Threads

- Operating Systems Principles & Practice
  - Volume II, chapter 5
- Unix Internals
  - Chapter, 3
- Multithreaded Programming Guide, Sun
  - Chapters 1 and 2

Concurrency

- In the real world different activities often proceed at the same time.
  - But inter-related (on activity affect the others)
  - They are concurrent
- Computers are concurrent
  - Multiple resources
    - dozen processors, 10 disks, and 4 network interfaces; a workstation might have a dozen active I/O devices including a screen, keyboard, mouse, camera, microphone, speaker, wireless network interface, wired network interface, printer, scanner, and disk drive
  - Multiple CPUs / Cores
  - Multiple users/applications

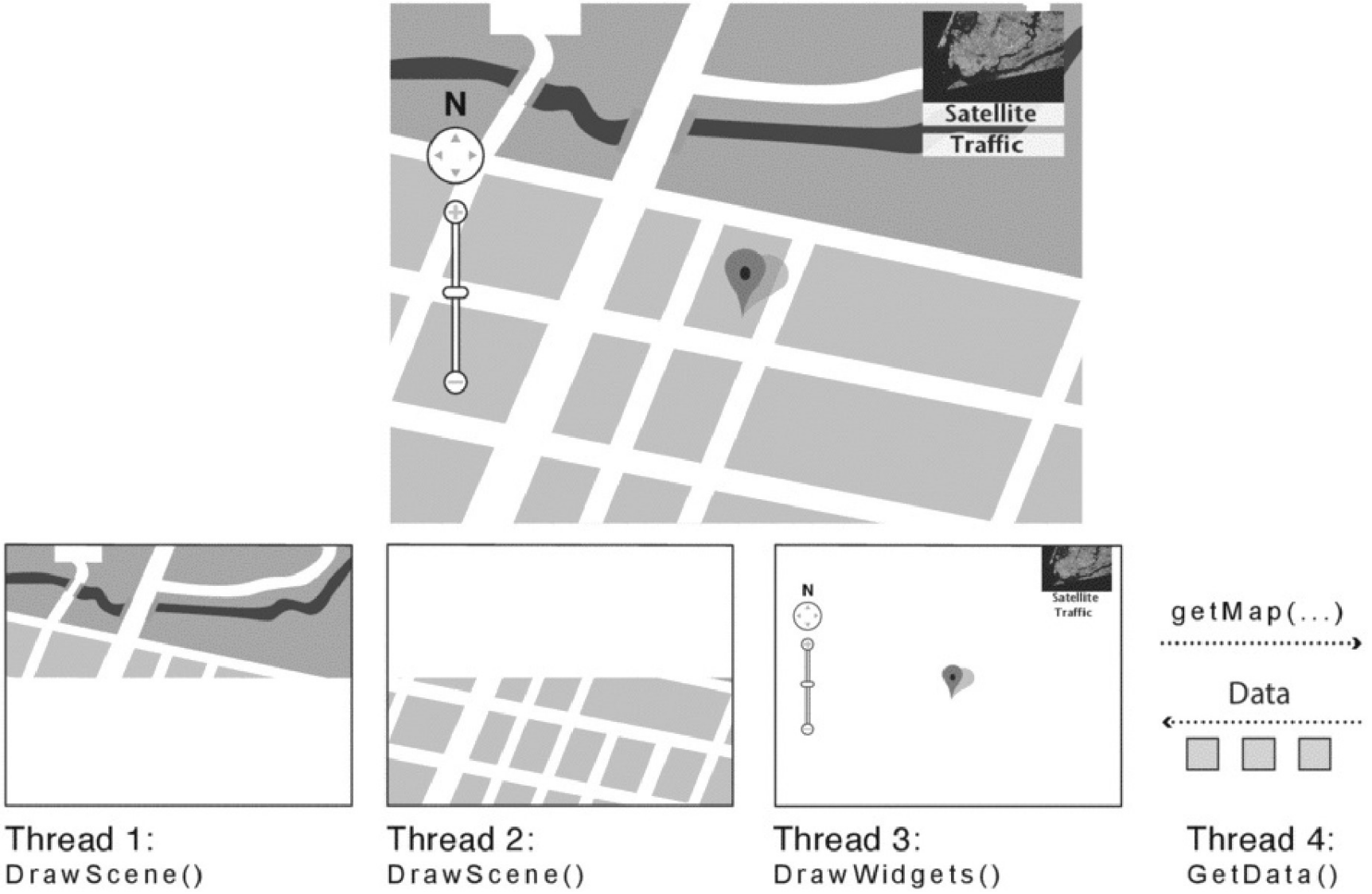
Concurrency

- Relevant to the programmer
  - Network services need to be able to handle multiple requests from their client
  - Most applications today have user interfaces while simultaneously executing application logic
  - Parallel programs need to be able to map work onto multiple processors
  - Need to mask the latency of disk and network operations

Threads

- A Thread is an independent stream of instructions that can be schedule to run as such by the OS.
- Think of a thread as a “procedure” that runs independently from its main program.
- Multi-threaded programs are where several procedures are able to be scheduled to run simultaneously and/or independently by the OS.
- A Thread exists within a process and uses the process resources.

