За да координират работата си, процесите комуникират както помежду си, така и с ядрото. Linux поддържа известен брой механизми на комуникация между процесите (IPC - Interprocess Communication Mechanisms). Два от тях са сигналите и конвейрите, но Linux поддържа и механизмите от System V IPC, наречени по името на Unix™ изданието, в което са използвани за първи път.

1. Сигнали

Сигналите са един от най-старите методи на междупроцесова комуникация, използвани в системите Unix™ . Използват се за сигнализиране на асинхронни явления в един или повече процеси. Сигнал може да бъде генериран от клавишно прекъсване или при грешка, като например даден процес опитва достъп до несъществуващо място във виртуалната си памет. Освен това сигналите се използват от шеловете за означаване на команди за управление на работата към техните подчинени процеси.

Съществува комплект от определени сигнали, които ядрото може да генерира или които могат да бъдат генерирани от другите процеси в системата, при условие, че те имат съответните привилегии. Списъкът на комплекта от сигнали на системата може да бъде изведен чрез командата kill (kill -l), в Intel Linux box, което дава:

1) SIGHUP
5) SIGTRAP
9) SIGKILL
13) SIGPIPE
18) SIGCONT
22) SIGTTOU
26) SIGVTALRM

2) SIGINT
6) SIGIOT
10) SIGUSR1
14) SIGALRM
19) SIGSTOP
23) SIGURG
27) SIGPROF

3) SIGQUIT
7) SIGBUS

11) SIGSEGV
15) SIGTERM
20) SIGTSTP
24) SIGXCPU
28) SIGWINCH

4) SIGILL
8) SIGFPE
12) SIGUSR2
17) SIGCHLD
21) SIGTTIN
25) SIGXFSZ
29) SIGIO

30) SIGPWR

При Alpha AXP Linux box номерата са различни. Процесите могат да избират игнориране на повечето генерирани сигнали, с две важни изключения: не може да бъде игнориран сигналът SIGSTOP, при който процесът преустановява изпълнението си, и сигналът SIGKILL, при който се излиза от процеса. Въпреки това, даден процес може да избира как "желае" да борави с различните сигнали. Процесите могат да блокират сигналите и, ако не ги блокират, те могат да изберат да боравят с тях сами или да позволят на ядрото да борави с тях. Ако със сигналите борави ядрото, то ще изпълни обичайното действие, изисквано за дадения сигнал. Например, обичайното действие, когато даден процес получи сигнал SIGFPE (floating point exception -) е да запише състоянието на системата в момента и да излезе. Сигналите нямат вътрешно присъщи приоритети. Ако

за даден процес се генерират едновременно два сигнала, то те могат да бъдат представени и обработени към процеса в какъвто и да е ред. Освен това няма механизъм за обработка на множество сигнали от един и същ вид. Няма начин, по който процесът да каже дали е получил 1 или 42 SIGCONT сигнала.

Linux възприема и използва сигналите, като използва информация за процеса, съхранявана в task_struct. Броят на поддържаните сигнали е ограничен от вида на процесора. 32-битови процесори могат да имат 32 сигнала, докато 64-битови процесори като Alpha AXP могат да имат до 64 сигнала. Висящите в момента сигнали се намират в полето signal, с маска от блокирани сигнали, съдържащи се в blocked. С изключение на SIGSTOP и SIGKILL, всички сигнали могат да бъдат блокирани. Ако се генерира блокиран сигнал, той остава висящ, докато бъде деблокиран. Освен това Linux поддържа информация как всеки процес борави с всеки възможен сигнал, като това се съхранява в полето на структурите на данните sigaction, посочени от task_struct за всеки процес. Измежду другите неща, там има информация или за адреса на инструкциите за боравене със сигнала, или флаг, посочващ на Linux, че процесът иска или да игнорира сигнала или да го предостави на ядрото, за да го обработи вместо него. Процесът модифицира обичайния начин на работа със сигнала, като създава системни обаждания, като тези обаждания променят sigaction за съответния сигнал, както и за маската blocked.

