Processes

CISC 3350 — CUNY Brooklyn College
Lecture Notes

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Processes

Programs are files passively stored on disk and contain **binary code**, which are machine language instructions that can run on the computer. Programs that are large and significant are referred to as **applications**.

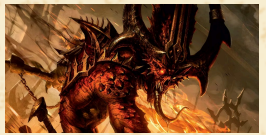
Once the binary code of a program is copied to memory in order to start executing, we call such code a **process**.

- That is, a process is an instance of a program that is executing. A program can have several process instances (since you could run the same program several times simultaneously. E.g.: running 2 instances of the `cd` command in two Linux terminal windows.)
- The notion of process also includes the memory sections associated with it, the files that it opens, the user who ran the program, and the threads that the process created. More on threads in the next topic.
- Processes are sometimes also referred to as **jobs**, and so both terms can be used interchangeably.

Every process has a unique identifier: its **process ID** (or `pid`, which is of the type: `pid_t`.)

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Processes



By thegamer.com

The **OS Scheduler**, also called the **idle process**, has the `pid` of `0`. This process runs when no other processes are currently running on the CPU, and makes the decision of what process should run next on the CPU.

The process scheduler also launches a process called **`init`** (or **`systemd`** in newer Linux versions) with the `pid` of `1`, which runs throughout the entire execution of the OS. **`init`** creates all the other processes on the computer, thereby becoming the Great Parent of all processes.

- The Scheduler searches for the code of `init` in one of the following locations, in the given order:
  - `/sbin/init`: the preferred and most likely location for the `init` program.
  - `/etc/init`: another likely choice.
  - `/bin/init`: a fallback choice.
  - `/bin/sh`: last choice if the kernel fails to find an init process.
- If `init` is found, it initializes the system, starts various vital background services (= **daemons**), and launches a login process. Otherwise, if all the 4 `init` programs aren't found or fail to launch, the Linux kernel shuts the system down.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Processes

By default, the kernel imposes a max `pid` value of 32768 (signed 16-bit), and increments them linearly, reusing `pid` values after it wraps around from the top.

- Example: `pid` = `2`, `3`, `4`, …, `32766`, `32767`, `32768`, `2`, `3`, `4`, etc. [`0` = idle and `1` = `init` are running.]

The Process Hierarchy:

- A process that **spawns** (= creates) a new process is called the **parent**; the new process is called the **child** and keeps a record of its parent process ID (`ppid`).
- Each process is owned by a **user** and a **group**, which is used to decide what resources the process has permission to use.
- Each child process inherits its parent's user and group owners.
- Each process is also a part of a **process group**.
  - For example, in the following pipelined Linux command, both commands belong to the same process group:
    ```
    $ ls | less
    ```
  - This makes it easy to send signals to an entire pipeline, as well as child processes.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Obtaining the `pid` and `ppid`

Name: `getpid()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: Return the process's own `pid` number.		

What libraries you must include:
`#include <sys/types.h>`
`#include <unistd.h> // Defines getpid().`		

Syntax: `pid_t getpid (void);`		
Description of arguments: – This syscall doesn't receive any arguments –		
What type it returns: `pid_t`	**On success:** the `pid` (non-negative integer) of the process.	
	On failure: – This syscall never fails –	
	If failure, does it set `errno`? No	
Quick example: `printf ("My pid = %jd.\n", (intmax_t) getpid ());`		
Other variations: `getppid()`		

Above, `intmax_t` is a C/C++ integer data type guaranteed to be capable of storing any signed integer.
It must be paired with the `printf()` variable specifier `%j`, which prints an `intmax_t` variable.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Obtaining the `pid` and `ppid`

<table>
<tr><td>Name: <code>getppid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: Return the <code>pid</code> number of the process's parent process.</td></tr>
<tr><td colspan="3">What libraries you must include:
<pre>#include <sys/types.h>
#include <unistd.h> // Defines getppid().</pre></td></tr>
<tr><td colspan="3">Syntax: <code>pid_t getppid (void);</code></td></tr>
<tr><td colspan="3">Description of arguments: – This syscall doesn't receive any arguments –</td></tr>
<tr><td rowspan="3">What type it returns: <code>pid_t</code></td><td colspan="2">On success: the <code>pid</code> (non-negative integer) of the process's parent.</td></tr>
<tr><td colspan="2">On failure: – This syscall never fails –</td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="3">Quick example:
<pre>printf ("Parent's pid = %jd.\n", (intmax_t) getppid ());</pre></td></tr>
<tr><td colspan="3">Other variations: <code>getpid()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Obtaining the `pid` and `ppid`

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Running a New Process

In Unix/Linux, loading and executing a program is separate from creating a new process!

The creation is called forking and is provided by the `fork()` syscall. The execution functionality is provided by the `exec()` family of calls. In short, a new process is executed by calling `exec()` after `fork()`.

`fork()` creates a new process called the **child**, which initially is a near-duplicate of the **parent** process (= the process that called `fork()`.) Then, `exec()` loads a new program image (code, variables, etc.) into the child process's memory.

The parent/child processes are identical in every way except for:

- The return value from `fork()` is always `0` inside the child process, and a non-zero integer inside the parent process.
- Resource statistics are reset to zero in the child.
- Any pending signals (e.g., interrupts) are cleared and not inherited by the child.
- Any acquired file locks are not inherited by the child.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The `fork()` System Call

<table>
<tr><td colspan="2">Name: <code>fork()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Creates a child process with identical code as the parent, and returns the <code>pid</code> of that created child.</td></tr>
<tr><td colspan="4">What libraries you must include:
<code>#include <sys/types.h></code>
<code>#include <unistd.h> // Defines fork().</code></td></tr>
<tr><td colspan="4">Syntax: <code>pid_t fork (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – This syscall doesn't receive any arguments –</td></tr>
<tr><td rowspan="3">What type it returns: <code>pid_t</code></td><td colspan="3">On success: the <code>pid</code> (non-negative integer) of the newly-created child.</td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<code>pid_t child = fork();</code>
<code>if (child == -1)</code>
<code>   /* Handle Error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The exec Family of Calls

- A family of `exec` functions are built on and internally invoke a single syscall (`execve()`). Here is one family member:

```
int execl (const char *path, const char *arg, ...);
```

- It loads the code of the program whose name is given by `path` into main memory. The argument `arg` should contain the name of the new child program.
- The ellipsis (`...`) represents variadic command-line arguments to the child program, which must end with `NULL`. Example:

```
int ret = execl ("/bin/vim", "vim", "text.txt", NULL); // Same as typing: vim text.txt.
if (ret == -1) // Error checks.
{
    perror ("execl");
    exit (EXIT_FAILURE);
}
```

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The `exec` Family of Calls

- Normally, `execl()` **doesn't** return - a successful invocation ends by jumping to the 1st code line of the new program (since this entire, parent-resembling code is replaced by the new program's instructions.)
- While certain attributes of the process are lost (like pending signals), the `pid`, `ppid`, priority, and owning user/group are all preserved and are accessible by the child.
- Open files are inherited as well (with the same `fd`s), but this may not be desired. One usually closes files before `exec` by using the `O_CLOEXEC` flag when calling `open()` to open these files.

