Лекция 3

- Режим ядра и пользовательский режим.
- Системные вызовы.
- Интерфейсы прикладного программирования.
- Управление ресурсами ВС.
- Объекты ядра.
- Процессы. Их реализация и управление ими.
- Создание процессов в Linux.

Взаимодействие прикладных программ и ОС

Режим ядра (режим супервизора, привилегированный режим):

- полный доступ к командам процессора;
- обработка прерываний и исключений;
- доступ к объектам ядра.

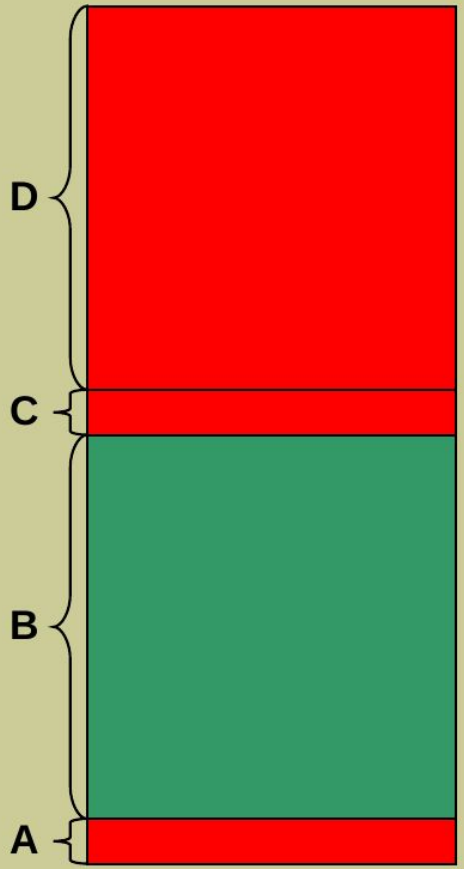
Пользовательский режим:

- ограниченный набор команд процессора;
- запрет на вызов обработчиков прерываний.

Интерфейс системных вызовов предоставляет контролируемый доступ прикладных программ к ресурсам компьютера посредством переход из пользовательского режима в режим ядра.

