Memory Management

- Memory management - allocation of memory chunks to programs and processes
  - 1) Operating system requiring 50Kb to store a program to be executed
  - 2) High level language program requesting block of memory for an instance of a structure
- Heap - portion of memory to be managed - M[0..N-1].
  - blocks are allocated from the heap
  - when a block has been allocated it is reserved or in use
  - blocks can later be freed or deallocated

Factors determining choice of memory management scheme

- Blocks of fixed size versus blocks of various sizes.
  - Some languages (LISP) usually request blocks of fixed size to build lists
  - Operating systems request blocks of various sizes to store programs
  - C++ programs require blocks of different sizes for different structures
- Linked blocks versus unlinked blocks.
  - If A and B are blocks of allocated memory, and A contains a pointer to B, then B cannot be moved without updating references to B from within A.
    - Might need to move B to create a block of a large size
  - Leads to dangling pointers

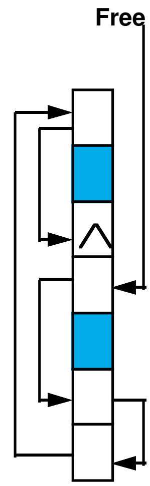
Memory management factors

- Small blocks versus large blocks.
  - Small blocks can be
    - moved easily
    - initialized efficiently since time spent is proportional to size of the block
- Time versus memory.
  - Trade-offs between utilization of heap and complexity of managing heap.
- Explicit versus implicit release.
  - Operating system can explicitly release blocks of programs no longer running
  - Many programs return storage implicitly through “garbage collection.”

Memory management factors

- Scheduled versus non-scheduled release.
  - Is information available about the relative order of allocation and deallocation requests?
  - Example: Management of run time stack - blocks are released in a last-in-first-out fashion.
- Initialized versus unitialized blocks.
  - Initialize large structuresto value specified by language semantics or programmer specification
  - Blank out memory so a program cannot read the results of a previous occupant of memory

Records of a single size



- Suppose all requests are for blocks of size k
- Partition heap once into floor[n/k] blocks of size k. Let T[0..m-1] be those blocks, where m = floor[n/k]
- At any time, some blocks in T are free and some are in use.
- Storage management
  - Maintain a free list of blocks in T - arrange free blocks in a singly linked list
  - List is accessed as a stack - allocation is a POP and de-allocation is a PUSH. So, allocation takes O(1) time.
  - How can we determine when to deallocate implicitly?

Reference count method

- Reference count - count of the number of pointers that point **to** a cell
  - when a new copy of the address of a cell is created, we increment the cell's reference count
  - when a copy is destroyed (say by a pointer assignment) we decrement the reference count
  - when the reference count goes to 0, we de-allocate
  - when cell, X, is deactivated, the reference count of any cell pointed at by X is decremented.

Disadvantages of reference counts

- Requires storage in each active cell for the reference count
  - Unless the maximum number of references is known beforehand, can overflow reference count field
- Doesn't work for circular lists
  - If X and Y point to one another, then even if there are no outside references to either, their reference counts will never go to 0.

Mark and sweep garbage collection

- Wait for the free list to become empty, and then scavange memory for unused cells.
- Associate a mark field (one bit) with each cell. Initially all mark fields are **off**.
- A marking process will set the mark fields **on** for all cells that can be reached by some chain of pointers beginning outside of T
- Memory is then scanned, and all cells with their mark fields off are collected into the free list. At the same time, the mark fields of used cells are turned off to prepare for the next round of garbage collection.
- Big question: How do we do the marking?

Marking

- Marking algorithms are like tree traversals except

  1) cells can have many pointers and not just 2 - although we will assume they all have the same number

  2) cells may be linked into graphs. But if a cell can be reached by more than one path, it is marked the first time it is encountered so that it is not explored when encountered along other paths.

- Let C_0, C_1, ...C_{k-1} be the k pointers stored in any cell
- Atom(P) is true if P points outside the heap (probably back to a named program variable).

