Concurrent Systems (ComS 527)

Ali Jannesari

Department of Computer Science

Iowa State University, Spring 2023

PROCESSES & THREADS

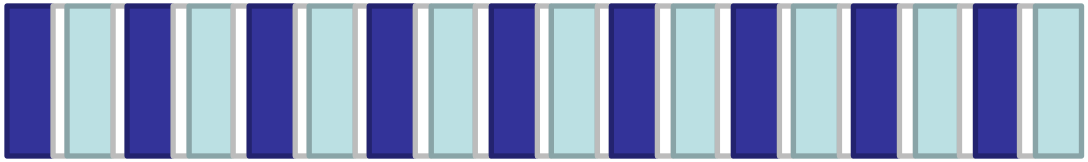
Outline

- Processes vs. threads
- Multithreading models
- On-chip multithreading
- Thread safety

Time sharing

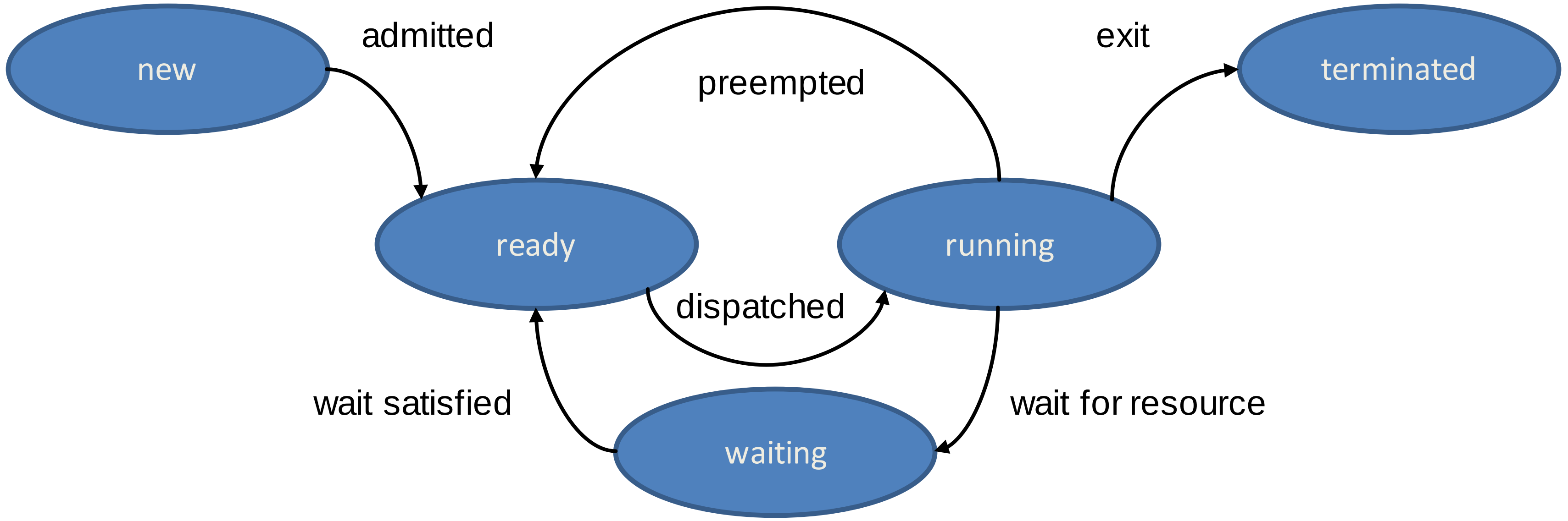
- Also called multitasking
- Logical extension of multiprogramming
- CPU executes multiple processes by switching among them
- Switches occur frequently enough so that users can interact with each program while it is running
- On a multiprocessor, processes can also run truly concurrently, taking advantage of additional processors

