Программа лабораторной работы по курсу ОС
по теме:
Управление процессами и потоками

1. *Управление заданиями*

1.1. Запустите в фоновом режиме несколько утилит, например:
*cat *.c > myprog & lpr myprog & lpr intro&*
Воспользуйтесь командой *jobs* для анализа списка заданий и очередности их выполнения.
Позаботьтесь об уведомлении о завершении одного из заданий с помощью команды notify.
Аргументом команды является номер задания.
Верните невыполненные задания в приоритетный режим командой *fg*. Например: *fg %3*
Отмените одно из невыполненных заданий.

1.2. Ознакомьтесь с командой *nohup(1*).
Запустите длительный процесс по *nohup(1)*. Завершите сеанс работы. Снова войдите в систему и проверьте таблицу процессов. Поясните результат.

1.3. Определите *uid* процесса, каково минимальное значение, и кому оно принадлежит. Каково минимальное и максимальное значение *pid*, каким процессам принадлежат. Отобразите в log-файле.

1.4. Проанализируйте множество системных процессов, как их отличить от прочих, предложите способ, подтвердите экспериментально. Перечислите назначение самых важных из них.

1.5. Дайте характеристику процессу (например, прародителю всех пользовательских процессов вашей сессии), используя информацию из псевдоФС */proc* . (Подготовьте и приведите необходимые фрагменты файлов и каталогов для подтверждения своих выводов).

2. *Порождение и запуск процессов*

Используя системные функции *fork();* семейства *execl(); wait(); exit()* :

2.1. Создайте программу на основе *одного файла* (исходного, а затем исполняемого) с *псевдораспараллеливанием* вычислений посредством *порождения процесса-потомка*.

В каждом процессе сначала выполните однократные вычисления и вывод на терминал идентифицирующую его информацию (*pid, ppid* и т.п.) в течение его исполнения. Обратите внимание, какой процесс на каком этапе владеет процессорным ресурсом. В последней исполняемой команде функции *main()* выведите сообщение о завершении программы. Объясните результаты.

Сделайте выводы *об использовании адресного пространства*.

Затем однократные вычисления *замените на циклы*, длительность исполнения которых достаточна для наблюдения *конкуренции процессов* за процессорный ресурс.

2.2. Создайте программы родителя и потомка с размещением *в разных файлах* (father.c и son.c).
Для фиксации состояния таблицы процессов в файле целесообразно использовать в коде программы системный вызов *system("command_line"), т.е.*

system("утилита -ключи > имяфайла"), применяя различные ключи используемой утилиты, например, *system("ps -l> file")* .

Запустите на выполнение father.out сначала в *обычном*, а затем в *фоновом* режимах (*father &*), получите таблицы процессов на др. теминале и в файле.

2.3. Выполните создание процессов с использованием *различных функций* семейства *exec()* с разными параметрами функций семейства, приведите результаты эксперимента, меняя *значения переменных окружения* и передавая *массив входных параметров*.

2.4. Изменяя длительности выполнения *родственных* процессов и параметры системных вызовов, рассмотрите **3 ситуации** с нормальным завершением процесса, со сменой родителя и фиксацией состояния *zombie* (м.б. иные названия состояния в системе):

а) процесс-отец запускает процесс-сын и ***ожидает*** его завершения;
б) процесс-отец запускает процесс-сын и, не ожидая его завершения, завершается сам. Зафиксируйте изменение родительского идентификатора процесса-сына, каков ***pid нового родителя***, что это за процесс, и каково его назначение в системе;
в) процесс-отец запускает процесс-сын и ***не ожидает*** его завершения; процесс-сын завершает свое выполнение. Зафиксируйте появление процесса-***зомби***, для этого включите команду *ps* в программу father.c

Организуйте программу *многопроцессного* функционирования так, чтобы результатом ее работы была демонстрация всех трех ситуаций с отображением в итоговом файле соответствующих таблиц процессов (направьте вывод как на терминал, так и в файл).

3. *Управление процессами посредством сигналов (Linux)*

3.1. С помощью команды *kill -l* ознакомьтесь с перечнем сигналов, поддерживаемых процессами, а также с системными вызовами *kill(2)*, *signal(2)*. Создайте программу, демонстрирующую все возможные **реакции процесса** (или процессов) на поступление сигнала.

Например, подготовьте программы следующего содержания:
а.) процесс father порождает процессы son1, son2, son3 и запускает на исполнение программные коды из соответствующих исполняемых файлов;
б.) далее родительский процесс осуществляет управление потомками, для этого он генерирует сигнал каждому пользовательскому процессу;
в.) в пользовательских процессах-потомках необходимо обеспечить:
для son1 - реакцию на сигнал *по умолчанию*;
для son2 - реакцию *игнорирования*;
для son3 - *перехватывание и обработку* сигнала.
Сформируйте файл-проект из четырех файлов, скомпилируйте, запустите программу.
Проанализируйте таблицу процессов до и после отправки сигналов с помощью вызова *system("ps -s >> file");*. Обратите особое внимание на реакцию для последнего потомка.

