Dynamic Memory Management

Dynamic Memory Management

I) Why care

Dynamic Memory Management

I) Why care

II) Requirements

Dynamic Memory Management

I) Why care

II) Requirements

III) Basic Techniques

Dynamic Memory Management

I) Why care

II) Requirements

III) Basic Techniques

IV) False Sharing

Dynamic Memory Management

I) Why care

II) Requirements

III) Basic Techniques

IV) False Sharing

V) RMalloc

Dynamic Memory Management

I) Why care

II) Requirements

III) Basic Techniques

IV) False Sharing

V) RMalloc

VI) Benchmarks

Motivation

Why do we have to care about memory management ?

Motivation

Why do we have to care about memory management ?

Answer:

LibBEM, building DLP-PC operator, $n = 16386$		
SUN	**PTmalloc**	**rmalloc**
1026.1 s (1930 MB)	951.7 s (1835 MB)	327.5 s (1894 MB)

Motivation

Why do we have to care about memory management ?

Answer:

LibBEM, building DLP-PC operator, $n = 16386$		
SUN	**PTmalloc**	**rmalloc**
1026.1 s (1930 MB)	951.7 s (1835 MB)	327.5 s (1894 MB)

$\mathcal{H}$-matrix Multiplication, constant rank			
p	**SUN**	**PTmalloc**	**rmalloc**
1	98.6 s (82 MB)	98.4 s (81 MB)	99.5 s (84 MB)
16	47.3 s (100 MB)	7.3 s (94 MB)	7.2 s (109 MB)

Requirements

Requirements

- $\mathcal{O}(1)$ complexity for malloc and free

Requirements

- $\mathcal{O}(1)$ complexity for malloc and free
- fast handling of small objects

Requirements

- $\mathcal{O}(1)$ complexity for malloc and free
- fast handling of small objects
- Parallel efficiency: no blocking when running in parallel

Requirements

- $\mathcal{O}(1)$ complexity for malloc and free
- fast handling of small objects
- Parallel efficiency: no blocking when running in parallel
- Parallel safety: no data corruption when called simultaneously

Requirements

- $\mathcal{O}(1)$ complexity for malloc and free
- fast handling of small objects
- Parallel efficiency: no blocking when running in parallel
- Parallel safety: no data corruption when called simultaneously
- should handle large memory footprint (> 64GB)

Requirements

- $\mathcal{O}(1)$ complexity for malloc and free
- fast handling of small objects
- Parallel efficiency: no blocking when running in parallel
- Parallel safety: no data corruption when called simultaneously
- should handle large memory footprint (> 64GB)
- minimal memory allocation, low *fragmentation*

Fragmentation

Fragmentation

External fragmentation:

- memory allocator has free chunks of memory but can't serve request
- usual cause: available blocks are too small

Fragmentation

External fragmentation:

- memory allocator has free chunks of memory but can't serve request
- usual cause: available blocks are too small

Internal fragmentation:

- returned block is larger than requested: remainder is wasted

Fragmentation

External fragmentation:

- memory allocator has free chunks of memory but can't serve request
- usual cause: available blocks are too small

Internal fragmentation:

- returned block is larger than requested: remainder is wasted

Options:

- *Splitting*: split larger block and put remainder back into allocator

Fragmentation

External fragmentation:

- memory allocator has free chunks of memory but can't serve request
- usual cause: available blocks are too small

Internal fragmentation:

- returned block is larger than requested: remainder is wasted

Options:

- *Splitting*: split larger block and put remainder back into allocator
- *Coalescing*: if free chunks are adjacent in memory, join them

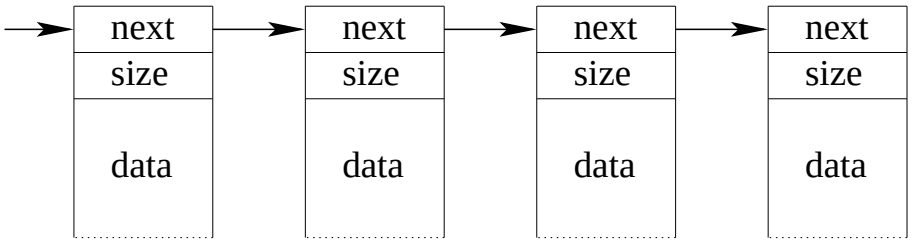
Basic Techniques

Sequential Fits

Basic Techniques

Sequential Fits

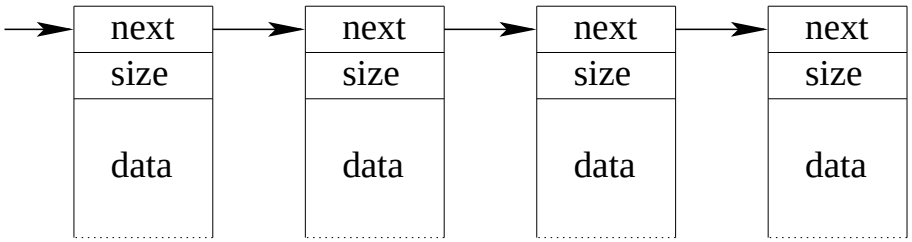
Single list of free blocks:



Basic Techniques

Sequential Fits

Single list of free blocks:



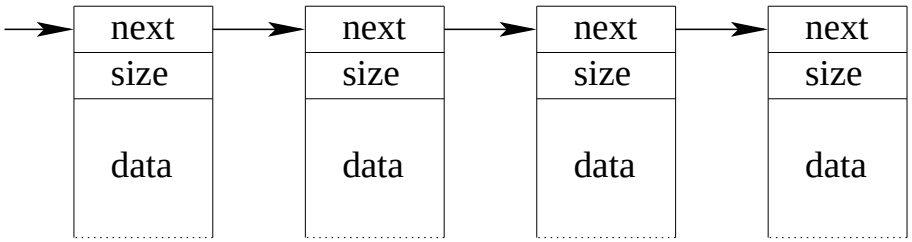
Strategies for malloc:

- *First fit*: use first chunk large enough to hold data
  - high internal fragmentation without splitting

Basic Techniques

Sequential Fits

Single list of free blocks:



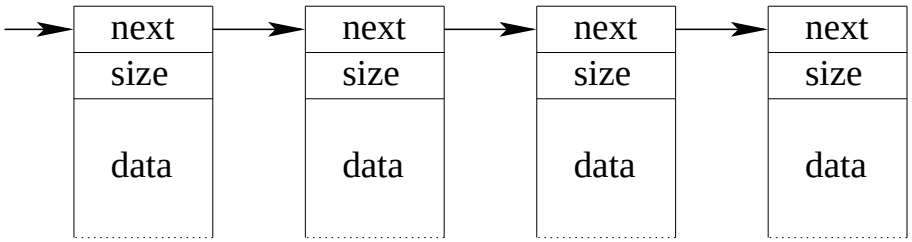
Strategies for malloc:

- *First fit*: use first chunk large enough to hold data
  - high internal fragmentation without splitting
- *Best fit*: use smallest chunk large enough to hold data
  - in theory: higher memory usage than first fit

Basic Techniques

Sequential Fits

Single list of free blocks:



Strategies for malloc:

- *First fit*: use first chunk large enough to hold data
  - high internal fragmentation without splitting
- *Best fit*: use smallest chunk large enough to hold data
  - in theory: higher memory usage than first fit

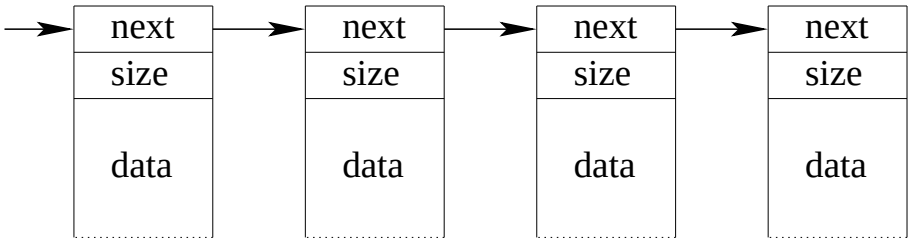
Properties:

- slow: using linear search (with trees: logarithmic)

Basic Techniques

Sequential Fits

Single list of free blocks:



Strategies for malloc:

- *First fit*: use first chunk large enough to hold data
  - high internal fragmentation without splitting
- *Best fit*: use smallest chunk large enough to hold data
  - in theory: higher memory usage than first fit

Properties:

- slow: using linear search (with trees: logarithmic)
- bad scaling w.r.t. processors and memory

