Архитектура ОС UNIX

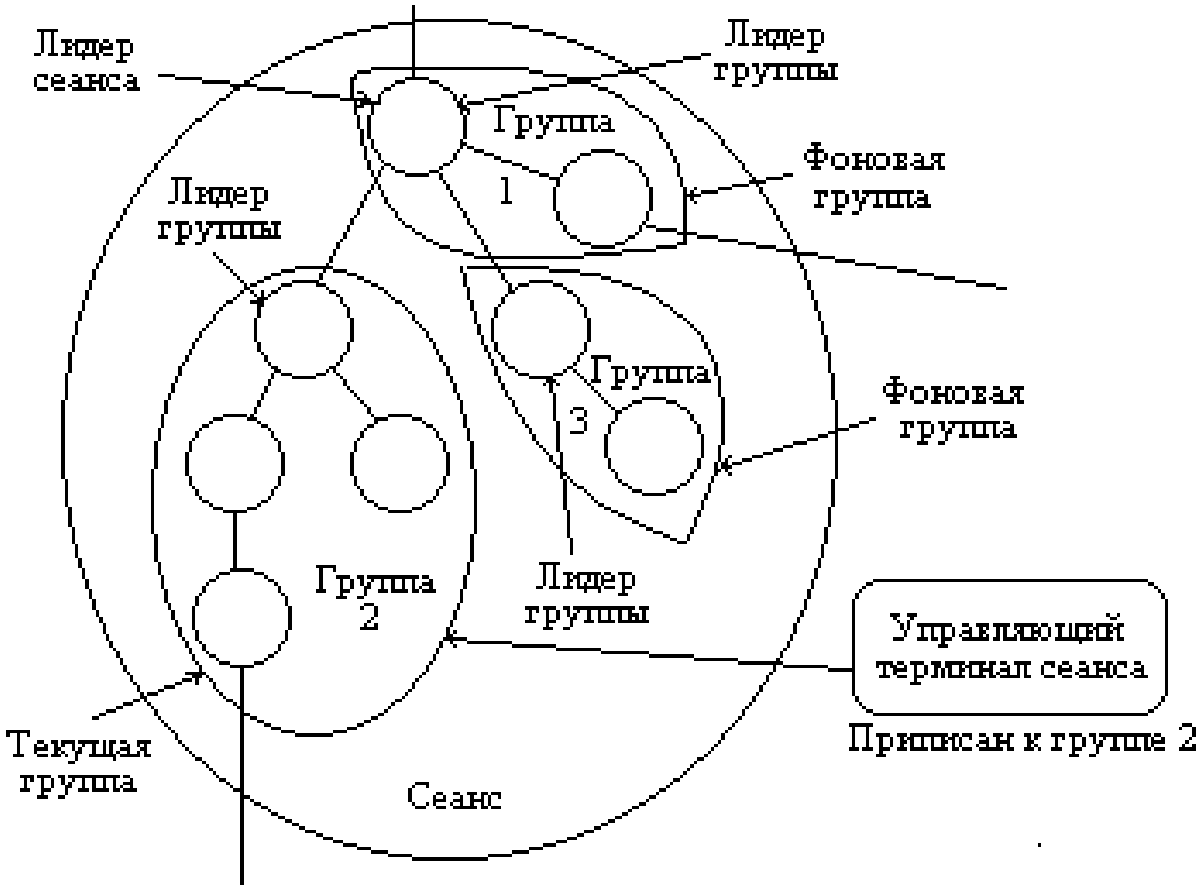
Санкт-Петербургский государственный политехнический университет

24 ноября 2011

Группы процессов

Группа процессов включает в себя один или более процессов и существует, пока в группе присутствует хотя бы один процесс. При рождении нового процесса он попадает в ту же группу процессов, в которой находится его родитель. Процессы могут мигрировать из группы в группу по своему собственному желанию или по желанию другого процесса.

Группы процессов



Группы процессов

В свою очередь, группы процессов объединяются в сеансы, образуя с родственной точки зрения некие кланы семей.
С каждым сеансом, поэтому, может быть связан в системе терминал, называемый управляющим терминалом сеанса, через который обычно и общаются процессы сеанса с пользователем.

Группы процессов

Каждая группа процессов в системе получает свой собственный уникальный номер. Узнать этот номер можно с помощью системного вызова getpgid() .
Для перевода процесса в другую группу процессов, возможно с одновременным ее созданием, применяется системный вызов setpgid(). Перевести в другую группу процесс может либо самого себя, либо свой дочерний процесс.

Группы процессов

Процесс, идентификатор которого совпадает с идентификатором его группы, называется лидером группы.
Одно из ограничений на применение вызовов setpgid() и setpgrp() состоит в том, что лидер группы не может перебраться в другую группу.
Каждый сеанс в системе также имеет свой собственный номер. Для того, чтобы узнать его можно воспользоваться системным вызовом getsid().

