

Οργάνωση και Σχεδίαση Υπολογιστών (ECE 219)

Χειμερινό Εξάμηνο 2020-2021

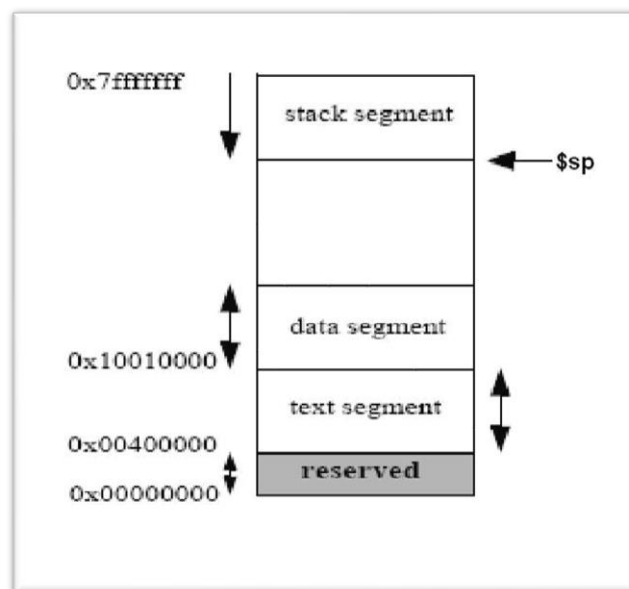
Εργαστήριο 2

Στόχοι του εργαστηρίου

- Χρήση στοίβας
- Συναρτήσεις

Στοίβα

Κατά την εκτέλεση ενός προγράμματος ένα μέρος της μνήμης δεσμεύεται και παραχωρείται για το συγκεκριμένο process. Το κομμάτι αυτό της μνήμης χωρίζεται σε ορισμένους τομείς όπου σε κάθε ένα αποθηκεύονται συγκεκριμένου τύπου δεδομένα. Όπως φαίνεται και στην εικόνα 1 οι τομείς όπου χωρίζεται η μνήμη είναι οι εξής:



Εικόνα 1 (memory segments)

Reserved: Δεσμευμένη από το λειτουργικό περιοχή της μνήμης όπου το πρόγραμμα σας δεν μπορεί να έχει πρόσβαση.

Text Segment: Περιοχή της μνήμης όπου αποθηκεύονται οι εντολές του προγράμματος. Πρόκειται για εντολές σε γλώσσα μηχανής. Όταν ο κώδικας που έχετε γράψει δεν αποτελείται από τις λεγόμενες ψευδοεντολές αλλά από πραγματικές εντολές τότε για κάθε

εντολή που έχετε συντάξει έχει αποθηκευτεί σε αυτό το κομμάτι της μνήμης η αντίστοιχη εντολή συστήματος σε binary μορφή.

Data Segment: Περιοχή της μνήμης όπου αποθηκεύονται τα δεδομένα του προγράμματος. Όλες οι μεταβλητές, οι πίνακες, και γενικά ό,τι έχετε δηλώσει στην περιοχή .data του κώδικα σας αποθηκεύεται σε αυτό το κομμάτι της μνήμης. Αν αναλύσουμε περισσότερο θα δούμε ότι χωρίζεται σε 2 κομμάτια. Το static data segment όπου αποθηκεύονται τα στατικά δεδομένα (αυτά που δηλώθηκαν στο .data) και το dynamic data segment όπου είναι ο επιπλέον χώρος που παραχωρεί δυναμικά το λειτουργικό.

Stack Segment: Περιοχή της μνήμης που χρησιμοποιείται από την στοίβα. Τα όρια αυτού του τομέα καθορίζονται από τον δείκτη \$sp τον οποίο μπορείτε να αλλάζετε κατά την εκτέλεση του προγράμματος. Αντίθετα τα όρια του data segment καθορίζονται από το λειτουργικό και για να δεσμεύσετε επιπλέον μνήμη πρέπει να καλέσετε την αντίστοιχη λειτουργία του συστήματος (εντολή malloc στην C).

