# Цель работы

Целью лабораторной работы является практическое освоение базовых средств API QNX Neutrino и примитива синхронизации условная переменная для управления взаимоисключающим доступом параллельно выполняющихся потоков к совместно используемым ресурсам кода программы.

Продолжительность работы – 2 часа.

# Теоретическая часть по базовым средствам API QNX Neutrin

Проблема синхронизированного доступа к критическим секциям кода является одной из самых практически важных проблем конкурентного программирования. Известны случаи серьёзных отказов систем реального времени из-за нарушения взаимоисключения при использовании задачами совместных ресурсов. Проблема правильно организованного взаимоисключения была одной из первых проблем, которую подробно исследовали теоретики, в результате чего были разработаны широко применяемые в настоящее время алгоритмы и примитивы синхронизации. На уровне флагов в пользовательских программах можно отметить, например, алгоритм Петерсона, в операционных системах широко применяются мьютексы (или мутексы) и семафоры.

Главное назначения мьютекса следует из его названия mutual exclusion («взаимное исключение»). Условно можно рассматривать мьютекс как глобальную переменную-флаг, принимающую два значения – поднят (мьютекс захвачен задачей, доступ закрыт)/опущен (мьютекс освобождён задачей, доступ открыт). Захват и освобождение мьютекса - всегда неделимые операции, именно это свойство обеспечивает взаимоисключение. В отличие от обычной переменной мьютекс всегда имеет своего владельца – только задача, захватившая мьютекс, может его освободить. Это обеспечивает мьютексу очень важное для систем реального времени свойство – возможность реализации различных протоколов борьбы с инверсией приоритета. Зная задачу-владельца

мьютекса, операционная система имеет возможность временно поднять приоритет владельца в тех случаях, когда более приоритетная задача заблокирована в ожидании того же мьютекса.

Семафор можно рассматривать как неотрицательную глобальную переменную, изменяемую, как и мьютекс, неделимой операцией. Счётный семафор может принимать несколько целых значений. При каждом захвате семафора задачей его значение неделимой операцией уменьшается на 1, при освобождении – увеличивается на 1. Это даёт возможность использовать такой семафор для многих разновидностей синхронизации задач, например, в проблемах поставщики-потребители, обедающие философы и т.п. Частным случаем семафора является двоичный, принимающий только значения 0 и 1. Этот вид семафора можно использовать так же, как мьютекс – для взаимоисключающего доступа к критическим секциям кода. Однако, в отличие от мьютекса, семафор владельца не имеет – захваченный одной задачей, он может быть освобождён другой задачей. С одной стороны, это свойство семафора делает его весьма гибким средством синхронизации, с другой стороны, при использовании как аналога мьютекса, не позволяет бороться с инверсией приоритета.

## МЬЮТЕКСЫ И СЕМАФОРЫ ОПЕРАЦИОННОЙ СИСТЕМЫ QNX NEUTRINO

ОС QNX Neutrino имеет развитый интерфейс средств (или примитивов) синхронизации потоков, в том числе мьютексы с различными характеристиками, и семафоры. Соответствующий API включает как функции и утилиты по стандарту POSIX, так и QNXверсии этих функций. В составе API – функции по инициализации, задании атрибутов, захвату-освобождению и уничтожению этих примитивов синхронизации и атрибутов. Ниже кратко описаны некоторые функции библиотеки и атрибуты мьютексов (стандарт POSIX 1003.1 THR, POSIX 1003.1 THR TPP|TPI, и проприетарные), которые могут использоваться в лабораторной работе. Подробное описание функций с примерами см. в справочной системе.

