Real-Time Programming
TI00AA55

Lecture 6 - 25.02.2015

Jarkko.Vuori@metropolia.fi

Process can “change” the program

- We have seen how process can create a new process using fork
  - The child process is executing the same program code as it’s parent
  - The child however can change the program it runs
    - This means that it can replace the inherited code with another program code
- A process can start a new program using function execXY
  - Usually this happens immediately after returning from fork
- It is possible of course for a parent to start a new program too
- The new program file (executable) is loaded from the disk and the execution starts from the start_up code (as you might expect)
- The process is still the same
  - The process id (and some other features of the process) are preserved
- There are several versions of execXY functions
  - The differences between them are how command line parameters and environment variables are passed to the new program
- Prototypes of different versions are on the next page

Functions execXY

- The prototypes of exec functions:

```
int execl(const char *path, const char *arg0, ... /*(char*) 0 */);
int execv(const char *path, char *const argv[]);
int execle(const char *path, const char *arg0, ... /*(char*) 0,
                    char *const envp[] */
int execve(const char *path, char *const argv[],
                    char *const envp[]);
int execlp(const char *filename, const char *arg0, ...
                    /*(char*) 0 */);
int execvp(const char *filename, char *const argv[]);
```

  - All functions return –1 if exec does not succeed.
  - Return value has no meaning in other cases, because exec function never returns.
- Only execve is system call
  - Other functions are library functions that finally call execve

Differences between exec functions

- The first four functions take the full path of the executable file as a parameter
- The last two take the filename and they use environment variable PATH to find the executable file
  - If the filename however contains character /, the filename is interpreted as a full path
- The explanations for additional letters at the end of the function name exec:
  - p path (environment variable PATH is used)
  - v vector (parameters are given as a vector (array))
  - l list (parameters are given as a list)
  - e environment variables are given as a list
- If additional character e is not at the end of the function name, it means that the new program continues to use the current environment variables of the process

Example

```
int main(void) {
   pid_t pid;
   int status;
   pid = fork();
   if (pid == 0) { //Child
      execl("./prgname", "prgname", "cmdlpar1",
            "cmdlpar2", (char*) 0);
      // Child never returns here,
      // for error checking
      perror("exec");
      exit(0);
   }
   //Parent
   wait(&status); //Return value of main function
                  // from prgname comes here
                  // if everything goes fine
                  // (macro WEXITSTATUS)
}
```

Program start

- To develop a better understanding what happens in exec and how the program starts, lets recall the memory layout of the program on the page 8 and the structure of the code segment of the program on page 17
- The memory area of the process is reconstructed in the following way:
  1. Text segment and initialized data segment is loaded from the disk
  2. Other segments are created by the kernel
  3. The command line parameters in the upper part of the memory area are constructed according the parameters passed for the exec function
  4. The process is put to the READY state
- When the process is scheduled to run the following happens
  - The execution starts from the start-up code
  - stdin, stdout and stderr are opened in startup code
  - Start-up code pushes command line parameters to the stack and environment variables to the global variable eviron to make them easily accessible for the main function
  - The main-function is called (from start-up code)
    - If start-up code had been written in C, the call would look as follows
      `exit(main(argc, argv));`

Program end

- We already know that a process can terminate with the system call `_exit` or with the library call `exit`
- It is not necessary that one of these calls is explicitly in the program
- A process can terminate when we simply return from the main function
  - We have just seen that when main function returns, it returns back to the caller (which in this case is the startup code)
  - The return value of the main function is passed to the caller
  - Startup code calls then the exit function and passes the return value received from the main function to the exit function as a parameter

