Лабораторная работа №1
«Скалярное время Лэмпорта и алгоритм взаимного исключения Лэмпорта»

Данная лабораторная работа состоит из трех последовательных этапов. Правильность результата, полученного на каждом из этапов, можно проверить, отправив код боту как указано в разделе «Требования к реализации и среда выполнения» у соответствующего этапа. Итоговым результатом работы является выполнение всех трех этапов.

Этап №1. Реализация межпроцессного взаимодействия посредством сообщений

Введение

Для изучения различных аспектов функционирования распределенных систем часто используют несколько моделей распределенной обработки информации. При этом та или иная модель выбирается в зависимости от исследуемой задачи из области распределенных вычислений. Взаимодействие процессов в моделях обычно происходит посредством обмена сообщениями.

Распределенное вычисление удобно рассматривать в виде совокупности дискретных событий, каждое из которых вызывает небольшое изменение состояния всей системы. Система становится «распределенной» благодаря тому обстоятельству, что каждое событие приводит к изменению только части глобального состояния всей системы. А именно, при наступлении события изменяется лишь локальное состояние одного процесса и, возможно, состояние одного или нескольких каналов связи.

В данном этапе работе необходимо реализовать библиотеку межпроцессного взаимодействия посредством обмена сообщениями, которая будет использована в дальнейшем для изучения основных свойств распределенных вычислительных систем.

Исходные данные

Число N процессов, составляющих распределенную систему. При проверке используются значения от 1 до 10.

Постановка задачи

Распределенная система состоит из конечного множества N независимых процессов $\{P_1, P_2, \ldots, P_N\}$. Если процесс P_i может напрямую отправлять сообщения процессу P_j, то мы будем говорить, что между процессом P_i и процессом P_j имеется канал C_{ij}. Для удобства все каналы будем считать однонаправленными. Чтобы два процесса имели возможность обмениваться сообщениями, между ними необходимо установить два разнонаправленных канала, каждый из которых служит для передачи сообщений в одном направлении. Топология сети распределенной системы является полносвязной, т.е. каждый процесс может взаимодействовать со всеми другим процессами. В данном задании используется асинхронный обмен сообщениями.

Теоретическая модель процесса определяется в виде множества его возможных состояний (в т.ч. начальных состояний) и множества дискретных событий. Событие e, происходящее в процессе P, представляет собой атомарное действие, которое может изменить состояние самого процесса P и состояние канала C, инцидентного этому процессу: состояние C будет изменено при отправке сообщения по этому каналу или при получении сообщения из этого канала. Поэтому все события могут быть классифицированы как внутренние события, события отправки и события получения

сообщения. Следует отметить, что в качестве одного атомарного события также часто рассматривают отправку сразу нескольких сообщений по нескольким каналам связи, инцидентным *P*, например при широковещательной или групповой рассылке. При этом для удобства мы будем полагать, что получение сообщения не может совпадать с отправкой или получением других сообщений в виде одного события.

Задание

Необходимо реализовать библиотеку межпроцессного взаимодействия для распределенной системы, описанной выше. В качестве процессов системы используются процессы ОС Linux, и обмен сообщениями осуществляется с помощью неименованных каналов (*pipes*). Родительский процесс создает все дочерние процессы при помощи функции *fork()* (завершение дочерних процессов отслеживается при помощи *wait()*), каналы открываются функцией *pipe()*.

Количество создаваемых дочерних процессов в полносвязной топологии определяется параметром командной строки «*-p X* », где *X* — количество процессов. При создании процессов следует не забывать, в каком процессе происходит выполнение, что позволяет предотвратить создание 2^X процессов. Таким образом, общее число процессов в распределенной системе $N = X + 1$.

Процессы обмениваются сообщениями посредством записи и чтения из каналов. Каждое сообщение состоит из заголовка и тела сообщения. В заголовок (структура *MessageHeader*) входят следующие поля:

1. «магическая подпись», которая используется при автоматической проверке лабораторных работ (константа *MESSAGE_MAGIC*);
2. длина тела сообщения;
3. тип сообщения;
4. метка времени.

Таким образом, максимальная длина сообщения составляет 64 Кб. В работе необходимо использовать структуру заголовка и константы из прилагаемого заголовочного файла *ipc.h*.

