UNIX. Введение в ОС

Thursday, September 13, 2018 11:43

Linux - самая популярная UNIX-подобная система. Это самый популярный Open Source - продукт, разрабатываемый в основном децентрализованно. Разработка такой система заняла примерно 73 000 человеко-лет.

Распространённость:

- Смартфоны - 86% (Android)
- Сервера - 60%
- Суперкомпьютеры - 97%
- Встраиваемые системы - 50%
- Нетбуки - 35%
- ПК - от 1% до 5% (+ 12.5% macOS)

UNIX - операционная система, появившаяся в конце 60-х в Bell Labs. Основная особенность **UNIX-подобных систем** - **стандартизация**. Несмотря на многообразие версий UNIX, основой всего семейства являются принципиально одинаковая архитектура и ряд стандартных интерфейов. Стандарт называется **POSIX**

Архитектура UNIX

[IMAGE1]

- Монолитное ядро
- Демоны
- Пользовательские программы

Ядро UNIX

[IMAGE2]

Выделение аппаратно-зависимой части - важное архитектурное решение в UNIX. Эта позволило с относительной легкостью переносить UNIX с платформы на платформу

Файловая система UNIX

[IMAGE3]

Индексный узел

[IMAGE4]

Виртуальная файловая система

[IMAGE5]

Вся работа с файловой системой абстрагируется до уровня простых системных вызовов. Пользователю и разработчику не важен источник данных. Это осуществляется благодаря тому, что ядро содержит драйвера

нужных файловых систем.

Монтирование файловых систем
[IMAGE6]
В UNIX'е файловая система представляется единым деревом с корнем в '/'. При подключении носителя он встраивается в дерево с помощью операции **монтирования**

Стандарт на файловую систему
Все каталоги в UNIX'е стандартизованы.

- '/' - начало
- **'bin'** - основные утилиты
- **'boot'** - ядро ОС
- **'dev'** - все подключенные устройства
- **'etc'** - конфигурационные файлы для установленного ПО
- **'home'** - пользовательский каталог
- **'lib'** - каталог с библиотеками
- **'media'** - точка монтирования по умолчанию
- **'root'** - каталог суперпользователя
- **'sbin'** - утилиты для системного администратора
- **'usr'** - утилиты пользователя
- **'var'** - все логи
- **'temp'** - папка, располагающаяся в оперативной памяти. Обычно сюда пишутся временные файлы текущих процессов

Устройства и драйверы
Все устройства - это файлы, находящиеся в каталоге dev.
/dev/null - специальное псевдоустройство, использующееся при разработке. Все, попадающее туда, пропадает.
/dev/zero - устройство, из которого можно читать нули.

Командная строка

- sh (shell), bash, dash, zsh
- csh (C shell) - Си-подобный синтаксис (не POSIX)
- ash (Aimquist shrll) - для встраиваемых систем

Типы команд

- Внешняя (date, pwd, cd, ls) - обычные программы, находящиеся в bin
- Внутренняя (echo) - команда, которую знает сам интерпретатор
- Псевдоним (alias)

Внутренние команды имеют приоритет над внешними.

Переменные окружения
Это способ задать заранее некоторые данные, доступ к которым есть у любой программы.

- Просмотра переменных - echo $[NAME]
- Установка переменных - export NAME=DATA

Стандартные переменные
- DISPLAY - указывает рабочий дисплей
- EDITOR - редактор по умолчанию
- HOME - домашний каталог
- PATH
- SHEL - имя текущего интерпретатора
- _ - последняя выполненная команда

Стандартные команды

Справочная подсистема

Чтобы узнать информацию о команде, можно использовать
`man [раздел] имя_команды`
Разделы man:
1. Пользовательские утилиты
2. Системные вызовы
3. Библиотечные функции
4. …

Чтобы узнать, где находится команда, можно использовать
`which имя_команды`

Команды ФС
- `pwd` - текущий путь
- `cd новый_путь` - смена каталога
- `ls [путь]` - содержимое каталога
- `mkdir новая_папка` - создание каталога
- …

Каталог '.' - специальный каталог, который ссылается на текущий.
Используется для создания относительных путей.

Команды по работе с текстом

…

Команды по работе с системой
- date - текущая дата
- ps - процессы
- free - показывает оперативную память
- kill - команда для жесткого завершения процесса.

Конвейер - перенаправление вывода одной программы на вход другой программе.

Перенаправление потоков ввода-вывода

0 - поток ввода (0>)
1 - поток вывода (1>, >)
2 - поток вывода ошибок (2>)

Создание скриптов

bash - Тьюринг-полный язык программирования.
Скрипт - обычный файл, у которого в начале может быть указано, с помощью чего его запускать

```
#!/bin/bash
#!/bin/python
```

Безопасность

Права в UNIX делится на три типа:

- Права владельца
- Права группы
- Права всех остальных

	Файл	Каталог
read	Чтение	Чтение списка файлов
write	Запись	Добавление файлов в каталог
execute	Выполнение	

Для изменения прав существует команда

```
chmod права_владельца права_группы права_остальных
chmod rwx r-x r-x
chmod 755
```

Для изменения владельца есть команда `chown`

Особые права

- `sticky bit` - удаление файла/каталога только владельца
- `setuid/setgid bit` - запись файла с правами владельца/группы

Введение

Thursday, October 4, 2018 11:42

Параллельный алгоритм - алгоритм, который может быть реализова

Причины использования параллельных алгоритм

1. Сокращение времени решения прикладных задач
2. Обеспечение решения большего количества задач за единицу времени

Кризис эффективности

Благодаря закону Мура увеличение производительности приводило к уменьшению времени выполнения некоторой задачи. Ответ на пределе тактовой частоты - многопроцессорные системы. Однако последовательные программы не могут работать на многопроцессорных системах быстрее, чем на последовательных. В общем случае автоматически сконвертировать последовательную программу в параллельную невозможно - обычно это требует новых алгоритмов.

Применение:

- Физика высоких энергий
- Климатология - симуляция происходящих в атмосфере явлений
- Генная инженерия
- Банковские транзакции
- Big Data
- Машинное обучение

<u>Параллельное вычисление</u>

- **Одна ЭВМ с одним ядром**
  Компилятор часто уже генерирует параллельный код, тогда, когда он может. Это расширение **SSE** - Streaming SIMD extension. Например, операции с векторами или строками параллелизуются.
- **Одна ЭВМ с несколькими процессорами**
  Примитивы:
  - Процессоры
  - Потоки
  - Средства межпроцессорного взаимодействия (IPC)
  - Примитивы синхронизации

  Технологии: OpenMP, Boost threads, Java threads, Intel TBB, Java.util.concurrent, Fork/Join Framework, CUDA, OpenCL
- **Несколько ЭВМ с несколькими процессорами**
  Примитивы:
  - Сокеты - передача данных по сети

Технологии: MPI, PVM, MapReduce (Hadoop, Spark)

Пример
Необходимо найти все простые числа от 1 до 10^N. Нужно это сделать максимально эффективно (в данном случае - максимально быстро). В примере компьютер двухъядерный

Время выполнения последовательного алгоритма - *27 секунд*.

Простое решение - разделить массив на несколько частей и посчитать каждую часть на каждом ядре, после чего объединить результат.
Время выполнения такого алгоритма - *18 секунд*
Проблема в том, что простые числа распределены неравномерно. Некоторые потоки будут работать гораздо медленнее других.

Оптимизация - создать общий пул для потоков. Каждый поток будет стартовать с 0 и брать значение для проверки из счётчика.
Время - *15 секунд*
Этот алгоритм имеет шанс дать неверный результат, так как счётик - разделяемый ресурс. Чтобы исправить это, нужно обеспечить синхронизацию метода. В Java это - ключевое слово synchronize. Оно гарантирует, что в один момент времени этот метод будет вызван одним потоком.

Накладные расходы параллельного алгоритма

- Синхронизация
  Синхронизация - это всегда системный вызов
- Обмен данными
  Нужно обеспечить передачу данных между потоками
- Дублирование операций
  Чтобы не обмениваться данными лишний раз, иногда дешевле провести одинаковые вычисления в нескольких потоках

Свойства идеального параллельного алгоритма

- Есть возможность одновременного выполнения операций (запас параллелизма)
- Допускает возможность равномерного распределения вычислительных операций между процессорами
- Обладать низкими накладными расходами

Согласно классическому определению алгоритма, любой детерминированный алгоритм представляет собой **последовательность** действий.

Пример

$x = (a + b) * (c + d)$
Последовательный алгоритм

$a + b$
$c + d$
$*$

Параллельный алгоритм

$a + b$	$c + d$
$*$	

Граф алгоритма $G(V, E) -$ это ориентированный граф, вершинами которого являются операции алгоритма.
Каждый ярус графа алгоритма представляет собой итерацию параллельного алгоритма, а каждая вершина - операцию, которая может быть выполнена последовательно.

Показатели эффективности паралельного алгоритма

- **Ускорение**
  Сравнение с последовательной версией самой быстрой известной версии параллельного алгоритма
  $S_p(n) = \frac{T_1(n)}{T_p(n)}$
  Максимальное ускорение - линейный рост производительности при увеличении количества процессоров.
- **Эффективность**
  Показывает долю времени, в течение которого процессор занят временем решения задачи. Максимальная эффективность - $100\% -$ недостижима
- **Стоимость**

<u>Оценка максимального достижимого параллелизма</u>
Закон Амдала

$$S_p = \frac{1}{\alpha + \frac{1 - \alpha}{p}}$$

$\alpha -$ доля последовательного кода, $p -$ число процессовор.

Пример
Есть 5 одинаковых комнат и 5 маляров. 5 маляров покрасят в 5 раз быстрее, чем 1 маляр. $S = \frac{5}{1} = 5$.
Если одна из комнат в два раза больше остальных, то ускорение:

$$S = \frac{1}{\frac{1}{6} + \frac{5}{6 * 5}} = 3$$

Идеальный алгоритм: $S = p, E = 1$. Такой идеальный алгоритм на практике недостижим, поэтому принято считать эффективным такой параллельный алгоритм, эффективность которого, по крайней мере, ограничена какой-либо величиной.

Важное следствие закона Амдала - всегда есть предел ускорения от увеличения количества процессоров. Даже если 95% программы можно распараллелить, максимальный прирост - 20 раз.

Пример. Задача вычисления общей суммы

$$S = \sum_{i=1}^{n} x_i$$

Традиционный алгоритм для решения этой задачи - последовательное суммирование. В этом виде алгоритм невозможно распараллелить - на каждом ярусе одна операция.

Благодаря тому, что сложение ассоциативно, можно применить каскадную схему суммирования. Все числа разбиваются на пары, все пары на каждом этапе суммируются. Общее количество итераций - $k = \log_2 N$.

