LECTURE 22

PARALLEL COMPUTATION

0. PARALLELISM THAT DOES NOT REQUIRE PROGRAMMER INTERVENTION

Pipelines

- CPU pipelines can be viewed as implementing some form of parallelism

  in the sense that multiple executions are being executed simultaneously

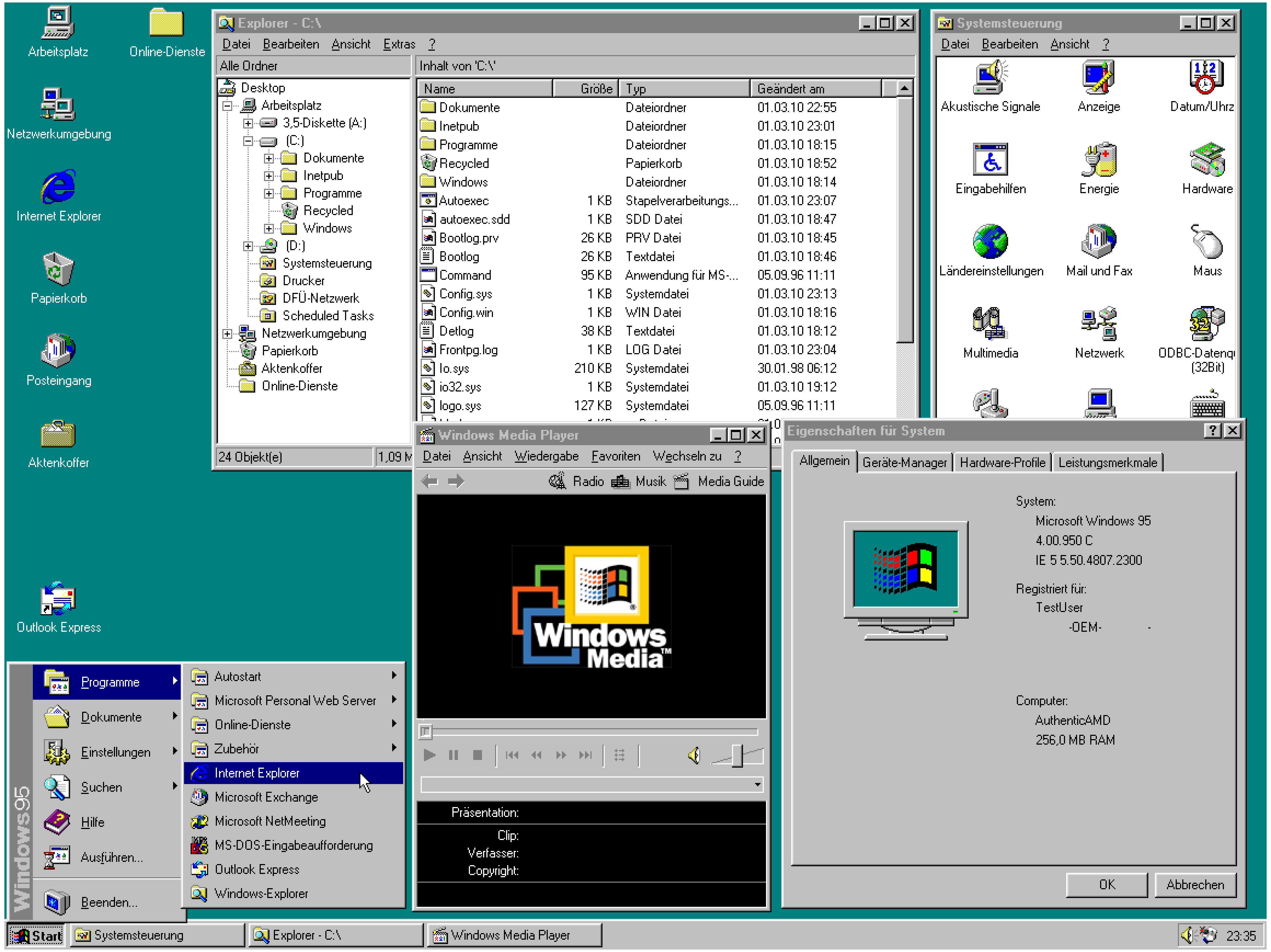
- For example, one instruction's arithmetic is performed (in an ALU)

