Dynamic Memory Allocation

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
The Memory Hierarchy
we now have:

Virtual Memory
now what?
- code, global variables, jump tables, etc.
- allocated at fork/exec
- lifetime: *permanent*

**Static Data**
The **Stack**

- function activation records
- local vars, arguments, return values
- lifetime: *LIFO*

_pages allocated as needed (up to preset stack limit)_
explicitly requested from the kernel

- for *dynamic allocation*
- lifetime: *arbitrary*!

The *Heap*
- starts out empty
- **brk** pointer marks top of the heap

The *Heap*
heap mgmt syscall:

```c
void *sbrk(int inc); /* resizes heap by inc, returns old brk value */
```

The *Heap*
The Heap

```c
void *hp = sbrk(N);
```
can use `sbrk` to allocate structures:

```c
int **make_jagged_arr(int nrows, const int *dims) {
    int i, j;
    int **jarr = sbrk(sizeof(int *) * nrows);
    for (i=0; i<nrows; i++)
        jarr[i] = sbrk(sizeof(int) * dims[i]);
    return jarr;
}
```
but we can’t “free” this memory!!!
after the kernel allocates heap space for a process, it is *up to the process* to manage it!
“manage” = tracking memory in use, tracking memory not in use, reusing unused memory
job of the *dynamic memory allocator*
— typically included as a user-level library and/or language runtime feature
User Process

application program

dynamic memory allocator

malloc

Heap

sbrk

OS kernel

RAM

Disk
User Process

- application program
- dynamic memory allocator

free(p)

Heap

OS kernel

RAM

Disk
User Process

(application program)

(dynamic memory allocator)

_heap space may not be returned to the kernel!

OS kernel

Heap

free(p)

RAM

Disk
the DMA constructs a *user-level* abstraction (re-usable “blocks” of memory) on top of a *kernel-level* one (virtual memory)
the user-level implementation must make good use of the underlying infrastructure (the memory hierarchy)
e.g., the DMA should:

- maintain data alignment
- maximize throughput of requests
- help maximize memory utilization
- leverage locality

how to quantify this?
utilization = fraction of memory in use
- “in use” is a relative concept
- for DMA, “in use” = amount of memory actually requested by user (aka payload)
- vs. heap space obtained via sbrk
p1 = malloc(1024);
// util = 1K/4K = 25%

Heap
(given: 4KB page size)
p1 = malloc(1024);
// util = 1K/4K = 25%
p2 = malloc(2048);
// util = 3K/4K = 75%

Heap
(given: 4KB page size)
p1 = malloc(1024);
// util = 1K/4K = 25%
p2 = malloc(2048);
// util = 3K/4K = 75%
free(p1);
// util = 2K/4K = 50%

Heap
(given: 4KB page size)
p1 = malloc(1024);
// util = 1K/4K = 25%
p2 = malloc(2048);
// util = 3K/4K = 75%
free(p1);
// util = 2K/4K = 50%
p3 = malloc(2048);
// util = 4K/8K = 50%
\begin{verbatim}
p1 = malloc(1024);
// util = 1K/4K = 25%
p2 = malloc(2048);
// util = 3K/4K = 75%
free(p1);
// util = 2K/4K = 50%
p3 = malloc(2048);
// util = 4K/8K = 50%
free(p3);
// util = 2K/8K = 25%
free(p2);
// util = 0/8K = 0%

// all non-leaking
// programs end in 0%
\end{verbatim}
makes no sense to measure utilization at the end of process execution,

and it makes no sense to arbitrarily measure utilization during execution
instead, measure *peak memory utilization*

- ratio between *maximum aggregate payload* and *maximum heap size*

- “high water mark” measure

- assuming the heap never shrinks,
  
  \[
  \text{end heap size} = \text{max heap size}
  \]
p1 = malloc(1024);
// util = 1K/4K = 25%
p2 = malloc(2048);
// util = 3K/4K = 75%
free(p1);
// util = 2K/4K = 50%
p3 = malloc(2048);
// util = 4K/8K = 50%
free(p3);
// util = 2K/8K = 25%
free(p2);
// util = 0/8K = 0%

// all non-leaking
// programs end in 0%

- max agg. payload = 4K
- max heap size = 8K
- peak memory util = 50%
```c
p1 = malloc(100);    // 100
p2 = malloc(200);    // 300
free(p1);           // 200
p3 = malloc(300);    // 500
free(p2);           // 300
p4 = malloc(100);    // 400
p5 = malloc(200);    // 600
free(p3);           // 300
p6 = malloc(100);    // 400
p7 = malloc(300);    // 700
free(p4);           // 600
free(p5);           // 400
p8 = malloc(200);    // 600
```

peak memory util
= 700 / 1024
≈ 68%

aggregate payload
utilization is affected by memory fragmentation
two forms:
  1. *internal* fragmentation
  2. *external* fragmentation
when allocating blocks of memory, it is convenient to make them *self-describing*
i.e., store metadata alongside blocks with size, allocation status, etc.
allocator must also adhere to alignment requirements (to help optimize cache/memory fetches)
amount of internal fragmentation is easy to predict, as it’s based on pre-determined factors

- metadata = fixed amount

- $k$-byte alignment $\rightarrow$ max $k-1$ padding
Heap
external fragmentation
external fragmentation may affect future heap utilization;
i.e., by preventing free space from being re-used
Heap

malloc?
Heap
Heap

forced to request more heap space

malloc?
hard to predict the effect of external fragmentation on utilization

in general, we might:

- prefer fewer, larger spans of free space
- try to keep similarly sized blocks together in memory
but these recommendations are *heuristics*!
- may be defeated by pathological cases
- don’t account for real-world behavior
It has been proven that for any possible allocation algorithm, there will always be the possibility that some application program will allocate and deallocate blocks in some fashion that defeats the allocator’s strategy and forces it into severe fragmentation ... Not only are there no provably good allocation algorithms, there are proofs that any allocator will be bad for some possible applications.

P. Wilson, M. Johnstone, M. Neely, D. Boles,
Dynamic Memory Allocation: A Survey and Critical Review