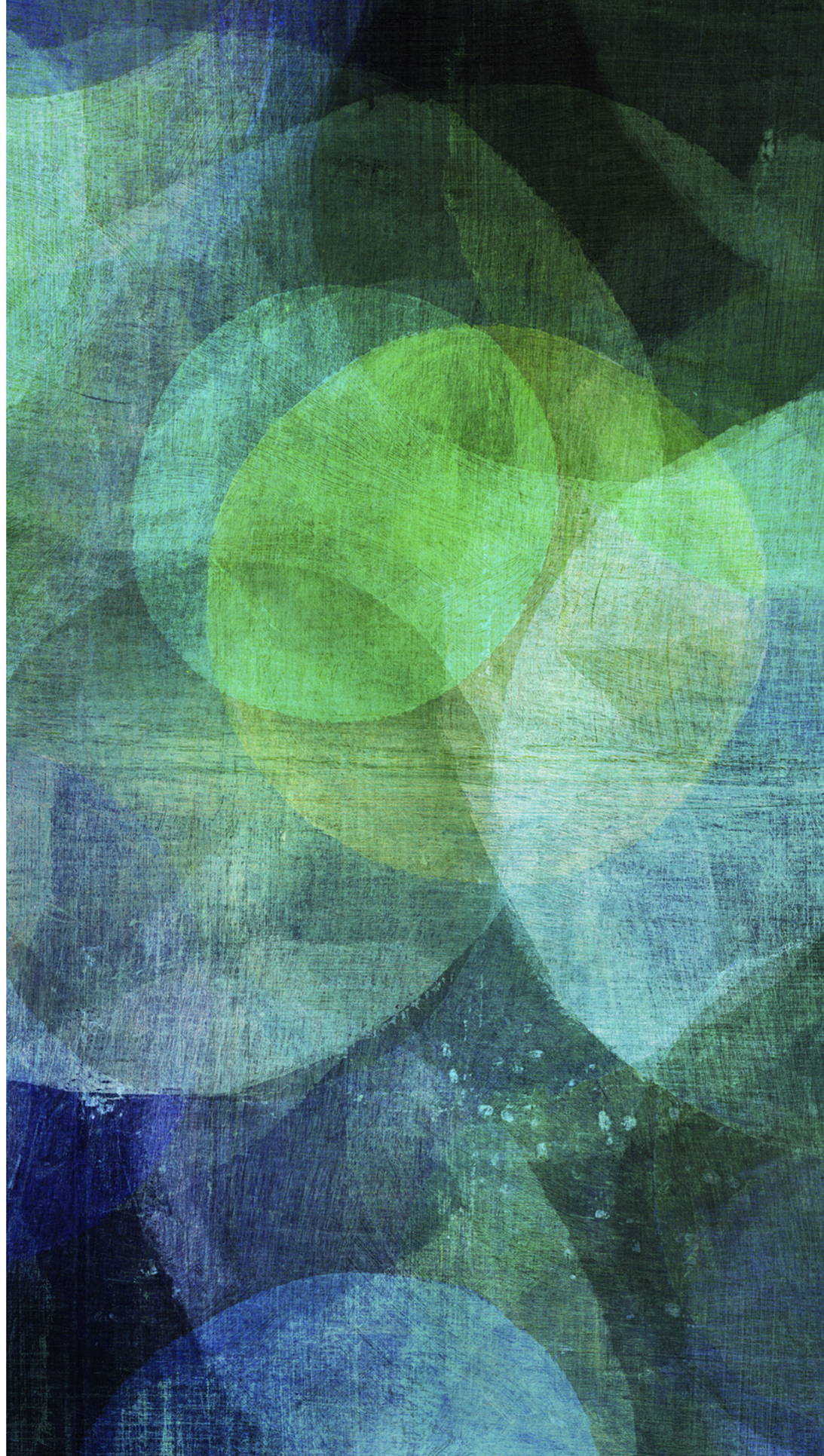
LINUX PROGRAMMING

Process
(3 Week)

PROCESS ENVIRONMENT



INTRODUCTION

- We need to examine the environment of a single process.
- We'll see how the main function is called when the program is executed
- How command-line arguments are passed to the new program
- What the typical memory layout looks like
- How to allocate additional memory

MAIN FUNCTION

- A C program start execution with a function called **main.**
- The prototype for the main function is

```
int main (int argc, char *argv[]);
```

- **argc** is the number of command-line arguments
- **argv** is an array of pointers to the arguments

MAIN FUNCTION

```
#include "apue.h"

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);
}
```

Figure 7.4 Echo all command-line arguments to standard output

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

PROGRAM VS PROCESS

What is the program?

