Операционные системы и сети

Управление процессами

Павел Филонов 24/01/2021

Определения

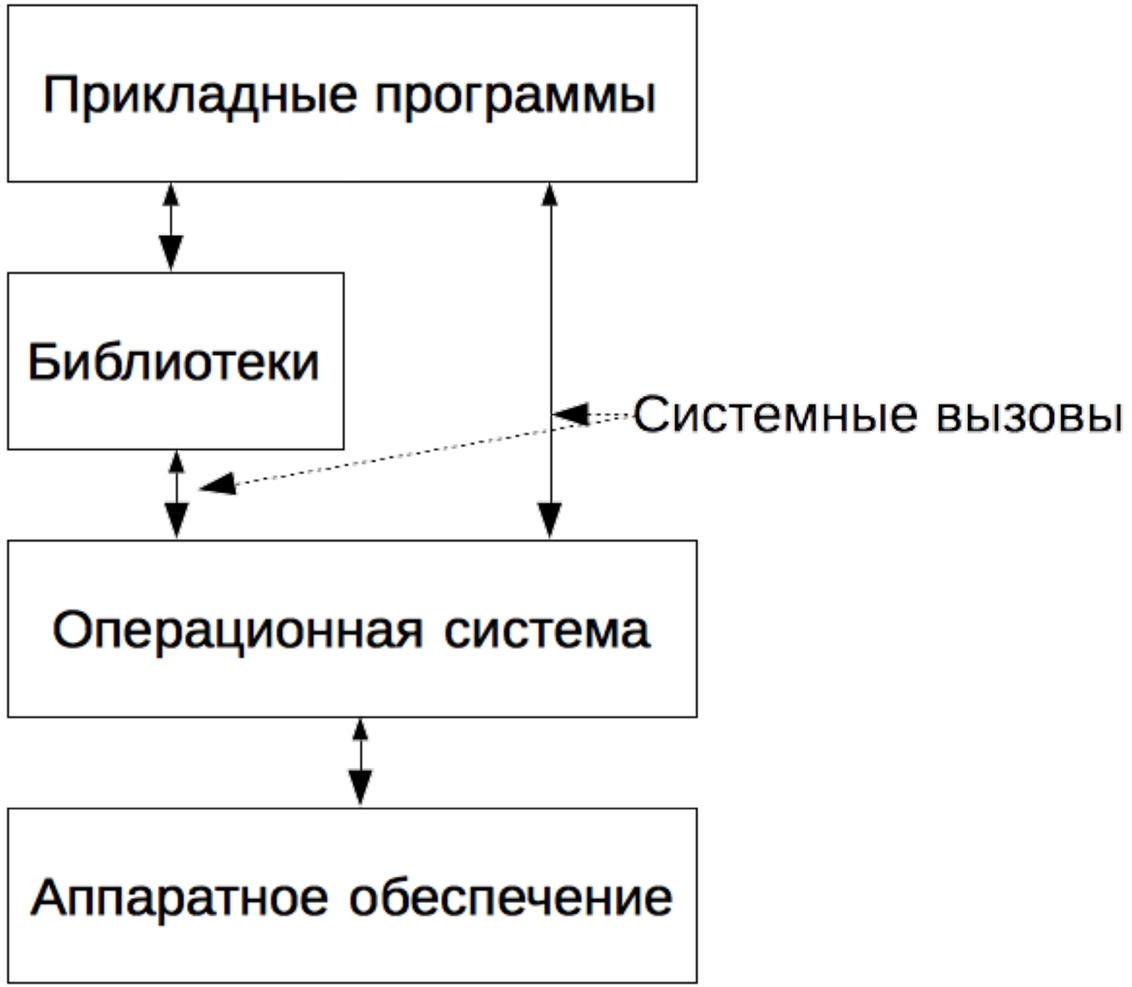
- **Операционная система** это совокупность системных программ, предназначенная для обеспечения определенного уровня эффективности системы обработки информации за счет автоматизированного управления ее работой и предоставляемого пользователю определенного набора услуг [1]

- С точки зрения пользователя **операционная система** выполняет функцию *расширенной машины* или *виртуальной машины*, для которой проще программировать и с которой легче работать, чем непосредственно с аппаратным обеспечением [2]

