УДК 004.42(07)
ББК 32.973.01я7
Х 93
Христинець Н.А.
Луцький національний технічний університет

ВИКОРИСТАННЯ ПРОГРАМНИХ ТА ІМЕНОВАНИХ КАНАЛІВ ЯК ЗАСОБІВ ВЗАЄМОДІЇ МІЖ ПРОЦЕСАМИ В UNIX/LINUX.

Важливими задачами ОС UNIX є задачі ефективності обміну даними та виключення конфліктів при взаємодії між процесами. Для вирішення завдання їх взаємодії в операційній системі існує набір спеціальних засобів, одними із яких є програмні та іменовані канали. Зазначено їх ефективність та односпрямованість використання для передачі та запису даних поміж родинними процесами.

Ключові слова: *процеси в UNIX/LINUX, канали, іменовані канали, FIFO та LIFO.*

Класична схема мультипрограмування, яка підтримується в операційній системі UNIX, дає можливість паралельного (або псевдопаралельного в разі наявності лише одного апаратного процесора) виконання декількох призначених для користувача програм. Кожному такому виконанню відповідає процес операційної системи. Кожен процес виконується у власній віртуальній пам'яті, і, тим самим, процеси захищені один від одного, тобто один процес не в змозі неконтрольованим чином прочитати чи записати що-небудь з пам'яті іншого процесу.

Повна ж ізоляція процесів в операційній системі безглузда, оскільки їм приватно необхідно обмінюватися даними в процесі роботи. Операційною системою допускаються контрольовані взаємодії процесів, у тому числі за рахунок можливості розділення сегменту віртуальної пам'яті декількох процесів. Так як в ОС UNIX процеси виконуються у власних адресних просторах і є ізольовані один від одного, то можливості впливу процесів один на одного, що є надто важливим у багатозадачних ОС, зведено до мінімуму. Власне концепція UNIX ґрунтується на модульному принципі й передбачає взаємодію поміж процесами.

При такій взаємодії вважливою необхідністю є розв'язування ряду завдань, таких як:

1. Задача передачі даних, обсяг яких може коливатись від десятків байт до кількох мегабайт.
2. Задача спільного використання даних. При цьому процеси можуть використовувати спільно одну копію даних, і зміни, внесені одним процесом, відразу будуть помітні для іншого. В цьому випадку кількість взаємодіючих процесів перевищує два. З метою збереження цілісності ресурсів процесам може стати необхідний протокол взаємодії для збереження цілісності даних та виключення конфліктів при доступі до них.
3. Задача використання повідомлень про певну подію, наприклад для синхронізації кількох процесів. Розв'язується ця задача засобами ОС, тому що власне самі процеси у рамках багатозадачної ОС діяли б неефективно і навіть небезпечно.

До засобів міжпроцесної взаємодії, притаманних усім версіям UNIX, належать:

- сигнали;
- канали;
- іменовані канали FIFO та LIFO;
- повідомлення та їх черги;
- семафори;
- розподілювана пам'ять;
- сокети.

Взаємодія поміж процесами має бути уніфікованою, незалежно від того, виконуються ці процеси на одному чи то на різних комп'ютерах у мережі. Комунікаційні характеристики взаємодії мають бути доступними для процесів в уніфікованій формі, тобто додаток може вимагати конкретного виду зв'язку, який ґрунтуватиметься на віртуальному каналі, дейтаграмах тощо. Будь-який спосіб взаємодії має забезпечувати такі сервіси:

- впорядковану доставку даних;
- відсутність дублювання даних;
- надійну доставку даних;

- підтримку передавання екстрених даних;
- попереднє встановлення зв'язку.

При організації взаємодії процесів каналами, їх може бути зорганізовано при роботі у командному рядку shell контейнером:

cat myfile |wc.

