Управление процессами (вычисления)

Процесс - это программа в исполнении и неотъемлемая часть любой современной операционной системы (ОС). ОС должна выделять ресурсы на процессы, обеспечивать процесс обмена информацией и обмениваться ими, защищать ресурсы каждого процесса от других процессов и обеспечивать синхронизацию между процессами. Чтобы соответствовать этим требованиям, ОС должна поддерживать структуру данных для каждого процесса, которая описывает государство и владение ресурсами этого процесса и которая позволяет операционной системе осуществлять контроль над каждым процессом.

Мультипрограммирование

В любой современной операционной системе может быть более одного экземпляра программы, загруженной в память одновременно. Например, более одного пользователя могут выполнять одну и ту же программу, причем каждый пользователь имеет отдельные копии программы, загруженные в память. С некоторыми программами можно загрузить один экземпляр в память, в то время как несколько пользователей имеют общий доступ к ней, чтобы каждый из них мог выполнять один и тот же программный код. Такая программа называется re-entrant [В данный момент процессор может выполнять только одну инструкцию из одной программы, но несколько процессов могут поддерживаться в течение определенного периода времени, назначая каждый процесс процессору с интервалами, а остальная часть становится временно неактивной. Выполнение нескольких процессов в течение определенного периода времени, а не одновременно, называется параллельным исполнением.

Многопрограммная или многозадачная ОС - это система, которая может выполнять множество процессов одновременно. Многопрограммирование требует, чтобы процессор был распределен для каждого процесса в течение определенного периода времени и де-распределен или выдан в соответствующий момент. Если процессор дераспределен во время выполнения процесса, это должно быть сделано таким образом, чтобы процесс мог перезапуститься позже как можно эффективнее.

Существует два способа для ОС восстановить контроль над процессором во время выполнения программы, чтобы ОС могла выполнить де-распределение или распределение:

1. Процесс выдает системный вызов (иногда называемый *прерыванием программного обеспечения*); например, возникает запрос ввода/вывода, запрашивающего доступ к файлу на жестком диске.
2. Происходит аппаратный прерывание; например, на клавиатуре нажимали клавишу, или таймер заканчивался (используется в упреждающей

многозадачности).

Остановка одного процесса и начало (или перезапуск) другого процесса называется переключением контекста или изменением контекста. Во многих современных операционных системах процессы могут состоять из множества подпроцессов. Это вводит понятие *нити*. Нитка может рассматриваться как *подпроцесс,* то есть отдельная, независимая последовательность выполнения в коде одного процесса. Темы становятся все более важными в проектировании распределенных и клиент-серверных систем и в программном обеспечении, работающем на многопроцессорных системах.

Как многопрограммирование повышает эффективность

Общей чертой, наблюдаемой среди процессов, связанных с большинством компьютерных программ, является то, что они чередуются между циклами процессора и циклами ввода-вывода. В течение части времени, необходимого для циклов процессора, этот процесс выполняется и занимает ЦП. В течение времени, необходимого для циклов ввода/вывода, процесс не использует процессор. Вместо этого он либо ждет выполнения Input/Output, либо фактически выполняет Input/Output. Примером этого является чтение или запись файла на диске. До появления многопрограммных программ компьютеры работали как однопользовательские системы. Пользователи таких систем быстро осознают, что в течение большей части времени, когда компьютер был выделен одному пользователю - например, когда пользователь вводил информацию или отладив программы - процессор простаивал. Компьютерные ученые заметили, что общая производительность машины может быть улучшена, позволяя другому процессу использовать процессор всякий раз, когда один процесс ожидает ввода/вывода. В системе *одноразового программирования,* если *N*N пользователи должны были выполнять программы с индивидуальным временем выполнения $t_1$1, $t_2$2, ..., m_N, то общее время, tu_{uni}uni, для обслуживания *N-процессов* (последовательно) всех *пользователей N было* бы:

$$tu_{uni} = t_{t\,1} + t_2 + \ldots + t_N.$$

Однако, поскольку каждый процесс потребляет как циклы процессора, так и циклы ввода/вывода, время, в течение которого каждый процесс фактически использует ЦП, составляет очень небольшую долю от общего времени выполнения для процесса. Итак, для процесса i :

$$t\,i_{i(\text{процессор})} \leq t_{i\,(\text{исполнение})}$$

где

$t_{i\,(\text{процессор})}$ - это процесс времени, который *я* проводит с использованием процессора, и $t\,i_{i(\text{исполнение})}$ - это общее время выполнения процесса; т.е. время для циклов процессора плюс циклы ввода/вывода, которые должны выполняться (выполняются) до завершения процесса.

