CS 351
Design of Large Programs
Concurrency

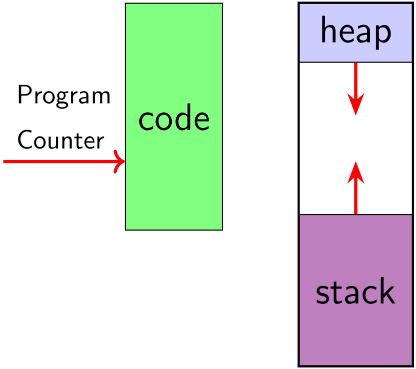
Brooke Chenoweth

University of New Mexico

Spring 2025

Sequential Process Characterization

- Program code (fixed)
- Control state (program counter)
- Memory state
  - stack
  - heap
- Formal properties
  - safety (does nothing wrong)
  - liveness (makes progress)

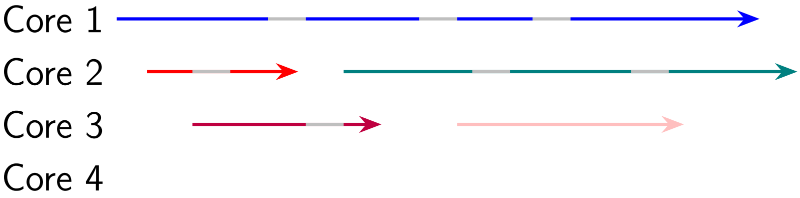


executing process

Physical Parallelism

Parallel execution of multiple independent processes takes place on separate physical hardware resources

- multiple cores
- specialized hardware interfaces
- parallel computers
- etc.



Logical Parallelism

- Interleaved execution of multiple independent processes takes place on a shared physical hardware resource (single CPU)
- Logical and physical parallelism coexist on modern computers
- Same two programs
  - may share a core at some point (interleaved execution)
  - may execute on separate cores at other times (parallel execution)

Process Scheduling

- It is the responsibility of the operating system to schedule the execution of the processes sharing one computing platform
- The scheduling policy significantly impacts the execution times of the individual processes
- Any attempt to perform a performance analysis needs to take the scheduling policy into account

Sample Scheduling Policies

- Fixed window
    - within a fixed-size window, each process has an assigned execution slot
- Round robin
    - each process gets a turn with no process being allowed to run forever
- Priority based
    - the process with the highest priority is scheduled first and runs to completion
    - the schedule may be preemptive or not

Concurrency

- *Concurrency* is an abstract unifying framework that enables one to reason about logical and physical parallelism
- It abstracts out
    - physical resources
    - timing considerations
- It achieves this by reducing all forms of parallelism to *nondeterministic execution of concurrent processes*
- It allows one to reason about the execution of concurrent processes while ignoring many of the complexities of the execution environment

Why Abstraction is Important

- Concurrent execution of multiple processes is an essential feature of modern computing
- Programming language development did not pay sufficient attention to concurrency, making programming more complex than it ought to be
- Some languages (including Java) include explicit constructs that address concurrent programming

Why Abstraction is Important

Concurrency introduces significant levels of complexity

- programs are rarely independent of each other
- programs need to coordinate with each other and compete for resources
- programs may need to coordinate even when
  - developed independently
  - residing on processors across the world

Fundamental Concepts

Atomicity

- An operation is atomic if it appears to be instantaneous and uninterruptable
- Programming languages provide only minimal atomicity guarantees
  - read a simple variable
  - write a simple variable
- This greatly complicates the programming task

Fundamental Concepts

Fairness

- Nondeterminism abstracts out the details of the scheduling policy
- Minimal guarantees are still needed in order to reason about process execution
  - *weak fairness* is a useful abstract concept, every program is eventually scheduled to execute
  - the operating system scheduling policy needs to be assessed when making such an assumption

Anomalies

Atomicity

- Let x=3 and y=5
- Consider the statement x := x + y
- What is the final value of x?

Anomalies

Fairness

- Assume a priority-based non-preemptive schedule
- Process P has the high priority 1
- Process Q has the low priority 2
- P is idle
- Q is busy (running)
- When will P run again?

Practical Concerns

In the absence of atomicity programming itself becomes impossible!

- Account balance \$245
- Teller 1: deposit \$100
    - read account balance
    - add \$100
    - update balance
- Teller 2: deposit \$300
    - read account balance
    - add \$300
    - update balance
- Account balance

Practical Concerns

In the absence of atomicity programming itself becomes impossible!

- Account balance \$245
- Teller 1: deposit \$100
  - read account balance (\$245) (1)
  - add \$100
  - update balance
- Teller 2: deposit \$300
  - read account balance
  - add \$300
  - update balance
- Account balance

Practical Concerns

In the absence of atomicity programming itself becomes impossible!

- Account balance \$245
- Teller 1: deposit \$100
  - read account balance (\$245) (1)
  - add \$100 (\$345) (2)
  - update balance
- Teller 2: deposit \$300
  - read account balance
  - add \$300
  - update balance
- Account balance

Practical Concerns

In the absence of atomicity programming itself becomes impossible!

