



Асинхронность и Многопоточность (2)

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

```
1 #include <iostream>
2 #include <unistd.h>
3
4 int main() {
5     pid_t pid = fork();
6
7     if (pid == -1) {
8         std::cerr << "Error creating process\n";
9         return 1;
10    } else if (pid == 0) {
11        // Child process code
12        std::cout << "This is a child process!\n";
13    } else {
14        // Parent process code
15        std::cout << "This is the parent process. Child process PID:" << pid <<
            "\n";
16    }
17
18    return 0;
19 }
```

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    }
```

```
1  if (pid == 0) {
2      // Child process writes data
3      close(pipefd[0]); // Closing the unused side for reading
4      const char* msg = "Hello from child process!\n";
5      write(pipefd[1], msg, strlen(msg));
6      close(pipefd[1]);
7  } else {
8      // Parent process reads data
9      close(pipefd[1]); // Closing the unused side for recording
10     read(pipefd[0], buffer, sizeof(buffer));
11     std::cout << "The parent received a message: " << buffer << "\n";
12     close(pipefd[0]);
13 }
14
15 return 0;
16 }
```

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

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <cstring>
4 #include <sys/msg.h>
5 struct Message {
6     long messageType;
7     char messageText[100];
8 };
9 int main() {
10     key_t key = ftok("queuefile", 65); // Create a unique key
11     int msgid = msgget(key, 0666 | IPC_CREAT); // Create a message queue
12
13     Message msg;
14     msg.messageType = 1;
15     if (fork() == 0) {
16         // Child process: send message
17         strcpy(msg.messageText, "Hello from child process!");
18         msgsnd(msgid, &msg, sizeof(msg.messageText), 0);
19         std::cout << "Message sent by child process\n";
20     } else {
21         // Parent process: receive message
22         msgrcv(msgid, &msg, sizeof(msg.messageText), 1, 0);
23         std::cout << "Message from child process: " << msg.messageText << std::endl;
24         // Delete the message queue after receiving
25         msgctl(msgid, IPC_RMID, nullptr);
26     }
27     return 0;
28 }
```

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

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <cstring>
4 #include <sys/shm.h>
5 int main() {
6     key_t key = ftok("shmfile", 65); // Generate a key
7     int shmid = shmget(key, 1024, 0666 | IPC_CREAT); // Create a shared memory
        segment
8
9     if (fork() == 0) {
10         // Child process: writes data to the shared memory
11         char *str = (char*) shmat(shmid, nullptr, 0);
12         strcpy(str, "Hello from the child process!");
13         std::cout << "Child process wrote data to shared memory\n";
14         shmdt(str); // Detach the shared memory
15     } else {
16         sleep(1); // Wait for the child process to write the data
17         // Parent process: reads data from the shared memory
18         char *str = (char*) shmat(shmid, nullptr, 0);
19         std::cout << "Parent process read from shared memory: " << str << std::
endl;
20         shmdt(str); // Detach the shared memory
21
22         // Remove the shared memory segment after the work is done
23         shmctl(shmid, IPC_RMID, nullptr);
24     }
25     return 0;
26 }
```

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

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <signal.h>
4
5 // Signal handler
6 void signalHandler(int signum) {
7     std::cout << "Signal received: " << signum << std::endl;
8     exit(signum);
9 }
10
11 int main() {
12     // Set the signal handler for SIGINT
13     signal(SIGINT, signalHandler);
14
15     if (fork() == 0) {
16         // Child process waits for a signal
17         std::cout << "Child process is waiting for SIGINT (Ctrl+C)...\n";
18         while (true) {
19             // Infinite loop, waiting for signals
20         }
21     } else {
22         sleep(5); // Wait for a while
23         std::cout << "Parent process is sending SIGINT to the child process\n";
24         kill(0, SIGINT); // Send the signal to the child process
25     }
26
27     return 0;
28 }
```

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

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <sys/wait.h>
4
5 int main() {
6     pid_t pid = fork();
7
8     if (pid == 0) {
9         std::cout << "Child process: running...\n";
10        sleep(2);
11        std::cout << "Child process: finished\n";
12        return 0;
13    } else if (pid > 0) {
14        std::cout << "Parent process: waiting for the child process to finish
15        ... \n";
16        wait(nullptr); // Waiting for the child process to finish
17        std::cout << "Parent process: child process finished\n";
18    }
19
20    return 0;
21 }
```

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

```
1 #include <iostream>
2 #include <unistd.h>
3 #include <sys/wait.h>
4
5 void performTask(const std::string& taskName) {
6     std::cout << "Process is performing task: " << taskName << "\n";
7     sleep(2);
8     std::cout << "Task " << taskName << " completed\n";
9 }
10
11 int main() {
12     for (int i = 0; i < 3; ++i) {
13         pid_t pid = fork();
14         if (pid == 0) {
15             // Child process performs its task
16             performTask("Task " + std::to_string(i + 1));
17             return 0;
18         }
19     }
20
21     for (int i = 0; i < 3; ++i) {
22         wait(nullptr); // Wait for all child processes to complete
23     }
24
25     std::cout << "All tasks are completed\n";
26     return 0;
27 }
```

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 — 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.

- `#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

