Теория и практика многопоточного программирования

ЛЕКЦИЯ 3. ПОСЛЕДОВАТЕЛЬНОСТЬ ИСПОЛНЕНИЯ. УПОРЯДОЧЕННОСТЬ И АТОМАРНОСТЬ

В прошлый раз...

- Архитектура компьютеров с общей памятью
  - Архитектура фон Неймана
- Способы решения и создания проблем в «узких местах»
  - Шина
  - Кэш
  - Многоядерность и NUMA
  - Многопоточность и Hyper Threading

Темы лекции

Процессы и потоки

Инструкции x86

Видимость результатов

Модель упорядоченности доступа к памяти

Атомарность и атомарные примитивы

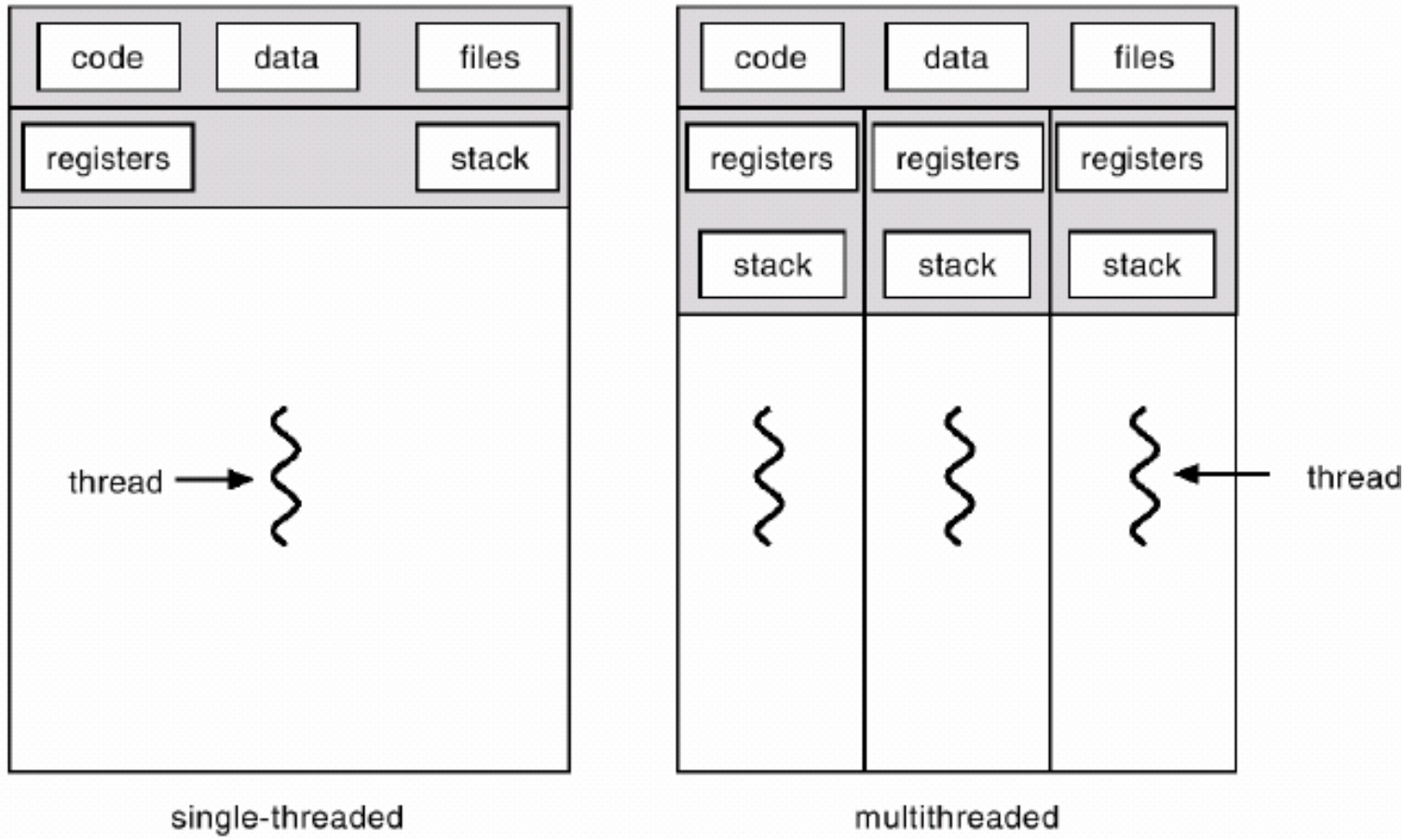
Процессы и потоки

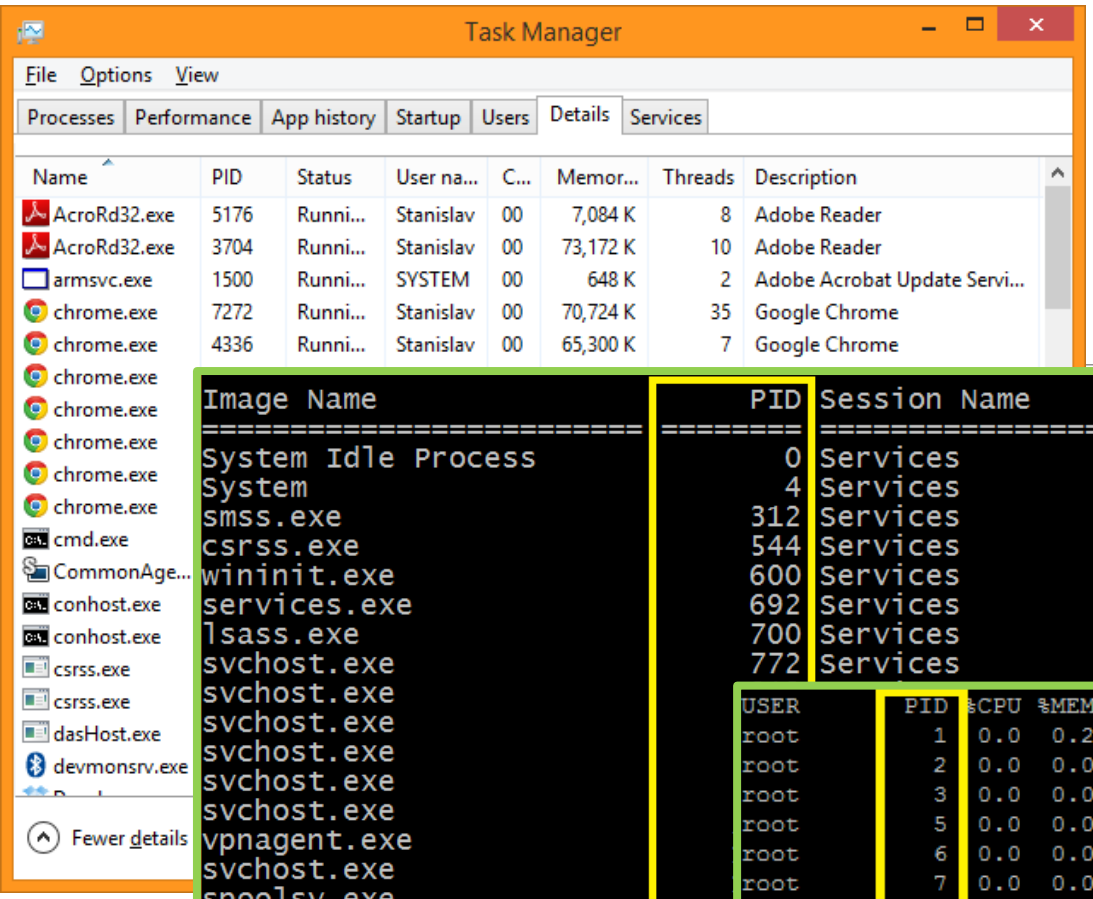
Процесс (process) – это объект операционной системы, экземпляр исполняющейся компьютерной программы. **Программа (program)** – это статичный **набор** *инструкций* (например, на диске), в то время как процесс – это инструкции в момент исполнения. Процесс обрадает дополнительным свойствами (*приоритет, потоки ввода-вывода, область памяти и т.п.*).

Инструкции в рамках процесса могут испольняться в несколько потоков (threads). «Рамки» процесса – это память и другие ресурсы.

Поток, нить (иногда – легковесный процесс) – объект, представляющий собой **последовательность** инструкций во время исполнения (часто – некоторой функции или метода). Обладает дополнительными свойствами (состояние регистров).