- A program is an executable file which contains a certain set of instructions written to complete the specific job on your computer.
- For example, Google browser **chrome.exe**

What is the process?

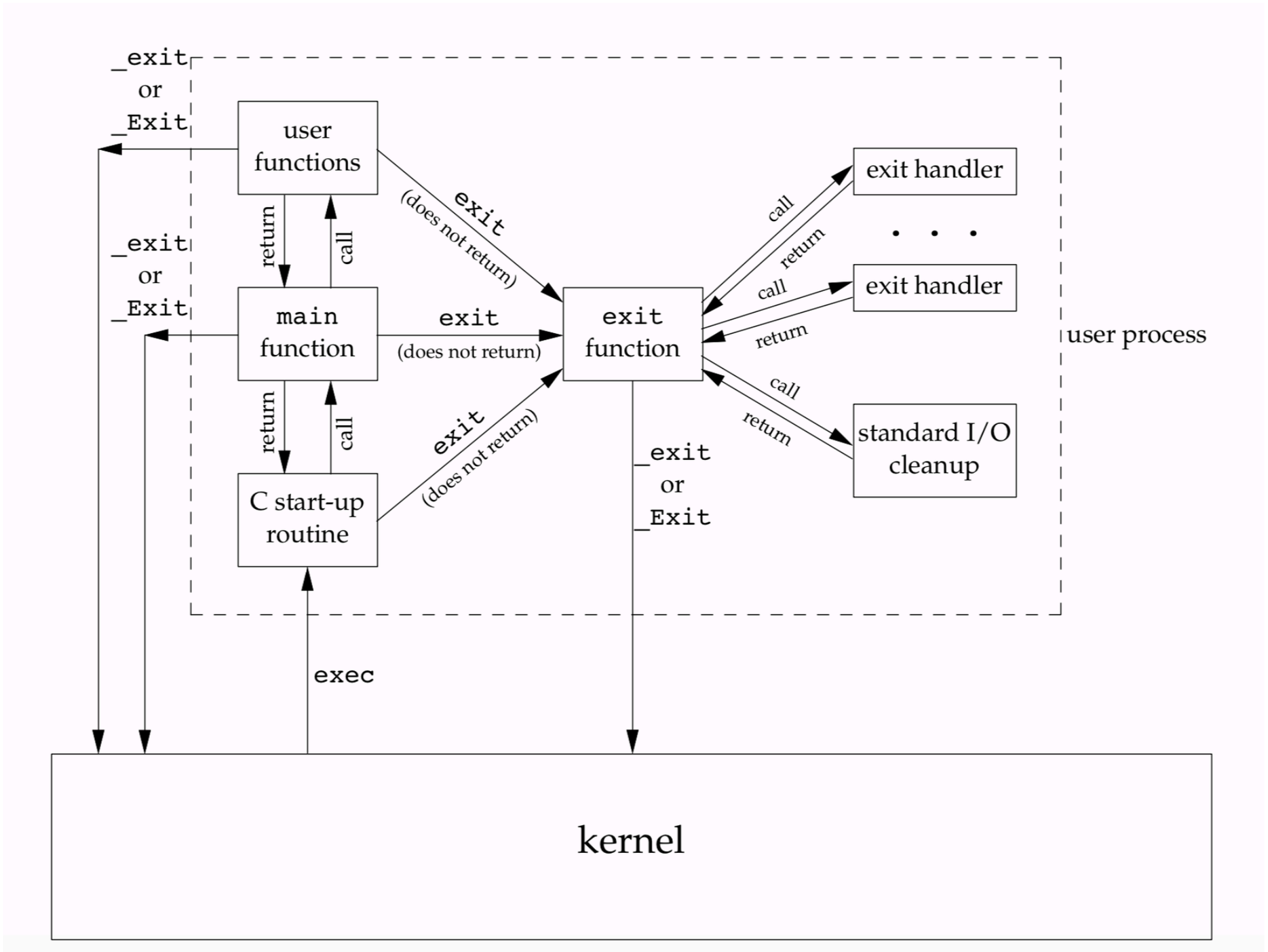
- A process is an execution of any specific program. It is considered an active entity that actions the purpose of the application.
- For example, If you are double click on your Google Chrome browser icon on your PC or laptop, you start a process which will run the Google Chrome program.

