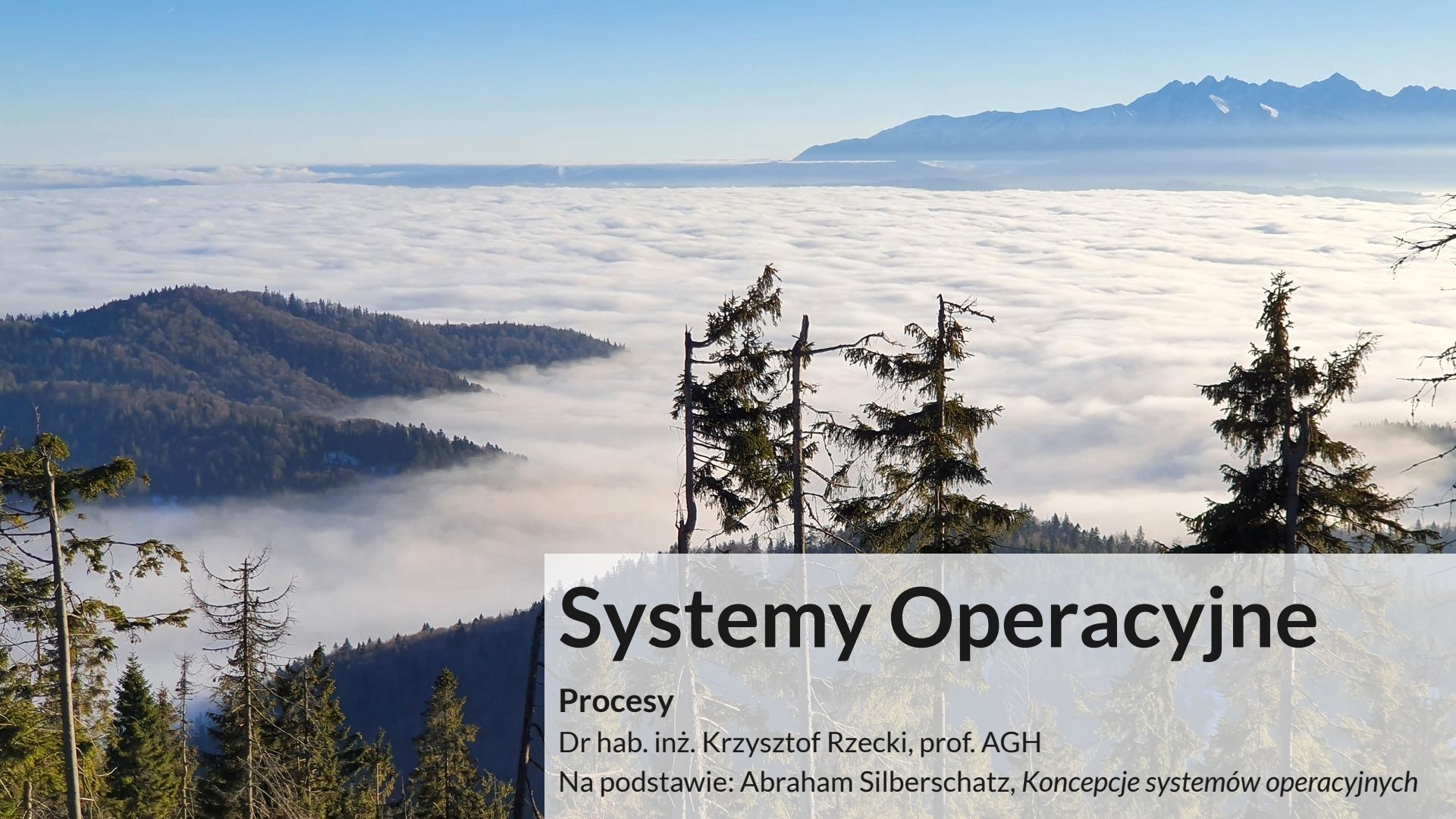


Systemy Operacyjne

Procesy

Dr hab. inż. Krzysztof Rzecki, prof. AGH

Na podstawie: Abraham Silberschatz, *Koncepcje systemów operacyjnych*

Od programu do procesu

Tworzenie programu:

Kod źródłowy -> Kompilacja -> Linkowanie -> Kod binarny

Uruchomienie programu:

Kod binarny, czyli program -> Wczytanie do pamięci operacyjnej -> Nadanie PCB -> Proces

Proces

- Pierwsze systemy operacyjne: jeden program, który ma dostęp do wszystkich zasobów
- Obecnie: zarządzanie **uruchomionymi programami** -> **procesami**
- Proces to inaczej:
  - wczytany do pamięci i uruchomiony kod
  - jednostka pracy w systemie z dzieleniem czasu
- Zarządzanie to obejmuje: kontrola i separacja
- Efekt: system zawiera kolekcje procesów:
  - procesy systemu operacyjnego
  - procesy użytkownika
- Potencjalnie wszystkie w/w procesy uruchamiane są jednocześnie
- Procesor(y) przełączają się między procesami, co zwiększa efektywność systemu komputerowego