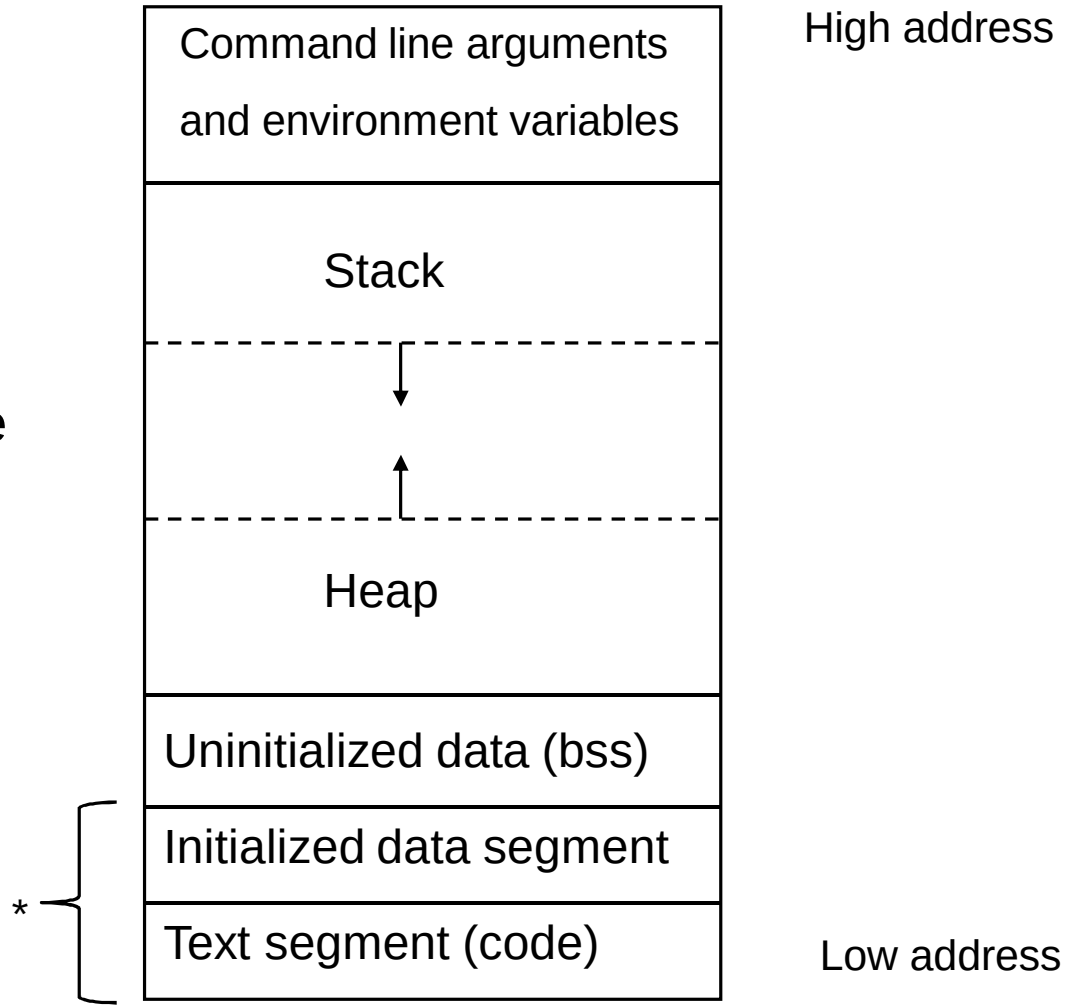
- A problem in the following program and how to fix it:

```
void main(void) {
    printf("\nOK");
   _exit(0);
}
```

Method 1: return 0;
Method 2: exit(0)

Memory layout of the program

- Only text segment and initialized data segment are stored on the disk (in the executable file)
- Function exec constructs other segments when the program is started



* This part is loaded from the disk

Parameters to program

Command line parameters

- Process that starts a new program with exec function can pass command line parameters for the new program
  - Exec constructs the program image in the memory and also a data structure from the command line parameters in the high address part of the image
  - Startup-code pushes the argc and argv onto the stack so that main function has easy access to this data
- Remark. It is guaranteed that argv[argc] is NULL

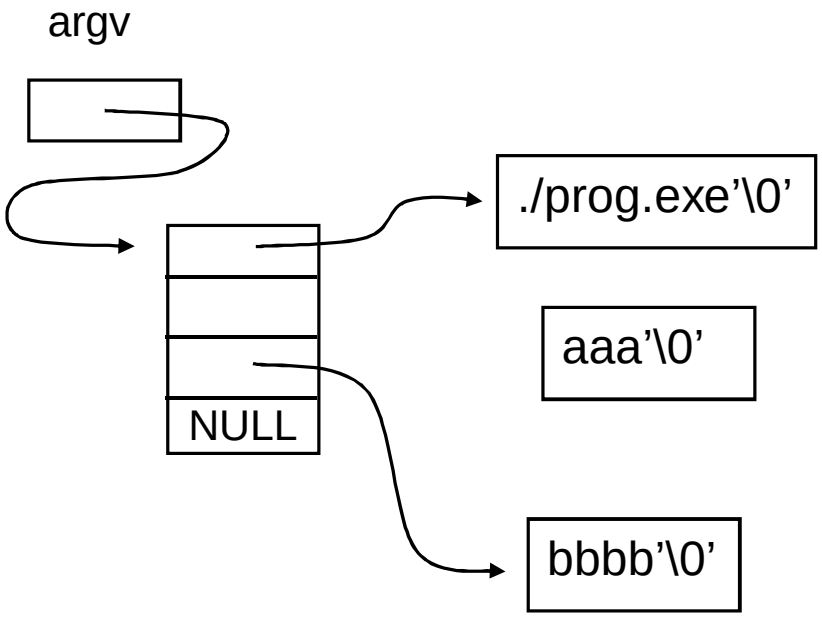
```
int main(void) {
  char *s;

  if ((s = getenv("HOME")) != NULL)
    printf("%s\n", s);
  else
    printf("No value\n");

  return 0;
}
```