Besides `execve()`, the other 5 members of the `exec` family are functions, not system calls.

In those functions' names, the '`l`' and '`v`' endings signify a **list** or a **vector** (array) of command-line arguments that are passed to the child, respectively.

The '`p`' ending denotes a full path search (just include the filename in `file`), and '`e`' denotes that a new environment is supplied for the new process.

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The exec Family of Calls

<table>
<tr><td colspan="2">Name: execve()</td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Replaces the code of the program that calls execve() with a new code at location path, and passes the arguments argv to that invoked program.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines execve().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int execve (const char *path, char *const argv[], char *const envp[]);</code></td></tr>
<tr><td colspan="4">Description of arguments: path: the location on disk where the code of the program you want to call is stored
argv: an array of command-line argument strings that will be passed to the new program
envp: an array of pointers to strings of the form key=value, representing the environment of the new program.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: int</td><td colspan="2">On success: It doesn't return: instead, the current program is replaced by the new program.</td></tr>
<tr><td colspan="2">On failure: -1</td></tr>
<tr><td colspan="2">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="4">Quick example:
<code>execve ("/bin/ls", newargv, newenviron);
perror ("execve"); /* execve() returns only on error */</code></td></tr>
<tr><td colspan="4">Other variations: execv(), execvp(), execle(), execl(), execlp()</td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The exec Family of Calls

<table>
<tr><td>Name: execv()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: Replaces the code of the program that calls it with a new code at location path, and passes the arguments argv to that invoked program.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <unistd.h> // Defines execv().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int execv (const char *path, char *const argv[]);</code></td></tr>
<tr><td colspan="3">Description of arguments: path: the location on disk where the code of the program you want to call is stored
argv: an array of command-line argument strings that will be passed to the new program</td></tr>
<tr><td rowspan="3">What type it returns: int</td><td colspan="2">On success: It doesn't return: instead, the current program is replaced by the new program.</td></tr>
<tr><td colspan="2">On failure: -1</td></tr>
<tr><td colspan="2">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="3">Quick example:
<pre>execv ("/bin/ls", newargv);
perror ("execv"); /* execv() returns only on error */</pre></td></tr>
<tr><td colspan="3">Other variations: execve(), execvp(), execle(), execl(), execlp()</td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The exec Family of Calls

Name: `execvp()`	**System call or function:** Function	**Links:** Online Manual | Course Packet
What it does: Replaces the code of the program that calls it with a new code in the file `file`, and passes the arguments `argv` to that invoked program. The OS will search for this file at the locations indicated in the `PATH` environmental variable, and, otherwise, in the current folder.		
What libraries you must include: `#include <unistd.h> // Defines execvp().`		
Syntax: `int execvp (const char *file, char *const argv[]);`		
Description of arguments: `file`: the name of the file containing the program to run `argv`: an array of command-line argument strings that will be passed to the new program		
What type it returns: `int`	**On success:** It doesn't return: instead, the current program is replaced by the new program.	
	On failure: `-1`	
	If failure, does it set `errno`? Yes	
Quick example: `execvp ("my_uniq", newargv);` `perror ("execvp"); /* execvp() returns only on error */`		
Other variations: `execve()`, `execv()`, `execle()`, `execl()`, `execlp()`		

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The exec Family of Calls

<table>
<tr><td colspan="2">Name: execle()</td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Replaces the code of the program that calls it with a new code at location path, and passes the arguments arg to that invoked program.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines execle().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int execle (const char *path, char *const arg, ..., char *const envp[]);</code></td></tr>
<tr><td colspan="4">Description of arguments: path: the location on disk where the code of the program you want to call is stored
arg: a list of command-line argument strings that will be passed to the new program
envp: an array of pointers to strings of the form key=value, representing the environment of the new program.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: int</td><td colspan="2">On success: It doesn't return: instead, the current program is replaced by the new program.</td></tr>
<tr><td colspan="2">On failure: -1</td></tr>
<tr><td colspan="2">If failure, does it set errno? Yes</td></tr>
<tr><td colspan="4">Quick example:
<code>execle ("/bin/ls", "ls", "~/cisc3350-hw", NULL, newenviron);
perror ("execle"); /* execle() returns only on error */</code></td></tr>
<tr><td colspan="4">Other variations: execve(), execv(), execvp(), execl(), execlp()</td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The exec Family of Calls

<table>
<tr><td>Name: <code>execl()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: Replaces the code of the program that calls it with a new code at location <code>path</code>, and passes the arguments <code>argv</code> to that invoked program.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <unistd.h> // Defines execl().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int execl (const char *path, char *const arg, ...);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>path</code>: the location on disk where the code of the program you want to call is stored
<code>arg</code>: a list of command-line argument strings that will be passed to the new program</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: It doesn't return: instead, the current program is replaced by the new program.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:
<pre>execl ("/bin/ls", "ls", "~/cisc3350-hw", NULL);
perror ("execl"); /* execl() returns only on error */</pre></td></tr>
<tr><td colspan="3">Other variations: <code>execve()</code>, <code>execv()</code>, <code>execvp()</code>, <code>execle()</code>, <code>execlp()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

The exec Family of Calls

<table>
<tr><td colspan="2">Name: <code>execlp()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: Replaces the code of the program that calls it with a new code in the file <code>file</code>, and passes the arguments <code>argv</code> to that invoked program. The OS will search for this file at the locations indicated in the <code>PATH</code> environmental variable, and, otherwise, in the current folder.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines execlp().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int execlp (const char *file, char *const arg, ...);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>file</code>: the name of the file containing the program to run
<code>arg</code>: a list of command-line argument strings that will be passed to the new program</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: It doesn't return: instead, the current program is replaced by the new program.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<code>execlp ("my_uniq", "input.txt", "output.txt", NULL);
perror ("execlp"); /* execlp() returns only on error */</code></td></tr>
<tr><td colspan="4">Other variations: <code>execve()</code>, <code>execv()</code>, <code>execvp()</code>, <code>execle()</code>, <code>execl()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Examples