PROCESS TERMINATION

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

- Three function terminate a program normally: **_exit** and **_Exit**, which return to the kernel immediately, and **exit**, which performs certain cleanup processing and then return to the kernel.
- Return 0; == exit(0);

HOW A C PROGRAM IS STARTED AND HOW IT TERMINATE



MEMORY LAYOUT OF A C PROGRAM

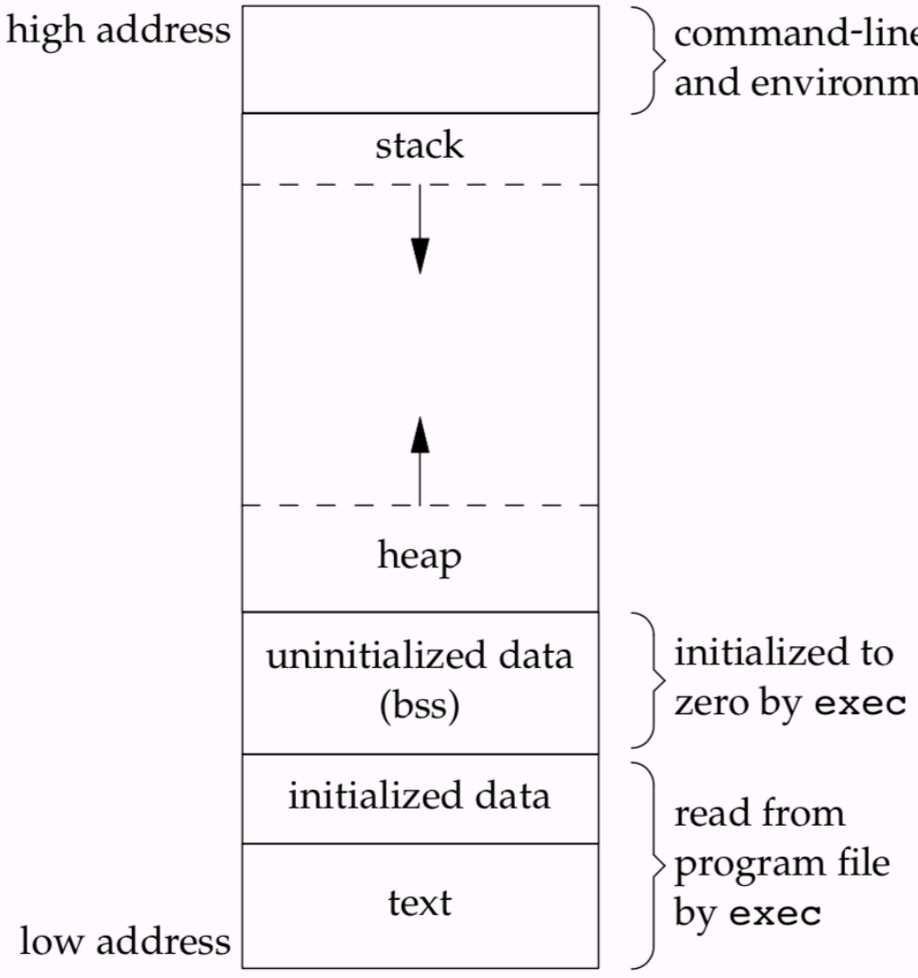


Figure 7.6 Typical memory arrangement

- **Text :**

  Consisting of 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

- **Initialized data :**

  Containing variables that are specifically initialized in the program.

