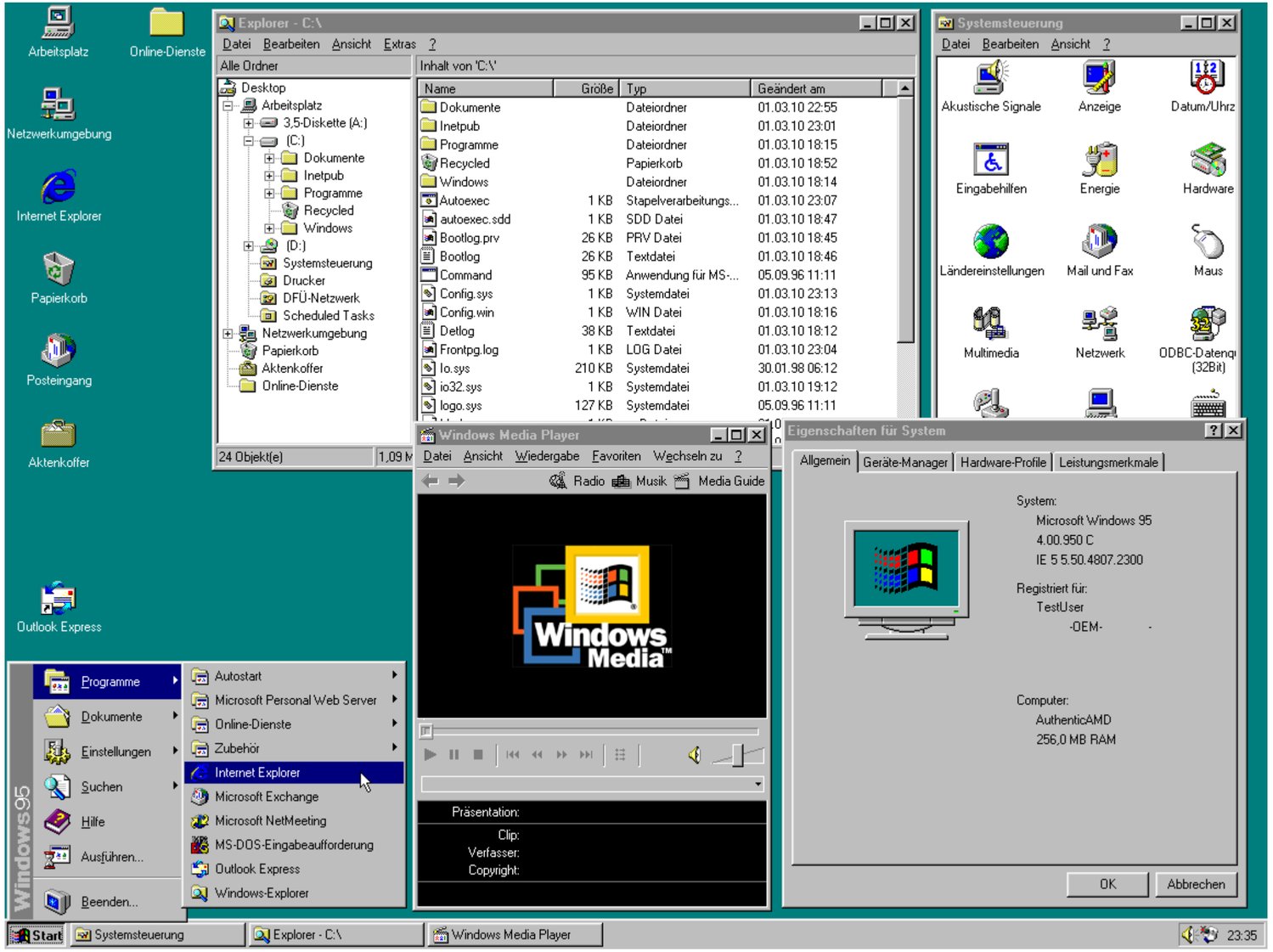
Parallel computation and multithreading

1. Multitasking

Types of parallel computations

0. CPU pipelines, SIMD
1. Multitasking ⟵ we are here
2. Thread-level concurrency
3. Distributed computing
4. Hardware acceleration