Κενό κομμάτι: Ένα κομμάτι μεταξύ του data segment και του stack segment το οποίο αν και δεν περιέχει δεδομένα δεσμεύεται και παραχωρείται από το λειτουργικό στο συγκεκριμένο process. Αν κατά την διάρκεια της εκτέλεσης χρειάζεται να αυξηθεί η περιοχή της στοίβας ή η περιοχή των δεδομένων τότε θα πάρουν επιπλέον χώρο από αυτό το κομμάτι.

Χρήση της στοίβας

Καταρχάς όταν αναφερόμαστε σε μια στοίβα εννοούμε μια LIFO (Last In First Out) λίστα. Αυτό σημαίνει ότι μπορούμε κάθε φορά να ανακτούμε τα στοιχεία με την ανάποδη σειρά από αυτή που τα αποθηκεύσαμε (πρώτα ανακτούμε τα τελευταία στοιχεία που μπήκαν στην λίστα). Από την εικόνα 1 βλέπετε ότι για να αυξήσουμε τον χώρο που καταλαμβάνει η στοίβα πρέπει να μετακινήσουμε τον δείκτη \$sp προς τα κάτω. Για να γίνει αυτό πρέπει να **μειώσουμε** την τιμή του τόσες θέσεις όσα και τα επιπλέον bytes που θέλουμε να αυξηθεί η στοίβα. Επειδή κατά κανόνα στην στοίβα σώζουμε τα περιεχόμενα καταχωρητών, οι οποίοι έχουν μέγεθος 4 bytes, ο επιπλέον χώρος στην στοίβα πρέπει να εξασφαλίσουμε ότι θα είναι πολλαπλάσιο του 4 χωρίς όμως αυτό να είναι δεσμευτικό.

Γιατί να χρησιμοποιήσω την στοίβα;

Κατά την εκτέλεση ενός προγράμματος οι καταχωρητές είναι όλοι τους ορατοί από οποιαδήποτε σημείο του κώδικα, ακόμα και από τις διάφορες συναρτήσεις. Αυτό όμως μπορεί να δημιουργήσει πρόβλημα κατά την εκτέλεση ενός προγράμματος. Όταν ένας συγκεκριμένος καταχωρητής χρησιμοποιείται για εγγραφή μέσα σε μια συνάρτηση (ας την πούμε A) και παράλληλα περιέχει δεδομένα απαραίτητα για μια άλλη συνάρτηση (ας την πούμε B) η οποία σε κάποιο σημείο της εκτέλεσης της κάλεσε την A τότε τα δεδομένα αυτού του καταχωρητή χάνονται για την συνάρτηση B.

Για να καταλάβετε καλύτερα το πρόβλημα που δημιουργείται ας δούμε το παρακάτω παράδειγμα:

Func1:

```

...
li $s0,50      # εντολή No1: Στον καταχωρητή $s0 αποθηκεύονται δεδομένα
               # απαραίτητα για την Func1. Έστω η τιμή 50.
...
jal    Func2      # κλήση της Func2. Η εκτέλεση μεταπηδά στην Func2:
...
move $a0, $s0  # Πρόβλημα. Τα δεδομένα στον $s0 δεν είναι αυτά της
               # εντολής No1. Η συνάρτηση περίμενε να βρει 50.
...
Func2:         #συνάρτηση func2
...
addi $s0, $zero,100  #Στον καταχωρητή $s0 αποθηκεύεται η τιμή 100.
                    #Ότι δεδομένα υπήρχαν πριν μέσα χάθηκαν.
...
jr $ra

```

Στο παραπάνω παράδειγμα βλέπουμε το πρόβλημα που δημιουργείται με τον καταχωρητή \$s0. Τόσο η func1 όσο και η func2 χρησιμοποιούν αυτόν τον καταχωρητή. Το πρόβλημα δημιουργείται στην func1 η οποία μετά την κλήση της func2 έχασε τα περιεχόμενα που είχε στον \$s0.

Θα μπορούσε να ισχυριστεί κανείς ότι σε τέτοιες περιπτώσεις προσέχοντας ποιους καταχωρητές θα χρησιμοποιήσουμε μπορούμε να αποφύγουμε τέτοιες δυσάρεστες παρενέργειες. Τι γίνεται όμως όταν καλείστε να υλοποιήσετε μια συνάρτηση την οποία αργότερα θα χρησιμοποιήσει κάποιος άλλος προγραμματιστής; Ή επίσης τι γίνεται αν χρειασθεί να υλοποιήσετε μεγάλους σε έκταση κώδικες όπου οι κλήσεις συναρτήσεων είναι πολύ συχνές;

Πως λύνεται το πρόβλημα με χρήση στοίβας.

