

Функции ОС по управлению процессами

- а) Создание и завершение;
- б) Планирование и диспетчеризация;
- в) Переключение;
- г) Синхронизация(взаимодействие);

Планировщик(диспетчер) процессов - та часть операционной системы, которая занимается планированием процессов.

Последовательность создания процесса

1. Создать и проинициализировать блок управления процессом PCB
2. Присвоить новому процессу уникальный идентификатор (занести новую запись в таблицу процессов);
3. Выделить адресное пространство для образа процесса;
4. Загрузить часть команд и данных процесса в оперативную память.
5. Поместить процесс в очередь “готовых” процессов;

Создание новых процессов.

1. Загрузка системы

- При загрузке ОС создаются несколько начальных процессов. В UNIX/linux процессы Sched(pid0) и init(pid1) в новых версиях systemd (pid1).

2. Начальные процессы создают другие процессы :

- одна часть из них взаимодействуют с пользователем;
- вторая часть является фоновыми процессами (в Linux - демоны) не связана с пользователем и выполняет системные функции.

3. Запрос пользователя на создание нового процесса.

Схема вызова fork()

- pid = fork();
- if(pid == -1){ ... /* ошибка */ ... }
- else if (pid == 0){ ... /* дочерний процесс */ ... }
- else { ... /* родительский процесс */ ... }

Завершение процесса

- 1. Обычное завершение (**добровольно**).
- 2. Превышение процессом времени, отведенной для его выполнение (**принудительно**).
- 3. Недостаточный объем памяти (**принудительно**).
- 4. Ошибки в самом процессе (коде программы, - деление на ноль, переполнение разрядной сетки; неверная команда и.т.д.) (**принудительно**).
- 5. Ошибки ввода-вывода (**принудительно**).
- 6. Завершение другим процессом (**принудительно**).
- 7. Вмешательство оператора (**принудительно**).

Завершение процесса

- `#include <stdlib.h>`
- `void exit(int status);`
- Завершает вызвавший функцию процесс.
- Передает родительскому процессу статус завершения (аргумент *status*)
 - 0 – нормальное завершение
 - 1 – завершение с ошибкой
 - Или другое значение от 2 – 255
- Освобождает все ресурсы (пользует контекст, закрывает файлы),
кроме полей дескриптора процесса и системного стека.
- Дочерний процесс переходит в состояние зомби пока родитель не вызовет `wait()`, который очищает дескриптор дочернего процесса.
- Если родительский процесс завершится до завершения всех его дочерних процессов, то новым родительским процессом становится один из процессов группы завершившегося родительского процесса, или процесс `init (systemd)`.

