UNIX PROCESSES

INTRODUCTION

A Process is a program under execution in a UNIX or POSIX system.
A **process**, in simple terms, is an instance of a running program. The operating system tracks **processes** through a five-digit ID number known as the pid or the **process** ID.

A program/command when executed, a special instance is provided by the system to the process. This instance consists of all the services/resources that may be utilized by the process under execution.

Whenever a command is issued in unix/linux, it creates/starts a new process. For example, pwd when issued which is used to list the current directory location the user is in, a process starts.

Through a 5 digit ID number unix/linux keeps account of the processes, this number is call process id or pid. Each process in the system has a unique pid.

Used up pid’s can be used in again for a newer process since all the possible combinations are used.

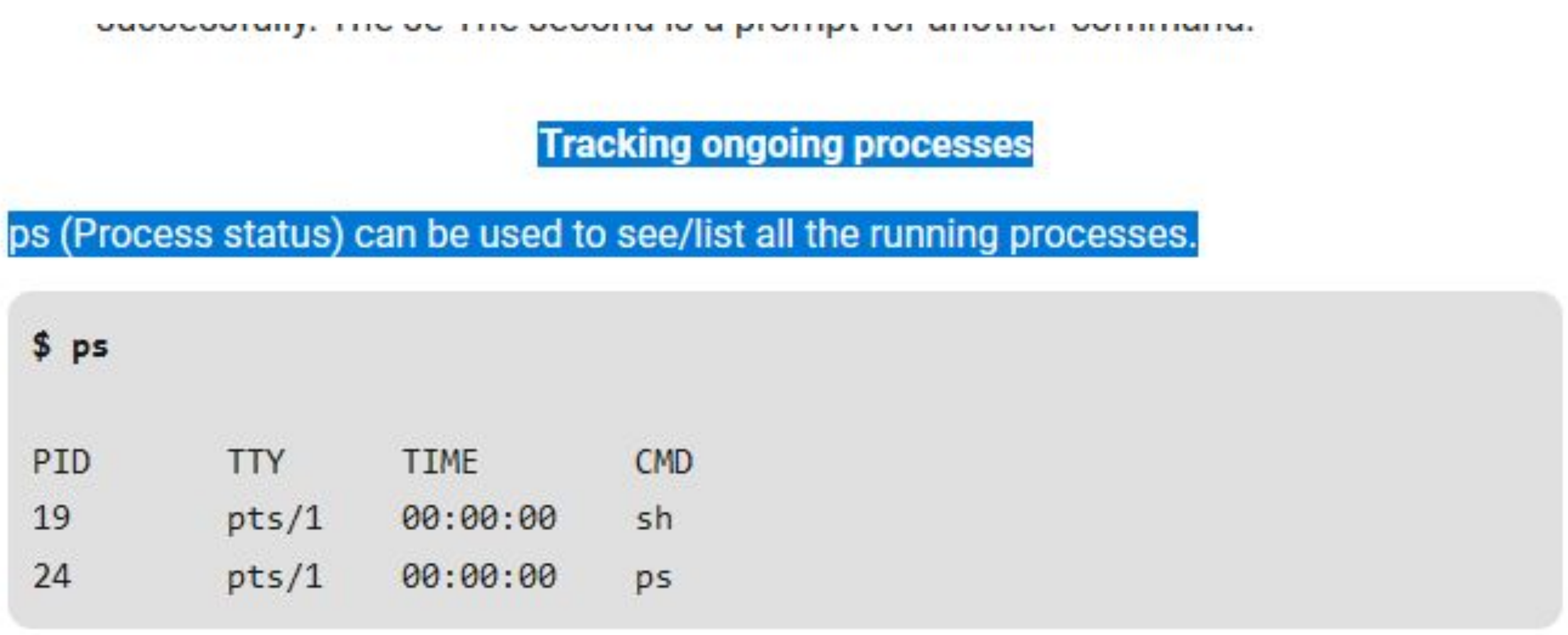
At any point of time, no two processes with the same pid exist in the system because it is the pid that Unix uses to track each process.

A process can be run as:
Foreground Process : Every process when started runs in foreground by default, receives input from the keyboard and sends output to the screen.
When issuing pwd command

```
$ ls pwd
Output:
$ /home/geeksforgeeks/root
```

When a command/process is running in the foreground and is taking a lot of time, no other processes can be run or started because the prompt would not be available until the program finishes processing and comes out.

Tracking ongoing processes
ps (Process status) can be used to see/list all the running processes.



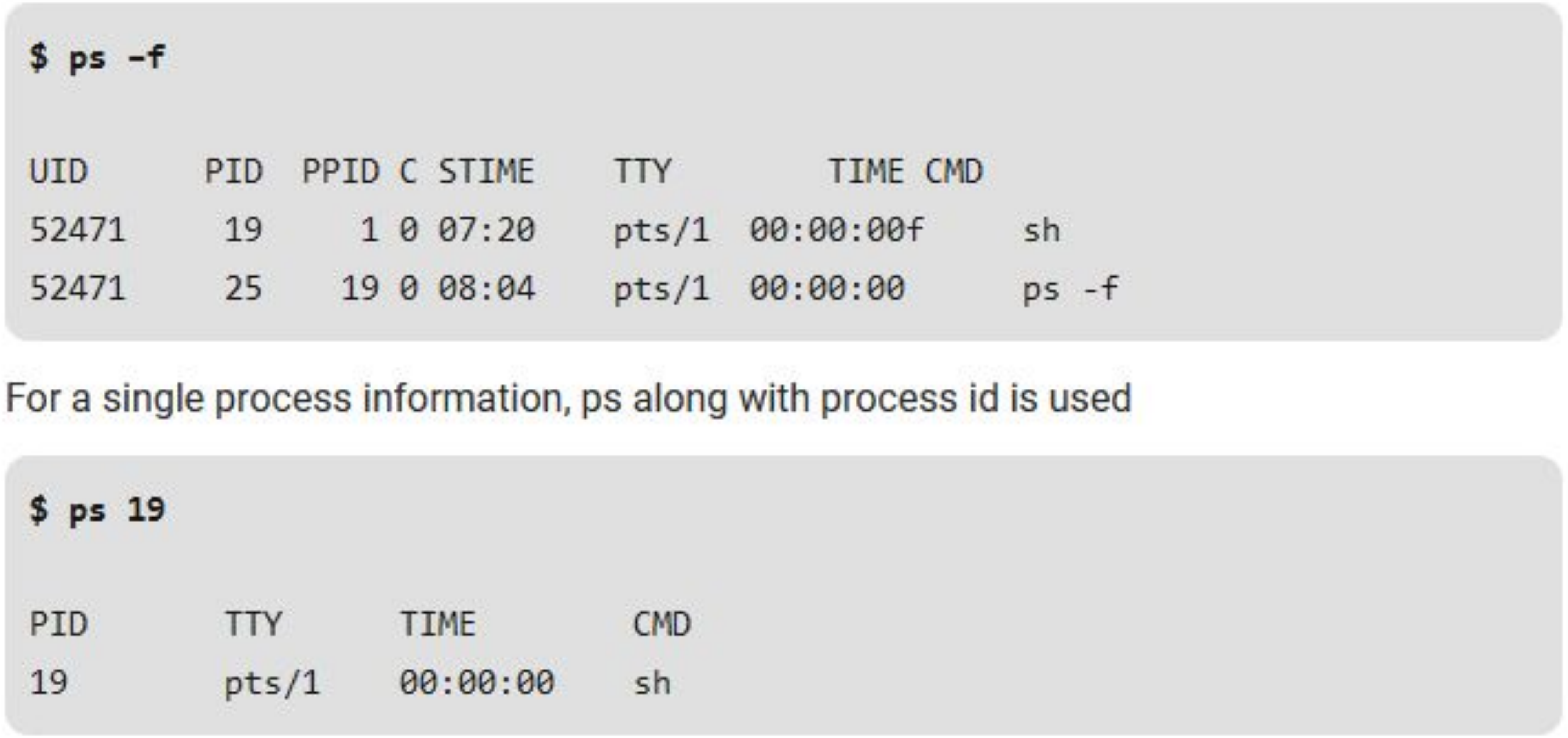
successfully. The se The second is a prompt for another command.

