* 1. **Мониторы[[1]](#footnote-1)**

Монитор (monitor) – это программное средство, разработанное Хоаром и дающее возможность управляемого совместного использование ресурсов среди асинхронных процессов, включая возможность управляемого обмена параметрами между процессами. Идея монитора заключается в создании механизма, который унифицирует взаимодействие параллельных процессов по разделяемым данным и процедурам, которые обрабатывают эти данные.

Монитор — это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации динамического распределения конкретного ресурса. Синтаксически монитор начинается ключевым словом monitor. Монитор защищает свои переменные. Доступ к переменным монитора можно получить, только используя процедуры монитора.

Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, вызывая процедуру монитора. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Если вызов был успешным, то процесс считается, находящимся в мониторе. Вход в монитор находится под жестким контролем — здесь осуществляется взаимоисключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессы, которые хотят войти в монитор, когда он уже занят, ставятся в очередь к монитору, причем режимом ожидания автоматически управляет сам монитор. Монитор сам является ресурсом.

При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждет. Проверка условия выполняется самим монитором, который и разблокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимоисключение процессов, отсутствуют серьезные проблемы, связанные с организацией параллельных взаимодействующих процессов. При первом обращении монитор присваивает своим переменным начальные значения. При каждом последующем обращении используются те значения переменных, которые сохранились от предыдущего обращения. Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдает команду ожидания с указанием условия ожидания. Процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится. Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса или, если уже имеются процессы, ожидающие освобождения данного ресурса, выполнить команду извещения (сигнализации), чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.



Рис.8

Монитор осуществляет доступ к разделяемым ресурсам посредством использования переменных типа условие (conditional). Будем такие переменные называть просто «условие». Для каждой отдельно взятой причины, по которой процесс переводится в состояние ожидания (блокировки), назначается своя переменная типа условие. На переменных «условие» определены два типа операций: wait(с) и signal(с), которые по сути являются макрокомандами. Команда wait(с) блокирует процесс, если ресурс занят, и открывает доступ к монитору, если ресурс свободен. Выполнение заблокированного процесса активизируется командой signal(с), которую вызывает другой процесс. Оператор signal(с) выполняется следующим образом: если очередь к переменной «условие» не пуста, то из очереди выбирается один из процессов и активизируется, иначе если очередь пуста, то signal(с) действий не выполняет.

*Простой монитор*

Простой монитор решает задачу выделения одиночного ресурса нескольким процессам (листинг 1).

|  |
| --- |
| RESOURCE MONITOR;  var  busy: logical;  X: conditional;  procedure acquire; // процедура «захватить»  begin  if busy then wait(X);  busy = true;  end;  procedure release; //процедура «осуществить»  begin  busy = false;  signal(X);  end;  begin  busy = false;  end. |

Листинг 1

Монитор обслуживает произвольное число процессов, ограниченное только длиной очереди. В мониторе две процедуры: acquire и release. Если процесс нуждается в захвате ресурса, он вызывает процедуру acquire. Если логическая переменная busy – ложь (false), то процесс без задержки устанавливает переменную busy в значение истина (true). Если же логическая переменная busy – истина, то по переменной условие X выполняется wait(X) и логическая переменная busy не меняется. Для освобождения ресурса процесс вызывает процедуру release. Переменной busy присваивается значение ложь (false) и signal(Х) проверяет список процессов, в очереди к переменной X, выбирает из очереди процесс и активизирует его.

*Монитор «кольцевой буфер»*

Монитор «кольцевой буфер» решает уже рассмотренную ранее задачу «производство-потребление». Буфер – это массив заданного размера. Производитель помещает в массив данные. Потребитель считывает эти данные в порядке, в котором они были помещены в буфер. Производитель последовательно заполняет элементы массива и наступит момент, когда последний элемент буфера будет заполнен. Но массив организован как кольцевой буфер. Когда заполняется последний элемент массива, происходит переход снова на первый элемент. Монитор представлен на листинге 2.

|  |
| --- |
| RESOURCE MONITOR;  var  ring\_buffer: array [0..n-1] of <type>;  pos: 0..n; //текущая позиция  j: 0..n-1; // заполняемая позиция  k: 0..n-1; // освобождаемая позиция  buffer\_full, buffer\_empty: conditional;  procedure producer (data: type)  begin  if pos == n then wait(buffer\_empty);  ring\_buffer[j] = data;  pos++; // инкремент  j = (j+1) mod n; // “заворачивание” позиции заполнения  signal(buffer\_full);  end;  procedure consumer (var data: type)  begin  if pos==0 then wait(buffer\_full);  data = ring\_buffer[k];  pos--;  k = (k+1) mod n;  signal(buffer\_empty);  end;  begin  pos = 0;  j = 0;  k = 0;  end. |

Листинг 2

Условие buffer\_full является признаком, который ждет процесс потребитель, если обнаружит, что буфер пуст. Этот признак устанавливает процесс производитель сигналом – signal(buffer\_full), когда он поместит данные в буфер. Производитель, наоборот, ждет события buffer\_empty (буфер пуст).

