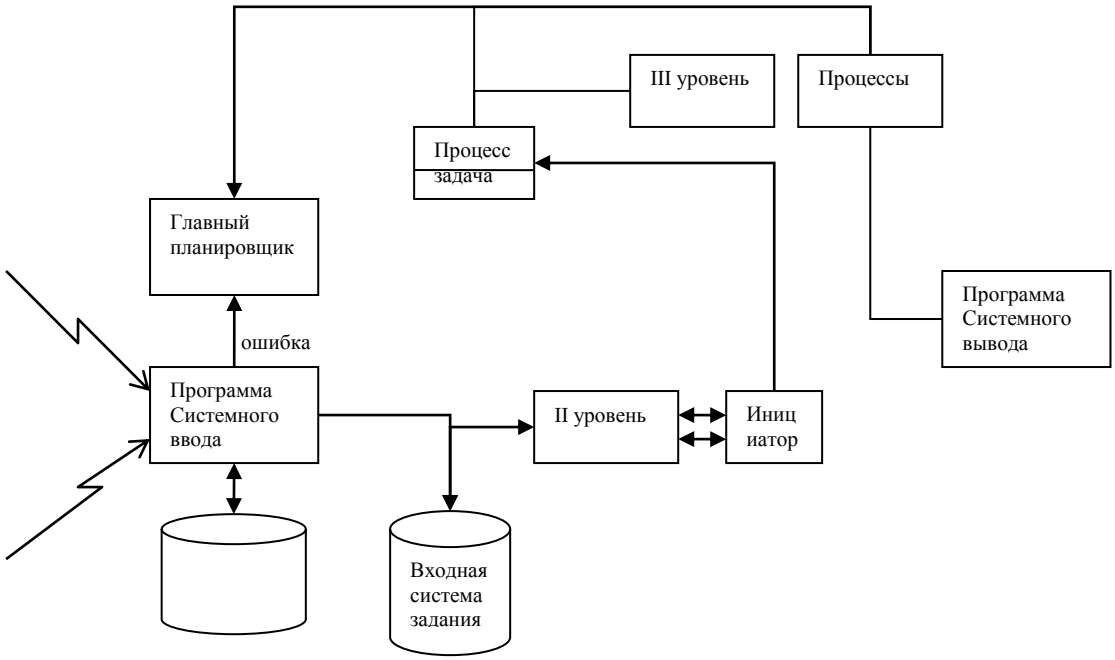
БИЛЕТ № 19

1) Ввод заданий в ОС и работа системных функций.

Задание – внешняя еденица работы системы, для которой система ресурсов не выделяет.



Программа системного ввода – планировщик 1 уровня
Командный интерпритатор берет входное задание
В том случае, если система считает, что есть ресурс для активизации задания, то активизируется планировщик 2-го уровня, то он выбирает задание с наивысшим приритетом. В том случае, если система считает, что есть ресурс для активизации задания, то активизируется планировщик. Инициатор проверяет наличие ресурсов для выполнения задания. Если ресурсов нет, то задание сбрасывается. Если всё в порядке, то образуется задача или процесс. Как только сформирован TCB система должна его обязательно выполнить. При формировании PCB определяется программа подчиненная задаче, и данные, которые должна выполнить программа. Как только освобождается процессор, всплывает планировщик 3-го уровня, который обрабатывает TCB или PSB, ищет наиболее приоритетный процесс, который можно запустить
Программа системного ввода — ридер (системное имя RDR, RDR400, RDR3200) — осуществляет прочтение и перенос любого задания с устройства системного ввода в одну из входных очередей, расположенных на системном НМД (набор данных SYS1.

После запуска она постоянно следит за устройством системного ввода, физический адрес которого оператор ЭВМ указал в соответствующей директиве.

Как только на устройстве системного ввода появляется какое-нибудь задание, ридер сразу же приступает к его обработке.

Если же на устройстве системного ввода долгое время отсутствуют какие-либо запуски, оператор ЭВМ для высвобождения объема памяти может снять программу системного ввода.

