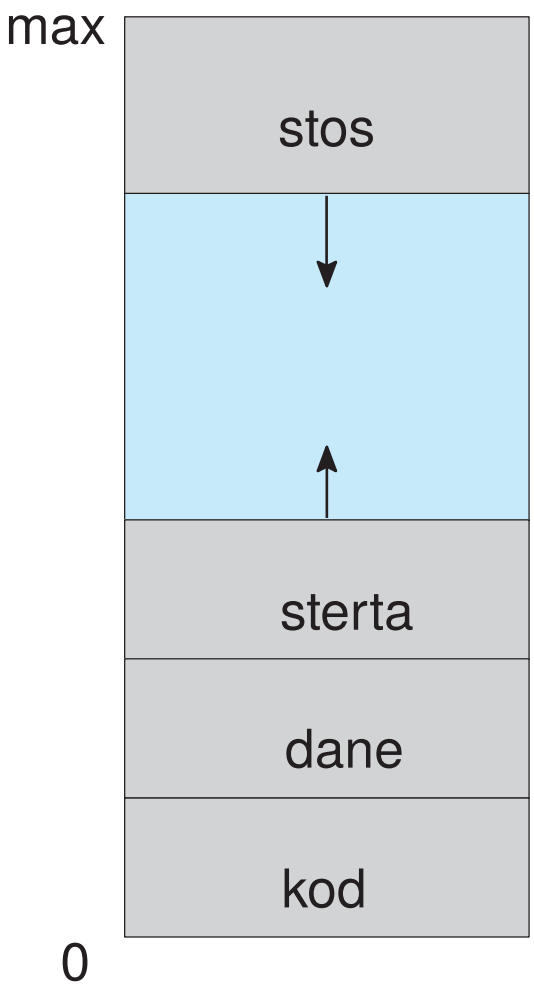
Wykład 3

Procesy i wątki

Pojęcie procesu

- Program = plik wykonywalny na dysku
- Proces = uruchomiony i wykonywany program w pamięci
  - Z jednego programu można utworzyć wiele procesów
- Program jest pojęciem statycznym.
- Proces ma naturę dynamiczną (zmieniającą się). Zmianie ulegają m.in.
  - Licznik rozkazów (adres ostatnio wykonywanej instrukcji)
  - Rejestry procesora
  - Wskaźnik stosu
  - Pamięć przydzielona procesowi
- Proces ma przestrzeń adresową (przydzielony obszar pamięci operacyjnej, składającą się z wielu części).

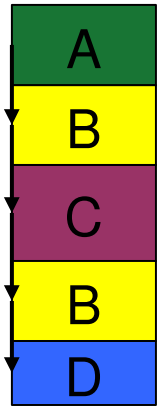
Proces w pamięci



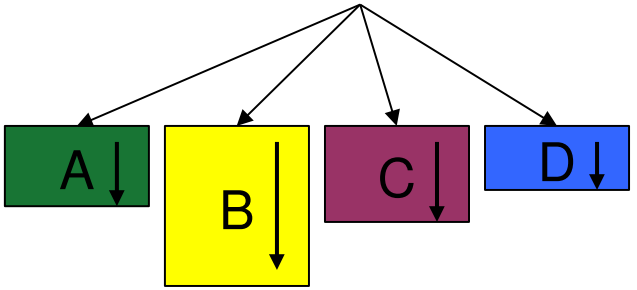
- Stos procesora (ang. CPU stack) przechowuje dane tymczasowe (parametry funkcji, adresy powrotu, zmienne lokalne – zakładając język C/C++).
- Sterta (ang. heap) – pamięć dynamicznie zaalokowana przez program.
  - Uwaga malloc/new są po stronie użytkownika !
- Dane - zmienne globalne i statyczne
- Kod – instrukcje maszynowe procesora.
- Wywołanie systemowe „wczytaj program do pamięci” inicjalizuje kod i dane
- W rzeczywistych systemach – obraz bardziej skomplikowany z powodu obecności bibliotek współdzielonych (Linuks) / dynamicznie linkowanych (Windows)

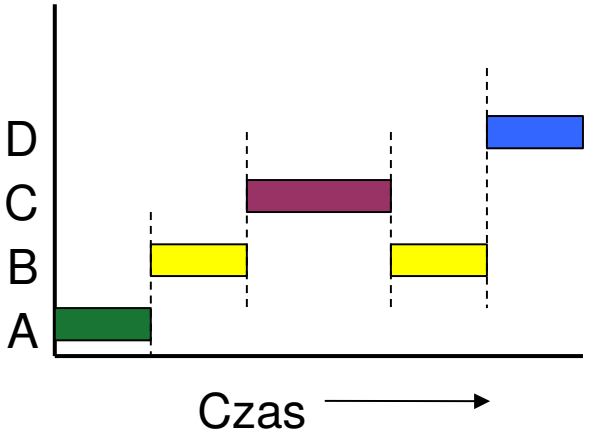
Model procesów

Pojedynczy licznik rozkazów (punkt widzenia procesora)



Wiele liczników rozkazów (punkt widzenia systemu oper.)

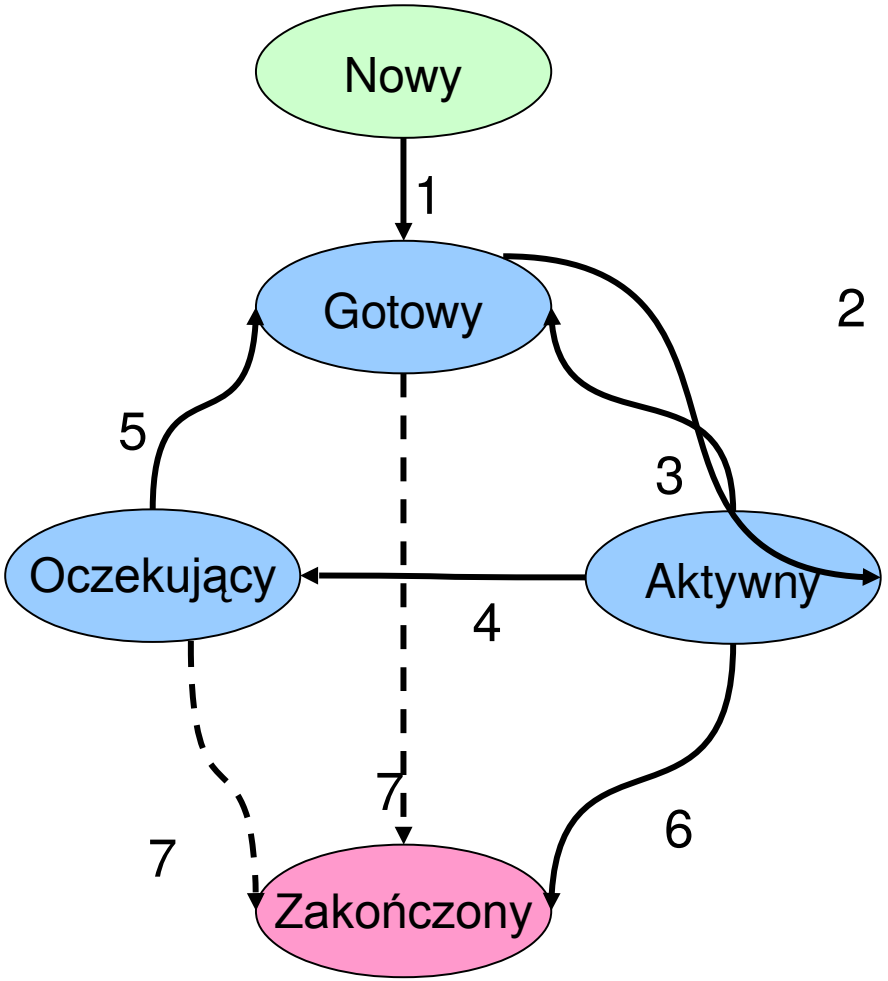




Z punktu widzenie procesora na komputerze jest wykonywany jeden program

Z punktu widzenia systemu jednocześnie jest wykonywanych wiele programów.

