Slide Set 17: Memory/Heap Management

CS240: Data Structures and Data Management

Jérémy Barbay

Outline

Goals

- Minimize expensive memory accesses (disks)
  - We saw data structures and algorithms that address this B-Trees
  - What can the OS do to help?
- Support dynamic memory allocation
  - Manage the program **heap**
  - Treat the heap as an ADT

Principle of Locality

- Keep important things as near to the CPU as possible
- A program does not use all pieces of disk/memory equally
- **Spatial Locality**
  - Access something at position p
  - More likely to access something at position $p + \epsilon$
- **Temporal Locality**
  - Access something at time t
  - More likely to access it again at time $t + \epsilon$

Exploiting the Tendencies

- Break external memory into blocks
- Spatial Locality
  - Ask for any address in a block and the whole block is brought in
- Temporal Locality
  - **Virtual Memory**
  - Keep a directory of all external blocks
  - Bring a block into memory only when accessed
  - Flag which blocks are in disk cache

Block Replacement

- What if the disk cache fills?
- Program accesses memory in block B

Access(B)
  if *B in disk cache* **then**
    Perform access
  else if *there exists free block F in cache* **then**
    Fetch B into F
    Perform access
  else
    Evict a block E from cache
    Fetch B into E
    Perform access
  end if

Example

- Suppose 3 internal blocks in cache and 8 external blocks
- Keep a directory for internal and external blocks
- Initially empty cache
- Access blocks 3, 6, 3, 1

External

0	1	2	3	4	5	6	7
–	–	–	–	–	–	–	–
–	–	–	0	–	–	–	–
–	–	–	0	–	–	1	–
–	–	–	0	–	–	1	–
–	2	–	0	–	–	1	–

Internal

0	1	2
–	–	–
3	–	–
3	6	–
3	6	–
3	6	1

Eviction Policies

- Suppose we now access block 0
- How do we determine which block to evict?
- Goal is to minimize total number of disk fetches
- *MA* (Memory Access) – Time to perform a memory access to block B
- *BR* (Block Replacement) – Time to determine block to evict and perform replacement

Random

- Randomly select an internal block to evict
- Example – 1, 2, 5, 4, 5, 3, 2, 3 (generator gives 1, 0, 0, 2, 1)

External

0	1	2	3	4	5	6	7
–	0	1	–	–	2	–	–

Internal

0	1	2
1	2	5

- *MA* – $O(1)$
- *BR* – $O(1)$
- Space – 0
- Worst case scenario – Request the element you just removed.

FIFO

- Evict block that has been there the longest
- Example – 1, 2, 5, 4, 1, 5, 2, 5

External

0	1	2	3	4	5	6	7
–	0	1	–	–	2	–	–

Internal

0	1	2
1	2	5

- *MA* – $O(1)$
- *BR* – $O(1)$ (with a cyclic order)
- Space – $O(1)$ (to remember where you are in the cycle)
- Worst case scenario – $(1, 2, 3, 4)^*$ if $Q = 3$

LFU

- Evict the least frequently used block in the cache
- Need to keep a count with each internal item
  - Keep a PQ of the counts
- If you access an internal item, update count
- *MA* – $O(1)$ (with a trick to normalize frequencies)
- *BR* – $O(\lg m)$
- Space – $O(n)$ (overhead for counting)
- Worst case scenario – $(1, 2, 3, 4)^*$
- Comments – Difference between FIFO and LFU?
  LFU (but not FIFO) updates its data counter when a block in the cache is queried.

Example

- Access blocks 4, 1, 5, 5

External

0	1	2	3	4	5	6	7
–	1	–	–	0	2	–	–

Internal

0	1	2
4	1	5

- Access block 4

–	1	–	–	0	2	–	–

4	1	5

- Access block 2

LRU

- Evict the least recently used block in the cache
- Whatever item was accessed furthest in the past
- Need to keep a queue of internal items
  - Front of queue is the least recently accessed
- If you access an internal item, move it to back of queue
- MA – $O(1)$
- BR – $O(n)$
- Space – $O(m)$ or $O(n)$, depends of the variant.
- Worst case scenario – $(1, 2, 3, 4)^*$ for $Q = 3$.
- Comments – This is used a lot at the lowest level.

