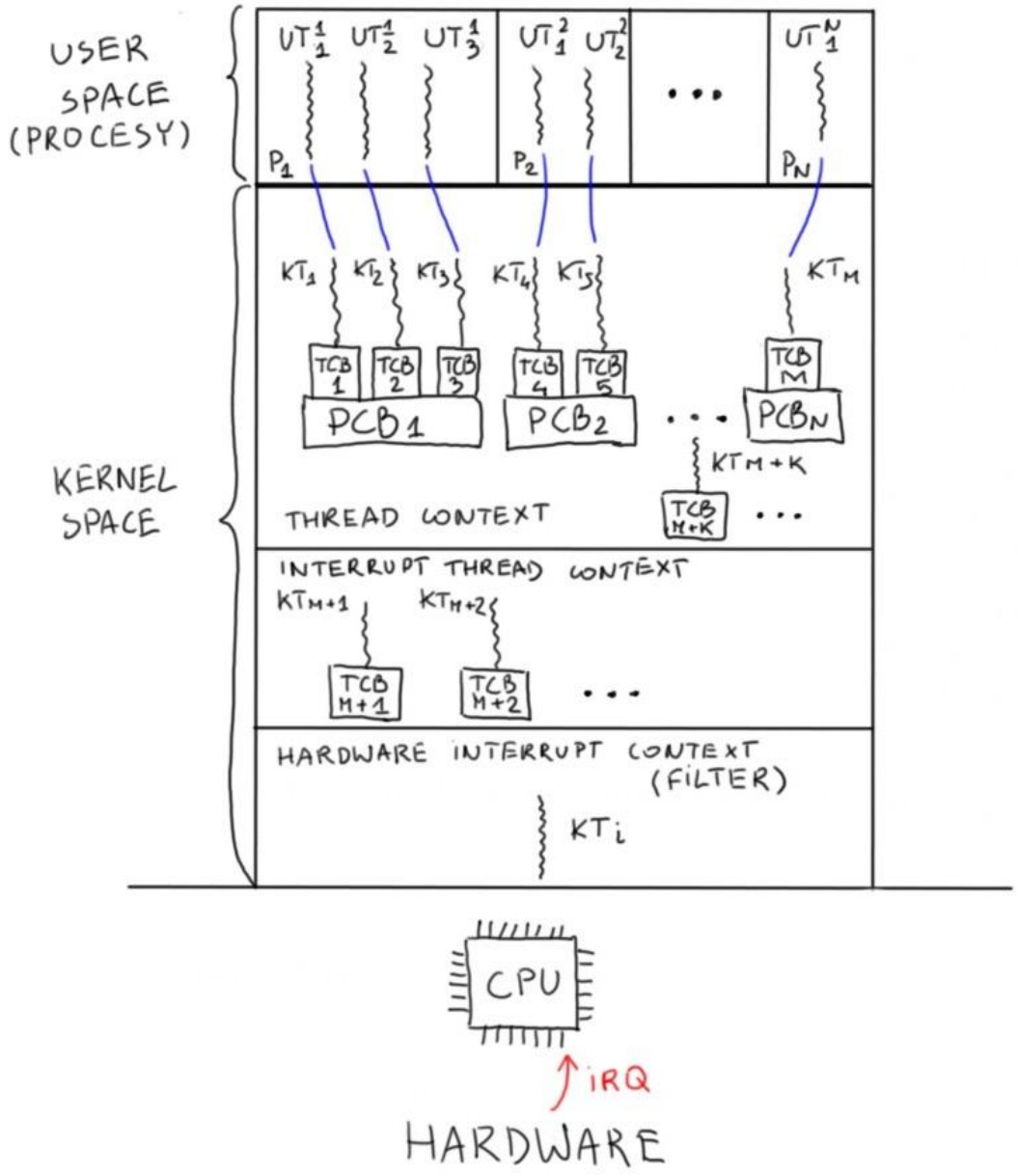
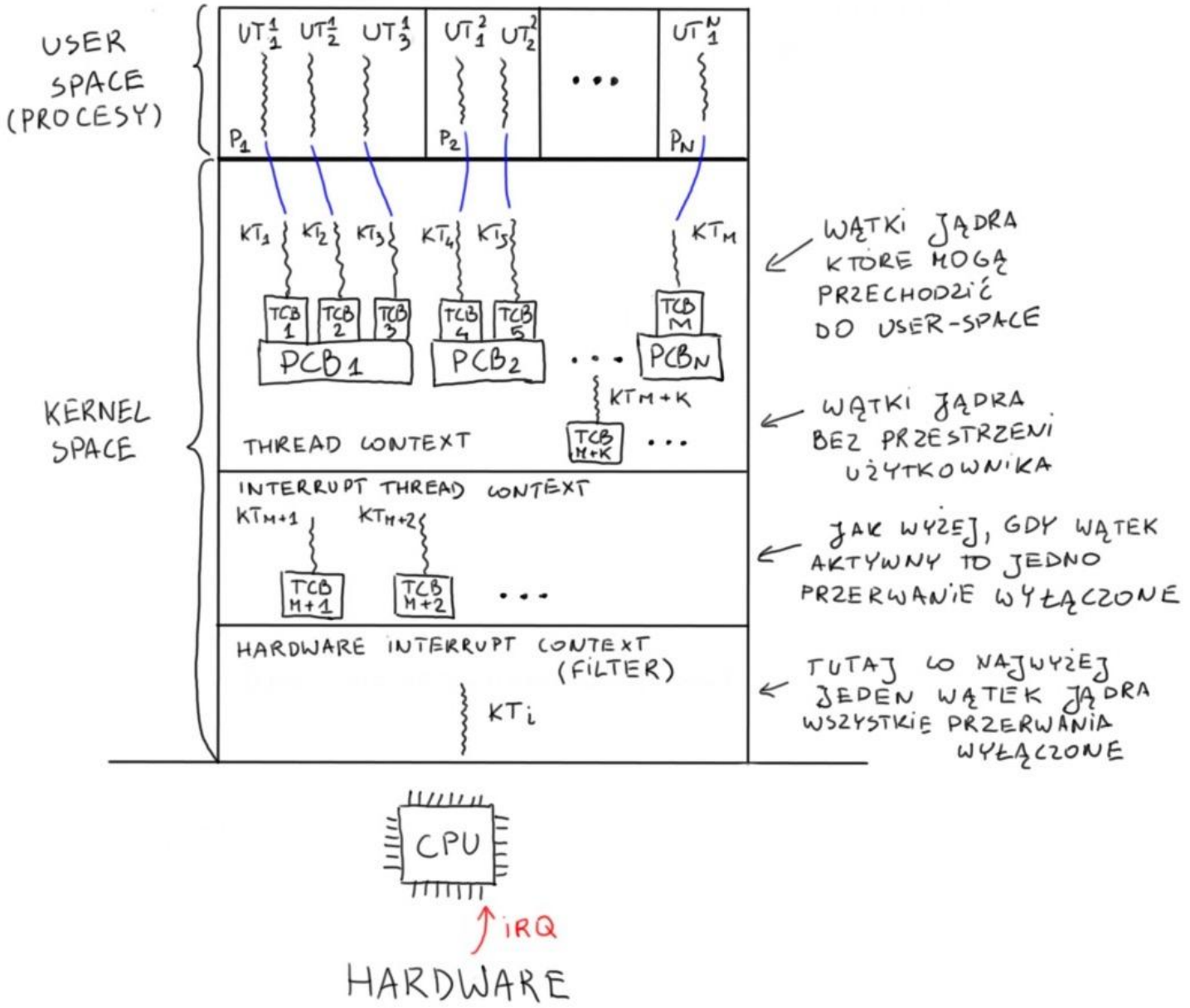
Struktura jądra UNIX

