Akademia Nauk Stosowanych
Wydział Nauk Inżynieryjnych
Kierunek: Informatyka
studia I stopnia, semestr 2

AKADEMIA NAUK STOSOWANYCH W NOWYM SĄCZU

Systemy operacyjne

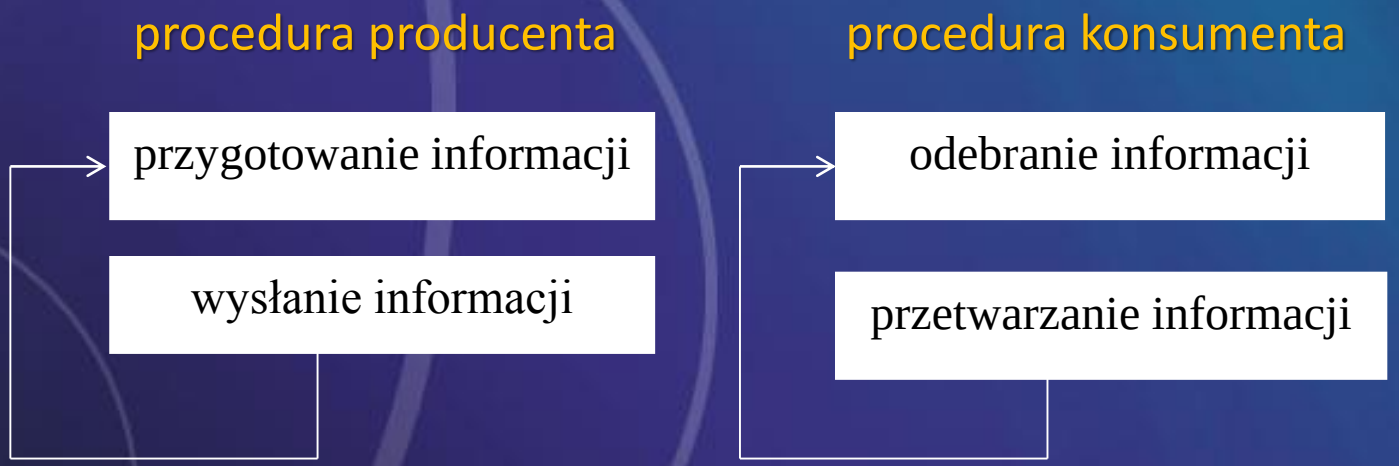
WYKŁAD 9

dr inż. Stanisława Plichta

splichta@ans-ns.edu.pl

Synchronizacja procesów producent - konsument

- W systemie pracuje P (P>=1) procesów producenta i K (K>=1) procesów konsumenta. Każdy proces producenta przygotowuje porcję informacji, a następnie przekazuje ją procesowi konsumenta.
- Procesy producenta i konsumenta muszą podlegać synchronizacji, aby konsument nie próbował konsumować tych jednostek, które nie zostały jeszcze wyprodukowane.



Synchronizacja procesów producent - konsument

- Zakładamy, że operujemy na puli n buforów, z których każdy mieści jedną jednostkę.
- Semafor s_1 umożliwia wzajemne wyłączanie dostępu do puli buforów i ma początkową wartość =1 ($s_1=1$).
- Semafory **pusty** i **pełny** zawierają odpowiednio liczbę pustych i pełnych buforów.
- Semafor **pusty** ma wartość początkową n (**pusty=n**).
- Semafor **pełny** ma wartość początkową 0 (**pełny=0**).

autor: dr inż. Stanisława Plichta

Synchronizacja procesów producent – konsument (semafory)

producent

begin

repeat

produkowanie jednostki

wait (pusty)

wait (s_1)

dodanie jednostki do bufora

signal (s_1)

signal (pełny)

end

konsument

begin

repeat

wait (pełny)

wait (s_1)

pobranie jednostki z bufora

signal (s_1)

signal (pusty)

end

Synchronizacja procesów czytelnicy i pisarze

Dwie grupy procesów silnie konkurujących o zasoby:

- piszący
- czytający

piszący - muszą mieć zapewnione wykluczenie wzajemne względem siebie oraz względem procesów czytających przy korzystaniu z zasobu,

czytający - wiele procesów może jednocześnie być w posiadaniu zasobu, przy czym nie może z niego wtedy korzystać żaden z procesów piszących.

