Threads

What is the thread? * it represents a single unit of execution on the OS-level * if you want to execute some code, you need a thread *scheduled* on a CPU

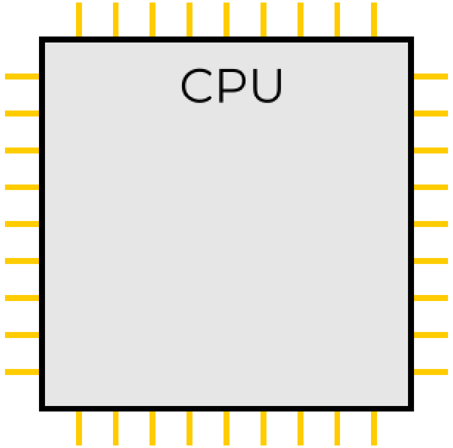
```
public void Method(int arg) {
   int x = CalculateX(arg);
   int y = CalculateY(arg);
   Thread.Sleep(1000);
   DoSomething(x, y);
}
```

```
C.Method(Int32)
    L0000: push rdi
    L0001: push rsi
    L0002: push rbx
    L0003: sub rsp, 0x20
    L0007: mov rsi, rcx
    L000a: mov edi, edx
    L000c: mov rcx, rsi
    L000f: mov edx, edi
    L0011: call C.CalculateX(Int32)
    L0016: mov ebx, eax
    L0018: mov rcx, rsi
    L001b: mov edx, edi
    L001d: call C.CalculateY(Int32)
    L0022: mov edi, eax
    L0024: mov ecx, 0x3e8
    L0029: call Thread.Sleep(Int32)
    L002e: mov rcx, rsi
    L0031: mov edx, ebx
    L0033: mov r8d, edi
    L0036: mov rax, 0x7ffa23c00088
    L0040: add rsp, 0x20
    L0044: pop rbx
    L0045: pop rsi
    L0046: pop rdi
    L0047: jmp rax
```

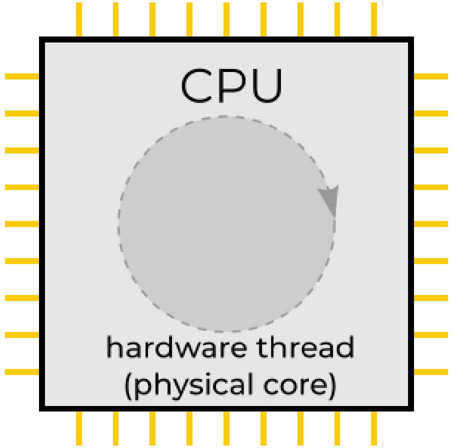
Threads vs hardware

- single-CPU
- multi-CPU systems
- Hyper-Threading (Intel), SMT (AMD) - *simultaneous multithreading*

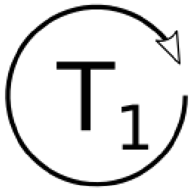
```
C.Method(Int32)
    L0000: push rdi
    L0001: push rsi
    L0002: push rbx
    L0003: sub rsp, 0x20
    L0007: mov rsi, rcx
    L000a: mov edi, edx
    L000c: mov rcx, rsi
    L000f: mov edx, edi
    L0011: call C.CalculateX(Int32)
    L0016: mov ebx, eax
    L0018: mov rcx, rsi
    L001b: mov edx, edi
    L001d: call C.CalculateY(Int32)
    L0022: mov edi, eax
    L0024: mov ecx, 0x3e8
    L0029: call Thread.Sleep(Int32)
    L002e: mov rcx, rsi
    L0031: mov edx, ebx
    L0033: mov r8d, edi
    L0036: mov rax, 0x7ffa23c00088
    L0040: add rsp, 0x20
    L0044: pop rbx
    L0045: pop rsi
    L0046: pop rdi
    L0047: jmp rax
```



```
C.Method(Int32)
    L0000: push rdi
    L0001: push rsi
    L0002: push rbx
    L0003: sub rsp, 0x20
    L0007: mov rsi, rcx
    L000a: mov edi, edx
    L000c: mov rcx, rsi
    L000f: mov edx, edi
    L0011: call C.CalculateX(Int32)
    L0016: mov ebx, eax
    L0018: mov rcx, rsi
    L001b: mov edx, edi
    L001d: call C.CalculateY(Int32)
    L0022: mov edi, eax
    L0024: mov ecx, 0x3e8
    L0029: call Thread.Sleep(Int32)
    L002e: mov rcx, rsi
    L0031: mov edx, ebx
    L0033: mov r8d, edi
    L0036: mov rax, 0x7ffa23c00088
    L0040: add rsp, 0x20
    L0044: pop rbx
    L0045: pop rsi
    L0046: pop rdi
    L0047: jmp rax
```



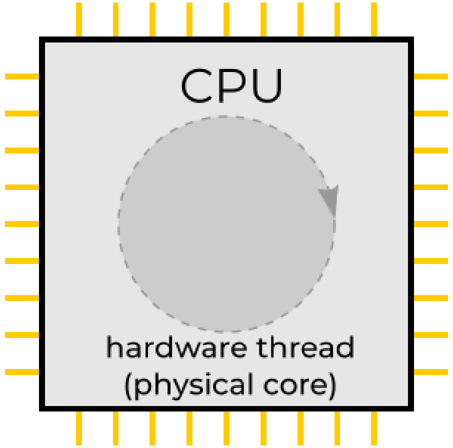
```
C.Method(Int32)
    L0000: push rdi
    L0001: push rsi
    L0002: push rbx
    L0003: sub rsp, 0x20
    L0007: mov rsi, rcx
    L000a: mov edi, edx
    L000c: mov rcx, rsi
    L000f: mov edx, edi
    L0011: call C.CalculateX(Int32)
    L0016: mov ebx, eax
    L0018: mov rcx, rsi
    L001b: mov edx, edi
    L001d: call C.CalculateY(Int32)
    L0022: mov edi, eax
    L0024: mov ecx, 0x3e8
    L0029: call Thread.Sleep(Int32)
    L002e: mov rcx, rsi
    L0031: mov edx, ebx
    L0033: mov r8d, edi
    L0036: mov rax, 0x7ffa23c00088
    L0040: add rsp, 0x20
    L0044: pop rbx
    L0045: pop rsi
    L0046: pop rdi
    L0047: jmp rax
```



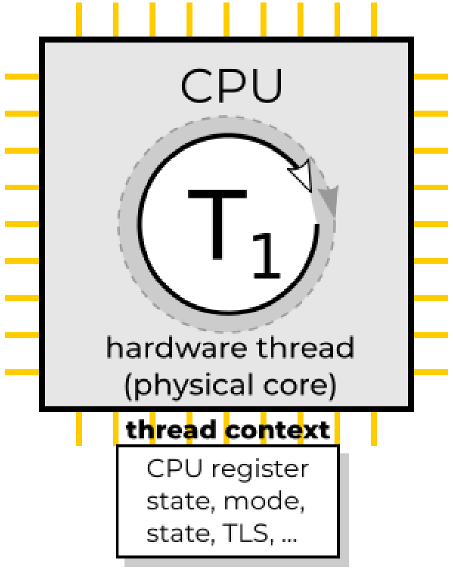
software thread

thread context

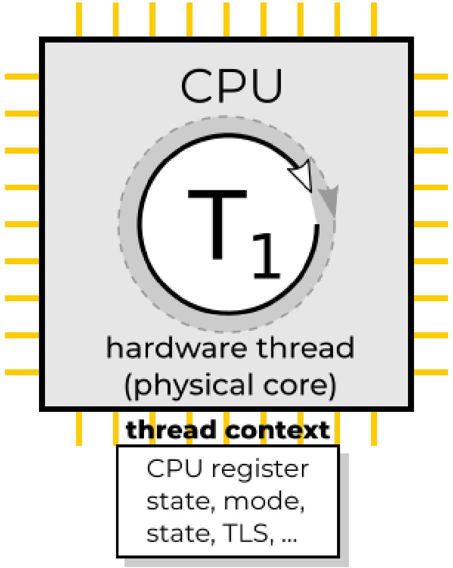
CPU register state, mode, state, TLS, ...



```
C.Method(Int32)
    L0000: push rdi
    L0001: push rsi
    L0002: push rbx
    L0003: sub rsp, 0x20
    L0007: mov rsi, rcx
    L000a: mov edi, edx
    L000c: mov rcx, rsi
    L000f: mov edx, edi
    L0011: call C.CalculateX(Int32)
    L0016: mov ebx, eax
    L0018: mov rcx, rsi
    L001b: mov edx, edi
    L001d: call C.CalculateY(Int32)
    L0022: mov edi, eax
    L0024: mov ecx, 0x3e8
    L0029: call Thread.Sleep(Int32)
    L002e: mov rcx, rsi
    L0031: mov edx, ebx
    L0033: mov r8d, edi
    L0036: mov rax, 0x7ffa23c00088
    L0040: add rsp, 0x20
    L0044: pop rbx
    L0045: pop rsi
    L0046: pop rdi
    L0047: jmp rax
```



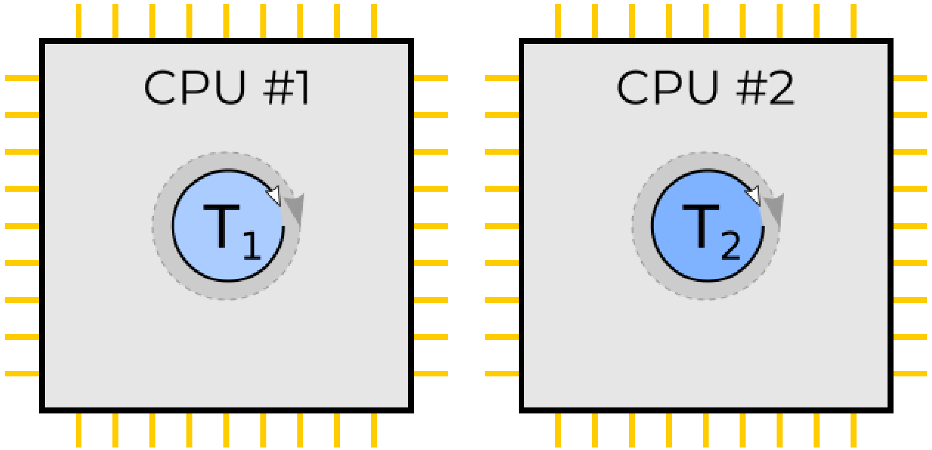
```
C.Method(Int32)
    L0000: push rdi
    L0001: push rsi
    L0002: push rbx
    L0003: sub rsp, 0x20
    L0007: mov rsi, rcx
    L000a: mov edi, edx
    L000c: mov rcx, rsi
    L000f: mov edx, edi
    L0011: call C.CalculateX(Int32)
    L0016: mov ebx, eax
    L0018: mov rcx, rsi
    L001b: mov edx, edi
    L001d: call C.CalculateY(Int32)
    L0022: mov edi, eax
    L0024: mov ecx, 0x3e8
    L0029: call Thread.Sleep(Int32)
    L002e: mov rcx, rsi
    L0031: mov edx, ebx
    L0033: mov r8d, edi
    L0036: mov rax, 0x7ffa23c00088
    L0040: add rsp, 0x20
    L0044: pop rbx
    L0045: pop rsi
    L0046: pop rdi
    L0047: jmp rax
```



