Московский Государственный Технический Университет имени Н. Э. Баумана

Пугачев Е.К.

Исследование способов организации и взаимодействия процессов.

Методические указания по выполнению лабораторной работы

по дисциплине "Операционные системы".

Москва 2013

Введение

Одной из задач любой операционной системы (ОС) является поддержание надежного и эффективного механизма управления процессами и ресурсами вычислительной системы. Функции управления системными ресурсами присущи любой развитой операционной системе и включают в себя управление оперативной памятью, файловой системой, средства создания, синхронизации и диспетчеризации задач (процессов), службу времени, обработку программных прерываний, клиент-серверные взаимодействия и т.д.

Цель работы - исследование способов управления ресурсами и процессами в вычислительной системе.

Продолжительность работы - 8 часов.

ЧАСТЬ 1. ПОРОЖДЕНИЕ НОВОГО ПРОЦЕССА И РАБОТА С НИМ. ЗАПУСК ПРОГРАММЫ В РАМКАХ ПОРОЖДЕННОГО ПРОЦЕССА. СИГНАЛЫ И КАНАЛЫ В ОС UNIX

Цель работы : Изучить программные средства создания процессов, получить навыки управления и синхронизации процессов, а также простейшие способы обмена данными между процессами. Ознакомиться со средствами динамического запуска программ в рамках порожденного процесса, изучить механизм сигналов ОС UNIX, позволяющий процессам реагировать на различные события, и каналы, как одно из средств обмена информацией между процессами.

Краткие теоретические сведения.

Для порождения нового процесса (процесс-потомок) используется системный вызов **fork()**. Формат вызова:

int fork();

Порожденный таким образом процесс представляет собой точную копию своего процесса-предка. Единственное различие между ними заключается в том, что процесс-потомок в качестве возвращаемого значения системного вызова **fork()** получает 0, а процесс-предок - идентификатор процесса-потомка. Кроме того, процесс-потомок наследует и весь контекст программной среды, включая дескрипторы файлов, каналы и т.д. Наличие у процесса идентификатора дает возможность и ОС UNIX, и любому другому пользовательскому процессу получить информацию о функционирующих в данный момент процессах.

Ожидание завершения процесса-потомка родительским процессом выполняется с помощью системного вызова **wait()**:

int wait(int *status);

В результате осуществления процессом системного вызова **wait()** функционирование процесса приостанавливается до момента завершения порожденного им процесса-потомка. По завершении процесса-потомка процесс-предок пробуждается и в качестве возвращаемого значения системного вызова **wait()** получает идентификатор завершившегося процесса-потомка, что позволяет процессу-предку определить, какой из его процессов-потомков завершился (если он имел более одного процесса-потомка). Аргумент системного вызова **wait()** представляет собой указатель на целочисленную переменную **status**, которая после завершения выполнения этого системного вызова будет содержать в старшем байте код завершения процесса-потомка, установленный последним в качестве системного вызова **exit()**, а в младшем - индикатор причины завершения процесса-потомка.

Формат системного вызова **exit()**, предназначенного для завершения функционирования процесса:

int exit(int status);

Аргумент **status** является статусом завершения, который передается отцу процесса, если он выполнял системный вызов **wait()**.

Для получения собственного идентификатора процесса используется системный вызов **getpid()**, а для получения идентификатора процесса-отца - системный вызов **getppid()**:

int getpid(); int getppid();

Вместе с идентификатором процесса каждому процессу в ОС UNIX ставится в соответствие также идентификатор группы процессов. В группу процессов объединяются все процессы, являющиеся процессами-потомками одного и того же процесса. Организация новой группы процессов выполняется системным вызовом **getpgrp()**, а получение собственного идентификатора группы процессов - системным вызовом **getpgrp()**. Их формат:

int setpgrp(); int getpgrp();

С практической точки зрения в большинстве случаев в рамках порожденного процесса загружается для выполнения программа, определенная одним из системных вызовов **execl()**, **execv()**,... Каждый из этих системных вызовов осуществляет смену программы, определяющей функционирование данного процесса:

execl(name, arg0, arg1, ... , argn, 0) char *name, *arg0, *arg1, ... , *argn; execv(name, argv) char *name, *argv[]; execle(name, arg0, arg1, ... , argn, 0, envp) char *name,

***arg0, *arg1, ... , *argn, *envp[]; execve(name, argv, envp) char *name, *arg[], *envp[];**

Сигналы - это программное средство, с помощью которого может быть прервано функционирование процесса в ОС UNIX. Механизм сигналов позволяет процессам реагировать на различные события, которые могут произойти в ходе функционирования процесса внутри него самого или во внешнем мире. Каждому сигналу ставятся в соответствие номер сигнала и строковая константа, используемая для осмысленной идентификации сигнала. Эта взаимосвязь отображена в файле описаний **signal.h**. Для посылки сигнала используется системный вызов, имеющий формат:

void kill(int pid, int sig);

В результате осуществления такого системного вызова сигнал, специфицированный аргументом **sig**, будет послан процессу, который имеет идентификатор **pid**. Если **pid** не превосходит 1, сигнал будет послан целой группе процессов.

Использование системного вызова **signal()** позволяет процессу самостоятельно определить свою реакцию на получение того или иного события (сигнала):

int sig; int (*func)(); int *signal(sig, func) ();

Реакцией процесса, осуществившего системный вызов **signal()** с аргументом **func**, при получении сигнала **sig** будет вызов функции **func()**.

Системный вызов **pause()** позволяет приостановить процесс до тех пор, пока не будет получен какой-либо сигнал:

void pause();

Системный вызов **alarm(n)** обеспечивает посылку процессу сигнала **SIGALARM** через **n** секунд.

