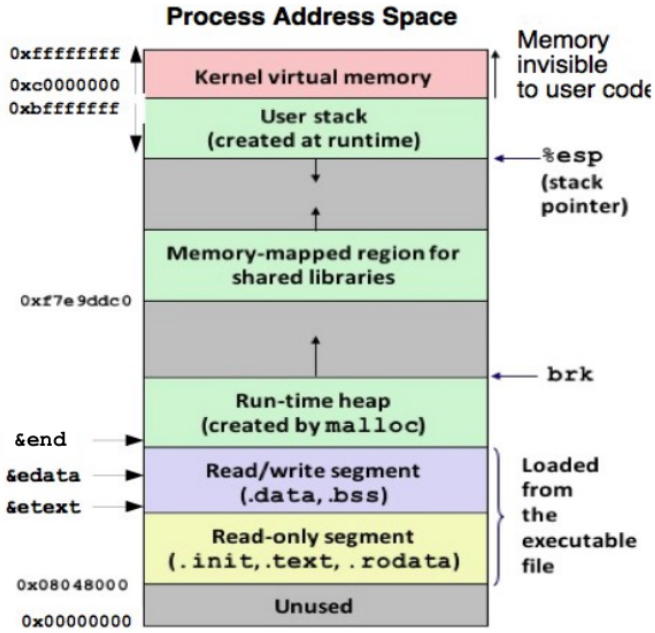
Stack & Heap

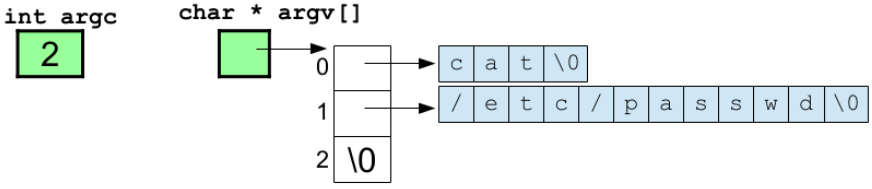

Process Address Space
0xffffffff
0xc0000000
0xbfffffff
0xf7e9ddc0
0x08048000
0x00000000
Kernel virtual memory
User stack
(created at runtime)
Memory-mapped region for
shared libraries
Run-time heap
(created by malloc)
Read/write segment
(.data, .bss)
Read-only segment
(.init, .text, .rodata)
Unused
&end
&edata
&etext
%esp
(stack
pointer)
brk
Memory
invisible
to user code
Loaded
from
the
executable
file

Command Line Arguments
&
Environment Variables

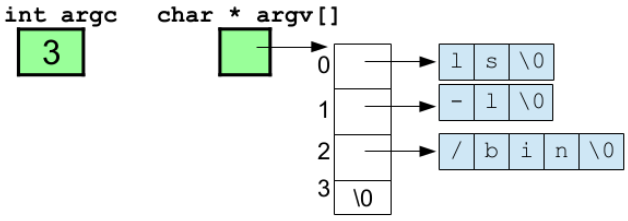
Command Line Arguments

```
int main(int argc, char *argv[]){
   printf("No of arguments passed are: %d\n",argc);
   printf("Parameters are:\n");
   for(int i = 0; argv[i] != NULL ; i++)
      printf("argv[%d]:%s \n", i, argv[i]);
   return 0;
}
```

```
$ cat /etc/passwd
```



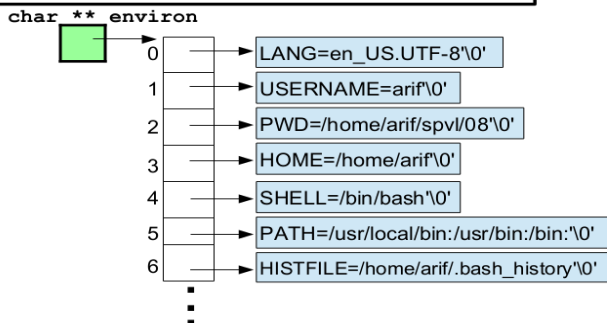

```
$ ls -l /bin
```



Accessing Environment Variables

```
int extern char **environ;
int main(){
   printf("\n Environment variable passed are:\n");
   for (int i = 0; environ[i] != NULL ; i++)
      printf("environ[%d]:%s\n", i, environ[i]);
   return 0;
}
```

```
$ ./myexe
```



