# Цель работы

Целью лабораторной работы является практическое знакомство с возможностями self-hosted варианта ОС РВ QNX Neutrino и освоение приёмов работы в режиме командной строки и освоение базовых средств API QNX Neutrino для создания параллельно выполняемых задач.

Продолжительность работы – 2 часа.

# Теоретическая часть

## Необходимые сведения об интерфейсе пользователя ОС QNX NEUTRINO

Современные универсальные (предназначенные для широкого круга практических задач) операционные системы реального времени имеют развитый набор средств разработки, включающий как интегрированные среды (IDE) с графическим интерфейсом пользователя, так и командные оболочки с текстовым интерфейсом. Тестово-отладочное выполнение разрабатываемых программ реального времени может осуществляться как на инструментальной (host) машине, если её аппаратная архитектура совпадает с архитектурой целевой (target) машины (возможно также применение эмулятора целевой машины), так и непосредственно на подсоединённой отдельной целевой машине. В лабораторных работах, предназначенных для знакомства с основными средствами программирования систем реального времени операционной системы QNX Neutrino (включая API по стандарту POSIX), используется наиболее простой и быстрый в освоении вариант - self-hosted (инструментальная и целевая машина – на одном физическом компьютере), и режим разработки в командной строке с использованием необходимых утилит графического режима оконной среды Photon. Self-hosted машина QNX Neutrino может быть установлена прямо на “голую” аппаратную платформу, либо запускаться в среде той или иной виртуальной машины, для выполнения лабораторных работ это значения не имеет.

После включения компьютера, ввода логина и пароля пользователя (сообщается студентам преподавателем) и загрузки ОС QNX необходимо запустить графическую среду Photon (если ОС сконфигурирована в варианте загрузки текстового терминала), набрав команду ph. Экран среды Photon выглядит так (рис.1):

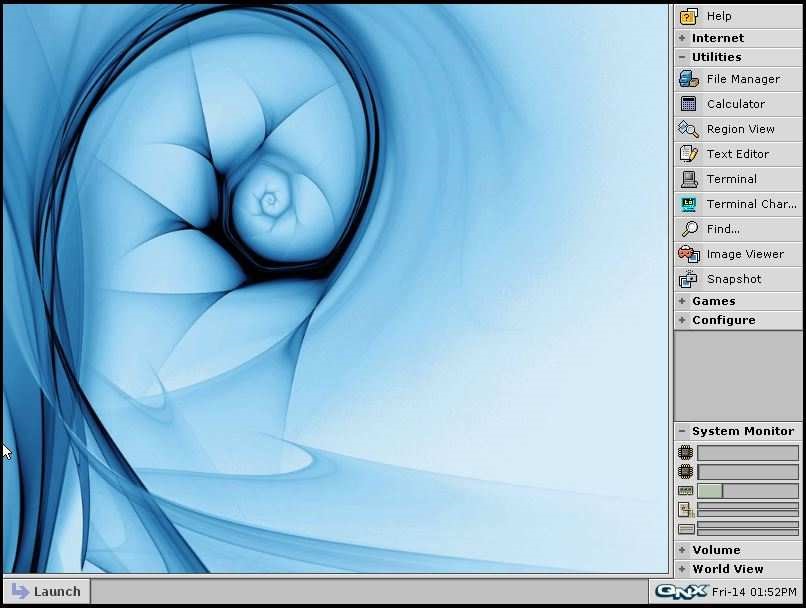


Рис.1 Общий вид окна графической среды Photon

Оконная среда Photon очень схожа с графическим интерфейсом ОС Windows, в частности, работают самые ходовые комбинации клавиш клавиатуры и действия элементов управления мыши. Хотя в доступных дистрибутивах ОС QNX не русифицирована, у студентов это не вызывает трудностей. По сравнению с ОС Windows среда Photon имеет одно важное исключение, которое следует помнить – при копировании текстовых данных из графического окна в текстовое окно терминала, и наоборот, используются комбинации клавиш Ctrl-C, Ctrl-Alt-V (вместо Ctrl-V).

В правой нижней части экрана расположена область Системного Монитора, визуализирующего загрузку аппаратуры при выполнении программ. Если среда Photon начинает тормозить, первым делом обратите внимание на полосы загрузки процессоров системного монитора – возможно, ошибочная программа съедает все ресурсы процессоров, её нужно принудительно завершить (см. команды ниже).

Необходимый для лабораторных работ набор утилит среды Photon расположен в вертикальной правой полосе экрана, называемой “полка” (shelf):

* Help – справочная система ОС QNX, очень подробная, содержащая все необходимые сведения по утилитам, функциям библиотеки, и заготовки программ для лабораторных работ,
* File Manager – аналог проводника Windows с интуитивно понятным интерфейсом,
* Text Editor - текстовый редактор, аналог блокнота Windows с интуитивно понятным интерфейсом. Удобным свойством редактора является то, что он по умолчанию открывает файлы с расширениями “c” и “txt” при двойном щелчке мыши по имени файла в файловом менеджере. В операционной системе QNX 6.5 в редакторе среды Photon доступен русский шрифт, при этом весь интерфейс Photon и терминала, а также любые вводимые в терминале символы не могут содержать русских букв,
* Terminal – текстовый терминал, в котором реализован интерфейс командной оболочки, аналогичный окну командной строки Windows.
* Find – утилита с графическим интерфейсом для поиска объектов файловой системы по их типу и фрагментам имени.