$\mathrm{K}_{\text{пост}} = n - 1$

$$S_p = \frac{T_1}{T_p} = \frac{n-1}{\log_2 N}$$

Эффективность этого алгоритма стремится к 0 при $n \to \infty$

Модифицированная каскадная схема

Все группы делятся $\log_2 n$, в каждой группе подсчёт происходит последовательно.

$$S = \frac{n-1}{2\log_2 n}$$

$$E = \frac{T_1}{pT_p}$$

$\lim E_p \to 0$ при $n \to \infty$

Литература

1. Воеводин В.В., Воеводин Вл. В. "Параллельные вычисления"
2. Анотов А.С. "Параллельное программирование с использованием технологии OpenMP"
3. Maurice Herithy "The Art of Multiprocessor Programming"

Процессы

Thursday, October 18, 2018 09:56

Процессы

Программа - бинарный файл
Процесс - программа в стадии исполнения

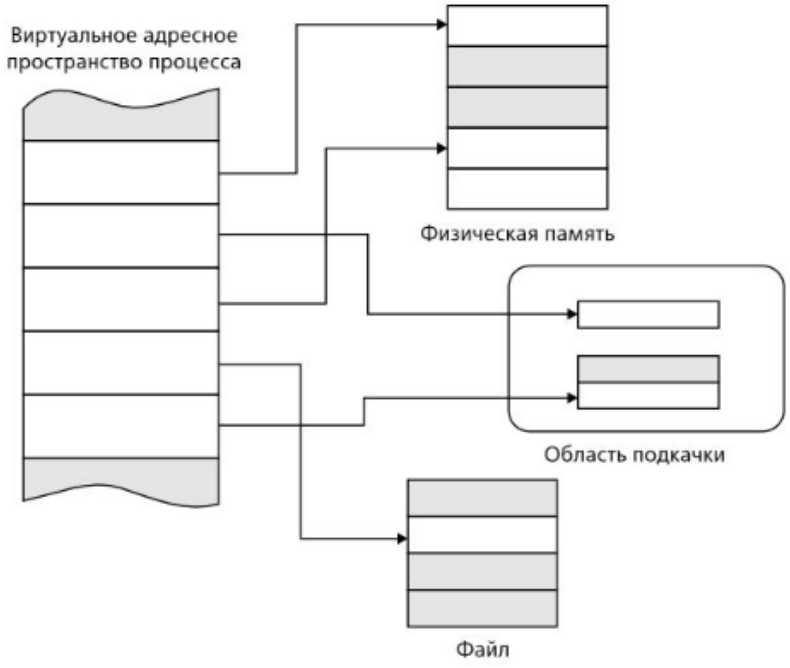
В UNIX процессы работают в пользовательском пространстве и имеют определённые атрибуты в ядре

Каждый процесс исполняется в собственном виртуальном адресном пространстве.
Это значит, что напрямую получить доступ к памяти другого процесса нельзя. Для взаимодействия нескольких процессов нужно использовать специальные механизмы - **IPC** (**Interprocess communication.**

Процесс:

Стек
Сегмент кода
Сегмент данных
Регистры

Виртуальная память процесса



Виртуальная память — метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём автоматического

перемещения частей программы между
основной памятью и вторичным хранилищем.

Процессы и потоки

Поток - набор команд в рамках процесса. У поток свой стек и IP, но тот же сегмент кода, данных и те же регистры

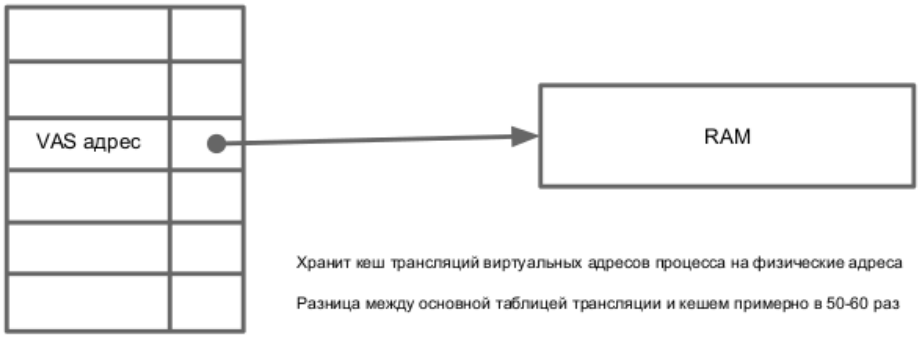
Процесс:

Стек
Поток: Стек IP
Сегмент кода
Сегмент данных
Регистры

Что выбрать?

- Процессы в целом надежнее, чем потоки, так как работают в своём адресном пространстве.
- Процессы позволяют организовать взаимодействие между программами, которые существуют в разное время.
- Потоки - быстрее, так как при переключении не нужно менять большинство параметров.

TLB (Translation lookaside buffer)



Это кэш, который хранит некоторые данные процесса. При использовании многопотоковых приложений нужные данные часто оказываются в кэше, что ускоряет работу потоков по сравнению с процессами 50-60 раз.

Запуск и завершение процесса

Старт процесса

В C++ - int main(int argc, char* argv)
argc - количество аргументов
argv - массив указателей на аргументы

Аргументы командной строки

```
int main(int argc, char *argv[]) {
     int i;
     for (i = 0; i < argc; i++)
          printf("argv[%d]: %s\n", i, argv[i]);
     exit(0);
}
```

По ISO C и POSIX элемент массива argv[argc] должен быть равен NULL:

```
for (i = 0; argv[i] != NULL; i++)
```

Завершение процесса

Нормальные способы:

- Возврат из функции main
- Вызов функции семейства exit
- Возврат из функции pthread_exit из последнего потока

Ненормальные способы:

- Вызов функции abort
- Получение сигнала

Функции семейства exit

- **_exit**, **_Exit** - моментальное возвращение управления ядру
- **exit** - перед передачей управления ядру происходит процедура завершения - очистка памяти и т.к.
- Можно установить свои обработчики exit с помощью **atexit**

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

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

Функция atexit

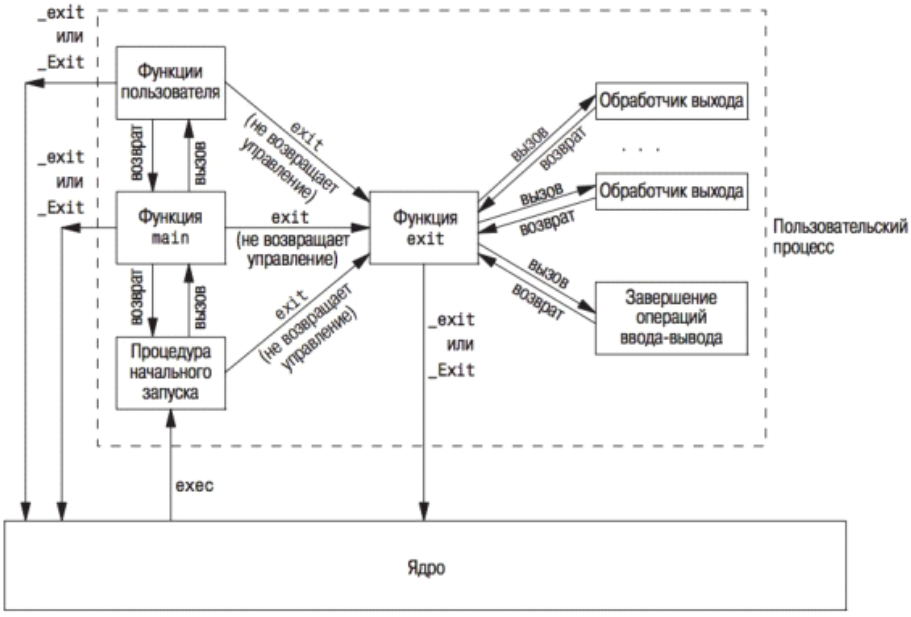
Позволяет задать функции - обработчики выхода

По стандарту ISO C можно задать до 32 обработчиков. Обработчики вызываются в порядке LIFO

```
#include <stdlib.h>
int atexit(void (*func)(void));
```

0 - в случае успеха, не 0 - в обратном случае

Процесс запуска и завершения



Управление процессами

Контекст процессора:

- **Пользовательский контекст**
  Содержимое виртуального адресного пространства, сегментов кода, данных, стека и сегментов файлов, отображаемых в виртуальную память.
- **Регистровый контекст**
  Содержимое аппаратных регистров
- **Контекст системного уровня**
  Структуры данных ядра, связанные с этим процессом
  - Статическая часть
  - Динамическая часть

Системный контекст процессора

Статическая часть

- **Идентификатор процесса (PID)**
  По умолчанию максимальный идентификатор в системе $2^{16} = 32768$, но это можно изменить
  ```
  #include <unistd.h>
  pid_t getpid(void); //Возвращает идентификатор вызывающего процесса
  ```
- **Идентификатор родительского процесса (PPID)**
  ```
  pid_t getppid(void); //Возвращает идентификатор родительского процесса
  ```
  

- **Реальный идентификатор пользователя процесса (UID)**
  ```
  uid_t getuid(void);
  ```
- **Реальный идентификатор группы процесса (GID)**
  ```
  gid_t getgid(void);
  ```
- **Приоритет процесса**
- **Таблица дескрипторов открытых файлов**

Системные вызовы возвращают pid_t, а не int, т.к. int-ы различаются в разных архитектурах. Это же относится к многим системы вызовам

PID и PPID

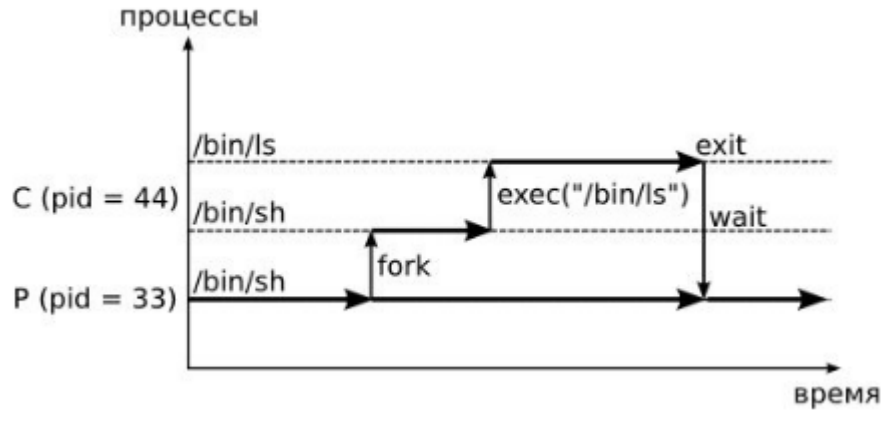
Состояние процесса

Пример

```
#include <unistd.h>
#include <stdio.h>
int main() {
    pid_t pid, ppid, uid, gid;
    pid = getpid();
    ppid = getppid();
    uid = getuid();
    gid = getgid();
    printf("pid = %d\nppid = %d\nuid = %d\ngid = %d\n", pid, ppid, uid, gid);
}
```

Создание новых процессов

- fork - создает новый процесс
- exec - запускает программу в адресном пространстве текущего процесса
- wait - ждет завершения процесса



Системный вызов Fork

Два основных случая, когда используется функция fork:

1. Когда процесс хочет продублировать себя, чтобы родительский и дочерний процессы могли выполнять различные участки программы одновременно. Используется, например, в сетевых серверах. Родительский процесс ожидает запроса на обслуживание от клиента, по его получении он вызывает fork и передает обслуживание запроса дочернему процессу, после чего возвращается к ожиданию следующего запроса.
2. Когда процесс хочет запустить другую программу. Это обычно используется в командных оболочках. В этом случае дочерний процесс вызывает функцию exec, как только функция fork вернет управление.

