# ЛЕКЦІЯ 18. ДОСТУП ПРОЦЕСІВ ДО ФАЙЛІВ І КАТАЛОГІВ В UNIX-ПОДІБНИХ.

## Поняття процесу

Процес – поняття сукупності програмного коду й даних, завантажених на згадку ЕОМ. Процес це не запущена програма (додаток) або команда, тому що додаток може створювати кілька процесів одночасно. Код процесу не обов'язково повинен виконуватися в сучасний момент часу, тому що процес може перебувати в стані сплячі. У цьому випадку виконання коду такого процесу припинено. Існує всього 3 стани, у яких може перебувати процес:

- ***Працюючий процес*** – у цей момент код цього процесу виконується.
- ***Сплячий процес*** – у цей момент код процесу не виконується в очікуванні якого або події (натискання клавіші на клавіатурі, надходження даних з мережі й т.д.)
- Процес-Зомбі – сам процес уже не існує, його код і дані вивантажені з оперативної пам'яті, але запис в таблиці процесів залишається по тим або іншим причинам.

Кожному процесу в системі призначаються числові ідентифікатори (особисті номери) у діапазоні від 1 до 65535 (PID - Process Identifier) і ідентифікатори батьківського процесу (PPID - Parent Process Identifier).

**PID** є ім'ям процесу, по якому ми можемо адресувати процес в операційній системі при використанні різних засобів перегляду й керування процесами.

**PPID** визначає родинні відносини між процесами, які в значній мірі визначають його властивості й можливості.

Інші параметри, які необхідні для роботи програми, називають “оточення процесу”. Одним з таких параметрів - керуючий термінал - мають далеко не всі процеси.

Процеси, не прив'язані до якогось конкретного термінала називаються “**демонами**” (daemons).

Такі процеси, будучи запущеними користувачем, не завершують свою роботу із закінченні сеансу, а продовжують працювати, тому що Вони не зв'язані ніяк з поточним сеансом і не можуть бути автоматично завершені. Як правило, за допомогою демонів реалізуються серверні служби, так наприклад сервер печатки реалізований процесом-демоном cupsd, а сервер журналювання - syslogd.

## Перегляд списку процесів

Для перегляду списку процесів в Linux існує команда **ps. ps [PID] options** – перегляд списку процесів. Без параметрів ps показує всі процеси, которы були запущені протягом поточної сесії, за винятком демонів. Options може приймати одне з наступних значень або їхньої комбінації:

**-A або -e** – показати всі процеси **-f** – відсортувати за алфавітом

**-w** – показати повні рядки опису процесів.

***Команда*** **ps -ef**

виведе всі процеси, відсортовані за алфавітом.

Процеси в ОС Linux мають ті ж права, якими володіє користувач, від чийого ім'я був запущений процес.

Насправді операційна система сприймає працюючого в ній користувача як набір запущених від його ім'я процесів. Адже й сам сеанс користувача відкривається в командній оболонці (або оболонці X) від імені користувача. Тому коли ми говоримо “права доступу користувача до файлу” те маємо на увазі “права доступу процесів, запущених від імені користувача до файлу”.

Для визначення ім'я користувача, що запустив процес, операційна система використовує реальні ідентифікатори користувача й групи, призначувані процесу. Але ці ідентифікатори не є вирішальними при визначенні прав доступу. Для цього в кожного процесу існує інша група ідентифікаторів - ефективні.

Як правило, реальні й ефективні ідентифікатори процесів однакові, але є й виключення. Наприклад, для роботи утиліти passwd необхідно використовувати ідентифікатор суперкористувача, тому що тільки суперкористувач має права на запис у файли паролів. У цьому випадку ефективні ідентифікатори процесу будуть відрізнятися від реальних. Виникає резонне питання - як це було реалізовано?

