

Mid-term Review

高级大数据解析

Linux command

目录操作命令

- **ls** 显示目录中的内容
- **pwd** 显示当前和工作目录
- **cd** 改变用户工作目录
- **mkdir** 建立用户目录
- **rmdir** 删除目录

-1 每列仅显示一个文件或目录名称

-a 显示所有文件或目录，包括以“.”为名称开头字符的文件、现行目录“.”与上层目录“..”

-l 使用详细格式列表。将权限标示、硬件接数目、拥有者与群组名称、文件或目录大小及更改时间一并列出

-R 递归处理，将指定目录下的所有文件及子目录一并处理

使用长列表方式列出某个子目录中的全部文件，
使用下面的命令：

```
[root@legend /root] # ls -la
total 16
drwxr-xr-x  4 root  root  4096 Jan  1 11:28 .
drwxr-x--- 11 root  root  4096 Jan  1 11:27 ..
drwxr-xr-x  2 root  root  4096 Jan  1 11:27
team01
drwxr-xr-x  2 root  root  4096 Jan  1 11:28
team02
```

列出子目录中以字母t打头的全部非隐藏文件，
使用下面的命令：

```
[root@legend /root] # ls t*
```

使用cd进入目录

```
# cd /home/111
```

```
# pwd
```

```
/home/111
```

“..”代表上一级目录

```
# cd ..
```

```
#pwd
```

```
/home
```

文件操作命令

cp 复制文件或目录

mv 移动文件和文件换名

rm 删除文件或目录

ln 在文件间建立连接

find 查找特定的文件

touch 改变文件的时间参数

建立用户目录命令**mkdir**

mkdir可以建立目录同时还可以给目录设置权限。

mkdir [-p] [-m][文件名]

-p 若所要建立目录的上层目录目前尚未建立，则会一并建立上层目录

-m 建立目录时，同时设置目录的权限。权限的设置法与**chmod** 指令相同

cat

显示和合并文件

more

分屏显示文件

head

显示文件的前几行

tail

显示文件的最后几行

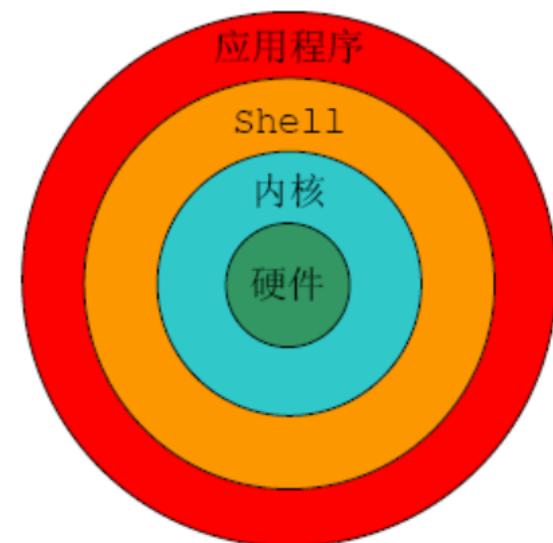
Comparing High-Performance Capabilities

Aspect	Typical Scenario	Big Data
Application development	Applications that take advantage of massive parallelism developed by specialized developers skilled in high-performance computing, performance optimization, and code tuning	A simplified application execution model encompassing a distributed file system, application programming model, distributed database, and program scheduling is packaged within Hadoop, an open source framework for reliable, scalable, distributed, and parallel computing
Platform	Uses high-cost massively parallel processing (MPP) computers, utilizing high-bandwidth networks, and massive I/O devices	Innovative methods of creating scalable and yet elastic virtualized platforms take advantage of clusters of commodity hardware components (either cycle harvesting from local resources or through cloud-based utility computing services) coupled with open source tools and technology
Data management	Limited to file-based or relational database management systems (RDBMS) using standard row-oriented data layouts	Alternate models for data management (often referred to as NoSQL or “Not Only SQL”) provide a variety of methods for managing information to best suit specific business process needs, such as in-memory data management (for rapid access), columnar layouts to speed query response, and graph databases (for social network analytics)
Resources	Requires large capital investment in purchasing high-end hardware to be installed and managed in-house	The ability to deploy systems like Hadoop on virtualized platforms allows small and medium businesses to utilize cloud-based environments that, from both a cost accounting and a practical perspective, are much friendlier to the bottom line

Shell 简介

□ shell 是系统的用户界面，它提供了用户和 Linux（内核）之间进行交互操作的一种接口。用户在命令行中输入的每个命令都由 shell 先解释，然后传给 Linux 内核去执行。

□ 如果把 Linux 内核想象成一个球体的中心，shell 就是围绕内核的外层，从 shell 向 Linux 操作系统传递命令时，内核就会做出相应的反应。



Bash 的功能

□ 命令行

当用户打开一个（虚拟）终端时，可以看到一个 **shell** 提示符，标识了命令行的开始。用户可以在提示符后面输入任何命令

command [选项] [参数]

例： **ls -l /home/jypan/linux/**

注意：命令行中选项先于参数输入

管道

□ 管道

◆ UNIX 系统的一个基本哲学是：一连串的小命令能够解决大问题。其中每个小命令都能够很好地完成一项单一的工作。现在需要有一些东西能够将这些简单的命令连接起来，这样管道就应运而生。

◆ 管道“|”的基本含义是：将前一个命令的输出作为后一个命令的输入。如：

```
ls /local | du -sh *
```

◆ 利用管道可以实现一些很强的功能。

管道举例

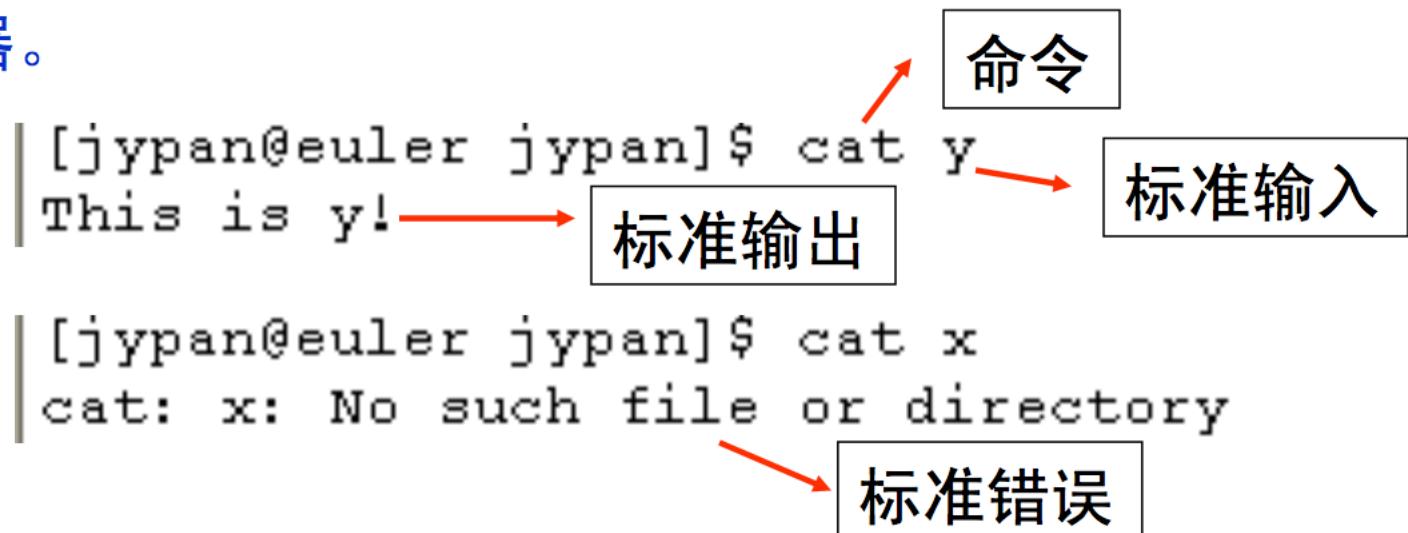
一个较复杂的例子：输出系统中用户名的一个排序列表。这里需要用到三个命令：`cat`、`awk`、`sort`，其中 `cat` 用来显示文件 `/etc/passwd` 的内容，`awk` 用来提取用户名，`sort` 用来排序。

```
cat /etc/passwd | \
awk -F: '{print $1}' | \
sort
```

重定向

□ 数据流

- ◆ Linux 中的数据流有三种：标准输入 (STDIN)、标准输出 (STDOUT) 和标准错误 (STDERR)。
- ◆ 标准输入通常来自键盘，标准输出是命令的结果，通常定向到显示器，标准错误是错误信息，通常也定向到显示器。



重定向

□ 输入输出重定向

◆ 输入重定向：“<”

可以使用文件中的内容作为命令的输入。

◆ 输出重定向：“>”

允许将命令的输出结果保存到一个文件中。

```
ls > list
```

```
sort < list > sort_list
```

□ 输入输出重定向

◆ 在使用输出重定向时，如果输出文件已经存在，则原文件中的内容将被删除。

◆ 如果希望保留原文件的内容，可以使用“>>”代替“>”，这样重定向输出的内容将添加到原文件的后面。

```
ls / > list
```

```
ls /home/ >> list
```

shell 变量大致可以分为三类:**内部变量**、**用户变量**和**环境变量**。

变量名	含义
HOME	用户主目录
LOGNAME	登录名
USER	用户名, 与登录名相同
PWD	当前目录/工作目录名
MAIL	用户的邮箱路径名
HOSTNAME	计算机的主机名
INPUTRC	默认的键盘映像
SHELL	用户所使用的 shell 的路径名
LANG	默认语言
HISTSIZE	history 所能记住的命令的最多个数
PATH	shell 查找用户输入命令的路径(目录列表)
PS1、PS2	shell 一级、二级命令提示符

bash 配置文件

□ bash 配置文件

- ◆ 在命令行中设置和修改的变量值，只在当前的 shell 中有效。一旦用户退出 bash，所做的一切改变都会丢失。
- ◆ 在启动交互式会话过程中，在出现提示符前，系统会读取几个配置文件，并执行这些文件中的命令。所以这些文件可以用来定制 bash 环境。如：设置 shell 变量值或建立别名等。
- ◆ **bash 配置文件：**

/etc/profile

~/.bash_profile

~/.bashrc

~/.bash_login

~/.profile

进程的优先级

- 进程运行后调整 **nice** 值: **renice**

进程已经运行，此时又有许多用户登录，他们使得各个进程分得的 CPU 时间下降。此时，**root** 可以提高进程的优先权，但普通用户没这个权限，在系统资源紧张时，只能通过降低其它不着急的进程的优先权，从而使得急用的进程能分得更多的 CPU 时间。

```
renice n [-p pid] [-u user] [-g pgid]
```

- 增加指定进程的 **nice** 值
- **n** 可以是正的，也可以是负数；
- 注意与 **nice** 命令的区别：没有减号
- **pgid** 是进程组的 ID

常用 bash 内部命令

□ 一些常用的 bash 内部命令

- **alias/unalias** : 设置和取消 bash 别名。
- **bg**: 使一个被挂起的进程在后台继续执行。
- **cd**: 切换当前工作目录。
- **exit**: 退出 shell。
- **export**: 使变量的值对当前 shell 的所有子进程都有效。
- **fc**: 用来显示和编辑历史命令列表里的命令。
- **fg**: 使一个被挂起的进程在前台继续执行。
- **help**: 显示帮助信息。
- **kill**: 终止某个进程。
- **pwd**: 显示当前工作目录。

- **id**: print real and effective UIDs and GIDs
 - **who**: show who is logged on
 - **whoami**: **id -un**
 - **hostname**: show or set the system's host name
 - **w**: show who is logged on and what they are doing
 - **last**: show listing of last logged in users
 - **finger**: displays information about the system users
 - **top**: display Linux tasks (很有用的系统监控工具)
- 更多 bash 内部命令见: **man bash --> 3370**
或任一 bash 内部命令的 manual, 例: **man bg**



Shell 脚本

□ Shell 脚本

当命令不在命令行中执行，而是从一个文件中执行时，该文件就称为 **shell** 脚本。**shell** 脚本按行解释。

□ Shell 脚本的编写

- **Shell** 脚本是纯文本文件，可以使用任何文本编辑器编写
- **Shell** 脚本通常是以 .sh 作为后缀名

□ Shell 脚本的执行

```
chmod +x script_name  
./script_name  
  
sh script_name
```

Shell 脚本的格式

- 第一行：指定用哪个程序来编译和执行脚本。

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

```
#!/bin/sh
```

```
#!/bin/csh
```

- 可执行语句和 **shell** 控制结构

一个 **shell** 脚本通常由一组 **Linux** 命令、**shell** 命令、控制结构和注释语句构成。

- 注释：以“#”开头，可独占一行，或跟在语句的后面。

在脚本中多写注释语句是一个很好的编程习惯

Manipulating Data on Linux

- Login to your linux server via SSH
 - putty, SSH Secure Shell

```
jatten@xiv-linux-mint: ~
Using username "jatten".
Using keyboard-interactive authentication.
Password:
Welcome to Linux Mint 13 Maya (GNU/Linux 3.2.0-23-generic x86_64)

Welcome to Linux Mint
 * Documentation: http://www.linuxmint.com

505 packages can be updated.
0 updates are security updates.

Last login: Sat Nov 17 13:52:48 2012 from xivmain.local
jatten@xiv-linux-mint ~ $ cd .ssh
-bash: cd: .ssh: No such file or directory
jatten@xiv-linux-mint ~ $ mkdir .ssh
jatten@xiv-linux-mint ~ $
```

```
\maketitle

We are given an input consisting of a triangular table of non-negative
real numbers  $A[i,j]$  defined for  $1 \leq j \leq i \leq n$ . The interpretation
is that  $A[i,j]$  denotes the amount of water that is poured into bucket
 $j$  at time  $i$ , assuming buckets are numbered from left to right.

A \emph{bucket sequence} is a function  $s$  mapping the set
 $\{1, 2, \dots, n\}$  to
itself, such that  $s(i) \leq i$  for all  $i$ . The interpretation is that
 $s(i)$  denotes the bucket that Mickey empties at time  $i$ . (Again
--u---F1 doc.tex 1/10 Tue 20:04 (LaTeX Fill)--L19--16%--)
Wrote /home/.../doQment/doc/doc.tex
```