Multitasking

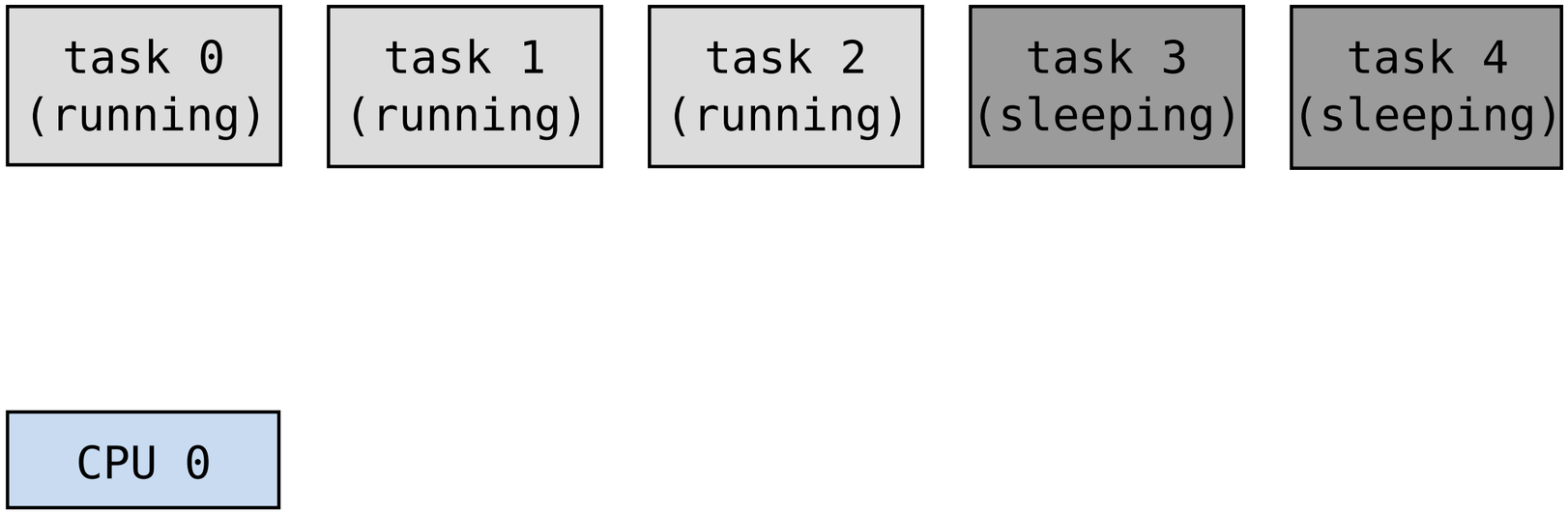


Multitasking

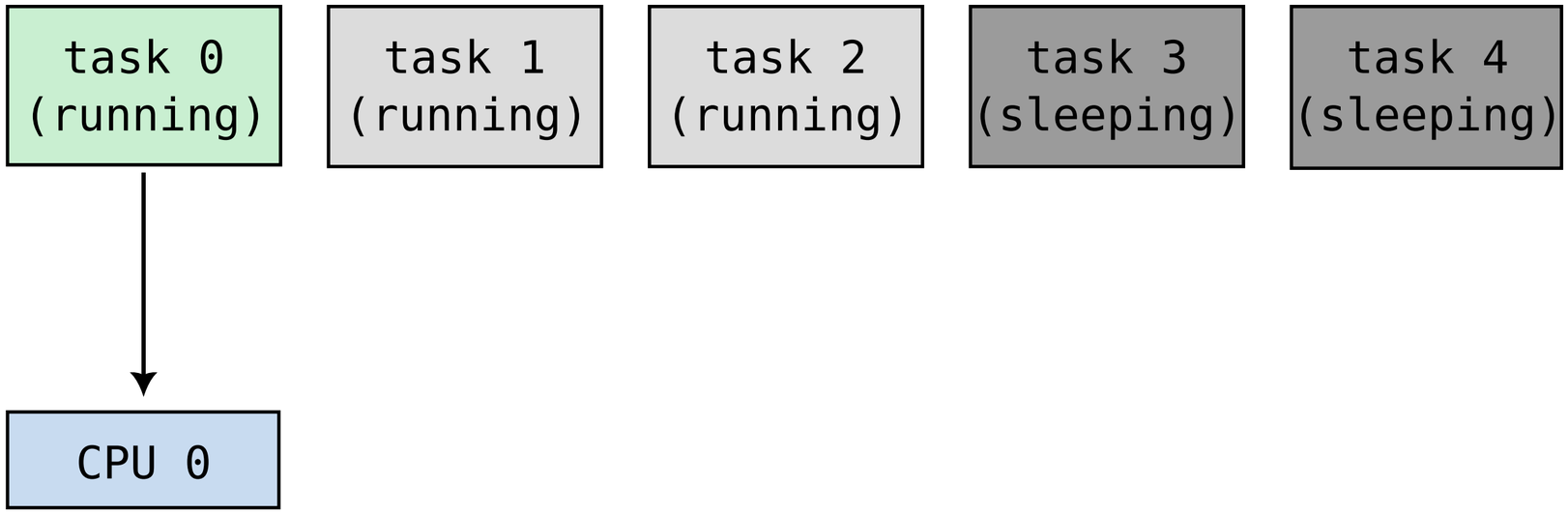
- Multitasking allows multiple executables to run “simultaneously”
  (even on a single processor)