Segregated Free Lists

Segregated Free Lists

Define *size classes*:

$$S_0 < S_1 < \cdots < S_n, \quad S_i \in \mathbb{N}$$

Segregated Free Lists

Define *size classes*:

$$S_0 < S_1 < \cdots < S_n, \quad S_i \in \mathbb{N}$$

A size $s \in \mathbb{N}$ *belongs* to size class S_i if

$$S_{i-1} < s \leq S_i$$

Segregated Free Lists

Define *size classes*:

$$S_0 < S_1 < \cdots < S_n, \quad S_i \in \mathbb{N}$$

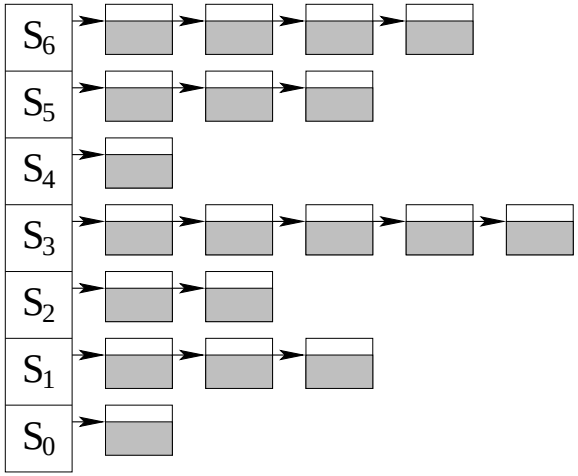
A size $s \in \mathbb{N}$ *belongs* to size class S_i if

$$S_{i-1} < s \leq S_i$$

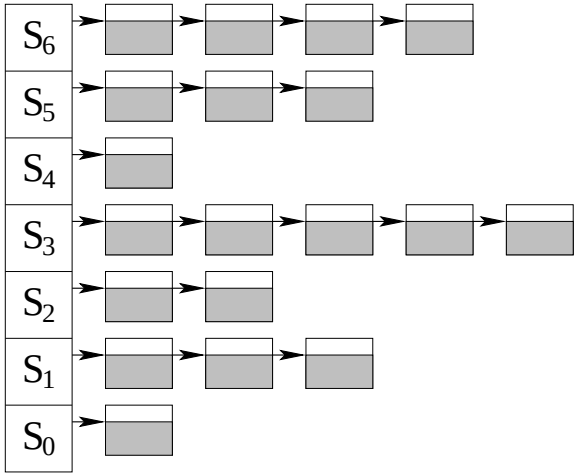
Typical definition: by power series

$$S_i = b^i, \quad b > 1$$

Segregated Free Lists: use array of lists for each size class

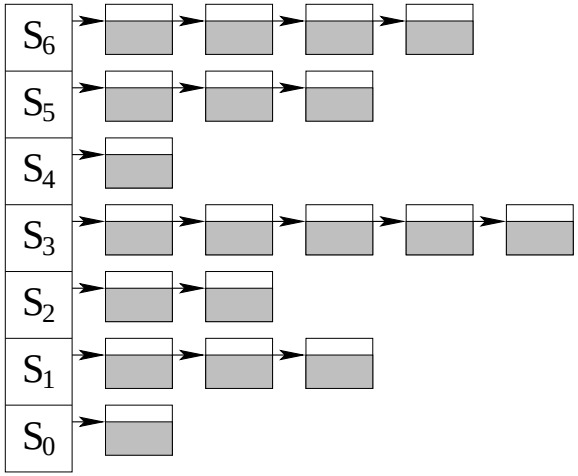


Segregated Free Lists: use array of lists for each size class



- fast malloc and free: only small lists for searching
- *good fit* (almost best fit) strategy: size of unused part is bounded (by b)

Segregated Free Lists: use array of lists for each size class



- fast malloc and free: only small lists for searching
- *good fit* (almost best fit) strategy: size of unused part is bounded (by b)

Option: round up each size to size of class

- $O(1)$ complexity for malloc and free (all blocks in class have same size)
- higher memory consumption, but bounded internal fragmentation

Buddy Systems

Buddy Systems

- special case of segregated free lists

Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules

Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy

Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy
- Advantage: very high speed, little memory overhead

Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy
- Advantage: very high speed, little memory overhead
- Disadvantage: high internal fragmentation possible due to fixed splitting rules

Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy
- Advantage: very high speed, little memory overhead
- Disadvantage: high internal fragmentation possible due to fixed splitting rules

Examples:

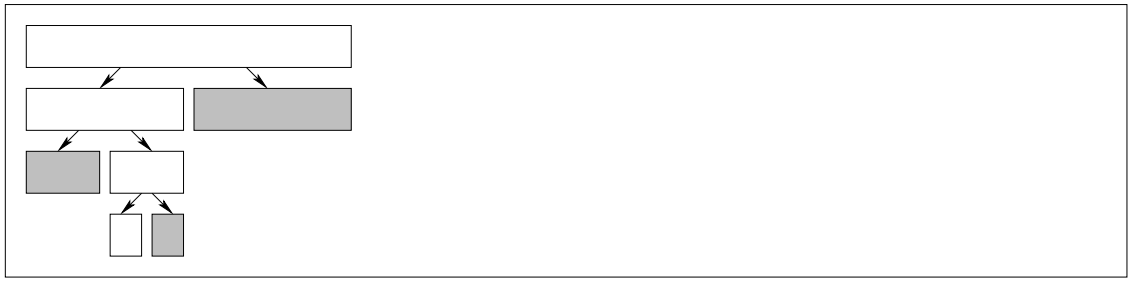
- *Binary buddies*: size classes are power of 2

Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy
- Advantage: very high speed, little memory overhead
- Disadvantage: high internal fragmentation possible due to fixed splitting rules

Examples:

- *Binary buddies*: size classes are power of 2

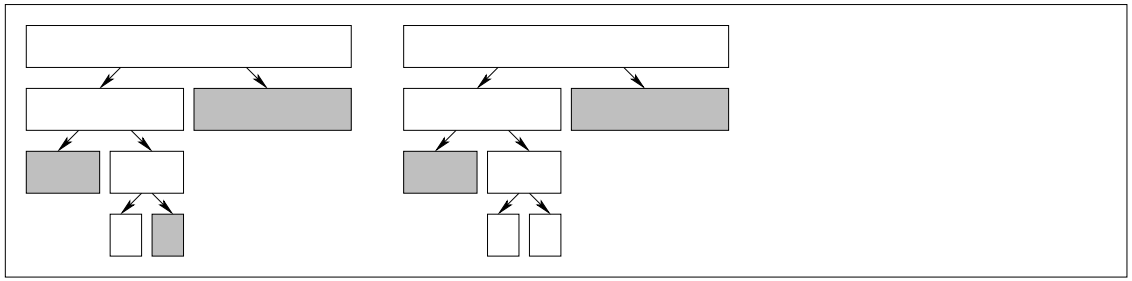


Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy
- Advantage: very high speed, little memory overhead
- Disadvantage: high internal fragmentation possible due to fixed splitting rules

Examples:

- *Binary buddies*: size classes are power of 2

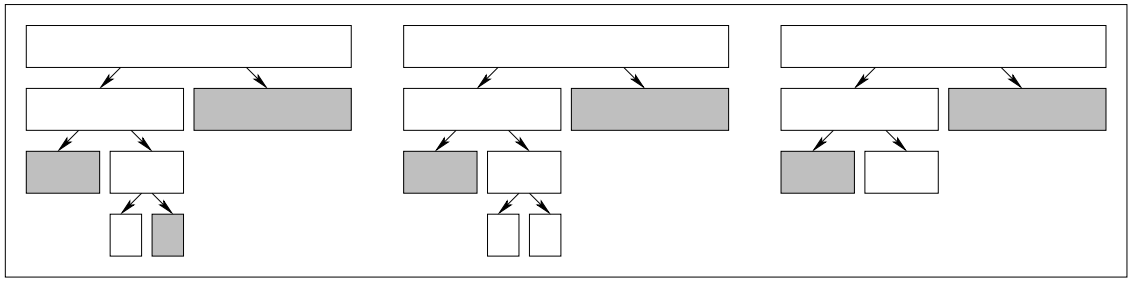


Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy
- Advantage: very high speed, little memory overhead
- Disadvantage: high internal fragmentation possible due to fixed splitting rules

Examples:

- *Binary buddies*: size classes are power of 2

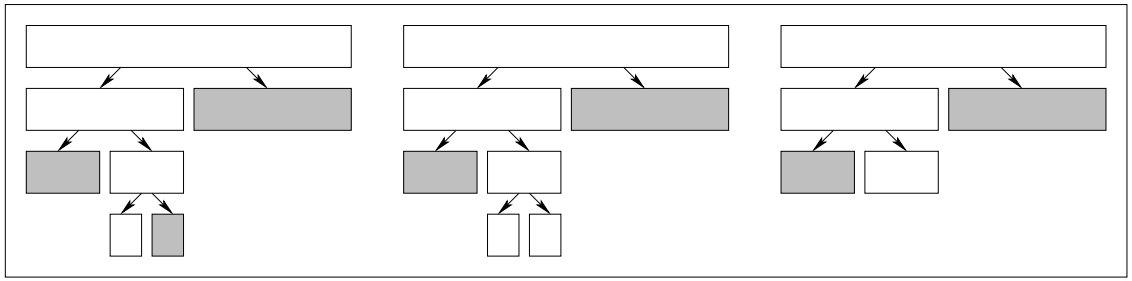


Buddy Systems

- special case of segregated free lists
- memory is hierarchically divided into two chunks (buddies) by fixed rules
- each chunk can only be merged with his buddy
- Advantage: very high speed, little memory overhead
- Disadvantage: high internal fragmentation possible due to fixed splitting rules

