Московский Государственный Университет им.
М.В.Ломоносова

Конспект лекций

по курсу "Распределённые операционные системы"

Автор: Нокель Михаил
Группа: 420
Факультет: ВМК
Лектор: Крюков Виктор Алексеевич, Бахтин Владимир

6 мая 2010 г. LaTeX

Содержание

1 Лекция 1. История операционных систем

12 февраля

Периоды:

1. *1940-1950-е годы*: "использование машин как персональных ЭВМ". Доступ предоставлялся монопольно одному пользователю. Появились компоненты – предшественники ОС:

   (a) управляющая программа, обеспечивающая интерфейс с пользователем.

   (b) программы управления вводом/выводом.

2. *1950-е годы*: пакетная обработка. Потребовались новые возможности:

   (a) механизмы защиты ОП (проверка по адресу с помощью двух регистров: начала и конца доступной памяти)

   (b) механизмы защиты диска – введение привилегированного режима

   (c) механизм прерываний:

   - передача управления с сохранением информации на диске
   - передача управления с сохранением информации в стеке
   - передача управления в виде сообщений ОС

   (d) механизм таймера

   Если какого-то компонента не хватает, то для реализации мультипрограммного режима необходима интерпретация, а это очень сильное замедление программ.

3. *Середина 1960-х годов*: режим разделения времени (РРВ). Появление:

   (a) терминала (устройства ввода/вывода, посылки сигнала ОС)

   (b) квантования времени

   (c) страничной и сегментной организации памяти (сегментная - для контроля за деятельностью программиста) -> сегментно-страничной организации памяти

   Появление multix.

4. *Середина 1970-х годов*: появление многомашинных комплексов, многопроцессорных ЭВМ и сетей ЭВМ. Цели появления: специализация, эффективность и надёжность.

2 Лекция 2. История операционных систем (продолжение)

19 февраля

Многомашинные комплексы – системы компьютеров с общими дисками (где помещался буфер ввода и буфер вывода). Также они могут быть связаны через каналы связей. Сети позволяют произвольное количество произвольных мащин подключать. Произошло разделение функций между ОС и протоколами передачи данных. Обеспечивали новые три функции:

- диалог с пользователем. Потребовалась работа с удалённой ЭВМ. Появились виртуальные терминалы.
- доступ к файлам. Запросы к файлам должны быть оттранслированы на машину, где эти файлы располагаются. Появление распределённых файловых систем.
- запуск в пакетном режиме.

1. *80-е годы*: появление ПЭВМ (персональных ЭВМ). Отличительные особенности: ПЭВМ обладали экраном и большой мощностью процессора, приходящейся на одного пользователя. Отрабатывались новые возможности взаимодействия с ОС. ОС в начале этого периода были однопользовательскими и однозадачными. Функции распределения ресурсов отсутствовали. Потом появилась сегментная организация памяти (начиная с 86х процессоров). В середине 80х годов появились 386е процессоры с сегментно-страничной памятью. ОС за это десятилетие усложнились и не уступали по возможностям тем ОС, которые работал на больших ЭВМ.
2. *90-е годы*: появление MPP - многопроцессорной ЭВМ с распределённой памятью; открытых систем, Интернета. В 1992 году самой производительной стала машина MPP с распределённой памятью. И с тех пор они и до сих пор являются лидерами. Способствовали этому 2 основные причины:

(a) работа с общей памятью: расположить много процессоров вокруг общей памяти - проблема.

(b) поддержка согласованного состояния КЭШ-памяти (в случае если много процессоров есть с общей памятью). Решение: использование сквозного КЭШ. Достигается путём введения общей шины между процессорами, которую можно "подслушать".

Постепенно в узлах MPP стали поддерживаться UNIX-системы, потом и Windows-системы. Но традиционные ЭВМ не справлялись с поставленными задачами: работа с большим количеством сетевых портов, например. Самая тонкая проблема: традиционные ОС использовали примерно 1% на свои нужды, из-за чего возникали простои. Решение: своеобразная настройка ОС. Открытые системы - это стандартизация работы машин, на которых работают различные ОС. В результате стало возможным создание Интернет.

