Memory Management
Part 2

Computer Operating Systems
BLG 312E

2016-2017 Spring

Virtual Memory

- to run, a process must be in memory
  - Question: must the whole of the process be in memory?
- physical addresses are determined after a process is loaded onto the memory
  - physical addresses may be different during the whole lifetime of the process
- parts of a process don’t have to be placed at contiguous locations in memory

Virtual Memory

- unused parts are in secondary memory
- initially, a part of the process is loaded onto the main memory
  - resident set
- if the part that is being accessed is not in memory
  - page fault – interrupt occurs
  - process is blocked
  - the requested part is loaded onto memory
    - operating system generates I/O request
    - interrupt occurs when I/O is completed; waiting processes are awakened and become READY

Virtual Memory

- due to virtual memory, there can be more processes in READY mode
  - more efficient multi-programming
  - only necessary parts of process are in main memory
  - processes larger than the whole main memory can also be run
- paging/segmentation is used in implementation
  - requires hardware support

Virtual Memory

Questions to answer:

- how is space allocated on the main memory and secondary storage?
  - easier with paging
  - harder with segmentation due to unequal segment sizes
- what must be considered when moving pages/segments between main memory ⇔ secondary storage?
- if main memory is full, which page/segment should be removed to the secondary storage ?

Allocation of Memory for Unequal Sized Segments

- keep free spaces in a linked list in increasing order of their address values
- in each record of the linked list:
  - address of free space
  - size of free space
  - pointer to next free space
- add memory locations to list as they are freed
  - combine with previous and next records if possible
- de-fragmentation is useful

Virtual Memory

Questions to answer:

- how is space allocated on the main memory and secondary storage?
  - easier with paging
  - harder with segmentation due to unequal segment sizes
- what must be considered when moving pages/segments between main memory ⇔ secondary storage?
- if main memory is full, which page/segment should be removed to the secondary storage ?

Allocation of Memory for Unequal Sized Segments

- first-fit
  - starting from the beginning of the list, allocate the first free space whose size is greater than or equal to the required size
  - leftover spaces are again added to the list
- next-fit
  - start looking for the first appropriate free space starting from the location of memory space allocated in the previous request (not from the beginning of the list)
  - better to have a circular list
- best-fit
  - try to find the free space whose size fits the requested size best (minimum leftover free space)
  - for each time, go through the whole list
- worst-fit
  - opposite of best-fit
  - again go through the whole list for each request

Allocation of Memory for Unequal Sized Segments

- order the free spaces in increasing order of their sizes:
  - best fit = first fit
  - harder to combine neighbor free spaces
- or keep pointers to locations in the list of free spaces of different sizes
  - takes time to update the pointers

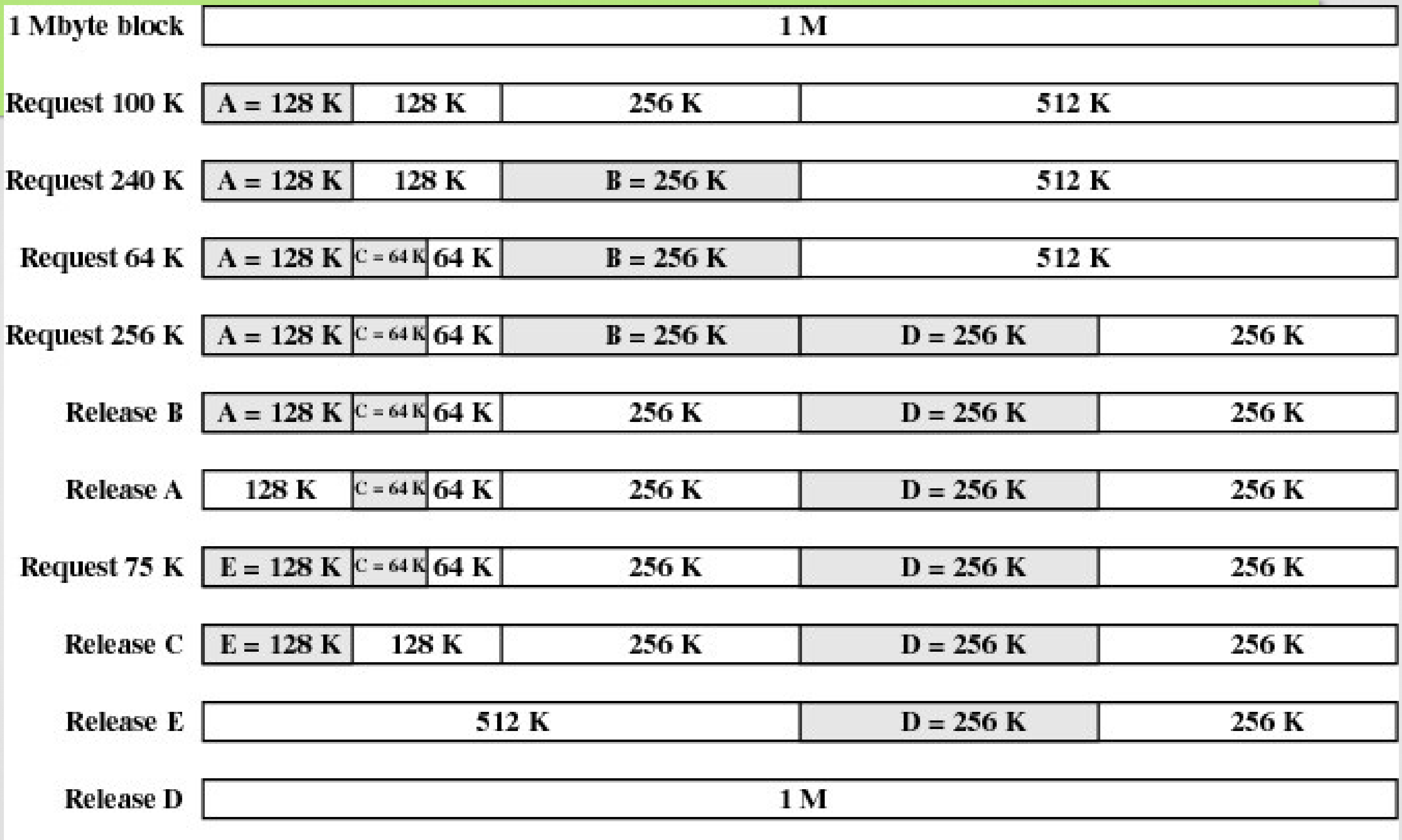
Allocation of Memory for Unequal Sized Segments

