**Морской бой**

Реализовать программу для игры в морской бой на локалхосте. Обмен данными через общий файл, так что стоит подумать про блокировки. (С)тырено у tat@jscc.ru

**Уведомления на телефон**

Написать plugin для libpurple, который бы пересылал все входящие сообщения (или уведомления о них) по протоколу SIP, либо на написанное в рамках этого же приложение для какой-либо мобильной ОС. Наброски проекта уже есть, ссылки на документацию отдельно.

**Доска сообщений**

Клиент отправляет сообщения через сокеты или SysV (или POSIX) message queue, сервер показывает последние N и может отдавать их через сокеты или общую память.

**Связь клиентов за NAT'ом**

Написать софт (клиент + сервер), позволяющий установить прямое соединение между клиентами, находящимися за NAT. Простой алгоритм: <https://en.wikipedia.org/wiki/UDP_hole_punching>

**Useless**

Из задавальника.

**Backup**

Из задавальника.

**Посудомойка**

Из задавальника, но лучше попилить на вариации - например, программа, выводящая сколько её раз запускали (используя расшареную память и семафоры).

**Disk Image Tools**

DITO (Disk Images TOols) - <https://github.com/thomasloven/dito> - убирание из кода неприятных компилятору конструкций, допиливание поддержки других файловых систем.

**Коробок**

Защищенный дропбокс с шифрованием контента на клиенте. Open-source, шифрование по ГОСТу. Есть рабочий прототип, работающий из командной строки. Нужно приделать интерфейс. Скорее всего, понадобится FUSE (File System in User Space).

**we.c**

Написать программу \*we.c\* в три этапа (результат каждого этапа должен быть помещен на сервер отдельным коммитом!):

1. Программа we.c (процесс-родитель) должна с помощью системного вызова fork() создать дочерний процесс (процесс - ребенок). Процесс-родитель должен выдать на консоль свой номер, а процесс-ребенок свой номер и номер родительского процесса. Возможная ошибочная ситуация при вызове fork() должна быть корректно обработана с помощью библиотечного вызова perror();
2. Изменить программу так, чтобы процесс-ребенок гарантированно усыновлялся процессом init и сообщал об этом на консоль.
3. Сделать так, чтобы в программе создавался процесс-внук, или процесс-брат, или оба процесса вместе в зависимости от Вашего индивидуального задания. Каждый процесс должен сообщить о себе на консоль, вывести свой номер и номер своего родителя.

**wefly.c**

Написать программу \*wefly.c\*, моделирующую следующую ситуацию. На кухне стоит банка с вареньем. До нее очень хотят добраться осы. Около банки стоит человек (пользователь программы), который взмахом руки может отгонять ос (если успеет, конечно). Взмаху руки соответствует нажатие CTRL-C, при этом экране должна появляться соответствующая фраза, например, «машу рукой». Осы летят по очереди, строго по одной. Когда оса находится в пределах досягаемости руки человека, программа должна сообщить на консоль, что летит оса. Эта фраза является сигналом для пользователя, который должен успеть нажать CTRL-C до того, как оса съест варенье. Если оса его съесть успела, то об этом сообщается на консоль. Если пользователь успел во время пролета осы нажать CTRL-C, то оса считается отогнанной, и об этом выводится сообщение на экран. Общее количество ос можно задать либо директивой #define, либо передать через аргумент командной строки. Человеку дается ровно столько же попыток махнуть рукой. Если попытки исчерпаны, а пользователь программы нажимает CTRL-C, на консоль нужно вывести сообщение о том, что рука устала. Программу проще всего написать в два этапа:

* Смоделировать полет ос. Полет каждой осы состоит из вывода на консоль сообщения о начале полета, паузы случайной длительности, задаваемой системным вызовом sleep(), вывода сообщения о том, что оса наелась и еще одной паузы случайной длительности. Если пользователь программы нажмет CTRL-C во время второй паузы, взмах руки пропадет впустую. Вторая пауза должна гарантированно длиться заданное время. Этого можно добиться с помощью проверки возвращаемого значения sleep(). См. man sleep. Когда пролетят все осы, нужно вывести на печать их общее количество, количество отогнанных и количество наевшихся.
* Смоделировать взмах руки в обработчике прерывания по CTRL-C. В этом обработчике после вывода на консоль сообщения о взмахе руки нужно сделать проверку. Если оса была в полете, то отогнать ее, если нет, то учесть, что взмах был впустую. Здесь же логично следить за оставшимся числом взмахов руки и выводить сообщение о том, что рука устала. Возможная ошибочная ситуация при вызове signal() должна быть корректно обработана с помощью библиотечного вызова perror().

