PBA Software Development
Parallel Programming

Concurrency

Concepts, problems, and solutions

Units of execution

- Reasoning about concurrency and parallelism requires us to have terms to discuss that which is executing.

- Two common types of execution unit:
  - Processes
  - Threads

Processes

- Common OS abstraction. Represents a running program.
- Private address space relative to other processes.
- Contextual information such as register set, IO handles, etc…

Threads

- A process contains one or more threads.
- Threads have private register set and stack.
- Threads within a process share memory with each other.
  - Threads may also have private per-thread “thread local memory”.

Modern operating systems

- The line between threads and processes can be blurry.
  - Often both are implemented on top of the same OS abstraction, *kernel threads*.
- The critical distinction for many discussions is data sharing.
  - Processes do not share address space without explicit assistance.
  - Threads within a process share address space.

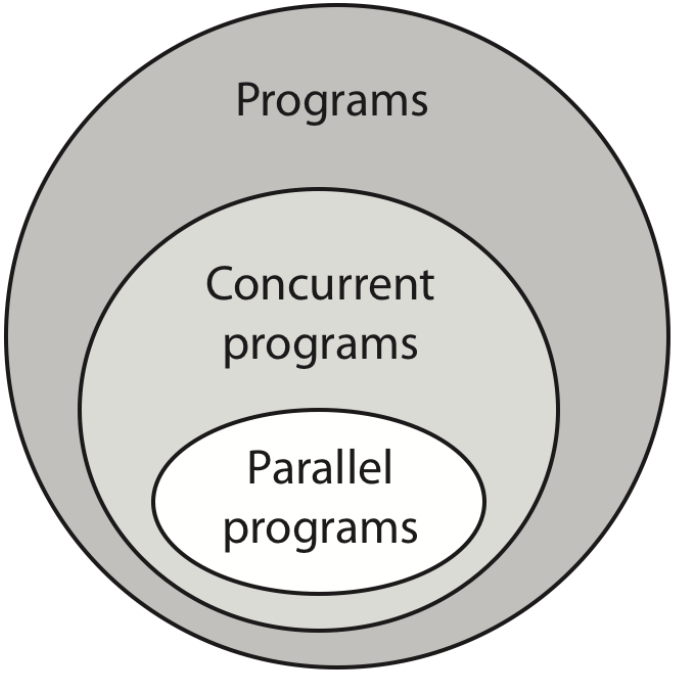
Terminology

- In the concurrent programming world, different projects use different terminology for execution units.
  - Threads
  - Tasks
  - Processes
  - Activities

- Generally a good idea to look up precisely what a term means in whatever context you encounter it in.

Parallel vs. concurrent

- What is the difference between a parallel and a concurrent program?

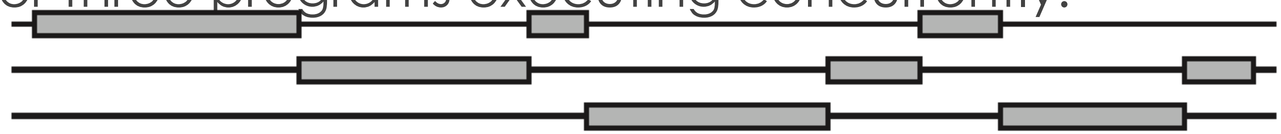


Parallel vs. concurrent

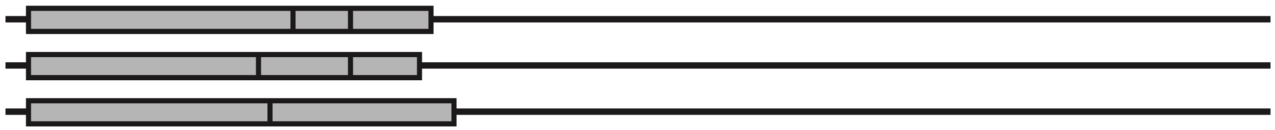
- Consider two units of execution that are started at the same time.
- If they run in **parallel**, then over any time interval during their execution, both may be executing at the same time.
- If they run **concurrently**, they *may* execute in parallel, or they may be sequentialized where only one makes progress at any given point in time and their execution is interleaved.
  - Multitasking operating systems have exploited this for many years.

Parallel vs. concurrent

- Consider three programs executing concurrently.



Concurrent, non-parallel execution



Concurrent, parallel execution

Shared vs. distributed memory

- Common distinction when considering parallel and networked computers.

- Consider a memory address space and a set of processing elements (CPUs or single CPU cores).

Shared memory

- Every processing unit can directly address every element of the address space.

- Examples include:
  - Multicore CPUs
    - Each core can see all of the RAM in the machine.
  - Shared memory parallel systems
    - Traditional parallel computers, such as servers.