Examples:

- *Binary buddies*: size classes are power of 2



- *Fibonacci buddies*: uses Fibonacci series for splitting

False Sharing

Problem:

- access to memory is done via cache (memory hierarchy)

False Sharing

Problem:

- access to memory is done via cache (memory hierarchy)
- smallest block in cache is a *cacheline* (usually 64 bytes long)

False Sharing

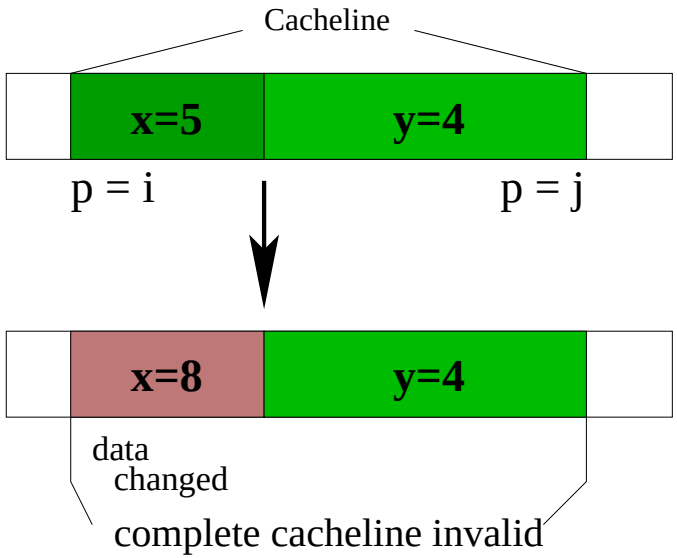
Problem:

- access to memory is done via cache (memory hierarchy)
- smallest block in cache is a *cacheline* (usually 64 bytes long)
- changing one byte in cacheline make whole cacheline invalid

False Sharing

Problem:

- access to memory is done via cache (memory hierarchy)
- smallest block in cache is a *cacheline* (usually 64 bytes long)
- changing one byte in cacheline make whole cacheline invalid
- writing data on one processor leads to memory access on other processor



How to avoid false sharing:

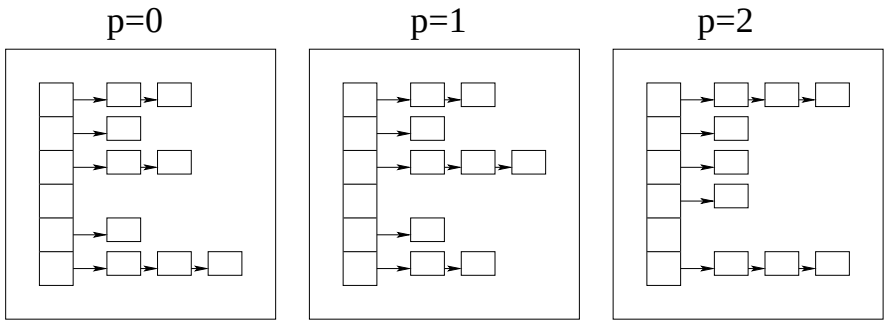
- align all data-blocks at 64 byte addresses

How to avoid false sharing:

- align all data-blocks at 64 byte addresses
  - cacheline can not be shared by different data
  - but: high internal fragmentation

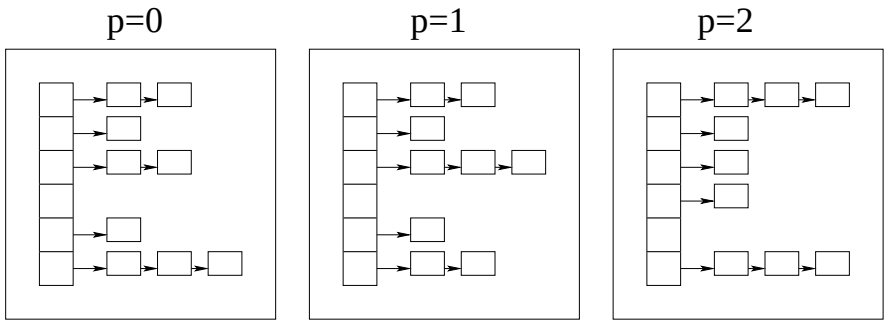
How to avoid false sharing:

- align all data-blocks at 64 byte addresses
  - cacheline can not be shared by different data
  - but: high internal fragmentation
- use *private heaps* for different processors (*PTmalloc*, *LKmalloc*)



How to avoid false sharing:

- align all data-blocks at 64 byte addresses
  - cacheline can not be shared by different data
  - but: high internal fragmentation
- use *private heaps* for different processors (*PTmalloc*, *LKmalloc*)



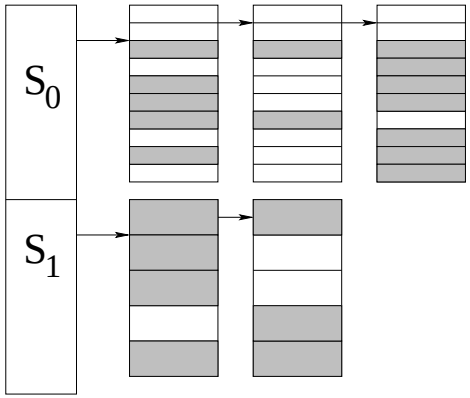
  - data on different processors is allocated from different areas
  - important: freed chunks must be returned to their heaps
  - memory management is completely independent on different processors
  - but: memory consumption grows linearly in p (higher external fragmentation)

Hoard: avoiding false sharing mostly with a bounded ext. fragmentation

- private heaps for each processor

Hoard: avoiding false sharing mostly with a bounded ext. fragmentation

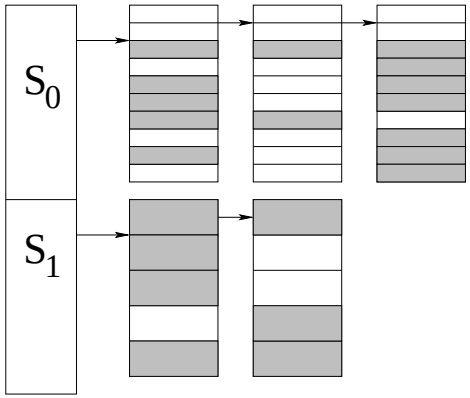
- private heaps for each processor
- chunks are collected in *superblocks* of a fixed size



thereby avoiding false sharing

Hoard: avoiding false sharing mostly with a bounded ext. fragmentation

- private heaps for each processor
- chunks are collected in *superblocks* of a fixed size

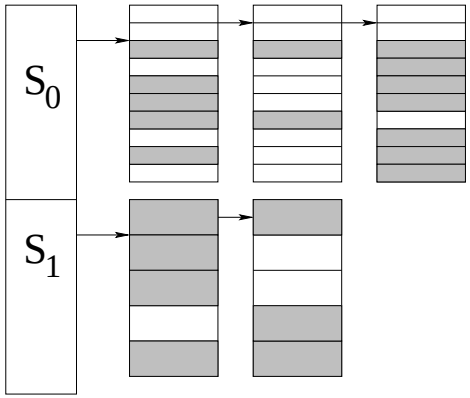


  thereby avoiding false sharing

- if free part of a superblock exceeds some threshold, move superblock to a *global heap* (bounded ext. fragmentation)

Hoard: avoiding false sharing mostly with a bounded ext. fragmentation

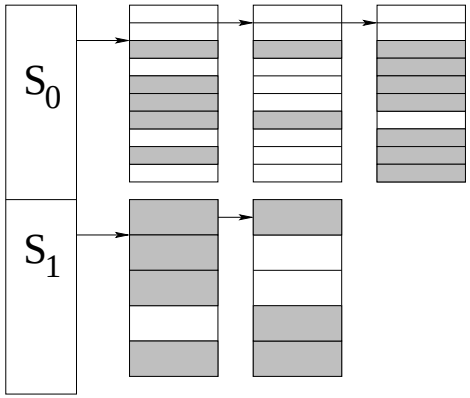
- private heaps for each processor
- chunks are collected in *superblocks* of a fixed size



  thereby avoiding false sharing
- if free part of a superblock exceeds some threshold, move superblock to a *global heap* (bounded ext. fragmentation)
- allocate new superblock by moving superblock from global heap

Hoard: avoiding false sharing mostly with a bounded ext. fragmentation

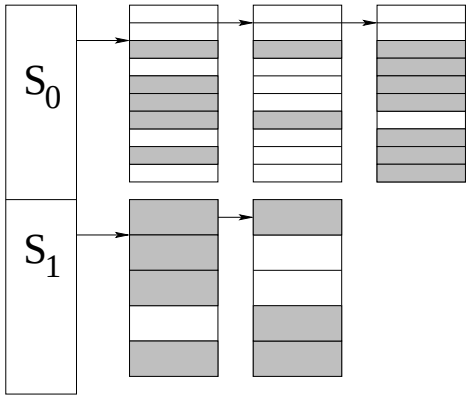
- private heaps for each processor
- chunks are collected in *superblocks* of a fixed size



  thereby avoiding false sharing
- if free part of a superblock exceeds some threshold, move superblock to a *global heap* (bounded ext. fragmentation)
- allocate new superblock by moving superblock from global heap
- blocking only when accessing global heap (good parallel speedup)