Программы лабораторных работ набираются на языке Си в редакторе, компилируются встроенным GNU-компилятором gcc, запускаются в окне командной оболочки, и выдают сообщения в этом же окне. Расширения имён файлов с исходным тестом на языке Си – “.c”. Исполняемые программы рекомендуется именовать по имени файла с исходным текстом и расширением имени файла “.out”. Командная оболочка ksh (Korn shell), запускаемая на QNX по умолчанию, в целом соответствует стандарту POSIX, что облегчает работу в её среде для студентов, изучавших основы операционной системы Linux.

Программы лабораторных работ рекомендуется сохранять в подкаталогах каталога /home/student/, каждый подкаталог рекомендуется именовать датой проведения лабораторной работы. На рис. 2 в файловом менеджере открыт каталог “14-02-2017”, содержащий две программы с исходным текстом prg1.c и prg2.c, и две выполняемые программы – prg1.out и prg2.out. Свойства каждого файла можно посмотреть, вызвав на имени файла контекстное меню правой кнопкой мыши, и выбрав пункт Inspect. В открывшемся окне свойств (см. рис. 2) можно установить разрешения на действия с файлом (Permissions) различных групп пользователей и права доступа (Access mode) к файлу.

В каталоге /fs располагаются файловые системы устройств /разделов диска, отличных от того, в котором установлена ОС QNX. В этот каталог автоматически монтируются файловые системы установленных дисков системы с файловой системой FAT32 (на рис.2 – hd0-dos), и съёмные USB-флэшки после их подключения (на рис.2 – usb0).

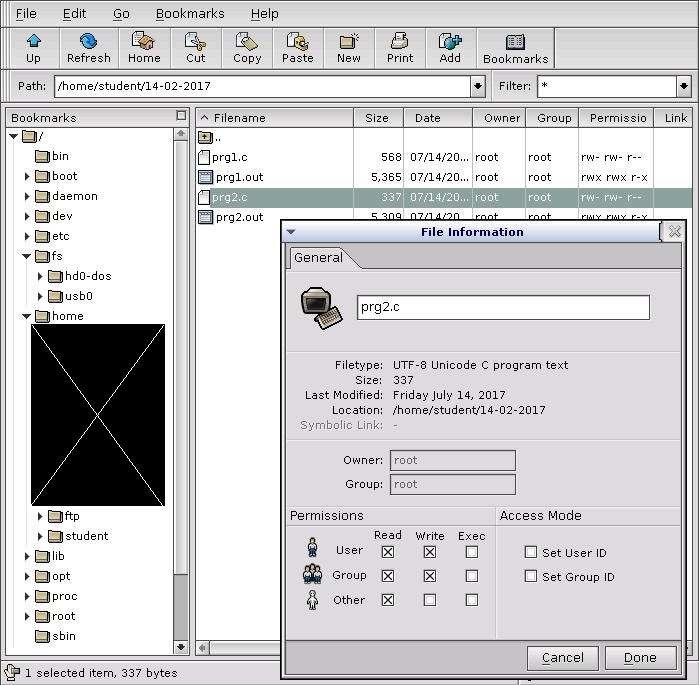


Рис.2 Окна файлового менеджера и свойств файла

При пользовании справочной системой следует правильно задавать условия поиска, поставив флажки в нужных местах. На рис.3 задан вывод справочных материалов, содержащих ключевое слово “timer” в заголовках тем, причём слово timer может присутствовать в любом регистре и не должно быть частью другого слова.