Wykład 7-8: Środki synchronizacji

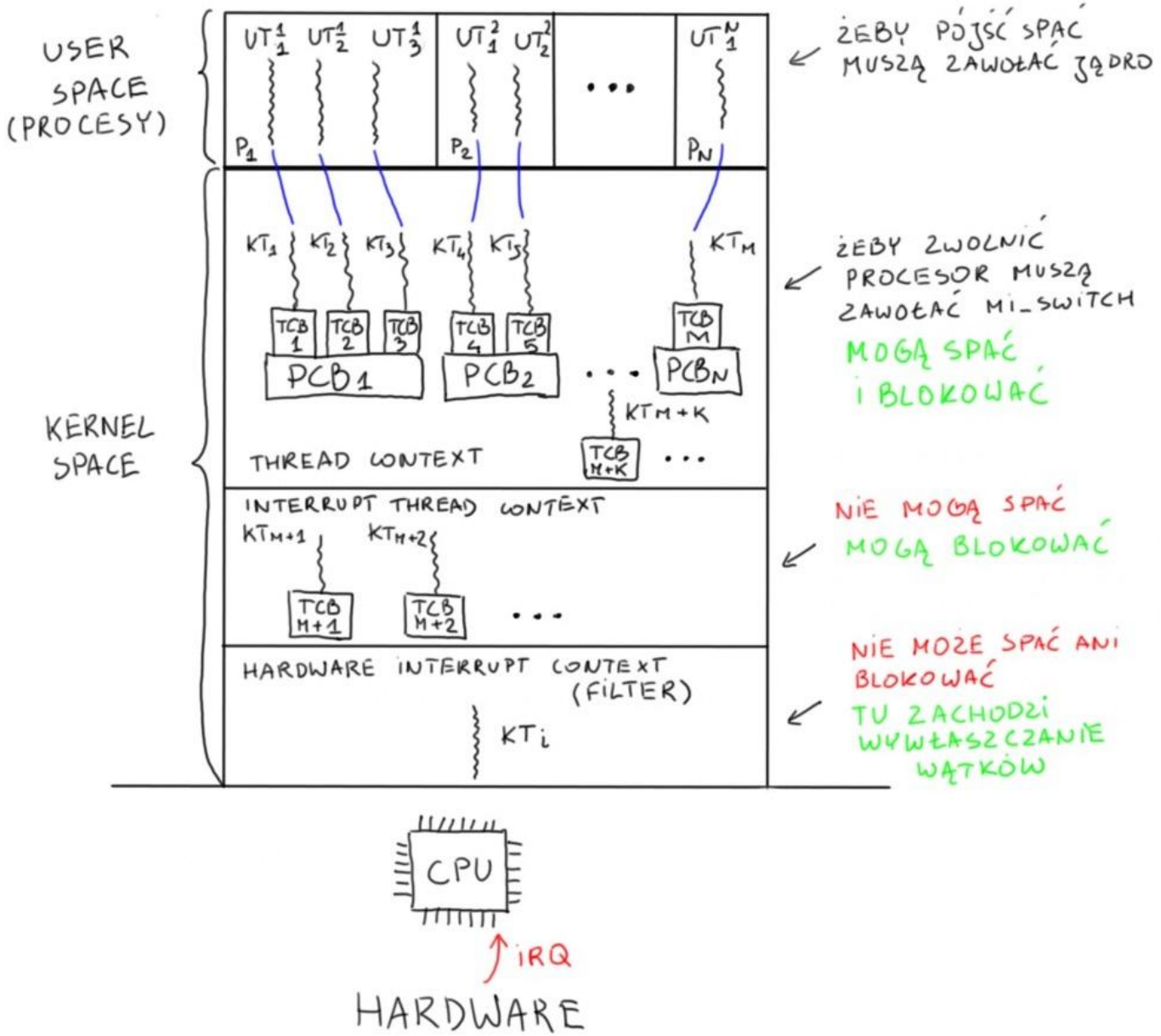
Pejzaż systemu jednoprocesorowego



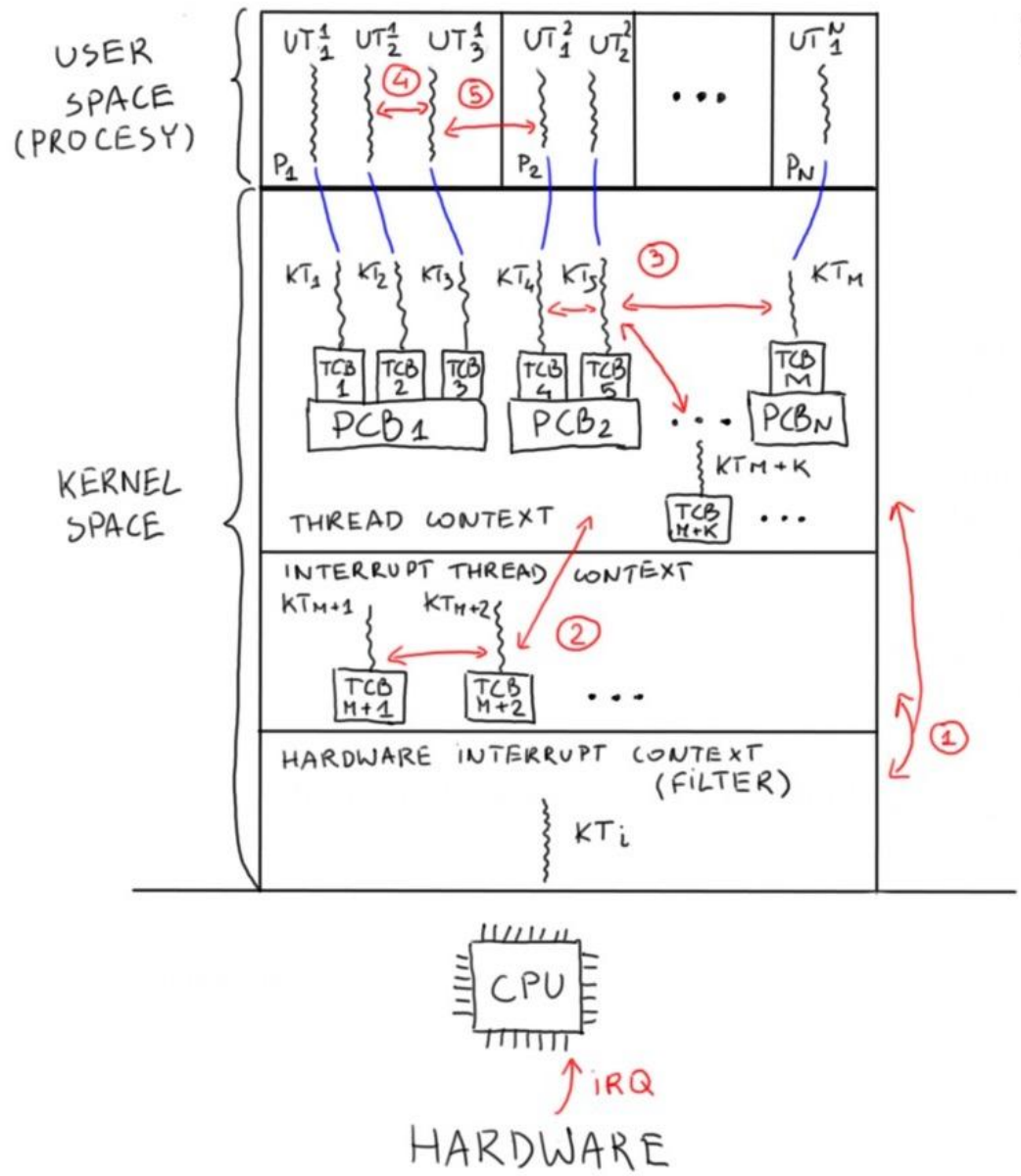
Pejzaż systemu jednoprocesorowego (c.d.)



Co już wiemy o blokadach?



Blokady w systemie jednoprocesorowym



JAK ZREALIZOWAĆ SEKCJĘ KRYTYCZNĄ

① WYŁĄCZANIE PRZERWAŃ

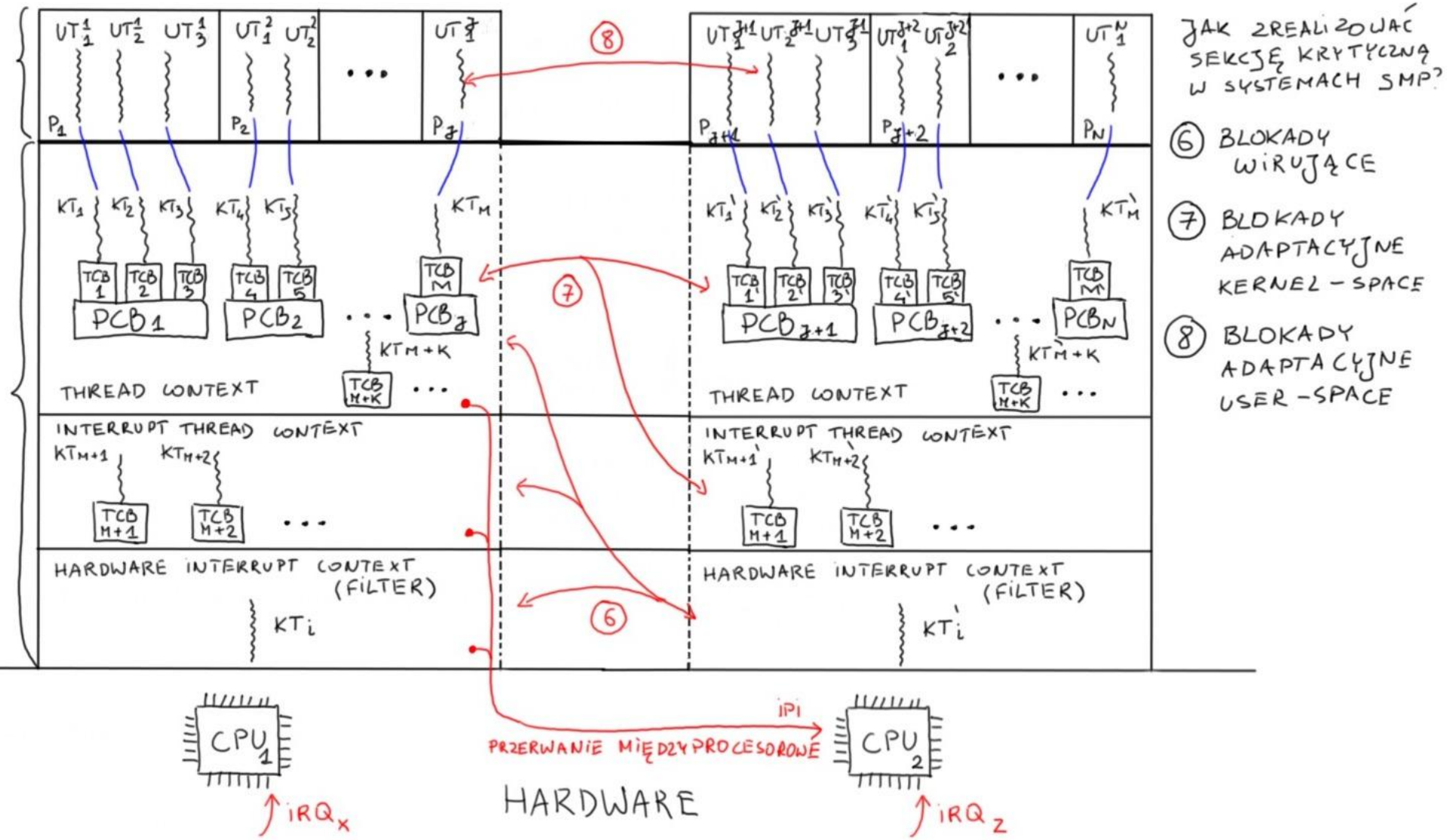
② BLOKADY JĄDRA ZE SNEM OGRANICZONYM
- SLEEP MUTEX
- RW LOCK

③ WSZYSTKIE DOSTĘPNE ŚRODKI SYNCHRONIZACJI JĄDRA:
- CONDVAR
- SX LOCK
- ...