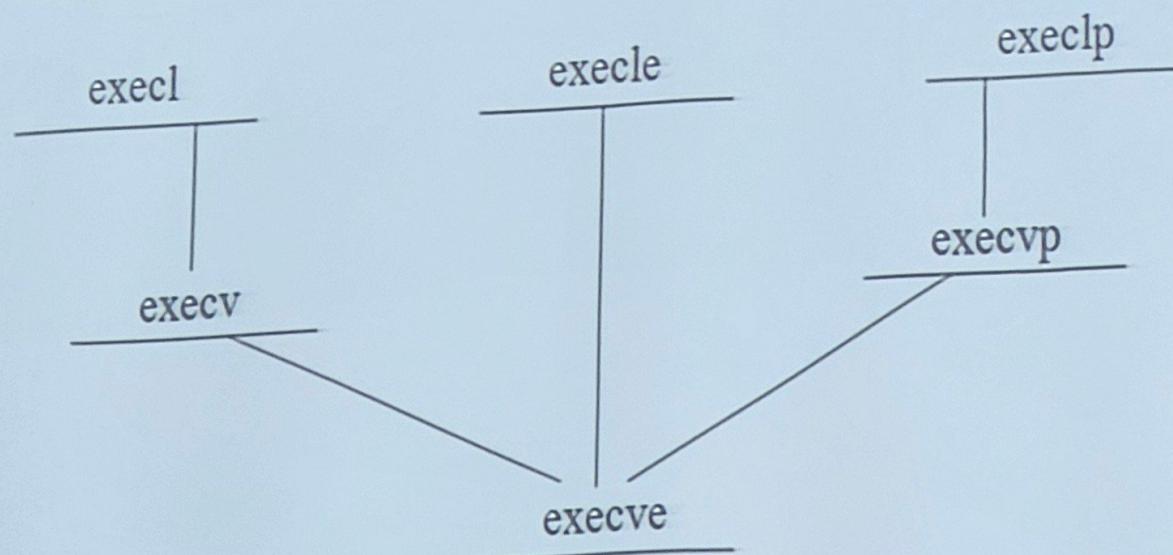
Ожидание завершения процесса

- `#include <sys/wait.h>`
- `pid_t waitpid(pid_t child_pid, unt *status_p, int options);`
- `pid_t child_pid` принимает значения:
 - > 0 - ожидать завершения процесса с конкретным PID;
 - -1 - ожидать завершения любого порождённого процесса;
 - 0 - ожидать завершения любого порождённого процесса, принадлежащего к той же группе, что и родительский процесс;
 - < -1 - ждать любого потомка, чей идентификатор группы процессов равен абсолютному значению *pid*.
