# Struktura jądra UNIX

Wykład 9: Obsługa terminów i planista ULE

# Kolejki kalendarzowe

**(do skrócenia)**

# `callout(9)`: obsługa terminów

Chcemy zawołać procedurę o typie **`timeout_t`** po upłynięciu pewnego czasu mierzonego w taktach zegara systemowego. Intensywnie używane przez stos TCP/IP do retransmisji pakietów.

```
typedef void timeout_t (void *);
```

Z wezwaniem skojarzony jest procesor, na którym zostanie uruchomiony. Każdy procesor obsługuje przerwanie lokalnego czasomierza w `hardclockintr`. Tam wołamy `callout_process`, które używając `swi(9)` deleguje procedury wezwań do wykonania w kontekście wątku przerwania **`softclock`** na lokalnym CPU.

**Pytanie:** Czego nie mogą robić procedury wezwań?

# `callout:` struktura wezwania

```
struct callout {
  union {
    LIST_ENTRY(callout) le;      // link on cc_callwheel[i]
    SLIST_ENTRY(callout) sle;    // link on cc_callfree
    TAILQ_ENTRY(callout) tqe;    // link on cc_expireq
  } c_links;
  sbintime_t c_time;             // ticks to the event
  sbintime_t c_precision;        // delta allowed wrt opt
  void *c_arg;                   // function argument
  void (*c_func)(void *);        // function to call
  struct lock_object *c_lock;    // lock to handle
  short c_flags;                 // user state
  short c_iflags;                // internal state
  volatile int c_cpu;            // CPU we're scheduled on
};
```

Termin upłynął → wołamy **`c_func(c_arg)`** z założonym **`c_lock`**.

# Inicjowanie struktury **`callout`**

Wywołanie procedury wezwania ma być wątkowo bezpieczne. Przed / po wywołaniu należy założyć / zdjąć blokadę. Poniższe blokady gwarantują, że **`softclock`** nie wejdzie w głęboki sen.

```
void callout_init(struct callout *c, int mpsafe);

void callout_init_mtx(struct callout *c,
                      struct mtx *mtx, int flags);
void callout_init_rm(struct callout *c,
                     struct rmlock *rm, int flags);
void callout_init_rw(struct callout *c,
                     struct rwlock *rw, int flags);
```

**`CALLOUT_RETURNUNLOCKED:`** procedura wezwania zwalnia **`mtx`**
**`CALLOUT_SHAREDLOCK:`** założenie blokady dzielonej na **`rw`**

# Stan wezwania – najbardziej istotne flagi

```
#define CALLOUT_ACTIVE        2   // co is currently active
#define CALLOUT_PENDING       4   // co is waiting for timeout
#define CALLOUT_DFRMIGRATION  64  // co in deferred migration mode
#define CALLOUT_PROCESSED     128 // co in wheel / processing list?
#define CALLOUT_DIRECT        256 // can exec from hw int context?
```

```
int callout_active(struct callout *c);
int callout_pending(struct callout *c);
```

Wezwanie **oczekujące** (**`pending`**), jeśli zostało dodane do kolejki kalendarzowej, ale termin jeszcze nie upłynął.

Wezwanie **aktywne** (**`active`**) po włożeniu do kolejki kalendarzowej do momentu zatrzymania (**`stop`**), wyswobodzenia (**`drain`**), deaktywacji (**`deactivate`**). Wywołanie wezwania nie deaktywuje!

# Zlecanie wezwań

Na procesorze **`cpu`** wykonaj procedurę **`func`** z argumentem **`arg`** po upłynięciu **`ticks`** taktów zegara. Jeśli z wezwaniem skojarzona blokada to musi być uprzednio założona.

```
int callout_reset_on(struct callout *c, int ticks,
                     timeout_t *func, void *arg, int cpu);
int callout_schedule_on(struct callout *c, int ticks, int cpu);
```

Jeśli w trakcie wykonania w/w procedur wezwanie było oczekujące to anuluje poprzednie wezwanie i zwraca 1, w p.p. zwraca 0.

**Przykład:** Pakiety przychodzą miarowo, więc wystarczy przepisać termin **`callout_schedule`**. W momencie nawiązywania połączenia po raz pierwszy **`callout_reset`**.

