# Лекция 27
# Файлы, отображаемые в память

# Особенности mmap

- Гранулярность работы — одна страница памяти (x86 — 4KiB):
  - Размер length должен быть кратен размеру страницы (кроме последней возможно частично заполненной страницы)
  - Смещение в файле offset должно быть кратно одной странице
  - Файл не должен быть пустым
- Хвост файла (< размера страницы) отображается на целую страницу, но размер не меняется
  - Чтение данных после конца файла вернет 0
  - Запись данных после конца файла не попадет в файл

# Типичное использование

- MAP_SHARED — если несколько процессов отобразят файл, они будут видеть изменения друг друга, измененное содержимое будет сохранено в файле — реализация общей памяти (shared memory) процессов
- MAP_PRIVATE — содержимое файла доступно для чтения, при модификации содержимого другие процессы не увидят изменений, они не будут сохранены в файле — отображение исполняемых файлов в память

# Типичное использование

- MAP_SHARED | MAP_ANONYMOUS — отображенная память доступна самому процессу и порожденным им процессам (они видят изменения) — реализация общей памяти для родственных процессов
- MAP_PRIVATE | MAP_ANONYMOUS — содержимое памяти видимо только для одного процесса — дополнительная память в адресном пространстве процесса

# Demand paging

- Логическое отображение – то, как должно быть (/proc/self/maps)
- Физическое отображение – то, как есть на самом деле (/proc/self/pagemap)
- Если страница есть в логическом отображении, но нет в физическом, то при первом обращении к этой странице ядро выделит новую физическую страницу ОЗУ или возьмет существующую и добавит ее в физическое отображение

# Demand paging

- Процесс начинает работу с настроенным логическим отображением и пустым физическим отображением (см. VmVSZ)
- Постепенно по мере обращения к страницам заполняется физическое отображение (см. VmRSS)
- Если к странице не было обращений, она не будет загружена в физическую память (ОЗУ)

# Страничная подкачка

- Физические страницы – ценный ресурс, в какой-то момент их может не хватить
- Ядро попытается освободить физические страницы для выполнения текущего запроса
- Если физическая страница соответствует отображению файла в память и не модифицировалась, она просто освобождается
- Страницы MAP_SHARED и модифицированные (dirty) сохраняются в файл и освобождаются
- Прочие страницы сохраняются в файл (раздел) страничной подкачки – swap file: стек, куча и т. п.

# Типы страниц в памяти

- Выгружаемые (страница может быть выгружена в область подкачки)
- Невыгружаемые (locked) — должны находиться в ОЗУ
- Процесс может пометить часть страниц как невыгружаемые (системный вызов mlock)
- Непривилегированный — макс. 32 KiB
- Все страницы ядра — невыгружаемые

# Резервирование swap

- Место в файле подкачки может быть зарезервировано при создании страницы, которую **может быть** потребуется сохранить в swap
  - Стек, куча
  - Все файлы, отображаемые в память с MAP_PRIVATE (т. е. исполняемые файлы и библиотеки) для каждого процесса
- В Linux место в файле подкачки выделяется при сохранении страницы в файле подкачки
- Возможны ситуации overcommit memory

# Расположение виртуальной страницы

- В физической памяти (ОЗУ) – после первого обращения к ней и пока она не выгружена
- В файле (при отображении файла в память) – подгрузится в ОЗУ при обращении к ней
- В области подкачки (swap file) – подгрузится обратно в ОЗУ при обращении к ней
- НИГДЕ – будет выделена в ОЗУ при обращении к ней (overcommited pages)