Synchronizacja procesów czytelnicy i pisarze

W rozwiązaniu zastosowano dwa semafory binarne:

- **sp** - dla zapewnienia wykluczenia wzajemnego procesu piszącego względem wszystkich innych procesów
- **w** - dla zapewnienia wykluczenia wzajemnego procesowi czytającemu w chwilach rozpoczynania i kończenia korzystania z zasobu

Faworyzowane są procesy czytające - uzyskują one bezzwłoczny dostęp do zasobu z wyjątkiem chwil, w których korzysta z niego proces piszący.

sp=w=1

Synchronizacja procesów czytelnicy i pisarze (semafory)

```
Czytanie
begin
repeat
  wait (w)
  lc=lc+1
  if lc=1 then wait (sp) end
  signal (w)
  czytanie
  wait (w)
 lc=lc-1
 if lc=0 then signal (sp) end
 signal (w)
end
```

```
Pisanie
begin
repeat
  wait (sp)
   pisanie
  signal (sp)
  end
```

Synchronizacja procesów czytelnicy i pisarze (semafory)

priorytet dla procesów piszących

Potrzebne semafory:

- **w1** - wykluczenie wzajemne procesów czytających w chwili rozpoczynania i kończenia korzystania z zasobu,
- **sp -** wykluczenia wzajemnego procesu piszącego względem wszystkich innych procesów
- **sc** - ochrona wejścia do sekcji krytycznej procesu czytającego
- **w2** - wykluczenie wzajemne procesów piszących w chwili rozpoczynania i kończenia korzystania z zasobu
- **w3 -** zapewnienie priorytetu pisania nad czytaniem

w1=w2=w3=sc=sp=1

Synchronizacja procesów czytelnicy i pisarze (semafory)

Czytanie

```
begin
repeat
  wait (w3)
    wait (sc)
      wait(w1)
        lc=lc+1
        if lc=1 then wait (sp) end
      signal (w1)
    signal(sc)
 signal(w3)
 czytanie
wait (w1)
 lc=lc-1
 if lc=0 then signal (sp) end
signal (w1)
 end
```

Pisanie

```
begin
repeat
 wait(w2)
   lp=lp+1
   if lp=1 then
     wait(sc) end
signal (w2)
wait(sp)
pisanie
signal(sp)
wait(w2)
lp=lp-1;
if lp=0 then
   signal(sc) end;
signal(w2)
end
```

Synchronizacja procesów czytelnicy i pisarze (monitor)

```
monitor czytelnicy_i_pisarze;
var      licznik: integer;
czytelnicy, pisarze: condition;
```

```
procedure wejście_czytelnika;
begin
                if (licznik = -1) or not empty(pisarze) then
                    wait(czytelnicy);
                licznik := licznik + 1;
                signal(czytelnicy)
end;
```

```
procedure wyjście_czytelnika;
begin
        licznik := licznik - 1;
        if licznik = 0 then signal(pisarze)
end;
```

