

# 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, semafony, itp.

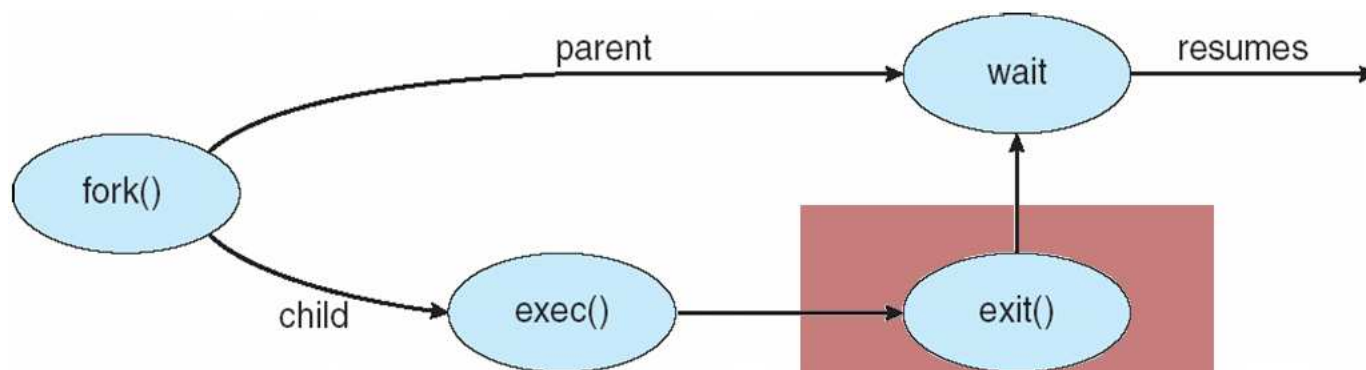
# Model procesów systemu UNIX (standard POSIX)

- 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).

# 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.

# Własności procesu i podprocesu

Dziedziczone (podproces posiada kopię atrybutu procesu):

- rzeczywisty i efektywny UID i GID, identyfikator grupy procesów (PGRP)
- kartoteka bieżąca i root
- środowisko
- maska tworzenia plików (umask)
- maska sygnałów i handlery
- ograniczenia zasobów

Wspólne (podproces współdzieli atrybut/zasób z procesem):

- terminal i sesja
- otwarte pliki (deskrytory)
- otwarte wspólne obszary pamięci

Różne:

- PID, PPID
- wartość zwracana z funkcji fork
- blokady plików nie są dziedziczone
- ustawiony alarm procesu nadrzędnego jest kasowany dla podprocesu
- zbiór wysłanych (ale nieodebranych) sygnałów jest kasowany dla podprocesu

Atrybuty procesu, które nie zmieniają się po exec:

- PID, PPID, UID(real), GID(real), PGID
- terminal, sesja
- ustawiony alarm z biegnącym czasem
- kartoteka bieżąca i root
- maska tworzenia plików (umask)
- maska sygnałów i oczekujące (nieodebrane) sygnały
- blokady plików
- ograniczenia zasobów



# 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. zapelnienia tablicy procesów).

# 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  $128 + \text{nr\_sygnału}$ .
- 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. zapelnienia 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 (choć przybranym) i wywołuje okresowo funkcję `wait()` aby umożliwić poprawne zakończenie swoich potomków.



