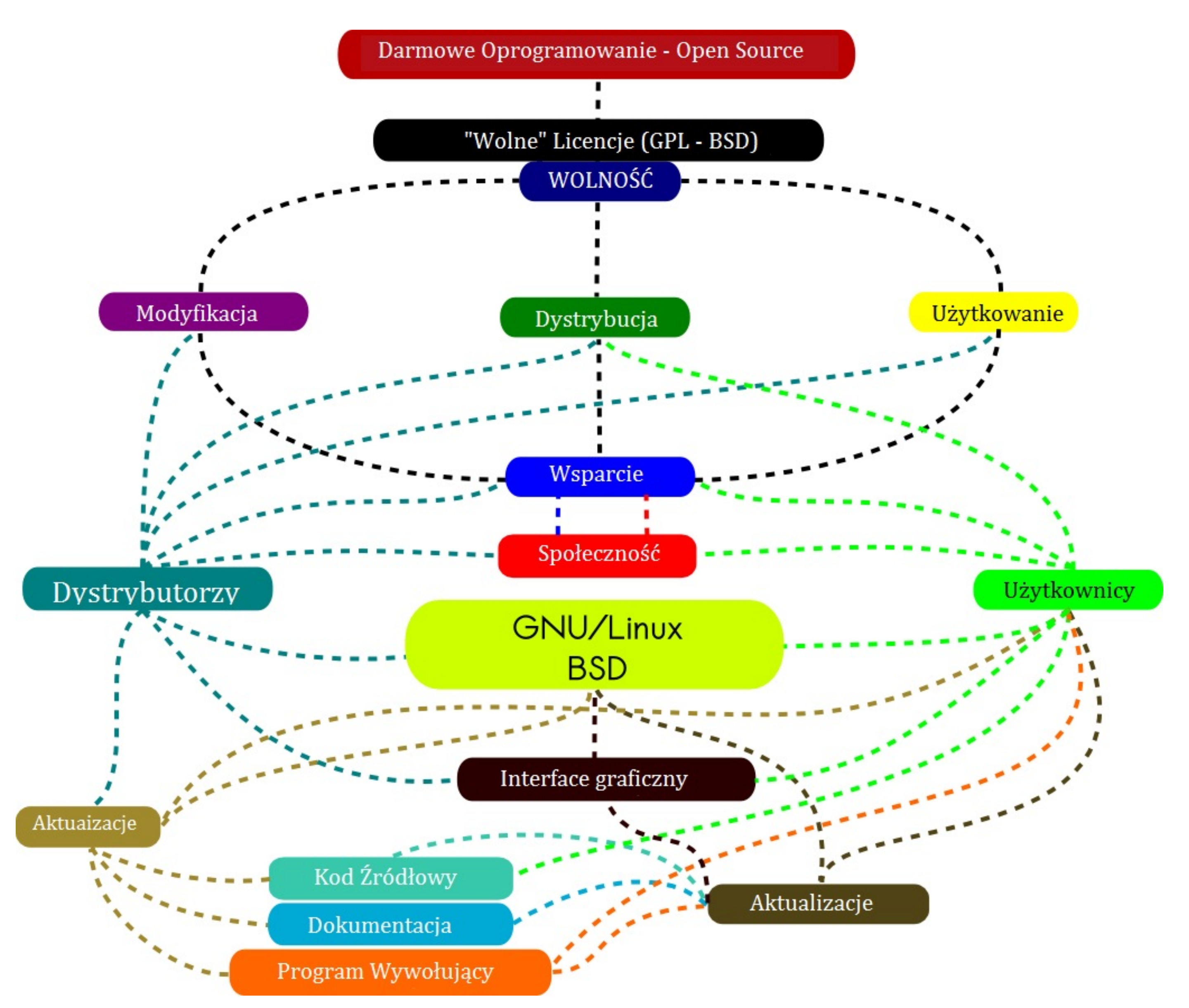
Programowanie równoległe

Rozwiązania systemowe

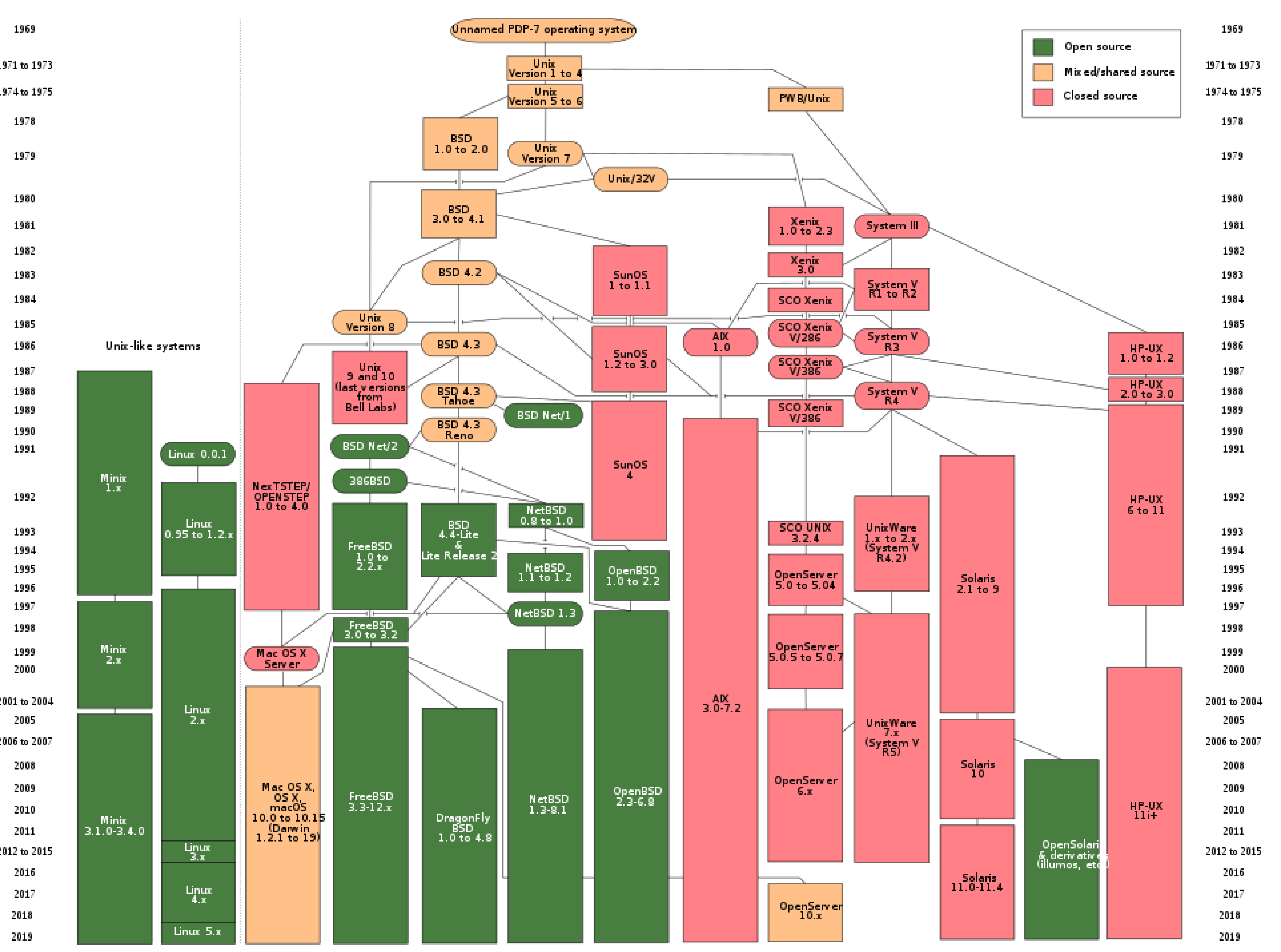
Historia systemu Linux

- Twórca - Linus Torvalds
- Stworzony z niezadowolenia systemem Minix
- Początkowa nazwa Freax (free + freak + [uni]X)
- Należy do Wolnego i otwartego oprogramowania (free and open-source software – FLOSS)
- Wiele rozwiązań tworzone przez programistów-pasjonatów



FLOSS

https://upload.wikimedia.org/wikipedia/commons/e/e5/ConceptualMapFLOSSPL.jpg



https://upload.wikimedia.org/wikipedia/commons/thumb/7/77/Unix_history-simple.svg/1280px-Unix_history-simple.svg.png

POSIX

- Portable Operating System Interface for UNIX
- Standaryzacja środowisk UNIX
- Standard istnieje od 1988 roku (pierwsza publikacja); podmiot standaryzujący IEEE (IEEE 1003)
- Pod standaryzację podlegają: interfejs programistyczny (API); interfejs użytkownika, np. polecenia systemowe takie jak awk, echo, ed; właściwości powłoki systemowej.

P1003.1, P1003.1a

- Definiuje interfejs aplikacji tak, aby była ona w pełni przenośna pomiędzy różnymi systemami operacyjnymi.
- Interfejs ten bazuje na modelach systemu UNIX.
- Zawiera zbiór funkcji które są często implementowane jako wywołania systemowe.

- Zestaw różnych interpretacji, wyjaśnień i rozszerzeń (linki symboliczne).