# MAP_PRIVATE

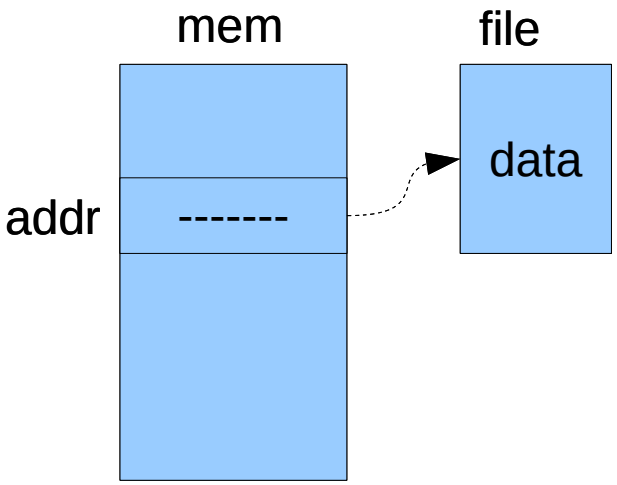
- Флаг MAP_PRIVATE в mmap – приватное отображение
- Изначально содержимое страницы берется из файла
- Но если страница модифицирована, то она “отвязывается” от файла
- Изменения модифицированных страниц обратно в файл не попадут

# Copy-on-write

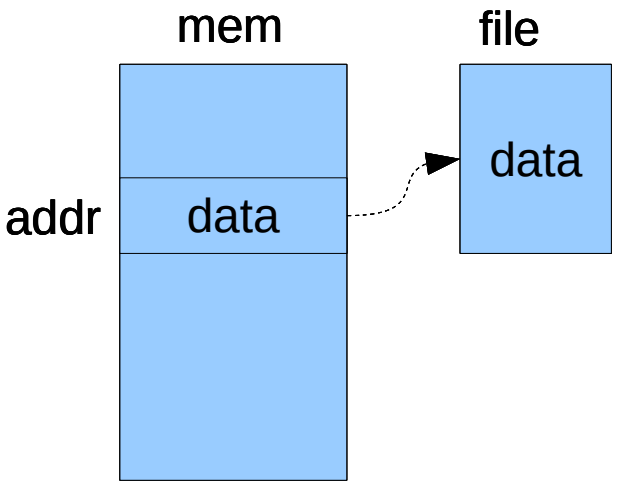
- Механизм оптимизации копирования страниц
- При обычном механизме копия страницы в физической памяти создается немедленно
- При механизме copy-on-write создание копии страницы откладывается до первой записи в страницу

# Copy-on-write

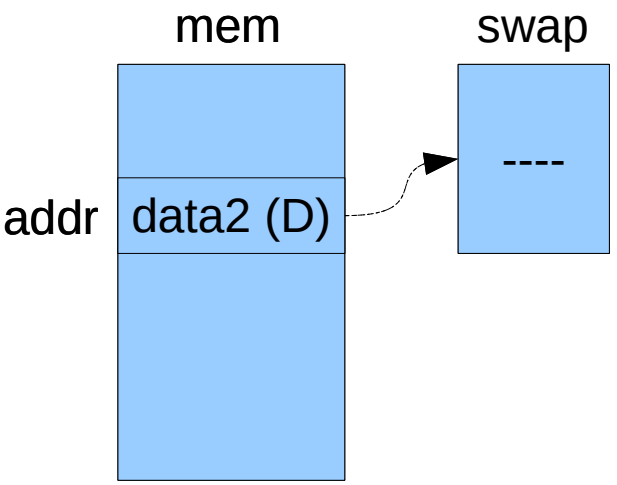
```
fd = open("file", O_RDWR, 0);
addr = mmap(0, 4096, PROT_READ|PROT_WRITE,MAP_PRIVATE,fd, 0);
```




