Задания по Операционным системам. Второй семестр. Special edition

1. Задачи на работу с потоками.

Блок задач на “удовлетворительно”

1.1. Создание потоков посредством POSIX API:

a. Склонируйте репозиторий git@github.com:mrutman/os.git. Изучите и запустите программу threads/thread.c. Добейтесь того, чтобы гарантированно выполнялись оба потока.

b. Измените программу, добавив создание 5-ти потоков с одинаковой поточной функцией.

c. В поточной функции распечатайте:

i. идентификаторы процесса, родительского процесса, потока. Для получения идентификатора потока используйте функции pthread_self() и gettid().

Сравните с тем что функция pthread_create() вернула через первый аргумент. Объясните результат.

Почему для сравнения идентификаторов POSIX потоков надо использовать функцию pthread_equal()?

ii. адреса локальной, локальной статической, локальной константной и глобальной переменных. Объясните результат.

d. В поточной функции попробуйте изменить локальную и глобальную переменные. Видны ли изменения из других потоков? Объясните результат.

e. Изучите /proc/pid/maps для полученного процесса. Найдите в нем стеки потоков.

f. Запустите программу из-под strace. Найдите системные вызовы, которые создали ваши потоки.

1.2. Потоки Joinable and Detached.

a. Напишите программу, в которой основной поток будет дожидаться завершения созданного потока.

b. Измените программу так чтобы созданный поток возвращал число 42, а основной поток получал это число и распечатывал.

c. Измените программу так чтобы созданный поток возвращал указатель на строку

“hello world”, а основной поток получал этот указатель и распечатывал строку.

d. Напишите программу, которая в бесконечном цикле будет создавать поток, с поточной функцией, которая выводит свой идентификатор потока и завершается. Запустите. Объясните результат.

e. Добавьте вызов pthread_detach() в поточную функцию. Объясните результат.

f. Вместо вызова pthread_detach() передайте в pthread_create() аргументы, задающие тип потока - DETACHED. Запустите, убедитесь что поведение не

изменилось.

1.3. Передача параметров в поточную функцию.

a. Создайте структуру с полями типа int и char*. Создайте экземпляр этой структуры и проинициализируйте. Создайте поток и передайте указатель на эту структуру в качестве параметра. В поточной функции распечатайте содержимое структуры.

b. Измените программу так чтобы поток создавался как detached поток. Объясните, в какой области памяти нужно располагать структуру в этом случае.

i. Объясните, какие проблемы могут быть у структуры на стеке. Что будет, если основной поток завершится раньше ребенка? А если после его завершения будет запущен новый поток, что-то пишущий по тому адресу, где была структура? Понаблюдайте за значением полей структуры, видимым из detached потока. Объясните увиденное (настоятельно рекомендуется почитать исходники)

ii. Объясните, какие проблемы могут быть у структуры на стеке. Кто должен ее освобождать? Предложите способы или идеи способов смягчения этих проблем (некоторые из них вам уже известны), реализуйте один из них в коде.

1.4 Прерывание потока.

a. Напишите программу, в которой поточная функция в бесконечном цикле распечатывает строки. Используйте pthread_cancel() для того чтобы ее остановить.

b. Измените программу так чтобы поточная функция ничего не распечатывала, а в бесконечном цикле увеличивала счетчик на 1. Используйте pthread_cancel() для того чтобы ее остановить. Объясните результат. Что можно сделать, чтобы pthread_cancel() прервал поток?

c. В поточной функции выделите память под строку “hello world” с помощью malloc(). Распечатывайте в бесконечном цикле полученную строку. Используйте pthread_cancel() для того чтобы прервать поточную функцию. Добейтесь, чтобы по завершению память, выделенная под строку освобождалась. Используйте pthread_cleanup_push/pop().

d. Что делать, если нет возможности инструментировать (дополнить) код поточной функции, и она не останавливается "стандартным" pthread_cancel? Как должен работать механизм, который обеспечит остановку в таком случае? Подсказка: pthread_cancel тут тоже поможет, но с некоторыми дополнительными действиями. Как должен работать = можно посмотреть в исходниках, а можно подумать с точки зрения разработчика ОС, не обязательно линукса.