Thread


N
Satellite
Traffic
N
Satellite
Traffic
getMap(...)
Data
Thread 1:
DrawScene()
Thread 2:
DrawScene()
Thread 3:
DrawWidgets()
Thread 4:
GetData()

- Threads only duplicate the essential resources it needs to be independently schedulable.
- A thread will die if the parent process dies.
- A thread is “lightweight” because most of the overhead has already been accomplished through the creation of the process.

  - Private
    - Processor register
    - Stack
- Shared
  - Memory
  - Resources

Why Use Threads

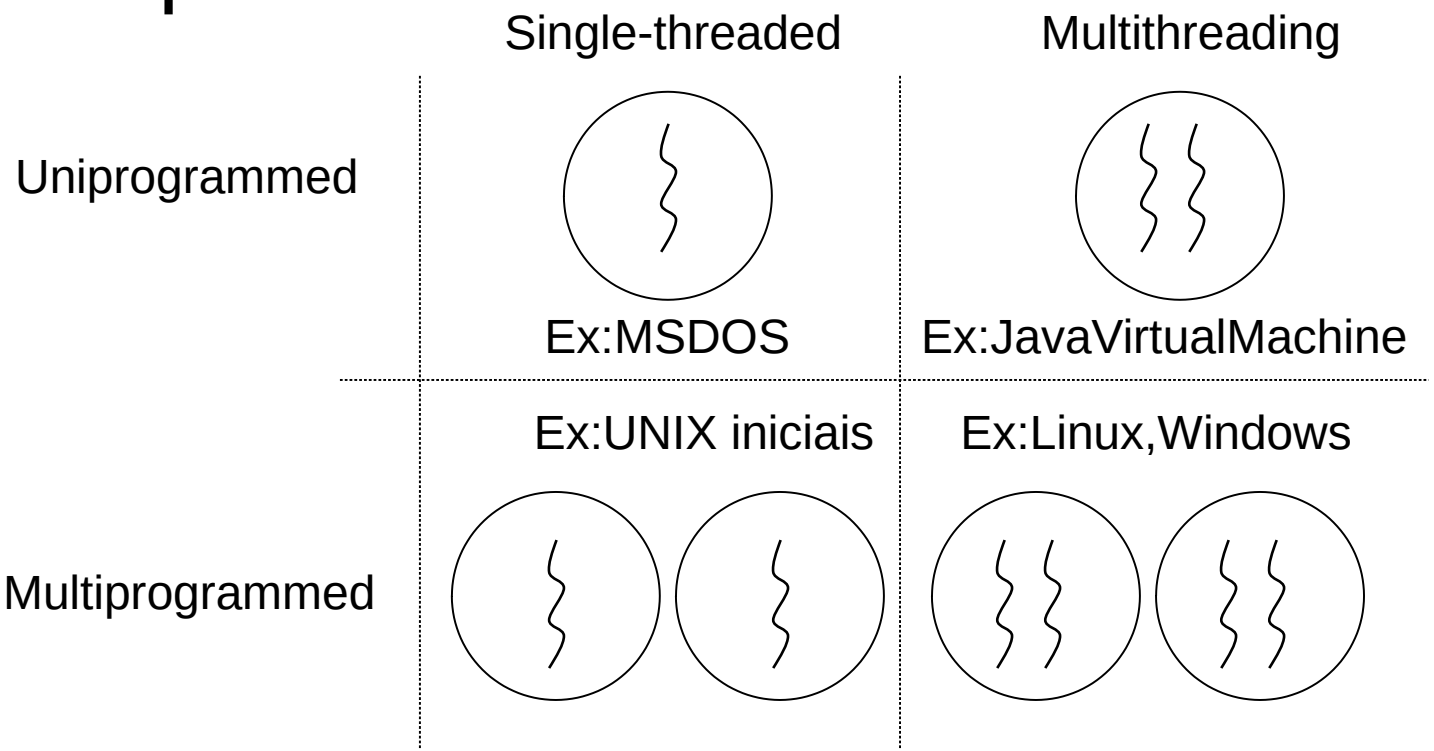
- The primary motivation behind Pthreads
  - is improving program performance.
  - Easing development
- Can be created with much less OS overhead.
- Needs fewer system resources to run.

