Heap memory allocation: `malloc` and `free`

John Lapinskas, University of Bristol

The two key functions

We've talked about memory allocation on the stack, which is incredibly powerful, but only for variables whose sizes are known at compile-time.

To handle run-time memory allocation in a C-like way, we need to write two functions. In C they're called `malloc` and `free`, in Jack they're called `Memory.alloc` and `Memory.deAlloc`. For this video, they will be:

- `alloc`. Takes one `int` argument, named `size`. Allocates a memory segment of `size` words in heap memory (addresses 0x800–0x3FFF) and returns the base address of the segment. Returns -1 if no segment can be allocated.
- `deAlloc`. Takes one `int` argument, named `base`. Frees the segment of memory with base address `base` (as returned by `alloc`).

Let's assume we're working in a high-level language rather than Hack VM, and focus on the algorithms themselves rather than on writing code.

What we need

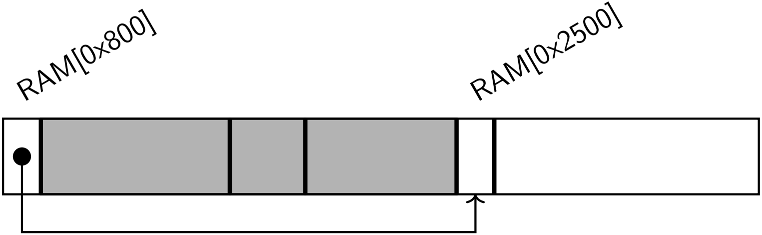
- `int alloc(int size)`. Allocates `size` words in heap memory and returns the base address of the new segment, or -1 as an error.
- `void deAlloc(int base)`. Frees the segment with base address `base`.

Our requirements are:

- After we have assigned `base = alloc(size)`, then our program should be free to write whatever it wants to $\text{RAM}[\texttt{base}], \ldots, \text{RAM}[\texttt{base} + \texttt{size} - 1]$.
- So subsequent calls `alloc(size2)` should not return any value `base2` with $\texttt{base2}, \ldots, \texttt{base2} + \texttt{size2} - 1$ intersecting $\texttt{base}, \ldots, \texttt{base} + \texttt{size} - 1$...
- at least until after we have called `free(base)`.
- We don't care what happens if our program writes to e.g. $\text{RAM}[\texttt{base} - 1]$ or $\text{RAM}[\texttt{base} + \texttt{size}]$, or writes to e.g. $\text{RAM}[\texttt{base}]$ after calling `free(base)`. (In a modern system, the OS would respond by generating a segfault.)

We'll step through four progressively better algorithms.

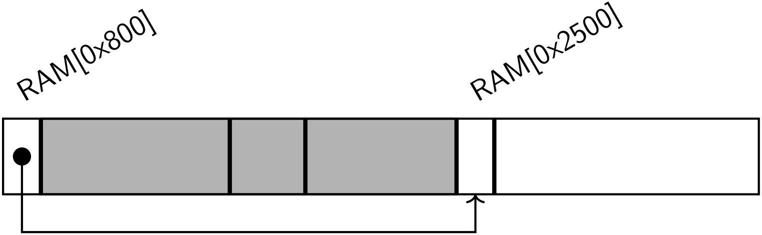
Attempt 1



- In RAM[0x800], we store a pointer to the first unallocated memory address.

Attempt 1

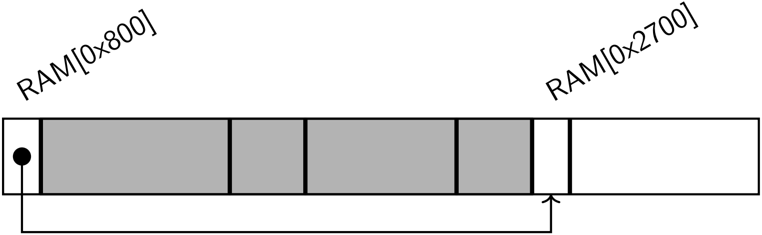
New call `alloc(0x200);`



- In RAM[0x800], we store a pointer to the first unallocated memory address.
- On a call `alloc(size)`, we return RAM[0x800], then add `size` to it so that it points to the new first unallocated memory address.

Attempt 1

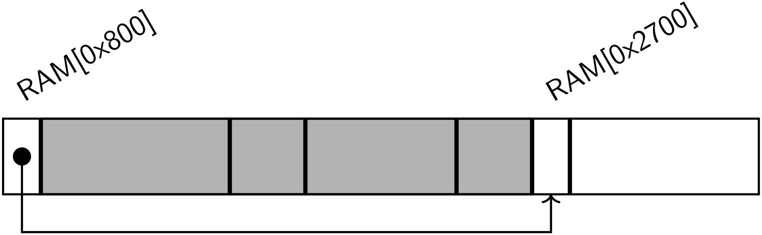
New call `alloc(0x200);`



- In RAM[0x800], we store a pointer to the first unallocated memory address.
- On a call `alloc(size)`, we return RAM[0x800], then add `size` to it so that it points to the new first unallocated memory address.

Attempt 1

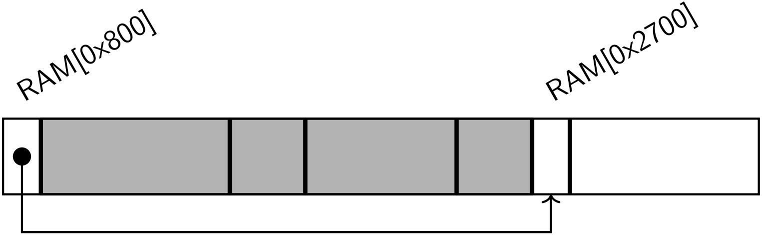
New call `deAlloc(0x2500);`



- In RAM[0x800], we store a pointer to the first unallocated memory address.
- On a call `alloc(size)`, we return RAM[0x800], then add `size` to it so that it points to the new first unallocated memory address.
- On a call to `deAlloc`, we do nothing.

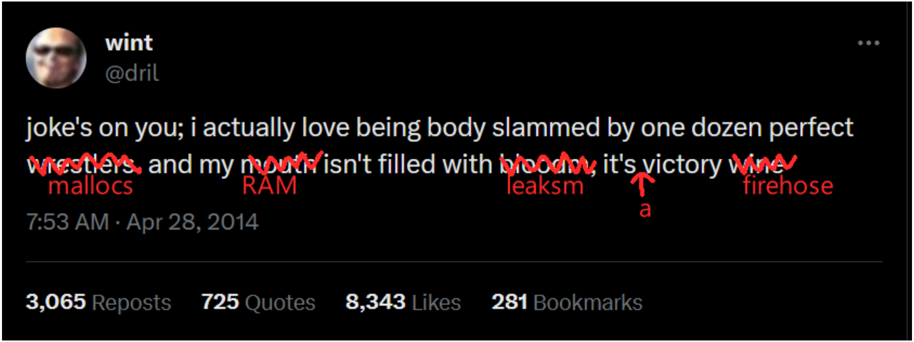
Attempt 1

New call `deAlloc(0x2500);`



- In RAM[0x800], we store a pointer to the first unallocated memory address.
- On a call `alloc(size)`, we return RAM[0x800], then add `size` to it so that it points to the new first unallocated memory address.
- On a call to `deAlloc`, we do nothing.
- This is the "basic" allocation algorithm in 12.1.3 of Nisan and Schocken.

