

Synchronizacja cz. III

Inne operacje semaforowe ⁽¹⁾

```

1. lock W:
2.   L: if W = 1 then go to L
3.   else W := 1;

4. unlock W:
5.   W := 0;

6. ENQ(r):
7.   if inuse[r] then // resource r is used
8.     begin
9.       Insert p on r-queue;
10.      Block p
11.    end // queue associated with r
12.   else
13.     inuse[r] := True;

14. DEQ(r):
15.   p := Remove from r-queue
16.   if p ≠ Ω
17.     then Activate p // p = Ω means that queue was empty
18.   else inuse[r] := False;
  
```



(c) Zakład Systemów Informatycznych

Slajd 5

Implementacja operacji P i V

```

1. program PV_IMPLEMENTATION;
2.   var active, delay: BOOLEAN;
3.   var NS: INTEGER;
4. procedure IMPLEMENTATION;
5.   var pActive: BOOLEAN;
6. begin
7.   Disable interrupts;
8.   pActive := True;
9.   while pActive do
10.    testandset(pActive, active);
11.    NS := NS - 1;
12.    if NS ≥ 0 then
13.      begin
14.        S := S - 1;
15.        active := False;
16.        Enable interrupts;
17.      end
18.    else
19.      begin
20.        Block process invoking P(S);
21.        p := Remove from RL;
22.        active := False;
23.        Transfer control to p with
24.        Enable interrupts;
25.      end
26.    end;
  
```

```

1. procedure IMPLEMENTATION;
2.   var vActive: BOOLEAN;
3. begin
4.   Disable interrupts;
5.   vActive := True;
6.   while vActive do
7.     testandset(vActive, active);
8.     NS := NS + 1;
9.     if NS > 0 then
10.      S := S + 1;
11.    else
12.      begin
13.        p := remove from LS;
14.        Add p to RL;
15.      end;
16.      active := False;
17.      Enable interrupts;
18.    end;
  
```

Legenda:
 • LS – List associated with S
 • RL – Ready List



(c) Zakład Systemów Informatycznych

Slajd 3

Inne operacje semaforowe ⁽²⁾

```

1. WAIT(e):
2.   if ¬ posted[e] then // only one process can wait for event e
3.     begin
4.       wait[e] := True;
5.       process[e] := p;
6.       Block p;
7.     end
8.   else posted[e] := False;

9. POST(e):
10.  if ¬ posted[e] then
11.    begin
12.      posted[e] := True;
13.      if wait[e] then
14.        begin
15.          wait[e] := False;
16.          posted[e] := False;
17.          Activate process[e];
18.        end;
19.    end;
  
```



(c) Zakład Systemów Informatycznych

Slajd 6

Implementacja operacji wait i signal

```

1. type SEMAPHORE = record
2.   value: INTEGER;
3.   L: list of process;
4. end;
  
```

Implementacja operacji wait(S) = P(S):

```

5. procedure WAIT(S);
6. begin
7.   S.value := S.value - 1;
8.   if S.value < 0 then
9.     begin
10.      add this process ID to S.L;
11.      block this process;
12.    end;
13. end;
  
```

Implementacja operacji signal(S) = V(S):

```

14. procedure SIGNAL(S);
15. begin
16.   S.value := S.value + 1;
17.   if S.value ≤ 0 then
18.     begin
19.       remove a process P from S.L;
20.       wakeup(P);
21.     end;
22. end;
  
```



(c) Zakład Systemów Informatycznych

Slajd 4

Inne operacje semaforowe ⁽³⁾

```

1. Block(i):
2.   if ¬ wws[i] // wait for Wakeup flag associated with process i
3.     then Block process i
4.   else wws[i] := False;

5. Wakeup(i):
6.   if ready(i) // process is ready
7.     then wws[i] := True
8.   else Activate process i;
  
```



(c) Zakład Systemów Informatycznych

Slajd 7

Event counters

Three operations are defined on a event counter E :

- ❖ $\text{read}(E)$ – return the current value of E .
- ❖ $\text{advance}(E)$ – automatically increment E by 1.
- ❖ $\text{await}(E, v)$ – wait until E has a value of v or more.

