Прикладные физико-технические и компьютерные методы исследований

Семинар 3

На прошлом семинаре…

- pwd
- man
- cd
- ls
- ls –al

Указатели

```
void Swap1(int a, int b)
{
  int temp;
  temp = a;
  a = b;
  b = temp;
}
```

```
void Swap2(int* a, int* b)
{
  int* temp;
  temp = a;
  a = b;
  b = temp;
}
```

```
void Swap3(int* a, int* b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}
```

Снова указатели …

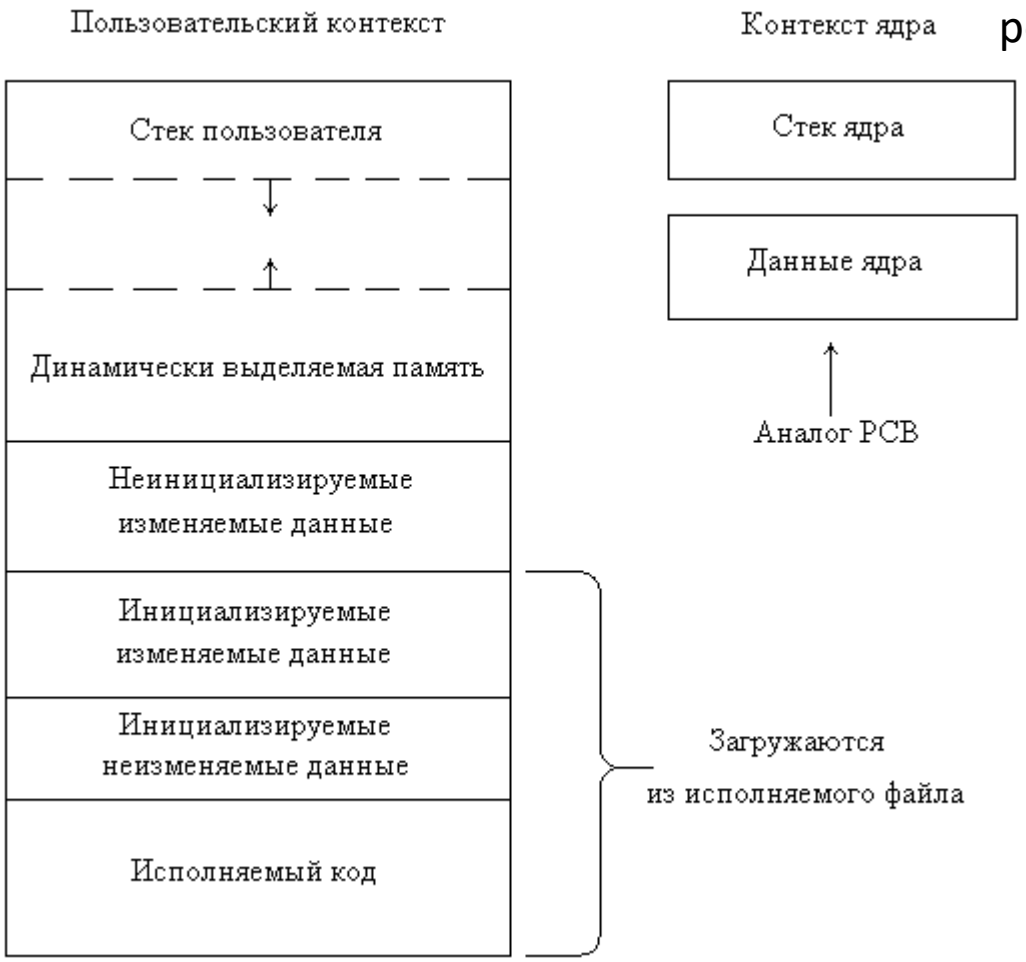
```
27  int main()
28  {
29      int x = 3;
30      int y = 4;
31      Swap1(x, y);
32      // Swap2(&x, &y);
33      // Swap3(&x, &y);
34      printf("%d %d\n", x, y);
35      return 0;
36  }
```

Процессы. Состояния процессов.

Состояние процесса.

Контекст процесса

системный + регистровый



UID, GID, PID, PPID и др.

