Week 1 - Introduction

Read, understand, and summarize the following computing paradigms: concurrent computing, parallel computing, distributed computing, cluster computing, grid computing, cloud computing, fog/edge computing, etc. (any more related you find?)

Concurrent computing

Concurrent computing is a paradigm in which tasks run in overlapping time periods instead of one after another (sequentially). With this paradigm, the tasks seem simultaneous to the user, though in reality they may not run in parallel.

In the past when computers were limited to single-core processors that could only do one thing at a time, concurrency was achieved with time-sharing, a method whereby tasks take turns to execute in small slices of time. This creates the illusion that these tasks are running at the same time, but in fact the processor is just alternating between them. Time-sharing is still used to this day for single-core processors; however, nowadays most computers have multi-core processors that enable tasks to run in true parallel fashion.

Concurrency allows today's computers to do many things at the same time. For example, a personal computer can be used to read emails, play music, and browse the Internet all at once. Even seemingly single tasks such as viewing a website can be broken down into multiple sub-tasks such as fetching data, drawing the UI, and responding to user interactions. Without concurrency, computers would be very limited.

Parallel computing

Parallel computing is a concept very similar to concurrent computing; however, in this paradigm, tasks are indeed running at the same time with no illusions of simultaneity to the user. With parallel computing, tasks are broken into independent subtasks that can run on different cores of the same processor or on different processors. Parallelism cannot be achieved on a single-core processor as it can only do one thing at a time. In this case, tasks can run concurrently with time-sharing as described above. However, since most computers nowadays have multi-core processors, true parallelism is possible.

Parallel computing finds application in many areas such as machine learning or computer graphics. These areas often make use of the parallelism of GPUs to perform many complex matrix computations at the same time.

Distributed computing

This paradigm involves multiple networked computers working together to solve a problem. While this may sound similar to parallel computing, there are two key distinctions. First, whereas parallel computing can be achieved with either a single computer with a multi-core processor or multiple computers, distributed computing always involves several computers on a network. Second, unlike parallel computing, computers in distributed computing do not have a shared memory and instead communicate through message passing, which can incur some network overhead.

One use case of distributed computing is in the energy industry. Energy companies can use distributed computers to process the data streams coming from dispersed sensors in order to improve operations.

Cluster computing

With cluster computing, many low-cost computers (called nodes) are grouped together to form a single high-power system. This is similar to distributed computing in the sense that it involves multiple networked computers. However, communication within a cluster is generally much faster than within a distributed system since the nodes are situated closer together and talk through a local area network.

One application of cluster computing is big data analysis. The combined computational power of the low-cost nodes can create a “supercomputer” that processes data quickly and efficiently.

Grid computing

Grid computing involves a network of geographically distributed computers working together to achieve a goal. Whereas nodes in cluster computing are generally homogenous and contribute computing power towards the same task, nodes in grid computing are more heterogeneous and perform different tasks. Grid computing also differs from cluster computing in terms of the level of coupling and proximity between nodes.

This paradigm can be used to solve problems that require community effort such as physics and biology research. However, it is becoming increasingly obsolete with the rise of cloud computing.

Cloud computing

Cloud computing refers to the on-demand delivery of computer resources such as data storage and computing power over the Internet (the cloud) without requiring the user to directly manage them on their local computers. The benefit of this paradigm is that the user can use computing resources directly without having to worry about low-level implementation details. However, the drawback is that they do not have control over the underlying platforms. Nowadays, cloud computing is offered as services by companies like Amazon or Microsoft and as such, its resources are centralized.

Cloud offerings from companies fall into one of three service models: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS).

Nowadays, cloud computing finds many applications such as web hosting and deployment (Firebase, AWS Amplify), file storage (Google Drive, Dropbox), software as a service (Google Docs, Zoom, etc.), and more.

Edge/fog computing

Edge/fog computing is a computing paradigm that places computation and data storage closer to the data sources. It aims to leverage the computational power of edge devices and reduce latency compared to sending data to and processing it on a centralized cloud server.

Edge/fog computing is most commonly used in Internet of Things (IoT) applications which can involve smart devices and sensors that generate very large amounts of data. Bringing the computation closer to these devices can make better use of their computational power, improve bandwidth, and lower latency.

Week 2 - Process

Read, understand and summarize “process”.

A **process** is an instance of a program that has been loaded into memory and can be executed by the CPU. It is different from a **program**, which is a static set of instructions stored in disk. By definition, a program can make many processes. For example, Chrome is a program, and if one opens Chrome in multiple windows, each window corresponds to a different Chrome process.