Regiony krytyczne - implementacja

Dla każdej deklaracji

```
var v: shared T ;
```

Kompilator generuje semafor $v\text{-mutex}$ z wartością początkową 1.

Dla każdej instrukcji

```
region v do S ;
```

Kompilator generuje następujący kod:

```
wait(v-mutex) ;
S ;
signal(v-mutex) ;
```



Producer-consumer problem using event counters

```
1. #include "prototypes.h"
2. #define N 100 // number of slots in the buffer
3. typedef int EVENT_COUNTER; // event counters are a special kind of int
4. EVENT_COUNTER in=0; // counts items inserted into buffer
5. EVENT_COUNTER out=0; // counts items removed from buffer

6. void PRODUCER(void) {
7.     int item, sequence=0;
8.     while(True) // infinite loop
9.     {
10.        produce_item(&item); // generate something to put in buffer
11.        sequence=sequence+1; // count items produced so far
12.        await(out, sequence-N); // wait until there is room in buffer
13.        enter_item(item); // put item in slot (sequence-1) % N
14.        advance(&in); // let consumer know about another item
15.    }

16. void CONSUMER(void) {
17.     int item, sequence=0;
18.     while(True) // infinite loop
19.     {
20.        sequence=sequence+1; // number of item to remove from buffer
21.        await(in, sequence); // wait until required item is present
22.        remove_item(&item); // take item from slot (sequence-1)%N
23.        advance(&out); // let producer know that item is gone
24.        consume_item(item); // do something with the item
25.    }
26. }
```



Warunkowy region krytyczny ⁽¹⁾

Następująca instrukcja jest instrukcją warunkowego regionu krytycznego

```
region v when B do S ;
```

w której B jest wyrażeniem boolowskim. Jak poprzednio, regiony odwołujące się do tych samych zmiennych dzielonych wykluczają się wzajemnie w czasie. Obecnie jednak, kiedy proces wchodzi do regionu sekcji krytycznej, wtedy następuje obliczenie wyrażenia boolowskiego B . Jeśli wyrażenie jest prawdziwe, to instrukcja S będzie wykonana. Jeśli jest fałszywe, to proces nie ubiega się o wyłączny dostęp i ulega opóźnieniu do czasu, aż wyrażenie B stanie się prawdziwe oraz żaden inny proces nie będzie przebywał w regionie związanym ze zmienną v .



Regiony krytyczne - definicja

Niech następująca deklaracja zmiennej v typu T określa zmienną dzieloną przez wiele procesów.

```
var v: shared T;
```

Zmienna v będzie dostępna tylko w obrębie instrukcji *region* o następującej postaci:

```
region v do S;
```



Warunkowy region krytyczny ⁽²⁾

```
1. var buffer: shared record
2.     pool: array [0..n - 1] of ITEM;
3.     count, in, out: INTEGER;
4. end;
```

Proces produkujący umieszcza nową jednostkę *nextp* w buforze dzielonym wykonując instrukcję:

```
5. region buffer when count < n
6. do begin
7.     pool[in]:=nextp;
8.     in:=(in + 1) mod n;
9.     count:=count + 1;
10. end;
```

Proces konsumujący usuwa jednostkę z bufora dzielonego i zapamiętuje ją w *nextk* za pomocą instrukcji:

```
11. region buffer when count > 0
12. do begin
13.     nextk:=pool[out];
14.     out:=(out + 1) mod n;
15.     count:=count - 1;
16. end;
```