When to use

- Message multiplexing in the same channel
  - Each thread handles a type of messages
  - Releases main thread to handle other messages
- Multiple channels
  - Each thread handles a channel
- Synchronized wait / Event notification
  - One thread is blocked waiting for an event
- Shared memory between parallel executions

Multithreadead systems

- A system that allows processes with multiple threads
  - Are called multithreaded
- OS can be split into:

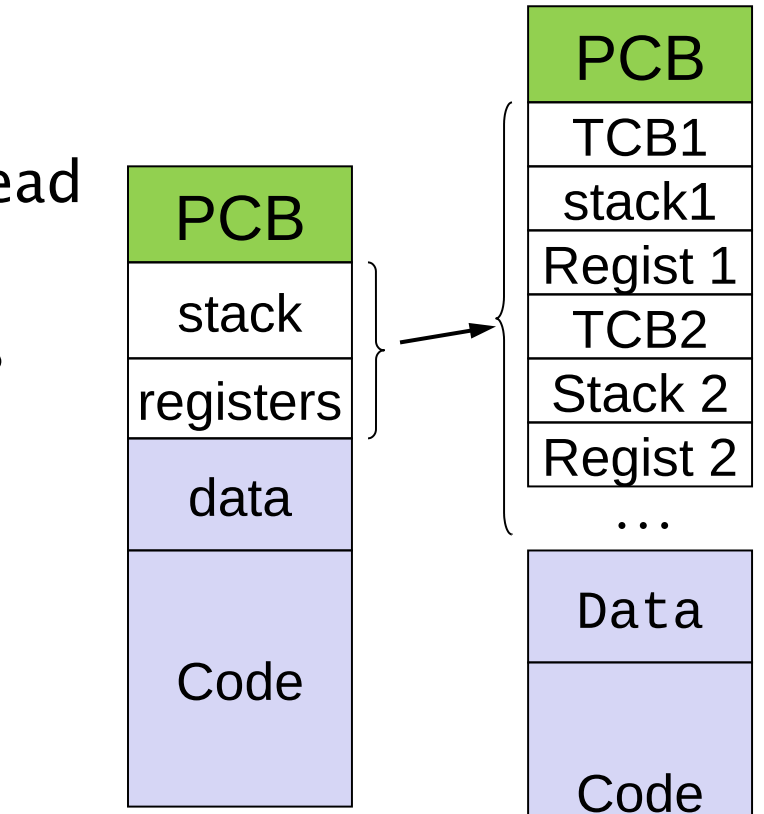


Thread

- Execution thread
  - Execution unit
  - private
    - Processor register
    - Stack
  - Shared (inside the sme process)
    - Memory
    - resources
- One process contains one thread when started

Process structure

- With threads the process imag changes:
  - Each thread contains a local TCBThread Control Block.
- In a process each thread contains its own stack
  - Local variables are local to each thread
- Threads share
  - Code
  - Global variables
  - Resources (FILES, IPC)



Process vs threads

Processes	Threads
Hierarquical struture • Parent • Child	Flat stucture • All siblings
Independent data space • No shared variables	Shared data space Shared DS Shared heap
Creation os expensive	Creation is cheap • More than 20 x faster

PLATFORM	fork()			pthread_create()		
	REAL	USER	SYSTEM	REAL	USER	SYSTEM
AMD 2.4 GHz Opteron (8cpus/node)	41.07	60.08	9.01	0.66	0.19	0.43
IBM 1.9 GHz POWER5 p5-575 (8cpus/node)	64.24	30.78	27.68	1.75	0.69	1.1
IBM 1.5 GHz POWER4 (8cpus/node)	104.05	48.64	47.21	2.01	1	1.52
INTEL 2.4 GHz Xeon (2 cpus/node)	54.95	1.54	20.78	1.64	0.67	0.9
INTEL 1.4 GHz Itanium2 (4 cpus/node)	54.54	1.07	22.22	2.03	1.26	0.67

Multithreaded Programs

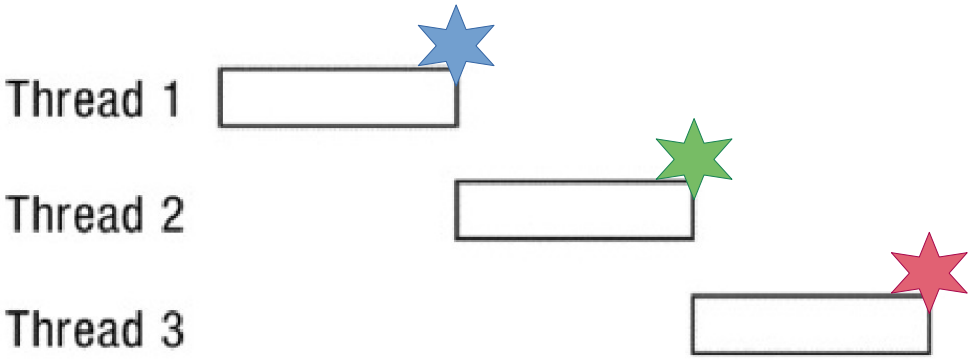
- Best used with programs that can be organized
  - into discrete, independent tasks which can execute concurrently.
- Threads can be
  - interchanged,
  - interleaved
  - overlapped

