Wątki

dr inż. Sławomir Samolej
Katedra Informatyki i Automatyki
Politechnika Rzeszowska

Program przedmiotu oparto w części na materiałach opublikowanych na:
http://wazniak.mimuw.edu.pl/

oraz
na materiałach opracowanych przez
dr inż. Jędrzeja Ułasiewicza:
jedrzej.ulasiewicz.staff.iiar.pwr.wroc.pl

Potrzeba wprowadzania wątków

- W pewnych przypadkach tworzenie programów złożonych z wielu „ciężkich” procesów uważane jest za mało wydajne.
- Oczekuje się możliwości tworzenia aplikacji współbieżnych, które w bardziej naturalny sposób współdzielą pamięć i zasoby.
- Niektóre systemy operacyjne nie dysponują tak szerokim zestawem mechanizmów programowania współbieżnego opartych na współpracy procesów jak Unix i udostępniają mechanizmy programowania współbieżnego tylko na bazie wątków.

Definicja wątku

- **Sekwencja działań, która może wykonywać się równolegle z innymi sekwencjami działań w kontekście danego procesu (programu).**

Tworzenie wątku a instrukcja fork

- Po uruchomieniu funkcji fork tworzony jest osobny proces z własnym zestawem zmiennych (odziedziczonych po rodzicu) i własnym identyfikatorem proces (PID).
- Po utworzeniu nowego wątku w procesie, otrzymuje on nowy stos (zmienne lokalne) ale współdzieli z innymi zmienne globalne, deskryptory plików, funkcje przechwytujące sygnały i stan bieżącego katalogu i procesu.

Zalety stosowania wątków

- Mniejszy nakład obliczeniowy ze strony systemu operacyjnego
- Możliwość tworzenia aplikacji, które mają wiele wątków pracujących na wspólnych danych lub zasobach (serwer bazy danych, serwer WWW, przetwarzanie w tle tekstu, zapis stanu jednostek w grach strategicznych itp.)
- Możliwość wydzielenia w aplikacji osobnych wątków pobierających dane, wysyłających dane i przetwarzających dane. Przetwarzanie może się odbywać, gdy operacje odczytu/zapisu są zablokowane.
- Przełączanie pomiędzy wątkami wymaga znacznie mniejszego nakładu obliczeniowego od SO

Wady stosowania wątków

- Tworzenie programów wielowątkowych wymaga precyzyjnego projektowania, ponieważ łatwo nieprzewidywalne wyniki spowodowane dostępem do zmiennych w czasie i przestrzeni
- Utrudnione jest śledzenie takich programów
- Aplikacja wielowątkowa na jednoprocesorowym komputerze **nie** musi działać szybciej niż jej jednowątkowa wersja.

Uwagi do tworzenia aplikacji wielowątkowych

- Należy włączyć bibliotekę pthread do opcji konsolidacji
- Należy włączyć makro: _REENTRANT (fragmenty kodu typu re-entrant mogą być równocześnie wywoływane przez wiele wątków w programie i nie zaszkodzi to ich spójności)

Tworzenie wątku

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void
*(*start_routine)(void *), void *arg);
```

Atrybuty:
thread – identyfikator wątku
attr – atrybuty wątku (można podać NULL)
start_routine – wskaźnik na funkcję, od której zaczyna pracę wątek
Wartość zwracana:
0 – sukces, inna liczba – kod będu

Kończenie pracy wątku

```
#include <pthread.h>
void pthread_exit(void *retval);
```

Atrybuty:
retval – identyfikator wątku (można się jeszcze do niego odwołać, choć wątek zakończył działanie)
Uwaga: Jako parametr retval nie należy podawać zmiennych lokalnych.

Oczekiwanie na zakończenie wątku

```
#include <pthread.h>
int pthread_join(pthread_t th, void **thread_return);
```

Atrybuty:
th – identyfikator wątku, na który oczekujemy
thread_return – wskaźnik na wskaźnik pokazujący na wartość zwracaną przez wątek
Funkcja zwraca:
0 – sukces, lub kod błędu.

Pierwszy program wielowątkowy

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

void *thread_function(void *arg);

char message[] = "Hello World";

int main() {
   int res;
   pthread_t a_thread;
   void *thread_result;
   res = pthread_create(&a_thread, NULL,
              thread_function, (void *)message);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
   }
   printf("Waiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
```

```
if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE);
   }
   printf("Thread joined, it returned %s\n", (char *)
                                 thread_result);
   printf("Message is now %s\n", message);
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   printf("thread_function is running. Argument
              was %s\n", (char *)arg);
   sleep(3);
   strcpy(message, "Bye!");
   pthread_exit("Thank you for the CPU time");
}
```

```
$ ./thread1
Waiting for thread to finish...
thread_function is running. Argument was Hello World
Thread joined, it returned Thank you for the CPU time
Message is now Bye!
```

Jak to działa?

- thread_function jest funkcją, która rozpocznie działanie w nowym wątku
- W funkcji powoływany jest nowy wątek, obsługiwany przez thead_function, wątek przyjmuje domyślne parametry (NULL), a do funkcji wykonującej jako parametr zostaje wysłana tablica message
- Program główny kontynuuje swoje obliczenia i oczekuje na zakończenie utworzonego wątku (pthread_join)
- Wątek potwierdza odebranie danych przez parametr wywołania funkcji, modyfikuje stan współdzielonej zmiennej i kończy swoje działanie (pthread_exit).
- Następuje zakończenie programu macierzystego.

Sprawdzenie, czy wątki się przełączają?

