Process Termination (1)

- Terminating process notify its parent how it terminated.
- Normal termination
  - Pass an *exit status* as argument to exit( ) or _exit( ).
  - Exit status is converted into a termination status by the kernel when _exit( ) is finally called.
- Abnormal termination
  - The kernel generates a termination status to indicate the reason for the abnormal termination.
- The parent of the terminated process can obtain the termination status from either wait( ) or waitpid( ).
- *init* process becomes the parent process of any process whose parent terminates (inherited by init).

Process Termination (2)

- Zombie state
  - The kernel has to keep a certain amount of information consists of PID, the termination status of the process, and the amount of CPU time taken by the process.
  - The process that has terminated, but whose parent has not yet waited for it, is called a *zombie*.

exit

- #include <stdlib.h>

  void exit(int *status*);
  - Causes normal program termination and the value of *status* is returned to the parent.
    - Close and flush opened files.
    - Call functions registered in atexit( ).
    - Call _exit( )

atexit

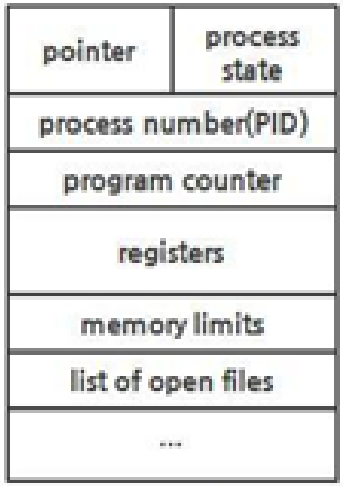
- #include <stdlib.h>

  int atexit(void (*_function_)(void));
  - Registers the given *function* to be called at normal program termination.
  - No arguments are passed.
  - Returns the value 0 if successful; otherwise the value -1 is returned.