Attempt 1: 💩 💩 💩 💩 💩



- In RAM[0x800], we store a pointer to the first unallocated memory address.
- On a call `alloc(size)`, we return RAM[0x800], then add `size` to it so that it points to the new first unallocated memory address.
- On a call to `deAlloc`, we do nothing.
- This is the "basic" allocation algorithm in 12.1.3 of Nisan and Schocken.

Attempt 2

OK, so we clearly need to store information somewhere about how many free/allocated segments we have and how long they are.

But that's going to be a list in memory of unknown size. Wasn't that the problem we were trying to solve in the first place?

Attempt 2: Storing information inside segments

OK, so we clearly need to store information somewhere about how many free/allocated segments we have and how long they are.

But that's going to be a list in memory of unknown size. Wasn't that the problem we were trying to solve in the first place?

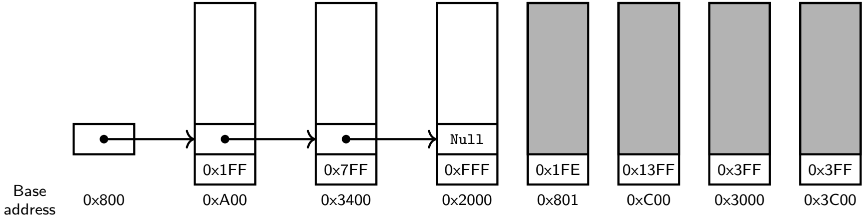
Idea: Store this information inside the segments themselves!

Attempt 2: Storing information inside segments

OK, so we clearly need to store information somewhere about how many free/allocated segments we have and how long they are.

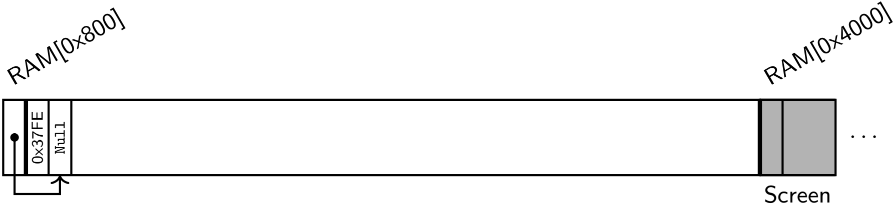
But that's going to be a list in memory of unknown size. Wasn't that the problem we were trying to solve in the first place?

Idea: Store this information inside the segments themselves!



- Every segment contains its usable size as the first word.
- Free segments are arranged in a linked list, storing pointers in the second word of each free segment. RAM[0x800] contains a pointer to the first segment of the list.
- Notice that [usable size] = [actual size] − 1! We lose a little space to overhead.

Behaviour of attempt 2



`alloc(size)` will:

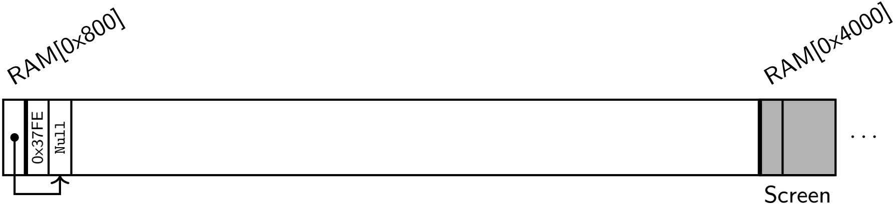
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFF)`



`alloc(size)` will:

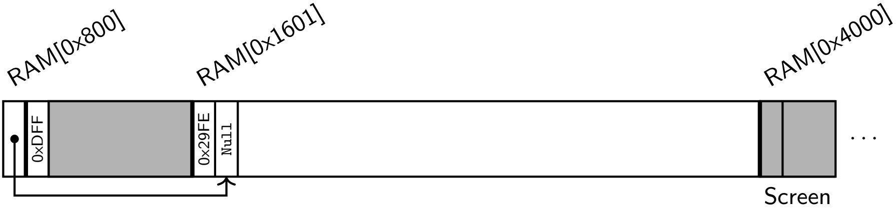
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFF)` returns 0x802



`alloc(size)` will:

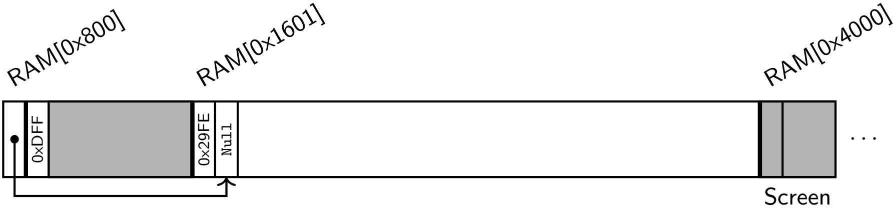
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFF)`



`alloc(size)` will:

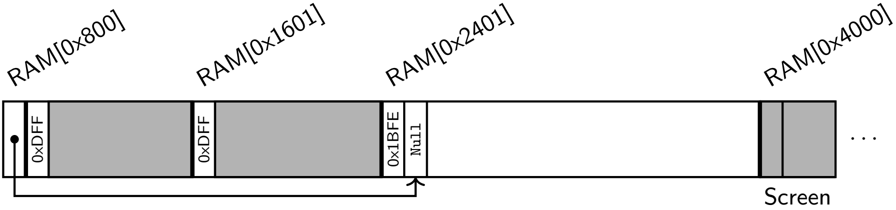
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFF)` returns 0x1602



`alloc(size)` will:

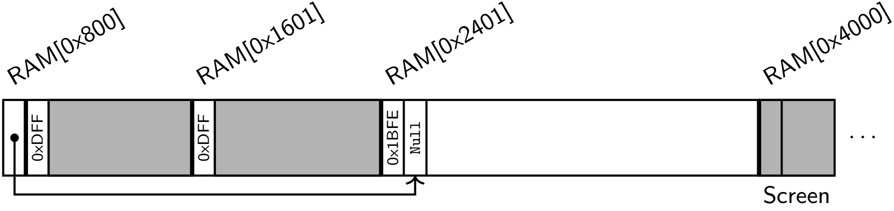
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFF)`