Single-CPU/core

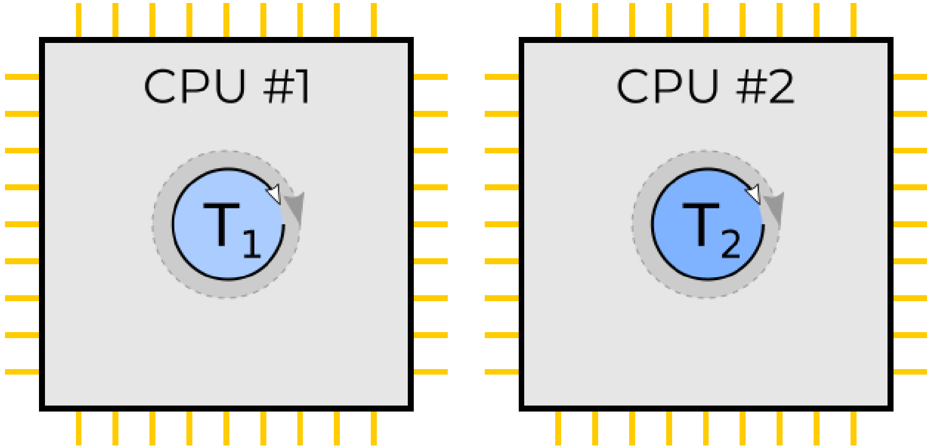
```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```

```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```



```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```

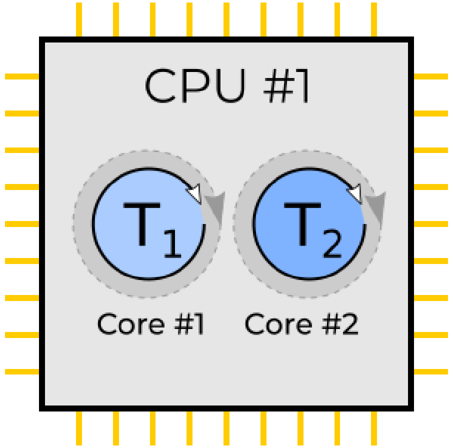
```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```



Multiple CPUs

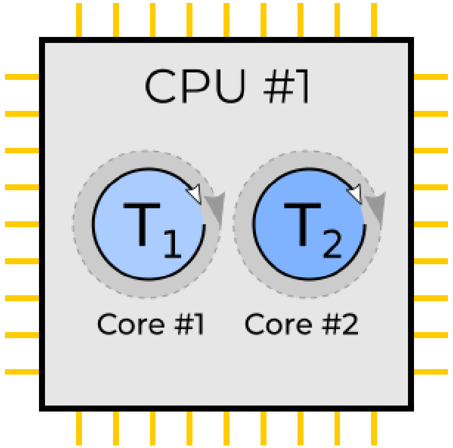
```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```

```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```



```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```

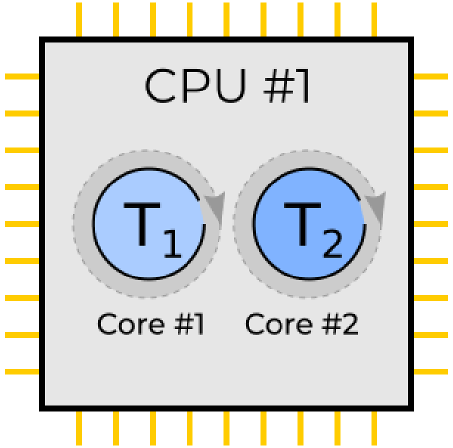
```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```



Single CPU with multiple cores

```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```

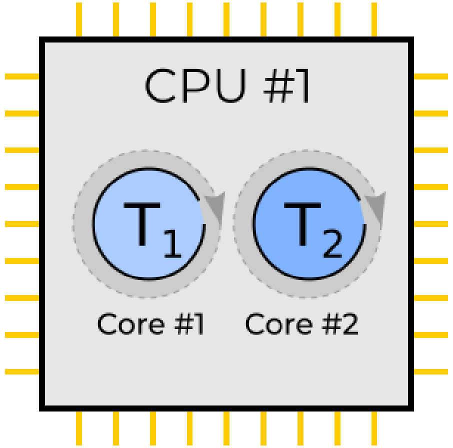
```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```



```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]

mov rcx, [rcx+0x10]

mov r9, r8
cmp edx, [r9+0x8]

jae L003e
movsxd r10, edx
add eax, [r9+r10]

mov r9, rcx
cmp edx, [r9+0x8]

jae L003e
add eax, [r9+r10]

inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```

```
...
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]

mov rcx, [rcx+0x10]

mov r9, r8
cmp edx, [r9+0x8]

jae L003e
movsxd r10, edx
add eax, [r9+r10]

mov r9, rcx
cmp edx, [r9+0x8]

jae L003e
add eax, [r9+r10]

inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...
```

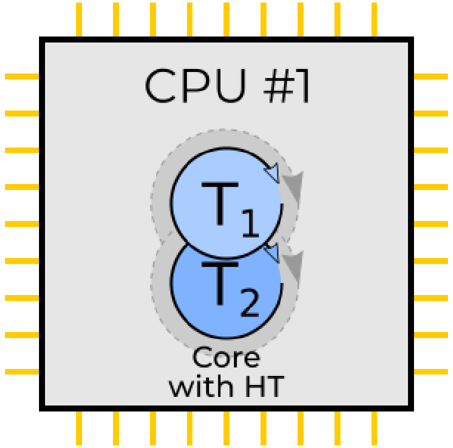
Micro stale
(fe. memory access)



Thread execution contains some micro-stalls on the microarchitecture level (like waiting for memory access), we waste Cores time

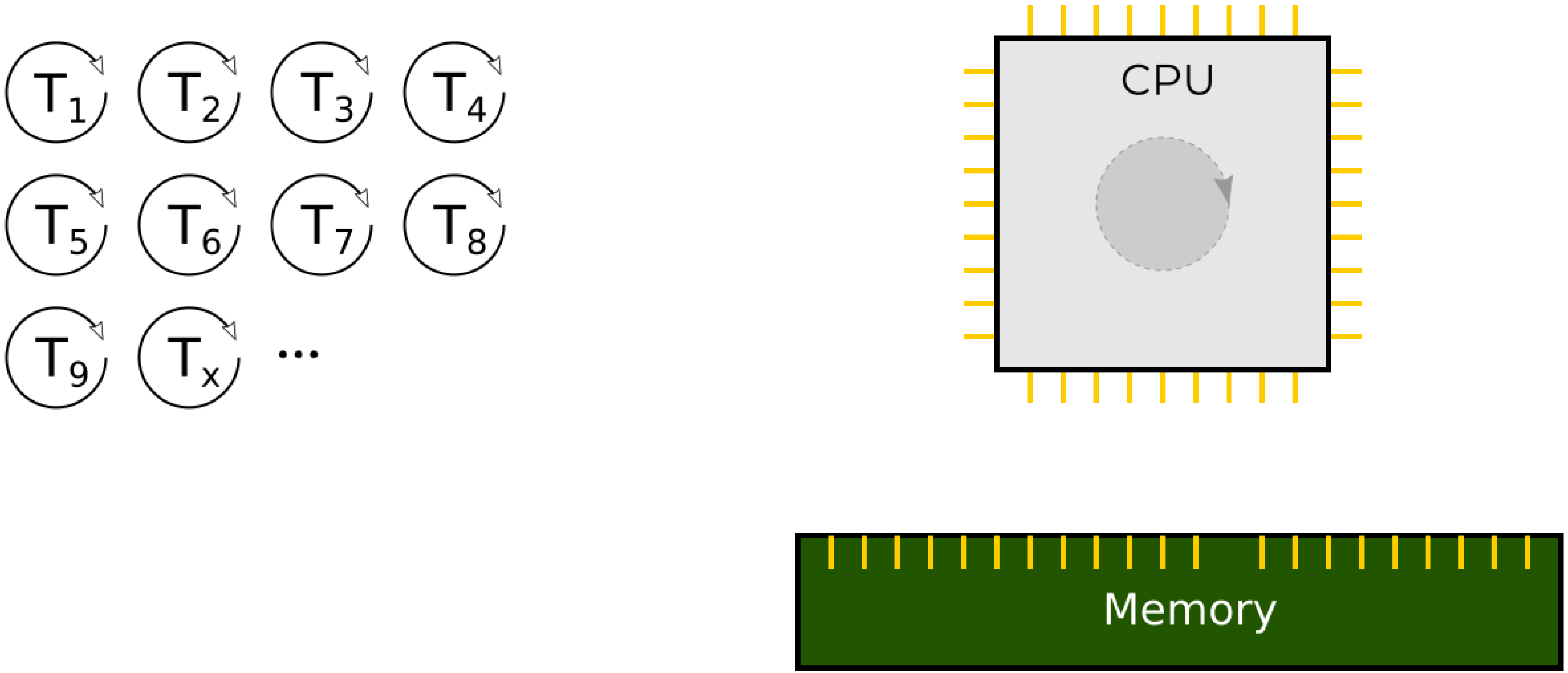
```
   ...
   sub rsp, 0x28
...xor eax, eax
sub rsp, 0x28
xor eax, eax
xor edx, edx
mov r8, [rcx+0x8]
   xor edx, edx
   mov r8, [rcx+0x8]
mov rcx, [rcx+0x10]
   mov rcx, [rcx+0x10]
mov r9, r8
cmp edx, [r9+0x8]
   mov r9, r8
   cmp edx, [r9+0x8]
jae L003e
movsxd r10, edx
add eax, [r9+r10]
   jae L003e
   movsxd r10, edx
   add eax, [r9+r10]
mov r9, rcx
cmp edx, [r9+0x8]
   mov r9, rcx
   cmp edx, [r9+0x8]
jae L003e
add eax, [r9+r10]
   jae L003e
   add eax, [r9+r10]
inc edx
cmp edx, 0x80
jl L0010
add rsp, 0x28
ret
...inc edx
   cmp edx, 0x80
   jl L0010
   add rsp, 0x28
   ret
   ...
```

Interleave during stale
(fe. memory access)

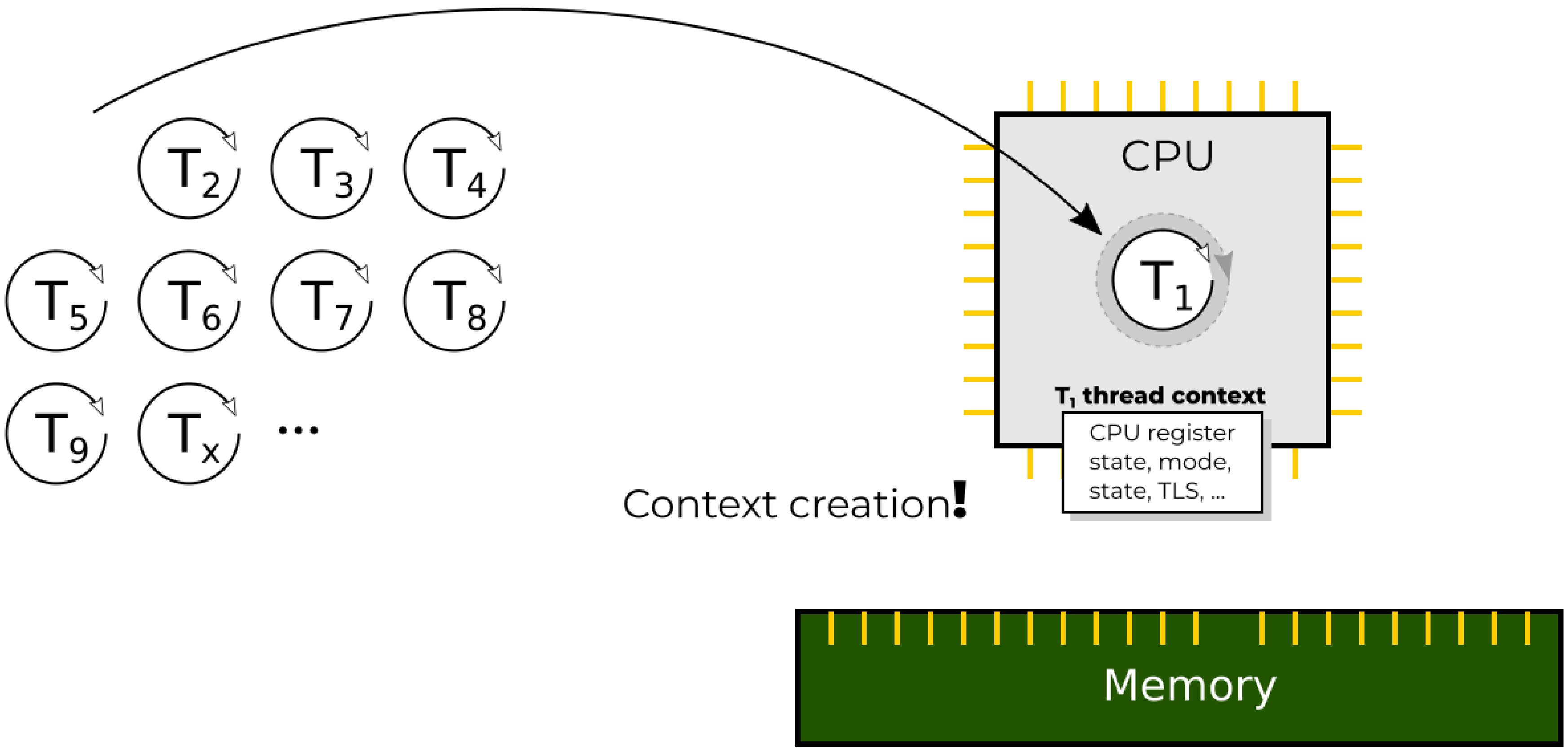


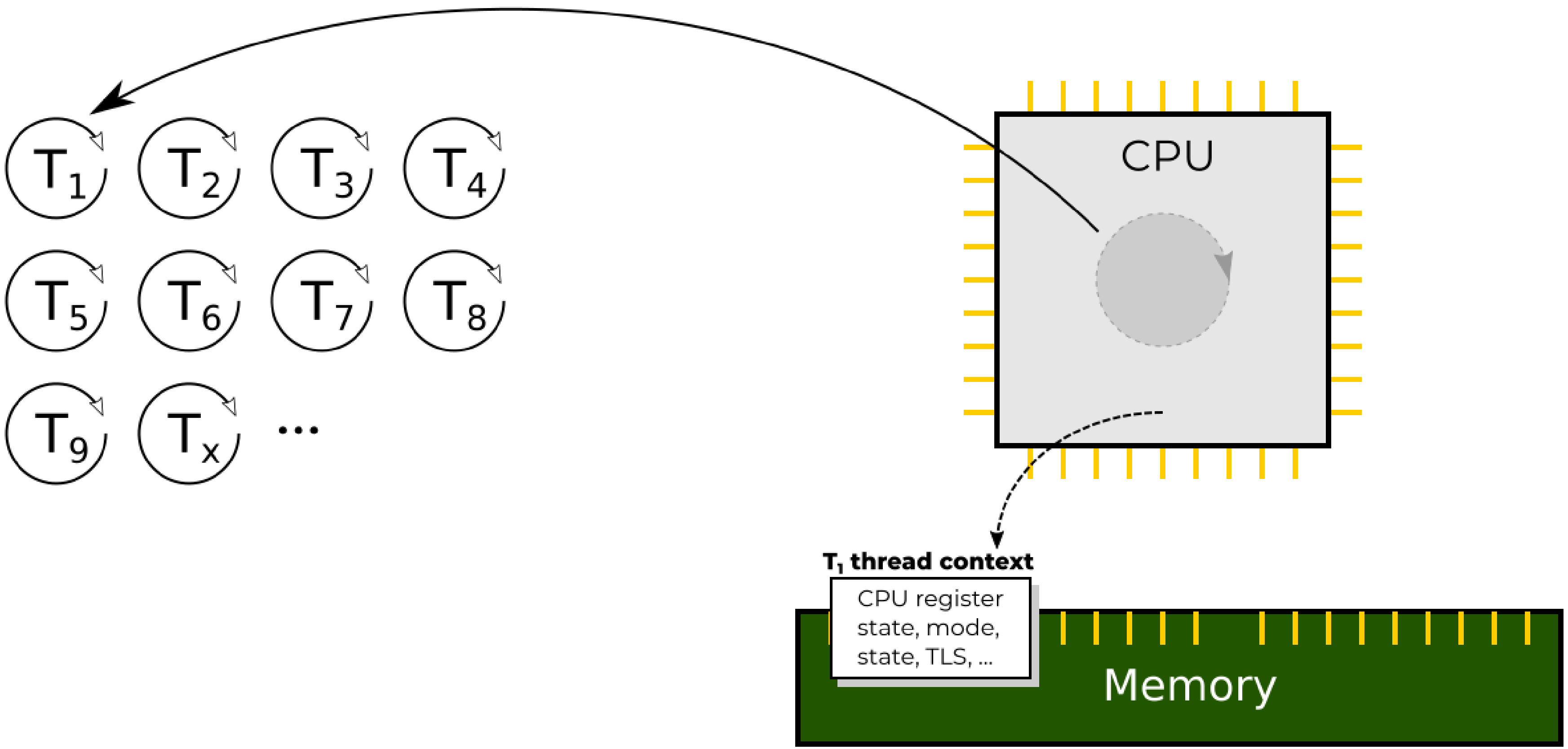
Interleave two threads to better utilize single Core

Many threads, single CPU

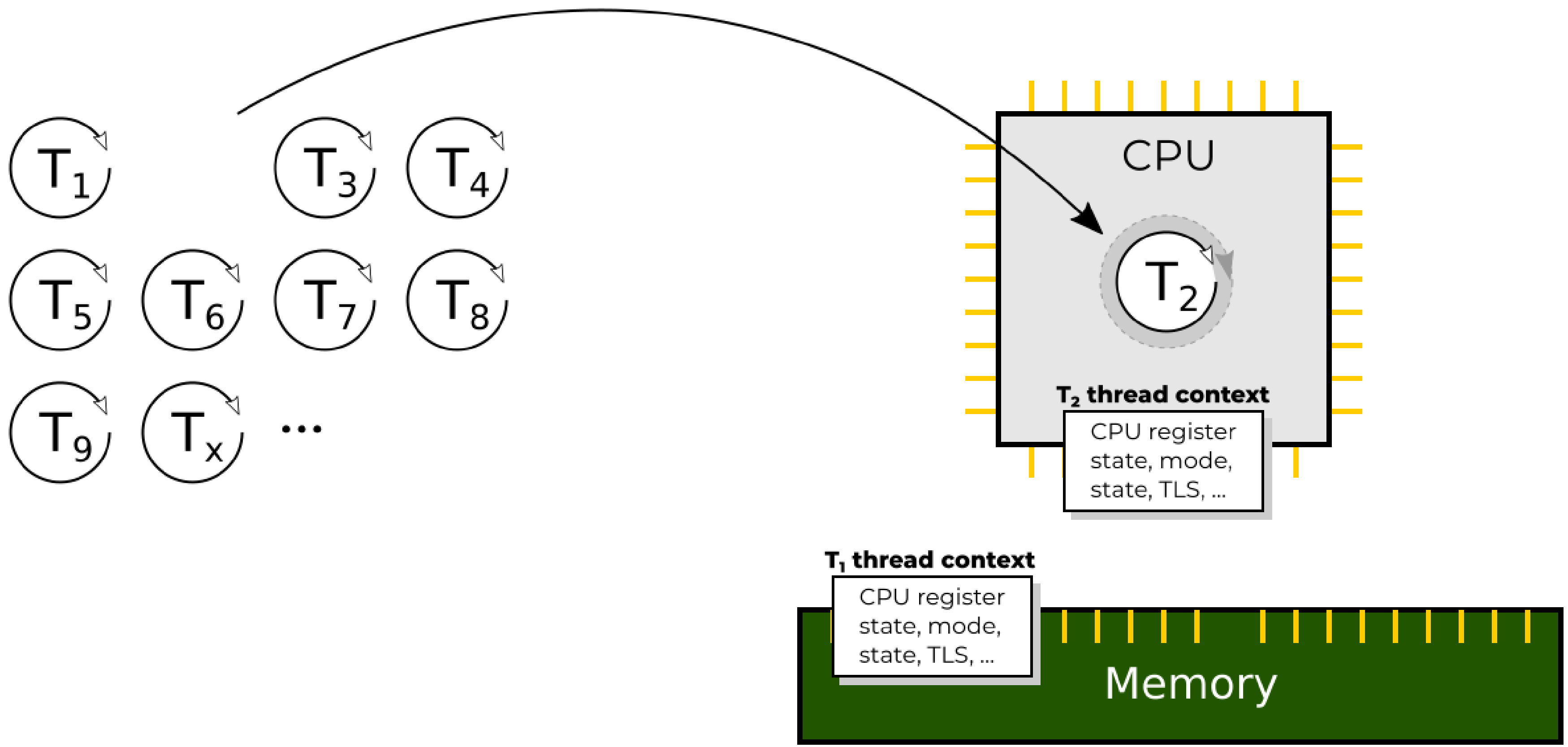


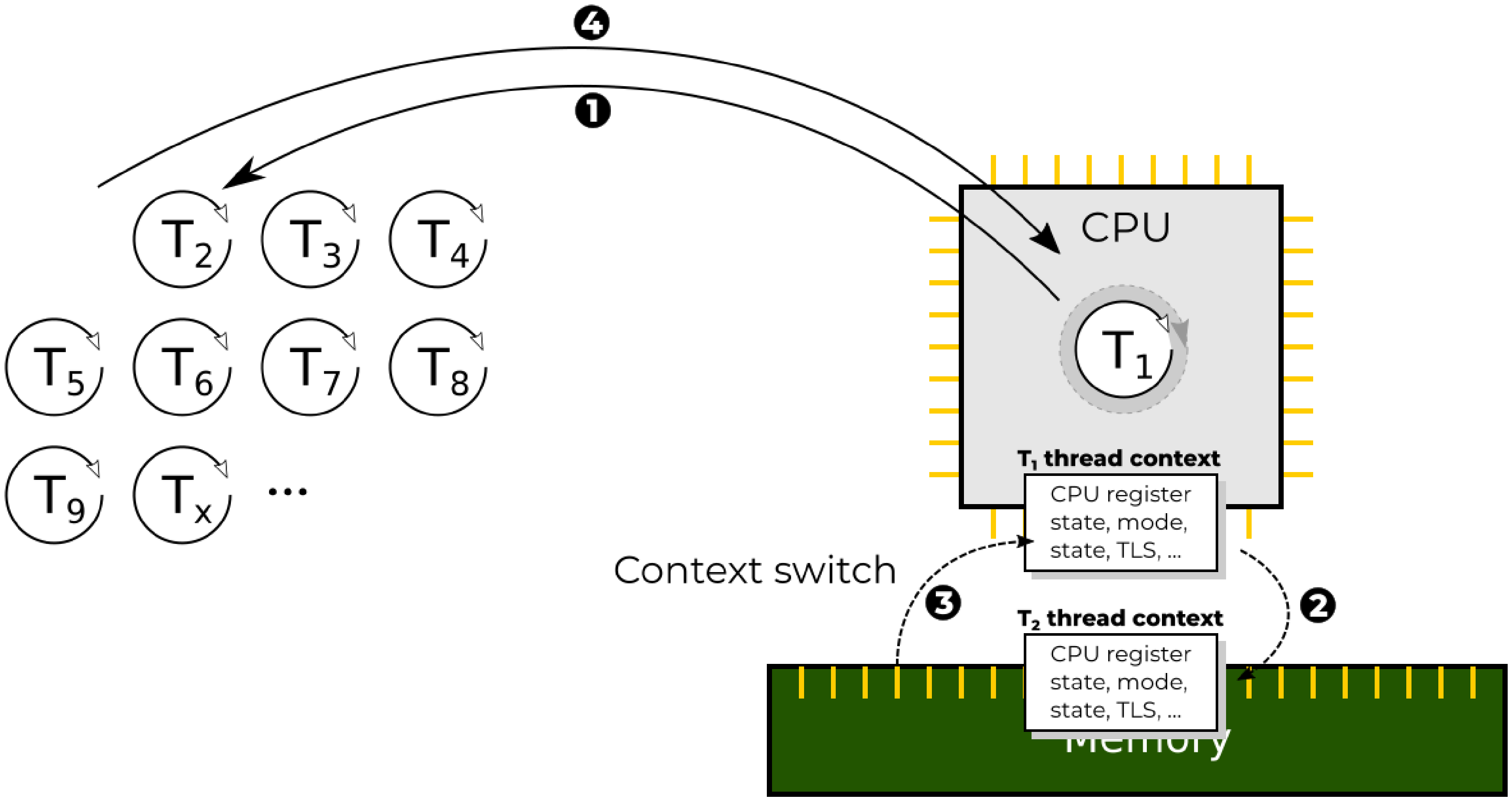
T_1 T_2 T_3 T_4
T_5 T_6 T_7 T_8
T_9 T_x ...
CPU
Memory


T_2 T_3 T_4
T_5 T_6 T_7 T_8
T_9 T_x ...
CPU
T_1
T_1 thread context
CPU register state, mode, state, TLS, ...
Context creation!
Memory



T_1 T_2 T_3 T_4
T_5 T_6 T_7 T_8
T_9 T_x ...
CPU
T_1 thread context
CPU register state, mode, state, TLS, ...
Memory


T1
T3
T4
T5
T6
T7
T8
T9
Tx
...
CPU
T2
T2 thread context
CPU register state, mode, state, TLS, ...
T1 thread context
CPU register state, mode, state, TLS, ...
Memory


❹
❶
T_2
T_3
T_4
T_5
T_6
T_7
T_8
T_9
T_x
...
CPU
T_1
T_1 thread context
CPU register state, mode, state, TLS, ...
Context switch
❸
❷
T_2 thread context
CPU register state, mode, state, TLS, ...
Memory

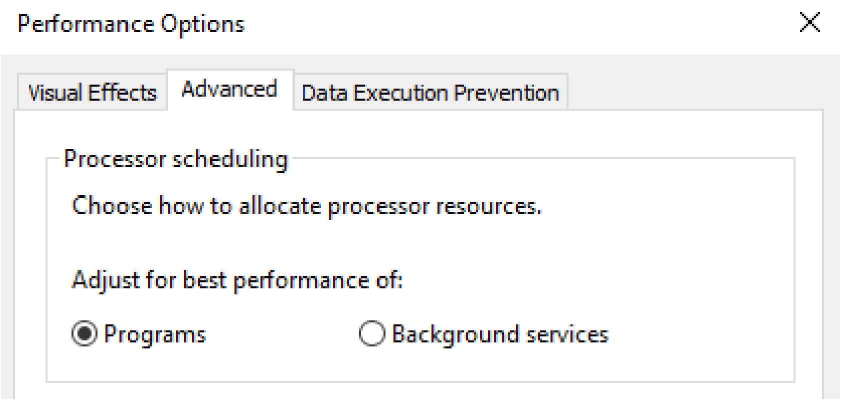
Many threads, single CPU

- *thread context* keeps the whole context:
  - CPU registers (including `IP`)
  - current mode (kernel/user)
  - two stacks (kernel/user)
  - *Thread Local Storage*
  - priority
  - state
  - ...
- expensive *Context switch* (4,000+ cycles, cache trashing)
- expensive creation

Many threads, single CPU

- how to decide which thread should run now?
  - we could do it using *round-robin* or *randomly*
  - we need something more sophisticated
- thread scheduler
  - on the operating system level (yes, there are Windows/Linux, versions differences)
  - system-wide - all threads from all processes in the same scheduling pool
  - **preemptive** - aggressively *kicking in/off* a thread from the CPU at any time
  - **quantum-based** - the thread runs for an amount of time called *quantum* **at its maximum!**
    - may be *preempted* by *higher priority* thread earlier
  - **priority-based** - at least one runnable thread always runs

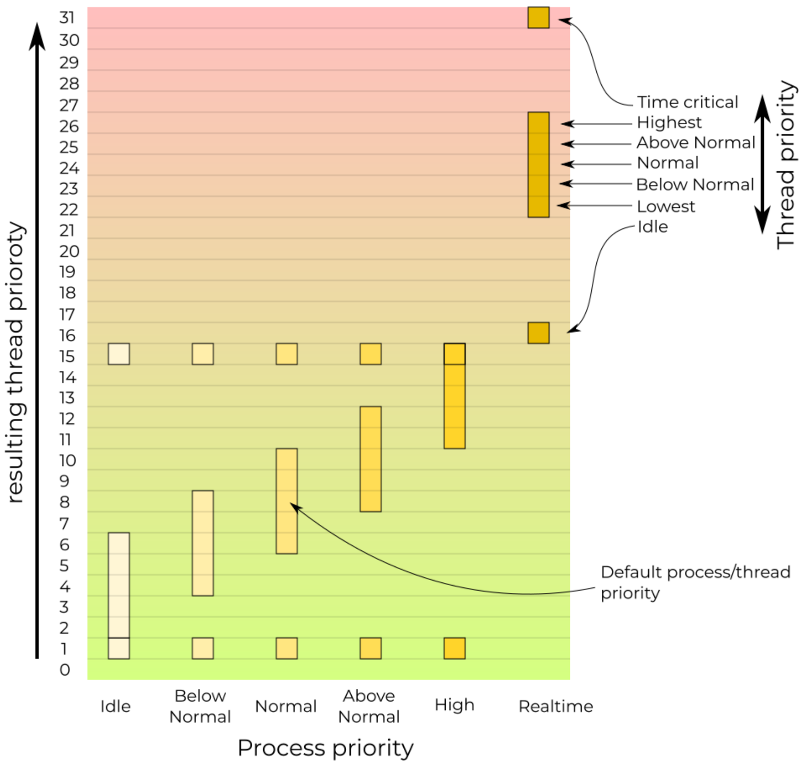
Quantum - system configuration



- *"Programs"* (desktop Windows default)
  - short, variable quantum (fe. foreground process, priority boosts)
  - ~2 clock intervals (more for foreground process)
  - maximize responsiveness
- *"Background services"* (Windows Server default)
  - long, fixed quantum
  - 12 clock intervals
  - minimize context switching
- immediate change - you can switch to *"Background services"* for long, night-running job
- more control under `HKLM\SYSTEM\CurrentControlSet\Control\PriorityControl\Win32PrioritySeparation` or job objects
- clock interval - ~15 ms

Thread priority - Windows

- number from 0 to 31
  - `0` reserved for a special so-called *zero page thread*
- based on the process priority - *Idle*, *Below normal*, ***Normal*** (default), *Above normal*, *High*, *Realtime*
- thread priorities: *Idle*, *Lowest*, *Below normal*, ***Normal*** (default), *Above normal*, *Highest*, *Time critical*



Thread priority

Overall process *priority class*:

```
using Process p = Process.GetCurrentProcess();
p.PriorityClass = ProcessPriorityClass.High;
```

Specific thread *priority*:

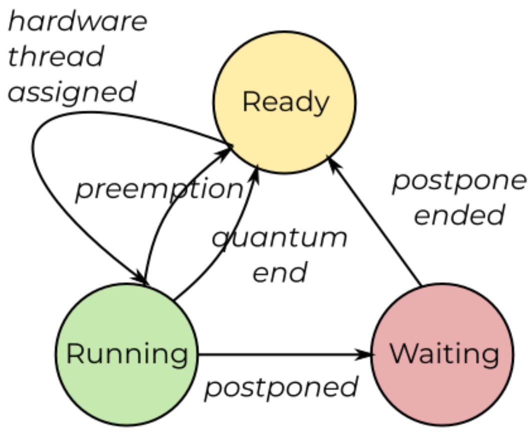
```
Thread thread = ...;
thread.Priority = ThreadPriority.AboveNormal;
```

Thread priority

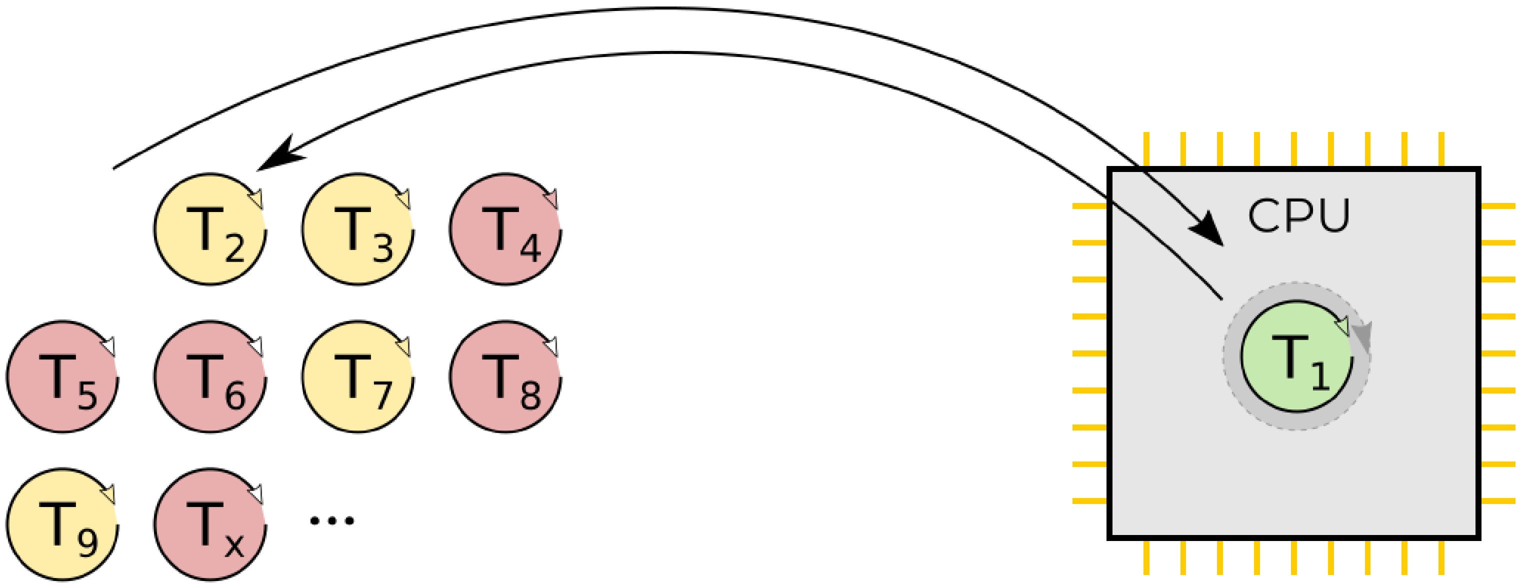
- priority boost - temporarily increases some threads priority
  - *Ready* thread not running for some time - avoid priority-based starvation
  - lock owning thread (both exclusive or shared) - avoid lock starvation
  - scheduler events - mutex/semaphore released, thread resumed, ...
  - I/O completion - for thread waiting on I/O
  - UI input - processing windows messages (responsiveness)
- there is more...
  - media/games
  - ...

Thread state

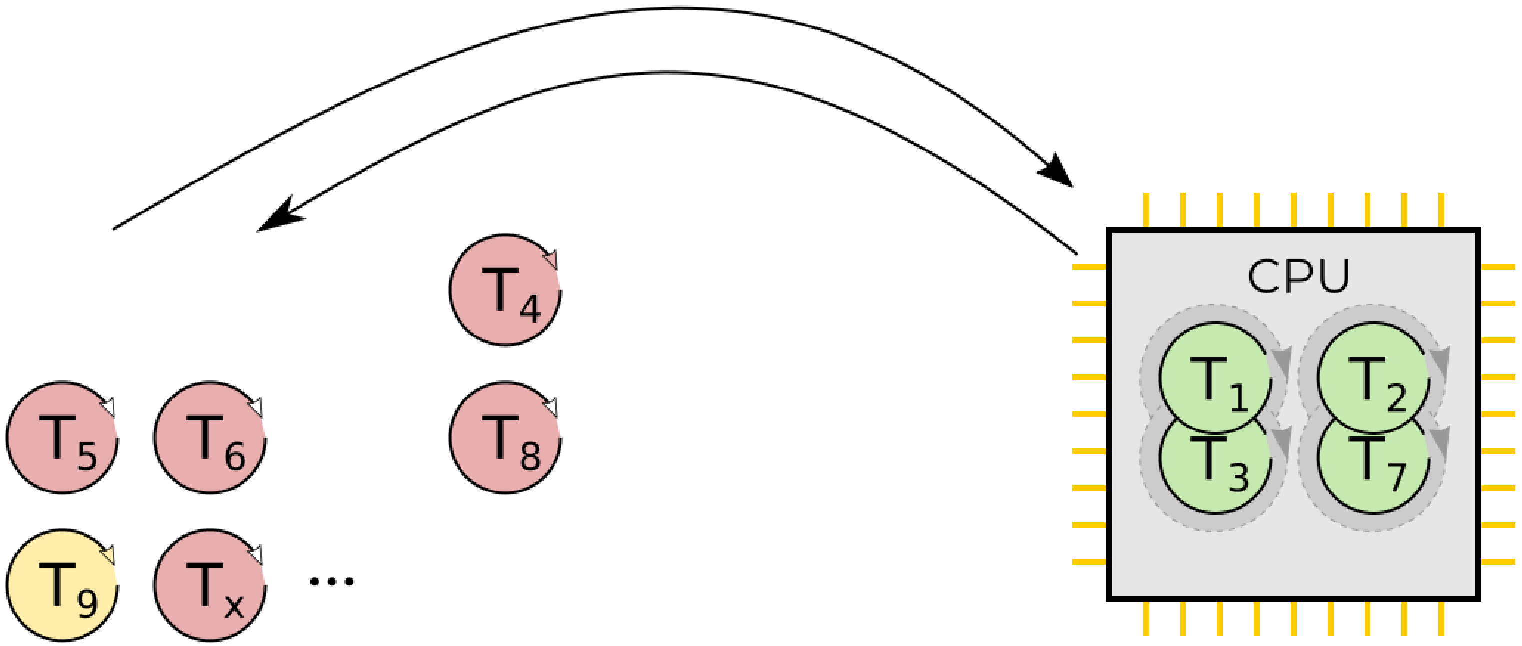
- `Ready` - can be executed immediately (but probably waiting for a hardware thread)
- `Running` - is executed on CPU (up to *quantum*)
- `Waiting` - needs to postpone execution (waits for something, has been suspended, …)



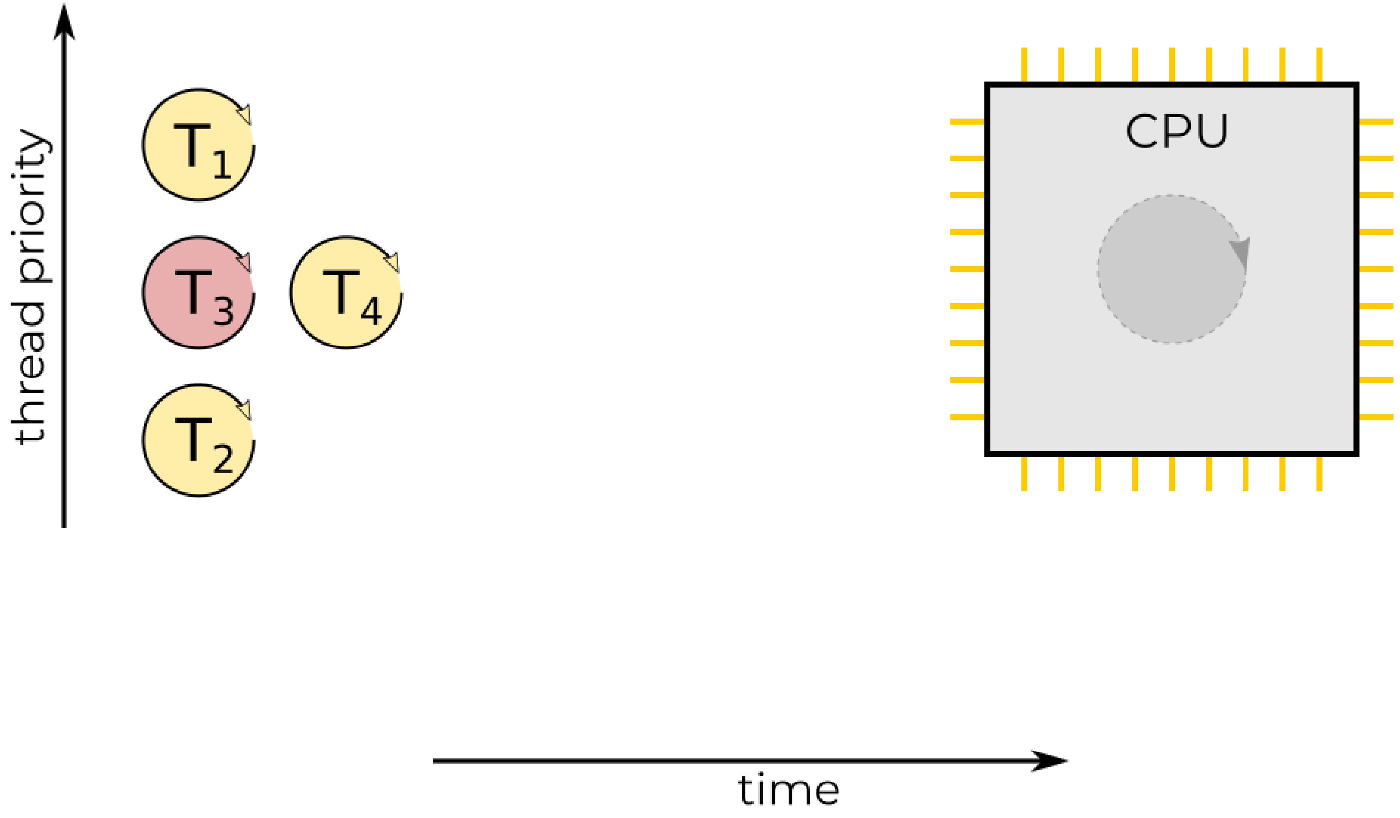
- `Standby`, `Terminated`, `Initialized`, … - more detailed states not needed for our consideration

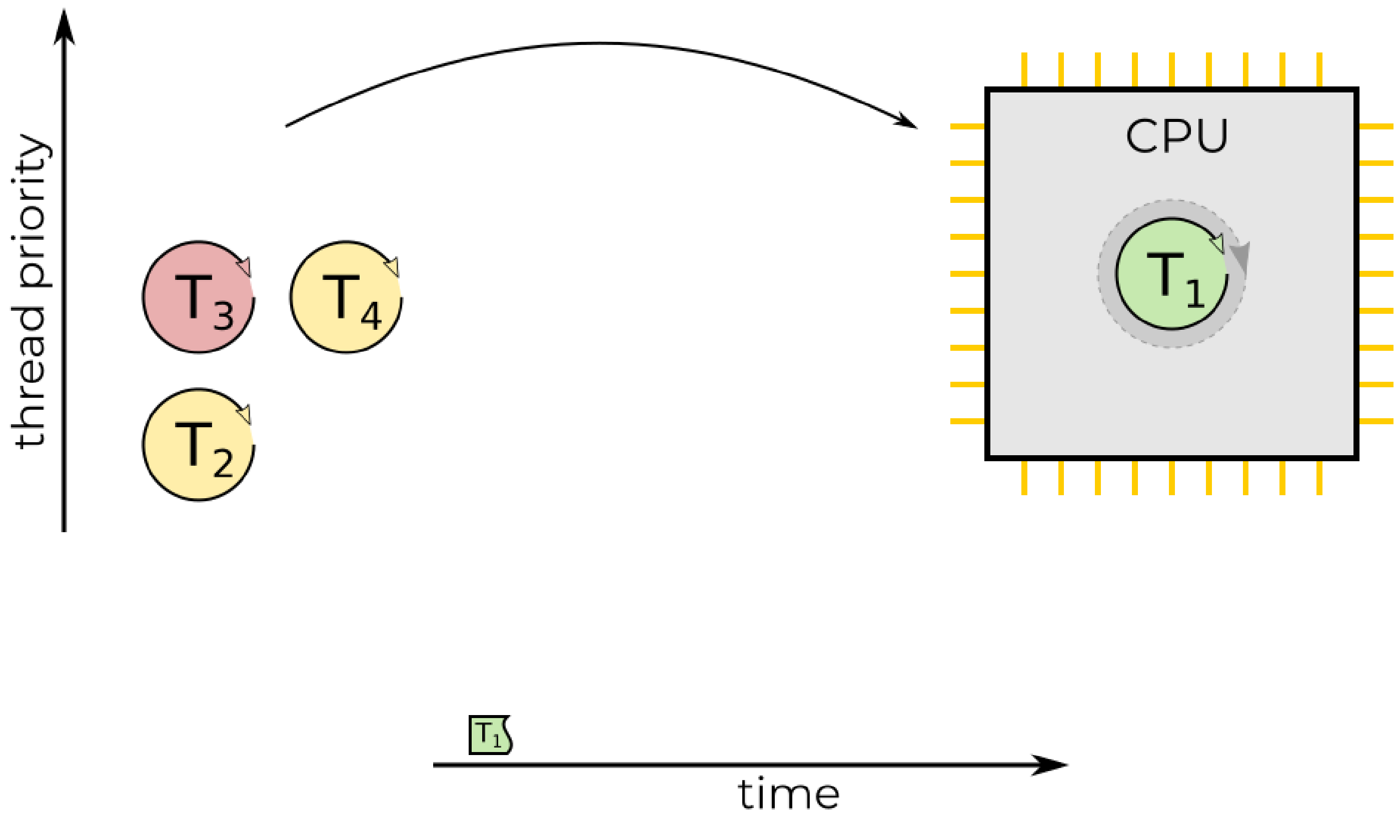


T2
T3
T4
T5
T6
T7
T8
T9
Tx
...
CPU
T1

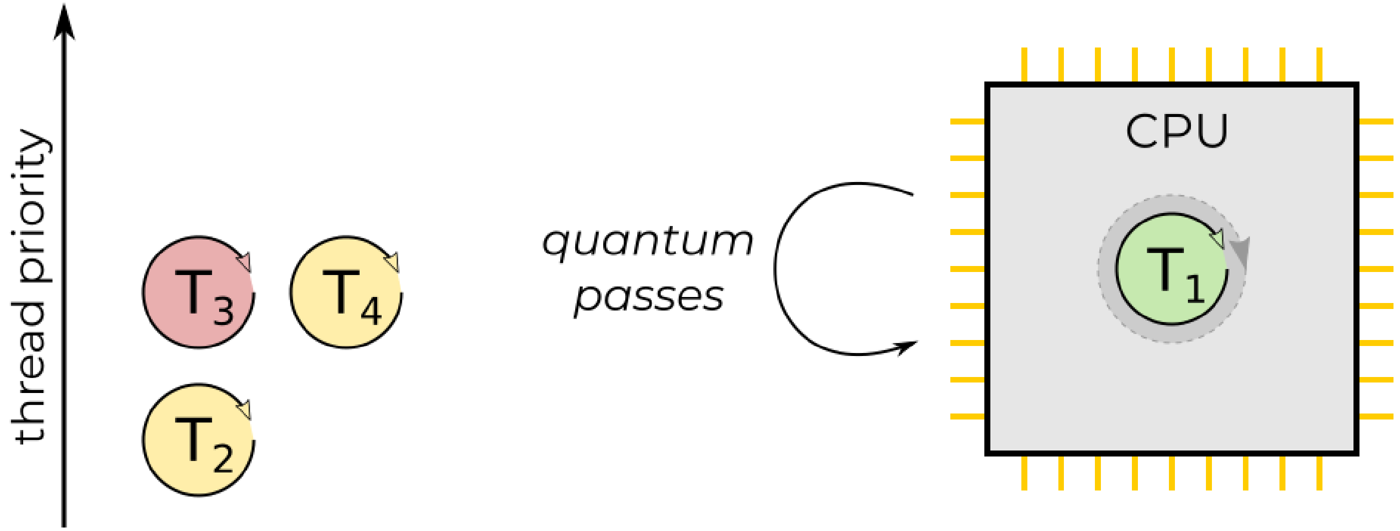

CPU
T_1
T_2
T_3
T_7
T_4
T_5
T_6
T_8
T_9
T_x
...

Threads scheduling

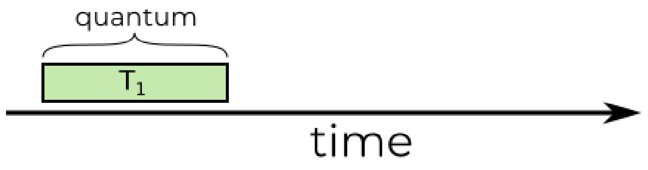

thread priority
T1
T3
T4
T2
CPU
time