④ ŚRODKI SYNCHRONIZACJI WĄTKÓW UŻYTKOWNIKA
- PTHREAD – MUTEX / CONDVAR
- FUTEX (LINUX)

⑤ IPC
- SYGNAŁY
- SEMAFORY POSIX
- KOLEJKI KOMUNIKATÓW POSIX

Blokady w systemie wieloprocesorowym



Wyłączanie przerwań

Jeśli wyłączymy przerwania to mamy wyłączność na używanie jednego rdzenia (ang. *CPU core*)! Problemy:

1. nie można udostępnić dla programów użytkownika,
2. powoduje opóźnienie w obsłudze przerwań,
3. nie chroni przed wyścigami, jeśli maszyna wieloprocesorowa.

Przydatne **wyłącznie w jądrze** (NetBSD: spl) do:

1. synchronizacji kodu obsługi przerwań z resztą systemu,
2. implementacji blokad wirujących (ang. *spin lock*).

We FreeBSD: `intr_enable`, `intr_disable` (machine dependant).

Blokady wirujące

Wyłączanie przerwań jest lokalne dla danego rdzenia.
Poza tym nie możemy tak po prostu zatrzymać innego procesora!

Jak synchronizować kod jądra między procesorami?

Rdzeń wchodzi pod *spinlock* → konkurujące rdzenie się “pętlą”.
Działa z wyłączonymi przerwaniami → nie może się zablokować!

Wykorzystywane w jądrze do zapobiegania wyścigom
na strukturach danych współdzielonych przez procesory.

Implementacja wykorzystuje **instrukcje atomowe** i wymaga
sprzętowej synchronizacji pamięci podręcznych (MOESI).

Instrukcje atomowe

```
int compare_and_swap(int* reg, int oldval, int newval) {
  int old_reg_val = *reg;
  if (old_reg_val == oldval)
     *reg = newval;
  return old_reg_val;
}
```

GCC udostępnia funkcje (atomic) tłumaczone na instrukcje atomowe:

```
T __sync_val_compare_and_swap(T *ptr, T oldval, T newval)
T __sync_lock_test_and_set(T *ptr, T value)
```

Dla standardu C11 funkcje w pliku nagłówkowym `stdatomic.h`.

Jądro FreeBSD posiada własny plik nagłówkowy `atomic.h`, zdefiniowany dla każdej architektury (machine dependant).

Model spójności pamięci (ASK)

Rdzenie procesora mają pamięć podręczną i wykonują instrukcje poza porządkiem programu. System musi działać tak, by rdzenie widziały spójną pamięć → **model pamięci** (np. Total Store Order w x86-64).

Pytanie: W jakiej kolejności procesor obserwuje zapisy wykonywane przez inne rdzenie? Jakie właściwości chcemy wymusić na operacjach dostępu do różnych komórek pamięci.

Problem: Im "ładniejszy" model tym wolniej działa :-(

Zmienne w pamięci:

`*A = 0; *B = 0;`

Uruchamiamy **P1** i **P2**, czy obydwa mogą zawołać **X**?

```
P1() {
  *A = 1;
  ...
  if (*B == 0)
    X();
}
```

```
P2() {
  *B = 1;
  ...
  if (*A == 0)
    X();
}
```

Sprzętowe wsparcie do realizacji blokad w NetBSD

1. Operacje atomowe atomic_ops(3):
   a. arytmetyczne: atomic_add, atomic_dec, atomic_inc,
   b. logiczne: atomic_and, atomic_or,
   c. zamiana: atomic_cas, atomic_swap.
2. Bariery pamięciowe membar_ops(3):
   a. membar_enter: wszystkie zapisy przed barierą będą obserwowalne globalnie przed realizacją odczytów i zapisów po barierze (W → RW),
   b. membar_exit: wszystkie odczyty i zapisy poprzedzające blokadę będą obserwowalne globalnie przed realizacją zapisów po barierze (RW → W),
   c. membar_producer: j.w. ale W → W,
   d. membar_consumer: j.w. ale R → R.

Przykład: Procedury membar_enter i membar_exit są z reguły używane w implementacji blokad odpowiednio w trakcie zakładania i zdejmowania blokady.

Wyłączanie wywłaszczania

Przerwania (w tym zegarowe) zostawiamy włączone, ale nie uruchamiamy dyspozytora w trakcie powrotu z przerwania.

W NetBSD: `KPREEMPT_DISABLE` i `KPREEMPT_ENABLE`.

We FreeBSD: `critical_enter`, `critical_leave`.

Problem? Być może w trakcie wykonywania kodu z wyłączonym wywłaszczaniem wygasł nam kwant czasu. Po wyjściu należy sprawdzić czy to nie czas oddać się w ręce planisty wątków.

Znów nie nadaje się dla programów użytkownika!

Używane wewnątrz jądra do implementacji innych metod synchronizacji takich jak: `mutex`, `condvar`, …

Przerwania międzyprocesorowe `ipi(9)`

1. Potrzebujemy synchronizować konfigurację procesorów, ale "z zewnątrz" nie mamy dostępu do rejestrów konfiguracyjnych.
   a. **zestrzeliwanie wpisów TLB** (ang. *TLB shootdown*) IPI_SHOOTDOWN,
   b. synchronizacja częstotliwości taktowania.
2. Wieloprocesorowy planista zadań musi umieć **wypychać zadania** na uśpiony procesor (ang. *push migration*), tj. włożyć mu coś do jego **`runqueue`** i wybudzić.
3. Zmieniamy priorytet wątku (uruchomiony na innym procesorze) na niższy → chcemy go wywłaszczyć `IPI_KPREEMPT`.

API asynchroniczne: **`ipi_trigger(_.*)?`**; oraz asynchroniczny: **`ipi_(.*)cast`**, **`ipi_wait`**. API wysokopoziomowe: `xcall(9)`.

Hierarchia blokad we FreeBSD

Type	Group	Description
witness	sleep	partially ordered sleep locks
lock manager	sleep	drainable shared/exclusive access
condition variable	sleep	event-based thread blocking
shared-exclusive lock	sleep	shared and exclusive access
read-mostly lock	block	optimized for read access
reader-writer lock	block	shared and exclusive access
sleep mutex	block	spin for a while, then sleep
spin mutex	spin	spin lock
compare-and-swap	hardware	memory-interlocked instructions
memory barrier	hardware	guarantee load-store ordering in SMP

FreeBSD: Synchronizacja w jądrze

Po co aż tyle środków synchronizacji? W `locking(9)` napisano:

> **BUGS**
>
> There are too many locking primitives to choose from.

1. interakcja z przerwaniami
2. obsługa przerwania sygnałami
3. debugowanie i diagnostyka zakleszczeń
4. propagacja priorytetów
5. optymalizacje pod określony scenariusz użycia
6. dodatkowe funkcje (drain, upgrade)

Przykłady zabronionych operacji na blokadach

- Nie można przełączyć kontekstu (`yield`) gdy:
  - wykonujemy **`filter`** w kontekście obsługi przerwania sprzętowego
  - trzymamy blokadę wirującą
- Nie można przejść w sen nieograniczony trzymając środek synchronizacji z grupy blokujących lub wirujących, np.:
  - założyć blokadę na wyłączność na `sx(9)` posiadając `mutex(9)`
  - przydzielać pamięć `malloc(9)` z flagą `M_WAITOK` trzymając `rwlock(9)`
  - korzystać z `lockmgr(9)` w kontekście wątku przerwania.

Uwaga! Blokada przekazywana do `sleep()` lub `cv_wait()` jest zwalniana przed wprowadzeniem wątku w stan nieograniczonego snu.

Pułapka! Nie można robić dostępów do pamięci stronicowalnej trzymając środek synchronizacji z grupy wirujących i blokujących! Dlaczego?

Wprowadzanie w sen nieograniczony

FreeBSD: Przykłady wykorzystania `sleep` / `wakeup`

Kiedy oczekujemy na zasób lub zdarzenie to `wchan` jest adresem struktury danych, która je jednoznacznie identyfikuje.

Przykład 1: Wątek używa adresu bufora dyskowego jako `wchan` kiedy idzie spać w oczekiwaniu na wypełnienie bufora przez sterownik dysku. Po wypełnieniu bufora sterownik wybudza wątek.