- `option` определяет модификацию вызова `waitpid`

Ожидание завершения процесса

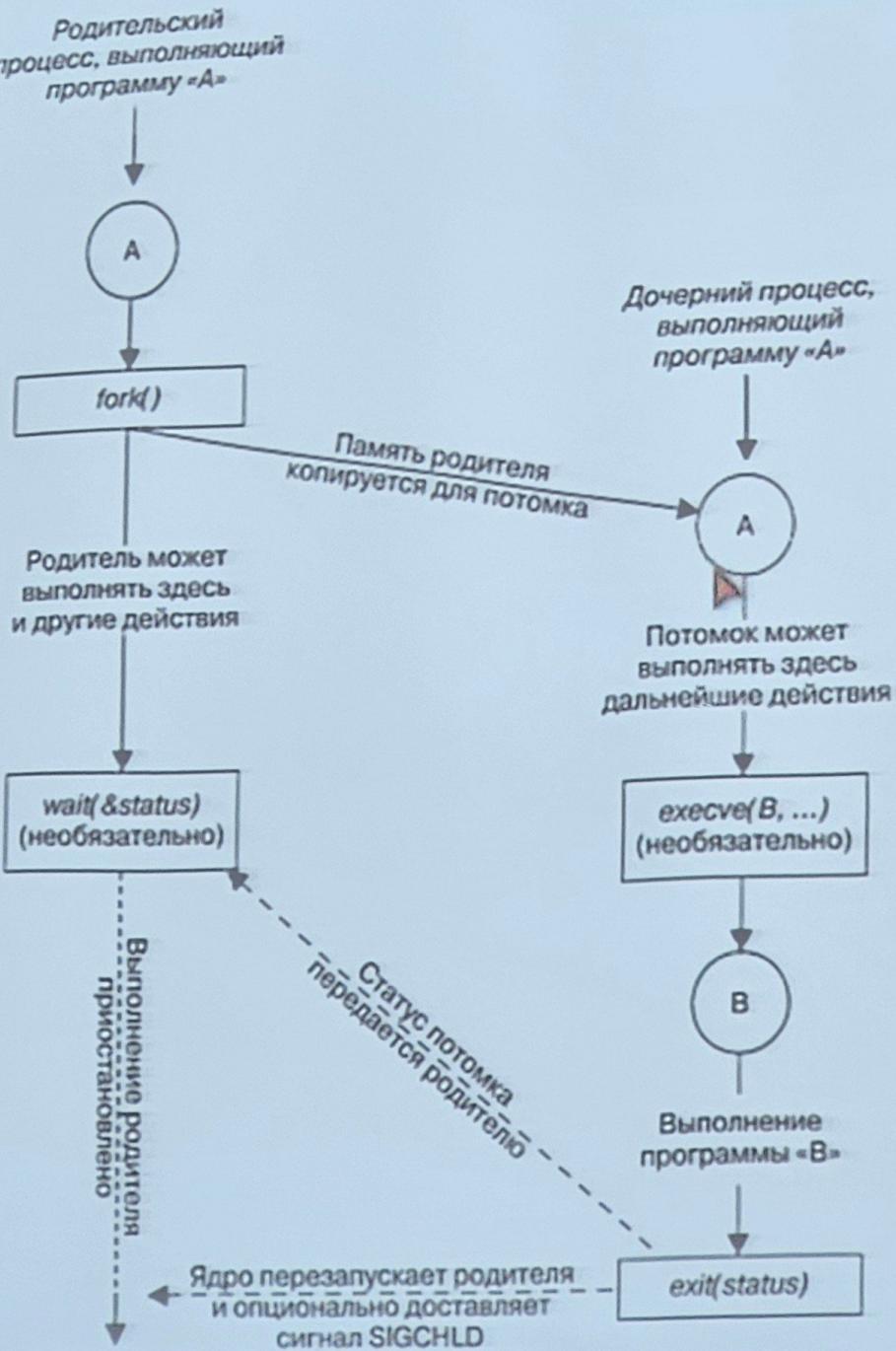
- `#include <sys/wait.h>`
- `pid_t wait(int *status_p);`
- приостанавливает выполнение родительского процесса до тех пор, пока ему не будет послан сигнал, либо пока один из его порождённых процессов не завершится
- Возвращает pid завершенного дочернего сигнала
- `*status_p – хранит статус завершенного дочернего процесса, переданного вызовом exit().`
- Очищает дескриптор дочернего процесса.

