# Министерство образования Республики Беларусь

УПРАВЛЕНИЕ ОБРАЗОВАНИЯ МОГИЛЕВСКОГО ОБЛАСТНОГО ИСПОЛНИТЕЛЬНОГО КОМИТЕТА

УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ

«Могилевский государственный политехнический колледж»

# Домашняя контрольная работа №1

По дисциплине: «**Системное программное обеспечение**»

Группа ПО-455

## Выполнил О. И. Любаль

Шифр 14

# 2022

**17 Дайте определение термину «Процесс» в ОС Linux**

Термин "процесс" впервые появился при разработке операционной системы Multix и имеет несколько определений, которые используются в зависимости от контекста. Процесс - это:

1. программа на стадии выполнения
2. "объект", которому выделено процессорное время
3. асинхронная работа

Для описания состояний процессов используется несколько моделей. Самая простая модель - это модель трех состояний. Модель состоит из:

1. состояния выполнения
2. состояния ожидания
3. состояния готовности

Выполнение - это активное состояние, во время которого процесс обладает всеми необходимыми ему ресурсами. В этом состоянии процесс непосредственно выполняется процессором.

Ожидание - это пассивное состояние, во время которого процесс заблокирован, он не может быть выполнен, потому что ожидает какое-то событие, например, ввода данных или освобождения нужного ему устройства.

Готовность - это тоже пассивное состояние, процесс тоже заблокирован, но в отличие от состояния ожидания, он заблокирован не по внутренним причинам (ведь ожидание ввода данных - это внутренняя, "личная" проблема процесса - он может ведь и не ожидать ввода данных и свободно выполняться - никто ему не мешает), а по внешним, независящим от процесса, причинам. Когда процесс может перейти в состояние готовности? Предположим, что наш процесс выполнялся до ввода данных. До этого момента он был в состоянии выполнения, потом перешел в состояние ожидания - ему нужно подождать, пока мы введем нужную для работы процесса информацию. Затем процесс хотел уже перейти в состояние выполнения, так как все необходимые ему данные уже введены, но не тут-то было: так как он не единственный процесс в системе, пока он был в состоянии ожидания, его "место под солнцем" занято - процессор выполняет другой процесс. Тогда нашему процессу ничего не остается как перейти в состояние готовности: ждать ему нечего, а выполняться он тоже не может.

Из состояния готовности процесс может перейти только в состояние выполнения. В состоянии выполнения может находится только один процесс на один процессор. Если у вас n-процессорная машина, у вас одновременно в состоянии выполнения могут быть n процессов.

Из состояния выполнения процесс может перейти либо в состояние ожидания или состояние готовности. Почему процесс может оказаться в состоянии ожидания, мы уже знаем - ему просто нужны дополнительные данные или он ожидает освобождения какого-нибудь ресурса, например, устройства или файла. В состояние готовности процесс может перейти, если во время его выполнения, квант времени выполнения "вышел". Другими словами, в операционной системе есть специальная программа - планировщик, которая следит за тем, чтобы все процессы выполнялись отведенное им время. Например, у нас есть три процесса. Один из них находится в состоянии выполнения. Два других - в состоянии готовности. Планировщик следит за временем выполнения первого процесса, если "время вышло", планировщик переводит процесс 1 в состояние готовности, а процесс 2 - в состояние выполнения. Затем, когда, время отведенное, на выполнение процесса 2, закончится, процесс 2 перейдет в состояние готовности, а процесс 3 - в состояние выполнения.

Диаграмма модели трех состояний представлена на рисунке 1.

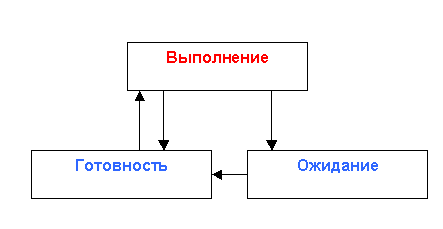


Рисунок 1. Модель трех состояний

Более сложная модель - это модель, состоящая из пяти состояний. В этой модели появилось два дополнительных состояния: рождение процесса и смерть процесса. Рождение процесса - это пассивное состояние, когда самого процесса еще нет, но уже готова структура для появления процесса. Как говорится в афоризме: "Мало найти хорошее место, надо его еще застолбить", так вот во время рождения как раз и происходит "застолбление" этого места. Смерть процесса - самого процесса уже нет, но может случиться, что его "место", то есть структура, осталась в списке процессов. Такие процессы называются зобми

Диаграмма модели пяти состояний представлена на рисунке 2.