Distributed memory

- Processing elements can directly address only a proper subset of the address space.
  - In order to access memory outside of this subset, processing elements must either:
    - Be assisted by another processing element that can directly address the memory.
    - Use specialized mechanisms (such as network hardware) to access the memory.

- We distinguish directly from indirectly accessible memory as ***local*** versus ***remote***.

Dependencies

- Dependencies are a critical property of any system that impacts concurrency.
  - Even impacts real-world activities like cooking.

- A dependency is a state (either of data or control) that must be reached before a part of a program can execute.

Simple example

- Both x and y must be defined before they can be read and used in the computation of z.
- Therefore z ***depends on*** the assignment statements that associate values with x and y.

```
x=6;
y=7;
z=x+y;
```

Dependencies limit parallelism

- If part of a program (a "subprogram") depends on another subprogram, then the dependent portion must wait until that which it depends on is complete.
  - The dependency dictates sequentialization of the subprograms.

- If no such dependency exists, the subprograms can execute in parallel ("embarrassingly" parallel).

Bernstein’s conditions

- We can formalize this via Bernstein’s conditions.
- Consider a subprogram P.
- Let IN(P) represent the set of memory locations (including registers) or variables that P uses as input by reading from them.
- Let OUT(P) represent a similar set of locations that P uses as output by writing to them.
- We can use these sets to determine if two subprograms P1 and P2 are dependent, and therefore whether or not they can execute in parallel.

Bernstein's conditions

- Given two subprograms P1 and P2, their execution in parallel is equivalent to their execution in sequence if the following conditions hold.

$$BC1 \quad : \quad OUT(P_1) \cap OUT(P_2) = \emptyset$$
$$BC2 \quad : \quad IN(P_1) \cap OUT(P_2) = \emptyset$$
$$BC3 \quad : \quad IN(P_2) \cap OUT(P_1) = \emptyset$$

Atomicity

- Atom derived from Greek word for *indivisible*.

- Often we write programs that include sequences of operations that we would like to behave as though they were a single indivisible operation.

  - Classic example:
    Two units of execution updating a shared variable

Intermediate results

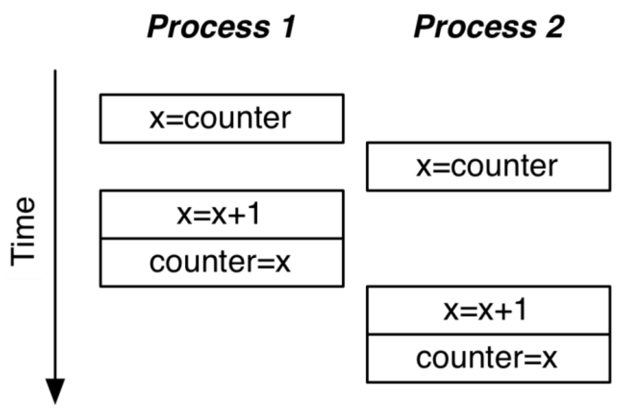
- Consider a shared counter.

```
shared int counter;
private int x;
x = counter;
x = x+1;
counter = x;
```

- We assume incrementing the counter is an atomic operation. If we do not, we could miss updates.

Missed update

- Poor interleaving of concurrent threads of execution can result in missed updates if atomicity isn't enforced.



Critical section

- Sequences of operations that can reveal invalid or intermediate data are referred to as *critical sections*.

- In the counter example, not only would this mean the increment operator would be a critical section, but…
  - Any other operation that modifies the counter would be.
  - As would any operation reading the counter.

Mutual exclusion

- Mutual exclusion refers to ensuring that one and only one thread of execution be in a critical section at any point in time.
- Concurrency control mechanisms can be used to implement this.

