CIS 452 - Operating Systems Concepts

Nathan Bowman

Images taken from Silberschatz book

Threads

Process is "a program in execution"

Process is a unit of work -- what we can schedule on the CPU

Process is a unit of resource ownership -- processes are assigned memory segments and open files

Process is "a program in execution"

~~Process is a unit of work -- what we can schedule on the CPU~~

Process is a unit of *resource ownership* -- processes are assigned memory segments and open files

In our previous discussions, processes were single-threaded

Threads are finer-grained than processes

Threads, not processes, are schedulable entity on the CPU

Previously, we said a process had:

- Memory regions
- Program counter
- Registers
- Open files and I/O devices

All of the following are shared by every thread within a process

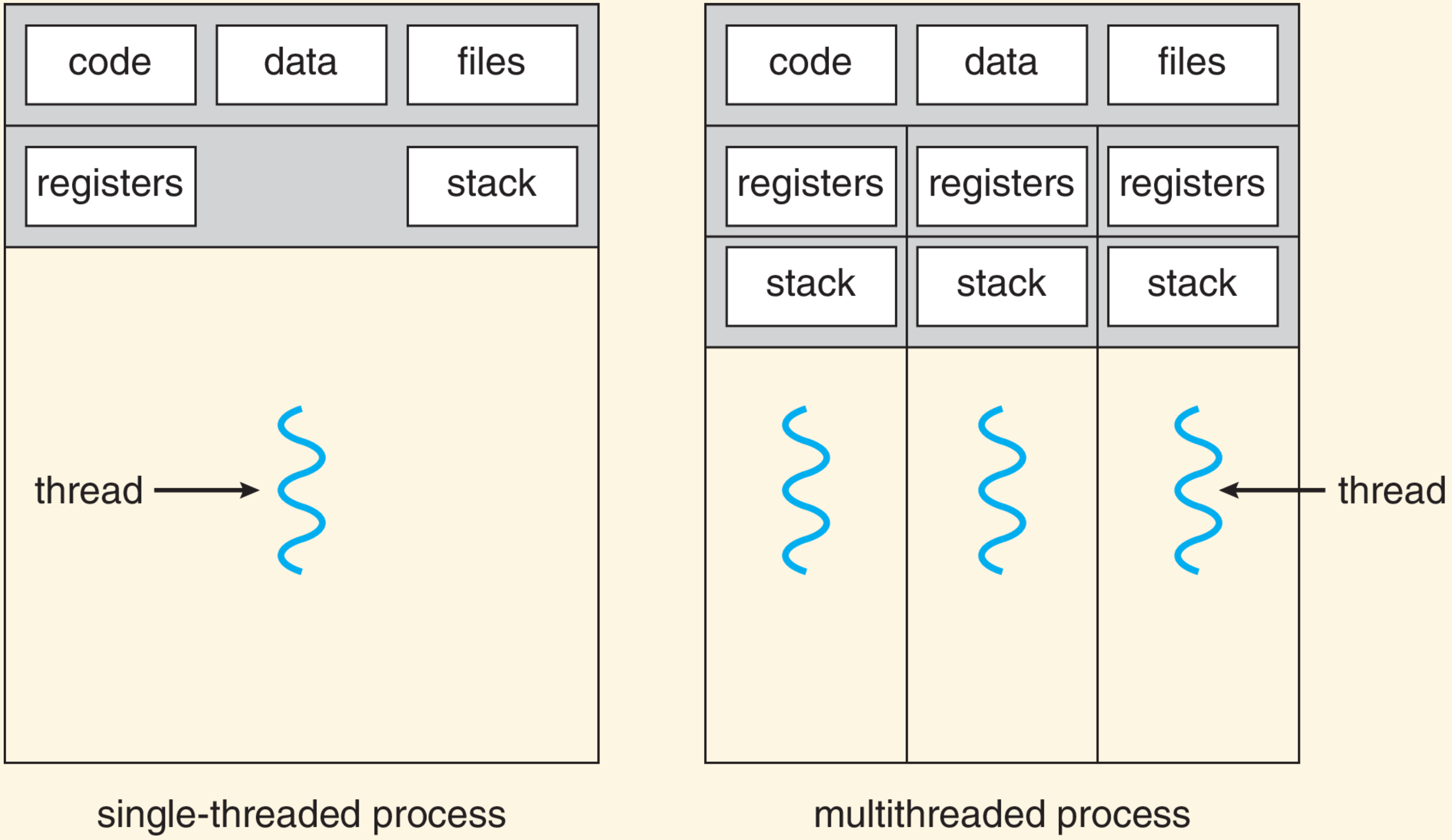
- Memory regions
  - global variables
  - heap
  - ~~stack~~
- ~~Program counter~~
- ~~Registers~~
- Open files and I/O devices

