



OS Lab 4

▼ Status	approved
☑ checkbox	☑
▼ class	OS
📅 due date	@Mar 3, 2021

Task

Напишите программу, которая вставляет строки, введенные с клавиатуры, в список. Память под узлы списка выделяйте динамически с использованием `malloc(3)`. Ввод завершается, когда в начале строки вводится точка `(.)`. Затем все строки из списка выводятся на экран.

Подсказка: Объявите массив символов размера, достаточного чтобы вместить самую длинную введенную строку. Используйте `fgets(3)`, чтобы прочитать строку, и `strlen(3)`, чтобы определить ее длину. Помните, что `strlen(3)` не считает нулевой символ, завершающий строку. После определения длины строки, выделите блок памяти нужного размера и внесите новый указатель в список.

Notes

```
#include <sys/types.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

typedef struct Node {
    char *data;
    struct Node* nextNode;
} Node;

void freeNode(Node* node){
    if (node == NULL){
        return;
    }
    node->nextNode = NULL;
    free(node->data);
    free(node);
}

Node* createNode(){
    Node* node = (Node*)malloc(1 * sizeof(Node));
    if (node == NULL){
        perror("malloc(3C) failed to allocate a new node\n");
        return NULL;
    }
    node->nextNode = NULL;
    node->data = NULL;
    return node;
}

Node* fillNode(char* line){
    Node* node = createNode();

    node->data = (char*)calloc((strlen(line) + 1), sizeof(char));
    if (node->data == NULL){
        free(node);
        return NULL;
    }

    memcpy(node->data, line, (strlen(line) + 1) * sizeof(char));

    node->nextNode = NULL;
    return node;
}

void freeList(Node* head){
    if (head == NULL){
        return;
    }

    Node* nextNode = head->nextNode;
    Node* savedNode = NULL;
    while (nextNode != NULL){
        savedNode = nextNode->nextNode;
        freeNode(nextNode);
        nextNode = savedNode;
    }

    freeNode(head);
}

void printList(Node* head){
    Node* node;
```

```
for (node = head->nextNode; node != NULL; node = node->nextNode){
    printf("%s", node->data);
}

}

int main(){
    char buffer[BUFSIZ];

    Node* head = createNode();
    if(head == NULL){
        perror("No available memory\n");
        return 0;
    }
    Node* currentNode = head;

    printf("Please enter your strings here: \n");

    while (fgets(buffer, BUFSIZ, stdin) != EOF){

        if(buffer[0] == '.'){
            break;
        }


        currentNode->nextNode = fillNode(buffer);
        if(currentNode->nextNode == NULL){
            perror("No available memory\n");
            freeList(head);
            return 0;
        }

        currentNode = currentNode->nextNode;
    }

    printList(head);
    freeList(head);

    return 0;
}
```

Как работает malloc?

 Есть переменная, которая хранит указатель на голову кольцевого связного списка свободных блоков. (так называемая стратегия first fit).
При вызове malloc мы идём с позиции головы, пока не найдём блок, режем найденный блок на 2 части: нужную и обрезанную.
Нужный блок помечается как занятый. Они добавляются в двусный связный список. Возвращаем обрезанный блок в список свободных блоков и ставим на него голову, чтобы в следующий раз обход начался с этой позиции.
Возвращаем адрес на начало нужного блока.

Пример реализации

brk()

BRK(2)System Calls

BRK(2)NAME

brk, sbrk - change the amount of space allocated for the calling process's data segment

SYNOPSIS

```
#include <unistd.h>

int brk(void *endds);
void *sbrk(intptr_t incr);
```

The brk() and sbrk() functions are used to change dynamically the amount of space allocated for the calling process's data segment (see exec(2)). The change is made by resetting the process's break value and allocating the appropriate amount of space. The break value is the address of the first location beyond the end of the data segment. The amount of allocated space increases as the break value increases. Newly allocated space is set to zero. If, however, the same memory space is reallocated to the same process its contents are undefined.