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

void *thread_function(void *arg);
int run_now = 1;

int main() {
   int res;    pthread_t a_thread;
   void *thread_result;
   int print_count1 = 0;
   res = pthread_create(&a_thread, NULL,
thread_function, (void *)message);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
   }

   while(print_count1++ < 20000000) {
      if (run_now == 1) {
         printf("1");
         run_now = 2;    } }
```

```
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE);
   }
   printf("Thread joined\n");
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   int print_count2 = 0;

   while(print_count2++ < 20000000) {
      if (run_now == 2) {
         printf("2");
         run_now = 1;     }}
   sleep(3);
}
```

```
$ ./thread2
12121212121212121212
Waiting for thread to finish...
Thread joined
```

Problem aktywnego czekania…

- Każdy z wątków aplikacji sprawdza stan zmiennej run_now.
- Jeśli przyjmuje ona zadaną wartość, program wypisuje tę wartość i zmienia na drugą z możliwych
- Oba wątki wykonują niekorzystne z punktu wydajności systemu **aktywne czekanie** i próbkowanie stanu zmiennej
- Większość mocy obliczeniowej procesora zużywana jest na sprawdzanie stanu zmiennej w każdym z wątków.

Definicja semafora

- Semafor S jest obiektem systemu operacyjnego z którym związany jest licznik L zasobu przyjmujący wartości nieujemne. Na semaforze zdefiniowane są *atomowe* operacje **sem_init**, **sem_wait** i **sem_post**.

Operacja	Oznaczenie	Opis
Inicjacja semafora S	sem_init(S,N)	Ustawienie licznika semafora S na początkową wartość N .
Zajmowanie	sem_wait(S)	Gdy licznik L semafora S jest dodatni (L > 0) zmniejsz go o 1 (L = L – 1). Gdy licznik L semafora S jest równy zero (L= 0) zablokuj proces bieżący.
Sygnalizacja	sem_post(S)	Gdy istnieje jakiś proces oczekujący na semaforze S to odblokuj jeden z czekających procesów. Gdy brak procesów oczekujących na semaforze S zwiększ jego licznik L o 1 (L=L+1).

Algorytmy funkcji semafora

```
sem_wait(S)
{   if(Licznik L sem. S dodatni)
    { L=L-1;}
    else
    { Zawieś proces bieżący }
}
```

```
sem_post (S)
{   if(Istnieje proc. czekający na zasób)
    {  Odblokuj jeden z tych procesów }
    else
    {    L=L+1;  }
}
```

Uwaga!

1. Semafor nie jest liczbą całkowitą na której można wykonywać operacje arytmetyczne .
2. Operacje na semaforach są operacjami atomowymi.

Niezmiennik semafora

Aktualna wartość licznika L semafora S spełnia następujące warunki:

1. Jest nieujemna czyli: L >= 0
2. Jego wartość wynosi: L= N - Liczba_operacji(sem_wait) + Liczba_operacji(sem_post).

(N jest wartością początkową licznika).

Semafor binarny

W semaforze binarnym wartość licznika przyjmuje tylko dwie wartości: 0 i 1.

Rodzaje semaforów

Wyróżniamy następujące rodzaje semaforów:

1. Semafor ze zbiorem procesów oczekujących (*ang. Blocked- set Semaphore*) – Nie jest określone który z oczekujących procesów ma być wznowiony.

2. Semafor z kolejką procesów oczekujących (*ang. Blockedqueue Semaphore*) – Procesy oczekujące na semaforze umieszczone są w kolejce FIFO.

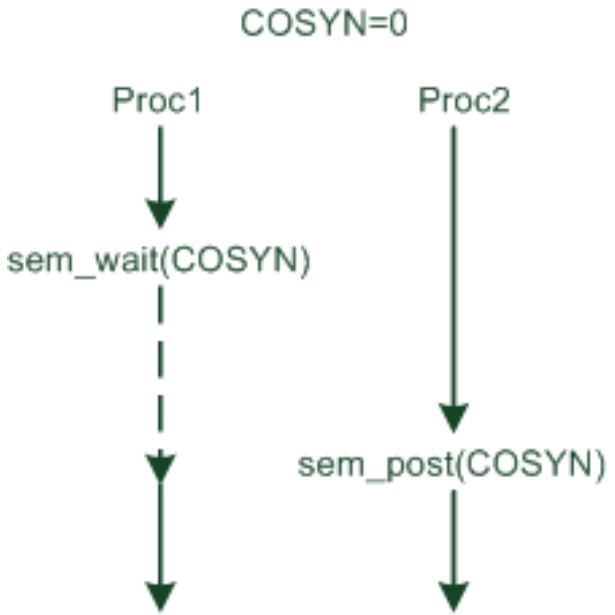
Uwaga!
Pierwszy typ semafora nie zapewnia spełnienia warunku zagłodzenia.

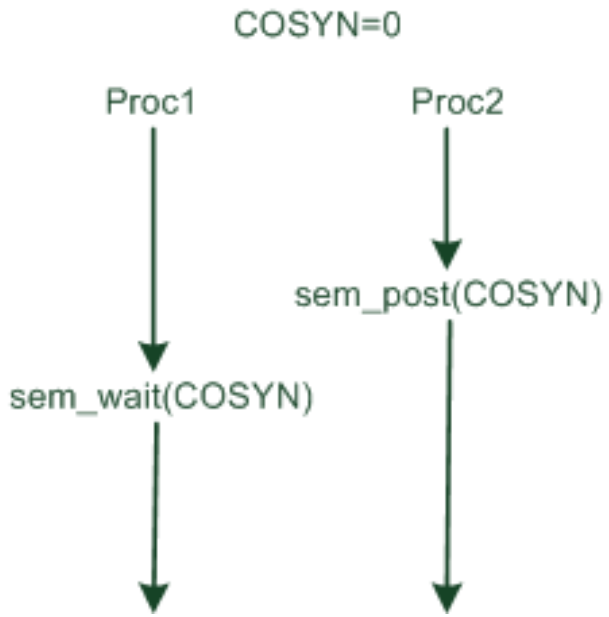
Podstawowe zastosowania semaforów - synchronizacja

```
var consyn : semaphore (* init 0 *)
```

```
process P1;
   (* waiting process *)
   statement X;
   sem_wait (consyn)
   statement Y;
end P1;
```

