CS3211 — PARALLEL AND CONCURRENT PROGRAMMING

DR. CRISTINA CARBUNARU

arsatis[*]

CONTENTS

[*]author: https://github.com/arsatis

1 INTRODUCTION

Lecture 1
10th January 2023

Concurrency is different from **parallelism**:

Concurrency	Parallelism
Tasks execute in overlapping time periods (not necessarily at the same instant)	Tasks execute simultaneously (at the same time)

Generally, computers are able to run more than one task in parallel, with the number of **hardware threads** dictating the amount of achievable concurrency.

- Typically, each core has 2 hardware threads (some cores may have > 2).

1.1 *Processes & threads*

Processes	Threads
Better safety (i.e., when a process crashes, other processes can continue executing)	Better performance (i.e., lower overhead)
Greater overhead (of process creation and communication over OS)	Lower overhead (share the address space of the process, faster thread generation and communication)

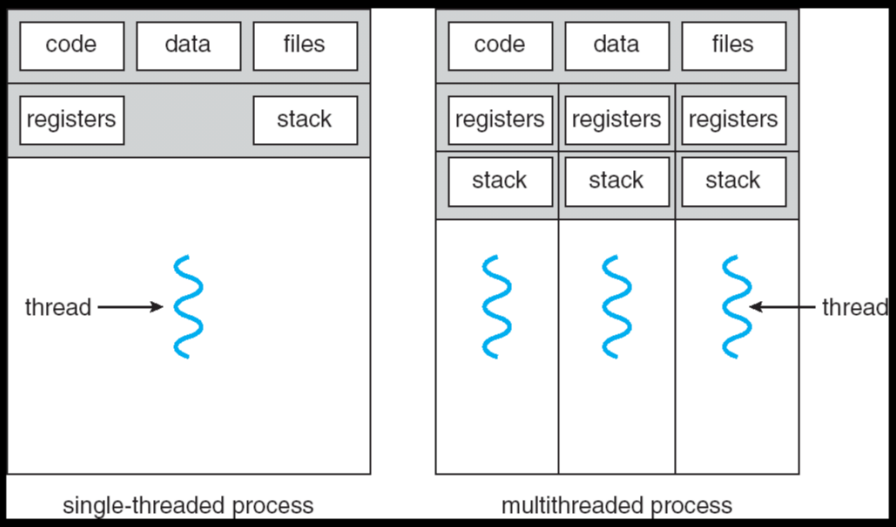


Figure 1: Differences between single-threaded vs multi-threaded processes.

1.1.1 Thread types

There are 2 types of threads:

User threads	Kernel threads
OS unaware of their existence	OS aware of their existence
No parallelism, since OS is unable to map different user threads of the same process to different resources	Efficient use of cores in a multi-core system
Scheduling performed at the process level → OS cannot switch to another thread if one thread is blocked (e.g., by I/O)	Blocking of one thread does not block other threads
More configurable and flexible (e.g., thread scheduling policies can be customized)	Generally less flexible

1.1.2 Process interaction with OS

The OS interacts with processes via two main ways: **exceptions** and **interrupts**.

Exceptions	Interrupts
Caused by *machine-level instructions*	Caused by *external events*, usually hardware related
Synchronous, occur due to program execution	Asynchronous, occur independently of program execution
Executes an *exception handler*	Executes an *interrupt handler*

1.2 *Synchronization*

A **race condition** occurs when:

1. two concurrent threads/processes **access a shared resource without any synchronization**, and
2. at least one thread/process **modifies the shared resource**.

To avoid race conditions and erroneous behaviour in parallel systems, we can utilize **mutual exclusion** to synchronize access to shared resources by implementing **critical sections (CS)** with the following properties:

1. **Mutual exclusion**:
    - If one thread is in the CS, then no other is.

2. **Progress**:
    - If some thread T_1 is not in the CS, then T_1 cannot prevent some other thread T_2 from entering the CS.
    - A thread in the CS will eventually leave it.
3. **Bounded wait** (no starvation):
    - If some thread T is waiting on the CS, it will eventually enter it.
4. **Performance**:
    - The overhead of entering and exiting the CS is small w.r.t. the work being done within it.

Critical sections are also expected to satisfy various requirements:

- **Safety property**: nothing bad happens.
- **Liveness property**: a system has to make progress.
- **Performance requirement**.

1.2.1 Mechanisms

- **Locks**:
    - Operations: `acquire(lock)` and `release(lock)`.
    - Locks can spin (i.e., spinlocks) or block (i.e., mutex).
- **Semaphores**: abstract data type that provides mutual exclusion through *atomic counters*.
    - Operations: `wait(S)` and `signal(S)`.
    - Safety property: its value is always ≥ 0.
    - Types: binary/mutex and general/counting.
    - However, they are essentially shared global variables and can potentially be accessed anywhere in the program.
- **Monitors**: allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become false.
- **Messages**: simple model of communication and synchronization based on atomic transfer of data across a channel.

1.2.2 Potential problems

- **Deadlock**: exists if every process in a set of processes is waiting for an event that can be caused only by another process in the set.
    - Exists iff the following conditions hold simultaneously:

1. **Mutual exclusion**: ≥ 1 resource must be held in a non-shareable mode.
2. **Hold and wait**: there must be one process holding one resource and waiting for another resource.
3. **No pre-emption**: resources cannot be pre-empted (i.e., critical sections cannot be aborted externally).
4. **Circular wait**: there must exist a set of processes $P_1, ..., P_n$ s.t. P_1 is waiting for P_2, and so on.

- **Starvation**: situation where a process is prevented from making progress because some other process has the resource it requires.
  - Typically a side effect of the scheduling algorithm.
- **Livelock**: two or more processes continually repeat the same interaction in response to changes in the other processes without doing any useful work.

1.3 *Benefits & drawbacks of concurrency*

There are several benefits and disadvantages to concurrency:

Benefits	Drawbacks
Separation of concerns	Concurrency issues
Increased performance	Maintenance difficulties
	Threading/Forking overhead

Other challenges involved in concurrent programming include:

- Finding sufficient avenues for parallelism/concurrency.
- Deciding on granularity of tasks.
- Coordination and synchronization.
- Locality considerations.
- Debugging.
- Performance modelling.

1.4 *Program execution in C++*

C++ programs go through the following steps during execution:

1. **Compilation** and **linking**:
   (a) The **preprocessor** replaces preprocessor directives (e.g., `#include` and `#define`) by including header files and expanding macros.

(b) The **compiler** parses C++ source code and converts it into assembly code.

(c) The **assembler** assembles assembly code into machine code (e.g., object code).

(d) The **linker** links the object files and libraries used in the original source code, and generates the final compilation output (i.e., an executable file).

2. **Loading**: the loader (an OS program) loads the executable from the disk into the primary memory (RAM) for execution; it allocates the memory space to the executable module in main memory and then transfers control to the first instruction of the program.
3. **Execution**: the executable is loaded from the disk to memory, and the computer's CPU executes the program one instruction at a time.

2 TASKS, THREADS, AND SYNCHRONIZATION

Lecture 2
17th January 2023

The execution of a C++ program consists of the following key steps:

1. **Decomposition** of the computations (to independent tasks).
2. **Assignment** of tasks to threads.
3. **Orchestration** of tasks, e.g.:
   - Structuring communication.
   - Adding synchronization to preserve dependencies.
   - Organizing data structures in memory.
   - Scheduling tasks.
4. **Mapping** of threads to physical cores.

This step is usually handled by the OS or threading library.

There are two ways to distribute work among threads:

1. **Task parallelism**: this can take place either through:
   - **Master-worker**: dividing the work into tasks, and making different threads specialize on different tasks.
   - **Pipelining**: dividing a sequence of tasks among threads, and each thread is responsible for a stage of the pipeline.
   - **Task pools**: a number of threads will be called to serve the task pool.

   Typically, some form of synchronization is required for task parallelism.
2. **Data parallelism**: generally, different threads will work on similar tasks in data parallelism, and there is little/no synchronization needed.

2.1 *C++ multithreading*

C++11 onwards has a **thread-aware memory model**, i.e., there is a standard which states how changes to a variable will be seen by other processes/threads.

