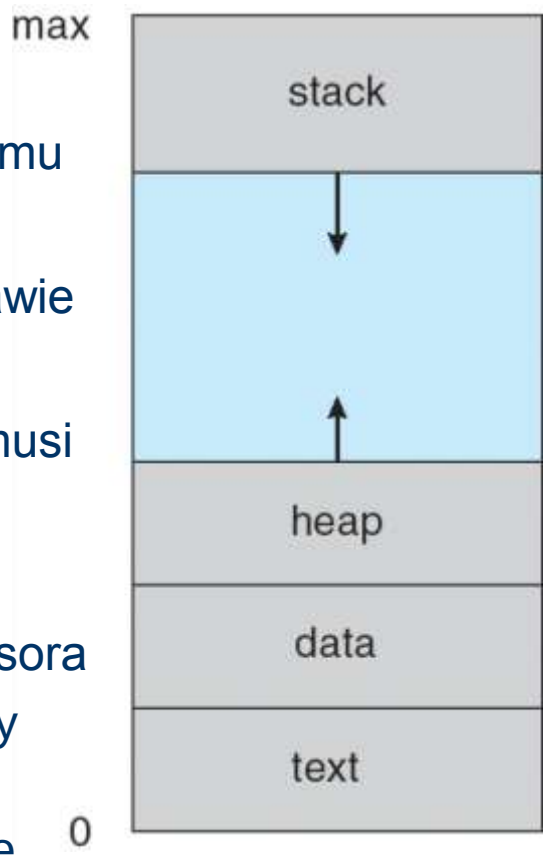
Procesy

1. **Koncepcja procesu**
2. Planowanie procesów
3. Działania na procesach
4. Procesy współpracujące. Komunikacja i synchronizacja międzyprocesowa

Data ostatniej modyfikacji: 09.10.2016

Koncepcja procesu

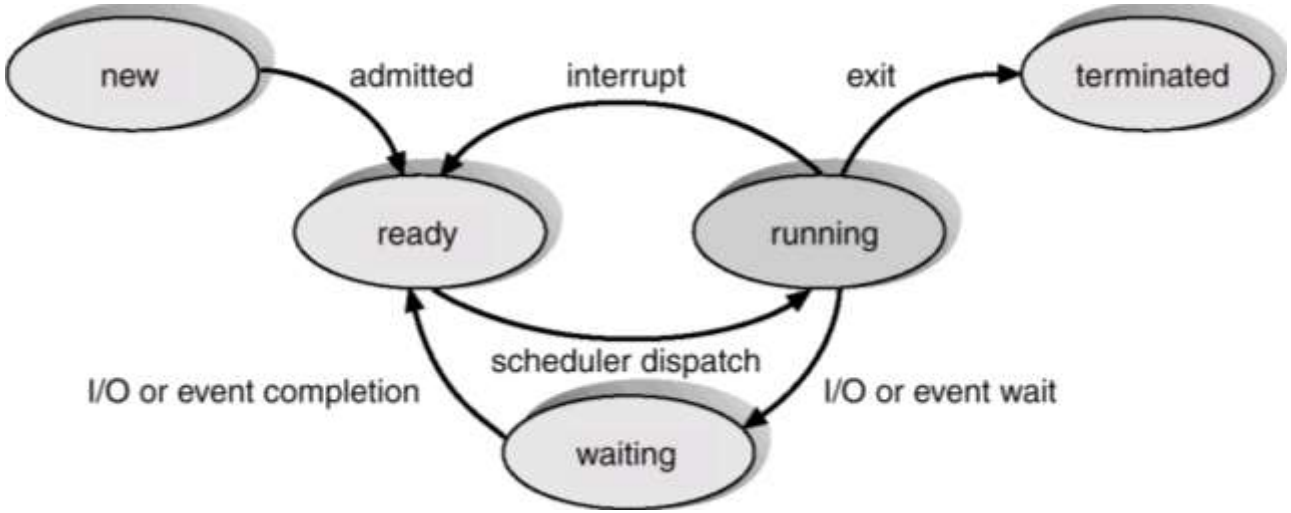
- System operacyjny wykonuje różnorodne prace:
  - system wsadowy - zadania (jobs)
  - system z podziałem czasu - programy użytkownika (user programs) lub zadania użytkownika czy systemu (tasks)
- W podręczniku terminy zadanie i proces używane są prawie zamiennie.
- Proces – wykonujący się program; wykonanie procesu musi być sekwencyjne (jeden wątek sterowania).
- Składowe procesu (nie wyłączne):
  - **Kod** (*text section*), licznik rozkazów i rejestry procesora
  - **Stos** (*stack*)– zawiera tymczasowe dane: parametry wywołania funkcji, zmienne lokalne (automat.)
  - **Sekcja danych** – zawiera zmienne glob. i statyczne
  - **Sterta** (*heap*) – zawiera dane przydzielane dynamicznie