Manipulating Data on Linux

- Viewing and Manipulating the data
 - The grep family

```
$ cat demo_file
THIS LINE IS THE 1ST UPPER CASE LINE IN THIS FILE.
this line is the 1st lower case line in this file.
This Line Has All Its First Character Of The Word With Upper Case.

Two lines above this line is empty.
And this is the last line.
```

```
$ grep "this" demo_file
this line is the 1st lower case line in this file.
Two lines above this line is empty.
And this is the last line.
```

Manipulating Data on Linux

- Viewing and Manipulating the data
 - The grep family

```
$ cp demo_file demo_file1

$ grep "this" demo_*
demo_file:this line is the 1st lower case line in this file.
demo_file:Two lines above this line is empty.
demo_file:And this is the last line.

demo_file1:this line is the 1st lower case line in this file.
demo_file1:Two lines above this line is empty.
demo_file1:And this is the last line.
```

Manipulating Data on Linux

- Viewing and Manipulating the data
 - awk

```
1) Amit Physics 80
2) Rahul Maths 90
3) Shyam Biology 87
4) Kedar English 85
5) Hari History 89
```

```
[jerry]$ awk '/a/ {print $0}' marks.txt
```

On executing the above code, you get the following result:

```
2) Rahul Maths 90
3) Shyam Biology 87
4) Kedar English 85
5) Hari History 89
```

Manipulating Data on Linux

- Viewing and Manipulating the data
 - sort

Sort a file in ascending order

```
$ sort names.txt
```

Sort a file in descending order

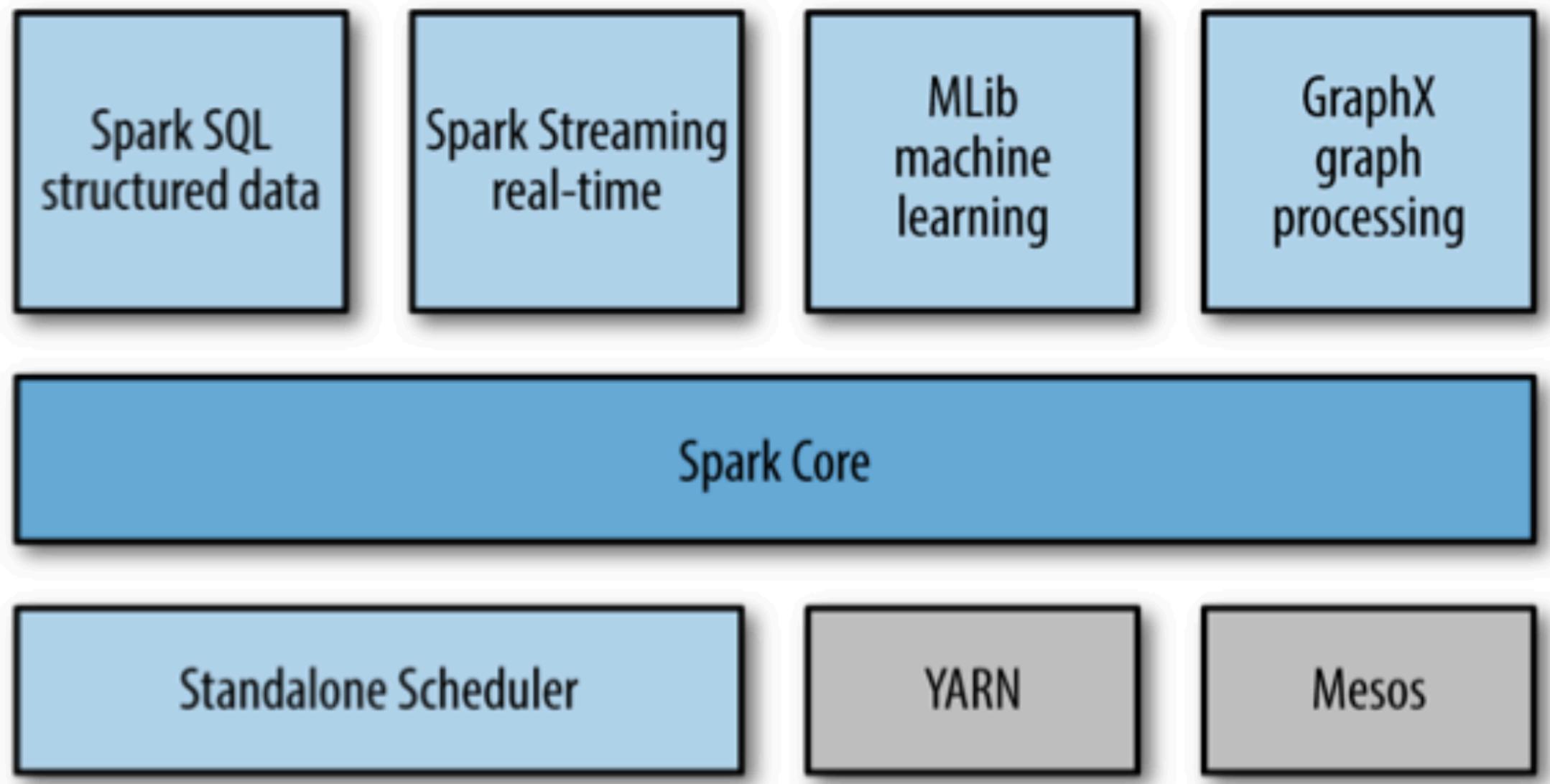
```
$ sort -r names.txt
```

Sort passwd file by 3rd field.

```
$ sort -t: -k 3n /etc/passwd | more
```

Spark

Spark Stack



PySparkShell - Spark Jobs

localhost:4040/jobs/

Apps Program researcher deep_learning computer vision 研究相关 tax OCR

APACHE 2.1.0 Jobs Stages Storage Environment Executors SQL

Spark Jobs [\(?\)](#)

User: yanwei

Total Uptime: 16 s

Scheduling Mode: FIFO

▶ [Event Timeline](#)

Running as a Standalone Application

In Python, you simply write applications as Python scripts, but you must run them using the `bin/spark-submit` script included in Spark. The `spark-submit` script includes the Spark dependencies for us in Python. This script sets up the environment for Spark's Python API to function. Simply run your script with the line given in [Example 2-6](#).

Example 2-6. Running a Python script

```
bin/spark-submit my_script.py
```

Example — word count

```
import sys
from operator import add

from pyspark import SparkContext

if __name__ == "__main__":
    if len(sys.argv) != 2:
        print >> sys.stderr, "Usage: wordcount <file>"
        exit(-1)
    sc = SparkContext(appName="PythonWordCount")
    lines = sc.textFile(sys.argv[1], 1)
    counts = lines.flatMap(lambda x: x.split(' '))
                  .map(lambda x: (x, 1))
                  .reduceByKey(add)
    output = counts.collect()
    for (word, count) in output:
        print "%s: %i" % (word, count)

    sc.stop()
```

Resilient Distributed Dataset (RDD) Basics

- An RDD in Spark is an immutable distributed collection of objects.
- Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster.
- Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects in their driver program.
- Once created, RDDs offer two types of operations: **transformations** and **actions**.

```
>>> lines = sc.textFile("README.md")                                <== create RDD

>>> pythonLines = lines.filter(lambda line: "Python" in line)    <== transformation

>>> pythonLines.first()                                         <== action
u'## Interactive Python Shell'
```

Transformations and actions are different because of the way Spark computes RDDs.
==> Only computes when something is, the first time, in an action.

What's RDD?

[What] 什么是Resilient Distributed Datasets (**RDD**) ?

- 不变的、容错的、并行的数据结构
- 显式地将数据存储到磁盘和**内存**中，并能控制数据的分区

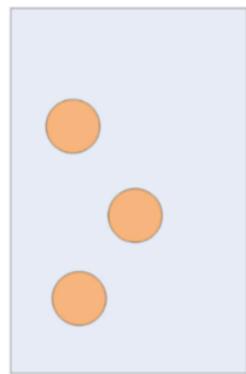
[Why] 为什么要RDD？

- 数据处理常见模型：1. Iterative Algorithms (迭代算法) ; 2.Relational Queries (关联查询) ; 3.MapReduce; 4. Stream Processing (流式处理)
- RDD支持**四种模型**

[How] 如何构建RDD？

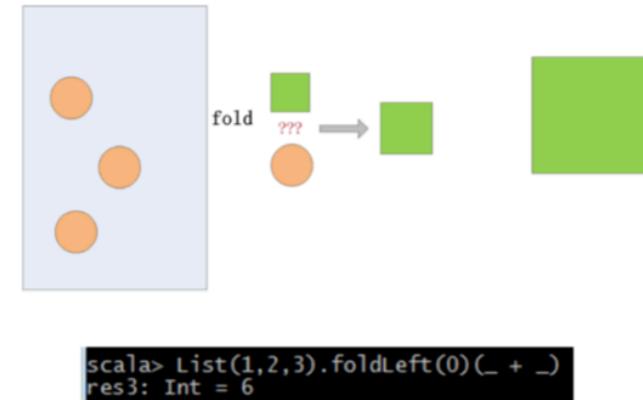
- 从文件系统
- 通过Scala集合对象并行化生成
- 通过对已存在的RDD transform生成
- 通过改变其他RDD的持久化状态

MAP运算



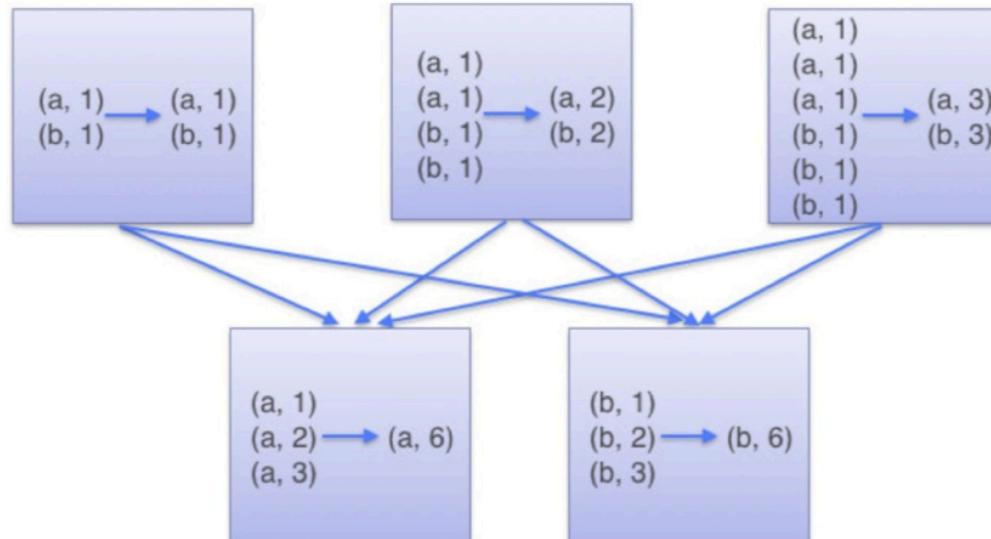
```
scala> List(1,2,3).map(_.toString)  
res1: List[String] = List(1, 2, 3)
```

FOLD/REDUCE运算

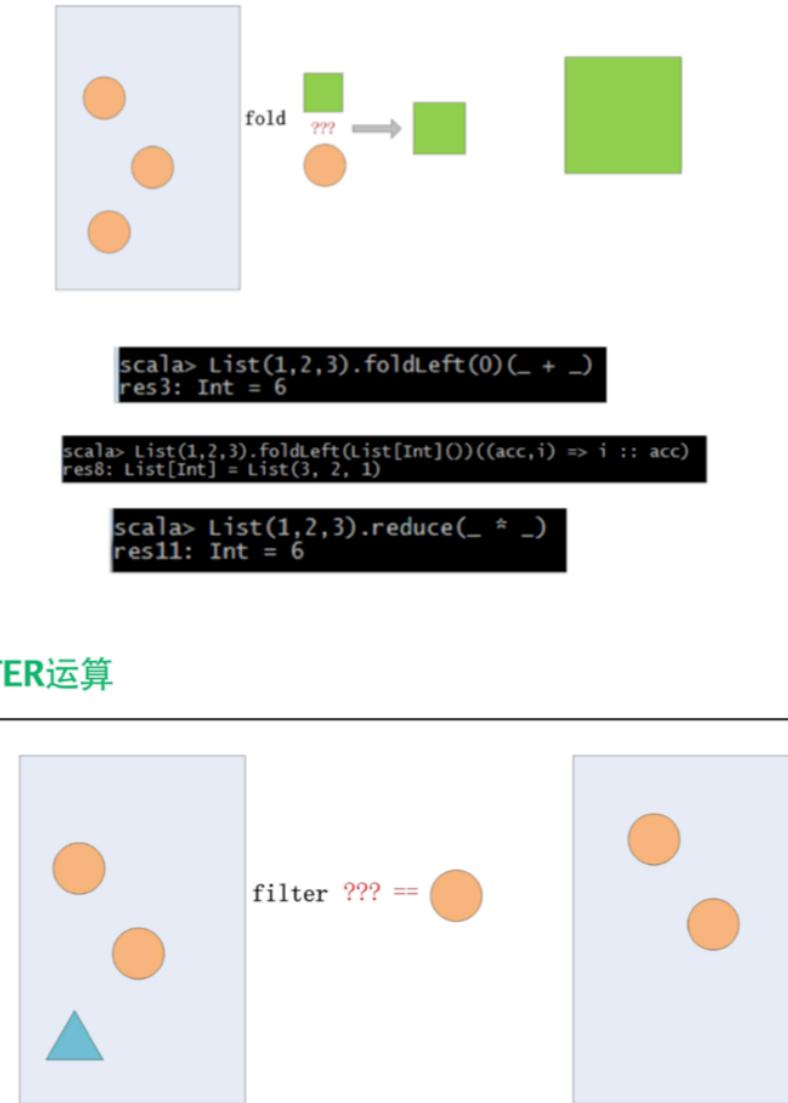


```
scala> List(1,2,3).reduce(_ * _)  
res11: Int = 6
```