Environment list

- The memory layout of environment variables is similar to the command line parameters but the access to environment variables is provided by the global variable environ:
  `extern char **environ;`
- Storage class specifier `extern` means that the variable is defined somewhere else than the current program file
- Note the terms:
  - Environment pointer (environ)
  - Environment list (an array of pointers)
  - Environment strings (NAME=“linux\0”)
- Functions getenv and putenv <stdlib>
  `char *getenv(const char *name);`
  returns NULL if not found
  `int putenv(const char *envstr);`
  returns 0, if OK, otherwise ≠0

The structure of command line parameters and environment variables

- Let's assume that the program has been started from the command line in the following way:
  …$ ./prog.exe aaa bbbb
- We also assume that the prototype of the main function is `int main(int argc, char **argv);`
- The memory layout of command line parameters is shown on the right

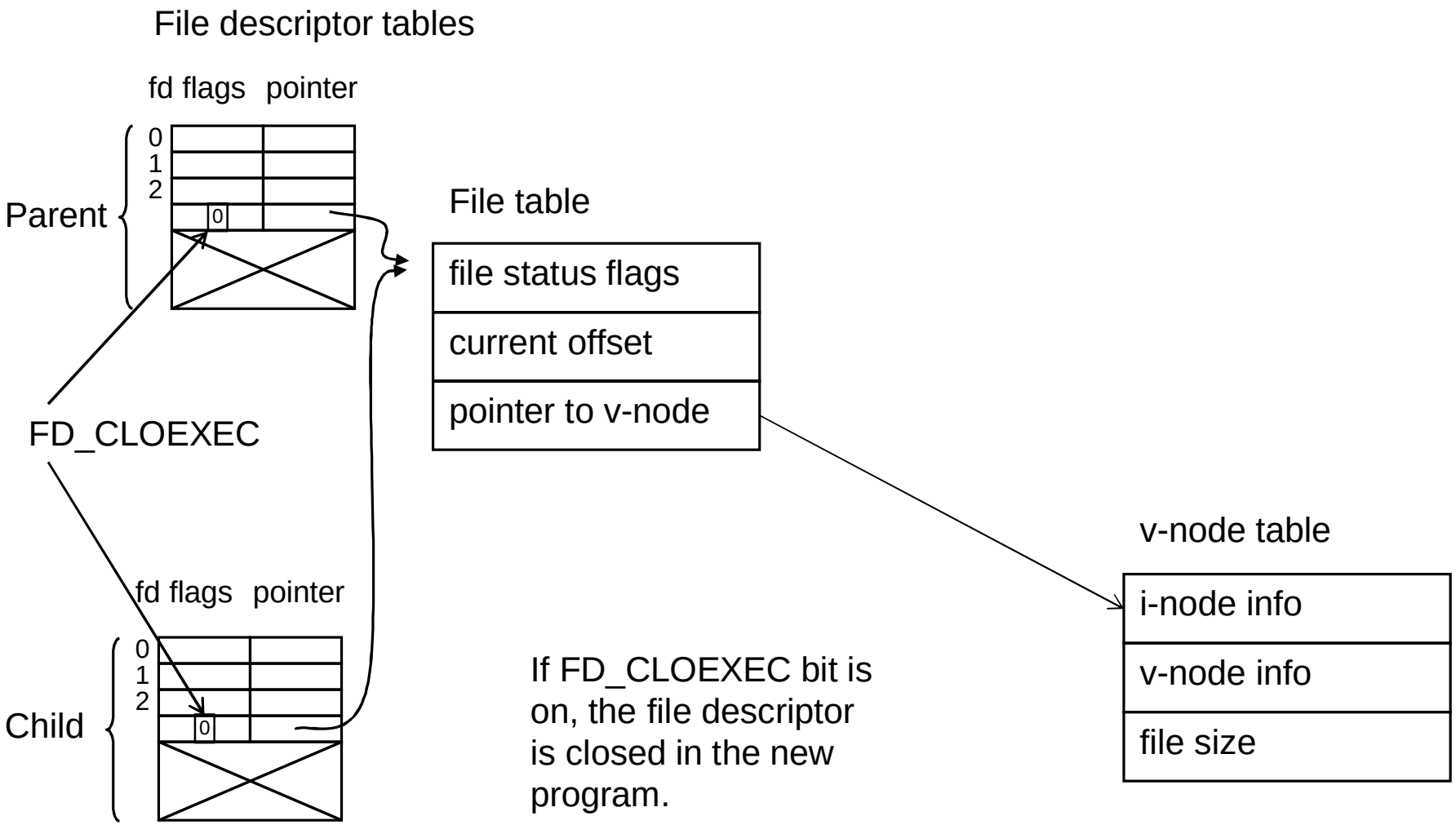


- The structure of the environment variables is very similar, but the global variable environ points to the array of pointers and the strings are like this example environment string PATH=/home/name'\0'

What does a new program inherit from the process

- We still remember that process does not change
- The following things are ”preserved” in exec:
  - Process id (and process id of parent)
  - Real user ID and real group ID
  - Process group ID
  - Current working directory
  - Root directory
  - File mode creation mask
  - File locks
  - Process signal mask
  - Pending signals
  - Time left until alarm clock
  - Resource limits
- These properties of the process are preserved although the program running in the process is changed
- **This is a different thing than what child inherits from the parent process**
- The new program can still use open file descriptors of the process if the close-on-exec file descriptor flag is not set. The default value for this bit is off meaning ”not close on exec”
  - See the next page
- The effective user id of the process can change, if set-user-id bit is set in the new executable file

Inheritance of open file descriptors



How the shell works

- The things, we have learned, also explain how the shell works when a program is started from the prompt
- Shell is like a server that is waiting for new commands from the client
  - The shell can respond to the new commands only if the previous commands allow it by putting their own service on the background (&)
  - It is possible that several processes are running under the control of the shell on the background and all the time new ones can be launched
  - After each command, the shell fetches the return status of every child process that has terminated during the wait of the command (in similar way than in the example on the page 26 in part 5)
