Real-time Operating Systems and Systems Programming

Understanding Memory (Heap)

Heap

- Section of memory for dynamic structures
- Bounded by brk pointer in kernel
- Function for allocation and deallocation:
  void *sbrk()
- Normally not used directly
  alloc(), malloc(), calloc(), free()
- Allocators divide heap into blocks

Why dynamic allocation?

- Programs often know the amount of memory needed and sizes for data structures runtime
- RTOS note: you might still prefer static allocation for predictability

Constraints for allocators

- Handling arbitrary request sequences
- Making immediate responses for requests
- Use only heap
- Block alignment must be kept
- Cannot modify allocated blocks

Fragmentation problem

- Allocation and deallocation sequences can result in “holes”.
  - Internal fragmentation: the holes within memory blocks themselves
  - External fragmentation: happens when there would be enough free memory for a block, but a single block cannot hold it.

Implementation

- Most naïve: just allocate, never reuse
- More clever:
  - Organize free blocks
  - Deal with placement of blocks
  - Splitting of blocks
  - Joining of blocks

Organizing blocks

- Implicit free list
- Blocks have headers which include
  - Block size
  - Allocated/Free field
- Header size: 1 word
- Return the pointer to content, use header internally

Header

- Due to alignment, the block sizes are multiple of 8
  - 3 lowest order bits are free!
  - Last bit used for free/allocated
- Terminating header with size 0
- “Contents” are located on double word alignment boundaries
- We have minimum block size

Alignment trick

```
typedef long Align;

union header {
    struct {
        union header *ptr;
        unsigned size;
    } s;
    Align x;
}
typedef union header Header;
```

Where to place?

- When searching for a free block, one can have policies for placement:
  - First fit – end of list is often free; fragments
  - Next fit – spreads allocation; fragments worse
  - Best fit – good, but slower

Should we split?

- Option to use entire block
- Or split
- If the fit is “good”, do not split

How to get free memory?

- Ask for more (mmap() or sbrk())
- Merge adjacent blocks upon freeing
  - Can also be done when needed

Merging

- Merging next block is simple: just add
- How to find the previous block?
  - Boundary tags (block footer)
  - Block header has 2 free bits, use one to show that the previous block is free (then only free blocks have footers)

Implementation details

- Initialize block list
- Decide policies
- Blocks may behave like data structures (linked or double linked lists)
- For faster allocation, keep free lists
- Segregation of free lists (see next)

Simple Segregation

- For memory storage, a memory class will store blocks up to size X (malloc({17-32}) → 32)
- If new memory is needed, allocate a page
- Split it into equal blocks sized according to the storage class
- Do not merge blocks
- Link them into free list
- Problems: extreme fragmentation
  (sounds like a grenade)

Segregated fit

- Allocator has an array of free lists, according to size classes
- Allocate according to class, first fit
- Split if needed
- If not found, search larger classes or ask more
- Thought to work well since GNU malloc() behaves like this

Array memory management

- Dynamically defined 2d array needs 2 allocations with malloc() and some tricky pointer initialization
- NOTE: due to issues some sources suggest using calloc() for any reasonable allocations on non-embedded hardware

Fixed 2d array

- ***Stack* allocation**
  Allocation: int fixed[50][100];
- **Access:** fixed[5][9] = 1; /* or */
  fixed[0][5*100+9] = 1; /* or */
  fixed[1][4*100+9] = 1; /* etc */
- **Initialization:**

  for(i=0;i<50;i++) for(j=0;j<100;j++) fixed[i][j] = 0; /* slooow */

  int *ptr = fixed[0]; int *end = fixed[49]+99; *end = 0; while(ptr != end) *ptr++=0;
- **Passing to a function:**
  Prototype: void func(int fixed[50][100]);

Dynamic 2d array

- Stored in *heap.*

Allocation

```
int **dynamic;
dynamic = (int**)malloc(sizeof(int*)*50);
dynamic[0] = (int*)malloc(sizeof(int)*50*100);
for (i=1;i<50;i++) dynamic[i]=dynamic[i-1]+100;
```

Access

```
dynamic[5][9] = 1; /* või */
dynamic[0][5*100+9] = 1;      /* või */
dynamic[1][4*100+9] = 1;      /* jne... */
```

Initialization

```
int *ptr = dynamic[0];
int *end = dynamic[49] + 99; *end = 0;
while (ptr !=end) *ptr++=0;
```

Prototype

```
func(int** vec);
```

Dynamic 2d array

- Stored in *heap.*

Allocation

```
int **dynamic;
dynamic = (int**)malloc(sizeof(int*)*50);
dynamic[0] = (int*)malloc(sizeof(int)*50*100);
for (i=1;i<50;i++) dynamic[i]=dynamic[i-1]+100;
```

Access

```
dynamic[5][9] = 1; /* või */
dynamic[0][5*100+9] = 1;      /* või */
dynamic[1][4*100+9] = 1;      /* jne... */
```

Initialization

```
int *ptr = dynamic[0];
int *end = dynamic[49] + 99; *end = 0;
while (ptr !=end) *ptr++=0;
```

Prototype

```
func(int** vec);
```

Notes for the test

- i++, ++i
- static
- a[1], a+1, *a+1, *(a+1), &a[1]
- {}
- x ? 1 : 0;
- 2,3 2.3
- case
- memory: struct , union, 2d array