Example of using `fork()`:

```
pid_t pid;

pid = fork ();

if (pid > 0)
    print ("I am the parent of pid=%d!\n", pid);
else if (pid == 0)
    print ("I am the child with pid=%d!\n", pid);
else if (pid == -1)
    perror ("fork");
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Examples

Example of using both `fork()` and `execv()`:

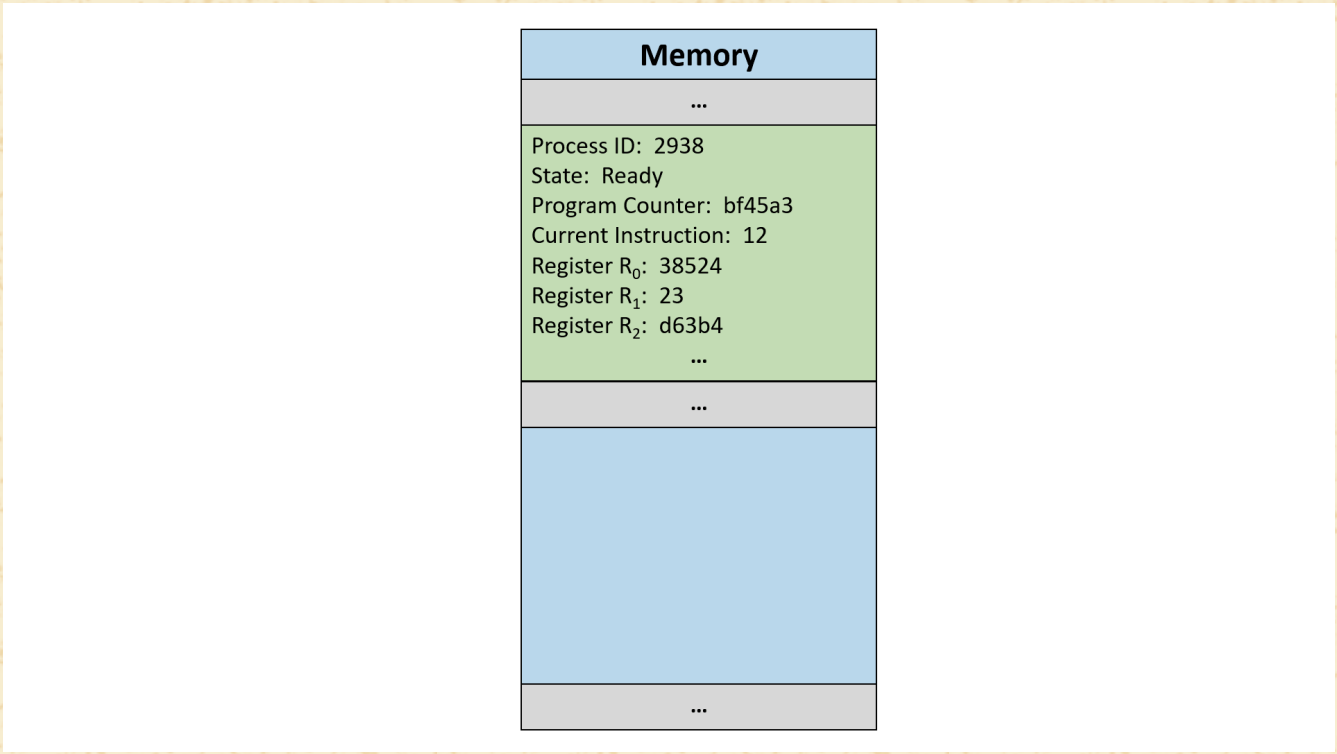
```
pid_t pid = fork ();

if (pid == -1)
    perror ("fork");

if (pid == 0) /* the child: */
{
    const char * args[] = {"ls", NULL};
    int ret = execv ("/bin/ls", args);
    perror ("execv");
    exit (EXIT_FAILURE);
}
```

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

fork() vs. exec()



Creating a New Process: `fork()` vs. `exec()`. Miriam Briskman, CC BY-NC 4.0.

Note: click the GIF amination above to open and play it in a browser window.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

child_creation_example.c: Creating a Child Process


```c
/* A C program that forks, executes, and waits upon
 *    a child process. It demonstrates the use of
 *    the getpid(), getppid(), fork(), execlp(),
 *    waitpid(), perror(), fprintf(), atexit(), and
 *    exit() syscalls/functions.
 *
 *    Miriam Briskman, 3/15/2023
 *    CISC 3350, Brooklyn College
 *    Licensed under CC BY-NC 4.0
 */

#include <sys/types.h> // pid_t, gid_t, etc.
#include <unistd.h>    // Defines some system calls.
#include <sys/wait.h>  // wait(), waitpid(), etc.
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Creating a Child Process

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Copy-on-Write (COW)

Suppose that a process created a child process using the `fork()` system call but hasn't called `exec()` yet.

In early Unix systems, the parent's memory was fully copied over to the child's memory, which was time-consuming.

To save time, newer versions of the Linux kernel can avoid the full copy of the parent's memory section to the child's memory section. We could decide to only copy those parts of memory called **pages** that *differ* (after a change was made) from those of the parent. This technique is called **Copy-on-Write** (**COW**).