# Zatrzymywanie i anulowanie wezwań

Oczekujące wezwanie można zatrzymać **`callout_stop`** chyba, że jest w trakcie obsługi → zwraca błąd. Jeśli chcemy poczekać na zakończenie obsługi wezwania to **`callout_drain`**. Wariant asynchroniczny zawoła procedurę po wyswobodzeniu wezwania.

```
int callout_stop(struct callout *c);
int callout_drain(struct callout *c);
int callout_async_drain(struct callout *c, timeout_t *drain);
```

Jak poprzednio należy założyć skojarzoną blokadę przed wywołaniem poniższych procedur.

# `callout:` struktura kolejki kalendarzowej

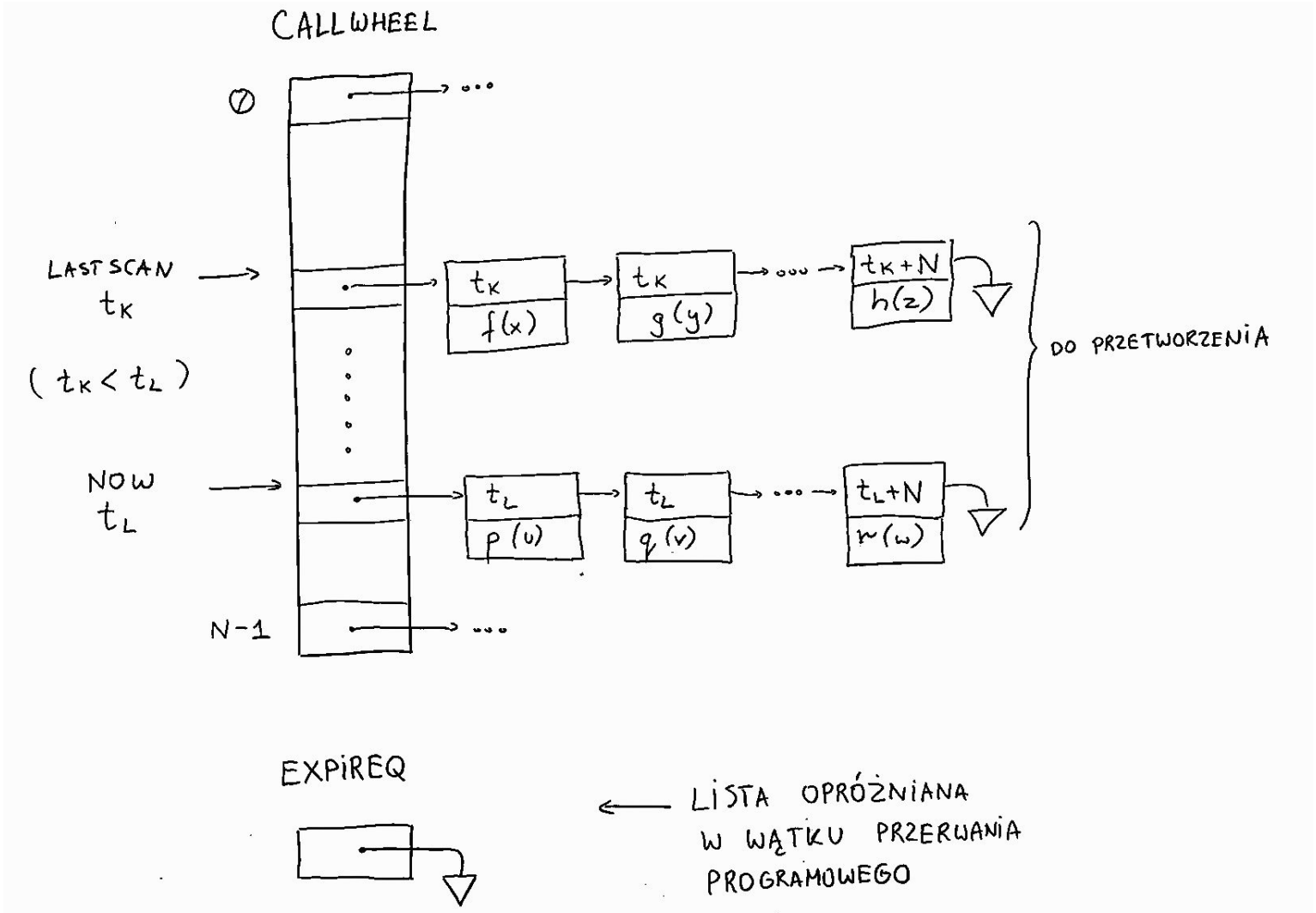
Calendar queues: a fast O(1) priority queue implementation for the simulation event set problem

```
struct callout_cpu {
  struct mtx_padalign  cc_lock;
  struct callout_list *cc_callwheel;
  struct callout_tailq cc_expireq;
  sbintime_t           cc_lastscan;
  ...
};
```