So, it is still fair to see that a process owns these things

The rest of the state is on a per-thread basis

- Stack
- Program counter
- Registers

Each thread also has a unique *thread ID*, much like processes have a PID


code
data
files
registers
stack
thread
single-threaded process
code
data
files
registers
registers
registers
stack
stack
stack
thread
multithreaded process

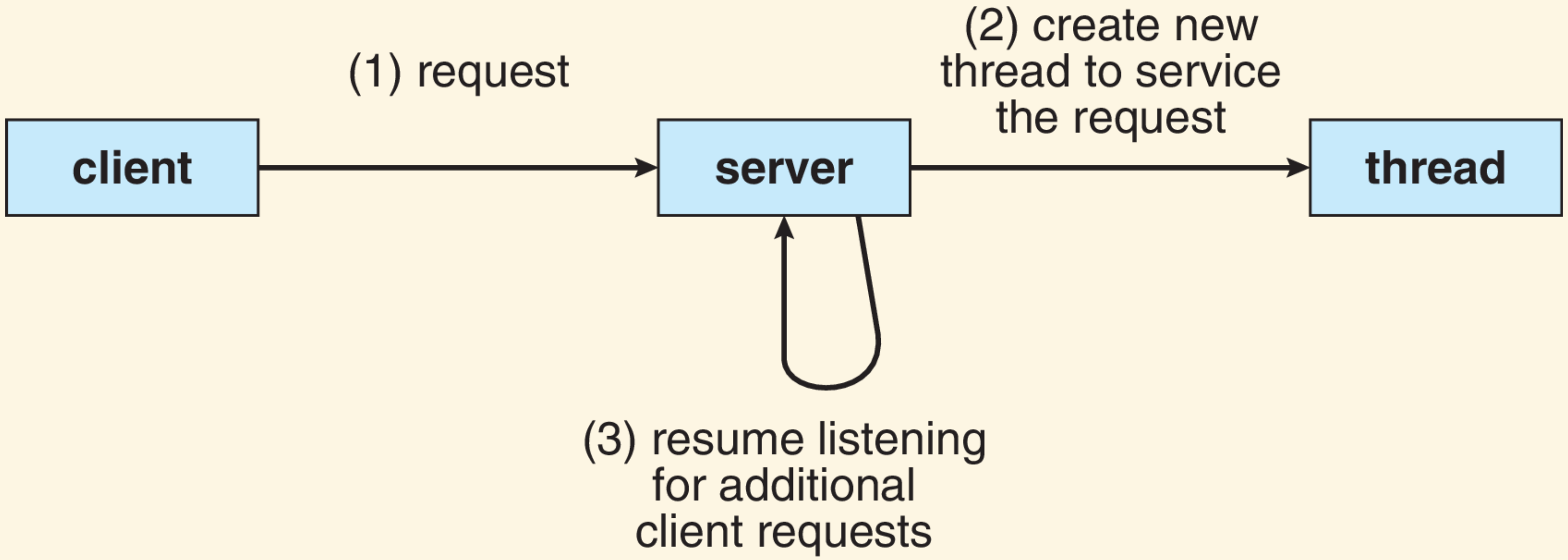
Most modern applications are multithreaded

Word processors, for example, may have threads to

- wait for user input from keyboard
- display information to the screen
- run spelling and grammar checks in the background

Many tasks that could be done with several processes are better done with several threads

Threads are cheaper to create and to context switch than processes


(1) request
(2) create new thread to service the request
client
server
thread
(3) resume listening for additional client requests

Benefits of multithreaded programming

Repsonsiveness -- if one part of a program must wait for I/O or run a computationally expensive function, other parts can still be available to do useful work

Resource sharing -- processes must use special IPC techniques, such as shared memory or message passing, to share data. Threads share by default

Benefits of multithreaded programming

Economy -- two (or more) threads in the same process require fewer resources than two (or more) separate processes. Thread creation and context switching are faster than the same operations on processes

