Урок 5

Многозадачность

Особенности реализации многозадачности и её применение в приложениях. Процессы и потоки, их отличия.

Введение

Многозадачность следует отличать от многопроцессорности: хотя многозадачность может быть реализована (и реализуется) с применением нескольких процессоров, но в общем случае многозадачность может быть реализована и на одном процессоре. Более того, в неявном виде она существует и в однозадачных операционных системах, поскольку в них реализован механизм прерываний. Исключение – Радио 86РК: это в полном смысле однозадачный компьютер. Во время операций ввода-вывода, например, при загрузке программы с ленты экран монитора гас. Интересно, что Радио 86РК базировался на отечественных клонах Intel 8086, но прерывания в машине не использовались. Сам вывод INT процессора использовался для управления звуком.

Если в системе есть механизм прерываний, то даже однозадачная операционная система на самом деле работает с несколькими задачами, между которыми происходит переключение: это код выполняемой в данный момент единственной задачи и код прерываний, который по определённому событию прерывают выполнение программы – процессор выполняет код обработчика прерывания и возвращает управление прерванной программе. Впрочем, механизм прерываний позволяет реализовать и более привычную для пользователя многозадачность, хотя качество такой реализации зависит также от возможностей процессора – в частности, от возможности защиты кода и данных (в 8086 процессоре таких механизмов не было, впервые появились в 80286, но полноценно они впервые были реализованы в процессоре Intel 80386).

Однопроцессорная многозадачность

Истинная однозадачность

Абсолютно однозадачными были калькуляторы на базе процессора Intel 4004. Процессор не имел поддержки прерываний, она появилась в Intel 4040.

Радио 86 РК Изображение с сайта: http://sfrolov.livejournal.com/79155.html

Однозадачным был Радио 86 РК, несмотря на поддержку прерываний в процессоре 8086. Вывод процессора INT использовался для формирования звука. Во время загрузки программы с магнитофонной ленты экран гас.

Похожая ситуация наблюдалась и у компьютера ZX-80: построенный на базе процессора Z80 с поддержкой прерываний, компьютер устроен был таким образом, что даже генерация изображения осуществлялась программно. Таким образом, и при нажатии кнопки, и при выполнении BASIC-программы экран гас.

Однозадачность с поддержкой прерываний

Поддержка прерываний позволяет эффективно реализовать работу с внешними устройствами. Процессор не должен читать порт клавиатуры и не должен заниматься отрисовкой видео. Аппаратные прерывания позволяют выполнять небольшие фрагменты кода обработчика прерывания, накапливая, например, полученные коды символов от клавиатуры в буфер, а при чтении соответствующий функции ПЗУ – передавать их программе.

Компьютер становится значительно более интерактивным, чем его ранние бытовые предшественники. Тем не менее задачи могут выполняться только по очереди. Если в MS DOS изначально запущен COMMAND.COM, при запуске программы (например, игры), чтобы вернуться в командный интерпретатор, придётся завершить программу. Это неудобно, поэтому программы либо имитировали командную строку сами (как это делали многочисленные коммандеры, такие, как Norton Commander, Volkov Commander, Dos Navigator), либо реализовали «выход в DOS» (QBASIC, Turbo Pascal, Lexicon), попросту запуская новый экземпляр command.com, оставаясь в памяти, но ничего не делая. Полноценной многозадачности здесь нет, даже невытесняющей: одна программа просто запускает другую, однако при этом уже выполняется не один код. Если нажать правый Ctrl, у экрана появляется красная рамка и регистр переключается с латиницы на русский алфавит – это сработал обработчик прерывания по клавиатуре драйвера-русификатора keyrus. Если переключить снова, программа опять на время прерывается: выполнился код. Таким образом, фактически параллельно работают две программы: запущенная программа (редактор NCEDIT.EXE из состава Norton Commander или встроенный в DOS EDIT.COM, а на самом деле QBASIC.EXE /EDCOM) и переключатель keyrus.

