Lecture 12
Storage Allocation

Saman Amarasinghe
2014
Outline

Memory Systems Revisited
Stacks
Fixed–Size Heap Allocation
Variable–Size Heap Allocation
Garbage Collection by Reference Counting
Stop–and–Copy Garbage Collection
MEMORY SYSTEMS REVISITED
# Intel Core 2 Quad Processor

## Memory Sub-system

<table>
<thead>
<tr>
<th>Cache Type</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>L1 Data Cache</strong></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L1 Instruction Cache</strong></td>
<td>32 KB</td>
<td>64 bytes</td>
<td>3 cycles</td>
<td>8-way</td>
</tr>
<tr>
<td><strong>L2 Cache</strong></td>
<td>6 MB</td>
<td>64 bytes</td>
<td>14 cycles</td>
<td>24-way</td>
</tr>
</tbody>
</table>

![Intel Core 2 Quad Processor Diagram](image-url)
Intel® Nehalem™ Microarchitecture – Memory Sub-system

Intel 6 Core Processor

<table>
<thead>
<tr>
<th>Level</th>
<th>Size</th>
<th>Line Size</th>
<th>Latency</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 Data Cache</td>
<td>32 KB</td>
<td>64 bytes</td>
<td>4 ns</td>
<td>8-way</td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td>32 KB</td>
<td>64 bytes</td>
<td>4 ns</td>
<td>4-way</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>256 KB</td>
<td>64 bytes</td>
<td>10 ns</td>
<td>8-way</td>
</tr>
<tr>
<td>L3 Cache</td>
<td>30 MB</td>
<td>64 bytes</td>
<td>50 ns</td>
<td>16-way</td>
</tr>
<tr>
<td>Main Memory</td>
<td>64 bytes</td>
<td>75 ns</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
for(rep=0; rep < REP; rep++)
for(a=0; a < N ; a++)

Capacity misses if larger than the cache at each level
for(rep=0; rep < REP; rep++)
  for(a=0; a < N ; a++)

Amazing prefetcher

Single core cannot saturate the memory system
mask = (1 << n) – 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) – 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) – 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
mask = (1<<n) – 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}
Virtual Memory System

You access virtual memory, your computer has physical memory & disk

- $2^{64}$ virtual memory
- Limited physical memory
- All memory backed up on disk

Virtual2physical mapped by pages

- x86 supports 4K small and 4M large pages

OS Manages Virtual memory

- Allocates virtual pages, maps them to physical
- Backs pages on disk and bring them in and out
- provides a page table to the hardware

Hardware caches the entries in the TLB

When you access a memory location

- If that page is mapped to physical memory and the mapping is cached in TLB $\rightarrow$ aok ($\sim1$ cycle)
- If mapping is not in TLB $\rightarrow$ TLB miss. ($\sim100$ cycles)
  - The HW gets the mapping from the page table and caches it in TLB
- If page is not mapped $\rightarrow$ Page fault. ($\sim1,000,000$ cycles)
  - The OS has to get involved in bringing in the page to physical memory from disk and updating the page table
mask = (1<<n) - 1;
for(rep=0; rep < REP; rep++) {
    addr = ((rep + 523)*253573) & mask;
}

L1 Cache Miss
L2 Cache Miss
L3 Cache Miss
TLB misses
Performance
• Page size is 4 KB
• Number of TLB entries is 512

• So, total memory that can be mapped by TLB is 2 MB
• L3 cache is 12 MB!

• TLB misses before L3 cache misses!
Intel® IvyBridge™v2 E5-2692 Microarchitecture – Memory Sub-system

Intel 12 Core Processor

<table>
<thead>
<tr>
<th></th>
<th>L1 Data Cache</th>
<th>L1 Instruction Cache</th>
<th>L2 Cache</th>
<th>L3 Cache</th>
<th>Main Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size</td>
<td>Line Size</td>
<td>Latency</td>
<td>Associativity</td>
<td>Size</td>
<td>Line Size</td>
</tr>
<tr>
<td>32 KB</td>
<td>64 bytes</td>
<td>4 ns</td>
<td>8-way</td>
<td>30 MB</td>
<td>64 bytes</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Data Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L1 Instruction Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>L3 Cache</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
STACKS
Stack Allocation

Array and pointer

Allocate $x$ bytes

```c
sp += x;
return sp - x;
```
Array and pointer

Allocate \( x \) bytes

\[
\begin{align*}
\text{sp} & \text{ += } x; \\
\text{return } \text{sp} - x;
\end{align*}
\]

How does check for stack overflow?
Guard Pages

Surround the page with two unallocated pages (guard pages)

