# Buffer overflows

A buffer overflow occurs when a program copies data into a variable for which it has not allocated enough space.

Ex:

```
char buf[80];
printf("Enter your first name:");
scanf("%s", buf);
```

What user's first name could be that long?

Regardless of the ultimate size that you choose, the code segment is still susceptible to a buffer overflow.

## Buffer overflows

A simple way to fix this problem.

The format specification limits the input string to one less than the size of the variable, allowing room for the string terminator.

Ex:

```
char buf[8];
printf("Enter your first name:");
scanf("%7s", buf);
```

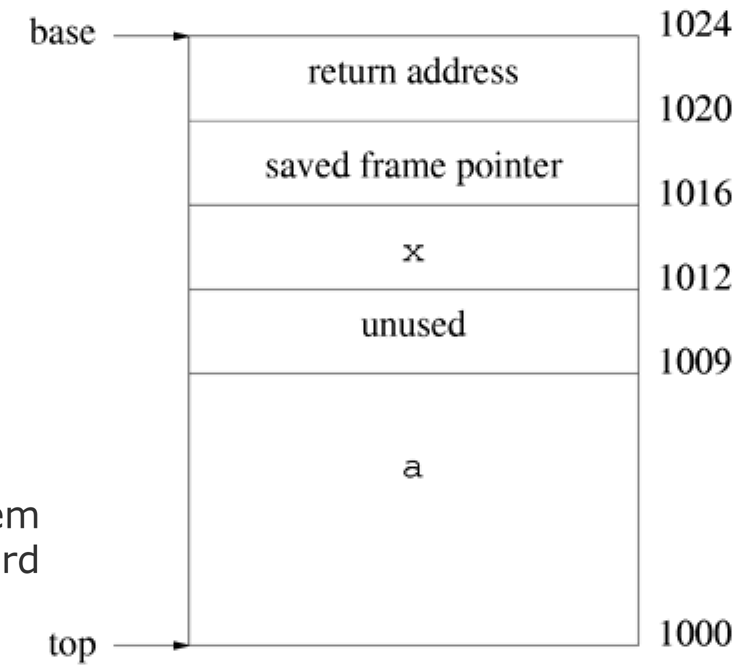
## Consequences of buffer overflows

Ex:

```
int checkpass(void){
int x;
char a[9];
x = 0;
fprintf(stderr,"a at %p and\nx at %p\n", (void *)a, (void *)&x);
printf("Enter a short word: ");
scanf("%s", a);
if (strcmp(a, "mypass") == 0)
x = 1;
return x;
}
```

Note: integers and pointers are 4 bytes
12 bytes for array a

But the program specifies only 9 bytes, so that the system can maintain a stack pointer that is aligned on a word boundary.



## Programs, Processes and Threads

A program is a prepared sequence of instructions to accomplish a defined task.

A process is an instance of a program whose execution has started but has not yet terminated.

The C source program contains exactly one main function

Also it contain variable and function declarations, type and macro definitions (e.g., typedef) and preprocessor commands (e.g., #ifdef, #include, #define).

## Programs, Processes and Threads

- The C compiler translates each source file into an object file.
- The compiler then links the individual object files with the necessary libraries to produce an executable module.

- When a program run or executed, the operating system copies the executable module into a program image in main memory.

When does a program become a process?