Proces w pamięci operacyjnej

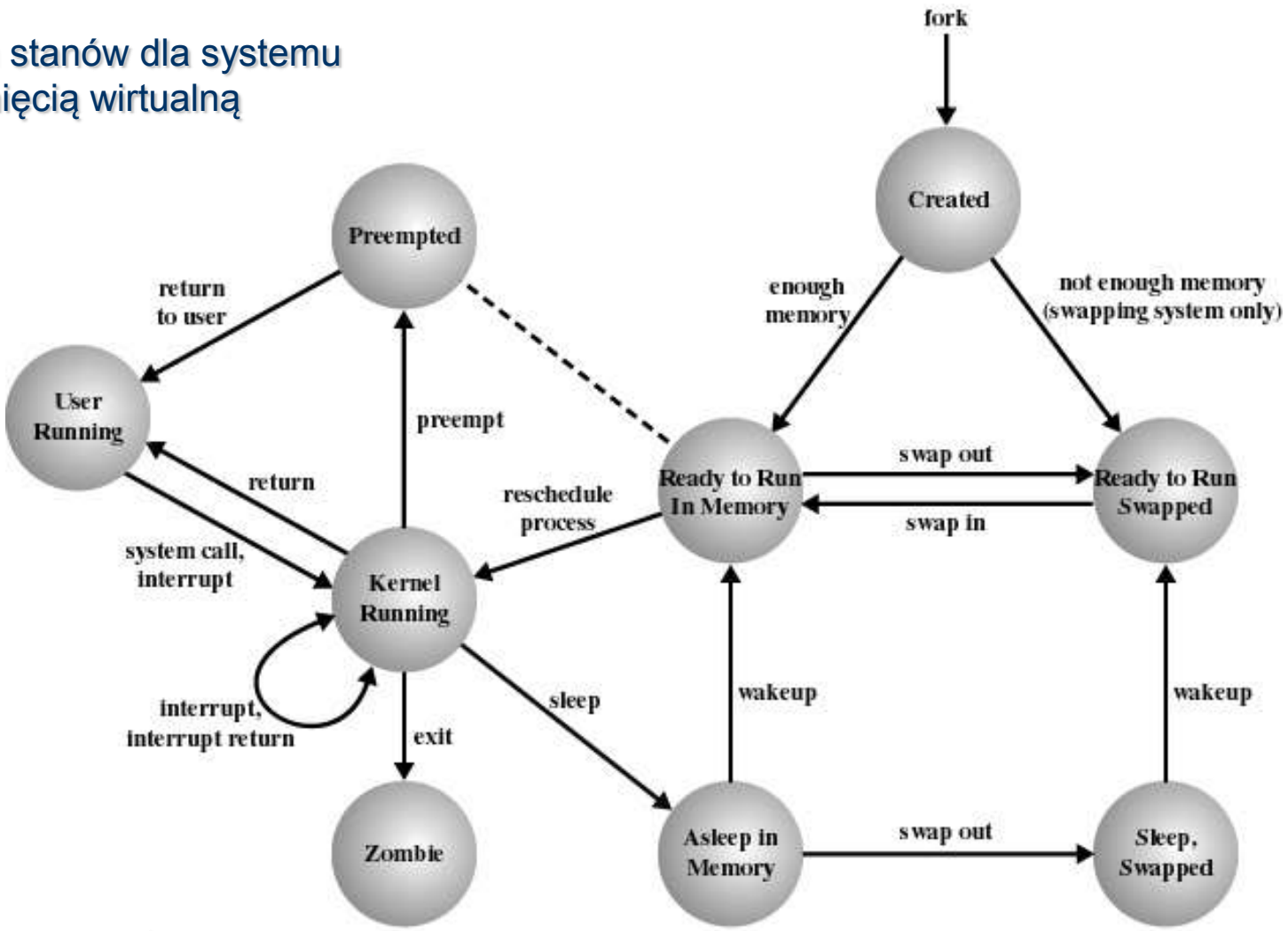
Stan procesu

- Wykonujący się proces zmienia swój **stan**
  - **nowy** (*new*): proces został utworzony
  - **aktywny** (*running*): są wykonywane instrukcje procesu
  - **oczekujący** (*waiting*): proces czeka na wystąpienie jakiegoś zdarzenia
  - **gotowy** (*ready*): proces czeka na przydział procesora
  - **zakończony** (*terminated*): proces zakończył działanie.
- Podstawowy (5-stanowy) diagram stanu procesów:



Stan procesu – c.d.

Diagram stanów dla systemu z pamięcią wirtualną

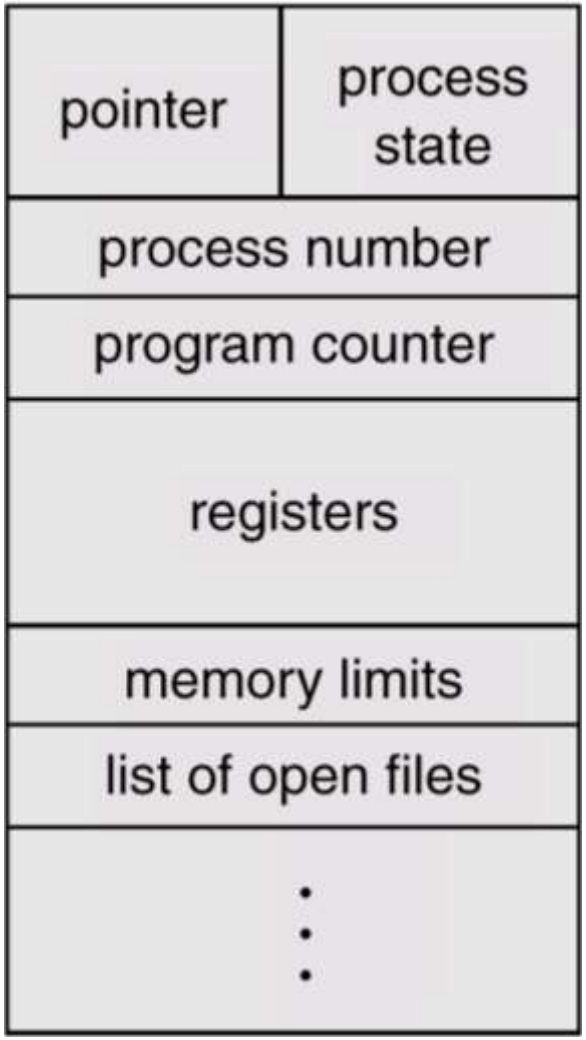


Autor rysunku: nieznany

Blok kontrolny procesu (PCB)

Informacje zawarte w **bloku kontrolnym procesu** (*process control block - PCB*):

- stan procesu
- licznik rozkazów
- rejestry procesora
- informacje o planowaniu przydziału procesora
- informacje o zarządzaniu pamięcią
- informacje do rozliczeń
- informacje o stanie wejścia/wyjścia