3. *2000-е года*: появление и распространение кластеров, распределённых систем, ГРИД-систем. Достоинство кластеров: широкая доступность. Сейчас идёт переход на многоядерные системы. Возникают проблемы:

   (a) проблема написания ПО

   (b) энергопотребление

   (c) проблема быстродействия памяти и быстродействия кристалла. В результате появились многопоточные процессоры. Суть: процессор работает не с одним потоком, а с несколькими, и переключение между ними идёт аппаратно.

   Все эти тенденции отражались в графических процессорах. Ещё в 1990х годах появились метакомпьютеры, впоследствии перешедшие в ГРИД. Сейчас широко используется и распространяются "облачные вычисления". Они позволяют не задумываться о конкретике, предоставляются только сервисы.

2.1 Причины создания распределённых систем

1. Экономическая причина. Закон Гроша: быстродействие процессора пропорционально квадрату его стоимости. Перестал действовать в 80х годах с появлением микропроцессоров.

2. Достижение высокой производительности путём объединения многих процессоров.

3. Наращивание производительности.

3 Лекция 3. Коммуникации в распределённых системах

26 февраля

Материалы по курсу лежат по следующим адресам:

- `http://sp.cs.msu.su/courses/os/distr-os-2010.zip`
- `ftp://ftp.keldysh.ru/K_student/distr-os-2010.zip`

Существуют 2 операции по передаче сообщений между машинами в распределённой системе:

1. послать (кому, адрес_начала, длина_сообщения)
2. принять (кому, адрес_памяти(куда), длина_сообщения)

При этом сообщение должно помещаться в выделенной памяти. Часто нужно как-то различать приходящие сообщения. Для этого добавляют теги - типы сообщений.

Существуют 2 способа пересылки сообщений:

1. синхронный
2. асинхронный (в этом случае надо добавить буферы)

Формула для вычисления времени передачи сообщения:

$$T = T_s + T_b * L,$$

где T_s – время старта (аппаратная составляющая), T_b – время передачи байта, L – длина сообщения.

Существуют 2 архитектуры распределённых систем:

1. *транспьютерная решётка.* Появилась в первой половине 80-х годов. Обладает двумя особенностями:
   (a) система команд: самые часто встречающиеся команды упаковываются в 1 байт (всего таким образом можно 15 команд упаковать)
   (b) функции планировщика процессов реализованы аппаратно.

У каждого транспьютера было по 4 входных/выходных канала. Следовательно, сообщения могли идти разными путями. Для ускорения передачи сообщений существуют 2 метода ускорения:

(a) разбиение сообщения на части при передаче, а части передаются разными маршрутами

(b) организация конвейера (разделение сообщение на k частей для организации полного зацепления). Возникает проблема: определить k оптимально.

Это были быстрые процессоры своего времени. Но транспьютеры следующего поколения не получились. Поэтому стали подвешивать к узлам процессоры Intel.

2. *сеть с шинной организацией*. Особенности:

(a) в 1 момент времени передаётся только 1 сообщение (нельзя передавать параллельно)

(b) арбитр шины разрешает возникающие коллизии

(c) можно передать 1 сообщение сразу всем

Существуют 2 способа передачи:

(a) с широковещанием

(b) без широковещания

Обе архитектуры на данный момент не соответствуют действительности:

1. много сетей существуют для параллельного обмена
2. много способов, чтобы сделать пути неодинаковыми

3.1 MPI

Message Passing Interface

Появился в 1994 году. Цели создания:

1. создать интерфейс прикладного программирования
2. обеспечить возможности эффективной коммуникации (убрать лишнее копирование в/из канала)
3. разрешить расширения для использования в гетерогенных системах

4. исходить из надёжности коммуникации (имеется проблема надёжности)

5. определить интерфейс, не очень сильно отличающийся от тех, что использовали раньше

6. интерфейс должен быть быстро переделан

Были включены:

1. коммуникации типа "точка-точка"

2. коллективные операции

3. понятие группы процессов

4. коммуникационный контекст

5. топология процессов

Все операции были синхронными или асинхронными, блокирующими или неблокирующими.

