Асинхронность и Многопоточность

Лекторы:
Аспирант МФТИ, Шер Артём Владимирович
Аспирант МФТИ, Зингеренко Михаил Владимирович

15 октября 2024

Преимущества процессов перед потоками

- Процессы изолированы друг от друга (нет общей памяти, как у потоков).
- Обработка ошибок и отладка легче.
- Повышенная безопасность и стабильность за счет изоляции.

Системный вызов fork()

Системный вызов для создания нового процесса, который является копией текущего (родительского). Новому процессу назначается уникальный идентификатор процесса (PID). Особенности работы fork():

- После вызова fork(), оба процесса (родительский и дочерний) продолжают выполняться параллельно.
- Различие в поведении: fork() возвращает 0 в дочернем процессе и PID дочернего процесса в родительском.

Пример

```cpp
#include <iostream>
#include <unistd.h>

int main() {
    pid_t pid = fork();

    if (pid == -1) {
        std::cerr << "Error creating process\n";
        return 1;
    } else if (pid == 0) {
        // Child process code
        std::cout << "This is a child process!\n";
    } else {
        // Parent process code
        std::cout << "This is the parent process. Child process PID:" << pid <<
     "\n";
    }

    return 0;
}
```

Межпроцессное взаимодействие (IPC)

Так как процессы изолированы друг от друга, для обмена данными между ними нужно использовать механизмы IPC. Популярные способы IPC:

- Pipe (каналы)
  - Используются для однонаправленного обмена данными между родительским и дочерним процессами.
- Message Queues (очереди сообщений)
  - Асинхронный способ передачи сообщений между процессами.
- Shared Memory (разделяемая память)
  - Позволяет процессам работать с общими данными напрямую.
- Signals (сигналы)
  - Используются для отправки уведомлений процессам.

Пример использования Pipe

```
1  #include <iostream>
2  #include <unistd.h>
3  #include <cstring>
4
5  int main() {
6      int pipefd[2];
7      pid_t pid;
8      char buffer[25];
9
10     if (pipe(pipefd) == -1) {
11         std::cerr << "Error creating channel\n";
12         return 1;
13     }
14
15     pid = fork();
16     if (pid == -1) {
17         std::cerr << "Error creating process\n";
18         return 1;
19     }
```

```
    if (pid == 0) {
        // Child process writes data
        close(pipefd[0]); // Closing the unused side for reading
        const char* msg = "Hello from child process!\n";
        write(pipefd[1], msg, strlen(msg));
        close(pipefd[1]);
    } else {
        // Parent process reads data
        close(pipefd[1]); // Closing the unused side for recording
        read(pipefd[0], buffer, sizeof(buffer));
        std::cout << "The parent received a message: " << buffer << "\n";
        close(pipefd[0]);
    }

    return 0;
}
```

Message Queues (Очереди сообщений)

Очереди сообщений — это механизм для асинхронной передачи данных между процессами. Процессы могут добавлять сообщения в очередь или извлекать их, когда это необходимо.

- ftok() — генерирует уникальный ключ для очереди сообщений на основе файла и идентификатора проекта.
- msgget() — создаёт очередь сообщений или получает доступ к существующей.
- msgsnd() — отправляет сообщение в очередь.
- msgrcv() — извлекает сообщение из очереди.
- msgctl() — управляет очередью сообщений, здесь используется для её удаления после завершения работы.

Пример использования Message Queues

```cpp
#include <iostream>
#include <unistd.h>
#include <cstring>
#include <sys/msg.h>
struct Message {
    long messageType;
    char messageText[100];
};
int main() {
    key_t key = ftok("queuefile", 65); // Create a unique key
    int msgid = msgget(key, 0666 | IPC_CREAT); // Create a message queue

    Message msg;
    msg.messageType = 1;
    if (fork() == 0) {
        // Child process: send message
        strcpy(msg.messageText, "Hello from child process!");
        msgsnd(msgid, &msg, sizeof(msg.messageText), 0);
        std::cout << "Message sent by child process\n";
    } else {
        // Parent process: receive message
        msgrcv(msgid, &msg, sizeof(msg.messageText), 1, 0);
        std::cout << "Message from child process: " << msg.messageText << std::
     endl;
        // Delete the message queue after receiving
        msgctl(msgid, IPC_RMID, nullptr);
    }
    return 0;
}
```

Shared Memory (Разделяемая память)

Разделяемая память позволяет нескольким процессам иметь доступ к одной и той же области памяти, что ускоряет обмен данными по сравнению с очередями сообщений и каналами. Однако нужно обеспечить синхронизацию между процессами, чтобы избежать конфликтов при доступе к данным.

