



Асинхронность и Многопоточность

Лекторы:

Аспирант МФТИ, Шер Артём Владимирович

Аспирант МФТИ, Зингеренко Михаил Владимирович

17 сентября 2024

Лямбда функция

Лямбда-функция — это анонимная функция, определяемая прямо в месте использования. Используется для передачи небольших функций как аргументов в другие функции (например, в алгоритмы).

[захват переменных из области видимости] (параметры)

спецификаторы — > возвращаемое_значение

```
{  
тело_функции;  
};
```

Лямбда код

```
1 auto sum = [](int a, int b) throw() -> int {  
2     return a + b;  
3 };  
4  
5 std::cout << sum(3, 5); // Prints: 8
```

Захват переменных из внешнего контекста:

- [=] — захват по значению.
- [&] — захват по ссылке.
- [x, &y] — захват конкретных переменных.

Удобны для коротких операций, таких как сортировка или фильтрация.

В чем разница

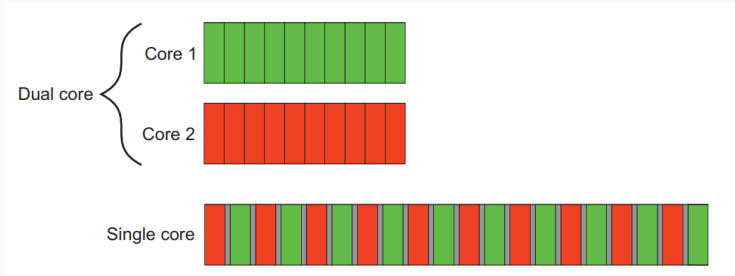
- Асинхронность: когда задачи запускаются, выполняются и завершаются в перекрывающиеся временные промежутки.
- Параллелизм: когда две или более задачи выполняются одновременно.

Зачем?

Улучшение производительности.

Снижение времени ожидания пользователя в программах.

Увеличение полезной нагрузки на CPU/GPU

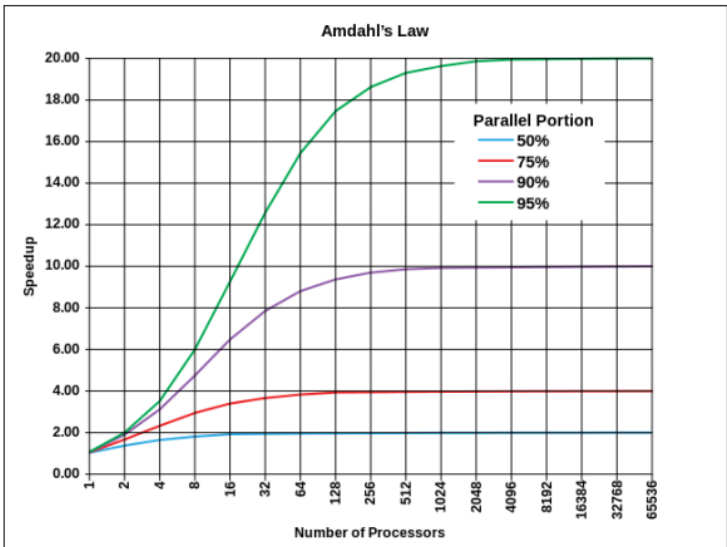


Два подхода к асинхронности: параллельное выполнение на двухъядерном процессоре и переключением задач на одноядерной машине

$$S = \frac{1}{(1 - P) + \frac{P}{N}}$$

- S – ускорение при использовании N процессоров.
- P – доля программы, которая может быть распараллелена.
- $1 - P$ – доля программы, которая остаётся последовательной.

Закон Амдала

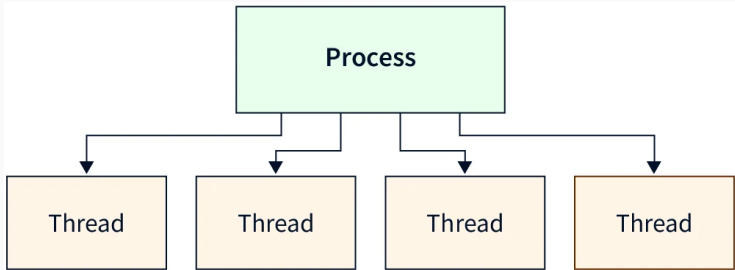


Процессы

Процесс в Linux — это экземпляр запущенной программы в операционной системе Linux. Он представляет собой независимую единицу исполнения, работающую в собственном пространстве памяти и имеющую собственный набор системных ресурсов. Каждому процессу присваивается уникальный идентификатор процесса (PID), который отличает его от других процессов.

