Threads: Principles

ICS332

Operating Systems

Henri Casanova (henric@hawaii.edu)

Concurrent Programming

- **Concurrency**: the execution of multiple “tasks” at the “same” time
- College students mostly write non-concurrent, or sequential, programs
  - At any point, you could stop the program and say exactly which instruction is being executed and what the runtime stack looks like
  - And there is a single answer to all the above for all execution of your program at the same point in its execution
- In a concurrent program, you design the program in terms of tasks, where each task has a “life of its own”
  - Each task has a specific job to do
  - Tasks may need to “talk” to each other or “wait” for one another
  - Tasks can be in different regions of the code or in the same region of the code a the same time
- It’s a different way of thinking and of programming

Example #1: Make it Fast

- Consider an input array of 10,000 integers: { 23, 56, 7, 68, 68 ...}
- I want to output a boolean array where each element is true if and only if the corresponding integer in the input array is odd
  { true, false, true, false, false ...}
- Assume it takes one millisecond to test an integer value and update the output array
- **Sequential programming:**
  - Iterate through the array, which would take 10,000 milliseconds.
- **Concurrent programming:**
  - If I create 10 “tasks” that each compute 1,000 output values, i.e., 1/10-th of the work, each task takes 1,000 milliseconds
  - Now if I can execute these 10 tasks independently (on a 10-core CPU), the whole execution could take as few as 1,000 milliseconds, i.e., 10x faster
  - In practice, we can’t go quite 10x faster due to various overheads and bottlenecks (e.g., the memory)
  - But we should go much faster than sequential provided be have multiple cores (which we all do in this day and age!)

Example #2: Make it Responsive

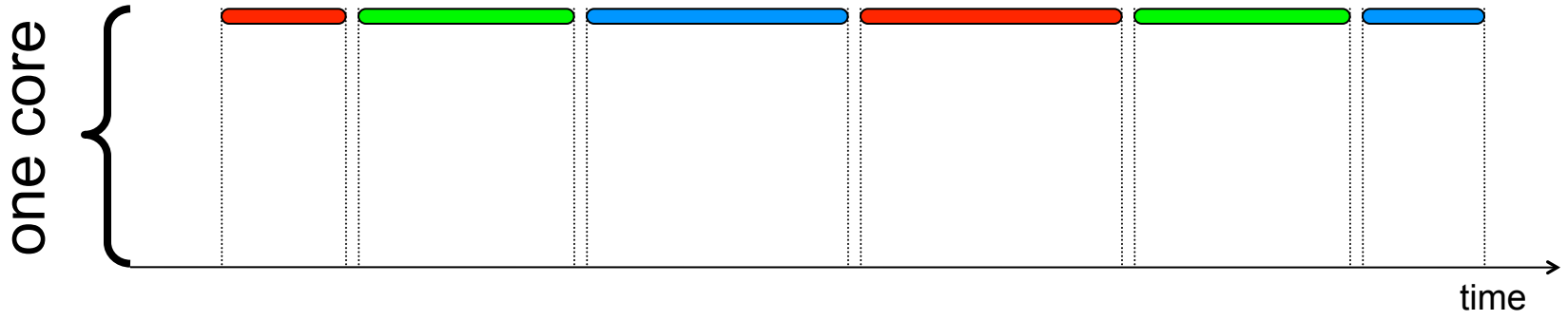
- Consider a Photoshop-like app in which a click of a button launches a transformation filter of all images that a user has selected
  - If many images are selected, this can take minutes
- **Sequential programming:**
  - While the transformation is happening, no other code can run, including the code that reacts to button clicks, meaning that the application is “frozen”, including whatever “Cancel” button one may have tried to implement
  - One solution, which is terrible, is to sprinkle “check whether the button is being clicked” code all over the code that performs the transformation
  - And it may not be feasible if that code is, for instance, a third-party library
- **Concurrent programming:**
  - Create a “task” in charge of watching buttons and reacting to clicks, which runs all the time
  - Whenever the user clicks on some “OK” button to perform the image transformation, create a “task” in charge of the transformation
  - Both tasks then run “at the same time”: while the image transformation is being performed, the user can still interact with the app (e.g., to quit!)

Why Concurrency

- The two previous examples illustrate the two main motivations for concurrency
- Make programs faster
  - Because multiple tasks can use different hardware components at the same time
  - e.g., while task #1 uses a core, task #2 uses another core, and task #3 uses the network card
- Make programs more responsive
  - While a task is blocked or doing something time-consuming, other tasks can still do their work
  - e.g., while a task is waiting for a network packet to arrive, another can animate a spinning beachball

