Waiting for children processes

Orphans, zombies

Daemons

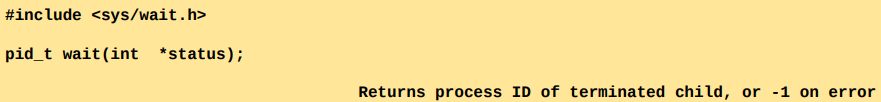
Monitoring Child Processes

In many application designs, a parent process need to know when one of its child processes changes state—when the child terminates or is stopped by a signal.

This lecture we will focus on two techniques used to monitor child processes: the wait() system call (and its variants) and the use of the SIGCHLD signal.

* Normal behavior is ignoring this signal

The wait() system Call



Caller of wait will be blocked until child returns.

The wait() system call waits for one of the children of the calling process to terminate and returns the termination status of that child in the buffer pointed to by status

If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, wait() returns immediately.

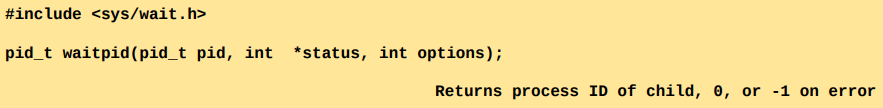
You don’t know which child died. You should check returned pid.

If status is not NULL , information about how the child terminated is returned in the integer to which status points.

Text

Description automatically generated with low confidence

The waitpid() System Call



If a parent process has created multiple children, it is not possible to wait() for the completion of a specific child; we can only wait for the next child that terminates.

If no child has yet terminated, wait() always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.

Using wait(), we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as SIGSTOP or SIGTTIN ) or when a stopped child is resumed by delivery of a SIGCONT signal.

waitpid ()

The return value and status arguments of waitpid() are the same as for wait().

The pid argument enables the selection of the child to be waited for, as follows:

* If pid is greater than 0, wait for the child whose process ID equals pid
* If pid equals 0, wait for any child in the same process group as the caller
* If pid is less than –1, wait for any child whose process group identifier equals the absolute value of pid.
* If pid equals –1, wait for any child. The call wait(&status) is equivalent to the call waitpid(1,&status,0).

Text

Description automatically generated

What’s going on?

The printWaitStatus() function is used in Listing 26-3. This program creates a child process that either loops continuously calling pause() or, if an integer command-line argument was supplied, exits immediately using this integer as the exit status.

In the meantime, the parent monitors the child via waitpid(), printing the returned status value and passing this value to printWaitStatus().

The parent exits when it detects that the child has either exited normally or been terminated by a signal.

Process Termination from a Signal Handler

In some circumstances, we may wish to have certain cleanup steps performed before a process terminates. For this purpose, we can arrange to have a handler catch such signals, perform the cleanup steps, and then terminate the process.

If we do this, we should bear in mind that the termination status of a process is available to its parent via wait() or waitpid().

For example, calling \_exit(EXIT\_SUCCESS) from the signal handler will make it appear to the parent process that the child terminated successfully

THERE IS NO WAY FOR THE PARENT TO KNOW WHICH CHILD HAS TERMINATED FOR WHICH REASON. WRITING SIGNAL HANDLER ON THE PARENT IS NOT VERY EFFICIENT. ONE WAY IS WRITING HANDLER FOR EACH CHILD.

If the child needs to inform the parent that it terminated because of a signal, then the child’s signal handler should first disestablish itself, and then raise the same signal once more. The signal handler would contain code such as the following:

Text, letter

Description automatically generated

SO THAT REST OF THE PROCESSES WILL BE AWARE THAT STH HAS HAPPENED DURING THE EXECUTION OF THE CODE.

