Systemy operacyjne

wykład 7 - zarządzanie procesami cz.1

dr Marcin Ziółkowski

Instytut Matematyki i Informatyki
Akademia im. Jana Długosza w Częstochowie

28 kwietnia 2016 r.

POJĘCIE PROCESU

PROCES (ZADANIE) to wykonujący się program. Jest jasne, że proces wykonuje się sekwencyjnie (instrukcje wykonywane są w określonej kolejności). Każdy proces posiada własny LICZNIK INSTRUKCJI oraz własny obszar przydzielonej pamięci operacyjnej, która zwykle jest podzielona na cztery segmenty:

- segment kodu - zawiera instrukcje programu
- segment danych - zawiera dane globalne programu
- stos - wykorzystywany do wywołania procedur, przekazywania parametrów i wyników oraz przechowywania danych lokalnych
- sterta - obszar przydziału pamięci dla zmiennych dynamicznych

Instrukcje z segmentu kodu mogą być wykorzystywane wielokrotnie, gdy na przykład program jest uruchamiany wielokrotnie, natomiast zawartość pozostałych segmentów może ulegać zmianom.

CYKL ŻYCIA PROCESU

Proces może znajdować się w jednym z pięciu stanów:

1. **nowy** - proces właśnie został utworzony
2. **aktywny** - proces jest właśnie wykonywany przez procesor
3. **czekający** - proces czeka na wykonanie pewnych akcji (np. operacji wejścia-wyjścia)
4. **gotowy** - proces czeka na przydzielenie mu procesora
5. **zakończony** - proces zakończył swoje działanie

MOŻLIWE TRANZYCJE MIĘDZY STANAMI PROCESU

Możliwe są jedynie następujące przejścia między tymi stanami:

1. **nowy** $\rightarrow$ **gotowy** - gdy program zostanie załadowany do pamięci operacyjnej
2. **gotowy** $\rightarrow$ **aktywny** - gdy moduł zwany PLANISTĄ przydzieli procesowi procesor
3. **aktywny** $\rightarrow$ **gotowy** - gdy PLANISTA odbierze procesowi procesor
4. **aktywny** $\rightarrow$ **czekający** - gdy proces rozpocznie oczekiwanie na zakończenie pewnego zdarzenia
5. **aktywny** $\rightarrow$ **zakończony** - gdy proces zakończy działanie
6. **czekający** $\rightarrow$ **gotowy** - gdy nastąpi oczekiwana przez proces akcja

Stan **zakończony** nie może zostać zmieniony (jest to stan końcowy). Przejście do tego stanu powoduje usunięcie procesu.

BLOK KONTROLNY PROCESU PCB

Blok kontrolny procesu (ang. **Process Control Block (PCB)**) to rekord zawierający zestaw informacji o stanie procesu. System operacyjny przechowuje informacje o procesach właśnie w postaci bloków kontrolnych i używa ich do planowania procesów. PCB zawiera następujące informacje:

- stan procesu
- licznik instrukcji
- wartości rejestrów procesora
- informacje związane z planowaniem,
- informacje związane z zarządzaniem pamięcią
- informacje rozliczeniowe
- informacje o stanie wejścia-wyjścia

Gdy proces przechodzi do stanu aktywny jego blok kontrolny służy do odtworzenia stanu obliczenia. Gdy proces opuszcza stan aktywny, w owym bloku zapisuje się stan obliczenia, żeby można je było potem odtworzyć.