The concept of processes came into existence as computers evolved to run multiple programs at the same time. When a program is run, a process is created for it. A process is made up of the program image, a block of isolated memory, the CPU state, and the operating system state.

- The **program image** is the compiled, executable instructions of the program.
- The isolated memory block is called the **address space** and cannot be accessed by other processes. It is divided into a stack, a heap, a data segment, and a program (or text) segment.

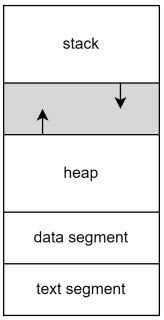

Fig. 1: The address space divided into stack, heap, data, and text segments.

- The **stack** stores automatically allocated variables (typically of primitives such as integers or booleans) and function return addresses. The stack grows downwards as in the diagram and if it exceeds a given limit or intersects with the heap, the result is a stack overflow error.
- The **heap** stores dynamically allocated variables, which typically are objects whose lifetime you want to control and whose size is unknown at compile time. Functions can refer to these objects with pointers that are stored in the stack. As per the diagram, the heap starts on top of the data segment and grows upwards.
- The **data segment** stores the program's global and static variables.
- The **program (or text) segment** stores the executable instructions of the program. This region is readable but not writable. The program counter iterates over the instructions in this segment.

A process can be in one of several states at a given moment, which are:

- **Created**: the process has just been created and is waiting to be marked "ready".
- **Running**: the process is having its instructions executed by the CPU.
- **Ready**: the process is awaiting execution because another process is running, etc.
- **Blocked**: the process cannot proceed without some external event happening (such as the user typing in some input).
- **Terminated**: the process has finished all its instructions or has been killed explicitly. It is not immediately removed from the process table but stays as a **zombie process** until its parent process reads its exit status with the wait system call.

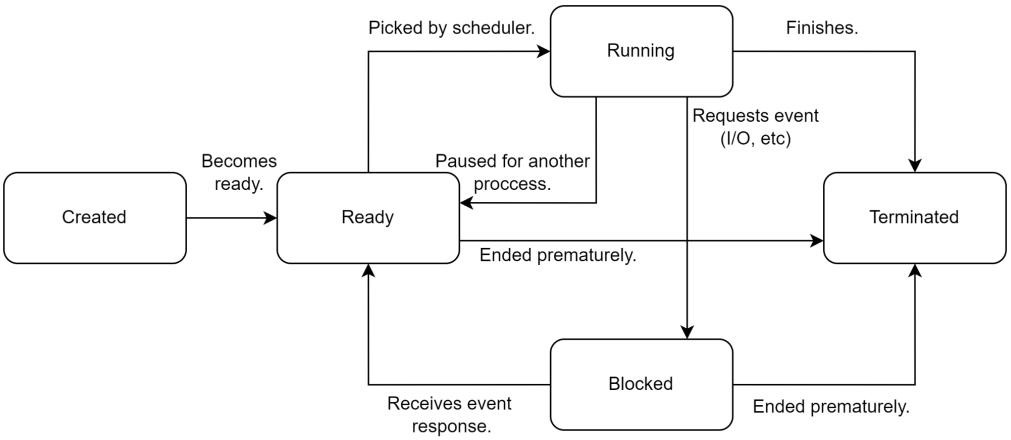

Fig. 2: The relationship between the different process states.

POSIX-compliant operating systems such as MacOS or most Linux distributions provide an API that lets you work with processes in C. Three key functions in this API are `fork`, `wait`, and `exec`.

Fork

`fork()` clones the current process into another process with its own memory. The new process starts executing from the instruction immediately after the `fork` call.

`fork` returns a number representing the status of the fork.

- -1 means an error happened while creating the child.
- 0 means that the child process has been created successfully and that this process is the child
- Any value greater than 0 means that the child process has been created successfully and that this process is the parent.

Refer to the code snipped in `wait` for an example of how to use `fork`.

Wait

`wait()` is used by the parent process to wait for its child to finish or die. During this time, the parent process is paused. It takes as parameter a pointer to an integer and will write the exit code of the child process into this variable when the child finishes. `wait` comes with two flavors: `wait()`, which waits for any child to finish, and `waitpid(int pid)`, which waits for the process with the specified process ID (pid) to finish.

