Архитектура ЭВМ. Операционные системы

Власов Евгений

Многозадачность

способность операционной системы выполнять несколько программ одновременно, то есть параллельно.

Если аппаратная среда имеет возможность в один момент времени выполнять только одну последовательность команд, а параллельное выполнение нескольких программ достигается за счет частого переключения контекста, то такая ситуация называется «псевдопараллельная» многозадачность.

Полноценная многозадачность может быть реализована **только** на соответствующей аппаратуре.

Виды многозадачности

- Вытесняющая – каждому процессу отводится квант времени Δt на выполнение, после истечения которого ОС прерывает выполнение процесса и запускает планирование.
- Не вытесняющая – процесс, выполняющийся самостоятельно решает отдавать управление ОС или нет.

Процесс

Программа и все необходимые ей данные на этапе исполнения в ОС.

Стандарт ISO 9000:2000 определяет процесс как совокупность взаимосвязанных и взаимодействующих действий, преобразующих входящие данные в исходящие.

Процесс характеризуется некоторую совокупность набора исполняющихся команд, ассоциированных с ним ресурсов (выделенная для исполнения память или адресное пространство, стеки, используемые файлы и устройства ввода-вывода и т. д.) и текущего момента его выполнения (значения регистров, программного счетчика, состояние стека и значения переменных), находящуюся под управлением операционной системы.

Процесс находится под управлением операционной системы, поэтому в нем может выполняться часть кода ее ядра, как в случаях, специально запланированных авторами программы (например, при использовании системных вызовов), так и в непредусмотренных ситуациях (например, при обработке внешних прерываний).

Блок управления процессом
Process Control Block

Для того чтобы операционная система могла выполнять операции над процессами, каждый процесс представляется в ней некоторой структурой данных.

Содержит информацию, специфическую для данного процесса:

- состояние, в котором находится процесс;
- программный счетчик процесса или, другими словами, адрес команды, которая должна быть выполнена для него следующей;
- содержимое регистров процессора;
- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу, общее время использования процессора данным процессом и т. д.);
- сведения об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов).

Состояние процесса

Состояние процесса

- При рождении процесс получает в свое распоряжение **адресное пространство**, в которое загружается программный код процесса;
- Процессу выделяются **стек** и **системные ресурсы**; устанавливается начальное значение программного счетчика этого процесса и т. д. Родившийся процесс переводится в состояние готовность. При завершении своей деятельности процесс из состояния исполнение попадает в состояние закончил исполнение.
- В операционных системах состояния процесса могут быть еще более детализированы, могут появиться некоторые новые варианты переходов из одного состояния в другое.
- Так, например, модель состояний процессов для операционной системы Windows NT содержит 7 различных состояний, а для операционной системы Unix – 9.

Управление процессами

Процесс не может перейти из одного состояния в другое самостоятельно. Изменением состояния процессов занимается ОС, совершая операции над ними.

Операции можно объединить в три пары:

- **создание процесса** – **завершение процесса**;
- **приостановка процесса** (перевод из состояния исполнение в состояние готовность) – **запуск процесса** (перевод из состояния готовность в состояние исполнение);
- **блокирование процесса** (перевод из состояния исполнение в состояние ожидание) – **разблокирование процесса** (перевод из состояния ожидание в состояние готовность).

Операции создания и завершения процесса являются **одноразовыми**.

Все остальные операции, связанные с изменением состояния процессов, будь то запуск или блокировка, как правило, являются **многоразовыми.**