У кожного файлу є ще один набір прав доступу - біти SUID і SGID. Ці біти дозволяють при запуску програми привласнити їй ефективні ідентифікатори власника й групи-власника відповідно й виконувати процес із правами доступу іншого користувача. Тому що файл passwd належить користувачеві root і в нього встановлений біт SUID, те при запуску процес passwd буде мати права користувача root. Установлюються біти SGID і SUID програмою chmod.

## Будова таблиці процесів

Ми з вами розглянули поняття процесу, способи відображення процесів і права доступу. Але для комфортної роботи в операційній системі цього, погодитеся, мало. Необхідно ще ефективно управляти процесами. Але для реалізації керування ми спочатку розглянь будову таблиці процесів:

Батьком всіх процесів у системі є процес *init*. Його PID завжди 1, PPID - 0. Всю таблицю процесів можна уявити собі у вигляді дерева, у якому коренем буде процес init. Цей процес хоч і не є частиною ядра, але виконує в системі дуже важливу роль, про яку ми з вами поговоримо на 16-ом занятті.

Процеси, імена яких укладені у квадратні дужки, наприклад “[keventd]” - це процеси ядра. Ці процеси управляють роботою системи, а точніше такими її частинами, як менеджер пам'яті, планувальник часу процесора, менеджери зовнішніх пристроїв і так далі.

Інші процеси є користувальницькими, запущеними або з командного рядка, або під час ініціалізації системи.

Життя кожного процесу представлена наступними фазами:

**Створення процесу** – на цьому етапі створюється повна копія того процесу, що створює новий. Наприклад, ви запустили з інтерпретатора на виконання команду ls. Командний інтерпретатор створює свою повну копію.

**Завантаження коду процесу й підготовка до запуску** – копія, створене на першому етапі заміняються кодом завдання, яку необхідно виконати й створюється її оточення – установлюються необхідні змінні й т.п.

Виконання процесу **Стан зомбі** – на цьому етапі виконання процесу закінчився, його код

вивантажується з пам'яті, оточення знищується, але запис у таблиці процесів ще залишається.

**Вмирання процесу** – після всіх завершальних стадій віддаляється запис із таблиці процесів – процес завершив свою роботу.

Під час роботи процесу, ядро контролює його стан, і у випадку виникнення непередбаченої ситуації управляє процесом за допомогою посилки йому сигналу. Процес може скористатися дією за замовчуванням, або, якщо в нього є оброблювач сигналу, то він може перехопити або ігнорувати сигнал. Сигнали SIGKILL і SIGSTOP неможливо не перехопити, не ігнорувати.

За замовчуванням можливі кілька дій:

- ігнорувати - продовжувати роботу, незважаючи на те, що отримано сигнал. завершити - завершити роботу процесу.
- завершити + core - завершити роботу процесу й створити файл у поточному каталозі з ім'ям core, що містить образ пам'яті процесу (код і дані).
- зупинити - призупинити виконання процесу, але не завершувати його роботу й не вивантажувати код з пам'яті.

## Запуск дочірніх процесів

Запуск одного *процесу* **замість** іншого організований в Linux за допомогою *системного виклику* exec(). Старий *процес* із пам'яті віддаляється назавжди, замість

нього завантажується новий, при цьому настроювання **оточення** не міняється, навіть **PID** залишається колишнім. **Повернутися** до виконання старого *процесу* неможливо, хіба що запустити його знову за допомогою того ж exec() (від "execute" – "виконати"). До речі,

**ім'я файлу** (програми), з якого запускається *процес*, і **власне ім'я** *процесу* (у *таблиці процесів*) можуть і не збігатися. Власне ім'я *процесу* – це такий же параметр командного рядка, як і ті, що передаються йому користувачем: для exec() потрібно й *шлях* до файлу,

і **повний** командний рядок, **нульовий** (стартовий) елемент якої – саме назва команди. От звідки "-" на початку ім'я *стартового командного інтерпретатора* (-bash):

його "підсунула" програма login, щоб була можливість відрізняти його від інших

запущених тем же користувачем оболонок.

