Задача 1 — Обедающие философы

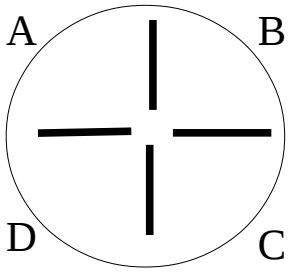
Требуется реализовать описанную модель в виде многопоточной программы, в которой каждый философ представлен отдельным потоком и которая удовлетворяет следующим требованиям:

- Ни один из философов не голодает (будет вечно чередовать приемы пищи и размышления при бесконечном выполнении программы);
- Ресурсы равномерно распределяются между философами (при одинаковом поведении философы едят примерно одинаковое количество раз, никто из них не получает преимущества);
- Одновременно могут есть несколько философов;
- Реализация масштабируется (количество приемов пищи не падает быстро) при увеличении количества философов.

При решении данной задачи будем использовать схему с официантом. Пусть кол-во официантов равно N. Для того, чтобы философы не голодали, требуется устранить возможность взаимной блокировки (ситуация, когда каждый философ взял левую вилку и ждет, когда освободится правя, и так по кругу). Для этого введем дополнительную сущность — официант, который будет контролировать кол-во философов, находящихся за столом, не допуская их всех одновременно за стол (больше, чем N-1), а как только один из философов заканчивает прием пищи, сразу же заходит ожидающий философ. Это, очевидно, устранит возможность взаимной блокировки. При этом кол-во философов, принимающих пищу может быть больше одного. Также в данной ситуации все философы будут есть примерно одинаковое кол-во раз, т. к. любой философ, ожидающий входа в столовую, войдет в нее сразу же, как выйдет другой (т. е. не будет бесконечно пропускать свою очередь), а философ, уже взявший одну вилку, возьмет и вторую, как только она освободится, (также не будет бесконечно пропускать свою очередь).

Запустим программу. Результаты представлены в таблице 1 (реализация 1).

Но данная реализация обладает недостатком, представим ситуацию с 4 философами:



допустим A взял две вилки, B возьмет левую вилку, C возьмет левую вилку, D будет в состоянии ожидания входа в столовую. В данном случае B будет блокировать C, пока не получит правую вилку от A. В итоге получаем, что из потенциально возможных двух философов, принимающих пищу (A и C) получаем одного (A). Чтобы избежать подобной ситуации можно предложить схему, в которой философ (в нашем случае B) не будет держать левую вилку до получения правой, а будет класть ее на определенное время и повторять попытку вновь через промежуток времени, определяемый алгоритмом «exponential backoff» (экспоненциальная задержка). В этом случае философ B положит левую вилку, которую, возможно (время задержки случайно), возьмет философ C, и одновременно 2 философа будут принимать пищу из 2 потенциально возможных. Посмотрим, как данное улучнение скажется на результате. Получаем следующие результаты (таблица 1 реализация 2). Для 5 философов прирост составил примерно 8%, для 25 и 100 около 40%.

Таблица 1. Результаты работы программы

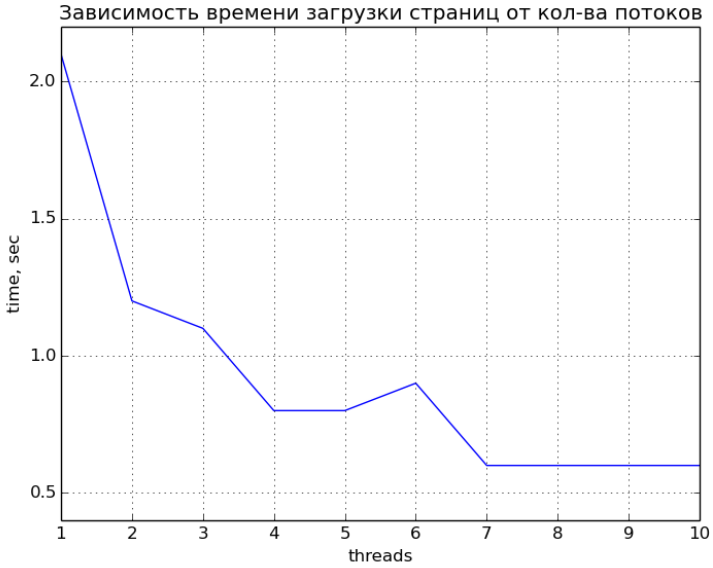
Параметры	Реализация 1	Реализация 2
./phil 5 20 100 100 0	[1] 128 7385 [2] 127 7212 [3] 129 7580 [4] 124 7151 [5] 136 7409 среднее: 128 вариация: 3.5%	[1] 131 6949 [2] 142 5965 [3] 141 6544 [4] 142 6590 [5] 134 5298 среднее: 138 вариация: 3.7%
./phil 25 20 100 100 0	[1] 107 10120 [2] 103 10163 [3] 104 10007 [4] 98 10389 [5] 97 10750 [6] 94 10525 [7] 91 10474 [8] 89 11003 [9] 89 10609 [10] 93 10407 [11] 94 10779 [12] 90 10514 [13] 90 10265 [14] 97 10877 [15] 95 10356 [16] 96 10640 [17] 100 10503 [18] 97 10701 [19] 94 10507 [20] 97 9961 [21] 95 10106 [22] 105 10108 [23] 104 10127 [24] 102 9604 [25] 103 9631 среднее 97 вариация: 5.5%	[1] 148 5953 [2] 129 7124 [3] 149 5505 [4] 140 5766 [5] 136 5910 [6] 140 5646 [7] 136 7127 [8] 147 5487 [9] 139 6165 [10] 136 6567 [11] 144 5433 [12] 135 6716 [13] 141 5435 [14] 131 7274 [15] 142 5236 [16] 137 6661 [17] 139 5784 [18] 132 6784 [19] 138 5873 [20] 138 6551 [21] 143 6019 [22] 138 6273 [23] 133 6479 [24] 138 6506 [25] 144 5723 среднее 138 вариация: 3.7%
./phil 100 20 100 100 0	… среднее: 94 вариация: 4.5%	… среднее: 140 вариация: 3.8%

