УДК: 004.031.4:539.1.075:539.17

ПОСТРОЕНИЕ СИСТЕМ СБОРА ДАННЫХ ДЛЯ МНОГОКАНАЛЬНЫХ ЯДЕРНО–ФИЗИЧЕСКИХ УСТАНОВОК НА ОСНОВЕ UNIX–ПОДОБНЫХ ОПЕРАЦИОННЫХ СИСТЕМ

А.Ю. Исупов[1], В.Е. Ковтун[2], А.Г. Фощан[2]

[1]*Лаборатория физики высоких энергий, Объединенный институт ядерных исследований*
Россия, 141980, г.Дубна, ул. Жолио–Кюри, 6
isupov@moonhe.jinr.ru
[2]*Физико–технический факультет, Харьковский национальный университет им. В.Н.Каразина*
Украина, 61108, г.Харьков, пр. Курчатова, 31

Поступила в редакцию 20 января 2009 г.

В работе рассмотрены некоторые идеи и принципы, используемые при построении программных средств систем сбора данных на основе современных UNIX–подобных операционных систем. В частности, это широкое использование интерфейсов межпроцессного и межмашинного взаимодействия для передачи данных, исключение промежуточного хранения данных на жестких дисках, модульная организация, позволяющая легко распределять и масштабировать систему сбора данных, реализация систем типовой инфраструктуры и т.д. Проведено сравнение UNIX–подобных операционных систем и DOS как сред, в которых реализуется программное обеспечение систем сбора данных. Описывается инфраструктура, предоставляемая системой *qdpb*, позволяющая реализовывать модульные распределенные системы сбора, транспортировки, обработки и представления данных для разнообразных ядерно–физических установок с электронным съемом данных. Изложены некоторые применявшиеся в Лаборатории высоких энергий (ЛВЭ) Объединенного института ядерных исследований (ОИЯИ, г.Дубна) схемы реализации на основе инфраструктуры *qdpb* систем сбора данных таких установок. Кратко представлен вариант дальнейшего развития инфраструктуры сбора данных – система *ngdp*.

КЛЮЧЕВЫЕ СЛОВА: компьютер, КАМАК, программное обеспечение, операционная система, сбор данных.

В настоящее время многие ядерно–физические эксперименты ставится с помощью экспериментальных установок, имеющих электронный съем данных, то есть работающих в так называемом онлайновом режиме (“в линию” с ЭВМ). Аппаратные и программные средства для обеспечения такого режима работы составляют систему сбора данных, которая, таким образом, является важной составной частью многоканальной экспериментальной установки, состоящей из большого числа детекторов. К аппаратным средствам относятся:
– оцифровывающая сигналы детекторов электроника, часто выполненная в одном из магистрально–модульных стандартов;
– компьютер, принимающий данные;
– интерфейс, сопрягающий первую со вторым.

В качестве магистрально–модульного стандарта все еще распространен КАМАК, компьютером часто является так называемый персональный, сопрягающим интерфейсом является пара контроллер крейта / адаптер шины расширения компьютера (соединенные некой линией двусторонней передачи данных). К программным средствам относятся микропрограммы (firmware) электроники и программное обеспечение (ПО, software) компьютера. Последнее в данной работе мы и будем понимать (в узком смысле) под системой сбора данных (Data AcQuisition, DAQ). Основой компьютерного ПО является операционная система (Operating System, OS), адекватный выбор которой может существенно упростить разработку, реализацию, сопровождение и использование остального ПО системы DAQ, в то время как неадекватный – существенно усложнить таковые или сделать их невозможными. Аргументация выбора в пользу UNIX–подобной OS проводится в данной работе. Далее в работе излагаются основные идеи и принципы, инспирированные UNIX и положенные в основу системы *qdpb* (Data Processing with Branchpoints, (система) обработки данных с точками ветвления). В частности, это модульный принцип построения, позволяющий одновременно исполнять многие элементы системы, в том числе, на различных процессорах и компьютерах. Другой особенностью является распространение информации, оформленной в некие пакеты, между различными программными модулями посредством потоков, а не файлов, что позволяет, в частности, избежать промежуточного хранения данных на жестком диске в ходе их прохождения через элементы системы. Система *qdpb* предоставляет инфраструктуру (framework) для построения модульных распределенных систем сбора, транспортировки, обработки и представления данных конкретных установок, являя собой своего рода конструктор с набором типовых деталей. Дальнейшим развитием инфраструктурных систем DAQ является *ngdp* (NetGraph Data Processing, (система) обработки данных на основе программного пакета *netgraph*), находящаяся в стадии разработки, которая позволит минимизировать копирование данных в оперативной памяти и избежать принудительного планирования критичных частей кода. Текущее состояние впервые кратко описано в данной работе. Далее в работе рассмотрены де-

тали реализации системы *qdpb* и нескольких успешно эксплуатируемых DAQ систем для ядерно–физических установок в Лаборатории высоких энергий ОИЯИ на ее основе: СФЕРА с числом каналов считывания до $\sim$ 200, СКАН, многочисленные варианты поляриметров. Целью данной работы является изложение взглядов авторов на принципы построения современных систем сбора данных для многоканальных ядерно–физических установок, сформированных на основе опыта такого построения в ЛВЭ ОИЯИ.

Оговоримся также, что в тексте данной работы имена файлов и пакетов ПО выделяются *курсивом*, конструкции языка C и воспроизводимые "как есть" литералы – `шрифтом пишущей машинки`, требующие подстановки актуального значения имена заключаются в угловые скобки: `<event_type>`. Ссылки на online–документацию в так называемой системе manual pages набраны следующим образом: ***cat(1)***. Эта общепринятая форма записи имени некой программы или другой сущности OS означает, что документация для данной сущности находится в разделе 1 ("Программы и команды командного интерфейса пользователя") под именем `cat` , то есть просмотреть эту документацию можно командой "`man 1 cat`" .

DOS И UNIX КАК СРЕДЫ РЕАЛИЗАЦИИ ПРОГРАММНЫХ СРЕДСТВ DAQ

Кратко проследим эволюцию аппаратных средств (hardware) так называемых персональных компьютеров (Personal Computer, PC), OS для них, а также взглядов их пользователя. По–видимому, первой массовой архитектурой (то есть реализацией некого набора аппаратных средств) PC была IBM PC на 16–разрядном процессоре i8086, а также последовавшие ее усовершенствования на процессорах до i80386 (или просто i386). Первой массовой OS под эту (собственно PC) архитектуру была дисковая операционная система (Disk Operating System, DOS). DOS по сути превращала компьютер в большой программируемый калькулятор с некими графическими средствами, накопителем на гибких (Floppy) и жестких (Hard) дисках (Disk Drive) – FDD и HDD, аппаратными средствами ввода (клавиатура) и вывода (монитор, принтер). "Калькулятор" потому, что, грубо говоря, поддерживалось одновременное выполнение лишь одной программы, которая поэтому вынужденно обменивалась данными только с файлами, расположенными на FDD и HDD, а также с файлами портов вывода COM1, COM2, LPT1, ... Конечно, мы несколько упрощаем, т.к. процессор i8086 сложнее (как правило, 8–разрядного) калькуляторного и уже реализует аппаратные прерывания (Interrupt ReQuest, IRQ), которые позволяют переключаться с исполнения единственной пользовательской программы на исполнение других частей кода – например, драйверов аппаратуры или резидентных программ, и затем возвращаться обратно.

Можно считать, что использовавшиеся в PC процессоры до i386 и не позволяли бóльшего, хотя существовали OS и более сложные, чем DOS, например, Xenix. Тем не менее уже тогда появились интерфейсы к магистрально–модульным стандартам ядерной электроники (например, самый известный и доживший до наших дней – КАМАК [1], VME и т.д.) с адаптерами ("платами расширения") для шин расширения (expansion bus) PC – сначала 8–разрядной XT, затем 16–разрядной ISA (ранее называвшейся AT), – позволявшие автоматизировать сбор оцифрованных данных с физических детекторов. Такой адаптер доступен для центрального процессора по определенным адресам в пространстве памяти и/или ввода/вывода, что позволяет обмениваться данными между адресами адаптера, основной памяти и регистрами процессора, то есть осуществить так называемый режим "в линию" (online) работы внешней электроники физических детекторов с компьютером.

Принципиальным шагом в развитии PC стал процессор i386 (кстати, первый 32–разрядный для PC), впервые включивший в себя средства, необходимые (как показала практика, и достаточные) для поддержки UNIX–подобных OS. Под практикой имеется в виду перенос ("портирование", porting) уже имевшихся для больших компьютеров (mainframe) UNIX–подобных OS (например, 386BSD Вильяма Джолитца (William F. Jolitz) на основе 4.3BSD и SCO UnixWare компании Santa Cruz Operation, Inc. на основе SysV) или написание новых (например, Linux Линуса Торвальдса (Linus Torvalds) просто на основе стандарта POSIX [2]). Начиная с этого момента DOS следует считать архаизмом, ибо средства UNIX–подобных OS существенно богаче (не претендуя на сколько–нибудь исчерпывающую библиографию, см., например, [3–5]) и на тот момент были примерно на 10 лет проработаннее, чем DOS, поскольку первые версии UNIX появились в конце 1969 г., первые версии DOS – в начале 1980–х. Более того, следует заметить, что сам DOS – это усеченная до способностей архитектуры первых PC UNIX–подобная OS. Итак, возможно стало исполнять одновременно более одной программы (процесса) – так называемая многозадачность (multitasking), и как следствием – многопользовательский режим, то есть одновременная поддержка более одного сеанса пользователя. Заметим, что некоторые надстройки над DOS, например, QuarterDeck Desqview или MS Windows 3.1, были скорее переключателями задач (task switcher), чем реализацией собственно multitasking'а, ибо иллюзии одновременного исполнения не было – переключением занимался сам пользователь, а не OS. Вследствие multitasking'а стал возможен обмен данными непосредственно между различными процессами, то есть в памяти, минуя существенно более медленные диски. Кроме того, поддержка постоянно (а не от случая к случаю, как принтеры и даже диски под DOS'ом) действующей периферии перестала препятствовать исполнению сеанса пользователя. Но, заметим, не перестала конкурировать, ибо многопроцессорность на архитектуре `i386` возникла позднее. OS UNIX – это арбитр такой конкуренции за неделимые аппаратные средства. Важнейшим представителем постоянно действующей периферии являются адаптеры компьютерных сетей (далее – просто сетей), в частности, Ethernet

(самая массовая аппаратная реализация сети). Вот почему поддержка их в UNIX естественна, а в DOS – медленна и неуклюжа, ибо требует целого букета резидентных программ.