Идея простейшего двухзадачного выполнения на 8086

Как быть, если в программе нет выхода в DOS? Можно написать обработчик прерывания, перехватить обработчик прерывания клавиатуры и, отследив нажатие одной фиксированной комбинации кнопок (например, Alt-Shift), запустить COMMAND.COM. Разумеется, необходимо сохранить состояние экрана, чтобы восстановить его после выхода из COMMAND.COM, а также сохранить и восстановить значения регистров процессора. Вместо COMMAND.COM можно запустить другую программу – например, VC.COM. Если же нажата не заданная комбинация Alt-Shift, управление передаём исходному обработчику прерываний клавиатуры.

В таком случае о передаче управления по-прежнему речи не идёт, но её можно реализовать – тоже через прерывания. Например, запустим исходное приложение (редактор, игру), нажмём Alt-Shift и при нажатии проверим состояние неких системных переменных. Допустим, одна из них хранит статус, показывающий, какая программа сейчас выполняется (исходная – 0 или «вторая» – 1), вторая – запущена ли вторая программа (0 – не запущена, 1 – запущена). Соответственно, если выполняется первая программа и вторая не запущена, запускаем вторую программу и устанавливаем системные переменные в значения 1 и 1. Запускается command.com, и мы как будто «выходим в DOS», но если опять нажать Alt-Shift, снова запускается обработчик прерывания. Обработчик проверяет, и так как выполняется вторая программа (1 – «вторая», 1 – запущена), меняем статус (0 – «исходная», 1 – запущена) и возвращаем управление в первую программу. Для этого необходимо восстановить состояние исходной программы – в частности, восстановить экран и поместить в стек исходный адрес возврата, чтобы по выходе из прерывания произошёл возврат не во вторую программу, а в первую. Независимо от того, из какой программы в какую мы переходим, приходится сохранять состояния стека, экрана и адрес возврата – указание на место, с которого мы продолжим выполнение программы.

После выхода из command.com необходимо установить соответствующую переменную (0 – исходная, 0 – не запущена) и вернуть управление в исходную программу. Можно реализовать и более сложную схему с поочерёдным переключением между несколькими программами. Впрочем, на процессоре 8086 реализовать полноценную многозадачность не получится по двум причинам.

Первая – недостаток памяти, который не позволяет запускать две сколь угодно сложные программы: при таком запуске получается, что функционирует одна программа и нечто вспомогательное, вроде «выхода в дос», но только по кнопке.

Вторая – отсутствие механизмов защиты. Неважно, предоставлен ли механизм «выход в DOS» самой программой или мы его реализовали через прерывание и даже возможность переключения между двумя задачами. Если запустить в режиме COMMAND.COM программу с ошибкой (например, переименовать расширение текстового файла в .com или .exe и запустить), система останавливается,

и данные в первой программе могут быть потеряны. При переключении между задачами иногда ещё можно было переключиться в исходную программу (без возможности остановить вторую), а иногда нет – например, если сбойная программа отключила прерывания.

Невытесняющая многозадачность

Дальнейшее развитие замысла представляет идея невытесняющей многозадачности. При соответствующей поддержке ОС (DR DOS 6, проект MS DOS 4) или соответствующей программой (DESQView, DOSShell) появляется возможность запускать несколько программ и выполнять их по очереди. В один момент выполняется только одна задача, в момент запуска остальные приостанавливаются.

Для полноценной реализации механизма требовался достаточный объём памяти, поэтому полноценно такие режимы были реализованы на процессоре 80386 (DESQView с помощью QEMM386, EMM386 для MS DOS и DR DOS).

Разумеется, для полноценной мультизадачности требовалась возможность параллельного выполнения задач.

Вытесняющая многозадачность

Казалось бы, на одном процессоре невозможно решить проблему многозадачности комфортно для повседневной работы. Но что, если осуществлять переключение между задачами не по редко возникающему событию (нажатию кнопки), а по таймеру или при поступлении любого прерывания? Тогда каждая из программ будет получать квант времени, и возникнет иллюзия параллельности. В современных операционных системах реализован именно такой подход.

Обратите внимание: процессор при этом по-прежнему ничего не знает о многозадачности – он выполняет текущую операцию, а управление переключением осуществляет операционная система.