Tracking ongoing processes

ps (Process status) can be used to see/list all the running processes.

```
$ ps

PID        TTY      TIME        CMD
19         pts/1    00:00:00    sh
24         pts/1    00:00:00    ps
```

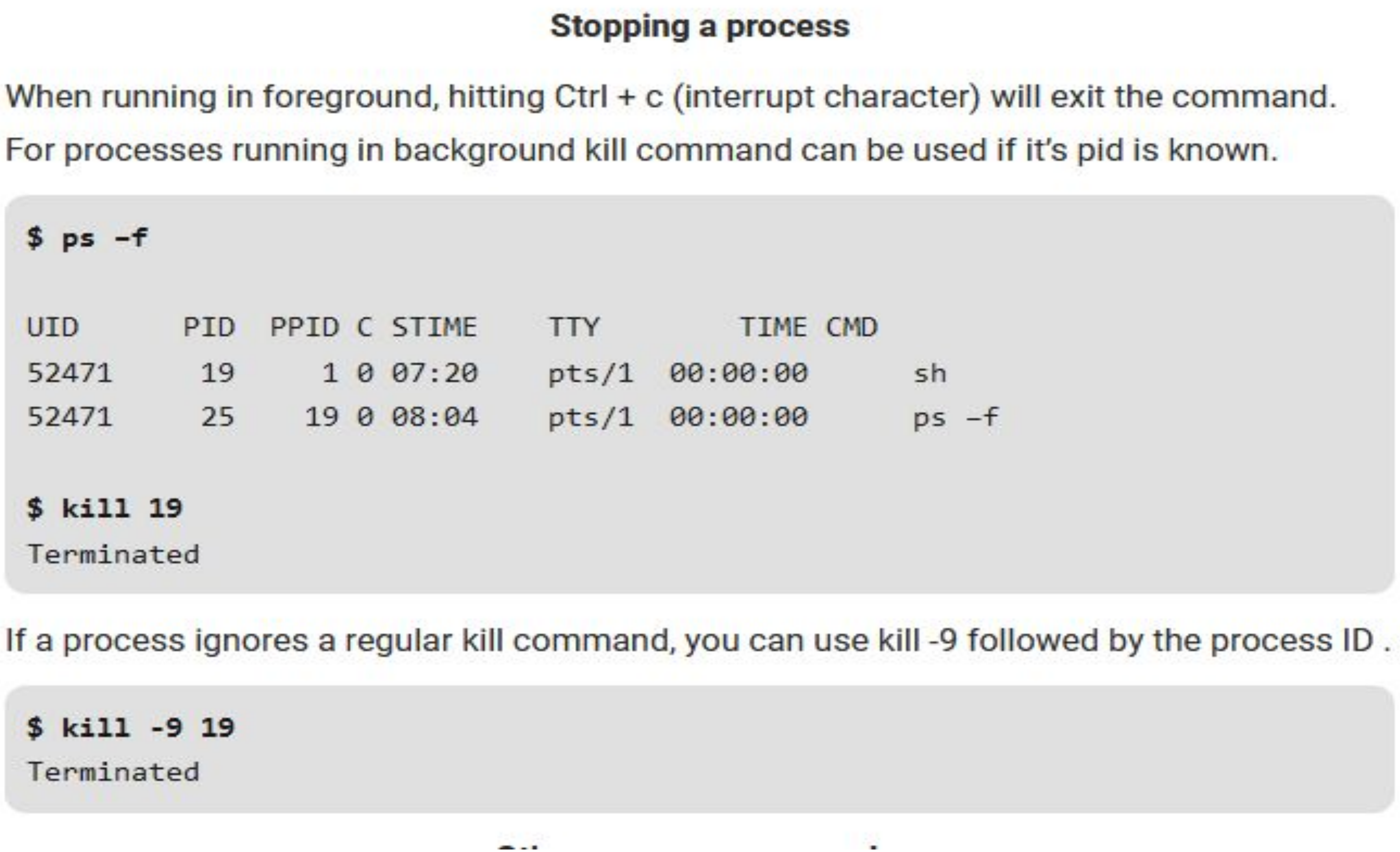
For More information



```
$ ps -f

UID        PID   PPID C STIME     TTY          TIME CMD
52471       19      1 0 07:20     pts/1   00:00:00f     sh
52471       25     19 0 08:04     pts/1   00:00:00      ps -f
```

For a single process information, ps along with process id is used

```
$ ps 19

PID         TTY       TIME         CMD
19          pts/1     00:00:00     sh
```



Stopping a process

When running in foreground, hitting Ctrl + c (interrupt character) will exit the command.

For processes running in background kill command can be used if it's pid is known.

```
$ ps -f

UID        PID  PPID C STIME    TTY         TIME CMD
52471       19     1 0 07:20    pts/1  00:00:00      sh
52471       25    19 0 08:04    pts/1  00:00:00      ps -f

$ kill 19
Terminated
```

If a process ignores a regular kill command, you can use kill -9 followed by the process ID .

```
$ kill -9 19
Terminated
```

Types of Processes

1) Parent and Child process : The 2nd and 3rd column of the ps –f command shows process id and parent’s process id number. For each user process there’s a parent process in the system, with most of the commands having shell as their parent.

2) Zombie and Orphan process : After completing its execution a child process is terminated or killed and SIGCHLD updates the parent process about the termination and thus can continue the task assigned to it. But at times when the parent process is killed before the termination of the child process, the child processes becomes orphan processes, with the parent of all processes “init” process, becomes their new ppid.

A process which is killed but still shows its entry in the process status or the process table is called a zombie process, they are dead and are not used.

main FUNCTION

A C program starts execution with a function called main.

The prototype for the main function is
int main(int argc, char *argv[]);

where argc is the number of command-line arguments, and argv is an array of pointers to the arguments.

- When a C program is executed by the kernel by one of the exec functions, a special start-up routine (creates heap/malloc work) is called before the main function is called.
- The executable program file specifies this routine as the starting address for the program.
- This is set up by the link editor when it is invoked by the C compiler.
- This start-up routine takes values from the kernel, the command-line arguments and the environment and sets things up so that the main function is called.

PROCESS TERMINATION

There are eight ways for a process to terminate.

Normal termination occurs in five ways:

1. Return from main
2. Calling exit
3. Calling _exit or _Exit
4. Return of the last thread from its start routine
5. Calling pthread_exit from the last thread

Abnormal termination occurs in three ways:

1. Calling abort
2. Receipt of a signal
3. Response of the last thread to a cancellation request

The exit() function performs some cleaning before terminating the program. It clears the connection termination, buffer flushes etc. This _Exit() function does not clean anything.

- main function returns integer value by default. if zero is returned as exit status to indicate to Operating system that the program worked successfully without any compile time or run time errors. if a non zero value(1 mostly) is returned to indicate some error happened in program execution.
- exit()
- The function exit() is used to terminate the calling function immediately without executing further processes. As exit() function calls, it terminates processes.
- #include <stdio.h>
- #include <stdlib.h>
- int main()
- {
- int x = 10;
- printf("The value of x : %d\n", x);
- exit(0);
- printf("Calling of exit()");
- return 0;}
- **In the above program, a variable 'x' is initialized with a value. The value of variable is printed and exit() function is called. As exit() is called, it exits the execution immediately and it does not print the statement in the printf(). The calling of exit() is as follows −**

Exit Functions

Three functions terminate a program normally:
_exit and _Exit, which return to the kernel immediately, and
exit, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>
void   exit(int status);
void   _Exit(int status);

#include <unistd.h>
void    _exit(int status);
```

All three exit functions expect a single integer argument, called the exit status.
Returning an integer value from the main function is equivalent to calling exit with the same value.
Thus **exit(0)** is the same as **return(0)** from the main function.

UNIT 4 UNIX PROCESSES

Exit Functions

Three functions terminate a program normally:
_exit and _Exit, which return to the kernel immediately, and
exit, which performs certain cleanup processing and then returns to the kernel.