Таким образом, появилась возможность обмена данными между процессами, исполняемыми на разных компьютерах – для увеличения совокупной вычислительной мощности или для передачи информации на расстояние. Что же касается интерфейса пользователя, то принципиальных различий между единственным сеансом DOS и одним из многих сеансов любой UNIX–подобной OS практически нет – кроме фундаментальной возможности запустить, помимо одного связанного с терминалом процесса (foreground process), одновременно многих фóновых (background) процессов. Различия командного языка оболочки (командного интерпретатора, shell'a) не принципиальны и даже не столь существенны.

Архитектура `i386` (другое название – `IA-32`) достаточно долго развивалась, после набора инструкций процессора i386 существовали как минимум: i486 (сюда же AMD K5), i586 (Pentium, Pentium II, AMD K6), i686 (Pentium Pro, Pentium III, Pentium 4, AMD K7). В настоящее время архитектура `i386` постепенно вытесняется 64–разрядной архитектурой `amd64` (другие названия – `EM64T`, `x86-64`, `IA-32e`), к которой относятся как AMD K8 Athlon 64, так и Intel Pentium Dual–Core. Все они пока тем не менее полностью поддерживают набор инструкций, достигнутый на архитектуре `i386`. "Родная" 64–разрядная архитектура Intel, `IA-64`, представленная Intel Ithanium, в настоящий момент не относится к массовому сегменту рынка.

Конечно, дополнительные возможности требуют также дополнительных расходов. Но это не накладные расходы на исполнение, безусловно, более сложного кода самого ядра OS, как можно предположить в первую очередь. Практика показывает, что такие расходы хотя и выше для UNIX, чем для DOS, но прогрессируют медленнее производительности процессоров и, начиная с i486, пренебрежимы. Основное увеличение расходов приходится не на "время исполнения" (runtime) кода, а на его создание. Имеется ли в виду усложнение программирования? И да, и нет. Программирование проходит несколько стадий – в частности, проектирование (или дизайн, design, то есть разработка архитектуры, алгоритмов, интерфейсов), а также собственно кодирование на каком–либо языке. Под DOS (вынужденно!) писалась "монолитная" программа, которая, разрастаясь, усложняла кодирование, т.к. упиралась в ограничения адресного пространства DOS, требовала ухищрений при работе с IRQ, и т.д. Под UNIX же кодирование упрощается благодаря богатым возможностям OS, "делающей работу за нас". Под DOS единая программа "знала" все обо всем DAQ и поэтому не требовала серьезного дизайна. Под UNIX (тоже вынужденно!) DAQ из программы становится системой, то есть усложняется концепция DAQ и основные расходы смещаются в область дизайна, поскольку нужно писать комплекс программ. Поэтому в максимально простых случаях типа однокрейтного стенда, работающего по опросу с одним PC, может даже показаться, что UNIX "не окупается". Но необходимо учитывать также следующее: 1) DOS давно не сопровождается и как таковая не поддерживает современного hardware, а старый непрерывно выходит из строя; 2) при малейшей попытке усложнения – например, пространственно распределить систему DAQ (для передачи данных и/или для наращивания вычислительной мощности), позволить удаленное управление ею и/или просмотр результатов, даже просто перейти от опроса к работе по IRQ – мы упираемся в ограничения DOS.

Привлекательность UNIX–подобных OS заключается еще и в наличии широкого выбора свободно распространяемых по GNU Public или BSD лицензиям версий, что существенно, когда необходимо уделять внимание соблюдению авторских и т.п. прав, да и просто экономит ресурсы небогатых исследовательских групп. (Возможно, лицензионная версия DOS стоила недорого, но в мире MS Windows оставаться в правовом поле становится все дороже.) Более того, зачастую свободно распространяются также исходные программные тексты (source), что немаловажно при разработке ПО вообще и практически необходимо при разработке элементов ядра OS, поскольку может понадобиться полная перекомпиляция последнего. Свободная распространяемость в виде sources способствует тому, что хорошо спроектированный код будет долго эксплуатироваться и применяться вновь и вновь, раз уж расходы на его дизайн хоть и неизбежны, но уже понесены. С другой стороны, свободные sources не препятствуют даже коммерциализации финальных продуктов с их использованием при соблюдении упомянутых лицензий.

Устройство UNIX таково, что для обеспечения бесперебойности работы OS в целом существуют привилегированный код, выполняющийся в (аппаратно обеспечиваемом) привилегированном режиме (контекст ядра, kernel context), и непривилегированный, то есть всякий другой (исполняется в непривилегированном режиме – пользовательский контекст, контекст задачи, user context). Работать непосредственно с аппаратными средствами может только первый, тогда как второй – лишь через интерфейс первого, то есть "просить" первый "сделать то–то". Все процессы пользовательского сеанса, как видно уже из названий, выполняются в непривилегированном контексте. Значит, для работы с адаптером KAMAK уже потребуется отдельная часть кода, так или иначе встроенная в ядро OS, которое в простейшем случае есть монолитный файл, размещаемый в памяти и исполняемый при старте OS во время загрузки (boot sequence) компьютера. Встраивание в такое ядро означает совместное линкование (linkage, оно же – редактирование связей, link editing, или компоновка) нашего кода с остальным ядром, а затем перезагрузка OS уже с новым ядром. Это небыстрая процедура и

вызывала бы известные неудобства для разработчика, но современные ядра имеют модульный характер, то есть так или иначе позволяют уже в ходе обычной работы OS загружать и выгружать специальным образом скомпилированные и слинкованные программы – модули ядра (по одной из терминологий – KLD–модули, от Kernel Link eDiting).

Кроме того, адресные пространства отдельных процессов для тех же целей бесперебойности и безопасности изолированы друг от друга и могут взаимодействовать (синхронизироваться, обмениваться данными, и т.п.) только через определенные интерфейсы межзадачного взаимодействия (InterProcess Communication, IPC). Термин IPC может употребляться также в узком смысле, означая лишь три механизма, пришедшие из SysV – разделяемую память (shared memory), очереди сообщений (message queue) и семафоры (semaphore). Одним их самых употребительных интерфейсов межзадачного взаимодействия является поток (stream, не путать со STREAMS – стиль взаимодействия (и написания) KLD, в основном реализующих сетевые протоколы). Так, каждому процессу предоставляется поток стандартного ввода (stdin) и два потока вывода – стандартный (stdout) и ошибок (stderr). Процесс может дополнительно открыть другие потоки (и даже переназначить на них стандартные), закрыть стандартные, либо просто воспользоваться теми, что предоставили ему при запуске "внешние силы", то есть shell. В частности, stdout одного процесса может быть направлен в stdin другого процесса, образуя так называемый конвейер (unnamed pipe) передачи данных для их последовательной обработки внутри процессов. Возможно, процессов в конвейере будет и более двух. Такой подход существенно облегчает программирование процесса (хотя может усложнить командную строку для его запуска). В частности, можно породить поток из некого (обычного, regular) файла(ов) – этим занимается такое стандартное средство OS, как ***cat(1)*** – и/или "слить" такой поток в файл (это делается посредством символов перенаправлений shell'а). Кроме того, можно оставить работу по поиску имен файлов shell'у, воспользовавшись его символами подстановки (wildcards) "**?**" и "*****".

Естественно, что когда процессов становится более одного, подход к их написанию меняется – нужно учитывать необходимость их синхронизации друг с другом и с доступом к неделимым ресурсам (в том числе – центральному процессору, Central Processor Unit, CPU). Фундаментальным понятием становится "сон" процесса – состояние, в котором следует ожидать ресурсов либо готовности партнерского процесса. Т.к. в любой конкретный момент скорее всего существует еще кто–то, готовый к исполнению, процессу следует предпринимать специальные действия, предоставляемые OS UNIX (например, вызов функции семейства ***sleep(3)*** или системного вызова ***select(2)***), в местах ожидания, а не исполнять, например, долгий пустой цикл. Ко сну ведут также многие функции и системные вызовы, связанные с предоставлением ресурсов, например, ***write(2)*** и ***read(2)***.

ИДЕОЛОГИЯ СОВРЕМЕННОЙ СИСТЕМЫ DAQ

Далее нас интересует более плодотворная для построения DAQ идея – исключить промежуточные файлы на HDD. В идеале мы должны записывать лишь более или менее конечные результаты и в конечное местоположение – не забываем, что дисковые операции записи/чтения существенно медленнее операций записи/чтения памяти! Это можно сделать, используя те или иные формы потоков в UNIX, не размещающие данные (хотя бы временно) на HDD – помимо упомянутого (unnamed) pipe это именованный конвейер (named pipe, он же "First Input – First Output", FIFO) и сокет (socket). Потоки естественным образом позволяют синхронизировать партнерские процессы, т.к. функции доступа к потокам погружают процесс в сон, если в данный момент данные отсутствуют, и пробуждают его сразу же по появлении таковых. Обычный файл имеет по сравнению с потоками лишь одно преимущество – произвольный доступ (direct access) вместо последовательного (sequential). Обратной его стороной является более важный для нас недостаток – механизм синхронизации процесса с поступлением данных ***select(2)*** бесполезен с обычными дописываемыми в конец файлами – он всегда мгновенно выдает успешный код завершения. Преимущество прямого доступа не столь существенно для DAQ, в любом случае порождающей данные порциями – событиями (event), имеющими в каждом случае известную длину. Будем считать событием информацию, логически сгруппированную по каким–либо причинам, например, полученную в результате обработки одного прерывания от КАМАК. События мы будем вычитывать последовательно (event by event) из потока в память процесса. Для удобства обращения с каждым событием предпошлем ему заголовок (header) фиксированной длины и назовем такую сборку пакетом (packet, см. также ***packet(5)***). Заголовок пусть содержит хотя бы полную длину пакета, дату и время создания его с точностью до микросекунд, тип события, порядковый номер, контрольную сумму (Cyclic Redundance Code, CRC), а также идентификационную строку, используемую для контроля правильности разбивки потока на пакеты, осуществляемой по длинам пакетов. Величина типа характеризует смысл содержащейся в пакете информации и позволяет построить классификацию пакетов, выделив, например, пакеты данных, управляющие пакеты, ответы на управляющие пакеты и т.д. Дата, время и CRC опциональны, то есть могут быть заполнены нулями, их актуальность указывается полем флагов заголовка пакета.