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        int exitStatus;

        printf("I am the parent process and my process ID (pid) is %d\n", getpid());

        int forkedPid = fork();

        if (forkedPid == -1) {
            printf("An error happened while creating the child process. My pid is %d\n",
getpid());
            return 1;
        } else if (forkedPid == 0) {
            printf("I am the child process and my pid is %d\n", getpid());
            printf("Doing some work in the child process...\n");
            return 0;
        } else {
            printf("I am the parent process. Doing some work here...\n");

            wait(&exitStatus);

            printf("Child process exited with code %d\n", exitStatus);
            printf("This will only be run after the child process is finished because we have
waited for it.\n");
            return 0;
        }
}
```

In the code above, the parent process prints the first line with its process ID, then calls `fork` to create a child process. If the fork fails, the code inside the first `if` block executes, notifying the user of an error. If the fork succeeds, a child process is created and immediately executes the line after fork, which is the `if` statement. Because `fork` returns 0 for the child process, the second `if` block is executed. In the

meantime, the parent process gets a `forkedPid > 0` and executes the code inside the third `if` block. After printing the first line, it will pause and wait for the child process to finish. When it does, the exit status is recorded into an integer. Afterwards, the parent process executes the last two print lines and returns a status code of 0, ending the program.

The output looks like this:

```
I am the parent process and my process ID (pid) is 36062
I am the parent process. Doing some work here...
I am the child process and my pid is 36063
Doing some work in the child process...
Child process exited with code 0
This will only be run after the child process is finished because we have waited for it.
```

Exec

`exec()` causes the process to execute another program, replacing any instructions after the `exec` call with those of the called program. In more technical terms, `exec` replaces this process' image with that of the invoked program. It comes in several flavors such as `execl`, `execle`, etc. and is often used as part of the fork-exec-wait pattern.

```c
#include <stdio.h>
#include <sys/wait.h>
#include <unistd.h>

int main(int argc, char* argv[])
{
        int exitStatus;

        printf("I am the parent process and my process ID (pid) is %d\n", getpid());

        int forkedPid = fork();

        if (forkedPid == -1) {
            printf("An error happened while creating the child process. My pid is %d\n",
getpid());
            return 1;
        } else if (forkedPid == 0) {
            printf("I am the child process and my pid is %d\n", getpid());
            printf("I am about to execute another program. Anything I want to do in this
process should be done before I call exec.\n");

            // Executes the terminal command: /bin/ls .
            execl("/bin/ls", ".", ((char*)0));
            printf("If I somehow fail to call the other program, my image won't be replaced
and this line will be printed.\n");
            return 1;
        }

        printf("From this point on, only the parent process can run this code because the
child process cannot reach this.\n");
        printf("Because the child terminates if it fails to call the other program, and
overwrites these instructions otherwise.\n");

        wait(&exitStatus);

        printf("Child process exited with code %d\n", exitStatus);
```

```
        printf("This will only be run after the child process is finished because we have
waited for it.\n");
        return 0;
}
```

In the code above, the parent process prints the first line with its process ID, then calls `fork` to create a child process. If this fails, the `if` block is run and the process returns 1. Otherwise, a child process is created and immediately executes the `else if` block. It prints some lines to the terminal, then `exec` (here `execl`, whose parameters are the shell program to be called and its arguments) calls the terminal command `/bin/ls .`, which prints the contents of the current directory in the terminal. If the `exec` call somehow fails, the print after `exec` is called and the child process terminates with status 1. All of the code after the `else if` block is accessible only by the parent process, because the child process either fails and returns early, or succeeds and `exec` overwrites all later instructions. Similar to the previous example, the parent waits for the child to finish before printing the last 2 lines and returning. The output looks like this:

```
I am the parent process and my process ID (pid) is 37095
From this point on, only the parent process can run this code because the child process cannot
reach this.
Because the child terminates if it fails to call the other program, and overwrites these
instructions otherwise.
I am the child process and my pid is 37096
I am about to execute another program. Anything I want to do in this process should be done
before I call exec.
01forker        02forkexec      03execlooper    a.out  exec.c  fork.c
01forker.c  02forkexec.c  03execlooper.c  exec        fork
Child process exited with code 0
This will only be run after the child process is finished because we have waited for it.
```

Week 3 - Scheduling

Read, understand, and summarize the following scheduling approaches: first come, first serve (FIFO); shortest job first (SJF); preemptive shortest job first/shortest remaining time/shortest time-to-completion first (PSJF/STCF), round robin, lottery scheduling, multilevel feedback queue. Discuss more models from your research (Optional).

CPU scheduling

CPU scheduling decides which process should run at a given point in time and for how long according to some strategy (or algorithm). Scheduling is necessary because the CPU (or each of its cores) can only execute one process at a time and therefore has to allocate processing time fairly and efficiently among processes.

Several scheduling strategies exist. They differ in how they select processes to run and how much time they allocate to them. Many strategies select processes to run based on their priorities. **Static scheduling** assigns fixed priorities to processes when they are first submitted while **dynamic scheduling** can adjust priorities during runtime.

Some strategies can pause the currently running process in favor of another, higher-priority process. These are called **preemptive scheduling**.

Some scheduling strategies such as SJF may suffer from **resource starvation**, a situation where some processes do not get any execution time.

A few time measurements are used in the context of CPU scheduling, which are:

- **Processing time** (or **burst time**): The amount of time needed to fully execute the process.
- **Waiting time**: The amount of time the process stays in the Ready state.
- **Turnaround time**: The amount of time between process submission and process completion. It is the sum of processing time and waiting time.
- **Response time**: The amount of time it takes for a process to get a response from the CPU.

To switch between processes or threads, the CPU has to maintain the state of the paused process/thread so that it can be restored for later execution. This is called a **context switch**. Different scheduling strategies incur different context-switching costs.

Scheduling strategies

First Come, First Serve (FCFS)

Executes processes in the order in which they arrive. The current process is run to completion before the next one begins.

- Non-preemptive scheduling.
- Does not assign priorities to processes.
- Advantages:
    - Easy to understand and implement (FIFO queue) and works similarly to a real-world queue.
- Disadvantages:
    - Results in long turnaround time for processes later in the queue, especially if the previous processes have long processing times. Due to this, it is not used in practice.

Shortest Job First (SJF)

Executes the process with the shortest processing time first until it either completes or enters the blocked state before moving on to the next. Similar to FCFS except the queue is ordered by processing time instead of arrival time.

- Non-preemptive, static scheduling.
- Advantages:
    - Shorter average waiting time compared to other strategies.
    - Processes with short processing times get completed more quickly.
- Disadvantages:
    - Resource starvation. Processes with long processing times may not get any CPU time if there are too many short processes coming.
    - Accurately calculating the processing time of a process is a challenge. Thus, it is only used in specialized situations.

Shortest Remaining Time (SRT)

Executes the process with the shortest remaining time first. Also called Preemptive Shortest Job First (PSJF) or Shortest Time-to-Completion First scheduling.

- Preemptive, dynamic scheduling.
- Advantages:
    - Shorter average waiting time compared to other strategies.
    - Short processes are executed quickly because their remaining time is low.
- Disadvantages:
    - Resource starvation similar to SJF.
    - Accurately calculating the remaining time of a process is a challenge. Thus, it is only used in specialized situations.

Round Robin (RR)

Assigns a time slice (or time quantum) to processes in rotation. Processes can only run when they get a time quantum and must pause and wait if they do not finish within the prescribed quantum. If a process terminates or switches to the Waiting state before its time quantum elapses, the next process gets to run.

- Preemptive, starvation-free.
- Advantages:
    - Starvation-free.
    - Fair because each process gets an equal slice of the CPU.
    - Faster response time than SJF and SRT because processes do not have to wait for others to finish before getting the CPU's attention.
- Disadvantages:
    - High turnaround time because processes have to wait for their turn to execute.
    - Selecting the optimal duration of the time quantum is challenging. If it is too small, response time improves but there is a higher context-switching overhead. If it is too large, response time is slow, especially if there are many processes.

Multilevel Feedback Queue (MLFQ)

Places processes in queues of different priority levels and selects them based on the following five rules:

> **Rule 1**: If process A is in a higher-priority queue than B, A gets to run before B.
>
> **Rule 2**: If process A is in the same priority queue as B, A and B are selected using round robin.
>
> **Rule 3**: When a process is first submitted, it is placed at the end of the highest-priority queue.
>
> **Rule 4**: When a process uses up its time quantum in its current queue, it is preempted and moved to the end of the next lower-priority queue.
>
> **Rule 5**: After some time period, all processes are moved to the highest-priority queue. This is done to resolve potential starvation for processes at the lowest-priority queue.

This strategy can be configured with a number of parameters:

1. The number of queues.
2. The length of the time quantum for each queue. Typically, higher-priority queues get shorter time quanta while lower-priority queues get longer quanta.
3. The frequency at which processes are boosted to the top.

Characteristics:

- Preemptive, dynamic scheduling.
- Starvation-free.
- Advantages:
    - Starvation-free.
    - Approximates SJF as initially processes are considered to be short. Long processes will eventually be moved to the lowest-priority queue.
    - Favors processes with short CPU bursts and high IO bursts.
    - Does not require previous knowledge of processing times.
    - Adopted by several operating systems such as BSD Unix, Solaris, and Windows Series.
- Disadvantages:
    - Selecting the right parameters can be challenging.
    - High overhead switching processes between queues.

Lottery Scheduling

Randomly assigns each process a number of lottery tickets. To select the next process, draw a random ticket and the process holding that ticket gets to run.

- Can be preemptive or non-preemptive.
- Probabilistic.
- Starvation-free.
- Advantages:
    - Starvation-free because each process gets a non-zero chance of being run at any given time.
- Disadvantages:
    - Unfair because processes with more lottery tickets have a higher chance of being run.
    - Probabilistic and thus unpredictable.

High overhead as it needs to generate a random ticket and search for the process having that ticket.

Stride Scheduling

Similar to Lottery Scheduling, but assigns each process a stride value that is the inverse of the number of tickets. Keeps a counter for each process, called its pass value, and selects the process with the lowest pass. When a process is run, increment its pass value by its stride.

- Can be preemptive or non-preemptive.
- Probabilistic.
- Starvation-free.
- Advantages:
    - Starvation-free because each process gets a non-zero chance of being run at any given time.
    - Ensures that each process gets exactly its prescribed proportion of the CPU time, unlike Lottery Scheduling.
- Disadvantages:
    - Requires global state to keep track of pass values.
    - Hard to determine the initial pass value for a process that enters after other processes have been running for a while. If the pass is 0, that new process will use up all the CPU until its pass value catches up with those of the other processes.

Week 4 - Threads

Read, understand, and summarize thread and related material in Lecture 4.

Threads

A **thread** is a sequence of instructions that represents an independent subtask of a process. A process can run multiple threads.

Each thread maintains its own CPU context, which includes a program counter, a stack pointer, the CPU register state, and a stack. Threads of the same process share that process' address space and other data such as PID and I/O status. This makes context switching for threads faster than for processes since only the CPU context needs to be switched. On the other hand, processes have isolated address spaces, making data sharing and communication costly.

It can be said that a thread is the basic unit of CPU scheduling because it represents a sequence of instructions. Meanwhile, a process is the basic unit of resources because it owns an isolated block of memory.

A program can be broken down into threads to achieve concurrency. The ability to have multiple threads running is called **multithreading**. In a process, threads run concurrently based on the scheduling strategy of the operating system.

There are many ways to work with threads, but two common strategies are:

1. **Manager/worker**: A manager thread deals with I/O while delegating work to worker threads. Worker threads can be created dynamically or obtained from a thread pool.
2. **Pipeline**: Each thread is responsible for a step in the whole process. The output of one thread is fed into the next in a producer-consumer relationship.

The advantages of using threads are:

- A task can be logically broken down into concurrently-executing units.
- Communication between threads is cheaper than between processes.
- Delegating work to other threads can free up the main thread for operations such as I/O.

The disadvantages of using threads are:

- Threads make the program harder to debug.
- Handling the interactions between threads is difficult.
- Creating and destroying threads is costly in terms of memory and computation.

Pthreads

In POSIX-compliant operating systems, the **Pthreads** API can be used for multi-threaded programming. The main thread exists in the `main()` function and spawns other threads. Some functions in the API are:

- `pthread_create`: Creates a thread to execute a given routine until its completion or until it calls `pthread_exit`.
- `pthread_exit`: Exits the calling thread and returns its status. This status is available to another thread in the same process if it calls `pthread_join` with the calling thread ID.
- `pthread_join`: Blocks this thread until another thread, specified by the given ID, finishes.
- `pthread_yield`: Gives up the CPU and places the calling thread at the end of the run queue.

User-level threads vs. kernel-level threads

The threads described in the sections above are **kernel-level threads**. These threads are created and managed by the kernel and are also called OS-level threads and native threads. They are supported by C and Java and are more difficult to work with because the programmer has to implement additional mechanisms like locks to prevent problems. Kernel-level threads have a higher context switching cost.

On the other hand, **user-level threads** or green threads are managed by the programming language or runtime environment. They have different names depending on the language, such as goroutine in Go or coroutine in Kotlin. These do not exist on the kernel level. They are easier to work with because they have been implemented by the language to avoid the problems with kernel-level threads. User-level threads can provide performance gains by living in the same kernel-level thread, minimizing the cost of context switching.

Threads in Java

There are two ways to create threads in Java:

1. Inherit from the `Thread` class and implement the `run()` method with the code that the thread wishes to run. Create an instance of that derived class and call its `start()` method. This approach is limited because Java only allows inheriting from one class.