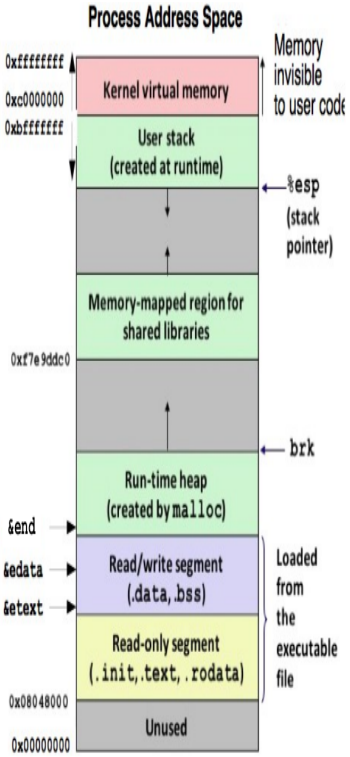
Modifying Environment Variables

The way we can change environment variables on the shell, we can also change them form within a C program, as well as can create a new environment variable using library functions like:

```
char *getenv(const  char *name)
int putenv(char *string)
int setenv(const char *name,const char *val,int overwrite)
int unsetenv(const char *name)
int clearenv()
```

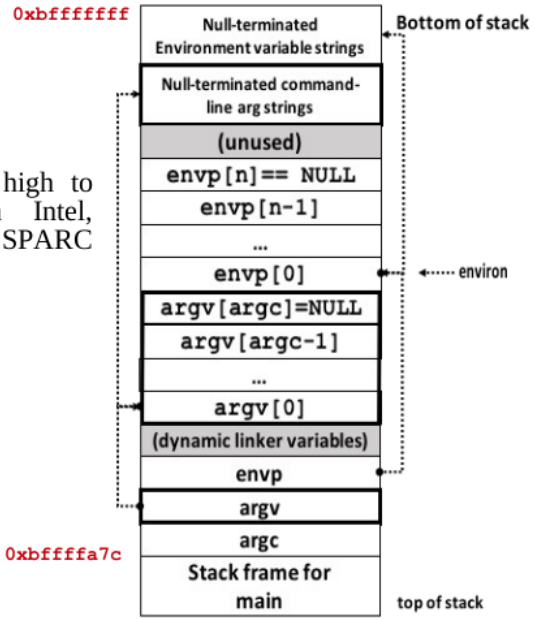
Reasons to modify the environment variables:

- To build a suitable environment for a process to run
- A form of IPC, since a child gets a copy of its parent's environment variables at the time it is created

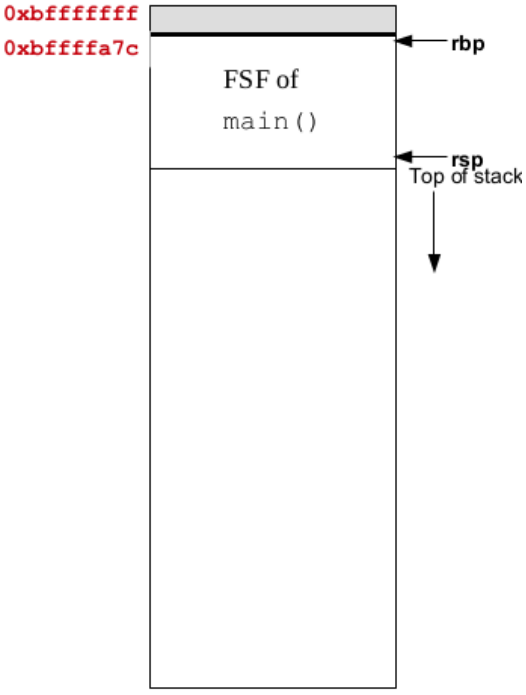


Layout of Process Stack

Stack grows from high to low addresses in Intel, MIPS, Motorola, & SPARC architectures.