Warunkowe regiony krytyczne - implementacja

```

1. region v when B do S;
2. var xMutex, xDelay : SEMAPHORE;
3.   xCount, xTemp : INTEGER;
   // xCount - the number of processes waiting for xDelay
   // xTemp - the number of processes that have been allowed
   // to test their Boolean condition during one trace

4. wait(xMutex);
5. if not B then
6.   begin
7.     xCount:=xCount + 1;
8.     signal(xMutex);
9.     wait(xDelay);
10.    while not B do
11.      begin
12.        xTemp:=xTemp + 1;
13.        if xTemp < xCount then
14.          signal(xDelay)
15.        else
16.          signal(xMutex);
17.          wait(xDelay);
18.        end;
19.        xCount:=xCount - 1;
20.      end;
21.    end;
22. end;
23. S;
24. if x_count > 0 then
25.   begin
26.     x-temp:=0;
27.     signal(x-delay);
28.   end;
29. else
30.   signal(x-mutex);
31. end;

```

(c) Zakład Systemów Informatycznych

Slajd 14

Rozwiązanie problemu pisarzy i czytelników z użyciem semaforów

```

1. shared var
2.   nReaders : INTEGER;
3.   mutex, wmutex, srmutex : SEMAPHORE;

4. procedure READER;
5.   begin
6.     P(mutex);
7.     if nReaders=0 then
8.       begin
9.         nReaders:=nReaders + 1;
10.        P(wmutex);
11.      end
12.    else
13.      nReaders:=nReaders + 1;
14.      V(mutex);
15.      read(f);
16.      P(mutex);
17.      nReaders:=nReaders - 1;
18.      if nReaders = 0 then
19.        V(wmutex);
20.        V(mutex);
21.      end;

22. procedure WRITER(d: data);
23.   begin
24.     P(srmutex);
25.     P(wmutex);
26.     write(f, d);
27.     V(wmutex);
28.     V(srmutex);
29.   end;

30. begin // initialization
31.   mutex:=wmutex:=srmutex:=1;
32.   nReaders:=0;
33. end.

```

(c) Zakład Systemów Informatycznych

Slajd 17

```

region v do
begin
  S1;
  await(B);
  S2;
end;

```

(c) Zakład Systemów Informatycznych

Slajd 15

Rozwiązanie problemu pisarzy i czytelników z użyciem regionów krytycznych

```

1. var v: shared record
2.   nReaders, nWriters: INTEGER;
3.   busy: BOOLEAN;
4. end;

Proces czytelnika
5. region v do
6.   begin
7.     await(nWriters=0);
8.     nReaders:=nReaders + 1;
9.     end;
10. ...
11. read file
12. ...
13. region v do
14.   begin
15.     nReaders:=nReaders - 1;
16.   end;

Proces pisarza
17. region v do
18.   begin
19.     nWriters:=nWriters + 1;
20.     await((not busy)and(nReaders=0);
21.     busy:=True;
22.     end;
23. ...
24. write file
25. ...
26. region v do
27.   begin
28.     nWriters:=nWriters- 1;
29.     busy:=False;
30.   end;

```

(c) Zakład Systemów Informatycznych

Slajd 18

Problem pisarzy i czytelników

In the readers-writers problem, the shared resource is a file that is accessed by both the reader and writer process. Reader processes simply read the information in the file without changing its content. Writer processes may change the information in the file. The basic synchronization constraint is that any number of readers should be able to concurrently access the file, but only one writer can access the file at a given time. Moreover, readers and writers must always exclude each other.



