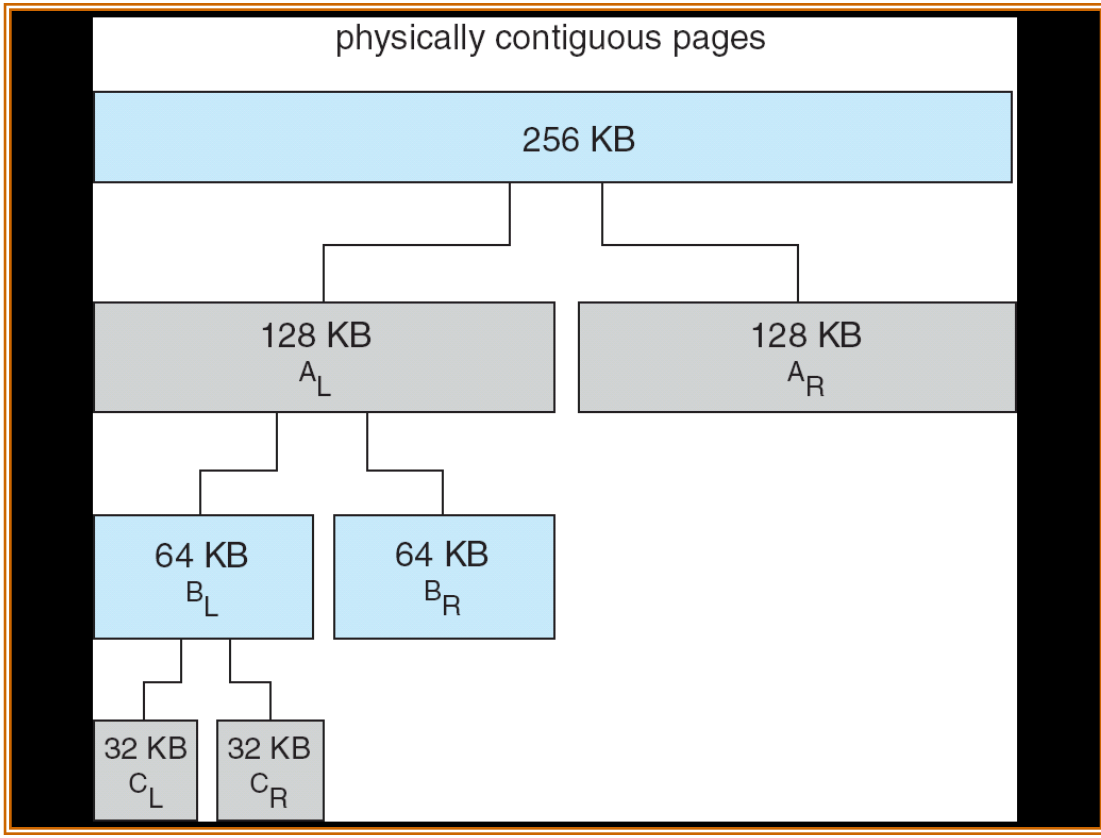
- Which free chunks should service request
  - Ideally avoid fragmentation… requires future knowledge
- Split free chunks to satisfy smaller requests
  - Avoids internal fragmentation
- Coalesce free contiguous blocks to form larger chunks
  - Avoids external fragmentation

1. Buddy-Block Scheme

- Invented by Donald Knuth, very simple
- Idea: Work with memory regions that are all powers of 2 times some “smallest” size
  - 2^k times *b*
  - *4KB, 8KB, 16KB, so on…*
- Satisfies requests in units sized as power of 2
- Round each request *up* to have form $b*2^k$
- When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

Buddy-Block Scheme

- E.g., Request for 21KB
- Advantage: Adjacent buddies can be combined to form larger buddy, called Coalescing

