

Programowanie współbieżne

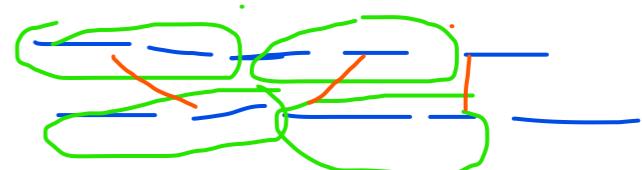
wstęp

Program współbieżny składa się z pewnej ilości zadań które mogą być wykonywane równolegle.

Pojedyncze zadanie to **proces sekwencyjny**, czyli ciąg operacji, w którym kolejna operacja zaczyna się po zakończeniu poprzedniej operacji.

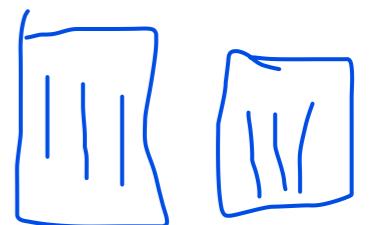
Jeżeli mamy kilka procesów sekwencyjnych, których operacje nakładają się w czasie to mówimy, że są to **procesy współbieżne**.

Procesy współbieżne mogą być wykonywane **jednocześnie**, jeżeli są realizowane przez osobne procesory, lub metodą przeplotu, jeżeli jeden procesor realizuje obliczenia wielu procesów dokonując **podziału czasu** między zadaniami pochodząymi od różnych procesów.



Procesy współbieżne mogą się komunikować ze sobą za pomocą **pamięci wspólnej** lub za pomocą **przesyłania wiadomości**

Proces jest tworzony, nadzorowany i likwidowany przez **system operacyjny**, który przydziela procesowi jego zasoby: obszar pamięci, w którym są przechowywane instrukcje, dane, stos, zbiór rejestrów (w tym licznik instrukcji i wskaźnik stosu) ...



Proces jest wykonywany przez jeden **wątek** (thread) lub zespół współpracujących wątków.

Wątek to jednostka systemu operacyjnego realizująca niezależny strumień instrukcji. Wszystkie wątki realizowane w ramach jednego procesu współdzielą przyznane temu procesowi zasoby, w szczególności przestrzeń adresową pamięci. Każdy wątek ma też zasoby do swojego wyłącznego użytku, np. stos, rejestr licznika instrukcji ...

Tworzenie procesu jest dla systemu operacyjnego znacznie bardziej obciążającą operacją niż tworzenie wątków.

Podstawowa trudność występująca w tworzeniu i analizie programów współbieżnych:

Zawsze trzeba pamiętać, że oprócz procesu, który programujemy/analizujemy są inne procesy, które mogą wpływać na nasz proces

Przykład.

Proces P1:

...

$x=x+1$

...

Proces P2:

...

$x=x+1$

...

x Współlna

Załóżmy, że x jest wspólną zmienną obu procesów (współdzielona pamięć)

Jeżeli instrukcja $x=x+1$ zostanie przez kompilator przetłumaczona na operację inkrementacji (atomową), to współbieżne wykonanie tych procesów zwiększy wartość zmiennej x o 2 niezależnie od kolejności wykonywania instrukcji obu procesów.

Proces P1:

...
x=x+1

Proces P2:

...
x=x+1

...

Załóżmy, że **x** jest wspólną zmienną obu procesów (współdzielona pamięć)

Jeżeli każda z tych instrukcji będzie przetłumaczona na sekwencję trzech operacji atomowych:

LOAD rejestr x
ADD rejestr 1
STORE rejestr x



to możliwa jest i taka kolejność wykonania, po której w zmiennej x będzie 1

P1: LOAD rejestr x
P2: LOAD rejestr x
P1: ADD rejestr 1
P2: ADD rejestr 1
P2: STORE rejestr x
P1: STORE rejestr x

zmienna x zwiększała się o 1

Ten wykład będzie miał dwie warstwy:

warstwa ogólna: typowe zagadnienia i problemy związane z synchronizacją procesów współbieżnych i analizowaniem procesów współbieżnych

warstwa konkretna: komunikacja między procesami w Uniksie (ta tematyka będzie dominować na laboratoriach)

Procesy i komunikacja między nimi w systemach uniksowych

Wybrane podstawowe parametry procesu uniksowego:

- identyfikator procesu PID (Process ID)
- identyfikator procesu rodzicielskiego PPID (Parent Process ID)
- identyfikator grupy procesów PGRP (Process Group)
- rzeczywisty identyfikator użytkownika UID (User ID)
- efektywny identyfikator użytkownika EUID (Effective User ID)
- rzeczywisty identyfikator grupy użytkowników GID (Group ID)
- efektywny identyfikator grupy użytkowników EGID (Effective Group ID)

Koordynacja procesów przy użyciu interpretatora komend systemu Unix

Pojedynczą komendę lub pojedynczy potok komend uruchomiony z linii poleceń powłoki (shell) nazywamy *zadaniem* (job).

komenda uruchamia proces pierwszoplanowy

komenda & uruchamia proces drugoplanowy (bez dostępu do terminala)

(*komenda* &) uruchamia proces drugoplanowy korzystając z podpowłoki

Ctrl-Z zawiesza proces pierwszoplanowy

bg *zadanie* (lub *zadanie* &) uruchamia w tle zawieszony proces

fg *zadanie* (lub *zadanie*) uruchamia zawieszony proces na pierwszym planie

kill -9 *zadanie* powoduje zakończenie procesu (działającego lub zawieszonego)

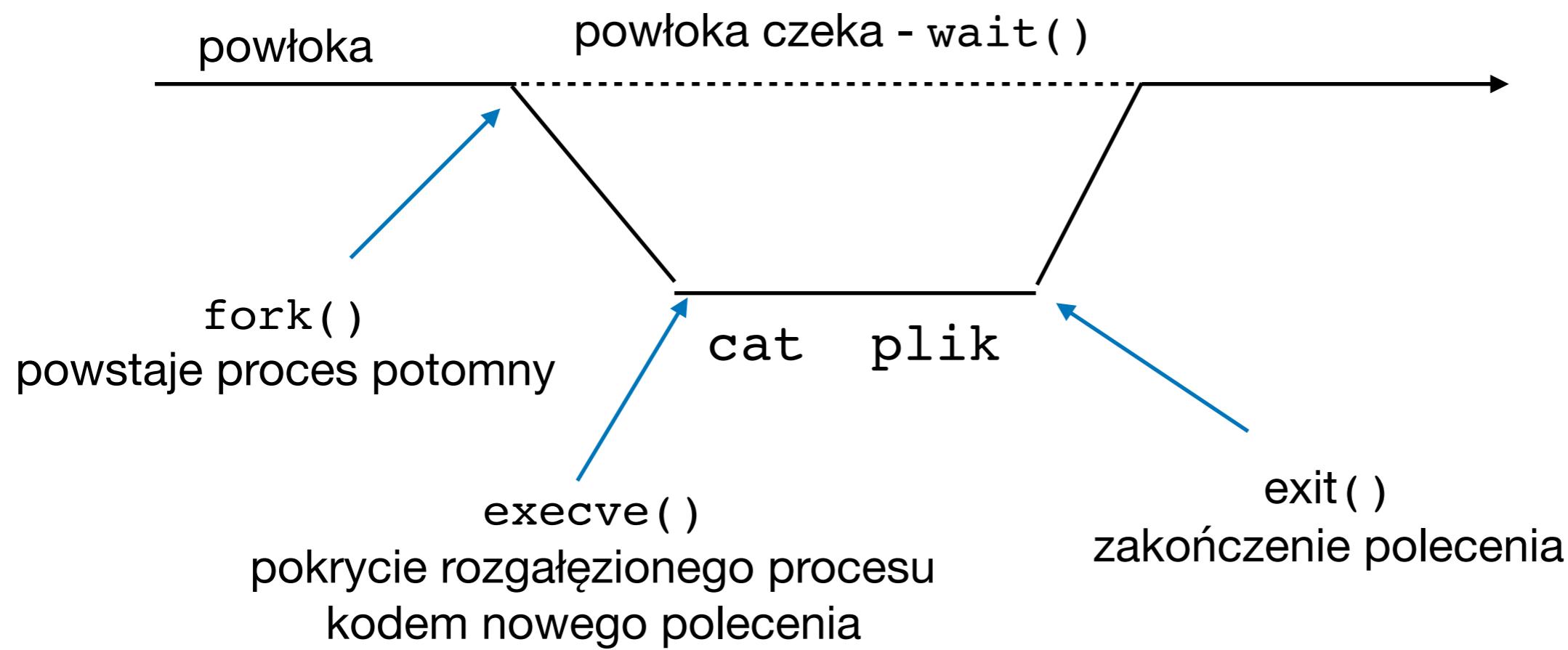
Można uruchomić kilka procesów jednocześnie z jednej linii poleceń

komenda1 & *komenda2* &

Tworzenie procesów z poziomu powłoki

na przykład:

`cat plik`



podobnie dla uruchomienie w tle, tylko powłoka nie czeka na zakończenie rozgałęzionego procesu

`(cat plik > plik1) &`

Synchronizacja procesów przy pomocy plików blokujących (lock files)

Dwa procesy P_1 i P_2 próbują uzyskać dostęp do jednego niewspółnego zasobu (drukarka, dysk, plik). Wyłączność dostępu zapewnia następujący protokół.

- ustalamy nazwę pliku blokującego i miejsce, w którym on ma się znajdować
- jeżeli P_1 chce uzyskać dostęp do chronionego zasobu sprawdza najpierw, czy plik blokujący istnieje.
 - jeżeli plik blokujący **nie istnieje**, P_1 tworzy go i zajmuje zasób
 - po skończeniu wykorzystywania zasobu, P_1 usuwa plik blokujący
 - jeżeli plik blokujący **istnieje**, P_1 w pętli oczekuje jakąś ilość czasu i znów sprawdza czy istnieje plik blokujący aż do czasu, kiedy pliku blokującego nie będzie
- P_2 działa identycznie

Problemy

- trzeba ustalić nazwę pliku blokującego i miejsce, w którym on ma się znajdować
- co się stanie, jeżeli jednemu z procesów nie uda się usunąć pliku blokującego?
- czekanie w pętli na zniknięcie pliku blokującego marnotrawi czas procesora
- jest niebezpieczeństwo, że pomiędzy stwierdzeniem przez P_1 że pliku blokującego nie ma i utworzeniem go, proces P_2 może też stwierdzić, że pliku blokującego nie ma (wystąpienie tzw. wyścigu)

Rozwiązańe uniksowe

```
int creat(const char *p, mode_t m)
```

"/kat/plik"

p - wskaźnik na ścieżkę (z nazwą) do tworzonego pliku

m - tryb otwierania tworzonego pliku (prawa dostępu); wartość 0 oznacza brak praw do odczytu, zapisu i wykonywania

zwracana wartość w przypadku sukcesu: deskryptor otwartego pliku (liczba nieujemna całkowita)

zwracana wartość w przypadku niepowodzenia: -1; w zmiennej errno będzie wtedy wartość wskazująca przyczynę niepowodzenia, np. wartość równa stałej EACCES z pliku nagłówkowego <fcntl.h> (czyli 13) oznacza brak praw dostępu do pliku

pliki nagłówkowe: <fcntl.h> <sys/types.h> <sys/stat.h>

Alternatywne rozwiązanie

```
open(p, O_WRONLY|O_CREAT, 0);
```

gdzie

```
int open(const char *p, int oflag, ... );
```

p - wskaźnik na ścieżkę (z nazwą) do tworzonego pliku

oflag - flagi (zdefiniowane w <fcntl.h>) specyfikujące zachowanie funkcji i tryb otwierania tworzonego pliku (prawa dostępu);

zwracana wartość - jak w funkcji creat()

Usuwanie pliku blokującego

```
int unlink(const char *p);
```

p - wskaźnik na ścieżkę (z nazwą) do usuwanego pliku

zwracana wartość w przypadku sukcesu: 0

zwracana wartość w przypadku niepowodzenia: podobnie jak w funkcji
`creat()`

Synchronizacja procesów przy pomocy plików blokujących (lock files, pliki zamkowe)

Dwa procesy P_1 i P_2 próbują uzyskać dostęp do jednego niewspółnego zasobu (drukarka, dysk, plik). Wyłączność dostępu zapewnia następujący protokół.

- ustalamy nazwę pliku blokującego i miejsce, w którym on ma się znajdować
- jeżeli P_1 chce uzyskać dostęp do chronionego zasobu sprawdza najpierw, czy plik blokujący istnieje.
- jeżeli plik blokujący **nie istnieje**, P_1 tworzy go i zajmuje zasób
 - po skończeniu wykorzystywania zasobu, P_1 usuwa plik blokujący
- jeżeli plik blokujący **istnieje**, P_1 w pętli oczekuje jakąś ilość czasu i znów sprawdza czy istnieje plik blokujący aż do czasu, kiedy pliku blokującego nie będzie
- P_2 działa identycznie

Problemy

- trzeba ustalić nazwę pliku blokującego i miejsce, w którym on ma się znajdować
- co się stanie, jeżeli jednemu z procesów nie uda się usunąć pliku blokującego?
- czekanie w pętli na zniknięcie pliku blokującego marnotrawi czas procesora
- jest niebezpieczeństwo, że pomiędzy stwierdzeniem przez P_1 że pliku blokującego nie ma i utworzeniem go, proces P_2 może też stwierdzić, że pliku blokującego nie ma (wystąpienie tzw. wyścigu)

atomowe

int creat(const char *p, mode_t m)

np. `creat("~/plikZamkowy", 0)`

p - wskaźnik na ścieżkę (z nazwą) do tworzonego pliku

m - tryb otwierania tworzonego pliku (prawa dostępu); wartość 0 oznacza brak praw do odczytu, zapisu i wykonywania

zwracana wartość w przypadku sukcesu: deskryptor otwartego pliku (liczba nieujemna całkowita)

zwracana wartość w przypadku niepowodzenia: -1; w zmiennej errno będzie wtedy wartość wskazująca przyczynę niepowodzenia, np. wartość równa stałej EACCES z pliku nagłówkowego `<fcntl.h>` (czyli 13) oznacza brak dostępu do pliku

pliki nagłówkowe: `<fcntl.h>` `<sys/types.h>` `<sys/stat.h>`

Alternatywne rozwiązanie

```
open(p, O_WRONLY|O_CREAT, 0);
```

np. `open("~/plikZamkowy", O_WRONLY|O_CREAT, 0);`

gdzie

```
int open(const char *p, int oflag, ... );
```

p - wskaźnik na ścieżkę (z nazwą) do tworzonego pliku

oflag - flagi (zdefiniowane w <fcntl.h>) specyfikujące zachowanie funkcji i tryb otwierania tworzonego pliku (prawa dostępu);

zwracana wartość - jak w funkcji `creat()`

```
open(p, O_WRONLY|O_CREAT|O_EXCL, 0777);
```

gdzie

```
int open(const char *p, int oflag, int m);
```

p - wskaźnik na ścieżkę (z nazwą) do tworzonego pliku

m - liczba kodująca informację o prawach dostępu (kto i jakie)

oflag - flagi (zdefiniowane w <fcntl.h>) specyfikujące zachowanie funkcji i tryb otwierania tworzonego pliku

zwracana wartość w przypadku sukcesu: deskryptor otwartego pliku (liczba nieujemna całkowita)

zwracana wartość w przypadku niepowodzenia: -1; w zmiennej errno będzie wtedy wartość wskazująca przyczynę niepowodzenia, np. wartość równa stałej EEXIST z pliku nagłówkowego <fcntl.h> oznacza, że plik już istnieje

Usuwanie pliku blokującego

```
int unlink(const char *p);
```

np. `int unlink("~/plikZamkowy")`

p - wskaźnik na ścieżkę (z nazwą) do usuwanego pliku

zwracana wartość w przypadku sukcesu: 0

zwracana wartość w przypadku niepowodzenia: podobnie jak w funkcji
`creat()`

Rozwiązańe uniksowe opakowane Pythonem

```
import os
import time
import errno

class FileLockException(Exception):
    pass

# tworzenie pliku zamkowego
while True:
    try:
        #Open file exclusively
        fd = os.open("plikZamkowy", os.O_CREAT|os.O_EXCL|os.O_RDWR)
        break;
    except OSError as e:
        if e.errno != errno.EEXIST:
            raise
        time.sleep(0.05)
print("plik zamkowy utworzony")
```



```
# operacje zabezpieczone plikiem zamkowym
print("operacje zabezpieczone plikiem zamkowym")
time.sleep(2)

# usuwanie pliku zamkowego
os.close(fd)
os.unlink("plikZamkowy")
print("koniec, plik zamkowy zlikwidowany")
```

uruchomienie dwóch kopii powyższego programu zapisanego w pliku lockf2.py

Dorotkas-MacBook-Pro:WspolbP pmp\$

Dorotkas-MacBook-Pro:WspolbP pmp\$

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 lockf2.py & python3 lockf2.py**

[2] 4539 ↵

plik zamkowy utworzony

operacje zabezpieczone plikiem zamkowym

koniec, plik zamkowy zlikwidowany

plik zamkowy utworzony ↵

operacje zabezpieczone plikiem zamkowym

koniec, plik zamkowy zlikwidowany

[2]- Done

python3 lockf2.py

Dorotkas-MacBook-Pro:WspolbP pmp\$

tu były 2 sekundy
przerwy

**uruchomienie powyższego programu zapisanego w pliku lockf2.py
z wydrukiem katalogu przed i w trakcie wykonywania**

Dorotkas-MacBook-Pro:W pmp\$

```
Dorotkas-MacBook-Pro:W pmp$ ls  
lockf2.py
```

```
Dorotkas-MacBook-Pro:W pmp$ python3 lockf2.py
```

plik zamkowy utworzony

operacje zabezpieczone plikiem zamkowym

z sleep(2)

python3 lockf2.py

Dorotkas-MacBook-Pro:W pmp\$ ls

lockf2.py plikZamkowy

Dorotkas-MacBook-Pro:W pmp\$ fg

python3 lockf2.py

koniec, plik zamkowy zlikwidowany

Dorotkas-MacBook-Pro:W pmp\$ ls

lockf2.py

Dorotkas-MacBook-Pro:W pmp\$

przed

w trakcie wykonywania

8

Link do ładnej implementacji plików zamkowych w Pythonie:

<https://programmer.group/a-python-implementation-of-lock-file.html>

Uwaga:

mechanizmu plików zamkowych, po angielsku **lock file**, nie należy mylić z mechanizmem **file lock**, czyli blokady zakładanej na dostęp do danego pliku

Rozgałęzianie procesów (Unix)

funkcja systemowa `fork()`

`pid_t fork(void)` - tworzy proces-potomka, który różni się od rodzica tylko numerami PID i PPID i tym, że informacje statystyczne są wyzerowane.

zwracana wartość to:

- w procesie macierzystym, czyli procesie-ojcu: PID utworzonego potomka (wartość > 0)
- w procesie synu: wartość 0
- w przypadku błędu, zwracana wartość (tylko w ojcu) to -1

Dostęp do funkcji systemowych z Pythona

moduł os

`os.fork()` - wywołuje funkcję systemową `fork()`

istnieją też inne moduły pozwalające tworzyć nowe procesy np.
`multiprocessing`

Przykład użycia fork()

```
import os
import time
import errno

# rozgałęziamy proces
pid = os.fork()
# niezerowy pid oznacza, że jesteśmy w procesie macierzystym
if pid>0 :
    print("ojciec: mój PID: ", os.getpid())
    print("ojciec: pid stworzonego syna: ",pid)
    print("ojciec: kontynuacja działania")
    time.sleep(1)
    print("ojciec: kończę")
elif pid == 0:
    print("syn: zaczynam")
    print("syn: mój PID: ", os.getpid())
    time.sleep(1)
    print("syn: kończę")
else:
    print("ojciec: błąd przy tworzeniu procesu")
```

Syn

uruchomienie powyższego programu zapisanego w pliku fork.py

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 fork.py

ojciec: mój PID: 5028

ojciec: pid stworzonego syna: 5029

ojciec: kontynuuacja działania

syn: zaczynam

syn: mój PID: 5029

syn: kończę

ojciec: kończę

Dorotkas-MacBook-Pro:WspolbP pmp\$

← Sleep

inne uruchomienie powyższego programu po drobnej modyfikacji

zmieniony czas czekania w procesie syna na
`time.sleep(2)`

Dorotkas-MacBook-Pro:WspolbP pmp\$ `python3 fork.py`

ojciec: mój PID: 5115

ojciec: pid stworzonego syna: 5116

ojciec: kontynuuacja działania

syn: zaczynam

syn: mój PID: 5116

ojciec: kończę

Dorotkas-MacBook-Pro:WspolbP pmp\$ `syn: kończę`

sleep Ojciec

sleep syn

Prymitywne mechanizmy koordynacji procesów

funkcje systemowe `exit()`, `_exit()` i `wait()`

`void _exit(int status)` - kończy działanie procesu i przesyła **jednobajtową** liczbę (kod wyjścia) do jego procesu rodzicielskiego

`void exit(int status)` - jak `_exit()` ale dodatkowo wywołuje jeszcze pewne funkcje sprzątające

status[0]
`pid_t wait(int *wstatus);`
`pid_t waitpid(pid_t pid, int *wstatus, int options)`

- powodują zawieszenie procesu w oczekiwaniu na zakończenie jego procesu potomnego; odbierają jego kod wyjścia

Dostęp do funkcji systemowych z Pythona

`os._exit()` - wywołuje funkcję systemową `_exit()`

`sys.exit()` - działa jak `exit()` *systemowy*

Przykład użycia exit() wait()

```
import os
import time
import sys

# rozgałęziamy proces
pid = os.fork()

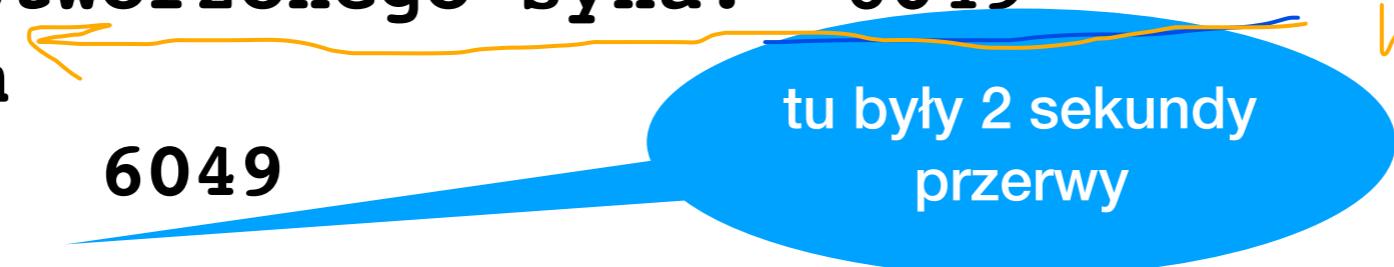
# niezerowy pid oznacza, że jesteśmy w procesie macierzystym
if pid>0 :
    print("ojciec: pid stworzonego syna: ",pid)
    # czekanie na zakończenie (jakiegoś) syna
    status = os.wait()

    print("ojciec: PID procesu zakończonego: ", status[0])
    if os.WIFSIGNALED(status[1]):
        print("ojciec: Sygnał, który zabił proc. syna",status[1])
    if os.WIFEXITED(status[1]):
        print("ojciec: zwrócony kod powrotu syna ",
              os.WEXITSTATUS(status[1]))
```

```
else :  
    print("syn: zaczynam")  
    print("syn: mój PID: ", os.getpid())  
    time.sleep(2)  
    print("syn: kończę")  
    os._exit(11)  
#     sys.exit(11)
```

uruchomienie powyższego programu zapisanego w pliku forkwait.py

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 forkwait.py
ojciec: pid stworzonego syna: 6049
syn: zaczynam
syn: mój PID: 6049
syn: kończę
ojciec: PID procesu zakońzonego: 6049
ojciec: zwrócony kod powrotu syna 11
Dorotkas-MacBook-Pro:WspolbP pmp$
```



ojciec czekał na syna, pomimo, że nie miał własnych operacji do wykonania

uruchomienie powyższego programu zapisanego w pliku forkwait.py z zatrzymaniem syna przez wysłanie sygnału SIGKILL

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 forkwait.py
ojciec: pid stworzonego syna: 6382
syn: zaczynam
syn: mój PID: 6382
^Z
[2]+ Stopped                  python3 forkwait.py
Dorotkas-MacBook-Pro:WspolbP pmp$ kill -9 6382
Dorotkas-MacBook-Pro:WspolbP pmp$ fg
python3 forkwait.py
ojciec: PID procesu zakończonego: 6382
ojciec: Sygnał, który zabił proces syna 9
Dorotkas-MacBook-Pro:WspolbP pmp$
```

syn miał wydłużony czas time.sleep(2), żeby zdążyć go zabić

Przykład użycia exit() wait() plus “przekazanie parametru” do procesu potomnego

```
parametr = 1
paramDlaSyna = 2
# rozgałęziamy proces
pid = os.fork()

# niezerowy pid oznacza, że jesteśmy w procesie macierzystym
if pid :
    print("ojciec: pid stworzonego syna: ",pid)
    .....
    Parametr = 1
else :
    print("syn: zaczynam")
    parametr = paramDlaSyna == 2
    print("syn: mój PID: ", os.getpid())
    time.sleep(1)
    print("syn: kończę")
    os._exit(parametr)
#     sys.exit(parametr)
```

Diagram illustrating the flow of parameters between the parent and child processes:

- The parent process (Ojciec) creates a child process.
- The parent process prints its own PID and sets the parameter to 1.
- The child process (syn) starts by printing "syn: zaczynam".
- The child process receives the parameter value from the parent (set to 2).
- The child process prints its own PID and sleeps for 1 second.
- The child process prints "syn: kończę" and exits with the parameter value (2).