Recursive, depth-first marking

- Preorder versions of recursive traversals
  - preorder is important so we mark a node before marking its children - this prevent re-entering circular structures

```
FullyRecursiveMark(pointer P);
if not Atom(P) and not marked(P) then
  Mark(P)
  for j from 0 to k-1 do
    FullyRecursiveMark(Cj(P))
```

- Problem - stack memory needed can be as large as number of links traversed before encountering an atom. But when garbage collection is called, memory is gone!

Link-Inversion Schorr-Deutsch-Waite Marking

- Store the stack as a pointer chain in the structure to be marked - algorithm will use roof[logk]•h bits for its "stack", where h is the length of the longest pointer chain from the root to any cell.
- Generalization of binary tree traversal with link inversion. Differences:

  1) Instead of a tag bit to remember whether we descended through left or right pointer, need a field large enough to store the index (0..k-1) of the link field traversed. Can be either kept as a field in the cells (bad idea!) or as an auxiliary stack.

  2) Ascend and descend code now indexed by pointer

Ascend and descend code

- descend via C_j (simultaneous assignment)

  P <-- Q

  Q <-- C_j(Q)

  C_j(Q) <-- P

- ascend via C_j (sim. assignment)

  Q <-- P

  P <-- C_j(P)

  C_j(P) <-- Q

```
procedure LinkinversionMark(
   pointer Q):
{Mark all cells reachable from Q}
P <-- Nil;
S <-- MakeEmptyStack()
repeat forever
   if Q ≠ Nil and not Atom(Q)  and
   not Marked(Q) then
      Mark(Q)
      Push(0,S)
      descend via C0
else if P = Nil then return
else
   j <-- Pop(S)
   ascend via Cj
   j <-- j+1
   If j < k then
      Push(j,S)
      descend via Cj
```

Other garbage collection algorithms

- Link inversion algorithms that take constant space, independent of the length of the longest path from the root to any cell.
  - still use a mark bit in each cell
  - require much more computation that stack-based link inversion
- Copying algorithms - break memory into two parts, and allocate storage from one. When that segment of memory is fully utilized, copy and compact non-garbage cells from one area to the other.
  - algorithms do not touch garbage
  - but must have a lot of memory

Cautions on garbage collection

- System performance usually degrades quickly once garbage collection is required
  - not much available storage is really available, so system spends a lot of time marking or copying to reclaim only a small amount of storage
  - almost immediately have to garbage collect again, as more memory is consumed
- Incremental garbage collection
  - spend a fixed amount of time collecting garbage
  - therefore, collect only a little garbage at a time.

Records of various sizes

- If blocks can be of different sizes, then we need a more complex memory management mechanism.
- **Method 1** :Allocate in address order from a single block of large memory.
  - when heap is exhausted, some blocks may have been de-allocated
  - **compress** them to get one big block, and continue

Records of various sizes

- **Method 2:** Keep a single list of blocks of all sizes.
  - If a request for a block of size k is encountered, find a block of size h > k, allocate k words and return a block of size h-k to free storage. The set of blocks is called the storage pool.
- Two main problems:

  1) the leftovers (h-k) might be too small for anything

  2) when a block is deallocated, it may need to be merged with an adjacent free block

Records of various sizes

- **Method 3**: Keep several pools of free blocks, according to size.
  - allocate by choosing a block from the correct pool
  - need a subdivision strategy that allows easy construction of large(st) blocks as memory is de-allocated.
  - "Buddy system"

Method 1: Compression

- Major problems:
  - 1) time it takes to move blocks within memory
  - 2) adjustment of pointers to blocks that have moved.
- Algorithm assumptions:
  - 1) The heap M[0..N-1] is partitioned into blocks, some of which are free and others in use.
  - 2) Marked(P) is true if P points to a block that is in use, and false otherwise
  - 3) First is the address of M[0] and Last is address for M[N-1].
  - 4) Each block has a size field; the block after the one beginning at address P begins at address P + Size(P)

Compression

- Assumptions

  5) Pointer fields of a block in use are known in advance (for updating references)

  6) Each block in use contains a pointer field **forwardingaddress** that is reserved for use by the compaction algorithm. The value of this field will be computed before any blocks are moved.

