Systemy operacyjne

Wykład #5

- implementacja i generowanie SO
- procesy i ich atrybuty
- planista (scheduler)
- cykl życia procesu
- wątki i modele wielowątkowości
- wątki w MS Windows i w Linuksie

wersja:1.10.2021

projektowanie systemu

cel użytkownika:
system operacyjny powinien być wygodny, łatwy w użyciu, prosty do nauczenia, niezawodny, bezpieczny i szybki

cel producenta:
system operacyjny powinien być łatwy do zaprojektowania, realizacji i pielęgnowania, a także elastyczny, niezawodny, wolny od błędów i wydajny

implementacja systemu

- **dawniej** → systemy operacyjne pisane w całości w językach asemblerowych
- **OS/360 –** pierwsza próba użycia języka wysokiego poziomu (PL/I)
- **obecnie** → systemy operacyjne pisane prawie wyłącznie w językach wyższego poziomu (głównie w C lub C++)

implementacja systemu

Zalety używania języka wysokiego poziomu:

- szybkość programowania
- łatwość rozumienia, modyfikowania i sprawdzania kodu
- wysoka przenośność

implementacja systemu

Wady:

- gorsza wydajność
- większe zużycie zasobów (szczególnie pamięci)
- wąskie gardła → przepisywane na język maszynowy

generowanie systemu

- **system operacyjny** - projektowany na pewną klasę architektur i konfiguracji sprzętu
- musi być skonfigurowany dla każdej specyficznej instalacji komputerowej → generowanie systemu
- generacja systemu wymaga znajomości konfiguracji danego sprzętu
- w chwili obecnej dominuje podejście polegające na generowaniu systemu o jak największej uniwersalności kosztem rozmiaru i wydajności

rozruch systemu (bootstrap)

- **bootstrap** – procedura rozpoczęcia pracy systemu operacyjnego
- **bootstrap loader** – kod przechowywany w pamięci ROM → lokalizuje miejsce w pamięci pomocniczej, z którego ładuje kolejny program ładujący
- program ładujący może zlokalizować kolejny program ładujący (stąd nazwa *bootstrap*)
- załadowanie i uruchomienie jądra
- jądro ładuje moduły i sterowniki oraz uruchamia usługi i programy systemowe

PROCESY i WĄTKI

Proces

- w systemach wsadowych → **JOB (praca)**
- w systemach z podziałem czasu → **TASK (zadanie)**
- **proces = program + wykonanie**
- innymi słowy: proces to wykonujący się kod
- jeden program może być treścią kilku procesów
- jeden proces ma źródło w tekście jednego programu
- aplikacja może składać się z więcej niż jednego procesu

Składowe procesu

- kod procesu **(text section)**
- licznik rozkazów **(program counter)**
- stos procesu **(process stack)**
- dane procesu **(data section)**

Stan procesu

Przebywający w systemie proces może wielokrotnie zmieniać swój stan:

NOWY → w chwili tworzenia

AKTYWNY → kod procesu jest wykonywany przez CPU

CZEKAJĄCY → proces nie wykonuje się, czeka na zdarzenie

GOTOWY → proces czeka na przydział procesora

ZAKOŃCZONY → zakończył działanie

Stan procesu

W danej chwili:

liczba aktywnych procesów jest co najwyżej równa liczbie CPU

liczba procesów czekających i gotowych jest *dowolnie* duża

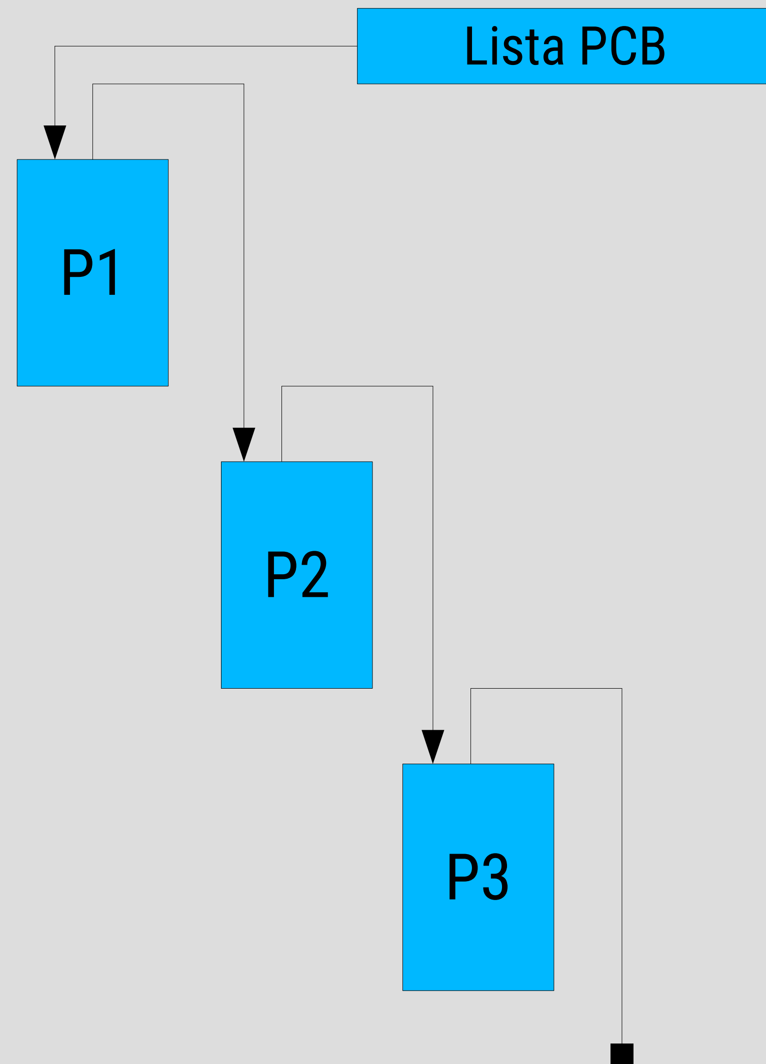
Diagram stanów procesu

Atrybuty procesu

pamiętane w bloku sterowania procesu
(*process control block* – **PCB**)

- **identyfikator** procesu (process identifier – PID)
  (unikalny w systemie) – nazwa lub numer
- **stan procesu** (process state)
- **licznik rozkazów** (program counter) – adres następnego rozkazu
- **kontekst** (program context) – rejestry procesora
- **dane planisty** – priorytet, kolejki zamówień, etc
- **dane zarządcy pamięci** – rejestry graniczne, klucze pamięci, etc
- **dane rozliczeniowe** – czas pracy, zużyte zasoby, etc
- **dane zarządcy we/wy** – przydzielone urządzenia, otwarte pliki

Hipotetyczna budowa PCB



Planowanie procesów (scheduling)

Cel: jak najefektywniejsze wykorzystanie procesora.