Стандартне виведення програми cat(1), яка виводить вміст файла myfile, передається на стандартний ввід програми wc(1), яка, у свою чергу, підраховує кількість рядків, слів та символів. Як наслідок на екран монітора буде виведено дані:

12 45 260,

що означає кількість рядків, слів та символів у файлі myfile. Отже, два процеси обмінялися даними. Програмний канал забезпечує односпрямоване передавання даних поміж двома процесами.

Для створення каналу використовується системний виклик pipe(2):

int pipe(int *filedes),

який повертає два файлових дескриптори: filedes[0] — для запису до каналу та filedes[1] — для читання з каналу. Якщо один процес записує дані до filedes[0], інший може отримати ці дані з filedes[1]. При створенні процесу атрибути батьківського процесу наслідуються дочірнім процесом, у тому числі файлові дескриптори. Доступ до дескрипторів filedes каналу може отримати сам процес, що викликається pipe(2), та його дочірні процеси. Отже, канали може бути використано для передавання даних лише поміж родинними процесами, а для міжпроцесної взаємодії поміж незалежними процесами не використовуються.

У наведеному прикладі обидва процеси, cat(1) та wc(1), створюються процесом shell і тому є родинними, хоча на перший погляд здаються незалежними.

FIFO є надто схожі на звичайні програмні канали, вони є односпрямованим засобом передавання даних, причому читання даних відбувається у тому ж самому порядку, що й записування. На відміну від програмних каналів, вони мають імена, які дозволяють незалежним програмам отримати доступ до цих об'єктів. Важливо зазначити, що FIFO є засобом версії UNIX SystemV і для використання в інших версіях потребують встановлення додаткових бібліотек системних викликів.

FIFO представляє собою окремий тип файла у файловій системі UNIX (результат виконання команди ls –l покаже символ p у першій позиції). Для створення FIFO використовується системний виклик mkfifo(2):

int mkfifo(char *pathname, int mode),

де: pathname — ім'я файла у файловій системі (ім'я FIFO),

mode — прапорець володіння, прав доступу тощо.

FIFO може бути створено і з командою рядка shell:

$ mkfifo name p

Після створення, FIFO може бути відкрито для запису та читання, причому запис до нього та читання можуть відбуватися в різних незалежних процесах. Канали FIFO та звичайні канали працюють за такими правилами:

1 При читанні меншої кількості байтів, аніж перебуває в каналі або FIFO, повертається потрібна кількість байтів, а решта зберігається для подальших читань.

2 При читанні більшої кількості байтів, аніж перебуває в каналі чи FIFO, повертається доступна кількість байтів. Процес, що читає з каналу, має опрацьовувати ситуацію, коли прочитано менше, ніж замовлено.

3 Якщо канал є порожній і жоден процес не відкрив його на записування, системний виклик read(2) буде заблоковано до появи там відповідних даних (якщо лише для каналу або FIFO не встановлено прапорець відсутності блокування O_NDELAY).

4 Запис кількості байтів меншої ємності каналу або FIFO гарантовано атомарно. Це означає, що в разі, коли кілька процесів водночас записують дані до каналу, порції даних від цих процесів не перемішуються.

5 У перебігу запису більшої кількості байтів, ніж це дозволяє канал або FIFO, виклик write(2) блокується до звільнення потрібного місця, але атомарність цієї операції не гарантується. Якщо процес намагається записати дані до каналу, що не відкривався жодним процесом, процесові генерується сигнал SIGPIPE, а виклик write(2) повертає 0 із встановленням помилки (errno = EPIPE). Якщо процес не встановив опрацювання сигналу SIGPIPE, опрацювання відбувається за замовчуванням і процес завершується.

У каналі може перебувати лише певна кількість байтів, перш ніж наступний виклик write(2) буде заблоковано. Мінімальний розмір каналу, визначений POSIX, дорівнює 512 байтів. Виклик write(2) виконується неподільними порціями, і запис виконується ядром за одну неперервну операцію. Батьківський процес виконується у нескінченному циклі, а дочірній надсилає повідомлення батьківському, опитуючи канал і перевіряючи, чи надійшли дані.

