Процеси

Процес – це програмне середовище, яке створює операційна система у якому виконується програма користувача. Для кожної програми створюються хоча б один процес. Усі процеси ізольовані один від одного. Це означа, що не має можливості отримати доступ до адресного простору іншого процесу.

При створенні процесу операційна система назначає йому унікальний номер – ідентификатор процесу **pid**. Для керування процесами **ОС** підтримує структуру данних, яка зберігає поточний стан процесу. Така структура має назву ***контескт процесу***.

Створення процесу

Для створення процесу слід виконати системний виклик **fork()**, який створює процес. Такий процес ми будемо називати дочірним, а процес, який викликав **fork()** будемо називати батьківським:

```
#include <unistd.h>
int fork()
```

Функція повертає **-1**, якщо створити процес не вдалося, **pid** дочірного процесу для батьківського, та **0** (нуль) для дочірного процесу.

Слід пам’ятати, що дочірній процес є копією батькіського процесу. Однак, деякі параметри контексту, такі як ідентификатор, час використання процесора, індівідуальні.

Завершення процесу

Для завершення процесу існує системний виклик **_exit()**:

```
#include <unistd.h>
void _exit(int status);
```

Оскільки системний виклик **_exit()** тільки завершує поточний процес і не виконує ніяких додаткових дій (закриття файлів, потоків), то для практичного застосування є не зручним. Більш зручним є використання бібліотечної функції **exit()**:

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

Функція у якості параметра приймає значення змінної **status**.

При завершенні процесу процес генерує сигнал **SIGCHLD**, який може бути оброблений батьківьским процесом. По замовченню, батьківський процес, “почувши” цей сигнал не виконє ні яких дій. Дія, яка виконується процесом при перехопленні сигналу називається **диспозицією сигналу**. Для визначення необхідної диспозиції сигналу, процес повинен викликати системній виклик **signal()**.

Стан процесу

Процес може бути у одному з п’яти станів: **Створення (Новий) (new)**, **Готовність (ready)**, **Виконання (running)**, **Очикування (Блокований) (waiting)**, **Завершення (terminated)**. Операційна система надає процесу той чи іншій стан. Для цього існує так званий планувальник процесів.

Перехоплення сігналу SIGCHLD

Щоб визначити диспозицію сигналу слід створити функцію **handler()**, яка і буде виконана при отриманні сигналу **signum**:

```
void handler(int) {
  // Необхідні дії
}
```

Та викликати функцію ядра ОС **signal()**:

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Наприклад, при отриманні сигналу **SIGINT** процес виводить повідомлення “***Hello, world!***”:

```
#include <unistd.h> // write()
#include <signal.h> // signal()
#include <stdlib.h> // exit()

void handle(int) {
  write(STDOUT_FILENO, “Hello, world!”, 13);
}

int main() {
  signal(SIGINT, handle); // Визначення диспозиції сигналу
  // Подальші команди програми
  exit(0);
}
```

Щоб повернути диспозицію сигналу, яка встановлюється системою по замовченню, параметр **handle** слід вказати як **SIG_DFL**, а щоб процес ігнорував сигнал, то **SIG_IGN**. Окрім системного виклику **signal()** для визначення диспозіції сигналу існує ще одна функція ядра операційної системи **sigaction()**.

Ця функція вважається більш надійною. Розглянемо перевизначення диспозіції сигналу за допомогою цієї функції. Для її використання необхідно розглянути системну структуру **sigaction**, яка використовується системним викликом **sigaction()** (спрощена версія).

```
struct sigaction {
  void (*sa_handler)(int);   // Обробник сигналу
  sigset_t sa_mask;          // Маска сигналів
  int sa_flags;              // Прапорці
};
```

Основні властивості (поля):

sa_handler Указатель на функцию, которая вызывается при получении сигнала. Можно также задать **SIG_IGN** (игнорировать) или **SIG_DFL** (поведение по умолчанию).

sa_mask Набор сигналов, которые будут автоматически заблокированы во время выполнения обработчика. Это предотвращает прерывание обработчика другими сигналами.

sa_flags Флаги, которые изменяют поведение обработки сигнала

Найбільш відомі прапорці:

SA_RESTART Автоматически перезапускает прерванные системные вызовы после обработки сигнала.

SA_SIGINFO Указывает, что вместо **sa_handler** нужно использовать **sa_sigaction**.

SA_NOCLDWAIT Используется с **SIGCHLD**, чтобы предотвратить образование зомби-процессов.