P1003.1b

- semafory (binarne),
- proces blokowania pamięci,
- pliki mapowane w pamięci i współdzielona pamięć,
- kolejkowanie priorytetowe,
- sygnały w czasie rzeczywistym,
- liczniki czasu (timers),
- komunikacja międzyprocesowa
- synchroniczne operacje wejścia/wyjścia (I/O),
- asynchroniczne operacje wejścia/wyjścia (I/O),

Kolejne standaryzacje:

- P1003.1c: Dodanie funkcji wspierających wątki (lekkie procesy).
- P1003.1d: rozszerzenia wspierające systemy czasu rzeczywistego.
- P1003.1e: Rozszerzenia dotyczące bezpieczeństwa systemu spełniające kryteria bezpieczeństwa opublikowane przez Departament Obrony USA w 'Trusted Computer System Evaluation Criteria' (TCSEC).

Cechy POSIX

- Zorientowanie na aplikacje
- Interfejs, a nie implementacja
- Przenośność źródeł, a nie binariów
- Język C
- Brak opisu administracji systemu
- Minimalizacja
- Wszechstronność

POSIX Threads

- Inaczej pthreads
- Uniwersalna implementacja wielowątkowości w systemach UNIX
- Tworzona dla języka C
- Korzystają z niej wszystkie systemu Unix/Unix-like oraz systemy Windows (częściowa zgodność lub używanie Cygwin)
- Ujednolicone nazewnictwo, spójna budowa funkcji

POSIX Threads – podstawowa funkcjonalność

- tworzenie wątków,
- synchroniczne kończenie wątków,
- asynchroniczne kończenie wątków (sygnał do zakończenia wysyła inny wątek),
- lokalne dane wątku,
- stos funkcji finalizujących (*cleanup*) ułatwiający zarządzanie zasobami w języku C.

POSIX Threads – komunikacja wielowątkowa

- mutexy,
- zmienne warunkowe,
- oczekiwanie na zakończenie wskazanego wątku.

POSIX Threads - pozostałe własności

- dodatkowe mechanizmy synchronizacji:
  - blokady do odczytu/zapisu,
  - bariery,
  - wirujące blokady;
- możliwość współdzielenia obiektów synchronizujących między wątkami różnych procesów;
- indywidualne ustalanie priorytetów wątku i innych parametrów szeregowania;
- ograniczone czasowo oczekiwanie na zajście niektórych zdarzeń (np. założenie blokady);
- odczyt czasu procesora zużyty przez wątek.

Native POSIX Threads Library

- Implementacja POSIX dla systemu Linux stworzona przez Red Hat
- Usprawnia działanie aplikacji wielowątkowych bezpośrednio w jądrze systemu
- Stanowi nadpisanie oryginalnych funkcji wspierających częściowo POSIX (LinuxThreads)
- Od jądra 2.6 stosowane powszechnie (wyparło pozostałe implementacje)

Funkcja fork() - Linux

- Podstawowa funkcja dostępna w każdym systemie Unix/Unix-like
- Wykonuje podział pierwotnego procesu na rodzica i dziecko
- Proces potomny jest wierną kopią pierwotnego (pamięć, zmienne, środowisko)
- Kod potomka rozpoczyna swoje działania w miejscu powołania go do życia (a i proces potomny będzie działał w tym samym miejscu)
- Sama funkcja fork() zwraca pid; może on mieć 3 stany wartości - -1, 0 i dowolną dodatnią

Funkcja fork() - Linux

- wartość -1 zwracana jest gdy nie udało się utworzyć procesu potomnego
- Wartość 0 zarezerwowana jest dla procesu potomnego. Może być użytkowany zarówno do wykonania kodu tylko jako potomek jak i w celach informacyjnych
- Dowolna wartość dodatnia wskaże na proces pierwotny. Sytuacja analogiczna do działania potomka.
- Fork()-bomb -> DoS systemu Linux

Sekcja krytyczna kodu

- W programowaniu współbieżnym może dojść do sytuacji, w której więcej niż jeden wątek będzie chciał zmodyfikować dane we wspólnej sekcji
- Sekcje z modyfikacją wspólnych zmiennych nazywa się "krytycznymi"
- Przeważnie zmienne te należy chronić przed losową bądź nieautoryzowaną modyfikacją
- Oczywiście kod nie musi zawierać żadnych mechanizów ochronnych; jednak tego typu rozwiązanie może skończyć się nieoczekiwanym wynikiem
- To samo tyczy się pozostałych czynności na tego typu zmiennych, jak odczyt, jednoczesny bądź sekwencyjny odczyt i zapis, wiele jednoczesnych zapisów i odczyt, itd.

