Московский Авиационный Институт

(Национальный Исследовательский Университет)

Факультет информационных технологий и прикладной математики

Кафедра вычислительной математики и программирования

Лабораторная работа №3 по курсу

«Операционные системы»

Студентка:

Варламова Анна Борисовна

Группа: М80-207Б-20

Преподаватель: Миронов Е.С.

Оценка: _____

Дата: 13.12.2021

Москва, 2021

Постановка задачи

Составить и отладить программу на языке C++, обрабатывающую данные в многопоточном режиме. Использовать стандартные средства создания потоков операционной системы Linix. Предусмотреть возможность ограничить максимальное количество потоков, используемых в программе.

Вариант 4:

Отсортировать массив целых чисел при помощи TimSort.

Метод решения

Используются следующие системные вызовы:

1. **thread() –** создает новый поток выполнения в программе.
2. **thread.join()** – *для блокировки потока.*

Краткий алгоритм решения поставленной задачи:

1. Вводится максимальное количество потоков, считываются числа
2. Если длина вектора меньше 64, то он сортируется вставкой
3. Если длина вектора больше 64, выполняется функция TimSort:

1. По длине вектора чисел вычисляется minrun – размер *run*а – минимальный размер подвектора.

Это функция GetMinrun, в ней используются побитовые операции, с помощью которых рассчитывается размер рана, больше 32 и меньше 64, который показал наилучшие результаты у Тима Сорта, к тому же он обеспечивает, что при делении размера вектора на минран будет получено число, лизкое к степени двойки.

2. Исходный вектор «разбивается» на части. На самом деле, создаётся новый вектор runs, в котором друг за другом хранятся индексы начала ранов и их длины. Минимальный размера рана – минран, но может быть и больше, если сохраняется порядок возрастания.

**** Часть с потоками ****

3. Сортировка вставками. Создаём один на всю функцию timsort массив потоков

размера minrun (можно и вектор, но памяти не критично много расходуется). Далее проходимся по вектору runs, создавая каждый раз поток, в который передаём функцию insert_sort, вектор чисел, индекс начала рана и его длину. Но в случае, если уже достигнуто максимальное количество потоков, дожидаемся их завершения и только тогда создаём новый. У меня завершаются все потоки, но я сейчас подумала, что можно было бы только первый завершить. Это бы повысило скорость.

4. Отсортированные массивы нужно слить в один. Проходясь по вектру runs каждый раз, соединяем пары векторов, пока в векторе не останутся 2 числа: индекс начала вектора 0 и его длина, равная длине исходного неортированного массива. Каждый раз, сливая векторы, передаём их координаты и функцию merge в поток.

Немного о функциях

```
void insert_sort(std::vector<int> &vec, size_t start, size_t length)
```

Стандартная сортировка вставками, но выглядит запутанно из-за индексов вектора forsort, переданного в функцию как vec. Обрабатывает только часть, заданную параметрами start и lenght.

```
void merge(std::vector<int> &vec, size_t start1, size_t len1, size_t start2, size_t len2)
```

В отличие от первой сортировки, в неё передаются параметры двух векторов, которые нужно слить. Это часть известной сортировки слиянием, когда уже отсортированные массивы соединяются в один. Здесь отклонение от алгоритма TimSort в порядке слияния векторов, сделано было, чтобы как можно больше разграничить память.

Тесты

Здесь я приведу пример для сортировки 500 чисел, так как большие массивы неудобно смотреть.

```
ann@ann:~/os/lab3$ ./a.out 4
Max threads number is 4
Enter vector for sorting:
95 665 998 -56 867 713 682 771 903 602 10 783 484 329 192 139 470 573 766 564 466 -93 828
506 950 816 620 722 80 -89 38 -15 752 756 639 232 -52 143 660 944 787 902 663 899 688 536
199 318 189 498 955 317 -100 9 226 215 426 786 465 626 313 633 861 646 731 192 843 578 932
```

