

CS206 Concurrency and Parallelism

Martin Odersky and Sanidhya Kashyap

EPFL, Spring 2022

- Garbage collection
- Understanding memory
- Manual memory reclamation

- Responsible for heap management
- A process of automatically freeing objects when they are no longer referenced by the program

- Responsible for heap management
- A process of automatically freeing objects when they are no longer referenced by the program
- Allocate memory from the OS and returns back
- Delivers memory to the application when required
- Determines which memory is still in use
- Reclaims the unused memory

Why do we need garbage collection?

Why do we need garbage collection?

- Free unreferenced memory
 - Leads to use-after-free bugs
 - Google/Apple pay you thousands of dollars for finding them

Why do we need garbage collection?

- Free unreferenced memory
 - Leads to use-after-free bugs
 - Google/Apple pay you thousands of dollars for finding them
- Allocates memory in an efficient manner
- Relieves programmers from manually freeing the memory
- Reset memory during deallocation and provide clean memory during allocation
- Ensures memory safety:
 - An object cannot use the memory for itself that has been allocated for another object

Why do we need garbage collection?

- Free unreferenced memory
 - Leads to use-after-free bugs
 - Google/Apple pay you thousands of dollars for finding them
- Allocates memory in an efficient manner
- Relieves programmers from manually freeing the memory
- Reset memory during deallocation and provide clean memory during allocation
- Ensures memory safety:
 - An object cannot use the memory for itself that has been allocated for another object
- Helps in ensuring program integrity

- OS provides the illusion of separate infinite (almost) resources
- Achieves through CPU and memory virtualization
- OS provides an **address space** abstraction for mapping address (memory location) to data
 - Static: code and global variables
 - Dynamic: stack, heap

Q: Why do we need dynamic memory?

- OS provides the illusion of separate infinite (almost) resources
- Achieves through CPU and memory virtualization
- OS provides an **address space** abstraction for mapping address (memory location) to data
 - Static: code and global variables
 - Dynamic: stack, heap

Q: Why do we need dynamic memory?

- Memory use is dependent on the task
- Input size is not known at the compile time
- Pre-allocation can be wasteful
- Recursive functions

- Local variables get allocated when executing a function
- Memory allocation size is known at compile time
- Allocation happens when a function is called and de-allocated after the function call is over
 - Called temporary memory allocation
- Data accessed is only used by the current task, no concurrency

- Local variables get allocated when executing a function
- Memory allocation size is known at compile time
- Allocation happens when a function is called and de-allocated after the function call is over
 - Called temporary memory allocation
- Data accessed is only used by the current task, no concurrency
- Implementation: bump or decrement a pointer
 - De-allocation must be in reverse order

A set of randomly allocated memory objects with statically unknown size and statically unknown allocation patterns. The size and lifetime of each allocated object is unknown

- Idea: abstract heap into a list of free blocks
 - Keep track of free space, program handles allocated space
 - Keep a list of all available memory objects and their size

- Idea: abstract heap into a list of free blocks
 - Keep track of free space, program handles allocated space
 - Keep a list of all available memory objects and their size
- Implementation:
 - Memory allocation: take a free block, split, put the remaining back on the free list
 - Memory deallocation: add the freed block back to the free list

Q: What is the issue with this approach?

- Allocation: Find a fitting object (first, best, worst fit)
 - first fit: find the first object in the list and split it
 - best fit: find the object that is closest to the size
 - worst fit: find the largest object and split it

- Allocation: Find a fitting object (first, best, worst fit)
 - first fit: find the first object in the list and split it
 - best fit: find the object that is closest to the size
 - worst fit: find the largest object and split it
- De-allocation: merge adjacent blocks
 - If the adjacent region is free, merge the two blocks

- On starting a process:
 - Runtime reserves a contiguous memory from the OS
 - Reserved address space is called managed heap
- Managed heap maintains a pointer for allocating the next object
- Initially, it points to the heap's base address

- On starting a process:
 - Runtime reserves a contiguous memory from the OS
 - Reserved address space is called managed heap
- Managed heap maintains a pointer for allocating the next object
- Initially, it points to the heap's base address

How will allocation happen when you run a program?