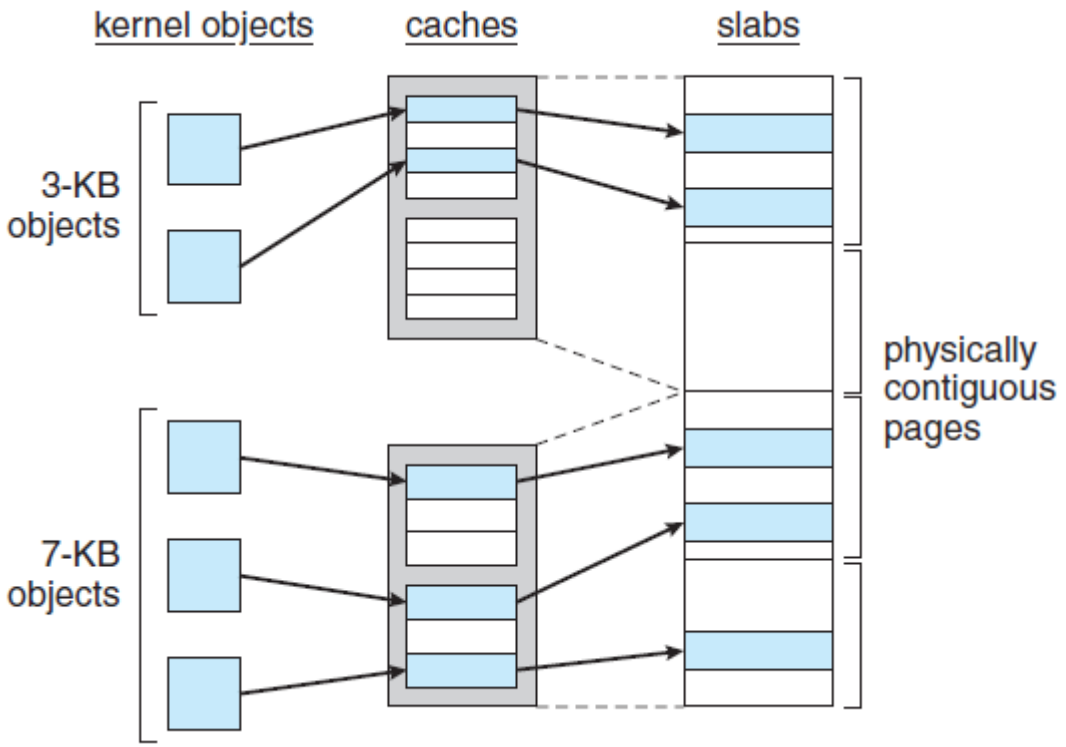


Buddy-Block Scheme

- Keep a free list for each block size (each *k*)
  - When freeing an object, combine with adjacent free buddies if this will result in a double-sized free buddy
- Basic actions on allocation request:
  - If request is a close fit to a buddy on the free list, allocate that region.
  - If request is less than half the size of a buddy on the free list, split the next larger size of region in half
  - If request is larger than *all* free buddies, double the size of the heap (this puts a new page frame on the free list)
- Disadvantages:
  - Internal and external fragmentation!
  - We can't guarantee that < 50% of allocated unit will be wasted due to internal fragmentation

2. Slab Allocation Scheme

- Slab: One or more physically contiguous page frames
- Cache: One or more slabs
- A separate cache for each unique kernel object (e.g., *task_struct*, semaphores, etc)
- When a cache is created, it is populated with free objects
- If no empty slabs available, a new slab is allocated from contiguous free frames in system and given to Cache
- Advantages:
  - No fragmentation
  - Memory allocation requests can be satisfied quickly, like Thread pool

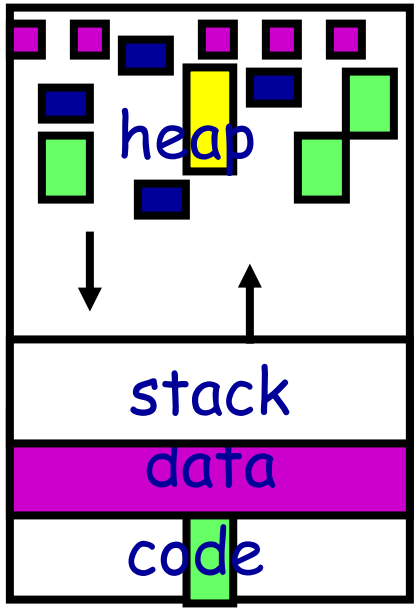


How to get more space?