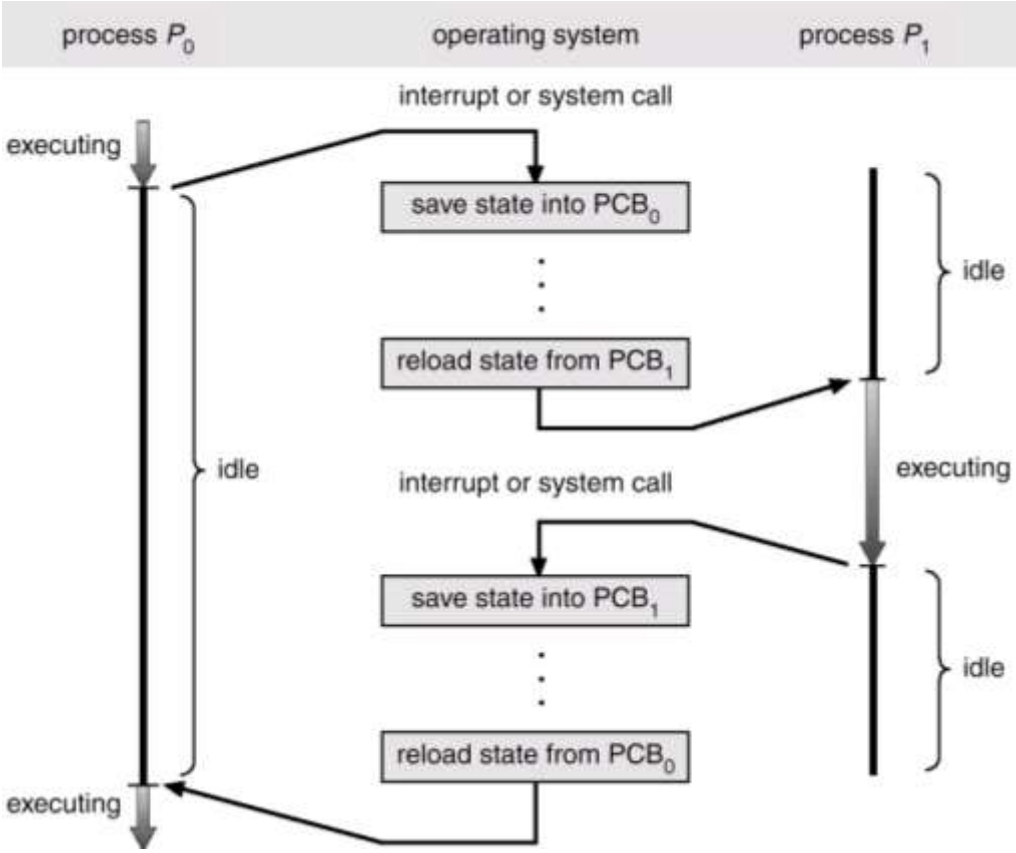


Przełączanie procesora pomiędzy procesami

Przyczyny przerwania wykonania procesu:

- przerwanie zegarowe
- przerwania od urządzeń
- wywołanie f. systemowej
- wystąpienie pułapki

W/w przerwania mogą być okazją do przełączenia procesora pomiędzy procesami, ale decyzja zależy od **planisty przydziału procesora**.



Przełączanie kontekstu

- Kiedy procesor ma zmienić obsługiwany proces (*przełączyć kontekst*) - system operacyjny musi przechować stan starego procesu i załadować przechowywany stan nowego procesu.

- Czas przełączania jest narzutem (*overhead*) na rzecz systemu, gdyż w czasie przełączania system nie wykonuje użytecznej pracy na rzecz procesów użytkownika.

- Czas przełączania kontekstu zależy od typu maszyny, szybkości pamięci, liczby rejestrów, istnienia specjalnych instrukcji zapamiętywania i odtwarzania stanu (typowo 1 do 1000μs).

Reprezentacja procesów w SO Linux

Struktura task_struct (<linux/include/linux/sched.h>) jest alokowana dynamicznie w przestrzeni adresowej jądra

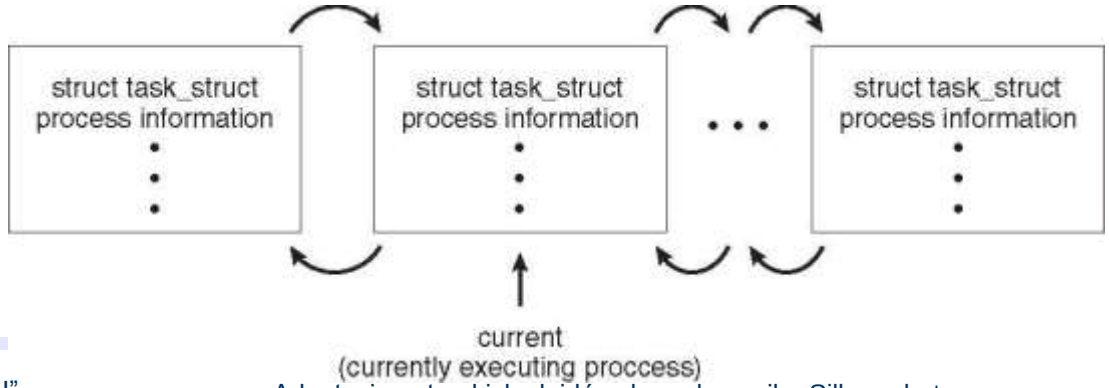
Wybrane pola:

```
volatile long state;/* -1 unrunnable, 0 runnable, >0 stopped */
void *stack;
pid_t  pid, tgid;
struct  task_struct__rcu *real_parent; /* real parent process */
struct  list_head children; /* list of my children */
struct  list_head  sibling; /* linkage in my parent's children list */
struct  task_struct *group_leader; /* threadgroup leader */
/* PID/PID hash table linkage. */
struct pid_link pids[PIDTYPE_MAX];
struct  list_head  thread_group, thread_node;

struct mm_struct *mm, *active_mm;
struct fs_struct *fs; /* filesystem information */
 struct files_struct *files; /* open file information */
/* signal handlers */
struct signal_struct *signal;
struct sighand_struct *sighand;
sigset_t blocked, real_blocked, saved_sigmask;
struct sigpending pending;
struct  list_head cpu_timers[3];
```