~~uruchomienie powyższego programu zapisanego w pliku
forkwaitP.py~~

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 forkwaitP.py
ojciec: pid stworzonego syna: 6160
syn: zaczynam
syn: mój PID: 6160
syn: kończę
ojciec: PID procesu zakońzonego: 6160
ojciec: zwrócony kod powrotu syna 2
Dorotkas-MacBook-Pro:WspolbP pmp$
```

uruchomienie powyższego programu zapisanego w pliku forkSilnia.py

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 forkSilnia.py

PID stworzonego syna: 6304

5.

syn: mój PID i n: 6304 4

PID stworzonego syna: 6305

syn: mój PID i n: 6305 3

PID stworzonego syna: 6306

syn: mój PID i n: 6306 2

PID stworzonego syna: 6307

syn: mój PID i n: 6307 1

PID stworzonego syna: 6308

syn: mój PID i n: 6308 0

syn 6308 kończę



syn 6307 kończę

$$2 \cdot 1 = 2$$

syn 6306 kończę

$$3 \cdot 2 = 6$$

syn 6305 kończę

$$3 \cdot 6 = 18$$

120

5:18

Dorotkas-MacBook-Pro:WspolbP pmp\$

Przykład użycia exit() wait() fork()

liczenie silni

```
def silnia(n):    h*(n-1)(n-2)...1
    if n == 0:
        return 1;
    # rozgałęziamy proces
    pid = os.fork()
    if pid>0 : #proces macierzysty
        print("PID stworzonego syna: ",pid)
        # czekanie na zakończenie (jakiegoś) syna
        status = os.wait()
        if os.WIFSIGNALED(status[1]):
            print("ojciec: Sygnał, który zabił proces syna",
                  status[1])
        if os.WIFEXITED(status[1]):
            return n*os.WEXITSTATUS(status[1])
    else : # syn
        n=n-1
        print("syn: mój PID i n: ", os.getpid(), n)
        wynik=silnia(n)
        print("syn ", os.getpid()," kończę")
        os._exit(wynik)
```

filia(5)

Bibliografia

<https://man7.org/linux/man-pages/man2/fork.2.html>

<https://man7.org/linux/man-pages/man2/wait.2.html>

<https://docs.python.org/3/library/os.html>

Klasyczne problemy programowania współbieżnego

Wzajemne wykluczanie

Dwa lub więcej procesów współbieżnych wykonuje następujące operacje

P_i:

```
while true
    sekcja_lokalna_i
    protokol_wjsciowy_i
    sekcja_krytyczna_i
    protokol_wjsciowy_i
```

Wymagane własności

- *wzajemne wykluczanie*: żadne dwa procesy nie mogą być równocześnie w swoich sekcjach krytycznych
- zatrzymanie się któregoś z procesów w jego sekcji lokalnej nie może zakłócić działania innych procesów (ich możliwości wchodzenia do ich sekcji krytycznych)

Dwa lub więcej procesów współbieżnych wykonuje następujące operacje

P_i:

```
while true
    sekcja_lokalna_i
    protokol_wejsciowy_i
    sekcja_krytyczna_i
    protokol_wyjsciowy_i
```

Wymagane własności - c.d.

deadlock, zakończenie

- brak blokady: nie może nastąpić taka sytuacja, że pewna ilość procesów próbuje wejść do swoich sekcji krytycznych (wykonując protokoły wejściowe), ale żaden z nich ani z pozostałych procesów do niej nie wchodzi.
- brak zagłodzenia: jeżeli proces podejmuje próbę wejścia do swojej sekcji krytycznej to w końcu do niej wejdzie
- Jeżeli proces chce wejść do sekcji krytycznej a inne nie, to wejdzie do niej

Propozycje rozwiązań dla dwoch procesów

zakładamy, że zapis i odczyt zmiennej z pamięci wspólnej jest operacją niepodzielną

Notacja: Dla $i = 1$ lub 2 , w definicji procesu P_i przez j będziemy oznaczać tę drugą wartość, czyli

jeżeli $i == 1$ to $j == 2$

jeżeli $i == 2$ to $j == 1$

Propozycja 1

zmienna wspólna czyja_kolej o wartości początkowej 1

P_i:

```
while true
    sekcja_lokalna_i
    do Pi ss while czyja_kolej != i      // aktywne czekanie
        sekcja_krytyczna_i
        czyja_kolej = j
```

Jest wzajemne wykluczanie

Nie ma blokady

Nie ma zagłodzenia

Pozostałe własności *nie są spełnione*

To rozwiązanie też nie jest satysfakcjonujące, jeżeli procesy działają w mocno
różnym tempie

Propozycja 2

dwie zmienne wspólne k_1 i k_2 o wartości początkowej 1

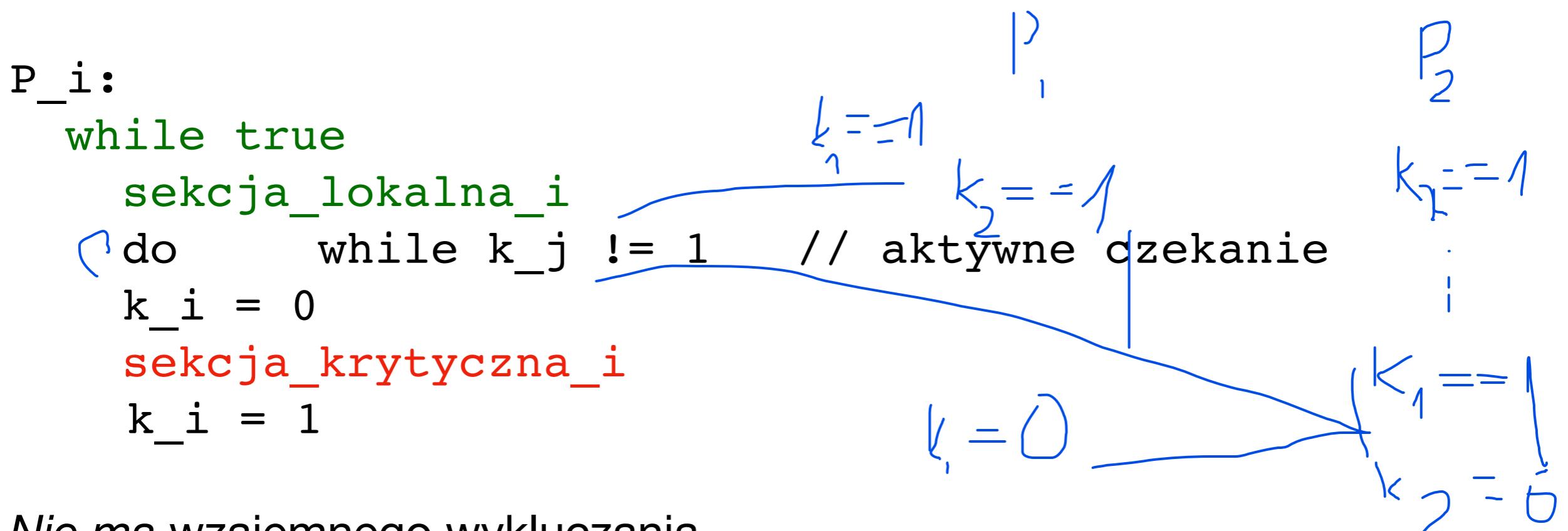
idea: zmienna k_i dostaje wartość 0 gdy P_i wchodzi do swojej sekcji krytycznej a gdy ją opuszcza zmienia wartość k_i na 1 czyli $k_i == 0$ oznacza "P_i jest w swojej sekcji krytycznej"

P_i:

```
while true
    sekcja_lokalna_i
        do      while k_j != 1 // aktywne czekanie
            k_i = 0
            sekcja_krytyczna_i
            k_i = 1
```

Nie ma wzajemnego wykluczania

Pomiędzy sprawdzeniem że $k_j == 1$ a instrukcją $k_1 = 0$ drugi proces może wejść do swojej sekcji krytycznej



Propozycja 3

dwie zmienne wspólne k_1 i k_2 o wartości początkowej 1

idea: zmienna k_i dostaje wartość 0 gdy P_i chce wejść do swojej sekcji krytycznej a gdy ją opuszcza zmienia wartość k_i na 1 czyli $k_i == 0$ oznacza "P_i chce wejść do swojej sekcji krytycznej"

P_i:

```
while true
    sekcja_lokalna_i
    k_i = 0
    do      while k_j != 1    // aktywne czekanie
        sekcja_krytyczna_i
        k_i = 1
```

Jest wzajemne wykluczanie
Może nastąpić blokada

P₁: $k_1 = 0$ $k_2 \neq 1$
P₂: $k_2 = 0$ $k_1 \neq 1$

Propozycja 4

dwie zmienne wspólne `k_1` i `k_2` o wartości początkowej 1

idea: jak w Propozycji 3, ale dajemy szansę na wyjście z blokady

`P_i:`

```
while true
    sekcja_lokalna_i
    k_i = 0
    do
        k_i=1 ; k_i = 0
        while k_j != 1 // aktywne czekanie
        sekcja_krytyczna_i
        k_i = 1
```

`P_j`

Jest wzajemne wykluczanie (`k_i==0` gdy sprawdzamy czy `k_j != 1`)
Może nastąpić blokada (właściwie półblokada czyli livelock)
Może nastąpić zagłodzenie

Propozycja 5. Algorytm Dekkera.

dwie zmienne wspólne `k_1` i `k_2` o wartości początkowej 1
zmienna wspólna `czyja_kolej` o wartości początkowej 1

idea: połączenie Propozycji 4 i 1

`P_i:`

```
while true
    sekcja_lokalna_i
        k_i = 0
        while k_j != 1
            if czyja_kolej == j
                k_i=1
                do      while czyja_kolej !=i
                    k_i = 0
    sekcja_krytyczna_i
        k_i = 1
        czyja_kolej = j
```

cdh

Rozwiązanie poprawne, wszystkie warunki spełnione

Klasyczne problemy programowania współbieżnego

Wzajemne wykluczanie

Dwa lub więcej procesów współbieżnych wykonuje następujące operacje

P_i:

```
while true
    sekcja_lokalna_i
    protokol_wjsciowy_i
    sekcja_krytyczna_i
    protokol_wjsciowy_i
```

Wymagane własności

- *wzajemne wykluczanie*: żadne dwa procesy nie mogą być równocześnie w swoich sekcjach krytycznych
- zatrzymanie się któregoś z procesów w jego sekcji lokalnej nie może zakłócić działania innych procesów (ich możliwości wchodzenia do ich sekcji krytycznych)

Dwa lub więcej procesów współbieżnych wykonuje następujące operacje

P_i:

```
while true
    sekcja_lokalna_i
    protokol_wjsciowy_i
    sekcja_krytyczna_i
    protokol_wjsciowy_i
```

Wymagane własności - c.d.

- *brak blokady*: nie może nastąpić taka sytuacja, że pewna ilość procesów próbuje wejść do swoich sekcji krytycznych (wykonując protokoły wejściowe), ale żaden z nich ani z pozostałych procesów do niej nie wchodzi.
- *brak zagłodzenia*: jeżeli proces podejmuje próbę wejścia do swojej sekcji krytycznej to w końcu do niej wejdzie
- Jeżeli proces chce wejść do sekcji krytycznej a inne nie, to wejdzie do niej

Algorytm Dekkera rozwiązuje problem wzajemnego wykluczania dla 2 procesów. Rozwiązania ogólne dla n procesów są bardziej skomplikowane i rzadko stosowane. Obejrzymy jeden z nich.

Prostsze rozwiązania powstają przy użyciu bardziej wyspecjalizowanych mechanizmów koordynacji procesów. Tutaj używaliśmy tylko wspólnych zmiennych.

Algorytm piekarniany (bakery algorithm)

opiera się na pomyśle pobierania przez procesy numerków tak jak czasem to jest w niektórych urzędach

wersja dla 2 procesów

$$\begin{array}{cc} P_1 & P_2 \end{array} \quad \begin{array}{l} i=1 \Rightarrow j=2 \\ i=2 \Rightarrow j=1 \end{array}$$

dwie zmienne wspólne n_1 i n_2 o wartości początkowej 0

```
while true
    sekcja_lokalna_i
        n_i = 1
        n_i = n_j + 1
    loop until n_j == 0 or
            n_i < n_j or
            (n_i == n_j and i < j)
    sekcja_krytyczna_i
        n_i = 0
```

Worun (i w dół)

n_1	n_2
0	0
1	1
2	3
0	3
1	0
4	1
6	5

Rozwiązanie poprawne, wszystkie warunki spełnione

Algorytm piekarniany wersja dla n procesów

wspólna tablica `nr[1..n]` o wartościach początkowych 0

wspólna tablica `wybór[1..n]` o wartościach początkowych 0

`P_i:`

```
while true
    sekcja_lokalna_i
    wybór[i] = 1
    nr[i] = 1 + max(nr)
    wybór[i] = 0
    for j = 1 to n
        if j!=i
            loop until wybór[j] == 0
            loop until nr[j] == 0 or
                nr[i] < nr[j] or
                (nr[i] == nr[j] and i<j)
    sekcja_krytyczna_i
    nr[i] = 0
```

Rozwiązańie poprawne, wszystkie warunki spełnione

nie wywieszczeć

Wzajemne wykluczanie wspomagane sprzętowo

operacja sprawdź i przypisz (*test-and-set*)

wspólna zmienna k o wartości początkowej 0

operacja niepodzielna

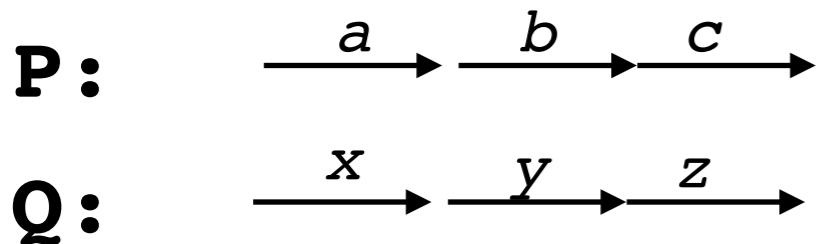
```
tset(l,k,x)
l=k
k=x
```

P_i:

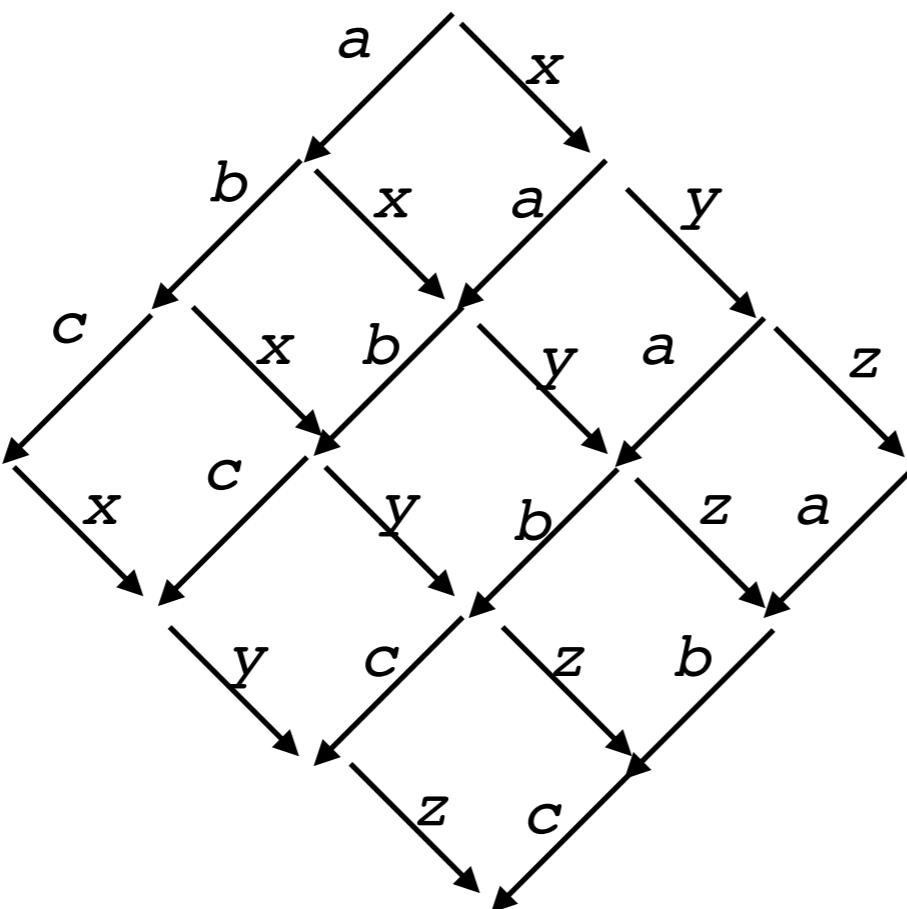
```
while true
    sekcja_lokalna_i
do
    tset(l_i,k,1)
    while l_i != 0
        sekcja_krytyczna_i
        k = 0
```

jest wzajemne wykluczanie, może być zagłodzenie

Obserwacja dotycząca analizowania programów współbieżnych: kombinatoryczna eksplozja stanów



p. 3 akcje, 4 stany



16 = 4² stanów

k procesów, każdy jest sekwencją n akcji:
równoległe wykonanie ma $(n+1)^k$ stanów

Semafora (koncept)

Semafor s jest zmienną całkowitą przyjmującą tylko wartości nieujemne. Zdefiniowane są na nim dwie operacje

$\text{wait}(s)$ jeżeli $s > 0$ to wykonaj $s = s - 1$, w przeciwnym razie wykonywanie procesu który zrobi tę operację wait jest wstrzymane. Taki proces nazywamy *wstrzymanym na semaforze s*

$\text{signal}(s)$ jeżeli są jakieś procesy wstrzymane na semaforze s to wznow jeden z nich, w przeciwnym razie wykonaj $s = s + 1$

operacje $\text{wait}(s)$ $\text{signal}(s)$ są **niepodzielne**

semafor musi mieć nadaną nieujemną początkową wartość

Semafor *binarny* to semafor przyjmujący tylko wartości 0 lub 1. Wtedy w operacji $\text{signal}(s)$ zamiast $s = s + 1$ wykonywane jest $s = 1$

Wzajemne wykluczanie z semaforem

wspólny semafor S o wartości początkowej 1

P_i:

```
while true
    sekcja_lokalna_i
    wait(S)
    sekcja_krytyczna_i
    signal(S)
```

Jest wzajemne wykluczanie

Nie ma blokady

Brak zagłodzenia zależy od sposobu implementacji semaforów

W przypadku dwóch procesów nie ma zagłodzenia

implementacje semafora

Semafor ze zbiorem procesów oczekujących: signal(S) wznawia jeden z procesów oczekujących z ogólnoty

Semafor z kolejką procesów oczekujących: signal(S) wznawia proces oczekujący najdłużej (oczekujące przechowywane są w kolejce)

bez zagi

Semafor z aktywnym oczekiwaniem: signal(S) sprawdza w pętli aktywnego oczekiwania stan zmiennej wait

własności semafora

Semafor jest *silnie uczciwy* jeżeli w sytuacji, gdy od pewnego miejsca działania procesu operacja `Signal(S)` jest wykonywana na semaforze nieskończenie wiele razy to w końcu każdy oczekujący proces zostanie wznowiony i wykona operację `wait(S)` na której był wstrzymany

Semafor jest *słabo uczciwy* jeżeli w sytuacji, gdy od pewnego miejsca działania procesu wartość semafora jest stale większa od zera, to w końcu proces wstrzymany na tym zostanie wznowiony i wykona operację `wait(S)` na której był wstrzymany

Semafor z aktywnym czekaniem i semafor ze zbiorem procesów oczekujących są słabo uczciwe, ale nie są silnie uczciwe

Semafor z kolejką procesów oczekujących jest silnie uczciwy

Propozycja 5. Algorytm Dekkera.

dwie zmienne wspólne k_1 i k_2 o wartości początkowej 1
zmienna wspólna czyja_kolej o wartości początkowej 1

idea: połączenie Propozycji 4 i 1

P_i:

```
while true
    sekcja_lokalna_i
    k_i = 0
    while k_j != 1
        if czyja_kolej == j
            k_i=1
            do      while czyja_kolej !=i
            k_i = 0
    sekcja_krytyczna_i
    k_i = 1
    czyja_kolej = j
```

Rozwiążanie poprawne, wszystkie warunki spełnione

Propozycja 5. Algorytm Dekkera.

dwie zmienne wspólne `k_1` i `k_2` o wartości początkowej 1
zmienna wspólna `czyja_kolej` o wartości początkowej 1

idea: połączenie Propozycji 4 i 1

`P_i:`

```
while true
    sekcja_lokalna_i
        k_i = 0
        while k_j != 1
            if czyja_kolej == j
                k_i=1
                do      while czyja_kolej !=i
                    k_i = 0
```

`sekcja_krytyczna_i`

`k_i = 1`

`czyja_kolej = j`

Rozwiązanie poprawne, wszystkie warunki spełnione

Stwierdzenie. Algorytm Dekkera zapewnia wzajemne wykluczanie

Dowód

$$C_i \rightarrow K_i$$

Przez cały czas wykonywania sekcji krytycznej przez proces P_i zmienna K_i ma wartość 0, bo jest to zapewnione przez P_i a drugi proces nie ma wpływu na wartość tej zmiennej.

Warunkiem wejścia do sekcji krytycznej przez P_1 jest $K_2 == 1$, a więc jeżeli P_2 jest w sekcji krytycznej, to P_1 nie może do niej wejść

P_2 ma symetryczną sytuację

Stwierdzenie. W algorytm Dekkera nie ma blokady.

Dowód Blokada oznaczałaby, że obydwa procesy wykonują zaznaczony kolorem protokół wejściowy przed sekcją krytyczną i żaden z nich nie przechodzi dalej.

W tych instrukcjach nie ma zmiany zmiennej `czyja_kolej`, która w takim razie ma cały czas wartość 1 albo 2.

Weźmy przypadek `czyja_kolej == 1`

Wtedy `P_1` sprawdza w pętli warunek `k_2 != 1`
natomiast `P_2` wykonuje swoje instrukcje