Algorytm wzajemnego wykluczenia

- mutual exclusion (mutex)
- Stosowane celem uniknięcia modyfikacji współdzielonych zasobów w jednym czasie
- Wykluczenie może odbyć się w części sprzętowej (flaga busy wait, spinlock, compare-and-swap, test-and-set)
- Wykluczenia można także wykonać poprzez odpowiednie operacje programowe

Zasady wykluczenia

- Niedopuszczenie do blokady/zakleszczenia
- Niedopuszczenie do zagłodzenia (uczciwość słaba i mocna)
- Stosowanie kolejkowania, o ile to możliwe
- Brak blokad dla pojedynczych wątków

Rozwiązania

- Algorytm Dekkera
- Algorytm Petersona
- Algorytm piekarniany
- Semafor
- Monitory
- Message passing
- Tuple space
- Problem pięciu filozofów (rozwiązania)

Algorytm Dekkera

```
//flag[] is boolean array; and turn is an integer
flag[0] = false
flag[1] = false
turn    = 0   // or 1
```

```
P0:
  flag[0] = true;
  while (flag[1] == true) {
     if (turn ≠ 0) {
        flag[0] = false;
        while (turn ≠ 0) {
          // busy wait
        }
        flag[0] = true;
     }
  }

  // critical section
  ...
  turn    = 1;
  flag[0] = false;
  // remainder section
```

```
P1:
  flag[1] = true;
  while (flag[0] == true) {
     if (turn ≠ 1) {
        flag[1] = false;
        while (turn ≠ 1) {
          // busy wait
        }
        flag[1] = true;
     }
  }

  // critical section
  ...
  turn    = 0;
  flag[1] = false;
  // remainder section
```

- https://codywu2010.files.wordpress.com/2014/10/dekker_code.jpg

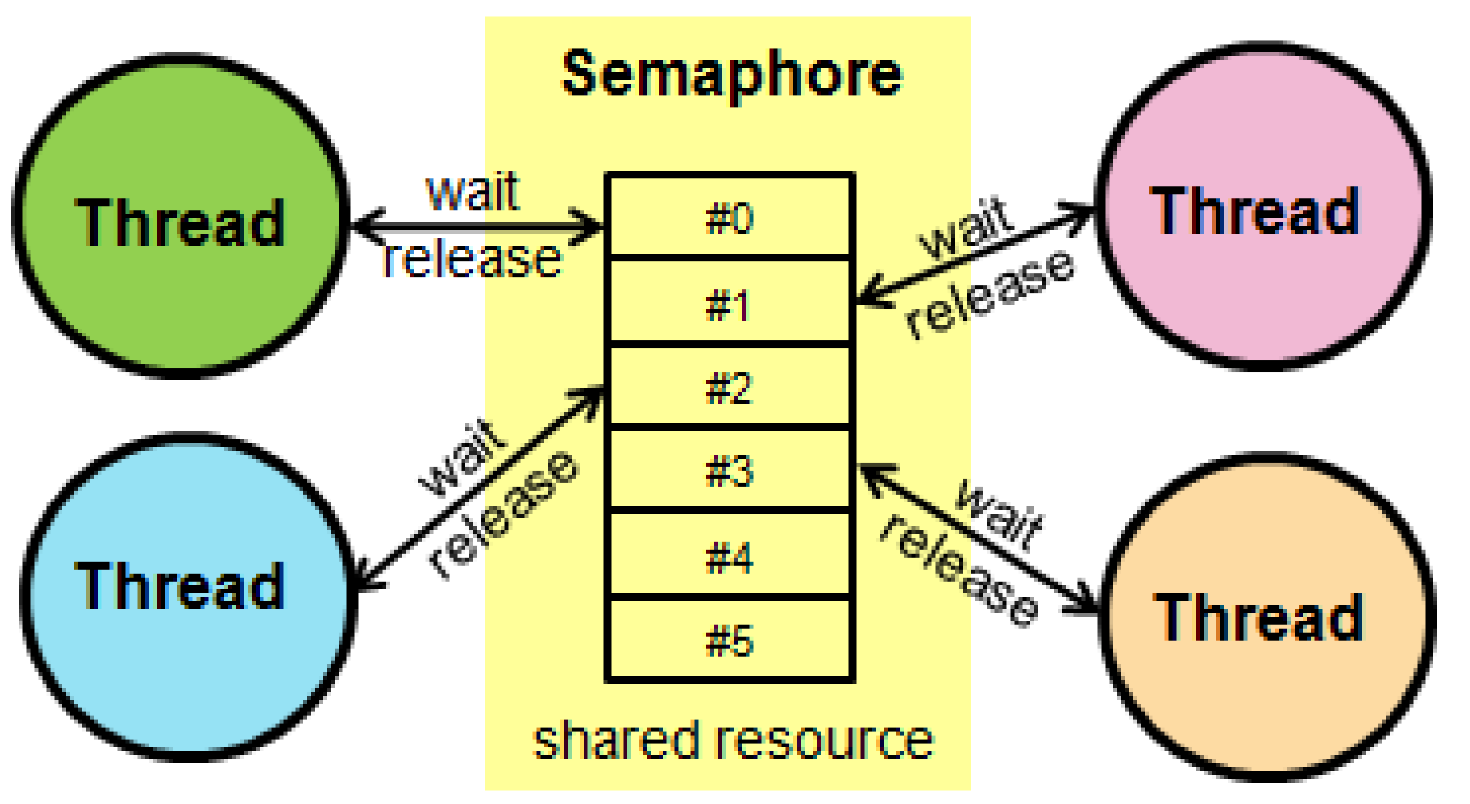
Which thread enters the critical section next?

```
1  class Bakery implements Lock {
2    boolean[] flag;
3    Label[] label;
4    public Bakery (int n) {
5      flag = new boolean[n];
6      label = new Label[n];
7      for (int i = 0; i < n; i++) {
8         flag[i] = false; label[i] = 0;
9      }
10   }
11   public void lock() {
12     int i = ThreadID.get();
13     flag[i] = true;
14     label[i] = max(label[0], ...,label[n-1]) + 1;
15     while ((∃k != i)(flag[k] && (label[k],k) << (label[i],i))) {};
16   }
17   public void unlock() {
18     flag[ThreadID.get()] = false;
19   }
20 }
```