- ftok() — генерирует ключ для разделяемой памяти.
- shmget() — создаёт сегмент разделяемой памяти.
- shmat() — присоединяет сегмент разделяемой памяти к адресу процесса.
- shmdt() — отсоединяет сегмент памяти после завершения работы.
- shmctl() — управляет сегментом разделяемой памяти, здесь используется для его удаления.

Пример

```cpp
#include <iostream>
#include <unistd.h>
#include <cstring>
#include <sys/shm.h>
int main() {
    key_t key = ftok("shmfile", 65); // Generate a key
    int shmid = shmget(key, 1024, 0666 | IPC_CREAT); // Create a shared memory segment

    if (fork() == 0) {
        // Child process: writes data to the shared memory
        char *str = (char*) shmat(shmid, nullptr, 0);
        strcpy(str, "Hello from the child process!");
        std::cout << "Child process wrote data to shared memory\n";
        shmdt(str); // Detach the shared memory
    } else {
        sleep(1); // Wait for the child process to write the data
        // Parent process: reads data from the shared memory
        char *str = (char*) shmat(shmid, nullptr, 0);
        std::cout << "Parent process read from shared memory: " << str << std::endl;
        shmdt(str); // Detach the shared memory

        // Remove the shared memory segment after the work is done
        shmctl(shmid, IPC_RMID, nullptr);
    }
    return 0;
}
```

Signals (Сигналы)

Сигналы — это механизм, с помощью которого один процесс может посылать другому процессу уведомления о различных событиях. Сигналы обычно используются для уведомления процесса о необходимости завершить выполнение, о поступлении данных или для выполнения специфичных обработок. Наиболее известные сигналы:

- SIGINT — сигнал завершения программы (обычно посылается при нажатии Ctrl+C).
- SIGKILL — сигнал принудительного завершения процесса.
- SIGTERM — сигнал запроса на завершение процесса

Пример

```cpp
#include <iostream>
#include <unistd.h>
#include <signal.h>

// Signal handler
void signalHandler(int signum) {
    std::cout << "Signal received: " << signum << std::endl;
    exit(signum);
}

int main() {
    // Set the signal handler for SIGINT
    signal(SIGINT, signalHandler);

    if (fork() == 0) {
        // Child process waits for a signal
        std::cout << "Child process is waiting for SIGINT (Ctrl+C)...\n";
        while (true) {
            // Infinite loop, waiting for signals
        }
    } else {
        sleep(5); // Wait for a while
        std::cout << "Parent process is sending SIGINT to the child process\n";
        kill(0, SIGINT); // Send the signal to the child process
    }

    return 0;
}
```

Ожидание завершения процессов: wait()

Родительский процесс может ожидать завершения дочернего процесса с помощью функции wait(), что позволяет избежать "зомби"процессов.

Процессзомби — это процесс, который завершился, но его родительский процесс не вызвал системный вызов wait() для получения информации о его завершении. Этот процесс находится в состоянии "terminated"(завершён), но ещё не был полностью удалён из таблицы процессов операционной системы, так как родитель не забрал информацию о его завершении.