Группы процессов

Если сеанс имеет управляющий терминал, то он обязательно приписывается к некоторой группе процессов, входящей в сеанс.
Такая группа процессов называется текущей группой процессов для данного сеанса. Все процессы, входящие в текущую группу процессов могут совершать операции ввода-вывода, используя управляющий терминал.
Все остальные группы процессов сеанса называются фоновыми группами, а процессы, входящие в них фоновыми процессами.

Сигналы

Сигнал — программное прерывание нормальной работы процесса.

Сигналы

События, которые генерируют сигналы, делятся на три основные группы:

- ошибки (деление на 0, неверное обращение к памяти)
- внешние события (получение данных, срабатывание таймера)
- явные запросы (используя функцию kill)

Сигналы

Сигналы могут генерироваться синхронно или асинхронно. Синхронный сигнал относится к некоторому действию в программе и доставляется во время этого действия, например, сигналы об ошибках. Асинхронные сигналы генерируются событиями, неподконтрольными процессу, который их получает, например, сигнал от одной программе к другой.

Сигналы

Системный вызов kill позволяет послать сигнал процессу или группе процессов.

```
int kill(pid_t pid, int sig);
int raise(int sig);
```

В случае успеха системный вызов возвращает 0. Если сигнал посылается группе процессов, вызов kill заканчивается успешно, если сигнал был послан хотя бы одному процессу в группе. В случае неудачи системный вызов возвращает значение -1, а дополнительный код ошибки записывается в переменную errno.

Сигналы

- pid больше нуля, заданный сигнал sig посылается процессу с заданным идентификатором процесса.
- pid равен 0, сигнал sig посылается всем процессам в группе процессов, к которой относится данный процесс.
- pid меньше -1, сигнал sig посылается всем процессам в группе процессов с идентификатором -pid.
- pid равен -1, сигнал sig посылается всем процессам с тем же эффективным идентификатором пользователя, что у текущего процесса.
- pid равен 0, сигнал не посылается, но все возможные ошибки проверяются. Таким образом можно проверить, например, существование процесса с заданным pid.

Сигналы

После генерации сигнала он становится ожидающим доставки, если тип сигнала заблокирован приложением, сигнал может оставаться ожидающим доставки неограниченно долгое время.

Сигналы

Большинство сигналов, может быть игнорировано, обработано или должно быть выполнено действие по умолчанию, за исключением сигналов SIGKILL и SIGSTOP.
Во время обработки сигнала в обработчке данный тип сигнала блокируется.
Если для определенных типов сигналов установлено действие — игнорирование, то сигнал данного типа будет сброшен сразу после генерации.
Если процесс получил сигнал, который ни оьрабатывается, ни игнорируется то выполняется действие по умолчанию.

Сигналы

Сигналы, обозначающие ошибки выполнения процесса, имеют специальное свойство: когда процесс завершается по одному из таких сигналов, ядро записывает на диск дамп памяти (core dump).
Дамп памяти имеет по умолчанию имя core и создаётся в текущем каталоге.

Работа с множествами сигналов

Аргументами многих функций, работающих с сигналами, могут быть множества сигналов, например, множество блокируемых сигналов.

```
int sigemptyset(sigset_t *pset);
int sigfillset(sigset_t *pset);
int sigaddset(sigset_t *pset, int signum);
int sigdelset(sigset_t *pset, int signum);
int sigismember(const sigset_t *pset, int signum);
```

Работа с множествами сигналов

Аргументами многих функций, работающих с сигналами, могут быть множества сигналов, например, множество блокируемых сигналов.

```
int sigemptyset(sigset_t *pset);
int sigfillset(sigset_t *pset);
```

Функция sigemptyset очищает множество сигналов, на которое указывает pset. Пустое множество не содержит ни одного сигнала.
Функция sigfillset полностью заполняет множество сигналов, на которое указывает pset. В получившееся множество включены все сигналы.

Работа с множествами сигналов

```
int sigdelset(sigset_t *pset, int signum);
int sigismember(const sigset_t *pset, int signum);
int sigaddset(sigset_t *pset, int signum);
```

Функция sigaddset добавляет сигнал signum в множество сигналов, на которое указывает pset.
Функция sigdelset удаляет из множества сигналов, на которое указывает pset, сигнал signum.
Функция sigismember проверяет, присутствует ли в множестве сигналов, на которое указывает pset, сигнал с номером signum.