* pthread\_mutex\_init(…) – инициализация мьютекса с заданными атрибутами. Аргументами функции являются указатель на мьютекс и атрибутную запись. Если мьютекс размещён в памяти статически и нерекурсивен, инициализацию можно провести с помощью макроса PTHREAD\_MUTEX\_INITIALIZER, для программ лабораторной работы этот способ предпочтителен.
* pthread\_mutexattr\_init(…) - – инициализация атрибутного объекта мьютекса. С помощью атрибутов можно задать (используются ф-ции pthread\_mutexattr\_set\*()):
* “тип” мьютекса, определяющий главным образом, что произойдёт с потоком, пытающимся выполнить в тех или иных особых условиях захват/освобождени мьютекса. Например, можно задать возврат функции освобождения мьютекса с ошибкой в случае, если мьютекс уже освобождён,
* схему изменения динамического приоритета владельца мьютекса для противодействия инверсии приоритета,
* значение максимального приоритета мьютекса в случае схемы priority ceiling,
* возможность рекурсивного захвата мьютекса его владельцем,
* доступность мьютекса из потоков, расположенных в процессах, отличных от того, где находится владелец мьютекса. Для этого мьютекс должен находиться в области памяти, доступной одновременно разным процессам. Работе с разделяемой памятью посвящена последующая лабораторная работа.

Если атрибуты мьютекса не заданы, по умолчанию используется протокол наследования приоритета, и запрет на рекурсивный захват мьютекса.

* pthread\_mutex\_lock(…) – захватить мьютекс
* pthread\_mutex\_unlock(…) – освободить мьютекс
* pthread\_mutex\_trylock()- попытка захватить мьютекс без блокирования потока в том случае, когда мьютекс уже занят.

Пример программы с мьютексами приведен в справке по функции pthread\_mutex\_lock().

QNX Neutrino поддерживает два вида семафоров – неименованные и именованные, оба вида относятся к категории счётных. API семафоров соответствует стандарту POSIX 1003.1 SEM. Неименованный семафор наиболее просто используется в пределах одного процесса. В случае синхронизации потоков из нескольких процессов неименованный семафор должен быть размещён в общей области памяти. В отличие от этого именованный семафор при создании регистрируется в файловой системе, и после этого становится доступен любому процессу. Функции для работы с семафорами:

* sem\_init(…)- инциализировать неименованный семафор. В качестве аргументов задаются указатель на семафор, флажок использования в пределах одного или разных процессов, и начальное значение (если оно равно 0, любой поток, пытающийся захватить семафор, сразу на нём заблокируется),
* sem\_open(…) – создать или получить доступ к уже созданному именованному семафору. Аргументы:
* имя семафора – строка символов. Регистрируется и становится видным в каталоге /dev/sem,
* флаги, управляющие доступом к семафору – если семафор не существует – создать его, иначе подключиться к существующему. Если семафор создаётся, задаются два дополнительных аргумента функции sem\_open():
* разрешения доступа по чтению, записи и выполнению для пользователя, группы и остальных, аналогично разрешениям для регулярных файлов,
* начальное значение.
* sem\_close(…)- закрыть именованный семафор, освободив занятые им системные ресурсы,
* sem\_destroy(…) - удалить из системы неименованный семафор,
* sem\_unlink(…) - удалить из системы именованный семафор. После вызова этой функции семафор остаётся доступным для процессов, которые открыли его, но исчезает сразу, как только последний из этих процессов вызовет sem\_close()
* sem\_wait(…) – ожидать на семафоре любого вида. Если значение семафора >0, оно уменьшается на 1, и поток продолжает выполнение кода, следующего за sem\_wait(). Если значение семафора в момент вызова функции равно 0, поток блокируется до тех пор, пока кто-то из конкурентов не освободит семафор,
* sem\_trywait(…) – уменьшить значение любого вида семафора на 1, если оно больше 0, в противном случае не блокировать вызывающий поток,
* sem\_post(…)- увеличить на 1 значение семафора любого вида.

Практически важной особенностью семафоров является возможность ложного уменьшения его значения (без вызова функции sem\_post()), в связи с чем рекомендуется ожидать на семафоре следующей конструкцией:

while (sem\_wait(sem) && (errno == EINTR)) {ничего не делать};

Войти в критическую секцию.

