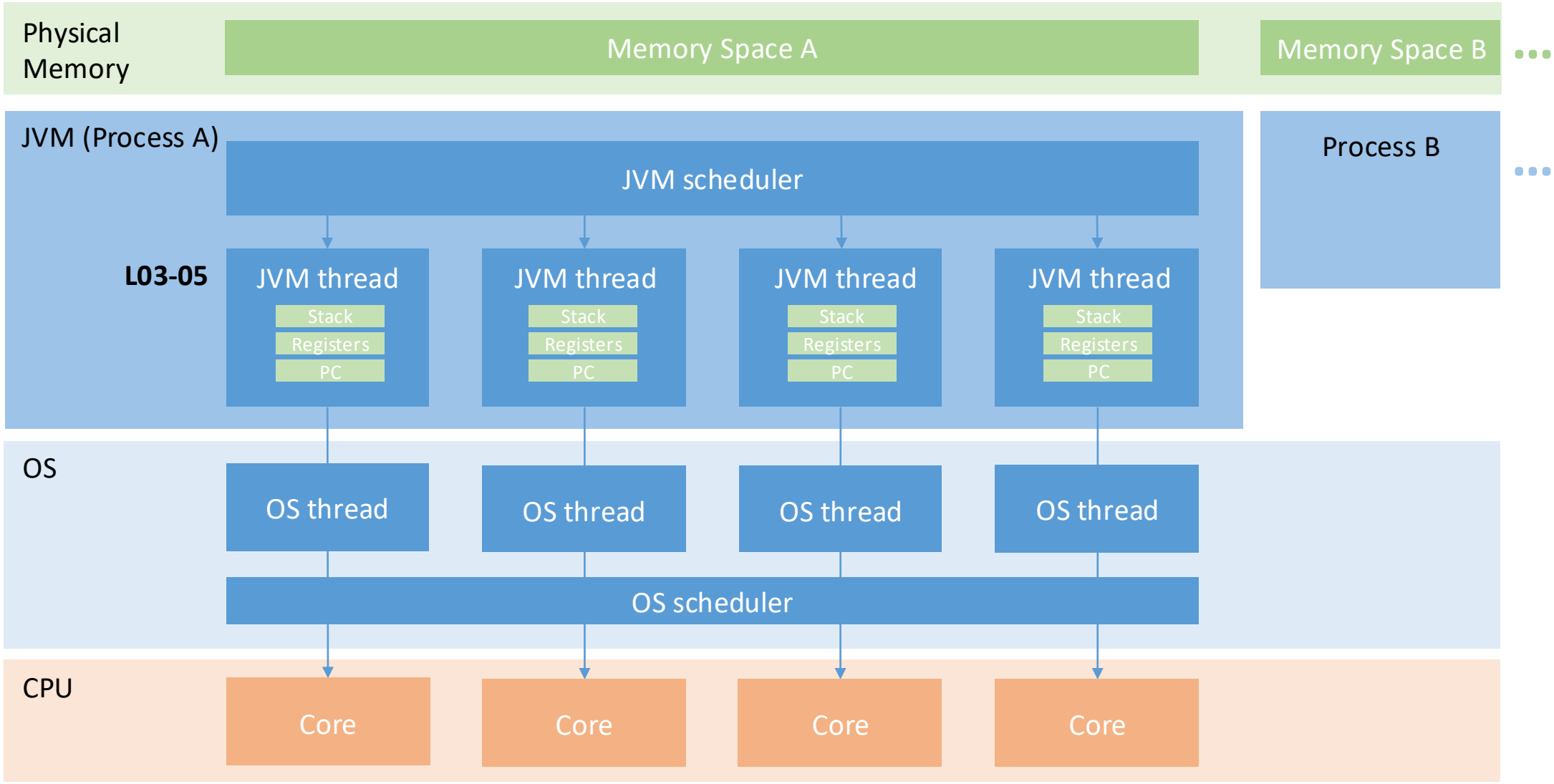
Parallel Programming

Introduction to Threads and Synchronization

Big picture (Part I)



Structure of next lectures

Processes and threads

Sequential vs parallelism vs concurrency

Java threads: creation, status, join

Shared resources, thread interleavings

Synchronization with `synchronize` blocks

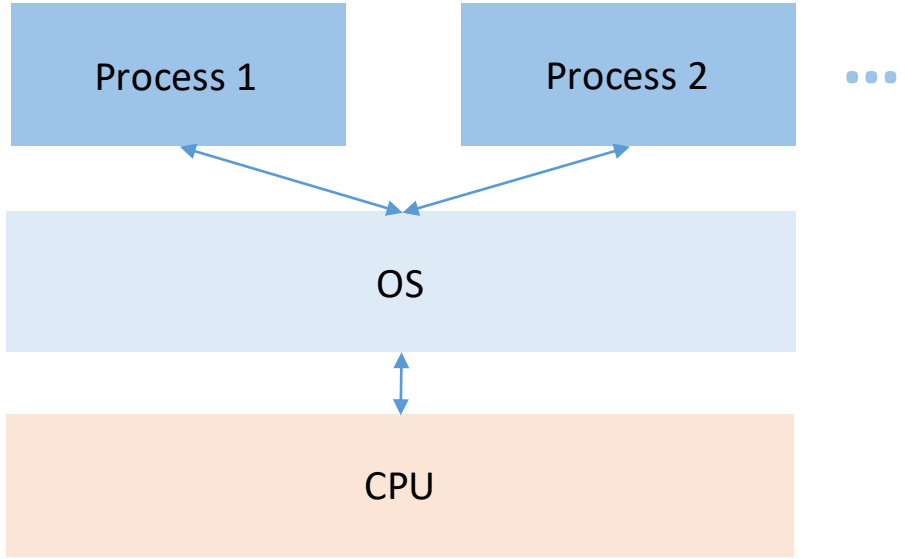
Coordination/communication: Producer-consumer with `wait` & `notify`

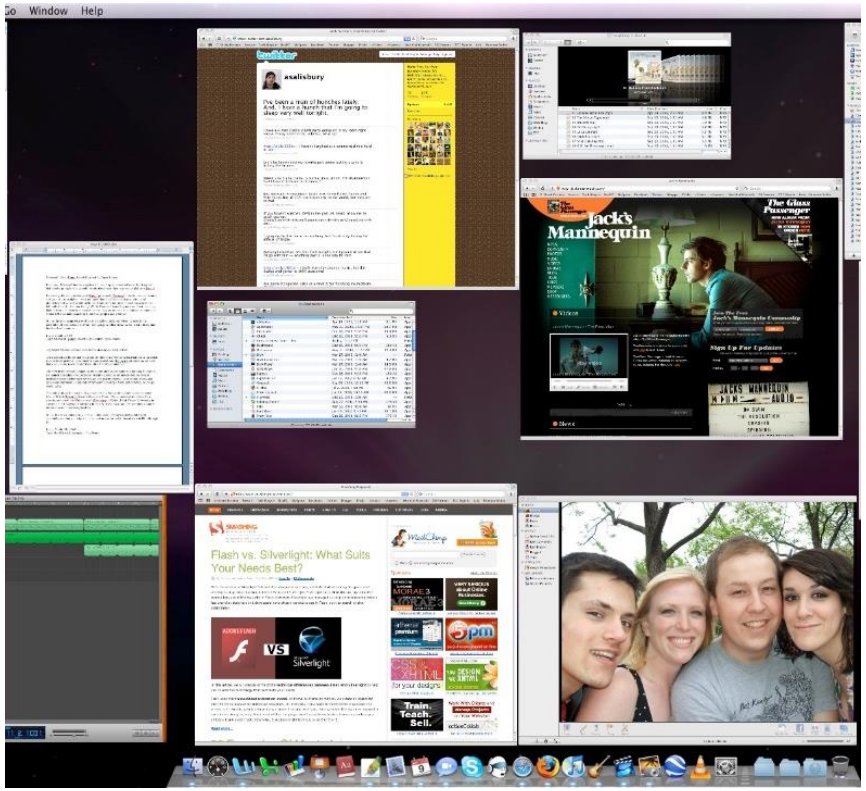
L03

L04

L05

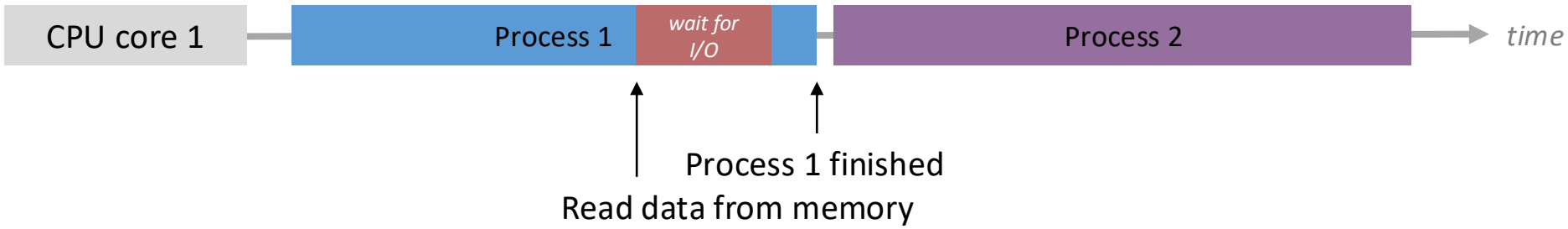
Processes


Process 1
Process 2
OS
CPU


Window
Help
asalisbury
Jack's Mannequin
Flash vs. Silverlight: What Suits Your Needs Best?
VS
Silverlight