Задача 2 — Поисковый робот

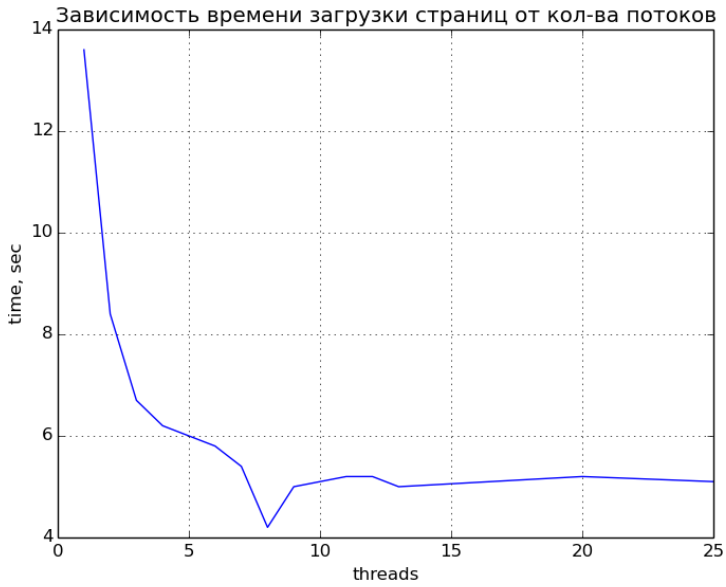
Для реализации поискового робота необходимо создать потокобезопасную очередь (concurrent_queue) для реализации очереди задач и потокобезопасное множество (concurrent_set) для сохранения и проверки посещенных страниц, оба представлены в файле «concurrent_structs.h». Также для удобной работы с потоками был создан пул потоков, представленный в файле «thread_pool.h». Метод loop выполняется в каждом потоке пула пока в очереди есть задачи (если задач нет, то поток ждет их появления на условной переменной) и не взведен флаг stop. Программа работает следующим образом: создается пул с необходимым количеством потоков, в него добавляется первая задача (загрузка начальной страницы). Поток загружает страницу, парсит ее, находит вложенные ссылки, и каждую из них добавляет в пул как новую задачу. Каждый поток возвращает результат работы в виде объекта future и кладет их в очередь выполненных задач. Главный поток дожидается получения результата от всех объектов future, выводит результат выполнения каждого из потоков (страница успешно загружена и сохранена или сообщение об ошибке), выставляет флаг stop у пула потоков и вызывает метод join для каждого из потоков и завершает выполнение.

Результат работы программы: time ./crawler http://www.yandex.ru P D ./pages/ N 10

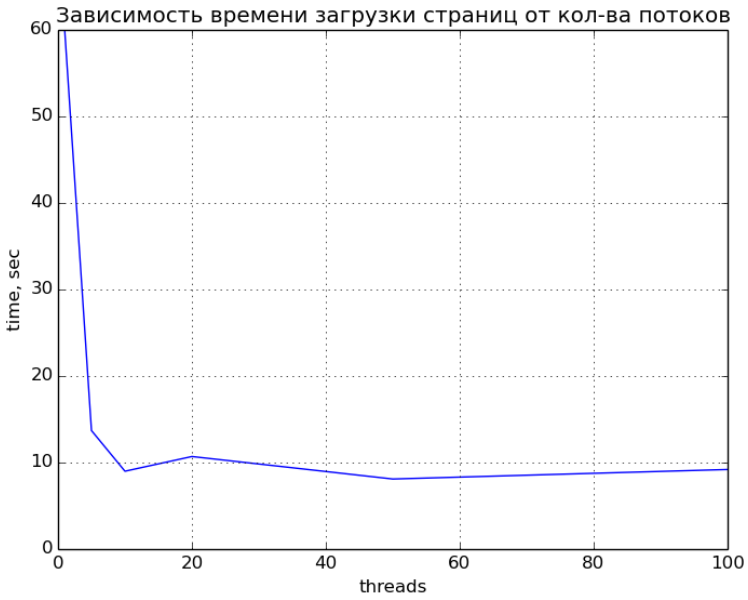
P = 10 страниц	
N, кол-во потоков	время, сек
1	2.1
2	1.2
3	1.0
4	1.0
5	0.8
6	0.8
7	0.9
8	0.6
9	0.6
10	0.6



P = 25 страниц	
N, кол-во потоков	время, сек
1	13.6
2	8.4
3	6.7
4	6.2
5	6.0
6	5.8
7	5.4
8	4.2
9	5.0
10	5.1
11	5.2
12	5.2
13	5.0
20	5.2
25	5.1



P = 100 страниц	
N, кол-во потоков	время, сек
1	60.0
5	13.7
10	9.0
20	10.7
50	8.1
100	9.2



Можно заметить, что после определенного кол-ва потоков в пуле суммарное время выполнения всех задач не уменьшается, а остается постоянной или даже возрастает. Скорее всего это связано с тем, что при возрастании кол-ва потоков возрастает время на синхронизацию, а также задачи (страницы на загрузку) не успевают генерироваться с нужной скоростью, и большинство потоков просто находятся в состоянии ожидания, не выполняя полезной работы.