Неблокирующие операции - управление возвращается сразу же. Блокирующие операции - сообщение копируется в системный буфер, управление возвращается процессору.

Send/receive:

- адрес буфера в памяти
- количество элементов
- тип данных
- номер процессора в группе
- тег сообщения
- коммуникатор (объединение понятия группы и контекста)

Для команды "receive"номер процессора в группе и тег сообщения могут не указываться, но зато обязательно добавляется статус, с помощью которого можно узнать тег, номер процессора, количество сообщений.

Существуют 4 режима посылки сообщений:

1. стандартный

2. буферизуемый

3. синхронный
4. готовности

И соответственно 4 префикса:

1. B - Block
2. S - Sync
3. R - Ready
4. I - для неблокирующих операций

Одной из стандартных ошибок при реализации пересылки сообщений являются тупики.

Коллективные операции:

1. Барьер
2. Передача всем от одного (broadcast)
3. Сбор данных от всех (gather)
4. Рассылка всем (scatter)
5. Сбор всех от всех (all gather)
6. Рассылка всем от всех (all to all)
7. Глобальные операции (редукция)

В 1997 году появился MPI-2.0. Нововведения:

1. односторонние коммуникации
2. динамическое создание/удаление процессов
3. параллельный ввод/вывод

4 Лекция 4. Синхронизация распределённых систем

5 марта 2010 г.

Выделяют 2 вида синхронизации:

1. взаимное исключение (в системах с общей памятью)
2. координация процессов (один процесс сделал работу и сообщает другим, что можно воспользоваться его результатами)

К *взаимному исключению* предъявляются следующие требования[1]:

1. 1 процесс может только находиться в критической секции
2. Если критическая секция свободна, то вход без задержки (конечно, всегда есть задержка на реализацию входа, но это незначительное время)
3. Нет бесконечного ожидания при условии, что $T_{KS} < \infty$
4. Нет никаких предположений относительно скоростей

Возможны следующие механизмы синхронизации:

1. Логическое время. Необходима синхронизация времени между процессами, чтобы не было абсурда. Если получилось, что $T_{receive} < T_{send}$, то прибавим что-нибудь к $T_{receive}$, чтобы получилось наоборот.
2. Каналы FIFO – каналы, связывающие два процесса и работающие по принципу FIFO (раньше отправили – раньше пришло). В MPI есть правило: если один процесс посылает другому сообщения с одним тегом, то они приходят в порядке отправки.
3. Неделимое широковещание – широковещание, обеспечивающее приход всех сообщений в одном и том же порядке.
4. Выбор координаторов. Координатор – процесс, выполняющий особую роль. Задача состоит в выборе координатора. Существуют 2 алгоритма выбора:

[1] Будем считать синонимами "критическая секция"и "критический интервал". Также будем использовать термины: "вход и выход в критическую секцию (интервал)"

(a) "Задира". Логика: k-й процесс обнаружил, что предыдущий координатор перестал отвечать. Запускается выбор координатора. Посылается номер своего процесса всем процессам с большими, чем у него номерами. Каждый, кто "жив"скажет, что "я беру выборы на себя". Всегда найдётся процесс с наибольшим номером среди живых, и он становится координатором. Но алгоритм не детерминирован: непонятно, кому вначале отсылать сообщения. От этого зависит скорость выборов.

(b) Круговой. Логика: все процессы связаны в какое-то логическое кольцо. Первый, кто обнаружил неработоспособность, начинает отправлять сообщение со своим номером своему соседу. Каждый следующий добавляет свой номер к уже существующим. После первого круга инициировавший выборы получает список всех живых процессов. На втором круге все узнают нового координатора (процесса с наибольшим номером).

Существуют следующие требования к децентрализованным алгоритмам:

1. Вся информация распределена между процессами. Пример: нет единого телефонного справочника.

2. Процессы принимают решения на основе локальной информации.

3. Не должно быть единственной критической точки, выход из строя которой приведёт к краху алгоритма. Это требование надёжности.

