Processes, Address Spaces, and Memory Management
Process

A running program and its associated data
Process’s Address Space

0x7fffffff

Stack

Data (Heap)

Globals

Text (Code)
Heap Memory Management
Bitmaps

Memory regions

A B C D

8 16 24 32

11111100 00111000 01111111 11111000

Bitmap
Minimal Units of Allocation

• Break memory up into fixed sized chunks

• Pros:
  – Easier to manage
    (When unsetting bit, surrounding 0s are implicitly coalesced to form a larger contiguous free block)
  – Need just one bit to represent a chunk

• Cons:
  – Internal fragmentation: a chunk being only partly full
  – Difficulty in finding large enough contiguous free block
Linked Lists

Memory regions

A | 0 6
---|-------------------
B | 10 3
---|-------------------
C | 17 9
---|-------------------
D | 26 3
---|-------------------

- 6 4
- 13 4
- 29 3
Reclaiming Freed Memory
Allocation Strategies

• First fit
  – Find the first free block, starting from the beginning, that can accommodate the request
  – Rationale: simple and reasonably fast

• Next fit
  – Find the first free block, starting where the last search left off, that can accommodate the request
  – Rationale: why search all the way from the beginning when it’s unlikely to turn up useful blocks

• Best fit
  – Find the free block that is closest in size to the request
  – Rationale: do not take up a larger block needlessly
Allocation Strategies Continued

• Worst fit
  – Find the free block with the most left over after fulfilling the allocation request
  – Rationale: best fit typically leaves very small blocks that are useless

• Quick fit
  – Keep several lists of free blocks of common sizes, allocate from the list that nearest matches the request
  – Rationale: allocation very fast and wastes little space
  – Challenge: how to coalesce smaller blocks

• All strategies suffer from external fragmentation
  – Having many free blocks that are too small to be useful
  – Need a better coalescing strategy
## Buddy Allocation

Allocation of size 2 in a region of size 16

<table>
<thead>
<tr>
<th>Size 16</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size 8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Size 8</td>
<td>Size 8</td>
</tr>
<tr>
<td></td>
<td>Size 4</td>
<td>Size 4</td>
</tr>
<tr>
<td></td>
<td>Size 4</td>
<td>Size 8</td>
</tr>
<tr>
<td>Size 2</td>
<td>Size 2</td>
<td>Size 4</td>
</tr>
<tr>
<td>Size 2</td>
<td>Size 2</td>
<td>Size 4</td>
</tr>
</tbody>
</table>
Buddy Allocation

Allocation of size 4 in a region of size 16

<table>
<thead>
<tr>
<th>Size 2</th>
<th>Size 2</th>
<th>Size 4</th>
<th>Size 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size 2</td>
<td>Size 2</td>
<td>Size 4</td>
<td>Size 8</td>
</tr>
<tr>
<td>Size 2</td>
<td>Size 2</td>
<td>Size 4</td>
<td>Size 8</td>
</tr>
</tbody>
</table>
## Buddy De-Allocation

Free region of size 2 in a region of size 16

| Size 2 | Size 2 | Size 4 | Size 8 |

Mark region as free

| Size 2 | Size 2 | Size 4 | Size 8 |

Combine with “buddy”

| Size 4 | Size 4 | Size 8 |
# Buddy De-Allocation

Free region of size 4 in a region of size 16

<table>
<thead>
<tr>
<th>Size 4</th>
<th>Size 4</th>
<th>Size 8</th>
</tr>
</thead>
</table>

Mark region as free

<table>
<thead>
<tr>
<th>Size 4</th>
<th>Size 4</th>
<th>Size 8</th>
</tr>
</thead>
</table>

Combine with “buddies”

<table>
<thead>
<tr>
<th>Size 8</th>
<th>Size 8</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Size 16</th>
</tr>
</thead>
</table>
Buddy Location