Информацию обо всех открытых дескрипторах каналов (чтение / запись) необходимо вывести в файл *pipes.log*. Это помогает обнаружить часто встречаемую ошибку: реализацию топологии «общая шина» вместо полносвязной. Кроме того, следует не забывать, что неиспользуемые дескрипторы необходимо закрыть.

Каждый процесс должен иметь свой локальный идентификатор: $[0..N-1]$. Причем родительскому процессу присваивается идентификатор *PARENT_ID*, равный 0. Данные идентификаторы используются при отправке и получении сообщений.

При запуске программы родительский процесс осуществляет необходимую подготовку для организации межпроцессного взаимодействия, после чего создает *X* идентичных дочерних процессов. Функция родительского процесса ограничивается созданием дочерних процессов и дальнейшим мониторингом их работы.

Выполнение каждого дочернего процесса состоит из трех последовательных фаз:

1. процедура синхронизации со всеми остальными процессами в распределенной системе;
2. «полезная» работа дочернего процесса;
3. процедура синхронизации процессов перед их завершением.

Первая фаза работы дочернего процесса заключается в том, что при запуске он пишет в лог (все последующие действия также логируются) и отправляет сообщение типа *STARTED* всем остальным процессам, включая родительский. Затем процесс дожидается сообщений *STARTED* от других дочерних процессов, после чего первая фаза его работы считается оконченной. На первом этапе дочерние процессы не выполняют никакой «полезной» работы, поэтому сразу переходят к третьей фазе завершения собственного

выполнения. В этой фазе дочерние процессы отправляют сообщение типа *DONE* всем, включая родителя. Условием завершения дочернего процесса является получение сообщений *DONE* от всех остальных дочерних процессов. В сообщениях *STARTED* и *DONE* в качестве тела сообщения необходимо использовать такие же строки, как были записаны в лог. Таким образом, для дочерних процессов определены следующие события (в скобках указаны имена строк форматирования для логирования):

- процесс начал выполнение работы (*log_started_fmt*);
- процесс получил сообщения о запуске всех остальных процессов (*log_received_all_started_fmt*);
- процесс окончил выполнение «полезной» работы (*log_done_fmt*);
- процесс получил сообщения о выполнении «полезной» работы всеми дочерними процессами (*log_received_all_done_fmt*).

Родительский процесс не должен отправлять сообщения дочерним процессам, однако сообщения *STARTED* и *DONE* должны быть им получены. Родительский процесс завершается при завершении всех остальных процессов.

Все события логируются на терминал и в файл *events.log*. При логировании необходимо использовать форматы сообщений из прилагаемого заголовочного файла. Стоит обратить внимание на то, что последовательности событий, регистрируемые при различных выполнениях программы, не совпадают.

В реализации запрещается использовать многопоточность: один процесс — один поток. Кроме того, нельзя использовать разделяемую память, примитивы синхронизации (семафоры и т.п.), функции *select()* и *poll()*.

Требования к реализации и среда выполнения

Реализацию необходимо выполнить на языке программирования Си с использованием предоставленных заголовочных файлов и библиотеки из архива *pa1_starter_code.tar.gz*. Архив содержит следующие файлы:

ipc.h	объявления структур данных и функций для организации межпроцессного взаимодействия. Объявленные функции необходимо реализовать;
common.h	некоторые константы;
pa1.h	форматы строк для логирования;

Заголовочные файлы содержат большое число важных комментариев, пояснений и рекомендаций. Запрещается модифицировать любые файлы из архива: при автоматической проверке они заменяются на оригинальные.

Работа присылается в виде архива с именем *pa1.tar.gz*, содержащим каталог *pa1*. Все файлы с исходным кодом и заголовки должны находиться в корне этого каталога. Среда выполнения — Linux (Ubuntu 14.04, clang-3.5). При автоматической проверке используется следующая команда: *clang -std=c99 -Wall -pedantic *.c*. При наличии варнингов работа не принимается. При успешном выполнении запущенные процессы не должны использовать *stderr*, код завершения программы должен быть равен 0.

Этап №2. Скалярное время Лэмпорта

Введение

В распределенных системах отсутствуют глобальные часы, отсчитывающие общее для всех процессов время, и к показаниям которых процессы могли бы получать мгновенный доступ. Существуют различные способы синхронизации физических часов: радио часы (WWV), протокол NTP и его производные — однако максимальная точность

синхронизации составляет порядка нескольких миллисекунд, что для ряда приложений недостаточно.