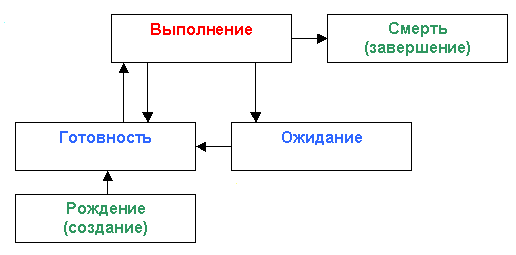


Рисунок 2. Модель пяти состояний

Над процессами можно производить следующие операции:

1. Создание процесса - это переход из состояния рождения в состояние готовности
2. Уничтожение процесса - это переход из состояния выполнения в состояние смерти
3. Восстановление процесса - переход из состояния готовности в состояние выполнения
4. Изменение приоритета процесса - переход из выполнения в готовность
5. Блокирование процесса - переход в состояние ожидания из состояния выполнения
6. Пробуждение процесса - переход из состояния ожидания в состояние готовности
7. Запуск процесса (или его выбор) - переход из состояния готовности в состояние выполнения

Для создания процесса операционной системе нужно:

1. Присвоить процессу имя
2. Добавить информацию о процессе в список процессов
3. Определить приоритет процесса
4. Сформировать блок управления процессом
5. Предоставить процессу нужные ему ресурсы

Процесс не может взяться из ниоткуда: его обязательно должен запустить какой-то процесс. Процесс, запущенный другим процессом, называется дочерним (child) процессом или потомком. Процесс, который запустил процесс называется родительским (parent), родителем или просто - предком. У каждого процесса есть два атрибута - PID (Process ID) - идентификатор процесса и PPID (Parent Process ID) - идентификатор родительского процесса.

Процессы создают иерархию в виде дерева. Самым "главным" предком, то есть процессом, стоящим на вершине этого дерева, является процесс init (PID=1).

**46 Объясните, что представляет собой индексный дескриптор в ОС Linux, какую информацию он содержит**

**inode** (произносится а́йнод или ино́д), индексный дескриптор — это структура данных в традиционных для ОС UNIX файловых системах (ФС), таких как UFS. В этой структуре хранится метаинформация о стандартных файлах, каталогах или других объектах файловой системы, кроме непосредственно данных и имени. Каждый файл ассоциирован с одним inode, хотя может иметь несколько имен в системе, каждое из которых указывает на один и тот же inode.

Индексный дескриптор содержит информацию о расположении данных файла. Поскольку дисковые блоки хранения данных файла в бщем случае располагаются не последовательно, inode, должен хранить физические адреса всех блоков, принадлежащих данному файлу. В индексном дескрипторе эта информация хранится в виде массива, каждый элемент которого содержит физический адрес дискового блока, а индексом массива является номер логического блока файла. Массив имеет фиксированный размер и состоит из 13 элементов. При этом первые 10 элементов адресуют непосредственно блоки хранения данных файла. Одиннадцатый элемент адресует блок, в свою очередь содержащий адреса блоков хранения данных файла. Двенадцатый элемент указывает на дисковый блок, также хранящий адреса блоков, каждый из который адресует блок хранения данных файла. И, наконец, тринадцатый элемент используется для тройной косвенной адресации, когда для нахождения адреса блока хранения данных файлаиспользуются три дополнительных блока. Такой подход позволяет при относительно небольшом фиксированном размере индексного дескриптора поддерживать работу с файлами, размер которых может изменяться от нескольких байтов до десятка мегабайтов. Для относительно небольших файлов (до 10 Кбайт при размере блока 1024 байтов) используется прямая индексация, обеспечивающая максимальную производительность. Для файлов, размер которых не превышает 266 кбайт (10 кбайт + 256х1024), достаточно простой косвенной адресации. Наконец, при использовании тройной косвенной адресации можно обеспечить доступ к 16777216 блокам (256х256х256).

Стандарты POSIX описывают поведение файловой системы как потомка традиционных файловых систем UNIX — UFS. Регулярные файлы должны иметь следующие атрибуты:

длина файла в байтах;

идентификатор (ID) устройства (это идентифицирует устройство, содержащее файл);

ID пользователя, являющегося владельцем файла;

ID группы файла;

режим файла, определяющий какие пользователи могут считывать, записывать и запускать файл;

Timestamp указывает дату последнего изменения инода (ctime, change time), последней модификации содержимого файла (mtime, modification time), и последнего доступа (atime, access time);

счетчик ссылок указывают количество жестких ссылок, указывающих на индексный дескриптор;

указатели на блоки диска, хранящие содержимое файла (подробнее...).

Системный вызов stat считывает номер индексного дескриптора файла и некоторую информацию из него.

