



Video Lecture # 09

Process Stack

Behind the Curtain

Course: SYSTEM PROGRAMMING

Instructor: Arif Butt

Punjab University College of Information Technology (PUCIT)
University of the Punjab



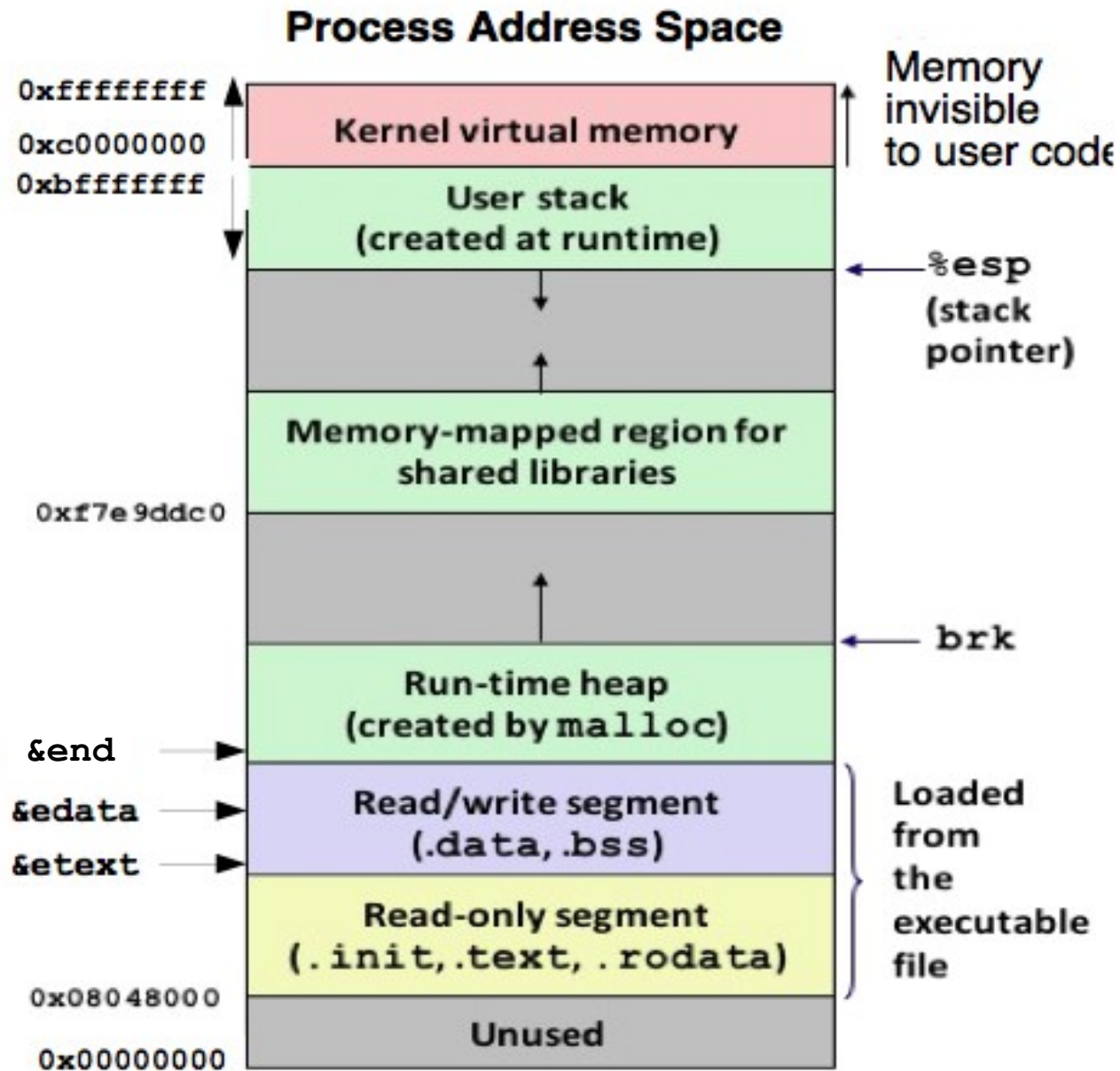
Agenda

- Review of a Process Logical Address Space
- Command Line Arguments
- Environment Variables
- Layout of a Process User Stack
- Growing and Shrinking of Stack
- Stack Buffer Overflow
- Non-Local goto using `longjmp()`