Single core

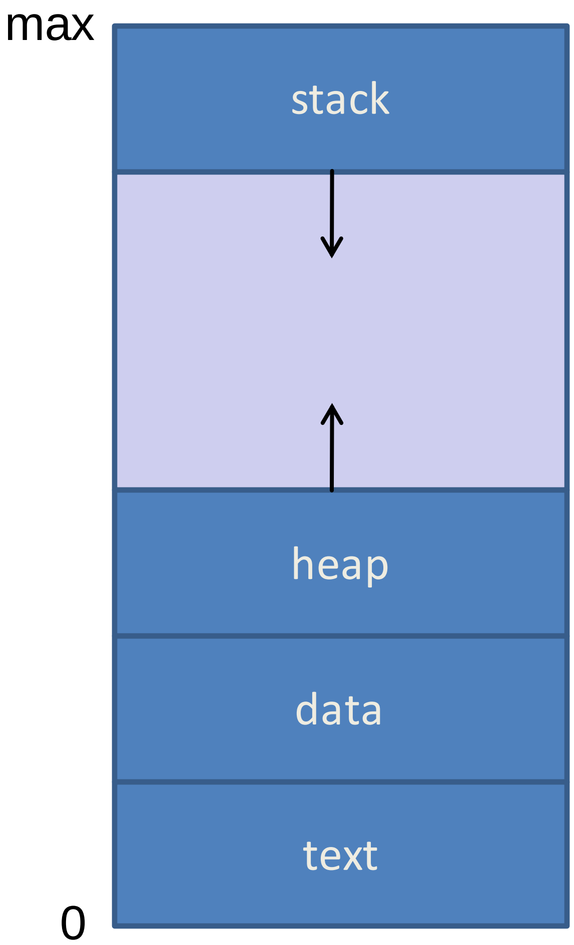


Process

- A process is a program in execution
- States of a process



Processes in memory



Temporary data: function parameters, return addresses, local variables

Dynamically allocated memory

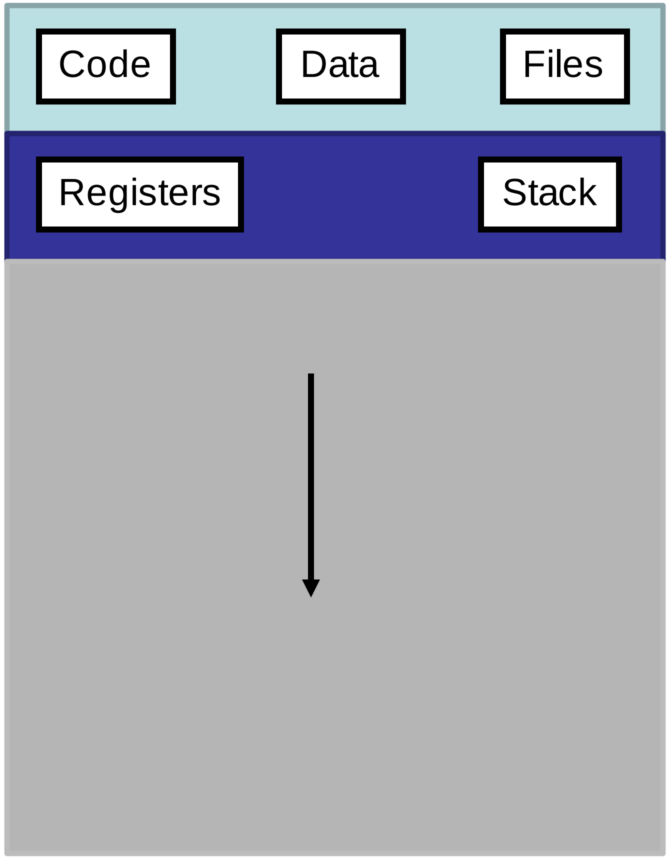
Global variables

Program code

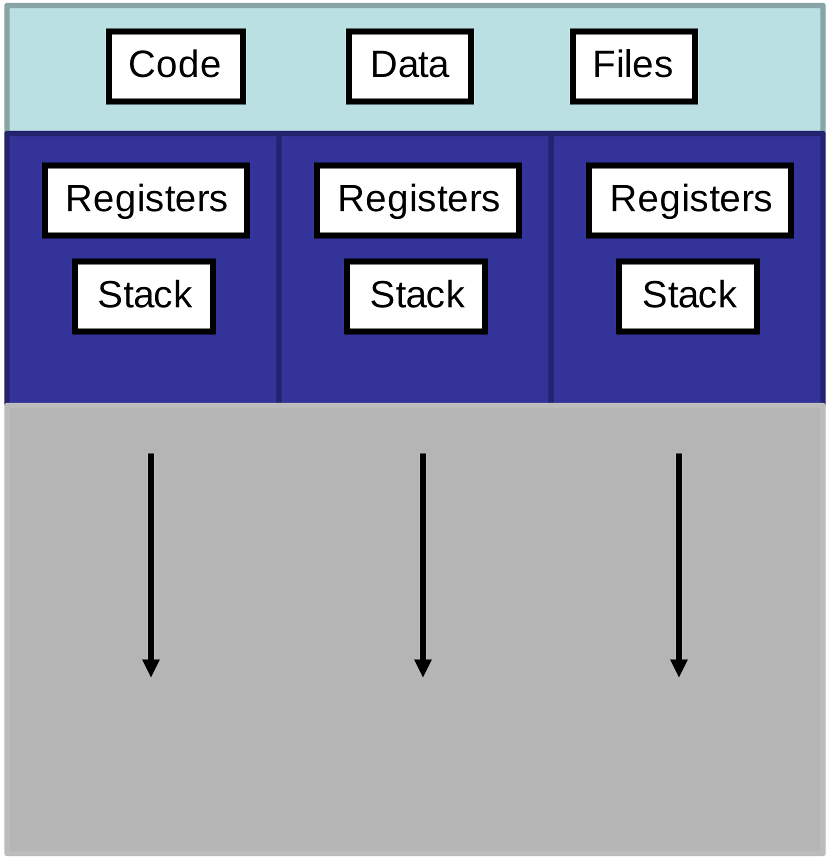
Thread

- Basic unit of CPU utilization
  - Flow of control within a process
- A thread includes
  - Thread ID
  - Program counter
  - Register set
  - Stack
- Shares resources with other threads belonging to the same process
  - Text (i.e., code) section
  - Data section (i.e., address space)
  - Other operating system resources
    - E.g., open files, signals