Pros

- Cost of bounds check is zero

Cons

- Bounds violation (a page fault) is very costly
- Data sizes has to be a multiple of available page size
  - 4KB or 4MB
- If offset > guard page size, may not catch the violation
Stack Deallocation

Array and pointer

A

used

unused

Allocate $x$ bytes

Free $x$ bytes

\begin{align*}
\text{sp} &\;+= \;x; \\
\text{return} \;\text{sp} \;-%\; x;
\end{align*}

\begin{align*}
\text{sp} &\;-%\; x;
\end{align*}
### Stack Deallocation

#### Array and pointer

- **Allocate** $x$ bytes:
  
  ```
  sp += x;
  return sp - x;
  ```

- **Free** $x$ bytes:
  
  ```
  sp -= x;
  ```

Should check for stack underflow.
Stack Storage

Array and pointer

 Allocate x bytes

\[
\text{sp += x;} \\
\text{return sp - x;} \\
\]

Free x bytes

\[
\text{sp -= x;} \\
\]

- Allocating and freeing take $\Theta(1)$ time.
- Must free consistent with stack discipline.
- Limited applicability, but great when it works!
- One can allocate on the call stack using `alloca()`, but this function is deprecated, and the compiler is more efficient with fixed-size frames.
FIXED-SIZE HEAP ALLOCATION
Heap Allocation*

C provides `malloc()` and `free()`. C++ provides `new` and `delete`.

Unlike Java and Python, C and C++ provide no garbage collector. Heap storage allocated by the programmer must be freed explicitly. Failure to do so creates a memory leak. Also, watch for dangling pointers and double freeing.

Memory checkers can assist in finding these pernicious bugs:

```bash
% valgrind --leakcheck=yes ./myprog <arguments>
```

Valgrind is installed on cloud machines. See [http://valgrind.org](http://valgrind.org) for details.

*Do not confuse with a heap data structure.
Fixed-Size Allocation

Free list

A

used

used

used

used

free
Fixed–Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```
Fixed-Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```
Fixed-Size Allocation

Free list

Allocate 1 object

\[
x = \text{free}; \\
\text{free} = \text{free}->\text{next}; \\
\text{return } x;
\]

Should check \text{free} \neq \text{NULL}.
Fixed-Size Allocation

Free list

Allocate 1 object

\[
x = \text{free}; \\
\text{free} = \text{free}\rightarrow \text{next}; \\
\text{return } x;
\]
Fixed–Size Allocation

Free list

Allocate 1 object

```c
x = free;
free = free->next;
return x;
```
Fixed-Size Deallocation

Free list

Allocate 1 object

\[
x = \text{free}; \\
\text{free} = \text{free}\rightarrow\text{next}; \\
\text{return } x;
\]

Free object \text{x}

\[
x\rightarrow\text{next} = \text{free}; \\
\text{free} = x;
\]
Fixed-Size Deallocation

Free list

Allocate 1 object

\[
x = \text{free}; \\
\text{free} = \text{free}\rightarrow\text{next}; \\
\text{return } x;
\]

free object \(x\)

\[
x\rightarrow\text{next} = \text{free}; \\
\text{free} = x;
\]
Fixed-Size Deallocation

Free list

Allocate 1 object

\[ x = \text{free}; \]
\[ \text{free} = \text{free} \rightarrow \text{next}; \]
\[ \text{return } x; \]

Free object \( x \)

\[ x \rightarrow \text{next} = \text{free}; \]
\[ \text{free} = x; \]
Fixed-Size Deallocation

Free list

Allocate 1 object

\[
x = \text{free}; \\
\text{free} = \text{free} \rightarrow \text{next}; \\
\text{return } x;
\]

free object x

\[
x \rightarrow \text{next} = \text{free}; \\
\text{free} = x;
\]
Free Lists

Free list

- Allocating and freeing take $\Theta(1)$ time.
- Good temporal locality.
- Poor spatial locality due to external fragmentation — blocks distributed across virtual memory — which can increase the size of the page table and cause disk thrashing.
- The translation lookaside buffer (TLB) can also be a problem.
Mitigating External Fragmentation

- Keep a free list per disk page.
- Allocate from the free list for the fullest page.
- Free a block of storage to the free list for the page on which the block resides.
- If a page becomes empty (only free-list items), the virtual-memory system can page it out without affecting program performance.

90–10 is better than 50–50:

Probability that 2 random accesses hit the same page

\[= 0.9 \times 0.9 + 0.1 \times 0.1 = 0.82 \text{ versus } 0.5 \times 0.5 + 0.5 \times 0.5 = 0.5\]
VARIABLE-SIZE HEAP ALLOCATION
Binned free lists

- Leverage the efficiency of free lists.
- Accept a bounded amount of internal fragmentation.