При этом вычислительный процесс не останавливается, поскольку новые задания поступают в ОП на решение из ранее заполненных системных очередей, но осуществить ввод дополнительных заданий с устройства системного ввода невозможно.

Иногда вычислительный процесс организуется таким образом, что задания в ЭВМ поступают сразу с двух устройств системного ввода (например, через перфокар-точный ввод и дисплейную станцию коллективного пользования).

В этом случае оператор запускает две программы системного ввода, каждая из которых обслуживает свое устройство.

Во время работы программа системного ввода считывает операторы входного потока; проверяет правильность написания операторов ЯУЗ; пополняет запуски заданий текстами из каталогизированных процедур; формирует в системных входных очередях специальные таблицы, характеризующие каждое задание, каждый пункт задания, все необходимые наборы данных; копирует массивы информации из входного потока на ВУ.

2) Виды связанного распределения памяти.

Управление памятью с помощью связных списков

Другой способ отслеживания состояния памяти предоставляет поддержка связных списков занятых и свободных фрагментов памяти, где сегментом является или процесс, или участок между двумя процессами. Память, показанная на рис. 4.7, *а*, представлена в виде связного списка сегментов на рис. 4.7, *в*. Каждая запись в списке указывает, является ли область памяти свободной (H, от hole — дыра) или занятой процессом (P, process); адрес, с которого начинается эта область; ее длину; содержит указатель на следующую запись.

В нашем примере список отсортирован по адресам. Такая сортировка имеет следующее преимущество: когда процесс завершается или скачивается на диск, изменение списка представляет собой несложную операцию. Закончившийся процесс обычно имеет двух соседей (кроме тех случаев, когда он находится на самом верху или на дне памяти). Соседями могут быть процессы или свободные фрагменты, что приводит к четырем комбинациям, показанным на рис. 4.8. На рис. 4.8, *а* корректировка списка требует замены P на H. На рис. 4.8, *б*, *в* две записи соединяются в одну, а список становится на запись короче. На рис. 4.8, *г* объединяются три записи, а из списка удаляются два пункта. Так как ячейка таблицы процессов для завершившегося процесса обычно будет непосредственно указывать на запись в списке для этого процесса, возможно, удобнее иметь список с двумя связями, чем с одной (последний показан на рис. 4.7, *в*). Такая структура упрощает поиск предыдущей записи и оценку возможности соединения.

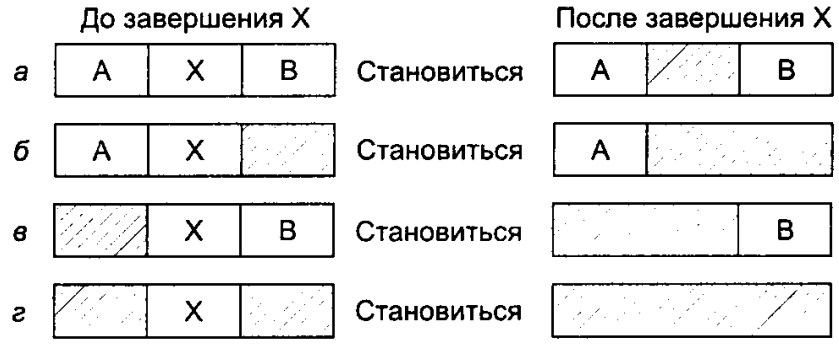


Рис. 4.8. Четыре комбинации соседей для завершения процесса X

Если процессы и свободные участки хранятся в списке, отсортированном по адресам, существует несколько алгоритмов для предоставления памяти процессу, создаваемому заново (или для существующих процессов, скачиваемых с диска). Допустим, менеджер памяти знает, сколько памяти нужно предоставить. Простейший алгоритм представляет собой выбор **первого подходящего участка**. Менеджер памяти просматривает список областей до тех пор, пока не находит достаточно большой свободный участок. Затем этот участок делится на две части: одна отдается процессу, а другая остается неиспользуемой. Так происходит всегда, кроме статистически нереального случая точного соответствия свободного участка и процесса. Это быстрый алгоритм, потому что поиск уменьшен настолько, насколько возможно.