1. ГОСТ 15971-90. Системы обработки информации. Термины и определения
2. Э. Таненбаум. Современные Операционные системы

Место операционных систем

- Прикладное ПО взаимодействует с аппаратным обеспечением через ОС с помощью системных вызовов [1]
- Системные вызовы осуществляются напрямую или через библиотечные функции



1. Курс “Архитектура ЭВМ и язык ассемблера”

Функции операционных систем

- Мультизадачный режим работы
- Управление оперативной памятью
- Управление файловой системой
- Управление вводом-выводом
- Взаимодействие процессов
- Разграничение полномочий

Процессы

Программа — файл с машинными кодами

Процесс — экземпляр программа, которая исполняется. Включает в себя машинный код и текущий контекст (значения регистров, оперативной памяти и проч.)

Идентификатор процесса (process identifier, PID) — целое, число которое ОС выдает при запуске нового процесса.

Пример. Список процессов

ps -e

```
  PID TTY          TIME CMD
    1 ?        00:01:52 systemd
   49 ?        00:00:00 vballoon
  395 ?        00:00:16 docker-proxy
  407 ?        00:08:08 containerd-shim
  426 ?        00:00:00 sh
 2805 ?        00:00:00 sshd
 2841 pts/0    00:00:00 bash
 2869 tty1     00:00:00 agetty
 3688 pts/0    00:00:00 ps
 6331 ?        00:00:13 nginx
10256 ?        00:02:59 nslcd
10262 ?        00:07:18 sshd
11311 ?        00:00:07 kworker/u2:0
15063 ?        00:10:10 containerd-shim
15095 ?        00:00:00 run
15387 ?        00:06:35 slapd
31192 ?        00:00:00 xfsalloc
31193 ?        00:00:00 xfs_mru_cache
31972 ?        02:17:12 containerd
```

Пример. Идентификатор процесса

man getpid

GETPID(2) BSD System Calls Manual GETPID(2)

NAME
getpid, getppid -- get parent or calling process identification

SYNOPSIS
#include <unistd.h>

pid_t
getpid(void);

pid_t
getppid(void);

DESCRIPTION
getpid() returns the process ID of the calling process. The ID is guaranteed to be unique and is useful for constructing temporary file names.

getppid() returns the process ID of the parent of the calling process.

ERRORS
The getpid() and getppid() functions are always successful, and no return value is reserved to indicate an error.