```
process P2;
   (* signalling proc *)
   statement A;
   sem_post (consyn)
   statement B;
end  P2; `
```



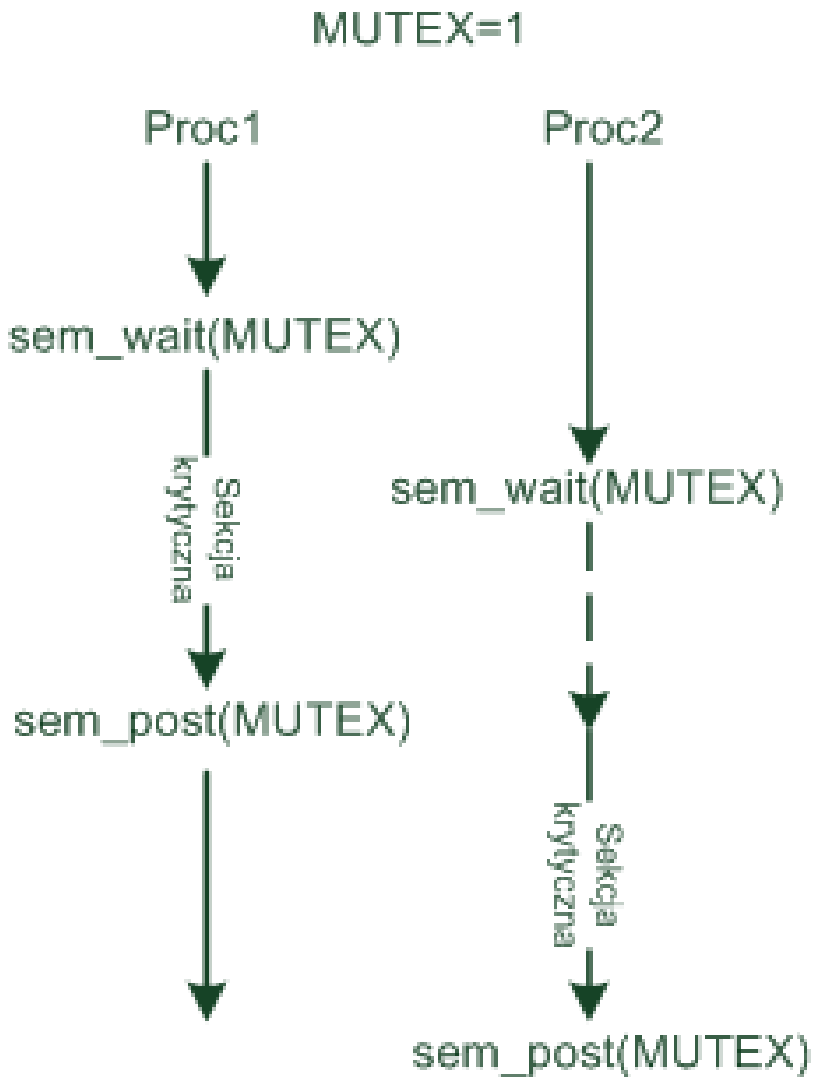


Podstawowe zastosowania semaforów – wzajemne wykluczanie dostępu

```
var mutex : semaphore; (* initially 1 *)
```

```
process P1;
   statement X
   sem_wait (mutex);
      statement Y
   sem_post (mutex);
   statement Z
end P1;
```

```
process P2;
   statement A;
   sem_wait (mutex);
      statement B;
   sem_post (mutex);
   statement C;
end P2;
```



Semafory POSIX (1)

Semafory nazwane identyfikowane są w procesach poprzez ich nazwę.

Funkcja tworząca semafor:

sem_t *sem_open(const char *name, int oflag,…);
sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode , int value);

Parametry:
name – nazwa semafora w systemie, powinna zacząć się od znaku „/”.
oflag – jest ustawiana na O_CREAT gdy chcemy utworzyć semafor (jeśli dodamy flagę O_EXCL funkcja zwróci błąd, w przypadku, gdy taki semafor już istnieje).
mode_t – ustalenie kontroli dostępu do semafora
value – ustalenie wartości początkowej semafora.

Funkcja zwraca „uchwyt” do semafora lub błąd SEM_FAILED z zapisaną odpowiednią wartością w zmiennej errno. Od tej chwili w programie dostęp do semafora odbywa się za pomocą tego „uchwytu”.

Pojedyncze wywołanie tworzy semafor, inicjalizuje go i ustala zasady dostępu do niego.

Semafory POSIX (2)

Funkcja inicjalizująca semafor: **int sem_init(sem_t *sem, int pshared, unsigned int value);**
pshared : 0-semofor jest dzielony pomiędzy wątki, >0 może być dzielony pomiędzy procesy
value: początkowa wartość semafora

Funkcja „opuszczająca” semafor: **int sem_wait(sem_t *sem);**
Funkcja „podnosząca” semafor: **int sem_post(sem_t *sem);**
Funkcja zamykająca dostęp do semafora: **int sem_close(sem_t *sem);**
Funkcja usuwająca semafor z sytemu: **int sem_unlink(const char *name);**

Uwagi: Po zakończeniu korzystania z semafora należy zamknąć do niego dostęp wywołując funkcję sem_close(). Usunięcie semafora z sytemu odbywa się po wywołaniu funkcji sem_unlink(). Jeśli jakiś inny proces lub wątek mają dostęp do tego semafora, to wywołanie funkcji sem_unlink() nie da żadnego efektu.
Funkcje dodatkowe:
int sem_trywait(sem_t *sem); - przejęcie semafora, jeśli jest dostępny, lub zgłoszenie błędu i kontynuowanie wątku obliczeniowego.

int sem_timedwait(sem_t *sem, const struct timespec *abs_timeout);
/* struct timespec { time_t tv_sec; /* Seconds */
long tv_nsec; /* Nanoseconds [0 .. 999999999] */ };
*/ - przejęcie semafora do wywołania jeśli jest dostępny lub oczekiwanie na przejęcie przez określony czas.

Synchronizacja z zastosowaniem semaforów

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

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
   int res;
   pthread_t a_thread;
   void *thread_result;

   res = sem_init(&bin_sem, 0, 0);
   if (res != 0) {
      perror("Semaphore initialization failed");
      exit(EXIT_FAILURE); }
   res = pthread_create(&a_thread, NULL,
thread_function, NULL);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE); }
```

```
printf("Input some text. Enter 'end' to finish\n");
   while(strncmp("end", work_area, 3) != 0) {
      fgets(work_area, WORK_SIZE, stdin);
      sem_post(&bin_sem);
   }
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE);
   }
   printf("Thread joined\n");
   sem_destroy(&bin_sem);
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   sem_wait(&bin_sem);
   while(strncmp("end", work_area, 3) != 0) {
      printf("You input %d characters\n",
strlen(work_area) -1);
      sem_wait(&bin_sem);
   }
   pthread_exit(NULL);
}
```