- Account balance \$245
- Teller 1: deposit \$100
  - read account balance (\$245) (1)
  - add \$100 (\$345) (2)
  - update balance
- Teller 2: deposit \$300
  - read account balance (\$245) (3)
  - add \$300
  - update balance
- Account balance

Practical Concerns

In the absence of atomicity programming itself becomes impossible!

- Account balance \$245
- Teller 1: deposit \$100
  - read account balance (\$245) (1)
  - add \$100 (\$345) (2)
  - update balance
- Teller 2: deposit \$300
  - read account balance (\$245) (3)
  - add \$300 (\$545) (4)
  - update balance
- Account balance

Practical Concerns

In the absence of atomicity programming itself becomes impossible!

- Account balance \$245
- Teller 1: deposit \$100
    - read account balance (\$245) (1)
    - add \$100 (\$345) (2)
    - update balance
- Teller 2: deposit \$300
    - read account balance (\$245) (3)
    - add \$300 (\$545) (4)
    - update balance (\$545) (5)
- Account balance

Practical Concerns

In the absence of atomicity programming itself becomes impossible!

- Account balance \$245
- Teller 1: deposit \$100
  - read account balance (\$245) (1)
  - add \$100 (\$345) (2)
  - update balance (\$345) (6)
- Teller 2: deposit \$300
  - read account balance (\$245) (3)
  - add \$300 (\$545) (4)
  - update balance (\$545) (5)
- Account balance \$345 (WRONG!)

A Programming Language Solution

Critical Region

- a block of code that is executed atomically
- a way to ensure mutual exclusion

A Programming Language Solution

This time, each deposit is a critical region.

- Account balance $245
- Teller 1: deposit $100
    - read account balance
    - add $100
    - update balance
- Teller 2: deposit $300
    - read account balance
    - add $300
    - update balance
- Account balance

A Programming Language Solution

This time, each deposit is a critical region.

- Account balance \$245
- Teller 1: deposit \$100
    - read account balance (\$245) (1)
    - add \$100 (\$345)
    - update balance (\$345)
- Teller 2: deposit \$300
    - read account balance
    - add \$300
    - update balance
- Account balance

A Programming Language Solution

This time, each deposit is a critical region.

- Account balance $245
- Teller 1: deposit $100
  - read account balance ($245) (1)
  - add $100 ($345)
  - update balance ($345)
- Teller 2: deposit $300
  - read account balance ($345) (2)
  - add $300 ($645)
  - update balance ($645)
- Account balance $645 (CORRECT!)

Basics of Mutual Exclusion

- Test and set
- Locks
- Semaphores
- Mutual exclusion constructs (programming language specific)

Test and Set

Simple boolean flags cannot ensure mutual exclusion

- let G guard some resource that demands mutually exclusive access
- let G = true indicating that the resource is available
- processes P and Q need the resource

Test and Set

Simple boolean flags cannot ensure mutual exclusion

- let G guard some resource that demands mutually exclusive access
- let G $=$ true indicating that the resource is available
- processes P and Q need the resource
- P reads G to be true

Test and Set

Simple boolean flags cannot ensure mutual exclusion

- let G guard some resource that demands mutually exclusive access
- let $G = \text{true}$ indicating that the resource is available
- processes P and Q need the resource
- P reads G to be true
- Q reads G to be true

Test and Set

Simple boolean flags cannot ensure mutual exclusion

- let G guard some resource that demands mutually exclusive access
- let G = true indicating that the resource is available
- processes P and Q need the resource
- P reads G to be true
- Q reads G to be true
- P sets G to false
- P starts using the resource

Test and Set

Simple boolean flags cannot ensure mutual exclusion

- let G guard some resource that demands mutually exclusive access
- let G = true indicating that the resource is available
- processes P and Q need the resource
- P reads G to be true
- Q reads G to be true
- P sets G to false
- P starts using the resource
- Q sets G to false
- Q starts using the resource

Test and Set

- Hardware support is needed
- A process must test and set the flag in a single atomic step

```
while (true) do
  if G then G := false Must be atomic
    use resource
    G := true
    break
  fi
od
```

- The busy-wait is a real problem!

Locks

- Test and set enables the introduction of locks
- Associate a lock with each resource
- Bracket the use of the resource with the operations
  - lock(G)
    - returns only when the lock is set
    - the process is suspended avoiding busy-wait
  - unlock(G)

Locks

- A process may secure multiple resources as needed
  lock(file1)
  lock(file2)
    transfer data from file1 to file2
  unlock(file2)
  unlock(file1)
- Possible anomalies:
  - accessing the resource without locking
  - failing to issue the unlock
  - deadlock

Deadlock Avoidance

- Deadlock occurs when two processes are waiting on each other to release some resource
- One way of avoiding deadlock is for all the processes to lock resources in the same order

Semaphores

- A semaphore is a construct designed to allow at most k processes get access to a given resource
- When k is 1, the semaphore becomes a basic lock (a binary semaphore)
- Traditionally the two operations over a semaphore are
  - P(s) – tests for zero and decrements s by one, if greater than zero
  - V(s) – increments s by one indicating the release of the resource
- All processes must follow the same protocol
  P(s) — guards entry to the resource use of resource
  V(s) — frees the resource