```
#include <stdlib.h>
void   exit(int status);
void   _Exit(int status);

#include <unistd.h>
void     _exit(int status);
```

All three exit functions expect a single integer argument, called the exit status.
Returning an integer value from the main function is equivalent to calling exit with the same value.
Thus **exit(0)** is the same as **return(0)** from the main function.

UNIT 4 UNIX PROCESSES

```
int main()
{
printf(“HKBKCE”);  return(5);
}
```

```
$ cc      demo.c
$ ./a.out  HKBKCE
$ echo $?   // print the exit status
5
```

UNIT 3 UNIX PROCESSES

<u>atexit Function</u>

With ISO C, a process can register up to 32 functions that are automatically called by exit. These are called exit handlers and are registered by calling the atexit function.

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

returns: 0 if OK, nonzero on error

- This declaration says that we pass the address of a function as the argument to atexit.
- When this function is called, it is not passed any arguments and is not expected to return a value.
- The exit function calls these functions in reverse order of their registration.
- Each function is called as many times as it was registered.

UNIT 3 UNIX PROCESSES

Example of exit handlers

```
static void my_exit1(void);
static void my_exit2(void);
 int main(void)
{
if (atexit(my_exit2) != 0)
perror("can't register my_exit2");

if (atexit(my_exit1) != 0)
  perror("can't register my_exit1");

printf("main is done\n");
return(0);
}
```

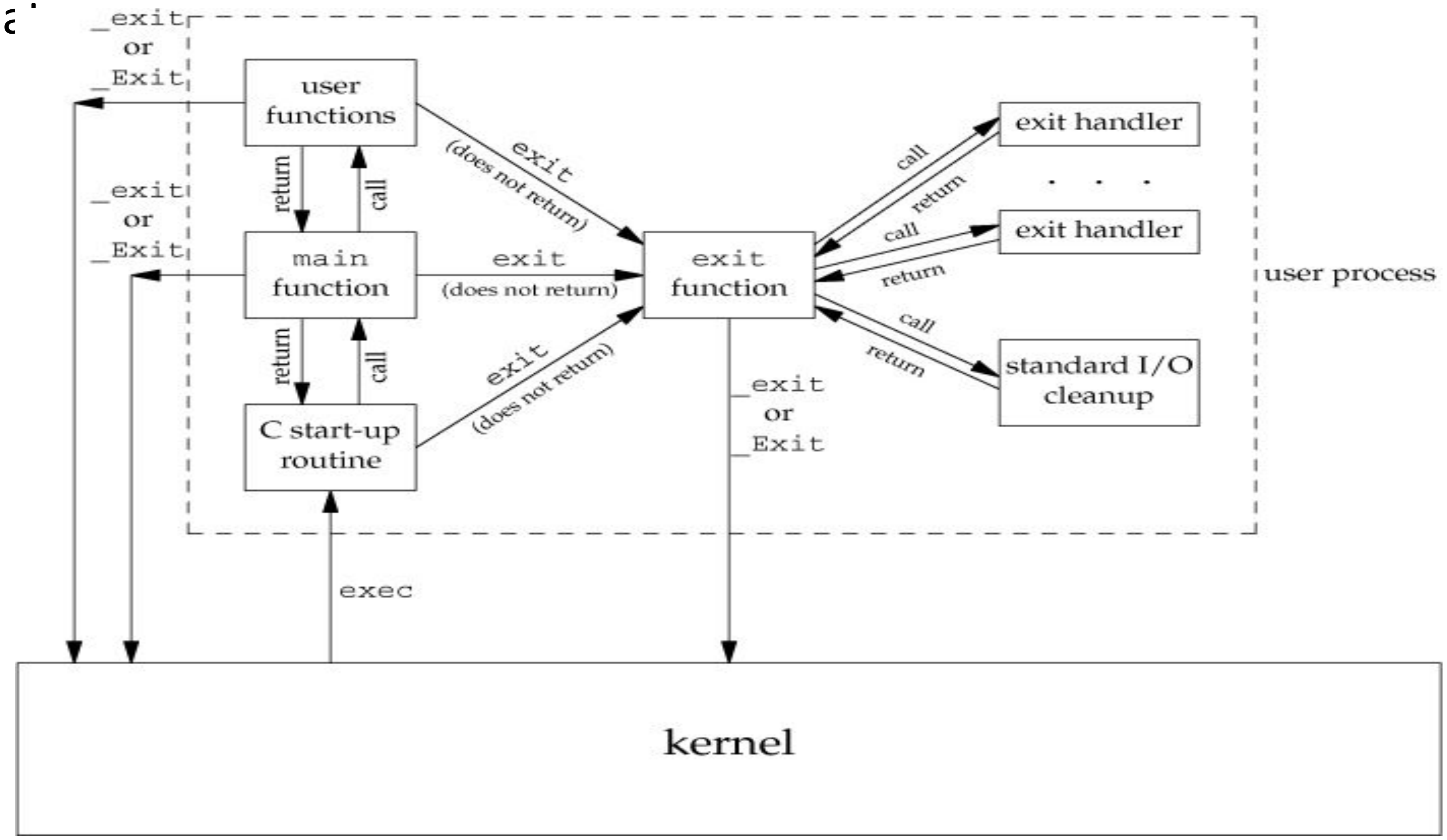
```
static void my_exit1(void)
{
printf("first exit handler\n");
}

static void my_exit2(void)
{
printf("second exit handler\n");
}
```

Output:

```
$ ./a.out
main is done
first exit handler
second exit handler
```

UNIT 3 UNIX PROCESSES

How a C program is started and the various ways it can termina



UNIT 3 UNIX PROCESSES

COMMAND-LINE ARGUMENTS

It is possible to pass some values from the command line to your C programs when they are executed. These values are called command line arguments,

When a program is executed, the process that does the exec can pass command-line arguments to the new program.

Example: Echo all command-line arguments to standard output

```
int main(int argc, char *argv[])
{
int    i;
for (i = 0; i < argc;i++)      /* echo all command-line args */
printf("argv[%d]: %s\n", i, argv[i]);
exit(0);
}
```

Output:

```
$ ./echoarg arg1 TEST foo
 argv[0]: ./echoarg
 argv[1]: arg1
argv[2]: TEST
 argv[3]: foo
```

UNIT 4 UNIX PROCESSES

MEMORY LAYOUT OF A C PROGRAM3 Historically, a C program has been composed of the following pieces:

Text segment:

It is known as code segment it contains the executable instructions or contains the machine code of the computer program.

The machine instructions that the CPU executes.

Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on.

Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Initialized data segment:

usually called simply the data segment, containing variables that are specifically initialized in the program.

For example, the C declaration

int maxcount = 99;

appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

UNIT 3 UNIX PROCESSES

MEMORY LAYOUT OF A C PROGRAM

Uninitialized data segment:

- Often called the "bss" segment, named after an ancient assembler operator that stood for "block started by symbol."
- Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing.

- The C declaration

- **long   sum[1000];**
- appearing outside any function causes this variable to be stored in the uninitialized data segment.

UNIT 3 UNIX PROCESSES

MEMORY LAYOUT OF A C PROGRAM

Stack:

Used to store all local variables and is used for passing arguments to functions along with the return address of instruction which is to be executed next once function call is over . All recursive function call are added to stack.

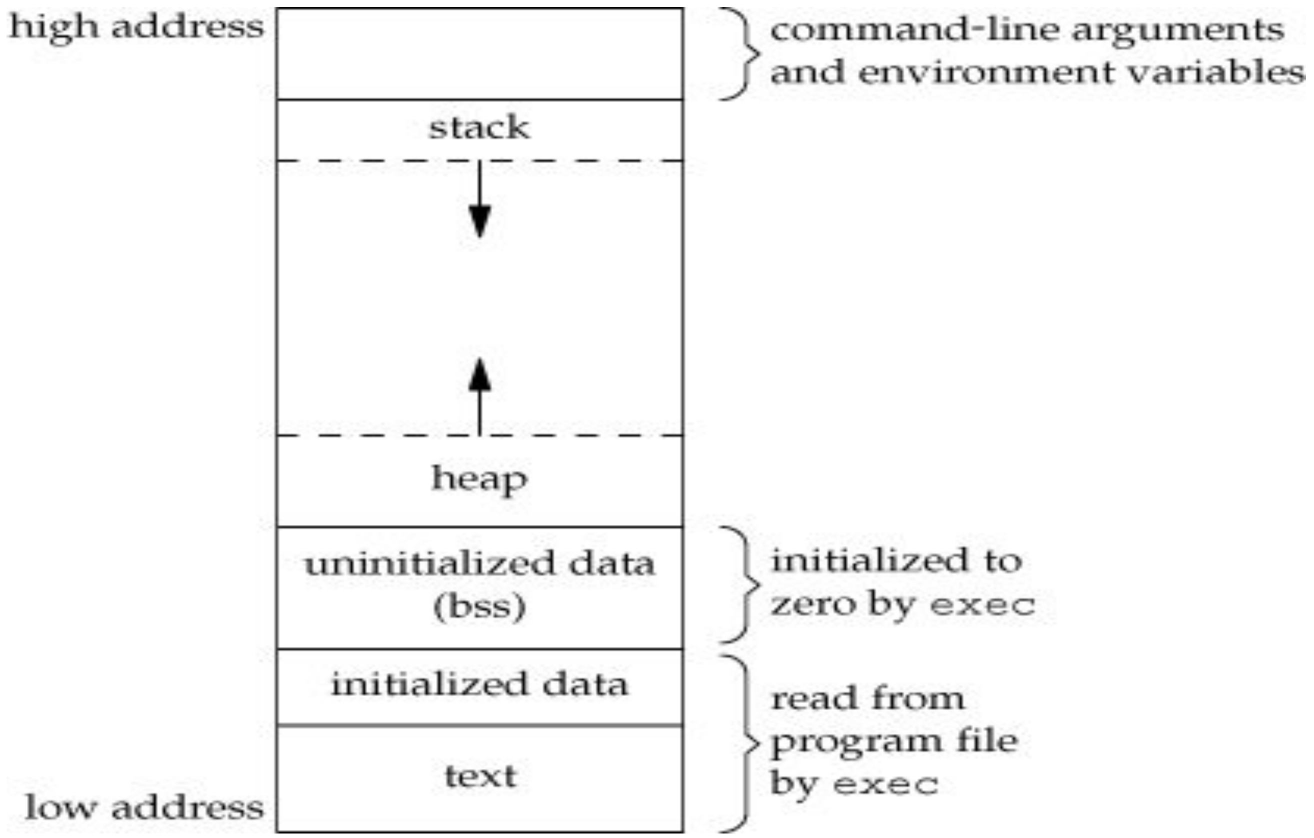
- where local or automatic variables are stored, along with information that is saved each time a function is called.
- Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack.
- The newly called function then allocates room on the stack for its automatic and temporary variables.
- This is how recursive functions in C can work.
- Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

MEMORY LAYOUT OF A C PROGRAM

Heap:

where dynamic memory allocation usually takes place.

Historically, the heap has been located between the uninitialized data and the stack.

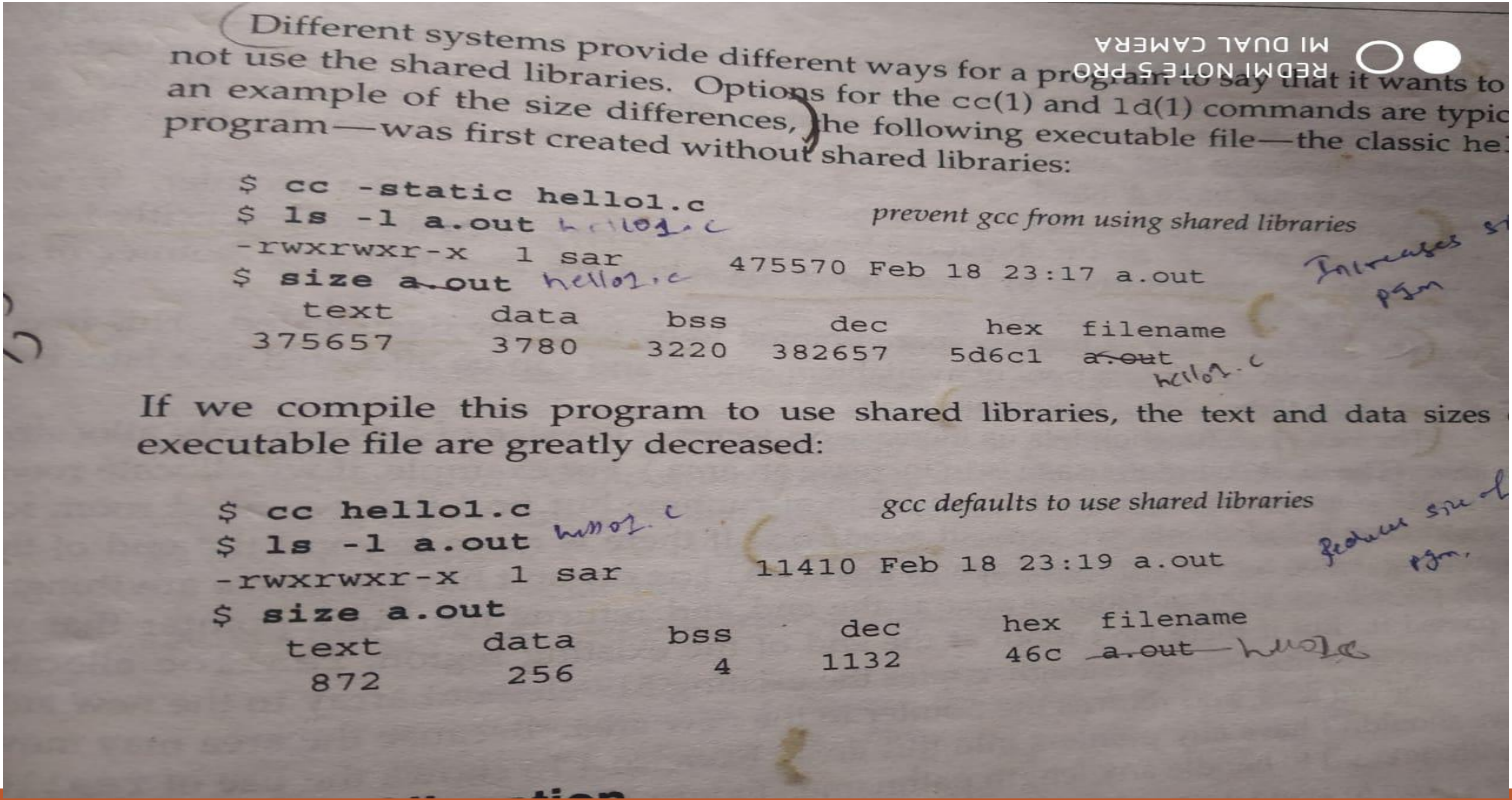

high address
command-line arguments and environment variables
stack
heap
uninitialized data (bss)
initialized to zero by exec
initialized data
text
read from program file by exec
low address

Shared Libraries

- **Shared library remove the common library routines from the executable file instead maintaining a single copy of library somewhere in memory that all processes reference.**

- **This reduces size of each executable file.**

- **Advantage – shared library is that library functions can be replaced with new versions without having to relink edit every program that uses the library.**



REDMI NOTE 5 PRO
MI DUAL CAMERA

Different systems provide different ways for a program to say that it wants to not use the shared libraries. Options for the cc(1) and ld(1) commands are typic an example of the size differences, the following executable file—the classic he program—was first created without shared libraries:

```
$ cc -static hello1.c                        prevent gcc from using shared libraries
$ ls -l a.out hello1.c
-rwxrwxr-x  1 sar        475570 Feb 18 23:17 a.out
$ size ~~a.out~~ hello1.c
    text       data      bss       dec       hex    filename
  375657       3780     3220    382657     5d6c1    ~~a.out~~
                                                       hello1.c
```

Increases st pgm

If we compile this program to use shared libraries, the text and data sizes executable file are greatly decreased:

```
$ cc hello1.c                                gcc defaults to use shared libraries
$ ls -l a.out  hello2.c
-rwxrwxr-x  1 sar         11410 Feb 18 23:19 a.out
$ size a.out
    text        data       bss       dec       hex   filename
     872         256         4      1132       46c   ~~a.out~~ hello2.c
```

Reduces size of pgm.

MEMORY ALLOCATION

- an array is a collection of a fixed number of values. Once the size of an array is declared, you cannot change it.
- Sometimes the size of the array you declared may be insufficient. To solve this issue, you can allocate memory manually during run-time. This is known as dynamic memory allocation in C programming.
- To allocate memory dynamically, library functions are malloc(), calloc(), realloc() and free() are used. These functions are defined in the <stdlib.h> header file.

MEMORY ALLOCATION

ISO C specifies three functions for memory allocation:

malloc:

which allocates a specified number of bytes of memory. The initial value of the memory is indeterminate.

calloc:

which allocates space for a specified number of objects of a specified size. The space is initialized to all 0 bits.

realloc:

which increases or decreases the size of a previously allocated area.
When the size increases, it may involve moving the previously allocated area somewhere else, to provide the additional room at the end. Also, when the size increases, the initial value of the space between the old contents and the end of the new area is indeterminate.

MEMORY ALLOCATION

```
#include <stdlib.h>
void *malloc(size_t size);
void *calloc(size_t nobj, size_t size);
void *realloc(void *ptr, size_t newsize);
```

On success, it returns: non-null pointer , NULL on Error.

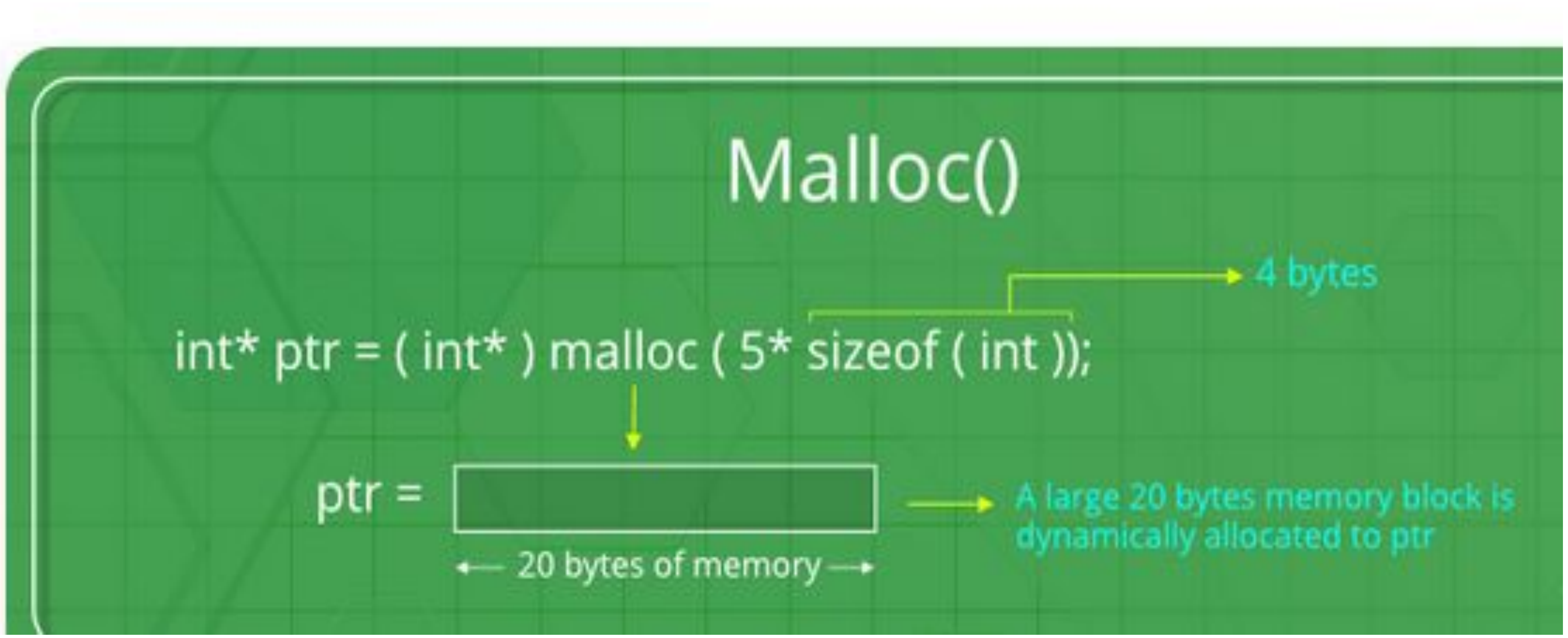
```
void free(void *ptr);
```

The function free causes the space pointed to by ptr to be deallocated.
This freed space is usually put into a pool of available memory and can be allocated in a later call to one of the three alloc functions.

- C malloc() method
- “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size.

- ptr = (int*) malloc(100 * sizeof(int));

- Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

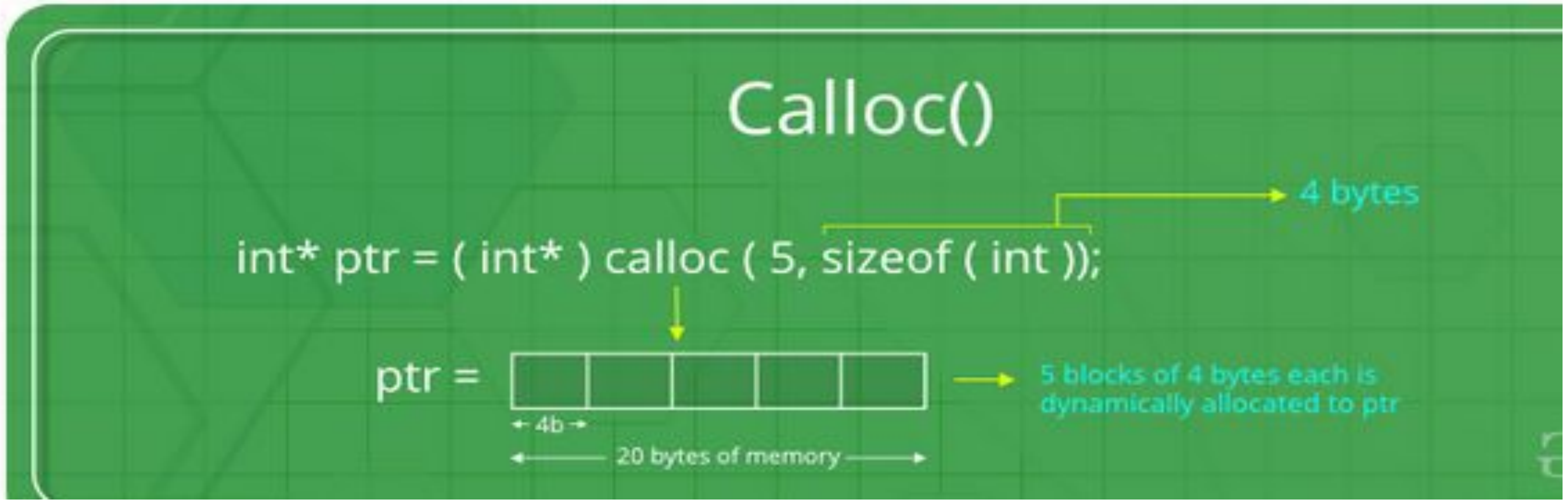

Malloc()
4 bytes
ptr =
20 bytes of memory
A large 20 bytes memory block is dynamically allocated to ptr

```
int* ptr = ( int* ) malloc ( 5* sizeof ( int ));
```

```
#include <stdio.h>
#include <stdlib.h>

int main()
{

  // This pointer will hold the
  // base address of the block created
  int* ptr;
  int n, i;

  // Get the number of elements for the array
  n = 5;
  printf("Enter number of elements: %d\n", n);

