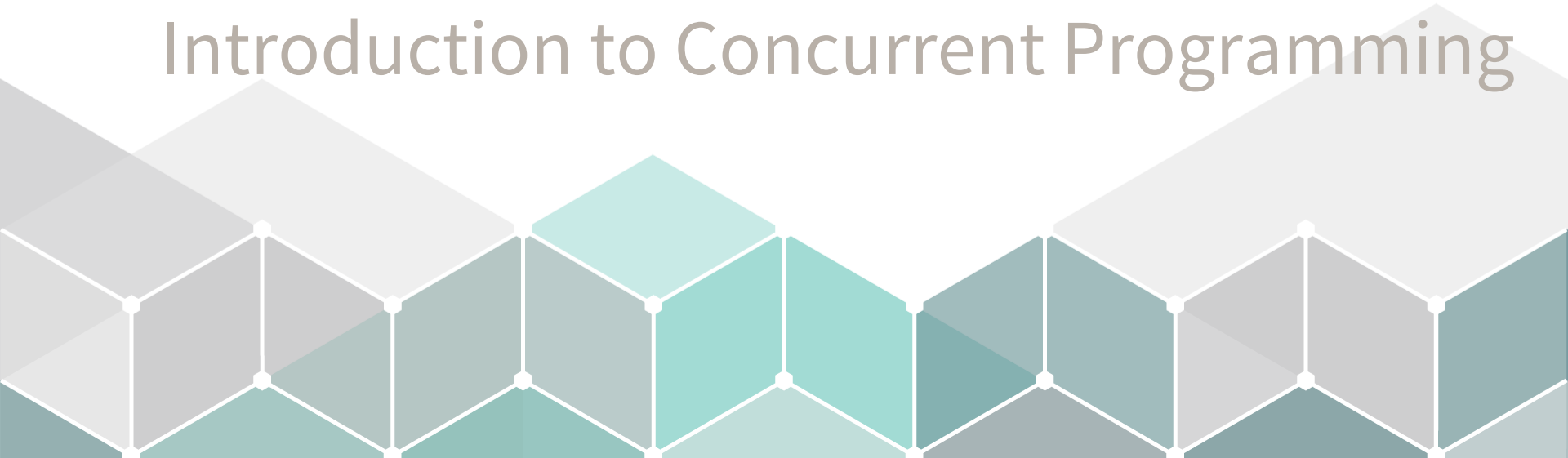
Østfold University College

# Thread Programming 1

## Introduction to Concurrent Programming

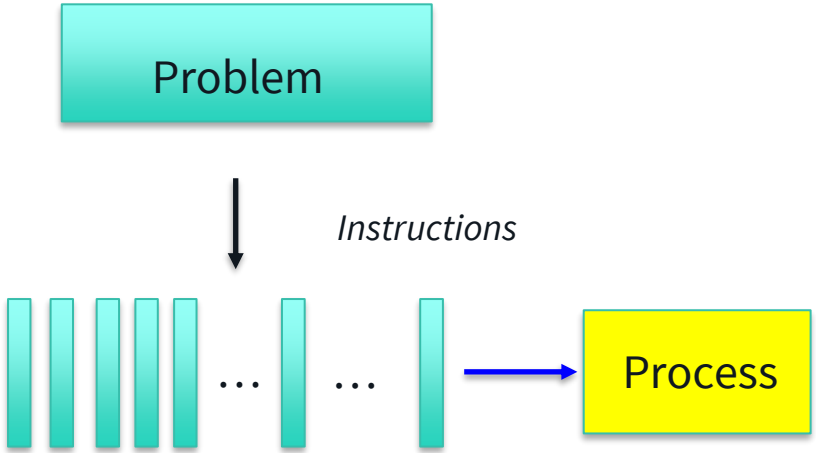


# Why Parallelism ?

Østfold University College

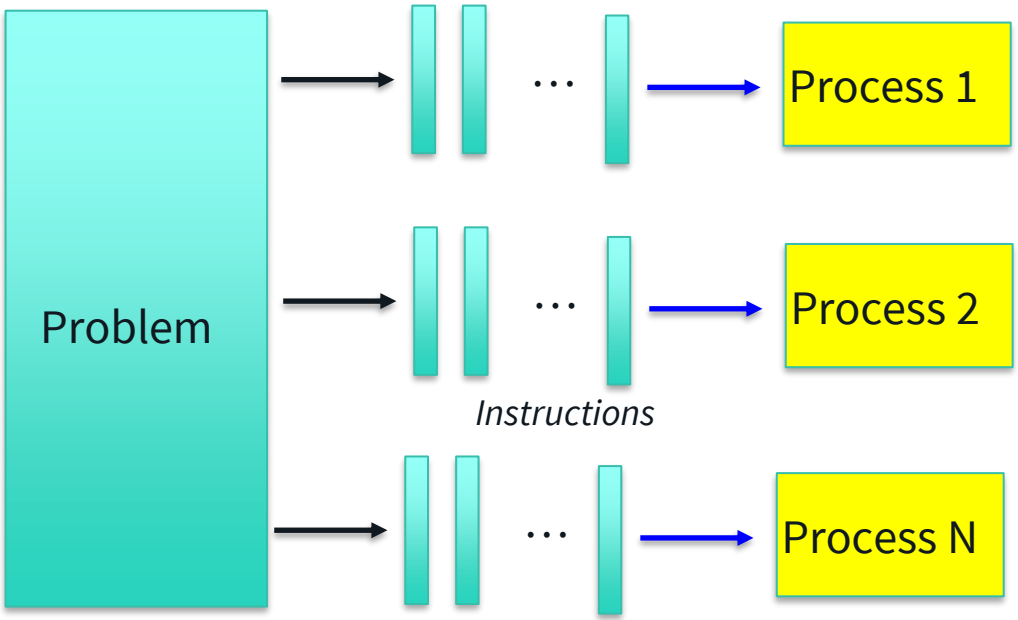
# Serial computing

- A problem is partitioned into a stream of instructions
- The instructions are executed sequentially, one after another