- “buddy” system
  - divide the whole memory into blocks of size 2^k
  - assume the whole memory size is 2^s
    - there are (s+1) linked lists
    - $2^0, 2^1, 2^2, ..., 2^s$
  - list(k): pointer to blocks of size 2^k (k=0,1,...,s)
  - initially list(s) points to the first location of the memory
    - all other lists are initially empty

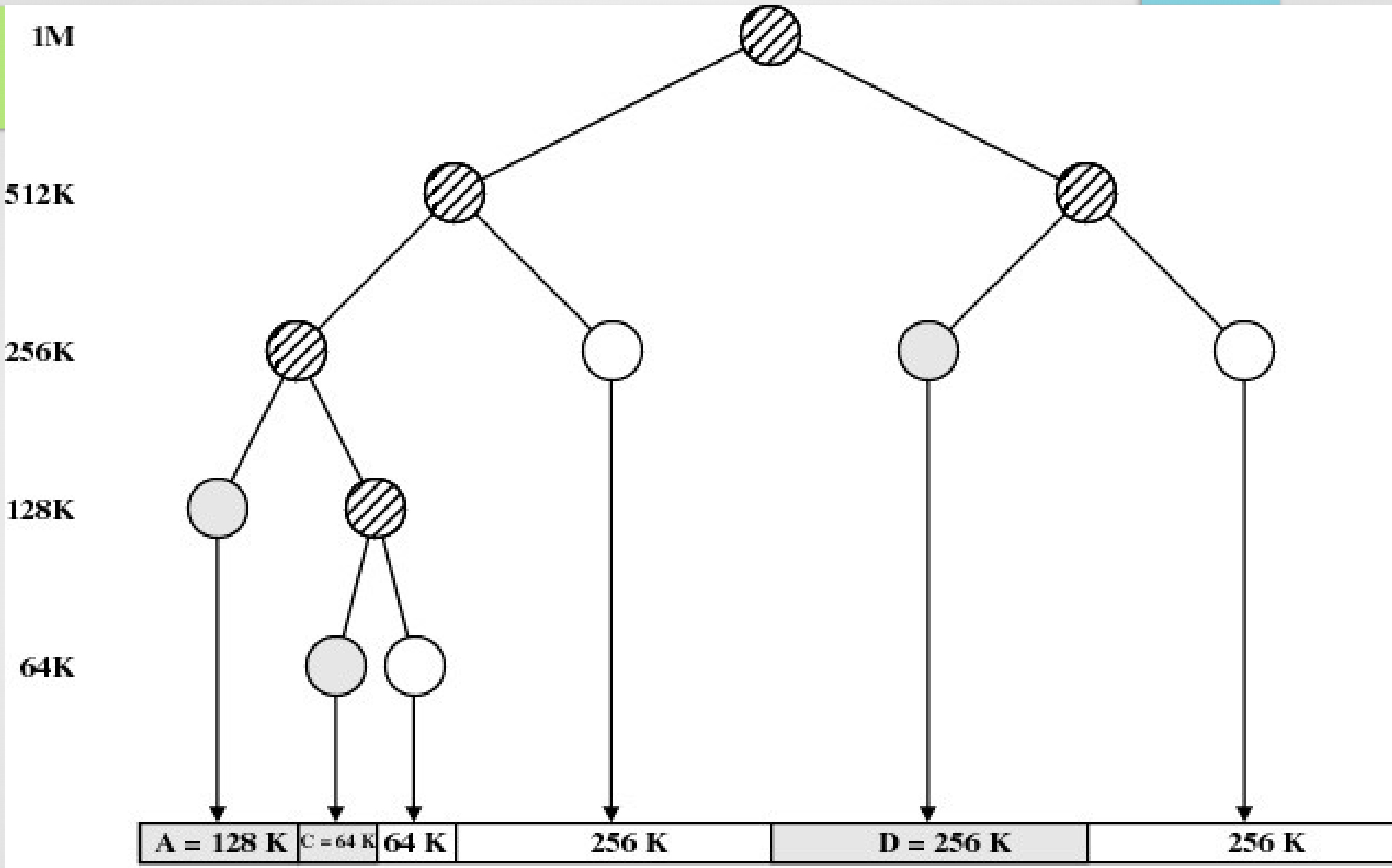
“Buddy” System

- assume a block of size 2^k is requested

  [$>2^{k-1}$ and $\leq 2^k$]
  - if list(k) is empty, try list(k+1)
    - if not empty, split the block into two
    - add one of the resulting blocks to liste(k)
    - use the other one for the request
  - if all lists are empty, the request cannot be satisfied
- when allocated blocks are retuned, they are added to appropriate lists
  - “buddy” blocks are combined



Buddy system example



Tree representation for the Buddy system

Fetching Techniques

- which criteria should be used when moving pages from secondary storage $\Rightarrow$ main memory ?
  - pre-paging
    - pages that will be accessed in the near future can be predicted
    - load pages onto memory before the actual access request
    - lesser page faults
    - high costs for wrong predictions
    - good for data pages for example
  - demand paging
    - bring pages to main memory only when they are accessed

Page Replacement

- if there is no available free space in the main memory, a page needs to be moved to the secondary storage
  - care must be given to possible page traffic
  - a page that is just removed from the main memory should not be accessed
    - “thrashing”: loss of time
  - main aim is to NOT remove USEFUL pages
    - pages that won’t be accessed in the near future can be removed
  - some operating system pages cannot be removed
    - frame locking is done through setting a bit
  - page selection can be at two levels :
    - local: choose from among the pages of the running process
    - global: choose from among all the pages

Page Replacement

- select randomly
  - easy to implement
  - USEFUL pages may be selected
- first in first out – FIFO
  - select page which has been in the main memory the longest
  - performance may be bad – the oldest page may not be the page that won’t be accessed in the near future
- BIFO (biased FIFO)