- `std::thread`
  - `join()`: waits for a thread to finish.
    - Joins are needed even when there is an exception.
  - `detach()`: separates the thread of execution from the thread object, allowing execution to continue independently.
    - Note: local variables passed as parameters to the thread might end their lifetime before the thread terminates.
    - Use *explicit casts* instead of *implicit casts* when passing arguments to threads which will be detached.
  - To pass arguments by reference to threads, wrap them in `std::ref`.

Else, the program will terminate with an error.

Instances of `std::thread` are **movable** but not **copyable**.

movable: ownership of the thread instance can be transferred.
copyable: having multiple references to the same resource/thread.

2.2 *RAII*

An **owner** is an object, containing a pointer to an object allocated by `new`, for which a `delete` is required.

- Every object on the free store (i.e., heap, dynamic store) must have exactly one owner.

Resource management in C++ is based on *constructors* and *destructors*.

- For **scoped objects**, destruction is implicit upon scope exit.
- For objects placed in the **free store** using `new`, `delete` is required.

Resource Acquisition Is Initialization (RAII) is a C++ programming technique which binds the *life cycle of a resource* that must be acquired before use to the *lifetime* of an object.

- The lifetime of an object begins when:
  1. Storage (with the proper alignment and size) is obtained, and
  2. Its initialization (if any) is complete.
- The lifetime of an object ends when:

[non-class type] the object is destroyed, or
[class type] the destructor call starts, or
[any] the storage which the object occupies is released or reused by an object that is not nested within it.

i.e., the lifetime of an object is equal to/nested within the lifetime of its storage.

The lifetime of a *reference* begins when its initialization is complete, and ends as if it were a scalar object.

2.3 *C++ synchronization*

There are two main scenarios where synchronization is required:

1. Concurrent *access to shared data*:
   - There are no issues if shared data is **read-only**, since the data read by one thread is unaffected by another thread.
   - When updates are performed on the shared data, the **invariants** of data structures will be broken. We need to ensure that while the invariants are broken, the shared data is not accessed.
2. Concurrent *actions*: a thread may wait for another to complete a task.

invariants: statements which are always true about a particular data structure.

A correctly synchronized program should behave as if:

- Memory operations are actually executed in an order that appears equivalent to some sequentially consistent ordering.
- Each write appears to be atomic and globally visible simultaneously to all processors.

2.3.1 Concurrent access to shared data

Updates to shared data must be synchronized among threads using (either internal or external) locks to prevent:

- **Race conditions**: the timing or relative ordering of events affects a program's correctness.
  - The problematic execution sequence is more likely to occur when:
    - There is high load in the system.
    - The operation is performed more times.
- **Data races**: happens when there are 2 memory accesses in a program, where both:
  1. target the same location,
  2. are performed concurrently by two threads,
  3. are not both reads, and
  4. are not both synchronization operations (or atomic).

  Data races cause **undefined behaviour**.

Solution: wrap that the data structure with a protection mechanism (e.g., **mutex**), such that only the thread performing a modification can see the

intermediate states while the invariants are broken. The following classes in C++ serve as an improved version of `std::mutex`, locking the supplied mutex on construction and unlocking it on destruction.

- `std::lock_guard`
- `std::unique_lock`: doesn't need to own the mutex it is associated with.
- `std::lock`: locks one or more mutexes at once, without risk of deadlock.
- `std::scoped_lock`: accepts and locks a list of mutexes.

Notes:

- Typically, the mutex and protected data are grouped together in a class, in line with the principle of encapsulation in OOP.
- Pointers and references to protected data should not be passed beyond the scope of the lock, either via:
  - Returning them from a function, or
  - Storing them in externally visible memory, or
  - Passing them as arguments to user-supplied functions.

2.4 *Concurrent actions*

The naïve solution is to keep checking a shared flag within a while loop (i.e., **busy waiting**), which is wasteful. A better solution would be to use a **condition variable** to wait for an event to be triggered by another thread (without any busy waiting).

std::condition_variable (to be used with `std::unique_lock`) is:

- Associated with an event or condition, and
- ≥ 1 threads can wait (i.e., `wait()`) for that condition to be satisfied.

When the condition is satisfied, the thread can then notify (i.e., `notify_one()` or `notify_all()`) ≥ 1 of the threads waiting on the condition variable, which would wake up and continue processing.

Notes:

- **Spurious wake**: waiting thread reacquires the mutex and checks the condition, but not in a direct response from a notification.
  - To combat the issue of spurious wake, conditional variables are usually placed within a `while` loop with the same condition.
- There is no guarantee about how the conditional variable is implemented, thus spurious wakes should always be considered.

i.e., a thread may wake up seemingly randomly even though its condition is false.

3 ATOMICS & MEMORY MODEL IN C++

Lecture 3
31st January 2023

Prior to program execution, operations may be modified and reordered by the processor and compiler for optimization.

Optimizations by the processor include:

- W → W **reordering**: the processor might reorder write operations in a write buffer.
- R → W and R → R **reorderings**: the processor might reorder independent instructions in an instruction stream.

Additionally, C++ compilers are permitted to perform any changes to the program as long as the following remains true:

- At program termination, data written to files is exactly *as if the program was executed as written*.
- Prompting text which is sent to interactive devices will be shown before the program waits for input.

This is known as the **as-if rule**. Programs with undefined behaviour are free from the as-if rule.

3.1 *Memory models*

Memory models provide a contract to programmers about how their memory operations will be reordered by the compiler. C++ has a *multithreading-aware memory model* with the following features:

- **Structure**:
  - Every variable is an object (including members of objects).
  - Every object occupies at least one memory location.
  - Variables of fundamental types (e.g., `int`, `char`) occupy *exactly one* memory location, regardless of their size.
  - Adjacent **bit-fields** are part of the same memory location.
- **Concurrency**: if two accesses to a single memory location from separate threads
  - have no enforced ordering between accesses,
  - are not both atomic,
  - are not both reads,

  this results in **undefined behaviour**.

undefined behaviour: behaviour of the program is now undefined, and it may do anything at all.

3.2 *Modification order*

Modification order (MO) rep. the order of writes to a memory location, composing of all writes to an object from all threads in the program. Note:

- MO varies between runs.
- Every object in the program has a MO.
- If **atomic operations** are used, the compiler is responsible for ensuring that the necessary synchronization is in place.
- If the object is not atomic, the programmer is responsible for ensuring that the threads agree on the modification order of each variable (i.e., through synchronization).

Once a thread has seen a particular entry in the modification order,

- Subsequent *reads* from that thread must return later values, and
- Subsequent *writes* from that thread to that object must occur later in the modification order.

Different threads:

- Must agree on the MO of each object in a program, but
- Need not agree on the relative order of operations on separate objects.

3.3 *Atomic operations and types*

Atomic operations are indivisible operations which are always observed as either not done or fully done by any thread in the system. In contrast, non-atomic operations may be seen as half-done by another thread (e.g., value incremented in register but yet to write to memory).

Atomic types:

- Enable low-level synchronization.
- Are contained inside the `<atomic>` header.
- Whether the atomic operations are implemented using **atomic instructions** can be determined with the `is_lock_free()` member function (returns true if atomicity was not emulated using an internal mutex).

3.4 *Memory order*

Each operation on an atomic type has an optional **memory-ordering argument**, which specifies how memory accesses are to be ordered around an atomic operation.

The memory-ordering argument is a value of the `std::memory_order` enum.

The basic building blocks of operation ordering in a program include:

1. **Sequenced-before**: if A is sequenced before B, the evaluation of A will be complete before the evaluation of B begins.
2. **Synchronizes-with**: if a *suitably-tagged* atomic write (e.g., release) W in thread A on x synchronizes with a *suitably-tagged* atomic read (e.g., acquire) R in thread B on x, then R will read the value stored by either:
   (a) W, or
   (b) a subsequent atomic write on x by A, or
   (c) a sequence of atomic read-modify-write operations on x by any thread.
3. **Inter-thread happens-before**: A inter-thread happens-before B if any of the following holds:
   - A synchronizes-with B.
   - A synchronizes-with some X that is sequenced-before B.
   - A is sequenced-before some X that inter-thread happens-before B.
   - A inter-thread happens-before some X that inter-thread happens-before B.
4. **Happens-before**: A happens-before B if any of the following holds:
   - A is sequenced-before B.
   - A inter-thread happens-before B.