Например, статические переменные

Глобальные переменные

Константы

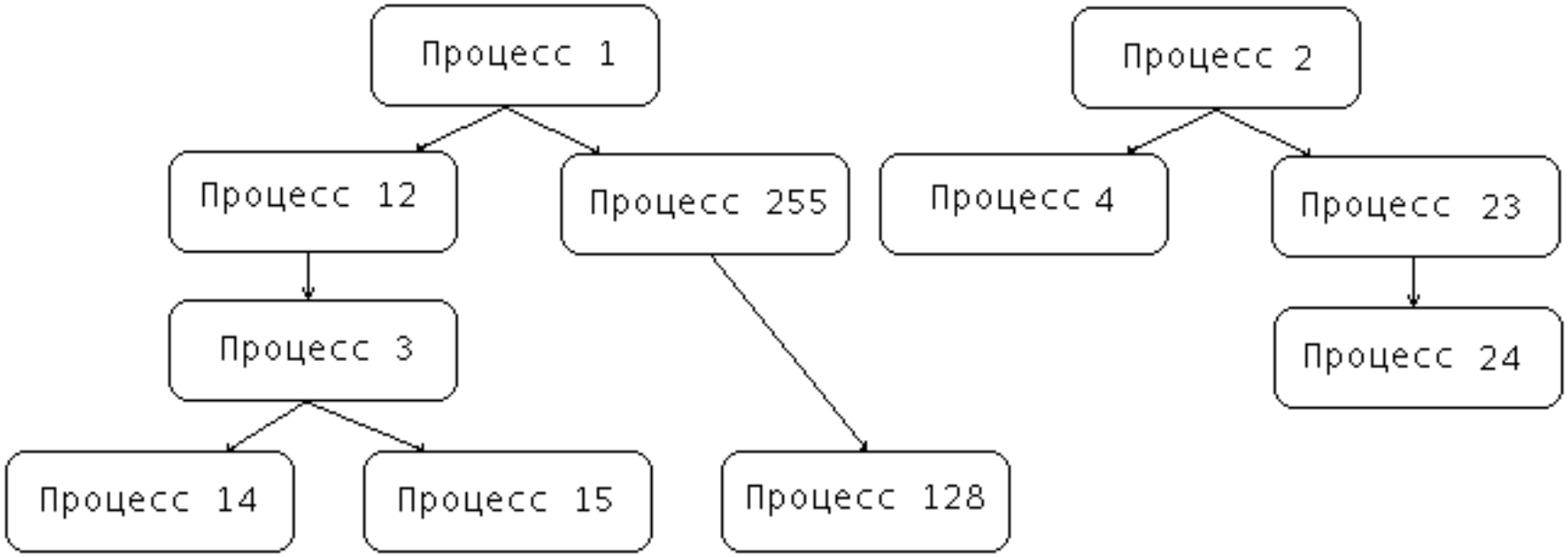
Контекст процесса

- Регистровый (значения регистров и программного счётчика, т.е. адрес команды, которая должна быть выполнена для него следующей)
- Пользовательский (код и данные, находящиеся в адресном пространстве процесса)
- Системный

Системный контекст

- данные, необходимые для планирования использования процессора и управления памятью (приоритет процесса, размер и расположение адресного пространства, общее время использования процессора данным процессом и т. д.);
- учетные данные (идентификационный номер процесса, какой пользователь инициировал его работу и т. д.);
- информацию об устройствах ввода-вывода, связанных с процессом (например, какие устройства закреплены за процессом, таблицу открытых файлов);

Дерево процессов



Системный контекст (на этом семинаре)

- идентификатор пользователя – UID
- групповой идентификатор пользователя – GID
- идентификатор процесса – PID – getpid()
- идентификатор родительского процесса – PPID – getppid()

Создание процесса в UNIX.

- Системный вызов ***fork()***
- Полная копия процесса
- Новые значения только у PID, PPID
- Вызывается один раз, а при успешной работе возвращается два раза

Пример работы с fork

- Пункт 8 из материала 3-го семинара.

```
int a = 0;
fork();
a++;
Выводим getpid, getppid, a;
```

Упражнение 1

```
pid = fork();
if(pid == -1){
        ...
        /* ошибка */
        ...
} else if (pid == 0){
        ...
        /* ребенок */
        ...
} else {
        ...
        /* родитель */
        ...
}
```

Ещё **fork**

- Что выведет программа?

```
printf("Hello, ");
fork();
printf("world!!!");
```

Ещё **fork**

Hello, world!!!

Hello, world!!!