PID	USER	PRI	NI	VIRT	RES	SHR	S	CPU%	MEM%	TIME+	Command
1177854	artem	20	0	14068	7552	3456	R	2.0	0.0	0:00.16	htop
1230	root	20	0	473M	6268	4732	S	0.7	0.0	11:54.08	/usr/sbin/NetworkManager --no-daemon
1891	artem	20	0	6698M	198M	143M	S	0.7	0.3	1h12:56	/usr/lib/xorg/Xorg vt2 -displayfd 3
1	root	20	0	24128	8404	3172	S	0.0	0.0	1:45.99	/sbin/init splash
470	root	19	-1	198M	24192	23808	S	0.0	0.0	0:13.95	/usr/lib/systemd/systemd-journald
571	root	20	0	38628	3916	972	S	0.0	0.0	0:03.03	/usr/lib/systemd/systemd-udev
1081	systemd-oo	20	0	17556	2944	2816	S	0.0	0.0	4:51.40	/usr/lib/systemd/systemd-oond
1082	systemd-re	20	0	22696	5376	3840	S	0.0	0.0	0:56.86	/usr/lib/systemd/systemd-resolved
1083	systemd-ti	20	0	91044	2304	2176	S	0.0	0.0	0:04.95	/usr/lib/systemd/systemd-timesyncd
1156	systemd-ti	20	0	91044	2304	2176	S	0.0	0.0	0:00.00	/usr/lib/systemd/systemd-timesyncd
1161	root	20	0	307M	2280	872	S	0.0	0.0	0:00.30	/usr/libexec/accounts-daemon
1164	avahi	20	0	9876	2048	1024	S	0.0	0.0	6:34.16	avahi-daemon: running [se0110-721-5
1165	root	20	0	13672	512	256	S	0.0	0.0	0:00.07	/usr/libexec/bluetooth/bluetoothd
1166	root	20	0	9428	1280	1280	S	0.0	0.0	0:03.66	/usr/sbin/cron -f -P
1167	messagebus	20	0	12368	4480	2176	S	0.0	0.0	1:08.79	@dbus-daemon --system --address=system
1170	gnome-remo	20	0	428M	452	452	S	0.0	0.0	0:00.02	/usr/libexec/gnome-remote-desktop-d
1173	polkitd	20	0	382M	8036	3268	S	0.0	0.0	0:01.23	/usr/lib/polkit-1/polkitd --no-debug
1176	root	20	0	306M	820	564	S	0.0	0.0	0:00.04	/usr/libexec/power-profiles-daemon
1186	root	20	0	2812M	22356	7424	S	0.0	0.0	0:00.02	/usr/lib/snapd/snapd

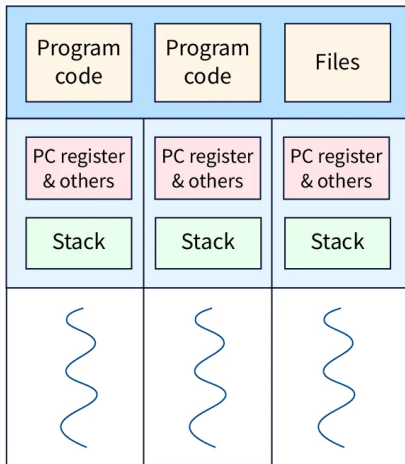
Треды и Процессы



Треды — это легкая единица выполнения внутри процесса, которая может работать независимо. Это позволяет одновременно выполнять задачи, обеспечивая параллельную обработку и эффективное использование ресурсов.

Структура тредов

Внутренняя структура потока состоит из трех основных компонентов: стека, набора регистров и данных, специфичных для потока.



Структура тредов

- стек — это область памяти, выделенная для выполнения потока. Он содержит локальные переменные потока, вызовы функций и адреса возврата. Каждый поток имеет свой стек, что позволяет выполнять его независимо.
- Набор регистров содержит контекст выполнения потока, включая значения регистров ЦП. В этих регистрах хранится важная информация, такая как счетчики программ, указатели стека и другие регистры, специфичные для процессора.
- Данные, специфичные для потока, позволяют каждому потоку сохранять свое уникальное состояние. Он может включать переменные, специфичные для потока, такие как идентификатор потока или хранилище для конкретного потока.

Создание тредов в C++

```
1 void task1() { /* executing the first task */ }
2 void task2() { /* executing the first task */ }
3
4 int main() {
5     std::thread t1(task1);
6     std::thread t2(task2);
7
8     t1.join();
9     t2.join();
10
11     return 0;
12 }
```

- Создание: Поток создается с использованием конструктора `std::thread`, передавая в него функцию или лямбда-выражение.
- Запуск: Поток начинает выполнять свою задачу сразу после создания.
- Ожидание завершения (`join`) или отделение (`detach`): Управление завершением потока.
- Завершение: Поток завершается, когда его функция завершает выполнение.

