Операционные системы
Часть 2. Процессы и потоки

Клименков С.В.
Версия 1.0.0
30.08.2020
vk.com/serge_klimenkov

tuneit

Основы процессов

2.1

- Вычисления
- Процесс. Характеристики процесса
- Разделение ресурсов
- Состояния процесса

Обобщение вычислений в ОС

- Компьютерная система состоит из набора ресурсов:
  - Процессоры, память, устройства ввода-вывода, таймеры и пр.
- Приложения решают практическую задачу
  - Входные данные → Обработка → Выходные данные
- ОС находится между оборудованием и приложениями
  - Обеспечивает функциональный, безопасный и единообразный интерфейс
  - Предоставляет абстрактное представление ресурсов

Итак, процесс это:

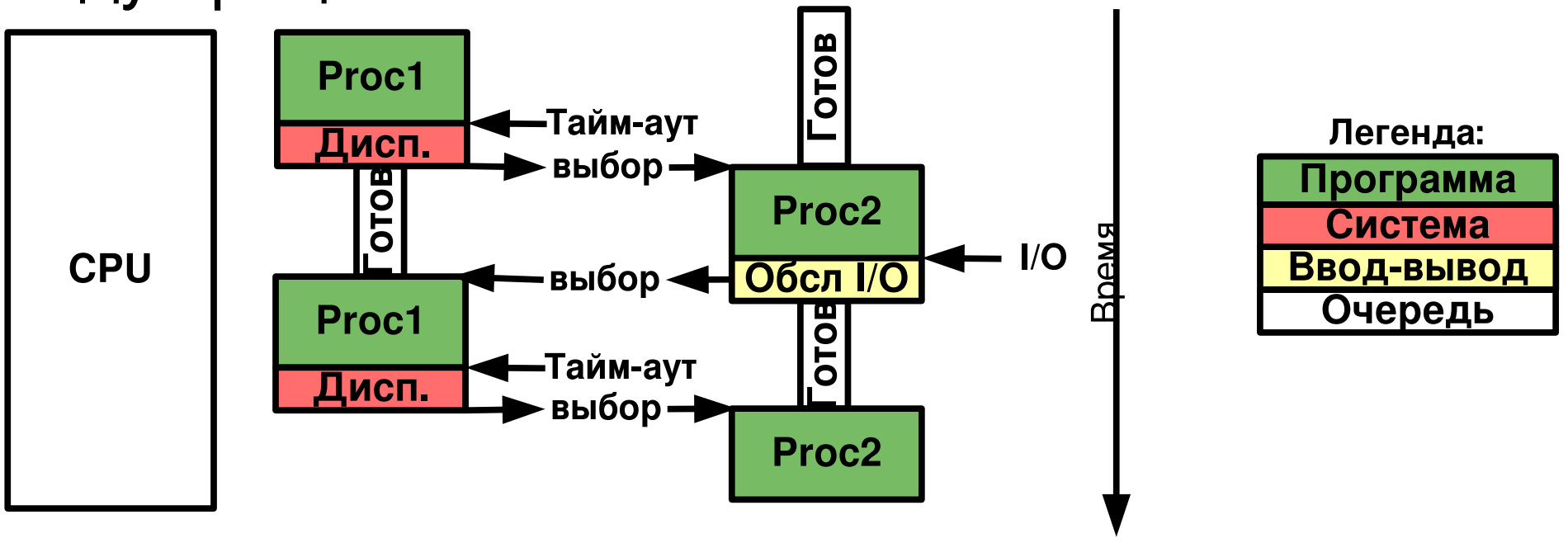
- Выполняемая программа
- Экземпляр программы, выполняющейся на компьютере
- Сущность, которая может быть назначена процессору и выполнена на нем
- Единица активности, характеризуемая выполнением последовательности команд, текущим состоянием и связанным с ней множеством системных ресурсов.

Характеристики процесса в момент выполнения

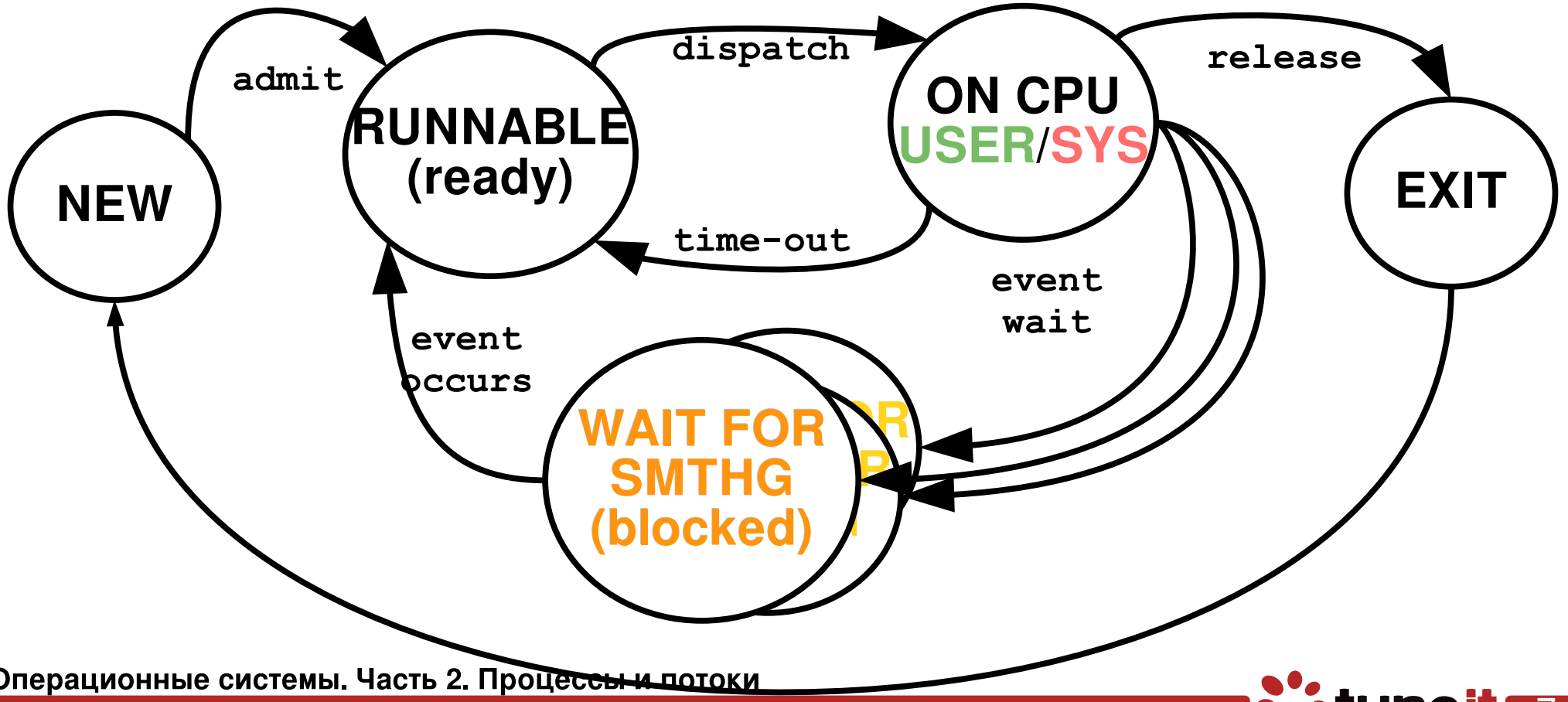
- Уникальный идентификатор
- Состояние (выполнение, очередь, ожидание…)
- Приоритет по отношению к другим процессам
- Счетчик команд
- Указатели на области памяти процесса
- Контекст процесса (регистры, user/kernel)
- Статус ввода-вывода
- Счетчики системных ресурсов
- Права доступа процесса
- ...

Состояние процесса. Разделение ресурсов

- Операционная система разделяет ресурсы между процессами



Модель процесса с 5-ю состояниями



New state

- Процесс создан, но еще не размещен в очереди процессов, готовых к исполнению
  - Создан PCB, но, например, память не выделена
- Типичные причины создания процессов:
  - Вход в систему в интерактивном режиме
  - Запуск скриптов пакетного задания
  - Запуск обработчика сервиса
  - Программа пользователя создает новую единицу работы

Exit state

- Процесс не может продолжить выполнение
  - Структуры процесса все еще существуют
- Причины попадания в состояние:
  - Нормальное завершение (вызов exit)
  - Превышение лимитов на время выполнения
  - Недостаток памяти
  - Ошибки границ и защиты памяти
  - Арифметическая ошибка
  - Ошибка ввода-вывода
  - Неправильная или привилегированная инструкция
  - Команда оператора или ОС
  - Завершение или запрос родительского процесса

Runnable state

- Процесс обладает всеми ресурсами для выполнения, но нет возможности исполнятся
  - Все CPU заняты другими задачами
- Причины нахождения в состоянии:
  - Низкий приоритет, по сравнению с другими процессами
  - Ожидание освобождения CPU
  - Закончился квант времени

Running state

- Команды приложения и ОС выполняются на процессоре
- Процесс остается в этом состоянии если:
  - Не истек квант времени
  - Ожидание на спин-блокировке
  - В runnable состоянии нет процессов с более высоким приоритетом
  - Обслуживание высокоприоритетных прерываний
  - Нет блокирующих вызовов (ввод-вывод, ожидание блокировки)

Wait (blocked) state

- Ожидание событий ОС, освобождения блокировки
- Состояние продолжается до тех пор, пока:
  - Освободится блокировка
  - Придет сообщение ОС о наступлении ожидаемого события (завершился ввод-вывод и пр.)
- Процесс не расходует ресурсов CPU
- Процесс может находится в ожидании неопределенно долго (вспомним deadlocks)

Процессы — suspending

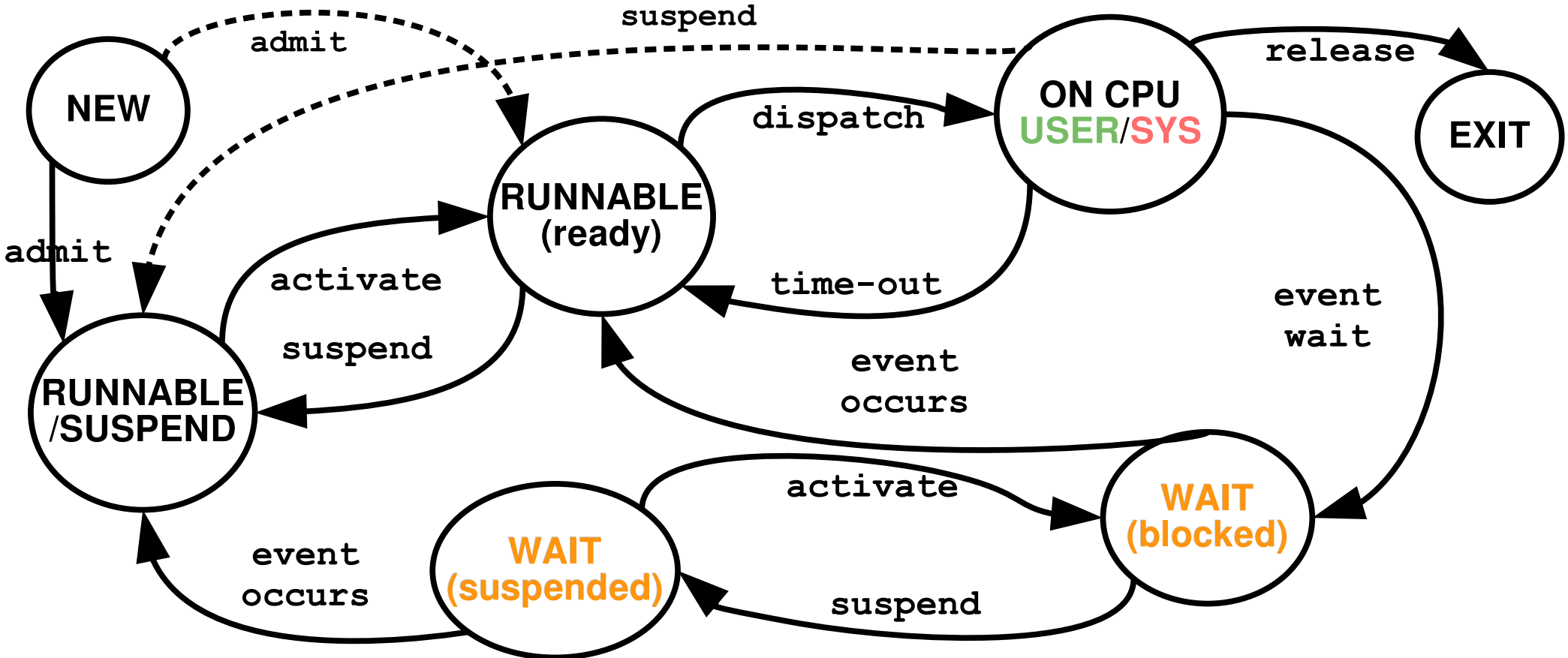
2.2

- Paging и Swapping
- Дополнительные состояния процесса
- Состояния приостановки
- Причины приостановки

Paging/Swapping

- Основной памяти всегда мало =(
  - Программисты как только видят больше памяти, стараются ее максимально «использовать»
  - Большое количество запущенных процессов
- Давайте поместим блокированный процесс на диск и освободим основную память для других процессов!
  - Нужно организовать область подкачки процессов на диске
- Paging (пейджиг) — выгрузка (и загрузка) неиспользуемых страниц процесс на диск
- Swapping (свопинг) — выгрузка всего процесса, кроме критически важных для ядра структур управления

Модель процесса с 7-ю состояниями



Wait/Suspended state

- Процесс приостановлен и выгружен в область подкачки.
- Причины попадания в состояние:
  - Длительное ожидание событий операционной системы
  - Недостаток памяти (зачем держать в памяти процесс, который не имеет возможности исполняться)
- По событию «suspend» процесс выгружается на диск
- По событию «activate» загружается в основную память
- Повышенная нагрузка на дисковую подсистему!

Runnable/Suspended state

- Процесс готов к выполнению, но он выгружен из памяти
- Почему?
  - Был неготов к выполнению и выгружен но произошло событие, которое позволяет выполнится
  - «Desperate memory conditions»
  - Команда пользователя
  - Создание процесса в «минимальном» варианте, без, например, создания сегментов памяти

Причины приостановки процессов

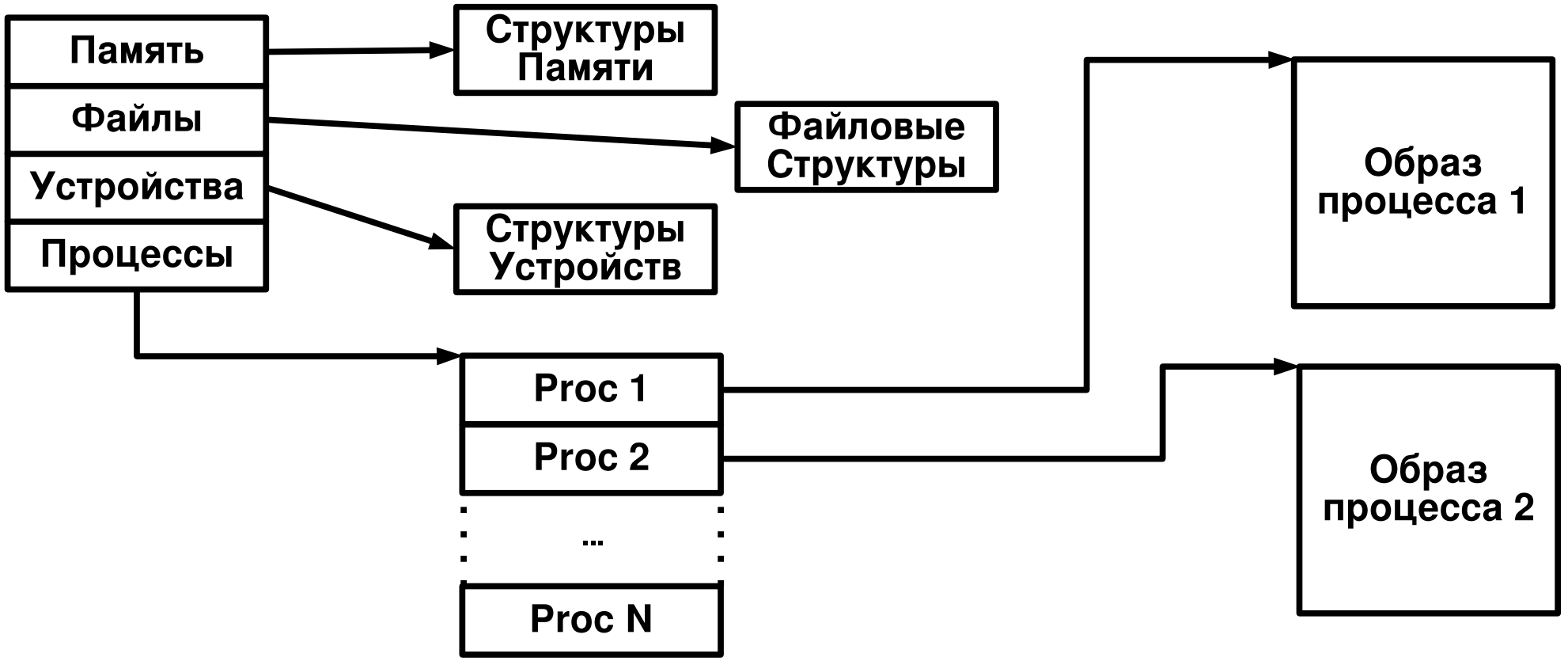
- Swapping
  - ОС нужно освободить память, чтобы загрузить готовый к исполнению процесс
- Другие причины ОС
  - Приостановка фонового, служебного или «подозрительного» процесса
- Интерактивный запрос пользователя
- Запрос родительского процесса
- Задание режима времени исполнения
  - Периодических характер исполнения процесса