FIGURE 2.10 Pseudocode for the Bakery lock algorithm.

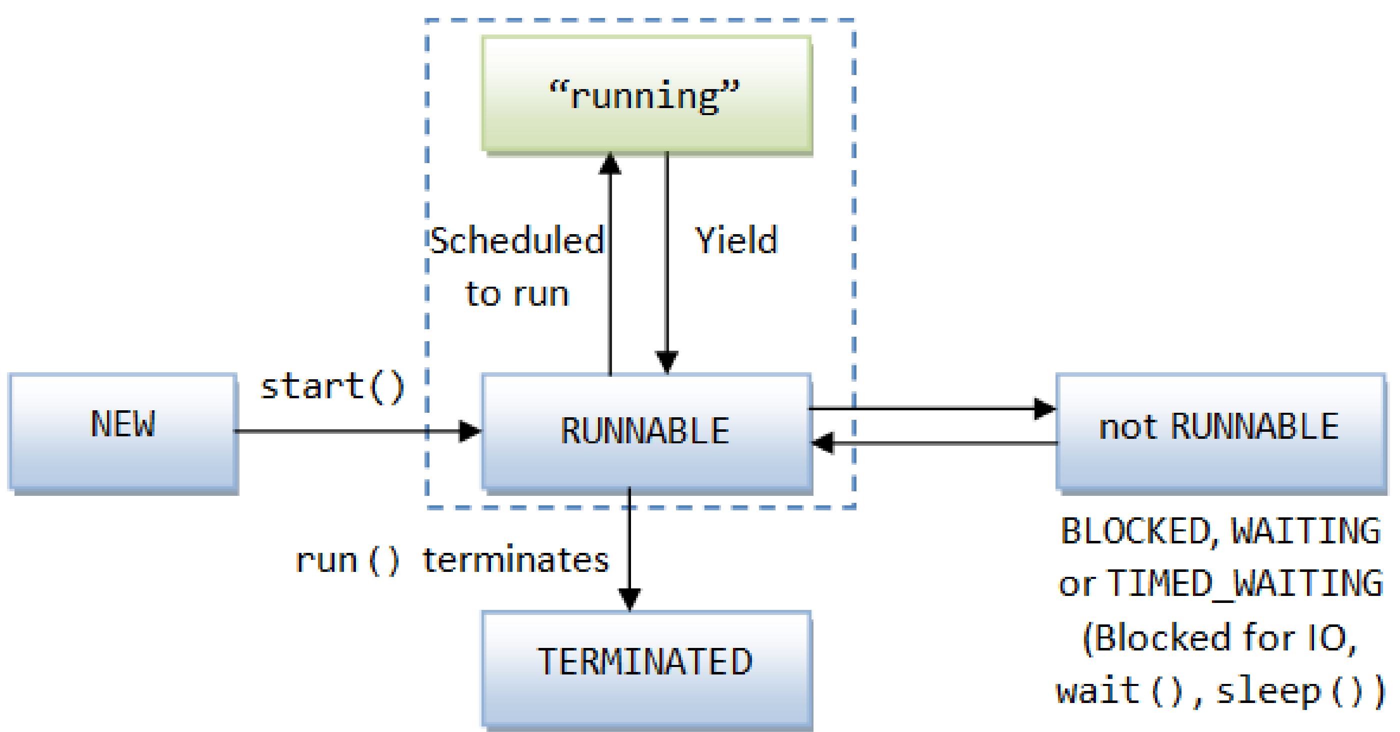
Algorytm piekarniany

- https://media.cheggcdn.com/media/70e/70ea92d5-43e0-44be-bb67-90c948867685/phpVqznQV