Process Logical Address Space



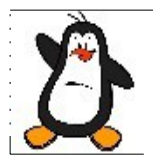


Example

`logicaladdresses.c`



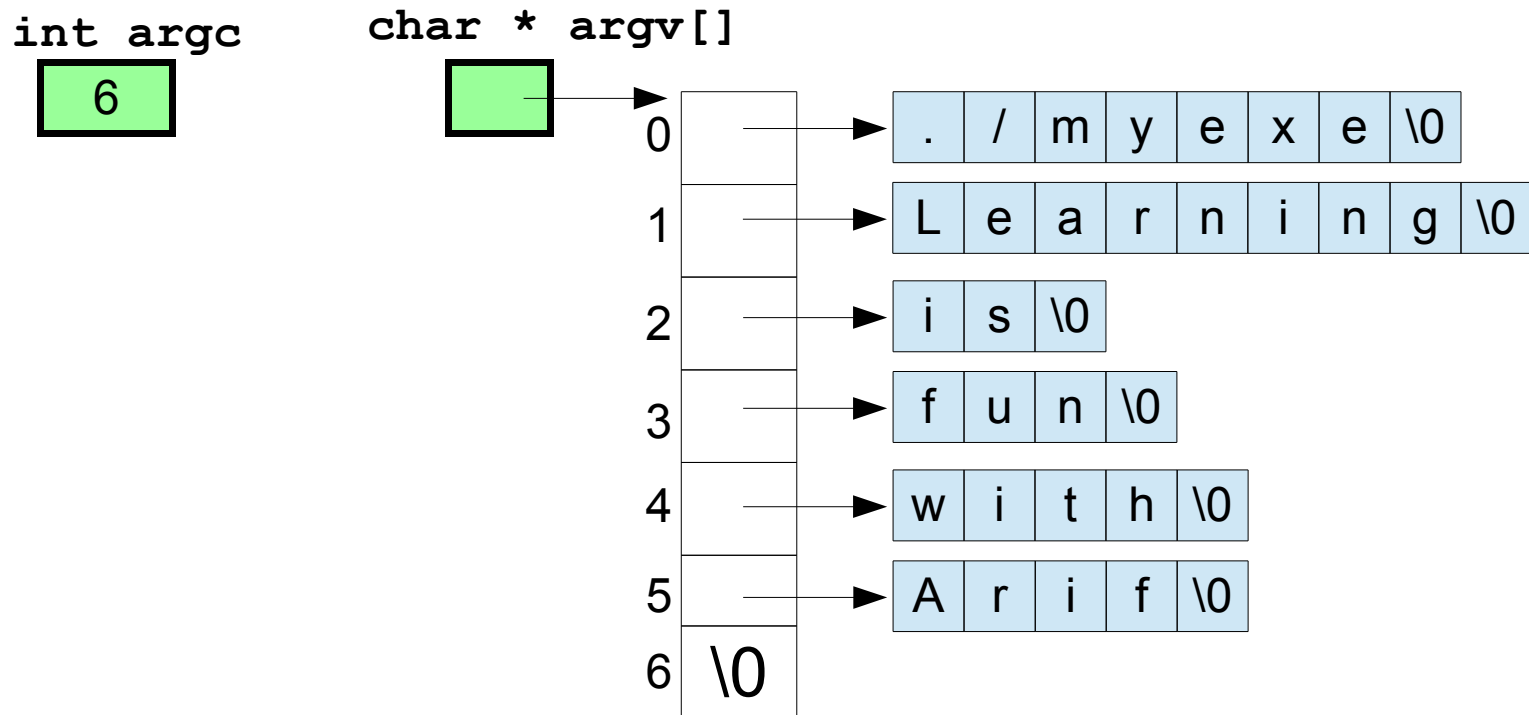
Command Line Arguments & Environment Variables

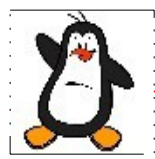


Command Line Arguments

```
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;
}
```

\$./myexe Learning is fun with Arif





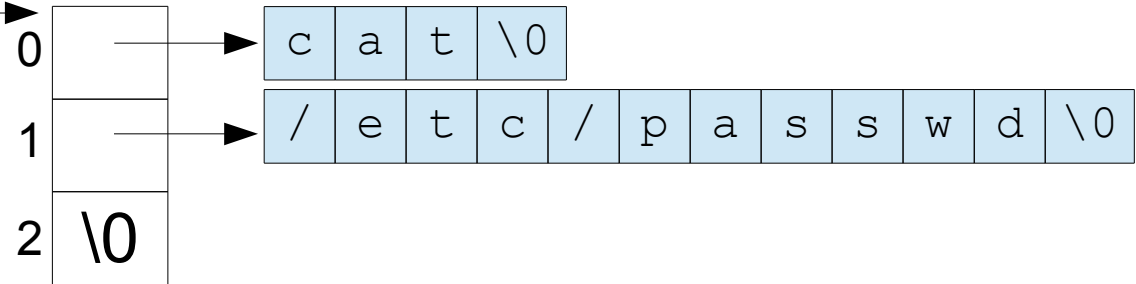
Use of Command Line Arguments

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

int argc

2

char * argv[]