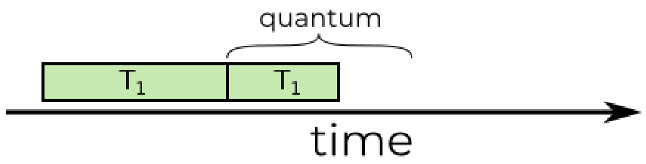
thread priority
T3
T4
T2
CPU
T1
T1
time

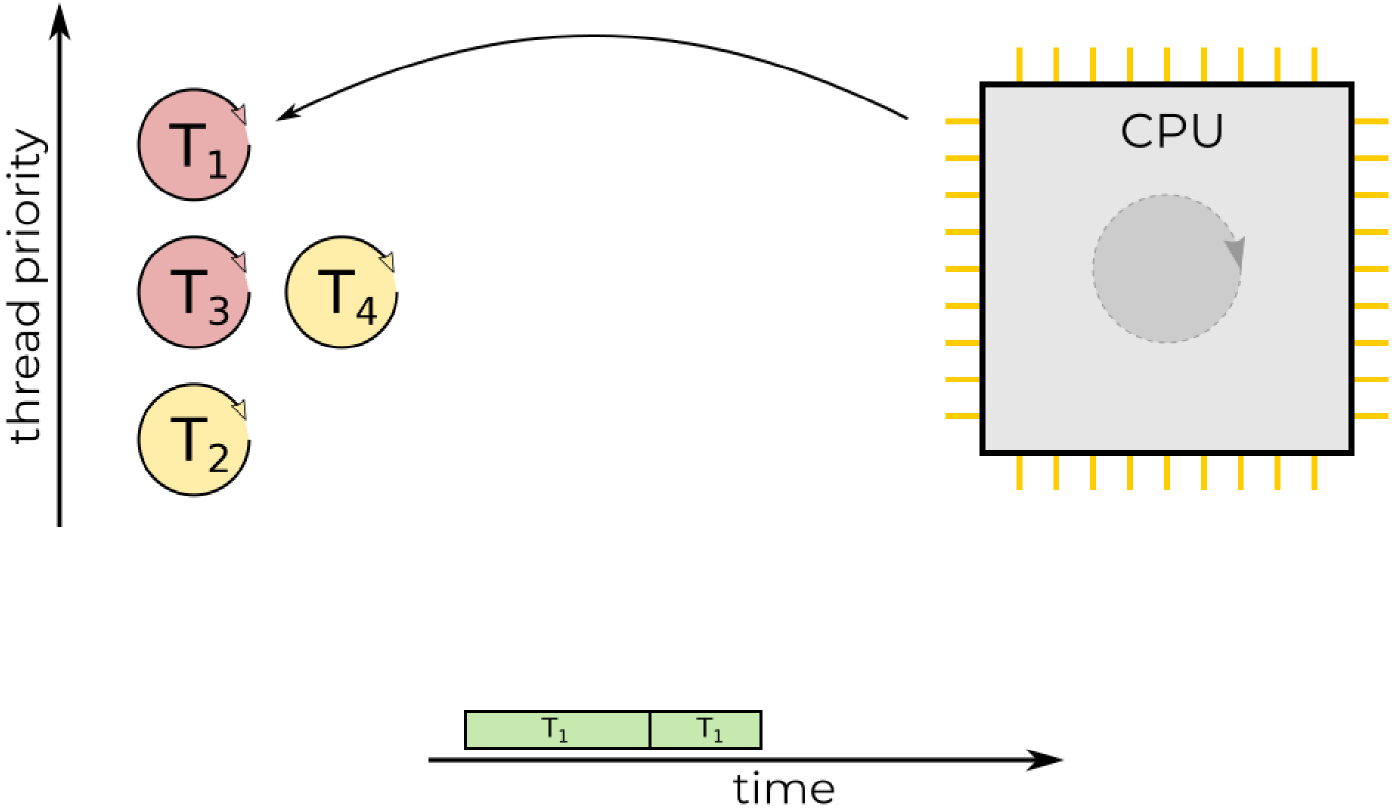


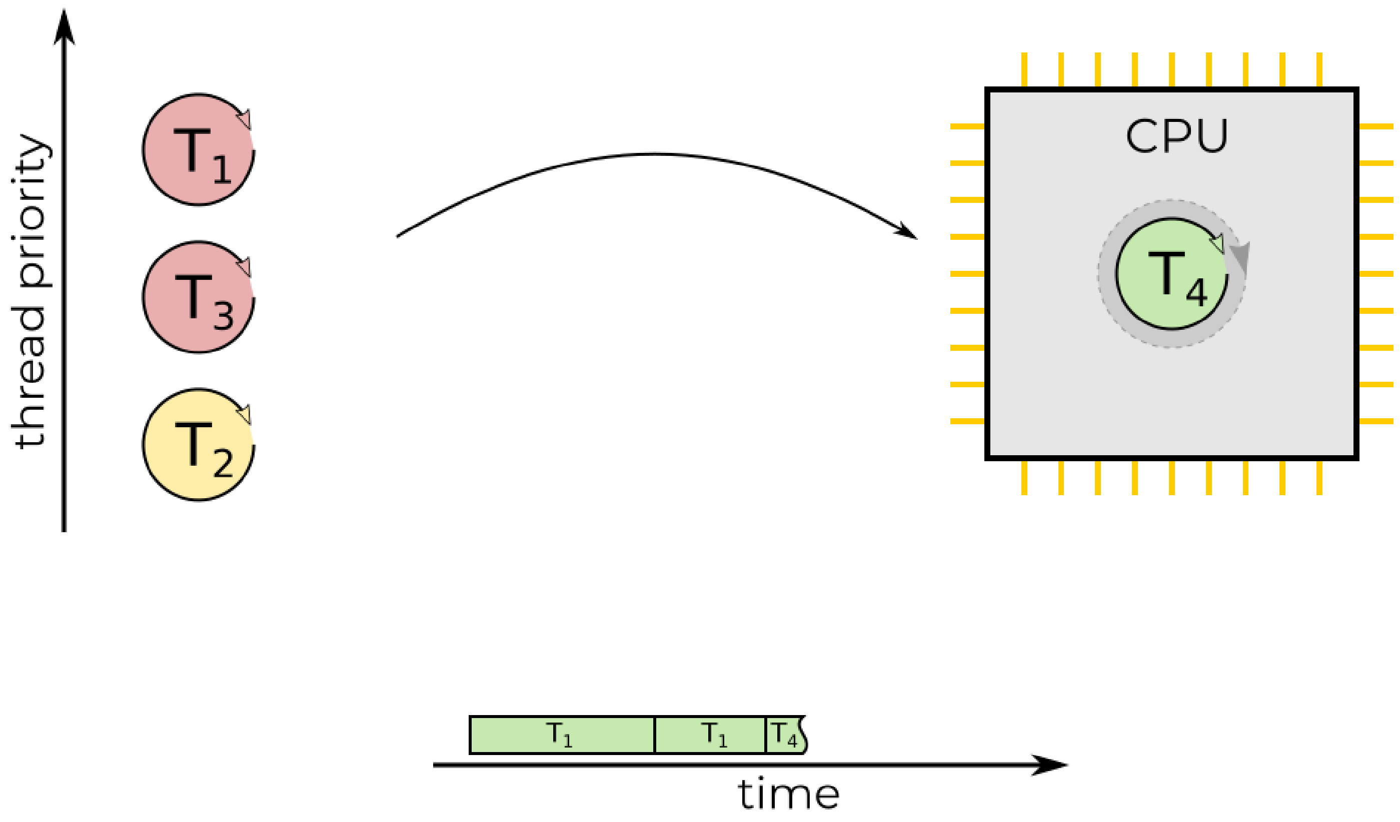
... but it is still the highest priority executable thread, so it continues...

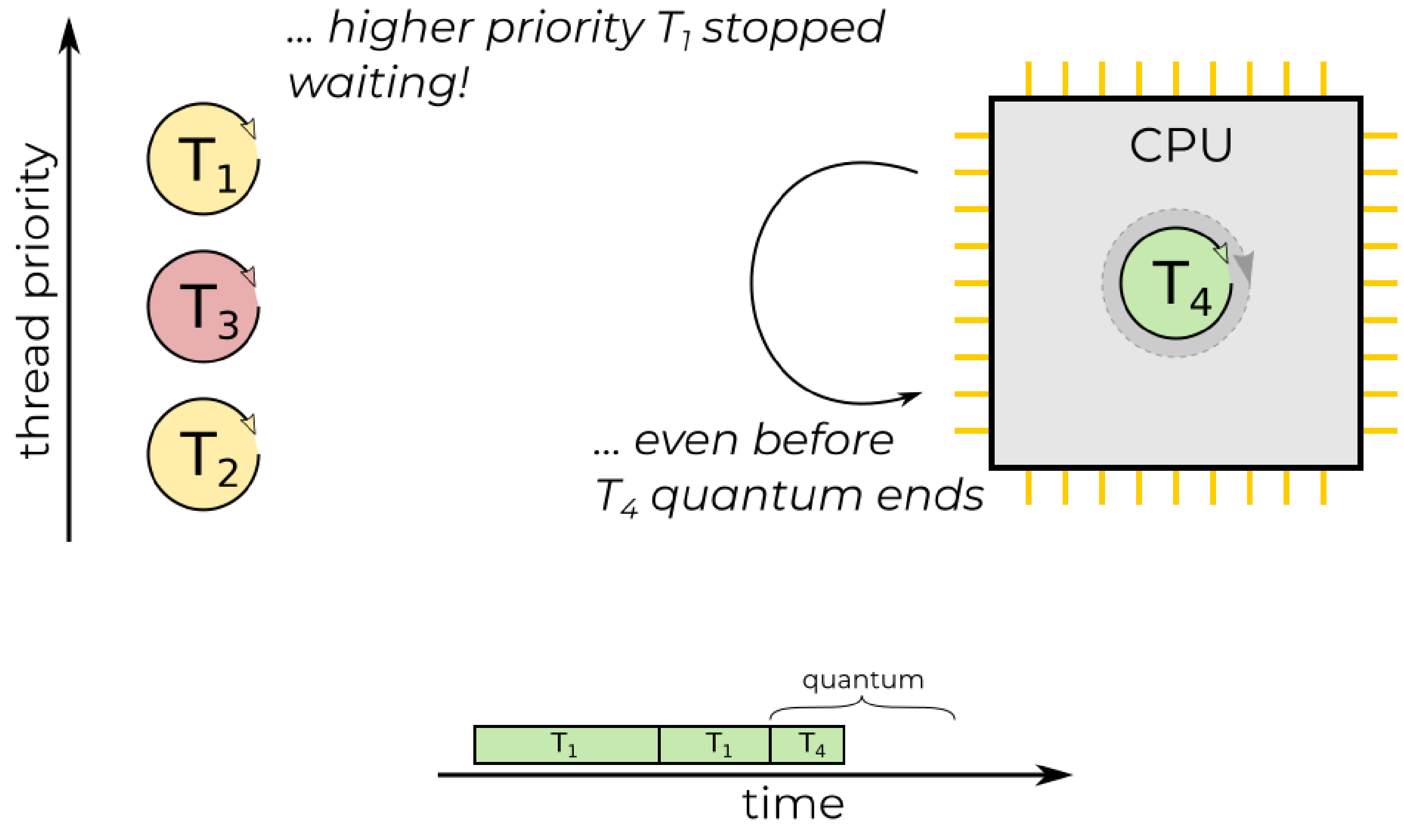


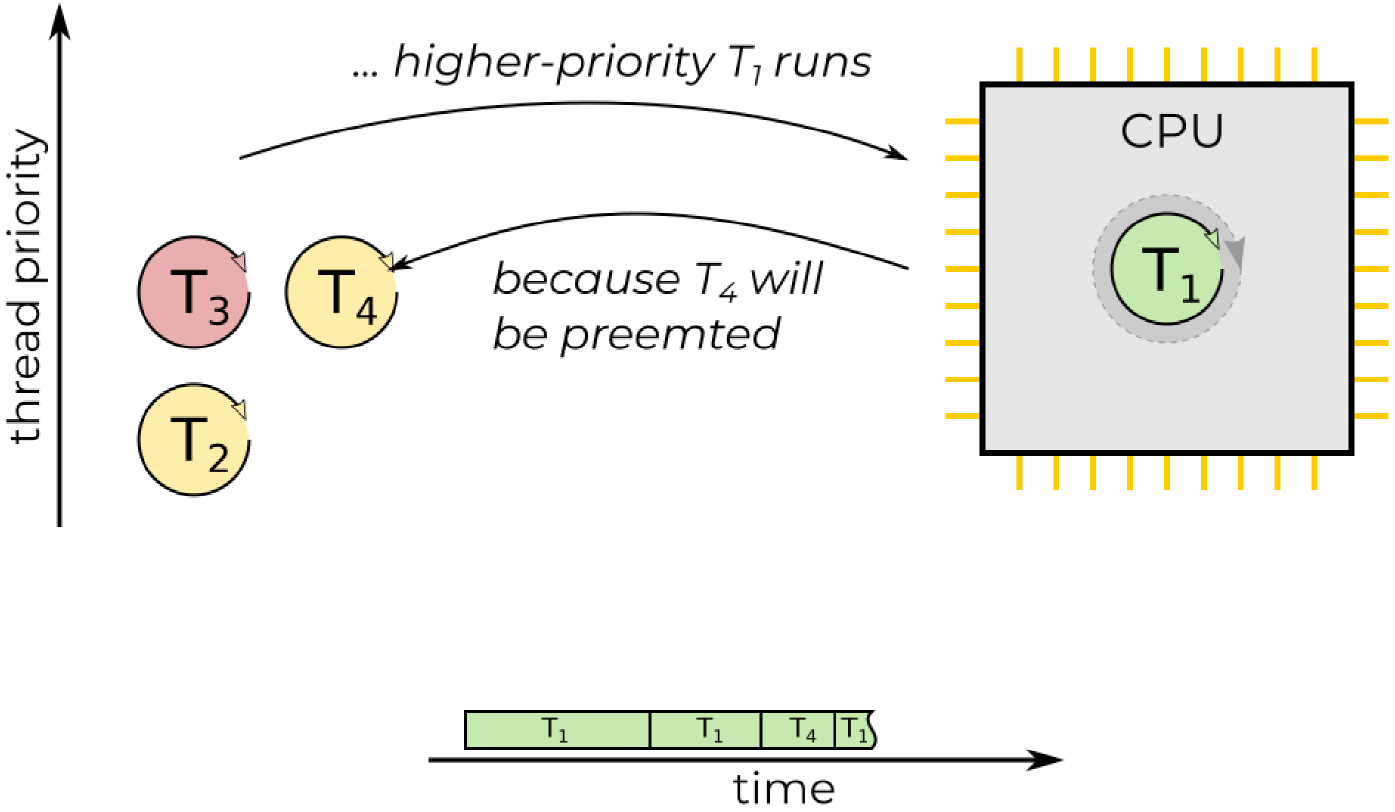
... T_1 started waiting on something, before quantum ends...



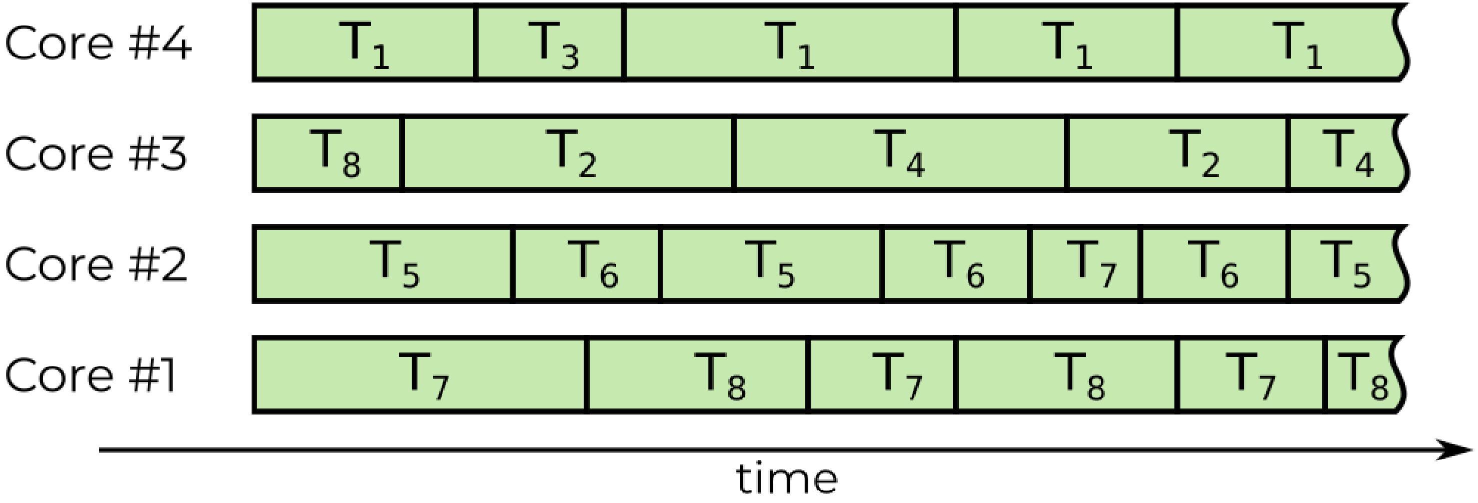

thread priority
T1
T3
T4
T2
CPU
T1
T1
time


thread priority
T1
T3
T2
CPU
T4
T1
T1
T4
time


... higher priority T_1 stopped waiting!
thread priority
T_1
T_3
T_2
CPU
T_4
... even before T_4 quantum ends
quantum
T_1
T_1
T_4
time



thread priority
... higher-priority T1 runs
CPU
T1
T3
T4
T2
because T4 will
be preemted
T1
T1
T4
T1
time


Core #4
T_1
T_3
T_1
T_1
T_1
Core #3
T_8
T_2
T_4
T_2
T_4
Core #2
T_5
T_6
T_5
T_6
T_7
T_6
T_5
Core #1
T_7
T_8
T_7
T_8
T_7
T_8
time

Threads summary

- *thread contention* - having too many **running** threads does not make sense
  - a lot of costly *context switches* & *cache misses*
  - CPU-bound code - ~ number of cores
- we can have many **waiting** threads
  - they do not cost a lot (sitting and `Wait`-ing)
  - still, not too much - *context switches* and *CPU thrashing*
  - I/O-bound - ~depends on "waiting ratio", hard to predict
- in the end
  - it would be great to have a wisely managed pool of threads for CPU-bound and I/O bound operations...

threads are evil - hard to implement, test and understand