Не всеки процес може да изпраща сигнали до всеки друг процес, могат го ядрото и привилегированите потребители. Нормално процесите могат да изпращат сигнали само до процеси със същия uid и gid или до процеси от същата група1. Сигналите се генерират чрез настройка на съответния бит в полето task_struct's signal. Ако процесът не е блокирал сигнала или изчаква, но е прекъсваем (в състояние Interruptible), тогава той се активира чрез промяна на състоянието му на Running - работещ - и като се провери дали той е в списъка на действащите сигнали. По този начин системата го разглежда като кандидат за стартиране за следващия път. Ако има нужда от обичайна обработка на сигнала, Linux може да я оптимизира. Например, ако сигналът е SIGWINCH (променен фокус на X window) и се използва обичайният начин на обработка, не се прави нищо.

Сигналите не се представят на процесите непосредствено след генерирането си, те трябва да изчакат докато процесът се стартира отново. Всеки път, когато даден процес излиза от системно обаждане, неговите полета signal и blocked се проверяват и ако има някакви деблокирани сигнали, те могат да бъдат доставени. Това може да изглежда много ненадежден метод, но всеки процес в системата извършва системно обаждане, например през цялото време изписва знак в терминала. Процесите могат да избират да изчакат сигнала, ако "желаят", като остават в състояние Interruptible - прекъсваем, докато получат сигнал. Кодът на обработка на сигналите на Linux се вижда в структурата sigaction за всеки от деблокираните в момента сигнали.

Ако програмата за боравене със сигнала (signal handler) е настроена за обичайно действие, сигналът ще бъде обработен от ядрото. Обичайната програма за обработка на сигнала SIGSTOP ще промени текущото състояние на процеса на Stopped - спрян и ще стартира scheduler-а за избиране на нов процес, който да бъде стартиран. Обичайното действие при сигнал SIGFPE ще създаде core dump - файл, в който записва състоянието в момента, и ще приключи процеса. Като алтернатива, процесът може да си е определил своя собствена routine - програма за обработка на сигнала. Това е

комплект от инструкции, които се извикват всеки път, когато се генерира сигнал, като структурата sigaction съхранява адреса на тези инструкции. Ядрото може да извика инструкциите за обработка на сигнала на процеса, като това, което става, е специфично за всеки процесор, но всички CPU трябва да могат да се справят с факта, че текущият процес работи в режим на ядрото и се връща към процеса, който е извикал ядрото или системните инструкции в потребителски режим. Проблемът се решава чрез обработка на стека и регистрите на процеса. Програмният брояч на процесите е настроен на адреса на съответните им инструкции, като параметрите на тези инструкции се добавят към рамката на обаждането или преминават в регистри. Когато процесът възстанови работа, изглежда като че ли инструкциите за обработка на сигнала са нормално извикани.

Тъй като Linux е съвместим с POSIX, процесите могат да определят кои сигнали се блокират, когато се извика определена инструкция за обработка на сигнала. Това означава смяна на blocked маската по време на повикването на програмата за обработка на сигнала на процесите. Маската blocked трябва да се върне към първоначалната й стойност, когато приключат инструкциите за обработка на сигнала. Освен това Linux добавя повикване към основните инструкции, които ще възстановят първоначалната blocked маска в библиотеката на сигналирания процес. Linux оптимизира и системата в случай, че трябва да се повикат няколко комплекта инструкции, като ги подрежда така, че всеки път, когато се излезе от определена инструкция, се повиква следващата, докато се повика основната.

2. Конвейри

Всички основни шелове на Linux позволяват пренасочване. Например:

```
$ ls | pr | lpr
```

отвежда изходния сигнал от командата ls, като извежда списък на файловете в директорията в стандартния вход на командата pr, която ги странира. Накрая стандартният изход от командата pr се отвежда към стандартния вход на командата lpr , която отпечатва резултатите чрез основния принтер. Така, че конвейрите са еднопосочни байтови потоци, свързващи стандартния изход от един процес към стандартния вход на друг процес. Никой от процесите "не знае", че е пренасочен и се държи както в нормална ситуация. Шелът е този, който настройва тези временни конвейри между процесите.