Пример

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

int main() {
    pid_t pid = fork();

    if (pid == 0) {
        std::cout << "Child process: running...\n";
        sleep(2);
        std::cout << "Child process: finished\n";
        return 0;
    } else if (pid > 0) {
        std::cout << "Parent process: waiting for the child process to finish
      ...\n";
        wait(nullptr); // Waiting for the child process to finish
        std::cout << "Parent process: child process finished\n";
    }

    return 0;
}
```

Варианты завершения процессов

1. Дочерний процесс завершился первым, родитель остался
   - Дочерний процесс завершает свою работу, а родитель вызывает wait() для получения информации о его завершении. Как только родитель получает эту информацию, запись о дочернем процессе удаляется из таблицы процессов, и зомби не создаётся.
2. Родительский процесс завершился первым, дочерний остался
   - Когда родительский процесс умирает, операционная система назначает дочерний процесс инициализирующему процессу (обычно это процесс с PID = 1, такой, как init или systemd). Этот процесс является "усыновителем"всех осиротевших процессов и будет вызван для их завершения (включая вызов wait()). Такие дочерние процессы не станут зомби, так как их завершение обработает init.

Пример создания нескольких процессов

```cpp
#include <iostream>
#include <unistd.h>
#include <sys/wait.h>

void performTask(const std::string& taskName) {
    std::cout << "Process is performing task: " << taskName << "\n";
    sleep(2);
    std::cout << "Task " << taskName << " completed\n";
}

int main() {
    for (int i = 0; i < 3; ++i) {
        pid_t pid = fork();
        if (pid == 0) {
            // Child process performs its task
            performTask("Task " + std::to_string(i + 1));
            return 0;
        }
    }

    for (int i = 0; i < 3; ++i) {
        wait(nullptr); // Wait for all child processes to complete
    }

    std::cout << "All tasks are completed\n";
    return 0;
}
```

OpenMP

API для многоядерных и многопроцессорных систем, встроена в C/C++ и Fortran.

```
1  #include <omp.h>
2  #include <iostream>
3
4  int main() {
5      int sum = 0;
6      #pragma omp parallel for reduction(+:sum)
7      for (int i = 0; i < 100; ++i) {
8          sum += i;
9      }
10     std::cout << "Sum: " << sum << std::endl;
11     return 0;
12 }
```

Плюсы:

- Легко интегрируется в существующий код.
- Простой синтаксис.
- Поддержка мультиплатформенности.

Прагмы OpenMP

Прагмы OpenMP — это специальные директивы компилятора, которые указывают, как нужно распараллеливать определённые участки кода. Они начинаются с #pragma и предназначены для компиляторов, поддерживающих OpenMP. Каждая директива содержит ключевое слово OpenMP и описание того, что она должна делать.

Прагмы OpenMP

- #pragma omp parallel - базовая директива, которая запускает параллельный регион — участок кода, выполняемый несколькими потоками одновременно.
- #pragma omp for - директива используется для распараллеливания циклов. Потоки делят между собой итерации цикла и выполняют их параллельно.
- #pragma omp sections - директива позволяет распараллеливать разные независимые участки кода.
- #pragma omp single - директива указывает, что следующий участок кода должен быть выполнен только одним потоком.
- #pragma omp critical - директива гарантирует, что код внутри неё будет выполнен только одним потоком за раз, избегая конфликтов (например, при работе с общими данными).

Пример

```
#include <omp.h>
#include <iostream>
#include <vector>
int main() {
    const int N = 1000;
    std::vector<int> data(N);
    // Initialize the array with values
    for (int i = 0; i < N; ++i) {
        data[i] = i + 1;
    }
    int sum = 0;
    int max_val = 0;
    // Parallel section
    #pragma omp parallel
    {
        // Parallelize the loop for summing the elements
        #pragma omp for
        for (int i = 0; i < N; ++i) {
            #pragma omp critical
            {
                sum += data[i];
            }
        }
        // Synchronize threads before the next operations
        #pragma omp barrier
        // Split tasks into sections: finding the maximum and output the result
        #pragma omp sections
        {
```

