Lecture 1

presentation

DAD – Distributed Applications Development

Cristian Toma

D.I.C.E/D.E.I.C – Department of Economic Informatics & Cybernetics
www.dice.ase.ro

Cristian Toma – Business Card

Agenda for Lecture 1

Distributed Systems and Distributed Applications Development

Distributed Systems

It’s not just about the programming, but providing smart solutions

DAD Sections & References

What about the DAD as it is @ Harvard/MIT?

Could you provide a solution for finding out the biggest mark in the class?

Do we have unicast, multicast, broadcast messages or client-server, P2P / hybrid paradigms?...GREAT...Please upload the solution in Java / C# / C/C++ / Python / Ruby till next week 23:50 in e-learning platform – SAKAI...

1.3 Distributed Systems

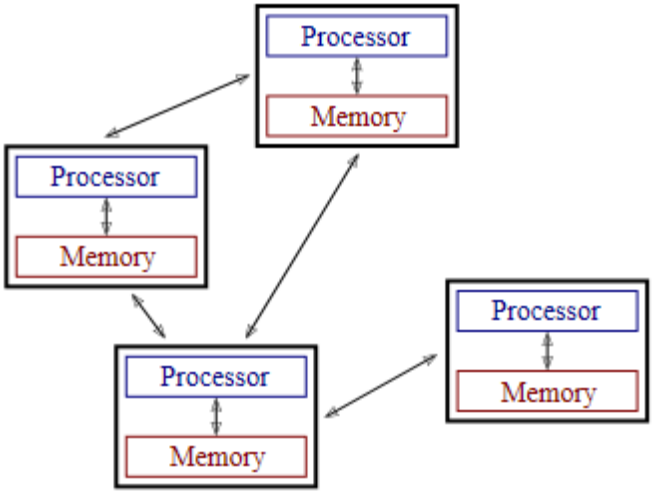
What is a Distributed System?

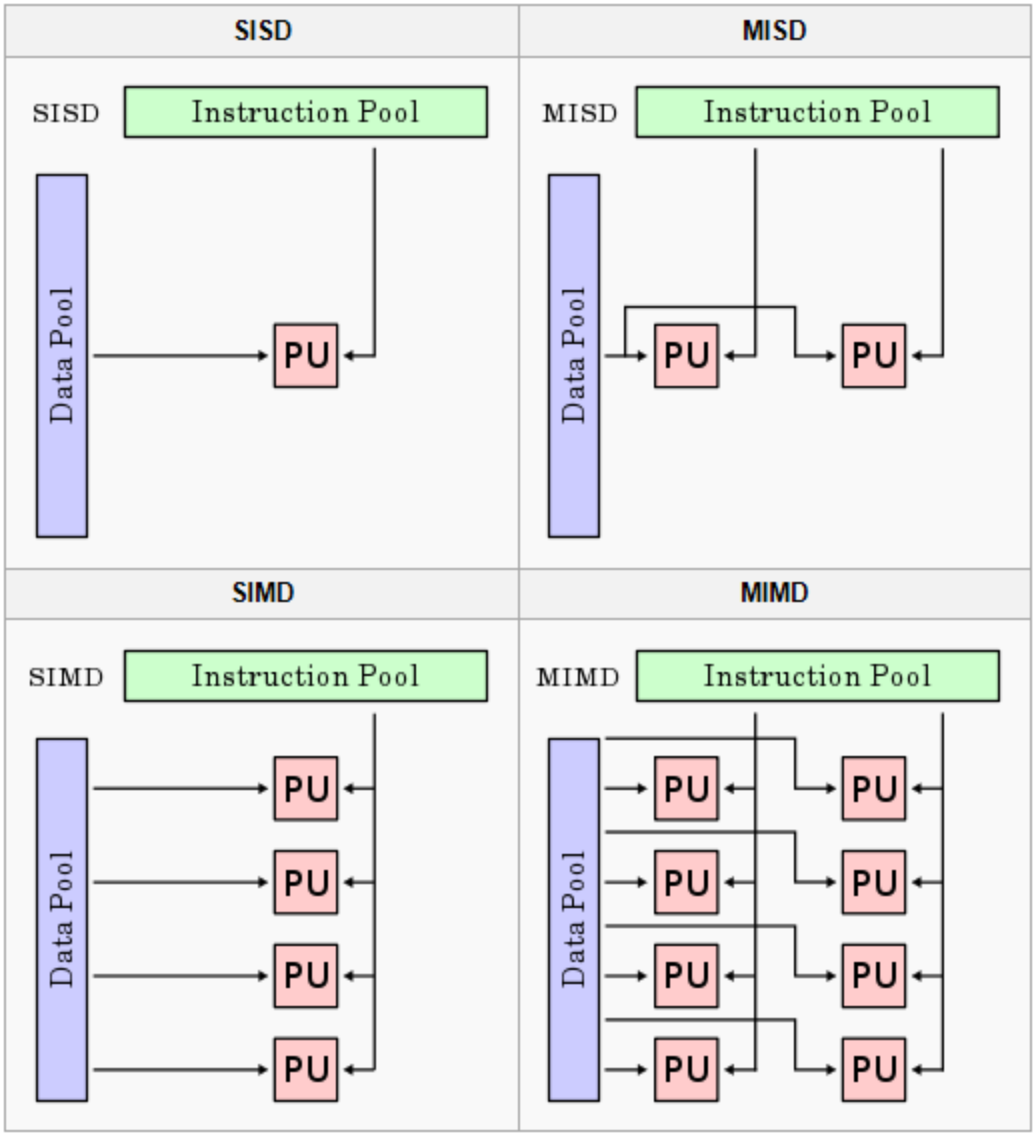
A distributed system is one in which components located into computers connected in network, communicate and coordinate their actions only by passing messages (sometimes, in addition, by migrating processes), in order to resolve a set of problems.

A distributed system consists of a collection of autonomous computers linked by a computer network and equipped with distributed system software. This software enables computers to coordinate their activities and to share the re- sources of the system hardware, software, and data.

Flynn Taxonomy
Parallel vs. Distributed Systems

Parallel vs. Distributed Computing / Algorithms





Where is the picture for:
Distributed System and ***Parallel System***?

http://en.wikipedia.org/wiki/Distributed_computing

http://en.wikipedia.org/wiki/Flynn's_taxonomy

1.3 Distributed Systems

How to characterize a distributed system?

- concurrency of components
- lack of global clock
- independent failures of components

Leslie Lamport :-)

You know you have a distributed system when the crash of a computer you've never heard of stops you from getting any work done!

Prime motivation = to share the resources

1.3 Distributed Systems

What are the challenges?

- heterogeneity of their components
- openness
- security
- scalability – the ability to work well when the load or the number of users increases
- failure handling
- concurrency of components
- transparency
- providing quality of service

1.3 Distributed Systems

What are the resources?

- Hardware
  - Not every single resource is for sharing
- Data
  - Databases
  - Proprietary software
  - Software production
  - Collaboration

Sharing Resources

• Different resources are handled in different ways, there are however some generic requirements:

– Namespace for identification
– Name translation to network address
– Synchronization of multiple access

1.3 Distributed Systems

Concept of Distributed Architecture

A distributed system can be demonstrated by the client-server architecture, which forms the base for multi-tier architectures; alternatives are the broker architecture such as CORBA, and the Service-Oriented Architecture (SOA). In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers.

There are several technology frameworks to support distributed architectures, including Java EE / Java-Kotlin Reactive Micro-services, .NET, CORBA, Scala Akka, Globus Toolkit Grid services, etc..

Middleware is an infrastructure that appropriately supports the development and execution of distributed applications.

1.3 Distributed Systems

Basis of Distributed Architecture

The basis of a distributed architecture is its transparency, reliability, and availability.

The following table lists the different forms of transparency in a distributed system

Transparency	Description
Access	Hides the way in which resources are accessed and the differences in data platform.
Location	Hides where resources are located.
Technology	Hides different technologies such as programming language and OS from user.
Migration / Relocation	Hide resources that may be moved to another location which are in use.
Replication	Hide resources that may be copied at several location.
Concurrency	Hide resources that may be shared with other users.
Failure	Hides failure and recovery of resources from user.
Persistence	Hides whether a resource (software) is in memory or disk.

1.3 Distributed Systems

Distributed Systems Advantages

It has following advantages:

- Resource sharing − Sharing of hardware and software resources.
- Openness − Flexibility of using hardware and software of different vendors.
- Concurrency − Concurrent processing to enhance performance.
- Scalability − Increased throughput by adding new resources.
- Fault tolerance − The ability to continue in operation after a fault has occurred.

Distributed Systems Disadvantages

Its disadvantages are:

- Complexity − They are more complex than centralized systems.
- Security − More susceptible to external attack.
- Manageability − More effort required for system management.
- Unpredictability − Unpredictable responses depending on the system organization and network load.

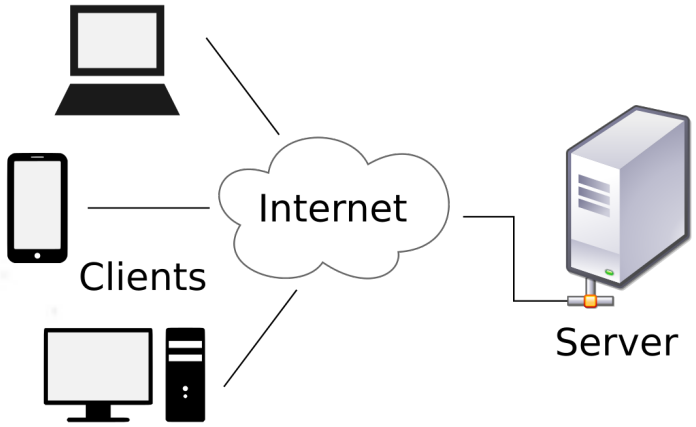
1.3 Distributed Systems

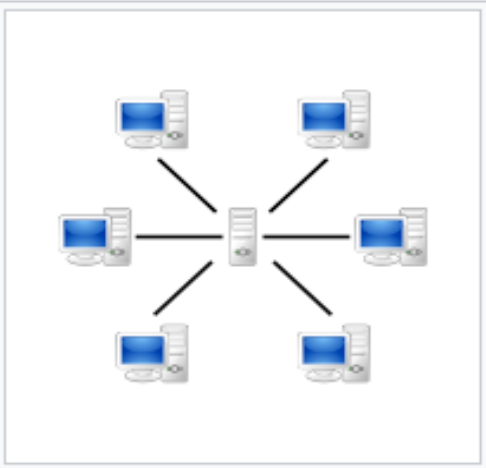
Client-Server Architecture

The client-server architecture is the most common distributed system architecture which decomposes the system into two major subsystems or logical processes −

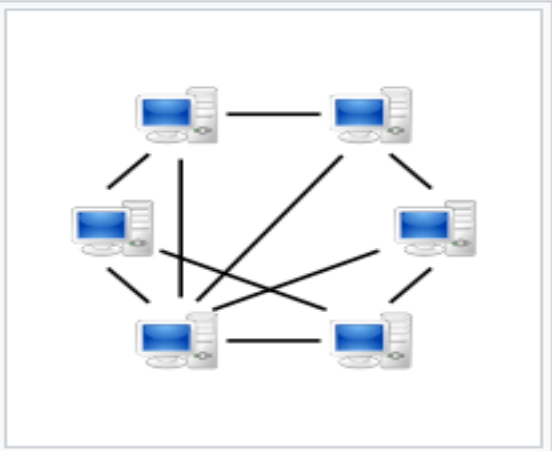
- Client − This is the first process that issues a request to the second process i.e. the server.
- Server − This is the second process that receives the request, carries it out, and sends a reply to the client.

In this architecture, the application is modelled as a set of services those are provided by servers and a set of clients that use these services. The servers need not to know about clients, but the clients must know the identity of servers.





A network based on the **client-server model**, where individual *clients* request services and resources from centralized servers



A **peer-to-peer (P2P) network** in which interconnected nodes ("peers") share resources amongst each other without the use of a centralized administrative system

1.3 Distributed Systems

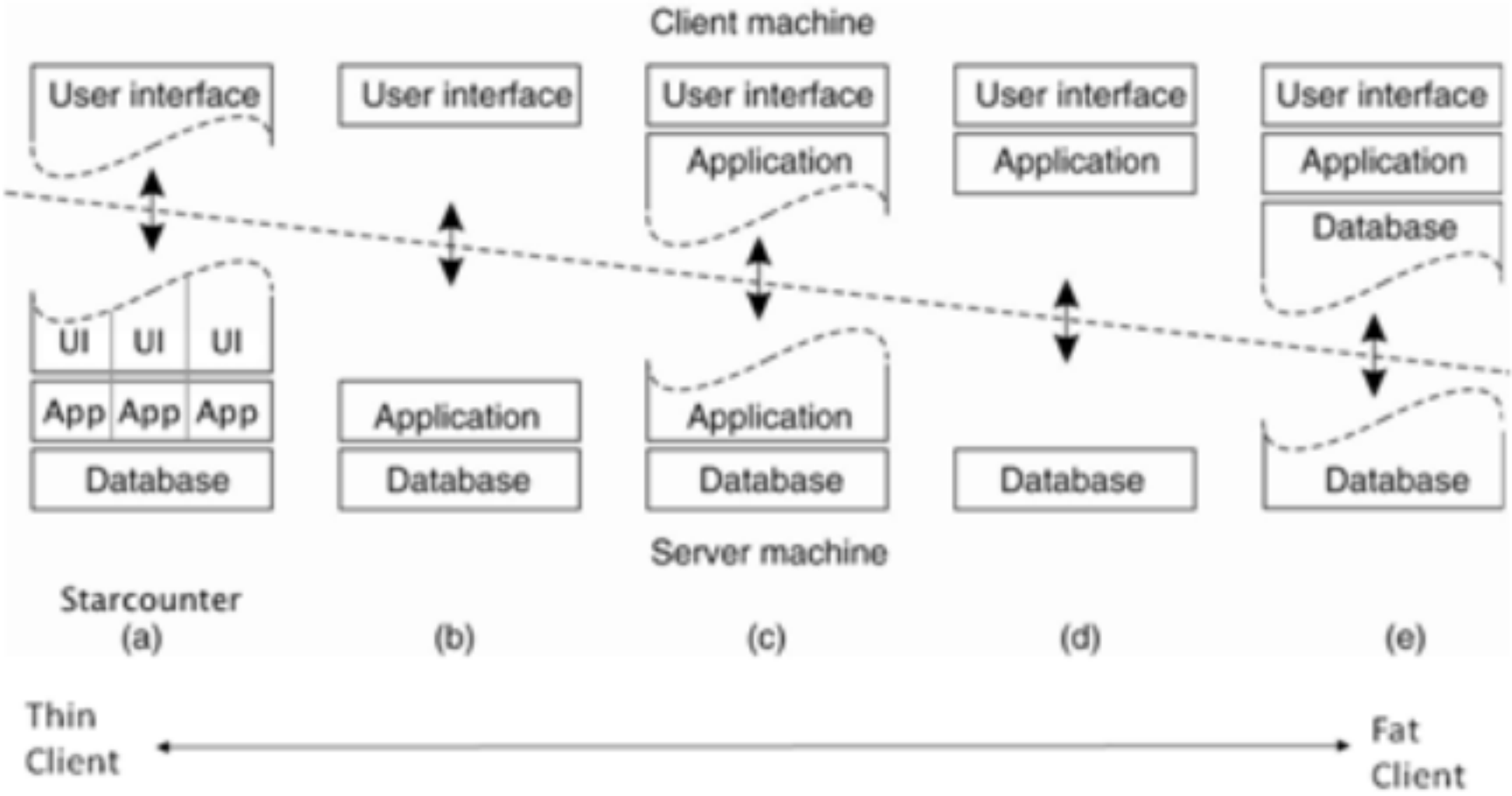
Thin-client model

In thin-client model, all the application processing and data management is carried by the server. The client is simply responsible for running the GUI software. It is used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client.
However, A major disadvantage is that it places a heavy processing load on both the server and the network.

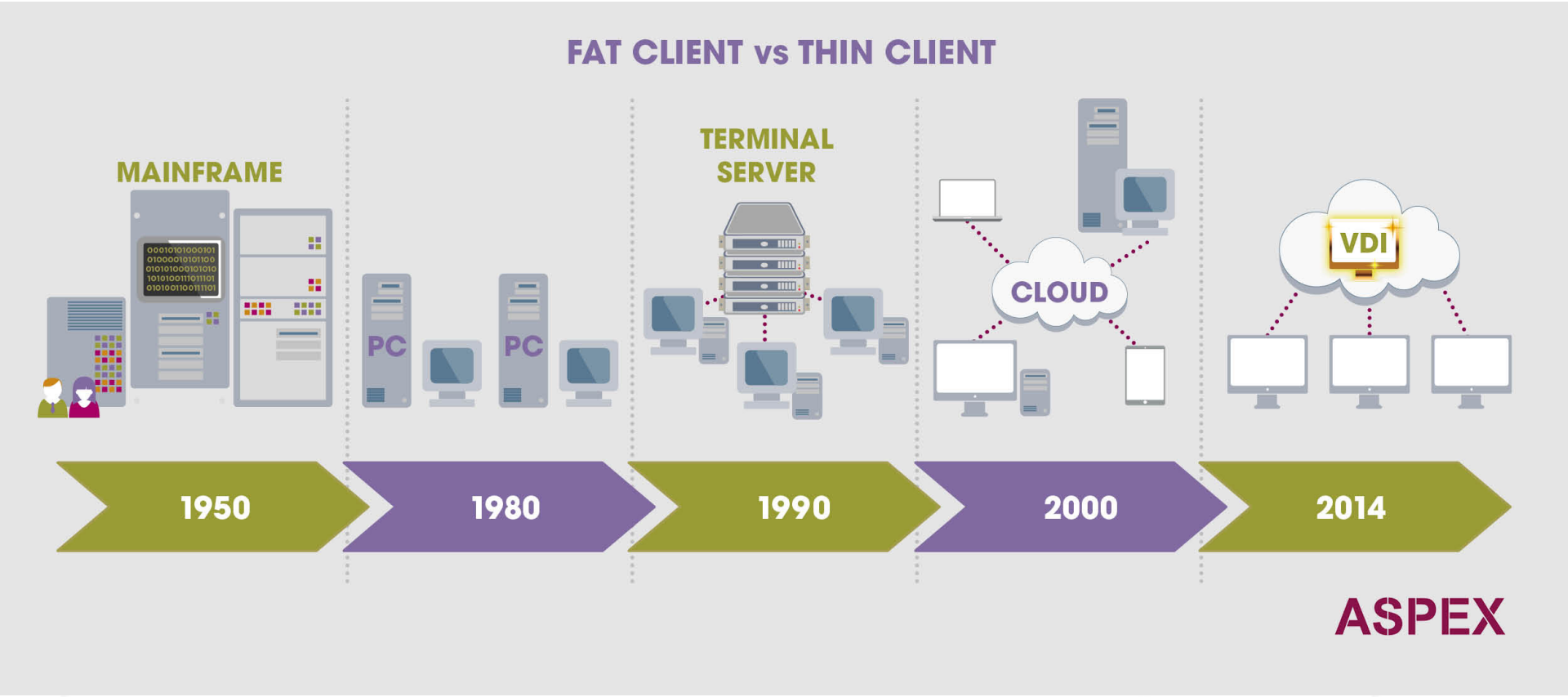
Thick/Fat-client model

In thick-client model, the server is in-charge of the data management. The software on the client implements the application logic and the interactions with the system user. It is most appropriate for new client-server systems where the capabilities of the client system are known in advance.
However, it is more complex than a thin client model especially for management, as all clients should have same copy/version of software application.

