CONCURRENCY PROGRAMMING

Contents

[1. How Does a Program Start? 2](#_Toc139987278)

[2. Process and Thread – How are these different? 2](#_Toc139987279)

[2.1. What is Process? 2](#_Toc139987280)

[2.2. What is Thread? 2](#_Toc139987281)

[2.3. Differences between Process and Thread? 2](#_Toc139987282)

[3. Multiprocessing 2](#_Toc139987283)

[3.1. Advantages of multiprocessing 2](#_Toc139987284)

[3.2. Disadvantages of multiprocessing 2](#_Toc139987285)

[3.3. Multiprocessing in C++ 2](#_Toc139987286)

[3.3.1. fork() 2](#_Toc139987287)

[3.3.2. Data of process 2](#_Toc139987288)

[3.4. Concurrency/Data sharing in multiprocessing 3](#_Toc139987289)

[3.4.1. Shared memory 3](#_Toc139987290)

[3.4.2. Socket 3](#_Toc139987291)

[3.4.3. Named Pipe 3](#_Toc139987292)

[4. Multithreading 3](#_Toc139987293)

[4.1. Advantages of multithreading 3](#_Toc139987294)

[4.2. Disadvantages of multithreading 3](#_Toc139987295)

[4.3. Multithreading in C++ 3](#_Toc139987296)

[4.4. Concurrency in multithreading 4](#_Toc139987297)

[4.4.1. <mutex> 4](#_Toc139987298)

[4.4.2. <condition\_variable> 4](#_Toc139987299)

[4.4.3. Semaphore concept 4](#_Toc139987300)

# How Does a Program Start?

A program starts by being executed or launched. It goes through these steps:

1. Compilation or Interpretation: The program's code is converted into machine-readable instructions.
2. Loading: The operating system loads the program into memory.
3. Initialization: The program sets up its initial state by allocating memory, establishing connections, initializing variables, and preparing any necessary resources.
4. Main Function: The program starts executing from a designated entry point, typically a main function.
5. Execution: The program executes its instructions sequentially, performing computations, interacting with input/output devices, and carrying out the desired tasks defined in its code.
6. Termination: The program finishes executing and releases any allocated resources.

# Process and Thread – How are these different?

[]

## What is Process?

[write 2-5 lines]

* A process in concurrency programming is an instance of a program executed concurrently.
* Processes run independently with their own memory space and system resources.
* They provide isolation and allow for parallel execution of multiple tasks.
* Processes communicate through inter-process communication (IPC) mechanisms.
* Process scheduling by the operating system determines execution order and resource allocation.

## What is Thread?

[write 2-5 lines]

* In concurrency programming, a thread is a lightweight unit of execution within a process.
* Threads run concurrently with other threads within the same process.
* They share the same memory space, file descriptors, and system resources.
* Threads enable parallelism and allow tasks to be performed simultaneously.
* Thread synchronization mechanisms are used to coordinate access to shared resources among threads

## Differences between Process and Thread?

… [Find at least 5 differences]

| **Process** | **Thread** |
| --- | --- |
| A process is an independent unit of execution. | A thread is a subset of a process. |
| Processes have their own memory space. | Threads within a process share the same memory space. |
| Inter-process communication is required for data sharing. | Threads can directly access shared memory. |
| Processes have their own file descriptors. | Threads within a process share file descriptors. |
| Process creation and termination involve more overhead. | Thread creation and termination are faster and require less overhead. |
| Processes provide better fault isolation. | A crash in one thread can lead to the termination of the entire process. |
| Processes can run on different processors and take advantage of multiprocessing. | Threads within a process run on the same processor. |
| Processes require explicit IPC mechanisms for communication and synchronization. | Threads can communicate and synchronize more easily through shared memory. |

# Multiprocessing

## Advantages of multiprocessing

**[Advantages 1]**: Increased Performance  
Multiprocessing allows for executing multiple processes simultaneously, improving overall performance.   
Example: Running multiple threads of a video encoding program concurrently to encode videos faster.

**[Advantages 2]**: Enhanced Resource Utilization  
Multiprocessing optimizes resource usage, such as CPU time and memory, by distributing tasks among multiple processes.   
Example: Running parallel processes to process large datasets, utilizing all available CPU cores effectively.