```
            // Section 1: Find the maximum value in the array
            #pragma omp section {
                int local_max = 0;
                for (int i = 0; i < N; ++i) {
                    if (data[i] > local_max) {
                        local_max = data[i];
                    }
                }
                #pragma omp critical {
                    if (local_max > max_val) {
                        max_val = local_max;
                    }
                }
            }
            // Section 2: Output partial results
            #pragma omp section {
                std::cout << "Partial sum (after sum): " << sum << std::endl;
            }
        }
        // Final thread synchronization
        #pragma omp barrier
        // Compute and output the result after all threads have completed
        #pragma omp single {
            double average = static_cast<double>(sum) / N;
            std::cout << "Final sum: " << sum << std::endl;
            std::cout << "Average: " << average << std::endl;
            std::cout << "Max value: " << max_val << std::endl;
        }
    }
    return 0;
}
```

MPI (Message Passing Interface)

Стандарт для параллельного программирования с использованием передачи сообщений. Каждый процесс может исполняться на отдельной машине и общается с другими через сообщения. Плюсы:

- Эффективно на распределённых системах.
- Подходит для программирования на кластерах и суперкомпьютерах.
- Высокая производительность.

Минусы:

- Более сложное программирование по сравнению с OpenMP.
- Требует явного управления обменом сообщениями между процессами.

Основные функции MPI

- MPI_Init — инициализация MPI. Эта функция вызывается в начале программы.
- MPI_Finalize — завершение работы MPI. Эта функция вызывается в конце программы.
- MPI_Comm_size — возвращает общее количество процессов.
- MPI_Comm_rank — возвращает ранг (идентификатор) текущего процесса.
- MPI_Send и MPI_Recv — используются для передачи сообщений между процессами.
- MPI_Reduce — агрегирует данные от всех процессов и отправляет результат в указанный процесс (обычно процесс с рангом 0).
- MPI_Bcast — отправляет данные от одного процесса всем остальным процессам.

Пример

```cpp
#include <mpi.h>
#include <iostream>
#include <vector>

int main(int argc, char** argv) {
    // Initialize MPI
    MPI_Init(&argc, &argv);

    int world_size, world_rank;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);  // Get the number of processes
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  // Get the rank (identifier) of
      the current process

    const int N = 1000;  // Array size
    int chunk_size = N / world_size;  // Size of the array chunk for each
     process
    std::vector<int> data(N);

    // Initialize the array only in the main process
    if (world_rank == 0) {
        for (int i = 0; i < N; ++i) {
            data[i] = i + 1;
        }
    }
```

```cpp
    // Distribute data among processes
    std::vector<int> local_data(chunk_size);
    MPI_Scatter(data.data(), chunk_size, MPI_INT, local_data.data(), chunk_size,
      MPI_INT, 0, MPI_COMM_WORLD);

    // Local summing in each process
    int local_sum = 0;
    for (int i = 0; i < chunk_size; ++i) {
        local_sum += local_data[i];
    }

    // Collect sums from each process in the main process
    int global_sum = 0;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    // The main process calculates the average value
    if (world_rank == 0) {
        double average = static_cast<double>(global_sum) / N;
        std::cout << "Final sum: " << global_sum << std::endl;
        std::cout << "Average: " << average << std::endl;
    }

    // Finalize MPI
    MPI_Finalize();

    return 0;
}
```

Cheat Notes

Инициализация MPI: MPI_Init(&argc, &argv) — эта функция запускает MPI, готовя среду для взаимодействия процессов. Она всегда вызывается первой.
MPI_Comm_size(MPI_COMM_WORLD, &world_size) — функция возвращает количество процессов в мире (то есть общее количество активных процессов).
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank) — функция возвращает ранг (идентификатор) текущего процесса. Обычно главный процесс имеет ранг 0. Инициализация данных:

Главный процесс (процесс с рангом 0) инициализирует массив данных data размером 1000 элементов, которые он должен будет распределить между всеми процессами.

MPI_Scatter: Данная функция разбивает массив data на части и отправляет каждому процессу его подмассив для локальных вычислений. Процесс 0 передаёт подмассивы всем остальным процессам, включая себя.

Локальные вычисления: Каждый процесс выполняет локальное суммирование данных из своего подмассива (local_data).

MPI_Reduce: После того как каждый процесс посчитал свою локальную сумму, он объединяет все локальные суммы в глобальную сумму (global_sum). Процесс 0 получает результат агрегирования всех сумм. Вычисление среднего значения;

Главный процесс (процесс с рангом 0) вычисляет среднее значение массива, разделив глобальную сумму на количество элементов N. Завершение работы:

MPI_Finalize() завершает работу MPI. Все процессы завершаются.

Особенности использования MPI:

MPI хорошо подходит для распределённых систем, например, кластеры или сетевые системы, где каждый процесс может работать на отдельной машине. В отличие от OpenMP, MPI не использует общую память, а обменивается сообщениями между процессами, что делает его эффективным для программирования на кластерах и суперкомпьютерах. MPI даёт больше контроля над процессами и обменом данными, но требует большего объёма кода по сравнению с OpenMP.

Intel TBB (Threading Building Blocks)

Intel TBB — это библиотека, разработанная компанией Intel для упрощения параллельного программирования на уровне задач (task-based parallelism). TBB автоматически управляет распределением задач между потоками, предлагая высокоуровневую абстракцию для создания многопоточных программ.

Основные концепции TBB

- tbb::parallel_for — параллельное выполнение циклов. Подобно обычному циклу for, но итерации распределяются между потоками.
- tbb::task_scheduler_init — инициализирует пул потоков. Обычно используется для задания числа потоков.
- tbb::parallel_reduce — параллельное суммирование элементов или выполнение других операций с агрегированием данных.
- tbb::blocked_range — определяет диапазон индексов для параллельных вычислений.

Автоматическое управление потоками: TBB распределяет задачи между доступными потоками автоматически.

Пример