The waitid() System Call

Like waitpid(), waitid() returns the status of child processes. However, waitid() provides extra functionality that is unavailable with waitpid()

Text, letter

Description automatically generated

Orphans and Zombies

The lifetimes of parent and child processes are usually not the same— either the parent outlives the child or vice verse. This raises two questions:

* Who becomes the parent of an orphaned child? The orphaned child is adopted by init, the ancestor of all processes, whose process ID is 1. In other words, after a child’s parent terminates, a call to getppid() will return the value 1 (This can be used as a way of determining if a child’s true parent is still alive)
* What happens to a child that terminates before its parent has had a chance to perform a wait()? The point here is that, although the child has finished its work, the parent should still be permitted to perform a wait() at some later time to determine how the child terminated. The kernel deals with this situation by turning the child into a zombie so that parent can go and ask “What happened to child?”. THOSE RESOURCES WILL STAY UNTIL PARENT PROCESS PERFORMS A waitpd OR waitid STATEMENT.

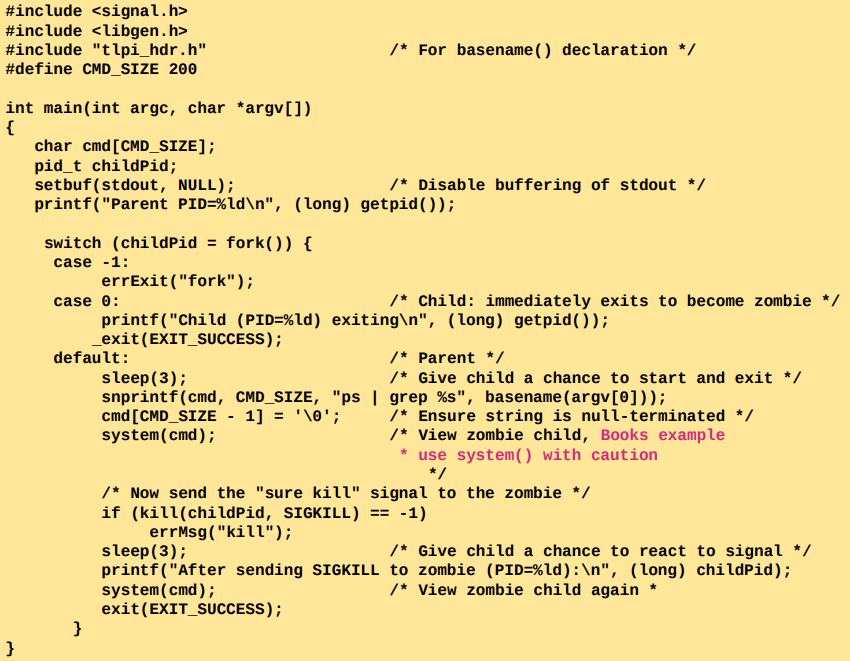
Zombies

Regarding zombies, UNIX systems imitate the movies—a zombie process can’t be killed by a signal, not even the (silver bullet) SIGKILL. This ensures that the parent can always eventually perform a wait()

When the parent does perform a wait(), the kernel removes the zombie, since the last remaining information about the child is no longer required. On the other hand, if the parent terminates without doing a wait(), then the init process adopts the child and automatically performs a wait(), thus removing the zombie process from the system.

If a parent creates a child, but fails to perform a wait(), then an entry for the zombie child will be maintained indefinitely in the kernel’s process table. If a large number of such zombie children are created, they will eventually fill the kernel process table, preventing the creation of new processes.

kill sinyaliyle yok edilmelerine rağmen, child processlerin terminationı tam handle edilmezse, childler çok fazla yer kaplayabilir. Kernel process tableda yok olmadıklarından memoryde resource olsa bile yeni process oluşturulamayabilir. wait edilmezse, init bunu fark edip process tableı temizleyene kadar yeni process oluşumunu engelleyebilirsin.



exiting doesn’t completely release all the resources. There are still zombies over there.

If you created a child, wait for it!!!!!!!!!!!!!!!!

system() is bad. It generates a terminal then runs the command inside that terminal. You generate 3 processes: terminal, inside terminal you run a system command, command that is generated.