Single-threaded vs. multi-threaded

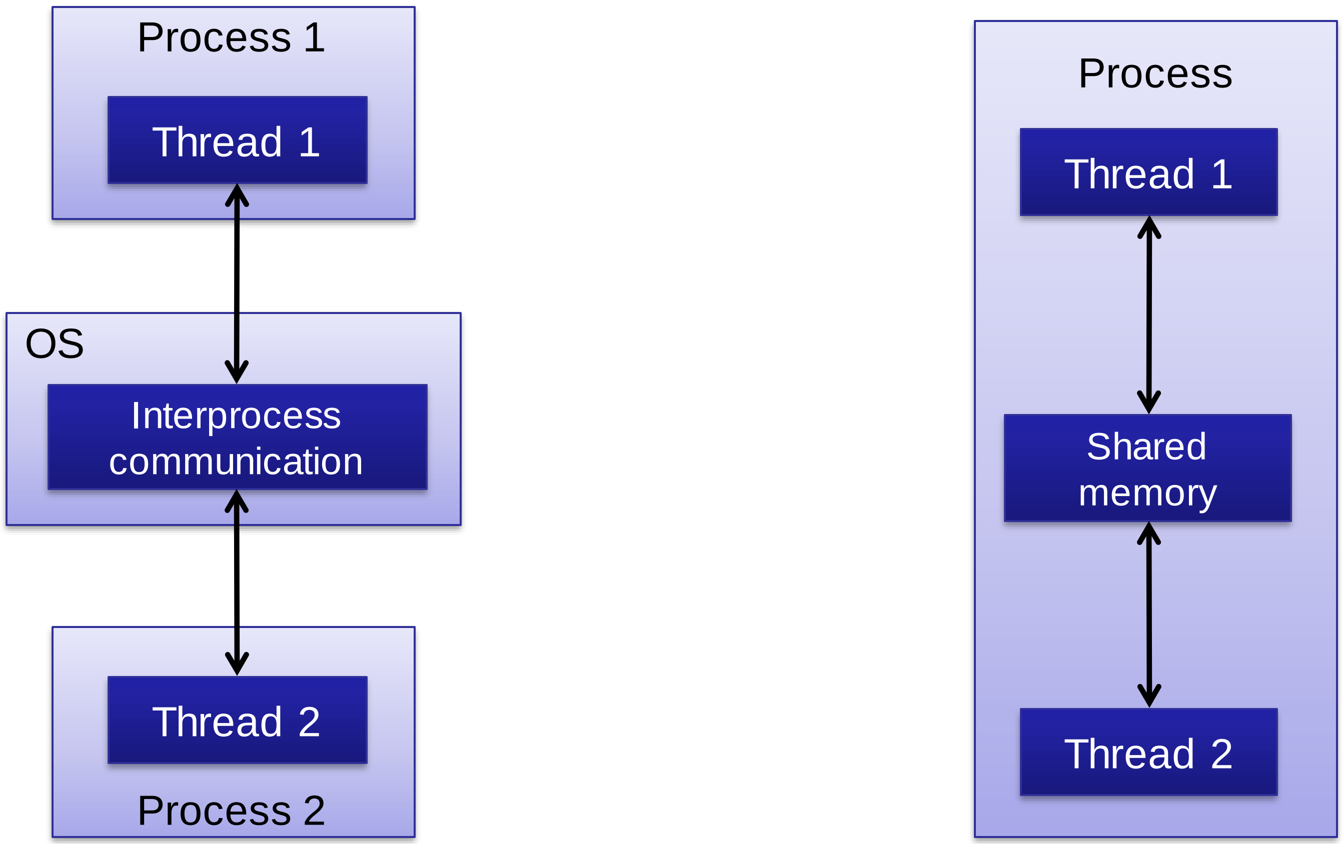


Single-threaded

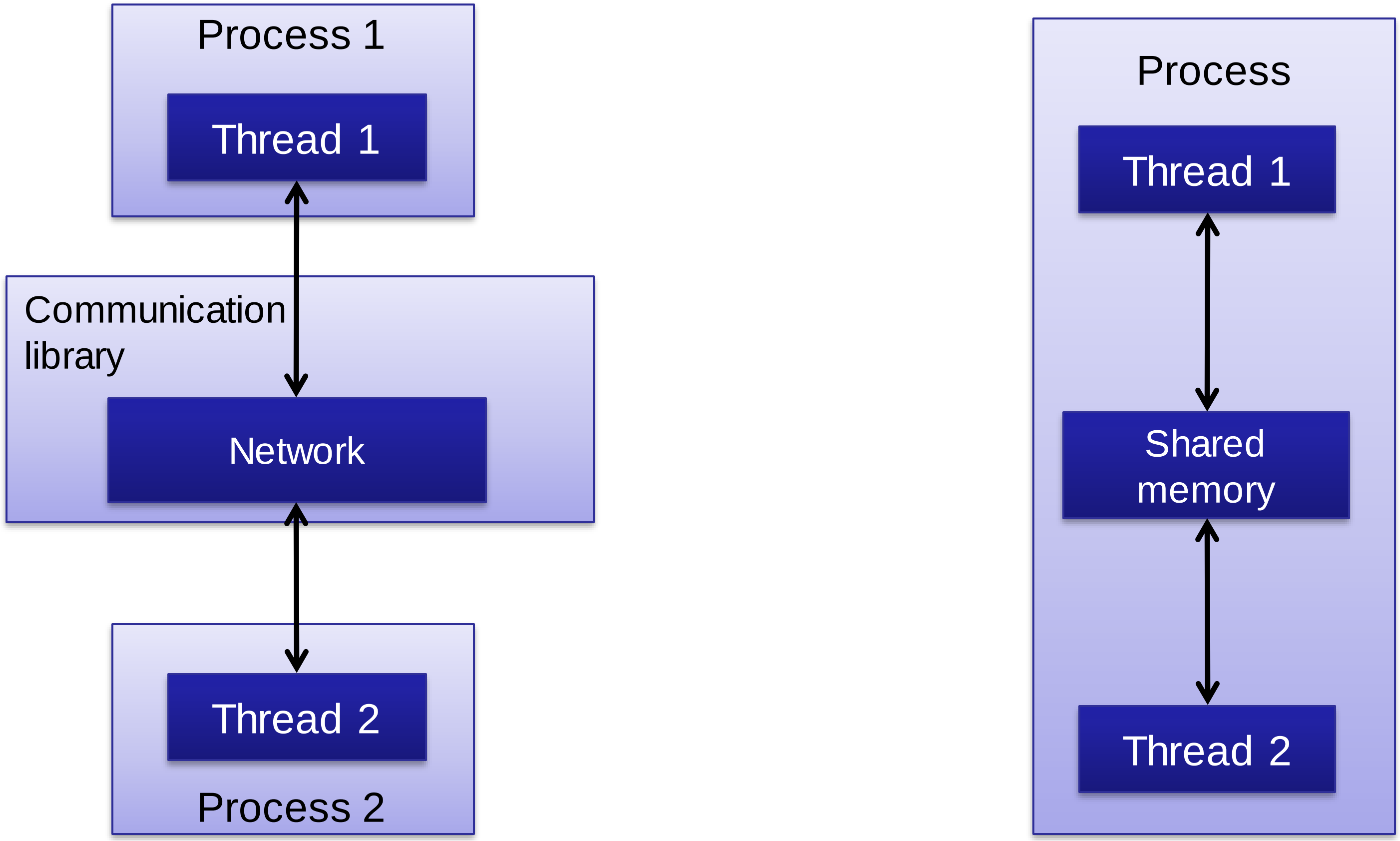


Multi-threaded

Concurrency – processes vs. threads



Concurrency – processes vs. threads (2)