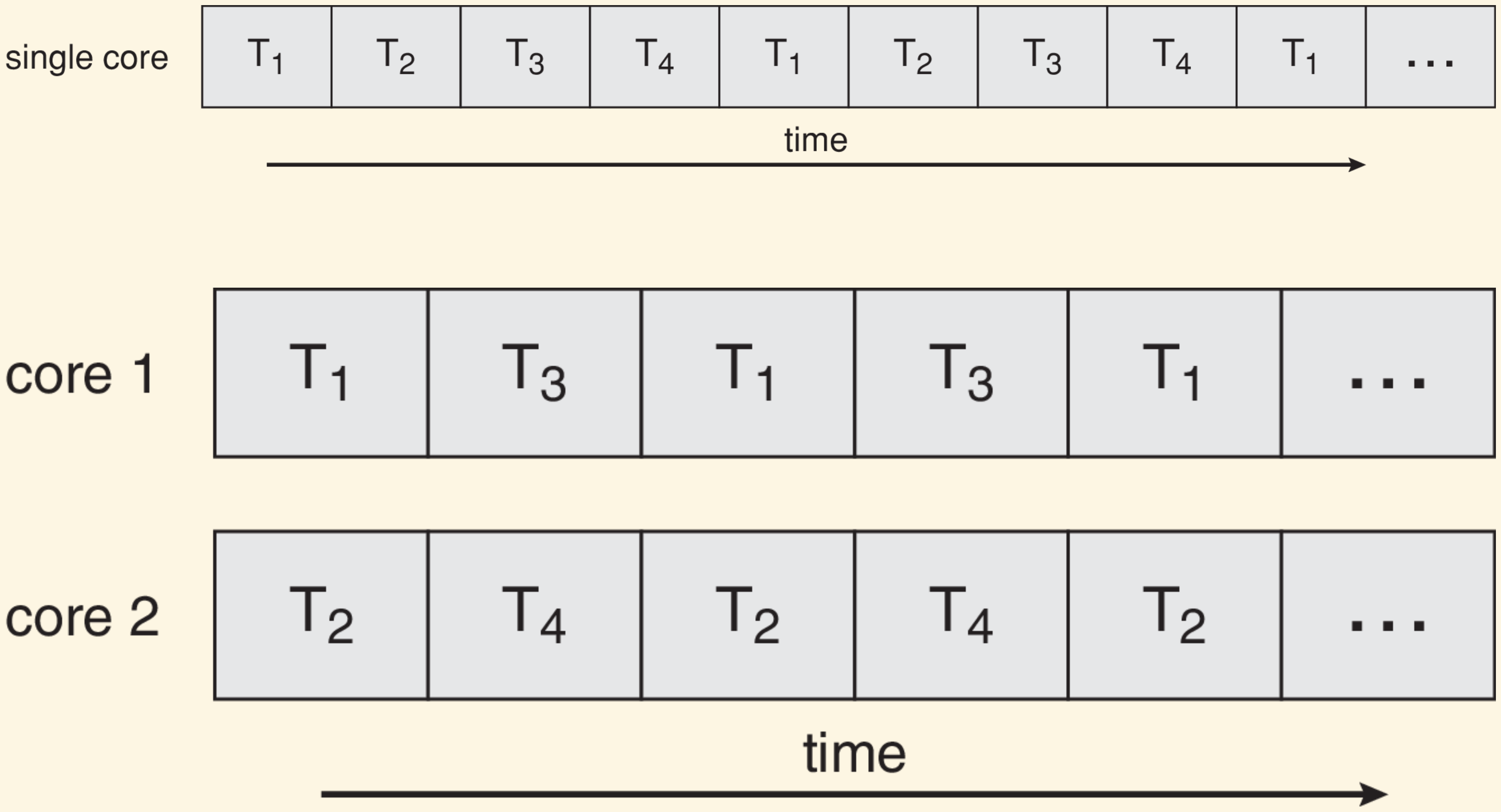
Scalability -- in a multiprocessor architecture, several threads can truly run at the same time, allowing for improved application performance compared to single-threaded programs. (This benefit can also be achieved using multiple processes)

Parallelism vs concurrency

We do not distinguish between multiprocessor and multicore systems

Single processor systems provide *concurrency* because all programs can make progress over time, but they provide only the illusion of *parallelism*

Multicore systems allow improved performance through the use of threads running in parallel


single core
T_1 T_2 T_3 T_4 T_1 T_2 T_3 T_4 T_1 . . .
time
core 1
T_1 T_3 T_1 T_3 T_1 . . .
core 2
T_2 T_4 T_2 T_4 T_2 . . .
time

Parallelism does have its limits though

In any program, part of the work will be unavoidably serial

Let S be the serial portion and N be the number of parallel threads

Amdahl's Law states:

$$speedup \leq \frac{1}{S + \frac{(1-s)}{N}}$$

$$speedup \leq \frac{1}{S + \frac{(1-s)}{N}}$$

$$S = 0.2$$

If $N = 2, speedup \leq 1.666\ldots$

If $N = 5, speedup \leq 2.777\ldots$

As $N \rightarrow \infty, speedup \rightarrow \frac{1}{S}$

Programming with threads does lead to some additional challenges:

- identifying opportunities for parallel work. Sometimes this is obvious, other times not as much
- giving each thread the right amount of work. Making threads too fine-grained can result in wasteful creation of threads that do not have enough work to do
- splitting data between threads in a useful way

Programming with threads does lead to some additional challenges:

- managing data dependencies. We mentioned in IPC lecture that dealing with parallel workers is **hard**
- testing and debugging parallel programs is more difficult because there are things happening simultaneously and they may not always happen in the same order

Problems can be made parallel in two main ways:

- data parallelism -- data is split and same operation performed on each chunk. For example, taking the sum of four different arrays
- task parallelism -- different jobs are performed by different threads, possibly on the same data, but possibly not. For example, finding the sum, maximum, and variance of some array

Real parallel programs are often some combination of these