Асинхронные операции

```
1 #include <iostream>
2 #include <future>
3
4 int computeSquare(int x) {
5     return x * x;
6 }
7
8 int main() {
9     std::future<int> result = std::async(std::launch::async, computeSquare, 10);
10
11     std::cout << "Calculating..." << std::endl;
12     std::cout << "Result: " << result.get() << std::endl;
13
14     return 0;
15 }
```

- `std::launch::async`: задача запускается в отдельном потоке.
- `std::launch::deferred`: задача запускается только при вызове метода `get()` или `wait()`.

```
1 std::async(std::launch::deferred, computeSquare, 10); // Will be executed when .  
   get() is called  
2 std::async(std::launch::async, computeSquare, 10);    // Execute immediately in  
   a separate thread
```

`std::packaged_task` – это обёртка для функции или callable-объекта, позволяющая выполнить его асинхронно. Создает объект `std::future`, который используется для получения результата выполнения задачи. Полезен для работы с задачами в многопоточном окружении.

Зачем использовать `std::packaged_task`?

- Превращает любую функцию в асинхронную задачу.
- Позволяет передавать задачи в потоки, не теряя возможность управлять результатом.
- Подходит для использования с `std::thread` или другими пулами потоков.

Packaged task Пример

```
1 #include <iostream>
2 #include <future>
3 #include <thread>
4
5 int calculateSquare(int x) {
6     return x * x;
7 }
8
9 int main() {
10     std::packaged_task<int(int)> task(calculateSquare);
11     std::future<int> result = task.get_future();
12
13     std::thread t(std::move(task), 5);
14     t.join();
15
16     std::cout << "Square: " << result.get() << std::endl;
17     return 0;
18 }
```

`std::packaged_task` и `std::future` работают в связке:

- `std::packaged_task` отвечает за выполнение задачи.
- `std::future` позволяет получить результат выполнения.

Можно использовать методы `get()`, `wait()`, `wait_for()` для получения результата.

Асинхронные операции с использованием `std::future` и `std::promise`

- `std::promise` позволяет явно установить результат для `std::future`
- Используется, когда результат задачи должен быть установлен вручную.

```
1 #include <iostream>
2 #include <thread>
3 #include <future>
4
5 void setValue(std::promise<int> p) {
6     p.set_value(42); // Setting a value for future
7 }
8
9 int main() {
10     std::promise<int> p;
11     std::future<int> f = p.get_future();
12
13     std::thread t(setValue, std::move(p));
14     std::cout << "Result from promise: " << f.get() << std::endl;
15
16     t.join();
17     return 0;
18 }
```

- Доступ к общим данным: Когда несколько потоков одновременно обращаются к одной и той же переменной или ресурсу без должной синхронизации, это может привести к состояниям гонок (data race).

Примеры

```
1 int counter = 0;
2
3 void increment() {
4     for (int i = 0; i < 1000; ++i) {
5         ++counter; // Unsynchronized access
6     }
7 }
8
9 int main() {
10     std::thread t1(increment);
11     std::thread t2(increment);
12
13     t1.join();
14     t2.join();
15
16     std::cout << "Counter: " << counter << std::endl; // Result may be invalid
17     return 0;
18 }
```

Бороться можно с помощью примитивов синхронизации.

Что такое примитивы синхронизации низкого уровня?

Примитивы синхронизации — это базовые конструкции, такие как:

- Мьютекс (Mutex)
- Атомарные операции (Atomic)
- Семафоры (Semaphore)
- Барьеры памяти (Memory Fence)
- Условные переменные (Condition variable)

Мьютексы

```
1 #include <iostream>
2 #include <thread>
3 #include <mutex>
4
5 int counter = 0;
6 std::mutex mtx;
7
8 void increment() {
9     for (int i = 0; i < 1000; ++i) {
10         std::lock_guard<std::mutex> lock(mtx);
11         ++counter;
12     }
13 }
14
15 int main() {
16     std::thread t1(increment);
17     std::thread t2(increment);
18
19     t1.join();
20     t2.join();
21
22     std::cout << "Counter: " << counter << std::endl;
23     return 0;
24 }
```

В этом примере используется `std::lock_guard`, который автоматически блокирует мьютекс при создании и разблокирует его при выходе из области видимости. Альтернатива `std::unique_lock` - более гибкий, позволяет вручную блокировать и разблокировать мьютекс.

```
1 std::unique_lock<std::mutex> lock(mtx);  
2 // access to protected data  
3 lock.unlock(); // Manual unlock
```

`std::atomic` — это шаблонный класс, предоставляющий атомарные операции с переменными. Гарантирует, что операции над переменной выполняются без гонок данных и не требуют использования мьютексов. Подходит для использования в многопоточных программах, где важно быстро и безопасно изменять значение переменной.