Threads in .NET

- *native threads* - understand those software threads provided by the operating system
- unmanaged thread - executes unmanaged code (C/C++/Rust/...)
  - obviously native
  - including .NET runtime code (like GC-threads)!
  - not suspended during the GC
- managed thread - executes our .NET code (C#/F#/VB.NET/...)
  - still backed by a native thread (mostly...)
  - a managed thread that calls unmanaged code (ie. P/Invoke) is still managed
  - keeps even bigger "context" than *native threads*
    - thread local statics
- foreground vs background
  - the application **waits for foreground threads** before closing
  - when the application exits, **all background threads are forcibly stopped**
    - beware of cleanup - `Dispose` will be ignored, `finally` blocks are ignored
    - beware of "calculations" - they will be lost
    - solution: let foreground thread wait for background threads?
    - and yes... `ThreadPool` uses background threads
- special threads
  - finalizer thread
  - GC threads
  - debugger thread

"As soon as you type `new Thread()`, it's over; your project already has legacy code."

"Concurrency in C# Cookbook, 2nd Edition", Stephen Cleary

"As soon as you type `new Thread()`***, it's over; your project already has legacy code."***

"Concurrency in C# Cookbook, 2nd Edition", Stephen Cleary

still... it allows us to explain some basic concepts

Threads in .NET

`Thread.Start` & `Thread.Join`