The SIGCHLD Signal

The termination of a child process is an event that occurs asynchronously. A parent can’t predict when one of its child will terminate. We have already seen that the parent should use wait() (or similar) in order to prevent the accumulation of zombie children, and have looked at two ways in which this can be done:

* The parent can call wait(), or waitpid() without specifying the WNOHANG flag, in which case the call will block if a child has not already terminated.
* The parent can periodically perform a nonblocking check (a poll) for dead children via a call to waitpid() specifying the WNOHANG flag

Both of these approaches are inconvenient.

On the one hand, we may not want the parent to be blocked waiting for a child to terminate. Making repeated nonblocking waitpid() calls wastes CPU time and adds complexity to an application design.

Establishing a Handler for SIGCHLD

The SIGCHLD signal is sent to a parent process whenever one of its children terminates. By default, this signal is ignored

A common way of reaping dead child processes is to establish a handler for the SIGCHLD signal. This signal is delivered to a parent process whenever one of its children terminates, and optionally when a child is stopped by a signal.

Alternatively, but somewhat less portable, a process may elect to set the disposition of SIGCHLD to SIG\_IGN , in which case the status of terminated children is immediately discarded (and thus can’t later be retrieved by the parent), and the children don’t become zombies.

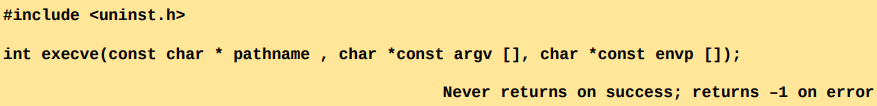
Executing a New Program: execve()

The execve() system call loads a new program into a process’s memory. During this operation, the old program is discarded, and the process’s stack, data, and heap are replaced by those of the new program.

The most frequent use of execve() is in the child produced by a fork(), although it is also occasionally used in applications without a preceding fork().

Various library functions, all with names beginning with exec, are layered on top of the execve() system call. Each of these functions provides a different interface to the same functionality. The loading of a new program by any of these calls is commonly referred to as an exec operation, or simply by the notation exec().

execve()



The pathname argument contains the pathname of the new program to be loaded into the process’s memory.

The argv argument specifies the command-line arguments to be passed to the new program. The value supplied for argv[0] corresponds to the command name.

The final argument, envp, specifies the environment list for the new program. The envp argument corresponds to the environ array of the new program; it is a NULL - terminated list of pointers to character strings of the form name=value

All the loaded program’s stuff will be inserted into the process. Everything except of the text (libraries…) of it.

Since it replaces the program that called it, a successful execve() never returns. We never need to check the return value from execve(); it will always be -1. The very fact that it returned informs us that an error occurred. Among the errors that may be returned in errno are the following:

* EACCES : The pathname argument doesn’t refer to a regular file, the file doesn’t have execute permission enabled, or one of the directory components of pathname is not searchable
* ENOENT : The file referred to by pathname doesn’t exist.
* ENOEXEC : The file referred to by pathname is marked as being executable, but it is not in a recognizable executable format (possibly, it is a script that doesn’t begin with a line specifying a script interpreter).
* ETXTBSY : The file referred to by pathname is open for writing by another process
* E2BIG : The total space required by the argument list and environment list exceeds the allowed maximum

Text, letter

Description automatically generated

envargs.c

BEST WAY IS MAKING THE COPY OF THE PROGRAM WITH fork() AND THEN USE execve().

USING execve IN YOUR PROGRAM DOESN’T MAKE SENSE. USE THAT PROGRAM THEN.

Text, letter

Description automatically generated

The exec() library functions

Table

Description automatically generated

Signals and exec()

During an exec(), the text of the existing process is discarded. This text may include signal handlers established by the calling program. Because the handlers disappear, the kernel resets the dispositions of all handled signals to SIG\_DFL.