Semafor

- https://3.bp.blogspot.com/-7nB5mH-BGhQ/WH0va83GJfI/AAAAAAAAAXf0/1nVX7w72J1Y7ubeRDqURYUtLXtsH7Yg7QCLcB/s1600/semaphore.png



Monitor

- https://www3.ntu.edu.sg/home/ehchua/programming/java/images/Multithread_ThreadLifeCycle.png

Message

Object A

Object B

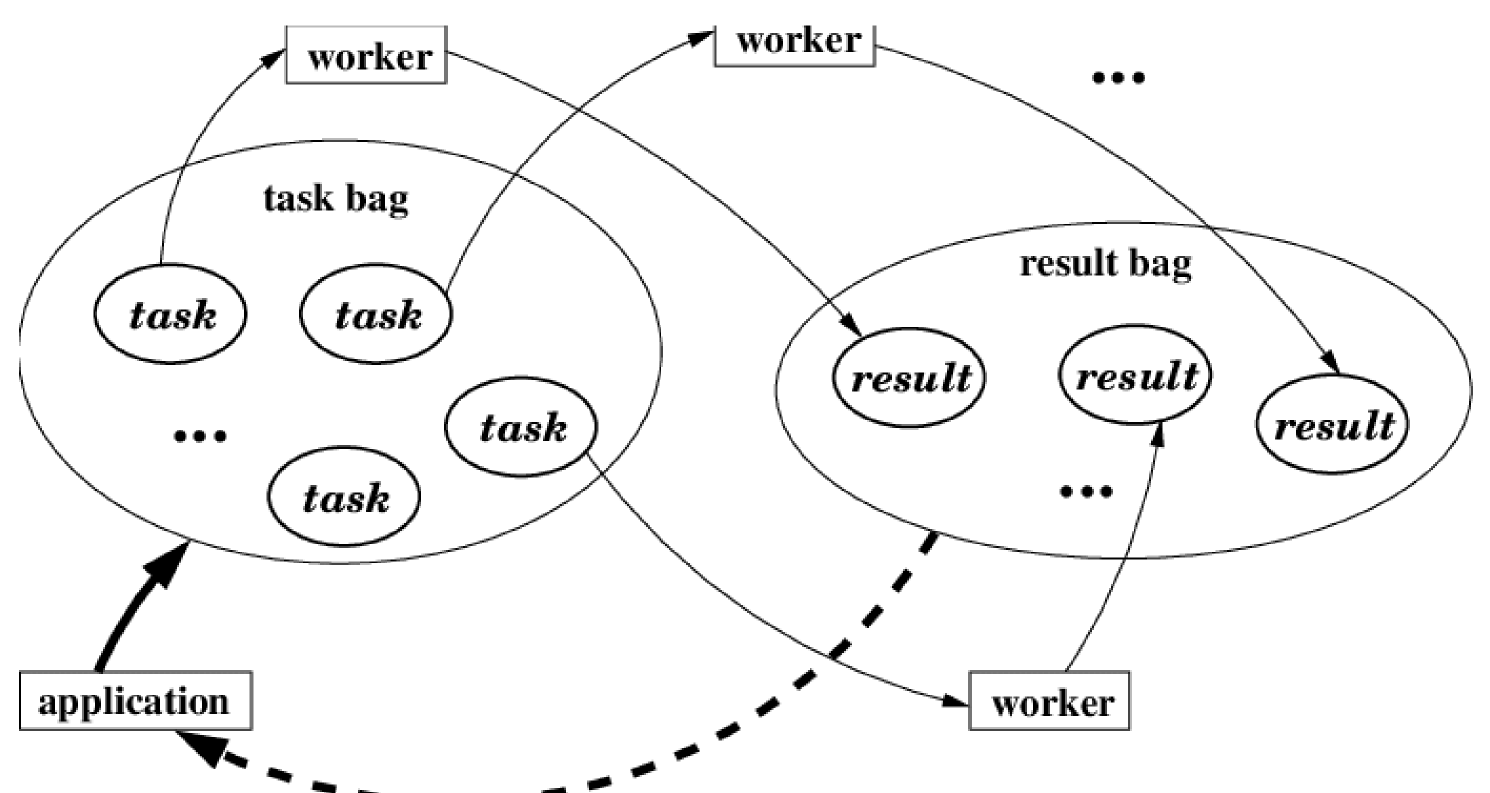
Sending Object

Receiving Object

Message Passing

Message Passing

- https://bparanj.gitbooks.io/ruby-basics/content/message-passing-diagram.png



Tuple Space

- https://www.researchgate.net/profile/Ulrich-Ruede/publication/221001498/figure/fig2/AS:667827098886168@1536233796778/The-tuple-space-paradigm-for-distributed-computing.png