  // Dynamically allocate memory using malloc()
  ptr = (int*)malloc(n * sizeof(int));
// Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
        exit(0);
    }
    else {
        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }
        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
    return 0;
}
```

- **Output:**
- **Enter number of elements: 5**
- **Memory successfully allocated using malloc.**
- **The elements of the array are: 1, 2, 3, 4, 5,**

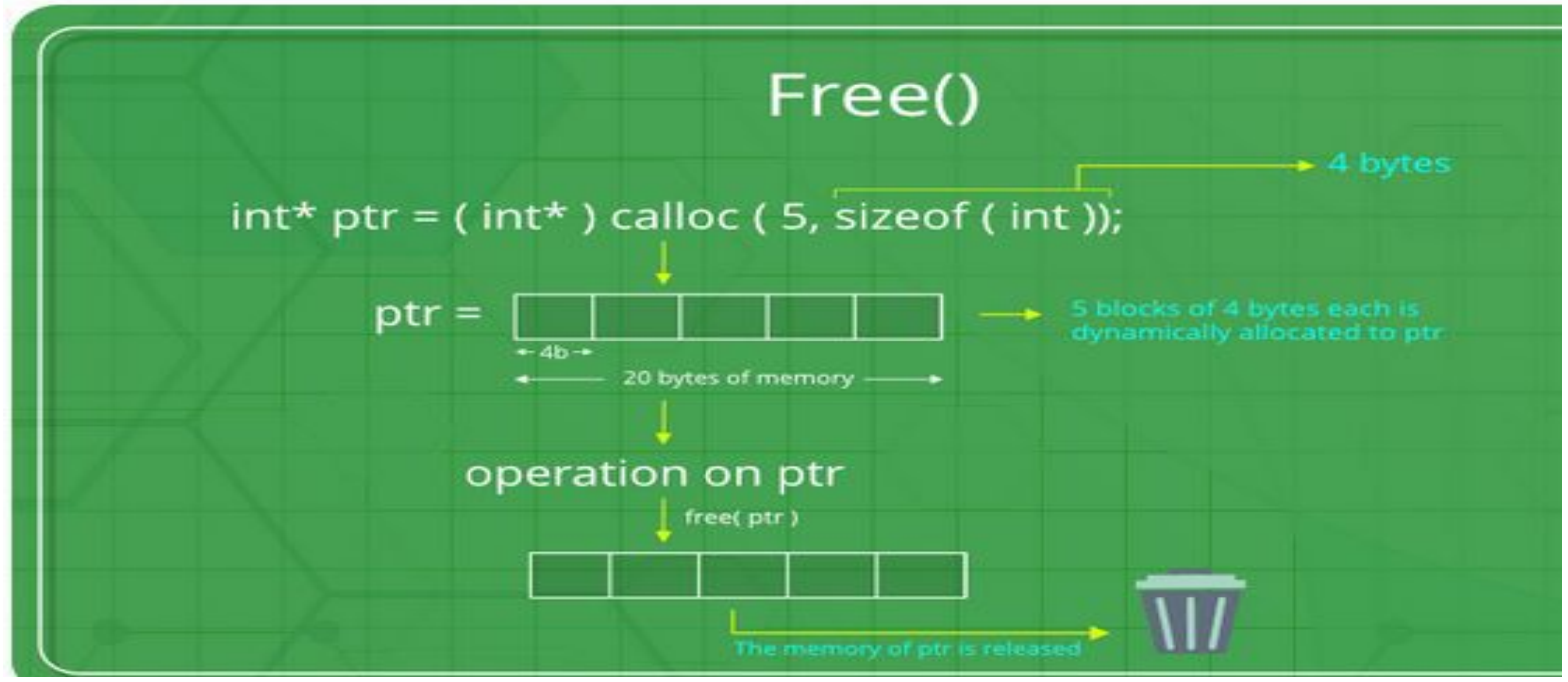
- “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. It initializes each block with a default value ‘0’.
- ptr = (float*) calloc(5, sizeof(float));
- This statement allocates contiguous space in memory for 25 elements each with the size of the float.


Calloc()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory

```c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  // This pointer will hold the
  // base address of the block created
  int* ptr;
  int n, i;
  // Get the number of elements for the array
  n = 5;
  printf("Enter number of elements: %d\n", n);

  // Dynamically allocate memory using calloc()
  ptr = (int*)calloc(n, sizeof(int));

  // Check if the memory has been successfully
  // allocated by calloc or not
  if (ptr == NULL) {
          printf("Memory not allocated.\n");
          exit(0);
  }

•   else {

•           // Memory has been successfully allocated
•           printf("Memory successfully allocated using calloc.\n");
•           // Get the elements of the array
•           for (i = 0; i < n; ++i) {
•                   ptr[i] = i + 1;             }
•           // Print the elements of the array
•           printf("The elements of the array are: ");
•           for (i = 0; i < n; ++i) {
•                   printf("%d, ", ptr[i]);
•           }
•     }     return 0; }
```

```
•   Enter number of elements:
•    5
•   Memory successfully allocated using calloc.
•    The elements of the array are: 1, 2, 3, 4, 5,
```

- **“free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.**
- **free(ptr);**


Free()
4 bytes
int* ptr = (int*) calloc (5, sizeof (int));
ptr =
5 blocks of 4 bytes each is dynamically allocated to ptr
4b
20 bytes of memory
operation on ptr
free(ptr)
The memory of ptr is released

- #include <stdio.h>
- #include <stdlib.h>
- int main()
- {   // This pointer will hold the
-   // base address of the block created
-   int *ptr, *ptr1;
-   int n, i;
-   // Get the number of elements for the array
-   n = 5;
-   printf("Enter number of elements: %d\n", n);
-   // Dynamically allocate memory using malloc()
-   ptr = (int*)malloc(n * sizeof(int));

- // Dynamically allocate memory using calloc()
- ptr1 = (int*)calloc(n, sizeof(int));

- // Check if the memory has been successfully
- // allocated by malloc or not
- if (ptr == NULL || ptr1 == NULL) {
-         printf("Memory not allocated.\n");
-         exit(0);
- }
-

- else {

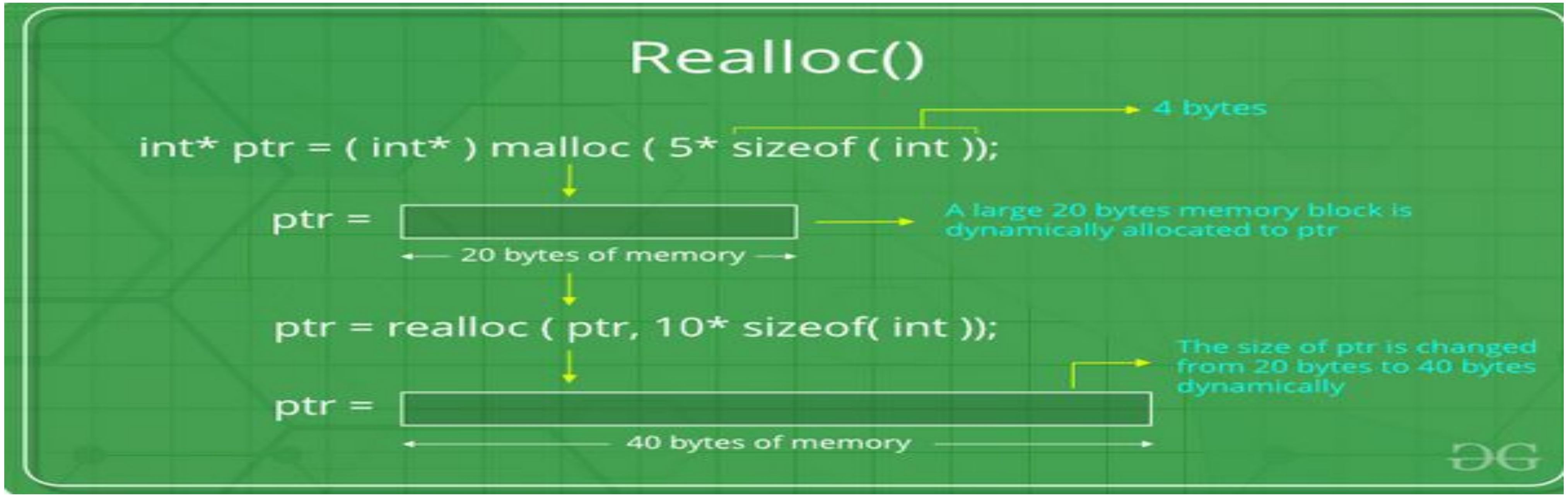
- // Memory has been successfully allocated
- printf("Memory successfully allocated using malloc.\n");

- // Free the memory
- free(ptr);
- printf("Malloc Memory successfully freed.\n");

- // Memory has been successfully allocated
- printf("\nMemory successfully allocated using calloc.\n");

- // Free the memory
- free(ptr1);
- printf("Calloc Memory successfully freed.\n");
- } return 0;
- }

- **“realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory.**
- **<u>ptr = realloc(ptr, newSize);</u>**
- **<u>where ptr is reallocated with new size 'newSize'.</u>**


Realloc()
4 bytes
int* ptr = (int*) malloc (5* sizeof (int));
ptr =
A large 20 bytes memory block is dynamically allocated to ptr
20 bytes of memory
ptr = realloc (ptr, 10* sizeof(int));
The size of ptr is changed from 20 bytes to 40 bytes dynamically
ptr =
40 bytes of memory

