Procesy

Procesy umożliwiają w systemach operacyjnych (quasi)równoległe wykonywanie wielu zadań. Mogą być wykonywane naprzemiennie i/lub jednocześnie.

Zatem system operacyjny musi to (quasi)równoległe wykonywanie zapewnić.

Poza tym musi umożliwić procesom pewne operacje, których nie mogą one sobie zapewnić same, np. dostęp do globalnych zasobów systemu jak port komunikacyjny.

Idąc dalej, systemy operacyjne mogą również dostarczać procesom dalszych usług, jak np. komunikacja międzyprocesowa, a także pewnych funkcji synchronizacji, jak blokady, semafory, itp.

Model procesów systemu UNIX (standard POSIX)

- Proces uniksowy jest wykonywalnym programem załadowanym do pamięci operacyjnej komputera. Każdy proces uniksowy wykonuje się we własnej wirtualnej przestrzeni adresowej.
- Proces jest normalnie tworzony z trzema otwartymi plikami: stdin, stdout i stderr przypisanymi do terminala użytkownika interakcyjnego.
- Jądro Uniksa utrzymuje tablicę wszystkich istniejących procesów wraz z zestawem informacji kontekstowych o nich, np. zestawem otwartych plików, terminalem sterującym, priorytetem, maską sygnałów, i innymi.
- Proces posiada **środowisko**, które jest zestawem dostępnych dla procesu zmiennych środowiskowych z wartościami (tekstowymi). Środowisko nazywa się zewnętrznym ponieważ jest początkowo tworzone dla procesu przez system, a potem jest również dostępne na zewnątrz procesu.
- Proces tworzony jest z zestawem argumentów wywołania programu, który jest wektorem stringów
- Proces charakteryzują: pid, ppid, pgrp, uid, euid, gid, egid

Tworzenie procesów

- Procesy tworzone są przez klonowanie istniejącego procesu funkcją `fork()`, zatem każdy proces jest podprocesem (*child process*) jakiegoś innego procesu nadrzędnego, albo inaczej rodzicielskiego (*parent process*).

- Jedynym wyjątkiem jest proces numer 1 — **init** — tworzony przez jądro Uniksa w chwili startu systemu. Wszystkie inne procesy w systemie są bliższymi lub dalszymi potomkami inita.

- Podproces dziedziczy i/lub współdzieli pewne atrybuty i zasoby procesu nadrzędnego, a gdy kończy pracę, Unix zatrzymuje go do czasu odebrania przez proces nadrzędny statusu podprocesu (wartości zakończenia).

Kończenie pracy procesów

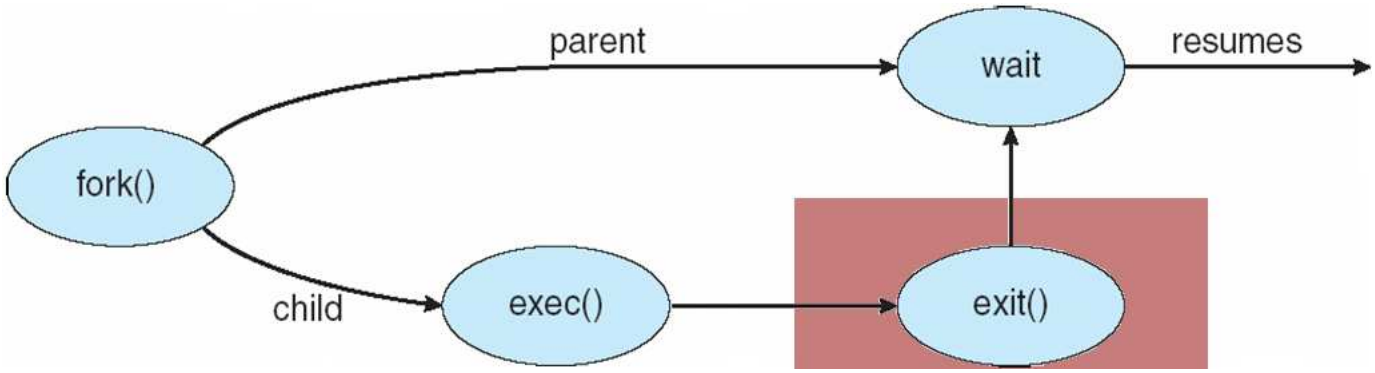
- Proces uniksowy kończy się normalnie albo anormalnie (np. przez otrzymanie sygnału), zawsze zwracając kod zakończenia, tzw. *status*.

- Po śmierci procesu jego rodzic normalnie powinien odczytać status potomka wywołując funkcję systemową `wait`. Dopóki status nie zostanie przeczytany, proces nie może umrzeć do końca i pozostaje w stanie zwanym *zombie*.

- Istnieje mechanizm adopcji polegający na tym, że procesy, których rodzic zginął przed nimi (sieroty), zostają adoptowane przez proces nr 1 (`init`). `init` wywołuje okresowo funkcję `wait` aby umożliwić poprawne zakończenie swoich potomków (zarówno naturalnych jak i adoptowanych).

- Gdy proces nadrzędny żyje w chwili zakończenia pracy potomka, i nie wywołuje funkcji `wait`, to potomek pozostaje w stanie *zombie* na czas nieograniczony, co może być przyczyną wyczerpania jakichś zasobów systemu (np. zapełnienia tablicy procesów).

Tworzenie procesów — funkcja `fork`



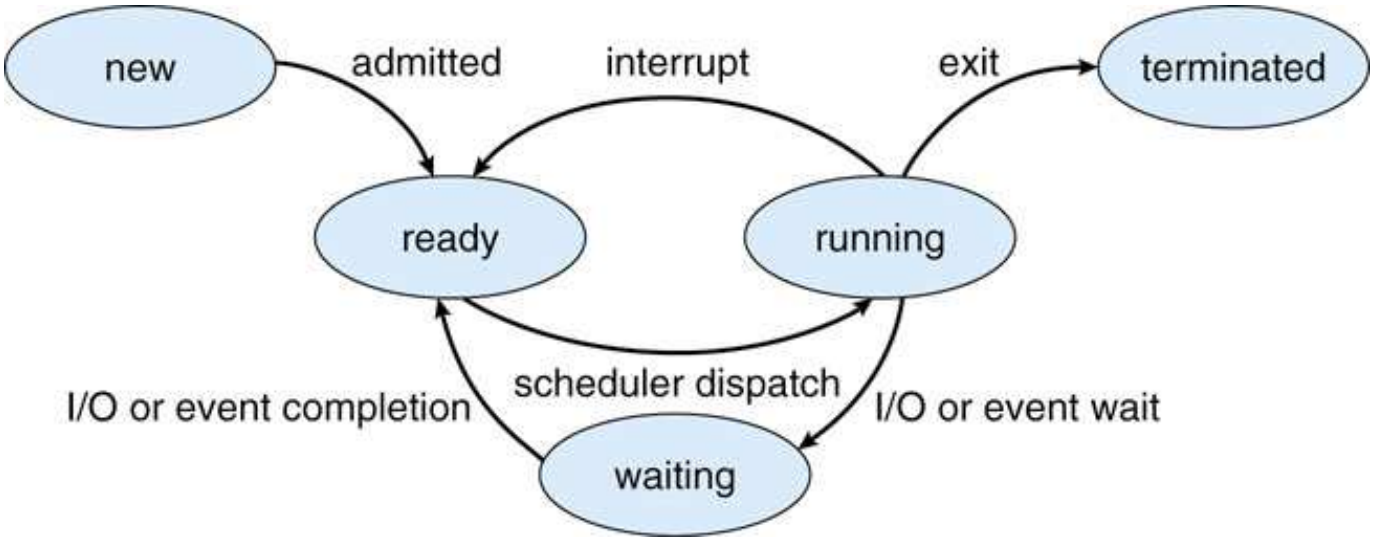
- Funkcja `fork` tworzy podproces, który jest kopią procesu macierzystego.
- Od momentu utworzenia oba procesy pracują równolegle i kontynuują wykonywanie tego samego programu. Jednak funkcja `fork` zwraca w każdym z nich inną wartość. Poza tym różnią się identyfikatorem procesu PID (rodzic zachowuje oryginalny PID, a potomek otrzymuje nowy).
- Po sklonowaniu się podproces może wywołać jedną z funkcji grupy `exec` i rozpocząć w ten sposób wykonywanie innego programu.
- Po sklonowaniu się oba procesy współdzielą dostęp do plików otwartych przed sklonowaniem, w tym do terminala (plików `stdin`, `stdout`, `stderr`).
- Funkcja `fork` jest jedyną metodą tworzenia nowych procesów w systemach uniksowych.