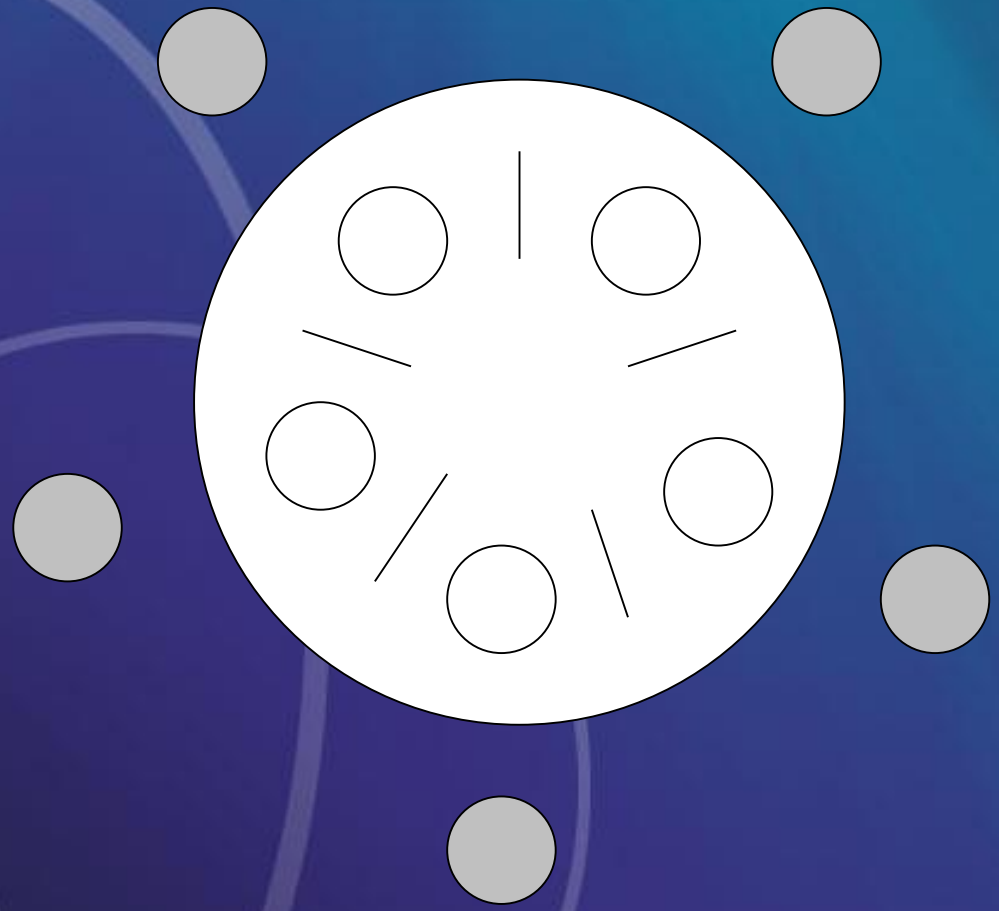
Synchronizacja procesów czytelnicy i pisarze (monitor)

```
procedure wejście_pisarza;
begin
      if licznik ≠ 0 then wait(pisarze);
      licznik := -1
end;
```

```
procedure wyjście_czytelnika;
begin
      licznik := licznik - 1;
      if licznik = 0 then signal(pisarze)
end;
```

```
begin
      licznik := 0;
end;
```

Synchronizacja procesów pięciu filozofów



autor: dr inż. Stanisława Plichta

Synchronizacja procesów pięciu filozofów (semafory)

```
begin
repeat
    myślenie;
    wait(sem[name]);
    wait(sem[(name+1)mod 5]);
    request (widelec[name], widelec[(name+1)mod5]);
    jedzenie;
    release(widelec[name], widelec[(name+1)mod5]);
end
```

możliwość zakleszczenia

Mechanizmy IPC

Mechanizmy IPC (Inter Process Communication) to grupa mechanizmów komunikacji i synchronizacji procesów działających w ramach tego samego systemu operacyjnego

Mechanizmy IPC obejmują:

- **Kolejki komunikatów** — umożliwiają przekazywanie określonych porcji danych.
- **Pamięć współdzieloną** — umożliwiają współdzielenie kilku procesom tego samego fragmentu wirtualnej przestrzeni adresowej,
- Semafory — umożliwiają synchronizacje procesów w dostępie do współdzielonych zasobów np. do pamięci współdzielonej

autor: dr inż. Stanisława Plichta

Utworzenie unikalnego klucza

key_t ftok(const char *path, int id)

- Zwraca ona numer klucza w oparciu o path pliku
- Parametr id daje dodatkowy poziom niepowtarzalności
- Ta sama path dla różnych id daje różne klucze.

- Klucze są liczbami używanymi do identyfikacji obiektów IPC w systemie UNIX

np. ftok(".", 'A')

Mechanizmy IPC

Działanie funkcji	kolejka komunikatów	pamięć współdzielona	semafory
Rezerwowanie obiektu IPC oraz uzyskiwanie do niego dostępu	msgget	shmget	semget
Sterowanie obiektem IPC, uzyskiwanie informacji o stanie modyfikowanych obiektów IPC, usuwanie obiektów IPC	msgctl	shmctl	semctl
Operacje na obiektach IPC: wysyłanie i odbieranie komunikatów, operacje na semaforach, rezerwowanie i zwalnianie segmentów pamięci wspólnej	msgsnd, msgrcv	shmat, shmdt	semop