**[Advantages 3]**: Improved Fault Tolerance  
Multiprocessing enhances system reliability as failures in one process do not impact others.   
Example: In a web server, if one process encounters an error, other processes can continue serving requests.

## Disadvantages of multiprocessing

**[Disadvantages 1]**: Increased Complexity  
Multiprocessing introduces complexity due to the need for inter-process communication, synchronization, and resource management.   
Example: Coordinating data sharing between multiple processes while avoiding race conditions or deadlocks.

**[Disadvantages 2]**: Overhead  
Managing multiple processes incurs overhead in terms of context switching, memory management, and inter-process communication.   
Example: Context switching between processes adds overhead as the system needs to save and restore the state of each process.

## Multiprocessing in C++

### fork()

The fork() function in C++ is used to create a new process by duplicating the existing process

**[Example]**

1. #include <unistd.h>

2. #include <iostream>

3. int main() {

4. pid\_t pid = fork();

5. if (pid == 0) {

6. // Child process

7. std::cout << "This is the child process.\n";

8. } else if (pid > 0) {

9. // Parent process

10. std::cout << "This is the parent process.\n";

11. } else {

12. // Error

13. std::cerr << "Failed to create a new process.\n";

14. }

15. return 0;

16. }

**[Return value of fork()]**The fork() function returns different values in the parent and child processes. In the parent process, it returns the process ID (PID) of the child process, while in the child process, it returns 0.

### Data of process

**[Demo how processes access its data.]**In the process of managing data, a process can use and manage data in two main types: global variables and data used throughout the lifecycle of the process.

1. Global Variables:

* Global variables are accessible and usable by all processes in the system.
* Processes can read from and write to global variables, enabling data sharing.
* Global variables are stored in shared memory, accessible by processes using the same shared memory region.

1. Data Used Throughout the Lifecycle:

* This refers to data created and used by a process during its operation.
* It includes local variables specific to each process, with limited scope and access.
* Each process has its own memory space for storing this data.

Here's a simple example illustrating how processes can access and manage their data:

1. #include <stdio.h>

2. #include <unistd.h>

3.

4. int globalVariable = 100; // Global variable

5.

6. int main() {

7. int localVariable = 200; // Local variable

8.

9. pid\_t pid = fork(); // Create a child process

10.

11. if (pid == 0) {

12. // Child process

13. printf("Child Process - Global Variable: %d\n", globalVariable);

14. printf("Child Process - Local Variable: %d\n", localVariable);

15. globalVariable = 300; // Modify the global variable

16. localVariable = 400; // Modify the local variable

17. printf("Child Process - Modified Global Variable: %d\n", globalVariable);

18. printf("Child Process - Modified Local Variable: %d\n", localVariable);

19. } else if (pid > 0) {

20. // Parent process

21. printf("Parent Process - Global Variable: %d\n", globalVariable);

22. printf("Parent Process - Local Variable: %d\n", localVariable);

23. sleep(2);

24. printf("Parent Process - After Child Execution - Global Variable: %d\n", globalVariable);

25. printf("Parent Process - After Child Execution - Local Variable: %d\n", localVariable);

26. } else {

27. // Fork failed

28. fprintf(stderr, "Fork Failed.\n");

29. return 1;

30. }

31.

32. return 0;

33. }

* The code provided creates a parent process and a child process. Both processes have a global variable (globalVariable) and a local variable (localVariable).
* The specific output will depend on the execution of the operating system and the scheduling of the processes. However, assuming the child process is successfully created, the result could be:

Parent Process - Global Variable: 100

Parent Process - Local Variable: 200

Child Process - Global Variable: 100

Child Process - Local Variable: 200

Child Process - Modified Global Variable: 300

Child Process - Modified Local Variable: 400

Parent Process - After Child Execution - Global Variable: 300

Parent Process - After Child Execution - Local Variable: 200

**[Create a graph/diagram to demo how processes manage data]**

**Modified**

GlobaleVariable = 100

main()

{

LocalVariable = 200

fork()  
 ….

