

how to get the time when a process started and time when it ended



To capture the time when a process starts and when it ends, you can do so within the code that manages the process. Here's a simple approach using C:

## 1. Capturing Start and End Time using `gettimeofday()`

You can use `gettimeofday()` to record the time when a process starts and ends. Here's how you can do it:

```
c Copy code

#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>

int main() {
    struct timeval start_time, end_time;

    // Capture start time
    gettimeofday(&start_time, NULL);

    // Simulate a process (e.g., sleep for 3 seconds)
    sleep(3);

    // Capture end time
    gettimeofday(&end_time, NULL);

    // Print start time
    printf("Process started at: %ld seconds and %ld microseconds\n",
        (long)start_time.tv_sec, (long)start_time.tv_usec);

    // Print end time
    printf("Process ended at: %ld seconds and %ld microseconds\n",
        (long)end_time.tv_sec, (long)end_time.tv_usec);

    // Calculate the duration
    long diff_sec = end_time.tv_sec - start_time.tv_sec;
    long diff_usec = end_time.tv_usec - start_time.tv_usec;
    printf("Duration: %ld seconds and %ld microseconds\n", diff_sec, diff_usec);
}
```

```
int main() {
    struct timeval start_time, end_time;

    // Capture start time
    gettimeofday(&start_time, NULL);

    // Simulate a process (e.g., sleep for 3 seconds)
    sleep(3);

    // Capture end time
    gettimeofday(&end_time, NULL);

    // Print start time
    printf("Process started at: %ld seconds and %ld microseconds\n",
        (long)start_time.tv_sec, (long)start_time.tv_usec);

    // Print end time
    printf("Process ended at: %ld seconds and %ld microseconds\n",
        (long)end_time.tv_sec, (long)end_time.tv_usec);

    // Calculate the duration
    long diff_sec = end_time.tv_sec - start_time.tv_sec;
    long diff_usec = end_time.tv_usec - start_time.tv_usec;
    printf("Duration: %ld seconds and %ld microseconds\n", diff_sec, diff_usec);
}
```

```

        (long)end_time.tv_sec, (long)end_time.tv_usec);


// Calculate the duration
long seconds = end_time.tv_sec - start_time.tv_sec;
long microseconds = end_time.tv_usec - start_time.tv_usec;
if (microseconds < 0) {
    seconds -= 1;
    microseconds += 1000000;
}

// Print elapsed time
printf("Elapsed time: %ld seconds and %ld microseconds\n", seconds, microseconds);

return 0;
}

```

## 2. Explanation:


- ``gettimeofday(struct timeval *tv, struct timezone *tz)``:  

  - Captures the current time in ``tv``, where ``tv_sec`` contains seconds since the Epoch

(January 1, 1970), and ``tv_usec`` contains the microseconds.

- **Calculating the Duration:**
  - The elapsed time is computed by subtracting the start time from the end time. If ``microseconds`` of the end time is less than the ``microseconds`` of the start time, you adjust by borrowing 1 second.

## 3. Output Example:

yaml

 Copy code

```

Process started at: 1693585123 seconds and 123456 microseconds
Process ended at: 1693585126 seconds and 123789 microseconds
Elapsed time: 3 seconds and 333 microseconds

```

This code allows you to determine when a process started, when it ended, and how long it took to complete.










how can i know if a background process exited normally or abnormally



To determine if a background process exited normally or abnormally in C, you can use the `waitpid()` function along with macros like `WIFEXITED()`, `WEXITSTATUS()`, `WIFSIGNALED()`, and `WTERMSIG()` provided by the `<sys/wait.h>` library.

Here's how you can do it:

## 1. Starting a Background Process and Monitoring It

You can fork a process to run in the background and then use `waitpid()` in the parent process to check the exit status of the child process.

## 2. Example Code:

c Copy code

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

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

    if (pid < 0) {
        // Fork failed
        perror("Fork failed");
        exit(EXIT_FAILURE);
    } else if (pid == 0) {
        // Child process (Background process)
        printf("Background process (PID: %d) started\n", getpid());

        // Simulate some work, or replace this with an actual command
        sleep(2);

        // Exit normally with status 0 (change this to a non-zero value for testing)
        exit(0);
    } else {
        // Parent process
        int status;
        pid_t result = waitpid(pid, &status, 0);
```