Multithreaded Programs

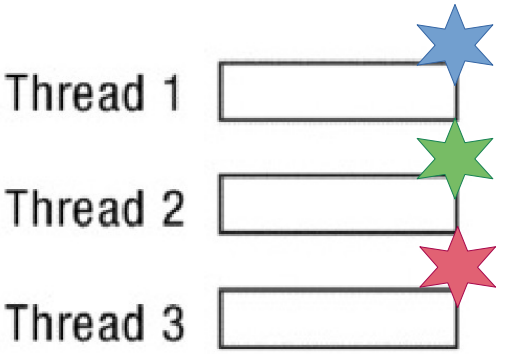
Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

Multithreaded Programs

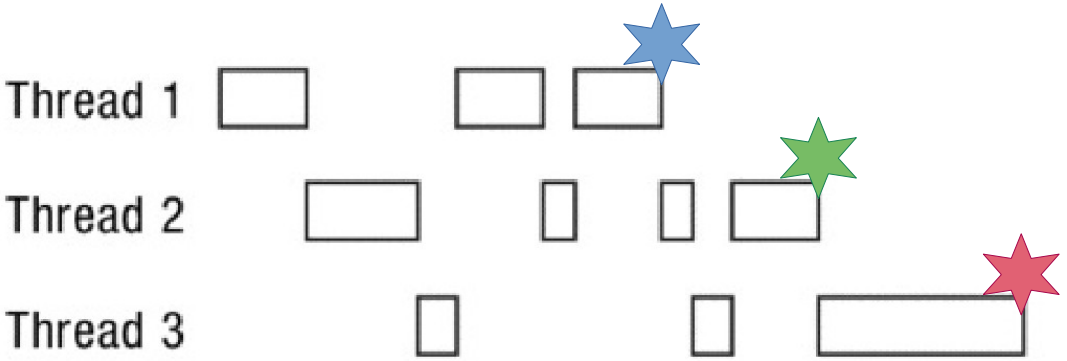
One Execution



Another Execution



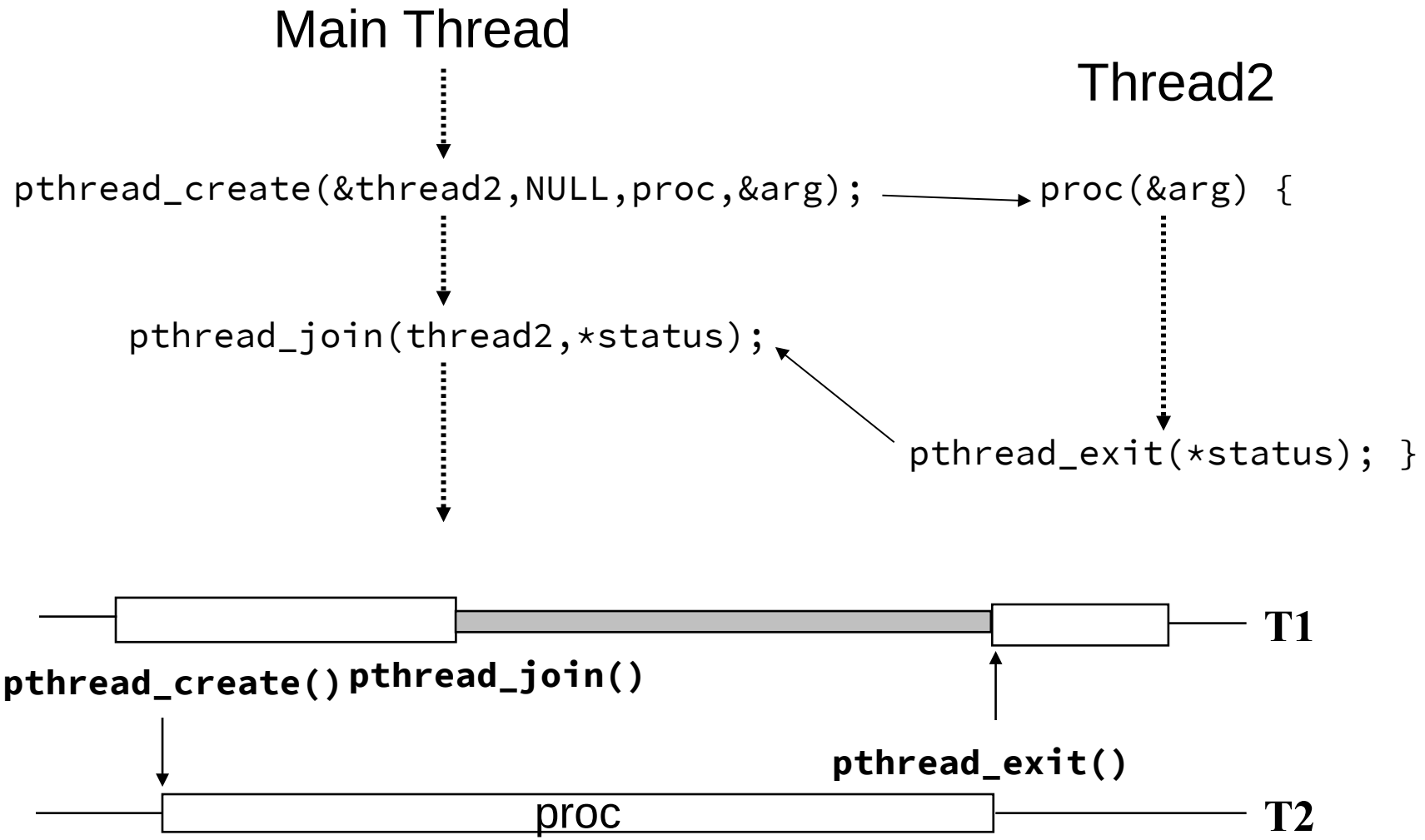
Another Execution



Unpredictable
Speed

Simple Threads API

- void thread_create (thread, func, arg)
  - Create a new thread, storing information about it in thread. Concurrently with the calling thread, thread executes the function func with the argument arg.
- void thread_yield ()
  - The calling thread voluntarily gives up the processor to let some other thread(s) run. The scheduler can resume running the calling thread whenever it chooses to do so.
- int thread_join (thread)
  - Wait for thread to finish if it has not already done so; then return the value passed to thread_exit by that thread. Note that thread_join may be called only once for each thread.
- void thread_exit (ret)
  - Finish the current thread. Store the value ret in the current thread's data structure. If another thread is already waiting in a call to thread_join, resume it.


Main Thread
Thread2
pthread_create(&thread2,NULL,proc,&arg);
proc(&arg) {
pthread_join(thread2,*status);
pthread_exit(*status); }
T1
pthread_create()
pthread_join()
pthread_exit()
proc
T2

- There a several API
  - Win32 threads.
  - C-Threads (user level)
  - Pthreads
    - POSIX IEEE 1003.1c, published in 1995
- POSIX defines functions for the management of threads
  - Functions/data started with the prefix **pthread_**
- Definitions available in the **pthread.h** file
- Code should be linked with the pthread library
  - **-lpthread**

Thread Identification

- Each thread has a unique identifier of type `pthread_t`