Контексты процесса

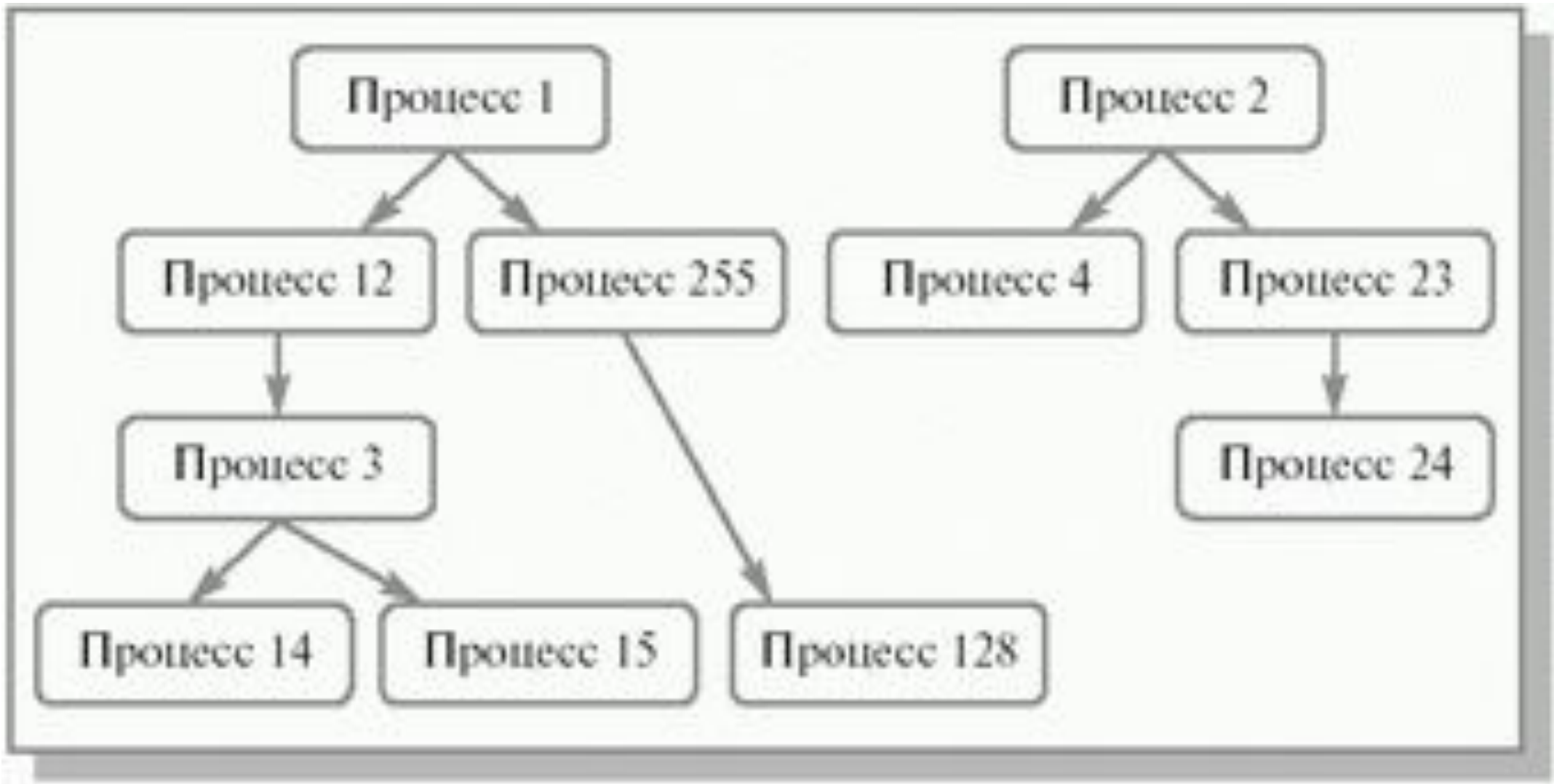
- Информацию, для хранения которой предназначен *блок управления процессом*, можно разделить на две части.
- Содержимое всех регистров процессора (включая значение программного счетчика) называться *регистровым контекстом процесса*, а все остальное – *системным контекстом процесса*.
- Знания *регистрового* и *системного контекстов процесса* достаточно для того, чтобы управлять его работой в операционной системе, совершая над ним *операции*. Однако этого недостаточно для того, чтобы полностью охарактеризовать *процесс*.
- Операционную систему не интересует, какими именно вычислениями занимается *процесс*, т. е. какой код и какие данные находятся в его адресном пространстве. С точки зрения пользователя, наоборот, наибольший интерес представляет содержимое адресного пространства *процесса*, возможно, наряду с *регистровым контекстом* определяющее последовательность преобразования данных и полученные результаты. Код и данные, находящиеся в адресном пространстве *процесса*, называется его *пользовательским контекстом*.
- Совокупность *регистрового*, *системного* и *пользовательского контекстов процесса* для краткости принято называть просто *контекстом процесса*. В любой момент времени *процесс* полностью характеризуется своим *контекстом*.

Рождение процессов

Любая ОС, поддерживающая концепцию *процессов*, обладает средствами для их *создания*. В очень простых системах все *процессы* могут быть порождены на этапе старта системы. Более сложные операционные системы создают *процессы* динамически, по мере необходимости.

Инициатором рождения нового *процесса* после старта операционной системы может выступить либо *процесс* пользователя, совершивший специальный системный вызов, либо сама операционная система, то есть, в конечном итоге, тоже некоторый *процесс*.

Процесс, инициировавший *создание* нового *процесса*, принято называть процессом-родителем (parent process), а вновь созданный *процесс* – процессом-ребенком (child process). Процессы-дети могут в свою очередь порождать новых детей и т. д., образуя, в общем случае, внутри системы набор генеалогических деревьев *процессов* – генеалогический лес.


Процесс 1
Процесс 2
Процесс 12
Процесс 255
Процесс 4
Процесс 23
Процесс 3
Процесс 24
Процесс 14
Процесс 15
Процесс 128

Завершение процессов

После того как *процесс* завершил свою работу, операционная система переводит его в *состояние* закончил исполнение и освобождает все ассоциированные с ним ресурсы, делая соответствующие записи в *блоке управления процессом*.

При этом сам *PCB* не уничтожается, а остается в системе еще некоторое время. Подобная информация сохраняется в *PCB* отработавшего *процесса* до запроса процесса-родителя или до конца его деятельности, после чего все следы завершившегося *процесса* окончательно исчезают из системы. В операционной системе Unix *процессы*, находящиеся в *состоянии* закончил исполнение, принято называть процессами-зомби.

В ряде ОС (например, в VAX/VMS) гибель процесса-родителя приводит к *завершению* работы всех его «детей».