- Run on a single processor/CPU
- One instruction executed at a time



# Parallel computing

- Parallel: doing things simultaneously
- Solving one large problem by doing things simultaneously

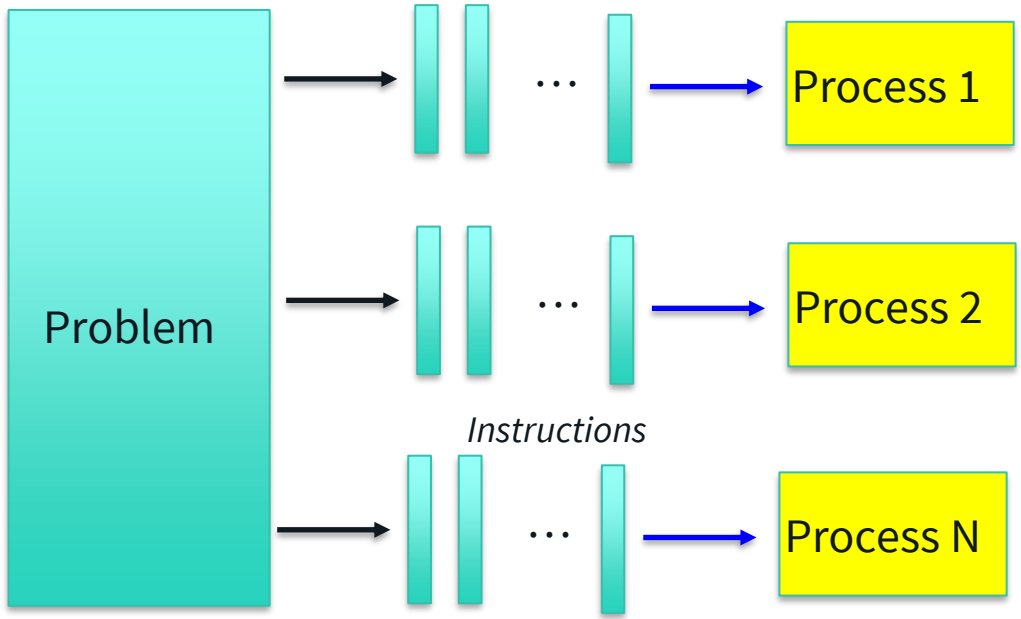


# Parallel computing: sub-problem

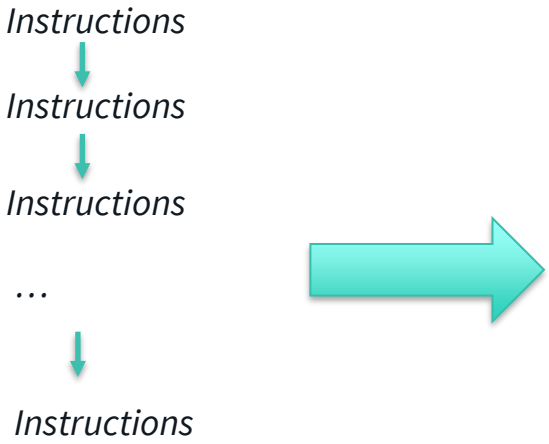
Østfold University College

A problem is broken into a number of sub-problems which can be solved simultaneously