Stany procesu – zmienna **state** (<linux/include/linux/sched.h>)

```
#define TASK_RUNNING              0
#define TASK_INTERRUPTIBLE        1
#define TASK_UNINTERRUPTIBLE      2
#define  TASK_STOPPED             4
#define  TASK_TRACED              8
/* in tsk->exit_state */
#define EXIT_DEAD                 16
#define EXIT_ZOMBIE               32
#define EXIT_TRACE (EXIT_ZOMBIE|EXIT_DEAD)
/* in tsk->state again */
#define TASK_DEAD 64
#define TASK_WAITKILL             128
#define TASK_WAKING               256
#define  TASK_PARKED              512
#define TASK_STATE_MAX            1024
```



Procesy

1. Koncepcja procesu
2. **Planowanie procesów**
3. Działania na procesach
4. Procesy współpracujące. Komunikacja i synchronizacja międzyprocesowa

Kolejki planowania procesów

- **Planista** ma maksymalizować wykorzystanie CPU, efektywnie przełączając procesor pomiędzy procesami gotowymi
- Planista zadań obsługuje różne kolejki systemowe
  - **Kolejka zadań** (*job queue*) – zawiera wszystkie zadania w systemie.
  - **Kolejka zadań gotowych** (*ready queue*) – zawiera wszystkie procesy gotowe do działania (w pamięci operacyjnej)
  - **Kolejki do urządzeń** (*device queues*) – listy procesów oczekujących na obsługę przez konkretne urządzenia

Procesy przemieszczają się pomiędzy kolejkami.

Kolejki procesów gotowych i do urządzeń

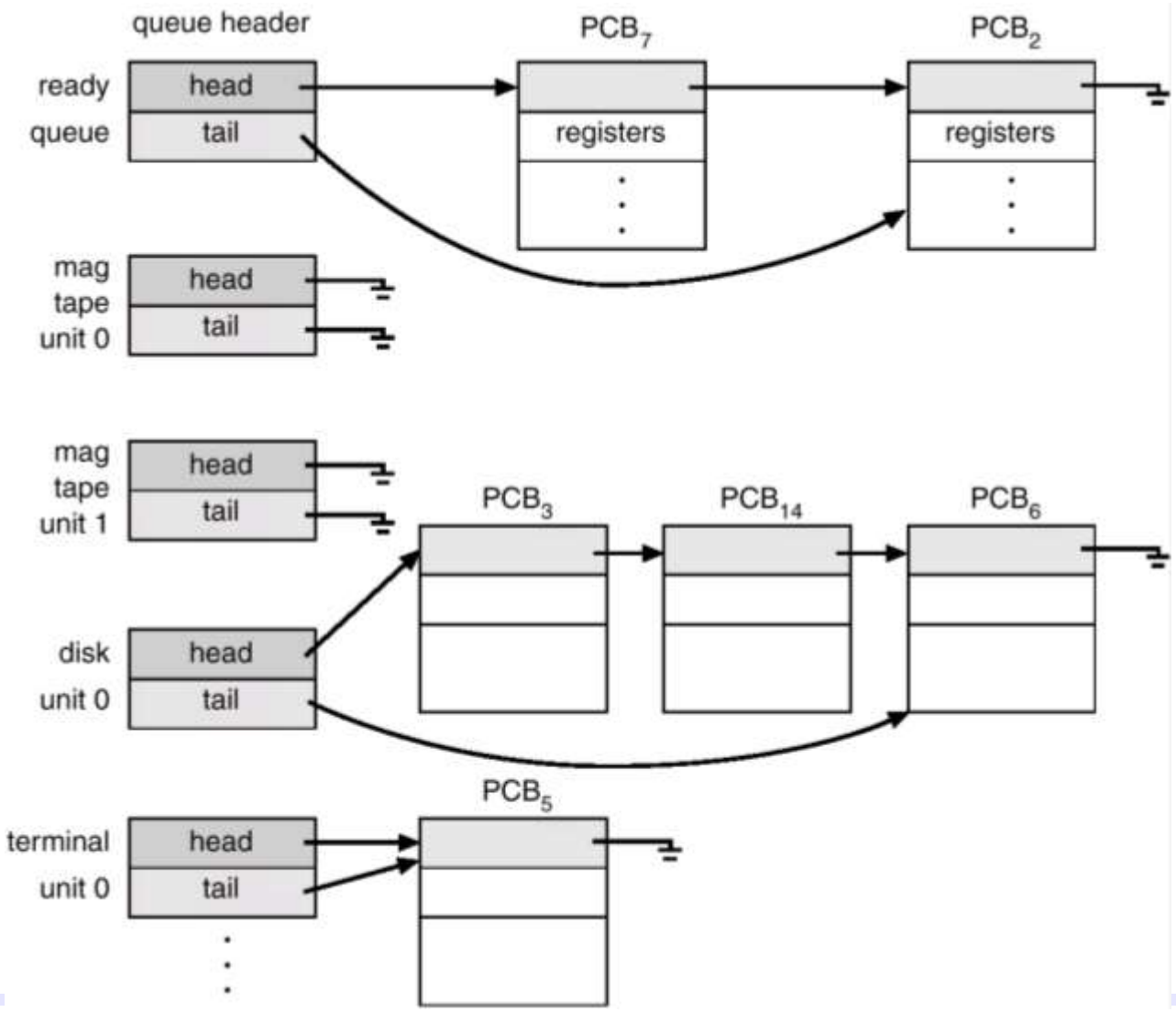
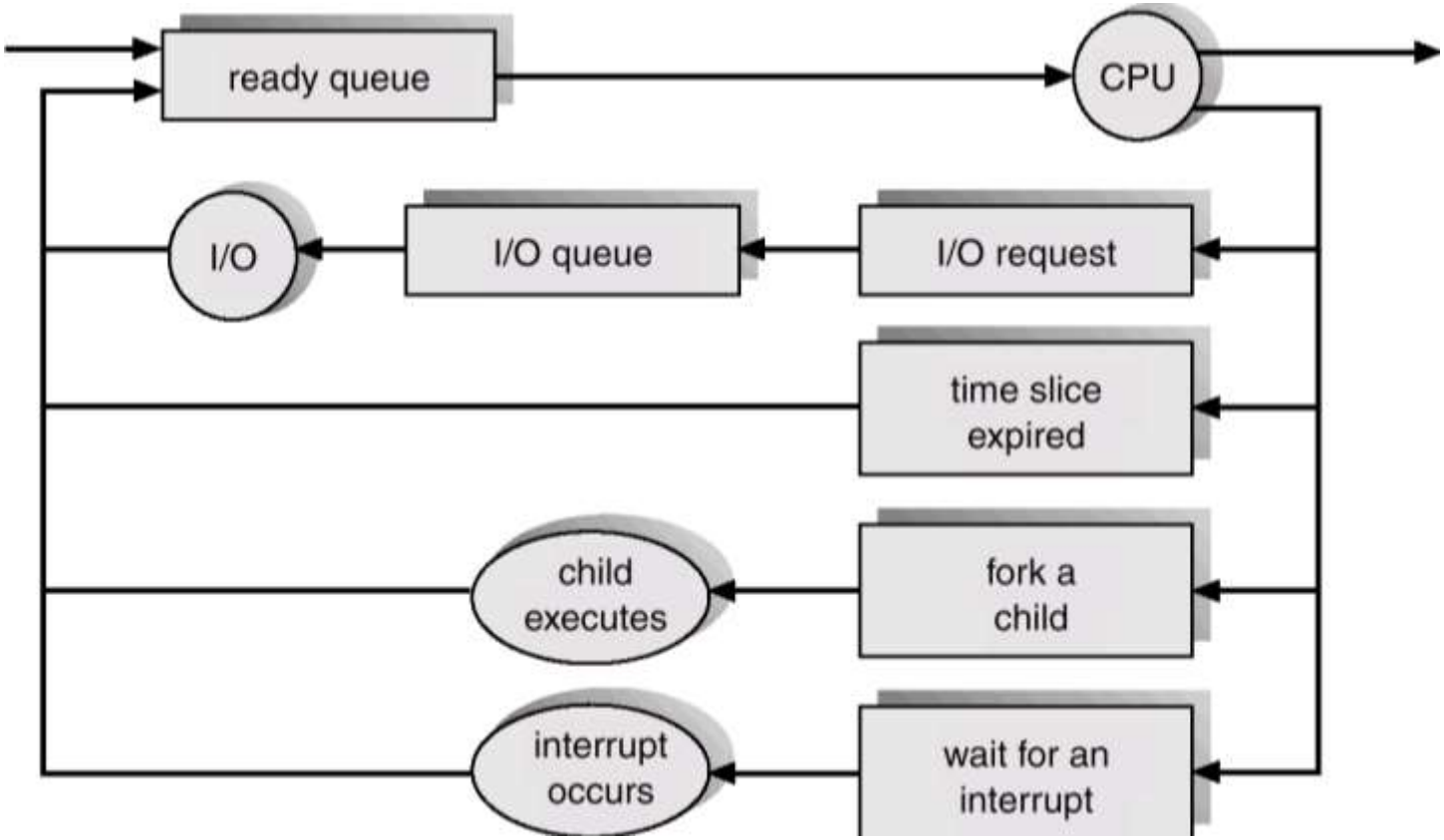


