Операційні системи. Практикум

2. Процеси в операційній системі UNIX

План заняття

Поняття процесу в UNIX. Його контекст

Операційна система UNIX заснована на використанні концепції процесів, що обговорювалася на лекції. Контекст процесу складається з користувальницького контексту та контексту ядра, як зображено на рис. 2.1.

Під користувальницьким контекстом процесу розуміють код і дані, розташовані в адресному просторі процесу. Всі дані розділяються на:

- ініціалізовані незмінні дані (наприклад, константи);
- ініціалізовані змінювані дані (всі змінні, початкові значення яких привласнюються на етапі компіляції);
- неініціалізовані змінювані дані (всі статичні змінні, котрим не привласнені початкові значення на етапі компіляції);
- стек користувача;
- дані, розташовані в пам'яті, що динамічно виділяється (наприклад, за допомогою стандартних бібліотечних функцій C malloc(), calloc(), realloc()).

Виконуваний код і початкові дані для ініціалізації становлять вміст файлу програми, що виконується в контексті процесу. Користувальницький стек застосовується при роботі процесу в користувальницькому режимі (user-mode).

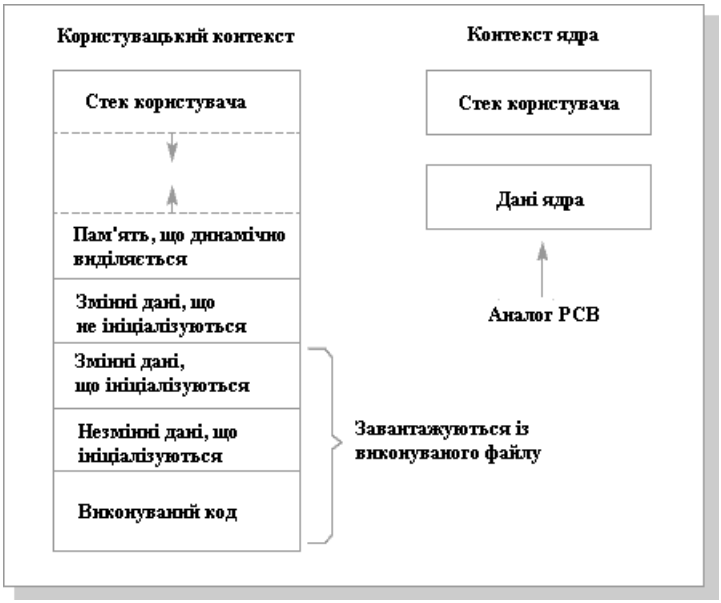

Рис. 2.1. Контекст процесу в UNIX

Під поняттям "контекст ядра" поєднуються системний контекст і регістровий контекст, розглянуті на лекції. Ми будемо виділяти в контексті ядра стек ядра, що використовується при роботі процесу в режимі ядра (kernel mode), і дані ядра, що зберігаються в структурах, що є аналогом блоку керування процесом – PCB. Склад даних ядра буде уточнено на наступних семінарах. На цьому занятті нам досить знати, що в дані ядра входять: ідентифікатор користувача – UID, ідентифікатор групи користувача – GID, ідентифікатор процесу – PID, ідентифікатор батьківського процесу – PPID.

Ідентифікація процесу

Кожний процес в операційній системі одержує унікальний ідентифікаційний номер – PID (process identificator). При створенні нового процесу операційна система намагається виділити йому вільний номер більший, ніж у процесу, створеного перед ним. Якщо таких вільних номерів не знайдеться (наприклад, ми досягли максимально можливого номера для процесу), то операційна система вибирає мінімальний номер із всіх вільних номерів. В операційній системі Linux виділення ідентифікаційних номерів процесів починається з номера 0, який одержує процес kernel при старті операційної системи. Цей номер згодом не може бути привласнений ніякому іншому процесу. Максимально можливе значення для номера процесу в Linux на базі 32-розрядних процесорів Intel становить 2^{31}-1.