Thin vs. Thick/Fat Client



Thin vs. Thick/Fat Client



1.3 Distributed Systems

Thin/Thick Clients Advantages:

- Separation of responsibilities such as user interface presentation and business logic processing.
- Reusability of server components and potential for concurrency
- Simplifies the design and the development of distributed applications
- It makes it easy to migrate or integrate existing applications into a distributed environment.
- It also makes effective use of resources when a large number of clients are accessing a high-performance server.

Thin/Thick Clients Disadvantages:

- Lack of heterogeneous infrastructure to deal with the requirement changes.
- Security complications.
- Limited server availability and reliability.
- Limited testability and scalability.
- Fat clients with presentation and business logic together.

1.4 Distributed Systems Challenges

1. Heterogeneity – variety and difference in:
- networks
- computer hardware
- OS
- programming languages
- implementations by different developers

2. OPENNESS

- computer system - can the system be extended and re-implemented in various ways?
- distributed system - can new resource-sharing services be added and made available for use by variety of client programs?

3. Security
- Confidentiality
- Integrity
- Availability

Main goal: High availability

4. Scalability
–the ability of the system system to work well when the system load or the number of users increases

Challenges with building scalable distributed systems:
- Controlling the cost of physical resources
- Controlling the performance loss
- Preventing software resources running out (like 32-bit internet addresses, which are being replaced by 128 bits)
- Avoiding performance bottlenecks

5. Failure handling
Techniques for dealing with failures
- Detecting failures
- Masking failures

- messages can be retransmitted
- disks can be replicated in a synchronous action
- Tolerating failures
- Recovery from failures
- Redundancy – redundant components

- at least two different routes
- database can be replicated in several servers

1.4 Distributed Systems Challenges

6. Concurrency – Several clients trying to access shared resource at the same time.

Any object with shared resources in a DS must be responsible that it operates correctly in a concurrent environment.

7. Transparency

Transparency – concealment from the user and the applications programmer of the separation of components in a Distributed System for the system to be perceived as a whole rather than a collection of independent components:

- Access transparency – access to local and remote re- sources identical
- Location transparency – resources accessed without knowing their physical or network location
- Concurrency transparency – concurrent operation of processes using shared resources without interference be- tween them
- Replication transparency – multiple instances seem like one
- Failure transparency – fault concealment
- Mobility transparency – movement of resources/clients within a system without affecting the operation of users or programs

8. Quality of service

Main nonfunctional properties of systems that affect *Quality of Service (QoS):*

- reliability
- security
- performance

Section Conclusion

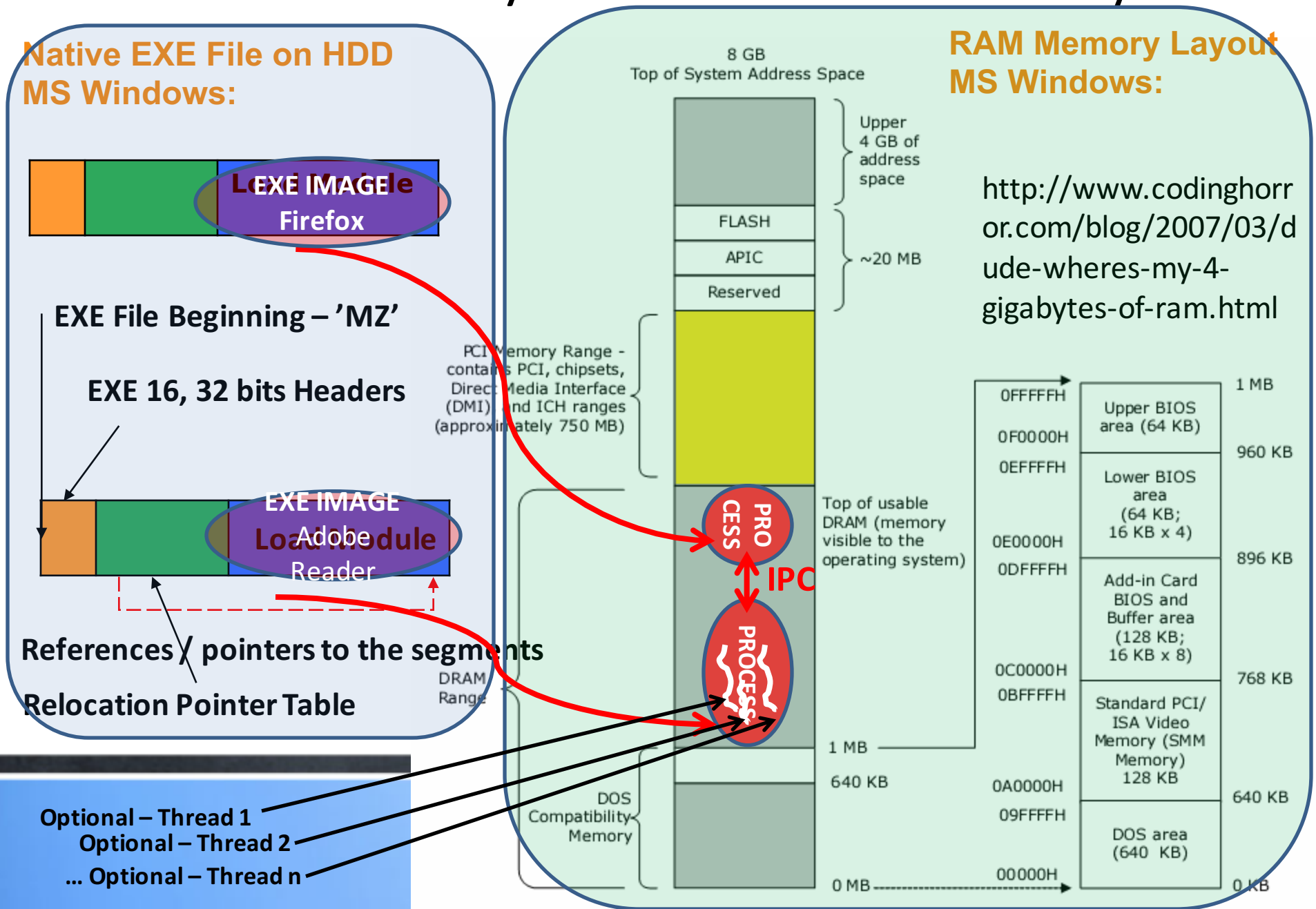
Fact: **DAD needs Java and ECMAScript/JS/node.js**

In few **samples** it is simple to remember: Types of the distributed architectural patterns of the distributed systems.

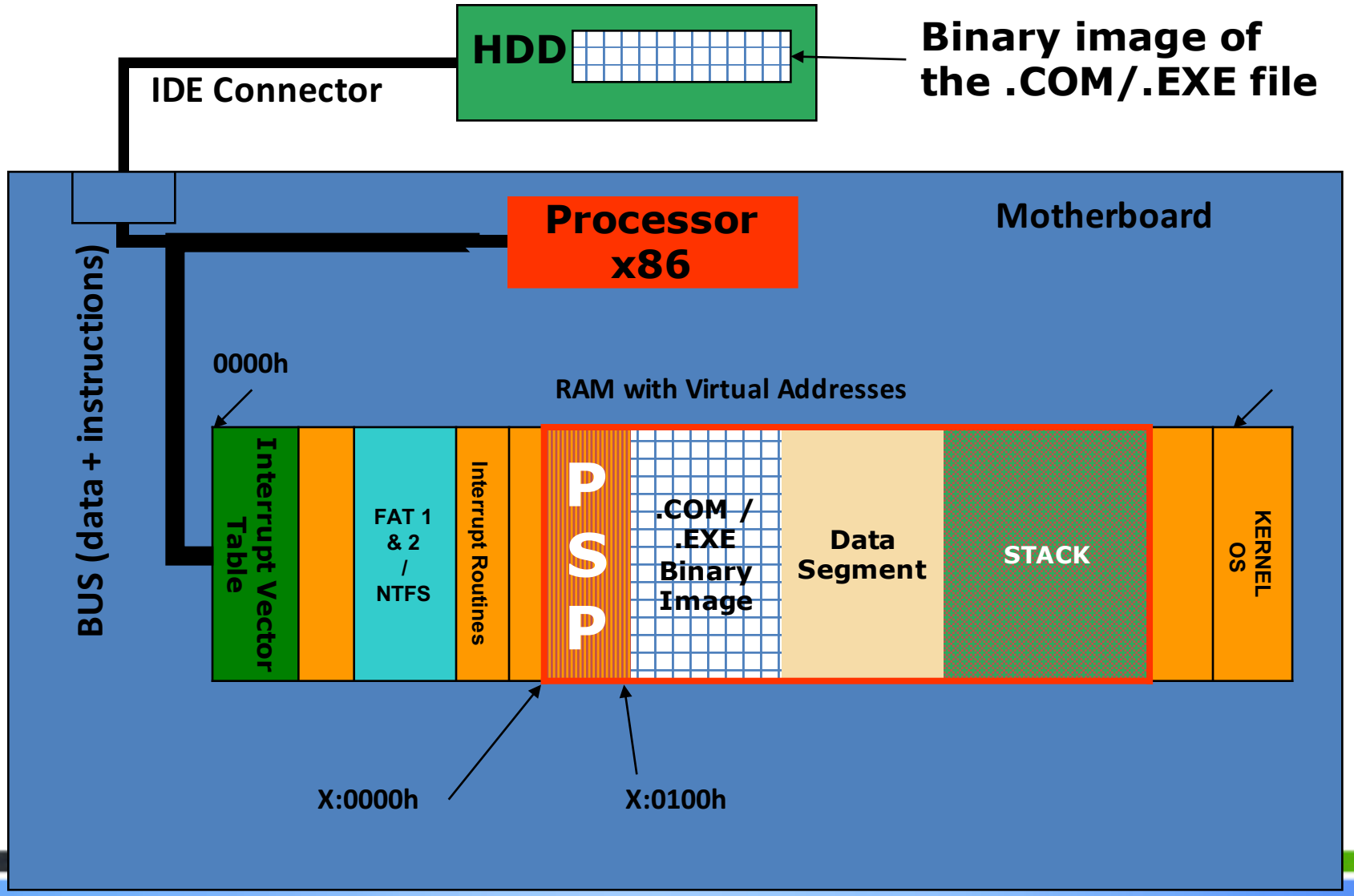
Linux IPC – Inter-Process Communication, light-weight processes / process thread in C/C++
Linux – pthread library and C++ ’11, JVM and OS threads, Java Multi-threading issues

Linux IPC, Multi-threading & Java

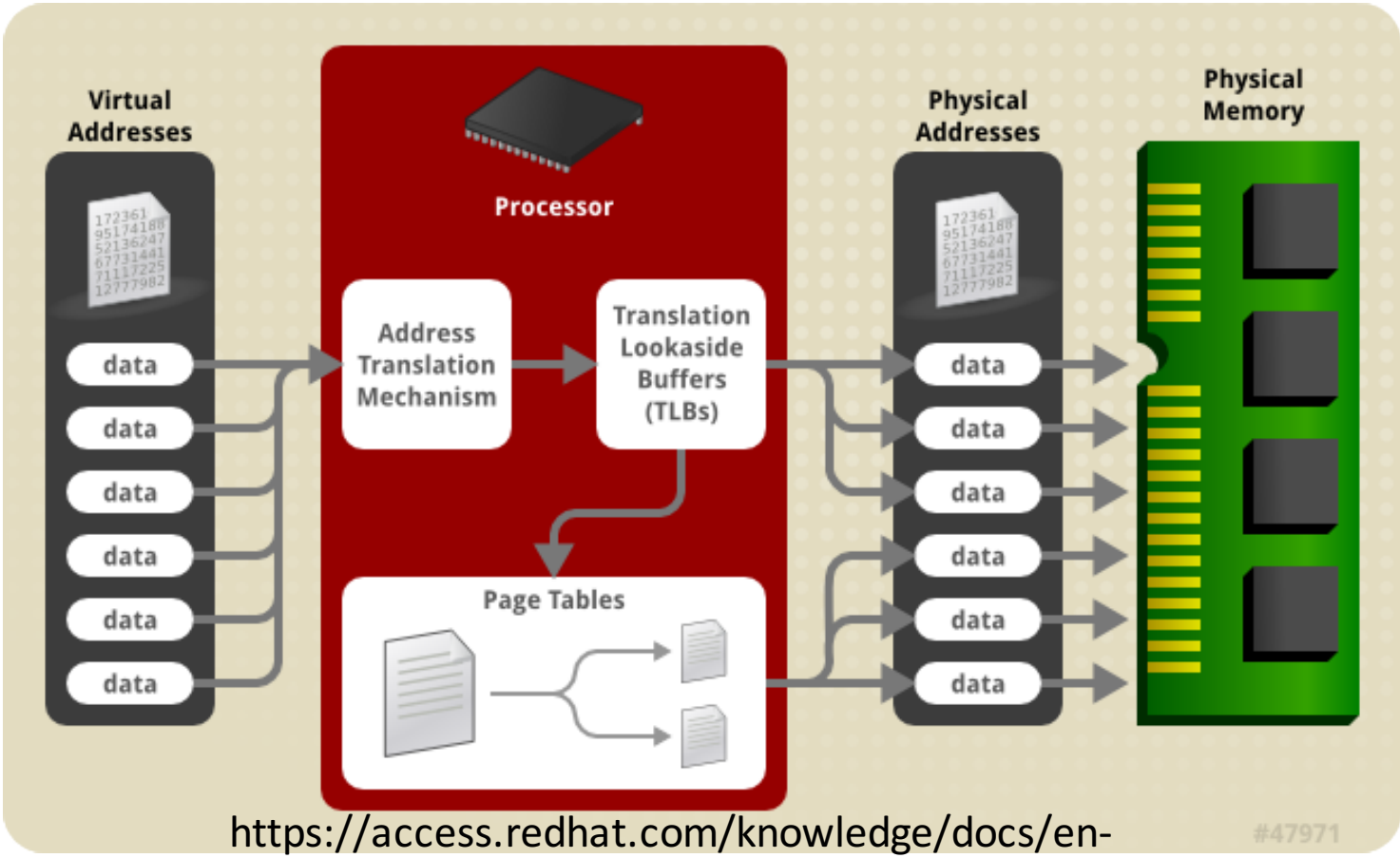
2.7 Summary of MS Windows Memory



2.7 Summary of MS Windows Process

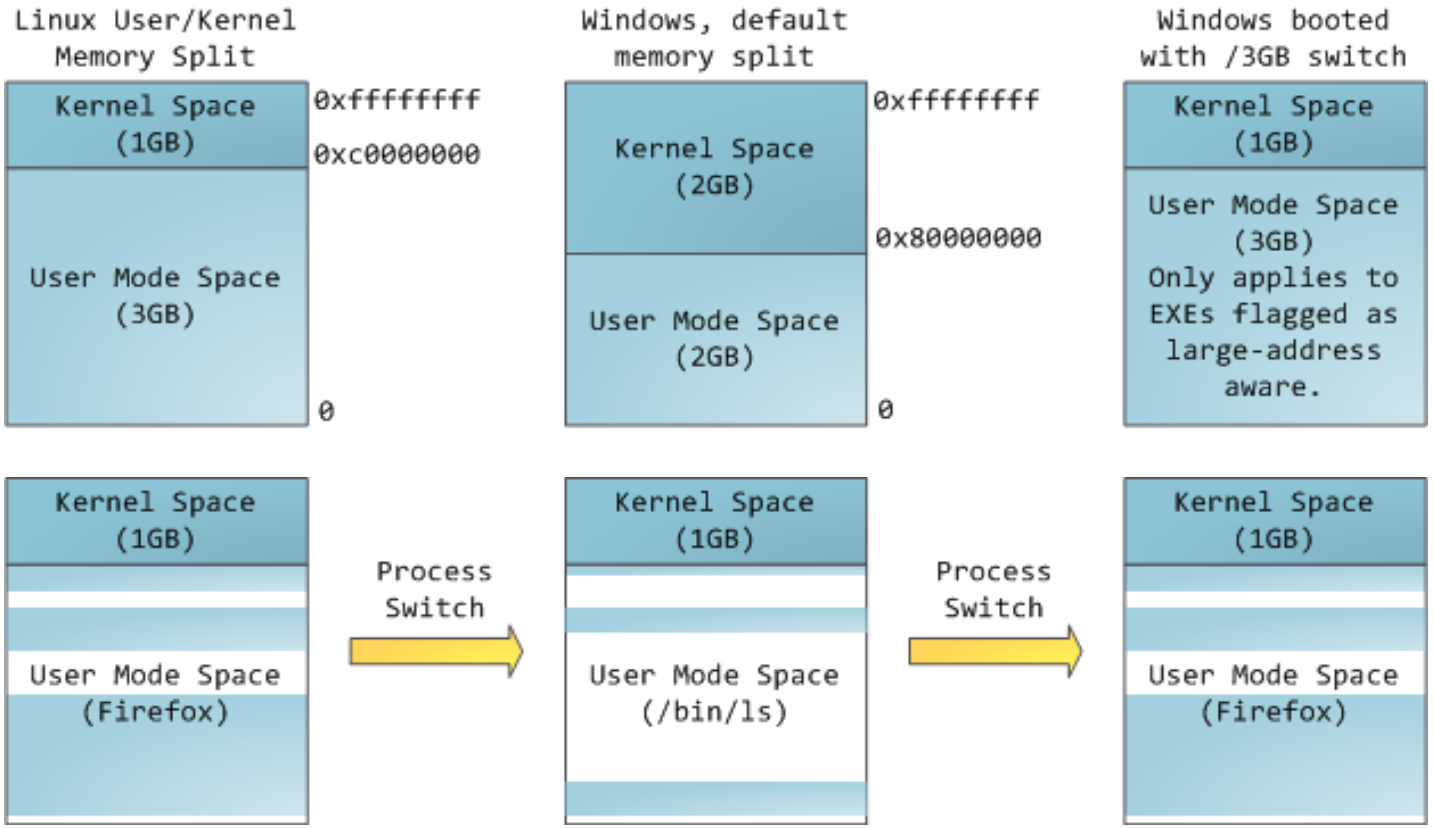


2.7 Summary of Linux/Windows Virtual Memory



https://access.redhat.com/knowledge/docs/en-US/Red_Hat_Enterprise_MRG/2/html/Realtime_Reference_Guide/chap-Realtime_Reference_Guide-Memory_allocation.html

2.7 Summary of Linux/Windows Virtual Memory

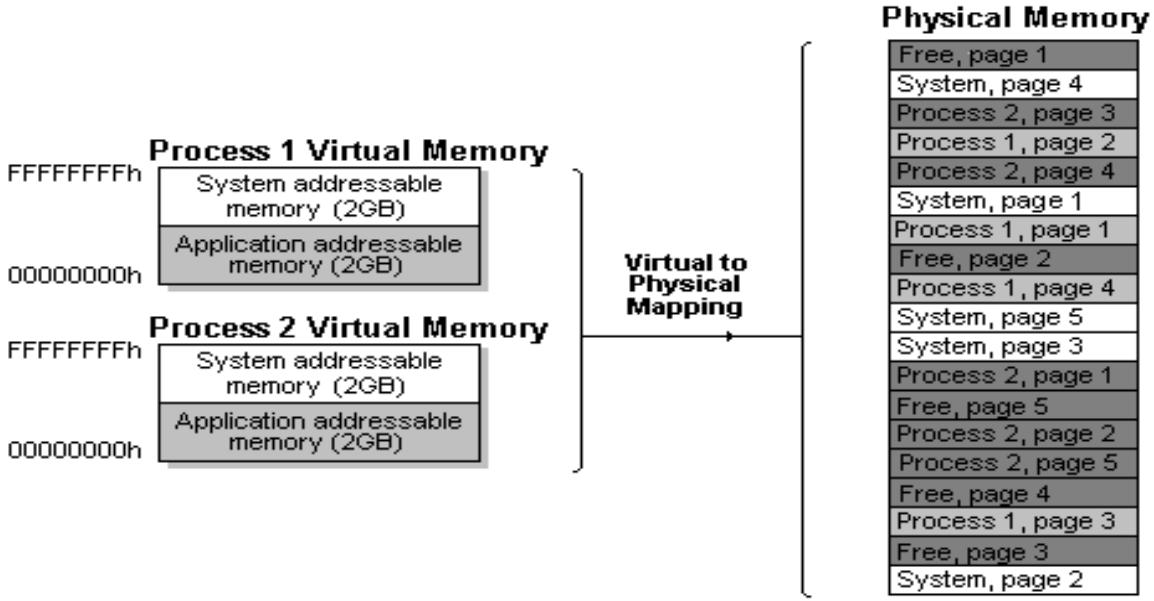


"Blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped. In the example above, Firefox has used far more of its virtual address space due to its legendary memory hunger. The distinct bands in the address space correspond to **memory segments** like the heap, stack, and so on. Keep in mind these segments are simply a range of memory addresses and *have nothing to do* with Intel-style segments."