- Атомарные операции:

- `++var` — атомарное увеличение.
- `--var` — атомарное уменьшение.
- `load()` и `store()` — чтение и запись значений.

- Методы `fetch`:

- `fetch_add()`, `fetch_sub()`, `exchange()` — более сложные операции.

Пример на атомиках

```
1  #include <iostream>
2  #include <atomic>
3  #include <thread>
4
5  std::atomic<int> counter(0);
6
7  void increment() {
8      for (int i = 0; i < 1000; ++i) {
9          ++counter;  // Atomic increment
10     }
11 }
12
13 int main() {
14     std::thread t1(increment);
15     std::thread t2(increment);
16
17     t1.join();
18     t2.join();
19
20     std::cout << "Counter: " << counter << std::endl;
21
22     counter.fetch_add(10);  // Atomic add 10
23     int value = counter.load();  // Atomic read of variable
24     return 0;
25 }
```

Семафоры

Семафор позволяет управлять доступом к ресурсу с определенным количеством доступных слотов.

```
1 #include <iostream>
2 #include <thread>
3 #include <semaphore>
4
5 std::counting_semaphore<2> sem(2); // Two threads can access simultaneously
6
7 void criticalSection() {
8     sem.acquire(); // Semaphore capture
9     std::cout << "Thread in critical section\n";
10    std::this_thread::sleep_for(std::chrono::seconds(1));
11    sem.release(); // Releasing a semaphore
12 }
13
14 int main() {
15     std::thread t1(criticalSection);
16     std::thread t2(criticalSection);
17     std::thread t3(criticalSection);
18
19     t1.join();
20     t2.join();
21     t3.join();
22
23     return 0;
24 }
```

Conditional variables

`std::condition_variable` — это механизм синхронизации, который позволяет потокам ожидать наступления определенного события. Используется совместно с мьютексами для организации ожидания и уведомления потоков. Позволяет одному потоку 'усыпить' себя до тех пор, пока другой поток не 'разбудит' его.

- Управляет потоками, ожидающими доступ к ресурсу или наступления условия.
- Позволяет сократить использование активного ожидания (busy-waiting).
- Удобен для реализации паттернов 'производитель-потребитель', задач очередей и синхронизации многопоточных программ.

Conditional variables пример

```
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 ready = false;
9
10 void worker() {
11     std::unique_lock<std::mutex> lock(mtx);
12     std::cout << "Lock mutex is proceeding...\n";
13     cv.wait(lock, [] { return ready; }); // Waiting, until ready does not true
14     std::cout << "Worker is proceeding...\n";
15 }
16
17 void setReady() {
18     std::lock_guard<std::mutex> lock(mtx);
19     std::cout << "Ready is proceeding...\n";
20     ready = true;
21     cv.notify_one(); // Notification one waiting thread
22 }
23
24 int main() {
25     std::thread t1(worker);
26     std::this_thread::sleep_for(std::chrono::seconds(1));
27     setReady();
28     t1.join();
29     return 0;
30 }
```

Основные методы

- `wait()`: переводит поток в состояние ожидания.
 - `wait(lock)` — ожидает до вызова `notify_one()` или `notify_all()`.
 - `wait(lock, predicate)` — ожидает до тех пор, пока `predicate` не вернет `true`.
- `notify_one()`: пробуждает один из ожидающих потоков.
- `notify_all()`: пробуждает все ожидающие потоки.

Дедлок возникает, когда два или более потока навсегда блокируются, ожидая освобождения ресурсов, которые захвачены друг другом. Условия возникновения дедлока:

- Взаимное исключение: один поток удерживает ресурс и не отпускает его.
- Удержание и ожидание: поток удерживает один ресурс и ожидает освобождения другого.
- Отсутствие принудительного освобождения: ресурс не может быть принудительно освобожден.
- Циклическое ожидание: существует цепочка потоков, каждый из которых удерживает ресурс, ожидая другого.

Дедлок

```
1 #include <iostream>
2 #include <thread>
3 #include <mutex>
4
5 std::mutex mtx1, mtx2;
6
7 void task1() {
8     std::lock_guard<std::mutex> lock1(mtx1);
9     std::this_thread::sleep_for(std::chrono::milliseconds(100));
10    std::lock_guard<std::mutex> lock2(mtx2);
11    std::cout << "Task 1 completed\n";
12 }
13
14 void task2() {
15     std::lock_guard<std::mutex> lock2(mtx2);
16     std::this_thread::sleep_for(std::chrono::milliseconds(100));
17     std::lock_guard<std::mutex> lock1(mtx1);
18     std::cout << "Task 2 completed\n";
19 }
20
21 int main() {
22     std::thread t1(task1);
23     std::thread t2(task2);
24
25     t1.join();
26     t2.join();
27
28     return 0;
29 }
```

До следующей лекции!