Стан процесу. Коротка діаграма станів

Модель станів процесів в операційній системі UNIX є деталізацією моделі станів, розглянутої в лекційному курсі. Коротка діаграма станів процесів в операційній системі UNIX зображена на рис. 2.2.

Рис. 2.2. Скорочена діаграма стану процесу в UNIX

Як ми бачимо, стан процесу **виконання** розщепився на два стани: виконання в режимі ядра і виконання в режимі користувача. У стані виконання в режимі користувача процес виконує прикладні інструкції користувача. У стані виконання в режимі ядра виконуються інструкції ядра операційної системи в контексті поточного процесу (наприклад, при обробці системного виклику або переривання). Зі стану виконання в режимі користувача процес не може безпосередньо перейти в стани **очікування**, **готовність** і **закінчив виконання**. Такі переходи можливі тільки через проміжний стан "виконується в режимі ядра" . Також заборонений прямий перехід зі стану **готовність** у стан виконання в режимі користувача.

Наведена вище діаграма станів процесів в UNIX не є повною. Вона показує тільки стани, для розуміння яких досить уже отриманих знань. Мабуть, найбільш повну діаграму станів процесів в операційній системі UNIX можна знайти в книзі [1] (малюнок 6.1.).

Ієрархія процесів

В операційній системі UNIX всі процеси, крім одного, що створюється при старті операційної системи, можуть бути породжені тільки якими-небудь іншими процесами. Як прабатько всіх інших процесів у подібних UNIX системах можуть виступати процеси з номерами 1 або 0. В операційній системі Linux таким родоначальником, що існує тільки при завантаженні системи, є процес kernel з ідентифікатором 0.

Таким чином, всі процеси в UNIX зв'язані відносинами процес-батько - процес-дитина і утворюють генеалогічне дерево процесів. Для збереження цілісності генеалогічного дерева в ситуаціях, коли процес-батько завершує свою роботу до завершення виконання процесу-дитини, ідентифікатор батьківського процесу в даних ядра процесу-дитини (PPID - parent process identificator) змінює своє значення на значення 1, що відповідає ідентифікатору процесу init, час життя якого визначає час функціонування операційної системи. Тим самим процес init як би всиновляє осиротілі процеси. Напевно, логічніше було б заміняти PPID не на значення 1, а на значення ідентифікатора найближчого існуючого процесу-прабатька померлого процесу-батька, але в UNIX чомусь така схема реалізована не була.

Системні виклики getppid() і getpid()

Дані ядра, що перебувають у контексті ядра процесу, не можуть бути прочитані процесом безпосередньо. Для одержання інформації про них процес повинен зробити відповідний системний виклик. Значення ідентифікатора поточного процесу може бути отримане за допомогою системного виклику getpid(), а значення ідентифікатора батьківського процесу для поточного процесу – за допомогою системного виклику getppid(). Прототипи цих системних викликів і відповідні типи даних описані в системних файлах <sys/types.h> і <unistd.h>. Системні виклики не мають параметрів і повертають ідентифікатор поточного процесу та ідентифікатор батьківського процесу відповідно.

```
Системні виклики getpid() і getppid()
```

Прототипи системних викликів

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Опис системних викликів

Системний виклик getpid повертає ідентифікатор поточного процесу.

Системний виклик getppid повертає ідентифікатор процесу-батька для поточного процесу.

Тип даних pid_t є синонімом для одного із цілочисельних типів мови С.

Написання програми з використанням getpid() і getppid()

Як приклад використання системних викликів getpid() і getppid() самостійно напишіть програму, що друкує значення PID і PPID для поточного процесу. Запустіть її кілька разів підряд. Подивіться, як змінюється ідентифікатор поточного процесу. Поясніть спостережувані зміни.