Processes on a single-core CPU

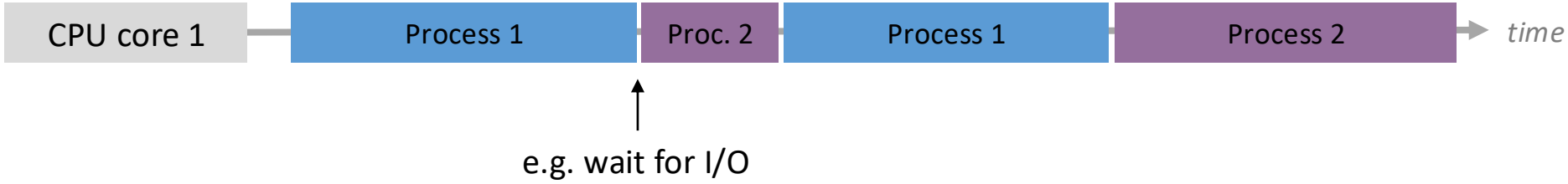
- More processes than CPU cores
- A single-core CPU can only execute one instruction at a time, meaning only one process runs any given time
- Let's assume a sequential execution: *only one process is executing in the system*
- If a process is waiting (I/O), CPU is idle until it can resume execution



Multitasking

Multitasking and process management

- OS scheduler handles overall task management (multitasking)
- Different types of OS schedulers, one is the CPU scheduler (short-term scheduler)
- Determines which process gets CPU time next
  - Processes arrive
  - Scheduler picks a process to run on the CPU
  - Selected process runs for a while
  - OS switches to another process (time slicing, event-driven interruptions)
- Efficient CPU utilization, fair CPU sharing



Multitasking and process management

- Allows for asynchronous I/O
  - I/O devices and CPU are truly parallel
  - 10ms waiting for HDD allows other processes to execute >10^{10} instructions
- Multitasking creates impression of parallelism even on a single core
- Concurrency: *multiple processes are active at the same time and make progress, though not necessarily simultaneously.*

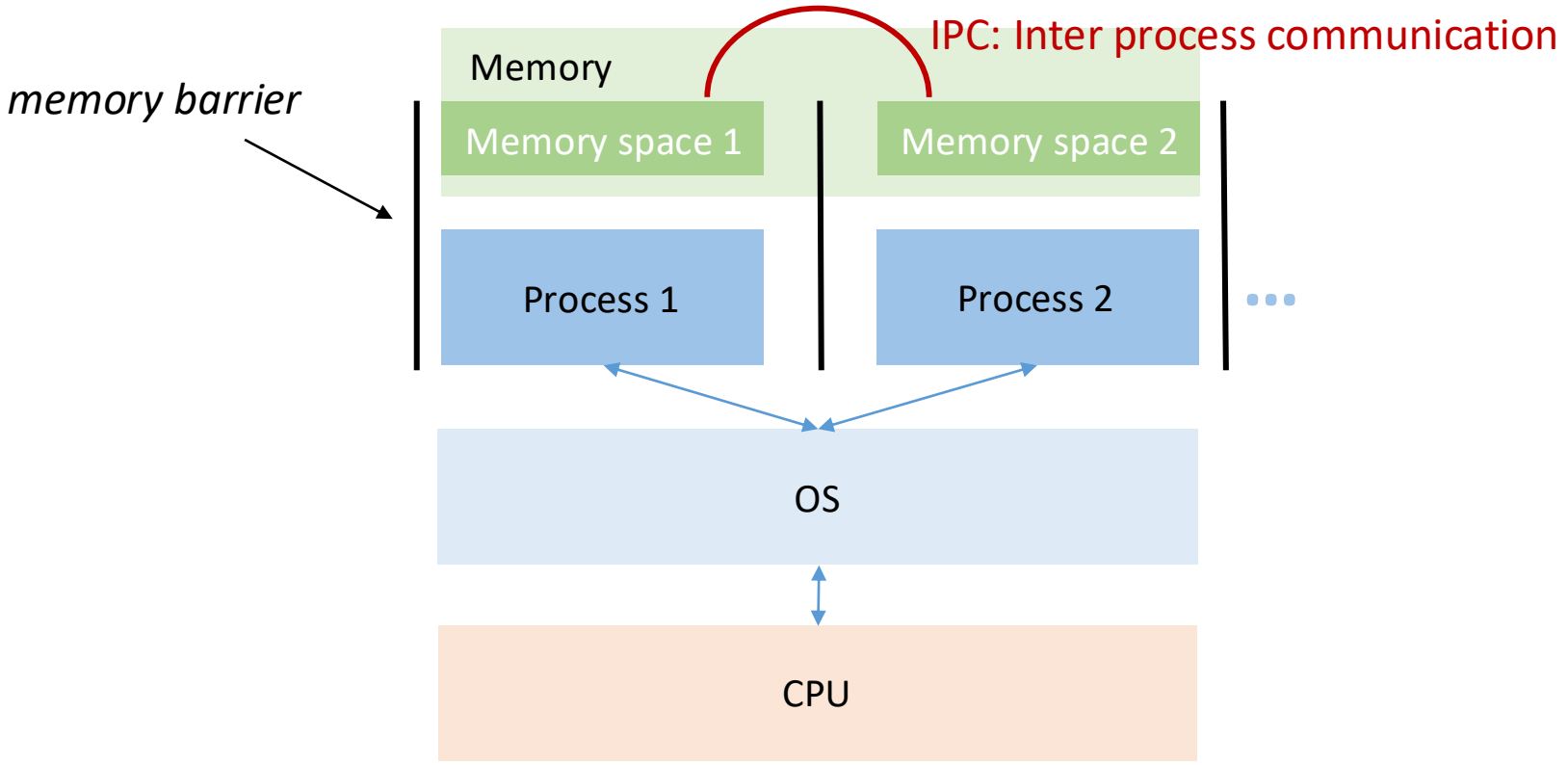
Process context

A process is (essentially) a program executing inside an OS

Each running instances of a program (e.g., multiple browser windows) is a separate process

Processes share CPU, but each process has own memory space

Separate memory per process

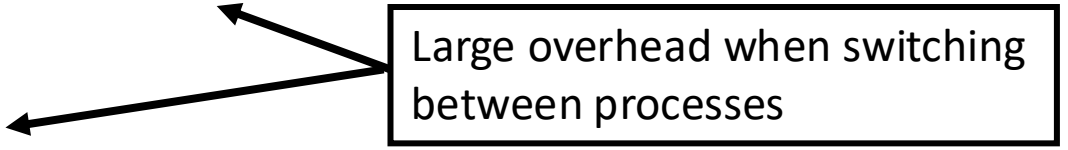


Process context

A process is (essentially) a program executing inside an OS

Each running instances of a program (e.g., multiple browser windows) is a separate process

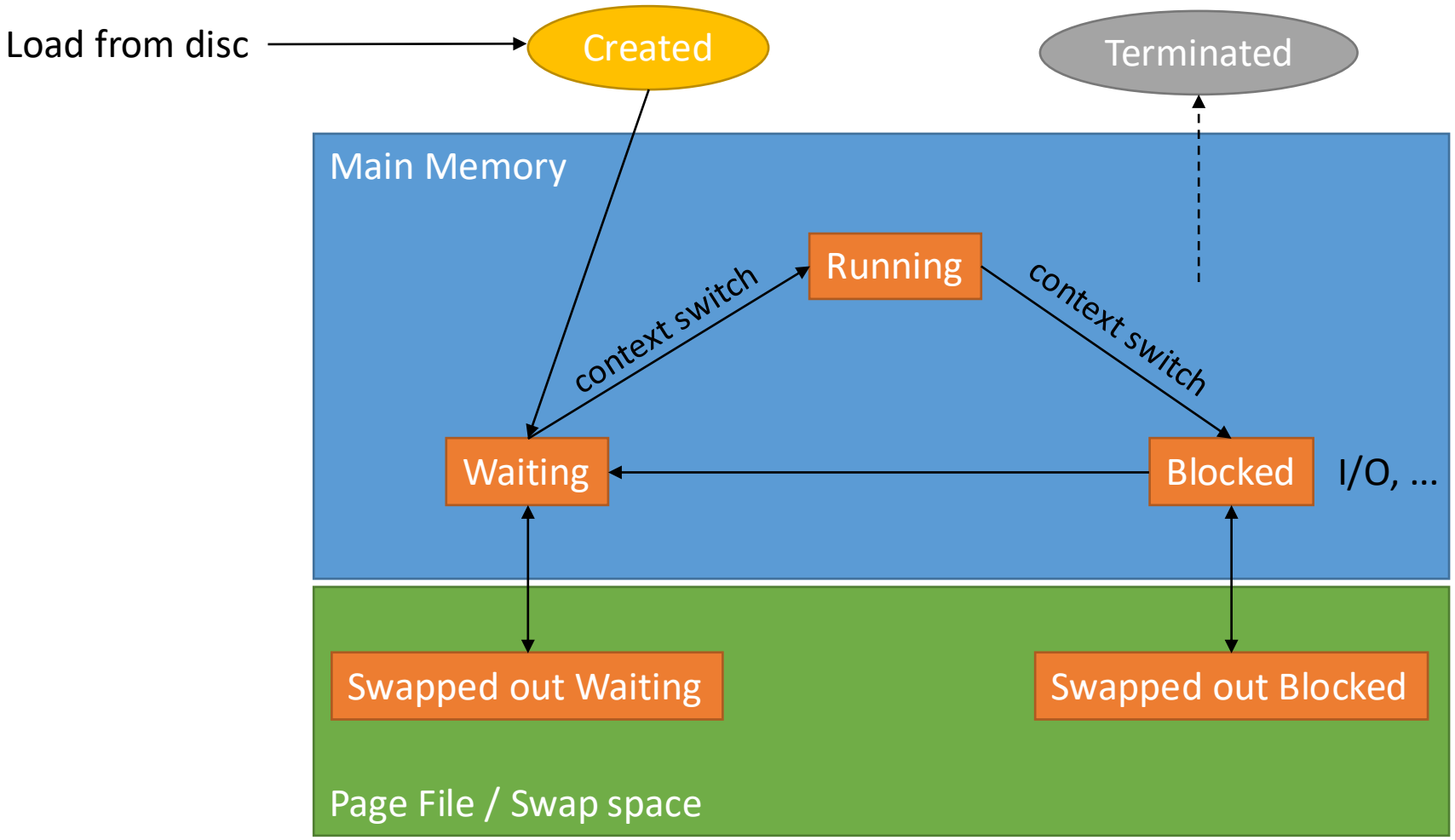
Processes share CPU, but each process has own memory space



Each process has a context:

- Hardware context: Instruction counter, CPU registers, execution state, proc flags
- Memory context: Virtual memory, stack, heap, ...
- OS-level context: Process ID, open files, ...