Wyjście programu

```
$./thread3
Input some text. Enter ‘end’ to finish
The Wasp Factory
You input 16 characters
Iain Banks
You input 10 characters
end
Waiting for thread to finish...
Thread joined
```

Komentarz

- W programie rozwiązano problem synchronizacji z zastosowaniem semaforów
- Wartość początkowa semafora ustawiana jest na 0
- Podstawowy wątek cyklicznie odbiera od użytkownika komunikaty tekstowe i zapisuje do bufora, a następnie zwalnia semafor (sem_post).
- Wątek potomny najpierw oczekuje na zwolnienie semafora (sem_wait), a potem cyklicznie analizuje zawartość bufora z danymi (długość tekstu) i oczekuje na ustawienie semafora (sem_wait).

Zależności czasowe

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

void *thread_function(void *arg);
sem_t bin_sem;

#define WORK_SIZE 1024
char work_area[WORK_SIZE];

int main() {
   int res;    pthread_t a_thread;
   void *thread_result;
   res = sem_init(&bin_sem, 0, 0);
   if (res != 0) {
      perror("Semaphore initialization failed");
      exit(EXIT_FAILURE);    }
   res = pthread_create(&a_thread, NULL,
                  thread_function, NULL);
   if (res != 0) {   perror("Thread creation failed");
      exit(EXIT_FAILURE);    }
```

```
   printf("Input some text. Enter 'end' to finish\n");
   while(strncmp("end", work_area, 3) != 0) {
    if (strncmp(work_area, "FAST", 4) == 0) {
      sem_post(&bin_sem);
      strcpy(work_area, "Wheeee...");
    } else {
      fgets(work_area, WORK_SIZE, stdin);}
    sem_post(&bin_sem);
   }
   printf("\nWaiting for thread to finish...\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) {
      perror("Thread join failed");
      exit(EXIT_FAILURE); }
   printf("Thread joined\n");
   sem_destroy(&bin_sem);
   exit(EXIT_SUCCESS);
}
void *thread_function(void *arg) {
   sem_wait(&bin_sem);
   while(strncmp("end", work_area, 3) != 0) {
      printf("You input %d characters\n",
                                  strlen(work_area) -1);
      sem_wait(&bin_sem);    }
   pthread_exit(NULL);
}
```

Wyjście programu

```
$ ./thread4a
Input some text. Enter ‘end’ to finish
Excession
You input 9 characters
FAST
You input 7 characters
You input 7 characters
You input 7 characters
end
Waiting for thread to finish...
Thread joined
```

Komentarz

- Kiedy użytkownik wprowadza słowo FAST na wejście programu to następuje **zwolnienie semafora**,
- Warunek zakończenia pętli while nie jest spełniony
- Następuje rozpoznanie, że użytkownik wprowadził wyraz „FAST” , ponowne **zwolnienie semafora**, a następnie wprowadzenie do współdzielonej zmiennej tekstu „Wheeee…”
- Ponowny raz semafor **zostaje zwolniony**
- Warunek zakończenia pętli while nie jest spełniony
- Program oczekuje na wprowadzenie nowego tekstu na standardowym wejściu
- Zmiany zawartości współdzielonej zmiennej odbywają się tak szybko, że wątek obliczający długość tekstu „gubi” jedną ze zmian.
- Wątek macierzysty podczas wywoływania funkcji sem_post() zwiększa licznik.
- W konsekwencji wątek potomny, który wcześniej nie uzyskał dostępu do współdzielonej zmiennej trzykrotnie „odblokowuje się na semaforze” i wypisuje obliczoną długość tekstu.

Mutexy

- Wątki dzielą wspólny obszar danych. Stąd współbieżny dostęp do danych może naruszyć ich integralność.
- Należy zapewnić synchronizację dostępu do wspólnych danych.
- W bibliotece pthreads do zapewnienia wyłączności dostępu do danych stosuje się mechanizm muteksu (*ang. mutex*). Nazwa ta pochodzi od słów *Mutual exclusion* czyli wzajemne wykluczanie.

Funkcje do obsługi mutexów

```
#include <pthread.h>
//Inicjalizacja:
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr);
// Zamknięcie dostępu do sekcji krytycznej:
int pthread_mutex_lock(pthread_mutex_t *mutex));
// Otwarcie dostępu do sekcji krytycznej:
int pthread_mutex_unlock(pthread_mutex_t *mutex);
// Zniszczenie mutex'a
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Uwagi:

Funkcje korzystają ze wcześniej zadeklarowanego obiektu typu:

pthread_mutex_t

Funkcja dodatkowa:

```
// - przejęcie mutex'a, jeśli jest dostępny, lub zgłoszenie błędu i kontynuowanie wątku obliczeniowego.
int pthread_mutex_trylock(pthread_mutex_t *mutex);
// - przejęcie mutex'a, jeśli jest dostępny lub oczekiwanie przez abs_timeout na jego przejęcie
int pthread_mutex_timedlock(pthread_mutex_t *restrict mutex,
      const struct timespec *restrict abs_timeout);
```

