Principles of Concurrency and Parallelism

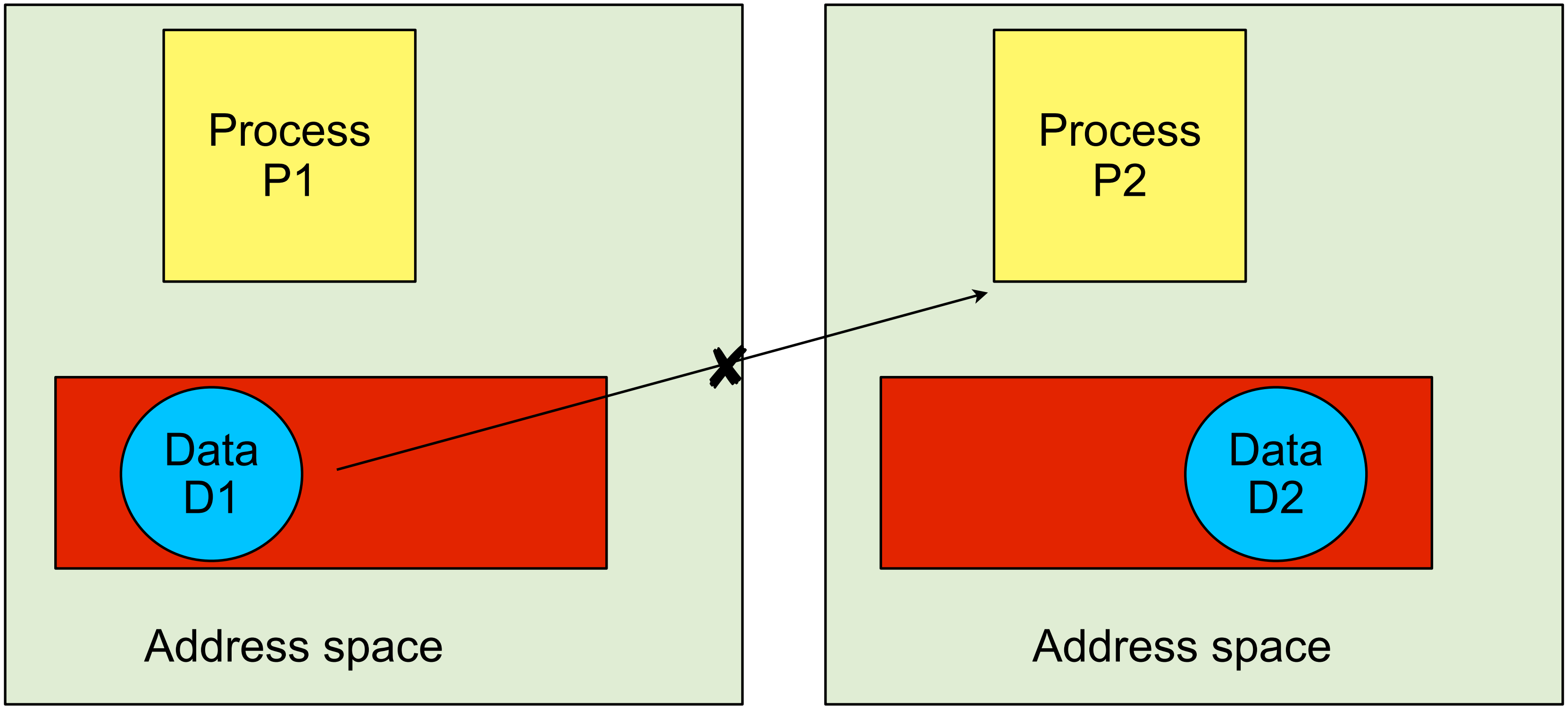
Lecture 3: Threads and Events

1/17/12

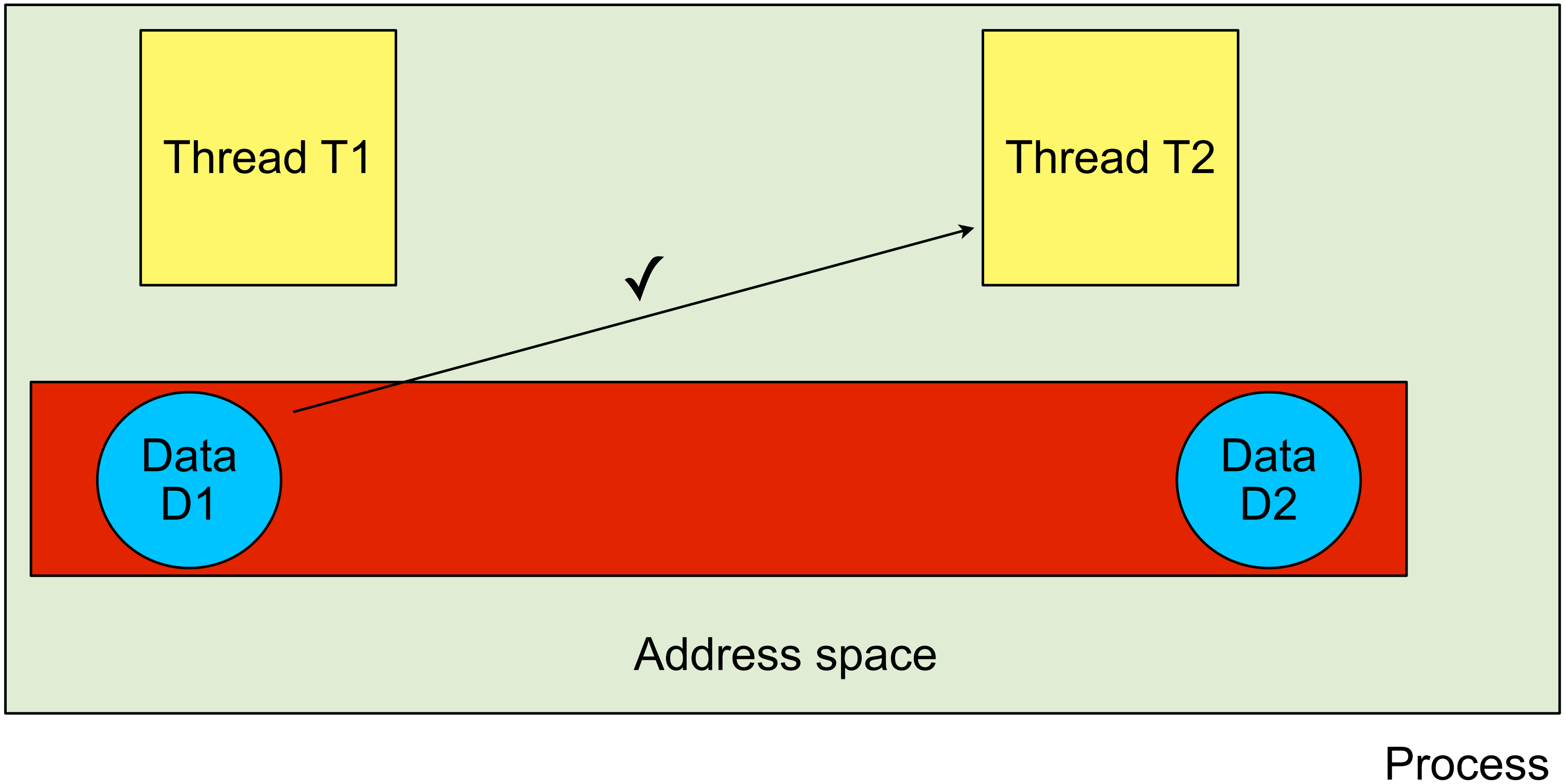
Threads and Processes

- A process is a representation of a computation managed by an operating system
  - Virtual address space
  - process control block
- A thread is a representation of a computation managed by an application
  - thread control block
- Process and thread control blocks contain all the information necessary to execute the computation (e.g., stacks, register contents, program memory, etc.)
- Main difference:
  - all threads within a computation execute within the same address space

Processes


Process
P1
Data
D1
Address space
Process
P2
Data
D2
Address space

Threads



Threads and Processes

- Critical distinction:
  - References (i.e., locations) have meaning between threads
  - They are interpreted independently between processes
    - Sharing state among processes requires special care
      - memory-mapped regions, devices, etc.