Помимо вышеуказанных функций, API QNX Neutrino содержит инструментарий для задания таймаутов на ожидание мьютекса и семафора, поскольку в системах реального времени недопустимо, если некоторая задача будет находиться в блокированном состоянии очень долго. Средства задания таймаутов разбираются в последующей лабораторной работе.

Какой из рассмотренных примитивов сихронизации наиболее подходит к решению задачи взаимоисключающего доступа к критической секции? В однопроцессной программе наиболее эффективным является мьютекс. Захват свободного мьютекса не требует исполнения кода ядра, он реализуется атомарной инструкцией “сравнение с обменом” на процессорах x86 и “load/store” инструкцией на RISC процессорах, поэтому выполняется очень быстро. Вход в ядро требуется только в тех случаях, когда мьютекс уже захвачен потоком с отличающимся приоритетом. Неименованный семафор – более сложная структура, поэтому даже в простейших случаях реализации взаимного исключения работает медленнее, чем мьютекс. Ещё более сложным и медленным, как объект файловой системы, является именованный семафор.

## ПРОГРАММА – ИЛЛЮСТРАЦИЯ

Программа иллюстрирует применение именованного семафора с анализом возможных ошибок, возвращаемых библиотечными функциями

#include <fcntl.h>

#include <semaphore.h>

#include <stdlib.h>

#include <errno.h>

#include <stdio.h>

main()

{ char \*sema\_name = "/named\_sem";

// Семафор будет зарегистрирован в каталоге /dev/sem) sem\_t \*sem; pid\_t child\_pid; setbuf(stdout, NULL); errno = EOK; sem\_unlink(sema\_name);

/\* Удалить семафор, если он остался после предыдущих

запусков программы\*/ printf( "Error unlinking semaphore: %s\n", strerror( errno)); errno = EOK; sem = sem\_open(sema\_name, O\_CREAT | O\_EXCL, S\_IRWXU, 1);

// Создать семафор printf( "Error opening semaphore: %s\n", strerror( errno)); printf("\*\n");

/\* Запустить дочерний процесс и выполняться параллельно с ним \*/ if( child\_pid=fork()) { //PARENT while (sem\_wait(sem) && (errno == EINTR)); printf("THIS IS PARENT PROCESS, pid=%d\n", getpid()); sem\_post(sem);

waitpid(child\_pid, NULL, 0); //Ожидать завершения дочери

} else

{ //CHILD

while (sem\_wait(sem) && (errno == EINTR)); printf("this is child process, pid=%d\n", getpid()); sem\_post(sem); system("pidin -p lab3\_namsem.out");

//sem\_post(sem); sem\_close(sem); // sem\_unlink(sema\_name);

}

}

# ТЕОРЕТИЧЕСКАЯ ЧАСТЬ ПО примитиваМ синхронизации условная переменная

Нередко при параллельной работе с общими данными, требующими эксклюзивного доступа каждого из потоков, возникает задача прервать выполнение критической секции одного из потоков, если нужные второму потоку данные уже получены, и передать управление второму потоку, разблокируя этот поток в его критической секции, и не снимая при этом защиты мьютексами критических секций ни одного из потоков. Примитив синхронизации условная переменная, или переменная условия (condition variable) служит как раз для этих целей.

Условная переменная обязательно связана с мьютексом, причём одна условная переменная может работать в критических секциях, защищённых разными мьютексами, и, наоборот, в одной критической секции можно использовать несколько различных условных переменных, сопоставив их с мьютексом. Такие возможности делают механизм условных переменных очень гибким, пригодным для решения самых разных задач синхронизации.