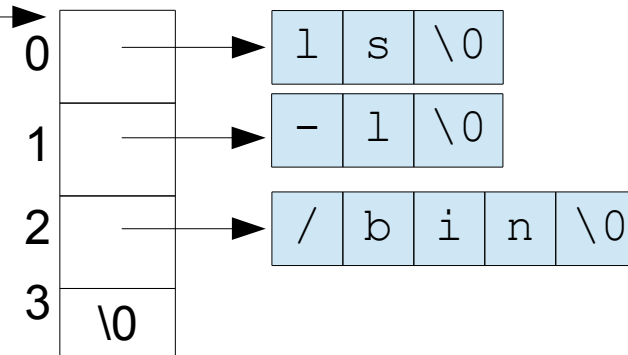


```
$ ls -l /bin
```

int argc

3

char * argv[]





Example

`cmdarg_ex2.c`

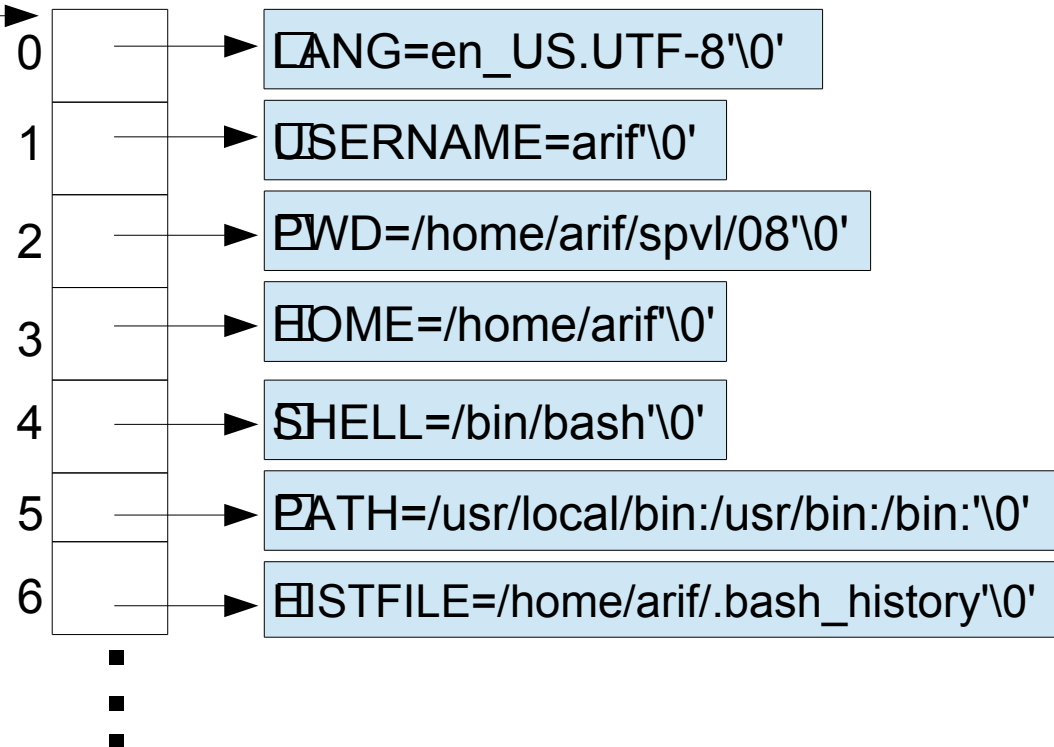
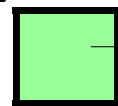