POSIX Threads

- Prosty przykład

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

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

    /* Create independent threads each of which will execute function */

     iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);

     /* Wait till threads are complete before main continues. Unless we  */
     /* wait we run the risk of executing an exit which will terminate   */
     /* the process and all threads before the threads have completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

POSIX Threads

- Mutex

```
void *functionC();
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   int rc1, rc2;
   pthread_t thread1, thread2;

   /* Create independent threads each of which will execute functionC */

   if( (rc1=pthread_create( &thread1, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc1);
   }

   if( (rc2=pthread_create( &thread2, NULL, &functionC, NULL)) )
   {
      printf("Thread creation failed: %d\n", rc2);
   }

   /* Wait till threads are complete before main continues. Unless we  */
   /* wait we run the risk of executing an exit which will terminate   */
   /* the process and all threads before the threads have completed.   */

   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionC()
{
```

POSIX Threads

- Join

```
void *thread_function(void *);
pthread_mutex_t mutex1 = PTHREAD_MUTEX_INITIALIZER;
int  counter = 0;

main()
{
   pthread_t thread_id[NTHREADS];
   int i, j;

   for(i=0; i < NTHREADS; i++)
   {
      pthread_create( &thread_id[i], NULL, thread_function, NULL );
   }

   for(j=0; j < NTHREADS; j++)
   {
      pthread_join( thread_id[j], NULL);
   }

   /* Now that all threads are complete I can print the final result.     */
   /* Without the join I could be printing a value before all the threads */
   /* have been completed.                                                */

   printf("Final counter value: %d\n", counter);
}

void *thread_function(void *dummyPtr)
{
   printf("Thread number %ld\n", pthread_self());
   pthread_mutex_lock( &mutex1 );
   counter++;
   pthread_mutex_unlock( &mutex1 );
}
```

POSIX Threads

- Condition Variable

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

pthread_mutex_t count_mutex     = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t condition_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t  condition_cond  = PTHREAD_COND_INITIALIZER;

void *functionCount1();
void *functionCount2();
int  count = 0;
#define COUNT_DONE  10
#define COUNT_HALT1  3
#define COUNT_HALT2  6

main()
{
   pthread_t thread1, thread2;

   pthread_create( &thread1, NULL, &functionCount1, NULL);
   pthread_create( &thread2, NULL, &functionCount2, NULL);
   pthread_join( thread1, NULL);
   pthread_join( thread2, NULL);

   exit(0);
}

void *functionCount1()
{
   for(;;)
   {
      pthread_mutex_lock( &condition_mutex );
      while( count >= COUNT_HALT1 && count <= COUNT_HALT2 )
      {
         pthread_cond_wait( &condition_cond, &condition_mutex );
      }
      pthread_mutex_unlock( &condition_mutex );

      pthread_mutex_lock( &count_mutex );
      count++;
      printf("Counter value functionCount1: %d\n",count);
      pthread_mutex_unlock( &count_mutex );

      if(count >= COUNT_DONE) return(NULL);
    }
}

void *functionCount2()
{
    for(;;)
    {
       pthread_mutex_lock( &condition_mutex );
       if( count < COUNT_HALT1 || count > COUNT_HALT2 )
       {
          pthread_cond_signal( &condition_cond );
       }
       pthread_mutex_unlock( &condition_mutex );

       pthread_mutex_lock( &count_mutex );
       count++;
       printf("Counter value functionCount2: %d\n",count);
       pthread_mutex_unlock( &count_mutex );

       if(count >= COUNT_DONE) return(NULL);
    }

}
```

```c
#include <stdio.h>       /* standard I/O routines                 */
#include <pthread.h>     /* pthread functions and data structures */

/* function to be executed by the new thread */
void* PrintHello(void* data)
{
    int my_data = (int)data;            /* data received by thread */

    pthread_detach(pthread_self());
    printf("Hello from new thread - got %d\n", my_data);
    pthread_exit(NULL);                 /* terminate the thread */
}

/* like any C program, program's execution begins in main */
int main(int argc, char* argv[])
{
    int         rc;                /* return value                           */
    pthread_t   thread_id;         /* thread's ID (just an integer)          */
    int         t         = 11;    /* data passed to the new thread          */

