Sem1

Самая главная команда в UNIX — man **!!!ЧИТАЕМ MAN!!!**

Разделы:
1. System commands
2. System calls
3. Libc

Если хотим получить из второго раздела, то пишем man 2 write
!Гипертекст! - там есть ссылки
Не боимся гуглить, лучше на английском

Отладка: основной отладчик — GDB

gdb filename, breakpoint, run или gdb -p <process> или gdb filename (a.out) core

ulimit говорит о размере core-файла : ulimit -c unlimited

gdb удобен невсегда, тк невсегда есть доступ к исходникам. Для наших программ он в целом не супер нужен, лучше юзать программу strace a.out — печатаются все системные вызовы
-f — флаг, отслеживающий все процессы, порожденные текущим.
-p <pid> - тоже полезно

Флаги компиляции:
-g — чтобы пользоваться отладчиком
-Wall — компилятор намекнет, где может быть ошибка

Задача — написать программу, выводящую числа от 1 до n

Sem2

fork() - системный вызов, порождающий процессы
Код возврата разный для родительского(pid child) и дочернего процесса(0).
По умолчанию тип pid — int, но правильно юзать pid_t
команда ps — список процессов текущей сессии
с флагом -axf — всех процессов в системе
getpid() - системный вызов для получения идентификатора текущего процесса
getppid() - получить идентификатор родительского процесса

Как мы стартуем программу?
Exec() - берет образ исполняемого файла и выкидывает, остаются только файловые дескрипторы и разделяемые ресурсы, и запускает заданный файл
Исключение: файлы, открытые с флагом O_CLOEXEC — они закрываются
Есть разные типы ексеков:

execl	
execlp	отличие от v: аргументы командной строки как аргуметы
execle	функции, иначе помещабтся в массив)
execv	execv(const char * filename, char * const argv[]
execve	(argv[0] = filename))
execvp	

Потоки (треды) — независимые потоки исполнения в рамках одного процесса
Процессы — изолированы друг от друга
int pthread_create(phtread_t * id, const pthread_attr_t * attr, void * (*run)(void * arg), void * arg) — создать поток (0 или меньше 0 — возвращает по ошибке) | указатель на функцию (стартует поток и сразу же эта функция, в качестве аргумента подсунут последний параметр), процесс живет до сих пор, пока родительский поток не прочитает закрытие дочернего

p = wait(int * status) (в p — pid, в статус — код возврата). Любой вызов вейт ждет любо порожденный процесс и уничтожает его.
Для тредов — pthread_join(id); (id получается ещё из создания)

Задача: три раминочных программки
1) Порождение n дочерних процессов, каждый из которых должен вывести свой id, порядковый номер, где порожден, и айди родителя
2) Запускалка произвольной программы с произвольным кол-вом аргументов (x ls -als → x -a -l -s)
3) Породить n потоков, которые инкрементируют одну и ту же переменную (к примеру каждый по 10к раз)

Sem3

Штатный способ для взаимодействия процессов — использование примитивов:
Pipe — космическая хрень(штруба) с двумя концами — на них файловые дескрипторы (в них можно писать и их можно читать). Данные хранятся внутри ядра. Это некоторый буфер в ядре относительно небольшого размера. Системный вызов:
pipe(int fd[2]); fd — пара дескрипторов. В man все написано. Fd1 для записи, 0 для чтения. Существует до сих пор, пока есть хотя бы один файловый дескриптор, который на него ссылается.
Что будет при форк? Дескрипторов станет 4, по паре в каждом процессе. Кол-во ссылок на пайп увеличилось. Если пайп открыт наполовину, то: при закрытом конце на запись чтение вернет накопленные данные и чтение потом будет возвращать 0, больше никогда 0 возвращать чтение из пайпа не будет. Если нет конца не чтение, то произойдет много нехорошего: вернется ошибка и текущий процесс убивается. Сделано так для того, чтобы работала полезная структура:
(основано на системе — каждая утилита делает одну вещь, но делает ее хорошо — основа юникс)
Утилита find . -name *.[ch] — найти все файлы с расширением с или h и распечатать на экран. Она умеет хорошо делать одно — искать файлы по написанным правилам.
Поток из стандартного вывода можно завернуть в поток стандартного ввода другой утилиты специальным значком: | (это по сути и есть пайп)
И далее другая утилита: xargs grep myfunc — закидывает все из стандартного ввода в другую утилиту (myfunc) как аргументы. Если нет конца на чтение, то этот конец(xargs) помирает, и зачем тогда делать лишнюю работу (вообще начинать фаинд).