```
1 #include <omp.h>
2 #include <iostream>
3 #include <vector>
4 int main() {
5     const int N = 1000;
6     std::vector<int> data(N);
7     // Initialize the array with values
8     for (int i = 0; i < N; ++i) {
9         data[i] = i + 1;
10    }
11    int sum = 0;
12    int max_val = 0;
13    // Parallel section
14    #pragma omp parallel
15    {
16        // Parallelize the loop for summing the elements
17        #pragma omp for
18        for (int i = 0; i < N; ++i) {
19            #pragma omp critical
20            {
21                sum += data[i];
22            }
23        }
24        // Synchronize threads before the next operations
25        #pragma omp barrier
26        // Split tasks into sections: finding the maximum and output the result
27        #pragma omp sections
28        {
```

```

1      // Section 1: Find the maximum value in the array
2      #pragma omp section {
3          int local_max = 0;
4          for (int i = 0; i < N; ++i) {
5              if (data[i] > local_max) {
6                  local_max = data[i];
7              }
8          }
9          #pragma omp critical {
10             if (local_max > max_val) {
11                 max_val = local_max;
12             }
13         }
14     }
15     // Section 2: Output partial results
16     #pragma omp section {
17         std::cout << "Partial sum (after sum): " << sum << std::endl;
18     }
19 }
20 // Final thread synchronization
21 #pragma omp barrier
22 // Compute and output the result after all threads have completed
23 #pragma omp single {
24     double average = static_cast<double>(sum) / N;
25     std::cout << "Final sum: " << sum << std::endl;
26     std::cout << "Average: " << average << std::endl;
27     std::cout << "Max value: " << max_val << std::endl;
28 }
29 }
30 return 0;
31 }

```

MPI (Message Passing Interface)

Стандарт для параллельного программирования с использованием передачи сообщений. Каждый процесс может выполняться 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.
- Требуется явное управление 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

```
1 #include <mpi.h>
2 #include <iostream>
3 #include <vector>
4
5 int main(int argc, char** argv) {
6     // Initialize MPI
7     MPI_Init(&argc, &argv);
8
9     int world_size, world_rank;
10    MPI_Comm_size(MPI_COMM_WORLD, &world_size); // Get the number of processes
11    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank); // Get the rank (identifier) of
        the current process
12
13    const int N = 1000; // Array size
14    int chunk_size = N / world_size; // Size of the array chunk for each
        process
15    std::vector<int> data(N);
16
17    // Initialize the array only in the main process
18    if (world_rank == 0) {
19        for (int i = 0; i < N; ++i) {
20            data[i] = i + 1;
21        }
22    }
```

```

1 // Distribute data among processes
2 std::vector<int> local_data(chunk_size);
3 MPI_Scatter(data.data(), chunk_size, MPI_INT, local_data.data(), chunk_size,
4             MPI_INT, 0, MPI_COMM_WORLD);
5
6 // Local summing in each process
7 int local_sum = 0;
8 for (int i = 0; i < chunk_size; ++i) {
9     local_sum += local_data[i];
10 }
11
12 // Collect sums from each process in the main process
13 int global_sum = 0;
14 MPI_Reduce(&local_sum, &global_sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
15
16 // The main process calculates the average value
17 if (world_rank == 0) {
18     double average = static_cast<double>(global_sum) / N;
19     std::cout << "Final sum: " << global_sum << std::endl;
20     std::cout << "Average: " << average << std::endl;
21 }
22
23 // Finalize MPI
24 MPI_Finalize();
25
26 return 0;
27 }

```

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.

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 — 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.

- `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

```
1 #include <tbb/tbb.h>
2 #include <iostream>
3 #include <vector>
4
5 class SumReducer {
6 public:
7     int sum;
8
9     SumReducer() : sum(0) {}
10    SumReducer(SumReducer& s, tbb::split) { sum = 0; }
11
12    void operator()(const tbb::blocked_range<int>& range) {
13        int local_sum = sum;
14        for (int i = range.begin(); i < range.end(); ++i) {
15            local_sum += i + 1;
16        }
17        sum = local_sum;
18    }
19
20    void join(const SumReducer& rhs) {
21        sum += rhs.sum;
22    }
23 };
```



```

1  int main() {
2      const int N = 1000;  // Array size
3      std::vector<int> data(N);
4
5      // Initialize the array
6      for (int i = 0; i < N; ++i) {
7          data[i] = i + 1;
8      }
9
10     // Use tbb::parallel_reduce for parallel sum calculation
11     SumReducer reducer;
12     tbb::parallel_reduce(tbb::blocked_range<int>(0, N), reducer);
13
14     // Calculate the average value
15     double average = static_cast<double>(reducer.sum) / N;
16     std::cout << "Final sum: " << reducer.sum << std::endl;
17     std::cout << "Average: " << average << std::endl;
18
19     return 0;
20 }

```

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 TBV:

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

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

Параллельные алгоритмы: TBV 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!