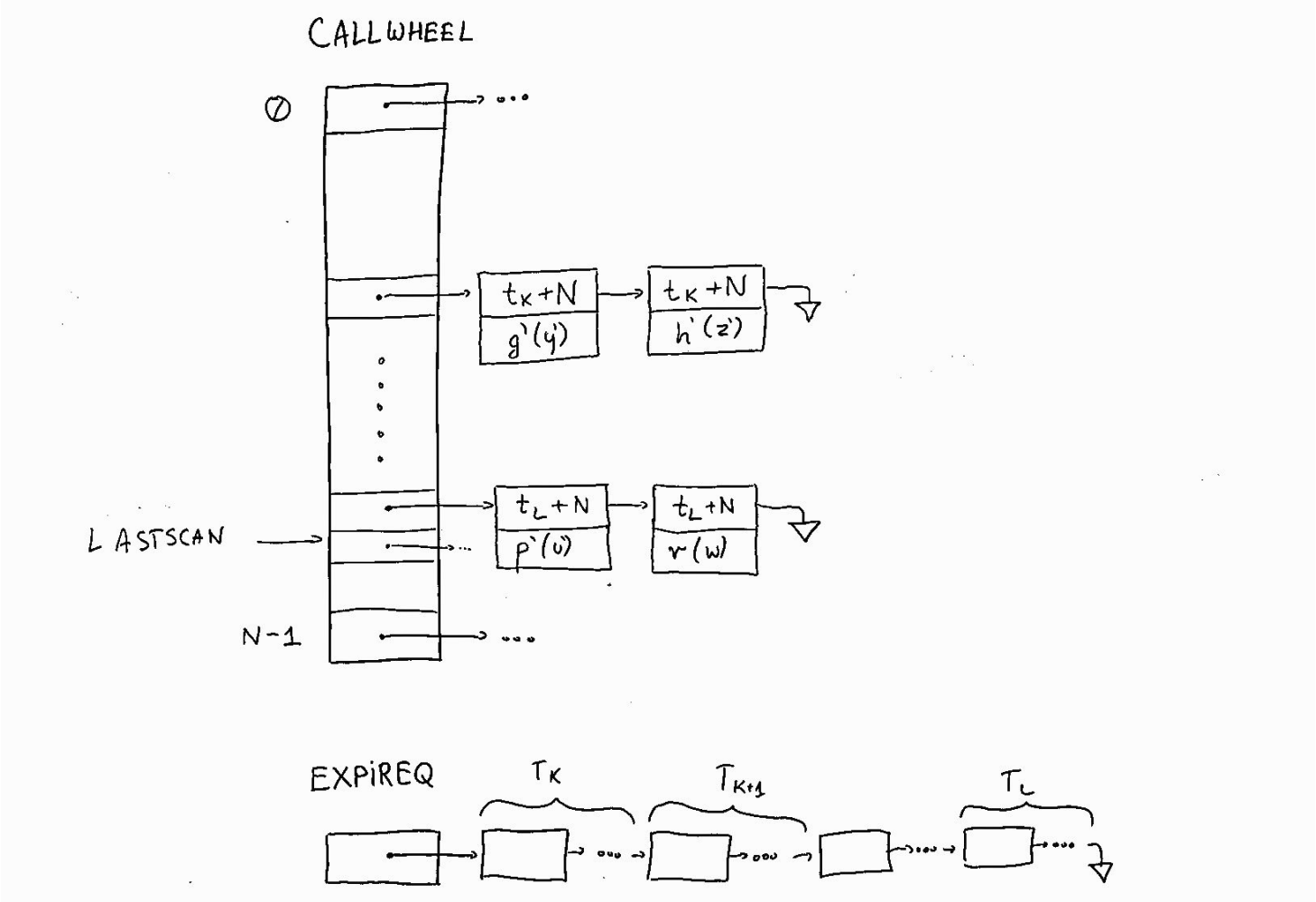
Jak działa `callout_process`

… i do czego służy struktura `callout_cpu`?