Accessing Environment Variables

```
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

char ** environ





Modifying Environment Variables

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

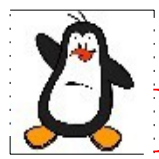
```
char *getenv(const char *name)
int putenv(char *string)
int setenv(const char *name, const char *val, int overwrite)
int unsetenv(const char *name)
int clearenv()
```

Reasons to modify the environment variables:

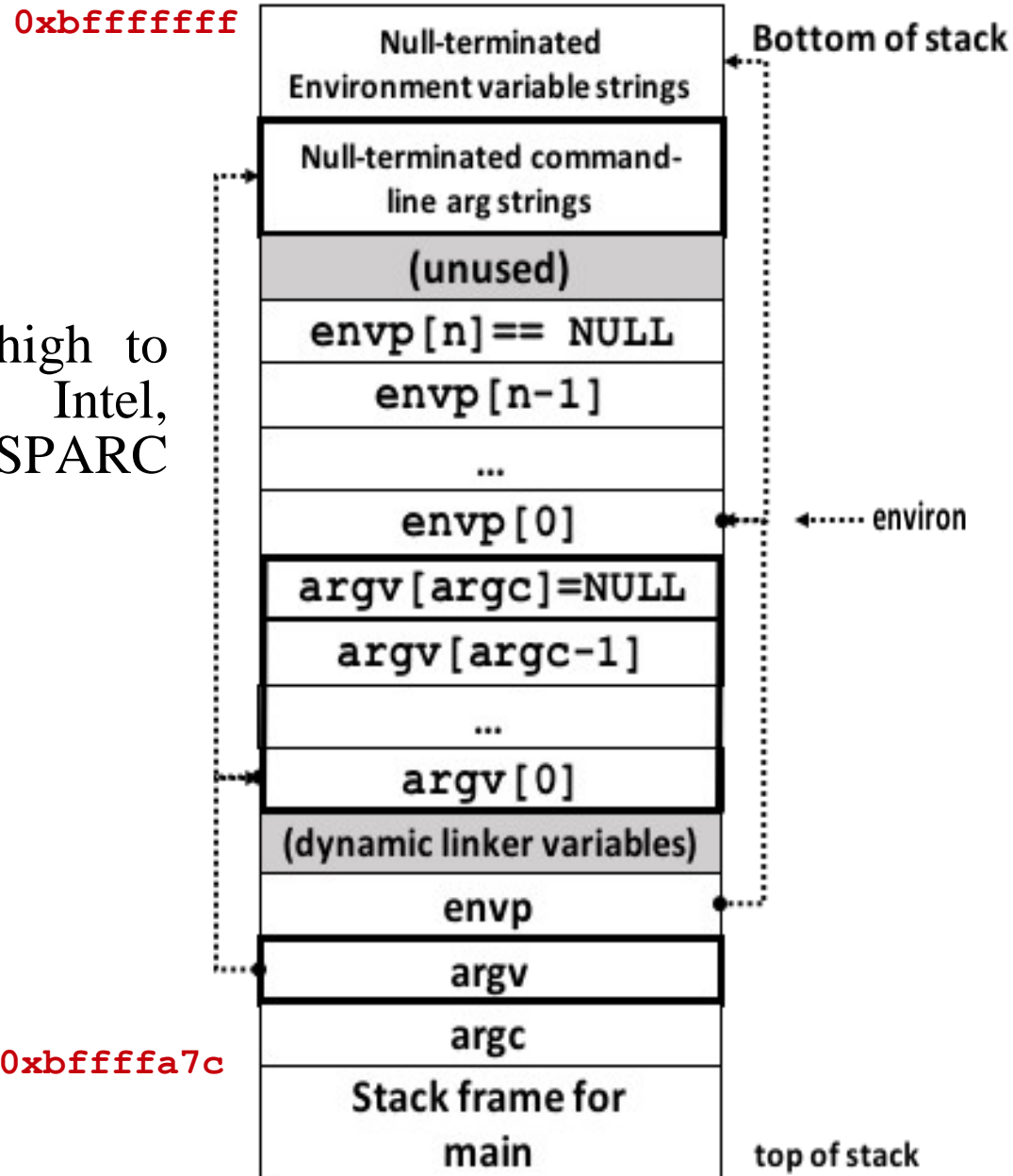
- To build a suitable environment for a process to run
- A form of IPC, since a child gets a copy of its parent's environment variables at the time it is created



Layout of a Process Stack



Layout of Process Stack



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

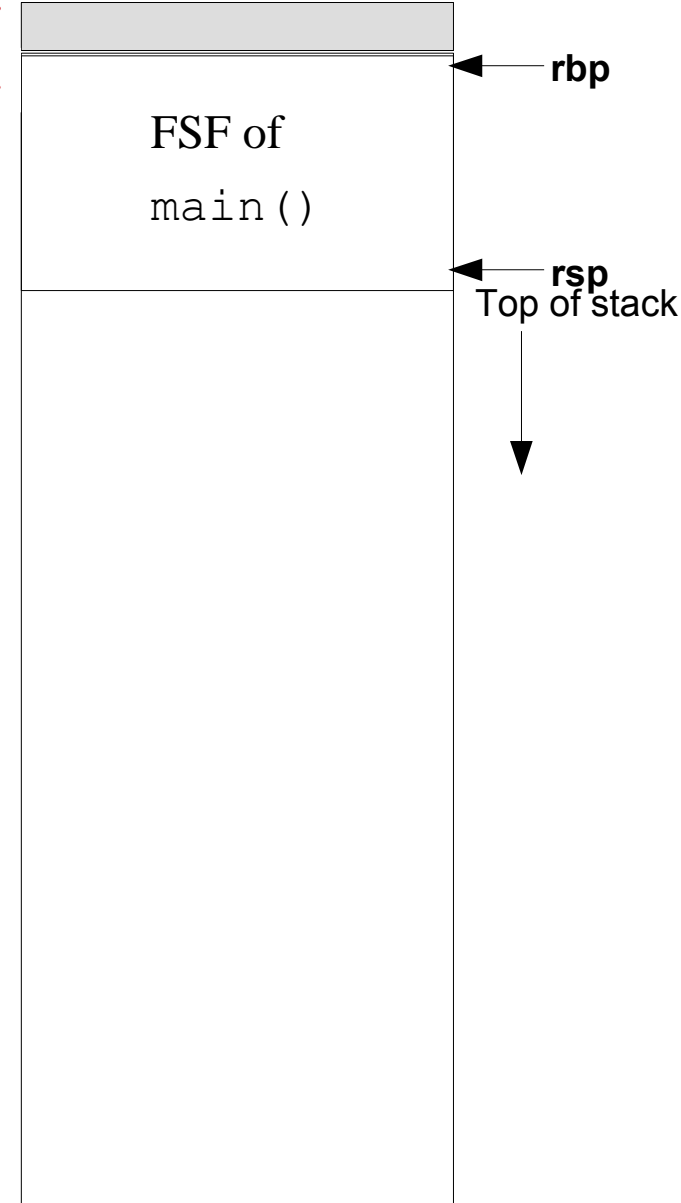


Layout of Process Stack (cont...)

