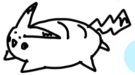
Memory Management I

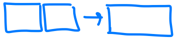


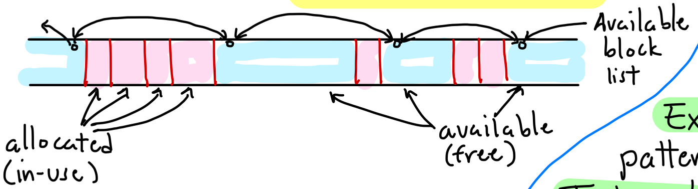
Deallocation Models:

Explicit: (C, C++)
- programmer deletes
- may result in leaks if not careful

Implicit: (Java, Python)
- runtime system deletes
  - Garbage collection
  - Slower runtime
  - Better memory compaction

Explicit Allocation/Deallocation

- Heap memory is split into blocks whenever requests made
- Available blocks:
  - merged when contiguous 
  - stored in available block list



What happens when you do

- new (Java)
- malloc/free (C)
- new/delete (C++) ?

Runtime System Mem. Mgr.

- Stack - local vars, recursion
- Heap - for "new" objects

Don't confuse with heap data structure / heapsort

Fragmentation:

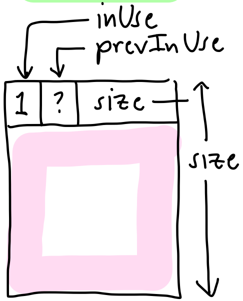
- Results from repeated allocation + deallocation (Swiss-cheese effect)

External: Caused by pattern of alloc/dealloc

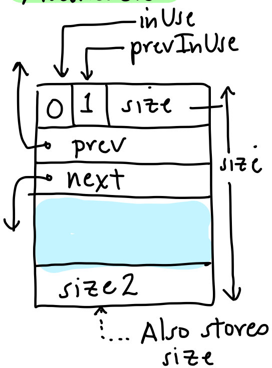
Internal: Induced by mem. manage. policies (not user)

Block Structure:

Allocated:



Available:



Guide:

prevInUse: 1 if prev. contig. block is allocated

prev/next: links in avail. list

size/size2: total block size (includes headers)

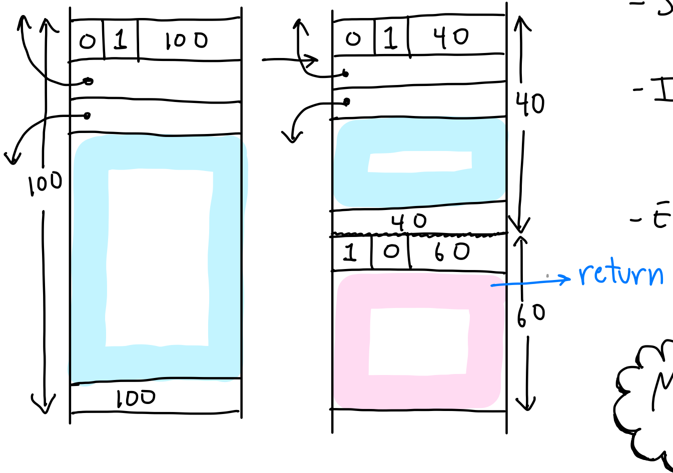
How to select from available blocks?

- **First-fit:** Take first block from avail. list that is large enough
- **Best fit:** Find closest fit from avail list

Surprise: First-fit is usually better
- faster + avoids small fragments

Memory Management II

Example: Alloc b=59



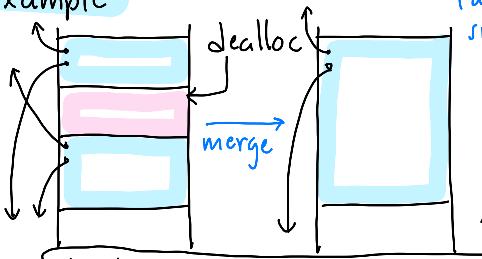
Allocation: malloc(b)