Diagram kolejek w planowaniu procesów

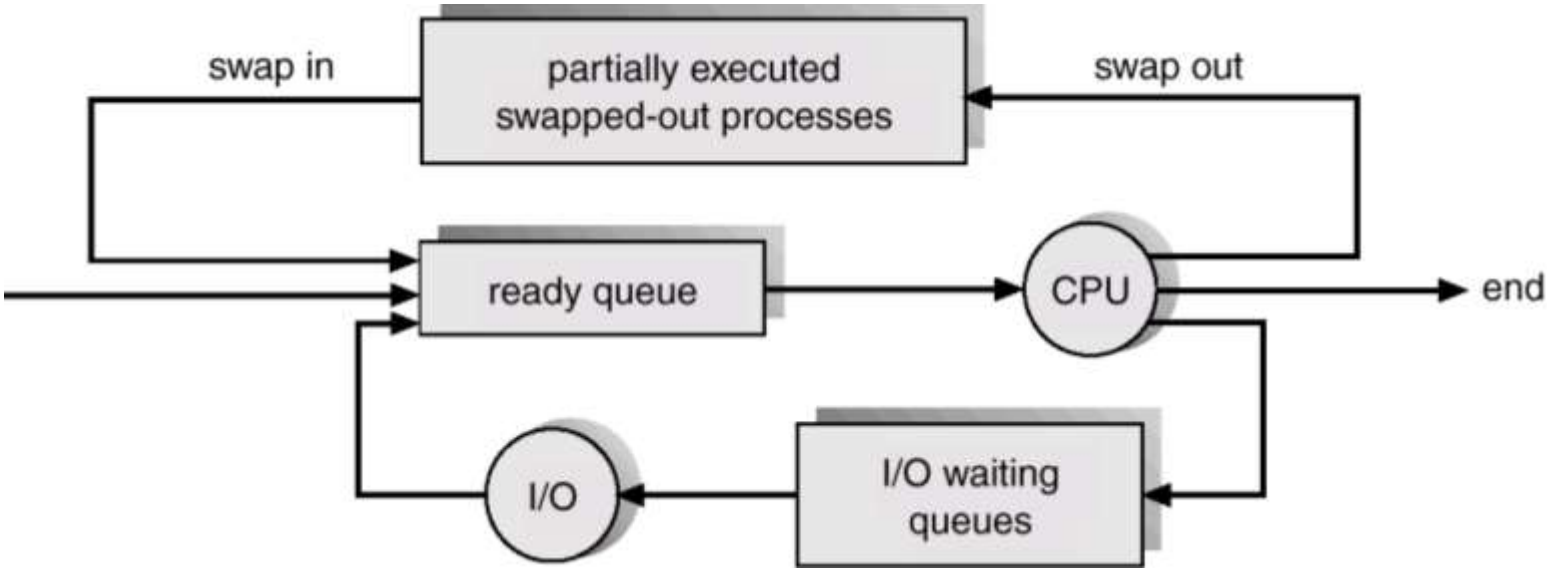


Planiści

- Procesy można dzielić na:
  - **ograniczone przez wejście wyjście** - spędzające dużo więcej czasu na wykonywanie operacji wejścia/wyjścia niż na obliczenia (wiele krótkich faz procesora).
  - **ograniczone przez dostęp do procesora** - sporadycznie generujące zamówienia na operacje wejścia/wyjścia (nieliczne ale długie fazy procesora).
- **Planista krótkoterminowy** (planista przydziału procesora - *short-term scheduler, CPU scheduler*)
  - wybiera do wykonania jeden z procesów gotowych i przydziela mu procesor.
  - podejmuje działanie bardzo często (n.p. co 100ms) $\Rightarrow$ musi działać szybko.
- **Planista długoterminowy** (planista zadań - *long-term scheduler, job scheduler*)
  - wybiera procesy do kolejki procesów gotowych
  - wywoływany dosyć rzadko (co sekundy, minuty) $\Rightarrow$ może działać powoli
  - kontroluje **stopień wieloprogramowości**
  - usiłuje realizować dobrą mieszankę procesów
  - Nie wszystkie systemy operacyjne mają planistę długoterminowego

Uzupełnienie o planowanie średnioterminowe

System operacyjny może wykorzystywać **planistę średniookresowego**, który zabezpiecza system przed zbyt dużym **stopniem wieloprogramowości**. Planista ten (w razie potrzeby) przesuwa proces z pamięci na dysk, a w dogodnym czasie przywraca go z powrotem do pamięci, by umożliwić kontynuację jego wykonania (***swapping in/out***)



Procesy

1. Koncepcja procesu
2. Planowanie procesów
3. **Działania na procesach**
4. Procesy współpracujące. Komunikacja i synchronizacja międzyprocesowa

Operacje na procesach

System operacyjny musi obsługiwać:

- Tworzenie procesów
- Kończenie procesów
- Komunikację i synchronizację procesów

Tworzenie procesu

- Procesy macierzyste (*parent processes)* tworzą **procesy potomne** (*children*), które też tworzą podprocesy. W rezultacie powstaje **drzewo procesów**.

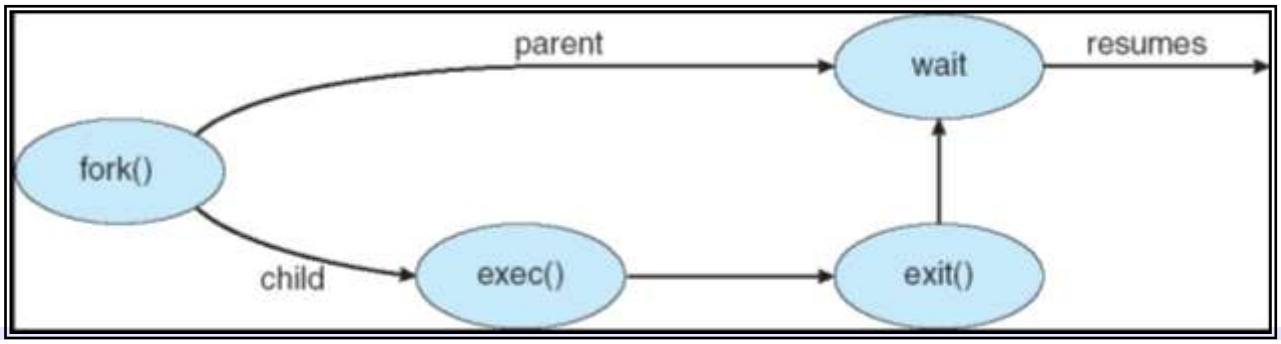
- Dzielenie zasobów – warianty:
  - Proces macierzysty i podprocesy współdzielą zasoby.
  - Podprocesy współdzielą część zasobów procesu macierzystego.
  - Proces macierzysty i podprocesy nie dzielą zasobów.

- Wykonywanie procesów – warianty:
  - Proces macierzysty i podprocesy są wykonywane współbieżnie.
  - Proces macierzysty czeka na zakończenie niektórych lub wszystkich swoich procesów potomnych.

Tworzenie procesu (c.d.)

- Przestrzeń adresowa nowego procesu może być zagospodarowana na 2 sposoby:
  - Proces potomny staje się **kopią procesu** macierzystego.
  - Proces potomny otrzymuje **nowe zasoby** (nowy program).