`alloc(size)` will:

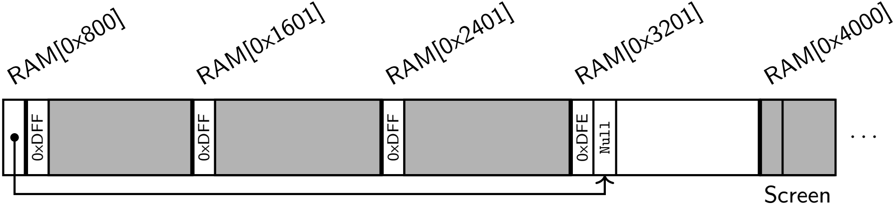
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFF)` returns 0x2402



`alloc(size)` will:

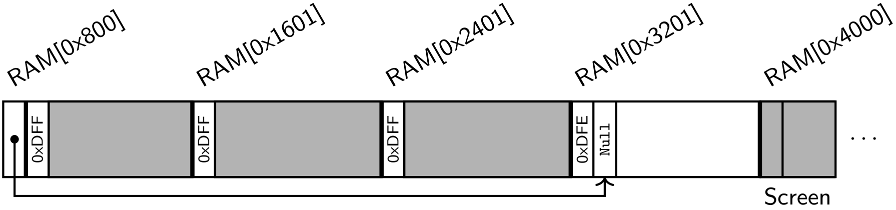
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFE)`



`alloc(size)` will:

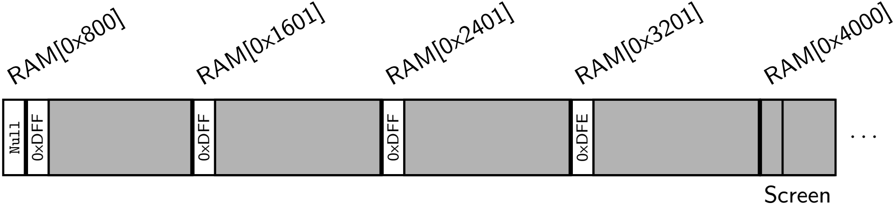
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0xDFE)` returns 0x3202



`alloc(size)` will:

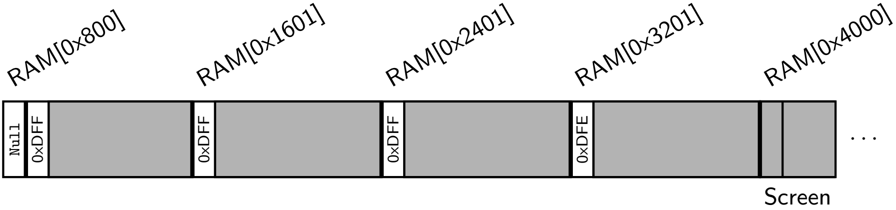
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x802)`



`alloc(size)` will:

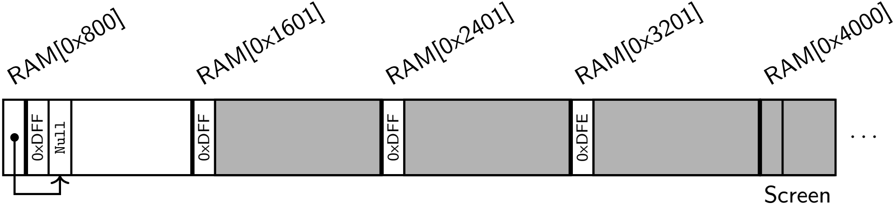
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x802)`



`alloc(size)` will:

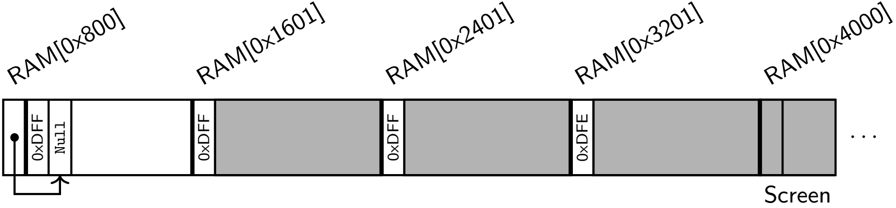
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x1602)`



`alloc(size)` will:

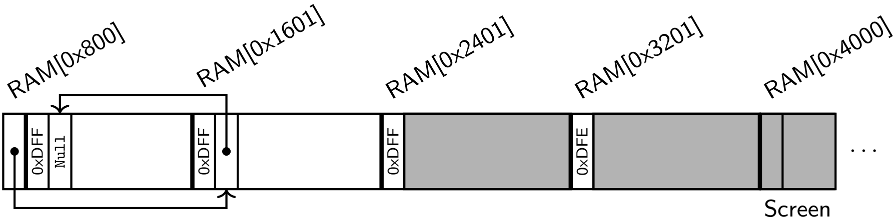
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x1602)`



`alloc(size)` will:

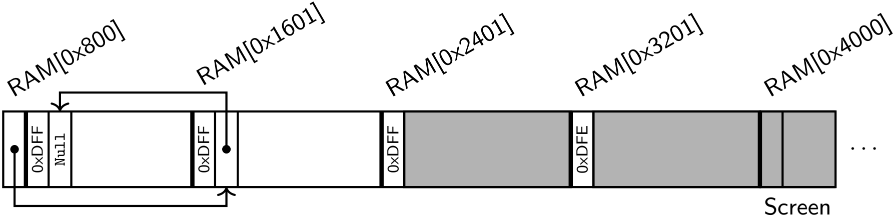
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x3202)`



`alloc(size)` will:

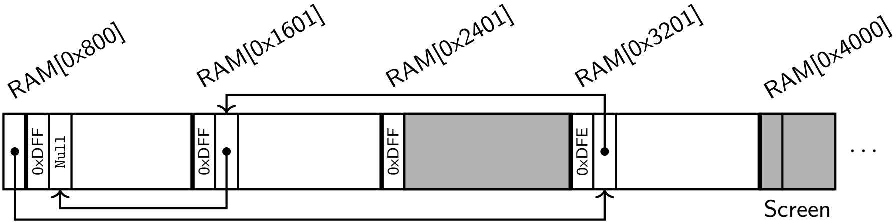
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x3202)`



`alloc(size)` will:

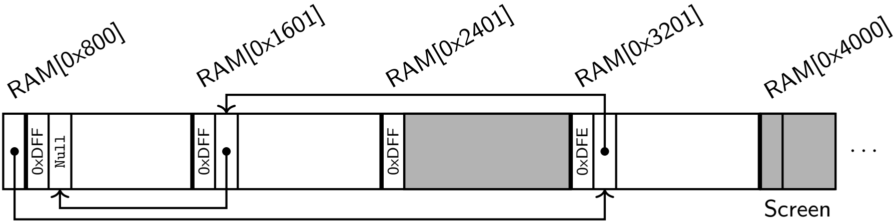
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x2402)`