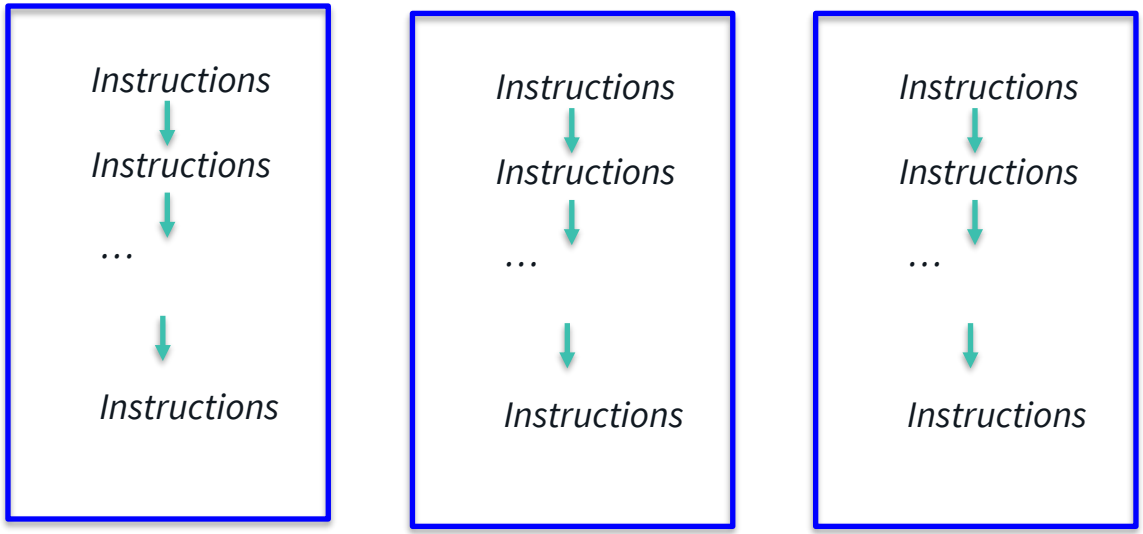
- Each sub-problem run in different processor/CPU
- Broken into stream of instructions
- Instructions run sequentially
- Instructions in different sub-problem run simultaneously



# Serial Programming




# Parallel Programming



# Why?

- Increase the speed-up of the system

Østfold University College

# Parallelism from Implementation point of view

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int n = 0, max = 0, lineCount = 0;
    char ch;
    FILE* file;

    file = fopen("Array.txt", "r");
    while ((ch = (char)fgetc(file)) != EOF) {
        if (ch == '\n'){
            lineCount++;
        }
    }
    fclose(file);
    file = fopen("Array.txt", "r");
    fscanf(file, "%d", &n);

    if (n > lineCount) {
        printf("Error: Not enough numbers in text file!\n");
        return 0;
    }
    int* A;
    A = (int*)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        fscanf(file, "%d", &A[i]);
        printf("%d\n", A[i]);

        if (A[i] > max) {
            max = A[i];
        }
    }
    printf("Max: %d\n", max);
    free(A);
    fclose(file);
    return 0;
}
```

Østfold University College

# Serial programming

Each instruction is run sequentially, one after another

```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int n = 0, max = 0, lineCount = 0;
    char ch;
    FILE* file;

    file = fopen("Array.txt", "r");
    while ((ch = (char)fgetc(file)) != EOF) {
        if (ch == '\n'){
            lineCount++;
        }
    }
    fclose(file);
    file = fopen("Array.txt", "r");
    fscanf(file, "%d", &n);

    if (n > lineCount) {
        printf("Error: Not enough numbers in text file!\n");
        return 0;
    }
    int* A;
    A = (int*)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        fscanf(file, "%d", &A[i]);
        printf("%d\n", A[i]);