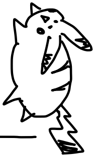
- Search avail. list for block of size $b' \geq b+1$
- If b' close to b : alloc entire block (unlink from avail list)
- Else : Split block

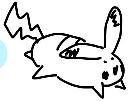
Deallocation:

- If prev + next contiguous blocks are allocated → add this to avail
- Else - merge with either/both to make max. avail block

Example:







Some C-style pointer notation

void* - pointer to generic word of memory

Let p be of type void* :

p+10 - 10 words beyond p

*(p+10) - contents of this

Let p point to head of block:

p.inUse, p.prevInUse, p.size - we omit bit manipulation

*(p+p.size-1) - references last word in this block

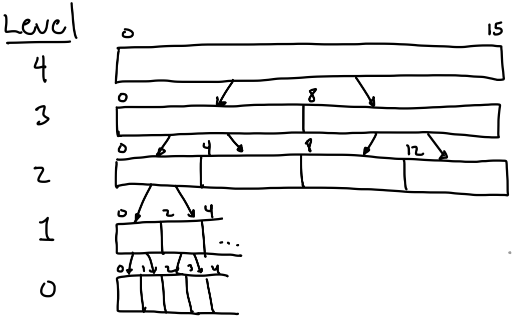
```
(void*) alloc(int b) {
    b += 1                    // add +1 for header
    p = search avail list for block
            size ≥ b
    if ( p == null ) Error-Out of mem!
    if ( p.size - b ≤ TOO_SMALL )
        unlink p from avail. list
        q = p
    else .... (continued)

else {
    p.size -= b               // remove allocation
    *(p+p.size-1) = p.size    // size2
    q = p + p.size            // start of new block
    q.size = b                } // new block
    q.prevInUse = 0           }    header
}

q.inUse = 1
(q+q.size).prevInUse = 1
        // update prevInUse for next
           contig. block
return q+1                    // skip over header
}
```

Memory Management III

Buddy System:

- Block sizes (including headers) are power of 2
- Requests are **rounded up** (internal fragmentation)
- Block size 2^k starts at address that is multiple of 2^k
- k = **level** of a block

Structure:

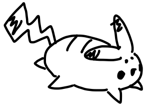


In practice: There is a minimum allowed block size

Buddy system only allows allocations aligning with these blocks

Coping with External Fragmentation

- Unstructured allocation can result in severe **external fragmentation**
- Can we **compress**? Problem of pointers
- By adding more **structure** we can reduce extern frag. at cost of internal frag.



Merging:

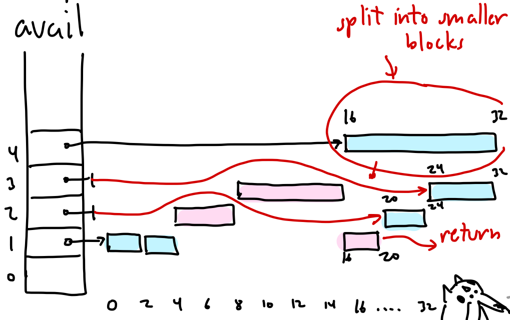
- When two adjacent blocks are available, we don't always merge them
  - → Must have same size: 2^k
  - → Must be **buddies** - siblings in this tree structure

Def:
$$\text{buddy}_k(x) = \begin{cases} x + 2^k & \text{if } 2^{k+1} \text{ divides } x \\ x - 2^k & \text{otherwise} \end{cases}$$

$\equiv$ **$\text{buddy}_k(x) = (1 \ll k) \oplus x$** [Bit manipulation]

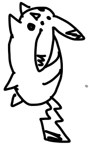
Example:

alloc(2) $\xrightarrow{\text{round up}}$ alloc(4)



Allocation: alloc(b)

- $k = \lceil \lg(b+1) \rceil$ add +1 for header
- if avail[k] non empty - return entry + delete
- else: find avail[j] ≠ ∅ for j > k
  - split this block



Big Picture:

- Avail list is organized by level: avail[k]
- Block header structure same as before except:

  prevInUse } not needed
  size2