- **Uninitialized data :**

  Data in the segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing.

  EX) long sum[100];

- **Stack:**

  The newly called function then allocates room on the stack for its automatic and temporary variables.

- **Heap:**

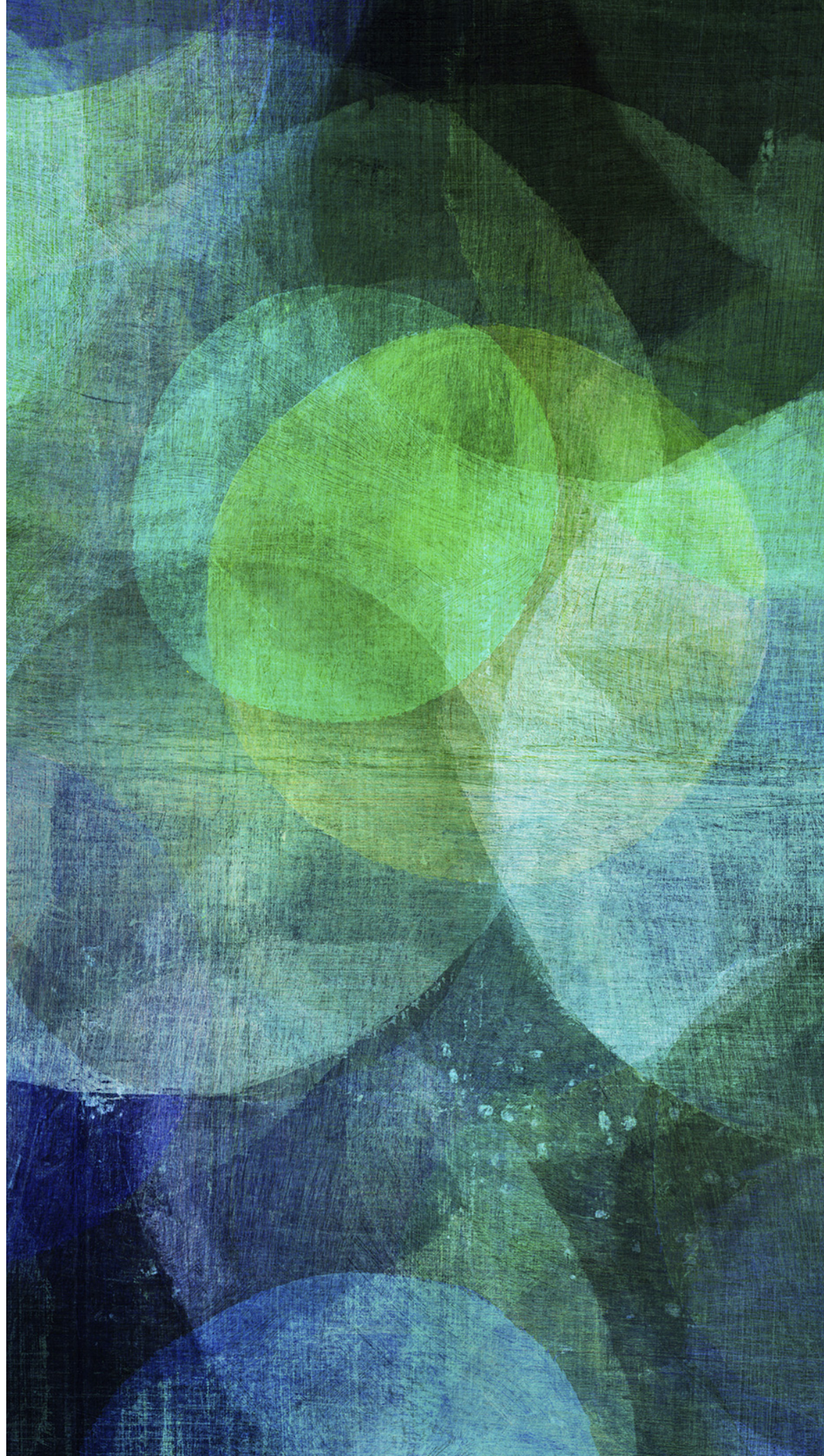
  Where dynamic memory allocation usually takes place.