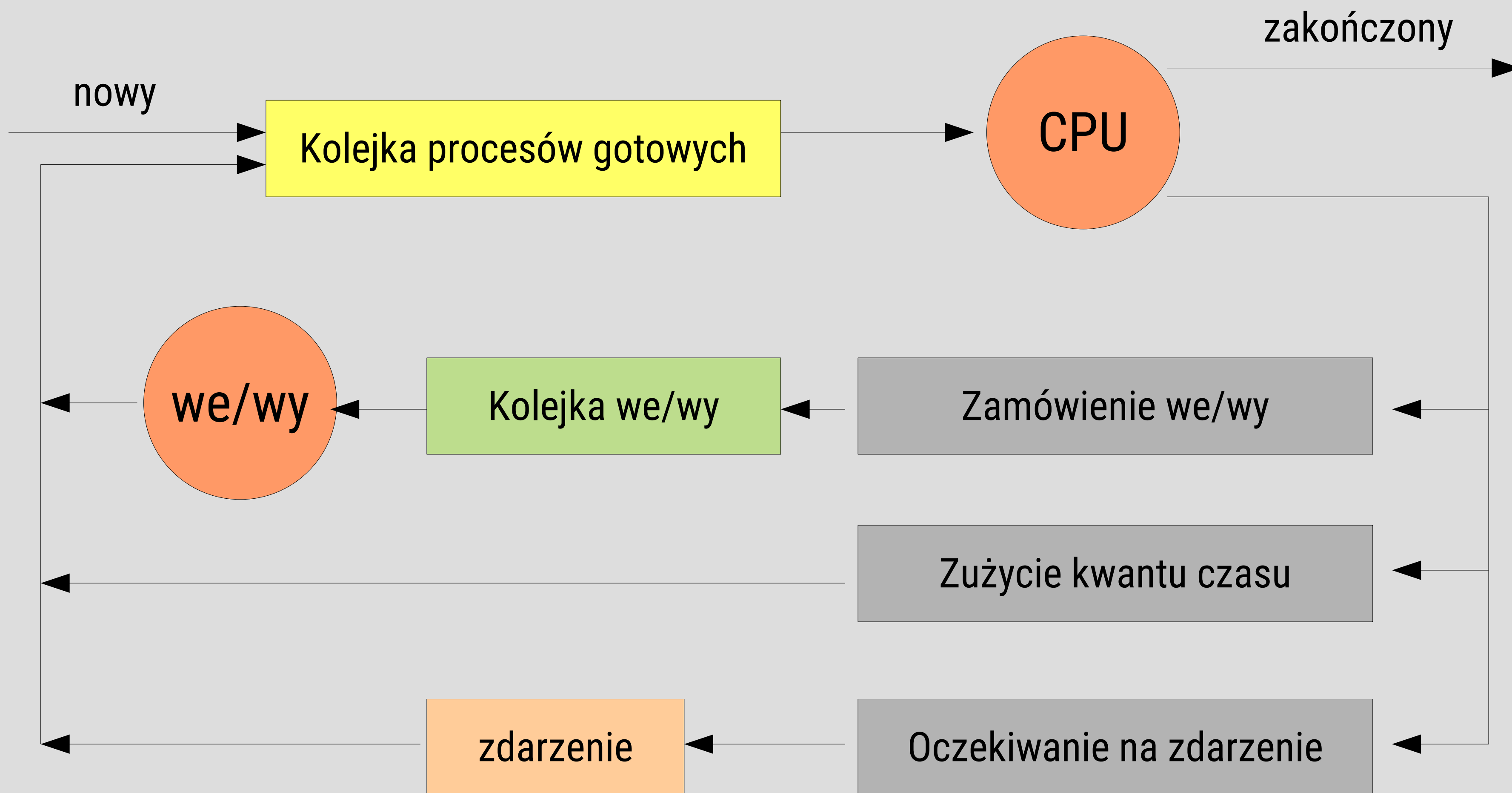
Planista (scheduler) – podejmuje decyzje co do kolejności aktywowania procesów.

Kolejki planowania (przykładowe):

- kolejka zadań (procesów)
- kolejka procesów gotowych (ready queue)
- kolejka do urządzenia (device queue)
  [każde urządzenie może mieć własną kolejkę]

PROCESY WĘDRUJĄ POMIĘDZY KOLEJKAMI W TAKT ZMIAN ICH STANU

Diagram czynności planisty



Kategorie planistów

Planista długoterminowy (long term scheduler – LTS)

- inaczej „planista zadań” (*job scheduler*)
- wybiera zadania z puli zadań (przechowywanej zewnętrznie)
- ładuje do pamięci i kolejkuje
- nadzoruje liczbę procesów w pamięci
- wywoływany rzadko – może być powolny

Planista krótkoterminowy (short term scheduler – STS)

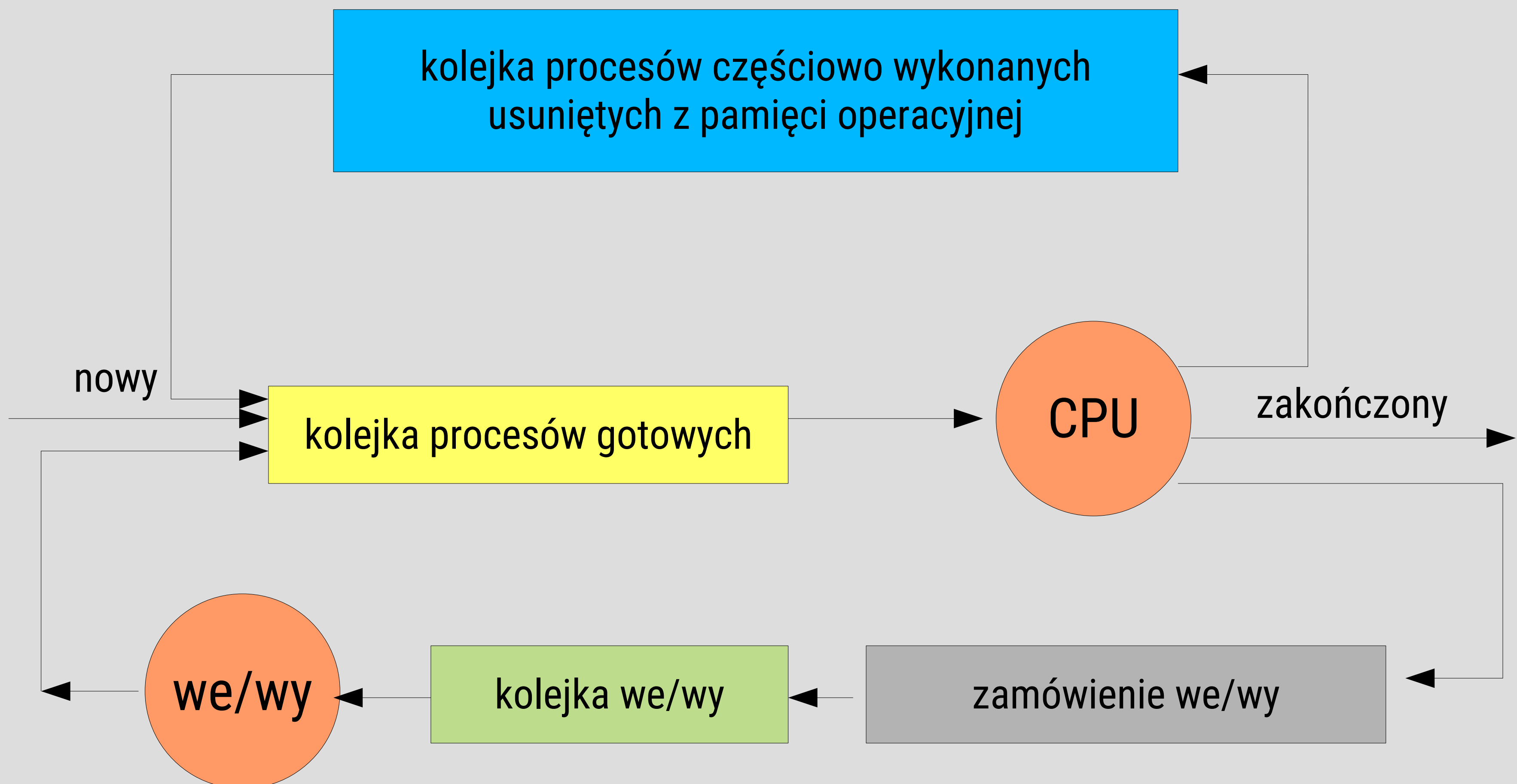
- wybiera jeden proces z kolejki procesów gotowych
- wywoływany często – musi być szybki

Dodatkowa kategoria planisty

Planista średnioterminowy (mid term scheduler – MTS)