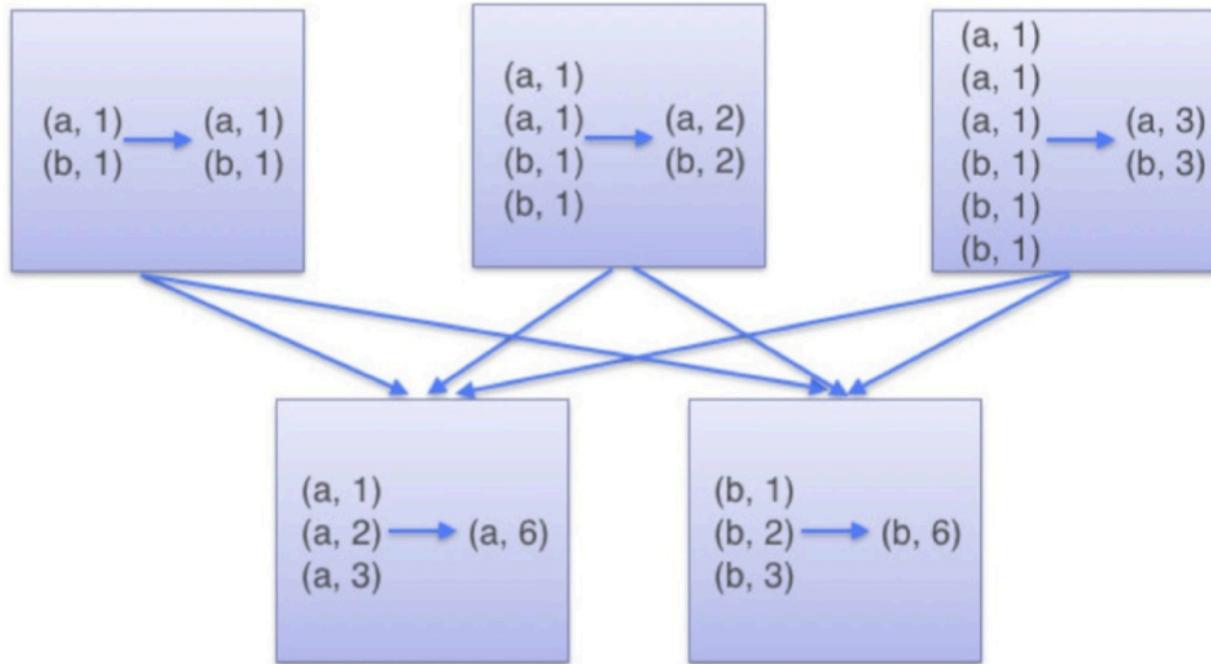
ReduceByKey



FILTER运算



ReduceByKey



FLATTEN运算



RDD Transformations和Actions

Transformations	$map(f : T \Rightarrow U)$: $\text{RDD}[T] \Rightarrow \text{RDD}[U]$ $filter(f : T \Rightarrow \text{Bool})$: $\text{RDD}[T] \Rightarrow \text{RDD}[T]$ $flatMap(f : T \Rightarrow \text{Seq}[U])$: $\text{RDD}[T] \Rightarrow \text{RDD}[U]$ $sample(fraction : \text{Float})$: $\text{RDD}[T] \Rightarrow \text{RDD}[T]$ (Deterministic sampling) $groupByKey()$: $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, \text{Seq}[V])]$ $reduceByKey(f : (V, V) \Rightarrow V)$: $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $union()$: $(\text{RDD}[T], \text{RDD}[T]) \Rightarrow \text{RDD}[T]$ $join()$: $(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (V, W))]$ $cogroup()$: $(\text{RDD}[(K, V)], \text{RDD}[(K, W)]) \Rightarrow \text{RDD}[(K, (\text{Seq}[V], \text{Seq}[W]))]$ $crossProduct()$: $(\text{RDD}[T], \text{RDD}[U]) \Rightarrow \text{RDD}[(T, U)]$ $mapValues(f : V \Rightarrow W)$: $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, W)]$ (Preserves partitioning) $sort(c : \text{Comparator}[K])$: $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$ $partitionBy(p : \text{Partitioner}[K])$: $\text{RDD}[(K, V)] \Rightarrow \text{RDD}[(K, V)]$
Actions	$count()$: $\text{RDD}[T] \Rightarrow \text{Long}$ $collect()$: $\text{RDD}[T] \Rightarrow \text{Seq}[T]$ $reduce(f : (T, T) \Rightarrow T)$: $\text{RDD}[T] \Rightarrow T$ $lookup(k : K)$: $\text{RDD}[(K, V)] \Rightarrow \text{Seq}[V]$ (On hash/range partitioned RDDs) $save(path : \text{String})$: Outputs RDD to a storage system, e.g., HDFS

Persistence in Spark

- By default, RDDs are computed each time you run an action on them.
- If you like to reuse an RDD in multiple actions, you can ask Spark to persist it using `RDD.persist()`.
- `RDD.persist()` will then store the RDD contents in memory and reuse them in future actions.
- Persisting RDDs on disk instead of memory is also possible.
- The behavior of not persisting by default seems to be unusual, but it makes sense for big data.

Example 3-4. Persisting an RDD in memory

```
>>> pythonLines.persist  
  
>>> pythonLines.count()  
2  
  
>>> pythonLines.first()  
u'## Interactive Python Shell'
```

To summarize, every Spark program and shell session will work as follows:

1. Create some input RDDs from external data.
2. Transform them to define new RDDs using transformations like `filter()`.
3. Ask Spark to `persist()` any intermediate RDDs that will need to be reused.
4. Launch actions such as `count()` and `first()` to kick off a parallel computation, which is then optimized and executed by Spark.

The simplest way to create RDDs is to take an existing collection in your program and pass it to `SparkContext`'s `parallelize()` method. But, that needs all dataset in memory on one machine.

Example 3-5. `parallelize()` method in Python

```
lines = sc.parallelize(["pandas", "i like pandas"])
```

Table 5-1. Common supported file formats

Format name	Structured	Comments
Text files	No	Plain old text files. Records are assumed to be one per line.
JSON	Semi	Common text-based format, semistructured; most libraries require one record per line.
CSV	Yes	Very common text-based format, often used with spreadsheet applications.
SequenceFiles	Yes	A common Hadoop file format used for key/value data.
Protocol buffers	Yes	A fast, space-efficient multilanguage format.
Object files	Yes	Useful for saving data from a Spark job to be consumed by shared code. Breaks if you change your classes, as it relies on Java Serialization.

Loading and Saving

Example 5-1. Loading a text file in Python

```
input = sc.textFile("file:///home/holden/repos/spark/README")
```

Example 5-5. Saving as a text file in Python

```
result.saveAsTextFile(outputFile)
```

Example 5-6. Loading unstructured JSON in Python

```
import json
data = input.map(lambda x: json.loads(x))
```

Example 5-9. Saving JSON in Python

```
(data.filter(lambda x: x['lovesPandas']).map(lambda x: json.dumps(x))
 .saveAsTextFile(outputFile))
```

Example 5-12. Loading CSV with `textFile()` in Python

```
import csv
import StringIO
...
def loadRecord(line):
    """Parse a CSV line"""
    input = StringIO.StringIO(line)
    reader = csv.DictReader(input, fieldnames=["name", "favouriteAnimal"])
    return reader.next()
input = sc.textFile(inputFile).map(loadRecord)
```

Example 5-18. Writing CSV in Python

```
def writeRecords(records):
    """Write out CSV lines"""
    output = StringIO.StringIO()
    writer = csv.DictWriter(output, fieldnames=["name", "favoriteAnimal"])
    for record in records:
        writer.writerow(record)
    return [output.getvalue()]
pandaLovers.mapPartitions(writeRecords).saveAsTextFile(outputFile)
```

Example 5-20. Loading a SequenceFile in Python

```
val data = sc.sequenceFile(inFile,
    "org.apache.hadoop.io.Text", "org.apache.hadoop.io.IntWritable")
```

Example of Spark Programming

Example 6-1. Sample call log entry in JSON, with some fields removed

```
{"address": "address here", "band": "40m", "callsign": "KK6JLK", "city": "SUNNYVALE",
"contactlat": "37.384733", "contactlong": "-122.032164",
"county": "Santa Clara", "dxcc": "291", "fullname": "MATTHEW McPherrin",
"id": 57779, "mode": "FM", "mylat": "37.751952821", "mylong": "-122.4208688735", ...}
```

Example 6-2. Accumulator empty line count in Python

```
file = sc.textFile(inputFile)
# Create Accumulator[Int] initialized to 0
blankLines = sc.accumulator(0)

def extractCallSigns(line):
    global blankLines    # Make the global variable accessible
    if (line == ""):
        blankLines += 1
    return line.split(" ")

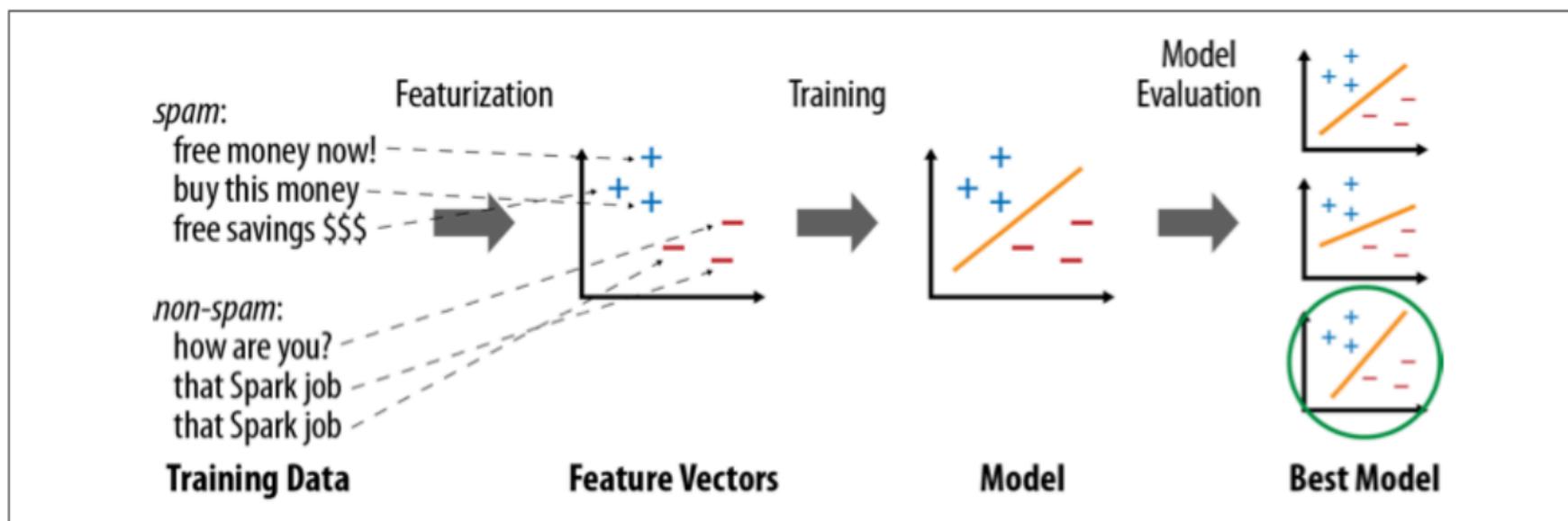
callSigns = file.flatMap(extractCallSigns)
callSigns.saveAsTextFile(outputDir + "/callsigns")
print "Blank lines: %d" % blankLines.value
```

Example 6-19. Removing outliers in Python

```
# Convert our RDD of strings to numeric data so we can compute stats and
# remove the outliers.
distanceNumerics = distances.map(lambda string: float(string))
stats = distanceNumerics.stats()
stddev = std.stdev()
mean = stats.mean()
reasonableDistances = distanceNumerics.filter(
    lambda x: math.fabs(x - mean) < 3 * stddev)
print reasonableDistances.collect()
```

Machine Learning Library in Spark — MLlib

1. Start with an RDD of strings representing your messages.
2. Run one of MLlib's *feature extraction* algorithms to convert text into numerical features (suitable for learning algorithms); this will give back an RDD of vectors.
3. Call a classification algorithm (e.g., logistic regression) on the RDD of vectors; this will give back a model object that can be used to classify new points.
4. Evaluate the model on a test dataset using one of MLlib's evaluation functions.