Przykład – ochrona współdzielonych danych z zastosowaniem mutex'a

```
#define _REENTRANT
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <string.h>

void *thread_function(void *arg);
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE];
char texts[3][50]={ "First text\n",
                    "Second text\n",
                    "Third text\n"};
int main() {
   int res;    pthread_t a_thread;
   void *thread_result; int i=0,j=0;
   res = pthread_mutex_init(&work_mutex, NULL);
   if (res != 0) {      perror("Mutex initialization failed");
                        exit(EXIT_FAILURE);    }
   res = pthread_create(&a_thread,
                              NULL,thread_function, NULL);
   if (res != 0) {      perror("Thread creation failed");
                        exit(EXIT_FAILURE);    }
```

```
while(j<15) {
      pthread_mutex_lock(&work_mutex);
         strcpy(work_area,texts[i]);
      pthread_mutex_unlock(&work_mutex);
      i = (i + 1) % 3;
      j++;
      sleep(2);
   }
   printf("Bye\n");
   res = pthread_join(a_thread, &thread_result);
   if (res != 0) { perror("Thread join failed");
                        exit(EXIT_FAILURE); }
   pthread_mutex_destroy(&work_mutex);
   exit(EXIT_SUCCESS);
}
void *thread_function(void *arg) {
   int i=0;
   while(i<20) {
      pthread_mutex_lock(&work_mutex);
         printf("Current text: %s",work_area);
      pthread_mutex_unlock(&work_mutex);
      sleep(1);
      i++;
   }
   pthread_exit(0);
}
```

Wyjście programu

```
Current text: First text
Current text: First text
Current text: Second text
Current text: Second text
Current text: Third text
Current text: Third text
Current text: First text
Current text: First text
Current text: Second text
Current text: Second text
Current text: Third text
Current text: Third text
Current text: First text
Current text: First text
Current text: Second text
Current text: Second text
Current text: Third text
Current text: Third text
Current text: First text
Current text: First text
Bye
```

Jak to działa?

- W obszarze zmiennych globalnych zadeklarowano nową zmienną **work_mutex**
- W głównej funkcji programu zainicjalizowano mutex
- Powołano nowy wątek który:
  - próbuje zamknąć mutex (jeśli jest zamknięty, to oczekuje na jego otwarcie),
  - wypisuje stan współdzielonej zmiennej **work_area**
  - otwiera mutex
- W wątku macierzystym :
  - następuje próba zamknięcia mutex'a
  - do współdzielonego obszaru pamięci następuje przypisanie kolejnej wartości z tablicy napisów **texts**
  - następuje otwarcie semafora
- Uwaga:
  - Program pokazuje zasadę wymiany informacji pomiędzy wątkami na zasadzie **wzajemnego wykluczania**. Tylko jeden z puli wątków może wykonywać „sekcję krytyczną" - zestaw indtrukcji „ochroninych" przez mutex.

Argumenty wątków (wybrane)

```
#include <pthread.h>
// Inicjalizacja możliwości ustalania atrybutów wątków
int pthread_attr_init(pthread_attr_t *attr);

// Ustalenie sposobu zakończenia wątku:
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);

// Sprawdzenie, w jaki sposób wątek będzie kończony
int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);

// Ustalenie sposobu szeregowania wątku:
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

// Sprawdzenie w jaki sposób wątek będzie szeregowany:
int pthread_attr_getschedpolicy(const pthread_attr_t *attr, int *policy);

//
//...
//
```

Kończenie wątku bez oczekiwania na wspólne zakończenie

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

void *thread_function(void *arg);

char message[] = "Hello World";
int thread_finished = 0;

int main() {
   int res;
   pthread_t a_thread;
   void *thread_result;
   pthread_attr_t thread_attr;

   res = pthread_attr_init(&thread_attr);
   if (res != 0) {
      perror("Attribute creation failed");
      exit(EXIT_FAILURE);
   }
   res = pthread_attr_setdetachstate(&thread_attr,
PTHREAD_CREATE_DETACHED);
   if (res != 0) {
      perror("Setting detached attribute failed");
      exit(EXIT_FAILURE);    }
```

```
res = pthread_create(&a_thread, &thread_attr,
thread_function, (void *)message);
   if (res != 0) {
      perror("Thread creation failed");
      exit(EXIT_FAILURE);
   }
   (void)pthread_attr_destroy(&thread_attr);
   while(!thread_finished) {
      printf("Waiting for thread to say it's finished...\n");
      sleep(1);
   }
   printf("Other thread finished, bye!\n");
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   printf("thread_function is running. Argument was
%s\n", (char *)arg);
   sleep(4);
   printf("Second thread setting finished flag, and exiting
now\n");
   thread_finished = 1;
   pthread_exit(NULL);
}
```

Wyjście programu

```
$ ./thread5
Waiting for thread to say it’s finished...
thread_function is running. Argument was Hello World
Waiting for thread to say it’s finished...
Waiting for thread to say it’s finished...
Waiting for thread to say it’s finished...
Second thread setting finished flag, and exiting now
Other thread finished, bye!
```

Komentarz

- W programie została zainicjalizowana zmienna typu pthread_attr_t
- Przed uruchomieniem wątku wykonana została funkcja pthread_attr_setdetachstate(&thread_attr, PTHREAD_CREATE_DETACHED);
- funkcja zmodyfikowała zmienną thread_attr w taki sposób, że, jeśli zostanie utworzony wątek z tą zmienną jako atrybut, to po jego zakończeniu wątek macierzysty nie będzie oczekiwał na wątek potomny

Wielo-wielowątkowa aplikacja

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

#define NUM_THREADS 6

void *thread_function(void *arg);

int main() {
   int res;
   pthread_t a_thread[NUM_THREADS];
   void *thread_result;
   int lots_of_threads;

   for(lots_of_threads = 0; lots_of_threads <
NUM_THREADS; lots_of_threads++) {

      res = pthread_create(&(a_thread[lots_of_threads]),
NULL, thread_function, (void *)&lots_of_threads);
      if (res != 0) {
         perror("Thread creation failed");
         exit(EXIT_FAILURE);
      }
      sleep(1);
   }
```

```
printf("Waiting for threads to finish...\n");
   for(lots_of_threads = NUM_THREADS - 1;
lots_of_threads >= 0; lots_of_threads--) {
      res = pthread_join(a_thread[lots_of_threads],
&thread_result);
      if (res == 0) {
         printf("Picked up a thread\n");
      }
      else {
         perror("pthread_join failed");
      }
   }
   printf("All done\n");
   exit(EXIT_SUCCESS);
}

void *thread_function(void *arg) {
   int my_number = *(int *)arg;
   int rand_num;

   printf("thread_function is running. Argument was
%d\n", my_number);
   rand_num=1+(int)(9.0*rand()/(RAND_MAX+1.0));
   sleep(rand_num);
   printf("Bye from %d\n", my_number);
   pthread_exit(NULL);
}
```