- inaczej „*planista swapowania*” (swap scheduler)
- odpowiedzialny za wymianę procesów pomiędzy pamięcią operacyjną, a pamięcią pomocniczą

Diagram planisty średnioterminowego



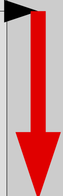
Przełączanie kontekstu

Proces P1	System operacyjny	Proces P2

przerwanie/wywołanie usługi systemu

Przechowaj kontekst P1 w PCB1

Odtwórz kontekst P2 w PCB2



przerwanie/wywołanie usługi systemu

Przechowaj kontekst P2 w PCB2

Odtwórz kontekst P1 z PCB1

Przełączanie kontekstu

- przełączenie procesora do innego procesu (ang. *context switch*) → wymaga przechowania stanu starego procesu i załadowania przechowanego stanu nowego procesu
- czas przełączania kontekstu → jeden z parametrów systemu operacyjnego (np. w RTS jest krytyczny)
- jest „daniną" na rzecz systemu – brak użytecznej pracy
- czas przełączania zależy od możliwości sprzętu (zwykle od 1 do 100 mikrosekund)
- niektóre procesory mają po kilka zbiorów rejestrów: przełączenie kontekstu = zmiana wartości wskaźnika do bieżącego zbioru rejestrów (jeden rozkaz maszynowy)
- przełączanie kontekstu może być wąskim gardłem systemu operacyjnego (ang. *bottleneck*)

Współpraca procesów

Proces niezależny (independent process)**:** proces, który nie może oddziaływać na inne procesy w systemie

Proces współpracujący (cooperating process) – proces, który może wpływać na inne procesy lub też inne procesy mogą wpływać na niego (np. przez dzielenie danych).

Współpraca procesów motywacja

- dzielenie informacji (np. plików);
- przyspieszanie obliczeń (np. podział zadania na podzadania i równoległe ich wykonywanie na wielu procesorach)
- modularyzacja (system modularny z podziałem na osobne procesy – np. GNU Hurd)
- wygoda (np. możliwość wykonywania wielu zadań równolegle)

Wątki (threads)

- tzw. lekki proces (ang. *light-weight process* – LWP)
- obecnie podstawowa jednostka wykorzystania CPU
- każdy wątek posiada odrębny:
  - licznik rozkazów
  - zbiór rejestrów
  - stos
- wspólne dla równorzędnych wątków są:
  - sekcja kodu
  - sekcja danych
  - zasoby systemowe (pliki, sygnały, etc)
- wszystkie wątki równorzędne → zadanie

Wątki (*threads*)

- tradycyjny proces → ciężki process (ang. *heavy-weight process*) → zadanie z jednym wątkiem
- wątek może przebiegać dokładnie w jednym zadaniu
- zalety wątków:

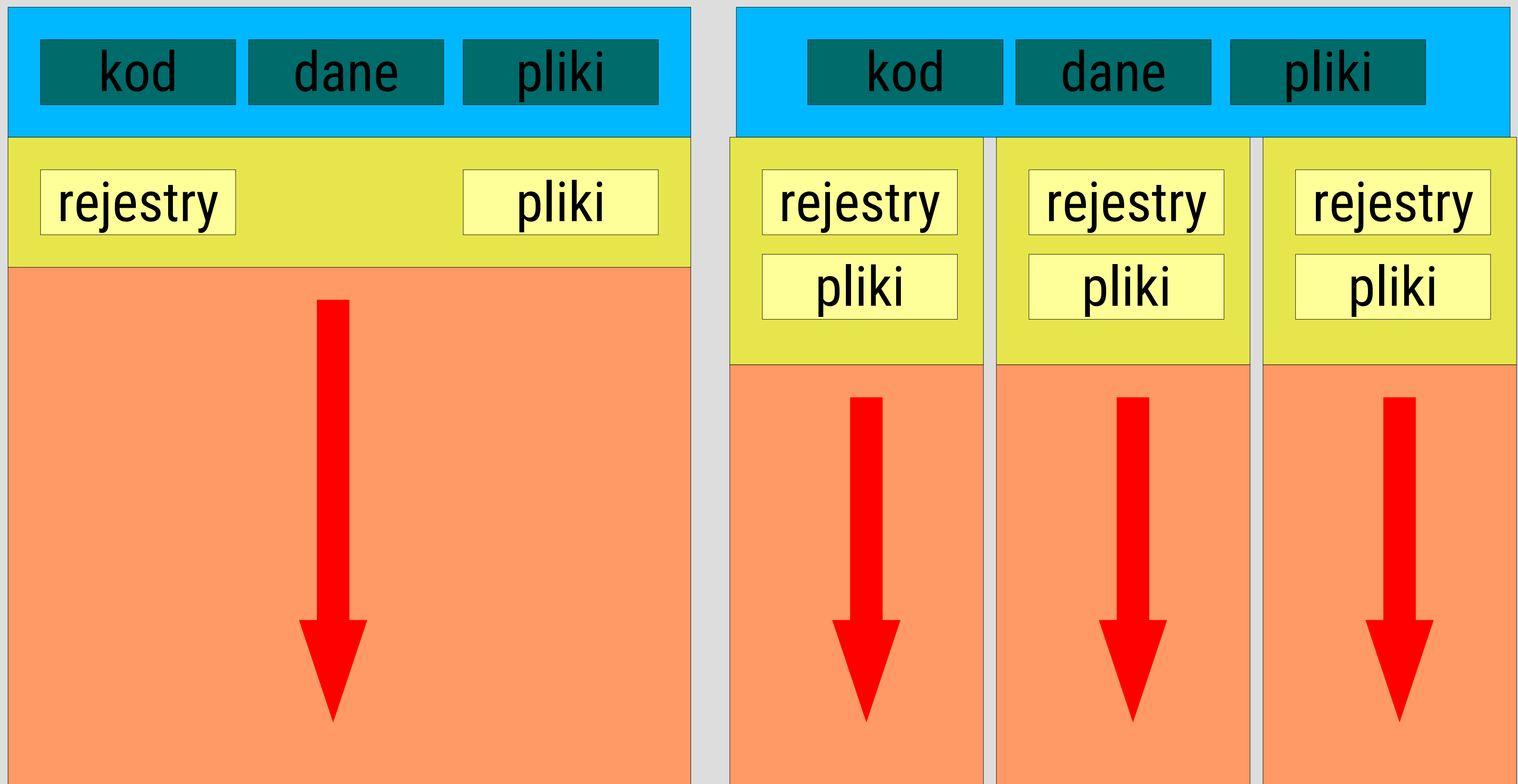
  → dzielenie zasobów

  → tańsze przełączanie między wątkami niż między procesami

  → tańsze tworzenie

  - oszczędne wykorzystanie zasobów → dzięki ich współużytkowaniu
  - współpraca wielu wątków pozwala zwiększyć przepustowość
  - poprawa wydajności → jeśli jeden wątek jest zablokowany, to może działać inny
  - efektywne wykorzystanie architektury wieloprocesorowej (trwałe związanie wątku ze wskazanym procesorem → koligacja).