Bin $k$ holds memory blocks of size $2^k$. 
Allocation for Binned Free Lists

Allocate \( x \) bytes

- If bin \( k = \lceil \lg x \rceil \) is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin \( k' > k \), split it up into blocks of sizes \( 2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k \), and distribute the pieces.

Example
\( x = 3 \Rightarrow \lceil \lg x \rceil = 2. \)
Bin 2 is empty.
Allocate $x$ bytes

- If bin $k = \lceil \lg x \rceil$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k$, and distribute the pieces.

Example

$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.
Bin 2 is empty.
Allocate $x$ bytes

- If bin $k = \lceil \lg x \rceil$ is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin $k' > k$, split it up into blocks of sizes $2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k$, and distribute the pieces.

Example
$x = 3 \Rightarrow \lceil \lg x \rceil = 2$.
Bin 2 is empty.
Allocation for Binned Free Lists

Allocate \( x \) bytes

- If bin \( k = \lfloor \log x \rfloor \) is nonempty, return a block.
- Otherwise, find a block in the next larger nonempty bin \( k' > k \), split it up into blocks of sizes \( 2^{k'-1}, 2^{k'-2}, \ldots, 2^k, 2^k \), and distribute the pieces. *

Example
\( x = 3 \Rightarrow \lfloor \log x \rfloor = 2 \).
Bin 2 is empty.

*If no larger blocks exist, ask the OS to allocate \( x \) more bytes of VM.
Storage Layout of a Program

- **high address**
- **virtual memory**
- **low address**

- **stack**
- **heap**
- **bss**
- **data**
- **text**

- Dynamically allocated
- Initialized to 0 at program start
- Read from disk
- Code

© 2014 Charles E. Leiserson, and Saman P. Amarasinghe
Q. Since a 64–bit address space takes over a century to write at a rate of 4 billion bytes per second, we effectively never run out of virtual memory. Why not just allocate out of virtual memory and never free?

A. External fragmentation would be horrendous! The performance of the page table would degrade tremendously leading to disk thrashing, since all nonzero memory must be backed up on disk in page–sized blocks.

Goal of storage allocators
Use as little virtual memory as possible, and try to keep the used portions relatively compact.
Theorem. Suppose that the maximum amount of heap memory in use at any time by a program is $M$. If the heap is managed by a BFL allocator, the amount of virtual memory consumed by heap storage is $O(M \lg M)$.

Proof. An allocation request for a block of size $x$ consumes $2^{\lceil \lg x \rceil} \leq 2x$ storage. Thus, the amount of virtual memory devoted to blocks of size $2^k$ is at most $2M$. Since there are at most $\lg M$ free lists, the theorem holds. ■
Coalescing

Binned free lists can sometimes be heuristically improved by splicing together adjacent small blocks into a larger block.

- Clever schemes exist for finding adjacent blocks efficiently — e.g., the “buddy” system — but the overhead is still greater than simple BFL.
- No good theoretical bounds exist that prove the effectiveness of coalescing.
- Coalescing seems to work in practice, because heap storage tends to be deallocated as a stack (LIFO) or in batches.
Idea

- Free the programmer from freeing objects.
- A garbage collector identifies and recycles the objects that the program can no longer access.
- GC can be built-in (Java, Python) or do-it-yourself.

Any other Pros and Cons?
Terminology

- **Roots** are objects directly accessible by the program (globals, stack, etc.).
- **Live** objects are reachable from the roots by following pointers.
- **Dead** objects are inaccessible and can be recycled.

How can the GC identify pointers in objects?

- Strong typing.
- Prohibit pointer arithmetic (which may slow down some programs).
Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.
Reference Counting

Keep a count of the number of pointers referencing each object. If the count drops to 0, free the dead object.

What are the pros and cons of reference counting?
Limitation of Reference Counting

Problem
A cycle is never garbage collected!
Limitation of Reference Counting

Problem
A cycle is never garbage collected!
Limitation of Reference Counting

Problem
A cycle is never garbage collected!
Limitation of Reference Counting

Problem
A cycle is never garbage collected!

Nevertheless, reference counting works well for acyclic structures.
STOP--&--COPY
GARBAGE COLLECTION

SPEED
LIMIT
∞
PER ORDER OF 6.172
Graph Abstraction

Idea
Objects and pointers form a directed graph $G = (V, E)$. Live objects are reachable from the roots. Use breadth-first search to find the live objects.

