Topics

• Heap allocation
• Manuel heap allocation
• Automatic memory reallocation (GC)
Limitations of Stack Frames

• A local variable of P cannot be stored in the activation record of P if its duration exceeds the duration of P

• Example: Dynamic allocation
  int * f() { return (int *) malloc(sizeof(int)); }

Currying Functions

```c
int (*)(int) f(int x)
{
    int g(int y)
    {
        return x + y;
    }
    return g ;
}

int (*h)() = f(3);
int (*j)() = f(4);

int z = h(5);
int w = j(7);
```
Program Runtime State

Code segment
Stack segment
Data Segment
Machine Registers

fixed heap
Data Allocation Methods

• Explicit deallocation
• Automatic deallocation
Explicit Deallocation

- Pascal, C, C++
- Two basic mechanisms
  - void * malloc(size_t size)
  - void free(void *ptr)
- Part of the language runtime
- Expensive
- Error prone
- Different implementations
Memory Structure used by malloc()/free()
Simple Implementation

SET the polymorphic chunk pointer First_chunk pointer TO
   Beginning of available memory;
SET the polymorphic chunk pointer One past available memory TO
   Beginning of available memory + Size of available memory;

SET First_chunk pointer .size TO Size of available memory;
SET First_chunk pointer .free TO True;

FUNCTION Malloc (Block size) RETURNING a polymorphic block pointer:
   SET Pointer TO Pointer to free block of size (Block size);
   IF Pointer /= Null pointer: RETURN Pointer;
   Coalesce free chunks;
   SET Pointer TO Pointer to free block of size (Block size);
   IF Pointer /= Null pointer: RETURN Pointer;

   RETURN Solution to out of memory condition (Block size);  
call gc

PROCEDURE Free (Block pointer):
   SET Chunk pointer TO Block pointer - Administration size;
   SET Chunk pointer .free TO True;
FUNCTION Pointer to free block of size (Block size)
    RETURNING a polymorphic block pointer:
    // Note that this is not a pure function
    SET Chunk pointer TO First chunk pointer;
    SET Requested chunk size TO Administration size + Block size;
    WHILE Chunk pointer /= One past available memory:
        IF Chunk pointer .free:
            IF Chunk pointer .size - Requested chunk size >= 0:
                // large enough chunk found:
                Split chunk (Chunk pointer, Requested chunk size);
                SET Chunk pointer .free TO False;
                RETURN Chunk pointer + Administration size;
                // try next chunk:
                SET Chunk pointer TO Chunk pointer + Chunk pointer .size;
        RETURN Null pointer;
PROCEDURE Split chunk (Chunk pointer, Requested chunk size):
    SET Left_over size TO Chunk pointer .size - Requested chunk size;
    IF Left_over size > Administration size:
        // there is a non-empty left-over chunk
        SET Chunk pointer .size TO Requested chunk size;
        SET Left_over chunk pointer TO
            Chunk pointer + Requested chunk size;
        SET Left_over chunk pointer .size TO Left_over size;
        SET Left_over chunk pointer .free TO True;
Coalescing Chunks

PROCEDURE Coalesce free chunks:
    SET Chunk pointer TO First_chunk pointer;

    WHILE Chunk pointer /= One past available memory:
        IF Chunk pointer .free:
            Coalesce with all following free chunks (Chunk pointer);
            SET Chunk pointer TO Chunk pointer + Chunk pointer .size;

    PROCEDURE Coalesce with all following free chunks (Chunk pointer):
        SET Next_chunk pointer TO Chunk pointer + Chunk pointer .size;
        WHILE Next_chunk pointer /= One past available memory
            AND Next_chunk pointer .free:
                // Coalesce them:
                SET Chunk pointer .size TO
                Chunk pointer .size + Next_chunk pointer .size;
                SET Next_chunk pointer TO Chunk pointer + Chunk pointer .size;
Fragmentation