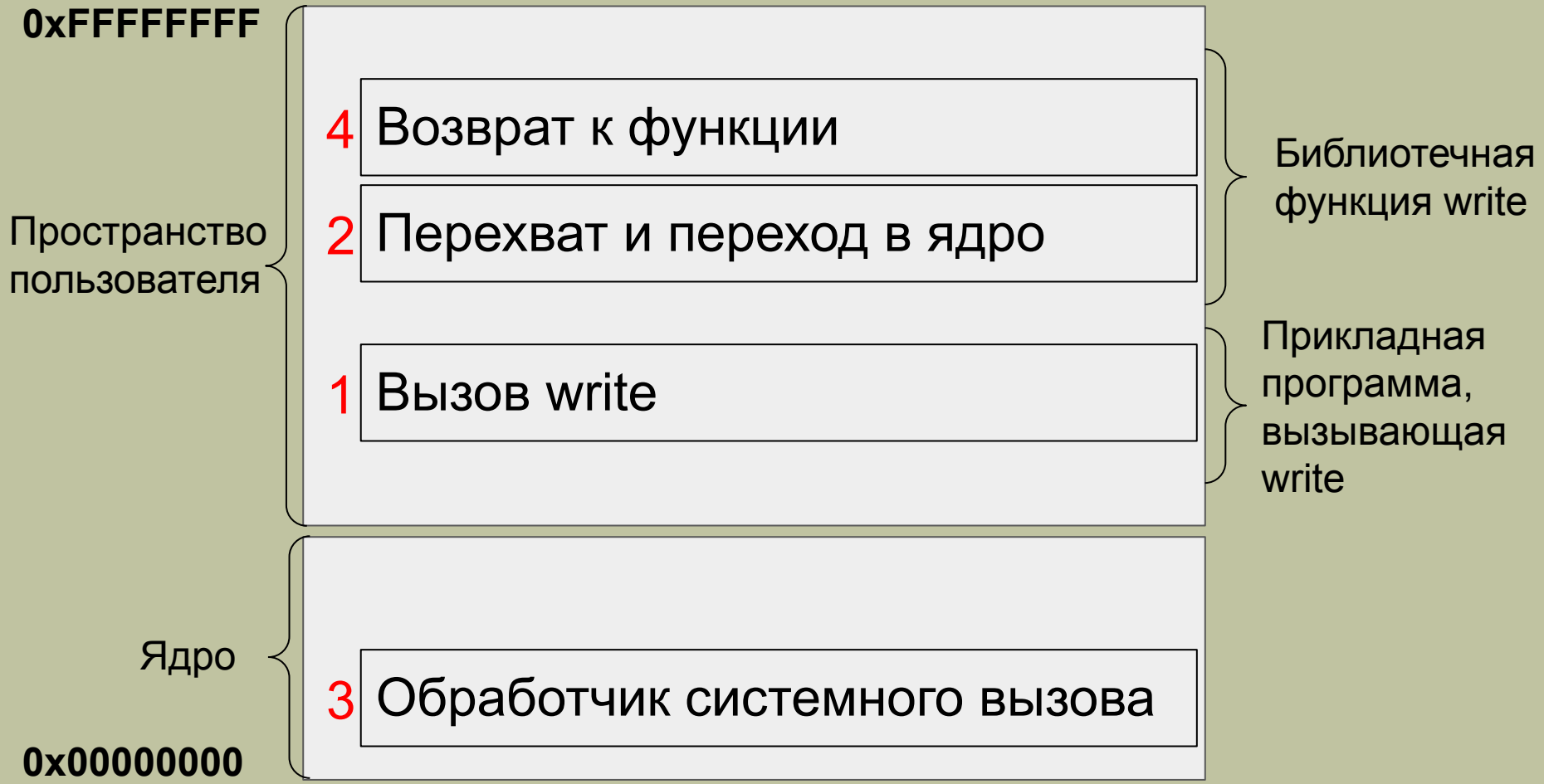
Интерфейс прикладного программирования - библиотечные функции.

Пример структуры адресного пространства 32-разрядной ОС.



A. 0x00000000 – 0x0000FFFF; используется для неинициализированных указателей; ***недоступно*** в пользовательском режиме.
B. 0x00010000 – 0x7FFEFFFF; адресное пространство процессов, содержит прикладные модули .exe и .dll, win32 (kernel32.dll, user32.dll и т.д.), файлы, отображаемые в память; ***доступно*** в пользовательском режиме.
C. 0x7FFF0000 – 0x7FFFFFFF; используется для некорректно инициализированных указателей; ***недоступно*** в пользовательском режиме.
D. 0x80000000 – 0xFFFFFFFF; зарезервировано ОС Windows для исполнительной системы, ядра и драйверов устройств; ***недоступно*** в пользовательском режиме.