Process lifecycle states



Processes' contexts stored in main memory, allowing the OS to manage / switch efficiently

Process management

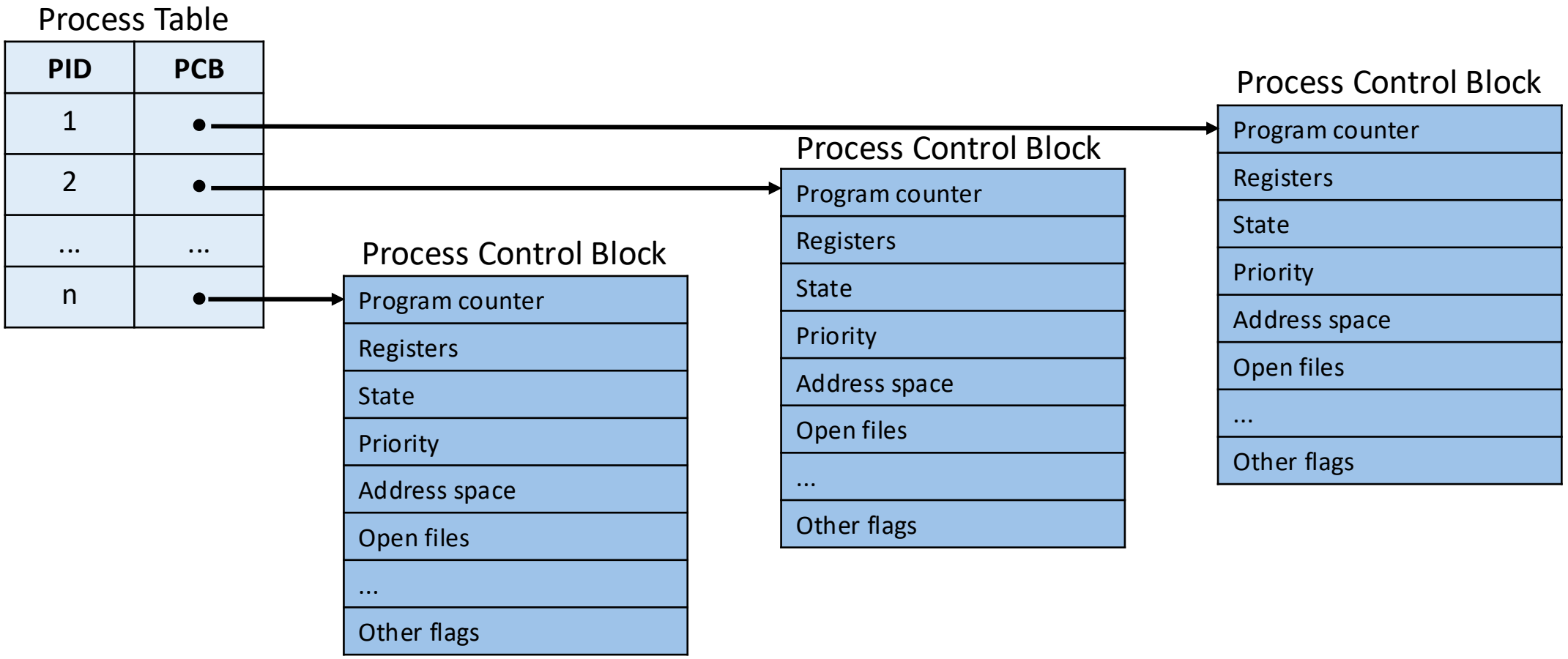
Processes need resources

- CPU time, memory, etc.

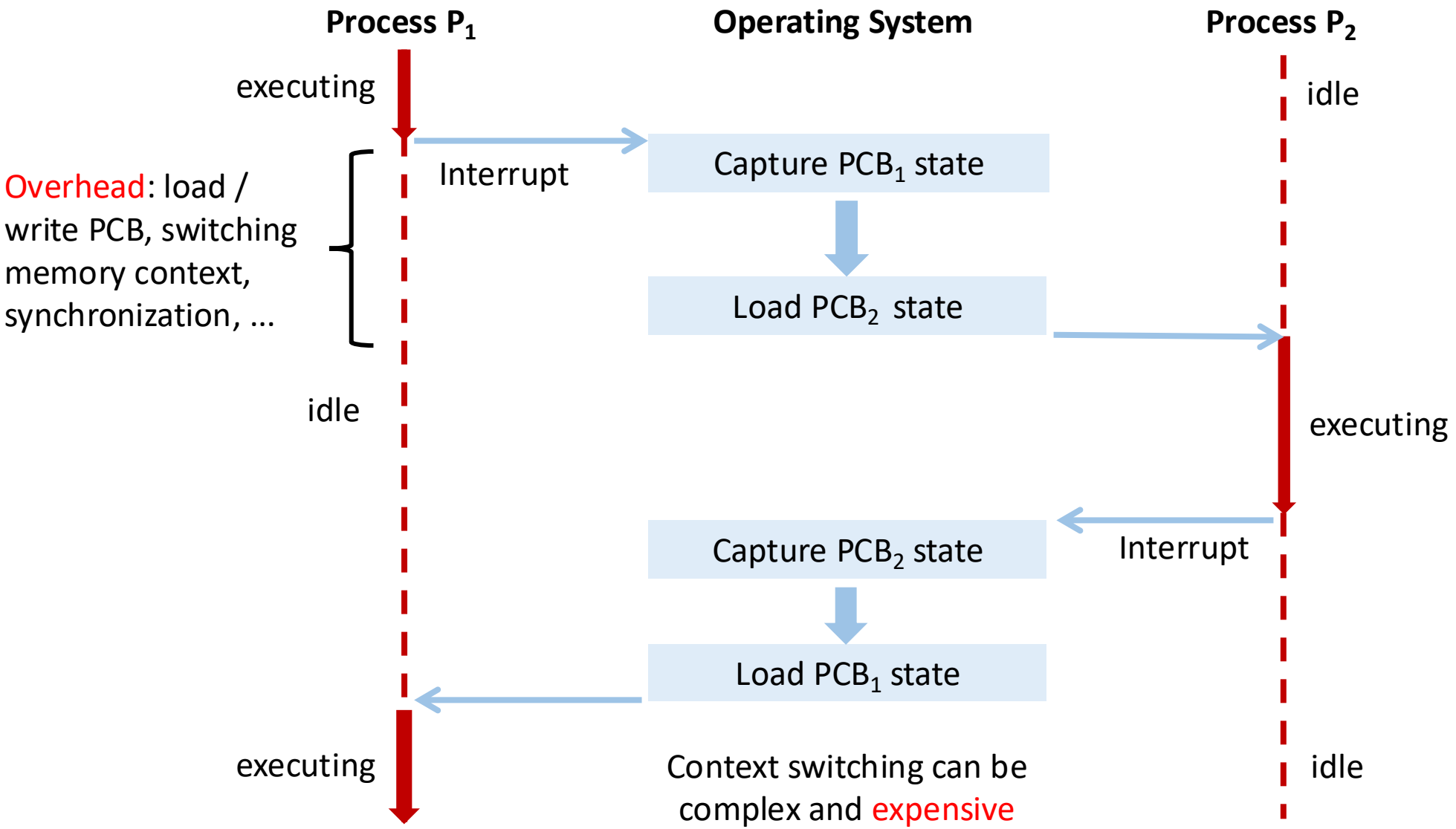
OS manages processes:

- Starts processes
- Terminates processes (frees resources)
- Controls resource usage (prevents monopolizing CPU time)
- Schedules CPU time
- Synchronizes processes if necessary
- Allows for inter process communication

Process control blocks (PCB)