4. Нет единого времени[2]. Добиться единого времени очень сложно, поэтому не должно быть к этому привязки.

Рассмотрим теперь алгоритмы взаимного исключения:

1. Централизованный. Среди множества процессов имеется 1 процесс-координатор. Каждый, кто хочет войти в критическую секцию, посылает координатору запрос. Координатор справедливо выдаёт право на вход в критическую секцию. Кто, первый послал, тот и получил разрешение на вход. В момент выхода процесс посылает координатору сообщение о том, что он вышел. Таким образом, всего требуются 3 сообщения.

2. Децентрализованный с временными метками. Каждый процесс спрашивает у всех остальных на предмет того, может ли он войти в

[2]Имеется в виду физическое время

критическую секцию. Каждый, кто получил его и кто не находится в критической секции и которому не нужен вход в критическую секцию, отвечает ему. Если процесс сам хочет войти в критическую секцию, то с помощью логического времени осуществляется сравнение своей метки с временем процесса, пославшего запрос[3]. Вход даётся только, если получены разрешения от всех процессов. В итоге потребуется $2 * (n - 1)$ запросов, где n – количество процессов.

Возникает вопрос: "А можно ли ускорить децентрализованный алгоритм?"При наличии неделимых широковещательных рассылок можно было бы отказаться от логического времени и получить выигрыш.

3. Маркерные алгоритмы. Тот, у кого есть маркер, может входить в критическую секцию.

   (a) круговой. Все процессы связаны в логическое кольцо, и передаётся маркер. Тот, у кого есть маркер, может войти в критическую секцию. После выхода маркер передаётся соседу по кругу. Самый эффективный, если все хотят в критическую секцию – всего 1 сообщение на вход/выход.

   (b) широковещательный. Процесс все раздаётся запрос о том, что требуется маркер. И от владельца каким-то образом получит потом. Решение:

   - организовать очередь запросов (длина N).
   - ввести нумерацию запросов – организовать массив с номерами последних удовлетворённых запросов $LN[1...N]$

   При входе в критическую секцию:

   - процесс k имеет вектор $RN_k[k] + 1$ и широковещательный запрос и должным образом корректирует вектор.
   - при этом если есть маркер, то просто вход в критическую секцию

   При выходе из критической секции:

   - коррекция массива LN. Если j-й процесс вышел, то корректируем $LN[j]$
   - новые запросы получаем из сравнения векторов RN и LN и помещаем их в очередь, предварительно проверив, что их там ещё нет.

[3]Совпасть временные метки не могут!

(c) древовидный. Идея: построение дерева процессов. Маркер будет найден, медленнее, но с меньшими затратами, чем в случае с широковещательным.

Все процессы выстроены в некоторое сбалансированное дерево и у каждой вершины-процесса есть:

- указатель в ту сторону, где находиться владелец маркера.
- очередь из тех, кому нужен маркер:
  - себе
  - соседу сверху
  - соседу снизу слева
  - соседу снизу справа

Главное отличие – отсутствие единой очереди, как в случае с широковещательным алгоритмом.

При входе в критическую секцию:

- Есть маркер → входим.
- Иначе → помещаем свой запрос в свою очередь.
- Посылаем сообщение "ЗАПРОС"

Получили сообщение:

- Если получили маркер:

  M1 Взять первый из очереди

  M2 Послать маркер автору (возможно и себе)

  M3 Поменять указатель (в сторону владельца маркера)

  M4 Исключить запрос, который был первым в очереди

  M5 Если в очереди остались запросы (т.е. маркер нам нужен), то послать сообщение-запрос в сторону маркера
- Если получили запрос:
  - Помещаем в очередь
  - Если нет маркера, посылаем запрос в сторону маркера[4]
  - Если есть маркер, переход на пункт "M1"(разобраться с очередью)

При выходе из очереди, снова на "M1"(если очередь не пуста)

Перейдём к рассмотрению координаторов процессов.