**При создании отображения страница в памяти помечена как отсутствующая, но отображенная на соответствующий файл**



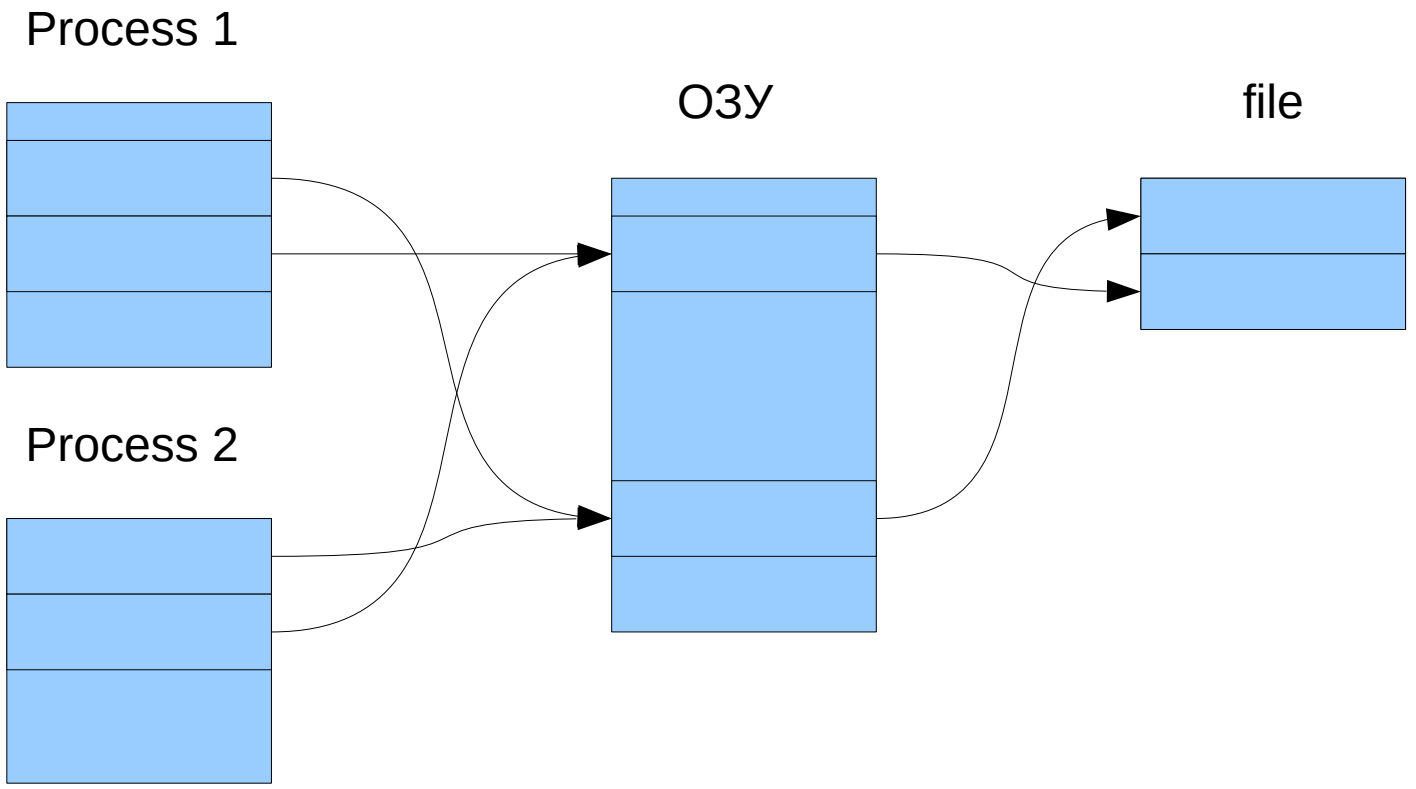

**При чтении содержимое страницы подгружается из файла, страница помечается как «только для чтения»**




**При записи в страницу выделяется место в области подкачки, при необходимости создается копия страницы в ОЗУ, отображение переключается на swap**

# Разделение страниц между процессами

- Процессы, выполняющие отображение одного и того же файла, разделяют физические страницы ОЗУ



# Необеспеченная память (memory overcommit)

- Стратегия выделения copy-on-write и выделение памяти по требованию приводят к тому, что хотя страница присутствует в логическом отображении, невозможно настроить физическое отображение (нет свободных физических страниц, исчерпан swap)
- Надо попытаться удовлетворить запрос этого процесса за счет других процессов
- Необходимо снять с выполнения какой-нибудь процесс и таким образом освободить память (OOM killer)

# OOM Killer

- Задача: выбрать минимальное число процессов, чтобы освободить максимальный объем памяти, но нанести минимальный ущерб системе
- Для каждого процесса вычисляется oom_score (/proc/${PID}/oom_score)
  - Чем больше RSS и Swap usage — тем хуже
  - Привилегированные процессы лучше обычных
  - Пользователь может задать поправку: /proc/${PID}/oom_score_adj