1.5. Обработка сигналов в многопоточной программе.

a. Напишите программу с тремя потоками, такими что: первый поток блокирует получения всех сигналов, второй принимает сигнал SIGINT при помощи обработчика сигнала, а третий - сигнал SIGQUIT при помощи функции sigwait().

b. Можно ли установить обработчики сигнала для каждого потока?

c. Создайте четвертый поток без обработки сигналов вообще. Попробуйте использовать для отправки сигналов kill из командной строки. Объясните результат.

d. Создайте четвертый поток, аналогичный второму или третьему, но с другим сообщением об обработке сигнала. Попробуйте отправлять различные сигналы различными способами. Объясните результат. Детерминировано ли такое поведение? (Исходники - хороший источник для ответа на вопрос).

Блок задач на “хорошо”

1.6. Разработать собственную функцию для создания ядерных потоков - аналог pthread_create():
int mythread_create(mythread_t thread, void *(start_routine), void *arg);
Функция должна возвращать успех-неуспех.
Оформите реализацию в виде динамической библиотеки.

a. mythread_join должен быть. Любого уровня примитивности.

b. mythread_create при ошибке в каком-то из внутренних системных вызовов должен сообщать пользователю, что что-то пошло не так, а не просто успешно отрабатывать (по аналогии с pthread_create, опять же).

c. Тестирование вашей реализации должно быть сложнее, чем create + join. Создавайте несколько потоков, проверяйте ошибки - пока хотя бы для себя не сможете сказать, что реализация протестирована.

d. mythread_cancel писать не нужно, но как его реализовали бы - быть готовым ответить надо.

Блок задач на “отлично”

1.7. Разработать собственную функцию для создания пользовательских потоков:
int uthread_create(uthread_t thread, void *(start_routine), void *arg);
Функция должна возвращать успех-неуспех.
Допускается реализация без вытеснения потока.

a. Если реализовали без вытеснения потока, то в примере, которым доказываете работоспособность, каждая потоковая функция должна отдавать поток управления (yeild) хотя бы несколько раз. Т.е. нельзя делать последовательное выполнение "1 поток начался, 1 поток закончился, 2 поток начался 2 поток закончился 3 поток начался" .. без возврата к 1 или 2 потоку.

2. Задачи на синхронизацию.

Блок задач на “удовлетворительно”

2.1. Проблема конкурентного доступа к разделяемому ресурсу.

a. В каталоге sync репозитория git@github.com:mrutman/os.git вы найдете простую реализацию очереди на списке. Изучите код, соберите и запустите программу queue-example.c. Посмотрите вывод программы и убедитесь что он соответствует вашему пониманию работы данной реализации очереди. Добавьте реализацию функции queue_destroy().

b. Изучите код программы queue-threads.c и разберитесь что она делает. Соберите программу.

i. Запустите программу несколько раз. Если появляются ошибки выполнения, попытайтесь их объяснить и определить что именно вызывает ошибку. Какие именно ошибки вы наблюдали?

ii. Поиграйте следующими параметрами:

1. размером очереди (задается в queue_init()). Запустите программу с размером очереди от 1000 до 1000000.
2. привязкой к процессору (задается функцией set_cpu()). Привяжите
потоки к одному процессору (ядру) и к разным.
3. планированием потоков (функция sched_yield()). Попробуйте убрать эту функцию перед созданием второго потока.
4. Объясните наблюдаемые результаты.

iii. Для пункта “заполняемость очереди” - экспериментально обоснуйте обьяснение наблюдаемого результата. Для этого может пригодиться отладочный вывод каких-то событий в очереди или какой-то информации о ее состоянии, например, когда она становится пустой или заполненной. Не переборщите с отладочным выводом - вывод штука долгая и влияет на работу самой очереди, замедляя ее.