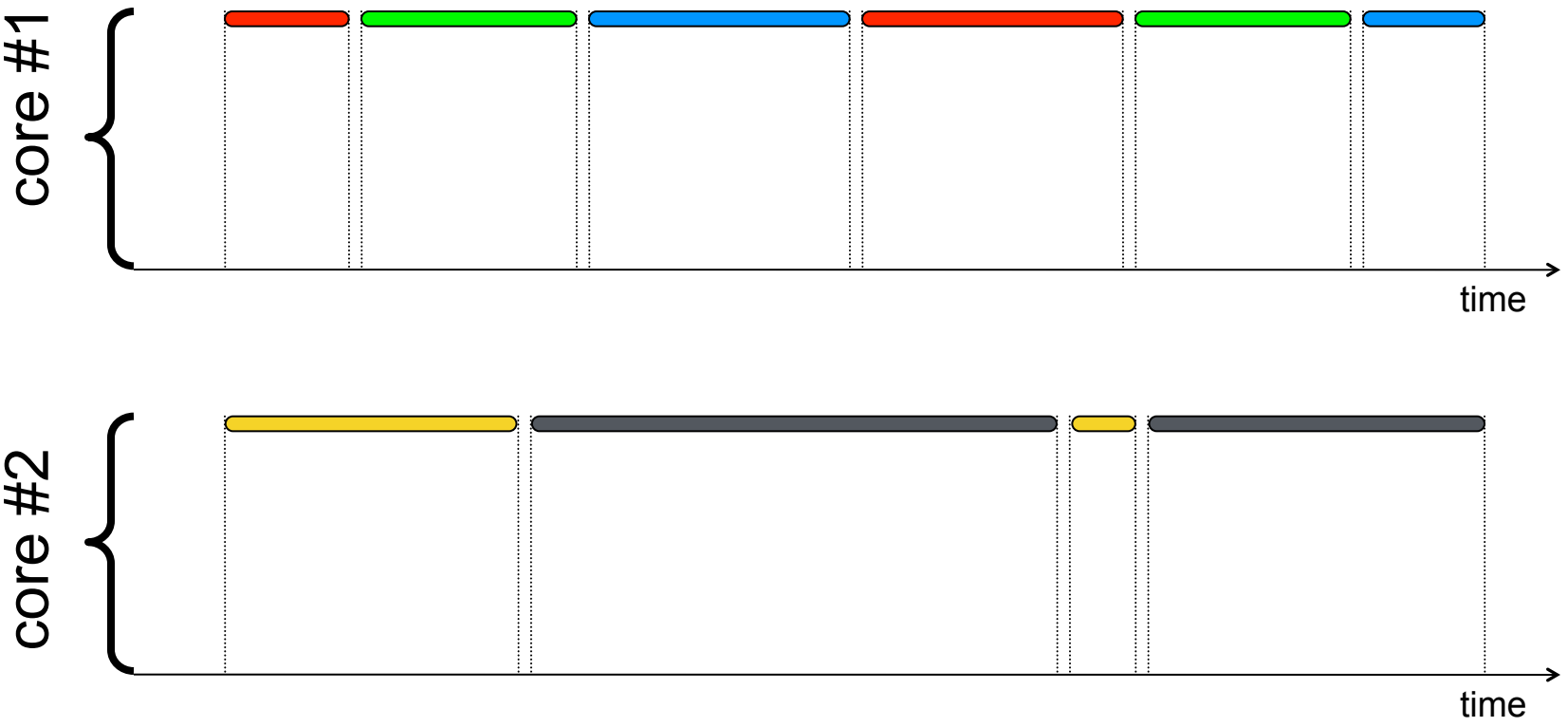
False Concurrency on One Core

- We now know that OSes use context-switching to alternate between processes/threads on a core
- This is known as False Concurrency
- Example (gaps = context-switching overhead):



- Provides the illusion of concurrency to a human because time quanta are short
- Increases core utilization because when a process/thread does I/O, the core is used by another process/thread

True Concurrency on Multiple Cores



- False concurrency within each core
- True concurrency across cores
  - e.g., the yellow and red threads sometimes experience true concurrency

True/False Concurrency

- The programmer should not have to care/know whether concurrency will be true or false
  - A concurrent program with 10 threads will work on a single-core processor, a quad-core processor, a 32-core processor, etc.
  - Typically you don't know on what kind of computer the program will run anyway
- A multi-threaded program will reach higher interactivity with True and/or False concurrency
- A multi-threaded program will reach higher performance only with True concurrency
- **Concurrency is not only about cores:** there can be concurrency between any two hardware resources
  - e.g., between the CPU and the Disk (a Web browser can have a thread that reads data from the disk and a thread that renders that data)

- A "let's just add threads and things will be more interactive and faster" approach often works
- The OS makes it all transparent because it virtualizes the CPU