`alloc(size)` will:

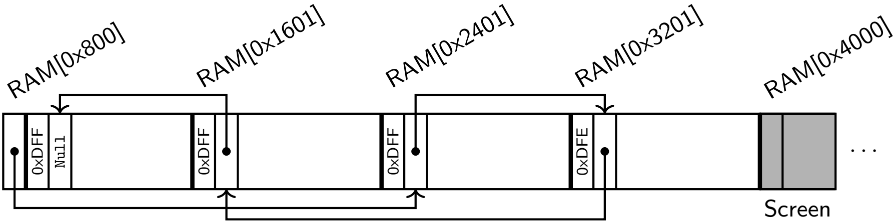
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x2402)`



`alloc(size)` will:

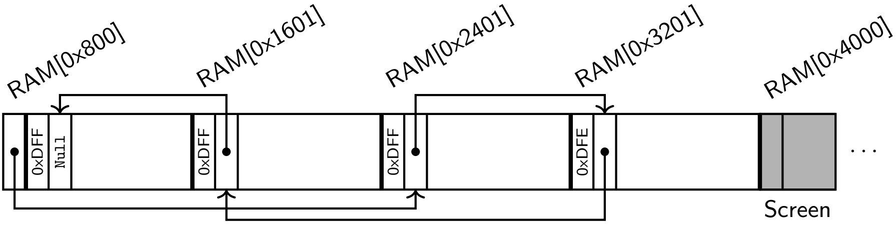
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0x6FF)`



`alloc(size)` will:

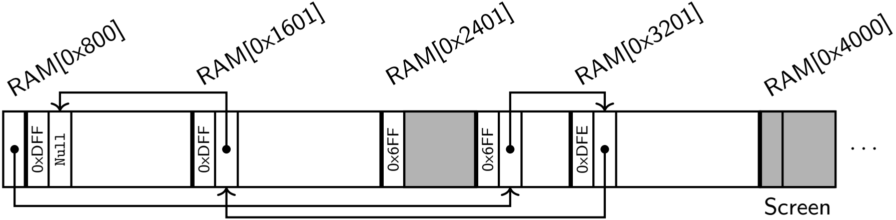
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`alloc(0x6FF)` returns 0x2402



`alloc(size)` will:

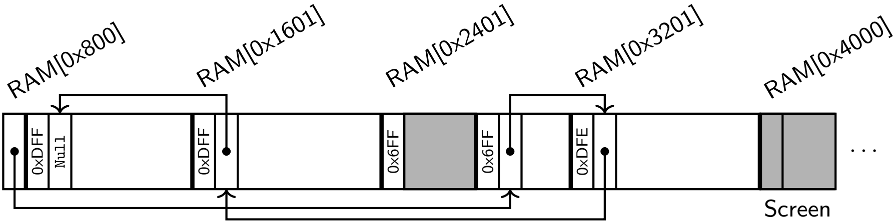
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x2402)`



`alloc(size)` will:

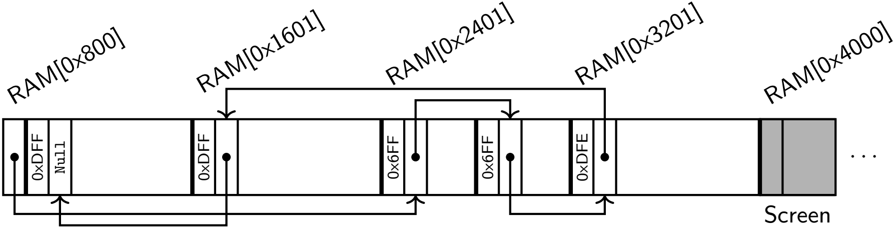
- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2

`deAlloc(0x2402)`



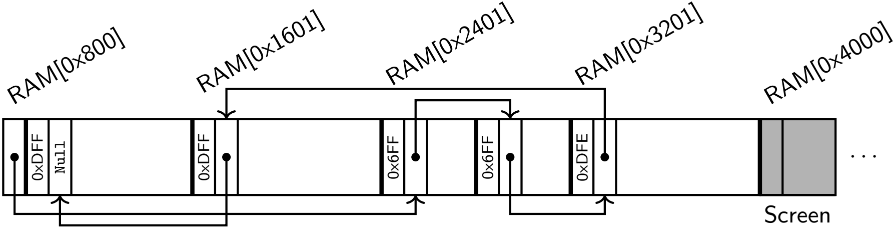
`alloc(size)` will:

- Iterate through the list of free segments looking for one large enough, say S.
- Remove S from the list of free segments and update the length of S to `size`.
- Create a new free segment at the end of S (unless S has usable size exactly `size`) and add it to the list.
- Return the base **usable** address (i.e. not the address with size information).

`deAlloc(base)` will:

- Insert the newly free segment into the start of the list from RAM[0x800], storing the pointer in RAM[`base`].

Behaviour of attempt 2



Notice that:

- Free chunks are not in any sorted order! But they don't need to be.
- We can now at least reuse memory after freeing it.
- But in this example, we can no longer allocate any segments larger than 0xDFF even after freeing our entire memory! Our `deAlloc` function has serious problems.

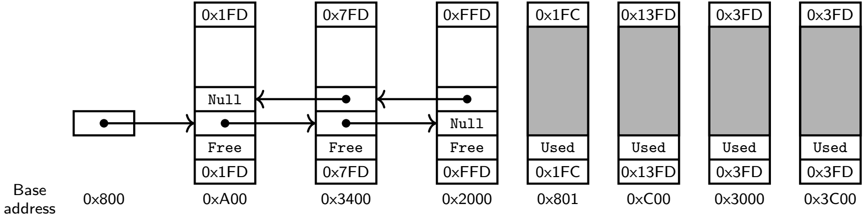
This is the "improved" allocation algorithm in 12.1.3 of Nisan and Schocken.

Attempt 3: Coalescing freed segments

We'd like to fix this by merging segments with adjacent memory segments as they're freed. (This process is called **coalescing**.)

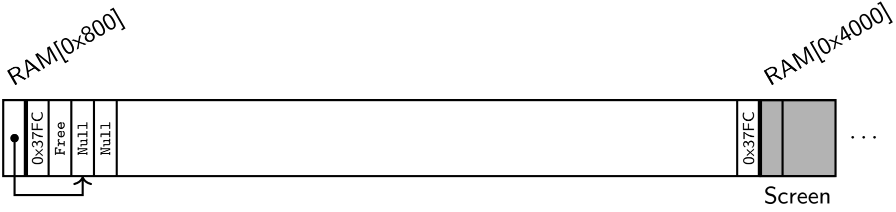
But since our free segment list isn't in sorted order, this will be very slow.