Stany procesu



Nowy – proces został utworzony.

Gotowy – proces czeka na przydział procesora.

Proces mógłby się wykonywać, ale nie wykonuje się, ponieważ w tej chwili wykonuje się jakiś inny proces.

Aktywny – wykonywane są instrukcje procesu.

W systemie z jednym procesorem w danej chwili jeden proces może być aktywny

Oczekujący (uśpiony) – proces czeka na zdarzenie (np. Zakończenie operacji we-wy).

Na poprzednim wykładzie proces, który zainicjalizował transmisję DMA lub wysłała znak do drukarki, był usypiany w oczekiwaniu na przerwanie.

Proces w stanie uśpionym nie otrzyma procesora.

Zakończony – proces zakończył działanie

Przejścia pomiędzy stanami procesu

1 (Nowy => Gotowy). Nowo utworzony proces przechodzi do kolejki procesów gotowych.

Planista długoterminowy (ang. long-term scheduler) w systemach wsadowych.

2 (Gotowy => Aktywny) Proces otrzymuje przydział procesora.

3 (Aktywny => Gotowy) Procesowi został odebrany procesor (i przekazany innemu procesowi).

Przejściami 2 oraz 3 zarządza planista krótkoterminowy (ang. short-term).

4 (Aktywny => Oczekujący) Proces przechodzi w stan oczekiwania na zajście zdarzenia. (np. na zakończenie transmisji wejścia -wyjścia – patrz poprzedni wykład)

5 (Oczekujący => Gotowy) Zdarzenie na które czekał proces nastąpiło (przerwanie we/wy na poprzednim wykładzie).

Przejścia 4 oraz 5 są wykonywane przy przeprowadzeniu synchronicznej operacji wejścia wyjścia (ale nie tylko).

6 (Aktywny => Zakończony). Proces zakończył pracę (np. funkcja **`exit`** w Uniksach, błąd ochrony)

7 (Gotowy =>Zakończony oraz Oczekujący => Zakończony). Proces został zakończony przez inny proces (np. funkcja **`kill`** w systemie Unix).

Dodatkowy stan – zawieszony (ang. suspended)

Proces oczekuje bardzo długo na operacje we-wy (np. polecenie login)

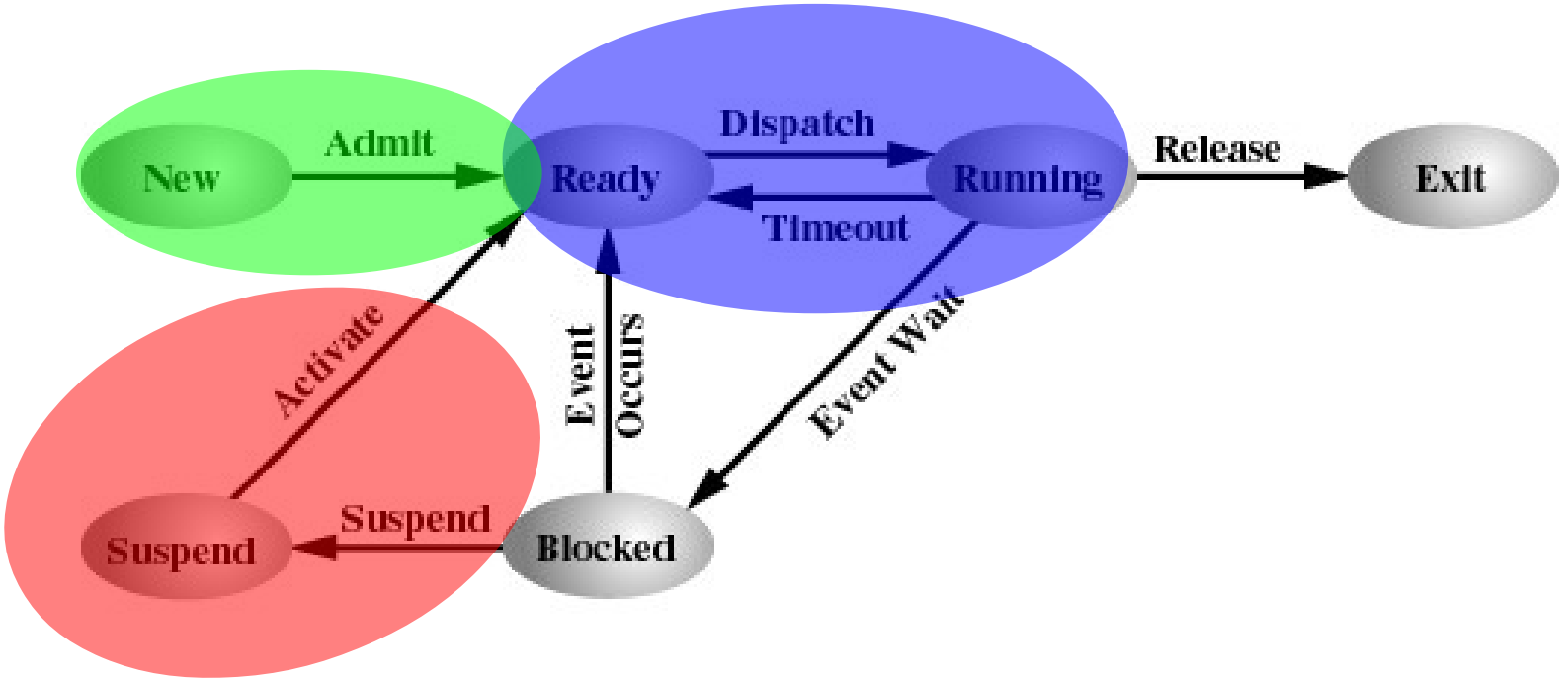
- Przejście do stanu zawieszonego
- Pamięć zajmowana przez proces podlega wymianie (ang. swapping) tzn. zapisaniu na dysk do obszaru wymiany (swap area).
- Zwolniona pamięć może być wykorzystana przez inne procesy.
- Po zajściu zdarzenia proces ponownie wczytywany z obszaru wymiany

Inne przyczyny zawieszenie procesu.

- Żądanie użytkownika
- Brak pamięci w systemie
- Proces co jakiś czas cyklicznie wykonuje jakąś czynność np. Sprawozdawczość

Przejściami do i z stanu zawieszenie zarządza planista średnioterminowy (ang. medium-term scheduler)

Diagram przejść z uwzględnieniem stanu zawieszenia.