# `callout:` kolejka kalendarzowa (1)



# `callout:` kolejka kalendarzowa (2)



# Algorytmy planowania zadań

**(to trzeba rozwinąć i ukonkretnić)**

# Klasy szeregowania (ang. *scheduling class*)

W jednym systemie mogą być uruchomione zadania o różnych wymaganiach co do kryteriów szeregowania:

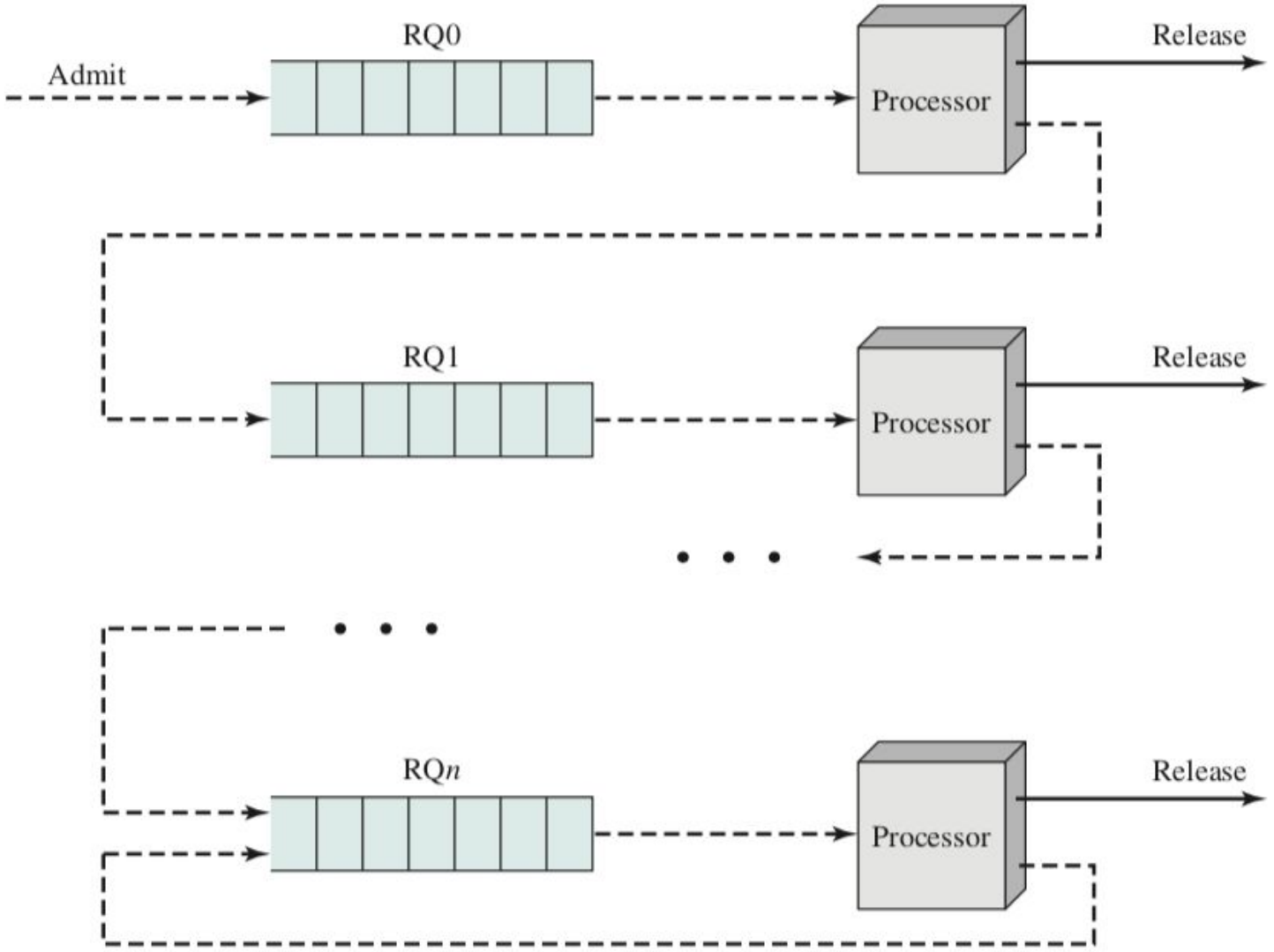
- czasu rzeczywistego (odtwarzacz filmów)
- z podziałem czasu / interaktywny (edytor tekstowy)
- wsadowe (kompilator)

