Robust Memory Management Schemes

Prepared by :

Fadi Sbahi & Ali Bsoul

Supervised By:

Dr. Lo'ai Tawalbeh

Jordan University of Science and Technology

Robust Memory Management Schemes

- Introduction.
- Memory Management
  - Allocation
  - Recycling
- Memory Management Problems
- Allocation techniques
  - First fit
  - Buddy system
- Recycling
  - Manual Memory Management.
  - Automatic Memory Management
    - Tracing
    - Counting
- Summary

Introduction

- Embedded and real-time systems often only have *limited resources* (time and space) and these must be carefully managed.
- Nowhere this is more apparent than in the area of *memory management*.
- Embedded systems usually have a limited amount of memory available.
- It may be necessary to control how this memory is allocated so that it can be reused effectively.

Memory Management

Memory management can be divided into three areas:

1. Memory management hardware (MMUs,RAM)
2. Operating system memory management (virtual memory,protection)
3. Application memory management

Memory management hardware

- Electronic devices(RAM, MMUs (memory management nits),caches, disks, and processor registers)

Operating System Memory Management

- Memory must be allocated to user programs
- Memory reused by other programs when it is no longer required.

Application Memory Management

- Supplying the memory needed for a program's objects and data structures
- Recycling that memory for reuse when it is no longer required.

Combine two related tasks:

- Allocation
- Recycling

Memory Management Constraints

- CPU overhead
  - The additional time taken by the memory manager while the program is running
  - Interactive pause times
- How much delay an interactive user observes
  - Memory overhead
- How much space is wasted for administration, rounding

Memory Management Problems

- Memory leak
- External fragmentation
- Poor locality of reference
- Inflexible design

Memory Management Problems

- Memory leak
  - Some programs continually allocate memory without ever giving it up and eventually run out of memory OOM. This condition is known as a memory leak.

Memory Management Problems

- External fragmentation
  - A poor allocator can do its job so badly that it can no longer give out big enough blocks despite having enough spare memory.
  - This is because the free memory can become split into many small blocks, separated by blocks still in use. This condition is known as external fragmentation.

Memory Management Problems

- Poor locality of reference

  successive memory accesses are faster if they are to nearby memory location, otherwise will cause performance problems.

Memory Management Problems

- Inflexible design
  - Any memory management solution tends to make *assumptions about the way in which the program is going to use memory*.
  - If these assumptions are wrong, then the memory manager may spend a lot more time doing bookkeeping work to keep up with what's happening.

Allocation

- It is the process of assigning blocks of memory on request.
- Typically the allocator receives memory from the system in a small number of large blocks that it must divide up to satisfy the requests for smaller blocks.

Allocation techniques

- First fit
- Buddy system

- These techniques can often be used in combination

First Fit

- The allocator keeps a list of free blocks (known as the free list)
- On receiving a request for memory, scans along the list for the first block that is large enough to satisfy the request

First Fit

- If the chosen block is significantly larger than that requested, then it is usually split, and the remainder added to the list as another free block.
- The first fit algorithm performs reasonably well, as it ensures that allocations are quick.

Buddy System

- The allocator will only allocate blocks of certain sizes
- has many free lists, one for each permitted size
- The permitted sizes are usually either powers of two, or form a Fibonacci sequence
- Any block except the smallest can be divided into two smaller blocks of permitted sizes
- When the allocator receives a request for memory, it rounds the requested size up to a permitted size

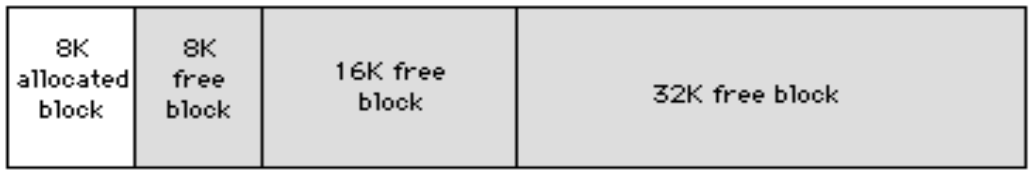
Buddy System

- returns the first block from that size's free list.
- If the free list for that size is empty, the allocator splits a block from a larger size and returns one of the pieces, adding the other to the appropriate free list.