  while the next is being decoded

- However, from the programmer's perspective,

  everything must happen **as if** there was no parallelism at all

Multitasking

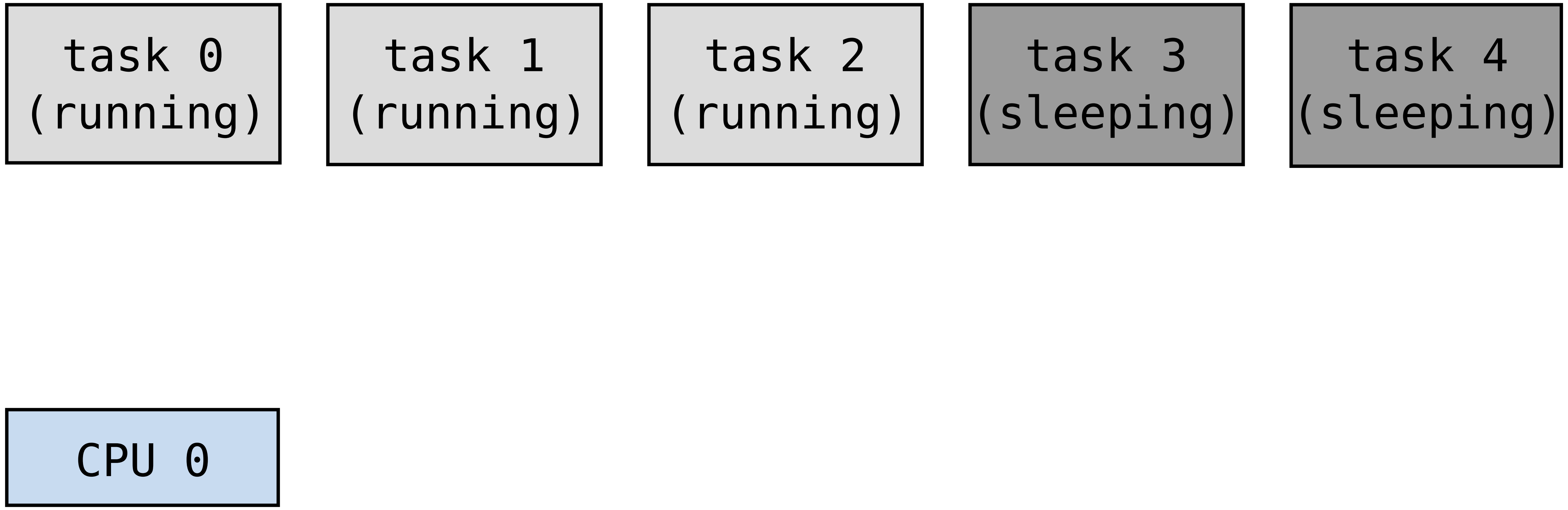


Multitasking

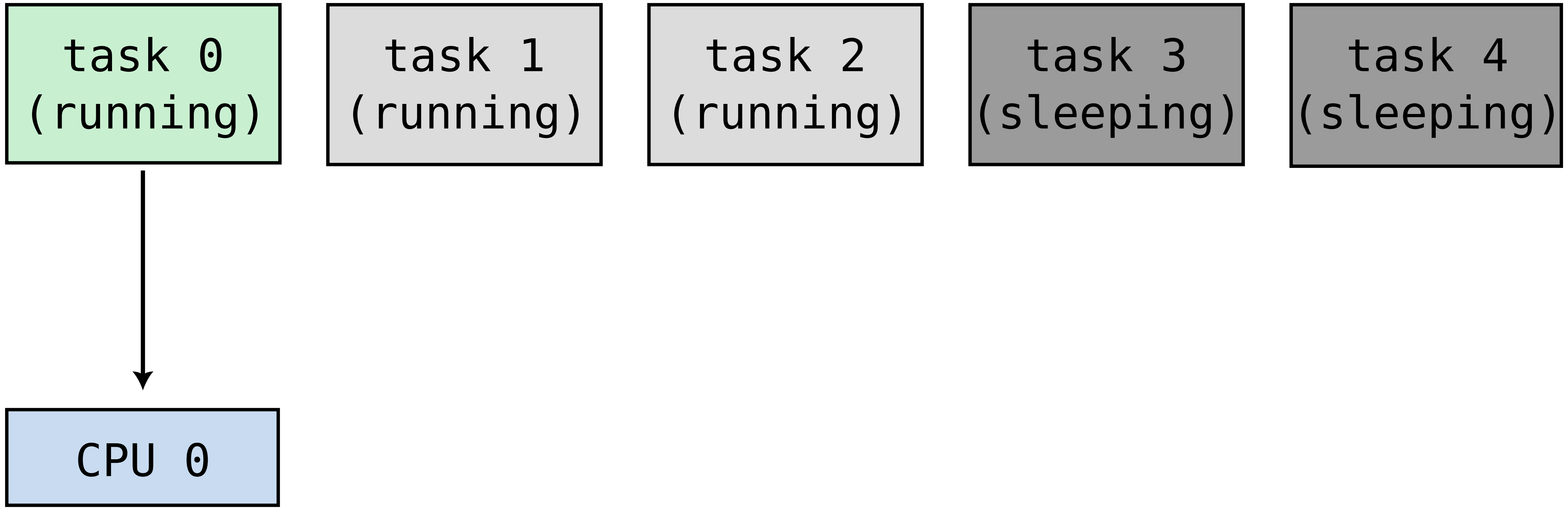
- Multitasking allows multiple executables to run “simultaneously”
  (even on a single processor)

- Regularly, the **scheduler** (part of the OS kernel) decides which **task** gets to run on a processor.