_exit

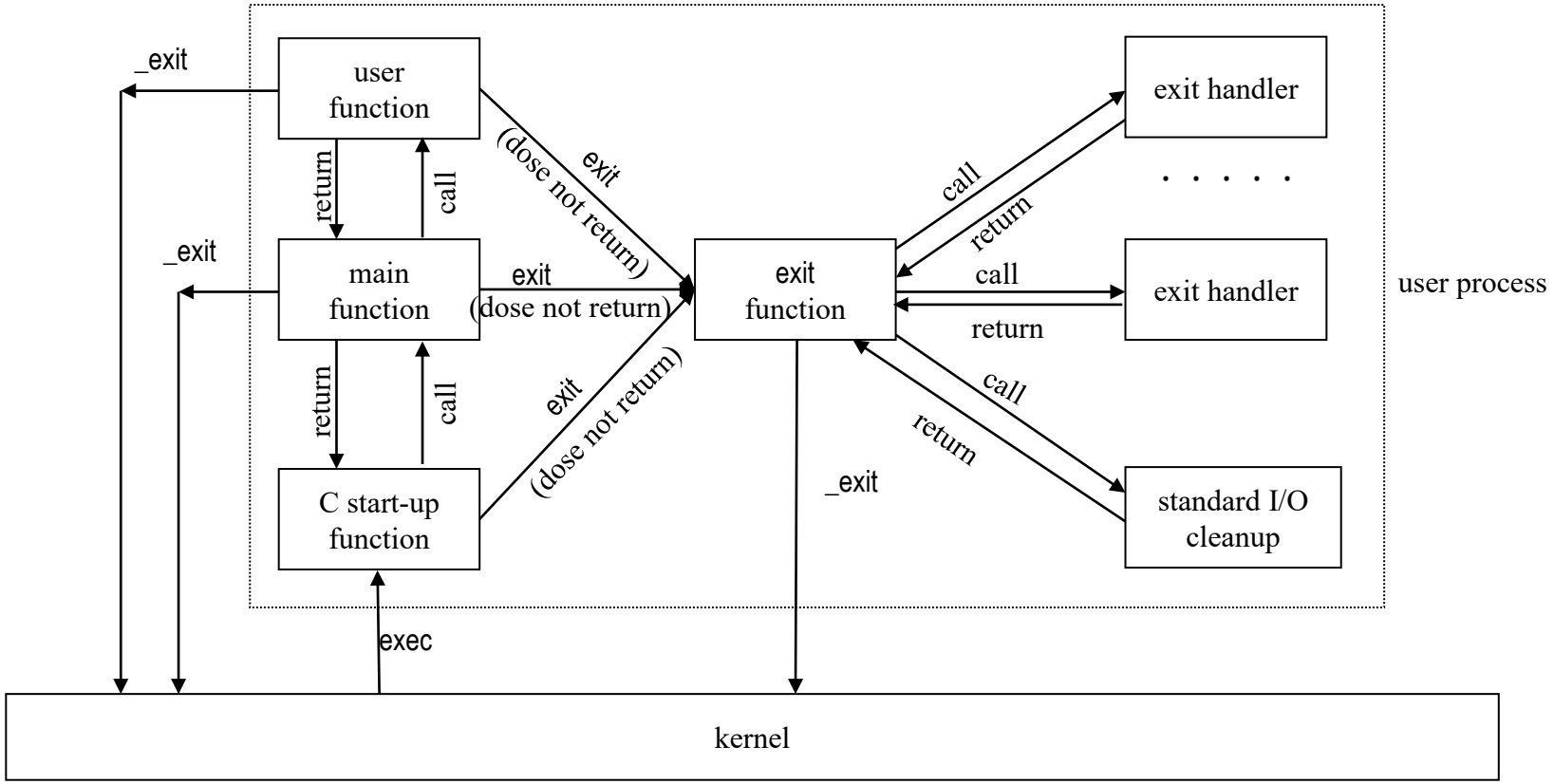
- #include <unistd.h>

  void _exit(int *status*);
  - Terminates the calling process immediately without performing some cleanup.
  - Deallocate everything but PCB entry.
  - *status* is returned to the parent process as the process's exit status.



PCB structure

Exit Functions



wait (1)

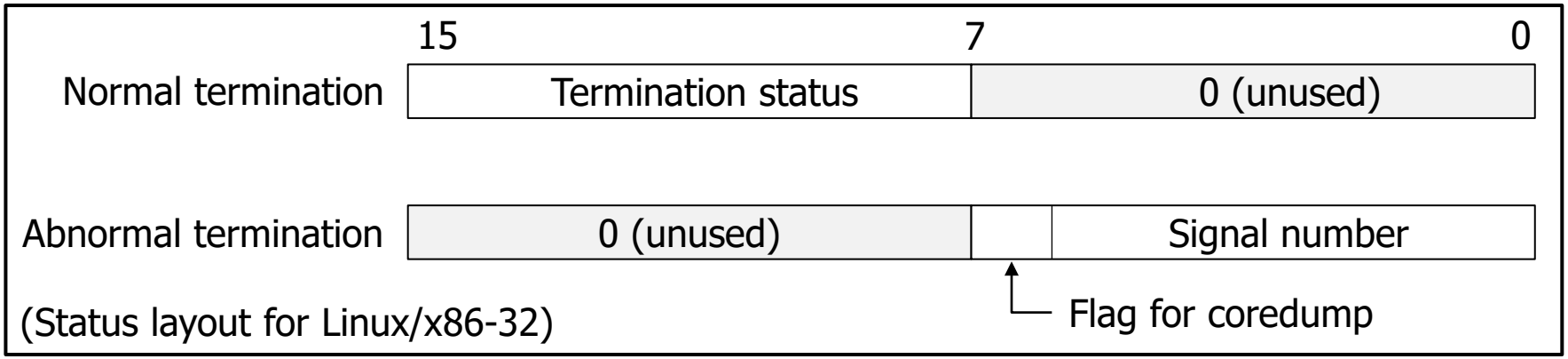
- #include <sys/types.h>

  #include <sys/wait.h>

  pid_t wait(int **status*)
  - Suspends execution of the current process until a child has exited, or until a signal is delivered whose action is to terminate the current process.
  - If the caller blocks and has multiple children, wait returns when one terminates.
  - We can always tell which child terminated because the process ID is returned by the function.