   Happens-before is a *transitive* relationship which specifies which operations see the effects of which other operations.
5. **Visible side-effect**: the side-effect W on a scalar x (e.g., write) is visible w.r.t. a read R on x if both of the following hold:
   (a) W happens-before R.
   (b) There is no other side effect Z on x where W happens-before Z and Z happens-before R.

   The **visible sequence of side-effects** of an operation X refers to the longest contiguous subset of side-effects in which the operation X does not happen before.
6. **Modification order**: see section 3.2.

There are several memory orderings, including:

- **Sequentially consistent** ordering: behaviour of the program is consistent with a sequential view of the world.
  - Arg: `memory_order_seq_cst`.
  - Features:
    * All threads must see the same order of operations.
    * Operations cannot be reordered.
    * Noticeable performance penalty on weakly-ordered machines (e.g., PC, TSO, PSO).

  In contrast, with non-sequentially consistent order,

  - All threads must also see the same order of operations **for each variable**, but
  - There is **no single global order of events** (i.e., different threads can see different views of the same operations).

Note: for all memory models (including relaxed consistency), all threads must agree on the same modification order for each atomic variable.

- **Relaxed** ordering.
  - Arg: `memory_order_relaxed`.
  - Features:
    * Operations on atomic types **do not participate** in *synchronizes-with* relationships.
    * Operations on the same variable **within a single thread** still obeys *happens-before* relationships.
    * Accesses to a single atomic variable from the same thread cannot be reordered.

However, operations on different variables can be reordered.
Also, there is no requirement on ordering relative to other threads.

- **Acquire-release** ordering: no total modification order, but introduces some synchronization.
  - Args: `memory_order_acquire`, `memory_order_release`, and `memory_order_acq_rel`.
  - Features:
    * If an atomic store in thread T_1 is tagged with release, and an atomic load in thread T_2 from the *same variable* is tagged with acquire, and the load in T_2 reads a value written by the store in T_1, then the store in T_1 *synchronizes-with* the load in T_2.
      · All memory writes (incl. non-atomic and relaxed atomic) that happened-before the atomic store in T_1 become *visible side-effects* in T_2.
      · The synchronization is established only between the threads releasing and acquiring the same atomic variable.

Most languages (e.g., C++11 onwards, Java) guarantee sequential consistency for data-race-free programs, if *every atomic operation uses* `memory_order_seq_cst`.

3.5 *Fences*

Lecture 4
9th February 2023

Fences are operations which enforce memory-ordering constraints without modifying any data, and are typically combined with atomic operations that use `memory_order_relaxed` ordering constraints.

- Specifically, an `atomic_thread_fence` with `memory_order_release` ordering prevents all preceding reads and writes from moving past all subsequent stores.

i.e., all preceding stores will be executed in the order which they were specified.

4 TESTING & DEBUGGING

We will focus on concurrency-related bugs, e.g.:

- **Unwanted blocking**:
  - **Deadlocks**.
  - **Livelocks**.
  - **Blocking on I/O** or other external input.
- **Race conditions**.
  - **Data races**: undefined behaviour due to unsynchronized concurrent access to a shared memory location.
  - **Broken invariants**, e.g.:
    - Dangling pointers: another thread deleted the data being accessed.
    - Random memory corruption: a thread reading inconsistent values resulting from partial updates.
    - Double-free.
  - **Lifetime issues**, e.g.:
    - Threads outliving the data they accessed.
    - Skipping `thread.join()` (esp. when exceptions are thrown).

Techniques for locating concurrency-related bugs include:

- Looking at the code.
- Testing.
  - However, precise scheduling of threads is indeterminate, thus:
    - the code does not always fail (*Heisenbug*).
    - it is difficult to reproduce problems.
  - Guidelines for testing:
    - Run the smallest amount of code that could potentially demonstrate a problem.

        * Eliminate the concurrency from the test to verify that the problem is concurrency-related.
        * Run the multithreaded code on a single core (to check for interleaving problems).
        * Test various scenarios.
        * Test various environments (e.g., processor architectures, number of threads, etc.)
    - To design for testability, ensure that
        * The responsibilities of each function and class are clear.
        * The functions are short and to the point.
        * Tests can take complete control of the environment surrounding the code being tested.
        * The code that performs the particular operation being tested is close together, rather than spread throughout the system.
        * The code can be broken down into:
            1. parts that operate on the communicate data within a single thread, and
            2. parts responsible for the communication paths between threads.
        * Concurrency-safe library calls are utilized.

Techniques for multithreaded testing include:

- Stress testing.
- Using special library implementations for synchronization primitives.
    - Mark shared data in some way, and allow the library to check that operations on a particular shared data are done with a particular mutex locked.
    - Record the sequence of locks if more than one mutex is held by a particular thread at once.
    - Give priorities to threads to acquire a resource.

Aside from testing for bugs, it is also important to test for performance, including:

- **Scalability**: the expected speedup is achieved when running with an increasing number of threads.
- **Contention** when accessing shared data from an increasing number of threads is minimized.

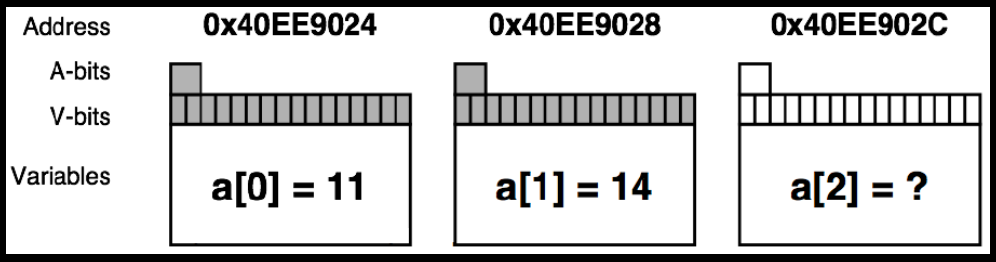
Some useful debugging tools include:

- **Valgrind**.
- **Sanitizers**.

4.1 *Valgrind*

Valgrind is a dynamic instrument (i.e., we can simply run the executable with it, without having to compile with it). It uses *shadow memory* to:

- Track and store information w.r.t. memory that is used by a program during its execution.
- Detect and report incorrect memory operations, e.g.:
  - Double-free of memory.
  - Accessing unallocated bytes.
  - Reading uninitialized bits.



- **A-bits** (valid address): the corresponding byte is *accessible*.
- **V-bits** (valid value): the corresponding bit is *initialized*.

However, it is expensive to run programs with Valgrind; there is an additional 10-20x overhead.

4.1.1 Helgrind

Helgrind in Valgrind intercepts calls to functions and instruments them. It is able to detect:

instrument: measuring of a product's performance, in order to diagnose errors and to write trace information.

- Misuses of pthread's API: by intercepting calls to pthread functions.
- Potential deadlocks arising from lock ordering problems: by building a directed graph indicating the order in which locks have been acquired.
  - When a thread acquires a new lock, the graph is updated, and then checked to see if it now contains a cycle.
  - The presence of a cycle indicates a potential deadlock involving the locks in the cycle.
- Data races: by checking the order in which memory accesses can happen.
  - Helgrind builds a directed acyclic graph representing the collective happens-before dependencies, and monitors all memory accesses.

However, there is a 100x overhead associated with Helgrind.

4.2 *Sanitizers*

Sanitizers adopt a compilation-based approach to detect issues.

- i.e., to use them, compile the program with `-fsanitize=[address|thread]`.

There is a smaller (5-10x) overhead associated with sanitizers.

- **Address sanitizer**: detects memory corruption bugs such as out-of-bounds accesses, use-after-free, double-free, and memory leaks.
- **Thread sanitizer**: detects a variety of concurrency bugs, including data races, potential deadlocks, leaked threads, etc.

  Tsan instruments a running program by intercepting and adding additional code to:

  – function entry and exit, and
  – memory accesses

  to identify problems.