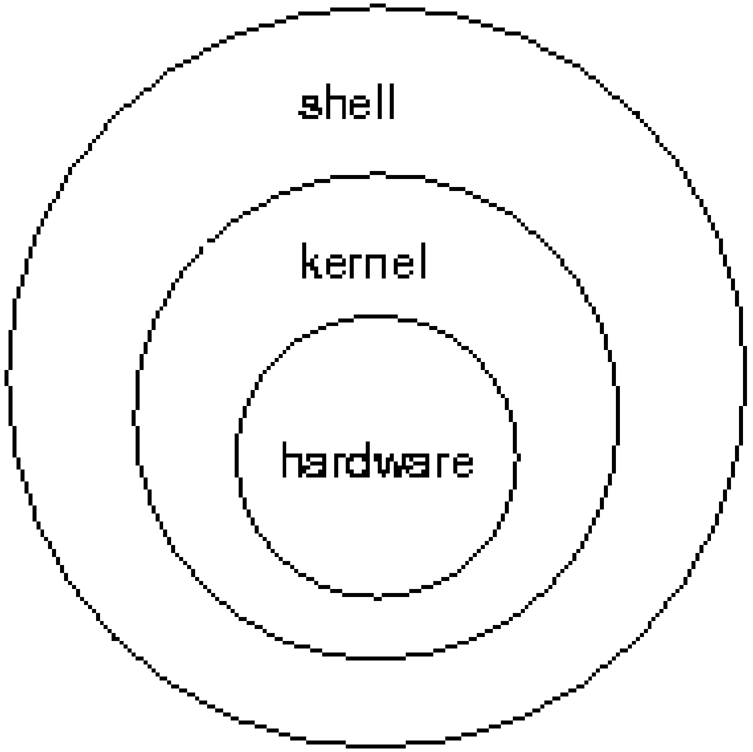
## Programs, Processes and Threads

- The operating system reads the program into memory.
- Allocate an ID (the process ID) so that the o.s can distinguish among individual processes.
- The o.s keeps track of the process IDs and corresponding process states and uses the information to allocate and manage resources for the system.
- The o.s also manages the memory occupied by the processes and the memory available for allocation.

## Programs, Processes and Threads

When the o.s has

- added the appropriate information in the kernel data structures
- allocated the necessary resources to run the program code,
- the program has become a process.

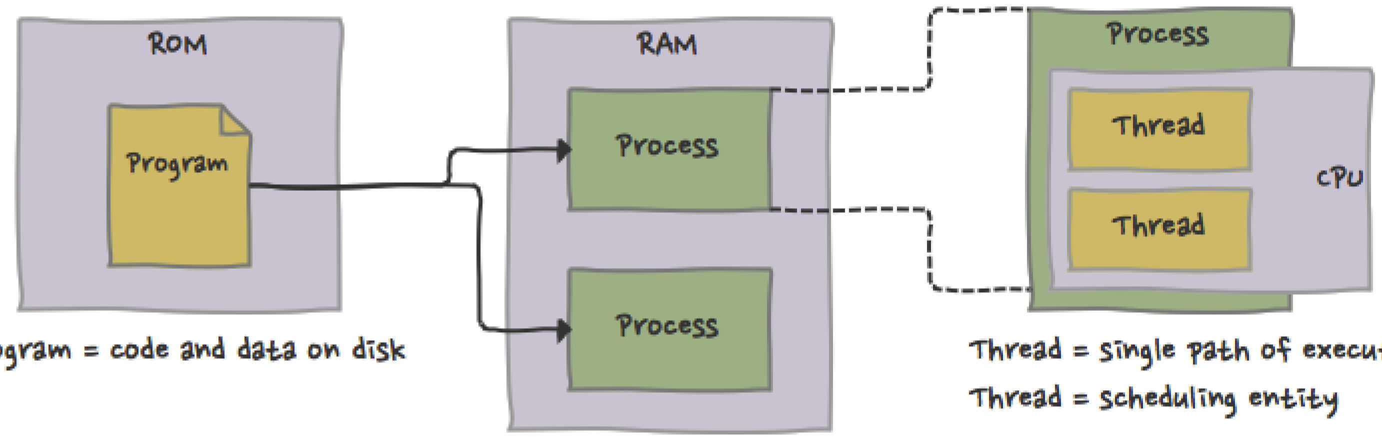


## Programs, Processes and Threads

A process has

- an address space (memory it can access)
- at least one flow of control called a thread.

The variables of a process can either remain in existence for the life of the process (static storage) or be automatically allocated when execution enters a block and deallocated when execution leaves the block (automatic storage).