Управление процессами

2.3

- Управляющие таблицы
- Образ процесса
- Функции ОС
- Процессы SVR4

Обобщенная структура управляющих таблиц операционной системы



Образ процесса

```
#include <stdio.h>
int var_int_data[1024];
static char var_char_data[4096];

void foo(int var_inc) {
    int var_a = 10;
    static int var_sa = 10;
    var_a += var_inc;
    var_sa += var_inc;
    printf("a = %d, sa = %d\n", var_a, var_sa);
}
int main(int argc, char**argv) {
    int var_i;
    for (var_i = 0; var_i < 10; ++var_i)
        foo(var_i);
}
```

```
serge@ra:/tmp$ pmap -x 275363
275363:   /tmp/a
Address           Kbytes     RSS   Dirty Mode  Mapping
0000555555554000       4       4       4 r---- a_prog
0000555555555000       4       4       4 r-x-- a_prog
0000555555556000       4       0       0 r---- a_prog
0000555555557000       4       4       4 r---- a_prog
0000555555558000       4       4       4 rw--- a_prog
0000555555559000       8       0       0 rw---   [ anon ]
00007ffff7dc0000     148     148       0 r---- libc-2.31.so
00007ffff7de5000    1504     484       8 r-x-- libc-2.31.so
00007ffff7f5d000     296      64       0 r---- libc-2.31.so
00007ffff7fa7000       4       0       0 ----- libc-2.31.so
00007ffff7fa8000      12      12      12 r---- libc-2.31.so
00007ffff7fab000      12      12      12 rw--- libc-2.31.so
00007ffff7fae000      24      20      20 rw---   [ anon ]
00007ffff7fcb000      12       0       0 r----   [ anon ]
00007ffff7fce000       4       4       4 r-x--   [ anon ]
00007ffff7fcf000       4       4       0 r---- ld-2.31.so
00007ffff7fd0000     140     140      24 r-x-- ld-2.31.so
00007ffff7ff3000      32      32       0 r---- ld-2.31.so
00007ffff7ffc000       4       4       4 r---- ld-2.31.so
00007ffff7ffd000       4       4       4 rw--- ld-2.31.so
00007ffff7ffe000       4       4       4 rw---   [ anon ]
00007ffffffde000     132      12      12 rw---   [ stack ]
ffffffffff600000       4       0       0 --x--   [ anon ]
---------------- ------- ------- -------
total kB            2368     960     120
```

Образ процесса

```
#include <stdio.h>
int var_int_data[1024];
static char var_char_data[4096];

void foo(int var_inc) {
    int var_a = 10;
    static int var_sa = 10;
    var_a += var_inc;
    var_sa += var_inc;
    printf("a = %d, sa = %d\n", var_a, var_sa);
}
int main(int argc, char**argv) {
    int var_i;
    for (var_i = 0; var_i < 10; ++var_i)
        foo(var_i);
}
```

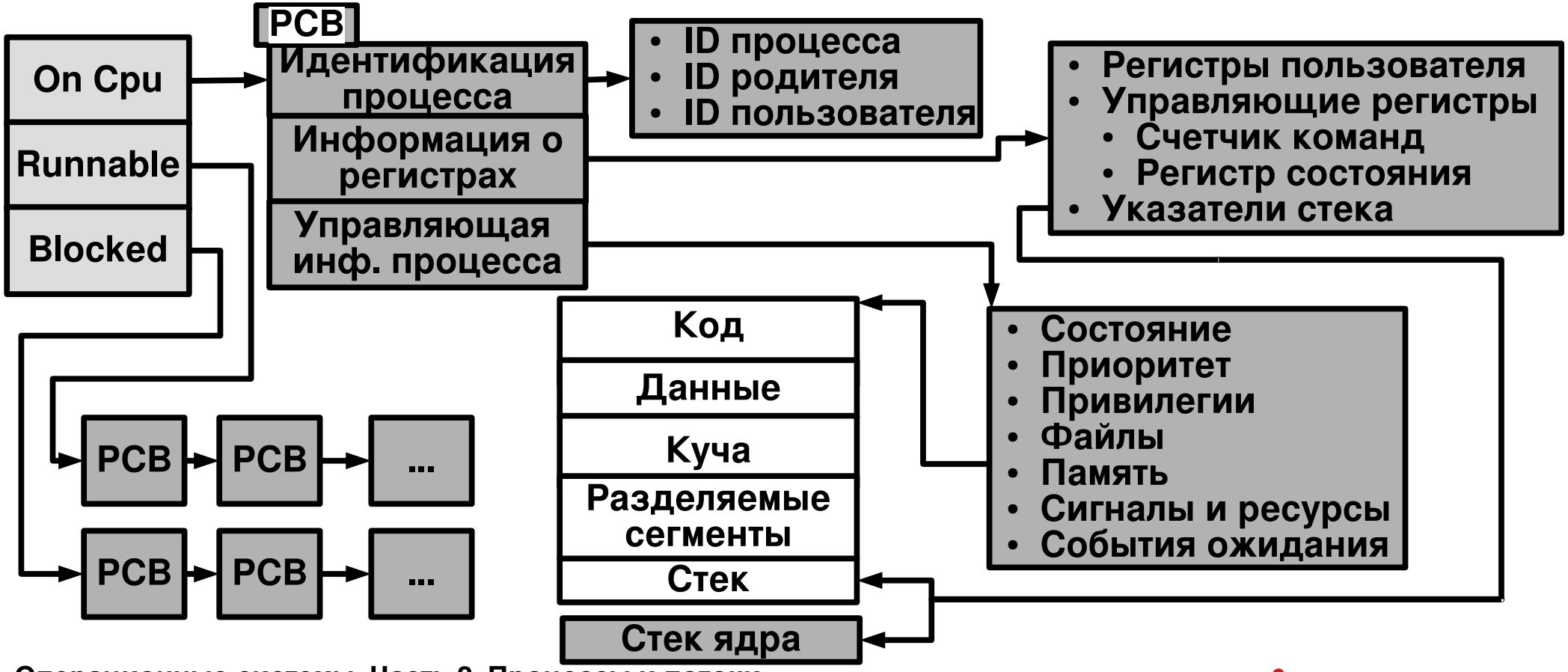
```
serge@ra:/tmp$ pmap -x 275363
275363:   /tmp/a
Address           Kbytes     RSS   Dirty Mode  Mapping
0000555555554000       4       4       4 r---- a_prog
0000555555555000       4       4       4 r-x-- a_prog
0000555555556000       4       0       0 r---- a_prog
0000555555557000       4       4       4 r---- a_prog
0000555555558000       4       4       4 rw--- a_prog
0000555555559000       8       0       0 rw---   [ anon ]
00007ffff7dc0000     148     148       0 r---- libc-2.31.so
00007ffff7de5000    1504     484       8 r-x-- libc-2.31.so
00007ffff7f5d000     296      64       0 r---- libc-2.31.so
00007ffff7fa7000       4       0       0 ----- libc-2.31.so
00007ffff7fa8000      12      12      12 r---- libc-2.31.so
00007ffff7fab000      12      12      12 rw--- libc-2.31.so
00007ffff7fae000      24      20      20 rw---   [ anon ]
00007ffff7fcb000      12       0       0 r----   [ anon ]
00007ffff7fce000       4       4       4 r-x--   [ anon ]
00007ffff7fcf000       4       4       0 r---- ld-2.31.so
00007ffff7fd0000     140     140      24 r-x-- ld-2.31.so
00007ffff7ff3000      32      32       0 r---- ld-2.31.so
00007ffff7ffc000       4       4       4 r---- ld-2.31.so
00007ffff7ffd000       4       4       4 rw--- ld-2.31.so
00007ffff7ffe000       4       4       4 rw---   [ anon ]
00007ffffffde000     132      12      12 rw---   [ stack ]
ffffffffff600000       4       0       0 --x--   [ anon ]
---------------- ------- ------- -------
total kB            2368     960     120
```

Образ процесса

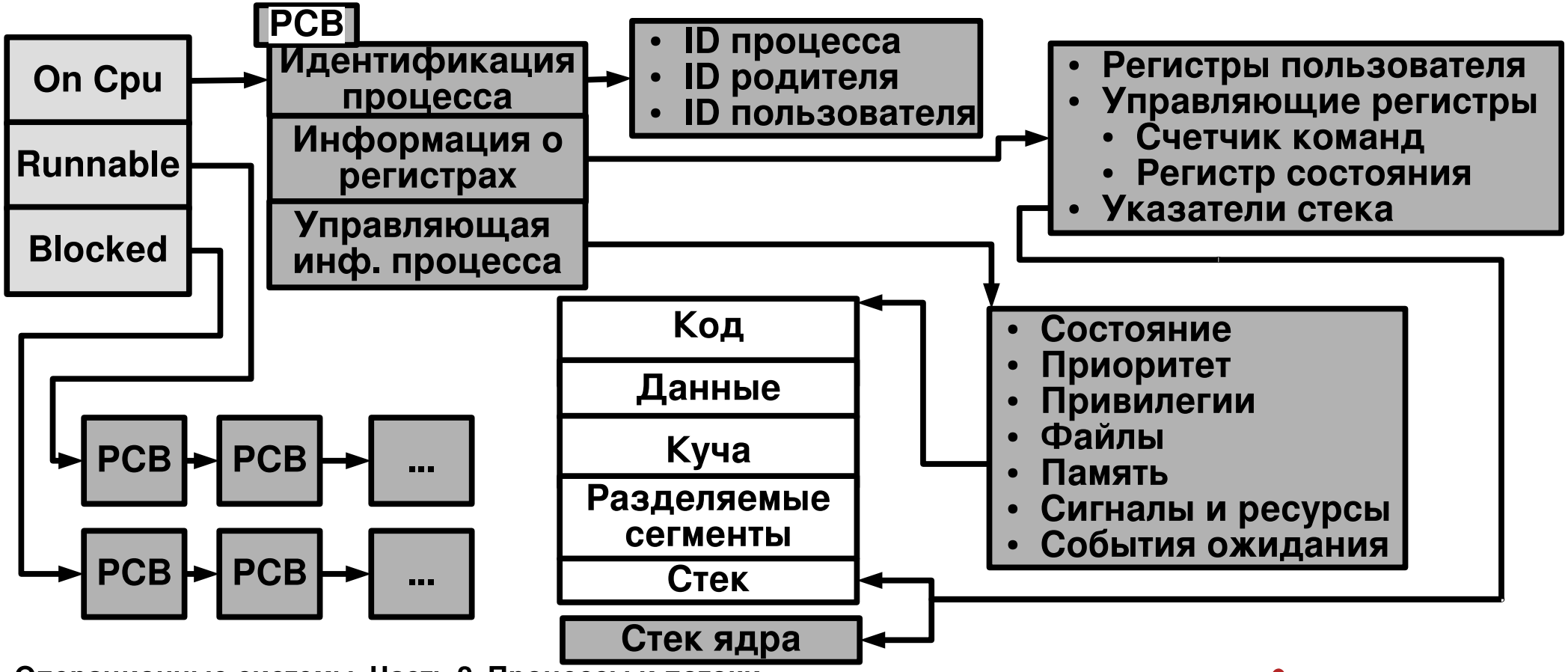
```
serge@ra:/tmp$ gcc -o a_prog -g a.c
serge@ra:/tmp$ gdb a_prog
GNU gdb (Ubuntu 9.1-0ubuntu1) 9.1
[...]
(gdb) break foo
Breakpoint 1 at 0x1149: file a.c, line 7
(gdb) run
Starting program: /tmp/a_prog
Breakpoint 1, foo (var_inc=32767) at a.c:7
7 {
(gdb) step 2
10      var_a += var_inc;
(gdb) print &var_char_data
[...] &var_int_data, &var_a, &var_sa, &var_inc
$4 = (char (*)[4096]) 0x555555558040 <var_char_data>
$3 = (int (*)[1024]) 0x555555559040 <var_int_data>
$5 = (int *) 0x7fffffffde5c #var_a
$6 = (int *) 0x555555558010 <var_sa>
$7 = (int *) 0x7fffffffde4c #var_inc
```

```
serge@ra:/tmp$ pmap -x 275363
275363:   /tmp/a
Address           Kbytes     RSS   Dirty Mode  Mapping
...
0000555555555000       4       4       4 r-x-- a_prog
...
0000555555558000       4       4       4 rw--- a_prog
0000555555559000       8       0       0 rw---   [ anon ]
00007ffff7dc0000     148     148       0 r---- libc-2.31.so
00007ffff7de5000    1504     484       8 r-x-- libc-2.31.so
...
00007ffff7fab000      12      12      12 rw--- libc-2.31.so
00007ffff7fae000      24      20      20 rw---   [ anon ]
00007ffff7fcb000      12       0       0 r----   [ anon ]
00007ffff7fce000       4       4       4 r-x--   [ anon ]
00007ffff7fcf000       4       4       0 r---- ld-2.31.so
00007ffff7fd0000     140     140      24 r-x-- ld-2.31.so
...
00007ffff7ffd000       4       4       4 rw--- ld-2.31.so
00007ffff7ffe000       4       4       4 rw---   [ anon ]
00007ffffffde000     132      12      12 rw---   [ stack ]
ffffffffff600000       4       0       0 --x--   [ anon ]
---------------- ------- ------- -------
total kB            2368     960     120
```

Управляющий блок процесса (PCB)



Управляющий блок процесса (PCB)



Функции ОС связанные с процессами

- Управление процессами
  - Создание и завершение процессов
  - Планирование и диспетчеризация процессов
  - Переключение процессов
  - Синхронизация и поддержка обмена информацией между процессами
  - Организация управляющих блоков процессов
- Управление памятью
  - Выделение адресного пространства процессам
  - Пейджинг и Свопинг
  - Управление страницами и сегментами
- Управление вводом-выводом
  - Управление буферами
  - Выделение процессам каналов и устройств ввода-вывода
- Функции поддержки
  - Обработка прерываний
  - Учет использования ресурсов
  - Текущий контроль системы

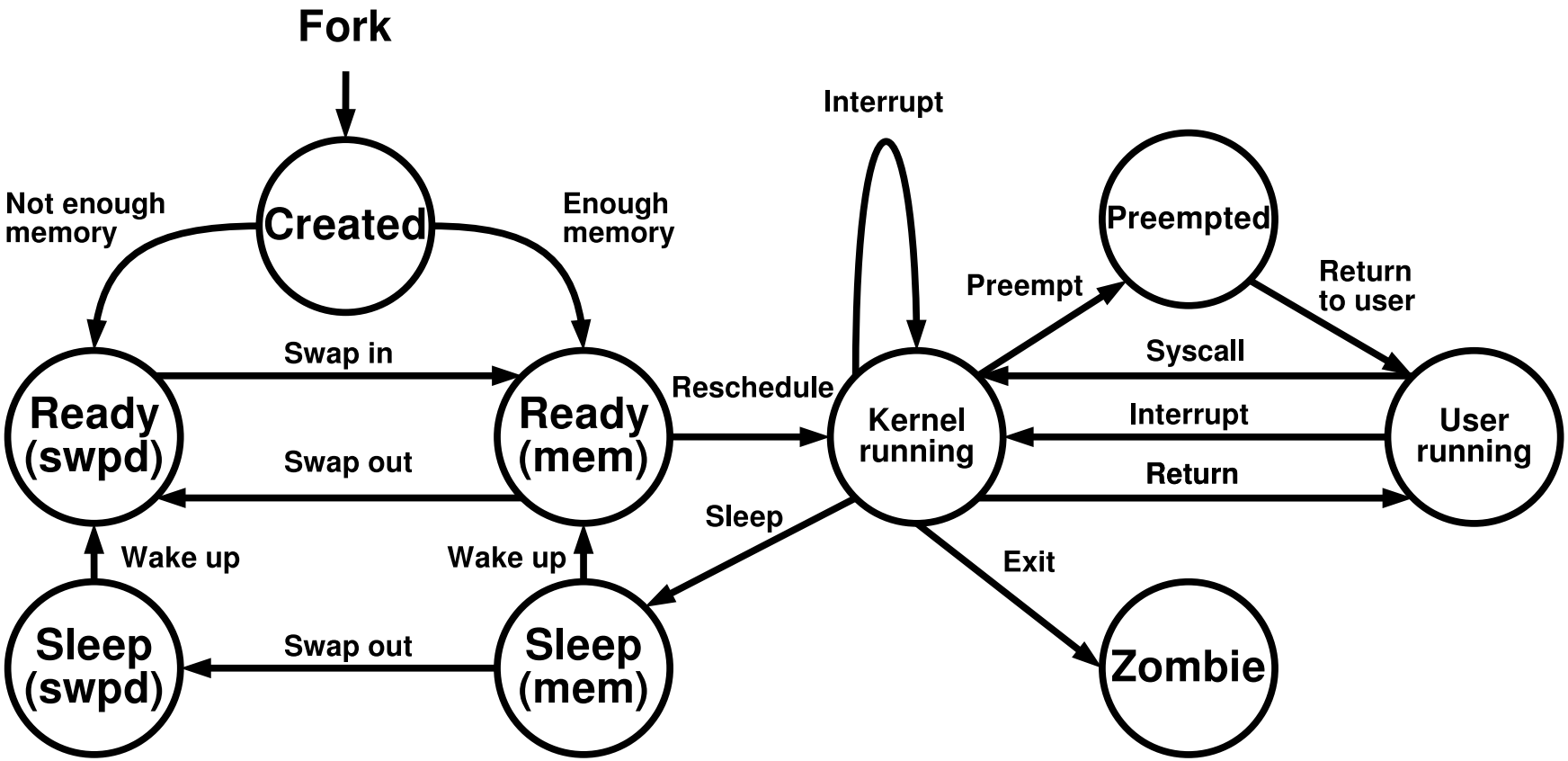
Создание процесса

- Присвоить процессу уникальный идентификатор
- Выделить память для процесса
- Инициализировать PCB
- Поставить процесс в очереди ядра
- Создать потоки ввода-вывода
- Создать другие управляющие структуры данных