Example 11-1. Spam classifier in Python

```
from pyspark.mllib.regression import LabeledPoint
from pyspark.mllib.feature import HashingTF
from pyspark.mllib.classification import LogisticRegressionWithSGD

spam = sc.textFile("spam.txt")
normal = sc.textFile("normal.txt")

# Create a HashingTF instance to map email text to vectors of 10,000 features.
tf = HashingTF(numFeatures = 10000)
# Each email is split into words, and each word is mapped to one feature.
spamFeatures = spam.map(lambda email: tf.transform(email.split(" ")))
normalFeatures = normal.map(lambda email: tf.transform(email.split(" ")))

# Create LabeledPoint datasets for positive (spam) and negative (normal) examples.

positiveExamples = spamFeatures.map(lambda features: LabeledPoint(1, features))
negativeExamples = normalFeatures.map(lambda features: LabeledPoint(0, features))
trainingData = positiveExamples.union(negativeExamples)
trainingData.cache() # Cache since Logistic Regression is an iterative algorithm.

# Run Logistic Regression using the SGD algorithm.
model = LogisticRegressionWithSGD.train(trainingData)

# Test on a positive example (spam) and a negative one (normal). We first apply
# the same HashingTF feature transformation to get vectors, then apply the model.
posTest = tf.transform("O M G GET cheap stuff by sending money to ...".split(" "))
negTest = tf.transform("Hi Dad, I started studying Spark the other ...".split(" "))
print "Prediction for positive test example: %g" % model.predict(posTest)
print "Prediction for negative test example: %g" % model.predict(negTest)
```

Feature Extraction Example — TF-IDF

Example 11-7. Using HashingTF in Python

```
>>> from pyspark.mllib.feature import HashingTF  
  
>>> sentence = "hello hello world"  
>>> words = sentence.split() # Split sentence into a list of terms  
>>> tf = HashingTF(10000) # Create vectors of size S = 10,000  
>>> tf.transform(words)  
SparseVector(10000, {3065: 1.0, 6861: 2.0})  
  
>>> rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())  
>>> tfVectors = tf.transform(rdd) # Transforms an entire RDD
```

Example 11-8. Using TF-IDF in Python

```
from pyspark.mllib.feature import HashingTF, IDF  
  
# Read a set of text files as TF vectors  
rdd = sc.wholeTextFiles("data").map(lambda (name, text): text.split())  
tf = HashingTF()  
tfVectors = tf.transform(rdd).cache()  
  
# Compute the IDF, then the TF-IDF vectors  
idf = IDF()  
idfModel = idf.fit(tfVectors)  
tfIdfVectors = idfModel.transform(tfVectors)
```

Naive Bayes

Training set:

sex	height (feet)	weight (lbs)	foot size(inches)
male	6	180	12
male	5.92 (5'11")	190	11
male	5.58 (5'7")	170	12
male	5.92 (5'11")	165	10
female	5	100	6
female	5.5 (5'6")	150	8
female	5.42 (5'5")	130	7
female	5.75 (5'9")	150	9

Classifier using Gaussian distribution assumptions:

sex	mean (height)	variance (height)	mean (weight)	variance (weight)	mean (foot size)	variance (foot size)
male	5.855	3.5033e-02	176.25	1.2292e+02	11.25	9.1667e-01
female	5.4175	9.7225e-02	132.5	5.5833e+02	7.5	1.6667e+00

Test Set:

sex	height (feet)	weight (lbs)	foot size(inches)
sample	6	130	8

$$\text{posterior}(\text{male}) = \frac{P(\text{male}) p(\text{height}|\text{male}) p(\text{weight}|\text{male}) p(\text{footsize}|\text{male})}{\text{evidence}}$$

$$\begin{aligned} \text{evidence} &= P(\text{male}) p(\text{height}|\text{male}) p(\text{weight}|\text{male}) p(\text{footsize}|\text{male}) \\ &+ P(\text{female}) p(\text{height}|\text{female}) p(\text{weight}|\text{female}) p(\text{footsize}|\text{female}) \end{aligned}$$

$$P(\text{male}) = 0.5$$

$$p(\text{height}|\text{male}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-(6 - \mu)^2}{2\sigma^2}\right) \approx 1.5789,$$

$$p(\text{weight}|\text{male}) = 5.9881 \cdot 10^{-6}$$

$$p(\text{foot size}|\text{male}) = 1.3112 \cdot 10^{-3}$$

$$\text{posterior numerator (male)} = \text{their product} = 6.1984 \cdot 10^{-9}$$

$$P(\text{female}) = 0.5$$

$$p(\text{height}|\text{female}) = 2.2346 \cdot 10^{-1}$$

$$p(\text{weight}|\text{female}) = 1.6789 \cdot 10^{-2}$$

$$p(\text{foot size}|\text{female}) = 2.8669 \cdot 10^{-1}$$

$$\text{posterior numerator (female)} = \text{their product} = 5.3778 \cdot 10^{-4}$$

=> female

Hadoop Vs. Spark

应用场景

Hadoop – 极大数据量 > TB - PB 级

- 单次海量数据的离线分析处理
- 大规模 Web 信息搜索
- 数据密集型并行计算

Spark – 内存可容纳 ≈ TB 级

- 多次操作特定数据集, 迭代运算
- 搜索引擎 – PageRank
- 计算相似 – Single Source Shortest Path (单源)

性能

- Hadoop : 适合不能全部读入内存；单次读取、类似 ETL (抽取、转换、加载) 操作的任务，比如数据转化、数据整合等时
- Spark : 适合数据不太大内存放得下，重复读取同样数据迭代计算

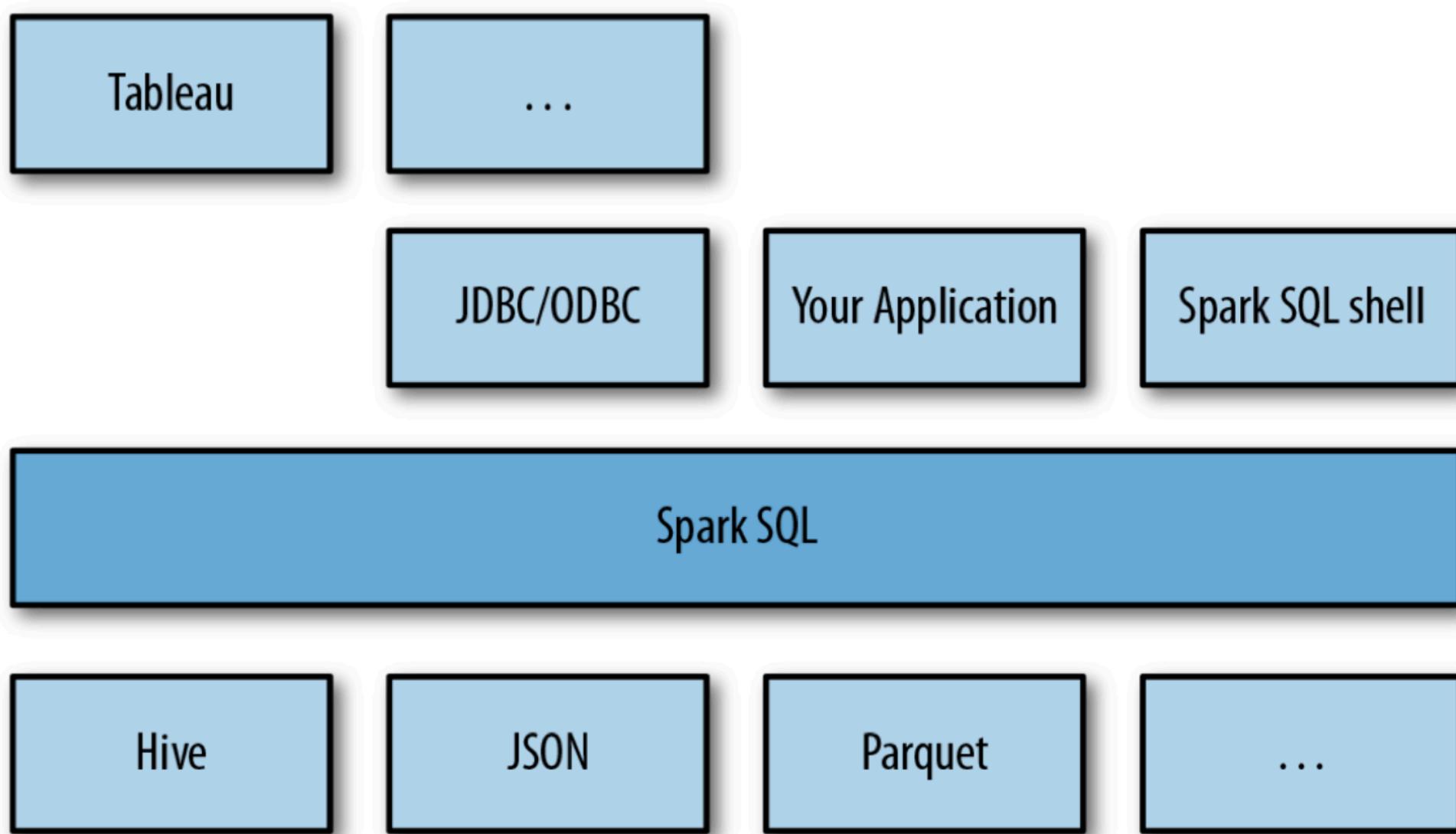
上手

- Hadoop : Java 编写，需要学习语法，有一些工具 (Pig、Hive 等) 简化
- Spark : Scala、Java 和 Python，还支持交互式命令模式

兼容性

- Spark 兼容 Hadoop 数据源

Spark SQL



Using Spark SQL — Steps and Example

Example 9-5. Python SQL imports

```
# Import Spark SQL
from pyspark.sql import HiveContext, Row
```

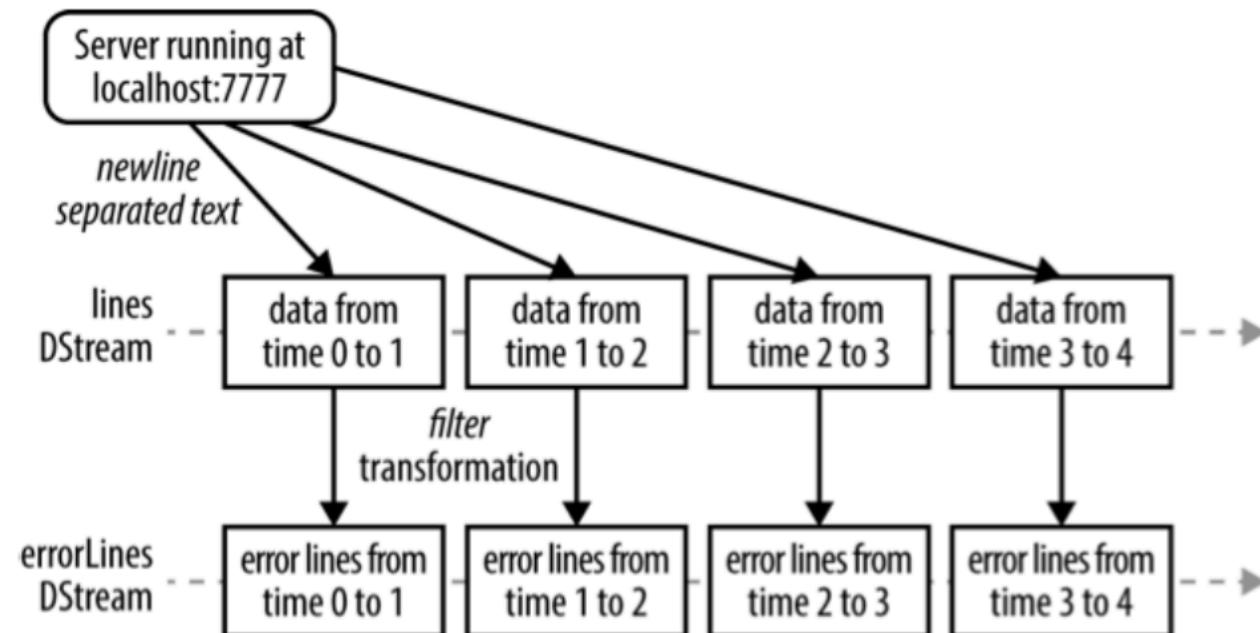
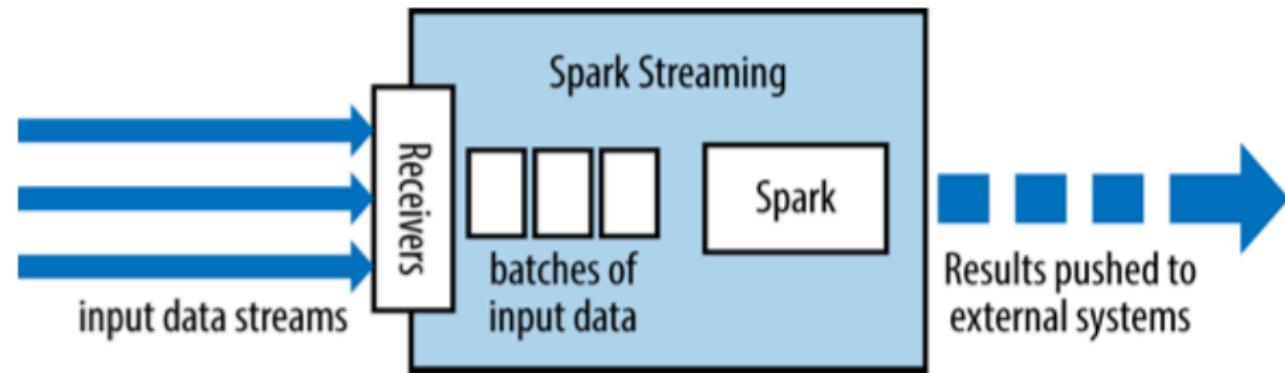
Example 9-8. Constructing a SQL context in Python

```
hiveCtx = HiveContext(sc)
```

Example 9-11. Loading and querying tweets in Python

```
input = hiveCtx.jsonFile(inputFile)
# Register the input schema RDD
input.registerTempTable("tweets")
# Select tweets based on the retweetCount
topTweets = hiveCtx.sql("""SELECT text, retweetCount FROM
tweets ORDER BY retweetCount LIMIT 10""")
```

