Przegląd technik wzajemnego wykluczania

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

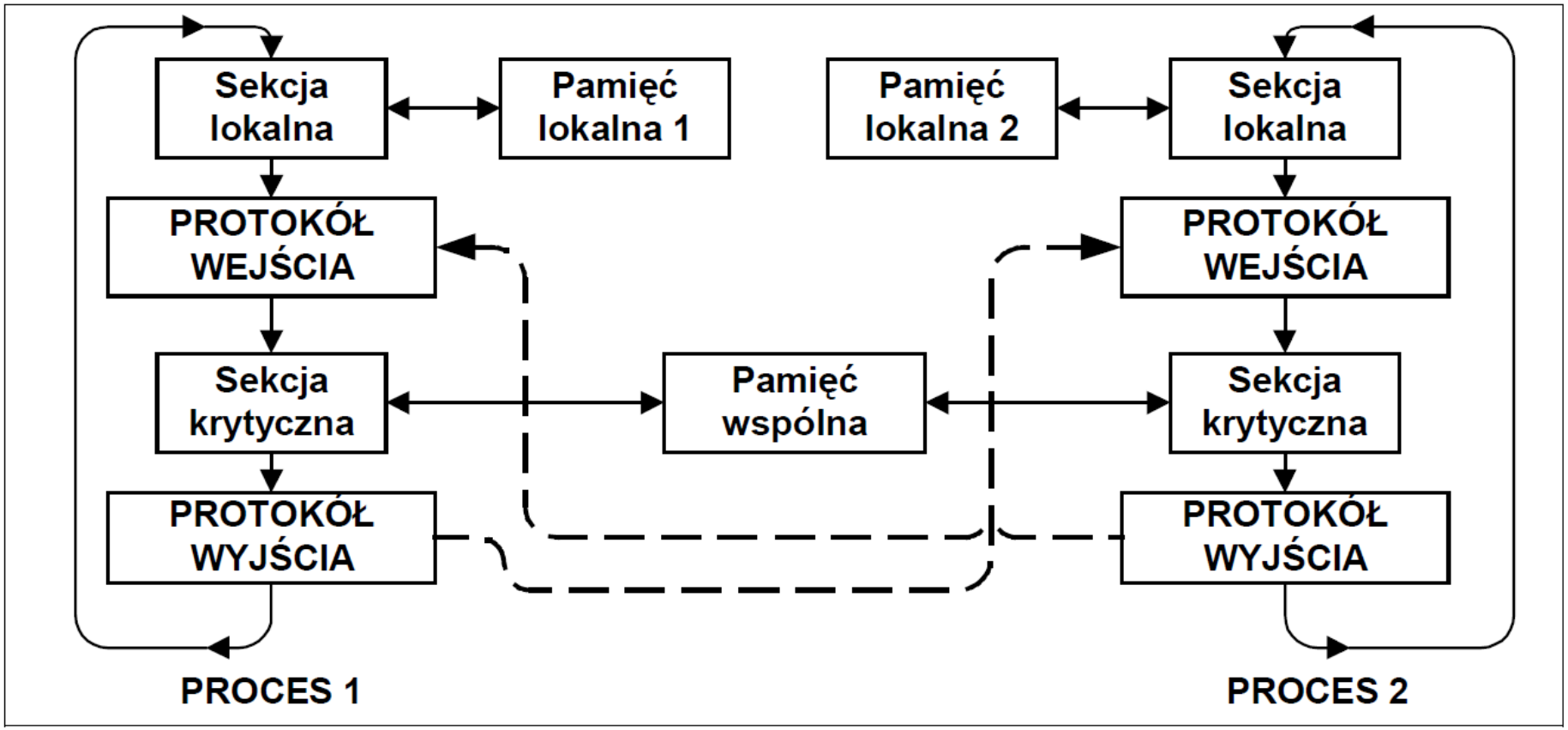
Postawienie problemu

- Poszukujemy dostępnych metod zapewniających wzajemne wykluczanie oraz technik synchronizacji pomiędzy procesami/wątkami.
- Przegląd zostanie dokonany częściowo w oparciu o materiał wprowadzony w poprzednich wykładach

Przypomnienie podstawowych pojęć

- **Operacja atomowa -** sekwencja jednego lub wielu działań elementarnych które nie mogą być przerwane. Wykonuje się w całości albo wcale. Działania pośrednie nie mogą być obserwowane przez inny proces.
- **Wzajemne wykluczanie** - wymaganie aby ciąg operacji na pewnym zasobie (zwykle pamięci) był wykonany w trybie wyłącznym przez tylko jeden z potencjalnie wielu procesów.
- **Sekcja krytyczna** – ciąg operacji na pewnym zasobie (zwykle pamięci) który musi wykonany w trybie wyłącznym przez tylko jeden z potencjalnie wielu procesów.
- Przy wejściu do sekcji proces wykonuje **protokół wejścia,** w którym sprawdza czy może wejść do sekcji krytycznej.
- Po wyjściu z sekcji wykonuje **protokół wyjścia** aby poinformować inne procesy że opuścił już sekcję krytyczną i inny proces może ją zająć.