Задание: программа, порождающая дочерний процесс, который читает файл, а родительский процесс печатает этот файл.
Пайп хорош быстротой, но плох тем, что связать можно только родителя и ребенка. Поэтому придумали создавать файл-метку для открытия пайпа, если в двух процесса открыть этот файл, то образуется труба между этими двумя процессами. – FIF0
Системный вызов mkfif0(name, 0644);

open(O_RDONLY); - блокируется, пока нет конца на запись, т.е. Открытия фифо в режиме O_WRONLY.
O_RDWR — в линуксе сразу открывает дескриптор, который можно использовать и для чтения, и для записи. Если есть флаг O_NONBLOCK, то O_WRONLY, если нет второго конца, возвращает ошибку, а рдонли все равно открывает дескриптор. Почему так — надо подумать на досуге.
В результате есть некоторые сложности — если будет несколько процессов через один один пайп, а это не оч.

Задача на 1 плюс: написать два процесса(две или одну программу, не отец и ребёнок), т.е. в двух разных терминалах работающие. Первый читает файл, а второй печатает этот файл. Если запускать еще читателя и писателя одновременно с работой первой пары, то файл не должен быть испорчен в процессе передачи. Можно пользоваться только файловым API(опен, рид, врайт), а файловыми локами пользоваться нельзя. Допустим в одной ветке один раз системный вызов слип, но лучше обойтись без него.

Sem4

Работая с ядром, мы получаем всякие дескрипторы — чиселки. Они иденфицируют всякие объекты и тд. Дескрипторы наследуются через форк. При нем создается новая таблица файловых дескрипторов, которая имеет то же содержимое, что и родители. При ексек все дескрипторы отсаются, кроме тех, кто имели флак о_клоексек, или фснтл.
Что такое файловый дескриптор? Он содержит внутри себя текущую позицию, с которой происходит чтение и ссылку на следующий объект. Следствие из такой структуры хранения — у одного файла может быть несколько имен — прямая ссылка или жесткая связь. Все объекты удаляются, когда удаляется ссылка на них. Кол-во имен для файла хранится в индексном узле.
Если мы хотим получить к нему доступ, есть системный вызов: fstat и stat. Если посмотреть на стрейс, то увидим, что прежде, чем открыть файл, на него зовут стат.
Какие бывают файлы? Обычные и директории. В директорию пользователь писать не может. Директория изменяется через системные вызовы ренейм, анлинм и креейт. Важное свойство — переименование атомарно.
Есть вызовы рид райт опен клоз, есть вызов стат. Есть фцнтл. Анлинк — удаление имени. Опендир, риддир. У фснтл есть сложные флажки — блокировка файлового дескриптора, к примеру. В юниксе локи адвайзери — лок не блокирует опен. Если процесс умер, блокировки на фд отпускаются. Есть OFD блокировки.
Какие бывают имена файлов? Полные и относительные. Относительно текущей директории процесса. Операция открытия файла безумно дорогая. Чтобы упростить операцию открытия используется свойства директории. К файлу приписывается его текущая директория. Setcwd getcwd. Полное имя всегда с /. (/хоум/нейм/1.с). ./1.с и 1.с одно и то же для опен, но для эксек нет. У него другие правила поиска файла. Если ./a.аут, то из текущей директории, а если a.аут, то ищет по всем директориям.
Если дисков несколько, то пофиг. В юниксе одна файловая система. / - корневая директория операционной системы и самого первого диска. Еще один диск присоединятся к текущему дереву файлов системный вызовом mount. Есть обратный вызов — umount.
Сплайс без юзерспейса перекладывает из дескриптора в дескриптор в кернелспейсе.