PLANOWANIE PROCESÓW, KOLEJKI

Planowanie procesów polega na wskazywaniu procesu, któremu ma być w danej chwili przydzielony procesor. W szczególności oznacza to decydowanie, kiedy i który proces ma przejść ze stanu gotowy do stanu aktywny. W systemie w każdej chwili może być aktywnych co najwyżej tyle procesów ile jest procesorów. W przypadku jednego procesora tylko jeden proces ma przydzielony procesor, a wszystkie pozostałe procesy muszą czekać na przydział procesora. Procesy nieaktywne czekają na przydział procesora w kolejkach. W systemie istnieje wiele takich kolejek:

- Kolejka zadań - znajdują się w niej wszystkie procesy w systemie
- Kolejka gotowych - znajdują się w niej procesy, które są gotowe do wykonania (ich stan to gotowy)
- Kolejka do urządzenia - dla każdego urządzenia wejścia-wyjścia istnieje odrębna taka kolejka. Znajdują się w niej procesy, które czekają na wykonanie operacji wejścia-wyjścia na tymże urządzeniu

PLANOWANIE PROCESÓW, KOLEJKI

Procesy mogą przenosić się między tymi kolejkami:

- Po upłynięciu kwantu czasu procesowi odbiera się procesor i umieszcza na końcu kolejki gotowych
- Po przydzieleniu procesowi procesora jest on usuwany z kolejki gotowych
- Gdy proces zleci wykonanie operacji wejścia-wyjścia, umieszcza się go w kolejce do urządzenia, na którym zlecił operację wejścia-wyjścia
- Po wykonaniu operacji wejścia-wyjścia proces trafia z powrotem do kolejki gotowych

Procesy mogą też oczekiwać w innych kolejkach, np. oczekiwać na jakieś zdarzenie albo na zakończenie działania procesu potomnego. Do obsługi każdej z tych kategorii oczekiwania służy oddzielna kolejka.

PRZEŁĄCZANIE KONTEKSTU

Zmiana wykonywanego procesu (gdy procesor jest przydzielany innemu procesowi z jakiegokolwiek powodu) wiąże się ze złożonymi operacjami zapisania stanu obliczenia poprzedniego procesu i odtworzenia stanu obliczeń nowego procesu. Jest to właśnie przełączenie kontekstu. Stanowi ono istotny narzut na działanie systemu, ponieważ w tym czasie system nie wykonuje niczego pożytecznego, a jedynie "czynności administracyjne". Planowanie procesów nie może więc zbyt często powodować zmiany wykonywanego procesu, ponieważ wówczas większość czasu pracy będzie poświęcona na ciągłe przełączanie kontekstu. Szybkość przełączania kontekstu zależy też od wsparcia sprzętowego. W pewnych architekturach występuje po kilka zbiorów rejestrów. Wówczas przełączenie kontekstu polega po prostu na zmianie wskaźnika bieżącego zestawu rejestrów, nie trzeba ich natomiast zapamiętywać i odtwarzać z PCB.

PLANISTA

Selekcji procesu, który ma przejść do stanu aktywny, dokonuje odpowiedni proces systemowy. Nazywamy go planistą. Możemy wyróżnić dwa rodzaje planistów:

1. Planista długoterminowy (albo planista zadań)

   Decyduje o tym, który z procesów ma być załadowany do pamięci i gotowy do wykonania. Swoje działania podejmuje stosunkowo rzadko (np. co kilka minut). Kontroluje on poziom wieloprogramowości, czyli liczbę współbieżnie wykonywanych procesów. Może on przyjąć na przykład strategię utrzymywania stałej liczby procesów gotowych. Wówczas podejmuje działania jedynie wtedy, gdy jakiś proces przechodzi do stanu zakończony.

2. Planista krótkoterminowy (albo planista procesora)

   Decyduje o tym, który z procesów gotowych ma być wykonywany na procesorze. Swoje działania podejmuje stosunkowo często (np. co 10-100 milisekund), żeby użytkownik miał wrażenie płynnej współbieżności wszystkich działających procesów. Jest to szczególnie ważne w systemach interakcyjnych z podziałem czasu.