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

Возвращает 0 в дочернем процессе, идентификатор дочернего процесса - в родительском, -1 в случае ошибки

При вызове fork текущий процесс полностью копируется. Однако, реально происходит **копирование при записи** - т.е. адресное пространство копируется только при обращении по адресам. Если процессы работают с большими данными, лучше всё же разделить память специальными средствами.

Возможные ошибки (errno)

EAGAIN — ядро не способно выделить определенные ресурсы, например, новый pid

ENOMEM — недостаточно ресурсов памяти ядра

Использование

```
...
pid_t pid;
pid = fork();
if ( pid == 0 ) {
     // Дочерний процесс
}
else if ( pid > 0 ) {
     // Родительский процесс
}
else {
     // ошибка
}
     printf(“Hello from child and parent”);
```

Пример

```
#include <unistd.h>
#include <stdio.h>
int main() {
    pid_t pid;
    pid = fork();
    if ( pid == 0 ) {
        // Child
        printf("Child\n");
```

```
        sleep(10);
    }
    else {
        // Parent
        printf("Parent\n");
        sleep(10);
    }
    printf("Finished\n");
    return 0;
}
```

wait и waitpid

Оповещение родителя о завершении потомка осуществляется при помощи сигнала SIGCHLD .Часто нужно знать, как именно процесс завершился

Функция wait()

```
#include<sys/types.h>
#include <sys/wait.h>
pid_t wait (int *status);
```

Возвращает pid завершенного дочернего процесса или –1 в случае ошибки

Параметр - указатель на переменную, в которую будет положен результат. Wait ждет завершения первого попавшегося дочернего процесса - чтобы дождаться несколько процессов, нужно вызвать wait нестолько раз.

Макросы статуса

Статус - int с установленными битами, а не return процесса. Чтобы расшифровывать его, можно использовать макросы sys/wait.h

`int WIFEXITED (status);`	Возвращает true, если процесс завершается через вызов _exit(), обычным образом
`int WEXITSTATUS (status);`	Код завершения
`int WIFSIGNALED (status)`	Возвращает true, если прерывание процесса вызвал сигнал
`int WTERMSIG (status);`	Номер сигнала, который вызвал прерывание
`int WCOREDUMP (status);`	Возвращает true, если процесс сбросил ядро в ответ на получение сигнала
`int WIFSTOPPED (status)`	Возвращает true, если процесс был остановлен или продолжен соответственно
`int WIFCONTINUED (status);`	Возвращает true, если процесс был остановлен или продолжен соответственно
`int WSTOPSIG (status);`	Номер сигнала, остановившего процесс

Ошибки (errno)

- ECHILD — вызывающий процесс не имеет дочерних
- EINTR — сигнал был получен во время ожидания

Функция waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int *status, int options);
```

Параметр pid:

-1	Ожидание любого дочернего процесса; поведение аналогично wait()
>0	Ожидание любого дочернего процесса, чей pid в точности равен указанной величине; например, при величине 500 ожидается дочерний процесс с pid, равным 500

Параметр options:

- WNOHANG — не блокировать вызов, немедленно вернуть результат, если ни один подходящий процесс еще не завершился (остановился или продолжился);
- WUNTRACED — при его выборе устанавливается параметр WIFSTOPPED, даже если вызывающий процесс не отслеживает свой дочерний; это свойство помогает реализовать более общее управление заданиями, как это сделано в оболочке;
- WCONTINUED — если установлен, то бит WIFCONTINUED в возвращаемом параметре статуса будет установлен, даже если вызывающий процесс не отслеживает свой дочерний; как и в случае с WUNTRACED, параметр полезен для реализации оболочки.

Ошибки (errno):

- ECHILD — процесс или процессы, указанные с помощью аргумента pid, не существуют или не являются потомками вызывающего;
- EINTR — сигнал был получен во время ожидания;
- EINVAL — аргумент options указан некорректно.

Особые случаи

- **Случай 1**
  Родительский процесс завершает раньше дочернего
  В этом случае у всех дочерних процессов становится родитель init (pid = 1).
- **Случай 2**
  Дочерний процесс завершает до того, как родитель об этом узнает
  В этом случае дочерний процесс превращается в **процесс-зомби**.
- **Случай 3**
  Потомок завершился до того, как родитель об этом узнал, но

родителем стал init.
Процесс init работает особым образом и при завершении потомка он самостоятельно вызывает функцию wait. Таким образом предотвращается появление зомби.

Семейство вызовов exec

```
#include <unistd.h>
int execl (const char *path, const char *arg, ...);
```

Другие вызовы exec

- execlp
- execle
- execv
- execvp
- Execve

Буква	Тип вызова
i	Аргументы передаются списком
v	Аргументы передаются массивом
p	Поиск файла по пользовательским путям
e	Создается новое окружение

Это системный вызов, который замещает адресное пространство процесса каким-либо кодом. Например, командный процессор, чтобы выполнить программу, делает fork и exec.
После этого системного вызова остальной части программы больше не существует.
При старте exec'а не изменятся PID, PPID, приоритет, UID, GID.

Пример

```
#include <stdio.h>
#include <unistd.h>
int main() {
    char cmd[10];
    pid_t pid;
    while( 1 ) {
        printf("Enter the command: ");
        scanf("%s", cmd);
        pid = fork();
        if ( pid == 0 ) {
            printf("Processing command: %s\n", cmd);
            int r = execlp( cmd, cmd, NULL );
            printf("This text should be never printed\n");
            return 0;

        }
        else {
            wait(NULL);
        }
    }
```

```
}
```

Межпроцессорное взаимодействие

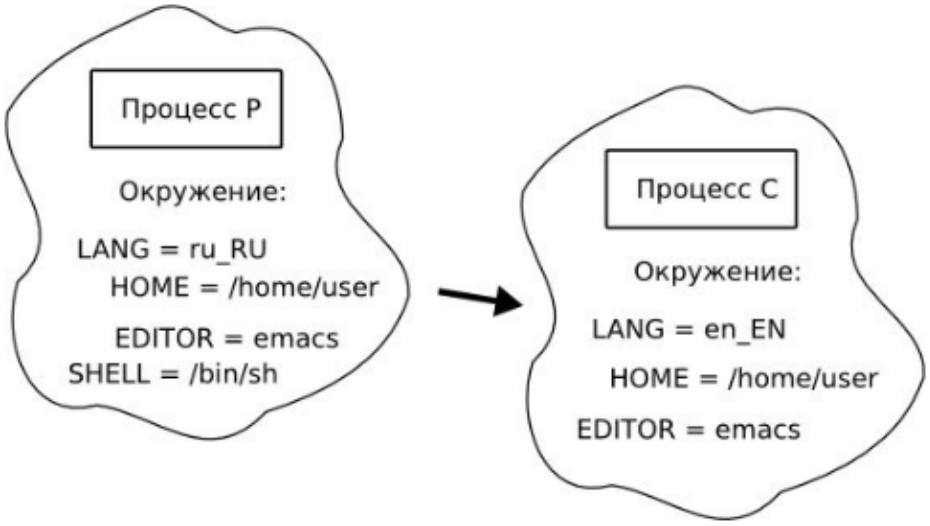
12 декабря 2018 г. 18:30

Межпроцессорное взаимодействие

Классические средства:

- Переменные окружения
- Сигналы
- Каналы
  - Именованные
  - Неименованные
- Сокеты

Переменные окружения

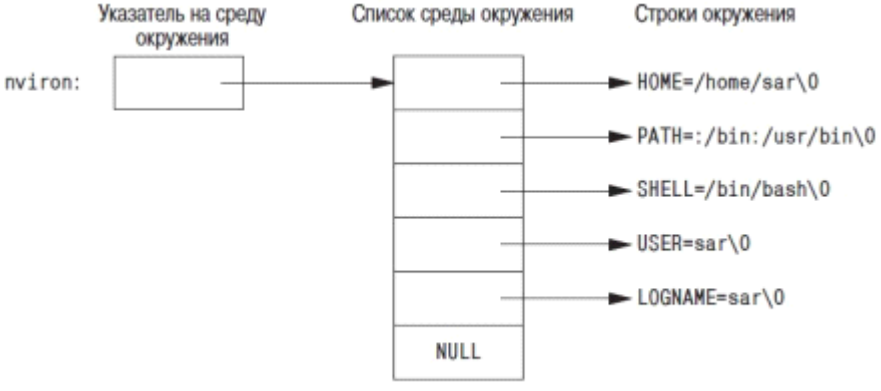


Переменная среды (англ. environment variable) — текстовая переменная операционной системы, хранящая какую-либо информацию — например, данные о настройках системы

Список переменных окружения

Чтобы вывести переменные окружения, нужно проитерироваться по

```
extern char** environ
```



Пример

```
// Вывод списка всех переменных окружения текущего процесса
#include <stdio.h>
```

```c
extern char **environ;
int main() {
     int i;
     for( i=0; environ[i] != NULL; i++ ) {
          printf("environ[%d]: %s\n", i, environ[i]);
     }
}
```

Системные вызовы