write(hFile, pBuffer, nToWrite) – библиотечная функция

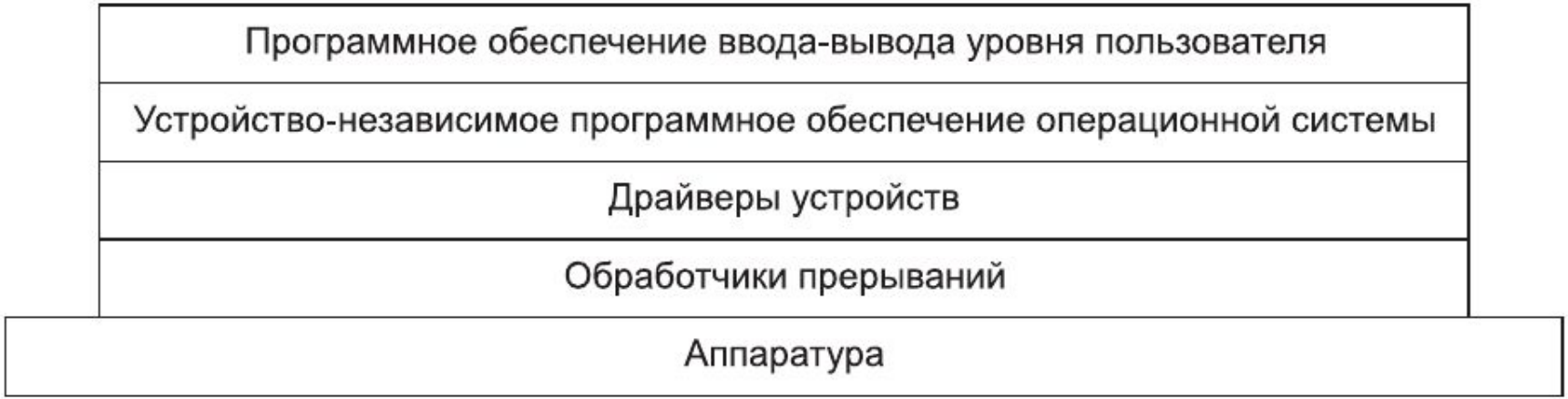


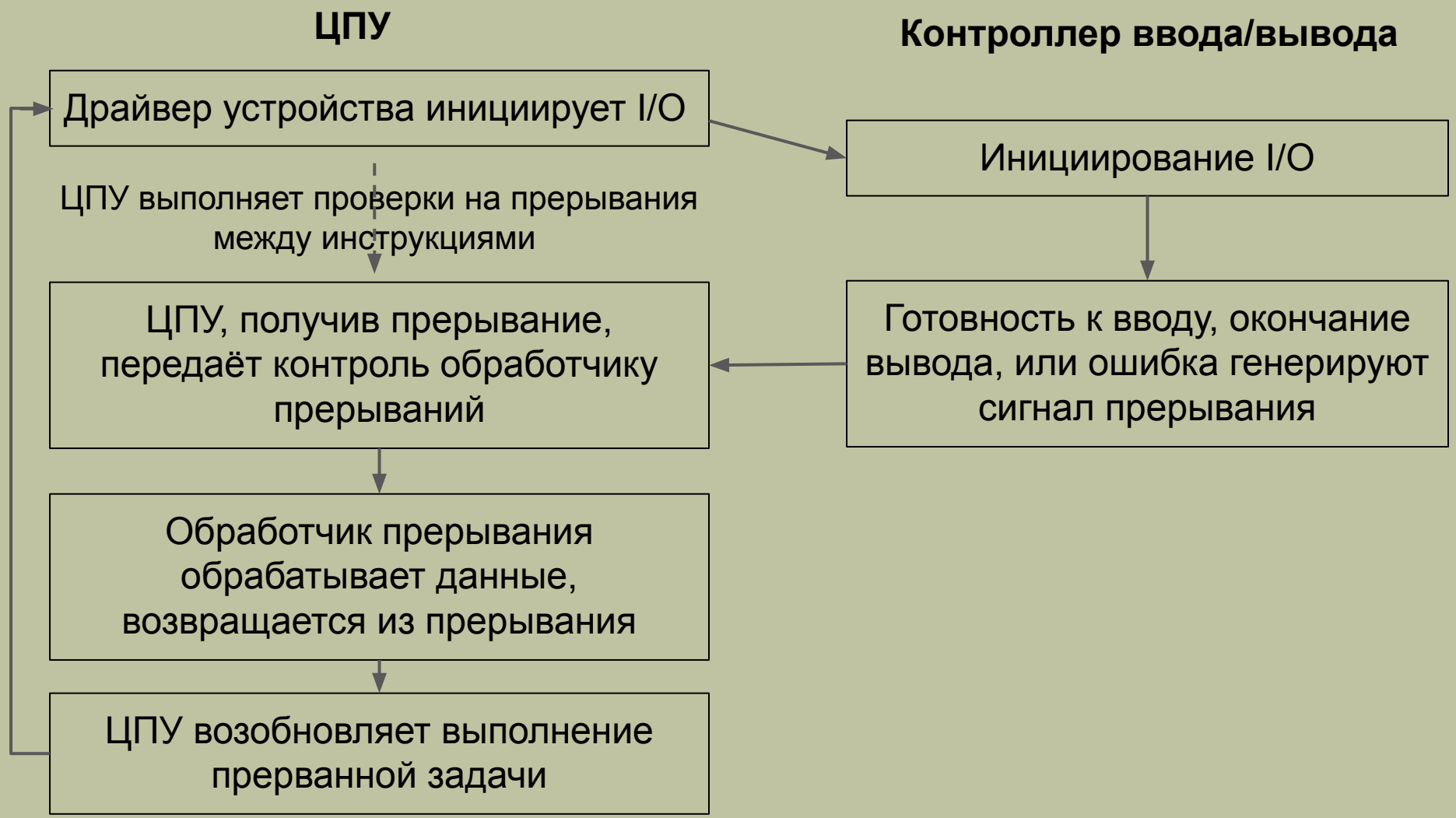
write(int fd, const void *buf, size_t count);
- библиотечная функция

```
mov  edx, 1       ;сколько байт записать
mov  ecx, hex     ;буфер, откуда писать
mov  ebx, 1       ;куда записывать, 1 - stdout
mov  eax, 4       ;номер системного вызова
int  80h          ;шлюз к ядру
```