Многопроцессорность

Поскольку каждый процессор работает с потоком команд и потоком данных, по способу распределения ресурсов выделяют следующие способы организации:

- SISD (англ. Single Instruction, Single Data) – один поток команд, один поток данных. Это традиционная однопроцессорная архитектура, задействующая практически современные процессоры. При этом SISD процессоры могут быть объединены в многопроцессорные системы;
- SIMD (англ. Single Instruction, Multiple Data) – один поток команд, множество потоков данных. Этот способ позволяет осуществлять параллельную обработку данных, но одной программой и реализуется в векторных процессорах и графических процессорах. При этом для работы с вещественными числами SIMD-расширения имеются в современных процессорах (MMX,SSE и т.д.);
- MISD (англ. Multiple Instruction, Single Data) – множество потоков команд, один поток данных. Фактическая польза от такой архитектуры – резервирование. Она может применяться в военной, аэрокосмической сфере, управлении технологическими процессами и т.д., где важна отказоустойчивость и продолжение работы даже при отказах оборудования;
- MIMD (англ. Multiple Instruction, Multiple Data) – множество потоков команд, множество потоков данных. Каждый процессор независимо выполняет команды и обрабатывает данные, при этом

одновременно разными процессорами могут выполняться разные фрагменты одной и той же программы и обрабатываться разные фрагменты одних и тех же данных.

Многопроцессорные системы с сильной связью

Многопроцессорные системы с сильной связью (англ. Tightly-coupled multiprocessor systems) содержат несколько процессоров, соединённых на шинном уровне. Эти процессоры могут иметь доступ к центральной разделяемой памяти (SMP или UMA) или участвовать в иерархии памяти и с локальной, и с разделяемой памятью (NUMA). Пример процессора, ориентированного на многопроцессорные системы, – Intel Xeon: каждый процессор имеет свой кеш, но доступ к разделяемой памяти имеет через общую шину.

Предельная форма многопроцессорных систем с сильной связью – многоядерные процессоры, где несколько процессов реализуется в одном чипе.

Многопроцессорные системы с гибкой связью

Многопроцессорные системы с гибкой связью (англ. Loosely-coupled multiprocessor systems), часто называемые кластерами, основаны на множественных автономных одиночных или двойных компьютерах, связанных через высокоскоростную систему связи (например, Gigabit Ethernet).

Как правило, в таких системах многопроцессорность реализуется на более высоком уровне. Межпроцессное взаимодействие реализуется с применением стека протоколов TCP/IP, содержащего компоненты межпроцессного взаимодействия. Вместе с технологиями виртуализации это даёт гибкий, масштабируемый и более дешёвый подход по сравнению с многопроцессорными системами с сильной связью. Как правило, компоненты многопроцессорных систем с гибкой связью вне системы могут функционировать как отдельные компьютеры.

Процессы и потоки в Linux

Процессы и сигналы в Linux

Самая простая программа «Hello, world!» будет выполнена как процесс.

hello.c:

```
#include <stdio.h>
int
main (void)
{
  printf ("Hello, World!\n");
  getchar ();
  return 0;
}
```

Компилируем и запускаем:

```
$ gcc hello.c -o hello
$ ./hello
```

Нажимаем Alt-F2 (или Ctrl-Alt-F2 если работали в X-сервер), смотрим id процесса.

```
$ ps ax| grep hello
```

Процессу можно направить сигнал: например, 9 – принудительно снять.

```
$ kill -9 4242
```

В Linux существуют следующие сигналы:

- 1 (SIGHUP) – информирует программу о потери связи с терминалом. Эта ситуация часто возникала в прошлом, в режиме терминального доступа. В настоящее время она может применяться для двух целей:
    - Информирование дочернего процесса о завершении родительского. При завершении родительского процесса ядро направит сигнал 1 дочерним процессам.
        - Именно поэтому, если приложение запущено в фоновом режиме, с помощью указания & в конце команды или fg при завершении консоли программа, исполняющаяся в фоновом режиме, также получит сигнал SIGHUP, если она была запущена таким образом. Это очень напоминает исходное значение SIGHUP

          ```
          $ someprogrum&
          ```
        - либо

          ```
          $ someprogrum
          ^Z
          $ fg
          ```
        - Этому можно воспрепятствовать, указав nohup:

          ```
          $ nohup someprogrum&
          ```
    - Очевидно, что по умолчанию сигнал приводит к завершению программы. Однако демоны перехватывают сигнал и используют для перечитывания файлов конфигурации.
- 2 (SIGINT) – формируется при нажатии Ctrl-C, завершает программу, но может перехватываться или блокироваться: например, в программах, в которых Ctrl-C используются для операции «копирование» или в принципе нет необходимости такого варианта остановки программы.
- 8 (SIGFPE) – означает ошибку операции с целочисленной арифметикой (переполнение, деление на ноль). Сохраняется дамп памяти.
- 9 (SIGKILL) – безусловное завершение программы. Сигнал не может быть перехвачен программой, потому позволяет её остановить в любом случае (но не позволит снять процесс-зомби).
- 11 (SIGSEGV) – формируется при попытке программы обратиться к не принадлежащей ей области памяти. Обычно выводится сообщение «Segmentation fault» и сохраняется дамп памяти. Как правило, такое случается в результате ошибок программиста при работе с указателями.
- 15 (SIGTERM) – вежливая просьба программе завершить работу. Программа может сохранить данные и т.д.

В Linux есть и другие варианты сигналов.

Для отправки сигнала из операционной системы можно использовать команду kill:

```
$ kill -15 4242
```

Приложение с несколькими процессами

Каждый процесс имеет PID – идентификатор процесса. Также у процесса есть PPID – идентификатор родительского процесса.

Программа, которая выполнялась в одном процессе, при необходимости параллельной обработки может «раздвоиться». При этом изначально запущенный процесс обозначается как процесс-родитель, а порождённый – процесс-потомок.

Как правило, для ветвления используется функция fork(). При выполнении fork операционная система копирует процесс (данные тоже будут скопированы: каждый процесс работает со своими данными). После того как процесс скопирован, и родительский, и дочерний процессы продолжат выполнение с точки fork. Родительский процесс в качестве кода возврата от fork() получит PID дочернего процесса, а дочерний – ноль. Тем не менее дочерний процесс тоже может узнать PPID – идентификатор родительского процесса с помощью getppid(). При этом, если дочерний процесс всегда может узнать PPID, в родительский процесс значение возвращается один раз – в результате fork().

Интересный пример применения процессов – запуск другой программы через execvp() или аналогичную (отличаются параметрами).

Если запустить программу в консоли bash, например, ssh, родительская программа словно бы приостанавливается. Происходит это потому, что выполнение ssh происходит в том же процессе, что и bash. Конечно, еще есть терминал, но даже если бы в bash запустили программу, обрабатывающую данные, нам пришлось бы ждать выполнения.

Формат вызова функции execvp():

```
int execvp(const char *file, char *const argv[]);
```

Первый аргумент – имя программы. Поиск осуществляется с учётом переменной окружения PATH.

Второй – перечень аргументов, так же, как и в функции main().

Дочернюю программу необходимо запускать в дочернем процессе. Выполним это на примере запуска программы ls:

```c
#include <sys/types.h>
#include <sys/wait.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char * argv[])
{ int pid, status;
if (argc < 2) {
printf("Usage: %s command, [arg1 [arg2]...]\n", argv[0]);
return EXIT_FAILURE;
}
printf("Starting %s...\n", argv[1]);
pid = fork();
if (pid == 0) {
execvp(argv[1], &argv[1]);
perror("execvp");
return EXIT_FAILURE; // Never get there normally
} else {
if (wait(&status) == -1) {
perror("wait");
return EXIT_FAILURE;
}
if (WIFEXITED(status))
printf("Child terminated normally with exit code %i\n",
WEXITSTATUS(status));
if (WIFSIGNALED(status))
printf("Child was terminated by a signal #%i\n", WTERMSIG(status));
if (WCOREDUMP(status))
printf("Child dumped core\n");
if (WIFSTOPPED(status))
printf("Child was stopped by a signal #%i\n", WSTOPSIG(status));
}
return EXIT_SUCCESS;
}
```

Потоки в Linux

Если процесс самостоятельно распоряжается памятью при создании форка, память копируется, поток похож на процесс, но все потоки работают с одной и той же памятью, представленной процессом.

Потоки также иногда называют нитями (threads).

