Operation-Systems-Voronezh-State-University

Это дз по курсу Операционных систем

Соколов Пётр 4 группа Информатика и выч техника 2 курс

Получение идентификации процессов

Системные вызовы `getpid`, `getppid` позволяют получить идентификатор текущего или родительского процесса.

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

Функция `getpid` возвращает идентификатор текущего процесса, а функция `getppid` возвращает идентификатор родительского процесса. Если родительский процесс завершил своё выполнение раньше сыновнего, «родительские права» передаются процессу с номером 1 — процессу init (но это не точно).

Идентификатор процесса - это положительное число в интервале от 1 до некоторого максимального значения. При создании нового процесса система назначает ему новый идентификатор. Когда идентификатор процесса достигнет максимального допустимого значения, счёт начнётся снова с 1 и операционная система назначит идентификатор процесса, не занятый в данный момент каким-либо процессом. Когда операционная система работает достаточно долго, один и тот же идентификатор процесса будет использоваться несколько раз.

создаёт копии самой себя (не уверен) 😳

Системный вызов fork создает новый процесс, адресное пространство которого является точной копией исходного процесса. В том числе, будут скопированы все структуры данных, находящиеся в адресном пространстве исходного процесса, например, буфера ввода-вывода высокоуровневых дескрипторов потока (FILE * или std::ostream).

Например, в следующем фрагменте кода

```
    printf("Hello");
    if (!fork()) {
        exit(0);
    }
```

строка Hello будет выведена дважды как HelloHello. Дело в том, что при создании процесса будет скопирован буфер стандартного потока вывода stdout в адресном пространстве процесса-родителя, который содержит строку Hello. Этот буфер не был записан в файловый дескриптор 1, так как по умолчанию включена либо построчная, либо полная буферизация. Вызов exit(0) в процессе-сыне перед завершением процесса выполнит операцию fflush(stdout), которая выполнит системный вызов write и запишет содержимое буфера на стандартный поток вывода еще раз.

Использование _exit вместо exit снимает эту проблему. _exit - это "чистый" системный вызов, который просто завершает процесс. При этом не выполняются вызовы fflush для дескрипторов потоков, и т. п. Ядро освободит все ресурсы, выделенные процессу при fork, поэтому утечек ресурсов не произойдет. Это особенно важно в библиотечных функциях, которые создают новые процессы, так как библиотечная функция может вызываться в разных программах в разных состояниях программы.

Получения файлового дескриптора процесса

Получить файловый дескриптор для процесса можно с помощью pidfd_open.

```
#include <sys/syscall.h>

#ifndef __NR_pidfd_open
#define __NR_pidfd_open 434   /* System call # on most architectures */
#endif

static int
pidfd_open(pid_t pid, unsigned int flags)
{
    return syscall(__NR_pidfd_open, pid, flags);
}
```

Для pidfd_open еще не реализована поддержка в glibc, поэтому приходится использовать низкоуровневую функцию syscall для системных вызовов напрямую.

Параметр pid - это идентификатор процесса, для которого нужно получить файловый дескриптор, параметр flags должен быть равен 0.

Системный вызов возвращает файловый дескриптор, с помощью которого можно управлять процессом. Для этого файлового дескриптора уже установлен флаг закрытия при exec (O_CLOEXEC). При ошибке возвращается -1, и переменная errno устанавливается в код ошибки.

Использовать этот системный вызов можно следующим образом:

```
    int pidfd = -1;
    pid_t pid = fork();
    if (pid > 0) {
        // в родителе получаем файловый дескриптор для процесса-сына
        pidfd = pidfd_open(pid, 0);
    }
```

Системный вызов pidfd_open вернет файловый дескриптор даже в случае, если процесс-сын успел завершиться и находится в состоянии зомби. Однако если он уже был похоронен и его pid был освобожден для повторного использования, остается возможность получить либо ошибку, либо доступ к совершенно другому процессу, то есть все равно сохраняется окно для race condition. Следует заметить, тем не менее, что это возможно только в специальных случаях, описанных выше в разделе про автоматическое освобождение pid, либо если был выполнен системный вызов семейства wait в другой нити. При использовании в однопоточной программе без переустановки обработчика SIGCHLD этот фрагмент совершенно корректен.

Если нужно гарантированно во всех случаях избежать race condition, следует использовать низкоуровневый системный вызов clone с флагом CLONE_PIDFD.