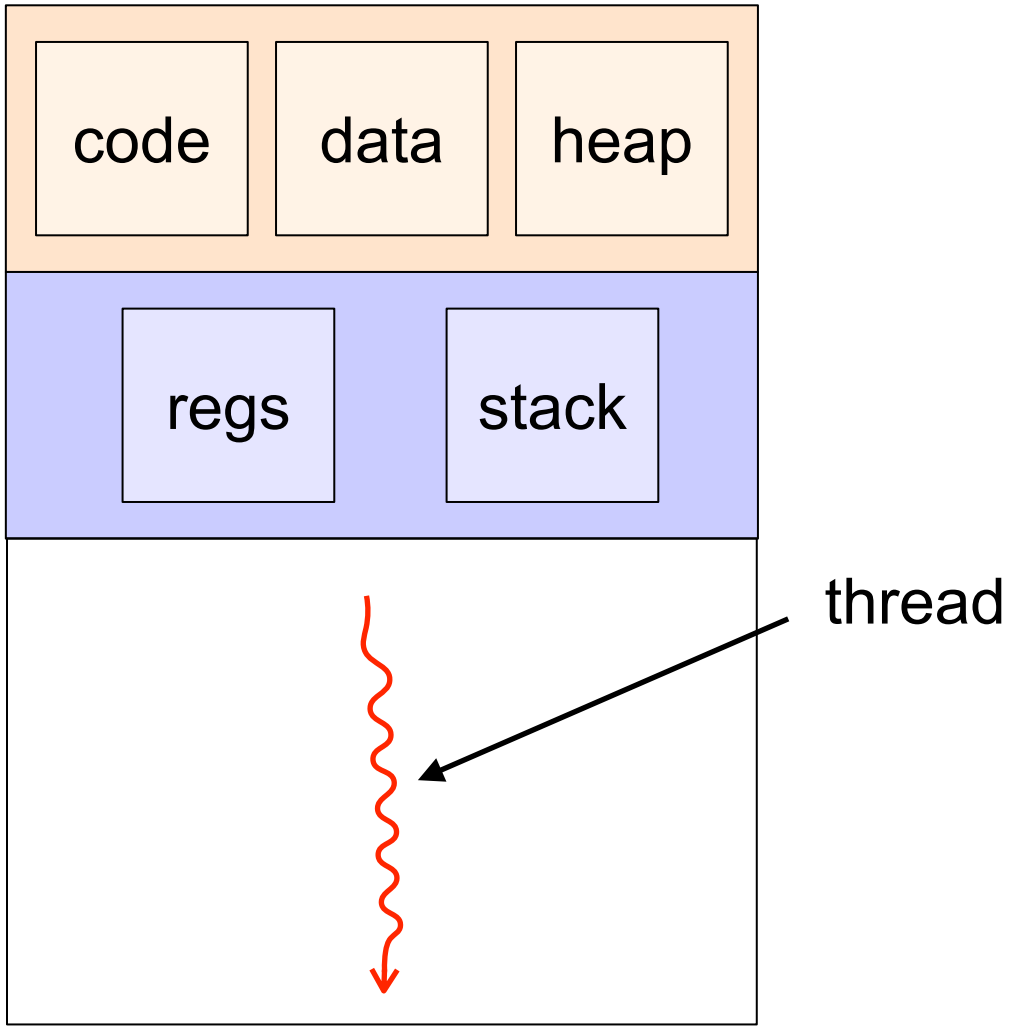
Concurrency with Processes

- We have already talked about concurrency
  - After all it's the 2nd "easy piece" in our textbook
- Processes run concurrently on the computer
  - They were used for concurrent programming a lot until the early 90's
  - And still used today
- But because the OS virtualizes memory, by default processes don't share memory
- We have seen that processes can communicate with IPC
  - *Message passing:* often not easy when processes have complicated cooperating behaviors
  - *Shared Memory:* often simpler, but requires many system calls and is cumbersome, up until the arrival of… threads!

Threads

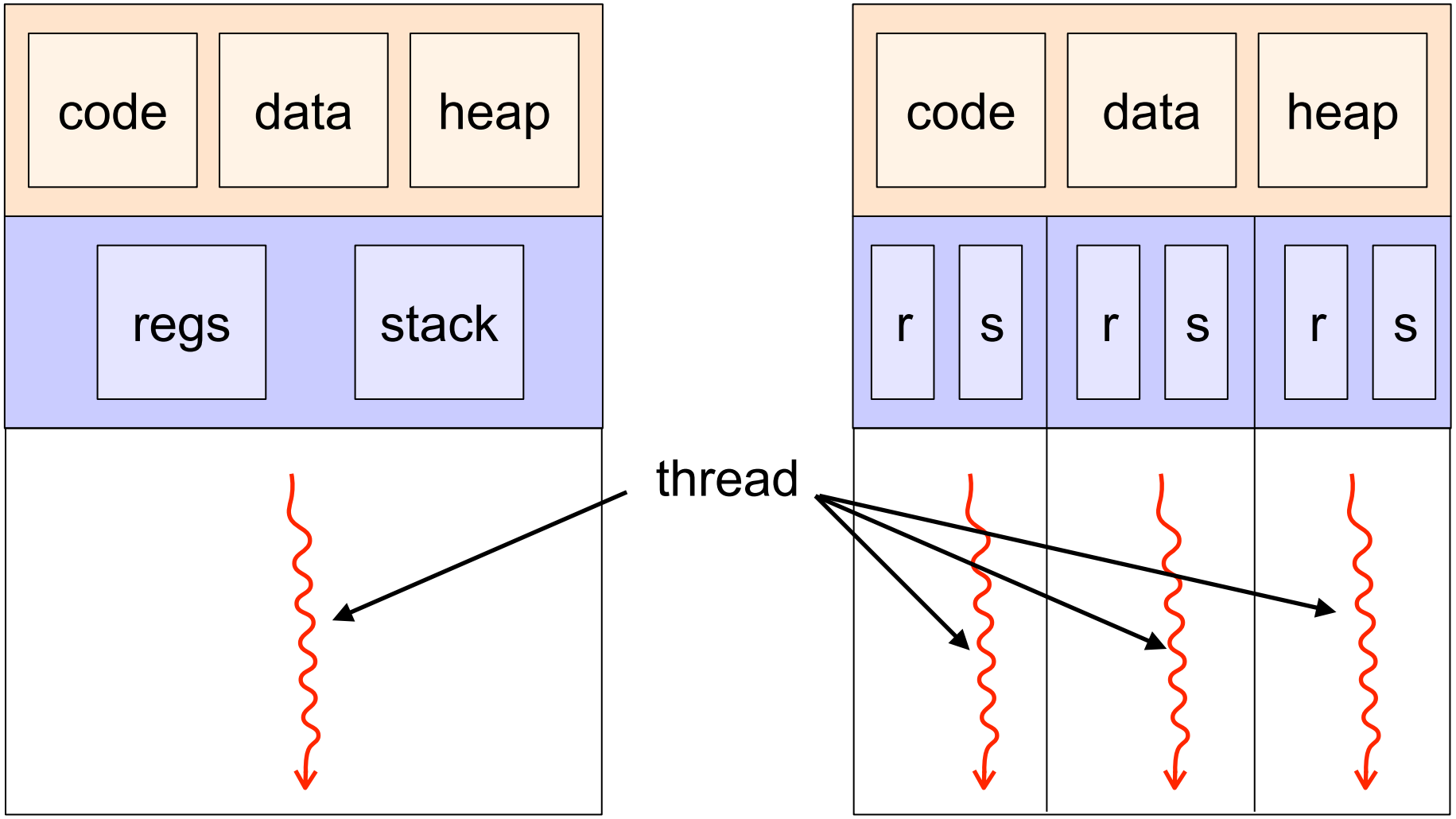
- A thread is a basic unit of CPU utilization within a process (i.e., it can be seen as a “task”)
- A multi-threaded process: Concurrent executions of different parts of the same running program, where each execution is a thread
- Each thread has its own:
  - Thread ID (assigned by the OS)
  - Program Counter (which instruction the thread currently executes)
  - Registers Set (which values are stored in registers)
  - Stack (bookkeeping of the thread’s function/method invocations)
- The above fully defines “what a thread is doing right now”
- But “within a process” threads share:
  - The code/text section
  - The data segment (global variables)
  - The heap
  - And other things (file descriptors, signal handlers, …)