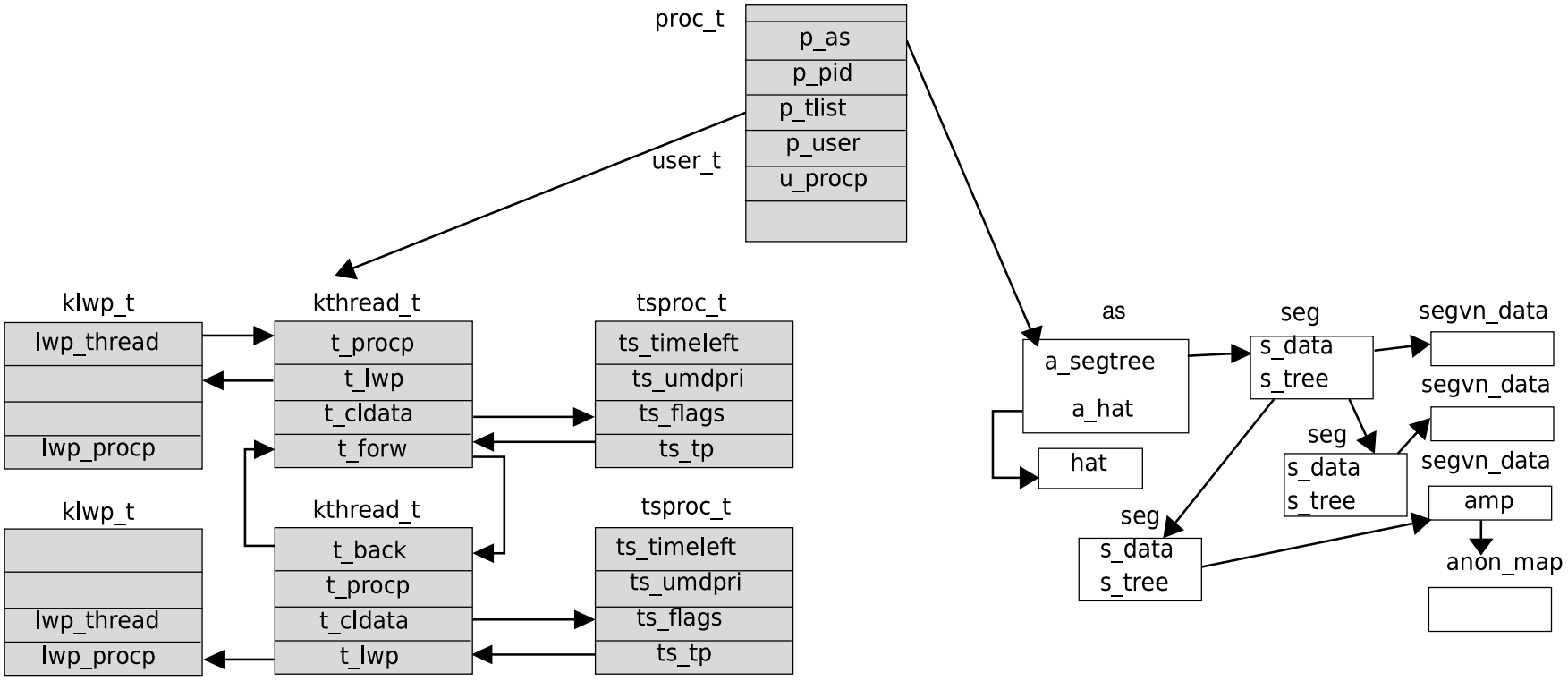
Переключение процессов

- Процесс может работать в user и kernel mode
- Используется механизм прерываний:
  - Внешнее прерывание (IO)
  - Ловушка — обработка ошибки или исключительной ситуации
  - Вызов ОС
- Переход из user в kernel-mode ситуации:
  - Прерывания таймера
  - Прерывания ввода-вывода
  - Page fault — отстутствие блока памяти

Состояние процессов Unix SVR4 (Solaris)



Структуры процессов Unix SVR4 (Solaris)



Потоки

2.4

- Понятие потока
- Связь потоков и процессов
- Преимущества потоков
- Состояние потока
- Варианты реализации
- Закон Амдала

Thread

- Абстрактная модель процесса подразумевает:
  - Владение ресурсами (сегменты памяти, файлы, каналы ввода вывода, контекст безопасности, …)
  - Планирование и диспетчеризацию (приоритеты, состояния)

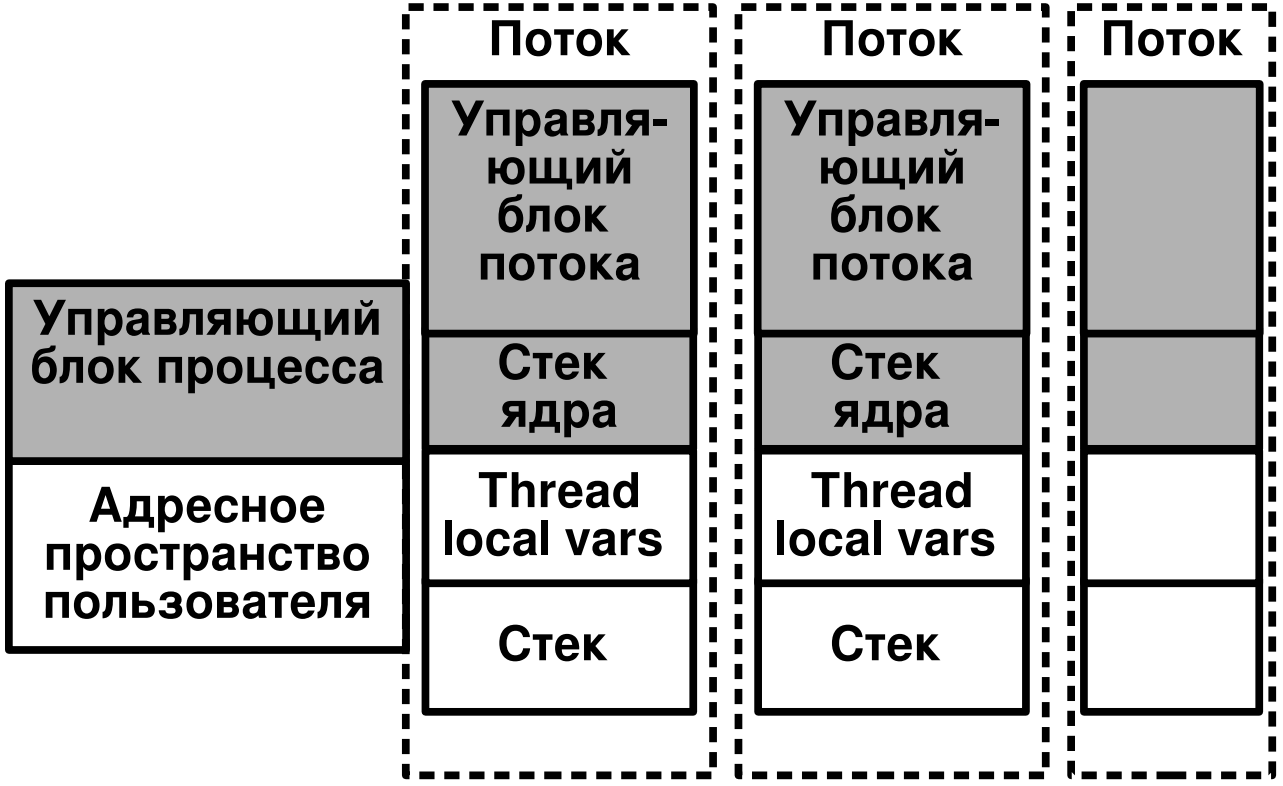
независимы

- Процесс — единица группировки общих ресурсов
- Thread (нить выполнения) — единица выполнения программного кода. Содержит:
  - Состояние выполнения
  - Сохраненный контекст потока (регистры, …)
  - Стек (ядра и пользователя)
  - Локальные переменные (thread_locals)
  - Доступ к памяти и другим ресурсам процесса-владельца

Связь структур ядра процесса и потока

Однопоточная модель

Многопоточная модель



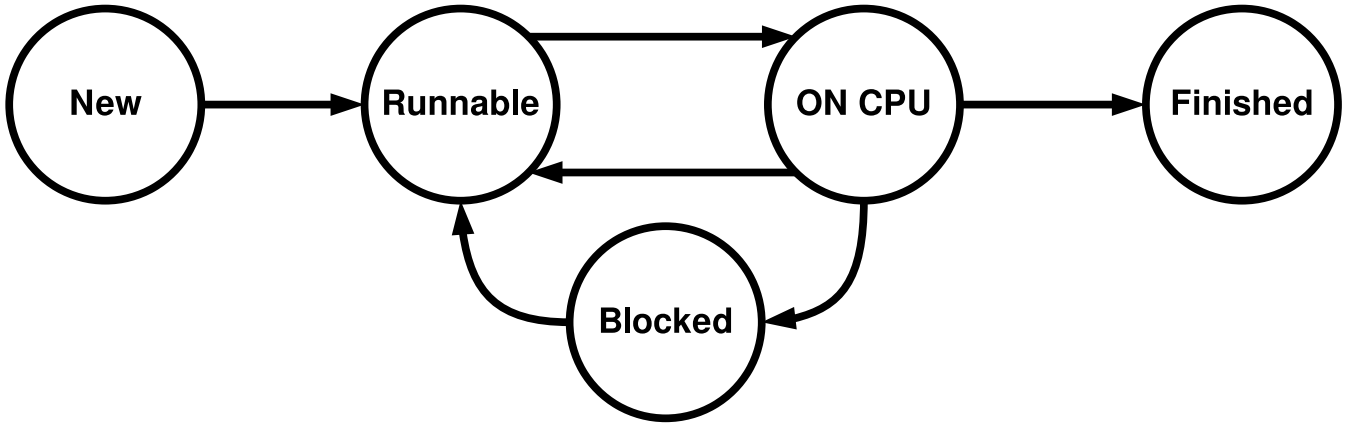
Преимущество потоков

- Общее преимущества в быстродействии:
  - Потоки создаются на порядок быстрее
  - Потоки переключаются быстрее
  - Потоки завершаются быстрее
  - Потоки могут обмениваться информацией быстрее (без участия ядра)
- Даже в однопроцессорной системе есть преимущества
  - Работа в приоритетном и фоновом режиме
  - Асинхронная обработка частей программы
  - Модульная структура программы

Преимущество потоков

- Общее преимущества в быстродействии:
  - Потоки создаются на порядок быстрее
  - Потоки переключаются быстрее
  - Потоки завершаются быстрее
  - Потоки могут обмениваться информацией быстрее (без участия ядра)
- Даже в однопроцессорной системе есть преимущества
  - Работа в приоритетном и фоновом режиме
  - Асинхронная обработка частей программы
  - Модульная структура программы

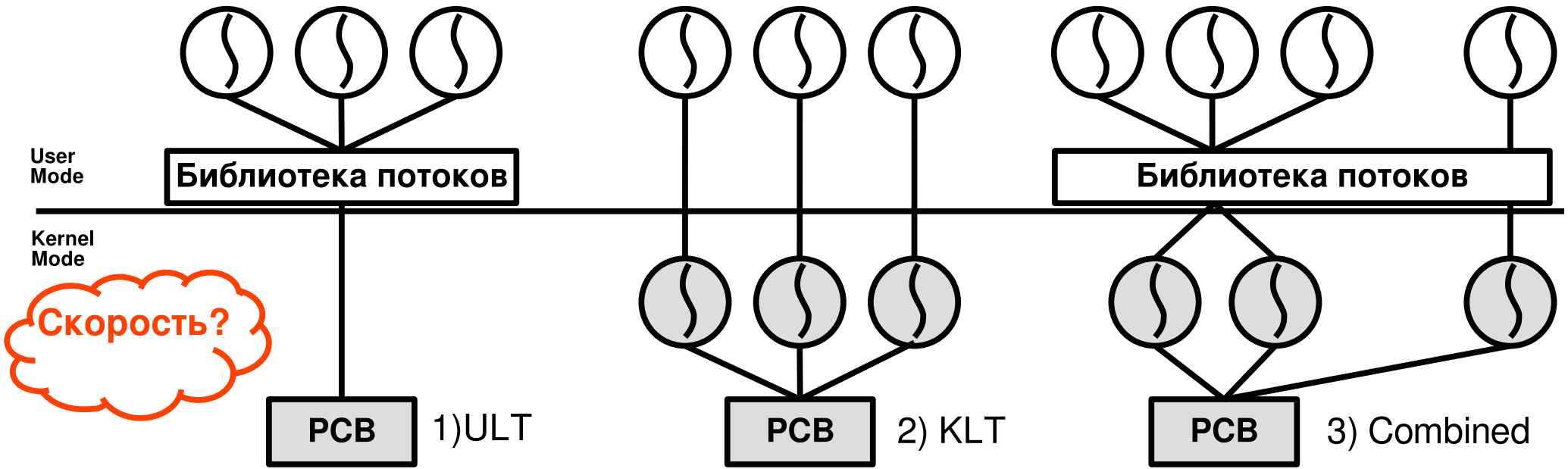
Состояние потоков



- NB! потоки используют адресное пространство процесса $\rightarrow$ необходима синхронизация по общим данным!
- Блокирование потока не должно приводить к блокированию процесса

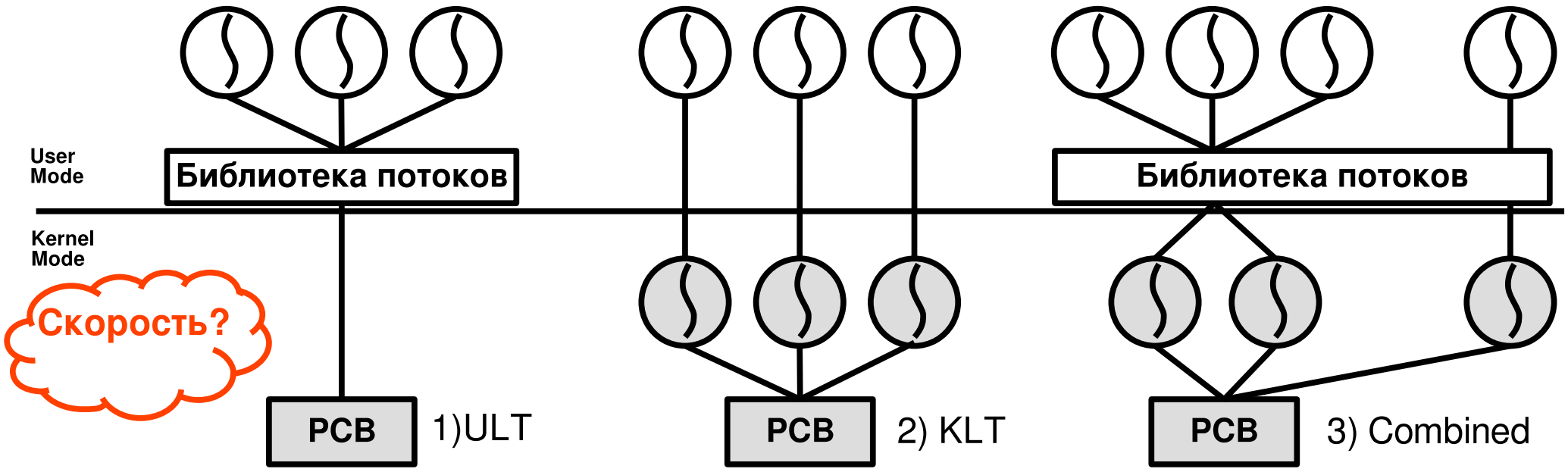
User Level Threads vs Kernel Level Threads

- ULT (Green Threads) — реализуются библиотеками (или приложениями) на стороне пользователя
- KLT (иногда LWP — light-weight processes) — реализуются ядром



User Level Threads vs Kernel Level Threads

- ULT (Green Threads) — реализуются библиотеками (или приложениями) на стороне пользователя
- KLT (иногда LWP — light-weight processes) — реализуются ядром



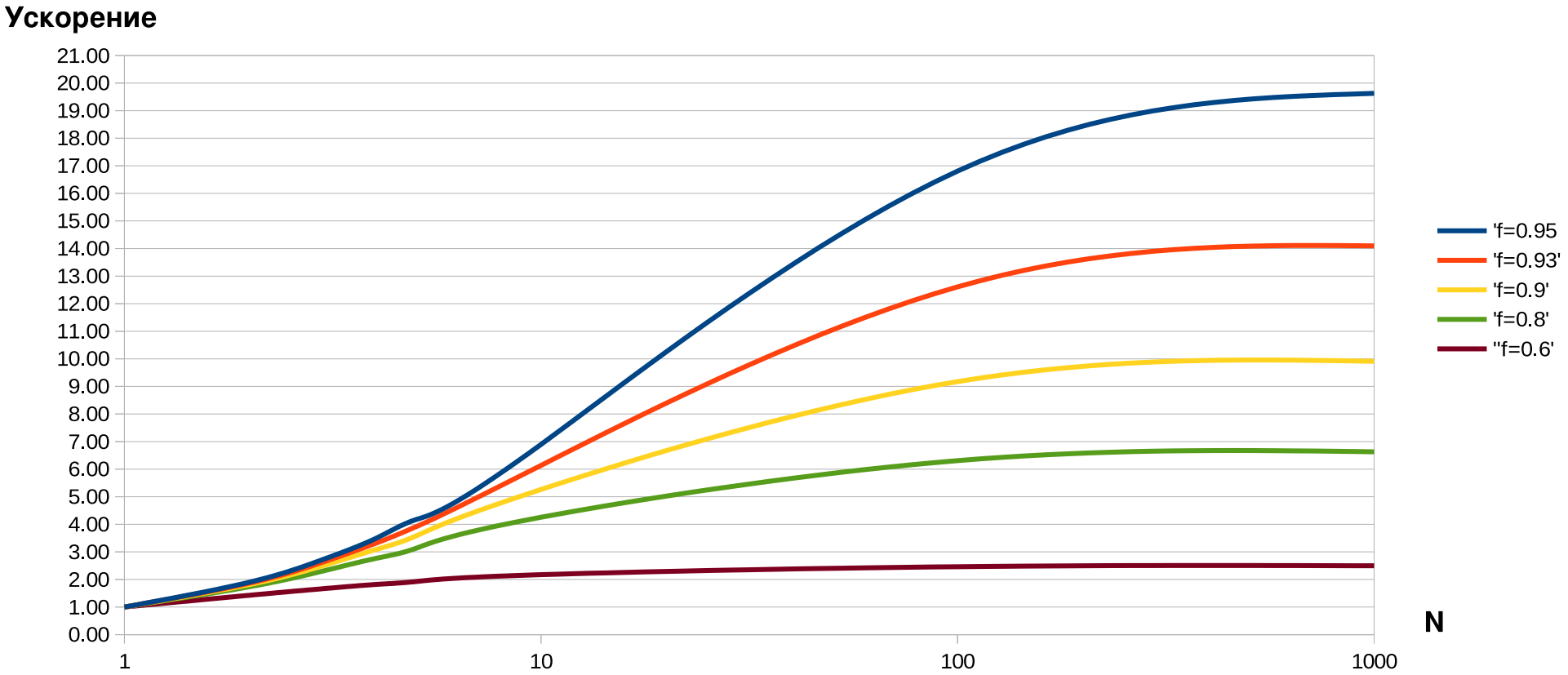
Многопроцессорность и многопоточность

- Насколько можно ускорить программу на N процессорах с использованием потоков?
- Закон Амдала:

$$Ускорение = \frac{Время\ работы\ на\ одном\ процессоре}{Время\ выполнения\ на\ N\ процессорах} = \frac{T \times (1-f) + T \times f}{T \times (1-f) + \frac{T \times f}{N}} = \frac{1}{(1-f) + \frac{f}{N}}$$