Thread safety

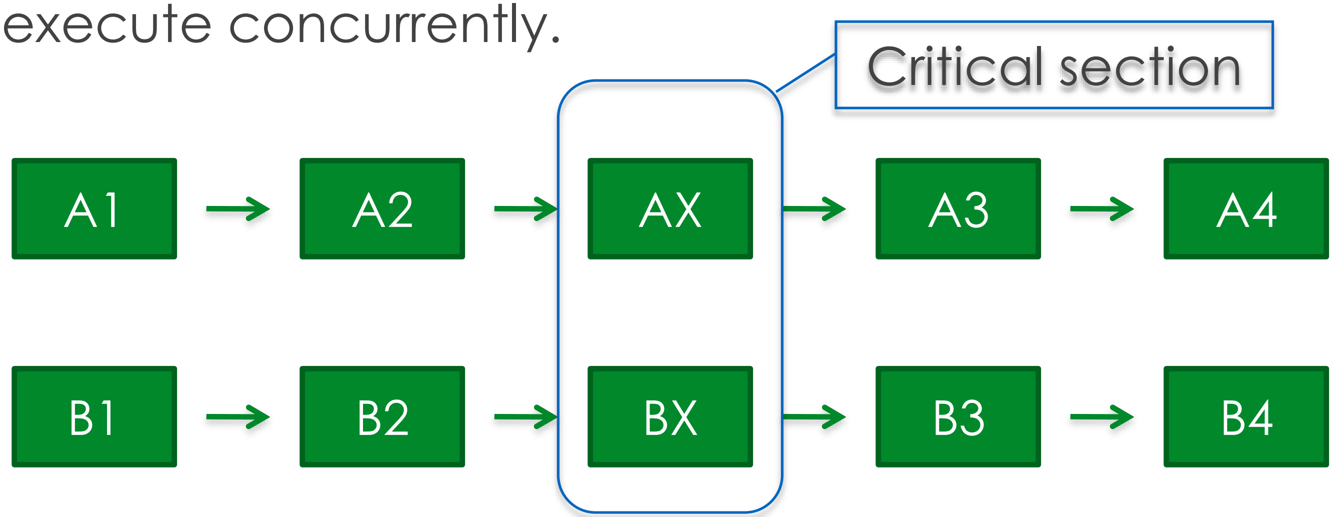
- A common term to encounter in programming is *thread safety*.
- Typically used to describe how a programmer can expect a subroutine, object, or data structure to behave in a concurrent context.
  - A thread safe implementation will behave as expected in the presence of multiple threads of execution, that is, the same behavior as if done sequentially.

Reentrant code

- A related term is *reentrant*.
- A reentrant routine (or region of code) may be entered by a thread of execution even if another thread has already started executing it at the same time.
- Non-reentrant code typically relies on state, such as global variables, that are not intended to be shared by multiple threads of execution.

Race conditions

- Consider two sequences of operations that can execute concurrently.



- A race condition exists if the result of the sequences depends on their relative arrival at some critical point in the sequence.
  - Indicated here as AX and BX.

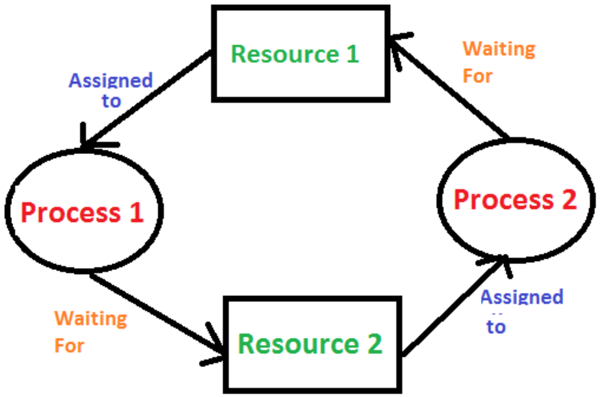
Race conditions

- If the result of operations after the critical point in the sequence (AX or BX) is dependent on which executes first (AX then BX versus BX then AX), a race is present in the code.
  - The result is dependent on which thread of execution "wins" the race to that point.

- Code without a race means that either:
  - The relative ordering doesn't matter.
  - Concurrency control constructs are used to enforce the required ordering.

Deadlock

- Deadlock results from a cyclical chain of dependencies.
  - A depends on B, and B depends on A.



- Often results from a misuse of concurrency control constructs.
  - A holds lock on resource that B wants, B holds lock on resource that A wants.
- Possible to observe when a program “freezes”, and simply ceases to make progress.

Livelock

- Similar to deadlock, but instead of freezing, a program enters into an endless cycle of operations that it cannot continue past.
  - Example: Acquire lock 1, attempt to acquire lock 2 and fail, release lock 1, retry.

- Livelock is detectable, but can be less obvious from simple observation than deadlock due to the fact that the program counter doesn't freeze.
  - It does enter a repeating cycle though.

Liveness

- If multiple threads of execution are started, we expect that they will all eventually finish.
- Often we will go further, and assume that during any reasonable interval of time (such as 1 second), all concurrent threads will make some progress.
- When a system or program ensures that this property holds, we say is ensures liveness.

Liveness

- Often liveness is a property of a scheduler, either within the OS, thread library, or user application.

- Concurrent programmers that manage their own threads must occasionally consider liveness of their management algorithms.

- When a system doesn’t ensure liveness, then starvation results.

Starvation

- A thread of execution will starve if it ceases to make progress in the presence of others.
  - Example: a program that is continuously preempted by others and never allowed to execute.
- Priority and aging schemes can be used to ensure that starvation does not occur due to a scheduler.
- Queuing disciplines (such as FIFO) can guarantee starvation will not occur due to synchronization primitives.

Nondeterminism

- Not always a problem – some algorithms can exploit nondeterminism to their advantage.
  - *Example*: concurrent queries to multiple search engines, return whichever result returns first.