Η διαδικασία που πρέπει να ακολουθήσετε για να αποθηκεύσετε κάτι στην στοίβα είναι η εξής:

1. Μειώνεται των δείκτη \$sp κατά τόσες θέσεις ανάλογα με το πλήθος των καταχωρητών που θέλετε να αποθηκεύσετε. Σε κάθε καταχωρητή αντιστοιχούν 4 bytes. Έστω ότι δεσμεύεται χώρο για N words. Μειώνετε κατά N*4 bytes.
2. Κάνετε store με την εντολή sw τα περιεχόμενα των καταχωρητών που θα χρησιμοποιήσετε στις θέσεις μνήμης από εκεί που δείχνει ο \$sp μέχρι συν N*4 θέσεις.
3. Χρησιμοποιείτε κανονικά μέσα στην συνάρτηση τους καταχωρητές.
4. Κάνετε load με την εντολή lw τα αρχικά περιεχόμενα των καταχωρητών αντίστοιχα όπως τα κάνατε store στο βήμα 2
5. Αυξάνετε τον \$sp κατά N*4 θέσεις ώστε να επανέλθει στη θέση που ήταν πριν το βήμα 1.
6. Η συνάρτηση επιστρέφει.

Παράδειγμα χρήσης της στοίβας

Παρακάτω ακολουθεί η υλοποίηση μιας συνάρτησης η οποία σώζει ότι είναι απαραίτητο στην στοίβα.

Παράδειγμα:

```
func1:
addi $sp,$sp,-12      #δεσμεύουμε θέσεις για τρεις λέξεις στην στοίβα.
sw $ra,0($sp)         #κάνουμε push στην στοίβα τα περιεχόμενα του $ra
sw $t1,4($sp)         #κάνουμε push στην στοίβα τα περιεχόμενα του $t1
sw $t2,8($sp)         #κάνουμε push στην στοίβα τα περιεχόμενα του $t2

...

addi $t1, $0,10      #αλλάζουμε χωρίς περιορισμούς τα περιεχόμενα των 2
move $t2, $t1        #καταχωρητών που κάναμε push

...

jal func2             #μπορούμε να καλέσουμε χωρίς πρόβλημα και άλλες
                     # συναρτήσεις

...

jal func1             #μπορούμε να καλέσουμε ακόμα και την ίδια.
                     #συνάρτηση(αναδρομικά) χωρίς κανένα πρόβλημα

...

lw $ra, 0($sp)        #κάνουμε pop από την στοίβα τα περιεχόμενα του $ra
lw $t1,4($sp)         #κάνουμε pop από την στοίβα τα περιεχόμενα του $t1
lw $t2,8($sp)         #κάνουμε pop από την στοίβα τα περιεχόμενα του $t2
addi $sp, $sp,12      #επαναφέρουμε τον stack pointer
jr $ra
```

Στον επεξεργαστή MIPS υπάρχει ένα μεγάλο πλήθος καταχωρητών οι περισσότεροι από τους οποίους έχουν και συγκεκριμένη χρήση. Οι καταχωρητές \$s0-\$s7 και \$t0-\$t9 ονομάζονται γενικής χρήσης και χρησιμοποιούνται συχνότερα κυρίως για πράξεις μεταξύ καταχωρητών. Μπορείτε να δείτε από τον πίνακα 1 ο οποίος υπάρχει και στο Instructions Set το πως πρέπει να μεταχειρίζεστε τον κάθε καταχωρητή ανάλογα με την ομάδα που ανήκει.

Register	Number	Usage
zero	0	Constant 0
at	1	Reserved for assembler
v0	2	Used for return values from function calls.
v1	3	
a0	4	Used to pass arguments to procedures and functions.
a1	5	
a2	6	
a3	7	
t0	8	Temporary (Caller-saved, need not be saved by called procedure)
t1	9	
t2	10	
t3	11	
t4	12	
t5	13	
t6	14	
t7	15	
s0	16	Saved temporary (Callee-saved, called procedure must save and restore)
s1	17	
s2	18	
s3	19	
s4	20	
s5	21	
s6	22	
s7	23	
t8	24	Temporary (Caller-saved, need not be saved by called procedure)
t9	25	
k0	26	Reserved for OS kernel
k1	27	
gp	28	Pointer to global area
sp	29	Stack pointer
fp	30	Frame pointer
ra	31	Return address for function calls.