# Загрузка файла на выполнение

- ELF-файл имеет структуру, оптимизированную для отображения файла mmap
- Секция кода (.text) отображается PROT_READ | PROT_EXECUTE, MAP_PRIVATE
- Константные данные (.rodata): PROT_READ, MAP_PRIVATE
- Данные (.data): PROT_READ | PROT_WRITE, MAP_PRIVATE
- Секции .text и .rodata у всех процессов, запущенных из одного файла, будут использовать одни и те же физические страницы памяти

# Разделяемые библиотеки

- Позволяют избежать дублирования кода в процессах (например, все процессы имеют общую реализацию printf)
- Делает возможным разделять код библиотек между процессами разных исполняемых файлов (при статической компоновке реализация printf может располагаться по разным адресам, что делает невозможным разделение)
- Облегчают обновление ПО

# Загрузка разделяемых библиотек

- ELF-файл содержит секцию .interp. Эта секция содержит путь к «интерпретатору» - /lib/ld-linux.so.2 — загрузчик динамических библиотек
- Загрузчик проходит по списку зависимостей библиотек, находит их в файловой системе и загружает в память, рекурсивно, пока все зависимости не будут удовлетворены
- Загрузка каждой библиотеки аналогична загрузке исполняемого файла (mmap)
- Но! Одна и та же библиотека может быть загружена в разных процессах по разным адресам

# Позиционно-независимый код

- В разделяемой библиотеке секция кода позиционно-независима, то есть страницы, занимаемые кодом, идентичны независимо от их виртуального адреса в каждом процессе
- Требуется одна копия кода в страницах физической памяти, на которую будут отображаться страницы виртуальной памяти разных процессов
- Секции разделяемой библиотеки, индивидуальные для каждого процесса (GOT, .data), малы по сравнению с секцией кода
- Огромная экономия физической памяти!

# Процессы

# Процессы

- Процесс — программа в состоянии выполнения.
- Процесс — субъект распределения ресурсов в ОС.
- Процесс — единица планирования ОС.
- Типы процессов:
  - «Тяжелые» (обычные процессы)
  - «Легковесные», нити, потоки, threads (несколько нитей исполняются в общем адресном пространстве)

# Атрибуты процесса в UNIX

- Атрибуты памяти
  - Таблицы страниц виртуального адресного пространства процесса
  - Разделяемые и неразделяемые страницы памяти
  - Отображения файлов в память
  - Стек режима ядра

# Атрибуты процесса

- Файловая система:
  - Таблица файловых дескрипторов
  - Текущий каталог
  - Корневой каталог
  - Umask
- Параметры планирования
  - Динамический и статический приоритеты
  - Тип планирования, приоритет реального времени

# Атрибуты процесса

- Регистры ЦП
- Командная строка, окружение
- Диспозиции обработки сигналов
- Счетчики потребленных ресурсов
- Идентификаторы пользователя:
  - uid, gid — реальные пользователь и группа
  - euid, egid — эффективные (то есть действующие в данный момент) пользователь и группа

# Идентификация процессов

- pid — идентификатор процесса, положительное целое число [1...]
  - 1 — процесс init
- ppid — идентификатор родительского процесса (если родитель процесса завершается, родителем становится init)
- pgid — идентификатор группы процессов (группа процессов выполняет одно задание)
- sid — идентификатор сессии (сеанса работы)

# Получение идентификаторов процесса

- getpid() -  идентификатор процесса
- getppid() - идентификатор родительского процесса

# Создание процесса

- Системный вызов fork — единственный способ создания нового процесса

```
int fork(void);
```

- При ошибке (нехватке ресурсов) возвращается -1
- Создается новый процесс — копия исходного
  - Родителю возвращается pid сына
  - Сыну возвращается 0

# Атрибуты создаваемого процесса

- Практически все атрибуты копируются, страницы памяти копируются в режиме copy-on-write
- Не копируются:
  - Идентификатор процесса (создается новый)
  - Идентификатор родительского процесса
  - Сигналы, ожидающие доставки
  - Таймеры
  - Блокировки файлов

# Пример работы fork

```
int main(void)
{
  int pid;
  if ((pid = fork()) < 0) {
    fprintf(stderr,"Err\n");
    return 1;
  } else if (!pid) {
    printf("son: %d\n",
           getpid());
  } else {
    printf("parent: %d\n",
           getpid());
  }
  printf("both\n");
  return 0;
}
```