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);

                    All three return: non-null pointer if OK, NULL on error

void free(void *ptr);
```

- **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.

PROCESS CONTROL



PROCESS IDENTIFIERS

- Every process has a unique process ID, a non-negative integer.Because the process ID is the only well-known identifier of a process that is always unique, it is often used as a piece of other identifiers, to guarantee uniqueness.

- Although unique, process IDs are reused. As a process terminate, their IDs become candidates for reuse.

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

FORK FUNCTION

```
#include <unistd.h>

pid_t fork(void);

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

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

FORK FUNCTION

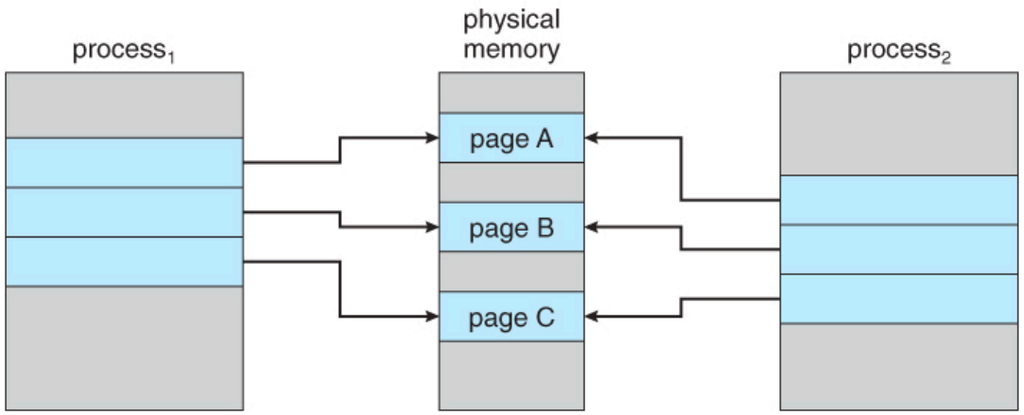
Why fork function return child PID in parent process and return 0 in child process?

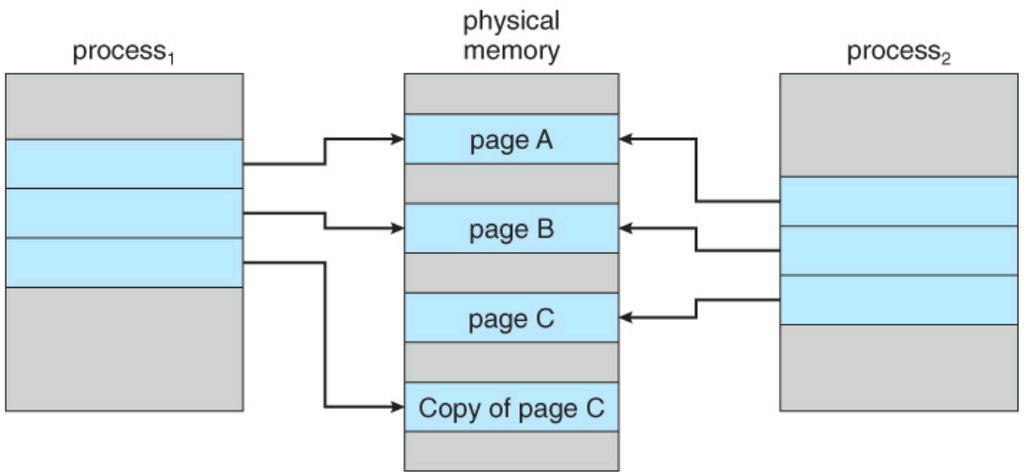
- The reason the child's process ID is returned to the parent is that parent is that a process can have more than one child, and there is no function that allows a process to obtain the process IDs of its children.
- The reason **fork** returns 0 to the child is that a process can have only a single parent, and the child can always call **getppid** to obtain the process ID of its parent.