```
#include <tbb/tbb.h>
#include <iostream>
#include <vector>

class SumReducer {
public:
    int sum;

    SumReducer() : sum(0) {}
    SumReducer(SumReducer& s, tbb::split) { sum = 0; }

    void operator()(const tbb::blocked_range<int>& range) {
        int local_sum = sum;
        for (int i = range.begin(); i < range.end(); ++i) {
            local_sum += i + 1;
        }
        sum = local_sum;
    }

    void join(const SumReducer& rhs) {
        sum += rhs.sum;
    }
};
```

```cpp
int main() {
    const int N = 1000;  // Array size
    std::vector<int> data(N);

    // Initialize the array
    for (int i = 0; i < N; ++i) {
        data[i] = i + 1;
    }

    // Use tbb::parallel_reduce for parallel sum calculation
    SumReducer reducer;
    tbb::parallel_reduce(tbb::blocked_range<int>(0, N), reducer);

    // Calculate the average value
    double average = static_cast<double>(reducer.sum) / N;
    std::cout << "Final sum: " << reducer.sum << std::endl;
    std::cout << "Average: " << average << std::endl;

    return 0;
}
```

Cheat Notes

Класс SumReducer: Этот класс отвечает за параллельное суммирование. В нем есть переменная sum для хранения промежуточной суммы. Конструктор по умолчанию и конструктор копирования (split) используются для инициализации объектов для каждого потока.

Метод operator() выполняет параллельное суммирование для подмножества данных.

Метод join() объединяет результаты из разных потоков.

Инициализация массива: Мы инициализируем массив data значениями от 1 до 1000.

tbb::parallel_reduce: Эта функция разбивает массив данных на части и распределяет их между потоками. Каждый поток выполняет свою часть суммирования с помощью метода operator() класса SumReducer.

Когда все потоки завершат выполнение, метод join() объединяет результаты в один общий результат.

Вычисление среднего значения: После того как все потоки завершат выполнение и будет получена общая сумма, главный поток вычисляет среднее значение массива.

Преимущества использования TBB:

Автоматическое управление потоками: Пользователю не нужно явно создавать потоки или управлять ими. TBB автоматически распределяет задачи между потоками.

Эффективность: TBB может значительно повысить производительность за счёт того, что задачи равномерно распределяются между доступными ядрами процессора.

Параллельные алгоритмы: TBB предоставляет высокоуровневые параллельные алгоритмы, такие как parallel_for, parallel_reduce, которые делают код более лаконичным.

Плюсы и минусы TBB

Плюсы:

- Высокий уровень абстракции: Простота программирования задач с использованием параллельных алгоритмов.
- Хорошая производительность для многопоточных задач на одном компьютере.

Минусы:

- Зависимость от библиотеки: TBB требует установки и правильной настройки библиотеки. Она тесно связана с процессорами Intel, хотя работает и на других платформах.
- Лучше подходит для задач, работающих с многопоточностью, чем для распределённых систем: В отличие от MPI, TBB не поддерживает работу с распределёнными системами и кластерами.

До следующей лекции!