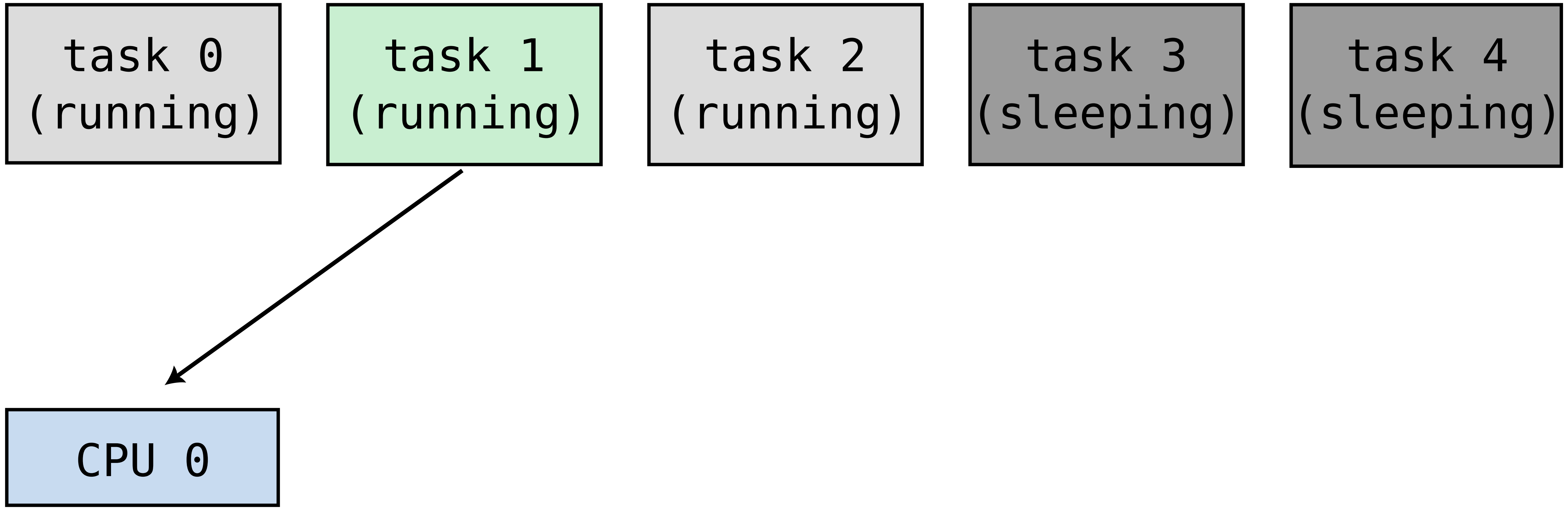
Multitasking on a single-core processor



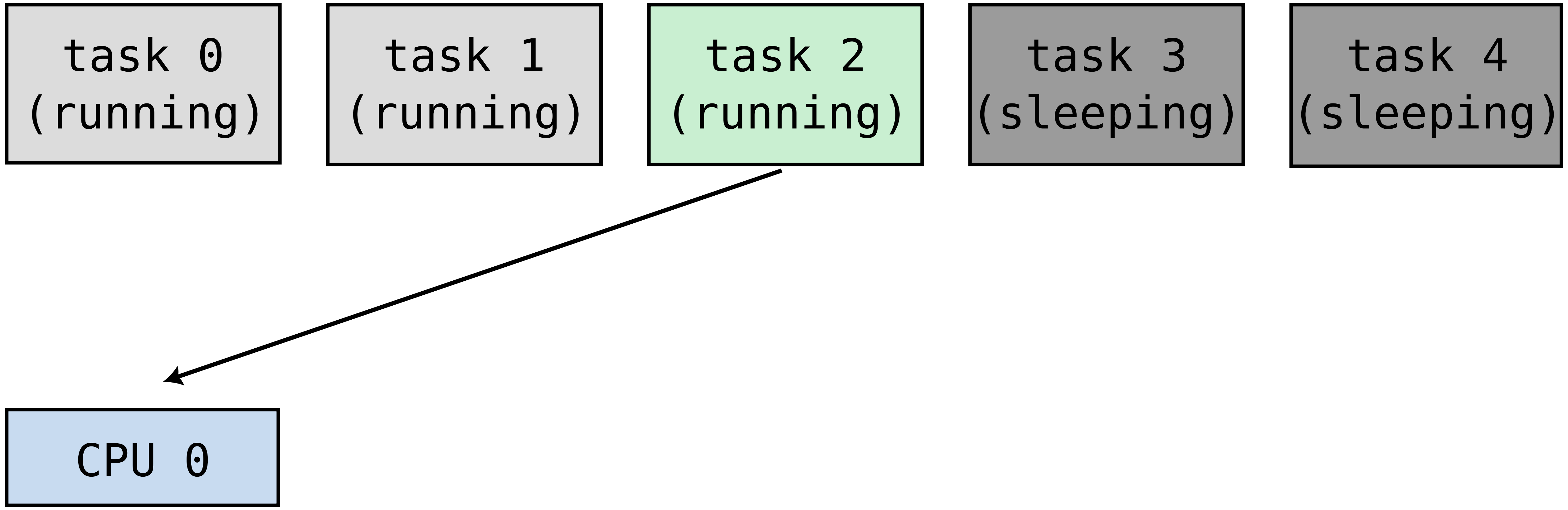
Multitasking on a single-core processor



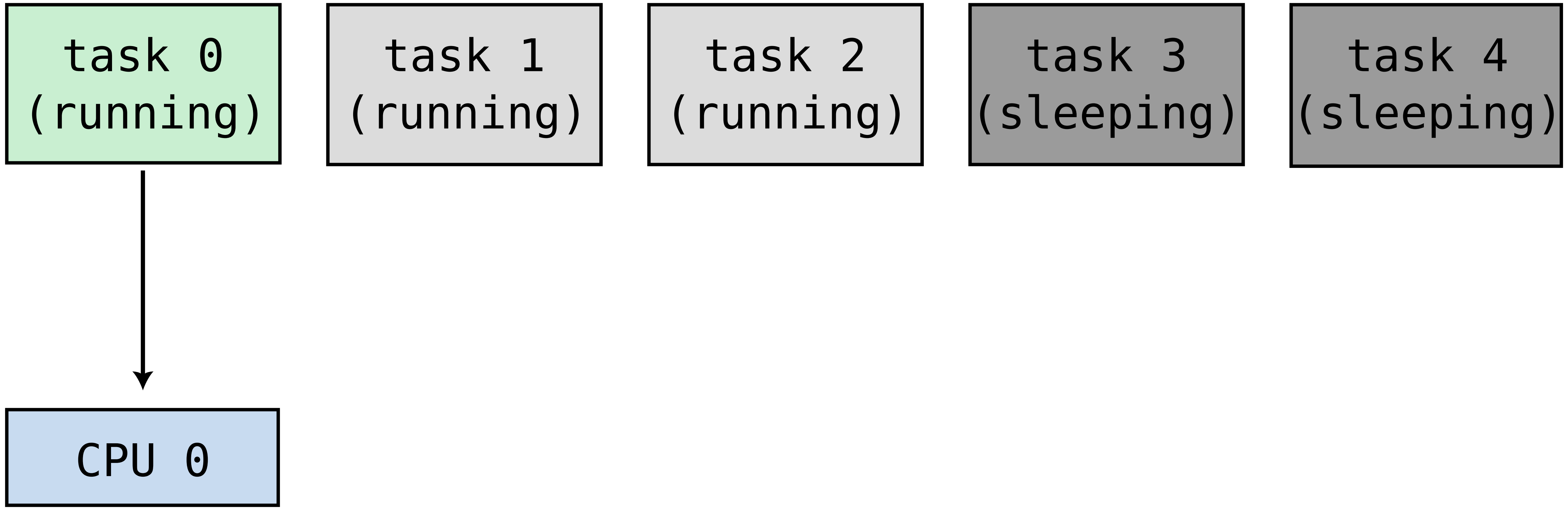
Multitasking on a single-core processor



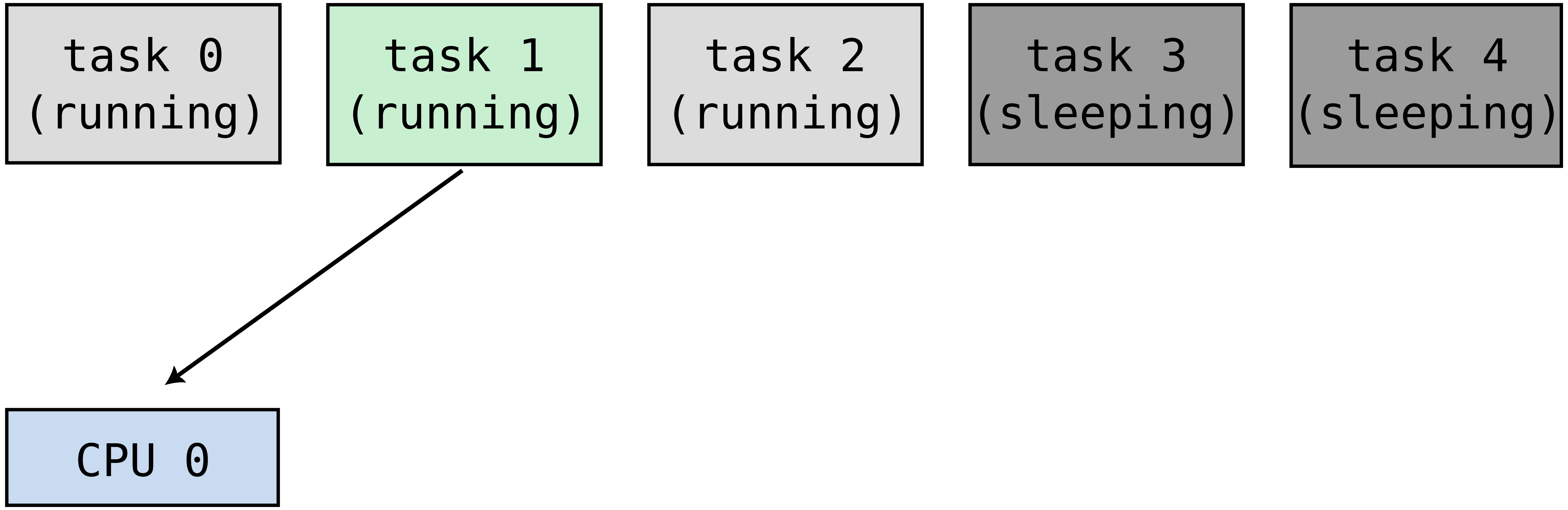
Multitasking on a single-core processor



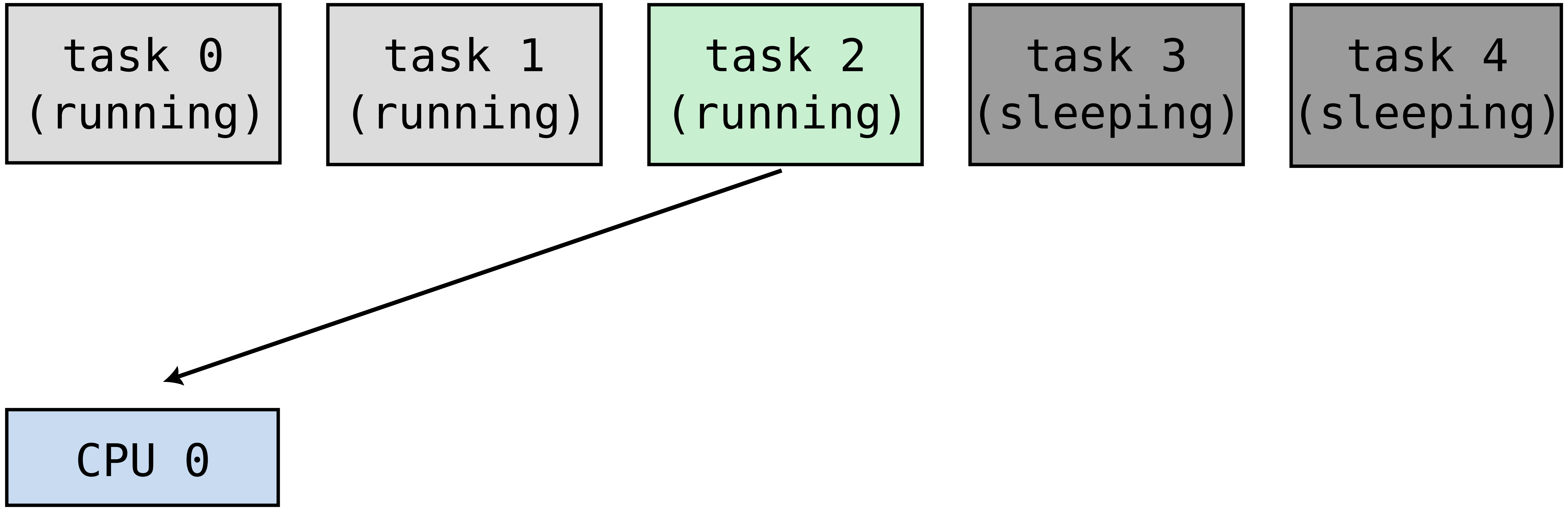
Multitasking on a single-core processor



Multitasking on a single-core processor



Multitasking on a single-core processor



- The scheduler is called:
  - at regular intervals f times per second, by default:
    - Linux: f = 1000 Hz (> see CONFIG_HZ)
    - MacOS: f = 100 Hz (> see sysctl kern.clockrate)
    - Windows 10: f = 64 Hz (> see timeBeginPeriod())
  - when an task performs a system call (`open()`, `write()`, `exit()`, ...)
  - when a "hardware interrupt" happens:
    - keyboard received a keypress
    - network device received data
    - storage device finished writing
    - sound/video device ready to receive next buffer
    - ...