Для роботи командного інтерпретатора одного exec() недостатньо.

Справді, shell

не просто запускає утиліту, а чекає її завершення, обробляє результати її роботи й продовжує діалог з користувачем.

Для цього в Linux служить *системний виклик* fork() (“розвилка"), застосування якого приводить до виникнення ще одного, ***дочірнього,*** *процесу* – точної копії вйого, що *породив,* батьківського. *Дочірній процес* нічим не відрізняється від *батьківського*: має таке ж оточення, ті ж стандартне уведення й *стандартний вивід*, однакове вміст пам'яті й продовжує роботу з тої ж самої крапки (повернення з fork()). Відмінностей два: по-перше, ці *процеси* мають різні **PID**, під якими вони зареєстровані в *таблиці процесів*, а по-друге, розрізняється *повертається значение, що,* fork(): *батьківський процес* одержує як результат fork() *ідентифікатор процесу-нащадка*, а процес-*нащадок* одержує "0".

Подальші дії shell при запуску якої-небудь програми очевидні.Shell-Нащадок негайно викликає цю програму за допомогою exec(), а shell-батько чекає завершення роботи процесу-нащадка (PID якого йому відомий) за допомогою ще одного системного виклику, wait(). Дочекавшись і проаналізувавши результат команди, shell продовжує роботу:

```
[methody@localhost methody]$ cat > loop while
  true; do true; done
^D
[methody@localhost methody]$ sh loop ^C
[methody@localhost methody]$
```

Якби в описаній вище ситуації *батьківський процес* не чекав, поки *дочірній* завершиться, а відразу продовжував працювати, вийшло б, що обидва *процеси* виконуються паралельно: поки запущений *процес* щось робить, користувач продовжує командувати оболонкою.

Для того щоб запустити *процес* паралельно, в shell досить додати "&" у кінець командного рядка:

```
[methody@localhost methody]$ sh loop&
  [1] 3634
[methody@localhost methody]$ ps -f
  UID          PID PPID  C STIME TTY          TIME CMD
  methody     3590  1850  0 13:58 tty3     00:00:00 -bash
  methody     3634  3590 99 14:03 tty3     00:00:02 sh loop
  methody     3635 3590  0 14:03 tty3     00:00:00 ps -f
```

У результаті стартовий командний інтерпретатор (PID 3590) виявився батьком відразу двох процесів: sh, що виконує сценарій loop, і ps.

Процес, що запускається паралельно, називається фоновим (background). Фонові процеси не мають можливості вводити дані з того ж термінала, що й породивший їх shell (тільки з файлу), зате виводити дані на цей термінал можуть (правда, коли на тому самому терміналі упереміж з'являються

повідомлення від декількох фонових процесів, починається плутанина). При кожному терміналі в кожний момент часу може бути не більше одного активного (foreground) процесу, якому дозволено вводити дані із цього термінала. На час, поки команда (наприклад, cat) працює в активному режимі, що
породив її командний інтерпретатор "іде в тло", і там, у тлі, виконує свій wait().

Варто помітити, що паралельність роботи процесів в Linux – дискретна. Тут і зараз виконуватися може стільки процесів, скільки центральних процесорів є в комп'ютері (наприклад, один). Давши цьому одному процесу небагато попрацювати, система запам'ятовує все, що необхідно йому для роботи, припиняє його, і запускає наступний процес, що потім випливає й так далі. Виникає черга процесів, що очікують виконання. Тільки що що попрацював процес міститься в кінець цієї черги, а наступний вибирається з її початку. Коли черга знову доходить до того, першого процесу, система згадує необхідні для його виконання дані (вони називаються контекстом процесу), і він продовжує працювати як ні в чому не бувало. Така схема поділу часу між процесами називається псевдопараллелизмом.