Status procesu

- W chwili zakończenia pracy proces generuje kod zakończenia, tzw. **status**, który jest wartością zwróconą z funkcji `main()` albo `exit()`. W przypadku śmierci przez otrzymanie sygnału system generuje dla procesu status o wartości 128+nr_sygnału.
- Rodzic normalnie powinien po śmierci potomka odczytać jego status wywołując funkcję systemową `wait` (lub `waitpid()`). Aby to ułatwić (w nowszych systemach) w chwili śmierci potomka jego rodzic otrzymuje sygnał `SIGCLD` (domyślnie ignorowany).
- Gdy proces nadrzędny żyje w chwili zakończenia pracy potomka, i nie wywołuje funkcji `wait()`, to potomek pozostaje (na czas nieograniczony) w stanie zwanym **zombie**. Może to być przyczyną wyczerpania jakichś zasobów systemu, np. zapełnienia tablicy procesów, otwartych plików, itp.
- Istnieje mechanizm adopcji polegający na tym, że procesy, których rodzic zginął przed nimi (sieroty), zostają adoptowane przez proces numer 1, init. Init jest dobrym rodzicem (chociaż przybranym) i wywołuje okresowo funkcję `wait()` aby umożliwić poprawne zakończenie swoich potomków.

Stany procesów

stan	ang.	znaczenie
wykonywalny/gotowy	*runnable*	proces gotowy w kolejce do wykonywania
wykonywany	*running*	proces wykonujący się na procesorze
oczekujący/uśpiony	*waiting/sleeping*	proces czeka na dostęp do zasobu



System przenosi proces w stan oczekiwania/uśpienia jeśli nie może mu czegoś dostarczyć, np. dostępu do pliku, danych, itp. W stanie uśpienia proces nie zużywa czasu procesora, i jest *budzony* i wznawiany w sposób przez siebie niezauważony.

Priorytety procesów

Procesy uniksowe mają priorytety określające kolejność ich wykonywania w procesorze w przypadku, gdy liczba procesorów komputera jest mniejsza niż liczba gotowych do wykonywania procesów. Jądro realizuje algorytm kolejkowania procesów do wykonania, zwany schedulerem, który opiera się na priorytetach procesów, i jednocześnie może je zmieniać. Ogólnie, procesy interakcyjne zyskują wyższe priorytety, ponieważ mają związek z pracą człowieka, który nie chce czekać na reakcję komputera, ale jednocześnie pracuje z przerwami, które zwykle pozwalają na wykonywanie innych procesów.

Wszelkie modyfikacje priorytetów procesów i ingerencje w pracę schedulera są zaawansowanymi operacjami, dla zwykłych użytkowników trudnymi i niebezpiecznymi. Jednak istnieje prosty mechanizm pozwalający użytkownikowi wspomagać scheduler w wyznaczaniu priorytetów procesów. Tym mechanizmem są liczby **nice** dodawane do priorytetów. Użytkownik może uruchamiać procesy z zadaną liczbą nice (służy do tego polecenie `nice`), lub zmieniać liczbę nice wykonujących się procesów (polecenie `renice`). Co prawda zwykli użytkownicy mogą tylko zwiększać domyślną wartość nice co obniża priorytet procesu, ale można — uruchamiając w ten sposób swoje procesy „obliczeniowe” — efektywnie jakby zwiększyć priorytet procesów „interakcyjnych”.

Ograniczenia zużycia zasobów dla procesu

System operacyjny może i powinien ograniczać wielkość zasobów zużywanych przez procesy, w celu realizacji swoich funkcji zapewnienia sprawnej i efektywnej pracy.
Systemy uniksowe definiują szereg zasobów, które mogą być limitowane dla procesu.
Proces może swoje ograniczenia ustawiać funkcją systemową `ulimit`. Istnieje również polecenie wbudowane shella o tej nazwie.

ulimit	limit	nazwa zasobu	opis zasobu
-c	coredumpsize	RLIMIT_CORE	plik core (zrzut obrazu pamięci) w kB
-d	datasize	RLIMIT_DATA	segment danych procesu w kB
-f	filesize	RLIMIT_FSIZE	wielkość tworzonego pliku w kB
-l	memorylocked	RLIMIT_MEMLOCK	obszar, który można zablokować w pamięci w kB
-m	memoryuse	RLIMIT_RSS	zbiór rezydentny w pamięci w kB
-n	descriptors	RLIMIT_NOFILE	liczba deskryptorów plików (otwartych plików)
-s	stacksize	RLIMIT_STACK	wielkość stosu w kB
-t	cputime	RLIMIT_CPU	wykorzystanie czasu CPU w sekundach
-u	maxproc	RLIMIT_NPROC	liczba procesów użytkownika
-v	vmemoryuse	RLIMIT_VMEM	pamięć wirtualna dostępna dla procesu w kB

Dla każdego zasobu istnieją dwa ograniczenia: miękkie i twarde. Obowiązujące jest ograniczenie miękkie, i użytkownik może je dowolnie ustawiać, lecz tylko do maksymalnej wartości ograniczenia twardego. Ograniczenia twarde można również zmieniać, lecz procesy zwykłych użytkowników mogą je jedynie zmniejszać.
Ograniczenia zasobów są dziedziczone przez podprocesy.

Konta użytkownika

Konta użytkownika istnieją dla zapewnienia izolacji i zabezpieczenia działających procesów. W systemach uniksowych istnieje jedno główne konto użytkownika o numerze 0 zwane *root*. Główne programy zapewniające funkcjonowanie systemu, i uruchamiane przez jądro, działają w ramach konta użytkownika root.