Powoływanie i terminowanie procesu (*)

Powoływanie (tworzenie ?):

- Nowe zadanie wsadowe
- Interaktywne logowanie
- Utworzenie usługi przez SO
- Podział (*spawn*) istniejącego procesu

Terminowanie:

- Prawidłowe zakończenie
- Brak pamięci
- Naruszenie ochrony
- Interwencja operatora lub SO

Proces informuje o zakończeniu:

- Instrukcja HALT
- Akcja użytkownika (np. log off)
- Awaria lub błąd
- Terminowanie procesu rodzica

Powoływanie procesu - interfejs

Program w ścieżce:

```
$ xclock
```

Program użytkownika:

```
$ ./program
```

Program w tle:

```
$ xclock &
```

Terminowanie programu:

```
ctrl + c
```

Podgląd zadań:

```
$ jobs
```

Przeniesienie w tło:

```
$ bg %1
```

Wstrzymanie pracy programu:

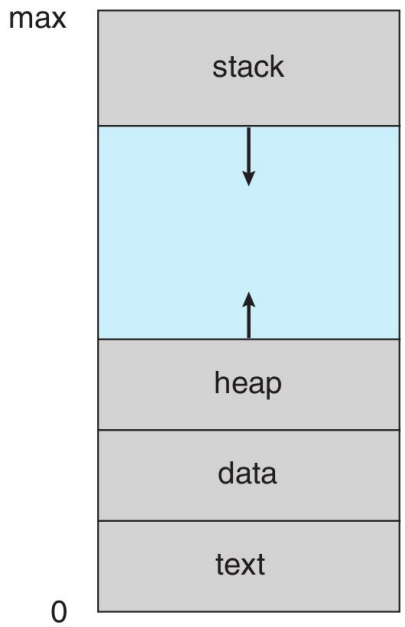
```
ctrl + z
```

Przeniesienie na pierwszy plan:

```
$ fg %1
```

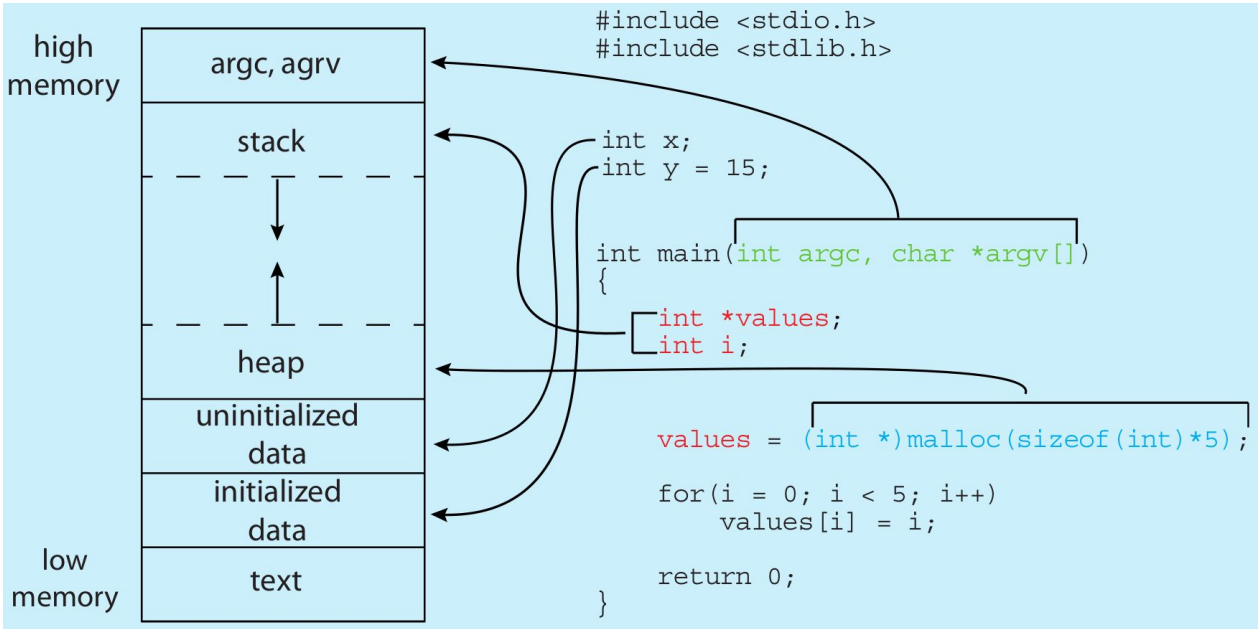
Koncepcja procesu

- System wsadowy: *jobs*, system współdzielony: *user programs* lub *tasks* -> obecnie: *process*
- Proces:
  - Kod programu, *text section*
  - Licznik programu, *program counter* == Licznik rozkazów == Wskaźnik wykonywanej instrukcji
  - Stos procesu, *process stack* (parametry procedur, adresy powrotne, zmienne tymczasowe)
  - Sekcja danych, *data section*, czyli zmienne globalne
  - Sterta, *heap*, czyli pamięć przydzielana dynamicznie
- Program jest pasywny - plik wykonywalny zawierający listę instrukcji
- Proces jest aktywny - licznik rozkazów wskazuje następną instrukcję



