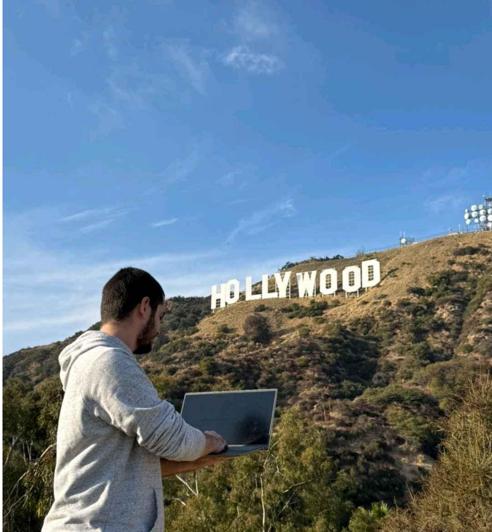
Διαχείριση Μνήμης

Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών
Εισαγωγή στον Προγραμματισμό (Κ04)
Δημήτρης Σκόνδρας Μέξης

whoami



- Αυτή την στιγμή βρίσκομαι στο 3ο έτος των σπουδών μου στο τμήμα.
- Είμαι βοηθός στο μάθημα για 2η χρονιά.
- Με ενδιαφέρουν οι τομείς που διασυνδέουν το υλικό με το λογισμικό (Αρχιτεκτονική Υπολογιστών & Λειτουργικά Συστήματα).
- Linux geek.

Τι θα δούμε σήμερα:

- Τι είναι το **Segmentation Fault** και πώς προκύπτει.

- Στατική Δέσμευση Μνήμης vs Δυναμική Δέσμευση Μνήμης.

- Θα καταρρίψουμε δύο επιχειρήματα στο ερώτημα: “Γιατί να μην κάνω free;”.

- Το εργαλείο **Valgrind** και δύο παραδείγματα χρήσης του.

Παράδειγμα Στατικής Δέσμευσης Μνήμης:

Ιδέα: θα φτιάξουμε ένα πρόγραμμα που δεσμεύει μνήμη για έναν πίνακα ακεραίων ο οποίος περιέχει N στοιχεία (έστω N = 400.000.000).

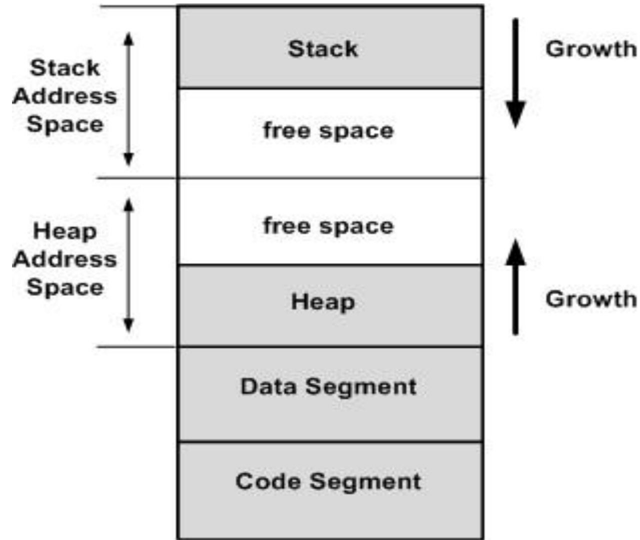
```
/* File: staticalloc.c */
#include <stdio.h>
#include <stdlib.h> // rand()

#define N 400000000

int main(void)
{
    int arr[N], i;
    for (i = 0; i < N; ++i)
        arr[i] = rand();

    for (i = 0; i < N; ++i)
        printf("%d\n", arr[i]);

    return 0;
}
```

Αποτέλεσμα:

```
$ gcc -o staticalloc staticalloc.c
$ ./staticalloc
zsh: segmentation fault (core dumped)  ./staticalloc
```

Πού είναι το πρόβλημα;