```c
#include <stdlib.h>
char *getenv(const char *name);
int putenv(char *str);
int setenv(const char *name, const char *value, int rewrite);
int unsetenv(const char *name);
```

Можно установить переменные на уровне интерпретатора или системы некоторые переменные.
Можно влиять на переменные окружения - добавлять, искать по имени и т.п. с помощью специальных процедур.
Этот способ передачи односторонний - нельзя изменить данные уже запущенного процесса. Объем данных также ограничен

Сигналы

Сигналы - программные прерывние, обеспечивающие асинхронную обработку событий

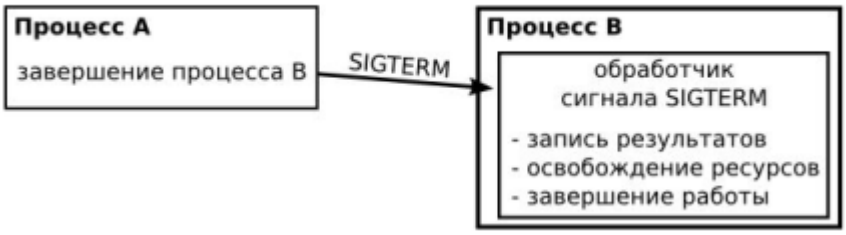
Возможные действия на событие:

- Игнорировать сигнал (кроме SIGKILL и SIGSTOP)
- Перехватить сигнал и обработать его
- Действие по умолчанию

Тип сигнала	Описание
SIGALRM (14)	Процесс может с помощью специального системного вызова abort задать время, через которое ему необходимо отправить сигнал. Через указанный промежуток времени операционная система доставит процессу сигнал SIGALARM. Обычно этот прием применяется для задания таймаутов. Если процесс не зарегистрировал обработчик этого сигнала, то обработчик по умолчанию завершает процесс.
SIGCHLD	Сигнал отправляется родительскому процессу в случае завершения его дочернего процесса. По умолчанию сигнал игнорируется.
SIGCONT	Сигнал продолжения исполнения программы после остановки. Обработчика по умолчанию нет.
SIGFPE (8)	Сигнал ошибки в вычислениях с плавающей точкой, отправляется операционной системой при некорректном

	исполнении программы. Обработчик по умолчанию завершает процесс.
SIGHUP (1)	Сигнал закрытия терминала, к которому привязан данный процесс. Обычно отправляется операционной системой всем процессам, запущенным из командной строки при завершении сеанса пользователя. Обработчик по умолчанию завершает процесс.
SIGKILL (9)	Сигнал аварийного завершения процесса. По этому сигналу процесс завершается немедленно — без освобождения ресурсов. Этот сигнал не может быть перехвачен, заблокирован или переопределён самим процессом, всегда используется стандартный обработчик операционной системы. Этот сигнал используется для гарантированного завершения процесса.
SIGPIPE (13)	Сигнал отправляется процессу, который пытается отправить данные в канал, закрытый с противоположной стороны. Такая ситуация может возникнуть в случае, если один из взаимодействующих процессов был аварийно завершён. Обработчик по умолчанию завершает процесс.
SIGSEGV (11)	Сигнал отправляется процессу операционной системой, если была произведена неверная операция с памятью (обращение по несуществующему или защищённому адресу). Обработчик по умолчанию завершает процесс.
SIGSTOP	Сигнал приостановки работы процесса. Этот сигнал не может быть перехвачен, заблокирован или переопределён. Используется для гарантированной приостановки работы процесса с полным сохранением его состояния и возможностью возобновления.
SIGTERM (15)	Сигнал завершения процесса, как правило используется для корректного завершения его работы.
SIGUSR1, SIGUSR2	«Пользовательские» сигналы — могут использоваться процессами для всевозможных уведомлений. Обработчик по умолчанию завершает процесс.

Список всех сигналов можно вывести с помощью команды kill - l



Системные вызовы

Обработчик сигнала можно переопределить с помощью системного вызова.

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

Функция signal

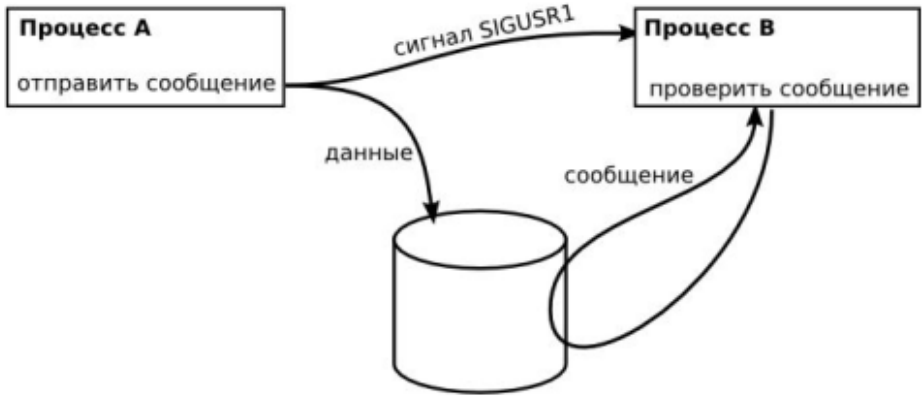
handler:

- функция void my_handler (int signo);
- SIG_DFL - действовать по умолчанию
- SIG_IGN - игнорировать сигнал

Пример

```
#include <signal.h>
void handler( int signum ) {
     printf("Catch signal!\n");
}
int main() {
     //signal( SIGINT, &handler );
     signal( SIGHUP, &handler );
     printf("Sleeping...\n");
     while( 1 )
          sleep(5);
}
```

Межпроцессорный обмен



Первый процесс, например, подготовив данные для второго процесса, отправляет сигнал-заглушку SIGUSR1 второму процессу.

Сигналы ведут себя по-разному в fork и exec. При fork'е обработчики

наследуются, при exec'e - не наследуются. Сигналы, ожидающие обработки, не наследуются в fork'e, но наследуются в exec'e

Отправка сигнала

Происходит с помощью kill из signal.h

```
#include <sys/types.h>
#include <signal.h>
int kill (pid_t pid, int signo);
```

Чтобы проверить, если и права на отправку сигналов, можно отправить сигнал 0 и посмотреть return code.

Ошибки (errno):

- EINVAL — сигнал, обозначенный signo, является недопустимым;
- EPERM — вызывающий процесс не обладает достаточными правами доступа, чтобы послать сигнал какому-либо запрошенному процессу;
- ESRCH — процесс не существует либо это процесс-зомби.

Определение прав доступа

```
int ret;
ret = kill (1722, 0);
if (ret != 0)
     ; /* право доступа отсутствует */
else
     ; /* право доступа имеется */
```

Неименованные каналы (pipe)

Канал - поток данных между двумя или несколькими процессами, имеющий интерфейс, аналогичный чтению или записи в файл

Ограничения

1. Исторически каналы симплексные - данные могут передаваться только в одном направлении.
2. Можно осуществлять только между процессами, имеющими общего родителя

Ограничения на размер данных нет, но есть ограничения на размер буфера.

Работа с каналами

- Один процесс пишет, второй читает как из файла
- Нет ограничения на размер передаваемых данных
- Есть ограничение на размер буфера

Обращение по дескриптору:

- open - не нужен, так как дескрипторы создают вызовом pipe
- write - блокирует выполнение, если нет места в буфере
- read - разрушает данные при чтении, не работает lseek

При записи без читателей процесс получает сигнал SIGPIPE

Создание

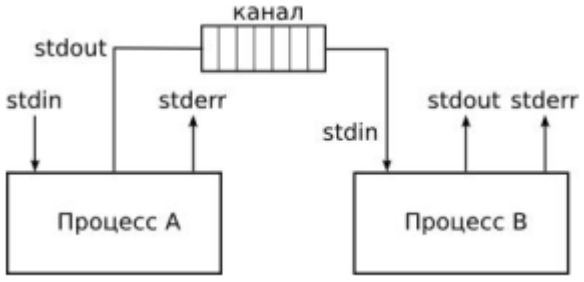
```
#include <unistd.h>
int pipe(int filedes[2]);
```

Сооздаются с помощью системного вызова pipe. Он принимает в качестве аргументов массив из двух элементов. Первый элемент будет дескриптором на чтение, второй - на запись.
Ненужные каналы стоит сразу же закрывать.

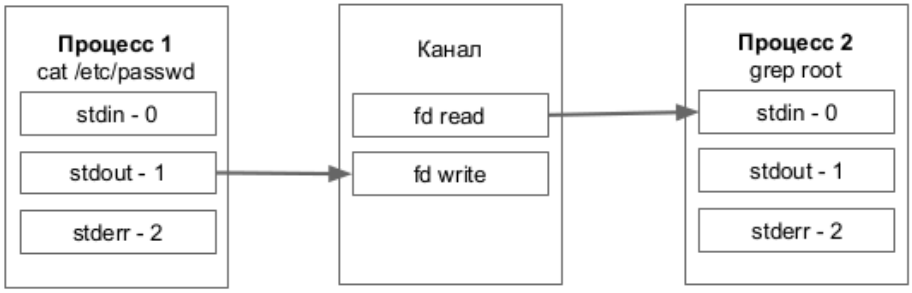
Пример

```
// Передача данных от родительского процесса дочернему через канал
#include <stdio.h>
#include <unistd.h>
int main(void) {
     int n;
     int fd[2];
     pid_t pid;
     char line[255];
     if (pipe(fd) < 0) {
          printf("ошибка вызова функции pipe\n");
          return 1;
     }
     if ((pid = fork()) < 0) {
          printf("ошибка вызова функции fork\n");
          return 1;
     } else if (pid > 0) { /* родительский процесс */
          close(fd[0]);
          write(fd[1], "Hello\n", 6);
     } else { /* дочерний процесс */
          close(fd[1]);
          n = read(fd[0], line, 255);
          printf("Message from parent: %s", line);
     }
     return 0;
}
```

Неименованные каналы используются в командной строке при реализации конвейера, например cat /etc/passwd | grep root



Пример



```c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>


int main(int argc, char **argv)
{
    int fd[2];
    pipe(fd); /*организован канал*/

    if (fork())
    {
        // Процесс 1
        dup2(fd[1], 1); /* отождествили стандартный вывод с файловым
дескриптором канала, предназначенным для записи */
        close(fd[1]); /* закрыли файловый дескриптор канала, предназначенный
для записи */
        close(fd[0]); /* закрыли файловый дескриптор канала, предназначенный
для чтения */
        execlp("cat", "cat", "/etc/passwd", NULL); /* запустили программу */
    }

    // Процесс 2
    dup2(fd[0], 0); /* отождествили стандартный ввод с файловым дескриптором
канала, предназначенным для чтения*/
    close(fd[0]); /* закрыли файловый дескриптор канала, предназначенный для
чтения */
    close(fd[1]); /* закрыли файловый дескриптор канала, предназначенный для
записи */
    execl("/bin/grep", "grep", "root", NULL); /* запустили программу wc */
}
```

Именованные каналы

FIFO - особый тип файлов. Из-за именования эти каналы можно использовать не только между родственными процессми.

Создание FIFO

```c
#include <sys/stat.h>
int mkfifo(const char *pathname, mode_t mode);
```

Пример: клиент-сервер


Сервер
запись ответов
чтение запросов
запись ответов
Канал FIFO для связи
с конкретным клиентом
Канал FIFO
с известным именем
Канал FIFO для связи
с конкретным клиентом
чтение ответов
запись запросов
запись запросов
чтение ответов
Клиент
. . .
Клиент

XSI IPC

Thursday, October 18, 2018 11:49

XSI IPC

Рассмотренные ранее средства межпроцессного взаимодействия не позволяют оорганизовать обмен между процессами, выполняющимися в различное время.

Идентификаторы и ключи

- Каждому XSI IPC соответствует идентификатор в системе.
- Идентификатор - внутреннее имя IPC
- В качестве внешнего идентификатора выступает специальный ключ, который используется в системных вызовах

Ключ

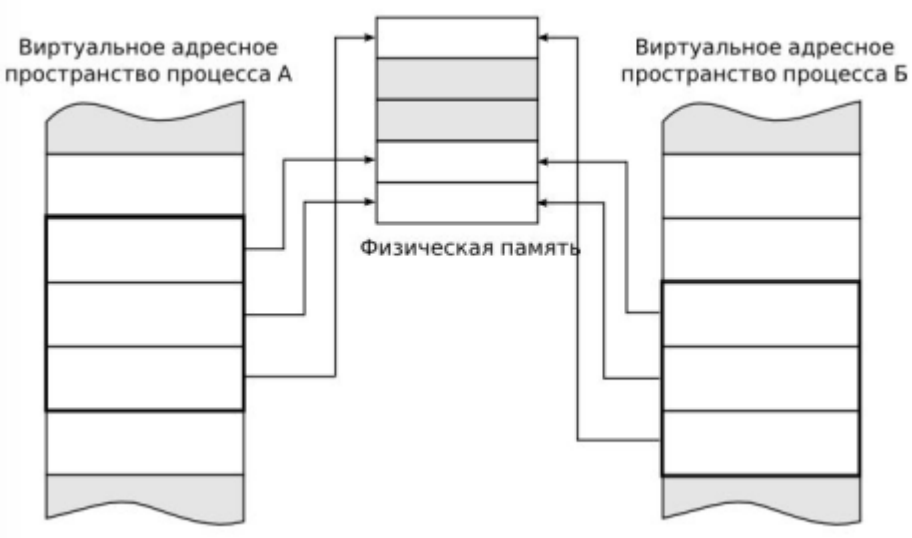
key_t - длинное целое со знаком. Ядро выполняет преобразование ключа в идентификатор

Задание ключа

```
#include <sys/ipc.h>
key_t ftok(const char *path, int id);
```

Ключ можно создавать и вручную, но лучше использовать готовую функцию, чтобы избежать коллизий.
Файл необходим для только генерации ключа

Разделяемая память



Разделяемая память позволяет преодолеть ограничения виртуального пространства процессов.

Поведение при fork и exec

При fork разделяемая память будет унаследована, exec - нет.

Создание разделяемой памяти

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, size_t size, int shmflg);
```

I аргумент: key_t key

Это результат вызова функции ftok().
Специальное значение IPC_PRIVATE, которое приводит к попытке создания новой разделяемой памяти с ключом, значение которого не совпадает ни с одним ключом уже существующих объектов IPC, и которое не может быть получено с помощью функции ftok() ни при каких комбинациях её параметров

II аргумент: size_t size

Аргумент size задает размер создаваемого сегмента памяти.
Если сегмент памяти с заданным ключом уже существует и заданное значение size не совпадает с размером существующего сегмента, то констатируется возникновение ошибки.