## Сигнали

***Сигнал*** – це здатність *процесів* обмінюватися стандартними короткими повідомленнями безпосередньо за допомогою системи.Повідомлення-*Сигнал* не містить ніякої інформації, крім номера *сигналу* (для зручності замість номера можна використовувати визначене системою ім'я). Для того щоб передати *сигнал*, *процесу* досить задіяти *системний виклик* kill(), а для того щоб **прийняти** *сигнал*, не потрібно нічого. Якщо *процесу* необхідно якось по^-особливому реагувати на *сигнал*, він може зареєструвати **оброблювач**, а якщо оброблювача ні, за нього відреагує система. Як правило, це приводить до негайного завершення *процесу*, що одержав *сигнал*. Оброблювач *сигналу* запускається **асинхронно**, негайно після одержання *сигналу*, що б *процес* у цей час не робив.

*Сигнал* - коротке повідомлення, що посилається системою або *процесом* іншому *процесу*. Обробляється асинхронно спеціальною підпрограмою-оброблювачем. Якщо *процес* не обробляє *сигнал* самостійно, це робить система.

Два сигнали – 9 (KILL) і 19 (STOP) – завжди обробляє система. Перший з них потрібний для того, щоб убити процес напевно (звідси й назва). Сигнал STOP припиняє процес: у такому стані процес не віддаляється з таблиці процесів, але й не виконується доти, поки не одержить сигнал 18 (CONT) – після чого продовжить роботу. В Linux сигнал
STOP можна передати активному процесу за допомогою керуючого символу "^Z":

```
[methody@localhost methody]$ sh loop
^Z
  [1]+ Stopped                 sh loop
[methody@localhost methody]$ bg
[1]+ sh loop & [methody@localhost
methody]$ fg sh loop
```

**^C**
**[methody@localhost methody]$**

Передавати сигнали з командного рядка можна будь-яким процесам за допомогою команди kill -сигнал PID або просто kill PID, що передає сигнал 15 (TERM).

В 5.1 наведено список всіх сигналів, що існують у системі.

Таблиця 5.1. Список сигналів Linux

| Назва | Дія за замовчуванням | Значення |
|---|---|---|
| SIGABRT | Завершити + core | Сигнал відправляється, якщо процес викликає системний виклик abort() |
| SIGALRM | Завершити | Сигнал відправляється, коли спрацьовує таймер, раніше встановлений. |
| SIGBUS | Завершити + core | Сигнал свідчить про деяку апаратну помилку. Звичайно цей сигнал відправляється при звертанні до неприпустимої віртуальної адреси, для якого відсутня відповідна фізична сторінка. |
| SIGCHLD | Ігнорувати | Сигнал, що посилається батьківському процесу при завершенні його нащадка. |
| SIGSEGV | Завершити + core | Сигнал свідчить про обіг процесу до неприпустимої адреси або області пам'яті, для якого в процесу недостатньо привілеїв доступу. |
| SIGFPE | Завершити + core | Сигнал свідчить про виникнення особливих ситуацій, таких як розподіл на 0 або переповнення операції із плаваючою крапкою. |
| SIGHUP | Завершити | Сигнал посилає лідерові сеансу, пов'язаному з керуючим терміналом, що термінал від'єднався (втрата лінії). Сигнал також посилає всім процесам поточної групи при завершенні виконання лідера. Цей сигнал іноді використовують як найпростіший засіб межпроцессного взаємодії. Зокрема, він застосовується для повідомлення демонам про необхідність обновити конфігураційну інформацію. Причина вибору саме сигналу |

| | | |
|---|---|---|
| | | SIGHUP полягає в тім, що демон по визначенню не має керуючого термінала й, відповідно, звичайно не одержує цього сигналу. |
| SIGILL | Завершити + core | Сигнал посилає ядром, якщо процес спробує виконати неприпустиму інструкцію. |
| SIGINT | Завершити | Сигнал посилає ядром всім процесам при натисканні клавіші переривання (<CTRL>+<C>) |
| SIGKILL | Завершити | Сигнал, при одержанні якого виконання процесу припиняється. Цей сигнал не можна не перехопити, не проігнорувати. |
| SIGPIPE | Завершити | Сигнал посилає при спробі запису в сокет, одержувач даних якого завершив виконання або закрив файловий покажчик на сокет. |
| SIGPOLL | Завершити | Сигнал відправляється при настанні певної події для пристрою, що є опитуваним (наприклад, отриманий пакет по мережі) |