Przykład 2: Rodzic oczekujący na zakończenie dzieci wywołaniem `waitpid(2)` nie zna pierwszego, które się zakończy → nie może użyć `PCB` jednego z potmków jako `wchan`. Ponieważ może iść spać tylko na jednym adresie to wybiera własne `PCB`. Dzieci kończące swe działanie wybudzają rodzica używając adresu jego `PCB`. Wybudzony rodzic skanuje listę swoich potomków w poszukiwaniu dziecka do pogrzebania.

FreeBSD: Usypianie wątków

```
int tsleep(void *chan, int prio,
           const char *wmesg, int timo);
```

Implementacja `tsleep` korzysta ze struktury `sleepqueue`.

Usypiamy wątek w punkcie oczekiwania (ang. *wait channel*), czyli umownym adresie zasobu, który jest czasowo niedostępny.

Po wybudzeniu dostaniemy odpowiedni priorytet. Flaga `PCATCH` → dostarczenie sygnału wybudza od razu (`EINTR`, `ERESTART`).

Z uśpionym wątkiem można skojarzyć komunikat i termin pobudki.

```
ps -a -o uid,pid,stat,wchan:20,command
```

FreeBSD: Wybudzanie wątków

```
void wakeup(void *chan);
void wakeup_one(void *chan);
```

Wybudzanie wszystkich wątków w podanym punkcie oczekiwania (ang. *broadcast*) albo jednego… ale według jakich kryteriów?

Wątki są przechowywanie w kolejce FIFO → najdłużej oczekujący?
Nie! Wątki mają priorytety → wybudzamy ten o najwyższym.

Opisane w dokumentacji do `sleepqueue(9)`.

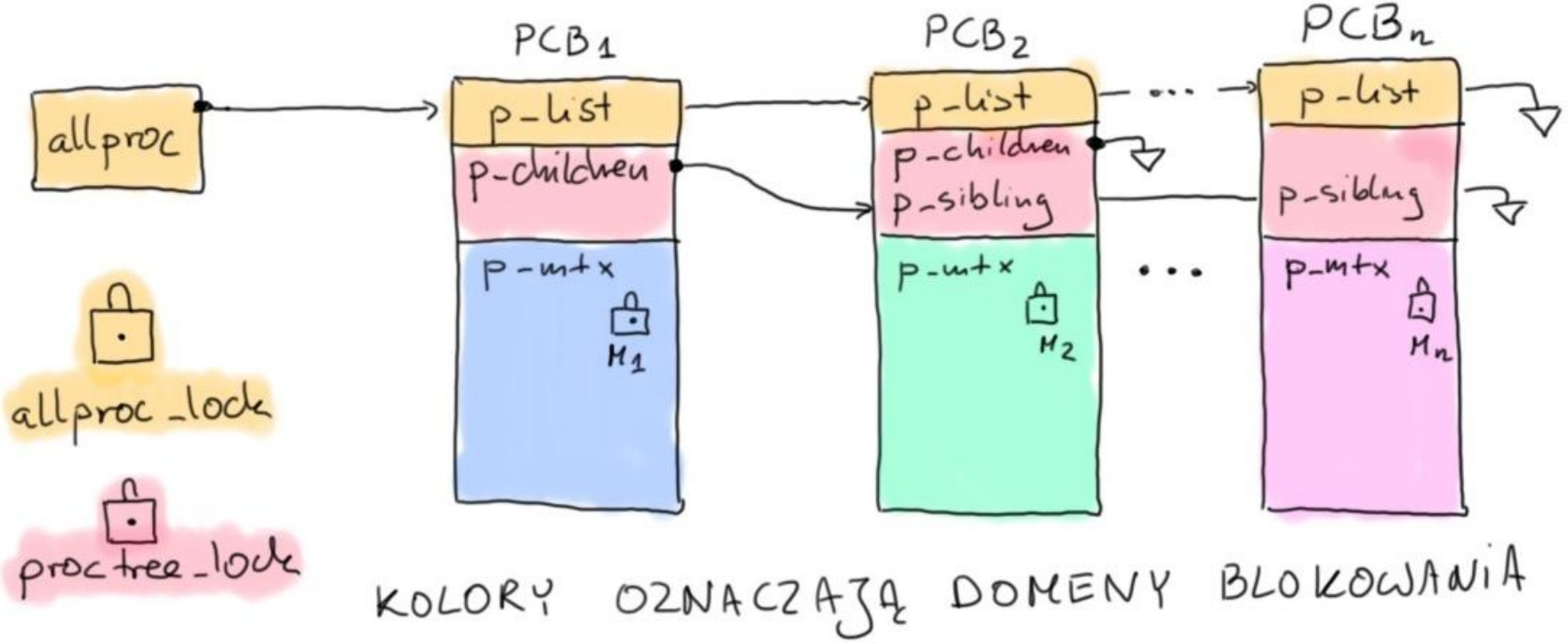
FreeBSD: Listy wątku a środki synchronizacji

Związane z poszczególnymi bazowymi środkami synchronizacji.

```
// [...] The lock is indicated by a reference to a specific character
// in parens in the associated comment. [...]
//  ...
//  (t) thread::td_lock
//  (q) td_contested_lock (defined in kern/subr_turnstile.c)

struct thread {
  struct mtx *volatile td_lock;         // replaces scheduler lock 🔥
  ...
  TAILQ_ENTRY(thread) td_runq;          // (t) Run queue (scheduler)
  TAILQ_ENTRY(thread) td_slpq;          // (t) Sleep queue (sleepqueue)
  TAILQ_ENTRY(thread) td_lockq;         // (t) Lock queue (turnstile)
  LIST_HEAD(, turnstile) td_contested; // (q) Contested locks (turnstile)
  ...
};
```

Domeny blokowania



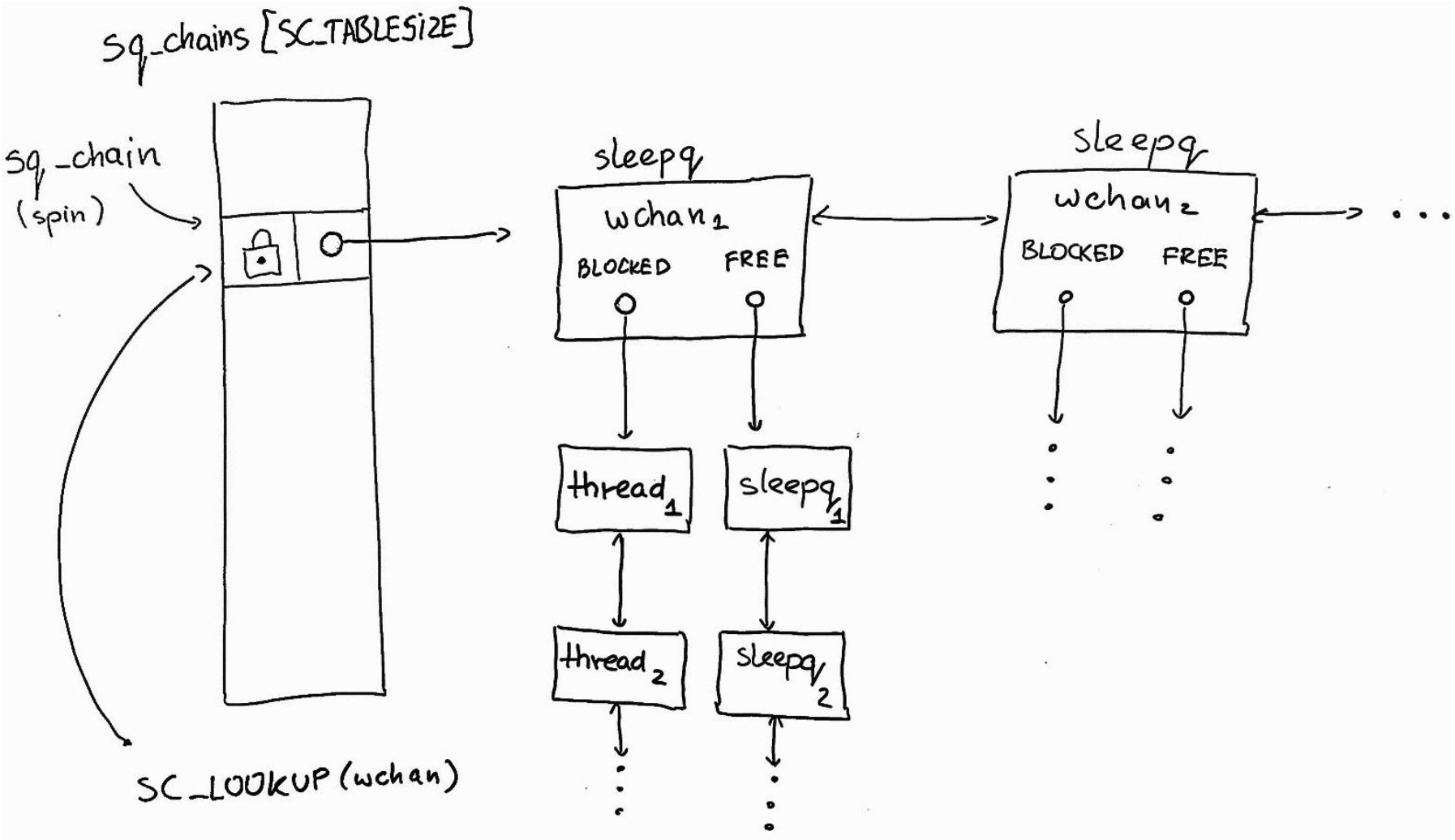
Blokady mogą chronić fragmenty pojedynczej instancji rekordu, jak w przypadku `proc::p_mtx`, ale mogą pokrywać fragmenty wszystkich instancji rekordu danego typu jak w przypadku listy wszystkich procesów i struktury drzewa procesów.