When a program begins execution using execve() the break is set at the highest location defined by the program and data storage areas.

The getrlimit(2) function may be used to determine the maximum permissible size of the data segment; it is not possible to set the break beyond the rlim_max value returned from a call to getrlimit(), that is to say, "end + rlim.rlim_max." See end(3C).

The brk() function sets the break value to endds and changes the allocated space accordingly.

RETURN VALUES

Upon successful completion, brk() returns 0. Otherwise, it returns -1 and sets errno to indicate the error.

Upon successful completion, sbrk() returns the prior break value. Otherwise, it returns (void *)-1 and sets errno to indicate the error.

ERRORS

The brk() and sbrk() functions will fail and no additional memory will be allocated if:

ENOMEM The data segment size limit as set by setrlimit() (see getrlimit(2)) would be exceeded; the maximum possible size of data segment (compiled into the system) would be exceeded; insufficient space exists in the swap area to support the expansion; or the new break value would extend into an area of the address space defined by some previously established mapping (see mmap(2)).

EAGAIN Total amount of system memory available for private pages is exhausted.

The `sbrk()` function adds `incr` function bytes to the break value and changes the allocated space accordingly. The `incr` function can be negative, in which case the amount of allocated space is decreased.

temporarily insufficient. This may occur even though the space requested was less than the maximum data segment size (see `ulimit(2)`).



Механизм виртуальной памяти. Этот механизм реализует иллюзию большего объема памяти программы, он связывает оперативную память и внешнюю память. Виртуальная память реализуется совместно процессором и операционной системой

Если кратко, то все адреса в программах - виртуальные, то есть если пойти по какому нибудь указателю физически в оперативную память (если бы ещё так можно было сделать), то там были бы совсем другие данные

Каждый процесс живёт в своем таком виртуальном адресном пространстве (где-то была картинка в лекциях)

Операционная система, когда процесс обращается по такому виртуальному адресу, транслирует его уже в реальный физический адрес в оперативной памяти.

Причём до обращения нужные данные могли быть кэшированы на диск (если они редко использовались; чтобы не занимали место в ОЗУ) и тогда система сначала их подгрузит в ОЗУ, потом уже оттранслирует адрес

Это нужно во первых чтобы защитить процессы друг от друга (все обращения в память контролирует ОС), во-вторых у нас (программистов) как будто большая оперативка (на 64 битных системах адрес на самом деле 48-битный, то есть можно адресовать 2^{48} байт, что намного больше размера ОЗУ у обычных пользователей), хотя на самом деле система может подгружать/отгружать данные на диск, незаметно для нас

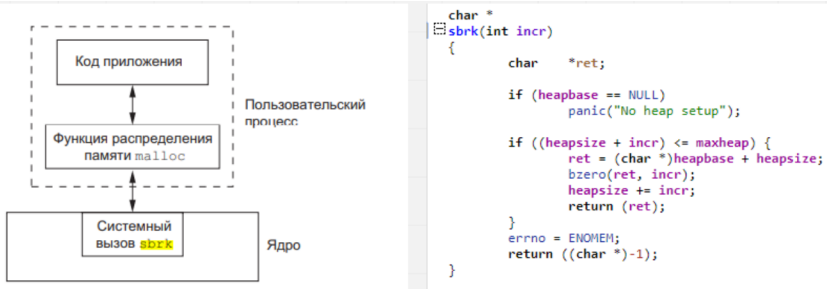
Таненбаум:

10.4.1. Фундаментальные концепции

У каждого процесса в системе Linux есть адресное пространство, состоящее из трех логических сегментов: текста, данных и стека. Пример адресного пространства процесса изображен на рис. 10.6 (процесс А). **Текстовый сегмент** (text segment) содержит машинные команды, образующие исполняемый код программы. Он создается компилятором и ассемблером при трансляции программы (написанной на языке высокого уровня, например C или C++) в машинный код. Как правило, текстовый сегмент доступен только для чтения. Самомодифицирующиеся программы вышли из моды примерно в 1950 году, так как их было слишком сложно понимать и отлаживать. Таким образом, не изменяются ни размеры, ни содержание текстового сегмента.