(c) Zakład Systemów Informatycznych

Slajd 16

Monitory – definicja

A monitor is characterized by a set of programmer-defined operators. The representation of a monitor type consists of declarations of variables whose values define the state of an instance of the type, as well as the bodies of procedures or functions that implement operations on the type. The syntax of a monitor is

```

type MONITOR_NAME = monitor
  variable declarations

  procedure entry P1 (...);
  begin ... end;

  procedure entry P2 (...);
  begin ... end;

  ...

  procedure entry Pn (...);
  begin ... end;

begin
  initialization code;
end;

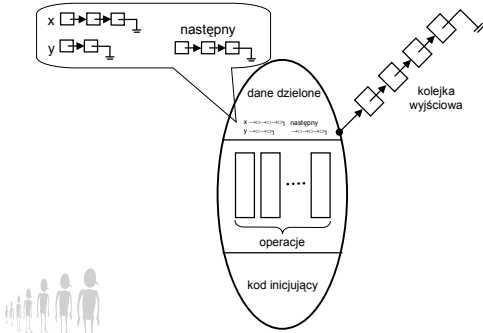
```



(c) Zakład Systemów Informatycznych

Slajd 19

Schemat monitora



(c) Zakład Systemów Informatycznych

Slajd 20

Alokacja zasobów z wykorzystaniem monitora

```

1. type RESOURCE_ALLOCATION = monitor
2. var busy: BOOLEAN;
3.   x: INTEGER;

4. procedure entry ACQUIRE(time : INTEGER);
5. begin
6.   if busy then x.wait(time); // process priority
7.   busy := True;
8. end;

9. procedure entry RELEASE;
10. begin
11.   busy := False;
12.   x.signal;
13. end;

14. begin
15.   busy := False;
16. end.
```



(c) Zakład Systemów Informatycznych

Slajd 23

Operacje wait i signal

Programista, który chce zapisać przykrojony na miarę własnych potrzeb schemat synchronizacji, może zdefiniować jedną lub kilka zmiennych typu warunek:

```
var x, y: CONDITION;
```



Jedynymi operacjami, które mogą dotyczyć warunku, są operacje:

□ x.wait

oznacza, że proces ją wywołujący zostaje zawieszony do czasu, aż inny proces wywoła operację x.signal

□ x.signal

wznawia dokładnie jeden z zawieszonych procesów. Jeśli żaden proces nie jest zawieszony, to operacja ta nie ma żadnych skutków, tzn. stan zmiennej x jest taki, jak gdyby operacji tej nie wykonano wcale.

(c) Zakład Systemów Informatycznych

Slajd 21

Rozwiązanie problemu czytelników i pisarzy z wykorzystaniem monitorów

```

1. type READERS_WRITERS = monitor;
2. var readerCount : INTEGER;
3.   busy : BOOLEAN;
4.   OKtoRead, OKtoWrite : CONDITION;

5. procedure entry STARTREAD;
6. begin
7.   if busy
8.   then OKtoRead.wait ;
9.   readerCount:=readerCount+1;
10.  OKtoRead.signal;
11.  // Once one reader can start, they all can
12. end;

13. procedure entry ENDREAD;
14. begin
15.   readerCount:=readerCount-1;
16.   if readerCount = 0
17.   then OKtoWrite.signal;

18. procedure entry STARTWRITE;
19. begin
20.   if busy or readerCount ≠ 0
21.   then OKtoWrite.wait;
22.   busy:=True;
23. end;

24. procedure entry ENDWRITE;
25. begin
26.   busy:=False;
27.   if OKtoRead.queue
28.   then OKtoRead.signal
29.   else OKtoWrite.signal;
30. end;

31. begin // initialization
32.   readerCount:=0;
33.   busy :=False;
34. end ;
```



(c) Zakład Systemów Informatycznych

Slajd 24

Rozwiązanie problemu producenta – konsumenta z wykorzystaniem monitorów

```

1. type PRODUCER_CONSUMER = monitor
2. var full, empty : CONDITION;
3. count : INTEGER;

4. procedure entry ENTER;
5. begin
6.   if count = N then full.wait;
7.   enter_item;
8.   count:=count + 1;
9.   if count = 1 then empty.signal;
10. end;

11. procedure entry REMOVE;
12. begin
13.   if count = 0 then empty.wait;
14.   remove_item;
15.   count:=count - 1;
16.   if count=N - 1 then full.signal;
17. end;

18. begin
19.   count:=0;
20. end monitor;

21. procedure PRODUCER;
22. begin
23.   while True do
24.   begin
25.     produce_item;
26.     PRODUCER_CONSUMER.enter;
27.   end
28. end;

29. procedure CONSUMER;
30. begin
31.   while True do
32.   begin
33.     PRODUCER_CONSUMER.REMOVE;
34.     consume_item
35.   end;
36. end.
```