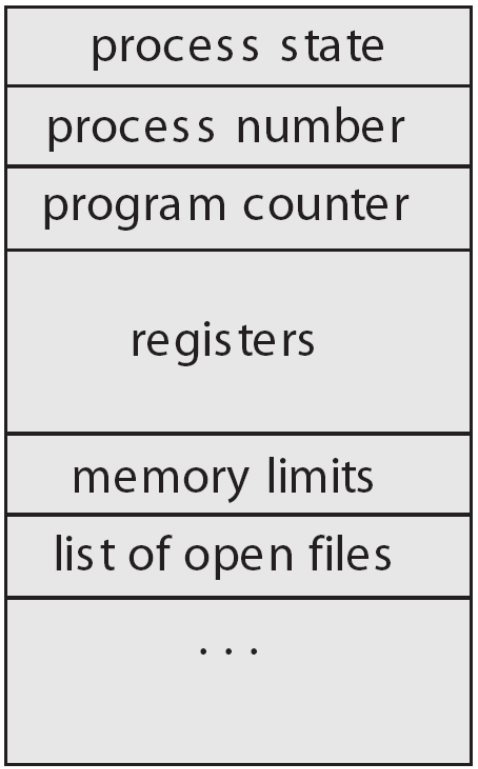
Planista długoterminowy

Planista krótkoterminowy – najważniejszy i występujący w każdym systemie.

Rozstrzyga problem: “Któremu procesowi w stanie gotowym mam przydzielić procesor”

Planista średnioterminowy

Blok kontrolny procesu
ang. Process Control Block - PCB



PCB służy do przechowywania informacji o procesie istotnych z punktu widzenia systemu operacyjnego

Stan procesu

Identyfikator procesu

Licznik rozkazów

Rejestry procesora

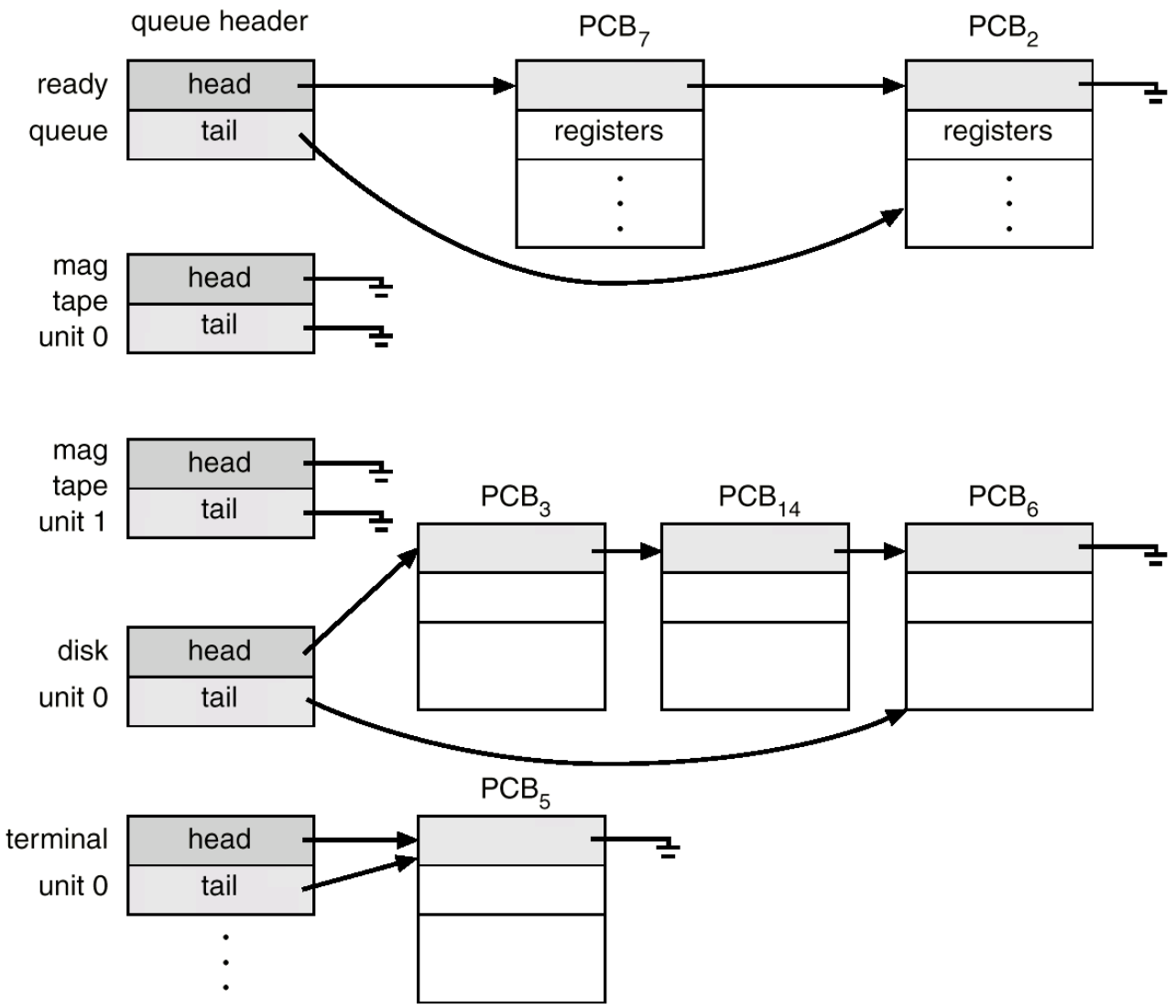
Informacja o przydzielonej pamięci

Informacja o otwartych plikach

Informacja o połączeniach sieciowych

Informacja niezbędna do tworzenia systemowych struktur danych. System operacyjny posługuje się różnymi kolejkami procesów. Jeżeli kolejki są implementowane jako listy z dowiązaniami, PCB może zawierać dowiązanie (wskaźnik) do następnego elementu w kolejce

Wykorzystanie kolejek w systemie operacyjnym



Ready queue – kolejka procesów w stanie gotowym.