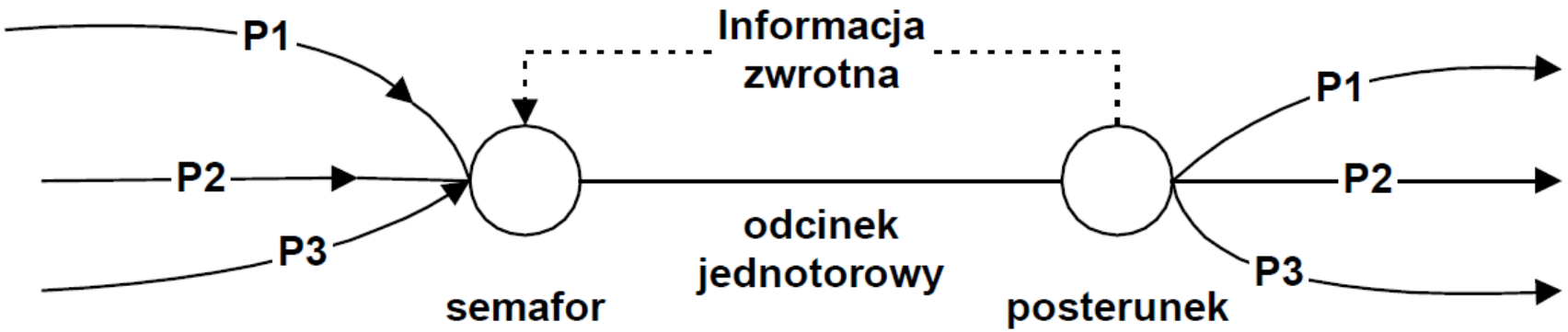
Sekcja lokalna i sekcja krytyczna



Model programowania z sekcją lokalną i sekcją krytyczną

W danej chwili w sekcji krytycznej może przebywać tylko jeden proces.

Odniesienie do ruchu kolejowego…



Na odcinku jednotorowym może przebywać tylko jeden pociąg

Warunki rozwiązania problemu wzajemnego wykluczania

1. W sekcji krytycznej może być tylko jeden proces to znaczy instrukcje z sekcji krytycznej nie mogą być przeplatane.
2. Nie można czynić żadnych założeń co do względnych szybkości wykonywania procesów.
3. Proces może się zatrzymać w sekcji lokalnej nie może natomiast w sekcji krytycznej. Zatrzymanie procesu w sekcji lokalnej nie może blokować innym procesom wejścia do sekcji krytycznej.
4. Każdy z procesów musi w końcu wejść do sekcji krytycznej.

Blokowanie przerwań – założenia

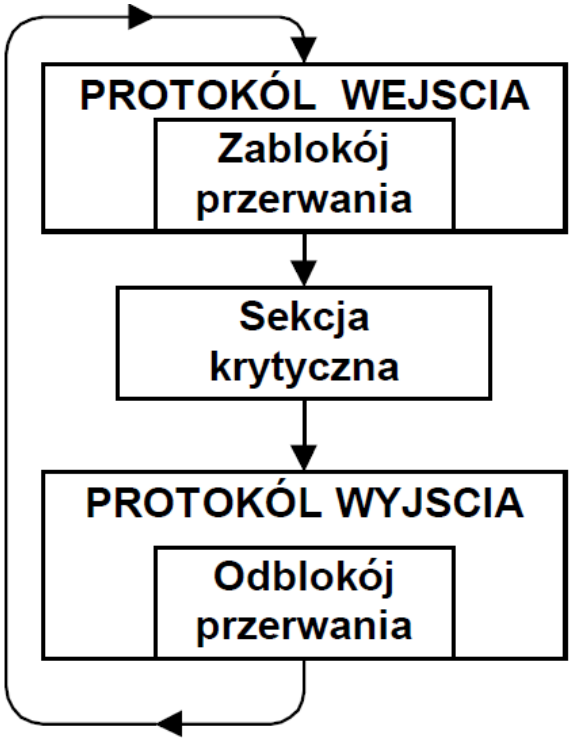
Metoda zapewnienia wzajemnego wykluczania poprzez blokowanie przerwań opiera się na fakcie że proces może być przełączony przez:

1. Przerwanie które aktywuje procedurę szeregującą
2. Wywołanie wprost procedury szeregującej lub innego wywołania systemowego powodującego przełączenie procesów.

Gdy żaden z powyższych czynników nie zachodzi procesy nie mogą być przełączane.

Blokowanie przerwań – realizacja

1 Protokół wejścia do sekcji – następuje zablokowanie przerwań.
2. Protokół wyjścia z sekcji – następuje odblokowanie przerwań.
3. Wewnątrz sekcji krytycznej nie wolno używać wywołań systemowych mogących spowodować przełączenie procesów.



Przykład:
System VxWorks oferuje funkcje:
taskLock()/taskUnlock () – blokujące/odblokowujące przełączanie zadania

intLock()/intUnlock() – blokujące/odblokowujące przerwania

Blokowanie przerwań – wady / zastosowania

1. Przełączanie wszystkich procesów jest zablokowane.
2. System nie reaguje na zdarzenia zewnętrzne co może spowodować utratę danych.
3. Skuteczne w maszynach jednoprocesorowych

<u>Zastosowanie metody:</u>
Wewnątrz systemu operacyjnego do ochrony wewnętrznych sekcji krytycznych.