Threads: Typical Representation

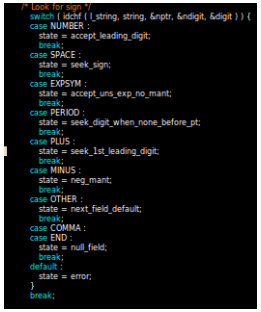


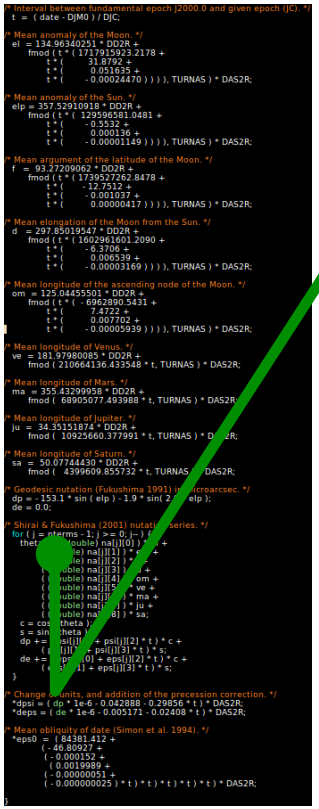
Single-Threaded Process

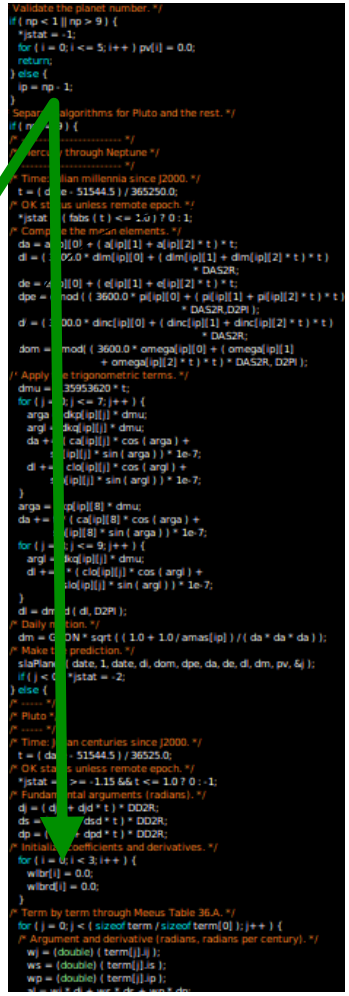
Threads: Typical Representation

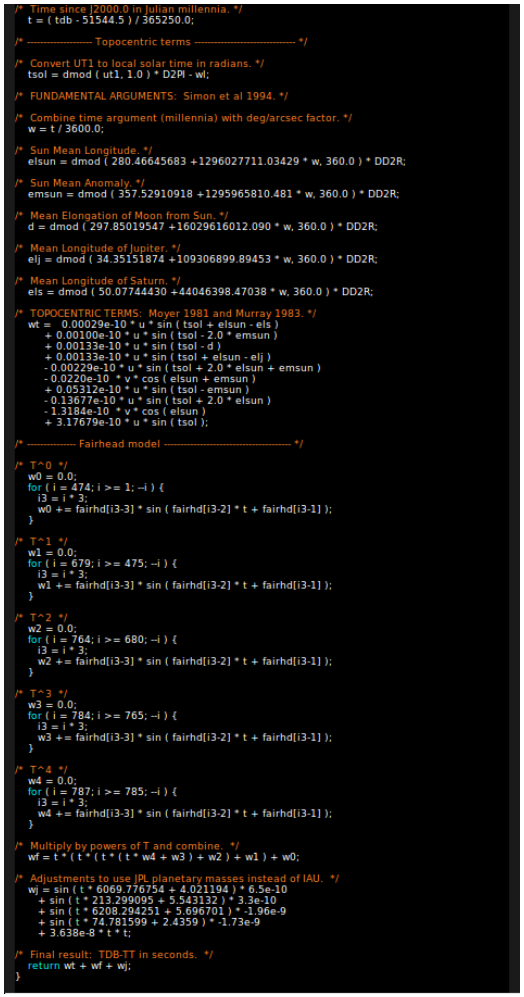


Multi-Threaded Program in Execution

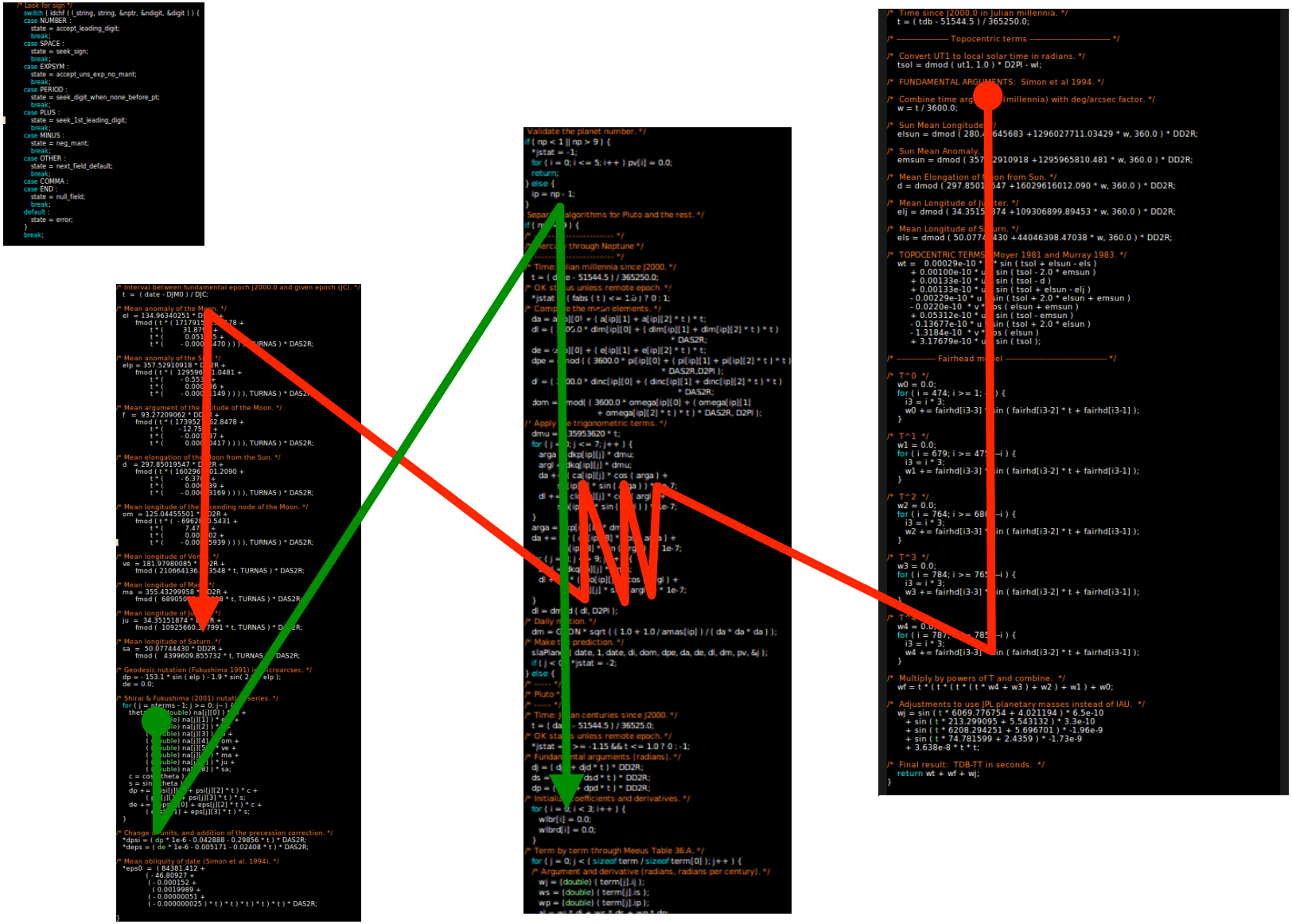




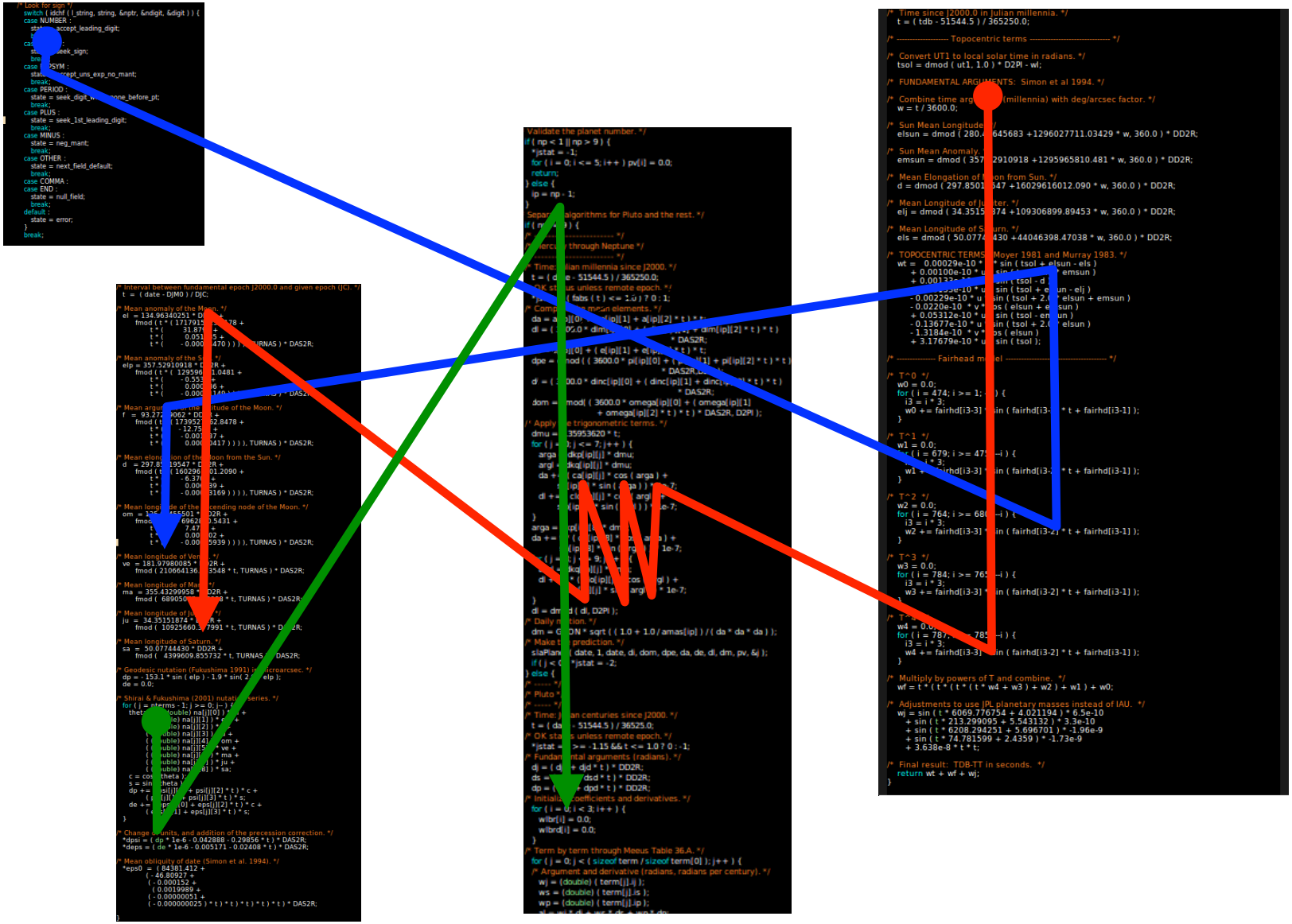




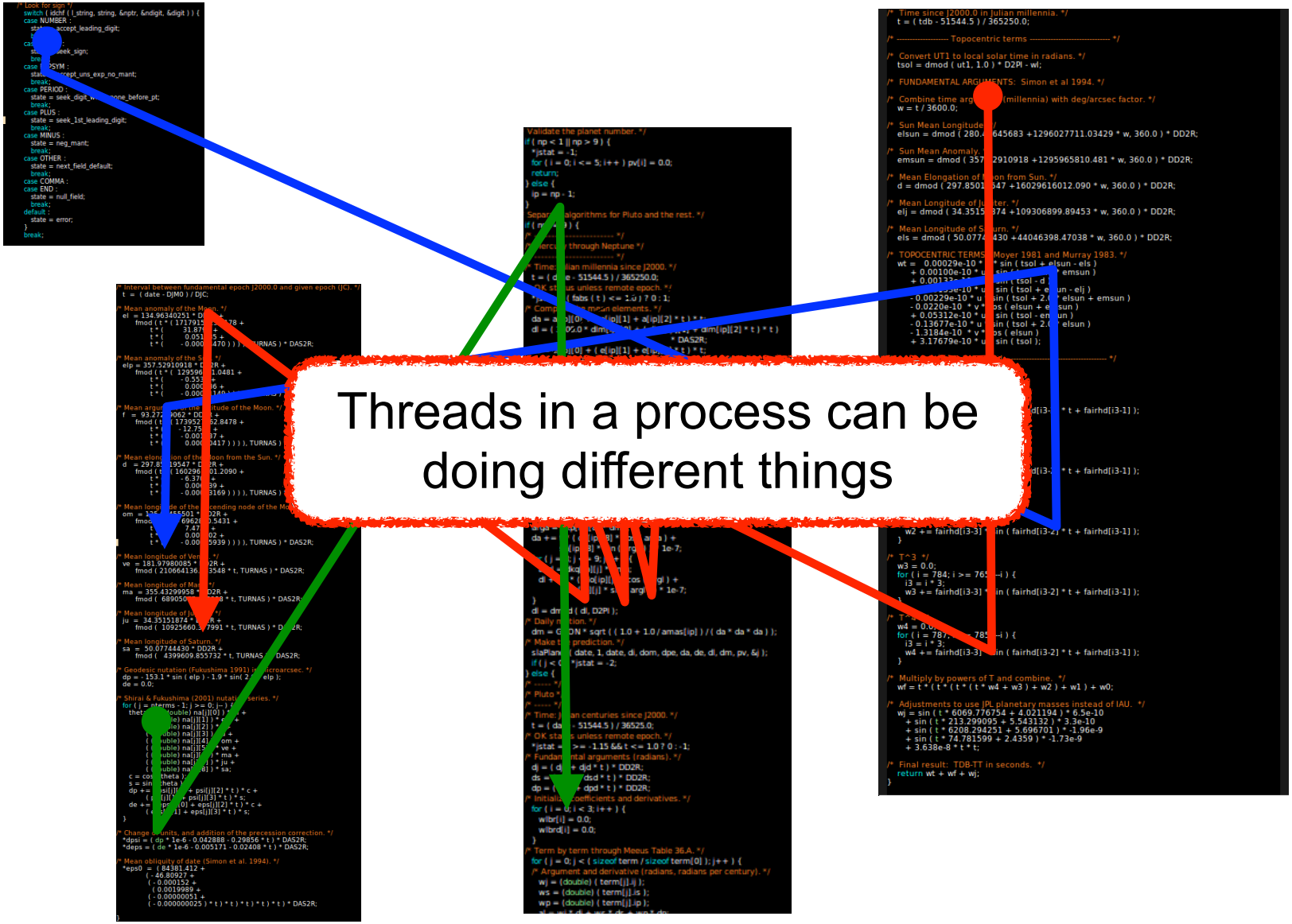
Multi-Threaded Program in Execution



Multi-Threaded Program in Execution

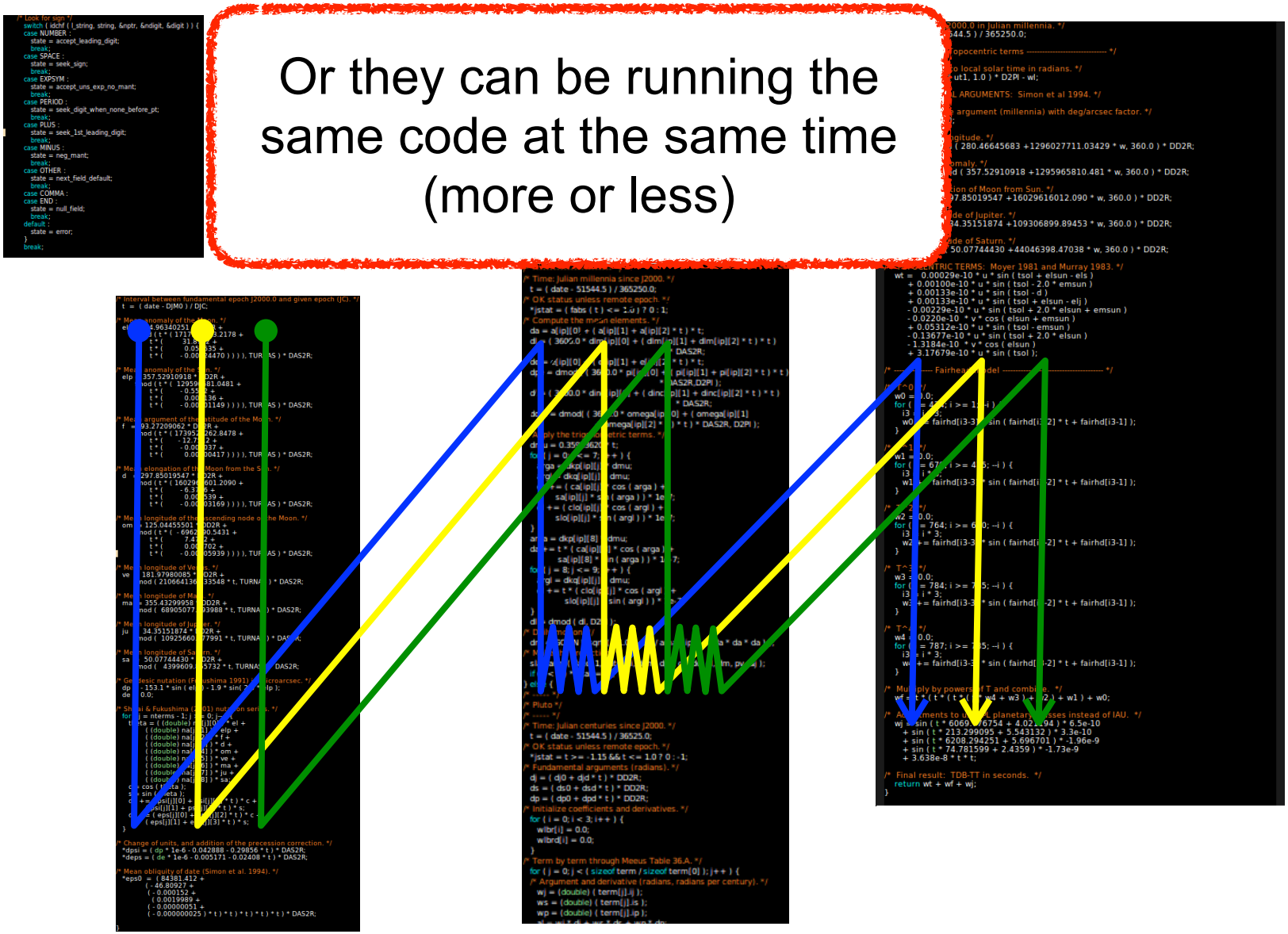


Multi-Threaded Program in Execution

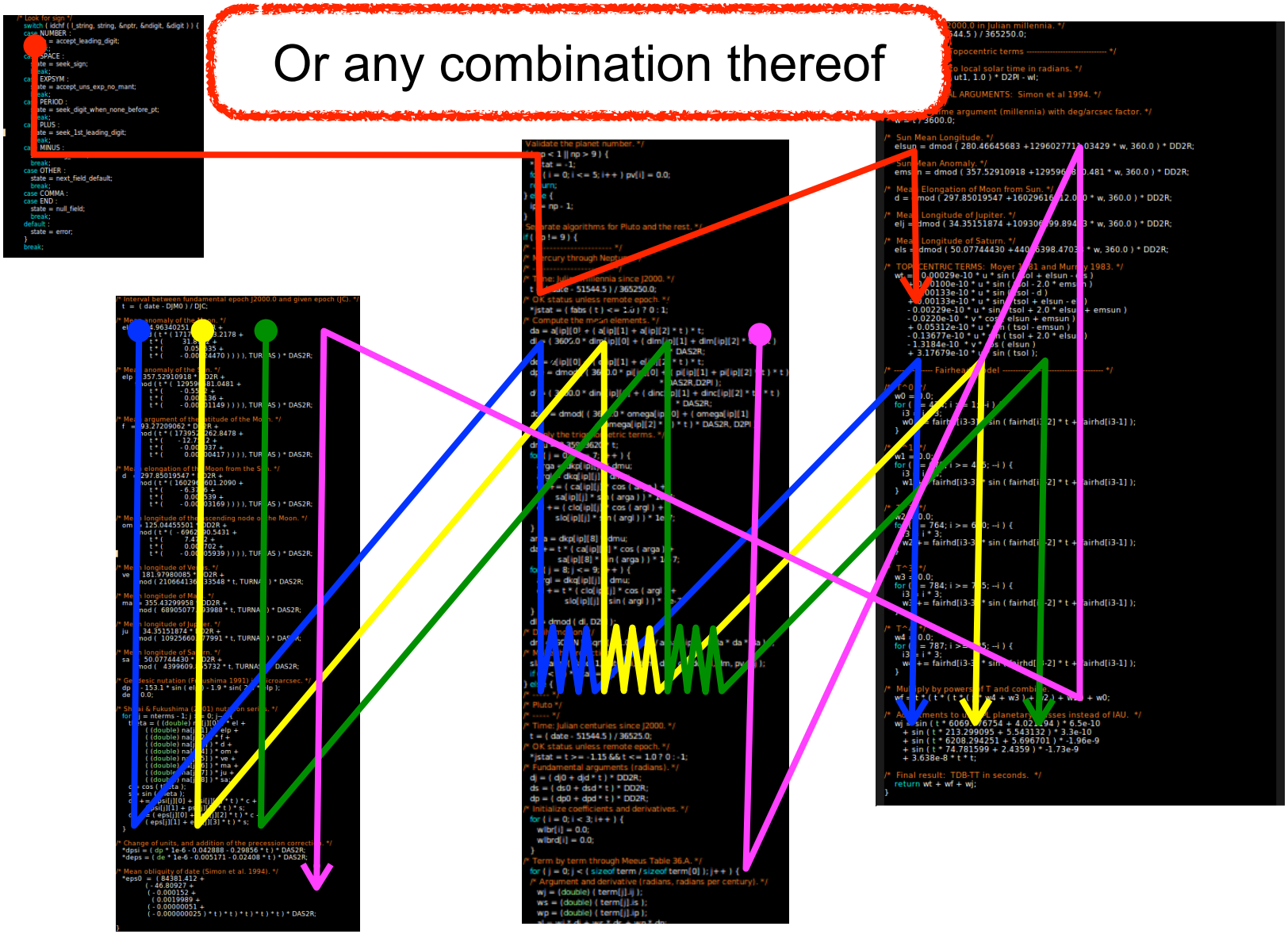


Multi-Threaded Program in Execution

Or they can be running the same code at the same time (more or less)



Multi-Threaded Program in Execution



Threads vs. Processes

- 😀 **Memory sharing**
  - Threads naturally share memory among each other
    - Provides a Shared Memory IPC mechanism with no system calls
  - Having concurrent activities in the same address space is very powerful
  - It makes it possible to implement all kinds of concurrency behaviors/patterns
- 😡 **No memory protection**
  - This is a “feature” since we *want* threads to share memory
  - But this can cause really, really difficult bugs