Program = code and data on disk

Thread = single path of execution

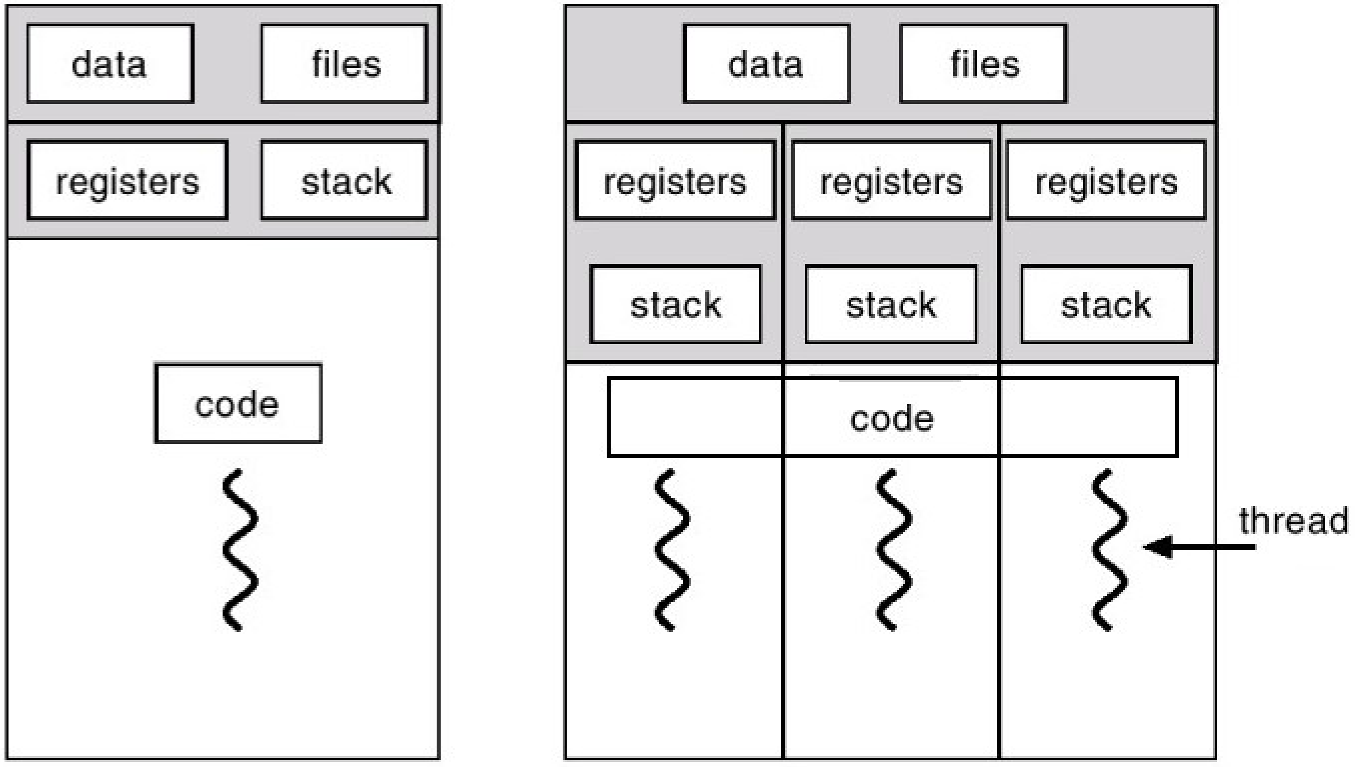
Thread = scheduling entity

Process = instance of a program

Process = organization of a program in RAM

## Programs, Processes and Threads

A process starts with a single flow of control that executes a sequence of instructions.
The processor(CPU) program counter keeps track of the next instruction to be executed by that processor.
The CPU increments the program counter after fetching an instruction and may further modify it during the execution of the instruction
Ex: when a branch occurs.
Multiple processes may reside in memory and execute concurrently, almost independently of each other.
For processes to communicate  they must explicitly interact through operating system constructs such as the file system, pipes, shared memory or a network


data
files
registers
stack
code
data
files
registers
registers
registers
stack
stack
stack
code
thread