Процессы и потоки






```
Image Name                     PID Session Name        Session#    Mem Usage
========================= ======== ================ =========== ============
System Idle Process              0 Services                   0         24 K
System                           4 Services                   0      2,684 K
smss.exe                       312 Services                   0        624 K
csrss.exe                      544 Services                   0      3,120 K
wininit.exe                    600 Services                   0      2,412 K
services.exe                   692 Services                   0      4,816 K
lsass.exe                      700 Services                   0     12,540 K
svchost.exe                    772 Services                   0     11,432 K
svchost.exe
svchost.exe
svchost.exe
svchost.exe
svchost.exe
vpnagent.exe
svchost.exe
spoolsv.exe
svchost.exe
armsvc.exe
```

TASKLIST

```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2   3516  1284 ?        Ss   2013   0:15 /sbin/init
root         2  0.0  0.0      0     0 ?        S    2013   0:01 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    2013   0:59 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S    2013   0:00 [kworker/u:0]
root         6  0.0  0.0      0     0 ?        S    2013   0:00 [migration/0]
root         7  0.0  0.0      0     0 ?        S    2013   1:22 [watchdog/0]
root         8  0.0  0.0      0     0 ?        S<   2013   0:00 [cpuset]
root         9  0.0  0.0      0     0 ?        S<   2013   0:00 [khelper]
root        10  0.0  0.0      0     0 ?        S    2013   0:00 [kdevtmpfs]
root        11  0.0  0.0      0     0 ?        S<   2013   0:00 [netns]
root        12  0.0  0.0      0     0 ?        S    2013   0:25 [sync_supers]
root        13  0.0  0.0      0     0 ?        S    2013   0:00 [bdi-default]
root        14  0.0  0.0      0     0 ?        S<   2013   0:00 [kintegrityd]
root        15  0.0  0.0      0     0 ?        S<   2013   0:00 [kblockd]
root        16  0.0  0.0      0     0 ?        S<   2013   0:00 [ata_sff]
root        17  0.0  0.0      0     0 ?        S    2013   0:00 [khubd]
root        18  0.0  0.0      0     0 ?        S<   2013   0:00 [md]
root        20  0.0  0.0      0     0 ?        S    2013   0:00 [kworker/u:1]
root        21  0.0  0.0      0     0 ?        S    2013   0:02 [khungtaskd]
root        22  0.0  0.0      0     0 ?        S    2013   0:02 [kswapd0]
root        23  0.0  0.0      0     0 ?        SN   2013   0:00 [ksmd]
```

ps aux

Команды x86/x64

Многие операции с памятью **неатомарны** и проходят через регистры

Атомарные операции — операции, выполняющиеся как единое целое либо не выполняющиеся вовсе.

Наблюдать результаты работы инструкции можно только выполняя другие инструкции, т.е. результат неизвестен до выполнения другой инструкции

Примеры атомарных операций

Чтение и запись

- **mov** to{reg|addr}, from{reg|addr}
- ~~mov addr, addr~~

- **xchg** {reg|addr}, {reg|addr}
- **xadd a, b**

Сравнить-и-поменять-местами (compare-and-swap)

- **cmpxchg** destrAddr, srcAddr

Пример

```
void spin_lock(int volatile *p)
{
    while(!__sync_bool_compare_and_swap(p, 0, 1))
    {
        while(*p) _mm_pause();
    }
}

void spin_unlock(int volatile *p)
{
    asm volatile (""); // acts as a memory barrier.
    *p = 0;
}
```

Порядок выполнения

Оператор != Инструкция

Порядок инструкций может не соответсвовать ожиданию, и управляется платформой и компилятором:

- Можно переставить [5] и [6]
- Нельзя переставить [2] и [3]

Промежуточное значение оператора может быть ненаблюдаемо (если [5] в регистре)

```
  …
1 int a=2,b;
2 a++;
3 b = a+1;
  …
4 int a=1,b=2;
5 a++;          // a == 2
6 b++;          // b == 3
7 a += b;       // a == 5
  …
```

Попробуем заставить

Ключи и директивы компилятора
`(-O0)` и `#pragma optimize`

Директива `volatile` – запрет на оптимизация доступа к этой области памяти… В пределах «наблюдаемого поведения»

- `volatile char A[N]` vs `volatile char *A;`

Барьеры памяти (memory fence, memory barrier):

- Полный (**Full**) = A + R
- **Accrue** – все RW после барьера не попадут до него (VR)
- **Release** – все RW до барьера не попадут после него (VW)

Упорядоченность не помогает синхронизации.
Синхронизации помогают примитивы синхронизации.

Атомарность

Ключевое понятие – **видимость** (результата)

Условия атомарности:

Невидимость - до момента завершения всех операций никто в системе (никакой процесс) не может наблюдать факт произведения изменений

Восстановимость - если операция по какой-либо причине не завершилась успешно, то состояние системы должно быть полностью восстановлено к моменту до начала работы атомарной операции

Выводы

Существуют понятия «**атомарность**», «**видимость результата**», «**упорядоченность**». Понятия атомарности и упорядоченности независимы.

Кроме программиста на **порядок выполнения** программы влияют компилятор и платформа. Отсюда проблема переносимости кода.

На любой платформе есть **атомарные примитивы**, на основе которых строится синхронизация