Wyjście programu

```
$ ./thread8
thread_function is running. Argument was 0
thread_function is running. Argument was 1
thread_function is running. Argument was 2
thread_function is running. Argument was 3
thread_function is running. Argument was 4
Bye from 1
thread_function is running. Argument was 5
Waiting for threads to finish...
Bye from 5
Picked up a thread
Bye from 0
Bye from 2
Bye from 3
Bye from 4
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
Picked up a thread
All done
```

Jak to działa?

- W pętli następuje wygenerowanie NUM_THREADS wątków.
- Każdy z wątków ustala losowy czas, kiedy ma zakończyć swoje obliczenia
- Główny program „zbiera” zakończone wątki (uwaga: wątki kończą się w różnej kolejności, ale „zbieranie” odbywa się po kolei zgodnie z numerami wątków zapisanymi w tablicy a_thread)

Interesujące zjawisko

- Interesujące wyniki programu otrzymuje się, gdy zrezygnuje się w nim z wywołań funkcji sleep:
  - okazuje się, że kilka wątków „startuje” z tą samą wartością przekazywaną im jako parametr wywołania
  - problematyczną linią programu jest:
    for(lots_of_threads = 0; lots_of_threads < NUM_THREADS; lots_of_threads++) {res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function, **(void *)&lots_of_threads**); …
  - do wątku przekazywane jest wskazanie na zmienną, pomimo, że zmienna zmienia się w pętli, komórka pamięci, w której powinien być przechowywany jej stan nie jest aktualizowana
  - Rozwiązanie problemu:
    res = pthread_create(&(a_thread[lots_of_threads]), NULL, thread_function, **(void *)lots_of_threads**); …
  - Modyfikacji musi też ulec fragment funkcji obsługującej wątek:

```
void *thread_function(void *arg) {
int my_number = (int)arg; …
```

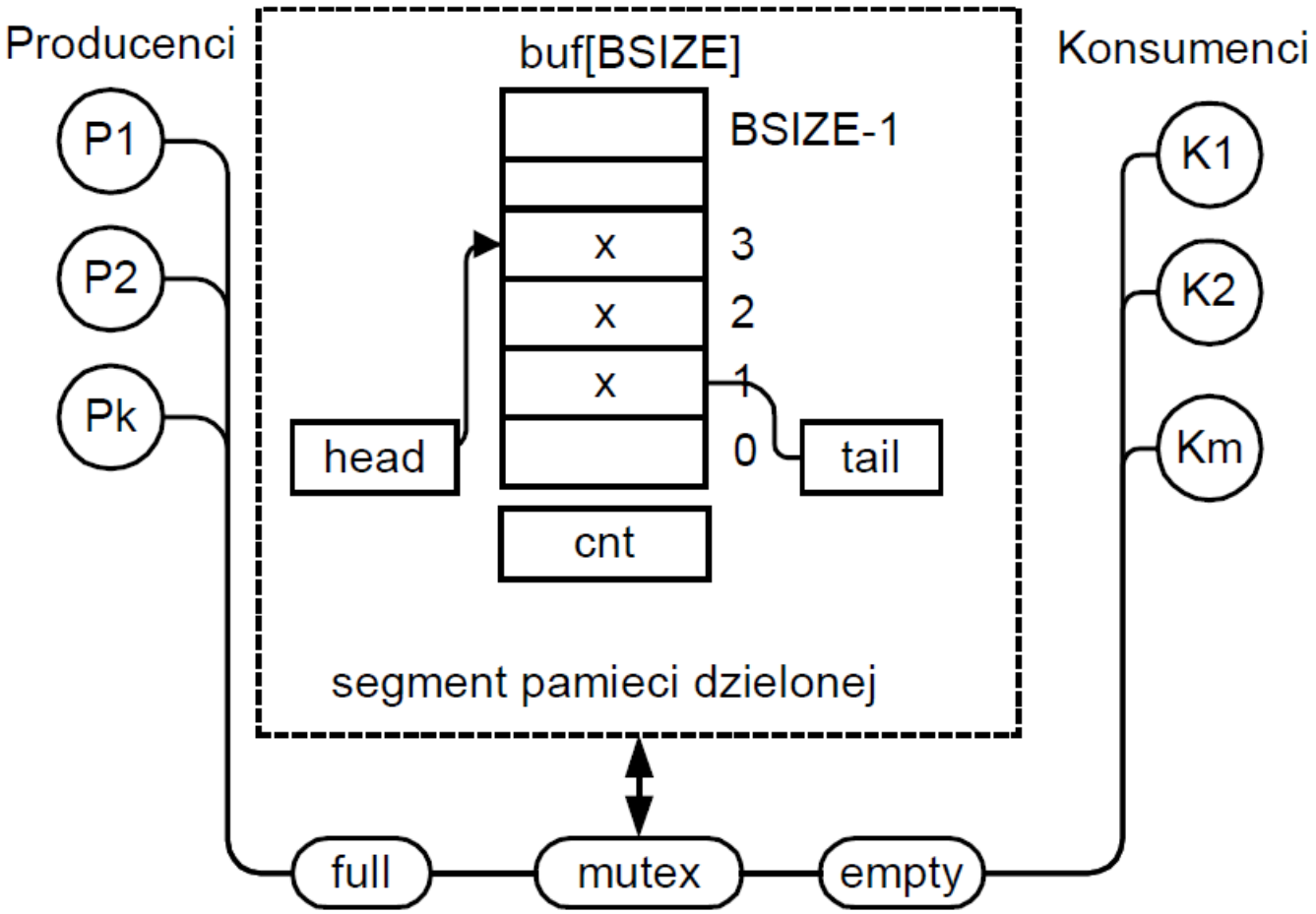
Producenci i konsumenci (1)