..\..\images/solution/ipc_pipes.gif

..\..\images/solution/ipc_pipes.gif

Фигура 1: Конвейри

В Linux конвейрът използва две файлови структури от данни, които посочват същия временен VFS i-node, който от своя страна посочва физическата страница в паметта. Фигура 1 показва, че всяка файлова структура от данни съдържа насочване към различни вектори на инструкции за обработка на файла; единият за писане в конвейра, другият - за четене от него.

С това се скриват вътрешните разлики от основните системни обаждания, които четат и пишат в обикновените файлове. Докато записващите процеси записват в конвейра и байтовете се копират в общоизползваните страници с данни, то четящите процеси копират байтовете от тях. Linux трябва да синхронизира достъпа към конвейра. Той трябва да провери дали четящият и пишещият процес работят, като за да направи, използва locks - заключвания, опашки и сигнали.

Когато пишещият процес иска да пише в конвейра, той използва стандартните функции на библиотеката на писане. Всички те подават описвания на файла, които са знаци в рамките на комплекта от файлови структури от данни на процеса, всеки от тях представлява отворен файл, или, както в този случай, отворен конвейр. Системното обаждане на Linux използва инструкцията за писане, посочена от файловата структура от данни, като описва съответния конвейр. Тази инструкция за писане използва информация, съхраняване в VFS i-node, представляващ конвейр, който управлява искането за писане.

Ако няма достатъчно място за записване на всички байтове в конвейра и докато конвейрът не е заключен от четеца си, Linux заключва записващия процес и копира байтовете, които трябва да бъдат записани, от мястото на адреса на процеса в общата страница на данните. Ако конвейрът е заключен от четеца или няма достатъчно място за данните, то текущият процес се включва в опашката на конвейрния inode и програмата за управляване на процесите се информира, че може да стартира друг процес. Той е непрекъсваем, така, че може да получава сигнали и ще бъде "събуден" от четеца, когато се появи достатъчно място за записване на данните или когато конвейрът се отключи. След записване на данните VFS inode на конвейра се отключва и всички четци, включени в опашката за изчакване на този i-node също ще бъдат "събудени".

Четенето на данни от конвейра е процес, много сходен с този на писане в него.

На процесите е позволено да изпълняват неблокиращи четения (в зависимост от режима, в който те са отворили файл или конвейр), като, в този случай, ако няма данни за четене или конвейрът е заключен, се връща грешка. Алтернативата е да се изчака на опашката на конвейра до приключването на процеса на писане. Когато и двата процеса приключат работата си по конвейра, неговият inode заедно с общата страница на данните се отстраняват.

Освен това Linux поддържа и named - именувани конвейри, познати като FIFO ("първи влязъл - първи излязъл"), тъй като конвейрите работят на принципа на изпращане по реда на пристигане. Първите данни, записани в конвейра, са първите, прочетени от него. За разлика от конвейрите, FIFOs не са временни обекти, те са цялости във файловата система и могат да бъдат създадени посредством командата mkfifo. Процесите имат свободата да използват FIFO докато имат права за достъп до него.

Начинът, по който се отварят FIFO, е малко по-различен от този при конвейрите. Конвейрът (неговите две файлови структури от данни, неговият VFS i-node и поделената страница за писане) се отварят наведнъж, докато FIFO вече съществува и се отваря и затваря от потребителите. Linux трябва да поддържа четящите операции, отварящи FIFO, преди пишещите да го отворят, както и четящите да четат преди която и да е пишеща операция да е записала в него. Освен това, FIFO се поддържат по почти точно същия начин както конвейрите и използват същите структури от данни и операции.

3. Сокети (sockets)

3.1 Механизми на междупроцесова комуникация System V

Linux поддържа три типа механизми на междупроцесова комуникация, които се появяват за пръв път в системата Unix™ System V (1983). Това са опашки за съобщения, семафори и shared - поделена памет. Всичките тези System V механизми на междупроцесова комуникация използват единни методи на общо установяване. Процесите могат да достигнат до тези ресурси само чрез предаване на единен идентификатор към ядрото чрез системни обаждания. Достъпът до тези System V обекти на междупроцесова комуникация се проверява чрез разрешения за достъп, така, както се проверяват разрешенията за достъп до файлове. Правата за достъп до такъв обект се настройват от специална създаваща го програма посредством системни обаждания. Справочният идентификатор на обекта се използва от всеки механизъм като индект в таблица на ресурсите. Той не е прост индекс, но изисква известна намеса, за да генерира индекса.