        if (A[i] > max) {
            max = A[i];
        }
    }
    printf("Max: %d\n", max);
    free(A);
    fclose(file);
    return 0;
}
```

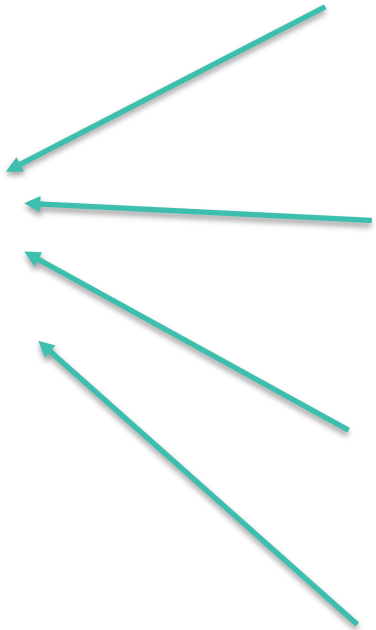
Østfold University College

# Parallel programming

- Several instructions or several blocks of code are running at the same time

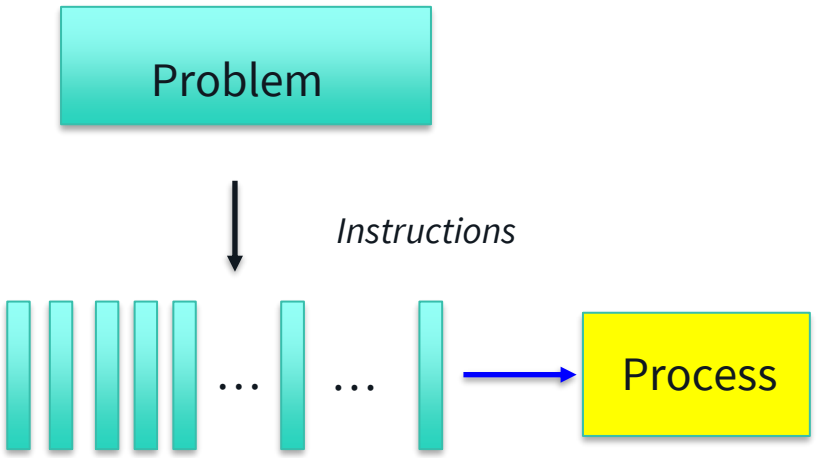
# Parallelism

Make them
run in parallel



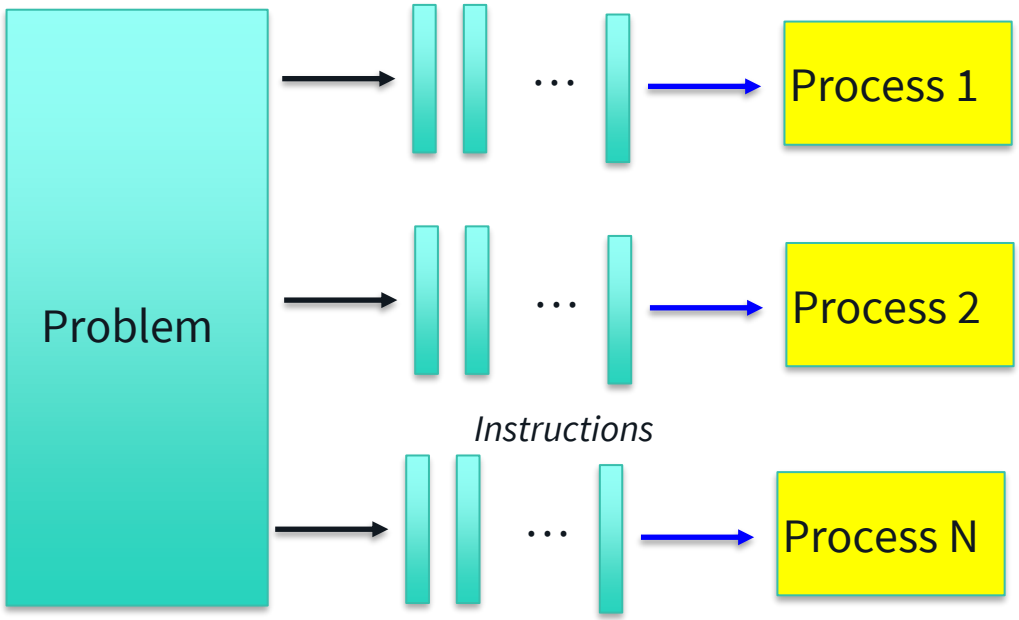
```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int n = 0, max = 0, lineCount = 0;
    char ch;
    FILE* file;

    file = fopen("Array.txt", "r");
    while ((ch = (char)fgetc(file)) != EOF) {
        if (ch == '\n'){
            lineCount++;
        }
    }
    fclose(file);
    file = fopen("Array.txt", "r");
    fscanf(file, "%d", &n);

    if (n > lineCount) {
        printf("Error: Not enough numbers in text file!\n");
        return 0;
    }
    int* A;
    A = (int*)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        fscanf(file, "%d", &A[i]);
        printf("%d\n", A[i]);