Proces "wędruje" pomiędzy kolejkami

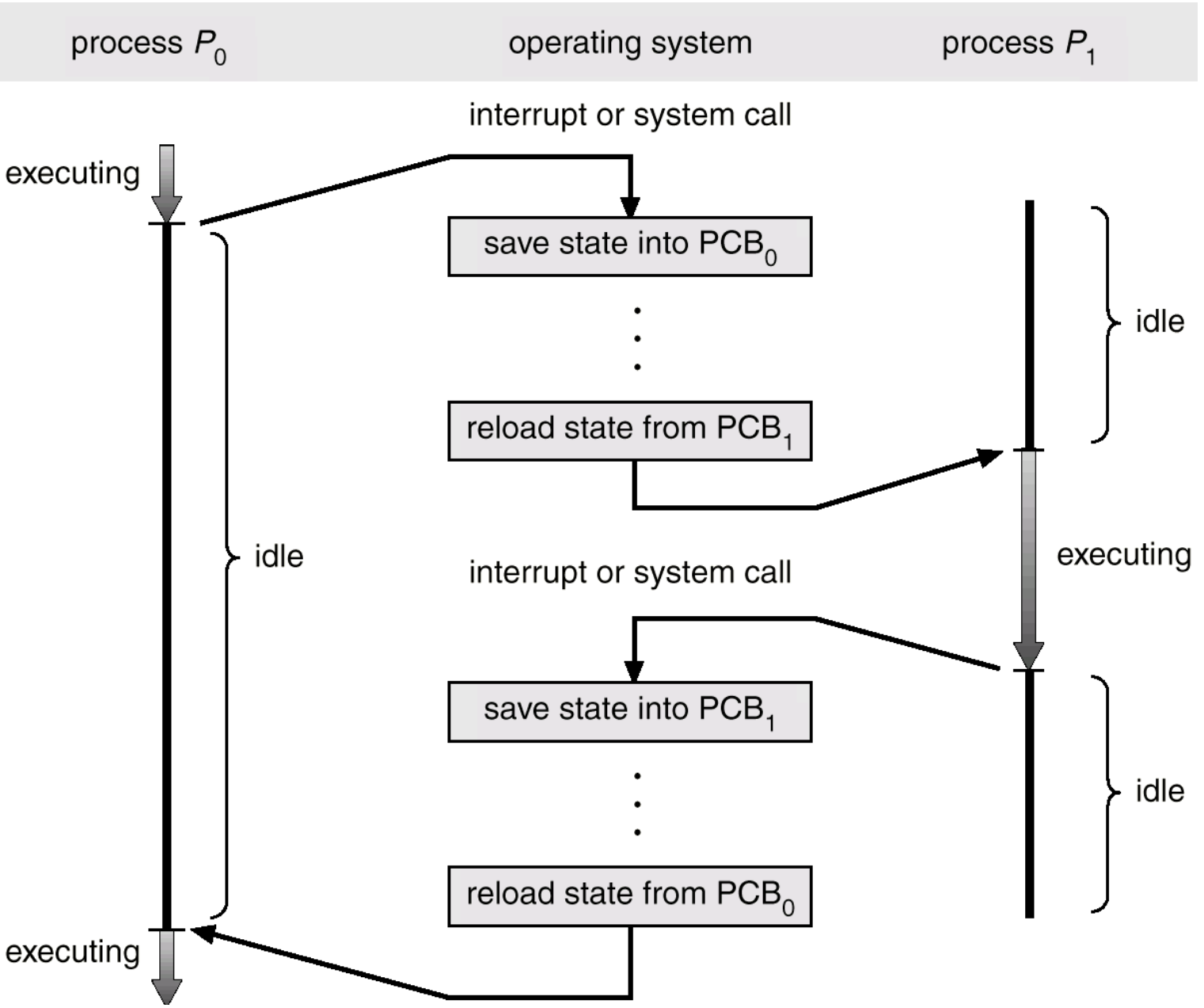
Kolejki procesów w stanie zablokowanym (oczekujących na wystąpienie zdarzenia).

Z każdym typem zdarzenia związana jedna kolejka.

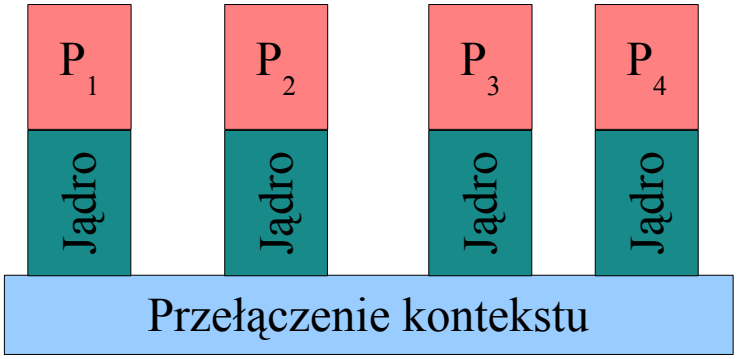
Dzięki temu system wydajnie implementuje przejścia 4 oraz 5

W rzeczywistym systemie takich mogą być setki kolejek dla procesów w stanie oczekującym.

Przełączenie kontekstu pomiędzy procesami



Przełączenie kontekstu (ang. context switch), a przełączenie trybu (ang. mode switch)



Przełączenie kontekstu to *zmiana procesu.*

Przełączenie trybu zmiana trybu pracy procesora (jądra <=> użytkownika)

W większości systemów (Uniksy,Windows) przyjęto model, w którym funkcje systemu wykonują się w kontekście procesu użytkownika. W uproszczeniu model ten zakłada że system operacyjny jest kolekcją procedur wywoływanych przez procesy w celu wykonania pewnych usług.

Stąd (tydzień temu) mówiłem o procesie wykonującym się w trybie jądra (jeżeli wykonywany jest kod jądra).

Przejście od programu użytkownika do programu jądra w wyniku wywołania funkcji systemowej (przerwanie programowe) lub przerwania sprzętowego wiąże się z przełączeniem trybu.

Zmiana trybu jest znacznie mniej kosztowna niż zmiana kontekstu.

Utworzenie procesu

Proces rodzicielski tworzy proces potomny, który z kolei może stworzyć kolejne procesy. Powstaje drzewo procesów.

Współdzielenie zasobów. Procesy rodzicielski i potomny mogą

- Współdzielić część zasobów
- Współdzielić wszystkie zasoby
- Nie współdzielić żadnych zasobów

Wykonywanie

- Procesy rodzicielski i potomny wykonują się współbieżnie
- Proces rodzicielski oczekuje na zakończenie procesu potomnego.

Przestrzeń adresowa

- Odrębna przestrzeń adresowa dla procesu potomnego
  - (***fork*** w systemie Unix – proces potomny wykonuje się w nowej przestrzeni adresowej będącej kopią przestrzeni procesu rodzicielskiego)
  - Proces ma nowy program załadowany do nowej przestrzeni adresowej (***CreateProcess*** w Win32)
- Proces potomny i rodzicielski wykonują się w tej samej przestrzeni adresowej (***clone*** w Linuksie; wątki Java i POSIX)

Wywołania systemowe ***fork***, ***exec***, ***wait*** w Unix-ie

```
#include <stdio.h>
#include <unistd.h>
int main(int argc, char *argv[])
{
  int pid;
    pid = fork();
    if (pid < 0) { /* Błąd !!! */
            fprintf(stderr, "Fork Failed");
            exit(-1);
    }
    else if (pid == 0) { /* proces potomny */
            execlp("/bin/ls","ls",NULL);
    }
    else { /* proces rodzicielski */
            wait(NULL);
            printf("Child Complete");
            exit(0);
    }
}
```