190 836 117 959 720 679 608 244 234 342 874 729 761 366 939 316 758 434 605 861 402 210
164 722 185 289 361 720 942 32 515 306 551 168 -41 101 62 -95 204 162 724 297 625 424 48 -80
975 45 764 104 39 857 964 776 326 878 124 46 28 408 614 23 805 889 710 457 857 749 620 367
560 151 854 489 424 817 625 545 183 409 645 544 619 419 619 148 983 64 491 144 786 575 -47
963 998 884 783 145 367 186 118 726 -66 860 356 355 820 903 329 643 918 621 685 917 108 -71
670 23 -99 727 -19 300 694 500 761 897 553 809 867 959 402 187 87 452 769 503 494 9 200 524
207 838 595 80 -41 723 74 808 187 287 635 858 342 877 562 428 29 745 -76 610 -59 395 448 79
987 655 443 475 810 77 203 340 358 22 -20 176 196 564 825 646 31 411 955 653 56 144 918 812
875 -67 990 272 268 945 272 938 25 788 -24 438 -60 -9 629 120 864 909 233 342 906 681 4 506
542 929 59 947 667 31 813 432 68 866 878 32 -50 250 457 662 843 340 637 547 119 77 177 371
376 684 864 393 989 -92 648 831 12 429 826 266 238 950 83 19 312 822 776 580 560 36 936 647
949 -62 692 155 939 138 332 749 233 593 -15 37 531 298 264 419 658 536 715 539 116 802 869
3 606 206 409 120 299 614 996 -90 252 210 800 537 584 282 998 623 242 153 443 816 306 806
212 730 762 564 327 902 172 -65 -52 611 34 411 410 -83 511 489 151 808 177 802 -41 -53 -27
713 860 717 381 -52 468 662 272 312 716 377 -92 956 34 307 42 758 996 807 -93 480 143 398 56
732 897 507 62 925 544 548 639 938 944 66 593 244 -14 256 493 167 970 431 831 579 696 39
439 129 567 117 693 603 270 620 900 754 764 71 929 39 -14 -80 189 474 195 624 876 396 64
520 997 984 740 824 894 331 26 155 577 351 510 484 243 611 466 613 0 203 663 152 784 201
626 709 732 231 962 333 868 534 114

Size = 500

MINRUN = 63

Time: 3156 microseconds

Sorted vector:

-100 -99 -95 -93 -93 -92 -92 -90 -89 -83 -80 -80 -76 -71 -67 -66 -65 -62 -60 -59 -56 -53 -52 -52 -
52 -50 -47 -41 -41 -41 -27 -24 -20 -19 -15 -15 -14 -14 -9 0 3 4 9 9 10 12 19 22 23 23 25 26 28 29
31 31 32 32 34 34 36 37 38 39 39 39 42 45 46 48 56 56 59 62 62 64 64 66 68 71 74 77 77 79 80 80
83 87 95 101 104 108 114 116 117 117 118 119 120 120 124 129 138 139 143 143 144 144 145
148 151 151 152 153 155 155 162 164 167 168 172 176 177 177 183 185 186 187 187 189 189
190 192 192 195 196 199 200 201 203 203 204 206 207 210 210 212 215 226 231 232 233 233
234 238 242 243 244 244 250 252 256 264 266 268 270 272 272 272 282 287 289 297 298 299
300 306 306 307 312 312 313 316 317 318 326 327 329 329 331 332 333 340 340 342 342 342
351 355 356 358 361 366 367 367 371 376 377 381 393 395 396 398 402 402 408 409 409 410
411 411 419 419 424 424 426 428 429 431 432 434 438 439 443 443 448 452 457 457 465 466
466 468 470 474 475 480 484 484 489 489 491 493 494 498 500 503 506 506 507 510 511 515
520 524 531 534 536 536 537 539 542 544 544 545 547 548 551 553 560 560 562 564 564 564
567 573 575 577 578 579 580 584 593 593 595 602 603 605 606 608 610 611 611 613 614 614
619 619 620 620 620 621 623 624 625 625 626 626 629 633 635 637 639 639 643 645 646 646
647 648 653 655 658 660 662 662 663 663 665 667 670 679 681 682 684 685 688 692 693 694
696 709 710 713 713 715 716 717 720 720 722 722 723 724 726 727 729 730 731 732 732 740
745 749 749 752 754 756 758 758 761 761 762 764 764 766 769 771 776 776 783 783 784 786
786 787 788 800 802 802 805 806 807 808 808 809 810 812 813 816 816 817 820 822 824 825

826 828 831 831 836 838 843 843 854 857 857 858 860 860 861 861 864 864 866 867 867 868 869 874 875 876 877 878 878 884 889 894 897 897 899 900 902 902 903 903 906 909 917 918 918 925 929 929 932 936 938 938 939 939 942 944 944 945 947 949 950 950 955 955 956 959 959 962 963 964 970 975 983 984 987 989 990 996 996 997 998 998 998
Невооружённым глазом заметно, что последовательность отсортировалась правильно.