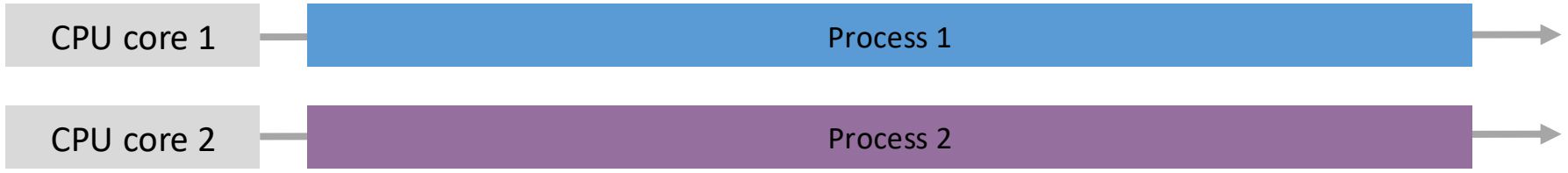


Context switch

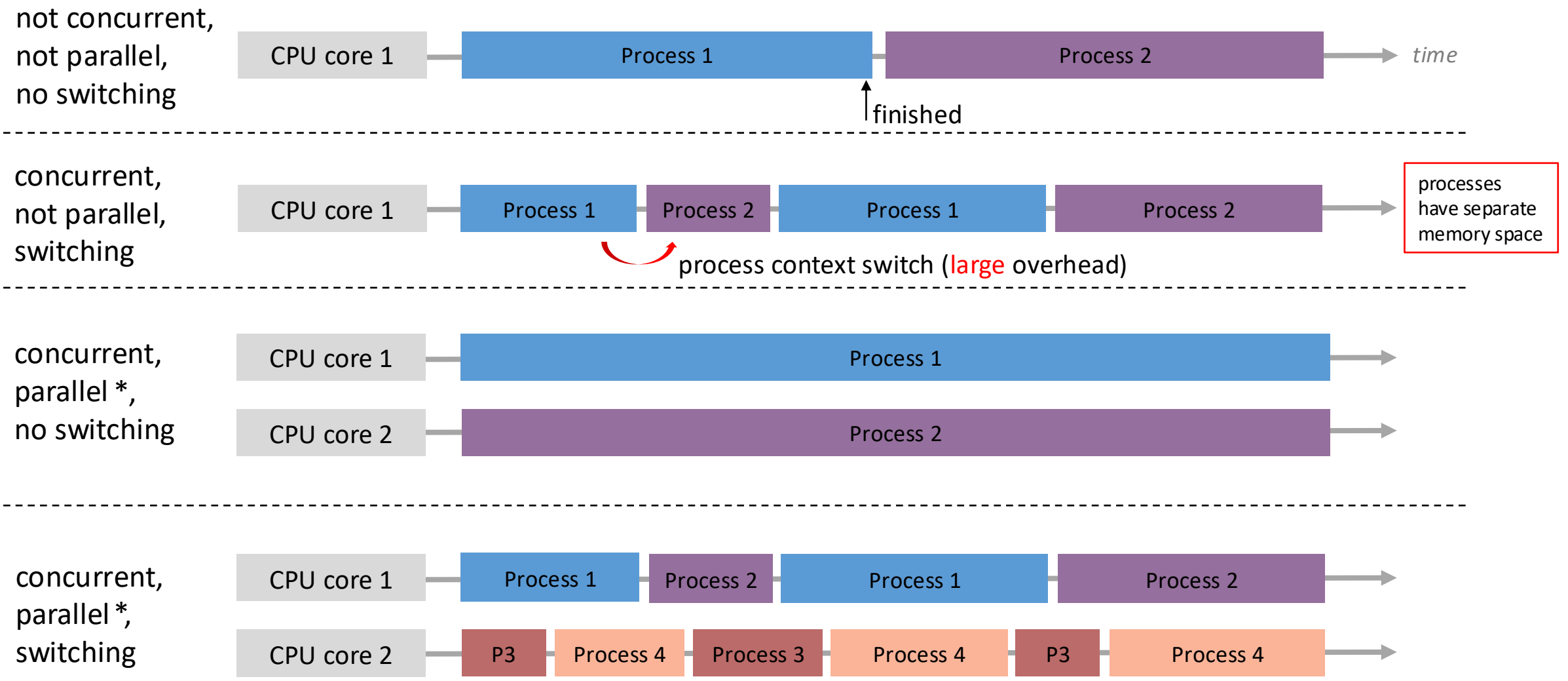


Parallelism

- Parallelism: *multiple processes execute simultaneously on different CPU cores.*
- Concurrency: *multiple processes are active at the same time and make progress, though not necessarily simultaneously.*
- Parallelism implies concurrency, but concurrency does not imply parallelism



Sequential, concurrent, parallel

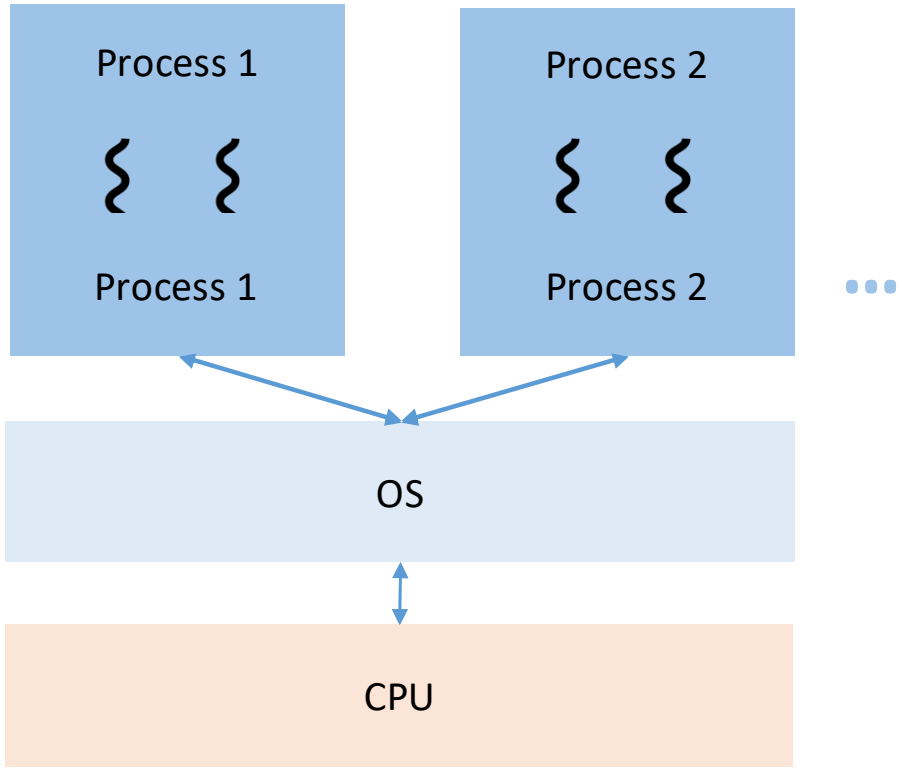


*multi-core and multi-processor systems

Multithreading

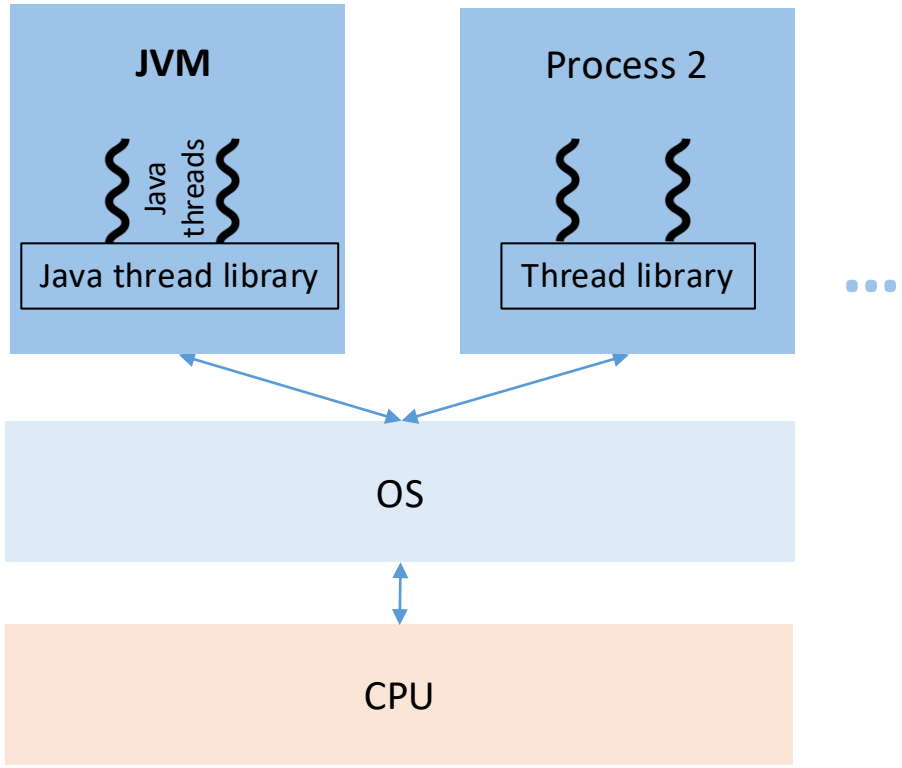
Processes and threads

- A process can have multiple user-level threads, that act as units of computation within the process
- These threads are managed by a user-space library



Processes and threads

- A process can have multiple user-level threads, that act as units of computation within the process
- These threads are managed by a user-space library
- E.g., a JVM process can have Java threads managed by the JVM using Java Thread library



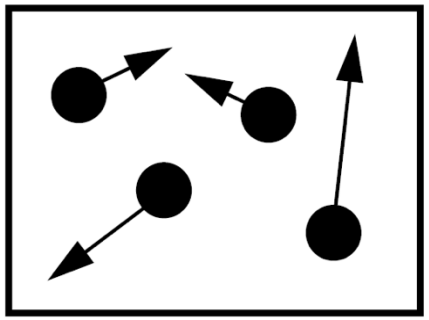
Usage of multithreading

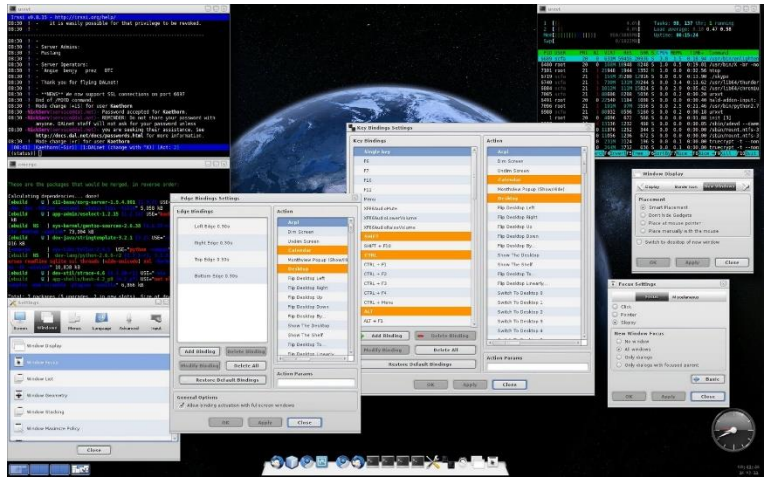
Reactive systems – constantly monitoring

More responsive to user input – GUI application can interrupt a time-consuming task

