system() — Short Notes

Purpose Runs a shell command from a C program.

```
system("ls");
```

- Creates a **child process**
- Shell executes `ls`
- Program **waits** until command finishes
- Output of `ls` prints on terminal

Execution Flow 1) Print: `the program has been started` 2) Run `ls` 3) Print: `end of the program`

Conceptual Output

```
the program has been started
<ls output>
end of the program
```

Key Points - Uses `/bin/sh` internally - Blocks until command completes

exit(), atexit(), _exit() — Short Notes

exit()

Standard library function (from `<stdlib.h>`).

- Performs **normal program termination**
- Flushes stdio buffers (prints pending output)
- Closes open files
- Runs all registered `atexit()` handlers

```
exit(status);
```

Used when you want a **clean and graceful shutdown**.

---

atexit()

Registers a function to run when `exit()` is called.

```
atexit(func_name);
```

- Functions run in **reverse order of registration**
- Useful for cleanup tasks
- freeing resources
- writing logs
- closing files

Does **not** run if `_exit()` is used.

_exit()

System call style termination (from `<unistd.h>`).

```
_exit(status);
```

- Terminates **immediately**
- Does **NOT**:
- flush stdio buffers
- run `atexit()` handlers
- close stdio streams gracefully

Mainly used in **child process after fork()** to avoid duplicate flushing.

Quick Comparison

Function	Flushes stdio	Runs atexit	Use case
`exit()`	Yes	Yes	Normal termination
`_exit()`	No	No	Child process / immediate exit
`atexit()`	—	Registers cleanup	Cleanup on exit

`exit()`, `atexit()`, and `_exit()` — Short Notes

`exit(int status)` (C Standard Library)

- Declared in `<stdlib.h>`
- Performs **normal program termination**
- Tasks performed before exiting:
- Calls all functions registered via `atexit()` (in **reverse order of registration**)
- Flushes all open output streams

- Closes open FILE streams
- Returns control to the OS with `status`
- Typical usage:
- `exit(0);` → success
- `exit(1);` → failure / error

Example

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

int main() {
    printf("Program exiting...
");
    exit(0);
}
```

`atexit(void (*func)(void))`

- Declared in `<stdlib.h>`
- Registers a function to be called **automatically when** `exit()` **is invoked**
- Maximum number of handlers is implementation-dependent
- Handlers run in **LIFO order** (last registered = executed first)

Example

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

void cleanup1() { printf("Cleanup 1 executed
"); }
void cleanup2() { printf("Cleanup 2 executed
"); }

int main() {
    atexit(cleanup1);
    atexit(cleanup2);

    printf("Exiting using exit()...
");
    exit(0);
}
```

Output order

```
Exiting using exit()...
Cleanup 2 executed
Cleanup 1 executed
```

`_exit(int status)` **(POSIX — `<unistd.h>`)**

- Used for **immediate program termination** (mainly after `fork()` failures or in child process)
- **Does NOT**:
- call `atexit()` handlers
- flush stdio buffers
- close standard FILE streams properly
- Directly terminates the process & returns status to kernel

Example (difference after fork)

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

void cleanup() { printf("Cleanup called
"); }

int main() {
    atexit(cleanup);

    pid_t pid = fork();

    if (pid == 0) { // Child
        printf("Child using _exit()
");
        _exit(0);    // cleanup() NOT called
    }
    else {
        printf("Parent using exit()
");
        exit(0);     // cleanup() WILL be called
    }
}
```

Quick Comparison

Feature	`exit()`	`atexit()`	`_exit()`
Flush stdio buffers	✅Yes	—	❌No
Call cleanup handlers	✅Yes	Registers handlers	❌No
Typical use	Normal termination	Resource cleanup	After `fork()` / abnormal end
Header	`<stdlib.h>`	`<stdlib.h>`	`<unistd.h>`

Key Takeaways

- Use `exit()` for normal termination
- Use `atexit()` to register cleanup actions
- Use `_exit()` when you must exit **immediately** (especially inside child process after `fork()`)