- 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

0xbfffffff

0xbfffa7c



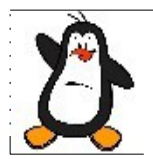


Function Calling Convention

Function calling convention means:

- How the function arguments are passed?
 - Via Stack
 - Via Registers
 - Mix of above two
- Order in which function arguments are passed?
 - Right to left
 - Left to right
- Who is responsible for creating the FSF?
 - Callee
 - Caller
- Who is responsible for unwinding the stack?
 - Callee
 - Caller

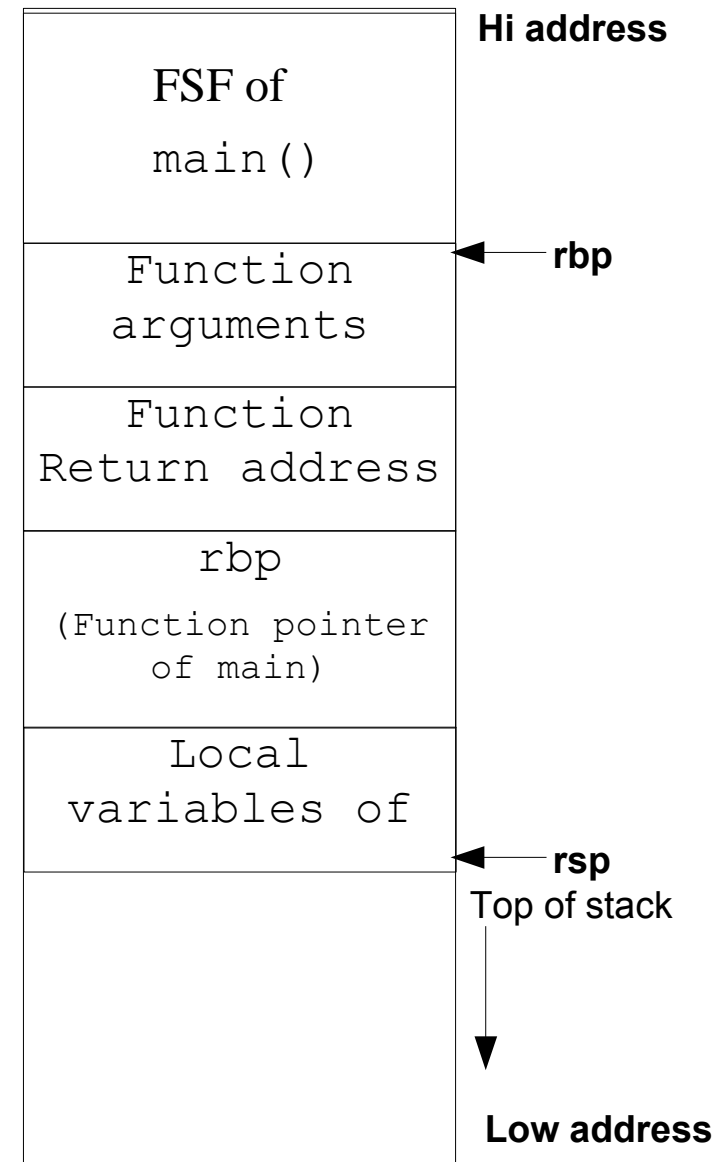
Different calling conventions used by C language are Cdecl, stdcall, and fastcall

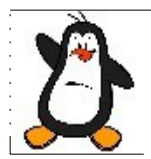


Stack Growing and Shrinking

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



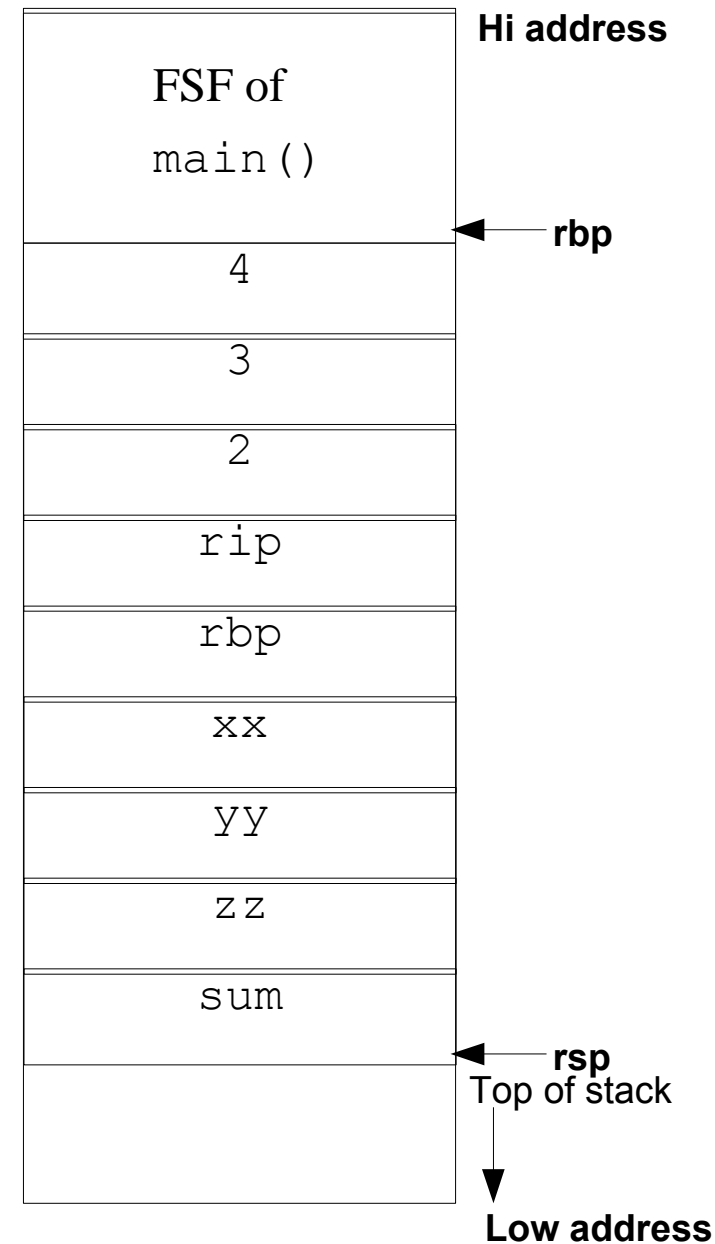


Stack Growing and Shrinking (cont...)