Πίνακας 1

Οι καταχωρητές \$t0 έως \$t9 ονομάζονται Temporary ή Caller-saved. Αυτό μας λέει ότι μια συνάρτηση δεν είναι υποχρεωμένη να κρατήσει το περιεχόμενο των καταχωρητών ίδιο όταν επιστρέψει. Αν ο Caller, που μπορεί να είναι μια άλλη συνάρτηση ή η main, θέλει να έχει το περιεχόμενο αυτών των καταχωρητών και μετά την κλήση κάποιων συναρτήσεων οφείλει να τα σώσει ο ίδιος στην στοίβα.

Οι καταχωρητές \$s0 \$s7 ονομάζονται Saved Temporary ή Callee-Saved(ο καλούμενος σώζει). Αυτό σημαίνει ότι μια συνάρτηση οφείλει να κρατήσει σταθερό το περιεχόμενο αυτών των καταχωρητών με αυτό που είχαν πριν. Για να γίνει αυτό πρέπει να σώσει στην αρχή το περιεχόμενο τους στην στοίβα και να το επαναφέρει μόλις τελειώσει. Αυτό ακριβώς γίνεται και στο παράδειγμα.

Για το επόμενο εργαστήριο έχετε να υλοποιήσετε τις παρακάτω ασκήσεις. Την ώρα του εργαστηρίου θα εξετασθείτε προφορικά πάνω στους κώδικες που θα παραδώσετε.

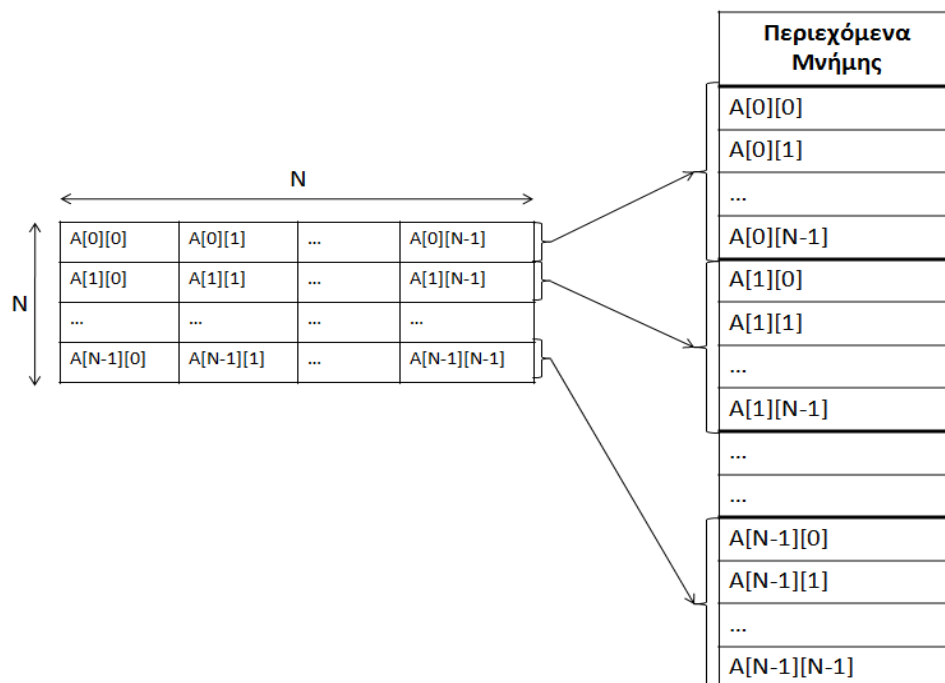
Άσκηση 1 Συμμετρικός πίνακας (5 μονάδες)

Να αναπτύξετε μία συνάρτηση `int IsSymmetric(int *A[], int N)` η οποία θα εξετάζει εάν ο τετραγωνικός πίνακας $A[N][N]$ είναι συμμετρικός. Η `main` θα ζητάει αρχικά από τον χρήστη το μέγεθος N του τετραγωνικού πίνακα $N \times N$ και στη συνέχεια τα στοιχεία που θα αποθηκευτούν στις N^2 θέσεις αυτού του πίνακα, κατά γραμμή. Η συνάρτηση θα επιστρέφει 1 εάν ο πίνακας είναι συμμετρικός, αλλιώς θα επιστρέφει 0.