FreeBSD: Kolejki wątków uśpionych

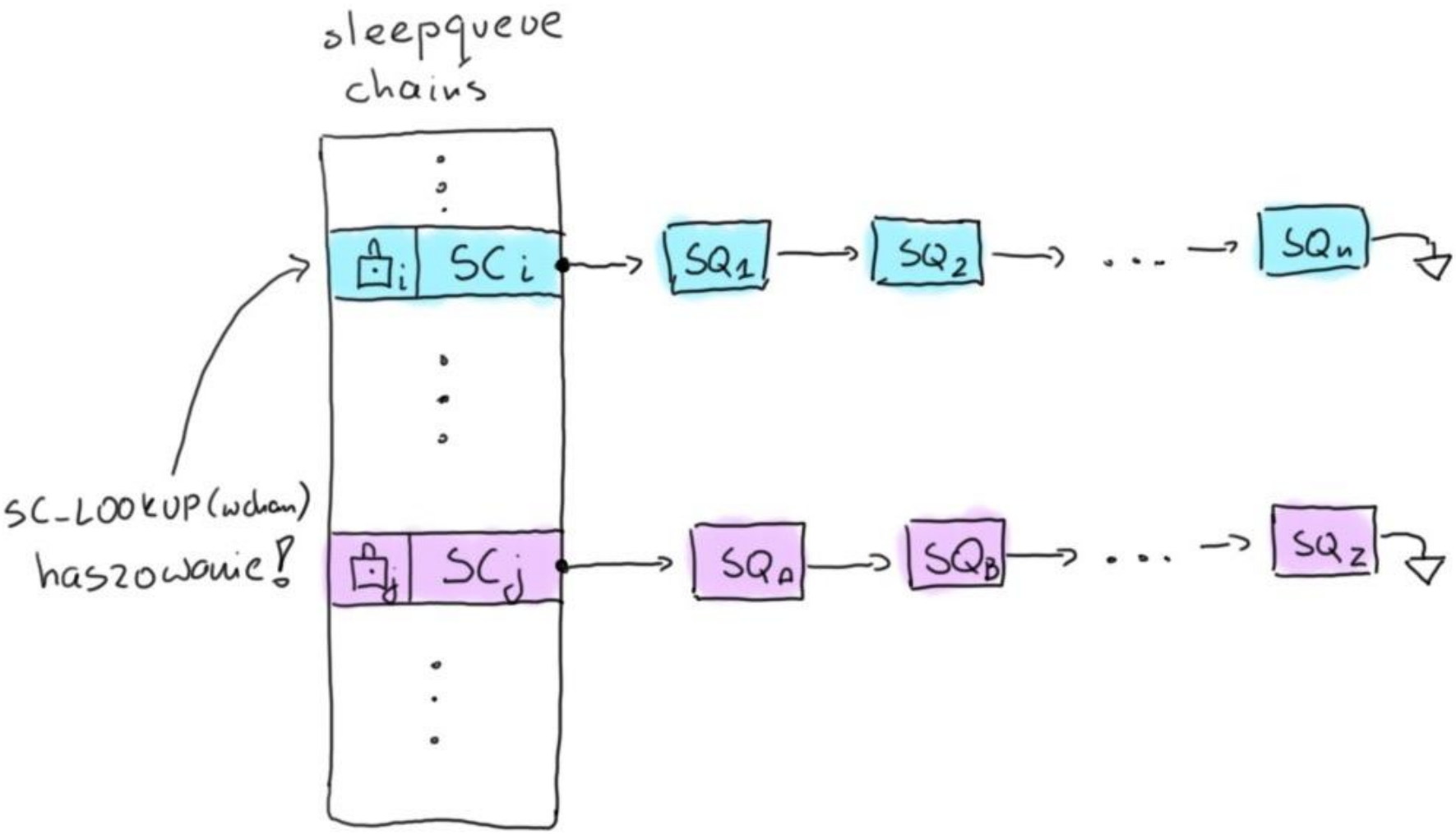
Każdy wątek ma swoją `sleepqueue` podpiętą pod `td_slpq`.
Wątek może spać co najwyżej na jednym **`wchan`** i w tym celu musi oddać własne **`sleepqueue`**. Zatem w systemie nie potrzeba więcej **`sleepqueue`** niż jest wątków jądra!

```
struct sleepqueue {
  TAILQ_HEAD(, thread) sq_blocked[2];
  u_int sq_blockedcnt[2];
  LIST_ENTRY(sleepqueue) sq_hash;
  LIST_HEAD(, sleepqueue) sq_free;
  void *sq_wchan;
  int sq_type;
};
```

FreeBSD: Łańcuchy kolejek wątków uśpionych



FreeBSD: Blokady łańcuchów `sleepqueue`



FreeBSD: Kolejki wątków uśpionych (c.d.)

Pierwszy uśpiony wątek użycza swojego rekordu **`sleepqueue`**, jako głowy kolejki wątków uśpionych **`sq_blocked`**, a następnie wkłada go do odpowiedniego łańcucha `sleepqueue_chain`.

Kolejne uśpione wątki są dodawane na **`sq_blocked`**,
ale najpierw odkładają swoje **`sleepqueue`** na **`sq_free`**.

Czemu dwie **`sq_blocked`**? Implementacja `sx(9)` z tego korzysta:

```
#define SQ_EXCLUSIVE_QUEUE 0
#define SQ_SHARED_QUEUE 1
```

Przy wybudzaniu, jeśli więcej niż jeden wątek na **`sq_blocked`**
to zdejmujemy z **`sq_free`**, w p.p. zdejmujmy z łańcucha.

sleepqueue(9): przydział i blokowanie łańcuchów

Tworzymy dokładnie tyle struktur **sleepqueue** ile jest wątków w systemie – więcej nie potrzebujemy!

```
struct sleepqueue *sleepq_alloc(void);

void sleepq_free(struct sleepqueue *sq);
```

Żeby móc odnaleźć **sleepqueue** w łańcuchu odpowiadającemu danemu wchan musimy wziąć blokadę **sc_lock**. Większość procedur API wymaga uprzedniego wywołania **sleepq_lock**!

```
void sleepq_lock(void *wchan);

void sleepq_release(void *wchan);

struct sleepqueue *sleepq_lookup(void *wchan);
```

sleepqueue(9): przygotowanie do snu

UWAGA! Dodanie do kolejki przygotowuje do snu, ale nie usypia!

Poniższe flagi używane do sprawdzania, czy usypianie
i wybudzanie przeprowadzone przez ten sam środek.

```
void sleepq_add(void *wchan, struct lock_object *lock,
                const char *wmesg, int flags, int queue);
```

```
#define SLEEPQ_SLEEP          0x00   /* Used by sleep/wakeup. */
#define SLEEPQ_CONDVAR        0x01   /* Used for a cv. */
#define SLEEPQ_PAUSE          0x02   /* Used by pause. */
#define SLEEPQ_SX             0x03   /* Used by an sx lock. */
#define SLEEPQ_LK             0x04   /* Used by a lockmgr. */
#define SLEEPQ_INTERRUPTIBLE  0x100  /* Sleep is interruptible. */
```

Można wprowadzić wątek w sen przerywalny!

sleepqueue(9): wprowadzenie w stan uśpienia

Dla snu ograniczonego czasowo najpierw trzeba ustalić termin!

```
void sleepq_set_timeout(void *wchan, int timo);
```

Wątek przygotowany do snu można uśpić na kilka sposobów.
W każdym przypadku po przebudzeniu dostajemy priorytet.

```
void sleepq_wait(void *wchan, int pri);

int sleepq_wait_sig(void *wchan, int pri);

int sleepq_timedwait(void *wchan, int pri);

int sleepq_timedwait_sig(void *wchan, int pri);
```

Wybudzanie z użyciem callout(9) na **td_slpcallout**.

sleepqueue(9): Wybudzanie

Możemy wybudzić jeden o najwyższym priorytecie lub wszystkie.
Po wybudzeniu podwyższenie priorytetu do **`pri`** jeśli niższy.

```
int sleepq_broadcast(void *wchan, int flags, int pri, int queue);

int sleepq_signal(void *wchan, int flags, int pri, int queue);
```

Jeśli wątek jest w śnie przerywalnym to możemy po prostu użyć:

```
int sleepq_abort(struct thread *td);
```

FreeBSD: implementacja `tsleep` i blokady

`_sleep`:
→ `sleepq_lock` — blokada na łańcuch `sc_lock`
→ `sleep_add` — oddaj swoje `sleepqueue`
→ `sleepq_set_timeout` — ustaw termin wybudzenia (`callout`)
→ `sleepq_wait` — uśpij jeśli nie oddano mi `sleepqueue`
  → `sleepq_switch` — wykonywane z założonym `td_lock`
    → `thread_lock_set`[1] — przepina `td_lock` na `sc_lock`
    → `mi_switch` — więcej magii z przepinaniem `td_lock` 😱

[1] Wykonuje sekwencję: `oldmtx←td_lock`, `td_lock←newmtx`, `unlock(oldmtx)`.

No dobra… ale o co chodzi z `td_lock`, `sc_lock`, itd.?

Czemu **`td_lock`** jest wskaźnikiem?