We'll need to store some more information to make it efficient.



- Every segment contains its free/used status as its second word.
- Every segment contains its usable size as the first and last words. (This lets us quickly iterate over *all* segments in memory in sorted order.)
- Free segments are arranged in a doubly-linked list, storing pointers in the third and fourth words of each free segment. RAM[0x800] contains a pointer to the first segment of the list. (This lets us quickly delete arbitrary segments from the list.)

Behaviour of attempt 3



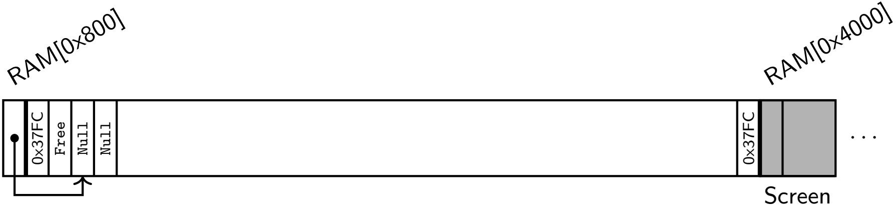
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFD)`



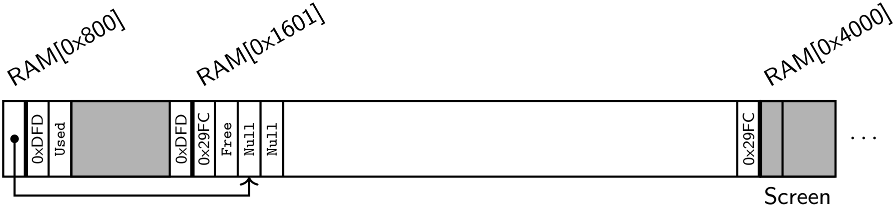
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFD)` returns 0x803



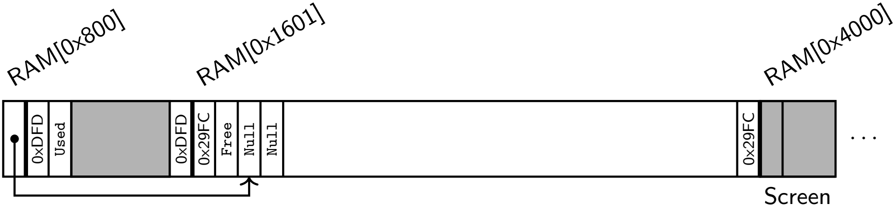
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFD)`



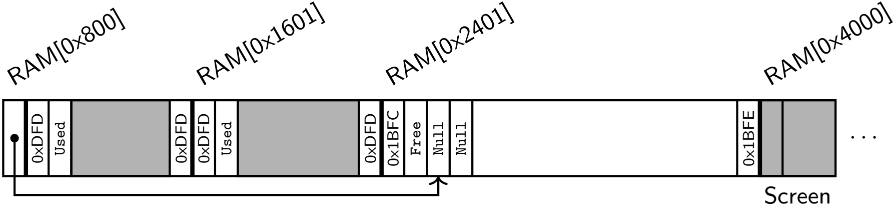
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFD)` returns 0x1603



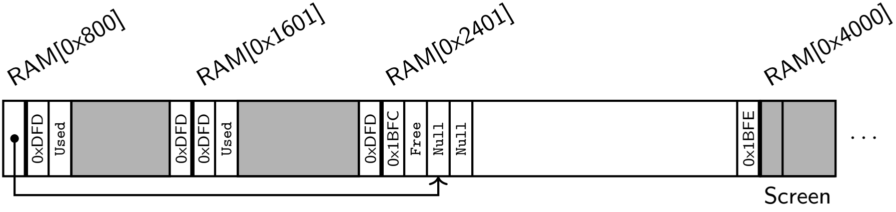
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFD)`



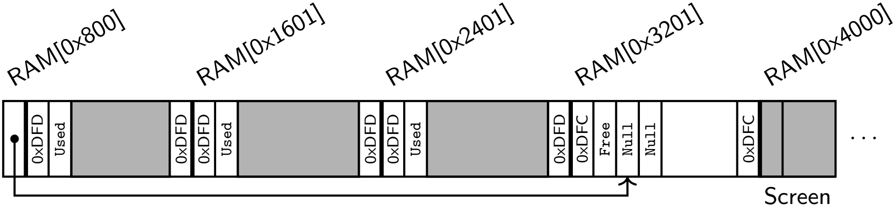
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFD)` returns 0x2403



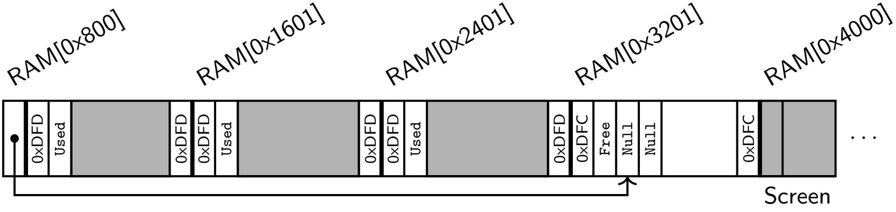
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFC)`



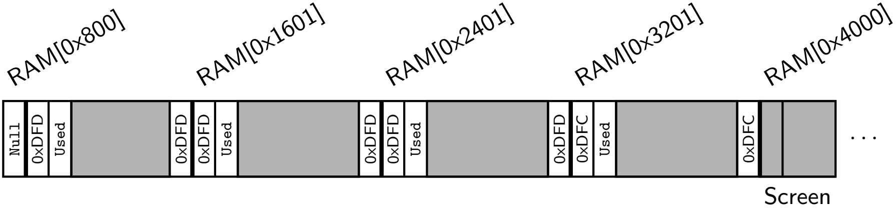
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`alloc(0xDFD)` returns 0x3203



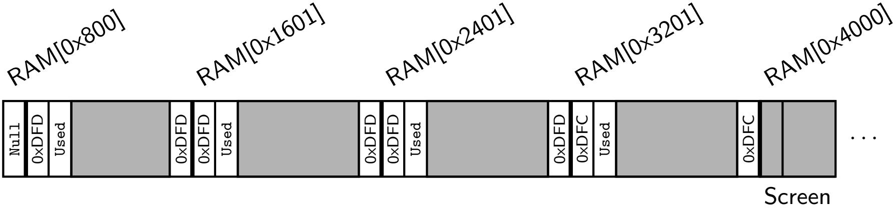
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x803)`