wait (2)

- *status*
  - If NULL, wait() ignores the status
  - If not NULL, wait() stores the status information (termination status) in the location pointed to by status.
  - Only the 16 low order bits are written.
  - If wait returns because a child process terminates, the low order 8 bits are 0, the high order 8 bits contains the exit value of the child process.
  - If the child exited because of a signal, the high order 8 bits are 0, and the low order 8 bits contain the signal number.



Macros for wait (1)

- WIFEXITED(*status*)
  - Returns true if the child exited normally.
- WEXITSTATUS(*status*)
  - Evaluates to *the least significant eight bits* of the return code of the child which terminated, which may have been set as the argument to a call to exit( ) or as the argument for a return.
  - This macro can only be evaluated if WIFEXITED returned non-zero.

Macros for wait (2)

- WIFSIGNALED(*status*)
  - Returns true if the child process exited *because of a signal*
- WTERMSIG(*status*)
  - Returns *the signal number* that caused the child process to terminate.
  - This macro can only be evaluated if WIFSIGNALED returned non-zero.

Example #3: wait

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid;
  int status;

  if ((pid = fork()) == 0) { /* child process */
  printf("I am a child\n");
  exit(123);
  }

  /* parent process */
  pid = wait(&status);
  printf("parent: child(pid = %d) is terminated with status (%d)\n", pid,
WEXITSTATUS(status));
  return 0;
}
```

```
./main
I am a child
parent: child(pid = 133) is terminated with status (123)
```

waitpid (1)

- #include <sys/types.h>

  #include <sys/wait.h>

  pid_t waitpid(pid_t *pid*, int **status*, int *options*);
  - waitpid wait for a specific child process, while wait wait for any child process
  - *pid* < -1
    - Wait for any child process whose process group ID is equal to the absolute value of *pid*.
  - *pid* == -1
    - Wait for any child process.
    - Same behavior which wait( ) exhibits.
  - *pid* == 0
    - Wait for any child process whose process group ID is equal to that of the calling process.

waitpid (2)

- *pid* > 0
  - Wait for the child whose process ID is equal to the value of *pid*.
- *options*
  - Zero or more of the following constants can be "OR"ed.
    - WNOHANG
      - » Return immediately if no child has exited.
    - WUNTRACED
      - » Also return for children which are stopped, and whose status has not been reported (because of signal).
    - WCONTINUED
      - » Also return for children which are continued because of SIGCONT.
- Return value
  - The process ID of the child which exited.
  - 0 or ECHILD if WNOHANG was used and no child was available.
  - -1 on error

Macros for waitpid

- WIFSTOPPED(*status*)
  - Returns true if the child process which caused the return is *currently stopped*.
  - This is only possible if the call was done using WUNTRACED.
- WSTOPSIG(*status*)
  - Returns *the signal number* which caused the child to stop.
  - This macro can only be evaluated if WIFSTOPPED returned non-zero.
- WIFCONTINUED(*status*)
  - Returns true if the child process is continued which caused by SIGCONT (Linux 2.6.10 or higher).

Example #4: waitpid (1)

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main() {
  pid_t pid;
  int status, exit_status;

  if ((pid = fork())<0)
   perror("fork failed");

  if (pid == 0) {
    printf ("Child %d sleeping... \n", getpid());
    sleep (4);
    exit (5) ;
  }
```

Example #4: waitpid (2)

```
 while (waitpid(pid, &status, WNOHANG) == 0) {
    printf ("Still waiting... \n");
    sleep (1);
  }

  if (WIFEXITED (status)) {
    exit_status = WEXITSTATUS (status);
    printf ("Exit status from %d was %d\n", pid, exit_status);
  }

  exit (0);
}
```

```
Child 413 sleeping...
Still waiting...
Still waiting...
Still waiting...
Still waiting...
Exit status from 413 was 5
```

```
Still waiting...
Child 226 sleeping...
Still waiting...
Still waiting...
Still waiting...
Exit status from 226 was 5
```