TWORZENIE PROCESU

Proces może zlecić utworzenie procesu potomnego. Ten pierwszy proces nazywamy wówczas procesem **macierzystym**, a drugi **potomnym**. W wyniku ciągu operacji tworzenia powstaje drzewo "genealogiczne", zwane drzewem procesów. Proces potomny potrzebuje zasobów. W systemach operacyjnych spotykamy różne podejścia do przydzielania zasobów procesom potomnym. Jedno z podejść polega na tym, że proces potomny może współdzielić zasoby z procesem macierzystym. Proces macierzysty decyduje wówczas, który z potomków może korzystać z jego konkretnych zasobów. Ograniczenie procesów dostępnych dla procesu potomnego do zasobów przodka zapobiega nadmiernemu obciążeniu systemu, do którego mogłoby dojść w wyniku namnożenia procesów potomnych. Jeśli stworzenie procesu potomnego nie powiększa zasobów przydzielonych do realizacji zadania, tworzenie podprocesów często nie ma uzasadnienia. Inna możliwość polega na przekazaniu procesowi potomnemu niezbędnych zasobów bezpośrednio przez system operacyjny. Wówczas oba procesy nie mają wspólnych zasobów.

TWORZENIE PROCESU

Proces macierzysty i potomny mogą wykonywać się współbieżnie. Często proces potomny jest przywoływany do życia w celu realizacji jakiegoś zadania potrzebnego procesowi macierzystemu. Wówczas proces macierzysty czeka na zakończenie procesu potomnego. Nota bene, tak właśnie wygląda wykonywanie poleceń przez bash-a. Interpreter poleceń tworzy proces potomny, który wykonuje zleconą operację, a proces macierzysty czeka na jego zakończenie. Po zakończeniu procesu potomnego, proces macierzysty jest gotowy do przyjęcia kolejnego polecenia.

PROCESY W UNIKSIE

W systemie Unix nowy proces jest tworzony w wyniku wywołania przez proces macierzysty funkcji systemowej **fork**. Powstaje wówczas nowy proces, który jest dokładną kopią macierzystego (wykonuje ten sam program, jest w tym samym miejscu obliczeń i jego pamięć ma taką samą zawartość). Procesy te mają oczywiście różne identyfikatory w systemie (gdyż każdy proces musi mieć unikatowy identyfikator), a także mają różne identyfikatory procesów macierzystych (przodkiem nowego potomka jest proces, który wywołał fork; przodkiem procesu, który wywołał fork, jest jakiś inny proces, ponieważ on sam nie mógł się z siebie narodzić). Różnią się one też wynikiem zwracanym przez funkcję fork - proces macierzysty otrzymuje w wyniku identyfikator procesu potomnego, a proces potomny otrzymuje liczbę 0. W ten sposób, choć oba procesy wykonują ten sam program i są w tym samym miejscu obliczeń, łatwo mogą odróżnić, który z nich jest macierzysty, a który potomny.

Proces potomny zwykle musi wykonać inny program niż ten wykonywany przez proces macierzysty. Do tego celu służy polecenie **exec**, które powoduje załadowanie nowego programu do przestrzeni adresowej procesu i przystąpienie do jego wykonywania. Stan starego obliczenia jest zamazywany. Nie jest to więc "wywołanie" jednego programu przez drugi, ale "przeistoczenie" się wykonania jednego programu w wykonanie drugiego programu. Zwróćmy uwagę na eleganckie rozdzielenie pojęć. Tworzenie procesu i zmiana wykonywanego programu to odrębne czynności. W Uniksie je rozdzielono, co doprowadziło do powstania naprawdę elastycznego mechanizmu. Nie zrobiono tego na przykład w wielu innych bibliotekach, w których występuje polecenie **spawn** będące połączeniem fork i exec. Nie jest to dobre rozwiązanie, ponieważ między fork i exec dzieje się bardzo dużo interesujących rzeczy.