Υπόδειξη:

Οι μονοδιάστατοι πίνακες αποθηκεύονται στην μνήμη έτσι ώστε το στοιχείο $A[i]$ να βρίσκεται αμέσως μετά το στοιχείο $A[i-1]$ του πίνακα. Η αποθήκευση ενός πίνακα A δύο διαστάσεων γίνεται με το να αποθηκεύεται η γραμμή $i-1$ αμέσως πριν την γραμμή i .

Παράδειγμα πίνακα δύο διαστάσεων στην μνήμη ($N \times N$):



Παράδειγμα εκτέλεσης:

Please give the size N of the square matrix: 2

Please give the elements row-wise:

1

2

2

1

Issymetric: 1

-- program is finished running --

Please give the size N of the square matrix: 3

Please give the elements row-wise:

1

2

3

4

5

6

7

8

9

Issymetric: 0

-- program is finished running --

Comments:

1) In **red** is the input of the user.

2) **Do not print** the last line. MARS prints it by default.

Άσκηση 2– Substring (4 μονάδες)

Να υλοποιηθεί η συνάρτηση `int substring(const char *str, const char *substr)` η οποία θα παίρνει ως ορίσματα το string `str`, και το string `substr`. Η συνάρτηση θα ψάχνει το μέγιστο τμήμα του `substr` (ξεκινώντας πάντα από την αρχή του `substring`) που βρίσκεται στο string `str` και θα επιστρέφει το μέγιστο μήκος του `substr` που βρέθηκε στο `str`.

Παράδειγμα:

`str: "abriaaklrivcariver"`

`substr: "river"`

Η συνάρτηση θα επιστρέψει 5.

Μέρος του `substr` που υπάρχει στο `str`, είναι το `ri`, το `riv`, και το `river`. Όμως, Το μέγιστο κομμάτι του `substr` που υπάρχει στο `str`, είναι το `river`, γι' αυτό η συνάρτηση επιστρέφει ως 5.

Ο χρήστης θα δίνει από την κονσόλα το αρχικό string, και το substring που ψάχνουμε, και ως έξοδο θα βγάζει το μέγιστο σε μήκος substring που βρέθηκε.

Παραδείγματα εκτέλεσης:

Please give string: **aafgbba**

Please give substring: **fab**

The max substring has length: 1

-- program is finished running --

Please give string: **kalmacc**

Please give substring: **vbd**

The max substring has length: 0

-- program is finished running --

Please give string: **alloavcllof**

Please give substring: **llof**

The max substring has length: 4

-- program is finished running --

Comments:

1) In **red** is the input of the user.

2) **Do not print** the last line. MARS prints it by default.

Άσκηση 3– Δέσμευση καταχωρητών (2 μονάδες)

```
int x, y, z;  
int *p = &x;
```

```
x = 10;  
*p = 12;  
y = x+1;  
z = x-3;
```

Έστω ότι αναπτύσσετε το τμήμα ενός compiler MIPS που υλοποιεί το κομμάτι της τοποθέτησης μεταβλητών σε καταχωρητές του MIPS (register allocator). Ο register allocator καθορίζει σε κάθε χρονική στιγμή ποιες μεταβλητές θα τοποθετηθούν σε κάθε έναν από τους υπάρχοντες καταχωρητές ενός επεξεργαστή. Ο register allocator προτιμά να τοποθετεί σε καταχωρητές μεταβλητές που χρησιμοποιούνται πολύ συχνά στο εγγύς μέλλον.

Έστω ότι η μεταβλητή `x` είναι μια τέτοια περίπτωση. Θα τοποθετούσατε την μεταβλητή `x` σε έναν καταχωρητή στον παραπάνω κώδικα; Ποιο είναι το πρόβλημα (εάν υπάρχει);

Υπόδειξη: Η απάντηση έχει να κάνει με τον pointer.