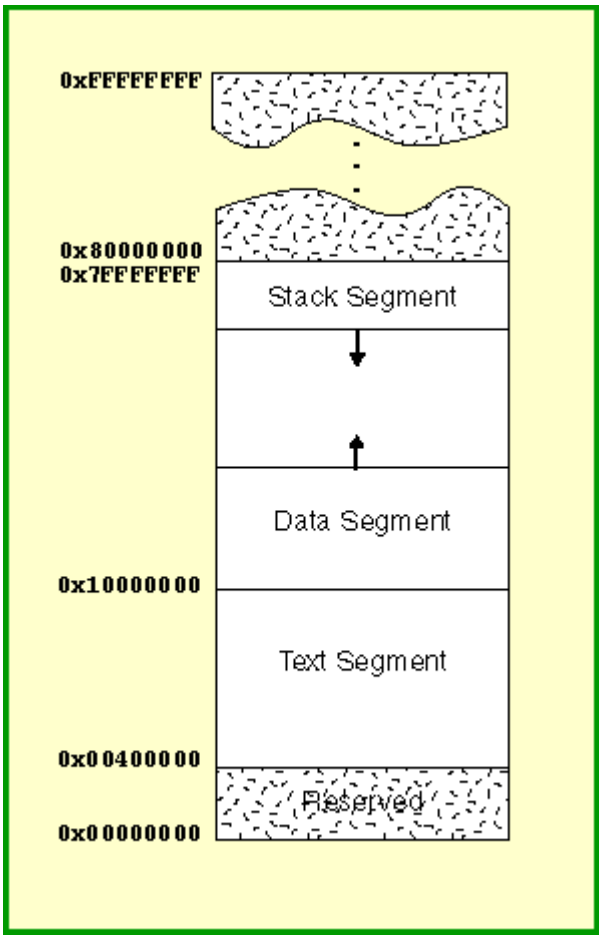
Тема 17. Архитектуры вычислительных машин и языки ассемблера, часть 6

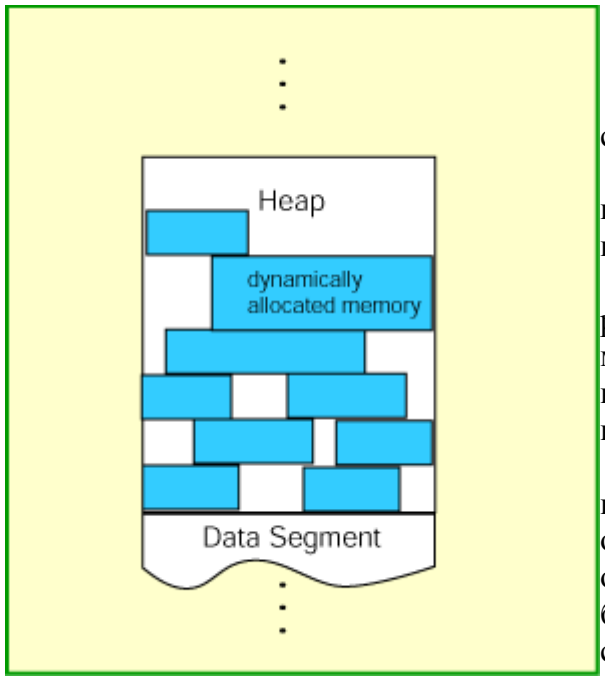
17.1 Динамическое распределение памяти

Как мы указывали ранее, непосредственно перед запуском программы загрузчик копирует машинный код из загружаемого модуля в текстовый сегмент памяти. Аналогично, данные копируются сегмент данных.

В исходном коде декларируется фиксированный объем памяти для сегмента данных. Однако, часто программе необходимо дополнительная память времени выполнения. Операционная система ищет блок свободной памяти и распределяет его программе. Это и есть **динамическое распределение памяти**.



В 32-битных MIPS-системах карта памяти процесса выглядит примерно так, как показано на рисунке. Как видно, между сегментом данных и сегментом стека расположена область нераспределенной памяти. Она во время выполнения программы может использоваться для стекового памяти (стрелка сверху вниз) или для динамических структур данных (стрелка снизу вверх).



Область памяти над сегментом данных, используемая для динамических структур, называется **кучей** (*Heap*).