Metoda zmiennej blokującej - nieprawidłowa

```
int lock = 0;

do {
   sekcja_lokalna;
   // Protokół wejścia
   while(lock != 0) (* czekanie aktywne *);
   lock = 1;
   sekcja_krytyczna;
   lock = 0;  // Protokół wyjścia
} while(1);
```

Metoda jest niepoprawna, gdyż operacja testowania wartości zmiennej lock i ustawiania jej na 1 może być przerwana (nie jest niepodzielna).
Dodatkową wadą metody jest angażowanie procesora w procedurze aktywnego czekania.

Pamięć dzielona IPC – przykład – program 1

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define TEXT_SZ 2048
struct shared_use_st {
   int written_by_you;
   char some_text[TEXT_SZ];};

int main()
{  int running = 1;
   void *shared_memory = (void *)0;
   struct shared_use_st *shared_stuff;
   int shmid;
   srand((unsigned int)getpid());
   shmid = shmget((key_t)1234,
   sizeof(struct shared_use_st), 0666 | IPC_CREAT);
   if (shmid == -1) {  fprintf(stderr, "shmget failed\n");
                       exit(EXIT_FAILURE);
   }
   shared_memory = shmat(shmid, (void *)0, 0);
   if (shared_memory == (void *)-1) {
      fprintf(stderr, "shmat failed\n");
      exit(EXIT_FAILURE);
   }
```

```
printf("Memory attached at %X\n", (int)shared_memory);

shared_stuff = (struct shared_use_st *) shared_memory;
shared_stuff->written_by_you = 0;
 while(running) {
      if (shared_stuff->written_by_you) {
         printf("You wrote: %s", shared_stuff->some_text);
         sleep( rand() % 4 );
         shared_stuff->written_by_you = 0;
         if (strncmp(shared_stuff->some_text,
                  "end", 3) == 0) {
            running = 0;
         }
      }
   }
   if (shmdt(shared_memory) == -1) {
      fprintf(stderr, "shmdt failed\n");
      exit(EXIT_FAILURE);
   }

   if (shmctl(shmid, IPC_RMID, 0) == -1) {
      fprintf(stderr, "shmctl(IPC_RMID) failed\n");
      exit(EXIT_FAILURE);
   }

   exit(EXIT_SUCCESS);
}
```

Pamięć dzielona IPC –przykład - program 2

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define TEXT_SZ 2048
struct shared_use_st {
   int written_by_you;
   char some_text[TEXT_SZ];};
int main()
{
   int running = 1;
   void *shared_memory = (void *)0;
   struct shared_use_st *shared_stuff;
   char buffer[BUFSIZ];
   int shmid;
   shmid = shmget((key_t)1234,
       sizeof(struct shared_use_st), 0666 | IPC_CREAT);
   if (shmid == -1) {
      fprintf(stderr, "shmget failed\n");
      exit(EXIT_FAILURE);    }
   shared_memory = shmat(shmid, (void *)0, 0);
   if (shared_memory == (void *)-1) {
      fprintf(stderr, "shmat failed\n");
      exit(EXIT_FAILURE);    }
   printf("Memory attached at %X\n",
(int)shared_memory);

shared_stuff = (struct shared_use_st *)
                                shared_memory;
   while(running) {
      while(shared_stuff->written_by_you == 1) {
         sleep(1);
         printf("waiting for client...\n");
      }
      printf("Enter some text: ");
      fgets(buffer, BUFSIZ, stdin);

      strncpy(shared_stuff->some_text, buffer, TEXT_SZ);
      shared_stuff->written_by_you = 1;

      if (strncmp(buffer, "end", 3) == 0) {
            running = 0;
      }
   }
   if (shmdt(shared_memory) == -1) {
      fprintf(stderr, "shmdt failed\n");
      exit(EXIT_FAILURE);
   }
   exit(EXIT_SUCCESS);
}
```

Wirujące blokady

Wirujące blokady są środkiem zabezpieczania sekcji krytycznej. Wykorzystują jednak czekanie aktywne zamiast przełączenia kontekstu wątku tak jak się to dzieje w muteksach.

Inicjacja wirującej blokady

`int pthread_spin_init( pthread spinlock t *blokada, int pshared)`

`blokada -` Identyfikator wirującej blokady **`pthread_spinlock_t`**

`pshared -`

- **`PTHREAD_PROCESS_SHARED`** – na blokadzie mogą operować wątki należące do różnych procesów
- **`PTHREAD_PROCESS_PRIVATE`** – na blokadzie mogą operować tylko wątki należące do tego samego procesu

Funkcja inicjuje zasoby potrzebne wirującej blokadzie. Każdy proces, który może sięgnąć do zmiennej identyfikującej blokadę może jej używać. Blokada może być w dwóch stanach:

· Wolna
· Zajęta

Zajęcie blokady

`int pthread_spin_lock( pthread spinlock t * blokada)`
`blokada` Identyfikator wirującej blokady – zmienna typu **`pthread_spinlock_t`**
Działanie funkcji zależy od stanu blokady. Gdy blokada jest wolna następuje jej zajęcie. Gdy blokada jest zajęta wątek wykonujący funkcję **`pthread_spin_lock(...)`** ulega zablokowaniu do czasu gdy inny wątek nie zwolni blokady wykonując funkcję **`pthread_spin_unlock(...)`**.