Dyspozytor wieloprocesorowy (składowe: turnstiles, sleepqueues, planista wątków, itp.) synchronizuje dostęp do wszystkich instancji struktur `thread`, a dokładniej pól oznaczonych `(t)`. Grupuje wątki według stanu i każdej grupie przypisuje pewien spin-mutex.

`td_lock` to spin-mutex → zatem jakie ma właściwości?

Think about `td_lock` like something what is lent by current thread owner. If a thread is running, it's owned by scheduler and `td_lock` points to scheduler lock. If a thread is sleeping, it's owned by sleeping queue and `td_lock` points to sleep queue lock. If a thread is contested, it's owned by turnstile queue and `td_lock` points to turnstile queue lock. [...] The `td_lock` pointer is changed atomically, so it's safe. – *Svatopluk Kraus*

Why do we need to acquire the current thread's lock before context switching?

Odczarowanie `mi_switch` w `sleepq_switch`

Struktur starego planisty zadań (`4BSD`) broni jeden `sched_lock`. Mamy jeszcze spin-mutex, który zawsze jest zablokowany → `blocked_mutex`.

Gdy wchodzimy do `mi_switch` wiemy, że `td->td_lock=sc_lock`, a `sc_lock` jest wzięty.

`sched_switch` zakłada `sched_lock`, a następnie atomowo zwalnia `td->td_lock` (ale trzyma na niego wskaźnik w `tmtx`) i przepina `td->td_lock` na `blocked_lock`. Następnie wybiera wątek przy pomocy `sched_choose`, który był na `runq`, więc `newtd->td_lock=sched_lock`. `cpu_switch` przełącza kontekst z `td` na `newtd`, a w trakcie przepina `td->td_lock` na `tmtx` (zastępując `blocked_lock`).

Pod koniec `sched_switch` zwalniamy `sched_lock`, ale najpierw poprawiamy właściciela blokady, żeby nie wysypały się asercje!

FreeBSD: implementacja wakeup i blokady

wakeup:

→ sleepq_broadcast — wybudza wątki na danym wchan

→ sleepq_lookup — bierze blokadę sc_lock

→ sleepq_remove_matching — wybiera wątki do wybudzenia

→ sleepq_resume_thread [1] — wybudza wybrany wątek

→ sleepq_remove_thread [2] — zwraca sleepqueue wątkowi

→ setrunnable → sched_wakeup → sched_add [3]

→ runq_add — dodaje do runq i zwalnia td_lock

[1] Zakłada td_lock jeśli td nie spał, w p.p. td_lock=sc_lock. Po wykonaniu sleepq_remove_thread zwalnia sc_lock.

[2] Zakłada, że td_lock i sc_lock są wzięte. Anuluje callout, jeśli był ustawiony.

[3] Jeśli td_lock≠sched_lock to [unlock(td_lock), td_lock←sched_lock].

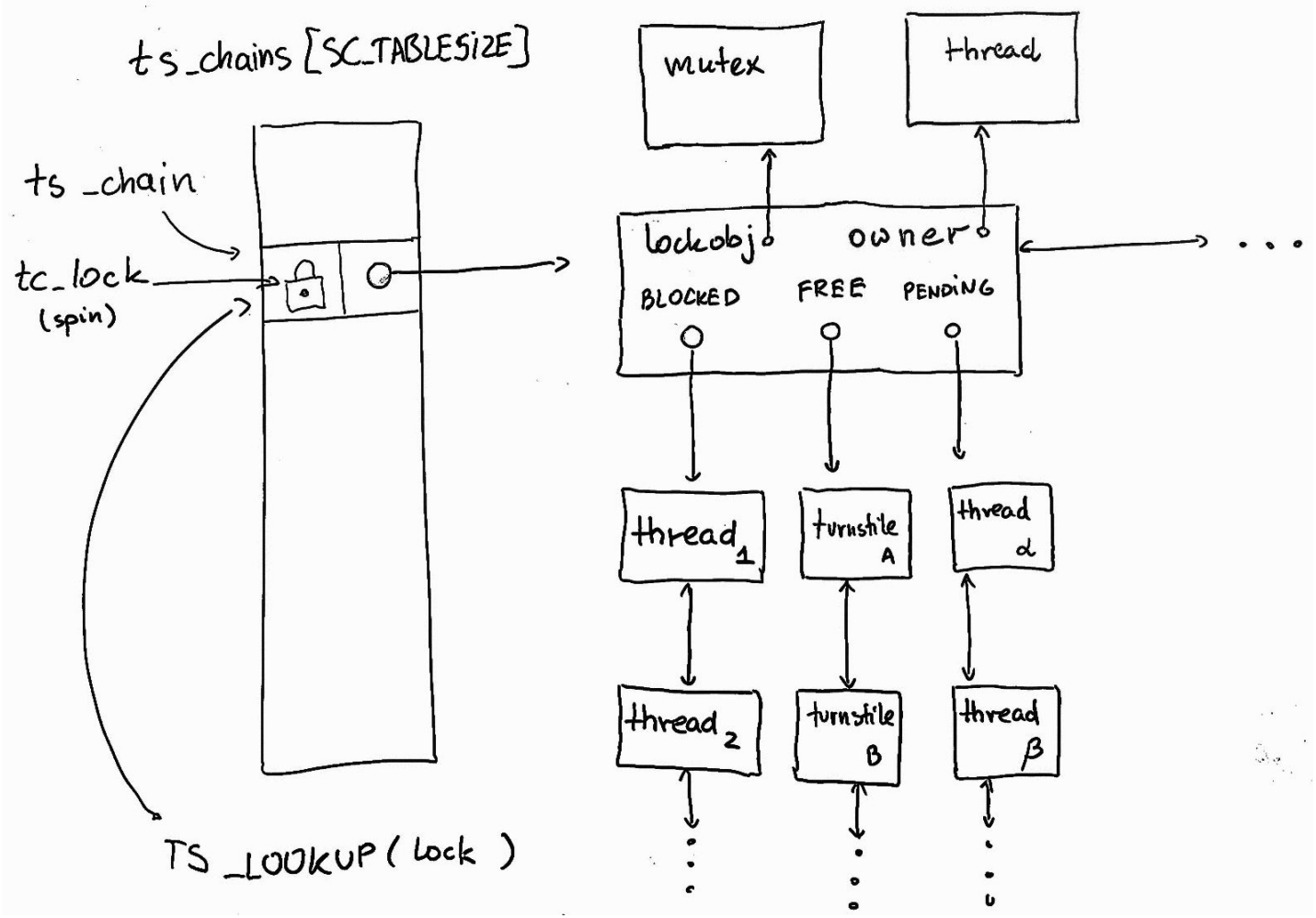
Wprowadzanie w sen ograniczony

FreeBSD: bramki obrotowe aka rogatki

Podobne do **`sleepqueue`**, ale wspierają (łańcuchową!) propagację priorytetów. Do procedur **`turnstile`** jako `wchan` podajemy wskaźnik na strukturę blokady, czyli **`lock_object`**.

```
struct turnstile {
  struct mtx ts_lock;                    // Spin lock for self
  TAILQ_HEAD(, thread) ts_blocked[2];    // (c+q) Blocked threads
  TAILQ_HEAD(, thread) ts_pending;       // (c) Pending threads
  LIST_ENTRY(turnstile) ts_hash;         // (c) Chain and free list
  LIST_ENTRY(turnstile) ts_link;         // (q) Contested locks
  LIST_HEAD(, turnstile) ts_free;        // (c) Free turnstiles
  struct lock_object *ts_lockobj;        // (c) Lock we reference
  struct thread *ts_owner;               // (c+q) Who owns the lock
};
```

Rogatki: struktura danych



Rogatki: komentarz (1)

Dwie kolejki **`blocked`** → read / shared lock, write / exclusive lock. Pozwalają na implementację blokad typu **`mutex`** i **`rwlock`**.

Pierwszy wątek, który musi zatrzymać się na blokadzie, oddaje swoje **`turnstile`**. Rogatka pamięta blokadę i jej właściciela, dzięki czemu można wypożyczyć mu priorytet **`sched_lend_prio`**.

Odblokowane wątki z użyciem **`turnstile_signal`** (najwyższy priorytet) lub **`turnstile_broadcast`** przenosimy do **`ts_pending`**, ale jeszcze nie wybudzamy! Oddajemy im nieaktywne **`turnstile`** z listy **`ts_free`**, ale mogą dostać inne rogatki niż wypożyczyły. Ostatni odblokowany wątek dostaje rogatkę z łańcucha rogatek.

Rogatki: komentarz (2)

W **turnstile_unpend** zanim wybudzimy wątki **sched_add** musimy przywrócić im oryginalny priorytet **sched_unlend_prio**.

Właściciel blokad pamięta rogatki tego podzbioru swoich blokad, o które istnieje współzawodnictwo. Wszystkie listy **td_contested** chronione jedną blokadą wirującą **td_contested_lock**.

Przed skorzystaniem z łańcucha rogatek, należy założyć blokadę **tc_lock** (**turnstile_chain_lock**). W trakcie wyszukiwania rogatki (**turnstile_lookup**) zostanie założona blokada **ts_lock**.