Sem5

Кроме пайпа есть очередь сообщений. СистемФайв предложил некоторое количество для межпроцессорной комуникации. Один из них — очередь сообщений

Создаем объект — очередь, на выходе имеем дескриптор этой очереди. Есть системный вызов: int msgget(key_t key, int flags). Флаги схожи семантикой с флагами опен. IPC_CREAT, IPC_EXCL, и права доступа 0666.
Как проверяется, имеем ли мы право открыть файл? Есть пользователь, от лица которого исполняется процесс. Идентификатор получаем через getuid() \\ geteuid(). Есть группа пользователь, от которых исполняется процесс. Групп может быть больше одной. И есть все остальные.
Права на файл выглядят так: первые три бита — для пользователя, вторые — для группы, третья группа — все остальные права. Проверяется, совпадают ли идентификаторы пользователя и процесса. Далее, если не совпали, проверяются группы, далее общие. Последовательной проверки нет, только по отдельности на каждые три бита смотрим. Три бита — read ,write, execute. Ексекют для директории — имеем право искать в этой директории.
Какие есть еще битики в правах? Важный битик — suid. Его наличие означает, что если мы запускаем этот исполняемый файл, то юсерайди будет не наш, а тот, которому этот файл принадлежит. (euid поставляется по suid)
Для того, чтобы поменять пароль в системе — надо записать данные в файл etc shedule. Для обычного пользователя этот файл закрыт. Мы запускаем программу пасвд, которая запускается от рута, и может тогда поменять пароль. При попытке в записи файл с сменившемся сюидом ничего не выйдет(к примеру, если скопируем эту программу пасвд).
Руту на все эти права плевать, для него ничего не проверяется.
Откуда брать ключ для мсггет? Можно создать уникальную очередь — IPC_PRIVATE. Дескриптор унаследуем через форк. А если в разных программа использовать один ключ? Получим один и тот же объект, но нифига не удобно.
Обычно этот ключ генерируют библиотечного вызова ftok(filename, id <0,256>) Айди, чтобы от одного файла генерить несколько разных ключей. Как работает фток — вопрос филосовский kekw.
Вот получили мы айди очереди - msgid. Нужно научиться уничтожать очередь. Для этого есть вызов msgctl. Он умеет больше, так-то. Передаем msgctl(msgid, IPC_RMID, NULL) и удалим.
Ipcs, ipcrm — две программы, для того, чтобы полистить все объекты из командной строки.

Чтобы послать сообщение в очередь — системный вызов msgsnd(int id, comst void * message, sie_t size, int flags) IPC_NOWAIT — флажок, чтоб не ждать.
Получить сообщение — msgrcv(int id, void * buff, size_t len, long msgtype, int flags). Для ознакомления с флагами надо читать man.
С типом все интереснее — посылаемое сообщение должно иметь определенную структуру. Первый лонг по адресу message рассматривается как тип сообщения. Обязательно целое. Размер это не просто размер, а без первого лонга. Ресив хочет получить из очереди первое сообщение с указанным типом, остальные остаются в очереди просто. Если тип — отрицательное число, то вытаскивается первое сообщение с типом, не превосходящим модуль указанного числа. Если указать в ресиве тип ноль, то просто получим сообщение из головы.
После завершения программы очереди не удаляется, их надо удалять САМОМУ.

Программа на 0,5 плюса: породить n детей, а потом каждый ребенок должен напечатать свой порядковый номер. Числа должны вывестись по порядку!!!! Все дочерние процессы — дети корневого. И сначала надо породить все процессы, потом печатать. Печать должна быть в ребенке. Синхронизироваться на очередях сообщений.

Sem6

Семафор. int semget(key_t, int n, int flags); Одного семафора мало, поэтому создаем пачку из n семафоров. Создается неинициализированным, но в большинстве систем инициализация нулём. Semctl().
Для взаимодействия с семафорами нужно использовать системный вызов semop(int fd, struct sembuf *, int n) — дескриптор, массив операция и колчество операций.
Struct sembuf
{
sem_num — порядковый номер семафора из набора
sem_op — чиселко, добавляемое к переменной семафора
sem_flag — IPC_NOWAIT — не ждать на семафорной операции(можно указывать для любой(или для одной??) операции), SEM_UNDO — В момент выполнения операции программа налетела на сегфолт или что-то ещё, то есть умерла так или иначе, поэтому у нас есть возможность в случае смерти процесса откатить какие-то операции. Если умрем, то откатим эту операцию. Но после выполнения другой операции надо обозначить, что больше откатывать не надо — тем же флагом. Есть ещё другие флаги.
}
Как работает внутри? Есть семафор. Для него есть массив(хеш-таблица). Когда выполняем семанду для конкретного процесса, работающего с семафором, анду каунт увеличивается на единицу(вернее на операцию), а после другой операции уменьшается на единицу(операцию). И потом из семафора вычитается анду каунт. То есть возможность вернуть семафор в изначальное значение.