Πώς φανταζόμαστε την μνήμη ενός προγράμματος:



Παράδειγμα που περιέχει όλες τις παραπάνω μνήμες.

```
/* File: segments.c */
#include <stdio.h>
#include <stdlib.h>

int global_var = 10;                    // data segment (καθολική μεταβλητή)
int main(void)                          // code segment
{
    static int static_var = 20;         // data segment (στατική μεταβλητή)
    int auto_var = 30;                  // stack (τοπική μεταβλητή)
    malloc(sizeof(int));                // heap
}
```

Παράδειγμα συμπεριφοράς Stack & Heap.

```c
/* File: stack_heap.c */
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   char a, b;
   printf("a: %p \t b:%p\n", &a, &b);
   printf("distance between a and b: %d\n", &a - &b);

   printf("\n");

   char *pa = malloc(sizeof(char)), *pb = malloc(sizeof(char));
   printf("pa: %p \t pb:%p\n", pa, pb);
   printf("distance between pa and pb: %d\n", pa - pb);
   free(pa);
   free(pb);

   return 0;
}
```

```
$ gcc -o stack_heap stack_heap.c
$ ./stack_heap
a: 0x7ffe3cf0e2af       b:0x7ffe3cf0e2ae
distance between a and b: 1

pa: 0xa5006b0    pb:0xa5006d0
distance between pa and pb: -32
```

Το ίδιο παράδειγμα με Δυναμική Δέσμευση Μνήμης:

```
/* File: dynamicalloc.c */
#include <stdio.h>
#include <stdlib.h>

#define N 400000000

int main(void)
{
   int *arr, i;
   arr = malloc(N * sizeof(int));
   if (arr == NULL) {
      printf("Malloc Failed!\n");
      return 1;
   }

   for (i = 0; i < N; ++i)
      arr[i] = rand();

   for (i = 0; i < N; ++i)
      printf("%d\n", arr[i]);

   free(arr);
   return 0;
}
```

```
$ gcc -o dynamicalloc dynamicalloc.c
$ ./dynamicalloc
1804289383
846930886
1681692777
…
628175011
1656478042
1131176229
```

Συμπέρασμα:
Για μεγάλα N προτιμάμε να δεσμεύουμε τα δεδομένα μας Δυναμικά

Θετικά και αρνητικά με την χρήση της malloc.

+ Μπορέσαμε να κάνουμε την δουλειά μας για μεγάλο αριθμό δεδομένων.

- Προσθέσαμε παραπάνω γραμμές κώδικα (έλεγχος αποτυχίας malloc & χρήση της free)

Ποιές είναι όμως οι κύριες διαφορές της Στοίβας και του Σωρού;

Στοίβα vs Σωρός

Στην Στοίβα:

- Ό,τι δηλώσουμε, δεσμεύεται και αποδεσμεύεται από μόνο του.
- Ωστόσο το μέγεθος της μας περιορίζει.

Στον Σωρό:

- Είμαστε εμείς υπεύθυνοι για την σωστή δέσμευση και αποδέσμευση της μνήμης.
- Κατάλληλο για μεγάλες εφαρμογές λόγω μεγέθους και ευελιξίας.

Γιατί να κάνουμε free;

Ο έλεγχος για την αποτυχία της malloc φαίνεται απόλυτα λογικός και χρήσιμος, άρα οι πρόσθετες γραμμές κώδικα έχουν νόημα. Στην free γιατί όμως;

Μία χρήσιμη πληροφορία:

Όταν μία διεργασία τερματίζει, η μνήμη που έχει δεσμεύσει, αποδεσμεύεται. Αυτό οφείλεται επειδή στις σύγχρονες πλατφόρμες τα λειτουργικά συστήματα έχουν ενσωματωμένους μηχανισμούς για την αποδέσμευση των πόρων μετά το τέλος της διεργασίας.

Δύο λανθασμένα επιχειρήματα για την αποφυγή της free στους κώδικες μας.