Итак, потоки, природу которых мы пока не конкретизируем, будут пересылать такие пакеты последовательно между сущностями OS – не говорим "процессами", потому что как минимум один раз данным следует

преодолеть также границу между ядром и пользовательским процессом, а разнообразным командам, в конечном счете исходящим от пользователя – в обратном направлении. Для этого также существует набор более или менее традиционных средств – как минимум интерфейс драйвера в узком смысле и системный вызов. Мы приходим к выводу, что необходимо спроектировать и реализовать инфраструктуру системы DAQ, которая бы организовывала нам необходимые потоки пакетов. Она может быть универсальной, т.к. не работает с собственно событиями, содержащимися в пакетах, а только с их заголовками. Такая система была реализована под названием *qdpb* [6, 7].

Теперь вспомним, что процессор в нашем компьютере, скорее всего, один. Поэтому многозадачность достигается посредством принудительного планирования (preemptive scheduling) предоставления CPU коду того или иного процесса OS для его исполнения. Конечно, существуют сложные алгоритмы планирования (scheduling algorithm), так что пользователь в ходе интерактивного сеанса не замечает задержки отклика, существуют системы приоритетов, включая так называемые приоритеты реального времени (RealTime Priorities, RTP), но... Основной вывод таков – в обычной UNIX–подобной OS общего назначения (например, Linux или FreeBSD) достичь гарантированного времени отклика на некое внешнее событие на уровне пользовательского процесса нельзя. Конечно, существуют коммерческие realtime OS под специализированный hardware, и некоторые даже имеют командный и программный интерфейсы стандарта POSIX, например LynxOS, но мы здесь говорим о свободно распространяемых, в том числе в виде sources, UNIX–подобных OS под массовые архитектуры. Поэтому существуют и еще как минимум две причины необходимости размещения части кода системы DAQ в ядре OS, кроме уже упоминавшегося доступа к аппаратным средствам вообще – это работа с IRQ и желание избежать принудительного планирования для некоторых критичных частей кода. IRQ по сути – штатный механизм для реагирования на внешние события, и время отклика на него, то есть время между выставлением сигнала на шине и передачей управления коду обработчика прерывания, на современном hardware массовой архитектуры `i386` вполне приемлемое [8]. Кроме того, работа с аппаратными средствами по IRQ (в отличие от работы по опросу) – практически единственный способ занимать CPU лишь при реальной необходимости этого, поэтому очень важно, чтобы адаптеры КАМАК и т.п. аппаратуры также были способны генерировать IRQ.

С другой стороны, существуют и причины необходимости размещения части кода системы DAQ вне ядра OS – интерфейс с пользователем неизбежно будет процессом, равно как и визуализирующая часть, а может быть, и некая преобразующая данные часть, если она использует в той или иной степени пакет ROOT [9–11] – а значит, его разделяемые библиотеки (shared library) и так называемые словари (dictionary). Поскольку для DAQ привлекательно было бы исключить принудительное планирование для возможно бóльшей части ее кода, мы можем постараться как можно дольше не выпускать данные из контекста ядра. Эта идея отсутствовала в *qdpb* и появилась в *ngdp*.

Итак, мы сделаем с данными прямо в контексте ядра все, что можно, и тогда нам могут понадобиться средства для организации некого потока данных между различными KLD–модулями ядра. Такие средства уже существуют, в частности уже упоминавшиеся STREAMS, а также пакет ***netgraph(4)***. Первые, судя по их описанию, не столь гибки и удобны для нас, к тому же отсутствуют во FreeBSD, которая по различным, в том числе историческим, причинам уже используется для реализации системы *qdpb*. Последний разработан для схожих с нашими целей – для пересылки сетевых пакетов между трансформирующими их узлами (нодами, node) при реализации сложных многоуровневых сетевых протоколов. Дело в том, что в сетевых протоколах данные тоже пересылаются в виде пакетов, то есть собственно данных с многочисленными заголовками от протоколов разных уровней. Заметим, что не следует путать сетевые пакеты с пакетами системы *qdpb*, у которых та же идея, похожий смысл, но другая реализация.

Таким образом, мы приходим от *qdpb*, когда в ядре OS располагалось лишь то, что не может быть реализовано иначе, к *ngdp*, когда вне ядра OS располагается лишь то, что не может быть реализовано в ядре. До недавнего времени таковым был поток исполнения (execution stream) – не следует путать его с потоками данных. Простейшим примером потока исполнения (или нескольких, если используются нити пользовательского контекста, ***pthread(3)***) является процесс. Код же традиционного ядра UNIX получал управление либо как обработчик прерывания, либо во исполнение системного вызова от процесса – так называемое переключение (исполнения) процесса в режим (контекст) ядра. По окончании системного вызова происходит обратное переключение в режим (контекст) задачи. Поэтому для некоторых целей приходилось создавать фиктивные процессы, сразу “уходившие в ядро” и уже не возвращавшиеся оттуда. В настоящее время существуют средства организации потоков исполнения внутри ядра – так называемые нити ядра, kernel thread, с появлением которых ситуация меняется, потому что они по сути позволяют делать многое из того, что мог ранее делать только процесс, но не подвергаются принудительному планированию. Понятно, что освобождать CPU, то есть возвращать управление и “впадать в сон” такие нити должны добровольно (для чего существует предоставляемый UNIX специальный интерфейс), в противном случае нормальное функционирование OS будет нарушено, возможно, необратимо. Так что вне ядра остается совсем немного – в основном, как уже сказано, то, что нуждается в

графическом интерфейсе пользователя (Graphical User Interface, GUI).

Средства реализации GUI внутри все той же единственной DOS программы появились вместе с разделением видеомод на текстовые и графические, то есть очень давно. Затем MS Windows, бывшая сперва надстройкой над однозадачной DOS, а затем поглотившая ее в качестве одной из возможных задач, предоставила полномасштабную оконную систему. Удивительно, но произошло это заметно позже появления разработанной для UNIX–подобных операционных систем так называемой X Window System (см., например, [12–14]), причем выбор разнообразных оконных менеджеров (window manager) в последней всегда был, в отличие от всегда единственного (возможно, неотъемлемого) в первой. С точки зрения систем DAQ потребность (и даже зачастую необходимость) в графических средствах возникает на достаточно позднем этапе визуализации. Таким образом, единственным преимуществом MS Windows перед UNIX–подобными OS – наличием многочисленных современных мультимедийных приложений – для наших целей можно пренебречь. Букет же недостатков – отсутствие доступных программных текстов ядра и драйверов, а равно открытой документации на них, без которых программировать работу с аппаратурой весьма затруднительно; необходимость покупки лицензий для легальной работы с собственно OS (например, для опубликования результатов такой работы); многолетний коммерческий стиль разработки при отсутствии предшествующего академического этапа, приведший к многочисленным узким местам: файловая система (File System, FS), дисковый кэш (cache), система виртуальной памяти (Virtual Memory, VM) и подкачки (swapping), алгоритмы (принудительного ли?) планирования, реализация некоторых сетевых служб и приложений (а возможно, даже их отсутствие); неприемлемые общая нестабильность, ресурсоемкость и малопредсказуемая производительность (результаты неаккуратной реализации на объектно–ориентированных языках программирования, Object–Oriented Programming, OOP) – весьма велик. Заметим, что до сих пор исполнимый код, генерируемый C++ компиляторами, объемистее и медленнее эквивалентного кода, генерируемого C компиляторами. Кроме того, вполне современные ядра UNIX написаны на C, а X Window System и ***netgraph(4)*** являются великолепными образцами реализации объектно–ориентированного программного кода чисто средствами C. Хотя, как известно, MS Windows никогда и не претендовала на обслуживание, например, систем жизнеобеспечения и т.п. (так называемых систем High Risk Activity, HRA), сопряженное как раз с достаточной надежностью, с одной стороны, и с адекватным обслуживанием внешней аппаратуры, с другой стороны. На худой конец, именно и только этап визуализации в распределенной системе DAQ легко предоставить MS Windows, хотя и в этом никакой существенной необходимости не наблюдается.

СРАВНИТЕЛЬНЫЙ АНАЛИЗ СИСТЕМ QDPB И NGDP

Итак, система *qdpb* состоит из подсистем:
– обслуживания аппаратных средств (например, подсистемы KAMAK) и
– обработки и транспортировки данных.

В качестве подсистемы KAMAK используется так называемый пакет *camac* [15, 16], написанный В. Ольшевским и К. Грицаем (тогда оба из ЛЯП ОИЯИ) под OS FreeBSD. К сожалению, публикаций, отражающих современное состояние пакета, не существует, хотя совсем недавно пакет исправлен под FreeBSD 7.0-RELEASE. Пакет *camac* включает в себя (разумеется, вместе с sources):
– общий интерфейс ***camac(4)*** для ядра OS (системный вызов);
– библиотеки для обоих контекстов – ***camac(9)*** (контекст ядра) и ***esone(3)*** (соответствует стандарту ESONE [17]), ***muspin(3)*** (несколько более удобный для использования на C вариант);
– драйверы KAMAK (в широком смысле, то есть по сути код для работы с конкретным аппаратным адаптером, но, как правило, не реализующие собственно интерфейс драйвера UNIX) специфических пар адаптер/контроллер крейта KAMAK. На текущий момент это:
  * ***kk(4)*** – для KK009/KK012 [18, 19];
  * ***ki(4)*** – для KK011 [20];
  * ***kl(4)*** – для KKL01;

  а также добавленные А.Ю.Исуповым:
  * ***kb(4)*** – для CCPC{4,5,6} [21, 22];
  * ***kh(4)*** – для CC02 разработки ФТФ ХНУ (ISA и PCI варианты) [23];
– шаблон *cm_cdd.c* модуля KAMAK (см. также ***camacmod(9)***). Вообще модуль KAMAK с именем `xxx` реализуется в виде KLD–модуля ядра, после загрузки посредством ***kldload(8)*** требуется его конфигурация утилитой `xxx`***conf(8)***, которая вызывает конфигурирующую функцию `xxx_conf()` (и, возможно, функцию проверки текущей конфигурации `xxx_test()`), после чего обработчик прерываний `xxx_hand()` присоединяется к драйверу KAMAK. Модуль KAMAK может реализовывать также интерфейсную функцию `xxx_oper()`, посредством вызова которой с различными аргументами утилита `xxx`***oper(8)*** управляет модулем;
– несколько утилит в виде пользовательских программ: ***c_master(1)***, ***crate(1)***, ***naf(1)***, ***c_control(8)***, ***cddconf(8)***;
– набор примеров и тестов;
– документацию в виде manual pages.