- W systemie działa *P>0* procesów, które produkują pewne dane oraz *K>0* procesów, które odbierają dane od producentów. Między producentami a konsumentami może znajdować się bufor o pojemności *B*, którego zadaniem jest równoważenie chwilowych różnic w czasie działania procesów. Procesy produkujące dane będziemy nazywać producentami, a procesy odbierające dane --- konsumentami. Zadanie polega na synchronizacji pracy producentów i konsumentów, tak aby:
  1. Konsument oczekiwał na pobranie danych w sytuacji, gdy bufor jest pusty. Gdybyśmy pozwolili konsumentowi odbierać dane z bufora zawsze, to w sytuacji, gdy nikt jeszcze nic w buforze nie zapisał, odebrana wartość byłaby bezsensowna.
  2. Producent umieszczając dane w buforze nie nadpisywał danych już zapisanych, a jeszcze nie odebranych przez żadnego konsumenta. Wymaga to wstrzymania producenta w sytuacji, gdy w buforze nie ma wolnych miejsc.
  3. Jeśli wielu konsumentów oczekuje, aż w buforze pojawią się jakieś dane oraz ciągle są produkowane nowe dane, to każdy oczekujący konsument w końcu coś z bufora pobierze. Nie zdarzy się tak, że pewien konsument czeka w nieskończoność na pobranie danych, jeśli tylko ciągle napływają one do bufora.
  4. Jeśli wielu producentów oczekuje, aż w buforze będzie wolne miejsce, a konsumenci ciągle coś z bufora pobierają, to każdy oczekujący producent będzie mógł coś włożyć do bufora. Nie zdarzy się tak, że pewien producent czeka w nieskończoność, jeśli tylko ciągle z bufora coś jest pobierane.

Producenci i konsumenci (2)

- Rozpatruje się różne warianty tego problemu:
  1. Bufor może być nieskończony.
  2. Bufor cykliczny może mieć ograniczoną pojemność.
  3. Może w ogóle nie być bufora.
  4. Może być wielu producentów lub jeden.
  5. Może być wielu konsumentów lub jeden.
  6. Dane mogą być produkowane i konsumowane po kilka jednostek na raz.
  7. Dane muszą być odczytywane w kolejności ich zapisu lub nie.
- Problem producentów i konsumentów jest abstrakcją wielu sytuacji występujących w systemach komputerowych, na przykład zapis danych do bufora klawiatury przez sterownik klawiatury i ich odczyt przez system operacyjny.

Rozwiązanie problemu producent-konsument (pamięć dzielona)



Szkic rozwiązania z zastosowaniem semaforów (1)

```
#define BufSize 8 // Bufor ma 8 elementów
RecType Buffer[BufSize]; // Buf. na elementy
semaphore Mutex; // Ochrona bufora
semaphore Puste; // Wolne bufory
semaphore Pelne; // Zajete bufory
int count; // Wskaźnik bufora
producent(void) {
RecType x;
do { ...
produkcja rekordu x;
// Czekaj na wolny bufor
sem_wait(Puste);
sem_wait(Mutex);
// Umieść element x w buforze
Buffer[count] = x; count++;
sem_post(Mutex);
// Pojawił się nowy element
sem_post(Pelne);
} while(1);}
```

```
konsument(void) {
RecType x;
do {
...
...
// Czekaj na element
sem_wait(Pelne);
sem_wait(Mutex);
// Pobierz element x z bufora
count--;
x = Buffer[count];
sem_post(Mutex);
// Zwolnij miejsce w buforze
sem_post(Puste);
konsumpcja rekordu x;
...
} while(1);
}
```

Szkic rozwiązania z zastosowaniem semaforów (1)

```
main(void) {
count = 0;
sem_init(Puste,BufSize);          // Inicjacja semafora Puste
sem_init(Pelne,0);                // Inicjacja semafora Pelne
sem_init(Mutex,1);                // Inicjacja semafora Mutex
pthread_create(…,producent,..);   // Start K wątków producenta
..
pthread_create(…,konsument,..);   // Start L wątków konsumenta
..
}
```

Rozwiązanie – pamięć dzielona (1)

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <pthread.h>
#include <semaphore.h>
#include <assert.h>
#include <sys/stat.h>
#include <sys/file.h>

#define BSIZE 4 // Buffer size
#define LSIZE 80 // LIne length
typedef struct { // Shared data
                char buf[BSIZE][LSIZE];
                int head;
                int tail;
                int cnt;
} bufor_t;
bufor_t shared_memory;
bufor_t* shared_stuff;
#define NUM_PROD 3
pthread_t a_prod_thread[NUM_PROD];
#define NUM_CONS 2
pthread_t a_cons_thread[NUM_CONS];
```