        if (A[i] > max) {
            max = A[i];
        }
    }
    printf("Max: %d\n", max);
    free(A);
    fclose(file);
    return 0;
}
```

# Serial Programming




- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

# Parallel Programming

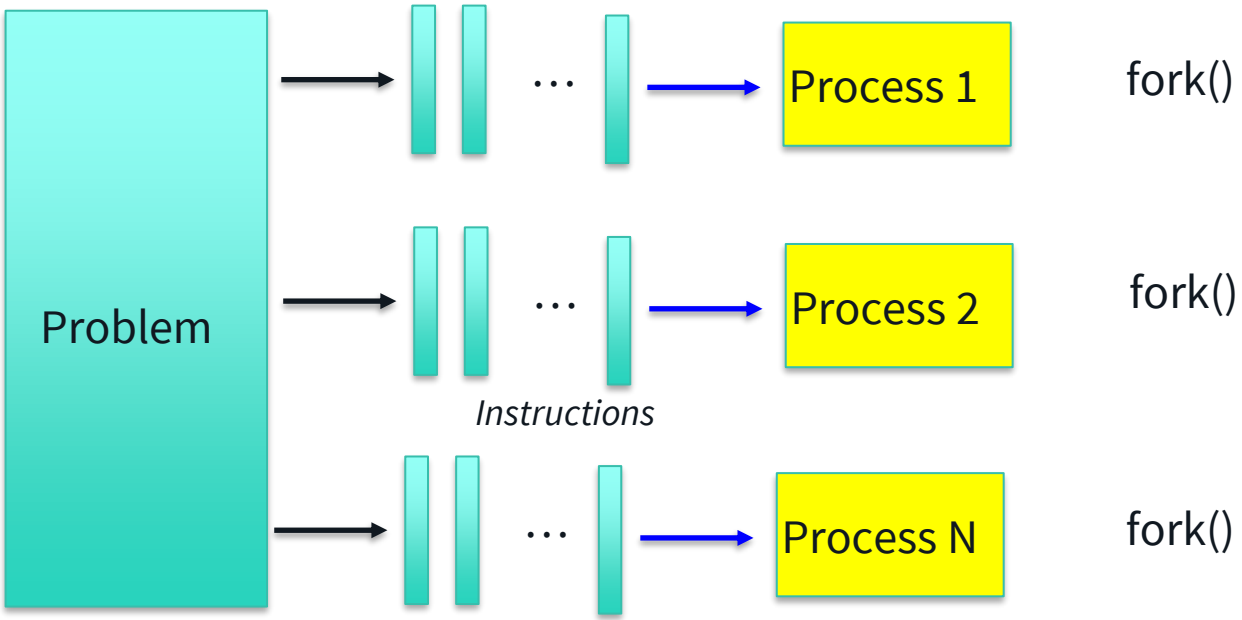



- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different processors
- An overall control/coordination mechanism is employed

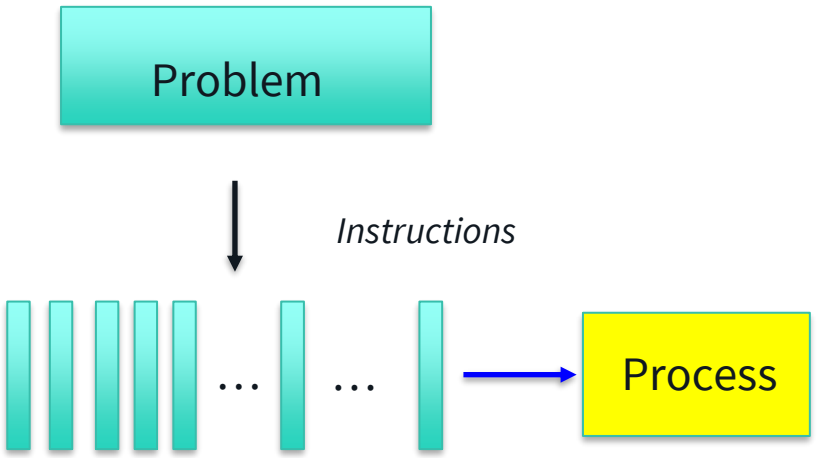
# Parallelism by using process

Østfold University College

- Parallel: doing things simultaneously
- Solving one large problem by doing things simultaneously