- Algorithm will use the follwing abbreviation:

  for each block P, in address order, do for

  P <-- First

  while P ≤ Last do

  ******

  P <-- P + size(P)

Compaction algorithm

procedure Compact;

{Compact blocks into low memory addresses, adjusting pointers; marking of blocks in use has already been done}

{Compute forwarding addresses}

Dest <-- First

for each block, P, in a-order, do

if Marked(P) then

ForwardingAddress(P) <-- Dest

Dest <-- Dest + Size(P)

{Adjust internal links}

for each block, P, in a-order, do

if Marked(P)then

for each pointer field Link do

Link(P) <--

ForwardingAddress(Link(P))

{move the cells}

for each block, P, in a-order, **do**

if Marked(P) **then**

Copy **Size**(P) bytes beginning at P to **ForwardingAddress**(P)

•**Observations**:

1) algorithm is linear in size of memory compacted

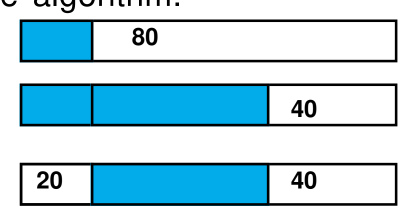
2) but, practically, it takes three scans over the memory to complete(four counting the initial marking)

Method 2: Blocks of various sizes

- If blocks are of different sizes, and some blocks are very large, compaction is not a good solution because it might get called too often.
- Assume explicit deallocation
- Strategy for selecting blocks, subdividing them and recombining them determined by:
  - Memory utilization - do not fail to satisfy a request if the aggregate amount of memory in use is only a small percentage of the heap size
  - Memory overhead - the memory manager data structures should be small compared to the heap
  - Time efficiency - allocation and deallocation should be fast

It's not a pefect woρld

- Not all sequences of requests for memory will be satisfied by any memory management algorithm, even if it is possible to satisfy them by some variation on the algorithm.
- Example: Heap of size 100
  - B_1 <-- allocate (20)
  - B_2 <-- Allocate (40)
  - Free(B_1)
  - B_3 <-- Allocate (50)



- If we had allocated B_2 from the end, then we could have satisfied the last request
  - But, generally, deciding if a (known) sequence of requests is satisfiable is NP-complete

Fragmentation

- Two blocks of size s_1 and s_2 are never more useful than one larger block of size $s_1 + s_2$.
- Allocation strategy should try to minimize the number of free blocks, and maximize their sizes
- External fragmentartion - splitting of free storage into a relatively large number of relatively small free blocks.
  - External fragmentation can be combatted by not subdividing a free block if the leftover space is too small to be useful - just allocate the larger block
  - smaller block would clutter free list.

Fragmentation

- Internal fragmentation - distribution of significant amount of free storage in allocated blocks
  - always some "wasted" space because blocks require fields for use by memory manager - size, pointers to other blocks, etc.

Summary

- Maintain a pool of free blocks
- Develop an allocation strategy for assigning a block to a request
  - retain unused portion of block for use by other requests
  - must decide on a data structure for the pool of free blocks. Can be organized either by size or by location or by both.
- Develop a freeing strategy for returning a freed block to the pool, and coalescing it with other free blocks to form largest possible free blocks

Allocation strategies

- Two basic strategies:

  1) **best fit** allocates a request for a block of size n by finding the smallest block larger than n.

  2) **first fit** allocates using the first block found of size at least n.
- Best fit
  - implement pool of free blocks as a linked list ordered by increasing size
  - first block on this list whose size is ≥ n is the best fit
  - on average, half the list is scanned to find the best fit.
  - must scan list again to put leftover block back into free list.
  - if list is maintained in address order, then entire list would have to be scanned to find the best fit, but reinsertion of leftover fragment is avoided.