Анализ скорости и эффективности

Рассмотрим сначала сортировку 11000 чисел.

Колисетво потоков **p**	Время сортировки с одним потоком **T1** (мкс)	Сортировка с **p** потоками **Tp** (мкс)	Ускорение (**Sp=T1/Tp**)	Эффективность (**Xp**=Sp/p)
1	31661	31661	1	1
2	31661	21764	1,4547	0,7273
3	31661	29352	1,0787	0,3596
4	31661	27828	1,1377	0,2844
5	31661	23379	1,3542	0,2708
6	31661	28879	1,0963	0,1827
7	31661	28184	1,1234	0,1605
8	31661	30146	1,0503	0,1313
9	31661	28031	1,1295	0,1255
10	31661	31372	1,0092	0,1009

Здесь видно, что эффективность понижается с увеличением числа потоков, но плохо просматривается закономерность в ускорении.

Возьмём массив из 30000 тысяч чисел.

Колисетво потоков **p**	Время сортировки с одним потоком **T1** (мкс)	Время сортировки с одним потоком **T1** (мкс)	Ускорение (**Sp=T1/Tp**)	Эффективность (**Xp**=Sp/p)
1	60991	60991	1	1
2	44843	60991	1,36	0,68
3	40417	60991	1,51	0,50

4	35674	60991	1,71	0,42
5	39613	60991	1,54	0,31
6	40304	60991	1,51	0,25
7	40304	60991	1,51	0,22
8	42587	60991	1,43	0,18
9	55509	60991	1,01	0,11
10	44426	60991	1,37	0,14
100	52484	60991	1,16	0,12

Здесь видно, что при большом количестве потоков ускорение падает, и всё же погрешность ещё велика.

Последней возьмём последовательность из 100000 чисел.

Колисетво потоков **p**	Время сортировки с одним потоком **T1** (мкс)	Сортировка с **p** потоками **Tp** (мкс)	Ускорение (**Sp**=**T1**/**Tp**)	Эффективность (**Xp**=Sp/p)
1	181325	181325	1	1
2	181325	146557	1,24	0,62
3	181325	138941	1,31	0,44
4	181325	134154	1,35	0,27
5	181325	145847	1,24	0,25
6	181325	143127	1,27	0,21
7	181325	150748	1,20	0,17
8	181325	145217	1,25	0,16
9	181325	168388	1,08	0,12
10	181325	143738	1,26	0,13
15	181325	140165	1,29	0,09
1000	181325	146601	1,24	0,001
10000	181325	168706	1,07	0,0001

Как мы видим, при большом количестве потоков эффективность сильно снижается. На большом количестве данных не очень большое ускорение.