In sum, a potential action plan for debugging involves:

1. Running Valgrind memory checks.
2. Running Tsan.
3. Running Asan.
4. Running Helgrind.

5 CONCURRENT DATA STRUCTURES

Lecture 5
14th February 2023

The goal of *thread-safe data structures* is to ensure that:

- Multiple threads can access the data structure concurrently (either performing the same or distinct operations), and
- Each thread sees a self-consistent view of the data structure.

We also need to ensure that:

- No data is lost or corrupted.
- All *invariants* are upheld (i.e., no thread can see a state where the invariants of the data structure have been broken by the actions of another thread), even in the presence of exceptions.
- There are no problematic race conditions.

While mutexes enable thread safety, they prevent true concurrent access of the protected data. Rather, **serialization** occurs; i.e., threads take turns to access the data protected by the mutex. To design for concurrency:

- **Minimize the amount of serialization** that must occur.
- **Lock at an appropriate granularity**: protect the smallest region of the code which we are trying to run concurrently.
- Allow **concurrent access for reads** and **exclusive access for writes**: using `std::shared_mutex`.
- Determine **which actions can be performed concurrently by other threads** while a thread is performing an action on the data structure.

5.1 *Lock-based concurrent data structures*

Most algorithms and data structures that use *mutexes*, *conditional variables*, and *futures* for synchronization are **blocking**. They invoke **blocking calls**, i.e., library functions which will suspend the execution of a thread until another thread performs an action.

Typically, the OS will suspend a blocked thread completely until it is unblocked by the appropriate action of another thread, e.g.:

- Unlocking a mutex, or
- Notifying a conditional variable, or
- Making a future ready.

When implementing lock-based data structures, ensure that:

- The correct mutex is locked when accessing the data.
- The lock is held for the minimum amount of time.
- The protected data cannot be accessed outside of the mutex.
- There are no inherent race conditions.

Typically, data structures provided by the STL library (e.g., stack, queue) are not thread-safe; manual implementation is required to utilize thread-safe data structures with fine-grained locks.

5.1.1 Lock-free concurrent data structures

Data structures and algorithms which do not use blocking library functions are said to be **nonblocking**. Types of nonblocking data structures include:

- **Obstruction-free**: all other threads are paused $\implies$ any given thread will complete its operation in a bounded number of steps.
- **Lock-free**: multiple threads are operating on a data structure $\implies$ one of them will complete its operation in a bounded number of steps.
  - Lock-free data structures allow ≥ 1 thread to access the data structure concurrently, irrespective of whether they are performing same or different operations.
  - If a thread accessing the data structure is suspended, the other threads must still be able to complete their operations without waiting for the suspended thread.
- **Wait-free**: all threads are operating on the same data structure will complete their operations in a bounded number of steps.
  - Does not contain algorithms which can involve an *unbounded number of retries* (e.g., using compare/exchange operations).

Properties of lock-free data structures:

(+) **Maximum concurrency**: some thread makes progress with every step.

(+) **Robustness**:
  - If a thread dies partway through an operation, nothing is lost except for that thread's data; other threads can proceed normally.
  - Threads cannot be excluded from accessing the data structure.

(−) **Livelocks** are possible.
  - E.g., t_1, t_2 try to change the data structure, but for each thread, the changes made by the other require the operation to be restarted, so both threads endlessly loop and try again.

(−) **Overall performance likely decreases**, even though the waiting time for each thread is reduced. This is because:
  - Atomic operations used by lock-free code can be much slower than non-atomic operations.
  - The hardware must synchronize data between threads that access the same atomic variables.
    - There may be significant overhead from **memory contention** and **write propagation**: if a thread modifies the shared data, this change has to be propagated to the cache on other cores. Depending on the memory ordering used, this modification may also cause the other cores to stop and wait for the change to propagate through the memory hardware.
    - **Cache ping-pong** arises due to multiple threads accessing the same atomic variables.
    - **False-sharing** (i.e., accessing data from the same cache line within multiple threads) can also lead to *cache ping-pong*.

Guidelines for writing lock-free code:

- Use `std::memory_order_seq_cst` for prototyping.
- Use a **lock-free memory reclamation scheme**:
  - Keep track of how many threads are accessing a particular object, and delete each object when it is no longer referenced anywhere.
  - Recycle nodes.
- Watch out for common problems, e.g.:
  - **ABA problem**: during synchronization, another thread can execute between two reads and change the value, do other work, then change the value back, thus fooling the first thread into thinking "nothing has changed" even though the second thread did work that violates that assumption.
  - **Use-after-free (UAF)**.
  - **Data races**: e.g., in the recycling stack, across two methods, etc.

6 CONCURRENCY IN GO

Lecture 6
28th February 2023

There are several differences between concurrent programming in C++ versus in Golang:

C++	Go
Model program in terms of *threads*	Model program in terms of *tasks*
Synchronize memory access between threads	Synchronize tasks by making them communicate
Thread pools can be used to limit the number of threads handled by the machine	

However, both C++ and Golang support various concurrent designs (i.e., task parallelism, data parallelism, and anything in between).

- **Task dependency graphs** can be used to visualize and evaluate the task decomposition strategy in task parallelism, where:
  - Nodes rep. tasks (node values rep. expected execution time).
  - Edges rep. control dependencies between tasks.
  - **Critical path** length: maximum completion time for all tasks.
  - **Degree of concurrency**: indication of amount of work that can be

done concurrently.

$$\deg = \frac{\text{total work}}{\text{critical path length}} = \frac{\sum_i v_i}{\sum_{i \in \text{critical path}} v_i}$$

- **Speedup**:

$$S_p(n) = \frac{T_{\text{best\_seq}}(n)}{T_p(n)} \tag{1}$$

  - $S_p(n)$ measures the benefit of parallelism.
  - Theoretically, $S_p(n) \leq p$ (< due to parallelization overheads).
  - In practice, $S_p(n) > p$ can occur, e.g., due to caching.

i.e., *superlinear speedup*.

- **Amdahl's law**: the speedup of parallel execution is limited by the fraction of the algorithm that cannot be parallelized (i.e., the *sequential fraction* $0 \leq f \leq 1$).
  - Amdahl's law, $S_p(n) = \frac{1}{f + \frac{1-f}{p}} \leq \frac{1}{f}$, holds only for *fixed problem sizes*.
  - For most computing problems, f is dependent on the problem size n, with $\lim_{n\to\infty} f(n) = 0$. Thus, $\lim_{n\to\infty} S_p(n) = \frac{p}{1+(p-1)f(n)} = p$.
- **Gustafson's law**: if the sequential fraction f decreases with problem size, then $S_p(n) = \frac{f(n)+p(1-f(n))}{f(n)+(1-f(n))} = p - (p-1)f(n) \leq p$.

6.1 *Concurrency & communication*

The Go model is based on the concept of **Communicating Sequential Processes (CSP)**.

"Processes" here actually refer to functions.

- **Concurrency**: structure a program by breaking it into pieces that can be executed independently.
  - **Goroutines** facilitate concurrency by enabling independent function executions.
- **Communication**: coordinate the independent executions.
  - **Channels** facilitate communication between goroutines.

CSP is a formal language for describing patterns of interaction in concurrent systems, and can be used to reason about program correctness.

6.2 *Goroutines*

Goroutines are functions which:

- Follow the fork-join model.
- Run independently of other goroutines.
- Share the same address space as other goroutines.
- Are cheaper than threads.
  - A newly minted goroutine is given a few kilobytes (which is typically enough). When the memory is insufficient, the runtime grows/shrinks the memory for storing the stack automatically.
  - The CPU overhead is about 3 cheap instructions per function call.
  - It is practical to create hundreds of thousands of goroutines in the same address space.
- Are not garbage collected.

They are a special class of concurrent subroutines (*coroutines*).

- When a goroutine blocks, only that thread blocks (other threads continue running).
- However, Go's runtime is able to suspend the blocking goroutine (i.e., **preemptable**).

The runtime multiplexes goroutines onto OS threads, thus decoupling concurrency from parallelism.

6.3 *Channels*

To synchronize access to shared memory locations, the `sync` package can be used (e.g., `sync.Mutex`, `sync.WaitGroup`, `sync.Once`, `sync.Pool`, etc.). However, it is used mostly in small scopes, such as a `struct`).