- See a separate example (shell_own_1.c) that demonstrates the functionality of the shell
- See also another example (shell_own_2.c) that concentrates on how redirection of input and output works in the shell
- Remark 1. Notice that these things are also used in the implementation of the library function system
- Remark 2. Before forking and execing, the shell also splits the command line into separate strings

vfork

- Function vfork is especially meant for the situations where we fork a new process to run a new program in it
- The syntax of vfork is identical to fork

- The differences of vfork compared to fork are:
  - The data area of parent is not copied to the child. The child shares the data area of parent (as long as exec is not called)
  - This is a question of effectiveness. Why to copy data because the child is not going to use it because it calls exec that causes the whole data segment to be constructed from scratch
  - Function fork often works as effectively as vfork (in computers having MMU), because the so called copy-on-write feature is used
  - In addition to not copying data areas vfork guaranties that the child executes first until it calls exec (or exit)

exit-handlers

- It is possible to write exit handler functions for a process
  - Exit handler functions are executed automatically when the process is terminated with the function exit
- Exit-handlers are installed with the function atexit:
  <stdlib>
  `int atexit(void (*func)(void));`
  Return value is 0, if OK, otherwise it is ≠0
- Function is used to register functions that are called by the function exit, when process is terminated
- The functions are executed in the opposite order they are registered
- It is possible to register same function more than once
  - It is executed as many times as it has been registered
- The things are done in the following order:
  1. Exit handlers are executed first in the order mentioned above
  2. Then i/o-library buffers are flushed and buffers are deallocated
  3. Open files are closed

Example of exit-handlers

- Example

In the program we have lines	At process termination, the following functions are called
`void f1(void);`	`f2();`
`void f2(void);`	`f2();`
…	`f1();`
`atexit(f1);`	buffers are flushed
`atexit(f2);`	buffers are freed
`atexit(f2);`	files are closed

- The following example demonstrates the difference between return and exit

```
int main(void) {
    …
    f();
    …
    return 0; //return to the
              //startup-code
    //exit(0);//(or _exit) does not
              //return to startup-
              //code
}

void f(void) {
    return;    //returns to main
    //exit(1); //(or _exit) we never
               //return to main
}
```

How executable is build