- Возможный вывод:

```
son: 12311
parent: 12305
both
both
```

# Выполнение fork

```
pid = fork();
```

**Родитель:**

```
movl $__NR_fork,%eax
int  0x80
movl %eax,-4(%ebp)
```

**eip**

**eax=12311**

**eip**

```
movl $__NR_fork,%eax
int  0x80
movl %eax,-4(%ebp)
```

**Сын:**

```
movl $__NR_fork,%eax
int  0x80
movl %eax,-4(%ebp)
```

**eip**

**eax=0**

# Побочные эффекты копирования адр. простр.

```
int main(void)
{
  int pid;
  printf("Hello, ");
  if ((pid = fork()) < 0) {
    fprintf(stderr,"Err\n");
    return 1;
  } else if (!pid) {
    printf("son\n");
  } else {
    printf("parent\n");
  }
  return 0;
}
```

Вывод программы?

# Побочные эффекты копирования адр. простр.

```
int main(void)
{
  int pid;
  printf("Hello, ");
  if ((pid = fork()) < 0) {
    fprintf(stderr,"Err\n");
    return 1;
  } else if (!pid) {
    printf("son\n");
  } else {
    printf("parent\n");
  }
  return 0;
}
```

- Возможный вывод:

```
Hello, parent
Hello, son
```

Или

```
Hello, son
Hello, parent
```

**При fork() копируются структуры данных stdout, находящиеся в адресном пространстве процесса.**

# Завершение работы процесса

- Нормальное: процесс завершает выполнение сам с помощью exit() или _exit() или return из функции main
- При получении сигнала, вызывающего завершение
  - kill -TERM ${pid} # завершить процесс pid
- При получении сигнала, вызывающего завершение работы и запись образа памяти
  - Обращение по нулевому адресу — Segmentation fault (core dumped)

# Нормальное завершение процесса

void exit(int code);

- Библиотечная функция — структуры данных стандартной библиотеки очищаются

void _exit(int code);

- Системный вызов — структуры стандартной библиотеки не очищаются

# Пример

```
int main(void)
{
  printf("Hello");
  exit(0);
}
```

- Вывод:

```
Hello
```

```
int main(void)
{
  printf("Hello");
  _exit(0);
}
```

- Вывод:

# Код завершения

- Код завершения — целое число, 1 байт
- Параметр функций exit и _exit преобразовывается: code & 0xff
- Код завершения 0 сигнализирует об успешном  завершении процесса
- Ненулевой код завершения сигнализирует об ошибочном завершении процесса
- Переменная $? shell содержит код завершения процесса

# Действия при завершении процесса

- Освобождение страниц памяти, использованных процессом
- Закрытие всех открытых дескрипторов файлов
- Освобождение прочих ресурсов, связанных с процессом, кроме статуса завершения и статистики ресурсов
- Если у процесса есть потомки, родителем потомков назначается процесс 1
- Родителю процесса посылается сигнал SIGCHLD

# Ожидание завершения процесса

- Системные вызовы семейства wait* - ожидание завершения сыновних процессов

int wait(int *pstatus);

- Ожидание завершения любого из сыновних процессов
- Возвращается pid завершившегося процесса или -1 при ошибке
  - ECHILD — нет сыновних процессов
  - EINTR — ожидание прервано получением сигнала