В ОС UNIX существует специальный вид взаимодействия между процессами - программный канал. Программный канал создается с помощью системного вызова **pipe()**, формат которого:

int fd[2]; pipe(fd);

Системный вызов **pipe()** возвращает два дескриптора файла: один для записи данных в канал, другой - для чтения. После этого все операции передачи данных выполняются с помощью системных вызовов ввода-вывода **read/write**. При этом система ввода-вывода обеспечивает приостановку процессов, если канал заполнен (при записи) или пуст (при чтении). Таких программных каналов процесс может установить несколько. Отметим, что установление

связи через программный канал опирается на наследование файлов. Взаимодействующие процессы должны быть родственными.

Задание к лабораторной работе

1. Разработать программу, реализующую действия, указанные в задании к лабораторной работе с учётом следующих требований:
   - все действия, относящиеся как к родительскому процессу, так и к порожденным процессам, выполняются в рамках одного исполняемого файла;
   - обмен данными между процессом-отцом и процессом-потомком предлагается выполнить посредством временного файла: процесс-отец после порождения процесса-потомка постоянно опрашивает временный файл, ожидая появления в нем информации от процесса-потомка;
   - если процессов-потомков несколько, и все они подготавливают некоторую информацию для процесса-родителя, каждый из процессов помещает в файл некоторую структурированную запись, при этом в этой структурированной записи содержатся сведения о том, какой процесс посылает запись, и сама подготовленная информация.
2. Модифицировать ранее разработанную программу с учётом следующих требований:
   - действия процесса-потомка реализуются отдельной программой, запускаемой по одному из системных вызовов **execl()**, **execv()** и т.д. из процесса-потомка;
   - процесс-потомок, после порождения, должен начинать и завершать свое функционирование по сигналу, посылаемому процессом-предком (это же относится и к нескольким процессам-потомкам);
   - обмен данными между процессами необходимо осуществить через программный канал.

Варианты заданий

1. Разработать программу, вычисляющую интеграл на отрезке **[A;B]** от функции **exp(x)** методом трапеций, разбивая интервал на **K** равных отрезков. Для нахождения **exp(x)** программа должна породить процесс, вычисляющий её значение путём разложения в ряд по формулам вычислительной математики.
2. Разработать программу, вычисляющую значение **f(x)** как сумму ряда от **k=0** до **k=N** от выражения **x^(2k+1)/(2k+1)!** для значений **x**, равномерно распределённых на интервале **[0;Pi]**, и выводящую полученный результат **f(x)** в файл в двоичном формате. В это время предварительно подготовленный процесс-потомок читает данные из файла,

преобразовывает их в текстовую форму и выводит на экран до тех пор, пока процесс-предок не передаст ему через файл ключевое слово (например, "STOP"), свидетельствующее об окончании процессов.

3.Разработать программу, вычисляющую плотность распределения Пуассона с параметром **lambda** в точке **k** (**k** - целое) по формуле **f(k)=lambda^k*exp(-lambda)/k!**. Для нахождения факториала и **exp(-lambda)** программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

4.Разработать программу, вычисляющую плотность выпуклого распределения в точке **x** по формуле **f(x)=(1-cos(x))/(Pi*x^2)**. Для нахождения **Pi** и **cos(x)** программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

5.Разработать программу, вычисляющую значение плотности лог-нормального распределения в точке **x** (**x>0**) по формуле **f(x)=(1/2)*exp(-(1/2)*ln(x)^2)/(x*sqrt(2*Pi))**. Для нахождения **Pi**, **exp(x)** и **ln(x)** программа должна породить три параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

6.Разработать программу, вычисляющую число размещений **n** элементов по **r** ячейкам **N=n!/n(1)!*n(2)!*...*n(r)!**, удовлетворяющее требованию, что в ячейку с номером **i** попадает ровно **n(i)** элементов (**i=1..r**) и **n(1)+n(2)+...+n(r)=n**. Для вычисления каждого факториала необходимо породить процесс-потомок.

7.Разработать программу, вычисляющую число сочетаний **C(k,n)=n!/(k!*(n-k)!)**. Для вычисления факториалов **n!**, **k!**, **(n-k)!** должны быть порождены три параллельных процесса-потомка.

8.Разработать программу, вычисляющую значение **f(x)** как сумму ряда от **k=1** до **k=N** от выражения **(-1)^(k+1)*x^(2k-1)/(2k-1)!** для значений **x**, равномерно распределённых на интервале **[0;Pi]**, и выводящую полученный результат **f(x)** в файл в двоичном формате. В это время предварительно подготовленный процесс-потомок читает данные из файла, преобразовывает их в текстовую форму и выводит на экран до тех пор, пока процесс-предок не передаст ему через файл ключевое слово (например, "STOP"), свидетельствующее об окончании процессов.

9.Разработать программу, вычисляющую плотность нормального распределения в точке **x** по формуле **f(x)=exp(-x^2/2)/sqrt(2*Pi)**. Для нахождения **Pi** и **exp(-x^2/2)** программа

должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

10. Разработать программу, вычисляющую интеграл в диапазоне от 0 до 1 от подинтегрального выражения **4*dx/(1+x^2)** с помощью последовательности равномерно распределённых на отрезке **[0;1]** случайных чисел, которая генерируется процессом-потомком параллельно. Процесс-потомок должен завершиться после заранее заданного числа генераций **N**.