fork tworzy nowy proces

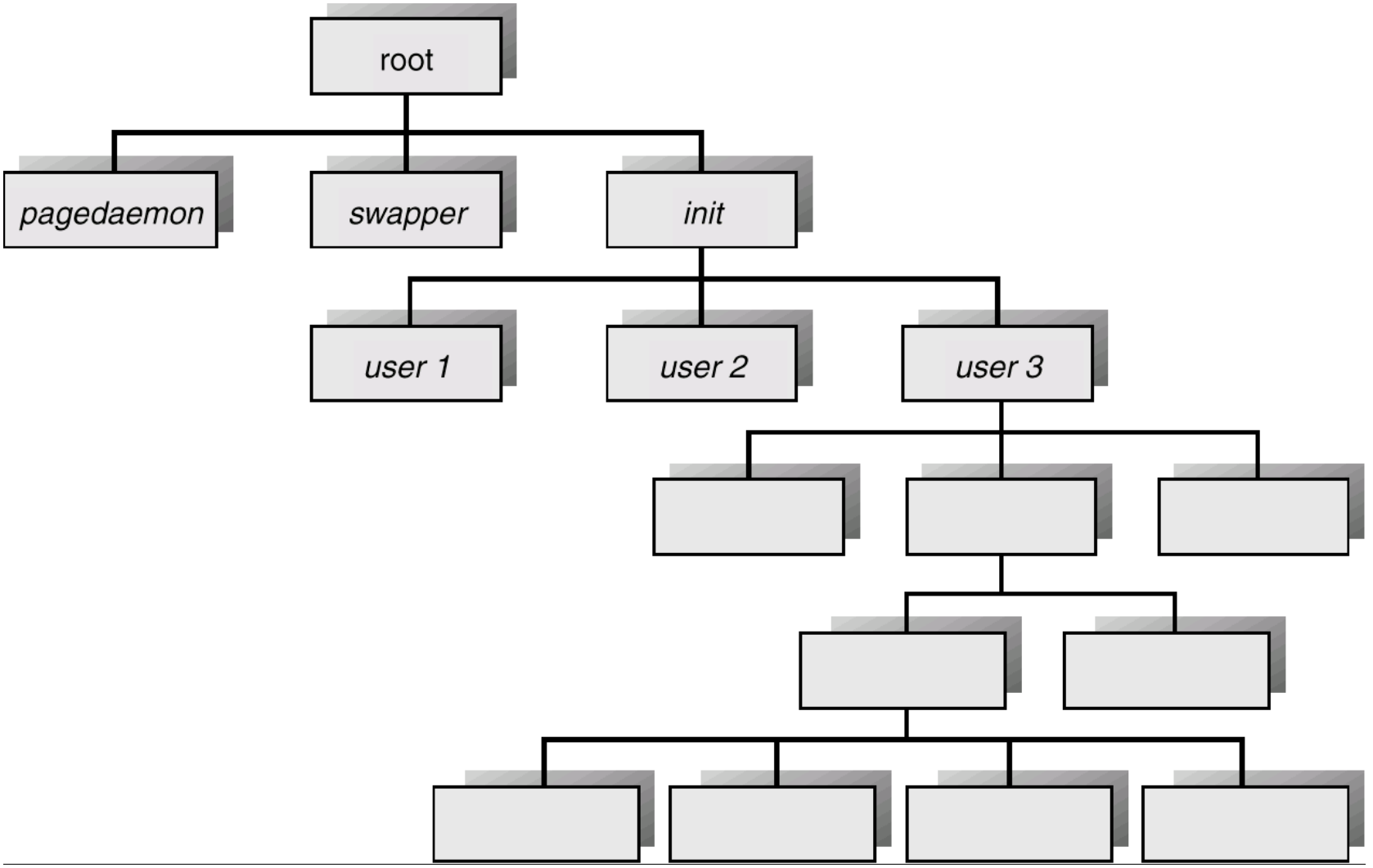
- Wywoływana z procesu rodzicielskiego
- Powracają z niej proces potomny i rodzicielski

exec – zastępuje obraz bieżącego procesu nowym programem.

- proces wołający nigdy nie wraca z exec (chyba że powstanie błąd)

wait – blokuje proces do momentu zakończenia procesu potomnego

Drzewo procesów w systemie Unix



Polecenie pstree

Zakończenie procesu

Zakończenie na własne żądanie. Proces sam podejmuje decyzję o zakończeniu pracy – wywołując odpowiednie wywołanie systemowe. (system Unix: ***exit***).

W programie w języku C jest to robione automatycznie po zakończeniu funkcji main

Proces został zakończony w wyniku akcji innego procesu

Unix: proces otrzymał sygnał SIGKILL.

Polecenie kill w shellu, funkcje systemowe ***raise*** oraz ***kill***.

Proces został zakończony przez system operacyjny

Naruszenie mechanizmów ochrony.

Przekroczenie ograniczeń na przyznany czas procesora.

Proces rodzicielski się zakończył (w niektórych systemach)

Cascading termination

Wielowątkowość

Jeden proces wykonuje się w wielu współbieżnych wątkach (ang. ***thread***).

Każdy wątek (inna nazwa: ***proces lekki, ang. lightweight***)

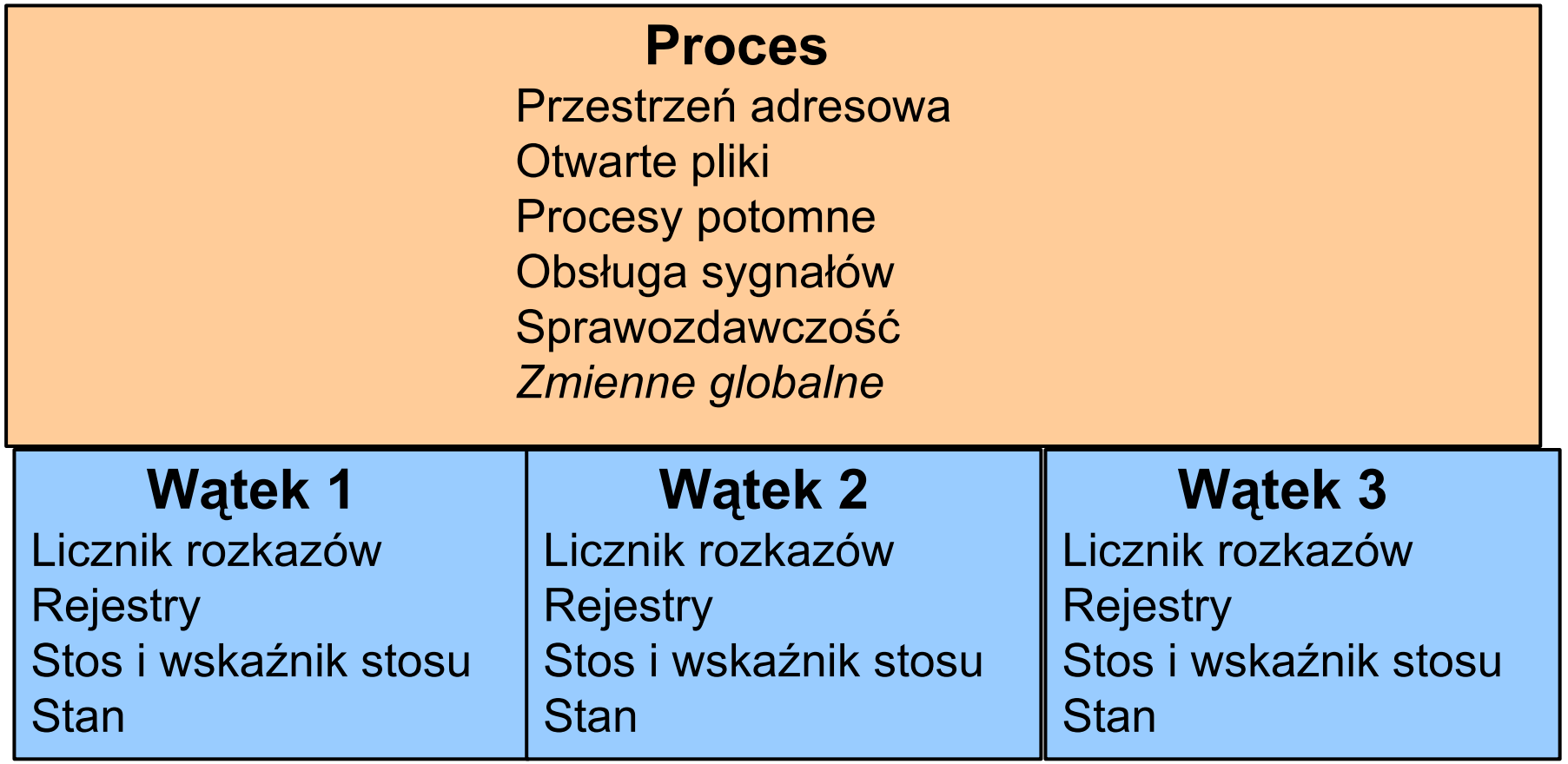
- Ma swój własny stan (Aktywny, Gotowy, Zablokowany, ...)
- Ma swoje wartości rejestrów i licznika rozkazów.
- Ma swój własny stos (zmienne lokalne funkcji !!!).
- Ma dostęp do przestrzeni adresowej, plików i innych zasobów procesu

WSZYSTKIE WĄTKI TO WSPÓŁDZIELĄ !!!