- **External**
  - Too many small chunks

- **Internal**
  - A use of too big chunk without splitting the chunk

- Freelist may be implemented as an array of lists
Garbage Collection

ROOT SET

a
b
c
d
e
f

Stack + Registers

HEAP
What is garbage collection

- The runtime environment reuse chunks that were allocated but are not subsequently used
- garbage chunks
  - not live
- It is undecidable to find the garbage chunks:
  - Decidability of liveness
  - Decidability of type information
- conservative collection
  - every live chunk is identified
  - some garbage runtime chunk are not identified
- Find the reachable chunks via pointer chains
- Often done in the allocation function
typedef struct list  {struct list *link; int key} *List;
typedef struct tree {int key;
    struct tree *left:
    struct tree *right} *Tree;

foo() { List x = cons(NULL, 7);
            List y = cons(x, 9);
            x->link = y;
        }

void main() { Tree p, r; int q;
        foo();
        p = maketree(); r = p->right;
        q= r->key;
        showtree(r);}
typedef struct list {struct list *link; int key} *List;

typedef struct tree {int key;
    struct tree *left;
    struct tree *right} *Tree;

foo() {
    List x = cons(NULL, 7);
    List y = cons(x, 9);
    x->link = y;
}

void main() {
    Tree p, r; int q;
    foo();
    p = maketree();
    r = p->right;
    q= r->key;
    showtree(r);
typedef struct list  {struct list *link; int key} *List;
typedef struct tree {int key;
    struct tree *left:
    struct tree *right} *Tree;

foo() {    List x = create_list(NULL, 7);
    List y = create_list(x, 9);
    x->link = y;
}

void main() {
    Tree p, r; int q;
    foo();
    p = maketree(); r = p->right;
    q= r->key;
    showtree(r);}


Outline

• Why is it needed?
• Why is it taught?
• Reference Counts
• Mark-and-Sweep Collection
• Copying Collection
• Generational Collection
• Incremental Collection
• Interfaces to the Compiler

Tracing
A Pathological C Program

```c
a = malloc(...);

b = a;

free (a);

c = malloc (...);

if (b == c) printf(“unexpected equality”);
```
Garbage Collection vs. Explicit MemoryDeallocation

- Faster program development
- Less error prone
- Can lead to faster programs
  - Can improve locality of references
- Support very general programming styles, e.g. higher order and OO programming
- Standard in ML, Java, C#
- Supported in C and C++ via separate libraries
- May require more space
- Needs a large memory
- Can lead to long pauses
- Can change locality of references
- Effectiveness depends on programming language and style
- Hides documentation
- More trusted code
Interesting Aspects of Garbage Collection

- Data structures
- Non constant time costs
- Amortized algorithms
- Constant factors matter
- Interfaces between compilers and runtime environments
- Interfaces between compilers and virtual memory management
Reference Counts

- Maintain a counter per chunk
- The compiler generates code to update counter
- Constant overhead per instruction
- Cannot reclaim cyclic elements
Another Example
Another Example \((x \rightarrow b = \text{NULL})\)
Code for p := q

IF Points into the heap (q):
    Increment q .reference count;
IF Points into the heap (p):
    Decrease p .reference count;
    IF p .reference count = 0:
        Free recursively depending on reference counts (p);
SET p TO q;
Recursive Free

PROCEDURE Free recursively depending on reference counts(Pointer);
    WHILE Pointer /= No chunk:
        IF NOT Points into the heap (Pointer): RETURN;
        IF NOT Pointer .reference count = 0: RETURN;

        FOR EACH Index IN 1 .. Pointer .number of pointers - 1:
            Free recursively depending on reference counts
            (Pointer .pointer [Index]);

        SET Aux pointer TO Pointer;
        IF Pointer .number of pointers = 0:
            SET Pointer TO No chunk;
        ELSE Pointer .number of pointers > 0:
            SET Pointer TO
            Pointer .pointer [Pointer .number of pointers];
        Free chunk(Aux pointer);  // the actual freeing operation
Lazy Reference Counters

• Free one element
• Free more elements when required
• Constant time overhead
• But may require more space
Reference Counts (Summary)