Системные вызовы exec

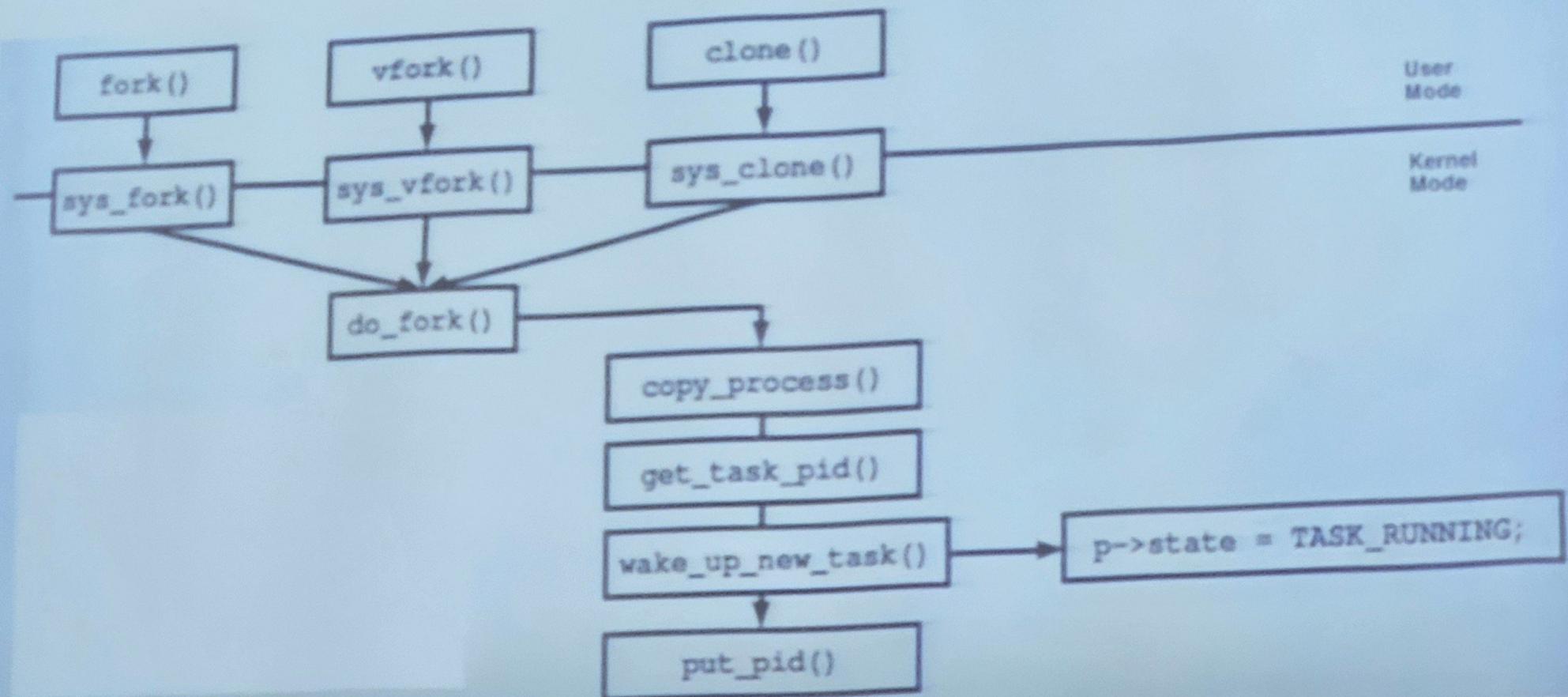


- Системный вызов **exec()** запускает на выполнение программу в созданном системным вызовом fork() процессе
 - **execl("/home/user/test",NULL,NULL);**
 - Вызов функции семейства exec не изменяет системный контекст текущего процесса, загруженная программа унаследует от загрузившего ее процесса его атрибуты:

`fork()`, `exit()`, `wait()`, `exec()`



Создание процесса



`fork()`, `vfork()` , `clone()`

- `fork()` – используя механизм COW, создает два адресных пространства
- `vfork()` - используется для создания новых процессов без копирования таблиц страниц родительского процесса.
 - Это может использоваться в приложениях, критичных к производительности, для создания дочерних процессов, сразу же запускающих `execve()`.
- `clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);`
- передаются флаги, указывающие на то, какие ресурсы должны использоваться совместно с родительским процессом