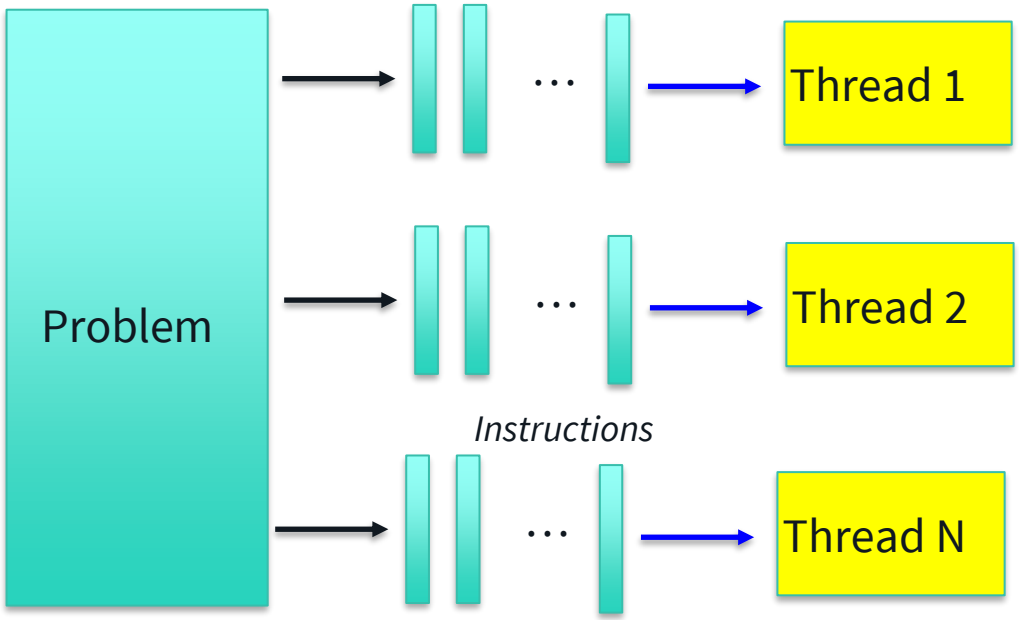


# Serial Programming




- A problem is broken into a discrete series of instructions
- Instructions are executed sequentially one after another
- Executed on a single processor
- Only one instruction may execute at any moment in time

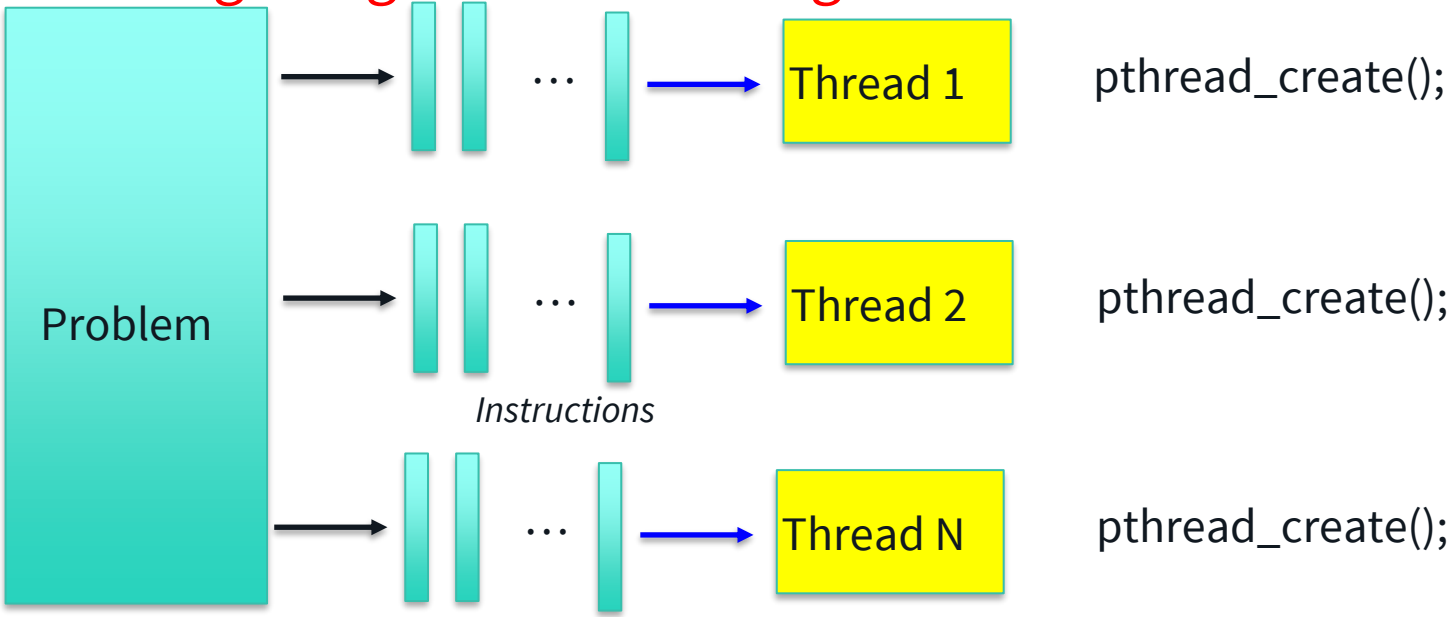
# Concurrent Programming



# Parallelism by using threads

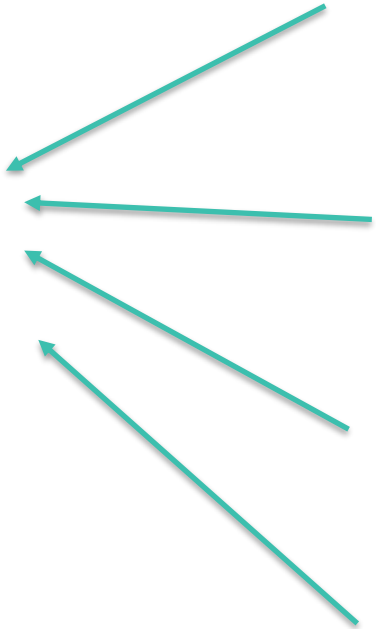
Østfold University College

- Parallel: doing things simultaneously
- Solving one large problem by doing things simultaneously
- Concurrent: doing things in time-sharing fashion



# Concurrent programming

Running in time-sharing fashion