Общий интерфейс ***camac (4)*** предоставляет унифицированный интерфейс к аппаратуре КАМАК как для процессов, так и для элементов ядра, скрывая особенности конкретных пар адаптер/контроллер, что позволяет писать (почти) аппаратно–независимые программные тексты. Впрочем, при необходимости (например, для снижения накладных расходов во время исполнения) в контексте ядра можно воспользоваться и аппаратно–зависимыми наборами макросов, реализующих доступ к адаптеру определенной модели (предоставляются в виде заголовочных файлов, в настоящий момент – *ci_kk.h*, *ci_kb.h*, *ci_kh.h*). Наличие общего интерфейса ***camac (4)*** позволяет упростить и формализовать написание драйверов КАМАК для обслуживания новых пар адаптер/контроллер, а также модулей КАМАК, реализующих обработчики прерываний. Последние, строго говоря, приходится перекомпилировать при каждом изменении состава обслуживаемых блоков КАМАК, однако разработана достаточно формальная схема описания такого состава (так называемые конфигурируемое представление аппаратуры КАМАК [7]), существенно упрощающая перенастройку обработчиков прерываний. Дело в том, что конфигурируемый во время исполнения по составу обслуживаемых блоков КАМАК обработчик прерываний будет, помимо сложности и небезопасности для OS, непозволительно медленным из–за переноса вычислений, которые могли бы быть выполнены уже компилятором, на время исполнения, причем не однократно, а в каждый цикл КАМАК.

Подсистема обработки и транспортировки данных включает в себя соединенные потоками пакетов рабочие модули, служебные модули и точки ветвления, а также не работающие с потоками управляющие модули.

Рабочий модуль есть процесс для преобразования информации событий, которые он получает и/или отдает в виде потоков пакетов, причем выходной поток может быть неидентичен входному. Рабочие модули могут быть источниками пакетов (***genpack (1)***), фильтрами пакетов (***filter (1)***), потребителями пакетов (***writer (1)***, ***statman (1)***, ***analyser (1)***, ***b2r (1)***).

Служебный модуль организует потоки пакетов, читая или записывая пакеты в поток, но не преобразует пакеты и порядок их следования. Примеры служебных модулей: интерфейс к точке ветвления – ***bpput (1)***, ***bpget (1)***, интерфейс для пересылки потока через пару сокетов TCP/IP (из так называемого пакета ***netpipes (1)***) – ***faucet (1)***, ***hose (1)***.

Точка ветвления ***branchpoint (4)*** была оформлена как KLD–модуль ядра и предоставляла системный вызов в качестве интерфейса для работы с собственным буфером из контекста задачи и функцию, доступную для обращений из контекста ядра. Именно, в наиболее поздних версиях FreeBSD Kernel Link eDitor разрешает (resolve) соответствующий функции символ во время загрузки KLD–модуля обработчика прерываний, если KLD–модуль точки ветвления уже загружен. Для этого требуются соответствующие декларации в текстах обоих KLD–модулей. (В более ранних версиях OS доступ к этой функции реализовывался по–другому.) Точка ветвления поддерживала регистрацию (возможно нескольких) входных потоков (в т.ч. одного – из контекста ядра, от обработчика прерываний КАМАК) и (возможно нескольких) выходных потоков. Таким образом, она объединяла в порядке времени поступления несколько различных входных потоков в идентичные выходные.

Управляющие модули предназначены для управления другими элементами системы *qdpb*, либо для иных целей (например, визуализации), не требующих работы с потоками пакетов. В частности, это: ***sv (1)*** – SuperVisor, ***histview (1)*** – представление гистограмм, ***cntview (1)*** – представление текстовых данных, ***alarm (1)*** – представление системного журнала (syslog'a), ***watcher (1)*** – слежение за состоянием обработчика прерываний.

Следует понимать, что некоторые элементы системы (например, служебные модули, точки ветвления, некоторые управляющие модули) могут быть реализованы независимыми от конкретного смысла экспериментальных данных и состава аппаратуры, с которыми работает система в данный момент, тогда как другие – не могут (например, некоторые рабочие и визуализирующие модули). Соответственно, вторые требуют переписывания либо перенастройки для поддержки изменений в составе аппаратуры и/или составе, компоновке, смысле данных. Поэтому на данный момент уже существуют как минимум три схемы реализации модулей обработки и представления данных, в хронологическом порядке это (плюсы означают перечисление независимых друг от друга элементов одного уровня, стрелки – направление распространения данных):

1. схема ***statman (1)*** → ***histview (1)*** + ***cntview (1)*** (применялась для SPHERE DAQ [8] и SCAN DAQ [24], обе в ЛВЭ ОИЯИ);
2. схема дампер (dumper) + ***analyser (1)*** (применялась для поляриметрических DAQ в ЛВЭ ОИЯИ [25–31] и на экспериментальных стендах [32]);
3. схема ***b2r (1)*** → ***r2h (1)*** → клиентские ROOT–скрипты (применена для QUADRO DAQ на ФТФ ХНУ и предполагается для COMBAS DAQ в ЛЯР ОИЯИ).

Первая схема (см. рис. 1) наиболее гибка, но также наиболее громоздка и сложна для сопровождения. Используемая в ней полностью конфигурируемая утилита ***statman (1)*** требует файлы четырех типов:

– ***cell.conf (5)*** – описывает универсальные контейнеры данных (cell), служащие для проведения над ними вычислений;

– ***knobj.conf (5)*** – описывает так называемые объекты knobj'es, "известные (утилите) объекты", см. ***known (3)***;

– ***RUN.conf (5)*** – именует поля бинарных форматов событий, позволяя доступ к ним как к так называемым

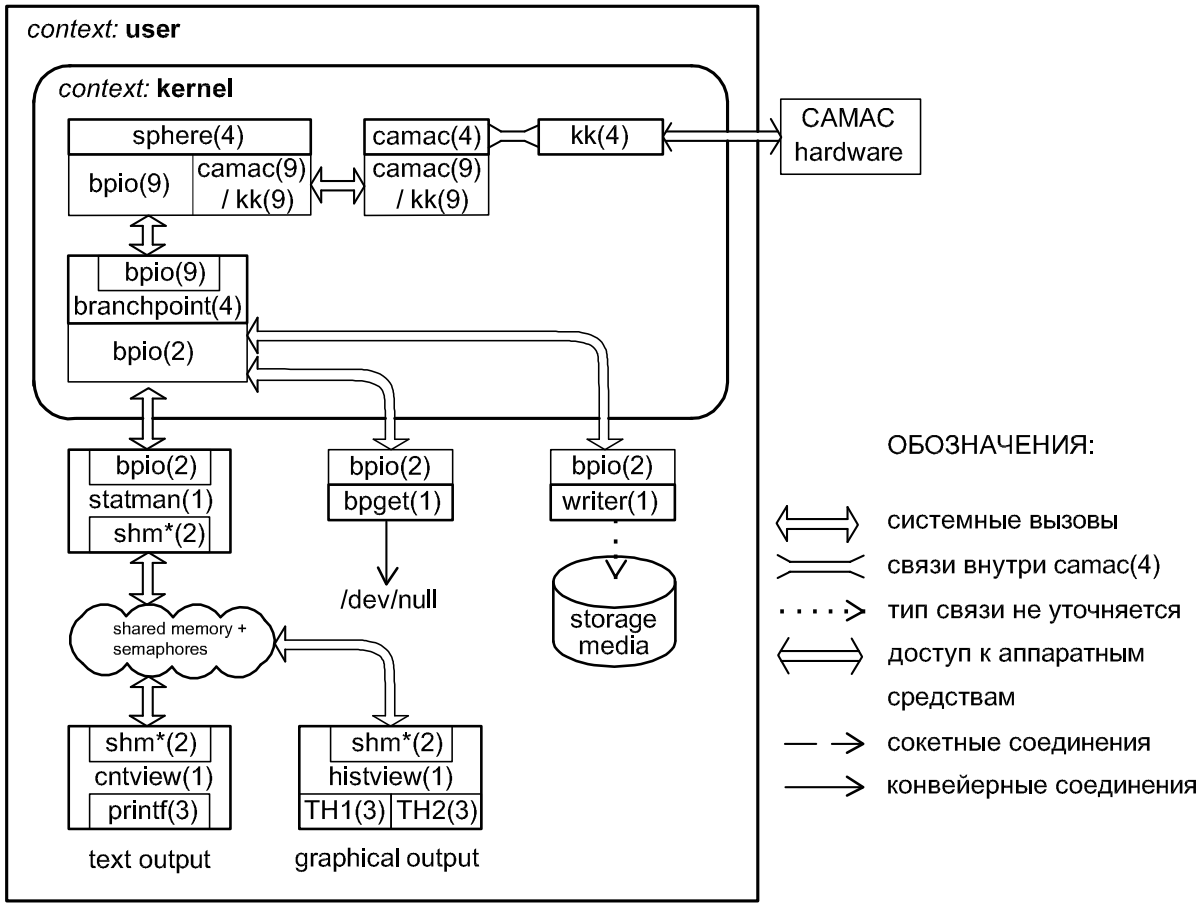

Рис. 1. Взаимодействие элементов системы *qdpb* в схеме визуализации ***statman (1)*** → ***histview (1)*** + ***cntview (1)***.

knvar's, "известным (утилите) переменным", см. ***known (3)***;
– ***clean.conf (5)*** – задает список knobj'es, подлежащих очистке по сигналу `SIGUSR2`.