```
Thread thread = new Thread(...); // Only managed representation, no underlying native yet
thread.IsBackground = true;      // Configure it as background thread
thread.Start(arg);               // Native assigned and started
thread.Join();                   // Current thread waits for `thread` to finish (blocking)
```

We can use the routine with a single parameter or not:

```
public delegate void ThreadStart();
public Thread (System.Threading.ThreadStart start);

public delegate void ParameterizedThreadStart(object obj);
public Thread (System.Threading.ParameterizedThreadStart start);
```

Threads in .NET - Exception Handling

Unhandled exception thrown from a thread will kill the entire application:

```
static void Main(string[] args)
{
    Thread thread = new Thread(DoWork)
    {
        IsBackground = true
    };
    thread.Start();
    Console.ReadLine();
}

static void DoWork()
{
    throw new NullReferenceException(); // ouch!
}
```

Threads in .NET - Exception Handling

`AppDomain.CurrentDomain.UnhandledException` can help to log/handle it but **does not prevent** killing an app:

```
static void Main(string[] args)
{
    AppDomain.CurrentDomain.UnhandledException += CurrentDomainOnUnhandledException;
    Thread thread = new Thread(DoWork)
    {
        IsBackground = true
    };
    thread.Start();
    Console.ReadLine();
}

private static void CurrentDomainOnUnhandledException(object sender, UnhandledExceptionEventArgs e)
{
    // Log & try to save work
}

static void DoWork()
{
    throw new NullReferenceException();
}
```

Threads - `Thread.Sleep`

- The current thread will not be scheduled for execution by the operating system for the specified amount of time
  - put into *Waiting* state (*WaitSleepJoin*)
  - resume to *Ready* state by a system timer
- the actual timeout will not be precise (especially for small timeouts)
  - limited to system timer resolution
    - default is 15.6 ms,
    - it may be modified by various programs system-wide (WPF, SqlServer, Chrome, ...)
  - obviously, also scheduling impact

We can check system timer resolution with [ClockRes] tool but it may be changed many times per second

```
> .\Clockres.exe

Clockres v2.1 - Clock resolution display utility
Copyright (C) 2016 Mark Russinovich
Sysinternals

Maximum timer interval: 15.625 ms
Minimum timer interval: 0.500 ms
Current timer interval: 1.000 ms
```

Threads - `Thread.Sleep`

Who does change system timer resolution?