ZAKOŃCZENIE PROCESU

Po wykonaniu swojej ostatniej instrukcji proces informuje system o ukończeniu działań (**exit**). System przekazuje informację o tym procesowi macierzystemu (jeżeli ten akurat na to oczekuje) i zwalnia zasoby przydzielone potomkowi. Do oczekiwania na zakończenie procesu potomnego służy funkcja systemowa **wait**. Proces macierzysty i potomny wykonują się współbieżnie, i nie da się zagwarantować, że proces macierzysty wywoła funkcję wait zanim proces potomny się zakończy. Problem ten rozwiązano w ten sposób, że jeżeli proces potomny kończy działanie, a proces macierzysty nie oczekuje na jego zakończenie, to proces potomny zamienia się w tzw. zombie. Wszystkie związane z nim zasoby systemowe zostają zwolnione, ale jego identyfikator jest nadal obecny w systemie. Gdy proces macierzysty wywoła funkcję wait, to wówczas zombie znika. W ten sposób, bez względu na to, czy proces potomny zakończy się pierwszy, czy też proces macierzysty pierwszy wywoła funkcję wait, proces macierzysty może czekać na zakończenie się procesu potomnego.

ZAKOŃCZENIE PROCESU

Proces macierzysty może też zażądać zakończenia procesu potomnego (**kill**). Dzieje się tak na przykład, gdy ten drugi przekroczy limit przydzielonych mu zasobów albo jego praca nie jest już potrzebna. Ciekawa sytuacja występuje, gdy proces macierzysty kończy działanie, a pracują jeszcze procesy potomne. System operacyjny może wówczas zakończyć wszystkie procesy potomne, albo też mogą one zostać "adoptowane" przez inny proces, który od tej pory będzie ich przodkiem. W Uniksie procesem, który adoptuje takie osierocone procesy jest proces o identyfikatorze 1 (tzw. **init**).

POJĘCIE WĄTKU

Proces jest bardzo złożoną i kosztowną strukturą (ma swój segment kodu, stos, stertę i segment danych). Utworzenie procesu potomnego wymaga stosunkowo dużo czasu i pamięci. Z tego powodu zwykłe procesy nazywamy ciężkimi. W ramach procesu można powołać do życia kilka lżejszych struktur nazywanych **wątkami** albo **procesami lekkimi**. Wątki jednego procesu są wykonywane współbieżnie (tak więc stwierdzenie o sekwencyjnym wykonywaniu procesów odnosi się raczej do wątków niż do ciężkich procesów).

POJĘCIE WĄTKU

Każdy z nich ma oddzielne rejestry i stos wywołań, jednak współdzieli z innymi wątkami w ramach tego samego procesu segment danych, segment kodu, stertę, tablicę otwartych plików itp. Można powiedzieć, że proces to grupa wykonujących go wątków (przynajmniej jednego). Korzyści ze stosowania wątków są rozmaite. Przede wszystkim wydajność: powołanie do życia wielu wątków jest znacznie tańsze niż stworzenie wielu ciężkich procesów. Wątki mogą bez ograniczeń współdzielić przestrzeń danych procesu. Gdybyśmy chcieli tak samo prosto skomunikować procesy (przez wspólny obszar pamięci), musielibyśmy skorzystać ze złożonej usługi systemu operacyjnego, jaką jest pamięć dzielona.

WĄTKI W LINUXIE

W Linuksie, do tworzenia wątków służy funkcja **clone**. Jej dwa najważniejsze argumenty to wskaźnik do procedury, która ma być wykonywana przez nowy wątek oraz wskaźnik na przestrzeń na stos dla nowego wątku. W odróżnieniu od procedury fork, nowy wątek nie kontynuuje tego samego fragmentu kodu, co wątek macierzysty. Zamiast tego, wykonuje procedurę, na którą wskazuje wskaźnik przekazany do funkcji clone. Gdy wykonanie tej procedury kończy się lub wątek wykona operację exit, wątek kończy się. Stos jest wykorzystywany do wywoływania procedur i funkcji. Ponieważ wątki działają współbieżnie, każdy z nich musi mieć własny stos - ten zasób nie może być dzielony. Z drugiej strony wątki wykonują ten sam program i współdzielą obszar danych - gdyby tak nie miało być, to zamiast wątków lepiej jest utworzyć osobne procesy. Jeśli chodzi o pozostałe zasoby (np. tablicę deskryptorów otwartych plików), to przekazując odpowiednie opcje funkcji clone można określić, czy wątek potomny ma współdzielić te zasoby z wątkiem macierzystym, czy też mieć ich własną kopię. Na wątkach można wykonywać podobne operacje jak na procesach: wait, kill itd.