В других операционных системах процессы-дети продолжают свое существование и после окончания работы процесса-родителя. При этом возникает необходимость изменения информации в *PCB* процессов-детей о породившем их *процессе* для того, чтобы генеалогический лес *процессов* оставался целостным.

Многоразовые операции

- ***Приостановка процесса***. Работа *процесса*, находящегося в *состоянии* исполнение, приостанавливается в результате какого-либо прерывания. Процессор автоматически сохраняет счетчик команд и, возможно, один или несколько регистров в стеке исполняемого *процесса*, а затем передает управление по специальному адресу обработки данного прерывания. На этом деятельность hardware по обработке прерывания завершается. По указанному адресу обычно располагается одна из частей операционной системы. Она сохраняет динамическую часть *системного* и *регистрового контекстов процесса* в его *PCB*, переводит *процесс* в *состояние* готовность и приступает к обработке прерывания, то есть к выполнению определенных действий, связанных с возникшим прерыванием.
- ***Блокирование процесса***. *Процесс* блокируется, когда он не может продолжать работу, не дождавшись возникновения какого-либо события в вычислительной системе. Для этого он обращается к операционной системе с помощью определенного системного вызова. Операционная система обрабатывает системный вызов (инициализирует операцию ввода-вывода, добавляет *процесс* в очередь *процессов*, дожидающихся освобождения устройства или возникновения события, и т. д.) и, при необходимости сохранив нужную часть *контекста процесса* в его *PCB*, переводит *процесс* из *состояния* исполнение в *состояние* ожидание.
- ***Разблокирование процесса***. После возникновения в системе какого-либо события операционной системе нужно точно определить, какое именно событие произошло. Затем операционная система проверяет, находился ли некоторый *процесс* в *состоянии* ожидание для данного события, и если находился, переводит его в *состояние* готовность, выполняя необходимые действия, связанные с наступлением события (инициализация *операции* ввода-вывода для очередного ожидающего *процесса* и т. п.).

Получение информации о процессе

Данные ядра, находящиеся в контексте ядра процесса, не могут быть прочитаны процессом непосредственно. Для получения информации о них процесс должен совершить соответствующий системный вызов. Значение идентификатора текущего процесса может быть получено с помощью системного вызова `getpid`(), а значение идентификатора родительского процесса для текущего процесса – с помощью системного вызова `getppid`().

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
```

Системные вызовы создания процесса

Системный вызов fork() служит для создания нового процесса в ОС UNIX. Процесс, который инициировал системный вызов fork(), принято называть родительским процессом (parent process). Вновь порожденный процесс принято называть процессом-ребенком (child process). дочерний процесс является почти полной копией родительского процесса.

У порожденного процесса по сравнению с родительским изменяются значения следующих параметров:

- идентификатор процесса;
- идентификатор родительского процесса;
- время, оставшееся до получения сигнала SIGALRM;
- сигналы, ожидавшие доставки родительскому процессу,
- не будут доставляться порожденному процессу.

Системный вызов fork()

При однократном системном вызове возврат из него может произойти дважды: один раз в родительском процессе, а второй раз в порожденном процессе. Для того чтобы после возвращения из системного вызова `fork`() процессы могли определить, какой из них является дочерним, а какой родительским, и, соответственно, по-разному организовать свое поведение, `fork`() возвращает в них разные значения. Если создать новый процесс не удалось, то системный вызов вернет в инициировавший его процесс отрицательное значение.

Системный вызов `fork`() является единственным способом породить новый процесс после инициализации операционной системы UNIX.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main() {
        pid_t pid;
        int rv;
        switch(pid=fork()) {
                case -1: perror("fork");
                        /* произошла ошибка */
                        exit(1); /*выход из родительского процесса*/
                case 0:
                        printf(" CHILD: Это процесс-потомок!\n");
                                        printf(" CHILD: Мой PID -- %d\n", getpid());
                                        printf(" CHILD: PID моего родителя -- %d\n", getppid());
                                        printf(" CHILD: Введите мой код возврата (как можно меньше):");
                                        scanf(" %d");
                                        printf(" CHILD: Выход!\n");
                        exit(rv);
                default:
                                        printf("PARENT: Это процесс-родитель!\n");
                                        printf("PARENT: Мой PID -- %d\n", getpid());
                                        printf("PARENT: PID моего потомка %d\n",pid);
                                        printf("PARENT: Я жду, пока потомок не вызовет exit()...\n"); wait();
                                        printf("PARENT: Код возврата потомка:%d\n", WEXITSTATUS(rv));
                                        printf("PARENT: Выход!\n");
        }
}
```

PID = 2757

сегмент кода

```
main()
{
…
if(pid=fork())>0)
{…}
else
{…}
}
```

fork()

PID = 2757

сегмент кода

```
main()
{
…
if(pid=fork())>0)
{…}
else if (pid==0)
{…}
}
```

Предок: выполнятся операторы в if-секции

PID = 2760

сегмент кода

```
main()
{
…
if(pid=fork())>0)
{…}
else if (pid==0)
{…}
}
```

Потомок: выполнятся операторы в else-секции