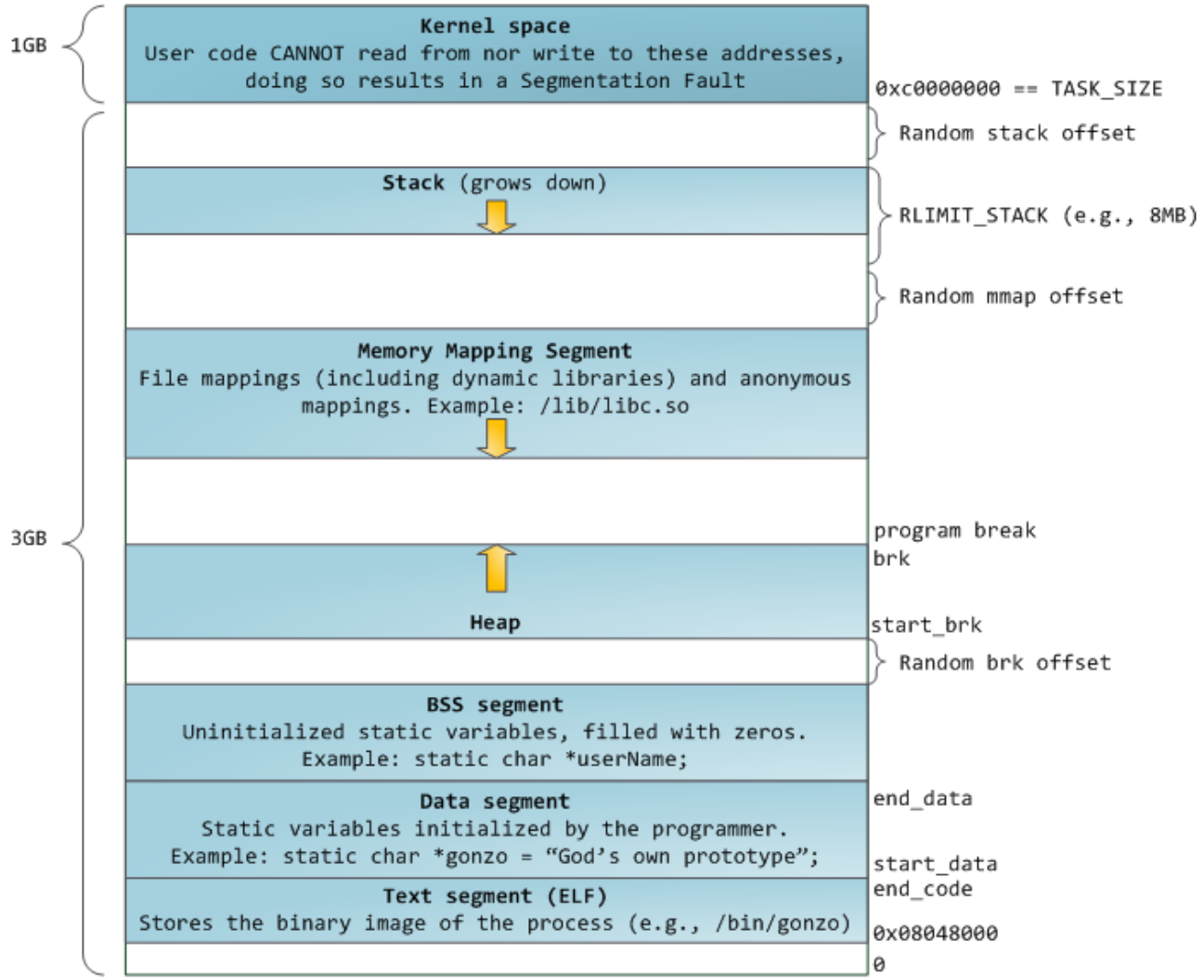
- In Unix, system call sbrk()

```
/* add nbytes of valid virtual address space */
void *get_free_space(unsigned nbytes) {
  void *p;
  if(!(p = sbrk(nbytes)))
        error("virtual memory exhausted");
  return p;
}
```