Таблица системных вызовов

%eax	Name	Source	%ebx	%ecx	%edx
1	sys_exit	kernel/exit.c	int	-	-
2	sys_fork	arch/i386/kernel/process.c	struct pt_regs	-	-
3	sys_read	fs/read_write.c	unsigned int	char*	size_t
4	sys_write	fs/read_write.c	unsigned int	const char*	size_t
5	sys_open	fs/open.c	const char*	int	int
6	sys_close	fs/open.c	unsigned int	-	-


Программное обеспечение ввода-вывода уровня пользователя
Устройство-независимое программное обеспечение операционной системы
Драйверы устройств
Обработчики прерываний
Аппаратура


ЦПУ
Контроллер ввода/вывода
Драйвер устройства инициирует I/O
Инициирование I/O
ЦПУ выполняет проверки на прерывания между инструкциями
ЦПУ, получив прерывание, передаёт контроль обработчику прерываний
Готовность к вводу, окончание вывода, или ошибка генерируют сигнал прерывания
Обработчик прерывания обрабатывает данные, возвращается из прерывания
ЦПУ возобновляет выполнение прерванной задачи

Объекты ядра операционной системы:

- **Process**
- Thread
- File
- File-mapping
- Pipe
- Mutex
- Semaphore

...

Процессы

Процесс – это исполняемый экземпляр программы и набор ресурсов, которые выделяются данной исполняемой программе.

Ресурсы процесса:

• виртуальное адресное пространство;

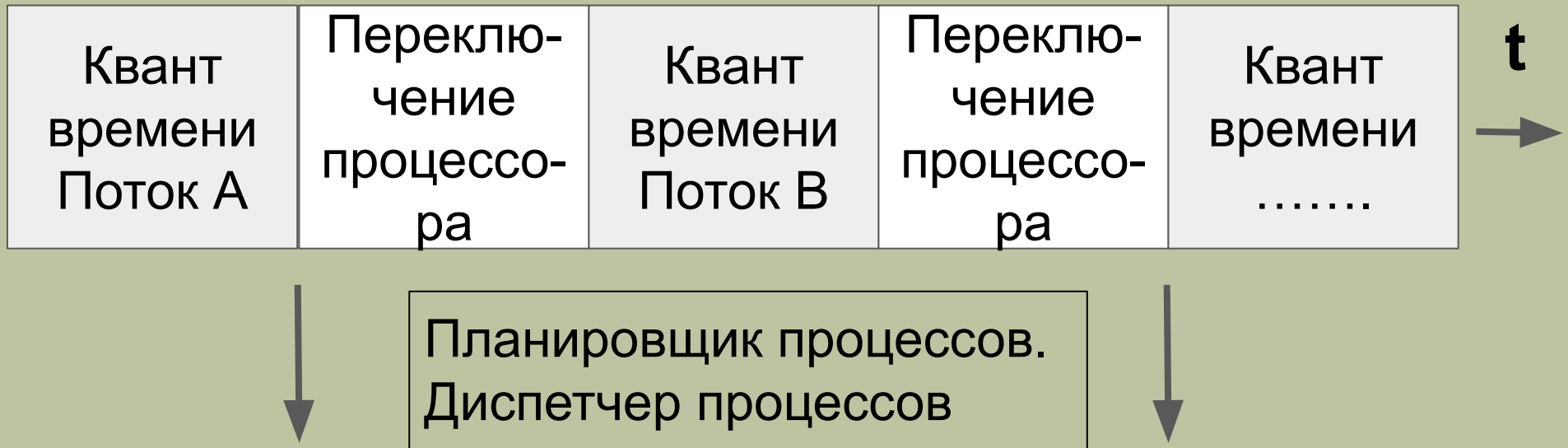
• системные ресурсы –области физической памяти, процессорное время, файлы, растровые изображения и т.д.;