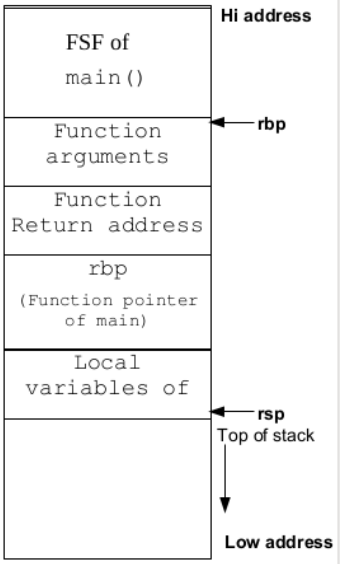
- Function Stack Frames
- Used to store **local variables**
- For passing **arguments** to the functions
- For storing the **return address**
- For storing the **base pointer**
- Stack grows downward
- Frame pointer (rbp)
- Stack top (rsp)
- Reclaiming stack memory



Stack Growing and Shrinking

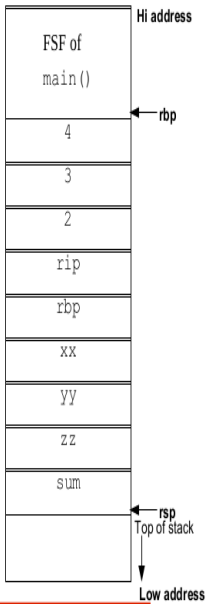
Suppose the `main()` calls another function `foo()` , the sequence of steps for creation of FSF of `foo()`:

- Arguments are pushed on the stack, in reverse order
- Contents of `rip` (return address) is also pushed on the stack
- The contents of `rbp` containing starting address of `main` stack frame is saved on stack for later use, and `rbp` is moved to where `rsp` is pointing to create new stack frame pointer of function `foo()`
- Space created for local variables by moving `rsp` down or to lower address



Stack Growing and Shrinking (cont...)

```
int main(...){
   ...
   return foo(2,3,4);
}
void foo(int a,int b, int c){
   int xx = a+2;
   int yy = b+2;
   int zz = c+2;
   int sum = xx + yy + zz;
   return sum; }
```

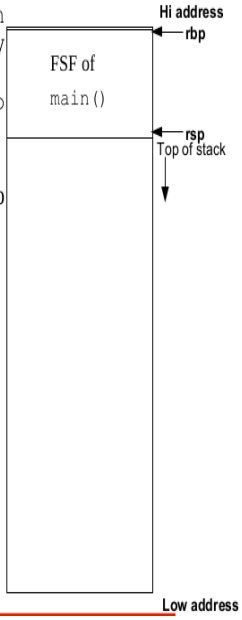


Stack Growing and Shrinking (cont...)

Finally when the `foo()` function executes `return` statement, memory on the stack is automatically, and very efficiently reclaimed:

- The saved base pointer is popped and placed in `rbp` which moves to the starting address of `main` FSF
- The saved return address is popped and placed in `rip`
- The stack is shrinked by moving the `rsp` further up to where `rbp` is pointing

```
int main(...){
   ...
   return foo(2,3,4);
}
void foo(int a,int b, int c){
   int xx = a+2;
   int yy = b+2;
   int zz = c+2;
   int sum = xx + yy + zz;
   return sum; }
```



Heap Allocators

Allocators come in two basic styles. Both styles require the application to explicitly allocate blocks. They differ about which entity is responsible for freeing allocated blocks

- **Explicit allocators** require the application to explicitly free any allocated blocks. For example, the C standard library provides an explicit allocator called the `malloc` package. C programs allocate a block by calling the `malloc` function and free a block by calling the `free` function. In C++, we normally use the `new` and `delete` operators
- **Implicit allocators** require the allocator to detect when an allocated block is no longer being used by the program and then free the block. Implicit allocators are also known as garbage collectors, and the process of automatically freeing unused allocated blocks is known as garbage collection. For example, higher-level languages such as Lisp, ML and Java rely on garbage collection to free allocated blocks

The `malloc` family in C

```
void *malloc (size_t size);
void*calloc(size_t noOfObjects, size_t size);
void *realloc (void* ptr, size_t newsize );
void free ( void* ptr );
```

- **`malloc()`** allocates size bytes from the heap and returns a pointer to the start of the newly allocated block of memory. On failure returns NULL and sets **errno** to indicate error
- **`calloc()`** allocates space for specific number of objects, each of specified size. Returns a pointer to the start of the newly allocated block of memory. Unlike **`malloc()`**, **`calloc()`** initializes the allocated memory to zero. On failure returns NULL and sets **errno** to indicate error
- **`realloc()`** is used to resize a block of memory previously allocated by one of the functions in **`malloc()`** package. Ptr argument is the pointer to the block of memory that is to be resized. On success **`realloc()`** returns a pointer to the location of the resized block, which may be different from its location before the call. On failure, returns NULL and leaves the previous block pointed to by pointer untouched.
- **`free()`** deallocates the block of memory pointed to by its pointer argument, which should be an address previously returned by functions of **`malloc`** package