The dispositions of all other signals are left unchanged by an exec().

Signals should not be blocked or ignored across an exec() of an arbitrary program ( Here, “arbitrary” means a program that we did not write ) It is acceptable to block or ignore signals when execing a program we have written or one with known behavior with respect to signals.

Executing a Shell Command: system()

The system() function allows the calling program to execute an arbitrary shell command.

Since one of our assignments will be a “shell” implementation, we will concentrate on how to implement system() using **fork(), exec(), wait(), and exit().**

A picture containing text, orange

Description automatically generated

The system() function creates a child process that invokes a shell to execute command.

system()

The principal advantages of system() are simplicity and convenience:

* We don’t need to handle the details of calling fork(), exec(), wait(), and exit().
* Error and signal handling are performed by system() on our behalf.
* Because system() uses the shell to execute command, all of the usual shell processing, substitutions, and redirections are performed on command before it is executed. This makes it easy to add an “execute a shell command” feature to an application.

The main cost of system() is inefficiency.

* Executing a command using system() requires the creation of at least two processes—one for the shell and one or more for the command(s) it executes—each of which performs an exec().
* If efficiency or speed is a requirement, it is preferable to use explicit fork() and exec() calls to execute the desired program.
  + fork
  + use the exec command to execute the function we want
  + wait for execution of that command to finish
  + exit

Text, letter

Description automatically generated

*Sample run:*

Text

Description automatically generated with medium confidence

CTRL+Z puts the process to the background, fg will take it to foreground.

**!! Important !!**

Set-user-ID and set-group-ID programs should never use system() while operating under the program’s privileged identifier.

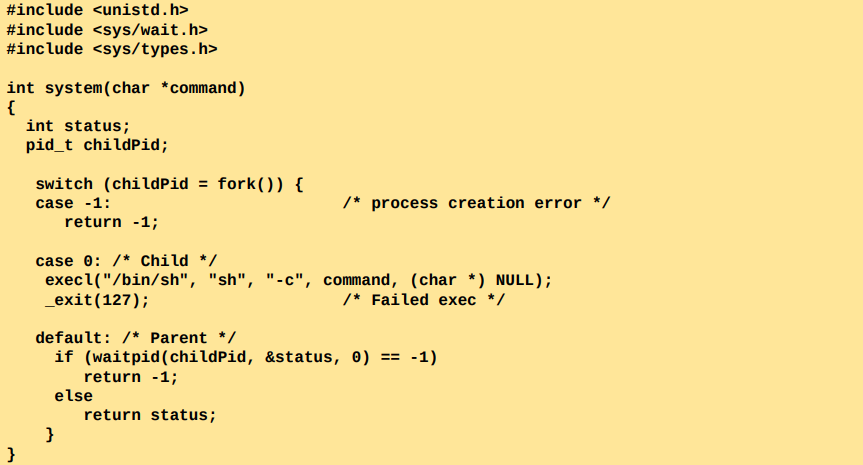
Even when such programs don’t allow the user to specify the text of the command to be executed, the shell’s reliance on various environment variables to control its operation means that the use of system() inevitably opens the door for a system security breach.

Implementing system()

To implement system(), we need to use fork() to create a child that then does an execl() to implement a command like



To collect the status of the child created by system(), we use a waitpid() call that specifies the child’s process ID.



Using wait() would not suffice, as wait() waits for any child, could accidentally collect the status of some other child created by the calling process.

Treating signals correctly inside system()

What adds complexity to the implementation of system() is the correct treatment with signals.

The first signal to consider is SIGCHLD . Suppose that the program calling system() is also directly creating children, and has established a handler for SIGCHLD that performs its own wait(). In this situation, when a SIGCHLD signal is generated by the termination of the child created by system(), it is possible that the signal handler of the main program will be invoked—and collect the child’s status—before system() has a chance to call waitpid().