Пример. Идентификатор процесса

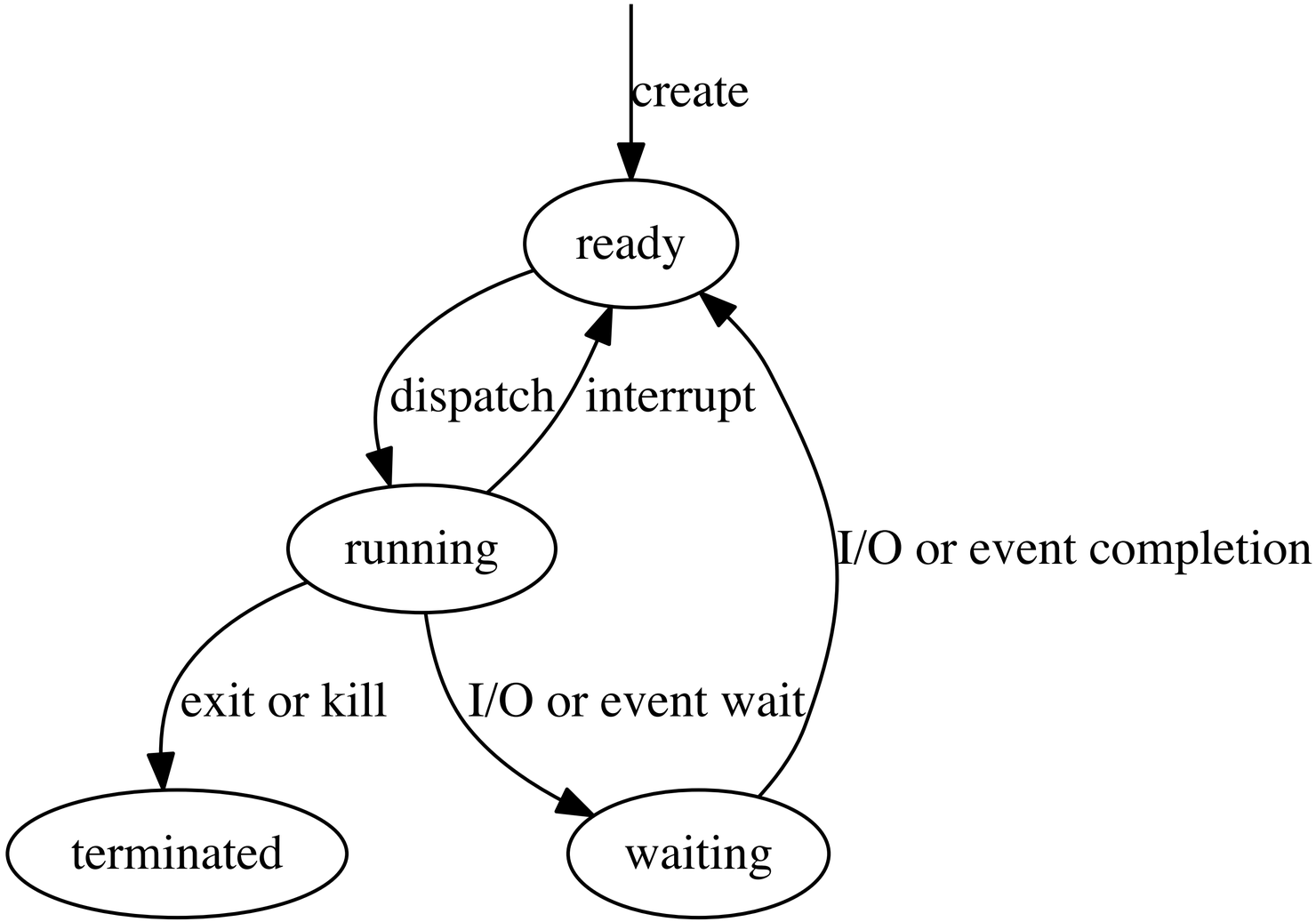
```c
/* Prints Process ID and exit */
#include <stdio.h>
#include <unistd.h> // getpid definition

int main()
{

  pid_t pid = getpid();
  printf("%d\n", pid);

  return 0;
}
```

Автомат состояний процесса



Пример. Состояния процесса в Linux

ps -eo pid,s,comm

```
  PID S COMMAND
    1 S systemd
 4179 S sshd
 4180 R ps
10262 S sshd
11311 S kworker/u2:0
15063 S containerd-shim
15095 S run
15387 S slapd
31192 S xfsalloc
31193 S xfs_mru_cache
31972 S containerd
32634 S dockerd
```

man ps

PROCESS STATE CODES

Here are the different values that the s, stat and state output specifiers (header "STAT" or "S") will display to describe the state of a process:

D uninterruptible sleep (usually IO)
R running or runnable (on run queue)
S interruptible sleep (waiting for an event to complete)
T stopped by job control signal
t stopped by debugger during the tracing
W paging (not valid since the 2.6.xx kernel)
X dead (should never be seen)
Z defunct ("zombie") process, terminated but not reaped by its parent

Иерархия процессов

- Дочерние (child) и родительские (parent) процессы
- Пример вывода утилиты pstree:

```
root@pm-test:~# pstree -p
systemd(1)─┬─accounts-daemon(6451)─┬─{gdbus}(6456)
           │                       └─{gmain}(6454)
           ├─acpid(1393)
           ├─agetty(1480)
           ├─agetty(2869)
           ├─atd(1366)
           ├─containerd(31972)─┬─containerd-shim(407)─┬─sh(426)─────run(462)─┬─run(485)─────openvpn(490)
           │                   │                      │                      ├─run(486)─────openvpn(489)
           │                   │                      │                      └─tail(488)
           │                   │                      └─{containerd-shim}(451)
           │                   └─{containerd}(15880)
           ├─cron(1371)
           ├─dbus-daemon(1383)
           ├─dockerd(32634)─┬─docker-proxy(373)─┬─{docker-proxy}(374)
           │                │                   └─{docker-proxy}(382)
           │                └─{dockerd}(15117)
           ├─sshd(2795)─────sshd(2805)─────bash(2841)─────pstree(2926)
           ├─systemd-udevd(1262)
           └─unattended-upgr(1489)─────{gmain}(1574)
```