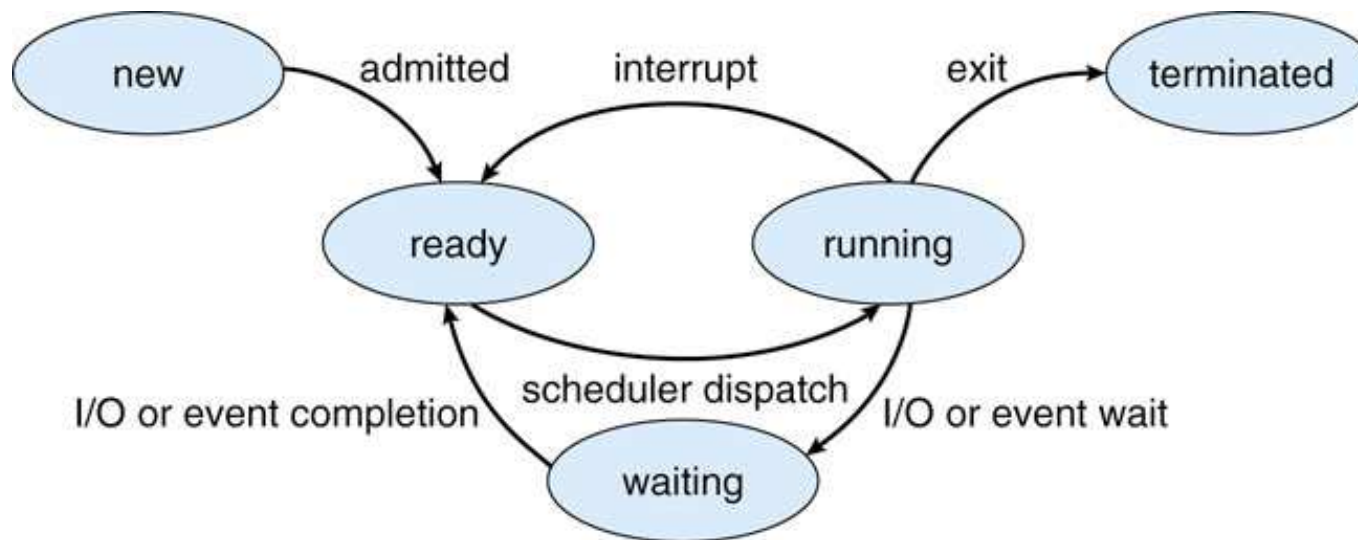
# Priorytety procesów

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”.**

# Stany procesów

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



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

# 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.

# 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.

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

# 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:

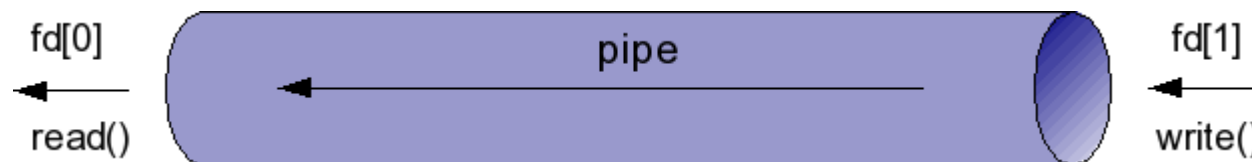
- zakończyć proces,
- wznowić proces od miejsca przerwania, jednak niektórych funkcji systemowych nie można wznowić dokładnie od miejsca przerwania,
- wznowić proces od miejsca przerwania z przekazaniem informacji przez zmienną globalną,
- wznowić proces od określonego punktu.



# Komunikacja międzyprocesowa — 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.



Dobłą 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 anonimowe i nazwane

Podstawowe pojęcie potoku określa strukturę dynamicznie tworzoną przez jądro systemu na żądanie danego procesu. Takie są identyfikowane tylko w ramach procesu, i z poziomu systemu nie można uzyskać do nich dostępu. Dostęp do takiego potoku ma tylko dany proces oraz jego procesy potomne — dzięki mechanizmowi dziedziczenia. Chcąc stworzyć grupę procesów komunikujących się przez potoki, należałoby stworzyć je wszystkie w ramach jednego procesu, który wcześniej utworzy odpowiedni potok/potoki. Jest to często niewygodne.

Istnieje podobny mechanizm **potoków nazwanych** (*named pipes*), które posiadają własności potoków, ale mają postać plików specjalnych, istniejących w dyskowym systemie plików. Dowolny proces może otworzyć taki plik do zapisu lub odczytu, i rozpocząć komunikację z innymi procesami.

Dla odróżnienia od potoków nazwanych potoki tworzone dynamicznie określa się czasami jako **anonimowe**.

# Komunikacja międzyprocesowa — potoki (cd.)

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.

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 1-1 o własnościach podobnych do potoków. Istnieje wiele wersji kolejek komunikatów z drobnymi różnicami.

W niektórych systemach kolejki komunikatów zostały zaimplementowane jako urządzenia sieciowe. Pozwalają one komunikować się procesom wykonującym się na różnych komputerach. Stanowią zatem mechanizm komunikacji międzyprocesowej dla systemów rozproszonych.