Concurrency – processes vs. threads (3)

Processes

- Communication explicit
- Often requires replication of data
- Address spaces protected
- Parallelization usually implies profound redesign
- Writing debuggers is harder
- More scalable

Threads

- Convenient communication via shared variables
- More space efficient - sharing of code and data
- Context switch cheaper
- Incremental parallelization easier
- Harder to debug – race conditions

Multithreading models

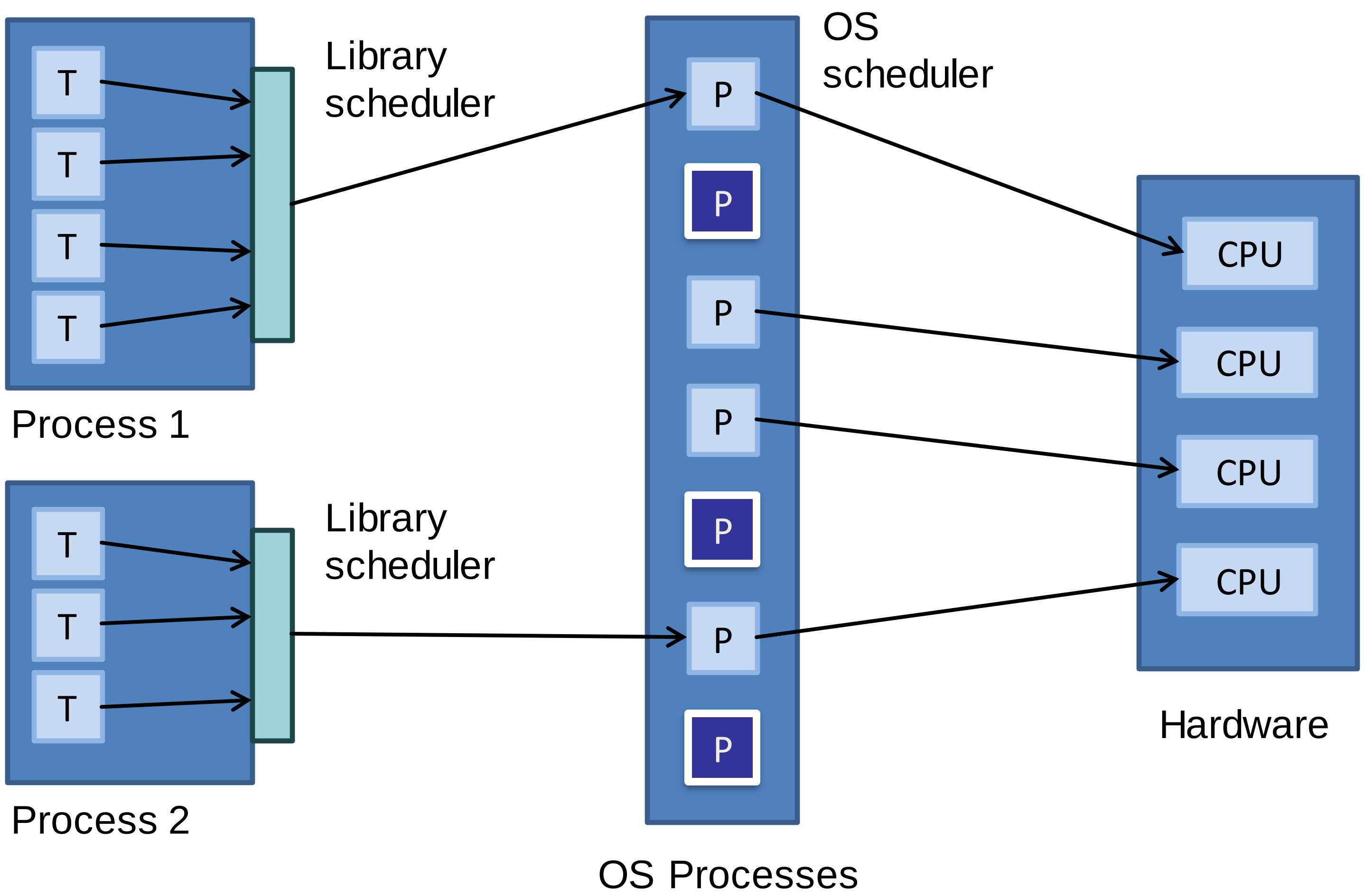
User-level threads

- Supported above the kernel
- Implemented by a thread library
- Creation, scheduling, and management in user space