[4] нет смысла посылать запрос в сторону маркера повторно

1. Если известен производитель и потребитель, то посылаем сообщение "точка-точка"
2. Если неизвестен потребитель, то:
   - либо всем
   - либо по запросу
3. Если неизвестен ни производитель, ни потребитель, то либо:
   - через координатора
   - широковещательный запрос

5 Лекция 5. Многопроцессорные системы

12 марта 2010 г.
Владимир Бахтин[5]

Есть разрыв между временем доступа к памяти и скоростью работы процессора. На примере Itanium 2:

- для доступа к памяти L1 - 1-2 такта
- для доступа к памяти L2 - 5-7 тактов
- для доступа к memory 180-225 тактов

При этом если увеличить производительность процессора, то выигрыш будет незначительным. Так появилась идея переключения между выполняемыми потоками, за счёт чего получился серьёзный прирост производительности.

Процесс - некоторое выполнение программы, с которым связана следующая информация:

1. Таблица страниц
2. Дескрипторы открытых файлов
3. Запросы на ввод/вывод
4. и др.

[5]Много скучного неинтересного кода и абсолютное неумение вести лекцию

Что же касается нитей, то в рамках одного процесса может быть запущено N нитей. У них может быть общая память, через которую они могут взаимодействовать. С ними связана следующая информация:

1. Регистр
2. Счётчик команд
3. Стек

Т.к. существует проблема энергопотребления, то появилась идея снижения частоты процессора (и как следствие, энергопотребления) и разбиения на несколько ядер (в результате получаем выигрыш). Пример четырёхядерного процессора - Intel Core I7. У AMD есть на данный момент шестиядерный Opteron, так же есть ещё Niagara II и др.

Способы организации многопроцессорных систем:

1. Главный-подчинённый (master-slave)
2. Симметричный

Процессы бывают двух типов:

1. Взаимодействующие
   (a) разделение памяти
      i. Оперативная память
      ii. Внешняя память
   (b) обмен сообщениями
2. Независимые

Рассмотрим пример:

Пусть имеются 2 процесса:

1. $P0: \ x = x + 1$
   $LOAD \ R1, X$
   $ADD \ R1, 1$
   $STORE \ R1, x$
2. $P1: \ x = x - 1$
   $LOAD \ R1, X$
   $SUB \ R1, 1$
   $STORE \ R1, X$

Возможный способ выполнения команд:

t	$P0$	$P1$
0	$LOAD\ R1, X$	
1	$ADD\ R1, 1$	$LOAD\ R1, X$
2	$STORE\ R1, X$	$SUB\ R1, 1$
3		$STORE\ R1, X$

Как видно, в итоге получим значение счётчика, равное -1. Это одно из трёх возможных значений. То есть в зависимости от устройства конкретной машины можно получить абсолютно различные значения. Подобные ошибки трудно обнаружить и с ними трудно бороться.

Для борьбы используется ряд синхронизационных конструкций. Введём следующие требования:

1. В любой момент времени в критическом интервале может находиться только 1 процесс

2. Если в критическом интервале нет процессов, то должны получить разрешение на вход без задержки

3. Ни один процесс не должен ждать бесконечно долго разрешения на вход

4. Не должны учитывать производительность процессов

Существуют 2 режима возможного выполнения (применялись на однопроцессорных системах): MONO/MULTI. Если процесс хотел завладеть единолично, то он запускался в режиме MONO.

Очевидно, для многопроцессорных систем это не подходило. Поэтому были разработаны различные алгоритмы:

1. Алгоритм Дейкера:[6]

   proc(0) && proc(1)

   int turn;

   boolean flag[2];

   turn i=0;

[6]Работает не всегда: например, в случае распределённых систем не будет работать из-за того, что алгоритм предполагает неделимость операции чтения/записи, а в случае распределённых систем это не так (всегда есть задержки)

```
flag[0]=flag[1]=FALSE;

proc(int i) {
    while(true) {
        <вычисления_i>
        enter_region(i)
        <критический интервал>
        leave_region(i)
    }
}
void enter_region(int i) {
    try:
        flag[i]=TRUE;
        while(flag[(n+1)%]) {
            if (turn==i) continue;
            flag[i]=FALSE;
            while(turn!=i);
            goto try;
    }
}
void leave_region(int i) {
    turn=(i+1)%2;
    flag[i]=FALSE;
}
```