При запуске ***statman (1)*** отводит (alloc) в IPC shared memory области, адресуемые известными ключами, в которых поддерживает статистику в виде различных реализованных типов данных, а также сопровождает так называемые вычислительные ячейки (cell). В настоящий момент реализованы следующие типы данных (все имеют моды накопления и перезаписи):
– одномерные ***hist (3,5)*** и
– двумерные ***hist2 (3,5)*** гистограммы,
– одномерные ***coord (3,5)*** и
– двумерные ***coord2 (3,5)*** "координатные" гистограммы,
– наборы счетчиков ***cnt (3,5)***,
– гистограммируемые массивы ***arrhist (3,5)***, и
– некое представление данных проволочных камер ***cham (3,5)***.

Прочитывая событие из потока пакетов, ***statman (1)*** преобразует его из бинарного формата в структурированный, который позволяет обращаться по (возможно, осмысленным) именам переменных, а не по сдвигам (offset) внутри события. Такое преобразование осуществляется функциями `b2s_<event_type>()` для соответствующих типов событий. Поэтому ***statman (1)*** требует лишь перекомпиляции (но не переписывания, как было бы, если бы он работал непосредственно с бинарным событием) при изменении формата события некоторого задействованного типа. Затем ***statman (1)*** выполняет цикл вычисления cell's и цикл заполнения/сброса knobj'es, оба параметризуемые типом события. То есть каждая ячейка производит предписанные ей вычисления, набор операций для которых примерно соответствует допустимым в правой части выражения присваивания языка C (поддерживаются также многие функции математической библиотеки C, *libm*), а каждый knobj вызывает свои функции заполнения и/или очистки лишь для соответствующих типов событий. Ячейки оперируют значениями knvar's, которые либо соответствуют величинам из события, либо являются внутренними переменными ***statman (1)***, с помощью функций ***known (3)*** становящимися knvar's, производя над ними вычисления и сохраняя результат требуемого типа. knobj'es оперируют результатами вычислительных ячеек и значениями knvar's, сохраняя результаты своих действий в shared memory.

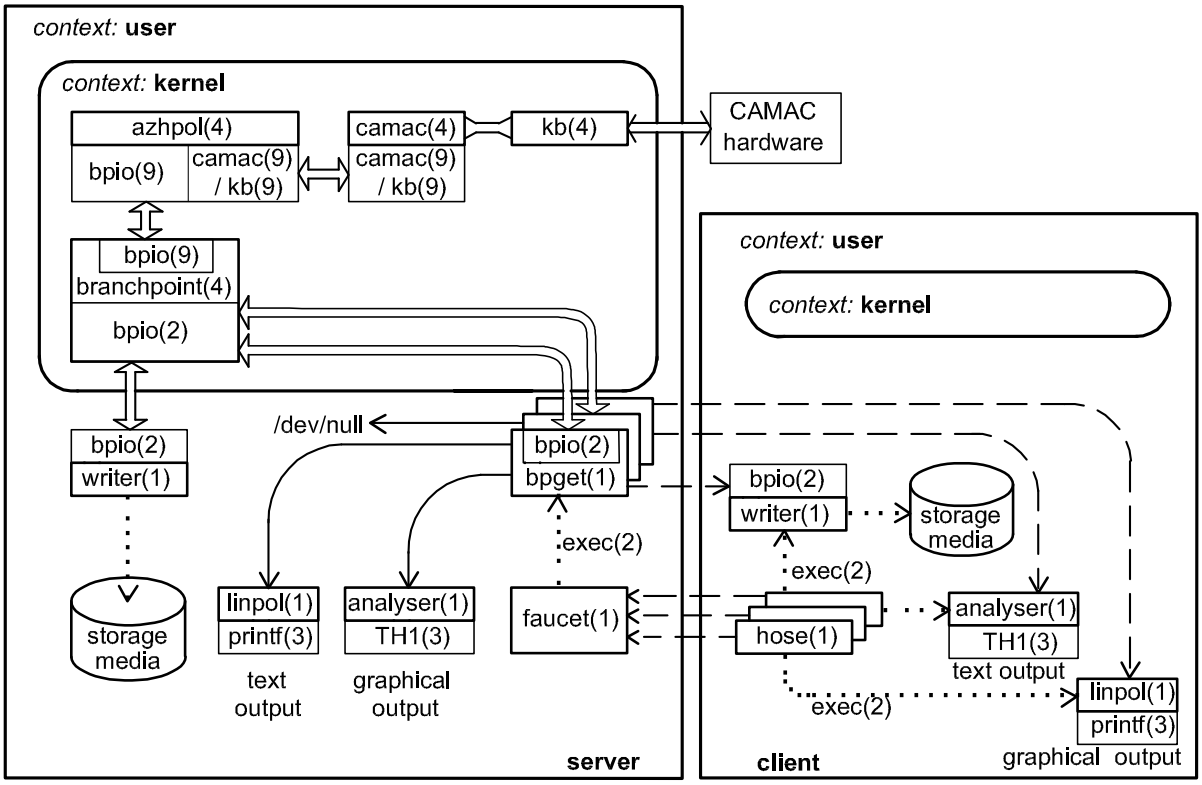

Рис. 2. Взаимодействие элементов системы *qdpb* в схеме визуализации dumper (в данном случае ***linpol(1)***) + ***analyser(1)***.

Полностью конфигурируемые (каждый требует файл типа ***knobj.conf(5)***) утилиты визуализации ***histview(1)*** и ***cntview(1)*** подключаются к тем же, что и ***statman(1)***, областям shared memory, потому что используют те же ключи. Синхронизация доступа осуществляется с помощью IPC семафоров. Раз в заданный период времени (обычно 1–2 раза за цикл ускорителя) визуализаторы выполняют цикл заполнения/сброса собственных knobj'es, таким образом вычитывая и отображая накопленную статистику. В настоящий момент для ***histview(1)*** реализованы типы графического представления данных из shared memory в виде одномерных ***TH1(3,5)*** и двумерных ***TH2(3,5)*** гистограмм (подходят для любых гистограммных типов knobj'es, поддерживаемых утилитой ***statman(1)***); для ***cntview(1)*** – тип текстового представления набора счетчиков ***dcnt(3,5)***. Заметим, что ***histview(1)*** пользуется гистограммными классами ROOT `TH1*`, `TH2*` лишь для отображения гистограмм, каждый раз заново заполняя их данными из shared memory. Посредством класса `TBufferFile` можно было бы разместить в shared memory обращенные в последовательный вид ("сериализованные") с помощью функций `Streamer()` собственно классы `TH1*`, `TH2*`, а не данные для них, таким образом поручив работу с ними непосредственно ***statman(1)***'у. Возможно, это упростило бы программирование, но наверняка увеличило бы накладные расходы – заполняющие функции гистограмм и их `Streamer()`'ы вызывались бы при получении каждого события, а не при каждой прорисовке (то есть существенно реже), как сейчас. Так или иначе, на момент реализации первой схемы удобного способа сериализации классов не было, т.к. класс `TBufferFile` отсутствовал.

Вторая схема (см. рис. 2) – самая простая из всех, первоначально разработанная для стендов. Об этом говорит и название утилиты ***analyser(1)***, примерно реализующей функциональность одноименных приборов прежних времен. В ней предполагается, что событие (возможно, не с начала, а с неким смещением) содержит фиксированное число 2–байтовых величин, каждая из которых означает номер канала соответствующей гистограммы, в который нужно добавить единицу. Исходя из этого ***analyser(1)*** накапливает гистограммы и с заданным интервалом времени прорисовывает те из них, что ему предписаны конфигурационным файлом ***analyser.conf(5)*** (см. рис. 4). Dumper занимается выдачей текстовой информации и требует исправлений и перекомпиляции для перехода на другие данные.

В третьей схеме (см. рис. 3) утилита ***b2r(1)*** переводит данные из "сырого" бинарного формата в пособытийное ROOT–представление `Event`, которое передается дальше 1) с помощью `TMessage` через `TSocket`, или 2) посредством `TBufferFile` через совместно доступную область памяти, которая организуется с помощью механизмов shared memory или ***mmap(2)***, или 3) в виде пакетов, тела которых содержат сериализованные по-