| **Назва** | **Дія за замовчуванням** | **Значення** |
|---|---|---|
| SIGPWR | Ігнорувати | Сигнал генерується при погрозі втрати живлення. Звичайно він відправляється, коли живлення системи перемикається на джерело безперебійного живлення (UPS). |
| SIGQUIT | Завершити | Сигнал посилає всім процесам поточної групи при натисканні клавіш <CTRL>+<\>. |
| SIGSTOP | Зупинити | Сигнал відправляється всім процесам поточної групи при натисканні користувачем клавіш <CTRL>+<Z>. Одержання сигналу викликає останов виконання процесу. |
| SIGSYS | Завершити + core | Сигнал відправляється ядром при спробі здійснення процесом неприпустимого системного виклику. |
| SIGTERM | Завершити | Сигнал звичайно представляє свого роду попередження, що процес незабаром буде знищений. Цей сигнал дозволяє процесу відповідним чином “підготуватися до смерті” - видалити тимчасові файли, завершити необхідні транзакції й т.д. Команда kill за замовчуванням відправляє саме цей сигнал |

| | | |
|---|---|---|
| SIGTTIN | Зупинити | Сигнал генерується ядром (драйвером керуючого термінала) при спробі процесу фонової групи здійснити читання з керуючого термінала. |
| SIGTTOU | Зупинити | Сигнал генерується ядром (драйвером термінала) при спробі процесу фонової групи здійснити запис на керуючий термінал. |
| SIGUSR1 | Завершити | Сигнал призначений для прикладних завдань як найпростіший засіб міжпроцесної взаємодії. |
| SIGUSR2 | Завершити | Сигнал призначений для прикладних завдань як найпростіший засіб міжпроцесної взаємодії. |

Немаловажну роль у житті процесів грає також планувальник – це частина ядра, відповідальна за многозадачность системи. Адже в одиницю часу на одному процесорі може виконуватися тільки одне завдання. Саме планувальник визначає, який із запущених процесів першим буде виконуватися, який другим. Для цього в кожного процесу існує ще один параметр, називаний пріоритетом. Для того, щоб подивитися пріоритет процесів, нам необхідно використовувати вже знайому команду ps з параметром -l (long - розширений вивід):

```
[gserg@WebMedia gserg]$ ps -l
F S UID PID PPID C PRI NI ADDR SZ WCHAN TTY TIME CMD
0 S 500 1554 1553 0 75 0 - 1135 wait4 pts/1 00:00:00 bash
0 R 500 1648 1554 0 81 0 - 794 - pts/1 00:00:00 ps
[gserg@WebMedia gserg]$
```

Під час своєї роботи, планувальник у першу чергу ставить на виконання завдання з меншим пріоритетом. Так, пріоритетом 0, володіють тільки критичні системні завдання, а негативним пріоритетом - процеси ядра. Завданням з більшим пріоритетом дістається менше процесорного часу й тому, працюють вони, як правило, повільніше, і споживають набагато менше системних ресурсів.

### Зміна пріоритету виконуваного процесу

Залишається тільки вирішити питання, а чи може користувач управляти процесами й системними параметрами? Звичайно може! Для цього в Linux є набір інструментів, що дозволяють змінювати пріоритет процесу, посилати процесам сигнали.

**nice -n command** - дозволяє змінювати пріоритет, з яким буде виконуватися процес після запуску. Без вказівки команди command видає поточний пріоритет роботи. n за замовчуванням дорівнює 10. Діапазон пріоритетів розташований від -20 (найвищий пріоритет) до 19 (найменший).