Próba zajęcia blokady

`int pthread_spin_trylock( pthread spinlock t * blokada)`
`blokada` Identyfikator wirującej blokady – zmienna typu **`pthread_spinlock_t`**
Działanie funkcji zależy od stanu blokady. Gdy blokada jest wolna następuje jej zajęcie – funkcja zwraca wartość **`EOK.`** Gdy blokada jest zajęta następuje natychmiastowy powrót i funkcja zwraca stałą **`EBUSY`**.

Zwolnienie blokady

`int pthread_spin_unlock( pthread spinlock t * blokada)`
`blokada` Identyfikator wirującej blokady – zmienna typu **`pthread_spinlock_t`**
Działanie funkcji zależy od stanu blokady. Gdy są wątki czekające na zajęcie blokady to jeden z nich zajmie blokadę. Gdy żaden wątek nie czeka na zajęcie blokady będzie ona zwolniona.

Skasowanie blokady

`int pthread_spin_destroy( pthread spinlock t * blokada)`
`blokada` Identyfikator wirującej blokady – zmienna typu **`pthread_spinlock_t`**
Funkcja zwalnia blokadę i zajmowane przez nią zasoby.

Przykład programu stosującego wirującą blokadę

```
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
void *thread_function(void *arg);
pthread_spinlock_t *blokada;
int main() {
   int res;   pthread_t a_thread;   void *thread_result;
   blokada = (pthread_spinlock_t *)
                malloc(sizeof(pthread_spinlock_t));
   res =
pthread_spin_init(blokada,PTHREAD_PROCESS_SHARED);
   if (res != 0) {  perror("Spinlock 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(1)
   {   pthread_spin_lock(blokada);
      printf("Spin lock taken by thread 1...\n");
      sleep(5);
      pthread_spin_unlock(blokada);
      printf("Spin lock released by thread 1...\n");
      sleep(3);    }
 pthread_spin_destroy(blokada);
   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 res;
   sleep(1);
   while(1)
   {  res = pthread_spin_trylock(blokada);
      if(res != 0)
      {  printf("Spin lock busy...\n");
         sleep(1); }
      else
      {   printf("Spin lock taken by thread 2...\n");
         sleep(1);
         pthread_spin_unlock(blokada);
         printf("Spin lock released by thread 2...\n");
         sleep(1); }
   }
   pthread_exit(0); }
```

Niesystemowe metody zapewniania wzajemnego wykluczania - podsumowanie

Niesystemowe metody stosowane są rzadko i ich znaczenie jest raczej teoretyczne.

Powody:

1. Prawie zawsze tworzymy aplikacje działające w środowisku systemu operacyjnego który z reguły dostarcza mechanizmów zapewnienia wzajemnego wykluczania.
2. Realizacja metod wzajemnego wykluczania polega na zawieszeniu pewnych procesów a wznowieniu innych. System operacyjny w naturalny sposób zapewnia takie mechanizmy. Proces zawieszony nie wykonuje czekania aktywnego a zatem nie zużywa czasu procesora.
3. Metody systemowe są znacznie prostsze i powiązane z innymi mechanizmami i zabezpieczeniami. Przykładowo awaryjne zakończenie się procesu w sekcji krytycznej odblokowuje tę sekcję. Można też narzucić maksymalny limit czasowy oczekiwania na wejście do sekcji krytycznej (*ang. Timeout*).

Systemowe metody wzajemnego wykluczania

Systemowe metody zapewnienia wzajemnego wykluczania – przykłady:

- Semafory POSIX,
- Muteksy POSIX,
- Monitory

Z niesystemowych metod wzajemnego wykluczania praktycznie stosowane są metody:

1. Wirujące blokady (*ang. Spin Locks*) wykorzystujące sprzętowe wsparcie w postaci instrukcji sprawdź i przypisz oraz zamień. Stosuje się je do synchronizacji wątków ze względu na mały narzut operacji systemowych.
2. Blokowanie przerwań – do ochrony wewnętrznych sekcji krytycznych systemu operacyjnego.

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

Przykład – ochrona współdzielonych danych z zastosowaniem mutex'a

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

void *thread_function(void *arg);
pthread_mutex_t work_mutex;
#define WORK_SIZE 1024
char work_area[WORK_SIZE]; int time_to_exit = 0;

int main() {
   int res;   pthread_t a_thread;   void *thread_result;
   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);}
   pthread_mutex_lock(&work_mutex);
   printf("Input some text. Enter 'end' to finish\n");
   while(!time_to_exit) {
      fgets(work_area, WORK_SIZE, stdin);
      pthread_mutex_unlock(&work_mutex);
```

```
while(1) {
        pthread_mutex_lock(&work_mutex);
        if (work_area[0] != '\0') {
          pthread_mutex_unlock(&work_mutex);
          sleep(1);   } else {   break;    }    }    }
   pthread_mutex_unlock(&work_mutex);
   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");
   pthread_mutex_destroy(&work_mutex);
   exit(EXIT_SUCCESS);}

void *thread_function(void *arg) { sleep(1);
   pthread_mutex_lock(&work_mutex);
   while(strncmp("end", work_area, 3) != 0) {
    printf("You input %d characters\n", strlen(work_area) -
1);   work_area[0] = '\0';
      pthread_mutex_unlock(&work_mutex); sleep(1);
      pthread_mutex_lock(&work_mutex);
      while (work_area[0] == '\0' ) {
         pthread_mutex_unlock(&work_mutex); sleep(1);
         pthread_mutex_lock(&work_mutex); } }
   time_to_exit = 1;    work_area[0] = '\0';
   pthread_mutex_unlock(&work_mutex);
   pthread_exit(0);}
```

Wyjście programu