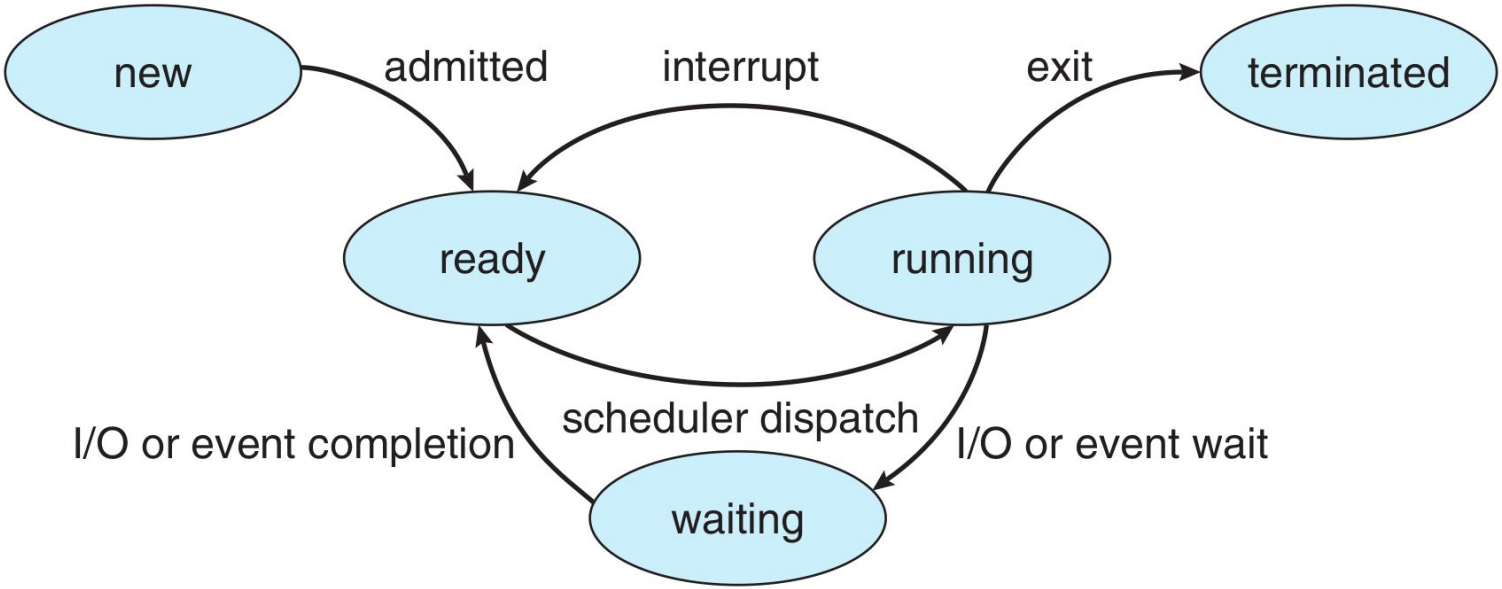
Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Program w C



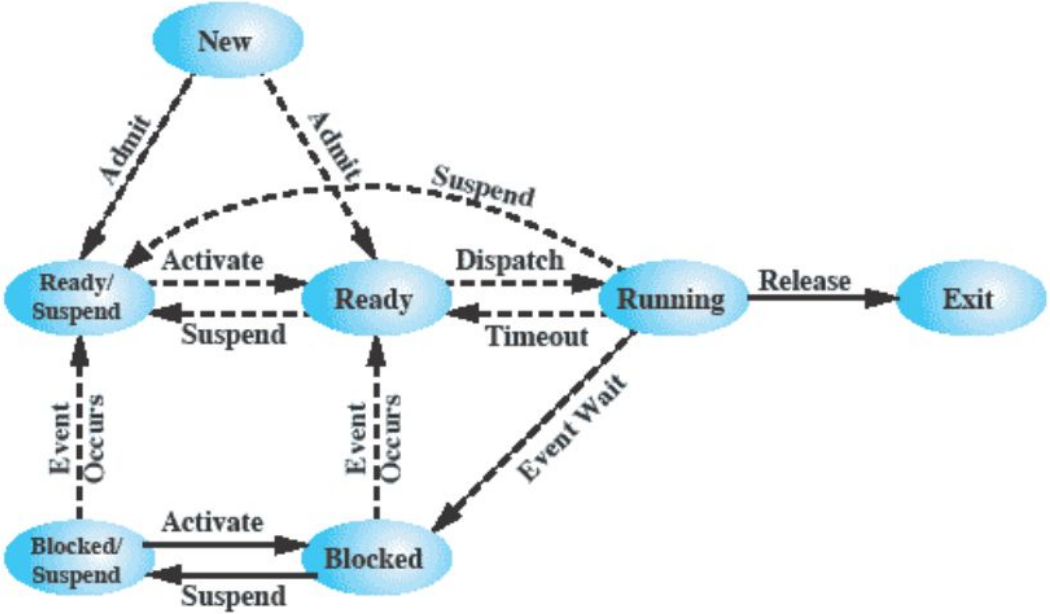
Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Stany procesu (model pięciostanowy)