FreeBSD: wybudzanie zablokowanych na muteksie

`__mtx_unlock_sleep`:

→ `turnstile_chain_lock` zakłada blokadę na łańcuch

→ `turnstile_lookup` wyszukuje rogatki na łańcuchu

→ `turnstile_broadcast` przenosi wątki na listę `pending`

→ `turnstile_unpend`[1] uruchamia wątki z `pending`

→ `turnstile_chain_unlock` zwalnia blokadę na łańcuch

[1] Wybudzamy wszystkie wątki w kolejności od najwyższego do najniższego priorytetu. Oczekujemy, że planista zleci je w tej samej kolejności. W przypadku równoległego wykonania wirowanie adaptacyjne powinno zapobiec ponownemu zablokowaniu.

FreeBSD: blokowanie na muteksie

`__mtx_lock_sleep`:

→ `turnstile_trywait` sprawdza czy ktoś już wisi na `turnstile`

→ `turnstile_cancel`[1] zwalnia blokadę `turnstile` i łańcuch

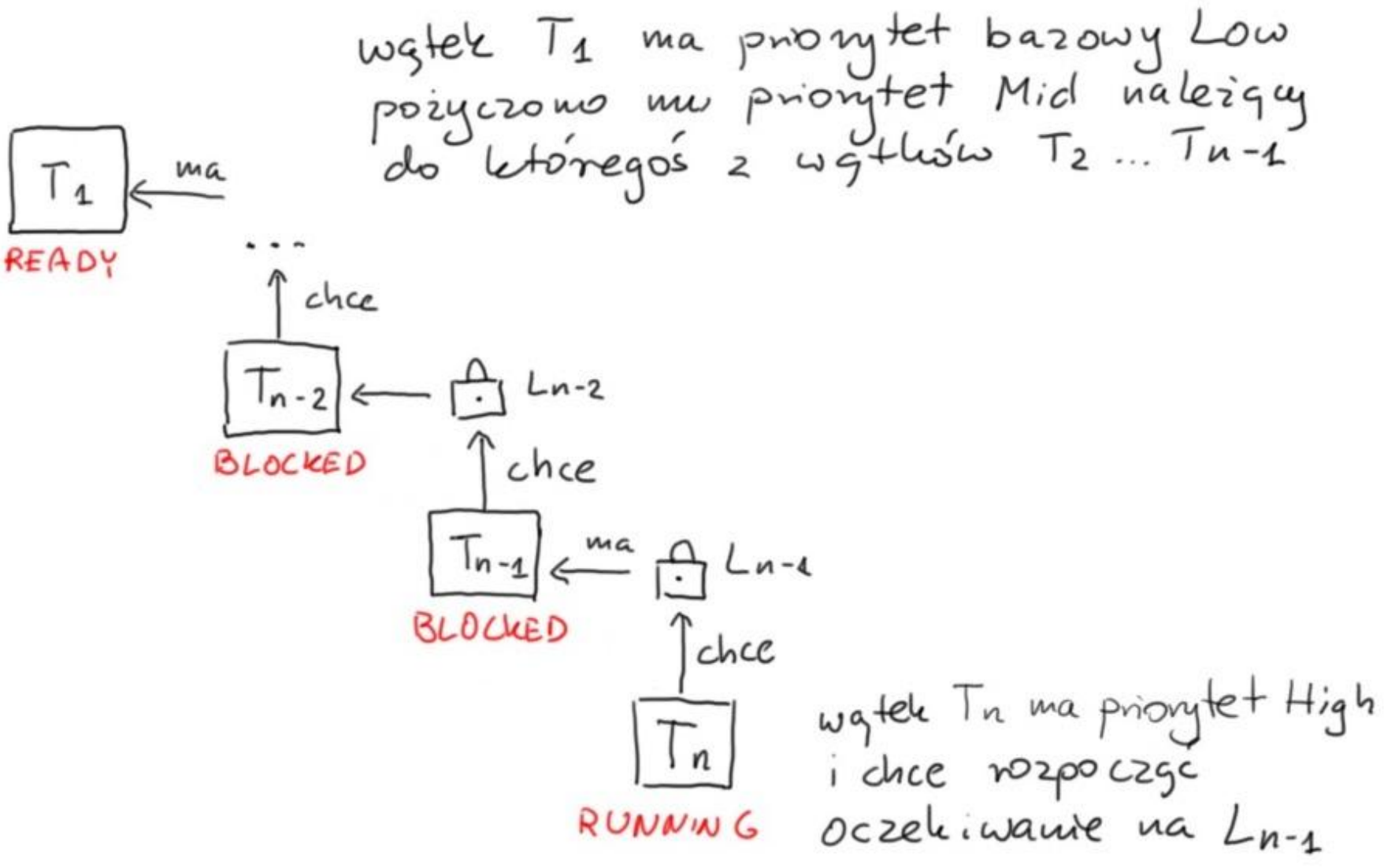
→ `turnstile_wait`[2]

→ `propagate_priority` pożycza priorytet wątkom czekającym

[1] Na wypadek, jeśli w międzyczasie jednak ktoś zwolnił blokadę.

[2] Blokuje na `turnstile` i zmienia kontekst trzymając blokady wirujące. Po drodze zwalnia blokadę łańcucha `tc_lock`. Przed `mi_switch` przepina blokadę `td_lock` na `ts_lock`. Po zmianie kontekstu `ts_lock` zostanie zwolnione.

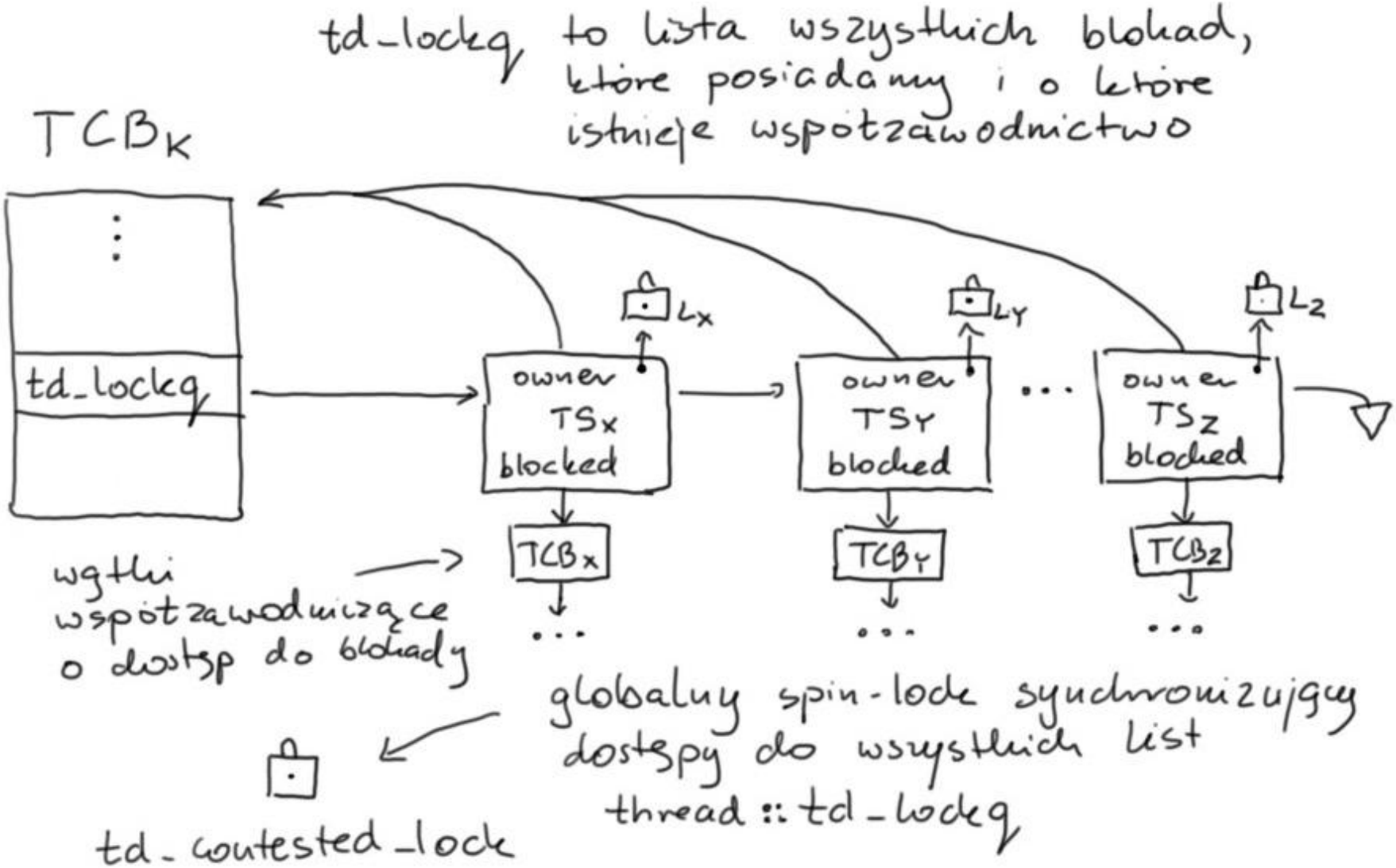
Łańcuchowa propagacja priorytetów



Mimiker: implementacja `propagate_priority`

1. Startujemy od rogatki $\texttt{ts}^1$ blokady na której chce zaczekać **`td`**. Właścicielem tej rogatki $\texttt{ts}^1$ jest $\texttt{td}^1$.
2. Dopóki $\texttt{td}^i$ jest zablokowany i ma mniejszy priorytet od **`td`**, to:
   a. weź rogatkę $\texttt{ts}^{i+1}$ blokady, na którą czeka $\texttt{td}^i$
   b. pożycz $\texttt{td}^i$ priorytet **`td`**
   c. $\texttt{td}^i$ jest na liście $\texttt{ts}^{i+1}\texttt{::ts\_blocked}$, zatem trzeba naprawić niezmiennik “`ts_blocked` jest posortowana względem malejących priorytetów”
   d. weź właściciela $\texttt{td}^{i+1}$ rogatki $\texttt{ts}^{i+1}$ i przejdź do następnej iteracji
3. Na końcu łańcucha możemy mamy wątek o priorytecie niższym od naszego. Wątek ten został wywłaszczony, więc jemu też pożyczamy, żeby nie wywłaszczył go nikt, kto ma priorytet niższy od naszego.