2.2. Синхронизация доступа к разделяемому ресурсу

a. Измените реализацию очереди, добавив спинлок для синхронизации доступа к разделяемым данным.

i. Подумайте, какие операции должны выполняться с захваченным примитивом синхронизации, а какие не обязательно. Объясните принятое решение. Что такое критическая секция?

b. Убедитесь, что не возникает ошибок передачи данных через очередь.

c. Поиграйте параметрами из пункта 2.1:

i. Оцените загрузку процессора.

ii. Оцените время проведенное в пользовательском режиме и в режиме ядра.

iii. Оцените текущую заполненность очереди, количество попыток чтения-записи и количество прочитанных-записанных данных.

iv. Объясните наблюдаемые результаты.

d. Часто бывает, что поток, пишущий данные в очередь вынужден ожидать их (например, из сети на select()/poll()). Проэмулируйте эту ситуацию, добавив в поточную функцию писателя периодический вызов usleep(1). Выполните задания из пункта c.
e. Измените реализацию очереди, заменив спинлок на мутекс. Проделайте задания из пунктов b, c и d. Сравните со спинлоком.
i. Загружает ли реализация на мутексах процессор почти на уровне спинлока?
Если да, то попробуйте исправить, вставив в нужном месте usleep. Поэкспериментируйте со значением usleep, чтобы производительность очереди была на уровне остальных вариантов.
f. Измените реализацию очереди, добавив условную переменную. Проделайте задания из пунктов b, c и d. Сравните со спинлоком и мутексом.
g. Используйте для синхронизации доступа к очереди семафоры. Проделайте задания из пунктов b, c и d. Сравните со спинлоком, мутексом и условной переменной.
h. Мутекс и бинарный семафор.
Зачем одно, если есть другое?
Если нашли ответ, не поленитесь его проверить - так ли это?
Если получилось не так, то что надо исправить в коде?

Блок задач на “хорошо”

2.3 Реализуйте односвязный список, хранящий строки длиной менее 100 символов, у которого с каждым элементом связан отдельный примитив синхронизации (за основу можно взять реализацию списка, на котором построен очередь queue_t). Объявление такого списка может выглядеть, например, так:

```
typedef struct _Node {
char value[100];
struct _Node* next;
pthread_mutex_t sync;
} Node;
typedef struct _Storage {
Node *first;
} Storage;
```

Первый поток пробегает по всему хранилищу и ищет количество пар строк, идущих по возрастанию длины. Как только достигнут конец списка, поток инкрементирует глобальную переменную, в которой хранится, количество выполненных им итераций и сразу начинает новый поиск.
Второй поток пробегает по всему хранилищу и ищет количество пар строк, идущих по убыванию длины. Как только достигнут конец списка, поток инкрементирует глобальную переменную, в которой хранится количество выполненных им итераций и сразу начинает новый поиск.
Третий поток пробегает по всему хранилищу и ищет количество пар строк, имеющих одинаковую
длину. Как только достигнут конец списка, поток инкрементирует

глобальную переменную, в которой хранится количество выполненных им итераций и сразу начинает новый поиск.
Запускает 3 потока, которые в непрерывном бесконечном цикле случайным образом проверяют - требуется ли переставлять соседние элементы списка (не значения) и выполняют перестановку. Каждая успешная попытка перестановки фиксируется в соответствующей глобальной переменной-счетчике.
Используйте для синхронизации доступа к элементам списка спинлоки, мутексы и блокировки чтения-записи. Понаблюдайте как изменяются (и изменяются ли) значения переменных счетчиков и объясните результат. Проверьте для списков длины 100, 1000, 10000, 100000
При реализации обратите внимание на следующие пункты:
- продумайте ваше решение, чтобы избежать ошибок соревнования.
- необходимо блокировать все записи с данными которых производится работа.
- при перестановке записей списка, необходимо блокировать три записи.
- чтобы избежать мертвых блокировок, примитивы записей, более близких к началу списка, всегда захватывайте раньше.

a. Внимательно перечитайте условие лабы. И еще раз.
b. “в непрерывном бесконечном цикле случайным образом проверяют - требуется ли переставлять” - допускаются оба варианта прочтения, и где случайный образ заключается в проверке с каким-то шансом для каждого элемента в порядке их следования, и где случайно выбирается элемент, для которого будет выполняться проверка, из всех. Второй вариант интереснее.
c. Докажите, что в вашей реализации не будет ошибок соревнования.
d. Сравните мутексы и rwlock по итоговым параметрам, обьясните, почему так.

Блок задач на “отлично”

2.4. Сделайте “грубую” реализацию спинлока и мутекса при помощи cas-функции и futex. Используйте их для синхронизации доступа к разделяемому ресурсу. Объясните принцип их работы.