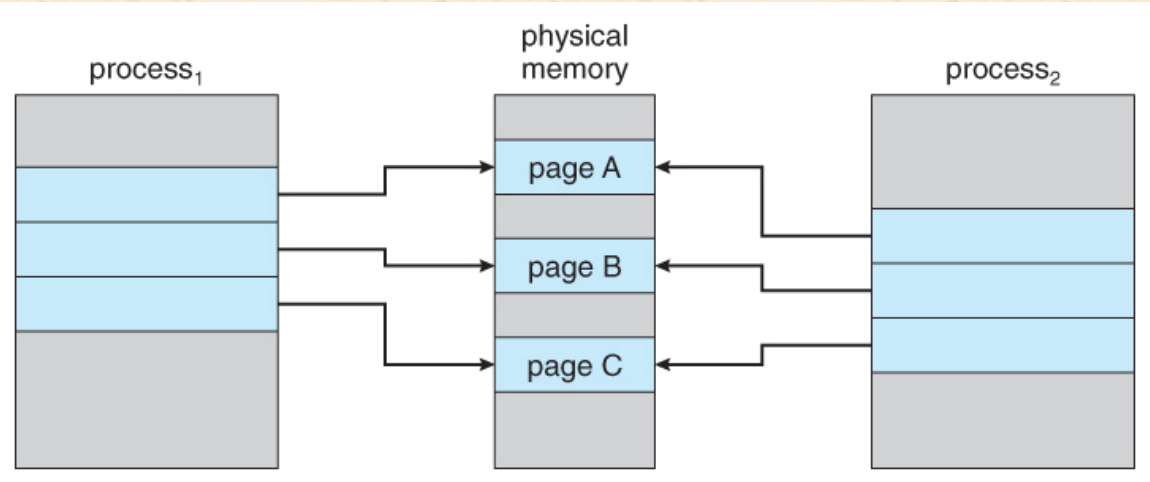
Since in many cases, we call the `exec()` system call right after `fork()`, this technique is actually very useful since we avoid doing an unnecessary copy before the call to `exec()`.

Before the COW method, a programmer could use a system call called `vfork()`, which didn't perform a full copy but required an immediate call to `exec()`.

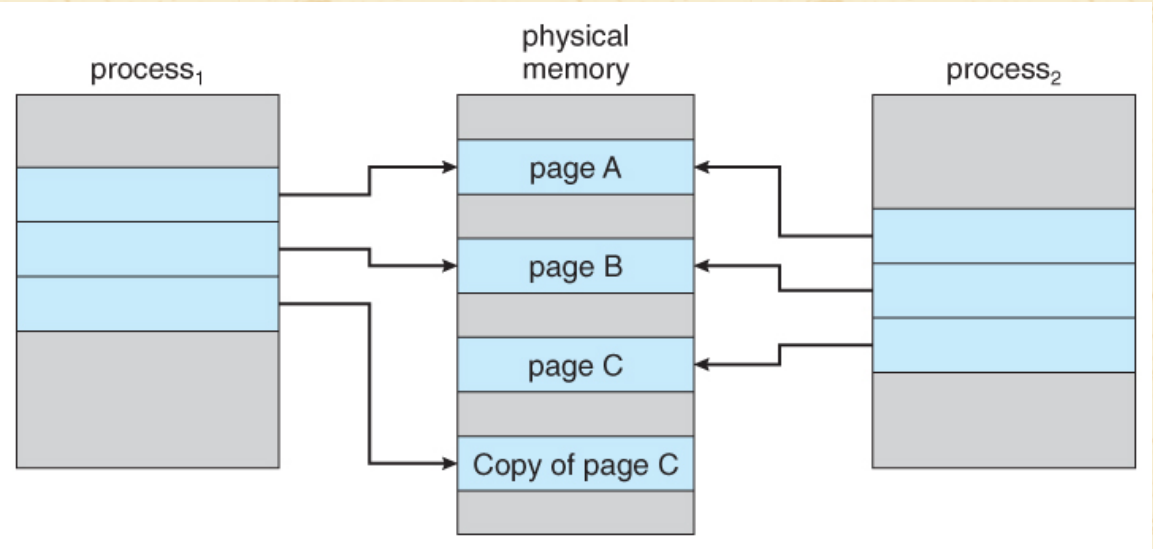
These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Copy-on-Write: Illustration

Suppose we have 2 processes, Process 1 (the parent) and Process 2 (the child of Process 1). When one of the pages of Process 1 changes, only then we create a new page copy. Otherwise, both processes will point to the same pages.



Before process 1 modifies page C. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.



After process 1 modifies page C. Taken from Bell, John T. "Virtual Memory." University of Illinois, Chicago.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Copy-on-Write: Illustration

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Terminating a Process

<table>
<tr><td colspan="2">Name: <code>exit()</code></td><td>System call or function: Function</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: performs shutdown steps, including (1) calling all the functions registered by <code>atexit()</code> (see slide 29) in the reverse order of registration, (2) flushing (saving) all the open standard I/O streams to the kernel, which will later save them to disk, (3) deleting any temporary files created with the <code>tmpfile()</code> function, and then (4) terminates the process by calling the system call <code>_exit()</code>. It doesn't return, so no instructions should follow a call to <code>exit()</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <stdlib.h> // Defines exit().</code></td></tr>
<tr><td colspan="4">Syntax: <code>void exit (int status);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>status</code>: the status with which we want to exit the process</td></tr>
<tr><td rowspan="3">What type it returns: <code>void</code></td><td colspan="3">On success: – This function doesn't return anything –</td></tr>
<tr><td colspan="3">On failure: – This function doesn't return anything –</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>exit (EXIT_FAILURE); /* Same as exit (1); */</code></td></tr>
<tr><td colspan="4">Other variations: <code>_exit()</code>, <code>atexit()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Terminating a Process

Name: `_exit()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet
What it does: lets the kernel terminate the process.		
What libraries you must include: `#include <unistd.h> // Defines _exit().`		
Syntax: `void _exit (int status);`		
Description of arguments: `status`: the status with which we want to exit the process		
What type it returns: `void`	**On success:** – This function doesn't return anything –	
	On failure: – This function doesn't return anything –	
	If failure, does it set `errno`? No	
Quick example: `_exit (EXIT_FAILURE); /* Same as _exit (1); */`		
Other variations: `exit()`, `atexit()`		

Once the program terminates, the parent process that called this program, or the Linux terminal/shell, could view the exit `status` of that program by typing:

```
$ echo "$?"
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Other Ways for a Process to Terminate

Besides using `exit()` to terminate, a process can also terminate in any of the following ways:

- When it returns from `main()`:
  - Behind the scenes, the compiler inserts a call to `exit()` wherever a `return` statement appears.
- When it receives a signal that terminates it.
  - Programs can communicate with each other by sending signals to one another.
  - The default action of some signal types is to terminate the program right away, such as with the `SIGTERM` or `SIGKILL` signals.
  - We will learn more about signals in Topic 10.
- When it executes an illegal/erroneous instruction that causes a segmentation fault, stack fault, division-by-0 error, etc.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Functions Called upon Termination

Name: `atexit()`	**System call or function:** Function	**Links:** Online Manual | Course Packet

What it does: takes and registers the name of a function, causing it to be executed right before the program terminates. Functions are called in the *reverse* order of registration (LIFO). The `function` must be defined somewhere in your program.

What libraries you must include:

```
#include <stdlib.h> // Defines atexit().
```

Syntax:

```
int atexit (void (*function)(void));
```

Description of arguments: `function`: the name of a function that accepts no arguments and returns nothing

What type it returns: `int`	**On success:** `0`
	On failure: `-1`
	If failure, does it set `errno`? No

Quick example:

```
void wow (void) { printf ("Bye, Bye!\n"); } /* We define a function called wow() */
if (atexit (wow) == -1)
    /* Handle the error */
exit (EXIT_SUCCESS); /* wow() will be called at this point */
```

Other variations: `exit()`, `_exit()`

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Examples of using `atexit()`


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

void out (void)
{
    printf ("atexit() succeeded!\n");
}

int main ()
{
    if (atexit (out))
        fprintf (stderr, "atexit() failed!\n");
    return EXIT_SUCCESS; // out() should be called here
}
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Examples of using `atexit()`


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

void bye (void)
{
    printf ("Signing off!\n");
}

int main ()
{
    short unsigned boroughs = 5;

    printf ("NYC has %hu boroughs.\n", boroughs);
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Examples of using `atexit()`

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Waiting for Terminated Child Processes

By spiritual musclehead

Generally, the parent receives a `SIGCHLD` signal when a child terminates.

But sometimes, a parent wants to obtain more info after the child finishes executing, such as the child's return status.

By design, when a child dies before its parent, the kernel puts the child into a special process state (known as a **zombie**) since the child is supposed to be terminated at this moment but is still alive (and hence the name zombie!)

A zombie waits for its parent to make an inquiry about its status (known as **waiting** on the child process.)

Only after the parent obtains the info, the zombie formally exits and ceases to exist.

Linux provides several ways for waiting, with the simplest being `wait()`.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Waiting for a Child to Terminate

<table>
<tr><td>Name: <code>wait()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: blocks (= waits for a child) if no child has terminated yet or returns immediately if some child did terminate and the <code>SIGCHILD</code> signal was received.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <wait.h> // Defines wait().</code></td></tr>
<tr><td colspan="3">Syntax: <code>pid_t wait (int *status);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>status</code>: a pointer to an <code>int</code> variable into which <code>wait()</code> will save the status of the returned child process. If you don't care about the status of the returned child, you can pass <code>NULL</code> instead of <code>status</code>.</td></tr>
<tr><td rowspan="3">What type it returns: <code>pid_t</code></td><td colspan="2">On success: The <code>pid</code> of the child process that terminates</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:

```
int status;
pid_t returned_child = wait (&status);
if (returned_child == -1)  /* Handle the error */
```

</td></tr>
<tr><td colspan="3">Other variations: <code>waitpid()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Checking the Child's Exit Status

An easy way to understand what the integer inside `status` means is by calling the following functions after a call to `wait()`:

```
if (WIFEXITED (status))
    printf ("Normal termination with the exit status = %d.\n",
            WEXITSTATUS (status));
if (WIFSIGNALED (status))
    printf ("Killed by the signal = %d%s.\n",
            WTERMSIG (status),
            WCOREDUMP (status) ? " (dumped core)" : "");
if (WIFSTOPPED (status))
    printf ("Stopped by the signal = %d.\n",
            WSTOPSIG (status));
if (WIFCONTINUED (status))
    printf ("Continued.\n")
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

wait_syscall_example.c: Waiting upon a Child Process


```c
// A program demonstrating the call to wait().
//
// This program is taken from Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     page 153.

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

int main (void)
{
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Waiting for a Specific Child to Terminate

<table>
<tr><td colspan="2">Name: <code>waitpid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: waits for a child with a specific pid or process group id, and lets you specify special waiting <code>options</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <wait.h> // Defines waitpid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>pid_t waitpid (pid_t pid, int *status, int options);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>pid</code>: the <code>pid</code> or <code>pgid</code> of the process(s) you are waiting for. More info on slide 38.
<code>status</code>: a pointer to an <code>int</code> variable into which <code>wait()</code> will save the status of the returned child process. If you don't care about the status of the returned child, you can pass <code>NULL</code> instead of <code>status</code>.
<code>options</code>: a binary OR of zero or more of: <code>WNOHANG</code>, <code>WUNTRACED</code>, <code>WCONTINUED</code>. More info on slide 38.</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>pid_t</code></td><td colspan="2">On success: The <code>pid</code> of the child process that terminates</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>int status;
pid_t returned_child = waitpid (2987, &status, 0);
if (returned_child == -1) /* Handle the error */</pre></td></tr>
<tr><td colspan="4">Other variations: <code>wait()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Waiting for a Specific Child to Terminate

`waitpid()` allows great flexibility in indicating who we are waiting for. Specifically, various values can be passed as the `pid` argument to the function and mean the following:

- A negative integer as `pid` will cause `waitpid()` to wait for any child process whose process group ID (`pgid`) is: |`pid`| (e.g., -500 means you are waiting for any child in the group with `pgid` of 500.)
- `-1` as `pid` will cause `waitpid()` to wait for any child process whatsoever (the same exact behavior as with `wait()`.)
- `0` as `pid` will cause `waitpid()` to wait for any child process that belongs to the same process group as the parent.
- A positive integer as `pid` will cause `waitpid()` to wait for the child process with the exact process id of `pid`.

Here are the meanings of the various `options` types:

- `0`: no option passed.
- `WNOHANG`: return immediately if no child has exited yet.
- `WUNTRACED`: return if a child has stopped.
- `WCONTINUED`: return if a stopped child has been resumed ('continued') via the signal `SIGCONT`

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Example of using `waitpid()`


