System Level Programming

A5

Alexander Ertl, Hannes Weissteiner

November 22, 2019

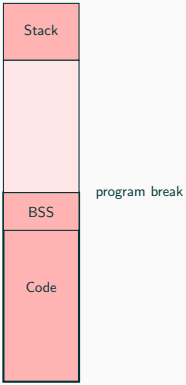
IAIK – Graz University of Technology

Dynamic Memory

```
int inputsize = 200;
int *buffer = malloc(inputsize * sizeof(int));
memcpy(buffer, input, inputsize)
// do something very important
free(buffer);
```

- Where in the memory is this buffer located?
- How can it be increased/decreased at runtime?

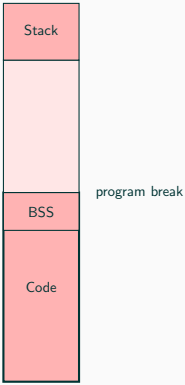
Memory of a process



Virtual memory space

- Linear address space for each process

Memory of a process

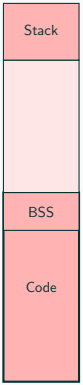


Virtual memory space

- Linear address space for each process

Code

- Segment for the binary code



Virtual memory space

- Linear address space for each process

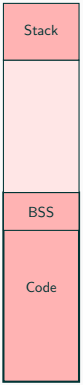
Code

- Segment for the binary code

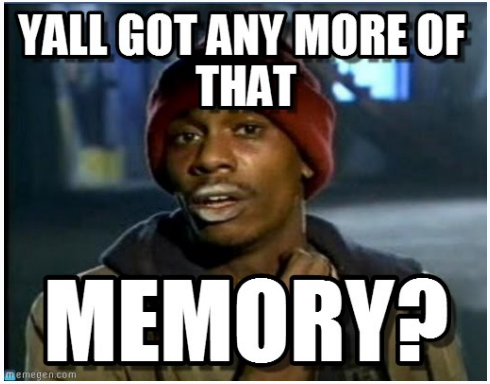
BSS

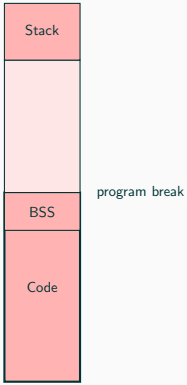
- Global/static variables with known size at compile time

Memory of a process



program break



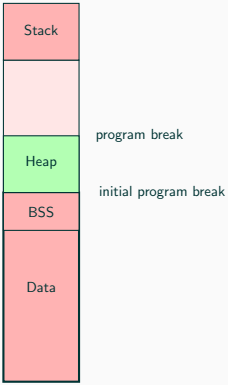


Program break

- end of data segment

Program break can be increased and decerased

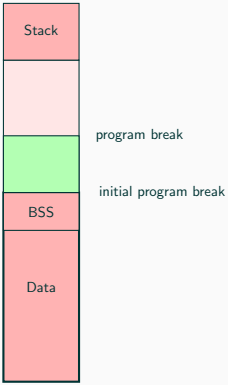
Heap



After increasing the program break (brk)

- Usable memory between end of BSS and brk

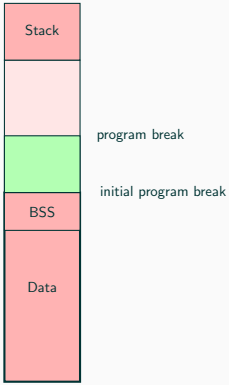
Heap



After increasing the program break (brk)

- Usable memory between end of BSS and brk
- Called the Heap

Heap

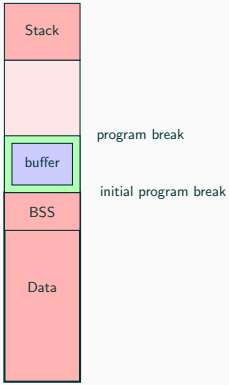


After increasing the program break (brk)

- Usable memory between end of BSS and brk
- Called the Heap

Program can use addresses below break

Heap

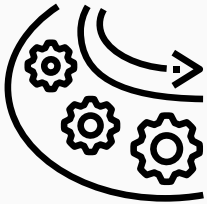


After increasing the program break (brk)

- Usable memory between end of BSS and brk
- Called the Heap

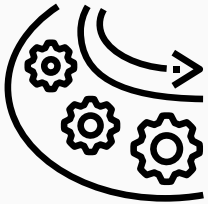
Program can use addresses below break

- Why don't we use this for our buffer?