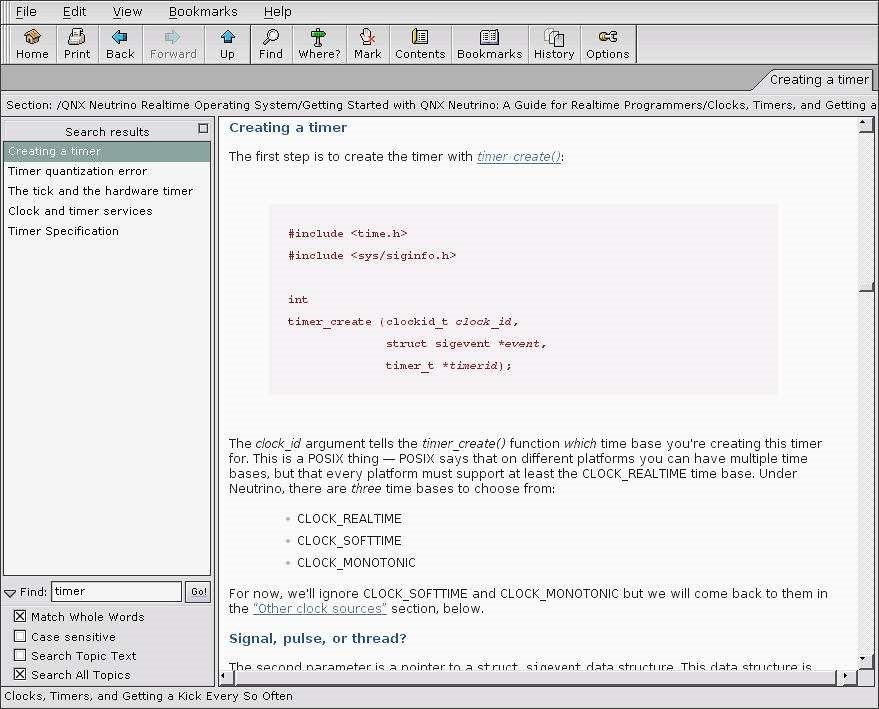


Рис.3. Окно справочной системы

## Необходимые команды оболочки KSH и утилиты ОС

Команды оболочки вводятся в окне терминала после приглашения, которое по умолчанию обозначается знаком # для суперюзера и $ для рядового пользователя. Автозавершение вводимых команд, если оболочке есть, что предложить, доступно по двойному нажатию Esc (подробности см. в справке по ключевому слову “shell”). В ряде случаев удобно в одной строке вводить несколько команд, объединяя их символом “;”, а также использовать конвейер и перенаправление ввода/вывода. Каждая команда имеет несколько режимов работы, задаваемых ключами, каждый ключ имеет буквенное обозначение, предшествующее знаком “-” Выполнение любой команды можно прекратить комбинацией клавиш Ctrl-C. Как общепринято в интерфейсе командной строки, все команды текущего сеанса сохраняются в буфере, последовательно перемещаться по истории команд можно клавишами клавиатуры ↑ и ↓. Вывод на экран всех ранее введённых команд можно осуществить командой fc с ключом “-l” (“эл”). Прокрутка экрана терминала осуществляется клавишами Ctrl-Alt-↑↓.

При проведении лабораторных работ чаще всего используются следующие команды оболочки:

* clear очистка экрана,
* pwd вывод текущего каталога,
* ls вывод содержимого текущего каталога,
* cd dir смена текущего каталога на dir,
* cat file вывод содержимого файла на экран,
* less file вывод содержимого файла на экран постранично.

Из большого числа утилит операционной системы самыми полезными являются следующие:

* slay завершает выполнение заданного идентификатором или именем процесса (программы), посылая ему по умолчанию сигнал SIGTERM. В примере на рис.4 программа prg2.out, запущенная в терминале ttyp0, принудительно завершается командой slay prg2.out, введённой в терминале ttyp1.
* kill завершает (убивает) выполнение заданного идентификатором или именем процесса (программы). Удобно применять в тех случаях, когда другие средства принудительного завершения не работают. В этом случае утилитой kill процессу посылается безальтернативный сигнал-убийца SIGKILL : kill -sigkill 100, где 100 – pid (идентификатор) убиваемого процесса.
* pidin выводит полную информацию о состоянии всех (если вводится без параметов) или заданных параметрами (в частности, c ключ -p) процессов. Как и другие утилиты, pidin имеет большое число режимов работы. На рис.5 показан фрагмент вывода утилиты pidin без параметров.

На рис.5 pid – идентификатор процесса, tid - идентификатор потока в процессе, name – имя процесса, prio – заданный приоритет потока, STATE – состояние потока, Blocked – обозначение в системе блокировки, на которой поток приостановлен (RECEIVE – поток заблокирован в ожидании сообщения по каналу 1).

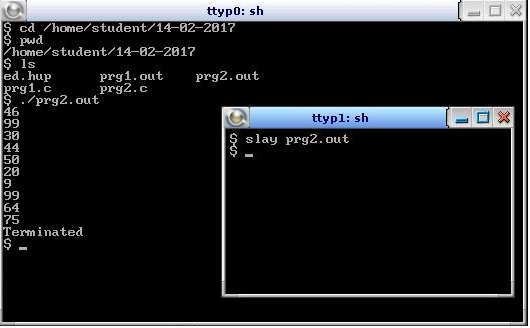


Рис.4 Окна терминалов с командами

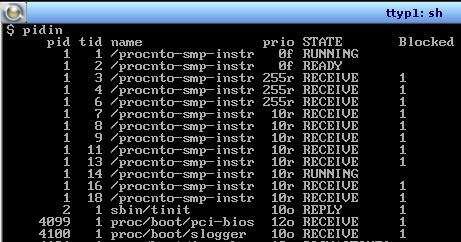


Рис.5. Фрагмент вывода утилиты pidin

Если ввести в терминале команду pidin –p prg2.out, получим вывод

pid tid name prio STATE Blocked

1221258 1 ./prg2.out 10o NANOSLEEP

Очевидно, для применения к уже запущенным из терминалов вашим программ утилиты slay, kill и pidin следует запускать из другого, параллельно запущенного, терминала.