Для правильной работы распределенных приложений необходимо определить отношение «произошло раньше», связывающее события процессов между собой, не опираясь на понятие единого физического времени. Поэтому распределенные приложения используют различные варианты логического времени.

Исходные данные

- Число N процессов, составляющих распределенную банковскую систему;
- Начальные балансы S для каждого из счетов (один процесс — один счет),

где N и S — целые числа. При автоматической проверке $N \in [2; 10]$, $S \in [1; 99]$.

Постановка задачи

За основу банковской системы необходимо взять модель распределенной системы, описанную в условии к этапу №1. Разделяют два типа процессов, составляющих банковскую систему:

- процессы, принимающие запросы от клиентов (тип «К»);
- процессы, отвечающие за обслуживание счетов (тип «С»).

В данном этапе лабораторной работе используется один процесс типа «К» и $N - 1$ процессов типа «С». Каждый процесс имеет собственные логические часы. Процессы обмениваются сообщениями в асинхронном режиме.

Процесс «К» на основе запросов клиента формирует запросы к соответствующим процессам «С». Поддерживаются следующие операции:

- перевод денег между счетами;
- получение информации об истории изменения баланса.

При переводе денег процесс «К» отправляет сообщение процессу «C_{src}», с которого осуществляется списание. Процесс «C_{src}» после выполнения требуемых операций пересылает исходное сообщение процессу «C_{dst}». Процесс «C_{dst}» после выполнения всех необходимых действий отправляет процессу «К» подтверждение о выполнении операции. Предполагается, что используются корректные запросы на перевод и каналы надежные, поэтому другие подтверждения о выполнении операций не требуются.

Задание

Используя топологию распределенной системы и библиотеку межпроцессного взаимодействия (IPC) из первого этапа лабораторной работы, необходимо реализовать банковскую систему, описанную выше.

Во время выполнения программы осуществляются переводы денег между счетами. При завершении на экран выводится таблица с информацией о балансе каждого счета и полной сумме денег, находящихся на всех счетах, в каждый момент времени $t \in 0, 1, \ldots, T$. Последняя отметка времени T определяется на момент завершения каждого из процессов «С» (при получении соответствующего сообщения от процесса «К», см. далее), $T \leq MAX\_T$.

При запуске программы, как и раньше, указывается число дочерних процессов в полносвязной топологии. Последние X параметров задают начальные балансы для каждого из счетов. Например, следующая команда:

./ра3 -р 3 10 20 30

означает, что в банковской системе три счета (точнее три филиала, каждый из которых обслуживает всего один счет) с идентификаторами 1, 2, 3 и с начальными балансами $10, $20, $30 соответственно.

В реализации банковской системы необходимо использовать скалярное время Лэмпорта. Для обмена отметками времени процессы должны использовать поле *s_local_time* структуры *MessageHeader*, которое должно содержать показания часов процесса-отправителя на момент отправки сообщения. Получение текущей отметки времени осуществляется посредством вызова функции *get_lamport_time()*, которую необходимо реализовать.

Логические часы инициализируются нулем. Считать, что у процессов отсутствуют внутренние события, т.е. линейному упорядочиванию подлежат только события отправки и получения сообщений. Перед выполнением любого события процесс увеличивает показания своих логических часов на единицу. Так при отправке сообщения показания часов сначала увеличиваются, а уже потом в сообщение вкладывается отметка времени. Отправка группового сообщения посредством функции *send_multicast()* продвигает время на единицу вне зависимости от числа отправленных сообщений. В случае, когда два события имеют одинаковые временные метки, линейный порядок определяется на основе идентификаторов процессов, в которых произошли данные события.

Процесс «К» реализуется на основе родительского процесса, процессы «С» — на основе дочерних процессов из этапа №1. После того, как получены сообщения *STARTED* от всех процессов «С», процесс «К» должен вызвать функцию *bank_robbery()*, которая выполняет ряд переводов денег между произвольными процессами «С» посредством вызовов функции *transfer()*. Перевод описывает структура *TransferOrder*, передаваемая сообщением типа *TRANSFER*. При выполнении перевода процесс «К» отправляет сообщение *TRANSFER* процессу «C_{src}», после чего переходит в режим ожидания подтверждения (пустое сообщение типа *ACK*) процессом «C_{dst}» получения перевода. Переводы могут быть инициированы только процессом «К».