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Dwa stany zawieszenia (*)



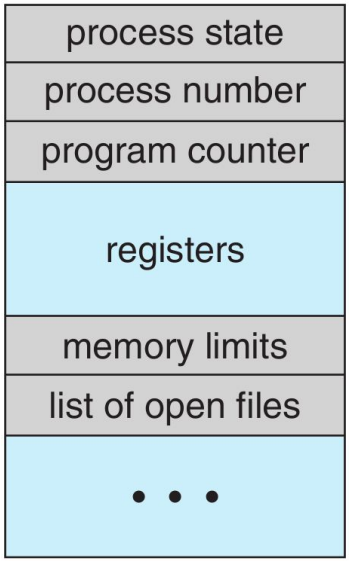
Powody zawieszania procesu (*)

- **Swapping** - system operacyjny musi zwolnić pewną ilość pamięci (głównej), aby uruchomić proces, który jest gotowy do wykonania.
- **Inny problem SO** - błędy.
- **Interaktywne żądanie użytkownika**
- Timing - periodyczne wywołanie procesu (np. monitorowanie) i może zostać on zawieszony do następnego wywołania.
- **Żądanie procesu macierzystego** - SIGSTOP / SIGTSTP / SIGCONT

Blok kontrolny procesu

PCB - ang. *process control block*

Jest to obszar pamięci zawierający różne informacje skojarzone z procesem, którego dotyczy.



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Blok kontrolny procesu

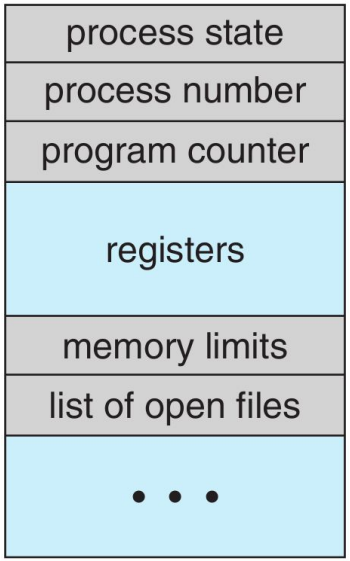
Linux kernel:

```
$ /usr/src/

$ linux-headers-[kernel v.]/

$ include/linux/sched.h
```

Zdefiniowane:

```
task_struct
```

Kilkadziesiąt pól !



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

User mode:

$ /proc/[PID procesu]

$ ps

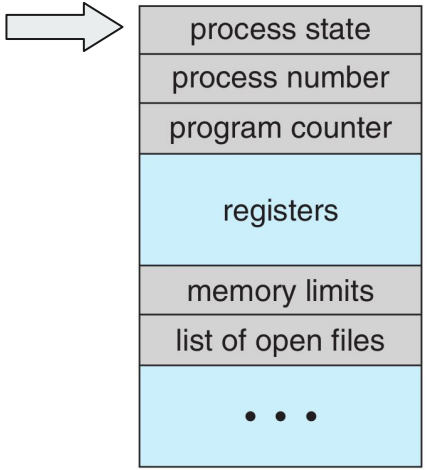
PCB - stan procesu

Process state - stan procesu

$ ps -p [PID procesu] -o pid,status,comm

Lub:

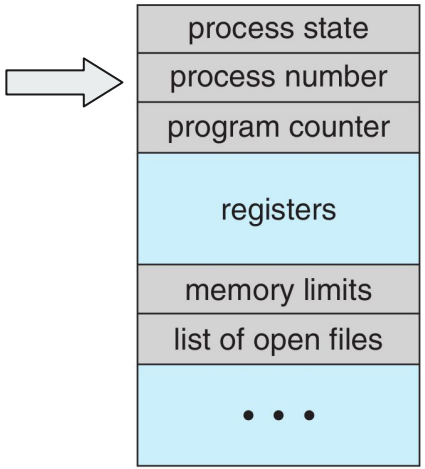
$ cat /proc/[PID procesu]/status | grep Stat



Możliwe stany:

- R=running,
- S=sleeping in an interruptible wait,
- D=waiting in uninterruptible disk sleep,
- Z=zombie,
- T=traced or stopped (on a signal),
- W=paging

PCB - numer procesu



PID - Process identifier - identyfikator procesu

```
$ pidof <nazwa programu>
```

lub:

```
$ ps aux | grep <nazwa programu>
```

W wyniku otrzymamy liczbę, która jest numerem PID procesu, który jest liczbą typu `integer` niepowtarzalną w przestrzeni systemu operacyjnego.