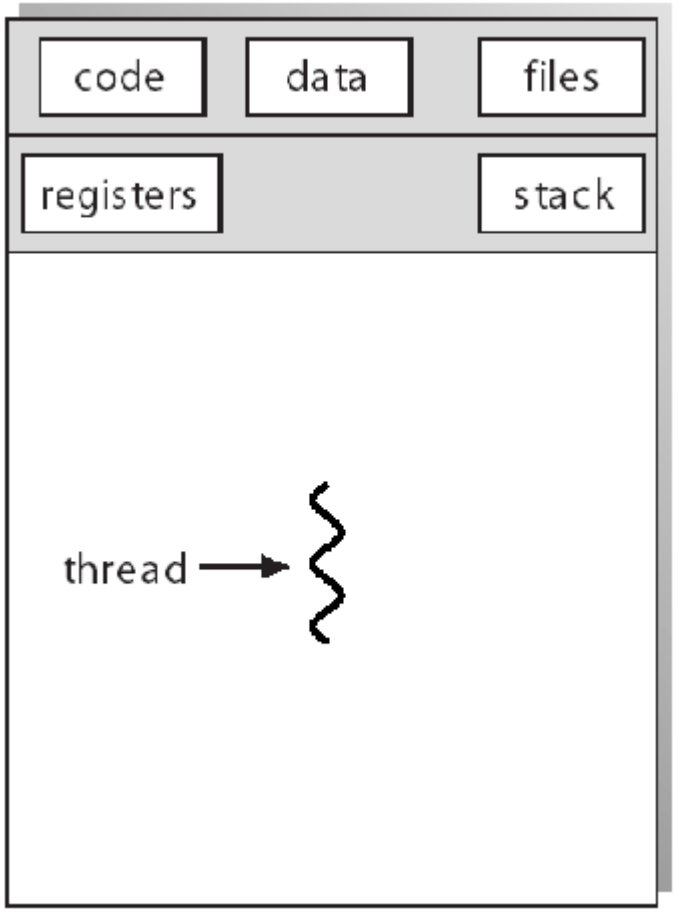
Operacje zakończenia, zawieszenia procesu dotyczą wszystkich wątków.

Procesy są od siebie izolowane, wątki nie !