Mechanizmy IPC

- Informacje na temat konkretnych obiektów: kolejek komunikatów, pamięci współdzielonej i semaforów otrzymamy stosując odpowiednio

przełączniki -q, -m, -s

- Informacja na temat kolejki komunikatów o identyfikatorze msgid

ipcs -q msgid

- Informacja na temat segmentu pamięci współdzielonej o identyfikatorze shmid

ipcs -m shmid

- Informacja na temat zestawu semaforów o identyfikatorze semid

ipcs -s semid

- Dodatkowo przełącznik -b pozwala uzyskać informację o maksymalnym rozmiarze obiektów IPC, czyli ilości bajtów w kolejkach, rozmiarze segmentów pamięci współdzielonej i ilości semaforów w zestawach.

Mechanizmy IPC

- Usunięcia obiektu IPC można dokonać wykonując polecenie systemowe ipcrm:
- Usunięcie kolejki komunikatów o identyfikatorze msgid

ipcrm -q msgid

- Usunięcie segmentu pamięci współdzielonej o identyfikatorze shmid

ipcrm -m shmid

- Usunięcie zestawu semaforów o identyfikatorze semid

ipcrm -s semid

Semafory

- Funkcja **semget** służy do alokowania semaforów, na podstawie klucza tworzy lub umożliwia nam dostęp do zbioru semaforów.

int semget(key_t key, int nsem, int permflags)

- Parametr **key** jest kluczem do zbioru semaforów.
- Jeżeli różne procesy chcą uzyskać dostęp do tego samego zbioru semaforów muszą użyć tego samego klucza.
- Parametr **nsem** to liczba semaforów, która ma znajdować się w tworzonym zbiorze.

Semafory

- Parametr **permflags** określa prawa dostępu do semaforów oraz sposób wykonania funkcji. Może przyjmować następujące wartości:
  - **IPC_CREAT** - uzyskanie dostępu do zbioru semaforów lub utworzenie nowego gdy zbiór nie istnieje
  - **IPC_EXCL** - w połączeniu z **IPC_CREAT** zwraca błąd gdy zbiór już istnieje - prawa dostępu (tak samo jak dla plików np. 0600)
- Poszczególne flagi można łączyć ze sobą przy pomocy sumy bitowej..
- Funkcja zwraca identyfikator zbioru semaforów lub -1 gdy wystąpił błąd (ustawiana jest zmienna errno)

Semafory

Z każdym semaforem w zestawie związane są następujące wartości:

- **semval** – wartość semafora zawsze dodatnia liczba całkowita (musi być ustawiona za pomocą funkcji systemowej) – semafor nie jest bezpośrednio dostępny dla programu jako obiekt.
- **sempid** – identyfikator procesu, który ostatnio miał do czynienia z semaforem.
- **semcnt** – liczba procesów, które czekają, aż semafor osiągnie wartość większą od aktualnej.
- **semzcnt** - liczba procesów, które czekają, aż semafor osiągnie wartość zerową.

Semafory

Utworzenie zbioru 3 semaforów dla klucza key oraz przypisanie zmiennej semid identyfikatora tego zbioru

```
key_t key;
int semid;
key = ftok(".", 'A');
semid=semget(key,3,IPC_CREAT|0666)
```

Semafory – wykonanie operacji semaforowej

- Operacja semaforowa może być wykonywana jednocześnie na kilku semaforach w tej samej tablicy identyfikowanej przez semid.

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

- sops - wskaźnik do tablicy operacji semaforowych (zawiera wartość, która zostanie dodana do zmiennej semaforowej pod warunkiem, że zmienna semaforowa nie osiągnie w wyniku tej operacji wartości mniejszej od 0).

- nsops - liczba elementów tablicy operacji semaforowych.