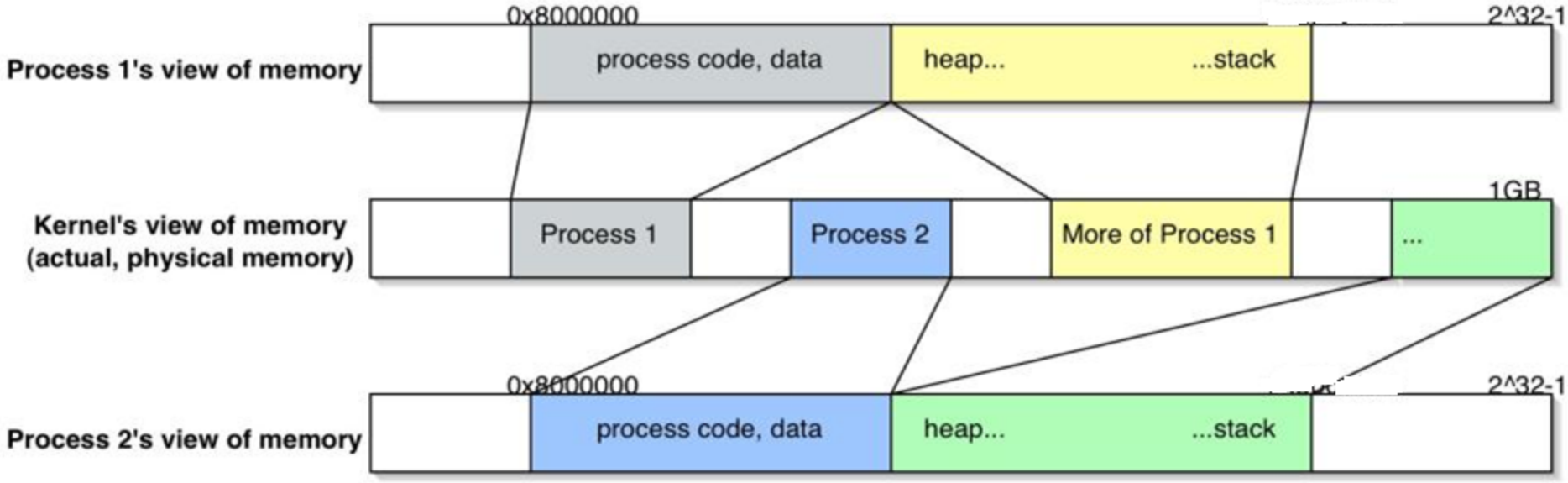
http://duartes.org/gustavo/blog/post/anatomy-of-a-program-in-memory
http://duartes.org/gustavo/blog/category/internals

2.7 Summary of Linux/Windows Virtual Memory

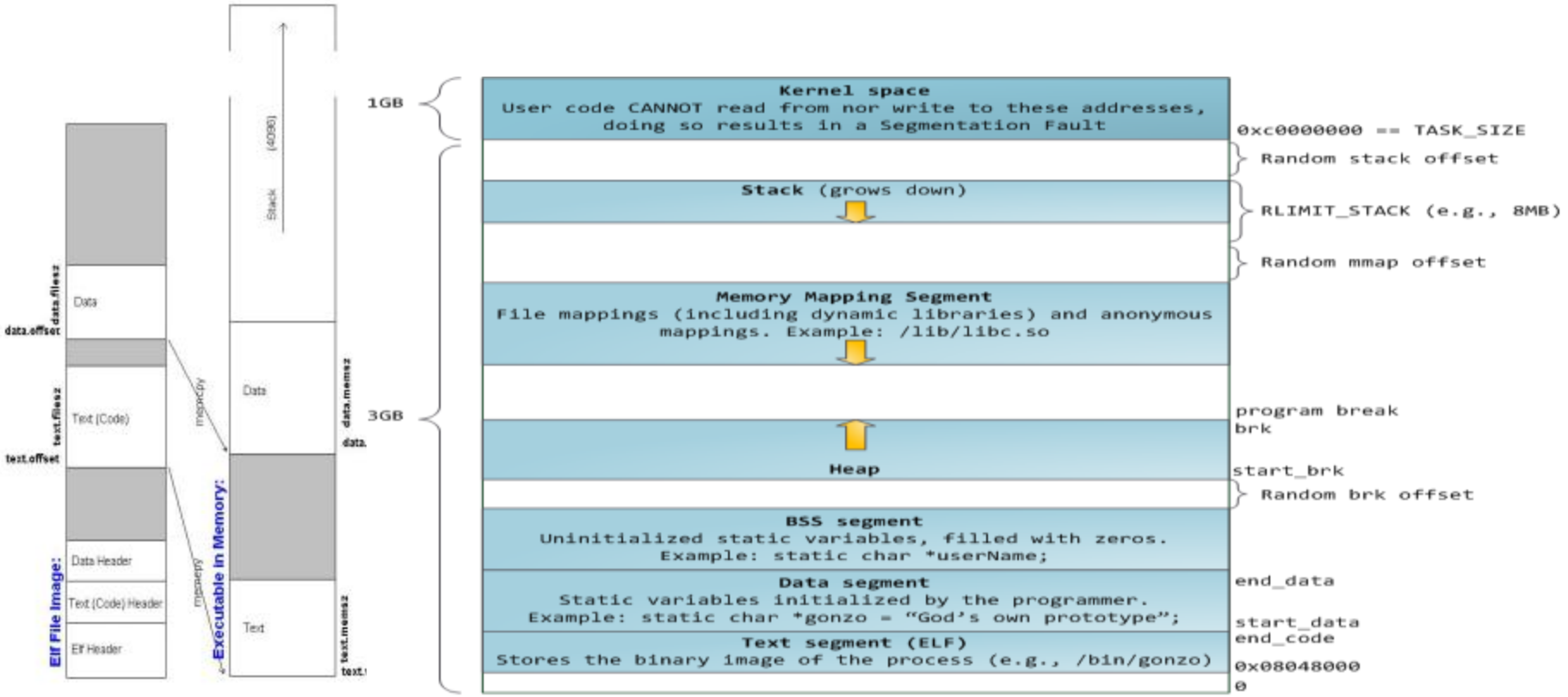
MS Windows:
http://technet.microsoft.com/en-us/library/cc751283.aspx



LINUX: http://www.read.cs.ucla.edu/111/2007fall/notes/lec4



2.7 Summary of Linux executable ELF to memory - Process



http://www.cs.umd.edu/~hollings/cs412/s04/proj1/index.html#cast

2.7 Summary of Processes & IPC in Linux

http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html

https://computing.llnl.gov/tutorials/pthreads/ http://www.advancedlinuxprogramming.com/alp-folder/

Before understanding a thread, one first needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of "overhead".

Processes contain information about program resources & program execution state, including:

- *Process ID, process group ID, user ID, and group ID;*
- *Environment;*
- *Working directory;*
- *Program instructions;*
- *Registers;*
- *Stack;*
- *Heap;*
- *File descriptors;*
- *Signal actions;*
- *Shared libraries;*
- *Inter-process communication tools (such as message queues, pipes, semaphores, or shared memory)*

2.7 Summary of Processes & IPC in Linux

Processes
- Fork
- Signals

Pipes

FIFO

File-locking

OS Message Queues

Semaphores

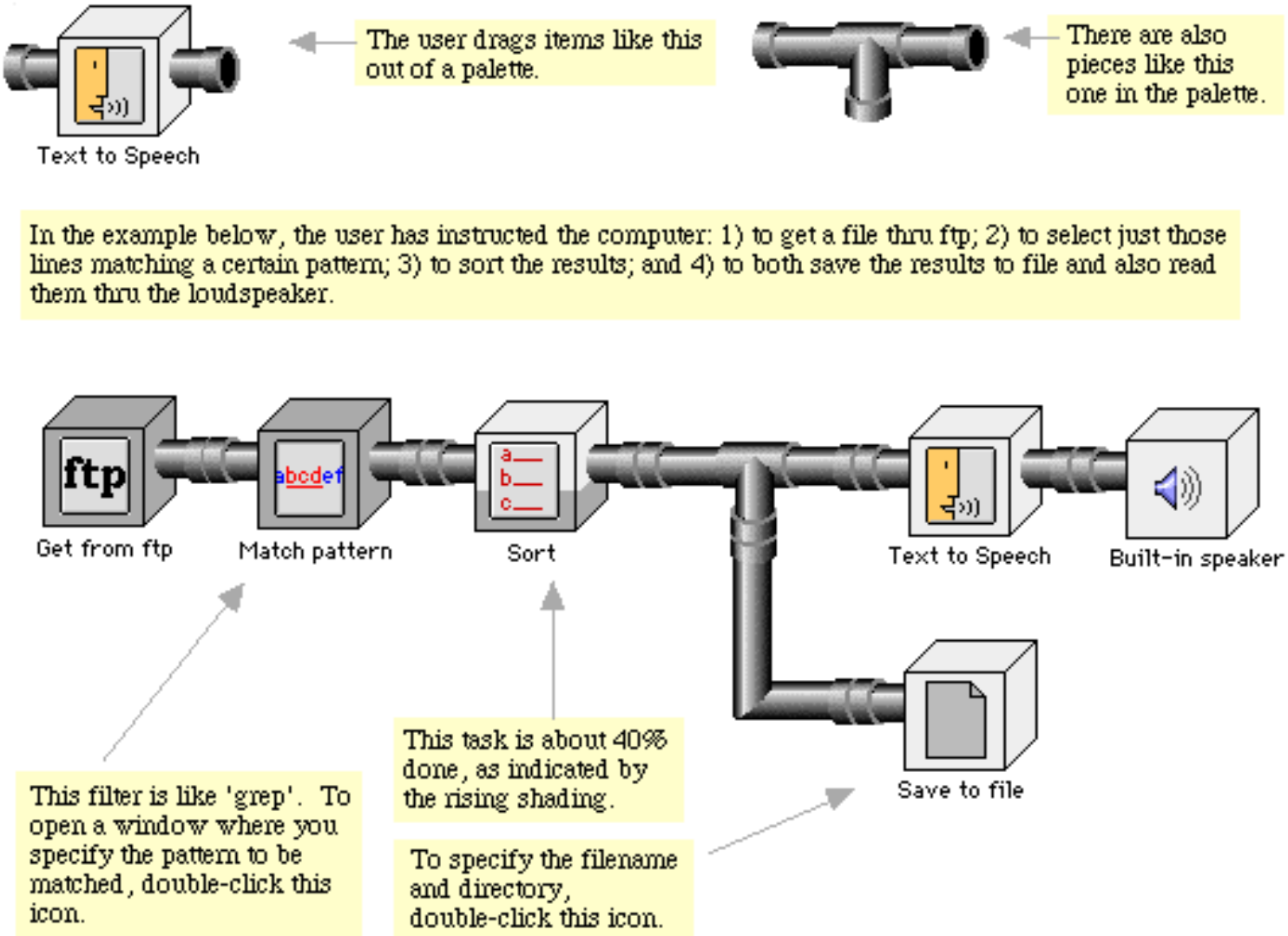
Shared Memory

Memory Mapped Files

Sockets

2.7 Summary of IPC in Linux – Why Pipes?

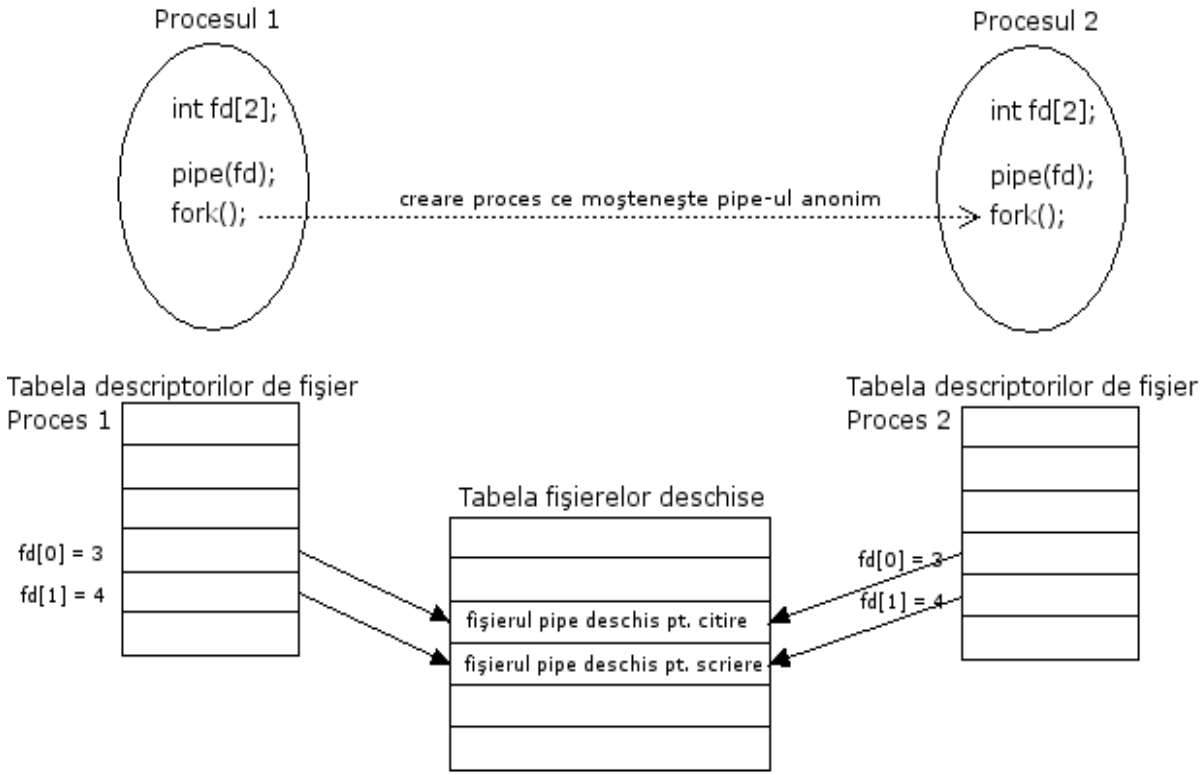
http://www.sean-crist.com/personal/pages/visual_pipes/

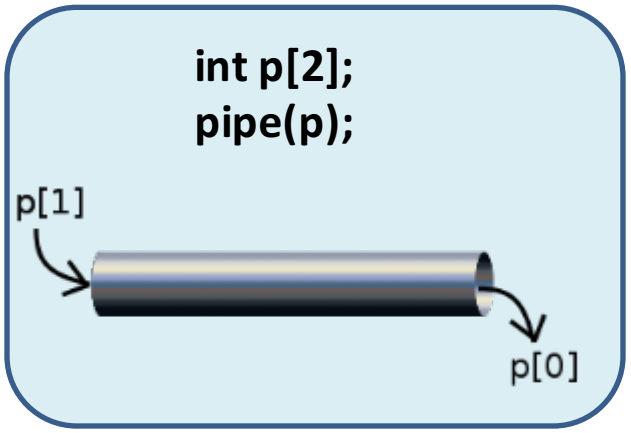


Many people have observed that Linux is difficult for casual users to learn, and that Linux would have a better chance of general acceptance as a desktop platform if it were made easier to use. Pipes are at the root of the great flexibility of Unix, and representing them graphically makes this functionality better accessible to the casual user.