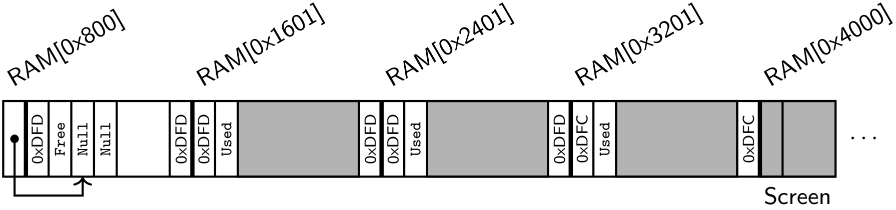
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x803)`



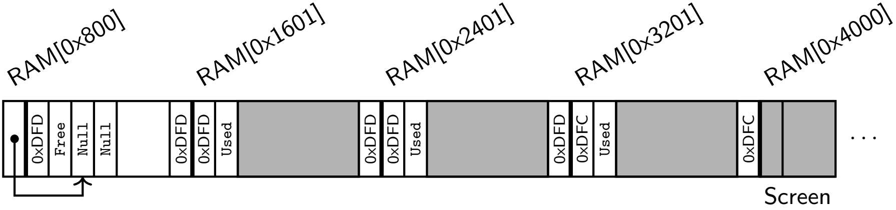
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x2403)`



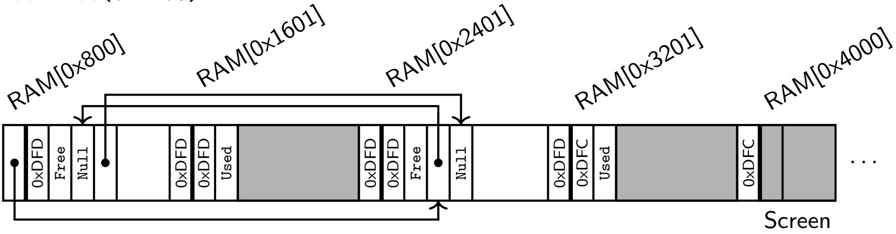
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x2403)`

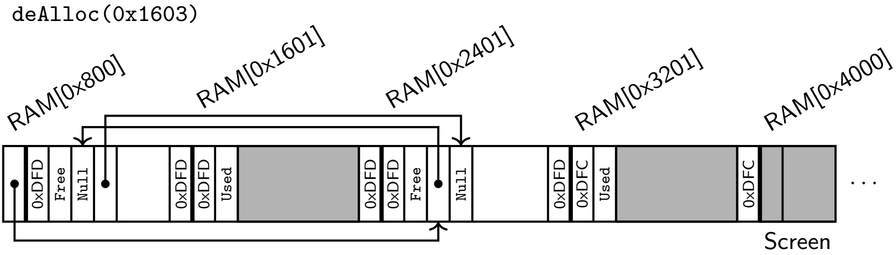


`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

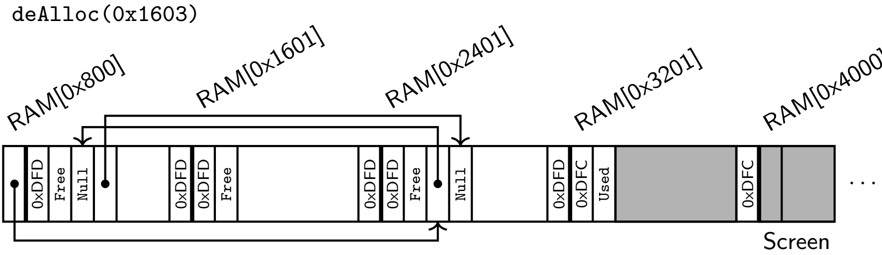


`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

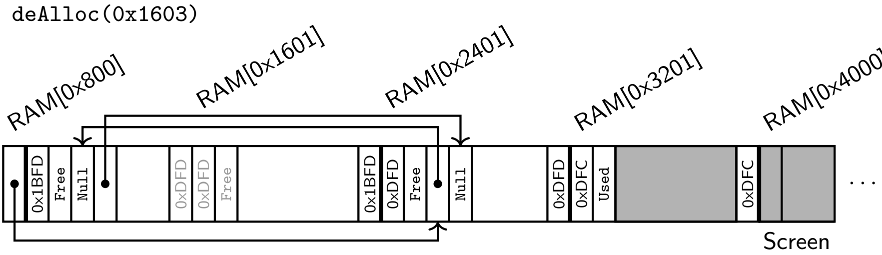


`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3



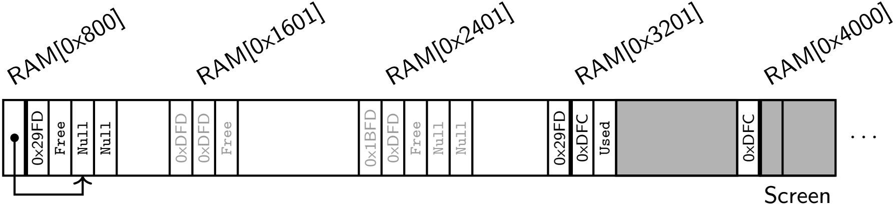
alloc(size) will behave as before, but also update the segment status to used.

deAlloc(base) will:

- Set base's segment status to Free.
- Check the segment immediately before base in memory. If it is free, merge base with it, updating the size accordingly.
- Check the segment immediately after base in memory. If it is free, merge base with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert base into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x1603)`



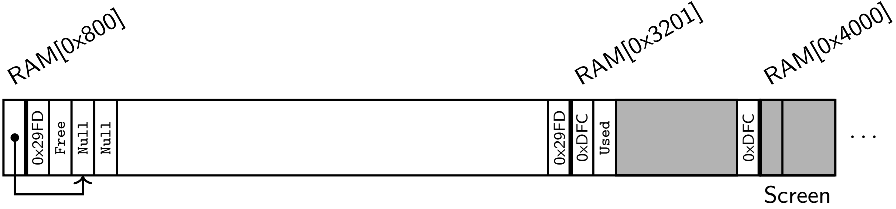
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x1603)`



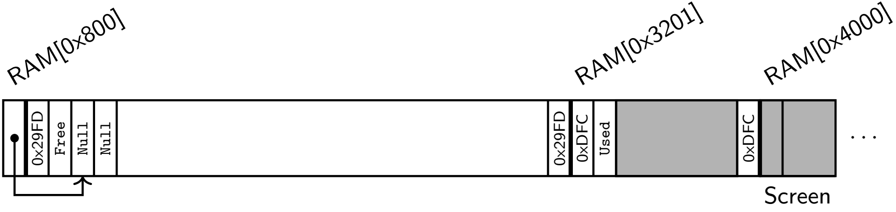
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x3203)`