Створення процесу в UNIX. Системний виклик fork()

В операційній системі UNIX новий процес може бути породжений єдиним способом – за допомогою системного виклику fork(). При цьому знову створений процес буде практично повною копією батьківського процесу. У породженого процесу в порівнянні з батьківським процесом (на рівні вже отриманих знань) змінюються значення наступних параметрів:

- ідентифікатор процесу – PID;
- ідентифікатор батьківського процесу – PPID.

Додатково може змінитися поводження породженого процесу стосовно деяких сигналів, про що докладніше буде розглянуто на семінарах 13-14, коли ми будемо говорити про сигнали в операційній системі UNIX.

Системний виклик для породження нового процесу

Прототип системного виклику

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

Опис системного виклику

Системний виклик fork служить для створення нового процесу в операційній системі UNIX. Процес, що ініціював системний виклик fork, прийнято називати батьківським процесом (parent process). Знову породжений процес прийнято називати процесом-дитиною (child process). Процес-дитина є майже повною копією батьківського процесу. У породженого процесу в порівнянні з батьківським змінюються значення наступних параметрів:

- ідентифікатор процесу;
- ідентифікатор батьківського процесу;
- час, що залишився до одержання сигналу SIGALRM;
- сигнали, що очікували доставки батьківському процесу, не будуть доставлятися породженому процесу.

При однократному системному виклику повернення з нього може відбутися двічі: один раз у батьківському процесі, а другий раз у породженому процесі. Якщо створення нового процесу відбулося успішно, то в породженому процесі системний виклик поверне значення 0, а в батьківському процесі – позитивне значення, рівне ідентифікатору

процесу-дитини. Якщо створити новий процес не вдалося, то системний виклик поверне в його процес, що ініціював, негативне значення.

Системний виклик fork є єдиним способом породити новий процес після ініціалізації операційної системи UNIX.

У процесі виконання системного виклику fork() породжується копія батьківського процесу і повернення із системного виклику буде відбуватися вже як у батьківському, так і в породженому процесах. Цей системний виклик є єдиним, котрий викликається один раз, а при успішній роботі повертається два рази (один раз у процесі-батьку і один раз у процесі-дитині)! Після виходу із системного виклику обидва процеси продовжують виконання регулярного користувацького коду, що розміщений за системним викликом.

Виконання програми з fork() з однаковою роботою батька (предка) та дитини (нащадка)

Для ілюстрації сказаного давайте розглянемо наступну програму:

```
/* Програма 02-1.c - приклад створення нового процесу з однаковою
   роботою процесів дитини та батька */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t pid, ppid;
    int a = 0;
    (void)fork();
    /* При успішному створенні нового процесу
       з цього місця псевдопаралельно
       починають працювати два процеси: старий
       і новий */
    /* Перед виконанням наступного виразу
       значення змінної a в обох процесах
       дорівнює 0 */
    a = a+1;
    /* Довідаємося ідентифікатори поточного і
       батьківського процесу (у кожному з
       процесів !!!) */
    pid = getpid();
    ppid = getppid();
    /* Друкуємо значення PID, PPID і обчислене
       значення змінної a (у кожному з
       процесів !!!) */
    printf("My pid = %d, my ppid = %d,
      result = %d\n", (int)pid, (int)ppid, a);
    return 0;
}
```

Лістинг 2.1. Програма 02-1.c – приклад створення нового процесу з однаковою роботою процесів дитини та батька.

Наберіть цю програму, відкомпілюйте її та запустіть на виконання (найкраще це робити не з оболонки mc, тому що вона не дуже коректно очищує буфери вводу-виводу). Проаналізуйте отриманий результат.