Обработка сигналов

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

typedef void (*sighandler_t)(int signum);
sighandler_t signal(int signum, sighandler_t
    handler);
```

Первый аргумент signum задает номер сигнала, обработку которого нужно изменить. Второй аргумент handler определяет, как будет обрабатываться сигнал, он может принимать следующие значения:

- SIG_DFL устанавливает обработку сигнала на стандартную обработку по умолчанию
- SIG_IGN задает, что сигнал должен игнорироваться.
- задать функцию для обработки сигнала.

Обработка сигналов

```c
#include <stdio.h>
#include <signal.h>
int cnt = 0;
void sigint_handler(int signo)
{
  printf("Ctrl-C pressed\n");
  if (++cnt == 3) {
    signal(SIGINT, SIG_DFL);
    raise(SIGINT);
  }
}
int main(void)
{
  signal(SIGINT, sigint_handler);
  while (1) {
    printf("Some string to print\n");
  }
  return 0;
}
```

Обработка сигналов

Опасность обработки сигналов заключается в том что процесс может быть приостановлен в любой момент времени, например, во время выполнения стандартной функции, что может негативно отразиться на работоспособности программы.
Функции которые могут вызываться из обработчика называются асинхронно-безопасными.
Для решения этой проблемы существует 2 способа:

- Не использовать стандартные функции внутри обраотчика, а устанавливать флаг глобальной переменной.
- Блокировать сигналы на время выполнения стандартной функции.

Функция sigaction

Системный вызов sigaction позволяет установить обработчик сигнала.

```
int sigaction(int signum,
              const struct sigaction *act,
              struct sigaction *oldact);



struct sigaction {
      void (*sa_handler)(int);
      void (*sa_sigaction)(int, siginfo_t *, void
          *);
      sigset_t sa_mask;
      int sa_flags;
      void (*sa_restorer)(void);
};
```

Функция sigaction

Функция sigaction устанавливает обработчик для сигнала signum, который задается в аргументе act, если этот аргумент не равен NULL.
Поле sa_handler задаёт обработчик сигнала.
Поле sa_mask задаёт множество сигналов, которые будут заблокированы на время ра боты обработчика сигнала.
Поле sa_flags позволяет определить режим, в котором будет обрабатываться сигнал.

Функция sigaction

```
struct sigaction orig_sigint_handler;
void sigint_handler(int signo)
{
  flag = 1;
  if (++cnt == 3) {
    sigaction(SIGINT, &orig_sigint_handler);
    raise(SIGINT);
  }
}
int main(void)
{
  struct sigaction sa;
  sa.sa_handler = sigint_handler;
  sigemptyset(&sa.sa_mask);
  sa.sa_flags = SA_RESTART;
  sigaction(SIGINT, &sa, &orig_sigint_handler);
```

Функция sigaction

```
    while (1) {
        if (flag) {
            printf("Ctrl-C pressed\n");
            flag = 0;
        } else {
            printf("Some string to print\n");
        }
    }
    return 0;
}
```

Блокирование сигналов

Временная блокировка сигнала с помощью sigprocmask позволяет предотвратить прерывание нормальной работы процесса в критической секции кода.
Функция sigprocmask позволяет изменить маску сигналов процесса.

Блокирование сигналов

```
int sigprocmask(int mode,
                const sigset_t *pset,
                sigset_t *poldset);
```

Аргумент mode задаёт, какая операция будет выполнена с маской сигналов. Он должен быть равен одному из следующих значений:

- SIG_BLOCK — Множество сигналов, на которое указывает pset, добавляется к маске сигналов процесса.
- SIG_UNBLOCK — Множество сигналов, на которое указывает pset, удаляется из маски сигналов процесса.
- SIG_SETMASK — Маска сигналов копируется из множества сигналов, на которое указывает pset.

Функция sigaction

```
int main(void)
{
  sigset_t blockset;
  sigemptyset(&blockset);
  sigaddset(&blockset, SIGINT);
  signal(SIGINT, sigint_handler);
  while (1) {
    sigprocmask(SIG_BLOCK, &blockset, 0);
    printf("Some string to print\n");
    sigprocmask(SIG_UNBLOCK, &blockset, 0);
  }
  return 0;
}
```

Функция pause

Простейший способ приостановить выполнение процесса до поступления сигнала заключается в использовании функции pause.

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

Функция pause приостанавливает выполнение процесса до поступления сигнала, который не блокируется и не игнорируется.

Функция sigsuspend

Функция sigsuspend служит для ожидания заданного сигнала.