- Used by malloc if heap needs to be expanded
- Notice that heap only grows on "one side"
- sbrk()/brk() sys calls are used to change size of heap segment of a process



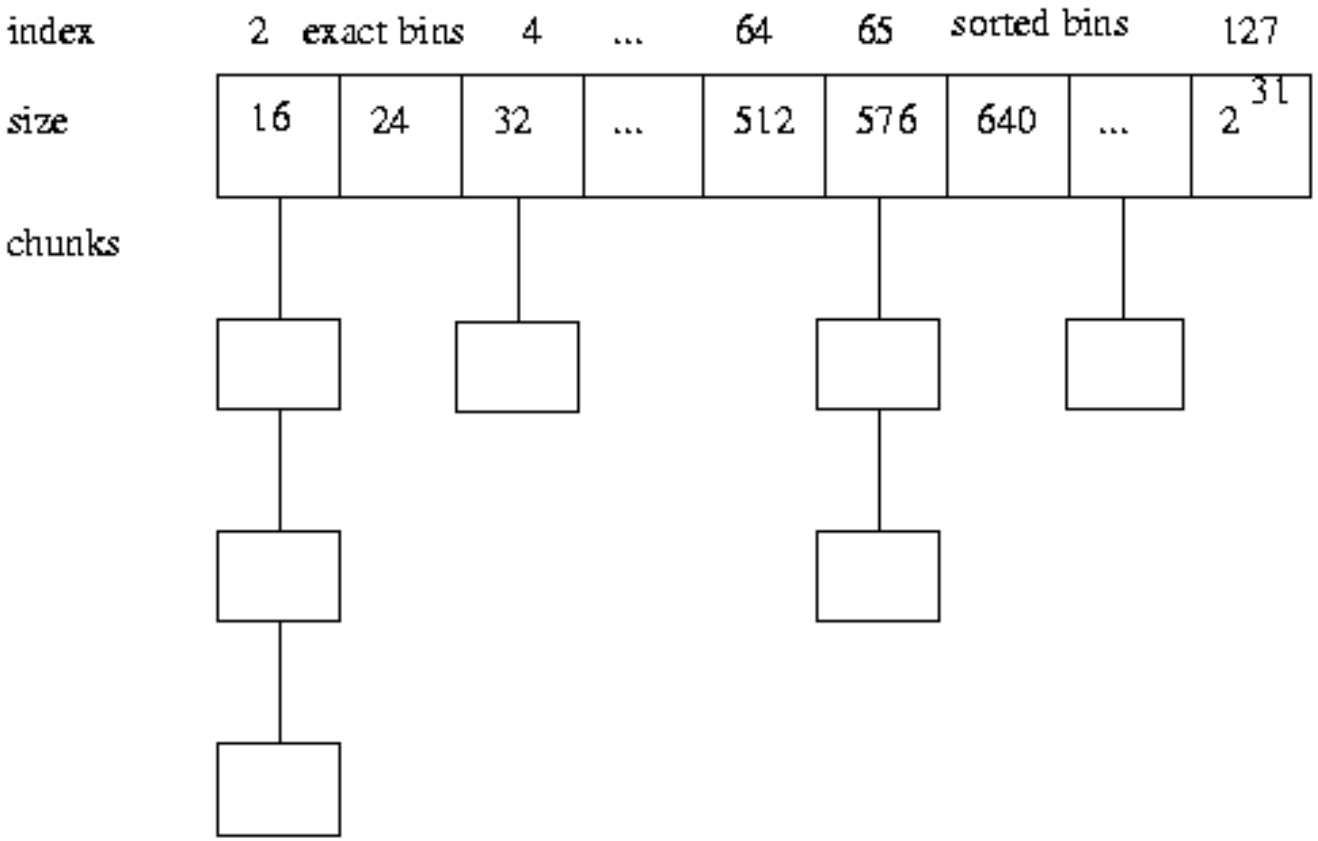
Process Address Space in Linux



How malloc() works?

- malloc() is a library call, not system call like sbrk()/brk()
- But it calls either sbrk()/brk(), or mmap() system calls to get a big contiguous chunk in virtual address space
- Steps involved in serving malloc(x) request
  - It checks in its allocated heap for free block to satisfy request
    - Doug Lea's dynamic memory allocator (Dlmalloc)
    - 128 bins to keep track of free blocks (fixed size and range based)
  - If not, it calls sbrk()/brk(), or mmap() to extend boundary of Heap segment in the process's virtual address space (*struct mm_struct*)
  - Note that no page frame is allocated by the end of malloc()
  - Only on page fault in accessing newly allocated memory, page frame is allocated

128 Bins in dlmalloc()



http://g.oswego.edu/dl/html/malloc.html

END of PART-II

Dynamic Memory Management

PART-III

- Page size selection
- TLB Reach
- Inverted Page Tables
- Memory-mapped Files
- Memory-mapped I/O
- Program Structure

Other Considerations: Page Size

- Concern 1: Page Table Size
  - Demands large page sizes
- Concern 2: Fragmentation
  - Demands small page sizes
- Concern 3: I/O Overhead
  - Seek, Latency, and transfer times for Disk I/O
  - Overhead is reduced with large page sizes
- Concern 4: Locality
  - Locality is better with small page sizes
- Concern 5: Page fault rate
  - Page size of 1 Byte?
- Conclusion??!!
  - Historically page sizes are increasing…

Other Considerations: TLB Reach

- **TLB Reach** - The amount of memory accessible from the TLB.
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB. Otherwise there is a high degree of page faults.

Increasing the Size of the TLB reach

- **Increase the Page Size**. This may lead to an increase in fragmentation as not all applications require a large page size.

- **Provide Multiple Page Sizes**. This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation.
  - Requires OS to manage TLB

Other Considerations: Inverted Page Table

- 1 entry per page frame
- Indexed by (PID, Page-No)
- But, it lacks info needed to serve page faults
- So, requires an external page table
- Do external page tables negate the utility of inverted page tables?