Instead, Golang encourages the usage of **channels** instead of modifying shared memory. On a high level, channels:

- Serve as a conduit for a stream of information (e.g., Unix pipes).
- Are references to a place in memory where the channel resides (i.e., they can be passed around the program).
- Take in values, and pass them downstream.
- Do not require callers to have knowledge about the other parts of the program that work with the channel.

Implementation-wise, channels:

1. Are typed (i.e., chan type).
2. Can be uni- or bi-directional.
3. Are blocking, i.e.:
   - Writes to a channel that is full block until the channel has been emptied.
   - Reads from a channel that is empty block until at least one item is placed in it.

Thus, deadlocks are possible.

Reading from a closed channel returns the default value of the channel.

Operation	Channel state	Result
Read	`nil`	Block
	Open and Not Empty	Value
	Open and Empty	Block
	Closed	<default value>, false
	Write Only	Compilation Error
Write	`nil`	Block
	Open and Full	Block
	Open and Not Full	Write Value
	Closed	**panic**
	Receive Only	Compilation Error
`close`	`nil`	**panic**
	Open and Not Empty	Closes Channel; reads succeed until channel is drained, then reads produce default value
	Open and Empty	Closes Channel; reads produces default value
	Closed	**panic**
	Receive Only	Compilation Error

Figure 2: Operations on channels and their corresponding results.

Channel ownership:

- The owner of a channel is the **goroutine** that instantiates, writes, and closes a channel.
- For unidirectional channels,
  - Owners have a *write-access* view into the channel (i.e., chan, chan<-).
  - Utilizers only have a *read-only* view into the channel (i.e., <-chan).

- Ownership helps to:
  - Remove the risk of *deadlocking* by writing to a `nil` channel.
  - Remove the risk of *panicking* by closing a `nil` channel, writing to a closed channel, or closing a channel more than once.
  - Prevent improper writes to the channel (i.e., by verifying that the type is correct).

The `select` statement binds channels together, and considers all channel reads/writes simultaneously to see if any of them are ready.

`time.After(...)` can be used with `select` statements to enforce timeout after a certain period of time.

- If none are ready, the entire `select` statement blocks (unless a `default` statement is included).
- If one of them is ready, the corresponding read/write will be executed.
- If multiple of them are ready, the Go runtime will perform a pseudorandom uniform selection over the set of ready cases/channels.

A `for-select` loop with a `default` statement can be used to allow a goroutine to make progress while waiting for another goroutine to report a result.

6.4 *Go memory model*

The **Go memory model** specifies the conditions under which reads of a variable in one goroutine can be guaranteed to observe values produced by writes to the same variable in a different goroutine.

- **Sequenced before**: within a single goroutine, reads and writes must behave as if they were executed in the order specified by the program.
  - The execution order observed by one goroutine may differ from the order perceived by another.
- **Synchronized before** is defined as follows:
  1. The go statement (which starts a new goroutine) is **synchronized before** the goroutine's execution begins.
  2. The exit of a goroutine is *not guaranteed to be synchronized before* any event in the program.
  3. A *send* on a channel is **synchronized before** the completion of the corresponding receive from that channel.
  4. A *receive* from an unbuffered channel is **synchronized before** the send on that channel completes.
     - In general, the k^{th} receive on a channel with capacity C is **synchronized before** the $(k + \text{C})^{\text{th}}$ send on that channel completes.
  5. The *closing* of a channel is **synchronized before** a receive that returns a zero value as a result of the channel being closed.

- **Happens before** is defined as the transitive closure of the union of the **sequenced before** and **synchronized before** relations.

7 CONCURRENCY PATTERNS IN GO

Lecture 7
7th March 2023

7.1 *Confinement*

Confinement allows us to achieve safe operations (e.g., reading, modifying) on a set of data. Confinement can be achieved in two ways:

- **Ad-hoc confinement**: data is modified from only one goroutine, even though it is accessible from multiple goroutines.
  - However, this is not recommended, since static analyses are needed to ensure safety.
- **Lexical confinement**: access to shared locations is restricted., e.g.:
  - Exposing only the read/write handle of the channel to different goroutines.
  - Exposing only a slice of the array.

7.2 *For-select loop*

For-select loops allow the sending of iteration variables out on a channel while waiting to be stopped (e.g., by a done channel).

```
func consumer(done chan struct{}, q chan int, sumCh chan int) {
    sum := 0
    for {
        select {
        case num := <-q:
            sum += num
        case <-done:
            sumCh <- sum
            return
        }
    }
}
```

7.3 *Preventing goroutine leaks*

Goroutines cost resources, thus is it important to ensure the termination of goroutines when they:

- have completed their work,
- cannot continue to work due to an unrecoverable error,
- are told to stop working,

in order to prevent memory leaks.

In general, the convention is to have the goroutine which has created another goroutine be responsible for stopping the new goroutine. This can be done by:

- Using a `done` channel (with the for-select pattern) to signal termination.
- Using a `sync.WaitGroup`, and waiting for all goroutines to finish execution before program termination.

7.4 *Error handling*

To gracefully handle erroneous states, we can have a goroutine in charge of maintaining complete information about the state of the program.

- All goroutines will send their errors to the state-goroutine, which can make an informed decision about what to do.
- When functions return, the potential result should be coupled with the potential error.

7.5 *Pipelining*

In a **pipeline**, a set of data processing elements are connected in series (i.e., stages), where the output of one element is the input of the next element.

In Go, pipelines can be constructed as a series of stages connected by channels, where each stage is a group of goroutines running the same function. In each stage, goroutines:

1. Receive values from upstream via inbound channels.
2. Perform some function on the data.
3. Send values downstream via outbound channels.

Each stage can have any number of inbound and outbound channels, except for the first (i.e., source/producer) and last (i.e., sink/consumer) stages.

Pipelining allows for the separation of concerns for each stage, such that:

- Stages can be modified independently of one another.
- Stages can be combined in different ways.
- Stages can be processed concurrently with upstream/downstream tasks.
- Portions of the pipeline can be fanned-out or rate-limited.

For best efficiency,

- Each stage should take roughly the same time to complete their tasks.
- Fan-out pattern can be used to decrease the processing time of a stage which is the bottleneck (e.g., I/O, data processing).

As compared to a task pool, pipelining is better if there is a cap on a specific resource that is needed by all tasks in the task pool (e.g., reading from/writing to a network).

7.6 *Fan-out & fan-in*

Fan-out and fan-in helps to resolve the issue where some stages in a pipeline might be slower than others, thus limiting the amount of concurrency in the program.

- **Fan-out**: start multiple goroutines to handle input from a stream.
  - We can fan-out stages of the processing if:
    - It does not rely on values that the stage has calculated before.
    - It takes a long time to run.
    - The order in which results arrive is unimportant.
      - There is no guarantee on the order that concurrent copies run, nor in what order they return.
  - `runtime.NumCPU()` can be used to find the number of OS threads that are used to run the goroutines.
    - In general, fan-out a maximum of `runtime.NumCPU()` goroutines, or profile the code to enhance performance.
- **Fan-in**: involves multiplexing/joining together multiple streams of data into a single stream.
  - This can be done by spinning up one goroutine for each incoming channel, and transferring the information from each stream into the multiplexed stream.
  - Consumers downstream will read from the multiplexed channel.

The example on using goroutines to implement a load balancer is not included in this summary.

7.7 Go runtime

Some naïve strategies for sharing goroutines between threads include:

- **Fair scheduling**: equally divide the number of tasks to the number of processors.
  - However, some tasks may take longer than others to execute, thus some OS threads will work more than others.
- **Centralized queue**: e.g., a task queue shared among all OS threads.
  - **Locality**: if a goroutine spins up another goroutine, it would be ideal for these goroutines will be executed on the same OS thread (to take advantage of locality).

The Go runtime allocates goroutines onto OS threads using a **work stealing** strategy.

1. At a **fork point**, add tasks to the tail of the deque associated with the thread.
2. If the **thread is idle**, steal work from the head of the deque associated with some other random thread.
3. At a **join point which cannot be realized yet** (i.e., the goroutine which it is synchronized with has not completed yet), pop work off the tail of the thread's own deque.
4. If the **thread's deque is empty**, either:
   - Stall at a join, or
   - Steal work from the head of a random thread's deque.