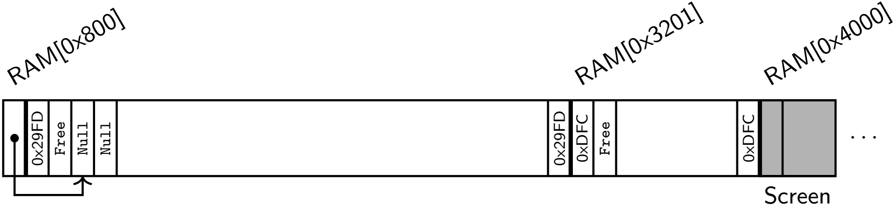
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x3203)`



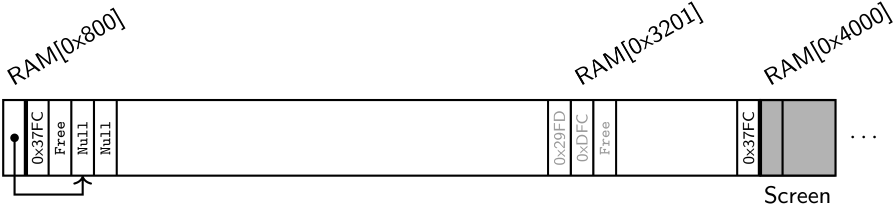
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x3203)`



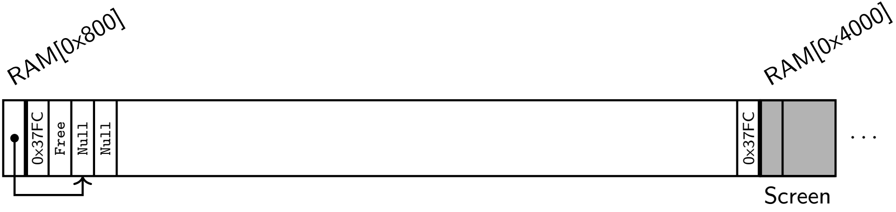
`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3

`deAlloc(0x3203)`

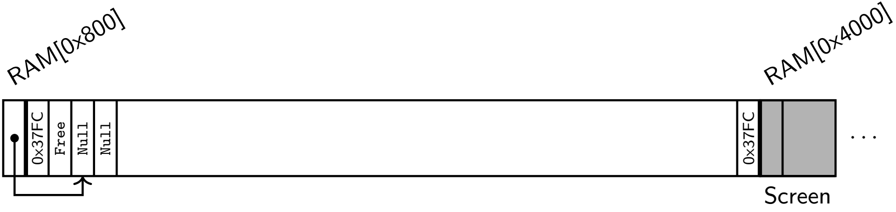


`alloc(size)` will behave as before, but also update the segment status to `used`.

`deAlloc(base)` will:

- Set `base`'s segment status to `Free`.
- Check the segment immediately before `base` in memory. If it is free, merge `base` with it, updating the size accordingly.
- Check the segment immediately after `base` in memory. If it is free, merge `base` with it, updating the size accordingly, and delete it from the list of free segments.
- If neither is free, insert `base` into the list of free segments.

Behaviour of attempt 3



Our `deAlloc` function now works properly!

And the only part of either `deAlloc` or `alloc` that takes longer than $O(1)$ time is when `alloc` looks through the free segment list for one that's big enough.

Fragmentation

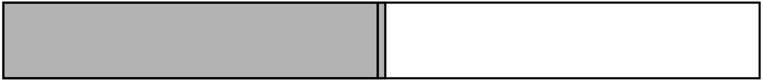
We can still have problems with the wrong sequence of `allocs` and `deAllocs`:

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:

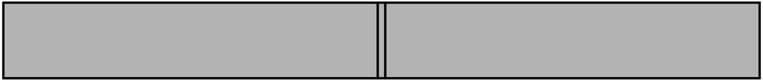
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



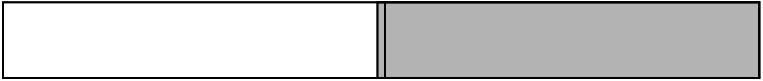
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



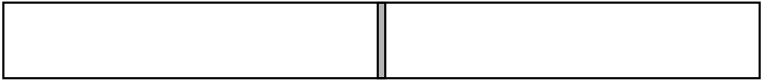
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



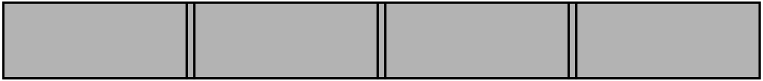
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



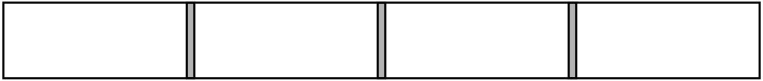
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



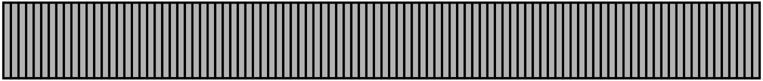
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



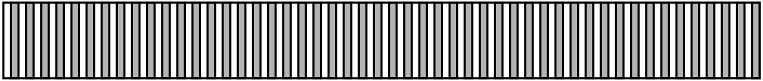
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



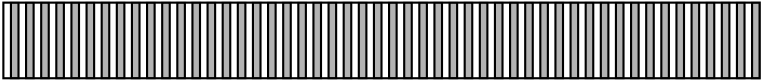
Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



Half our memory is free, but we can't `alloc` more than a few words!

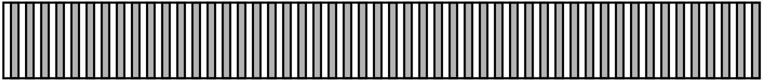
Note we didn't need to do anything too awful to get into this situation — it would be enough to e.g. `alloc` a lot of small segments and then free half of them in a random order.

This problem is called **fragmentation**, and affects both memory and file systems.

In one sense, there's an easy solution:

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



Half our memory is free, but we can't `alloc` more than a few words!

Note we didn't need to do anything too awful to get into this situation — it would be enough to e.g. `alloc` a lot of small segments and then free half of them in a random order.

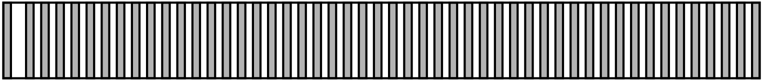
This problem is called **fragmentation**, and affects both memory and file systems.

In one sense, there's an easy solution: `deAlloc` each segment in increasing order and then `reAlloc` it, copying the data from the old location to the new one.

With algorithm #3 or #4, the resulting free space will always be in one segment. This process is called **defragmentation**, and is a bit slow but certain to work.

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



Half our memory is free, but we can't `alloc` more than a few words!

Note we didn't need to do anything too awful to get into this situation — it would be enough to e.g. `alloc` a lot of small segments and then free half of them in a random order.

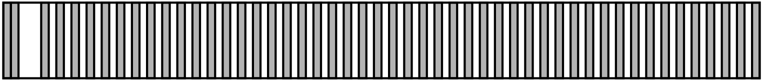
This problem is called **fragmentation**, and affects both memory and file systems.