2. Алгоритм Петерсона. Отличается от предыдущего лишь функцией входа в критический интервал.

```
void enter_region(int i) {
    int other;
    other=1-i;
    flag[i]=TRUE;
    turn=i;
    while((turn==i)&&(flag[other]==TRUE)) ;
```

```
}
void leave_region(int i) {
    flag[i]=FALSE;
}
```

3. Алгоритм для любого числа процессов. Введём дополнительную функцию:

   TEST AND SET LOCK

   tsl(r,s) [r=s; s=1]

   Теперь реализуем функции входа/выхода:[7]

```
inter_region: tsl reg,flag
    cmp reg,0
    jnz enter_region
    ret
leave_region: move flag,#0
    ret
```

4. Алгоритм семафоров Дейкстры. У всех вышеперечисленных алгоритмов есть недостаток – активное ожидание. Решение предложил Дейкстра:

   P(S) [if (s==0) <заблокировать текущий процесс> else s=s-1] – функция захвата семафора

   V(S) [if (s==0) <разблокировать один из ранее заблокированных процессов>; s=s+1] – функция освобождения семафора

   Пример использования семафора:

```
proc (int i) {
    while (TRUE) {
        <вычисления>
        P(S)
        <КИ>
        V(S)
    }
}
```

[7] код на ассемблере!

6 Лекция 6. Синхронизация многопроцессорных систем (продолжение)

19 марта 2010 г.

Напомним:

P(S) [if (s==0) <Заблокировать процесс> else s=s-1]

V(S) [if (s==0) <Разблокировать один процесс> s=s+1]

Рассмотрим задачу производителей и потребителей. Для её реализации используем 3 семафора:

```
semaphore s=1;
semaphore empty=N;
semaphore full=0;

producer() { int item; while(TRUE) {
        produce_item(&item);
        P(empty);
        P(S);
        enter_item(item);
        V(S);
        V(full);
    }
}
consumer() {
    int item;
    while(TRUE) {
        P(full);
        remove_item(&item);
        V(S);
        V(empty);
        consume_item(item);
    }
}
```

Приведём пример из мира Unix:

```
#include <pthread.h>
pthread_mutex_t mutex=PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_lock(&mutex); //P(mutex)
pthread_mutex_unlock(&mutex); //V(mutex)
pthread_mutex_trylock(&mutex); //возврат код ответа
```

Если потребителей много, то семафоры здесь не справятся. Необходим другой механизм, а именно механизм *событий*.

Вводятся следующие операции для работы с событиями:

1. POST(S)
2. WAIT(S)
3. CLEAR(S)

Разница между семафорами о событиями существенная. При выполнении V(S) будет разблокирован один процесс, а при выполнении POST(S) – все процессы (вся очередь). В принципе, эти понятия очень похожи и можно попробовать реализовать один механизм через другой. К примеру, можно реализовать события через семафоры: при выполнении POST(S) берём все процессы из очереди и для каждого выполняем P(S),V(S).

Рассмотрим следующий алгоритм в качестве примера:

```
for (i=1;i<L1-1;i++) {
    for j=1;j<L2-1;j++) {
        A[i][j]=(A[i-1][j]+A[i][j-1]+A[i][j]+A[i][j+1])/4;
    }
}
```

Заметим, что витки этого цикла можно выполнять параллельно. Реализовать это можно по-разному: можно циклически, можно давать тому, кто раньше закончил и т.д.

```
float A[L1][L2];
struct event S[L1][L2];
int i,j;
for (i=0;i<L1;i++)
    for (j=0;j<L2;j++)
        CLEAR(S[i][j]);
for (i=0;i<L1;i++) POST(S[i][0]);
for (j=0;j<L2;j++) POST(S[0][j]);
parfor (i=1;i<L1-1;i++)
    parfor (j=1;j<L2-1;j++) {
        WAIT(S[i-1][j]);
        WAIT(S[i][j-1]);
        A[i][j]=(A[i-1][j]+A[i][j-1]+A[i+1][j]+A[i][j+1])/4;
        POST(S[i][j]);
    }
```

Здесь осуществляется запуск конвейера: сначала только 1 процесс может выполняться и с каждым шагом работы программы всё большее число процессов будут вовлечены в работу. Тем самым получаем ускорение за счёт параллельной работы программы по сравнению с последовательным.