```

        if (result < 0) {
            // waitpid failed
            perror("waitpid failed");
            exit(EXIT_FAILURE);
        }

        if (WIFEXITED(status)) {
            // Process exited normally
            int exit_status = WEXITSTATUS(status);
            printf("Background process (PID: %d) exited normally with status %d\n",
        } else if (WIFSIGNALED(status)) {
            // Process was terminated by a signal
            int signal_number = WTERMSIG(status);
            printf("Background process (PID: %d) was terminated by signal %d\n", pi
        }
    }

    return 0;
}

```

### 3. Explanation:

- `fork()`:
  - Creates a new process. If `fork()` returns `0`, it means the current process is the child. If it returns a positive value, it is the parent process, and the returned value is the PID of the child.
- `waitpid(pid_t pid, int *status, int options)`:
  - Waits for the specified process to change state. The `status` parameter will hold the exit status of the child process.
- `WIFEXITED(status)`:
  - Returns `true` if the child process exited normally (using `exit()` or returning from `main()`).
- `WEXITSTATUS(status)`:
  - Retrieves the exit status of the process if it exited normally.
- `WIFSIGNALED(status)`:

- Normal Exit:

SCSS

Copy code

```
Background process (PID: 12345) started
Background process (PID: 12345) exited normally with status 0
```

- Abnormal Exit (terminated by a signal):

SCSS

Copy code

```
Background process (PID: 12345) started
Background process (PID: 12345) was terminated by signal 9
```

This code allows you to determine whether a background process ended normally or was terminated by a signal, and you can also retrieve the specific exit code or signal number.

⏮️ 📄 ↺️ 👍 🗨️ ⚙️



how to convert relative paths into absolute paths  
like ../, ../../ etc



To convert relative paths like `../..` or `../..` into absolute paths in C, you can use the `realpath()` function. This function will resolve the relative paths to their canonical absolute paths by resolving symbolic links, relative path components (like `..` and `.`), and returning the absolute pathname.

### 1. Using `realpath()` to Convert Relative Paths to Absolute Paths:

Here's how you can use `realpath()` to convert relative paths into absolute paths:

C

Copy code

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



```

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

int main() {
    const char *relative_path = "../../some_folder/some_file.txt"; // Replace with
    char *absolute_path;

    // Convert relative path to absolute path
    absolute_path = realpath(relative_path, NULL);
    if (absolute_path != NULL) {
        printf("Absolute path: %s\n", absolute_path);
        free(absolute_path); // Remember to free the allocated memory
    } else {
        // Handle error if realpath fails
        perror("Error resolving path");
    }

    return 0;
}

```

```

return 0;
}

```

## 2. Explanation:

- `relative_path`: This is your relative path, which could be something like `../../`, `../`, or any other relative path.
- `realpath(relative_path, NULL)`: This function call converts the relative path into an absolute path. The second argument is `NULL`, which means `realpath()` will allocate memory for the absolute path, and you must `free()` it after use.
- **Error Handling:** If `realpath()` returns `NULL`, it indicates an error, and you can use `perror()` to print a descriptive error message.


## 3. Output Example:

If the relative path is `../../some_folder/some_file.txt` and assuming your current working directory is `/home/user/docs`, the output might be:

### 3. Output Example:

If the relative path is `../../some_folder/some_file.txt` and assuming your current working directory is `~/home/user/docs`, the output might be:

javascript

 Copy code

**Absolute path:** `/home/some_folder/some_file.txt`

### 4. Key Points:

- **Resolving ``.`` and `..``:** The ``.`` refers to the current directory, and `..`` refers to the parent directory. `realpath()` automatically resolves these components to their respective absolute directories.
- **Symbolic Links:** `realpath()` also resolves symbolic links to their target locations, so the final path is fully canonicalized.
- **Buffer Allocation:** By passing `NULL` as the second argument, `realpath()` handles buffer allocation internally, but you must free this buffer once you're done with it to avoid memory leaks.



how to get these informations for a process  
Process Status (R/R+/S/S+/Z)  
Process group  
Virtual Memory  
Executable path of process



To obtain information about a process, such as its status, process group, virtual memory usage, and executable path, you can access the `/proc` filesystem in Linux. The `/proc` directory contains subdirectories for each running process, named by their process IDs (PIDs), and each of these subdirectories contains files that provide various details about the process.

Here's how you can retrieve the requested information:

#### 1. Process Status

The process status (R/R+/S/S+/Z) can be found in the `/proc/[pid]/stat` or `/proc/[pid]/status` file.



#### 2. Process Group



## 2. Process Group

The process group ID can be found in the `/proc/[pid]/stat` or `/proc/[pid]/status` file.

## 3. Virtual Memory

The virtual memory size (in bytes) is also available in the `/proc/[pid]/status` or `/proc/[pid]/statm` file.

## 4. Executable Path

The executable path of the process can be found by reading the symbolic link `/proc/[pid]/exe`.

### Example Code:

Here's a sample C program that retrieves this information for a given process ID:

```
c Copy code

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

```
    sscanf(buffer, "%d", &state);
} else if (strcmp(buffer, "Tgid:") == 0) {
    // Get process group (process group ID)
    sscanf(buffer, "Tgid:%d", &process_group);
} else if (strcmp(buffer, "VmSize:") == 0) {
    // Get Virtual Memory Size
    sscanf(buffer, "VmSize:%d", &vm_size);
}
}
fclose(fp);

printf("Process Status: %d\n", state);
printf("Process Group: %d\n", process_group);
printf("Virtual Memory: %d KB\n", vm_size);

// 2. Get Executable Path
snprintf(path, sizeof(path), "/proc/%d/exe", pid);
ssize_t len = readlink(path, buffer, sizeof(buffer) - 1);
if (len != -1) {
    buffer[len] = '\0';
    printf("Executable Path: %s\n", buffer);
}
```



```

    if (len != -1) {
        buffer[len] = '\0';
        printf("Executable Path: %s\n", buffer);
    } else {
        perror("Error reading executable path");
    }
}

int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <pid>\n", argv[0]);
        exit(EXIT_FAILURE);
    }

    pid_t pid = atoi(argv[1]);
    get_process_info(pid);


    return 0;
}

```

## Output:

Assuming the process ID is 1234 and the information exists:

yaml

 Copy code

```

Process Status: S
Process Group: 1234
Virtual Memory: 54000 kB
Executable Path: /usr/bin/some_executable

```

## Notes:

- Ensure your program runs with sufficient permissions to access the `/proc` files of the process you're querying.
- The `/proc` filesystem is specific to Unix-like operating systems, particularly Linux.

how to get the details when i do reveal  
how to extract details (in shell , ls command with -l flag gives some details)



To extract and display detailed file information similar to the ``ls -l`` command in a custom shell implementation, you can follow these steps. This involves retrieving file details and formatting the output. Here's how you can achieve this:

### Steps to Extract and Display File Details:

1. **Retrieve File Metadata:** Use the ``stat()`` function to get file metadata. This function provides information such as file size, permissions, modification time, and more.
2. **Format Permissions:** Translate the file's mode (permissions) into a human-readable format (e.g., ``-rw-r--r--``).
3. **Get User and Group Names:** Use ``getpwuid()`` and ``getgrgid()`` to retrieve the username and group name.

and group name.

4. **Format Modification Time:** Convert the modification time into a human-readable format using ``strftime()``.
5. **Print the Information:** Combine all the information into a formatted output similar to ``ls -l``.

### Example Code:

Here's a C program that demonstrates how to extract and print file details:

```
c Copy code
#include <stdio.h>
#include <sys/stat.h>
#include <pwd.h>
#include <grp.h>
#include <time.h>
#include <unistd.h>

void print_file_details(const char *file_name) {
    struct stat st;
```

```

if (stat(file_name, &st) == -1) {
    perror("stat");
    return;
}

// Get file permissions
char permissions[11];
snprintf(permissions, sizeof(permissions), "%C%C%C%C%C%C%C%C%C",
        (S_ISDIR(st.st_mode)) ? 'd' : '-',
        (st.st_mode & S_IRUSR) ? 'r' : '-',
        (st.st_mode & S_IWUSR) ? 'w' : '-',
        (st.st_mode & S_IXUSR) ? 'x' : '-',
        (st.st_mode & S_IRGRP) ? 'r' : '-',
        (st.st_mode & S_IWGRP) ? 'w' : '-',
        (st.st_mode & S_IXGRP) ? 'x' : '-',
        (st.st_mode & S_IROTH) ? 'r' : '-',
        (st.st_mode & S_IWOTH) ? 'w' : '-',
        (st.st_mode & S_IXOTH) ? 'x' : '-');

// Get user and group information.
struct passwd *usr_det = getpwuid(st.st_uid);

```

```

struct passwd *usr_det = getpwuid(st.st_uid);
struct group *grp_det = getgrgid(st.st_gid);

if (usr_det == NULL || grp_det == NULL) {
    perror("getpwuid or getgrgid");
    return;
}

// Format the modification time
char time_str[25];
strftime(time_str, sizeof(time_str), "%b %d %H:%M", localtime(&st.st_mtime));

// Print the details
printf("%s %3ld %s %s %9ld %s %s\n",
        permissions,
        (long)st.st_nlink,
        usr_det->pw_name,
        grp_det->gr_name,
        (long)st.st_size,
        time_str,
        file_name);

```

```
int main(int argc, char *argv[]) {
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file_name>\n", argv[0]);
        return 1;
    }

    print_file_details(argv[1]);
    return 0;
}
```

### Explanation:

1. **Permissions Formatting:** The `permissions` array is formatted based on the file's mode bits. This provides a string like `-rw-r--r--`, where:

- `d` indicates a directory (if `S_ISDIR` is true).
- `r`, `w`, `x` indicate read, write, and execute permissions, respectively.