Что такое семафоры - https://habr.com/ru/post/261273/ http://www.codenet.ru/progr/cpp/7/3.php

Разделяемая память. int shmget(key_t key, size_t size, int flags)
shmctl();
Получить адрес по системному вызову void * shmat(int shmid, NULL, 0); (считает количество сегментов\аттачей)
В 99% случаем нулл и 0. shmdt(void * addr) - автоматически при смерти процесса. Разблокирует адресс для работы.

Задача на 0,5 плюса: через семафоры и разделяемую память решить первую задачу.

Sem7

Сигналы — программные прерывания. Сигнал доходит до процесса, признак того, что такой сигнал пришел, проставляется. Тот, кто послал сигнал, возвращается из системного вызова. Обработка сигнала произойдет тогда, когда планировщик передаст процессу, куда послан сигнал, управление.
Если послать два сигнала очень быстро, обработчик сигнала на стороне, куда послали, может вызваться только один раз.
Не все сигналы так себя ведут. Только сигнали из стандарта POSIX так себя ведут. Есть ещё real-time сигналы, но их обсуждать не будем.
Как послать сигнал? При помощи системного вызова kill(pid_t pid, int signum);
pid — штука хитрая. Если посылать сигнал кому-то, указам положительное число, то пошлем сигналу с таким pid. Если число отрицательное, то посылаем сигнал группе процессов с таким номером. Если kill(-1, 9), то это все процессы, кроме init. По сути прав это делать у нас нет, но если мы будет root`ом…. ПОЧТИ ВСЕ ПОГИБНЕТ
Системный вызов kill — единственный легальный способ проверить наличие кнотркетного процесса. kill(pid, NULL);

Как обрабатывать сигнал? Системный вызов signal(). Использовать не рекомендуется — поведение на UNIX System5 и на UNIX BSD оно разное.
Пользоваться нужно int sigaction(int signum, coonst struct sigaction * s, struct sigaction * old);
struct sigaction
{
	.handler = h;
	.sa_mask = ….
};
У нас есть ситуация, когд мы не хотим, чтоы сигналы к нам приходили. К примеру, когда работаем с переменными, которые затрагивает обработчик сигнала. Так как сигнал может случиться в любое произвольное время, когда вызван планировщик(то есть, в любой момент).
Запретить вызов обработчика сигналов можно системным вызовом int sigprocmask(int how, const sigset_t * set, sigset_t * old);
Масочка — абрактный тип, для работы с которым есть 5 функций:
sigemptyset(&set);
sigfillset();
sigaddset(); - добавить в масочку один конкретный сигнал
sigdelset(); - убрать конкретный сигнал
sigismember(); - проверить, есть ли сигнал в маске
Что за параметр how? С каждым процессом связано две маски. Маска пришедших и маска заблокированных сигналов. Добавить или убрать что-то в маску уже самого процесса, определяет параметр how:
SIG_BLOCK
SIG_UNBLOCK
SIG_SETMASK — поставить маску как есть

Заблокировать можно и все сигналы, но на некоторые блокировка просто не действует.
Можно подождать приход сигнала. Системный вызов pause() ожидает, пока сигнал придёт. Но пользовать им не надо, это порождает непредвиденные последствия. К примеру, были запрещены сигналы, мы их разрешили, потом зовем пауз, в итоге получим хрен знает что, потому что сработает обработчик, ведь синал лежал в маске пришедших сигналов, а пауза встанет навсегда.
Рекомендуется использовать sigsuspend(&set); Он берет, и ставит маску, указанную в качестве параметра, ждет прихода сигналов, и возвращает ту маску, которая была до его вызова. По сути этот вызов атомарен sigprocmask(SIG_SET, &set, &old); pause(); sigrocmask(SIG_SET, &old, ..). Но это исполняется в ядре и атомарно.
В sigaction обработка сигнала может иметь два предопрееленных значения. SIG_IGN — игнорировать, SIF_DFL — поставить по умолчанию обработчик.
При fork() обработчик сигналов наследуется, как было.
При exec() то что было в SIG_IGN остается там, а то, что было в каких-то функциях менятеся на SIF_DFL.