Алгоритм **«следующий подходящий участок»** действует с минимальными отличиями от правила «первый подходящий». Он работает так же, как и первый алгоритм, но всякий раз, когда находит соответствующий свободный фрагмент, он запоминает его адрес. И когда алгоритм в следующий раз вызывается для поиска, он стартует с того самого места, где остановился в прошлый раз вместо того, чтобы каждый раз начинать поиск с начала списка, как это делает алгоритм «первый подходящий». Моделирование работы алгоритма, произведенное Бэйсом, показало, что производительность схемы «следующий подходящий» немного хуже, чем «первый подходящий» [21].

Другой хорошо известный алгоритм называется **«самый подходящий участок»**. Он выполняет поиск по всему списку и выбирает наименьший по размеру подходящий свободный фрагмент. Вместо того чтобы делить большую незанятую область, которая может понадобиться позже, этот алгоритм пытается найти участок, близко подходящий к действительно необходимым размерам.

Чтобы привести пример работы алгоритмов «первый подходящий» и «самый подходящий», снова обратимся к рис. 4.7. Если необходим блок размером 2, правило «первый подходящий» предоставит область по адресу 5, а схема «самый подходящий» разместит процесс в свободном фрагменте по адресу 18.

Алгоритм «самый подходящий» медленнее «первого подходящего», потому что каждый раз он должен производить поиск во всем списке. Но, что немного удивительно, он выдает еще более плохие результаты, чем «первый подходящий» или «следующий подходящий», поскольку стремится заполнить память очень маленькими, бесполезными свободными областями, то есть фрагментирует память. Алгоритм «первый подходящий» в среднем создает большие свободные участки.

Пытаясь решить проблему разделения памяти на практически точно совпадающие с процессом области и маленькие свободные фрагменты, можно задуматься об алгоритме **«самый неподходящий участок»**. Он всегда выбирает самый большой свободный участок, от которого после разделения остается область достаточного размера и ее можно использовать в дальнейшем. Однако моделирование показало, что это также не очень хорошая идея.

Все четыре алгоритма можно ускорить, если поддерживать отдельные списки для процессов и свободных областей. Тогда поиск будет производиться только среди незанятых фрагментов. Неизбежная цена, которую нужно заплатить за увеличение скорости при размещении процесса в памяти, заключается в дополнительной сложности и замедлении при освобождении областей памяти, так как ставший свободным фрагмент необходимо удалить из списка процессов и вставить в список незанятых участков.

Если для процессов и свободных фрагментов поддерживаются отдельные списки, то последний можно отсортировать по размеру, тогда алгоритм «самый подходящий» будет работать быстрее. Когда он выполняет поиск в списке свободных фрагментов от самого маленького к самому большому, то, как только находит подходящую незанятую область, алгоритм уже знает, что она — наименьшая из тех, в которых может поместиться задание, то есть наилучшая. В отличие от схемы с одним списком, дальнейший поиск не требуется. Таким образом, если список свободных фрагментов отсортирован по размеру, схемы «первый подходящий» и «самый подходящий» одинаково быстры, а алгоритм «следующий подходящий» не имеет смысла.

При поддержке отдельных списков для процессов и свободных фрагментов возможна небольшая оптимизация. Вместо создания отдельного набора структур данных для списка свободных участков, как это сделано на рис. 4.7, *в*, можно использовать сами свободные области. Первое слово каждого незанятого фрагмента может содержать размер фрагмента, а второе слово может указывать на следующую запись. Узлы списка на рис. 4.7, *в*, для которых требовались три слова и один бит (Р/Н), больше не нужны.