2. **User and Group Information:**



- `getpwuid(st.st_uid)` returns a `struct passwd` pointer with user information

```
}
```

### Explanation:

1. **Permissions Formatting:** The `permissions` array is formatted based on the file's mode bits. This provides a string like `-rw-r--r--`, where:

- `d` indicates a directory (if `S_ISDIR` is true).
- `r`, `w`, `x` indicate read, write, and execute permissions, respectively.

2. **User and Group Information:**

- `getpwuid(st.st_uid)` returns a `struct passwd` pointer with user information.
- `getgrgid(st.st_gid)` returns a `struct group` pointer with group information.

3. **Time Formatting:**

- `strftime()` converts the last modification time (`st.st_mtime`) into a readable format.

4. **Output:** The `printf` function combines all the pieces of information into a formatted string similar to the `ls -l` output.



how to search for a file in a directory (recursively)  
that search for a file in the given directory and in the subdirectories  
of the given directory as well  
you will be given the file name to be searched and the target  
directory where the file is to be searched



To perform a recursive search for a file in a given directory and its subdirectories in C, you can use the POSIX `dirent.h` library to navigate directories and the `stat()` function to check file details.

Here's a C program that demonstrates how to search for a file recursively within a specified directory:

### Code Example

c

Copy code

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



```
#include <string.h>
#include <dirent.h>
#include <sys/stat.h>
#include <unistd.h>

void search_file(const char *dir_path, const char *file_name) {
    DIR *dir;
    struct dirent *entry;
    struct stat file_stat;
    char path[1024];

    if ((dir = opendir(dir_path)) == NULL) {
        perror("opendir");
        return;
    }

    while ((entry = readdir(dir)) != NULL) {
        // Skip '.' and '..' entries
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..") == 0) {
            continue;
        }
        if (entry->d_name[0] != '.') {
            if (stat(path, &file_stat) == 0) {
                if (file_stat.st_ino == 0) {
                    printf("File found: %s\n", path);
                }
            }
        }
        if (entry->d_name[0] == '.') {
            if (entry->d_name[1] == '/') {
                if (stat(path, &file_stat) == 0) {
                    if (file_stat.st_ino == 0) {
                        printf("File found: %s\n", path);
                    }
                }
            }
        }
    }
}
```



```

        // Construct the path to the file
        snprintf(path, sizeof(path), "%s/%s", dir_path, entry->d_name);

        // Check if it's a directory
        if (entry->d_type == DT_DIR) {
            // Recursively search in subdirectories
            search_file(path, file_name);
        } else {
            // Check if the file matches
            if (strcmp(entry->d_name, file_name) == 0) {
                if (stat(path, &file_stat) == 0) {
                    printf("Found file: %s\n", path);
                    // You can print additional information about the file here if
                } else {
                    perror("stat");
                }
            }
        }
    }
}

```



```

        closedir(dir);
    }

    int main(int argc, char *argv[]) {
        if (argc != 3) {
            fprintf(stderr, "Usage: %s <directory> <file_name>\n", argv[0]);
            return EXIT_FAILURE;
        }

        const char *target_directory = argv[1];
        const char *file_name = argv[2];

        search_file(target_directory, file_name);

        return EXIT_SUCCESS;
    }

```

## Explanation



1. Open Directory:



## Explanation

### 1. Open Directory:

- ``opendir(dir_path)`` opens the specified directory. If it fails, an error is reported.

### 2. Read Directory Entries:

- ``readdir(dir)`` reads each entry in the directory. ``readdir`` returns ``NULL`` when there are no more entries.

### 3. Skip ``.`` and ``.``:

- These entries represent the current and parent directories, respectively, and should be skipped.

### 4. Construct Path:

- ``snprintf(path, sizeof(path), "%s/%s", dir_path, entry->d_name)`` constructs the full path of the entry.

### 5. Check Directory:

- If the entry is a directory (``entry->type == DT_DIR``), recursively call ``search_file()`` to search within the subdirectory.


### 7. Close Directory:

- ``closedir(dir)`` closes the directory stream.

## Usage

Compile and run the program with a directory and file name:

bash

 Copy code

```
gcc -o search search.c
./search /path/to/directory filename
```

## Notes

- **Path Buffer Size:** The ``path`` buffer size of 1024 is arbitrary. Adjust it based on the maximum path length you expect. POSIX specifies ``PATH_MAX``, but some systems might have different limits.
- **Error Handling:** Basic error handling is included. You might want to add more specific error handling depending on your needs.
- **Permissions:** Ensure your program has the necessary permissions to read the directories and