Всички структури данни на Linux, представляващи обекти на междупроцесова комуникация от System V в системата, включват структура ipc_perm, която съдържа идентификаторите на собственика и потребителя на програмата за създаване на процеса и групата. Режимът за достъп за този обект (собственик, група или друго) и ключът за IPC обекта. Ключът се използва като път за откриване на справочния идентификатор на System V IPC обекта. Поддържат се два комплекта ключове: общ и личен. Ако ключът е общ, то всеки процес в системата, предмет на проверка на правата, може да намери справочния идентификатор за System V IPC обекта. За обектите System V IPC не може да се получи информация чрез ключ, а само чрез техния справочен идентификатор.

3.2 Опашки за съобщения

Опашките за съобщения позволяват на един или повече процеси да пишат съобщения, които да бъдат прочетени от един или повече четящи процеси. Linux поддържа списък от опашки за съобщения, векторът msgque, всеки елемент от който сочи към структура данни msqid_ds, описваща напълно опашката за съобщения. При създаването на опашка за съобщения, системната памет открива нова структура данни msqid_ds и я въвежда във вектора.

..\..\images/solution/ipc_msg.gif
..\..\images/solution/ipc_msg.gif
Фигура 2: Опашки за съобщения в System V IPC

Всяка структура данни msqid_ds съдържа структура данни ipc_perm и указатели към съобщенията, въведени в тази опашка. Освен това Linux указва и времената на промяна на опашката, като например последното време на записване в опашката и т.н. Структурата msqid_ds съдържа и две опашки за изчакване - едната за записване, а другата за четене на съобщенията.

Всеки път, когато даден процес се опита да запише съобщение в опашката за запис, идентификаторите на потребителя и групата му се сравняват с режима, зададен в структурата ipc_perm на опашката. Ако процесът може да направи запис в опашката, то съобщението може да бъде копирано от адреса на процеса в структура от данни msg, като се поставя в края на опашката. Всяко съобщение се асоциира със специалния тип приложение, съгласувано между работещите съвместно процеси. Може обаче да няма място за съобщението, тъй като Linux ограничава броя и дължината на съобщенията, които могат да бъдат записани. В този случай процесът се добавя в опашката за изчакване за записване, като се повиква програмата за управляване на процесите, за да избере нов процес за стартиране. Тя ще бъде "събудена" когато от опашката на съобщенията се прочетат едно или повече съобщения.

Четенето от опашката е сходен процес. Тук отново се проверяват правата на процесите за достъп до опашката за писане. Четящият процес може да избере да получи първото съобщение от опашката независимо от типа му, или да избере съобщения с определен тип. Ако няма съобщение, което да отговаря на този критерий, четящият процес се добавя към опашката за изчакване на четене и се стартира програмата за управление на процесите. Когато в опашката се запише ново съобщение, тази програма се "събужда" и стартира отново.

3.3 Семафори

В своята най-проста форма семафорът е място в паметта, чиято стойност може да бъде тествана и настроена от повече от един процес. Операцията по тестване и настройване е, доколкото важи за всички процеси, непрекъсваема или атомна, стартирана веднъж, нищо не може да я спре. Резултатът от нея е добавяне на текущата стойност на семафора и стойността на настройване, която може да бъде положителна или отрицателна. В зависимост от резултата от операцията по тестване и настройване, на даден процес може да се наложи да "заспи", докато стойността на семафора бъде променена от друг процес. Семафорите могат да се използват за въвеждане на критични области, области на критичен код, който може да се изпълни само от един процес в дадения момент.

Да речем, че имате много работещи съвместно процеси, четящи и записващи в един

файл с данни. Искате достъпът до файла да бъде стриктно координиран. Можете да използвате семафор с първоначална стойност 1 и, около операционния код на файла, да включите две семафорни операции, първата - да тества и намали стойността на семафора, а втората - да я тества и увеличи. Първият процес за достъп до файла ще опита да намали стойността на семафора и ако успее, тази стойност сега става 0. След това този процес ще продължи нататък и ще използва данните на файла, но ако друг процес, който иска да го използва, се опита да намали стойността на семафора, той няма да успее и резултатът ще бъде -1. Този процес ще бъде прекъснат докато първият процес приключи работата си с файла. Когато приключи, той ще увеличи стойността на семафора, като я направи отново 1. Сега чакащият процес ще бъде събуден и този път опитът му да увеличи семафора ще бъде успешен.