Индексный дескриптор не содержит:

имени файла, которое содержится в блоках хранения данных каталога;

содержимого файла, которое размещено в блоках хранения данных.

Множество программ, используемых системными администраторами в операционной системе (ОС) UNIX, часто используют номера индексных дескрипторов для обозначения файлов. Популярная встроенная программа проверки жестких дисков fsck или команда pfiles могут послужить в данном случае примерами, так как у них есть необходимость естественным образом конвертировать номера индексных дескрипторов в пути файлов и обратно. Это может быть дополнено использованием программы поиска файлов find с ключом -inum или командой ls с соответствующим ключом (которым на большинстве платформ является -i). Иноды могут 'закончиться'. В этом случае нельзя записать информацию на устройство, даже если там достаточно свободного места.

**75 Опишите алгоритм Деккера, позволяющий разрешить проблему взаимного исключения путем использования только одной блокировки памяти**

Алгоритм Деккера — первое известное корректное решение проблемы взаимного исключения в параллельном программировании. Эдсгер Дейкстра ссылается на голландского математика Т. Деккера[en] как на автора данного алгоритма в своей работе о межпроцессном взаимодействии. Он позволяет двум потокам выполнения совместно использовать неразделяемый ресурс без возникновения конфликтов, используя только общую память для коммуникации.

Если два процесса пытаются перейти в критическую секцию одновременно, алгоритм позволит это только одному из них, основываясь на том, чья в этот момент очередь. Если один процесс уже вошёл в критическую секцию, другой будет ждать, пока первый покинет её. Это реализуется при помощи использования двух флагов (индикаторов «намерения» войти в критическую секцию) и переменной turn (показывающей, очередь какого из процессов наступила). Рассмотрим пример (рисунок 1).

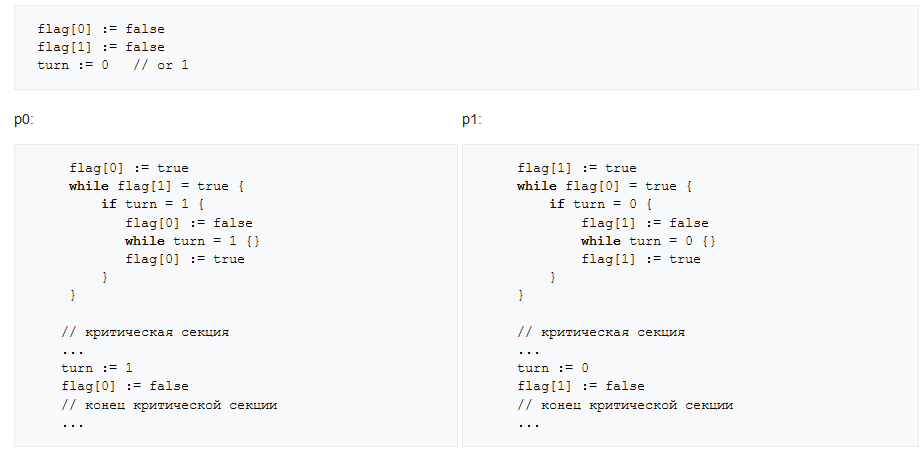


Рисунок 1. Пример блокирующих процедур (псевдокод)

Процессы объявляют о намерении войти в критическую секцию; это проверяется внешним циклом «while». Если другой процесс не заявил о таком намерении, в критическую секцию можно безопасно войти (вне зависимости от того, чья сейчас очередь). Взаимное исключение всё равно будет гарантировано, так как ни один из процессов не может войти в критическую секцию до установки этого флага (подразумевается, что, по крайней мере, один процесс войдёт в цикл «while»). Это также гарантирует продвижение, так как не будет ожидания процесса, оставившего «намерение» войти в критическую секцию. В ином случае, если переменная другого процесса была установлена, входят в цикл «while» и переменная turn будет показывать, кому разрешено войти в критическую секцию. Процесс, чья очередь не наступила, оставляет намерение войти в критическую секцию до тех пор, пока не придёт его очередь (внутренний цикл «while»). Процесс, чья очередь пришла, выйдет из цикла «while» и войдёт в критическую секцию.