```
Input some text. Enter ‘end’ to finish
Whit
You input 4 characters
The Crow Road
You input 13 characters
end
Waiting for thread to finish...
Thread joined
```

Jak to działa? (1)

- W obszarze zmiennych globalnych zadeklarowano nowe zmienne work_mutex i time_to_exit
- W głównej funkcji programu zainicjalizowano mutex
- Powołano nowy wątek który:
  - próbuje zamknąć mutex (jeśli jest zamknięty, to oczekuje na jego otwarcie),
  - sprawdza, czy spełniony jest warunek zakończenia wątku, jeśli tak, ustawia zmienną time_to_exit na 1 i pierwszy element tablicy work_area na 0, otwiera mutex
  - jeśli nie, to obliczana jest długość tekstu i ustawiana pierwszy element tablicy work_area na 0, następuje też otwarcie mutex'a (ustawienie pierwszego elementu tablicy na 0 oznacza, że wątek zakończył swoje przetwarzanie współdzielonej zmiennej)
  - po odczekaniu 1 sekundy wątek próbuje zamknąć mutex
  - jeśli zamknięcie się powiedzie, to cyklicznie sprawdza, czy pierwszy element tablicy jest w dalszym ciągu 0, zwalnia mutex, odczekuje 1 sekundę, próbuje zamknąć mutex
  - jeśli zawartość tablicy ulegnie zmianie, to pętla sprawdzająca jest przerywana, mutex pozostaje zamknięty

Jak to działa? (2)

- W wątku macierzystym :
  - następuje próba zamknięcia mutex'a