WĄTKI W JAVIE

W języku programowania Java jest możliwość tworzenia wątków. Pamiętajmy, że programy w Javie są wykonywane przez maszynę wirtualną Javy, która działa pod jakimś systemem operacyjnym. Zwykle, wątkom programu w Javie odpowiadają systemowe wątki maszyny wirtualnej, która je wykonuje. Wątki można tworzyć w Javie na dwa, bliźniaczo podobne, sposoby. Po pierwsze, należy utworzyć albo podklasę klasy **Thread**, albo klasę implementującą interfejs **Runnable** (to drugie rozwiązanie stosuje się zwykle wtedy, gdy chcemy utworzyć podklasę innej klasy niż Thread). W obu przypadkach utworzona klasa musi udostępniać bezargumentową metodę **run**. Każdy obiekt takiej klasy może być wątkiem. Konstruktor takiego obiektu może skonfigurować wszystko co jest potrzebne do późniejszego działania wątku. Wątek jest uruchamiany dopiero, gdy wykona się na nim bezargumentową metodę **start**. Wykonuje on metodę run, a gdy skończy ją wykonywać, wątek kończy się.

PRZYKŁADY WĄTKÓW W JAVIE

Oto przykład programu w javie realizującego wątki:

JAVA - PROGRAM REALIZUJĄCY WĄTKI

```
public class MyThread extends Thread {
    public void run() {
      System.out.println("To mój pierwszy wątek!");
    }
    public static void main(String args[]) {
      System.out.println("Ruszamy ...");
      new MyThread().start();
      System.out.println("Koniec.");
    }
 }
```

PRZYKŁADY WĄTKÓW W JAVIE

I jeszcze jeden przykład:

JAVA - PROGRAM WIELOWĄTKOWY

```
public class Watki extends Thread {
    String w;
    Watki (String s) {w = s;}
    public void run() {
      System.out.println("To mój " + w + " wątek!"); }
public static void main(String args[])
throws InterruptedException {
      Thread t1 = new Watki("1");
      Thread t2 = new Watki("2");
      Thread t3 = new Watki("3");
      Thread t4 = new Watki("4");
      Thread t5 = new Watki("5");
      t1.start();
      t2.start();
      t3.start();
      t4.start();
      t5.start(); }}
```

WĄTKI W JAVIE

Metody odpowiednich klas w JAVIE realizują odpowiednie przejścia między stanami wątków.

- nowy wątek jest tworzony przez operację **new Thread()**, wątek po utworzeniu jest w stanie **wstrzymany**
- przejście wątku ze stanu **wstrzymany** do stanu **aktywny** jest realizowane przez metodę **run()**
- przejście wątku ze stanu **aktywny** do stanu **gotowy** jest realizowane przez metodę **yeld()**
- przejście wątku ze stanu **aktywny** do stanu **czekający** jest realizowane przez metody **join()** oraz **wait()**
- przejście wątku ze stanu **czekający** do stanu **gotowy** jest realizowane przez metodę **notify()**
- przejście wątku ze stanu **aktywny** do stanu **wstrzymany** jest realizowane przez metodę **sleep(...)**
- przejście wątku ze stanu **aktywny** do stanu **zakończony** jest realizowane przez metodę **stop()**

Przypomina to diagram możliwych stanów procesu. Jest jednak parę różnic: nowo utworzony wątek pozostaje w stanie wstrzymany, aż nie zostanie jawnie uruchomiony. Może on też powrócić do tego stanu (na pewien czas) jeżeli wykona operację sleep. Wątek może też sam odebrać sobie procesor i zmienić stan na gotowy, jeśli wykona operację yeld.