Threads

- The state (resources) needed to execute a thread is managed directly by a process
  - lightweight user-level threads
  - managed by an underlying runtime or virtual machine
- Kernel threads
  - typically user-level threads are multiplexed on top of kernel threads

Design Choices

- One process - One thread
- One process - Multiple threads
- Multiple processes - Multiple threads

Tradeoffs

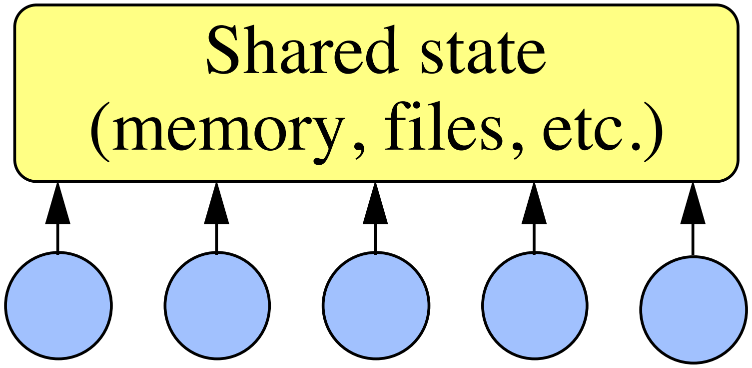
Cost of thread creation, management, and scheduling
Blocking and I/O
Application sensitivity

Coordination

- Synchronous
  - co-routines
  - cooperative
- Asynchronous
  - preemptive
  - callbacks
- Demand-driven
  - events

Threads

- An initial model



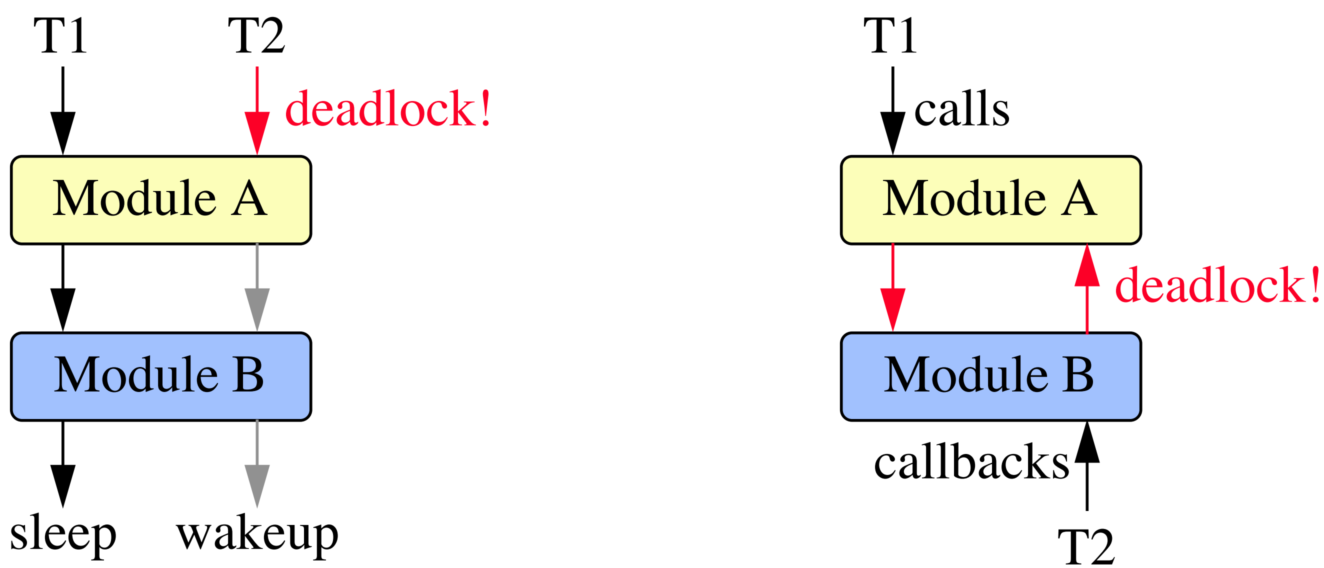
- Mediation among threads through explicit synchronization (locks, monitors,)
- Scheduling is asynchronous
  - Very flexible
  - But, raises lots of problems
    - Deadlock, livelock, fairness, etc.