  - kiedy zamknięcie się powiedzie w pętli następuje odczytywanie tekstów wprowadzanych przez użytkownika (tekst „end" kończy działanie programu) i odblokowanie mutexa
  - w wewnętrznej pętli następuje sprawdzenie, czy tekst nie został przetworzony, sprawdzenie następuje z zachowaniem wzajemnego wykluczania w dostępie do współdzielonej zmiennej (próba zamknięcia a potem otwarcie mutex'a)
  - po wykryciu odebrania danych przez wątek przetwarzający następuje ponownie próba przjęcia dostępu do danych współdzielonych i wpisanie do nich nowego tekstu
  - po wpisaniu do tablicy współdzielonej tekstu „end" następuje zamknięcie wątku macierzystego i potomnego.
- Uwagi:
  - Oba wątki stosują swego rodzaju pooling do wykrywania zmiany stanu zmiennej dzielonej, w profesjonalnym rozwiązaniu należy zastosować semafor lub zmienną warunkową (nowość) do synchronizacji

Monitory

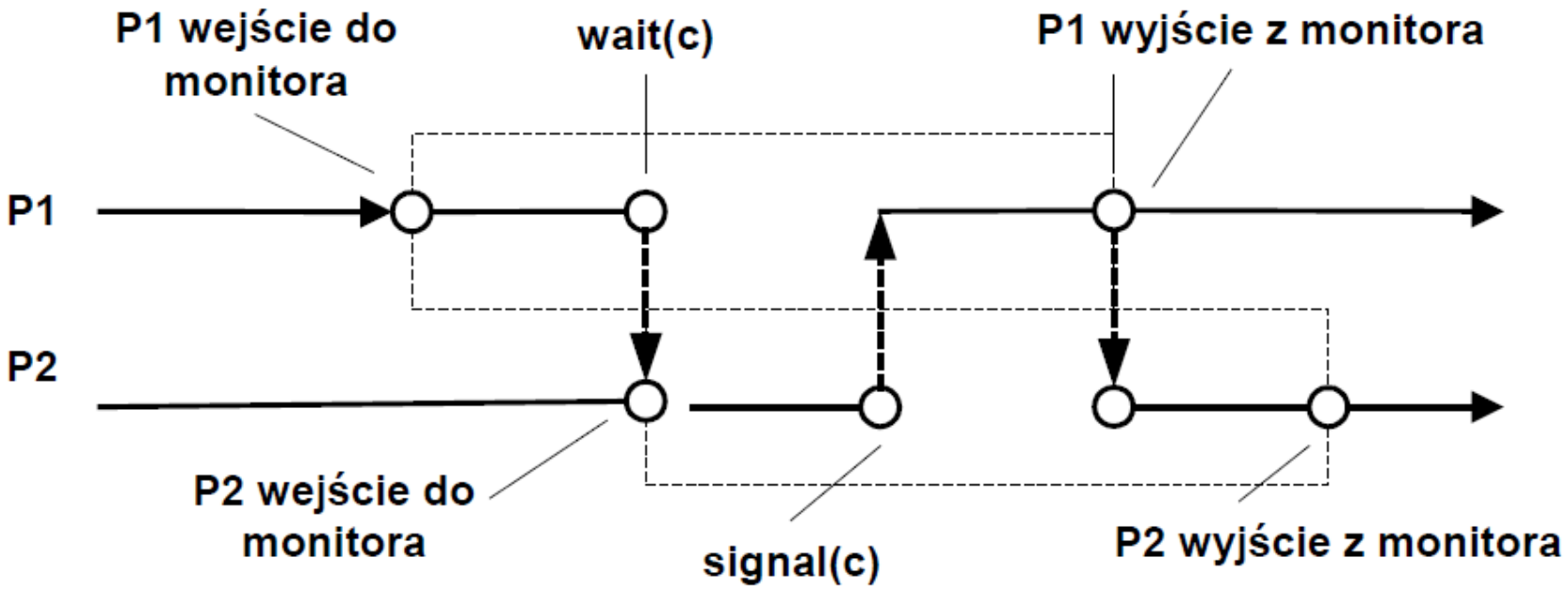
- **Semafor** nie jest mechanizmem strukturalnym. Aplikacje pisane z użyciem semaforów są podatne na błędy. Np. brak operacji sem_post blokuje aplikację.
- Monitor (Brinch Hansen, Hoare) jest strukturalnym narzędziem synchronizacji.
- Monitory zaimplementowane w językach: Java, Modula-3, Concurrent Pascal, Concurrent Euclid.

Definicja monitora