```
public class MyThread extends Thread {
    @Override
    public void run() {
        // Tasks to perform here.
    }
}

public class Program {
    public static void main(String[] args) {
        MyThread thread = new MyThread();

        thread.start();
    }
}
```

2. Implement the `Runnable` interface and its `run()` method with the code that the thread wishes to run. Create an instance of that implementing class and pass it into the constructor of the `Thread` class.

```
public class SomeOperation implements Runnable {
    @Override
    public void run() {
        // Tasks to perform here.
    }
}

public class Program {
    public static void main(String[] args) {
        Thread thread = new Thread(new SomeOperation());

        thread.start();
    }
}
```

Many classes provided by Java are not thread-safe, so checking the documentation is necessary. An example of this is `ArrayList` from `java.util`.

When sharing a variable among different threads, any action performed on it must be **atomic** (a.k.a. cannot be divided into sub-steps). If not, the result will be inconsistent due to race conditions among the threads. This example shows the inconsistency caused by the non-atomic increment action on a shared integer:

```
public class Increment implements Runnable {
    static int classData = 0; // shared integer

    @Override
    public void run() {
        for (int i = 0; i < 10000000; i++) {
            classData++; // non-atomic action
        }

        System.out.println("classData: " + classData);
    }
```

```
    public static void main(String[] args) {
        Increment instance1 = new Increment();
        Increment instance2 = new Increment();

        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance2);

        t1.start();
        t2.start();
    }
}
```

The result, in theory, should be 20,000,000, but the console instead shows:

```
classData: 10014322
classData: 10014322
```

As there is a cost to creating threads, creating a large number of short-lived threads is expensive. Instead, one should use a **thread pool**, which creates a fixed number of threads and keeps them alive for reuse. This is useful in client-server applications like a web server, which receive a large number of quick-to-process requests. If a thread pool is used, incoming requests can be kept in a queue waiting to be assigned to a thread. Determining the right size for the thread pool is important to avoid starvation.

Week 5 - Lock I

Read, understand, and summarize lock and related materials in Lecture 5.

The need for mutual exclusion

The challenge to sharing mutable data among threads is that they can all attempt to change it concurrently, leading to inconsistent and unpredictable results. In concurrent programming, a **critical section** is the portion of code that accesses the shared mutable data. The code below contains contains a critical section in the `run` method:

```
public class Increment implements Runnable {
    static int classData = 0; // shared integer

    @Override
    public void run() {
        // CRITICAL SECTION
        for (int i = 0; i < 10000000; i++) {
            classData++; // non-atomic action
        }

        System.out.println("classData: " + classData);
        // END OF CRITICAL SECTION
    }

    public static void main(String[] args) {
        Increment instance1 = new Increment();
        Increment instance2 = new Increment();

        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance2);

        t1.start();
```

```
        t2.start();
    }
}
```

Multiple threads running the critical section at the same time is dangerous, especially if the actions they perform on the shared data is **non-atomic**. In the example above, the increment operation actually consists of 3 machine-level instructions: (1) load the data from the `classData` variable, (2) increment `classData`, and (3) write the new value back to the variable.

A **race condition** occurs when multiple threads access the critical section at around the same time. Race conditions make the program **indeterminate**, meaning that its output is unpredictable and varies from run to run. This happens because the timing of threads may cause non-atomic actions to be unfinished before another one is performed. Bugs caused by race conditions are very hard to trace and reproduce due to their unpredictability.

To prevent bugs like this from happening, threads must not be allowed to access the critical section at the same time. This act is called **mutual exclusion** or **synchronization**.

Locks

Locks are a way to achieve mutual exclusion. A lock is an abstraction that can be “owned” by a thread, giving it exclusive access to the critical section. A thread must “own” the lock if it wants to enter the critical section and cannot do so unless the lock has been released. After the lock-owning thread is finished with the critical section, it can release the lock for other threads.

A lock is an abstraction that support two key operations:

- `lock`: Gives the calling thread ownership of the lock. If the lock is already owned by another thread, the current thread blocks until the lock is released, at which point it will contend with the other threads for lock ownership.
- `unlock`: Releases the lock from the calling thread, allowing other threads to take ownership.

One potential danger to using locks is **deadlock**. Deadlock refers to the situation when two threads are waiting for each other to release locks and thus become stuck indefinitely. Imagine a case where an operation requires a thread to obtain both locks L1 and L2. Due to the timing of the threads, thread A acquires L1 first and thread B acquires L2 first. Now, A has to wait for B to release L2 before it can release L1, but B needs L1 to release L2.

Locks in Java

The Lock interface

In Java, the Lock abstraction is captured in the `Lock` interface, which also specifies several other useful Lock operations. Three standard classes that implement this interface are ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock. This example, which is based on the flawed code snippet above, shows how `ReentrantLock` can be used:

```
import java.util.concurrent.locks.Lock;
import java.util.concurrent.locks.ReentrantLock;

public class IncrementTest implements Runnable {
    static int classData = 0; // shared integer

    static Lock lock = new ReentrantLock();
```

```
    @Override
    public void run() {
        lock.lock();

        try {
            // CRITICAL SECTION
            for (int i = 0; i < 10000000; i++) {
                classData++; // non-atomic action
            }

            System.out.println("classData: " + classData);
            // END OF CRITICAL SECTION
        } finally {
            lock.unlock();
        }
    }

    public static void main(String[] args) {
        IncrementTest instance1 = new IncrementTest();
        IncrementTest instance2 = new IncrementTest();

        Thread t1 = new Thread(instance1);
        Thread t2 = new Thread(instance2);

        t1.start();
        t2.start();
    }
}
```

The critical section is sandwiched between `lock()` and `unlock()` to ensure that only one thread can run it and that the lock is released after the critical section finishes. As the code inside the critical section can throw an exception and halt the program, it is advisable to wrap it in a `try` block with a `finally` block that guarantees to release the lock. The output of the program now becomes what we expect it to be:

```
classData: 10000000
classData: 20000000
```

The `synchronized` keyword

An alternative to using the `Lock` implementations is using the `synchronized` keyword, which can be applied to methods and code blocks. Synchronized methods and code blocks both call `lock` and `unlock` underneath the hood. They make use of objects' intrinsic locks instead of requiring the programmer to instantiate an explicit `Lock` instance.

Synchronized instance methods make use of the instance's intrinsic lock while static methods make use of the class's intrinsic lock. The code below shows the above program rewritten using synchronized methods, with both threads now sharing one instance of `IncrementTest` to use its intrinsic lock:

```
public class IncrementTest implements Runnable {
    static int classData = 0; // shared integer

    @Override
    public synchronized void run() {
        // CRITICAL SECTION
        for (int i = 0; i < 10000000; i++) {
            classData++; // non-atomic action
```

```
        }

        System.out.println("classData: " + classData);
        // END OF CRITICAL SECTION
    }

    public static void main(String[] args) {
        IncrementTest instance = new IncrementTest();

        // Note that both threads must share the same instance here.
        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);

        t1.start();
        t2.start();
    }
}
```

Synchronized code blocks require the programmer to specify an object whose intrinsic lock would be used. It can be the current instance, signified by the `this` keyword, or an instance of `Object`. Use code blocks instead of methods when only a portion of the method body requires synchronization. The code below shows an example. Again, note that both threads share the same instance of `IncrementTest`. `this` tells the threads to use the instance's lock.

```
public class IncrementTest implements Runnable {
    static int classData = 0; // shared integer

    @Override
    public void run() {
        synchronized(this) {
            // CRITICAL SECTION
            for (int i = 0; i < 10000000; i++) {
                classData++; // non-atomic action
            }

            System.out.println("classData: " + classData);
            // END OF CRITICAL SECTION
        }

        // I can also do some stuff that does not require synchronization here.
    }

    public static void main(String[] args) {
        IncrementTest instance = new IncrementTest();

        // Note that both threads must share the same instance here.
        Thread t1 = new Thread(instance);
        Thread t2 = new Thread(instance);

