**Лабораторная работа**

**Тема**

Разработка многопоточного сетевого сервера

**Цель работы**

Получить практические навыки разработки многопоточного сетевого сервера.

**Время выполнения работы**

2 академических часа

**Используемое методическое и лабораторное обеспечение**

1. Дистрибутив Debian версии.
2. ПК с текстовым редактором (для подготовки отчета).

**Теоретические сведения**

**Потоки**

В Unix-системах для исполнения программ используются объекты, называемые процессами. Процессу выделяется собственное виртуальное адресное пространство и некоторые иные ресурсы. Для каждого процесса создается иллюзия последовательного исполнения. Программа исполняется в рамках процесса так же, как она исполнялась бы в однозадачной ОС.

Процесс взаимодействует с ядром ОС при помощи системных вызовов. При исполнении системного вызова, процесс исполняет специальную команду (у современных версий x86 эта команда – SYSCALL, у SPARC – TA 0x8 для 32-битных программ, TA 0x40 для 64-битных программ), которая переключает адресное пространство и передает управление ядру. Процессы в традиционных Unix-системах могут взаимодействовать друг с другом только при помощи системных вызовов – чтения и записи в каналы и разделяемые файлы, а в более современных системах – при помощи System V IPC.

Таким образом, процессы в Unix надежно изолированы друг от друга. Нарушения целостности данных одного процесса (например, в результате переполнения буфера или ошибок при работе с указателями) приводят к аварийному завершению этого процесса, но не затрагивают напрямую другие процессы. Даже удаленное исполнение кода в рамках одного из процессов позволяет злоумышленнику, написавшему код, получить лишь те привилегии, которые имел данный процесс.

Однако ряд задач – примеры, которые будут рассмотрены далее в этом разделе – требует реализации в виде нескольких параллельно (или, на однопроцессорной машине, квазипараллельно) исполняющихся процессов. Традиционные Unix-системы предполагали решать такие задачи при помощи нескольких взаимодействующих процессов. Среди системных вызовов, поддерживаемых всеми современными Unix-системами можно выделить несколько групп, специально предназначенных для такого взаимодействия – это вызовы для работы с каналами (pipe), сетевые средства (которые можно применять и для взаимодействия между процессами на одной машине) и System V IPC. Нельзя сказать, что для всех задач эти средства оптимальны, а для некоторых случаев даже могут стать критическими.

Кроме этого, нередко возникает потребность в переносе приложений, разработанных для ОС, допускавших несколько нитей исполнения в пределах одной задачи или процесса, под Unix. Поэтому в современных Unix-системах было введено понятие нитей (thread), которые соответствуют единицам планирования в рамках одного процесса. Нити разделяют общее адресное пространство, но планируются независимо. Иллюзия последовательного исполнения создается для нити, а не для процесса в целом.

**2. Зачем нужны многопоточные программы**

1. Улучшение времени реакции интерактивных программ.
2. Улучшение времени реакции серверных приложений. Возможность обрабатывать несколько запросов одновременно.
3. Использование дополнительных ресурсов на многопроцессорных и гипертрединговых компьютерах.
4. Задачи реального времени.

**2.1 Улучшение времени реакции интерактивных программ**

1. Фоновое скачивание страницы в браузере.
2. Фоновый ввод-вывод (например, утилита просмотра файла может считывать файл по мере его просмотра).
3. Фоновая проверка орфографии.
4. Фоновое переразбиение текста на страницы в WYSIWYG текстовых процессорах.

**2.2 Улучшение времени реакции серверных приложений**

Основные две характеристики серверов приложений – это среднее время исполнения запроса и количество запросов, обрабатываемых в единицу времени. Причем, это две разные характеристики.

Среднее время исполнения запроса – это основная характеристика производительности сервера с точки зрения отдельного пользователя, а конкретнее со стороны клиентского приложения. Увеличение этого времени ведет к снижению производительности пользователя, ему приходится тратить больше времени на исполнение однотипных самых задач. Также, слишком большое время реакции системы негативно влияет на опыт эксплуатации системы пользователем.

Количество запросов, обрабатываемых в единицу времени – это основная характеристика производительности сервера с точки зрения владельца этого сервера, то есть непосредственно со стороны серверного приложения. Нахождение способа увеличить данное количество, не ухудшая или незначительно ухудшая показатели исполнения каждого из запросов, позволит произвести подключение большего количества пользователей к серверу. Следовательно, оптимизация программной части сервера может позволить компенсировать недостатки аппаратной.

