Задания практикума «Многопоточное программирование»

1. Создание нити

Напишите программу, которая создает нить. Используйте атрибуты по умолчанию. Родительская и вновь созданная нити должны распечатать десять строк текста.
Решение: pthread_create/pthread_create.c (cd pthread_create; make)

2. Ожидание нити

Модифицируйте программу упр. 1 так, чтобы вывод родительской нити производился после завершения дочерней. Используйте pthread_join.
Решение: pthread_create_join/pthread_create_join.c

3. Параметры нити

Напишите программу, которая создает четыре нити, исполняющие одну и ту же функцию. Эта функция должна распечатать последовательность текстовых строк, переданных как параметр. Каждая из созданных нитей должна распечатать различные последовательности строк.
Решение: pthread_create_param/pthread_create_param.c

4. Принудительное завершение нити

Дочерняя нить должна распечатывать текст на экран. Через две секунды после создания дочерней нити, родительская нить должна прервать ее вызовом функции pthread_cancel.
Решение: pthread_cancel/pthread_cancelc

5. Обработка завершения нити

Модифицируйте программу упр. 4 так, чтобы дочерняя нить перед завершением распечатывала сообщение об этом. Используйте pthread_cleanup_push.
Решение: pthread_cancel_handler/pthread_cancel_handler.c

6. Многопоточный cp -R

Реализуйте многопоточную программу рекурсивного копирования дерева подкаталогов, функциональный аналог команды cp(1) с ключом -R. Программа должна принимать два параметра – полное путевое имя корневого каталога исходного дерева и полное путевое имя целевого дерева. Программа должна обходить исходное дерево каталогов при помощи opendir(3C)/readdir_r(3C) и определять тип каждого найденного файла при помощи stat(2). Для определения размера буфера для readdir_r используйте pathconf(2) (sizeof (struct dirent) + pathconf(directory)+1).
Для каждого подкаталога должен создаваться одноименный каталог в целевом дереве и запускаться отдельная нить, обходящая этот подкаталог. Для каждого регулярного файла

должна запускаться нить, копирующая этот файл в одноименный файл целевого дерева при помощи open(2)/read(2)/write(2). Файлы других типов (символические связи, именованные трубы и др.) следует игнорировать.

При копировании больших деревьев каталогов возможны проблемы с исчерпанием лимита открытых файлов. Очень важно закрывать дескрипторы обработанных файлов и каталогов при помощи close(2)/closedir(3C). Тем не менее, для очень больших деревьев этого может оказаться недостаточно. Допускается обход этой проблемы при помощи холостого цикла с ожиданием (если open(2) или readdir(3C) завершается с ошибкой EMFILE, то допускается сделать sleep(3C) и повторить попытку открытия через некоторое время).

Обратите также внимание, что значения дескрипторов открытых файлов могут переиспользоваться, т.е. в разные моменты времени один и тот же дескриптор может указывать на разные файлы. Чтобы избежать связанных с этим проблем, избегайте передачи дескрипторов между нитями. Вся работа с дескриптором от создания до закрытия должна происходить в одной нити.

Дополнительное упражнение: при помощи команды time(1) сравните ресурсы, потребляемые вашей программой и командой cp -R при копировании одного и того же дерева каталогов. Объясните наблюдаемые различия. Каким образом их можно устранить? Следует ли вообще реализовать копирование файлов таким способом и если да, то в каких условиях?
Решения: pthread_cp-R/pthread_cp_r.c,
pthread_cp-R_correct/pthread_cp_r.c (использует condvar для решения проблемы нехватки файловых дескрипторов).

7. Вычисление π

Напишите программу, которая вычисляет число Пи при помощи ряда Лейбница .
Однопоточная версия такой программы доступна в файле pi_serial.c. Количество потоков программы должно определяться параметром командной строки. Количество итераций может определяться во время компиляции. Для передачи частичных сумм ряда, подсчитанных потоками, используйте pthread_exit(3C)/pthread_join(3C).