Буферизація даних у каналі стандартно реалізується шляхом відокремлення дискового простору у структурі файлової системи. Отже, запис та читання пов'язані з дисковим введенням/виведенням даних, що зменшує його продуктивність. Сучасні серверні ОС забезпечують роботу каналів через спеціальну файлову систему HPPS (High Performance Pipe System). З її допомогою дані буферизуються в оперативній пам'яті, що прискорює запис/читання.

Умовну схему цього каналу подано на рис.1 а). Усі заявки надходять у кінець черги. Першими обслуговуються заявки, які перебувають на початку черги.

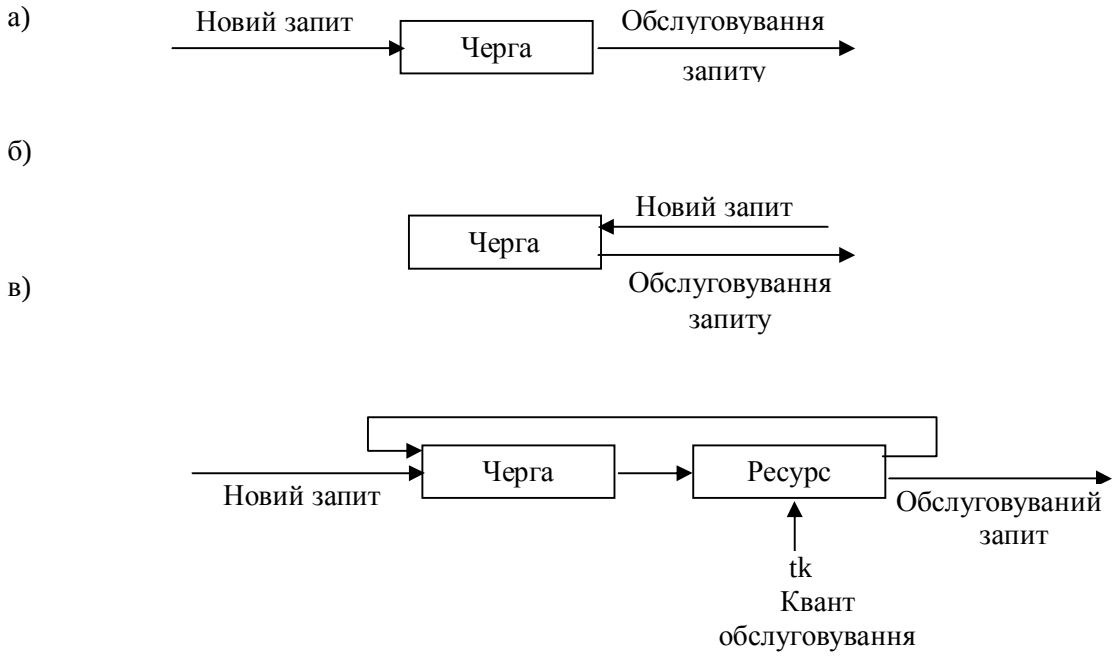

Рисунок 1 – Схеми каналу обслуговування процесів:
а) – FIFO;
б) – LIFO;
в) – круговий циклічний алгоритм

LIFO – теж простий і широко розповсюджений на практиці канал, який є підґрунтям побудови стекової пам'яті. Умовне зображення каналу подано на рис.1 б).

Обидва канали (і FIFO, і LIFO), є широко використовувані стосовно "довгих" та "коротких" запитів. Середній час очікування в черзі не залежить від характеристик процесів-користувачів і в середньому є однаковий для всіх.

Круговий циклічний алгоритм є підґрунтям каналу FIFO. Час обслуговування кожного процесу обмежено й визначається квантом надаваного часу, – tk . Якщо запит на використання ресурсу з початку черги обслуговується до кінця за час tk (наприклад, програму процесу за час tk повністю виконано на процесорі), то він вилучається з черги. Якщо цей запит не встигає обслуговуватися до кінця, то його обслуговування переривається, та він переходить у кінець черги.