        t1.start();
        t2.start();
    }
}
```

While a synchronized instance method is being executed by a thread, other threads cannot execute that method on the same instance. However, they are still allowed to execute that method on other instances. This requires programmers to be mindful of which instances (and thus locks) are being used by the threads.

The constructor of a class cannot be synchronized.

Synchronized methods and blocks offer more syntactic convenience but using the `Lock` interface is generally better because it offers greater control and flexibility. For example, one can specify a timeout, check if the lock is acquired, or interrupt the waiting thread.

Week 6 - Lock II

Read, understand, and summarize lock II and related materials in Lecture 6.

Building a good lock

A good lock requires support from both the operating system and underlying software. It should meet the following criteria:

1. **Mutual exclusion**: The lock should prevent threads from accessing a critical section at the same time.
2. **Fairness**: The lock should ensure that every thread gets a fair chance to own it.
3. **Performance**: The lock should incur minimal overhead when used.

Lock implementations

Controlling interrupts

A crude implementation of locks, which works on single-processor CPUs, is to prevent the CPU from interrupting the owning threads. When a thread calls `lock()` and the lock is free, that thread cannot be interrupted by the CPU by any means. Only when that thread calls `unlock()` is it free for interruption. As a single-processor CPU can only execute one thread at a time, doing so guarantees that no other threads can access the critical section.

This approach comes with clear disadvantages:

1. It does not work on multi-processor CPUs, which can run threads on different processors.
2. It places too much trust on the programs. A program can lock and never unlock, thus hogging all the CPU time.
3. Turning off interrupts can interfere with other applications. For example, a process that requests an I/O operation cannot be woken up by the CPU when that request finishes.
4. It is inefficient.

Due to these disadvantages, controlling interrupts is only used in certain situations, such as by the OS itself.

Spinning

Spinning (also called busy-waiting) is the use of a `while` loop to repeatedly check if a condition is fulfilled. Spinning blocks the thread until the condition becomes true, making it a reasonable basis for a lock. Spin locks have bad performance, especially on single-processor CPUs, due to the time wasted by the loop.

With Loads/Stores

The most basic (and also futile) attempt at spin locks is to use a single boolean variable to indicate whether the lock is owned or not. A thread that calls `lock` first checks this flag to see if the lock is owned, and if not, sets the flag to true to block other threads. When it finishes the critical section, it calls `unlock` to clear the flag, allowing other threads to take ownership of the lock.

The problem with using this primitive flag is that the act of checking and setting the flag is non-atomic, thus breaking mutual exclusion.

With Test-And-Set or Compare-And-Swap

Suppose the underlying hardware supports an atomic operation called Test-And-Set which assigns a new value to a given memory location and returns the old value. Instead of checking and setting the flag like with Loads/Stores, the lock now uses this atomic operation to do both things at once. This achieves mutual exclusion and results in a correct lock, though it does not guarantee fairness and leads to poor performance on single-processor CPUs.

A more powerful alternative to Test-And-Set is Compare-And-Swap, which is also atomic. It compares the value at a given memory location with an expected value, assigns a new value to that location if they are equal, and returns the old value.

With Fetch-And-Add

Suppose the underlying hardware supports an atomic operation called Fetch-And-Add which increments the value at a given memory location by 1 and returns the old value. A spin lock with Fetch-And-Add uses two numbers: a ticket and a turn (both initially 0). When a thread calls `lock`, it first obtains a ticket number from the lock with Fetch-And-Add. The lock maintains the ticket number, which is incremented every time Fetch-And-Add is performed. The thread then spins until the received ticket is equal to the lock's turn variable. The only way for the turn variable to change is if the lock-owning thread calls `unlock`, incrementing the turn number by 1.

This approach guarantees a correct and fair lock as now all threads are given an equal chance to own the lock. However, due to spinning, its performance is still bad, especially on single-processor CPUs.

Yielding

One problem with using spin locks is when a context switch happens in a critical section, other threads will have to wait indefinitely until the lock-owning thread is run again, wasting CPU time.

An alternative to spinning is yielding. Suppose the operating system supports an operation called Yield that can be called by a thread to relinquish the CPU for other threads. Underneath the hood, Yield enables a thread to deschedule itself into the ready state.

A yield lock is structured in the same way as a spin lock using Test-And-Set or Compare-And-Swap. However, instead of waiting and doing nothing in the while loop, it calls `yield()` to deschedule itself.

Using queues

The problem with the above approaches is that they are impacted by the choice of the scheduler. If the scheduler deschedules the lock-owning thread, the other threads will either run-and-spin or run-and yield, leading to waste and starvation.

To solve these issues, we need a queue to keep track of the threads that are waiting for the lock and two operations supported by the OS:

1. `park`: Puts the calling thread to sleep.
2. `unpark(threadId)`: Wakes up the thread with the provided thread ID.

When a thread tries to acquire the lock while it is being owned by another thread, it is put to sleep with `park` and its thread ID is added to the queue. When the lock-owning thread releases the lock, the first waiting thread is popped from the queue, woken up with `unpark`, and granted ownership of the lock.