• Given an allocation at address \texttt{addr}, where is its buddy?
• In the previous example, we had two buddies of size 4 at addresses 0 and 4
• Since we always halve our space, we can force all of our sizes to be powers of 2.
  – Then our two buddies only differ by 1 bit in their number

\[
\text{buddy} = \text{addr} \oplus \text{size}
\]
Buddy Allocation

- Used in the Linux kernel
- Nodes maintained as a binary tree
- Efficient
  - Little external fragmentation (first tries to find empty region before splitting existing region)
  - Coalescing block of size N takes at most $\log_2 N$ steps
Where Do We Store the Nodes?
Memory Management Pitfall

```
#include <stdlib.h>

int main()
{
    char s1[10], *s2;
    s2 = (char *)realloc(s1, 20);
    free(s2);
    return 0;
}
```

- Never mix manual memory management (heap) with automatic management (stack)!

```
>> gcc -g main.c
>> ./a.out
Segmentation fault (core dumped)
>> valgrind ./a.out
[sic]
==26279== Invalid free() / delete / delete[] / realloc()
==26279==    at 0x4A06BE0: realloc (vg_replace_malloc.c:662)
==26279==    by 0x40051C: main (main34.c:6)
==26279==  Address 0x7feffff40 is on thread 1's stack
```
OS-Level Memory Management
Issue: Sharing of Physical Memory Among Multiple Processes

- Translation from logical to physical addresses
Goals for OS Memory Management

• **Transparency**
  – Processes not aware memory is shared
  – Run regardless of number and/or locations of processes

• **Protection**
  – Cannot corrupt OS or other processes
  – Privacy: Cannot read data of other processes

• **Efficiency**
Main memory - An array of M contiguous byte-sized cells, each with a unique physical address

Physical addressing
- Most natural way to access it – Addresses generated by the CPU correspond to bytes in it
- Used in simple systems like early PCs and embedded microcontrollers (e.g. cars and elevators)
A system with virtual addressing

- Modern processors use virtual addresses
- CPU generates virtual address and address translation is done by dedicated hardware (*memory management unit*) via OS-managed lookup table
Virtual Memory Example

- Mapping of virtual addresses to physical memory

**Address space 1**

<table>
<thead>
<tr>
<th>Address</th>
<th>0x0000</th>
<th>0x1000</th>
<th>0x2000</th>
<th>0x3000</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL bits</td>
<td>1 0</td>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
</tr>
</tbody>
</table>

**Page Table for process 1**

<table>
<thead>
<tr>
<th>Base Address</th>
<th>CTRL bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1 0</td>
</tr>
<tr>
<td>4</td>
<td>1 1</td>
</tr>
<tr>
<td>6</td>
<td>1 1</td>
</tr>
<tr>
<td>10</td>
<td>1 1</td>
</tr>
</tbody>
</table>

Physical Memory

Free Page
Free Page
Free Page
Pages

• Unit of memory allocation from the OS
• Efficient (compared to variable sized blocks)
  – Fast to allocate and free (no need to find a ‘fit’)
  – Easier to translate
• Mostly 4KB, but not always
  – 32 bit x86 supports 4KB and 4MB
  – 64 bit x86 supports 4KB, 2MB and 1GB
Page Translation

• How are virtual addresses translated to physical addresses
  – Upper bits of address designate page number

• Happens in MMU address translation hardware in CPU
• Page offsets concatenated instead of added due to fixed size
  ➔ More efficient address translation
Transparency: Separate virtual addr. spaces

- Each process has its own virtual address space
  - OS controls how virtual pages are assigned to physical mem.
  - If OS runs out of physical mem., disk ‘swap’ space is assigned to lesser used pages

Virtual Address Space for Process 1:

Virtual Address Space for Process 2:
Protection: Separate Address Spaces + Permission bits

- Page table entry contains access rights information
  - HW enforces this protection (trap into OS if violation occurs)

Process i:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>SYS</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 6</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 4</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 2</td>
<td></td>
</tr>
</tbody>
</table>

Process j:

<table>
<thead>
<tr>
<th>VP 0:</th>
<th>SYS</th>
<th>READ</th>
<th>WRITE</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>PP 9</td>
<td></td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 6</td>
<td></td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>PP 11</td>
<td></td>
</tr>
</tbody>
</table>

Must be running in kernel (sys) mode

Page tables with permission bits

Physical memory