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

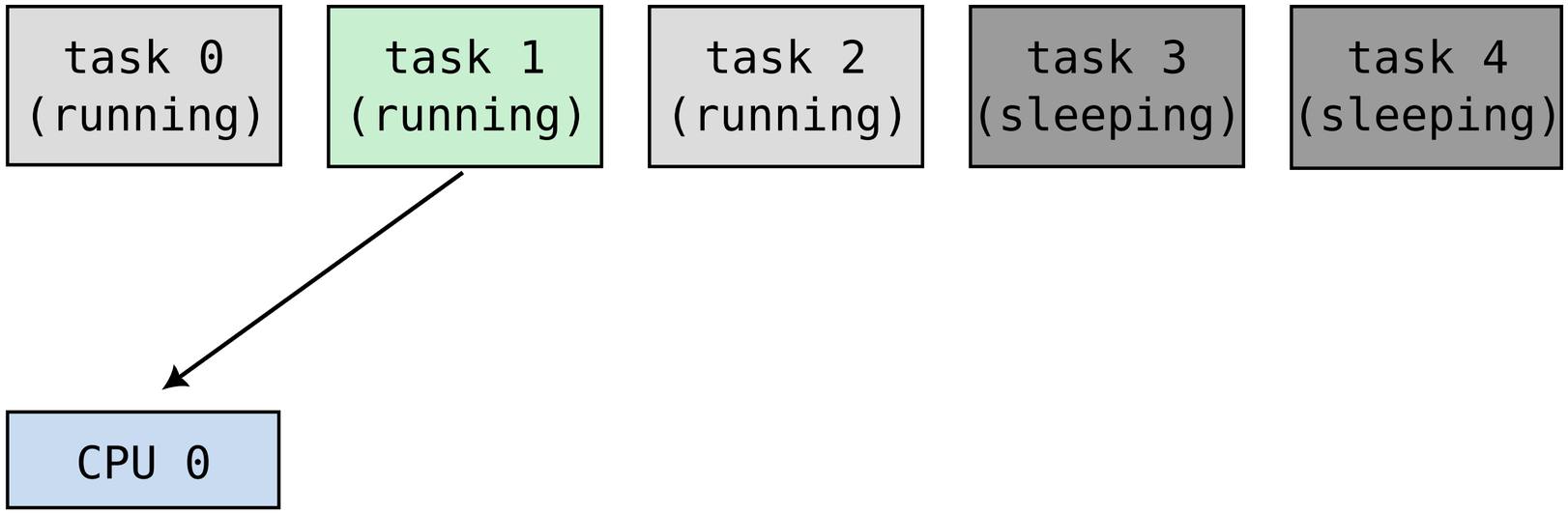
Multitasking on a single CPU



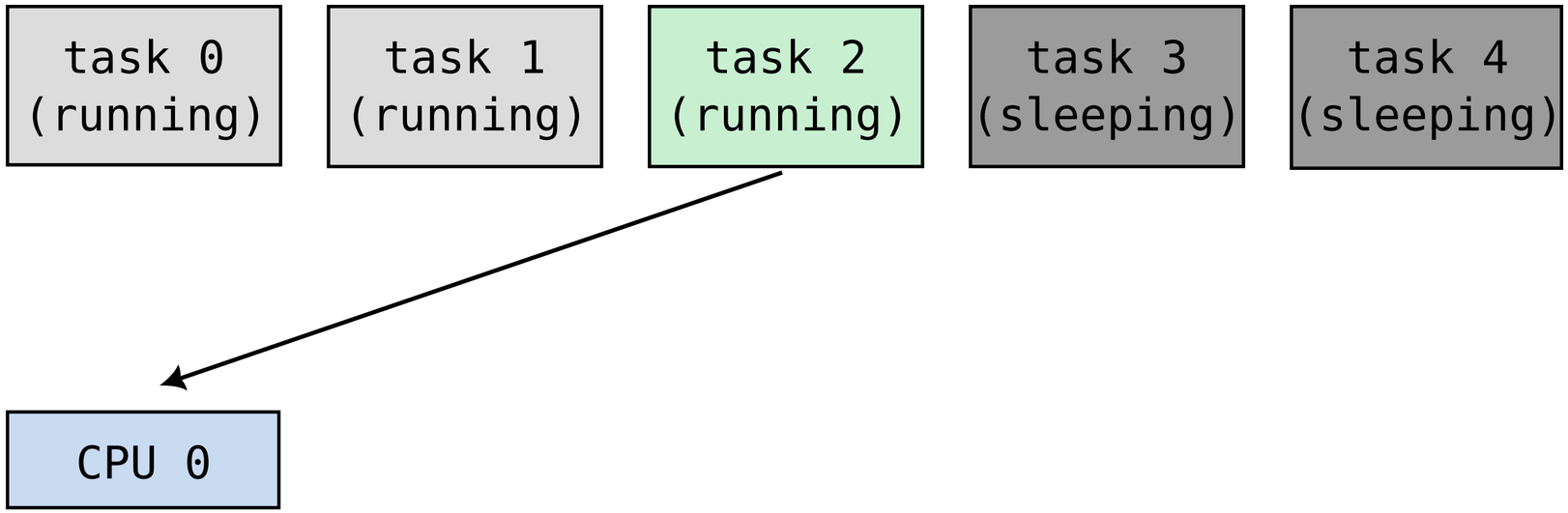
Multitasking on a single CPU



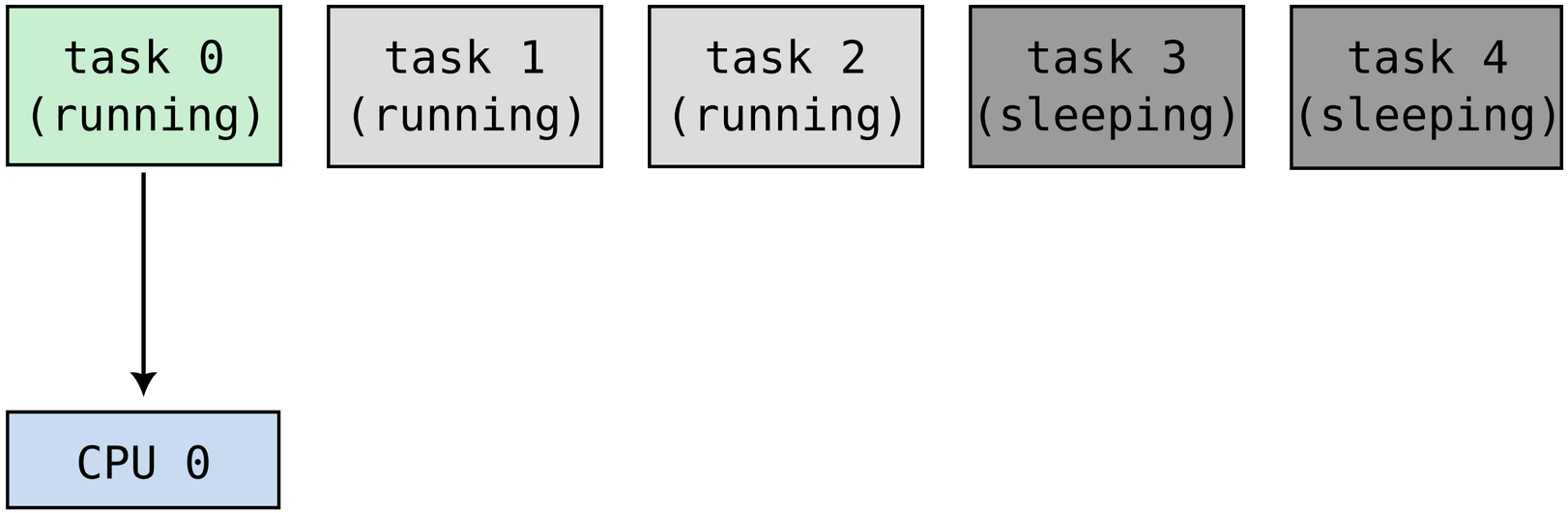
Multitasking on a single CPU



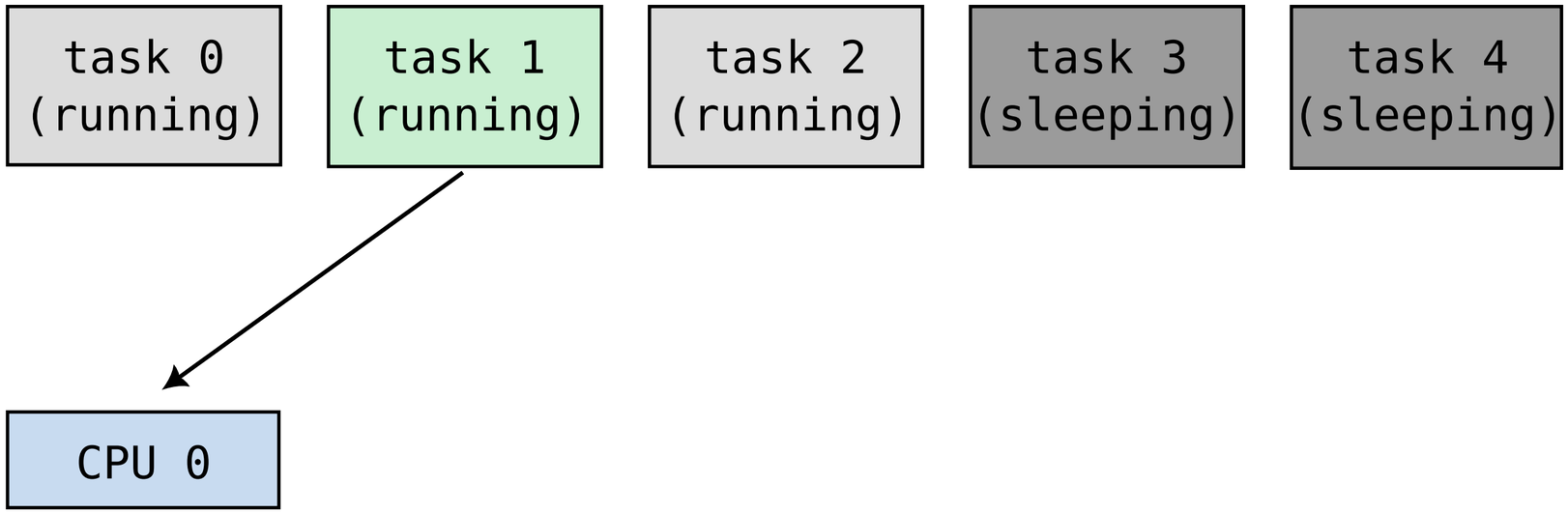
Multitasking on a single CPU



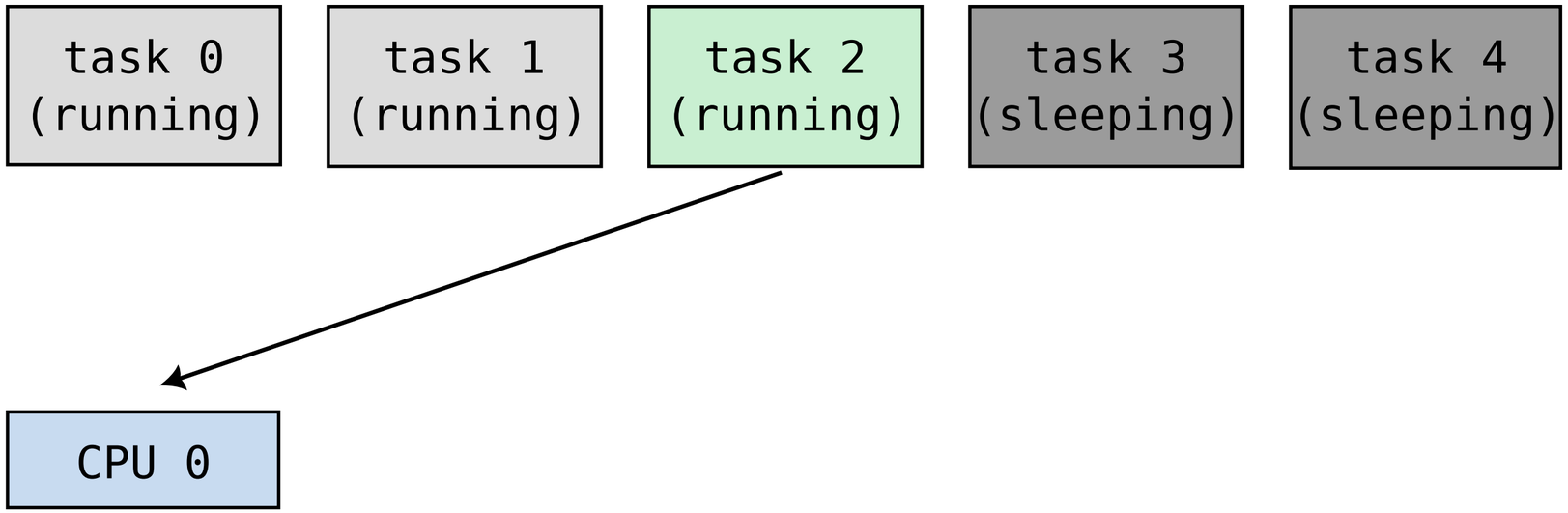