Exec Functions (1)

- #include <unistd.h>
  extern char **environ;

  int execl(const char *_pathname_, const char *_arg_, ...);
  int execv(const char *_pathname_, char *const _argv[ ]_);

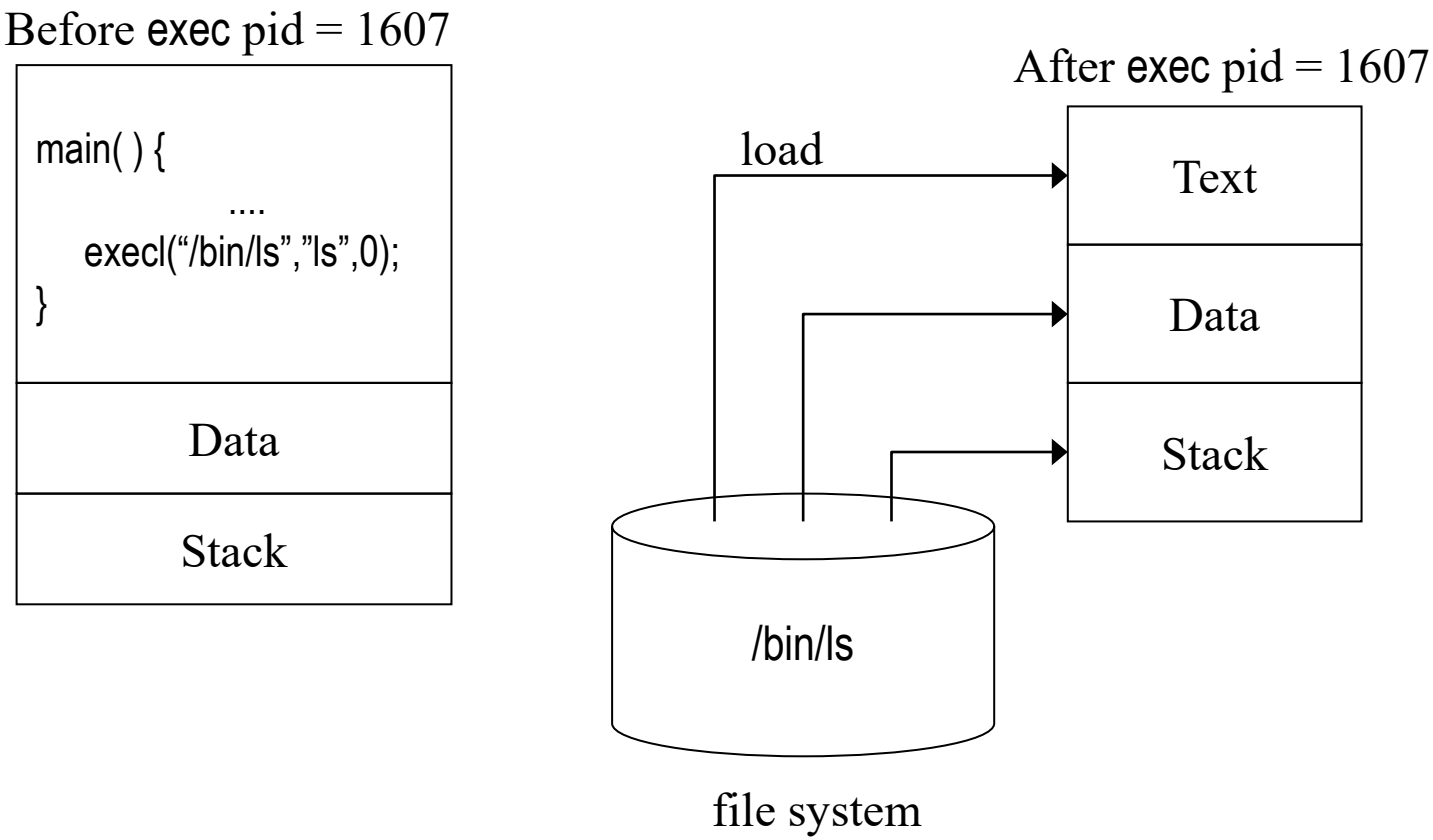
  int execle(const char *_pathname_, const char *_arg_ , ..., char * const _envp[ ]_);
  int execve (const char *_pathname_, char *const argv[ ], char *const _envp[ ]_);