Сегмент данных (data segment) содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных и неинициализированных данных. По историческим причинам вторая часть называется **BSS** (Block Started by Symbol). Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске программы. Все переменные в BSS должны быть инициализированы в нуль после загрузки.

Например, на языке C можно объявить символьную строку и в то же время проинициализировать ее. Если программа запускается, она предполагает, что эта строка уже имеет свое начальное значение. Чтобы реализовать это, компилятор назначает строке определенное место в адресном пространстве и гарантирует, что в момент запуска программы по этому адресу будет располагаться необходимая строка. С точки зрения операционной системы инициализированные данные не отличаются от текста программы — и тот и другой сегменты содержат сформированные компилятором последовательности битов, которые должны быть загружены в память при запуске программы.



Стандарт POSIX не определяет системные вызовы для управления памятью. Эту область посчитали слишком машинно зависимой, чтобы ее стандартизировать. Вместо этого просто сделали вид, что проблемы не существует, и заявили, что программы, которым требуется динамическое управление памятью, могут использовать библиотечную процедуру `malloc` (определенную стандартом ANSI C). Таким образом, вопрос реализации процедуры `malloc` был вынесен за пределы стандарта POSIX. В некоторых кругах такой подход считают переключиванием бремени решения проблемы на чужие плечи. На практике в большинстве систем Linux есть системные вызовы для управления памятью. Наиболее распространенные системные вызовы перечислены в табл. 10.5. Системный вызов `brk` указывает размер сегмента данных, задавая адрес первого байта за его пределами. Если новое значение больше старого, то сегмент данных увеличивается, в противном случае он уменьшается.

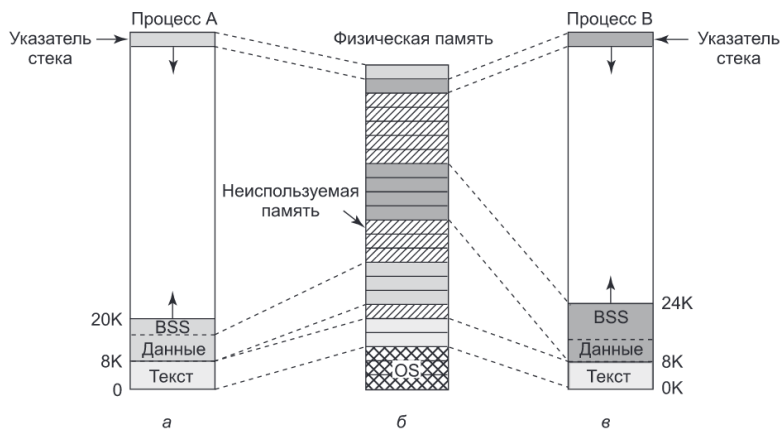


Рис. 10.6. а — виртуальное адресное пространство процесса А; б — физическая память; в — виртуальное адресное пространство процесса В

В отличие от текстового сегмента, который не может изменяться, сегмент данных изменяться может. Программы все время модифицируют свои переменные. Более того, многим программам требуется динамическое выделение памяти во время выполнения. Для этого операционная система Linux разрешает сегменту данных расти при выделении памяти и уменьшаться при освобождении памяти. Программа может установить размер своего сегмента данных при помощи системного вызова `brk`. Таким образом,

чтобы выделить больше памяти, программа может увеличить размер своего сегмента данных. Этим системным вызовом активно пользуется библиотечная процедура `malloc` языка C, используемая для выделения памяти. Дескриптор адресного пространства процесса содержит информацию о диапазоне динамически выделенных областей памяти процесса (который обычно называется **кучей** — heap).