Buddy System

A binary buddy heap before allocation

A binary buddy heap after allocating a 8 kB block



A binary buddy heap after allocating a 10 kB block and the 6 kB wasted because of rounding up

Buddy System

- When blocks are recycled, there may be some attempt to merge adjacent blocks into ones of a larger permitted size .
- To make this easier, the free lists may be stored in order of address.
- Advantage :

coalescence is cheap because the "buddy" of any free block can be calculated from its address.

Recycling

There are two approaches

- Manual memory management
  where the programmer must decide when memory can be reused.
- Automatic memory management
  where the memory manager must be able to work it out.

I- Manual Memory Management

- The programmer has direct control over memory.
- Usually this is by explicit calls functions (for example free in C).
- The memory manager does not recycle any memory without an instruction.

I- Manual Memory Management

Advantages :

- It can be easier for the programmer to understand exactly what is going on.
- Some manual memory managers perform better when there is a shortage of memory.

I- Manual Memory Management

Disadvantages :

- The programmer must write a lot of code to do repetitive bookkeeping of memory.
- Memory management must form a significant part of any module interface.
- Manual memory management typically requires more memory overhead per object.
- Memory management bugs are common.

II- Automatic Memory Management

- Automatically recycles memory that a program would not use again.
- Automatic memory managers (often known as garbage collectors) usually do their job by recycling blocks that are unreachable from the program variables.

II- Automatic Memory Management

The Advantages :

- The programmer is freed to work on the actual problem.
- There are fewer memory management bugs
- Memory management is often more efficient.

The Disadvantages:

- Memory may be retained because it is reachable, but won't be used again.

II- Automatic Memory Management

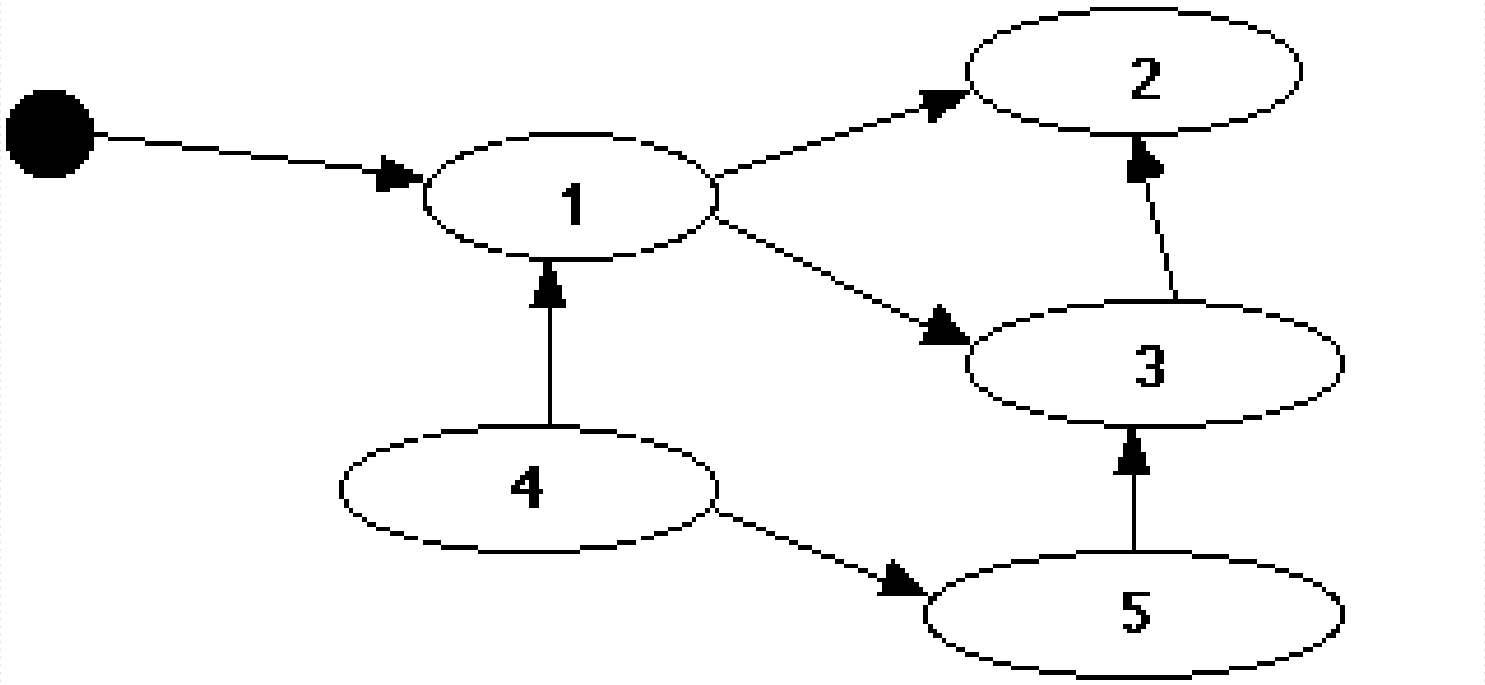
Garbage collection techniques can be split into two broad categories:

- Tracing
  - Mark-Sweep Collection
  - Copying Collection
  - Incremental Collection
  - Conservative Garbage Collection
- Reference Counting
  - Simple Reference Counting

Mark-Sweep Collection

- The collector first examines the program variables (root set).
- Any blocks of memory pointed to are added to a list of blocks to be examined.
- For each block on that list, it sets a flag (the mark) on the block to show that it is still required.

Mark-sweep collection



Mark-sweep Collection

Two drawbacks of simple mark-sweep collection are:

- It must scan the entire memory in use before any memory can be freed.
- It must run to completion or, if interrupted, start again from scratch

Mark-Sweep Collection

- It adds to the list any blocks pointed to by that block that have not yet been marked.
- All blocks that can be reached by the program are marked.
- In the second phase, the collector *sweeps* all allocated memory, searching for blocks that have not been marked. If it finds any, it returns them to the allocator for reuse.

Copying Collection

- A copying garbage collector may move allocated blocks around in memory and adjust any references to them to point to the new location.
- This is a very powerful technique and can be combined with many other types of garbage collection such as mark-sweep collection

Copying Collection

- The disadvantages :
  - Extra storage is required while both new and old copies of an object exist.
  - Copying data takes extra time (proportional to the amount of live data).
  - It is difficult to combine with conservative garbage collection because references cannot be confidently adjusted.

Incremental Collection

- Incremental collection allow garbage collection to be performed in a series of small steps while the program is never stopped for long.
- The program that uses and modifies the blocks is sometimes known as the mutator.

Incremental Collection

- While the collector is trying to determine which blocks of memory are reachable by the mutator, the mutator is busily allocating new blocks, modifying old blocks, and changing the set of blocks it is actually looking at.

Incremental collection

- Ensures that, whenever memory in crucial locations is accessed, a small amount of necessary bookkeeping is performed to keep the collector's data structures correct.

Conservative Garbage Collection

- Assumes that anything *might* be a pointer.
- It regards any data value that looks like a pointer to or into a block of allocated memory as preventing the recycling of that block.
- The collector does not know for certain which memory locations contain pointers.

Reference Counts

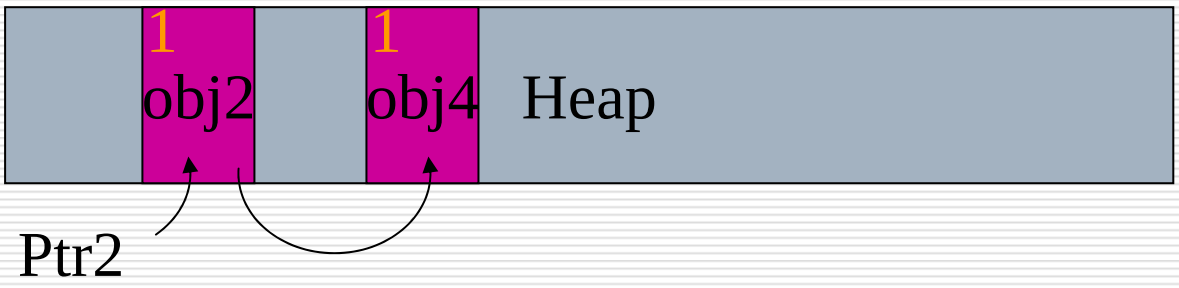
- A reference count is a count of how many references there are to a particular memory block from other blocks.
- It is used as the basis for some automatic recycling techniques that *do not rely on tracing*.