Multitasking on a single CPU



Multitasking on a single CPU



Multitasking on a single CPU



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
  - …

Preemption

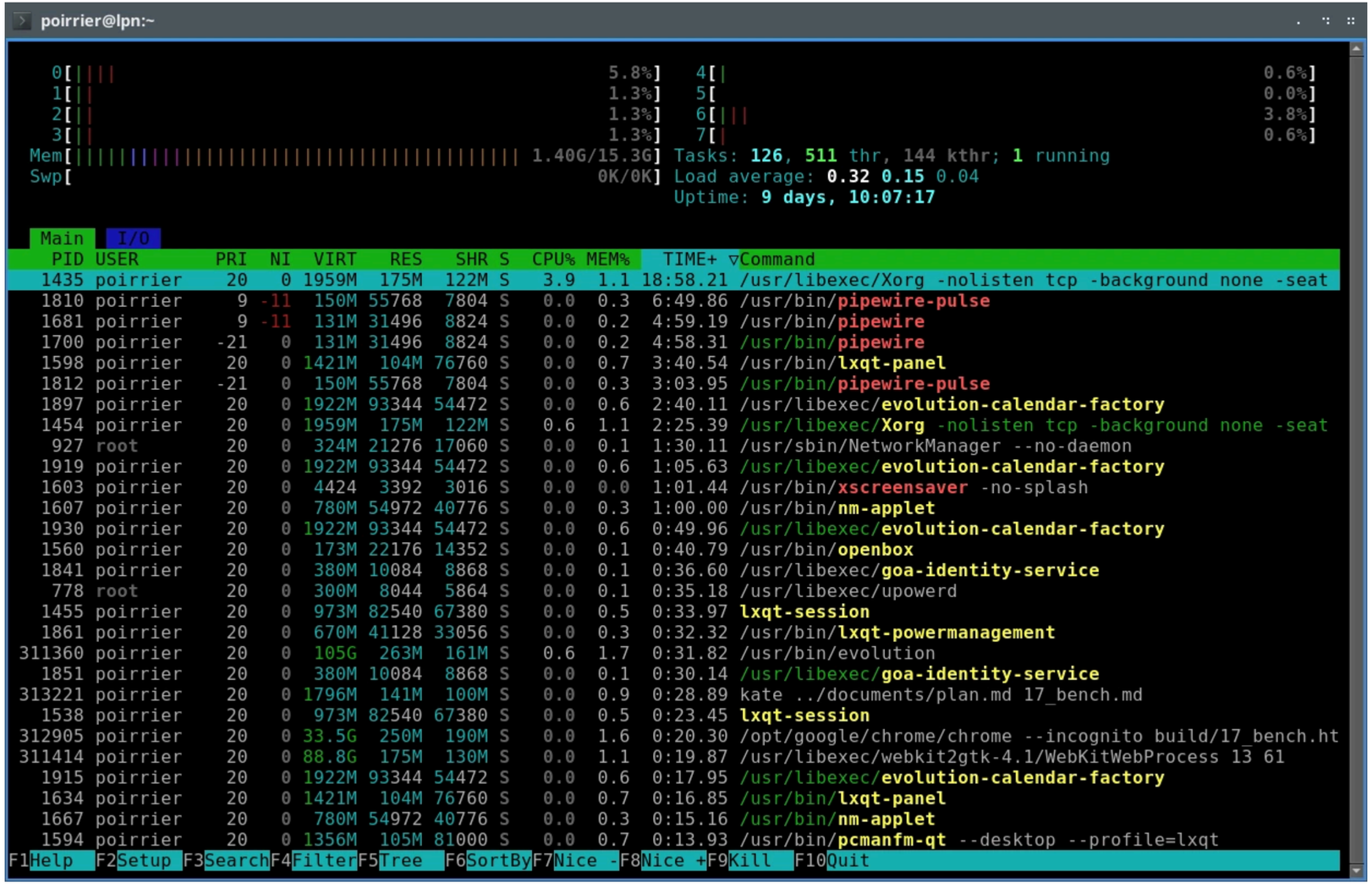
- When the scheduler decides to interrupt a **running** process (e.g. to run another)
  - the process is said to "**preempted**"
  - it becomes "**runnable**"