III аргумент: int shmflg

Имеет значение только при создании новой разделяемой памяти и определяет права доступа к данной очереди различных пользователей, а также необходимость создания новой очереди и правила поведения функции при этом.
IPC_CREAT — если РП для указанного ключа не существует, она должна быть создана;
IPC_EXCL — применяется совместно с флагом IPC_CREAT. При совместном их использовании и существовании РП с указанным ключом доступ к РП не производится и констатируется ошибочная ситуация

Подключение РП к процессу

```
#include <sys/types.h>
#include <sys/shm.h>
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Возвращает адрес сегмента памяти в адресном пространстве процесса в случае успеха, -1 в случае ошибки

II аргумент: *smaddr

Аргумент *shmaddr является указателем на область памяти к которой необходимо подключить новый сегмент. Вместо конкретного указателя на память может быть введено значение NULL, в этом случае память будет присоединена к первому свободному участку памяти в адресном пространстве процесса.

III аргумент: *shmflg

Аргумент shmflg может принимать множество различных значений, но в рамках интересующих нас в ключе обмена информации между процессами — нас будет интересовать лишь два значения

0 — осуществление чтения и записи в сегмент памяти
SHM_RDONLY — только чтение из сегмента памяти

Отсоединение РП от процеса

```
#include <sys/types.h>
#include <sys/shm.h>
int shmdt(char *shmaddr);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Удаление РП

Удалить РП происходит только вручную. Перед завершением программы нужно сделать системный вызов

```
#include <sys/ipc.h>
#include <sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

Память можно удалить с помощью системной утилиты ipc:

- ipcs -a просмотр всех средств IPC
- ipcrm - удаление IPC из системы

Пример. Использование разделяемой памяти между процессами, существующими в разное время в системе

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/wait.h>
#include <stdlib.h>
int main() {
     int *SHARED;
     int sm;
     sm = shmget( IPC_PRIVATE, sizeof(int), 0666 | IPC_CREAT | IPC_EXCL );
     SHARED = (int*)shmat(sm, NULL, 0);
     *SHARED = 777;
     int pid = fork();
     if ( pid == 0 ) {
          printf("Child initial SHARED = %d\n", *SHARED);
          sleep(5);
          printf("Child next SHARED = %d\n", *SHARED);
     }
     else {
          printf("Parent initial SHARED = %d\n", *SHARED);
          sleep(2);
          *SHARED = 999;
          printf("Parent modify SHARED = %d\n", *SHARED);
```

```
            wait(NULL);
        }
        shmctl(sm, IPC_RMID, NULL);
}
```

Проблемы при работе с РП

- Гонки данных
- Проблема читателей и писателей (ОС)
  Пусть есть 64-битная машина и мы хотим скопировать 128-битное число.
  C: n1 = n2
  asm:
  mov <1-е 64-бита>
  mov <2-е 64 бита>
  Параллельный читатель может прочитать между mov'ами и получить мусор.

Семафоры

Семафоры определяются некоторым целочисленным неотрицательным типом и над ними возможны две операции down и up

- down - сравнивает значение переменной с нулём, и если значение переменной больше нуля, то происходит уменьшение на 1 и возвращение управления программе. Если же нет - программа блокируется и ждёт, пока семафор не станет больше 0.
- up - повышает значение переменной

Решение проблемы читателей и писателей

1. Создается семафор с начальным значением равным единице.
2. Перед тем как производить запись в разделяемую память процесс выполняет операцию down над семафором, и тогда если семафор равен 1, то процесс сможет спокойно произвести запись, а если значение равно 0, то это будет означать, что некоторый процесс уже работает с памятью и процесс будет переведен в состояние ожидания.
3. После окончания операций работы с памятью процесс выполняет операцию up над семафором

Мьютекс

Мьютекс - это бинарный семафор, который знает, какой поток его заблокировал. Разблокировать мьютекс обычно может тот же процесс.

Пример реализации на Java

```java
class LockOne implements Lock {
    private boolean[] flag = new boolean[2];

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true;
        while (flag[j]) {} // wait
    },

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

Вариант 2

```java
class LockTwo implements Lock {
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        victim = i;

        while(victim == i) {} // wait
    }

    public void unlock() {}
}
```

Рабочий вариант

```java
class Peterson implements Lock {
    private volatile boolean[] flag = new boolean[2];
    private volatile int victim;

    public void lock() {
        int i = ThreadID.get();
        int j = 1 - i;
        flag[i] = true; // Поток заинтересован
        victim = i;
        while (flag[j] && victim == i) {}; // wait
    }

    public void unlock() {
        int i = ThreadID.get();
        flag[i] = false;
    }
}
```

Примитивы синхронизации

Нерекурсивные - при повторном захвате тем же потоком вызывают deadlock

Рекурсивные - позволяют захватить себя тем же потоком несколько раз. Использовать рекурсивные примитивы может быть проще, но при этом они понижают читаемость кода.

Часто в классах реализуются безопасные и небезопасные методы, причем последние не блокируют объект. Это может быть полезно, если mutex

нерекурсивный и нужно вызвать одну процедуру из другой.

Пример 1

```
Class Vector {
    Mutex m;

    public void add() {
        m.lock();
        size();
        extend();
        m.unlock();
    }

    public int size() {
        m.lock();
        int size = getSize();
        m.unlock()

        return size;
    }
}
```

Пример 2

```
Class Vector {
    Mutex m;

    public void add() {
        m.lock();
        unsafeAdd();
        m.unlock();
    }

    public int size() {
        m.lock();
        size = unsafeSize()
        m.unlock()

        return size;
    }

    public void unsafeAdd() {
        unsafeSize();
        extend();
    }

    public int unsafeSize() {
        return getSize();
    }
}
```

Виды мьютексов

- **Timed mutex** - пытается захватить мьютекс в течении заданного времени. В таком случае, если произошёл deadlock, он через некоторое время разрешится.
- **Shared mutex** - позволяет блокировать только на чтение, на запись или смешанно. Позволяет сворачивать lock-и на отдельные операции в один lock
  `w r r r w r r w r r r = w r w r w r`
- **Spin mutex** - мьютекс с активным ожиданием без ухода в ядро. Это может быть полезно, если в системе нет большой конкуренции.
- **Futex** - работает по возможности без обращения к ядру.
  Захват на чтение - не уходит в ядро
  Захват на запись - не уходит в ядро, пока нет второго запроса на

запись

CAS-операции

CAS - compare and set, compare and swap.

```
bool compare_and_set( int *source, int oldValue, int newValue)
```

- Они атомарны на уровне процессора i486+
- Возвращают признак успешности установки значения

Семафоры XSI IPC

1. up(S,n) - увеличение значения семафора на величину n
2. down(S,n) - блокировка процесса, если значение S<n, после чего S=s-n
3. z(S) - процесс блокируется до тех пор, пока значение семаформа S не станет равным 0.

Создание семафоров

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t key, int nsems, int semflg);
```

Аргументы:

- nsems - определяет количество семафоров в создаваемом массиве, или количество семафоров в уже созданном, в случае обращения по известному ключу.
- semflg аналогичен shmflg

При создании семафор всегда равен 0.

Выполнение операций

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, size_t nsops);
```

Возвращает 0 в случае успеха, -1 в случае ошибки

Структура аргумента *sops:

```
struct sembuf{
     short sem_num; //Номер семафора в массиве
     short sem_op; //Выполняемая операция
     short sem_flg; //Флаги
}
```

Значения sem_op:

- Для выполнения операции up(S,n) значение должно быть равно n
- Для выполнения операции down(S,n) значение должно быть равно -n
- Для выполнения операции z(S) значение должно быть равно 0

Пример

```
semd = semget(IPC_PRIVATE, 1, 0666 | IPC_CREAT);
struct sembuf mybuf;
// задаем начальное значение семафора равное 1
```

```
mybuf.sem_num = 0;
mybuf.sem_op = 1;
mybuf.sem_flg = 0;
semop(semd,&mybuf,1);
```

Удаление семафора

Как и РП, семафоры нужно удалять вручную с помощью semctl.

```
semctl(semid, 0, IPC_RMID, NULL);
```

Очереди сообщений

Можно представить как почтовый ящик, в который различные процессы системы отправляют сообщения, помечая их некоторым типом.

Выборка сообщений производится процессами по собственному желанию следующими способами:

1. Независимо от типа сообщения в порядке FIFO (First In First Out).
2. В рамках определенного типа в порядке FIFO.
3. Первым выбирается самое старое сообщение имеющее минимальный тип.

Данный вид IPC считается устаревшим и не рекомендуется к применению в реальных системах, т.к. они значительно уступают в быстродействии.

Недостатки XSI IPC

- XSI IPC привязаны к ядру, а не к процессу
- Не имеют счетчика ссылок
- Не имеют имени в файловой системе (возникает необходимость использования специальных системных вызовов)

Матричные операции

Thursday, November 1, 2018 09:54

Матричные операции

Матрицы и матричные операции широко используются при математическом моделировании самых разнообразных процессов, явлений и систем. Матричные вычисления составляют основу многих научных и инженерных расчетов – среди областей приложений могут быть указаны вычислительная математика, физика, экономика и др.

Являясь вычислительно-трудоемкими, матричные вычисления представляют собой классическую область применения параллельных вычислений.

Умножение матрицы на вектор

Постановка задачи

В результате умножения матрицы A размерности $m \times n$ и вектора $\vec{b}$, состоящего из n элементов, получается вектор с размера m, каждый i-й элемент которого есть результат скалярного умножения i-й строки матрицы A (обозначим эту строчку a_i) и вектора $\vec{b}$.

$$c_i = (a_i, b) = \sum_{j=1}^{n} a_{ij} b_j \,; 1 \le i \le m$$

$$A \cdot \vec{b} = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix} \cdot \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix} = \begin{pmatrix} a_{11} \cdot b_1 + \cdots + a_{1n} \cdot b_n \\ \vdots \\ a_{m1} \cdot b_1 + a_{mn} \cdot m_n \end{pmatrix} = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix}$$

Последовательный алгоритм