```
if czyja_kolej == 1
    k_2=1
        do while czyja_kolej !=2
```

zatem wykonuje tę wewnętrzną pętle do `while` dając możliwość przejścia do sekcji krytycznej procesowi `P_1` bo `k_2==1`

Przypadek `czyja_kolej == 2` jest symetryczny.
Blokada jest więc niemożliwa.

→ ←

Stwierdzenie. W algorytmie Dekkera nie występuje zagłodzenie żadnego z procesów, czyli jeżeli P_i wejdzie do protokołu wejściowego poprzedzającego sekcję krytyczną, to kiedyś wejdzie do sekcji krytycznej.

wynika z poprośdniego + Zniana 67

Dowód - poprośdzimy krótką informacją o *logice temporalnej*, która w naturalny sposób pojawia się w rozumowaniach o przebiegu działania programów współbieżnych, czy reaktywnych.

alternatywne rozumowanie

naukowane skrótnie,
nie obowiązuje na programie

Logika temporalna - nieformalne wprowadzenie

Działania programu współbieżnego reprezentujemy jako zbiór ciągów przejść do kolejnych stanów obliczenia

2 $s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$

W każdym ze stanów mamy określone wartościowanie zmiennych programu i określone miejsce “gdzie jesteśmy w programie”, czyli można tworzyć formuły postaci

φ : jeżeli P_1 jest przed wejściem do sekcji krytycznej to $k_1 = 0$

i sprawdzać, czy w danym stanie s_i taka formuła φ jest prawdziwa czy nie.

$s_0 \rightarrow s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow \dots$

Dodatkowo, w logice temporalnej mamy operatorы modalne $\diamond \square$ i możemy tworzyć formuły postaci

$\diamond \varphi$ - kiedyś φ będzie prawdziwe

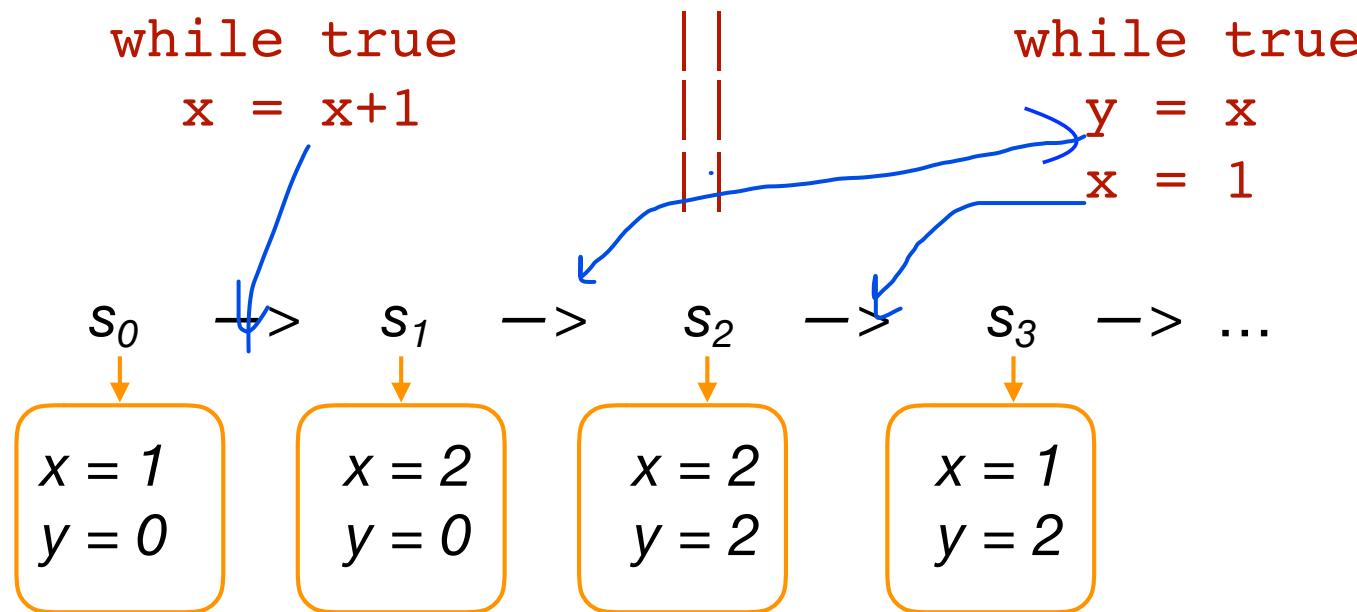
$\square \varphi$ - φ jest zawsze prawdziwe

“kiedyś” i “zawsze” odnoszą się do stanów następujących od bieżącego stanu, łącznie z bieżącym stanem.

Dodatkowo można łączyć takie formuły zwykłymi spójnikami logicznymi

Przykład.

Dwa równolegle działające procesy, początkowo $x = 1, y = 0$



$\Box(\Diamond(x=1))$ - zawsze jest tak, że kiedyś $x=1$

$\Diamond\Box y>0$ - kiedyś będzie tak, że zawsze od tego miejsca $y>0$

$\Box(x=3 \Rightarrow \Diamond y \geq 3)$

Stwierdzenie. W algorytmie Dekkera nie występuje zagłodzenie żadnego z procesów, czyli jeżeli P_i wejdzie do protokołu wejściowego poprzedzającego sekcję krytyczną, to kiedyś wejdzie do sekcji krytycznej.

Dowód

P_i :

```
while true
    sekcja_lokalna_i
        k_i = 0
        while k_j != 1
            if czyja_kolej == j
                k_i=1
                do      while czyja_kolej != i
                    k_i = 0
    sekcja_krytyczna_i
        k_i = 1
        czyja_kolej = j
```

```
P_1:  
while true  
    sekcja_lokalna_1  
    k_1 = 0  
    while k_2 != 1  
        if czyja_kolej == 2  
            k_1 = 1  
            do while czyja_kolej !=1  
            k_1 = 0  
    sekcja_krytyczna_1  
    k_1 = 1  
    czyja_kolej = 2  
  
P_2:  
while true  
    sekcja_lokalna_2  
    k_2 = 0  
    while k_1 != 1  
        if czyja_kolej == 1  
            k_2 = 1  
            do while czyja_kolej !=2  
            k_2 = 0  
    sekcja_krytyczna_2  
    k_2 = 1  
    czyja_kolej = 1
```

P_1:

```
while true
    sekcja_lokalna_1
    k_1 = 0
    while k_2 != 1
        if czyja_kolej == 2
            ✗ k_1 = 1
            do while czyja_kolej != 1
                k_1 = 0
    sekcja_krytyczna_1
    k_1 = 1
    czyja_kolej = 2
```

P_2:

```
while true
    sekcja_lokalna_2
    k_2 = 0
    while k_1 != 1
        if czyja_kolej == 1
            k_2 = 1
            do while czyja_kolej != 2
                k_2 = 0
    sekcja_krytyczna_2
    k_2 = 1
    czyja_kolej = 1
```

1. $\square k_1=0 \wedge \square \text{czyja_kolej}=1 \Rightarrow \diamond \square k_2=1$

jeżeli k_1 ma stałe wartość 0 i czyja_kolej ma stałe wartość 1 to kiedyś wartością k_2 stałe będzie 1

2.P_1:

```
while true
    sekcja_lokalna_1
    k_1 = 0
    while k_2 != 1    //w1
        if czyja_kolej == 2 //w2
            k_1 = 1
            do   while czyja_kolej !=1
            k_1 = 0
    sekcja_krytyczna_1
    k_1 = 1
    czyja_kolej = 2
```

P_2:

```
while true
    sekcja_lokalna_2
    k_2 = 0
    while k_1 != 1
        if czyja_kolej == 1
            k_2 = 1
            do   while czyja_kolej !=2
            k_2 = 0
    sekcja_krytyczna_2
    k_2 = 1
    czyja_kolej = 1
```

$$2. \quad \square k_1=1 \wedge \underline{\square \text{czyja_kolej} = 2} \Rightarrow \diamond \underline{\text{czyja_kolej} = 1}$$

jeżeli k_1 ma stałe wartość 1 i $\underline{\text{czyja_kolej} = 2}$ to kiedyś czyja_kolej będzie miała wartość 1

1. $\square k_1=0 \wedge \square \text{czyja_kolej}=1 \Rightarrow \diamond \square k_2=1$
2. $\square k_1=1 \wedge \square \text{czyja_kolej} = 2 \Rightarrow \diamond \text{czyja_kolej} = 1$

Przypuśćmy, że

3. P_1 nie wchodzi do swojej sekcji krytycznej.

Próbujemy dojść do sprzeczności.

4. $\square \text{czyja_kolej}=2 \Rightarrow \diamond \square k_1=1$ (wynika z 3)
5. $\square \text{czyja_kolej}=2 \Rightarrow \diamond \text{czyja_kolej} = 1$ (wynika z 2 i 4)
6. $\neg \text{czyja_kolej}=2 \Rightarrow \diamond \text{czyja_kolej} = 1$
7. $\diamond \text{czyja_kolej} = 1$ (wynika z 5 i 6)
8. $\diamond \square \text{czyja_kolej} = 1$ $\geq 3,$
9. $\diamond \square (przed(w1) \vee przed(w2))$ ≥ 3
10. $\diamond \square k_1=0$
11. $\diamond \square k_2=1$ (wynika z 8, 10 i 1)

sprzeczność przypuszczenia 3 z punktem 9 i 11.

taki program można próbować sprawdzić mechanicznie (pod względem poprawności)

sygnały

polecenia powłoki kill i trap

Linux

`kill -sygnał id` - wysłanie sygnału do procesu o identyfikatorze `id`

`trap komenda sygnał` - przechwycenie przysłanego sygnału i wykonanie komendy `komenda`

```
Dorotkas-MacBook-Pro:WspolbP pmp$ trap -l
1) SIGHUP  2) SIGINT  3) SIGQUIT 4) SIGILL
5) SIGTRAP 6) SIGABRT 7) SIGEMT  8) SIGFPE
9) SIGKILL 10) SIGBUS  11) SIGSEGV12) SIGSYS
13) SIGPIPE 14) SIGALRM 15) SIGTERM16) SIGURG
17) SIGSTOP 18) SIGTSTP 19) SIGCONT20) SIGCHLD
21) SIGTTIN 22) SIGTTOU 23) SIGIO24) SIGXCPU
25) SIGXFSZ 26) SIGVTALRM27) SIGPROF28) SIGWINCH
29) SIGINFO 30) SIGUSR1 31) SIGUSR2
```

więcej dokumentacji - na przykład:

<http://manpages.ubuntu.com/manpages/trusty/pl/man7/signal.7.html>

Przykład

plik petla.py

```
while True:  
    pass
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 petla.py  
^CTraceback (most recent call last):  
  File "petla.py", line 2, in <module>  
    pass  
KeyboardInterrupt
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$
```

plik petla.py

while True:

pass

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 petla.py &**
[2] 7069

Dorotkas-MacBook-Pro:WspolbP pmp\$ ps

PID	TTY	TIME	CMD
1980	ttys000	0:00.54	-bash
3406	ttys000	0:00.43	emacs ipcql.py
7069	ttys000	0:07.46	/Library/Frameworks/Python.framework/

Dorotkas-MacBook-Pro:WspolbP pmp\$

Dorotkas-MacBook-Pro:WspolbP pmp\$ **kill -SIGINT 7069**

Traceback (most recent call last):

 File "petla.py", line 2, in <module>
 pass

KeyboardInterrupt

Dorotkas-MacBook-Pro:WspolbP pmp\$ ps

PID	TTY	TIME	CMD
1980	ttys000	0:00.55	-bash
3406	ttys000	0:00.43	emacs ipcql.py
[2]-	Interrupt: 2		python3 petla.py

signal – moduł obsługi sygnałów w Pythonie

[https://docs.python.org/3/library signal.html](https://docs.python.org/3/library	signal.html)

signal.signal(signalnum, handler)

Przypisuje funkcję `handler` do obsługi sygnału o numerze `signalnum`, gdzie funkcja

`handler(signum, frame)`

to zdefiniowana przez programistę funkcja obsługi sygnału (wywoływana automatycznie po pojawienniu się sygnału). Parametry (przekazywane przez system) to numer sygnału i ramka stosu

Jako handler mogą być podane specjalne wartości

`signal.SIG_IGN` lub `signal.SIG_DFL`



oznaczające ignorowanie sygnału lub obsługę domyślną

moduł `signal` definiuje też stałe oznaczające numery sygnałów, np.

`signal.SIGUSR1` - sygnał zdefiniowany przez użytkownika

`signal.SIGTERM` - sygnał przerwania z klawiatury `^C`

itd.

(moduł `signal` definiuje też wiele innych rzeczy)

Przykład

plik petlaS.py

```
import signal, os

def handler(signum, frame):
    print('Obsługa sygnału ', signum)

# przypisanie obsługi sygnału do SIGINT
signal.signal(signal.SIGINT, handler)

while True:
    pass
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 petlaS.py
```

```
^CObsługa sygnału 2
```

```
^CObsługa sygnału 2
```

```
^Z
```

```
[2]+ Stopped                  python3 petlaS.py
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ps
```

PID	TTY	TIME	CMD
1980	ttys000	0:00.60	-bash
3406	ttys000	0:00.81	emacs ipcql.py
7423	ttys000	0:24.04	/Library/Frameworks/Python.framework/

```
Dorotkas-MacBook-Pro:WspolbP pmp$ kill -SIGINT 7423
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ps
```

PID	TTY	TIME	CMD
1980	ttys000	0:00.60	-bash
3406	ttys000	0:00.81	emacs ipcql.py
7423	ttys000	0:24.04	/Library/Frameworks/Python.framework/

```
Dorotkas-MacBook-Pro:WspolbP pmp$ kill -9 7423
```

```
[2]+ Killed: 9                  python3 petlaS.py
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ps
```

PID	TTY	TIME	CMD
1980	ttys000	0:00.61	-bash
3406	ttys000	0:00.81	emacs ipcql.py

```
Dorotkas-MacBook-Pro:WspolbP pmp$
```

Przykład, c.d.

plik petlaS1.py

```
import signal, sys

def handler(signum, frame):
    print(' Obsługa sygnału ', signum)

def handler1(signum, frame):
    print(' Inna obsługa sygnału ', signum)
    sys.exit(0)

# przypisanie obsługi sygnału do SIGINT
signal.signal(signal.SIGINT, handler)

# przypisanie obsługi sygnału do SIGUSR1
signal.signal(signal.SIGUSR1, handler1)

while True:
    pass
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 petlaS1.py

^C Obsługa sygnału 2

Obsługa sygnału 2

Inna obsługa sygnału 30

Dorotkas-MacBook-Pro:WspolbP pmp\$

drugie okno:

Dorotkas-MacBook-Pro:WspolbP pmp\$ ps

PID	TTY	TIME	CMD
1980	ttys000	0:00.64	-bash
3406	ttys000	0:01.05	emacs ipcql.py
7822	ttys000	0:34.05	/Library/Frameworks/ Python.framework/
7830	ttys001	0:00.03	-bash

Dorotkas-MacBook-Pro:WspolbP pmp\$ kill -SIGINT 7822

Dorotkas-MacBook-Pro:WspolbP pmp\$ kill -SIGUSR1 7822

Dorotkas-MacBook-Pro:WspolbP pmp\$

Łącza nazwane (kolejki FIFO) (potoki nazwane) z poziomu powłoki

PIPE

`mkfifo nazwa`

`mknod nazwa p` - tworzy kolejkę FIFO

Zapis/odczyt oraz usunięcie odbywają się dla kolejki FIFO tak samo, jak dla zwykłego pliku (mogą to robić wszystkie procesy, które mają odpowiednie prawa dostępu do kolejki). Musi zachodzić *synchronizacja operacji zapisu i odczytu*.

Przykład

`mkfifo kolejka`

`ls -l`

`prw-r--r-- 1 pmp staff 0 Oct 26 23:58 kolejka`

`echo "xxxxx" > kolejka`

`oczekiwanie na synchronizację odczytu`

— drugie okno:

`cat < kolejka`

`xxxxx`

`rm kolejka`

łącza nazwane (kolejki FIFO) (potoki nazwane) z poziomu programu w C

tworzone za pomocą funkcji bibliotecznych

```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *path, mode_t mode);
int mknod(const char *path, mode_t mode, dev_t dev);
```

path - ścieżka do tworzonego łącza (czyli specjalnego pliku)

mode - typ tworzonego pliku i prawa dostępu

dla `mknod()` - `S_IFIFO` i prawa dostępu

dla `mkfifo()` - tylko prawa dostępu, np. 0664

dev - w przypadku typu `S_IFIFO` podajemy tu wartość 0

łącza nazwane (kolejki FIFO) (potoki nazwane) z poziomu programu w Pythonie i modułu bibliotecznego os

tworzone za pomocą funkcji bibliotecznych

```
os.mkfifo(path, mode=438, *, dir_fd=None)
os.mknod(path, mode=384, device=0, *, dir_fd=None)
```

path - ścieżka do tworzonego łącza (czyli specjalnego pliku)
dir_fd deskryptor katalogu względem którego tworzone jest łącze
mode - typ tworzonego pliku i prawa dostępu

dla mknod() - stat.S_IFIFO i prawa dostępu

dla mkfifo() - tylko prawa dostępu, np. 0664

dev - w przypadku typu S_IFIFO podajemy tu wartość 0

usuwanie jak zwykły plik, np.

```
os.unlink(path)
```

Przykład - pisanie do łącza

```
import os
```

```
FIFO = 'kolejka'
```

```
fifo_out = os.open(FIFO, os.O_WRONLY)  
os.write(fifo_out, 'abcde'.encode())
```

Unicode
UTF8

Przykład - czytanie z łącza z pewną sztuczką

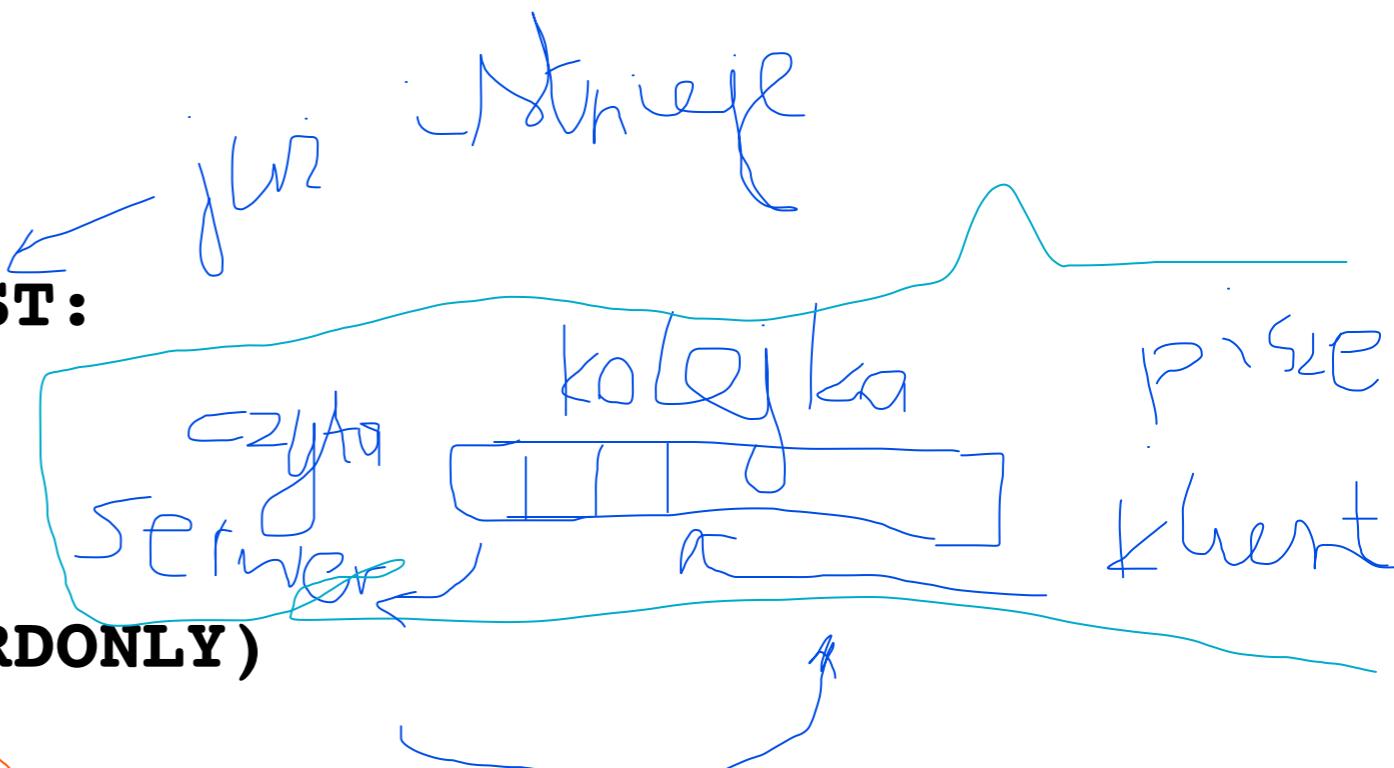
```
import os
import errno
import time

FIFO = 'kolejka'

# utworzenie kolejki
try:
    os.mkfifo(FIFO)
except OSError as oe:
    if oe.errno != errno.EEXIST:
        raise

# kolejka otwarta do odczytu
fifo_in = os.open(FIFO, os.O_RDONLY)

# kolejka otwarta do zapisu,
# żeby zakończenie klienta jej nie zamknął
fifo_out1 = os.open(FIFO, os.O_WRONLY|os.O_NDELAY)
```



Przykład - czytanie z łącza, c.d.

Serwer

```
while True:  
    r = os.read(fifo_in, 2) # czytanie 2 bajtów  
    if len(r)>0:  
        print("Serwer: %s" % r.decode())  
    else:  
        print("Klient skończył")  
        break  
    time.sleep(5) # spowolnienie do testowania
```

Serwer.py

Serwer: ab

- // - cd

- // - e

- // - ab

- // - cd

Klient.py → abcde

Klient.py → abcde
Klient.py → abcde

Zapis/odczyt oraz usunięcie odbywają się dla kolejki FIFO tak samo, jak dla zwykłego pliku (mogą to robić wszystkie procesy, które mają odpowiednie prawa dostępu do kolejki). Musi zachodzić *synchronizacja operacji zapisu i odczytu, czyli:*

- Jeżeli proces próbuje czytać z pustego łącza, to `read()` jest zablokowany aż pojawią się dane do odczytu
- Jeżeli proces próbuje pisać do pełnego łącza, to `write()` jest zablokowany aż odczyty zrobią tyle miejsca, żeby zmieścił się cały zapis

flaga O_NDELAY w funkcji open

wpływ na zachowanie funkcji `read()` i `write()`

jeżeli jest ustawiona:

`read()` - przy próbie odczytu bez otwartej kolejki do zapisu zwróci wartość 0 oznaczającą koniec pliku.