Zadania jedno i wielowątkowe



Implementacja wątków

- **wątki poziomu użytkownika** (ang. *user-level threads*)
- tworzone za pomocą funkcji bibliotecznych
- przełączanie między wątkami nie wymaga wzywania systemu operacyjnego (POSIX threads = Pthreads)
- **zalety:**
  - szybkie przełączanie między wątkami
- **możliwe wady:**
  - przy jednowątkowym jądrze każde odwołanie do niego z wątku użytkownika powoduje wstrzymanie całego zadania
  - nieadekwatny przydział czasu procesora (zadanie wielowątkowe i jednowątkowe mogą dostawać tyle samo kwantów czasu)

Implementacja wątków

- **wątki jądra** (*kernel threads*)
- obsługiwane przez jądro systemu operacyjnego
- przełączanie między wątkami wymaga wzywania systemu operacyjnego
- **zalety:**
  - wydajniejsze planowanie przydziału czasu procesora
- **wady:**
  - wolniejsze przełączanie wątków – bo zajmuje się tym jądro za pomocą przerwań

Modele wielowątkowości

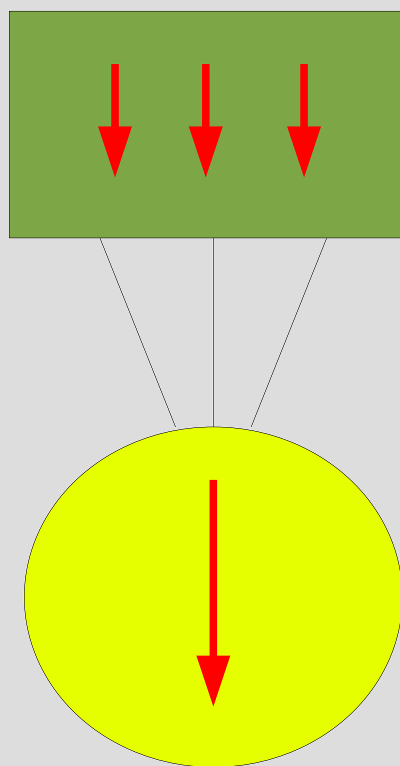
- **„wiele na jeden” (many to one)**
  - wiele wątków użytkownika odwzorowanych na jeden wątek jądra
  - stosowany w systemach nie posiadających wątków jądra
  - np. SUN/Oracle Solaris

- **„jeden na jeden” (one to one)**
  - każdy wątek poziomu użytkownika odwzorowany jednoznacznie na jeden wątek jądra
  - np. Windows NT, OS/2

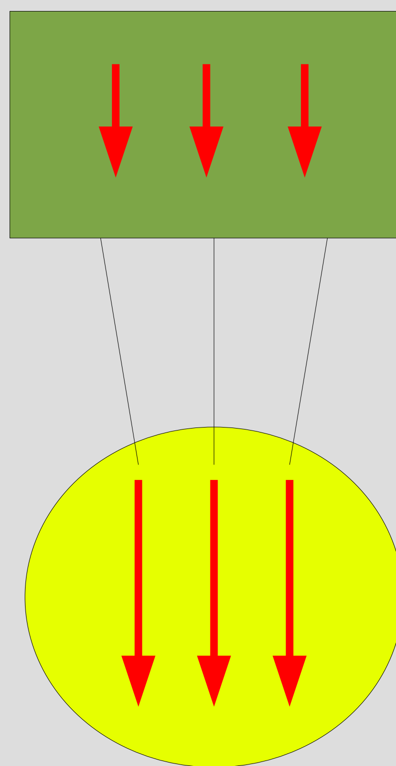
Modele wielowątkowości

- **„wiele na wiele” (many to many)**
- wiele wątków użytkownika jest multipleksowanych na mniejszą lub równą liczbę wątków jądra
- pozwala systemowi operacyjnemu utworzyć dostateczną liczbę wątków jądra → dobra współbieżność i wydajność
- często występuje poziom pośredni w postaci procesów lekkich (LWP) będących dla wątków użytkownika rodzajem wielowątkowych wirtualnych procesorów → z każdym LWP związany jest jeden wątek jądra, natomiast zwykły proces może składać się z jednego lub więcej LWP
- np. SUN/Oracle Solaris 2, IRIX, HP-UX, Tru64 UNIX

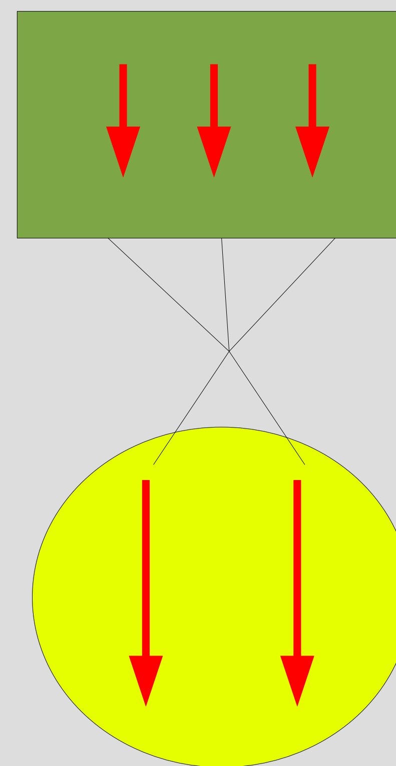
Modele wielowątkowości



wiele na jeden



jeden na jeden



wiele na wiele

Kasowanie wątków

- *thread cancellation*
- kasowanie asynchroniczne – wątek ma możliwość natychmiastowego skasowania innego wątku (tzw. *target thread*)
- niebezpieczne przy współdzielonych zasobach
- kasowanie odroczone – *target thread* okresowo sprawdza, czy polecono mu się zakończyć (tzw. punkty kasowania – *cancellation points*)

pule wątków

- wielowątkowość - powszechnie stosowana w serwerach WWW
- pojawienie się nowego klienta (np. połączenia od przeglądarki) powoduje wytworzenie wątku do jego obsługi
- brak ograniczeń liczby wątków może prowadzić do wyczerpania zasobów systemu
- rozwiązanie: pula wątków – w chwili uruchomienia proces serwera tworzy pewną liczbę wątków (pulę), a wątki się „usypia”
- nadejście połączenia budzi wątek z puli (o ile jest dostępny)
- po obsłużeniu połączenia wątek wraca do puli i jest usypiany

pule wątków

- obsłużenie połączenia za pomocą istniejącego wątku jest tańsze, niż tworzenie nowego wątku
- pula wątków ogranicza ich liczbę, co chroni system przed wyczerpaniem zasobów i spadkiem wydajności

Dane specyficzne wątku

- założenie → wątki dzielą dane procesu macierzystego
- złamanie założenia → dopuszczenie możliwości posiadania przez wątek własnych danych
- np. identyfikatory transakcji bazodanowych
- thread-specific data

Wątki w specyfikacji POSIX

- udostępniane jako biblioteka Pthreads
- dostępne głównie w systemach uniksowych (np. Linux, Solaris, MacOS X)
- nie mają natywnego wsparcia w MS Windows
- z oczywistych powodów brak wyraźnych związków między P-wątkami, a stowarzyszonymi z nimi wątkami jądra.

Biblioteka pthreads

- poprawne skompilowanie kodu używającego **pthreads** wymaga użycia dyrektywy:

```
#include <pthread.h>
```

- poprawne skonsolidowanie programu używającego **pthreads** wymaga użycia opcji:

```
gcc -pthread prog.c
```

Biblioteka pthreads

- pojedynczy wątek jest reprezentowany jako dana typu:

  `pthread_t`

- (definicja typu jest zależna od implementacji, jednak najczęściej jest to `long int`)
- daną tę wykorzystuje się jako identyfikator wątku (tzw. TID – *thread identifier*)

Biblioteka pthreads

- funkcja zawierająca tekst wątku musi być zadeklarowana jako:

```
void *thread(void *data);
```

- nazwa funkcji jest – rzecz jasna – dowolna
- wynikiem funkcji może być wykorzystany do zwrócenia wyniku obliczeń wykonanych przez wątek (najprawdopodobniej będzie konieczne rzutowanie)
- parametr `data` może być wykorzystany do przekazania danych inicjujących wątek (np. może to być wskaźnik na strukturę)

Biblioteka pthreads

- wystartowanie nowego wątku realizuje funkcja:

```
int pthread_create(
      pthread_t *thread,
     pthread_attr_t *attr,
      void *(*start_routine) (void *),
     void *arg
);
```