- Fixed but big constant overhead
- Fragmentation
- Cyclic Data Structures
- Compiler optimizations can help
- Can delay updating reference counters from the stack
- Implemented in libraries and file systems
  - No language support
- But not currently popular
- Will it be popular for large heaps?
Mark-and-Sweep(Scan) Collection

- **Mark** the chunks reachable from the roots (stack, static variables and machine registers)
- **Sweep** the heap space by moving unreachable chunks to the freelist (Scan)
The Mark Phase

for each root v

DFS(v)

function DFS(x)

if x is a pointer and chunk x is not marked

mark x

for each reference field f_i of chunk x

DFS(x.f_i)
The Sweep Phase

\[
p := \text{first address in heap}
\]

while \( p < \text{last address in the heap} \)

\[
\text{if chunk } p \text{ is marked}
\]

\[
\text{unmark } p
\]

else let \( f_1 \) be the first pointer reference field in \( p \)

\[
p.f_1 := \text{freelist}
\]

\[
\text{freelist := } p
\]

\[
p := p + \text{size of chunk } p
\]
Sweep

freelist

p
q
r
37

12
left
right

15
left
right

7
link

37
left
right

59
left
right

20
left
right

9
link
Cost of GC

- The cost of a single garbage collection can be linear in the size of the store
  - may cause quadratic program slowdown
- Amortized cost
  - collection-time/storage reclaimed
  - Cost of one garbage collection
    - $c_1 R + c_2 H$
  - $H - R$ Reclaimed chunks
  - Cost per reclaimed chunk
    - $(c_1 R + c_2 H)/(H - R)$
  - If $R/H > 0.5$
    - increase $H$
  - if $R/H < 0.5$
    - cost per reclaimed word is $c_1 + 2c_2 \sim 16$
  - There is no lower bound
The Mark Phase

for each root \( v \)

\[ \text{DFS}(v) \]

function DFS(x)

if \( x \) is a pointer and chunk \( x \) is not marked

mark \( x \)

for each reference field \( f_i \) of chunk \( x \)

\[ \text{DFS}(x.f_i) \]
Efficient implementation of Mark (DFS)

- Explicit stack
- Parent pointers
- Pointer reversal
- Other data structures
Adding Parent Pointer
Avoiding Parent Pointers
(Deutch-Schorr-Waite)

- Depth first search can be implemented without recursion or stack
- Maintain a counter of visited children
- Observation:
  - The pointer link from a parent to a child is not needed when it is visited
  - Temporary store pointer to the parent (instead of the field)
  - Restore when the visit of child is finished
Arriving at C
Visiting n-pointer field D

SET old parent pointer TO parent pointer;
SET Parent pointer TO chunk pointer;
SET Chunk pointer TO n-th pointer field of C;
SET n-th pointer field in C TO Old parent pointer;
About to return from D

SET old parent pointer TO parent pointer ;
SET Parent pointer TO n-th pointer field of C ;
SET n-th pointer field of C TO chunk pointer;
SET chunk pointer TO Old parent pointer;
Compaction

• The sweep phase can compact adjacent chunks
• Reduce fragmentation
Copying Collection

- Maintains two separate heaps
  - from-space
  - to-space
- Pointer `next` to the next free chunk in from-space
- A pointer `limit` to the last chunk in from-space
- If `next = limit` copy the reachable chunks from from-space into to-space
  - Set `next` and `limit`
  - Switch from-space and to-space
- Requires type information

```
next
limit
```
Breadth-first Copying Garbage Collection

next := beginning of to-space

scan := next

for each root r

r := Forward(r)

while scan < next

for each reference field \( f_i \) of chunk at scan

scan.\( f_i \) := Forward(scan.\( f_i \))

scan := scan + size of chunk at scan
The Forwarding Procedure

function Forward(p)