- A thread knows its ID calling **pthread_self()**
  - `pthread_t pthread_self();`
- To compare thread identifiers use
  - `int pthread_equal(pthread_t,pthread_t)`
- To print use format `%lu` (long unsigned)

Thread creation

- The main() method comprises a single, default thread.
- pthread_create() creates a new thread and makes it executable.
- The maximum number of threads that may be created by a process in implementation dependent.
- Once created, threads are peers, and may create other threads.

Thread creation

- A thread is started with

int pthread_create(pthread_t *thread,
const pthread_attr_t *attr,
void *(*start_routine) (void *),
void *arg);

  - 1st parameter – Pointer to thread identifier (out)
  - 2nd parameter – Pointer to thread attributes (IN)
    - Can be NULL .
  - 3rd parameter – Pointer to function containing the thread code
    - Function should be: **void * (func*) (void * arg).**
  - 4th parameter – Pointer to thread arguments
    - Pointer to array, structure, int, …. (can be NULL)

- Returns 0 if successful

Data transfer

- Data can be transmitted into the thread in several ways
  - Global variables
    - Accessible by all threads (synchronization should be applied)
  - 4th parameter of pthread_create
    - This parameter points to any data structure the programmer defines
    - Not use same memory location to multiple threads
      - Coherency not guaranteed
- Out data follows similar pattern

Thread termination

- Several ways to terminate a thread:
  - The thread is complete and returns
  - The pthread_exit() method is called
  - The pthread_cancel() method is invoked
  - The exit() method is called
- The pthread_exit() routine is called
  - By the exiting thread
  - after it has completed its work and it no longer is required to exist.