Preemptive multitasking

- When the scheduler decides to interrupt a **running** process (e.g. to run another)
  - the process is said to "**preempted**"
  - it becomes "**runnable**"

- When a process executes a system call,
  - it starts "**sleeping**"
  - after the requested operation is performed,
    - in some cases, it will **run** again
    - in other cases, it becomes **runnable** and will only run when a CPU is available
  - many system calls can take a long time to perform ("**blocking**" system calls):

    `read(), write(), recv(), send()`

Preemptive multitasking

- At any given time, most tasks are **sleeping**
  - waiting for data (e.g. from network)
  - waiting for user interaction (e.g. keyboard or touch input)
  - waiting on a timer (tasks that run at regular interval)

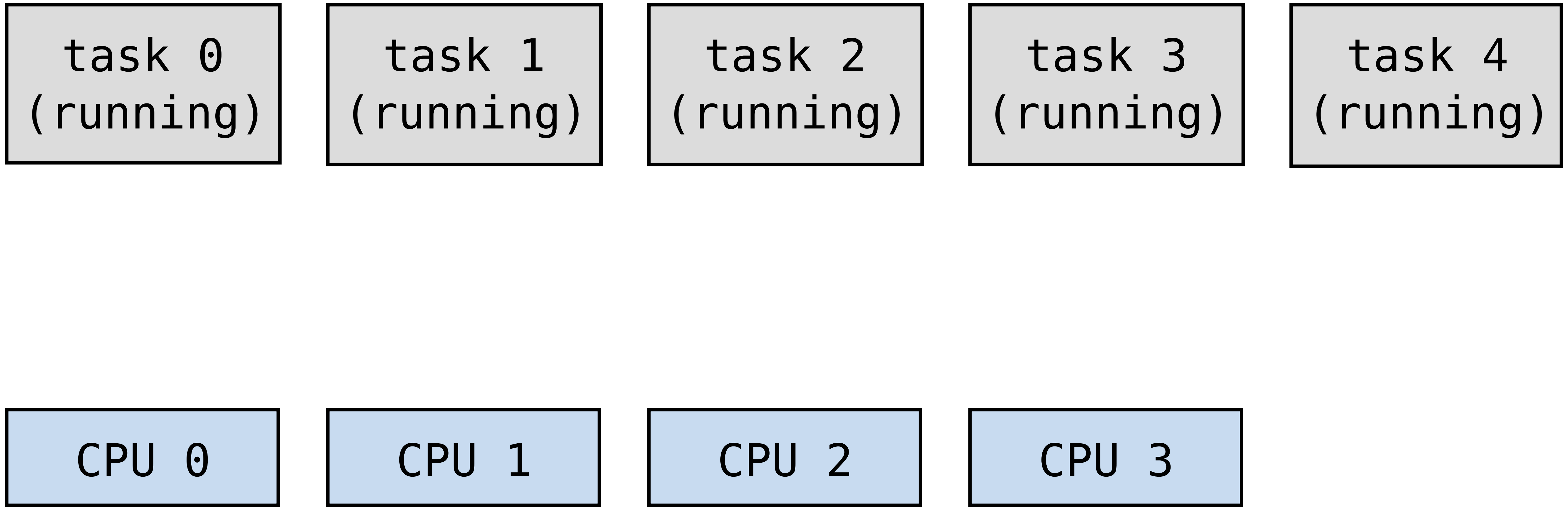
- The only tasks that are normally **running**/**runnable** are those performing CPU-intensive operations
  - graphics rendering
  - audio/video/data compression and decompression
  - computations
  - etc.