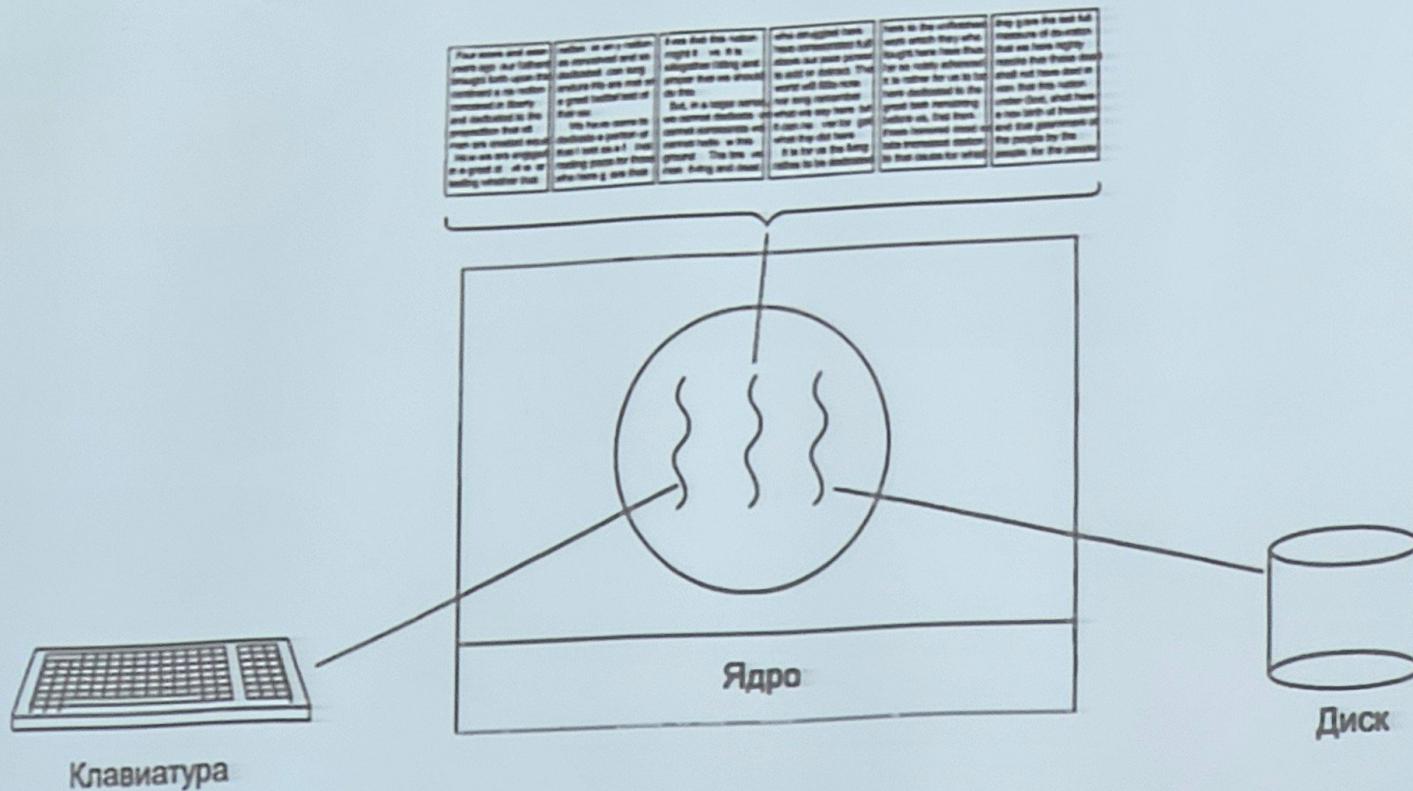
Потоки

- Основная задача процессов – обеспечение режима многозадачности.
- Для повышения эффективности выполнения процесса и загрузки процессора, процесс разбивается на более мелкие смысловые части **называемые ПОТОКАМИ**.
- Потоки по очереди выполняются процессором под управлением ОС, в рамках одного процесса.
- Потоки – единица планирования процессора.

Необходимость создания потоков

- Распараллеливание вычислительного процесса (например при использовании блокирующих функций).
- Повышение скорости выполнения процессов, в которых большая часть времени тратится на ожидание ввода-вывода.
- Не все задачи удобно распараллеливать с помощью процессов (должны быть общие адресные пространства);
- Потоки полезны для систем, имеющих несколько процессоров/ядер, где есть реальная возможность параллельных вычислений (отдельные потоки могут выполняться на отдельных процессорах/ ядрах).

Пример многопоточного приложения



Текстовый процессор использующий три потока

- поток чтения данных с клавиатуры;
- поток вывода на экран ;
- поток сохранения на диск