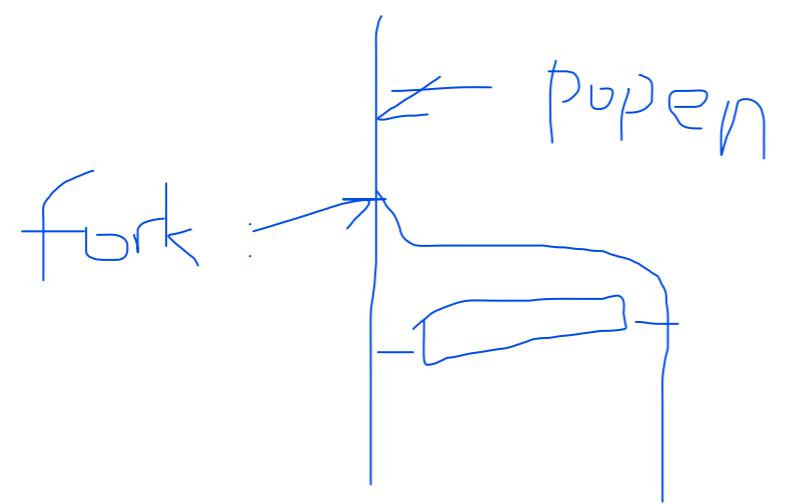
`write()` - przy próbie zapisu jeżeli żaden proces nie jest podłączony do odczytu spowoduje wygenerowanie sygnału SIGPIPE co doprowadzi do zakończenia działania procesu o ile sygnał nie zostanie obsłużony

ilość danych zawartych w kolejce FIFO jest ograniczona przez system operacyjny

Kiedy kolejka nie jest skojarzona z żadnym procesem, jej zawartość zostaje wyzerowana przez system.

Wszystkie procesy zaangażowane w komunikację muszą znać nazwy kolejek przed rozpoczęciem komunikacji

Nie można zidentyfikować procesu zapisującego dane (chyba, że zapisze to w danych)



oprócz potoków nazwanych istnieją też potoki nienazwane, tworzone zwykle do komunikacji między procesem rodzicielskim i potomnym przed rozgałęzieniem procesu (funkcja `popen()`)

w Pythonie można operować na kolejkach przez funkcje Pythona: `open`, `read`, `write`. Pozwala to na przesyłanie/odczytywanie linijek tekstu.

Komunikacja międzyprocesowa (Interprocess communications) (IPC)

POSIX



- kolejki komunikatów
- semafory
- pamięć wspólna

obiekt IPC

- musi zostać utworzony przed użyciem
- ma swojego właściciela i prawa dostępu
- polecenie systemowe `ipcs -b` wyświetla informacje o aktualnych obiektach IPC
- polecenie systemowe `ipcrm` (z odpowiednimi flagami i parametrami) usuwa obiekty IPC
(np. `ipcrm -q 235` usunie kolejkę o identyfikatorze 235)

Dorotkas-MacBook-Pro:WspolbP pmp\$ **ipcs**
IPC status from <running system> as of Wed Nov 10 07:07:41 CET 2021
T ID KEY MODE OWNER GROUP

Message Queues:

T ID KEY MODE OWNER GROUP

Shared Memory:

T ID KEY MODE OWNER GROUP

Semaphores:

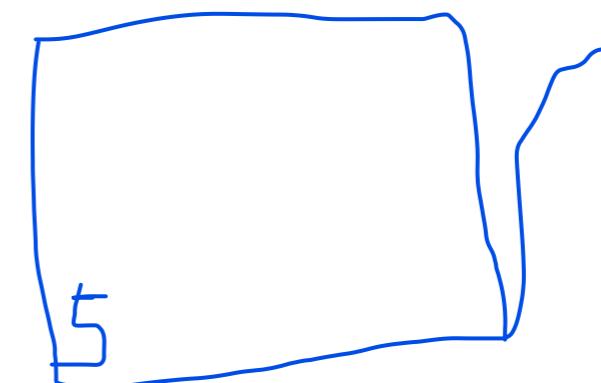
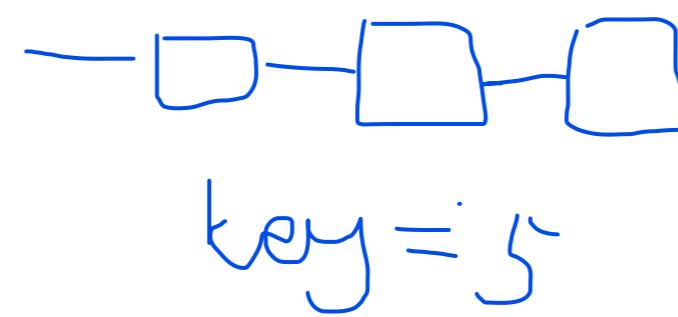
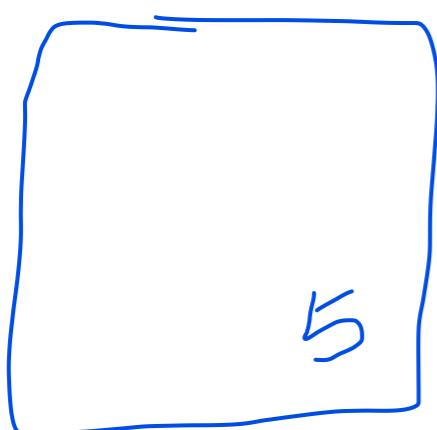
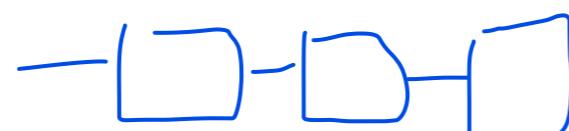
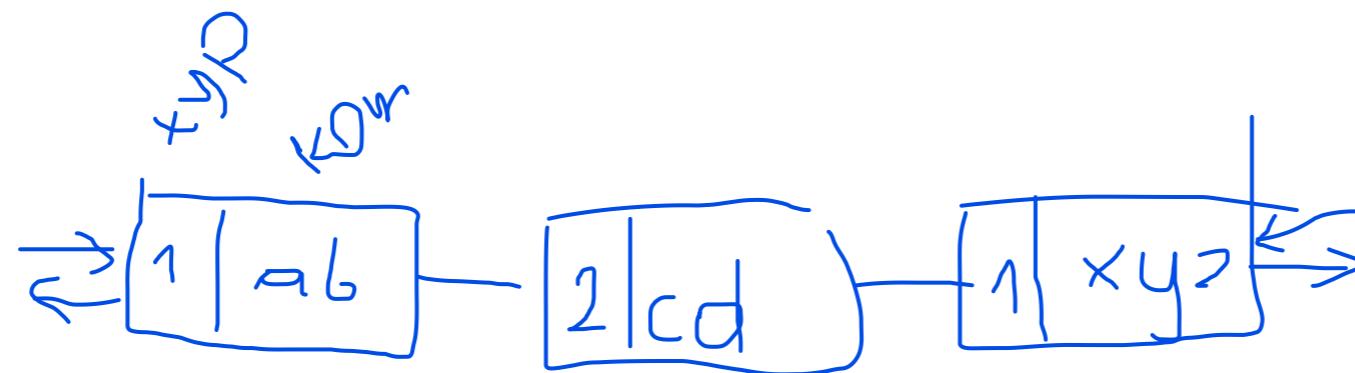
Dorotkas-MacBook-Pro:WspolbP pmp\$

kolejki komunikatów - wersja w C/C++

Komunikaty posiadają pewną strukturę (nie są tylko ciągami bajtów, jak w przypadku łącz); dokładniej komunikat jest strukturą, której pierwsze pole musi mieć typ `long`, np.

```
struct msghdr {  
    long mtype ;  
    ...  
}
```

Wartość pola `mtype` pozwala rozróżnić komunikaty z różnych źródeł i wpływa na kolejność ich pobierania z kolejki



Tworzenie kolejki komunikatów

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget(key_t key, int msgflg)
```

`key` to tzw. klucz kolejki czyli wartość (liczba całkowita nieujemna) identyfikująca kolejkę w systemie;
zalecane było generowanie kluczy przez funkcję `ftok`; (zdezaktualizowane) użycie w tym miejscu stałej `IPC_PRIVAT` powoduje wygenerowanie przez system unikatowej wartości identyfikatora kolejki w systemie

`msgflg` to flagi zawierające informację o prawach dostępu plus ewentualnie wartości `IPC_CREAT`, `IPC_EXCL`

zwracany wynik to identyfikator kolejki lub -1 gdy wystąpił błąd

wysyłanie komunikatu

```
int msgsnd(int ident, struct msghdr *komunikat,  
          int rozmiar, int flagi)
```

wstawia komunikat wraz z podanym typem na koniec kolejki
zwraca 0 w przypadku sukcesu, -1 w przypadku błędu.

ident - identyfikator kolejki (zwrócony przez msgget)

komunikat - wskaźnik do struktury przechowującej typ komunikatu i sam
komunikat (bufora komunikatu)

rozmiar - rozmiar komunikatu nie licząc typu (!)

flagi - 0 lub **IPC_NOWAIT** - decydują, czy w sytuacji przeppełnienia kolejki
proces ma być zawieszony

pobieranie komunikatu

```
int msgrcv (int ident; struct msghdr *komunikat,  
            int rozmiar, long mtype, int flagi);
```

pobiera z kolejki najdawniej wstawiony komunikat:

- o danym typie, jeżeli $mtype > 0$ (i jeżeli taki komunikat istnieje),
- o dowolnym typie, jeżeli $mtype == 0$ (i jeżeli taki komunikat istnieje)
- o typie mniejszym lub równym wartości bezwzględnej z $mtype$, jeżeli $mtype < 0$

zwraca liczbę faktycznie pobranych bajtów z kolejki w przypadku sukcesu, a -1 w przypadku błędu.

ident - identyfikator kolejki

komunikat - wskaźnik do bufora komunikatu

rozmiar - rozmiar struktury komunikatu (nie licząc typu)

mtype - typ komunikatu, jaki chcemy pobrać z kolejki (może być 0)

flagi - można ustawić **IPC_NOWAIT** i / lub **MSG_NOERROR** (powoduje odpowiednie zachowanie, jeśli komunikat jest większy, niż przewiduje rozmiar

regulowanie kolejki

```
int msgctl ( int ident, int polecenie,  
             struct msgqid_ds *struktura );
```

ident - identyfikator kolejki

polecenie - kod czynności do wykonania na strukturze kontrolnej kolejki

struktura - wskaźnik do bufora struktury kontrolnej

Może wykonywać różne czynności na kolejce, np. zmieniać prawa dostępu, odczytywać informacje o ostatnio wykonanej operacji na kolejce itp.

Zwraca 0 w przypadku sukcesu -1 w przypadku błędu.

Najczęściej jest wykorzystywana do usunięcia kolejki:

```
msgctl ( ident, IPC_RMID, 0 );
```

Kolejki komunikatów IPC, wersja w Pythonie

https://semanchuk.com/philip/sysv_ipc/

Tworzenie kolejki komunikatów

```
MessageQueue(key, [flags = 0, [mode = 0600,  
[max_message_size = 2048]]]
```

Tworzy nową kolejkę komunikatów lub otwiera istniejącą.

key może być **None**, **IPC_PRIVATE** lub **int > 0 (ale ≤ KEY_MAX)**.
Jeżeli **key** jest **None**, wybierany jest losowy niezajęty numer.

flags specyfikują, czy będzie otwierana nowa kolejka, czy już istniejąca:

- 0 oznacza już istniejącą, powoduje **ExistentialError** jeżeli nie istnieje.
- **IPC_CREAT** oznacza stworzenie nowej lub otwarcie już istniejącej
- **IPC_CREX** (czyli **IPC_CREAT | IPC_EXCL**), tworzy nową, powoduje **ExistentialError** jeżeli już istnieje.

max_message_size daje możliwość zwiększenie domyślnego ograniczenia rozmiaru komunikatu

```
send(message, [block = True, [type = 1]])
```

wstawia komunikat do kolejki.

message powinno być obiektem bajtowym

block specyfikuje, czy wysyłanie jest blokujące (gdy kolejka jest pełna) czy zgłosi błąd **BusyError**

False oznacza brak blokowania, True oznacza blokowanie

type jest typem przypisany komunikatowi (> 0).

```
receive([block = True, [type = 0]])
```

odbiera komunikat zwracając parę (**message**, **type**).

block specyfikuje, czy odbierania jest blokujące (gdy nie ma komunikatu danego typu) czy zgłosi błąd **BusyError**

False oznacza brak blokowania, **True** oznacza blokowanie

type jest typem odbieranego komunikatu

- 0 oznacza odbieranie pierwszego komunikatu z kolejki, nie zważając na typ,
- > 0, pobiera pierwszy komunikat danego typu
- < 0, pobiera pierwszy komunikat typu mniejszego lub równego **type** co do wartości bezwzględnej

remove()

usuwa kolejkę komunikatów.

Proces tworzący kolejkę

```
import sysv_ipc
```

plik ipcS.py

```
klucz=11
```

```
mq = sysv_ipc.MessageQueue(klucz, sysv_ipc.IPC_CREAT)
```

```
for i in range(0, 3):
    s, t = mq.receive(True,1)
    s = s.decode()
    print("Serwer: odbrałem %s    " % s)
    s, t = mq.receive(True,2)
    s = s.decode()
    print("Serwer: odbrałem %s    " % s)
```

```
mq.remove()
```

Procesy otwierające utworzoną kolejkę

```
import sysv_ipc
```

plik ipcK1.py

```
klucz=11
s='aaa'
mq = sysv_ipc.MessageQueue(klucz)

for i in range(0, 3):
    mq.send(s.encode(),True,1)
    print('1 wysyła')
    1
```

```
import sysv_ipc
```

plik ipcK2.py

```
klucz=11
s='bb'
mq = sysv_ipc.MessageQueue(klucz)

for i in range(0, 3):
    mq.send(s.encode(),True,2)
    print('2 wysyła')
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 ipcS.py
```

```
^Z
```

```
[2]+ Stopped python3 ipcS.py
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcs -b
```

```
IPC status from <running system> as of Wed Nov 10 07:31:46 CET 2021
```

T	ID	KEY	MODE	OWNER	GROUP	QBYTES
---	----	-----	------	-------	-------	--------

Message Queues:

q	655360	0x0000000b	--rw-----	pmp	staff	2048
---	--------	------------	-----------	-----	-------	------

T	ID	KEY	MODE	OWNER	GROUP	SEGSZ
---	----	-----	------	-------	-------	-------

Shared Memory:

T	ID	KEY	MODE	OWNER	GROUP	NSEMS
---	----	-----	------	-------	-------	-------

Semaphores:



```
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcrm -q 655360
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcs -b
```

```
IPC status from <running system> as of Wed Nov 10 07:32:53 CET 2021
```

T	ID	KEY	MODE	OWNER	GROUP	QBYTES
---	----	-----	------	-------	-------	--------

Message Queues:

T	ID	KEY	MODE	OWNER	GROUP	SEGSZ
---	----	-----	------	-------	-------	-------

Shared Memory:

T	ID	KEY	MODE	OWNER	GROUP	NSEMS
---	----	-----	------	-------	-------	-------

Semaphores:

```
Dorotkas-MacBook-Pro:WspolbP pmp$ fg
```

```
python3 ipcS.py
```

```
Traceback (most recent call last):
```

```
  File "ipcS.py", line 8, in <module>
```

```
    s, t = mq.receive(True,1)
```

```
sysv_ipc.ExistentialError: The queue no longer exists
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 ipcS.py**

Serwer: odebrałem aaa

Serwer: odebrałem bb

Serwer: odebrałem aaa

Serwer: odebrałem bb

Serwer: odebrałem aaa

Serwer: odebrałem bb

Dorotkas-MacBook-Pro:WspolbP pmp\$

(1,abab) (1,bbb) (1,aaa) (2,bb) (2,bb) (2,bb)

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 ipcK1.py**

1 wysyła

1 wysyła

1 wysyła

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 ipcK2.py**

2 wysyła

2 wysyła

2 wysyła

Dorotkas-MacBook-Pro:WspolbP pmp\$

Proces odbierający z kolejki z typem komunikatów 0

```
import sysv_ipc
```

plik ipcS1.py

```
klucz=11
```

```
mq = sysv_ipc.MessageQueue(klucz, sysv_ipc.IPC_CREAT)
```

```
for i in range(0, 3):
```

```
    s, t = mq.receive(True, 0)
```

```
    s = s.decode()
```

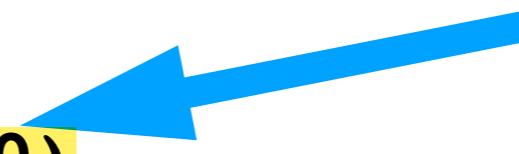
```
    print("Serwer: odebrałem %s    "% s)
```

```
    s, t = mq.receive(True, 0)
```

```
    s = s.decode()
```

```
    print("Serwer: odebrałem %s    "% s)
```

```
mq.remove()
```



```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 ipcS1.py
```

```
Serwer: odebrałem aaa
```

```
Serwer: odebrałem aaa
```

```
Serwer: odebrałem aaa
```

```
Serwer: odebrałem bb
```

```
Serwer: odebrałem bb
```

```
Serwer: odebrałem bb
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 ipcK1.py
```

```
1 wysyła
```

```
1 wysyła
```

```
1 wysyła
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 ipcK2.py
```

```
2 wysyła
```

```
2 wysyła
```

```
2 wysyła
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$
```

<https://pypi.org/project/ipcqueue/> (wersja System V a nie POSIX)

class ipcqueue.sysvmsg.Queue(key=None, max_bytes=None)

close()

get(block=True, msg_type=0)

put(item, block=True, msg_type=1, pickle_protocol=1)

Komunikacja międzyprocesowa (Interprocess communications) (IPC)

- kolejki komunikatów
- semafory
- pamięć wspólna

obiekt IPC

- musi zostać utworzony przed użyciem
- ma swojego właściciela i prawa dostępu
- polecenie systemowe `ipcs -b` wyświetla informacje o aktualnych obiektach IPC
- polecenie systemowe `ipcrm` (z odpowiednimi flagami i parametrami) usuwa obiekty IPC

pamięć współdzielona (shared memory)

Poza własnym segmentem danych, przydzielonym w momencie utworzenia, proces może mieć przydzielone w trakcie wykonywania obszary pamięci z przestrzeni dostępnej wszystkim procesom. (Rozmiar tej przestrzeni jest ograniczony przez system.)

wersja w C

Przegląd bez wdrożenia

w szczegóły

Tworzenie segmentu pamięci wspólnej

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
```

```
int shmget(key_t key, size_t size, int shmflg)
```

Zwraca identyfikator segmentu pamięci w przypadku sukcesu a w przypadku błędu wartość -1

Nowy segment, o rozmiarze równym wartości parametru **size** zaokrąglonym w górę do wielokrotności PAGE_SIZE, zostanie utworzony jeżeli

- parametr **key** będzie mieć wartość IPC_PRIVATE lub
- będzie mieć inną wartość oraz segment skojarzony z key nie istnieje, a w parametrze **shmflg** zostanie przekazany znacznik IPC_CREAT.

(parametr **size** jest nieistotny, jeśli segment już istnieje)

shmflg to flagi zawierające informację o prawach dostępu plus ewentualnie wartości IPC_CREAT, IPC_EXCL

Jeżeli w parametrze `shmflg` podano zarówno `IPC_CREAT`, jak i `IPC_EXCL` oraz już istnieje segment w pamięci współdzielonej o kluczu `key`, to `shmget()` kończy się błędem, ustawiając `errno` na wartość `EEXIST`.

Dołączanie segmentu pamięci do przestrzeni adresowej procesu

```
void *shmat(int shmid, const void *shmaddr, int shmflg)
```

shmid - identyfikator segmentu (zwrócony przez funkcję `shmget`)

shmaddr - wskaźnik do miejsca, gdzie programista *proponuje* dołączyć segment (może być 0)

shmflg - dodatkowe flagi (na przykład mogą nakazać dołączenie segmentu tylko do odczytu)

Dołącza segment (i zwiększa jego licznik dowiązań o 1) pod podanym adresem (w miarę możliwości) jeśli adres jest większy od 0, albo pod adresem wybranym przez system, jeśli podany adres jest równy 0 (najczęściej stosowane postępowanie).

Zwraca wskaźnik do miejsca, gdzie rzeczywiście został dołączony segment - w przypadku sukcesu, a w przypadku błędu wartość -1

Utworzenie pamięci współdzielonej w jednym programie i zapis do niej

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define klucz 10

int pamiec;
char *adres;

main(){
    pamiec=shmget(klucz,256,0700|IPC_CREAT);
    adres=shmat(pamiec,0,0);
    strcpy(adres,"aaaaaa");
}
```

Zwykła alokacja pamięci

```
char *adres;  
  
main(){  
    adres=(char*) malloc(10*sizeof(char));  
    strcpy(adres,"aaaaaa");  
  
    printf("odczytane: %s\n",adres);  
    free(adres);  
}
```

Odłączanie segmentu pamięci z przestrzeni adresowej procesu

```
int shmdt(const void *shmaddr)
```

Wyłącza segment pamięci wspólnej odwzorowany pod adresem podanym w `shmaddr` z przestrzeni adresowej procesu wywołującego tę funkcję.

Licznik dowiązań odłącznego segmentu jest zmniejszany o 1. Jeżeli stan licznika dowiązań zmniejszył się w wyniku tego do 0, a segment był oznaczony do usunięcia, w tym momencie następuje jego usunięcie z tablicy segmentów.

Przekazany funkcji w parametrze `shmaddr` adres musi być równy adresowi zwróconemu wcześniej przez wywołanie `shmat`

Zwraca 0 w przypadku sukcesu, a w przypadku błędu wartość -1

Regulowanie pamięci wspólnej

```
int shmctl(int shmid, int polecenie, struct shmid_ds *struktura)
```

shmid - identyfikator segmentu (zwrócony przez funkcję `shmget`)

polecenie - kod polecenia do wykonania na strukturze zarządzającej segmentem

struktura - wskaźnik do bufora struktury zarządzającej segmentem

Działanie podobnie, jak w przypadku kolejek komunikatów, może wykonywać wiele różnych czynności, a najczęściej wykonywaną jest *oznaczenie* segmentu do usunięcia:

```
shmctl(ident, IPC_RMID, 0)
```

Uwaga. Zalecane jest odłączenie segmentu (wykonanie funkcji `shmdt`) przed oznaczeniem segmentu

Zwraca 0 w przypadku sukcesu, a w przypadku błędu wartość -1

pamięć współdzielona IPC, wersja w Pythonie

https://semanchuk.com/philip/sysv_ipc/

```
SharedMemory(key, [flags = 0, [mode = 0600, [size = 0 or PAGE_SIZE,  
[init_character = ' ']]]])
```

Tworzy nowy segment pamięci lub otwiera istniejący. Nowy segment jest automatycznie dołączony

key, flags, mode - jak w kolejkach IPC

size:

- przy otwieraniu istniejącego, nie może być większe od jego rozmiaru (można podać 0)
- przy otwieraniu nowego powinno być większe od 0, może być zaokrąglone w górę do wielokrotności PAGE_SIZE.

import sysv_ipc

ipcM1.py

klucz = 11

NULL_CHAR = '\0'

mem = sysv_ipc.SharedMemory(klucz, sysv_ipc IPC_CREAT)

def pisz(mem, s):

s += NULL_CHAR

s = s.encode()

mem.write(s)

zapis

s = 'aaaaaa'

pisz(mem, s)

```
read([byte_count = 0, [offset = 0]])
```

Czyta nie więcej niż **byte_count** bajtów z segmentu pamięci zaczynając od **offset** i zwraca je jako obiekt bajtowy. W przypadku gdy **byte_count** jest zero (domyślne) zwraca cały bufor

```
write([bytes = 0, [offset = 0]])
```

Wpisuje obiekt bajtowy **bytes** do segmentu pamięci zaczynając od **offset**

```
detach()
```

Odłącza proces od współdzielonej pamięci

```
remove()
```

usuwa współdzieloną pamięć, ale dopiero gdy wszystkie używające procesy się od niej odłączyły

```
import sysv_ipc  
  
klucz = 11  
NULL_CHAR = '\0'  
  
mem = sysv_ipc.SharedMemory(klucz)  
  
def czytaj(mem):  
    s = mem.read()  
    s = s.decode()  
    i = s.find(NULL_CHAR)  
    if i != -1:  
        s = s[:i]  
    return s  
  
s = ''  
s=czytaj(mem)  
print(s)  
sysv_ipc.remove_shared_memory(mem.id)
```

ipc M2.py

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 ipcM1.py
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcs
```

```
IPC status from <running system> as of Wed Nov 17 07:47:32 CET 2021
```

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Message Queues:

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Shared Memory:

m	786432	0x0000000b	--rw-----	pmp	staff
---	--------	------------	-----------	-----	-------

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Semaphores:

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 ipcM2.py
```

```
aaaaa
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ **ipcs**

IPC status from <running system> as of Wed Nov 17 07:47:50 CET 2021

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Message Queues:

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Shared Memory:

T	ID	KEY	MODE	OWNER	GROUP
---	----	-----	------	-------	-------

Semaphores:

Semafora (koncept)

Semafor s jest zmienną całkowitą przyjmującą tylko wartości nieujemne. Zdefiniowane są na nim dwie operacje

`wait(s)` jeśli $s > 0$ to wykonaj $s = s - 1$, w przeciwnym razie wykonywanie procesu który zrobi tę operację `wait` jest wstrzymane. Taki proces nazywamy *wstrzymanym na semaforze s*

`signal(s)` jeśli są jakieś procesy wstrzymane na semaforze s to wznow jeden z nich, w przeciwnym razie wykonaj $s = s + 1$

operacje `wait(s)` `signal(s)` są *niepodzielne*

semafor musi mieć nadaną nieujemną początkową wartość

Semafora (realizacja w IPC) w C

Występują nie jako pojedyncze obiekty, ale jako elementy tablic semaforów, na których można wykonywać jednocześnie (niepodzielne) operacje. Maksymalny rozmiar tablicy semaforów zależy od ustawień systemowych. Zakres wartości przyjmowanych przez pojedynczy semafor jest zakresem wartości typu ushort (czyli od 0 do 255).