```c
#include <stdlib.h>
#include <stdio.h>

int main() {
    int n = 0, max = 0, lineCount = 0;
    char ch;
    FILE* file;

    file = fopen("Array.txt", "r");
    while ((ch = (char)fgetc(file)) != EOF) {
        if (ch == '\n'){
            lineCount++;
        }
    }
    fclose(file);
    file = fopen("Array.txt", "r");
    fscanf(file, "%d", &n);

    if (n > lineCount) {
        printf("Error: Not enough numbers in text file!\n");
        return 0;
    }
    int* A;
    A = (int*)malloc(n * sizeof(int));

    for (int i = 0; i < n; i++) {
        fscanf(file, "%d", &A[i]);
        printf("%d\n", A[i]);

        if (A[i] > max) {
            max = A[i];
        }
    }
    printf("Max: %d\n", max);
    free(A);
    fclose(file);
    return 0;
}
```

# Example

- Exercise 3, Lab 2 assignment

## 8.3 Exercise 3 (60 points)

In an input file which includes all integer numbers, the first element indicates the total number of elements in the file (excluding itself).

- Write a C program to calculate the summation of all elements in *testcase0.txt* except the first element and print the result into screen. **(50 points).**
- Run your above code with *testcase1.txt*. What would you see and what would be your explaination for the output? **(10 points).**

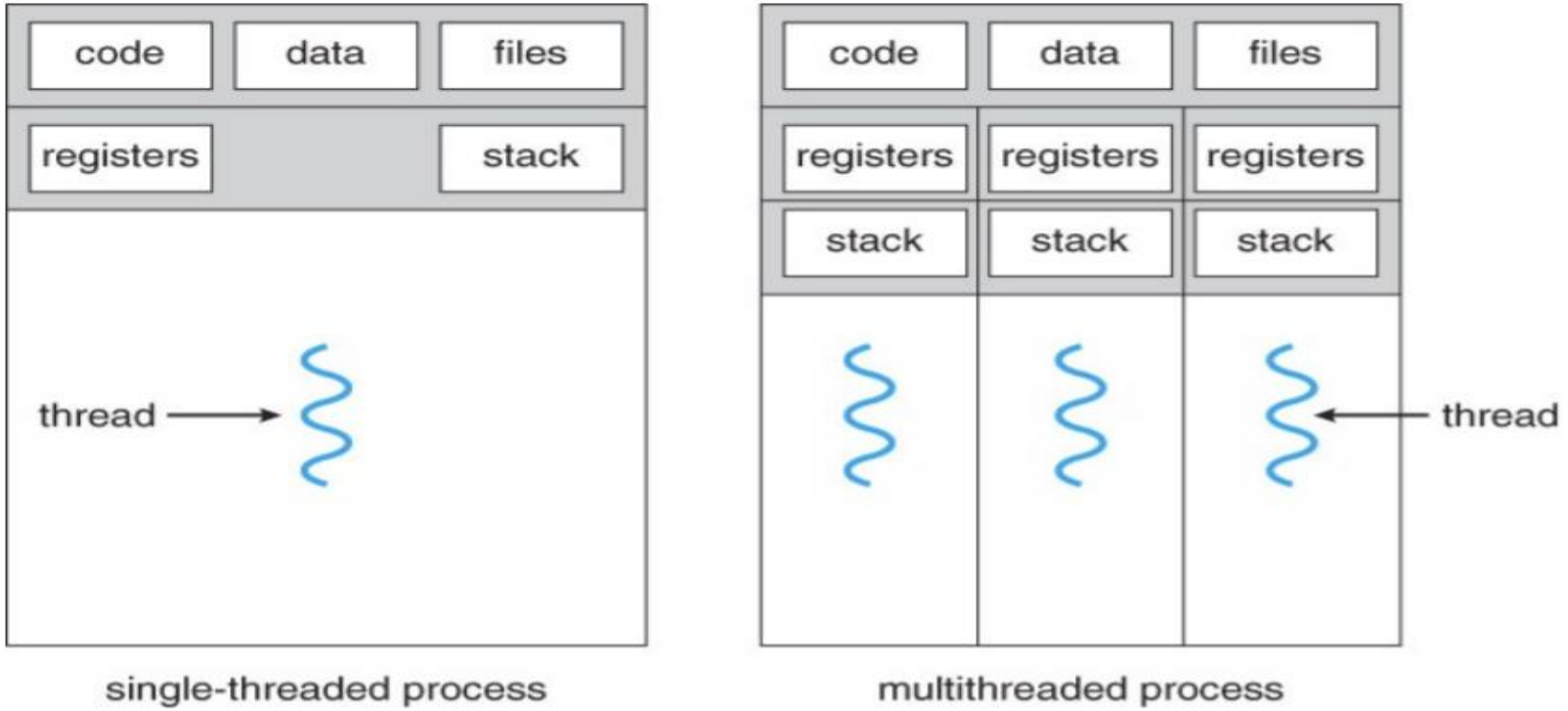
```c
int fileSummation(FILE* filePointer) {