Allocation strategies

- Use a linked list representation of the pool of free blocks
- Problem - blocks near the beginning of the list get subdivided more often then blocks near the end
- **Solution**: The **roving pointer** - begin search for a block at the point where the previous search ended.
  - Small blocks will not accumulate at any one area of the list.
  - A block that has just been subdivided, or was too small to fulfill the previous request, will not be considered again until after all blocks have been examined. This gives it time to coalesce with freed blocks

Data structures for freeing

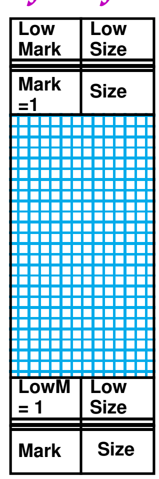
- What fields of information would we need to implement block deallocation and coalescing efficiently?
  1) Mark field = 0 for a free block and 1 for a block in use
  2) Size field
  3) Next and Prev fields to maintain a doubly linked list of free blocks.
- Only needed when the block is free - process can use the fields when the block is in use
- Makes subdivision easy - allocate from the end and set Size <- Size - n

Data structures for freeing

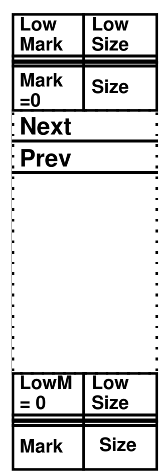
- Do not depend on assumptions about order of blocks in free list to support the Free operation
  - Either size order or address order will entail search of the free list, and we want to avoid this search
- Suppose we free the block B.
  - To determine if the next higher (in address) block is free, we look at Mark(B + Size(B))
  - Determining if the previous block is free is harder
  - Replicate the mark field at the **end** of each block
  - So, can examine a field at a **fixed displacement** before the beginning of block B to determine if the previous block is free
  - This is called **LowMark(B)** .

Data structures for freeing

- If B's lower neighbor is free, then its Size field is also replicated near its end
  - Call this field **LowSize(B)** .
- These fields are stored in the block prior to B, and not in B itself



Block in use



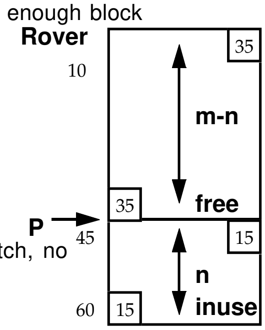
Free block

• This is called the Boundary Tag Method

Allocation algorithm

function BoundaryTagAllocate (**integer**:n):**pointer**
{return address of a free block of size n, or nil if no large enough block is available}

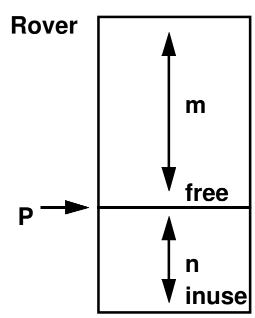
```
SaveRover <-- Rover
repeat
        m <-- Size(Rover)
        if m < n then
                Rover <-- Next(Rover)
        else
                if m-n < delta then  {almost exact match, no leftover}
                  P <-- Rover
                  DoublyLinked Delete (P)
                else          {inexact match - divide the block}
                  P <-- Rover + m - n
                  Size(Rover) <-- LowSize (P) <-- m-n
                  Size (P) <-- LowSize (P+n) <--n
                  LowMark(P) <-- 0
```



m = 50
n = 15

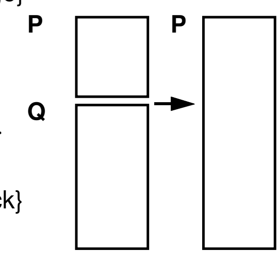
Allocation algorithm

```
Mark (P) <-- 1
LowMark (P + Size(P)) <-- 1
Rover <-- Next (Rover)
Return (P)
until Rover = SaveRover
return nil
```