3.2. Проанализируйте значение, возвращаемое функцией *wait(&status)*. Как связана *wait()* с SIGCHLD. Предложите эксперимент, позволяющий родителю отслеживать *подмножество* порожденных потомков, используя функции *waitpid()* (для ожидания завершения процесса с указанным *pid*).

4. *Многонитевое функционирование*

4.1. Подготовьте программу, формирующую несколько нитей (потоков). Нити для эксперимента могут быть практически идентичны. Используйте различные функции *pthread_create()*, *clone()*.
4.2. После запуска программы проанализируйте выполнение нитей, их *идентификацию, распределение во времени, наследуемые* (от процесса) *параметры* (например, используя *ps* с ключами и/или др. способы).
4.3. Проанализируйте *ресурсы*, *разделяемые нитями* одного процесса, подтвердите экспериментально.
4.4. Попробуйте удалить одну из нитей, зная ее идентификатор, командой *kill*. Приведите и прокомментируйте результат.

5. *Планирование*

Для упорядочивания экспериментов и упрощения восприятия результатов сначала имеет смысл сосредоточиться на **одноядерном** (однопроцессорном) выполнении однонитевых процессов при наблюдении за планированием. А затем при желании добавить многопроссорность/многоядерность.

5.1. Определите *политику планирования* и *приоритет*, установленные ***по умолчанию***, для процессов и потоков, запускаемых пользователем из *shell* (сначала из таблицы процессов, а затем программно). Проанализируйте *очередность* исполнения процессов, проведя соответствующий эксперимент
(например, в программах, созданных ранее, можно предусмотреть вывод условных идентификаторов процессов/нитей (без перевода строки) так, чтобы последовательность предоставления им процессора была очевидна на длительном интервале времени (множество квантов).

5.2. Ознакомьтесь с выполнением команды *nice(1)* и системного вызова *getpriority(2).* Попытайтесь ***изменить приоритет***, зафиксируйте реакцию системы, есть ли разница в приоритетах для системных и пользовательских процессов.

5.3. Измените процедуру планирования на **FIFO**, 5.3.1. задайте при этом **одинаковый** приоритет процессам, повторите эксперимент п.5.1 для определения порядка (очередности) предоставления процессора процессам. 5.3.2. Определите *границы приоритетов* (создайте для этого программу). 5.3.3. Задайте **разные приоритеты** процессам: как это повлияло на очередность исполнения процессов, подтвердите экспериментально.

5.4. Измените процедуру планирования на **RR** и полностью повторите эксперименты п.5.3 (очередность, приоритеты).
Определите величину *кванта*.
Поменяйте порядок очереди в RR-процедуре, используя функцию *sched_yield()*.

5.5. Можно ли задать *разные процедуры* планирования *разным процессам* с *одинаковыми приоритетами*. Как они будут конкурировать, подтвердите экспериментально. Существует ли понятие *приоритетности* по отношению к *политике планирования*?

5.6. Для ***потоков одного процесса*** определите политику планирования по *умолчанию*.
5.6.1. Приведите эксперименты с **изменением политики** для *всех* потоков одного процесса при условии их *равных приоритетов.* Измените политику только для *одного* или нескольких потоков из множества всех равноприоритетных потоков одного процесса.
Совпадают ли результаты с результатами предыдущих пунктов этого раздела, обоснуйте.
5.6.2. В каких случаях (при каких политиках) установка *разных* приоритетов повлияет на очередность исполнения потоков одного процесса. Сравните с результатами по независимым потокам.

6. *Наследование*
Проанализируйте наследование на этапах *fork()* и *exec()*. Проведите эксперименты с родителем и потомками:
6.1. по доступу к одним и тем же *файлам*, открытым родителем.
6.2. наследованию приоритетов и политике планирования,
6.3. *диспозиции* и наследованию *сигналов*.
6.4. Уточните наследование для функции *clone()*.

Отчетность по данной работе включает*:*
- лог-файл с **профилем системы**, на кот. выполнялась работа (+ идентификация студента-исполнителя),
- исходные **коды** программ по каждому пункту,
- а также **лог-файлы** исполнения работы по каждому пункту с комментариями исполнителя. Для формирования собственных логов можно использовать «отфильтрованные» системные логи (с указанием пути их размещения в системе).

Для претендующих на **«отлично»** желательно дополнение результатов п.5 (*Планирование*) выборочно результатами средств трассировки и визуализации.