1. Strace

```
 ann@ann:~/os/lab3$ strace ./a.out 4 <test.txt
execve("./a.out", ["./a.out", "4"], 0x7ffc5ed400d8 /* 63 vars */) = 0
brk(NULL)                               = 0x558aa21db000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=79721, ...}) = 0
mmap(NULL, 79721, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc166b0c000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/usr/lib/x86_64-linux-gnu/libstdc++.so.6", O_RDONLY|O_CLOEXEC) =
3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\220\304\10\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=1594864, ...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7fc166b0a000
mmap(NULL, 3702848, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7fc16656e000
mprotect(0x7fc1666e7000, 2097152, PROT_NONE) = 0
mmap(0x7fc1668e7000, 49152, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x179000) = 0x7fc1668e7000
mmap(0x7fc1668f3000, 12352, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fc1668f3000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libgcc_s.so.1", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\300*\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=96616, ...}) = 0
mmap(NULL, 2192432, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7fc166356000
mprotect(0x7fc16636d000, 2093056, PROT_NONE) = 0
mmap(0x7fc16656c000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x16000) = 0x7fc16656c000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libpthread.so.0", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0000b\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=144976, ...}) = 0
mmap(NULL, 2221184, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7fc166137000
```

```
mprotect(0x7fc166151000, 2093056, PROT_NONE) = 0
mmap(0x7fc166350000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19000) = 0x7fc166350000
mmap(0x7fc166352000, 13440, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fc166352000
close(3)                        = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20\35\2\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2030928, ...}) = 0
mmap(NULL, 4131552, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7fc165d46000
mprotect(0x7fc165f2d000, 2097152, PROT_NONE) = 0
mmap(0x7fc16612d000, 24576, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x1e7000) = 0x7fc16612d000
mmap(0x7fc166133000, 15072, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x7fc166133000
close(3)                        = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
openat(AT_FDCWD, "/lib/x86_64-linux-gnu/libm.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\200\272\0\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0644, st_size=1700792, ...}) = 0
mmap(NULL, 3789144, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_DENYWRITE, 3,
0) = 0x7fc1659a8000
mprotect(0x7fc165b45000, 2093056, PROT_NONE) = 0
mmap(0x7fc165d44000, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_DENYWRITE, 3, 0x19c000) = 0x7fc165d44000
close(3)                        = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7fc166b08000
mmap(NULL, 12288, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
0) = 0x7fc166b05000
arch_prctl(ARCH_SET_FS, 0x7fc166b05740) = 0
mprotect(0x7fc16612d000, 16384, PROT_READ) = 0
mprotect(0x7fc165d44000, 4096, PROT_READ) = 0
mprotect(0x7fc166350000, 4096, PROT_READ) = 0
mprotect(0x7fc16656c000, 4096, PROT_READ) = 0
mprotect(0x7fc1668e7000, 40960, PROT_READ) = 0
mprotect(0x558aa1a97000, 4096, PROT_READ) = 0
mprotect(0x7fc166b20000, 4096, PROT_READ) = 0
munmap(0x7fc166b0c000, 79721)           = 0
```

```
set_tid_address(0x7fc166b05a10)         = 4040
set_robust_list(0x7fc166b05a20, 24)     = 0
rt_sigaction(SIGRTMIN, {sa_handler=0x7fc16613ccb0, sa_mask=[],
sa_flags=SA_RESTORER|SA_SIGINFO, sa_restorer=0x7fc166149980}, NULL, 8) = 0
rt_sigaction(SIGRT_1, {sa_handler=0x7fc16613cd50, sa_mask=[],
sa_flags=SA_RESTORER|SA_RESTART|SA_SIGINFO, sa_restorer=0x7fc166149980}, NULL,
8) = 0
rt_sigprocmask(SIG_UNBLOCK, [RTMIN RT_1], NULL, 8) = 0
prlimit64(0, RLIMIT_STACK, NULL, {rlim_cur=8192*1024, rlim_max=RLIM64_INFINITY})
= 0
brk(NULL)                               = 0x558aa21db000
brk(0x558aa21fc000)                     = 0x558aa21fc000
futex(0x7fc1668f409c, FUTEX_WAKE_PRIVATE, 2147483647) = 0
futex(0x7fc1668f40a8, FUTEX_WAKE_PRIVATE, 2147483647) = 0
fstat(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 0), ...}) = 0
write(1, "Max threads number is 4\n", 24Max threads number is 4
) = 24
write(1, "Enter vector for sorting:\n", 26Enter vector for sorting:
) = 26
fstat(0, {st_mode=S_IFREG|0664, st_size=1944, ...}) = 0
read(0, "95 665 998 -56 867 713 682 771 9"..., 4096) = 1944
read(0, "", 4096)                       = 0
write(1, "Size = 500\n", 11Size = 500
)           = 11
write(1, "MINRUN = 63\n", 12MINRUN = 63
)          = 12
mmap(NULL, 8392704, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -
1, 0) = 0x7fc1651a7000
mprotect(0x7fc1651a8000, 8388608, PROT_READ|PROT_WRITE) = 0
clone(child_stack=0x7fc1659a6fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1659a79d0, tls=0x7fc1659a7700, child_tidptr=0x7fc1659a79d0) =
4041
mmap(NULL, 8392704, PROT_NONE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7fc1649a6000
mprotect(0x7fc1649a7000, 8388608, PROT_READ|PROT_WRITE) = 0
clone(child_stack=0x7fc1651a5fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1651a69d0, tls=0x7fc1651a6700, child_tidptr=0x7fc1651a69d0) =
```

4042
mmap(NULL, 8392704, PROT_NONE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7fc1641a5000
mprotect(0x7fc1641a6000, 8388608, PROT_READ|PROT_WRITE) = 0
clone(child_stack=0x7fc1649a4fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1649a59d0, tls=0x7fc1649a5700, child_tidptr=0x7fc1649a59d0) =
4043
mmap(NULL, 8392704, PROT_NONE,
MAP_PRIVATE|MAP_ANONYMOUS|MAP_STACK, -1, 0) = 0x7fc15f7ff000
mprotect(0x7fc15f800000, 8388608, PROT_READ|PROT_WRITE) = 0
clone(child_stack=0x7fc15fffefb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc15ffff9d0, tls=0x7fc15ffff700, child_tidptr=0x7fc15ffff9d0) = 4044
clone(child_stack=0x7fc15fffefb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc15ffff9d0, tls=0x7fc15ffff700, child_tidptr=0x7fc15ffff9d0) = 4045
clone(child_stack=0x7fc1649a4fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1649a59d0, tls=0x7fc1649a5700, child_tidptr=0x7fc1649a59d0) =
4046
clone(child_stack=0x7fc1651a5fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1651a69d0, tls=0x7fc1651a6700, child_tidptr=0x7fc1651a69d0) =
4047
clone(child_stack=0x7fc1659a6fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1659a79d0, tls=0x7fc1659a7700, child_tidptr=0x7fc1659a79d0) =
4048
clone(child_stack=0x7fc1659a6fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1659a79d0, tls=0x7fc1659a7700, child_tidptr=0x7fc1659a79d0) =
4049
clone(child_stack=0x7fc1651a5fb0,

**flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1651a69d0, tls=0x7fc1651a6700, child_tidptr=0x7fc1651a69d0) =
4050**
**clone(child_stack=0x7fc1649a4fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1649a59d0, tls=0x7fc1649a5700, child_tidptr=0x7fc1649a59d0) =
4051**
**clone(child_stack=0x7fc15fffefb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc15ffff9d0, tls=0x7fc15ffff700, child_tidptr=0x7fc15ffff9d0) = 4052**
futex(0x7fc15ffff9d0, FUTEX_WAIT, 4052, NULL) = 0
**clone(child_stack=0x7fc15fffefb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc15ffff9d0, tls=0x7fc15ffff700, child_tidptr=0x7fc15ffff9d0) = 4053**
**clone(child_stack=0x7fc1649a4fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|C
LONE_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEA
RTID, parent_tidptr=0x7fc1649a59d0, tls=0x7fc1649a5700, child_tidptr=0x7fc1649a59d0) =
4054**
futex(0x7fc15ffff9d0, FUTEX_WAIT, 4053, NULL) = 0
clone(child_stack=0x7fc1649a4fb0,
flags=CLONE_VM|CLONE_FS|CLONE_FILES|CLONE_SIGHAND|CLONE_THREAD|CLON
E_SYSVSEM|CLONE_SETTLS|CLONE_PARENT_SETTID|CLONE_CHILD_CLEARTID,
parent_tidptr=0x7fc1649a59d0, tls=0x7fc1649a5700, child_tidptr=0x7fc1649a59d0) = 4055
futex(0x7fc1649a59d0, FUTEX_WAIT, 4055, NULL) = 0
write(1, "Time: 5028 microseconds\n", 24Time: 5028 microseconds
) = 24
write(1, "Sorted vector:\n", 15Sorted vector:
) = 15
write(1, "-100 -99 -95 -93 -93 -92 -92 -90"..., 1024-100 -99 -95 -93 -93 -92 -92 -90 -89 -83 -80 -80
-76 -71 -67 -66 -65 -62 -60 -59 -56 -53 -52 -52 -52 -50 -47 -41 -41 -41 -27 -24 -20 -19 -15 -15 -14
-14 -9 0 3 4 9 9 10 12 19 22 23 23 25 26 28 29 31 31 32 32 34 34 36 37 38 39 39 39 42 45 46 48
56 56 59 62 62 64 64 66 68 71 74 77 77 79 80 80 83 87 95 101 104 108 114 116 117 117 118 119
120 120 124 129 138 139 143 143 144 144 145 148 151 151 152 153 155 155 162 164 167 168
172 176 177 177 183 185 186 187 187 189 189 190 192 192 195 196 199 200 201 203 203 204
206 207 210 210 212 215 226 231 232 233 233 234 238 242 243 244 244 250 252 256 264 266
268 270 272 272 272 282 287 289 297 298 299 300 306 306 307 312 312 313 316 317 318 326

```
327 329 329 331 332 333 340 340 342 342 342 351 355 356 358 361 366 367 367 371 376 377
381 393 395 396 398 402 402 408 409 409 410 411 411 419 419 424 424 426 428 429 431 432
434 438 439 443 443 448 452 457 457 465 466 466 468 470 474 475 480 484 484 489 489 491
493 494 498 500 503 506 506 507 510 511 515 520 524 531 534 536 536 537) = 1024
write(1, " 539 542 544 544 545 547 548 551"..., 922 539 542 544 544 545 547 548 551 553 560
560 562 564 564 564 567 573 575 577 578 579 580 584 593 593 595 602 603 605 606 608 610
611 611 613 614 614 619 619 620 620 620 621 623 624 625 625 626 626 629 633 635 637 639
639 643 645 646 646 647 648 653 655 658 660 662 662 663 663 665 667 670 679 681 682 684
685 688 692 693 694 696 709 710 713 713 715 716 717 720 720 722 722 723 724 726 727 729
730 731 732 732 740 745 749 749 752 754 756 758 758 761 761 762 764 764 766 769 771 776
776 783 783 784 786 786 787 788 800 802 802 805 806 807 808 808 809 810 812 813 816 816
817 820 822 824 825 826 828 831 831 836 838 843 843 854 857 857 858 860 860 861 861 864
864 866 867 867 868 869 874 875 876 877 878 878 884 889 894 897 897 899 900 902 902 903
903 906 909 917 918 918 925 929 929 932 936 938 938 939 939 942 944 944 945 947 949 950
950 955 955 956 959 959 962 963 964 970 975 983 984 987 989 990 996 996 997 998 998 998
) = 922
exit_group(0)                           = ?
+++ exited with 0 +++
```