Other Considerations: Memory-Mapped Files

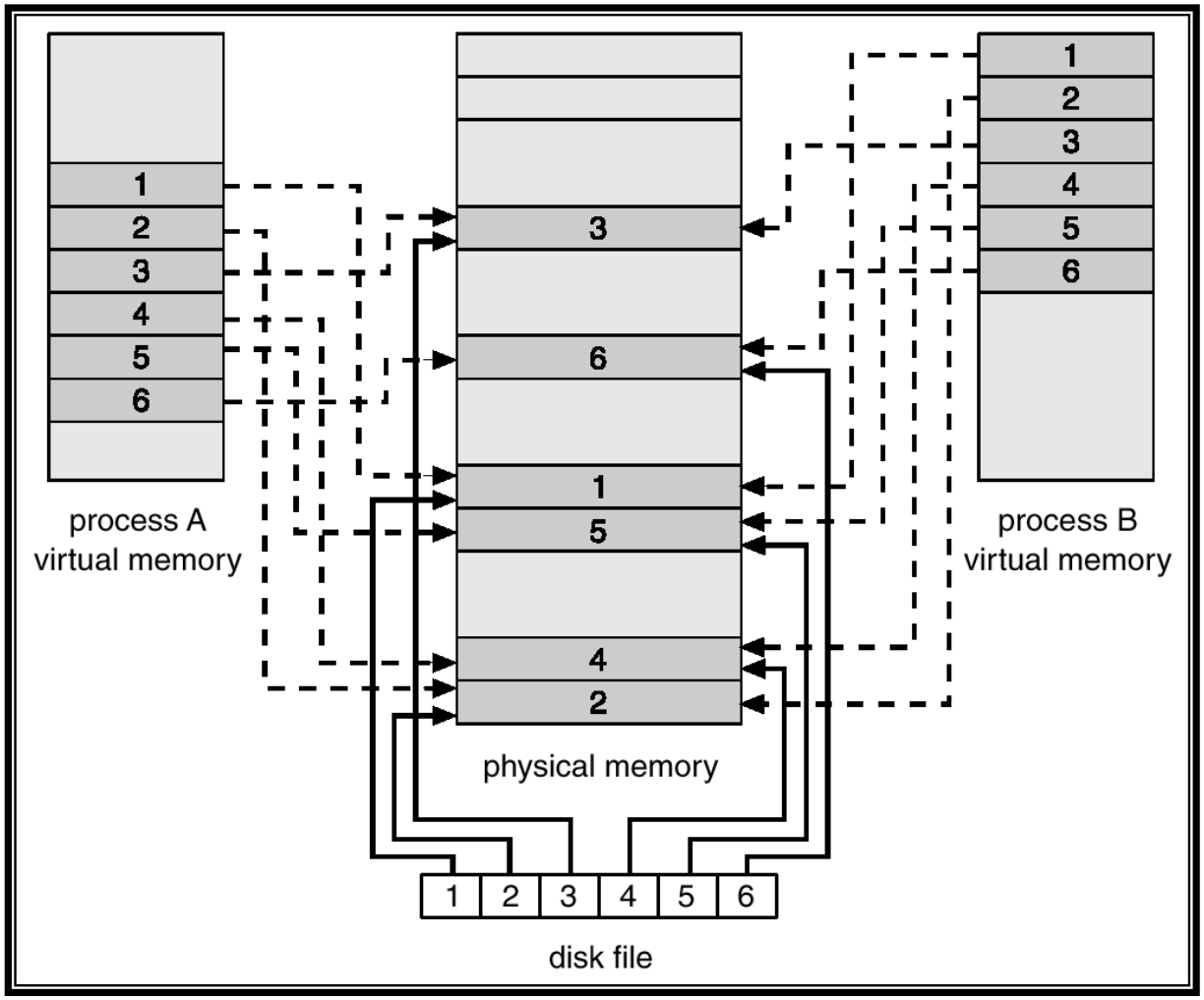
- Memory-mapped file I/O allows file I/O to be treated as routine memory access by *mapping* a disk block to a page in memory.

- A file is initially read using demand paging.
- A page-sized portion of the file is read from the file system into a physical page.
- Subsequent reads/writes to/from the file are treated as ordinary memory accesses, no system calls needed.

- Simplifies file access by treating file I/O through memory rather than **read()** **write()** system calls.

- Also allows several processes to map the same file allowing the pages in memory to be shared.