  int execlp(const char *_filename_, const char *_arg_, ...);
  int execvp(const char *_filename_, char *const _argv[ ]_);

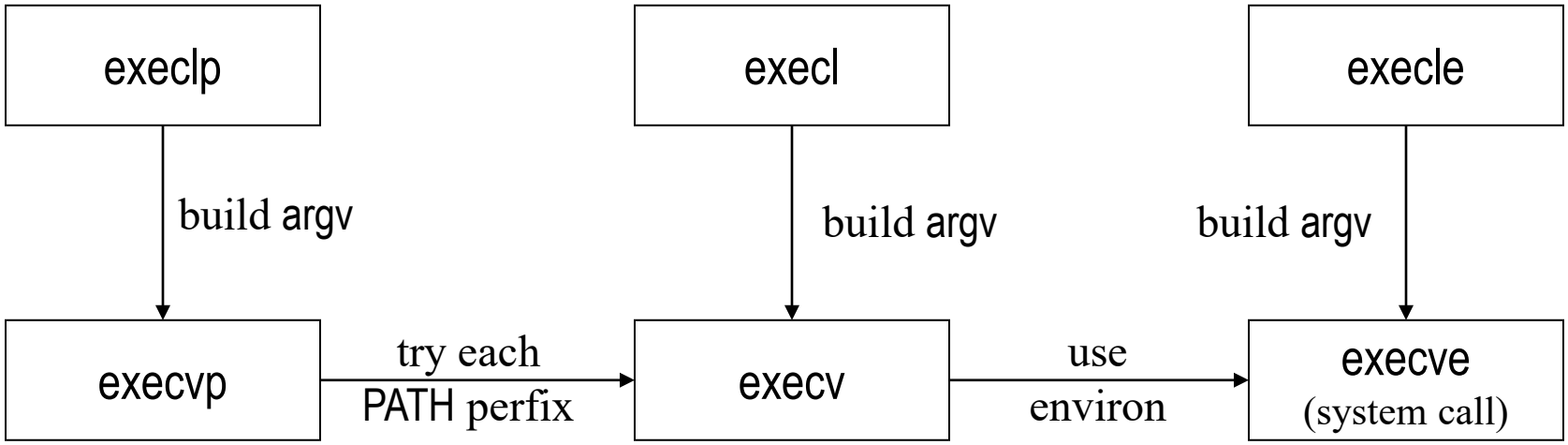
exec Functions (2)

- Execute program
  - When a process calls one of the exec functions, that process is completely replaced by the new program.
  - The new program starts executing at its main( ) function.
  - exec merely replaces the current process (text, data, heap, stack) with a brand-new program from disk.
- Only execve( ) is a system call.
  - execl, execv, execle, execlp, execvp are front-ends for the function execve( ).

exec Functions (3)



exec Functions (4)



exec Functions (5)

- Parameters of exec function
  - *pathname*(*filename*)
    - The initial argument for these functions is the *pathname* (*filename*) of a file which is to be executed.
    - execl, execv, execle, execve take a pathname argument.
    - execlp, execvp take a filename argument.
      - If *filename* contains a slash, it is taken as a pathname.
      - Otherwise, the executable file is searched for in the directories specified by the PATH environment variable.
      - If PATH isn't specified, the default path ":/bin:/usr/bin" is used.
      - If permission is denied for a file (the attempted execve( ) returned EACCES), these functions will continue searching the rest of the search path.

```
runner@repl.it:~$ echo $PATH
/home/runner/.apt/usr/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
```

exec Functions (6)

