**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра «Информационные системы»**

Реферат № 2

**по дисциплине «Системы реального времени»**

Тема: Диспетчеризация задач

Дата представления реферата на проверку: 17.02.2023

|  |  |  |
| --- | --- | --- |
| Студентгр. 8362 |  | Ларионова Е.Е.  ekaterinalarionova25@gmail.com |
| Преподаватель |  | Сидельников В.В. |

Санкт-Петербург

2023

**содержание**

|  |
| --- |
|  |
| 1. | Задачи – процессы и потоки   * 1. Процессы, потоки. задачи   2. Основные свойства задач |  |
| 2. | Режим разделения времени |  |
| 3. | Диспетчеризация задач |  |
| 4. | Состояния задач   * 1. Основные состояния задач   2. Переходные состояния задач |  |
| 5 | Очереди задач |  |
| 6  7 | Приоритетное управлени  Статические и динамические алгоритмы |  |
|  | Заключение |  |
|  | Список использованных источников |  |
|  |  |  |

1. **ЗАДАЧИ – ПРОЦЕССЫ И ПОТОКИ**
   1. **Процессы, потоки, задачи**

Существуют различные определения термина «задача» для многозадачной ОС РВ. Мы будем считать задачей набор операций (машинных инструкций), предназначенный для выполнения логически законченной функции системы. При этом задача конкурирует с другими задачами за получение контроля над ресурсами вычислительной системы.

Принято различать две разновидности задач: ***процессы и потоки***.

***Процесс*** представляет собой «среду обитания» запущенной программы. Т.е. при запуске программы ОС создает процесс, выделяя программе все необходимые ей (для нормального функционирования) ресурсы.

***Поток*** – это часть кода программы, которая непосредственно исполняется на процессоре в данный момент. Поэтому потоки могут использовать общие ресурсы одного процесса.

Хорошим примером многопоточной программы является редактор текста MS Word, где в рамках одного приложения может одновременно происходить и набор текста, и проверки правописания.

**Преимущества потоков.**

1. Так как множество потоков способно размещаться внутри одного EXE-модуля, это позволяет экономить ресурсы как внешней, так и внутренней памяти.

2. Использование потоками общей области памяти позволяет эффективно организовать межзадачный обмен сообщениями (достаточно передать указатель на сообщение). Процессы не имеют общей области памяти. Поэтому ОС должна либо целиком скопировать сообщение из области памяти одной задачи в область памяти другой (что для больших сообщений весьма накладно), либо предусмотреть специальные механизмы, которые позволили бы одной задаче получить доступ к сообщению из области памяти другой задачи.

3. Как правило, контекст потоков меньше, чем контекст процессов, а значит, время переключения между задачамипотоками меньше, чем между задачами-процессами.

4. Так как все потоки, а иногда и само ядро РВ размещаются в одном ЕХЕ-модуле, значительно упрощается использование программ-отладчиков (debugger).

**Недостатки потоков.**

1. Как правило, потоки не могут быть подгружены динамически. Чтобы добавить новый поток, необходимо провести соответствующие изменения в исходных текстах и перекомпилировать приложение. Процессы, в отличие от потоков, подгружаемы, что позволяет динамически изменять функции системы в процессе ее работы. Кроме того, так как процессам соответствуют отдельные программные модули, они могут быть разработаны различными компаниями, чем достигается дополнительная гибкость и возможность использования ранее наработанного ПО.

2. То, что потоки имеют доступ к областям данных друг друга, может привести к ситуации, когда некорректно работающий поток способен испортить данные другого потока. В отличие от этого процессы защищены от взаимного влияния, а попытка записи в «не свою» память приводит, как правило, к возникновению специального прерывания по обработке «исключительных ситуаций».

Реализация механизмов управления процессами и потоками, возможность их взаимного сосуществования и взаимодействия определяются конкретным ПО РВ.

* 1. **Основные свойства задач**

Как правило, вся важная, с точки зрения операционной системы, информация о задаче хранится в унифицированной структуре данных – управляющем блоке (Task Control Block, TCB).

В блоке хранятся такие параметры, как имя и номер задачи, верхняя и нижняя границы стека, ссылка на очередь сообщений, статус задачи, приоритет и т. п.

***Приоритет*** – это некое целое число, присваиваемое задаче и характеризующее ее важность по сравнению с другими задачами, выполняемыми в системе. Приоритет используется в основном планировщиком задач для определения того, какая из готовых к работе задач должна получить управление. Различают системы с динамической и статической приоритетностью. В первом случае приоритет задач может меняться в процессе исполнения, в то время как во втором приоритет задач жестко задается на этапе разработки или во время начального конфигурирования системы.

***Контекст задачи*** – это набор данных, содержащий всю необходимую информацию для возобновления выполнения задачи с того места, где она была ранее прервана. Часто контекст хранится в TCB и включает в себя такие данные, как счетчик команд, указатель стека, регистры CPU и FPU и т. п. Планировщик задач в случае необходимости сохраняет контекст текущей активной задачи и восстанавливает контекст задачи, назначенной к исполнению. Такое переключение контекстов и является, по сути, основным механизмом ОС РВ при переходе от выполнения одной задачи к выполнению другой.

1. **РЕЖИМ РАЗДЕЛЕНИЯ ВРЕМЕНИ**

Существуют различные реализации в рамках этого алгоритма, и некоторые западные специалисты даже различают такие, в общем-то идентичные для нас понятия, как ***time-slicing*** (нарезание времени) и ***time-sharing*** (разделение времени). Как правило, алгоритм реализуется следующим образом: каждой задаче отводится определенное количество квантов времени (обычно кратно 1 мс), в течение которых задача может монопольно занимать процессорное время. После того как заданный интервал времени истекает, управление передается следующей готовой к выполнению задаче, имеющей наивысший приоритет. Та, в свою очередь, выполняется в течение отведенного для нее промежутка времени, после чего все повторяется в стиле round robin. Легко заметить, что такой алгоритм работы может привести к определенным проблемам. Представим, что в системе работают 7 задач, 3 из которых имеют высокий приоритет, а 4 – низкий. Низкоприоритетные задачи могут никогда не получить управление, так как три высокоприоритетные задачи будут делить все процессорное время между собой. Единственную возможность для низкоприоритетных задач получить управление предоставляет ситуация, когда все высокоприоритетные задачи находятся в блокированном состоянии.

Для решения этой проблемы применяется прием, получивший название равнодоступность (fairness). При этом реализуется принцип адаптивной приоритетности, когда приоритет задачи, которая выполняется слишком долго, постепенно уменьшается, позволяя менее приоритетным задачам получить свою долю процессорного времени. Равнодоступность применяется главным образом в многопользовательских системах и редко применяется в системах реального времени.

***Кооперативная многозадачность*** – это еще один алгоритм переключения задач, с которым широкие массы компьютерной общественности знакомы по oперационной системе Windows. Задача, получившая управление, выполняется до тех пор, пока она сама по своей инициативе не передаст управление другой задаче. По сути это продолжение идеологии round robin, и нет нужды объяснять, почему алгоритм кооперативной многозадачности в чистом виде мало применяется в системах реального времени.

***Приоритетная многозадачность с вытеснением*** – это наиболее часто используемый в ОС РВ принцип планирования. Основная идея состоит в том, что высокоприоритетная задача, как только для нее появляется работа, немедленно прерывает (вытесняет) низкоприоритетную. Другими словами, если какая-либо задача переходит в состояние готовности, она немедленно получает управление, если текущая активная задача имеет более низкий приоритет. Такое «вытеснение» происходит, например, когда высокоприоритетная задача получила ожидаемое сообщение, освободился запрошенный ею ресурс, произошло связанное с ней внешнее событие, исчерпался заданный интервал времени и т. п.

1. **ДИСПЕТЧЕРИЗАЦИЯ ЗАДАЧ**

*Планирование* — это работа по определению того, в какой момент необходимо прервать выполнение активного текущего потока и какому потоку предоставить возможность выполняться. Системная программа, выбирающая один из готовых для исполнения потоков, называется *планировщиком.*

*Диспетчеризация* заключается в реализации найденного в результате планирования решения, то есть переключении процессора с одного потока на другой. Механизм диспетчеризации непосредственно влияет на минимальное время реакции управляющей системы на внешние события.

Важной частью любой ОС РВ является планировщик задач. Несмотря на то, что в разных источниках он может называться по-разному (диспетчер задач, супервизор и т. п.), его функции остаются теми же: определить, какая из задач должна выполняться в системе в каждый конкретный момент времени. Самым простым методом планирования, не требующим никакого специального ПО и планировщика как такового, является использование циклического алгоритма в стиле round robin.

Каждая «задача», представляющая собой отдельную подпрограмму, выполняется циклически. При этом надо придерживаться следующих правил:

1. Подпрограммы не должны содержать циклов ожидания

2. Подпрограммы должны выполнять свою работу как можно быстрее, чтобы дать возможность работать следующей подпрограмме.

3. При необходимости подпрограмма может сохранять свое окружение и текущие результаты, чтобы в следующем цикле возобновить работу с того же места.

Можно отметить следующие преимущества циклического алгоритма.

1. Простота использования и прозрачность для понимания.

2. Если исключить из рассмотрения прерывания, система полностью детерминирована. Задачи всегда вызываются в одной и той же последовательности, что позволяет достаточно просто произвести анализ «наихудшего случая» и вычислить максимальную задержку.

3. Минимальные размеры кода и данных. Кроме того, в отличие от алгоритмов с вытеснением, для всех задач необходим только один стек.

4. Отсутствуют ошибки, обусловленные «гонками».

К недостаткам циклического алгоритма можно отнести отсутствие приоритетности и очередей. К тому же задачи вызываются независимо от того, должны ли они в данный момент что-либо делать или нет, а на прикладного программиста ложится максимальная ответственность за работоспособность системы.

1. **СОСТОЯНИЯ ЗАДАЧ**
   1. **Основные состояния задач**

С точки зрения операционной системы, задача может находиться в нескольких состояниях. Число и название этих состояний различаются от одной ОС к другой. Тем не менее практически в любой ОС РВ загруженная на выполнение задача может находиться, по крайней мере, в трех состояниях.

1. *Активная задача* – это задача, выполняемая системой в текущий момент времени.

2. *Готовая задача* – это задача, готовая к выполнению и ожидающая у планировщика своей «очереди».

3. *Блокированная задача* – это задача, выполнение которой приостановлено до наступления определенных событий. Такими событиями могут быть освобождение необходимого задаче ресурса, поступление ожидаемого сообщения, завершение интервала ожидания и т. п.

***Пустая задача (Idle Task)*** – это задача, запускаемая самой операционной системой в момент инициализации и выполняемая только тогда, когда в системе нет других готовых для выполнения задач. Пустая задача запускается с самым низким приоритетом и, как правило, представляет собой бесконечный цикл «ничего не делать». Наличие пустой задачи предоставляет операционной системе удобный механизм отработки ситуаций, когда нет ни одной готовой к выполнению задачи.

***Многократный запуск задач.*** Как правило, многозадачные ОС позволяют запускать несколько копий одной и той же задачи. При этом для каждой такой копии создается свой TCB и выделяется своя область памяти. В целях экономии памяти может быть предусмотрено совместное использование одного и того же исполняемого кода для всех запущенных копий. В этом случае программа должна обеспечивать повторную входимость (реентерабельность). Кроме того, программа не должна использовать временные файлы с фиксированными именами и должна корректно осуществлять доступ к глобальным ресурсам.

***Реентерабельность*** (повторная входимость) означает возможность без негативных последствий временно прервать выполнение какой-либо функции или подпрограммы, а затем вызвать эту функцию или подпрограмму снова. Частным проявлением реентерабельности является рекурсия, когда тело подпрограммы содержит вызов самой себе. Классическим примером нереентерабельной системы является DOS, a типичной причиной нереентерабельности служит использование глобальных переменных. Предположим, что у нас есть функция, реализующая низкоуровневую запись на диск, и пусть она использует глобальную переменную write\_sector, которая устанавливается в соответствии с параметром, передаваемым этой функции при вызове. Предположим теперь, что Задача А вызывает эту функцию с параметром 3, то есть хочет записать данные в сектор номер 3. Допустим, что когда переменная write\_sector уже равна 3, но сама запись еще не произведена, выполнение Задачи А прерывается и начинает выполняться Задача В, которая взывает ту же функцию, но с аргументом 10. После того как запись в сектор номер 10 будет произведена, управление рано или поздно вернется к Задаче А, которая продолжит работу с того же места. Однако, так как переменная write\_sector имеет теперь значение 10, данные Задачи А, предназначавшиеся для сектора номер 3, будут вместо этого записаны в сектор номер 10. Из приведенного примера видно, что ошибки, связанные с нереентерабельностью, трудно обнаружить, а последствия они могут вызвать самые катастрофические.

* 1. **Переходные состояния**

***Процесс*** – это динамическая сущность программы, ее код в процессе своего выполнения. Имеет:

* собственные области памяти под код и данные, включая значения регистров и счетчика команд;
* собственный стек;
* собственное отображение виртуальной памяти (в системах с виртуальной памятью) на физическую;
* собственное состояние. Процесс может находиться в одном из следующих типичных состояний:
* «остановлен» – процесс остановлен и не использует процессор (например, в таком состоянии процесс находится сразу после создания);
* «терминирован» – процесс терминирован и не использует процессор (например, процесс закончился, но еще не удален операционной системой);
* «ждет» – процесс ждет некоторого события (им может быть аппаратное или программное прерывание, сигнал или другая форма межпроцессного взаимодействия);
* «готов» – процесс не остановлен, не терминирован, не ожидает, не удален, но и не работает (например, процесс не может получить доступ к процессору, если в данный момент выполняется другой, более высокоприоритетный процесс);
* «выполняется» – процесс выполняется и использует процессор. В ОСРВ это обычно означает, что этот процесс является самым приоритетным среди всех процессов, находящихся в состоянии «готов».

1. **ОЧЕРЕДИ ЗАДАЧ**

Очередь процессов, готовых к исполнению, необязательно устроена по принципу «первым пришел — первым ушел» (FIFO). Помимо FIFO, очередь готовых процессов может быть реализована в виде очереди с приоритетами, дерева или просто связанного списка. Однако, при любой реализации очереди все процессы в ней упорядочены и находятся в ожидании возможности выполниться на процессоре.

Задача планирования выполнения процессов может проявляться в следующих ситуациях:

1. процесс переходит из состояния исполнения в состояние ожидания (например, в случае запроса ввода/вывода, или ожидания завершения одного из дочерних процессов);

2. процесс переходит из состояния исполнения в состояние готовности (например, при возникновении прерывания);

3. процесс переходит из состояния ожидания в состояние готовности (например, завершение ввода/вывода);

4. процесс завершается.

**6. ПРИОРИТЕТНОЕ УПРАВЛЕНИЕ**

***Алгоритм SJF*** является частным случаем более общего алгоритма планирования с приоритетами (priority-scheduling algorithm). С каждым процессом ассоциируется некоторый приоритет, и процессор выделяется процессу с наивысшим приоритетом. Процессы с одинаковыми приоритетами обслуживаются по схеме FCFS.

Алгоритм SJF можно представить как разновидность алгоритма планирования с приоритетом, где приоритет (p) задается как величина, обратная длине следующего (предсказанного) участка счета. Чем длиннее интервал счета, тем ниже приоритет, и наоборот.

Приоритет обычно представляет собой число из фиксированного диапазона, например от 0 до 7 или от 0 до 4095. Однако нет общего соглашения по поводу того, соответствует ли 0 самому низкому или самому высокому приоритету. На некоторых системах меньшие значения используются для представления низкого приоритета, на других — для высокого.

Приоритеты могут быть *внутренними* или *внешними*. Внутренне задаваемые приоритеты используют для вычисления приоритета процесса некоторые измеримые величины. Например, при вычислении приоритета могут использоваться ограничения по времени, требования к памяти, количество открытых файлов и соотношение средней продолжительности участка ввода/вывода к средней продолжительности участка счета. Внешние приоритеты задаются критериями, внешними по отношению к операционной системе, такими как важность процесса, количество денег, уплаченных за пользование компьютером, отдел, спонсирующий работу, и другие факторы.

Планирование с приоритетами может быть как *вытесняющим*, так и *невытесняющим*. Когда процесс поступает в очередь готовых процессов, его приоритет сравнивается с приоритетом текущего выполняемого процесса. Вытесняющий вариант алгоритма передаст процессор поступившему процессу, если его приоритет будет выше, чем у исполняемого процесса. Невытесняющий вариант просто поместит новый процесс в начало очереди.

Основной проблемой алгоритмов с приоритетами является возможность неопределенно долгой блокировки (indefinite blocking) или дискриминации (starvation). Процесс, готовый к исполнению, но не получивший процессор, остается заблокированным. Алгоритм планирования с приоритетами может заставить низкоприоритетный процесс ждать процессора неопределенно долго. В сильно нагруженной вычислительной системе постоянный поток высокоприоритетных задач может не дать низкоприоритетной задаче получить время процессора. Обычно происходит одно из двух: или процесс будет когда-нибудь запущен (в 2 часа дня в воскресенье, когда система наконец-то будет слабо нагружена), или вычислительная система через некоторое время выдаст сбой, и все незаконченные низкоприоритетные процессы будут потеряны.

Решением проблемы неопределенно долгой блокировки является механизм старения — постепенное повышение приоритета процесса, который ждет в системе долгое время. Например, если приоритеты изменяются в диапазоне от 127 (низкий) до 0 (высокий), то можно уменьшать приоритет ожидающего процесса на единицу каждые 15 минут. По прошествии какого-то времени даже процесс с начальным приоритетом 127 будет иметь самый высокий приоритет в системе и будет запущен. В указанном случае, процессу с приоритетом 127 потребуется не более 32 часов, чтобы его приоритет дошел до 0.

**7. СТАТИЧЕСКИЕ И ДИНАМИЧЕСКИЕ АЛГОРИТМЫ**

Алгоритмы планирования заданий могут быть разделены на *статические* и *динамические*. Статические алгоритмы определяют приемлемый план выполнения заданий по их априорным характеристикам, динамический алгоритм модифицирует план во время исполнения заданий. Издержки на статическое планирование низки, но оно крайне нечувствительно и требует полной предсказуемости той системы реального времени, на которой оно установлено. Динамическое планирование связано с большими издержками, но способно адаптироваться к меняющемуся окружению.

Алгоритмы планирования будем рассматривать на примере 3- х периодических процессов A, B, C. Предположим, что процесс A запускается с периодом 30 мс и временем обработки 10 мс. Процесс B имеет период 40 мс и время обработки 15 мс. Процесс C запускается каждые 50 мс и обрабатывается за 5 мс. Суммарно эти процессы потребляют 0,808 процессорного времени, что меньше единицы. Соответственно, система в данном примере поддается планированию.

На рис. 1 представлена временная диаграмма работы процессов. Видно, что необходимо применить некоторый алгоритм планирования, так как в определенные моменты времени имеется сразу несколько готовых к выполнению процессов.

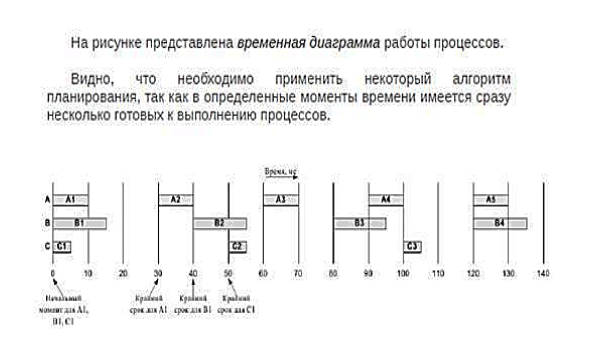


Рисунок 1 – Три периодических процесса с разным периодом и временем обработки

**Статический алгоритм планирования RMS**

Классическим примером статического алгоритма планирования реального времени для прерываемых периодических процессов является алгоритм **RMS** (Rate Monotonic Scheduling – планирование с приоритетом, пропорциональным частоте).

Этот алгоритм может использоваться для процессов, удовлетворяющих следующим условиям:

1. Каждый периодический процесс должен быть завершен за время его периода.

2. Ни один процесс не должен зависеть от любого другого процесса.

3. Каждому процессу требуется одинаковое процессорное время на каждом интервале.

4. У непериодических процессов нет жестких сроков.

5. Прерывание процесса происходит мгновенно, без накладных расходов.

**Динамический алгоритм планирования EDF**

Другим популярным алгоритмом планирования является алгоритм **EDF** (Earliest Deadline First – процесс с ближайшим сроком завершения в первую очередь). Алгоритм EDF представляет собой динамический алгоритм, не требующий от процессов периодичности. Он также не требует и постоянства временных интервалов использования процессора. Каждый раз, когда процессу требуется процессорное время, он объявляет о своем присутствии и о своем сроке выполнения задания. Планировщик хранит список процессов, сортированный по срокам выполнения заданий. Алгоритм запускает первый процесс в списке, то есть тот, у которого самый близкий по времени срок выполнения. Когда новый процесс переходит в состояние готовности, система сравнивает его срок выполнения со сроком выполнения текущего процесса. Если у нового процесса график более жесткий, он прерывает работу текущего процесса.

**ЗАКЛЮЧЕНИЕ**

Концепция многозадачности (псевдопараллелизм) является существенной для системы реального времени с одним процессором, приложения которой должны быть способны обрабатывать многочисленные внешние события, происходящие практически одновременно. Концепция процесса, пришедшая из мира UNIX, плохо реализуется в многозадачной системе, поскольку процесс имеет тяжелый контекст.

В однопроцессорных системах потоки конкурируют за ресурсы процессора. В этом случае они выполняются фактически квазипараллельно и поэтому взаимно влияют друг на друга, порождая проблему эффективного планирования. Механизмы планирования потоков вместе с системой приоритетов устанавливают правила и условия переключения потоков.

**СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

1. Системы реального времени : конспект лекций / Владим. гос. ун-т ; сост. А. С. Голубев. – Владимир : Изд-во Владим. гос. ун-та, 2010. – 127 с.
2. Системы реального времени: учебное пособие / А. С. Луканов. – Самара: Издательство Самарского университета, 2020. – 156 с.: ил.
3. Системы реального времени: обзорный курс лекций/ К.Е. Климентьев. - Самара: Самар. гос. аэрокосм. ун-т. Самара, 2008. – 45 с.