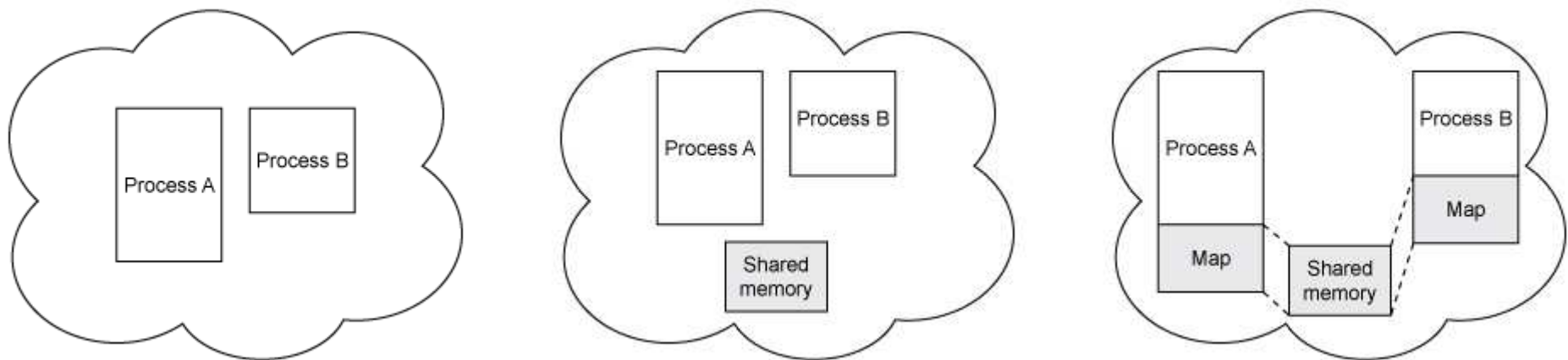
Gniazdko są jeszcze innym mechanizmem komunikacji dwukierunkowej typu 1-1 przeznaczonym zarówno do komunikacji międzyprocesowej w ramach jednego komputera, lub między procesami na różnych komputerach za pośrednictwem sieci komputerowej.

Zasadnicza 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 jednym końcem łącza komunikacyjnego. Przed rozpoczęciem komunikacji konieczne jest jego połączenie z innym gniazdkiem stworzonym na tym samym komputerze, lub innym, połączonym z nim siecią. Ze względu na ten proces nawiązywania połączenia, gniazdko wykorzystywane są na ogół w trybie komunikacji klient-serwer.

# Komunikacja międzyprocesowa — obszary pamięci wspólnej

Komunikacja przez pamięć wspólną jest najszybszym rodzajem komunikacji międzyprocesowej. Wymaga stworzenia obszaru pamięci wspólnej w systemie operacyjnym przez jeden z procesów, a następnie **odwzorowania** tej pamięci do własnej przestrzeni adresowej wszystkich pragnących się komunikować procesów.

Komunikacja w tym trybie 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/](http://www.ibm.com/developerworks/aix/library/au-spunix_sharedmemory/)

# Komunikacja międzyprocesowa — semafony i muteksy

W teorii semafor jest mechanizmem synchronizacyjnym, domyślnie kontrolującym dostęp lub przydział pewnego zasobu. Wartość semafora oznacza liczbę dostępnych jednostek zasobu. Określone są następujące operacje na semaforze:

**P(sem)** — oznacza zajęcie zasobu sygnalizowane zmniejszeniem wartości semafora o 1, a jeśli jego aktualna wartość jest 0 to oczekiwanie na jej zwiększenie,

**V(sem)** — oznacza zwolnienie zasobu sygnalizowane zwiększeniem wartości semafora o 1, a jeśli istnieje(a) proces(y) oczekujący(e) na semaforze to, zamiast zwiększać wartość semafora, wznowiany jest jeden z tych procesów.

Istotna jest niepodzielna realizacja każdej z tych operacji, tzn. każda z operacji P, V może albo zostać wykonana w całości, albo w ogóle nie zostać wykonana. Z tego powodu niemożliwa jest prywatna implementacja operacji semaforowych przy użyciu zmiennej globalnej przez proces pracujący w warunkach przełączania procesów. Operacje semaforowe realizowane są przez system operacyjny, który zapewnia ich niepodzielność.