Условная переменная реализована непосредственно в ядре операционной системы, и, наряду с мьютексом, является основой, на которой построены другие примитивы синхронизации ОС QNX Neutrino – барьеры, спящие блокировки, и т.д. Это легко увидеть в выводе утилиты pidin без параметров - для многих потоков системных процессов вы встретите состояния блокировки в ядре MUTEX и CONDVAR, может встретиться также состояние SEM, никаких других состояний блокировки в ядре на примитивах синхронизации вы не увидите.

## API ОПЕРАЦИОННОЙ СИСТЕМЫ QNX NEUTRINO ДЛЯ РАБОТЫ С УСЛОВНОЙ ПЕРЕМЕННОЙ

Этапы жизненного цикла переменной условия такие же, как у мьютекса и других примитивов синхронизации – создание, задание атрибутов, использование непосредственно по назначению, уничтожение. Так же, как почти все примитивы синхронизации QNX Neutrino, условную переменную можно использовать и в пределах одного процесса, и в разных процессах, для этого она совместно со связанными с ней мьютексами должна размещаться в разделяемой памяти. Ниже кратко описаны необходимые для лабораторной работы библиотечные функции. Подробное описание функций с примерами см. в справочной системе.

* pthread\_cond\_init(…) – инициализация условной переменной с заданными атрибутами. Аргументами функции являются указатель на условную переменную и атрибутную запись. Если условная переменная размещена в памяти статически, инициализацию можно провести с помощью макроса PTHREAD\_COND\_INITIALIZER, для программ лабораторной работы этот способ предпочтителен.
* pthread\_condattr\_init(…) – инициализация атрибутного объекта условной переменной. Имеется возможность задать (используется ф-ция pthread\_condattr\_setpshared()) вариант использования переменной в одном или нескольких процессах. Если атрибут условной переменной не задан, по умолчанию условная переменная доступна только потокам в том же процессе, что и поток, создавший переменную,
* pthread\_cond\_wait(…) – ожидать на условной переменной. Поток заблокирован в состоянии ожидания до тех пор, пока другой поток не просигналит по условной переменной. Специфика работы фунции pthread\_cond\_wait(…) состоит в том, что она прозрачно для программиста освобождает связанный с условной переменной мьютекс, осуществляет ожидание, и снова захватывает мьютекс перед возвращением. Тем самым программист освобождается от явного написания в своей программе циклического повторения освобождений мьютекса, проверки условия, и снова захватов мьютекса до тех пор, пока условие не будет выполнено. Это делает механизм условной переменной очень удобным.

Поток, ожидающий на условной переменной, разблокируется только при одновременном наступлении двух событий:

* поступил сигнал по условной переменной, и
* нарушено условие, при котором заблокированный поток должен находиться в состоянии ожидания

В коде программы это выглядит так:

while (условие) pthread\_cond\_wait(…);

Обратите внимание на проверку условия в цикле while(). Рекомендуется всегда делать так, поскольку возможны ложные разблокирования условной переменной аналогично тому, как это может произойти с семафором.

* pthread\_cond\_signal(…) – разблокировать ожидающий на условной переменной поток. Этот поток – либо с наивысшим приоритетом в очереди к условной переменной, либо (при совпадении приоритетов) – тот, который ждёт дольше всех,
* pthread\_cond\_broadcast(…) – разблокировать все ожидающие на условной переменной потоки. Потоки разблокируются либо в порядке приоритетов, либо (при совпадении приоритетов) – в порядке FIFO,
* pthread\_condattr\_destroy(...)- уничтожить атрибутный объект условной переменной,
* pthread\_cond\_destroy(…)- уничтожить условную переменную.
* Пример программы с переменными условия приведён в справке по функции pthread\_cond\_wait().

Помимо вышеуказанных функций, API QNX Neutrino содержит инструментарий для задания таймаутов на ожидание условной переменной. Средства задания таймаутов разбираются в последующей лабораторной работе.

## ПРОГРАММА – ИЛЛЮСТРАЦИЯ

Программа иллюстрирует применение условной переменной в простой задаче “один производитель – один потребитель”.