Freeing algoritm

```
procedure BoundaryTagFree (pointer P)
{deallocate the block pointed to by P}
if LowMark (P) = 0 then {Preceding block is free, merge}
   Q <-- P
   P <-- P - LowSize (P)
   Size (P) <-- Size (P) + Size (Q)
else  { Preceding block in use; mark this block as free}
   Mark (P) <-- 0
Q <-- P + Size (P)  { Q <-- address of subsequent block}
if Mark(Q) = 0 then  {merge with following block}
   Size (P) <-- Size (P) + Size (Q)
   if Rover = Q then Rover <-- P
   DoublyLinkedDelete (Q)
   Q <-- P + Size (P)  {P points to block to be
   placed in free; Q points just after the end of this block}
```




```
LowSize(Q) <--
   Size(P)
LowMark(Q) <-- 0
DoublyLinkedInsert(P,
   Free)
```


Method 3: Buddy systems

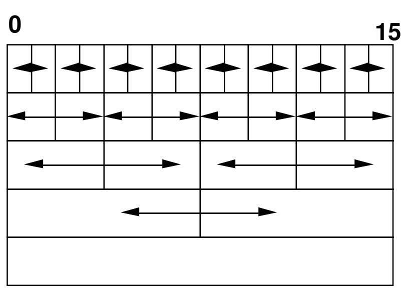
- Trade external fragmentation for internal fragmentation
  - only blocks of small number of fixed sizes are allocated. Additionally, the starting addresses for the blocks are constrained.
  - a request for a block of size m is rounded up to the smallest size n that is supported
- When a block is freed, it may not be merged with an adjacent free block
  - every block has a "buddy block" of the same size
  - when a block is freed, if its buddy is free, it is merged with its buddy to create a larger block
  - this larger block may be merged with its buddy ...

Binary buddies

- All allocatable blocks have sizes that are powers of 2.
- Heap size is also a power of 2, say 2^m.
- The allowable block sizes are 2^m, 2^{m-1}, ..., 2^1, 2^0
  - this means that there are m+1 Free lists, with list headers Free[m], Free[m-1],..., Free[0].
- Each block of size 2^k begins at an address which is a multiple of 2^k - begins at $p*2^k$
  - ends at $(p+1)*2^k - 1$
  - $0 \le p \le 2^{m-k}$

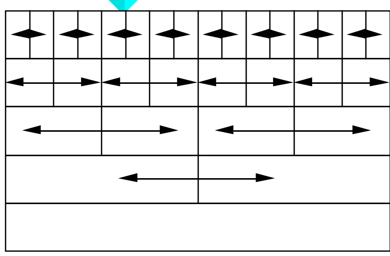
Binary buddies

- Who is the buddy of a block of size 2^k?
  - it is the **other** block contained in the enclosing block of size 2^{k+1}
- So, the largest block has no buddy



- Blocks of size 4 begin at addresses 0, 4, 8, 12 - all multiples of 4
- There are 4 = $2^{(4-2)}$ blocks of size 4 in a heap of size 16
- In a heap of size 32 there would be $2^{(5-2)}$ = 8
- The buddies are (0,4) and (8,12)

Binary buddies

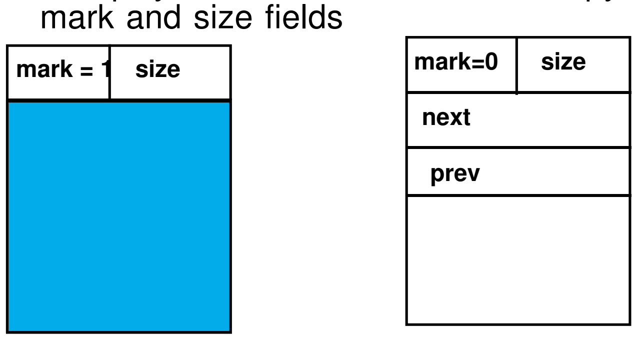


- Block of size 4 beginning at P=4 = 0100. k'th bit = 1 (start from right at 0, so buddy is at 0000)
- Block of size 2 starting at P=10 = 1010 has buddy at 1000 = 8.
- Buddy of a block of size 2^k at address P is the block of size 2^k beginning at $P \oplus 2^k$

- If block N of size 2^k begins at address P, then B's buddy begins at either
  - P - 2^k or P + 2^k
- Easy to tell by looking at k'th bit of P. If k'th bit is 1, then buddy is at P-2^k. If k'th bit is 0, buddy is at P+2^k
  - So just **complement** bit k in the binary representation of B's address
  - But this is just the **exclusive or** of the block's position and its size!