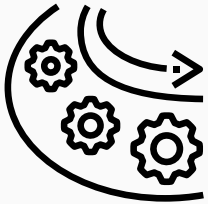
OS offers syscalls `brk` and `sbrk`

- To change the program break of the own process



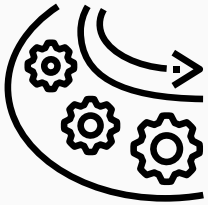
OS offers syscalls `brk` and `sbrk`

- To change the program break of the own process
- `int brk(void *addr);`
  - `brk(ptr)` sets the break to the value specified by `ptr`



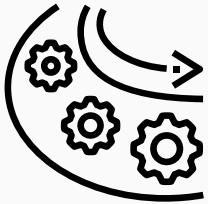
OS offers syscalls `brk` and `sbrk`

- To change the program break of the own process
- `int brk(void *addr);`
  - `brk(ptr)` sets the break to the value specified by `ptr`
- `void* sbrk(intptr_t increment);`
  - `sbrk(inc)` increments the break by inc bytes



OS offers syscalls `brk` and `sbrk`

- To change the program break of the own process
- `int brk(void *addr);`
  - `brk(ptr)` sets the break to the value specified by `ptr`
- `void* sbrk(intptr_t increment);`
  - `sbrk(inc)` increments the break by inc bytes
  - Returns the address of the previous program break
    - That is, a pointer to the newly allocated memory

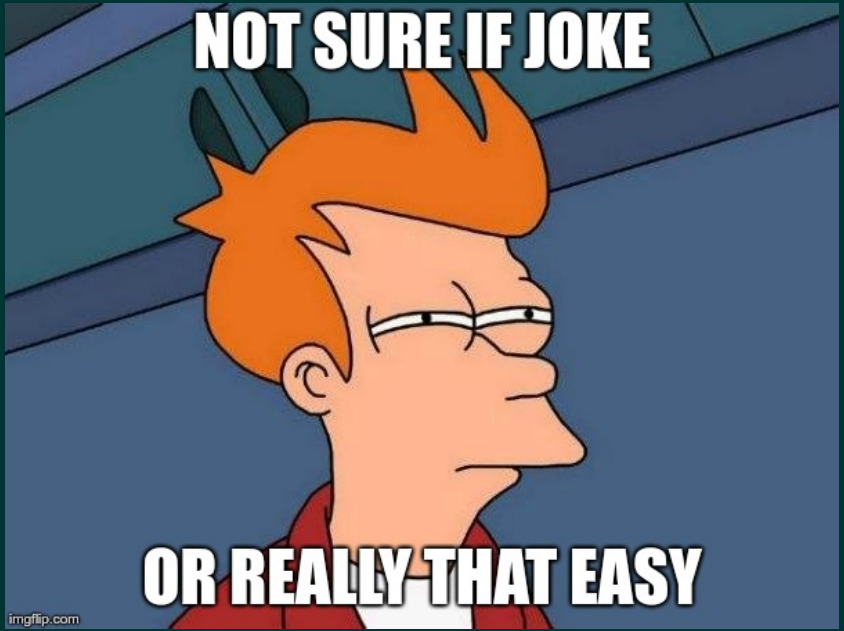


OS offers syscalls `brk` and `sbrk`

- To change the program break of the own process
- `int brk(void *addr);`
  - `brk(ptr)` sets the break to the value specified by `ptr`
- `void* sbrk(intptr_t increment);`
  - `sbrk(inc)` increments the break by inc bytes
  - Returns the address of the previous program break
    - That is, a pointer to the newly allocated memory
  - `sbrk(0)` returns current location of the break

A straight forward solution?

```
void *malloc(size_t size){
  return sbrk(size)
}
```


NOT SURE IF JOKE
OR REALLY THAT EASY
imgflip.com

A straight forward solution?

Because ...

```
void* t = malloc(100);
void* u = malloc(100);

free(t);
```

A straight forward solution?