  - More about this in the Synchronization module
- 😀 **Economy**
  - Creating a thread is cheap
    - Slightly cheaper than creating a process in MacOS/Linux
    - Much cheaper than creating a process in Windows
  - Context-switching between threads in a process is cheaper than between different processes
    - Because they share the same address space (TLB… see much later this semester)
  - So if you can use threads instead of processes, then you likely can go a bit faster
    - In old OSes (Solaris 4), threads were called “lightweight processes”

Threads vs. Processes

- 😡 **Less fault-tolerance**
  - If a thread fails/crashes, then the whole process fails/crashes, instead of processes, which are independent of each other
  - This motivates developers to use both processes and threads (see next slide)
- 😡 **Possibly more memory-constrained**
  - Since threads execute in the same address space, and an OS can bound the size of a process' address space
  - But that's typically not a big deal (one can configure the OS if need be)
- The advantages here are well worth the drawbacks/limitations
  - The big drawback/feature is "no memory protection"
  - We have developed many, many approaches/solutions to deal with it (see the Synchronization module)
- Natural question: is concurrency with processes obsolete?

Concurrency with Processes?

- Should we still care about concurrency with processes?
- **YES** because many applications consists of multiple processes (each of which are often multi-threaded)
- Well-known examples are some popular Web browser (Chrome, Firefox)!
  - They calls fork() each time you open a tab
  - Each tab is a (possibly heavily) multi-threaded process
  - As a result, the code contains processes that do IPC because they don't "see" the same memory naturally
  - But if a tab crashes (due to running bad JavaScript code, for instance) your browser doesn't crash!
  - Google "firefox chrome processes threads" :)
- In real-world settings you often have to put together different software products to make up a whole system
  - Some may just be executables instead of libraries with nice APIs
  - So you have to create processes
  - You interact with them via stdin/stdout/stderr streams for instance (see our programming assignment) or via any supported IPC mechanisms
- **Bottom-line:** don't drink the "I'll only do threads, not processes" Kool-Aid

User vs. Kernel Threads