```
int semget(key_t klucz, int liczbasem, int flagi)
          ^----->
int semop(int semid, struct sembuf *sops, unsigned nsops)

int semctl(int ident, int numer, int polecenie,
           union semun argument)
```

Tworzenie semaforów

```
int semget(key_t klucz, int liczbasem, int flagi)
```

Tworzy nową tablicę semaforów, jeśli wcześniej nie istniała

klucz, flagi - jak dla kolejek komunikatów i pamięci dzielonej
liczbasem - liczba semaforów w tablicy (argument nieistotny, jeśli tablica już istnieje)

Zwraca identyfikator tablicy semaforów w przypadku sukcesu, a w przypadku błędu wartość -1

Operacje na semaforach

```
int semop(int semid, struct sembuf *sops, unsigned nsops)
```

gdzie **struct sembuf {**

ushort sem_num;	numer semafora w tablicy
short sem_op;	operacja na semaforze
short sem_flg; }	flagi operacji

Zwraca 0 w przypadku sukcesu, -1 w przypadku błędu.

semid - identyfikator tablicy semaforów (zwrócony przez funkcję `semget`)

sops - wskaźnik do początku tablicy operacji (tablicy struktur `sembuf`)

nsops - liczba elementów w tablicy wskazywanej przez **sops**

Działanie: system wykonuje niepodzielnie wszystkie operacje nakazane w tablicy struktur wskazywanej przez **sops** - albo nie wykonuje żadnej, jeśli choć jedna z nich jest w danej chwili niemożliwa.

Pojedyncza operacje na semaforze

```
struct sembuf{  
    ushort sem_num;           numer semafora w tablicy  
    short sem_op;             operacja na semaforze  
    short sem_flg; }         flagi operacji
```

jeżeli wartość `sem_op` jest dodatnia, wartość semafora zwiększa się o nią (może wzrosnąć o więcej, niż 1), a jednocześnie jest budzona odpowiednia liczba procesów śpiących pod tym semaforem (jeśli są takie)

jeżeli wartość `sem_op` jest ujemna, wartość semafora odpowiednio się zmniejsza, jeśli to możliwe, a jeśli niemożliwe, zmniejszenie nie jest wykonywane, a proces albo zasypia czekając na zaistnienie takiej możliwości, albo od razu następuje powrót z funkcji z błędem (w zależności od ustawienia flagi `IPC_NOWAIT`)

jeżeli wartość `sem_op` wynosi zero, proces zasypia, jeśli wartość semafora nie jest zerem (lub wraca od razu z błędem, jeśli jest ustawiona flaga `IPC_NOWAIT`) i budzi się dopiero po osiągnięciu wartości zero przez semafor.

Regulowanie semaforów

```
int semctl (int ident, int numer, int polecenie,  
            union semun argument)
```

gdzie

```
union semun{
```

int val;	do ustawienia wartości pojedynczego semafora
struct semid_ds *buf;	bufor struktury zarządzającej tablicą semaforów
ushort *array;	wskaźnik do tablicy ustawień wartości całej tablicy sem.
struct seminfo *__buf;	specyficzne dla Linuxa, używane przez
void *__pad;	jądro systemu operacyjnego

Zwraca liczbę dodatnią będącą wynikiem wykonania polecenia - w przypadku sukcesu albo -1 w przypadku błędu

ident - identyfikator tablicy semaforów (zwrócony przez funkcję `semget`)

numer - numer semafora w tablicy (istotny w przypadku, gdy polecenie dotyczy pojedynczego semafora)

polecenie - kod polecenia do wykonania

argument - argument jednego z typów wchodzących w skład unii, zależnego od polecenia

Najczęściej używanymi poleceniami są:

IPC_RMID

GETALL

SETALL

GETVAL

SETVAL

usunięcie tablicy semaforów

odczytanie wartości wszystkich semaforów w tablicy

nadanie wartości wszystkim semaforom w tablicy

odczytanie wartości pojedynczego semafora

nadanie wartości pojedynczemu semaforowi

semafory IPC, wersja w Pythonie

https://semanchuk.com/philip/sysv_ipc/

```
Semaphore(key, [flags = 0, [mode = 0600, [initial_value = 0]]])
```

Tworzy nowy semafor (obiekt) lub otwiera istniejący. Nowy segment jest automatycznie dołączony

key, flags, mode - jak w kolejkach IPC

initial_value - wartość początkowa

Wait

```
acquire([timeout = None, [delta = 1]])
```

Czeka aż wartość semafora będzie > 0, zmniejsza wartość semafora o **delta**

Signal

```
release([delta = 1])
```

Zwalnia semafor zwiększając wartość semafora o **delta**


```
import sysv_ipc
```

```
klucz = 11
```

```
NULL_CHAR = '\0'
```

```
sem1 = sysv_ipcSemaphore(klucz, sysv_ipc IPC_CREX, 0o700, 0)
sem2 = sysv_ipcSemaphore(klucz+1, sysv_ipc IPC_CREX, 0o700, 1)
mem = sysv_ipcSharedMemory(klucz, sysv_ipc IPC_CREX)
```

```
def pisz(mem, s):
    s += NULL_CHAR
    s = s.encode()
    mem.write(s)
```

```
def czytaj(mem):
    s = mem.read()
    s = s.decode()
    i = s.find(NULL_CHAR)
    if i != -1:
        s = s[:i]
    return s
```

k0m 1.py

poč2.

```
s = ''  
pisz(mem,s)  
  
for i in range(0, 3):  
    print(s)      czeka  
    sem1.acquire()  
    s=czytaj(mem)  
    s=s+'a'  
    pisz(mem,s)  
    sem2.release() → 1
```

```
print(s)  
sem1.acquire()  
sysv_ipc.remove_shared_memory(mem.id)  
sysv_ipc.remove_semaphore(sem1.id)  
sysv_ipc.remove_semaphore(sem2.id)  
# możliwa alternatywnie mem.remove() i  
# sem1.remove() i sem2.remove()
```

```
klucz = 11  
NULL_CHAR = '\0'
```

kom 2.py

```
sem1 = sysv_ipc.Semaphore(klucz)  
sem2 = sysv_ipc.Semaphore(klucz+1)  
mem = sysv_ipc.SharedMemory(klucz)
```

.....

```
s = ''
```

```
for i in range(0, 3):  
    print(s)  
    sem2.acquire() → 0  
    s=czytaj(mem)  
    s=s+'b'  
    pisz(mem,s)  
    sem1.release() → 1  
  
print(s)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 kom1.py
```

^Z

[2]+ Stopped

python3 kom1.py

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcs -b
```

IPC status from <running system> as of Wed Nov 17 09:33:52 CET 202

T	ID	KEY	MODE	OWNER	GROUP	QBYTES
---	----	-----	------	-------	-------	--------

Message Queues:

T	ID	KEY	MODE	OWNER	GROUP	SEGSZ
---	----	-----	------	-------	-------	-------

Shared Memory:

m	851968	0x0000000b	--rw-----	pmp	staff	4096
---	--------	------------	-----------	-----	-------	------

T	ID	KEY	MODE	OWNER	GROUP	NSEMS
---	----	-----	------	-------	-------	-------

Semaphores:

s	720896	0x0000000b	--ra-----	pmp	staff	1
---	--------	------------	-----------	-----	-------	---

s	720897	0x0000000c	--ra-----	pmp	staff	1
---	--------	------------	-----------	-----	-------	---

```
Dorotkas-MacBook-Pro:WspolbP pmp$ fg  
python3 kom1.py
```

^C

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 kom1.py  
Traceback (most recent call last):  
  File "kom1.py", line 6, in <module>  
    sem1 = sysv_ipc.Semaphore(klucz, sysv_ipc.IPC_CREAT, 0o700, 0)  
sysv_ipc.ExistentialError: A semaphore with the specified key  
already exists
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcrm -s 720896  
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcrm -s 720897  
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcrm -m 720896  
ipcrm: shmid(720896): : Invalid argument  
Dorotkas-MacBook-Pro:WspolbP pmp$ ipcrm -m 851968  
Dorotkas-MacBook-Pro:WspolbP pmp$
```

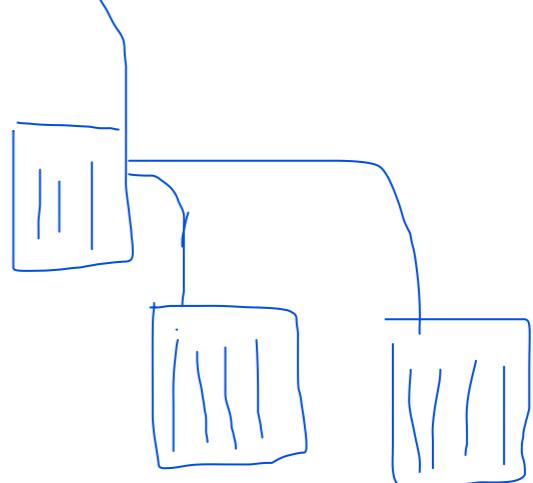
Kom2.py

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 kom2.py  
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 kom1.py  
  
ba  
bab  
babab  
bababa Dorotkas-MacBook-Pro:WspolbP pmp$  
Dorotkas-MacBook-Pro:WspolbP pmp$
```

1
kom1.py

Wątki współbieżne

Wątki a procesy



Wątek funkcjonuje w obrębie procesu i używa jego zasoby

Ma swój niezależny przepływ sterowania

Duplikuje tylko te nieliczne zasoby procesu, które są potrzebne do niezależnego działania

Mожет współdzielić zasoby procesu z innymi wątkami

Ginie, gdy jego proces otaczający znika

Wymaga znacznie mniejszego zaangażowania ze strony systemu operacyjnego do utworzenia, niż zwykły proces

w C: wątki POSIX, czyli biblioteka pthread

w Pythonie: moduły

- _thread

https://docs.python.org/3.8/library/_thread.html#module-thread

- **threading (tego użyjemy)**

<https://docs.python.org/3/library/threading.html>

Global Interpreter Lock (GIL) w CPython



mechanizm powodujący, że **tylko jeden wątek** może być w danym momencie wykonywany przez interpreter Pythona, nawet w architekturze wielordzeniowej. Wątki współbieżne są więc szeregowane przez podział czasu

Inne wersje Pythona (Jython, IronPython and PyPy napisane w Javie, C#, Pythonie) nie mają tego ograniczenia

Alternatywne formy współbieżności:

- **multiprocessing**
- `concurrent.futures.ProcessPoolExecutor`

tworzenie wątku współbieżnego

```
class threading.Thread(group=None, target=None,  
                      name=None, args=(), kwargs={}, *, daemon=None)
```

(wywoływany z argumentami kluczowymi, nie pozycyjnymi)

group - nieużywany

target - funkcja, którą wątek ma wykonać

name - nazwa wątku

args - lista argumentów pozycyjnych dla funkcji **target**

kwargs - słownik argumentów kluczowych dla funkcji **target**

deamon - informacja, czy ma być demonem

Tworzy nowy wątek (obiekt) działający współbieżnie z wątkiem rodzicielskim i wykonujący wskazaną funkcję.

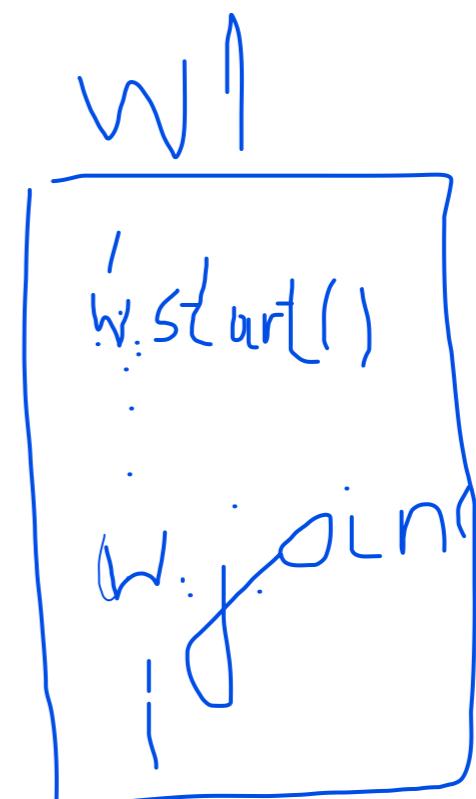
uruchamianie wątku

start()

wcielanie wątku

join(timeout=None**)**

timeout - czas po którym oczekiwanie zostanie przerwane



Projektad:

blik:
threading.py

```
import threading  
import time
```

```
def f(arg, name, s):  
    for i in range(arg):  
        print(name, 'i=', i)  
        time.sleep(s)
```

0 1 2 3 4

```
t1 = threading.Thread(target = f, args = (5, 't1', 1))  
t2 = threading.Thread(target = f, args = (5, 't2', 0.7))  
t1.start()  
t2.start()  
t1.join()  
t2.join()  
print ("koniec")
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 threadingP.**

t1 i= 0

t2 i= 0

t2 i= 1

t1 i= 1

t2 i= 2

t1 i= 2

t2 i= 3

t2 i= 4

t1 i= 3

t1 i= 4

koniec

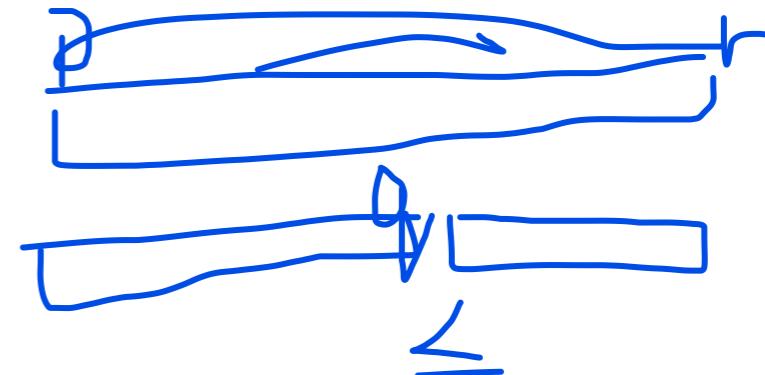
Dorotkas-MacBook-Pro:WspolbP pmp\$

sortowanie szybkie - wersja wielowątkowa

```
def quickSort(A, p, r):
    if p < r:
        q = partition(A, p, r)
        quickSort(A, p, q)
        quickSort(A, q + 1, r)
```

```
def partition(A, p, r):
    k = random.randint(p, r)
    A[k], A[r] = A[r], A[k]
    i = p - 1
    x=A[r]
    for j in range(p, r + 1):
        if A[j] <= x:
            i = i + 1
            A[i], A[j] = A[j], A[i]
    if i < r:
        return i
    else:
        return i - 1
```

quickWW.py



wersja
zwykła

```
def quickSortWW(a,p,r,g):
# quicksort z wywołaniami rekursywnymi w osobnych wątkach
# dla g>0
# a dla g=0 działa jak wersja jednowątkowa
if g>0:
    if p<r:
        q=partition(a,p,r)
        t1 = threading.Thread(target = quickSortWW,
                              args = (a,p,q,g-1))
        t2 = threading.Thread(target = quickSortWW,
                              args = (a,q+1,r,g-1))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
    else :
        quickSort(a,p,r)
print ("skończył poziom ", g)
```

```
def sprawdz(A,p,r):
    # sprawdzenie, czy tablica jest posortowana
    for i in range (p,r):
        if A[i]>A[i+1]:
            print("nie posortowane!\n")
            return
    print("posortowane\n")
    return
```

funkcje
pomocnicze

#wypełnienie tablic losowymi elementami

```
a = []
b = []

for i in range (0,M):
    x = random.randint(1, M)
    a.append(x)
    b.append(x)
```

```
start = timer()      5000  
quickSort(b, 0, M-1)  
stop = timer()  
print(round((stop - start), 6))  
sprawdz(b, 0, M-1)
```

testy

```
start = timer()  
quickSortWW(a, 0, M-1, G) 3  
stop = timer()  
print(round((stop - start), 6))  
sprawdz(a, 0, M-1)
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 quickWW.py

0.033708

posortowane

skończył poziom 0
skończył poziom 0
skończył poziom 0
skończył poziom 1
skończył poziom 0
skończył poziom 0
skończył poziom 0
skończył poziom 1
skończył poziom 0
skończył poziom 1
skończył poziom 0
skończył poziom 1
skończył poziom 0
skończył poziom 2
skończył poziom 1
skończył poziom 2
skończył poziom 3

0.041267

posortowane



Lock

proste blokady (semac-

fory binarne)

Lock() - tworzy (otwarty)

acquire() — opuszcza sem

release() - Podnosić

import threading

Pizyktord

lockSuma = threading.Lock()
sum=0

def f1(x):
 global sum
 with lockSuma:
 sum = sum+x*x

def f2(y):
 global sum
 lockSuma.acquire()
 sum = sum+10*y
 lockSuma.release()

Sekcje
krytyczne

```
def oblicz(x,y):  
    # wylicza  $x^2 + 10^y$   
    t1 = threading.Thread(target = f1, args = (x,))  
    t2 = threading.Thread(target = f2, args = (y,))  
    t1.start()  
    t2.start()  
    t1.join()  
    t2.join()  
    return suma
```

```
#test  
x=2  
y=10  
w = oblicz(x,y)  
  
if w==x*x+10*y:  
    print("dobrze")  
else:  
    print("zle");
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 lock1.py  
dobrze  
Dorotkas-MacBook-Pro:WspolbP pmp$
```

jest wiele innych mechanizmów

Synchronizacji.

Wątki współbieżne

tworzenie wątku współbieżnego

```
class threading.Thread(group=None, target=None,  
                      name=None, args=(), kwargs={}, *, daemon=None)
```

(wywoływany z argumentami kluczowymi, nie pozycyjnymi)

group - nieużywany

target - funkcja, którą wątek ma wykonać

name - nazwa wątku

args - lista argumentów pozycyjnych dla funkcji **target**

kwargs - słownik argumentów kluczowych dla funkcji **target**

deamon - informacja, czy ma być demonem

Tworzy nowy wątek (obiekt) działający współbieżnie z wątkiem rodzicielskim i wykonujący wskazaną funkcję.

uruchamianie wątku

start()

wcielenie wątku

join(timeout=None)

timeout - czas po którym oczekiwanie zostanie przerwane

```
def quickSortWW(a,p,r,g):
# quicksort z wywołaniami rekurencyjnymi w osobnych wątkach
# dla g>0
# a dla g=0 działa jak wersja jednowątkowa
if g>0:
    if p<r:
        q=partition(a,p,r)
        t1 = threading.Thread(target = quickSortWW,
                               args = (a,p,q,g-1))
        t2 = threading.Thread(target = quickSortWW,
                               args = (a,q+1,r,g-1))
        t1.start()
        t2.start()
        t1.join()
        t2.join()
    else :
        quickSort(a,p,r)
print ("skończył poziom ", g)
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 quickWW.py

0.033708

posortowane

skończył poziom 0

skończył poziom 0

skończył poziom 0

skończył poziom 1

skończył poziom 0

skończył poziom 0

skończył poziom 0

skończył poziom 1

skończył poziom 0

skończył poziom 1

skończył poziom 0

skończył poziom 2

skończył poziom 1

skończył poziom 2

skończył poziom 3

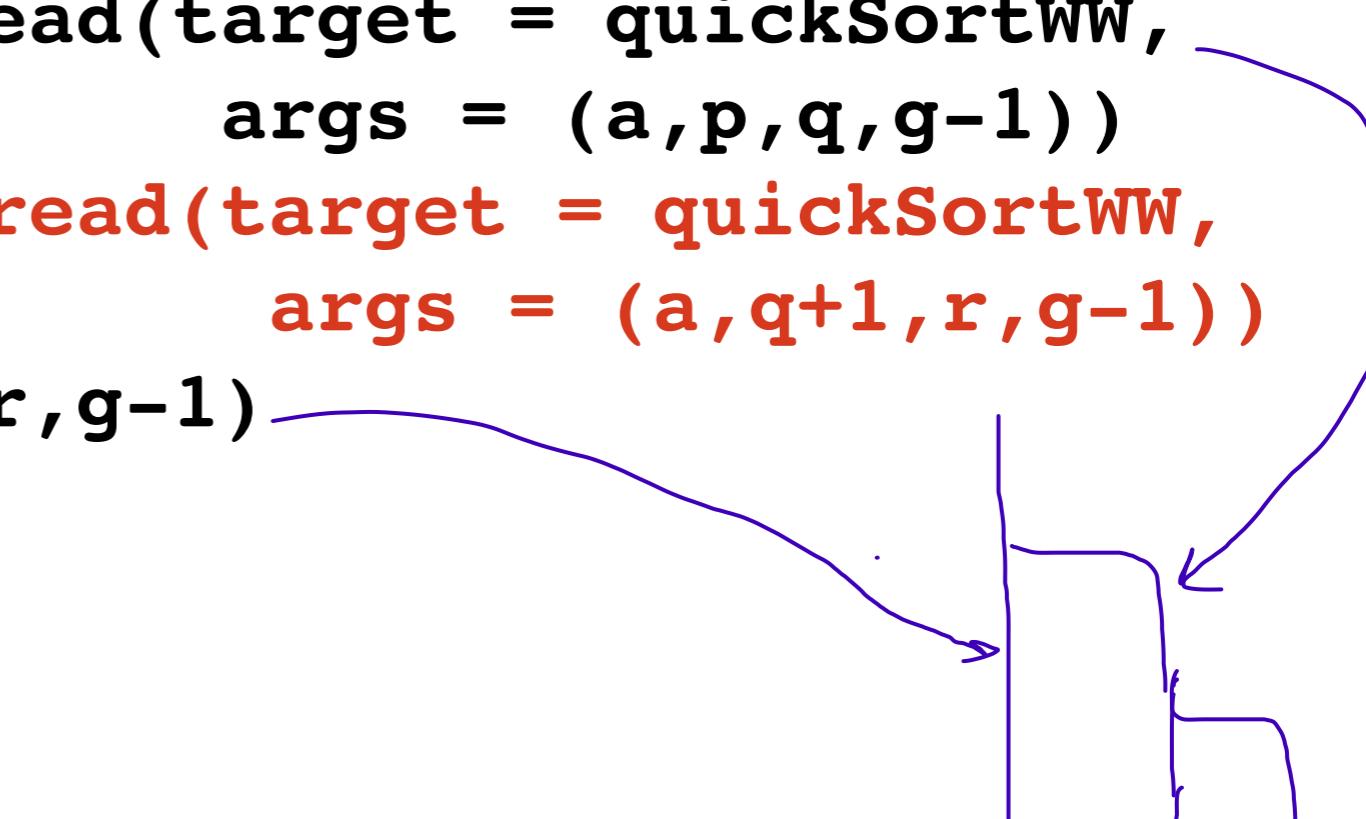
0.041267

posortowane

$$\text{6} = 3$$

quickWW1.py