Illustration: 1D Array on Heap

```
char*str=(char*)malloc(sizeof(char)*10);
   …
   …
   …
free(str);
```

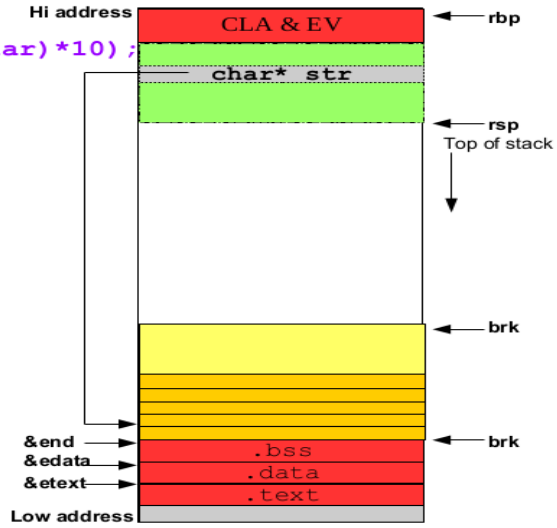


Illustration: 2D Array on Heap

```
int i, int rows = 4, cols = 12;
char ** names = (char**)malloc(sizeof(char*) * rows);
for(i = 0; i < rows; i++)
   names[i] = (char*)malloc(sizeof(char) * cols);
…
…
…
for(i = 0; i < rows; i++)
   free(names[i]);
free(names);
```

System Call: brk()

```
int brk(void* end_data_segment);
```

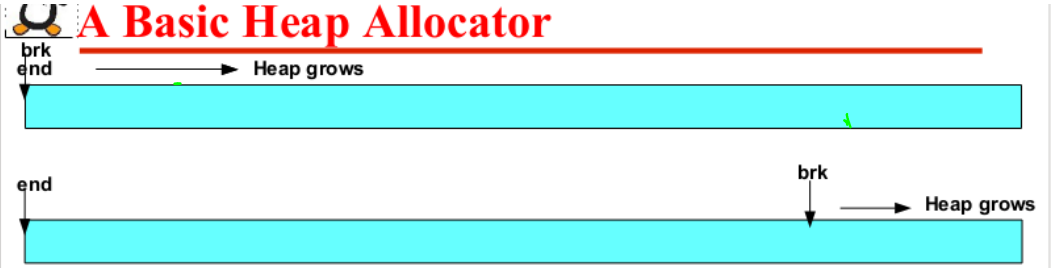
- Resizing the heap is actually telling the kernel to adjust the process's **program break**, which lies initially just above the end of the uninitialized data segment (i.e `end` variable)
- **`brk()`** is a system call that sets the **program break** to location specified by **end_data_segment**. Since virtual memory is allocated in pages, this request is rounded up to the page boundary. Any attempt to lower the program break than **`end`** results in segmentation fault
- The upper limit to which the program break can be set depends on range of factors like:
  - Process resource limit for size of data segment
  - Location of memory mappings, shared memory segment and shared libraries

Library Call: sbrk()

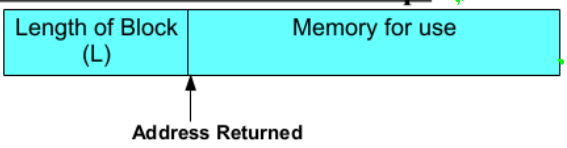
```
void *sbrk (intptr_t increment);
```

- **`sbrk()`** is a C library wrapper implemented on top of **`brk()`**. It increments the program break by **increment** bytes
- On success, **`sbrk()`** returns a pointer to the end of the heap before **`sbrk()`** was called, i.e., a pointer to the start of new area
- So calling **`sbrk(0)`** returns the current setting of the program break without changing it
- On failure -1 is returned with **`errno ENOMEM`**