2. Текст программы

```
#include <iostream>
#include <vector>
#include <thread>
#include <chrono>


// function to thread
void insert_sort(std::vector<int> &vec, size_t start, size_t length) {
    for (size_t i = start + 1; i < length + start; ++i){
            for (size_t j = i; (j > start) && (vec[j - 1] > vec[j]); --j) {
                int tmp = vec[j - 1];
                vec[j - 1] = vec[j];
                vec[j] = tmp;
            }
        }
```

```
}

// function to thread
void merge(std::vector<int> &vec, size_t start1, size_t len1, size_t start2, size_t len2) {
    int tmp[len1 + len2];
    size_t beg = start1;
    size_t cur1 = start1, cur2 = start2;
    for (size_t i = 0; i < len1 + len2; ++i) {
        if ((vec[cur1] <= vec[cur2]) && (cur1 < start2) && (cur2 < start2 + len2)) {
            tmp[i] = vec[cur1];
            ++cur1;
        }
        else if ((vec[cur1] > vec[cur2]) && (cur1 < start2) && (cur2 < start2 + len2)) {
            tmp[i] = vec[cur2];
            ++cur2;
        }
        else if ((cur1 < start2) && (cur2 == start2 + len2)) {
            tmp[i] = vec[cur1];
            ++cur1;
        }
        else if ((cur1 == start2) && (cur2 < start2 + len2)) {
            tmp[i] = vec[cur2];
            ++cur2;
        }
    }
    for (size_t i = 0; i < len1 + len2; ++i) {
        vec[beg] = tmp[i];
        ++beg;
    }
```

```
}

size_t GetMinrun(size_t n)
{
        size_t r = 0;
    while (n >= 64) {
            r |= n & 1;
            n >>= 1;
        }
        return n + r;
}

void TimSort(std::vector<int> &vec, size_t n) {
    size_t minrun = GetMinrun(vec.size());
    std::cout << "MINRUN = " << minrun << std::endl;
    std::vector<int> runs;
    size_t cur = 0, tmp = 0;
    while (cur < vec.size() - 1) {
        size_t run_cur = cur;
        if (vec[run_cur] > vec[run_cur + 1]) {
            tmp = vec[run_cur];
            vec[run_cur] = vec[run_cur + 1];
            vec[run_cur + 1] = tmp;
        }
        bool right = true;
        while (right) {
            // если не соблюдается порядок возрастания
            if (vec[run_cur] > vec[run_cur + 1])
                right = false;
```

```
        // но минран ещё не достигнут
        if (!(right) && (run_cur - cur < minrun))
            right = true;
        // конец вектора
        if (run_cur == vec.size() - 1)
            right = false;

        if (right)
            ++run_cur;
    }
    size_t size = run_cur - cur + 1;

    runs.push_back(cur);
    runs.push_back(size);

    cur = run_cur + 1;
}

std::thread thr[n];
size_t c = 0;

for (size_t i = 0; i < runs.size() - 1; i += 2) {
    if (c < n) {
        thr[c] = std::thread(insert_sort, std::ref(vec), runs[i], runs[i + 1]);
        ++c;
    } else {
        for (size_t j = 0; j < c; ++j) {
            thr[j].join();
        }
```

```
        c = 0;
        thr[c] = std::thread(insert_sort, std::ref(vec), runs[i], runs[i + 1]);
        ++c;
    }
}

for (size_t i = 0; i < c; ++i) {
    thr[i].join();
}

// merge
while(runs.size() > 2) {
    size_t k = 0;
    for (size_t i = 0; i < runs.size(); i += 4) {
        int x_start = runs[i], x_len = runs[i + 1];
        int y_start = runs[i + 2], y_len = runs[i + 3];
        if (k < n) {
            thr[k] = std::thread(merge, std::ref(vec), x_start, x_len, y_start, y_len);
            ++k;
        } else {
            for (size_t j = 0; j < k; ++j)
                thr[j].join();
            k = 0;
            thr[k] = std::thread(merge, std::ref(vec), x_start, x_len, y_start, y_len);
            ++k;
        }
        x_len += y_len;
        runs[i + 1] = x_len;
        runs[i + 2] = -1;
```

```
            runs[i + 3] = -1;
        }

        for (int j = runs.size() - 1; j >= 0; --j) {
            if (runs[j] == -1) {
                runs.erase(runs.begin() + j);
            }
        }

        for (size_t i = 0; i < k; ++i) {
            thr[i].join();
        }
    }
}

int main(int argc, char* argv[]) {
    int m, n;
    if ((argc != 2) || (atoi(argv[1]) < 1)) {
        std::cout << "Invalid number" << std::endl;
        exit(EXIT_FAILURE);
    }
    n = strtol(argv[1], NULL, 10);
    std::cout << "Max threads number is " << n << std::endl;
    std::vector<int> forsort;
    std::cout << "Enter vector for sorting:" << std::endl;
    while (std::cin >> m) {
        forsort.push_back(m);
    }
    std::cout << "Size = " << forsort.size() << std::endl;
```

```
    if (forsort.size() < 64) {
        insert_sort(forsort, 0, forsort.size());
    } else {
        auto t1 = std::chrono::high_resolution_clock::now();
        TimSort(forsort, n);
        auto t2 = std::chrono::high_resolution_clock::now();
        auto duration = std::chrono::duration_cast<std::chrono::microseconds>( t2 - t1 ).count();
        std::cout << "Time: " << duration << " microseconds" << std::endl;
    }
    std::cout << "Sorted vector:" << std::endl;
    for (size_t i = 0; i < forsort.size(); ++i) {
        std:: cout << forsort[i] << " ";
    }
    std::cout << std::endl;
}
```