Куча используется программами для разных целей, в качестве одной из которых может рассматриваться добавление узла к некоторой структуре данных или создание нового объекта.

Очевидно, что более не нужная программе динамическая память может быть освобождена. Как распределение, так и освобождение памяти производится по блокам. Порядок освобождения не обязан совпадать с порядком распределения памяти.

Отсюда возникает проблема **фрагментации** пространства памяти кучи, когда общий объем свободных участков достаточен для распределения, но запрос не может быть выполнен. Пример фрагментации

приведен на следующем рисунке.

Симулятор SPIM поддерживает (симулирует) динамическое распределение памяти. Следующий код иллюстрирует это с использованием системного вызова 9.

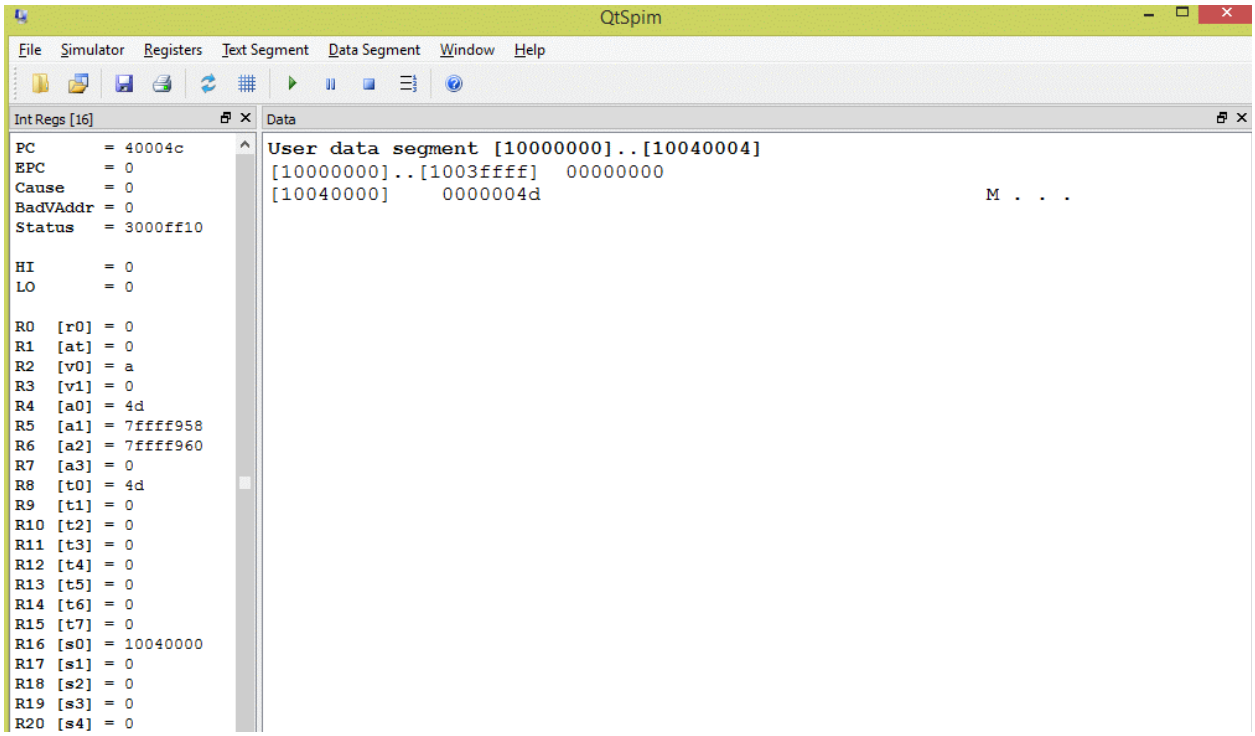
```
li      $a0, xxx    # $a0 contains the number of bytes you need.
                    # This must be a multiple of four.
li      $v0, 9      # code 9 == allocate memory
syscall             # call the service.
                    # $v0 <-- the address of the first byte
                    # of the dynamically allocated block
```

В качестве возвращаемого значения (регистр $*v*0) будет получен адрес первого байта распределенного блока памяти. Размер блока в байтах указывается в регистре $*a*0.