Процессы и потоки

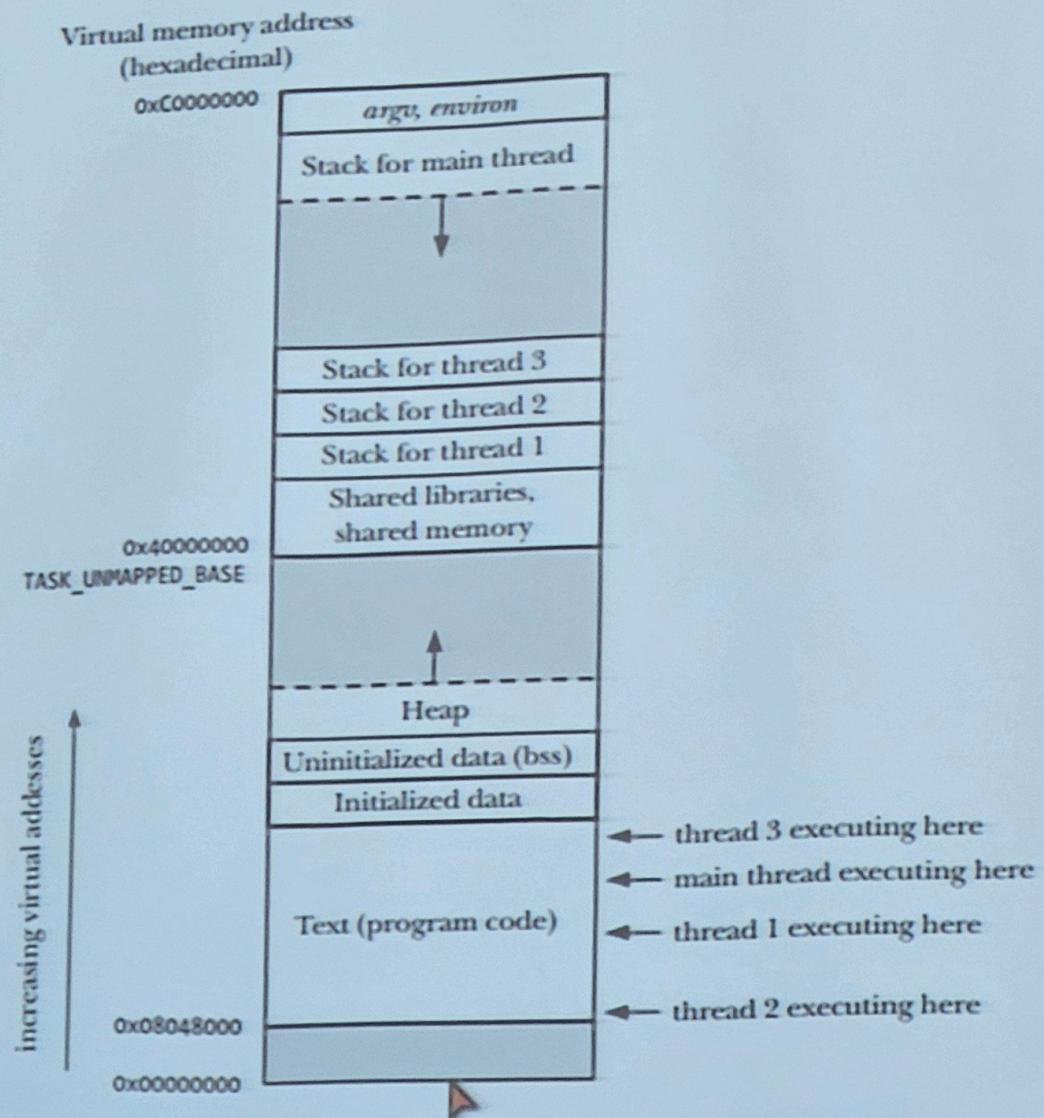
Элементы, присущие каждому процессу

- Адресное пространство
- Глобальные переменные
- Открытые файлы
- Дочерние процессы
- Необработанные аварийные сигналы
- Сигналы и обработчики сигналов
- Учетная информация