```
int status;
pid_t pid = waitpid (4871, &status, WNOHANG);

if (pid == -1)
    perror ("waitpid");
else
{
    if (WIFEXITED (status))
        printf ("Child with pid %d normally terminated with exit status = %d.\n",
                pid, WEXITSTATUS (status));
    if (WIFSIGNALED (status))
```

Note that, if `WNOHANG` is specified as an option, and no child or children have returned or changed status when `waitpid()` runs, `waitpid()` won't wait at all for children and will immediately return `0`.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

What Happens if We Don't Wait for a Child?

By spiritual musclehead

A **zombie** (also called a **ghost**) process is a terminated process but not waited upon by its parent.

Zombie/ghost processes continue to consume a small portion of the operating system's resources.

Be a responsible parent: if you `fork()` a child process, you must `wait()` on the child – even if you don't care about the child's return status.

But what happens if the parent dies before the child? Whenever a process terminates, the kernel walks its children and re-parents them to the `init` process. (***Bonus question***: what is the `pid` of `init`?)

The `init` process periodically waits on all its children and allows them to exit normally and fully, ensuring zombies don't live for long and, therefore, don't take up valuable OS resources.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

What Happens if We Don't Wait for a Child?

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Users and Groups

Taken from The New York Times

A process is associated with a user (= **owner**) and a group (= **owning group**): the process's `uid` (= **user ID**) and `gid` (= **group ID**) dictate what operations the process can do.

From a computer security viewpoint, use the least-privileged rights for a process, and only to elevate them when necessary.

There are 3 types of user and group IDs: Real, Effective, and Saved IDs:

- The **real `uid`** (= `ruid`) is that of the user who originally ran the process. It is set to the real `uid` of the process's parent and doesn't change during an `exec()` call.
  - The `login` process sets the `uid` of the user's shell process (e.g., `/bin/bash`) to that of the user.
- The **effective `uid`** (= `euid`) is the `uid` that the process is currently using. It could be different from its real `uid` after executing a command/program that internally calls `setuid()` (such as `passwd`, whose owner is root, to modify the user's password.) Thus, the effective `uid` becomes the root's `uid`!
- The **saved `uid`** (= `suid`) is the process's original effective user ID, but upon an `exec()` call, the kernel sets the saved `uid` to the new effective `uid`, facilitating switches between them.
- Ultimately, the effective `uid` is the value that matters in dealing with permissions.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Changing the User ID

<table>
<tr><td>Name: <code>setuid()</code></td><td colspan="2">System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: changes the effective user ID of the current process to <code>uid</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines setuid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int setuid (uid_t uid);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>uid</code>: the user ID that you want to set as the effective ID of the current process. A non-root user can only pass its real or saved user ID for <code>uid</code>.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="3">On success: <code>0</code></td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:<pre><code>if (setuid (1673) == -1)
    /* Handle the error */</code></pre></td></tr>
<tr><td colspan="4">Other variations: <code>setgid()</code>, <code>seteuid()</code>, <code>setegid()</code></td></tr>
</table>

If the current effective user ID is `0` (the `root` user ran this program!), then all 3 user IDs (real, effective, and saved) are set to `uid`, which could be any value, and once they are changed into a nonzero ID, its impossible to re-gain root privileges.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Changing the Group ID

<table>
<tr><td>Name: <code>setgid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: changes the effective group ID of the current process to <code>gid</code>.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <unistd.h> // Defines setgid().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int setgid (gid_t gid);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>gid</code>: the group ID that you want to set as the effective group ID of the current process. A non-root user can only pass its real or saved group ID for <code>gid</code>.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:
<code>if (setgid (745) == -1)
    /* Handle the error */</code></td></tr>
<tr><td colspan="3">Other variations: <code>setuid()</code>, <code>seteuid()</code>, <code>setegid()</code></td></tr>
</table>

If the current effective group ID is `0` (the `root` user ran this program!), then all 3 group IDs (real, effective, and saved) are set to `gid`, which could be any value, and once they are changed into a nonzero ID, its impossible to re-gain root privileges.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Changing the Effective User ID

<table>
<tr><td colspan="2">Name: <code>seteuid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: changes the effective user ID of the current process to <code>euid</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines seteuid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>int seteuid (uid_t euid);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>euid</code>: the user ID that you want to set as the effective ID of the current process. If the <code>root</code> runs this process, it may provide any value for <code>euid</code>; Nonroot users can only use their real or saved user IDs.</td></tr>
<tr><td rowspan="3" colspan="2">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>if (seteuid (1673) == -1)
    /* Handle the error */</pre></td></tr>
<tr><td colspan="4">Other variations: <code>setuid()</code>, <code>setgid()</code>, <code>setegid()</code></td></tr>
</table>

For nonroot users, `seteuid()` is equivalent to a `setuid()` call. But for the `root` user, this version doesn't change real/saved IDs.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Changing the Effective Group ID

<table>
<tr><td>Name: <code>setegid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: changes the effective group ID of the current process to <code>egid</code>.</td></tr>
<tr><td colspan="3">What libraries you must include: <code>#include <unistd.h> // Defines setegid().</code></td></tr>
<tr><td colspan="3">Syntax: <code>int setegid (gid_t egid);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>egid</code>: the group ID that you want to set as the effective ID of the current process. If the <code>root</code> runs this process, it may provide any value for <code>egid</code>; Nonroot users can only use their real or saved group IDs.</td></tr>
<tr><td rowspan="3">What type it returns: <code>int</code></td><td colspan="2">On success: <code>0</code></td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:<pre>if (setegid (1673) == -1)
    /* Handle the error */</pre></td></tr>
<tr><td colspan="3">Other variations: <code>setuid()</code>, <code>setgid()</code>, <code>seteuid()</code></td></tr>
</table>

For nonroot users, `setegid()` is equivalent to a `setgid()` call. But for the `root` user, this version doesn't change real/saved group IDs.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

More on User/Group IDs

Well, which of these system calls should I use in my program?

- Nonroot processes should use `seteuid()` to change their effective user IDs.
- Root processes should use `setuid()` if all 3 IDs need to be changed, and `seteuid()` if only the effective user ID needs to be changed.

Next, we'll quickly cover the system calls that we can use to get a certain ID: all of these syscalls never fail and always return the requested ID!

- `uid_t getuid (void);  // Get the real user ID.`
- `gid_t getgid (void);  // Get the real group ID.`
- `uid_t geteuid (void); // Get the effective user ID.`
- `gid_t getegid (void); // Get the effective group ID.`

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Getting the Real User ID

<table>
<tr><td colspan="2">Name: <code>getuid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: returns the real user ID of the calling process.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines getuid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>uid_t getuid (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – This system call doesn't accept any argument. –</td></tr>
<tr><td rowspan="3">What type it returns: <code>uid_t</code></td><td colspan="3">On success: The real user ID of the process.</td></tr>
<tr><td colspan="3">On failure: – This syscall never fails –</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>printf ("My real user ID is: %d\n", getuid());</code></td></tr>
<tr><td colspan="4">Other variations: <code>getgid()</code>, <code>geteuid()</code>, <code>getegid()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Getting the Real Group ID

<table>
<tr><td colspan="2">Name: <code>getgid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: returns the real group ID of the calling process.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines getgid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>gid_t getgid (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – This system call doesn't accept any argument. –</td></tr>
<tr><td rowspan="3">What type it returns: <code>gid_t</code></td><td colspan="3">On success: The real group ID of the process.</td></tr>
<tr><td colspan="3">On failure: – This syscall never fails –</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>printf ("My real group ID is: %d\n", getgid());</code></td></tr>
<tr><td colspan="4">Other variations: <code>getuid()</code>, <code>geteuid()</code>, <code>getegid()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Getting the Effective User ID

<table>
<tr><td colspan="2">Name: <code>geteuid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: returns the effective user ID of the calling process.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines geteuid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>uid_t geteuid (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – This system call doesn't accept any argument. –</td></tr>
<tr><td rowspan="3">What type it returns: <code>uid_t</code></td><td colspan="3">On success: The effective user ID of the process.</td></tr>
<tr><td colspan="3">On failure: – This syscall never fails –</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>printf ("My effective user ID is: %d\n", geteuid());</code></td></tr>
<tr><td colspan="4">Other variations: <code>getuid()</code>, <code>getgid()</code>, <code>getegid()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Getting the Effective Group ID

<table>
<tr><td colspan="2">Name: <code>getegid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: returns the effective group ID of the calling process.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines getegid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>gid_t getegid (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – This system call doesn't accept any argument. –</td></tr>
<tr><td rowspan="3">What type it returns: <code>gid_t</code></td><td colspan="3">On success: The effective group ID of the process.</td></tr>
<tr><td colspan="3">On failure: – This syscall never fails –</td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? No</td></tr>
<tr><td colspan="4">Quick example: <code>printf ("My effective group ID is: %d\n", getegid());</code></td></tr>
<tr><td colspan="4">Other variations: <code>getuid()</code>, <code>getgid()</code>, <code>geteuid()</code></td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Getting Various IDs

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Sessions and Process Groups

Each process is a member of a **process group**: a set of one or more processes associated with each other.

- Signals can be sent to all the processes in the group (e.g., terminating them all at once.)

Each process group has a **process group ID** (`pgid`) and a **process group leader**: the main process in the group.

- The process group will keep existing as long as there is one remaining member in the group.
- It continues to exist even if the process group leader terminates (a different process becomes the new leader.)

When a new user logs into a Linux machine, the `login` process creates a new **session** (= a set of one or more process groups) that consists of a single process: the user's shell (e.g., `sh` or `bash`.)

- The shell becomes the **session leader** (= the main process in the session.)
- The **session ID** (`sid`) is the `pid` of the session's leader process.
- Sessions control the activities of a logged-in user, and associate the user with a **controlling terminal** (= the terminal where all these activities take place.)

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Sessions and Process Groups

Sessions exist in order to relate logins with the terminals in which actions take place.

- Every session has exactly one foreground process group and zero or more (0+) additional background process groups.
- When a user exits a terminal, a `SIGQUIT` signal is sent to all processes in the foreground process group, which, in most cases, results in terminating all these foreground processes.
- On a typical OS, there are many sessions: one for each user session, and others for processes that are not tied to user sessions, such as daemons.

Example of a scenario involving sessions and groups of processes:

1. A user logs into the system. The shell that starts running, `bash`, has the `pid` of `253`.
2. This `bash` process is now the sole member and leader of a new process group with `pgid` = `253`. Also, it is the sole member and the leader of a new session with `sid` = `253`.
3. Commands that this user runs in the shell run in a new process group but still within session number `253`.
4. One of these groups is the foreground group, and all the other are background process groups.

 These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Sessions and Process Groups

The following 'piped' Linux command results in one process group containing 3 processes:

```
$ cat myFile.txt | grep Hello | sort
```

Since no '&' symbol was added to at the end of the commands, this group will run in the foreground.

The diagram here shows the relationship between sessions, groups, processes, and controlling terminals:

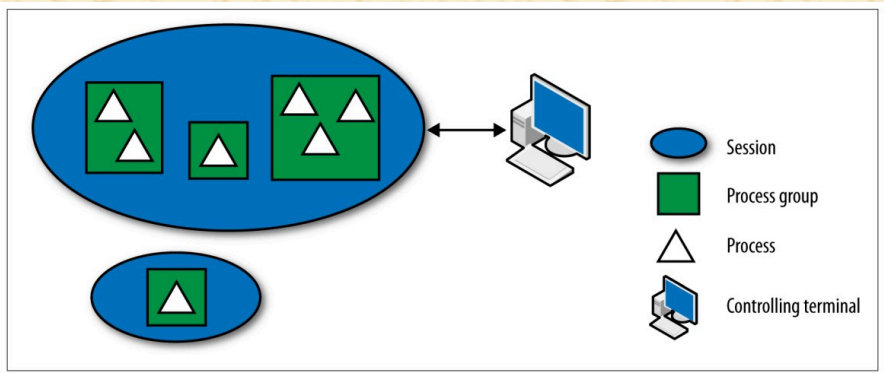


Figure 5-1. Relationship between sessions, process groups, processes, and controlling terminals

Relationship between sessions, groups, processes, and controlling terminals. "Figure 5-1" (page 169), Love, Robert. *Linux System Programming: Talking Directly to the Kernel and C Library*. 2013.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Sessions and Process Groups

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Session System Calls

<table>
<tr><td colspan="2">Name: <code>setsid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: creates a new session (only if the calling process isn't some group's leader,) and a group inside this session, and then becomes both the session's and the group's leader; both the <code>sid</code> and <code>pgid</code> are set to the process's <code>pid</code>.</td></tr>
<tr><td colspan="4">What libraries you must include: <code>#include <unistd.h> // Defines setsid().</code></td></tr>
<tr><td colspan="4">Syntax: <code>pid_t setsid (void);</code></td></tr>
<tr><td colspan="4">Description of arguments: – This system call doesn't accept any argument. –</td></tr>
<tr><td colspan="2" rowspan="3">What type it returns: <code>pid_t</code></td><td colspan="2">On success: The ID of the created session.</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<code>pid_t new_session = setsid();</code>
<code>if (new_session == -1) /* Handle error */</code></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

The only possible error `errno` value is `EPERM`, indicating that the process is already the leader of an existing process group.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

More on `setsid()`

To ensure that the current process is NOT a process group leader already and to eliminate the error possibility when calling `setsid()`, do the following in your code:

1. From the current process, call `fork()`.
2. Exit the parent.
3. Let the child call the `setsid()` syscall.

Why does this solution help?

- Because the child isn't a leader of any group.
- As such, the `EPERM` error won't occur when calling `setsid()`.