Zwracanie (odpożyczanie?) priorytetu



Mimiker: odpożyczanie priorytetu `unlend_self`

Odpożyczanie trzeba przeprowadzić przy zwalnianiu blokady,
o którą istniało współzawodnictwo.

1. Wypinamy **`ts`** z **`td_lockq`** naszego wątku **`td`**.
   (jeśli dalej istnieje o współzawodnictwo o blokadę skojarzoną z **`ts`**
   to włoży ją na swoją listę **`td_lockq`** pierwszy nowy właściciel blokady)
2. **`p`** przypisz najniższy priorytet (według konwencji BSD: 255)
3. Dla każdej rogatki $\mathbf{ts}^{\mathbf{i}}$ na liście $\mathbf{td}\rightarrow\mathbf{td\_lockq}$:
   a. **`td`** musi być jej właścicielem $\mathbf{ts}^{\mathbf{i}}$
   b. jeśli pierwszy (czyli o najwyższym priorytecie) na liście
      $\mathbf{ts}^{\mathbf{i}}\rightarrow\mathbf{ts\_blocked}$ ma priorytet wyższy od **`p`**,
      to weź jego priorytet jako nowe **`p`**
4. Powiedz planiście zadań, że **`td`** żąda obniżenia priorytetu do **`p`**.

Implementacja środków synchronizacji

FreeBSD: obiekty blokad

Każda blokada ma wspólny nagłówek **`lock_object`**:

```
struct lock_object {
  const char *lo_name;         // Individual lock name
  u_int lo_flags;
  u_int lo_data;               // General class specific data
  struct witness *lo_witness; // Data for witness
};
```

Flagi stwierdzają do jakiej klasy blokad należy dany obiekt:

```
#define LC_SLEEPLOCK  0x01 // Sleep lock
#define LC_SPINLOCK   0x02 // Spin lock
#define LC_SLEEPABLE  0x04 // Sleeping allowed with this lock
#define LC_RECURSABLE 0x08 // Locks of this type may recurse
#define LC_UPGRADABLE 0x10 // Upgrades and downgrades permitted
```

FreeBSD: klasy blokad

Wszystkie blokady mają wspólny interfejs dzięki, któremu można ich używać np. ze zmiennymi warunkowymi.

```
struct lock_class {
  const char *lc_name;
  u_int       lc_flags;
  ...
  void (*lc_assert)(const struct lock_object *lock, int what);
  void (*lc_lock)(struct lock_object *lock, uintptr_t how);
  int  (*lc_owner)(const struct lock_object *lock,
                   struct thread **owner);
  uintptr_t (*lc_unlock)(struct lock_object *lock);
};
```

FreeBSD: struktura muteksa

Właściwe dla implementacji muteksa jest tylko pole **`mtx_lock`**.

```
struct mtx {
  struct lock_object  lock_object; // Common lock properties
  volatile uintptr_t  mtx_lock;    // Owner and flags
};
```

Przechowuje wskaźnik na właściciela oraz flagi:

```
#define MTX_UNOWNED   0   // Cookie for free mutex
#define MTX_RECURSED  1   // lock recursed (for MTX_DEF only)
#define MTX_CONTESTED 2   // lock contested (for MTX_DEF only)
#define MTX_DESTROYED 4   // lock destroyed
```

Na polu **`mtx_lock`** wyłącznie operacje atomowe!

Muteksy adaptacyjne

Obserwacja: W systemie SMP oczekiwany czas na wejście pod blokadę wirującą jest zerowy (brak rywalizacji) lub mniejszy niż wykonanie przełączenie kontekstu (przy rywalizacji).

Najczęstszy przypadek użycia? Bardzo krótka sekcja krytyczna!

Muteks adaptacyjny. Jeśli w trakcie zakładania blokady rywalizacja → chwilę się kręcimy (`lock_delay`) optymistycznie uznając, że blokada zostanie zwolniona za pewną liczbę cykli procesora. Jeśli nadal zajęta, to prosimy jądro o uśpienie wątku w oczekiwaniu na zwolnienie blokady.

`ADAPTIVE_MUTEXES` w procedurze `__mtx_lock_sleep`.

FreeBSD: zmienne warunkowe

Wysokopoziomowy interfejs nad **`sleepqueue`**.

```
struct cv {
  const char *cv_description;
  int         cv_waiters;
};
```

Oczekiwanie dodatkowo atomicznie zwalnia i zakłada blokadę, gdzie argument **`lock`** może być mutex(9), rwlock(9) albo sx(9):

```
void cv_wait(struct cv *cvp, lock);
int cv_wait_sig(struct cv *cvp, lock);
int cv_timedwait(struct cv *cvp, lock, int timo);
int cv_timedwait_sig(struct cv *cvp, lock, int timo);
```

Blokady współdzielone

Znane jako ***reader-write lock***, albo ***shared-exclusive lock***.

Synchronizacja dostępu do struktury danych → w jednej chwili #R ⩾ 0 wątków może ją czytać, albo #W ⩽ 1 wątków może ją modyfikować.

Możemy nadać priorytet czytelnikom (ang. *read-mostly RW-lock*) lub pisarzom (ang. *write-mostly RW-lock*). Kto wyczuwa głodzenie?

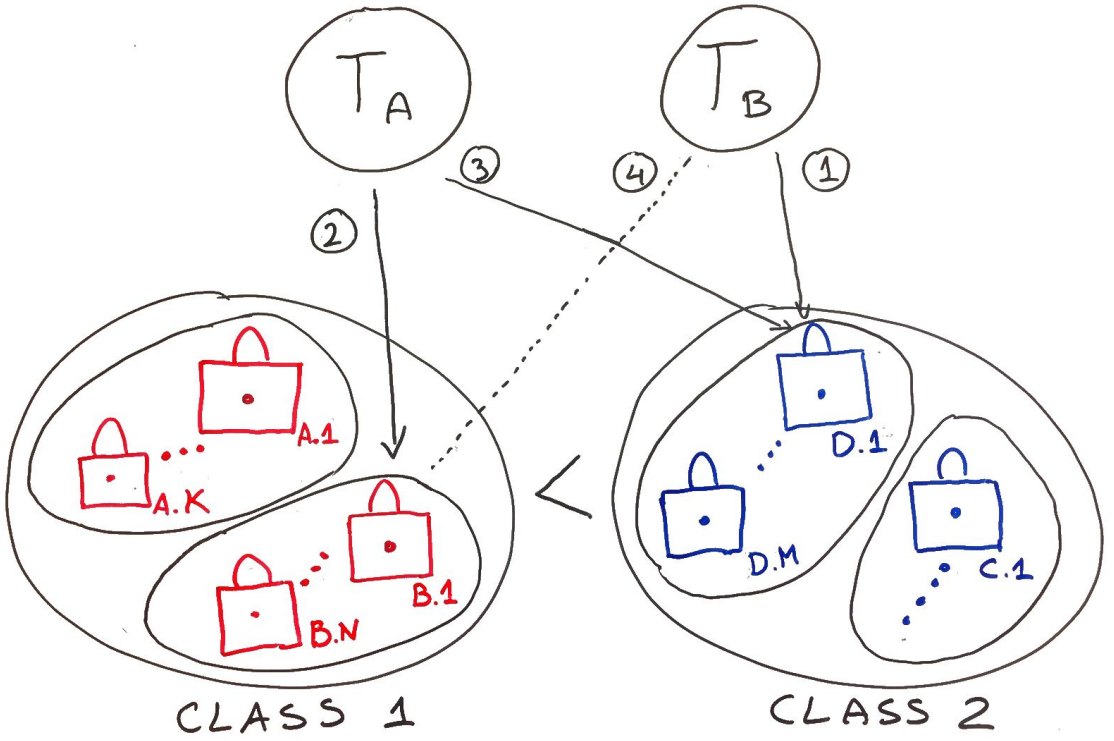
Niektóre implementacje blokad współdzielonych dopuszczają operacje:

1. zdegradowania (ang. *downgrade*) (W → R),
2. awansowania (ang. upgrade) (R → W),
3. opróżniania (ang. drain) (#R + #W → 0).

FreeBSD: `rwlock(9)`, `rmlock(9)`, `sx(9)`, `lock(9)`.

Zapobieganie zakleszczeniom

Niech kłódeczka oznacza grupę blokad z klasy o danym kolorze.



Nie możemy wziąć więcej niż jednej blokady z danej grupy. Musimy zakładać blokady w tej samej kolejności **(klasa, grupa)**.

FreeBSD: monitorowanie zakładania blokad

Moduł `witness(4)` dynamicznie konstruuje częściowy porządek na blokadach, po czym go weryfikuje.

Sprawdza również czy próbujemy wejść w głęboki sen posiadając blokadę (np. **`mutex`**), która na to nie zezwala.

Pytania?