```
powercfg /energy -duration 5
```

Information

Platform Timer Resolution:Timer Request Stack

The stack of modules responsible for the lowest platform timer setting in this process.

Requested Period	**10000**
Requesting Process ID	**16540**
Requesting Process Path	**\Device\HarddiskVolume4\Program Files (x86)\Google\Chrome\Application\chrome.exe**
Calling Module Stack	**\Device\HarddiskVolume4\Windows\System32\ntdll.dll**
	\Device\HarddiskVolume4\Windows\System32\kernel32.dll
	\Device\HarddiskVolume4\Program Files (x86)\Google\Chrome\Application\81.0.4044.122\chrome.dll
	\Device\HarddiskVolume4\Program Files (x86)\Google\Chrome\Application\chrome.exe
	\Device\HarddiskVolume4\Windows\System32\kernel32.dll
	\Device\HarddiskVolume4\Windows\System32\ntdll.dll

Platform Timer Resolution:Timer Request Stack

The stack of modules responsible for the lowest platform timer setting in this process.

Requested Period	**10000**
Requesting Process ID	**13852**
Requesting Process Path	**\Device\HarddiskVolume4\Program Files\Docker\Docker\resources\com.docker.proxy.exe**
Calling Module Stack	**\Device\HarddiskVolume4\Windows\System32\ntdll.dll**
	\Device\HarddiskVolume4\Windows\System32\kernel32.dll
	\Device\HarddiskVolume4\Program Files\Docker\Docker\resources\com.docker.proxy.exe

Threads - `Thread.Sleep` **&** `Thread.Yield`

Checking a condition in a tight loop could kill CPU (100%):

```
while (!condition) { ...no or little code... }
```

Give up some 'time-slice' for other threads:

```
while (!condition)
{
    Thread.PossiblityGiveUpSomeTimeSliceAPI();
}
```

A few possibilities (or combination of them):

```
Thread.Sleep(0); // 0ms?
Thread.Sleep(1); // 1ms?
Thread.Yield();  // ns?
Thread.SpinWait(20); // ?
```

Threads - `Thread.Sleep` **&** `Thread.Yield`

`Thread.Sleep(1)`

The slowest. **Forces** a context to switch, any process/CPU. Limited by the current system timer resolution & OS scheduling.

`Thread.Sleep(0)`

Switches if there's a *Ready* thread from any process/CPU. The current one remains *Ready* too. If there are none, current thread is not suspended at all

- it may lead to busy waiting (if no threads to switch to)
- before Windows Server 2003 it switched to the threads **of equal priority only**
  - it's dangerous because we are not sure what other threads are doing (and with what priorities)
  - ie. thread changing `condition` may have lower priority - a risk of starvation and inefficiency
  - starvation may be OS-specific (refer to *priority boosts* as a workaround)

Threads - `Thread.Sleep` **&** `Thread.Yield`

```
bool Thread.Yield()
```

Very fast. Give back time-slice to a *Ready* thread (with regular OS priority-based scheduling) but only from **the same physical CPU**.

Excellent diagnostic tool - inserting it may break/fix your code :)

Threads - `Thread.SpinWait`

`Thread.SpinWait(int iterations)`

- calls *X* number of times a special CPU instruction for spin waiting (`pause` on x86/x64 and `yield` on ARM64)
- *X* is normalized: `iterations` multiplied by a normalization factor because `pause` takes 14-150 CPU cycles (depending on architecture)
- used typically with exponential back-off of `iterations`

Threads - `Thread.SpinWait` internals

```
FCIMPL1(void, ThreadNative::SpinWait, int iterations)
{
    // If we're not going to spin for long, it's ok to remain in cooperative mode.
    // The threshold is determined by the cost of entering preemptive mode; if we're
    // spinning for less than that number of cycles, then switching to preemptive
    // mode won't help a GC start any faster.
    //
    if (iterations <= 100000)
    {
        YieldProcessorNormalized(iterations);
        return;
    }
    //
    // Too many iterations; better switch to preemptive mode to avoid stalling a GC.
    //
    HELPER_METHOD_FRAME_BEGIN_NOPOLL();
    GCX_PREEMP();
    YieldProcessorNormalized(iterations);
    HELPER_METHOD_FRAME_END();
}
FCIMPLEND
```

Threads - `Thread.SpinWait` internals

```
FORCEINLINE void YieldProcessorNormalized(const YieldProcessorNormalizationInfo &normalizationInfo,
                                          unsigned int count)
{
   // ...
   SIZE_T n = (SIZE_T)count * normalizationInfo.yieldsPerNormalizedYield;
    _ASSERTE(n != 0);
    do
    {
        System_YieldProcessor();
    } while (--n != 0);
}
```

`System_YieldProcessor` is:

- Intel pre-Skylake processor: measured typically **14-17 cycles per yield**
- Intel post-Skylake processor: measured typically **125-150 cycles per yield**

[Why Skylake CPUs Are Sometimes 50% Slower – How Intel Has Broken Existing Code]

Threads - Thread.SpinWait internals

```
YieldProcessor()
{
#if defined(HOST_X86) || defined(HOST_AMD64)
    __asm__ __volatile__(
        "rep\n"
        "nop");
#elif defined(HOST_ARM64)
    __asm__ __volatile__( "yield");
#else
    return;
#endif
}
```

PAUSE—Spin Loop Hint

Opcode	Instruction	Op/ En	64-Bit Mode	Compat/ Leg Mode	Description
F3 90	PAUSE	ZO	Valid	Valid	Gives hint to processor that improves performance of spin-wait loops.

Instruction Operand Encoding

Op/En	Operand 1	Operand 2	Operand 3	Operand 4
ZO	NA	NA	NA	NA

Description

Improves the performance of spin-wait loops. When executing a "spin-wait loop," processors will suffer a severe performance penalty when exiting the loop because it detects a possible memory order violation. The PAUSE instruction provides a hint to the processor that the code sequence is a spin-wait loop. The processor uses this hint to avoid the memory order violation in most situations, which greatly improves processor performance. For this reason, it is recommended that a PAUSE instruction be placed in all spin-wait loops.

An additional function of the PAUSE instruction is to reduce the power consumed by a processor while executing a spin loop. A processor can execute a spin-wait loop extremely quickly, causing the processor to consume a lot of power while it waits for the resource it is spinning on to become available. Inserting a pause instruction in a spin-wait loop greatly reduces the processor's power consumption.

Threads - `Thread.Sleep` **&** `Thread.Yield` **&** `Thread.SpinWait`

In fact, some experts make educated guesses about the combination of them that works, depending on what scenarios/architectures we want to cover.

`SpinWait` **type**

- *"`SpinWait` is just a little value type that encapsulates some common spinning logic."*
- A smart combination of `Thread.SpinWait`, `Thread.Sleep(0)`, `Thread.Sleep(1)` and `Thread.Yield`.

```
SpinWait wait = new SpinWait();
while (!condition) { wait.SpinOnce(); }
```

or even:

```
SpinWait wait = new SpinWait();
while (!p) {
   if (wait.NextSpinWillYield) { /* block! */ }
   else { wait.SpinOnce(); }
}
```

Threads - SpinWait internals

```
internal const int YieldThreshold = 10;          // When to switch over to a true yield.
private const int Sleep0EveryHowManyYields = 5; // After how many yields should we Sleep(0)?
internal const int DefaultSleep1Threshold = 20; // After how many yields should we Sleep(1) frequently?
private void SpinOnceCore(int sleep1Threshold)
{
   if ((_count >= YieldThreshold && ((_count >= DefaultSleep1Threshold) || (_count - YieldThreshold) % 2 == 0))
       || Environment.IsSingleProcessor)
   {
       if (_count >= sleep1Threshold && sleep1Threshold >= 0) {
           Thread.Sleep(1);
       }
       else {
           int yieldsSoFar = _count >= YieldThreshold ? (_count - YieldThreshold) / 2 : _count;
           if ((yieldsSoFar % Sleep0EveryHowManyYields) == (Sleep0EveryHowManyYields - 1)) {
               Thread.Sleep(0);
           }
           else {
               Thread.Yield();
           }
       }
   }
   else {
       int n = Thread.OptimalMaxSpinWaitsPerSpinIteration;
       if (_count <= 30 && (1 << _count) < n) { n = 1 << _count; }
       Thread.SpinWait(n);
   }
   _count = (_count == int.MaxValue ? YieldThreshold : _count + 1); // Increment the spin counter.
}
```

Threads - spinning and waiting

Materials:

- yieldprocessornormalized.cpp
- How is Thread.SpinWait actually implemented?
- Priority-induced starvation: Why Sleep(1) is better than Sleep(0) and the Windows balance set manager

Threads - termination

`Thread.Abort`

Raises out-of-bound `ThreadAbortException`.

`Thread.Interrupt`

Raises `ThreadInterruptedException` when a thread is in `WaitSleepJoin` state (but where?!)

Threads - termination

`Thread.Abort`

Raises out-of-bound `ThreadAbortException`.

`Thread.Interrupt`

Raises `ThreadInterruptedException` when a thread is in `WaitSleepJoin` state (but where?!)

Simple NO!

Threads - termination

`Thread.Abort`

Raises out-of-bound `ThreadAbortException`.

`Thread.Interrupt`

Raises `ThreadInterruptedException` when a thread is in `WaitSleepJoin` state (but where?!)

Simple NO!

And they are not supported on .NET Core

Threads - cancellation

Instead of terminating a thread out-of-bound, make it cooperative (*"should I end now?"*):

- create `CancellationTokenSource` that has "write rights" (`Cancel`)
- pass its `.Token` property of type `CancellationToken` to cooperative operations that will observe it

```
using CancellationTokenSource cts = new CancellationTokenSource();
cts.Cancel();
// or
cts.CancelAfter(2000);
// or
using CancellationTokenSource cts = new CancellationTokenSource(millisecondsDelay: 2000);

DoSomeWork(cts.Token);
```

and:

```
void DoSomeWork(CancellationToken token)
{
   token.ThrowIfCancellationRequested();
   // or
   token.IsCancellationRequested
}
```

Threads - cancellation

```
CancellationToken.Register (Action callback)
CancellationToken.Register (Action<object> callback, object state);
CancellationToken.Register (Action callback, bool useSynchronizationContext);
CancellationToken.Register (Action<object> callback, object state, bool useSynchronizationContext);
```

- registers a delegate that will be called when this token is cancelled
- returns `CancellationTokenRegistration` - with `Unregister` method, and it is `IDisposable`!
- if the token is already cancelled, delegate is immediately, synchronously run
- we can pass the `state` and/or capture `SynchronizationContext`

```
var cts = new CancellationTokenSource();
cts.Token.Register(() => Thread.Sleep(1000));
```

Beware that `Cancel` executes registrations **synchronously**!

```
cts.Cancel(); // It takes 1 second because of the registered callback
```

Threads - simple coordination

- `AutoResetEvent` - signals one thread and closes (resets)
- `ManualResetEvent`/`ManualResetEventSlim` - signals many threads and we need to close it

We create such a "flag" (to share between threads):

```
AutoResetEvent autoEvent = new AutoResetEvent(false);
```

One or more threads are waiting for a flag to be *set* (blocking wait):

```
autoEvent.WaitOne();
```

And "the work is done" one (or more) signals the flag to be *set*:

```
autoEvent.Set();
```

which will wake up one of the threads blocked by `.WaitOne()` (in case of `AutoResetEvent`)

Note: Remember to cleanup:

```
autoEvent.Close(); // or AutoEvent.Dispose();
```