  где, T- время работы; f — доля распараллеливания [0..1))
- Когда f мало, использование параллельного выполнения неэффективно
- Когда N $\rightarrow \infty$, то ускорение ограниченно 1/(1-f)

Закон Амдала



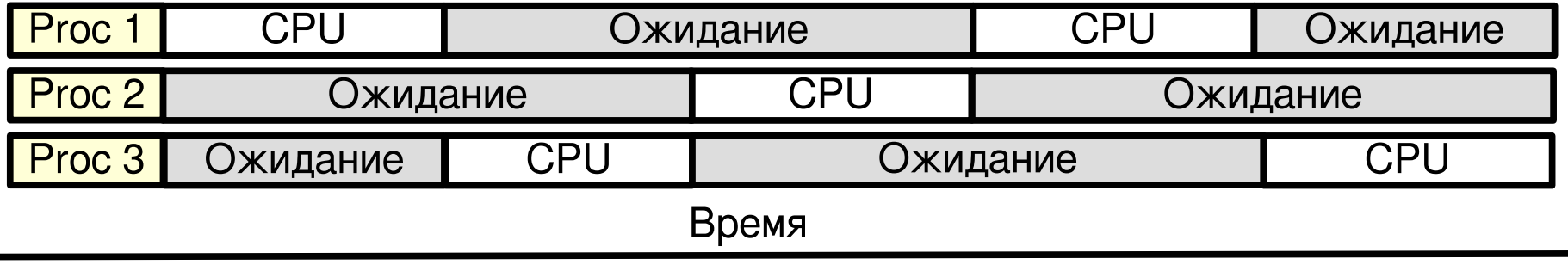
Параллельные вычисления. Блокировки.

2.5

- Параллельность программ
- Функции ОС поддержки параллельности
- Проблемы
- Аппаратная поддержка
- Взаимодействие процессов

Механизм «параллельных» вычислений

- Однопроцессорные системы — процессы чередуются



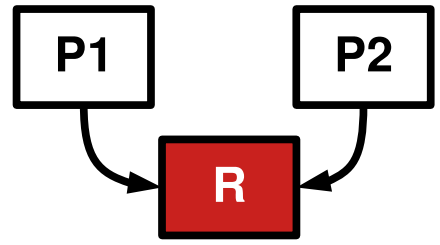
- Многопроцессорные — чередуются и перекрываются.
  - Конкуренция за общие ресурсы. Голодание.

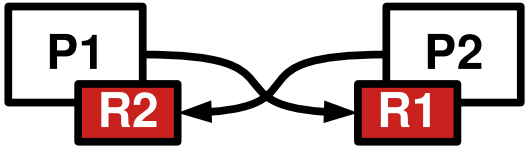
Требуемые функции ОС

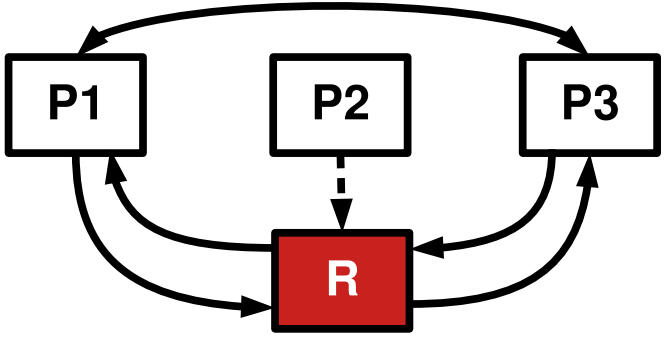
- Отслеживание ресурсов процесса/потока
- Распределение и освобождение ресурсов для каждого активного процесса/потока (CPU, файлы, память, ...)
- Защита ресурсов процесса/потока от непреднамеренного воздействия других процессов/потоков
- Независимость результата процесса/потока от скорости его выполнения и скорости других процессов/потоков

Проблемы параллельного выполнения

- Взаимоисключения (Mutual Exclusion) — процессы/потоки не должны одновременно использовать критический ресурс
- Взаимоблокировки (DeadLocks, LiveLocks) — процессы/потоки не должны взаимозахватывать требуемые ресурсы.
- Голодание (Starvation) — конкуренция за ресурсы не должна порождать невозможность доступа к ресурсу







Требования к взаимным исключениям

- Взаимоисключения осуществляются в принудительном порядке.
  - В критическом участке кода должен находится один процесс/поток
- Процесс/поток не должен влиять на другие процессы/потоки в некритическом участке
- Противодействие бесконечному ожиданию доступа к критическому участку
- Вход в свободный критический участок должен незамедлительно предоставляться
- Отсутствие предположений о количестве процессов или их скорости
- Ограничение времени нахождения в критических участках

Аппаратная поддержка взаимных исключений

- Запрет прерываний на однопроцессорных системах
  - DI; Критический участок; EI
- Атомарные инструкции TAS, CAS, CMPXCHG, ...

```
spin_acqure_lock: // locked = 1, released = 0
      movl    $1, %ebx
      movl    lock_var_addr, %ecx
loop: pause   //rep nop
      movl    (%ecx), %eax //avoid unness. lock
      test    %eax, %eax
      jnz     loop
      lock cmpxchg %ebx, (%ecx) // %eax = 0
      jnz     loop
      ret

spin_unlock:
      mfence
      movl$0, (lock_var_addr)
      ret
```

```
/* Atomic compare-and-exchange: */
Compare eax with memory (%ecx)
if equal
  load %ebx in memory (%ecx), ZF=1
else
  load memory in %eax, ZF=0
```

```
pause – hint CPU we are spining
mfence – load/store bariers
```

https://www.cs.virginia.edu/~cr4bd/4414/F2018/slides/20180925--slides-1up.pdf

Аппаратная поддержка взаимных исключений

- Запрет прерываний на однопроцессорных системах
  - DI; Критический участок; EI
- Атомарные инструкции TAS, CAS, CMPXCHG, ...

```
spin_acqure_lock: // locked = 1, released = 0
      movl    $1, %ebx
      movl    lock_var_addr, %ecx
loop: pause   //rep nop
      movl    (%ecx), %eax //avoid unness. lock
      test    %eax, %eax
      jnz     loop
      lock cmpxchg %ebx, (%ecx) // %eax = 0
      jnz     loop
      ret

spin_unlock:
      mfence
      movl$0, (lock_var_addr)
      ret
```

```
/* Atomic compare-and-exchange: */
Compare eax with memory (%ecx)
if equal
  load %ebx in memory (%ecx), ZF=1
else
  load memory in %eax, ZF=0
```

```
pause – hint CPU we are spining
mfence – load/store bariers
```

https://www.cs.virginia.edu/~cr4bd/4414/F2018/slides/20180925--slides-1up.pdf

Аппаратная поддержка взаимных исключений

- Запрет прерываний на однопроцессорных системах
  - DI; Критический участок; EI
- Атомарные инструкции TAS, CAS, CMPXCHG, ...

```
spin_acqure_lock: // locked = 1, released = 0
      movl    $1, %ebx
      movl    lock_var_addr, %ecx
loop: pause   //rep nop
      movl    (%ecx), %eax //avoid unness. lock
      test    %eax, %eax
      jnz     loop
      lock cmpxchg %ebx, (%ecx) // %eax = 0
      jnz     loop
      ret

spin_unlock:
      mfence
      movl$0, (lock_var_addr)
      ret
```

```
/* Atomic compare-and-exchange: */
Compare eax with memory (%ecx)
if equal
  load %ebx in memory (%ecx), ZF=1
else
  load memory in %eax, ZF=0
```

```
pause – hint CPU we are spining
mfence – load/store bariers
```

https://www.cs.virginia.edu/~cr4bd/4414/F2018/slides/20180925--slides-1up.pdf

Взаимодействие процессов/потоков

- Процессы/потоки ничего не знают друг о друге
  - Конкуренция за ресурсы.
  - Взаимоисключения, взаимоблокировки, голодание
- Процессы/потоки сотрудничают используя общий ресурс
  - Взаимоисключения, взаимоблокировки, голодание, связь данных
- Процессы/потоки совместно выполняются
  - Взаимоблокировки, голодание

Примитивы синхронизации ОС

2.6

- Семафоры, Мьютексы
- Условные переменные
- Блокировки чтения/записи
- Мониторы
- Флаги событий, Почтовые ящики
- Неблокирующие алгоритмы

Основные примитивы

- Семафоры (Semaphore) — захват и освобождение множественного ресурса
  - или одного (бинарные семафоры)
- Мьютексы (Mutex) — блокировка и освобождение ресурса единственным процессом/потоком
- Условные переменные (Conditional Variable) — Блокировка до выполнения какого-либо условия
- Блокировки чтения/записи (rw-lock) — отдельные блокировки на чтение и запись
- Мониторы (Monitor) — конструкции языков программирования, которые скрывают низкоуровневые примитивы синхронизации
- Флаги событий (Event Flags) — связвание условий продолжения выполнения с одним или несколькими флагами (битами блокирующей переменной)
- Почтовые ящики (Message Passing) — передача сообщений

Столлингс гл. 5.1 — Эволюция подхода к блокировке — самостоятельно прочитать!

Counting Semaphores (Дейкстра)

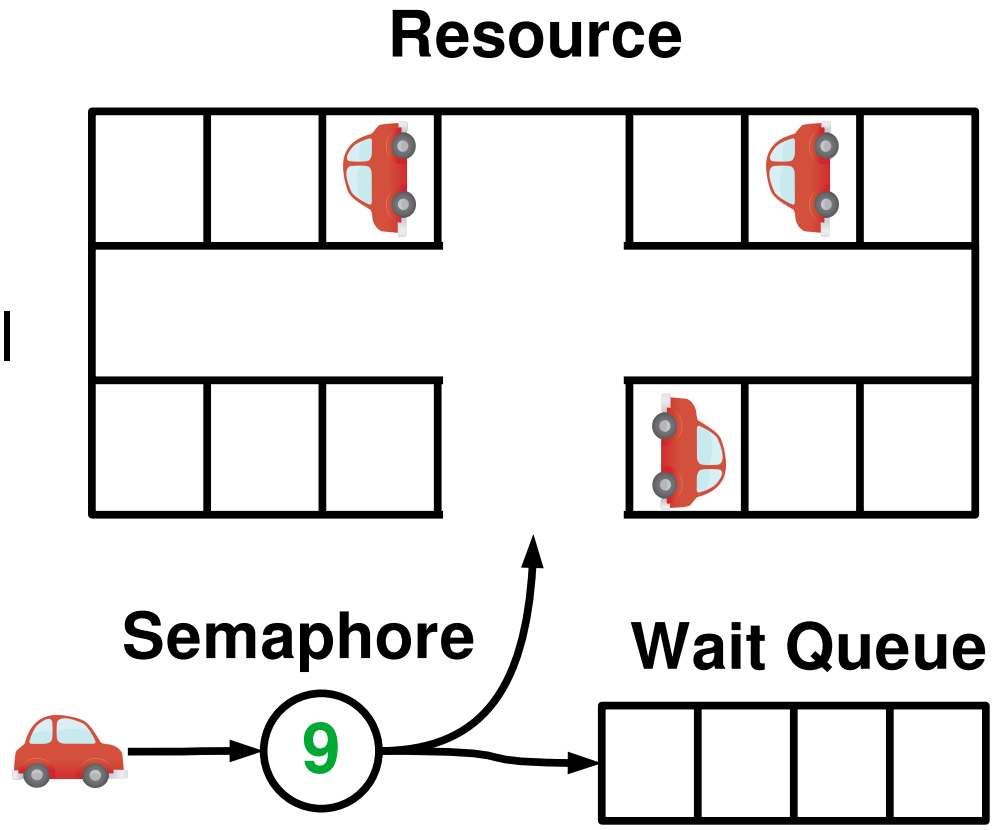
- sema_p() «proberen», semWait

```
count--
if count<0
    thread blocks in WQ
```

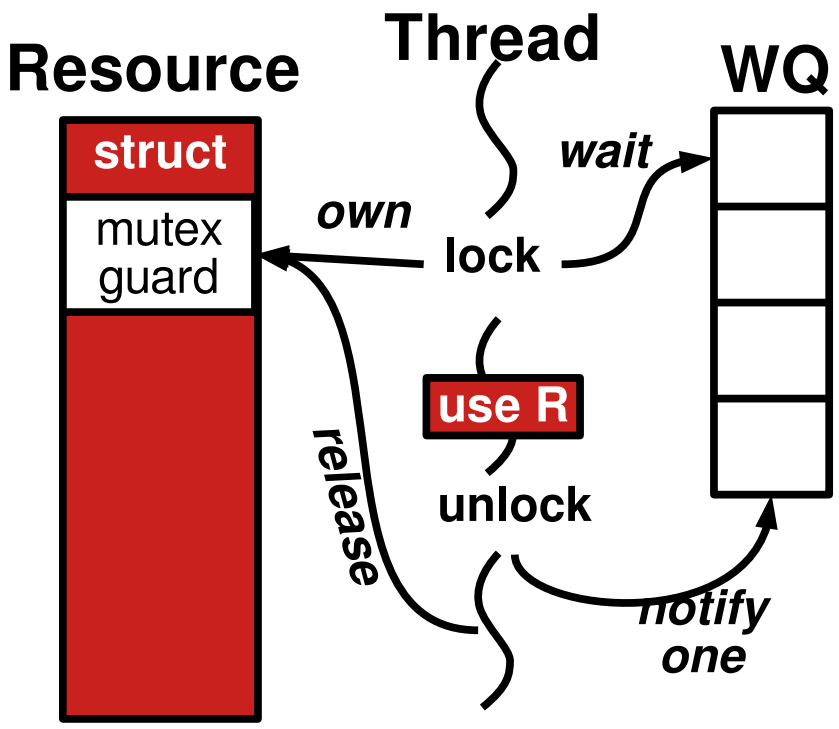
- sema_v() «verhogen», semSignal

```
count++
if count<=0
    notify thread
```

- Политика выборки разная
  - строгая, слабая, приоритет



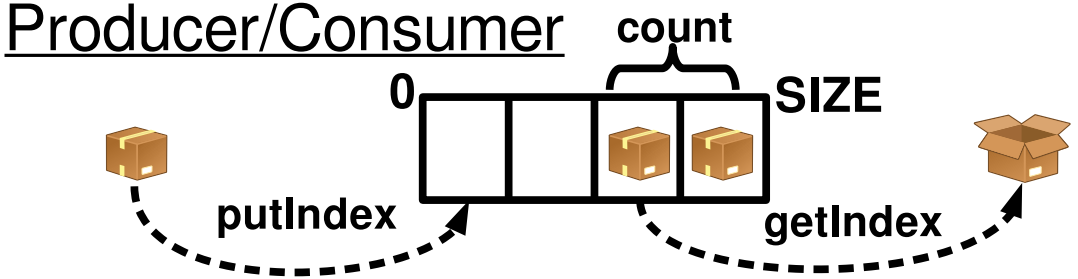
Mutexes



- Множество различных реализаций:
  - Блокирующие, спин, адаптивные, фьютексы, ...
- Не- бинарный семафор!
  - Захвативший блокировку должен ее освободить
- Захват должен быть коротким
- Priority inversion

Conditional Variables

- wait(condition, lock)
- signal(condition, lock)
- broadcast(condition, lock)




```
void produce(char c) {
    lock_acquire(&lock);
    while (count == SIZE) {
        cond_wait(&spaceAvailable, &lock);
    }
    count++;
    buffer[putIndex] = c;
    putIndex++;
    if (putIndex == SIZE) {
        putIndex = 0;
    }
    cond_signal(&dataAvailable, &lock);
    lock_release(&lock);
}
```

```
char consume() {
    char c;
    lock_acquire(&lock);
    while (count == 0) {
        cond_wait(&dataAvailable, &lock);
    }
    count--;
    c = buffer[getIndex];
    getIndex++;
    if (getIndex == SIZE) getIndex = 0;
    cond_signal(&spaceAvailable, &lock);
    lock_release(&lock);
    return c;
}
```

Multiple-reader, single-writer locks (rwlocks)

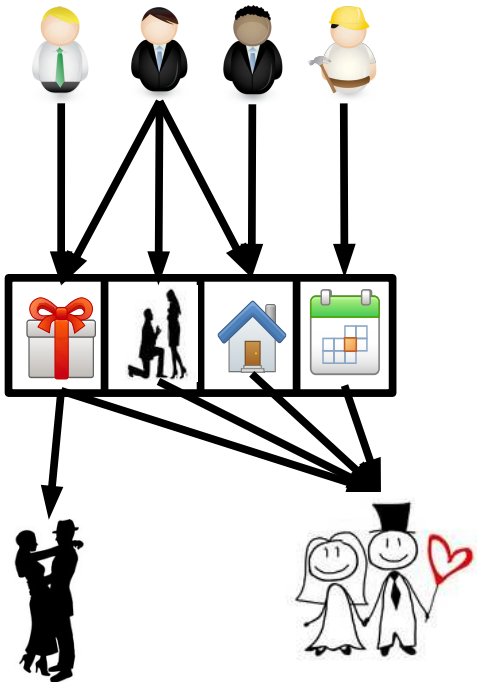
- Когда читатели читают, они захватывает readlock, количество одновременных читателей содержится в rwlock
- Когда писателю требуется записать — он устанавливает требование записи (want write) и ожидает на rwlock
- rwlock ждет освобождения readlock
- Оповещает писателей
  - Один захватывает writelock (читатели не могут читать)
- Оповещает читателей
  - Захватывается readlock (писатели должны требовать)

Monitors

- Набор процедур, выполняющих операции над общими данными
- Каждая процедура подразумевает захват блокировки общих данных
- Локальные переменные используются только внутри монитора
- Для ожидания и оповещения используются Conditional Variables
- Parallel Pascal, Java
- Реализуются высокоуровнево, программисту не требуется возится с захватом/освобождением или ожиданием/нотификацией
- ИМХО В операционных системах не используются

Event Flags

- Наборы (битовых) флагов ожидания событий
- Операции
  - Установка флага
  - Сброс флага
  - Ожидание флага
  - Ожидание какого-либо флага
  - Ожидание всех флагов
- Реализованы VMS, python



Message passing

- Две операции — send(получатель, сообщение) и receive(отправитель, сообщение)
- Необходим метод адресации получателя и отправителя
  - Прямая адресация (direct): явная и постфактум
  - Косвенная адресация (indirect) — номер «почтового ящика» - 1:1, 1:N, M:1, M:N
- Раздельная синхронизация посылки и получения
  - Блокирующая, неблокирующая, гарантия доставки
- Формат сообщения
  - Фиксированная, переменная длинна, файл, ...
- Выборка из очереди — Приоритет, FIFO, ...