• find, xargs, grep используются в комплексе для поиска в файловой системе объектов по маске имён (утилита find), содержащих заданные подстроки текста (утилита grep). Пример - в каталоге /home/student и всех вложенных каталогах найти файлы все файлы с расширением имени “.c” (find . -name '\*.c'), найденные файлы передать по конвейеру утилите xargs, которая сформирует список аргументов и запустит утилиту grep , отыскивающую в файлах заданную подстроку “rand”. Результаты поиска сохранить в файле rand\_find.txt:

Команды:

$ cd /home/student

$ find . -name '\*.c' | xargs grep -F rand >/home/student/rand\_find.txt

Содержимое файла rand\_find.txt с найденными вхождениями “rand”:

./14-02-2017/prg1.c:// Из справки по srand() - start a new sequence of pseudorandom integers

./14-02-2017/prg1.c: srand( (int) time( NULL ) );

./14-02-2017/prg1.c: for( i = 1; i < 10; ++i ) printf( "%d\n", rand() ); ./14-02-2017/prg2.c: for(i=0; i<10; i++) printf("%d\n", rand\_r( &seed )%100);

Каждая из утилит имеет большое число параметров и ключей. Если возникнет необходимость использования утилит с расширенными по сравнению с приведёнными примерами возможностями, рекомендуется прежде всего смотреть примеры (Examples) в конце справки по утилите.

• gcc запускает компилятор. Для программ лабораторных работ на языке Си достаточно использовать следующий формат вызова компилятора:

gcc program.c –o program.out,

где program.c – файл с исходным текстом, program.out – скомпилированный исполняемый файл, “-o” (строчная буква “o”) – ключ, предшествующий имени результирующего файла. В тех случаях, когда требуется подключить дополнительные библиотеки функций, сокращённые имена библиотек записываются в строке вызова компилятора с ключом “–l” (строчная буква “эл”). Пример. Скомпилировать программу p1.c, включающую функции из графической библиотеки среды Photon (сокращённо ph), и математической библиотеки

(сокращённо m):

gcc p1.c –o p1.out –l ph –l m

Компилятор gcc не требует абсолютно строгого соблюдения “необязательных” правил синтаксиса программ на языке Си. Например, можно записывать функцию main(), опустив тип возвращаемого ей значения и аргументы, т.е. вместо int main(int argc, char \*argv[], char \*argp[]) писать просто main(). Вместе с тем компилятор очень не любит, и практически всегда отказывается принимать строки программ, в которых объявления переменных перемешаны с исполняемыми операторами. Рекомендуется все переменные объявить в начале функций, и только после этого записывать строки исполняемого кода.

Для запуска скомпилированной программы необходимо обязательно указывать путь к исполняемому файлу, даже если рабочий каталог в терминале - тот, в котором программа расположена. Пример см. на рис.4.

## Работа с процессами

Операционная система QNX Neutrino относится к категории ОС, рассчитанных на платформы с аппаратной поддержкой виртуальной памяти и мультипроцессорностью. Наличие в таких платформах устройства управления памятью (MMU) обеспечивает возможность организации многопроцессности с изолированными адресными пространствами каждого из процессов. Такой подход соответствует требованиям стандарта POSIX. ОС QNX Neutrino также в прозрачном для пользователя виде поддерживает мультипроцессорные/многоядерные архитектуры, и может быть сконфигурирована для выполнения как на одном процессоре/ядре, так и на всём множестве предоставляемых аппаратурой процессорных компонентов. Гибкая архитектура QNX Neutrino даёт возможность программисту выбрать один из трёх вариантов:

− Асимметричная мультипроцессорность (AMP) – на каждом процессоре выполняется отдельный экземпляр операционной системы QNX или другой операционной системы.

− Cимметричная мультипроцессорность (SMP) – на каждом процессоре одновременно выполняется единый экземпляр операционной системы QNX, а приложения могут мигрировать на каждый из процессоров.

− Ограниченная мультипроцессорность (BMP) – единый экземпляр операционной системы одновременно использует все процессоры, но программист может привязать отдельные приложения или потоки к определённым процессорам.

В настоящей лабораторной работе предполагается, что операционная система сконфигурирована и работает в варианте SMP прозрачно для пользователя – программисту нет надобности задумываться о том, сколько ядер на платформе, на каких ядрах выполняется код операционной системы, на каких – пользовательские программы. Все эти действия операционная система берёт на себя, причём используемые в лабораторной работе библиотечные функции создания и управления выполнением задач не требуют каких-либо указаний от программиста о типе используемой операционной системой платформенной многозадачности.