Буферизация вывода…

Завершение процесса

- return в конце функции main()
- функция exit() (сначала сбрасываются все буфера ввода-вывода)
- после завершения процесс не исчезает, а остаётся в состоянии **zombie** (зомби)

Системный вызов wait

- Родительский процесс дожидается завершения дочернего.
- **pid_t wait(int* status);**
- Возвращает id завершившегося ребёнка или -1 при ошибке. В status запишется код завершения.
- **pid_t waitpid(pid_t pid, int* status, int options);**

Что возвращает wait в качестве статуса?

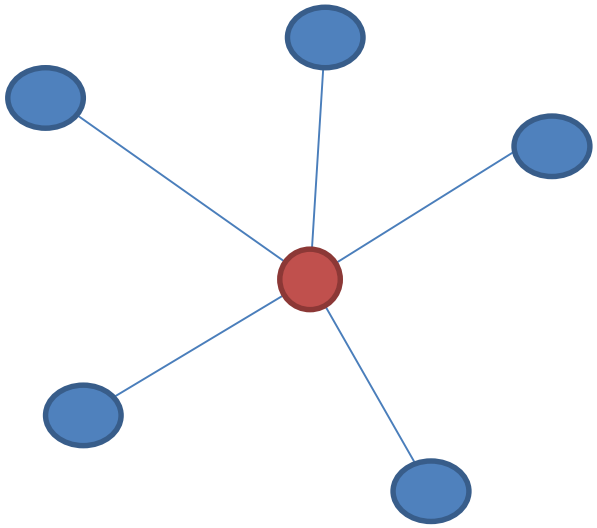
- Для 32-разрядной ОС - 16 бит:
  - в старших 8 битах – то, что вернул дочерний процесс exit(…)
  - в младших 8 битах – причина его завершения
- Побитовые операции:
  - x & 255
  - x % 256
  - x >> 8

Что выведет программа?

```
for (i = 0 ; i < N; ++i)  {
        pid_t pid = fork();
        if (pid == 0)  {
                sleep(i + 1);
                printf("%d\n" , i);
        }
}
```

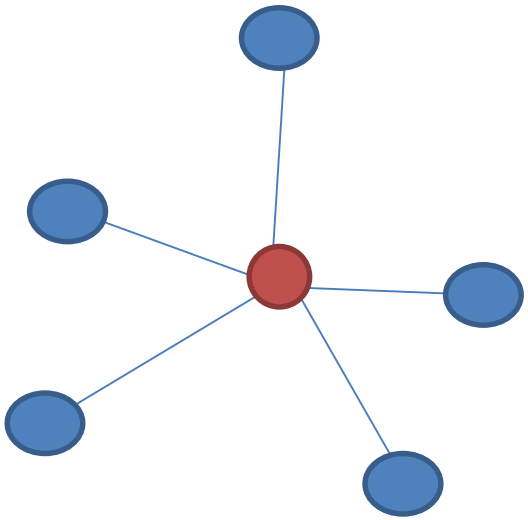
Упражнение 2

- Создать N процессов так, чтобы дерево процессов выглядело:



Упражнение 2

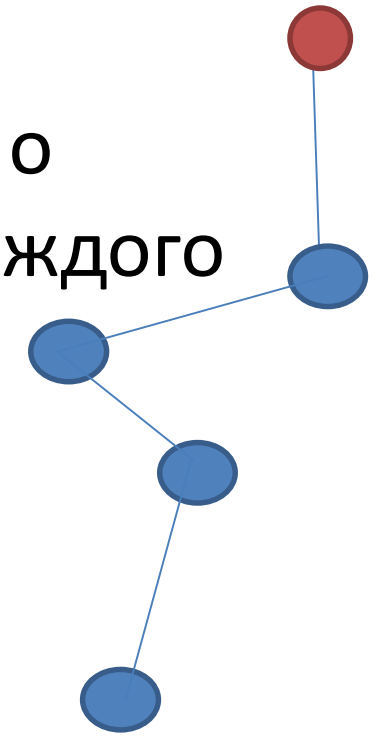
```
for (i = 0 ; i < N; ++i)  {
        pid_t pid = fork();
        if (pid == 0) {
                sleep(i + 1);
                printf("%d\n" , i);
                exit(0);
        }
}
```



Сделать так, чтобы родительский процесс дождался завершения всех дочерних и вывел статусы их завершения.

Упражнение 2б

- Создать N процессов. Каждый i-й процесс создаёт один дочерний i+1 и дожидается его завершения.
- На экран выводится информация о создании нового процесса + от каждого родительского, что его ребёнок завершился.