Procesy i wątki



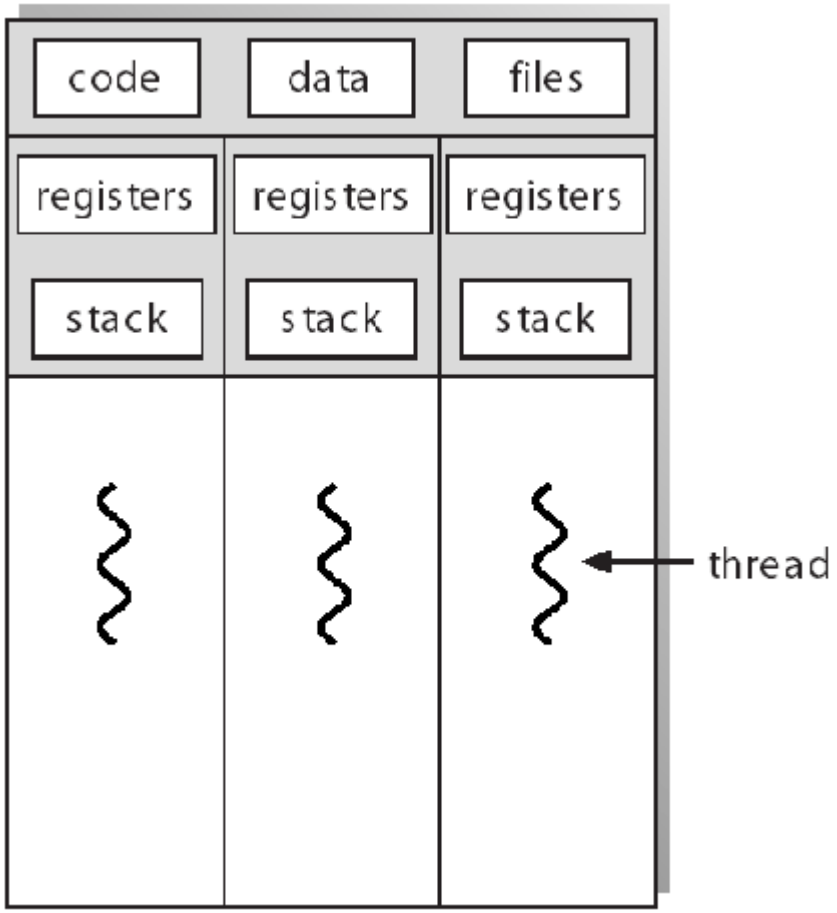
Proces z jednym wątkiem



Standardowy Unix

MS-DOS

Proces z wieloma wątkami



Linux

MS-Windows

POSIX

OS/2

Solaris

Cechy wątków

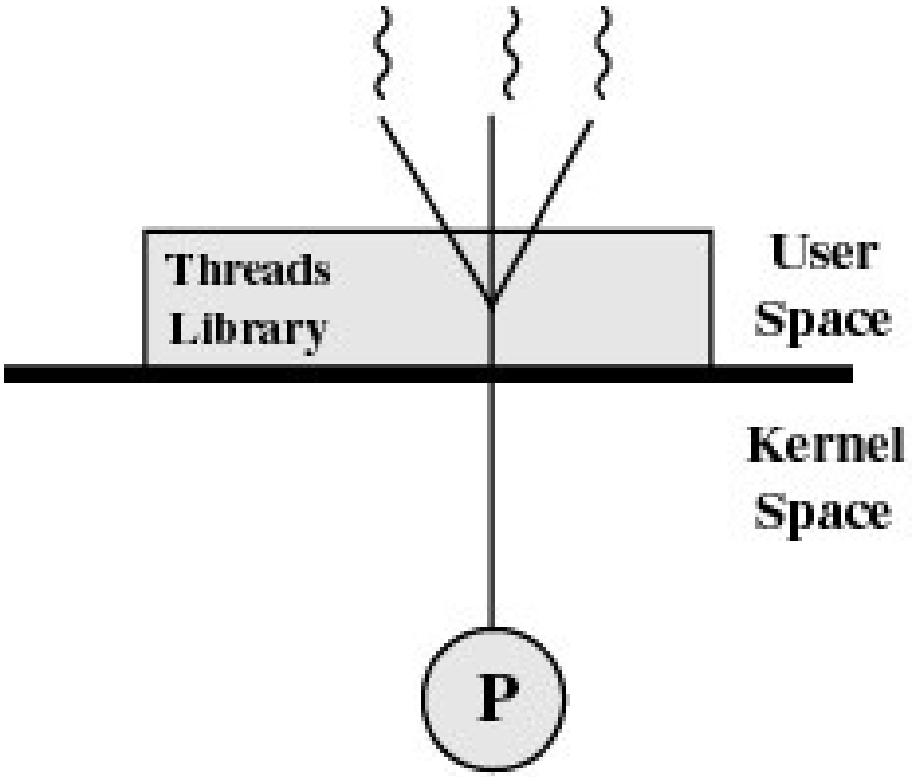
Zalety

- Utworzenie i zakończenie wątku zajmuje znacznie mniej czasu niż w przypadku procesu
- Możliwość szybkiego przełączania kontekstu pomiędzy wątkami tego samego procesu
- Możliwość komunikacji wątków bez pośrednictwa systemu operacyjnego
- Możliwość wykorzystania maszyn wieloprocesorowych SMP

Wady

- "Źle zachowujący się wątek" może zakłócić pracę innych wątków tego samego procesu.
  - W przypadku dwóch procesów o odrębnych przestrzeniach adresowych nie jest to możliwe

Wątki na poziomie użytkownika
ang. user-level threads



System operacyjny nie jest świadom istnienia wątków.

Zarządzanie wątkami jest przeprowadzane przez bibliotekę w przestrzeni użytkownika.

Wątek A wywołuje funkcję *read*

Standardowo funkcja systemowa read jest synchroniczna (usypia do momentu zakończenia operacji)

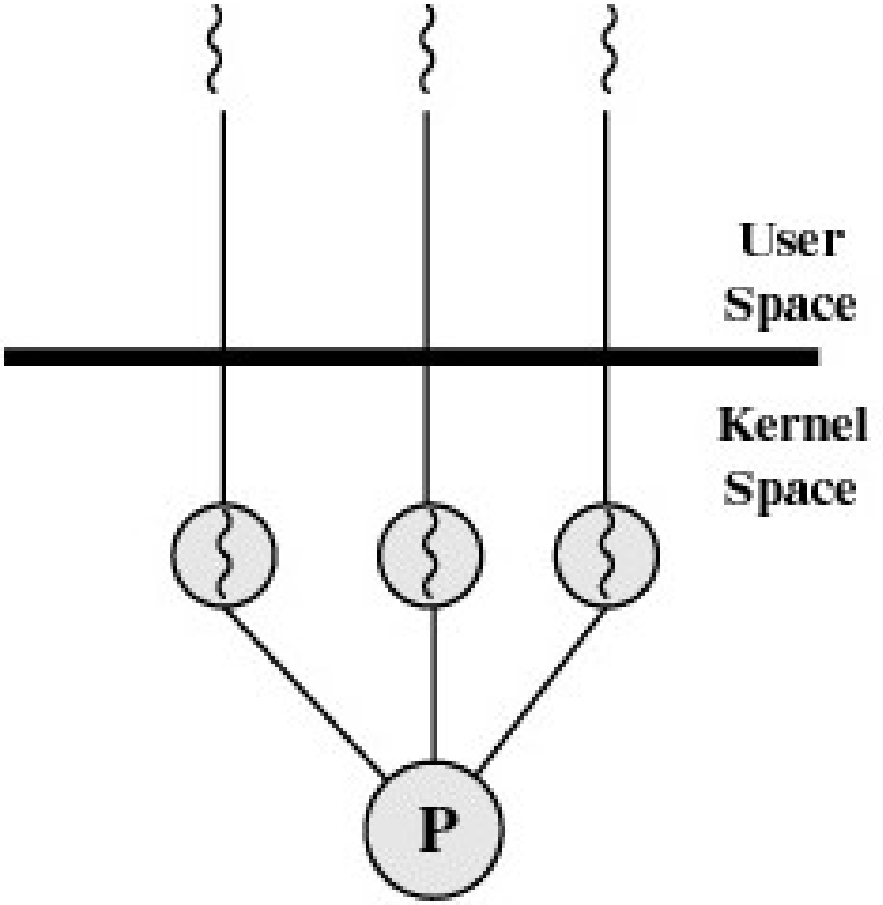
Jednak "sprytna" implementacja w bibliotece wywołuje wersję asynchroniczną i przełącza się do wątku B.

Rozwiązanie to jest szybkie, ma jednak wady:

Dwa wątki nie mogą się wykonywać współbieżnie na dwóch różnych procesorach.

Nie można odebrać procesora jednemu wątkowi i przekazać drugiemu

Wątki na poziomie jądra
ang. kernel-level threads



Wątek jest jednostką systemu operacyjnego.

Wątki podlegają szeregowaniu przez jądro.

W systemie SMP wątki mogą się wykonywać na różnych procesorach.

Przetwarzanie równoległe.

Windows i Linux wykorzystują tę metodę.

Przykład użycia wątków: Program do przetwarzania obrazów

Użytkownik wydał polecenie “Save”

- Jego wykonanie może potrwać kilkadziesiąt sekund.
- Kto w tym czasie będzie obsługiwał mysz i klawiature ?

Wątek główny odpowiadający za interakcję z użytkownikiem.