Обратите внимание, что на 32разрядных платформах sizeof(double)>sizeof(void *), поэтому частичную сумму ряда нельзя преобразовывать к указателю, для нее надо выделять собственную память.
Решение: PI/pi_parallel.c

8. Вычисление π пока не надоест

Модифицируйте программу упражнения 7 так, чтобы сама по себе она не завершалась. Вместо этого, после нажания ^-C (после получения сигнала SIGINT) программа должна как можно скорее завершаться, собирать частичные суммы ряда и выводить полученное приближение числа.

Рекомендации: ожидаемое решение состоит в том, что вы установите обработчик SIGINT. Этот обработчик должен устанавливать глобальную флаговую переменную. Вычислительные нити должны просматривать значение флага через некоторое количество итераций, например через 1000000, и выходить при помощи pthread_exit, если флаг установлен. Подумайте, как минимизировать ошибку, обусловленную тем, что разные потоки к моменту завершения успели пройти разное количество итераций. Скорее всего, такая минимизация должна обеспечиваться за счет увеличения времени между получением сигнала и выходом.
Решение: PI_interruptible/pi_parallel.c

9. Обедающие философы

Возьмите за основу программу din_phil.c. Эта программа симулирует известную задачу про обедающих философов. Пять философов сидят за круглым столом и едят спагетти. Спагетти едят при помощи двух вилок. Каждые двое философов, сидящих рядом, пользуются общей вилкой. Философ некоторое время размышляет, потом пытается взять вилки и принимается за еду. Съев некоторое количество спагетти, философ освобождает вилки и снова начинает размышлять. Еще через некоторое время он снова принимается за еду, и т.д., пока спагетти не кончатся. Если одну из вилок взять не получается, философ ждет, пока она освободится. В программе din_phil.c философы симулируются при помощи нитей, периоды размышлений и еды – при помощи usleep(3C), а вилки – при помощи мутексов. Философы всегда берут сначала левую вилку, а потом правую. При некоторых обстоятельствах это может приводить к мертвой блокировке. Измените протокол взаимодействия философов с вилками таким образом, чтобы мертвых блокировок не происходило.
Решение: philosophers/din_phil_ordered.c

10. Синхронизированный вывод

Модифицируйте программу упр. 1 так, чтобы вывод родительской и дочерней нитей был синхронизован: сначала родительская нить выводила первую строку, затем дочерняя, затем родительская вторую строку и т.д. Используйте мутексы. Рекомендуется использовать мутексы типа PTHREAD_MUTEX_ERRORCHECK.

Явные и неявные передачи управления между нитями (sleep(3C)/usleep(3C), sched_yield(3RT)) и холостые циклы разрешается использовать только на этапе инициализации.
Решение: producer_consumer_print/p_c_funny.c (это решение, конечно, нельзя признать good practice – как и саму задачу – но очень уж забавно код выглядит)

11. Синхронизированный вывод 2

Докажите, что задача 10 не может быть решена с использованием двух мутексов без использования других средств синхронизации.

12. Синхронизированный вывод 3

Решите задачу 10 с использованием условной переменной и минимально необходимого количества мутексов.

Решение: producer_consumer_print/p_c_condvar.c

13. Синхронизированный вывод 4

Решите задачу 10 с использованием двух семафоров-счетчиков
Решение: producer_consumer_print/p_c_sem.c

14. Синхронизированный вывод 5

Если вы решили задачи 11 и 13, объясните, почему ваше доказательство неприменимок семафорам-счетчикам.