Można tworzyć dowolną liczbę innych kont, zarówno dla ludzi-użytkowników systemu, jak i dla określonych programów lub ich grup, których uruchamianie wymaga izolacji lub zabezpieczenia przez innymi użytkownikami.

Procesy jednego użytkownika nie mogą ingerować w procesy innego użytkownika. To znaczy, nie mogą ich zatrzymywać, wysyłać im sygnałów, tworzyć ani usuwać blokad, ustawiać ograniczeń zasobów, itp. Nie dotyczy to konta użytkownika root, który jest traktowany jako użytkownik nadrzędny, i jego procesy mogą ingerować w procesy innych użytkowników systemu.

Procesy różnych użytkowników mogą natomiast brać udział w komunikacji międzyprocesowej (wysyłać sobie komunikaty), o ile dobrowolnie taką komunikację nawiążą.

Sygnały

- Sygnały są mechanizmem asynchronicznego powiadamiania procesów o wydarzeniach.

- Domyślną reakcją procesu na otrzymanie większości sygnałów jest natychmiastowe zakończenie pracy, czyli śmierć procesu. Oryginalnie sygnały miały służyć do sygnalizowania błędów i sytuacji nie dających się skorygować. Niektóre sygnały powodują tuż przed uśmierceniem procesu, wykonanie zrzutu jego obrazu pamięci do pliku dyskowego o nazwie `core`.

- W trakcie wieloletniego rozwoju systemów uniksowych mechanizm sygnałów wykorzystywano do wielu innych celów, i wiele sygnałów nie jest już w ogóle związanych z błędami. Zatem niektóre sygnały mają inne reakcje domyślne, na przykład są domyślnie ignorowane.

- Proces może zadeklarować własną procedurę obsługi lub ignorowanie sygnału, (z wyjątkiem sygnałów SIGKILL i SIGSTOP, których nie można przechwycić ani ignorować).

nr	nazwa	domyślnie	zdarzenie
1	SIGHUP	śmierć	rozłączenie terminala sterującego
2	SIGINT	śmierć	przerwanie z klawiatury (zwykle: Ctrl-C)
3	SIGQUIT	zrzut	przerwanie z klawiatury (zwykle: Ctrl-\)
4	SIGILL	zrzut	nielegalna instrukcja
5	SIGTRAP	zrzut	zatrzymanie w punkcie kontrolnym (breakpoint)
6	SIGABRT	zrzut	sygnał generowany przez funkcję abort
8	SIGFPE	zrzut	nadmiar zmiennoprzecinkowy
9	SIGKILL	śmierć	bezwarunkowe uśmiercenie procesu
10	SIGBUS	zrzut	błąd dostępu do pamięci
11	SIGSEGV	zrzut	niepoprawne odwołanie do pamięci
12	SIGSYS	zrzut	błąd wywołania funkcji systemowej
13	SIGPIPE	śmierć	błąd potoku: zapis do potoku bez odbiorcy
14	SIGALRM	śmierć	sygnał budzika (timera)
15	SIGTERM	śmierć	zakończenie procesu
16	SIGUSR1	śmierć	sygnał użytkownika
17	SIGUSR2	śmierć	sygnał użytkownika
18	SIGCHLD	ignorowany	zmiana stanu podprocesu (zatrzymany lub zakończony)
19	SIGPWR	ignorowany	przerwane zasilanie lub restart
20	SIGWINCH	ignorowany	zmiana rozmiaru okna
21	SIGURG	ignorowany	priorytetowe zdarzenie na gniazdku
22	SIGPOLL	śmierć	zdarzenie dotyczące deskryptora pliku
23	SIGSTOP	zatrzymanie	zatrzymanie procesu
24	SIGTSTP	zatrzymanie	zatrzymanie procesu przy dostępie do terminala
25	SIGCONT	ignorowany	kontynuacja procesu
26	SIGTTIN	zatrzymanie	zatrzymanie na próbie odczytu z terminala
27	SIGTTOU	zatrzymanie	zatrzymanie na próbie zapisu na terminalu
30	SIGXCPU	zrzut	przekroczenie limitu CPU
31	SIGXFSZ	zrzut	przekroczenie limitu rozmiaru pliku

Mechanizmy generowania sygnałów

- wyjątki sprzętowe: nielegalna instrukcja, nielegalne odwołanie do pamięci, dzielenie przez 0, itp. (SIGILL, SIGSEGV, SIGFPE)
- naciskanie pewnych klawiszy na terminalu użytkownika (SIGINT, SIGQUIT)
- wywołanie komendy kill przez użytkownika (SIGTERM, i inne)
- funkcje `kill` i `raise`
- mechanizmy software-owe (SIGALRM, SIGWINCH)

Sygnały mogą być wysyłane do konkretnego pojedynczego procesu, do wszystkich procesów z danej grupy procesów, albo wszystkich procesów danego użytkownika.
W przypadku procesu z wieloma wątkami sygnał jest doręczany do jednego z wątków procesu.

Obsługa sygnałów

Proces może zadeklarować jedną z następujących możliwości reakcji na sygnał:

- ignorowanie,
- wstrzymanie doręczenia mu sygnału na jakiś czas, po którym odbierze on wysłane w międzyczasie sygnały,
- obsługa sygnału przez wyznaczoną funkcję w programie, tzw. handler,
- przywrócenie domyślnej reakcji na dany sygnał.

Typowa procedura obsługi sygnału przez funkcję handlera może wykonać jakieś niezbędne czynności (np. skasować pliki robocze), ale ostatecznie ma do wyboru:

- zakończyć proces,
- wznowić proces od miejsca przerwania, jednak niektórych funkcji systemowych nie można wznowić dokładnie od miejsca przerwania,
- wznowić proces od miejsca przerwania z przekazaniem informacji przez zmienną globalną,
- wznowić proces od określonego punktu.

Komunikacja międzyprocesowa (IPC)

Procesy tworzone w ramach systemu operacyjnego mogą się komunikować. Jeżeli duża aplikacja jest budowana z wielu współpracujących procesów, to ta komunikacja może być intensywna, i jej poprawna oraz sprawna realizacja może być krytyczna dla poprawnej pracy całej aplikacji. Usługi komunikacji między procesowej (*Inter-Process Communication* — IPC), należą do ważnych funkcji systemu operacyjnego.

Ogólnie, rozróżnia się następujące modele komunikacji międzyprocesowej:

- przesyłanie komunikatów (*message passing*),
- pamięć wspólna/współdzielona (*shared memory*).