2.7 Summary of IPC in Linux – Fork & Pipes

http://www.reloco.com.ar/linux/prog/pipes.html

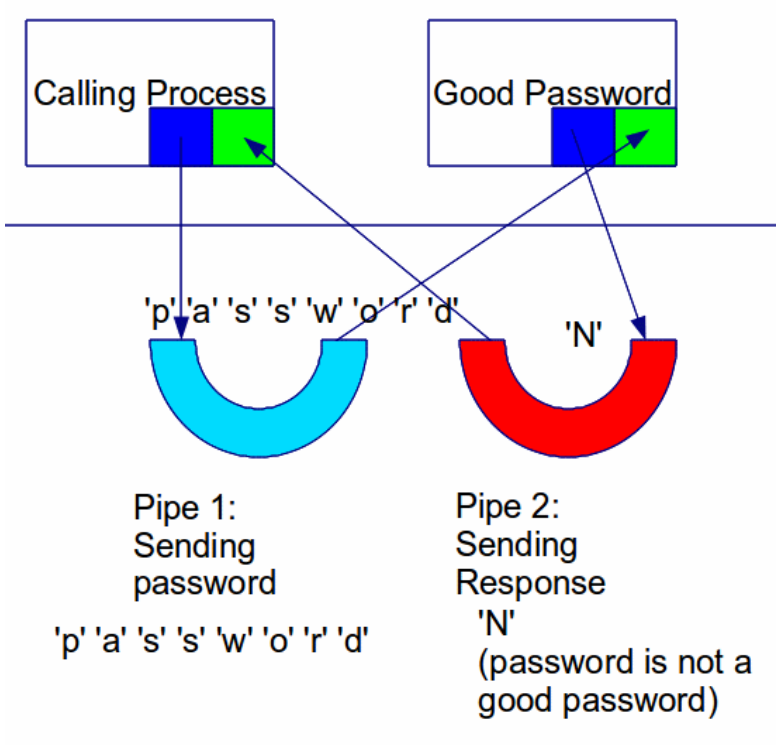


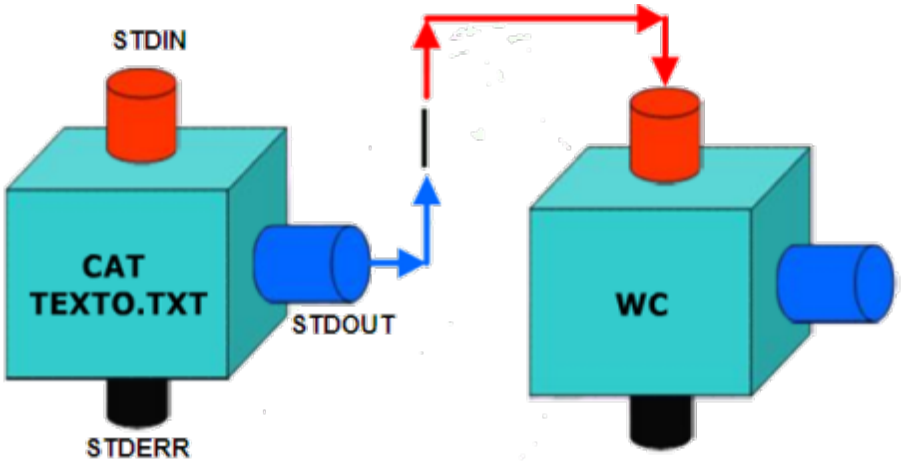


http://os.obs.utcluj.ro/OS/Lab/08.Linux%20Pipes.html

2.7 Summary of IPC in Linux – Fork & Pipes

http://www.vivaolinux.com.br/dica/Pipes-no-Linux

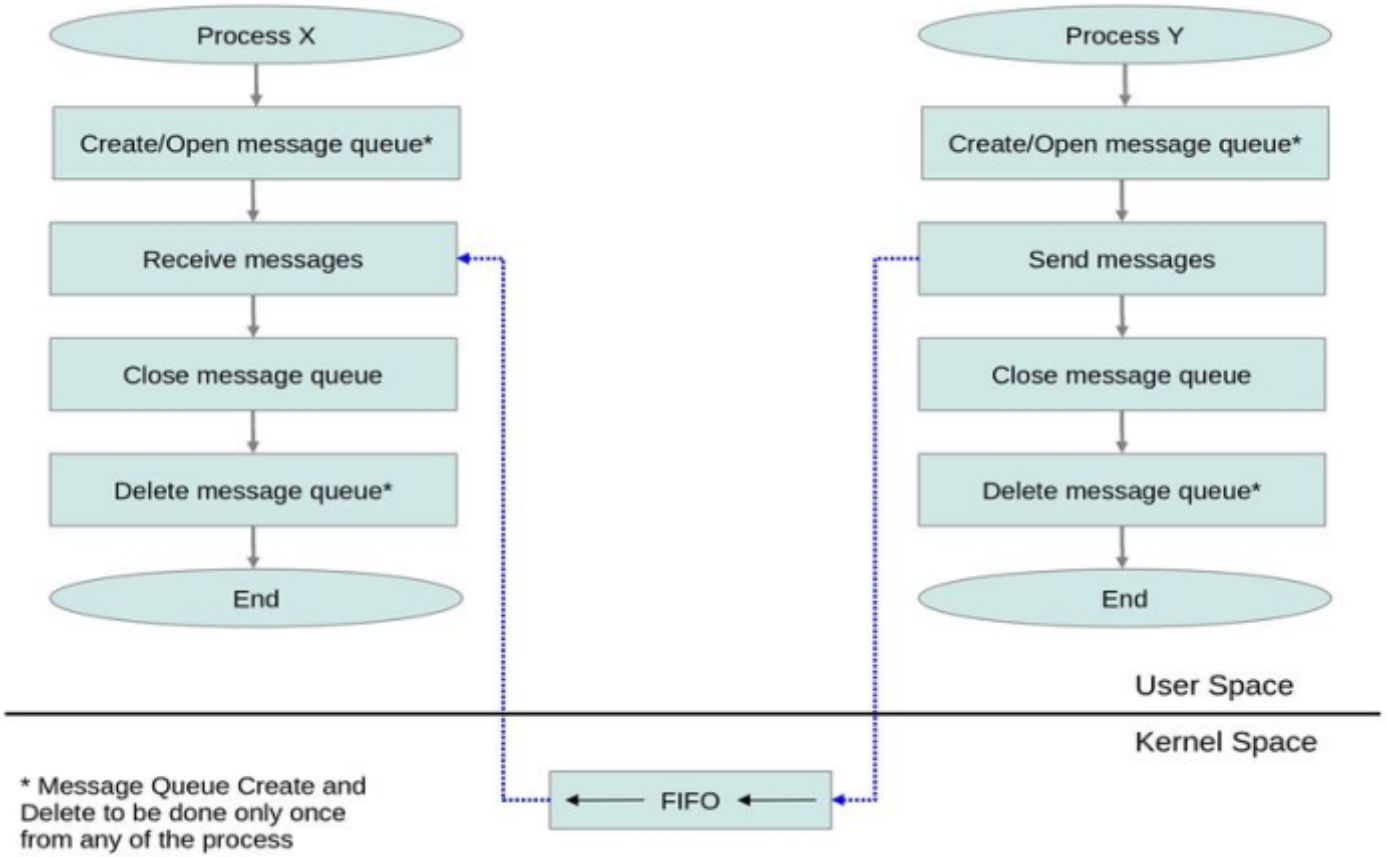




http://www.read.cs.ucla.edu/111/_media/notes/ipc_pipes_1.gif

2.7 Summary of IPC in Linux – Message Queues

Linux C - System V API / POSIX API



http://www.linuxpedia.org/index.php?title=Linux_POSIX_Message_Queue

2.7 Summary of IPC in Linux – Message Queues

Linux C - System V API / POSIX API

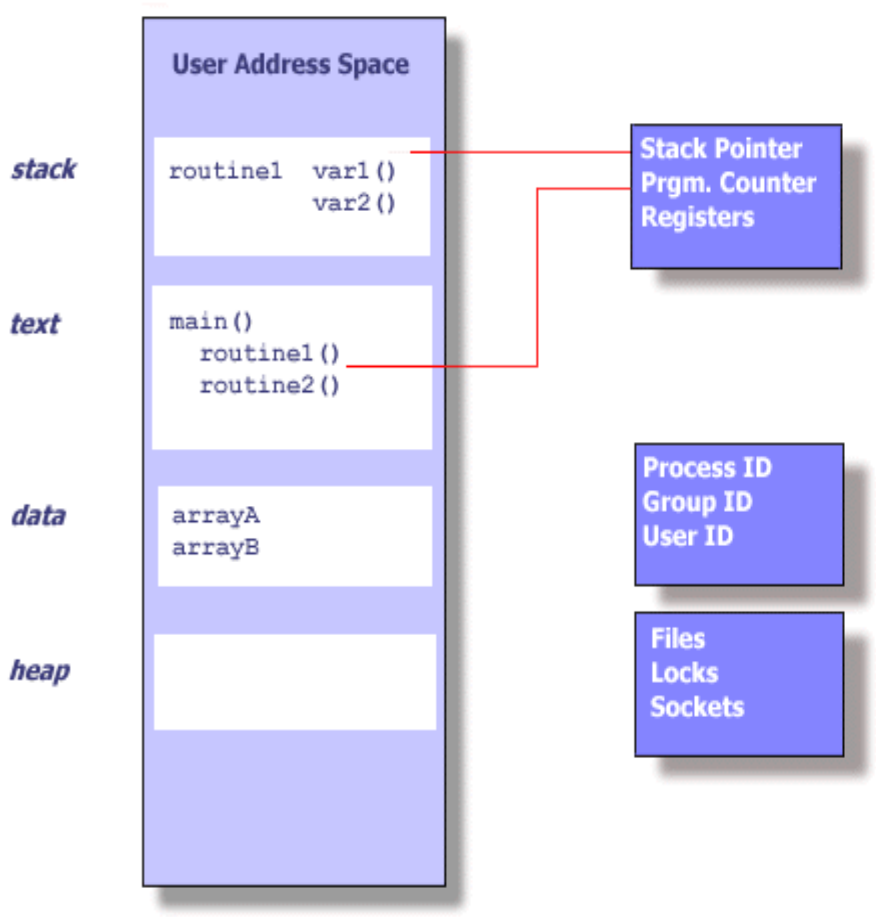
Operation	POSIX Function	SVR4 Function
Gain access to a queue, creating it if it does not exist.	mq_open(3)	msgget(2)
Query attributes of a queue and number of pending messages.	mq_getattr(3)	msgctl(2)
Change attributes of a queue.	mq_setattr(3)	msgctl(2)
Give up access to a queue.	mq_close(3)	n.a.
Remove a queue from the system.	mq_unlink(3), rm(1)	msgctl(2), ipcrm(1)
Send a message to a queue.	mq_send(3)	msgsnd(2)
Receive a message from a queue.	mq_receive(3)	msgrcv(2)
Request asynchronous notification of a message arriving at a queue.	mq_notify(3)	NA

http://menehune.opt.wfu.edu/Kokua/More_SGI/007-2478-008/sgi_html/ch06.html

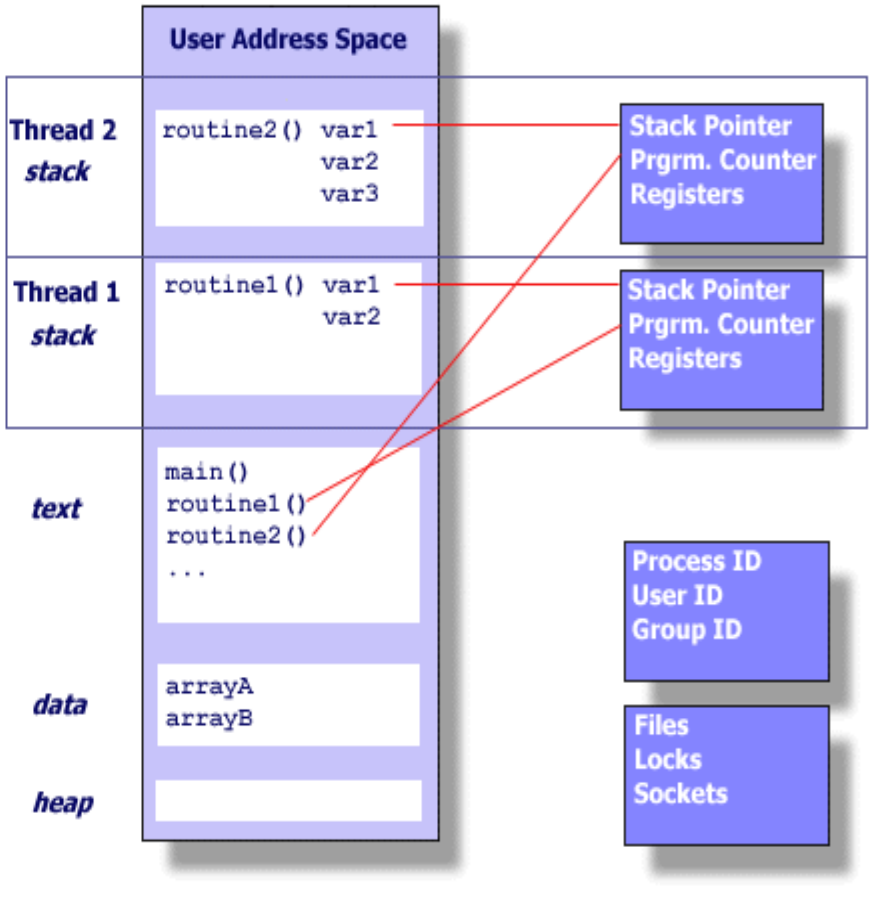
http://www.users.pjwstk.edu.pl/~jms/qnx/help/watcom/clibref/mq_overview.html

2.8 Summary of Multi-threading in C vs. Java

Multi-threading vs. Multi-process development in UNIX/Linux:



UNIX PROCESS



THREADS WITHIN A UNIX PROCESS

https://computing.llnl.gov/tutorials/pthreads/

2.8 Summary of Multi-threading in C vs. Java

Threads Features:

Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.

A thread does not maintain a list of created threads, nor does it know the thread that created it.

All threads within a process share the same address space.

Threads in the same process share:

- Process instructions
- Most data
- open files (descriptors)
- signals and signal handlers
- current working directory
- User and group id

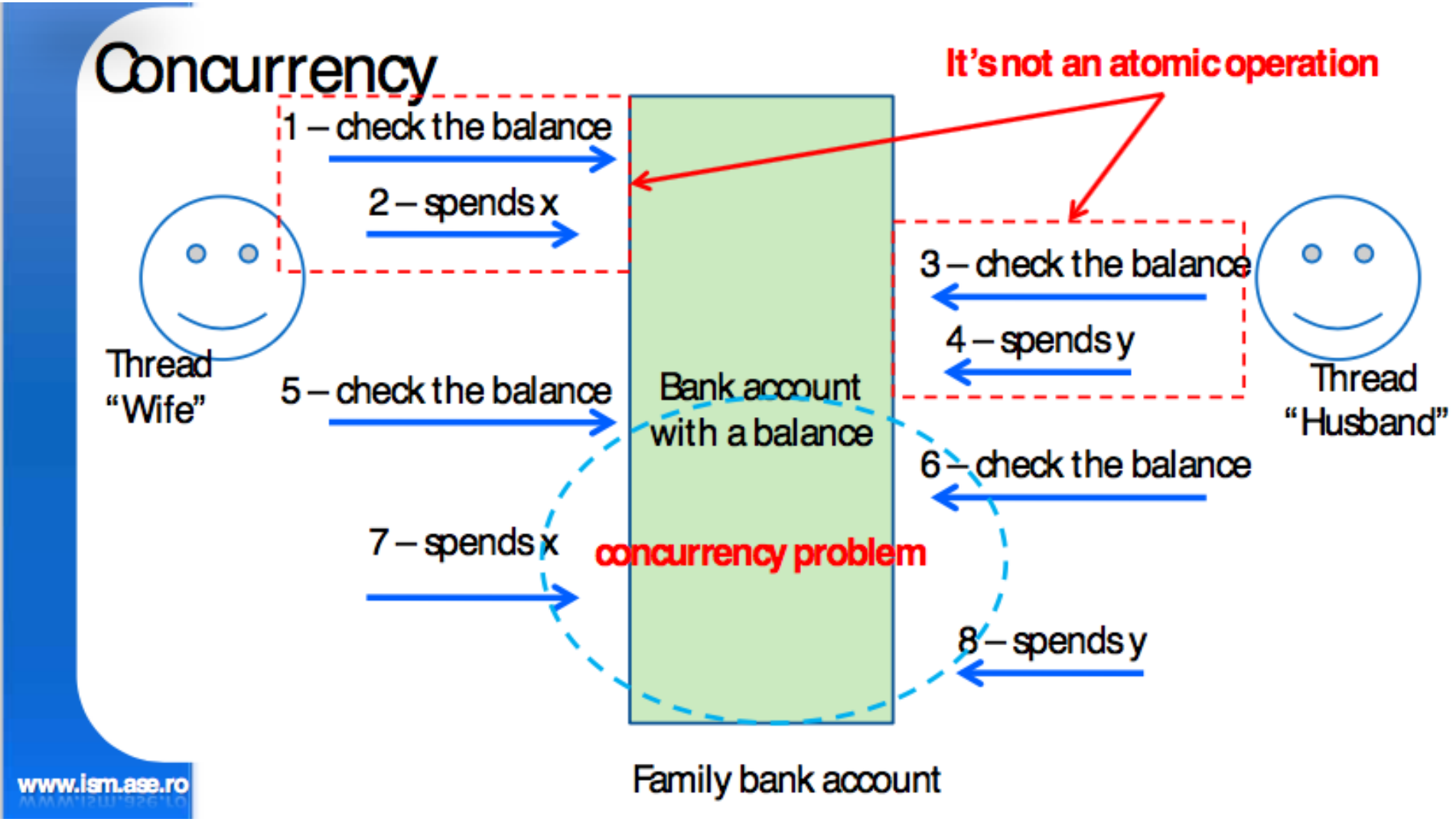
Each thread has a unique:

- Thread ID
- set of registers, stack pointer
- stack for local variables, return addresses
- signal mask
- priority
- Return value: errno

pthread functions return "0" if OK.

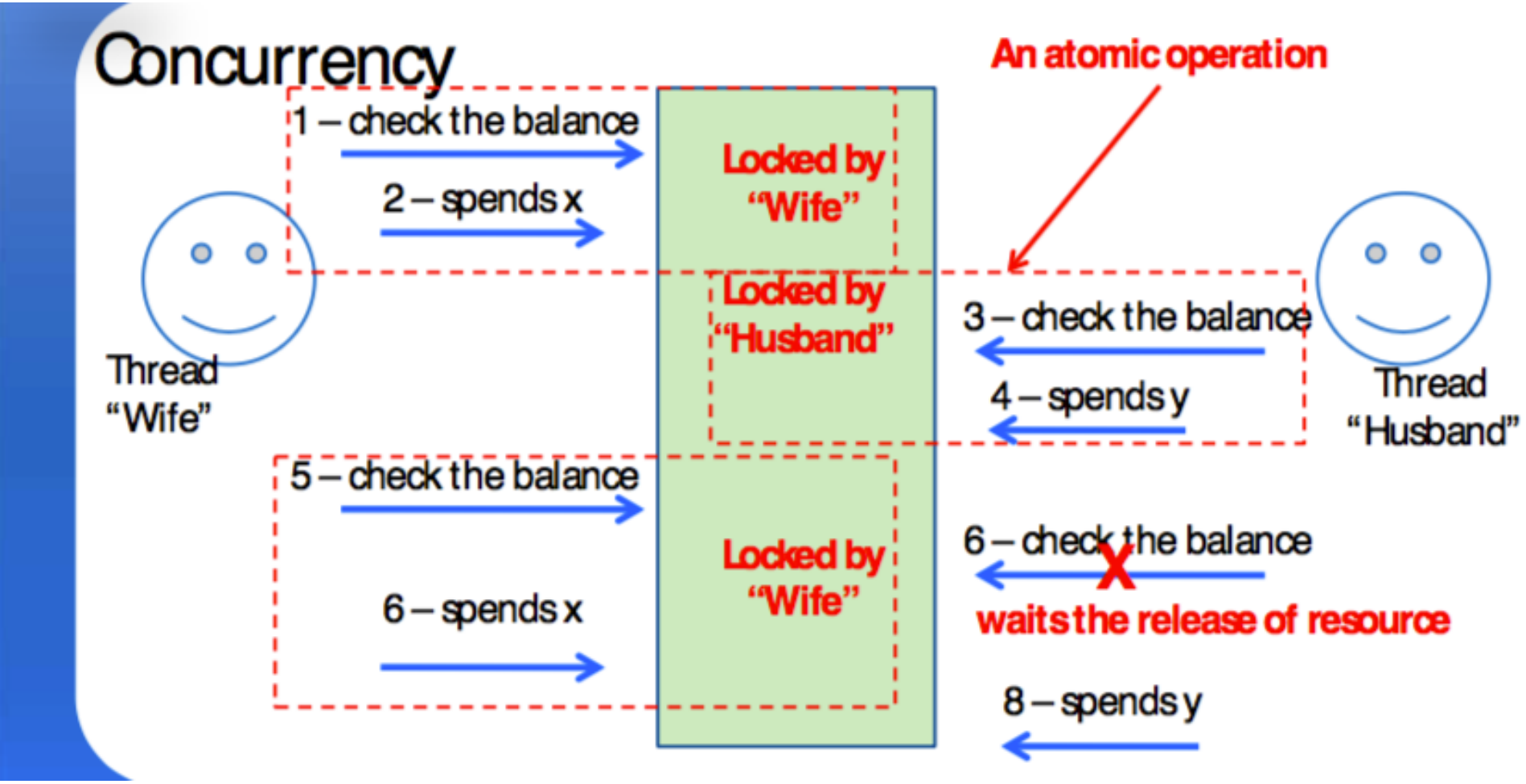
2.8 Summary of Race Condition

Race Condition Feature – Family Bank Account:



2.8 Summary of Race Condition

Race Condition Feature – Family Bank Account:



2.8 Summary of Multi-threading in C vs. Java

Multi-threading in C/C++ with pthread *(Is "counter++" an atomic operation?)*:

Without Mutex

```
1  int counter=0;
2
3  /* Function C */
4  void functionC()
5  {
6
7      counter++
8
9  }
```

With Mutex

```
01 /* Note scope of variable and mutex are the same
   */
02 pthread_mutex_t mutex1 =
   PTHREAD_MUTEX_INITIALIZER;
03 int counter=0;
04
05 /* Function C */
06 void functionC()
07 {
08     pthread_mutex_lock( &mutex1 );
09     counter++
10     pthread_mutex_unlock( &mutex1 );
11 }
```

Possible execution sequence

Thread 1	Thread 2	Thread 1	Thread 2
counter = 0	counter = 0	counter = 0	counter = 0
counter = 1	counter = 1	counter = 1	Thread 2 locked out. Thread 1 has exclusive use of variable `counter`
			counter = 2

2.8 Summary of Multi-threading in C vs. Java

Multi-threading vs. Multi-process mini-terms:

Mutexes are used to prevent data inconsistencies due to race conditions.