Process virtual memory for user data structures is allocated from the heap segment, which resides

Figure 5.3 illustrates a process's virtual address space.

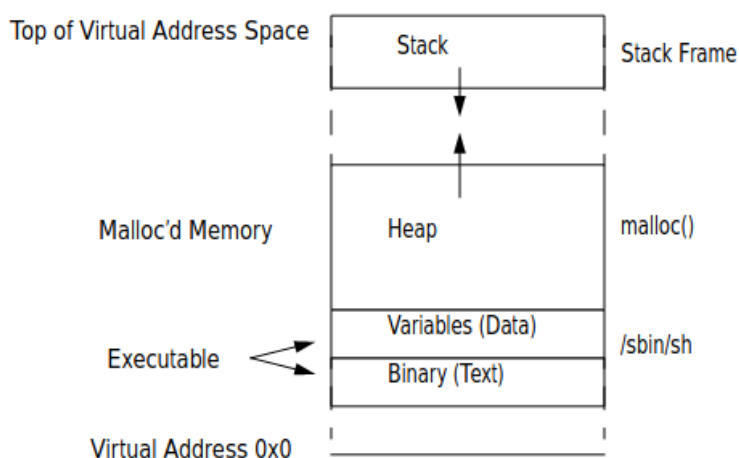


Figure 5.3 Process Virtual Address Space

Table 5-1 Maximum Heap Sizes

Solaris Version	Maximum Heap Size	Notes
Solaris 2.5	2 Gbytes	
Solaris 2.5.1	2 Gbytes	
Solaris 2.5.1 with patch 103640-08 or greater	3.75 Gbytes	Need to be root to increase limit above 2 GB with <code>ulimit(1M)</code> .
Solaris 2.5.1 with patch 103640-23 or greater	3.75 Gbytes	Do not need to be root to increase limit.
Solaris 2.6	3.75 Gbytes	Need to increase beyond 2 GB with <code>ulimit(1M)</code> .
Solaris 2.7 32 bit mode	3.75 Gbytes 3.90 Gbytes	(Non-sun4u platform) (sun4u platforms)
Solaris 2.7 64 bit mode	16 Tbytes on UltraSPARC-I and -II	Virtually unlimited.

above the executable data segment. The heap starts out small and then grows as virtual memory is allocated. The heap grows in units of pages and is simply a large area of virtual memory available for reading and writing. A single, large, virtual memory area is difficult to program to, so a general-purpose memory allocator manages the heap area; thus, arbitrarily sized memory objects can be allocated and freed. The general-purpose memory allocator is implemented with `malloc()` and related library calls. A process grows its heap space by making the `sbrk()` system call. The `sbrk()` system call grows the heap segment by the amount requested each time it is called. A user program does not need to call `sbrk()` directly because the `malloc()` library calls `sbrk()` when it needs more space to allocate from. The `sbrk()` system call is shown below.

```
void *sbrk(intptr_t incr);
```

Memory pages are allocated to the process heap by zero-fill-on-demand and then remain in the heap segment until the process exits or until they are stolen by the page scanner. Calls to the memory allocator `free()` function do not return physical memory to the free memory pool; `free()` simply marks the area within the heap space as free for later use. For this reason, it is typical to see the amount of physical memory allocated to a process grow, but unless there is a memory short-age, it will not shrink, even if `free()` has been called. The heap can grow until it collides with the memory area occupied by the shared libraries. The maximum size of the heap depends on the platform virtual memory layout and differs on each platform. In addition, on 64-bit platforms, processes may execute in either 32- or 64-bit mode. As shown in Figure 5.5 on page 134, the size of the heap can be much larger in processes executing in 64-bitmode. Table 5-1 shows the maximum heap sizes and the operating system requirements that affect the maximum size.

Reading list