Контрольные вопросы

1. Каким образом может быть порожден новый процесс? Какова структура нового процесса?
2. Если процесс-предок открывает файл, а затем порождает процесс-потомок, а тот, в свою очередь, изменяет положение указателя чтения-записи файла, то изменится ли положение указателя чтения-записи файла процесса-отца?
3. Что произойдет, если процесс-потомок завершится раньше, чем процесс-предок осуществит системный вызов **wait()**?
4. Могут ли родственные процессы разделять общую память?
5. Каков алгоритм системного вызова **fork()**?
6. Какова структура таблиц открытых файлов, файлов и описателей файлов после создания процесса?
7. Каков алгоритм системного вызова **exit()**?
8. Каков алгоритм системного вызова **wait()**?
9. В чем разница между различными формами системных вызовов типа **exec()**?
10. Для чего используются сигналы в ОС UNIX?
11. Какие виды сигналов существуют в ОС UNIX?
12. Для чего используются каналы?
13. Какие требования предъявляются к процессам, чтобы они могли осуществлять обмен данными посредством каналов?
14. Каков максимальный размер программного канала и почему?

Порядок выполнения работы.

1. Изучить правила использования системных вызовов **fork()**, **wait()**, **exit()**.

2.Ознакомиться с системными вызовами **getpid**(), **getppid**(), **setpgrp()**, **getpgrp**().

3.Изучить средства динамического запуска программ в ОС UNIX (системные вызовы **execl**(), **execv**(),...).

4.Изучить средства работы с сигналами и каналами в ОС UNIX.

5.Ознакомиться с заданием к лабораторной работе.

6.Для указанного варианта составить программу на языке Си, реализующую задание.

7.Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.

8.Защитить лабораторную работу, ответив на контрольные вопросы.

Требования к отчету.

Отчет должен включать :

- название работы и ее цель;
- результаты работы программы в соответствии с заданием;
- код программы с комментариями.

ЧАСТЬ 2. СИНХРОНИЗАЦИЯ ПРОЦЕССОВ

Цель работы: Практическое освоение механизма синхронизации процессов и их взаимодействия посредством программных каналов.

Краткие теоретические сведения.

В предыдущей лабораторной работе были рассмотрены различные программные средства, связанные с созданием и управлением процессами в рамках ОС UNIX. Данная лабораторная работа предполагает комплексное их использование при решении задачи синхронизации процессов и их взаимодействия посредством программных каналов. Кратко перечислим состав системных вызовов, требуемых для выполнения данной лабораторной работы:

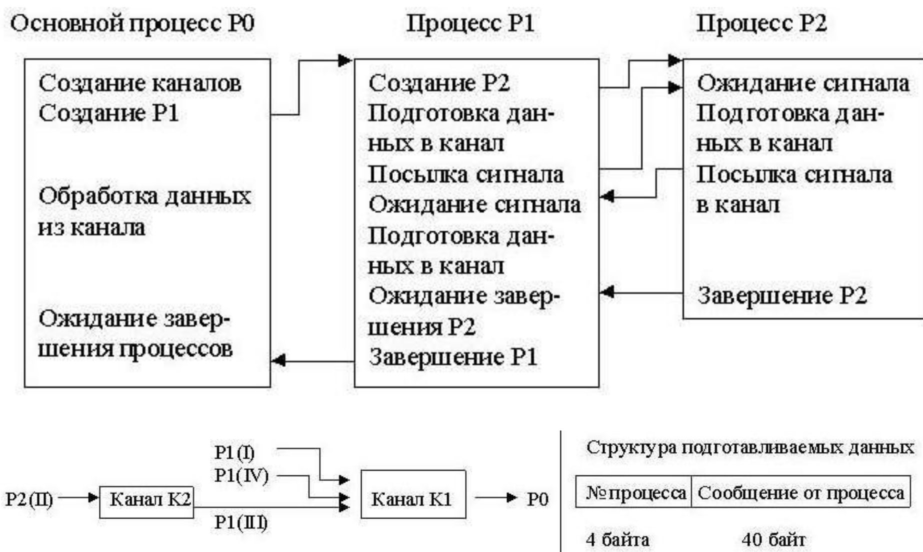
1.Создание, завершение процесса, получение информации о процессе, - **fork**(), **exit**(), **getpid**(), **getppid**().

2.Синхронизация процессов - **signal**(), **kill**(), **sleep**(), **alarm**(), **wait**(), **pause**().

3.Создание информационного канала и работа с ним - **pipe()**, **read()**, **write()**.

Варианты заданий

1.Исходный процесс создает два программных канала К1 и К2 и порождает новый процесс Р1, а тот, в свою очередь, еще один процесс Р2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс Р1 помещает в канал К1, а процесс Р2 в канал К2, откуда они процессом Р1 копируются в канал К1 и дополняются новой порцией данных. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:



Обработка данных основным процессом заключается в чтении информации из программного канала К1 и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