Элементы, присущие каждому потоку

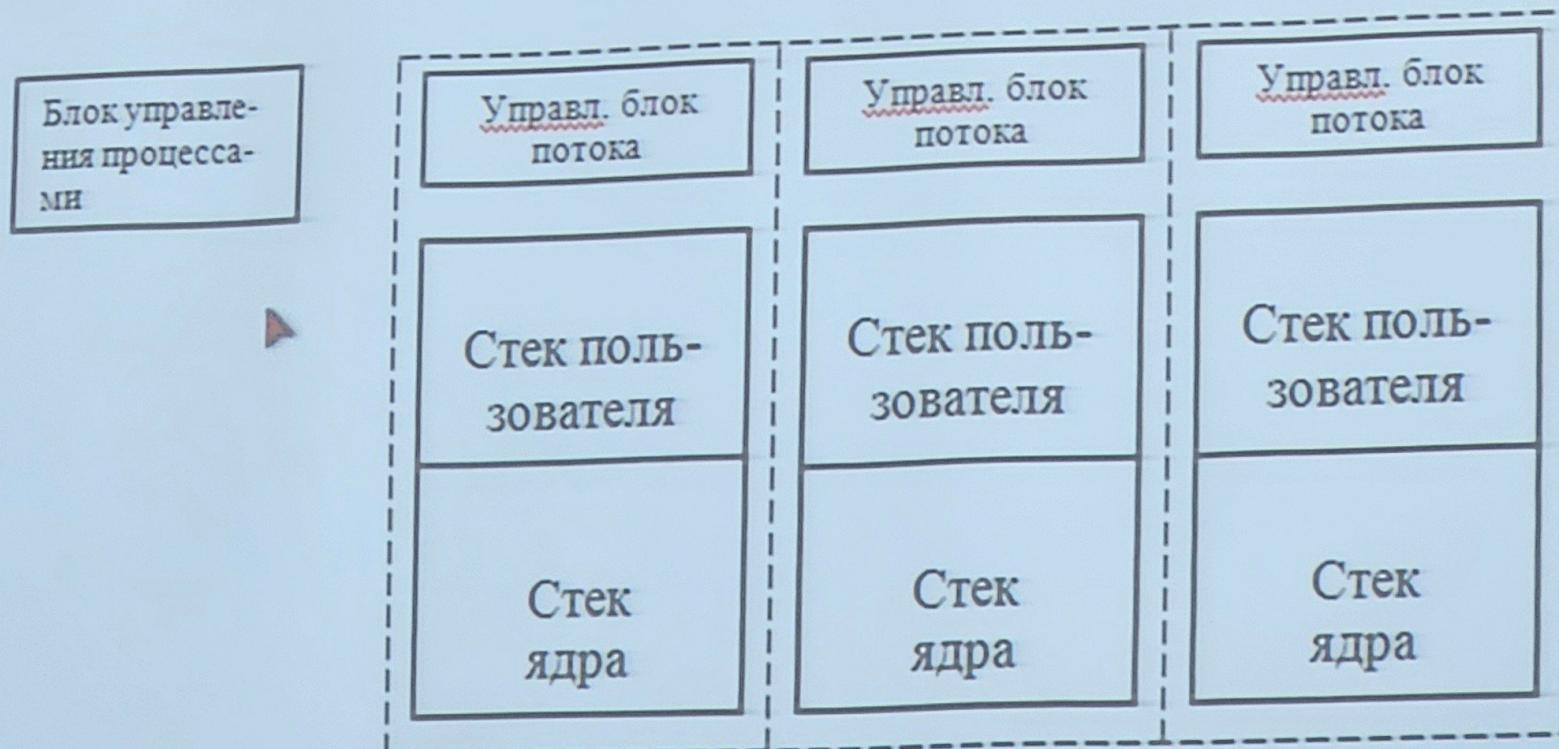
- Счетчик команд
- Регистры
- Стек ядра и стек пользователя
- Состояние

Карта памяти процесса с потоками



Многопоточная модель процесса

Для каждого потока создаётся:
Управляющий блок потока;
Стек потока и стек ядра.



Создание потока

- `#include <pthread.h>`
- `pthread_create(pthread_t *tid, const pthread_attr_t *attr, void *(*function)(void *), void * arg),`
- *tid* – является адресом (переменной) для хранения идентификатор потока, этот идентификатор используется для ссылки на поток в последующих вызовах других функций pthreads.
- *attr* - атрибуты потока : размет стека, приоритет родителя и др. (NULL - атрибуты по умолчанию),
- *function* - указатель на потоковую функцию
- *arg* - указатель на передаваемые данные в поток.

Ожидание завершения потока

```
■ #include <pthread.h>
■ pthread_t tid;
■ int ret;
■ int status;
■ /* ожидание завершения потока "tid" со статусом status */
■ ret = pthread_join(tid, &status);
■ /* ожидание завершения потока "tid" без статуса */
■ ret = pthread_join(tid, NULL);
■ }
```

Завершение потока

- `#include <pthread.h>`
- `int status;`
- `/* выход возвращает статус status */`
- `pthread_exit(&status);`

Краткосрочное планирование процессов

При краткосрочном планировании решаются две задачи:

- При краткосрочном планировании решаются две задачи:

- 1. Когда выбирать процесс на исполнение его процессором;

- 2. Какой процесс выбирать на исполнение из очереди готовых;

Уровни планирования