    if (filePointer == NULL) {
        printf("Error! Could not open file\n");
        exit(-1);
    }

    int amountOfNumbers;
    fscanf(filePointer, "%d", &amountOfNumbers);
    printf("Number of files in testcase1: %d \n", amountOfNumbers);

    int sum = 0;
    int num;
    while(feof(filePointer) == 0) {
        fscanf(filePointer, "%d", &num);
        sum += num;
    }

    return sum;
}

int main() {

    FILE *testcase1 = fopen("testcase1.txt", "r");
    int sumTest1 = fileSummation(testcase1);
    fclose(testcase1);

    FILE *testcase2 = fopen("testcase2.txt" , "r");
    int sumTest2 = fileSummation(testcase2);
    fclose(testcase2);

    printf("Sum from testcase1: %d \n", sumTest1);

    //This returns a negative number. This is due to our testcase2 containt a
    //the maxmimum number that an int can hold. We could try to save out sum
    printf("Sum from testcase2: %d \n", sumTest2);
```

# Parallelism ?

Østfold University College

# Processes vs Threads

# Threads in a process




single-threaded process

multithreaded process

# Processes vs Threads

- Faster to create a new thread and terminate a thread compared to a process
- Faster to switch between two threads within a process
- More efficient communication
- Easy programming model

Østfold University College

# POSIX Threads

# POSIX Thread (Pthread)

- Used for Parallelism
- POSIX
  - Portable Operating System Interface
  - Specified by IEEE POSIX 1003.1c standard
- Popular in Unix System
- API and user-level thread libraries

# PThread API

- Pthread management
  - Creation, Termination and Joining
- Pthread synchronization
  - Race condition, mutex, semaphores…

# Thread Identifying

- Thread ID
  - Unique in a current process
  - Presesnted by type pthread_t
- Header file
  - #include <pthread.h>

# Create a Thread: pthread_create ()

Østfold University College

- int pthread_create (pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg);
- pthread_t *thread
  - Pointer to a pthread_t variable which is used to store thread if of new created thread
- const pthread_attr_t *attr
  - Pointer to a thread attribute object used to set thread attributes
  - NULL can be used to create a thread with default arguments
- void *(*start_routine) (void *)
  - Pointer to thread function containing code segment which is executed by the thread
- void *arg
  - Thread functions argument to the void

# Exit a Thread

- Entire process is terminated
- Interrupted by pthread_cancel()
- One of the threads calls exec()

# Example: Thread Creation and Termination

```c
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <unistd.h>


#define NUM_OF_THREADS      10

void *PrintMessage(void *ThreadId){
    long tid;
    tid = (long)ThreadId;
    printf("Hello World from Thread #%ld!\n", tid);
    pthread_exit(NULL);
 }

 int main (int argc, char *argv[]){
    pthread_t threads[NUM_OF_THREADS];
    int ret;
    long i;

    for(i=0; i<NUM_OF_THREADS; i++){
       printf("Creating Thread %ld in the main() function\n", i);
       ret = pthread_create(&threads[i], NULL, PrintMessage, (void *)i);
       if (ret){
          printf("ERROR in creating thread; return ERROR code %d\n", ret);
          exit(-1);
       }
    }
    pthread_exit(NULL);
    return 0;
 }
```

# Waiting for a thread: pthread_join ()

- int pthread_join (pthread_t thread, void **retval);
- pthread_join(): waits for the thread specified by thread to terminate
- If thread has already terminated, pthread_join() returns immediately
- thread must be joinable (default)


Q&A