Therefore, system() must block delivery of SIGCHLD while it is executing

The other signals to consider are those generated by the terminal interrupt (usually Control-C) and quit (usually Control-\) characters, SIGINT and SIGQUIT , respectively

SIGINT and SIGQUIT should be ignored in the calling process while the command is being executed (while the execution of other processes inside the child). These signals might cause the parent to terminate so whatever you execute on child will be orphaned. In order to make sure no orphan processes around, before executing sth from a child, it is a good programming practice just to block or ignore SIGINT or SIGQUIT signals before executing those things.

Text, letter

Description automatically generated

Text, letter

Description automatically generated

If you just wanna make sure child is doing execution properly, it is good idea to ignore SIGINT and SIGQUIT

Block all the child related signals, ignore the SIGINT and SIGQUIT signals.

Daemons

A daemon is a process with the following characteristics:

* It is long-lived. Often, a daemon is created at system startup and runs until the system is shut down.
* It runs in the background and has no controlling terminal. The lack of a controlling terminal ensures that the kernel never automatically generates any job-control or terminal-related signals (such as SIGINT , SIGTSTP , and SIGHUP ) for a daemon.
* It is a convention (not universally observed) that daemons have names ending with the letter d.

Daemon shouldn’t be related with terminal. Some signals should also be removed from a daemon.

Creating a Daemon

To become a daemon, a program performs the following steps:

* Perform a fork(), after which the parent exits and the child continues.
  + Daemon should be parentless.
  + Daemon shouldn’t write on stdout and it shouldn’t get input from stdin.
  + Daemon should write to device or get input from device.
  + So stdout, stdin, and also stderr should be closed for daemon.
* The child process calls setsid() to start a new session and free itself of any association with a controlling terminal
  + Daemon shouldn’t be related with terminal and any other processes other than init.
  + Daemon should remove the dependencies of the signals inside itself.
  + setsid() 🡪 Ben çalıştırıldığım ortamda en yüksek pid’ye sahip adam olmak istiyorum diyoruz. En yüksek pid’yi alırız, ama tabii ki initi alamayız. Böylece çalıştırıldığı terminal tarafından yok edilemeyecek.
* If the daemon might later open a terminal device, then we must take steps to ensure that the device does not become the controlling terminal
* Clear the process umask to ensure that, when the daemon creates files and directories, they have the requested permissions.
* Change the process’s current working directory, typically to the root directory
* Close all open file descriptors that the daemon has inherited from its parent
* After having closed file descriptors 0, 1, and 2, a daemon normally opens /dev/null and uses dup2() (or similar) to make all those descriptors refer to this device

Kontrolü terminalde olmayan, stdin stdoutu ekrana göstermeyen, dev/null üzerinden iletişim kuran, arka planda çalışan ve uzun süre çalışacak bir program.

Text, letter

Description automatically generated

Text

Description automatically generatedGraphical user interface, text

Description automatically generated

terminalin dışında iş yapabildiğimizi kendimize gösterme isteği

terminalin üzerine çıkma isteği



Up to here, we have created a program that doesn’t depend on any processes, its current working directory is root directory, its stdin stdout stderr are disabled and go to dev/null directory. Now we are program running in the background in kernel space with no dependencies. Communication with us will only be through dev/null.

Daemons perform specific tasks, such as providing a network login facility or serving web pages. To become a daemon, a program performs a standard sequence of steps, including calls to fork() and setsid().

Where appropriate, daemons should correctly handle the arrival of the SIGTERM and SIGHUP signals. The SIGTERM signal should result in an orderly shutdown of the daemon, while the SIGHUP signal provides a way to trigger the daemon to reinitialize itself by rereading its configuration file and reopening any log files it may be using.

Özellikle öldürmediğin sürece daemon ölmez.

Sadece SIGKILL hariç diğer bütün sinyalleri initialize edip yok edersin. Daemon onları ignore eder.