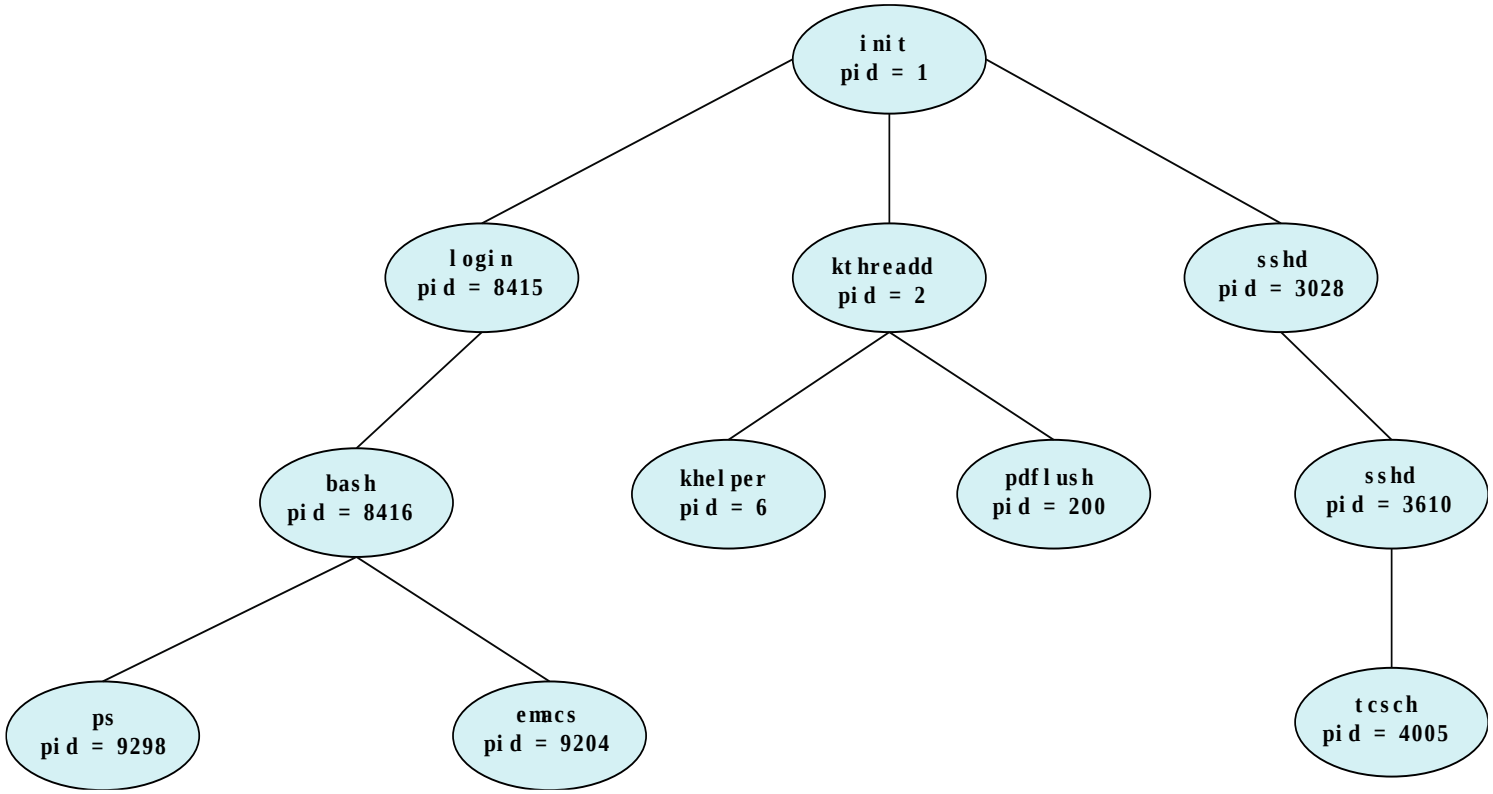
- W systemie UNIX
  - funkcja systemowa `fork()` tworzy nowy proces
  - funkcja systemowa `execve()` (użyta typowo po wywołaniu funkcji `fork()`) zastępuje zawartość przestrzeni adresowej procesu przez nowy program.



Tworzenie procesu – interfejs POSIX

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(){
pid_t  pid1, pid2;
int status;
   pid1 = fork(); /* fork a child process*/
   if (pid1 < 0) { /* error occurred */
          perror("Fork failed");        exit(1);
   } else if (pid1 == 0) { /* child process */
          execl("/bin/ls", "ls", " -l",NULL); /* overwriting the child proces
                                                 memory with new program */
          perror(”execlp”);  exit(2);/* executed only if execl fails */
   } else { /* parent process */
          if((pid2=wait (&status))>0){/* waiting for the child proces to complete
   */
                fprintf (stderr,„Potomek: PID=%d, status=%d,(int)pid2,status);
          } else {
                   perror(”wait”);  exit(3); /* wait() error */
          }
          return(0); /* alternaively: exit(0) */
   }
}
```

Drzewo procesów systemu Linux



Uwaga: we współczesnych systemach zamiast procesu **init** z pid=1 może występować proces **systemd**

Tworzenie procesu – interfejs Win32

```
#include <windows.h>
#include <stdio.h>
int main( VOID ){
    STARTUPINFO si;
    PROCESS_INFORMATION pi;
    ZeroMemory( &si, sizeof(si) ); si.cb = sizeof(si);ZeroMemory( &pi, sizeof(pi) );
//  Start the child process.
    if( !CreateProcess( NULL,   // No module name (use command line).
        "C:\\WINDOWS\\system32\\notepad.exe", // Command line.
        NULL,             // Process handle not inheritable.
        NULL,             // Thread handle not inheritable.
        FALSE,            // Set handle inheritance to FALSE.
        0,                // No creation flags.
        NULL,             // Use parent's environment block.
        NULL,             // Use parent's starting directory.
        &si,              // Pointer to STARTUPINFO structure.
        &pi )             // Pointer to PROCESS_INFORMATION structure.
    ) {        fprintf(stderr, "CreateProcess failed (%d).\n", GetLastError() ); return -1;
   }
    WaitForSingleObject( pi.hProcess, INFINITE ); // Wait until child process exits.
    CloseHandle( pi.hProcess ); CloseHandle( pi.hThread ); //Close process and thread handles
}
```

Wykonanie polecenia systemowego w podprocesie – standardowa biblioteka C