Системний виклик fork() (продовження)

Для того щоб після повернення із системного виклику fork() процеси могли визначити, хто з них є дитиною, а хто батьком, і, відповідно, по-різному організувати своє поводження, системний виклик повертає в них різні значення. При успішному створенні нового процесу в

батьківський процес повертається позитивне значення, що дорівнює ідентифікатору процесу-дитини. У процес-нащадок повертається значення 0. Якщо з якої-небудь причини створити новий процес не вдалося, то системний виклик поверне в процес що ініціював його, значення -1. Таким чином, загальна схема організації різної роботи процесу-дитини і процесу-батька виглядає так:

```
pid = fork();
if(pid == -1){
    ...
    /* помилка */
    ...
} else if (pid == 0){
    ...
    /* дитина */
    ...
} else {
    ...
    /* батько */
    ...
}
```

Написання, компіляція і запуск програми з використанням виклику fork() з різним поводженням процесів дитини та батька

Змініть попередню програму з fork() так, щоб батько і дитина робили різні дії (які – не важливо).

Завершення процесу. Функція exit()

Існує два способи коректного завершення процесу в програмах, написаних мовою С. Перший спосіб ми використовували дотепер: процес коректно завершувався по досягненні кінця функції main() або при виконанні оператора return у функції main(), другий спосіб застосовується при необхідності завершити процес у якому-небудь іншому місці програми. Для цього використовується функція exit() зі стандартної бібліотеки функцій для мови С. При виконанні цієї функції відбувається скидання всіх частково заповнених буферів вводу-виводу із закриттям відповідних потоків, після чого ініціюється системний виклик припинення роботи процесу і переведення його в стан **закінчив виконання**.

Повернення з функції в поточний процес не відбувається і функція нічого не повертає.

Значення параметра функції exit() – коду завершення процесу – передається ядру операційної системи і може бути потім одержано процесом, що породив процес, який завершився. Насправді при досягненні кінця функції main() також неявно викликається ця функція зі значенням параметра 0.

Функція для нормального завершення процесу

Прототип функції

```
#include <stdlib.h>
void exit(int status);
```

Опис функції

Функція exit служить для нормального завершення процесу. При виконанні цієї функції відбувається скидання всіх частково заповнених буферів вводу-виводу із закриттям відповідних потоків (файлів, pipe, FIFO, сокетів), після чого ініціюється системний виклик припинення роботи процесу і переводу його в стан закінчив виконання.

Повернення з функції в поточний процес не відбувається, і функція нічого не повертає.

Значення параметра status – коду завершення процесу – передається ядру операційної системи і може бути потім одержане процесом, що породив процес, який

завершився. При цьому використовуються тільки молодші 8 біт параметра, тому для коду завершення допустимі значення від 0 до 255. За домовленістю, код завершення 0 означає безпомилкове завершення процесу.

Якщо процес завершує свою роботу раніше, ніж його батько, і батько явно не вказав, що він не хоче одержувати інформацію про статус завершення породженого процесу (про це буде розказано докладніше на семінарах 13–14 при вивченні сигналів), то процес, що завершився, не зникає із системи остаточно, а залишається в стані **закінчив виконання** або до завершення процесу-батька, або до того моменту, коли батько одержить цю інформацію. Процеси, що перебувають у стані **закінчив виконання**, в операційній системі UNIX прийнято називати процес-зомбі (zombie, defunct).

Параметри функції main() у мові С. Змінні середовища і аргументи командного рядка

У функції main() в мові програмування С існує три параметри, які можуть бути передані їй операційною системою. Повний прототип функції main() виглядає в такий спосіб:

```
int main(int argc, char *argv[], char *envp[]);
```

Перші два параметри при запуску програми на виконання командним рядком дозволяють довідатися повний зміст командного рядка. Весь командний рядок розглядається як набір слів, розділених пробілами. Через параметр argc передається кількість слів у командному рядку, в якому була запущена програма. Параметр argv є масивом покажчиків на окремі слова. Так, наприклад, якщо програма була запущена командою

```
a.out 12 abcd
```

то значення параметра argc буде дорівнювати 3, argv[0] буде вказувати на ім'я програми – перше слово – "a.out", argv[1] – на слово "12", argv[2] – на слово "abcd". Оскільки ім'я програми завжди присутнє на першому місці в командному рядку, то argc завжди більше 0, а argv[0] завжди вказує на ім'я запущеної програми.