- 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()`

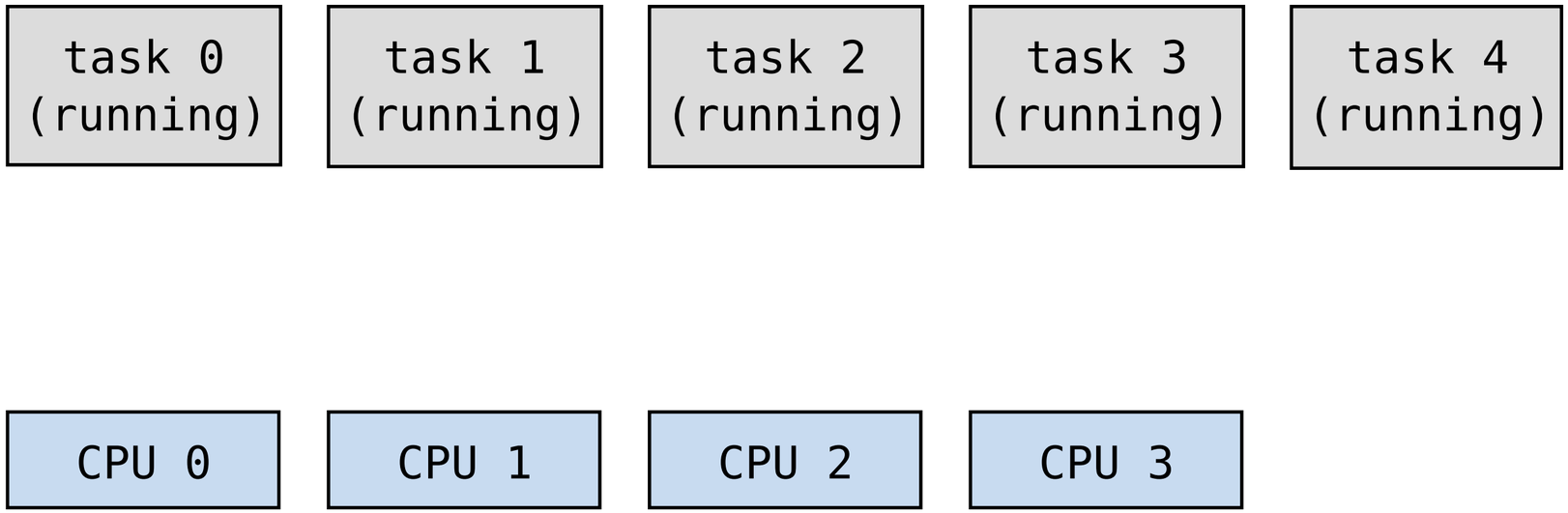
Preemptive multitasking

- 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)

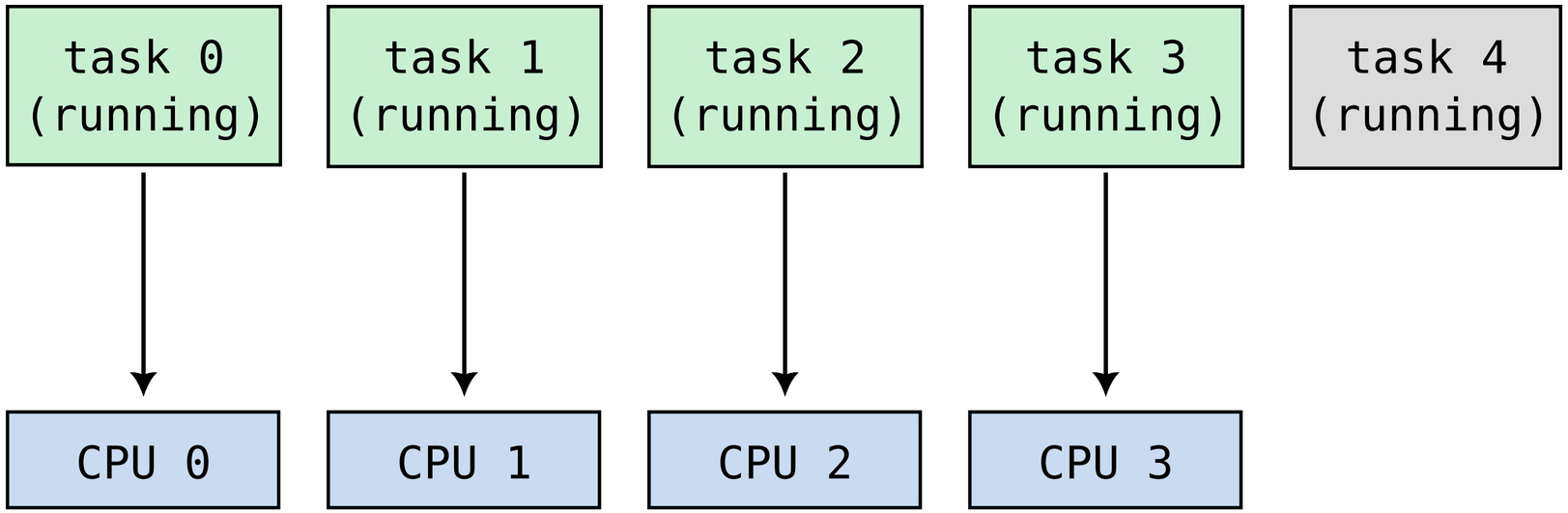
- The only tasks that are normally **running**/**runnable** are those performing CPU-intensive operations
  - graphics rendering
  - audio/video/data compression and decompression
  - computations
  - etc.


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 F3Search F4Filter F5Tree F6SortBy F7Nice - F8Nice + F9Kill F10Quit