    /* create a new thread that will execute 'PrintHello' */
    rc = pthread_create(&thread_id, NULL, PrintHello, (void*)t);
    if(rc)                         /* could not create thread */
    {
        printf("\n ERROR: return code from pthread_create is %d \n", rc);
        exit(1);
    }
    printf("\n Created new thread (%u) ... \n", thread_id);

    pthread_exit(NULL);            /* terminate the thread */
}
```

POSIX Threads

- Poprawne zakończenie

C++ Threads

- Prosty przykład

```
// thread example
#include <iostream>       // std::cout
#include <thread>         // std::thread

void foo()
{
  // do stuff...
}

void bar(int x)
{
  // do stuff...
}

int main()
{
  std::thread first (foo);     // spawn new thread that calls foo()
  std::thread second (bar,0);  // spawn new thread that calls bar(0)

  std::cout << "main, foo and bar now execute concurrently...\n";

  // synchronize threads:
  first.join();                // pauses until first finishes
  second.join();               // pauses until second finishes

  std::cout << "foo and bar completed.\n";
```

C++ Threads

- Detach

```
clude <iostream>       // std::cout
clude <thread>         // std::thread, std::this_thread::sle
clude <chrono>         // std::chrono::seconds

d pause_thread(int n)

td::this_thread::sleep_for (std::chrono::seconds(n));
td::cout << "pause of " << n << " seconds ended\n";



 main()

td::cout << "Spawning and detaching 3 threads...\n";
td::thread (pause_thread,1).detach();
td::thread (pause_thread,2).detach();
td::thread (pause_thread,3).detach();
td::cout << "Done spawning threads.\n";

td::cout << "(the main thread will now pause for 5 seconds)\
/ give the detached threads time to finish (but not guarante
ause_thread(5);
eturn 0;
```

C++ Threads

- Operacje globalne

```
// constructing threads
#include <iostream>       // std::cout
#include <atomic>         // std::atomic
#include <thread>         // std::thread
#include <vector>         // std::vector

std::atomic<int> global_counter (0);

void increase_global (int n) { for (int i=0; i<n; ++i) ++global_counter; }

void increase_reference (std::atomic<int>& variable, int n) { for (int i=0; i<n; ++i) ++variable; }

struct C : std::atomic<int> {
  C() : std::atomic<int>(0) {}
  void increase_member (int n) { for (int i=0; i<n; ++i) fetch_add(1); }
};

int main ()
{
  std::vector<std::thread> threads;

  std::cout << "increase global counter with 10 threads...\n";
  for (int i=1; i<=10; ++i)
    threads.push_back(std::thread(increase_global,1000));

  std::cout << "increase counter (foo) with 10 threads using reference...\n";
  std::atomic<int> foo(0);
  for (int i=1; i<=10; ++i)
    threads.push_back(std::thread(increase_reference,std::ref(foo),1000));

  std::cout << "increase counter (bar) with 10 threads using member...\n";
  C bar;
  for (int i=1; i<=10; ++i)
    threads.push_back(std::thread(&C::increase_member,std::ref(bar),1000));

  std::cout << "synchronizing all threads...\n";
  for (auto& th : threads) th.join();

  std::cout << "global_counter: " << global_counter << '\n';
  std::cout << "foo: " << foo << '\n';
  std::cout << "bar: " << bar << '\n';

  return 0;
}
```

C++ Threads

Member types

id	Thread id (public member type)
native_handle_type	Native handle type (public member type)

fx Member functions

(constructor)	Construct thread (public member function)
(destructor)	Thread destructor (public member function)
operator=	Move-assign thread (public member function)
get_id	Get thread id (public member function)
joinable	Check if joinable (public member function)
join	Join thread (public member function)
detach	Detach thread (public member function)
swap	Swap threads (public member function)
native_handle	Get native handle (public member function)
hardware_concurrency [static]	Detect hardware concurrency (public static member function)

fx Non-member overloads

swap (thread)	Swap threads (function)

Odnośniki i materiały dodatkowe

- https://opensource.com/article/19/7/what-posix-richard-stallman-explains
- https://students.mimuw.edu.pl/SO/Projekt05-06/temat4-g2/tomek/index.html
- https://pl.wikipedia.org/wiki/POSIX
- https://pl.wikipedia.org/wiki/POSIX_Threads
- https://akkadia.org/drepper/nptl-design.pdf
- https://students.mimuw.edu.pl/SO/LabLinux/PROCESY/PODTEMAT_2/fork.html
- https://students.mimuw.edu.pl/SO/Linux/Temat02/fork.html

Odnośniki i materiały dodatkowe

- https://docplayer.pl/110653890-Elekcja-wzajemne-wykluczanie-i-zakleszczenie.html
- https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html#SYNCHRONIZATION
- http://www.csc.villanova.edu/~mdamian/threads/posixthreads.html
- https://www.cplusplus.com/reference/thread/thread/