}

Parent Process

PID > 0

PID = 0

**Modified**

**Modified**

## Concurrency/Data sharing in multiprocessing

### Shared memory

**[Concept – 2-5 lines]**  
Shared memory is a method of inter-process communication (IPC) where multiple processes can access the same memory region. This allows processes to share data without the need for explicit message passing.

**[Example code]**

1. #include <stdio.h>

2. #include <sys/ipc.h>

3. #include <sys/shm.h>

4.

5. int main() {

6. // Create a shared memory segment

7. key\_t key = ftok("shared\_memory\_example", 1234);

8. int shmid = shmget(key, 1024, 0666 | IPC\_CREAT);

9.

10. // Attach the shared memory segment to the process

11. char \*data = (char\*)shmat(shmid, (void\*)0, 0);

12.

13. // Write data to the shared memory

14. sprintf(data, "Hello, shared memory!");

15.

16. // Detach the shared memory segment

17. shmdt(data);

18.

19. return 0;

20. }

21.

**[Output of example code]**This code creates a shared memory segment and writes the string "Hello, shared memory!" into it. The shared memory can then be accessed by other processes that attach to the same shared memory segment

### Socket

**[Concept – 2-5 lines]**Sockets are a method of IPC that allow communication between processes over a network or the same machine. Processes can send and receive data through sockets.

**[Example code]**

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <unistd.h>

4. #include <string.h>

5. #include <sys/types.h>

6. #include <sys/socket.h>

7. #include <netinet/in.h>

8.

9. int main() {

10. // Create a socket

11. int sockfd = socket(AF\_INET, SOCK\_STREAM, 0);

12.

13. // Bind the socket to a port

14. struct sockaddr\_in server\_address;

15. server\_address.sin\_family = AF\_INET;

16. server\_address.sin\_port = htons(12345);

17. server\_address.sin\_addr.s\_addr = INADDR\_ANY;

18. bind(sockfd, (struct sockaddr\*)&server\_address, sizeof(server\_address));

19.

20. // Listen for connections

21. listen(sockfd, 5);

22.

23. // Accept a client connection

24. int client\_socket = accept(sockfd, NULL, NULL);

25.

26. // Send data to the client

27. char message[] = "Hello, socket!";

28. send(client\_socket, message, sizeof(message), 0);

29.

30. // Close the sockets

31. close(client\_socket);

32. close(sockfd);

33.

34. return 0;

35. }

**[Output of example code]**This code creates a socket, binds it to port 12345, and listens for incoming connections. When a client connects, it sends the message "Hello, socket!" to the client.

### Named Pipe

**[Concept – 2-5 lines]**Named pipes, also known as FIFOs (First-In-First-Out), are a form of inter-process communication where processes can write and read data to and from a common pipe file

[Example code]

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <unistd.h>

4. #include <sys/types.h>

5. #include <sys/stat.h>

6. #include <fcntl.h>

7. #include <string.h>

8.

9. int main() {

10. // Create a named pipe (FIFO)

11. mkfifo("named\_pipe\_example", 0666);

12.

13. // Open the named pipe for writing

14. int fd = open("named\_pipe\_example", O\_WRONLY);

15.

16. // Write data to the named pipe

17. char message[] = "Hello, named pipe!";

18. write(fd, message, sizeof(message));

19.

20. // Close the named pipe

21. close(fd);

22.

23. return 0;

24. }

**[Output of example code]**This code creates a named pipe called "named\_pipe\_example" and opens it for writing. It then writes the message "Hello, named pipe!" into the pipe.

# Multithreading

## Advantages of multithreading

**[Advantages 1]**: Real-Time Processing and Responsiveness  
Multithreading allows for real-time processing and responsiveness in applications that require immediate or continuous updates.  
Example: A music production software that uses multithreading to handle real-time audio processing while providing a responsive user interface

**[Advantages 2]**: Simplified Design and Modularity  
Multithreading facilitates a modular design approach, dividing complex tasks into smaller threads, simplifying the codebase, and improving maintainability  
Example: An image processing application with separate threads for loading, filtering, and saving images.:

…

## Disadvantages of multithreading

**[Disadvantages 1]**: Complexity of Synchronization  
Multithreading introduces the need for synchronization mechanisms to manage shared resources, which can lead to complexity and synchronization issues.  
Example: Coordinating access to a shared database connection in a multi-threaded application, which requires careful synchronization to avoid data inconsistencies.