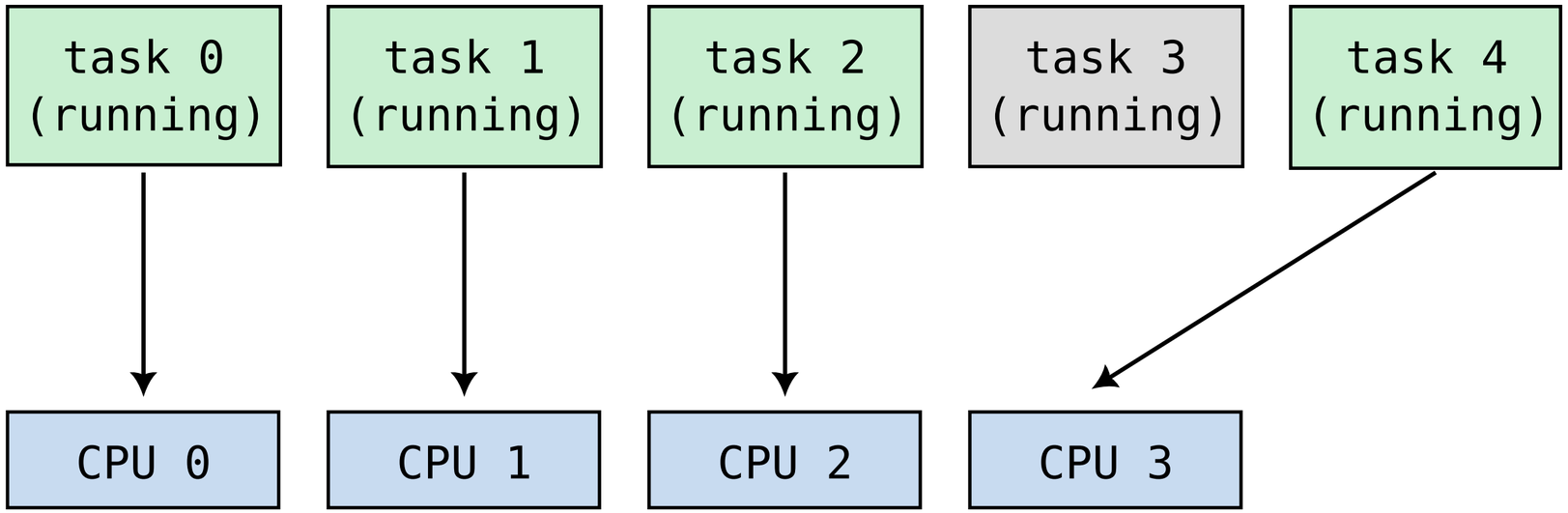
Multitasking on a multi-core processor



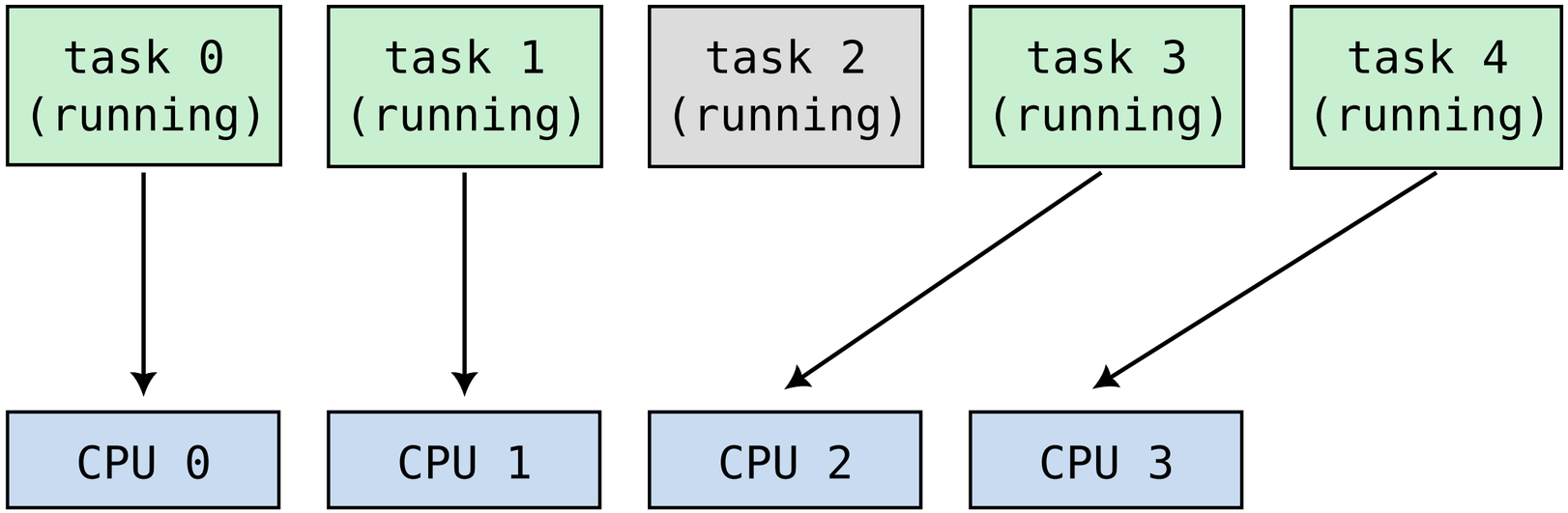
Multitasking on a multi-core processor



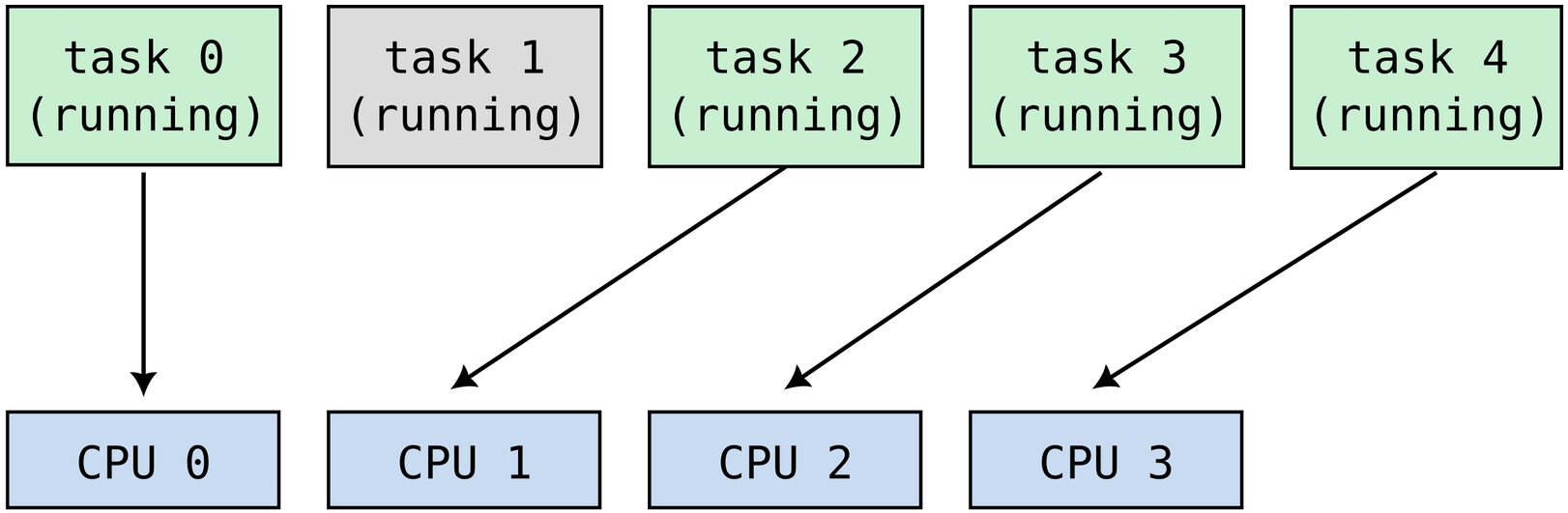
Multitasking on a multi-core processor



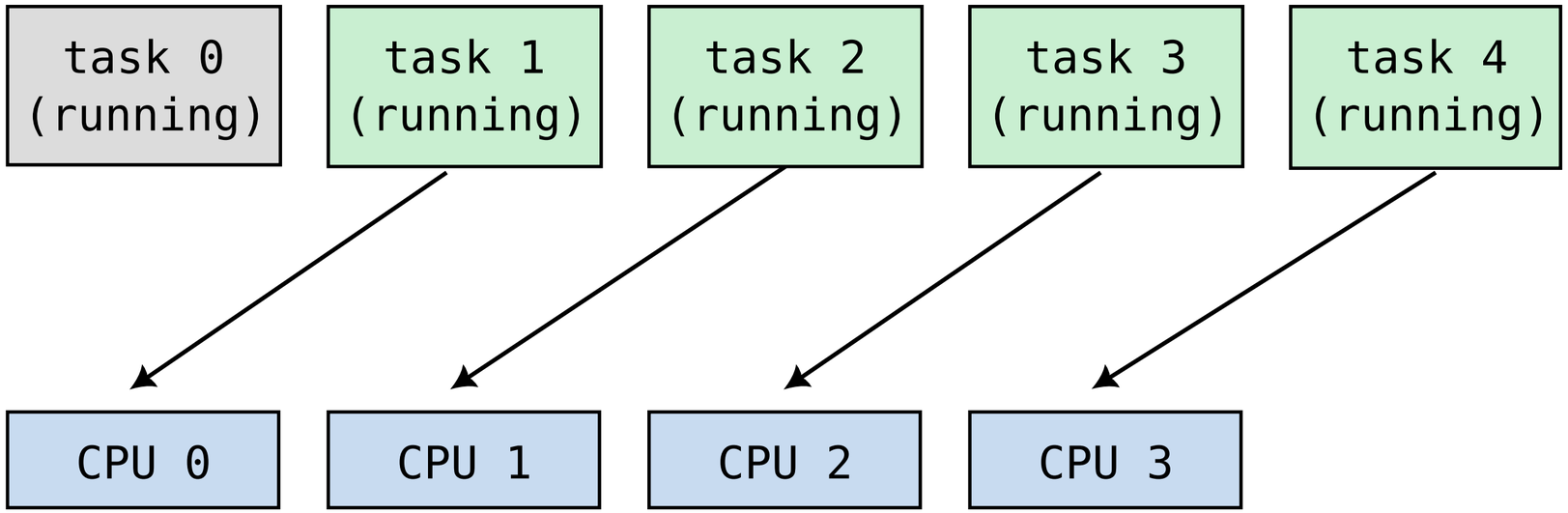
Multitasking on a multi-core processor



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 288 on datacenter CPUs
  - With Simultaneous Multithreading (SMT) a.k.a. Hyperthreading,

    a core can run two tasks simultaneously.

- 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

2. Thread-level concurrency

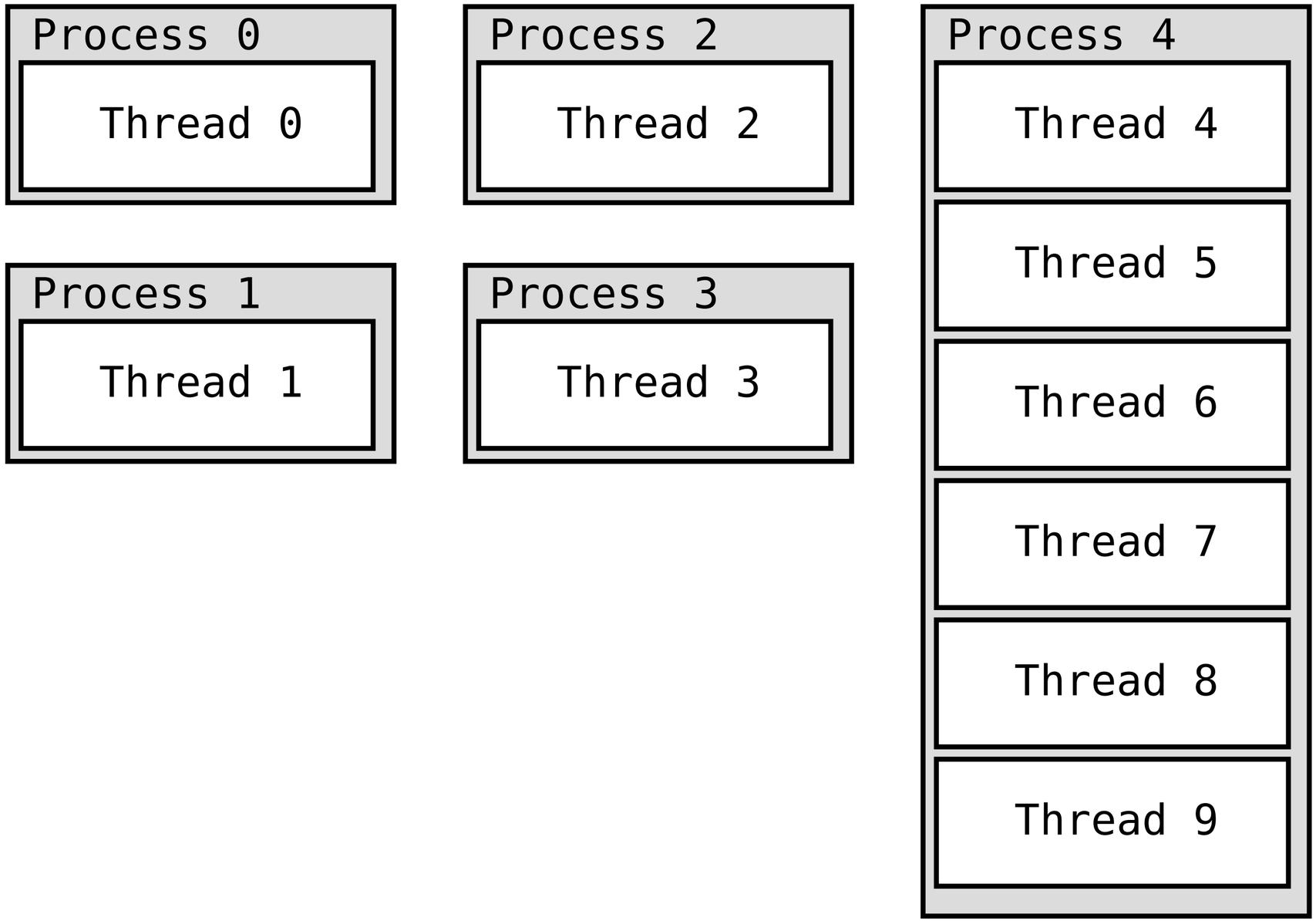
Types of parallel computations

0. CPU pipelines, SIMD
1. Multitasking
2. Thread-level concurrency ⟵ we are here
3. Distributed computing
4. Hardware acceleration

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

- 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
  - It is difficult for humans to reason about
  - The C language was not designed for multithreading

    no semantics for concurrent read/write to shared variables until C11

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:

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

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

into this:

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

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

    ready = 1;
}
```

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

The C compiler can also notice that this loop has either

- zero iterations, or
- infinitely many iterations without side effects (UB!)

thus remove the loop!

The `volatile` keyword

An object that has volatile-qualified type [C23, p119]

- may be modified in ways unknown to the implementation
- or have other unknown side effects.

Example:

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

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)

```
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 ever see the memory in an intermediate state

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

  locks, queues, etc.

  - C11 threads
  - Win32 threads (Windows)
  - Posix threads (“`pthreads`”; Linux, MacOS)
  - OpenMP (Open Multi-Processing; higher-level, portable)

C11 threads

- Introduced in C11
- Was initially defective and thus amended in C17
- Support is optional for compilers writers
- Very little documentation besides the C standard

Other multithreading APIs

- Win32 threads: official documentation.
- Posix threads: official documentation, current reference.
- OpenMP: tutorials, specification.