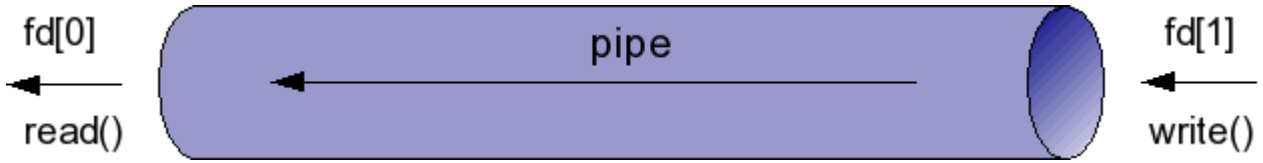
Następujące zagadnienia muszą być rozwiązane dla zapewnienia komunikacji międzyprocesowej:

- mechanizm komunikacji,
- zapobieganie kolizjom dostępu do wspólnego medium,
- organizacja oczekiwania i kolejkowania, gdy żądanie jednego procesu nie może być od razu zaspokojone.

Komunikacja międzyprocesowa — potoki

Potoki są jednym z najbardziej podstawowych mechanizmów komunikacji międzyprocesowej w systemach uniksowych. Potok jest urządzeniem komunikacji szeregowej, jednokierunkowej, o następujących własnościach:

- na potoku można wykonywać tylko operacje odczytu i zapisu, funkcjami `read` i `write`, jak dla zwykłych plików,
- potoki są dostępne i widoczne w postaci jednego lub dwóch deskryptorów plików, oddzielnie dla końca zapisu i odczytu,
- potok ma określoną pojemność, i w granicach tej pojemności można zapisywać do niego dane bez odczytywania,
- próba odczytu danych z pustego potoku, jak również zapisu ponad pojemność potoku, powoduje zawiśnięcie operacji I/O (normalnie), i jej automatyczną kontynuację gdy jest to możliwe; w ten sposób potok **synchronizuje** operacje I/O na nim wykonywane.



Dobrą analogią potoku jest rurka, gdzie strumień danych jest odbierany jednym końcem, a wprowadzany drugim. Gdy rurka się zapełni i dane nie są odbierane, nie można już więcej ich wprowadzić.

Potoki — podsumowanie

Potoki są prostym mechanizmem komunikacji międzyprocesowej opisane standardem POSIX i istniejącym w wielu systemach operacyjnych. Pomimo iż definicja określa potok jako mechanizm komunikacji jednokierunkowej, to wiele implementacji zapewnia komunikację dwukierunkową. Oznacza to istnienie w potoku dwóch przeciwbieżnych strumieni danych.

Podstawowe zalety potoków to:

- brak limitów przesyłania danych,
- synchronizacja operacji zapisu i odczytu przez stan potoku.
  Praktycznie synchronizuje to komunikację pomiędzy procesem, który dane do potoku zapisuje, a innym procesem, który chciałby je odczytać, niezależnie który z nich wywoła swoją operację pierwszy.

Jednak nie ma żadnego mechanizmu umożliwiającego synchronizację operacji I/O pomiędzy procesami, które chciałyby jednocześnie wykonać operacje odczytu, lub jednocześnie operacje zapisu. Z tego powodu należy uważać potoki za mechanizm komunikacji typu 1-1, czyli jeden do jednego. Komunikacja typu wielu-wielu przez potok jest utrudniona i wymaga zastosowania innych mechanizmów synchronizacji.

Komunikacja międzyprocesowa — inne mechanizmy

Kolejki komunikatów są mechanizmem komunikacji nieco podobnym do potoków. Istnieje wiele rodzajów kolejek komunikatów różniących się własnościami. W niektórych systemach kolejki komunikatów są urządzeniami sieciowymi. Pozwalają one komunikować się procesom wykonującym się na różnych komputerach. Takie kolejki stanowią mechanizm komunikacji międzyprocesowej dla systemów rozproszonych.

Gniazdka są mechanizmem komunikacji dwukierunkowej przeznaczonym zarówno do komunikacji międzyprocesowej w ramach jednego komputera, lub między procesami na różnych komputerach za pośrednictwem sieci komputerowej.

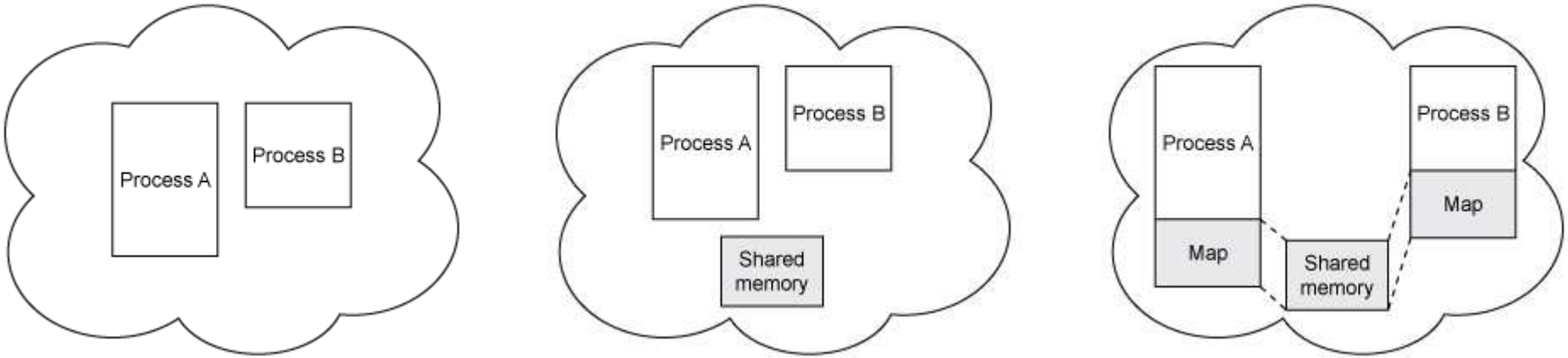
W odróżnieniu od potoków, służących do przesyłania strumieni bajtów, przez kolejki komunikatów jak i przez gniazdka przesyłane są kompletne komunikaty. Dzięki temu te mechanizmy mogą być używane w trybie wielu-wielu, a nie tylko 1-1 jak potoki.

Inna różnica pomiędzy gniazdkami a potokami i kolejkami komunikatów polega na nawiązywaniu połączenia. Potoki i kolejki komunikatów są zawsze tworzone jako gotowe łącza komunikacyjne — posiadające dwa końce do których uzyskują dostęp procesy. Gniazdko jest jakby tylko jednym końcem łącza komunikacyjnego. Przed rozpoczęciem komunikacji konieczne jest jego połączenie (lub powiązanie) z innym gniazdkiem stworzonym na tym samym komputerze, lub innym, połączonym z nim siecią. Ze względu na ten proces nawiązywania połączenia, gniazdka wykorzystywane są na ogół w trybie komunikacji klient-serwer, który jest trybem 1-wielu.

Komunikacja międzyprocesowa — obszary pamięci wspólnej

Komunikacja przez pamięć wspólną wymaga stworzenia obszaru pamięci wspólnej w systemie operacyjnym przez jeden z procesów, oraz **odwzorowania** tej pamięci do własnej przestrzeni adresowej wszystkich pragnących się komunikować procesów. Następnie komunikacja odbywa się przez zwykłe operacje odczytu/zapisu pamięci.