Аналізуючи в програмі вміст командного рядка, ми можемо передбачити її різне поводження залежно від слів, що розміщуються за іменем програми. Таким чином, не вносячи змін у текст програми, ми можемо змусити її працювати по-різному від запуску до запуску. Наприклад, компілятор gcc, викликаний командою gcc 1.c буде генерувати виконуваний файл з іменем a.out, а при виклику командою gcc 1.c -o 1.exe – файл із іменем 1.exe.

Третій параметр – envp – є масивом покажчиків на параметри оточуючого середовища процесу. Початкові параметри оточуючого середовища процесу задаються в спеціальних конфігураційних файлах для кожного користувача і встановлюються при вході користувача в систему. Надалі вони можуть бути змінені за допомогою спеціальних команд операційної системи UNIX. Кожний параметр має вигляд: змінна=рядок. Такі змінні використовуються для зміни довгострокового поводження процесів, на відміну від аргументів командного рядка. Наприклад, задання параметра TERM=vt100 може говорити процесам, що здійснюють вивід на екран дисплея, що працювати їм доведеться з терміналом vt100. Міняючи значення змінної середовища TERM, наприклад на TERM=console, ми повідомляємо таким процесам, що вони повинні змінити своє поводження і здійснювати вивід на системну консоль.

Розмір масиву аргументів командного рядка у функції main() ми одержували в якості її параметра. Оскільки для масиву посилань на параметри оточуючого середовища такого параметра немає, то його розмір визначається іншим способом. Останній елемент цього масиву містить покажчик NULL.

Написання, компіляція і запуск програми, що роздруковує аргументи командного рядка і параметри середовища

Як приклад самостійно напишіть програму, що роздруковує значення аргументів командного рядка і параметрів оточуючого середовища для поточного процесу.

Зміна користувальницького контексту процесу. Сімейство функцій для системного виклику exec()

Для зміни користувальницького контексту процесу застосовується системний виклик exec(), який користувач не може викликати безпосередньо. Виклик exec() заміняє користувальницький контекст поточного процесу на вміст деякого виконуваного файлу і встановлює початкові значення регістрів процесора (в тому числі встановлює програмний лічильник на початок програми, що завантажується). Цей виклик вимагає для своєї роботи задання імені виконуваного файлу, аргументів командного рядка і параметрів оточуючого середовища. Для здійснення виклику програміст може скористатися однією із шести функцій: execlp(), execvp(), execl() і, execv(), execle(), execve(), що відрізняються одна від одної поданням параметрів, необхідних для роботи системного виклику exec(). Взаємозв'язок зазначених вище функцій зображено на рис. 2.3.

Рис. 2.3. Взаємозв'язок різних функцій для виконання системного виклику exec()

Функції зміни користувальницького контексту процесу

Прототипи функцій

```
#include <unistd.h>
int execlp(const char *file,
    const char *arg0,
... const char *argN,(char *)NULL)
int execvp(const char *file, char *argv[])
int execl(const char *path,
    const char *arg0,
... const char *argN,(char *)NULL)
int execv(const char *path, char *argv[])
int execle(const char *path,
    const char *arg0,
... const char *argN,(char *)NULL,
    char * envp[])
int execve(const char *path, char *argv[],
    char *envp[])
```

Опис функцій

Для завантаження нової програми в системний контекст поточного процесу використовується сімейство взаємозалежних функцій, що відрізняються одна від одної формою подання параметрів.

Аргумент file є покажчиком на ім'я файлу, що повинен бути завантажений. Аргумент path – це покажчик на повний шлях до файлу, що повинен бути завантажений.