Задача на 1 плюс: породить дочерний процесс, который читает файл, а родительский процесс его печатает. Усложняющее вводное: использовать функции ввода-вывода(и любые системные вызовы и libc) в обработчике сигналов нельзя, пользоваться передачей байтика через расширенную посылку сигнала нельзя, родитель должен быть готов, что ребенок умрет, ребенок толже должен быть готов обработать смерть родителя.

Sem8

Те сигналы, которые мы посылаем, когда удобно нам. Но есть ситуация, когда сигналы посылаются самой ОСью. Какие сигналы перехватить нельзя? Сигкилл, сигстоп, сигкоунт. Чтобы процесс выкинулся и финально уничтожился для зомби процесса не обязательно, ОСь сделает это сама.
1 — sighup - процессы объединены в группы, а группы в сеансы(sessions). Сеанс порождается вместе с запуском терминала. Закроем терминал, попробуем посмотреть его через другой терминал и получим этот сигнал.
2 — sigint - ctrl+c
4 — sigill — при некорректной инструкции в коде
6 — sigabrt во время sigassert или abort
sigfpe — если случилось исключение на сопрцессоре (к примеру деление на 0)
sigkill
sigterm — джентельменская версия килла — закрыть все аккуратно, дописать текущие транзакции и тд
sigegv — segmentation fault
sigpipe — при записывании в закрытый на чтение пайп
sigalrm — генерируется alarm(), но через таймаут, аказанные в качестве параметра системного вызова
sigio — синхронный ввод-вывод, но этот сигнал не очень удобен

Sem9

Административная часть: очередь в чатике, туда ссылку на гитхаб свой.
Операции ввода-вывода: read и write. Они блокруют процесс, пока не появятся данные в канале, если блокирующий дескриптор. Если неблокирующий, то будем ловить EAGAIN при отсутвии\переполнении буфера.
Как найти момент, когда можем корректно запросить рид и райт?
Решить эту проблему правильного ожидания события на пачке файловых дескрипторов можно системными вызовами: call и select. В общем плане делают одно и то же.
Select();
fd_set* - внутри него поместить признак того, каким файловым дескриптором мы интересуемся.
Вместо таймаута можно указать NULL — ждать вечно.
Первый параметр — оптимизация. Можно указывать максимальный файловый дескриптор + 1.
Селект возвращает кол-во файловых дескрипторов, по которым возможна операция.
После надо проверить, какие операции возможны. После успеха селекта, рид тоже должен вернуть успех. Успех — неотрицательный еррор код.
Помимо полного успеха возможен частичный успех — можем прочитать и записать не все данные, которые иначально хотели. Отсюда разные тонкости.
К примеру: есть пайп, пишем туда мегабайт, а запишет только 128 кб, тк то размер буфера на пайп в ядре.

Последняя задача: программа порождает n детей. Родитель читает из ребенка номер n и передает данные в ребенка с номером n+1. В каждый дочерний процесс два пайпа из родителя, кроме самого первого — он читает из файла. Ребенок читает в блокирующем режиме, а родитель в неблокирующем режиме перекладывает данные по цепочке. Идея — эмоляция однопоточного прокси. В конечном итоге все данные выводит родитель. Усложняющее вводное: размер буфера не степени двойки, а степени тройки, плюс буфера всех детей разные. Буффер уменьшается с каждым ребенком: 3^(n-i) * 1000, где i — номер текущего соединения. Держать се данные

одного соединения в одной структуре данных — это банально удобнее. Привязка структуры идет именно к соединению, то есть к двум детям, а не иначе.