Самостоятельно выполняемой задачей в ОС QNX Neutrino является программный поток (нить). Поток – это наименьшая единица кода, которая планируется и выполняется микроядром. Процесс создаёт необходимый контекст (главным образом, изолированное от других процессов адресное пространство) для выполняемых в нём потоков и может рассматриваться как контейнер, содержащий потоки. Отдельной программе в операционной системе может соответствовать один или несколько процессов. Каждый процесс обязательно содержит один поток, выполняющий код функции main(), и может содержать произвольное количество дочерних (порождённых кодом main()) потоков, а также их потомков. На выполнение всегда запускается поток. Если говорят, что запущен процесс, это значит, что начал выполняться код функции main()этого процесса.

Каждый процесс имеет свой уникальный в пределах всей системы идентификатор – целое положительное число. Каждый поток в пределах процесса имеет свой числовой идентификатор (номер). Идентификатор 1 всегда получает поток, выполняющий функцию main(), для потомков используются номера 2, 3, и т.д. В тех случаях, когда это удобно, потоку может присваиваться собственное символьное имя.

QNX Neutrino поддерживает несколько дисциплин вытесняющего планирования потоков на основе приоритетов. При принятии решения о том, какой поток ставить на выполнение планировщик микроядра учитывает всю совокупность потоков в системе независимо от того, в каких процессах эти потоки содержатся. По умолчанию потоки пользовательских программ и многие системные потоки занимают процессоры в соответствии с карусельной дисциплиной планирования (Round-Robin), и уровнем приоритета 10 из 255 доступных уровней. Всего доступно 255 уровней -от 1 до 255 для привилегированных (с ID владельца 0, т.е. root) потоков, и 63 уровня от 1 до 63 – для пользовательских. Параметры планирования каждого из потоков в системе можно посмотреть в выводе утилиты pidin. В столбце “prio” для каждого из потоков число – текущий приоритет потока, следующая за числом буква – дисциплина планирования, заданная для этого потока (“r” - Round-Robin, “o” – то же самое, что “r”, “f” – дисциплина FIFO). В настоящей лабораторной работе все потоки пользовательских программ имеют параметры планирования по умолчанию.

“Игры” с изменением приоритетов и политик планирования нужно проводить очень осторожно. Дело в том, что графическая среда Photon, включая все её утилиты, в том числе терминалы, сама имеет приоритет по умолчанию 10. Если вы запустите свою программу с более высоким приоритетом и/или политикой FIFO, то будете лишены возможности каким-то образом влиять на операционную систему до тех пор, пока ваша программа сама не завершит свою работу. Если вы допустили ошибку в коде и программа зациклилась либо считает неприемлемо долго, единственный способ прекратить это – аппаратная перезагрузка.

## API QNX NEUTRINO для организации многозадачности

ОС QNX Neutrino имеет развитый интерфейс прикладного программирования систем реального времени, в том числе средства создания процессов и запуска потоков “на все случаи жизни”. API включает как функции и утилиты по стандарту POSIX, так и QNXверсии этих функций, а также свой проприетарный API. Ниже описан минимальный набор средств для программной поддержки многозадачности в лабораторной работе.

* Запуск на выполнение из терминала любой скомпилированной программы автоматически создаёт процесс, так что в этом случае никаких специальных средств создания процессов не требуется. В ряде случаев бывает необходимо вручную запустить процесс с особыми требованиями. Для этих целей предусмотрена утилита on. Эта утилита расширяет стандартные возможности оболочки по созданию процессов. Утилита позволяет запускать процессы на удалённых узлах сети, в других терминалах, с отличными от умолчаний параметрами диспетчеризации, и т.д. Для задания приоритета и политики планирования используется ключ –p.

Примеры.

запустить программу prg.out с уровнем приоритета 9: on –p 9 ./prg.out

запустить программу prg.out с уровнем приоритета 15, политикой планирования FIFO, передав программе аргументы arg1 и arg2 on –p 15f ./prg.out arg1 arg2.

Политики планирования сокращённо обозначаются так:

* f— FIFO
* r— round-robin
* s— sporadic

Особые средства API становятся нужны, когда программным путём нужно создать поток или их группу, которые были бы изолированы по памяти от других потоков. В этих случаях применяются следующие утилиты и группы библиотечных функций (указаны только те, которые используются в лабораторных работах):

* system(). Функция библиотеки языка Си, позволяющая из программы выполнить командную строку, как если бы она была введена в окне терминала.
* spawn\*() семейство функций для создания новых процессов, имена соответствующих выполняемых файлов задаются как параметры. Из этого семейства в лабораторной работе используется функция spawnl(), аргументом которой является нуль-терминированная строка, включающая имя исполняемого файла и передаваемые ему при запуске аргументы. Пример. Создать дочерний процесс, который выполнит утилиту ls с параметрами "-t", "-r", "-l":

spawnl (P\_WAIT, "/bin/ls", "/bin/ls", "-t", "-r", "-l", NULL);

Здесь первый аргумент P\_WAIT задаёт “отношение параллелизма” выполнения родительского (вызвавшего функцию spawnl()) и дочернего процессов. P\_WAIT означает, что после запуска утилиты ls приостановить выполнение до тех пор, пока ls не завершится. Альтернативный вариант P\_NOWAIT предписывает родителю выполняться параллельно с дочерним процессом в рамках заданной для них дисциплины планирования. Обратите внимание на совпадение 2го и 3го аргументов функции spawnl().