poirrier@lpn:~

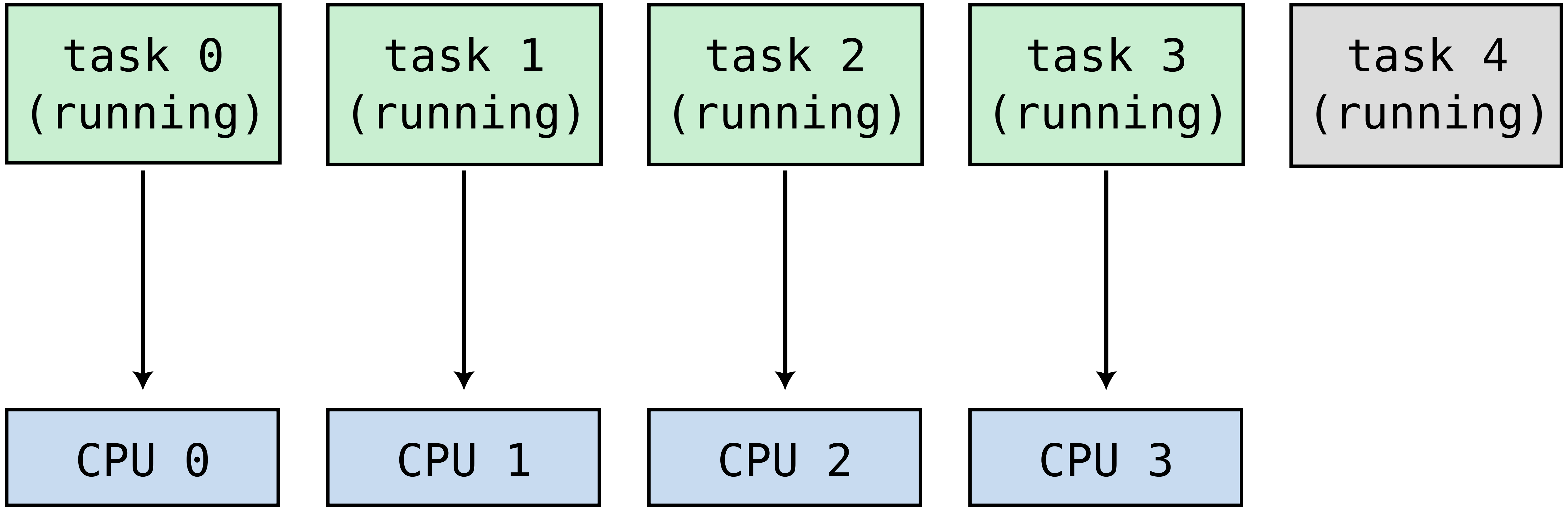
```
    0[||||                                           5.8%]   4[|                                         0.6%]
    1[||                                             1.3%]   5[                                          0.0%]
    2[||                                             1.3%]   6[|||                                       3.8%]
    3[||                                             1.3%]   7[|                                         0.6%]
  Mem[|||||||||||||||||||||||||||||||||||||||| 1.40G/15.3G]  Tasks: 126, 511 thr, 144 kthr; 1 running
  Swp[                                             0K/0K]  Load average: 0.32 0.15 0.04
                                                          Uptime: 9 days, 10:07:17

  Main  I/O
    PID USER       PRI  NI  VIRT   RES   SHR S  CPU% MEM%   TIME+ ▽Command
   1435 poirrier    20   0 1959M  175M  122M S   3.9  1.1 18:58.21 /usr/libexec/Xorg -nolisten tcp -background none -seat
   1810 poirrier     9 -11  150M 55768  7804 S   0.0  0.3  6:49.86 /usr/bin/pipewire-pulse
   1681 poirrier     9 -11  131M 31496  8824 S   0.0  0.2  4:59.19 /usr/bin/pipewire
   1700 poirrier   -21   0  131M 31496  8824 S   0.0  0.2  4:58.31 /usr/bin/pipewire
   1598 poirrier    20   0 1421M  104M 76760 S   0.0  0.7  3:40.54 /usr/bin/lxqt-panel
   1812 poirrier   -21   0  150M 55768  7804 S   0.0  0.3  3:03.95 /usr/bin/pipewire-pulse
   1897 poirrier    20   0 1922M 93344 54472 S   0.0  0.6  2:40.11 /usr/libexec/evolution-calendar-factory
   1454 poirrier    20   0 1959M  175M  122M S   0.6  1.1  2:25.39 /usr/libexec/Xorg -nolisten tcp -background none -seat
    927 root        20   0  324M 21276 17060 S   0.0  0.1  1:30.11 /usr/sbin/NetworkManager --no-daemon
   1919 poirrier    20   0 1922M 93344 54472 S   0.0  0.6  1:05.63 /usr/libexec/evolution-calendar-factory
   1603 poirrier    20   0  4424  3392  3016 S   0.0  0.0  1:01.44 /usr/bin/xscreensaver -no-splash
   1607 poirrier    20   0  780M 54972 40776 S   0.0  0.3  1:00.00 /usr/bin/nm-applet
   1930 poirrier    20   0 1922M 93344 54472 S   0.0  0.6  0:49.96 /usr/libexec/evolution-calendar-factory
   1560 poirrier    20   0  173M 22176 14352 S   0.0  0.1  0:40.79 /usr/bin/openbox
   1841 poirrier    20   0  380M 10084  8868 S   0.0  0.1  0:36.60 /usr/libexec/goa-identity-service
    778 root        20   0  300M  8044  5864 S   0.0  0.1  0:35.18 /usr/libexec/upowerd
   1455 poirrier    20   0  973M 82540 67380 S   0.0  0.5  0:33.97 lxqt-session
   1861 poirrier    20   0  670M 41128 33056 S   0.0  0.3  0:32.32 /usr/bin/lxqt-powermanagement
 311360 poirrier    20   0  105G  263M  161M S   0.6  1.7  0:31.82 /usr/bin/evolution
   1851 poirrier    20   0  380M 10084  8868 S   0.0  0.1  0:30.14 /usr/libexec/goa-identity-service
 313221 poirrier    20   0 1796M  141M  100M S   0.0  0.9  0:28.89 kate ../documents/plan.md 17_bench.md
   1538 poirrier    20   0  973M 82540 67380 S   0.0  0.5  0:23.45 lxqt-session
 312905 poirrier    20   0 33.5G  250M  190M S   0.0  1.6  0:20.30 /opt/google/chrome/chrome --incognito build/17_bench.ht
 311414 poirrier    20   0 88.8G  175M  130M S   0.0  1.1  0:19.87 /usr/libexec/webkit2gtk-4.1/WebKitWebProcess 13 61