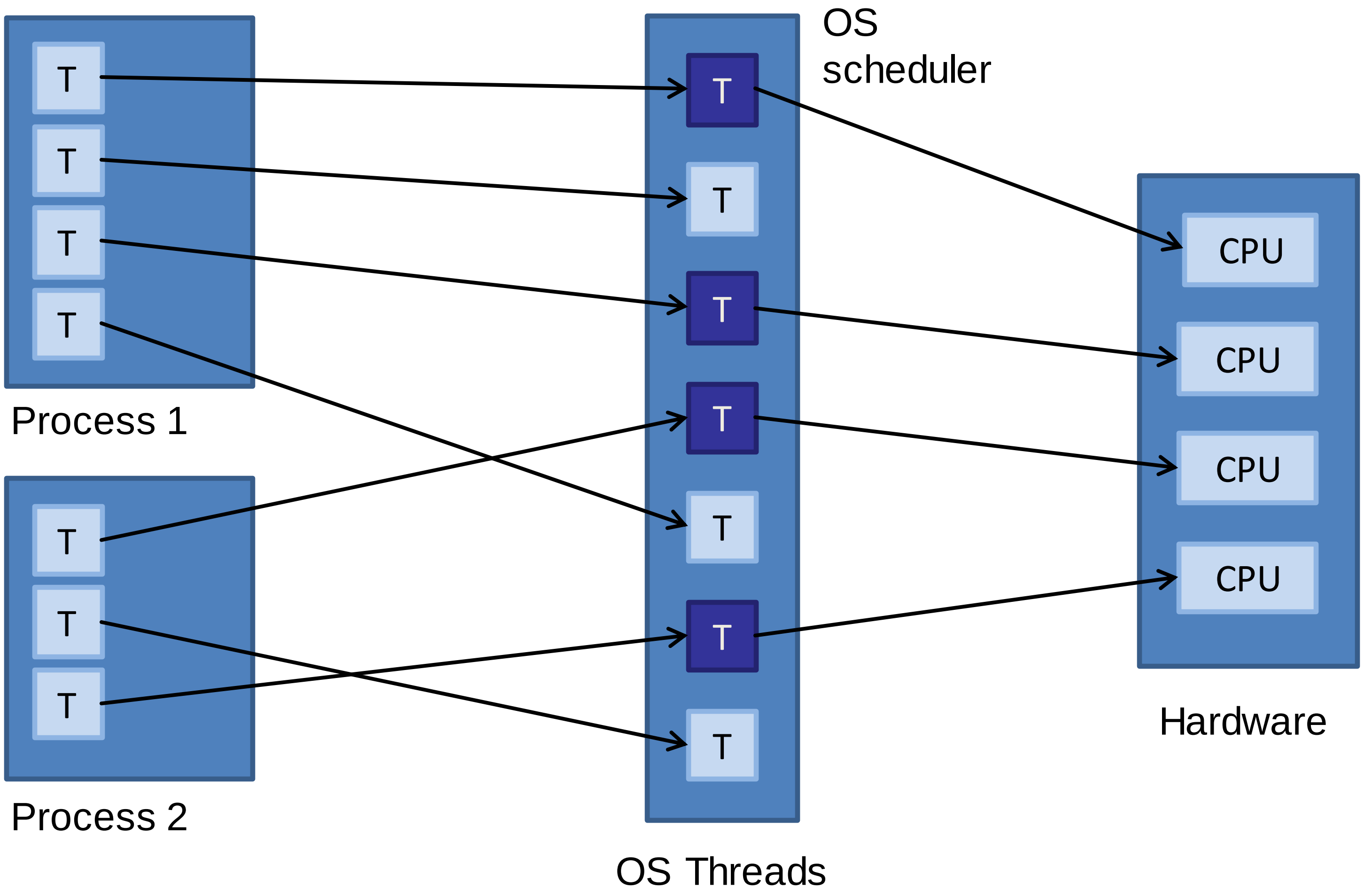
Kernel-level threads

- Supported directly by the operating system
- Creation, scheduling, and management in kernel space
- Virtually all contemporary operating systems support kernel-level threads – including Windows, Linux, Mac OS