Неблокирующие примитивы

- Блокировка вызывает переключение контекста
  - Дорого (долго)!
- Сейчас модно пользоваться неблокирующими примитивами синхронизации и неблокирующими структурам
  - Wait-free (N-steps), lock-free (some N-steps, other retry), obstruction-free
- Атомарные операции и примитивы синхронизации
- Неблокирующие структуры данных — списки, деревья, очереди, ... over 9000.

Столлингс гл. 6.1-6.6 — Разрешение блокировок — самостоятельно прочитать!

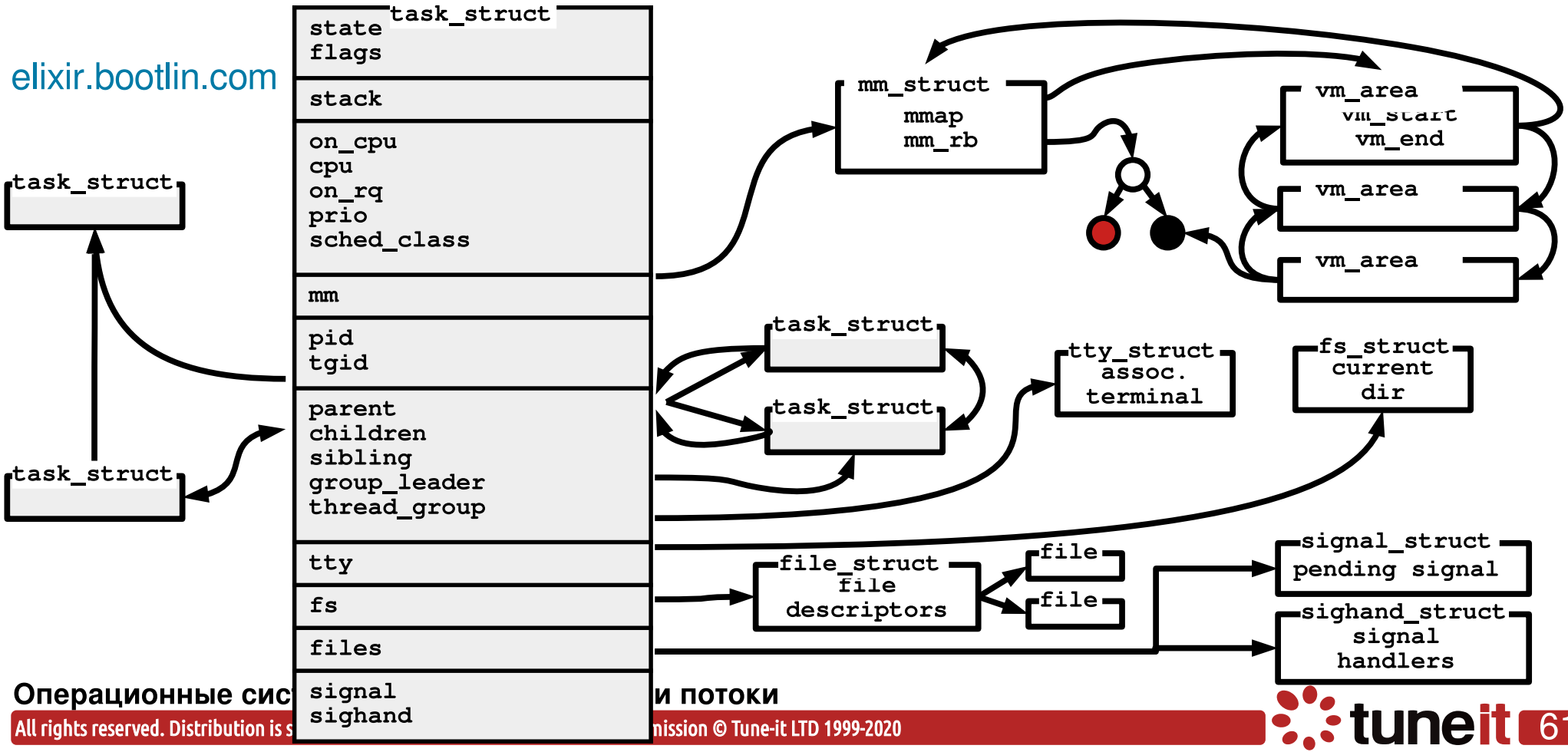
Процессы и потоки Linux

2.7

- Структуры процесса
- «состояния»
- Создание task
- Thread, Kthread, Tasklet
- Уничтожение task

task_struct (include/linux/sched.h)

elixir.bootlin.com



task_struct (include/linux/sched.h)

elixir.bootlin.com

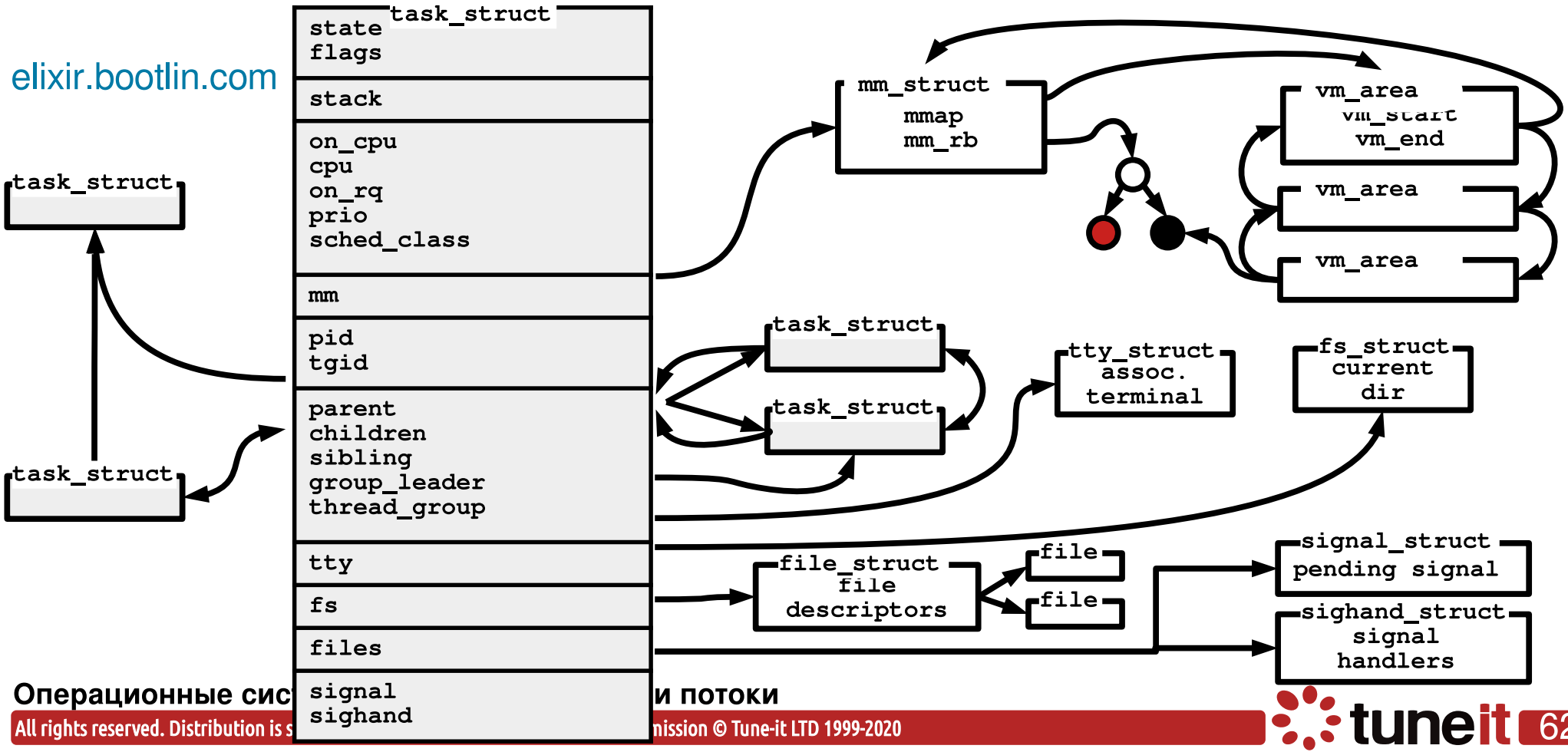
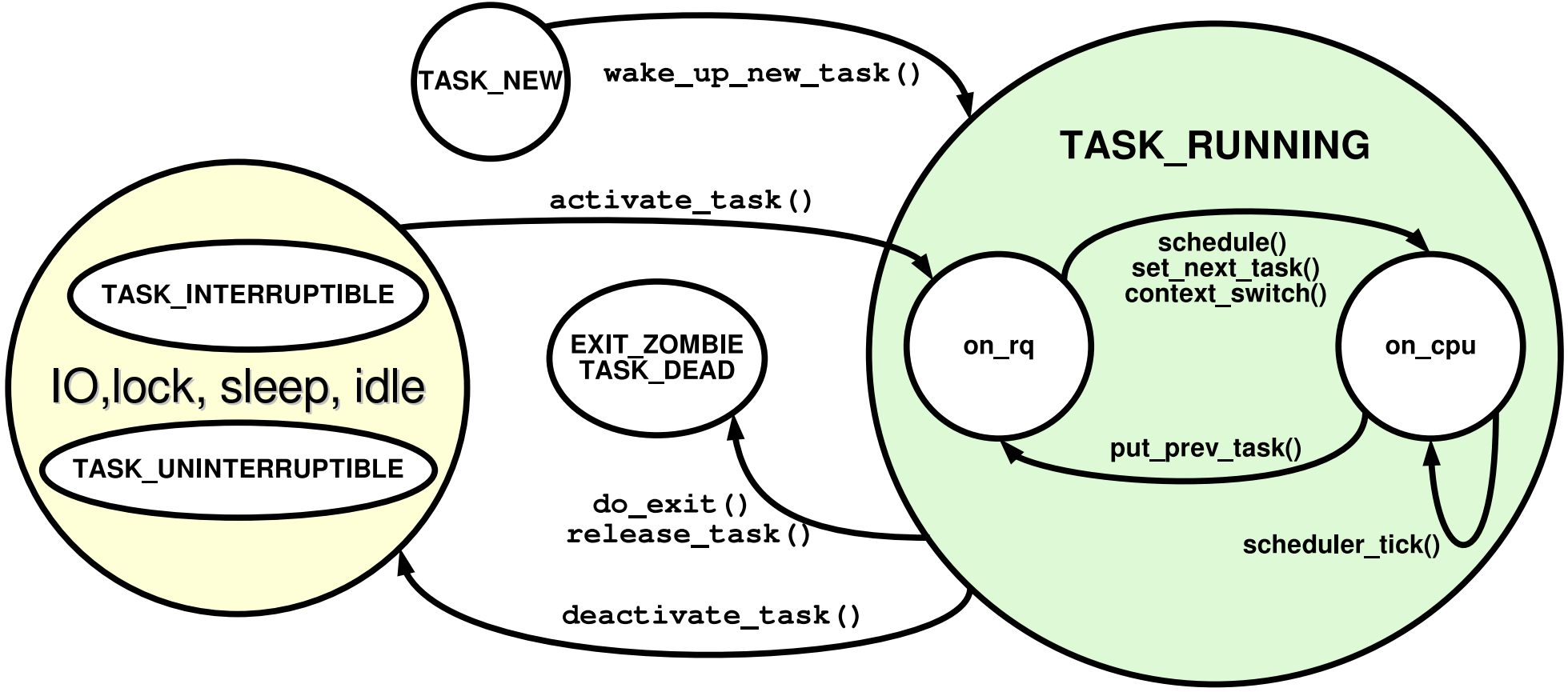


Диаграмма «состояний» процессов



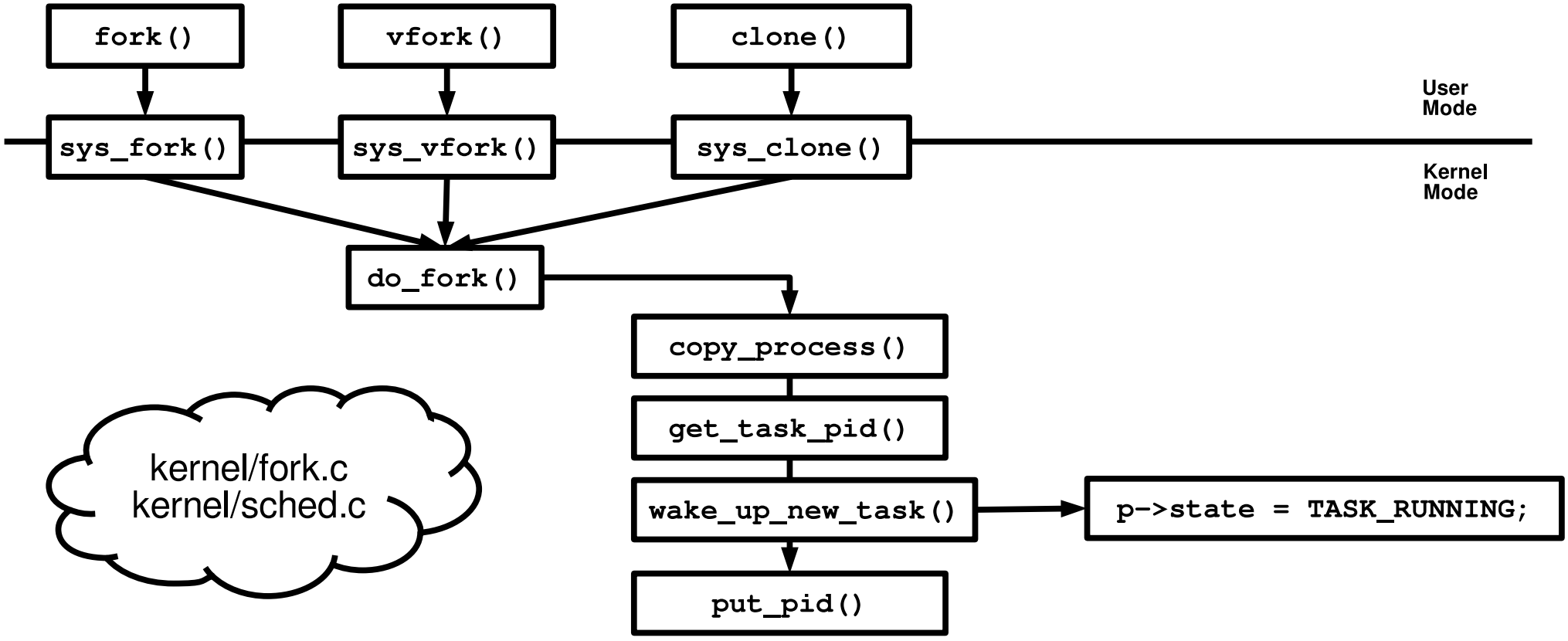
Userland: Создание процесса

- fork() - создать процесс
  - Возвращает номер процесса родителю, 0 — созданному процессу, -1 в случае ошибки.
- vfork() - создать процесс с копией адресного пространства родителя
  - Родительский процесс блокирован до тех пор, пока дочерний не вызовет exit() или execve()
- clone() - создать процесс, управляя копированием выбранных частей процесса
  - clone(CLONE_VM | CLONE_FS | CLONE_FILES | CLONE_SIGHAND, 0);
- execv(const char *path, const char *arg, ...) - перекрытие образа процесса