  - select from among the n_i pages of the i. process, use FIFO for the n_i pages
  - Different processes may have different number of pages in memory
  - n_i for each process may change over time

Page Replacement

- LRU (Least Recently Used)
  - high implementation cost, hardware support needed
  - keep a table of records for each page of the time that has passed since the last access to that page
  - at the end of each quantum, all entries are updated
    - clear the access time counters for the accessed pages
    - increment the access time counters for all other pages in the main memory (the ones that were not accessed)
    - when choosing a page to remove from memory, choose the one with the highest counter value (means the page has not been accessed for the longest time)

Page Replacement – Example

Assume there are 4 page frames in memory and the following pages are accessed in the given order. Initially all page frames are empty.

1 – 2 – 3 – 4 – 1 – 5 – 3 – 6 – 7 – 5 – 2

Give the contents of the page frames after each access. Mark the page faults.

(a) Use LRU

(b) Use FIFO

Page Replacement

- use pre-defined priorities
  - the compiler can determine which page should have higher priority (do not remove from memory)
  - the structure of the program can give this info
    - principle of locality): tendency of code and data accessed to remain in the same area
      - e.g. loops, data lists

Page Replacement

- use system defined priorities
  - possible to use the same priorities used in scheduling
  - in case of a page fault, a page belonging to the process with the lowest priority is selected to be removed from main memory
    - e.g. through LRU
  - if the last page of the process with the lowest priority is removed, it has to wait until space becomes available in the main memory
    - PROBLEM: there may be unused pages of higher priority processes in main memory

Page Replacement

- use hybrid techniques
  - some rules can be combined
  - in order of decreasing preference:
    - select a read-only-access page of a blocked process
    - select a read/write-access page of a blocked process
    - select an operating system page that has not been access during the previous ½ seconds
    - select a page of a process waiting for I/O
    - select a page of an active (running)