A race condition often occurs when two or more threads need to perform operations on the same memory area, but the results of computations depends on the order in which these operations are performed.

Mutexes are used for serializing shared resources. Anytime a global resource is accessed by more than one thread the resource should have a Mutex associated with it.

One can apply a mutex to protect a segment of memory ("critical region") from other threads.

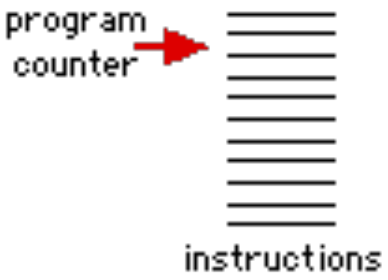
Mutexes can be applied only to threads in a single process and do not work between processes as do semaphores.

In Java Mutex is quite ⇔ synchronized

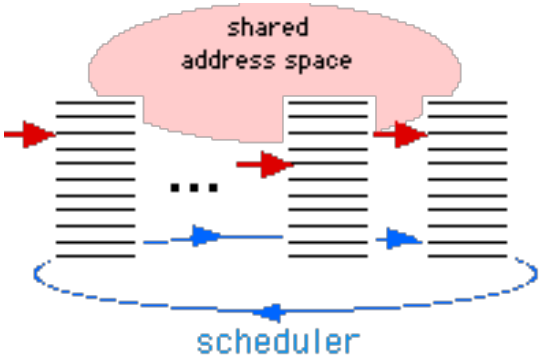
2.8 Summary of Multi-threading in C vs. Java

Multi-threading Models:

http://www-sop.inria.fr/indes/rp/FairThreads/FTJava/documentation/FairThreads.html#One-one-mapping

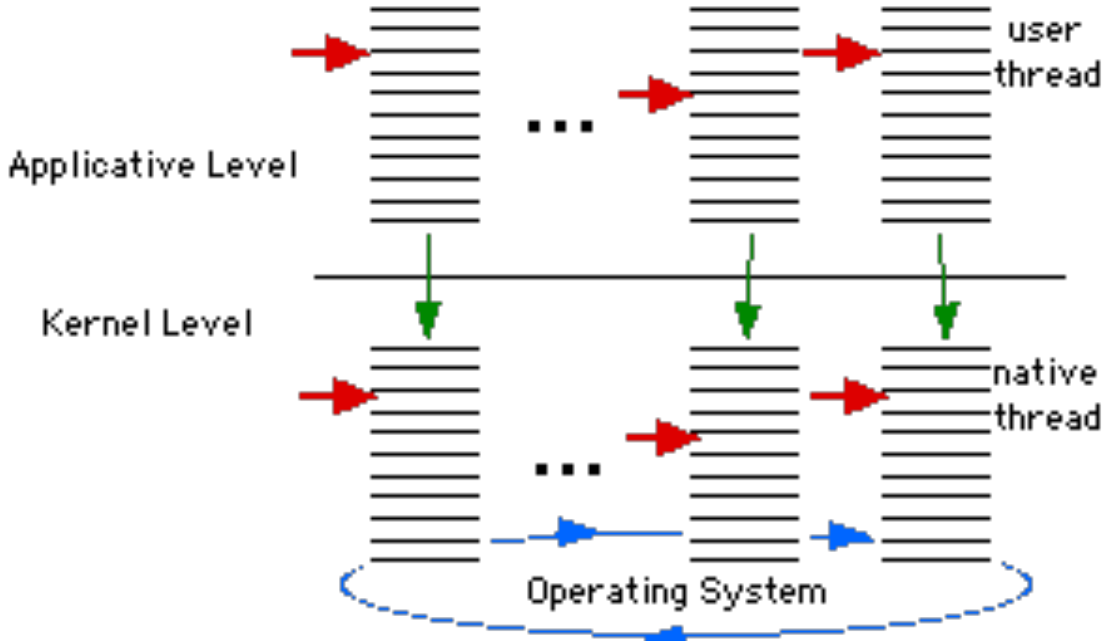


A thread



A scheduler

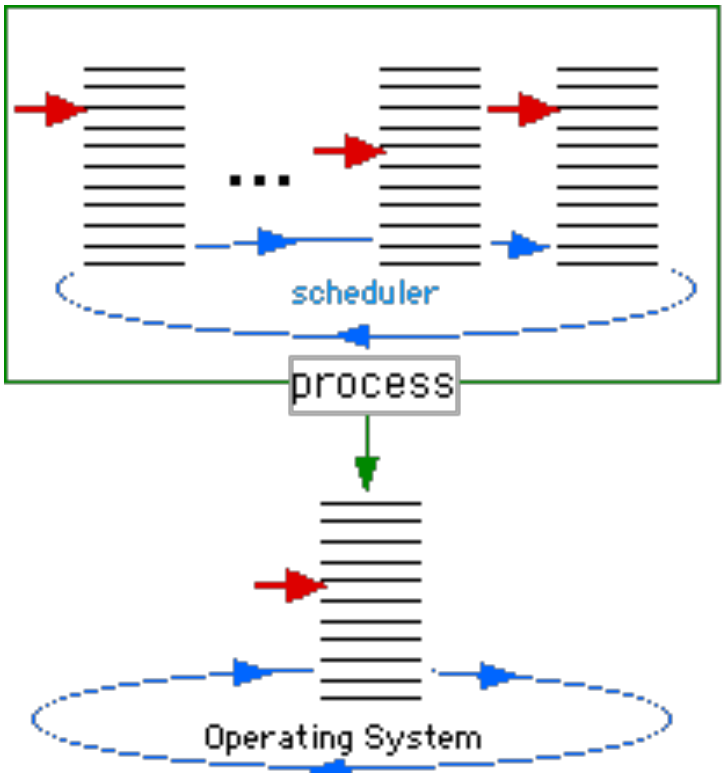
One-to-one Mapping



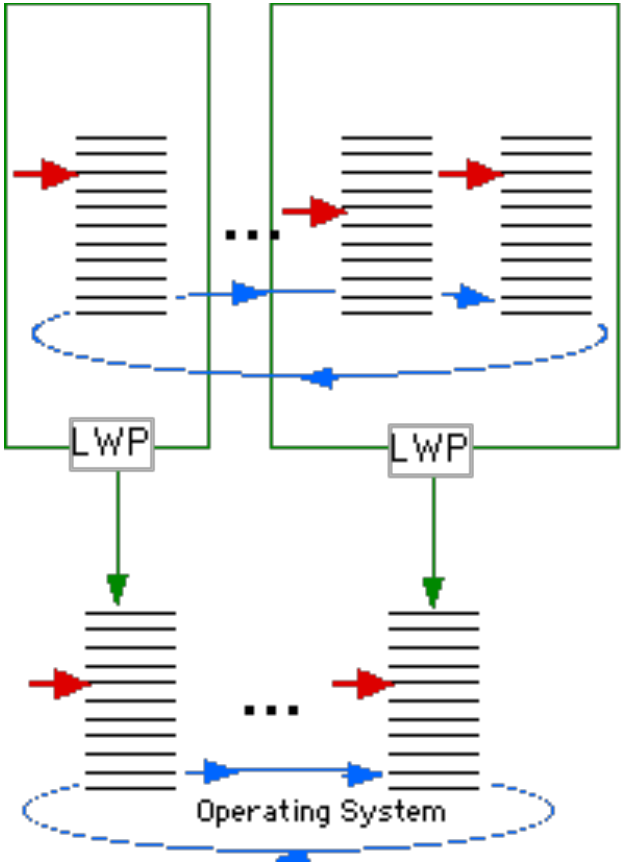
2.8 Summary of Multi-threading in C vs. Java

Multi-threading Models:

http://www-sop.inria.fr/indes/rp/FairThreads/FTJava/documentation/FairThreads.html#One-one-mapping



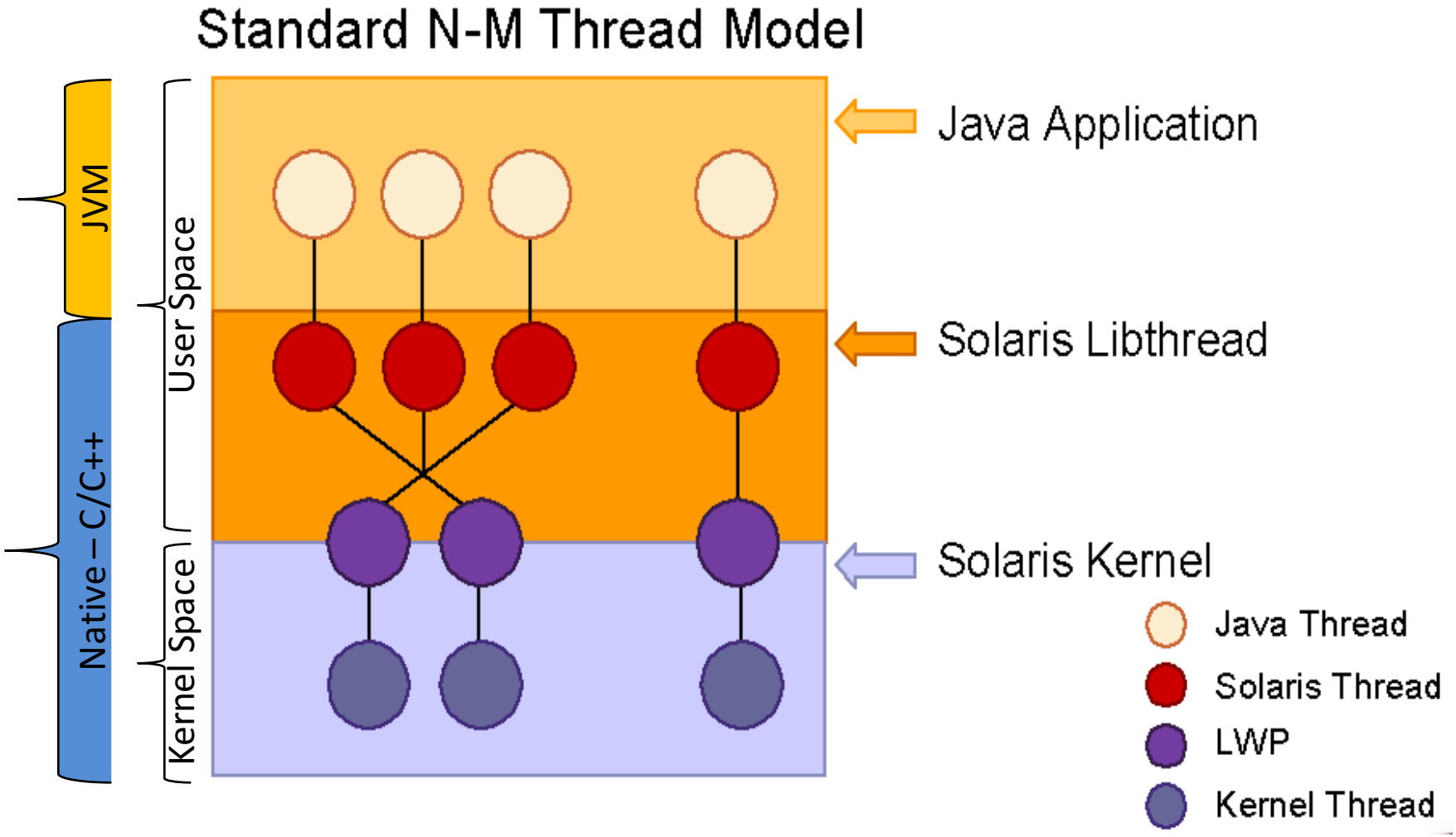
Many-to-one Mapping



Many-to-many Mapping

2.8 Summary of Multi-threading in C vs. Java

JVM Multi-threading Model mapping to OS native threads:

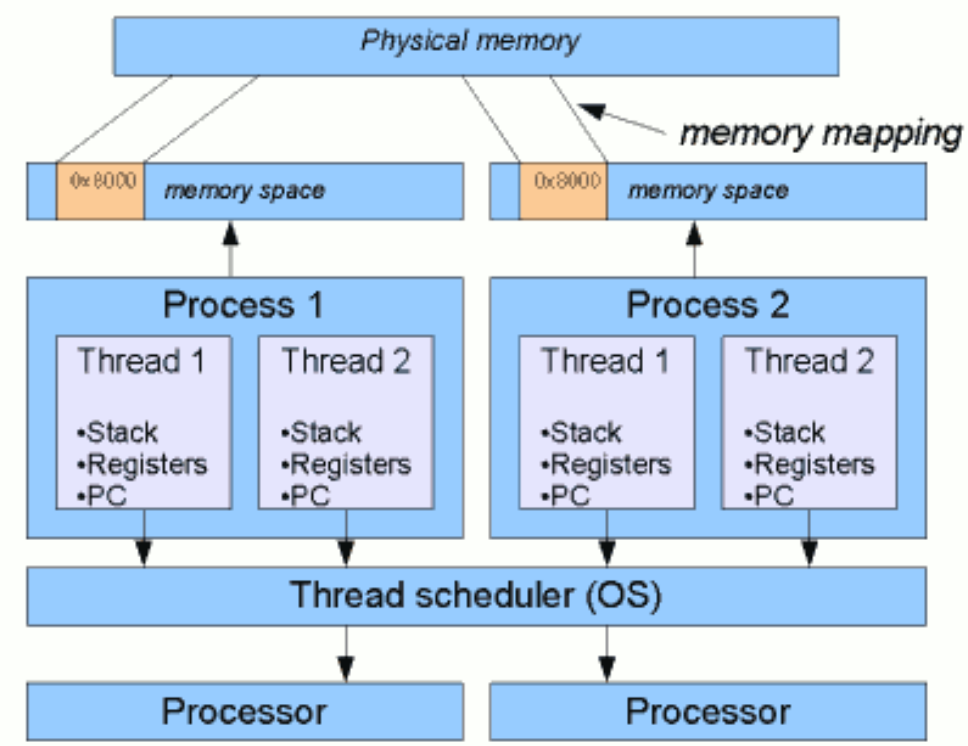


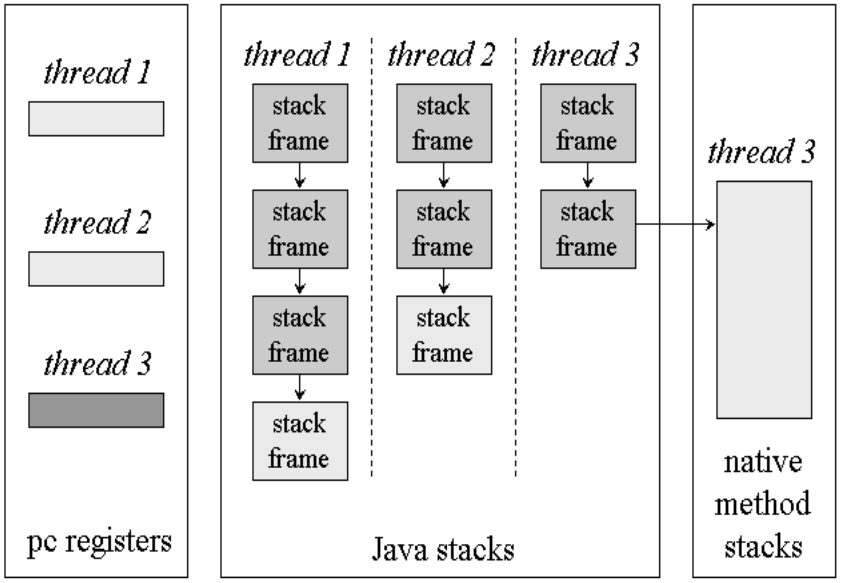
http://192.9.162.55/docs/hotspot/threads/threads.html

2.8 Summary of Multi-threading in C vs. Java

JVM Multi-threading Model mapping to OS native threads:

http://www.javamex.com/tutorials/threads/how_threads_work.shtml





Inside the Java Virtual Machine, Bill Venners

2.9 Summary of Multi-threading in Java

Java Multi-threading API

Option 1 – Java API for Multi-Threading Programming	Option 2 – Java API for Multi-Threading Programming
Defining the classes:	

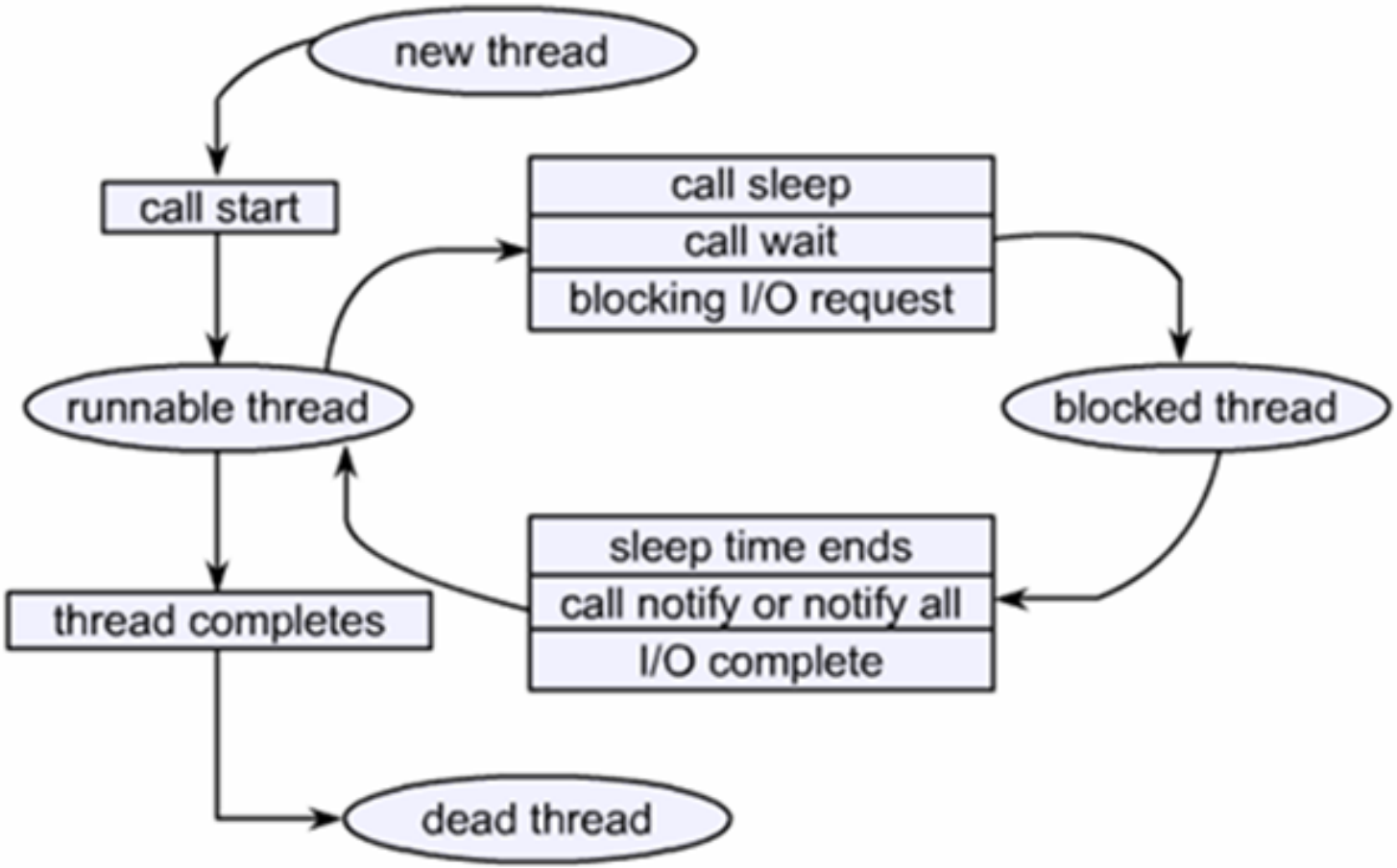
class Fir extends Thread {
  public void run() {…}
}
	class Fir extends Ceva implements Runnable {
	  public void run() {…}
	}

Instantiates the objects	
Fir f = new Fir();	Fir obf = new Fir(); Thread f = new Thread(obf);
Set the thread in 'Runnable' state	
f.start();	f.start();
Specific Thread methods calls	

public void run() {
  Thread.sleep(…);
  String fName = this.getName();
  …
}
	public void run() {
	  Thread.sleep(…);
	  Thread t = Thread.currentThread();
	  String fName = t.getName();
	  …
	}

2.9 Summary of Multi-threading in Java

Java Thread States



Multi-threading & IPC Summary
for easy sharing

Section Conclusions

All threads in a program must run the same executable. A child process, on the other hand, may run a different executable by calling an exec function.