- Argument list
  - The const char *arg and subsequent ellipses in the execl, execlp, and execle functions can be thought of as *arg0*, *arg1*, ..., *argn.*
  - Together they describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program.
  - The first argument, by convention, should point to the file name associated with the file being executed.
  - The list of arguments must be terminated by a NULL pointer.
- Example

  execl("/bin/ls", "ls", "-al", 0);

exec Functions (7)

- Argument vector
  - execv, execve and execvp provide an array of pointers to null-terminated strings that represent the argument list available to the new program.
  - The first argument, by convention, should point to the file name associated with the file being executed.
  - The array of pointers must be terminated by a NULL pointer.
- Example

```
char *argv[3] = {“ls”, “-al”, 0};
execv(“/bin/ls”, argv);
```

exec Functions (8)

- Environment variable
  - execve and execle specifies the environment of the executed process by following the NULL pointer that terminates the list of arguments in the parameter list or the pointer to the *argv* array with an additional parameter.
  - This additional parameter is an array of pointers to null-terminated strings and must be terminated by a NULL pointer.
  - The other functions take the environment for the new process image from the external variable environ in the current process.
- Example

```
char *env[2] = {“TERM=vt100”, 0};
execle(“/bin/ls”, “ls”, 0, env);
```

exec Functions (9)

- Environment variable example

```
runner@repl.it:~$ printenv
LC_ALL=en_US.UTF-8
LD_LIBRARY_PATH=/home/runner/.apt/usr/lib/x86_64-linux-gnu:/home/runner/.apt/usr/lib/i386-linux-g
XDG_CONFIG_HOME=/config
LANG=en_US.UTF-8
DISPLAY=MAGIC
HOSTNAME=0320986151f8
OLDPWD=/home/runner/yeonseub
PWD=/home/runner
HOME=/home/runner
CPATH=
LIBRARY_PATH=/home/runner/.apt/usr/lib/x86_64-linux-gnu:/home/runner/.apt/usr/lib/i386-linux-gnu:
APT_OPTIONS=-o debug::nolocking=true -o dir::cache=/tmp/apt/cache -o dir::state=/tmp/apt/state -o
TERM=xterm-256color
SHLVL=1
CPPPATH=
PATH=/home/runner/.apt/usr/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PS1=\[\033[01;32m\]\u@repl.it\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$
PKG_CONFIG_PATH=/.apt/usr/lib/x86_64-linux-gnu/pkgconfig:/.apt/usr/lib/i386-linux-gnu/pkgconfig:/
LD_PRELOAD=/usr/local/lib/repl.so
INCLUDE_PATH=/home/runner/.apt/usr/include:/.apt/usr/include/x86_64-linux-gnu:
_=/usr/bin/printenv
```

exec Functions (10)

- Properties that the new program inherits.
  - Open files (depends on *close-on-exec* flag for each file descriptor)
  - pid, ppid, real-user(group)-id
  - Process group id, session id, controlling terminal
  - Current working directory, current root directory
  - Environment variables (except for execle( ), execve( ))
- Properties that the new program does not inherits.
  - Signals pending on the parent process are *cleared*.
  - Any signals set by the calling process are *reset* to their default behavior.
- Return value
  - On success, never returns. On error, -1 is returned.

Example #5: execlp

```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <sys/wait.h>

int main(void) {
  pid_t pid;
  int status;

  if ((pid = fork()) == 0) { /* child */
    execlp("ls", "ls", "-a", NULL);
  } else {
    wait(&status);
    printf("exit status of child = %d\n", WEXITSTATUS(status));
  }
}
```

```
./main
.   bar2    dir   foo    foo-sh  main.c   prog
..  bar-hl  file  foo-d  main    main-sh  test
exit status of child = 0
```

Thank you