- start wątku następuje niezwłocznie po wywołaniu funkcji
- funkcja zwraca:
  - 0 w przypadku poprawnego wystartowania nowego wątku
  - kod błędu w przeciwnym przypadku

Biblioteka pthreads

- parametry funkcji `pthread_create()`

```
pthread_t *thread,
```

- wskaźnik na daną, w której funkcja umieści daną identyfikującą wątek (TID)

Biblioteka pthreads

- parametry funkcji `pthread_create()`

```
pthread_attr_t *attr,
```

- wskaźnik na daną specyfikującą atrybuty nowego wątku albo `NULL`, jeśli wątek ma przyjąć atrybuty domyślne

Biblioteka pthreads

- parametry funkcji `pthread_create()`

```
void *(*start_routine) (void *)
```

- wskaźnik na funkcję zawierającą tekst wątku (dostarczoną przez użytkownika biblioteki)

Biblioteka pthreads

- parametry funkcji `pthread_create()`

```
void *arg
```

- wskaźnik na dane inicjujące wątek
- wskaźnik ten zostanie przekazany jako argument do funkcji zawierającej tekst wątku

Biblioteka pthreads

- oczekiwanie na zakończenie wskazanego wątku realizuje funkcja:

```
int pthread_join(
     pthread_t thread,
     void **retval
);
```

- funkcja jest „wątkowym” odpowiednikiem funkcji waitpid() używanej w odniesieniu do procesów
- proces/wątek wywołujący `pthread_join()` jest zawieszany do chwili zakończenia wykonania funkcji
- funkcja zwraca:
  - 0 w przypadku poprawnego wykonania
  - kod błędu w przeciwnym przypadku

Biblioteka pthreads

- parametry funkcji `int pthread_join()`:

  `thread_t thread`

- identyfikator wątku, na zakończenie którego oczekujemy

Biblioteka pthreads

- parametry funkcji `int pthread_join()`:

  `void **retval`

- wskaźnik na daną typu `void*`, w której funkcja umieści wynik zwrócony przez tekst wątku albo `NULL`, jeżeli nie potrzebujemy tej informacji

Biblioteka pthreads

- pobranie własnego TID realizuje funkcja:

```
pthread_t pthread_self(void);
```

- funkcja zawsze kończy się powodzeniem

Biblioteka pthreads

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>

#define THREADS 3

void *thread(void *data) {
    char *msg = (char *) data;
    pthread_t self = pthread_self();
    srand(self);
    printf("Thread %lxd started: \'%s\'\n", self, msg);
    long delay = rand() % 5 + 1;
    sleep(delay);
    printf("Thread %lxd finished\n", self);
    return (void *)delay;
}

:
```

Biblioteka pthreads

```
:

int main(void) {
    pthread_t threads[THREADS];
    char      data[THREADS][20];

    printf("main() started...\n");
    for(int i = 0; i < THREADS; i++) {
        sprintf(data[i],"Thread #%d!", i+1);
        pthread_create(threads + i, NULL, thread, data[i]);
    }
    for(int i = 0; i < THREADS; i++) {
        long retval;
        pthread_join(threads[i], (void *)&retval);
        printf("Thread %d joined: %ld\n", i+1, retval);
    }
    return 0;
}
```

Biblioteka pthreads

```
$ gcc -pthread prog.c -o prog
$ ./prog
main() started...
Thread 7fa971fb6700d started: 'Thread #1!'
Thread 7fa9717b5700d started: 'Thread #2!'
Thread 7fa970fb4700d started: 'Thread #3!'
Thread 7fa971fb6700d finished
Thread 7fa970fb4700d finished
Thread 1 joined: 3
Thread 7fa9717b5700d finished
Thread 2 joined: 4
Thread 3 joined: 3
$
```

Wątki w Windows NT

- aplikacja Windows działa jako osobny proces, który może zawierać jeden lub więcej wątków – stosowane jest odwzorowanie „jeden na jeden”
- dostarczana jest także tzw. biblioteka włókien (fiber) wg. modelu „wiele na wiele”.

Wątki w Windows NT

- poprawne skompilowanie kodu używającego wątków wymaga użycia dyrektywy:

```
#include <windows.h>
```

- sam wątek jest w Windows NT identyfikowany uchwytem, czyli daną o typie:

```
HANDLE
```

Wątki w Windows NT

- funkcja zawierająca tekst wątku musi być zadeklarowana jako:

```
DWORD WINAPI thread(LPVOID data);
```

- funkcję taką nazywa się w terminologii systemu Windows mianem *worker thread* i uznaje się, że wskaźnik na nią jest jest typu `LPTHREAD_START_ROUTINE`
- w przypadku poprawnego zakończenia funkcja powinna zwrócić 0 (zero)
- parametr `data` może być wykorzystany do przekazania danych inicjujących wątek (np. może to być wskaźnik na strukturę)

Wątki w Windows NT

- do wystartowania wątku służy funkcja:

```
HANDLE WINAPI CreateThread(
  LPSECURITY_ATTRIBUTES  lpThreadAttributes,
  SIZE_T                 dwStackSize,
  LPTHREAD_START_ROUTINE lpStartAddress,
  LPVOID                 lpParameter,
  DWORD                  dwCreationFlags,
  LPDWORD                lpThreadId
);
```

- funkcja zwraca:
  - w przypadku powodzenia uchwyt (nie TID!) wystartowanego wątku
  - `NULL` w przeciwnym przypadku

Wątki w Windows NT

- parametry funkcji `CreateThread()`:

```
  LPSECURITY_ATTRIBUTES
lpThreadAttributes,
```

- wskaźnik na zbiór atrybutów zabezpieczeń nowego wątku albo `NULL`, jeśli wątek ma wystartować z atrybutami domyślnymi

Wątki w Windows NT

- parametry funkcji `CreateThread()`:

```
SIZE_T  dwStackSize
```

- rozmiar stosu przydzielonego wątkowi albo `0` dla rozmiaru domyślnego
- konieczność powiększenia stosu może wynikać z dużego rozmiaru zmiennych lokalnych wątku lub ze stosowania rekurencji

Wątki w Windows NT

- parametry funkcji `CreateThread()`:

```
LPTHREAD_START_ROUTINE lpStartAddress
```

- wskaźnik na funkcję dostarczającą tekst wątku

Wątki w Windows NT

- parametry funkcji `CreateThread()`:

```
LPVOID  lpParameter
```

- wskaźnik na dane, które zostaną przekazane do funkcji dostarczającej tekst wątku

Wątki w Windows NT

- parametry funkcji `CreateThread()`:

```
DWORD  dwCreationFlags
```

- maska bitowa określająca opcje tworzenia nowego wątku albo `0` w przypadku ustawienia opcji domyślnych

Wątki w Windows NT

- parametry funkcji `CreateThread()`:

```
LPDWORD lpThreadId
```

- wskaźnik na zmienną, w której zostanie umieszczony TID nowego wątku

Wątki w Windows NT

- do czekania na zakończenie wątku/wątków używa się znanych już funkcji:

```
WaitForSingleObject()
WaitForMultipleObject()
```

Wątki w Windows NT

- do pobrania własnego TID używa się funkcji:

```
DWORD WINAPI GetCurrentThreadId(void);
```

- funkcja zwraca tę samą wartość, którą `CreateThread()` umieszcza w swoim ostatnim argumencie

Wątki w Windows NT

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#define THREADS 3

typedef struct {
    char   msg[20];
    int retval;
} DATA;

DWORD WINAPI thread(LPVOID data) {
        DATA *dat = (DATA *) data;
        DWORD self = GetCurrentThreadId();
        printf("Thread  %xd  started:  \'%s\'\n",  self,  dat-
>msg);
        srand(self);
        DWORD delay = rand() % 5 + 1;
        Sleep(delay * 1000);
        printf("Thread %xd finished\n", self);
        dat->retval = delay;
        return 0;
```