SA_NODEFER Не блокирует сигнал, который сейчас обрабатывается.

SA_RESETHAND После одного срабатывания обработчика сигнал возвращается к поведению по умолчанию.

Продемоструємо використання функції **sigaction()** для перехоплення сигналу **SIGINT** та його обробка. Програма визначає диспозіцію сигналу і заходить у біскінечний цикл. Якщо програма отримає сигнал **SIGINT**, наприклад якщо натиснути комбінацію **CTRL + C**, то програма завершує роботу.

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

// Обробник сигналу
void handle_sigint(int sig) {
  printf("\n[INFO] Перехоплен сигнал SIGINT (номер %d)\n", sig);
  printf("[INFO] Завершення програми\n");
  exit(0);
}

int main() {
  // Створення структури
  struct sigaction sa;
  // Підготовка структури
  sigemptyset(&sa.sa_mask);          // Обнулення наборів сигналів (ініціалізація)
  sa.sa_handler = handle_sigint;
  sa.sa_flags = 0;
  // Встановлення обробника для SIGINT
  if (sigaction(SIGINT, &sa, NULL) == -1) {
    perror("sigaction");
    return 1;
  }
  printf("Натисніть Ctrl+C для завершення...\n");
  // Безкінечний цикл, очикуємо сигнал
  while (1) {
    printf("Працюю...\n");
    sleep(1);
  }
  return 0;
}
```

Зомбі процеси

Процес, який викликав функцію **exit()** завершує своє існування, та переходит у стан **зомбі**. У цьому стані усі ресурси, які були задіяні процесом звільняються, процесорного часу процес більше не використовує, але контекст процесу залишається. За знищення контексту завершенного процесу відповідає його батьківьский процес, який повинен викликати відповідну функцію ядра системи, та якщо цього не зробити, то завершенний процес залишиться у стані **зомбі**, що може призвести до неможливості створювати нові процеси (у випадку великої кількості зомбі процесів).

Для знищення контексту завершенного процесу існує група системних викліків **wait()**.

Знищення контексту процесу

Знищити контекст процесу повинен батьківьский процес. Для цього він повинен викликати одну з функцій ядра ОС:

```
#include <sys/wait.h>
pid_t wait(int *wstatus);
pid_t waitpid(pid_t pid, int *wstatus, int options);
pid_t wait3(int *wstatus, int options, struct rusage *usage);
pid_t wait4(pid_t pid, int *wstatus, int options,
                                        struct rusage *usage);
```

Параметр **options** дорівнює нулю, структура **rusage** зберігає обсяг використуваних ресурсів завершенним процесом. Структура описана у ядрі ОС як:

```
struct rusage {
   struct timeval ru_utime;   /* user CPU time used */
   struct timeval ru_stime;   /* system CPU time used */
   long ru_maxrss;            /* maximum resident set size */
```

```
    long ru_ixrss;          /* integral shared memory size */
    long ru_idrss;          /* integral unshared data size */
    long ru_isrss;          /* integral unshared stack size */
    long ru_minflt;         /* page reclaims (soft page faults) */
    long ru_majflt;         /* page faults (hard page faults) */
    long ru_nswap;          /* swaps */
    long ru_inblock;        /* block input operations */
    long ru_oublock;        /* block output operations */
    long ru_msgsnd;         /* IPC messages sent */
    long ru_msgrcv;         /* IPC messages received */
    long ru_nsignals;       /* signals received */
    long ru_nvcsw;          /* voluntary context switches */
    long ru_nivcsw;         /* involuntary context switches */
};
```

Щоб отримати обсяг використаних ресурсів у будь який час, процес може викликати системну функцію **getrusage()**:

```
#include <sys/resource.h>
int getrusage(int who, struct rusage *usage);
```

Параметр **who** визначає ресурси якого процесу треба отримати:

0 – поточного

pid – по вказаному ідентифікатору процесу

Запуск зовнішньої програми

Для запуску зовнішньої програми у адресному просторі поточного процесу слід використовувати бібліотечні функції групи **exec()**:

```
#include <unistd.h>
int execl(const char *pathname, const char *arg, /*, (char *) NULL */);
int execlp(const char *file, const char *arg, /*, (char *) NULL */);
```