Дублирование предусмотрено для таких программ, в которых имя исполняемого файла одно и то же, а режим работы программы определяется тем, как её задали при запуске. Например, GNU-утилита компрессии/декомпрессии файлов имеет общее имя исполняемого файла, но вызывается по-разному (gzip или gunzip) в зависимости от того, что конкретно нужно сделать с файлом [2].

* fork() создаёт точную копию вызывающего процесса. Дочь получает свой уникальный идентификатор процесса PID и наследует файловые дескрипторы и открытые потоки директорий родителя. Некоторые (не важные для настоящей лабораторной работы) вещи не наследуются. Чаще всего имеет смысл дублировать процесс с помощью fork(), когда дочь должна выполнять параллельно с родителем некоторые свои действия. Чтобы указать в коде дублируемого процесса действия дочернего процесса, его код отличают от родительского по возвращаемому функцией fork() значению: для кода родителя возвращается не нулевое значение, равное PID’у дочери, для дочери возвращается 0. Пример [2]:

printf ("The parent is pid %d\n", getpid ());

fflush (stdout);

if (child\_pid = fork ()) {

printf ("This is the parent, child pid is %d\n", child\_pid);

} else {

printf ("This is the child, pid is %d\n", getpid ()); }

В отличие от других функций, fork() нельзя использовать для запуска процессов, в которых создаются дочерние потоки.

Если в пределах оного процесса нужно создать дочерние потоки, используется функция pthread\_create(). Эта функция создаёт дочерний поток с заданными атрибутами. Атрибуты определяют:

состояние дочернего потока по отношению к родительскому - присоединяемый – joinable (это состояние по умолчанию) или обособленный - detached. Родитель, создающий joinableпоток, синхронизируется с ним (см. справку по ф-ции pthread\_join()), оставаясь заблокированным в ожидании завершения дочери. С потоком в состоянии detached родительский поток работает параллельно. Заставить родителя не занимать процессор во время работы дочери можно также, погрузив его в сон,

Наследование дочерним потоком параметров планирования родительского, или работа со своими приоритетом и дисциплиной диспетчеризации, Размер стека потока, и т.д.

Вызов и аргументы функции pthread\_create():

int pthread\_create( pthread\_t\* thread,

const pthread\_attr\_t\* attr, void\*(\*start\_routine)( void\* ), void\* arg );

thread – указатель на идентификатор (tid) созданного потока, целое число. Поток main всегда получает tid=1, дочерние потоки нумеруются с 2,

attr – указатель на структуру – атрибутную запись потока. Атрибутная запись инициализируется функцией pthread\_attr\_init() и заполняется с помощью

семейства функций pthread\_attr\_set\_\*(),

start\_routine – функция, выполняемая потоком при запуске,

arg - передаваемый функции start\_routine аргумент.

Простой и понятный пример вызова функции pthread\_create() приведен в соответствующей теме справочной системы ОС QNX.

Если вы пишете многопоточную программу, и более чем один поток использует некоторую библиотечную функцию, необходимо использовать потокобезопасный (thread-safe) вариант этой функции с суффиксом “\_r” в имени, например, rand() и rand\_r().

# ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. После загрузки операционной системы запустите графическую среду Photon командой ph, если она не появилась на экране.
2. Ознакомьтесь с содержимым экрана. Обратите внимание на и запустите минимально необходимые для работы инструменты, расположенные на полке справа – справочную систему, файловый менеджер, редактор текста и терминал. Учтите, что копировать и помещать текстовые фрагменты из/в окно терминала следует комбинациями клавиш Ctrl-Alt-C и Ctrl-Alt-V.
3. Используя файловый менеджер, создайте в каталоге /home/student рабочий каталог для текущей лабораторной работы, назвав его по вашей фамилии в латинской транскрипции с добавлением текущей даты, например. Ivanov\_14-02 (т.е. 14 февраля).
4. В справочной системе правильно задайте область поиска, установив соответствующие флажки слева внизу окна. Найдите справки по библиотечным функциям языка Си rand() и srand(), скопируйте в окно редактора тексты программпримеров из справки.
5. Используя фрагменты справочных примеров использования функций rand() и srand(), составьте простую программу, которая генерирует 10 случайных чисел в диапазоне от 1 до 99. Начальное значение датчика случайных чисел seed задайте как сумму двух чисел – текущего времени (фунция time()) и целого числа, сформированного из цифр вашего дня и месяца рождения (например, 15 августа даст число 1508). Исходный текст программы назовите prg1.c и сохраните его в созданном вами рабочем.
6. Скомпилируйте вашу программу, дав исполняемому файлу имя prg1.out. Убедитесь, что скомпилированная программа появилась в рабочем каталоге. В файловом менеджере посмотрите свойства программы с помощью пункта контекстного меню “Inspect”.
7. Запустите скомпилированную программу, задав её полное (с путём) имя в командной строке терминала. Дождитесь окончания работы, проанализируйте и убедитесь в том, что вывод программы соответствует её исходному тексту. Учтите, что по умолчанию используется буферизация выводимых данных – данные появляются на экране только после заполнения достаточно большого буфера устройства stdout. Чтобы отключить буферизацию, добавьте в программу строку setbuf(stdout, NULL);
8. Добавьте в конец вашей программы сон в течении 100 секунд с помощью функции sleep(100). Перекомпилируйте программу и запустите её
9. Откройте в отдельном окне ещё один терминал, запустите в нём утилиту “pidin” сразу после запуска в первом терминале вашей программы. Ознакомьтесь с выводом утилиты “pidin”, включая состояние потоков вашей и системных программ. Обратите внимание на приоритеты потоков и дисциплины их диспетчеризации. При необходимости задайте для “pidin” опции (ключи), обеспечивающие вывод только нужной вам информации. Чаще всего это – информация только о потоках, относящихся к конкретному процессу, соответствующие ключи “–p” или “-P”.
10. Проверьте, как работают средства принудительного завершения программ – комбинация клавиш “Ctrl-C”, утилиты “slay” и “kill”.
11. Ознакомьтесь с дополнительными средствами комплекта разработки QNX:

* Поиском объектов файловой системы по маске имён и типу с помощью утилиты с “find” оконной среды Photon. Найдите, в каком каталоге расположены файлы справочной системы, Эти файлы имеют расширение имени “html”.
* Используя утилиты командной строки find, xargs и grep, найдите в справочных html-файлах все темы справки, где имеются примеры (ключевое слово Examples).
* Запустите из меню Launch слева внизу экрана среды Photon построитель графических интерфейсов Photon Application Builder и ознакомьтесь с его интерфейсом;
* Найдите в справочной системе готовый пример программы с графическим интерфейсом, иллюстрирующей применение виджета meter, скопируйте его в текстовый редактор, сохраните, откомпилируйте и запустите. Проверьте, как работают элементы управления графическим интерфейсом.

1. Главный процесс запускается заданным образом (P\*) с приоритетом Prio\*. После запуска главный процесс создаёт DT\* дочерних потоков в состоянии TS\* и передаёт им в качестве аргументов их уникальные имена. Дочерний поток главного процесса с номером NT\* с помощью средства API “P\*” запускает дочерний процесс, синхронизируясь с его окончанием. Дочерний процесс вызывает способом PIDIN\* утилиту pidin с параметрами по умолчанию, которая выводит информацию о всех созданных потоках и запущенных процессах. Каждый вновь появляющийся процесс или поток выводит свой PID (ф-ция getpid()), TID (pthread\_self()) и переданное ему родителем имя (если таковое существует), а также время своего старта в формате часы:минуты:секунды (формат %T функции strftime()). После окончания своих действий каждый процесс или поток выдаёт текущее время и сообщение о завершении.

* Варианты задания по отдельным пунктам:
* Запуск процессов:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| P1 | P2 | P3 | P4 | P5 |
| spawnl() | system() | fork() | on | “голая” командная строка |

* Приоритет главного процесса

|  |  |  |
| --- | --- | --- |
| Prio1 | Prio2 | Prio3 |
| 9 | 10 | 11 |

* Число потоков-дочерей

|  |  |  |
| --- | --- | --- |
| DT1 | DT2 | DT3 |
| 1 | 2 | 3 |

* Состояния дочерних потоков:

|  |  |
| --- | --- |
| TS1 | TS2 |
| joinable | detached |

* Номер дочернего потока:

|  |  |
| --- | --- |
| NT1 | NT2 |
| первый | последний |

* Запуск утилиты pidin:

|  |  |
| --- | --- |
| PIDIN1 | PIDIN2 |
| spawnl() | system() |

* Варианты индивидуальных заданий:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| № варианта | Запуск главного  процесса | Приоритет  главного процесса | Количество  дочерних потоков | Состояние  дочерних потоков | Какой по счёту  поток запускает  дочерний процессс | Способ запуска  дочернего процесса | Способ вызова  утилиты pidin |
| 1 | P1 | Prio3 | DT3 | TS1 | NT2 | P3 | PIDIN1 |
| 2 | P2 | Prio2 | DT2 | TS2 | NT2 | P2 | PIDIN2 |
| 3 | P3 | Prio1 | DT3 | TS1 | NT1 | P1 | PIDIN2 |
| 4 | P4 | Prio2 | DT1 | TS2 | NT1 | P2 | PIDIN1 |
| 5 | P5 | Prio3 | DT2 | TS1 | NT1 | P1 | PIDIN2 |
| 6 | P4 | Prio1 | DT1 | TS2 | NT2 | P3 | PIDIN1 |
| 7 | P3 | Prio2 | DT4 | TS1 | NT1 | P2 | PIDIN1 |
| 8 | P2 | Prio3 | DT3 | TS2 | NT2 | P3 | PIDIN1 |
| 9 | P1 | Prio2 | DT2 | TS1 | NT2 | P1 | PIDIN2 |
| 10 | P3 | Prio3 | DT3 | TS2 | NT2 | P1 | PIDIN2 |
| 11 | P5 | Prio1 | DT1 | TS1 | NT2 | P2 | PIDIN1 |
| 12 | P2 | Prio1 | DT1 | TS2 | NT1 | P3 | PIDIN2 |
| 13 | P4 | Prio2 | DT2 | TS1 | NT1 | P2 | PIDIN1 |
| 14 | P1 | Prio2 | DT2 | TS2 | NT2 | P1 | PIDIN1 |
| 15 | P5 | Prio3 | DT3 | TS1 | NT1 | P3 | PIDIN1 |
| 16 | P2 | Prio3 | DT3 | TS2 | NT1 | P3 | PIDIN2 |
| 17 | P1 | Prio2 | DT2 | TS1 | NT2 | P1 | PIDIN2 |
| 18 | P4 | Prio3 | DT1 | TS2 | NT2 | P2 | PIDIN1 |
| 19 | P3 | Prio3 | DT3 | TS1 | NT1 | P2 | PIDIN2 |
| 20 | P2 | Prio1 | DT2 | TS2 | NT2 | P1 | PIDIN1 |

**Пример выполнения работы с процесами**

Родительская процесс: создаёт дочерний поток в состоянии joinable, передавая ему как аргумент имя “daughter”, присоединяется к нему и ждёт его окончания. После создания дочери родительский поток запускает с помощью fork() дочерний процесс, который с помощью system() запускает утилиту pidin, выводящую основные параметры созданных процессов и потоков.

Обратите внимание на строки 2 и 3 вывода программы. У вас, возможно, получится подобная “каша” из знаков. В рамках лабораторной работы это не будет считаться ошибкой. Как упорядочить вывод на экран нескольких параллельно выполняющихся потоков будет рассмотрено в последующих лабораторных работах

//Исполняемый файл lab2\_2.out

#include <sys/types.h>

#include <process.h>

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

void\* function( void\* arg )

{

char \*name = (char\*) arg;

printf( "This is thread tid %d name %s\n", pthread\_self(), name); sleep(19); printf("child dies\n"); return( 0 );

}

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

{

int child\_pid; pthread\_attr\_t attr; pthread\_t tid;

char name1[]="daughter";

printf ("The parent is pid %d\n", getpid ()); setbuf(stdout, NULL);

if (child\_pid = fork ())

{

printf ("THIS IS THE PARENT,CHILD PID IS %d\n", child\_pid); pthread\_create( &tid, /\*&attr\*/ NULL, &function, (void\*) name1 ); pthread\_join(tid, NULL); printf("parent dies\n");

}

else

{

printf ("this is the child, pid is %d\n", getpid ()); system("pidin -p lab2\_2.out");

}

}

//------------------------------------------------------------------------

/\*

# ./lab2\_2.out

The parent is pid 548904

THItSh iIsS iTsH Et hPeA RcEhNiTl,dC,H IpLiDd PiIsD 553006I

S 553006

This is thread tid 2 name daughter

pid tid name prio STATE Blocked 548904 1 ./lab2\_2.out 10o JOIN 2

548904 2 ./lab2\_2.out 10o NANOSLEEP 553006 1 ./lab2\_2.out 10o REPLY 1 child dies parent dies

# \*/

# Контрольные вопросы

1. Перечислите утилиты, безусловно необходимые для комплекса действий по выполнению лабораторных работ.
2. Какие ключи безусловно необходимы в командной строке вызова компилятора gcc?
3. Перечислите возможности редактора текстов среды Рhoton.
4. Какими из них вы пользовались?
5. В какой кодировке по умолчанию сохраняет тексты редактор среды Рhoton?
6. Какая библиотека подключается при компилировании программ по умолчанию?
7. В чём отличие с точки зрения организации многозадачности дочерних процессов, запущенных с помощью функций семейства spawn\*() и функций \*fork()?
8. Каким образом можно задать приоритет и дисциплину планирования для программы, запускаемой вручную из терминала?
9. Каким образом можно заставить родительский поток не занимать процессорные ресурсы во время выполнения дочерних потоков в едином или отдельном процессе?
10. С помощью каких функций поток может узнать свои собственные идентификатора процесса и потока?
11. Как можно узнать идентификатор дочернего процесса. создаваемого с помощью функции fork()?
12. Какие характеристики запускаемого процесса можно задать в параметрах утилиты on?