   1915 poirrier    20   0 1922M 93344 54472 S   0.0  0.6  0:17.95 /usr/libexec/evolution-calendar-factory
   1634 poirrier    20   0 1421M  104M 76760 S   0.0  0.7  0:16.85 /usr/bin/lxqt-panel
   1667 poirrier    20   0  780M 54972 40776 S   0.0  0.3  0:15.16 /usr/bin/nm-applet
   1594 poirrier    20   0 1356M  105M 81000 S   0.0  0.7  0:13.93 /usr/bin/pcmanfm-qt --desktop --profile=lxqt
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

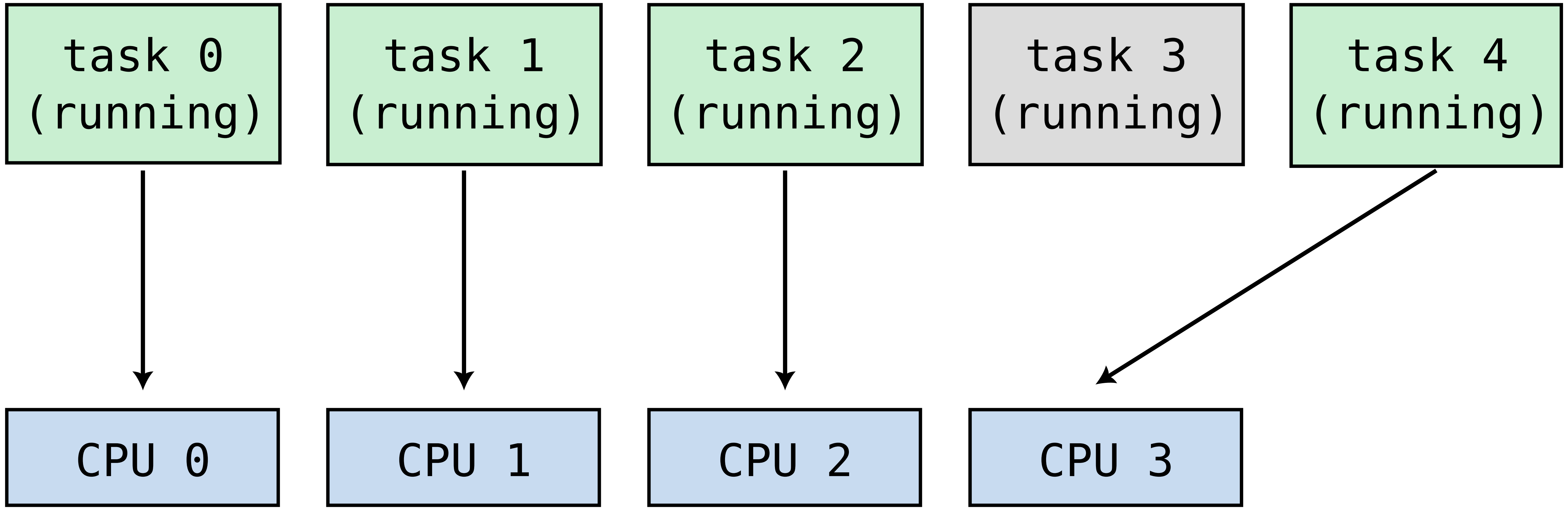
Multitasking on a multi-core processor



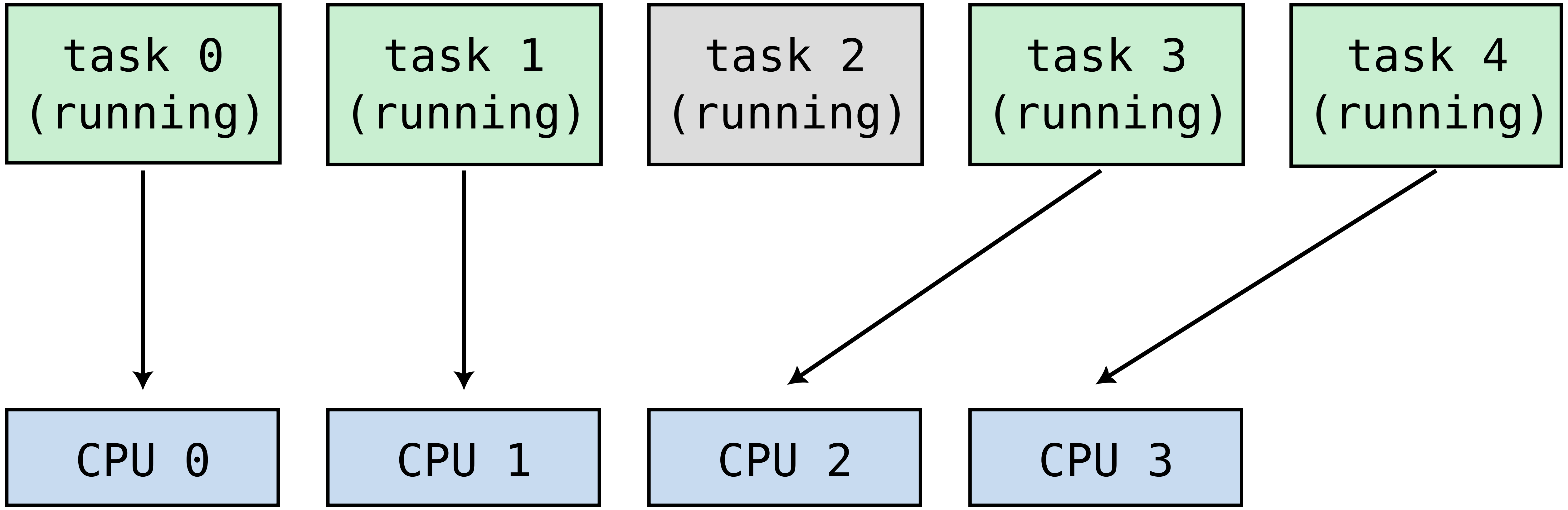
Multitasking on a multi-core processor



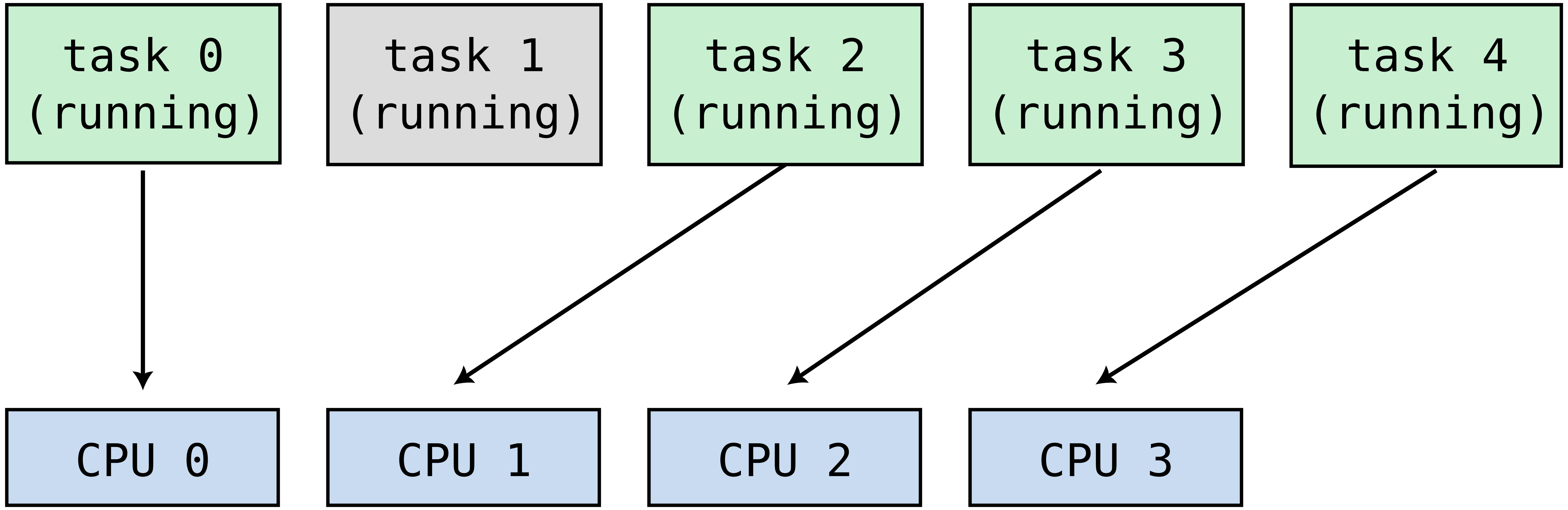
Multitasking on a multi-core processor



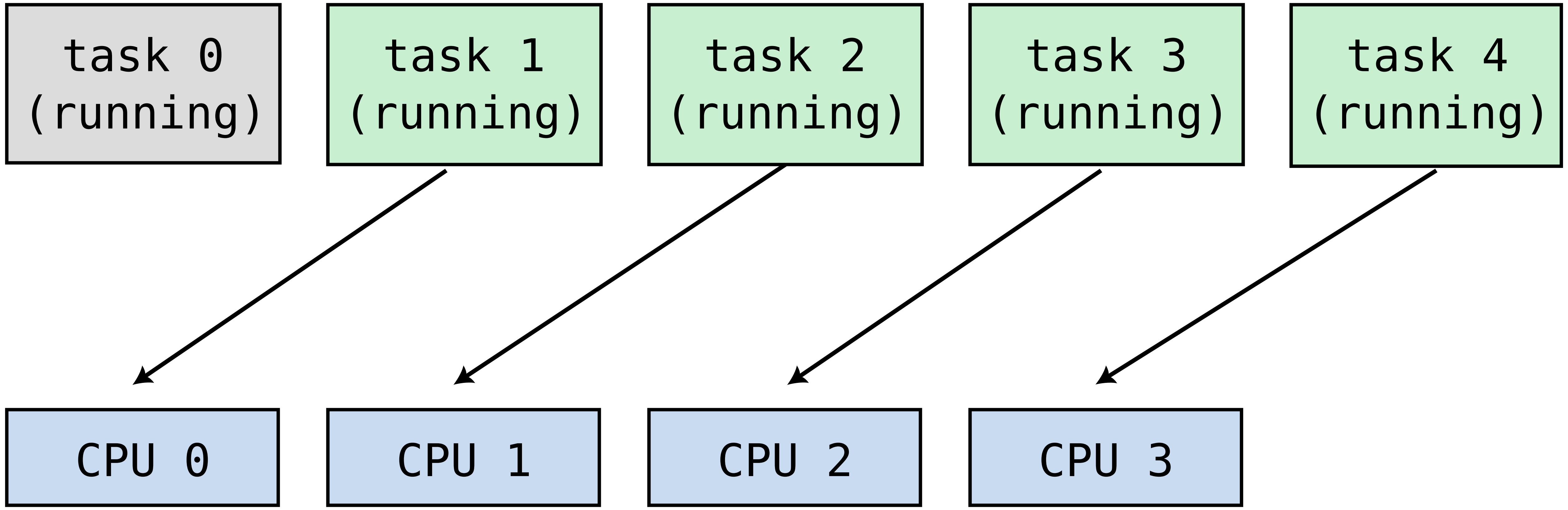
Multitasking on a multi-core processor



Multitasking on a multi-core processor



Multitasking on a multi-core processor



- From a hardware perspective:
  - A CPU corresponds to a single integrated circuit (“IC”) package
  - A computer can (rarely) have multiple CPUs

    Typically only found in datacenters, rarely more than 2
  - Each CPU can have multiple **cores**
    - generally 2-8 cores on laptops
    - up to 128 on datacenter CPUs

- From a software perspective:
  - Everything that can run a task is generally called a “CPU”
  - Only the kernel’s scheduler will (sometimes) care about CPU vs. core
  - All other software is unaware of the difference

- a CPU can have multiple copies of some logic blocks
- very common for arithmetic and logic units (ALUs)

	`add rdx, rdi`		
	`add rcx, rbx`		
	`mov rax, [rsi]`		
memory	______________		
ALU 1	______________	______________	ALU 2
	-		
	-		
	-		

Simultaneous Multithreading (SMT)

- From a hardware perspective:
  - With Simultaneous Multithreading (SMT) (a.k.a. Hyperthreading),
    - each core can run multiple (generally 2) tasks (“threads”)
    - but they share many logic blocks (in particular ALUs)
    - SMT works well when those logic blocks would otherwise be idle
    - SMT is ineffective when those logic blocks are the bottleneck

- From a software perspective:
  - Everything that can run a task is generally called a “CPU”
  - Only the kernel’s scheduler will (sometimes) care about CPU vs. core vs. **thread**
  - All other software is unaware of the difference
  - “Thread” has a different meaning in software

1. SIMD

SIMD

- SIMD stands for Single Instruction Multiple Data
- new, larger registers (in addition to the general purpose ones): “**vector** registers”

bits	255..224	223…192	191…160	159…128	127…96	95…64	63…32	31…0
256	ymm0							
64	fp64 #3		fp64 #2		fp64 #1		fp64 #0	
32	fp32 #7	fp32 #6	fp32 #5	fp32 #4	fp32 #3	fp32 #2	fp32 #1	fp32 #0
16								
8								

- **but**
  - SIMD registers cannot be treated as big integers
  - individual “**lanes**” (8-, 16-, 32- or 64-bit parts) generally cannot be accessed individually

SIMD registers

- On Intel (and AMD) ISAs:
  - SSE (~1999): 8 128-bit registers `xmm0` - `xmm7`
  - AVX (~2011): 16 256-bit registers `ymm0` - `ymm15`
  - AVX-512 (~2016, but not yet generally available): 32 512-bit registers `zmm0` - `zmm31`

- On ARM:
  - Neon (~2005): 16 128-bit registers `Q0` - `Q15`

Example