Создание процессов

man fork

NAME
fork -- create a new process

SYNOPSIS
#include <unistd.h>

pid_t
fork(void);

DESCRIPTION
fork() causes creation of a new process. The new process (child process) is an exact copy of the calling process (parent process) except for the following:

- The child process has a unique process ID.
- The child process has a different parent process ID (i.e., the process ID of the parent process).
- The child process has its own copy of the parent's descriptors. These descriptors reference the same underlying objects, so that, for instance, file pointers in file objects are shared between the child and the parent, so that an lseek(2) on a descriptor in the child process can affect a subsequent read or write by the parent. This descriptor copying is also used by the shell to establish standard input and output for newly created processes as well as to set up pipes.
- The child processes resource utilizations are set to 0; see setrlimit(2).

RETURN VALUES
Upon successful completion, fork() returns a value of 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, a value of -1 is returned to the parent process, no child process is created, and the global variable errno is set to indicate the error.

Создание процессов

```c
/* Create child process and prints PIDs */

#include <stdio.h>
#include <unistd.h> //getpid, getppid, fork

int main()
{
    pid_t pid = fork();
    if (pid > 0)  {
        printf("Parent process pid: %d\n", getpid());
    } else if (pid == 0) {
        printf("Child process pid: %d\n", getpid());
        printf("Parent process pid: %d\n", getppid());
    } else {
        perror("fork");
    }
    return 0;
}
```

Завершение процесса

man 3 exit

EXIT(3) BSD Library Functions Manual EXIT(3)

NAME
exit, _Exit -- perform normal program termination

LIBRARY
Standard C Library (libc, -lc)

SYNOPSIS
#include <stdlib.h>

void
exit(int status);

void
_Exit(int status);

DESCRIPTION
The exit() and _Exit() functions terminate a process.

Ожидание дочернего процесса

man 2 wait

WAIT(2) BSD System Calls Manual WAIT(2)

NAME

wait -- wait for process termination

SYNOPSIS

```
#include <sys/wait.h>

pid_t
wait(int *stat_loc);
```

DESCRIPTION

The wait() function suspends execution of its calling process until stat_loc information is available for a terminated child process, or a signal is received. On return from a successful wait() call, the stat_loc area contains termination information about the process that exited as defined below.

RETURN VALUES

If wait() returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, a value of -1 is returned and errno is set to indicate the error.

Пример. Ожидание дочернего процесса

```c
/* Print "Hello, world" from child process */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/wait.h>

int main()
{
  pid_t pid = fork();
  if (!pid) { /* child process */
    printf("Hello from child process!\n");
    exit(EXIT_SUCCESS);
  } else if (pid == -1)
    perror("fork");
    exit(EXIT_FAILURE);
  }
  pid = wait(NULL);
  if (pid == -1) {
      perror("wait");
      exit(EXIT_FAILURE);
  }
  return 0;
}
```

Замещение процесса

man 3 exec

EXEC(3) BSD Library Functions Manual EXEC(3)

NAME

execl, execle, execlp, execv, execvp, execvP -- execute a file

LIBRARY

Standard C Library (libc, -lc)

SYNOPSIS

```
#include <unistd.h>

int
execl(const char *path, const char *arg0, ... /*, (char *)0 */);

int
execlp(const char *file, const char *arg0, ... /*, (char *)0 */);

int
execv(const char *path, char *const argv[]);

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

Пример. Замещение процесса

execlp

```c
/* exec ls -l -a command */
#include <unistd.h>

int main()
{
    int retcode = execlp("ls", "ls", "-l", "-a", NULL);
    if (retcode == -1) {
        perror("execlp");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Пример. Замещение процесса

execvp

```c
/* exec ls -l -a command */
#include <unistd.h>

int main()
{
    char *args[] = {"ls", "-l", "-a", NULL};
    int retcode = execvp(args[0], args);
    if (retcode == -1) {
        perror("execlp");
        return EXIT_FAILURE;
    }
    return EXIT_SUCCESS;
}
```