Komunikacja przez pamięć wspólną jest najszybszym rodzajem komunikacji międzyprocesowej, ponieważ dane nie są nigdzie przesyłane. W momencie ich utworzenia w lokalizacji źródłowej są od razu również dostępne w lokalizacji docelowej. Jednak ten rodzaj komunikacji wymaga **synchronizacji** za pomocą oddzielnych mechanizmów, takich jak muteksy albo blokady zapisu i odczytu.



Obrazki zapożyczone bez zezwolenia z:

`http://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/`

Komunikacja i synchronizacja — wyścigi i sekcja krytyczna

Gdy procesy pracujące współbieżnie, czyli równolegle, albo quasi-równolegle, operują na współdzielonych globalnych strukturach danych, może dojść do korupcji tych danych.

Rozważmy wieloprocesowy program obsługujący operacje na dowolnych kontach bankowych, spływające z oddziałów i bankomatów. Operacja może być wpłatą lub wypłatą, w zależności od wartości zmiennej `operacja` (dodatnia lub ujemna).

Proces 1:
```
// otrzymany numer konta
// oraz wartosc operacji
saldo[konto] = saldo[konto] + operacja;
```

Proces 2:
```
// otrzymany numer konta
// oraz wartosc operacji
saldo[konto] = saldo[konto] + operacja;
```

Zarówno równoległe jak i quasi równoległe wykonanie tych procesów może spowodować takie pofragmentowanie tych operacji, że jeśli jednocześnie pojawią się dwie operacje na tym samym koncie, to zostanie obliczona niepoprawna wartość salda. To zjawisko jest nazywane **wyścigami** (ang. *race condition*).

Fragment programu, w ramach którego może dojść do wyścigów, nazywamy **sekcją krytyczną**.

Ochrona sekcji krytycznej: muteksy

Dla ochrony sekcji krytycznej można użyć **muteksu** (ang. *mutex* = *mutual exclusion*). W danej chwili tylko jeden proces może posiadać blokadę muteksu, a drugi będzie musiał na nią czekać. Użycie blokady muteksu powoduje, że żadna z operacji na saldzie nie będzie mogła być przerwana przez drugą.

Proces 1:

```
// otrzymany numer konta
// oraz wartosc operacji
pthread_mutex_lock(transakcyjny);
saldo[konto] = saldo[konto] + operacja;
pthread_mutex_unlock(transakcyjny);
```

Proces 2:

```
// otrzymany numer konta
// oraz wartosc operacji
pthread_mutex_lock(transakcyjny);
saldo[konto] = saldo[konto] + operacja;
pthread_mutex_unlock(transakcyjny);
```

Wprowadzenie muteksu powoduje **serializację** sekcji krytycznej. Nie będzie ona nigdy wykonywana równolegle, lecz zawsze szeregowo. Na komputerze posiadającym wiele procesorów lub rdzeni z reguły powoduje to spowolnienie programu przez ograniczenie jego współbieżności. Dlatego zakres sekcji krytycznej zabezpieczonej muteksem powinien być minimalny, obejmując wyłącznie instrukcje mogące spowodować wyścigi.

Czym jest mutex — niepodzielność operacji

Mutex jest zasadniczo prostą koncepcją. W istocie jest jednobitową zmienną, normalnie o wartości 0, oznaczającej brak blokady. Założenie blokady muteksu nadaje mu wartość 1, być może uprzednio czekając na jego wyzerowanie, natomiast jej zwolnienie przywraca wartość 0.

Jednak operacje na muteksie muszą być zrealizowane niepodzielnie (*atomic*). Inaczej wątek, który sprawdziłby, że wartość muteksu wynosi 0 (wolny), i następnie ustawił go na 1 (zajęty), mógłby zostać wywłaszczony zaraz po sprawdzeniu wartości, i w wyniku wyścigów, po wznowieniu, zająć mutex w międzyczasie zajęty już przez inny wątek.

Jedynie system operacyjny może zapewnić nierozdzielne wykonanie sprawdzenia i zajęcia muteksu. Dlatego muteksy, podobnie jak inne mechanizmy synchronizacji, mają swój zestaw funkcji systemowych na nich operujących.

Wątki

Procesy stanowią podstawową jednostkę, z której tradycyjnie składa się większość systemów operacyjnych. Procesy posiadają dwie podstawowe cechy:

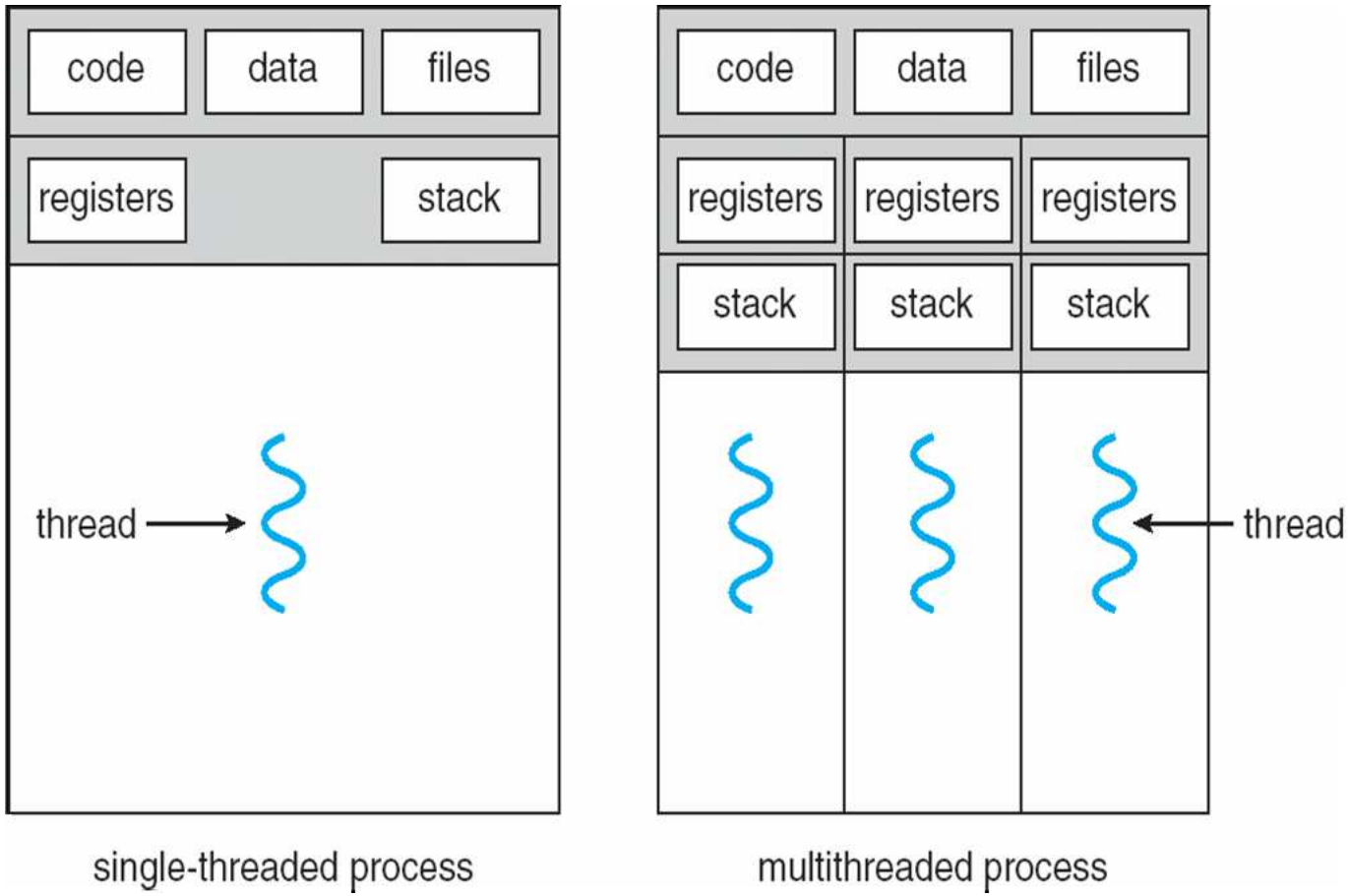
- kod programu załadowany do pamięci operacyjnej, wraz ze środowiskiem (zbiorem zmiennych posiadających wartości) i zasobami (np. zbiorem otwartych plików, i innymi przydzielonymi zasobami, np. portami komunikacyjnymi, itd.) przechowywanym przez jądro systemu,