На самом деле, обычно, сумма всего процессорного времени, используемого *N-процессами,* редко превышает небольшую долю времени для выполнения любого из

процессов;

$$\sum_{j=1}^{N} t_{j\,(\text{processor})} < t_{i\,(\text{execution})}$$

Поэтому в однопрограммных системах процессор простаивает в течение значительной части времени. Чтобы преодолеть эту неэффективность, многопрограммирование теперь реализовано в современных операционных системах, таких как Linux, UNIX и Microsoft Windows. Это позволяет процессору переключаться с одного процесса, X, на другой, Y, всякий раз, когда X участвует в фазе ввода/вывода его выполнения. Поскольку время обработки намного меньше, чем время выполнения одной работы, общее время для обслуживания всех пользователей *N* с многопрограммной системой может быть сокращено примерно до:

$$t_{\text{multi}} = t\max(t_1 1, t_2 2, \ldots, t_N)$$

Создание процесса

Операционные системы требуют некоторых способов создания процессов. В очень простой системе, предназначенной для запуска только одного приложения (например, контроллер в микроволновой печи), может быть возможно, что все процессы, которые когда-либо будут необходимы, присутствуют, когда система встанет. Однако в системах общего назначения необходим некоторый способ создания и завершения процессов, необходимых во время работы.
Существует четыре основных события, которые вызывают процесс создания:

- Инициализация системы.
- Выполнение системы создания процессов вызовом в запущенном процессе.
- Пользователь требует создания нового процесса.
- Начало пакетной работы.

При загрузке операционной системы обычно запускается несколько основных процессов для подготовки системы к работе. Некоторые из них являются процессами на переднем плане, которые взаимодействуют с (человеком) пользователем и выполняют работу для них. Другие — это фоновые процессы, которые не связаны с конкретными пользователями, а вместо этого имеют определенную функцию. Например, один фоновый процесс может быть разработан для приема входящей электронной почты, спящих большую часть дня, но внезапно оживающего, когда приходит входящее электронное письмо. Альтернативный фоновый процесс может быть разработан для приема входящих запросов на веб-страницы, размещенные на машине, пробужденные только тогда, когда приходит запрос на его обслуживание.

Создание процессов в UNIX и Linux осуществляется через системные вызовы fork() или clone(). Есть несколько шагов, участвующих в создании процесса. Первым шагом является проверка того, имеет ли родительский процесс достаточное разрешение для создания процесса. После успешной проверки родительский процесс копируется почти полностью, с изменениями только в уникальном идентификаторе процесса,

родительском процессе и пользовательском пространстве. Каждый новый процесс получает свое пользовательское пространство.[1]]

Создание процессов в Windows осуществляется посредством системного вызова CreateProcessA(). Новый процесс выполняется в контексте безопасности процесса вызова, но в остальном выполняется независимо от процесса вызова. Существуют методы, изменяющие контекст безопасности, в котором запускаются новые процессы. Новым процессам назначаются идентификаторы, с помощью которых к ним можно получить доступ. Предоставляются функции для синхронизации вызывающих потоков с вновь созданными процессами.[[3]

Расторжение процесса

Существует множество причин для прекращения процесса:

- Пакетные вопросы работы останавливают обучение
- Пользователь выходит из системы
- Процесс выполняет запрос на обслуживание для прекращения
- Ошибка и условия неисправности
- Нормальное завершение
- Срок превышен
- Память недоступна
- Нарушение границ; например: попытка доступа (несуществующего) 11-го элемента 10-элементной массива
- Ошибка защиты; например: попытка записать в файл только для чтения
- Арифметическая ошибка; например: попытка деления на ноль
- Перерасход времени; например: процесс ждал более длительного, чем указанный максимум для события
- I/OПровал ввода/вывода
- Неверная инструкция; например: когда процесс пытается выполнить данные (текст)
- Привилегированное обучение
- Неправильное использование данных
- Вмешательство операционной системы; например: для разрешения тупика
- Родитель прекращает, чтобы детские процессы прекратились (каскадное прекращение)
- Запрос родителей

Двухгосударственная модель управления процессами

Основной обязанностью операционной системы является контроль выполнения

процессов. Это включает в себя определение взаимоудерживающейся структуры исполнения и распределения ресурсов на процессы. Одна из частей разработки ОС заключается в описании поведения, которое мы хотели бы, чтобы каждый процесс демонстрировал. Простейшая модель основана на том факте, что процесс либо выполняется процессором, либо нет. Таким образом, процесс может считаться находящимся в одном из двух состояний, *запущенном* или *НЕ ЗАПУЩЕННОМ*. Когда операционная система создает новый процесс, этот процесс первоначально помечается как *НЕ ЗАПУЩЕННЫЙ*, и помещается в очередь в системе в состоянии *НЕ* ЗАПУСКА. Процесс (или какая-то его часть) затем существует в основной памяти, и он ждет в очереди, чтобы получить возможность быть выполненным. Через некоторое время процесс *запущенного* в настоящее время будет прерван и перенесен из состояния *RUNNING* в состояние *НЕ ЗАПУСКА,* что сделает процессор доступным для другого процесса. Затем диспетчерская часть ОС будет выбирать из очереди процессов *NOT RUNNING* один из процессов ожидания для передачи в процессор. Выбранный процесс затем пересчитывается из *НЕИСТОРОЖНОГО* состояния в состояние*, бегающее,* и его исполнение либо начинается, если это новый процесс, либо возобновляется, если это процесс, который был прерван в более раннее время.

Из этой модели можно выделить некоторые элементы дизайна ОС:

- Необходимость представлять и отслеживать каждый процесс
- Состояние процесса
- Очереди процессов *NON RUNNING*

Трехгосударственная модель управления процессами

Хотя двухгосударственная модель управления процессами является совершенно правильной конструкцией для операционной системы, отсутствие *БЛОКИРОВАННОГО* состояния означает, что процессор простаивает, когда активный процесс меняется от циклов процессора к циклам ввода/вывода. Такая конструкция не позволяет эффективно использовать процессор. Модель управления тремя государственными процессами призвана преодолеть эту проблему, путем введения нового состояния под названием *БЛОКИРОВАННОЕ* государство. Это состояние описывает любой процесс, который ожидает события ввода/вывода. В этом случае событие ввода/вывода может означать использование какого-либо устройства или сигнала из другого процесса. Три государства в этой модели:

- *БЕГ:* Процесс, который в настоящее время выполняется.
- *ГОТОВНОСТЬ:* Процесс, который стоит в очереди и готов выполнять, когда предоставляется такая возможность.
- *БЛОКИРОВАННЫЙ:* процесс, который не может выполнить до какого-то события, например, завершение операции ввода-вывода.

В любой момент процесс находится в одном из трех государств. Для одного процессора только один процесс может быть в состоянии *запущенности* в любой момент. В *ГОТОВЫХ* и *БЛОКИРОВАННЫХ* состояниях может быть много процессов, и каждое из

этих состояний будет иметь связанную очередь на процессы.

Процессы, входящие в систему, должны сначала перейти в состояние *READY,* и процессы могут входить в состояние *RUNNING* только через состояние *READY*. Процессы обычно оставляют систему из состояния *запущенного*. Для каждого из трех состояний процесс занимает пространство в основной памяти. Хотя причина большинства переходов из одного состояния в другое может быть очевидной, некоторые из них могут быть не такими ясными.