Реализация процессов

- ОС поддерживает таблицу процессов
- Примеры полей в таблице процессов
  - Регистры, счетчик команд, состояние процесса, приоритет, использованное процессоров время, имя процесса
  - Указатели на сегменты .text, .data, .stack, идентификатор процесса, родительский процесс,
  - Корневой каталог, рабочий каталог, дескрипторы файлов,

Переключение процессов

Обработка прерывания

1. Счетчик команд сохраняется в стеке
2. Новый счетчик команда загружается из вектора прерываний
3. Сохраняются регистры
4. Устанавливается новый стек
5. Запускается обработка прерывания
6. Отмечается готовность получателя сообщения
7. Планировщик выбирает следующий процесс
8. Восстанавливаются значения регистров выбранного процесса
9. Счетчик команд загружается их сохраненного для процесса значения

Планирование процессов

Планировщик процессов (process scheduler) — часть ОС, которая отвечает за распределение ресурсов процессора между задачами

Когда требуется планирование:

- Обязательно
  - При завершении процесс
  - При блокировке процесса
- Необязательно
  - При создании нового процесса
  - При прерывании ввода/вывода
  - При прерывании от таймера

Категории алгоритмов планирования

- Системы пакетной обработки
- Интерактивные системы
- Системы реального времени

Цели алгоритмов планирования

- Все системы
  - Равноправие - предоставление каждому процессу справедливой доли процессорного времени
  - Применение политик - наблюдение за соблюдение установленной политики
  - Баланс - обеспечение работой всех компонентов системы

Цели алгоритмов планирования

- Системы пакетной обработки
  - Пропускная способность - выполнение максимального числа задач в единицу времени
  - Время оборота - минимизация времени, затрачиваемого на ожидание обслуживания и обработку задания
  - Коэффициент использования процессора - обеспечение постоянной занятости процессора

Цели алгоритмов планирования

- Интерактивные системы
  - Время отклика - быстрая реакция на запросы
  - Пропорциональность - соответствие ожиданиям пользователя

Цели алгоритмов планирования

- Системы реального времени
  - Соответствие временным ограничениям - избежание потерь данных
  - Предсказуемость - детерминированное время обработки в худшем случае

Алгоритмы планирования

Системы пакетной обработки

- Первым пришел - первым обслужен (FIFS - First In First Served)
- Самое короткое время - первое (SJF - Shortest Job First)
- Задание с наименьшим временем завершения - следующее
- Трёхуровневое планирование
  - Планировщик доступа
  - Планировщик памяти
  - Планировщик процессора

Алгоритмы планирования

Интерактивные системы

- Циклическое планирование (RR - Round Robin)
  - Каждому процессу предоставляется временной интервал, называемый **квантом**
- Приоритетное планирование (priority planning)
  - Каждому процессу пристраивается приоритет (в UNIX - nice)
  - Для каждого приоритета - своя очередь процессов

Алгоритмы планирования

Интерактивные системы

- Гарантированное планирование
- Лотерейное планирование
- Справедливое планирование

Алгоритмы планирования

Системы реального времени

- Системы “жесткого” реального времени (hard real time) - нарушение сроков выполнения недопустим в худшем случае
- Системы “мягкого” реального времени (soft real time) - нарушение сроков выполнения недопустимо в среднем

Пример. Linux scheduler

CFS - Completely Fair Scheduler

- https://nihal111.github.io/CFS-visualizer/
-

Пример. Windows scheduler

Processes, Threads, and Jobs in the Windows Operating System

- Планировщике на основе приоритетов
- 32 уровня приоритетов (16 real-time, 15 dynamic, 1 reserved by system)
-

Пример. QNX RTOS Scheduler

Where Does the Time Go?

- Поддерживаемые алгоритмы планирования:
  - FIFO
  - Round Robin
  - Sporadic scheduling
- Деление времени (Time Partitioning)
  - Процессы объединяются в разделы (partitions)
  - Для каждого раздела резервируется бюджет времени

Пример. FreeRTOS

Real Time Scheduling

- 2 политики планирования
  - Round-Robin с выделенным квантом времени
  - Приоритетное планирование

Источники

- Э. Таненбаум. Операционные системы. Разработка и реализация
- Man POSIX (getpid, ps, fork, exit, wait, exec)
- Linux CFS Scheduler - kernel.org
- Windows system scheduler - docs.microsoft.com
- MINIX - minix3.org