**[Disadvantages 2]**: Difficult Debugging and Testing  
Debugging and testing multi-threaded applications can be challenging due to the non-deterministic nature of thread execution.  
Example: Resolving race conditions in a multi-threaded financial trading application, where timing-dependent thread interactions can be hard to reproduce and debug.

…

## Multithreading in C++

[Example with no concurrency]

1. #include <iostream>

2. #include <thread>

3. #include <chrono>

4.

5. // Function to be executed in a separate thread

6. void threadFunction(int id) {

7. std::cout << "Thread " << id << " started." << std::endl;

8.

9. // Simulating some work by sleeping for a specific duration

10. std::this\_thread::sleep\_for(std::chrono::seconds(2));

11.

12. std::cout << "Thread " << id << " completed." << std::endl;

13. }

14.

15. int main() {

16. const int numThreads = 4;

17. std::thread threads[numThreads];

18.

19. // Create multiple threads and execute the thread function

20. for (int i = 0; i < numThreads; ++i) {

21. threads[i] = std::thread(threadFunction, i);

22. }

23.

24. std::cout << "Main thread started." << std::endl;

25.

26. // Wait for all threads to finish execution

27. for (int i = 0; i < numThreads; ++i) {

28. threads[i].join();

29. }

30.

31. std::cout << "Main thread completed." << std::endl;

32.

33. return 0;

34. }

35.

* In this example, Four separate threads are created by using an array of std::thread objects. The threadFunction() takes an id parameter, and each thread executes this function independently**.**
* Each thread starts by printing a message indicating its ID. Then, the thread simulates some work by sleeping for 2 seconds using std::this\_thread::sleep\_for(). After that, the thread prints another message indicating its completion
* The main thread also starts by printing a message and then proceeds to create and start all the separate threads. It then waits for each thread to complete its execution using the join() function. Finally, the main thread prints a completion message

**[Output of example]**

Main thread started.

Thread 0 started.

Thread 1 started.

Thread 2 started.

Thread 3 started.

Thread 0 completed.

Thread 1 completed.

Thread 2 completed.

Thread 3 completed.

Main thread completed.

## Concurrency in multithreading

### <mutex>

**[Concept – 2-5 lines]**  
<mutex> is a header in C++ that provides synchronization primitives, specifically mutexes, to achieve mutual exclusion and protect shared resources in a multi-threaded environment. Mutexes ensure that only one thread can access the protected resource at a time, preventing data races and ensuring data integrity

**[Example code]**

1. #include <iostream>

2. #include <thread>

3. #include <mutex>

4.

5. std::mutex mtx;

6. int sharedData = 0;

7.

8. void incrementData() {

9. std::lock\_guard<std::mutex> lock(mtx); // Lock the mutex

10. sharedData++; // Increment the shared data

11. }

12.

13. int main() {

14. const int numThreads = 5;

15. std::thread threads[numThreads];

16.

17. // Create multiple threads to increment the shared data

18. for (int i = 0; i < numThreads; ++i) {

19. threads[i] = std::thread(incrementData);

20. }

21.

22. // Wait for all threads to finish execution

23. for (int i = 0; i < numThreads; ++i) {

24. threads[i].join();

25. }

26.

27. // Print the final value of the shared data

28. std::cout << "Shared data: " << sharedData << std::endl;

29.

30. return 0;

31. }

32.

* In this example, we have a shared data variable sharedData and multiple threads that increment its value. To ensure that the shared data is accessed in a thread-safe manner, we use a mutex (mtx) to lock and unlock access to the shared resource
* The program creates five threads, and each thread `increments the shared data variable sharedData. Since we are using a mutex to protect the access to sharedData, only one thread can increment it at a time. The final value of sharedData is the sum of all the increments performed by each thread.

**[Output of example code]**

Shared data: 5

### <condition\_variable>

**[Concept – 2-5 lines]?**<condition\_variable> is a header in C++ that provides synchronization mechanisms for thread coordination and communication. It allows threads to wait until a specific condition is met before proceeding, enabling efficient synchronization between threads.