In one sense, there's an easy solution: `deAlloc` each segment in increasing order and then `reAlloc` it, copying the data from the old location to the new one.

With algorithm #3 or #4, the resulting free space will always be in one segment. This process is called **defragmentation**, and is a bit slow but certain to work.

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



Half our memory is free, but we can't `alloc` more than a few words!

Note we didn't need to do anything too awful to get into this situation — it would be enough to e.g. `alloc` a lot of small segments and then free half of them in a random order.

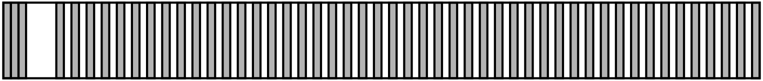
This problem is called **fragmentation**, and affects both memory and file systems.

In one sense, there's an easy solution: `deAlloc` each segment in increasing order and then `reAlloc` it, copying the data from the old location to the new one.

With algorithm #3 or #4, the resulting free space will always be in one segment. This process is called **defragmentation**, and is a bit slow but certain to work.

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



Half our memory is free, but we can't `alloc` more than a few words!

Note we didn't need to do anything too awful to get into this situation — it would be enough to e.g. `alloc` a lot of small segments and then free half of them in a random order.

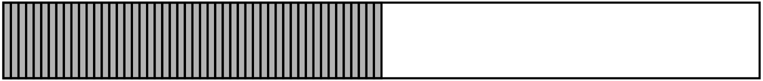
This problem is called **fragmentation**, and affects both memory and file systems.

In one sense, there's an easy solution: `deAlloc` each segment in increasing order and then `reAlloc` it, copying the data from the old location to the new one.

With algorithm #3 or #4, the resulting free space will always be in one segment. This process is called **defragmentation**, and is a bit slow but certain to work.

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



Half our memory is free, but we can't `alloc` more than a few words!

Note we didn't need to do anything too awful to get into this situation — it would be enough to e.g. `alloc` a lot of small segments and then free half of them in a random order.

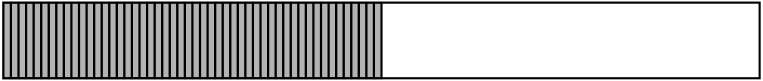
This problem is called **fragmentation**, and affects both memory and file systems.

In one sense, there's an easy solution: `deAlloc` each segment in increasing order and then `reAlloc` it, copying the data from the old location to the new one.

With algorithm #3 or #4, the resulting free space will always be in one segment. This process is called **defragmentation**, and is a bit slow but certain to work.

Fragmentation

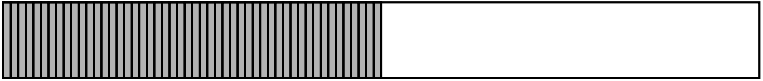
We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



There's one serious problem:

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:

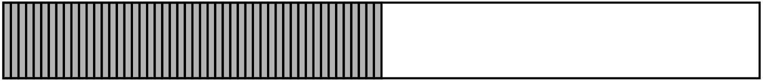


There's one serious problem: we <u>cannot</u> defragment as part of `alloc` or `deAlloc` calls. Remember, each `alloc` call just returns a pointer. We don't know what the calling code has done with that pointer, or what will happen if we change it.

As programmers, we have to deal with this ourselves — not just in Hack, but in C too! This is a major advantage of languages like Java (next TB) that deny the programmer direct access to memory via pointers, instead using **references** that behave similarly but go through a layer of indirection rather than containing actual memory addresses.

Fragmentation

We can still have problems with the wrong sequence of `allocs` and `deAllocs`:



The best we can do as part of `alloc` and `deAlloc` is try to stop memory becoming too fragmented to begin with by choosing our `alloc` return values carefully.

Our `alloc` currently uses the **first-fit** heuristic: returning the first available memory segment that's big enough. This is fast, but prone to fragmentation.

We can instead use the **best-fit** heuristic: look at the whole list of free segments and return the one whose size is closest to the requested size. This is less prone to fragmentation, but very slow.

What if we could make best-fit, or something close to best-fit, much faster?

Attempt 4: Bins

We can store not one doubly-linked list of free segments, but ten, which we call **bins**!

- In RAM[0x800], we store a pointer to the first free segment up to 0x1C words long.
- In RAM[0x801], we store a pointer to the first free segment 0x1D–0x38 words long.
- In RAM[0x802], we store a pointer to the first free segment 0x39–0x70 words long.
- . . .
- In RAM[0x809], we store a pointer to the first free segment 1C01–3800 words long.

Attempt 4: Bins

We can store not one doubly-linked list of free segments, but ten, which we call **bins**!

- In RAM[0x800], we store a pointer to the first free segment up to 0x1C words long.
- In RAM[0x801], we store a pointer to the first free segment 0x1D–0x38 words long.
- In RAM[0x802], we store a pointer to the first free segment 0x39–0x70 words long.
- ...
- In RAM[0x809], we store a pointer to the first free segment 1C01–3800 words long.

`alloc` and `deAlloc` work almost exactly as before, with two differences:

- On calling `alloc(size)` and looking for a free segment, we start scanning from the bin that will contain free segments of length `size`. If we don't find one, we scan through the bin containing $2 \times$ `size`, and so on.[1]
- When merging/splitting segments, we need to check which bin they go in.

[1] Alternatively, we can start scanning from the bin that will contain free segments of length $2 \times$ `size`, using the bin containing `size` as a last resort. This will lead to more fragmentation but will be much faster on average.

Attempt 4: Bins

We can store not one doubly-linked list of free segments, but ten, which we call **bins**!

- In RAM[0x800], we store a pointer to the first free segment up to 0x1C words long.
- In RAM[0x801], we store a pointer to the first free segment 0x1D–0x38 words long.
- In RAM[0x802], we store a pointer to the first free segment 0x39–0x70 words long.
- ...
- In RAM[0x809], we store a pointer to the first free segment 1C01–3800 words long.

`alloc` and `deAlloc` work almost exactly as before, with two differences:

- On calling `alloc(size)` and looking for a free segment, we start scanning from the bin that will contain free segments of length `size`. If we don't find one, we scan through the bin containing $2 \times$ `size`, and so on.[1]
- When merging/splitting segments, we need to check which bin they go in.

If we want to get *really* fancy, we can replace the doubly-linked lists with e.g. balanced binary search trees (e.g. 2-3-4 trees) and quickly pick the best-fit memory segment! (Modern `malloc` implementations do sort their bins, so it's worth the overhead...)

[1] Alternatively, we can start scanning from the bin that will contain free segments of length $2 \times$ `size`, using the bin containing `size` as a last resort. This will lead to more fragmentation but will be much faster on average.