Example

- Access blocks 4, 1, 5, 1, 4, 2

External

0	1	2	3	4	5	6	7
–	1	–	–	0	2	–	–

Internal

0	1	2
4	1	5

Summary

- Various ways to manage Memory.
- It is always a trade-off between speed and efficiency.
- It has a big impact on the real performance.

Reference:

- Goodrich and Tamassia, 646–648 and 668-669
- Aho and Ullman, 146–159
- most OS books.

Outline

Memory Management

- Principle of Locality
- Block Replacement Strategies

Heap Management

- Fixed Sized Blocks
- Variable Sized Blocks
- Buddy Block

Final Review

Operators

- Treat the heap as an ADT
- Two operators:
  - $p \leftarrow$ `Allocate(` *size* `)`
  - `Free(` p `)`

Issues

- Fixed or variable sized blocks?
- If fixed, small or large sized blocks?
- User or OS freeing memory blocks?
  Explicit vs Implicit freeing of memory.
- Time or memory?
- Can blocks reference each other?

Fixed Sized Blocks

- Treat memory as an array of blocks
- Some are in use, some are on a free list
- User can **only** ask for one block at a time
- Allocation – grab any block from the free list
- Free – when no longer needed, return to free list

Fixed Sized Blocks

Issues

- **Internal Fragmentation**
  If user does not need all space in block, the extra space is wasted
- How do we know when block is no longer needed?
- **Explicit Free**
  Simplest to implement

Fixed Sized Blocks

Implicit Freeing

- **Implicit Free** with reference counts
  - Update number of references to an allocated block
  - Islands of garbage may result
- **Implicit Free** with garbage collection
  - Periodically clean memory
  - 2-pass Mark and Sweep algorithm

Variable Sized Blocks

- Allocate a contiguous chunk of memory
- All the issues with fixed sized blocks and more
- **External Fragmentation**
  - Division of free space leaves no chunk large enough
  - May require expensive **Memory Compaction**
- **Internal Fragmentation**
  - To simplify allocation, OS may give a larger block than was requested
  - For example, round up to a multiple of a minimum block size

Variable Sized Blocks

Implementation

- Maintain an ordered list of each chunk of memory
- `Allocate`:
  - Search for a free hole that is “big enough”
  - Mark node as allocated, updating neighbour
- `Free`:
  - Mark node as free
  - Merge with any free neighbours

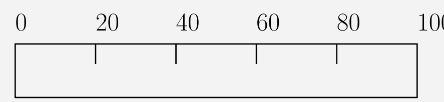
Variable Sized Blocks

Selection Policies

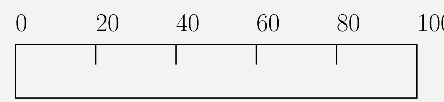
- Consider the following sequence on a RAM of size 100:

```
A <- Allocate( 40 )
B <- Allocate( 40 )
Free( A )
C <- Allocate( 10 )
```

- Various strategies for choosing free block
- First Fit



- Best Fit

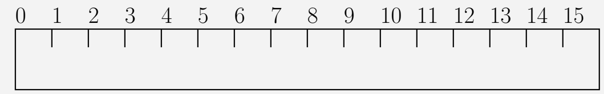


Variable Sized Blocks
Comments

- All strategies can outperform the others in certain cases
- First-fit and Alternating are simplest to implement
- Worst-fit needs a priority queue
- Best-fit needs a more complicated priority queue

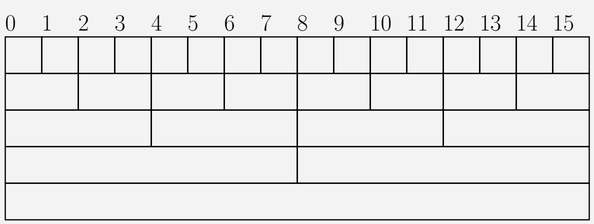
Buddy Block

- Trade less external fragmentation for internal fragmentation
- OS gives block sizes that are powers of two
  - Suppose heap has size $N = 2^m$
  - Valid allocation sizes are 2^0, 2^1, ... , 2^m blocks
  - Maintain $m + 1$ free lists
  - Initially all but the 2^m list is empty