Issues

- Synchronization
  - How should two threads communicate?
    - Use a lock
      - What happens if we forget, or we use the wrong lock?
        - Race conditions
      - What is the computation model we are trying to adhere to?
    - Message-passing
      - May need to greatly restructure existing sequential algorithms
  - Aggressive synchronization can lead to deadlock

Composability

- Threads that communicate using locks can easily break abstractions
  - Lower layers in the software stack may need to know behavioral properties of higher layers, and vise versa

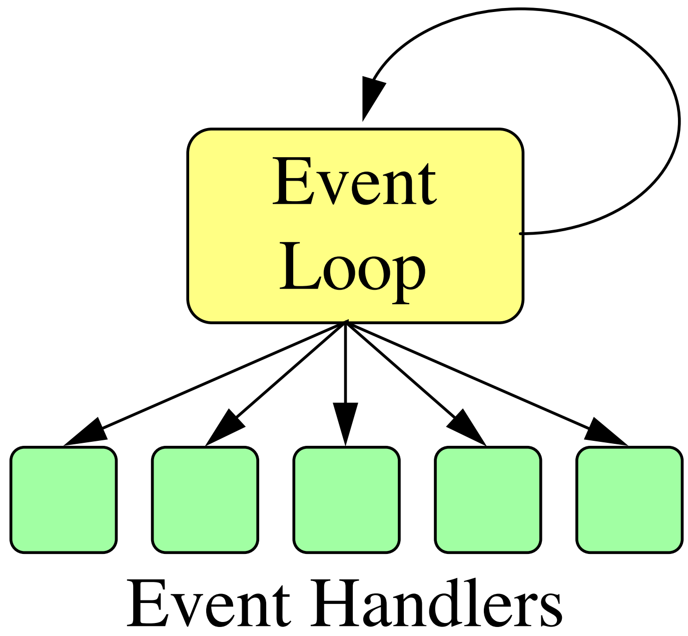


Performance and Correctness

- Even if there are no races, performance is an issue.
  - Too many locks: limits concurrency; too few: safety
  - Message-passing has similar overheads and safety issues
  - Inherently non-deterministic
- Performance at the expense of correctness
  - Many core applications not "thread-safe"
    - OS kernel calls, windowing toolkits, etc.
  - How do we migrate a sequential program to a concurrent one?
    - Identify places where concurrency is beneficial
    - Protect regions where concurrency may be harmful

Events

- Demand-driven strategy
  - Single execution stream, much like co-routines
  - Register interest in events
    - Wait for event to happen
    - Invoke handler when it does
  - No preemption
    - No locking necessary



Why is event-based programming useful?

- GUIs:
  - A handler for each interaction event (mouse click, drop-down action, etc.)
  - Handler implements dedicated behavior
- Distributed programming
  - One handler for each source of input
- Sometimes referred to a "specialist" concurrency

Issues

- Can’t have long-lived handlers
  - Composability
    - Suppose handler calls a function. How does the handler know how long the function will run? Suppose the function blocks?
- What about state?
  - No guarantee on consistency when handler resumes
  - “stack-ripping” (cooperative stack management)
    - continuations as callbacks
    - blocking I/O

Spectrum

- Event-based programming eschews concurrency
  - Easy to write, but hard to scale
    - No preemption, synchronization, deadlock
    - Simple control-flow
    - Debugging strategy similar to sequential programming
- Thread-based programming embraces concurrency
  - Harder to write, but easier to scale

Readings

- Why Threads are a Bad Idea (for most purposes), Ousterhout, 1996
- Why Events are a Bad Idea (for high-concurrency servers), von Behren et. al (2003)
- Cooperative Task Management without Manual Stack Management, Adya et. al (2002)