..\..\images/solution/ipc_sem.gif
..\..\images/solution/ipc_sem.gif
Фигура 3: Семафори System V IPC

Всеки от семафорните обекти в System V IPC описват семафорен ред, като Linux използва структура от данни semid_ds, за да представи това. Всички структури данни semid_ds в системата са посочени от semary, вектор от указатели. Във всеки семафорен ред има sem_nsems, като всеки от тях е описан от структура данни sem и указан от sem_base. Всички процеси, на които е позволено да манипулират семафорния ред на семафорен обект System V IPC, могат да правят системни повиквания, които изпълняват операциите върху тях. Системното повикване може да определи много операции, като всяка операция се описва от три входни сигнала: индекс на семафора, операционна стойност и комплект флагове.

Семафорният индекс е индекс в семафорния ред, а операционната стойност е числена стойност, която се добавя към текущата стойност на семафора. Най-напред Linux проверява дали изобщо или че не всички операции ще успеят. Дадена операция ще бъде успешна ако операционната стойност, добавена към текущата стойност на семафора е по-голяма от 0 или ако и операционната стойност, и текущата стойност на семафора са 0. Ако някоя от семафорните операции има изгледи да се провали, Linux може да прекъсне процеса но само ако флаговете на операцията не са поискали системното повикване да бъде неблокиращо.

Ако процесът трябва да се прекъсне, то Linux трябва да запише състоянието на семафорните операции, които трябва да се изпълнят и да постави текущия процес в опашката за изчакване. Той прави това чрез изграждане на структура от данни sem_queue в стека и я попълва (чрез указателите sem_pending и sem_pending_last). Текущият процес се поставя в опашката за изчакване в структурата от данни sem_queue (sleeper) и се повиква scheduler-a, за да избере да стартира друг процес.

Ако всички семафорни операции ще бъдат успешни и текущият процес не се нуждае от прекъсване, Linux отива нататък и прилага операциите към съответните членове на семафорния ред. Сега Linux трябва да провери дали някой от чакащите, прекъснатите

процеси може да приложи своите семафорни операции. Той "поглежда" всеки член от опашката на висящите операции (sem_pending) последователно, проверявайки, за да види дали семафорните операции този път ще успеят. Ако те ще успеят, той премахва структурата от данни sem_queue от списъка на висящите операции и прилага семафорните операции в семафорния ред. Той събужда спящия процес, като прави възможно рестартирането му следващия път, когато се стартира scheduler. Linux продължава да следи списъка на висящите от старта докато се убеди, че няма повече семафорни операции за прилагане и няма повече процеси за събуждане.

При семафорите съществува един проблем, deadlocks. Те се получават, когато даден процес е променил стойността на семафора при влизане в критична област, но след това не е успял да излезе от нея, защото се е "забил" или е бил "убит". Срещу това Linux поддържа списъци от настройки за семафорните редове. Идеята е, че когато се приложат тези настройки, семафорите се привеждат в състоянието, в което са били преди прилагането на комплекта от семафорни операции на процеса. Тези настройки се съхраняват в структурите от данни sem_undo, в опашката на структурите от данни semid_ds и task_struct за процесите, използващи тези семафорни редове.

Всяка отделна семафорна операция може да поиска да се поддържа настройка. Linux ще поддържа най-много една структура от данни sem_undo на процес за всеки семафорен ред. Ако исканият процес няма такава, тя се създава, когато се наложи. Новата структура от данни sem_undo се поставя в опашката както в структурата от данни task_struct на процеса, така и в semid_ds на семафорния ред. Тъй като операциите се прилагат към семафорите в семафорния ред, отрицателната стойност на операционната стойност се прибавя към записа в семафора в реда на настройка на структурата от данни sem_undo на процеса. Така, че, ако операционната стойност е 2, в записа на настройката за този семафор се прибавя -2.