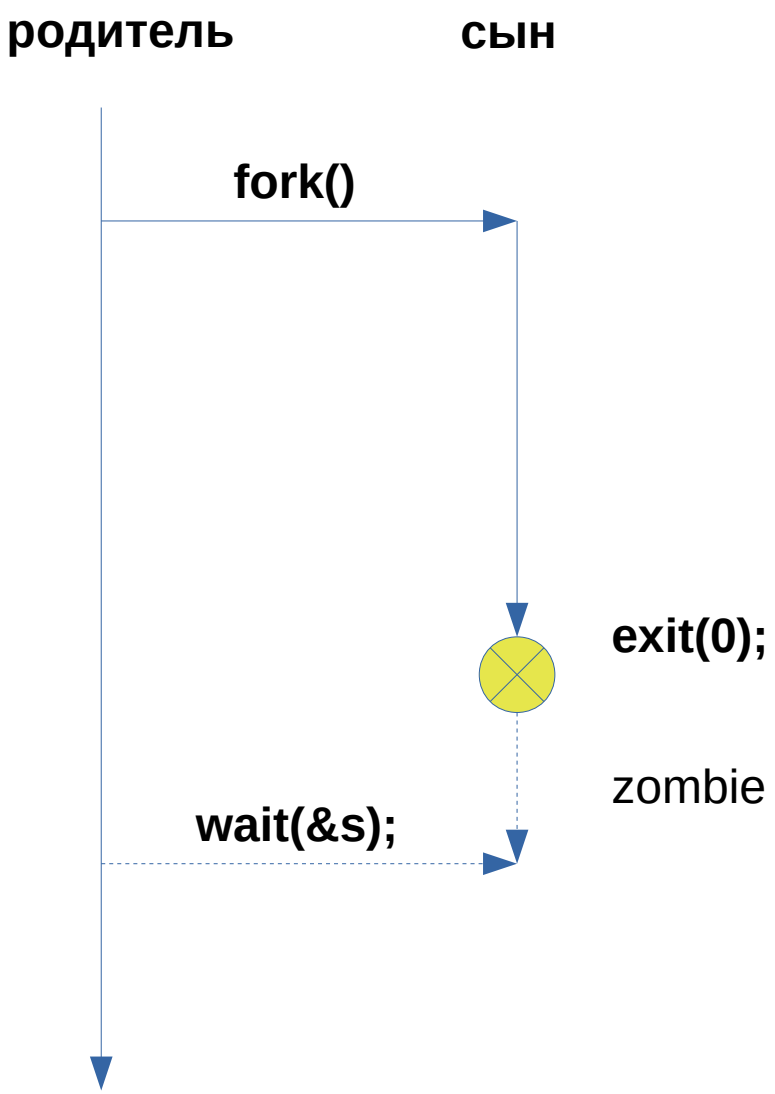
# Слово состояния процесса

- WIFEXITED(status) — процесс завершился нормально?
- WEXITSTATUS(status) — код завершения процесса
- WIFSIGNALED(status) — процесс завершился из-за сигнала?
- WTERMSIG(status) — сигнал, приведший к завершению процесса
- WCOREDUMP(status) — был сгенерирован образ памяти (core dump)?

# Пример

```
int pid, status;
  // запускаем 10 процессов
  for (i = 0; i < 10; ++i) {
    pid = fork();
    if (!pid) {
      // в сыне выполняем действия
      srand(time(0) + getpid());
      usleep(10000*(rand() %20 + 1));
      _exit(i);
    }
  }
  // здесь код отца
  for (i = 0; i < 10; ++i) {
  // while ((pid = wait(&status)) > 0) {
    pid = wait(&status);
    printf("pid: %d, завершился с кодом: %d\n",
           pid, WEXITSTATUS(status));
  }
```

# Процессы-зомби




- Запись в таблице процессов не уничтожается, пока родительский процесс не прочитает статус завершения процесса с помощью wait
- От момента завершения до уничтожения записи процесс находится в состоянии «зомби»
- Зомби-процесс не потребляет системных ресурсов, однако занимает место в таблице процессов

# Системный вызов waitpid

int waitpid(int pid, int *pstatus, int flags);