Алгоритм Деккера гарантирует взаимное исключение, невозможность возникновения взаимной блокировки или зависания. Рассмотрим, почему справедливо последнее свойство. Предположим, что p0 остался внутри цикла «while flag[1]» навсегда. Поскольку взаимная блокировка произойти не может, рано или поздно p1 достигнет своей критической секции и установит turn = 0 (значение turn будет оставаться постоянным пока p0 не продвигается). p0 выйдет из внутреннего цикла «while turn = 1» (если он там находился). После этого он присвоит flag[0] значение true и будет ждать, пока flag[1] примет значение false (так как turn = 0, он никогда не выполняет действия в цикле «while»). В следующий раз когда p1 попытается войти в критическую секцию, он будет вынужден исполнить действия в цикле «while flag[0]». В частности, он присвоит flag[1] значение false и будет исполнять цикл «while turn = 0» (так как turn остаётся равной 0). Когда в следующий раз управление перейдёт к p0, он выйдет из цикла «while flag[1]» и войдёт в критическую секцию.

Если модифицировать алгоритм так, чтобы действия в цикле «while flag[1]» выполнялись без проверки условия «turn = 0», то появится возможность зависания (англ. starvation). Таким образом, все шаги алгоритма являются необходимыми.

Одним из преимуществ алгоритма является то, что он не требует специальных команд «проверить-установить» — атомарных операций чтения, модификации и записи — и вследствие этого он легко переносим на разные языки программирования и архитектуры компьютеров. Недостатками можно назвать его применимость только к случаю с двумя процессами и использование ждущего цикла вместо приостановки процесса: использование ждущего цикла предполагает, что процессы должны проводить минимальное количество времени внутри критической секции.

Современные операционные системы предоставляют примитивы синхронизации, более общие и гибкие по сравнению с алгоритмом Деккера. Тем не менее, следует заметить, что в случае отсутствия реальной одновременности между двумя процессами операции входа в критическую секцию и выхода из неё будут являться очень эффективными при использовании этого алгоритма.

Многие современные микропроцессоры исполняют инструкции не по порядку, даже порядок доступа к памяти может не соблюдаться (см. упорядоченность доступа к памяти). Алгоритм не будет работать на SMP-машинах, оборудованных такими процессорами, если не использовать барьеры памяти.

Кроме того, оптимизирующие компиляторы могут проводить такие преобразования программы, что данный алгоритм перестанет работать независимо от проблем аппаратной платформы. Такой компилятор может обнаружить, что индикаторные переменные flag[0] и flag[1] не читаются внутри цикла. Тогда, с помощью процесса, который называется выносом инварианта из цикла, он удалит из генерируемого кода операции записи в эти переменные, посчитав их избыточными. Если компилятор обнаружит, что переменная turn никогда не изменяется во внутреннем цикле, то он может выполнить аналогичное преобразование, что приведёт к потенциальному бесконечному циклу. Если будет сделано любое из этих преобразований, алгоритм перестанет работать вне зависимости от аппаратной архитектуры. Язык программирования может предусматривать ключевые слова (директивы), запрещающие компилятору производить описанные оптимизации для указанной переменной.

**100 Опишите семафоры Дейкстры. Объясните, чем обеспечивается взаимное исключение при выполнении примитивов P и V.**

Понятие семафорных механизмов было введено Дейкстрой.

Семафор (semaphore) — это переменная специального типа, которая доступна параллельным процессам только для двух операций — закрытия и открытия, названных соответственно операциями Р и V.

Эти операции являются примитивами относительно семафора, который указывается в качестве параметра операций. Здесь семафор играет роль вспомогательного критического ресурса, так как операции Р и V неделимы при своем выполнении и взаимно исключают друг друга.

Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, идентифицируемое значением семафора, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. При отказе доступа к критическому ресурсу используется режим пассивного ожидания. Поэтому в состав механизма включаются средства формирования обслуживания очереди ожидающих процессов. Эти средства реализуются супервизором операционной системы. Необходимо отметить, что в силу взаимного исключения примитивов попытка в различных параллельных процессах одновременно выполнить примитив над одним и тем же семафором приведет к тому, что он окажется успешной только для одного процесса. Все остальные процессы на время выполнения примитива будут взаимно исключены.

Основным достоинством семафорных операций является отсутствие состояния активного ожидания, что может существенно повысить эффективность работы мультизадачной системы.

В настоящее время на практике используется много различных видов семафорных механизмов. Варьируемыми параметрами, которые отличают различные виды примитивов, являются начальное значение и диапазон изменения значения семафора, логика действий семафорных операций, количество семафоров, доступных для обработки при исполнении отдельного примитива. Обобщенный смысл примитива P(S) состоит в проверке текущего значения сема­фора S. Если оно не меньше нуля, то осуществляется переход к следующей за примитивом операции. В противном случае процесс снимается на некоторое время с выполнения и переводится в состояние пассивного ожидания. Находясь в списке заблокированных, ожидающий процесс не проверяет семафор непрерывно, как в случае активного ожидания. Вместо него процессор может исполнять другой процесс, реально выполняющий какую-то полезную работу.