```
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; }

```



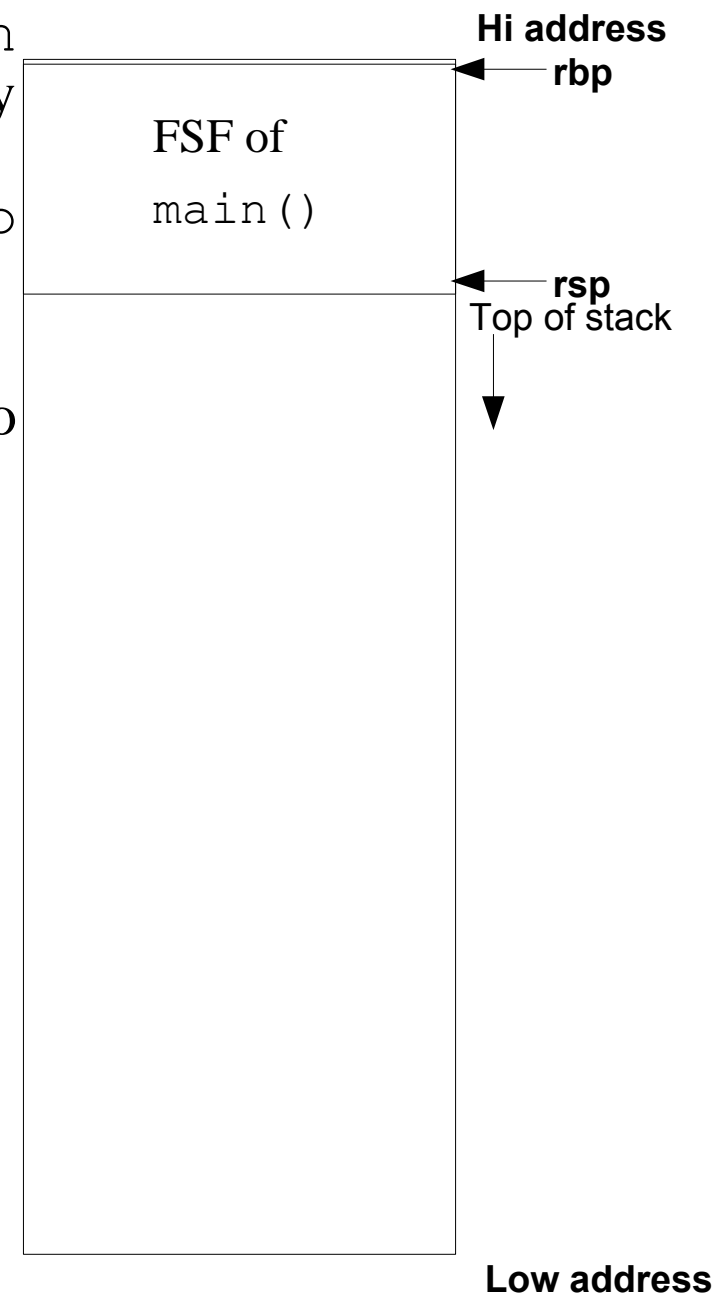


Stack Growing and Shrinking (cont...)

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 shrunk 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; }
```