- ***ГОТОВНОСТЬ:*** Наиболее распространенной причиной этого перехода является то, что процесс выполнения достиг максимально допустимого времени для бесперебойного выполнения; т.е. происходит тайм-аут. Другими причинами могут быть введение уровней приоритетов, определяемых политикой планирования, используемой для Планировщика низкого уровня, и поступление процесса с более высоким приоритетом в состояние ГОТОВОГО.
- ***ЗАБЛОКИРОВАННЫЙ:*** процесс вводится в *ЗАБЛОКИРОВАННОЕ* состояние, если он запрашивает что-то, что он должен ждать. Запрос на ОС обычно находится в форме системного вызова (т.е. вызов из процесса выполнения в функции, которая является частью кода ОС). Например, процесс может стать *ЗАБЛОКИРОВАННЫМ,* если он запрашивает файл с диска или сохранение раздела кода или данных из памяти в файл на диске.

Описание процессов и контроль

Каждый процесс в системе представлен структурой данных, называемой блоком управления процессами (PCB) или Process Descriptor в LinuxLinux.

Идентификация процесса: Каждый процесс однозначно идентифицируется идентификацией пользователя и указателем, соединяющим его с дескриптором.

Состояние процесса: это указывает на текущее состояние процесса; *READY*ГОТОВЫ, *БЕЖАТЬ*, *ЗАБЛОКИРОВАТЬ, ГОТОВО ПОДВЕШИВАТЬ, ЗАБЛОКИРОВАТЬ ПОДВЕШЕННОЕ.*

Состояние процесса: В нем содержится вся информация, необходимая для указания текущего состояния работы.

Учет: Это содержит информацию, используемую главным образом для целей выставления счетов и для оценки эффективности. Он показывает, какие ресурсы использовался в процессе и как долго.

Режимы процессора

Современные процессоры включают в себя режим для определения возможности выполнения программы в процессоре. Этот бит можно установить *в режим ядра* или *в пользовательском режиме*. Режим ядра также обычно называют *режимом супервизора, режимом мониторинга* или *кольцом 0*.

В режиме ядра процессор может выполнять каждую инструкцию в своем аппаратном репертуаре, тогда как в пользовательском режиме он может выполнять только подмножество инструкций. Инструкции, которые могут выполняться только в режиме ядра, называются ядровыми, привилегированными или защищенными инструкциями, чтобы отличить их от инструкций пользовательского режима. Например, инструкции ввода-вывода являются привилегированными. Таким образом, если прикладная программа выполняется в пользовательском режиме, она не может выполнять свой собственный ВН. Вместо этого он должен запросить ОС для выполнения ввода-вывода от своего имени.

Концепция системы ядра

Критические части ОС работают в режиме ядра, в то время как другие программы (такие как системные утилиты и прикладные программы) работают в пользовательском режиме. Это служит фундаментальным различием между ОС и другим системным программным обеспечением. Часть системы, исполняемая в режиме ядра, называется ядром, или ядром, ОС. Ядро разработано как надежное программное обеспечение, что означает, что оно реализует механизмы защиты, которые не могут быть скрыто модифицированы ненадежным программным обеспечением, работающим в пользовательском режиме. Расширения ОС работают в пользовательском режиме, поэтому основная функциональность ОС не зависит от этих расширений для ее правильной работы.

Ключевым решением по дизайну для любой функции ОС является определение того, следует ли ее внедрять в ядре. При внедрении в ядре оно работает в режиме ядра, получая доступ к другим частям ядра и получая от них доверие. И наоборот, если функция выполняется в пользовательском режиме, она не имеет доступа к структурам данных ядра, но требует минимальных усилий для вызова. Хотя функции, реализованные в ядре, могут быть простыми, механизм ловушки и процесс аутентификации, необходимый во время вызова, могут быть относительно ресурсоемкими. В то время как сам код ядра работает эффективно, накладные расходы, связанные с вызовом, могут быть значительными. Это тонкое, но важное различие.