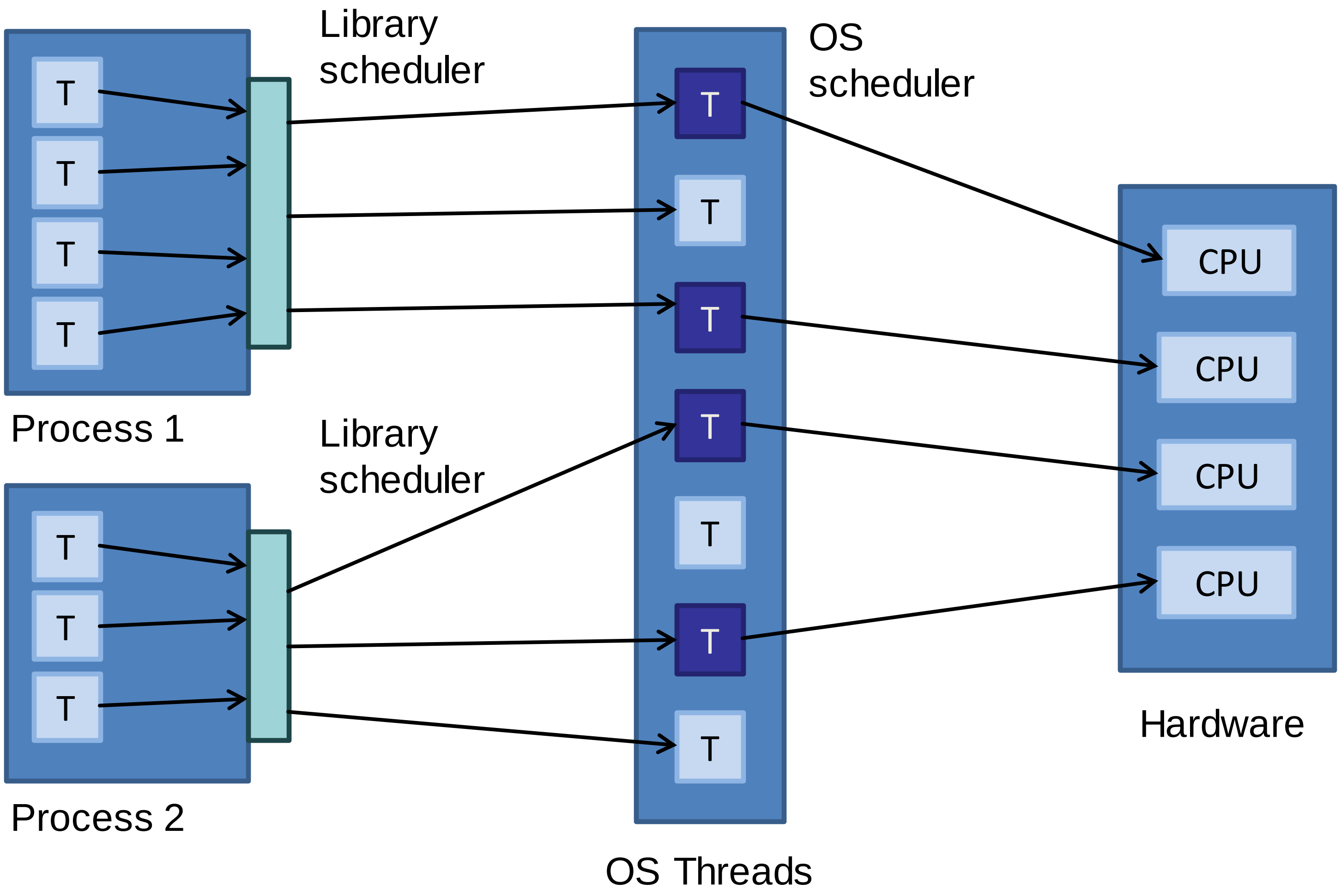
Thread mapping – Many to one



Thread mapping – One to one



Thread mapping – Many to many



Summary thread mapping

- Many to one
  - Efficient thread management, switching between threads cheap
  - Problem: one thread can block entire process
  - No true parallelism on multiprocessors
  - Example: Green threads library for Solaris, GNU Portable Threads
- One to one
  - True parallelism on multiprocessors
  - Creating a user thread incurs overhead of creating a kernel thread
  - Therefore most implementations restrict number of threads
  - Example: Linux and Windows
- Many to many
  - User threads multiplexed to smaller or equal number of kernel threads
  - Allows as many threads as desired
  - Additionally, user-level thread may be bound to kernel thread (two-level)
  - Example: IRIX, HP-UX, Tru64 UNIX, Solaris < 9