• модули процесса, то есть исполняемые модули, загруженные (отображенные) в его адресное пространство – основной загрузочный модуль, библиотеки динамической компоновки и т.д.;

• уникальный идентификационный номер, называемый идентификатором процесса;

• потоки (по крайней мере, один поток).

Модель процесса:

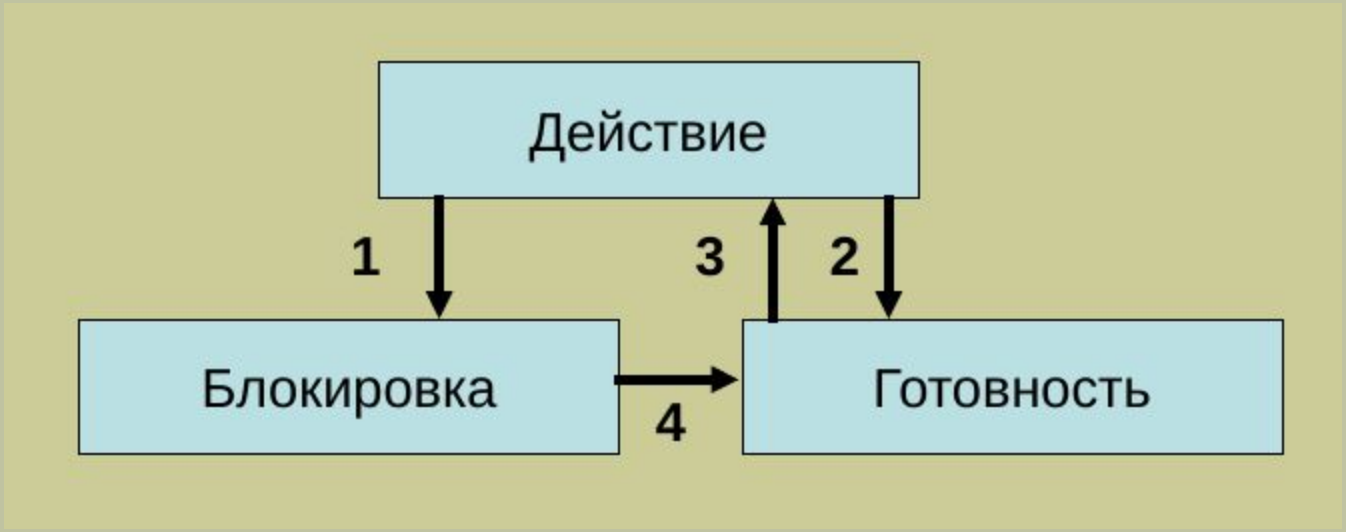


- Сохранение контекста текущего потока, регистров, стека и областей памяти.
- Определение очередного потока.
- Восстановление контекста очередного потока.

Последовательность исполнения потоков в среде с вытесняющей многозадачностью:

В системе определен **квант времени** (порядка десятков миллисекунд) – процессорное время выделяемое одному потоку (каждому - своё). **Длительность выполнения одного потока не может превышать одного кванта.** Когда это время заканчивается, диспетчер процессов переключает процессор на выполнение другого потока. При этом состояние регистров, стека и областей памяти – контекст потока, сохраняется в стеке потока. Очередность потоков определяется их ***состоянием и приоритетом***.