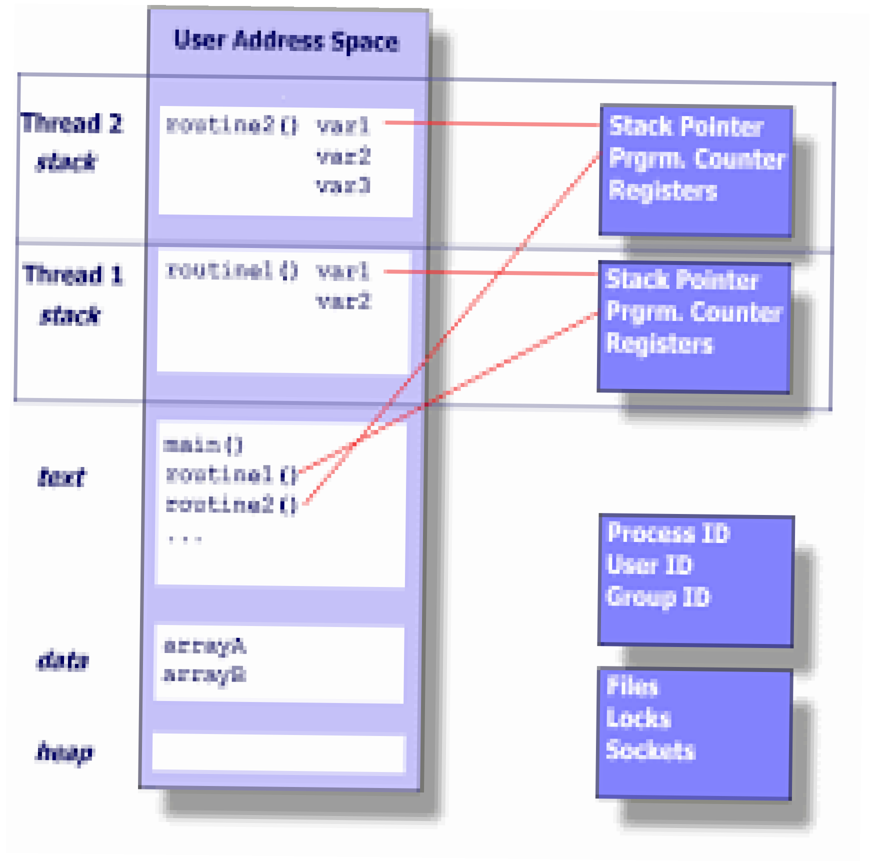
threaded

# Threads and Thread of Execution

When a program executes, the value of the program counter determines which process instruction is executed next.

The resulting stream of instructions, called a thread of execution.

Is represented by the sequence of instruction addresses assigned to the program counter during the execution of the program's code.



User Address Space
Thread 2 stack
routine2() var1 var2 var3
Thread 1 stack
routine1() var1 var2
text
main() routine1() routine2() ...
data
arrayA arrayB
heap
Stack Pointer Prgrm. Counter Registers
Stack Pointer Prgrm. Counter Registers
Process ID User ID Group ID
Files Locks Sockets