Hoard: avoiding false sharing mostly with a bounded ext. fragmentation

- private heaps for each processor
- chunks are collected in *superblocks* of a fixed size



  thereby avoiding false sharing
- if free part of a superblock exceeds some threshold, move superblock to a *global heap* (bounded ext. fragmentation)
- allocate new superblock by moving superblock from global heap
- blocking only when accessing global heap (good parallel speedup)
- but: maximal managed size defined by size of superblock

RMalloc

- uses per-processor heap (although $\mathcal{O}(p)$ memory consumption)

RMalloc

- uses per-processor heap (although $\mathcal{O}(p)$ memory consumption)
- different strategy for different sizes: small, middle and large chunks
  - Large chunks ($\geq 260\mathrm{MB}$):
    - directly managed by operating system (mmap)

RMalloc

- uses per-processor heap (although $\mathcal{O}(p)$ memory consumption)
- different strategy for different sizes: small, middle and large chunks
  - Large chunks ($\geq 260\mathrm{MB}$):
    - directly managed by operating system (mmap)
  - Middle sized chunks ($256\mathrm{B} < s < 260\mathrm{MB}$):
    - segregated free lists with immediate coalescing and splitting

RMalloc

- uses per-processor heap (although $\mathcal{O}(p)$ memory consumption)
- different strategy for different sizes: small, middle and large chunks
  - Large chunks ($\geq 260\mathrm{MB}$):
    - directly managed by operating system (mmap)
  - Middle sized chunks ($256\mathrm{B} < s < 260\mathrm{MB}$):
    - segregated free lists with immediate coalescing and splitting
  - Small chunks ($\leq 256\mathrm{B}$):
    - stored in *containers* (superblocks) of a fixed size ($8\mathrm{kB}$)
    - unused containers are cached and can be reused by different size classes
    - avoids splitting/coalescing for small blocks, reduces external fragmentation

- heaps are tightly bounded to threads; only if thread finishes, heap can be reused by new thread: no sharing of data possible

- heaps are tightly bounded to threads; only if thread finishes, heap can be reused by new thread: no sharing of data possible
- memory is preallocated from operating system to reduce no. of *system calls*

- heaps are tightly bounded to threads; only if thread finishes, heap can be reused by new thread: no sharing of data possible
- memory is preallocated from operating system to reduce no. of *system calls*
- size of preallocated area dependent on current memory usage

Benchmarks

Benchmarks

$\mathcal{H}$-building, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	1.2 s	5.3 s	24.9 s	147.8 s	592 MB

Benchmarks

$\mathcal{H}$-building, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	1.2 s	5.3 s	24.9 s	147.8 s	592 MB
PT	1.6 s	6.7 s	30.7 s	170.4 s	543 MB

Benchmarks

$\mathcal{H}$-building, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	1.2 s	5.3 s	24.9 s	147.8 s	592 MB
PT	1.6 s	6.7 s	30.7 s	170.4 s	543 MB
RM	1.2 s	5.0 s	20.9 s	87.2 s	572 MB

Benchmarks

$\mathcal{H}$-building, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	1.2 s	5.3 s	24.9 s	147.8 s	592 MB
PT	1.6 s	6.7 s	30.7 s	170.4 s	543 MB
RM	1.2 s	5.0 s	20.9 s	87.2 s	572 MB

LU-factorisation, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	3.4 s	27.1 s	181.2 s	1195.1 s	1613 MB

Benchmarks

$\mathcal{H}$-building, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	1.2 s	5.3 s	24.9 s	147.8 s	592 MB
PT	1.6 s	6.7 s	30.7 s	170.4 s	543 MB
RM	1.2 s	5.0 s	20.9 s	87.2 s	572 MB

LU-factorisation, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	3.4 s	27.1 s	181.2 s	1195.1 s	1613 MB
PT	3.3 s	27.2 s	201.7 s	1685.1 s	1562 MB

Benchmarks

$\mathcal{H}$-building, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	1.2 s	5.3 s	24.9 s	147.8 s	592 MB
PT	1.6 s	6.7 s	30.7 s	170.4 s	543 MB
RM	1.2 s	5.0 s	20.9 s	87.2 s	572 MB

LU-factorisation, fixed precision					
malloc	**4096**	**16384**	**65536**	**262144**	**memory**
SUN	3.4 s	27.1 s	181.2 s	1195.1 s	1613 MB
PT	3.3 s	27.2 s	201.7 s	1685.1 s	1562 MB
RM	3.4 s	26.0 s	168.1 s	1001.8 s	1599 MB