Semafory – wykonanie operacji semaforowej

Każdy element tablicy opisuje jedną operację semaforową i ma następującą strukturę:

```
struct sembuf {
/* indeks  semafora w zestawie  */
short sem_num;
/* operacja semaforowa - zawiera liczbę, która określa,
jakie zadanie ma być wykonane */
short sem_op;
/* flagi operacji */
short sem_flg;
};
```

Semafory – wykonanie operacji semaforowej

Funkcja semctl wykonuje operację sterującą określoną przez *cmd* na zestawie semaforów określonym przez *semid* lub na określonym semaforze tego zestawu wskazanym przez *semnum*

semctl(int semid, int semnum, int cmd, union semun arg)

semid – identyfikator semafora zwracany przez funkcję semget

semnum – używany do identyfikacji konkretnego semafora

cmd - mówi systemowi jaka operacja ma być wykonana, podając dokładnie wymaganą funkcję

numeracja semaforów zaczyna się od 0

kody funkcji semctl
operacje na pojedynczym semaforze

/* Kopiowanie informacji ze struktury kontrolnej zestawu semaforów do struktury wskazywanej przez ***arg*.buf** - proces wywołujący funkcję musi mieć prawa do odczytu zestawu semaforów

IPC_STAT

/* Zapis wartości niektórych pól struktury **semid_ds** wskazywanej przez ***arg*.buf** do struktury kontrolnej zestawu semaforów */

IPC_SET

/* natychmiastowe usunięcie zestawu semaforów i związanych z nim struktur danych */

IPC_RMID

kody funkcji semctl
operacje na pojedynczym semaforze

/* zwraca wartość **semval** semafora o numerze **semnum** w zestawie

GETVAL

/* ustawia wartość semafora o numerze semnum w zestawie

SETVAL

/* zwraca wartość sempid (identyfikator procesu, który ostatnio miał do czynienia z semaforem)

GETPID

/* zwraca wartość **semncnt** skojarzoną z semaforem numer *semnum* (tzn. liczbę procesów oczekujących na zwiększenie się wartości **semval** skojarzonej z semaforem numer *semnum*)

GETNCNT

kody funkcji semctl
operacje na pojedynczym semaforze

/*zwraca wartość semzcnt skojarzoną z semaforem o numerze *semnum* w zestawie. (tzn. liczbę procesów oczekujących na osiągnięcie przez semafor o numerze *semnum* wartości 0)

GETZCNT

/* zwraca wartości **semval** wszystkich semaforów z zestawu umieszczając je w tablicy *arg*.**array**

GETALL

/* przypisuje wartości **semval** wszystkim semaforom zestawu, korzystając z tablicy *arg*.**array**

SETALL

Podstawowe elementy programów

Pliki nagłówkowe

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

Podstawowe elementy programów

```
union semun
   {
/* wartość dla SETVAL */
 int val;

/* bufor dla IPC_STAT i IPC_SET */
struct semid_ds *buf;

/* tablica dla GETALL i SETALL */
short *array;
};
union semun sem_ctrl;
```

Podstawowe elementy programów

tworzenie klucza

```
klucz=ftok(".",'A');
```

tworzenie zbioru semaforów (1 semafor)

```
If ((semid=semget(klucz,1,IPC_CREAT|
                          IPC_EXCL|0666)) == -1)
{
      perror("Blad funkcji semget\n");
      exit(1);
  }
```

Podstawowe elementy programów

zainicjowanie semafora

```
sem_ctrl.val=1;
if(semctl(semid,0,SETVAL,sem_ctrl) == -1)
{
        perror("Blad  semctl\n");
        exit(2);
}
```

Podstawowe elementy programów

usunięcie zbioru semaforów

```
union semun un_sem;
If (semctl(semid,0,IPC_RMID,un_sem)==-1)
{
    perror("Blad przy usuwaniu semafora\n");
    exit(5);
  }
```

Podstawowe elementy programów

pobranie i wypisanie wartości semafora

```
printf("Oczekuje na sekcje krytyczna  %d
semafor =%d\n",getpid(),semctl(semid,0,GETVAL,0));
```

sekcja krytyczna

```
sem_opusc();

printf("\n w sekcji krytycznej     %d     semafor = %d\n",
getpid(), semctl(semid,0,GETVAL,0));

sem_podnies();
```