Выводы

Я осуществила многопоточность в своей программе, выполняя лабораторную работу. Оценив результаты, я сделала для себя некоторые выводы и заметила недочёты (один описан в алгоритме).

Я поняла, что не каждому алгоритму подходит многопоточность, это зависит от того, насколько линейный алгоритм, то есть, как часто используется один и тот же фрагмент памяти, и на скорость это также влияет. Почему это важное замечание? Потоки используют общую память, им не выделятся, как процессам, своя, в результате чего несколько потоков могут пытаться изменить один и тот же участок памяти, но здесь
это решалось завершением потока.

Также не стоит надеяться, что многопоточность – панацея для ускорения программ. Это в какой-то мере абстракция, так как настоящая параллельная работа может осуществляться только на разных ядрах. В другом случае процессор работает в режиме

псевдопараллельности. У меня максимальная скорость была на 4 потоках, а максимальная эффективность на 2, потому что реальных 2 ядра, ещё 2 виртуальных, то есть всего 4 потока.

Создание потоков также занимает время. Отсюда следует, что никому не нужно огромное количество потоков. Это сильно нагружает процессор, который ещё может обслуживать другие процессы. Если представить, что большинство программ использует несколько потоков, кажется, что выигрыша в скорости мы не получим.

И всё-таки на небольшом количестве потоков был заметен существенный выигрыш в скорости, так что потоки – полезная и важная вещь, которую нужно знать.