Low address



Overview of Buffer Overflow

A buffer overflow is a bug in a program, which occurs when more data is written to a block of memory than it can handle



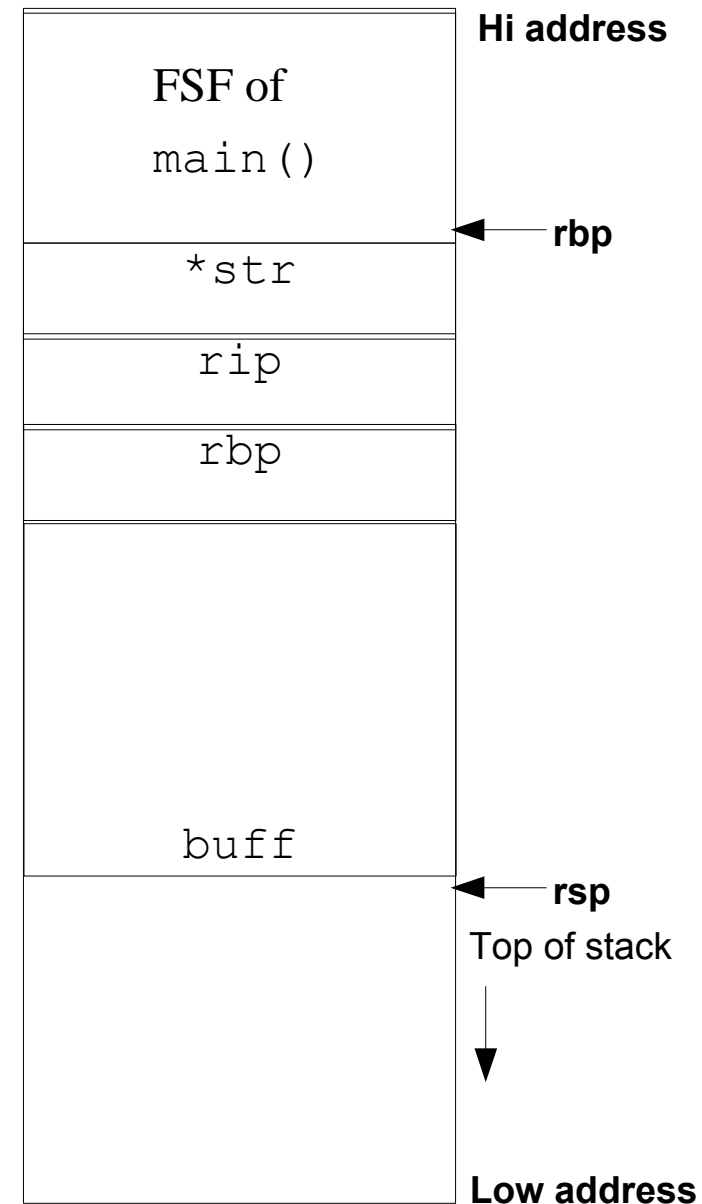
Stack Growing and Shrinking (cont...)

```

int main(...) {
    ...
    display(argv[1]);
    ...
}

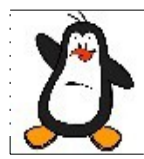
void display(char* str) {
    char buff[10];
    strcpy(buff, str);
    printf("Data is:%s\n", buff);
}