Jeśli w obrębie danej **klasy** na stałe przypiszemy priorytety to istnieje możliwość zagłodzenia zadań o niższym priorytecie.

Niektóre zadania będą zmieniać swoją charakterystykę w czasie, np. edytor filmów → faza edycji interaktywnej i faza renderingu.

Systemy uniksowe dają możliwość zmiany priorytetów procesów użytkownikowi → `nice` (uprzejmy proces ma niższy priorytet).

# Multi-level feedback queue (1)



# Multi-level feedback queue (2)

Monitorujemy zachowanie zadań, **obciążenie systemu**, trzymamy historię decyzji planisty i przenosimy zadania między kolejkami.

- Jaki algorytm szeregowania przydzielić kolejce?
- Jaki kwant czasu przydzielić zadaniom w każdej z kolejek?
- Jaką heurystykę wybrać do podejmowania decyzji o podwyższeniu lub obniżeniu priorytetu?

Możemy badać następujące właściwości zadań:

- Czy wykorzystało w pełni swój kwant czasu?
- Ile czasu spędziło ostatnio / średnio wykonując instrukcje / czekając na obsługę wej.-wyj.?

# Różne idee w algorytmach szeregowania

**następny najkrótszy proces:** niech każda reakcja interaktywnego programu będzie osobnym zadaniem; trzeba dobrze oszacować czas trwania np. **średnią ważoną** $\alpha \cdot T_{prev} + (1-\alpha) \cdot T_{curr}$: **postarzanie** (ang. *ageing*)

**szeregowanie gwarantowane:** każdy proces powinien dostać $1/n$ czasu procesora; śledzimy czas wykonania zadania i obliczamy $K = T_{run} / T_{alloted}$; planista wybiera zadanie o $\min(K)$

**szeregowanie loteryjne:** każdy los oznacza otrzymanie kwantu czasu, im ważniejsze zadanie tym więcej losów dostanie; wrzucamy losy do urny, a planista losuje; losy można przekazywać potrzebującym procesom

**szeregowanie sprawiedliwe:** przypisujemy wagi do użytkowników $\sum w_i = 1$, między procesy użytkownika $i$ zostanie rozdysponowane $w_i$ czasu

# ULE: struktura kolejek (na każdy procesor osobne)

**TIMESHARE** wygląda jak kolejka kalendarzowa. Jeśli lista pusta, to w następnym takcie zegara przejdź o jedną do przodu, w p.p. należy ją opróżnić. Zatem czas oczekiwania na uruchomienie wątku uwzględnia priorytet i obciążenie systemu! Każde zadanie dostaje taki sam kwant czasu. Przy dodawaniu do kolejki, jeśli priorytet wysoki to dodaj blisko, w p.p. daleko. Ograniczony czas oczekiwania na uruchomienie → brak głodzenia!

**REALTIME**, **KERN**, **IDLE**, **ITHD** → zwykłe struktury danych runqueue, bez przenoszenia między kolejkami.

Zadania interaktywne w REALTIME!

| Range | Class | Thread type |
|---|---|---|
| 0 – 47 | ITHD | bottom-half kernel (interrupt) |
| 48 – 79 | REALTIME | real-time user |
| 80 – 119 | KERN | top-half kernel |
| 120 – 223 | TIMESHARE | time-sharing user |
| 224 – 255 | IDLE | idle user |