(c) Zakład Systemów Informatycznych

Slajd 22

Monitor – implementacja

```

1. wait(mutex);
2. ...
3. treść procedury F;
4. ...
5. if nextCount > 0
6. then signal(next);
7. else signal(mutex);
```

- mutex – semafor gwarantujący wzajemne wykluczenie
- nextCount – ilość procesów, które the number of processes invoking x.signal and suspended
- next – semaphore to suspend a process invoking x.signal
- xCount – the number of processes waiting for x.signal
- xSem – semaphore to suspend a process invoking x.wait

```

x.wait:
8. xCount:=xCount+1;
9. if nextCount > 0
10. then signal(next)
11. else signal(mutex);
12. wait(xSem);
13. xCount:=xCount-1;
```

```

x.signal:
14. if xCount > 0
15. then
16.   begin
17.     nextCount:=nextCount+1;
18.     signal(xSem);
19.     wait(next);
20.     nextCount:=nextCount-1;
21.   end.
```



(c) Zakład Systemów Informatycznych

Slajd 25

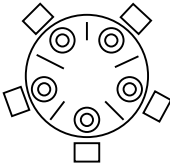
Problem jedzących filozofów

The dining philosophers problem is a classic problem that has formed the basis for a large class of synchronization problems. In one version of this problem five philosophers are sitting in a circle, attempting to eat spaghetti with the help of forks. Each philosopher has a bowl of spaghetti but there are only five forks (with one fork placed on the left and one to the right of each philosopher) to share among them. This creates a dilemma, as both forks (to the left and right) are needed by each philosopher to consume the spaghetti.

A philosopher alternates between two phases: thinking and eating. In the *thinking* mode, the philosopher does not hold a fork. However, when hungry (after staying in the thinking mode for a finite time), a philosopher attempts to pick up both forks on the left and right sides. (At any given moment, only one philosopher can hold a given fork, and a philosopher cannot pick up two forks simultaneously). A philosopher can start eating only after obtaining both forks. Once a philosopher starts eating, the forks are not relinquished until the eating phase is over. When the eating phase concludes (which lasts for finite time), both forks are put back in their original position and the philosopher reenters the thinking phase.

Note that no two neighboring philosophers, can eat simultaneously. In any solution to this problem, the act of picking up a fork by a philosopher must be a critical section. Devising a deadlock-free solution to this problem, in which no philosopher starves, is nontrivial.

Schemat filozofów



Rozwiązanie problemu jedzących filozofów z wykorzystaniem monitorów

```
1. type DINNING_PHILOSOPHERS = monitor
2. var state : array [0..4] of (Thinking, Hungry, Eating);
3. var self : array [0..4] of CONDITION;

4. procedure entry PICKUP(i: 0..4);
5. begin
6.   state[i]:=Hungry;
7.   test(i);
8.   if state[i] ≠ eating
9.   then self[i].wait;
10. end ;

11. procedure entry PUTDOWN(i: 0..4);
12. begin
13.   state[i]:=Thinking;
14.   test(i + 4 mod 5);
15.   test(i + 1 mod 5);
16. end;

17. procedure TEST(k: 0..4);
18. begin
19.   if state[k+4 mod 5] ≠ Eating
20.   and state[k] = Hungry
21.   and state[k+1 mod 5] ≠ Eating
22.   then
23.     begin
24.       state[k]:=Eating;
25.       self[k].signal;
26.     end;
27. end;

28. begin
29.   for i:=0 to 4 do
30.     state[i]:=Thinking;
31.   end ;
```