Podstawowe elementy programów

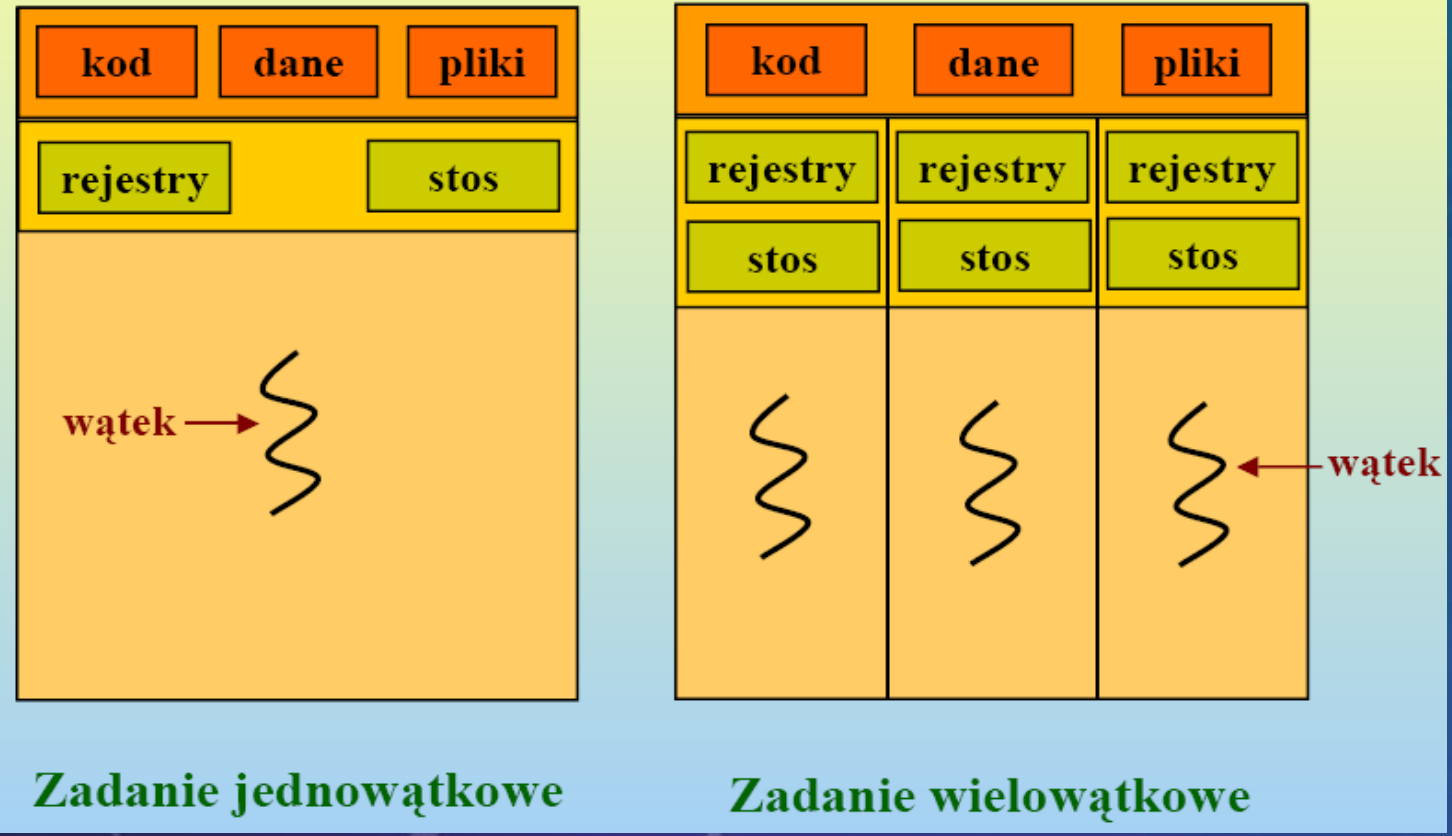
```
int sem_opusc()
 {
  struct sembuf b_sem;
  b_sem.sem_num=0;
  b_sem.sem_op=-1;
  b_sem.sem_flg=SEM_UNDO;
  semop(semid,&b_sem,1);
 return(1);
 }
```

```
int sem_podnies()
 {
  struct sembuf b_sem;
  b_sem.sem_num=0;
  b_sem.sem_op=1;
  b_sem.sem_flg=SEM_UNDO;
  semop(semid,&b_sem,1);
 return(1);
 }
```

Wątki (threads)

- Wątek *(thread)*, zwany także procesem lekkim *(light-weight process*), jest podstawową jednostką wykorzystania CPU.
- Posiada: licznik rozkazów, zbiór rejestrów i obszar stosu.
- Wątek dzieli wraz z innymi równorzędnymi wątkami: sekcję kodu, sekcję danych oraz zasoby systemowe (otwarte pliki, sygnały itd.)
- Tradycyjny proces, tzw. ciężki *(heavy-weight)*, jest równoważny zadaniu z jednym wątkiem.
- Zadanie nic nie robi, jeśli nie ma w nim ani jednego wątku. Wątek może przebiegać dokładnie w jednym zadaniu.