- Memory allocation is faster as objects can be allocated contiguously, similar to stack

- GC determines best time to collect the garbage based on the allocations
- Releases memory for object when the application is not using the object
- GC figures out as follows:
 - GC maintains a set of application's roots
 - Root are storage locations, referring to objects
 - Roots include static field, local variables, CPU registers, GC specific data
 - List of active roots is maintained by the runtime and GC can access it

- GC first constructs a graph of reachable objects by assuming that all objects are garbage

- GC first constructs a graph of reachable objects by assuming that all objects are garbage

- GC walks through the heap linearly and shifts non garbage objects to remove all gaps

- Java provides `finalize()` method to clean other resources that are part of that object.
 - Ex: Outside resources, such as file descriptor
- On detecting the object to be a garbage, GC calls the `finalize` method to clean up all the resources as well as the associated objects

- Reference counting
- Mark and sweep
- Many more. . .

- Maintain a count for each object
- The count increase with increasing reference
- When count reaches 0, the object is eligible for reclamation

- Maintain a count for each object
- The count increase with increasing reference
- When count reaches 0, the object is eligible for reclamation

Q: Do you need an atomic count or a simple counter variable?

- Maintain a count for each object
- The count increase with increasing reference
- When count reaches 0, the object is eligible for reclamation

Q: Do you need an atomic count or a simple counter variable?

- Advantages:
 - Easiest to implement
 - Immediate memory re-use
- Disadvantages:
 - Cannot detect cycles, eg. doubly linked-list
 - Overhead of atomic counting

- Divided into two steps: mark and sweep phase
- Mark phase:
 - Traverse all roots
 - Marks all objects if the object is in use by setting a bit
 - Do it recursively for all objects
- Sweep phase:
 - Scan the heap from start until end
 - If the object is marked, then unmark it
 - Else free the object

- Divided into two steps: mark and sweep phase
- Mark phase:
 - Traverse all roots
 - Marks all objects if the object is in use by setting a bit
 - Do it recursively for all objects
- Sweep phase:
 - Scan the heap from start until end
 - If the object is marked, then unmark it
 - Else free the object
- This approach requires pausing the whole application
 - Stop the world model
- Advantages: Handles cycles and more space efficient
- Disadvantages: Long pauses are possible

- Concurrent data structures mostly use lock-free operations.
- These algorithms assume that threads operate on any object at any time
- Operations of a concurrent data structure:
 - Add/remove
 - Traverse the data structure

NOTE: A thread removing the object, makes the object unreachable and prohibits the creation of new references to that object.

- Concurrent data structures mostly use lock-free operations.
- These algorithms assume that threads operate on any object at any time
- Operations of a concurrent data structure:
 - Add/remove
 - Traverse the data structure

NOTE: A thread removing the object, makes the object unreachable and prohibits the creation of new references to that object.

Q. How do we free the memory in this case?

- Let's understand it from a linked list perspective

- Ensure that reading of data is consistent
- Several approaches:
 - Quiescent-state-based reclamation (QSBR)
 - Epoch-based reclamation (EBR)
 - Hazard pointers (HP)

Idea: Identify a quiescent state in each thread. Once all threads achieve their quiescent state, then the system has finished the grace period. After finishing the grace period, it is safe to invoke the reclamation of the element.

- Opposite of a critical section
- In a quiescent state, a thread does not hold any reference to any shared object.
- After a quiescent state, it is guaranteed that no critical section will hold that object.

Idea: Associate a number of single-writer multi-reader pointers for each process. Such pointers are called hazard pointers that indicate which nodes they might be able to access without further validation. If a node is marked by hazard pointer, it is unsafe for reclamation.

High-level pseudocode:

- 1 Read a reference of the object O
- 2 Assign a hazard pointer to O
- 3 Check if O is still valid
- 4 Access the object O
- 5 Release the reference to O

- Each thread individually maintain a set of pointers
 - The thread can write to it while others will read it
- The thread will first add the reference of the node in its list before accessing the dynamic list.
- Each thread also maintain a list of nodes it has deleted and wants to free
- When the length of the list reaches a threshold, the thread scans hazard pointer list of all other threads to see if its safe to free