- The pthread_cancel() routine is called
  - By any thread
  - Terminates other running thread
- If the main thread finishes with pthread_exit
  - the other threads will continue to exist
- The pthread_exit() method does not close files;
  - any files opened inside the thread will remain open, so cleanup must be kept in mind.
- Exit() in main() will terminate all threads

- A thread kills itself by calling
  - int pthread_exit(void *ret);
  - 1st parameter – pointer to the return code/data
    - Can be pointer to any data type
    - Memory location should be accessible outside (either global variable or malloc)
  - RETURN VALUE – This function does not return to the caller.
  - ERRORS – This function always succeeds.
- fazer return(cod) implicitly calls pthread_exit()
- After pthread_exit resources arte maintained
  - Resources are released only after pthread_join().
- int pthread_detach(pthread_t);
  - Resources are immediately releases
  - pthread_join() can not be done.

Wait for a thread

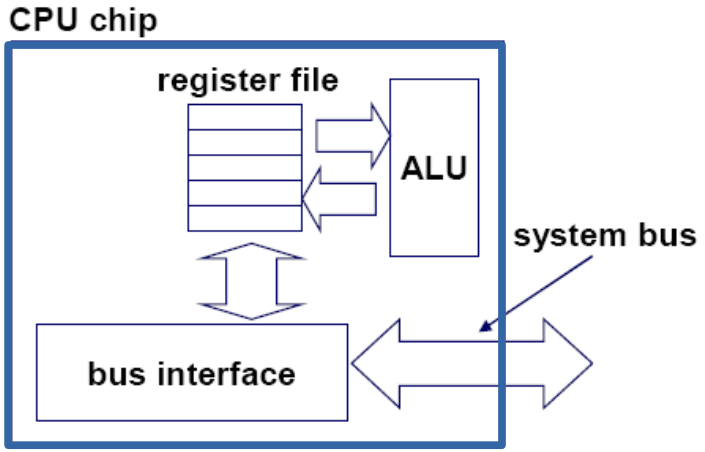
- A thread waits for another thread executing **pthread_join**
  - To release resources
  - To fetch returned data
  - **int pthread_join(pthread_t thread, void **retval);**
    - 1st parameter
      - thread identifier.
    - 2nd parameter
      - Pointer to location of returned valued
- function waits for the thread specified by thread to terminate.
- If that thread has already terminated, then pthread_join() returns immediately.
- Only one thread can wait/join another thread

Multicore CPUs

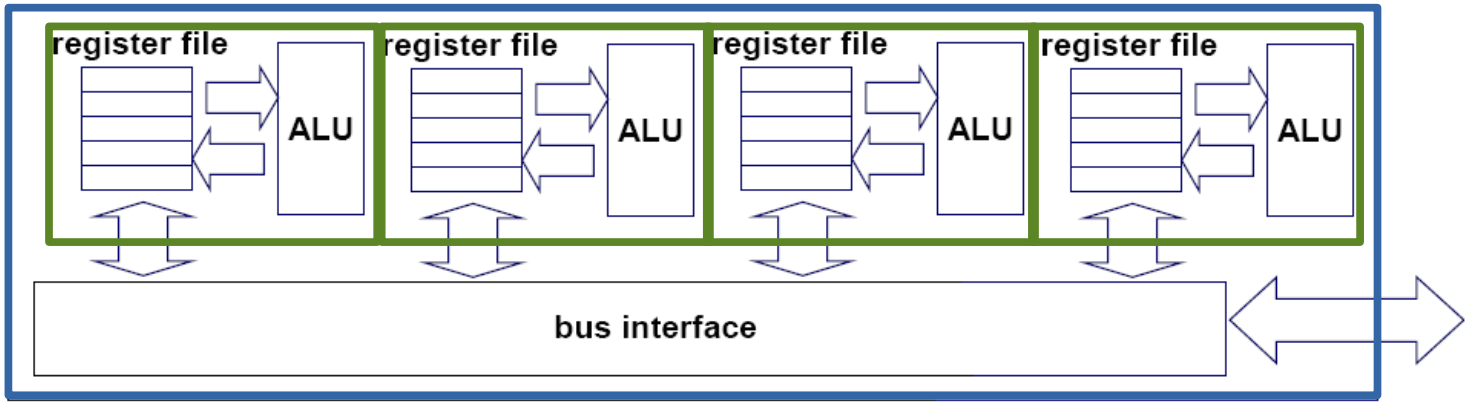
- A core is a processing unit
  - Inside the same packaging (“CPU”)
  - A dual/quad core CPU include 2/4 cores
- Evolution from multi-processor
  - Several CPU connected by a BUS
- First dualcore CPU
  - Power 4, from IBM in 2000.
- First Dual core from intel
  - Jan 2006.
- Commodity processors <12 cores
- Specialized processors < 100 cores

Multicore Architecture

- Single Core Processors



- Multi-core Processors



Multicore Architecture

- Cores share RAM memory
- Caches are:
  - L1 – private (one per cores)
  - L2 – private on most systems
  - L3 – shared
- Threads are assigned to cores
  - Possibly different cores
  - Can be controlled with thread affinity