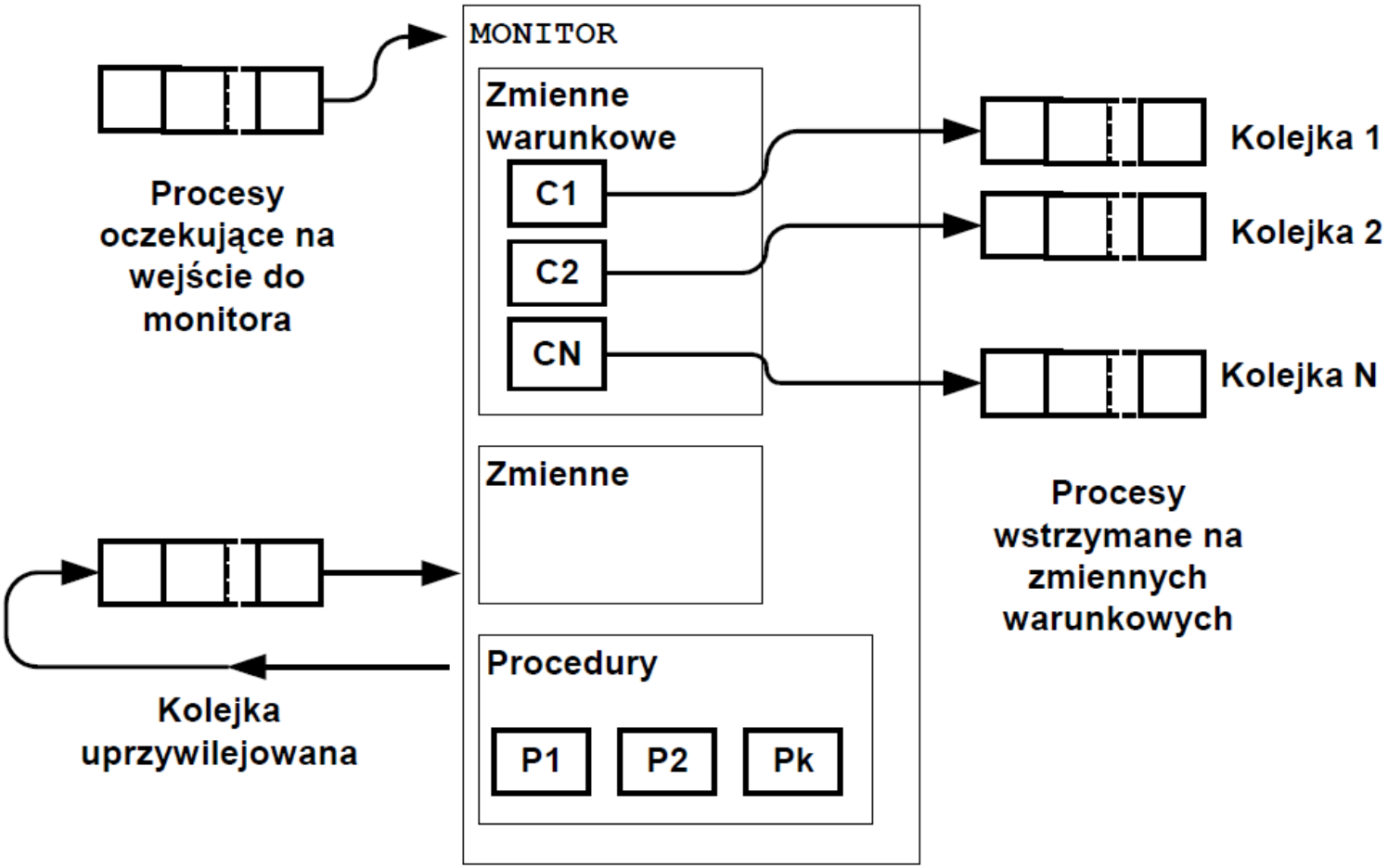
1. Zmienne i działające na nich procedury zebrane są w jednym module. Dostęp do zmiennych monitora możliwy jest tylko za pomocą procedur monitora.
2. W danej chwili tylko jeden proces wykonywać może procedury monitora. Gdy inny proces wywoła procedurę monitora będzie on zablokowany do chwili opuszczenia monitora przez pierwszy proces.
3. Istnieje możliwość wstrzymania i wznowienia procedur monitora za pomocą zmiennych warunkowych (*ang. conditional variables*). Na zmiennych warunkowych można wykonywać operacje **Wait** i **Signal**.

- **Wait(c)** - Wstrzymanie procesu bieżącego wykonującego procedurę monitora i wstawienie go na koniec kolejki związanej ze zmienną warunkową c. Jeżeli jakieś procesy czekają na wejście do monitora to jeden z nich będzie wpuszczony.
- **Signal(c)** – Odblokowanie jednego z procesów czekających na zmiennej warunkowej c. Gdy brak czekających procesów operacja nie daje efektów. Operacja **Signal** jest bezpamięciowa (nie posiada licznika).
- **Notempty(c)** – Funkcja zwraca true gdy kolejka c jest niepusta, false gdy pusta.

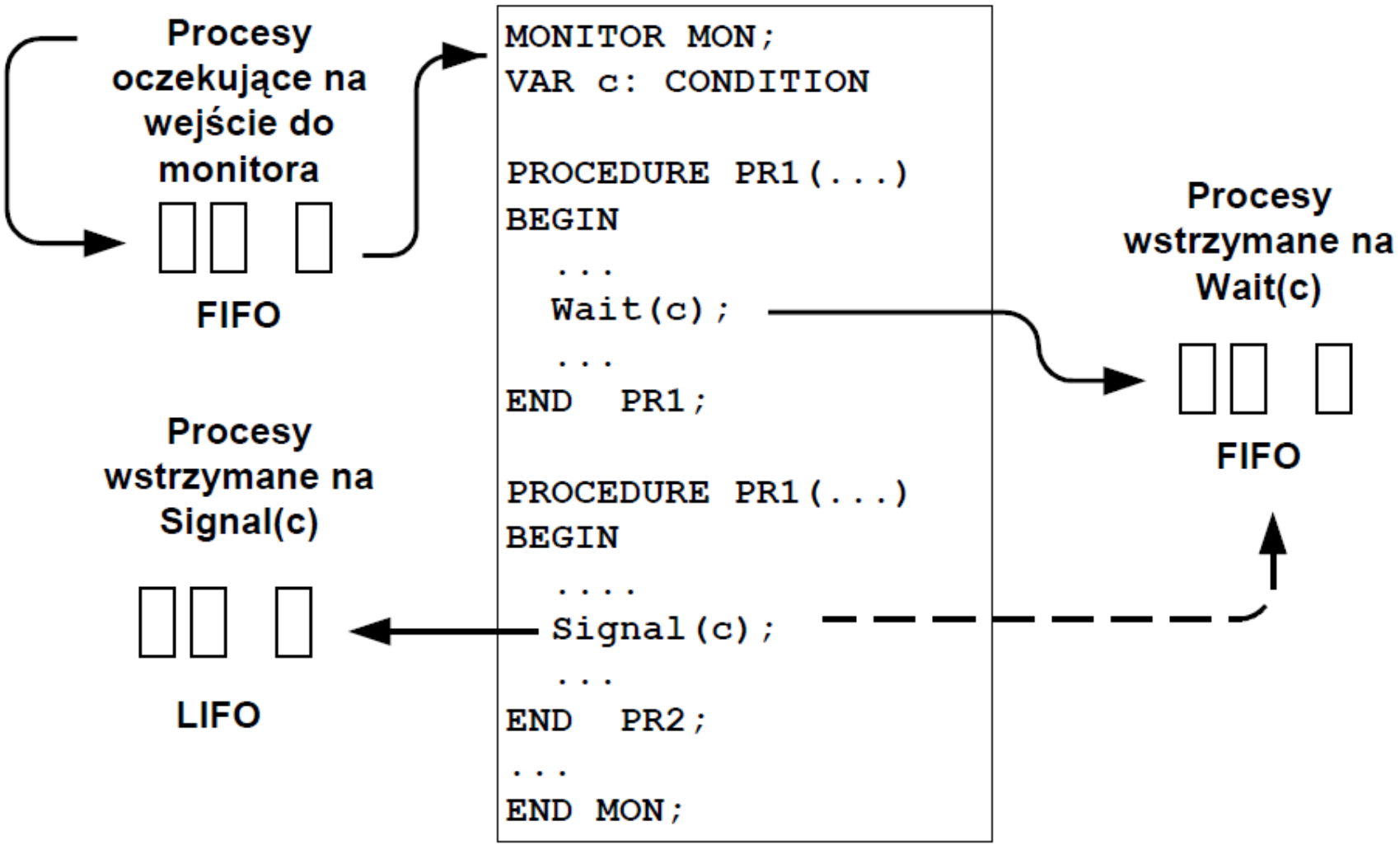
Operacje Wait/Signal



Kolejki monitora



Przykład monitora



Rozwiązanie problemu wzajemnego wykluczania