```
int execle(const char *pathname, const char *arg, /*,
                                (char *) NULL, char *const envp[] */);
int execv(const char *pathname, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[], char *const envp[]);
```

Слід пам’ятати, что функція **exec()** заміщує адресний простір поточного процесу кодом та даними нової програми, яка вказана у параметрі **file**.

Поля у файлі /proc/<pid>/stat

Файл **`/proc/<pid>/stat`** в Linux містить інформацію про процес з ідентифікатором PID (Process ID). Цей файл є частиною віртуальної файлової системи **`/proc`**, яка надає інформацію про процеси та систему в реальному часі. У файлі **`/proc/<pid>/stat`** міститься багато даних, що стосуються процесу, і вони представлені у вигляді одного рядка, де різні поля розділені пробілами:

1. **pid**: Ідентифікатор процесу.
2. **comm**: Назва виконуваного файлу (в дужках).
3. **state**: Стан процесу (наприклад, виконується, спить тощо). Основні стани:
   - **R**: Виконується
   - **S**: Спить
   - **D**: Очікує в незаперечному стані (в основному через диск)
   - **T**: Зупинений
   - **Z**: Зомбі
   - **X**: Мертвий
4. **ppid**: Ідентифікатор батьківського процесу (PID процесу, що створив цей).
5. **pgrp**: Ідентифікатор групи процесів.
6. **session**: Ідентифікатор сесії.
7. **tty_nr**: Терминал (TTY), асоційований з процесом (якщо є).
8. **tpgid**: Ідентифікатор групи процесів, що володіє терміналом.
9. **flags**: Прапори процесу.
10. **minflt**: Кількість незначних помилок сторінки.
11. **cminflt**: Кількість незначних помилок сторінки у дочірніх процесів.

12. **majflt**: Кількість значних помилок сторінки.
13. **cmajflt**: Кількість значних помилок сторінки у дочірніх процесів.
14. **utime**: Час у користувацькому режимі, витрачений процесом (в тактах).
15. **stime**: Час у ядерному режимі, витрачений процесом (в тактах).
16. **cutime**: Загальний час у користувацькому режимі для всіх дочірніх процесів (в тактах).
17. **cstime**: Загальний час у ядерному режимі для всіх дочірніх процесів (в тактах).
18. **priority**: Пріоритет процесу.
19. **nice**: «Ввічливість процесу» - Значення процесу, яке впливає на його пріоритет планування.
20. **num_threads**: Кількість потоків у процесі.
21. **itrealvalue**: (Заархівоване) Час в джіфах до наступного сигналу SIGALRM для процесу.
22. **starttime**: Час запуску процесу (в тактах з моменту завантаження системи).
23. **vsize**: Розмір віртуальної пам'яті в байтах.
24. **rss**: Розмір резидентної пам'яті (фізична пам'ять, яку процес використовує, у сторінках).
25. **rlim**: Поточний ліміт для розміру віртуальної пам'яті процесу (в байтах).
26. **startcode**: Адреса початку сегмента коду процесу.
27. **endcode**: Адреса кінця сегмента коду процесу.
28. **startstack**: Адреса початку стеку процесу.
29. **endstack**: Адреса кінця стеку процесу.
30. **kstkesp**: Поточне значення регістра ESP (вказівник на стек) в ядерному режимі.
31. **kstkeip**: Поточне значення регістра EIP (вказівник на інструкцію) в ядерному режимі.

32. **signal**: Очікувані сигнали (бітова маска).
33. **blocked**: Сигнали, що заблоковані (бітова маска).
34. **sigignore**: Сигнали, які ігноруються (бітова маска).
35. **sigcatch**: Сигнали, що ловляться (бітова маска).
36. **wchan**: Адреса функції в ядрі, де процес зараз знаходиться в очікуванні (якщо є).
37. **nswap**: Кількість сторінок, які були виміняні.
38. **cnswap**: Кількість сторінок, які були виміняні дочірніми процесами.
39. **exit_signal**: Сигнал, що буде надісланий батьківському процесу після завершення процесу.
40. **processor**: Процесор, на якому останній раз виконувався процес.
41. **rt_priority**: Пріоритет для реального часу.
42. **policy**: Політика планування (наприклад, SCHED_OTHER, SCHED_FIFO).
43. **delayacct_blkio_ticks**: Час, витрачений на блокові операції вводу/виводу.
44. **guest_time**: Час, витрачений процесом у віртуальній CPU (якщо процес працює в віртуальній машині).
45. **cguest_time**: Час, витрачений дочірніми процесами в віртуальних CPU.