15. Синхронизированный вывод двух процессов

Решите задачу 10 с использованием двух процессов (а не нитей) и именованных семафоров-счетчиков.
Решение: semtest.c

16. Синхронизированный доступ к списку

Родительская нить программы должна считывать вводимые пользователем строки и помещать их в начало связанного списка. Строки длиннее 80 символов можно разрезать на несколько строк. При вводе пустой строки программа должна выдавать текущее состояние списка. Дочерняя нить пробуждается каждые пять секунд и сортирует список в лексикографическом порядке (используйте пузырьковую сортировку). Все операции над списком должны синхронизоваться при помощи мутекса.
Решение: ordered_list/global_mutex.c

17. Синхронизированный доступ к списку 2

Переделайте программу упр. 16 так, чтобы с каждой записью (а также с заголовком списка) был связан свой собственный мутекс.

Примечание: при перестановке записей списка, необходимой при реализации пузырьковой сортировки, необходимо блокировать мутексы трех записей.

Примечание 2: чтобы избежать мертвых блокировок, мутексы записей, более близких к началу списка, всегда захватывайте раньше.

Примечание 3: преподаватель может потребовать, чтобы программа включала две или более сортирующие нити, а также потребовать изменить интервал между сортировками.
Решение: ordered_list/local_mutex.c

18. Синхронизированный доступ к списку 3

Модифицируйте программу упр. 17 так, чтобы дочерняя нить засыпала на одну секунду между исполнениями каждого шага сортировки (между перестановками записей в списке). При этом можно будет наблюдать процесс сортировки по шагам.
Решение: ordered_list/local_mutex2.c

19. Использование блокировки чтения-записи

Модифицируйте программу упр. 16 так, чтобы вместо мутекса использовалась блокировка чтения-записи.
Решение: ordered_list/global_rwlock.c

20. Использование блокировки чтения-записи 2

Модифицируйте аналогичным образом программу упр. 18
Решение: ordered_list/local_rwlock.c

21. Обедающие философы 2

Решите задачу упр. 9 при помощи атомарного захвата вилок. Когда философ может взять одну вилку, но не может взять другую, он должен положить вилку на стол и ждать, пока освободятся обе вилки.

Рекомендация: создайте еще мутекс forks и условную переменную. При попытке взять вилку философ должен захватывать forks и проверять доступность обоих вилок при помощи pthread_trylock(3C). Если одна из вилок недоступна, философ должен освободить вторую вилку (если он успел ее захватить) и заснуть на условной переменной. Освобождая вилки, философ должен оповещать остальных философов об этом при помощи условной переменной. Тщательно продумайте процедуру захвата и освобождения мутексов, чтобы избежать ошибок потерянного пробуждения.
Решение: /philosophers/din_phil_transaction.c

22. Производственная линия

Разработайте имитатор производственной линии, изготавливающей винтики (widget). Винтик собирается из детали C и модуля, который, в свою очередь, состоит из деталей A и B. Для изготовления детали A требуется 1 секунда, B – две секунды, C – три секунды. Задержку изготовления деталей имитируйте при помощи sleep. Используйте семафоры-счетчики.
Решение: semaphore_production/production_line.c

23. Производитель-потребитель

Реализуйте очередь сообщений, которая может использоваться для обмена данными между двумя или большим количеством нитей. Реализация очереди должна поддерживать функции
void mymsginit(queue *);
void mymsqdrop(queue *);
void mymsgdestroy(queue *);

int mymsgput(queue *, char * msg);
int mymsgget(queue *, char * buf, size_t bufsize);

Допускается реализация на C++ с заменой mymsginit и mymsgdestroy на конструктор и деструктор, а операций get и put на соответствующие методы.
mymsgput принимает в качестве параметра ASCIIZ строку символов, обрезает ее до 80 символов (если это необходимо) и помещает ее в очередь. Если очередь содержит более 10 записей, mymsgput блокируется. Функция возвращает количество переданных символов.
mymsgget возвращает первую запись из очереди, обрезая ее до размера пользовательского буфера (если это необходимо). В любом случае, запись извлекается из очереди полностью. Если очередь пуста, mymsgget блокируется. Функция возвращает количество прочитанных символов.
mymsgdrop должна приводить к разблокированию ожидающих операций get и put. Ожидавшие вызовы и все последующие вызовы get и put должны возвращать 0.
mymsqdestroy должна вызываться после того, как будет известно, что ни одна нить больше не попытается выполнять операции над очередью.