```
int sigsuspend(const sigset_t *pset);
```

Функция заменяет маску сигналов процесса маской, на которую указывает аргумент pset, затем приостанавливает выполнение процесса до поступления сигнала, который либо вызывает завершение процесса, либо обрабатывается процессом.

Функция sigsuspend

```
void sigint_handler(int signo)
{
  flag = 1;
}
int main(void)
{
  sigset_t mask, oldmask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  signal(SIGINT, sigint_handler);
  while (1) {
    sigprocmask(SIG_BLOCK, &mask, &oldmask);
    while (!flag)
      sigsuspend(&oldmask);
    flag = 0;
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
    printf("Ctrl-C pressed\n");
  }
  return 0;
```

Слияние сигналов

Сколько бы раз в процесс не поступил сигнал SIGINT в то время, пока он заблокирован, обработчик сигнала будет вызван после разблокировки не более одного раза.
Ядро не поддерживает очередь сигналов, ожидающих доставки, а хранит для каждого типа сигнала единственный бит, означающий наличие недоставленного сигнала соответствующего типа. Эти биты все вместе образуют маску сигналов процесса, ожидающих доставки.

Слияние сигналов

```
void sigint_handler(int signo)
{
  printf("Ctrl-C␣pressed\n");
}
int main(void)
{
  sigset_t mask;
  sigemptyset(&mask);
  sigaddset(&mask, SIGINT);
  signal(SIGINT, sigint_handler);
  while (1) {
    sigprocmask(SIG_BLOCK, &mask, NULL);
    sleep(2);
    sigprocmask(SIG_UNBLOCK, &mask, NULL);
  }
  return 0;
}
```

Компиляция приложений

Когда разрабатываемая программа состоит более чем из одного исходного .c файла, или когда для получения нужного результата, нужно выполнить много зависящих друг от друга действий, задавать команды компиляции для каждого файла вручную становится очень сложно.

Утилита GNU make

Описание проекта для утилиты make содержится в файле, Makefile, либо makefile.
Утилита make отслеживает зависимости между компонентами проекта, и, если один из файлов был модифицирован, все файлы, которые от него зависят, перекомпилируются.

Утилита GNU make

Определения переменных записываются следующим образом:

```
<имя> = <определение>
```

Для получения значения переменной:

```
$(<имя>) или ${<имя>}
```

Использование переменной означает подстановку текста из определения переменной в точку использования.

Утилита GNU make

В шаблонных правилах, можно использовать специальные переменные, которые раскрываются в зависимости от правила, в котором они стоят.
Переменная $@ раскрывается в имя цели, стоящей в левой части правила.
Переменная $< раскрывается в имя первой зависимости в правой части правила.
Переменная $^ раскрывается в список всех зависимостей в правой части.

Утилита GNU make

Зависимости между компонентами определяются следующим образом:

```
<цель> : <цель1> <цель2> ... <цельn>
```

Где, цель — либо имя файла, либо действие.
Цель, стоящая в левой части правила, считается устаревшей, если необходимо выполнить перекомпиляцию для её обновления. Это происходит в одном из трёх случаев:

- Одна из целей, стоящих в правой части правила, является устаревшей.
- Файл с именем <цель> не существует.
- Файл с именем одной из целей, стоящих в правой части правила, имеет более позднюю дату модификации, чем файл с именем <цель>.

Утилита GNU make

Например, шаблонное правило для зависимостей .o файлов от .c файлов может выглядеть следующим образом:

```
.c.o:
	$(CC) -c $(CFLAGS) $<
```

Утилита GNU make

```
TARGET=test
CC=gcc
CFLAGS=-g -Wall
SOURCES=test.c lib.c
OBJECTS=$(SOURCES:.c=.o)

all: $(TARGET)

clean:
	rm -f *.o $(TARGET)
$(TARGET): $(OBJECTS)
	$(CC) -o $@ $(CFLAGS) $(OBJECTS)
.c.o:
	$(CC) -c $(CFLAGS) $< -o $@

test.o: lib.h
```

Утилита GNU make

```
TARGET=test
CC=gcc
CFLAGS=-g -Wall
SOURCES=test.c lib.c
OBJECTS=$(SOURCES:.c=.o)

all:$(TARGET)
clean:
        rm -f *.o $(TARGET)
$(TARGET): $(OBJECTS)
        $(CC) -o $@ $(CFLAGS) $(OBJECTS)
.c.o:
        $(CC) -c $(CFLAGS) $< -o $@

test.o: lib.h
```

Утилита GNU make

Для вывода зависимостей между файлами полезно использоваьт команду gcc -MM, а потом включить этот файл сборки в исходный makefile.

```
HFILES=a.h b.h c.h d.h # список всех .h файлов
CFILES=a.c b.c c.c d.c # список всех .c файлов
SRCFILES=$(CFILES) $(HFILES)
deps.make: $(SRCFILES)
        gcc -MM $(CFILES) > deps.make
include deps.make
```

Автоматическая генерация make

Для автоматического создания Makefile'ов, существуют основные системы сборки:

- Autoconf/automake
- Cmake
- Scons
- Qmake