Wątki w Windows NT

```
:
int main(void) {
        HANDLE   threads[THREADS];
        DWORD  thrdids[THREADS];
        DATA     data[THREADS];

        printf("main() started...\n");
        for(int i = 0; i < THREADS; i++) {
            sprintf(data[i].msg,"Thread #%d!", i+1);
            threads[i]                                      =
CreateThread(NULL,0,thread,data+i,0,thrdids+i);
            }

            for(int i = 0; i < THREADS; i++) {
                WaitForSingleObject(threads[i], INFINITE);
            printf("Thread     %d     completed:     %ld\n",     i+1,
data[i].retval);
              CloseHandle(threads[i]);
            }
            return 0;
 }
```

Wątki w Windows NT

```
C:\> cl prog.c
C:\> prog
main() started...
Thread e30d started: 'Thread #1!'
Thread e28d started: 'Thread #3!'
Thread e34d started: 'Thread #2!'
Thread e34d finished
Thread e28d finished
Thread e30d finished
Thread 1 completed: 5
Thread 2 completed: 3
Thread 3 completed: 4
C:\>
```

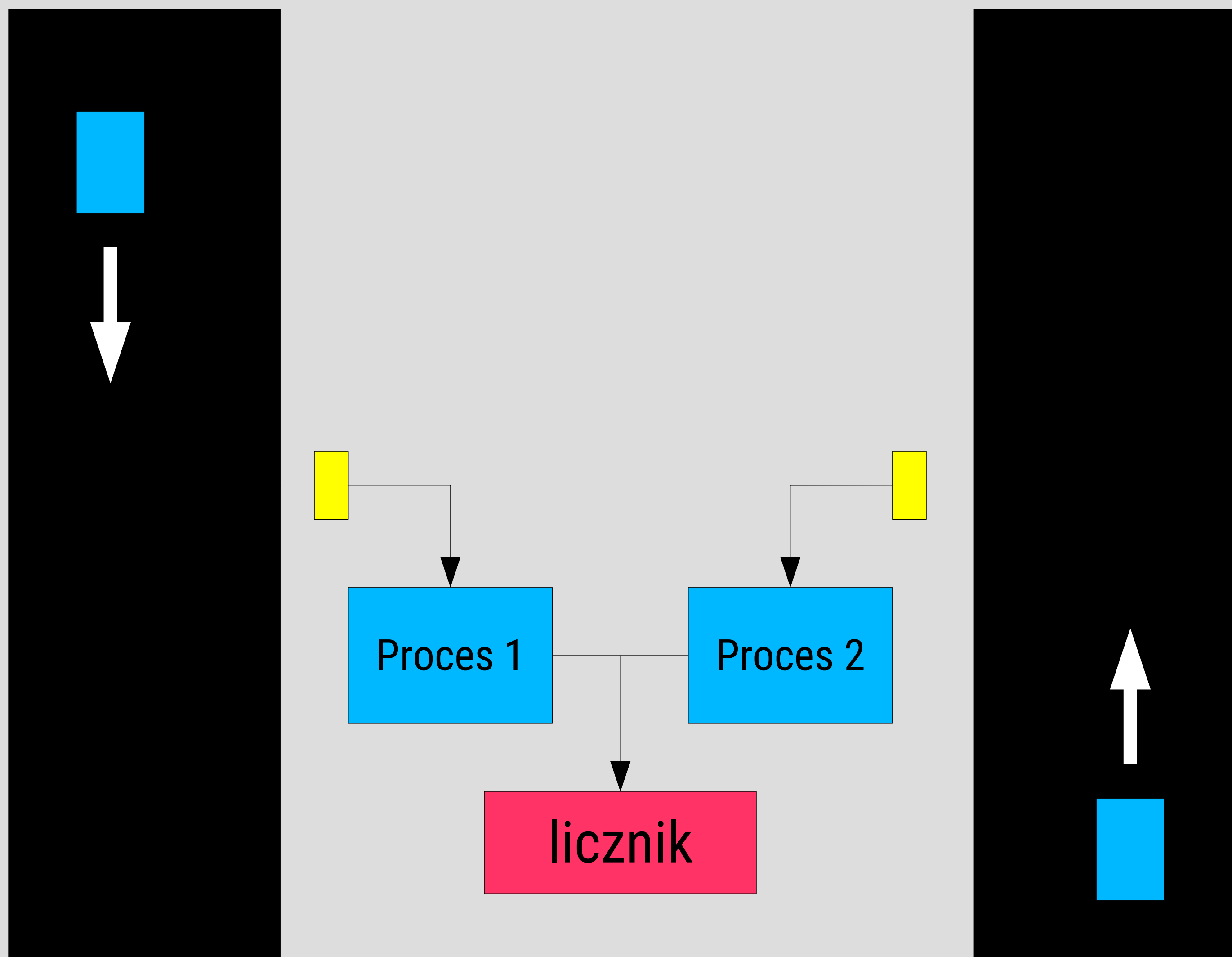
Wątki jądra w Linuksie

- oprócz funkcji systemowej fork() Linux posiada także funkcję systemową clone(), która umożliwia tworzenie oddzielnego procesu dzielącego przestrzeń adresową procesu macierzystego
- istnieje możliwość wyspecyfikowania, co ma być dzielone
- taki proces zachowuje się niemal jak oddzielny wątek
- Linux nie rozróżnia procesów i wątków – z punktu widzenia jądra wątki są osobnymi procesami

Wątki w JVM

- wątki na poziomie języka programowania
- realizowane przez JVM
- konieczne odwzorowanie w wątki jądra systemu macierzystego
- zależy od implementacji JVM w danym systemie.

Zagadnienie przykładowe



Zagadnienie przykładowe

Beztroski kod procesu licznika:

```
while(1) {
   czekaj_na_sygnal_z_czujnika();
   licznik = licznik + 1;
}
```

Pojawiają się błędy zliczeń - dlaczego?

Zagadnienie przykładowe

Ostrożniejszy kod procesu licznika:

```
flaga = 0;
while(1) {
   czekaj_na_sygnal_z_czujnika();
   while(flaga != 0) ;
   flaga = 1;
   licznik = licznik + 1;
   flaga = 0;
}
```

Nadal działa źle - dlaczego?

Sekcja krytyczna

Definicja:

Sekcja krytyczna – fragment kodu uzyskujący dostęp do dzielonego zasobu, który to kod może być wykonywany w co najwyżej jednym procesie (wątku).

Aktualizacja licznika jest **sekcją krytyczną**.

Sekcja krytyczna

Schemat konstrukcji procesu z sekcją krytyczną:

```
while(1) {
    wejście;
    sekcja_krytyczna;
    reszta;
}
```

Sekcja krytyczna

Warunki poprawności rozwiązania problemu **SK**:

Wzajemne wykluczanie – jeżeli jeden proces jest w sekcji krytycznej, to nie ma tam żadnego innego procesu.

Postęp – jeżeli żaden proces nie wykonuje swojej sekcji krytycznej oraz istnieją procesy, które chcą wejść do sekcji krytycznej, to tylko procesy nie wykonujące swoich reszt mogą konkurować o wejście do sekcji krytycznej, a wybór jednego nie może być odwlekany w nieskończoność.

Ograniczone czekanie – między chwilą zgłoszenia chęci wejścia do sekcji krytycznej, a faktycznym wejściem musi istnieć graniczna wartość liczby wejść innych procesów.