- ślad wykonania (*execution trace*), tzn. ciąg wykonywanych instrukcji, wraz z niezbędnymi przy ich wykonywaniu konstrukcjami, np. licznikiem programu (PC), zbiorem rejestrów z zawartością, stosem wywoływanych procedur wraz z ich argumentami, itp.

W starszych systemach operacyjnych te dwie cechy procesu były integralnie ze sobą związane. W tych systemach proces miał zawsze kod programu (jeden) i jeden wykonujący go ślad wykonania. Nowsze systemy rozdzielają te cechy, pozwalając by jeden kod programu był jednocześnie (albo quasi-jednocześnie) wykonywany przez wiele śladów wykonania, nazywanych **wątkami** (*threads*).

Procesy wielowątkowe

Wątki z natury rzeczy są jednostkami podrzędnymi procesów, tzn. jeden proces może składać się z jednego lub więcej wątków.



Współdzielone dane wątków

Ponieważ wątki danego procesu współdzielą między sobą jego kod, zatem wszystkie zmienne globalne są ich wspólną własnością. Każdy z wątków może w dowolnym momencie odczytać lub nadpisać wartość zmiennej globalnej.

Jest to zarazem zaleta i wada. Zaleta polega na tym, że jest to najszybsza możliwa metoda komunikacji między wątkami, ponieważ wartości generowane przez wątek źródłowy są od razu dostępne w wątku docelowym. Jednak taki wspólny dostęp do danych wymaga **synchronizacji**, ponieważ wątek docelowy nie wie kiedy dokładnie dane zostały już wygenerowane i kiedy może je odczytać (wyobraźmy sobie, że dane te mają znaczną objętość, np. gdy jest to tablica lub struktura). Nie wiedząc tego, może odczytać stare dane, albo np. dane częściowo zmodyfikowane, które mogą w ogóle nie mieć sensu.

Podobnie synchronizacji wymaga sytuacja, gdy dwa wątki (lub więcej) chcą modyfikować dane. Bez synchronizacji takie operacje mogłyby doprowadzić do trwałego zapisania danych niespójnych.

Oczywiście synchronizacji nie wymagają operacje odczytu wspólnych danych.

Programowanie z użyciem wątków

Wątki pozwalają tworzyć programy działające współbieżnie. Współbieżność przydaje się wielokrotnie w programowaniu, na przykład:

- w różnego rodzaju serwerach obsługujących równocześnie wiele żądań,

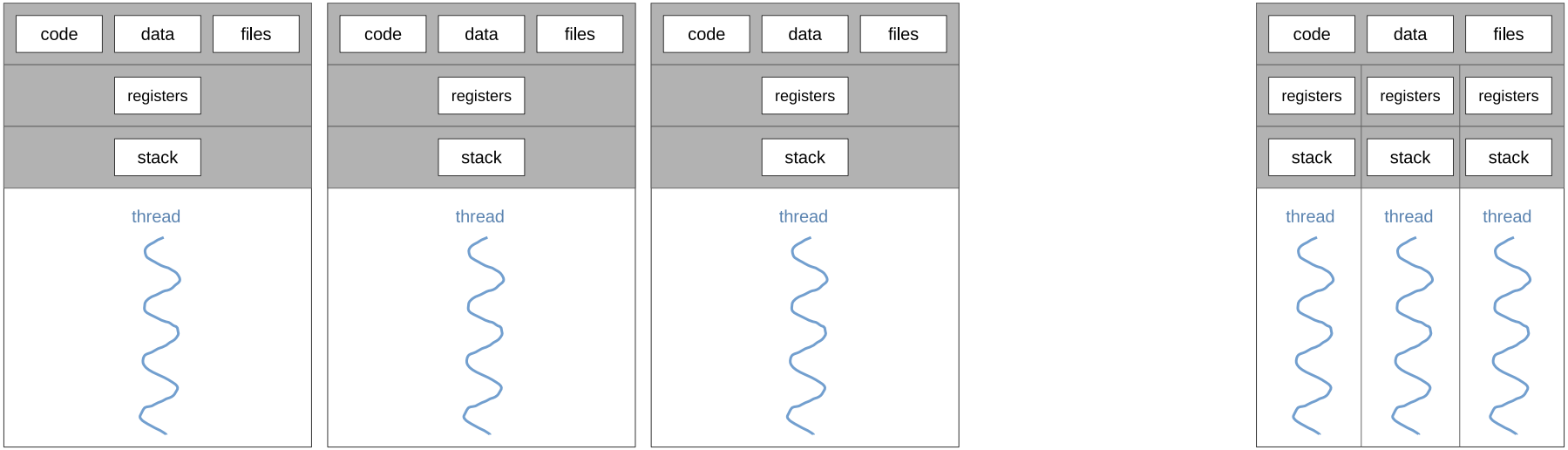
- w programach okienkowych, gdzie aktualizacja stanu okienek, komunikacja z użytkownikiem (odczyt zdarzeń z klawiatury i myszy), obliczenia związane z funkcjami danego programu, jak również operacje I/O na urządzeniach zewnętrznych, wszystkie mogą być realizowane współbieżnie i niezależnie od siebie,

- tworzenie współbieżnych aplikacji ma sens również w związku z coraz większą liczbą procesorów, a także rdzeni i wątków, dostępnych we współczesnych komputerach; program napisany współbieżnie może być wtedy wykonywany szybciej niż program napisany jednowątkowo.

W niektórych z tych sytuacji adekwatne rozwiązanie można uzyskać również za pomocą współbieżnie wykonujących się i komunikujących się procesów, ale w pewnych sytuacjach model wątków ma zdecydowaną przewagę.

Współbieżność z użyciem wątków i procesów

Zarówno procesy jak i wątki służą do realizacji obliczeń współbieżnych. Jeśli chcemy zrównoleglić obliczenia jakiegoś programu, to możemy to uzyskać przez tworzenie wielu procesów wykonujących ten program, bądź tworzenie przez ten program wielu wątków.