Przydatnym przypadkiem szczególnym jest semafor binarny, który kontroluje dostęp do zasobu na zasadzie wyłączności. Wartość takiego semafora może wynosić 1 lub 0. Semafony binarne zwane są również muteksami (ang. *mutex* = mutual exclusion).

# Wątki

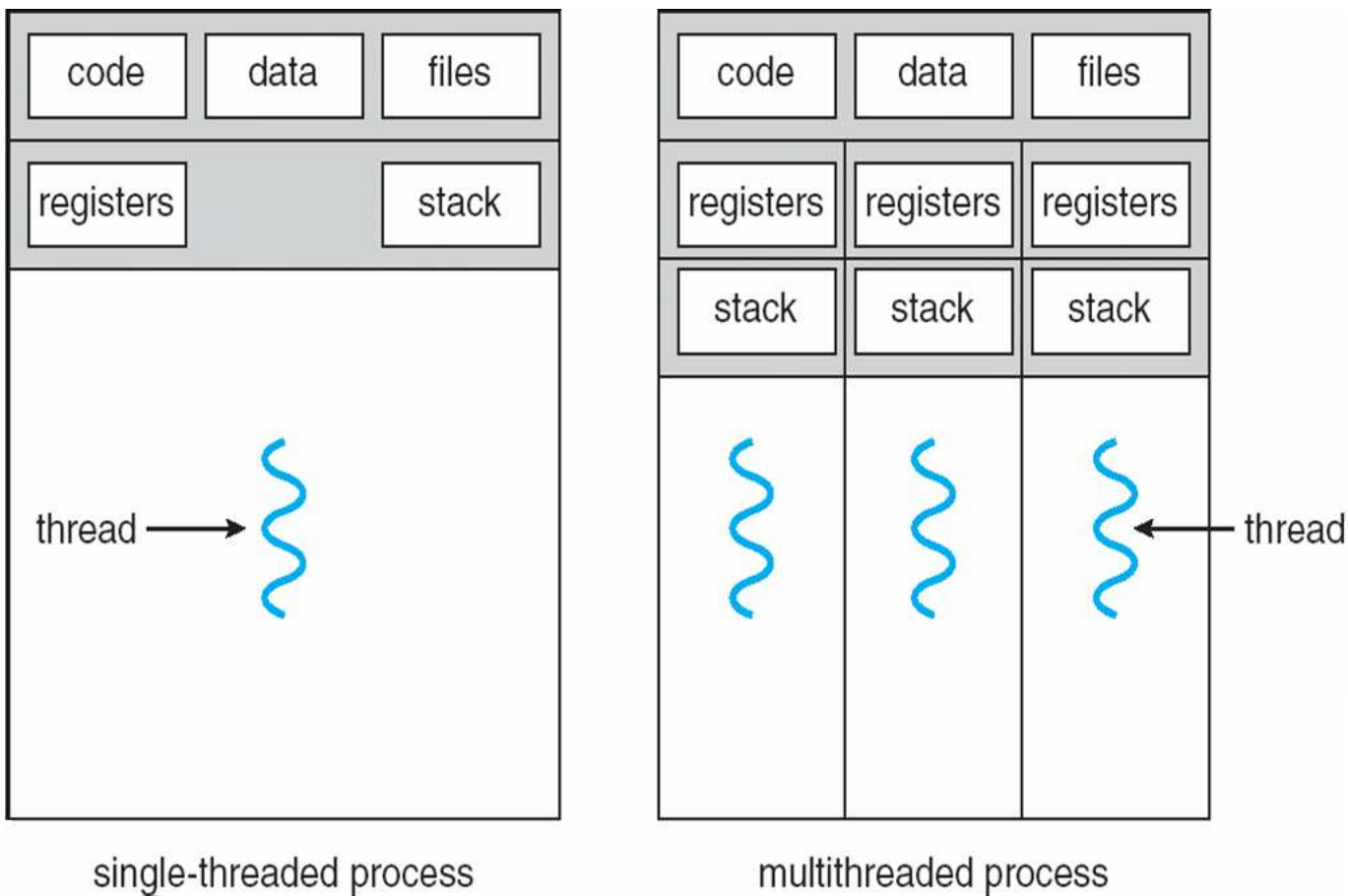
Procesy stanowią podstawową jednostkę, z której tradycyjnie składa się większość systemów operacyjnych. Procesy posiadają dwie podstawowe cechy:

- 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. Nowsze systemy pozostawiają pierwszą cechę procesom, ale drugą im odbierają na rzecz wątków (*threads*).