    if p points to from-space

        then if p.f₁ points to to-space

            return p.f₁

        else for each reference field fᵢ of p

            next.fᵢ := p.fᵢ

            p.f₁ := next

            next := next size of chunk p

            return p.f₁

    else return p
Amortized Cost of Copy Collection

c₃R / (H/2 - R)
Locality of references

• Copy collection does not create fragmentation
• Cheney's algorithm may lead to subfields that point to far away chunks
  – poor virtual memory and cache performance
• DFS normally yields better locality but is harder to implement
• DFS may also be bad for locality for chunks with more than one pointer fields
• A compromise is a hybrid breadth first search with two levels down (Semi-depth first forwarding)
• Results can be improved using dynamic information
The New Forwarding Procedure

function Forward(p)
    if p points to from-space
        then if p.f₁ points to to-space
            return p.f₁
        else Chase(p); return p.f₁
    else return p

function Chase(p)
    repeat
        q := next
        next := next + size of chunk p
        r := null
        for each reference field fᵢ of p
            q.fᵢ := p.fᵢ
            if q.fᵢ points to from-space and q.fᵢ.f₁ does not point to to-space
                then r := q.fᵢ

        p.f₁ := q
        p := r
    until p = null
Generational Garbage Collection

- Newly created objects contain higher percentage of garbage
- Partition the heap into generations $G_1$ and $G_2$
- First garbage collect the $G_1$ heap
  - chunks which are reachable
- After two or three collections chunks are promoted to $G_2$
- Once a while garbage collect $G_2$
- Can be generalized to more than two heaps
- But how can we garbage collect in $G_1$?
Scanning roots from older generations

• remembered list
  – The compiler generates code after each destructive update $b.f_i := a$
    to put $b$ into a vector of updated objects scanned by the garbage collector

• remembered set
  – remembered-list + “set-bit”

• Card marking
  – Divide the memory into $2^k$ cards

• Page marking
  – $k = \text{page size}$
  – virtual memory system catches updates to old-generations using the dirty-bit
Incremental Collection

• Even the most efficient garbage collection can interrupt the program for quite a while
• Under certain conditions the collector can run concurrently with the program (mutator)
• Need to guarantee that mutator leaves the chunks in consistent state, e.g., may need to restart collection
• Two solutions
  – compile-time
    • Generate extra instructions at store/load
  – virtual-memory
    • Mark certain pages as read(write)-only
    • a write into (read from) this page by the program restart mutator
Tricolor marking

• Generalized GC
• Three kinds of chunks
  – White
    • Not visited (not marked or not copied)
  – Grey
    • Marked or copied but children have not been examined
  – Black
    • Marked and their children are marked
Basic Tricolor marking

while there are any grey objects

    select a grey chunk p

    for each reference field \( f_i \) of chunk p

        if chunk \( p.f_i \) is white

            color chunk \( p.f_i \) grey

            color chunk p black

        Invariants

        • No black points to white

        • Every grey is on the collector's (stack or queue) data structure
Establishing the invariants

• Dijkstra, Lamport, et al
  – Mutator stores a white pointer $a$ into a black pointer $b$
    • color $a$ grey (compile-time)
• Steele
  – Mutator stores a white pointer $a$ into a black pointer $b$
    • color $b$ grey (compile-time)
• Boehm, Demers, Shenker
  – All black pages are marked read-only
  – A store into black page mark all the objects in this page grey (virtual memory system)
• Baker
  – Whenever the mutator fetches a pointer $b$ to a grey or white object
    • color $b$ grey (compile-time)
• Appel, Ellis, Li
  – Whenever the mutator fetches a pointer $b$ from a page containing a non black object
    • color every object on this page black and children grey (virtual memory system)
Interfaces to the Compiler

• The semantic analysis identifies chunk fields which are pointers and their size
• Generate runtime descriptors at the beginning of the chunks
  – Can employ different allocation/deallocation functions
• Pass the descriptors to the allocation function
• The compiler also passes pointer-map
  – the set of live pointer locals, temporaries, and registers
• Recorded at ?-time for every procedure
Summary

• Garbage collection is an effective technique
• Leads to more secure programs
• Tolerable cost
• But is not used in certain applications
  – Realtime
• Generational garbage collection works fast
  – Emulates stack
• But high synchronization costs
• Compiler can allocate data on stack
  – Escape analysis
• May be improved