In Go, goroutines are **tasks**, and everything after a goroutine is a **continuation**. More specifically, work stealing involves stealing **continuations** instead of tasks from other threads.

8 CLASSICAL SYNCHRONIZATION PROBLEMS

Lecture 8
14th March 2023

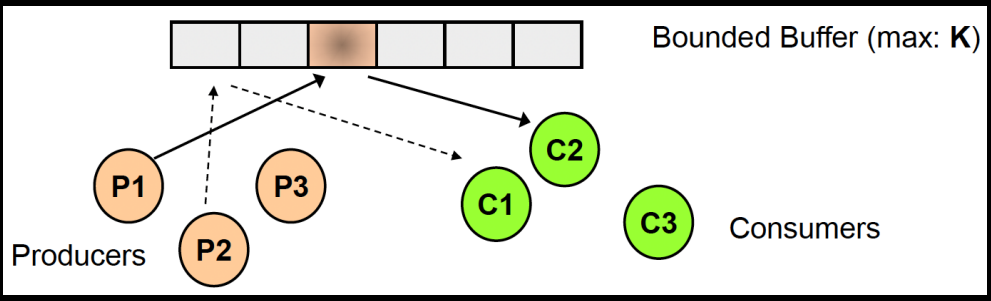
Classical synchronization problems model problems in computer systems.

Classical problem	Problem in computer systems
Producer-consumer	Interactions between processors and devices through FIFO channels.
Readers-writers	Access to shared memory.
Barrier	Waiting until processes/threads reach a specific point in the execution.
Dining philosophers	Allocation of limited resources to a group of processes in a deadlock-free and starvation-free manner.
Barbershop	Coordination of the execution of processor(s).
FIFO semaphore	Starvation avoidance (note: semaphores are not FIFO by default).
H_2O	Allocation of resources to processes.
Cigarette smokers	Allocation of resources by the OS to processes/applications.

8.1 *Producer-consumer*

In the producer-consumer problem,

- All processes (i.e., producers and consumers) share a buffer.
- Producers produce items and insert into the buffer when the buffer is not full.
- Consumers consume items from the buffer when the buffer is not empty.



Solution:

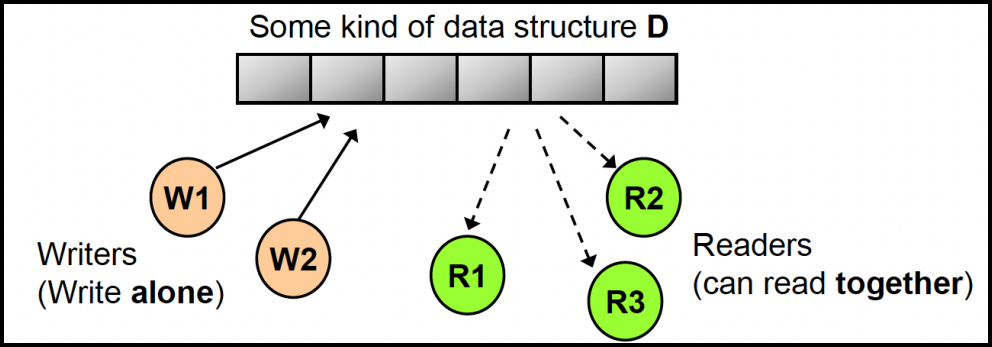
```
1: function Producer
2:     produce item
3:     wait(NotFull)
4:     wait(mutex)
5:     buffer[in] ← item
6:     in ← (in + 1) % k
7:     signal(mutex)
8:     signal(NotEmpty)
```

```
1: function Consumer
2:     wait(NotEmpty)
3:     wait(mutex)
4:     item ← buffer[out]
5:     out ← (out + 1) % k
6:     signal(mutex)
7:     signal(NotFull)
8:     consume item
```

8.2 *Readers-writers*

In the readers-writers problem,

- Processes share a data structure D.
- Readers retrieve information from D, and can read concurrently with other readers.
- Writes modify information in D, and must have exclusive access to D when writing.



Solutions:

- In C++, `std::shared_mutex` can be used to acquire different types of locks (i.e., exclusive or shared) on a data structure.
- In Golang, there is `sync.RWMutex` which serves the same purpose.
- Without libraries, the problem can also be solved with starvation by using a binary semaphore as a turnstile.

```
1: function Writer
2:     turnstile.wait()
3:     roomEmpty.wait()
4:     // enter CS
5:     turnstile.signal()
6:     roomEmpty.signal()
```

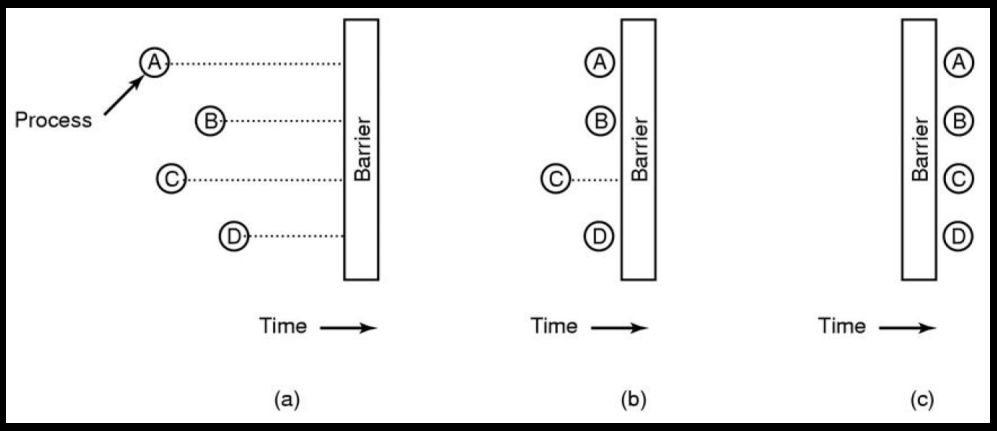
```
1: function Reader
2:     turnstile.wait()
3:     turnstile.signal()
4:     readSwitch.lock()
5:     // enter CS
6:     readSwitch.unlock()
```

8.3 *Barrier*

Barriers should be used sparingly to minimize unnecessary blocking.

Barriers are constructs where threads/processes are stopped until all threads/processes reach the barrier. There are two types of barriers:

- **Single use** barrier (a.k.a. **latch**): starts in the raised state, and cannot be re-raised once it is in the lowered state.
- **Reusable** barrier: once the arriving threads are unblocked from a barrier's phase synchronization point, the barrier can be reused.



Implementations:

- In C++, there is `std::latch` and `std::barrier`.
  - OpenMP also has a barrier construct, i.e., `#pragma omp barrier`.
  - MPI has barriers in **collective communication**, i.e., `MPI_Barrier`.
- In Golang, `sync.WaitGroup.wait()` functions as a barrier.

8.4 *Dining philosophers*

The dining philosophers problem models the problem of allocating limited resources to a group of processes in a deadlock-free and starvation-free manner.

Solutions:

- **Deadlock avoidance algorithms** for assigning mutexes: e.g., implementation of `std::scoped_lock` in C++.
- **Odd-even ring communication**: processes with odd and even ids take turns eating.
- **Limited eater solution**: let $k < n$ processes go first (use a **footman semaphore** to limit the number of eaters).
  - However, starvation is possible because mutexes are not FIFO/fair.