В 1978 году был предложен механизм сообщений для того, чтобы уйти от разделяемой памяти, что было достаточно узким местом. Были предложены 2 функции для реализации:

1. $SEND\ (destination, \&message, nsize)$
2. $RECIEVE\ ([source], \&message, nsize)$

Сообщения бывают двух типов:

1. Буферизуемые
2. Небуферизуемые

Рассмотрим пример:

```
#define nsize 4
typedef int message[nsize];
#define N 100

producer() {
    message m;
    int item;
    while (TRUE) {
        RECEIVE(consumer,&m,nsize);
        produce_item(&item);
        build_message(&m,item);
        SEND(consumer,&m,nsize);
    }
}
consumer() {
    message m;
    int i,item;
    for (i=0;i<N;i++) SEND(producer,&m,nsize);
    while (TRUE) {
        RECEIVE(producer,&m,nsize);
        extract_item(&m,item);
        SEND(producer,&m,nsize);
        consume_item(item);
    }
}
```

Теперь перейдём к задаче читателей и писателей. Попробуем реализовать её с помощью двоичных семафоров:

```
int reader=0;
```

```
    semaphore s=1;
    semaphore reader=1;
    semaphore writer=1;

writer_enter() {
    P(writer);
    P(reader);
}
writer_exit() {
    V(reader);
    V(writer);
}
reader_enter() {
    P(writer);
    P(S);
    readers++;
    if (readers==1) P(reader);
    V(S);
    V(writer);
}
reader_exit() {
    P(S);
    readers–;
    if (readers==0) V(reader);
    V(S);
}
```

Также есть ещё классическая задача обедающих философов. Есть несколько философов, тарелка со спагетти и у каждого философа есть вилка слева и вилка справа. Философы сидят за круглым столом. В любой момент времени философ может захотеть покушать: он берёт правую вилку, берёт левую и приступает к трапезе. Чтобы не было deadlocka, вводится семафор на вилку.

Существуют накладки при переключении процессов. Для борьбы с этим можно использовать семафоры. Аналогично поступают и при работе с КЭШ-памятью: в некоторых операционных системах (например, в MAC OS) существуют подсказки для управления работой процессов в критических секциях.

7 Распределённая общая память DSM

26 марта 2010 г.[8]

Реализовать работу с распределённой общей памятью гораздо труднее, чем в случае с виртуальной.

7.1 Достоинства DSM

1. Удобство программирования
2. Суммарный объём памяти может быть огромным
3. DSM-системы могут наращиваться практически беспредельно
4. Программы пишутся аналогично программам для мултипроцессорных систем

Всё хорошо, но есть обман пользователя: такие машины тут же попали в top 500, а проверка для попадания туда осуществлялась на ассемблерных программах (т.е. допустим всё эффективно, а компиляторы приблизятся к этому не могут). Когда машины начали развиваться (размеры дошли до десятков процессоров), выяснилось, что на них сложно программировать.

7.2 Алгоритмы реализации DSM

Существуют следующие проблемы:

1. как поддерживать информацию о расположении удалённых данных
2. как снизить коммункиационные задержки (алгоритмы изгнания), т.е. как реже обращаться к системе через коммуникационные службы. Добиться этого можно с помощью разделяемых данных.

Отсюда вытекает несколько алгоритмов:[9]

1. *Алгоритм с центральным сервером.* Вся работа идёт через центральный сервер, являющийся узким местом.

[8] Лектор отжёг - вывел на экран документ в Worde и его скролил всю лекцию

[9] Все эти алгоритмы не годятся для использования

2. *Миграционный алгоритм.* Идея: передать действие тому, у кого под руками память. Реализация мало отличается от страничной памяти. Принципиальное различие между DSM и обычной памятью: в случае с DSM страница памяти может быть нужна и диску (в отличие от обычной памяти), эффект трешинга также присутствует. Реализация этого алгоритма возможна с использованием аппаратуры. Информация в одном месте → неэффективен.