**[Example code]**

1. #include <iostream>

2. #include <thread>

3. #include <mutex>

4. #include <condition\_variable>

5.

6. std::mutex mtx;

7. std::condition\_variable cv;

8. bool isDataReady = false;

9. int sharedData = 0;

10.

11. void producer() {

12. // Simulate some work

13. std::this\_thread::sleep\_for(std::chrono::seconds(2));

14.

15. // Produce data

16. {

17. std::lock\_guard<std::mutex> lock(mtx);

18. sharedData = 42;

19. isDataReady = true;

20. }

21.

22. // Notify the consumer

23. cv.notify\_one();

24. }

25.

26. void consumer() {

27. // Wait for the producer

28. std::unique\_lock<std::mutex> lock(mtx);

29. cv.wait(lock, []{ return isDataReady; });

30.

31. // Consume data

32. int data = sharedData;

33. std::cout << "Consumer received data: " << data << std::endl;

34. }

35.

36. int main() {

37. std::thread producerThread(producer);

38. std::thread consumerThread(consumer);

39. producerThread.join();

40. consumerThread.join();

41.

42. return 0;

43. }

* In this example, we have a producer and a consumer thread. The producer simulates some work and then produces data by setting the sharedData variable to 42 and the isDataReady flag to true. The consumer waits for the producer using a condition variable and consumes the data when the isDataReady flag becomes true.
* The producer and consumer threads are synchronized using a mutex and condition variable. The producer produces data and notifies the consumer, while the consumer waits for the producer's notification before consuming the data**.**

**[Output of example code]**

Consumer received data: 42

### Semaphore concept

**[Concept – 2-5 lines]**A semaphore is a synchronization primitive that allows controlling access to a shared resource by multiple threads. It maintains a counter that represents the number of available resources. Threads can acquire and release resources from the semaphore, and if no resources are available, the thread will be blocked until a resource becomes available.

**[Example code]**

1. #include <iostream>

2. #include <thread>

3. #include <mutex>

4. #include <condition\_variable>

5.

6. class Semaphore {

7. public:

8. Semaphore(int count = 0) : m\_count(count) {}

9.

10. void acquire() {

11. std::unique\_lock<std::mutex> lock(m\_mutex);

12. while (m\_count == 0) {

13. m\_cv.wait(lock);

14. }

15. --m\_count;

16. }

17.

18. void release() {

19. std::lock\_guard<std::mutex> lock(m\_mutex);

20. ++m\_count;

21. m\_cv.notify\_one();

22. }

23.

24. private:

25. int m\_count;

26. std::mutex m\_mutex\_;

27. std::condition\_variable m\_cv;

28. };

29.

30. Semaphore sem(1); // Semaphore with an initial count of 1

31.

32. void worker(int id) {

33. sem.acquire(); // Acquire the semaphore

34.

35. std::cout << "Worker " << id << " is performing some task." << std::endl;

36. // Simulate some work

37. std::this\_thread::sleep\_for(std::chrono::seconds(2));

38.

39. sem.release(); // Release the semaphore

40. }

41.

42. int main() {

43. const int numWorkers = 3;

44. std::thread workers[numWorkers];

45.

46. // Create multiple worker threads

47. for (int i = 0; i < numWorkers; ++i) {

48. workers[i] = std::thread(worker, i);

49. }

50.

51. // Wait for all worker threads to finish execution

52. for (int i = 0; i < numWorkers; ++i) {

53. workers[i].join();

54. }

55.

56. return 0;

57. }

* In this example, we have a Semaphore class that implements a counting semaphore with the acquire() and release() methods. The acquire() method waits until the count is greater than zero before decrementing it, and the release() method increments the count and notifies any waiting threads
* In the worker() function, each worker thread acquires the semaphore, performs some task (simulated by a sleep), and then releases the semaphore.
* By initializing the semaphore sem with an initial count of 1, we ensure that only one worker thread can acquire the semaphore at a time. This allows for controlled access to the critical section of code inside the worker() function.

**[Output of example code]**

Worker 0 is performing some task.

Worker 1 is performing some task.

Worker 2 is performing some task.