```
• #include <stdio.h>
• #include <stdlib.h>
• int main()
• { // This pointer will hold the
•   // base address of the block created
•   int* ptr;
•   int n, i;
•   // Get the number of elements for the array
•   n = 5;
•   printf("Enter number of elements: %d\n", n);
•   // Dynamically allocate memory using calloc()
•   ptr = (int*) calloc(n, sizeof(int));
•   // Check if the memory has been successfully
•   // allocated by malloc or not
•   if (ptr == NULL) {
•           printf("Memory not allocated.\n");
•           exit(0);
•   }       else {
•           // Memory has been successfully allocated
•           printf("Memory successfully allocated using calloc.\n");

•           // Get the elements of the array
•           for (i = 0; i < n; ++i) {
•                   ptr[i] = i + 1;
•           }

•
// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
// Get the new size for the array
n = 10;
printf("\n\nEnter the new size of the array: %d\n", n);
// Dynamically re-allocate memory using realloc()
ptr = realloc(ptr, n * sizeof(int));
// Memory has been successfully allocated
printf("Memory successfully re-allocated using realloc.\n");

// Get the new elements of the array
for (i = 5; i < n; ++i) {
ptr[i] = i + 1;
}
// Print the elements of the array
printf("The elements of the array are: ");
for (i = 0; i < n; ++i) {
printf("%d, ", ptr[i]);
}
free(ptr);
}
```

- Enter number of elements: 5
- Memory successfully allocated using calloc.
- The elements of the array are: 1, 2, 3, 4, 5,
- Enter the new size of the array: 10
- Memory successfully re-allocated using realloc.
- The elements of the array are: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,

Environment Variables

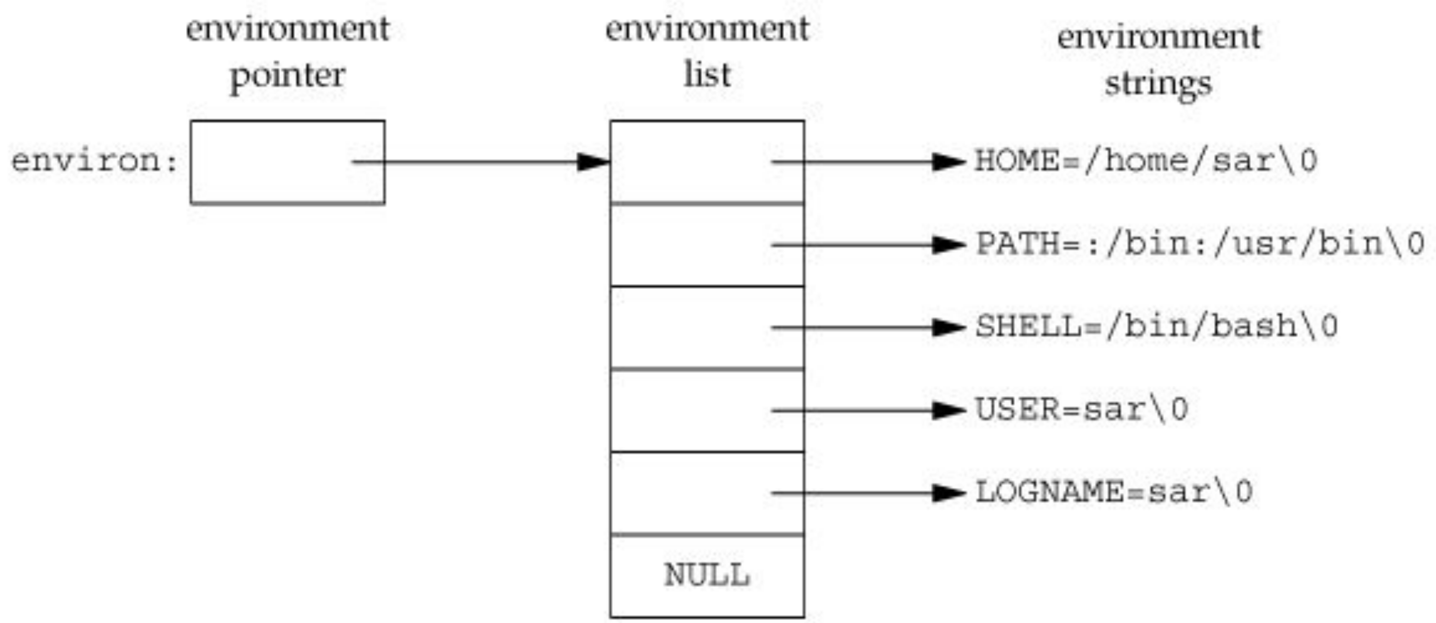
- They are part of the operating environment in which a process runs. For example, a running process can query the value of the TEMP environment variable to discover a suitable location to store temporary files, or the HOME or USERPROFILE variable to find the directory structure owned by the user running the process.
- Environment variables control the behavior of the system. They determine the environment in which you work.
- General form
- *variable=string*
- Examples:
- HOME=/usr1/stevens
- Few Environmental Variables are : CDPATH, COLUMNS, EDITOR, ENV, HISTFILE, HISTSIZE, HOME, LINES, LOGNAME, MAIL, MAILCHECK, MAILPATH, OLDPWD, PATH, PS1, PS2, PS3, PS4, PWD, RANDOM, REPLY, SECONDS, SHELL, TERM, TMOUT, VISUAL
- Environment List:
- Comprises list of Environment Variables.

ENVIRONMENT LIST

Each program is also passed an environment list.
Like the argument list, the environment list is an array of character pointers, with each pointer containing the address of a null-terminated C string.
The address of the array of pointers is contained in the global variable environ:

extern char **environ;

Generally any environmental variable is of the form: ***name=value.***



Accessing Environment list

- **External variable named *environ***
- **External variable named *environ* can be used to access the environment list**
- ***extern char **environ;***
- **Function *getenv()***

UNIT 3 UNIX PROCESSES

ENVIRONMENT VARIABLES

The environment strings are usually of the form: ***name=value***.

The functions that we can use to set and fetch values from the variables are setenv, putenv, and getenv functions.

The prototype of these functions are:

```
#include <stdlib.h>
char *getenv(const char *name);
```

Returns: pointer to value associated with name, NULL if not found.

Eg:

```
char *res=getenv(“HOME”);
cout<<“HOME=“<<res<<endl;
```

output:

```
HOME=/home/syed
```

UNIT 3 UNIX PROCESSES

<u>ENVIRONMENT VARIABLES</u>

int putenv(char ****str*);**
int setenv(const char ****name*, const char ****value***, int** ***rewrite*****);**
int unsetenv(const char ****name*);**

All return: 0 if OK, nonzero on error.

- The **putenv** function takes a string of the form **name=value** and places it in the environment list.
- If name already exists, its old definition is first removed.

ENVIRONMENT VARIABLES

int setenv(const char ****name*, const char ****value***, int** ***rewrite*****);**
All return: 0 if OK, nonzero on error.

The **setenv** function sets name to value.
If name already exists in the environment, then
if rewrite is nonzero, the existing definition for name is first removed;
if rewrite is 0, an existing definition for name is not removed, name is not set to the new value, and no error occurs.

UNIT 3 UNIX PROCESSES

ENVIRONMENT VARIABLES

int unsetenv(const char ****name*);**

All return: 0 if OK, nonzero on error.

The **unsetenv** function removes any definition of name. It is not an error if such a definition does not exist.

UNIX PROCESSES

<u>Setjmp() & longjmp() FUNCTIONS</u>

In C, we can't goto a label that's in another function.

Instead, we must use the setjmp and longjmp functions to perform this type of branching.

#include <setjmp.h>

int setjmp(jmp_buf env);

Returns: 0 if called directly, nonzero if returning from a call to longjmp

void longjmp(jmp_buf env, int val);

- The setjmp function records or marks a location in a program code so that later when the longjmp function is called from some other function, the execution continues from the location onwards.
- The env variable(the first argument) records the necessary information needed to continue execution.
- The env is of the jmp_buf defined in <setjmp.h> file, it contains the task.

- **Setjmp- places the book mark.**

- **Longjmp- goes to where bookmark was placed.**

- **#include<stdio.h>**
- **Jmp_buf resume_here;**
- **Int main()**
- **{**

```
 if setjmp(resume_here)
 {
 Printf(“After longjmp back in main()”);
 Printf(“jmp buffer variable resume here”);
 }
 Else{ printf(“setjmp returns first time “);
         hello();
 } }
```

- **Void hello()**
- **{ hello1();**
- **}**

hello1()

{

Printf("<u>setjmp returns first time </u>");

longjmp(resume_here,1);

Printf("cant be reached here because I did longjmp");

}

- **Output**
- **setjmp returns first time**
- **setjmp returns first time**
- **After longjmp back in main()**
- **jmp buffer variable resume here**

- **When setjmp called first time returns value zero and hence else clause is executed . This time setjmp returned value 1 passed by longjmp and if clause is executed.**

UNIT 3 UNIX PROCESSES

Setjmp() & longjmp()

```
#include <stdio.h>  #include <setjmp.h>
jmp_buf jb;
int main(int argc, char *argv[])
{
int a, b, c;
printf ("Give two numbers for division : ");
scanf("%d %d", &a, &b);  // 5  and 0
if(setjmp(jb) == 0)
{
c = division(a, b);
printf ("%d / %d = %d", a, b, c);
return 0;
}
else
{
handle_error();
return -1;
}
}
```

```
int division(int a, int b)
{
if(b == 0)
longjmp(jb, 1);
else
return (a/b);
}

void handle_error(void)
{
printf("Divide by zero error !");
}
```

UNIT3 UNIX PROCESSES

getrlimit() AND setrlimit() FUNCTIONS

Every process has a set of resource limits, some of which can be queried and changed by the getrlimit and setrlimit functions.

It allows process to read and set limits on the system resources that it can consume.

```
#include <sys/resource.h>
int getrlimit(int resource, struct rlimit *rlptr);
int setrlimit(int resource, const struct rlimit *rlptr);
```

Both return: 0 if OK, nonzero on error

Each call to these two functions specifies a single resource and a pointer to the following structure:

```
struct rlimit
{
rlim_t  rlim_cur;   /* soft limit: current limit */
rlim_t  rlim_max; /* hard limit: maximum value for rlim_cur */
};
```

getrlimit() AND setrlimit() FUNCTIONS

Three rules govern the changing of the resource limits.

1. A process can change its soft limit to a value less than or equal to its hard limit.
2. A process can lower its hard limit to a value greater than or equal to its soft limit.
3. Only a super user process can raise or change a hard limit.

An infinite limit is specified by the constant RLIM_INFINITY.

UNIT 3 UNIX PROCESSES

Resource Argument takes one of the following values.

RLIMIT_AS	The maximum size in bytes of a process's total available memory.
RLIMIT_CORE	The maximum size in bytes of a core file. A limit of 0 prevents the creation of a core file.
RLIMIT_CPU	The maximum amount of CPU time in seconds. When the soft limit is exceeded, the SIGXCPU signal is sent to the process.
RLIMIT_DATA	The maximum size in bytes of the data segment: the sum of the initialized data, uninitialized data, and heap.
RLIMIT_FSIZE	The maximum size in bytes of a file that may be created. When the soft limit is exceeded, the process is sent the SIGXFSZ signal.
RLIMIT_LOCKS	The maximum number of file locks a process can hold.
RLIMIT_NOFILE	The maximum number of open files per process. Changing this limit affects the value returned by the sysconf function for its _SC_OPEN_MAX argument
RLIMIT_NPROC	The maximum number of child processes per real user ID. Changing this limit affects the value returned for _SC_CHILD_MAX by the sysconf function

- Int main()
- {
- Struct rlimit r1;
- Getrlimit (RLIMIT_CPU, &r1);
- //set CPU limit of 1 second
- R1.rlim_cur=1;
- Setrlimit(RLIMIT_CPU,&r1);
- Return 0;
- }

UNIX KERNEL SUPPORT FOR PROCESS

The process will be assigned with attributes, which are either inherited from its parent or will be set by the kernel.

Attributes	Meaning
real user identification number (rUID)	the user ID of a user who created the parent process
real group identification number (rGID)	the group ID of a user who created that parent process
effective user identification number (eUID)	this allows the process to access and create files with the same privileges as the program file owner.
effective group identification number (eGID)	this allows the process to access and create files with the same privileges as the group to which the program file belongs.
Saved set-UID and saved set-GID	these are the assigned eUID and eGID of the process respectively
Process group identification number (PGID) and session identification number (SID)	these identify the process group and session of which the process is member
Supplementary group identification numbers	this is a set of additional group IDs for a user who created the process

UNIX KERNEL SUPPORT FOR PROCESS

In addition to the above attributes, the following attributes are different between the parent and child processes:

Attributes	Meaning
Process identification number (PID)	an integer identification number that is unique per process in an entire operating system.
Parent process identification number (PPID)	the parent process PID
Pending signals	the set of signals that are pending delivery to the parent process
Alarm clock time	the process alarm clock time is reset to zero in the child process
File locks	the set of file locks owned by the parent process is not inherited by the chid process

UNIT 3 PROCESS CONTROL

PROCESS IDENTIFIERS

#include <unistd.h>	
pid_t getpid(void);	Returns: process ID of calling process
pid_t getppid(void);	Returns: parent process ID of calling process
uid_t getuid(void);	Returns: real user ID of calling process
uid_t geteuid(void);	Returns: effective user ID of calling process
gid_t getgid(void);	Returns: real group ID of calling process
gid_t getegid(void);	Returns: effective group ID of calling process

UNIT 3 PROCESS CONTROL

<u>fork FUNCTION</u>

An existing process can create a new one by calling the fork function.

#include <unistd.h>
pid_t fork(void);

Returns: 0 in child, process ID of child in parent, 1 on error.

The new process created by fork is called the child process.
This function is called once but returns twice.
The only difference in the returns is that the return value in the child is 0, whereas the return value in the parent is the process ID of the new child.
The reason the child's process ID is returned to the parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.

UNIT 3 PROCESS CONTROL

fork FUNCTION

Example programs:

Program 1

```
/* Program to demonstrate fork function Program name – fork1.c */
#include<unistd.h>
void main( )
{
fork( );
printf("\n hello USP");
}
```

Output :

```
$ cc fork1.c
$ ./a.out
hello USP
hello USP
```

Note : The statement hello USP is executed twice as both the child and parent have executed that instruction.

UNIT 3 PROCESS CONTROL

fork FUNCTION

Example programs:

Program 2 /* Program name – fork2.c */

```
#include<unistd.h>
 void main( )
{
printf("\n 6 sem ");
 fork( );
printf("\n hello USP");
}
```

Output :

```
$ cc fork2.c
$ ./a.out
  6 sem
 hello USP
 hello USP
```

Note: The statement 6 sem is executed only once by the parent because it is called before fork and statement hello USP is executed twice by child and parent.