- Threads can be supported solely in User Space (User Threads)
  - You can write your own thread implementation without help from the OS
  - Often a homework assignment in a graduate OS course
- The main advantage of User Threads is low overhead
  - e.g., because no system calls
- User Threads have several drawbacks:
  - If one thread blocks, all other threads block
  - All threads run on the same core (because the OS doesn't know that there are threads within a process)
- For these reasons User Threads are (no longer) heavily used
  - But Java recently re-introduced its old "Green (user) threads"!!
- All OSes today provide support for threads (Kernel Threads) that can run on different cores and be truly independent of each other
- We typically just call them "threads"

Linux/MacOS Threads

- Processes and Threads are implemented as tasks
  - Kernel data structure: **`task struct`**
  - We already looked at it when we talked about processes
- The **`clone()`** syscall is used to create a task
- It can be invoked with several options, each set or not set
- Each option specifies something the child should share or not share with its parent
- **`fork()`** just calls **`clone()`** with a particular set of options
  - Preserved as a system call for backward compatibility to create processes
- From the man page: “if CLONE_VM is set, the calling process and the child process run in the same memory space”
- To create a process, **`clone()`** is called without the CLONE_VM option
- To create a thread, **`clone()`** is called with, among other things, the CLONE_VM option

Main Takeaways

- Concurrency is used to make programs faster and/or more responsive
- False and True concurrency
- Threads within a process
  - Share the code and the heap
  - Have their own stack, registers, and program counter
- Concurrency with processes is not obsolete, and many programs use both multiple processes and multiple threads

Conclusion

- Threads have been mainstream for decades: multi-threading today is everywhere, in part due to us having multi-core architectures
- Let's do a **`ps auxM`** on my MacOS laptop and see how many processes are multi-threaded…
  - When I did this while back writing this slides I got 350 processes and 1157 threads. Almost all processes are multithreaded, with up to 60+ threads for a process
- Let's move on to seeing how to use them in programming languages…
  - Almost all languages have some way to create and manage threads!