Posix threads

Using with `libpthread`

```
#include <pthread.h>

...
```

Link with `-lpthread`:

```
clang -o executable obj0.o obj1.o obj2.o -lpthread
```

Creating threads

```c
#include <pthread.h>

typedef void *thread_fn_t(void *arg);

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, thread_fn_t *fn, void *arg);
```

- `thread` is a pointer to a `pthread_t` (in practice: an integer of some size).

  On success, it is set to a thread identifier (for the just-created thread).
- `attr` allows overriding default parameters (stack size and address, thread priority, …).

  Set to `NULL` for defaults.
- `fn` is a pointer to a function that will run the thread code.

  (When the function returns, the thread terminates.)
- `arg` is a pointer passed as an argument to `fn`.

  (Allows differentiating threads if they run the same `fn`).

```c
#include <pthread.h>

typedef void *thread_fn_t(void *arg);

int pthread_create(pthread_t *thread, const pthread_attr_t *attr, thread_fn_t *fn, void *arg);
```

Example:

```c
void *thread_fn(void *arg)
{
    int data = *((int *)arg);
    printf("Thread %d\n", data);
    return NULL;
}

int main()
{
    int data0 = 0;
    int data1 = 1;
    pthread_t id0, id1;

    if (pthread_create(&id0, NULL, thread_fn, &data0))
        return 1;

    if (pthread_create(&id1, NULL, thread_fn, &data1))
        return 1;

    // ...
}
```

Terminating threads

```
void pthread_exit(void *retval);
```

- `pthread_exit()` is an alternative to returning from the thread function
- It exits as if the thread function returned `retval`

Waiting for a thread to terminate

```
int pthread_join(pthread_t thread, void **retval);
```

We pass:

- the thread identifier `thread`, and
- a pointer to a void pointer `retval`.

  (the void pointer will be set to the return value of the thread function)

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <pthread.h>

int x = 0;

void *mythread(void *arg)
{
    x = 1;
    printf("Child process set x=1\n");
    return NULL;
}

int main(int argc, char *argv[])
{
    pthread_t tid;
    int e0 = pthread_create(&tid, NULL, mythread, NULL);

    if (e0) {
        fprintf(stderr, "pthread_create(): %s\n", strerror(e0));
        exit(EXIT_FAILURE);
    }

    void *vp;
    int e1 = pthread_join(tid, &vp);

    if (e1) {
        fprintf(stderr, "pthread_join(): %s\n", strerror(e1));
        exit(EXIT_FAILURE);
    }

    printf("Returned pointer: %p\n", vp);
    printf("Parent process sees x=%d\n", x);
    return EXIT_SUCCESS;
}
```

```
Child process set x=1
Returned pointer: (nil)
Parent process sees x=1
```

Synchronizing primitives

- mutex
- condition variable

Mutexes

Mutexes

“Mutex” stands for mutual exclusion.

Mutexes have two states:

- locked and “owned” by a specific thread
- unlocked

Attempting to lock an already-locked mutex blocks until it is unlocked first

```c
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

- `pthread_mutex_lock()` locks the given mutex
  1. if `mutex` was unlocked,

     it becomes locked and owned by the calling thread

     returns immediately
  2. if `mutex` is already locked by another thread

     suspends the calling thread until `mutex` is unlocked.

     when `mutex` is unlocked, see a.

- `pthread_mutex_unlock()` unlocks the given mutex

  (currently locked and owned by the calling thread)

  Both functions return zero in normal operation.

Example

```
char shared_buffer[16777216];
pthread_mutex_t shared_buffer_mutex = PTHREAD_MUTEX_INITIALIZER;


void *thread_function(void *arg)
{
    while (1) {
        if (pthread_mutex_lock(&shared_buffer_mutex))
            abort();

        process_data(shared_buffer);

        if (pthread_mutex_unlock(&shared_buffer_mutex))
            abort();
    }

    return NULL;
}
```

Consider the following task:

- We have a shared buffer:

```
pthread_mutex_t buff_mutex = PTHREAD_MUTEX_INITIALIZER;
char *buff_data;
size_t buff_size;
size_t buff_used;
```

- We have a thread with a single mission:
  When the buffer is full, it calls a function `buffer_flush()`

```c
pthread_mutex_t buff_mutex = PTHREAD_MUTEX_INITIALIZER;
char *buff_data;
size_t buff_size;
size_t buff_used;
```

```c
void *flusher_thread(void *arg)
{
    while (1) {
        // wait until buffer is full
        while (1) {
            if (pthread_mutex_lock(&buff_mutex))
                abort();

            if (buff_used == buff_size) {
                // stop waiting, while owning the mutex
                break;
            }

            if (pthread_mutex_unlock(&buff_mutex))
                abort();
        }

        // flush buffer (while owning the mutex)
        buffer_flush();

        // unlock mutex
        if (pthread_mutex_unlock(&buff_mutex))
            abort();
    }

    return NULL;
}
```

- This implementation has high *lock contention*:

  `flusher_thread()` will keep trying own `buff_mutex` all the time

- If it succeeds in doing so without waiting, it is essentially busy waiting.

Solution: condition variables

Condition variables

Condition variables

A **condition variable** allows efficiently blocking until an arbitrary condition becomes true.

Other threads cooperatively signal it when the condition **may** have become true.

```c
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait()`:
  - unlocks `mutex`
  - waits for `cond` to be signaled
  - re-locks `mutex`

- `pthread_cond_signal()` restarts one thread waiting `cond`
  - if no thread is waiting, nothing happens
  - if several threads are waiting, exactly one is restarted (it is not specified which)

```c
pthread_mutex_t buff_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t buff_cond = PTHREAD_COND_INITIALIZER;
char *buff_data;
size_t buff_size;
size_t buff_used;
```

```c
void *flusher_thread(void *arg)
{
    while (1) {
        // wait until buffer is full
        if (pthread_mutex_lock(&buff_mutex))
            abort();

        while (buff_used < buff_size) {
            if (pthread_cond_wait(&buff_cond, &buff_mutex))
                abort();
        }

        // flush buffer (while owning the mutex)
        buffer_flush();

        // unlock mutex
        if (pthread_mutex_unlock(&buff_mutex))
            abort();
    }

    return NULL;
}
```

```c
void *filler_thread(void *arg)
{
    while (1) {
        // lock mutex
        if (pthread_mutex_lock(&buff_mutex))
            abort();

        // fill buffer
        size_t old_used = buff_used;

        buffer_fill();

        if (buff_used > old_used) {
            if (pthread_cond_signal(&buff_cond))
                abort();
        }

        // unlock mutex
        if (pthread_mutex_unlock(&buff_mutex))
            abort();
    }

    return NULL;
}
```

Resources

Resources

- “Perf Book”:
  Is Parallel Programming Hard, And, If So, What Can You Do About It?
  https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html

- Book about (among other things) concurrency:
  Operating Systems: Three Easy Pieces
  https://pages.cs.wisc.edu/~remzi/OSTEP/

- pthreads tutorial:
  https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html