Zadania jedno i wielowątkowe



Implementacja wątków

- **Wątki poziomu użytkownika** *(user-level threads)* – tworzone za pomocą funkcji bibliotecznych - przełączanie między wątkami nie wymaga wzywania systemu operacyjnego (POSIX **Pthreads**).

  Zalety:

  – Szybkie przełączanie między wątkami.

  – Wydajne obsługiwanie wielu zamówień.

  Wady:

  – Przy jednowątkowym jądrze każde odwołanie wątku poziomu użytkownika do systemu powoduje wstrzymanie całego zadania.

  – Nieadekwatny przydział czasu procesora (zadanie wielowątkowe i jednowątkowe mogą dostawać tyle samo kwantów czasu).

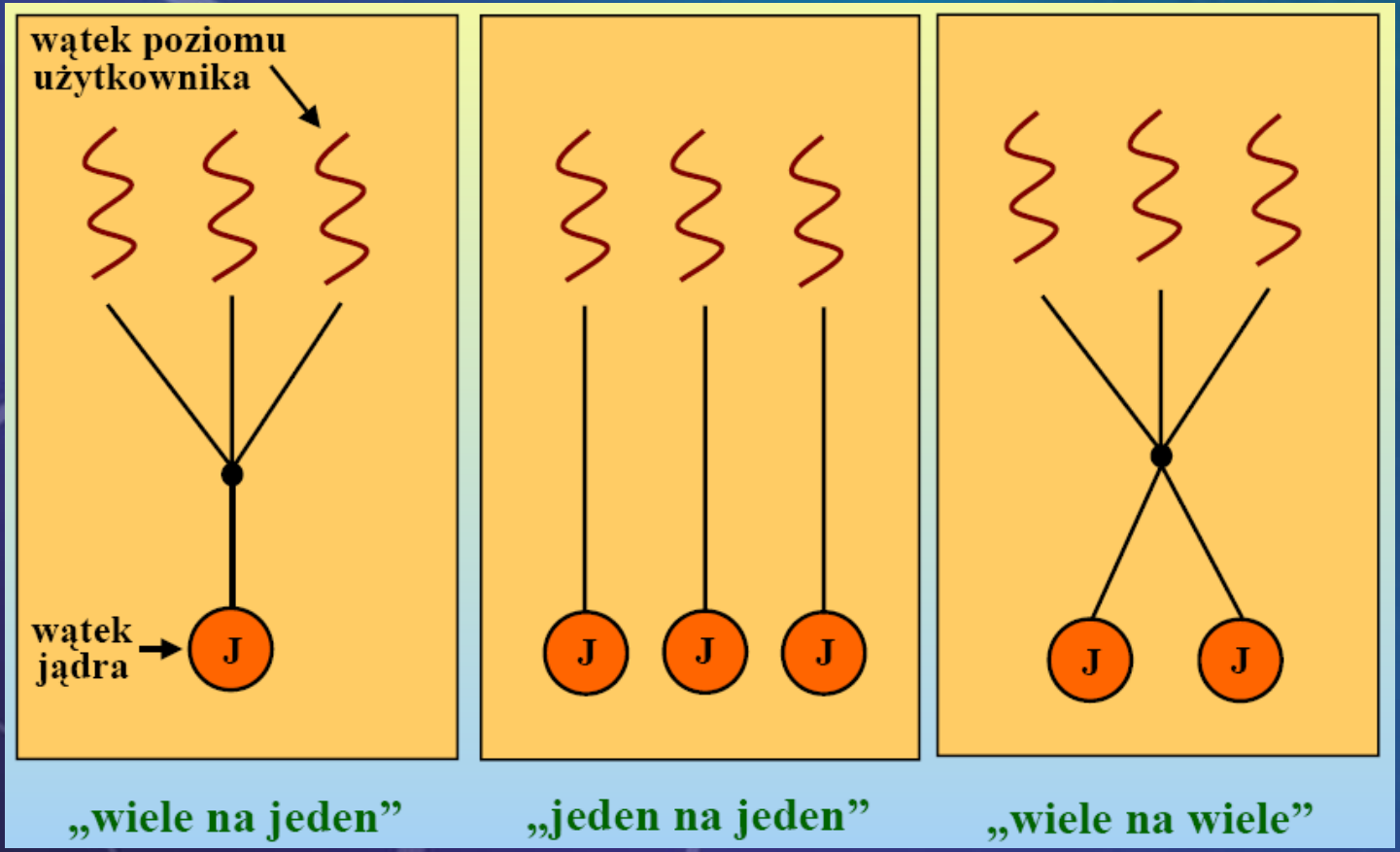
- **Wątki jądra** *(kernel threads)* – obsługiwane przez jądro systemu (np. systemy Windows, Solaris, Tru64 UNIX).