- **Chopstick-swapping**: using Golang's for-select pattern.

```
breakLock:
    for {
        <-evenChpStickCh
        select {
        case <-oddChpStickCh:
            break breakLock
        default:
            // Couldn't get oddChpStickCh, swap order of chpSticks
        }
        evenChpStickCh <- ChpStick{}

        <-oddChpStickCh
        select {
        case <-evenChpStickCh:
            break breakLock
        default:
            // Couldn't get evenChpStickCh, swap order of chpSticks
        }
        oddChpStickCh <- ChpStick{}
    }

    eat_callback(pid)

    evenChpStickCh <- ChpStick{}
    oddChpStickCh <- ChpStick{}
}
```

- **Tanenbaum solution**:

```
1:  function PHILOSOPHER(i)
2:      while true do
3:          think()
4:          TAKECHPSTICKS(i)
5:          eat()
6:          PUTCHPSTICKS(i)
7:
8:  function SAFETOEAT(i)
9:      if state[i] == HUNGRY && left, right processes are not eating then
10:         state[i] = EATING
11:         signal(s[i])
```

```
1:  function TAKECHPSTICKS(i)
2:      wait(mutex)
3:      state[i] = HUNGRY
4:      SAFETOEAT(i)
5:      signal(mutex)
6:      wait(s[i])
7:
8:  function PUTCHPSTICKS(i)
9:      wait(mutex)
10:     state[i] = THINKING
11:     SAFETOEAT(LEFT)
12:     SAFETOEAT(RIGHT)
13:     signal(mutex)
```

8.5 *Barbershop*

A barbershop consists of a room with n chairs and k barbers (typically, $k = 1$).

- If there are no customers, the barber goes to sleep.
- When a customer arrives,
  - If the barber is busy and there are chairs available, the customer sits on one of the free chairs. If the barber is asleep, the customer wakes up the barber.
  - If all the chairs are occupied, the customer leaves the shop.

Solution:

```
1:  function Customer
2:      wait(mutex)
3:      if customers == n then
4:          signal(mutex)
5:          exit()
6:      customers++
7:      signal(mutex)
8:      signal(customer)
9:      wait(barber)
10:     getHairCut()
11:     signal(customerDone)
12:     wait(barberDone)
13:     wait(mutex)
14:     customers–
15:     signal(mutex)
```

```
1:  function Barber
2:      while true do
3:          wait(customer)
4:          signal(barber)
5:          cutHair()
6:          wait(customerDone)
7:          signal(barberDone)
```

9 SAFETY IN RUST

Lecture 9
21st March 2023

Tools:

- **Crate**: a binary or library.
- **Package**: one or more crates that provide a set of functionality.
- **Modules**: used to organize code within a crate into groups.
- **Rustup**: for installing Rust tools.

Rust code does not compile if it is not safe.

- **Rustc**: Rust compiler.
- **Cargo**: compiles the package using a config file written in TOML format.

Rust features:

- **Strong (memory) safety guarantees**: expressive type system, ensures no segmentation faults and memory issues (e.g., data race, use after free, double free, dangling points, etc.).
  - **Aliased pointers**: pointers that point to the same chunk of memory.
  - **Mutation**: changing a pointer.
  - Most memory issues arise due to **aliasing + mutation**, i.e., modifying pointers that point to the same chunk of memory.
  - Rust eliminates such issues by introducing the concept of **ownership**.

Note: memory safety → thread safety.

- **No compromise on performance**: no garbage collector, zero-cost abstraction, lightweight iterators.

Zero-cost abstraction: Rust guarantees that its compiler generates code which is better than (or as good as) handwritten code.
E.g., for its iterators, Rust's compiler optimizes the cache locality which would be difficult to implement by hand.

9.1 *Ownership*

Ownership rules in Rust:

1. Each value in Rust has an owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

Ownership prevents aliasing.

- When a value is passed to a function:
  - Without `&`: ownership of the value is transferred to the function.
    - The runtime (shallow) copies over the fields from the caller's stack to the callee's stack, and discards the original values in the caller.
    - Deep copy of values can be done using `clone()`.
    - Transfer of ownership is enforced at compilation time.
  - With `&`: **shared borrow**, Rust creates a shared reference to the data.
    - Shared borrows allow for aliasing, but not mutation (i.e., data cannot be modified).
  - With `&mut`: **mutable borrow**, Rust creates a mutable reference to the data.
    - Mutable borrows allow for mutation, but not aliasing (i.e., only one function can mutably borrow a variable at one time).

  The borrow checker *statically* prevents aliasing + mutation.
- While there are no data races in Rust, race conditions are still possible.

9.2 *Concurrency*

Concurrency in Rust is library-based, and not built into the language. It leverages on the concepts of ownership and borrowing.

- **Thread creation**: `thread::spawn()`, returns a `JoinHandle` object.
  - When the `JoinHandle` object is dropped/freed, the spawned thread is detached.
  - To join a thread, call `joinhandle.join()`.
  - To obtain the return result of the thread (or a panic), call `unwrap()`.
- `move`: allows closures to take ownership of the values they use from the environment, *transferring ownership* of those values from one thread to another.
  - Without `move`, threads capture everything as borrows; whether the borrow is shared or mutable is inferred by Rust, depending on what happens in the execution of the thread.
- **Traits**: are interfaces (potentially with methods) that can be implemented for a given type. Some examples include:
  - **Send**: can be transferred across thread boundaries.
  - **Sync**: safe to share references between threads.
  - **Copy**: safe to memcpy (for built-in types).
- **Reference counting**:
  - `Rc<T>`: can be used when we want to allocate some data on the heap for multiple parts of our program to read, and we cannot determine at compile time which part will finish using the data last.
    - However, it is only for use in single-threaded scenarios, since it is not atomically managed (i.e., there is no **Send** trait).
  - `Arc::new()`: thread-safe reference counter, which allows only shared references to the variable it is holding onto.

Closures are anonymous functions that can be saved in a variable, passed as arguments to other functions, or returned from a function.

Putting data into an Arc does not make it thread-safe.

9.3 *Synchronization*

- **Mutex**:
  - `mutex.lock()` returns a `Guard<T>`, which is a proxy through which we can access the data.
  - Mutexes are often used together with `Arc` to be shared across threads (i.e., `Arc::new(Mutex::new(val))`).
- **Atomics**: Rust atomics are similar to C++ atomics, and they share the same memory model (e.g., SeqCst, Relaxed, Release/Acquire, etc.).
  - E.g., `AtomicUsize::new(x)`.

- **Channels**: Rust's channels are multi-producer, single-consumer FIFO queues.
  - `mpsc::channel()` returns a reading and writing reference.
  - The writing reference can be cloned to enable multiple threads to write to the channel, but the reading reference cannot be cloned.

9.4 *Useful libraries*

- **Crossbeam**:
  - Enables the creation of **scoped threads**.
    - This allows threads in different scopes to mutably borrow the same value.
    - They have since been added to Rust's `std` library.
    - E.g., `std::thread::scope(|scope| {...});`
  - Provides a multiple-producer multiple-consumer channel for message passing.
    - The channels can also be specified to be bounded or unbounded.
    - **Exponential backoff** can be used: if resources are not available right now, the program waits for a period of time before retrying to prevent the resource from being more overloaded (e.g., `backoff.snooze()`, `backoff.spin()`).
- **Rayon**: a data parallelism library with similar functionality as OpenMP.

10 ASYNCHRONOUS PROGRAMMING IN RUST

Lecture 10
28th March 2023

There are several overheads associated with the use of threads, including:

- **Context switching**:
  - Threads give up the rest of the CPU time slice when *blocking functions* are called, before the switch happens.
  - Many features are reset, e.g., registers are restored, virtual address space is switched, and cache gets stepped on, etc. → costly for high-performance situations (e.g., servers) involving multiple threads.
- **Memory overhead**:
  - Each thread has its own stack space that is managed by the OS.

Some solutions to this include:

- Using lightweight/green threads (e.g., goroutines).
  - However, a runtime process is needed to manage the threads (e.g., handling of thread context switching).

  - The runtime process itself introduces additional overhead.
- Using non-blocking functions (e.g., non-blocking I/O operations).
  - While the operation takes place, the thread can continue to perform some other operations, and come back later to check on the status of the I/O operation.
  - However, the context needs to be changed (which causes changes in the stack and cache), and this can be complicated to handle.

10.1 *Non-blocking I/O*

Generally, I/O functions (e.g., `read()`) will block if there is more data to be read, but no data is available. Instead of blocking, it would be better to have I/O functions return a special error value when no data is available.

- `epoll`: kernel-provided mechanism that notifies users of which file descriptors are ready for I/O.
  - When `epoll` is called, the thread will block and wait for something to happen in one (or more) of the file descriptors.
  - The `select` statement in Golang is implemented using `epoll`.
- This allows for the usage of **event loops**.
  - If the client has not sent anything yet, the thread can do other useful work (e.g., reading from other descriptors).
  - When an event happens on one of the file descriptors, the user is notified regarding which file descriptor has values to be read.