После выполнения функции *bank_robbery()* процесс «К» отправляет сообщение *STOP* всем процессам «С» и дожидается получения сообщения DONE от всех дочерних процессов. После этого процесс «К» должен получить от каждого процесса «С» сообщение *BALANCE_HISTORY*, содержащее структуру *BalanceHistory*. Структуры *BalanceHistory* от всех процессов «С» агрегируются в структуру *AllHistory*, которая должна быть использована в качестве аргумента функции *print_history()* перед завершением родительского процесса. Функция *print_history()* реализуется библиотекой, поставляемой вместе с заданием.

«Полезной» работой процесса «С» является ожидание и обработка сообщений двух типов: *TRANSFER* и *STOP*. При получении сообщения *TRANSFER* процесс «C_{src}» после выполнения всех необходимых операций пересылает это сообщение процессу «C_{dst}». Процесс «C_{dst}» обрабатывает сообщение и отправляет сообщение ACK процессу «К». При этом каждый процесс «С» должен хранить в структуре типа *BalanceHistory* информацию о состоянии своего баланса *BalanceState* в каждый момент времени t. Обратите внимание, что если в момент времени t=1 баланс процесса равен $10, а в t=3 ему пришел перевод в $5, то в t=2 следует указать баланс $10. Семантика структур *BalanceHistory* и *BalanceState* подробно описана в заголовочном файле *banking.h*.

Также при подсчете полной суммы денег в каждый момент времени необходимо учитывать состояние каналов между процессами, т.е. сохранять информацию о переводах, которые были отправлены, но еще не были получены. Для этого необходимо заполнять поле *s_balance_pending_in* структуры *BalanceState*. Обратите внимание, что ввиду синхронности протокола данная задача является упрощенным вариантом задачи подсчета полной суммы, рассмотренной на лекции.

При получении сообщения *STOP* процесс «С» переходит к выполнению третьей фазы. Во время выполнения третьей фазы процесс «С» может получать сообщения

«*TRANSFER*» от других процессов «С», при этом гарантируется, что после отправки сообщения *STOP* процесс «К» не выполняет новых переводов. После получения сообщений *DONE* от всех остальных дочерних процессов и перед завершением процесс «С» отправляет процессу «К» сообщение *BALANCE_HISTORY*, содержащее структуру *BalanceHistory*.

В дополнение к событиям из предыдущего этапа (строки форматирования изменены) для процессов типа «С» добавлено два новых события:

• процесс отправил перевод на другой счет (*log_transfer_out_fmt*);

• процесс получил перевод с другого счета (*log_transfer_in_fmt*).

Для реализации асинхронного обмена сообщениями необходимо использовать неблокирующие функции *read()* и *write()*. Как и в предыдущем этапе, в реализации запрещается использовать многопоточность: один процесс — один поток. Кроме того, нельзя использовать разделяемую память, примитивы синхронизации (семафоры и т.п.), функции *select()* и *poll()*. Логирование (*events.log*, *pipes.log* и терминал), как и в предыдущем задании.

Требования к реализации и среда выполнения

Реализацию необходимо выполнить на языке программирования Си с использованием предоставленных заголовочных файлов и библиотеки из архива *pa2345_starter_code.tar.gz*.

Архив содержит следующие файлы:

ipc.h	объявления структур данных и функций для организации межпроцессного взаимодействия. Объявленные функции необходимо реализовать;
banking.h	объявления структур данных, констант и функций, связанных с банковскими операциями. Часть функций реализовано преподавателем;
pa2345.h	форматы строк для логирования;
bank_robbery.c	набор вызовов *transfer()* для тестирования реализации;
libruntime.so	библиотека, реализующая вспомогательные функции, в частности *print_history()*. Инструкции по использованию см. ниже;

Заголовочные файлы содержат большое число важных комментариев, пояснений и рекомендаций. Запрещается их модификация: при автоматической проверке они заменяются на оригинальные.