  Zalety: Wydajniejsze planowanie przydziału czasu procesora.

  Wady: Wolniejsze przełączanie wątków – zajmuje się tym jądro (za pomocą przerwań).

- **Wątki mieszane** – zrealizowane oba rodzaje wątków (Solaris 2).

Schemat wielowątkowości



autor: dr inż. Stanisława Plichta

Przykłady implementacji wątków

P-wątki *(Pthreads)*

- Dostępne głównie w systemach uniksowych (np. Linux, Solaris, Mac OS X).
- MS Windows na ogół ich nie udostępniają, ale można je zainstalować korzystając z oprogramowania *shareware*.
- Udostępniane jako biblioteka poziomu użytkownika – brak wyraźnych związków między P-wątkami, a stowarzyszonymi z nimi wątkami jądra.

Utworzenie wątku

int pthread_create(pthread_t *tid , const pthread_attr_t *attr,

void *(*func)(void *), void *arg)

- Do każdego wątku odnosimy się za pośrednictwem identyfikatora wątku,
- Każdy wątek ma wiele atrybutów, które przy jego tworzeniu można określić – zazwyczaj korzysta się z wartości domyślnych przekazując wskaźnik pusty
  w miejsce argumentu attr

Wątek może pobrać wartość własnego identyfikatora

pthread_t pthread_self(void)

(odpowiednik funkcji getpid())

Przyłączenie i odłączenie wątku

przyłączenie wątku

```
int pthread_join(pthread_t tid,void **status)
```

odłączenie wątku

```
int pthread_detach(pthread_t tid)
```

Wątki - przykład

```
#include <stdio.h>
#include <pthread.h>
#define REENTRANT
int utworz_watek;
int przylacz_watek;
int odlacz_watek;
char bufor[BUFSIZ];
void koniec(char tekst[])
  {
    printf("%s",tekst);
    exit(-1);
  }
```

```
void *rob()
  {
    printf("[watek] Jestem watkiem programu.\n");
    printf("Podaj dane : ");
    fgets(bufor,BUFSIZ,stdin);
    return bufor;
    pthread_exit(0);
  }
```

Wątki - przykład

```
int main()
 {
   pthread_t id_watku;
   printf("[system] Wykonuje funkcje PTHREAD_CREATE ...\n");
   utworz_watek=pthread_create(&id_watku,NULL,rob,NULL);
   if (utworz_watek==-1) koniec("PORAZKA\n");
   printf("[program] Jestem programem glownym.\n");
   printf("[program] Czekam na dane od mojego watku...\n");
   printf("[system] Przylaczam watek ...\n");
   przylacz_watek=pthread_join(id_watku,NULL);
```

Wątki - przykład

```
if (przylacz_watek==-1) koniec("PORAZKA\n");
   printf("[program] Otrzymalem dane ... %s\n",bufor);
   printf("[system] Odlaczam watek...\n");
   odlacz_watek=pthread_detach(id_watku);
   if (odlacz_watek==-1) koniec("PORAZKA\n");
      printf("[Program] Koncze dzialanie.\n");
   exit(0);
 }
```

Do kompilacji należy użyć polecenia

cc –o watek watek.c -lpthread