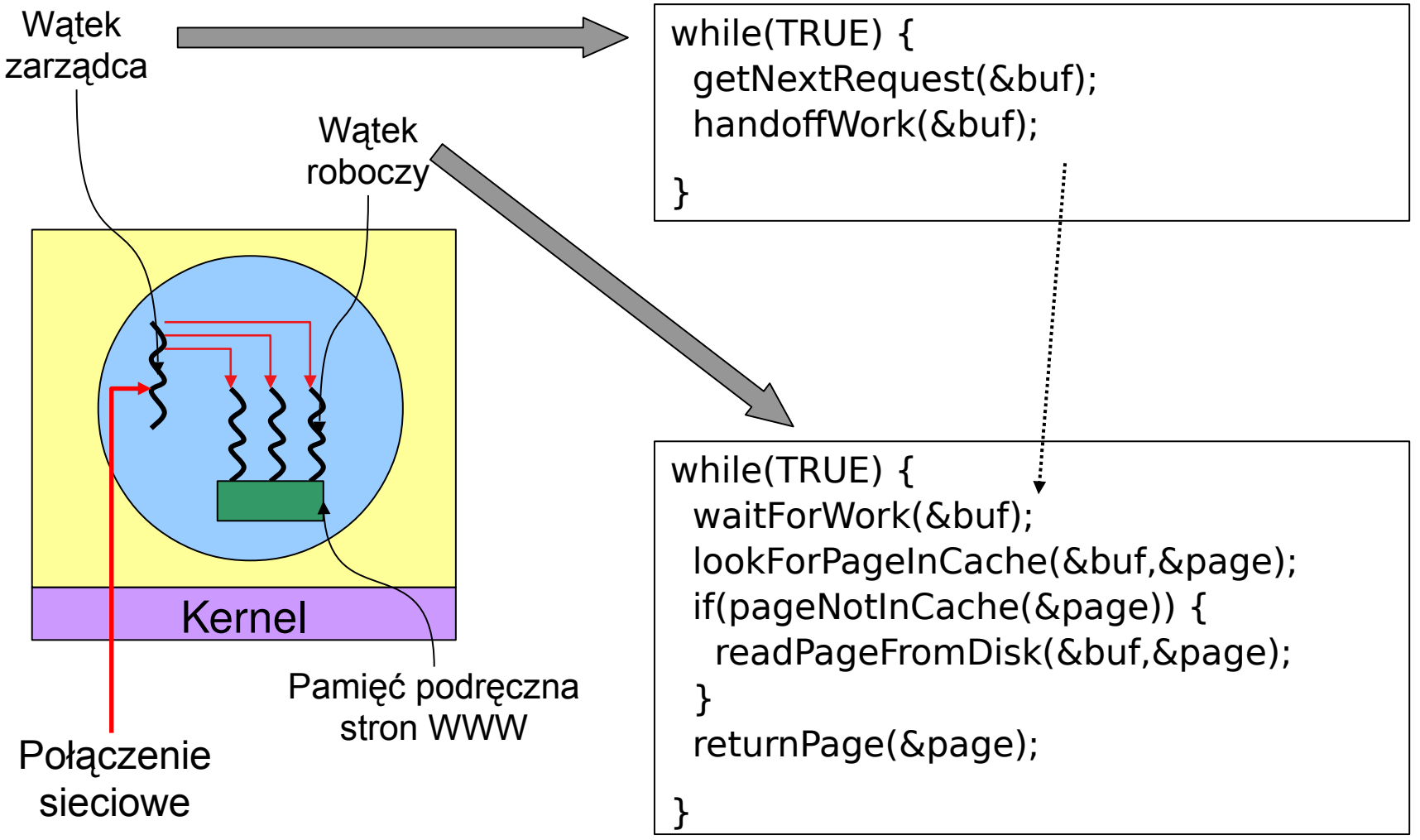
- Po wydaniu polecenia “Save” tworzy wątek roboczy zapisujący obraz do pliku.
- Następnie powraca do konwersacji z użytkownikiem
- Dzięki temu aplikacja nie jest zablokowana (klepsydra w Win) na czas zapisywania

Wątek roboczy wykonuje zapis i kończy pracę.

Uwaga na synchronizację wątków !!!

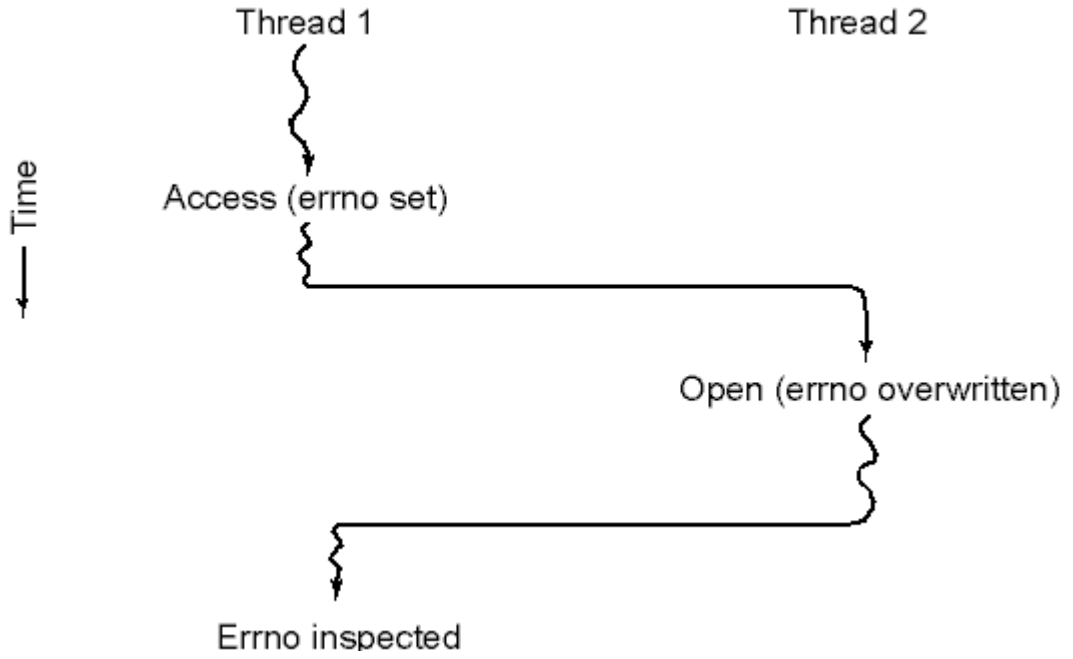
- Co się stanie gdy po wydaniu polecenia “Save” natychmiast wydamy polecenie “Usuń obraz”?
- Sytuacja ***wyścigu*** (ang. ***race***).

Przykład użycia wątków: Serwer WWW



Technika puli wątków: Zamiast tworzyć nowy wątek do obsługi kolejnego żądania, mamy zbiór wątków stale gotowych do działania. Nie ponosimy kosztu tworzenia i usuwania wątków.

Problemy z wątkami



W standardowej bibliotece C, w wersji wielowątkowej, **errno** jest implementowane jako prywatna zmienna globalna (nie współdzielona z innymi wątkami)

Kilka wątków jednocześnie wywołuje funkcje **malloc/free** - może dojść do uszkodzenia globalnych struktur danych (listy wolnych bloków pamięci)

Potrzeba synchronizacji => może prowadzić do spadku wydajności

Problemy z wątkami

Proces otrzymuje sygnał:

- Wszystkie wątki otrzymują sygnał
- Wybrany wątek otrzymuje sygnał
- Wątek aktualnie aktywny otrzymuje sygnał

Proces wykonuje ***fork***.

- Czy duplikować jedynie działający wątek, czy też wszystkie wątki ?

Proces wywołuje ***exit***.

- Zakończyć proces czy też jedynie aktywny wątek ?

Anulowanie wątku (ang cancellation).

- Wykonać natychmiast .
- Wątek co jakiś czas sprawdza czy nie został anulowany.

Standard POSIX zawiera odpowiedzi na powyższe problemy.

Implementacje wątków

Java threads – pomijamy na tym wykładzie, domena dr. Bołdaka

POSIX threads – na pracowni, za chwilę

Windows threads – podobne do POSIX

POSIX threads – utworzenie i dołączenie wątku

```
void *thread(void *param) {
// tu kod wątku


// możemy przekazać wynik
   return NULL
}

int main() {
    pthread_t id;
    // Parametr przekazywany wątkowi
    void *param=NULL;
    pthread_create(&id,NULL,&thread,param);
    // Funkcja thread w odrębnym wątku
    // współbieżnie z main
    // id przechowuje identyfikator wątku

    void *result;
    // Czekaj na zakończenie wątku
    pthread_join(id,&result);
    // Wynik w result,zamiast &result można
    // przekazać NULL
}
```

Funkcja pthread_create tworzy nowy wątek. Rozpoczyna on pracę od funkcji, której adres przekazano jako trzeci argument.

Funkcja pthread_join usypia wywołujący ją wątek do momentu, kiedy wątek o identyfikatorze przekazanym jako pierwszy argument zakończy pracę.

Zakończenie pracy wątku – powrót z funkcji która go rozpoczyna.