Because ...

```
void* t = malloc(100);
void* u = malloc(100);

free(t);
```

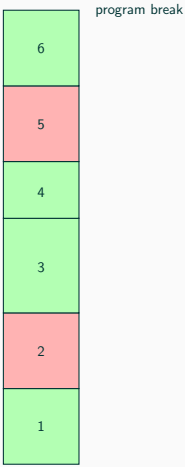
It's not that easy, but not much harder!

Goals

- Efficient usage of memory
- Reuse of freed memory areas
- Avoid fragmentation of heap segment

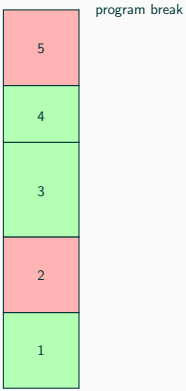
Decrease program break if possible

- If there is free memory area just below the break
- Decrease the program break



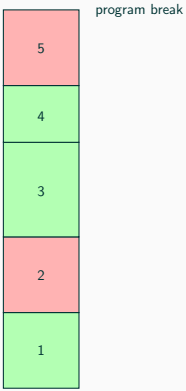
Decrease program break if possible

- If there is free memory area just below the break
- Decrease the program break



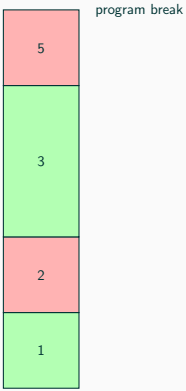
Merge free memory areas

- Only possible to merge with next or previous area



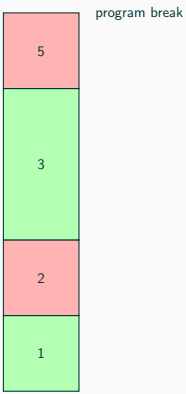
Merge free memory areas

- Only possible to merge with next or previous area



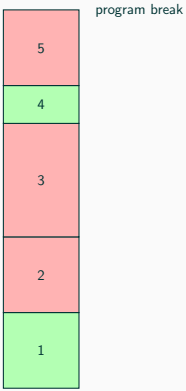
Reuse free memory areas

- When new memory is `malloc`ed
- Search for a free memory area $\geq$ requested size and split it



Reuse free memory areas

- When new memory is `malloc`ed
- Search for a free memory area $\geq$ requested size and split it



We need to know

- the size,
- the state,
- and the location

of the memory areas for an efficient memory management

We need to know

- the size,
- the state,
- and the location

of the memory areas for an efficient memory management

Any ideas how to organize this information?

What errors we want to detect?

- Double free → straight forward

What errors we want to detect?

- Double free → straight forward
- Out of memory→ straight forward
  - Have a look at the sbrk manpage

What errors we want to detect?

- Double free → straight forward
- Out of memory→ straight forward
  - Have a look at the sbrk manpage
- Buffer overflow / memory corruption

What errors we want to detect?

- Double free → straight forward
- Out of memory→ straight forward
  - Have a look at the sbrk manpage
- Buffer overflow / memory corruption
  - Special value at begin of every memory area

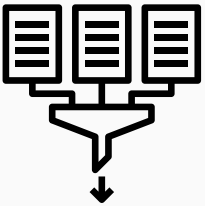
What errors we want to detect?

- Double free → straight forward
- Out of memory→ straight forward
  - Have a look at the sbrk manpage
- Buffer overflow / memory corruption
  - Special value at begin of every memory area
  - Check if first word == special value

What errors we want to detect?

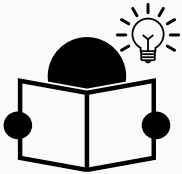
- Double free → straight forward
- Out of memory→ straight forward
  - Have a look at the sbrk manpage
- Buffer overflow / memory corruption
  - Special value at begin of every memory area
  - Check if first word == special value


GOOD MEMORY
YOU HAVE
memegenerator.net



- Choose a structure to organise the memory areas
- Decrease program break if possible
- Avoid heap fragmentation
  - Split large free memory areas to the needed size
  - Bonus: Merge free neighboring memory areas
- Detect overflows, double frees and out of memory
- Your implementation has to conform to POSIX (manpage)

cf. assignment page for further details



Make use of pointer arithmetic

- `int* p; p+5;` $\longrightarrow$ address in `p` is increased by `5*sizeof(int)`

How many bytes does a pointer need?

- Correct use of `sizeof`

Double-Linked-List of memory areas

Be careful to test the right malloc implementation ;)

Debugging with `printf`?