Как правило, реализовать несложно.
a. Мне хочется, чтобы не оставалось ощущения, что использование футекса, атомарного CAS и чего-нибудь еще необычного (atomic_int?) воспринималось как применение магии, про которую сказали что она сработает - надо уметь обьяснить, что именно оно делает.
Для этого может быть хорошей идеей:

i. если ассемблерная вставка - то покажите ее,
ii. если футекс - расскажите, что там внутри.

Может быть полезным заглянуть в линуксовую реализацию.
b. Сравнить грубую реализацию мутекса с pthread_mutex; Обьяснить.

i. какой-нибудь общий тестовый сценарий на разных мутексах
ii. для "обьяснить" опять же можно посмотреть, что там в реализации pthread_mutex.

3. Мини проекты.

Из этого раздела достаточно сделать одну задачу на оценку, на которую вы претендуете.
Общие требования к минипроектам:
0. Проверяйте, что оно работает
1. Проверяйте, что оно работает где-то, кроме тривиального случая (1 коннект/копирование одного файла.)
Это может быть скриптик на кучу коннектов, попытки кэшировать видос, etc
2. Нарисуйте перед сдачей схему архитектуры лабы. Какие модули за что отвечают и что делают, etc.
Неплохим вариантом будет куда-то туда добавить инфу о использованных примитивах синхронизации, кто когда их захватывает.
Это все может ускорить сдачу.

Блок задач на “удовлетворительно”

3.1 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?

Блок задач на “хорошо”

3.2 Реализуйте многопоточный HTTP-proxy (версия HTTP 1.0). Прокси должен принимать соединения на 80 порту и перенаправлять их на требуемый сервер. Вся обработка соединения должна происходить в отдельном потоке.

Блок задач на “отлично”

3.3. Реализуйте многопоточный кэширующий HTTP-proxy (версия HTTP 1.0). Прокси должен принимать соединения на 80 порту и возвращать данные из кэша. В случае если для запроса нет записей в кэше, то должен быть создан отдельный поток, который загрузит в кэш требуемые данные. Данные должны пересылаться клиенту как только они начали появляться в кэше.

a. Надо уметь демонстрировать основную фишку прокси - только один поток скачивает и кэширует, когда потребителей может быть несколько. Если кто-то уже начал кэшировать, то новый клиент с тем же запросом не ждет скачивания, не начинает качать сам, а получает из кэша по мере скачивания первым клиентом. обратите внимание на требования 13, 20.

Остальные требования к прокси (3.2 если нет слова кэш, 3.3)

1. Должен поддерживаться HTTP 1.0. Прокси должен корректно информировать как клиента, так и сервер об используемой версии протокола
2. Клиенты и серверы HTTP 0.9 (даже если таковые удастся найти) могут не поддерживаться, т.е. прокси может отказываться работать с ними, либо работа с ними может осуществляться в режиме HTTP 1.0. Возникающие при этом проблемы не являются основанием для отказа в приеме задания.
3. Поддержка HTTP 1.1 не обязательна. Поддерживать персистентные соединения 1.1 не требуется.
4. Необходимо поддерживать операцию GET. Поддержка всех остальных операций опциональна; отказ поддерживать остальные операции не является основанием для отказа в приеме задания. Ответы на операции PUT и POST, если сами эти операции будут реализованы, кэшировать не следует.
5. Кэшировать надлежит только ответы типа 200 (нормальная передача страницы). Все остальные ответы следует передавать браузеру без изменений и кэшировать не следует.
6. Кэшировать следует как текстовые, так и бинарные ресурсы, с обязательным сохранением MIME типа ресурса.
7. Кэш хранится в памяти и может полностью теряться при перезапуске прокси.
8. Обработка полей заголовка, управляющих кэшированием, таких, как last modified и pragma no cache, не обязательна. Некорректная работа сайтов с динамическим HTML, обусловленная некорректной обработкой этих параметров, не является основанием для отказа в приеме задания.

9. Поддержка cookies не обязательна. Некорректная работа сайтов, использующих cookie (в том числе и для авторизации) не является основанием для отказа в приеме задания

10. Поддержка любых механизмов авторизации на сайтах не обязательна. Проверяется только корректность анонимного доступа.

11. Допускается как самостоятельная реализация анализа заголовка HTTP, так и использование third-party библиотек парсеров заголовка.