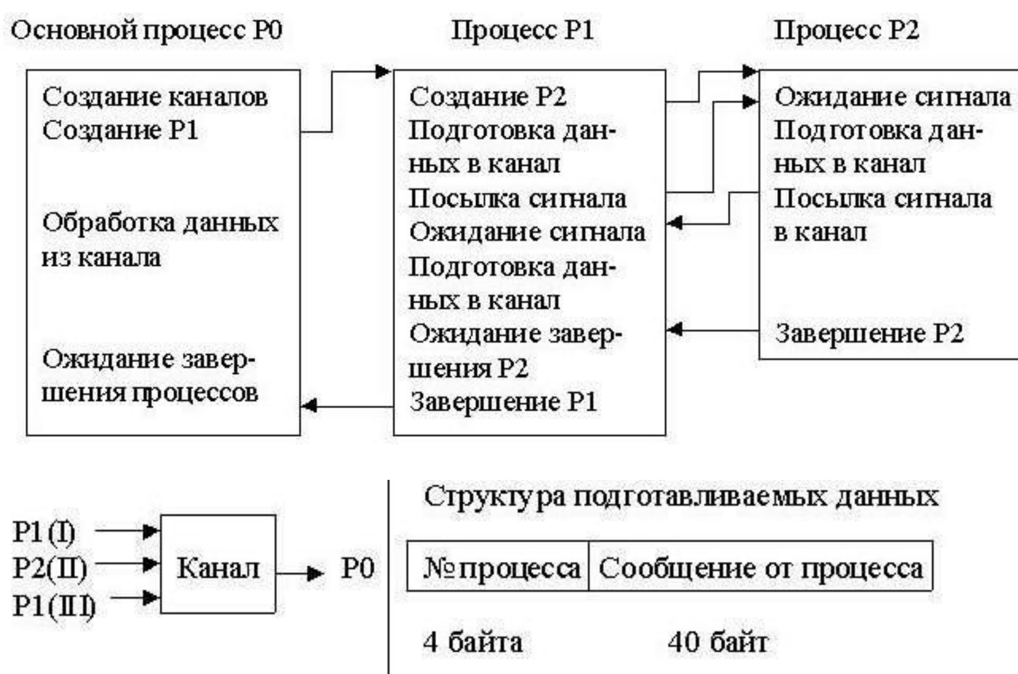
2.Исходный процесс создает программный канал К1 и порождает два процесса Р1 и Р2, каждый из которых готовит данные для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

3.Исходный процесс создает программный информационный канал К1, канал синхронизации К0 и порождает два процесса Р1 и Р2, из которых один (Р1) порождает еще один процесс Р3. Назначение всех трех порожденных процессов - подготовка данных для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал К1 и передаются основному процессу. Кроме того, процесс Р1 через канал синхронизации К0 сообщает процессу Р2 идентификатор процесса Р3 с тем, чтобы процесс Р2 мог послать процессу Р3 сигнал. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

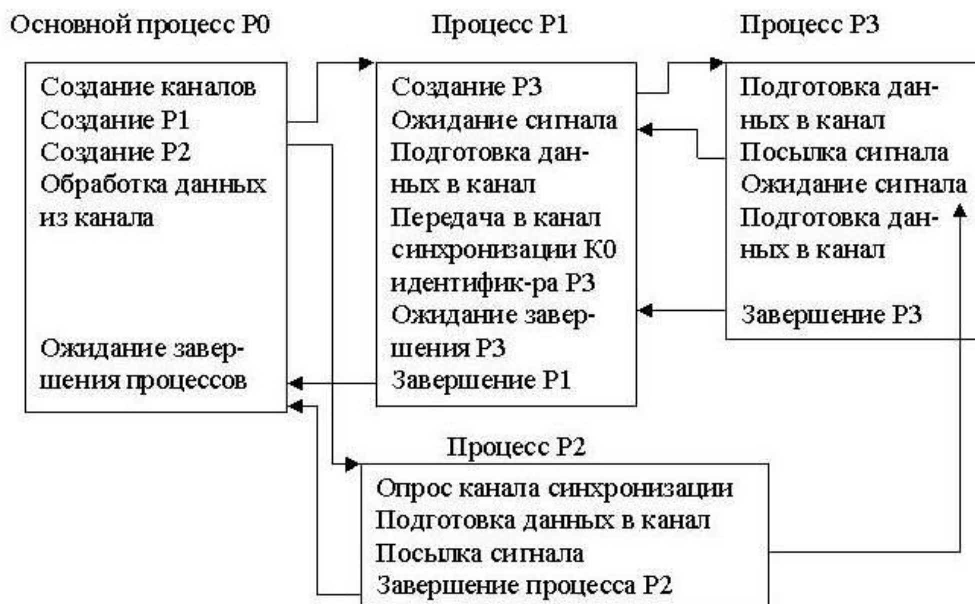
4.Исходный процесс создает программный канал К1 и порождает новый процесс Р1, а тот, в свою очередь, еще один процесс Р2, каждый из которых готовит данные для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

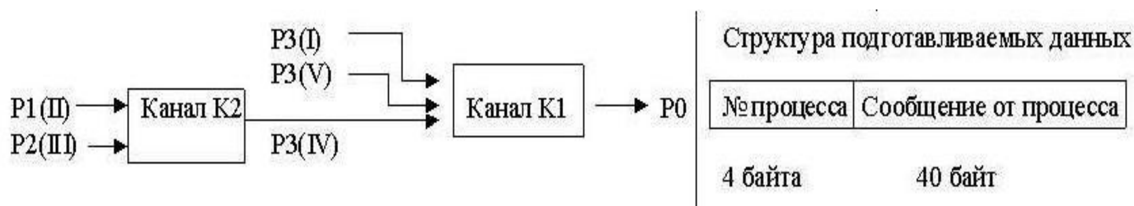


Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

5.Исходный процесс создает два программных информационных канала К1 и К2, канал синхронизации К0 и порождает два процесса Р1 и Р2, из которых один (Р1) порождает еще один процесс Р3. Назначение всех трех порожденных процессов - подготовка данных для обработки их основным процессом. Подготавливаемые данные процесс Р3 помещает в канал К1, а процессы Р1 и Р2 в канал К2, откуда они процессом Р3 копируются в канал К1 и дополняются новой порцией данных. Кроме того, процесс Р1 через канал синхронизации К0 сообщает процессу Р2 идентификатор процесса Р3 с