#include <stdio.h>

#include <pthread.h>

int data\_ready = 0; pthread\_mutex\_t mutex = PTHREAD\_MUTEX\_INITIALIZER; pthread\_cond\_t condvar = PTHREAD\_COND\_INITIALIZER; ///////////////////////////////////////////////////////////// void \*consumer ()

{ while (1) { pthread\_mutex\_lock (&mutex); while (!data\_ready) pthread\_cond\_wait (&condvar, &mutex);

// обработать полученные данные printf ("consumer: got data from producer\n"); data\_ready = 0; pthread\_cond\_signal (&condvar); pthread\_mutex\_unlock (&mutex);

}

}

///////////////////////////////////////////////////////////// void \*producer ()

{

while (1) {

// Получить данные с аппаратуры. Симулируется sleep (1) sleep (1); printf ("producer: got data from h/w\n"); pthread\_mutex\_lock (&mutex); while (data\_ready) pthread\_cond\_wait (&condvar, &mutex); data\_ready = 1; pthread\_cond\_signal (&condvar); pthread\_mutex\_unlock (&mutex);

}

}

///////////////////////////////////////////////////////////// main ()

{

printf ("Starting consumer/producer example...\n");

// create the producer and consumer threads pthread\_create (NULL, NULL, producer, NULL); pthread\_create (NULL, NULL, consumer, NULL);

sleep (20); // let the threads run for a bit

}

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Сравните между собой временны́ е затраты на захват и освобождение рассмотренных примитивов синхронизации на вашем лабораторном компьютере. Для этого:

* Составьте простую однопоточную программу, в которой 10000 раз в цикле захватывается и сразу освобождается
  + a) мьютекс,
  + неименованный семафор,
  + именованный семафор.
* Время, затрачиваемое на выполнение цикла во всех трёх случаях замерьте, используя отсчёты счётчика тактов процессора на основе системного вызова ClockCycles(). Пример приведён в справочной системе в теме по макросу SYSPAGE\_ENTRY()
* Чтобы устранить погрешность от неодинакового хода “процессорных часов” на мультипроцессорных / мультиядерных платформах, программу запускайте на процессоре/ядре 0, используя утилиту on с ключом –C: “on –

C 0 имя\_программы”. Для каждого примитива синхронизации прогоните программу несколько раз и запишите результаты в файл. Сделайте для себя вывод, насколько стабильно по времени работает ваша система.

1. Составьте программу по индивидуальному заданию.

Индивидуальное задание состоит из двух этапов.

Вначале главный процесс запускает NP\* дочерних процессов, используя библиотечные функции FP\*. Дочерние процессы после запуска работают параллельно с родителем. Приоритеты (10) и дисциплины планирования всех процессов (RR) берутся по умолчанию. Родительский процесс выводит на экран строку “THIS IS PARENT PROCESS PID=\*\*\*\*\*\*” (здесь заглавные буквы обязательны). Каждый дочерний процесс выводит на экран строку “this is child process pid=\*\*\*\*\*\*” (все буквы строчные). Замещённый символами “\*” фрагмент строки – реальный идентификатор процесса. Строка выводится в цикле посимвольно, вывод соседних символов производится через случайный промежуток времени, задаваемый по варианту TS\*. Наличие случайного промежутка времени между выводами символов приведёт к тому, что возникнет искусственно организованный эффект гонок, и на экране появится “каша” из символов. На первом этапе выполнения индивидуального задания необходимо синхронизировать вывод строк целиком каждого из процессов с помощью семафора по варианту SEM\*. Пока идёт вывод строк, в отдельном терминале запустить pidin и вывести состояния процессов.