An errant thread can harm other threads in the same process because threads share the same virtual memory space and other resources. For instance, a wild memory write through an uninitialized pointer in one thread can corrupt memory visible to another thread.

An errant process, on the other hand, cannot do so because each process has a copy of the program's memory space.

Copying memory for a new process adds an additional performance overhead relative to creating a new thread. However, the copy is performed only when the memory is changed, so the penalty is minimal if the child process only reads memory.

Threads should be used for programs that need fine-grained parallelism. For example, if a problem can be broken into multiple, nearly identical tasks, threads may be a good choice. Processes should be used for programs that need coarser parallelism.

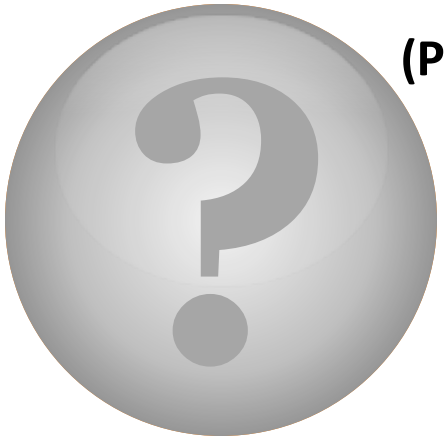
Sharing data among threads is trivial because threads share the same memory. However, great care must be taken to avoid race conditions. Sharing data among processes requires the use of IPC mechanisms. This can be more cumbersome but makes multiple processes less likely to suffer from concurrency bugs.

Share knowledge, Empowering Minds

Communicate & Exchange Ideas

Some “myths”:

(Distributed Systems).Equals(Distributed Computing) == true?



(Parallel System).Equals(Parallel Computing) == true?

(Parallel System == Distributed System) != true?

(Sequential vs. Parallel vs. Concurrent vs. Distributed *Programming*) *? (Different) : (Same)*

Questions & Answers!

But wait…

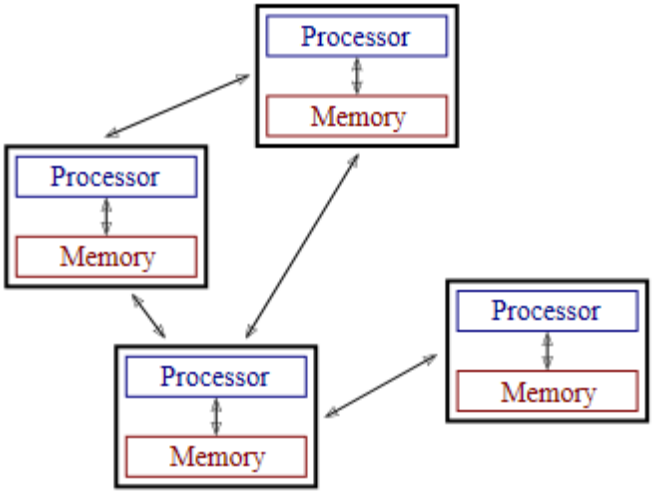
There’s More!

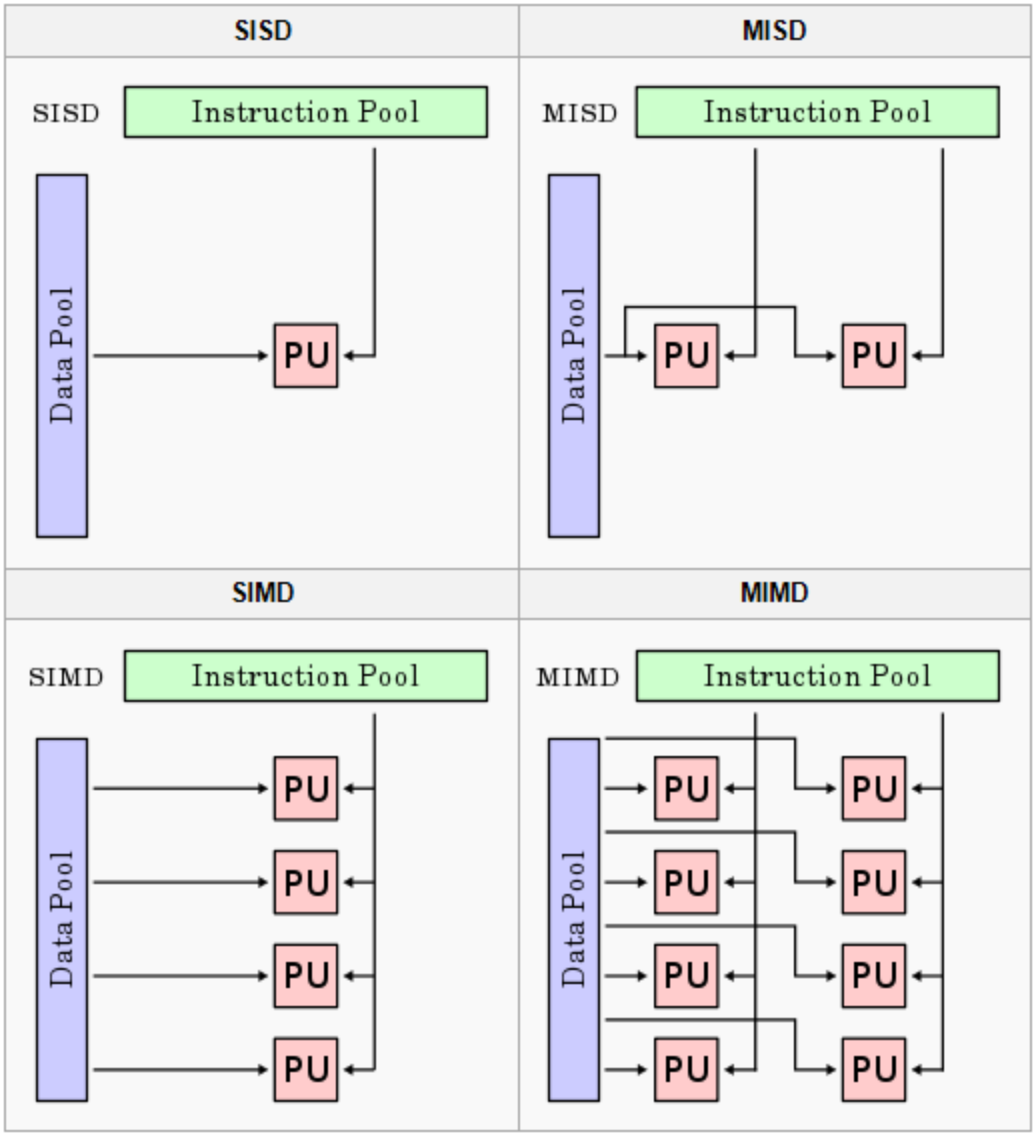
```
if (HTC !=  HPC)
    HTC  (High Throughput Computing) >
    MTC (Many Task  Computing) >
    HPC (High Performance Computing);
```

… Will be continued! - In the next lectures …

Flynn Taxonomy
Parallel vs. Distributed Systems

Parallel vs. Distributed Computing / Algorithms





Where is the picture for:
Distributed System and ***Parallel System***?

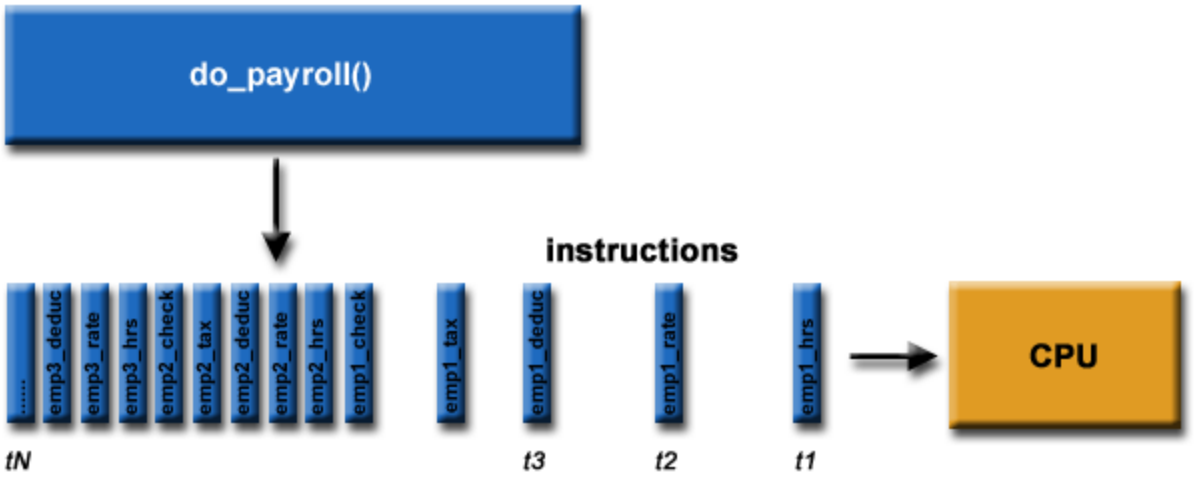
http://en.wikipedia.org/wiki/Distributed_computing

http://en.wikipedia.org/wiki/Flynn's_taxonomy

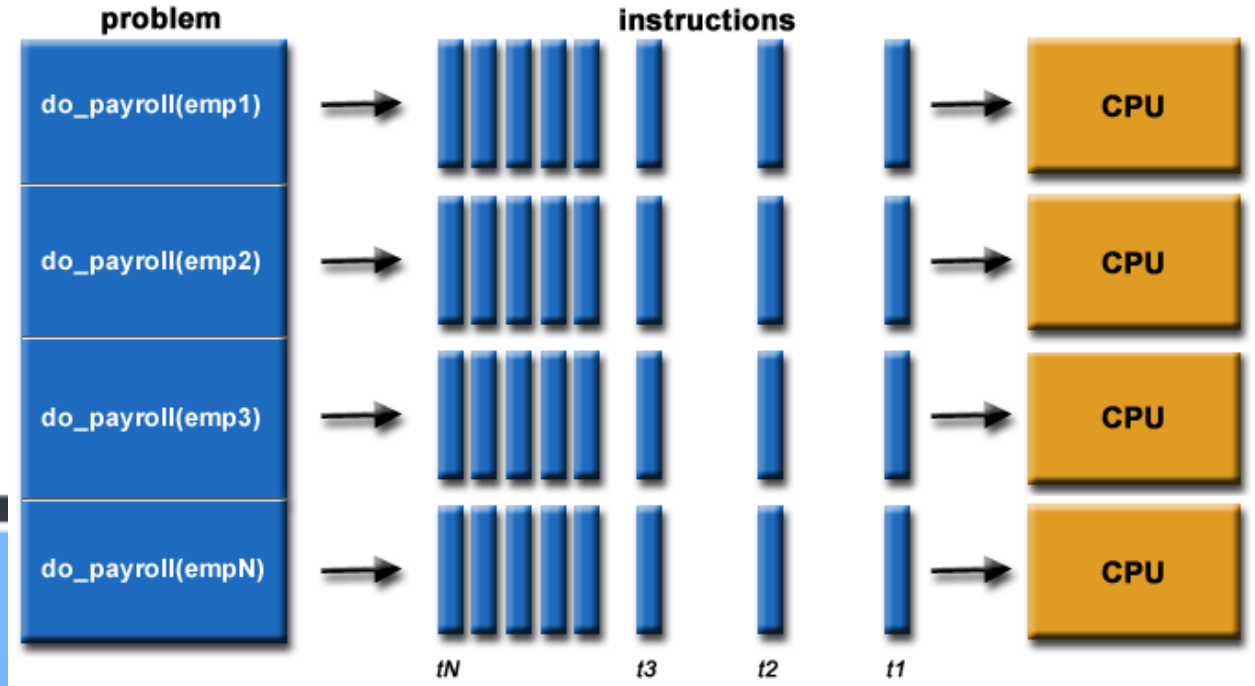
Parallel Computing & Systems - Intro

https://computing.llnl.gov/tutorials/parallel_comp/

Serial Computing



Parallel Computing



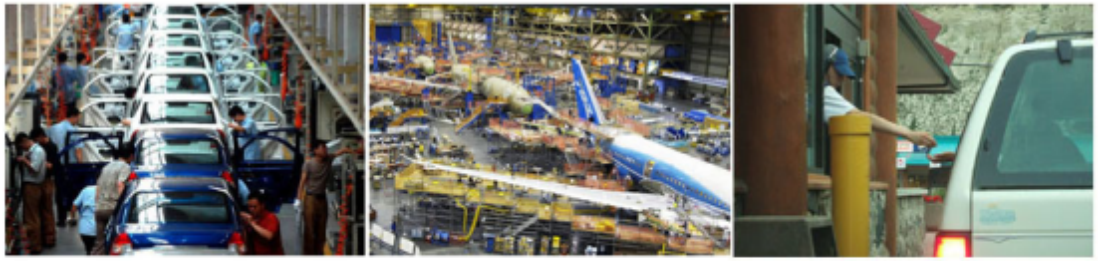
Parallel Computing & Systems - Intro

https://computing.llnl.gov/tutorials/parallel_comp/

The Real World is Massively Parallel

Galaxy Formation | Planetary Movments | Climate Change

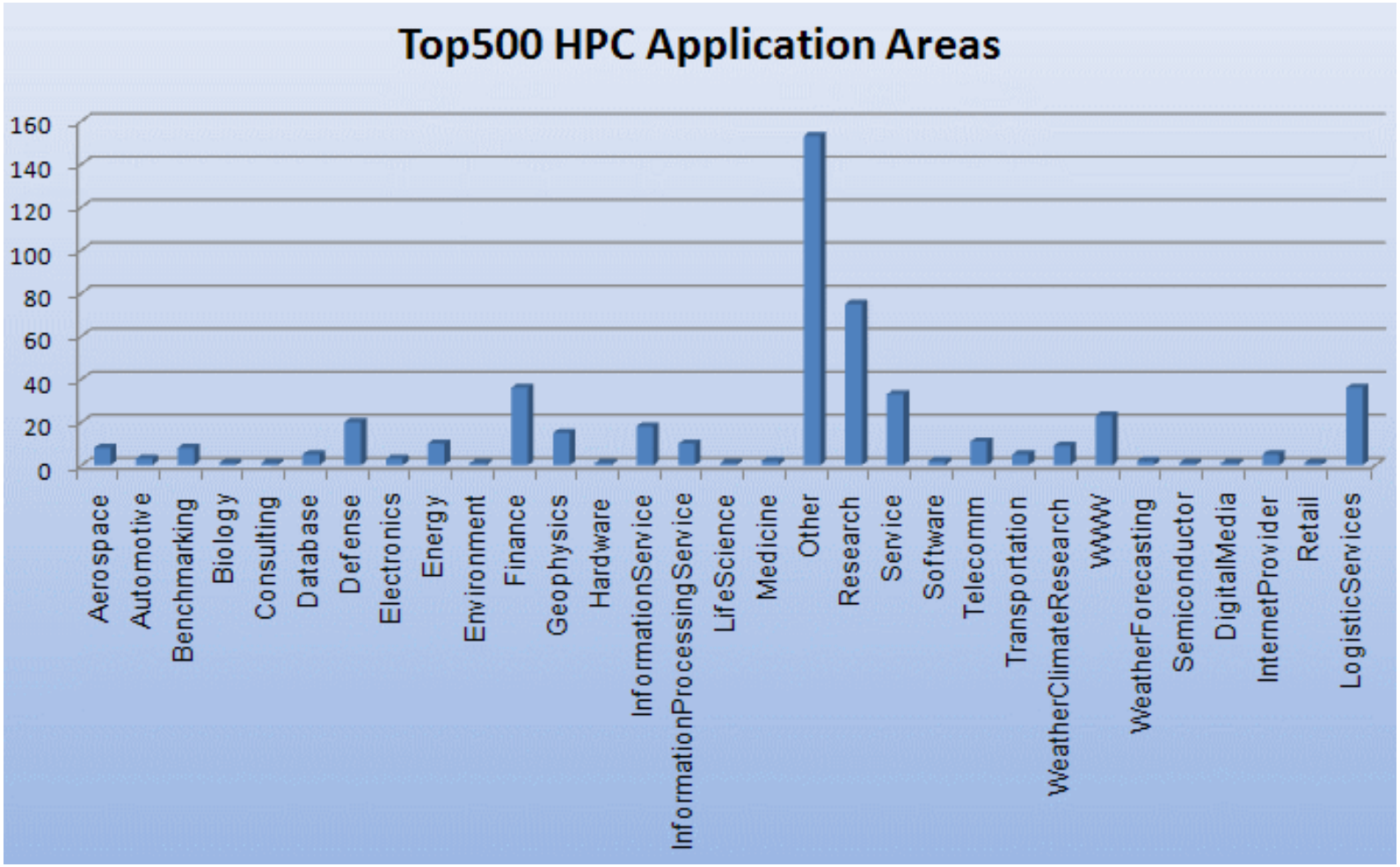
Rush Hour Traffic | Plate Tectonics | Weather