$\mathcal{H}$-inversion, fixed precision				
malloc	**4096**	**16384**	**65536**	**memory**
SUN	56.7 s	539.2 s	4065.3 s	1399 MB

$\mathcal{H}$-inversion, fixed precision				
malloc	**4096**	**16384**	**65536**	**memory**
SUN	56.7 s	539.2 s	4065.3 s	1399 MB
PT	56.3 s	571.7 s	5174.5 s	1373 MB

$\mathcal{H}$-inversion, fixed precision				
malloc	**4096**	**16384**	**65536**	**memory**
SUN	56.7 s	539.2 s	4065.3 s	1399 MB
PT	56.3 s	571.7 s	5174.5 s	1373 MB
RM	56.1 s	510.7 s	3683.0 s	1399 MB

$\mathcal{H}$-inversion, fixed precision				
malloc	**4096**	**16384**	**65536**	**memory**
SUN	56.7 s	539.2 s	4065.3 s	1399 MB
PT	56.3 s	571.7 s	5174.5 s	1373 MB
RM	56.1 s	510.7 s	3683.0 s	1399 MB

$\mathcal{H}$-matrix Multiplication, fixed rank						
malloc	$t(1)$	$t(4)$	$t(8)$	$t(12)$	$t(16)$	**memory**
SUN	98.6 s	34.9 s	39.4 s	45.6 s	47.3 s	100 MB

$\mathcal{H}$-inversion, fixed precision				
malloc	**4096**	**16384**	**65536**	**memory**
SUN	56.7 s	539.2 s	4065.3 s	1399 MB
PT	56.3 s	571.7 s	5174.5 s	1373 MB
RM	56.1 s	510.7 s	3683.0 s	1399 MB

$\mathcal{H}$-matrix Multiplication, fixed rank						
malloc	$t(1)$	$t(4)$	$t(8)$	$t(12)$	$t(16)$	**memory**
SUN	98.6 s	34.9 s	39.4 s	45.6 s	47.3 s	100 MB
PT	98.4 s	26.8 s	13.7 s	9.3 s	7.3 s	94 MB

$\mathcal{H}$-inversion, fixed precision				
malloc	**4096**	**16384**	**65536**	**memory**
SUN	56.7 s	539.2 s	4065.3 s	1399 MB
PT	56.3 s	571.7 s	5174.5 s	1373 MB
RM	56.1 s	510.7 s	3683.0 s	1399 MB

$\mathcal{H}$-matrix Multiplication, fixed rank						
malloc	$t(1)$	$t(4)$	$t(8)$	$t(12)$	$t(16)$	**memory**
SUN	98.6 s	34.9 s	39.4 s	45.6 s	47.3 s	100 MB
PT	98.4 s	26.8 s	13.7 s	9.3 s	7.3 s	94 MB
MT	99.3 s	26.1 s	13.3 s	9.1 s	7.2 s	144 MB

$\mathcal{H}$-inversion, fixed precision				
malloc	**4096**	**16384**	**65536**	**memory**
SUN	56.7 s	539.2 s	4065.3 s	1399 MB
PT	56.3 s	571.7 s	5174.5 s	1373 MB
RM	56.1 s	510.7 s	3683.0 s	1399 MB

$\mathcal{H}$-matrix Multiplication, fixed rank						
malloc	$t(1)$	$t(4)$	$t(8)$	$t(12)$	$t(16)$	**memory**
SUN	98.6 s	34.9 s	39.4 s	45.6 s	47.3 s	100 MB
PT	98.4 s	26.8 s	13.7 s	9.3 s	7.3 s	94 MB
MT	99.3 s	26.1 s	13.3 s	9.1 s	7.2 s	144 MB
RM	99.5 s	26.2 s	13.3 s	9.0 s	7.2 s	109 MB

libBEM, building PC operator				
malloc	**4098**	**16386**	**65538**	**memory**
SUN	48.6 s	290.6 s	1477.2 s	8609 MB

libBEM, building PC operator				
malloc	**4098**	**16386**	**65538**	**memory**
SUN	48.6 s	290.6 s	1477.2 s	8609 MB
PT	46.4 s	223.3 s	838.4 s	8176 MB

libBEM, building PC operator				
malloc	**4098**	**16386**	**65538**	**memory**
SUN	48.6 s	290.6 s	1477.2 s	8609 MB
PT	46.4 s	223.3 s	838.4 s	8176 MB
RM	45.6 s	219.2 s	811.2 s	8395 MB