```
def quickSortWW(a,p,r,g):
    # quicksort z jednym wywołaniem rekursywnym w osobnym
    # wątku dla g>0
    # a dla g=0 działa jak wersja jednowątkowa
    if g>0:
        if p<r:
            q=partition(a,p,r)
            t1 = threading.Thread(target = quickSortWW,
                                  args = (a,p,q,g-1))
            # t2 = threading.Thread(target = quickSortWW,
            #                       args = (a,q+1,r,g-1))
            quickSortWW(a,q+1,r,g-1)
            t1.start()
            # t2.start()
            t1.join()
            # t2.join()
    else :
        quickSort(a,p,r)
print ("skończył poziom ", g)
```



quickWW1.py

```
def quickSortWW(a,p,r,g):
    # quicksort z jednym wywołaniem rekursywnym w osobnym
    # wątku dla g>0
    # a dla g=0 działa jak wersja jednowątkowa
    if g>0:
        if p<r:
            q=partition(a,p,r)
            t1 = threading.Thread(target = quickSortWW,
                                  args = (a,p,q,g-1))
            # t2 = threading.Thread(target = quickSortWW,
            #                       args = (a,q+1,r,g-1))
            quickSortWW(a,q+1,r,g-1)
            t1.start()
            # t2.start()
            t1.join()
            # t2.join()
        else :
            quickSort(a,p,r)
    print ("skończył poziom ", g)
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 quickWW1.py

0.392515

posortowane

skończył poziom 0

skończył poziom 0

skończył poziom 1

skończył poziom 0

skończył poziom 0

skończył poziom 1

skończył poziom 2

skończył poziom 0

skończył poziom 0

skończył poziom 1

skończył poziom 0

skończył poziom 0

skończył poziom 1

skończył poziom 2

skończył poziom 3

0.409964

posortowane

mechanizmy synchronizacji wątków

- wcielenie (join)
- blokady (Lock) ✓
- blokady rekurencyjne (RLock) ✓
- semafory (Semaphore) — acquire, release
- zmienne warunku (condition variable) ↪
- bariery (Barrier) ↪
- Timer ↪
- kolejki Queue ↪ Obiektowy moduł

class `threading.Lock`

`Lock` to rodzaj semafora binarnego (otwarty lub zamknięty, odblokowany lub zablokowany)

`acquire(blocking=True, timeout=-1)`

Przejmuje zamek blokującą lub nieblokującą

`blocking True`: blokuje się, dopóki zamek nie będzie odblokowany, po czym przejmuje go i blokuje. Zwraca w wyniku `True`

`blocking False`: nie blokuje się, tylko zwraca od razu `False`, gdy zamek jest zablokowany i nie można go przejąć, w przeciwnym przypadku przejmuje go i blokuje i zwraca w wyniku `True`

Argument `timeout` oznacza maksymalny czas oczekiwania na odblokowanie zamka; `-1` oznacza nieskończone oczekiwanie (wykluczona wartość gdy `blocking` jest `False`)

release()

zwalnia zamek, może być wywołane przez dowolny wątek, niekoniecznie ten co nałożył blokadę. Jeżeli jakieś wątki czekają zablokowane na tym zamku to dokładnie jeden z nich zostanie odblokowany i znowu zablokuje zamek

wywołane na niezablokowanym zamku powoduje zgłoszenie wyjątku

RuntimeError

locked()

zwraca True, jeżeli zamek jest w stanie zamkniętym.

```
import threading

lockSuma = threading.Lock()
suma=0

def f1(x):
    global suma
    with lockSuma:
        suma = suma+x*x

def f2(y):
    global suma
    lockSuma.acquire()
    suma = suma+10*y
    lockSuma.release()
```

class threading.RLock

RLock narzędzie synchronizacyjne podobne do Lock z dwoma zasadniczymi różnicami

- wątek może wielokrotnie zablokować ten sam zamek a doblokowanie nastąpi po tej samej ilości odblokowań
- wątek, który zablokował zamek *przejmuje go* i tylko on może go odblokowywać

```
acquire(blocking=True, timeout=-1)
```

Przejmuje zamek blokującą lub nieblokującą

bez argumentów:

- jeżeli zamek jest odblokowany przejmuje go i blokuje,
- jeżeli już był przejęty przez ten wywołujący acquire wątek, to zwiększa licznik blokad o 1,
- jeżeli zamek jest przejęty przez inny wątek, to blokuje się, dopóki zamek nie będzie odblokowany, po czym przejmuje go i blokuje.
- w każdej sytuacji nie zwraca wyniku

blocking True: jak bez argumentów, dodatkowo zwraca w wyniku True

blocking False: nie blokuje się, tylko zwraca od razu False, gdy zamek jest zablokowany i nie można go przejąć, w przeciwnym przypadku działa jak w wariantie bez argumentów ale zwraca w wyniku True

release()

zwiększa liczbę blokad o 1. Jeżeli dojdzie do 0 blokad to zwalnia zamek i jeżeli jakieś wątki czekają zablokowane na tym zamku to dokładnie jeden z nich zostanie odblokowany, przejmie zamek i znowu zablokuje go.

wywołane na niezablokowanym zamku powoduje zgłoszenie wyjątku

RuntimeError

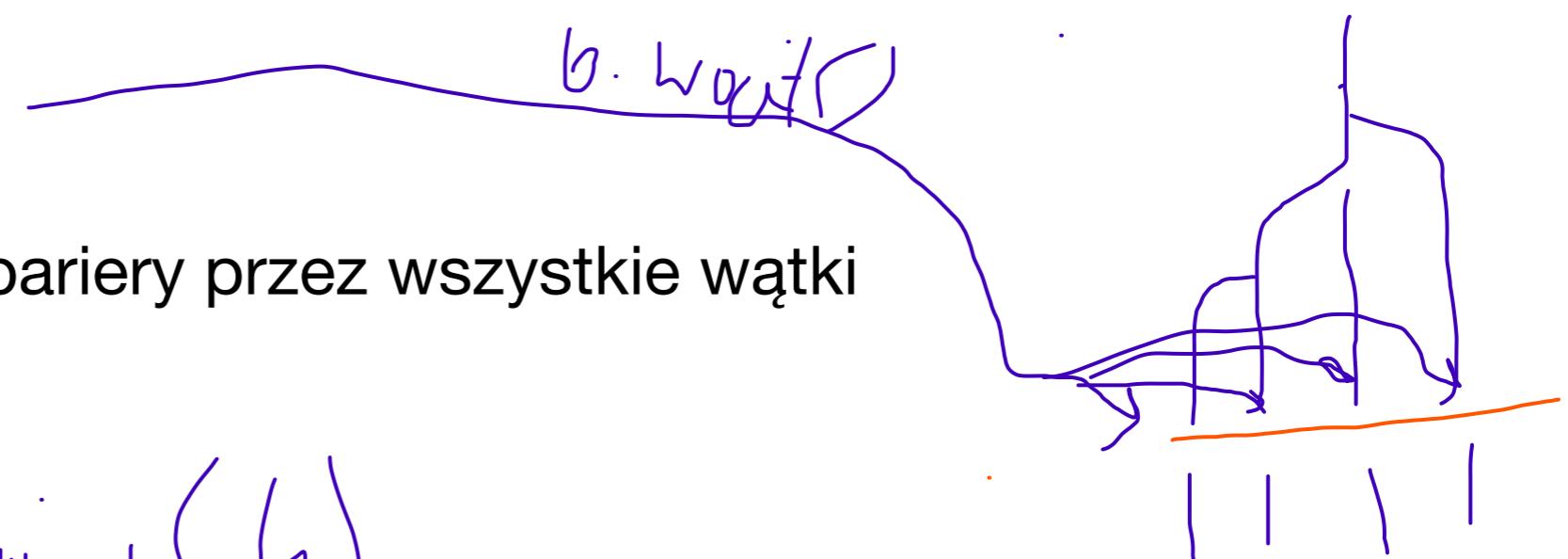
```
class threading.Barrier(parties, action=None, timeout=None)
```

↑
tworzy bariere dla ilości wątków parties, parametr action to funkcja wywoływana przez jeden z wątków po przekroczeniu bariery

```
wait(timeout=None)
```

czekanie na dojście do bariery przez wszystkie wątki

b = Barrier(4)



```
import threading
import time

def f(arg,name,s):
    for i in range(arg):
        print(name, 'i=', i)
        time.sleep(s)

t1 = threading.Thread(target = f, args = (5, 't1',1))
t2 = threading.Thread(target = f, args = (5, 't2',0.7))
t1.start()
t2.start()
t1.join()
t2.join()
print ("koniec")
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 threadingP.

t1 i= 0

t2 i= 0

t2 i= 1

t1 i= 1

t2 i= 2

t1 i= 2

t2 i= 3

t2 i= 4

t1 i= 3

t1 i= 4

koniec

Dorotkas-MacBook-Pro:WspolbP pmp\$

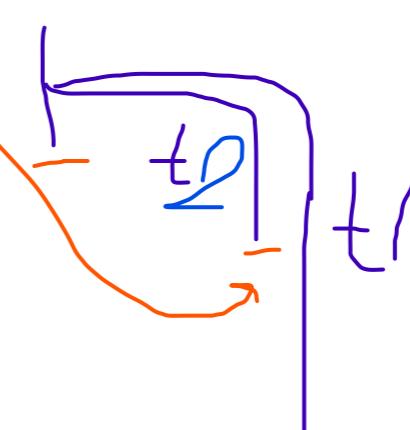
```

# utworzenie dwóch wątków współbieżnych wykonujących
# funkcję f
# synchronizacja barierą
import threading
import time

def f(arg,name,s,bar):
    for i in range(arg):
        print(name, 'i=', i)
        time.sleep(s)
    print ("koniec pracy "+name)
    bar.wait()
    print ("koniec czekania "+name)

b = threading.Barrier(3)
t1 = threading.Thread(target = f, args = (5, 't1',1,b))
t2 = threading.Thread(target = f, args = (2,'t2',0.7,b))
t1.start()
t2.start()
print ("koniec pracy ")
b.wait()
# t1.join()
# t2.join()
print ("koniec czekania")

```



Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 threadingP1.py

t1 i= 0

t2 i= 0

koniec pracy ← głowny

t2 i= 1

t1 i= 1

koniec pracy t2

t1 i= 2

t1 i= 3

t1 i= 4

koniec pracy t1

koniec czekania ← głowny

Dorotkas-MacBook-Pro:WspolbP pmp\$



quickWW1.py

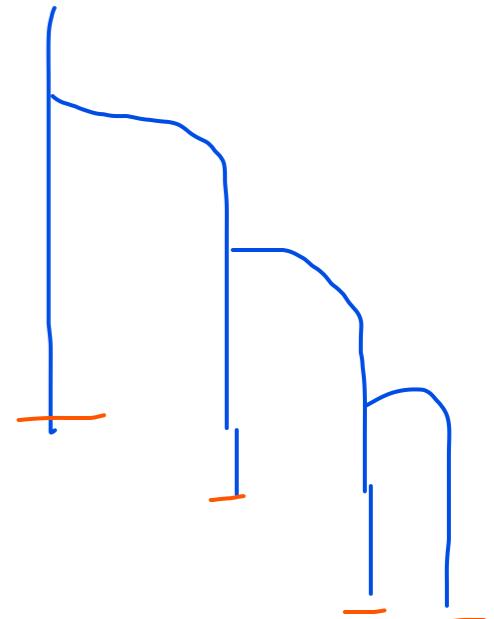
```
def quickSortWW(a,p,r,g):
    # quicksort z jednym wywołaniem rekursywnym w osobnym
    # wątku dla g>0
    # a dla g=0 działa jak wersja jednowątkowa
    if g>0:
        if p<r:
            q=partition(a,p,r)
            t1 = threading.Thread(target = quickSortWW,
                                  args = (a,p,q,g-1))
            # t2 = threading.Thread(target = quickSortWW,
            #                       args = (a,q+1,r,g-1))
            quickSortWW(a,q+1,r,g-1)
            t1.start()
            # t2.start()
            t1.join()
            # t2.join()
        else :
            quickSort(a,p,r)
    print ("skończył poziom ", g)
```

```

def quickSortWW(a,p,r,g,bar):
    # quicksort z wywołaniami rekursywnymi w osobnych wątkach dla
    # a dla g=0 działa jak wersja jednowątkowa
    if g>0:
        if p<r:
            bariera = threading.Barrier(2)
            q=partition(a,p,r)
            t1 = threading.Thread(target = quickSortWW,
                                args = (a,p,q,g-1,bariera))

            t1.start()
            quickSortWW(a,q+1,r,g-1,bariera)
        #
        t1.join()
    else :
        quickSort(a,p,r)
    bar.wait()
    print ("skończył poziom ", g)

```



```

bariera = threading.Barrier(1)
quickSortWW(a,0,M-1,G,bariera)

```

Dorotkas-MacBook-Pro:WspolbP pmp\$ python3 quickWWC1.py
0.031263
posortowane

skończył poziom 0
skończył poziom 1
skończył poziom 1
skończył poziom 1
skończył poziom 1
skończył poziom 2
skończył poziom 3
skończył poziom 2

0.034004
posortowane

Gniazda (sockets)

Gniazdo to struktura danych używana do tworzenia końcówki kanału komunikacyjnego, który służy do odbierania i wysyłania danych do innego procesu

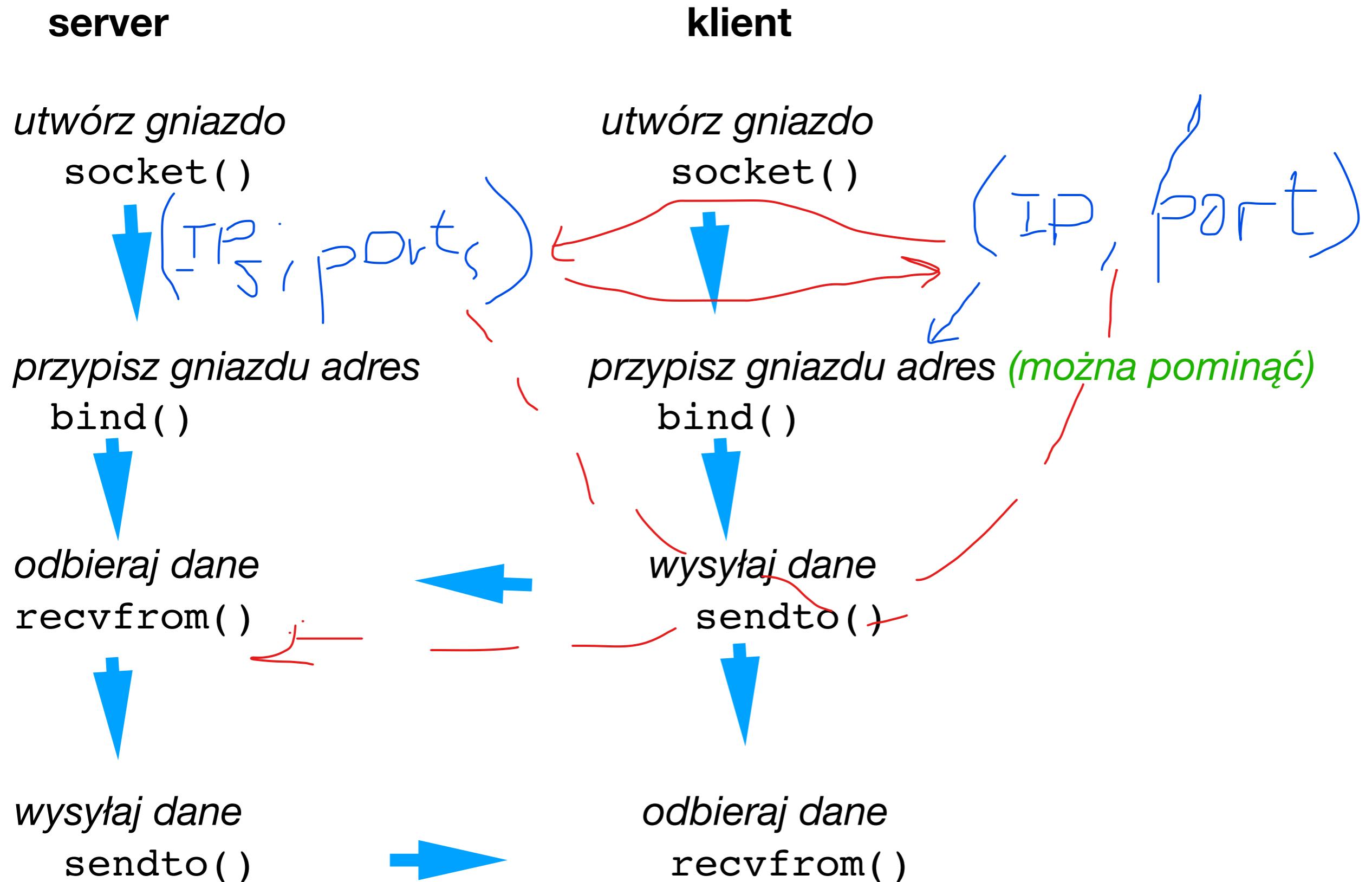
Typy gniazd:

- strumieniowe
- datagramowe
-

Mogą być tworzone w domenie

- uniksowej (lokalnie)
- internetowej

Gniazda bezpołączeniowe (SOCK_DGRAM)



Informacja o funkcjach w C obsługujących gniazda

`int socket (int rodzina, int typ, int protokół)`

tworzy deskryptor gniazda, czyli odsyłacz (`int`) do odpowiedniej struktury w systemie

`rodzina` rodzina adresów/protokołów (`AF_INET`, `AF_UNIX`, ...)

`typ` typ gniazda: strumieniowe (`SOCK_STREAM`), datagramowe (`SOCK_DGRAM`)

`protokół` protokół komunikacyjny, zazwyczaj 0 co oznacza domyślny wybór

zwraca: deskryptor gniazda w przypadku sukcesu, a w przypadku niepowodzenia -1

`int bind(int deskryptor, struct sockaddr *adres, int długość)`

wiąże z `deskryptorem` gniazda adres do komunikacji, zależny od użytej rodziny protokołów

`inet_pton()` - funkcja wypełniająca strukturę adresową na podstawie adresu IP

```
#include <arpa/inet.h>
#define IP "127.0.0.1" // lub inny numer IP
.....
sockaddr_in sad;
.....
if( inet_pton( AF_INET, IP, &sad.sin_addr ) <= 0 ){
    perror( "inet_pton() ERROR" );
    exit( 1 );
}
```



```
int sendto (int deskryptor, char *bufor, int wielkość,  
           int flagi, strust sockaddr *adres, int długość)
```

deskryptor deskryptor gniazda

bufor wskaźnik na komunikat do wysłania

wielkość rozmiar komunikatu

flagi flagi, typowo 0 (może być MSG_OOB (poza kolejnością) MSG_PEEK)

adres adres odbiorcy komunikatu (tak jak w funkcji bind())

długość to długość struktury przekazanej jako adres

zwraca: liczbę przesłanych bajtów w przypadku sukcesu, a w przypadku niepowodzenia -1

```
int recvfrom ( int deskryptor, char *bufor, int wielkość,  
               int flagi, struct sockaddr *adres, int *długość )
```

deskryptor deskryptor gniazda

bufor wskaźnik na komunikat do wysłania

wielkość rozmiar bufora na komunikat (będzie obcięty do tej wielkości!)

flagi flagi, typowo 0 (może być MSG_OOB (poza kolejnością) MSG_PEEK)

adres miejsce na adres nadawcy komunikatu (tak jak w funkcji bind())

długość to wskaźnik na długość struktury z otrzymanym adresem

zwraca: liczbę odebranych bajtów w przypadku sukcesu, a w przypadku niepowodzenia -1

inne funkcje pomocnicze

`unsigned long htonl(unsigned long)`
konwersja **host** do **network** dla **long int** (4 bajty)

`unsigned short htons(unsigned short)`
konwersja **host** do **network** dla **short int** (2 bajty)

`unsigned long ntohl(unsigned long)`
konwersja **network** do **host** dla **long int** (4 bajty)

`unsigned short ntohs(short)`
konwersja **network** do **host** dla **short int** (2 bajty)

wersja w Pythonie

<https://docs.python.org/3/library/socket.html>

```
socket(family=AF_INET, type=SOCK_STREAM, proto=0,  
fileno=None)¶
```

Tworzy nowe gniazdo (obiekt) i zwraca jako wynik

family rodzina adresów/protokołów (*AF_INET*, *AF_UNIX*, ...)

type typ gniazda: strumieniowe (*SOCK_STREAM*), datagramowe
(*SOCK_DGRAM*)

proto protokół komunikacyjny, zazwyczaj 0 co oznacza domyślny wybór

fileno pozwala wziąć wcześniejsze parametry z pliku

bind(address)

$s = \text{socket.socket}(\dots)$
 $s.bind$

związuje gniazdo z adresem (nie może być już zвязane), Adres, to para: adres IP i numer portu (dla rodziny połączeń internetowych)

close()

zamyka gniazdo

recvfrom(bufsize[, flags])

bufsize maksymalna ilość pobieranych bajtów (komunikat będzie obcięty do tej wielkości!)

flags flagi, typowo 0 (może być MSG_OOB (poza kolejnością) MSG_PEEK)

zwraca parę: (bytes , address) czyli obiekt bajtowy będący odebranym komunikatem i adres nadawcy, czyli też parę: nr IP i nr portu

sendto(bytes, address)

sendto(bytes, flags, address)

wysyła obiekt bajtowy bytes na adres address (adres, czyli parę: nr IP i nr portu)
wynikiem jest ilość wysłanych bajtów