**alive.c**

Написать «упрямую» программу с бесконечным циклом и установкой собственных обработчиков сигналов, используя системный вызов signal(), которая при нажатии CTRL-C (сигнал SIGINT), CTRL-Z (сигнал SIGTSTP) или CTRL-4 (сигнал SIGSTOP) не будет завершаться, а будет печатать какие-нибудь фразы на экране, разные для каждой комбинации клавиш.

**Модуль логгирования log.c и описание его интерфейса log.h**

1) Реализовать отдельный модуль логгирования log.c, который отвечает за трассировку состояний программы, чтобы пользователь впоследствии мог отследить в том числе корректность ее работы.

У этого модуля должен быть следующий **интерфейс** (декларируется в log.h):

/\*!

Creates a new log file.

\param filePath - path where the log should be created e.g. /tmp/log.txt

\return 0 if successful or -1 if error

\*/

int createLog(char \* filePath);

/\*!

Adds a new message to the log.

\note the log should be successfully created via createLog() prior to this call

\param message - the message to be logged

\*/

void addMessage(char \* message);

/\*!

Closes the log, so it couldn't be used after

\note the log should be successfully created via createLog() prior to this call

\return 0 if successful or -1 if error

\*/

int closeLog();

**Требования к реализации:** Этот интерфейс должен быть реализован в отдельном модуле log.c

* createLog() - открывает файл ( например, через fopen() )
* addMessage() - добавляет сообщение в открытый файл, причем в начале каждого сообщения в файле должно печататься время, когда это сообщение было добавлено (см. пример далее)
* closeLog() - закрывает файл лога, чтобы его содержимое сохранилось на диске.

Пример вида лога:

Feb 13 15:46:11 2013 - The program started.

Feb 13 15:46:12 2013 - A new client connected!

Пример, как можно сделать вывод времени, можно посмотреть тут: <http://www.cplusplus.com/reference/ctime/asctime> , формат времени (то есть как и в каком порядке записан год, месяц, число и тп) - выбирайте сами.

\*2) Добавить\* модуль логирования в индивидуальную задачу из прошлого семестра. (Если индивидуальная задача недоделана, то добавьте в любую другую работающую из прошлого семестра.) Для этого нужно добавить в файл с программой #include «log.h» Тогда программа сможет пользоваться функциями модуля лога.

\*Далее\*, добавить в исходный код программы вызовы логгирования с соответствующими сообщениями в случае ошибок или важных этапов выполнения программы (например, добавление нового клиента к серверу, переход к новому файлу, блоку данных и тп). В начало программы - открытие лога с помощью createLog(), в конец - закрытие через closeLog().

**Класс логгирования, реализация на Си**

Условия схожи с теми, что есть в 10.1, только ставятся дополнительные требования к модулю.

Модуль должен поддерживать несколько одновременно работающих файлов логгирования: один для важных событий программы (например, добавился новый пользователь), другой для отладочных сообщений, третий - для ошибок. Для каждого из этих логов должен настраиваться: а) максимальная длина строки, которая может содержаться в логе, если входная строка превышает данную длину, то она не записывается б) максимальный размер файла лога после достижения которого запись в лог прекращается

Указание по решению: Интерфейс log.h должен быть изменен таким образом, чтобы функция создания лога создавала и возвращала структуру-описатель, содержащую все необходимые настройки для логгирования (файл, длинна строки, максимальный размер лога). Указатель на эту структуру должен передаваться в функцию записи сообщения, чтобы варьировать ее действия в зависимости от данных, указанных в ней. Структура должна быть объявлена в log.h

**Base64 multithreaded converter**

В протоколах сети Интернет бинарные данные (например файлы с картинками) часто перекодируются для передачи в формат base64 (см. <http://ru.wikipedia.org/wiki/Base64> )

Программа получает на вход имя файла, открывает этот файл. Программа запрашивает у системы размер файла и, например, на каждый 1 Мегабайт этого файла создает отдельный поток, который перконвертирует кусок этого файла в base64. В конце работы программы, она «сшивает» куски в base64 и сохраняет итоговый файл с именем «<имя-исходного-файла>.base64». Возможен другой (более разумный) вариант разбиения.