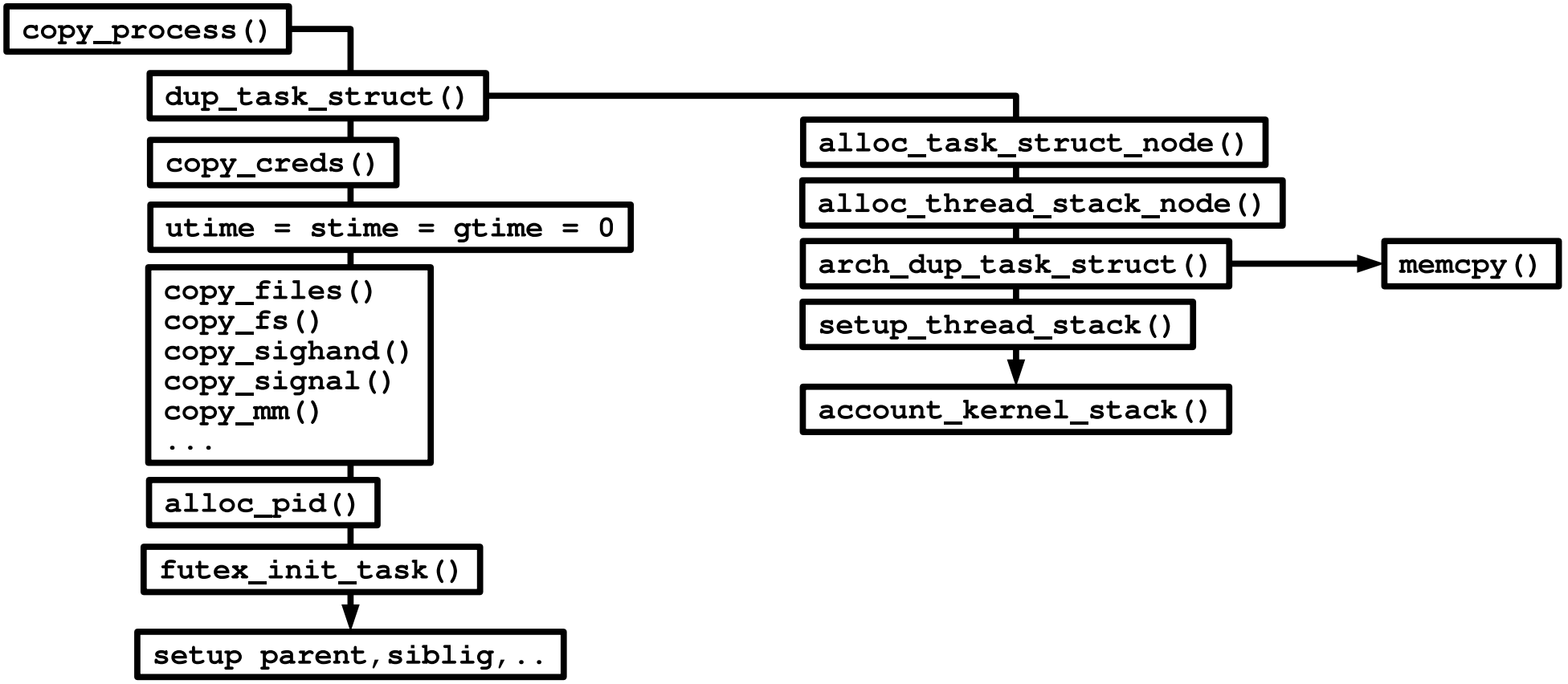
Linux Threads

- В ядре Linux нет примитива Thread, соответствующего пользовательскому потоку
  - Соответственно нет планирования потоков
- Потоки создаются так же, как и процессы
  - Разделяют ресурсы вызвавшего fork процесса
  - Каждый поток — отдельная task_struct
  - Создание: `clone(CLONE_VM | CLONE_FILES | CLONE_FS | CLONE_SIGHAND | CLONE_THREAD | CLONE_SETTLS | CLONE_PARENT_SETTID | CLONE_CHILD_CLEARTID | CLONE_SYSVSEM, 0);` (NPTL version)

Создание процесса



copy_process (kernel/fork.c)



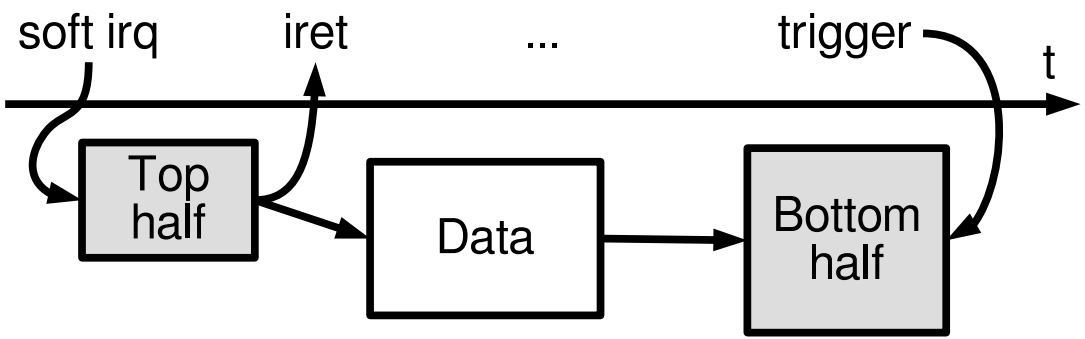
KThread

- Предназначен для выполнения фоновых операций в ядре
  - Планировщик работает с ними так же, как с процессами
- Нет своего адресного пространства
  - mm в task_struct равен NULL
- kthread являются потомками kthreadd (pid==2)
  - Все фоновые потоки ядра ps --ppid=2
- Создание — kthread_create()
  - После создания нужно выполнить wake_up_process()
  - Одновременное создание и запуск макрос kthread_run()
- Для остановки другие потоки вызывают kthread_stop()
  - А сам поток должен проверять kthread_should_stop()

Tasklets

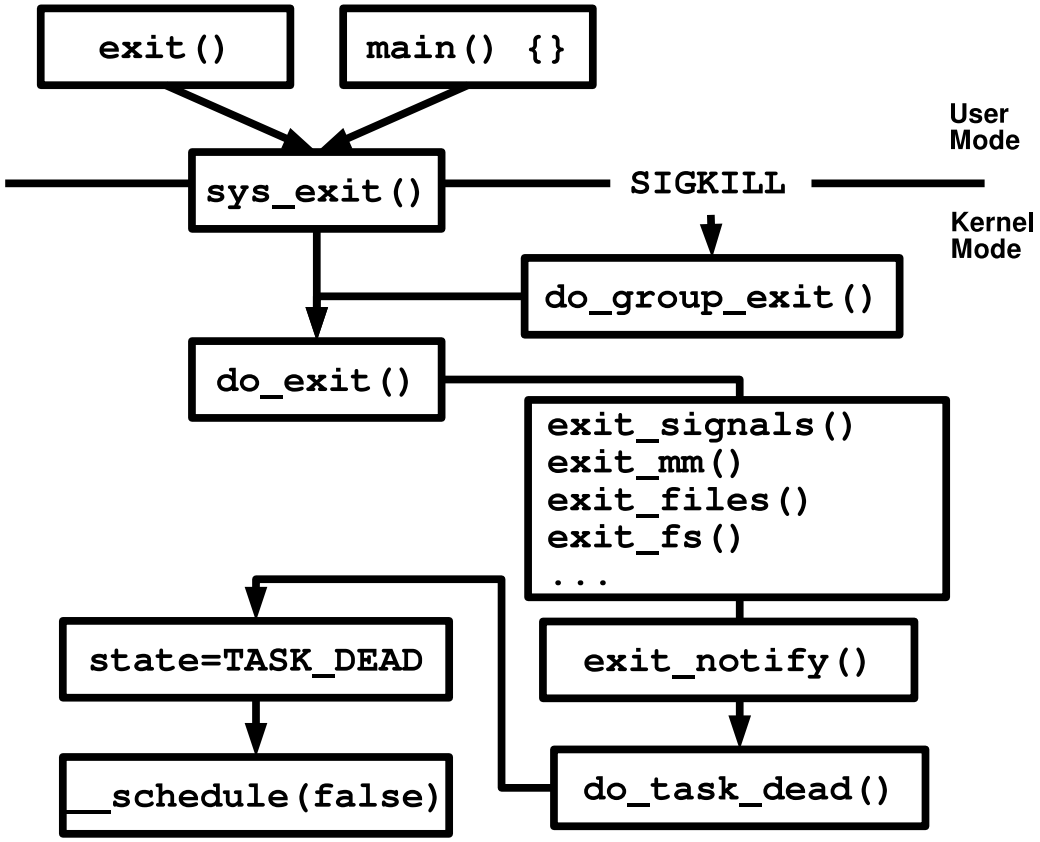
include/linux/interrupt.h

```
struct tasklet_struct
{
  struct tasklet_struct *next;
  unsigned long state;
  atomic_t count;
  void (*func)(unsigned long);
  unsigned long data;
};
```



- Обслуживание «bottom half» для обработки программного прерывания (soft irq)
- Один tasklet одновременно на одном CPU, разные одновременно на разных CPU
- Два приоритера — normal и hi
  - tasklet_schedule и tasklet_hi_schedule
  - «HI» выполняются раньше
- count - 1 - деактивирован

Завершение процесса (kernel/exit.c)



- Нормальное завершение - окончание main()
- Если родитель завершился раньше дочернего (zombie) нужно найти кому отдать код
  - Процессу в группе
  - init

Тренировка! (по fs/exec.c)

- (на экзамене будет)
- Разберите по действиям функцию do_execve()
- Хинт — функция, выполняющая какие либо полезные действия называется.....

(отгадка)
https://elixir.bootlin.com/linux/v5.4.72/source/fs/exec.c#L1895

Примитивы синхронизации Linux

2.8

- Spinlock
- Qspinlock
- Semaphore
- Mutex
- rw_semaphore
- seqlocks

https://0xax.gitbooks.io/linux-insides/content/SyncPrim/

Who are you, mister Spinlock?

include/linux/spinlock_types.h

```
typedef struct spinlock {
  union {
    struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE ...
    struct {
      u8 __padding[LOCK_PADSIZE];
      struct lockdep_map dep_map;
    };
#endif
  };
} spinlock_t;
```

include/linux/spinlock_types.h

```
typedef struct raw_spinlock {
  arch_spinlock_t raw_lock;

#ifdef CONFIG_DEBUG_SPINLOCK
  unsigned int magic, owner_cpu;
  void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
  struct lockdep_map dep_map;
#endif

} raw_spinlock_t;
```

include/asm-generic/qspinlock_types.h

```
typedef struct qspinlock { ... } arch_spinlock_t;
```

Who are you, mister Spinlock?

include/linux/spinlock_types.h

```
typedef struct spinlock {
  union {
    struct raw_spinlock rlock;

#ifdef CONFIG_DEBUG_LOCK_ALLOC
# define LOCK_PADSIZE ...
    struct {
      u8 __padding[LOCK_PADSIZE];
      struct lockdep_map dep_map;
    };
#endif
  };
} spinlock_t;
```

include/linux/spinlock_types.h

```
typedef struct raw_spinlock {
  arch_spinlock_t raw_lock;

#ifdef CONFIG_DEBUG_SPINLOCK
  unsigned int magic, owner_cpu;
  void *owner;
#endif
#ifdef CONFIG_DEBUG_LOCK_ALLOC
  struct lockdep_map dep_map;
#endif

} raw_spinlock_t;
```

include/asm-generic/qspinlock_types.h

```
typedef struct qspinlock { ... } arch_spinlock_t;
```

Реализация и операции

include/asm-generic/qspinlock_types.h

```
typedef struct qspinlock {
  union {
    atomic_t val;
#ifdef __LITTLE_ENDIAN
    struct {
      u8  locked;
      u8  pending;
    };
    struct {
      u16 locked_pending;
      u16 tail;
    };
#else
    // reverse order and alignment
#endif
  };
} arch_spinlock_t;
```

- spin_lock_init
- spin_lock
- spin_lock_bh
  - disables software interrupts and lock
- spin_lock_irqsave
  - disables irq for local CPU and save state
- spin_lock_irq
- spin_unlock
- spin_unlock_bh
- spin_is_locked
- ...

Инициализация spin_lock_init (или еще немного магии)

include/linux/spinlock.h

```
#define spin_lock_init(_lock)\
do { \
   spinlock_check(_lock); \
   raw_spin_lock_init(&(_lock)->rlock); \
} while (0)
```

```
static __always_inline raw_spinlock_t
*spinlock_check(spinlock_t *lock)
{
     return &lock->rlock;
}
```

```
# define raw_spin_lock_init(lock) \
do { \
   *(lock) = __RAW_SPIN_LOCK_UNLOCKED(lock); \
} while (0)
```

```
(*(&(_lock)->rlock)).raw_lock.val = ATOMIC_INIT(0);
```

```
#define __RAW_SPIN_LOCK_UNLOCKED(lockname) \
   (raw_spinlock_t) __RAW_SPIN_LOCK_INITIALIZER(lockname)

#define __RAW_SPIN_LOCK_INITIALIZER(lockname) \
{ \
   .raw_lock = __ARCH_SPIN_LOCK_UNLOCKED, \
   SPIN_DEBUG_INIT(lockname) \
   SPIN_DEP_MAP_INIT(lockname)\
}
```

```
#define __ARCH_SPIN_LOCK_UNLOCKED \
   { { .val = ATOMIC_INIT(0) } }
```

Блокировка и разблокировка

include/linux/spinlock.h

```
static __always_inline void spin_lock(spinlock_t *lock)
{
  raw_spin_lock(&lock->rlock);
}
```

Уличная #define-магия для SMP/NON-SMP

```
static inline void __raw_spin_lock(raw_spinlock_t *lock)
{
  preempt_disable();
  spin_acquire(&lock->dep_map, 0, 0, _RET_IP_);
  LOCK_CONTENDED(lock, \
     do_raw_spin_trylock,\
     do_raw_spin_lock);
}
```

```
static inline void
  __raw_spin_unlock(raw_spinlock_t *lock)
{
  ...
  preempt_enable();
}
```

actual lock

```
#define spin_acquire(l, s, t, i) lock_acquire_exclusive(l, s, t, NULL, i)
#define lock_acquire_exclusive(l, s, t, n, i) lock_acquire(l, s, t, 0, 1, n, i)
```

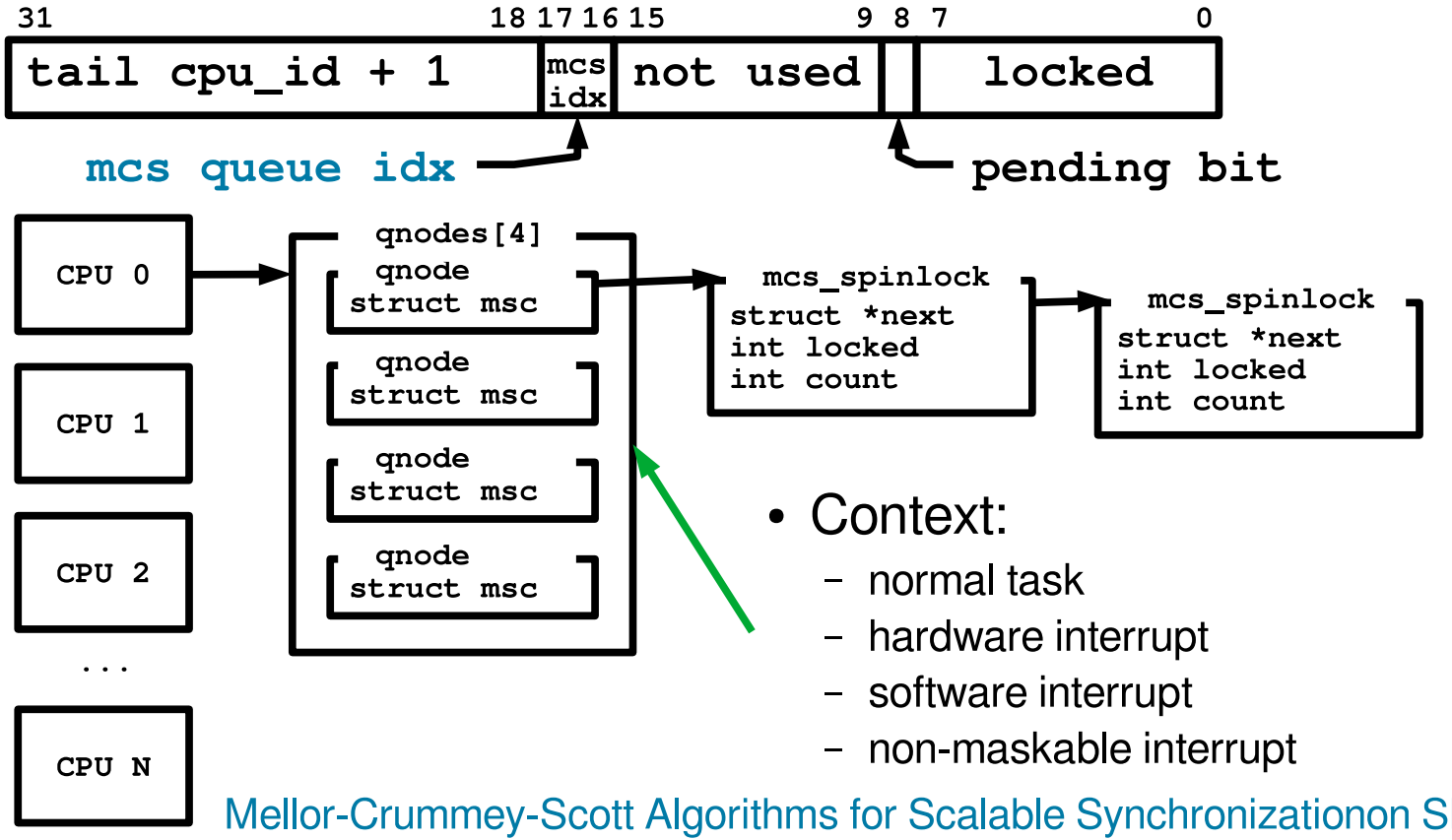
```
void lock_acquire(struct lockdep_map *lock,
                  unsigned int subclass,
                  int trylock, int read, int check,
                  struct lockdep_map *nest_lock,
                  unsigned long ip)
```

lockdep part

```
do_raw_spin_lock()→
arch_spin_lock()
```

```
#define arch_spin_lock(lock)
   queued_spin_lock(lock)
```

Qspinlock



- Context:
  - normal task
  - hardware interrupt
  - software interrupt
  - non-maskable interrupt

- Fast path
  - Uncontented
- Slow path
  - Pending
  - Uncontented queue
  - Contented queue

Mellor-Crummey-Scott Algorithms for Scalable Synchronizationon Shared Memory Multiprocessors