*Монитор «Читатели-писатели»*

Задача «Читатели-писатели» является одной из известнейших в ОС. Для этой задачи характерно наличие двух типов процессов: процессов «читателей», которые могут только читать данные, и процессов «писателей», которые могут только изменять данные. Читатели могут работать параллельно, поскольку они друг другу не мешают, а писатели могут работать только в режиме монопольного доступа: только один писатель может получить доступ к разделяемой переменной, причем, когда работает писатель, то другие писатели и читатели не могут получить доступ к этой переменной. Рассмотрим монитор Хоара «Читатели-писатели», для которого характерно наличие четырех процедур: start\_read(), stop\_read(), start\_write(), stop\_write() (листинг 3).

|  |
| --- |
| RESOURCE MONITOR;  var  active\_readers : integer;  active\_writer : logical;  can\_read, can\_write : conditional;  procedure star\_read  begin  if (active\_writer or turn(can\_write)) then  wait(can\_read);  active\_readers++; //инкремент читатетей  signal(can\_read);  end;  procedure stop\_read  begin  active\_readers--; //декремент читателей  if (active\_readers == 0) then signal(can\_write);  end;  procedure start\_write  begin  if ((active\_readers > 0) or active\_writer) then wait(can\_write);  active\_writer:= true;  end;  procedure stop\_write  begin  active\_writer = false;  if (turn(can\_read) then  signal(can\_read)  else signal(can\_write);  end;  begin  active\_readers = 0;  active\_writer = false;  end. |

Листинг 3

Когда число читателей равно 0, процесс писатель получает возможность начать работу. Новый процесс читатель не сможет начать свою работу пока работает процесс писатель и не появится истинное значение условия can\_read.

Писатель может начать свою работу, когда условие can\_write станет равно истине (true).

Когда процессу читателю нужно выполнить чтение, он вызывает процедуру start\_read. Если читатель заканчивает читать, то он вызывает процедуру stop\_read. При входе в процедуру star\_read новый процесс читатель сможет начать работать, если нет процесса писателя, изменяющего данные, в которых заинтересован читатель, и нет писателей, ждущих свою очередь (turn(can\_write)), чтобы изменить эти данные. Второе условие нужно для предотвращения бесконечного откладывания процессов писателей в очереди писателей.

Процедура start\_read завершается выдачей сигнала signal(can\_read), чтобы следующий читатель в очереди читателей смог начать чтение. Каждый следующий читатель, начав чтение выдает signal(can\_read), активизирует следующего читателя в очереди читателей. В результате возникает цепная реакция активизации читателей и она будет идти до тех пор, пока не активизируются все ожидающие читатели.

«Цепная реакция» читателей является отличительной особенностью данного решения, которое эффективно «запускает» параллельное выполнение читателей.

Процедура stop\_read уменьшает количество активных читателей: читателей, начавших чтение. После ее многократного выполнения количество читателей может стать равным нулю. Если число читателей равно нулю, выполняется signal(can\_write), активизирующий писателя из очереди писателей.

Когда писателю необходимо выполнить запить, он вызывает процедуру start\_write. Для обеспечения монопольного доступа писателя к разделяемым данным, если есть читающие процессы или другой активный писатель, то писателю придется подождать, когда будет установлено значение «истина» в переменной типа условие can\_write. Когда писатель получает возможность работать логической переменной can\_write присваивается значение «истина», что заблокирует доступ других процессов писателей к разделяемым данным.

Когда писатель заканчивает работу, предпочтение отдается читателям при условии, что очередь ждущих читателей не пуста. Иначе для писателей устанавливается переменная can\_write. Таким образом исключается бесконечное откладывание читателей.

1. Монитор – это пример множественного значения [М.186]:

   - устройство: а) видеотерминал и клавиатура как пульт управления; б) видеомонитор;

   - другое название супервизора

   - программное средство, разработанное Хоаром. [↑](#footnote-ref-1)