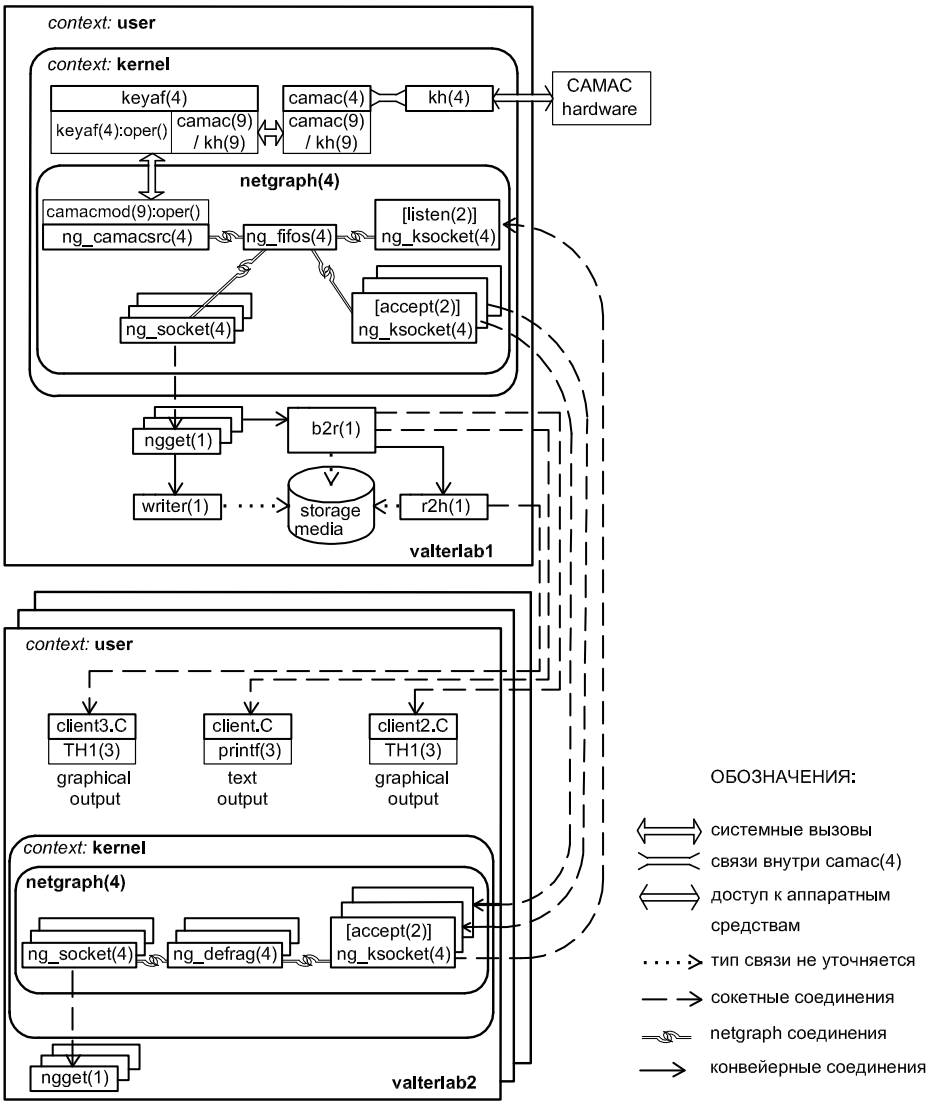

Рис. 3. Взаимодействие элементов системы *ngdp* в схеме визуализации ***b2r(1)*** → ***r2h(1)*** → клиентские ROOT–скрипты.

средством `TBufferFile` `Event`'ы. Производимые `Event`'ы также могут быть сохранены на HDD в ROOT `TFile` в виде ROOT–класса `TTree`. Далее накопление гистограмм и т.п. может быть реализовано раздельно либо совмещено с визуализацией. В первом случае `Event`'ы поступают к утилите ***r2h(1)***, заполняющей гистограммы (в виде соответствующих гистограммных классов ROOT `TH1*`, `TH2*`) и т.п., в которой и "прячется" зависимость от смысла экспериментальных данных. Поэтому ***r2h(1)*** требует либо переписывания и перекомпиляции при каких–либо изменениях обрабатываемых данных, либо реализации некой схемы runtime–конфигурирования. ***r2h(1)*** передает гистограммы с помощью `TMessage` по запросам визуализатора (столь простого, что может быть ROOT–скриптом, например, *client3.C*), а также может сохранять их (при своем окончании и, возможно, время от времени) на HDD в ROOT `TFile`. Удобно здесь то, что завершение визуализатора не приводит к потере накопленных гистограмм, как во втором случае, который на данный момент реализован в виде ROOT–скриптов *client.C*, *client2.C*. Существует и более важная причина разделения накопления статистики и ее визуализации. Дело в том, что любая графическая выдача требует взаимодействия с оконной системой, всегда представляющего собой бесконечный цикл обработки так называемых событий оконной системы (event, не следует путать с событиями в смысле *qdpb*). Такой цикл неудобно (и не всегда возможно) совмещать в рамках единого потока исполнения с бесконечным же циклом чтения пакетов из некого входного потока. Для организации же отдельных потоков исполнения нужно разделение – либо на отдельные процессы (что и сделано), либо на отдельные нити ***pthread(3)***.

Вообще же к элементам системы *qdpb* следует относиться как к конструктору с набором типовых деталей, позволяющему собрать систему DAQ конкретной установки, исправляя либо реализуя заново лишь минимальное количество элементов. В настоящий момент система *qdpb* содержит порядка 60000 строк оригинального

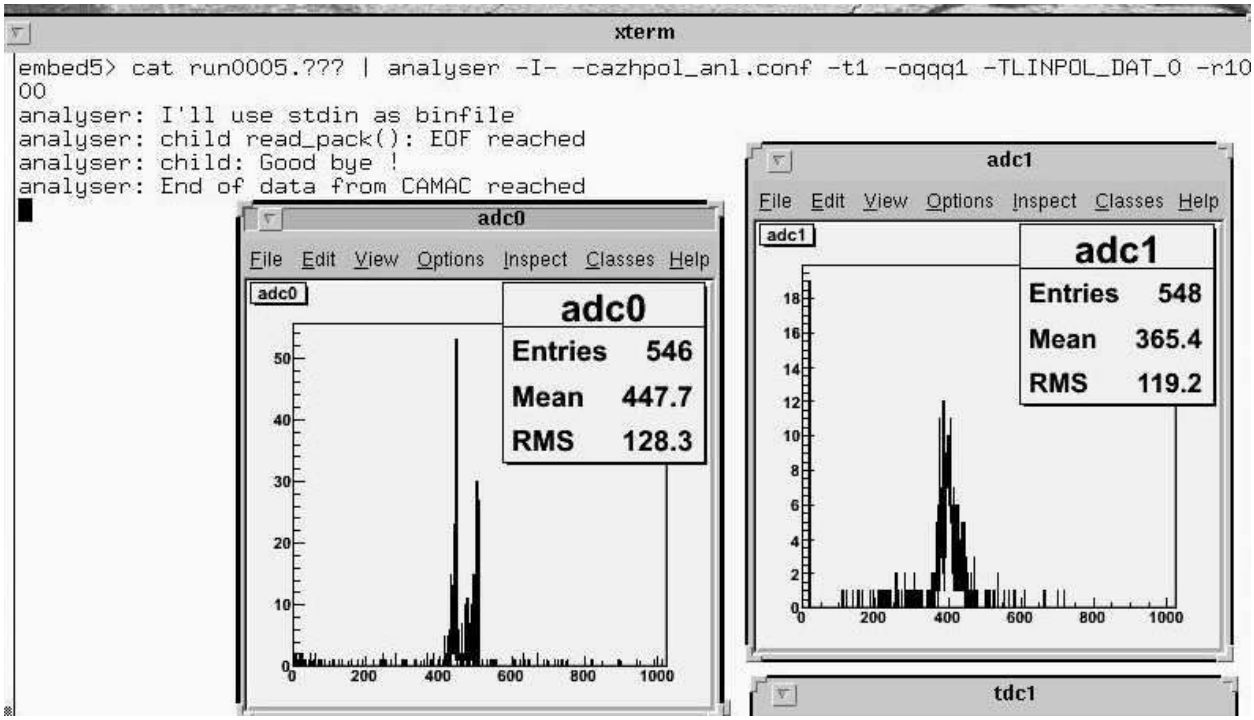

Рис. 4. Одномерные гистограммы ***TH1 (3)*** пакета ROOT, построенные и отображенные модулем ***analyser (1)***.

кода на C и C++ и 30000 строк документации.

Простейшие действия над отдельными элементами системы (загрузка, выгрузка, конфигурирование, инициализация, посылка сигнала, тестирование и т.п.) собраны в единый Makefile, то есть конфигурационный файл утилиты ***make (1)***. Оригинально такой файл представляет собой описание связей и зависимостей при генерации файлов и программ, но близко подходит для нашей задачи соотнести наборы упомянутых действий с командами оператора, такими как запуск или остановка набора данных и т.п. Таким образом, команды оператора представлены так называемыми target'ами Makefile'а, а их исполнение сводится для самого оператора к выполнению командной строки примерно следующего вида

```
make -f sv.conf continue
```

Поверх командно–строчного интерфейса модуль ***sv (1)*** предоставляет GUI (см. рис. 5), так что, например, по нажатию кнопки Continue будет выполнена вышеприведенная команда.

Подробное описание системы *ngdp*, задуманной для достижения максимальной производительности и пропускной способности для данных аппаратных средств, выходит за рамки данной работы. Хотя ее реализация принципиально отличается от системы *qdpb*, идеологически с пользовательской точки зрения они близки. Поэтому ограничимся кратким проведением параллелей между этими двумя инфраструктурными системами.

Так, в *ngdp* вместо точки ветвления реализованы:

– интерфейс между обработчиком прерываний КАМАК и системой ***netgraph (4)*** – нода (то есть узел, "вершина графа" системы ***netgraph (4)***) типа ***ng_camacsrc (4)*** (позволяет получать данные от нескольких обработчиков прерываний, то есть драйверов КАМАК, на одном и том же компьютере);

– нода типа ***ng_fifo (4)***, поддерживающая несколько буферных дисциплин и позволяющая отправлять поток(и) пакетов как в контекст задачи локального компьютера (посредством ***ng_socket (4)***, см. ниже), так и в контекст ядра удаленного (remote) компьютера по протоколу TCP/IP. В последнем случае вместо пары процессов ***faucet (1)***, ***hose (1)*** используется пара нод – локальная и удаленная – типа ***ng_ksocket (4)***, стандартно предоставляемого системой ***netgraph (4)***.

Вместо сервисных модулей интерфейса с точкой ветвления (***bpput (1)***, ***bpget (1)***) используются сервисные модули интерфейса с системой ***netgraph (4)*** – ***ngput (1)***, ***ngget (1)***. Стандартный механизм такого взаимодействия, позволяющий передавать и получать как данные, так и так называемые управляющие сообщения (control message) – сокет в специальном домене, являющийся также нодой типа ***ng_socket (4)***.

