User-Level Threads

- If we want to break a single process into multiple threads, one option is to use a user-level thread package
- This user-level thread system runs as a single process, and manages multiple threads for one application. This means:
  - All threads will get a fraction of the CPU time for a single process
  - E.g. the GNU Pth system
  - Advantage: A low cost of creation/management for threads since we do everything in user space
  - Disadvantage: if one thread blocks, all other threads block too

Kernel-Level Threads

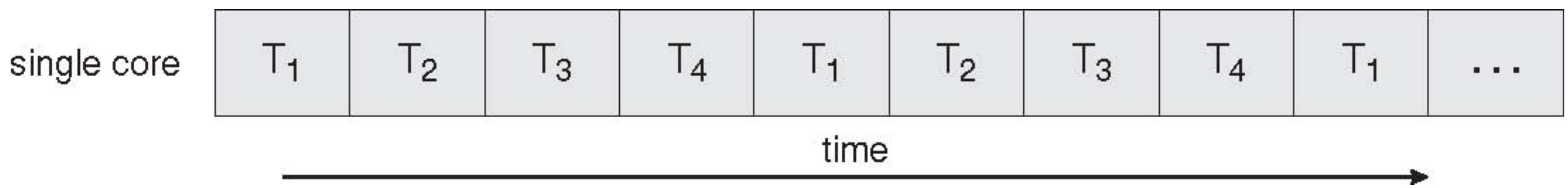
- The other option is to have threads managed by the kernel itself
- Advantage: one thread blocking will not affect the other threads for the process
- Disadvantage: threads have a high cost of creation/ management since we must make system calls

Multicore Programming

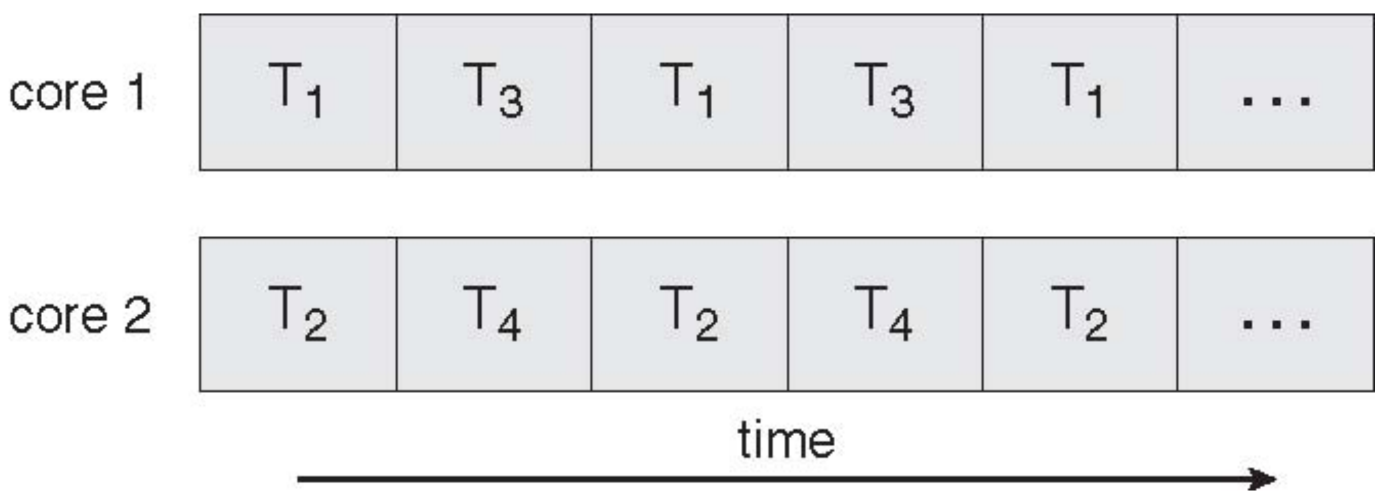
- The trend now is to place multiple processor cores on same physical chip → Faster and consume less power
- Multicore systems make having multi-threaded applications desirable, this puts pressure on programmers, challenges include:
  - **Dividing activities** – parallelism
  - **Balance** – need balance between threads bc if one thread does all work, others do none its bad
  - **Data splitting**
  - **Concurrency issues**
  - **Testing and debugging**

Concurrent Execution

Threads on a single-core system:



Threads on a multi-core system:



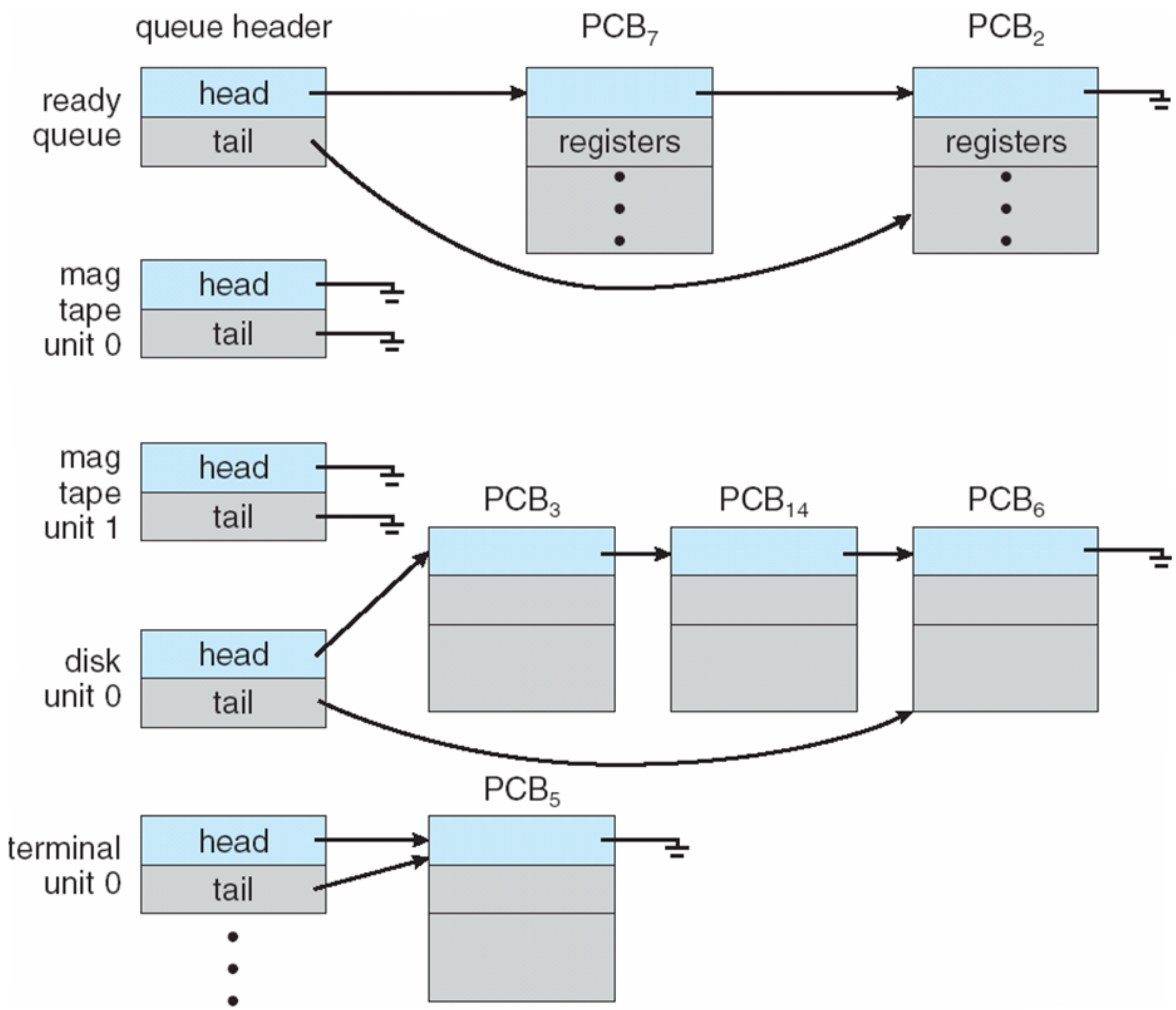
Threads, CPU Scheduling, and Multicore Processors

- Studies have shown that a thread may spend up to 50% of its time paused because of a *memory stall*
  - When a cache miss occurs and thread must wait for memory retrival from main memoory
- To reduce the effect of this, multiple threads are assigned to a single core and quick switches between threads occur whenever a memory stall happens
  - OS sees multiple logical processors for each core

Chapter 5: Back to Processes - Process States & Queues

- Recall the states that a process can be in during its lifetime: *running*, *ready*, *blocked*, and *deadlock*
- This leads to the idea of queues of processes in each state, e.g.
  - **Job queue** – set of all processes in the system
  - **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
  - **Device queues** – set of processes waiting for an I/O device
- Processes migrate among the various queues

Ready Queue & Device Queues



Process Scheduling

- Choosing which process gets the CPU next is the job of the CPU scheduler

- The goal of the scheduler is to maximize *throughput* and minimize *turnaround time:*
  - The interval between when the process enters the ready queue and when its next I/O burst starts

    CPU burst = executing instructions until reaching an I/O burst where it gets blocked

Types of Schedulers

- Two types of schedulers:
  - **Long-term scheduler** (or job scheduler) – selects which processes should be brought into the ready queue
  - **Short-term scheduler** (or CPU scheduler) – selects which process from ready queue should be executed next and allocates CPU
- Short-term scheduler is invoked very frequently (milliseconds) → must be fast
- Long-term scheduler is invoked very infrequently (seconds, minutes) → may be slow

Process Scheduling

- Three main issues in process scheduling:
  - *Decision Mode*: When (what time/how often) is a ready process selected to run?
  - *Priority Function*: Which process should run first?
  - *Arbitration Rule*: What if two processes have the same priority?

There are two possible *Decision Modes*:

1. Non-preemptive:
   - A running process is allowed to execute until its CPU burst ends
   - unsuitable for multi-user or real-time systems

Process Scheduling

2. Preemptive:
   - A running process can be stopped during a CPU burst to allow a process with higher or equal priority to run
   - When can a process be preempted?
     - When a blocked process is awakened
     - When a higher priority process arrives
     - When the process has executed for a certain amount of time (time quantum)

Process Scheduling

The *Priority Function* defines how priorities are assigned. Processes with higher priority are selected first for execution. Priorities can be based on:

- Memory requirements
- Time already taken/ time in system
- Expected time to complete
- User–assigned priorities

The *Arbitration Rule* breaks ties when there is more than one process with the same priority in the ready queue.

- E.g., Random, Round-robin, Chronological order