Файл *bank_robbery.c* содержит функцию *bank_robbery()*. Версия *bank_robbery.c* из архива отличается от той, что используется при проверке задания.

Для использования *libruntime.so* необходимо определить следующие переменные программного окружения:

```
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/path/to/pa3/dir";
### пустая строка
LD_PRELOAD=/full/path/to/libruntime.so ./pa3 -p 2 10 20
```

Использование неблокирующих *read()* и *write()* может потребовать дополнительных изменений в реализации IPC. Новая версия библиотеки IPC должна быть совместима с версией для предыдущего этапа лабораторной работы.

Работа присылается в виде архива с именем *pa3.tar.gz*, содержащим каталог *pa3*. Все файлы с исходным кодом и заголовки должны находиться в корне этого каталога. Среда выполнения — Linux (Ubuntu 14.04, clang-3.5). При автоматической проверке используется следующая команда: *clang -std=c99 -Wall -pedantic *.c –L. -lruntime*. При наличии варнингов работа не принимается. При успешном выполнении запущенные процессы не должны использовать *stderr*, код завершения программы должен быть равен 0.

Этап №3. Алгоритм взаимного исключения Лэмпорта

Введение

Взаимное исключение — требование, согласно которому во время выполнения критической области одного процесса ни один другой процесс не должен выполняться в этой же критической области. Различная скорость выполнения процессов и их число, произвольные задержки передачи сообщений и отсутствие полной информации о состоянии всей системы и общей памяти в распределенных системах делают реализацию взаимного исключения одной из фундаментальных проблем в области распределенных вычислений.

Исходные данные

Следует использовать исходные данные из этапа №1.

Задание

Данная работа выполняется на основе распределенной системы, реализованной на этапе №1. При этом понятие «полезной» работы дочернего процесса определяется следующим образом: каждый дочерний процесс должен *N = process_local_id * 5* раз распечатать сообщение, определенное строкой форматирования *log_loop_operation_fmt*, посредством вызова функции *print()*, входящей в состав прилагаемой библиотеки *libruntime.so*. Обратите внимание, что при формировании строки на основе *log_loop_operation_fmt* нумерация итераций должна выполняться, начиная с единицы, а не с ноля.

В *IPC* из этапа №1 следует внести следующие изменения: вызовы *read()* и *write()* должны быть неблокирующими, а сообщения должны содержать значение скалярных часов отправителя, как на этапе №2. Это фактически означает, что можно использовать без изменений библиотеку *IPC,* реализованную для этапа №2.

При запуске программы с параметром командой строки «*--mutexl*» перед каждым вызовом *print()* процесс должен входить в критическую область, а после вызова выходить из нее, т.е. запрещается выполнять несколько вызовов *print()* в пределах одной критической области. При отсутствии параметра «*--mutexl*» программа должна выполняться без использования критической области, обратите внимание на разницу в выводе программы при использовании критической секции и без нее.

Вход в критическую область выполняется с помощью вызова функции *request_cs(),* выход — *release_cs()*. Обе функции необходимо реализовать самостоятельно, используя алгоритм взаимного исключения Лэмпорта. Для этого определены следующие пустые сообщения: *CS_REQUEST, CS_REPLY* и *CS_RELEASE*. Идентификатор процесса-отправителя данных сообщений определяется по дескриптору канала, через который сообщение получено. Значение локальных часов отправителя извлекается из заголовка сообщения. Семантика сообщений *CS_REQUEST, CS_REPLY* и *CS_RELEASE* подробно описана в лекционных материалах.

Процедура синхронизации процессов при запуске и завершении распределенной системы, как в этапе №1.

Требования к реализации и среда выполнения

Реализацию необходимо выполнить на языке программирования Си с использованием предоставленных заголовочных файлов и библиотеки из архива *pa2345_starter_code.tar.gz*.

Работа присылается в виде архива с именем *pa4.tar.gz*, содержащим каталог *pa4*. Все файлы с исходным кодом и заголовки должны находиться в корне этого каталога. Среда выполнения — Linux (Ubuntu 14.04, clang-3.5). При автоматической проверке используется следующая команда: *clang -std=c99 -Wall -pedantic *.c –L. -lruntime*. При наличии варнингов работа не принимается. При успешном выполнении запущенные процессы не должны использовать *stderr*, код завершения программы должен быть равен 0.