```
import socket
```

serwerUDP.py

```
IP      = "127.0.0.1"  
port    = 5001  
bufSize = 1024
```

```
# utworzenie gniazda UDP
```

```
UDPServerSocket = socket.socket(socket.AF_INET, socket.SOCK_
```

DR&KM

```
# związanie gniazda z IP i portem  
UDPServerSocket.bind((IP, port))
```

```
print("serwer UDP działa")
```

```
# obsługa nadchodzących datagramów
```

```
while(True):
```

```
    komB,adres = UDPServerSocket.recvfrom(bufSize)
```

```
    print(komB)
```

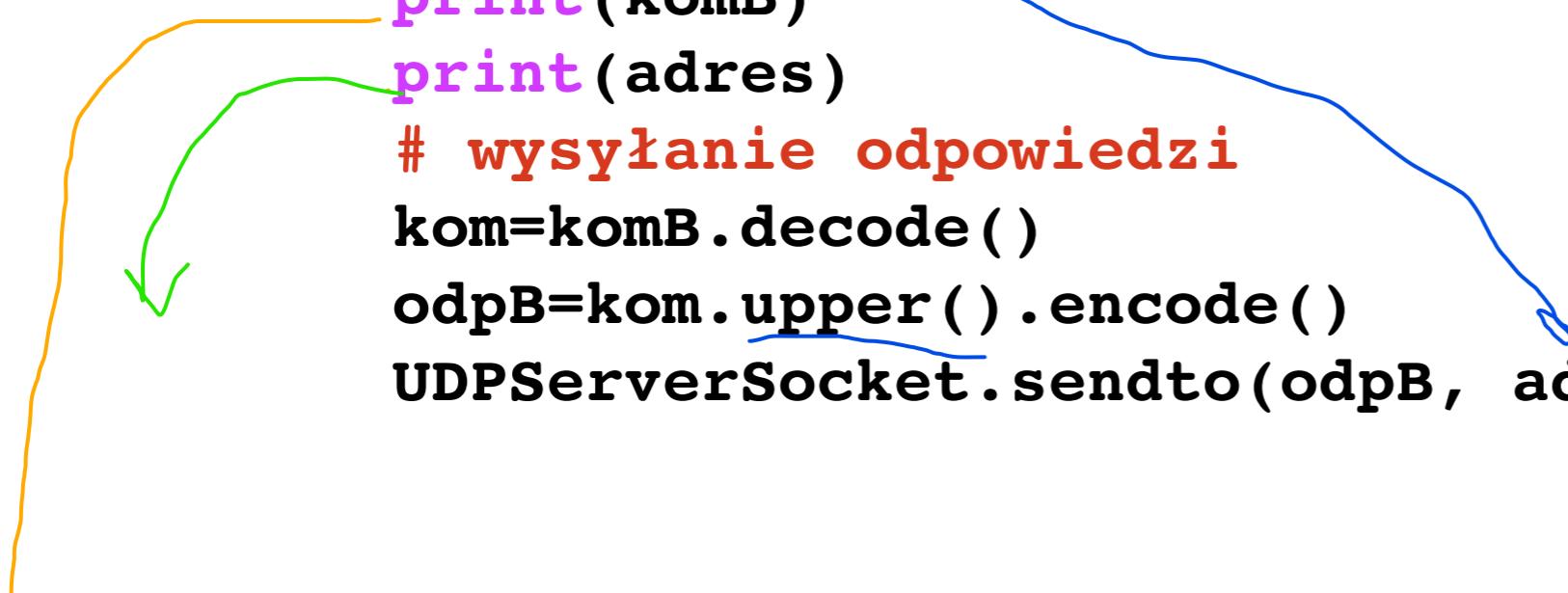
```
    print(adres)
```

```
    # wysyłanie odpowiedzi
```

```
    kom=komB.decode()
```

```
    odpB=kom.upper().encode()
```

```
    UDPServerSocket.sendto(odpB, adres)
```



```
import socket
```

klientUDP.py

```
komA = "aaaaa"  
komAB = str.encode(komA)  
serwerAdresPort = ("127.0.0.1", 5001)  
#klientAdresPort = ("127.0.0.1", 5002)  
bufSize = 1024  
# tworzy gniazdo UDP po stronie klienta  
UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)  
  
# związuje gniazdo z parą adres, port - można pominąć  
#UDPClientSocket.bind(klientAdresPort)  
  
# wysyła do serwera przez utworzone gniazdo  
UDPClientSocket.sendto(komAB, serwerAdresPort)  
odp = UDPClientSocket.recvfrom(bufSize)  
print(odp)  
UDPClientSocket.sendto("bbb".encode(), serwerAdresPort)  
odp = UDPClientSocket.recvfrom(bufSize)  
print(odp)
```

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 serwerUDP.py**

serwer UDP działa

b'aaaaaa'
('127.0.0.1' , 49721)

b'aaaaaa'
('127.0.0.1' , 54240)

b'bbb'
('127.0.0.1' , 49721)

b'bbb'
('127.0.0.1' , 54240)

Dorotkas-MacBook-Pro:WspolbP pmp\$ **python3 klientUDP.py & python3 klientUDP.py**

[1] 5263
(b'AAAAAA' , ('127.0.0.1' , 5001))
(b'AAAAAA' , ('127.0.0.1' , 5001))
(b'BBB' , ('127.0.0.1' , 5001))
(b'BBB' , ('127.0.0.1' , 5001))
[1]+ Done

python3 klientUDP.py

Dorotkas-MacBook-Pro:WspolbP pmp\$

po odkomentowaniu

```
#UDPClientSocket.bind(klientAdresPort)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 serwerUDP.py
serwer UDP działa
b'aaaaa'
('127.0.0.1', 5002)
b'bbb'
('127.0.0.1', 5002)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientUDP.py & python3  
klientUDP.py  
[1] 5298  
Traceback (most recent call last):  
  File "klientUDP.py", line 12, in <module>  
(b'AAAAAA', ('127.0.0.1', 5001)) |  
    UDPClientSocket.bind(klientAdresPort)  
OSError: [Errno 48] Address already in use  
(b'BBB', ('127.0.0.1', 5001)) |  
[1]+  Exit 1                                         python3 klientUDP.py .  
Dorotkas-MacBook-Pro:WspolbP pmp$
```

przesyłanie liczb - trzeba przekształcić do postaci obiektu bajtowego

<https://docs.python.org/2/library/struct.html#struct.pack>

komA = 1

komNP = struct.pack('!i', komA)

silowy format

klientUDP1.py

serwerAdresPort = ("127.0.0.1", 5001)

klientAdresPort = ("127.0.0.1", 5002)

bufSize = 1024

tworzy gniazdo UDP po stronie klienta

UDPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

związuje gniazdo z parą adres, port - można pominąć

#UDPClientSocket.bind(klientAdresPort)

wysyła do serwera przez utworzone gniazdo

UDPClientSocket.sendto(komNP, serwerAdresPort)

odpNP = UDPClientSocket.recvfrom(bufSize)

odp = struct.unpack('!i', odpNP[0])

print(odp)

wiadom.

UDPClientSocket.sendto(struct.pack('!i', 2), serwerAdresPort)

odpNP = UDPClientSocket.recvfrom(bufSize)

print(struct.unpack('!i', odpNP[0]))

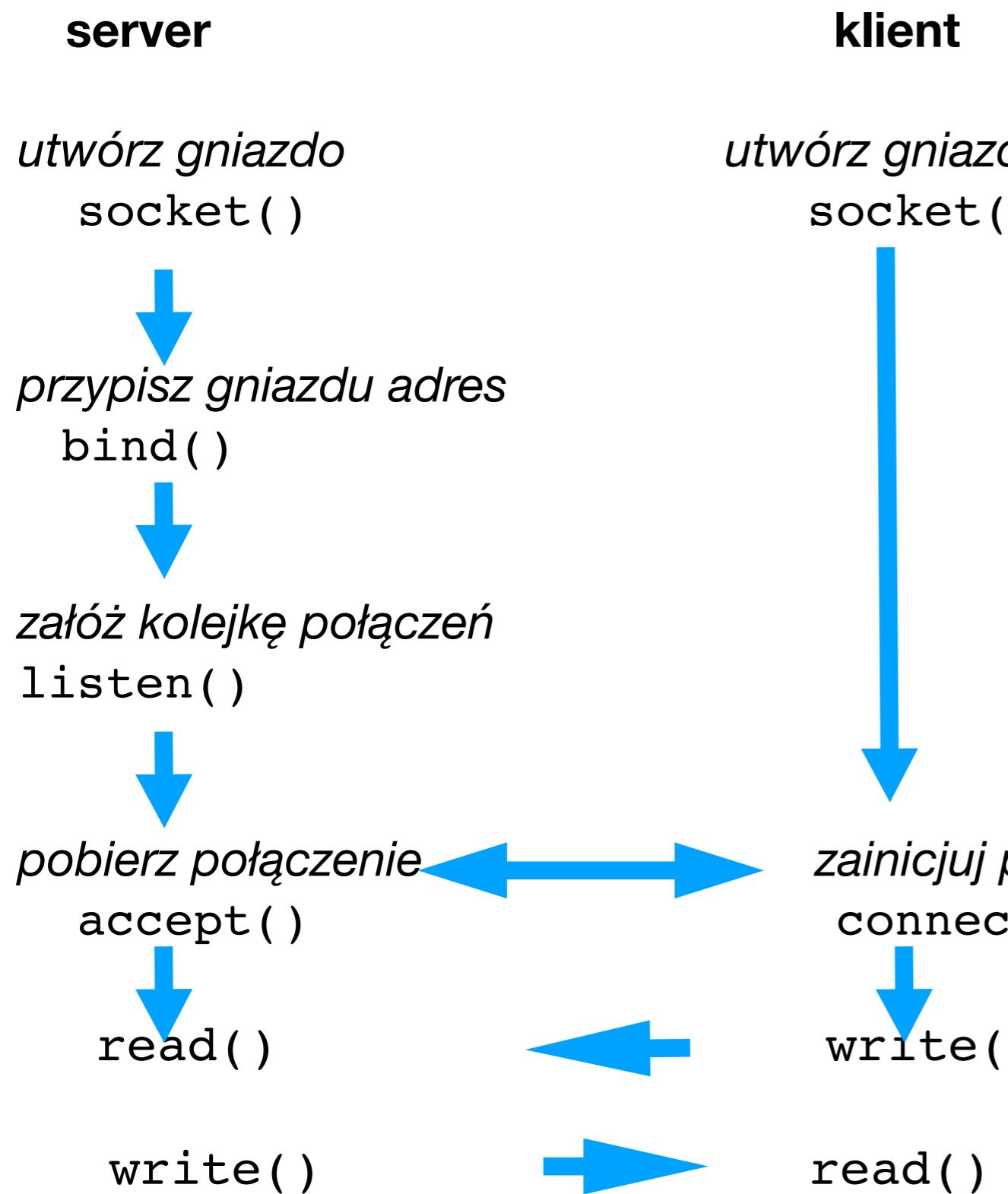
serwerUDP1.py

```
# obsługa nadchodzących datagramów
while(True):
    komNP,adres = UDPServerSocket.recvfrom(bufSize)
    kom=struct.unpack('!i',komNP)
    print(kom)
    print(adres)
    # wysyłanie odpowiedzi
    odpNP = struct.pack('!i',kom[0]+100)
    UDPServerSocket.sendto(odpNP, adres)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 serwerUDP1.py
serwer UDP działa
(1,
('127.0.0.1', 52072)
(1,
('127.0.0.1', 51346)
(2,
('127.0.0.1', 52072)
(2,
('127.0.0.1', 51346)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientUDP1.py & python3 klientUDP1.py
[1] 4154
(101,
(101,
(102,
(102,
[1]+ Done
python3 klientUDP1.py
Dorotkas-MacBook-Pro:WspolbP pmp$ fg
```

Gniazda połacienniowe (SOCK_STREAM)



Gniazda (sockets)

Gniazdo to struktura danych używana do tworzenia końcówki kanału komunikacyjnego, który służy do odbierania i wysyłania danych do innego procesu

Typy gniazd:

- strumieniowe
- datagramowe
-

TCP

Mogą być tworzone w domenie

- uniksowej (lokalnie)
- internetowej

wersja w Pythonie

<https://docs.python.org/3/library/socket.html>

```
socket(family=AF_INET, type=SOCK_STREAM, proto=0,  
fileno=None)
```

Tworzy nowe gniazdo (obiekt) i zwraca jako wynik

family rodzina adresów/protokołów (AF_INET, AF_UNIX, ...)

type typ gniazda: strumieniowe (SOCK_STREAM), datagramowe
(SOCK_DGRAM)

proto protokół komunikacyjny, zazwyczaj 0 co oznacza domyślny wybór

fileno pozwala wziąć wcześniejsze parametry z pliku

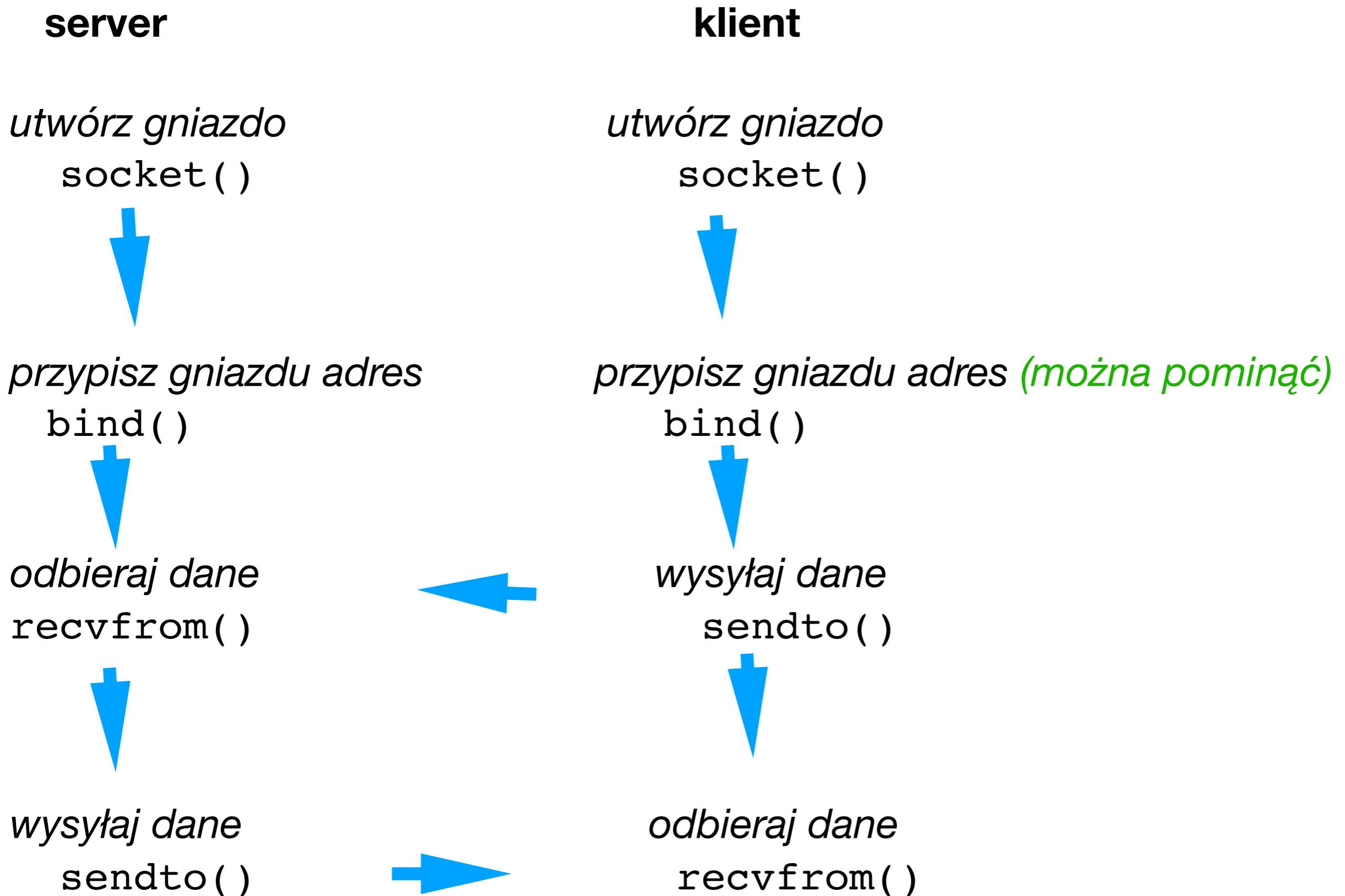
bind(address)

związuje gniazdo z adresem (nie może być już związane), Adres, to para: adres IP i numer portu (dla rodziny połączeń internetowych)

close()

zamyka gniazdo

Gniazda bezpołączeniowe (SOCK_DGRAM)



recvfrom(bufsize[, flags])

bufsize maksymalna ilość pobieranych bajtów (komunikat będzie obcięty do tej wielkości!)

flags flagi, typowo 0 (może być MSG_OOB (poza kolejnością) MSG_PEEK)

zwraca parę: (bytes, address) czyli obiekt bajtowy będący odebranym komunikatem i adres nadawcy, czyli też parę: nr IP i nr portu

sendto(bytes, address)

sendto(bytes, flags, address)

wysyła obiekt bajtowy *bytes* na adres *address* (adres, czyli parę: nr IP i nr portu)
wynikiem jest ilość wysłanych bajtów

przesyłanie liczb - trzeba przekształcić do postaci obiektu bajtowego

<https://docs.python.org/2/library/struct.html#struct.pack>

```
komA = 1  
komNP = struct.pack('!i', komA)
```

```
serwerAdresPort = ("127.0.0.1", 5001)  
klientAdresPort = ("127.0.0.1", 5002)
```

```
bufSize = 1024
```

tworzy gniazdo UDP po stronie klienta

```
UDPCClientSocket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

związuje gniazdo z parą adres, port - można pominąć
#UDPCClientSocket.bind(klientAdresPort)

wysyła do serwera przez utworzone gniazdo

```
UDPCClientSocket.sendto(komNP, serwerAdresPort)
```

```
odpNP = UDPCClientSocket.recvfrom(bufSize)
```

```
odp = struct.unpack('!i', odpNP[0])
```

```
print(odp)
```

```
UDPCClientSocket.sendto(struct.pack('!i', 2), serwerAdresPort)
```

```
odpNP = UDPCClientSocket.recvfrom(bufSize)
```

```
print(struct.unpack('!i', odpNP[0]))
```

klientUDP1.py

```
import socket
```

serwerUDP1.py

```
IP      = "127.0.0.1"
port    = 5001
bufSize = 1024

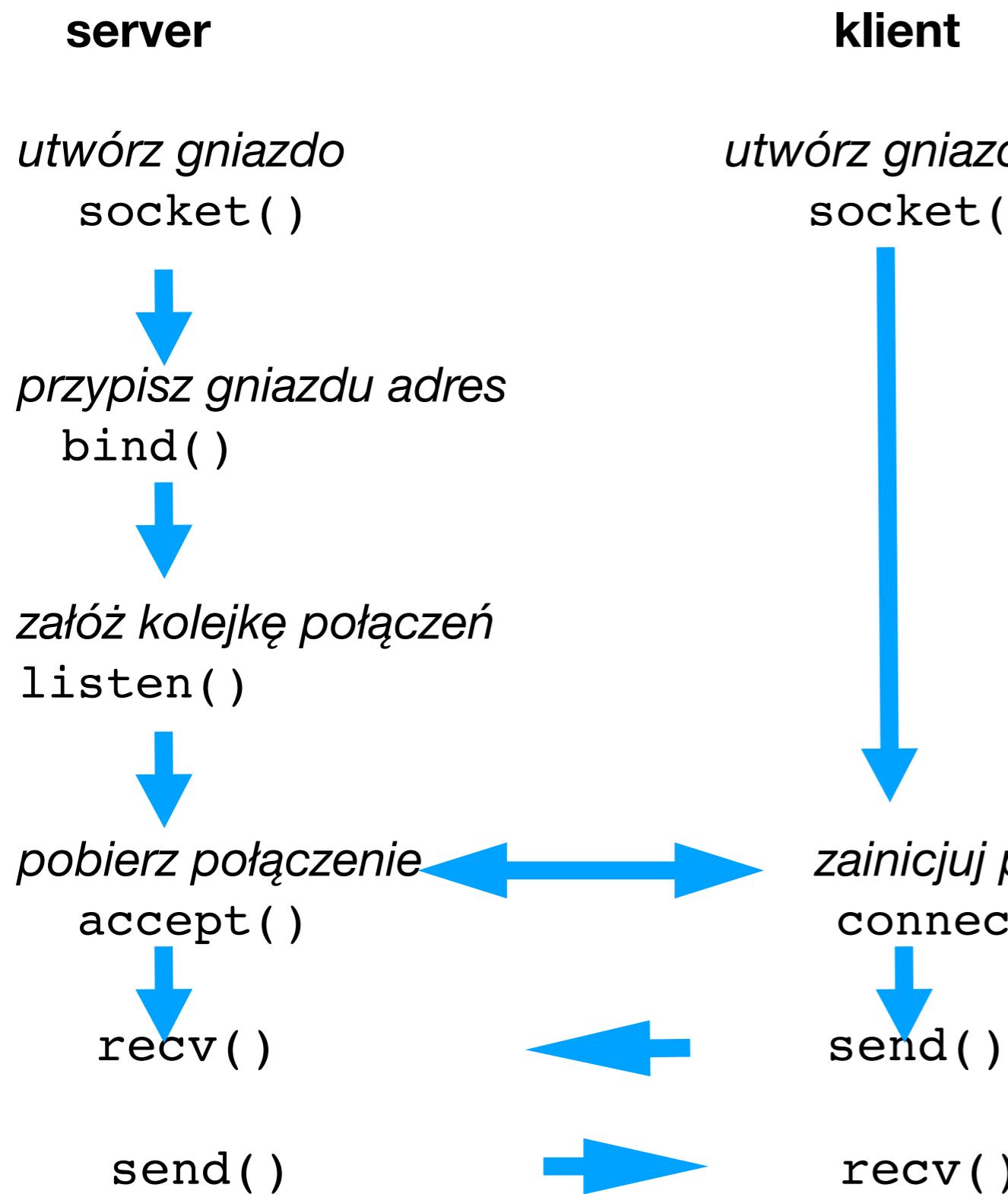
# utworzenie gniazda UDP
UDPServerSocket = socket.socket(socket.AF_INET, socket.SOCK_

# związanie gniazda z IP i portem
UDPServerSocket.bind((IP, port))

print("serwer UDP działa")

# obsługa nadchodzących datagramów
while(True):
    while(True):
        komNP,adres = UDPServerSocket.recvfrom(bufSize)
        kom=struct.unpack('!i',komNP)
        print(kom)
        print(adres)
        # wysyłanie odpowiedzi
        odpNP = struct.pack('!i',kom[0]+100)
        UDPServerSocket.sendto(odpNP, adres)
```

Gniazda połacienniowe (SOCK_STREAM)



recv(bufsize[, flags])

odbiera dane z gniazda. Zwraca obiekt bajtowy będący odebranym komunikatem. Odbieranie jest zazwyczaj blokujące (tzn. czekamy na pojawienie się danych w gnieździe) chyba, że ustawiono na odbiór nieblokujący.

bufsize maksymalna ilość pobieranych bajtów

flags flagi, domyślnie 0 (może być MSG_OOB (poza kolejnością)
MSG_PEEK, ...)

setblocking(flag)

Ustawia tryb blokujący (flag = True) lub nieblokujący (flag = False)

settimeout(value)

Ustala timeout na blokujących operacjach na gnieździe. Po podanym w parametrze value czasie zostanie zgłoszony wyjątek, jeżeli w międzyczasie operacja sie nie wykona.

`send(bytes[, flags])`

wysyła obiekt bajtowy *bytes* do gniazda. Wynikiem jest ilość wysłanych bajtów.
Programista jest odpowiedzialny za sprawdzenie, czy wszystko co było podane w parametrze *bytes* zostało wysłane i w razie potrzeby dosłać pozostałość.