COPY OF THE PARENT

- Both the child and the parent continue executing with the instruction that follows the call to **fork.**
- The child is a copy of the parent. For example, the child gets a copy of the parent's data space, heap, and stack.
- Note this is a copy for the child. The parent and the child do not share these portions of memory

COW(COPY ON WRITE)





- Instead, a technique called *copy-on-write* (COW) is used in modern.

- These regions are shared by the parent and the child and have their protection changed by the kernel to read-only.

FORK FUNCTION

```c
#include<unistd.h>
#include<stdio.h>

int main(){
    pid_t pid;

    pid = fork();

    if(pid==0)
        printf("child process : fork return is %d\n",pid);
    else if(pid>0)
        printf("parent process : fork return is %d\n",pid);

    return 0;
}
```

```
jinu@jeongjin-uui-MacBook-Pro [22:59:05] ~/Desktop/git/linuxprogramming/pa3
$ ./fork
parent process : fork return is 8426
child process : fork return is 0
```

WAIT AND WAITPID FUNCTION

```
#include <sys/wait.h>

pid_t wait(int *statloc);

pid_t waitpid(pid_t pid, int *statloc, int options);
```

Both return: process ID if OK, 0 (see later), or –1 on error

- We use the **wait** function for sequence process.
- When a process terminates, the kernel notifies the parent by sending the signal to the parent.
- Because the termination of a child is an asynchronous event — it can happen at any time while the parent is running —this signal is the asynchronous notification from the kernel to the parent.
- If a child has already terminated and is a zombie, wait return immediately with that child's status.

ZOMBIE AND ORPHAN PROCESS

What is the **orphan process**?

- If the parent process exits before the child process, the init process becomes the new parent process of the child process.

What is the **zombie process**?

- Conversely, child process may terminate before the parent process.
- Since the parent process may want to know the status of the child process after the child process has terminated, the kernel will have minimal information (process ID, process exit status, etc .) even if the child process terminates.
- When the parent process reclaims the exit status of the tomb process (through a call to the wait system call), the zombie process is removed.

WAIT FUNCTION

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

int main(){
    pid_t pid;
    int status;
    pid = fork();

    if(pid==0){
        sleep(1);
        printf("Child process\n");
        exit(1);
    }else if(pid > 0){
        wait(&status);
    }else{
        perror("fork() failed");
        exit(0);
    }