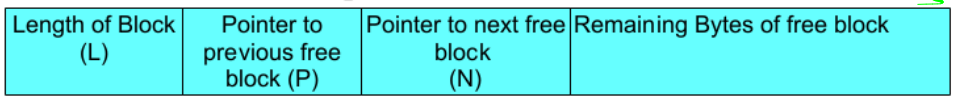
A Basic Heap Allocator



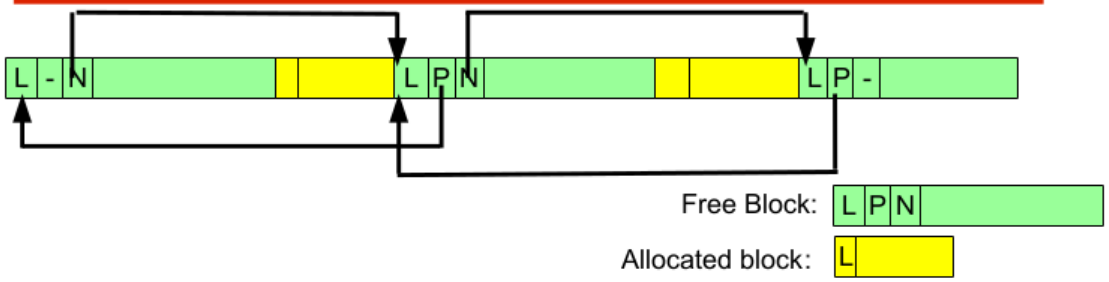
Structure of Allocated Block on Heap:



Structure of Free Block on Heap:

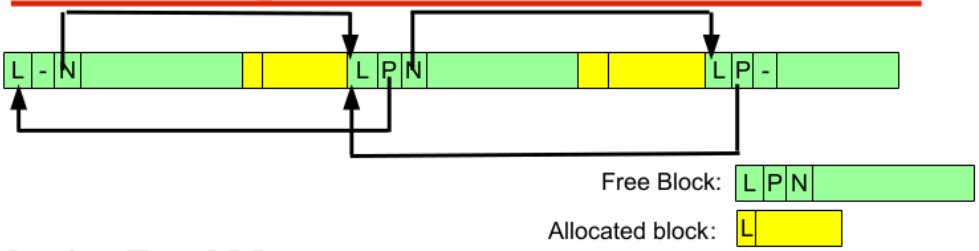


A Basic Heap Allocator (cont...)



- When a program calls **`malloc`** the allocator scans the link list of free memory blocks as per one of the contiguous memory allocation algorithm's (**first fit, best fit, next fit**), assigns the block and update the data structures
- If no block on the free list is large enough then **`malloc()`** calls **`sbrk()`** to allocate more memory
- To reduce the number of calls to **`sbrk()`**, **`malloc()`** do not allocate exact number of bytes required rather increase the **`program break`** in large units (some multiples of virtual memory page size) and putting the excess memory onto the free list

A Basic Heap Allocator (cont...)



Coalescing Freed Memory

- When you call `free()`, you put a chunk of memory back on the free list
- There may be times when the chunk immediately before it in memory, and/or the chunk immediately after it in memory are also free
- If so, it make sense to try to merge the free chunks into one free chunk, rather than having three contiguous free chunks on the free list
- This is called "**coalescing**" free chunks

Why not use `brk()` and `sbrk()`

C program use **`malloc`** family of functions to allocate & deallocate memory on the heap instead of **`brk()` & `sbrk()`**, because:

- **`malloc`** functions are standardized as part of C language
- **`malloc`** functions are easier to use in threaded programs
- **`malloc`** functions provide a simple interface that allows memory to be allocated in small units
- **`malloc`** functions allow us to deallocate blocks of memory

Why **free()** doesn't lower the program break? rather adds the block of memory to a lists of free blocks to be used by future calls to **`malloc()`**. This is done for following **reasons**:

- Block of memory being freed is somewhere in the middle of the heap, rather than at the end, so lowering the program break is not possible
- It minimizes the numbers of **sbrk()** calls that the program must perform

Common programming errors while using heap:

1. Reading/writing freed memory areas
2. Reading/writing memory addresses before or after the allocated memory using faulty pointer arithmetic
3. Freeing the same piece of allocated memory more than once
4. Freeing heap memory by a pointer, that wasn't obtained by a call to **`malloc`** package
5. Memory leaks, i.e., not freeing memory and keep just allocating it