Процесс исполнения одиночного запроса типичным серверным приложением описан на рис. 1.1. Он состоит из пяти этапов: приема запроса (выполняется сетевым интерфейсом), анализа запроса (выполняется центральным процессором), считывания данных с диска (выполняется дисковым контроллером), создания ответа (выполняется центральным процессором) и передачи ответа. Соответственно, время исполнения каждого из этих этапов может различаться в зависимости от типа сервиса, предоставляемого сервером, и от самих запросов. Сгладив данные требование, рассмотрим теоретический сервер, обрабатывающий однотипные запросы, у которых структура обработки запроса именно такова, как представлена на рис. 1.1, а время исполнения всех этапов относительно одинаково.

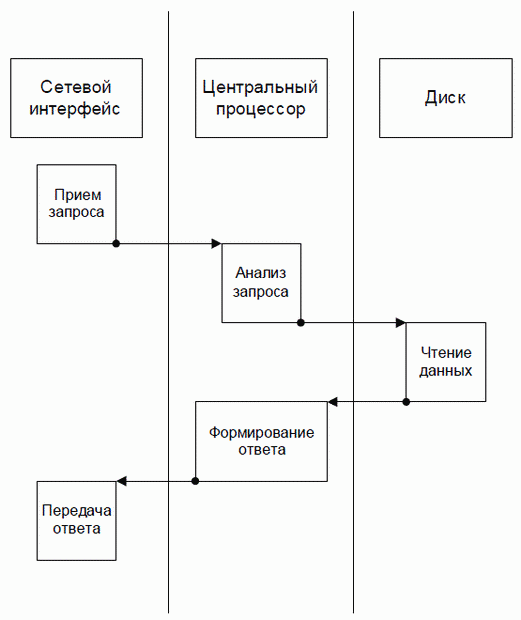


Рисунок 1 — йцу

Рассмотрим теперь исполнение потока запросов. Однопоточный сервер должен был бы исполнять запросы строго последовательно, поэтому максимальное количество запросов, исполняемых в секунду, было бы равно 1/t, где t – время исполнения одиночного запроса.

При этом среднее время исполнения запроса не будет равно t, а будет расти в зависимости от вероятности перекрытия запросов во времени. Теоретико-вероятностные расчеты и практика показывают, что когда поток запросов приближается к 1/t в секунду, эта вероятность становится весьма значительной, так что время исполнения одиночного запроса может увеличиться во много раз. Однако к многопоточному серверу эти расчеты неприменимы. Рассмотрим, как многопоточный сервер мог бы обрабатывать поток запросов (см. [рис. 1.2](https://intuit.ru/studies/courses/570/426/lecture/9688?page=2#image.1.2)).

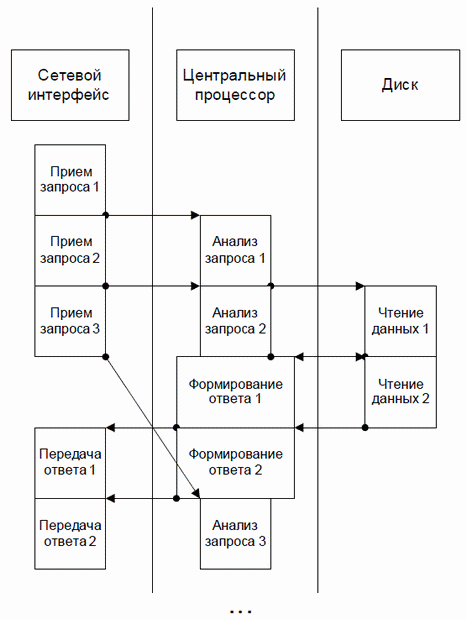


Рисунок 2 –

Видно, что сервер совмещает этапы исполнения перекрывающихся во времени запросов. Если относительные времена исполнения этапов запросов таковы, как на [рис. 1.1](https://intuit.ru/studies/courses/570/426/lecture/9688?page=2#image.1.1) и [рис. 1.2](https://intuit.ru/studies/courses/570/426/lecture/9688?page=2#image.1.2), избежать взаимодействия запросов не удается, так что среднее время исполнения запроса может вырасти по сравнению с t. Но количество запросов, исполняемых в секунду, оказывается значительно больше, чем 1/t.

Более того, видно, что мы могли бы повысить производительность системы, установив второй процессор и второй сетевой интерфейс.

Однако переделка однопоточного серверного приложения в многопоточное требует специальной поддержки со стороны операционной системы и значительной переработки кода приложения. Все современные ОС общего назначения – Sun Solaris, другие системы семейства Unix, IBM zOS, Win 32 и Win 64 – обеспечивают соответствующую поддержку, но далеко не все разработчики приложений умеют этой поддержкой пользоваться.