Tworzenie oddzielnych procesów jest poważniejszą operacją, zajmującą więcej czasu i zasobów (pamięci).

Jednak po ich utworzeniu, system operacyjny poddaje wątki szeregowaniu na równi z procesami w trzech stanach: wykonywany, gotowy (wykonywalny), i oczekujący.

Programowanie współbieżności z użyciem wątków i procesów

Programowanie współbieżności z wykorzystaniem wątków jest podobne jak z wykorzystaniem procesów, z następującymi różnicami:

- Wszystkie wątki danego procesu zawsze wykonują ten sam program, podczas gdy procesy są kompletnie oddzielnymi obiektami, każdy z własną przestrzenią adresową, mogącymi wykonywać dowolne programy, jak również przejść do wykonywania innego programu jedną z funkcji `exec*`.
- Wątki mają automatycznie dostępną szybką komunikację przez zmienne globalne, która jednak wymaga synchronizacji. Realizacja takiej komunikacji przez procesy wymaga stworzenia segmentu pamięci współdzielonej.
- Tworzenie wątków jest tanie, a więc znacznie szybsze niż tworzenie procesów. Ma to znaczenie np. w serwerach obsługujących bardzo wiele zgłoszeń na sekundę.
  - Czas potrzebny na pełne utworzenie nowego procesu (`fork()`) na współczesnych procesorach z zegarem 1.5÷2.8 GHz wynosi w granicach od 0.1 nawet do 2 ms. Czas potrzebny na utworzenie nowego wątku jest o około rząd wielkości mniejszy: 15 do 50 μs (`pthread_create()`).
  - Jednak należy pamiętać, że utworzone już wątki nie mają żadnej przewagi, i wykonują się tak samo szybko jak utworzone procesy.

Wątki jądra

Dotychczas omówione własności wątków dotyczą sytuacji, gdy wątki tworzone są na poziomie systemu operacyjnego, z wykorzystaniem jego mechanizmów. Takie wątki zwane są **wątkami jądra** (*kernel level threads*, KLT). Dla programów wykorzystujących te wątki:

- jądro zajmuje się obsługą wątków (tworzeniem, przełączaniem); trwa to typowo dłużej niż dla wątków użytkownika,
- w systemach wieloprocesorowych jądro ma możliwość jednoczesnego wykonywania wielu wątków jednej aplikacji na różnych procesorach,
- wątek czekający na coś w funkcji systemowej nie blokuje innych wątków.

Większość współczesnych systemów operacyjnych posiada zdolności tworzenia i obsługi wątków w jądrze systemu, włącznie z systemami w skali mikro przeznaczonymi do wykonywania na mikrokontrolerach.

Wątki użytkownika

Istnieje również możliwość tworzenia wątków całkowicie w ramach programu, z ich pełną obsługą przez dany program, bez udziału systemu operacyjnego. Takie wątki nazywamy **wątkami użytkownika** (*user level threads*, ULT). W takim przypadku:

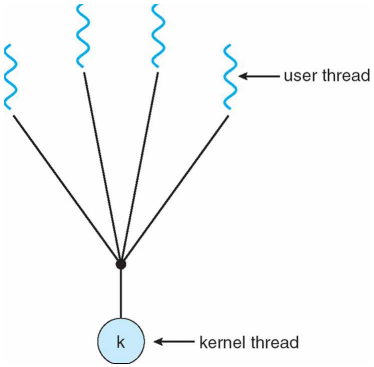
- jądro systemu nie ma świadomości istnienia wątków, ani nie zapewnia im wsparcia,
- możliwe jest ich zastosowanie nawet w systemach nieobsługujących wątków,
- tworzenie i przełączanie wątków jest szybkie, ponieważ wykonuje tylko tyle kodu ile wymaga, bez przełączania kontekstu jądra,
- nie ma możliwości wykorzystania wielu procesorów/rdzeni do wykonywania różnych wątków programu,
- jeśli jeden z wątków wywoła funkcję systemową powodującą blokowanie (czekanie), to pozostałe wątki nie będą mogły kontynuować pracy.

Obsługa wątków w programie jest skomplikowana, i często są do tego wykorzystywane odpowiednie biblioteki. Przykładem popularnej, przenośnej biblioteki wątków użytkownika dla systemów POSIX/ANSI-C jest GNU Pth (*GNU Portable Threads*).

Jednak ani czysty model wątków użytkownika ani czysty model wątków jądra nie zapewniają połączenia elastyczności programowania z efektywnością wykonywania programu. Dobrym modelem jest natomiast taka implementacja biblioteki wątków użytkownika, która odwzorowuje wątki użytkownika na wątki jądra.

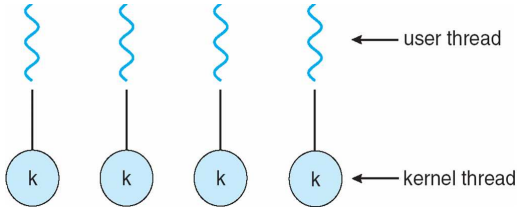
Odwzorowania wątków użytkownika na wątki jądra

Wiele-jeden



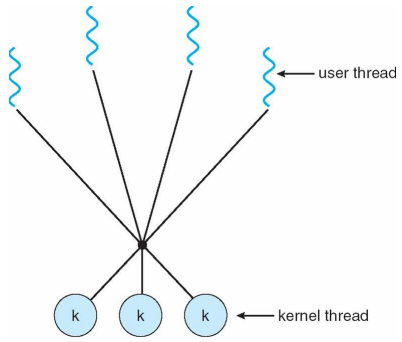
Ten model ma zastosowanie do czystej biblioteki wątków użytkownika, przenośnych bibliotekach wątków (np. GNU Portable Threads), w systemach jednoprocesorowych, lub nieobsługujących wątków.

Jeden-jeden



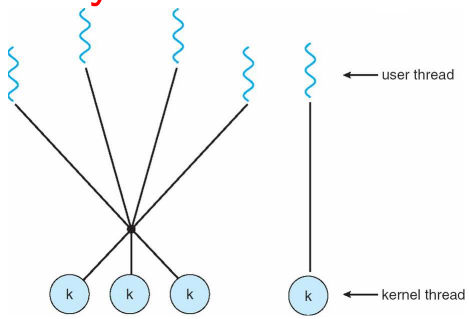
Ten model jest najczęściej obecnie stosowany, np. w systemach Solaris, Linux, Windows XP.

Wiele-wiele



Model stosowany w dawniejszych implementacjach wątków w systemach operacyjnych. Pozwala kontrolować rzeczywisty stopień współbieżności programów przez przydzielanie odpowiedniej liczby wątków jądra.

Model mieszany



Model podobny do modelu „wiele-wiele” ale pozwala na przydzielenie wątku jądra wybranym wątkom użytkownika. Stosowany np. w systemach: IRIX, HP-UX, Tru64 UNIX.