Аргументи arg0, ..., argN є покажчиками на аргументи командного рядка. Відмітимо, що аргумент arg0 повинен вказувати на ім'я завантажуваного файлу. Аргумент argv є

масивом покажчиків на аргументи командного рядка. Початковий елемент масиву повинен вказувати на ім'я завантажуваної програми, а закінчуватися масив повинен елементом, що містить покажчик NULL.

Аргумент envp є масивом покажчиків на параметри оточуючого середовища, задані у вигляді рядків "змінна=рядок". Останній елемент цього масиву повинен містити покажчик NULL.

Оскільки виклик функції не змінює системний контекст поточного процесу, завантажена програма успадкує від її процесу, що завантажив, наступні атрибути:

- ідентифікатор процесу;
- ідентифікатор батьківського процесу;
- груповий ідентифікатор процесу;
- ідентифікатор сеансу;
- час, що залишився до виникнення сигналу SIGALRM;
- поточну робочу директорію;
- маску створення файлів;
- ідентифікатор користувача;
- груповий ідентифікатор користувача;
- явне ігнорування сигналів;
- таблицю відкритих файлів (якщо для файлового дескриптора не встановлювалася ознака "закрити файл при виконанні exec()").

У випадку успішного виконання повернення з функцій у програму, що здійснила виклик, не відбувається, а керування передається завантаженій програмі. У випадку невдалого виконання в програму, що ініціювала виклик, повертається негативне значення.

Оскільки системний контекст процесу при виклику exec() залишається практично незмінним, більшість атрибутів процесу, доступних користувачеві через системні виклики (PID, UID, GID, PPID і інші, зміст яких стане зрозумілий у міру поглиблення наших знань на подальших заняттях), після запуску нової програми також не змінюється.

Важливо розуміти різницю між системними викликами fork() **і** exec()**. Системний виклик** fork() **створює новий процес, у якого користувальницький контекст збігається з користувальницьким контекстом процесу-батька. Системний виклик** exec() **змінює користувальницький контекст поточного процесу, не створюючи новий процес.**

Виконання програми з використанням системного виклику exec()

Для ілюстрації використання системного виклику exec() давайте розглянемо наступну програму

```
/* Програма 02-2.c, що змінює користувацький
   контекст процесу ( яка запускає іншу програму) */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char *argv[],
         char *envp[]){
/* Ми будемо запускати команду cat з аргументом
   командного рядка 02-2.c без зміни
   параметрів середовища, тобто фактично виконувати
   команду "cat 02-2.c", яка повинна видати
   вміст даного файлу на екран. Для
   функції execle як ім'я програми
   ми вказуємо її повне ім'я зі шляхом від
   кореневої директорії -/bin/cat.
   Перше слово в командному рядку в нас
   повинне збігатися з іменем програми,яка
   запускається. Друге слово в командному рядку
   - це ім'я файлу, вміст якого ми
```

```
   хочемо роздрукувати. */
(void) execle("/bin/cat", "/bin/cat",
              "03-2.c", 0, envp);
/* Сюди попадаємо тільки при
   виникненні помилки */
printf("Error on program start\n");
exit(-1);
return 0;     /* Ніколи не виконується, потрібний
                 для того, щоб компілятор не
                 видавав warning */
}
```

Лістинг 2.2. Програма 02-2.c, що змінює користувальницький контекст процесу

Відкомпілюйте її і запустіть на виконання. Оскільки при нормальній роботі буде роздруковуватися вміст файлу з іменем 02-2.c, такий файл при запуску повинен бути присутнім у поточній директорії (найпростіше записати вихідний текст програми під цим іменем). Проаналізуйте результат.

Написання, компіляція і запуск програми для зміни користувальницького контексту в породженому процесі

Для закріплення отриманих знань модифікуйте програму, створену при виконанні завдання розділу "Написання, компіляція і запуск програми з використанням виклику fork() з різним поводженням процесів дитини та батька" так, щоб породжений процес запускав на виконання нову (довільну) програму.