**3. Проблемы многопоточности**

Первые попытки организации параллельных вычислений предпринимались еще в 60е годы XX столетия. Теория межпроцессного и многопоточного взаимодействия была в основном разработана еще тогда, однако массовое распространение многопоточное программирование получило лишь в 90е годы. Это было обусловлено рядом причин, основные из которых перечислены ниже:

1. Несовместимость со старыми (однопоточными) компиляторами
2. Несовместимость со старыми библиотеками
3. Несовместимость или ограниченная поддержка многопоточных программ другими инструментальными средствами, в первую очередь отладчиками
4. Несовместимость многих принятых практик программирования с многопоточностью.

Первые три из названных причин чисто технические, для их решения требуется переработка инструментальных средств разработки программ. Однако, поскольку такая переработка нарушает совместимость, массовый переход на новый инструментарий занял многие годы. К тому же, поскольку первые версии многопоточных инструментальных средств обладали различными недостатками, это также затрудняло их принятие разработчиками.

Первые среды программирования, реализовавшие многопоточность, такие, как Simula 67, использовали для создания потоков и межпоточного взаимодействия специальные конструкции языка. Эта традиция продолжалась и до 80х годов и воплощена в таких языках, как Ada, Occam, Parallel Fortran. Пожалуй, только последний из этих языков –Parallel Fortran – получил широкое практическое применение.

При программировании на C / C++ с использованием POSIX Thread Library специальные компиляторы не требуются. POSIX Thread Library можно использовать с любыми компиляторами, реализующими соглашения о вызовах, соответствующие ABI вашей аппаратной платформы. Так, в Solaris 10 многопоточные программы можно писать как на GNU C, так и с помощью Sun Studio C compiler. Все, необходимое для поддержки многопоточности, реализовано на уровне библиотек.

Sun Studio 11 C compiler также поддерживает средства для параллельного программирования OpenMP. При компиляции с ключом -xopenmp включается поддержка директив параллелизации OpenMP в исходном коде программы. При компиляции с ключом -xautopar компилятор пытается автоматически найти параллелизуемые участки в программе и реализовать их многопоточное исполнение. OpenMP в Solaris реализован на основе тех же технологий, что и POSIX Threads, поэтому следует проявлять осторожность при совместном использовании OpenMP и POSIX Threads в одной программе. В рамках данного курса OpenMP не рассматривается. Более сложной проблемой на практике оказалась поддержка многопоточности на уровне библиотек. Проблемы, возникающие со старыми библиотеками, в основном описываются на следующей лекции, но полный масштаб этих проблем станет вам понятен лишь после завершения курса в целом. Первые реализации стандартных библиотек C и Fortran для Unix-систем были рассчитаны на однопоточное исполнение, поэтому перенос в многопоточную среду потребовал их переработки. Первые версии C и С++ компиляторов для старых Unix-систем (как и старые реализации C / C++ для OS/2 и Win 32) использовали две разные версии библиотек для однопоточных и многопоточных программ.

В Solaris 10 в основном завершена переработка стандартных библиотек языка C для оптимальной поддержки многопоточности. Стандартная библиотека /usr/lib/libc.so и основная масса других библиотек, входящих в стандартную поставку системы, работают как в однопоточных, так и в многопоточных программах. Тем не менее, эта поддержка до сих пор сопряжена с некоторыми ограничениями. Многие функции, в том числе некоторые функции стандартных библиотек ANSI/ISO C в многопоточных программах следует использовать с осторожностью. Необходимая информация приводится в секции ATTRIBUTES страниц системного руководства (man(1)) по соответствующим функциям. Подробнее этот вопрос обсуждается на следующей лекции.

При использовании библиотек других поставщиков необходимо получить информацию о поддержке многопоточности у поставщика библиотеки. Обычно эта информация должна указываться в документации по библиотеке, но, к сожалению, не все разработчики библиотек это делают. Поддержка многопоточной отладки – также относительно недавнее достижение. В современных версиях популярного отладчика gdb (GNU Debugger) многопоточность поддерживается, но при сборке отладчика ее можно выключить, поэтому до сих пор нередко можно встретить бинарный дистрибутив gdb без поддержки многопоточности. Для проверки, поддерживает ли ваша версия gdb многопоточность, следует исполнить команду info threads. Если поддержка включена, эта команда выводит список нитей отлаживаемой программы, если выключена — команда игнорируется отладчиком.

gdb) info threads

3 process 35 thread 27 0x34e5 in sigpause ()

2 process 35 thread 23 0x34e5 in sigpause ()

\* 1 process 35 thread 13 main (argc=1, argv=0x7ffffff8) at threadtest.c:68

1.1.

В поставку Sun Studio 11 входит отладчик dbx, который полностью поддерживает многопоточную отладку как в режиме командной строки, так и в экранном режиме под управлением Sun Studio IDE. Некоторые сведения об этом отладчике сообщаются в нашем курсе.

**Порядок выполнения работы**

**Контрольные вопросы**