```
monitor WzajemneWykluczanie
int z1,z2;

Pisz(int x1, int x2){
  z1 = x1;
  z2 = x2;
};

Czytaj(int *x1, int *x2){
 *x1 := z1;
 *x2 := z2;
};
```

Procedury Czytaj lub pisz wykonane będą w sposób wyłączny co wynika z definicji monitora.

Rozwiązanie problemu producent - konsument

```
monitor ProducentKonsument
#define N 8
Record Buf[N];    // Bufor na rekordy
int Ile;          // Ile rekordów w buforze
int Inp,Out;      // Indeks wejściowy i wyjściowy
CONDITION Prod;   // Czekający producenci
CONDITION Cons;     // Czekający konsumenci

Wstaw(Record x ) {
  if(Ile == N) {
    Wait(Prod);
  }
  Inp := (Inp+1) % N;
  Buf(Inp) = x;
  Ile ++;
  Signal(Cons);
 }
};

Pobierz(Record *x ) {
  if(Ile == 0) {
    Wait(Cons);
  }
  Out := (Out+1) % N;
  *x = Buf(Inp)x;
  Ile --;
  Signal(Prod);
 }
};
```

...

POSIX a monitory

- Standard POSIX dostarcza mechanizmów zgodnych z działaniem monitorów, chociaż nie oferuje wysokopoziomowych funkcji do tworzenia monitorów na wzór ich definicji.
- Stworzenie mechanizmu monitora w POSIX opiera się na odpowiednim zastosowaniu zmiennych warunkowych i mutexów
- Do rozwiązywania problemu wzajemnego wykluczania standard POSIX sugeruje zastosowanie mutexów
- Do eleganckiego rozwiązania problemów czytelników i pisarzy oraz producenta konsumenta można zastosować kombinację mutexów i zmiennych warunkowych odzwierciedlających zasadę działania monitorów.
- Otrzymuje się wtedy możliwość zawieszenia wykonywania sekcji krytycznej do chwili spełnienia określonego warunku.

Zmienna warunkowa - przypomnienie

- Zmienne warunkowe dostarczają nowego mechanizmu synchronizacji pomiędzy wątkami. Podczas gdy muteksy implementują synchronizację na poziomie dostępu do współdzielonych danych, zmienne warunkowe pozwalają na synchronizację na podstawie stanu pewnej zmiennej.
- Bez zmiennych warunkowych wątki musiałyby cyklicznie monitorować stan zmiennej (w sekcji krytycznej), aby sprawdzić, czy osiągnęła ona ustaloną wartość. Podejście takie jest z założenia „zasobożerne”. Zmienna warunkowa pozwala na osiągnięcie podobnego efektu bez „odpytywania”.
- Zmienną warunkową stosuje się zawsze wewnątrz sekcji krytycznej w powiązaniu z zamknięciem muteksu (mutex lock).

Funkcje obsługujące zmienną warunkową

pthread_cond_init	Inicjacja zmiennej warunkowej.
pthread_cond_wait	Czekanie na zmiennej warunkowej
pthread_cond_timedwait	Ograniczone czasowo czekanie na zmiennej warunkowej
pthread_cond_signal	Wznowienie wątku zawieszonego w kolejce danej zmiennej warunkowej
pthread_cond_brodcast	Wznowienie wszystkich wątków zawieszonych w kolejce danej zmiennej warunkowej.
pthread_cond_destroy	Skasowanie zmiennej warunkowej i zwolnienie jej zasobów

Producent- konsument z zastosowaniem „monitorów”

```
#include <stdio.h>
#include <pthread.h>
#define N 4
int pocz ,kon,licznik = 0;
int bufor[N];
pthread_mutex_t mutex;
pthread_cond_t puste, pelne;

void* producent( void* arg ) {
int num = 0;
int cnt = 1;
num = (int) arg;
printf("Start producent: %d\n",num);
while( 1 ) {
pthread_mutex_lock( &mutex );
  while(licznik >= N)
      pthread_cond_wait(&puste,&mutex);
bufor[kon] = cnt++;
kon = (kon+1) %N;
licznik++;
pthread_cond_signal(&pelne);
pthread_mutex_unlock( &mutex );
printf("Prod%d kon: %d wst: %d\n", num, kon,cnt,
licznik );
sleep( 1 );
}
}
```

```
void* konsument( void* arg ) {
int num,x = 0;
num = (int) arg;
printf("Start konsument: %d\n",num);
while( 1 ) {
pthread_mutex_lock( &mutex );
while(licznik <=0 )
pthread_cond_wait(&pelne,&mutex);
x = bufor[pocz];
pocz = (pocz+1) %N;
licznik--;
pthread_cond_signal(&puste);
pthread_mutex_unlock( &mutex );
printf("Kons%d pocz: %d pobral: %d licz: %d\n",
num, pocz, x,licznik );
sleep( 1 );
}}
int main( void ) {
pthread_t p1,p2,k1,k2;
pthread_mutex_init(&mutex,NULL);
pthread_cond_init(&puste,NULL);
pthread_cond_init(&pelne,NULL);
pthread_create(&p1, NULL, &producent, (void *)1 );
pthread_create(&k1, NULL, &konsument, (void *)1 );
pthread_join(p1,NULL);
pthread_join(k1,NULL);
return 0;}
```

Uwagi

- Zamknięcie funkcji producent() i konsument() w osobnej klasie lub pliku zbliżyłoby podany wyżej program do definicji monitora
- Zastosowanie i ograniczenia monitorów w języku Java było wyjaśniane na wykładach dotyczących technik programowania współbieżnego w języku Java
- Za najbardziej zaawansowane techniki zarządzania wzajemnym wykluczaniem uznaje się **obiekty chronione** zaimplementowane między innymi w języku Ada.