- Nondeterminism is an inherent property of parallel systems, and must be kept in mind at all times.
  - Both for positive and negative reasons.

Control

- To prevent correctness problems, we must control how concurrently executing programs interact.
- Synchronization
  - Mutual exclusion
  - Semaphores, locks, monitors
- Transactions
  - Speculative execution and conflict resolution

Synchronization

- Synchronization allows concurrent threads of execution to communicate and coordinate information about their relative state.
- This can be used to coordinate their execution and prevent correctness problems.

Semaphores

- Semaphores are one of the original synchronization primitives due to Dijkstra.
- Represented as:
  - A single integer.
  - A queue of blocked accessors.
- Two operations are used to manipulate semaphores.

Semaphores

- Consider a semaphore **s**.
  - Initialized to 1 with an empty queue.
- **P(s)**:
  - Executed just before entering a critical section
  - If s>0 then decrement s,
    Otherwise, block the thread that attempted to perform P(s).
- **V(s)**:
  - Executed just before leaving a critical section
    If another thread is blocked on a call to P(s), unblock it.
    Otherwise, increment s.
- Increment/decrement apply **atomically** to the integer.
- Collection used to manage set of blocked threads.

Semaphores

- Binary Semaphore
  - Initial value is 1.
  - Allows one process in the critical section at a time (Mutual Exclusion).
- Counting Semaphore
  - Initial value is N.
  - The semaphore allows up to N processes in the critical section at the same time.
- Semaphores are a general structure.

Semaphores in .Net

- System.Threading.Semaphore
  - Counting semaphore
  - With no name: Local semaphore
  - With name: System semaphore (cross process).

  - Methods:
    - WaitOne: Blocking wait to enter Critical Section
    - Release: Release the Critical Section.

Semaphores in .Net

- System.Threading.SemaphoreSlim
  - Counting semaphore
  - Local semaphore

  - Methods:
    - **Wait**: Blocking wait to enter Critical Section
    - **WaitAsync**: Non-blocking wait for Critical Section. Returns a task containing Critical Section code.
    - **Release**: Release the Critical Section.

Locks

- A lock is a familiar synchronization construct ensuring mutual exclusion.
  - Consider locks on doors that have only one key.
  - Someone obtains key, is able to unlock (“acquire”) the lock.
  - Others must wait for the key to be released before they can unlock the lock themselves.
- Locks can be implemented using binary semaphores.

Locks in .Net

- lock(lockObject) { … }
  - Local mutual exclusion.
  - Any object may be used as "key".
  - Performance overhead to lock and unlock: 20 ns.

- Mutex
  - Cross-process mutual exclusion.
  - Performance overheds to lock and unlock: 1000 ns.
  - Methods:
    - **WaitOne**: Blocking wait to enter critical section.
    - **ReleaseMutex**: Releasing critical section.

Locks in .Net

- ReaderWriterLockSlim
  - Local Locking protocol.
  - Allows more readers simultaniously (shared lock) but only one writer at a time (exclusive lock)
  - Performance overheds to lock and unlock: 40 ns.
  - Methods:
    - **EnterReadLock**
    - **ExitReadLock**
    - **EnterWriteLock**
    - **ExitWriteLock**

Monitors

- Object-oriented approach to synchronization.
- Encapsulates the synchronization structures along with the ressource to be guarded, and makes the ressource **Thread-Safe**.
- Better than leaving the synchronization problem to the developers.

Monitors

- A binary semaphore is used to enforce mutual exclusion of the monitor, that is, only one process at a time will be using the monitor.

Monitors – Condition variable

- Important feature of monitors beyond encapsulation is introduction of *condition variables*.
- Condition variables allow threads to signal each other.
  - Beyond simple lock semantics.

Monitors

- Condition variable signaling
  - A thread may be executing within a monitor routine and reach a state where it must block and yield the monitor to allow another thread to access it so it can later pick up where it left off and continue.
  - Two operations are provided for condition variables.
    - Wait()
    - Signal()

Monitors

- A thread may *wait* on a condition variable.
  - Yield the synchronization structures for the monitor to allow other threads in.
  - Block waiting for another thread to signal it.
- Another thread may later *signal* threads blocked and waiting on a condition variable.
  - Signaled threads will wake up and attempt to reacquire the synchronization structures of the monitor.
  - When they succeed, they pick up immediately after the point where they *wait()*-ed.

Monitor in .Net

- System.Threading.Monitor
  - Used to ensure mutual access to a data ressource.
  - Performance overhead: 200 ns
  - Methods:
    - Enter: Blocking wait to enter the monitor.
    - Exit: releasing the monitor.
    - Wait(object): blocking wait for a condition object.
    - Pulse(object): Signalling release on a condition object.

Further reading

- http://www.albahari.com/threading/part2.aspx#_Locking