```
for (i=0; i < m; i++){
     c[i] = 0;
     for (j=0; j < n; j++){
          c[i] += A[i][j]*b[j]
     }
}
```

- A[m][n] – матрицы размерности m×n.
- b[n] – вектор, состоящий из n элементов.
- c[m] – вектор из m элементов.

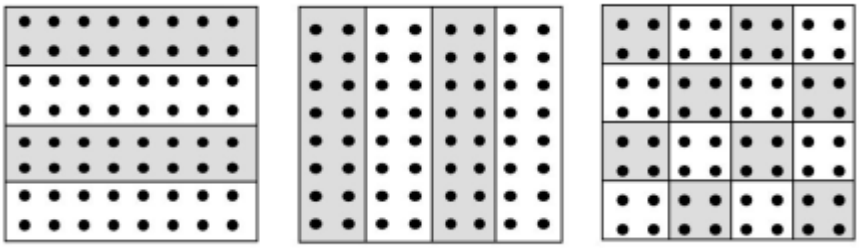
Принципы распараллеливания

Для многих методов матричных вычислений характерным является повторение одних и тех же вычислительных действий для разных элементов матриц. Данный момент свидетельствует о наличии параллелизма по данным при выполнении матричных расчетов и, как результат, распараллеливание матричных операций сводится в большинстве случаев к разделению обрабатываемых матриц между

процессорами используемой вычислительной системы.

Существуют два основных способа разбиения матриц по процессорам:

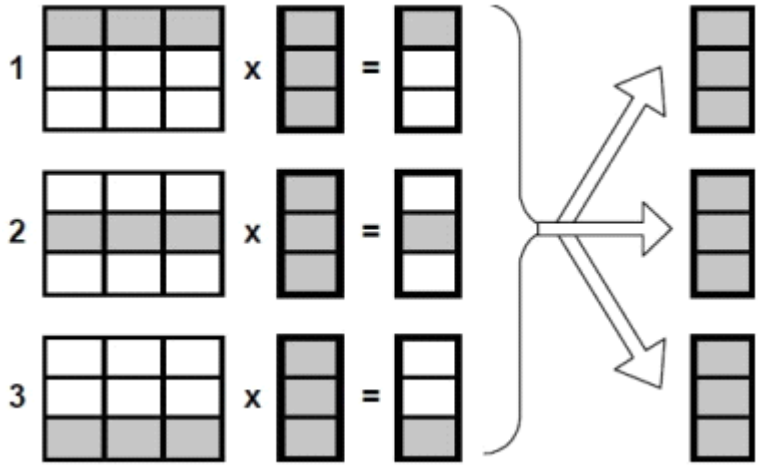
- Ленточное разбиение матрицы
- Блочное разбиение матрицы



Разделение данных

При выполнении параллельных алгоритмов умножения матрицы на вектор, кроме матрицы A необходимо разделить еще вектор b и вектор результата c. Элементы векторов можно продублировать, то есть скопировать все элементы вектора на все процессоры, составляющие многопроцессорную вычислительную систему, или разделить между процессорами.

<u>Умножение матрицы на вектор при разделении данных по строкам</u>



Для выполнения базовой подзадачи скалярного произведения процессор должен содержать соответствующую строку матрицы A и копию вектора $\vec{b}$. После завершения вычислений каждая базовая подзадача определяет один из элементов вектора c.

Анализ эффективности

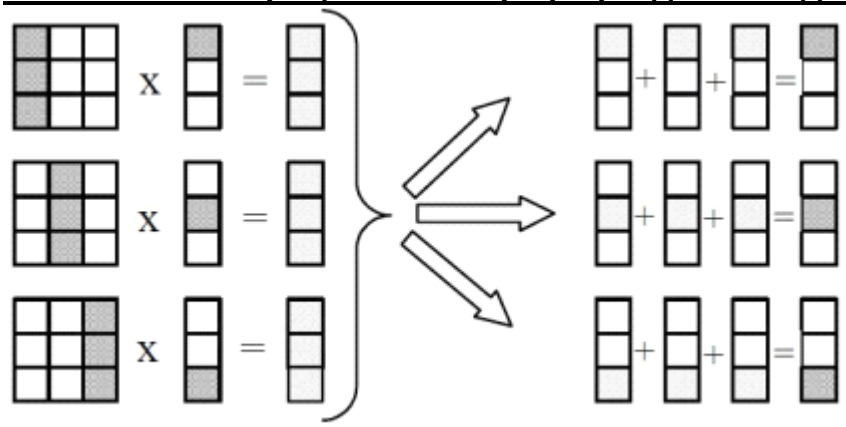
$$S_p = \frac{n^2}{n^2/p} = p$$

$$E_p = \frac{n^2}{p^2(n^2/p)} = 1$$

Построенные оценки времени выражены в количестве операции и определены без учета времени передачи данных. Каждый процессор

производит умножение только части (полосы) матрицы A на вектор b, размер этих полос равен n/p строк. При вычислении скалярного произведения одной строки матрицы и вектора необходимо произвести n операций умножения и (n-1) операций сложения

Умножение матрицы на вектор при разделении данных по столбцам



При таком способе разделения данных в качестве базовой подзадачи может быть выбрана операция умножения столбца матрицы A на один из элементов вектора b. Для организации вычислений в этом случае каждая базовая подзадача $i, 0 \le i < n$, должна содержать $i-$ й столбец матрицы A и $i-$ е элементы векторов $\vec{b}$ и $\vec{c}$

Анализ эффективности

$$S_p = \frac{n^2}{n^2/p} = p;\ E_p = \frac{n^2}{p^2(n^2/p)} = 1$$

Как и ранее, построенные оценки времени вычислений выражены в количестве операций и определены без учета затрат на выполнение операций передачи данных

Дублирование вектора по процессорам

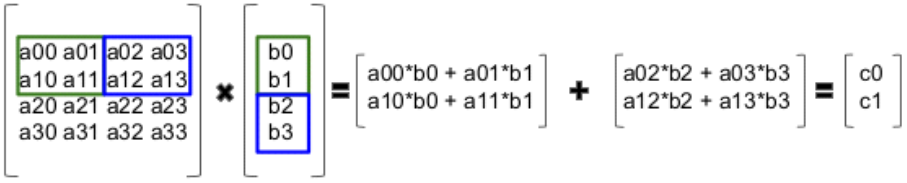
- Если процессор хранит одну строку матрицы и весь вектор, то общее число элементов имеет порядок $O(n+n)$
- Если процессор хранит одну строку матрицы и один элемент вектора, то общее число элементов имеет порядок $O(n+1)$
- $O(n+n) \sim O(n+1) \sim O(n)$ - Класс сложности по памяти не изменяется

Блочное разделение данных

При блочном разделении матрица делится на прямоугольные наборы элементов – при этом, как правило, используется разделение на непрерывной основе. Пусть количество процессоров составляет $p = s \cdot q$, количество строк матрицы является кратным s, а количество столбцов – кратным q, то есть $m = k \cdot s$ и $n = l \cdot q$. Представим исходную матрицу A в виде набора прямоугольных
блоков следующим образом:

$$A = \begin{pmatrix} A_{00} & A_{02} & \dots & A_{0(q-1)} \\ \dots & \dots & \dots & \dots \\ A_{(s-1)1} & A_{(s-1)2} & \dots & A_{(s-1)(q-1)} \end{pmatrix}; A_{ij} = \begin{pmatrix} a_{i_0 j_0} & \cdots & a_{i_0 j_{i-1}} \\ \vdots & \ddots & \vdots \\ a_{i_{k-1} j_0} & \cdots & a_{i_{k-1} j_{l-1}} \end{pmatrix}$$

$$i_v = ik + v; 0 \le v < k; k = \frac{m}{s}; j_n = jl + u; 0 \le u \le l; l = \frac{n}{q}$$



Анализ эффективности

$$S_p = \frac{n^2}{n^2/p} = p; E_p = \frac{n^2}{p \cdot (n^2/p)} = 1$$

Этот способ разделения данных в первую очередь направлен на архитектуруру вычислителя - если связь процессоров идет по принципу решетки, то данный алгоритм будет быстрее.

Потоки

Thursday, November 1, 2018 10:09

Потоки

Поточность - создание и управление множеством исполняемых элементов внутри одного процесса. Процесс содержит один или несколько потоков. Потоки существуют в одном адресном пространстве процесса.

Преимущества и недостатки потоков

- ✓ Параллелизм
- ✓ Переключение контекста
- ✓ Экономия памяти
- – Необходима синхронизация
- – Меньшая стабильность - падения одного потока может привести к падению процесса

Виды реализаций

- Пользовательские потоки (Linux 2.4)
- Потоки ядра (Linux >= 2.6, NTPL)
- Гибридная реализация

Пользовательские потоки

- ✓ Переключение потоков не требует участия ядра - нет переключения из режима задачи в режим ядра
- ✓ Планирование может определятся приложением - при этом выбирается наилучший алгоритм
- ✓ Пользовательские потоки могут применятся в любой ОС - необходимо лишь наличие совместимой библиотеки потоков
- – Большинство системных вызовов является блокирующими и ядро блокирует процессы - включая все потоки в пределах процесса;
- – Ядро может направлять на процессоры только процессы - два потока в пределах одного и того же процесса не могут выполняться одновременно на двух разных процессорах.

Потоки ядра

- ✓ Ядро может одновременно планировать выполнение нескольких потоков одного процесса на нескольких процессорах, блокирование выполняется на уровне потока;
- ✓ Процедуры ядра могут быть многопоточными
- – Переключение потоков в пределах одного процесса требует участия ядра.

Гибридная реализация

Создание пользовательских потоков внутри потоков ядра

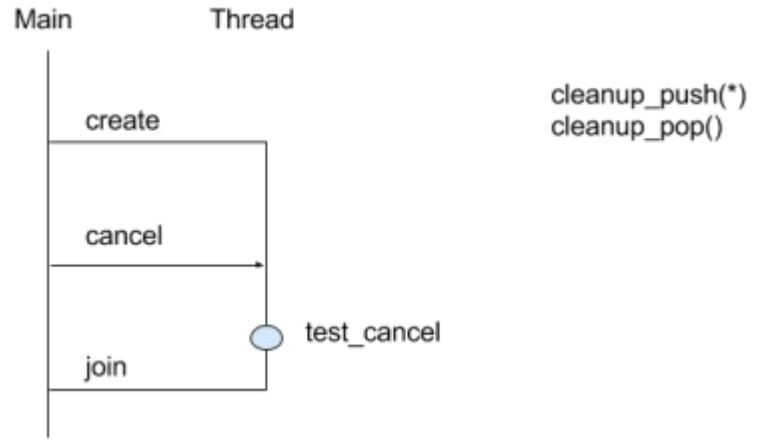
POSIX Threads

Существует множество реализаций потоков: Android, Apache, Mozilla, GNOME. Единый стандарт потоков - POSIX Threads.
API обозначено в файле <pthread.h>. Он содержит около 100 системных вызовов:

- управление потоками — функции для создания, уничтожения, соединения и отсоединения потоков
- синхронизация — функции для управления синхронизацией потоков

Библиотека потоков - libpthreads. Чтобы её слинковать, нужно добавить в gcc ключ -pthread

Жизненный цикл потока



Потоку можно отправить просьбу завершится, но есть возможность явно запретить завершение потока.
Один из способов завершения потока - Cancellation Points - точки остановки потока. В этих точках проверяется, нужно ли остановить поток, и если стоит нужный флаг, то производится остановка
Другой способ - включить специальный тип остановки потока, когда ОС сама решит, когда его можно остановить.

Создание потоков