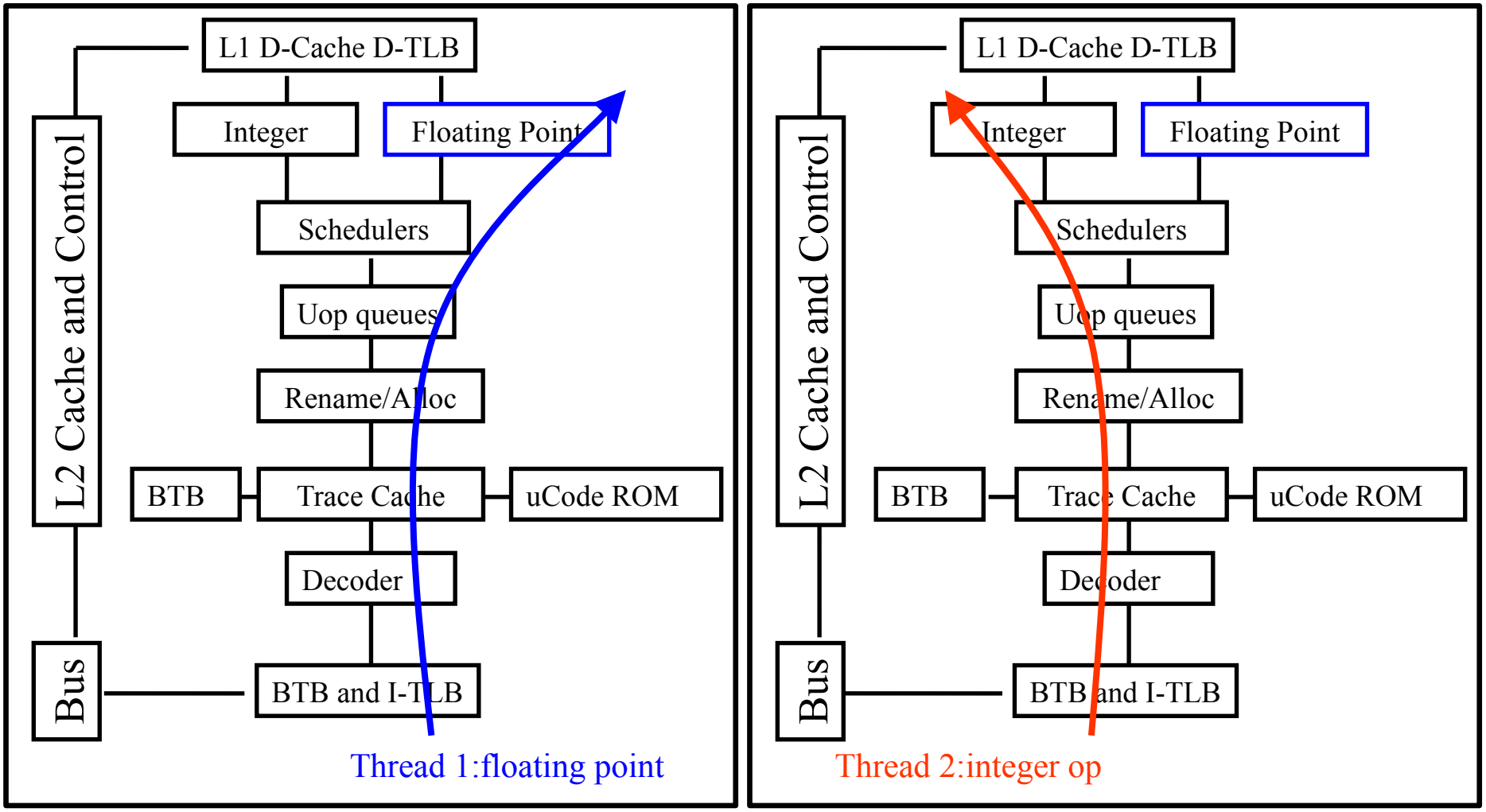
Cache cohrence

- Since each core has one private cache
  - Data coherence is fundamental
- Int var_i = 125 ← shared by thread 1 and 2
  - Thread 1 runs on core 1
  - Thread 2 runs on core 2
- var_i = 0
- var_i is invalidated
- Priv = var_i
  - Cache miss
  - var_i copied from
    - From thread 1 cache

Cache cohrence

- Since each core has one private cache
  - Data coherence is fundamental
- Int var_i, var_j = 125 ← shared by thread 1 and 2
  - Thread 1 runs on core 1
  - Thread 2 runs on core 2
- var_i = 0
- var_i is invalidated
- Priv = var_j
  - Cache miss or hit?