# ULE: stopień interaktywności (ang. *interactivity score*)

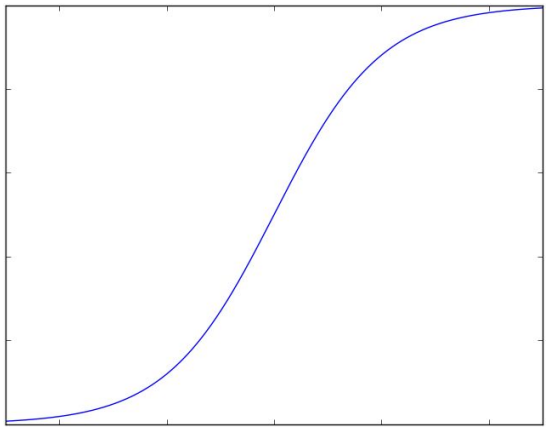
Wartość z zakresu $0...100$. Jeśli przekroczy ustalony próg, to wrzucamy do klasy z podziałem czasu, w p.p do interaktywnych.

Jeśli $T_S > T_R$ to $I = 50 \times (T_S / T_R)$,
w p.p. $I = 50 \times (1 + T_R / T_S)$.

$T_S$ i $T_R$ aktualizowane w trakcie życia wątku, skalowane tak by $I$ zawsze było w przedziale.

$I = f(T_R, T_S)$ ma przypominać sigmoid w 3D.

$I$ : interactivity score
$T_S$ : sleep time
$T_R$ : run time



**UWAGA!** Wzór na górze do poprawki!

# Szeregowanie zadań na wieloprocesorach

**Obserwacja:** wątki współdzielące przestrzeń adresową również współdzielą zasoby sprzętowe: TLB, predyktor, pamięć podręczną.

**szeregowanie według powinowactwa** (ang. *affinity scheduling*) zwraca uwagę na to by kolejny wątek do uruchomienia pochodził z tej samej przestrzeni adresowej.

**Obserwacja:** wątki często synchronizujące swoje działania powinny działać równolegle, by szybko postępować z obliczeniami.

**szeregowanie zespołowe** (ang. *gang scheduling*)
zespół współpracujących wątków szeregowane jako jednostka, ekspediowane jednocześnie do różnych procesorów, członkowie grupy zaczynają i kończą kwant czasu wspólnie.

# ULE: szeregowanie na wieloprocesorach

Planista `4BSD` przeliczał priorytety ~1 na sekundę. Brał blokadę na listę wszystkich wątków wstrzymując operacje na procesach → `fork`, `exec`, …

Planista ULE przelicza priorytety na bieżąco i każdemu wątkowi z osobna. Każdy procesor posiada własny zestaw kolejek. Izolacja stanu ogranicza potrzebę synchronizacji procesorów.

Kod zależny od sprzętu wyznacza i udostępnia **topologię architektury** tj. procesory, rdzenie, wątki, pamięć podręczna, pamięć operacyjna.

Z przeniesieniem zadania na inny procesor związany jest pewien koszt. Można go szacować uwzględniając wielkość współdzielonych pamięci podręcznych. Trzeba brać pod uwagę również odległość do pamięci wątku w systemach **NUMA** (ang. *Nonuniform Memory Access*).

# ULE: algorytmy równoważenia obciążenia

**przyciąganie zadań** (ang. *pull migration*) gdy procesor się nudzi ustawia flagę “wolny” i szuka przeciążonego procesora; jak znajdzie to kradnie zadanie o najwyższym priorytecie, w p.p. wchodzi w stan uśpienia, a wybudzić można go wysyłając `IPI` (ang. *Inter-Processor Interrupt*).

**wypychanie zadań** (ang. *push migration*) zanim przeciążony procesor doda zadanie do swojej kolejki szuka “wolnego”; jak znajdzie to dodaje do jego kolejki i go wybudza.

**Scenariusz:** 2 procesory i 3 wątki – jak każdemu dać 66% procesora?

Żeby symulować sprawiedliwy przydział czasu, system musi regularnie zmieniać przypisanie zadań do rdzeni. Długoterminowy **algorytm równoważenia obciążenia** wyrównuje obciążenie elementów topologii.

# Szeregowanie w szerszym kontekście

- ruchów głowicy dyskowej (Tanenbaum §5.4.3)
- operacji wejścia-wyjścia:
  - deadline I/O scheduler
  - anticipatory I/O scheduler
  - Complete Fairness Queuing
  - Budget Fair Queueing
- pakietów sieciowych
  - Alternate Queueing
  - PF: Packet Queueing and Prioritization

Pytania?