Algorytm Dekkera

- Teodor J. Dekker (ur. 1929) – holenderski matematyk
- historycznie pierwsze poprawne rozwiązanie problemu sekcji krytycznej
- spełnia wszystkie trzy warunki
- rozwiązuje dostęp do sekcji krytycznej dla dwóch współpracujących procesów
- kłopotliwy przy uogólnianiu na n procesów
- kłopotliwy przy próbie zabudowania w jądrze jako usługa systemu operacyjnego

Algorytm Dekkera

```
f1 = 0;
f2 = 0;
np = 1; (albo 2 - bez znaczenia)
```

```
f1 := 1;

while(f2 == 1) {
   if(np != 1) {
       f1 := 0;
       while(np != 1) ;
       f1 := 1;
   }
}

<sekcja krytyczna>

np := 2;
f1 := 0;
```

```
f2 := 1;

while(f1 == 1) {
   if(np != 2) {
       f2 := 0;
       while(np != 2) ;
       f2 := 1;
   }
}

<sekcja krytyczna>

np := 1;
f2 := 0;
```

Rozkaz maszynowy TS (Test & Set)

- składnia:

  `TS(zmienna)`
- działanie:

  `zmienna = 1`
- wynik:

  wartość zmiennej **przed** podstawieniem

- gwarantowana atomowość (rozkaz jest wykonywany przy zablokowanych przerwaniach)
- po raz pierwszy pojawił się w komputerze IBM/360

Rozkaz maszynowy TS (Test & Set)

przykład użycia w dostępie do sekcji krytycznej:

```
while(TS(flaga)) ; // czekanie na zwolnienie SK
sekcja_krytyczna();
flaga = 0;           // wyjście z SK
```

Rozkaz maszynowy XCHG (eXCHanGe)

składnia:

`XCHG(zmienna1, zmienna2)`

działanie:

zamiana wartości dwóch zmiennych

wynik:

wartość zmiennej2 **przed** podstawieniem

- gwarantowana atomowość
- dostępny w procesorach x86 Intela

Rozkaz maszynowy XCHG (eXCHanGe)

przykład użycia w dostępie do sekcji krytycznej:

```
mojaflaga = 1;
while(XCHG(mojaflaga,globalna_flaga)) ;
sekcja_krytyczna();
globalnaflaga = 0;
```

Rozwiązanie dla n procesów

- algorytm piekarni (ang. *bakery algorithm*)
- założenia:
  - przed wejściem do SK każdy proces otrzymuje numer
  - proces z najniższym numerem wchodzi do SK
  - jeśli P_i i P_j dostały ten sam numer i $i < j$, to P_i wchodzi pierwszy
  - potrzebne zmienne:
    - `bool wybieranie[N] = { false, false, ... }`
    - `int numer[N] = {0, 0, ... }`

Rozwiązanie dla n procesów

```
do {
     wybieranie[i] = true;
     numer[i] = max(numer) + 1;
     wybieranie[i] = false;
     for (j = 0; j < n; j++){
        while (wybieranie[j]) ;
        while ( numer[j] != 0 &&
               (numer[j] < numer[i] || i < j)) ;
        }
        sekcja krytyczna
        numer[i] = 0;
        reszta
} while (1);
```

Semafory

- semafor (*semaphore*) – pierwszy mechanizm synchronizacyjny w językach wysokiego poziomu (*Dijkstra, 1965*)
- **`semaphore`** – abstrakcyjny typ danych;
- **`semaphore S;`**
  zmienna semaforowa o wartościach całkowitych

Semafory

Operacje na semaforze:

zajęcie semafora (P - *hol. passeren, proberen*)

```
P(S)
{
   while (S <= 0) ; // czekaj
   S--;
}
```

Semafory

Operacje na semaforze:

zwolnienie semafora (V - hol. *vrijmaken, verhogen*)

```
V(S)
{
    S++;
}
```

Semafory

warunki dodatkowe:

- operacje **P** i **V** muszą być zrealizowane jako atomowe (!)
- mogą być dostarczane przez system operacyjny bądź realizowane jako funkcje biblioteczne
- semafor ogólny (ang. *counting semaphore*) – S = {0..n}
- semafor binarny (ang. *binary semaphore*) – S = {0..1}
- semafor ogólny można zaimplementować przy pomocy semafora binarnego i odwrotnie

Semafory - implementacja

- wada semafora zdefiniowanego wcześniej: operacja P zawiera aktywne czekanie (ang. *busy waiting*) → marnowanie cykli procesora
- semafor tak zdefiniowany nazywany jest *wirującą blokadą* (ang. *spinlock*).

Semafory - implementacja

implementacja semafora bez aktywnego czekania:

semafor rozumiany jako struktura (nie jako wartość całkowita):

```
typedef struct {
        int value;
        struct process *L; /* lista procesów */
 } semaphore;
```

system operacyjny powinien dostarczać dwie poniższe usługi:

`block()` – wstrzymuje (blokuje) proces, który ją wywołuje;
`wakeup()` – wznawia zablokowany proces (zmienia stan na gotowy);

Semafory – implementacja P

```
void P(semaphore S)
{
    S.value--;
    if (S.value < 0) {
       // dodaj ten proces do S.L;
       block();
    }
}
```

Semafory – implementacja V

```
void V(semaphore S)
{
    S.value++;
    if (S.value >= 0){
        // usuń jakiś proces P z S.L;
        wakeup(P);
    }
}
```

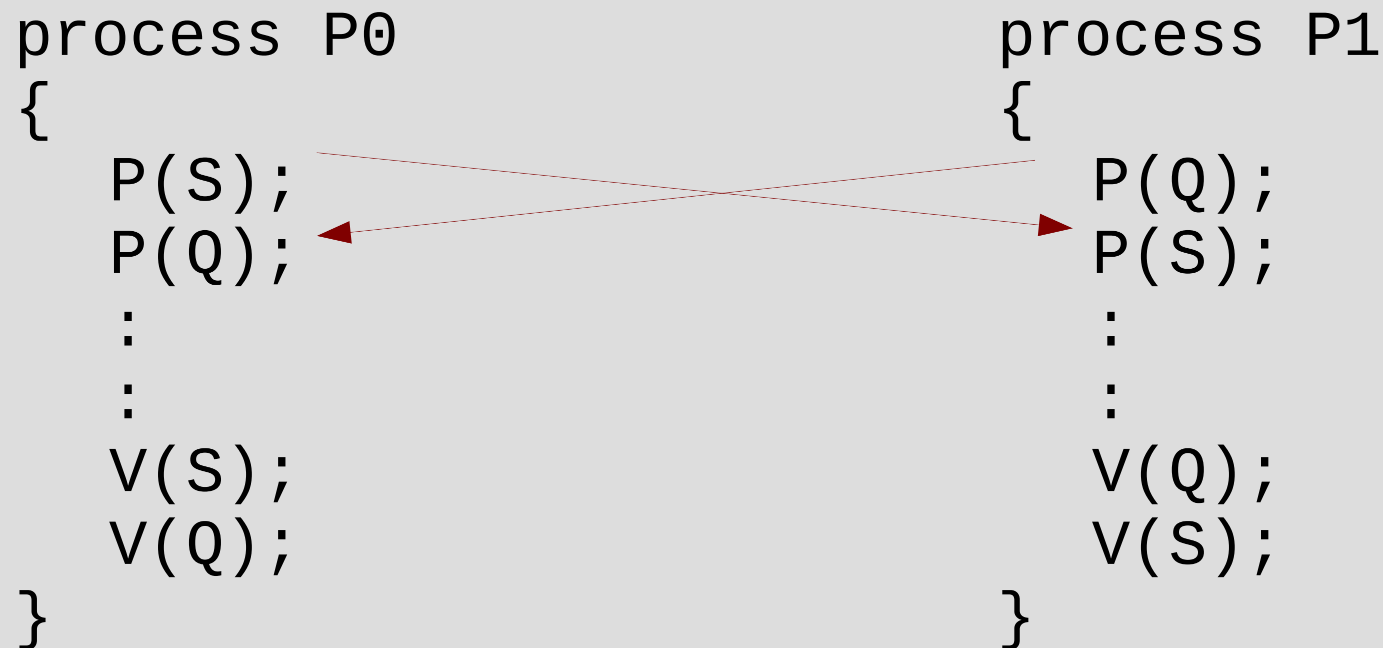
Semafory – użycie

```
semaphore S;

:
P(S);
sekcja_krytyczna();
V(S);
:
```

Semafory - problemy

zakleszczenie (blokada) (ang. *deadlock*) → kilka procesów czeka na zdarzenie, które może być wywołane tylko przez jeden z czekających procesów.