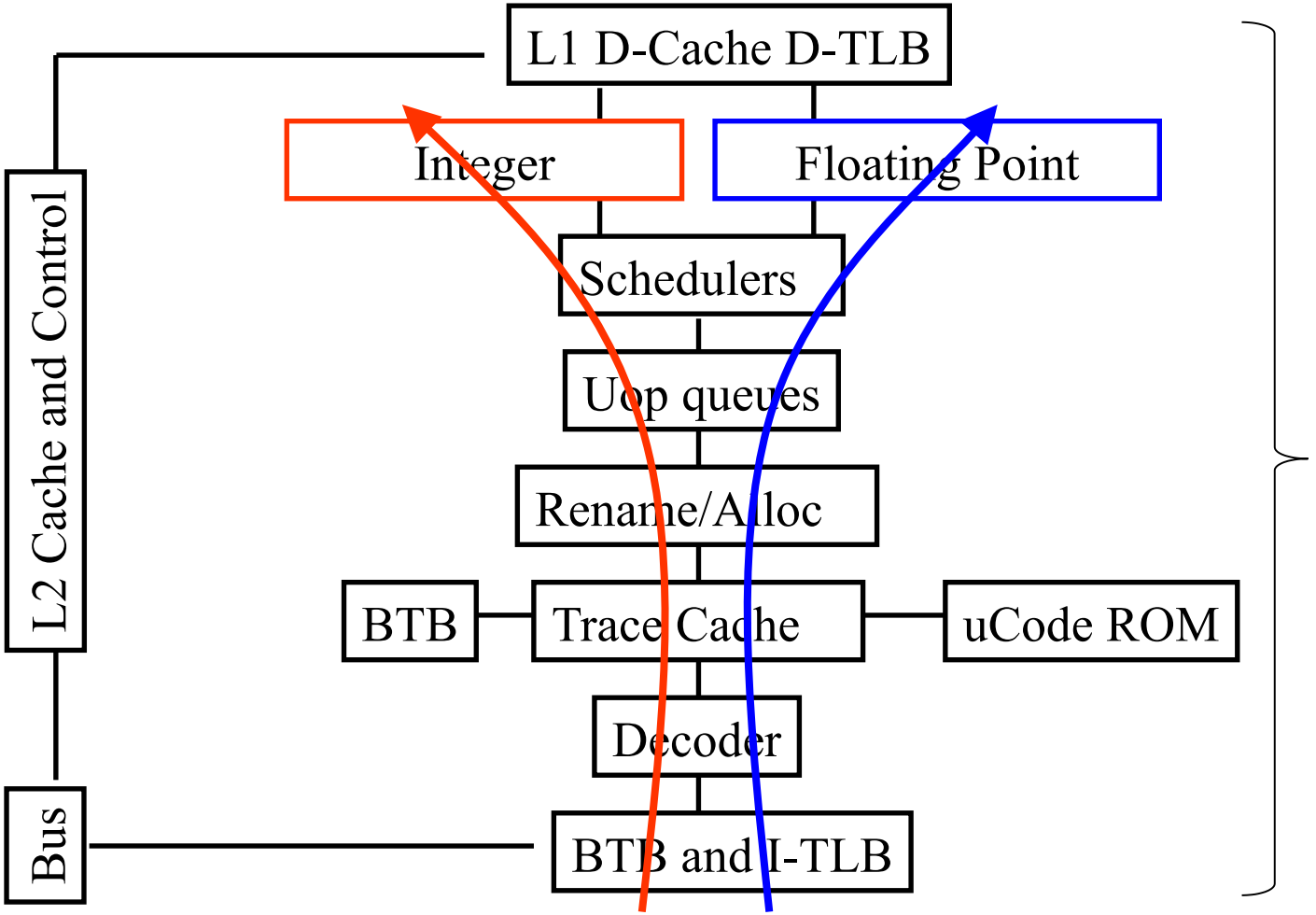
Hiperthreading



Hiperthreading

- Processing can be stalled...
  - Wait from fp result
  - Wait from memory (cache miss)
- In SMT-Simultaneous Multithreading, several thread execute concurrently in the same core, but:
  - On thread processes integred other floats
- Bubles on the piple from one thread
  - Are used by the other threads
- 2 “Virtual” cores per real core
  - /proc/cpuinfo (processor, cpu cores, core id)

Hiperthreading



Hyperthreading

- First intel Simultaneous MT
  - 2002 Xeon, the Pentium 4
  - Hyper-Threading.
- Gains of 15%-30%
  - 5% increase of CPU area
- Each core contains
  - 2 Logical processor
    - Registers , L1 cache of 16KB, Interrupt control.
- 1 real processor
  - System BUS,L2 cache, ALU, FPU.