3. *Алгоритм размножения для чтения.* Такой алгоритм вполне приемлем (одновременно может писать только 1), но для реальных программ не подходит.

4. *Алгоритм размножения для чтения и для записи.* Этот алгоритм всё равно не эффективен → нельзя эффективно реализовать DSM, не поменяв работу с памятью.

Проблемы остаются: как работать с надёжностью?

7.3 Модели консистентности

Обычное дело - *строгая консистентность*: из ячейки читаем то последнее значение, которое мы туда записали.

Рассмотрим мультипроцессор. Один процессор записал значение переменной, а чуть позже другой её прочитал (за это время не успело обновиться это значение в кэше → будет считано не последнее значение). В этом случае имеем дело с нестрогой консистентностью.

Рассмотрим модели консистентности по мере ухода от строгой:

1. *Последовательная консистентность.* Результат выполнения такой же, как если бы все операторы выполнялись бы в какой-нибудь последовательности.

   Рассмотрим пример:

$P1:$	$W(x)1$			$W(y)1$	
$P2:$			$W(z)1$		
$P3:$		$R(x)0$	$R(y)0$	$R(z)1$	$R(y)0$
$P4:$		$R(x)0$	$R(y)1$	$R(z)1$	$R(x)1$

   Здесь всё хорошо.

$P1:$	$W(x)1$			$W(y)1$	
$P2:$			$W(z)1$		
$P3:$		$R(x)0$	$R(y)1$	$R(z)0$	$R(y)1$
$P4:$		$R(x)1$	$R(y)1$	$R(z)0$	$R(y)1$

Это самый строгий алгоритм.

Возможны разные реализации: централизованные и децентрализованные, но будем предпочитать децентрализованный алгоритм. Если рассмотреть централизованный, то у нас есть координатор и то, что первое ему пришло, обратно ушло всем. Те, кто послал изменения переменной, должны дождаться подтверждения, иначе будет ошибка. Если же у нас есть неделимые широковещательные рассылки, то всё равно не можем двигаться дальше, пока не дойдёт до нас подтверждение. Главный принцип: всем изменения доходят в одном порядке, и необходимо дождаться подтверждения своего изменения.

2. *Причинная консистентность.* Записи делят на:

   - причинно-следственные
   - причинно-зависимые.

   В случае с DSM вместо причинно-зависимых берём потенциально причинно-зависимые. Основная идея реализации: все модификации переменных нумеруются.

3. *PRAM-консистентность.* Записи, которые делает процессор, должны быть видны всем и сразу, но порядок произволен. Не ждём никаких ответов. Возможны противоречивые ситуации.

4. *Процессорная консистентность.* Добавилась когерентность памяти - должен быть виден одинаковый порядок для всех. Решение – метод обгона. Можно и с помощью одного координатора (централизованно), а можно и децентрализованно.

5. *Слабая консистентность.* Главная идея этой и последующих двух моделей: использовать помимо обычных синхронизационные переменные, обеспечивать нужное видение переменных. С помощью этого удаётся обеспечить вполне приемлемые по эффективности решения. Также только когда встретятся операции синхронизации, необходимо выслать значения из КЭШа.

   У данной модели синхронизация имеет 2 смысла: то, что ты наменял, и то, что наменяли другие. Одной операции выталкивания недостаточно. Пример:

$$\begin{array}{lccccc} P1: & W(x)1 & W(x)2 & S & & \\ P2: & & & R(x)1 & R(x)2 & S \\ P3: & & & R(x)2 & R(x)1 & S \end{array}$$

   Это был пример допустимой последовательности событий.

$P1:\ \ W(x)1 \quad W(x)2 \quad S$
$P2:\ \ \qquad\qquad\qquad\quad S \quad R(x)1$

А это пример недопустимой операции.

Основное правило работы с синхронизационными переменными – доступ к синхронизационным переменным определяется моделью последовательно консистентности