Thus, non-blocking I/O enables concurrency with a single thread. However, it is hard to manage the state associated with each conversation.

10.2 *State management*

In Rust, state management is done by using a state machine for each conversation to manage the state of each client.

10.2.1 Futures

Futures allow processes to keep track of in-progress operations along with their associated states.

- Formally, futures represent values that will exist sometime in the future.

- They are computations which have not happened yet, but are probably going to happen at some point in future.
- **Event loops** are needed to manage Futures.
  - They serve as user-space schedulers for Futures.
  - Within the loop, Futures are continuously polled until they are ready.

`futures.rs` is a crate that provides zero-cost abstraction for futures.

- The code in the binary has no allocation nor runtime overhead, as compared to handwritten code.
- Behind the scenes, asynchronous state machines are built.
- Rust's futures implement the `Future` trait.
- Futures can be combined with various **combinators**.
  - However, combinators can be messy as code complexity increases, and there is a need to share mutable data with functions which do not actually require them.
  - To combat this, Rust introduces `async` functions and `await` calls.

Combinators favor a functional (method chaining) style of code, e.g. `this.then(do_that()).and_then(do_that())`.

10.2.2 Executors

Executors loop over futures that can currently make progress.

- Executors can be either single or multi-threaded.
  1. To initiate a future, an executor thread calls `poll()` on it.
     - When `poll()` is called, a `Context` structure is passed in, which includes a `wake()` function.
       - `wake()` is called when the future can make progress again.
       - It is implemented using system calls.
  2. The future will then execute code until it can no longer progress.
     - If the future is complete, it returns `Poll::Ready(T)`.
     - Else if the future needs to wait for some event, it returns `Poll::Pending` and lets the thread work on another future.
  3. When no futures can make progress, the executor goes to sleep until one or more futures call `wake()`.
  4. After `wake()` is called, the executor can use `Context` to see which `Future` can be polled to make new progress.
- Futures should not block, or the executor running the code will sleep.
  - Asynchronous code should use non-blocking versions of objects.
  - Non-blocking implementations are provided by executor runtimes, e.g., `Tokio`.

Including mutexes, system calls, etc.

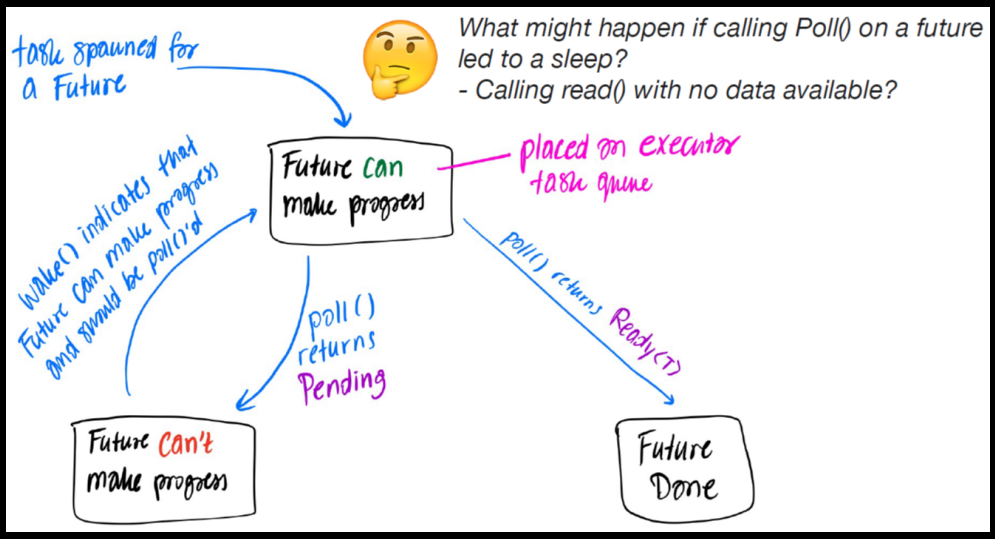


Figure 3: Workflow of an executor.

A popular executor in the Rust ecosystem is `Tokio`, which wraps around `mio.rs` and `futures.rs`.

- `mio.rs` is a fast, low-level I/O library for Rust focusing on non-blocking APIs and event notification for building high performance I/O apps with as little overhead as possible over the OS abstractions.
- `futures.rs` provides a number of core abstractions for writing asynchronous code.

Note: `Futures` do not actually do anything unless an executor executes them.

- With `Tokio`, the macro `#[tokio::main]` submits the future to the executor for execution.

10.2.3 Async & await

- An async function is a function that returns a `Future`.
  - Any `Futures` used in the function are automatically chained together by the compiler.
  - Rust's compiler transforms the code into a `Future` with a `poll()` method.
  - Note that async functions will not do any actual work when they are run. They serve to produce a future that does the stuff written inside the function.
- `.await()` waits for a future and gets its value.
  - Note that `await` can only be called in an async function or block.

Conceptually, async functions are implemented in the following manner:

- An enum is used to store the state of each possible location where we might get paused, e.g.:
  1. **Starting state**: before anything has happened, i.e., `Future` has been created but not yet `poll()`-ed.
  2. **Intermediate states**: awaiting for the execution of each step in the async function.
  3. **Terminal state**: `Future` has completed.
- When the user attempts to execute `poll()` on this `Future`,
  - Rust looks at the current state, and
  - Executes the appropriate code from the async function.

```
fn poll() {
    match self.state {
        NotYetStarted(email_id, recipient_id) => {
            let next_future = load_message(email_id);
            switch to WaitingLoadMessage state
        },
        WaitingLoadMessage(email_id, recipient_id, state) => {
            match state.poll() {
                Ready(message) => {
                    let next_future = get_recipient(recipient_id);
                    switch to WaitingGetRecipient state
                },
                Pending => return Pending,
            }
        },
        WaitingGetRecipient(message, recipient_id, state) => {
            match state.poll() {
                Ready(recipient) => {
                    recipient.verifyHasSpace(&message)?;
                    let next_future = recipient.addToInbox(message);
                    switch to WaitingAddToInbox state
                },
                Pending => return Pending,
            }
        },
        ...
```

```
enum AddToInboxState {
NotYetStarted { email_id: u64, recipient_id: u64 },
WaitingLoadMessage {
    recipient_id: u64, state: LoadMessageFuture },
WaitingGetRecipient {
    message: Message, state: GetRecipientFuture },
WaitingAddToInbox {
    state: AddToInboxFuture },
Completed { result: Result<(), Error> },
}
```

```
async fn addToInbox(email_id: u64, recipient_id: u64)
    -> Result<(), Error>
{
    let message = loadMessage(email_id).await?;
    let recipient = get_recipient(recipient_id).await?;
    recipient.verifyHasSpace(&message)?;
    recipient.addToInbox(message).await
}
```

Figure 4: Illustration of how async functions work.

Implications:

- Async functions are *stackless coroutines*.
  - They do not have a stack; all task states are self-contained within the generated `Future`.
  - The executor thread still has a stack, but
    - It is used to run normal/synchronous functions.
    - It is not used to store state when switching between async tasks.
- No recursion is allowed (yet).
  - This ensures that the `Future` returned by async functions have a fixed size, known at compile time.
  - Otherwise, the size of the enum (as discussed previously) will have to grow dynamically during runtime.

- Async functions in Rust are nearly optimal in terms of memory usage and allocation; there is very low overhead.

However, usage of async code only makes sense when:

- There is a need for an extremely high degree of concurrency.
- Work is primarily I/O bound (context switching is expensive if we are only using a tiny fraction of each time slice).

10.3 *Other languages*

- **Javascript**: similar toolbox with `Promises` and `async/await`.
  - However, it is not as efficient as Rust because more dynamic memory allocation is involved.
  - Javascript executors are also single-thread; there is no parallelism even on multicore architectures.
- **Golang**: goroutines are asynchronous tasks, but they are not stackless.
  - The stacks are **resizable**, but that is possible because Golang is garbage-collected.
  - The runtime knows where all pointers are, and can reallocate memory.
- **C++20**: stackless coroutines are available.
  - However, they are newly implemented and are difficult to use due to a lack of library support.

* * *