Упражнение 2б(*) (домашнее, дедлайн – 2 недели)

P.S. Более сложный вариант:

задаётся произвольное дерево и

нужно создать соответствующее

дерево процессов.

Аргументы командной строки

int main(int argc, char *argv[], char *envp[]);

a.out 12 abcd

argc – 3

argv[0] – всегда название программы

envp – список строк вида *переменная=строка, для изменения долгосрочного поведения программы (последний элемент массива - NULL).*

Упражнение 3

- Написать программу, распечатывающую значения аргументов командной строки и параметров окружающей среды для текущего процесса.

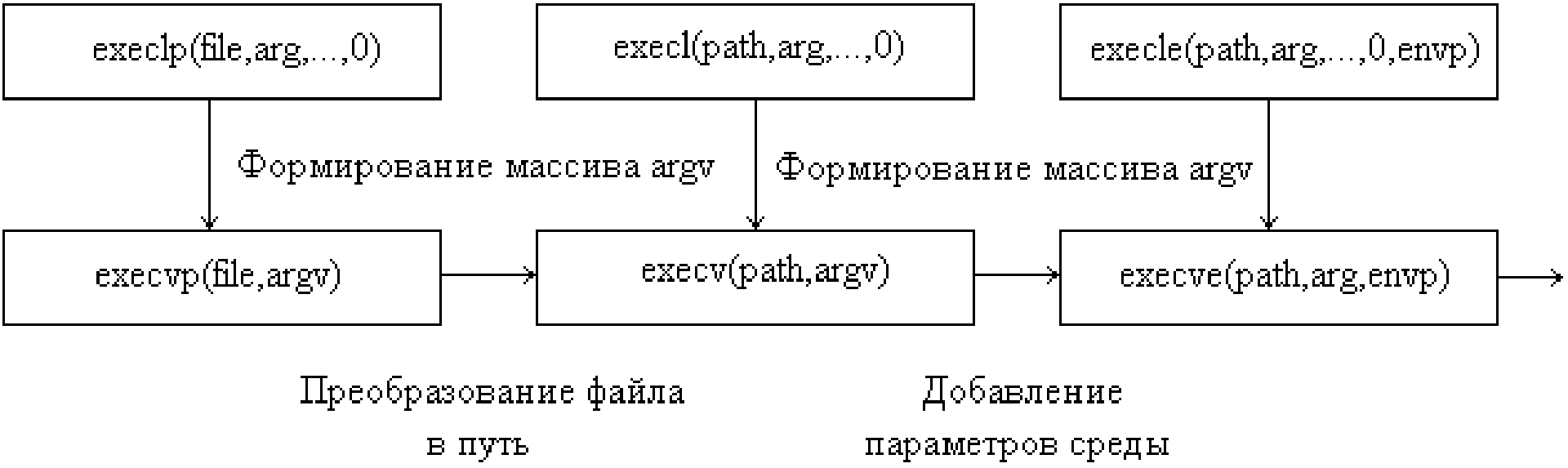
Про синтаксис языка C

- typedef int MyInt;
- ~~typedef MyInt int;~~

Изменение пользовательского контекста процесса

- Какие бывают контексты?
  - Регистровый
  - Системный
  - Пользовательский (в том числе ***исполняемый код***)

Изменение пользовательского контекста процесса



Семейство системных вызовов exec(). Упражнение 4

- Напишите программу «Hello, world!»
- Напишите вторую программу, создающую дочерний процесс и запускающую первую из него.

```c
#include <unistd.h>
#include <stdio.h>

int main()
{
    execlp("ls", "ls", "-al", NULL);
    //execlp("pwd", "pwd", NULL);
    printf("Error\n");
    return 0;
}
```

Упражнение 5

- Есть файл, в котором описаны названия программ и аргументы, с которыми их надо запустить:

  3

  ls –al

  pwd

  echo Hello, world!

- Написать программу, запускающую каждую программу из этого файла.
- На работу каждой программы есть timeout – 5 sec. Не уложившись, команду нужно «убить», написав на экран соответствующее сообщение

(*см. int kill(pid_t pid, int sig)*);

Упражнение 5 (домашнее, дедлайн – 2 недели)

- Также в файле для каждой программы указывается ещё и *время в секундах*, спустя которое ваша программа должна запустить данную программу.
- Преобразовать строку в число можно с помощью функции ***atoi***