Muteksy

- **mutex** (od ang. *MUTual Exclusion*) – uproszczony semafor binarny
- dostarcza dwie atomowe operacje:
  - **lock** - wejście do sekcji krytycznej: wstrzymuje wykonanie wywołującego go procesu/wątku do chwili zwolnienia SK, a następnie zajmuje ją
  - **unlock** - wyjście z sekcji krytycznej: zwolnienie SK
- implementowany jako funkcja biblioteczna lub jako usługa systemu operacyjnego

Muteksy w bibliotece pthread

- **mutex** jest reprezentowany przez daną typu:

  `pthread_mutex_t`

- dana tego typu może zostać zainicjowana na dwa sposoby:
  - funkcją `pthread_mutex_init()`
  - przez przypisanie symbolu `PTHREAD_MUTEX_INITIALIZER`

    ```
    pthread_mutex_t mutex =
    PTHREAD_MUTEX_INITIALIZER;
    ```

Muteksy w bibliotece pthread

- zajęcie SK wykonuje funkcja:

```
int
pthread_mutex_lock(pthread_mutex_t
*mutex);
```

- w argumencie przekazuje się wskaźnik na zajmowany mutex
- funkcja zwraca:
  - 0 w przypadku powodzenia
  - w przeciwnym przypadku kod błędu

Muteksy w bibliotece pthread

- zwolnienie SK wykonuje funkcja:

```
int
pthread_mutex_unlock(pthread_mutex_t
*mutex);
```

- w argumencie przekazuje się wskaźnik na zwalniany mutex
- funkcja zwraca:
  - 0 w przypadku powodzenia
  - w przeciwnym przypadku kod błędu

Muteksy w bibliotece pthread

- schemat użycia muteksu:

```
pthread_t mymutex = PTHREAD_MUTEX_INITIALIZER;

void *thrd(void *data) {
    :
    pthread_mutex_lock(&mymutex);
    // sekcja krytyczna
    pthread_mutex_unlock(&mymutex);
    :
}
```

Muteksy w Windows NT

- **mutex** jest reprezentowany przez daną typu:

  HANDLE

Muteksy w Windows NT

- utworzenie muteksa wykonuje funkcja:

```
HANDLE WINAPI CreateMutex(
    LPSECURITY_ATTRIBUTES
 lpMutexAttributes,
    BOOL                              bInitialOwner,
    LPCTSTR                           lpName
 );
```

- w wypadku powodzenia funkcja zwraca uchwyt nowego muteksa
- w razie niepowodzenia funkcja zwraca `NULL`

Muteksy w Windows NT

- parametry funkcji `CreateMutex()`:

```
   LPSECURITY_ATTRIBUTES
 lpMutexAttributes,
```

- wskaźnik na zbiór atrybutów zabezpieczeń nowego muteksa albo `NULL` jeśli mutex ma posiadać atrybuty domyślne

Muteksy w Windows NT

- parametry funkcji `CreateMutex()`:

```
BOOL bInitialOwner
```

- jeśli ma wartość różną od zera, mutex zostanie utworzony i od razu **zajęty**
- jeśli równe zero, muteks jest tylko tworzony

Muteksy w Windows NT

- parametry funkcji `CreateMutex()`:

```
LPCTSTR lpName
```

- łańcuch określający nazwę tworzonego muteksa (przydatne, jeśli do tego samego muteksa odwołują się różne procesy – w takim przypadku różne wywołania `CreateMutex()` z tą samą nazwą odnoszą się do tego samego muteksa
- jeśli `NULL`, tworzony jest tzw. *mutex anonimowy*

Muteksy w Windows NT

- zajęcie muteksa wykonuje znana już nam doskonale funkcja:

```
WaitForSingleObject()
```

Muteksy w Windows NT

- zwolnienie muteksa wykonuje funkcja:

```
BOOL WINAPI ReleaseMutex(HANDLE hMutex);
```

- parametrem funkcji jest uchwyt zwalnianego muteksa
- funkcja zwraca wartość różną od zera w razie powodzenia i zero w przeciwnym przypadku

Muteksy w Windows NT

- schemat użycia muteksu:

```
HANDLE mutex;

int main(void) {
    :
    mutex = CreateMutex(NULL, FALSE, NULL);
    :
}

DWORD WINAPI thrd(LPVOID *data) {
      :
      WaitForSingleObject(mutex, INFINITE);
      // sekcja krytyczna
      ReleaseMutex(mutex);
      :
  }
```

Semafory - problemy

(za)głodzenie (blokowanie nieskończone) (ang. ***starvation***) → proces nie zostaje wznowiony, mimo że zdarzenie, na które czeka występuje dowolną liczbę razy – za każdym razem, gdy proces ten mógłby zostać wznowiony, wybierany jest inny czekający proces.

Klasyczne problemy synchronizacji

Producent i Konsument

- dwa rodzaje procesów: Producent i Konsument
- Producent i Konsument dzielą wspólny zasób - bufor dla produkowanych (konsumowanych) jednostek
- Producent wytwarza produkt, umieszcza go w buforze i rozpoczyna pracę od nowa
- Konsument pobiera produkt z bufora i przetwarza go
- Problem → synchronizacja procesów, aby producent nie dodawał nowych jednostek, gdy bufor jest pełny, a konsument nie pobierał, gdy bufor jest pusty

Klasyczne problemy synchronizacji

szkic rozwiązania:

- Producent jest usypiany w momencie, gdy bufor jest pełny
- Konsument, który pobierze element z bufora, budzi proces producenta, który uzupełnia bufor
- Konsument próbujący pobrać z pustego bufora jest usypiany
- Producent po dodaniu nowego produktu umożliwi dalsze działanie konsumentowi
- rozwiązanie może wykorzystywać komunikację międzyprocesową z użyciem semaforów
- nieprawidłowe rozwiązanie może skutkować zakleszczeniem
- rozważać można również uproszczoną wersję problemu z buforem o nieograniczonej pojemności, a także trudniejszą z większą liczbą Producentów i Konsumentów

Klasyczne problemy synchronizacji

```
Semaphore pełny = 0;
Semaphore pusty = ROZMIAR_BUFORA;

process producent() {
    while (1) {
        produkt = produkuj();
        P(pusty);
        dodajProduktDoBufora(produkt);
        V(pełny);
    }
 }

process konsument() {
    while (1) {
        P(pełny);
        produkt = pobierzProduktZBufora();
        V(pusty);
        użyjProdukt(produkt);
    }
}
```

Klasyczne problemy synchronizacji

Czytelnicy i Pisarze

- dwie grupy procesów
- Czytelnicy i Pisarze konkurują o dostęp do wspólnej czytelni
- Czytelnik odczytuje informacje zgromadzone w czytelni i może to robić razem z innymi czytelnikami
- Pisarz zapisuje nowe informacje i musi przebywać sam w czytelni

Klasyczne problemy synchronizacji

szkic rozwiązania:

- Czytelnik powinien wejść do czytelni najszybciej jak to możliwe → możliwość zagłodzenia pisarzy
- Pisarz powinien wejść do czytelni najszybciej jak to możliwe → możliwość zagłodzenia czytelników
- Czytelnicy i pisarze wpuszczani są do czytelni na przemian np. według kolejności zgłoszeń (kolejka), przy czym Pisarze wchodzą pojedynczo, a wchodzący Czytelnik może wpuścić do czytelni wszystkich czekających Czytelników → brak zagłodzenia