FIFO queue $Q$

```
for (∀ v∈V) {
    if (root(v)) {
        v.mark = 1;
        enqueue(Q, v);
    } else v.mark = 0;

    while (Q != ∅) {
        u = dequeue(Q);
        for (∀ v∈V such that (u,v)∈ E) {
            if (v.mark == 0) {
                v.mark = 1;
                enqueue(Q, v);
            }
        }
    }
}
```
Breadth-First Search
Breadth-First Search
Breadth-First Search

A sequence of vertices labeled a, b, c, d, e, f, g, h, i, j is shown in a graph. The vertex r is placed at the start of a queue Q, and the sequence of vertices is explored breadth-first, starting with r, then visiting b, c, d, e, f, g, h, i, j.

The diagram illustrates the order in which vertices are visited: r, b, c, d, e, f, g, h, i, j. The queue Q is shown with r at the head and the other vertices in the order they are visited.
Breadth-First Search

Graph with nodes labeled a, b, c, d, e, f, g, h, i, j, and r, and edges connecting them. A queue Q labeled r, b, with pointers to head and tail.
Breadth-First Search
Breadth–First Search

The diagram illustrates a breadth-first search algorithm. The queue (Q) contains nodes in the order they are visited. The head and tail of the queue are indicated.

Nodes and edges in the graph show the structure of the search, with nodes representing locations and edges representing connections between them. The search starts at node r and explores all its neighbors before moving on to the next level of nodes.
Breadth-First Search

The diagram illustrates a breadth-first search in a graph. The vertices are labeled from 'a' to 'j', and the edges connect them. The queue (Q) is shown with 'r', 'b', and 'c' as the elements. The head and tail of the queue are indicated.
Breadth-First Search
Breadth-First Search

![Graph Diagram]

Q

r b c d e

head tail

© 2014 Charles E. Leiserson, and Saman P. Amarasinghe
Breadth-First Search

Q

head
tail
Breadth-First Search

A breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores the neighbor nodes first, before moving to the next level neighbors. The BFS algorithm is used in many applications such as network routing protocols, social network analysis, and recommendation systems.
Breadth-First Search

![Graph diagram showing Breadth-First Search with nodes a, b, c, d, e, f, g, h, i, and j, and an example queue Q with elements r, b, c, d, e, f.]
Breadth-First Search

```
Q
```

```
head
tail
```
Breadth-First Search

The diagram illustrates the concept of Breadth-First Search (BFS) in graph theory. BFS explores the graph level by level, starting from the root node. Each circle represents a node, and the arrows indicate the connections between nodes.

At the bottom of the diagram, there is a queue (Q) labeled with letters: r, b, c, d, e, f, g. The queue is used to keep track of the nodes to be visited next. The head and tail of the queue are marked to show the direction of node exploration.

- The root node is r, and the queue initially contains r, b, c, d, e, f, g.
- Nodes are visited in the order they are added to the queue, ensuring all nodes at a given level are visited before moving to the next level.
- BFS explores all the vertices of a graph in breadth-first order, meaning it visits all the vertices at the current depth before moving on to the vertices at the next depth.
Breadth-First Search

Q [r, b, c, d, e, f, g]

head tail
Breadth-First Search

Q
r b c d e f g

head
tail

Done!
Observation
All live vertices are placed in contiguous storage in Q.
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
Copying Garbage Collector

FROM space

next allocation

live

dead

unused
Copying Garbage Collector

FROM space

next allocation

live

dead

unused
Copying Garbage Collector

FROM space

next allocation

live
dead
unused
When the **FROM** space is “full,” copy live storage using BFS with the **TO** space as the FIFO queue.
Copying Garbage Collector

When the FROM space is “full,” copy live storage using BFS with the TO space as the FIFO queue.
Updating Pointers

Since the FROM address of an object is not generally equal to the TO address of the object, pointers must be updated.

- When an object is copied to the TO space, store a forwarding pointer in the FROM object, which implicitly marks it as moved.
- When an object is removed from the FIFO queue in the TO space, update all its pointers.
Remove an item from the queue.
Example

Remove an item from the queue.
Enqueue adjacent vertices.
Example

Enqueue adjacent vertices. Place forwarding pointers in FROM vertices.
Update the pointers in the removed item to refer to its adjacent items in the TO space.
Update the pointers in the removed item to refer to its adjacent items in the **TO** space.
Linear time to copy and update all vertices.
Dynamic Storage Allocation

Lots more is known and unknown about dynamic storage allocation. Strategies include

- buddy system,
- mark–and–sweep garbage collection,
- generational garbage collection,
- real–time garbage collection,
- multithreaded storage allocation,
- parallel garbage collection,
- etc.