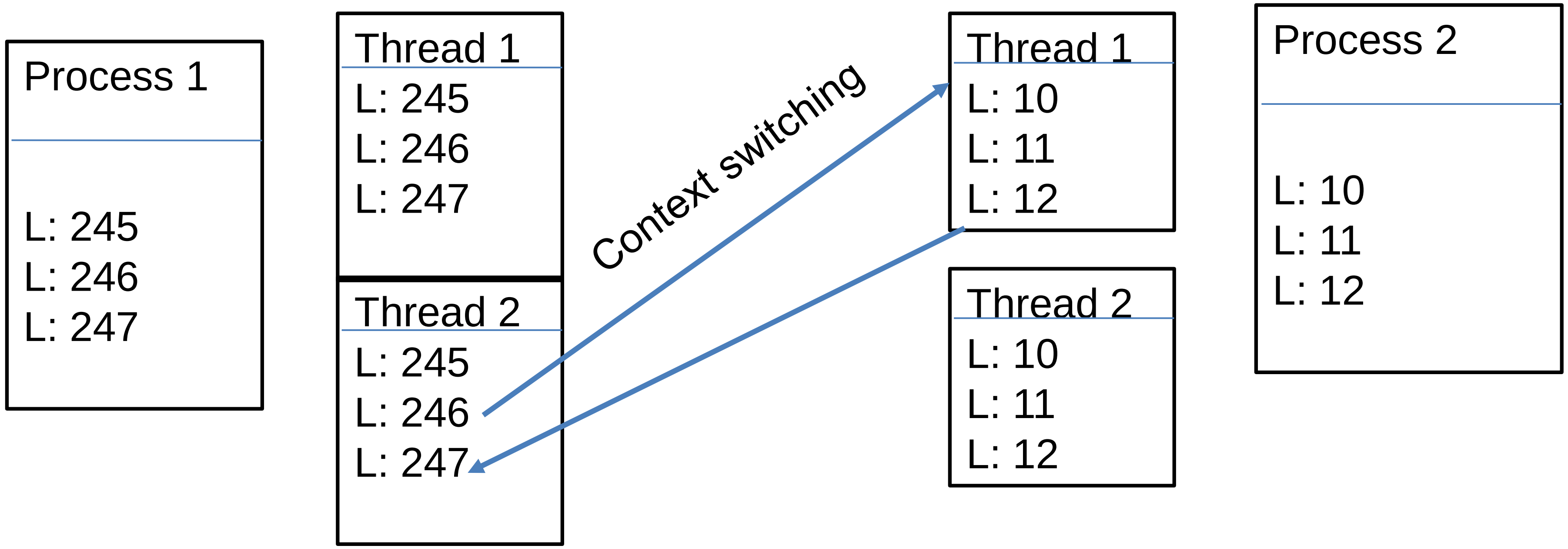
Process 1 executes
statements
245,
246 and
247 in a loop.

Its thread of execution can be represented as
$245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1, 247_1 \ldots,$

where the subscripts identify the thread of execution as belonging to process 1

# Threads and Thread of Execution

The sequence of instructions in a thread of execution is appears to the process as an uninterrupted stream of addresses.
However, the threads of execution from different processes are intermixed.
Point at which execution switches from one process to another is called a *context switch*



## Threads and Thread of Execution

- ✓ multiple threads execute within the same process on machine with multiple processor, improve program performance.

- ✓ It avoid context switches and allow sharing of code and data.

??threads are sometimes called lightweight processes.

A thread is an abstract data type that represents a thread of execution within a process.
Thread has its own execution stack, program counter value, register set and state.