Semaphore

include/linux/semaphore.h

```
struct semaphore {
  raw_spinlock_t lock;
  unsigned int count;
  struct list_head wait_list;
};
```

- void **down**(struct semaphore *sem);
- void **up**(struct semaphore *sem);
- int **down_interruptible**(struct semaphore *sem);
- int **down_killable**(struct semaphore *sem);
- int **down_trylock**(struct semaphore *sem);
- int **down_timeout**(struct semaphore *sem, long jiffies);

```
#define __SEMAPHORE_INITIALIZER(name, n) \
{ \
  .lock    = __RAW_SPIN_LOCK_UNLOCKED((name).lock), \
  .count   = n, \
  .wait_list = LIST_HEAD_INIT((name).wait_list), \
}
static inline void sema_init(struct semaphore *sem, int val)
{
  static struct lock_class_key __key;
  *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
  lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
```

Semaphores up/down

kernel/locking/semaphore.c

```
int down_interruptible(struct semaphore *sem)
{
  unsigned long flags;
  int result = 0;

  raw_spin_lock_irqsave(&sem->lock, flags);
  if (likely(sem->count > 0))
    sem->count--;
  else
    result = __down_interruptible(sem);
    raw_spin_unlock_irqrestore(&sem->lock, flags);

return result;
}
```

kernel/locking/semaphore.c

```
void up(struct semaphore *sem)
{
  unsigned long flags;

  raw_spin_lock_irqsave(&sem→lock, flags);
  if (likely(list_empty(&sem→wait_list)))
    sem->count++;
  else
    __up(sem);
  raw_spin_unlock_irqrestore(&sem→lock,flags);
}
```

```
__down_common(sem,
 TASK_INTERRUPTIBLE,
 MAX_SCHEDULE_TIMEOUT);
```

```
struct semaphore_waiter {
  struct list_head list;
  struct task_struct *task;
  bool up;
};
```

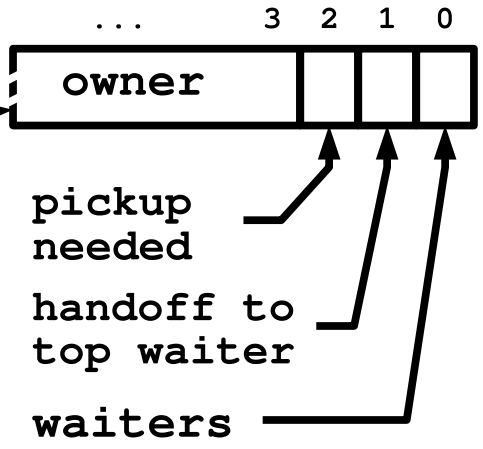
```
static void __up(struct semaphore
*sem)
{
  struct semaphore_waiter *waiter =
    list_first_entry(&sem→wait_list,
    struct semaphore_waiter, list);
  list_del(&waiter->list);
  waiter->up = true;
  wake_up_process(waiter->task);
}
```

```
list_add_tail();
__set_current_state(state);
```

Mutex

include/linux/mutex.h

```
struct mutex {
  atomic_long_t   owner;
  spinlock_t  wait_lock;
#ifdef CONFIG_MUTEX_SPIN_ON_OWNER
  /* Spinner MCS lock */
  struct optimistic_spin_queue osq;
#endif
  struct list_head wait_list;
  //DEBUG STUFF
};
```



- Only one task can hold the mutex at a time.
- Only the owner can unlock the mutex.
- Multiple unlocks are not permitted.
- Recursive locking/unlocking is not permitted.
- A mutex must only be initialized via the API.
- A task may not exit with a mutex held.
- Memory areas where held locks reside must not be freed.
- Held mutexes must not be reinitialized.
- Mutexes may not be used in hardware or software interrupt contexts such as tasklets and timers.

- ww_mutex
  - deadlock free
- Wound-Wait
  - wait-kill

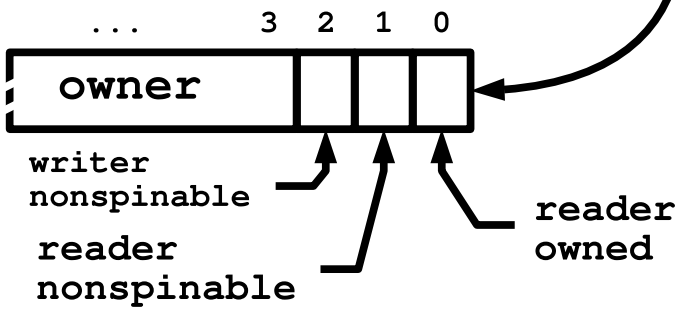
- Fast path
  - uncontented
- Mid path
  - optimistic locking
- Slow path
  - block waiters
- Unlock path
  - wake up

rw_semaphore

include/linux/rwsem.h

```
struct rw_semaphore {
  atomic_long_t count;
  atomic_long_t owner;
#ifdef CONFIG_RWSEM_SPIN_ON_OWNER
  /* spinner MCS lock */
  struct optimistic_spin_queue osq;
#endif
  raw_spinlock_t wait_lock;
  struct list_head wait_list;
  //DEBUG STUFF
};
```

```
Bit   0     - writer locked bit
Bit   1     - waiters present bit
Bit   2     - lock handoff bit
Bits  3-7   - reserved
Bits  8-62  - 55-bit reader count
Bit   63    - read fail bit
```



- void **down_read**(struct rw_semaphore *sem);
- int **down_read_killable**(struct rw_semaphore *sem);
- int **down_read_trylock**(struct rw_semaphore *sem);
- void **down_write**(struct rw_semaphore *sem);
- int **down_write_killable**(struct rw_semaphore *sem);
- int **down_write_trylock**(struct rw_semaphore *sem);
- void **up_read**(struct rw_semaphore *sem);
- void **up_write**(struct rw_semaphore *sem);

Sequence counters and seqlock

- Механизм для реализации неблокирующего чтения (с повторами) и записи без голодания

```
do {
  seq = read_seqbegin(&foo_seqlock);
  /* [[read-side critical section]] */
}
while (read_seqretry(&foo_seqlock, seq));
```

- Несколько версий
  - seqcount_t — запись должна быть синхронизирована внешними средствами
  - seqcount_LOCKTYPE_t — запись синхронизирована средствами выбранного LOCKTYPE
  - seqcount_t с семантикой защелки (две копии данных для четного/нечетного значения счетчика)
  - seqlock_t — запись синхронизирована spinlock и «non preemptible»

Sequence counters and sequential locks

Процессы Windows

2.9

- Типы процессов
- Структуры процесса и заданий
- Состояния процесса потока
- Запуск и завершение

Типы процессов

- «Современные процессы»
  - Universal Windows Platform (UWP) processes (AKA Immersive processes) — Windows 8 (Windows Store)
  - Protected processes (Windows Media Certificate)
    - Protected Processes Light
- Minimal processes
  - Нет пользовательского адресного пространства
  - System process и Memory Compression process
- Pico-процессы
  - Drawbridge project (Pico providers, например Lxss.sys и LxCore.sys)
  - Ограничивают доступ к системным функциям, предоставляют набор callback
- Trustlets - Isolated User Mode (IUM) Processes
  - используют виртуализацию VTL1 (Virtual Trust Level 1), VTL0 — остальная система
- Windows on Windows (WOW) - 32 бита в 64 битном режиме
- JOBS — средство группировки процессов
- Dos, Win16, bat-файлы

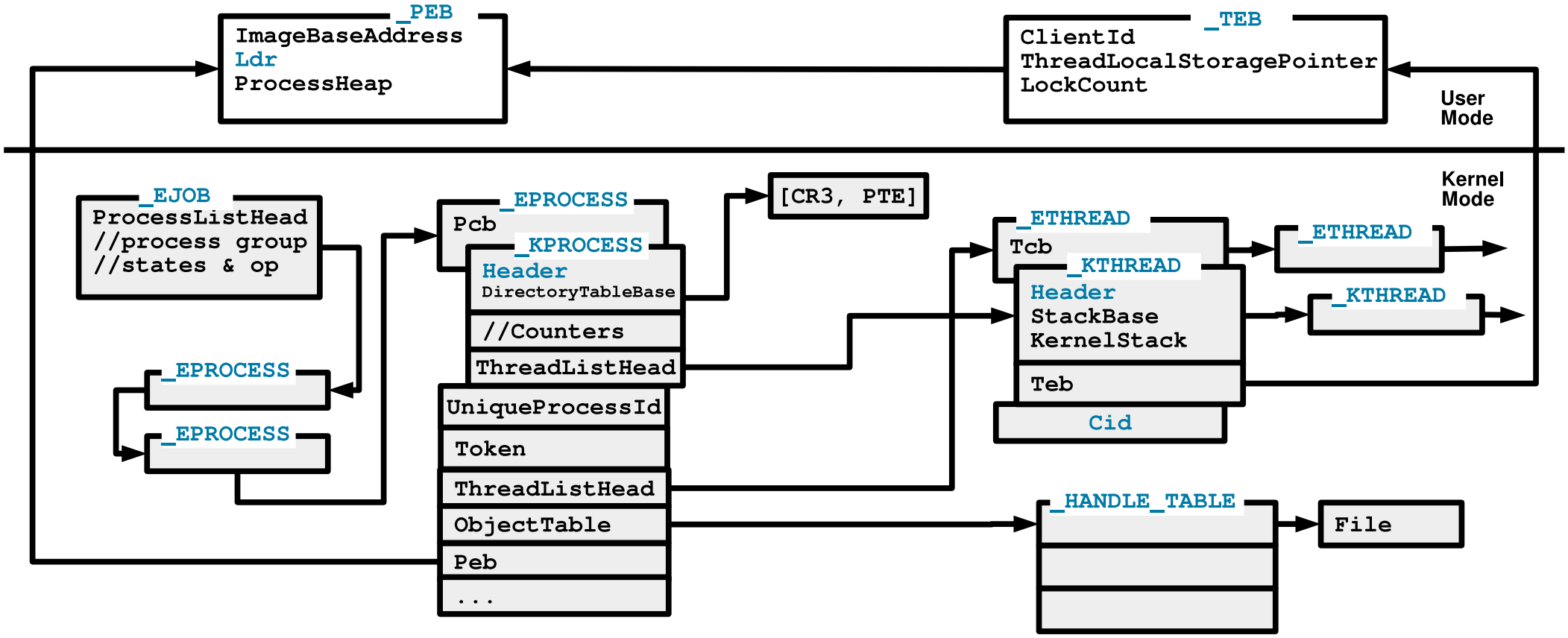
Типы процессов

- «Современные процессы»
  - Universal Windows Platform (UWP) processes (AKA Immersive processes) — Windows 8 (Windows Store)
  - Protected processes (Windows Media Certificate)
    - Protected Processes Light
- Minimal processes
  - Нет пользовательского адресного пространства
  - System process и Memory Compression process
- Pico-процессы
  - Drawbridge project (Pico providers, например Lxss.sys и LxCore.sys)
  - Ограничивают доступ к системным функциям, предоставляют набор callback
- Trustlets - Isolated User Mode (IUM) Processes
  - используют виртуализацию VTL1 (Virtual Trust Level 1), VTL0 — остальная система
- Windows on Windows (WOW) - 32 бита в 64 битном режиме
- JOBS — средство группировки процессов
- Dos, Win16, bat-файлы

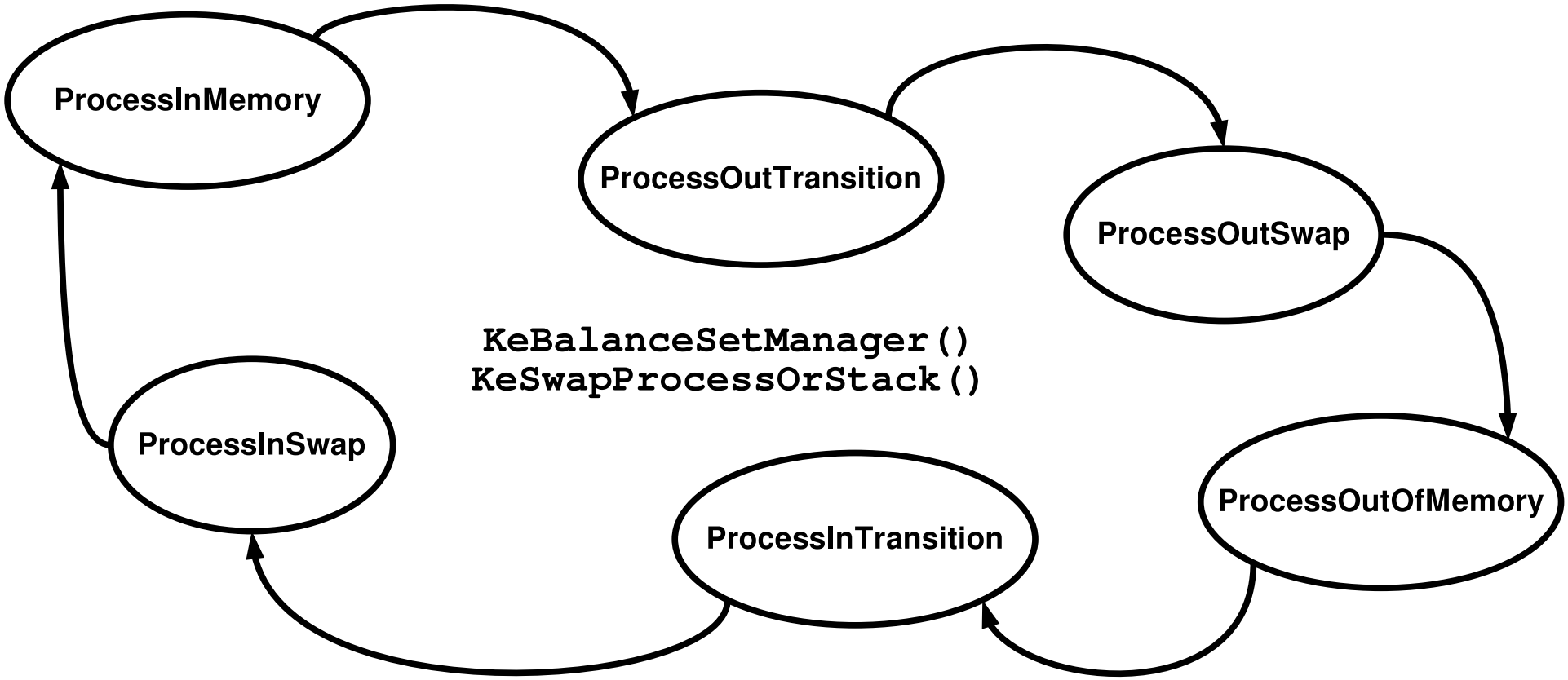
Типы Потоков

- Обычные потоки (1:1 User-Kernel)
  - В т.ч. базовая реализация posix threads
- Fiber — чисто пользовательская реализация потоков, невидимы ядру
  - Kernel32.dll (ConvertThreadToFiber, CreateFiber)
  - Cooperative multitasking, совместно используют один контекст потока ядра
- User-mode scheduling threads (UMS)
  - В 64 битной версии
  - Есть контекст потока в ядре, поэтому можно получить управление при блокирующем вызове потока, можно использовать несколько процессоров
- Asyncronious Procedure Call (APC) и Deffered Procedure Call (DPC)