PCB - numer procesu

Odczyt własnego PID (programistycznie):

```
$ pid_t getpid(void);   // unistd.h
```

Odczyt PID rodzica:

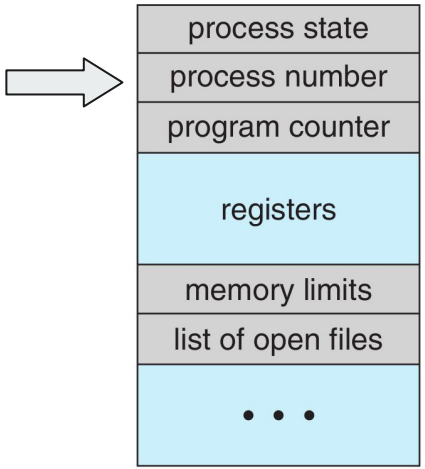
```
$ pid_t getppid(void); // unistd.h
```

Utworzenie procesu i pobranie jego PID:

```
$ pid_t fork(void);     // unistd.h
```

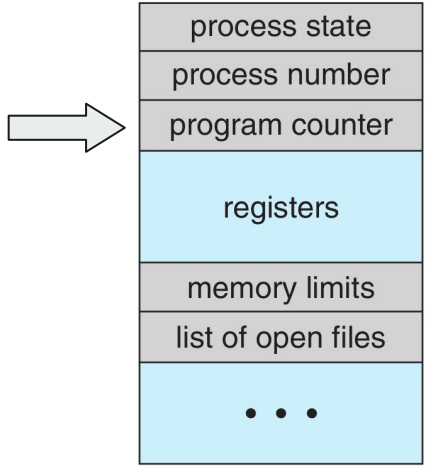
Funkcje, w których argumentem jest PID:

```
kill, wait, nice
```

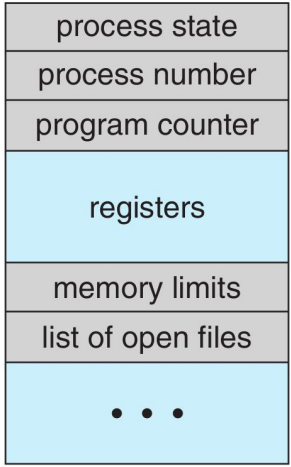


PCB - licznik rozkazów

Licznik rozkazów - program counter - adres kolejnej instrukcji do wykonania



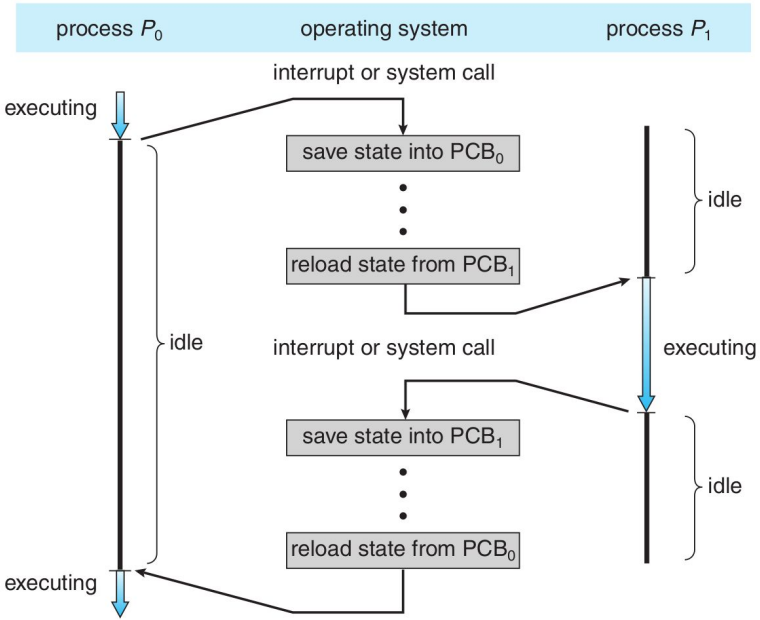
Blok kontrolny procesu - pozostałe pola



Obszar pamięci zawierający:

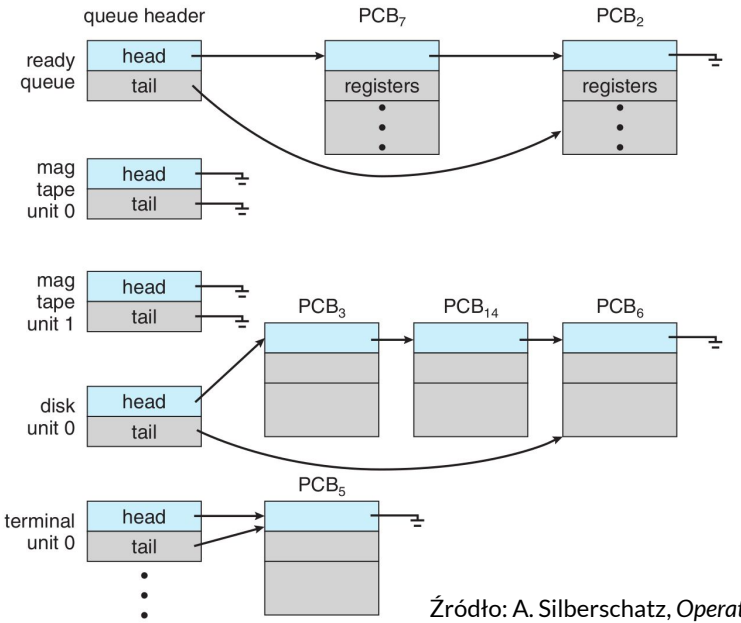
- Licznik rozkazów - program counter - adres kolejnej instrukcji do wykonania
- Rejestry procesora - CPU registers - m.in. akumulatory, rejestry indeksowe, wskaźniki stosu, rejestry ogólnego przeznaczenia, rejestry warunków, etc.
- Informacje o planowaniu przydziału procesora - CPU-scheduling information - m.in. priorytet procesu, wskaźnik do kolejek, etc.
- Informacje o zarządzaniu pamięcią - Memory-management information - m.in. zawartość rejestrów granicznych, tablice stron lub tablice segmentów, etc.
- Informacje do rozliczeń - accounting information - ilość zużytego czasu procesora i czasu rzeczywistego, ograniczenia czasowe, numery kont, numer zadania lub procesu, itp.
- Informacja o stanie wejścia-wyjścia - I/O status information - m.in. lista urządzeń we/wy przydzielonych do procesu, lista otwartych plików, itd.

Blok kontrolny procesu - przełączanie CPU



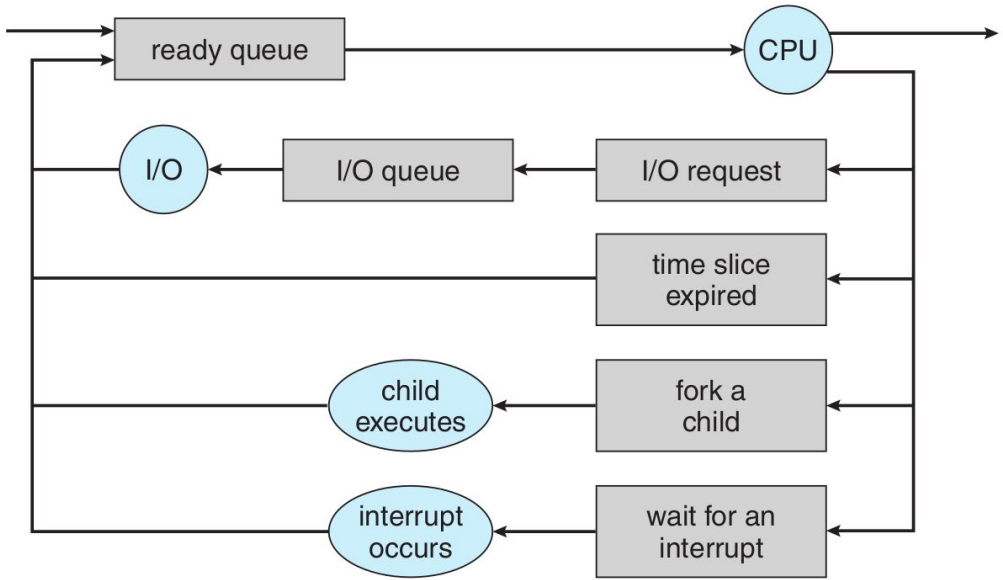
Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Kolejkowanie urządzeń we/wy



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Diagram kolejkowania procesu



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Operacje na procesach - tworzenie procesu (1)

- Tworzenie procesu:
  - Dany proces (rodzic - *parent*) może utworzyć wiele nowych procesów (dziecko - *child*)
  - Każdy z nowo powstałych procesów może tworzyć kolejne, powstaje drzewo - *tree* procesów
  - Identyfikacja procesu w większości systemów odbywa się przez identyfikator PID

```
$ pstree -p | head -n 10
systemd(1)-+-ModemManager(824)-+-{ModemManager}(854)
           |                   `-{ModemManager}(857)
           |-NetworkManager(711)-+-{NetworkManager}(819)
           |                     `-{NetworkManager}(821)
           |-accounts-daemon(10363)-+-{accounts-daemon}(10366)
           |                        `-{accounts-daemon}(10372)
           |-acpid(10526)
           |-agetty(876)
```

```
$ ps -ax | head -n 10
    PID TTY      STAT   TIME COMMAND
      1 ?        Ss     0:01 /sbin/init splash
      2 ?        S      0:00 [kthreadd]
      3 ?        I<     0:00 [rcu_gp]
      4 ?        I<     0:00 [rcu_par_gp]
      6 ?        I<     0:00 [kworker/0:0H-kblockd]
      9 ?        I<     0:00 [mm_percpu_wq]
     10 ?        S      0:00 [ksoftirqd/0]
     11 ?        I      0:01 [rcu_sched]
     12 ?        S      0:00 [migration/0]
```

Pytanie: skąd różnica w nazwie: systemd vs. init ?