Дочерние процессы после вывода своих строк завершаются, родительски процесс приступает ко второму этапу. Поток main создаёт NT\* дочерних потоков и засыпает на 30 секунд. Аналогично этапу 1 каждый дочерний поток посимвольно со случайными временными промежутками между символами по варианту TS\* выводит строку “This is child thread tid=\*\*\*\*\*\*”, после чего завершается. С помощью синхронизации мьютексами необходимо устранить эффект гонок и синхронизировать вывод строк целиком. Пока идёт вывод строк, в отдельном терминале запустить pidin и вывести состояния потоков.

Варианты задания по отдельным пунктам:

Количество дочерних процессов:

|  |  |
| --- | --- |
| NP1 | NP2 |
| 2 | 3 |

Способ запуска дочерних процессов

|  |  |
| --- | --- |
| FP1 | FP2 |
| spawnl() | fork() |

Организация случайных интервалов времени

|  |  |
| --- | --- |
| TS1 | TS2 |
| delay(случайный  интервал в мс) | “Паразитные вычисления” в цикле со случайным числом повторений |

Случайные интервалы подбираются на каждом компьютере отдельно так, чтобы эффект гонок был очевиден.

Вид синхронизирующих семафоров

|  |  |
| --- | --- |
| SEM1 | SEM2 |
| Неименованный | Именованный |

Количество дочерних потоков:

|  |  |
| --- | --- |
| NT1 | NT2 |
| 3 | 4 |

Варианты индивидуальных заданий:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| № варианта | Количество дочерних  процесса | Запуск дочерних  процессов | Случайный интервал  времени | Синхронизирующий  семафор | Количество дочерних  потоков |
| 1 | NP1 | FP1 | TS1 | SEM1 | NT1 |
| 2 | NP1 | FP1 | TS1 | SEM1 | NT2 |
| 3 | NP1 | FP1 | TS2 | SEM1 | NT1 |
| 4 | NP1 | FP1 | TS2 | SEM1 | NT2 |
| 5 | NP1 | FP1 | TS1 | SEM1 | NT1 |
| 6 | NP1 | FP2 | TS1 | SEM2 | NT2 |
| 7 | NP1 | FP2 | TS2 | SEM2 | NT1 |
| 8 | NP1 | FP2 | TS2 | SEM2 | NT2 |
| 9 | NP1 | FP2 | TS1 | SEM2 | NT1 |
| 10 | NP1 | FP2 | TS1 | SEM2 | NT2 |
| 11 | NP2 | FP1 | TS2 | SEM1 | NT1 |
| 12 | NP2 | FP1 | TS2 | SEM1 | NT2 |
| 13 | NP2 | FP1 | TS1 | SEM1 | NT1 |
| 14 | NP2 | FP1 | TS1 | SEM1 | NT2 |
| 15 | NP2 | FP1 | TS2 | SEM1 | NT1 |
| 16 | NP2 | FP2 | TS2 | SEM2 | NT1 |
| 17 | NP2 | FP2 | TS1 | SEM2 | NT2 |
| 18 | NP2 | FP2 | TS1 | SEM2 | NT1 |
| 19 | NP2 | FP2 | TS2 | SEM2 | NT2 |
| 20 | NP2 | FP2 | TS2 | SEM2 | NT1 |

1. Задание разделяется для четных и нечетных вариантов и включают составление двух программ.

Нечетные варианты: Чередование потоков в заданном порядке. Однопроцессная программа содержит NT потоков, включая поток main, с атрибутами по умолчанию. Каждый поток выполняет свою поточную функцию в бесконечном цикле. В поточной функции выводится идентификатор потока tid, далее поток засыпает на 1 секунду, и, проснувшись, передаёт управление потоку с идентификатором по варианту TID\*. Программа должна воспроизвести несколько циклов заданного чередования потоков, после чего принудительно завершается.