```
pid_t pid = fork();
if (pid == -1) {
    perror ("fork");
    exit (EXIT_FAILURE);
}
else if (pid != 0)
    exit (EXIT_SUCCESS);
// Exit the parent

if (setsid () == -1) {
    perror ("setsid");
    exit (EXIT_FAILURE);
}
```

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Obtaining the Current Session ID

<table>
<tr><td colspan="2">Name: <code>getsid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="4">What it does: returns the ID of the session in which the process with the ID of <code>pid</code> is a member; used mainly for diagnostics.</td></tr>
<tr><td colspan="4">What libraries you must include:
<pre>#define _XOPEN_SOURCE 500 // Needed for getsid().
#include <unistd.h> // Defines getsid().</pre></td></tr>
<tr><td colspan="4">Syntax: <code>pid_t getsid (pid_t pid);</code></td></tr>
<tr><td colspan="4">Description of arguments: <code>pid</code>: the ID of the process whose session ID we want to get. If you pass <code>0</code> as <code>pid</code>, then <code>getsid()</code> returns the session ID of the current process.</td></tr>
<tr><td rowspan="3">What type it returns: <code>pid_t</code></td><td colspan="3">On success: A session ID of the indicated process.</td></tr>
<tr><td colspan="3">On failure: <code>-1</code></td></tr>
<tr><td colspan="3">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="4">Quick example:
<pre>pid_t my_session = getsid (0);
if (my_session == -1) /* Bonus question: why won't getsid (0) result in an error? */</pre></td></tr>
<tr><td colspan="4">Other variations: N/A</td></tr>
</table>

The only possible error `errno` value is `ESRCH`, indicating that `pid` is an invalid process ID (e.g., nonexisiting.)

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Process Group System Calls

Name: `setpgid()`	**System call or function:** System Call	**Links:** Online Manual | Course Packet

What it does: sets the process group ID of the process identified by `pid` to `pgid` (i.e., moves it over to that group.)

What libraries you must include:
```
#define _XOPEN_SOURCE 500 // Needed for setpgid().
#include <unistd.h> // Defines setpgid().
```

Syntax:
```
int setpgid (pid_t pid, pid_t pgid);
```

Description of arguments: `pid`: the ID of the process that you want to move to a different group. If you pass `0` as `pid`, then `setpgid()` moves the current process. Also, `pid` must either be the calling process or its child before calling `exec()` be in the same session as the calling process. Also, it can't be a session leader (i.e., you can't move a leader to a different group.)
`pgid`: the ID of the destination group. If you pass `0` as `pgid`, `pid` is used as the group ID (i.e., it becomes the group's leader!) Note that the to-be-assigned `pgid` must be in the same session as the calling process.

What type it returns: `int`	**On success:** `0`
	On failure: `-1`
	If failure, does it set `errno`? Yes

Quick example:
```
if (setpgid (0, 3875) == -1) /* Handle the error */
```

Other variations: N/A

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Process Group System Calls

<table>
<tr><td>Name: <code>getpgid()</code></td><td>System call or function: System Call</td><td>Links: Online Manual | Course Packet</td></tr>
<tr><td colspan="3">What it does: returns the process group ID of the process identified by <code>pid</code>.</td></tr>
<tr><td colspan="3">What libraries you must include:
<code>#define _XOPEN_SOURCE 500 // Needed for getpgid().</code>
<code>#include <unistd.h> // Defines getpgid().</code></td></tr>
<tr><td colspan="3">Syntax: <code>pid_t getpgid (pid_t pid);</code></td></tr>
<tr><td colspan="3">Description of arguments: <code>pid</code>: the ID of the process whose group ID you want to find. If you pass <code>0</code> as <code>pid</code>, then <code>getpgid()</code> returns the group ID of the current process.</td></tr>
<tr><td rowspan="3">What type it returns: <code>pid_t</code></td><td colspan="2">On success: the group ID of the indicated process</td></tr>
<tr><td colspan="2">On failure: <code>-1</code></td></tr>
<tr><td colspan="2">If failure, does it set <code>errno</code>? Yes</td></tr>
<tr><td colspan="3">Quick example:
<code>pid_t my_group = getpgid (0);</code>
<code>if (my_group == -1) /* Handle the error */</code></td></tr>
<tr><td colspan="3">Other variations: N/A</td></tr>
</table>

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Process Group System Calls

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Daemons

As we learned earlier, a **daemon** is a process that runs in the background, not connected to any controlling terminal.

Usually, it starts at the computer's boot (launch) time, runs as `root`, and handles system-level tasks.

Daemons conventionally have a 'd' as the last letter of their names (e.g., `sshd`, `httpd`.)

To be a daemon, a process must satisfy 2 requirements:

- Must run as a child of `init`.
- Must not be connected to a terminal.

The next slide shows pseudo-code steps that a process needs to undergo to become a daemon.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Daemon: Steps

To become a daemon, a process must do the following:

1. Call `fork()` – this creates a new process.
2. In the parent, call `exit()` – so that `init`, the great-grandparent, recognizes that the parent of the child is no longer running, and that the child is not a process group leader.
3. Call `setsid()` – this gives the child a new process group and session, both of which it leads. This ensures the process has no associated controlling terminal.
4. Change the working directory to the root directory (`/`) by calling the `chdir()` function.
5. Close all `fd`s – you don't want to keep any files open anymore, including the keyboard and the screen!.
6. Open `fd`s 0, 1, and 2 and redirect the input & output of all of them to `/dev/null`.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

daemon_creation.c: Creating a Daemon!


```
// A program that shows how a daemon process is
//    created in Linux.
//
// This program is taken from Linux System Programming:
//     Talking Directly to the Kernel and C Library,
//     2nd Edition, by Love. ISNB: 978-1-44933953-1,
//     pages 173-174.

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

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Daemons

Any questions?

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.

Processes: Sources

Topic 7: Processes. Briskman, Miriam. *CISC 3350 OER Course Packet.* 2024, CC BY-NC 4.0.
URL: https://www.sci.brooklyn.cuny.edu/~briskman/cisc/3350/CISC_3350_OER_Course_Packet.pdf#topic+7:+processes.

Partly based on the teaching of Professor **Yongqing Xiang** of Brooklyn College.

These notes by Miriam Briskman are licensed under CC BY-NC 4.0 and based on sources.