Состояние процессов:



1. Процесс заблокирован в ожидании ввода.
2. Диспетчер выбирает другой процесс.
3. Диспетчер выбирает данный процесс.
4. Входные данные стали доступны.

Реализацией процессов является **таблица процессов**, программно реализованная, как список структур) (***P****rocess* ***C****ontrol* ***B****lock*).

Информация о процессах хранится в таблице процессов и обновляется планировщиком процессов.

Некоторые поля типичной записи таблицы процессов:

Регистры
Счетчик команд
Состояние процесса
Приоритет
Идентификатор процесса
Родительский процесс
Время запуска процессора
Использованное время процессора

Корневой каталог
Рабочий каталог
Дескрипторы файлов
Идентификатор пользователя

Создание процесса:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void oldman();
void recreation();

int main(){
  pid_t  child_pid, parent_pid;
  int i=0;
```

```
 fprintf(stdout, "Before RECREATION %i\n",
                                          parent_pid=(int)getpid());

child_pid=fork();

 while(i++<5)
   if(child_pid!=0)
     oldman();
   else
      recreation();
 return 0;
}
```

```
#include  <sys/types.h>
#include  <unistd.h>

void oldman(){
fprintf(stdout, "I'm not yet dead! My ID is %i\n", (int) getpid());
}

void recreation(){
fprintf(stdout, "Who I am? My ID is %i\n", (int) getpid());
}
```

```
~> ./2
Before RECREATION 6169
I'm not yet dead! My ID is 6169
I'm not yet dead! My ID is 6169
I'm not yet dead! My ID is 6169
Who I am? My ID is 6170
I'm not yet dead! My ID is 6169
I'm not yet dead! My ID is 6169
Who I am? My ID is 6170
Who I am? My ID is 6170
Who I am? My ID is 6170
Who I am? My ID is 6170
```

```
~> ./2
Before RECREATION 6154
I'm not yet dead! My ID is 6154
I'm not yet dead! My ID is 6154
Who I am? My ID is 6155
Who I am? My ID is 6155
Who I am? My ID is 6155
Who I am? My ID is 6155
Who I am? My ID is 6155
I'm not yet dead! My ID is 6154
I'm not yet dead! My ID is 6154
I'm not yet dead! My ID is 6154
```

```
 #include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
pid_t child_pid, parent_pid;
double s=0.0;

child_pid=fork();
```

```
if(child_pid!=0){
  s+=3.14;
  fprintf(stdout, "CHILD: %i s=%g &s=%u\n", (int) getpid(),s,&s);
}
else{
 s+=2.72;
 fprintf(stdout, "PARENT: %i s=%g &s=%u\n", (int) getpid(),s, &s);
}
return 0;
}
```

PARENT: 5404 s=2.72 &s=2309295864
CHILD: 5403 s=3.14 &s=2309295864

При создании процесса с помощью системного вызова fork() копируется адресное пространство, - переменная s имеет один и тот же адрес. Однако отображение на физическую память для родительского и дочернего процесса различно, - значения переменной s различны.

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
pid_t child_pid;
pid_t parent_pid;
double s=0.0;;
FILE* fp;

child_pid=fork();
fp=fopen("test.dat","a+");
```

```
if(child_pid!=0){
  s+=3.14;
  fprintf(fp, "CHILD: %i s=%g &s=%u fp=%u\n", (int) getpid(),
s, &s, fp);
}
else{
  s+=2.72;
  fprintf(fp, "PARENT: %i s=%g &s=%u fp=%u\n",(int) getpid(),
  s, &s,fp);
}
fclose(fp);
return 0;
}
```

test.dat

```
PARENT: 5450 s=2.72 &s=760346688 fp=6299664
CHILD: 5449 s=3.14 &s=760346688 fp=6299664
```

Дескрипторы файлов при копировании сохраняются.