Simple Reference Counting

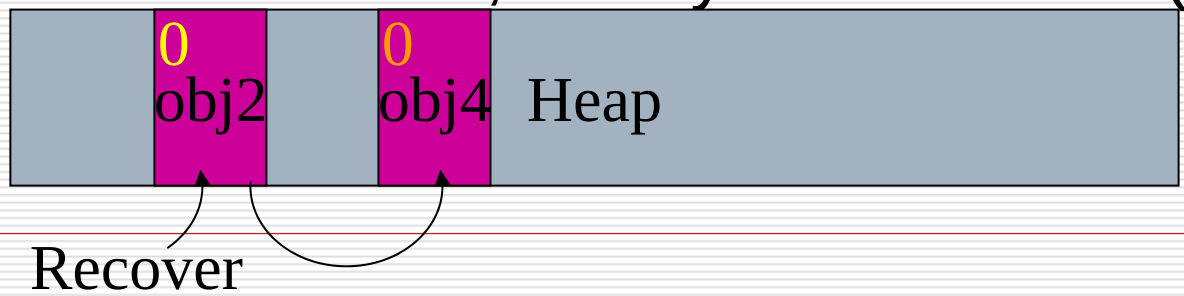
- A reference count is kept for each object.
- This count is incremented for each new reference, and is decremented if a reference is overwritten, or if the referring object is recycled.
- If a reference count falls to zero, then the object is no longer required and can be recycled.

Reference Counting

- How does it work?
  - Each object has a reference count.



  - When cnt=0, ready to be freed (walk



Simple Reference Counting

- It is frequently chosen as an automatic memory management strategy because it seems simple to implement.
- It is hard to implement efficiently because of the cost of updating the counts.

Simple Reference Counting

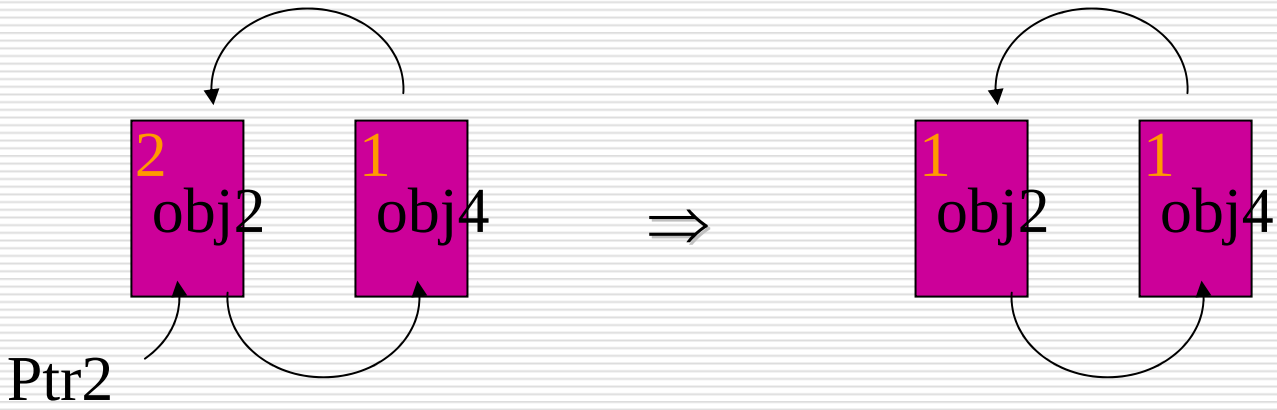
- It is also hard to implement reliably, because the standard technique cannot reclaim objects connected in a loop.
- In many cases, it is an inappropriate solution, and it would be preferable to use tracing garbage collection instead.

Simple Reference Counting

- Reference counting is most useful in situations.
  - Where it can be guaranteed that there will be no loops.
  - Where modifications to the reference structure are infrequent.
- Reference counting may be useful if it is important that objects are recycled *immediately*, such as in systems with tight memory constraints.

Simple Reference Counting

- Cycles cannot be recovered directly
  - Requires either manual intervention or 2nd recovery method.



Real-time and Garbage Collection

- Running the garbage collector may have a significant impact on the response time of a time-critical thread
- Consider a time-critical periodic thread which has had all its objects pre-allocated.

Real-time and Garbage Collection

- It may have a higher priority than a non time-critical thread and will not require any new memory, it may still be delayed when garbage collection has been initiated by an action of non time-critical thread .
- In this instance, it is not safe for the time-critical thread to execute until garbage collection has finished (particularly if memory compaction is taking place).

Summary

- The basic problem in managing memory is knowing when to keep the data it contains, and when to throw it away so that the memory can be reused.
- Most programmers wouldn't have to worry about memory management issues.

Summary

- There are many ways in which poor memory management practice can affect the robustness and speed of programs, both in manual and in automatic memory management.

References

- http://www.memorymanagement.org/articles/begin.html
- http://www.hanappe.org/Rapports/PeterHanappe99/ch1/node3.html
- http://java.sun.com/docs/books/realtime