Отметим, что на любом компьютере, входящем в систему DAQ, одновременно может исполняться (продолжая аналогию ***netgraph (4)***'а с OOP реализацией, можно сказать "инстанцироваться", instantiation) любое (ограниченное лишь ресурсами памяти) количество экземпляров ("инстанций", instance) правильно реализованной ноды любого типа ("класса", class), тогда как в *qdpb* точка ветвления в виде системного вызова может быть

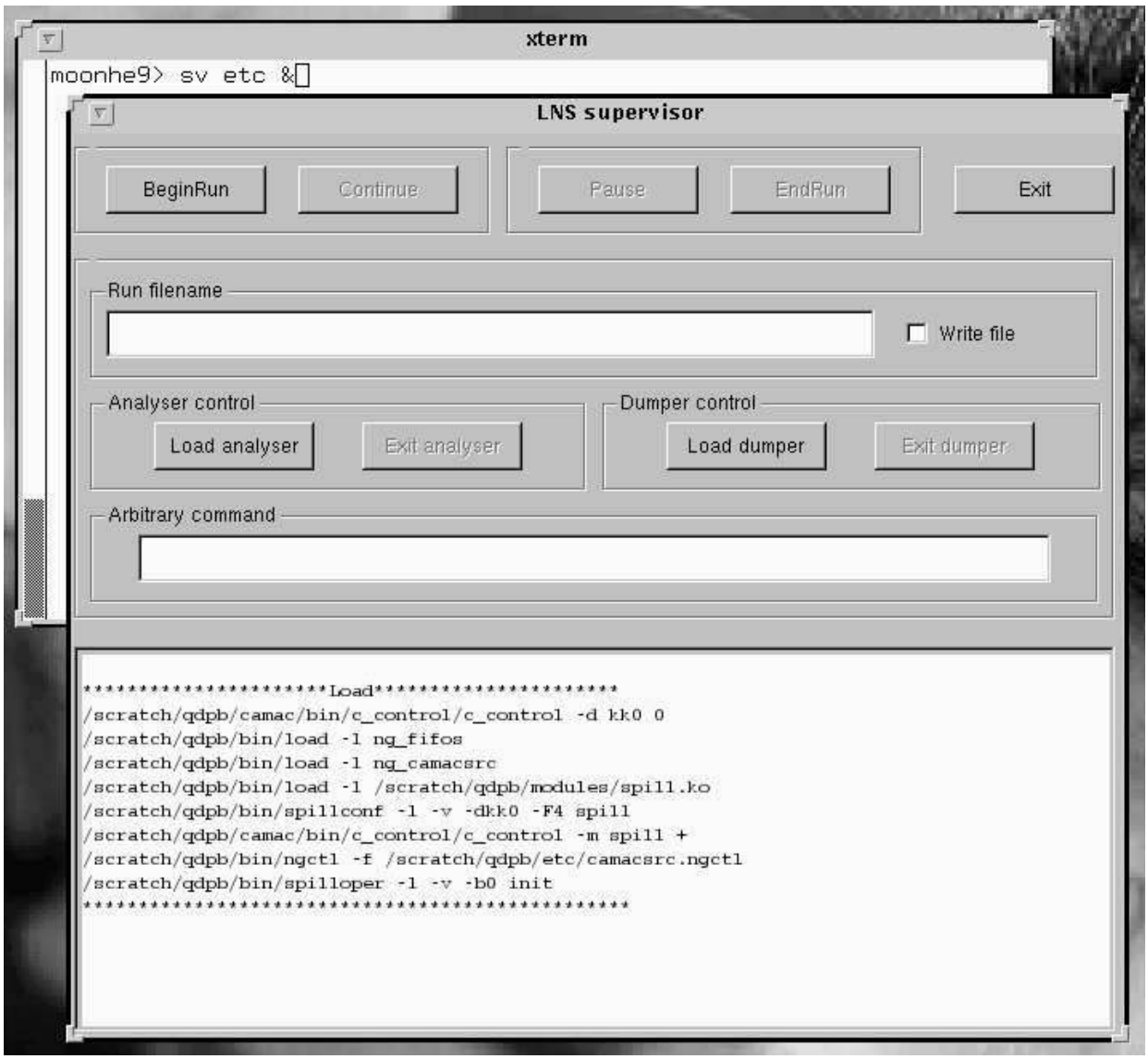

Рис. 5. GUI управляющего модуля ***sv(1)***.

только одна (либо размножена путем занятия многих номеров системных вызовов, что совершенно неудобно), а следовательно, и источник прерываний КАМАК только один.

Для принимающего (удаленного) компьютера реализована нода типа ***ng_defrag(4)***, дефрагментирующая пакеты после пересылки через пару ***ng_ksocket(4)***'ов, после чего поток пакетов может либо быть выведен в контекст задачи, либо продолжать прохождение через ***netgraph(4)*** – например, на один из входов ноды типа ***ng_em(4)*** или ***ng_pool(4)***. Нода ***ng_em(4)*** представляет собой сшиватель событий (Event Merger, отсюда название) как некий простейший вариант построителя событий, event builder'а. Соответствующий вариант точки ветвления, ***eventmerger(4)*** был задуман еще для *qdpb*, но ни разу не применялся. Нода ***ng_pool(4)*** осуществляет копирующее (без удаления из буфера) получение каждого N–го пакета запрошенного типа из ***ng_fifo(4)***, предоставляя таким образом выборку для некой express online предобработки и/или визуализации в системе с большим полным потоком данных.

В достаточно простых вариантах систем DAQ могут быть использованы "самотечные", так называемые ASAP (As Soon As Possible, "как только так сразу") варианты нод: ***ng_fifos(4)***, ***ng_ems(4)*** (***s*** от Simple), ***ng_bp(4)*** ("самотечный" вариант ***ng_pool(4)***). Это значит, что данные обрабатываются сразу по получении и передаются сразу по готовности к передаче, при этом промежуточная буферизация с последующей "досылкой" в случаях, когда синхронная передача невозможна, реализуется самим ***netgraph(4)***'ом. В такой идеологии "первотолчка" от обработчика прерываний, порождающего и отправляющего пакет, хватает для прохождения пакета "самотеком" через всю систему DAQ (см. также рис. 6), то есть нигде в ней не требуется потоков

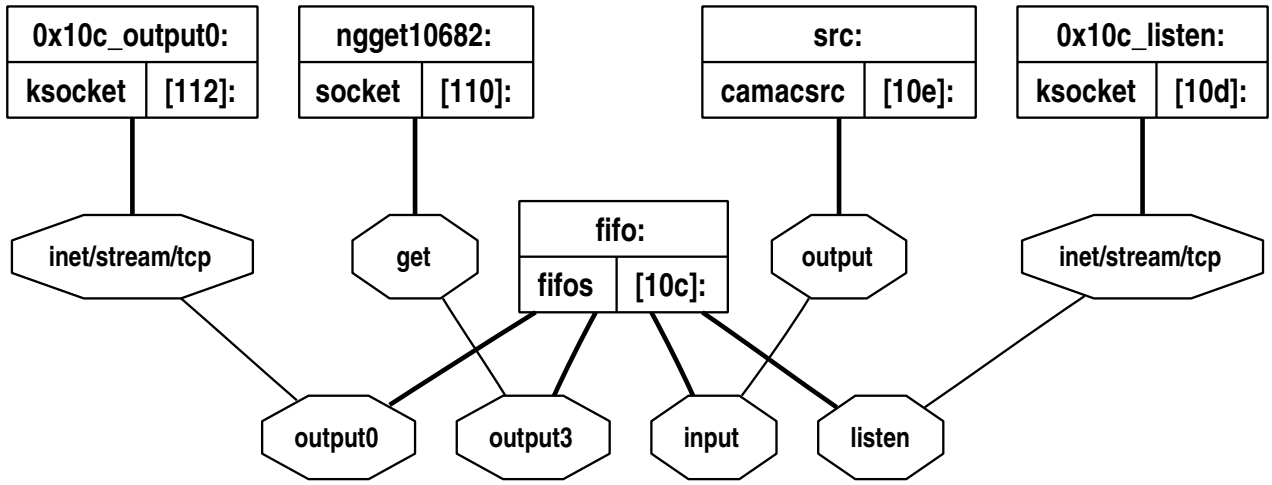

Рис. 6. Граф системы *ngdp*, реализующий "самотечную" DAQ QUADRO.

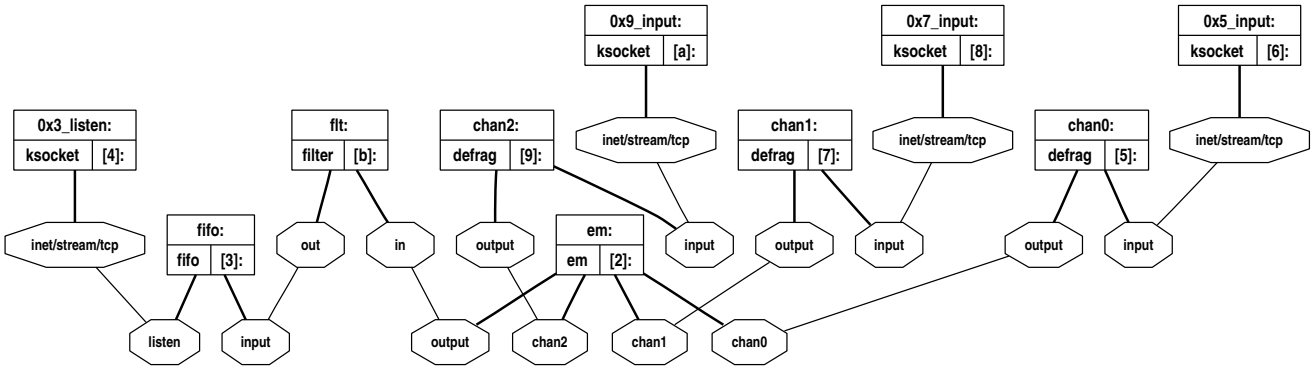

Рис. 7. Граф системы *ngdp*, реализующий уровень SubEvB.

исполнения.

Для более сложных многоуровневых систем DAQ такое синхронное прохождение невозможно, поэтому для каждого уровня предполагается наличие хотя бы одного автономного потока исполнения (в виде нити ядра, kthread), который занимается посылкой запросов (в виде управляющих пакетов, control packet) на нижележащий (относительно направления потока данных) уровень, на котором нода ***ng_fifo(4)***, поддерживающая в своем буфере очереди (queue) пакетов данных с необходимыми правилами (disciplines) помещения и выемки, отвечает либо отправкой пакета данных, либо ответным пакетом (answer packet) с кодом ошибки. В существующей реализации нити встроены в ноды ***ng_em(4)*** и ***ng_pool(4)***. При этом типовой уровень на одном компьютере выглядит как `ng_defrag` → `ng_em` → `ng_filter` → `ng_fifo` (см. также рис. 7).