Memory Mapped Files

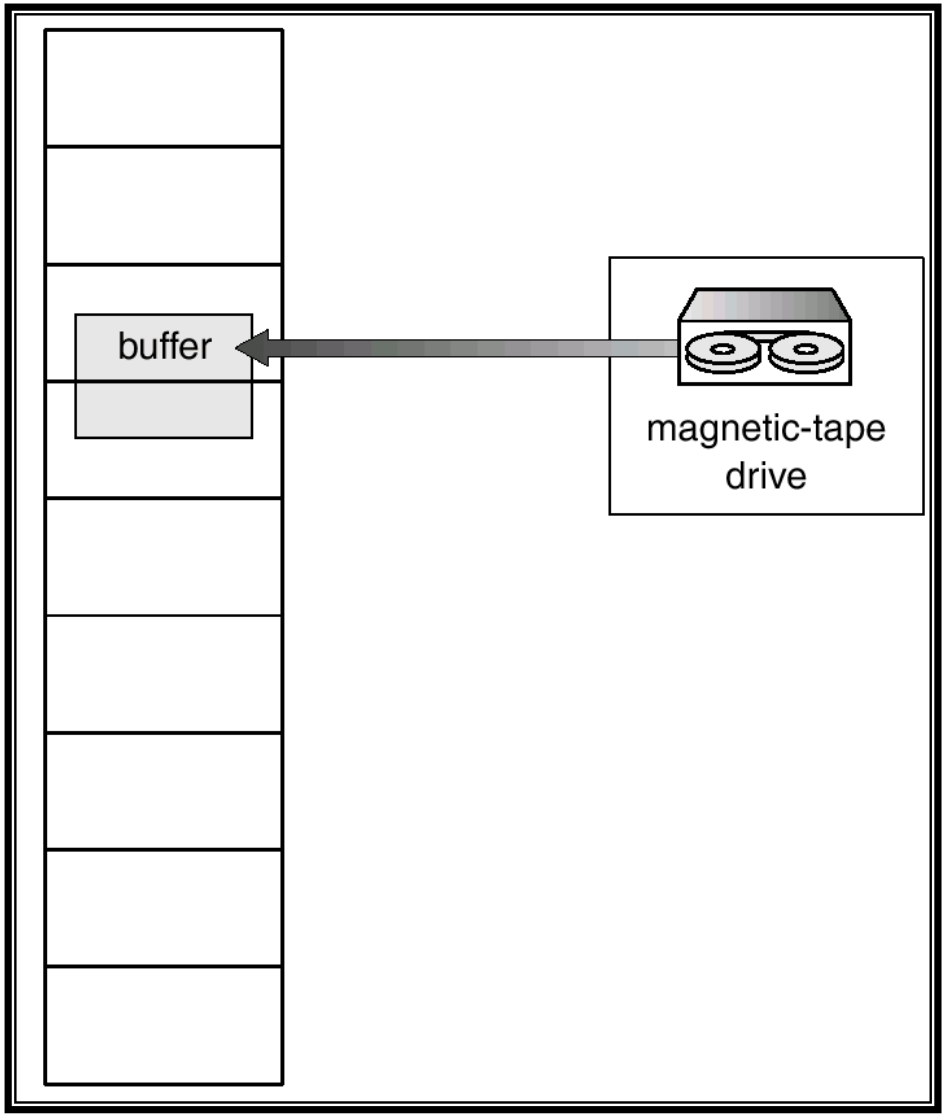


Other Considerations: Memory-mapped I/O

- **I/O Interlock** – Pages must sometimes be locked into memory.

- Consider I/O: Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm.

Reason Why Frames Used For I/O Must Be In Memory



Other Considerations: Program Structure

- `Int[128,128] data;`
- Each row is stored in one page
- 1 Page Frame is given to the process
- Program 1
  - ```
    for (i = 0; i < 128; i++)
        for (j = 0; j < 128; j++)
            data[i,j] = 0;
    ```
  - *128* page faults

- Program 2
  - ```
    for (j = 0; j <128; j++)
        for (i = 0; i < 128; i++)
            data[i,j] = 0;
    ```
  - 128 x 128 = *16,384* page faults!

Summary

- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process
- Working Set:
  - Set of pages touched by a process recently
  - Recently is Δ references or time units
- Dynamic memory allocation
  - Same problems as contiguous memory allocation
  - First-fit, Best-fit, Worst-fit, still suffer from fragmentation
  - Buddy scheme + Best-fit is simple strategy

Reading Assignment

- Chapter 9 from OSC by Galvin et al
- Dlmalloc page:
  http://gee.cs.oswego.edu/dl/html/malloc.html