On-chip multithreading

- All modern, pipelined CPUs suffer from the following problem
  - When a memory reference misses L1 or L2 caches, it takes a long time until the requested word is loaded into the cache
- On-chip multithreading allows the CPU to manage multiple threads to mask these stalls
- If one thread is stalled, the CPU can run another thread and keep the hardware busy
- Four hardware threads often sufficient to hide latency

Pipelined CPU – Example

Basic five-stage pipeline

Instr. No. \ Clock cycle	1	2	3	4	5	6	7
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX

(IF = Instruction Fetch, ID = Instruction Decode, EX = Execute, MEM = Memory access, WB = Register write back).

In the fourth clock cycle (the green column), the earliest instruction is in MEM stage, and the latest instruction has not yet entered the pipeline.

Source: https://en.wikipedia.org/wiki/Instruction_pipelining

Fine-grained vs. coarse-grained multithreading

3 threads

A1	A2			A3	A4	A5			A6	A7	A8
B1			B2			B3	B4	B5	B6	B7	B8
C1	C2	C3	C4			C5	C6			C7	C8

Fine grained – threads run round-robin

A1	B1	C1	A2	B2	C2	A3	B3	C3	A4	B4	C4

Coarse grained – switch occurs only upon stall

A1	A2		B1		C1	C2	C3	C4		A3	A4

Fine-grained vs. coarse-grained multithreading

- Optimization of course-grained multithreading
  - Switch on any instruction that might cause a stall
  - Avoids wasted cycles
- Both options require bookkeeping
  - Fine-grained MT – attach thread ID to an instruction before it enters the pipeline
  - Coarse-grained MT – also possible to let the pipeline run dry before starting the next thread

Five ways of improving CPU performance

- Increase the clock speed
- Put two cores on a chip
- Add functional units
- Make pipeline longer
- Use on-chip multithreading

On-chip multithreading support can improve performance over-proportionally in comparison to the required extra chip area

Hyperthreading in the Intel Core i7

- Two threads (or processes) can run at once on the same core
- Looks from far like a dual processor in which both CPUs share a common cache and main memory
- However, many hardware resources are shared between threads
- Advantage – enables true concurrency within the same core
- Disadvantage – resource contention (e.g., cache) may lower throughput

Hyperthreading – resource sharing strategies

- Duplication
  - E.g., program counter, table that maps architectural onto physical registers
- Partitioning
  - E.g., slots in queue between stages of a functional pipeline
  - May lead to underutilized resources
- Full sharing
  - More flexible than partitioning but danger of starvation
- Threshold sharing
  - A thread can acquire resources dynamically up to a maximum
  - Compromise between fixed partitioning and full sharing

A function is thread safe…

- … if it can be called by more than one thread without restrictions
  - No further action required by the caller (e.g., malloc() and free() in thread-safe version of libc)
  - Does not protect access to memory under control of caller (e.g., when using memcpy())
- Reasons for unsafe behavior
  - Internal state on function, object, or library level (e.g., errno variable)

```
void foo() {
   static int count = 0;
   count = count + 1;
   printf("foo called %d times.\n", count);
}
```

Unsafe!

Thread safety (2)

- Serializability – thread-safe functions often behave "atomically" as if called in some serial order (e.g., behavior often found for printf())
- Restrictions on concurrency
  - Functions without access to global data or read-only access are trivially thread safe
  - Others need to be made thread safe by means of synchronization – at the expense of some concurrency
- Typical restrictions for thread-unsafe functions
  - Package-unsafe functions – requires locking on package level
  - Object-unsafe functions – requires locking on object level

Summary

- A process is a program in execution
- Threads are light-weight processes that can share code and a global address space
  - Advantages – responsiveness, utilization of multiprocessors
  - Disadvantages – synchronization overhead, programming complexity
- Mapping of user onto kernel threads
  - One to one, many to one, many to many
- Hardware threads can hide latency
- Hyperthreading can boost performance
- Thread safety through synchronization