Spark Streaming architecture

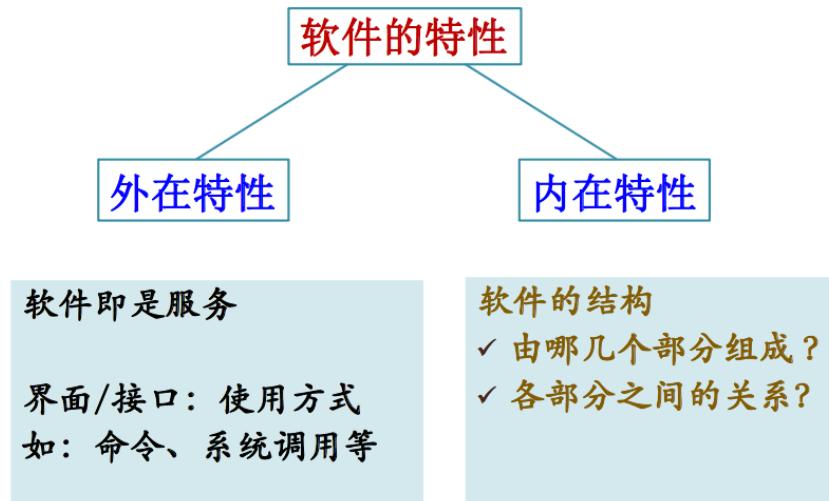


Operation System

CPU

从不同角度认知操作系统

1.作为软件来看的观点



怎样管理资源？

- 跟踪记录资源使用状况
 - 如：哪些资源空闲，好坏与否，被谁使用，使用多长时间等
- 分配和回收资源（**资源分配策略与算法**）
 - 静态分配策略
 - 动态分配策略 ✓
- 提高资源利用率
- 保护
- 协调多个进程对资源请求的冲突

公平竞争
防止非法
使用

2.资源管理的观点

自底向上 → 操作系统 是 **资源的管理者**

硬件资源：

**CPU, 内存, 设备 (I/O设备、磁盘、时钟、网络
接口等)**

软件资源：

磁盘上的文件、信息

从资源管理的角度—五大基本功能

- 进程和线程管理（CPU管理、调度）
进程控制、同步互斥、通信、调度
- 存储管理
分配/回收、地址映射、存储保护、内存扩充
- 文件管理
文件目录、磁盘空间、文件系统布局、存取控制
- 设备管理
设备驱动、分配回收、缓冲技术
- 用户接口
系统命令、编程接口

从不同角度认知操作系统

3. 进程的观点

从操作系统运行的角度动态的观察操作系统

按照这一观点：

- 操作系统是由一些可同时、独立运行的进程和一个对这些进程进行协调的核心组成

进程：完成某一特定功能的程序
是程序的一次执行过程
动态的、有生命的，存在/消亡

4. 虚机器观点

从操作系统内部结构来看：

- ✓ 把操作系统分成若干层 分层结构
- ✓ 每一层完成其特定功能从而构成一个虚机器，并对上一层提供支持
- ✓ 通过逐层功能扩充，最终完成整个操作系统虚机器
- ✓ 而操作系统虚机器向用户提供各种功能，完成用户请求

VMM(虚拟机管理器)

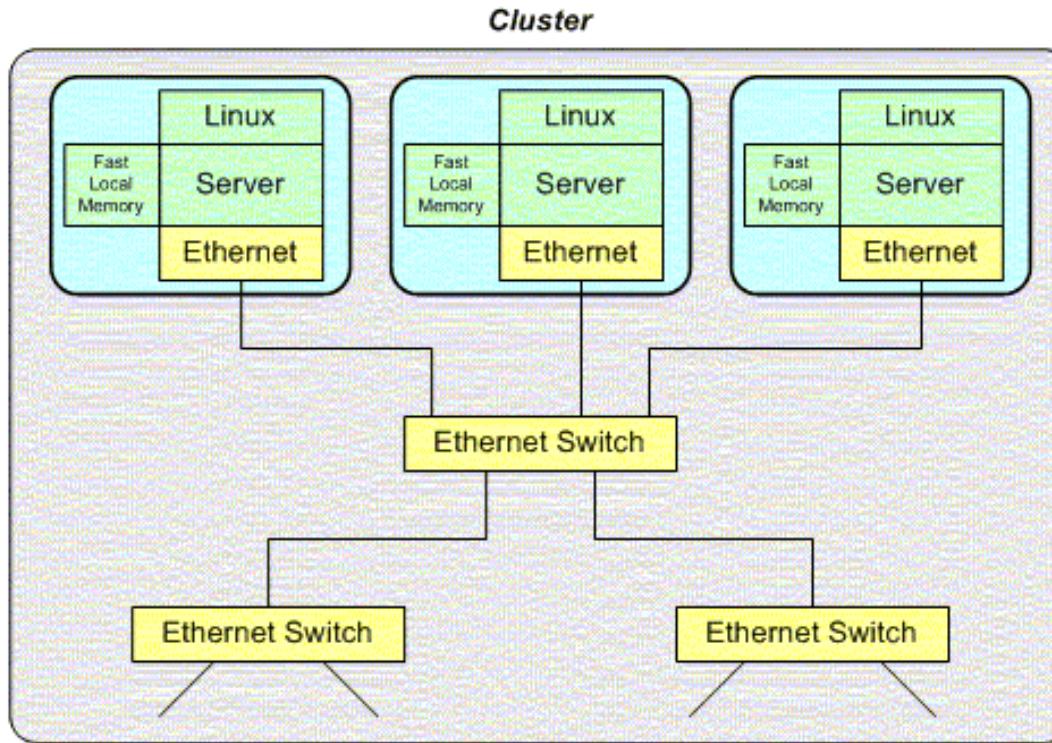
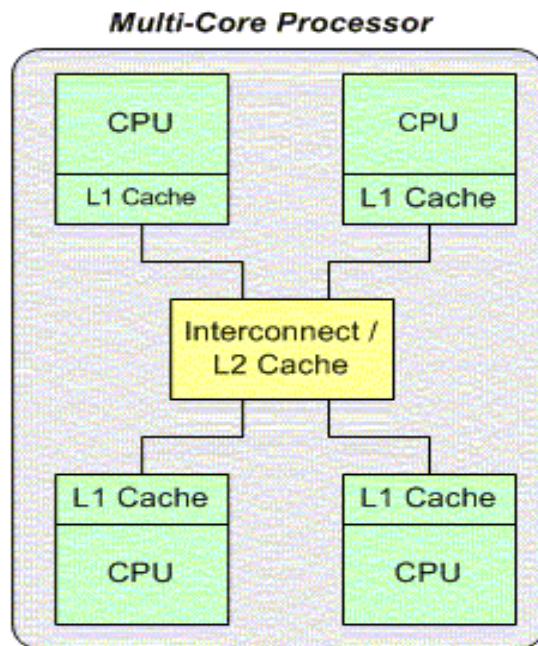


Linux操作系统内核



2.1 中央处理器 (CPU)

- 单处理器
 - 指令流水线
- 多处理器
 - 松耦合
 - 紧耦合



单指令流单数据流(SISD, Single instruction, single data)
单指令流多数据流(SIMD)
多指令流单数据流(MISD)
多指令流多数据流(MIMD)

紧耦合多处理器

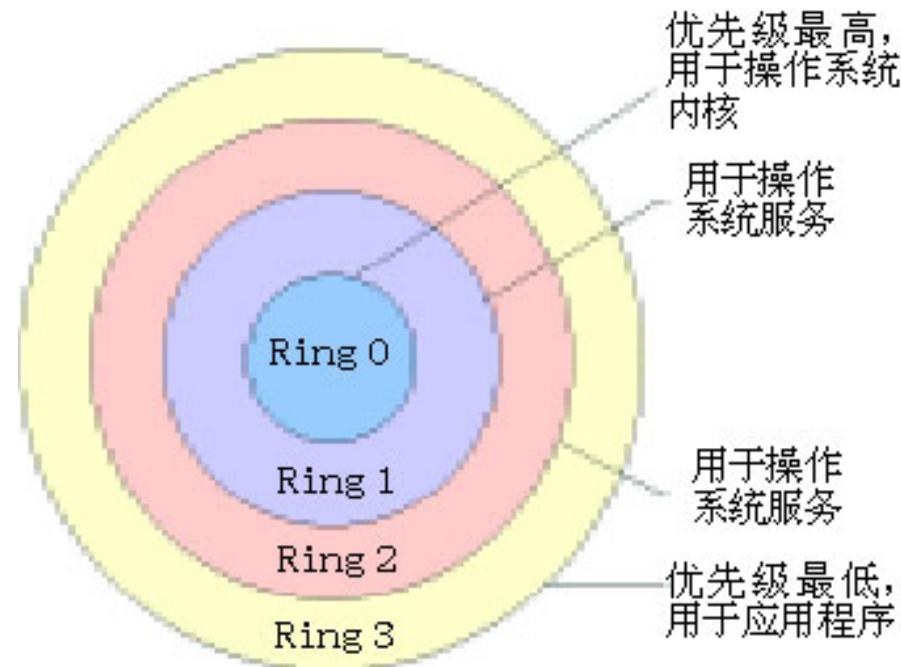
- 主从式
 - 主：操作系统内核
 - 从：其他程序
- 对称式(SMP)
 - 每个处理器是对等的
 - 内核可以运行在任意处理器上
 - 负载均衡

AMD	AMD X2	SMP, 双 CPU
Intel®	Xeon	SMP, 双 CPU 或四 CPU
Intel	Core2 Duo	SMP, 双 CPU
ARM	MPCore	SMP, 最多四 CPU



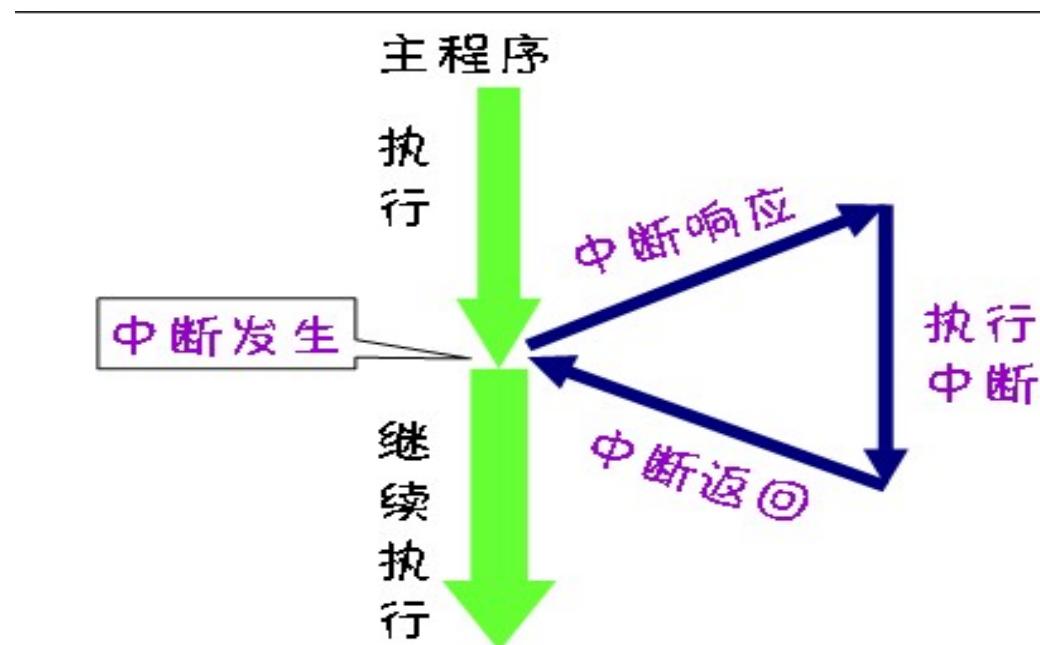
处理器状态

- 处理器怎么知道当前是操作系统还是一般用户程序在运行呢?
 - 程序状态字寄存器
- 处理器状态
- 核心态(Ring 0)
 - 运行内核代码
 - 可执行特权指令
- 用户态→核心态
 - 系统调用
 - 中断

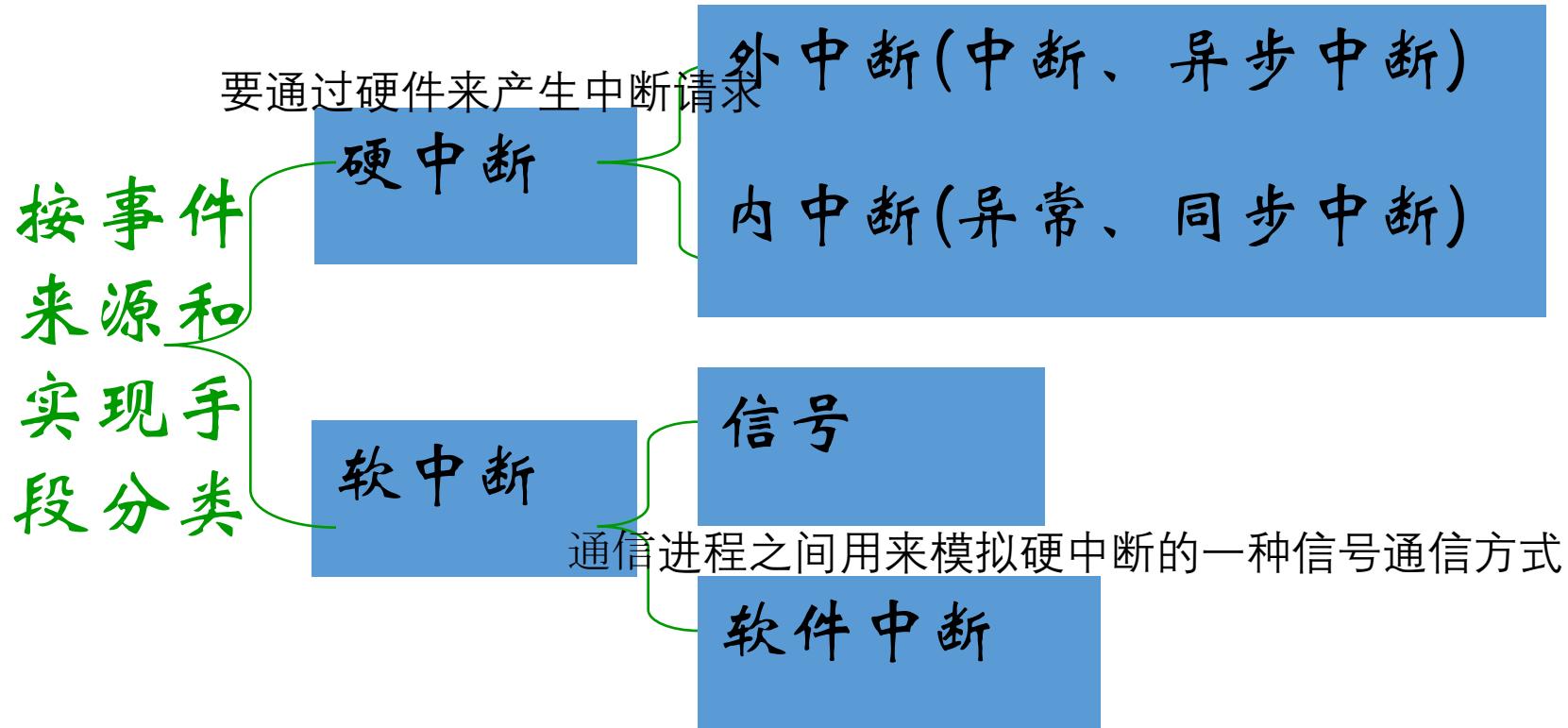


2.2 处理器的中断技术

- 中断是指程序执行过程中，遇到急需处理的事件时，暂时中止CPU上现行程序的运行，转去执行相应的事件处理程序，待处理完成后再返回原程序被中断处或调度其他程序执行的过程。