```c
void add_one(float v[4])
{
    v[0] += 1.0;
    v[1] += 1.0;
    v[2] += 1.0;
    v[3] += 1.0;
}
```

```
add_one:
        vbroadcastss    xmm0, DWORD PTR .LC1[rip]         # xmm0 <- { 1.0, 1.0, 1.0, 1.0 }
        vaddps          xmm0, xmm0, XMMWORD PTR [rdi]     # xmm0 <- xmm0 + [v]     (4x 32-bits)
        vmovups         XMMWORD PTR [rdi], xmm0           # [v] <- xmm0
        ret
```

Counter-example

```c
void many_ops(float v[4])
{
    v[0] += 1.0;
    v[1] -= 2.0;
    v[2] *= 3.0;
    v[3] /= v[2];
}
```

```asm
many_ops:
        vmovss  xmm1, DWORD PTR .LC0[rip]
        vmovss  xmm3, DWORD PTR [rdi+12]
        vmulss  xmm1, xmm1, DWORD PTR [rdi+8]      # <---
        MUL
        vmovss  xmm2, DWORD PTR [rdi+4]
        vmovss  xmm0, DWORD PTR .LC1[rip]
        vsubss  xmm2, xmm2, DWORD PTR .LC2[rip]    # <---
        SUB
        vaddss  xmm0, xmm0, DWORD PTR [rdi]        # <---
        ADD
        vdivss  xmm3, xmm3, xmm1                   # <---
        DIV
        vunpcklps       xmm0, xmm0, xmm2
        vunpcklps       xmm1, xmm1, xmm3
        vmovlhps        xmm0, xmm0, xmm1
        vmovups XMMWORD PTR [rdi], xmm0
        ret
```

This code cannot by performed by a single SIMD instruction

How to use SIMD

- Rely on compilers (“autovectorization”)
- Write assembly code
- Use compiler “intrinsics”
  - Intrinsics look like C functions

    but the compiler knows how to translate them to specific assembly code
  - > Intel intrinsics guide
  - > ARM intrinsics

```
for (int x = kx; x < nx; x++) {
    __m256d v = _mm256_andnot_pd(sign, gx[x]);
    __m256d oldmax = maxv[x];
    __m256d newmax = _mm256_max_pd(oldmax, v);
    __m256i keep = _mm256_castpd_si256(_mm256_cmp_pd(oldmax, newmax, _CMP_EQ_OQ));

    maxv[x] = newmax;
    maxi[x] = _mm256_or_si256(_mm256_and_si256(keep, maxi[x]), _mm256_andnot_si256(keep, ix));
}
```

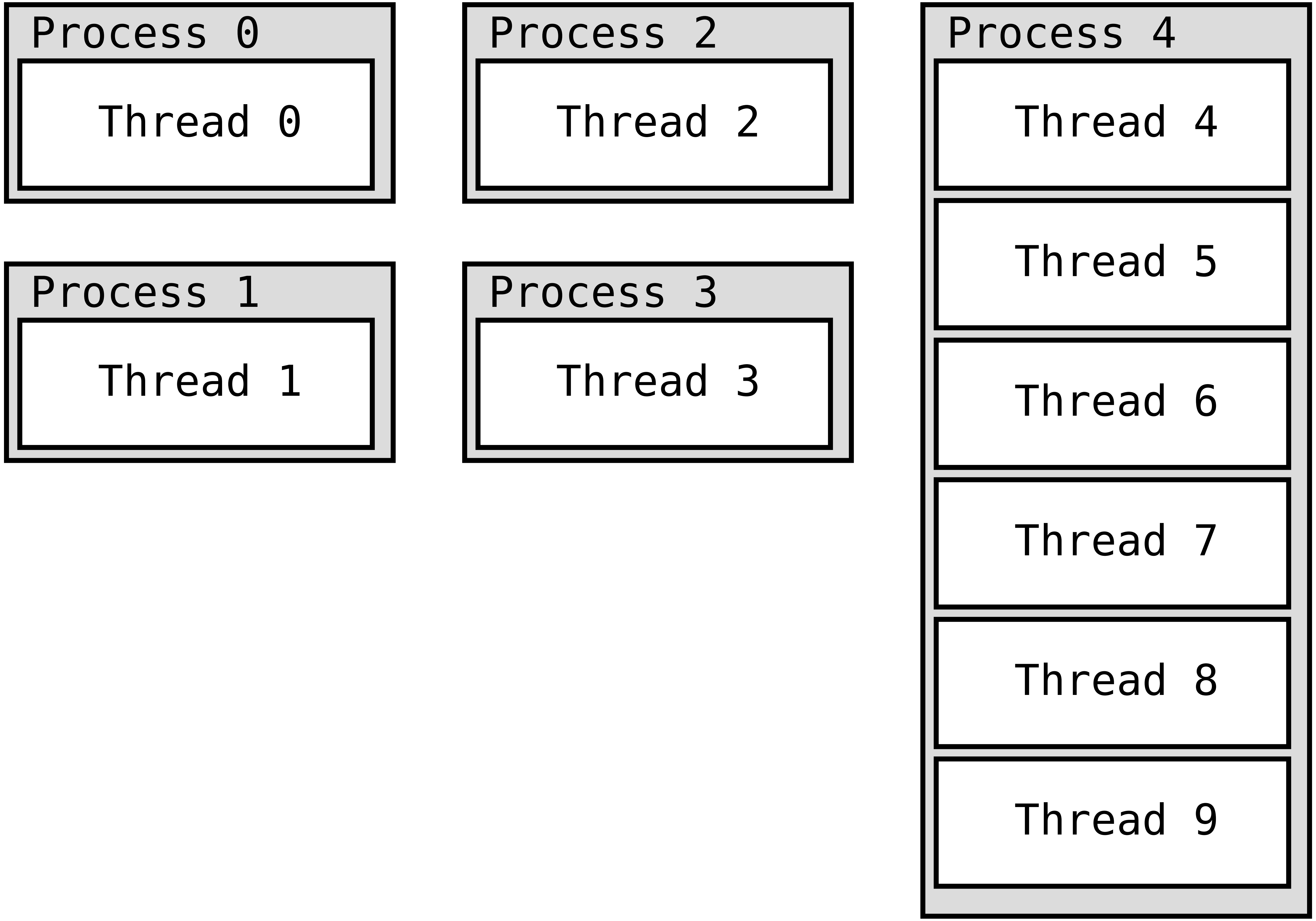
> refer to the intrinsics guide

2. THREAD-LEVEL CONCURRENCY

Processes and threads

- When the OS runs an executable, it gets its own **process**
- A single executable (if run multiple times) can have multiple independent processes
- Memory is virtualized: each process has its own view of the memory it owns

- A process can create (“spawn”) multiple **threads**
- Like processes, each thread is an individual task from the point of view of the scheduler
- Within a process, threads share a same view of the process memory


Process 0
Thread 0
Process 1
Thread 1
Process 2
Thread 2
Process 3
Thread 3
Process 4
Thread 4
Thread 5
Thread 6
Thread 7
Thread 8
Thread 9

- Pro: Communication between threads is extremely efficient
  - Just write something to memory,
  - let other threads read it through the same pointer

- Con: Because memory is shared, **synchronizing** threads is very complex

Wrong code (1)