**Multiprocessor Backup**

Пусть есть папка /source в которой лежит несколько больших текстовых файлов. В папку /source.backup/ складываются архивированные копии этих файлов (бэкапы).

Для бэкапа, каждый файл разбирвается на некоторое количестов кусков (пусть по 10 Мб), для каждого куска в отдельном процессе вызывается программа gzip (одновременно может обрабатываться не более 4ех кусков), которая компрессирует этот кусок. Далее все эти файлы собираются в общий файл архива .tar (через программу tar). Эти файлы собираются в папке /source.backup Внимание! Для файла происходит резервное копирование только если не существует для него архива или дата создания архива произошло раньше даты последнего изменения файла в папке. Возможен другой (более разумный) вариант разбиения.

**weeat\_over\_network.c**

На самом деле в нашем цифровом сетевом мире даже осы и те кормятся удаленно по интернету и сетям. Теперь оса - это не просто поток на той же машине, а отдельный клиентский процесс на другой машине. Оса-клиент подключается к серверу и обменивается сним сообщениями об удачности ее кормежки.

Связь через сеть можно получить двумя способами: а) если интересно больше разобраться с устройством сети вообще (вне зависимости от операционки), то читай сокеты (они есть в Advanced linux programming, куча всего можно найти в инете) б) если интересно больше разобраться в линуксе, его администрировании и конфигурации, то можно сделать соединение через xinetd.

**Как это начиналось**

Настало время подкрепиться и самим осам. Для этого они приметили веранду, где как раз обедает целая семья. На столе стоит много интересной еды (например, варенье и яичница). Но чтобы добраться до стола, сначала нужно преодолеть какие-то опасности (например, отец семейства отгоняет осу веником, а в дедовом халате/усах можно и запутаться). Впрочем, просто добраться до еды мало, например, в варенье можно утонуть, а яичница безопасна, но даёт меньше *единиц пищи*. Кроме того, на столе можно успеть схватить только одну осиную порцию еды, после чего выгонять начинают всей семьёй и для полного пополнения энергии придётся делать несколько заходов. *Для стимуляции фантазии нужно прочесть*[*http://www.owl.ru/morits/det/secret13.htm*](http://www.owl.ru/morits/det/secret13.htm)*, там как раз описана наша ситуация.*

Необходимо реализовать программу, которая бы моделировала взаимодействие мира веранды с осами, при этом потоки-осы являются клиентами для потока-сервера, обсчитывающего взаимодействия в этом небольшом мире (*здравствуй, CS или Q3. В них ещё играют на Физтехе?*). В этой задаче максимально приветствуется творчество и если будет много сделано, то можно будет зачесть её как индивидуальную. Но следующие ключевые моменты должны обязательно присутствовать:

* поток-оса выбирает тип еды (случайно или по какой-то стратегии) и посылает запрос к потоку-серверу;
* поток-сервер живёт в вечном цикле приёма и обработки запросов (не забудьте предусмотреть специальное сообщение для команды завершения сервера!);
* получив сообщение, сервер выбирает тип опасности (например, случайно равновероятно) и определяет, подействовала ли эта опасность;
* существует понятие состояния осы: если опасность подействовала, то состояние меняется, например, на «отогнана» или «раздавлена», соотв. ответ отправляется потоку-клиенту и оса уходит на следующую попытку/выбывает из игры;
* если опасность миновала, то сервер смотрит на запрошенный тип еды и обрабатывает его, состояние осы меняется (например, на «накормлена» или «утонула») и вот уже тогда отправляется ответ;
* поток-оса обрабатывает полученный ответ и выполняет соотв. действие (выбывает из игры/идёт на следующий заход/успокаивается т.к. наелась и т.д.);
* при завершении сервер выводит значения счётчиков статистики по осам.

**Многопоточное кэширование и изменение файлов в памяти**

Несколько потоков запрашивают открыть и сохранить данные запрашиваемых по имени файлов в собственную папку потока, внося в них небольшие правки.

