UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

# Dynamic Memory Allocation

Questions answered in this lecture:

When is a stack appropriate? When is a heap?

What are best-fit, first-fit, worst-fit, and buddy allocation algorithms?

How can memory be freed (using reference counts or garbage collection)?

# Motivation for Dynamic Memory

Why do processes need dynamic allocation of memory?

- Do not know amount of memory needed at compile time
- Must be pessimistic when allocate memory statically
  - Allocate enough for worst possible case
  - Storage is used inefficiently

Recursive procedures

- Do not know how many times procedure will be nested

Complex data structures: lists and trees

- `struct my_t *p=(struct my_t *)malloc(sizeof(struct my_t));`

Two types of dynamic allocation

- Stack
- Heap

# Stack Organization

Definition: Memory is freed in opposite order from allocation

```
alloc(A);
alloc(B);
alloc(C);
free(C);
alloc(D);
free(D);
free(B);
free(A);
```

Implementation: Pointer separates allocated and freed space

- Allocate: Increment pointer
- Free: Decrement pointer

# Stack Discussion

OS uses stack for procedure call frames (local variables)

```
main () {
   int A = 0;
   foo (A);
   printf("A: %d\n", A);
}
void foo (int Z) {
   int A = 2;
   Z = 5;
   printf("A: %d Z: %d\n", A, Z);
}
```

Advantages

- Keeps all free space contiguous
- Simple to implement
- Efficient at run time

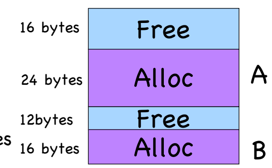
Disadvantages

- Not appropriate for all data structures

## Heap Organization

Definition: Allocate from any random location

- Memory consists of allocated areas and free areas (holes)
- Order of allocation and free is unpredictable




Advantage

- Works for all data structures

Disadvantages

- Allocation can be slow
- End up with small chunks of free space

Where to allocate 16 bytes? 12 bytes? 24 bytes??

## Fragmentation

Definition: Free memory that is too small to be usefully allocated

- External: Visible to allocator
- Internal: Visible to requester (e.g., if must allocate at some granularity)

Goal: Minimize fragmentation

- Few holes, each hole is large
- Free space is contiguous

Stack

- All free space is contiguous
- No fragmentation

Heap

- How to allocate to minimize fragmentation?

## Heap Implementation

Data structure: free list

- Linked list of free blocks, tracks memory not in use
- Header in each block
  - size of block
  - ptr to next block in list

void *Allocate(x bytes)

- Choose block large enough for request (>= x bytes)
- Keep remainder of free block on free list
- Update list pointers and size variable
- Return pointer to allocated memory

Free(ptr)

- Add block back to free list
- Merge adjacent blocks in free list, update ptrs and size variables

## Heap Allocation Policies

Best fit

- Search entire list for each allocation
- Choose free block that most closely matches size of request
- Optimization: Stop searching if see exact match

First fit

- Allocate first block that is large enough

Rotating first fit (or "Next fit"):

- Variant of first fit, remember place in list
- Start with next free block each time

Worst fit

- Allocate largest block to request (most leftover space)

## Heap Allocation Examples

Scenario: Two free blocks of size 20 and 15 bytes

Allocation stream: 10, 20

- Best
- First
- Worst

Allocation stream: 8, 12, 12

- Best
- First
- Worst

## Buddy Allocation

Fast, simple allocation for blocks of $2^n$ bytes [Knuth68]

void *Allocate (k bytes)

- Raise allocation request to nearest s = $2^n$
- Search free list for appropriate size
  - Represent free list with bitmap
  - Recursively divide larger free blocks until find block of size s
  - "Buddy" block remains free
- Mark corresponding bits as allocated

Free(ptr)

- Mark bits as free
- Recursively coalesce block with buddy, if buddy is free
  - May coalesce lazily (later, in background) to avoid overhead

## Buddy Allocation Example

Scenario: 1MB of free memory

Request stream:

- Allocate 70KB, 35KB, 80KB
- Free 35KB, 80KB, 70KB

## Comparison of Allocation Strategies

No optimal algorithm

- Fragmentation highly dependent on workload

Best fit

- Tends to leave some very large holes and some very small holes
  - Can't use very small holes easily

First fit

- Tends to leave "average" sized holes
- Advantage: Faster than best fit
- Next fit used often in practice

Buddy allocation

- Minimizes external fragmentation
- Disadvantage: Internal fragmentation when not 2^n request

## Memory Allocation in Practice

How is malloc() implemented?

Data structure: Free lists

- Header for each element of free list
  - pointer to next free block
  - size of block
  - magic number
- Where is header stored?
- What if remainder of block is smaller than header?

Two free lists

- One organized by size
  - Separate list for each popular, small size (e.g., 1 KB)
  - Allocation is fast, no external fragmentation
- Second is sorted by address
  - Use next fit to search appropriately
  - Free blocks shuffled between two lists

## Freeing Memory

C: Expect programmer to explicitly call free(ptr)

Two possible problems

- Dangling pointers: Recycle storage that is still in-use
  - Have two pointers to same memory, free one and use second

```
foo_t *a = malloc(sizeof(foo_t));
foo_t *b = a;
b->bar = 50;
free(a);
foo_t *c = malloc(sizeof(foo_t));
c->bar = 20;
printf("b->bar: %d\n", b->bar);
```

- Memory leaks: Forget to free storage
  - If lose pointer, can never free associated memory
  - Okay in short jobs, not okay for OS or long-running servers

```
foo_t *a = malloc(sizeof(foo_t));
foo_t *b = malloc(sizeof(foo_t));
b = a;
```

## Reference Counts

Idea: Reference counts

- Track number of references to each memory chunk
  - Increment count when new pointer references it
  - Decrement count when pointer no longer references it
- When reference count = 0, free memory

Examples

- Hard links in Unix

```
echo Hi > file
ln file new
rm file
cat new
```

- Smalltalk

Disadvantages

- Circular data structures --> Memory leaks

## Garbage Collection

Observation: To use data, must have pointer to it

- Without pointer, cannot access (or find) data
- Memory is free implicitly when no longer referenced
  - Programmer does not call free()

Approach

- When system needs more memory, free unreachable chunks

Requirements

- Must be able to find all objects (referenced and not)
- Must be able to find all pointers to objects
  - Strongly typed language
  - Compiler cooperates by marking data type in memory
    - Size of each object
    - Which fields are pointers

# Mark and Sweep Garbage Collection

Pass 1: Mark all reachable data

- Start with all statically allocated and local (stack) variables
- Mark each data object as reachable
- Recursively mark all data objects can reach through pointers

Pass 2: Sweep through all memory

- Examine each data object
- Free those objects not marked

Advantages

- Works with circular data structures
- Simple for application programmers

Disadvantages

- Often CPU-intensive (poor caching behavior too)
- Difficult to implement such that can execute job during g.c.
- Requires language support (Java, LISP)