```
#include <pthread.h>
int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void
*(*start_routine)(void*), void *arg);
```

Аргументы

1. Указатель на переменную, куда будет положен идентификатор потока
2. Структура с атрибутами
3. Функция, которая станет телом потока
4. Аргументы функции

Возвращаемый результат

0 - в случае ошибки, иначе номер ошибки (без использования errno)

EAGAIN	Вызывающему процессу существенно недостает ресурсов для создания нового потока; обычно это вызвано тем, что процесс достиг предела количества потоков для каждого пользователя
EINVAL	Объект pthread_attr_t, указанный через attr, включает недопустимые атрибуты;
EPERM	Вызывающий процесс не имеет полномочий для установки некоторых атрибутов объекта pthread_attr_t, указанного через attr

Пример

```
pthread_t thread;
int ret;
ret = pthread_create (&thread, NULL, start_routine, NULL);
if (!ret) {
     errno = ret;
     perror("pthread_create");
     return -1;
}
```

Идентификатор потоков

В отличие от PID-а назначается библиотекой. Чтобы получит id, есть системный вызов pthread_self. Пользоваться gettid() не имеет смысла

```
#include <pthread.h>
pthread_t pthread_self (void);
```

Сравнение идентификаторов

Для сравнения идентификаторов нужна отдельная функция, т.к. стандарт не накладывает ограничений на содержимой идентификатор

```
#include <pthread.h>
int pthread_equal (pthread_t t1, pthread_t t2);
```

Завершение потока

- Если поток возвращается из стартовой процедуры, он прерывается; это аналог "выхода за пределы" в main()
- Если поток вызывает функцию pthread_exit(), он завершается; это аналог функции exit()
- Если поток отменяется другим потоком через функцию pthread_cancel(), он завершается; это аналог отправки сигнала SIGKILL через kill().

Самозавершение

```
#include <pthread.h>
void pthread_exit (void *retval);
```

pthread_exit может принимать произвольные данные. Это можно использовать для передачи данных между потоками

Завершение других потоков

```
#include <pthread.h>
int pthread_cancel (pthread_t thread);
```

Функция pthread_cancel устанавливает бит в данных потока, который поток может проверить и завершиться, когда ему удобно.

Присоединение потоков

```
#include <pthread.h>
int pthread_join (pthread_t thread, void **retval);
```

Это аналог системного вызова wait() для процессов - поток ждет, пока завершится другой поток

Возможные ошибки

- EDEADLK - взаимная блокировка - thread уже ожидает присоединения к вызывающему потоку или сам является вызывающим потоком;
- EINVAL - невозможно присоединить поток
- ESRCH - значение thread недопустимо

Пример 1

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
void * start_thread (void *message)
{
     printf("PID: %d, %s\n", getpid(), (const char *) message);
     //printf("PThreadID: %d\n", pthread_self());
     return message;
}
int main (void)
{
     pthread_t thing1, thing2;
     const char *message1 = "Thing 1";
     const char *message2 = "Thing 2";
     /* Создаются два потока, каждый со своим сообщением */
     pthread_create (&thing1, NULL, start_thread, (void *) message1);
     pthread_create (&thing2, NULL, start_thread, (void *) message2);
     pthread_join (thing1, NULL);
     pthread_join (thing2, NULL);
     return 0;
}
```

Пример 2

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
void * thread_func(void *arg)
{
    int i;
    int loc_id = *(int *) arg;
    for (i = 0; i < 4; i++) {
        printf("Thread %i is running\n", loc_id);
        sleep(1);
    }
```

```
}
int main(int argc, char * argv[])
{
    int id1, id2, result;
    pthread_t thread1, thread2;
    id1 = 1;

    result = pthread_create(&thread1, NULL, thread_func, &id1);
    id2 = 2;
    result = pthread_create(&thread2, NULL, thread_func, &id2);
    result = pthread_join(thread1, NULL);
    result = pthread_join(thread2, NULL);
    printf("Done\n");
    return EXIT_SUCCESS;
}
```

Условия отмены

Состояние отмены потока (по умолчанию включено)

```
#include <pthread.h>
int pthread_setcancelstate (int state, int *oldstate);
```

state: PTHREAD_CANCEL_ENABLE или PTHREAD_CANCEL_DISABLE

Тип отмены потоков

- Асинхронный - ОС ищет ближайший интервал времени, когда можно завершить поток
- Отложенный - только в определённые моменты

Смена типа отмены

```
#include <pthread.h>
int pthread_setcanceltype (int type, int *oldtype);
```

type: PTHREAD_CANCEL_ASYNCHRONOUS или PTHREAD_CANCEL_DEFERRED

Точка проверки отложенной отмены потока

```
#include <pthread.h>
void pthread_testcancel(void);
```

Проверяет, установлен и флаг остановки, и если остановлен - вызываются обработчики завершения потока

Освобождение ресурсов перед завершением

```
pthread_cleanup_push (void *func, void *arg)
```

Устанавливает функцию, которая будет запущена при:

- Отмене потока
- Выходе через pthread_exit
- pthread_cleanup_pop

Пример. Отложенная остановка потока

```
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <errno.h>
#include <sys/types.h>
```

```c
#include <sys/stat.h>
#include <fcntl.h>
long s = 0;
void * start_thread () {
     //pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
     //pthread_setcanceltype( PTHREAD_CANCEL_ASYNCHRONOUS, NULL );
     pthread_setcanceltype( PTHREAD_CANCEL_DEFERRED, NULL );
     while( 1 ) {
          s += 1;
          sleep(1);
          pthread_testcancel();
     }
}
int main (void)
{
     pthread_t thing1;
     pthread_create (&thing1, NULL, start_thread, NULL);
     int d;
     printf("Press any key and enter: ");
     scanf("%d", &d);
     int r = pthread_cancel( thing1 );
     printf("pthread_cancel result: %d\n", r);
     r = pthread_join (thing1, NULL);
     printf("pthread_join result: %d\n", r);
     printf("Var result: %lu\n", s);
     }
```

Многие системные вызовы сами делают проверки на Cancellation Point - например, sleep()

Пример потоков в Java

```java
Thread t;

t.start();
t.interrupt() // isInterrupted
t.join();
```

Особый метод t.interrupted()

```java
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

Мьютексы

Инициализация мьютексов

```c
#include <pthread.h>
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

Запирание мьютексов

```c
#include <pthread.h>
int pthread_mutex_lock (pthread_mutex_t *mutex);
```

Успешный вызов заблокирует вызывающий поток, пока мьютекс, указанный как mutex, не станет доступным

Ошибки:

- EDEALDK - вызывающий поток уже владеет запрашиваемым мьютексом
- EINVAL - значение mutex недопустимо

Отпирание мьютексов

```
#include <pthread.h>
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Успешный вызов высвобождает мьютекс, указанный как mutex, и возвращает 0

Ошибки:

- EINVAL - значение mutex недопустимо
- EPERM - вызывающий процесс не владеет мьютексом, указанным как mutex; попытка освободить мьютекс, которым вы не владеете, является ошибкой.

Моделирование параллельных процессов

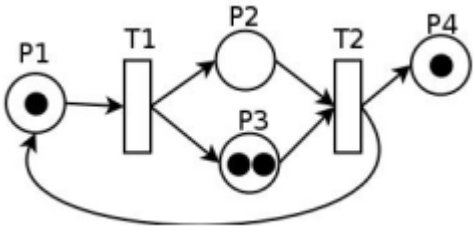
Thursday, November 1, 2018 10:56

Моделирование паралелльных процессов

Отладка параллельных программ - сложное дело, зачастую выявить потенциальные "дедлоки"
В простых задачах можно воспользоваться классическими средствами отладки - дебаггером, профайлером, printf-ом и т. д.
При отладке или проектировании сложных параллельных программ необходимо прибегать к некоторому формализму, позволяющему описать параллельный алгоритм в формальном виде и вычислить его свойства

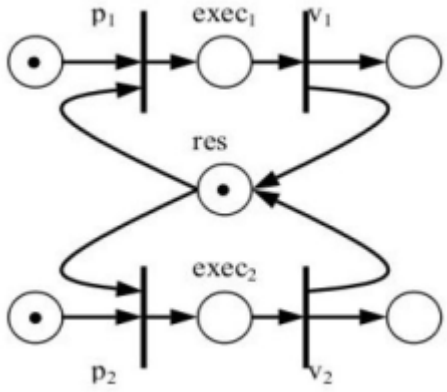
Сети Петри

Одним из распространенных средств описания параллельных алгоиртмов являются **сети Петри**



Сеть Петри представляет собой граф особого вида, состоящий из вершин двух типов - позиций и переходов, соединенных ориентированными дугами, причем каждая дуга может связывать лишь разнотипные версины (позицию с переходом или переход с позицией)

Пример. Описание критической секции с помощью сетей Петри



Переход считается **активным** (событие может произойти), если в каждой его входной позиции есть хотя бы одна фишка. Расположение фишек в позиция сети называется **разметкой сети**.

Свойства сетей Петри

Ограниченность . Это свойство связано с введением ограничений на число

меток в позициях. Позиция p_i называется k-ограниченной, если количество фишек в ней не может превышать целого числа k . Сеть Петри называется ограниченной, если все ее позиции ограничены. Ограниченную сеть Петри можно реализовать аппаратно, а неограниченную нельзя.

Безопасность. Позиция сети Петри называется **безопасной**, если число фишек в ней никогда не превышает единицы. Сеть Петри безопасна, если безопасны все ее позиции. Это свойство важно при интерпретации позиций как простых условий: если в позиции есть фишка, то условие выполняется, если нет, то не выполняется. Безопасную позицию можно реализовать одним триггером.

Сохраняемость. Сеть Петри называется **строго сохраняющей**, если сумма фишек по всем позициям остается строго постоянной в процессе выполнения сети. Важно при моделировании перемещения ресурсов (они не должны исчезать и появляться).

Живость. Под живостью переход понимают принципиальную возможность его срабатывания при функционировании сети Петри. Тупик в сети Петри – это переход (или множество переходов), который в имеющейся маркировке μ' и в последующих достижимых из μ' маркировках не разрешен.

Достижимость. Свойство достижимости используется при установлении возможности возникновения некоторой ситуации в системы. Пусть проверяемая ситуация описывается разметкой 'μ. Возникает задача: достижима ли маркировка μ' из начальной маркировки μ0 данной сети Петри. Задача достижимости является одной из наиболее важных задач анализа сетей Петри.

Задача об обедающих философах
Пять безмолвных философов сидят вокруг круглого стола, перед каждым философом стоит тарелка спагетти. Вилки лежат на столе между каждой парой ближайших философов.

Каждый философ может либо есть, либо размышлять. Приём пищи не ограничен количеством оставшихся спагетти — подразумевается бесконечный запас. Тем не менее, философ может есть только тогда, когда держит две вилки — взятую справа и слева.

Каждый философ может взять ближайшую вилку (если она доступна), или положить — если он уже держит её.