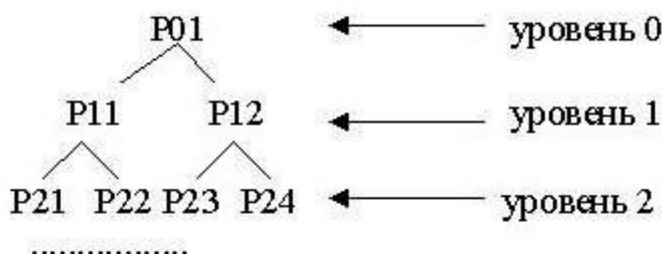
тем, чтобы процесс P2 мог послать процессу P3 сигнал. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:



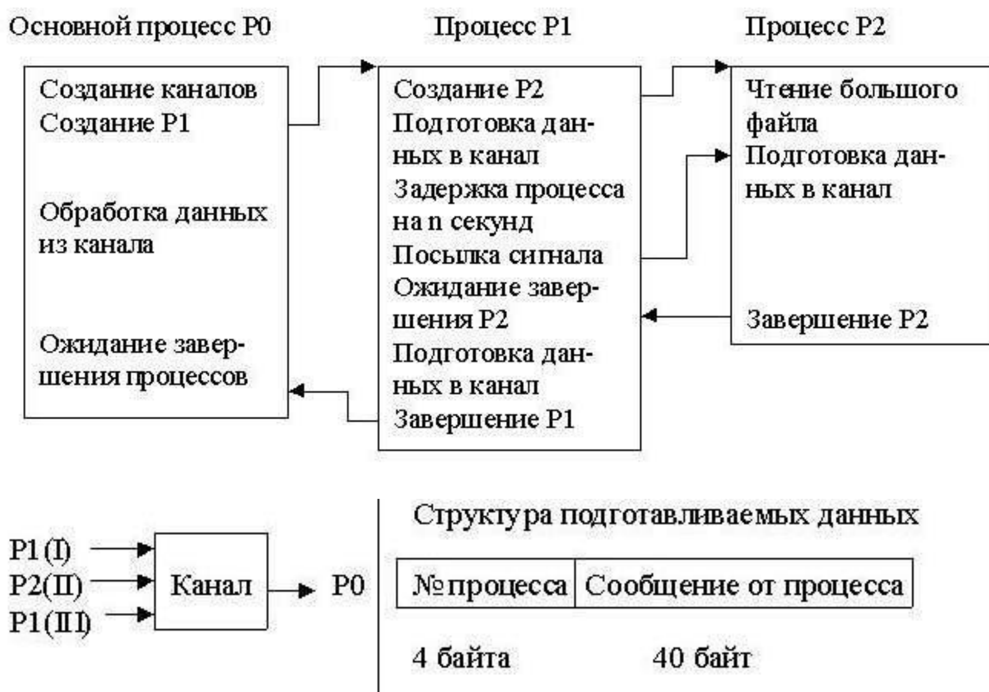


Обработка данных основным процессом заключается в чтении информации из программного канала K1 и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

6.Программа порождает иерархическое дерево процессов. Каждый процесс выводит сообщение о начале выполнения, создает пару процессов, сообщает об этом, ждет завершения порожденных процессов и затем заканчивает работу. Поскольку действия в рамках каждого процесса однотипны, эти действия должны быть оформлены отдельной программой, загружаемой системным вызовом **exec()**. Параметр программы - число уровней (не более 5).



7.Исходный процесс создает программный канал К1 и порождает новый процесс Р1, а тот, в свою очередь, порождает ещё один процесс Р2. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Файл, читаемый процессом Р2, должен быть достаточно велик с тем, чтобы его чтение не завершилось ранее, чем закончится установленная задержка в n секунд. После срабатывания будильника процесс Р1 посылает сигнал процессу Р2, прерывая чтение файла. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

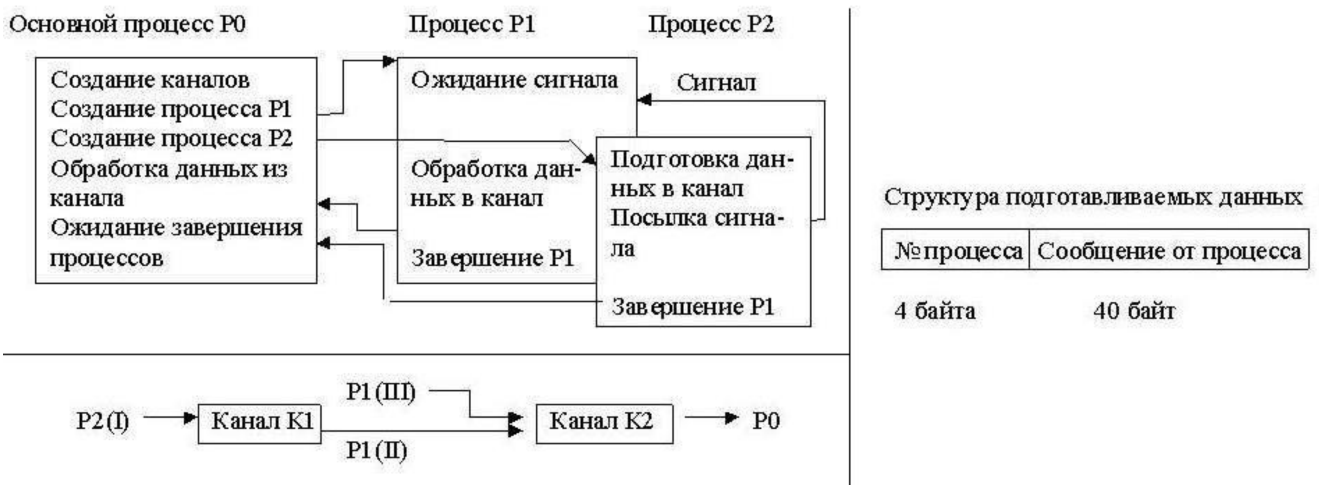


Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

8.Исходный процесс создает программный канал К1 и порождает два процесса Р1 и Р2, каждый из которых готовит данные для обработки их основным процессом. Подготовленные данные последовательно помещаются процессами-сыновьями в программный канал и передаются основному процессу. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных показаны ниже:

Обработка данных основным процессом заключается в чтении информации из программного канала и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

9.Исходный процесс создает два программных канала К1 и К2 и порождает два процесса Р1 и Р2, каждый из которых готовит данные для обработки их основным процессом. Подготавливаемые данные процесс Р2 помещает в канал К1, затем они оттуда читаются процессом Р1, переписываются в канал К2, дополняются своими данными. Схема взаимодействия процессов, порядок передачи данных в канал и структура подготавливаемых данных изображены ниже:



Обработка данных основным процессом заключается в чтении информации из программного канала К2 и печати её. Кроме того, посредством выдачи сообщений необходимо информировать обо всех этапах работы программы (создание процесса, завершение посылки данных в канал и т.д.).

Порядок выполнения работы.

1.Ознакомиться с заданием к лабораторной работе.

2.Выбрать набор системных вызовов, обеспечивающих решение задачи.

3.Для указанного варианта составить программу на языке Си, реализующую требуемые действия.

4.Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.

5.Защитить лабораторную работу, ответив на контрольные вопросы.

Требования к отчету.

Отчет должен включать :

- название работы и ее цель;
- результаты работы программы в соответствии с заданием;
- код программы с комментариями.

ЧАСТЬ3. МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ ПРОГРАММ

Цель работы : Освоение средств IPC. Написание программ, использующих механизм семафоров, очередей сообщений, сегментов разделяемой памяти.

Краткие теоретические сведения.

Механизм IPC (Inter-Process Communication Facilities) включает:

- средства, обеспечивающие возможность синхронизации процессов при доступе к совместно используемым ресурсам (семафоры - semaphores);
- средства, обеспечивающие возможность посылки процессом сообщений другому произвольному процессу (очереди сообщений - message queries);
- средства, обеспечивающие возможность наличия общей для процессов памяти (сегменты разделяемой памяти - shared memory segments).

Наиболее общим понятием IPC является ключ, хранимый в общесистемной таблице и обозначающий объект межпроцессного взаимодействия, доступный нескольким процессам. Обозначаемый ключом объект может быть очередью сообщений, набором семафоров или сегментом разделяемой памяти. Ключ имеет тип **key_t**, состав которого зависит от реализации и определяется в файле **sys/types.h**. Ключ используется для создания объекта межпроцессного взаимодействия или получения доступа к существующему объекту. Обе операции выполняются посредством операции **get**. Результатом операции **get** является его целочисленный идентификатор, который может использоваться в других функциях межпроцессного взаимодействия.

Семафоры

Для работы с семафорами поддерживаются три системных вызова:

- **semget()** для создания и получения доступа к набору семафоров;
- **semop()** для манипулирования значениями семафоров (это тот системный вызов, который позволяет процессам синхронизироваться на основе использования семафоров);
- **semctl()** для выполнения разнообразных управляющих операций над набором семафоров.

Прототипы перечисленных системных вызовов описаны в файлах:

#include <sys/ipc.h> #include <sys/sem.h>

Системный вызов **semget()** имеет следующий синтаксис:

semid = int semget(key_t key, int count, int flag);

Его параметрами являются ключ (**key**) набора семафоров и дополнительные флаги (**flags**), определенные в **<sys/ipc.h>**, число семафоров в наборе семафоров (**count**), обладающих одним и тем же ключом. Системный вызов возвращает идентификатор набора семафоров **semid**. После вызова **semget()** индивидуальный семафор идентифицируется идентификатором набора семафоров и номером семафора в этом наборе.

Флаги системного вызова **semget()** приведены в таблице:

IPC_CREAT	**semget()** создает новый семафор для данного ключа. Если флаг **IPC_CREAT** не задан, а набор семафоров с указанным ключом уже существует, то обращающийся процесс получит идентификатор существующего набора семафоров.
IPC_EXLC	Флаг **IPC_EXLC** вместе с флагом **IPC_CREAT** предназначен для создания (и только для создания) набора семафоров. Если набор семафоров уже существует, **semget()** возвратит -1, а системная переменная **errno** будет содержать значение **EEXIST**.

Младшие 9 бит флага задают права доступа к набору семафоров.

Системный вызов **semctl()** имеет формат:

int semctl (int semid, int sem_num, int command, union semun arg);

где **semid** - это идентификатор набора семафоров, **sem_numb** - номер семафора в группе, **command** - код операции, а **arg** - указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции.

Структура **msg** имеет вид:

union semun { int val; struct semid_ds *buf; unsigned short *array; };

С помощью **semctl()** можно:

- уничтожить набор семафоров или индивидуальный семафор в указанной группе (**IPC_RMID**);
- вернуть значение отдельного семафора (**GETVAL**) или всех семафоров (**GETALL**);
- установить значение отдельного семафора (**SETVAL**) или всех семафоров (**SETALL**);
- вернуть число семафоров в наборе семафоров (**GETPID**).

Основным системным вызовом для манипулирования семафором является:

int semop (int semid, struct sembuf *op_array, count);

где **semid** - это ранее полученный дескриптор группы семафоров, **op_array** - массив структур **sembuf**, определенных в файле **<sys/sem.h>** и содержащих описания операций над семафорами группы, а **count** - размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива **op_array** имеет следующую структуру (структура **sembuf**):

- номер семафора в указанном наборе семафоров;
- операция над семафором;
- флаги.