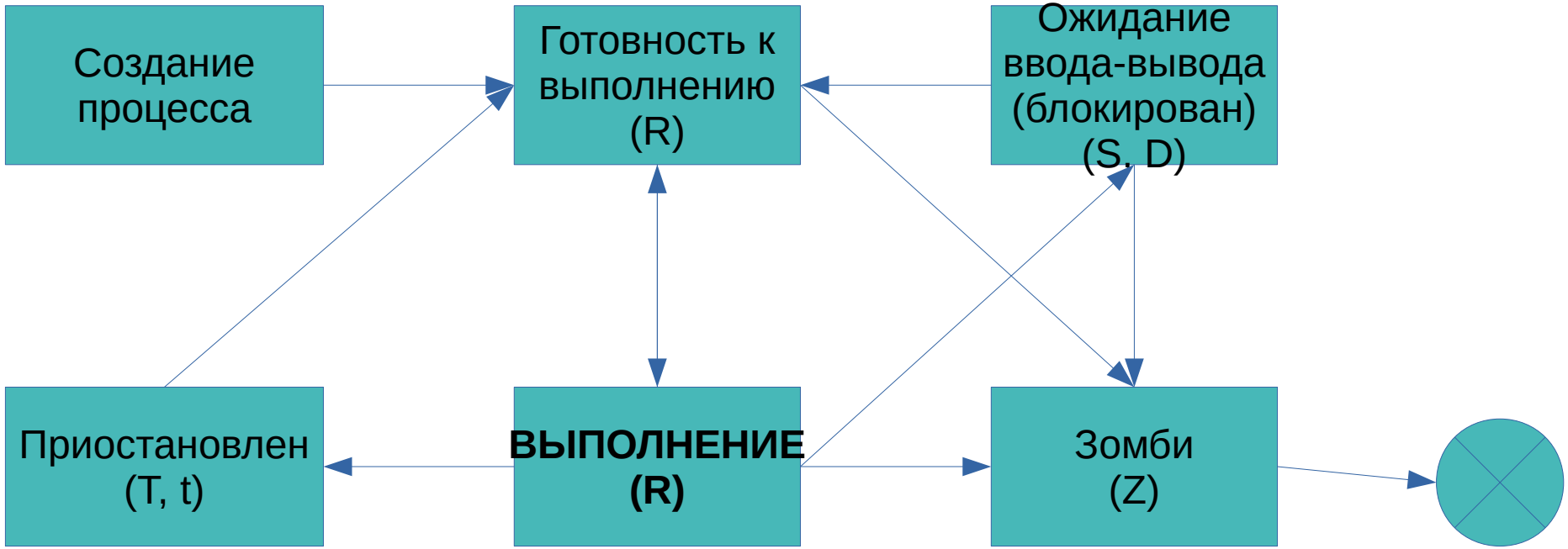
- Допустимые значения pid сыновних процессов
  - < -1 — любой процесс из указанной группы
  - -1 — любой процесс
  - 0 — любой процесс из текущей группы
  - > 0 — указанный процесс
- Допустимые значения flags
  - WNOHANG — не блокировать процесс, если нет завершившихся сыновних процессов — в этом случае возвращается 0

# Системный вызов waitpid

int waitpid(int pid, int *pstatus, int flags);

- Допустимые значения pid сыновних процессов
  - < -1 — любой процесс из указанной группы
  - -1 — любой процесс
  - 0 — любой процесс из текущей группы
  - > 0 — указанный процесс
- Допустимые значения flags
  - WNOHANG — не блокировать процесс, если нет завершившихся сыновних процессов — в этом случае возвращается 0

# Жизненный цикл процесса



# Состояния процесса

- Готов к выполнению/выполняется (TASK_RUNNING) — ядро берет процессы из очереди готовых к выполнению и ставит на выполнение на процессор, обозначается 'R'
- Ожидает ввода-вывода (спит)
  - TASK_INTERRUPTIBLE ('S') - обычное ожидание, ожидание может быть прервано сигналом (в частности, завершения процесса)
  - TASK_UNINTERRUPTIBLE ('D') — непрерываемое ожидание, процесс не может быть убит. Обычно используется при операциях ввода-вывода с устройством, на котором размещена файловая система (диски)

# Состояния процесса

- TASK_ZOMBIE ('Z') — ожидание опроса состояния завершения процесса
- TASK_STOPPED ('T') — процесс приостановлен, то есть не ждет i/o, но и не готов к выполнению. Либо приостановлен пользователем (Ctrl-Z с терминала), либо сигналом (SIGSTOP), либо попытался считать с терминала в фоновом режиме
- TASK_TRACED ('t') — процесс отлаживается

# Состояния процессов

sleep(10);

Из состояния 'R' процесс переводится в состояние 'S', через 10 с процесс будет разбужен и переведен в состояние 'R'

read(0, buf, sizeof(buf)); // чтение из stdin

Если символов в буфере ввода нет, процесс переводится в состояние 'S' и будет оставаться в нем, пока не появятся данные, после чего будет переведен в состояние 'R'

read(fd, buf, sizeof(buf)); // чтение с диска

На время обмена с диском процесс находится в 'D'