Необходимо продемонстрировать работу очереди с двумя производителями и двумя потребителями.

Для синхронизации доступа к очереди используйте семафоры-счетчики.
Решение: msgqueue/msgqueue_sem.c

24. Производитель-потребитель 2

Реализуйте задачу 22 с использованием условных переменных и минимально необходимого числа мутексов.
Решение: msgqueue/msgqueue_cond.c

25. Многопоточный сервер

Реализуйте сервер, который принимает TCP соединения и транслирует их. Сервер должен получать из командной строки следующие параметры:

1. Номер порта P, на котором следует слушать.
2. Имя или IP-адрес узла N, на который следует транслировать соединения.
3. Номер порта P', на который следует транслировать соединения.

Сервер принимает все входящие запросы на установление соединения на порт P. Для каждого такого соединения он открывает соединение с портом P' на сервере N. Затем он транслирует все данные, получаемые от клиента, серверу N, а все данные, получаемые от сервера N – клиенту. Если сервер N или клиент разрывают соединение, наш сервер также должен разорвать соединение. Если сервер N отказывает в установлении соединения, следует разорвать клиентское соединение.

Сервер должен обеспечивать трансляцию 510 соединений при лимите количества открытых файлов на процесс 1024. Сервер не должен быть многопоточным и никогда не должен блокироваться при операциях чтения и записи. Не следует использовать неблокирующиеся сокеты. Следует использовать select или poll.
Решение: network_apps/forwarder.c

26. псевдомногопоточный HTTP-клиент

Реализуйте простой HTTP-клиент. Он принимает один параметр командной строки – URL. Клиент делает запрос по указанному URL и выдает тело ответа на терминал как текст (т.е. если в ответе HTML, то распечатывает его исходный текст без форматирования). Вывод производится по мере того, как данные поступают из HTTP-соединения. Когда будет выведено более экрана (более 25 строк) данных, клиент должен продолжить прием данных, но должен остановить вывод и выдать приглашение Press space to scroll down.

При нажатии пользователем клиент должен вывести следующий экран данных. Для одновременного считывания данных с терминала и из сетевого соединения используйте системный вызов select.
Решение: network_apps/httpclients/body_select.c

27. псевдомногопоточный HTTP-клиент 2

Реализуте задачу упр. 26, используя системные вызовы aio_read/aio_write
Решение: network_apps/httpclients/body_aio.c

28. Многопоточный HTTP-клиент

Реализуйте задачу упр. 26, используя две нити, одну для считывания данных из сетевого соединения, другую для взаимодействия с пользователем.
Решение: network_apps/httpclients/body_pthread.c

29. Псевдомногопоточный кэширующий прокси

Реализуйте простой кэширующий HTTP-proxy с кэшем в оперативной памяти.

Прокси должен быть реализован как один процесс и один поток, использующий для одновременной работы с несколькими сетевыми соединениями системный вызов select или poll. Прокси должен обеспечивать одновременную работу нескольких клиентов (один клиент не должен ждать завершения запроса или этапа обработки запроса другого клиента).

30. Многопоточный кэширующий прокси

Реализовать задачу 29, создавая для каждого входящего HTTP-соединения свою нить. При невозможности создать поток допускается блокировать входящие соединения или возвращать ошибку.

31. Многопоточный кэширующий прокси с рабочими потоками

Реализовать задачу 29, используя рабочие потоки (worked threads). При запуске прокси должен принимать параметр, целое число, указывающее размер пула потоков. Прокси должен запустить указанное число нитей. Необходимо обеспечить одновременную обработку количества запросов, превосходящего количество нитей в пуле; блокировка входящих соединений недопустима. Разумеется, при этом каждая из нитей в разные моменты времени будет вынуждена обрабатывать разные соединения. Для управления соединениями используйте select или poll.