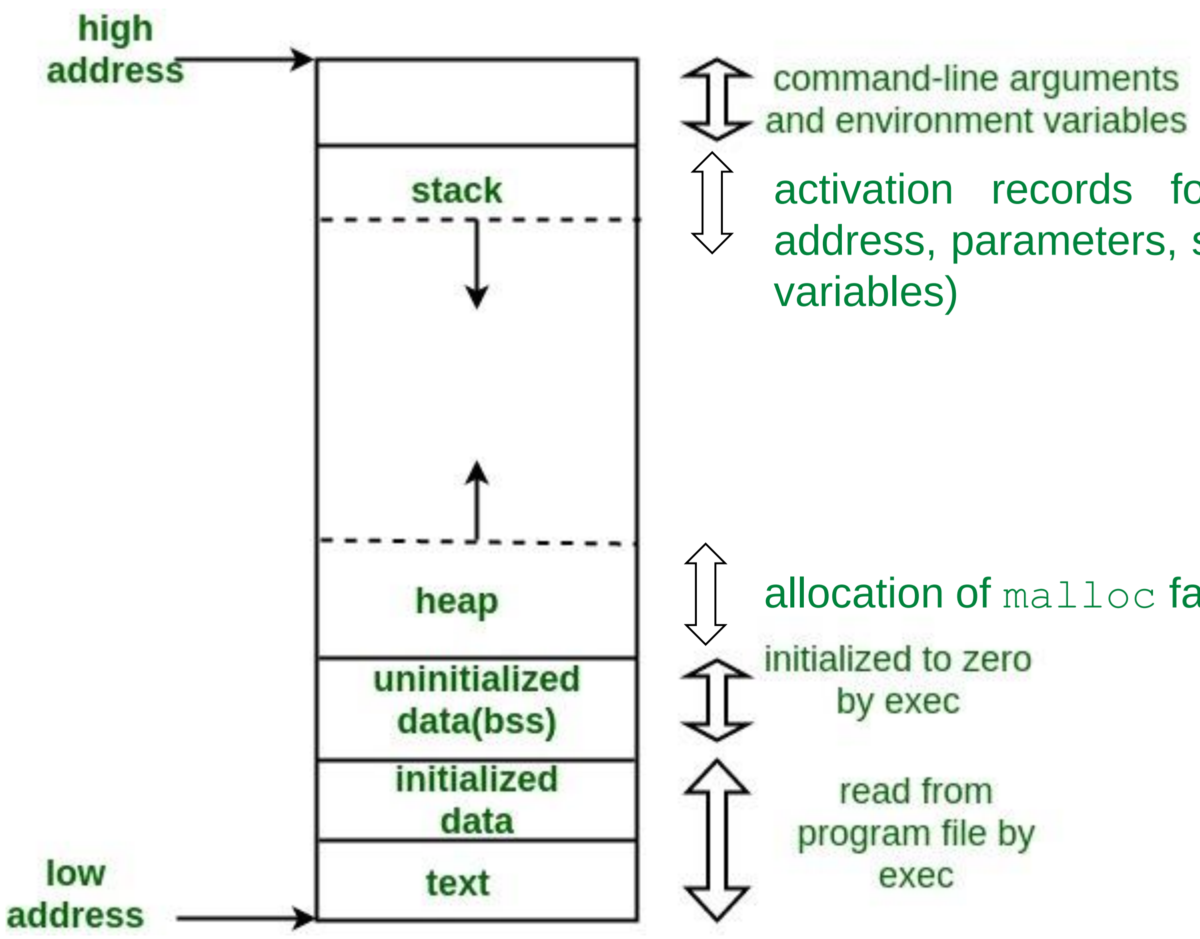
Using many threads in a process programmer achieve parallelism with low overhead.
But need synchronization, as they reside in the same process address space and share process resources.

# Process Vs Thread

| Process | Thread |
|---|---|
| Process are heavy weight operation | Threads are light weight operation |
| Every process has its own memory space | Threads use the memory of the process they belong |
| Inter process communication is slow as different process have different memory space | Inter thread communication is fast as threads of same process same memory address of the process they belong to. |
| Context switching between the process is more expensive | Context switching between threads of the same process is less expensive |
| Process don’t share the memory with other process | Threads share the memory with other threads of same process |

# Layout of a Program Image

The contiguous block of memory at where the executable program occupy is called as *program image*


high address
stack
heap
uninitialized data(bss)
initialized data
text
low address
command-line arguments and environment variables
initialized to zero by exec
read from program file by exec

activation records for function calls (return address, parameters, saved registers, automatic variables)

allocation of `malloc` family

## Layout of a Program Image

*Sections of program image :*

- The program text or code in low-order memory.
- The initialized and uninitialized static variables
- Other sections include the heap, stack and environment.

A typical memory representation of C program consists of following sections.

1. Text segment
2. Initialized static data segment
3. Uninitialized static data segment
4. Stack
5. Heap

# Layout of a Program Image

**1. Text Segment:**

- A text segment (code segment), is one of the sections in memory, which contains executable instructions of a program.
- It is placed below the heap or stack in order to prevent heaps and stack overflows.
- It is sharable so that only a single copy needs to be in memory for frequently executed programs.
- Often it is read-only, to prevent a program from accidentally modifying its instructions.