Buddy Block
Implementation

- Only allowed to merge free blocks that are "buddies"
- Similar to extendible hashing
- Each block of size 2^k starts at some multiple of 2^k
- Each block of size 2^k has a buddy of the same size
- The two buddies are within the same block of size 2^{k+1}



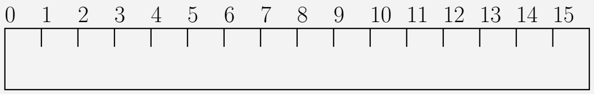
Buddy Block
Operators

- $p \leftarrow$ `Allocate(` n `)`
  - Round n up to the nearest power of two: 2^i
  - If no free block of size 2^i, split a block of size 2^{i+1}
  - This split may require another split ...
- `Free(` p `)`
  - Add p's block to the free list of size 2^i
  - If buddy is in the free list, merge and add to free list of size 2^{i+1}
  - Again, this may trigger another merge ...

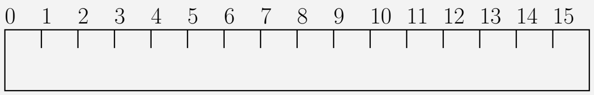
Buddy Block

Example

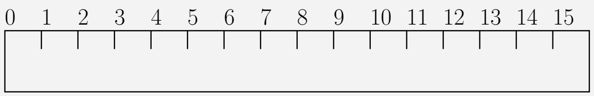
- `A <- Allocate( 3 )`



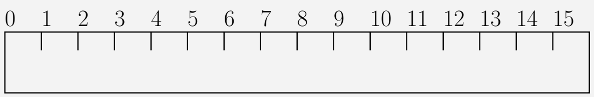
- `B <- Allocate( 3 )`



- `C <- Allocate( 1 )`



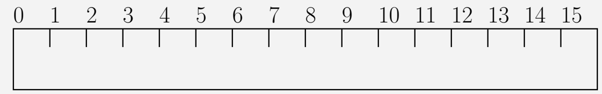
- `D <- Allocate( 2 )`



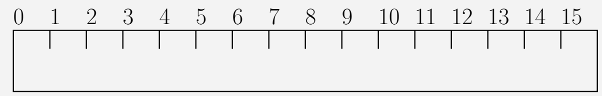
Buddy Block

Example

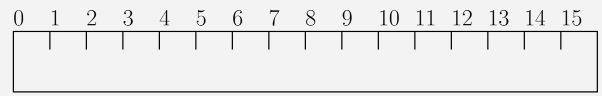
- `Free( B )`



- `Free( C )`



- `Free( D )`



Buddy Block

Comments

- $\Theta(1)$ to find buddy
  - If free list is implemented with an array
- $\Theta(m)$ worst case for both Free and Allocate
- Internal fragmentation wastes approximately 30%
- External fragmentation still possible

Summary

- Two extremes and one intermediate solution:
  Fixed, Variable Sized Blocks, and Buddy Blocks.
- As for memory management, big impact on performance.

Reference: none in GT nor CLRS.

Outline

Final Review

The final can/should cover the following concepts:

- Algorithmic Analysis
- ADTs
  - LIFO, FIFO
  - Graphs
  - Trees, ordinal and binary cardinal
  - Priority Queues and Heaps
  - Unsorted Dictionaries
  - Sorted Dictionaries
  - Valued Dictionaries
- Applications
  - Relational Databases and SQL
  - Compression Algorithms
  - Pattern Matching Algorithms
  - Heap and Memory Management.

1. Algorithmic Analysis

- Asymptotics
- Sorting algorithms
- Recursivity (and its analysis)
- Worst and Average Case complexity
- Randomized algorithms
- Lower bounds in the comparison model

2. ADTs

- LIFO, FIFO
- Graphs
- Trees, ordinal and binary cardinal
- Priority Queues and Heaps
- Unsorted Dictionaries
  - with Array
  - with Lists
- Sorted Dictionaries
  - Array
  - Lists
  - BST
  - AVL
  - (2,4) Trees
  - B Trees
- Valued Dictionaries
  - Hash Table

3. Applications

- Relational Databases and SQL
- Compression Algorithms
- Pattern Matching Algorithms
- Heap and Memory Management.

Advices

- Make sure that you know how to do the assignments, and the midterm.
- Do the assignments from previous years.
- Try to remember Why and How we do things.

Good Luck...