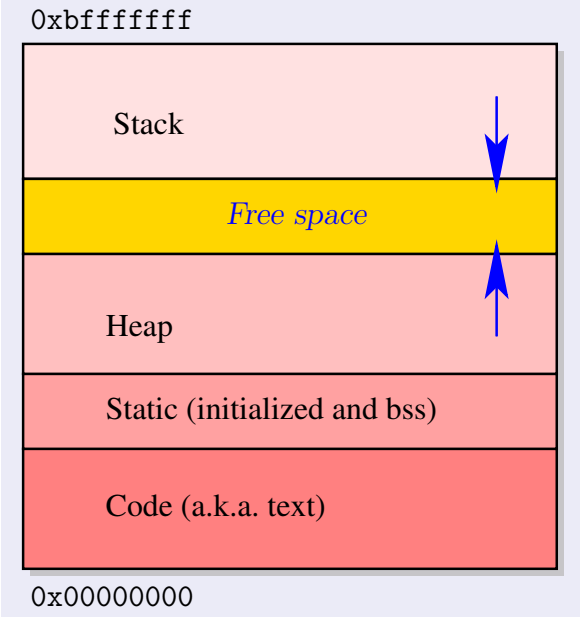
Memory Address Space of a Process



Memory for a process is allocated and initialized when loading and executing a program. Memory access in user mode is restricted to this address space. This address space consists of the following four segments:

1. Code (also called text) segment: .o and executable code
2. Static Data segments: Initialized global (and C static) variables and uninitialized global variables that are zeroed when initializing the process, also called bss
3. Stack segment: Stack frames of function call arguments and local variables, also called automatic variables in C
4. Heap segment: Dynamic allocation (malloc())

System Call: brk( )

#include <unistd.h>

int brk(void *end_data_segment);
void *sbrk(intptr_t displacement);

brk() sets the end of the data segment, which is also the end of heap to the value specified by *end_data_segment*, when that value is reasonable, the system does have enough memory and the process does not exceed its max data size (see man pages of setrlimit( ) and getrlimit( ))

sbrk() adds a displacement (possibly 0) and returns the starting address of the new area (it is a C function, front-end to sbrk())

Both are deprecated as “programmer interface” functions, i.e., they are meant for kernel development only.

Memory Address Space Example

Compile and run the following C program:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

double t[0x02000000];

void segments()
{
   static int s = 42;
   void *p = malloc(1024);
   printf("stack\t%010p\nbrk\t%010p\nheap\t%010p\n"
         "static\t%010p\nstatic\t%010p\ntext\t%010p\n",
         &p, sbrk(0), p, t, &s, segments);
}

int main(int argc, char *argv[])
{
   segments();
   exit(0);
}
```

Your output will look something like this. Try to understand this program and interpret the output with respect to the memory address space of the program.

```
stack   0x7ffd3af708d0
brk     0x10e33000
heap    0x10e12010
static  0x601080
static  0x601058
text    0x400666
```