```



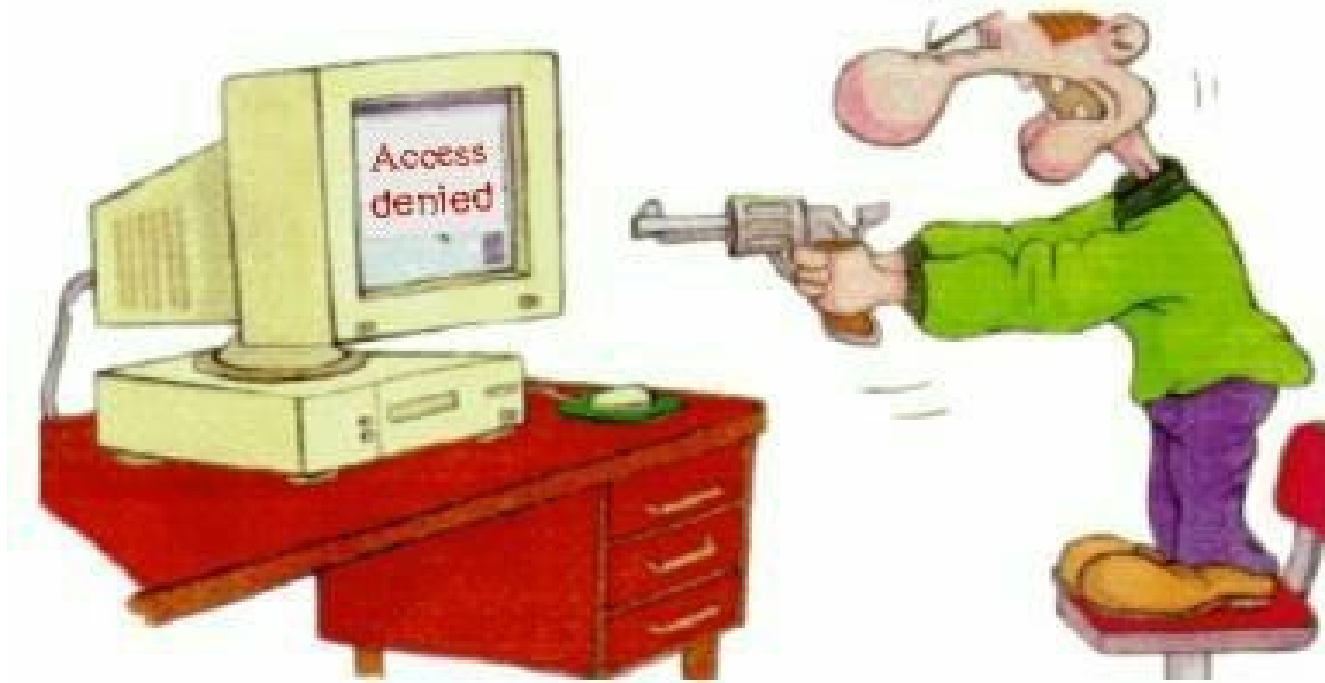


Non-Local goto



Things To Do

O.k., and now you'll do exactly what I'm telling you !



If you have problems visit me in counseling hours. . . .