Если указанные в массиве **op_array** номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов последовательно меняет значение семафора (если это возможно) в соответствии со значением поля "операция". Возможны три случая:

1.Отрицательное значение **sem_op**:

- если значение поля операции **sem_op** отрицательно, и его абсолютное значение меньше или равно значению семафора **semval**, то ядро прибавляет это отрицательное значение к значению семафора;
- если в результате значение семафора стало нулевым, то ядро активизирует все процессы, ожидающие нулевого значения этого семафора;
- если же значение поля операции **sem_op** по абсолютной величине больше семафора **semval**, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора и усыпляет текущий процесс до наступления этого события.

2.Положительное значение **sem_op**. Если значение поля операции **sem_op** положительно, то оно прибавляется к значению семафора **semval**, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии UNIX).

3.Нулевое значение **sem_op**:

- если значение поля операции **sem_op** равно нулю, то если значение семафора **semval** также равно нулю, выбирается следующий элемент массива **op_array**;
- если же значение семафора **semval** отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания.

При использовании флага **IPC_NOWAIT** ядро ОС UNIX не блокирует текущий процесс, а лишь сообщает в ответных параметрах о возникновении ситуации, приведшей бы к блокированию процесса при отсутствии флага **IPC_NOWAIT**.

Очереди сообщений

Для обеспечения возможности обмена сообщениями между процессами механизм очередей поддерживается следующими системными вызовами:

- **msgget()** для образования новой очереди сообщений или получения дескриптора существующей очереди;
- **msgsnd()** для постановки сообщения в указанную очередь сообщений;
- **msgrcv()** для выборки сообщения из очереди сообщений;
- **msgctl()** для выполнения ряда управляющих действий.

Прототипы перечисленных системных вызовов описаны в файлах:

#include <sys/ipc.h> #include <sys/msg.h>

По системному вызову **msgget()** в ответ на ключ (**key**) и набор флагов (полностью аналогичны флагам в системном вызове **semget()**) ядро либо создает новую очередь сообщений и возвращает пользователю идентификатор созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий идентификатор очереди:

int msgqid = msgget(key_t key, int flag);

Для помещения сообщения в очередь служит системный вызов **msgsnd()**:

int msgsnd (int msgqid, void *msg, size_t size, int flag);

где **msg** - это указатель на структуру длиной **size**, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив-сообщение.

Структура **msg** имеет вид:

struct msg { long mtype; /* тип сообщения */ char mtext[SOMEVALUE]; /* текст сообщения (SOMEVALUE - любое) */ };

Параметр **flag** определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти (флаг **IPC_NOWAIT** со значением, рассмотренным выше).

Условиями успешной постановки сообщения в очередь являются:

- наличие прав процесса по записи в данную очередь сообщений;
- непревышение длиной сообщения заданного системой верхнего предела;
- положительное значение указанного в сообщении типа сообщения.

Если же оказывается, что новое сообщение невозможно буферизовать в ядре по причине превышения верхнего предела суммарной длины сообщений, находящихся в данной очереди сообщений (флаг **IPC_NOWAIT** при этом отсутствует), то обратившийся процесс откладывается (усыпляется) до тех пор, пока очередь сообщений не разгрузится процессами, ожидающими получения сообщений.

Для приема сообщения используется системный вызов **msgrcv()**:

int msgrcv (int msgqid, void *msg, size_t size, long msg_type, int flag);

Системный вызов **msgctl()**:

int msgctl (int msgqid, int command, struct msqid_ds *msg_stat);

используется:

- для опроса состояния описателя очереди сообщений (**command** = **IPC_STAT**) и помещения его в структуру **msg_stat**;
- изменения его состояния (**command** = **IPC_SET**), например, изменения прав доступа к очереди;
- для уничтожения указанной очереди сообщений (**command** = **IPC_RMID**).

Работа с разделяемой памятью

Для работы с разделяемой памятью используются системные вызовы:

- **shmget()** создает новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;
- **shmat()** подключает сегмент с указанным описателем к виртуальной памяти обращающегося процесса;
- **shmdt()** отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;
- **shmctl()** служит для управления разнообразными параметрами, связанными с существующим сегментом.

Прототипы перечисленных системных вызовов описаны в файлах:

#include <sys/ipc.h> #include <sys/shm.h>

После того, как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи.

Системный вызов

int shmid = shmget (key_t key, size_t size, int flag);

на основании параметра **size** определяет желаемый размер сегмента в байтах. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является идентификатор существующего сегмента. В противном случае создается новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Создание сегмента не означает немедленного выделения под него основной памяти и это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Флаги **IPC_CREAT** и **IPC_EXCL** аналогичны рассмотренным выше.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову **shmat()**:

void *virtaddr = shmat(int shmid, void *daddr, int flags);

Параметр **shmid** - это ранее полученный идентификатор сегмента, а **daddr** - желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является фактический виртуальный адрес начала сегмента. Если значением **daddr** является **NULL**, ядро выбирает наиболее удобный виртуальный адрес начала сегмента.

Флаги системного вызова **shmat()**:

SHM_RDONLY	ядро подключает участок памяти только для чтения.
SHM_RND	определяет, если возможно, способ обработки ненулевого значения **daddr**.

Для отключения сегмента от виртуальной памяти используется системный вызов **shmdt()**:

int shmdt(*daddr);

где **daddr** - это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова **shmat()**.

Системный вызов **shmctl()**

int shmctl (int shmid, int command, struct shmid_ds *shm_stat);

по синтаксису и назначению полностью аналогичен **msgctl()**.