Нода типа ***ng_filter(4)*** предназначена для фильтрации пакетов по правилам, реализованным в виде функций в отдельных KLD–модулях, и предоставляет интерфейс для подключения и отключения последних во время исполнения (своего рода plug–in). Возможно также пропускание потока через внешний фильтр, вход которого подсоединен к hook'у `subout`, а выход – к `subin`. (Соединенные пары hook'ов представляют собой "ребра графа" системы ***netgraph(4)***, вдоль которых в ней передаются данные.) Фильтрация в принципе может как угодно преобразовывать пакет либо полностью отбросить его и/или сформировать новый. Правила фильтрации применяются в порядке их подключения. Внешний фильтр, которым может быть как нода (цепочка нод), так и пользовательский процесс (конвейер процессов), применяется последним (если он присоединен).

Для управления элементами системы ***netgraph(4)*** используются управляющие сообщения, оригинально реализованные лишь локально. Для удаленной доставки таковых необходимы реализация специальной ноды (например, типа ***ng_sv(4)*** – NetGraph SuperVisor) и некоторые исправления базовой реализации ***netgraph(4)***, опыт работы с которой показал также несколько других мест (***ngctl(8)***, ***netgraph(3)***) для возможных улучше-

ний под наши задачи. В настоящий момент система *ngdp* содержит порядка 50000 строк оригинального кода на C и C++ и 6000 строк документации.

ВЫВОДЫ

Таким образом, построение систем сбора данных под современными UNIX–подобными операционными системами, с одной стороны, требует серьезных знаний архитектуры и концепций последних, умения "вписываться" в их стиль и использовать довольно широкий арсенал системных средств и интерфейсов. С другой стороны, это окупается, позволяя строить модульные системы сбора данных под различные задачи экспериментальной ядерной физики. Модульная организация позволяет легко распределять и масштабировать системы DAQ, а также изолировать код, специфический для той или иной конкретной экспериментальной установки, и реализовать набор неспецифических модулей (инфраструктурную систему), использующихся в неизменном виде в любых системах DAQ. Инфраструктурный подход минимизирует усилия по разработке и сопровождению многих систем DAQ, поскольку один и тот же отлаженный код можно применять снова и снова. В работе подробно рассмотрена инфраструктурная система *qdpb*, а также три схемы реализации на ее основе систем DAQ, применявшихся на нескольких установках в ЛВЭ ОИЯИ. Рассмотренные различия этих схем сосредоточены на заключительном этапе обработки и в визуализации данных, тогда как сбор и транспортировка реализованы однотипно. Система *ngdp* как развитие инфраструктурного подхода реализует модули в контексте ядра иным, чем *qdpb*, образом (в стиле пакета *netgraph*), что улучшает производительность и масштабируемость, но совместима на уровне модулей пользовательского контекста.

СПИСОК ЛИТЕРАТУРЫ

1. Система КАМАК. Крейт и сменные блоки. Требования к конструкции и интерфейсу: ГОСТ 26.201–80. – М.: Изд–во стандартов – 21.04.1980.
2. IEEE Standard Portable Operating System Interface for Computer Environments. // IEEE Std 1003.1. – 1988.
3. Bach M.J. The design of the UNIX operating system. New Jersey: Prentice–Hall Corp., 1986.
4. Vahalia U. UNIX internals: the new frontiers. New Jersey: Prentice–Hall Corp., 1996.
5. Робачевский А.М. Операционная система UNIX. СПб.: БХВ–Петербург, 2000.
6. Gritsaj K.I., Isupov A.Yu. A Trial of Distributed Portable Data Acquisition and Processing System Implementation: the *qdpb* – Data Processing with Branchpoints // JINR Communications, E10–2001–116. – 2001. – P.1–19.
7. Isupov A.Yu. Configurable data and CAMAC Hardware Representations for Implementation of the SPHERE DAQ and Offline Systems // JINR Communications, E10–2001–199. – 2001. – P.1–16.
8. Isupov A.Yu. SPHERE DAQ and off–line systems: implementation based on the *qdpb* system // JINR Communications, E10–2003–187. – 2003. – P.1–17.
9. Brun R., Rademakers F. ROOT – An Object Oriented Data Analysis Framework. // In Proc. of the AIHENP'96 Workshop, Lausanne, Switzerland. – Nucl. Instr. and Meth. in Phys. Res. A – 1997. – Vol. 389. – P.81–86.
10. Brun R., Buncic N., Fine V., Rademakers F. ROOT. Classes Reference Manual // CodeCERN, 1996.
11. `http://root.cern.ch/` .
12. Quercia V., O'Reilly T. Volume Three: X Window System User's Guide // O'Reilly & Associates, 1990.
13. Cutler E., Gilly D., O'Reilly T. The X Window System in a Nutshell // O'Reilly & Associates, 1992.
14. Mui L., Pearce E. Volume Eight: X Window System Administrator's Guide // O'Reilly & Associates, 1992.
15. Ольшевский В.Г., Помякушин В.Ю. Использование ОС UNIX на управляющей ЭВМ установки МЮСПИН // Сообщения ОИЯИ, Р10–94–416. – 1994.
16. Грицай К.И., Ольшевский В.Г. Программный пакет для работы с КАМАК в операционной системе FreeBSD // Сообщения ОИЯИ, Р10–98–163. – 1998.
17. ESONE Committee. Subroutines for CAMAC // ESONE/SR/01 – 1978.
18. Георгиев А., Чурин И.Н. // Сообщения ОИЯИ, Р10–88–381. – 1988.
19. Антюхов В.А., Журавлев Н.И., Игнатьев С.В. и др. Цифровые блоки в стандарте КАМАК (выпуск XVIII) // Сообщения ОИЯИ, Р10–90–589. – 1990. – С.20.
20. Антюхов В.А., Журавлев Н.И., Игнатьев С.В. и др. Цифровые блоки в стандарте КАМАК (выпуск XVIII) // Сообщения ОИЯИ, Р10–90–589. – 1990. – С.16.
21. Базылев С.Н., Слепнев В.М., Шутова Н.А. Контроллер крейта КАМАК ССРС4 на базе полнокомплектного IBM PC // В трудах XVII Международного симпозиума по ядерной электронике NEC'97, Варна, Болгария. – Дубна: ОИЯИ, 1998. – С.192.
22. `http://ccpc4.jinr.ru/ccpc` .
23. СС02. Руководство пользователя. Харьков: ООО "РиЭС", 2004.
24. Афанасьев С.В. и др. Система сбора данных и триггер установки СКАН // ПТЭ. – 2008. – Т.1. – С.34–39.
25. Isupov A.Yu. DAQ System for High Energy Polarimeter at the LHE, JINR: Implementation Based on the *qdpb* (Data Processing with Branchpoints) System // JINR Communications, E10–2001–198. – 2001. – P.1–15.
26. Isupov A.Yu. Software for realtime polarimetry on the LHE Nuclotron: design, implementation and usage. // In Proceedings of the XIX International Symposium on Nuclear Electronics and Computing NEC'2003, Varna, Bulgaria. – Dubna: JINR, 2004. – P.157–163.

27. Анисимов Ю.С. и др. Поляриметр для внутреннего пучка Нуклотрона // Письма в ЭЧАЯ. – 2004. – Т.1, №1. – С.68–79.
28. Isupov A.Yu. DAQ systems for the High Energy and Nuclotron Internal Target Polarimeters with network access to polarization calculation results and raw data // JINR Communications, E10–2004–13. – 2004. – P.1–12.
29. Isupov A.Yu. Data acquisition systems for the high energy and Nuclotron internal target polarimeters with network access to polarization calculation results and raw data // Czech. J. Phys. Suppl. – 2005. – Vol. A55. – P.A407–A414.
30. Исупов А.Ю. Измерения тензорной анализирующей способности T_{20} в реакции фрагментации дейтронов в пионы под нулевым углом и разработка программного обеспечения для систем сбора данных установок на поляризованных пучках. // Автореферат диссертации к.ф.–м.н., спец. 01–04–01 и 01–04–16. – Дубна: ОИЯИ 1-2005-169, 2005.
31. Isupov A.Yu. Upgrade of the DAQ systems for the LHE polarimeters to support Vector–Tensor Polarimeter on the Nuclotron internal target // Czech. J. Phys. Suppl. – 2006. – Vol. C56. – P.C385–C392.
32. Isupov A.Yu. Software utilities for using on an experimental stand. // In Proceedings of the International Workshop – Relativistic Nuclear Physics: from Hundreds of MeV to TeV, RNP'2005, Dubna, Russia. – Dubna: JINR, 2006. – P.252–259.

IMPLEMENTATION OF DATA ACQUISITION SYSTEMS FOR MULTICHANNEL NUCLEAR PHYSICS SETUPS ON BASE OF THE UNIX–LIKE OPERATING SYSTEMS

A.Yu. Isupov[1], **V.E. Kovtun**[2], **A.G. Foshchan**[2]
[1]*Laboratory of High Energy Physics, Joint Institute for Nuclear Research*
Russia, 141980, Dubna, Joliot–Curie str., 6
[2]*Department of Physics and Technology, V.N.Karazin Kharkov National University*
Ukraine, 61108, Kharkov, Kurchatov av., 31
E–mail: isupov@moonhe.jinr.ru

Some ideas and principles, used at design of software for data acquisition systems on base of the modern UNIX–like operating systems, are considered. In particular, ones follows: wide using of interprocess and intercomputer interfaces for data transfer, elimination of intermediate storages on HDD, modular structure allows to easy distribute and scale data acquisition system, implementation of typical framework systems, and so on. The comparison of a UNIX–like operating systems and DOS as environments for data acquisition software implementation is done. The framework provided by the *qdpb* system, which allows to implement a modular distributed systems for data acquisition, transfer, processing, and visualization for various nuclear physics experimental setups with electronic readout, is described. Some used at the Laboratory of High Energies (LHE) of the Joint Institute for Nuclear Research (JINR, Dubna) schemes of *qdpb* framework based implementation of the data acquisition systems for such setups are explained. A some variant of the data acquisition frameworks future development – the *ngdp* system – is presented briefly.

KEY WORDS: computer, CAMAC, software, operating system, data acquisition.