Why do we need threads/processes and a scheduler?

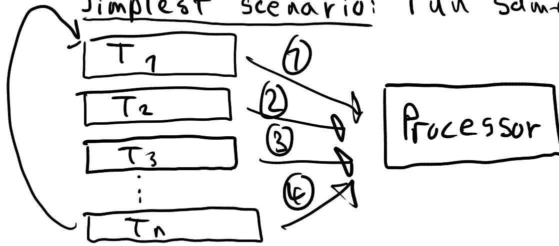
"a collection of one or more programs that do something we want to get done"

Situation:

by task here I mean ↑

We have a processor and want to occupy it with some tasks (for a long time).

Simplest scenario: run same tasks again and again in a fixed order.



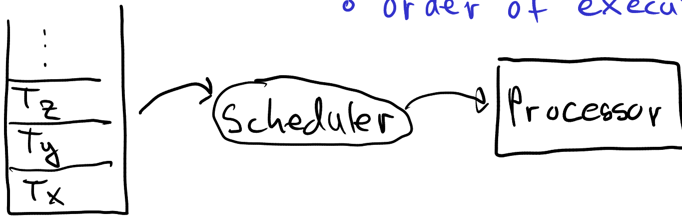
simple table in memory

e.g. a watch
1. measure time
2. update display

More dynamic: allow for tasks to come and go, e.g. via interaction with the outside world (e.g. pressing a button).

→ new problems:
- keep track of different tasks → list
- order of execution? → scheduling

(e.g. we would like make sure to periodically check whether a button was pressed, even if a large number of other tasks are waiting)



tasks would now be stored more as some kind of list

Remember: we assumed up to now that tasks always run from beginning to end one after the other

Remaining problems:
- what if a task is long or gets stuck? (e.g. while loop)
- What if we really need to run something with high priority? (e.g. car breaks in a Tesla)
- What if we want long-running tasks? (e.g. desktop environment, browser)

Make it preemptive (allow for tasks to be interrupted):

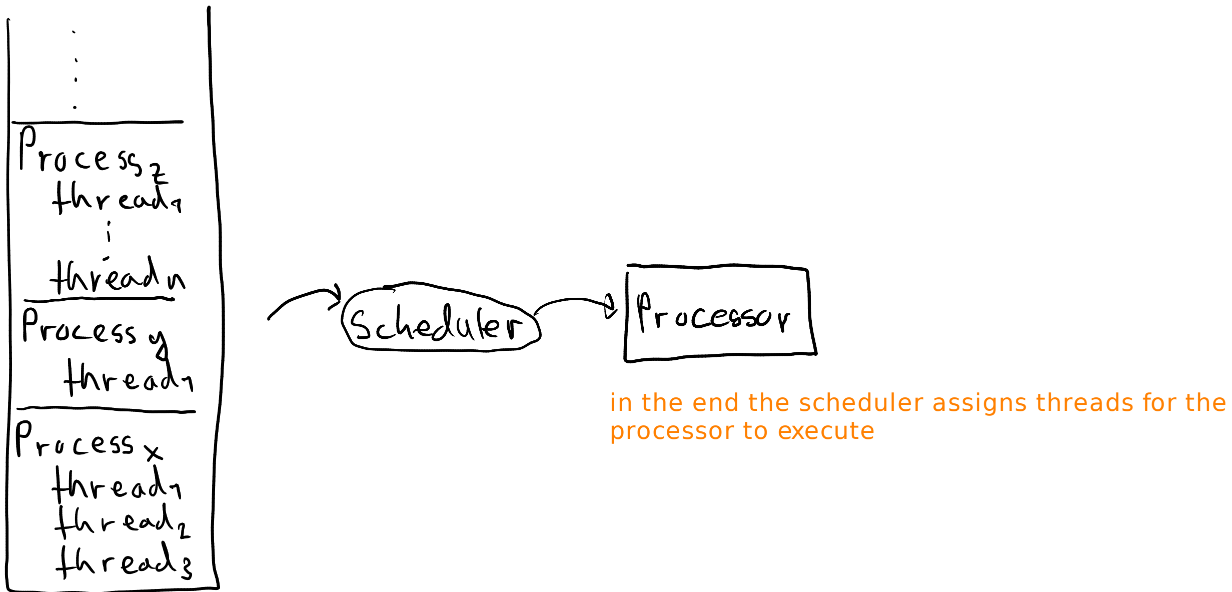
→ how do we do this?

If we interrupt what is being executed on the processor and intend to continue later, we somehow need to remember where we stopped. We now need to specify more precisely what a "task" exactly is, such that we know what "state" or CONTEXT to store when interrupting one task and what state to restore when loading another in its place. Lets call the tasks PROCESSES and the action of interrupting one process to run another instead a CONTEXT SWITCH (between processes).

A process (e.g. a web browser) may want to run multiple unrelated sequences of (machine code) instructions / THREADS of execution on the processor to ...
- allow these instructions to run in parallel (by relying on the scheduler to schedule each of your different threads of execution often enough) e.g. to react to user input while playing your cat video
- gain more computing power by exploiting the fact that your processor has multiple cores

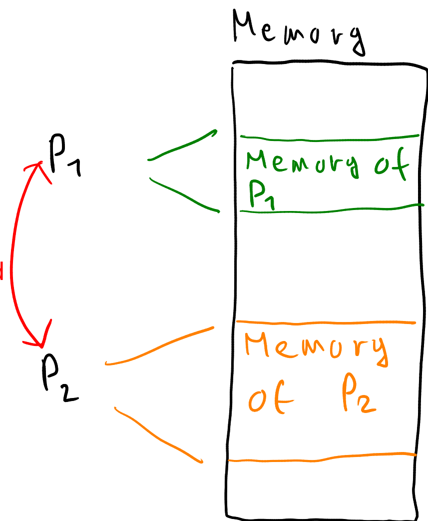
Lets call such a thread of execution a THREAD. Since a thread literally is what points to a sequence of instructions for the processor, whenever the processor is executing something it will be executing the instructions of some thread (belonging to some process). A process could also only have one thread, like a single-threaded Java program.

We can also have a context switch between threads of the same process. Our final view of keeping the processor busy looks something like this:



Processes are generally isolated from each other and for example have different views of memory (this is accomplished with "virtual memory", which you will learn about in DDCA).

A context switch between processes will be relatively EXPENSIVE, since the process' "view of memory" must be saved and restored along with other metadata



However threads of the same process share the memory of the process:

A context switch is comparatively CHEAP, since mostly only CPU registers and the stack (local variables) need to be restored