Note: Лучше взять что-то уже готовое. Например взять вот этот https://github.com/h2o/picohttpparser.

12. Допускается использование языков C и C++

Note: Но если вы собираетесь писать на плюсах как на си, с голыми указателями и без шаблонов/ооп, то задумайтесь еще 10 раз, а стоит ли оно того..

13. Все варианты задания предполагают параллельную обработку запросов, т.е. при тестировании необходимо продемонстрировать возможность открыть несколько клиентских сессий и показать, что ни одна из сессий не ждет завершения операции ни одной из других сессий.

14. Стандартной реализации асинхронных DNS запросов не существует, поэтому задержки на gethostbyname(3C) в однопоточном прокси допустимы.

15. Прокси не должны иметь ограничений по количеству клиентских сессий (кроме количества дескрипторов открытых файлов). Т.е. при наличии пула надо уметь поддерживать больше сессий, чем имеется потоков в пуле.

16. Если пула нет, реализация имеет право отвергать или задерживать входящие соединения при невозможности создать новый поток.

17. Если две клиентские сессии скачивают одну и ту же страницу, необходимо, чтобы обе сессии работали с одной и той же записью кэша, понимали, что запись неполная и адекватно реагировали на докачку.

18. Прокси должен корректно обрабатывать сброс клиентских сессий. В том числе, в случае, когда две или более сессий работали с одной записью кэша, после сброса одной из них, остальные сессии должны корректно продолжить докачку страницы.

19 Допускается создание двух потоков на каждое клиентское соединение: «клиентский» поток обрабатывает соединение с клиентом, «серверный» – с сервером. При работе двух клиентских сессий с одной записью кэша при этом следует создавать два клиентских потока, но один серверный. При передаче клиенту полной записи кэша, серверный поток можно не запускать. В случае пула потоков, допускается разделение пула на два, серверных и клиентских потоков, исполняющих разный код.

20. При сбросе единственной сессии, работавшей с записью кэша, допускается как сброс докачки страницы и уничтожение записи в кэше, так и фоновое продолжение докачки. Преподаватель имеет право потребовать изменения стратегии обработки этой ситуации, т.е. потребовать переделать сброс докачки на ее фоновое продолжение или наоборот (это может быть полезно для проверки корректности управления записями в кэше).

Note: скорее всего, преподаватель не будет требовать изменения стратегии.

21. Использование явных и неявных холостых циклов, а в особенности холостых циклов с ожиданием, для синхронизации потоков не допускается. Допускается использование стандартных примитивов синхронизации pthread, системных вызовов select и poll и вызовов асинхронного ввода-вывода.

22. В частности, высокая загрузка процессора (по показаниям top) при малой активности соединений интерпретируется как явный или неявный холостой цикл и является основанием для отказа в приеме задания. При обнаружении такого поведения, преподаватель имеет право потребовать доказательства корректности используемой схемы синхронизации.

Note: если при паре клиентов прокси улетает в 30%, что-то не так. Преподаватель тогда потребует, это точно..

23. Задержки при открытии сайтов интерпретируются как холостой цикл с ожиданием, и являются основанием для отказа в приеме задания.

24. Вызовы функций sleep, usleep, pause и т.д. в коде приложения интерпретируются как холостой цикл с ожиданием и являются основанием для отказа в приеме задания.

25. Трудновоспроизводимые «глюки» при работе прокси интерпретируются как ошибки соревнования, и являются основанием для отказа в приеме задания. После обнаружения таких «глюков» преподаватель имеет право потребовать доказательства корректности используемой схемы синхронизации.

Новые правила:

26. Кэш должен уметь продолжать работать после заполнения. Или аллоцируйте на лету еще один сегмент, или пилите примитивный garbage collector, удаляющий ненужные записи. что-то простое, вроде LRU - подойдет

27. В любом случае, GC или аллокация должны быть в отдельном потоке, и не должны мешать работе прокси. Т.е. условный монитор следит за кэшем, чтобы всегда было место для записи.

28. Если запрашивается файл больше вместимости кэша, допускается ожидание его расширения либо сброс запроса с логированием ошибки

29. Реализовывать TTL не обязательно

30. Если запись уже есть в кэше, ходить на сервер и спрашивать last updated не обязательно