# Layout of a Program Image

**2. Initialized static data segment**

- It contains the global variables and static variables that are initialized by the programmer.
- This section is not read-only, since the values of the variables can be altered at run time.

Ex: static int i = 10 , global int i = 10

# Layout of a Program Image

**3. Uninitialized static data segment (**“block started by symbol(bss)**)**

- Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing

- All uninitialized data starts at the end of the data segment

- It contains all global variables and static variables that are initialized to zero or do not have explicit initialization in source code.

Ex: static int i , global int j

# Layout of a Program Image

**4. Heap**

- Dynamic memory allocation usually takes place in this segmnet.
- Begins at the end of the BSS segment
- Grows towards larger address space
- It is managed by malloc, realloc, and free
- Storage allocated on the heap persists **until it is freed or until the program exits.**
- the brk and sbrk system calls to adjust its size

## Layout of a Program Image

**5. Stack**

- It is adjoined the heap area and grew in opposite direction.
- The stack area contains the program stack, located in the higher parts of memory.
- A “stack pointer” register tracks the top of the stack, and adjusted each time when value pushed to stack
- The set of values pushed for one function call is termed a “stack frame or activation record;
- Each function call creates a new activation record on the stack.

# Layout of a Program Image

Activation record is a block of memory allocated on the top of the process stack to hold the execution context of a function during a call.

It is removed from the stack when the function returns, such that the last-called-first-returned (LIFO) order for nested function calls.

| **Activation Record (AR)** |
|---|
| Return address |
| Parameters |
| Status information |
| Some of the CPU register values |
| Automatic variable |

Format of an AR depends on the H/W and on the programming language size(1) command reports the sizes (in bytes) of the text, data, and bss segments.

```
#include <stdio.h>
int main(void)
{
    return 0;
}
$ gcc memory-layout.c -o memory-layout
$ size memory-layout
```

```
//add one global variable (uninitialized)
#include <stdio.h>
int g;
int main(void)
{    return 0;
}
//Check bss
```

```
//add one static variable (uninitialized)
#include <stdio.h>
int g;
int main(void)
{  static int i;
    return 0;
}
//Check bss
```

```
//one static variable (initialized)
#include <stdio.h>
int g;
int main(void)
{  static int i=10;
    return 0;
}

//Check data
```

```
$size size1
   text      data      bss      dec      hex    filename
   1099       544        8     1651      673    size1
```

# Layout of a Program Image

Check with ls –l

**Static variables can make a program unsafe for threaded execution, so they are not thread safe.**

**Version 1:** `largearrayinit.c`

```
int myarray[50000] = {1, 2, 3, 4};
int main(void) {
myarray[0] = 3;
return 0;
}
```

**Version 2:** `largearray.c`

```
int myarray[50000];
int main(void) {
myarray[0] = 3;
return 0;
}
```

```
MN:~$gcc -o size1 size1.c
MN:~$ls -l size1
-rwxrwxr-x 1 mamata mamata 208624 Jan  8 09:22 size1
```

```
MN:~$ gcc -o size1 size1.c
MN:~$ ls -l size1
-rwxrwxr-x 1 mamata mamata 8608 Jan  8 09:25 size1
```

# Process Termination

Check with ls –l

**Static variables can make a program unsafe for threaded execution, so they are not thread safe.**

**Version 1:** `largearrayinit.c`

```
int myarray[50000] = {1, 2, 3, 4};
int main(void) {
myarray[0] = 3;
return 0;
}
```

**Version 2:** `largearray.c`

```
int myarray[50000];
int main(void) {
myarray[0] = 3;
return 0;
}
```

```
MN:~$gcc -o size1 size1.c
MN:~$ls -l size1
-rwxrwxr-x 1 mamata mamata 208624 Jan  8 09:22 size1
```

```
MN:~$ gcc -o size1 size1.c
MN:~$ ls -l size1
-rwxrwxr-x 1 mamata mamata 8608 Jan  8 09:25 size1
```