На старте, программа должна получить от пользователя количество и список файлов, которые нужно получить, например: 7 a.html b.html a.html b.html c.html c.html d.html

Программа запускает несколько (MAX\_FILE\_READERS) потоков-обработчиков, которые читают файлы из списка, добавляя к каждому файлу строчку «<p> People visited this page %u», где %u - количество запросов, произошедших к этому файлу и сохраняют их в отдельной папке «out%u» , где %u - номер потока с именем «%d\_%s» - %d - номер запроса этого потока, %s - имя запрошенного файла.

Так, на входных данных из примера и MAX\_FILE\_READERS=2 может быть вывод с такими файлами. out1\1\_a.html out2\1\_b.html out1\2\_a.html out2\2\_b.html out2\3\_c.html out1\3\_c.html out1\4\_d.html

Для того, чтобы оптимизировать скорость работы и учесть то, что к одному и тому же файлу могут обращаться несколько раз, при открытии нового файла этот файл проецируется в память через mmap. Если файл уже был открыт, то автоматически будет записываться его содержимое с необходимым добавлением «<p> People visited this page %u».

**Browser monitor**

Сделать программку, которая работает в интерактивном режиме и принимает команды вида:

* browse <file> - запускает в фоновом режиме веб-браузер (например, firefox), с параметром file (имя файла, который откроет браузер) и напишет pid порожденного процесса веб-браузера.
* close <id> - насильно закроет браузер по заданному pid
* wait <id> - ждет пока браузер не закроется пользователем
* show - показывает pid и имена открытых файлов всех браузеров
* quit - завершает программу

**HTTP echo server**

Посредством сокетов создать интернет-сервер, прослушивающий порт 80. На каждое сетевое соединение нужно создать отдельный поток обработчик, который должен а) получить сообщение от клиента б) сгенерировать ответ клиенту. Сам ответ в качестве своих частей будет содержать изначальный запрос клиента, данные о самом сервере, количество уже обработанных сервером запросов за все время начиная со старта, а также текущее время на сервере.

Вид ожидаемых запросов клиента: GET /index.html HTTP/1.1 + некоторое количество произвольных данных.

Конец сообщения-запроса отмечается комбинацией следующих символов: «\r\n\r\n». Ответ должен содержать (все строки разделены через пару «\r\n»)

HTTP/1.1 200 OK Date: <текущая дата, в формате Wed, 11 Feb 2009 11:20:59 GMT> Server: WASP Content-Type: text/html; charset=utf-8 Content-Length: <тут размер запрошенной страницы в HTML> Connection: close

<далее следует запрошенная страница в HTML>

Формат страницы: Initial request: <тут идет сам изначальный запрос, все «\r\n» в нем заменены на «<br>», так чтобы ответ считался просто HTML-документом> > <br> Request number: %d (текущий номер запроса).

Цель: зайти на запущенный локально сервер через веб-браузер, вбив в строке навигации [http://localhost](http://localhost/) и получить итоговую HTML-страницу.

**pipeparam.c**

Написать программу pipeparam.c, определяющую размер буфера в ядре, выделяемого под pipe, и пропускную способность передачи через pipe.

* Для определения размера буфера нам не нужен процесс-ребенок, достаточно запретить блокирование записи при заполнении буфера:

int a; // заводим переменную для сохранения текущих параметров дескриптора.

a = fcntl(pipefd[1], F\_GETFL); // получаем текущий набор флагов

a |= O\_NONBLOCK; // отключаем блокирование, тогда попытка записи при заполненном буфере вернёт ошибку

a = fcntl(pipefd[1], F\_SETFL, a); // применяем изменённый набор флагов

* Один из способов определения пропускной способности. Определив размер буфера, нужно завести свой временный буфер точно такого же размера (для минимизации накладных расходов на системные вызовы в процессе измерения), и для очистки буфера в ядре считать в него содержимое pipe через выходной дескриптор. Затем включить блокировку на место (точно так же получить набор флагов, потом a &= ~O\_NONBLOCK, потом применить новый набор флагов), создать дочерний процесс, записать в заранее заведённую структуру первый отсчёт времени (понадобится функция gettimeofday или clock\_gettime) и записать условленное (#define) число раз в pipe свой временный буфер. Дочерний же процесс условленное число раз считывает буфер и затем записывает в свою структуру второй отсчёт времени. Затем через pipe от родительского процесса передаётся значение первого отсчёта времени, дочерний считает суммарный объём данных и разность между отсчётами, вычисляя скорость передачи данных.
* Возможны вариации. Например, передавать второй отсчёт времени от дочернего процесса к родительскому.
* \_Настоятельно рекомендуется\_ не помещать весь код в main(), а выносить какие-то части (например, тело дочернего процесса) в отдельные функции.