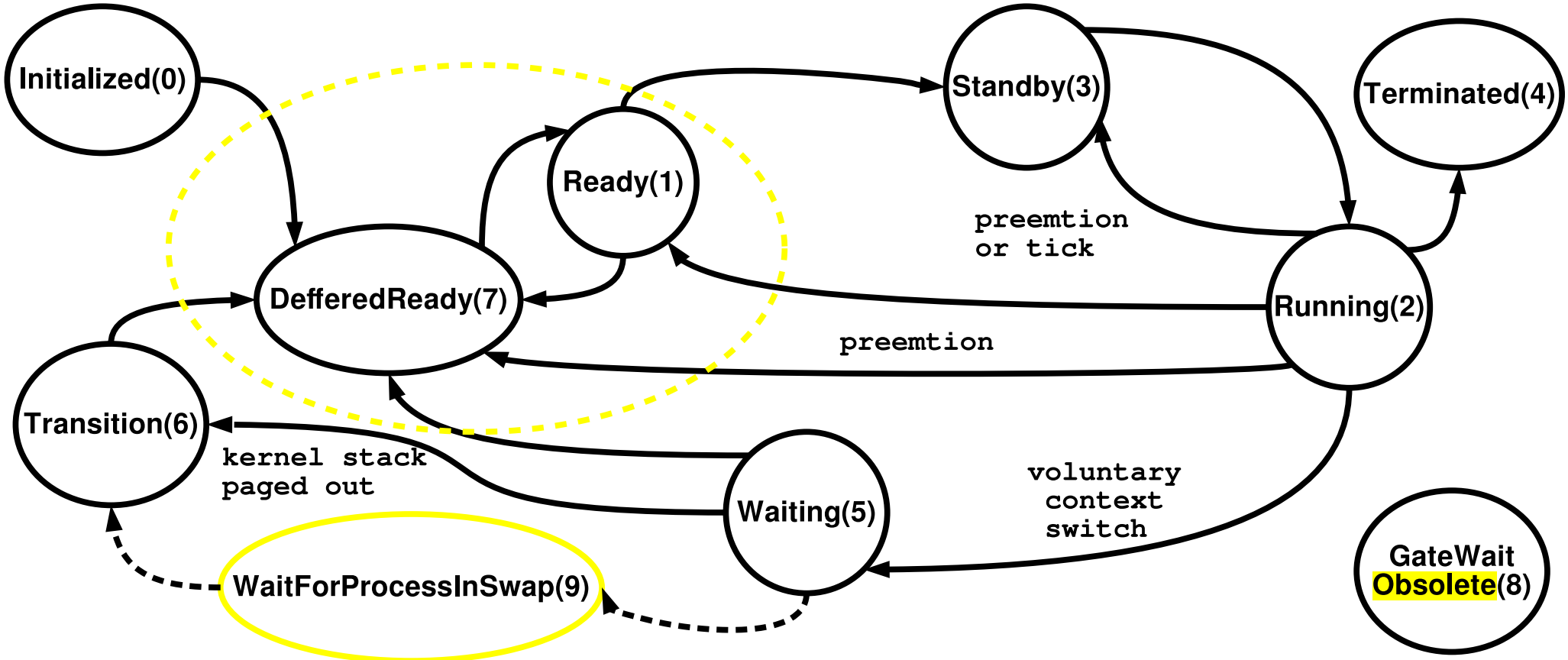
Windows Process Structures



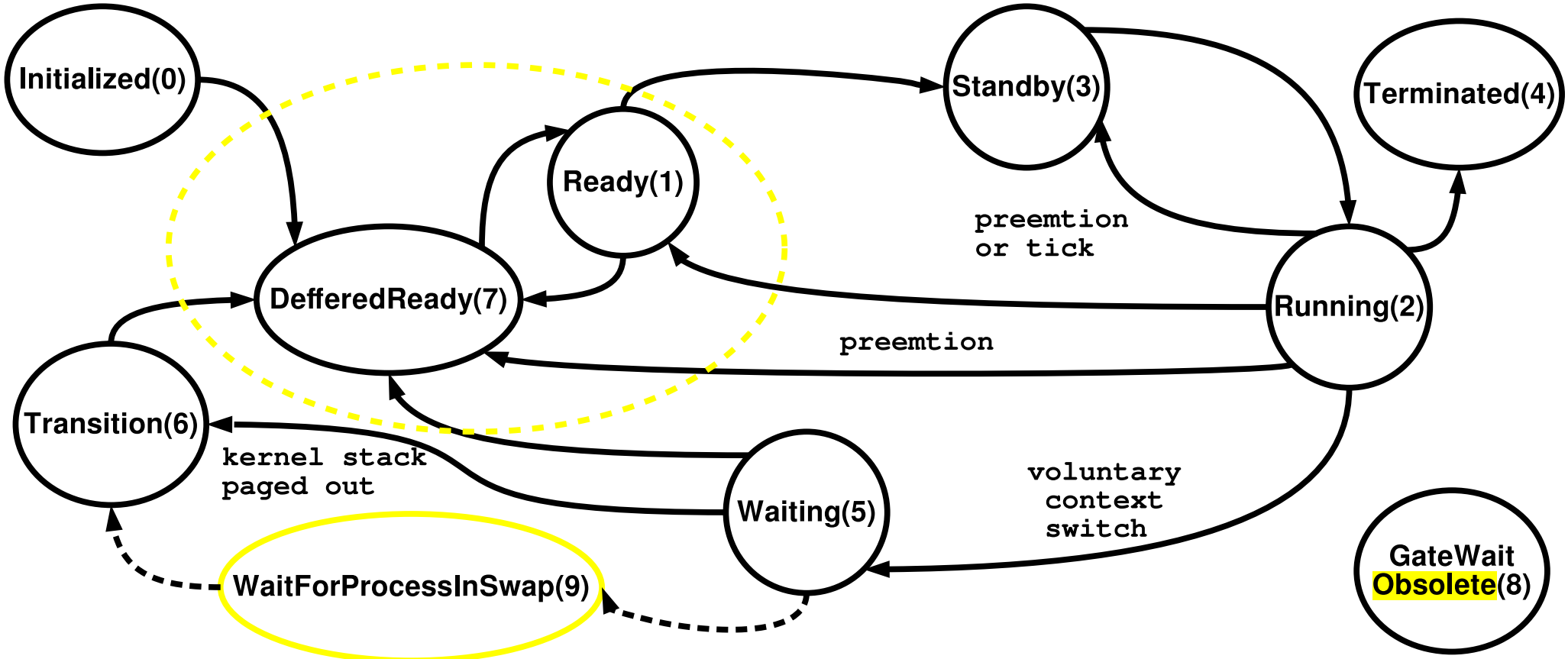
_KPROCESS_STATE



_KTHREAD_STATE



_KTHREAD_STATE



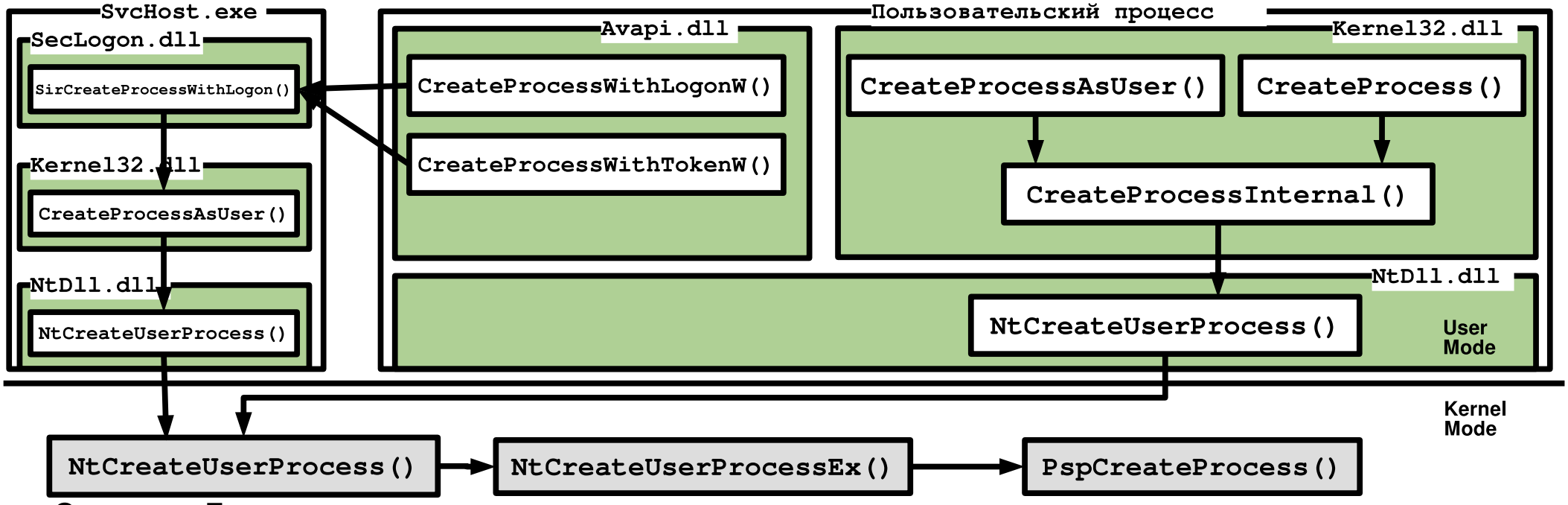
Поля структур процесса

- Поля _EPROCESS
  - PCB
  - Защищающие блокировки
  - UniqueProcessID
  - Сылки списка процессов (начало в PsActiveProcessHead)
  - Флаги
  - Времена создания и завершения
  - Информация о квотах
  - Ссылка на сессию
  - Основной токен доступа
  - Ссылка на задание
  - Объекты процесса (Handle Table)
  - Окружение процесс (PEB)
  - Имя фала образа процесса
  - Счетчики производительности
  - Список потоков
  - Win32K структура
  - WOW64 структура
  - Pico Context
  - DirectX process
  - Набор рабочих страниц WorkingSet
  - Секции (сегменты) образа
- Поля _KPROCESS
  - Заголовок диспетчера
  - Ссылка на таблицу страниц
  - Kernel/User/Cycle Times
  - Context Switches
  - Список Thread
  - Аффинити
  - Флаги
  - Базовый приоритет

Поля структур потока

- Поля _ETHREAD
  - TCB
  - Время создания и завершения
  - Ссылка на процесс
  - Флаги потока
  - Токен доступа
  - Стартовый адрес
  - Ожидаемые запросы ввода-вывода
  - Таймеры
  - Код выхода
  - Потребление энергии
  - Процессорный набор
- Поля _KTHREAD
  - Заголовок диспетчера
  - TEB
  - Kernel/User Time
  - Указатель на SSDT (system services descriptor table)
  - Freeze/Suspend Count
  - Win32 Thread Object
  - Процесс (_KPROCESS)
  - Иформация о стеках
  - Информация планировщика
  - Фрейм прерывания
  - Информация о синхронизации
  - Информация о wait
  - Список объектов ожидания для потока
  - Список APC, ожидающих обработки

Создание процесса



- Открывает Exe
- Создает память
- Создает структуру процесса
- Создает основной поток

- Уведомляет Windows о создании
- Запускает новый поток и завершается. В новом потоке:
  - Финальные настройки
  - Запуск со стартового адреса в образе процесса

Завершение процессов

- Корректное завершение - Exit Process()
- Прекращение функционирования процесса другим процессом TerminateProcess()
- Последовательность:
  1. Оповещение DLL
     - Если не использован TerminateProcess()
  2. Закрываются все handles и kernel objects
  3. Закрываются все активные потоки
  4. Код возврата изменяется с STILL_ACTIVE на указанный
  5. Когда все ссылки на процесс == 0, объект процесса удаляется

Префиксы функций

- A1pc - Расширенные локальные вызовы процедур
- Cc - Общий кэш
- Cm - Диспетчер конфигурации
- Dbg - Поддержка отладки ядра
- Dbgk - Отладочная инф раструктура для пользовательского режима
- Em - Диспетчер ошибок
- Etw - Трассировка событий для Windows
- Ex - Исполнительные вспомогательные функции
- FsRtl - Библиотека файловой системы времени выполнения

- FsRtl - Библиотека файловой системы времени выполнения

- Hv - Библиотека Hive
- Hvl - Библиотека гипервизора
- Io - Диспетчер ввода/вывода
- Kd - Отладчик ядра
- Ke - Ядро
- Ki - Внутренние функции ядра
- Kse - Оболочка совместимости ядра
- Lsa - Локальная система безопасности
- Psp - внутренние вспомогательные функции процессов.
- Mm - Диспетчер памятиClick to add Text

Примитивы синхронизации Windows

2.10

- Объекты диспетчера
- Состояние ожидания
- События, мьютексы, семафоры ....
- Spinlocks

https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/

Kernel Object = { Dispatcher Object, Control Object }

- Любой объект ядра, на котором можно ожидать события - «dispatcher object»
  - Некоторые предназначены только для синхронизации (events, mutexes, semaphores, queues, ...)
  - Другие для ожидания событий от процессов, потоков, файлов ....
- Общая структура данных
- Два состояния
  - «Signaled» - ожидание удовлетворено и «Not-signaled» - еще нет.
  - Различные объекты отличаются в том, что именно изменяет их состояние
  - Wait и unwait — операции общие.

ntos/inc/ntosdef.h

```
typedef struct
_DISPATCHER_HEADER {
  union {
    struct {
      UCHAR Type;
      union {
        UCHAR Absolute;
        UCHAR NpxIrql;
      };
      union {
        UCHAR Size;
        UCHAR Hand;
      };
      union {
        UCHAR Inserted;
        BOOLEAN DebugActive;
      };
    };
    volatile LONG Lock;
  };
  LONG SignalState;
  LIST_ENTRY WaitListHead;
} DISPATCHER_HEADER;
```

В win10 больше!

Вызовы Wait

- Гибкие вызовы ожидания — KeWaitForSingleObject, KeWaitForMultipleObjects
  - Ожидает одного или нескольких объектов (“any” или “all”). В случае всех — все объекты должны быть в состоянии «signaled»
  - Таймаут задается опционально
- Объекты, которые можно ожидать:
  - Events, Mutexes, Semaphores, Timers
  - Processes and Threads (exit or terminate)
  - Directories (change notification)
- Нет гарантированного порядка завершения ожидания
  - Потоки ожидающие события пробуждаются в неопределенном порядке, зависит от диспетчера
  - Обычный порядок - FIFO; Однако APC может изменить этот порядок

ntos/inc/ke.h

```
typedef enum _KWAIT_REASON {
    Executive,
    FreePage,
    PageIn,
    PoolAllocation,
    DelayExecution,
    Suspended,
    UserRequest,
    WrExecutive,
    WrFreePage,
    WrPageIn,
    WrPoolAllocation,
    WrDelayExecution,
    WrSuspended,
    WrUserRequest,
    WrEventPair,
    WrQueue,
    WrLpcReceive,
    WrLpcReply,
    WrVirtualMemory,
    WrPageOut,
    WrRendezvous,
    Spare2,
    Spare3,
    Spare4,
    Spare5,
    Spare6,
    WrKernel,
    WrResource,
    WrPushLock,
    WrMutex,
    WrQuantumEnd,
    WrDispatchInt,
    WrPreempted,
    WrYieldExecution,
    WrFastMutex,
    WrGuardedMutex,
    WrRundown,
    MaximumWaitReason
} KWAIT_REASON;RE;
```

EventObject

- Может сбрасываться вручную, автоматически и «pulsed»
- Используется для оповещении о наступления события двух типов:
  - Синхронизация — один из ждунов продолжает выполнение когда наступает состояние Signaled. EO автоматически сбрасывается в Not-Signaled
  - Нотификация — пробуждает всех ждунов, требует ручного сброса состояния
- Операции:
  - KeInitializeEvent - Initialize an event object
  - KePulseEvent - Set/reset event object state atomically
  - KeReadStateEvent - Read state of event object
  - KeResetEvent - Set event object to Not-Signaled state
  - KeSetEvent - Set event object to Signaled state

ntos/inc/ntosdef.h

```
typedef struct _KEVENT {
  DISPATCHER_HEADER Header;
} KEVENT, *PKEVENT, *PRKEVENT;
```

Mutexes и Mutants

- Mutually exclusive, deadlock free доступ к разделяемым ресурсам
- В нормальном режиме создается в «signaled» состоянии
- Возможен рекурсивный захват (сколько захватов, столько освобождений)
- Захваченный mutex блокирует выход из Kernel Mode
- Снятие Lock — mutant: любой поток (abandoned state); mutex: владелец
- В mutex запрещены APC, в mutant — нет.
- Возможно использовать mutant в UserLand
- Increment — добавлятся к приоритету потока если wait satisfied
- Wait — за KeRelease* сразу следует функция ожидания (освободить и занять в рамках «атомарной операции»)

ntos/inc/ke.h

```
typedef struct _KMUTANT {
  DISPATCHER_HEADER Header;
  LIST_ENTRY MutantListEntry;
  struct _KTHREAD *OwnerThread;
  BOOLEAN Abandoned;
  UCHAR ApcDisable;
} KMUTANT,KMUTEX;
```

- Mutant:
  - KeInitializeMutant
  - KeReadStateMutant
  - KeReleaseMutant
    - KPRIORITY Increment,
    - BOOLEAN Abandoned,
    - BOOLEAN Wait
- Mutex:
  - KeInitializeMutex
  - KeReadStateMutex
  - KeReleaseMutex
    - KeReleaseMutant(Mutex, 1, FALSE, Wait)

Fast mutexts и Guarded mutexes

- Больше похожи на «нормальные» мьютексы
- Поле Count: 0-й бит — lock, 1-й — single waiter woken
- Нельзя захватывать рекурсивно
- Операции:
  - ExInitializeFastMutex
  - ExAcquireFastMutex
  - ExTryToAcquireFastMutex
  - ExReleaseFastMutex
- До Windows 8 — разные реализации, после - идентичны

ntos/inc/ex.h

```
typedef struct _FAST_MUTEX {
    LONG Count;
    PKTHREAD Owner;
    ULONG Contention;
    KEVENT Event;
    ULONG OldIrql;
} FAST_MUTEX, *PFAST_MUTEX;
```

Semaphores

- Управляют захватом разделяемого ресурса
  - Limit — максимальное количество такого ресурса
- Count — начальное состояние, помещается в Header.SignaledState
- Семафор открыт, когда SignaledState > 0
- Если SignaledState+Adjustment > Semaphore→Limit
  - вызывается ExRaiseStatus(STATUS_SEMAPHORE_LIMIT_EXCEEDED)
- Increment — добавлятся к приоритету потока если wait satisfied
- Wait — за KeReleaseSemaphore сразу следует функция ожидания (освободить и занять в рамках «атомарной операции»)

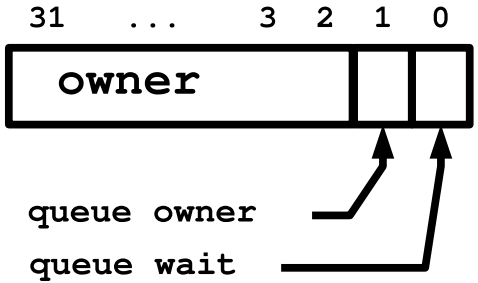
ntos/inc/ke.h

```
typedef struct _KSEMAPHORE {
  DISPATCHER_HEADER Header;
  LONG Limit;
} KSEMAPHORE;
```

- KeInitializeSemaphore
  - LONG Count
  - LONG Limit
- KeReadStateSemaphore
- KeReleaseSemaphore
  - KPRIORITY Increment,
  - LONG Adjustment,
  - BOOLEAN Wait

Spinlock (Queued Spinlock)

- Используется атомарная опеация test-and-modify
  - Синхронизация на многопоцессорных системах при помощи локальной переменной
- Один владелец в один момент времени
- Реализация - одна ячейка памяти
  - `typedef ULONG_PTR KSPIN_LOCK;`
- Для Windows > XP рекомендуется использовать AcquireInStack



- ntos/ke/amd64/queuelock.c
  - KeAcquireQueuedSpinLock
  - KeAcquireQueuedSpinLockRaiseToSynch
  - KeAcquireQueuedSpinLockAtDpcLevel
  - KeTryToAcquireQueuedSpinLock
  - KeTryToAcquireQueuedSpinLockRaiseToSynch
  - KeTryToAcquireQueuedSpinLockAtRaisedIrql
  - KeReleaseQueuedSpinLock
  - KeReleaseQueuedSpinLockFromDpcLevel
  - KeAcquireInStackQueuedSpinLock
  - KeAcquireInStackQueuedSpinLockRaiseToSynch
  - KeAcquireInStackQueuedSpinLockAtDpcLevel
  - KeReleaseInStackQueuedSpinLock
  - KeReleaseInStackQueuedSpinLockFromDpcLevel

https://docs.microsoft.com/en-us/windows-hardware/drivers/kernel/introduction-to-spin-locks