Следующий пример содержит код программы, которая запрашивает 4 байта памяти, затем сохраняет базовый адрес блока в регистре $*s*0. После этого записывает по этому адресу целое число с использованием инструкции *sw* $*t*0,0($*s*0). Содержимое блока памяти затем загружается в регистр $*a*0, чтобы затем вывести его значение на экран терминала.

```
# malloc.s
#
# Allocate one block of memory, put an integer into it,
# print out the integer.

        .text
        .globl  main

main:
        li      $v0, 9              # (1) Allocate a block of memory
        li      $a0, 4              # 4 bytes long
        syscall                     # $v0 <-- address
        move    $s0, $v0            # (2) Make a safe copy

        li      $t0, 77             # (3) Store value 77
        sw      $t0, 0($s0)         # into the block

        lw      $a0, 0($s0)         # (4) Load from the block
        li      $v0, 1              # into $a0.
        syscall                    # (5) Print the integer

        li      $v0, 10             # Return to OS
        syscall

## end of file
```

На следующем рисунке показан результат работы этой программы в симуляторе SPIM. Адрес блока памяти (10040000) в регистре $*s*0. Обратившись по данному адресу в секции данных, находим блок из 4 байт. Программа разместила в нем значение 0x4d (или 77 в десятичной системе).



17.2 Записи и структуры

Как известно, **запись** — это блок памяти, содержащий несколько элементов данных разных типов. В языке C мы называем этот блок **структурой**. Их зачастую хранят в динамической памяти. Например, нам необходимо работать со списками, содержащих следующие элементы данных:

```
struct
{
  int age;
  int pay;
  int class;
};
```

Очевидно, каждый элемент данных содержит три целочисленных значения. Каждый такой блок памяти имеет размер 12 байт. К полям структуры можно обращаться по смещению. Так, *age* расположен с нулевым смещением относительно адреса блока, *pay* – со смещением в 4 байта, *class* – 8 байт.

Следующий пример иллюстрирует копирование элементов структур.

```
# StructCopy.s
#
        .text
        .globl  main

main:
        # create the first struct
        li      $v0, 9              # allocate memory
        li      $a0, 12             # 12 bytes
        syscall                     # $v0 <-- address
        move    $s1, $v0            # $s1 first struct

        # initialize the first struct
        li      $t0, 34             # store 34
        sw      $t0, 0($s1)         # in age
        lw      $t0, pay            # store 24000
```

```
        sw      $t0, 4($s1)         # in pay
        li      $t0, 12             # store 12
        sw      $t0, 8($s1)         # in class

        # create the second struct
        li      $v0, 9              # allocate memory
        li      $a0, 12             # 12 bytes
        syscall                     # $v0 <-- address
        move    $s2, $v0            # $s2 second struct

        # copy data from first struct to second
        lw      $t0, 0($s1)         # copy age from first
        sw      $t0, 0($s2)         # to second struct

        lw      $t0, 4($s1)         # copy pay from first

        sw      $t0, 4($s2)         # to second struct

        lw      $t0, 8($s1)         # copy class from first

        sw      $t0, 8($s2)         # to second struct

        li      $v0, 10             # return to OS
        syscall

        .data
pay:    .word   24000               # rate of pay, in static memory
```

Следует различать работу с адресами и содержимым блоков памяти с этими адресами. Можно попробовать дополнить наш пример кодом вывода на экран терминала. Заменяем две последние строчки кода на следующие:

```
        # write out the second struct
        la       $a0, agest         # print "age:"
        li       $v0, 4             # print string service
        syscall
        lw       $a0, 0($s2)        # print age
        li       $v0, 1             # print int service
        syscall

        li       $v0, 10            # return to OS
        syscall

        .data
pay:    .word   24000               # rate of pay, in static memory
agest:  .asciiz "age:   "
```

Однако более разумным представляется написание подпрограммы для вывода содержимого полей структур. Это может выглядеть следующим образом, хотя мы ограничимся полем *age*:

```
# Subroutine PStruct: print a struct
#
# Registers on entry: $a0 -- address of the struct
#                     $ra -- return address
#
# Registers:          $s0 -- address of the struct
#
        .text

PStruct:
        sub     $sp, $sp, 4         # push $s0
```

```
        sw      $s0, ($sp)          # onto the stack

        move    $s0, $a0            # make a safe copy
                                    # of struct address
        la      $a0, agest          # print "age:"
        li      $v0, 4
        syscall
        lw      $a0, 0($s0)         # print age
        li      $v0, 1
        syscall

        add     $sp, $sp, 4          # restore $s0 of caller
        lw      $s0, ($sp)
        jr      $ra                # return to caller

        .data
agest:  .asciiz "age:   "
```