中断分类 (2)



中断 vs 异常

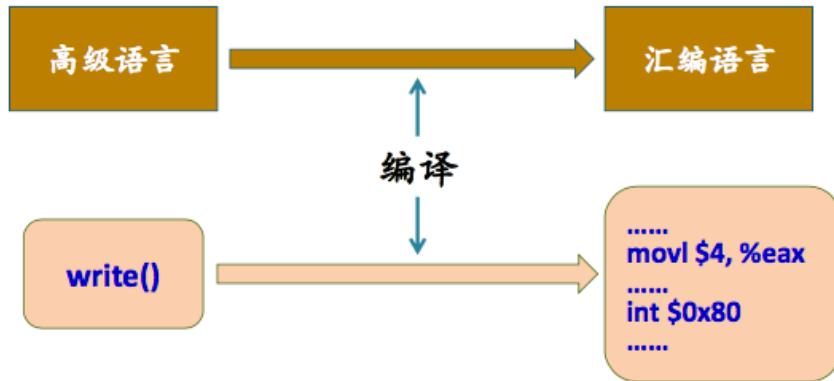
- 中断
 - 现行指令无关的中断信号触发的(异步的)
 - 指令之间才允许中断
 - 中断的发生与CPU处在用户模式或内核模式无关
 - 一般来说，中断处理程序提供的服务不是为当前进程所需的
- 异常
 - 由处理器正在执行现行指令而引起
 - 一条指令执行期间允许响应异常
 - 异常处理程序提供的服务是为当前进程所用
 - 异常包括很多方面，有出错(fault)，也有陷入(trap)等

中断与异常的小结

类别	原因	异步/同步	返回行为
中断 Interrupt	来自I/O设备、其他硬件部件	异步	总是返回到下一条指令
陷入Trap	有意识安排的	同步	返回到下一条指令
故障Fault	可恢复的错误	同步	返回到当前指令
终止Abort	不可恢复的错误	同步	不会返回

- 硬件该做什么事? —— **中断/异常响应**
捕获中断源发出的中断/异常请求, 以一定方式响应, 将处理器控制权交给特定的处理程序
- 软件要做什么事? —— **中断/异常处理程序**
识别中断/异常类型并完成相应的处理

系统调用举例(1/3)



系统调用举例(3/3)

```
1. .section .data
2. output:
3.   .ascii "Hello!\n"
4. output_end:
5.   .equ len, output_end - output

6. .section .text
7. .globl _start
8. _start:
9.   movl $4, %eax
10.  movl $1, %ebx
11.  movl $output, %ecx
12.  movl $len, %edx
13.  int $0x80
14. end:
15.  movl $1, %eax
16.  movl $0, %ebx
17.  int $0x80
```

eax存放系统调用号
引发一次系统调用
!这个系统调用的作用?

系统调用举例(2/3)

```
#include <unistd.h>
int main(){
    char string[7] = {'H','e','l','l','o','!', '\n'};
    write(1, string, 7);
    return 0;
}
```

输出结果: Hello!

高级语
言视角

系统调用的执行过程

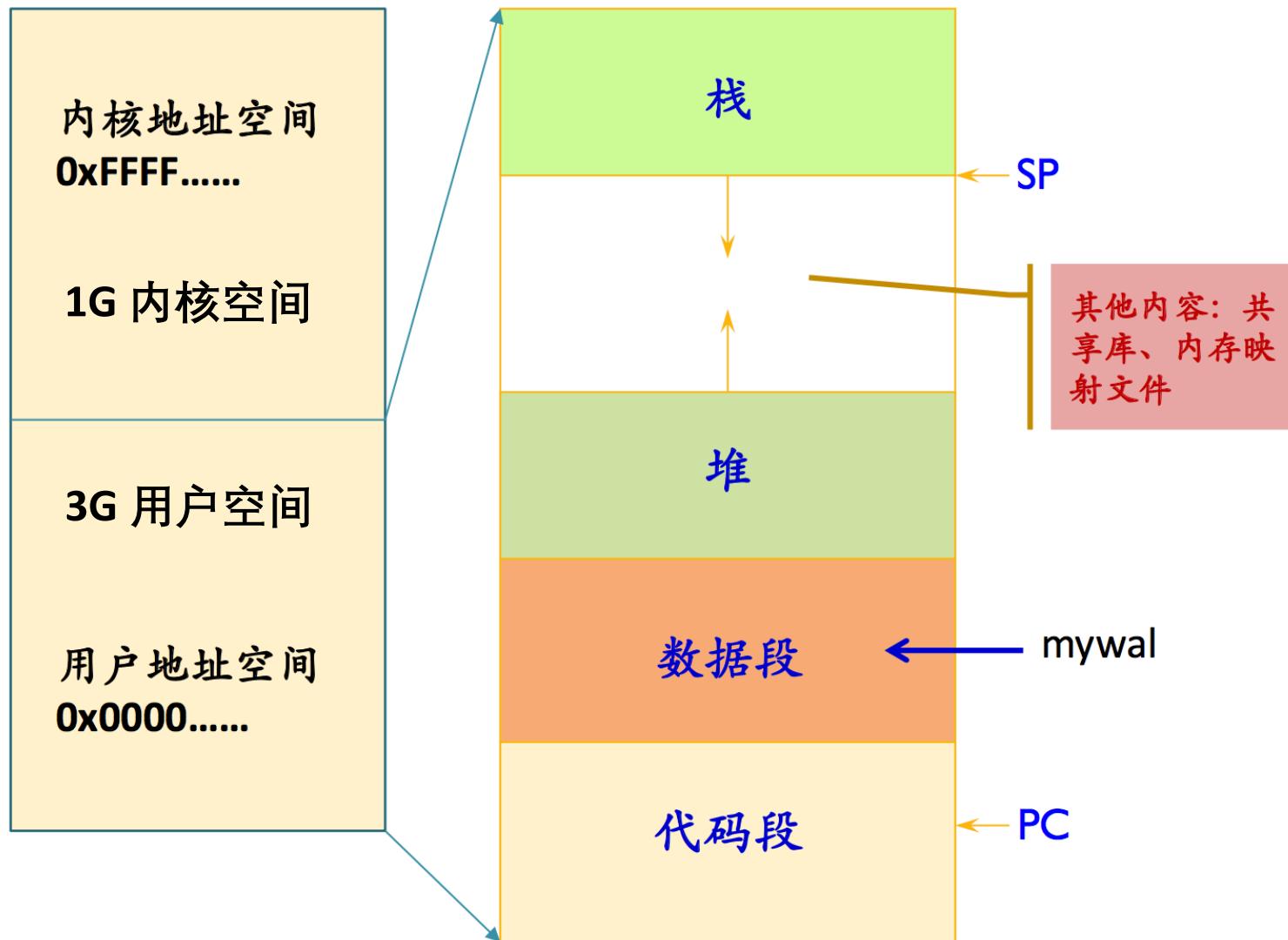
当CPU执行到特殊的陷入指令时:

- **中断/异常机制**: 硬件保护现场; 通过查中断向量表把控制权转给系统调用总入口程序
- **系统调用总入口程序**: 保存现场; 将参数保存在内核堆栈里; 通过查系统调用表把控制权转给相应的系统调用处理例程或内核函数
- **执行系统调用例程**
- **恢复现场, 返回用户程序**

Operation System

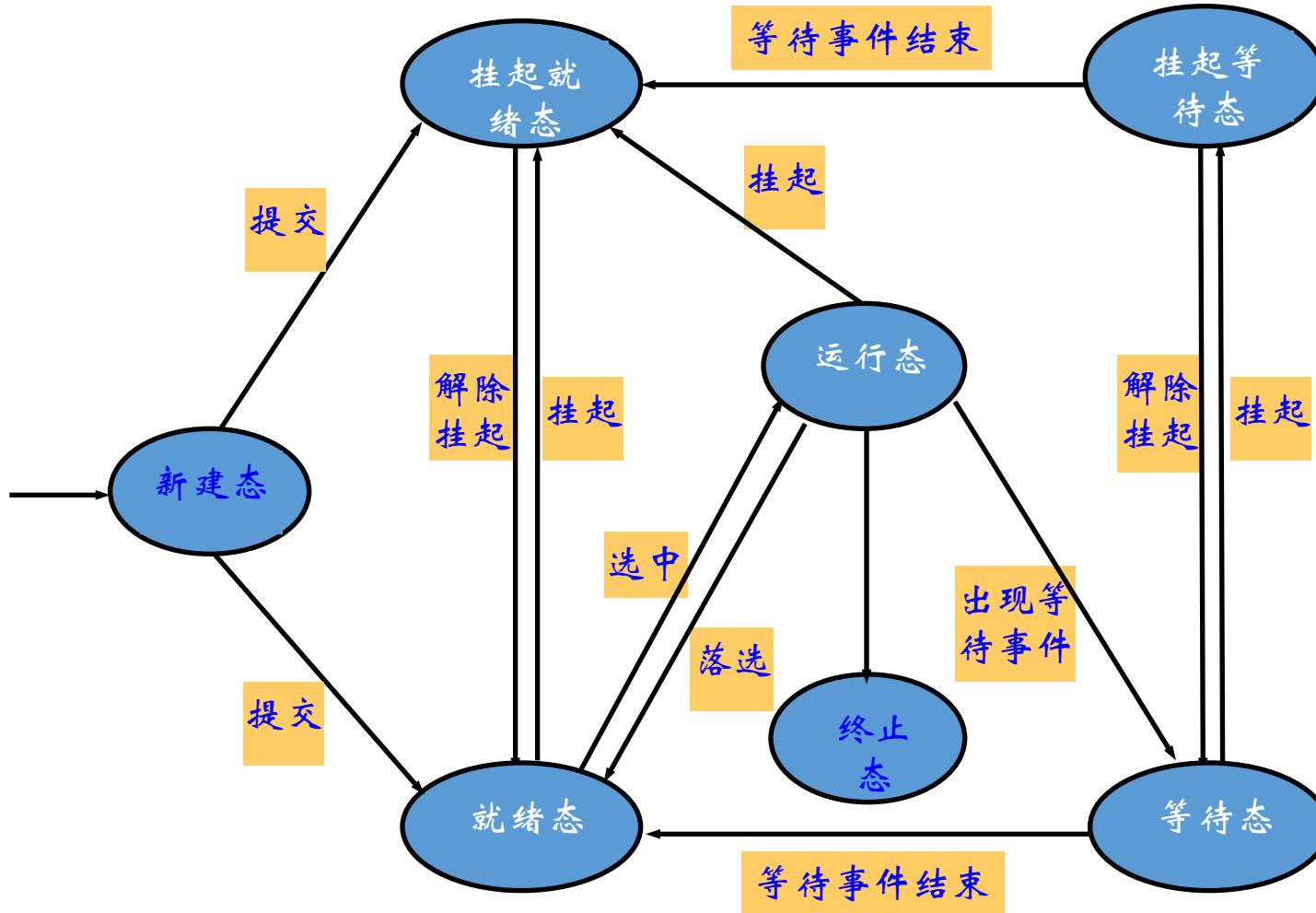
Process

操作系统给每个进程都分配了一个地址空间



进程的挂起

- 进程可“交换”到外存，解决资源不足



进程的定义和性质

- 进程是可并发执行的程序在某个数据集合上的一次计算活动，也是操作系统进行资源分配和保护的基本单位。
- 进程是一个既能用来共享资源，又能描述程序并发执行过程的一个基本单位。

ps

ps -A

显示所有进程信息

ps -u root

显示指定用户信息

ps -ef

显示所有进程信息，连同命令行

ps -ef|grep ssh ps 与grep组合，查找特定进程.