Server can handle multiple clients simultaneously

Generally: take advantage of multiple CPUs/cores





Threads

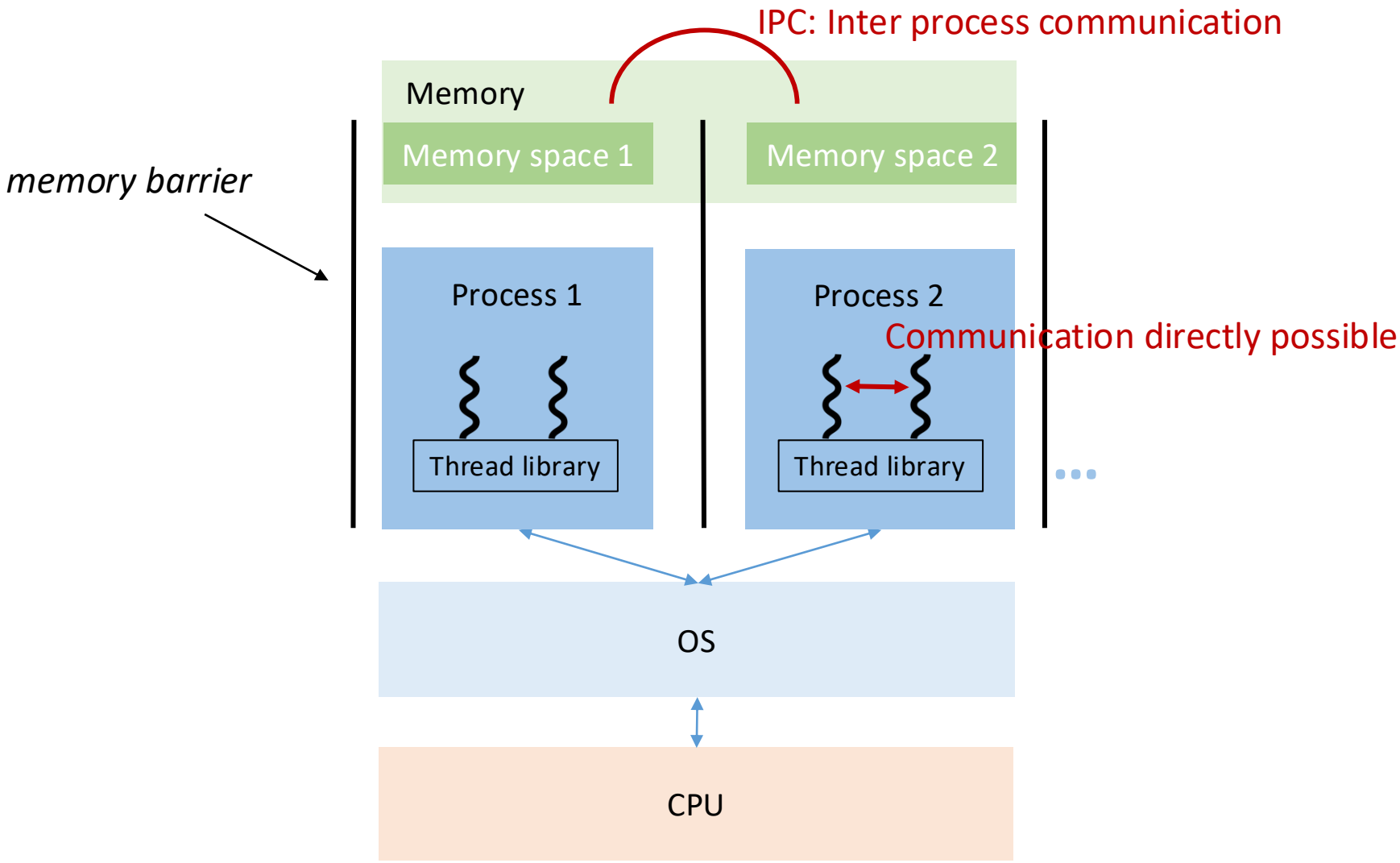
Threads (of control) are

- Independent sequences of execution
- Running in the same OS process

Multiple threads share the same address space

- Threads are not shielded from each other
- Threads share resources and can communicate more easily

Threads share address space



Threads

Threads (of control) are

- independent sequences of execution
- running in the same OS process

Multiple threads share the same address space

- Threads are not shielded from each other
- Threads share resources and can communicate more easily

More vulnerable for programming mistakes

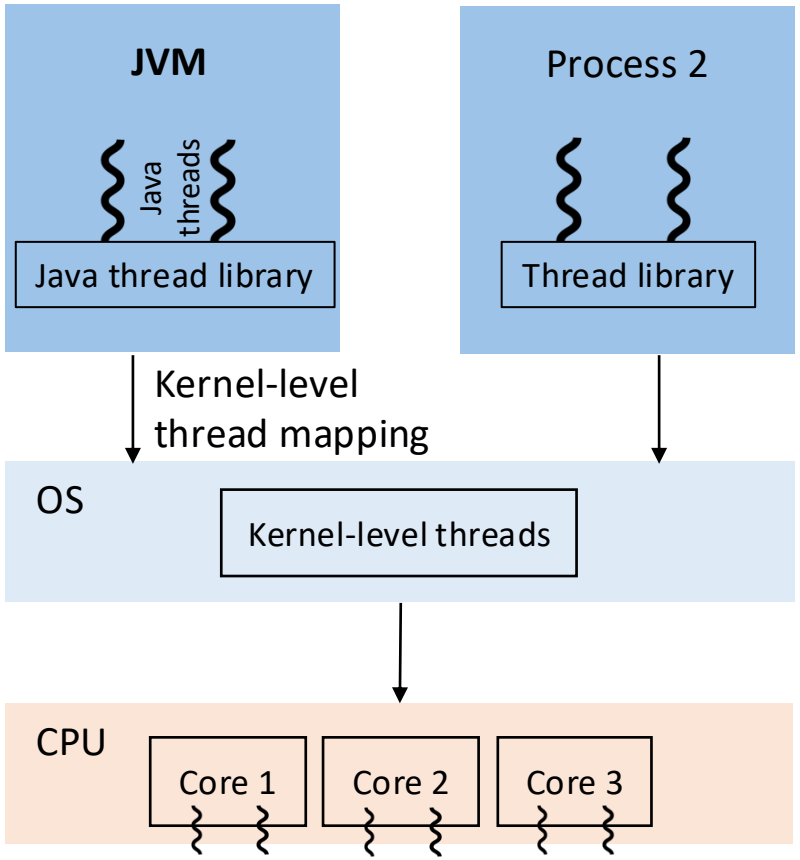
Context switching between threads is efficient

- No change of address space
- No automatic scheduling
- No saving / (re-)loading of PCB (OS process) state

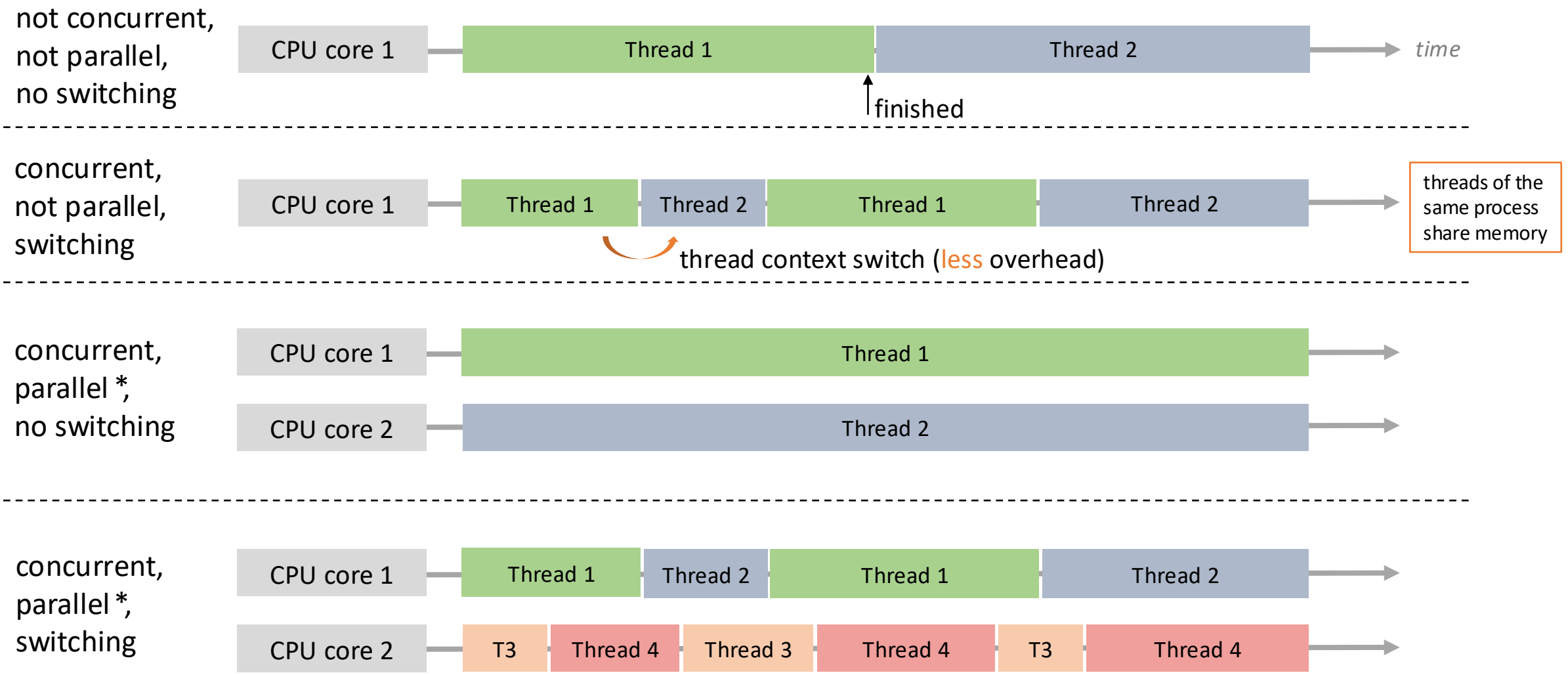
Smaller overhead

Threads across different levels

- **User-level thread:** Managed by the application using a thread library
  - Java thread, managed by JVM using Java Thread API
  - Kernel sees the entire JVM process as a single unit, not individual Java threads
- **Kernel-level thread:** Managed by the OS
  - In modern JVM implementations, Java threads are mapped to kernel-level threads (one-to-one)
  - OS can schedule them on different cores
- **CPU-level thread**:
  - Enables execution of threads simultaneously on physical cores (simultaneous multithreading, hyper-threading)
  - E.g., 4 physical cores appear to the OS as 8 logical cores)



Sequential, concurrent, parallel



*multi-core and multi-processor systems

Terminology

Process: Independently running instance of a program/application, typically on the operation system level. Similar to a thread, but usually more heavy-weight (since a whole program) and encapsulated in memory.