    printf("Parent process\n");
    return 0;
}
```

```
jinu@jeongjin-uui-MacBook-Pro [23:48:01] ~/Desktop/git/linuxprogramming/pa3
$ ./wait
Child process
Parent process
```

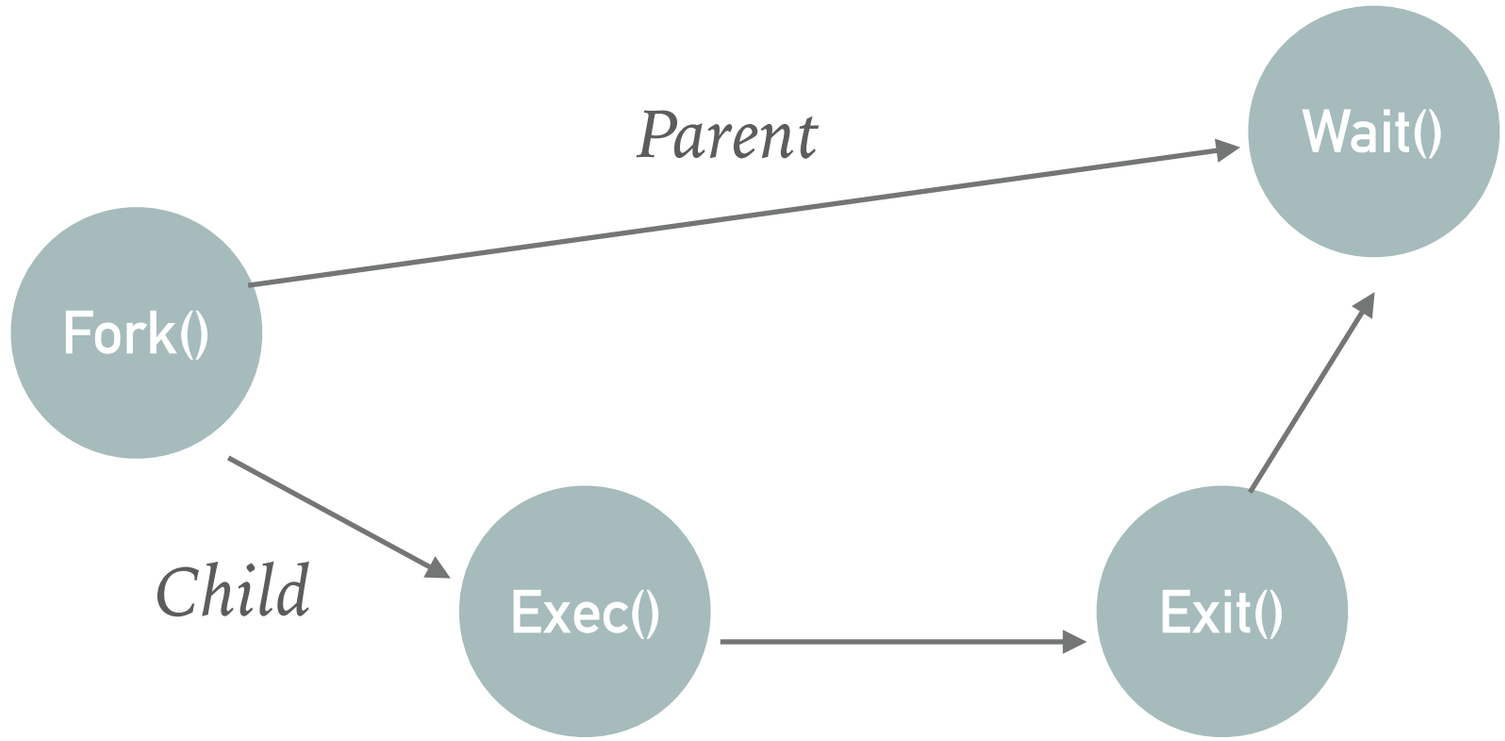
EXEC FUNCTION

```
#include <unistd.h>

int execl(const char *pathname, const char *arg0, ... /* (char *)0 */ );

int execv(const char *pathname, char *const argv[]);

int execle(const char *pathname, const char *arg0, ...
           /* (char *)0, char *const envp[] */ );

int execve(const char *pathname, char *const argv[], char *const envp[]);

int execlp(const char *filename, const char *arg0, ... /* (char *)0 */ );

int execvp(const char *filename, char *const argv[]);

int fexecve(int fd, char *const argv[], char *const envp[]);
```

All seven return: –1 on error, no return on success

- When a process calls one of the exec functions, that process is completely replaced by the new program, and the new program starts executing at its main function

EXEC FUNCTION



- exec() system call used after fork() to replace the process's memory space with a new program.

EXEC FUNCTION

```c
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include<stdlib.h>

int main(){
    pid_t pid;
    int status;

    pid = fork();
    if(pid<0){
        perror("fork() failed");
        exit(1);
    }else if(pid==0){
        execl("/bin/ls","ls",NULL);
    }else{
        wait(&status);
    }

    return 0;
}
```

```
jinu@jeongjin-uui-MacBook-Pro [00:16:42] ~/Desktop/git/linuxprogramming/pa3
$ ls
a.out*  argv*   argv.c  exec*   exec.c  fork*   fork.c  wait*   wait.c

jinu@jeongjin-uui-MacBook-Pro [00:16:46] ~/Desktop/git/linuxprogramming/pa3
$ ./exec
a.out   argv    argv.c  exec    exec.c  fork    fork.c  wait    wait.c
```

ENVIRONMENT VARIABLE

It's a variable that persists for the life of a terminal session. Application running in that session access these variables when they need information about the user.

PATH environment variable

```
dbettis@rhino[~]$ echo $PATH
/usr/local/bin:/bin:/usr/bin:/sbin:/usr/sbin:.
```

- This has a special format.
- When you execute a command, the shell **searches through each of these directories, one by one**, until it finds a directory where the executable exists.