ps aux

列出目前所有的正在内存当中的程序

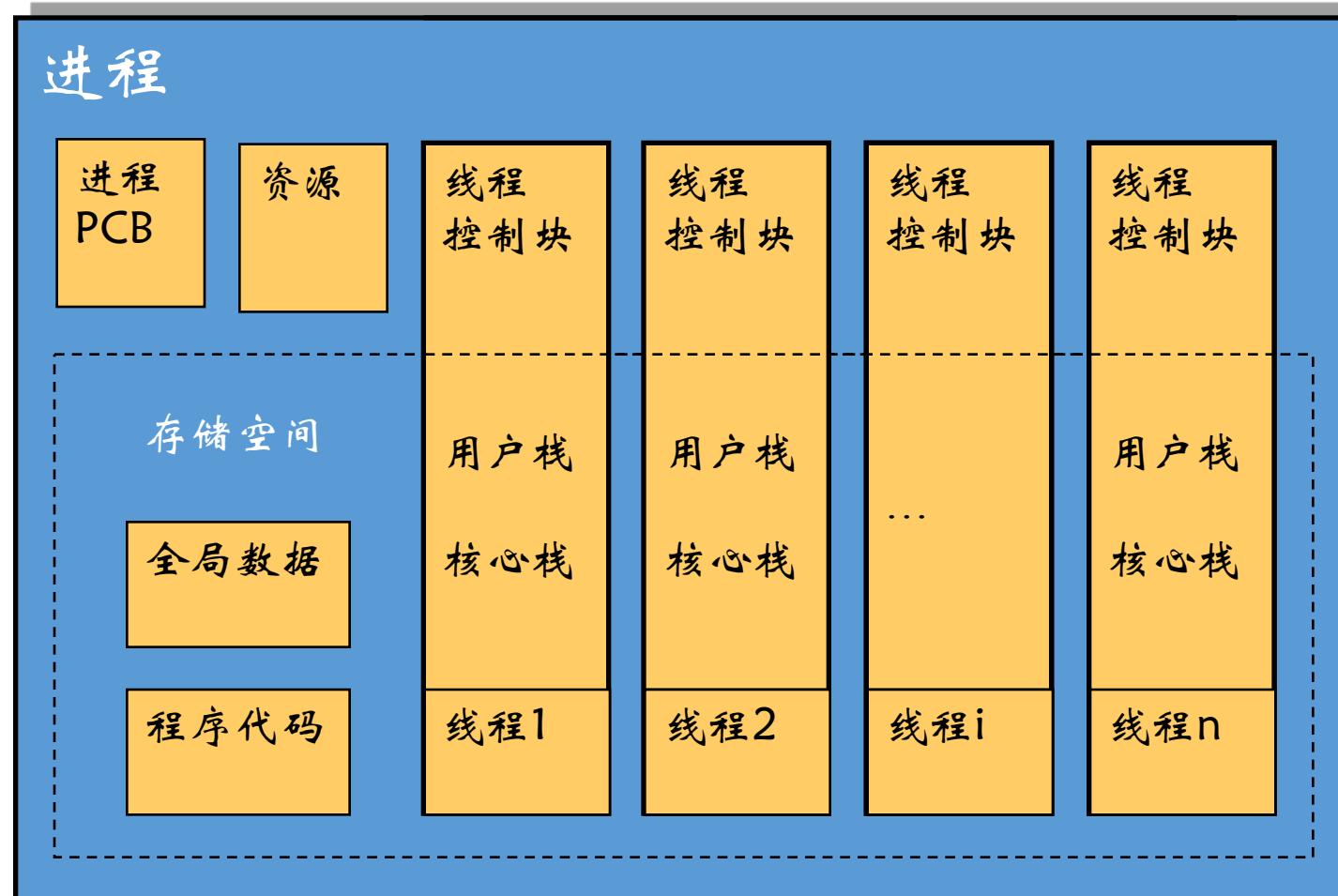
ps aux |more

可以用|管道和more连接起来分页查看

引入线程的动机

- 引入进程的动机
 - 使多个程序可以并发执行，以改善资源使用率和提高系统效率
- 引入线程的动机
 - 减少进程并发执行时所付出的时空开销，使得并发粒度更细、并发性更好
- 解决思路
 - 分离进程的两项功能：“独立分配资源”与“被调度分派执行”
 - 进程：资源分配单位，无需频繁切换
 - 线程：调度单位，体量小，频繁切换

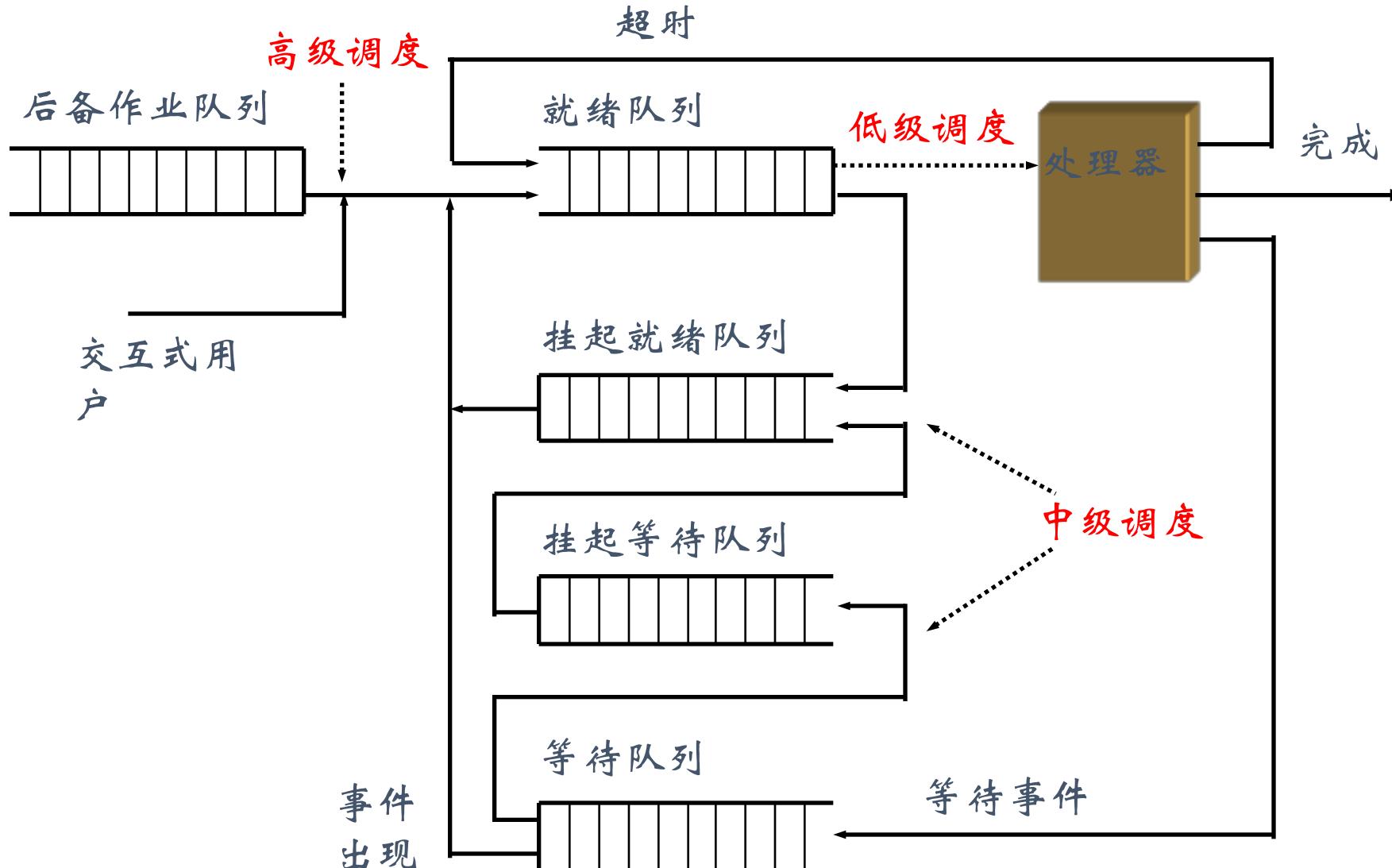
多线程环境中的进程与线程



处理其调度的层次

- 高级调度
 - 多道批处理系统，选择作业进入主存
 - 分时系统通常不需要
- 中级调度
 - 外存与内存中的进程对换
 - 提高主存的利用率
- 低级调度
 - 就绪→运行
 - 操作系统必备

处理器的三级调度模型



选择调度算法的原则 (1)

- 资源(CPU)利用率
 - CPU利用率=CPU有效工作时间/CPU总的运行时间
 - CPU总的运行时间=CPU有效工作时间+CPU空闲等待时间
- 吞吐率
 - 单位时间内CPU处理作业的数目
- 公平性
 - 避免饥饿

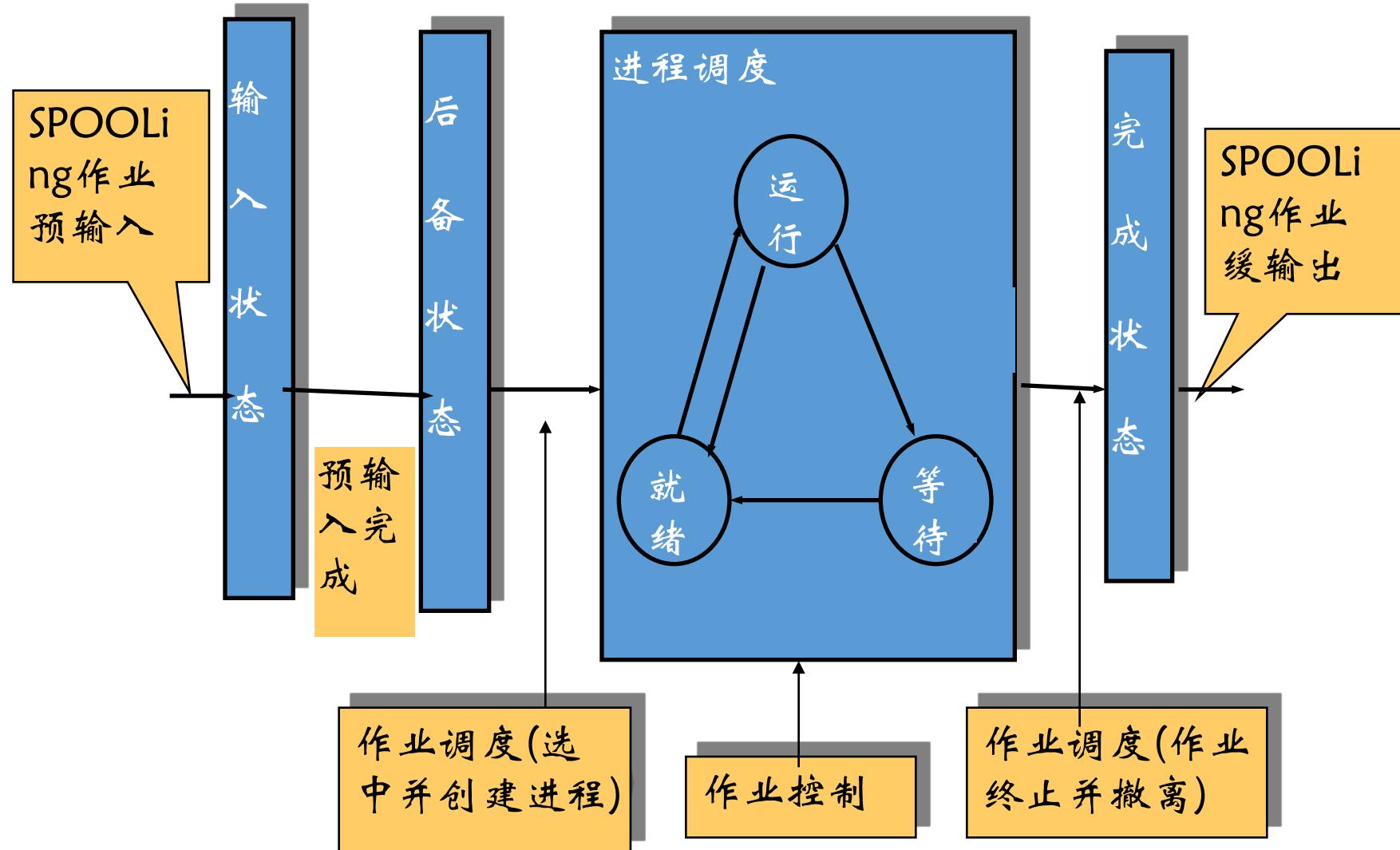
选择调度算法的原则 (2)

- 响应时间
 - 交互式进程从提交一个请求(命令)到接收到响应之间的时间间隔称响应时间
 - 实时系统、分时系统的重要指标
- 周转时间
 - 批处理用户从作业提交给系统开始，到作业完成为止的时间间隔称作业周转时间
 - 批处理系统的重要指标

平均作业周转时间

- 作业周转时间
 - 如果作业*i*提交给系统的时刻是 t_s , 完成时刻是 t_f , 该作业的周转时间为 : $t_i = t_f - t_s$
- 平均作业周转时间 : $T = (\sum t_i) / n$
- 如果作业*i*的周转时间为 t_i , 所需运行时间为 t_k , 则称 $w_i = t_i / t_k$ 为该作业的带权周转时间
- 平均作业带权周转时间 $W = (\sum w_i) / n$

进程调度 vs 作业调度



常用的调度算法

- 先来先服务(FCFS-First Come First Serve)
- 最短作业优先(SJF-Shortest Job First)
- 最短剩余时间优先(SRT-Shortest Remaining Time Next)
- 最高相应比优先(HRRN-Highest Response Ratio Next)

Operation System

Spark

Job Scheduling

- [Overview](#)
- [Scheduling Across Applications](#)
 - [Dynamic Resource Allocation](#)
 - [Configuration and Setup](#)
 - [Resource Allocation Policy](#)
 - [Request Policy](#)
 - [Remove Policy](#)
 - [Graceful Decommission of Executors](#)
 - [Scheduling Within an Application](#)
 - [Fair Scheduler Pools](#)
 - [Default Behavior of Pools](#)
 - [Configuring Pool Properties](#)

Overview

Spark has several facilities for scheduling resources between computations. First, recall that, as described in the [cluster mode overview](#), each Spark application (instance of `SparkContext`) runs an independent set of executor processes. The cluster managers that Spark runs on provide facilities for [scheduling across applications](#). Second, *within* each Spark application, multiple “jobs” (Spark actions) may be running concurrently if they were submitted by different threads. This is common if your application is serving requests over the network. Spark includes a [fair scheduler](#) to schedule resources within each `SparkContext`.