Process context: All state associated with a process, including CPU state (registers, program counter), program state (stack, heap, resource handles), and additional management information.
A thread also has a context, but it is typically much smaller.

Thread: In general, an independent (i.e. capable of running in parallel) unit of computation that executes a piece of code. The concept of threads exists on various levels: hardware (CPU), operating systems, programming languages. In Java, thread also refers to an instance of the Thread class.

Context switch: Given a computation unit (CPU), a context switch denotes the action of switching the unit from one computation to another. Typically refers to switching between processes, but can also refer to switching between threads.

Context switch overhead: Depending on the size of the context ("large" for a process, "small" for thread), a context switch might be computationally expensive, i.e. require comparably much CPU time and memory. CPU needs to store to save the local data, program pointer etc. of the current thread/process, and load the local data, program pointer etc. of the next thread/process to execute.

https://cgl.ethz.ch/teaching/parallelprog26/pages/terminology.html

Terminology

Scheduling: A management process, e.g. on the operating system level, that performs context switches. I.e. it interrupts/pauses/sends to sleep the currently running process (or thread), performs a context switch, and selects the next process (or thread) to run. Schedulers typically do not give guarantees when and how often they act, who gets selected next, etc.

Concurrency: Dealing with multiple things at the same time. Multiple tasks make progress at the same time but not necessarily executing at the same time. Reasoning about and managing shared resources.

Parallelism: Doing multiple things at the same time. Performing computations simultaneously; either actually, if sufficient computations units (CPUs, cores, ...) are available, or virtually, via some form of alternation

Thread mapping: How a Java/JVM thread is related to an operating system thread. In native threading (most common), each JVM thread is mapped to a dedicated operating system thread.

Java Threads

Java Threads

Every Java program runs inside a JVM process

Every Java program has at least one execution thread: the **main thread**

A JVM process can contain multiple threads created by the program

Threads execute concurrently within the same process

Threads share the **same memory space (heap)**

Each thread has its **own execution stack** (method calls, local variables)

Each thread has its **own instruction stream** (independent execution units within a process)

Create Java Threads: Option 1 (oldest)

Instantiate a subclass of `java.lang.Thread` class

- Override run method (must be overridden)
- run() is called when execution of that thread begins
- A thread terminates when run() returns
- start() method invokes run()
- Calling run() does not create a new thread

```
class ConcurrWriter extends Thread { …
    public void run() {
        // if multiple threads started -> concurrent execution
    }
}
ConcurrWriter writerThread = new ConcurrWriter();
writerThread.start();   // calls ConcurrWriter.run()
```

Creating the Thread object does not start the thread!

Need to actually call start() to start it.

Two threads - concurrent execution

```
class ConcurrWriter extends Thread { …

    public void run() {
        // if multiple threads started -> concurrent execution
        System.out.println("Hello world");
     }
}

ConcurrWriter writerThread1 = new ConcurrWriter();
ConcurrWriter writerThread2 = new ConcurrWriter();

writerThread1.start();    // calls ConcurrWriter.run()
writerThread2.start();    // calls ConcurrWriter.run()
```

writerThread1 and writerThread2 executes run() independently

objects live on heap

When calling start(), the JVM
- creates a new call stack for the thread
- assigns a program counter
- schedules it for execution
- invokes run() on its own stack

We now have:
- Main thread -> own stack
- Thread writerThread1 -> own stack
- Thread writerThread2 -> own stack

Create Java Threads: Option 2 (better)

Implement `java.lang.Runnable`

- Single method: public void run()
- Class implements Runnable

```
public class ConcurrWriter implements Runnable {
      …
      public void run() { …
       // if multiple threads started -> concurrent execution
       }
}

ConcurrWriter writerTask = new ConcurrWriter(); // task
Thread t = new Thread(writerTask);              // executor
t.start();   // calls ConcurrWriter.run()
```

Clear separation between executor and task

Two threads - concurrent execution

```
class ConcurrWriter implements Runnable {
    int x = 0;

    public void run() {
        // if multiple threads started -> concurrent execution
        System.out.println("Hello world");
    }
}

ConcurrWriter writerTask = new ConcurrWriter();

Thread t1 = new Thread(writerTask);    // calls ConcurrWriter.run()
Thread t2 = new Thread(writerTask);    // calls ConcurrWriter.run()
```

Shared instance variable
(both threads use the same task object)

One task object (heap)

Two thread
objects (heap)

Java Threads: some key points

Every Java program has at least one execution thread

- First execution thread calls `main()`

Each call to `start()` method of a `Thread` object creates an actual execution thread

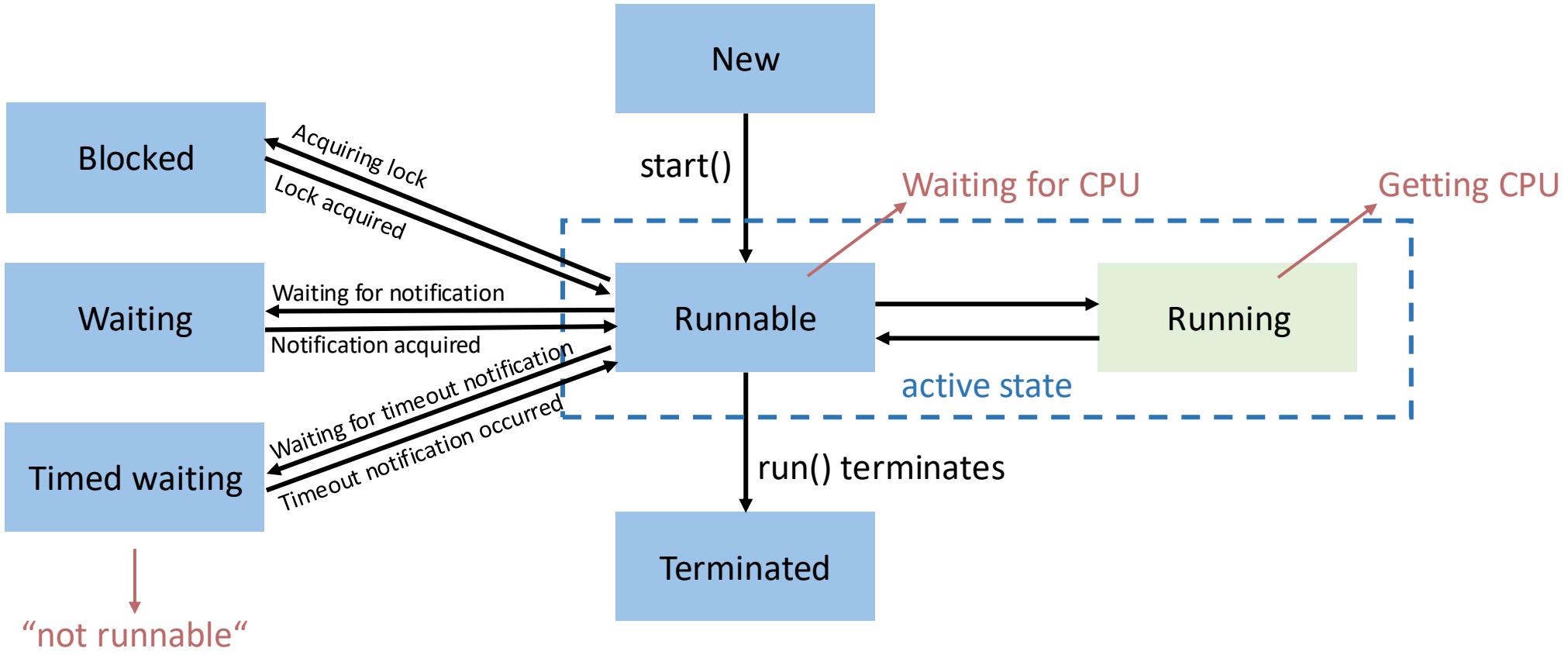
Program ends when all threads (non-daemon threads) finish.

Threads can continue to run even if `main()` returns

Creating a `Thread` object does not start a thread

Calling `run()` doesn't start thread either (need to call `start()`!)

Life cycle of a Thread



Code example: PP-L03-01ParallelCalculator

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1-10

```
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
         this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1-10

```
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
         this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

```
public static void main(String[] args) {
//Launch 10 threads that make the operation with a
different number
  for (int i=1; i <= 10; i++){
    Calculator calculator = new Calculator(i);
    Thread thread = new Thread(calculator);
    thread.start();
}}
```

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1-10

```
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
         this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

```
public static void main(String[] args) {
//Launch 10 threads that make the operation with a
different number
  for (int i=1; i <= 10; i++){
    Calculator calculator = new Calculator(i);
    Thread thread = new Thread(calculator);
    thread.start();
}}
```

Sample output:

....