Еще один алгоритм распределения называется **«быстрый подходящий»**, он поддерживает отдельные списки для некоторых из наиболее часто запрашиваемых размеров. Например, могла бы существовать таблица с *n* записями, в которой первая запись указывает на начало списка свободных фрагментов размером 4 Кбайт, вторая запись является указателем на список незанятых областей размером 8 Кбайт, третья — 12 Кбайт и т. д. Свободный фрагмент размером, скажем, 21 байт, мог бы располагаться или в списке областей 20 Кбайт или в специальном списке участков дополнительных размеров. При использовании правила «быстрый подходящий» поиск фрагмента требуемого размера происходит чрезвычайно быстро. Но этот алгоритм имеет тот же самый недостаток, что и все схемы, которые сортируют свободные области по размеру, а именно: если процесс завершается или выгружается на диск, поиск его соседей с целью узнать, возможно ли их соединение, является дорогой операцией. А если не производить слияния областей, память очень скоро окажется разбитой на огромное число маленьких свободных фрагментов, в которые не поместится ни один процесс.

3) Особенности хранения пустого дискового пространства в различных ОС.

1. Битовая карта – минимизирует объем информации по кластерам. Обычно битовая карта привязывается к непосредственному указанию кластеров, занятых файлом
2. FAT – на каждый кластер выделено поле для указания следующего кластера
3. В определенном месте файловой системы выделена зона для хранения некоторого числа свободных кластеров.

4) Особенности ОС вычислительных сетей.

Ответ:

Доп.инфа:

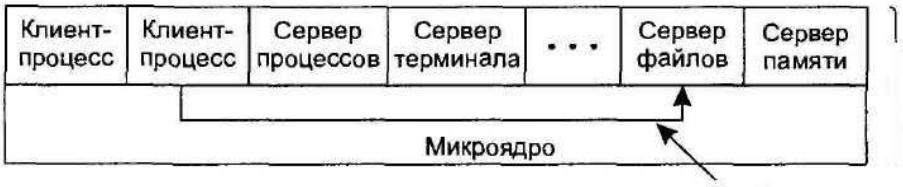
В сетевой операционной системе пользователи знают о существовании многочисленных компьютеров, могут регистрироваться на удаленных машинах и копировать файлы с одной машины на другую. Каждый компьютер работает под управлением локальной операционной системы и имеет своего собственного локального пользователя (или пользователей).

Сетевые операционные системы несущественно отличаются от однопроцессорных операционных систем. Ясно, что они нуждаются в сетевом интерфейсном контроллере и специальном низкоуровневом программном обеспечении, поддерживающем работу контроллера, а также в программах, разрешающих пользователям удаленную регистрацию в системе и доступ к удаленным файлам. Но эти дополнения, по сути, не изменяют структуры операционной системы.

Сетевая ОС поддерживает клиент-серверные технологии.

В развитии современных операционных систем наблюдается тенденция в сторону дальнейшего переноса кода в верхние уровни и удалении при этом всего, что только возможно, из режима ядра, оставляя минимальное **микроядро.** Обычно это осуществляется перекладыванием выполнения большинства задач операционной системы на средства пользовательских процессов. Получая запрос на какую-либо операцию, например чтение блока файла, пользовательский процесс (теперь называемый **обслуживаемым процессом** или **клиентским процессом)** посылает запрос **серверному** (обслуживающему) **процессу,** который его обрабатывает и высылает назад ответ.

В модели, показанной в задачу ядра входит только управление связью между клиентами и серверами. Благодаря разделению операционной системы на части, каждая из которых управляет всего одним элементом системы (файловой системой, процессами, терминалом или памятью), все части становятся маленькими и управляемыми. К тому же, поскольку все серверы работают как процессы в режиме пользователя, а не в режиме ядра, они не имеют прямого доступа к оборудованию. Поэтому если происходит ошибка на файловом сервере, может разрушиться служба обработки файловых запросов, но это обычно не приводит к остановке всей машины целиком.



Другое преимущество модели клиент-сервер заключается в ее простой адаптации к использованию в распределенных системах. Если клиент общается с сервером, посылая ему сообщения, клиенту не нужно знать, обрабатывается ли его сообщение локально на его собственной машине или оно было послано по сети серверу на удаленной машине. С точки зрения клиента происходит одно и то же в обоих случаях: запрос был послан, и на него получен ответ

5 Состав и функции блока обработки процессов.

Шпора(книга Симона):

Количество PCB определяется количеством выполняемый в данный момент процессов.

Блок управления процессом (PCB — process control block)

Выполнение функций ОС, связанных с управлением процессами, осуществляется с помощью **блока управления процессом** (PCB). **Вход в процесс** (фиксация системой процесса) — это создание его блока управления (PCB), а **выход из процесса** — это его уничтожение, т. е. уничтожение его блока управления.

Таким образом для каждого активизированного процесса система создает PCB, в котором в сжатом виде содержится информация о процессе, используемая при управлении. PCB — это системная структура данных, содержащая определенные сведения о процессе и имеющая следующие поля:

1. Уникальный идентификатор процесса (имя)
2. Текущее состояние процесса.
3. Приоритет процесса.
4. Указатели участка памяти выделенного программе, подчиненной данному процессу.
5. Указатели выделенных ему ресурсов.
6. Область сохранения регистров.
7. Права процесса (список разрешенных операций)
8. Связи зависимости в иерархии процессов (список дочерних процессов, имя родительского процесса)
9. Пусковой адрес программы, подчиненной данному процессу.

Когда ОС переключает процессор с процесса на процесс, она использует области сохранения регистров в PCB для запоминания информации, необходимой для рестарта (повторного запуска) каждого процесса с точки прерывания, когда он в следующий раз получит в свое распоряжение процессор. Количество процессов в системе ограничено и определяется самой системой, пользователем во время генерации ОС или при загрузке. Неудачное определение количества одновременно исполняемых программ может привести к снижению полезной эффективности работы системы, т.к. переключение процессов требует выполнения дополнительных операций по сохранению и восстановлению состояния процессов. Блоки управления системных процессов создаются при загрузке системы. Это необходимо, чтобы система выполняла свои функции достаточно быстро, и время реакции ОС было минимальным. Однако, количество блоков управления системными процессами меньше, чем количество самих системных процессов. Это связано с тем, что структура ОС имеет либо оверлейную, либо динамически — последовательную структуру иерархического типа, и нет необходимости создавать для программ, которые никогда не будут находиться одновременно в оперативной памяти, отдельные PCB. При такой организации легко учитывать приоритеты системных процессов, выстроив их по приоритетам заранее при инициализации системы. Блоки управления проблемными (пользовательскими) процессами создаются в процессе активизации процессов динамически. Все PCB находятся в выделенной системной области памяти.

В каждом PCB есть поле состояния процесса. Все блоки управления системными процессами располагаются в порядке убывания приоритетов и находятся в системной области памяти. Если приоритеты системных блоков можно определить заранее, то для проблемных процессов необходима таблица приоритетов проблемных программ. Каждый блок PCB имеет стандартную структуру, фиксированный размер, точку входа, содержит указанную выше информацию и дополнительную информацию для синхронизации процессов. Для синхронизации в PCB имеются четыре поля:

1-2. Поля для организации цепочки связи.

3-4. Поля для организации цепочки ожидания.

В цепочке связи указывается адрес PCB вызываемого (поле 1) и вызывающего (поле 2) процесса.

В цепочке ожидания, в поле 3 указывается адрес PCB вызываемого процесса, если вызываемый процесс занят. В поле 4 занятого процесса находится число процессов, которые ожидают данный.

Если процесс A пытается вызвать процесс B, а у процесса B в PCB занята цепочка связей, то есть он является вызываемым по отношению к другим процессам, тогда адрес процесса B записывается в цепочке ожидания PCB процесса A, а в поле счетчика ожидания PCB процесса B добавляется 1. Как только процесс B завершает выполнение своих функций, он передает управление вызывающему процессу следующим образом: B проверяет состояние своего счетчика ожидания, и, если счетчик больше 0, то среди PCB других процессов ищется первый (по приоритету или другим признакам) процесс, в поле 3 PCB которого стоит имя ожидаемого процесса, в данном случае B, тогда управление передается этому процессу.