**Base64 decoding server**

В протоколах сети Интернет бинарные данные (например файлы с картинками) часто перекодируются для передачи в формат base64 (см. <http://ru.wikipedia.org/wiki/Base64> )

Задача: написать приложения клиента и сервера. Клиенты устанавливают соединение с сервером и отправляют зашифрованные в base64 сообщения серверу. Сервер должен принимать сообщения, расшифровывать их и сохранять в специальный файл лога в котором будет хранятся сообщения вида:

<date>

<client-ip>

<base64>

<decoded>

Например:

17-05-2012 15:34 GMT +3

192.168.1.15

SGkgVGhlcmU=

Hi There

17-05-2012 16:44 GMT +3

192.168.1.16

QnllLUJ5ZQ==

Bye-Bye

Внимание! Клиентов может быть очень-очень много. Поэтому предлагается использовать select() для обработки запросов от всех клиентов. (да необязательно, есть и другие идеи годные)

**xinetd-based time server**

Сделать программку, совместимую с xinetd, которая принимает от клиента по stdin (стандртный поток ввода) комнды типа set time, get time.(время вымышленное, не системное, время с момента сотворения мира по древнерусскому календарю)

get time - получить время из общего разделяемого файла time.config set time - установить время через в файл time.config

формат хранения времени - произвольный.

(Добавление) Сделать тоже самое на 80м порте и HTTP-протоколе.

**Сервер поиска файлов**

Несколько потоков запрашивают поток-сервер найти и передать данные файлов по их имени и вывести данные на экран.

На старте, программа должна получить от пользователя количество, место поиска и список файлов, которые нужно получить. Например: 7 /home/user/www-server/ a.html b.html c.html d.html e.html f.html g.html

Запросы к серверу происходят посредством очереди сообщений: тип сообщения - номер потока, данные - имя файла размером до FILENAME\_MAX (пусть 256 байт). Серверный поток доджен найти в заданной папке поиска или в любой его подпапке заданный файл, прочесть его и переслать его данные с небольшими правками по очереди сообщений-ответов. Ответы на запросы потоки принимают через очередь сообщений-ответов, в типе сообщения - поток адресат и данные файла размером до DATA\_SIZE\_MAX (пусть 2 MByte) с дополнительной строчкой «<b>Reply to the requestor %d by file %s</b>», где %d - номер потока, а %s - имя запрошенного файла. Если окажется, что файла найти не удалось, то нужно переслать ответ «<b>404 File not found</b>»

После получения файла запрашивающий поток должен вывести его содержимое на экран. Внимание: данные двух разных файлов должны выводиться раздельно, строки их не должны перемешиваться друг с другом.

Всё ок, разве что константы системные надо брать из системных же заголовочных файлов, они там есть.

Для справки (по именам констант) то что у тебя нашёл:

#ifndef \_LINUX\_LIMITS\_H

#define \_LINUX\_LIMITS\_H

#define NR\_OPEN 1024

#define NGROUPS\_MAX 65536 /\* supplemental group IDs are available \*/

#define ARG\_MAX 131072 /\* # bytes of args + environ for exec() \*/

#define LINK\_MAX 127 /\* # links a file may have \*/

#define MAX\_CANON 255 /\* size of the canonical input queue \*/

#define MAX\_INPUT 255 /\* size of the type-ahead buffer \*/

#define NAME\_MAX 255 /\* # chars in a file name \*/

#define PATH\_MAX 4096 /\* # chars in a path name including nul \*/

#define PIPE\_BUF 4096 /\* # bytes in atomic write to a pipe \*/

#define XATTR\_NAME\_MAX 255 /\* # chars in an extended attribute name \*/

#define XATTR\_SIZE\_MAX 65536 /\* size of an extended attribute value (64k) \*/

#define XATTR\_LIST\_MAX 65536 /\* size of extended attribute namelist (64k) \*/

#define RTSIG\_MAX 32

#endif

С сообщениями есть некоторая проблема:

/\*

\* Scaling factor to compute msgmni:

\* the memory dedicated to msg queues (msgmni \* msgmnb) should occupy

\* at most 1/MSG\_MEM\_SCALE of the lowmem (see the formula in ipc/msg.c):

\* up to 8MB : msgmni = 16 (MSGMNI)

\* 4 GB : msgmni = 8K

\* more than 16 GB : msgmni = 32K (IPCMNI)

\*/

#define MSG\_MEM\_SCALE 32

#define MSGMNI 16 /\* <= IPCMNI \*/ /\* max # of msg queue identifiers \*/

#define MSGMAX 8192 /\* <= INT\_MAX \*/ /\* max size of message (bytes) \*/

#define MSGMNB 16384 /\* <= INT\_MAX \*/ /\* default max size of a message queue \*/

Тащем-то, запрашивающий может создавать pipe и передавать в запросе номер соотв. дескриптора, через этот pipe и получать инфу. Ну ещё можно нагло динамической памятью и указателями разрулить ситуацию, благо, всё равно в тредах живём.

**Universal file open**

Сделать программку, которая работает в интерактивном режиме и принимает команды вида:

* open <file.ext> - ищет в файле конфигурации autoopen.cfg необходимую команду для открытия файла с расширением .ext. Если такая программа есть - запускает ее в фоновом режиме. Если нету - то открывает в фоновом режиме какой-нибудь стандартный редактор (например, mcedit), чтобы пользователь смог внести изменения в файл конфигурации, ждет пока процесс редактора закончится и после этого делает попытку выполнить команду снова.
* show - показывает pid и имена открытых файлов
* quit - завершает программу

Формат файла autoopen.cfg: строка, начинающаяся с #, – комментарий и никак не обрабатывается. Все остальные строки вида: <ext>=<appname>

Пример файла:

# web-browser

html=firefox

# text

txt=mcedit

# just output to console

log=cat

**Сервер поиска владельца файлов**

Несколько потоков запрашивают поток-сервер найти и передать или изменить данные о владельце файлов по их имени и вывести данные на экран.

На старте, программа должна получить от пользователя количество, место поиска и список команд, которые нужно исполнить. Например: 7 /home/user/www-server/ get a.html get b.html get c.html change x.html root change e.html root get a.html get c.html

Запросы к серверу происходят посредством очереди сообщений: Есть два разных типов запроса: получение (get) пользователя и группы файла, изменение (change) пользователя-владельца файла.

На get: Серверный поток доджен найти в папке поиска заданный файл, прочесть его и переслать данные о владельце файла и группе через очередь сообщений-ответов. На change: Серверный поток передает данные о старом владельце и группе.

В случае ошибки (например, файл не найден), должно возвращаться специальное сообщение, содержащее код ошибки.

После получения данных запрашивающий поток должен вывести имя файла и его владельца и группу на экран. Внимание: данные для двух разных файлов должны выводаиться раздельно, строки их не должны перемешиваться друг с другом. Получать ответ на свой запрос должен именно тот поток, который запрашивал данные.

**HTTP Server prototype**

На основе любого вида транспорта (pipe, FIFO, message queue, socket) сделать простейший HTTP Server, который выполняет запросы типа GET, HEAD.

Возвращает файл html страницы, которая находится по пути /www/index.html на сервере.

команда серверу: GET /www/index.html HTTP/1.1\r\n\r\n

Ответ: HTTP/1.1 200 OK\r\n\r\n <index.html contents>\r\n\r\n

Понятно дело, что для одного человека задача может быть великоватой. Соответственно декомпозируется на несколько задачек:

1. Поиск и чтение из определенной директории всех html-файлов. Файлы читаются в shared\_memory (или просто общую память). В образ каждого файла в памяти вставляется дополнительный «<p> People visited this page %u», где %u - количество запросов к этой странице. Далее моделируется очередь запросов, которые запрашивают нужный файл. После работы программы, новые файлы сохраняются на диск в другую папку.

2. Реализация получения какого стандартного запроса-ответа от любого числа клиентов: а) тред на каждый сокет + слушающий б) слушающий тред + распределяющий тред на select + пул тредов на обработку запросов

3. Реализация именно HTTP: GET, HEAD для какого-нибудь одного файла. а) через pipe б) через message queue в) через shared memory