Суть проблемы заключается в том, чтобы разработать модель поведения

(параллельный алгоритм), при котором ни один из философов не будет голодать, то есть будет вечно чередовать приём пищи и размышления.

OpenMP

Thursday, November 1, 2018 11:53

OpenMP

OpenMP - открытый стандарт для распараллеливания программ C, C++, Fortran. Это стандарт интерфейа для многопоточного программирования над общей памятью

Преимущества

- ✓ Легкость использования
- ✓ Кросс-платформенность для систем с общей памятью
- ✓ Сокрытие низкоуровневых операций
- ✓ Поддержка параллельной и последовательной версии программ

Пишется последовательная программа, которая распараллеливается путем добавления метаинформации.

Основные компоненты

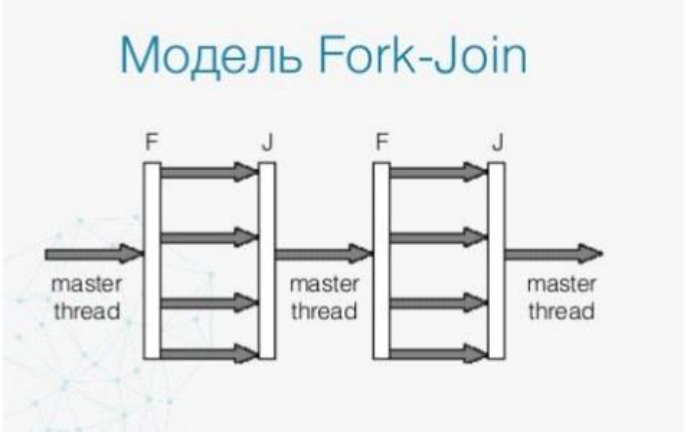
- Переменные окружения
- Директивы компилятора
- Функции в коде

Порядок разработкии программы

1. Написать и отладить последовательную программу
2. Дополнить программу директивами OpenMP
3. Скомпилировать программу компиляторов с поддержкой OpenMP
4. Задать переменные окружения
5. Запустить программу

Компиляция происходит с помощью флага -fopenmp

Модель работы



- Явное указание параллельных секиций
- Поддержка вложенного параллелизма
- Поддержка динамических потоков

Переменные окружения

- OMP_NUM_THREADS - устанавливает количество потоков в параллельном блоке. По умолчанию, количество потоков равно количеству виртуальных процессоров
- OMP_DYNAMIC - разрешает или запрещает динамическое изменение количества потоков, которые реально используются для вычислений (в зависимости от загрузки системы). Значение по умолчанию зависит от реализации
- OMP_NESTED - Разрешает или запрещает вложенный параллелизм (распараллеливание вложенных циклов). По умолчанию – запрещено.

Функции OpenMP

omp_set_num_threads	Установить количество потоков
omp_get_num_threads	Вернуть количество потоков в группе
omp_get_max_threads	Максимальное количество потоков
omp_get_thread_num	ID потока
omp_get_num_procs	Максимальное количество процессоров
omp_in_parallel	Проверка нахождения в параллельном регионе

Директивы

#pragma omp <имя> [<предложение> …]

- имя — имя директивы;
- предложение — конструкция, задающая дополнительную информацию и зависящая от директивы;

Директива parallel

```
#pragma omp parallel [<предложение>]
<структурный блок>
```

- Поток, встречающий конструкцию parallel, создает пул потоков, становясь для неё основным
- Потокам команды присваиваются уникальные номера
- Выполнение будет продолжено, когда все потоки из пула завершаться - неявный барьер

```
#include <omp.h>
int main(){
    // последовательный код
    #pragma omp parallel if (expr)
    {
        // параллельный код
    }
    // последовательный код
    return 0;
}
```

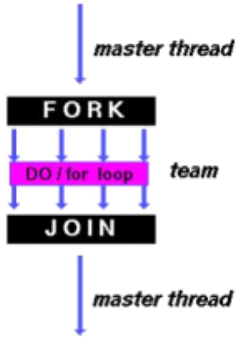
Пример

```
#include <iostream>
#include <omp.h>
int main() {
     #pragma omp parallel
     std::cout << "Hello, world!" << std::endl;
}
```

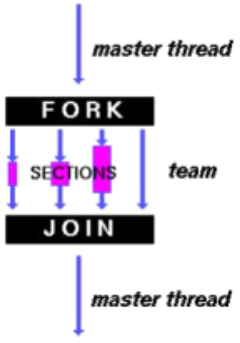
Способы разделения работы между потоками

1. Директива for
2. Директива section
3. Директива single

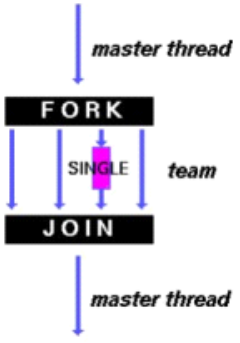
```
#pragma omp for
  for (i=0;i<N;i++)
  {
    // code
  }
```




```
#pragma omp sections
{
 #pragma omp section
 // code 1
 #pragma omp section
 // code 2
}
```




```
#pragma omp single
{
  // code
}
```



Директива for

```
#pragma omp for [<педложения>]
     <циклы for>
```

Условия:

- Счётчик должен быть целого типа или итератором произвольного доступа. Должен изменятся только в заголовке цикла
- Условие цикла - сравнение с переменной с инвариантными к циклу выражением при помощи <, <=, >, >=
- Изменение счётчика: при помощи ++, --, i += d, i -= d, i = i + d, i = d + i, i = i - d (d — инвариантное к циклу целое выражение)

Пример

```
#include <stdio.h>
#include <omp.h>
int main(){
     int i;
     #pragma omp parallel if (expr)
     {
```

```
        #pragma omp for
        for (i = 0; i < 1000; i++)
            printf("%d ", i)
    }
    return 0;
}
```

Директива sections

Позволяет самим решать, как код будет распараллелен. Секции будут выполнятся параллельно.
Модификатор private делает локальную копию I для каждого потока.
При подобном разбиении задач необходимо помнить о необходимости поддержки последовательной версии программы.

Пример

```
#include <stdio.h>
#include <omp.h>
int main(){
    int i;
    #pragma omp parallel sections private(i)
    {
        #pragma omp section
        {
            printf("1st half \n");
            for (i = 0; i < 500; i++) printf("%d ", i)
        }
        #pragma omp section
        {
            printf("2nd half \n");
            for (i = 501; i < 1000; i++) printf("%d ", i)
        }
    }
    return 0;
}
```

Директива single

Позволяет описать область кода, которая будет выполнена только одним потоком.

Пример

```
#include <stdio.h>
#include <omp.h>
int main(){
    int i;
    #pragma omp parallel private(i)
    {
        #pragma omp for
            for (i = 0; i < 1000; i++) printf("%d ", i)
        #pragma omp single
            print("I'm thread %d!\n", get_thread_num());
        #pragma omp for
            for (i = 0; i < 1000; i++) printf("%d ",
    }
    return 0;
}
```

Строчка

```
print("I'm thread %d!\n", get_thread_num());
```
создает барьер. Чтобы убрать барьер, нужно добавить модификатор nowait:
```
#pragma omp single nowait
```

Директива master

Так же, как и single, но будет выполнена только в главном потоке, который не занят параллельными вычислениями. Барьер также не создается.

Пример

```
#include <stdio.h>
#include <omp.h>
int main(){
     int i;
     #pragma omp parallel private(i)
     {
          #pragma omp for
               for (i = 0; i < 1000; i++) printf("%d ", i)
          #pragma omp master
               print("I'm Master!\n");
          #pragma omp for
               for (i = 0; i < 1000; i++) printf("%d ",
     }
     return 0;
}
```

Область видимости переменных

Переменные, объявленные внутри блока, являются локальными. Переменные, объявленные вне блока, являются общими. Если потоки меняют внешнюю переменную, может возникнуть проблема с разделяемой памятью.

Модификаторы доступа к переменным

private	Своя локальная переменная в каждом потоке	`int num;`
		`#pragma omp parallel private(num)`
		`{`
		`  num = omp_get_thread_num();`
		`  printf("%d\n",num);`
		`}`

firstprivat	Локальная переменная с инициализацией	`int num = 5;`
		`#pragma omp parallel \`
		`  firstprivate(num)`
		`{`
		`  printf("%d\n",num);`
		`}`

lastprivat	Локальная переменная с сохранением последнего значения (в последовательном исполнении)	`int i,j;`
		`#pragma omp parallel for \`
		`  lastprivate(j)`
		`for (i=0;i<100;i++) j = i;`
		
		`printf("Последний j = %d\n",j);`

shared	Явное указание разделяемой переменной	`int i,j;`
		
		`#pragma omp parallel for \`
		`  shared(j)`
		`for (i=0;i<100;i++) j = i;`
		
		`printf("j = %d\n",j);`

reduction	Переменная для выполнения редукционной операции	`int i,s = 0;`
		`#pragma omp parallel for \`
		`  reduction(+:s)`
		`  for (i=0;i<100;i++)`
		`  s += i;`
		
		`printf("Sum: %d\n",s);`

Синхронизация потоков

master	Код выполняется только главным потоком	`#pragma omp parallel`
		`{`
		`  //code`
		`  #pragma omp master`
		`  {`
		`  // critical code`
		`  }`
		`  // code`
		`}`

barrier	Выполняется после завершения всех потоков	`#pragma omp parallel`
		`{`
		`  printf("Hello!\n");`
		
		`  #pragma omp barrier`
		`  printf("I am thread %d\n",`
		`  omp_get_thread_num());`
		`}`

critical	Критическая секция. Устанавливает мьютекс на входе и выходе	`int i,idx[N],x[M];`
		
		`#pragma omp parallel for`
		`for (i=0;i<N;i++)`
		`{`
		`  #pragma omp critical`
		`  {`
		`  x[idx[i]] += count(i);`
		`  }`
		`}`

atomic	Тоже делает операцию выполненной только одним потоком, но с той разницей, что он попытается сделать это атомарно на уровне процессора.	`int i,idx[N],x[M];`
		
		`#pragma omp parallel for`
		`for (i=0;i<N;i++)`
		`{`
		`  #pragma omp atomic`
		`  x[idx[i]] += count(i);`
		`}`

Блокировки

Блокировка (мьютекс в OpenMP) - особый объект, общий для потоков.

Потоки могут захватывать (lock) и освобождать (unlock) блокировку. Только один поток в одно время может захватить блокировку. При попытке захвата блокировки потоки ждут её освобождения.
С помощью блокировок можно контролировать доступ к общим ресурсам

Пример. Использование блокировки

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int x[1000];
int main()
{ int i,max;
  omp_lock_t lock;
  omp_init_lock(&lock);
  for (i=0;i<1000;i++) x[i]=rand();
  max = x[0];
  #pragma omp parallel for
    for(i=0;i<1000;i++)
    { omp_set_lock(&lock);
      if (x[i]>max) max = x[i];
      omp_set_unlock(&lock);
    }
  omp_destroy_lock(&lock);
  return 0;
}
```

Пример. Single vs Critical

```
int a=0, b=0;
#pragma omp parallel num_threads(4)
{
    #pragma omp single
        a++;
    #pragma omp critical;
        b++;
}
printf("single: %d -- critical: %d\n", a, b);
```

Вывод: single: 1 -- critical: 4