Так как регистр $*s*0 используется, его содержимое следует сохранить в стеке. Поскольку наша подпрограмма, никаких других подпрограмм не вызывает, поэтому нет смысла сохранять в стеке регистр $*ra*. Аргументом нашей подпрограммы является адрес всей структуры.

Вызов подпрограммы также не представляет большой проблемы.

```
 . . . .

        # $s1 contains the address of the first struct
        # $s2 contains the address of the second struct
        #
        # write out the first struct
        move    $a0, $s1
        jal     PStruct

        # write out the second struct
        move    $a0, $s2
        jal     Pstruct
```

Очевидно, подпрограмме передается не сама структура, а ее базовый адрес. Более полная версия нашего кода приведена далее.

```
# StructPrint.s
#
# Allocate memory for a struct and then initialize it.
# Allocate memory for a second struct and copy data
# from the first into it. Print both structs

        .text
        .globl  main

main:
        # create the first struct
        li      $v0, 9              # allocate memory
        li      $a0, 12             # 12 bytes
        syscall                     # $v0 <-- address
        move    $s1, $v0            # $s1 first struct

        # initialize the first struct
        li      $t0, 34             # store 34
        sw      $t0, 0($s1)         # in age
        lw      $t0, pay            # store 24000
        sw      $t0, 4($s1)         # in pay
        li      $t0, 12             # store 12
        sw      $t0, 8($s1)         # in class
```

```
        # create the second struct
        li      $v0, 9              # allocate memory
        li      $a0, 12             # 12 bytes
        syscall                     # $v0 <-- address
        move    $s2, $v0            # $s2 second struct

        # copy data from first struct to second
        lw      $t0, 0($s1)         # copy age from first
        sw      $t0, 0($s2)         # to second struct
        lw      $t0, 4($s1)         # copy pay from first
        sw      $t0, 4($s2)         # to second struct
        lw      $t0, 8($s1)         # copy class from first
        sw      $t0, 8($s2)         # to second struct

        # write out the first struct
        move    $a0, $s1
        jal     PStruct

        # write out the second struct
        move    $a0, $s2
        jal     PStruct

        li      $v0, 10             # return to OS
        syscall

        .data
pay:    .word   24000               # rate of pay, in static memory

        .text
# Subroutine PStruct: print a struct
#
# Registers: $a0 -- address of the struct
#            $ra -- return address
PStruct:
        sub     $sp, $sp, 4         # push $s0
        sw      $s0, ($sp)          # onto the stack

        move    $s0, $a0            # safe copy of struct address

        la      $a0, agest          # print "age:"
        li      $v0, 4
        syscall
        lw      $a0, 0($s0)         # print age
        li      $v0, 1
        syscall

        la      $a0, payst          # print " pay: "
        li      $v0, 4
        syscall
        lw      $a0, 4($s0)         # print pay
        li      $v0, 1
        syscall

        la      $a0, classt          # print " class: "
        li      $v0, 4
        syscall
        lw      $a0, 8($s0)         # print class
        li      $v0, 1
        syscall

        la      $a0, lf             # end the line
        li      $v0, 4
        syscall
```

```
        add     $sp, $sp, 4         # restore $s0 of caller
        lw      $s0, ($sp)
        jr      $ra                 # return to caller

        .data
agest:  .asciiz "age:   "
payst:  .asciiz "  pay:   "
classt: .asciiz "  class: "
lf:     .asciiz "\n"

## end of file
```

Литература и дополнительные источники к Теме 17

1. MIPS32 Architecture - https://imgtec.com/mips/architectures/mips32/
2. http://labs.cs.upt.ro/labs/so2/html/resources/nachos-doc/mipsf.html
3. http://logos.cs.uic.edu/366/notes/mips%20quick%20tutorial.htm
4. http://www.mrc.uidaho.edu/mrc/people/jff/digital/MIPSir.html