Synchronizacja wątków

- Ponieważ wątki są z definicji wykonywane współbieżnie i asynchronicznie, a także operują na współdzielonych globalnych strukturach danych, konieczne są mechanizmy dla zapewnienia wyłączności dostępu do tych współdzielonych globalnych struktur.

- Mechanizmy wzajemnej synchronizacji i blokowania ograniczają współbieżność, zatem należy stosować jak najdrobniejsze blokady. Przykładem ilustrującym tę zasadę jest minimalizacja sekcji krytycznej chronionej blokadą.

- Mechanizmy synchronizacji standardu POSIX:
  - muteksy (*mutual exclusion*)
  - blokady odczytu i zapisu (*read-write locks*)
  - semafory
  - zmienne warunkowe (*conditional variables*)
  - bariery

Synchronizacja wątków: blokady zapisu i odczytu

Poznaliśmy wcześniej pojęcie muteksu. **Blokady zapisu i odczytu** są innym mechanizmem synchronizacyjnym, będącym pewnym uogólnieniem muteksu.

Wyobraźmy sobie, że wśród napływających operacji bankowych większość stanowi sprawdzenie salda. Pobranie salda nie może być wykonywane równocześnie z jakąkolwiek operacją wpłaty ani wypłaty z tego rachunku, ponieważ w tym czasie saldo mogłoby być niepoprawnie odczytane. Jednak nic nie stoi na przeszkodzie, żeby sprawdzanie salda nie mogło się odbywać równocześnie z innymi operacjami sprawdzenia salda, nawet tego samego konta.

Aby to zrealizować, do zabezpieczenia obsługi transakcji zamiast muteksu należy użyć blokady zapisu i odczytu. Przy realizacji transakcji wpłaty lub wypłaty zakładana będzie blokada zapisu, co wyklucza wszelkie inne jednoczesne operacje. Jednak przy realizacji jakiejkolwiek operacji sprawdzenia salda, można użyć blokady odczytu, co powoduje, że inne operacje sprawdzenia salda mogą przebiegać równolegle, a wstrzymywane będą tylko operacje wpłaty i wypłaty.

Korzyść z użycia tego mechanizmu będzie znacząca tylko gdy operacji wymagających blokad odczytu będzie znacznie więcej niż tych wymagających blokad zapisu. W wielu systemach tak jest, np. w systemie sprawdzania haseł lub PINów będzie znacznie więcej operacji odczytu (sprawdzenia) niż operacji zapisu (zmiany) hasła/PINu.

Synchronizacja wątków: semafory

Semafory są innym uogólnieniem muteksów, pozwalającym zarządzać zasobami, mierzonymi w sztukach.[1] Zamiast w całości zajmować dany zasób, wątki zajmują pewną liczbę jednostek tego zasobu, w granicach aktualnie dostępnej puli. Wątek, który zażąda zajęcia większej liczby niż jest aktualnie dostępna musi czekać, aż inny wątek/inne wątki nie zwolnią dostatecznej liczby jednostek.

Analogią, ilustrującą zastosowanie semaforów może być wystawa (np. malarstwa), na którą ze względów bezpieczeństwa można wpuścić jednorazowo maksymalnie N osób. Semafor wstępnie ustawiamy na wartość N. Każda wchodząca osoba zgłasza żądanie jednostkowego zajęcia semafora, co dekrementuje jego licznik (wyrażający liczbę wolnych miejsc na wystawie). Po osiągnięciu maksimum (wyzerowaniu semafora), następne osoby zgłaszające żądania zajęcia semafora muszą czekać. Każda wychodząca osoba zwalnia (inkrementuje) semafor, co może spowodować wpuszczenie do środka jednej osoby oczekującej w żądaniu zajęcia semafora.

[1] Użycie określenia semafor w informatyce jest myląca. W kolejnictwie semafor jest stosowany do sygnalizacji zajętego toru. Informatycznym odpowiednikiem tego mechanizmu jest mutex. Semafory wprowadził w 1965 holenderski informatyk Edsger Dijkstra. Pojęcie muteksu, jako semafora binarnego, zostało zdefiniowane dużo później.

Muteks jest szczególnym przypadkiem semafora przyjmującego wyłącznie wartości 1 lub 0. Zatem semafory są uogólnieniem muteksów, ponieważ dostarczają większych możliwości. W rzeczywistości zastosowania muteksów i semaforów są całkiem inne.

O ile muteks jest jednobitowym rejestrem pozwalającym kontrolować **wyłączność dostępu** (*mutual exclusion*), jak w przypadku dostępu do segmentu pamięci wspólnej, to semafor jest licznikiem pozwalającym kontrolować przydział zasobów.

Modelowym przykładem zagadnienia, do którego ma zastosowanie semafor jest zagadnienie ograniczonego bufora, gdzie wiele procesów/wątków chce jednocześnie wykorzystywać bufor o pojemności pewnej liczby jednostek, każdorazowo zajmując część z tych jednostek. Gdy liczba pozostałych (wolnych) jednostek bufora jest zbyt mała aby kolejny proces mógł wykonać swoją pracę (albo w ogóle spada do zera), musi on czekać na zwolnienie odpowiedniej liczby jednostek.

Bariery — inny rodzaj synchronizacji

Bariera jest mechanizmem synchronizacji innego rodzaju. Zamiast zapewniać, że wątki nie będą wykonywały jednocześnie sekcji krytycznej programu, bariera służy do zapewnienia, że wszystkie wątki wystartują — lub przejdą do kolejnej sekcji programu — równocześnie (albo quasi-równocześnie).

Uwagi ogólne o współbieżności i synchronizacji

Współbieżność jest ważną cechą systemów informatycznych, lecz jej programowanie jest trudną sztuką. W szczególności *debugowanie* (czyli znajdowanie i poprawianie błędów w programach) jest trudne, ponieważ nie zawsze jest możliwe odtworzenie identycznych warunków w jakich dany błąd się objawił.

Typowym skutkiem błędów w programowaniu współbieżnym, poza wyścigami, są **zakleszczenia**. Powstają one w sytuacjach, gdy jeden proces/wątek zajął już jakiś zasób, i żąda innego, ale musi czekać, gdyż ten inny zasób jest już przydzielony innemu procesowi/wątkowi. Gdy ten drugi proces/wątek z kolei zażąda dostępu do zasobu już przydzielonego pierwszemu, powstaje zakleszczenie — sytuacja gdy oba zadania czekają, i nie ma możliwości by to na co czekają zostało kiedykolwiek zwolnione.

Innym zagadnieniem w programowaniu współbieżności jest **drobnoziarnistość** synchronizacji. Procesy można sychronizować na poziomie programu, tak, że gdy jeden pracuje, to inne muszą czekać. Gdy jednak proces wykonuje się długo, również długie jest oczekiwanie pozostałych. Gdyby zamiast tego, sychronizowana była tylko sekcja krytyczna w programach, procesy prawdopodobnie mogłyby wykonywać się poprawnie, minimalizując wzajemne oczekiwanie. Co więcej, sekcje krytyczne można czasami podzielić, być może dodając oddzielne muteksy, i oddzielnie synchronizując ich fragmenty, jeszcze bardziej zwiększając możliwość równoległego wykonania programów.