```
#include <stdlib.h>
#include <stdio.h>
int main( void){
    char buf[512];
    if(fgets(buf, sizeof(buf), stdin)){/* wczytanie
  polecenia do bufora */
        int ret;
/* wykonanie polecenia w podprocesie interpretera
  poleceń (powłoki) */
        ret=system(buf);
        if(ret) fprintf(stderr,"ret=%d\n",ret);
        return ret;
    }
   return 0;
}
```

Kończenie procesu

- Proces za pomocą wywołania funkcji systemowej (POSIX/C: `exit()`) prosi system operacyjny, aby go usunął.
  - Kod wyjścia podprocesu jest przekazywany do procesu macierzystego (POSIX: za pomocą funkcji `wait()`). POSIX:
    - zakończony proces potomny dla którego rodzic nie wykonał `wait()`, `to` **zombie**
    - Potomek, którego rodzic się zakończył, staje się **sierotą** (adoptowaną przez proces **init**).
  - Wszystkie zasoby procesu są odebrane przez system operacyjny.

- Proces macierzysty może spowodować zakończenie innego procesu (zazwyczaj potomka) za pomocą funkcji systemowej (np. w Win32: `TerminateProcess()`) z kilku powodów, n.p.
  - proces potomny nadużył przydzielonego mu zasobu
  - wykonywane przez potomka zadanie jest zbędne

  W niektórych systemach, gdy proces macierzysty kończy się - system nie pozwala potomkowi kontynuować działania (***kaskada zakończeń****:* potomkowie, potomkowie potomków itd.) .

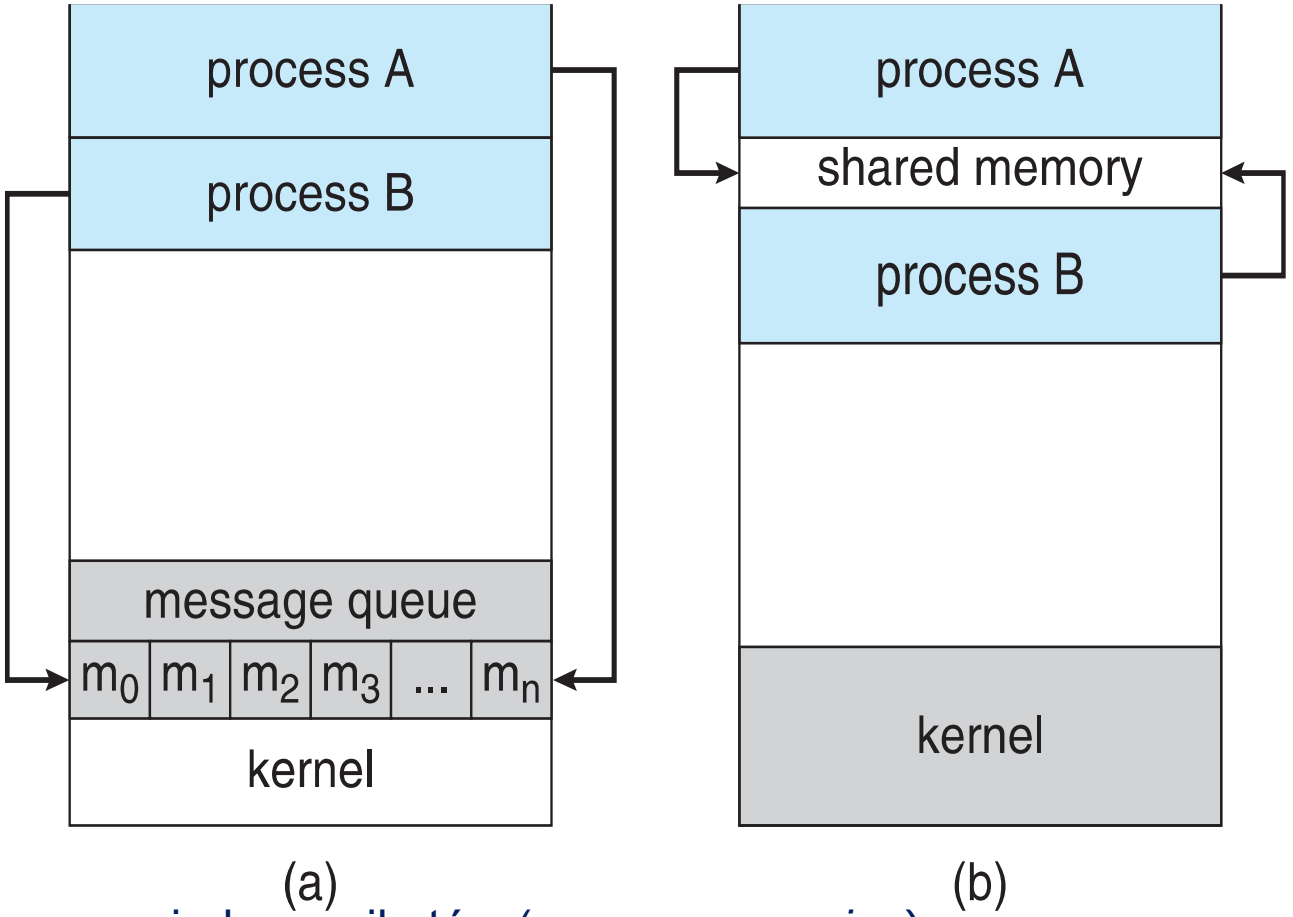
Procesy

1. Koncepcja procesu
2. Planowanie procesów
3. Działania na procesach
4. **Procesy współpracujące. Komunikacja i synchronizacja międzyprocesowa**

Procesy współpracujące

- **Proces niezależny** nie może oddziaływać na inne procesy, a te z kolei nie mogą oddziaływać na jego działanie.
- **Procesy współpracujące** oddziałują wzajemnie na swoje wykonanie (dzielą dane). System udostępnia takim procesom:
  - możliwość współbieżnego wykonania
  - usługi komunikacji (międzyprocesowej)
  - usługi synchronizacji
- **Zalety współpracy procesów**
  - dzielenie informacji (np. pomiędzy różnymi użytkownikami)
  - przyspieszanie obliczeń (dla komputera o wielu procesorach czy kanałach wejścia/wyjścia)
  - modularność
  - wygoda (z tytułu równoległego wykonywania kilku czynności)

Modele komunikacji międzyprocesowej



a) Przekazywanie komunikatów (*message passing*).

b) Pamięć współdzielona (*shared memory*). Poprawne współbieżne wykorzystywanie operacji R/W wymaga synchronizacji (**sekcja krytyczna**)