Рассмотрим простейший вариант – двоичные семафоры, они же мьютексы (mutex, от слов MUTual EXclusion – взаимное исключение).

Двоичным семафором называется переменная S, которая может принимать значения 0 и 1 и для которой определены только две операции.

· P(S) – операция занятия (закрытия) семафора. Она ожидает, пока значение S не станет равным 1, и, как только это случится, присваивает S значение 0 и завершает свое выполнение. Очень важно: операция P по определению неделима, т.е. между проверкой и присваиванием не может вклиниться другой процесс, который бы изменил значение S.

· V(S) – операция освобождения (открытия) семафора. Она просто присваивает S значение 0.

Чем переменная-семафор отличается от обычной булевой переменной? Тем, что для нее недопустимы никакие иные операции, кроме P и V. Нельзя написать в программе S:=1 или if(S)then ... , если S определена как семафор.

Чем операция P отличается от варианта с проверкой и присваиванием, который мы выше признали неудовлетворительным? Неделимостью. Но это «по определению», а как на практике добиться этой неделимости? Это отдельный, вполне решаемый вопрос.

Заслуга Дейкстры как раз в том, что он разделил проблему взаимного исключения на две независимые проблемы разных уровней:

· на уровне реализации: как обеспечить работу семафоров в соответствии с их определением;

· на уровне взаимодействия процессов: как написать корректно работающую программу, если в распоряжении программиста имеются семафоры.

Решать эти две задачи по отдельности легче, чем обе вместе, при этом решать их обычно должны разные люди: первую – разработчики ОС, а вторую – разработчики прикладной программы.

Рассмотрим сначала реализацию. Очевидно, функции P и V удобнее и надежнее один раз реализовать в ОС, чем каждый раз по-новому – в прикладных программах. (Названия этих функций могут в конкретных системах быть и иными, более выразительными.)

Системная функция P(S) должна проверить, свободен ли семафор S. Если свободен (S = 1), то система занимает его (S := 0) и на этом функция завершается. Если же семафор занят, то система блокирует процесс, вызвавший функцию P, и запоминает, что этот процесс блокирован по ожиданию освобождения семафора S. Таким образом, при реализации семафоров удается избежать активного ожидания.

Неделимость операции обеспечивается тем, что во время выполнения системой функции P переключение процессов запрещено. В крайнем случае, ОС имеет возможность для этого на короткое время запретить прерывания.

Системная функция V(S) – это, конечно, не просто присваивание S := 1. Кроме этого, система должна проверить, нет ли среди спящих процессов такого, который ожидает освобождения семафора S. Если такой процесс найдется, система разблокирует его, а переменная S в этом случае сохраняет значение 0 (семафор снова занят, теперь уже другим процессом).

Может ли случиться так, что несколько спящих процессов ждут освобождения одного и того же семафора? Да, так вполне может быть. Какой из этих процессов должен быть разбужен системой? С точки зрения корректности работы и соответствия определениям функций P и V – любой, но только один. С точки зрения эффективности работы – вероятно, надо разбудить самый приоритетный процесс, а в случае равенства приоритетов… ну, видимо, тот, который спит дольше.

Теперь, когда мы разобрались с реализацией семафоров, можно о ней забыть и помнить только, что семафоры существуют и могут быть использованы при необходимости.

Рассмотрим теперь вторую половину задачи – использование семафоров для управления взаимодействием процессов. Как можно реализовать корректную работу процессов с критическими секциями, если использовать двоичный семафор? Да очень просто.

Процесс A: Процесс B:

P(S); (критическая секция A) V(S); . . .. P(S); (критическая секция B) V(S);

Надо только проследить, чтобы до начала работы процессов семафор S был открыт.

**Список используемых источников**

1. Артамонова, Н.В. Операционные системы для организации производства в промышленности/Н.В. Артамонова – СПб: ГУАП, 2012.
2. Гордеев, А.В. Системное программное обеспечение / А.В.Гордеев, А.Ю. Молчанов– СПб: Питер, 2003.
3. Дроздов, С.Н. Операционные системы: учебное пособие / С.Н.Дроздов. – РнД: Феникс, 2016.
4. Иртегов, Д. Введение в операционные системы / Д. Иртегов – СПб: BHV , 2012.
5. Коньков, К.А. Устройство и функционирование ОС Windows / К.А. Коньков – М: Бином, 2012.
6. Партыка, Т.Л. Операционные системы, среды и оболочки: Учебное пособие/ Т.Л.Партыка, И.И.Попов. – М: Форум, 2018.