Реализация потоков, как и процессов, определяется операционной системой. Если в Windows процесс – контейнер потоков, то в Linux поток, по существу, реализован так же, как и процесс.

В соответствии со стандартом POSIX у всех потоков должен быть один и тот же идентификатор процесса. В Linux это достигнуто через ухищрение: в качестве идентификатора возвращается идентификатор первого процесса многопоточного приложения.

Для создания потока используется функция pthread_create.

Функция потока должна иметь заголовок вида:

```
void * func_name(void * arg)
```

Аргумент arg – указатель, который передаётся в последнем параметре функции pthread_create().

Для завершения потока используется функция pthread_exit().

Для синхронизации потоков и получения значений используется pthread_join().

Вызов функции pthread_join() приостанавливает выполнение вызвавшего её потока, пока поток, чей идентификатор передан функции в качестве аргумента, не завершит работу.

```
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <pthread.h>
void * thread_func(void *arg)
{ int i;
int loc_id = * (int *) arg;
for (i = 0; i < 4; i++) {
printf("Thread %i is running\n", loc_id);
sleep(1);
}
}
int main(int argc, char * argv[])
{ int id1, id2, result;
pthread_t thread1, thread2;
id1 = 1;
result = pthread_create(&thread1, NULL, thread_func, &id1);
if (result != 0) {
perror("Creating the first thread");
return EXIT_FAILURE;
}
id2 = 2;
result = pthread_create(&thread2, NULL, thread_func, &id2);
if (result != 0) {
perror("Creating the second thread");
return EXIT_FAILURE;
}
result = pthread_join(thread1, NULL);
if (result != 0) {
perror("Joining the first thread");
return EXIT_FAILURE;
}
result = pthread_join(thread2, NULL);
if (result != 0) {
perror("Joining the second thread");
return EXIT_FAILURE;
}
printf("Done\n");
return EXIT_SUCCESS;
}
```

Вызывая pthread_create() дважды, мы оба раза передаём в качестве третьего параметра адрес функции thread_func, в результате чего два созданных потока будут выполнять одну и ту же функцию. Разумеется, не каждая функция может корректно работать таким образом.

Функция, вызываемая из нескольких потоков одновременно, должна обладать свойством реентерабельности. Реентерабельная функция – это функция, которая может быть вызвана повторно в то время, когда она уже вызвана (отсюда и происходит её название). Реентерабельные функции используют локальные переменные и локально выделенную память в тех случаях, когда их нереентерабельные аналоги могут воспользоваться глобальными переменными. Приведём в пример рекурсивные функции: они также должны обладать свойством реентерабельности.

Компилируем:

```
gcc threads.c -D_REENTERANT -I/usr/include/nptl -L/usr/lib/nptl -lpthread -o
threads
```

Практическое задание

1. Изучить, какие еще сигналы бывают, кроме описанных. Написать простейшую программу, посмотреть, как она реагирует на преднамеренно сделанные ошибки деления на ноль, Ctrl-C, обращения не к своей памяти. Исправить ошибку, попробовать воспроизвести завершение программы уже с помощью kill.

2. Написать программу, которая выполняет какое-либо действие в фоновом режиме благодаря использованию процесса. Вариант: написать программу, которая запускает архиватор/разархиватор, но при этом оставляет работу в консоли.

3. * Написать на С программу, которая складывает две матрицы и результат сохраняет в третьей. Распараллелить вычисления с помощью потоков.

Примечание. Задание со звёздочкой предназначено для тех, кому недостаточно заданий 1-2 и требуются более сложные задачи.

Дополнительные материалы

1. https://habrahabr.ru/post/141206/
2. http://www.opennet.ru/man.shtml?topic=signal&category=2&russian=2
3. http://tetraquark.ru/archives/47

Используемая литература

Для подготовки данного методического пособия были использованы следующие ресурсы:

1. Радио 86 РК – советский самодельный компьютер https://geektimes.ru/post/172405/
2. https://ru.wikipedia.org/wiki/Многопроцессорность
3. https://stackoverflow.com/questions/284325/how-to-make-child-process-die-after-parent-exits
4. http://citforum.ru/programming/unix/signals/
5. http://citforum.ru/programming/unix/proc_&_threads/
6. http://citforum.ru/programming/unix/threads/