Auto Assembly | Jet Construction | Drive-thru Lunch

Parallel Computing & Systems - Intro

https://computing.llnl.gov/tutorials/parallel_comp/



Intel Accepted Technologies Overview

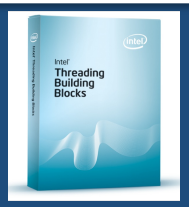
Parallel Programming

C/C++ in Linux with:

MP – Multi-processing Programming (OpenMP)	MPI – Message Passing Interface (OpenMPI)	TBB – Thread Building Blocks (Intel TBB)	OpenCL – Open Computing Language (Intel OpenCL SDK)	Multi-threaded Parallel Computing (Intel Cilk Plus, POSIX Threads, C++'11 Multithread)

OpenMP

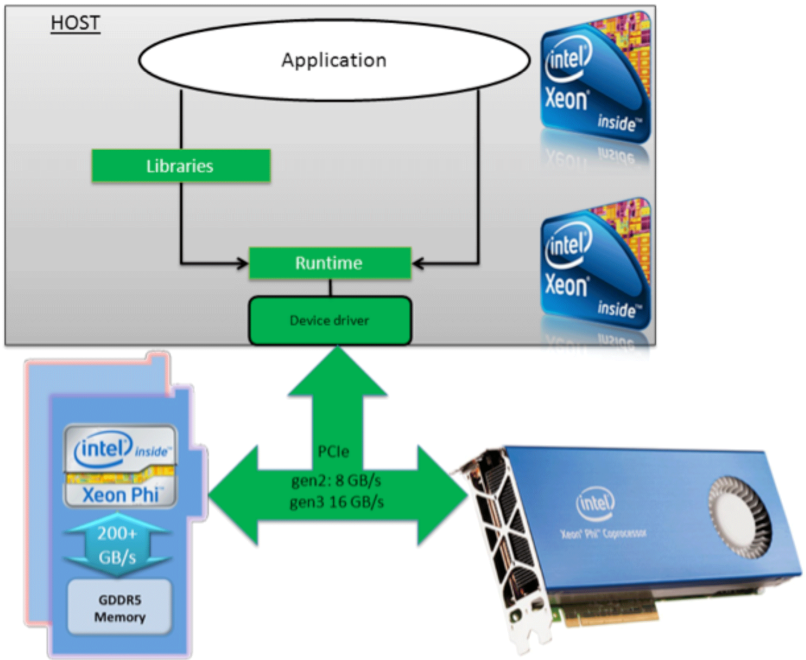
Open MPI: Open Source HPC

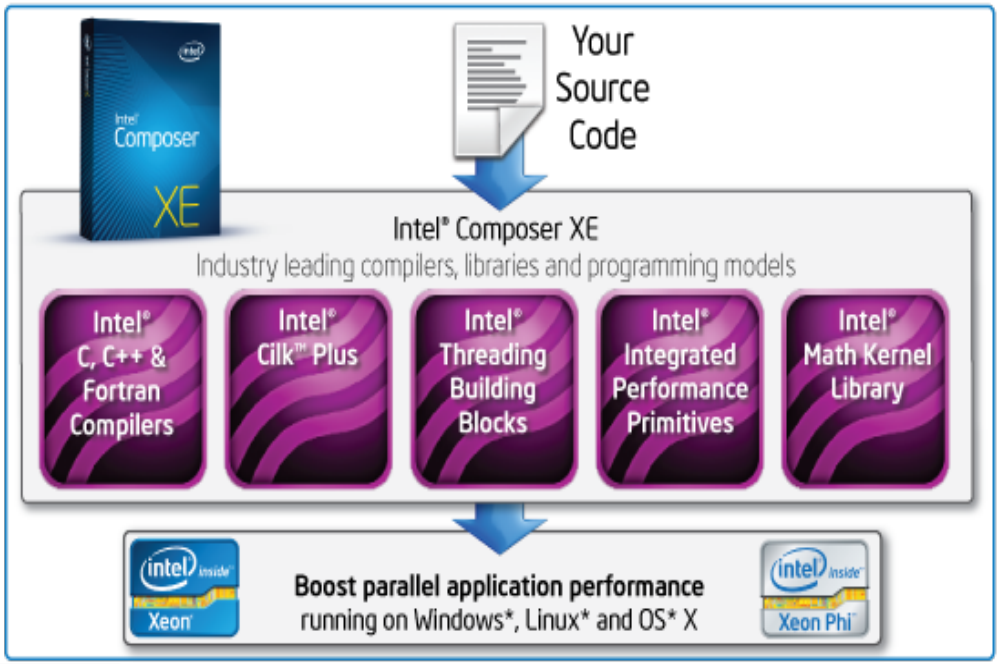


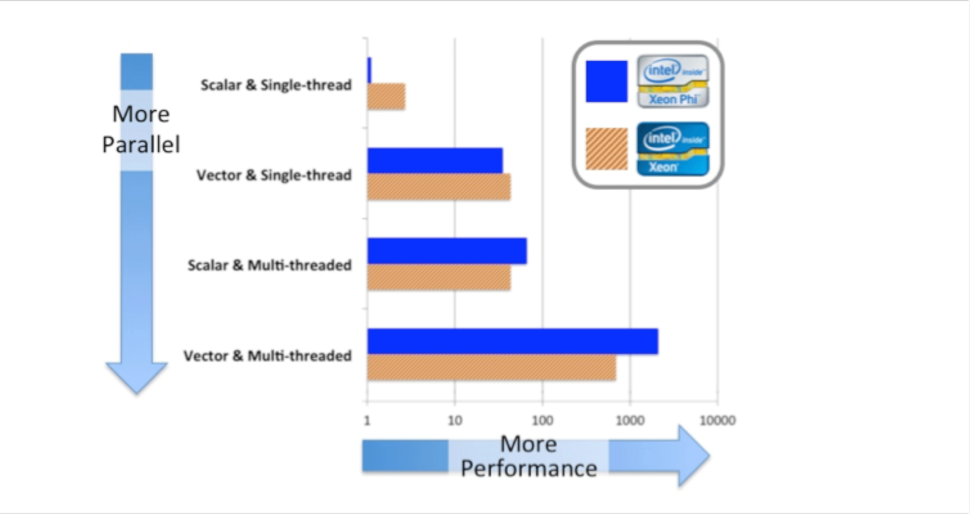
OpenCL

Intel® Cilk™ Plus
C/C++ compiler extension for simplified parallelism

HW & SW Platform







Alternative to:

1. C/C++ Nvidia CUDA

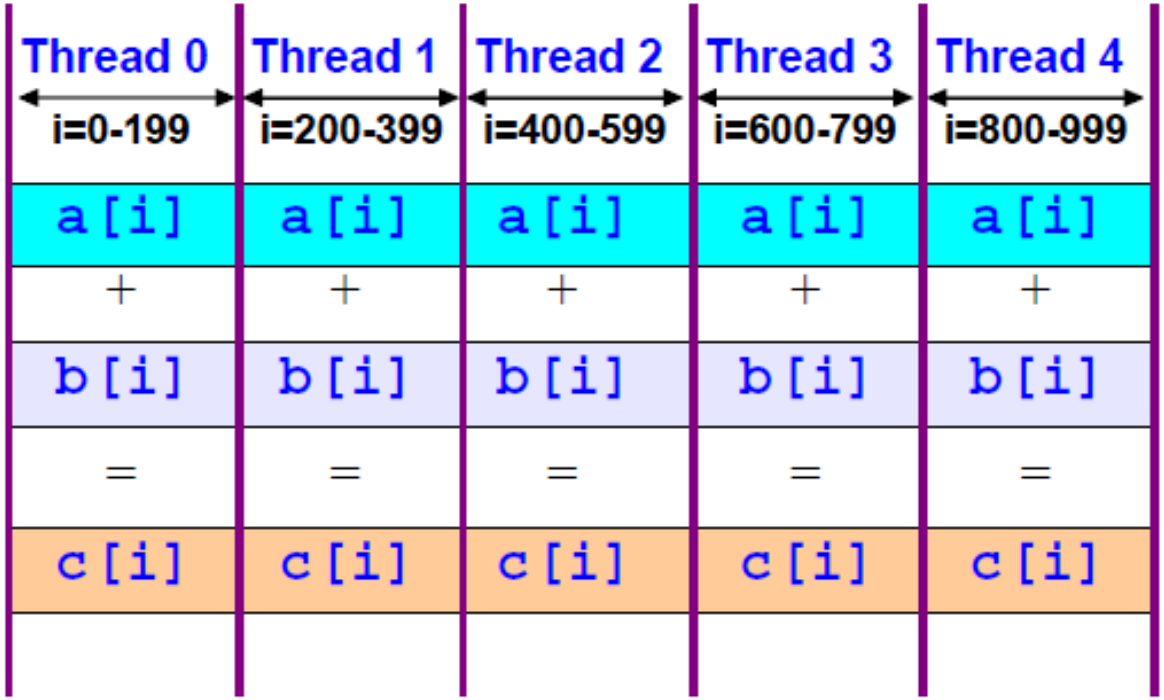
2. C/C++ OpenCL

programming on GPU – video boards

Vector Adding with Parallel Computing

http://ism.ase.ro/ | http://acs.ase.ro/java

Download the VM-Ware virtual machine with Linux Ubuntu x64 - Intel 2 cores, RAM 6048MB, HDD 25 GB



Adding two vectors sample:

- POSIX Threads
- C++'11 Threads
- Java Threads
- C++ (in OpenMP) – OpenMP mini Tutorial

Parallel Programming Restrictions

Hardware : We use a variety of servers and Xeon PHI accelerators (60 cores, 240 threads). Your code has to be portable between the execution environments, but the large workloads will be run on the Xeon PHI only.
Xeon PHI has its own small operating system and libraries, all you can use is : intel compiler, C/C++, OpenMP, TBB, MPI, Cilk (also available on servers). We only accept native Xeon PHI code for this contest, no offloading. *Late edit : we discourage you from using OpenCL, it's not optimal for this problem and without offloading.*

http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

Parallel Programming Restrictions

http://www.drdobbs.com/parallel/programming-intels-xeon-phi-a-jumpstart/240144160

The source code for C/C++ can be compiled without modification by the Intel compiler (icc), to run in the following modes:

- ***Native:*** The entire application runs on the Intel Xeon Phi.
- ***Offload:*** The host processor runs the application and offloads compute intensive code and associated data to the device as specified by the programmer via pragmas in the source code.
- ***Host:*** Run the code as a traditional OpenMP application on the host.

```
# compile for host-based OpenMP
icc -mkl -O3 -no-offload -openmp -Wno-unknown-pragmas -std=c99 -vec-report3
\
   matrix.c -o matrix.omp
# compile for offload mode
icc -mkl -O3 -offload-build -Wno-unknown-pragmas -std=c99 -vec-report3 \
matrix.c -o matrix.off
# compile to run natively on the Xeon Phi
icc -mkl -O3 -mmic -openmp -L /opt/intel/lib/mic -Wno-unknown-pragmas \
-std=c99 -vec-report3 matrix.c -o matrix.mic -liomp5
```

```
1  /* matrix.c (Rob Farber) */
2  #ifndef MIC_DEV
3  #define MIC_DEV 0
4  #endif
5
6  #include <stdio.h>
7  #include <stdlib.h>
8  #include <omp.h>
9  #include <mkl.h>
10 #include <math.h>
11
12 // An OpenMP simple matrix multiply
13 void doMult(int size, float (* restrict A)[size],
14         float (* restrict B)[size], float (* restrict C)[size])
15 {
16 #pragma offload target(mic:MIC_DEV) \
17                 in(A:length(size*size)) in( B:length(size*size))    \
18                 out(C:length(size*size))
19   {
20     // Zero the C matrix
21 #pragma omp parallel for default(none) shared(C,size)
22     for (int i = 0; i < size; ++i)
23       for (int j = 0; j < size; ++j)
24         C[i][j] =0.f;
25
26     // Compute matrix multiplication.
27 #pragma omp parallel for default(none) shared(A,B,C,size)
28     for (int i = 0; i < size; ++i)
29       for (int k = 0; k < size; ++k)
30         for (int j = 0; j < size; ++j)
31           C[i][j] += A[i][k] * B[k][j];
32   }
33 }
```

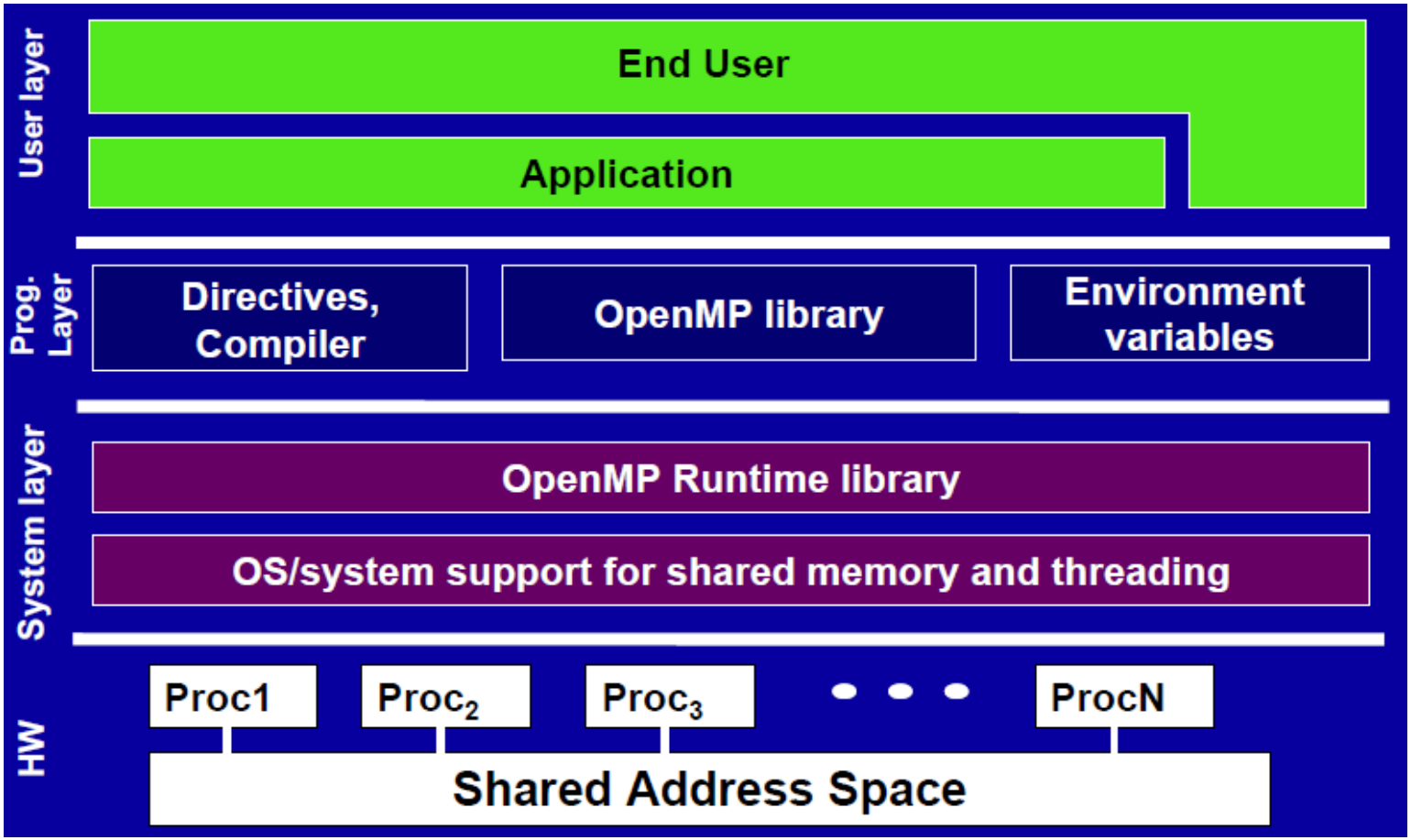
Open MP

Open specifications for Multi Processing

Long version: Open specifications for Multi-Processing via collaborative work between interested parties from the hardware and software industry, government and academia.

- An Application Program Interface (API) that is used to explicitly direct multi-threaded, shared memory parallelism.
- API components:
  - Compiler directives (Compilers that supports: GNU & Intel C/C++ compilers - **gcc/g++** & **icc**)
  - Runtime library routines
  - Environment variables
- Portability
  - API is specified for C/C++ and Fortran
  - Implementations on almost all platforms including Unix/Linux and Windows
- Standardization
  - Jointly defined and endorsed by major computer hardware and software vendors
  - Possibility to become ANSI standard

OpenMP Architecture – Version 3



OpenMP Mini-Tutorial – Version 3

Thread

- A **process** is an instance of a computer program that is being executed. It contains the program code and its current activity.
- A **thread of execution** is the smallest unit of processing that can be scheduled by an operating system.
- Differences between threads and processes:
  - A thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory. The threads of a process share the latter's instructions (code) and its context (values that its variables reference at any given moment).
  - Different processes do not share these resources.

http://en.wikipedia.org/wiki/Process_(computing) |

Process

- A process contains all the information needed to execute the program:
  - Process ID
  - Program code
  - Data on run time stack
  - Global data
  - Data on heap

Each process has its own address space.

- In multitasking, processes are given time slices in a round robin fashion.
  - If computer resources are assigned to another process, the status of the present process has to be saved, in order that the execution of the suspended process can be resumed at a later time.

OpenMP - Mini-Tutorial – Version 3

Threads Features:

- Thread model is an extension of the process model.
- Each process consists of multiple independent instruction streams (or threads) that are assigned computer resources by some scheduling procedure.
- Threads of a process share the address space of this process.
  - Global variables and all dynamically allocated data objects are accessible by all threads of a process.
- Each thread has its own run time stack, register, program counter.
- Threads can communicate by reading/writing variables in the common address space.

Thread operations include thread creation, termination, synchronization (joins, blocking), scheduling, data management and process interaction.

A thread does not maintain a list of created threads, nor does it know the thread that created it.

All threads within a process share the same address space.