`flags` flagi, jak w funkcji `recv`

send all .

accept()

Akceptuje połączenie. Gniazdo musi być związane z adresem i nasłuchiwać na połączenia. Wynikiem jest para (conn, address) gdzie conn to nowe gniazdo do komunikacji a address jest adresem związanym z tym gniazdem po drugiej stronie.

listen([backlog])

Umożliwia nasłuchiwanie na połączenia. Parametr backlog to rozmiar kolejki oczekujących na połączenie, czyli na accept. Połączenia nie mieszczące się w kolejce będą odrzucane.

connect(address)

nawiązuje połączenie z gniazdem o podanym adresie

serwera

serwerTCP2.py

```
# serwer TCP, odbiera int, odsyła wartość zwiększoną o 100

import socket
import struct

IP      = "127.0.0.1"
port    = 5001
bufSize = 1024

# utworzenie gniazda TCP
TCPserverSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# związanie gniazda z IP i portem
TCPserverSocket.bind((IP, port))
TCPserverSocket.listen(5)
```

po stronie serwera

```
while (True):
    (clientsocket, address) = TCPserverSocket.accept()
    print(address)
    # obsługa połączenia
    while(True):
        komNP = clientsocket.recv(bufSize)
        if len(komNP) == 0: ← koniec, gniazdo po str. klienta zamkn.
            break
        kom=struct.unpack('!i', komNP)
        print(kom)
        time.sleep(2)
        # wysyłanie odpowiedzi
        odpNP = struct.pack('!i', kom[0]+100)
        clientsocket.send(odpNP)
```

kolejny klient

klientTCP1.py

```
# klient TCP, dwukrotnie wysyła i odbiera int

import socket
import struct

serwerAdresPort      = ("127.0.0.1", 5001)
bufSize = 1024
# tworzy gniazdo TCP po stronie klienta
TCPClientSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

# wysyła do serwera przez utworzone gniazdo
TCPClientSocket.connect(serwerAdresPort)
komA = 1
komANP = struct.pack('!i', komA)
TCPClientSocket.send(komANP)
odpNP = TCPClientSocket.recv(bufSize)
odp = struct.unpack('!i', odpNP)
print(odp)

TCPClientSocket.send(struct.pack('!i', 2))
odpNP = TCPClientSocket.recv(bufSize)
print(struct.unpack('!i', odpNP))
TCPClientSocket.close()
```

The diagram illustrates the flow of data between the client and the server. A yellow line connects the `TCPClientSocket` variable in the code to the `TCPClientSocket` in the diagram. Blue arrows indicate the flow of data: one arrow points from `komA` to the server, another from the server back to `odp`, and a third from `2` to the server.

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 serwerTCP2.py
serwer TCP działa
('127.0.0.1', 53236)
(1,)
(2,)
('127.0.0.1', 53237)
(1,)
(2,)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientTCP1.py
(101,)
(102,)

Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientTCP1.py
(101,)
(102,)

Dorotkas-MacBook-Pro:WspolbP pmp$
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 serwerTCP2.py  
serwer TCP działa
```

```
('127.0.0.1', 53288)  
(1,)  
(2,)  
'127.0.0.1', 53289)  
(1,)  
(2,)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientTCP1.py& python
```

```
klientTCP1.py
```

```
[1] 6680
```

```
(101,)
```

```
(102,)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ (101,)
```

```
(102,)
```

Serwer powołuje proces potomny do obsługi połączenia a sam nasłuchuje dalej na następnych klientów.

serwerTCP3.py

```
while (True):
    (clientsocket, address) = TCPserverSocket.accept()
    pid = os.fork()
    if pid == 0:
        # obsługa połączenia
        print(address)
        while(True):
            komNP = clientsocket.recv(bufSize)
            if len(komNP) == 0:
                break
            kom=struct.unpack('!i', komNP)
            print(kom)
            time.sleep(2)
            # wysyłanie odpowiedzi
            odpNP = struct.pack('!i', kom[0]+100)
            clientsocket.send(odpNP)
    else:
        pass
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 serwerTCP3.py  
serwer TCP działa
```

```
('127.0.0.1', 52582)  
(1,)  
(127.0.0.1', 52583)  
(1,)  
(2,)  
(2,)  
(127.0.0.1', 52584)  
(127.0.0.1', 52585)
```

```
(1,)  
(1,) Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientTCP1.py & python  
klientTCP1.py  
(2,)  
(2, [1] 4532
```

```
(101,)  
(101,)  
(102,)  
(102,)
```

```
[1]+ Done
```

```
python3 klientTCP1.py
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientTCP1.py & python  
klientTCP1.py
```

```
[1] 4537  
(101,)  
(101,)  
(102,)
```

Przykład komunikacji połączeniowej, wątek zamiast procesu potomnego

Serwer i klient, serwer powołuje **nowy wątek** do obsługi połączenia a sam nasłuchiwa dalej na następnych klientów.

Klient bez zmian

ten fragment zostanie zmieniony.

```
while (True):
    (clientsocket, address) = TCPserverSocket.accept()
    pid = os.fork()
    if pid == 0:
        # obsługa połączenia
        print(address)
        while(True):
            komNP = clientsocket.recv(bufSize)
            if len(komNP) == 0:
                break
            kom=struct.unpack('!i',komNP)
            print(kom)
            time.sleep(2)
            # wysyłanie odpowiedzi
            odpNP = struct.pack('!i',kom[0]+100)
            clientsocket.send(odpNP)
    else:
        pass
```

```
def handler(clientsocket,address,bufSize):
    # obsługa połączenia
    print(address)
    while(True):
        komNP = clientsocket.recv(bufSize)
        if len(komNP) == 0:
            break
        kom=struct.unpack('!i',komNP)
        print(kom)
        time.sleep(2)
        # wysyłanie odpowiedzi
        odpNP = struct.pack('!i',kom[0]+100)
        clientsocket.send(odpNP)
```

```
while (True):
    (clientsocket, address) = TCPserverSocket.accept()
    # pid = os.fork()
    t = threading.Thread(target = handler, args =
    (clientsocket,address,bufSize))
    t.start()
```

serwerTCPt.py

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 serwerTCPt.py  
serwer TCP działa
```

```
('127.0.0.1', 53363)  
(1,)  
(127.0.0.1', 53364)  
(1,)  
(2,)  
(2,)
```

```
Dorotkas-MacBook-Pro:WspolbP pmp$ python3 klientTCP1.py & python3 klientTCP1.py  
[1] 6903  
(101,)  
(101,)  
(102,)  
(102,)  
[1]+ Done
```

```
python3 klientTCP1.py
```

Klasyczne problemy programowania współbieżnego (ciąg dalszy)

Problem producenta-konsumenta

Producent tworzy zestawy danych, które przesyła do konsumenta

Konsument odbiera zastawy danych, które następnie przetwarza

komunikacja może być synchroniczna lub asynchroniczna

przy komunikacji asynchronicznej potrzebny jest bufor na składowanie przesyłanych danych

Semafony (przypomnienie)

Semafor s jest zmienną całkowitą przyjmującą tylko wartości nieujemne. Zdefiniowane są na nim dwie operacje

wait(s) jeżeli $s > 0$ to wykonaj $s = s - 1$, w przeciwnym razie wykonywanie procesu który zrobi tę operację **wait** jest wstrzymane. Taki proces nazywamy *wstrzymanym na semaforze* s

signal(s) jeżeli są jakieś procesy wstrzymane na semaforze s to wznow jeden z nich, w przeciwnym razie wykonaj $s = s + 1$

operacje **wait(s)** **signal(s)** **są niepodzielne**

semafor musi mieć nadaną nieujemną początkową wartość

Semafor binarny to semafor przyjmujący tylko wartości 0 lub 1. Wtedy w operacji **signal(s)** zamiast $s = s + 1$ wykonywane jest $s = 1$

rozwiązańie z semaforami

Queue Q // rozmiar N, początkowo pusta

$N > 0$

Semafor miejsce = N -- ilość miejsc w Q

Semafor dane = 0 -- il. danych w Q

Producent:

```
while true:  
    d = produkuj()  
    wait(miejsce)  
    enqueue(Q, d) ← operacja niepodzielna  
    signal(dane)
```

Konsument:

```
while true:  
    wait(dane)  
    d = dequeue(Q, d) ←  
    signal(miejsce)  
    konsumuj(d)
```

$$|\text{miejsce}| + |\text{dane}| = N$$

Stwierdzenie W powyższym rozwiązaniu nie ma blokady

Stwierdzenie W powyższym rozwiązaniu nie ma zagłodzenia.
Przy większej ilości konsumentów i producentów to stwierdzenie jest prawdziwe przy założeniu, że semafory są silnie uczciwe

rozwiązanie z semaforami binarnymi

Queue Q // rozmiar N, początkowo pusta

Semafor niepusty = 0

Semafor niepełny = 0

Semafor semL = 0

licznik = 0 ← ile t w Q

Producent:

licznikP = 0

while true

d = produkuj()

if licznikP == N

wait(niepełny)

enqueue(Q, d)

wait(semL)

licznik=licznik+1

licznikP=licznik

signal(semL)

if licznikP == 1

signal(niepusty)

Konsument:

licznikK = 0

while true

if licznikK == 0

wait(niepusty)

d = dequeue(Q)

wait(semL)

licznik=licznik-1

licznikK=licznik

signal(semL)

if licznikK == N-1

signal(niepełny)

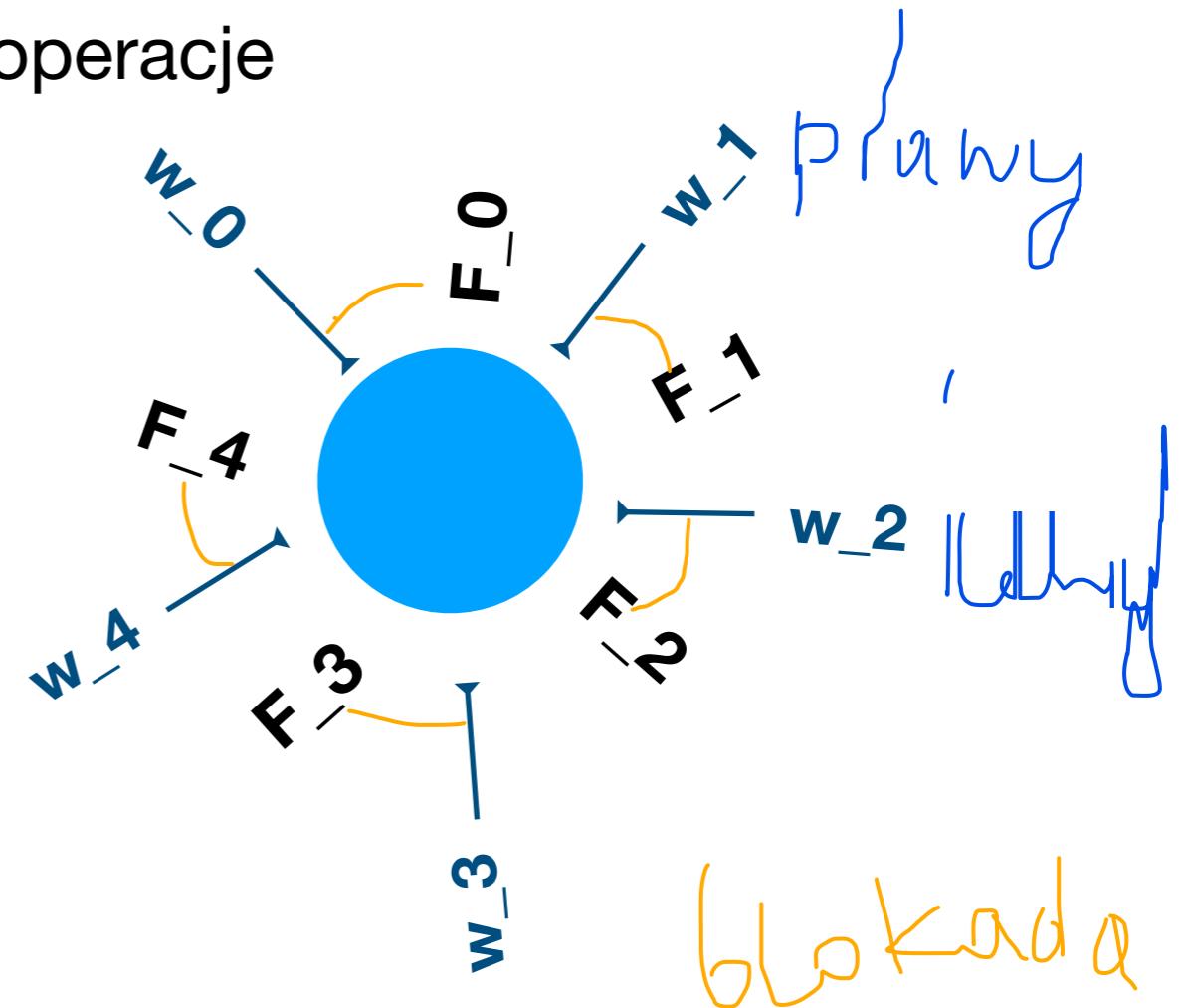
konsumuj(d)

Problem pięciu filozofów

Pięciu filozofów wykonuje następujące operacje

F_i:

```
while true  
myślenie_i  
weź_dwa_widelce  
jedzenie_i  
odłóż_dwa_widelce
```



- filozof potrzebuje dwa widelce, żeby jeść
- dwóch filozofów nie może równocześnie używać tych samych widelców
- filozof może brać te widelce, które są koło niego i to pojedynczo

Wymagane własności

- wzajemne wykluczanie: dwóch filozofów nie może równocześnie trzymać tego samego widelca
- brak blokady
- nikt nie może być zagłodzony
- efektywne zachowanie przy braku współzawodnictwa

Pierwszy szkic rozwiązania

F_i:

```
while true
    myślenie_i
    weź_widelic(i) prawy
    weź_widelic((i+1) mod 5) lewy
    jedzenie_i
    odłóż_widelic(i)
    odłóż_widelic((i+1) mod 5)
```

nie zapewnia wzajemnego wykluczania przy pobieraniu widelców (może dojść do szarpaniny)

potrzebne akcje krytyczne
dla operacji na widelcach

Rozwiążanie z semaforami

Tablica pięciu semaforów $w[0], \dots, w[4]$ o wartościach początkowych 1

F_i :

```
while true
    myślenie_i
    wait(W[i])           weź(i)
    wait(W[(i+1) mod 5])  jedzenie_i
    jedzenie_i
    signal(W[i])          vóz(i+1)
    signal(W[(i+1) mod 5]) dodaj(i)
```

zapewnia wzajemne wykluczanie przy pobieraniu widelców,

jest możliwość blokady

Poprawione rozwiązanie z semaforami

Tablica pięciu semaforów $w[0], \dots, w[4]$ o wartościach początkowych 1

Semafor J o wartości początkowej 4 (ograniczający ilość filozofów, którzy mogą zaczynać pobierać widelce

F_i :

```
while true
    myślenie_i
    wait(J)
    wait(W[i])
    wait(W[(i+1) mod 5])
    jedzenie_i
    signal(W[i])
    signal(W[(i+1) mod 5])
    signal(J)
```

- zapewnia wzajemne wykluczanie przy pobieraniu widelców,
- nie ma blokady
- nie ma zagłodzenia, przy założeniu, że semafor J ma kolejkę czekających

- nie ma zagłodzenia, przy założeniu, że semafor J ma kolejkę czekających

Uzasadnienie

Rozpatrujemy i wykluczamy trzy przypadki możliwego zagłodzenia:

- nieskończone czekanie na $\text{wait}(J)$
- nieskończone czekanie na $\text{wait}(W[i])$
- nieskończone czekanie na $\text{wait}(W[(i+1) \bmod 5])$

Wszystko < ekacja na
↓ dalszym, wykazane
górę

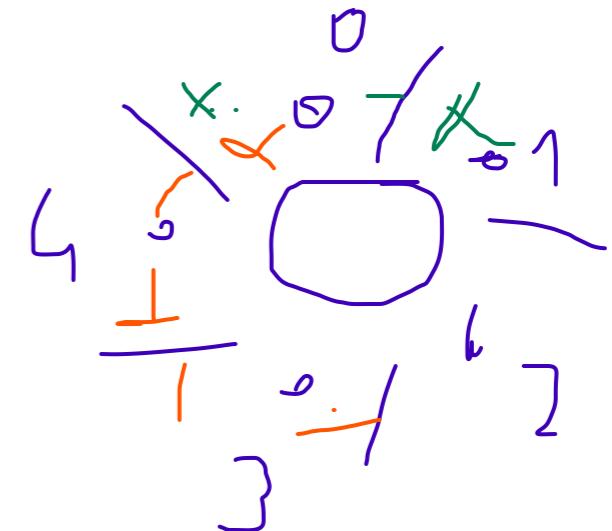
Rozwiążanie z semaforami asymetryczne

Tablica pięciu semaforów $w[0], \dots, w[4]$ o wartościach początkowych 1

F_0, \dots, F_3 jak poprzednio

F_4 :

```
while true
    myślenie_i
        wait(W((i+1) mod 5)) // wait(W[0])
        wait(W[i])           // wait(W[4])
        jedzenie_i
            signal(W[i])      // signal(W[4])
            signal(W((i+1) mod 5)) // signal(W[0])
```



- zapewnia wzajemne wykluczanie przy pobieraniu widelców,
- nie ma blokady
- nie ma zagłodzenia

Monitor - koncepcja

monitor M

warunki w, w1, w2, ...

zmienne

funkcja1()

...

wait(w)

...

funkcja2()

...

signal(w)

...

funkcje monitora wykonywane są na zasadzie **wzajemnego wykluczania**

dostęp do zmiennych monitora też na zasadzie **wzajemnego wykluczania**

i zbiornik monitora

wait(w) zawiesza proces, który wykonał wait(w) do czasu, gdy inny proces wykona signal(w)

signal(w) wznowia jeden z procesów czekających na warunek w

Koncepcja monitora jest zrealizowana np. w Javie, chociaż nazwa "monitor" nie występuje jawnie w konstrukcjach języka (metody i bloki synchronized) ale pojawia się w wyjątkach, jeżeli coś zrobimy źle

Rozwiązanie z monitorem dla pięciu filozofów

Przypomnijmy działanie filozofa

F_i:

```
while true
    myślenie_i
    wesz_widelce(i)
    jedzenie_i
    odłóż_widelce(i)
```

monitor pięciu_filozofów

warunki $M[0], \dots, M[4]$

Tablica liczb $W[0], \dots, W[4]$ o wartościach początkowych 2

// $W[i]$ = ilość widelców dostępnych dla filozofa F_i

wróć_widelce(i)

if $W[i] \neq 2$

wait($M[i]$)

$W[(i+1) \bmod 5] = W[(i+1) \bmod 5] - 1$

$W[(i-1) \bmod 5] = W[(i-1) \bmod 5] - 1$

odłóż_widelce(i)

$W[(i+1) \bmod 5] = W[(i+1) \bmod 5] + 1$

$W[(i-1) \bmod 5] = W[(i-1) \bmod 5] + 1$

if $W[(i+1) \bmod 5] == 2$

signal($M[(i+1) \bmod 5]$)

if $W[(i-1) \bmod 5] == 2$

signal($M[(i-1) \bmod 5]$)

] = il. jedzacych

$\sum W_i = 10 - 2 \cdot 3$

W_0

F_6

F_1

W_1

2

1

W_3

F_2

6

W_2

1

6

1

0

Cze ka na $M_i \Rightarrow W_i < 2$



Rozwiązańie z monitorem

blokada : $\{ \} = 0 \Rightarrow \sum w_i = 10$ SPOŁECZNOŚĆ

wszystcy czekają $\Rightarrow \sum w_i \leq 5$

F_i:

```
while true  
myślenie_i  
weź_widelce(i)  
jedzenie_i  
odłóż_widelce(i)
```

- zapewnia wzajemne wykluczanie przy pobieraniu widelców,
- nie ma blokady
- może być zagłodzenie, nawet przy założeniu, że signal() i wait() kolejują procesy uczciwie

Problem czytelników i pisarzy

Abstrakcja dostępu do bazy danych

Czytelnicy dczytują dane zapisane przez pisarzy, nie wykluczają się wzajemnie

Pisarze tworzą jakieś dane i je zapisują, wykluczają się wzajemnie i również wykluczają czytelników

Działanie czytelników i pisarzy

Czytelnik

```
while true
    zaczni_j_czytanie()
    czytaj
    zakońc_z_czytanie()
```

Pisarz

```
while true
    zaczni_j_pisanie()
    pisz
    koniec_pisania()
```

Rozwiążanie z monitorem

potrzebna jest jeszcze jedna funkcja monitora:

`nonempty(W)`

gdzie `W` jest warunkiem,

zwraca wartość logiczną informującą, czy kolejka procesów oczekujących na warunku `W` jest niepusta

Rozwiążanie z monitorem

monitor czytelnicy_pisarze

warunki: możliwaPisać, możliwaCzytać

czytelnicy = 0 // ilość czytających

pisanie = false // czy ktoś pisze

zaczni_j_czytanie()

if pisanie or nonempty(możnaPisać)

 wait(możnaCzytać)

czytelnicy = czytelnicy +1

signal(możnaCzytać) ← dopuszczalny

zakończ_czytanie()

czytelnicy = czytelnicy-1

if czytelnicy == 0

 signal(możnaPisać)

```
zaczni_j_pisanie()
  if czytelnicy != 0 or pisanie
    wait(możnaPisać)
pisanie = true
```

```
koniec_pisania()
  pisanie = false
  if nonempty(możnaCzytać) ← pierwszeństwo
    signal(możnaCzytać)
  else
    signal(możnaPisać)
```

dla czytelni / kobiety

Kolejność wykonywania czytania i pisania wynikająca z synchronizacji przez monitor:

- jeżeli są oczekujący piarze, to nowy czytelnik musi zaczekać do co najmniej zakończenia pisania przez jednego pisarza
- jeżeli są oczekujący czytelnicy, to wszyscy będą wznowieni przed kolejnym pisaniem

Poprawność rozwiązania

oznaczmy:

- C liczba aktualnie czytających procesów
- P liczba aktualnie piszących procesów

Stwierdzenie. Jeżeli pewne procesy czytają, to nikt nie pisze, a jeżeli ktoś pisze, to jest tylko jeden pisarz i nikt nie czyta.

$$(C>0 \Rightarrow P=0) \wedge (P>0 \Rightarrow (P=1 \wedge C=0))$$

Szkic uzasadnienia

Zauważmy:

C = czytelnicy

$P > 0 \Leftrightarrow$ pisanie

$\text{nonempty}(\text{możnaCzytać}) \Rightarrow \text{pisanie} \vee \text{nonempty}(\text{możnaPisać})$

$\text{nonempty}(\text{możnaPisać}) \Rightarrow (\text{czytelnicy} \neq 0 \vee \text{pisanie})$

Te własności można uzasadnić analizując operacje procesów i zakładając, że operacje monitora są wykonywane niepodzielnie i na zasadzie wzajemnego wykluczania

Następnie zauważamy, że formuła

$$(C>0 \Rightarrow P=0) \wedge (P>0 \Rightarrow (P=1 \wedge C=0))$$

jest początkowo prawdziwa i pokazujemy, że każde przejście w działaniu procesów zachowuje jej prawdziwość. To wymaga rozpatrzenia szeregu przypadków.

Na przykład, weźmy $C>0 \Rightarrow P=0$, założymy, że jest to prawda i sprawdźmy, czy może stać się fałszem.

1. $C>0$ i $P=0$ 
2. $C=0$ i $P=0$
3. $C=0$ i $P>0$

W przypadku 1, zmiana implikacji $C>0 \Rightarrow P=0$ z prawdziwej na fałszywą wymagałaby zmiany P z zera na wartość niezerową, co może się zdarzyć tylko gdy jakiś pisarz przejdzie przez zaczni j_pisanie() – wykluczamy możliwość takiego zachowania

itd.

Stwierdzenie. Żaden proces nie będzie zagłodzony.

Szkic uzasadnienia

Rozpatrujemy i wykluczamy możliwe sytuacje wiecznego oczekiwania przez proces.

Weźmy na przykład czytelnika

- czytelnik wiecznie oczekuje na dopuszczenie przez monitor do wykonania funkcji `zaczni_j_czytanie()`
- czytelnik jest wiecznie wstrzymany na `wait(możnaCzytać)`

Podobnie analizujemy pisarza