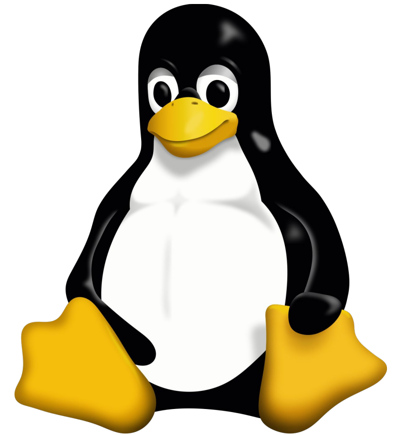
1) Εφόσον όταν τερματίσει το πρόγραμμα μου, όποια malloc κι αν κάνω, στο τέλος το λειτουργικό σύστημα θα κάνει την βαρετή δουλειά για μένα. Άρα κανένα νόημα να χρησιμοποιήσω την free.

2) Έχω ένα μηχάνημα με 32GB RAM, ακόμα κι αν χρησιμοποιήσω ένα παλιό λειτουργικό σύστημα, η μνήμη μου θα χωρέσει όλες τις malloc που δεν αποδεσμεύτηκαν με free.

Απάντηση στο πρώτο επιχείρημα:

Πράγματι, ιδιαίτερα στο παράδειγμα που είδαμε στην αρχή, θα εκτυπωθούν οι τυχαίοι αριθμοί, θα τερματίσει η διεργασία και όλη η μνήμη θα αποδεσμευτεί.

Ωστόσο, ας σκεφτούμε λογισμικά και τις λειτουργίες τους.



Απάντηση στο δεύτερο επιχείρημα:

Πράγματι, ένα μηχάνημα με τέτοια μνήμη θα μπορούσε να ανταποκριθεί σε εφαρμογές που έχουν διαρροές μνήμης. Ωστόσο, αν σκεφτούμε απαιτητικές εφαρμογές, για παράδειγμα, εικονικές μηχανές να τρέχουν συνέχεια και προγράμματα επεξεργασίας video, η σωστή διαχείριση μνήμης σε αυτά τα προγράμματα έχει τεράστια σημασία!

Ακολουθεί ένα παράδειγμα ενός παιχνιδιού στο οποίο η διαχείριση μνήμης είναι για σεμινάριο.

DOOM!

Μερικές πληροφορίες για το Doom:

- Κυκλοφόρησε το 1993 από την id Software.
- Είναι το πιο **γνωστό και επιδραστικό** 3D παιχνίδι της εποχής του.
- Παρόλο που σχεδιάστηκε αρχικά για υπολογιστές, αργότερα κυκλοφόρησε και στις διάσημες κονσόλες όπως Atari, Super Nintendo, Playstation, Game Boy Advance…
- Τα γραφικά και ο σχεδιασμός ενός τέτοιου 3D παιχνιδιού έμοιαζε αδύνατο στους υπολογιστές εκείνης της εποχής, οι οποίοι είχαν μνήμη κάτω των **4MB RAM**!

Ένα τρομερό βίντεο που δείχνει την σπουδαιότητα του συγκεκριμένου παιχνιδιού αυτήν την περίοδο: https://www.youtube.com/watch?v=KFDlVgBMomQ&t=389s

Τι είναι το valgrind;

Το valgrind είναι ένα ισχυρό εργαλείο προφίλ και ανάλυσης δυναμικής συμπεριφοράς προγραμμάτων. Θα το χρησιμοποιήσουμε για να ανιχνεύσουμε τις διαρροές μνήμης σε δύο προγράμματα.

Άλλες χρήσεις του valgrind:

- Παράνομες προσβάσεις μνήμης.
- Αποσφαλμάτωση νημάτων.
- Κακή απόδοση cache.

Σας ευχαριστώ πολύ για τον χρόνο σας, καλή συνέχεια!

Keep coding ;)