Poszukiwanie PID procesu macierzystego

```
$ pidof <nazwa programu>
```

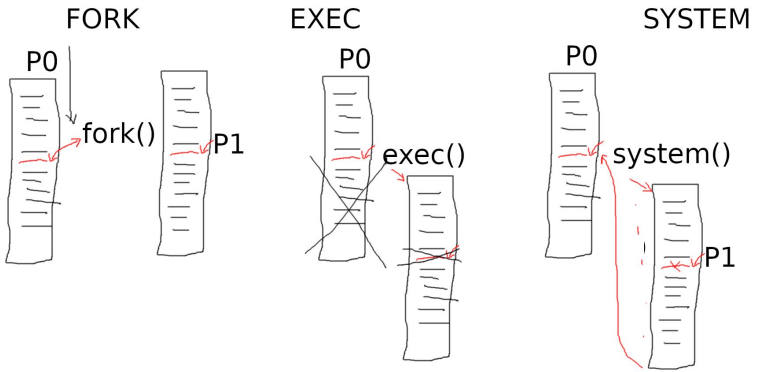
```
$ ps -p `pidof <nazwa programu>` -o ppid,pid,status,comm
```

lub:

```
$ cat /proc/`pidof <nazwa programu>`/status | grep PPid
```

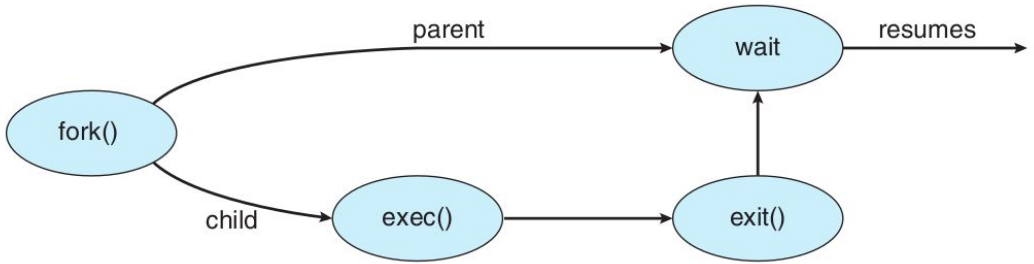
Operacje na procesach - tworzenie procesu (2)

- Kiedy dany proces utworzy proces potomny, mogą zaistnieć dwa przypadki:
  - Proces rodzica wykonuje się nadal jednocześnie z procesem potomnym (fork)
  - Proces rodzica czeka, aż któryś lub wszystkie procesy potomne zakończą działanie (fork lub system)
- Są także dwie możliwości adresowania pamięci dla nowego procesu:
  - Proces potomny jest duplikatem procesu rodzica, tzn. ma ten sam kod programu jak rodzic (fork)
  - Proces potomny to na nowo załadowany program (exec i system)



Operacje na procesach - tworzenie procesu (3)

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```



Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Uruchamianie skryptów, a proces potomny

Tryb bez źródła:

$ ./skrypt.sh

Tryb ze źródłem:

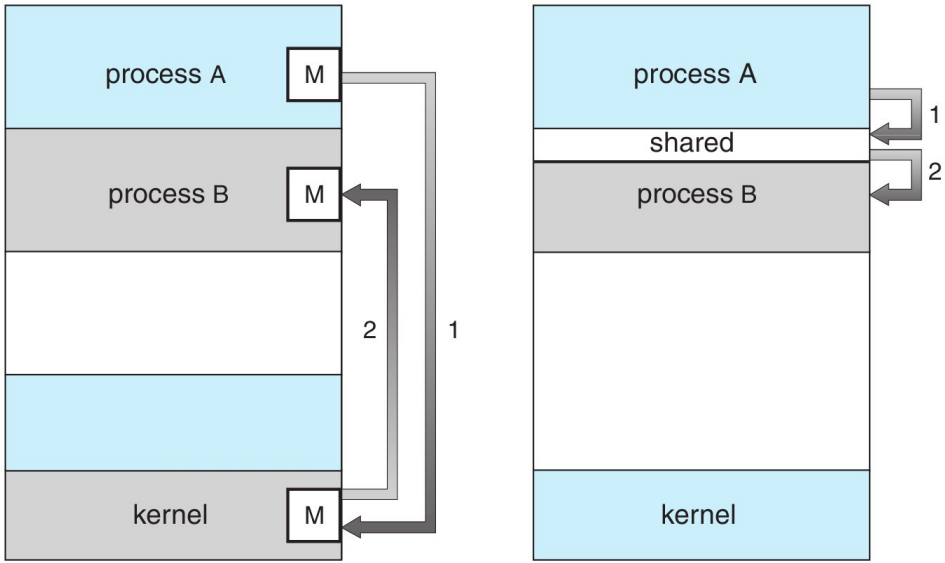
$ . ./skrypt.sh

$ source ./skrypt.sh

Komunikacja międzyprocesowa

- Współistnienie procesów:
  - Proces niezależny - nie wpływają na niego, ani nie wpływa na inne procesy
  - Proces kooperujący - może wpływać na inne procesy lub inne procesy mogą wpływać na niego
- Znaczenie komunikacji międzyprocesowej:
  - Współdzielenie informacji (dane, wymiana komunikatów)
  - Przyspieszenie obliczeń (czy wszystkie można zrównoleglić?)
  - Modularność systemu
  - Wygoda
- **IPC - interprocess communication = komunikacja międzyprocesowa**

Modele komunikacji międzyprocesowej (1)



Message passing

Shared memory

Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Modele komunikacji międzyprocesowej (2)

- Message passing:
  - Użyteczny do wymiany niewielkich ilości danych (brak konieczności zapobiegania konfliktom)
  - Łatwiejszy w implementacji
- Shared memory:
  - Najszybsza możliwa komunikacja
  - Jedyna wymagana interwencja z kernela to utworzenie tej pamięci

Producent i konsument

- Model producent - konsument
  - Serwer www - przeglądarka html
  - Kompilator tex - pdf viewer
  - Etc.

Shared memory - buforowanie (1)

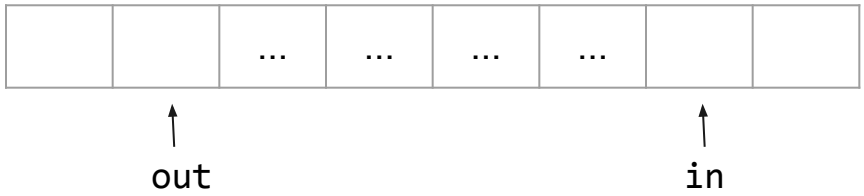
- Bufory wymiany danych:
  - Bufor nieograniczony (unbounded buffer) - brak limitu wielkości:
    - Producent nigdy nie czeka, konsument czeka gdy bufor jest pusty
  - Bufor ograniczony (bounded buffer) - określona wielkość bufora:
    - Producent czeka, gdy bufor jest pełny, konsument czeka gdy bufor jest pusty

Shared memory - buforowanie (2)

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
}item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

- kolejka FIFO
- `in` - następna wolna pozycja w buforze
- `out` - pierwsza wolna pozycja w buforze
- `in == out` - bufor jest pusty
- `((in+1) % BUFFER_SIZE) == out` - bufor jest pełny



Shared memory - buforowanie (3)

Producent

```
item nextProduced;

while (true) {
     /* produce an item in nextProduced */
     while (((in + 1) % BUFFER_SIZE) == out)
        ; /* do nothing */
     buffer[in] = nextProduced;
     in = (in + 1) % BUFFER_SIZE;
}
```

Konsument

```
item nextConsumed;

while (true) {
     while (in == out)
        ; // do nothing

     nextConsumed = buffer[out];
     out = (out + 1) % BUFFER_SIZE;
     /* consume the item in nextConsumed */
}
```

Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Shared memory

- W oparciu o przykład napisz program typu chat, w którym dwa procesy wymieniają między sobą komunikaty (liczby lub tekst).

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the identifier for the shared memory segment */
int segment_id;
/* a pointer to the shared memory segment */
char *shared_memory;
/* the size (in bytes) of the shared memory segment */
const int size = 4096;

    /* allocate a shared memory segment */
    segment_id = shmget(IPC_PRIVATE, size, S_IRUSR | S_IWUSR);

    /* attach the shared memory segment */
    shared_memory = (char *) shmat(segment_id, NULL, 0);

    /* write a message to the shared memory segment */
    sprintf(shared_memory, "Hi there!");

    /* now print out the string from shared memory */
    printf("*%s\n", shared_memory);

    /* now detach the shared memory segment */
    shmdt(shared_memory);

    /* now remove the shared memory segment */
    shmctl(segment_id, IPC_RMID, NULL);

    return 0;
}
```

Źródło: A. Silberschatz, *Operating Systems Concepts Essentials*

Synchronizacja

- Message passing może być blokujący lub nieblokujący:
  - Blokujące wysyłanie
  - Nieblokujące wysyłanie
  - Blokujący odbiór
  - Nieblokujący odbiór

Buforowanie

- Zerowa pojemność (zero capacity) - kolejka ma długość zero
- Ograniczona długość (bounded capacity) - kolejka ma ustaloną długość
- Nieograniczona długość (unbounded capacity) - kolejka ma nieograniczoną długość


Turbacz