```
void *prod_thread_function(void *arg);
void *cons_thread_function(void *arg);
void *thread_result;
sem_t *mutex, *empty, *full;

int main(){
   int res;    int i;
   shared_stuff = &shared_memory;
   shared_stuff->head=0;
   shared_stuff->tail=0;
   shared_stuff->cnt=0;
   mutex = sem_open("mutex",O_CREAT,S_IRWXU,1);
   empty = sem_open("empty",O_CREAT,S_IRWXU,BSIZE);
   full =  sem_open("full",O_CREAT,S_IRWXU,0);
   printf("Producers are created...\n");
   for(i=0;i<NUM_PROD;i++)
   {   res = pthread_create(&(a_prod_thread[i]), NULL,
                  prod_thread_function, (void *)i);
      if (res != 0) { perror("Thread creation failed");
                  exit(EXIT_FAILURE); } }
   printf("Consumers are created...\n");
   for(i=0;i<NUM_CONS;i++)
   {   res = pthread_create(&(a_cons_thread[i]), NULL,
                  cons_thread_function, (void *)i);
      if (res != 0) { perror("Thread creation failed");
                  exit(EXIT_FAILURE); } }
```

Rozwiązanie – pamięć dzielona (2)

```
    printf("Waiting for producer threads to finish...\n");
    for(i = NUM_PROD - 1; i >= 0; i--) {
        res = pthread_join(a_prod_thread[i], &thread_result);
        if (res == 0) { printf("Picked up a thread\n");}
        else { perror("pthread_join failed");}
    }
    printf("Waiting for consumer threads to finish...\n");
    for(i = NUM_CONS - 1; i >= 0; i--) {
        res = pthread_join(a_prod_thread[i], &thread_result);
        if (res == 0) { printf("Picked up a thread\n"); }
        else { perror("pthread_join failed"); }
    }
    printf("All done\n");
    exit(EXIT_SUCCESS);
}
```

```
void *prod_thread_function(void *arg) {
    int my_number = (int)arg;
    printf("Producer %d is running\n",my_number);
    sleep(1);
    while(1){
        static int j=0;
        sem_wait(empty);
        sem_wait(mutex);
        printf("Producer %d - cnt: %d head: %d tail: %d\n",
                my_number, shared_stuff->cnt,
                shared_stuff->head,shared_stuff->tail);
        sprintf(shared_stuff->buf[shared_stuff->head],
            "Message %d from producer %d",j++,my_number);
        shared_stuff-> cnt ++;
        shared_stuff->head=(shared_stuff->head +1)%BSIZE;
        sem_post(mutex);
        sem_post(full);
        sleep(8);
    }
    printf("Bye from producer %d\n", my_number);
    pthread_exit(NULL);
}
```

Rozwiązanie – pamięć dzielona (3)

```
void *cons_thread_function(void *arg) {
   int my_number = (int)arg;
   printf("Consumer %d is running\n",my_number);
   sleep(2);
   while(1) {
      sem_wait(full);
      sem_wait(mutex);
      printf("Consumer %d - cnt: %d data received: %s\n",
            my_number, shared_stuff->cnt,
            shared_stuff->buf[shared_stuff->tail]);
      shared_stuff-> cnt --;
      shared_stuff->tail = (shared_stuff->tail +1) % BSIZE;
      sem_post(mutex);
      sem_post(empty);
      sleep(1);
   }
   printf("Bye from Consumer %d\n", my_number);
   pthread_exit(NULL);
}
```

__

Dyskusja

- Medium komunikacyjnym jest kolejka zrealizowana we współdzielonym obszarze pamięci
- Zalety:
  - Mniejsze ograniczenia na rozmiar przesyłanych danych
- Wady:
  - Konieczność „samodzielnego” zarządzania prawidłowym dostępem do danych
  - Bardziej złożony kod
- Takie rozwiązanie zmniejsza ograniczenie ze względu na pojemność bufora do wymiany danych. Warto stosować takie rozwiązanie, gdy pomiędzy producentem, a konsumentem przesyłane są większe porcje danych/komunikaty

Producent-konsument kolejki (1)

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <mqueue.h>
#include <sys/stat.h>
#include <sys/wait.h>

struct message {
char mtext[128];
 };

void producer(int prod_no, mqd_t qid){
   int j= 0, r;
   struct message mes;
   while(1){
   sprintf(mes.mtext,"Message %d from producer
%d",j++,prod_no);
   r = mq_send(qid, mes.mtext, strlen(mes.mtext) + 1,
1);
   if (r == -1) { perror("mq_send"); break;}
   sleep(3);
   }
}
```

```
void consumer(int cons_no, mqd_t qid){
   u_int pri;
   struct message msg;
   ssize_t len;
   while(1){
   len = mq_receive(qid, (char *) &msg, sizeof(msg),
&pri);
   if (len == -1) { perror("mq_receive"); break;}
   printf("Consumer %d, data received: %s\n",
            cons_no, msg.mtext);
   }
}
```

Producent-konsument kolejki (2)

```
int main(int argc, char *argv[])
 {
 // Allow up to 10 messages before blocking.
 // Message size is 128 bytes (see above).
 struct mq_attr mattr = {
 .mq_maxmsg = 10,
 .mq_msgsize = sizeof(struct message)
 };
 mqd_t mqid = mq_open("/myq",
 O_CREAT | O_RDWR,
 S_IREAD | S_IWRITE,  &mattr);
 if (mqid == (mqd_t) -1) { perror("mq_open"); exit(1); }
 // Fork a producer process, we'll be the consumer.
 pid_t pid = fork();
 if (pid == 0) {
 producer(0,mqid);
 mq_close(mqid);
 exit(0);
 }
 else {
 consumer(0,mqid);
 mq_close(mqid);
 int status;
 wait(&status);
 }
 mq_unlink("/myq");
 return 0;  }
```

--

Dyskusja

- Jako medium komunikacyjne zastosowano kolejkę
- Zalety:
  - Prostota
  - Automatyczna synchronizacja i wzajemne wykluczanie w dostępie do współdzielonego kanału komunikacyjnego
  - System operacyjny „sam” zarządza dostępem do współdzielonego zasobu
- Wady:
  - Ograniczoność pojemności kolejki
  - W skrajnym wypadku nie będzie możliwe przesłanie danych, lub będzie możliwe przesłanie tylko pojedynczej struktury danych
- Warto stosować takie rozwiązanie, gdy pomiędzy producentem, a konsumentem przesyłane są niewielkie porcje danych/komunikaty