Варианты заданий

1. Два дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через очередь сообщений родительскому процессу очередные

четыре строки некоторого стихотворения, при этом первый процесс передает нечетные четырехстишья, второй - четные. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы обоих процессов. Решить задачу с использованием аппарата семафоров.

2.Два дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через один и тот же сегмент разделяемой памяти родительскому процессу очередные четыре строки некоторого стихотворения, при этом первый процесс передает нечетные четырехстишья, второй - четные. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы обоих процессов. Решить задачу с использованием аппарата семафоров.

3.Четыре дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через один и тот же сегмент разделяемой памяти родительскому процессу очередную строку некоторого стихотворения, при этом первый процесс передает 1-ю, 5-ю, 9-ю и т.д. строки, второй - 2-ю, 6-ю, 10-ю и т.д. строки, третий - 3-ю, 7-ю, 11-ю и т.д. строки, четвертый - 4-ю, 8-ю, 12-ю и т.д. строки. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы всех процессов. Решить задачу с использованием аппарата семафоров.

4.Программа моделирует работу примитивной СУБД, хранящей единственную таблицу в оперативной памяти. Выполняя некоторые циклы работ, K порожденных процессов посредством очереди сообщений передают родительскому процессу номер строки, которую нужно удалить из таблицы. Родительский процесс выполняет указанную операцию и возвращает содержимое удалённой строки.

5.Программа моделирует работу примитивной СУБД, хранящей единственную таблицу в оперативной памяти. Выполняя некоторые циклы работ, K порожденных процессов посредством очереди сообщений передают родительскому процессу номер строки и её содержимое, на которое нужно изменить хранящиеся в ней данные. Родительский процесс выполняет указанную операцию и возвращает старое содержимое изменённой строки.

6.Программа моделирует работу примитивной СУБД, хранящей единственную таблицу в оперативной памяти. Выполняя некоторые циклы работ, K порожденных процессов

посредством очереди сообщений передают родительскому процессу содержимое строки, которую нужно добавить в таблицу. Родительский процесс проверяет, нет ли в таблице такой строки, и, если нет, добавляет строку и возвращает количество хранящихся в таблице строк.

7. Четыре дочерних процесса выполняют некоторые циклы работ, передавая после окончания очередного цикла через очередь сообщений родительскому процессу очередную строку некоторого стихотворения, при этом первый процесс передает 1-ю, 5-ю, 9-ю и т.д. строки, второй - 2-ю, 6-ю, 10-ю и т.д. строки, третий - 3-ю, 7-ю, 11-ю и т.д. строки, четвертый - 4-ю, 8-ю, 12-ю и т.д. строки. Циклы работ процессов не сбалансированы по времени. Родительский процесс компонует из передаваемых фрагментов законченное стихотворение и выводит его по завершении работы всех процессов. Решить задачу с использованием аппарата семафоров.
8. Родительский процесс помещает в сегмент разделяемой памяти имена программ из предыдущих лабораторных работ, которые могут быть запущены. Выполняя некоторые циклы работ, порожденные процессы случайным образом выбирают имена программ из таблицы сегмента разделяемой памяти, запускают эти программы, и продолжают свою работу. Посредством аппарата семафоров должно быть обеспечено, чтобы не были одновременно запущены две программы от одного процесса. В процессе работы через очередь сообщений родительский процесс информируется, какие программы и от имени кого запущены.
9. Родительский процесс помещает в сегмент разделяемой памяти имена программ из предыдущих лабораторных работ, которые могут быть запущены. Выполняя некоторые циклы работ, порожденные процессы случайным образом выбирают имена программ из таблицы сегмента разделяемой памяти, запускают эти программы, и продолжают свою работу. Посредством аппарата семафоров должно быть обеспечено, чтобы не были одновременно запущены две одинаковые программы. В процессе работы через очередь сообщений родительский процесс информируется, какие программы и от имени кого запущены.
10. Программа моделирует работу монитора обработки сообщений. Порожденные процессы, обладающие различными приоритетами и выполняющие некоторые циклы работ, посредством очереди сообщений передают родительскому процессу имена программ из предыдущих лабораторных работ, которые им должны быть запущены. Родительский процесс, обрабатывая сообщения в соответствии с их приоритетами, следит, чтобы одновременно было запущено не более трех программ.

Контрольные вопросы

1. В чем разница между двоичным и общим семафорами?
2. Чем отличаются **P()** и **V()**-операции от обычных операций увеличения и уменьшения на единицу?
3. Для чего служит набор программных средств IPC?
4. Для чего введены массовые операции над семафорами в ОС UNIX?
5. Каково назначение механизма очередей сообщений?
6. Какие операции над семафорами существуют в ОС UNIX?
7. Каково назначение системного вызова **msgget()**?
8. Какие условия должны быть выполнены для успешной постановки сообщения в очередь?
9. Как получить информацию о владельце и правах доступа очереди сообщений?
10. Каково назначение системного вызова **shmget()**?

Порядок выполнения работы.

1. Ознакомиться с заданием к лабораторной работе.
2. Ознакомиться с основными понятиями механизма IPC.
3. Изучить набор системных вызовов, обеспечивающих решение задачи.
4. Отладить и оттестировать составленную программу, используя инструментарий ОС UNIX.
5. Защитить лабораторную работу, ответив на контрольные вопросы.

Требования к отчету.

Отчет должен включать :

- название работы и ее цель;
- Основные этапы отладки программы;
- иллюстрации с примерами системных вызовов, обеспечивающих решение задачи.

Список литературы.

1. Таненбаум Э., Современные операционные системы. 3-е издание. – Питер, 2010 – 1116с.
2. Леонов В. Команды Linux. – Эксмо, 2011 – 176с.