```
Thread-9: 10 * 10 = 100
Thread-4: 5 * 8 = 40
Thread-4: 5 * 9 = 45
Thread-4: 5 * 10 = 50
Thread-5: 6 * 7 = 42
Thread-2: 3 * 4 = 12
Thread-5: 6 * 8 = 48
Thread-0: 1 * 5 = 5
```

....

Example: The parallel calculator

Create 10 threads: each calculates and prints multiplication tables between 1-10

```
public class Calculator implements Runnable {
   private int number;
   public Calculator(int number) {
         this.number = number;
   }
   public void run() { // Override run()
      for (int i = 1; i <= 10; i++){
         System.out.printf("%s: %d * %d = %d\n",
         Thread.currentThread().getName(),
         number,i,i*number);
}}}
```

```
public static void main(String[] args) {
//Launch 10 threads that make the operation with a
different number
  for (int i=1; i <= 10; i++){
    Calculator calculator = new Calculator(i);
    Thread thread = new Thread(calculator);
    thread.start();
}}
```

Sample output:

....

```
Thread-9: 10 * 10 = 100
Thread-4: 5 * 8 = 40
Thread-4: 5 * 9 = 45
Thread-4: 5 * 10 = 50
Thread-5: 6 * 7 = 42
Thread-2: 3 * 4 = 12
Thread-5: 6 * 8 = 48
Thread-0: 1 * 5 = 5
```

....

Note that threads do not appear in the order they were created…

The parallel calculator: Observations

- The output order changes between runs
- All outputs are correct
- The program contains multiple possible execution orders

Why does the order change?

Interleaving and parallelism

- Threads may execute at the same physical time on different CPU cores
- Even with true parallel execution, individual operations complete at specific moments in time
- -> The operations of multiple threads form a single global timeline
- The operations of different threads may be **interleaved** in many possible ways
- Different relative orderings of operations are possible
- Concurrent programs may have multiple possible execution orders
- The execution order is **non-deterministic**

Terminology

Interleaving: Given multiple threads, each executing a sequence of instructions, an interleaving is a sequence of instructions obtained by merging the individual sequences of instructions from the threads. The order of instructions within each thread is preserved, but instructions from different threads can appear in any order in the final sequence.

Non-deterministic order: Refers to the situation where the sequence or timing of operations performed by different threads is not fixed or predictable. The execution order of these operations may vary each time the program runs, due to factors like thread scheduling, system load, or the operating system's thread management. In the context of interleaving, this is not always a problem, as different interleavings can lead to correct results, provided that the program is designed to handle concurrency safely.

(Some) Useful Thread attributes and methods

ID: this attribute denotes the unique identifier for each Thread.

```
Thread t =  Thread.currentThread();  // get the current thread
System.out.println("Thread ID" + t.getId()); // prints the current ID.
```

Name: this attribute denotes the name of Thread.

```
t.setName("PP" + 2025);   // can be modified like this
```

Priority: denotes the priority of the thread. Threads can have a priority between 1 and 10:

JVM uses the priority of threads to select the one that uses the CPU at each moment

```
t.setPriority(Thread.MAX_PRIORITY);   // updates the thread's priority
```

Status: denotes the status the thread is in: one of new, runnable, blocked, waiting, time waiting, or terminated (we will discuss the different statuses in more detail later):

```
if (t.getState() == State.TERMINATED)  //check if thread's status is terminated
```

Code example: PP-L04-01ThreadInfo

Using Thread states and priorities

```
public static void main(String[] args) {
// Launch 10 threads to do the operation, 5 with the max
// priority, 5 with the min
Thread threads[] = new Thread[10];
Thread.State status[] = new Thread.State[10];

for (int i=0; i<10; i++){
   threads[i]=new Thread(new Calculator(i));
   if ((i%2)==0){
      threads[i].setPriority(Thread.MAX_PRIORITY);
   } else {
      threads[i].setPriority(Thread.MIN_PRIORITY);
   }
   threads[i].setName("Thread "+i);
} ...
```

High-priority threads should be favored by the scheduler and allocated more CPU time

Cont'd on next slide

Using Thread states and priorities

```
try (FileWriter file = new FileWriter(".\\data\\log.txt");PrintWriter pw = new PrintWriter(file);
for (int i=0; i<10; i++){
   pw.println("Main : Status of Thread "+i+" : "+threads[i].getState());
   status[i]=threads[i].getState();
   threads[i].start();
}
boolean finish=false;
while (!finish) {
   for (int i=0; i<10; i++){
     if (threads[i].getState()!=status[i]) {
       writeThreadInfo(pw, threads[i],status[i]);
       status[i]=threads[i].getState();
     }
   }
...
```

write thread states into the log file

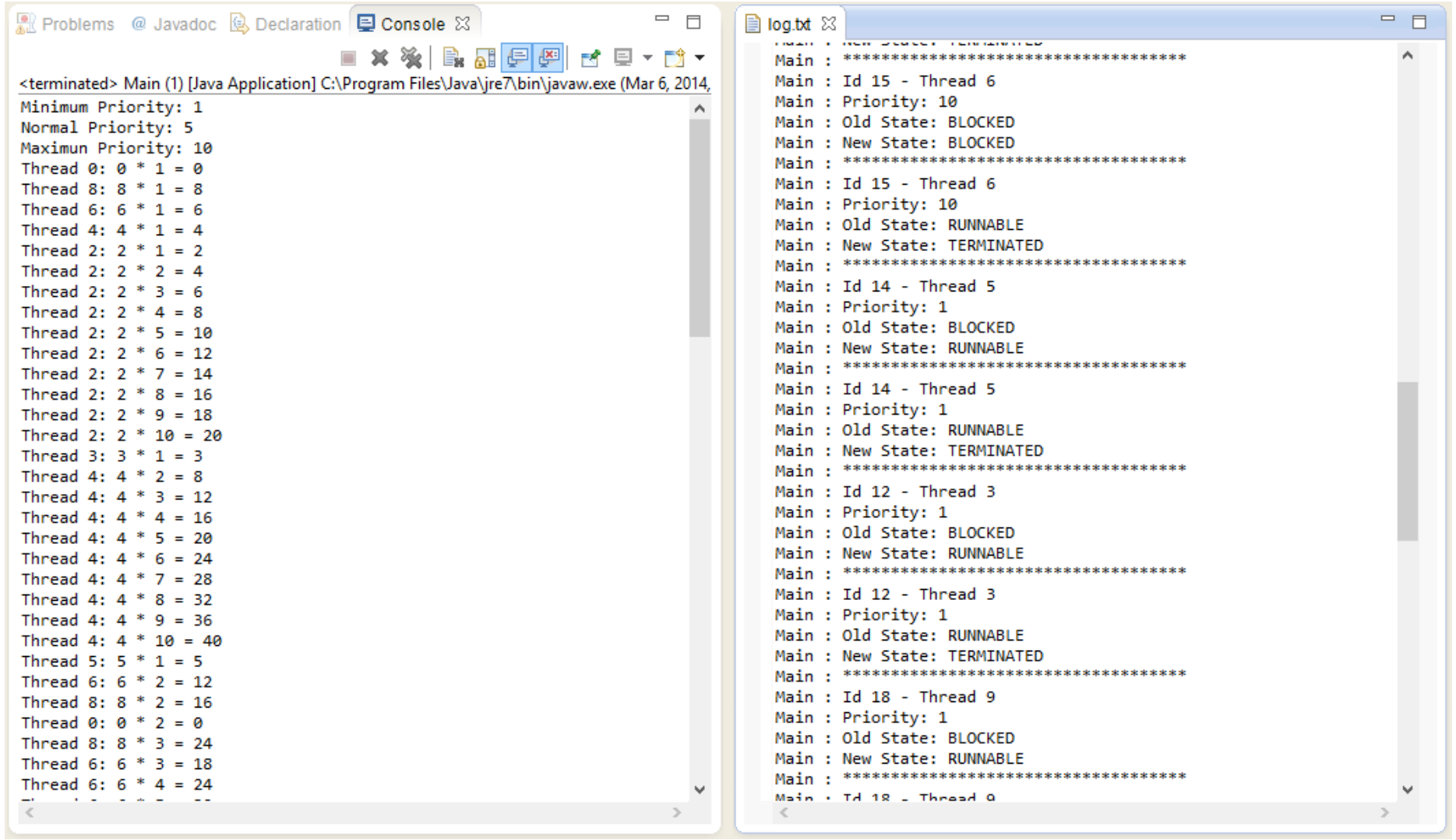
Cont'd on next slide

Using Thread states and priorities

```
   ...
   finish=true;
   for (int i=0; i<10; i++){
      finish=finish &&(threads[i].getState()==State.TERMINATED);
   }
}//end while

} catch (IOException e) {
e.printStackTrace();
}
```

Thread priorities: Output

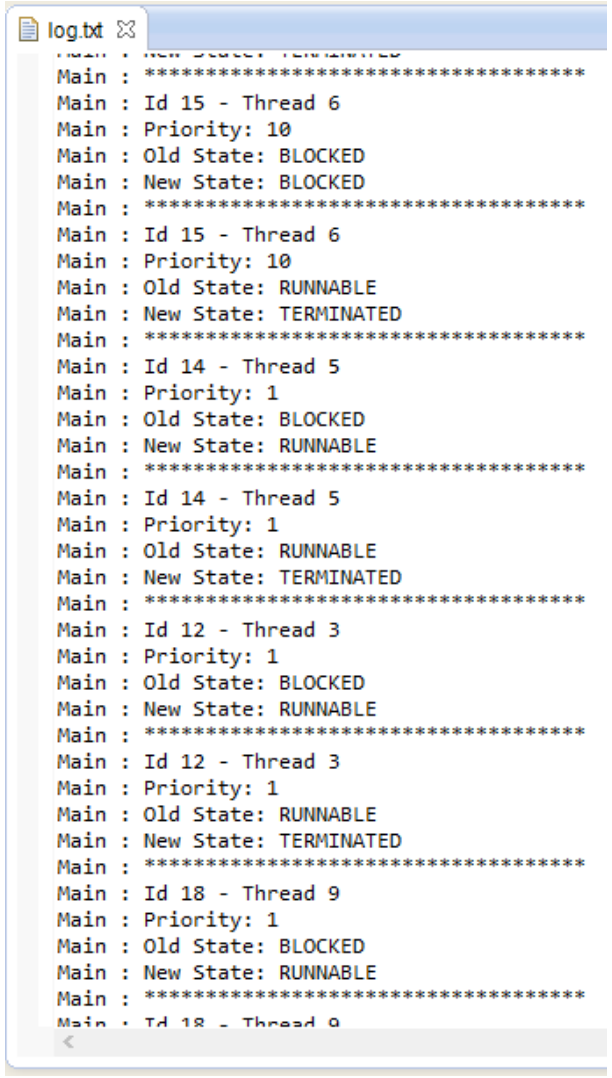


Thread priorities: Observations

Parallel calculators perform I/O (print)
→ most threads typically blocked

High-priority threads typically finish before low-priority thread

JVM assigns priorities, but OS scheduler ultimately decides which thread gets CPU time


log.txt