Когато процесите се изтриват при изхода си, Linux работи посредством техния комплект структури от данни sem_undo, като прилага настройките към семафорните редове. Ако даден семафорен комплект се изтрие, то структурите от данни sem_undo остават в опашката в task_struct на процеса, но идентификаторът на семафорния ред става невалиден. В този случай изчистващият код на семафора просто отстранява структурата от данни sem_undo.

3.4 Shared Memory - Поделена памет

Поделената памет позволява един или повече процеси да комуникират посредством памет, която се появява във всички места на техния виртуален адрес. Страниците на виртуалната памет се указват от записи в таблицата на страниците във всяка от таблиците на поделените процеси. Тя не трябва да бъде на същия адрес във всяка виртуална памет на процесите. Както при всички обекти в System V IPC, достъпът до областите на поделената памет се контролира от проверка на ключовете и правата на достъп. След като паметта се подели, вече няма проверки как процесите я използват. Те трябва да разчитат на други механизми, например System V семафори, за да синхронизират достъпа до паметта.

..\..\images/solution/ipc_shm.gif
..\..\images/solution/ipc_shm.gif
Фигура 4: Поделена памет в System V IPC

Всяка област на новосъздадена поделена памет се представя от структура от данни shmid_ds. Те се съхраняват във вектора shm_segs.
Структурата от данни shmid_ds описва колко е голяма областта на поделената памет, колко процеса я използват, както и информация как поделената памет се разпределя в местата на техните адреси. Именно създателят на поделената памет контролира разрешенията за достъп до тази памет и дали ключът е общ или личен. Ако няма достатъчно права за достъп, той може да заключи поделената памет във физическата памет.
Всеки процес, който желае да подели паметта, трябва да се прикачи към тази виртуална памет посредством системно обаждане. Това създава нова структура от данни vm_area_struct, описваща поделената памет за този процес. Процесът може да избере къде да отиде поделената памет в мястото на виртуалния му адрес или може да остави Linux да избере достатъчно голямо свободно място. Новата структура vm_area_struct се поставя в списъка на vm_area_struct, посочена от shmid_ds. Указателите на vm_next_shared и vm_prev_shared се използват, за да ги свържат. В действителност виртуалната памет се създава не при "закачането", а когато първият процес направи опит за достъп до нея.
Първият път, когато даден процес отвори една от страниците на поделената виртуална памет, става page fault - грешка на страницата. Когато Linux поправи тази грешка, той намира структурата от данни vm_area_struct, която я описва. Тя съдържа указатели за инструкциите за обработка на този тип поделена виртуална памет. Поддържащият код на грешката на страницата на поделената памет поглежда в списъка на записите на таблицата на страниците за този shmid_ds, за да види дали съществува такъв за тази страница от поделената виртуална памет. Ако няма такъв, той ще създаде физическа страница и ще създаде запис в таблицата на страниците за нея. Освен, че отива в таблиците на страницата на текущия процес, този запис се записва и в shmid_ds. Това означава, че когато следващият процес, който направи опит за достъп до паметта, създаде page fault, поддържащият код на грешката на поделената памет също ще използва тази новосъздадена физическа страница. Така, че, първият процес, отворил страница от поделената памет, я създава, а по-нататъшният достъп на другите процеси добавя тази страница в местата на техния виртуален адрес.
Когато процесите повече не желаят да използват виртуалната памет, те се отделят от нея. Докато други процеси все още използват паметта, това отделяне засяга само текущия процес. Неговият vm_area_struct се отстранява от структурата от данни shmid_ds и се делокализира. Таблиците на страницата на текущия процес се актуализират, за да освободят пространството от виртуалната памет, използвано за общ достъп. Когато и последният процес, използващ поделената памет, се отдели от нея, нейните страници във физическата памет се освобождават, както и структурата от данни shmid_ds за тази поделена памет.
По-нататък стават усложнения, когато поделената виртуална памет не е заключена

във физическата памет. В този случай в периоди на голямо използване на паметта страниците от поделената памет могат да бъдат прехвърлени към системния swap диск.