Threads in the same process share:

- Process instructions
- Most data
- open files (descriptors)
- signals and signal handlers
- current working directory
- User and group id

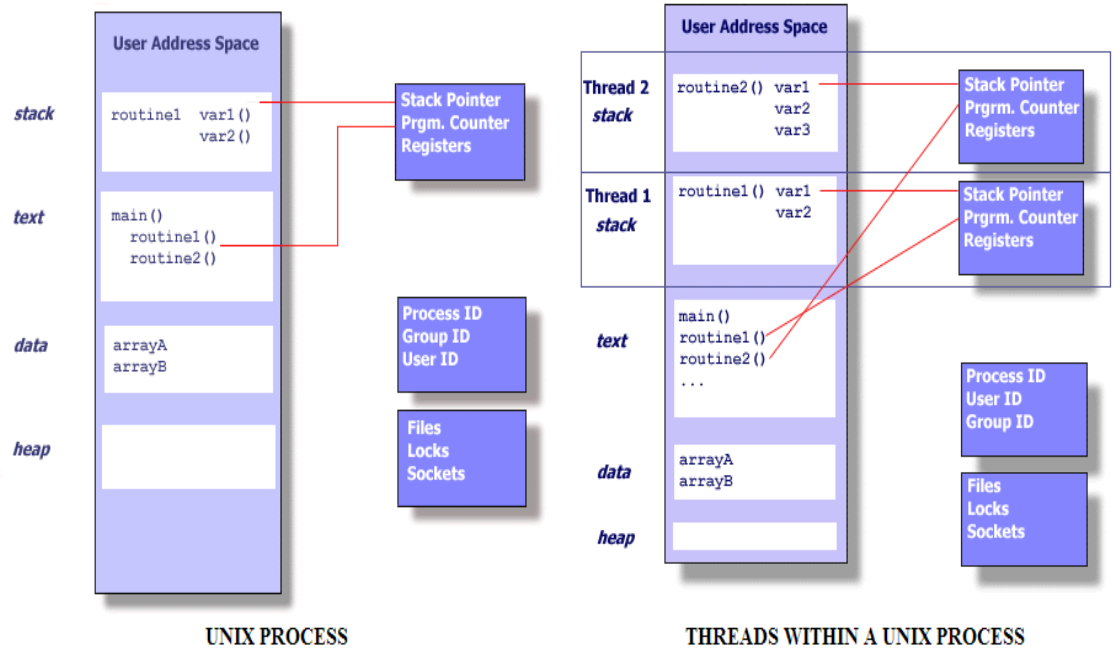
Each thread has a unique:

- Thread ID
- set of registers, stack pointer
- stack for local variables, return addresses
- signal mask
- priority
- Return value: errno

pthread functions return "0" if OK.

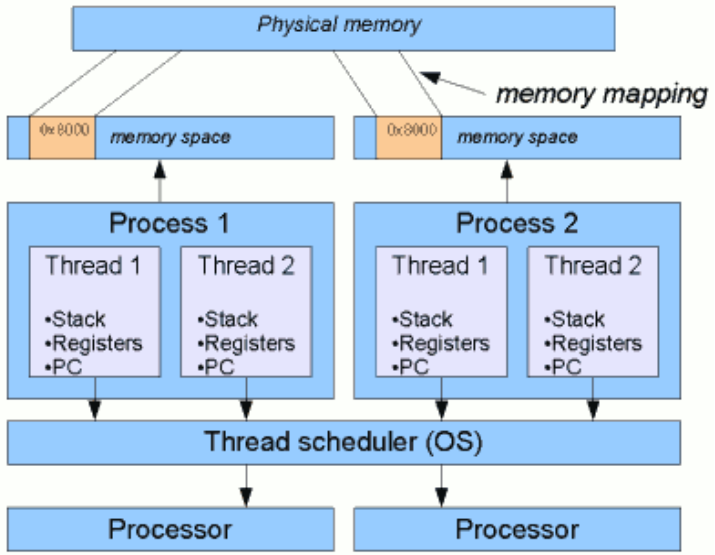
OpenMP - Mini-Tutorial – Version 3

Multi-threading vs. Multi-process development in UNIX/Linux:



UNIX PROCESS

THREADS WITHIN A UNIX PROCESS



https://computing.llnl.gov/tutorials/pthreads/

http://www.javamex.com/tutorials/threads/how_threads_work.shtml

OpenMP - Mini-Tutorial – Version 3

OpenMP Programming Model:

- Shared memory, thread-based parallelism
  - OpenMP is based on the existence of multiple threads in the shared memory programming paradigm.
  - A shared memory process consists of multiple threads.
- Explicit Parallelism
  - Programmer has full control over parallelization. OpenMP is not an automatic parallel programming model.
- Compiler directive based
  - Most OpenMP parallelism is specified through the use of compiler directives which are embedded in the source code.

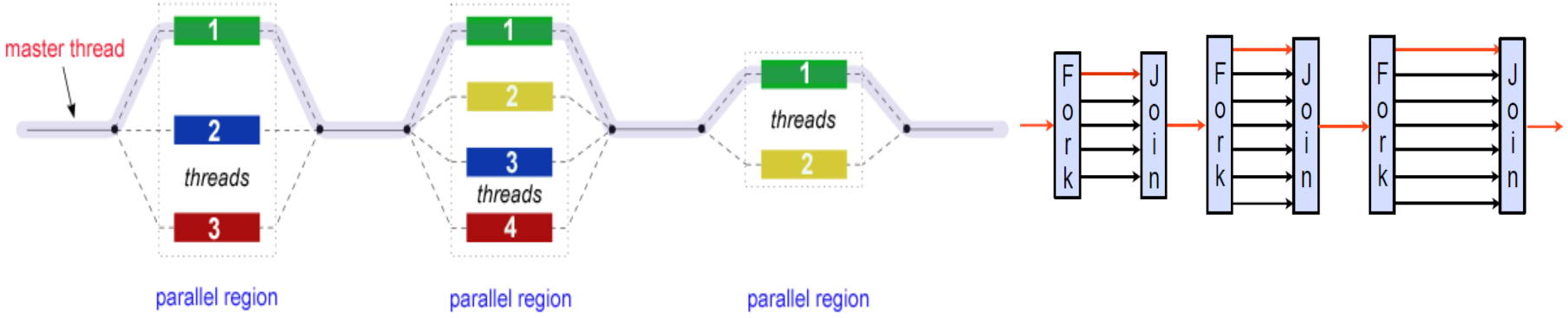
OpenMP is NOT:

- Necessarily implemented identically by all vendors
- Meant for distributed-memory parallel systems (it is designed for shared address spaced machines)
- Guaranteed to make the most efficient use of shared memory
- Required to check for data dependencies, data conflicts, race conditions, or deadlocks
- Required to check for code sequences
- Meant to cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization
- Designed to guarantee that input or output to the same file is synchronous when executed in parallel.

OpenMP - Mini-Tutorial – Version 3

OpenMP - Fork-Join Parallelism Model:

- OpenMP program begin as a single process: the *master thread (in pictures in red/grey)*. The master thread executes sequentially until the first *parallel region* construct is encountered.
- When a parallel region is encountered, master thread
  - Create a group of threads by **FORK**.
  - Becomes the master of this group of threads, and is assigned the thread id 0 within the group.
- The statement in the program that are enclosed by the *parallel region* construct are then executed in parallel among these threads.
- **JOIN**: When the threads complete executing the statement in the *parallel region* construct, they synchronize and terminate, leaving only the master thread.



OpenMP - Mini-Tutorial – Version 3

I/O

- OpenMP does not specify parallel I/O.
- It is up to the programmer to ensure that I/O is conducted correctly within the context of a multi-threaded program.

Memory Model

- Threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
- When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is updated by all threads as needed.

```
//OpenMP Code Structure
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main()
{
  #pragma omp parallel
  {
    int ID = omp_get_thread_num();
    printf("Hello (%d)\n", ID);
    printf(" world (%d)\n", ID);
  }
}
```

Set # of threads for OpenMP:

- In csh:

```
setenv OMP_NUM_THREAD 8
```

- In bash:

```
set OMP_NUM_THREAD=8
export $OMP_NUM_THREAD
```

Compile: g++ -fopenmp hello.c

Run: ./a.out

Compiler / Platform	Compiler	Flag
Intel Linux Opteron/Xeon	icc icpc ifort	-openmp
PGI Linux Opteron/Xeon	pgcc pgCC pgf77 pgf90	-mp
GNU Linux Opteron/Xeon IBM Blue Gene	gcc g++ g77 gfortran	-fopenmp

OpenMP - Mini-Tutorial – Version 3

OpenMP Core Syntax

```
#include “omp.h”
void main ()
{
      int var1, var2, var3;
      // 1. Serial code
      . . .
      // 2. Beginning of parallel section.
      // Fork a team of threads. Specify variable scoping
      #pragma omp parallel private(var1, var2) shared(var3)
      {
      // 3. Parallel section executed by all threads
      . . .
      // 4. All threads join master thread and disband
      }
      // 5. Resume serial code . . .
}
```

OpenMP C/C++ Directive Format

- OpenMP directive forms
  - C/C++ use compiler directives
- Prefix: #pragma omp …
  - A directive consists of a directive name followed by *clauses*

Example:

```
#pragma omp parallel default (shared)
private (var1, var2)
```

OpenMP Directive Format - General Rules:

- Case sensitive
- Only one directive-name may be specified per directive
- Each directive applies to at most one succeeding statement, which must be a structured block.
- Long directive lines can be “continued” on succeeding lines by escaping the newline character with a backslash “\” at the end of a directive line.

OpenMP - Mini-Tutorial – Version 3

OpenMP *parallel* Region Directive

#pragma omp parallel [*clause list*]

Typical clauses in [*clause list*]

- Conditional parallelization
  - if (scalar expression)
    - Determine whether the parallel construct creates threads
- Degree of concurrency
  - num_threads (integer expression)
    - number of threads to create
- Date Scoping
  - private (variable list)
    - Specifies variables local to each thread
  - firstprivate (variable list)
    - Similar to the private
    - Private variables are initialized to variable value before the parallel directive
  - shared (variable list)
    - Specifies variables that are shared among all the threads
  - default (data scoping specifier)
    - Default data scoping specifier may be shared or none

Example:

```
#pragma omp parallel if (is_parallel == 1)
num_threads(8) shared (var_b) private (var_a)
firstprivate (var_c) default (none)
{
/* structured block */
}
```

- if (is_parallel == 1) num_threads(8)
  - If the value of the variable is_parallel is one, create 8 threads
- shared (var_b)
  - Each thread shares a single copy of variable var_b
- private (var_a) firstprivate (var_c)
  - Each thread gets private copies of variable var_a and var_c
  - Each private copy of var_c is initialized with the value of var_c in main thread when the parallel directive is encountered
- default (none)
  - Default state of a variable is specified as none (rather than shared)
  - Signals error if not all variables are specified as shared or private

Number of Threads:

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  1. Evaluation of the if clause
  2. Setting of the num_threads() clause
  3. Use of the omp_set_num_threads() library function
  4. Setting of the OMP_NUM_THREAD environment variable
  5. Implementation default – usually the number of cores on a node

- Threads are numbered from 0 (master thread) to N-1

OpenMP - Mini-Tutorial – Version 3

Thread Creation: Parallel Region Example - **Create threads with the parallel construct**

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main()
{
   int nthreads, tid;
   #pragma omp parallel num_threads(4) private(tid)
   {
      tid = omp_get_thread_num();
      printf("Hello world from (%d)\n", tid);
      if(tid == 0)
      {
         nthreads = omp_get_num_threads();
         printf("number of threads = %d\n", nthreads);
      }
   } // all threads join master thread and terminates
}
```

Clause to request threads

Each thread executes a copy of the code within the structured block

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"

int main(){
   int nthreads,  A[100] , tid;
   // fork a group of threads with each thread having a private tid variable
   omp_set_num_threads(4);
   #pragma omp parallel private (tid)
   {
      tid = omp_get_thread_num();
      foo(tid, A);
   } // all threads join master thread and terminates
}
```

A single copy of A[] is shared between all threads

Work-Sharing Construct:

- A parallel construct by itself creates a "Single Program/Instruction Multiple Data" (SIMD) program, i.e., each thread executes the same code.
- Work-sharing is to split up pathways through the code between threads within a team.
  - Loop construct (for/do)
  - Sections/section constructs
  - Single construct
- Within the scope of a parallel directive, work-sharing directives allow concurrency between iterations or tasks
- ***Work-sharing constructs do not create new threads.***
- A work-sharing construct must be enclosed dynamically within a parallel region in order for the directive to execute in parallel.
- ***Work-sharing constructs must be encountered by all members of a team or none at all.***
- Two directives to be presented
  - – do/for: concurrent loop iterations
  - – sections: concurrent tasks

OpenMP - Mini-Tutorial – Version 3

Work-Sharing do/for Directive

do/for:

- Shares iterations of a loop across the group
- *Represents a "data parallelism"*.

for directive partitions parallel iterations across threads
do is the analogous directive in Fortran

- Usage:

#pragma omp for [clause list]
/* for loop */

- Implicit barrier at end of for loop

```
#include <stdlib.h>
#include <stdio.h>
#include "omp.h"
void main()
{
  int nthreads, tid;

  omp_set_num_threads(3);

  #pragma omp parallel private(tid)
  {
     int i;
     tid = omp_get_thread_num();
     printf("Hello world from (%d)\n", tid);
     #pragma omp for
     for(i = 0; i <=4; i++)
     {
       printf("Iteration %d by %d\n", i, tid);
     }
  } // all threads join master thread and terminates
}
```

OpenMP - Mini-Tutorial – Version 3

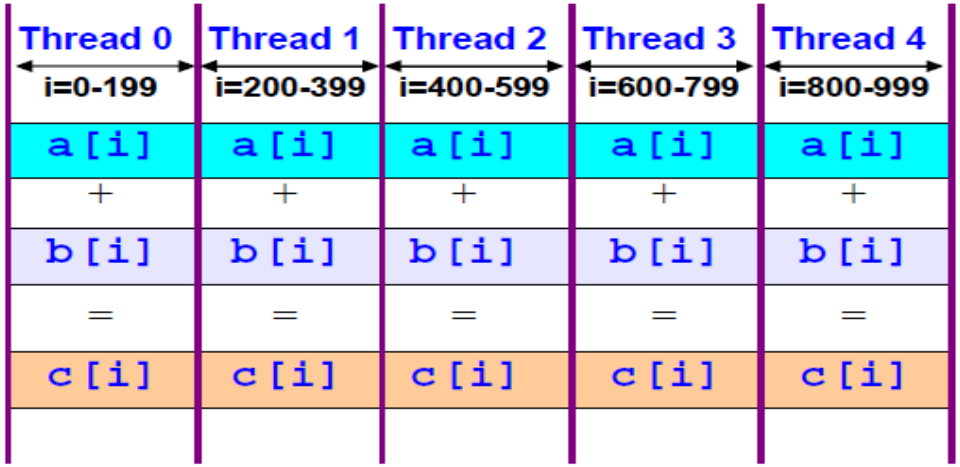
```
//Sequential code to add two vectors:
for(i=0;i<N;i++) {
  c[i] = b[i] + a[i];
}
```

```
//OpenMP implementation 1 (not desired):
#pragma omp parallel
{
   int id, i, Nthrds, istart, iend;
   id = omp_get_thread_num();
   Nthrds = omp_get_num_threads();
   istart = id*N/Nthrds;
   iend = (id+1)*N/Nthrds;

   if(id == Nthrds-1) iend = N;
   for(I = istart; i<iend; i++) {
      c[i] = b[i]+a[i];
   }
}
```

```
//A worksharing for construct to add vectors:
#pragma omp parallel
{
   #pragma omp for
   {
      for(i=0; i<N; i++) { c[i]=b[i]+a[i]; }
   }
}
```

```
//A worksharing for construct to add vectors:
#pragma omp parallel for
for(i=0; i<N; i++) { c[i]=b[i]+a[i]; }
```



OpenMP - Mini-Tutorial – Version 3

C/C++ for Directive Syntax:

```
#pragma omp for [clause list]
        schedule (type [,chunk])
        ordered
        private (variable list)
        firstprivate (variable list)
        shared (variable list)
        reduction (operator: variable list)
        collapse (n)
        nowait
/* for_loop */
```

For Directive Restrictions

For the "*for* loop" that follows the *for* directive:

- It must not have a break statement
- The loop control variable must be an integer
- The initialization expression of the "*for* loop" must be an integer assignment.
- The logical expression must be one of <,≤,>,≥
- The increment expression must have integer increments or decrements only.

How to combine values into a single accumulation variable (avg)?

```
//Sequential code to do average value from an
array-vector:

{
   double avg = 0.0, A[MAX];
   int i;
   …
   for(i =0; i<MAX; i++) {
      avg += a[i];
   }

   avg /= MAX;
}
```

OpenMP - Mini-Tutorial – Version 3

Reduction Clause

- *Reduction (operator: variable list)*:
  specifies how to combine local copies of a variable in different threads into a single copy at the master when threads exit. Variables in *variable list* are implicitly private to threads.
- Operators used in Reduction Clause: +, *, -, &, |, ^, &&, and ||
- Usage Sample:

```
#pragma omp parallel reduction(+: sums) num_threads(4)
{
  /* compute local sums in each thread */
}
/* sums here contains sum of all local instances of sum */
```

Reduction Operators/Initial-Values in C/C++ OpenMP

Operator	Initial Value	Operator	Initial Value
+	0	|	0
*	1	^	0
-	0	&&	1
&	~0	||	0

Reduction in OpenMP for:

Inside a parallel or a work-sharing construct:

- A local copy of each list variable is made and initialized depending on *operator* (e.g. 0 for "+")
- Compiler finds standard reduction expressions containing *operator* and uses it to update the local copy.
- Local copies are reduced into a single value and combined with the original global value when returns to the master thread.

```
//A work-sharing for average value from a vector:
{
   double  avg = 0.0, A[MAX];
   int i;
   …
   #pragma omp parallel for reduction (+:avg)
   for(i =0; i<MAX; i++) {avg += a[i];}

   avg /= MAX;
}
```

OpenMP - Mini-Tutorial – Version 3

Matrix-Vector Multiplication

```
#pragma omp parallel default (none) \
shared (a, b, c, m,n) private(i,j,sum) num_threads(4)
for(i=0; i < m; i++)
{
        sum = 0.0;
        for(j=0; j < n; j++)
                sum += b[i][j]*c[j];

        a[i] =sum;
}
```

```
for (i=0,1,2,3,4)
 i = 0
sum =    b[i=0][j]*c[j]
   a[0] = sum
 i = 1
sum =    b[i=1][j]*c[j]
   a[1] = sum
```

Thread 0,

```
for (i=5,6,7,8,9)
 i = 5
sum =    b[i=5][j]*c[j]
   a[5] = sum
 i = 6
sum =    b[i=6][j]*c[j]
   a[6] = sum
```

Thread 1, ...etc...

OpenMP - Mini-Tutorial – Version 3

Matrix-Vector | Matrix-Matrix Multiplication

schedule clause

- Describe how iterations of the loop are divided among the threads in the group. The default schedule is implementation dependent.
- Usage: schedule (scheduling_class[, parameter]).

– ***static -*** Loop iterations are divided into pieces of size chunk and then statically assigned to threads. If chunk is not specified, the iteration are evenly (if possible) divided contiguously among the threads.
– ***dynamic*** - Loop iterations are divided into pieces of size chunk and then dynamically assigned to threads. When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.
– ***guided*** - For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k(k>1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.
– ***runtime*** - The scheduling decision is deferred until runtime by the environment variable OMP_SCHEDULE. It is illegal to specify a chunk size for this clause
– ***auto*** - The scheduling decision is made by the compiler and/or runtime system.

Static scheduling - 16 iterations, 4 threads:

Thread	0	1	2	3
*no chunk**	1-4	5-8	9-12	13-16
chunk = 2	1-2 9-10	3-4 11-12	5-6 13-14	7-8 15-16

```
// Static schedule maps iterations to threads at compile time
// static scheduling of matrix multiplication loops
#pragma omp parallel default (private) \
shared (a, b, c, dim) num_threads(4)
#pragma omp for schedule(static)
for(i=0; i < dim; i++)
{
  for(j=0; j < dim; j++)
  {
    c[i][j] = 0.0;
    for(k=0; j < dim; k++)
      c[i][j] += a[i][k]*b[k][j];
  }
}
```

Thanks!

DAD – Distributed Application Development
End of Lecture 1