Scheduling Across Applications

When running on a cluster, each Spark application gets an independent set of executor JVMs that only run tasks and store data for that application. If multiple users need to share your cluster, there are different options to manage allocation, depending on the cluster manager.

The simplest option, available on all cluster managers, is *static partitioning* of resources. With this approach, each application is given a maximum amount of resources it can use, and holds onto them for its whole duration. This is the approach used in Spark’s [standalone](#) and [YARN](#) modes, as well as the [coarse-grained Mesos mode](#). Resource allocation can be configured as follows, based on the cluster type:

- **Standalone mode:** By default, applications submitted to the standalone mode cluster will run in FIFO (first-in-first-out) order, and each application will try to use all available nodes. You can limit the number of nodes an application uses by setting the `spark.cores.max` configuration property in it, or change the default for applications that don’t set this setting through `spark.deploy.defaultCores`. Finally, in addition to controlling cores, each application’s `spark.executor.memory` setting controls its memory use.
- **Mesos:** To use static partitioning on Mesos, set the `spark.mesos.coarse` configuration property to `true`, and optionally set `spark.cores.max` to limit each application’s resource share as in the standalone mode. You should also set `spark.executor.memory` to control the executor memory.
- **YARN:** The `--num-executors` option to the Spark YARN client controls how many executors it will allocate on the cluster (`spark.executor.instances` as configuration property), while `--executor-memory` (`spark.executor.memory` configuration property) and `--executor-cores` (`spark.executor.cores` configuration property) control the resources per executor. For more information, see the [YARN Spark Properties](#).

应用间调度 (1)

- 调度策略1: 资源静态分区

- 资源静态分区是指整个集群的资源被预先划分为多个partitions，资源分配时的最小粒度是一个静态的partition。
- 根据应用对资源的申请需求为其分配静态的partition(s)是Spark支持的最简单的调度策略。
- 若Spark集群配置了static partitioning的调度策略，则它对提交的多个应用间默认采用FIFO顺序进行调度，
- 每个获得执行机会的应用在运行期间可占用整个集群的资源，这样做明显不友好，所以应用提交者通常需要通过设置spark.cores.max来控制其占用的core/memory资源。

应用间调度 (2)

- 调度策略2: 动态共享CPU cores
 - 若Spark集群采用Mesos模式, 可以采用该策略 ;
 - 每个应用仍拥有各自独立的cores/memory, 但当应用申请资源后并未使用时 (即分配给应用的资源当前闲置) , 其它应用的计算任务可能会被调度器分配到这些闲置资源上。
 - 当提交给集群的应用有很多是非活跃应用时 (即它们并非时刻占用集群资源) , 这种调度策略能很大程度上提升集群资源利用效率。
 - 风险是 : 若某个应用从非活跃状态转变为活跃状态时, 且它提交时申请的资源当前恰好被调度给其它应用, 则它无法立即获得执行的机会。

应用间调度 (3)

- 动态资源申请
 - 它允许根据应用的workload动态调整其所需的集群资源。
 - 若应用暂时不需要它之前申请的资源，则它可以先归还给集群，
 - 当它需要时，可以重新向集群申请。当Spark集群被多个应用共享时，这种按需分配的策略显然是非常有优势的。

应用内调度 (1)

在应用内部，每个action (e.g. save, collect) 以及计算这个action结果所需要的一系列tasks被统称为一个"job"

- FIFO的调度策略
 - 每个job被分解为不同的stages;
 - 当多个job各自的stage所在的线程同时申请资源时，第1个job的stage优先获得资源;
 - 如果job queue头部的job恰好是需要最长执行时间的job时，后面所有的job均得不到执行的机会，这样会导致某些job(s)饿死的情况。

应用内调度 (2)

- "fair sharing"
 - Spark对不同jobs的stages提交的tasks采用Round Robin的调度方式，如此，所有的jobs均得到公平执行的机会。因此，即使某些short-time jobs本身的提交时间在long jobs之后，它也能获得被执行的机会，从而达到可预期的响应时间
 - 要启用fair sharing调度策略，需要在spark配置文件中将spark.scheduler.mode设置为FAIR。
 - fair sharing调度也支持把不同的jobs聚合到一个pool，不同的pools赋予不同的执行优先级。
 - 这是FIFO和fair sharing两种策略的折衷策略，既能保证jobs之间的优先级，也能保证同一优先级的jobs均能得到公平执行的机会

How to set?

To enable the fair scheduler, simply set the `spark.scheduler.mode` property to `FAIR` when configuring a `SparkContext`:

```
val conf = new SparkConf().setMaster(...).setAppName(...)  
conf.set("spark.scheduler.mode", "FAIR")  
val sc = new SparkContext(conf)
```

Without any intervention, newly submitted jobs go into a *default pool*, but jobs' pools can be set by adding the `spark.scheduler.pool` “local property” to the `SparkContext` in the thread that's submitting them. This is done as follows:

```
// Assuming sc is your SparkContext variable  
sc.setLocalProperty("spark.scheduler.pool", "pool1")
```

After setting this local property, *all* jobs submitted within this thread (by calls in this thread to `RDD.save`, `count`, `collect`, etc) will use this pool name. The setting is per-thread to make it easy to have a thread run multiple jobs on behalf of the same user. If you'd like to clear the pool that a thread is associated with, simply call:

```
sc.setLocalProperty("spark.scheduler.pool", null)
```

OS

进程同步

并发带来的问题

- 数据一致性问题

- 丢失修改

- 不可重复读

- 读脏数据

思考：数据库并发带来的问题？

死锁与饥饿

- 资源竞争引起死锁与饥饿
- 死锁， deadlock
 - 一组进程都获得部分资源， 又要申请彼此占有的资源→死锁
 - 死锁条件：
 - 互斥条件、不可剥夺、请求和保持、循环等待
- 饥饿， starvation
 - 进程一直抢占不到资源
 - SJF 调度方法 （最短作业优先算法）。

并发进程的关系(1)

- 完全无关

Bernstein 条件

- 条件 $R(P_1) \cap W(P_2) \cup R(P_2) \cap W(P_1) \cup W(P_1) \cap W(P_2) = \emptyset$
- 并发进程执行与时间无关，不会产生任何错误

$R(p_i) = \{a_1, a_2, \dots, a_n\}$, 表示程序 p_i 在执行期间引用的变量集；

$W(p_i) = \{b_1, b_2, \dots, b_m\}$, 表示程序 p_i 在执行期间改变的变量集。

- 竞争

- 彼此不知道对方的存在
- 竞争独占性资源
- 需要互斥
- 问题：死锁、饥饿

并发进程的关系(2)

- 共享合作
 - 进程不知道彼此的存在
 - 有共享数据：变量、文件、数据库等
 - 互斥需求
 - 问题：数据一致性问题、死锁、饥饿等
- 通信合作
 - 进程知道彼此的存在
 - 通过通信协作
 - 同步需求
 - 死锁、饥饿

进程的同步

进程同步：synchronization

指系统中多个进程中发生的事件存在某种时序关系，需要相互合作，共同完成一项任务

具体地说，一个进程运行到某一点时，要求另一伙伴进程为它提供消息，在未获得消息之前，该进程进入阻塞态，获得消息后被唤醒进入就绪态

信号量与PV操作

- 1965年E.W.Dijkstra提出了新的同步工具--信号量和P、V操作
- 信号量(semaphore)
 - 旗语
 - 特殊变量，交互进程在关键点上一直等待直到接收到特殊变量值
- P操作原语和V操作原语
 - 对信号量进行特殊操作
 - P：检测，V：增量
 - 对信号量可以实施的操作：初始化、P和V(P、V分别是荷兰语的test(proberen)和increment(verhogen))



```
struc semaphore  
{  
    int count;  
    queueType queue;  
}
```

P、V操作定义

P(s)

{

s.count --;

if (s.count < 0)

{

该进程状态置为阻塞状态；

将该进程插入相应的等待队列s.queue末尾；

重新调度；

}

}

down, semWait

V(s)

{

struc semaphore

{

int count;

queueType queue;

}

s.count ++;

if (s.count <= 0)

{

唤醒相应等待队列s.queue中等待的一个进程；

改变其状态为就绪态，并将其插入就绪队列；

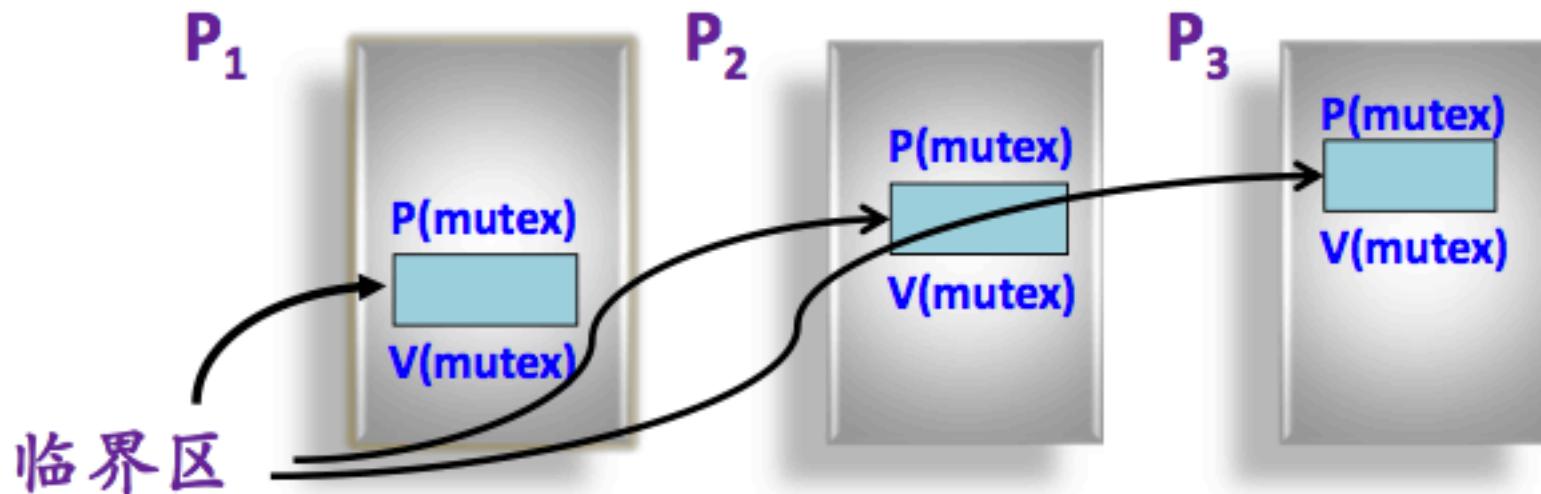
}

up, semSignal

}

用PV操作解决进程间互斥问题

- 分析并发进程的关键活动，划定临界区
- 设置信号量 mutex，初值为1
- 在临界区前实施 P(mutex)
- 在临界区之后实施 V(mutex)



管程的定义

- 是一个特殊的模块
- 有一个名字
- 由关于共享资源的数据结构及在其上操作的一组过程组成

□ 进程与管程

进程只能通过调用管程中的过程来间接地访问管程中的数据结构

monitor example

integer i ;
condition c ;

变量

procedure $producer()$;

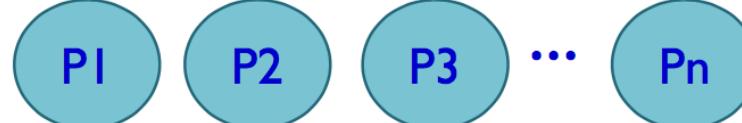
过程

end;

procedure $consumer()$;

end;

end monitor;



管程的定义和特性

- 管程的定义
 - 由局部与自己的若干公共变量和所有访问这些公共变量的过程（临界区）所组成的软件模块
- 管程的特性
 - 共享性
 - 对外接口可以被多个进程共享
 - 安全性
 - 管程的局部变量只能由管程自己访问，管程也不访问任何外部变量
 - 互斥性
 - 一次只让一个进程（线程）进入管程

管道通信机制

- 管道(pipeline)是连接读写进程的一个特殊文件, 允许进程按先进先出方式传送数据, 也能使进程同步执行操作
- 发送进程以字符流形式把大量数据送入管道, 接收进程从管道中接收数据, 所以叫管道通信
- 管道的实质是一个共享文件, 基本上可借助于文件系统的机制实现, 包括(管道)文件的创建、打开、关闭和读写。

死锁定义

- 如果在一个进程集合中的每个进程都在等待只能由该集合中的其他一个进程才能引发的事件，则称一组进程或系统此时发生死锁。
- 例如， n 个进程 P_1, P_2, \dots, P_n ， P_i 因为申请不到资源 R_j 而处于等待状态，而 R_j 又被 P_{i+1} 占有， P_n 欲申请的资源被 P_1 占有，此时这 n 个进程的等待状态永远不能结束，则说这 n 个进程处于死锁状态。

形成死锁的四个条件

- 四个条件（1971年Coffman总结）
 - 互斥条件：进程互斥使用资源
 - 请求与保持条件：申请新资源时不释放已占有资源
 - 不可剥夺条件：一个进程不能抢夺其他进程占有的资源
 - 循环等待条件：存在一组进程循环等待资源
- 前三个为必要条件、非充分条件
- 前三个+第四个→死锁

死锁解决办法

- 死锁防止(Deadlock Prevention)
 - 破坏产生死锁的必要条件, 防止死锁发生
 - 强制规则 → 降低进程的并发度
- 死锁避免(Deadlock Avoidance)
 - 允许前三个条件, 避免第四个条件
 - 并发度高
- 死锁检测和恢复(Deadlock detection & recovery)

GPU

How to use GPU in Matlab?