Binary buddies

- Free lists are kept as doubly linked to facilitate deletions
- Fields are same as boundary tag method, except you don't need the extra copy of the mark and size fields



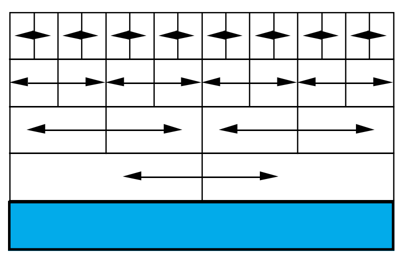
Binary buddies - allocation

function BinaryBuddyAllocate (**integer** n): **pointer**
{Return pointer to a block whose size is the next power of 2 > n}
{ Return nil if no sufficiently large block exists}

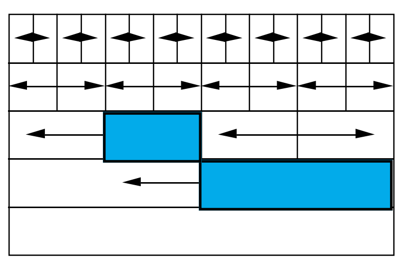
j <--k <-- roof(log n)
while j ≤ m **and** IsEmptyList (Free[j]) **do** j <-- j+1
if j > m **then return** nil
P <-- Free[j]
DoublyLinkedDelete (P)
while j > k **do**
  j <-- j-1
  Q <-- P + 2^j
  Mark(Q) <-- 0
  Size(Q) <-- j
  DoublyLinkedInsert (Q, Free[[j]) {Inserts Q onto an **empty** list Free[j]}
Mark(P) <-- 1
Size(P) <-- k
return P.

Example

- Free[4] = 0
- Free[3] = Free[2] = Free[1] = Free[0] = NIL
- Allocate a block of size 3
- Fill with a block of size 4
  - j <-- k<-- 2
  - j gets bumped up to 4 and the block of size 16 starting at P = 0 is used



• Place the block of size 8 starting at position 8 (0 + 2^3) into Free[3]
• Place the block of size 4 starting at location 4 (0 + 2^2) into Free[2]



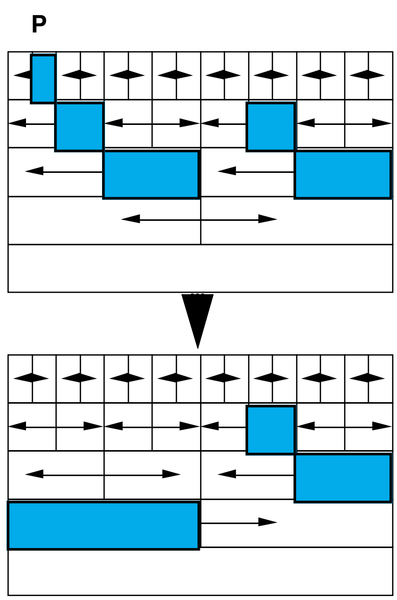
Block freeing

- May reverse the splits that occured during allocation
  - for example, if we allocate a block of size k and then immediately free it, we restore free lists to their original state.
- If the freed block's buddy is free, it is removed from free list and merged to create a larger block.
- This is repeated until either the buddy is in use, or the block becomes the entire heap.
- Can have at most log(n) frees, so freeing is O(log n) in the worst case.

Memory management - 41

Block freeing

procedure BinaryBuddyFree (**pointer** P):
{Free the block of size 2^k beginning at P}
k <-- Size(P)
while k < m **and** Mark(P EOR 2^k) = 0 **and**
  Size(P EOR 2^k) = k **do**
  {P's buddy is free, so merge}
  Q <-- P EOR 2^k
  DoublyLinked Delete (Q)
  if Q < P **then** P <-- Q
  k <-- k + 1
Mark(P) <-- 0
Size (P) <-- k
DoublyLinkInsert (P, Free[k])



Memory management - 42