```
int ready = 0;      //  one if there is some data in the buffer, zero otherwise
int buffer = 0;     //  data in the buffer

// Every push()ed element must be pop()ed exactly once.
// - push() will block until the buffer is empty/available/"not ready"
// - pop() will block until the buffer is nonempty/"ready"
void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}

int pop()
{
    while (ready == 0) {
        // wait
    }

    ready = 0;
    return buffer;
}
```

The C compiler is free to reorder this:

```c
void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}
```

into this:

```c
void push(int value)
{
    buffer = value;

    while (ready == 1) {
        // wait
    }

    ready = 1;
}
```

Wrong code (1)

The C compiler is free to infer that this loop:

```
while (ready == 1) {
    // wait
}
```

has either zero or infinitely many iterations without side effects (UB);

thus remove the loop!

Wrong code (2)

```
volatile int ready = 0;       //  one if there is some data in the buffer, zero otherwise
volatile int buffer = 0;      //  data in the buffer

void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}

int pop()
{
    while (ready == 0) {
        // wait
    }

    ready = 0;
    return buffer;
}
```

Thread 0

```
                            // ready = 1       buffer = 'A'
int pop()
{
    while (ready == 0) {
        // wait
    }
                            // ready = 1       buffer = 'A'
    ready = 0;              // ready = 0       buffer = 'A'







                            // ready = 0       buffer = 'B'
                            // ready = 1       buffer = 'B'


    return buffer;
}
```

Thread 1

```
void push(int value)              // push('B')
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}
```

Wrong code (3)

```c
volatile int ready = 0;       //  one if there is some data in the buffer, zero otherwise
volatile int buffer = 0;      //  data in the buffer

void push(int value)
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}

int pop()
{
    while (ready == 0) {
        // wait
    }

    int b = buffer;
    ready = 0;
    return b;
}
```

Thread 0

```
                            // ready = 0      buffer = 'X'
void push(int value)  // push('A')
{
    while (ready == 1) {
        // wait
    }
                            // ready = 0      buffer = 'X'




                            // ready = 0      buffer = 'B'
                            // ready = 1      buffer = 'B'


    buffer = value;         // ready = 1      buffer = 'A'
    ready = 1;              // ready = 1      buffer = 'A'
}
```

Thread 1

```
void push(int value)    // push('B')
{
    while (ready == 1) {
        // wait
    }

    buffer = value;
    ready = 1;
}
```

Solution

- low-level: compiler intrinsics for “**atomic**” operations:

  combined operations that are performed as a single unit

  no thread will every see the memory in an intermediate state

- high-level: use libraries that correctly implement some primitives:

  locks, queues, etc.

  - Posix threads (“`pthreads`”; Linux, MacOS)
  - OpenMP (Open Multi-Processing; portable)

3. DISTRIBUTED COMPUTING

Distributed computing

- In distributed computing, processes do not share memory
- They must communicate by explicitly sending data to each other

  (`send()`, `recv()`, etc.)

  typically over the network

Distributed computing

- Con: Communication is much slower than multithreading

- Pros:
  - Easier to implement and reason about
  - Scales to higher levels of parallelism
    - As of today, off-the-shelf computers can have up to

      2 processors × 128 cores × 2 SMT threads = 512 concurrent software threads
    - With distributed computing, networked computers can work together in parallel

- Libraries:
  - Message Passing Interface (MPI)
  - …

4. HARDWARE ACCELERATION

Graphics processing units (GPUs)

- GPUs were designed to perform the same simple, repetitive operations
  - on many pixels (“fragment shaders”), or
  - on many 3D coordinates (“vertex shaders”)

Examples (GLSL)

```
float box(in vec2 st, in vec2 size){
    size = vec2(0.5) - size*0.5;
    vec2 uv = smoothstep(size,
                        size+vec2(0.001),
                        st);
    uv *= smoothstep(size,
                    size+vec2(0.001),
                    vec2(1.0)-st);
    return uv.x*uv.y;
}
```

Examples (GLSL)

```
vec3 rgb2hsb( in vec3 c ){
    vec4 K = vec4(0.0, -1.0 / 3.0, 2.0 / 3.0, -1.0);
    vec4 p = mix(vec4(c.bg, K.wz),
                 vec4(c.gb, K.xy),
                 step(c.b, c.g));
    vec4 q = mix(vec4(p.xyw, c.r),
                 vec4(c.r, p.yzx),
                 step(p.x, c.r));
    float d = q.x - min(q.w, q.y);
    float e = 1.0e-10;
    return vec3(abs(q.z + (q.w - q.y) / (6.0 * d + e)),
                d / (q.x + e),
                q.x);
}
```

Examples (CUDA)

```
inline __device__ float3 roundAndExpand(float3 v, ushort *w) {
  v.x = rintf(__saturatef(v.x) * 31.0f);
  v.y = rintf(__saturatef(v.y) * 63.0f);
  v.z = rintf(__saturatef(v.z) * 31.0f);

  *w = ((ushort)v.x << 11) | ((ushort)v.y << 5) | (ushort)v.z;
  v.x *= 0.03227752766457f;  // approximate integer bit expansion.
  v.y *= 0.01583151765563f;
  v.z *= 0.03227752766457f;
  return v;
}
```

- GPUs were designed to perform the same simple, repetitive operations
  - on many pixels (“fragment shaders”), or
  - on many 3D coordinates (“vertex shaders”)
- they generally adopt a SIMT (“single instruction, multiple threads”) model
  - hundreds of threads working on different sets of data
  - but running the exact same instructions
- good fit for long loops performing repetitive operations
- bad fit for if/then/else

How do we use GPUs?

- GPUs are programmed in special-purpose languages

- Typically, all GPU code is compiled
  - during application startup,
  - by the device driver
  - for the specific GPU device installed (amount and subdivision of threads, memory, etc.)

- Two dominant players in the GPU market: nVidia and AMD

- Three major GPU programming languages:
  - CUDA (nVidia, proprietary)
  - ROCm (AMD, open-source)
  - OpenCL (cross-platform, open-source)

MATRIX MULTIPLICATION

N = 8192

8192 x 8192 matrix multiplication
precision: fp64 (“double”)
CPU: AMD Ryzen 7900 x3d

matmul_1	straightforward implementation	2932.059 s	1x	
matmul_2	transpose B matrix	357.569 s	8x	
matmul_3	block multiply	67.105 s	44x	
matmul_4	same code as matmul_3, SIMD	32.876 s	89x	
matmul_5	OpenBLAS	15.555 s	188x	1x
matmul_6	OpenBLAS, 24 threads	1.962 s	1494x	8x

N = 32768

```
32768 x 32768 matrix multiplication
precision: fp32 (“float”) - total 4 GB per matrix
CPU Ryzen 7900 x3d (released Feb 2023)
```

matmul_7	OpenBLAS, 1 thread	550.350 s	1x
matmul_8	OpenBLAS, 24 threads	50.577 s	11x
matmul_9	cuBLAS, nVidia A10G (Apr 2021)	13.152 s	42x
matmul_9	cuBLAS, nVidia H100 (Mar 2022)	? s	84x?