Схему каналу кругового циклічного алгоритму приведено на рис.1, в). Канал широко використовується на практиці, зокрема при реалізації режиму розподілу часу. Хоча в цьому каналі немає явно окреслених пріоритетів, автоматично відбувається дискримінація "довгих" та "коротких" запитів. За найбільш сприятливих умов "короткі" запити відокремлюються від процесів, для яких потрібен менший час використовування ресурсів. "Короткі" запити мають менший середній час очікування в системі, аніж "довгі" запити. Квант наданого на користування

ресурсом часу обирається як компромісне значення: чим менше значення tk, тим більше буде сприяння "коротким" запитам, але більшим буде час, який витрачається на перерозподіл ресурсів поміж процесами через зростання частости переривань, і тим більш несприятливі умови складаються для довгих запитів.

На практиці використовуються різні модифікації цього алгоритму.

Прикладом функцій при роботі з каналами є:

1 Функція pipe(2), що створює односпрямований симплексний канал для анонімного обміну даними поміж двома спорідненими процесами, позаяк лише вони можуть отримати доступ до одного й того самого каналу. Після завершення роботи канал знищується.

Функція має вигляд

#include <unistd.h>

int pipe(int filedes[2])

Функція повертає два файлових дескриптори fd у масиві fd[0] та fd[1]; filedes[0] дозволяє читання даних з каналу, а filedes[1] використовується для записування даних у канал.

2 Функція fcntl(2) забезпечує керування файловими операціями у вже відкритих файлах, заданих дескрипторами файлу — filedes.

#include < fcntl.h> — заголовний файл, у якому визначено цілі константи: O_RDONLY (для решти лише читати), O_WRONLY (для решти лише писати), O_RDWR (для решти читати та писати), які встановлюють або знімають блокування на файли чи їхні частки.

int fcntl (int filedes, int cmd, …) — функція fcntl(2) виконує дію, зазначену в cmd, з файлом, а третій аргумент залежить від конкретної дії:

а) F_SETLK — встановлює блокування запису файла; структура flock описує блокування, а покажчик на неї передається у третьому аргументі; за неможливості блокування fcntl(2) повертається з помилкою EACCESS або EAGAIN.

б) F_SETLKW — аналогічна до попередньої, але аргумент використовується за неможливості блокування, якщо запис вже заблоковано; процес переходить до стану сну, очікуючи на зняття блокування (W_WAIT, очікувати).

в) F_SETFL — визначає режим запису даних наприкінці файла.

3 Функція printf(2) —формує запис до форматизованого стандартного виводу.

4 Системний виклик unlink(2) вилучає файл, наприклад unlink ("/tmp/usedfile"); виклик повертає 0 у разі успішного завершення та -1 — у разі помилки.

5 Системний виклик read(2) копіює довільну кількість символів чи байтів з файла до буфера (в ASCII кодах).

6 Системний виклик open(2) відкриває файл для читання, запису або створює порожній файл.

7 Системний виклик close(2) закриває файл, повертає його в разі успішного завершення та -1 — у разі помилки:

#include <unistd.h>

int close (int filedes)

filedes = open("file", O_RDONLY);

close (filedes);

8 Системний виклик write(2) — копіює дані з буфера програми, що трактується як масив, до зовнішнього файла. Він, як і read(2), має три аргументи: дескриптор файла filedes, покажчик на записувані дані buffer та n — додатне число, котре визначає кількість записуваних байтів.

#include <unistd.h>

ssize_t write(int filedes, const void *buffer, size_t n).

1. В. Столлингс. Операционные системы. – М.: Изд. дом «Вильямс», 2002.

2. Гордеев А.В., Молчанов А.Ю. Системное программное обеспечение. – СПб.: Питер, 2001.

3. В. Г. Олифер, Н. А. Олифер «Сетевые операционные системы» учебник для вузов, издательство: Питер, 2008 г.

4. Эндрю Таненбаум: «Современные операционные системы» 2-е изд. издательство: Питер, 2005 г.