# Procesy wielowątkowe

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





# Wspólne 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, np. oczekująca na odczyt danych z urządzenia, może być realizowana asynchronicznie do pętli oczekującej na reakcję użytkownika; tego rodzaju oczekiwanie na kilka różnych rzeczy naraz stwarza trudność w programowaniu,
- tworzenie współbieżnych aplikacji ma sens również w związku z coraz większą liczbą procesorów (lub rdzeni) dostępnych we współczesnych komputerach; program napisany współbieżnie może być wtedy wykonywany szybciej niż program napisany jednowątkowo.

# Współbieżność z użyciem wątków i procesów

Programowanie z wątkami jest podobne do programowania procesów, z następującymi różnicami:

- Wszystkie wątki danego procesu wykonują ten sam program, podczas gdy procesy są kompletnie oddzielnymi obiektami, każdy z własną przestrzenią adresową, mogącymi wykonywać dowolne programy.
- Wątki mają automatycznie dostępną szybką komunikację przez zmienne globalne, która jednak wymaga synchronizacji.
- Tworzenie wątków jest tanie, a więc znacznie szybsze niż tworzenie procesów, co ma 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  $\mu$ s (`pthread_create()`).**
  - Jednak należy pamiętać, że **utworzone już wątki nie mają żadnej przewagi, i wykonują się tak samo szybko jak utworzone procesy.**

Pomijając interakcje między sobą, współbieżnie wykonujące się wątki zachowują się całkiem podobnie do współbieżnie wykonujących się procesów.

# Wątki użytkownika

Ze względu na sposób realizacji wątków w systemie operacyjnym można wyróżnić dwa rodzaje wątków: wątki użytkownika i wątki jądra. **Wątki użytkownika** (*user level threads*, ULT) są tworzone i w całości obsługiwane w przestrzeni użytkownika, zaimplementowane samodzielnie, lub za pomocą odpowiedniej biblioteki. Wtedy:

- 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,
- jeśli jeden z wątków wywoła funkcję systemową powodującą blokowanie (czekanie), to pozostałe wątki nie będą mogły kontynuować pracy,
- nie ma możliwości wykorzystania wielu procesorów/rdzeni do wykonywania różnych wątków programu.

# Wątki jądra

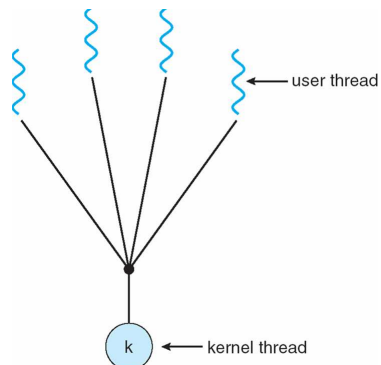
Większość współczesnych systemów operacyjnych realizuje wątki w jądrze systemu, zwane **wątkami jądra** (*kernel level threads, KLT*). Dla programów wykorzystujących te wątki:

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

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.

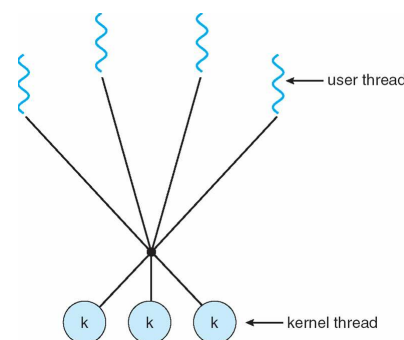
# Odwzorowania wątków użytkownika na wątki jądra

## Wiele-jeden



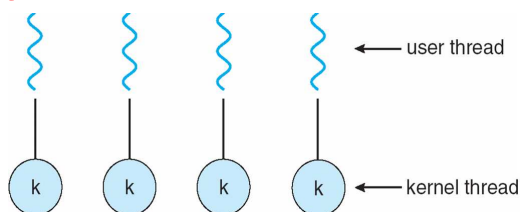
Ten model ma zastosowanie do czystej biblioteki wątków użytkownika, przenośnych bibliotekach wątków (np. GNU Portable Threads), w systemach jednoprosesorowych, lub nieobsługujących wątków.