Запрос системных услуг

Существует два способа, с помощью которых программа, выполняющая в режиме пользователя, может запросить kernelуслуги ядра:

- Системный вызов
- Передача сообщения

Операционные системы спроектированы с одним или другим из этих двух объектов, но не с обоими. Во-первых, предположим, что пользовательский процесс желает вызвать определенную функцию целевой системы. Для подхода к системному вызову пользовательский процесс использует инструкцию ловушки. Идея заключается в том, что системный вызов должен казаться обычной процедурой для прикладной программы; ОС предоставляет библиотеку пользовательских функций с именами, соответствующими

каждому фактическому вызову системы. Каждая из этих функций содержит ловушку для функции ОС. Когда прикладная программа вызывает заглушку, она выполняет инструкцию ловушки, которая переключает ЦП в режим ядра, а затем ветвь (непосредственно через ОС[]), в точку входа функции, которая должна быть вызвана. Когда функция завершается, она переключает процессор в режим пользователя, а затем возвращает управление в пользовательский процесс, тем самым имитируя нормальную процедуру возврата.

В подходе передачи сообщений пользовательский процесс конструирует сообщение, которое описывает желаемый сервис. Затем он использует доверенную функцию отправки для передачи сообщения доверенному процессу ОС. Функция отправки служит той же цели, что и ловушка; то есть она тщательно проверяет сообщение, переключает процессор в режим ядра, а затем доставляет сообщение в процесс, который реализует целевые функции. Между тем, пользовательский процесс ждет результата запроса на обслуживание с получением сообщения операции. Когда процесс ОС завершает операцию, он отправляет сообщение обратно в процесс пользователя.

Различие между двумя подходами имеет важные последствия в отношении независимости поведения ОС от поведения процесса приложения и полученной производительности. Как правило, операционные системы, основанные на интерфейсе системного вызова, могут быть сделаны более эффективными, чем те, которые требуют обмена сообщениями между различными процессами. Это так, даже если системный вызов должен быть реализован с помощью инструкций по ловушкам; то есть, даже если ловушка относительно дорогая в выполнении, она более эффективна, чем подход передачи сообщений, где, как правило, более высокие затраты, связанные с мультиплексированием процесса, формированием сообщений и копированием сообщений. Подход к системному вызову имеет интересное свойство, что не обязательно существует какой-либо процесс ОС. Вместо этого процесс, выполняющий в пользовательском режиме, изменяется в режим ядра, когда он выполняет код ядра, и переключается обратно в пользовательский режим, когда он возвращается с вызова ОС. Если, с другой стороны, ОС разработана как набор отдельных процессов, ее обычно легче спроектировать так, чтобы она получила контроль над машиной в особых ситуациях, чем если бы ядро представляло собой просто совокупность функций, выполняемых пользовательскими процессами в режиме ядра. Операционные системы, основанные на процедурах, обычно включают в себя, по крайней мере, несколько системных процессов (называемых демонами в UNIX) для обработки ситуаций, в которых машина простаивает, например, планирование и обработка сети.

См. также

- Изоляция процесса

Ссылки

1. «A Sneak-Peek into Linux Kernel - Глава 2: Создание процессов» (http://sunnyeves.blogspot.com/2010/09/sneak-peek-into-linux-kernel-chapter-2.html)

2. "Создать функцию ProcessA (Processthreadsap.h) - (https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa) Приложения Win32. (https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-createprocessa) 9 февраля 2023 года.
3. «Создание процессов — Win32 (https://web.archive.org/web/20230329231822/https://learn.microsoft.com/en-us/windows/win32/procthread/creating-processes) приложений». 9 февраля 2023 года. Архивировано из оригинала (https://docs.microsoft.com/en-us/windows/win32/procthread/creating-processes) на 2023-03-29.

Источники

- Операционная система включает Windows и UNIX, Колин Ричи. ISBN 0-8264-6416-5
- Операционные системы, Уильям Столлингс, Прентис Холл, (4-е издание, 2000)
- Мультипрограммирование, описание процессов и контроль
- Операционные Системы — Современная Перспектива, Гэри Натт, Эддисон Уэсли, (2-е Издание, 2001).
- Модели Управления Процессами, Планирование, UNIX System V Release 4:
- Современные операционные системы, Эндрю Таненбаум, Прентис Холл, (2-е издание, 2001).
- Концепции операционной системы, Silberschatz & Galvin & Gagne (http://codex.cs.yale.edu/avi/os-book/OS9/slide-dir/), Джон Уайли и сыновья, (6-е издание, 2003)