Четные варианты: Задача производители-потребители. В однопроцессной программе NP\* потоков выступают в качестве производителей, и NC\* потоков являются потребителями. Все потоки имеют атрибуты по умолчанию, все поточные функции работают в бесконечном цикле. Производители выполняются по очереди: 1, 2, …, NP\*, Каждый производитель генерирует случайное число в диапазоне от 10 до 99, выводит его на экран вместе со своим tid, сообщает об этом потребителям, и засыпает на 0.3 секунды. Каждый из потребителей ждёт числа, удовлетворяющего условиям, зависящим от количества NC\* (см. таблицу ниже). Если очередное сгенерированное производителем число удовлетворяет потребителя, он увеличивает счётчик принятых чисел на 1, выводит свой tid, принятое от производителя число, и значение счётчика, и блокируется в ожидании следующего подходящего числа. Программа завершается принудительно по Ctrl-C.

Варианты задания по отдельным пунктам.

Программа для нечетных вариантов.

Количество потоков NT равно увеличенному на 4 остатку от деления на 7 числа, составленного из цифр дня и месяца рождения студента. Пример. День рождения 15, месяц 8, NT = 158%7 + 4 = 4 + 4 = 8. Порядок чередования потоков

|  |  |  |  |
| --- | --- | --- | --- |
| TID1 | TID2 | TID3 | TID4 |
| 1, 2, … , NT | NT, NT-1, … , 1 | Сначала с нечётным tid, потом с чётным | Сначала с чётным tid, потом с нечётным |

Программа для четных вариантов

Количество потоков-производителей

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| NP1 | NP2 | NP3 | NP4 | NP5 |
| 2 | 3 | 4 | 5 | 6 |

Количество потоков-потребителей

|  |  |  |  |
| --- | --- | --- | --- |
| NC1 | NC2 | NC3 | NC4 |
| 3 | 4 | 5 | 6 |

Диапазон чисел, удовлетворяющих потребителя

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Потребитель | Количество потребителей NC | | | |
| NC1=3 | NC2=4 | NC3=5 | NC4=6 |
| 1 | 10…30 | 10…22 | 10…28 | 10…25 |
| 2 | 31…60 | 23…46 | 29…46 | 26…40 |
| 3 | 61…90 | 47…67 | 47…62 | 41…55 |
| 4 |  | 68..99 | 63…80 | 56…70 |
| 5 |  | 81…99 | 71…85 |
| 6 |  | 86…99 |

Индивидуальные варианты количества производителей/ потребителей для программы 2:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
| NP1 | NP1 | NP1 | NP1 | NP2 | NP2 | NP2 | NP2 | NP3 | NP3 | NP3 | NP3 | NP4 | NP4 | NP4 | NP4 | NP5 | NP5 | NP5 | NP5 |
| NC1 | NC2 | NC3 | NC4 | NC1 | NC2 | NC3 | NC4 | NC1 | NC2 | NC3 | NC4 | NC1 | NC2 | NC3 | NC4 | NC1 | NC2 | NC3 | NC4 |

# Контрольные вопросы

1. Чем принципиально различаются мьютексы и семафоры при использовании их для взаимного исключения?
2. Какой из рассмотренных в лабораторной работе примитивов синхронизации работал быстрее всего? Во сколько раз его скорость отличалась от самого медленного примитива?
3. В каком каталоге размещается именованный семафор после его создания?
4. Будет ли доступен именованный семафор для отрывших его процессов после вызова функции sem\_unlink()?
5. Сколько семафоров требуется для организации строго чередующегося выполнения двух потоков?
6. Какие дополнительные средства API нужно использовать помимо мьютексов, чтобы обеспечить строго чередующееся выполнение двух потоков?
7. Опишите словами, как можно реализовать примитив синхронизации барьер, используя условную переменную.
8. Какие нюансы поведения делают условную переменную похожей на семафор?
9. Какие события должны совпасть, чтобы поток, ожидающий на условной перменной, разблокировался?
10. В чём разница между библиотечными функциями pthread\_cond\_signal() и pthread\_cond\_broadcast()?
11. В каком порядке разблокируются потоки, ожидающие выполнения одного и того же условия на переменной условия?