## Wiele-wiele



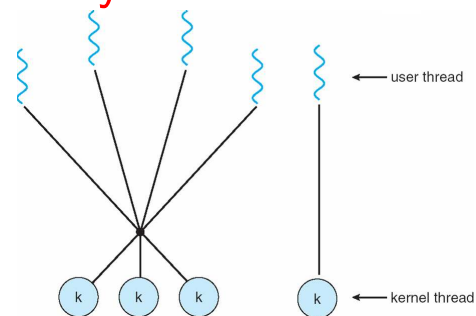
Model stosowany w dawniejszych implementacjach wątków w systemach operacyjnych. Pozwala kontrolować rzeczywisty stopień współbieżności programów przez przydzielanie odpowiedniej liczby wątków jądra.

## Jeden-jeden



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

## 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 wyłączości dostępu.
- Mechanizmy wzajemnej synchronizacji i blokowania ograniczają współbieżność, zatem należy stosować jak najdrobniejsze blokady.
- Mechanizmy synchronizacji standardu POSIX:
  - muteksy (*mutual exclusion*)
  - blokady odczytu i zapisu (*read-write locks*)
  - semafony
  - zmienne warunkowe (*conditional variables*)
  - bariery

# Synchronizacja wątków: muteksy

Muteksy służą do wprowadzenia ochrony fragmentu programu, zwanego **sekcją krytyczną**, która może wprowadzić błędy jeśli będzie wykonana równocześnie przez różne procesy/wątki. Rozważmy wielowątkowy program obsługujący operacje na dowolnych kontach bankowych, spływające z oddziałów i bankomatów:

Wątek 1:

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

Wątek 2:

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

Tak równoległe jak i quasi równoległe wykonanie tych wątków może spowodować takie pofragmentowanie tych operacji, że jeśli jednocześnie pojawią się dwie operacje na tym samym koncie, to zostanie obliczona niepoprawna wartość salda. Zjawisko takie jest nazywane **wyścigami**. Można mu zapobiec za pomocą muteksu:

Wątek 1:

```
pthread_mutex_lock(transakcyjny);  
saldo[konto] += operacja;  
pthread_mutex_unlock(transakcyjny);
```

Wątek 2:

```
pthread_mutex_lock(transakcyjny);  
saldo[konto] += operacja;  
pthread_mutex_unlock(transakcyjny);
```

Założenie blokady muteksu powoduje, że żadna z tych operacji nie może być przerwana przez drugą. W danej chwili tylko jedna może posiadać blokadę muteksu, a druga będzie musiała na nią czekać.



# Synchronizacja wątków: blokady zapisu i odczytu

**Blokady zapisu i odczytu** są innym rodzajem mechanizmu synchronizacyjnego, 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, 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 jakiegokolwiek operacji sprawdzenia salda, należy 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 wtedy 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: semafor

**Semafor** są innym uogólnieniem muteksów, pozwalającym zarządzać zasobami, mierzonymi w sztukach.<sup>1</sup> 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 aktualnie dostępna musi czekać, aż inny wątek/inne wątki nie zwolnią dostatecznej liczby,

Dobłą analogią, ilustrującą zastosowanie semaforów może być wystawa (np. malarstwa), na którą ze względów bezpieczeństwa można wpuścić jednorazowo maksymalnie  $N$  osób. Semafor jest wstępnie ustawiony na wartość  $N$ . Każda wchodząca osoba zgłasza żądanie jednostkowego zajęcia semafora, co dekrementuje jego licznik (symbolizujący liczbę wolnych miejsc na wystawie). Po osiągnięciu maksimum (wyzerowaniu semafora), następne osoby po zgłoszeniu żądania zajęcia semafora muszą czekać. Każda wychodząca osoba zwalnia (inkrementuje) semafor, co może spowodować wpuszczenie jednej osoby do środka, o ile czeka w żądaniu zajęcia semafora.

---

<sup>1</sup> Jak to było już wyjaśnione wcześniej, nazwa semafor jest myląca. W kolejnictwie semafor jest stosowany do sygnalizacji zajętego toru. Informatycznym odpowiednikiem tego mechanizmu jest mutex. Semafor wprowadził w 1965 holenderski informatyk Edsger Dijkstra. Pojęcie muteksu, jako semafora binarnego, zostało zdefiniowane dużo później.