```
Main : **********************************
Main : Id 15 - Thread 6
Main : Priority: 10
Main : Old State: BLOCKED
Main : New State: BLOCKED
Main : **********************************
Main : Id 15 - Thread 6
Main : Priority: 10
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : **********************************
Main : Id 14 - Thread 5
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : **********************************
Main : Id 14 - Thread 5
Main : Priority: 1
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : **********************************
Main : Id 12 - Thread 3
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : **********************************
Main : Id 12 - Thread 3
Main : Priority: 1
Main : Old State: RUNNABLE
Main : New State: TERMINATED
Main : **********************************
Main : Id 18 - Thread 9
Main : Priority: 1
Main : Old State: BLOCKED
Main : New State: RUNNABLE
Main : **********************************
```

Joining Threads

Results, please!

Common scenario:

- Main thread starts (forks, spawns) several worker threads...
- ... then needs to wait for the worker's results to be available

Previously:

- Busy waiting by spinning (looping) until each worker's state is `TERMINATED`
- Boilerplate code
- Inefficient! Main thread spinning uses up CPU time

```
...
finish = false;
while (!finish) {
  ...
  finish = true;
  for (int i=0; i<10; i++){
    finish = finish && (threads[i].getState() == State.TERMINATED);
  }
}
```

Wake me up when work is done

From main thread's perspective:

- Instead of busily waiting for the results (ready? now ready? now?) ...
- ... go to sleep and be woken up once the results are ready

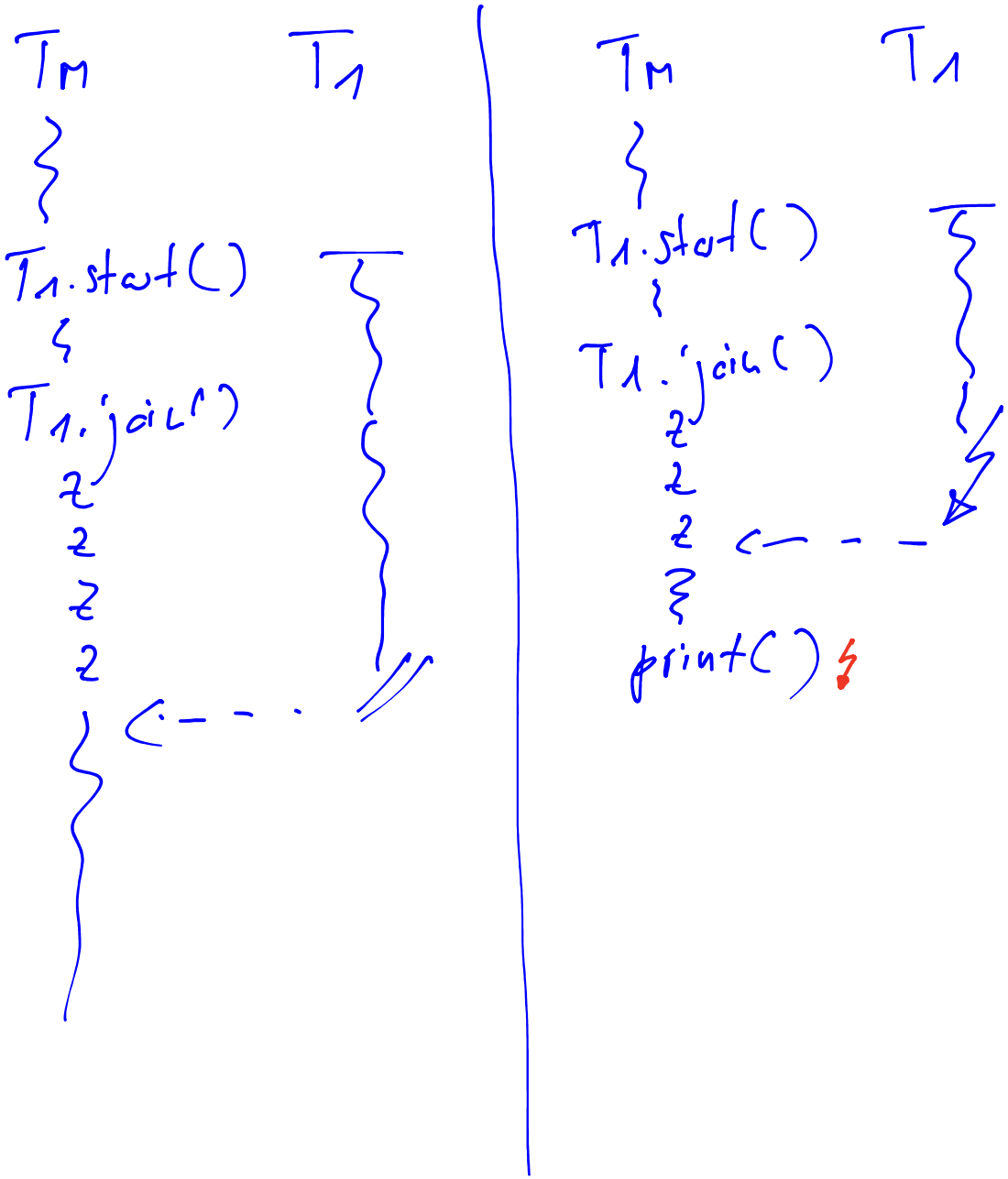
```
...
for (int i=0; i<10; i++) {
  threads[i].join(); // May throw InterruptedException
}
```

Performance trade-off:

- Join (sleep, wakeup) typically incurs context switch overhead
- If worker threads are short-lived, busy waiting may perform better
- Later in the course: `SpinLock`

Question: Is joining `threads[0]`, ..., `threads[9]`optimal?

Code example: PP-L04-02JoinExceptions


Tm
T1
T1.start()
T1.join()
Tm
T1
T1.start()
T1.join()
print()

Exceptions

Exceptions in a single-threaded (i.e. sequential) program terminate the program, if not caught

What if a worker thread throws an exception?

- Exception is (usually) shown on console
- Behaviour of thread.join() is unaffected
- → **Main thread may not be aware of an exception inside a worker thread**

```
public class Worker extends Thread {
  Data result;
  ...

  @Override
  public void run() {
    ...
    // someObject could be null → NPE
    result = calculate(someObject.getData());
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    Worker worker = new Worker(...);
    worker.start();

    worker.join(); // Unaffected
    println(worker.result); // Another NPE
  }
}
```

Setting UncaughtExceptionHandlers

Implementing `UncaughtExceptionHandler` interface allows us to handle unchecked exceptions

Three options:

- Register exception handler with `Thread` object
- Register exception handler with `ThreadGroup` object
- Use `setDefaultUncaughtExceptionHandler()` to register handler for all threads

Handler can then record which threads terminated exceptionally, or restart them, or ...

UncaughtExceptionHandlers: Example

```
public class ExceptionHandler
    implements UncaughtExceptionHandler {

  public Set<Thread> threads = new HashSet<>();

  @Override
  public void uncaughtException(Thread thread,
                                Throwable throwable) {

    println("An exception has been captured");
    println(thread.getName());
    println(throwable.getMessage());
    ...
    threads.add(thread);
  }
}
```

```
public class Main {
  public static void main(String[] args) {
    ...

    ExceptionHandler handler = new ExceptionHandler();

    thread.setUncaughtExceptionHandler(handler);

    ...

    thread.join();

    if (handler.threads.contains(thread)) {
      // bad
    } else {
      // good
    }
  }
}
```

Code example: PP-L04-03Interrupt

Interrupt

```
public static void main(String[] args) {
         Thread buggyThread = new Thread(new RunnableThatTakesForever());
         buggyThread.start();
         …
         buggyThread.join();
         …
}
```

```
private static class RunnableThatTakesForever implements Runnable {
    public void run() {
        try {
                // This is a bug: Threads takes forever to finish
                Thread.sleep(10000000000L);
           } catch (InterruptedException e) {
           …
            }
     }
}
```

thread sleeps forever – main waits forever

Interrupt

```
public static void main(String[] args) {
        Timer timer = new Timer();
        Thread buggyThread = new Thread(new RunnableThatTakesForever());
        buggyThread.start();
        timer.schedule(new MyTimerTask(buggyThread), 8000);

        …
        buggyThread.join();
        …
}
```

calls interrupt on buggyThread

```
private static class MyTimerTask extends TimerTask {
        …
        public void run() {
                System.out.println("Now interrupting the other thread");
                thread.interrupt();
        }
    }
```

invoking interrupt on the thread object

Interrupt

```
private static class RunnableThatTakesForever implements Runnable {
   public void run() {
        try {
                  Thread.sleep(10000000000L); // Bug
        } catch (InterruptedException e) {
                  // interrupt exception in buggyThread
                  System.out.println("Interrupted exception RunnableThatTakesForever");
        }
   }
}
```

For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption