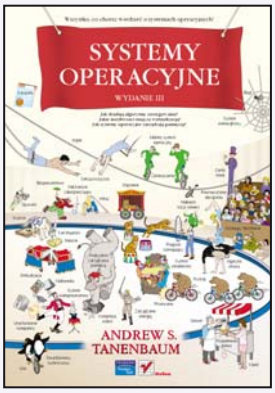
Systemy operacyjne. Wydanie III



Autor: Andrew S. Tanenbaum
Tłumaczenie: Radosław Meryk, Mikołaj Szczepaniak
ISBN: 978-83-246-2311-2
Tytuł oryginału: Modern Operating Systems (3rd Edition)
Format: 172×245, stron: 1288

Wszystko, co chcesz wiedzieć o systemach operacyjnych!

- Jak działają algorytmy szeregowania?
- Jakie możliwości stoją za wirtualizacją?
- Jak systemy operacyjne zarządzają pamięcią?

Ta książka to aktualne wydanie światowego bestsellera, będącego kompletnym źródłem wiedzy na temat współczesnych systemów operacyjnych. Autor tego podręcznika – Andrew S. Tanenbaum – przez wiele lat projektował trzy systemy operacyjne lub współuczestniczył w ich projektowaniu, dzięki czemu może dzielić się swą ogromną wiedzą i doświadczeniem praktyka. W tej publikacji szczególny nacisk kładzie on na możliwie szczegółową prezentację takich aspektów systemów, jak procesy, wątki, zarządzanie pamięcią, systemy plików, operacje wejścia-wyjścia, zakleszczenia, projektowanie interfejsu, multimedia, dylematy związane z wydajnością czy najnowsze trendy w projektach systemów operacyjnych. W trakcie lektury poznasz dokładnie systemy operacyjne Windows, Linux oraz Symbian. Pozycja ta stanowi niezastąpione kompendium wiedzy na temat systemów operacyjnych, zarówno dla studentów informatyki, jak i wszystkich pasjonatów komputera.

- Rys historyczny systemów operacyjnych
- Sprzęt komputerowy – przegląd komponentów
- Rodzaje systemów operacyjnych
- Struktura systemów operacyjnych
- Obsługa wywołań systemowych
- Zarządzanie procesami i wątkami oraz komunikacja między nimi
- Szeregowanie zadań
- Zarządzanie pamięcią
- Obsługa systemów plików
- Urządzenia wejścia-wyjścia
- Metody zarządzania energią
- Rozwiązywanie problemu zakleszczeń
- Charakterystyka multimedialnych systemów operacyjnych
- Obsługa systemów wieloprocesorowych
- Wirtualizacja
- Bezpieczeństwo danych na poziomie systemu operacyjnego
- Zagrożenia ze strony złośliwego oprogramowania
- System Linux – historia, budowa, działanie
- System Windows Vista – studium przypadku
- System Symbian – przeznaczenie, działanie, obsługa

Kompendium wiedzy o współczesnych systemach operacyjnych!

SPIS TREŚCI

10 Pierwsze studium przypadku: Linux 851

2

PROCESY I WĄTKI

Zanim rozpocznie się szczegółowe studium tego, w jaki sposób systemy operacyjne są zaprojektowane i skonstruowane, warto przypomnieć, że kluczowym pojęciem we wszystkich systemach operacyjnych jest **proces**: abstrakcja działającego programu. Wszystkie pozostałe elementy systemu operacyjnego bazują na pojęciu procesu, dlatego jest bardzo ważne, aby projektant systemu operacyjnego (a także student) jak najszybciej dobrze zapoznał się z pojęciem procesu.

Procesy to jedne z najstarszych i najważniejszych abstrakcji występujących w systemach operacyjnych. Zapewniają one możliwość wykonywania (pseudo-) współbieżnych operacji nawet wtedy, gdy dostępny jest tylko jeden procesor. Przekształcają one pojedynczy procesor CPU w wiele wirtualnych procesorów. Bez abstrakcji procesów istnienie współczesnej techniki komputerowej byłoby niemożliwe. W niniejszym rozdziale przedstawimy szczegółowe informacje na temat tego, czym są procesy oraz ich pierwsi kuzynowie — wątki.

2.1. PROCESY

Wszystkie nowoczesne komputery bardzo często wykonują wiele operacji jednocześnie. Osoby przyzwyczajone do pracy z komputerami osobistymi mogą nie być do końca świadome tego faktu, zatem kilka przykładów pozwoli przybliżyć to zagadnienie. Na początek rozważmy serwer WWW. Żądania stron WWW mogą nadchodzić z wielu miejsc. Kiedy przychodzi żądanie, serwer sprawdza, czy potrzebna strona znajduje się w pamięci podręcznej. Jeśli tak, jest przesyłana do klienta. Jeśli nie, inicjowane jest żądanie dyskowe w celu jej pobrania. Jednak z perspektywy procesora

obsługa żądań dyskowych zajmuje wieczność. W czasie oczekiwania na zakończenie obsługi żądania na dysk może nadejść wiele kolejnych żądań. Jeśli w systemie jest wiele dysków niektóre z żądań może być skierowanych na inne dyski na długo przed obsłużeniem pierwszego żądania. Oczywiste, że potrzebny jest sposób zamodelowania i zarządzania tą współbieżnością. Do tego celu można wykorzystać procesy (a w szczególności wątki).

Teraz rozważmy komputer osobisty użytkownika. Podczas rozruchu systemu następuje start wielu procesów. Często użytkownik nie jest tego świadomy. Na przykład może być uruchomiony proces oczekujący na wchodzące wiadomości e-mail. Inny uruchomiony proces może działać w imieniu programu antywirusowego i sprawdzać okresowo, czy są dostępne jakieś nowe definicje wirusów. Dodatkowo mogą działać jawne procesy użytkownika — na przykład drukujące pliki lub wypalające płytę CD — podczas gdy użytkownik przegląda strony WWW. Działaniami tymi trzeba zarządzać. W tym przypadku bardzo przydaje się system z obsługą wieloprogramowości, obsługujący wiele procesów jednocześnie.

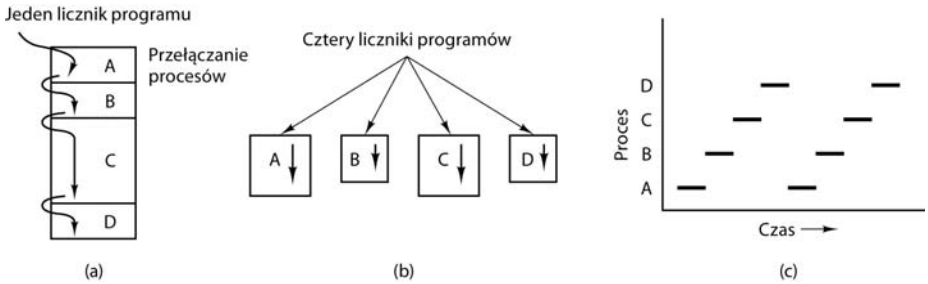
W każdym systemie wieloprogramowym procesor szybko przełącza się pomiędzy procesami, poświęcając każdemu z nich po kolei dziesiątki bądź setki milisekund. Chociaż ściśle rzecz biorąc w dowolnym momencie procesor realizuje tylko jeden proces, w ciągu sekundy może obsłużyć ich wiele, co daje iluzję współbieżności. Czasami w tym kontekście mówi się o **pseudowspółbieżności**, dla odróżnienia jej od rzeczywistej, sprzętowej współbieżności systemów **wieloprocesorowych** (wyposażonych w dwa procesory współdzielące tę samą fizyczną pamięć lub większą liczbę takich procesorów). Śledzenie wielu równoległych działań jest bardzo trudne. Z tego powodu projektanci systemów operacyjnych w ciągu wielu lat opracowali model pojęciowy (procesów sekwencyjnych), które ułatwiają obsługę współbieżności. Ten model, jego zastosowania oraz kilka innych konsekwencji stanowią temat niniejszego rozdziału.

2.1.1. Model procesów

W tym modelu całe oprogramowanie możliwe do uruchomienia w komputerze — czasami włącznie z systemem operacyjnym — jest zorganizowane w postaci zbioru **procesów sekwencyjnych** (lub w skrócie **procesów**). Proces jest egzemplarzem uruchomionego programu włącznie z bieżącymi wartościami licznika programu, rejestrów i zmiennych. Pojęciowo każdy proces ma własny wirtualny procesor CPU. Oczywiście w rzeczywistości procesor fizyczny przełącza się od procesu do procesu. Aby jednak zrozumieć system, znacznie łatwiej jest myśleć o kolekcji procesów działających (pseudo) współbieżnie, niż próbować śledzić to, jak procesor przełącza się od programu do programu. To szybkie przełączanie się procesora jest określane jako **wieloprogramowość**, o czy mówiliśmy w rozdziale 1.

Na rysunku 2.1(a) pokazaliśmy komputer, w którym w pamięci działają w trybie wieloprogramowym cztery programy. Na rysunku 2.1(b) widać cztery procesy —

każdy ma własny przepływ sterowania (tzn. własny logiczny licznik programu) i każdy działa niezależnie od pozostałych. Oczywiście jest tylko jeden fizyczny licznik programu, dlatego kiedy działa wybrany proces, jego logiczny licznik programu jest kopiowany do rzeczywistego licznika programu. Kiedy proces kończy działanie (na pewien czas), jego fizyczny licznik programu jest zapisywany w logicznym liczniku programu umieszczonym w pamięci. Na rysunku 2.1(c) widać, że w dłuższym przedziale czasu nastąpił postęp we wszystkich procesach, jednak w danym momencie działa tylko jeden proces.



Rysunek 2.1. (a) Cztery programy uruchomione w trybie wieloprogramowym; (b) pojęciowy model czterech niezależnych od siebie procesów sekwencyjnych; (c) w wybranym momencie jest aktywny tylko jeden program

W tym rozdziale założymy, że jest tylko jeden procesor CPU. Coraz częściej jednak takie założenie okazuje się nieprawdziwe. Nowe układy często są wielordzeniowe — mają dwa procesory, cztery lub większą ich liczbę. O układach wielordzeniowych i systemach wieloprocesorowych powiemy więcej w rozdziale 8. Na razie będzie prościej, jeśli przyjmiemy, że maszyna wykorzystuje jednorazowo tylko jeden procesor. Jeśli zatem mówimy, że procesor w danym momencie może wykonywać tylko jeden proces, to jeśli zawiera dwa rdzenie (lub dwa procesory), na każdym z nich w określonym momencie może działać jeden proces.

Ze względu na szybkie przełączanie się procesora pomiędzy procesami, tempo, w jakim proces wykonuje obliczenia, nie jest jednolite, a nawet trudne do powtórzenia w przypadku ponownego uruchomienia tego samego procesu. A zatem nie można programować procesów z wbudowanymi założeniami dotyczącymi czasu działania. Rozważmy dla przykładu proces wejścia-wyjścia, który uruchamia taśmę streamera w celu odtworzenia plików z kopii zapasowej, następnie wykonuje 10 000 iteracji pustej pętli w celu rozpędzenia streamera do właściwej szybkości i, na koniec, wydaje polecenie przeczytania pierwszego rekordu. Jeśli procesor zdecyduje się na przełączenie do innego procesu podczas trwania pętli, w której streamer się rozpędza, proces obsługi taśmy nie będzie mógł ponownie się uruchomić do momentu, kiedy pierwszy rekord znajdzie się za głowicą czytającą. Kiedy proces obowiązują tak ścisłe wymagania działania w czasie rzeczywistym — to znaczy określone zdarzenia *muszą* wystąpić w ciągu określonej liczby milisekund — trzeba przedsięwziąć

specjalne środki w celu zapewnienia, że tak się stanie. Zazwyczaj jednak większości procesów nie dotyczą ograniczenia wieloprogramowości procesora czy też względne szybkości działania różnych procesów.

Różnica pomiędzy procesem a programem jest subtelna, ale ma kluczowe znaczenie. Do wyjaśnienia tej różnicy posłużymy się analogią. Załóżmy, że pewien informatyk o zdolnościach kulinarnych piecze urodzinowy tort dla swojej córki. Ma do dyspozycji przepis na tort urodzinowy oraz kuchnię dobrze wyposażoną we wszystkie składniki: mąkę, jajka, cukier, aromat waniliowy itp. W tym przykładzie przepis spełnia rolę programu (tzn. algorytmu wyrażonego w odpowiedniej notacji), informatyk jest procesorem (CPU), natomiast składniki ciasta odgrywają rolę danych wejściowych. Proces jest operacją, w której informatyk czyta przepis, dodaje składniki i piecze ciasto.

Wyobraźmy sobie teraz, że z krzykiem wbiega syn informatyka i mówi, że użądliła go pszczoła. Informatyk zapamiętuje, w którym miejscu przepisu się znajdował (zapisuje bieżący stan procesu), bierze książkę o pierwszej pomocy i zaczyna postępować zgodnie z zapisanymi w niej wskazówkami. W tym momencie widzimy przełączenie się procesora z jednego procesu (pieczenie) do procesu o wyższym priorytecie (udzielanie pomocy medycznej). Przy czym każdy z procesów ma inny program (przepis na ciasto, książka pierwszej pomocy medycznej). Kiedy informatyk poradzi sobie z opatrzeniem użądlenia, powraca do pieczenia ciasta i kontynuuje od miejsca, w którym skończył.

Kluczowe znaczenie ma uświadomienie sobie, że proces jest pewnym działaniem. Charakteryzuje się programem, wejściem, wyjściem i stanem. Jeden procesor może być współdzielony przez kilka procesów za pomocą algorytmu szeregowania. Algorytm ten decyduje, w którym zatrzymać pracę nad jednym programem i rozpocząć obsługę innego.

Warto zwrócić uwagę na to, że jeśli program uruchomi się dwa razy, liczy się jako dwa procesy. Na przykład często istnieje możliwość dwukrotnego uruchomienia edytora tekstu lub jednoczesnego drukowania dwóch plików, jeśli system komputerowy jest wyposażony w dwie drukarki. Fakt, że dwa działające procesy korzystają z tego samego programu, nie ma znaczenia — są to oddzielne procesy. System operacyjny może mieć możliwość współdzielenia kodu pomiędzy nimi w taki sposób, że w pamięci znajduje się jedna kopia. Jest to jednak szczegół techniczny, który nie zmienia faktu działania dwóch procesów.

2.1.2. Tworzenie procesów

Systemy operacyjne wymagają sposobu tworzenia procesów. W bardzo prostych systemach lub w systemach zaprojektowanych do uruchamiania tylko jednej aplikacji (na przykład kontrolera w kuchence mikrofalowej), bywa możliwe zainicjowanie wszystkich potrzebnych procesów natychmiast po uruchomieniu systemu. Jednak

w systemach ogólnego przeznaczenia potrzebny jest sposób tworzenia i niszczenia procesów podczas ich działania. W tym punkcie przyjrzymy się niektórym spośród tych mechanizmów.

Są cztery podstawowe zdarzenia, które powodują tworzenie procesów:

1. Inicjalizacja systemu.
2. Uruchomienie wywołania systemowego tworzącego proces przez działający proces.
3. Żądanie użytkownika utworzenia nowego procesu.
4. Zainicjowanie zadania wsadowego.

W momencie rozruchu systemu operacyjnego zwykle tworzonych jest kilka procesów. Niektóre z nich są procesami pierwszego planu — to znaczy są to procesy, które komunikują się z użytkownikami i wykonują dla nich pracę. Inne są procesami drugoplanowymi, które nie są powiązane z określonym użytkownikiem, ale spełniają pewną specyficzną funkcję. Na przykład jeden proces drugoplanowy może być zaprojektowany do akceptacji wchodzących wiadomości e-mail. Taki proces może być uśpiony przez większość dnia i nagle się uaktywnić, kiedy nadchodzi wiadomość e-mail. Inny proces drugoplanowy może być zaprojektowany do akceptacji wchodzących żądań stron WWW zapisanych na serwerze. Proces ten budzi się w momencie odebrania żądania strony WWW w celu jego obsłużenia. Procesy działające na drugim planie, które są przeznaczone do obsługi pewnych operacji, takich jak odbiór wiadomości e-mail, serwowanie stron WWW, aktualności, drukowanie itp., są określane jako **demony**. W dużych systemach zwykle działają dziesiątki takich procesów. W systemie UNIX, aby wyświetlić listę działających procesów, można skorzystać z programu `ps`. W systemie Windows można skorzystać z menedżera zadań.

Procesy mogą być tworzone nie tylko w czasie rozruchu, ale także później. Działający proces często wydaje wywołanie systemowe w celu utworzenia jednego lub kilku nowych procesów mających pomóc w realizacji zadania. Tworzenie nowych procesów jest szczególnie przydatne, kiedy pracę do wykonania można łatwo sformułować w kontekście kilku związanych ze sobą, ale poza tym niezależnych, współdziałających ze sobą procesów. Jeśli na przykład przez sieć jest pobierana duża ilość danych w celu ich późniejszego przetwarzania, to można utworzyć jeden proces, który pobiera dane i umieszcza je we współdzielonym buforze, oraz drugi proces, który usuwa dane z bufora i je przetwarza. W systemie wieloprocesorowym, w którym każdy z procesów może działać na innym procesorze, zadanie może być wykonane w krótszym czasie.

W systemach interaktywnych użytkownicy mogą uruchomić program poprzez wpisanie polecenia lub kliknięcie (ewentualnie dwukrotne kliknięcie) ikony. Wykonanie dowolnej z tych operacji inicjuje nowy proces i uruchamia w nim wskazany

program. W systemach uniksowych bazujących na systemie X Window nowy proces przejmuje okno, w którym został uruchomiony. W systemie Microsoft Windows po uruchomieniu procesu nie ma on przypisanego okna. Może on jednak stworzyć jedno (lub więcej) okien i większość systemów to robi. W obydwu systemach użytkownicy mają możliwość jednoczesnego otwarcia wielu okien, w których działają jakieś procesy. Za pomocą myszy użytkownik może wybrać okno i komunikować się z procesem, na przykład podawać dane wejściowe wtedy, kiedy są potrzebne.

Ostatnia sytuacja, w której są tworzone procesy, dotyczy tylko systemów wsadowych w dużych komputerach mainframe. W systemach tego typu użytkownicy mogą przesyłać do systemu zadania wsadowe (czasami zdalnie). Kiedy system operacyjny zdecyduje, że ma zasoby wystarczające do uruchomienia innego zadania, tworzy nowy proces i uruchamia następne zadanie z kolejki.

Z technicznego punktu widzenia we wszystkich tych sytuacjach proces tworzy się poprzez zlecenie istniejącemu procesowi wykonania wywołania systemowego tworzenia procesów. Może to być działający proces użytkownika, proces systemowy, wywołany z klawiatury lub za pomocą myszy, albo proces zarządzania zadaniami systemowymi. Proces ten wykonuje wywołanie systemowe tworzące nowy proces. To wywołanie systemowe zleca systemowi operacyjnemu utworzenie nowego procesu i wskazuje, w sposób pośredni lub bezpośredni, jaki program należy w nim uruchomić.

W systemie UNIX istnieje tylko jedno wywołanie systemowe do utworzenia nowego procesu: `fork`. Wywołanie to tworzy dokładny klon procesu wywołującego. Po wykonaniu instrukcji `fork` procesy rodzic i dziecko mają ten sam obraz pamięci, te same zmienne środowiskowe oraz te same otwarte pliki. Po prostu są identyczne. Wtedy zazwyczaj proces-dziecko uruchamia wywołanie `execve` lub podobne wywołanie systemowe w celu zmiany obrazu pamięci i uruchomienia nowego programu. Kiedy użytkownik wpisze polecenie w środowisku powłoki, na przykład `sort`, powłoka najpierw tworzy proces-dziecko za pomocą wywołania `fork`, a następnie proces-dziecko wykonuje polecenie `sort`. Powodem, dla którego dokonuje się ten dwuetapowy proces, jest umożliwienie procesowi-dziecku manipulowania deskryptorami plików po wykonaniu wywołania `fork`, ale przed wywołaniem `execve` w celu przekierowania standardowego wejścia, standardowego wyjścia oraz standardowego urządzenia błędów.

Dla odróżnienia w systemie Windows jedna funkcja interfejsu Win32 — `Create ↪Process` — jest odpowiedzialna zarówno za utworzenie procesu, jak i załadowanie odpowiedniego programu do nowego procesu. Wywołanie to ma 10 parametrów. Są to program do uruchomienia, parametry wiersza polecenia przekazywane do programu, różne atrybuty zabezpieczeń, bity decydujące o tym, czy otwarte pliki będą dziedziczone, informacje dotyczące priorytetów, specyfikacja okna, jakie ma być utworzone dla procesu (jeśli proces ma mieć okno), oraz wskaźnik do struktury, w której są zwracane do procesu wywołującego informacje o nowo tworzonym procesie. Oprócz wywołania `CreateProcess` interfejs Win32 zawiera około 100 innych funkcji do zarządzania i synchronizowania procesów oraz wykonywania powiązanych z tym operacji.

Zarówno w systemie UNIX, jak i Windows po utworzeniu procesu rodzic i dziecko mają osobne przestrzenie adresowe. Jeśli dowolny z procesów zmieni słowo w swojej przestrzeni adresowej, zmiana nie jest widoczna dla drugiego procesu. W systemie UNIX początkowa przestrzeń adresowa procesu-dziecka jest *kopią* przestrzeni adresowej procesu-rodzica. Są to jednak całkowicie odrębne przestrzenie adresowe. Zapisywalna pamięć nie jest współdzielona pomiędzy procesami (w niektórych implementacjach Uniksa tekst programu jest współdzielony pomiędzy procesami rodzica i dziecka, ponieważ nie może on być modyfikowany). Nowo utworzony proces może jednak współdzielić niektóre inne zasoby procesu swojego twórcy — na przykład otwarte pliki. W systemie Windows przestrzenie adresowe procesów rodzica i dziecka od samego początku są różne.

2.1.3. Kończenie działania procesów

Po utworzeniu proces zaczyna działanie i wykonuje swoje zadania. Nic jednak nie trwa wiecznie — nawet procesy. Prędzej czy później nowy proces zakończy swoje działanie. Zwykle dzieje się to z powodu jednego z poniższych warunków:

1. Normalne zakończenie pracy (dobrowolnie).
2. Zakończenie pracy w wyniku błędu (dobrowolnie).
3. Błąd krytyczny (przymusowo).
4. Zniszczenie przez inny proces (przymusowo).

Większość procesów kończy działanie dlatego, że wykonały swoją pracę. Kiedy kompilator skompiluje program, wykonuje wywołanie systemowe, które informuje system operacyjny o zakończeniu pracy. Tym wywołaniem jest `exit` w systemie UNIX oraz `ExitProcess` w systemie Windows. W programach wyposażonych w interfejs ekranowy zwykle są mechanizmy pozwalające na dobrowolne zakończenie działania. W edytorach tekstu, przeglądarkach internetowych i podobnych im programach zawsze jest ikona lub polecenie menu, które użytkownik może kliknąć, aby zlecić procesowi usunięcie otwartych plików tymczasowych i zakończenie działania.

Innym powodem zakończenia pracy jest sytuacja, w której proces wykryje błąd krytyczny.

Jeśli na przykład użytkownik wpisze polecenie:

```
cc foo.c
```

w celu skompilowania programu *foo.c*, a taki plik nie istnieje, to kompilator po prostu skończy działanie. Procesy interaktywne wyposażone w interfejsy ekranowe zwykle nie kończą działania, jeśli zostaną do nich przekazane błędne parametry. Zamiast tego wyświetlają okno dialogowe z prośbą do użytkownika o ponowienie próby.

Trzecim powodem zakończenia pracy jest błąd spowodowany przez proces — często wynikający z błędu w programie. Może to być uruchomienie niedozwolonej instrukcji, odwołanie się do nieistniejącego obszaru pamięci lub dzielenie przez zero. W niektórych systemach (na przykład w Uniksie) proces może poinformować system operacyjny, że sam chce obsłużyć określone błędy. W takim przypadku, jeśli wystąpi błąd, proces otrzymuje sygnał (przerwanie), zamiast zakończyć pracę.

Czwartym powodem, dla którego proces może zakończyć działanie, jest wykonanie wywołania systemowego, które zleca systemowi operacyjnemu zniszczenie innego procesu. W Uniksie można to zrobić za pomocą wywołania systemowego `kill`. Odpowiednikiem tego wywołania w interfejsie Win32 API jest `TerminateProcess`. W obu przypadkach proces niszczący musi posiadać odpowiednie uprawnienia do niszczenia innych procesów. W niektórych systemach zakończenie procesu — niezależnie od tego, czy jest wykonywane dobrowolnie, czy przymusowo — wiąże się z zakończeniem wszystkich procesów utworzonych przez ten proces. Jednak w taki sposób nie działa ani system UNIX, ani Windows.

2.1.4. Hierarchie procesów

W niektórych systemach, kiedy proces utworzy inny proces, to proces-rodzic jest w pewien sposób związany z procesem-dzieckiem. Proces-dziecko sam może tworzyć kolejne procesy, co formuje hierarchię procesów. Zwróćmy uwagę, że w odróżnieniu od roślin i zwierząt rozmnażających się płciowo proces może mieć tylko jednego rodzica (ale zero, jedno dziecko lub więcej dzieci).

W Uniksie proces wraz z wszystkimi jego dziećmi i dalszymi potomkami tworzy grupę procesów. Kiedy użytkownik wyśle sygnał z klawiatury, sygnał ten jest dostarczany do wszystkich członków grupy procesów, które w danym momencie są powiązane z klawiaturą (zwykle są to wszystkie aktywne procesy utworzone w bieżącym oknie). Każdy proces może indywidualnie przechwycić sygnał, zignorować go lub podjąć działanie domyślne — to znaczy zostać zniszczonym przez sygnał.

W celu przedstawienia innego przykładu sytuacji, w której hierarchia procesów odgrywa rolę, przyjrzyjmy się sposobowi, w jaki system UNIX inicjuje się podczas rozruchu. W obrazie rozruchowym występuje specjalny proces o nazwie `init`. Kiedy rozpoczyna działanie, odczytuje plik i informuje o liczbie dostępnych terminali. Następnie tworzy po jednym nowym procesie na terminal. Procesy te czekają, aż ktoś się zaloguje. Kiedy logowanie zakończy się pomyślnie, proces logowania uruchamia powłokę, która jest gotowa na przyjmowanie poleceń. Polecenia te mogą uruchamiać nowe procesy itd. Tak więc wszystkie procesy w całym systemie należą do tego samego drzewa — jego korzeniem jest proces `init`.

Dla odróżnienia w systemie Windows nie występuje pojęcie hierarchii procesów. Wszystkie procesy są sobie równe. Jedyną oznaką hierarchii procesu jest to, że podczas tworzenia procesu rodzic otrzymuje specjalny znacznik (nazywany **uchwytem** — ang. *handle*), który może wykorzystać do zarządzania dzieckiem. Może jednak

swobodnie przekazać ten znacznik do innego procesu i w ten sposób zdezaktualizować hierarchię. Procesy w Uniksie nie mają możliwości „wydziedziczenia" swoich dzieci.

2.1.5. Stany procesów

Chociaż każdy proces jest niezależnym podmiotem, posiadającym własny licznik programu i wewnętrzny stan, procesy często muszą się komunikować z innymi procesami. Jeden proces może generować wyjście, które inny proces wykorzysta jako wejście. W poleceniu powłoki:

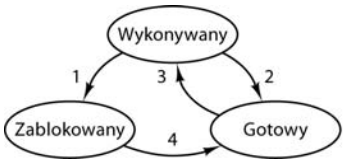
```
cat rozdzial1 rozdzial2 rozdzial3 | grep drzewo
```

pierwszy proces uruchamia polecenie `cat`, łączy i wyprowadza trzy pliki. Drugi proces uruchamia polecenie `grep`, wybiera wszystkie wiersze zawierające słowo „drzewo". W zależności od względnej szybkości obu procesów (co z kolei zależy zarówno od względnej złożoności programów, jak i tego, ile czasu procesora każdy z nich ma do dyspozycji), może się zdarzyć, że polecenie `grep` będzie gotowe do działania, ale nie będą na nie czekały żadne dane wejściowe. Proces będzie się musiał zablokować do czasu, aż będą one dostępne.

Proces blokuje się, ponieważ z logicznego punktu widzenia nie może kontynuować działania. Zazwyczaj dzieje się tak dlatego, że oczekuje na dane wejściowe, które jeszcze nie są dostępne. Jest również możliwe, że proces, który jest gotowy i zdolny do działania, zostanie zatrzymany ze względu na to, że system operacyjny zdecydował się przydzielić procesor na pewien czas jakiemuś innemu procesowi. Te dwie sytuacje diametralnie różnią się od siebie. W pierwszym przypadku wstrzymanie pracy jest ściśle związane z charakterem problemu (nie można przetworzyć wiersza poleceń wprowadzanego przez użytkownika do czasu, kiedy użytkownik go nie wprowadzi). W drugim przypadku to techniczne aspekty systemu (niewystarczająca liczba procesorów do tego, aby każdy proces otrzymał swój prywatny procesor). Na rysunku 2.2 pokazano diagram stanów pokazujący trzy stany, w jakich może znajdować się proces:

1. Działanie (rzeczywiste korzystanie z procesora w tym momencie).
2. Gotowość (proces może działać, ale jest tymczasowo wstrzymany, aby inny proces mógł działać).
3. Blokada (proces nie może działać do momentu, w którym wydarzy się jakieś zewnętrzne zdarzenie).

Z logicznego punktu widzenia pierwsze dwa stany są do siebie podobne. W obu przypadkach proces chce działać, ale w drugim przypadku chwilowo brakuje dla niego czasu procesora. Trzeci stan różni się od pierwszych dwóch w tym sensie, że proces nie może działać nawet wtedy, gdy procesor w tym czasie nie ma innego zajęcia.



Rysunek 2.2. Proces może byś w stanie działania, blokady lub gotowości. Na rysunku pokazano przejścia pomiędzy tymi stanami

Tak jak pokazano na rysunku, pomiędzy tymi trzema stanami możliwe są cztery przejścia. Przejście nr 1 występuje wtedy, kiedy system operacyjny wykryje, że proces nie może kontynuować działania. W niektórych systemach proces może wykonać wywołanie systemowe, na przykład `pause`, w celu przejścia do stanu zablokowania. W innych systemach, w tym w Uniksie, kiedy proces czyta dane z potoku lub pliku specjalnego (na przykład terminala) i dane wejściowe są niedostępne, jest automatycznie blokowany.

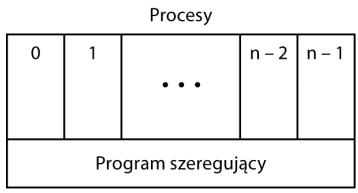
Przejścia nr 2 i nr 3 są realizowane przez program szeregujący (ang. *process scheduler*) — część systemu operacyjnego, a procesy nie są o tym nawet informowane. Przejście nr 2 zachodzi wtedy, gdy program szeregujący zdecyduje, że działający proces działał wystarczająco długo i nadszedł czas, by przydzielić czas procesora jakiemuś innemu procesowi. Przejście nr 3 zachodzi wtedy, gdy wszystkie inne procesy skorzystały ze swojego udziału i nadszedł czas na to, by pierwszy proces otrzymał procesor i wznowił działanie. Zadanie szeregowania procesów — to znaczy decydowania o tym, który proces powinien się uruchomić, kiedy i na jak długo — jest bardzo ważne. Przyjrzymy się mu bliżej w dalszej części tego rozdziału. Opracowano wiele algorytmów mających na celu zapewnienie równowagi pomiędzy wymaganiami wydajności systemu jako całości oraz sprawiedliwego przydziału procesora do indywidualnych procesów. Niektóre z tych algorytmów omówimy w dalszej części niniejszego rozdziału.

Przejście nr 4 występuje wtedy, gdy zachodzi zewnętrzne zdarzenie, na które proces oczekiwał (na przykład nadejście danych wejściowych). Jeśli w tym momencie nie działa żaden inny proces, zajdzie przejście nr 3 i proces rozpocznie działanie. W innym przypadku może być zmuszony do oczekiwania w stanie *gotowości* przez pewien czas, aż procesor stanie się dostępny i nadejdzie jego kolejka.

Wykorzystanie modelu procesów znacznie ułatwia myślenie o tym, co dzieje się wewnątrz systemu. Niektóre procesy uruchamiają programy realizujące polecenia wprowadzane przez użytkownika. Inne procesy są częścią systemu i obsługują takie zadania, jak obsługa żądań usług plikowych lub zarządzanie szczegółami dotyczącymi uruchamiania napędu dysku lub taśm. Kiedy zachodzi przerwanie dyskowe, system podejmuje decyzję o zatrzymaniu działania bieżącego procesu i uruchamia proces dyskowy, który był zablokowany w oczekiwaniu na to przerwanie. Tak więc zamiast myśleć o przerwaniach, możemy myśleć o procesach użytkownika, procesach dysku, procesach terminala itp., które blokują się w czasie oczekiwania, aż coś

się wydarzy. Kiedy nastąpi próba czytania danych z dysku albo użytkownik przyciśnie klawisz, proces oczekujący na to zdarzenie jest odblokowywany i może wznowić działanie.

Ten stan rzeczy jest podstawą modelu pokazanego na rysunku 2.3. W tym przypadku na najniższym poziomie systemu operacyjnego znajduje się program szeregujący, zarządzający zbiorem procesów występujących w warstwie nad nim. Cały mechanizm obsługi przerwań i szczegółów związanych z właściwym uruchamianiem i zatrzymywaniem procesów jest ukryty w elemencie nazwanym tu zarządcą procesów. Element ten w rzeczywistości nie zawiera zbyt wiele kodu. Pozostała część systemu operacyjnego ma strukturę procesów. W praktyce jednak istnieje bardzo niewiele systemów operacyjnych, które miałyby tak przejrzystą strukturę.



Rysunek 2.3. Najniższa warstwa systemu operacyjnego o strukturze procesów zarządza przerwaniami i szeregowaniem. Powyżej tej warstwy znajdują się sekwencyjne procesy

2.1.6. Implementacja procesów

W celu zaimplementowania modelu procesów w systemie operacyjnym występuje tabela (tablica struktur), zwana **tabelą procesów**, w której każdemu z procesów odpowiada jedna pozycja — niektórzy autorzy nazywają te pozycje **blokami zarządzania procesami**. W blokach tych są zapisane ważne informacje na temat stanu procesu. Zawierają one wartości licznika programu, wskaźnika stosu, dane dotyczące przydziału pamięci, statusu otwartych procesów, rozliczeń i szeregowania oraz wszystkie inne informacje, które trzeba zapisać w czasie przełączania procesu ze stanu *wykonywany* do stanu *gotowy* lub *zablokowany*. Dzięki nim proces może być później wznowiony, tak jakby nigdy nie został zatrzymany.

W tabeli 2.1. pokazano kilka kluczowych pól w typowym systemie. Pola w pierwszej kolumnie są związane z zarządzaniem procesami. Pozostałe dwa łączą się odpowiednio z zarządzaniem pamięcią oraz zarządzaniem plikami. Należy zwrócić uwagę na to, że obecność poszczególnych pól w tabeli procesów w dużym stopniu zależy od systemu. Poniższa tabela daje jednak ogólny obraz rodzajów potrzebnych informacji.

Teraz, kiedy przyjrzeliśmy się tabeli procesów, możemy wyjaśnić nieco dokładniej to, w jaki sposób iluzja wielu sekwencyjnych procesów jest utrzymywana w jednym procesorze (lub każdym z procesorów). Z każdą klasą wejścia-wyjścia wiąże się lokalizacja (zwykle pod ustalonym adresem w dolnej części pamięci) zwana **wektorem przerwań**. Jest w niej zapisany adres procedury obsługi przerwania. Załóżmy,

Tabela 2.1. Przykładowe pola typowego wpisu w tabeli procesów

Zarządzanie procesami	**Zarządzanie pamięcią**	**Zarządzanie plikami**
Rejestry	Wskaźnik do informacji segmentu tekstu	Katalog główny
Licznik programu	Wskaźnik do informacji segmentu danych	Katalog roboczy
Słowo stanu programu	Wskaźnik do informacji segmentu stosu	Deskryptory plików
Wskaźnik stosu		Identyfikator użytkownika
Stan procesu		Identyfikator grupy
Priorytet		
Parametry szeregowania		
Identyfikator procesu		
Proces-rodzic		
Grupa procesów		
Sygnały		
Czas rozpoczęcia procesu		
Wykorzystany czas CPU		
Czas CPU procesów-dzieci		
Godzina następnego alarmu		

że w momencie wystąpienia przerwania związanego z dyskiem ma działać proces użytkownika nr 3. Sprzęt obsługujący przerwania odkłada na stos licznik programu procesu użytkownika nr 3, słowo stanu programu i czasami jeden lub kilka rejestrów. Następnie sterowanie przechodzi pod adres określony w wektorze przerwań. To jest wszystko, co robi sprzęt. Od tego momentu obsługą przerwania zajmuje się oprogramowanie — w szczególności procedura obsługi przerwania.

Obsługa każdego przerwania rozpoczyna się od zapisania rejestrów — często pod pozycją tabeli procesów odpowiadającą bieżącemu procesowi. Następnie informacje odłożone na stos przez mechanizm obsługi przerwania są z niego zdejmowane, a wskaźnik stosu jest ustawiany na adres tymczasowego stosu używanego przez procedurę obsługi procesu. Takich działań, jak zapisanie rejestrów i ustawienie wskaźnika stosu, nawet nie można wyrazić w językach wysokopoziomowych, takich jak C. W związku z tym operacje te są wykonywane przez niewielką procedurę w języku asemblera. Zazwyczaj jest to ta sama procedura dla wszystkich przerwań, ponieważ zadanie zapisania rejestrów jest identyczne, niezależnie od tego, co było przyczyną przerwania.

Kiedy ta procedura zakończy działanie, wywołuje procedurę w języku C, która wykonuje resztę pracy dla tego konkretnego typu przerwania (zakładamy, że system operacyjny został napisany w języku C — w tym języku napisana jest większość systemów operacyjnych). Kiedy procedura ta wykona swoje zadanie (co może spowodować, że pewne procesy uzyskają gotowość do działania), wywoływany jest program szeregujący, który ma sprawdzić, jaki proces powinien zostać uruchomiony w następnej kolejności. Następnie sterowanie jest przekazywane z powrotem do

kodu w asemblerze, który ładuje rejestry i mapę pamięci nowego bieżącego procesu oraz rozpoczyna jego działanie. Obsługę przerwań i szeregowanie podsumowano w tabeli 2.2. Warto zwrócić uwagę, że różne systemy nieco się różnią pewnymi szczegółami.

Tabela 2.2. Szkielet działań wykonywanych przez najniższy poziom systemu operacyjnego w momencie wystąpienia przerwania

1. Sprzęt odkłada na stos licznik programu itp.
2. Sprzęt ładuje nowy licznik programu z wektora przerwań.
3. Procedura w języku asemblera zapisuje rejestry.
4. Procedura w języku asemblera ustawia nowy stos.
5. Uruchamia się procedura obsługi przerwania w C (zazwyczaj czyta i buforuje dane wejściowe).
6. Program szeregujący decyduje o tym, który proces ma być uruchomiony w następnej kolejności.
7. Procedura w języku C zwraca sterowanie do kodu w asemblerze.
8. Procedura w języku asemblera uruchamia nowy bieżący proces.

Kiedy proces zakończy działanie, system operacyjny wyświetla symbol zachęty i oczekuje na nowe polecenie. Po otrzymaniu polecenia ładuje do pamięci nowy program, nadpisując starą zawartość pamięci.

2.1.7. Modelowanie wieloprogramowości

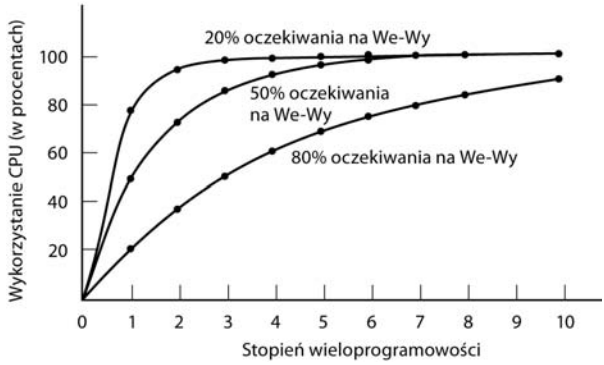
Zastosowanie wieloprogramowości pozwala na poprawę wykorzystania procesora. Z grubsza rzecz biorąc, jeśli przeciętny proces jest przetwarzany przez 20% czasu rezydowania w pamięci, to w przypadku gdy w pamięci jest jednocześnie pięć procesów, procesor powinien być zajęty przez cały czas. Ten model jest jednak nierealistycznie optymistyczny, ponieważ zakłada, że w żadnym momencie nie zdarzy się sytuacja, w której wszystkie pięć procesów będzie jednocześnie oczekiwało na operację wejścia-wyjścia.

Lepszym modelem jest spojrzenie na wykorzystanie procesora z probabilistycznego punktu widzenia. Załóżmy, że proces spędza fragment p swojego czasu na zakończeniu operacji wejścia-wyjścia. Przy n procesach znajdujących się jednocześnie w pamięci prawdopodobieństwo tego, że wszystkie n procesów będzie jednocześnie oczekiwało na obsługę wejścia-wyjścia (wtedy procesor pozostanie bezczynny), wynosi p^n. W takim przypadku wykorzystanie procesora można opisać za pomocą wzoru:

$$\text{Wykorzystanie procesora} = 1 - p^n$$

Na rysunku 2.4 pokazano procent wykorzystania procesora w funkcji n — co określa się jako **stopień wieloprogramowości**.

Z rysunku jasno wynika, że jeśli procesy spędzają 80% czasu w oczekiwaniu na operacje wejścia-wyjścia, to aby współczynnik marnotrawienia procesora utrzymać



Rysunek 2.4. Wykorzystanie procesora w funkcji liczby procesów w pamięci

na poziomie poniżej 10%, w pamięci musi być jednocześnie co najmniej 10 procesów. Kiedy zdamy sobie sprawę ze stanu, w którym proces interaktywny oczekuje, aż użytkownik wpisze na terminalu jakieś dane, stanie się oczywiste, że czasy oczekiwania na wejścia-wyjścia rzędu 80% i więcej nie są niczym niezwykłym. Nawet na serwerach procesy wykonujące wiele dyskowych operacji wejścia-wyjścia często charakteryzują się tak wysokim procentem.

Dla ścisłości należy dodać, że model probabilistyczny opisany przed chwilą jest tylko przybliżeniem. Zakłada on niejawnie, że wszystkie n procesów jest niezależnych. Oznacza to, że w przypadku systemu z pięcioma procesami w pamięci dopuszczalnym stanem jest to, aby trzy z nich działały, a dwa czekały. Jednak przy jednym procesorze nie ma możliwości jednoczesnego działania trzech procesów. W związku z tym proces, który osiąga gotowość w czasie, gdy procesor jest zajęty, będzie musiał czekać. Tak więc procesy nie są niezależne. Dokładniejszy model można stworzyć z wykorzystaniem teorii kolejokowania, jednak teza, którą sformułowaliśmy — wieloprogramowość pozwala procesom wykorzystywać procesor w czasie, gdy w innej sytuacji byłby on bezczynny — jest oczywiście w dalszym ciągu prawdziwa. Faktu tego nie zmieniłaby nawet sytuacja, w której rzeczywiste krzywe stopnia wieloprogramowości nieco odbiegałyby od tych pokazanych na rysunku 2.4.

Mimo że model z rysunku 2.4 jest uproszczony, można go wykorzystywać w celu tworzenia specyficznych, jednak przybliżonych prognoz dotyczących wydajności procesora. Przypuśćmy na przykład, że komputer ma 512 MB pamięci, przy czym system operacyjny zajmuje 128 MB, a każdy z programów użytkownika również zajmuje do 128 MB. Te rozmiary pozwalają na to, aby w pamięci jednocześnie znajdowały się trzy programy użytkownika. Przy średnim czasie oczekiwania na operacje wejścia-wyjścia, wynoszącym 80%, mamy procent wykorzystania procesora na poziomie $1-0{,}8^3$ czyli około 49%. Dodanie kolejnych 512 MB pamięci operacyjnej umożliwia przejście systemu z trójstopniowej wieloprogramowości do siedmiostopniowej, co przyczyni się do wzrostu wykorzystania procesora do 79%. Mówiąc inaczej, dodatkowe 512 MB pamięci podniesie przepustowość o 30%.

Dodanie kolejnych 512 MB spowodowałoby zwiększenie stopnia wykorzystania procesora z 79% do 91%, a zatem podniosłoby przepustowość tylko o kolejne 12%. Korzystając z tego modelu, właściciel komputera może zadecydować, że pierwsza rozbudowa systemu jest dobrą inwestycją, natomiast druga nie.

2.2. WĄTKI

W tradycyjnych systemach operacyjnych każdy proces ma przestrzeń adresową i jeden wątek sterowania. W rzeczywistości prawie tak wygląda definicja procesu. Niemniej jednak często występują sytuacje, w których korzystne jest posiadanie wielu wątków sterowania w tej samej przestrzeni adresowej, działających quasi-równolegle — tak jakby były (niemal) oddzielnymi procesami (z wyjątkiem współdzielonej przestrzeni adresowej). Sytuacje te oraz wynikające z tego implikacje omówiono w kolejnych punktach.

2.2.1. Wykorzystanie wątków

Do czego może służyć rodzaj procesu wewnątrz innego procesu? Okazuje się, że istnieją powody istnienia tych miniprocesów zwanych **wątkami**. Spróbujmy przyjrzeć się kilku z nich. Głównym powodem występowania wątków jest to, że w wielu aplikacjach jednocześnie wykonywanych jest wiele działań. Niektóre z nich mogą być zablokowane od czasu do czasu. Dzięki dekompozycji takiej aplikacji na wiele sekwencyjnych wątków działających quasi-równolegle model programowania staje się prostszy.

Taką samą dyskusję przedstawiliśmy już wcześniej. Dokładnie te same argumenty przemawiają za istnieniem procesów. Zamiast myśleć o przerwaniach, licznikach czasu i przełączaniu kontekstu, możemy myśleć o równoległych procesach. Tyle że teraz, przy pojęciu wątków, dodajemy nowy element: zdolność równoległych podmiotów do współdzielenia pomiędzy sobą przestrzeni adresowej oraz wszystkich swoich danych. Zdolność ta ma kluczowe znaczenie dla niektórych aplikacji, dlatego właśnie obecność wielu procesów (z oddzielnymi przestrzeniami adresowymi) w tym przypadku nie wystarczy.

Drugi argument, który przemawia za istnieniem wątków, jest taki, że — ponieważ są one mniejsze od procesów — w porównaniu z procesami łatwiej (tzn. szybciej) się je tworzy i niszczy. W wielu systemach tworzenie wątku trwa 10 – 100 razy krócej od tworzenia procesu. Ponieważ liczba potrzebnych wątków zmienia się dynamicznie i gwałtownie, szybkość nabiera dużego znaczenia.

Trzecim powodem istnienia wątków są względy wydajności. Istnienie wątków nie poprawi wydajności, jeśli wszystkie one będą związane z procesorem. Jednak w przypadku wykonywania intensywnych obliczeń i jednocześnie znaczącej liczby operacji wejścia-wyjścia występowanie wątków pozwala na nakładanie się na siebie tych działań, co w efekcie końcowym przyczynia się do przyspieszenia aplikacji.

Na koniec — wątki przydają się w systemach wyposażonych w wiele procesorów, gdzie możliwa jest rzeczywista współbieżność. Do tego zagadnienia powrócimy w rozdziale 8.

Najłatwiej przekonać się o przydatności wątków, analizując konkretne przykłady. W roli pierwszego przykładu rozważmy edytor tekstu. Edytory tekstu zazwyczaj wyświetlają na ekranie tworzony dokument sformatowany dokładnie w takiej postaci, w jakiej będzie on wyglądał na drukowanej stronie. Zwłaszcza wszystkie znaki podziału wierszy i stron znajdują się na prawidłowych i ostatecznych pozycjach. Dzięki temu użytkownik ma możliwość przeglądania i poprawienia dokumentu, jeśli zajdzie taka potrzeba (na przykład w celu wyeliminowania sierot i wdów — niekompletnych wierszy na początku i na końcu strony, które uważa się za nieestetyczne).

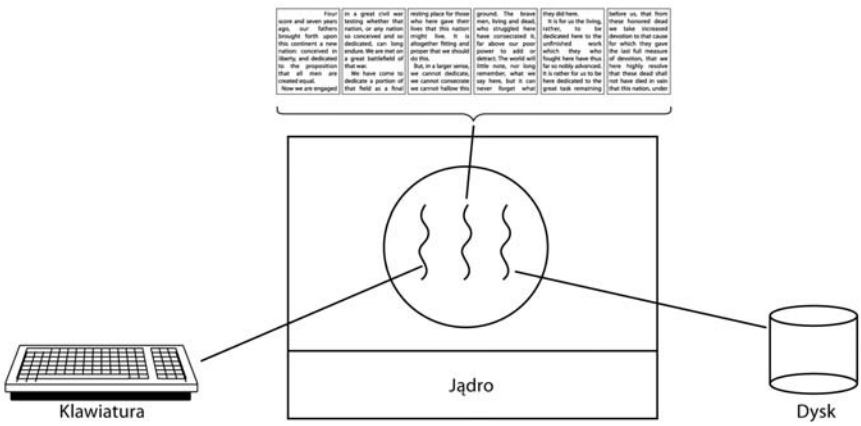
Załóżmy, że użytkownik pisze książkę. Z punktu widzenia autora najłatwiej umieścić całą książkę w pojedynczym pliku, tak by łatwiej było wyszukiwać tematy, wykonywać globalne operacje zastępowania itp. Alternatywnie można umieścić każdy z rozdziałów w osobnym pliku. Jednak umieszczenie każdego podrozdziału i punktu w osobnym pliku, na przykład gdyby zaszła potrzeba globalnego zastąpienia jakiegoś terminu w całej książce, byłoby prawdziwym utrapieniem. W takim przypadku trzeba by było bowiem indywidualnie edytować każdy z kilkuset plików. Jeśli na przykład zaproponowany termin „standard xxxx" zostałby zatwierdzony tuż przed oddaniem książki do druku, trzeba by było w ostatniej chwili zastąpić wszystkie wystąpienia terminu „roboczo: standard xxxx" na „standard xxxx". Jeśli książka znajduje się w jednym pliku, taką operację można wykonać za pomocą jednego polecenia. Dla odróżnienia, gdyby książka składała się z 300 plików, każdy z nich trzeba by osobno otworzyć w edytorze.

Rozważmy teraz, co się zdarzy, kiedy użytkownik nagle usunie jedno zdanie z pierwszej strony 800-stronicowego dokumentu. Po sprawdzeniu poprawności zmodyfikowanej strony zdecydował, że chce wykonać inną zmianę na stronie 600 i wpisuje polecenie zlecające edytorowi przejście do tej strony (na przykład poprzez wyszukanie frazy, która znajduje się tylko tam). Edytor tekstu jest w tej sytuacji zmuszony do natychmiastowego przeformatowania całej książki do strony 600, ponieważ nie będzie wiedział, jaką treść ma pierwszy wiersz na stronie 600, dopóki nie przetworzy wszystkich poprzednich stron. Zanim będzie można wyświetlić stronę 600, może powstać znaczące opóźnienie, co doprowadzi do niezadowolenia użytkownika.

W takim przypadku może pomóc wykorzystanie wątków. Załóżmy, że edytor tekstu jest napisany jako program składający się z dwóch wątków. Jeden wątek zajmuje się komunikacją z użytkownikiem, a drugi przeprowadza w tle korektę formatowania. Natychmiast po usunięciu zdania ze strony 1 wątek komunikacji z użytkownikiem informuje wątek formatujący o konieczności przeformatowania całej książki. Tymczasem wątek komunikacji z użytkownikiem kontynuuje nasłuchiwanie klawiatury i myszy i odpowiada na proste polecenia, takie jak przeglądanie strony 1. W tym samym czasie drugi z wątków w tle wykonuje intensywne obliczenia. Przy odrobinie

szczęścia zmiana formatu zakończy się, zanim użytkownik poprosi o przejście na stronę 600. Jeśli tak się stanie, przejście na stronę 600 będzie mogło się odbyć bezzwłocznie.

Kiedy już jesteśmy przy edytorach, odpowiedzmy sobie na pytanie, dlaczego by nie dodać trzeciego wątku. Wiele edytorów tekstu jest wyposażonych w mechanizm automatycznego zapisywania całego pliku na dysk co kilka minut. Ma to zapobiec utracie całodniowej pracy w przypadku awarii programu, awarii systemu lub problemów z zasilaniem. Trzeci wątek może obsługiwać wykonywanie kopii zapasowych na dysku, nie przeszkadzając w działaniu pozostałym dwóm. Sytuację z trzema wątkami pokazano na rysunku 2.5.



Rysunek 2.5. Edytor tekstu składający się z trzech wątków

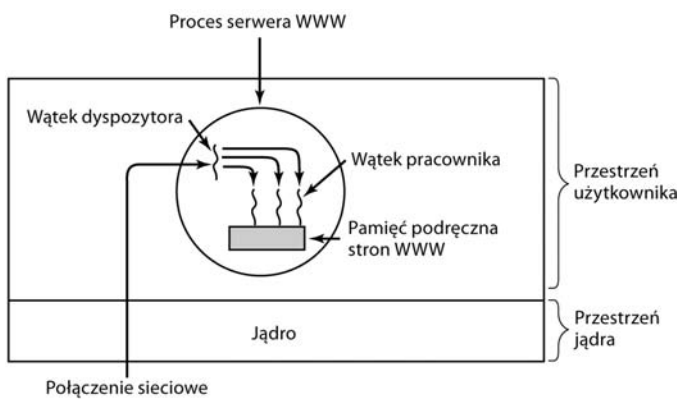
Gdyby program zawierał jeden wątek, to każde rozpoczęcie wykonywania kopii zapasowej na dysk powodowałoby, że polecenia z klawiatury i myszy byłyby ignorowane do czasu zakończenia wykonywania kopii zapasowej. Użytkownik z pewnością by to zauważył jako obniżoną wydajność. Alternatywnie zdarzenia związane z klawiaturą i myszą mogłyby przerwać wykonywanie kopii zapasowej na dysk, co pozwoliłoby na zachowanie dobrej wydajności, ale prowadziłoby do skomplikowanego modelu programowania bazującego na przerwaniach. W przypadku zastosowania trzech wątków model programowania jest znacznie prostszy. Pierwszy wątek zajmuje się jedynie interakcjami z użytkownikiem. Drugi wątek przeformatowuje dokument, kiedy otrzyma takie zlecenie. Trzeci wątek okresowo zapisuje zawartość pamięci RAM na dysk.

W tym przypadku powinno być jasne, że istnienie trzech oddzielnych procesów w tej sytuacji się nie sprawdzi, ponieważ wszystkie trzy wątki muszą operować na tym samym dokumencie. Dzięki występowaniu trzech wątków zamiast trzech procesów wątki współdzielą pamięć i w efekcie wszystkie mają dostęp do edytowanego dokumentu.

Analogiczna sytuacja występuje w przypadku wielu innych interaktywnych programów. Na przykład elektroniczny arkusz kalkulacyjny jest programem umożliwiającym użytkownikowi obsługę macierzy — niektóre z jej elementów są danymi wprowadzanymi przez użytkownika. Inne elementy są wyliczane na podstawie wprowadzonych danych i z wykorzystaniem potencjalnie skomplikowanych wzorów. Kiedy użytkownik zmodyfikuje jeden element, może zajść potrzeba obliczenia wielu innych elementów. Dzięki zdefiniowaniu działającego w tle wątku zajmującego się przeliczaniem wątek interaktywny pozwala użytkownikowi na wprowadzanie zmian w czasie, gdy są wykonywane obliczenia. Na podobnej zasadzie trzeci wątek może samodzielnie obsługiwać kopie zapasowe wykonywane na dysku.

Rozważmy teraz jeszcze jeden przykład zastosowania wątków: serwer ośrodka WWW. Przychodzą żądania stron, a w odpowiedzi żądane strony są przesyłane do klienta. W większości ośrodków WWW niektóre strony WWW są częściej odwiedzane niż inne. Na przykład główna strona serwisu Sony jest odwiedzana znacznie częściej od strony umieszczonej głęboko w drzewie katalogów i zawierającej specyfikację techniczną jakiegoś modelu kamery wideo. Serwery WWW wykorzystują ten fakt do poprawy wydajności. Utrzymują kolekcję często używanych stron w pamięci głównej, aby wyeliminować potrzebę odwoływania się do dysku w celu ich pobrania. Taka kolekcja jest nazywana pamięcią podręczną (ang. *cache*) i wykorzystuje się ją również także w wielu innych kontekstach. Na przykład w rozdziale 1. zetknęliśmy się z pamięciami podręcznymi procesora.

Jeden ze sposobów organizacji serwera WWW pokazano na rysunku 2.6(a). W tym przypadku jeden z wątków — **dyspozytor** — odczytuje z sieci przychodzące żądania. Po przeanalizowaniu żądania wybiera bezczynny (tzn. zablokowany) **wątek pracownika** i przekazuje mu żądanie — na przykład poprzez zapisanie wskaźnika do komunikatu w specjalnym słowie powiązanym z każdym wątkiem. Następnie dyspozytor budzi uśpiony wątek pracownika — to znaczy zmienia jego stan z „zablokowany” na „gotowy”.



Rysunek 2.6. Serwer WWW z obsługą wielu wątków

Kiedy wątek się obudzi, sprawdza, czy jest w stanie spełnić żądanie z pamięci podręcznej strony WWW, do której mają dostęp wszystkie wątki. Jeśli tak nie jest, rozpoczyna operację odczytu w celu pobrania strony z dysku i przechodzi do stanu „zablokowany", trwającego do chwili zakończenia operacji dyskowej. Kiedy wątek zablokuje się na operacji dyskowej, inny wątek zaczyna działanie, na przykład dyspozytor, którego zadaniem jest przyjęcie jak największej liczby żądań, albo inny pracownik, który jest gotowy do działania.

W tym modelu serwer może być zapisany w postaci kolekcji sekwencyjnych wątków. Program dyspozytora zawiera pętlę nieskończoną, w której jest pobierane żądanie pracy, później wręczane pracownikowi. Kod każdego pracownika zawiera pętlę nieskończoną, w której jest akceptowane żądanie od dyspozytora i następuje sprawdzenie, czy żądana strona jest dostępna w pamięci podręcznej serwera WWW. Jeśli tak, strona jest zwracana do klienta, a pracownik blokuje się w oczekiwaniu na nowe żądanie. Jeśli nie, pracownik pobiera stronę z dysku, zwraca ją do klienta i blokuje się w oczekiwaniu na nowe żądanie.

W uproszczonej formie kod przedstawiono na listingu 2.1. W tym przypadku, podobnie jak w pozostałej części tej książki, założono, że `TRUE` odpowiada stałej o wartości 1. Natomiast `buf` i `strona` są strukturami do przechowywania odpowiednio żądania pracy i strony WWW.

Listing 2.1. Uproszczona postać kodu dla struktury serwera z rysunku 2.6. (a) Wątek dyspozytora; (b) wątek pracownika

(a)

```
while (TRUE){
    pobierz_nast_zadanie(&buf);
    przekaz_prace(&buf);
}
```

(b)

```
while (TRUE){
    czekaj_na_prace(&buf)
    szukaj_strony_w_pamieci_cache(&buf,
    ↪&strona);
    if ((&strona))
        czytaj_strone_z_dysku(&buf,
        ↪&strona);
    zwroc_strone(&strona);
}
```

Zastanówmy się, jak mógłby być napisany serwer WWW, gdyby nie było wątków. Jedna z możliwości polega na zaimplementowaniu go jako pojedynczego wątku. W głównej pętli serwera WWW następowałyby pobieranie żądania, jego analiza i realizacja. Dopiero potem serwer WWW mógłby pobrać następne żądanie. Podczas oczekiwania na zakończenie operacji dyskowej serwer byłby bezczynny i nie przetwarzałby żadnych innych przychodzących żądań. Jeśli serwer WWW działa na dedykowanej maszynie, tak jak to zwykle bywa, w czasie oczekiwania serwera WWW na dysk procesor pozostałby bezczynny. W efekcie końcowym można by było przetworzyć znacznie mniej żądań na sekundę. A zatem skorzystanie z wątków pozwala na uzyskanie znaczącego zysku wydajności, ale każdy z wątków jest programowany sekwencyjnie — w standardowy sposób.

Do tej pory omówiliśmy dwa możliwe projekty: wielowątkowy serwer WWW i jednowątkowy serwer WWW. Załóżmy, że wątki nie są dostępne, ale projektanci systemu uznali obniżenie wydajności spowodowane istnieniem pojedynczego wątku za niedopuszczalne. Jeśli jest dostępna nieblokująca wersja wywołania systemowego `read`, możliwe staje się trzecie podejście. Kiedy przychodzi żądanie, analizuje go jeden i tylko jeden wątek. Jeżeli żądanie może być obsłużone z pamięci podręcznej, to dobrze, ale jeśli nie, inicjowana jest nieblokująca operacja dyskowa.

Serwer rejestruje stan bieżącego żądania w tabeli, a następnie pobiera następne zdarzenie do obsługi. Może to być żądanie nowej pracy albo odpowiedź dysku dotycząca poprzedniej operacji. Jeśli jest to żądanie nowej pracy, rozpoczyna się jego obsługa. Jeśli jest to odpowiedź z dysku, właściwe informacje są pobierane z tabeli i następuje przetwarzanie odpowiedzi. W przypadku nieblokujących dyskowych operacji wejścia-wyjścia odpowiedź zwykle ma postać sygnału lub przerwania.

W tym projekcie model „procesów sekwencyjnych" omawiany w pierwszych dwóch przypadkach nie występuje. Stan obliczeń musi być jawnie zapisany i odtworzony z tabeli, za każdym razem, kiedy serwer przełącza się z pracy nad jednym żądaniem do pracy nad kolejnym żądaniem. W rezultacie wątki i ich stosy są symulowane w trudniejszy sposób. W projektach takich jak ten wszystkie obliczenia mają zapisany stan. Ponadto istnieje zbiór zdarzeń, których wystąpienie może zmieniać określone stany. Takie systemy nazywa się **automatami o skończonej liczbie stanów** — pojęcie to jest powszechnie używane w branży komputerowej.

Teraz powinno być jasne, co oferują wątki. Pozwalają na utrzymanie idei procesów sekwencyjnych wykonujących blokujące wywołania systemowe (na przykład dotyczące dyskowych operacji wejścia-wyjścia) z jednoczesnym uzyskaniem efektu współbieżności. Blokujące wywołania systemowe ułatwiają programowanie, a współbieżność poprawia wydajność. Jednowątkowy serwer zachowuje prostotę blokujących wywołań systemowych, ale gwarantuje wydajność. Trzecie podejście pozwala na osiągnięcie wysokiej wydajności dzięki współbieżności, ale wykorzystuje nieblokujące wywołania i przerwania, dlatego jest trudne do zaprogramowania. Dostępne modele zestawiono w tabeli 2.3.

Tabela 2.3. Trzy sposoby konstrukcji serwera

Model	Charakterystyka
Wątki	Współbieżność, blokujące wywołania systemowe
Proces jednowątkowy	Brak współbieżności, blokujące wywołania systemowe
Automat o skończonej liczbie stanów	Współbieżność, nieblokujące wywołania systemowe, przerwania

Trzecim przykładem zastosowania wątków są aplikacje, które muszą przetwarzać duże ilości danych. Normalne podejście polega na przeczytaniu bloku danych, przetworzeniu go, a następnie ponownym zapisaniu. Problem w takim przypadku

polega na tym, że jeśli dostępne są tylko blokujące wywołania systemowe, proces blokuje się, kiedy dane przychodzą oraz kiedy są wysyłane na zewnątrz. Doprowadzenie do sytuacji, w której procesor jest bezczynny w czasie, gdy jest wiele obliczeń do wykonania, to oczywiste marnotrawstwo i w miarę możliwości należy unikać takiej sytuacji.

Rozwiązaniem problemu jest wykorzystanie wątków. Wewnątrz procesu można wydzielić wątek wejściowy, wątek przetwarzania danych i wątek wyprowadzania danych. Wątek wejściowy czyta dane do bufora wejściowego. Wątek przetwarzania danych pobiera dane z bufora wejściowego, przetwarza je i umieszcza wyniki w buforze wyjściowym. Wątek wyprowadzania danych zapisuje wyniki z bufora wyjściowego na dysk. W ten sposób wprowadzanie danych, ich wyprowadzanie i przetwarzanie mogą być realizowane w tym samym czasie. Oczywiście model ten działa tylko wtedy, kiedy wywołanie systemowe blokuje wyłącznie wątek wywołujący, a nie cały proces.

2.2.2. Klasyczny model wątków

Teraz, kiedy pokazaliśmy, do czego mogą się przydać wątki i jak ich można używać, spróbujmy przeanalizować to zagadnienie nieco dokładniej. Model procesów bazuje na dwóch niezależnych pojęciach: grupowaniu zasobów i uruchamianiu. Czasami wygodnie jest je rozdzielić — wtedy można skorzystać z wątków. Najpierw przyjrzymy się klasycznemu modelowi wątków. Następnie omówimy model wątków Linuksa, w którym linia pomiędzy wątkami i procesami jest rozmyta.

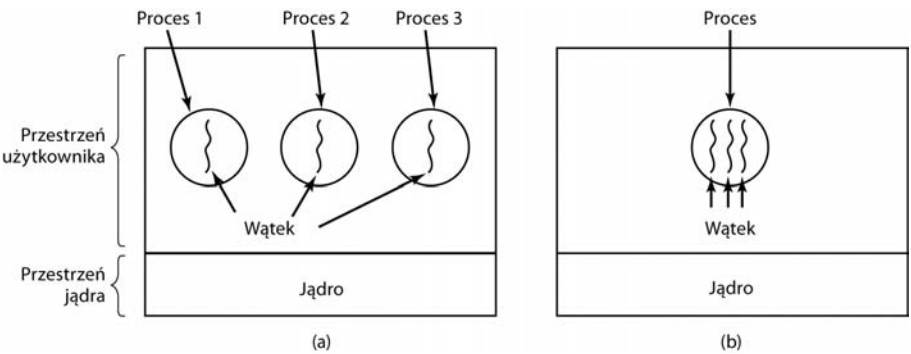
Jednym ze sposobów patrzenia na proces jest postrzeganie go jako sposobu grupowania powiązanych ze sobą zasobów. Proces dysponuje przestrzenią adresową zawierającą tekst programu i dane, a także inne zasoby. Do zasobów tych można zaliczyć otwarte pliki, procesy-dzieci, nieobsłużone alarmy, porcedury obsługi sygnałów, informacje rozliczeniowe i wiele innych. Dzięki pogrupowaniu ich w formie procesu można nimi łatwiej zarządzać.

W innym pojęciu proces zawiera wykonywany wątek — zwykle w skrócie używa się samego pojęcia wątku. **Wątek** zawiera licznik programu, który śledzi to, jaka instrukcja będzie wykonywana w następnej kolejności. Posiada rejestry zawierające jego bieżące robocze zmienne. Ma do dyspozycji stos zawierający historię działania — po jednej ramce dla każdej procedury, której wykonywanie się rozpoczęło, ale jeszcze się nie zakończyło. Chociaż wątek musi realizować jakiś proces, wątek i jego proces są pojęciami odrębnymi i można je traktować osobno. Procesy są wykorzystywane do grupowania zasobów, wątki są podmiotami zaplanowanymi do wykonania przez procesor.

Wątki dodają do modelu procesu możliwość realizacji wielu wykonań w tym samym środowisku procesu, w dużym stopniu w sposób wzajemnie od siebie niezależny. Równoległe działanie wielu wątków w obrębie jednego procesu jest analogiczne do równoległego działania wielu procesów w jednym komputerze. W pierwszym z tych przypadków wątki współdzielą przestrzeń adresową i inne zasoby.

W drugim przypadku procesy współdzielą pamięć fizyczną, dyski, drukarki i inne zasoby. Ponieważ wątki mają pewne właściwości procesów, czasami nazywa się je **lekkimi procesami**. Do opisania sytuacji, w której w tym samym procesie może działać wiele wątków, używa się także terminu **wielowątkowość**. Jak widzieliśmy w rozdziale 1., niektóre procesory mają bezpośrednią obsługę sprzętową wielowątkowości i pozwalają na przełączanie wątków w skali czasowej rzędu nanosekund.

Na rysunku 2.7(a) widać trzy tradycyjne procesy. Każdy proces ma swoją własną przestrzeń adresową oraz pojedynczy wątek sterowania. Dla odmiany w układzie z rysunku 2.7(b) widzimy jeden proces z trzema wątkami sterowania. Chociaż w obu przypadkach mamy trzy wątki, w sytuacji z rysunku 2.7(a) każdy z nich działa w innej przestrzeni adresowej, podczas gdy w sytuacji z rysunku 2.7(b) wszystkie współdzielą tę samą przestrzeń adresową.



Rysunek 2.7. (a) Trzy procesy, z których każdy posiada jeden wątek; (b) jeden wątek z trzema wątkami

Kiedy wielowątkowy proces działa w jednoprocesorowym systemie, wątki działają po kolei. Na rysunku 2.1 widzieliśmy, jak działa wieloprogramowość procesów. Dzięki przełączaniu pomiędzy wieloma procesami system daje iluzję oddzielnych procesów sekwencyjnych działających współbieżnie. Wielowątkowość działa w taki sam sposób. Procesor przełącza się w szybkim tempie pomiędzy wątkami, dając iluzję, że wątki działają współbieżnie — chociaż na wolniejszym procesorze od fizycznego. Przy trzech wątkach obliczeniowych w procesie wątki będą sprawiały wrażenie równoległego działania, ale tak, jakby każdy z nich działał na procesorze o szybkości równej jednej trzeciej szybkości fizycznego procesora.

Różne wątki procesu nie są tak niezależne, jak różne procesy. Wszystkie wątki posługują się dokładnie tą samą przestrzenią adresową, co również oznacza, że współdzielą one te same zmienne globalne. Ponieważ każdy wątek może uzyskać dostęp do każdego adresu pamięci w obrębie przestrzeni adresowej procesu, jeden wątek może odczytać, zapisać, a nawet wyczyścić stos innego wątku. Pomiędzy wątkami nie ma zabezpieczeń, ponieważ (1) byłyby one niemożliwe do realizacji, a (2) nie powinny być potrzebne. W odróżnieniu od różnych procesów, które potencjalnie

należą do różnych użytkowników i które mogą być dla siebie wrogie, proces zawsze należy do jednego użytkownika, który przypuszczalnie utworzył wiele wątków, a zatem powinny one współpracować, a nie walczyć ze sobą. Oprócz przestrzeni adresowej wszystkie wątki mogą współdzielić ten sam zbiór otwartych plików, procesów-dzieci, alarmów, sygnałów itp., tak jak pokazano w tabeli 2.4. Tak więc organizacja pokazana na rysunku 2.7(a) mogłaby zostać użyta, jeśli trzy procesy są ze sobą niezwiązane, natomiast organizacja z rysunku 2.7(b) byłaby właściwa w przypadku, gdyby trzy wątki były częścią tego samego zadania i gdyby aktywnie i ściśle ze sobą współpracowały.

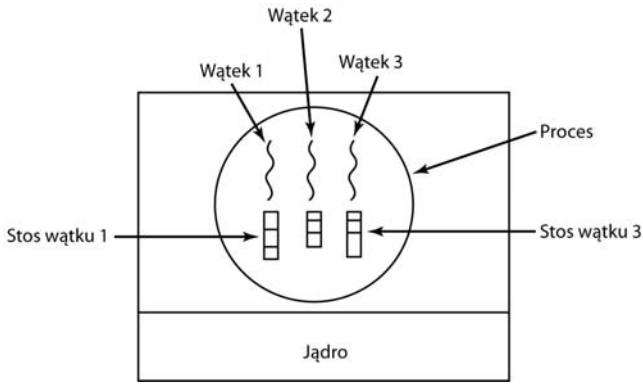
Tabela 2.4. W pierwszej kolumnie wyszczególniono cechy wspólne dla wszystkich wątków w procesie. W drugiej kolumnie zamieszczono niektóre elementy prywatne dla każdego wątku

Komponenty procesu	**Komponenty wątku**
Przestrzeń adresowa	Licznik programu
Zmienne globalne	Rejestry
Otwarte pliki	Stos
Procesy-dzieci	Stan
Zaległe alarmy	
Sygnały i procedury obsługi sygnałów	
Informacje dotyczące rozliczeń	

Elementy w pierwszej kolumnie są właściwościami procesu, a nie wątku. Jeśli na przykład jeden wątek otworzy plik, będzie on widoczny dla innych wątków w procesie. Wątki te będą mogły czytać dane z pliku i je zapisywać. To logiczne, ponieważ właśnie proces, a nie wątek jest jednostką zarządzania zasobami. Gdyby każdy wątek miał własną przestrzeń adresową, otwarte pliki, nieobsłużone alarmy itd., byłby osobnym procesem. Wykorzystując pojęcie wątków, chcemy, aby wiele wątków mogło współdzielić zbiór zasobów. Dzięki temu mogą one ze sobą ściśle współpracować w celu wykonania określonego zadania.

Podobnie jak tradycyjny proces (czyli taki, który zawiera tylko jeden wątek), wątek może znajdować się w jednym z kilku stanów: „działający", „zablokowany", „gotowy" lub „zakończony". Działający wątek posiada dostęp do procesora i jest aktywny. Zablokowany wątek oczekuje na jakieś zdarzenie, by mógł się odblokować. Kiedy na przykład wątek realizuje wywołanie systemowe odczytujące dane z klawiatury, jest zablokowany do czasu, kiedy użytkownik wpisze dane wejściowe. Wątek może się blokować w oczekiwaniu na wystąpienie zdarzenia zewnętrznego lub może oczekiwać, aż odblokuje go inny wątek. Wątek gotowy jest zaplanowany do uruchomienia i zostanie uruchomiony, kiedy nadejdzie jego kolej. Przejścia pomiędzy stanami wątków są identyczne jak przejścia pomiędzy stanami procesów. Zilustrowano je na rysunku 2.2.

Istotne znaczenie ma zdanie sobie sprawy, że każdy wątek posiada własny stos, co zilustrowano na rysunku 2.8. Stos każdego wątku zawiera po jednej ramce dla każdej procedury, która została wywołana, a z której jeszcze nie nastąpił powrót. Ramka ta zawiera zmienne lokalne procedury oraz adres powrotu, który będzie wykorzystany po zakończeniu obsługi wywołania procedury. Jeśli na przykład procedura X wywoła procedurę Y, a procedura Y wywoła procedurę Z, to w czasie, kiedy działa procedura Z, na stosie będą ramki dla procedur X, Y i Z. Każdy wątek, ogólnie rzecz biorąc, będzie wywoływał inne procedury, a zatem będzie miał inną historię wywołań. Dlatego właśnie każdy wątek potrzebuje własnego stosu.



Rysunek 2.8. Każdy wątek ma własny stos

W przypadku gdy system obsługuje wielowątkowość, procesy zazwyczaj rozpoczynają działanie z jednym wątkiem. Wątek ten posiada zdolność do tworzenia nowych wątków za pomocą wywołania procedury, na przykład `thread_create`. Parametr procedury `thread_create` zwykle określa nazwę procedury, która ma się uruchomić dla nowego wątku. Nie jest konieczne (ani nawet możliwe) ustalenie czegokolwiek na temat przestrzeni adresowej nowego wątku, ponieważ wątek automatycznie działa w przestrzeni adresowej wątku tworzącego. Czasami wątki są hierarchiczne i zachodzą pomiędzy nimi relacje rodzic – dziecko, często jednak takie relacje nie występują, a wszystkie wątki są sobie równe. Niezależnie od tego, czy pomiędzy wątkami zachodzi relacja hierarchii, wątek tworzący zwykle zwraca identyfikator wątku zawierający nazwę nowego wątku.

Kiedy wątek zakończy swoją pracę, może zakończyć działanie poprzez wywołanie procedury bibliotecznej, na przykład `thread_exit`. W tym momencie wątek znika i nie może być więcej zarządzany. W niektórych systemach obsługi wątków jeden wątek może czekać na zakończenie innego wątku poprzez wywołanie procedury, na przykład `thread_join`. Procedura ta blokuje wątek wywołujący do czasu zakończenia (specyficznego) wątku. Pod tym względem tworzenie i kończenie wątków przypomina tworzenie i kończenie procesów i wymaga w przybliżeniu tych samych opcji.

Innym popularnym wywołaniem dotyczącym wątków jest `thread_yield`. Umożliwia ono wątkowi dobrowolną rezygnację z procesora w celu umożliwienia działania innemu wątkowi. Takie wywołanie ma istotne znaczenie, ponieważ nie istnieje przerwanie zegara, które wymuszałoby wieloprogramowość, tak jak w przypadku procesów. W związku z tym istotne znaczenie ma to, aby wątki były „uprzejme" i od czasu do czasu dobrowolnie rezygnowały z procesora, tak by inne wątki miały szanse na działanie. Są również inne wywołania — na przykład pozwalające na to, aby jeden wątek poczekał, aż następny zakończy jakąś pracę, lub by ogłosił, że właśnie zakończył jakąś pracę itd.

Chociaż wątki często się przydają, wprowadzają także szereg komplikacji do modelu programowania. Na początek przeanalizujmy efekty na uniksowe wywołanie systemowej `fork`. Jeśli proces-rodzic ma wiele wątków, to czy proces-dziecko również powinien je mieć? Jeśli nie, to proces może nie działać prawidłowo, ponieważ wszystkie wątki mogą mieć istotne znaczenie.

Tymczasem gdy proces-dziecko otrzyma tyle samo wątków, co rodzic, to co się stanie, jeśli wątek należący do rodzica zostanie zablokowany przez wywołanie `read`, powiedzmy, z klawiatury? Czy teraz dwa wątki są zablokowane przez klawiaturę — jeden w procesie-rodzicu i drugi w dziecku? Kiedy użytkownik wpisze wiersz, to czy kopia pojawi się w obu wątkach? A może tylko w wątku rodzica? Lub tylko w wątku dziecka? Ten sam problem występuje dla twardych połączeń sieciowych.

Inna klasa problemów wiąże się z faktem współdzielenia przez wątki wielu struktur danych. Co się dzieje, jeśli jeden wątek zamknie plik, podczas gdy inny ciągle z niego czyta? Przypuśćmy, że jeden z wątków zauważa, że jest za mało pamięci, i rozpoczyna alokowanie większej ilości pamięci. W trakcie tego działania następuje przełączenie wątku. Nowy wątek również zauważa, że jest za mało pamięci i także rozpoczyna alokowanie dodatkowej pamięci. Pamięć prawdopodobnie będzie alokowana dwukrotnie. Przy odrobinie wysiłku można rozwiązać te problemy, jednak poprawna praca programów wykorzystujących wielowątkowość wymaga dokładnych przemyśleń i dokładnego projektowania.

2.2.3. Wątki POSIX

Aby było możliwe napisanie przenośnego programu z obsługą wielu wątków, organizacja IEEE zdefiniowała standard 1003.1c. Pakiet obsługi wątków, który tam zdefiniowano, nosi nazwę `Pthreads`. Jest on obsługiwany przez większość systemów uniksowych. W standardzie zdefiniowano ponad 60 wywołań funkcji. To o wiele za dużo, by można je było dokładnie omówić w tej książce. Omówimy zatem kilka najważniejszych. Dzięki temu Czytelnik uzyska obraz ich działania. Wywołania, które opiszemy, zostały wyszczególnione w tabeli 2.5.

Tabela 2.5. Niektóre wywołania funkcji należące do pakietu Pthreads

Wywołanie obsługi wątku	Opis
`Pthread_create`	Utworzenie nowego wątku
`Pthread_exit`	Zakończenie wątku wywołującego
`Pthread_join`	Oczekiwanie na zakończenie specyficznego wątku
`Pthread_yield`	Zwolnienie procesora w celu umożliwienia działania innemu wątkowi
`Pthread_attr_init`	Utworzenie i zainicjowanie struktury atrybutów wątku
`Pthread_attr_destroy`	Usunięcie struktury atrybutów wątku

Wszystkie wątki pakietu `Pthreads` mają określone właściwości. Każdy z nich posiada identyfikator, zbiór rejestrów (łącznie z licznikiem programu) oraz zbiór atrybutów zapisanych w pewnej strukturze. Do atrybutów tych należy rozmiar stosu, parametry szeregowania oraz inne elementy potrzebne do korzystania z wątku.

Nowy wątek tworzy się za pomocą wywołania `pthread_create`. Jako wartość funkcji zwracany jest identyfikator nowo utworzonego wątku. Wywołanie to nieprzypadkowo przypomina wywołanie systemowe `fork`. W tym przypadku identyfikator wątku spełnia rolę identyfikatora PID, głównie do celów identyfikacji wątków w innych wywołaniach.

Kiedy wątek zakończy pracę, która została do niego przydzielona, może zakończyć swoje działanie poprzez wywołanie funkcji `pthread_exit`. Wywołanie to zatrzymuje wątek i zwalnia jego stos.

Często wątek musi czekać, aż inny wątek zakończy swoją pracę. Dopiero później może kontynuować działanie. Wątek oczekujący na zakończenie specyficznego innego wątku wywołuje funkcję `pthread_join`. Identyfikator wątku, który ma się zakończyć, jest przekazywany jako parametr.

Czasami się zdarza, że wątek nie jest logicznie zablokowany, ale czuje, że działa już dość długo, i chce dać innemu wątkowi szansę działania. Cel ten można osiągnąć za pomocą wywołania `pthread_yield`. Nie ma takiego wywołania w przypadku procesów, ponieważ zakłada się, że procesy ze sobą rywalizują i każdy z nich chce uzyskać maksymalnie dużo czasu procesora. Ponieważ jednak wątki procesu współdziałają ze sobą, a ich kod jest pisany przez tego samego programistę, czasami programista chce, aby każdy z wątków otrzymał swoją szansę.

Następne dwa wywołania obsługi wątków dotyczą atrybutów wątku. Wywołanie `Pthread_attr_init` tworzy strukturę atrybutów powiązaną z wątkiem i inicjuje ją do wartości domyślnych. Wartości te (takie jak priorytet) można zmieniać poprzez modyfikowanie pól w strukturze atrybutów.

Na koniec — wywołanie `pthread_attr_destroy` usuwa strukturę atrybutów wątku i zwalnia pamięć. Wywołanie to nie ma wpływu na wątki korzystające z atrybutów. Wątki te w dalszym ciągu istnieją.

Aby uzyskać lepszy obraz tego, jak działa pakiet Pthreads, rozważmy prosty przykład z listingu 2.2. Główny program wykonuje się w pętli NUMBER_OF_THREADS razy. W każdej iteracji program wyświetla komunikat i tworzy nowy wątek. Jeśli tworzenie wątku nie powiedzie się, program wyświetla komunikat o błędzie i kończy działanie. Po utworzeniu wszystkich wątków program główny kończy działanie.

Listing 2.2. Przykładowy program wykorzystujący wątki

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>

#define NUMBER_OF_THREADS 10

void *print-hello_world(void *tid)
{
    /* Funkcja wyświetla identyfikator wątku i kończy działanie. */
    printf("Witaj, Świecie. Pozdrowienia od wątku %d0, tid);
    pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
    /* Program główny tworzy 10 wątków i kończy działanie. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Tu program główny. Tworzenie wątku %d0, i);
        status = pthread_create(&threads[i], NULL, print_hello_world,
        ↪(void *)i);

        if (status != 0) {
        printf("Oops. Funkcja pthread_create zwróciła kod błędu %d0, status);
        exit(-1);
        }
}
exit(NULL);
}
```

Podczas tworzenia wątek wyświetla jednowierszowy komunikat, w którym się przedstawia, a następnie kończy działanie. Kolejność, w jakiej będą się pojawiały poszczególne komunikaty, nie jest określona i może być różna w kolejnych uruchomieniach programu.

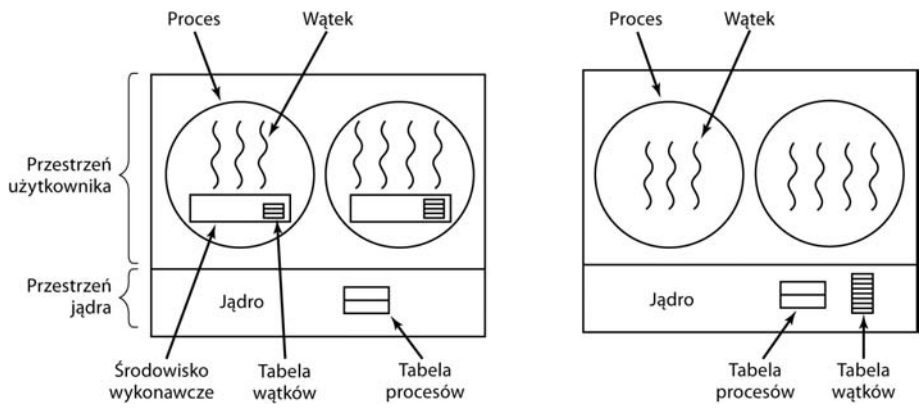
Pakiet Pthreads w żadnym razie nie ogranicza się do funkcji opisanych powyżej. Jest ich znacznie więcej. Niektóre z kolejnych wywołań opiszemy później, po omówieniu zagadnienia synchronizacji procesów i wątków.

2.2.4. Implementacja wątków w przestrzeni użytkownika

Istnieją dwa główne sposoby implementacji pakietu obsługi wątków: w przestrzeni użytkownika i w jądrze. Podział ten jest dość płynny. Możliwe są również implementacje hybrydowe. Poniżej opiszemy obie metody razem z ich zaletami i wadami.

Pierwsza metoda polega na umieszczeniu pakietu wątków w całości w przestrzeni użytkownika. Jądro nic o nich nie wie. Z jego punktu widzenia procesy, którymi zarządza, są standardowe — jednowątkowe. Pierwsza i najbardziej oczywista zaleta tego rozwiązania polega na tym, że pakiet obsługi wątków na poziomie przestrzeni użytkownika można zaimplementować w systemie operacyjnym, który nie obsługuje wątków. Do tej kategorii w przeszłości należały wszystkie systemy operacyjne i nawet dziś niektóre do niej należą. Przy takim podejściu wątki są implementowane za pomocą biblioteki.

Wszystkie tego rodzaju implementacje mają taką samą ogólną strukturę, co zilustrowano na rysunku 2.9(a). Wątki działają na bazie środowiska wykonawczego — kolekcji procedur, które nimi zarządzają. Do tej pory zapoznaliśmy się z czterema procedurami z tej grupy: `pthread_create`, `pthread_exit`, `pthread_join` oraz `pthread_yield`. Zwykle jednak jest ich więcej.



Rysunek 2.9. (a) Pakiet obsługi wątków na poziomie użytkownika; (b) pakiet obsługi wątków zarządzany przez jądro

Jeśli wątki są zarządzane w przestrzeni użytkownika, każdy proces potrzebuje swojej prywatnej **tabeli wątków**, która ma na celu śledzenie wątków w tym procesie. Tabela ta jest analogiczna do tabeli procesów w jądrze. Różnica polega na tym, że śledzi ona właściwości tylko na poziomie wątku — na przykład licznik programu każdego z wątków, wskaźnik stosu, rejestry, stan itp. Tabela wątków jest zarządzana przez środowisko wykonawcze. Kiedy wątek przechodzi do stanu gotowości lub zablokowania, informacje potrzebne do jego wznowienia są zapisywane w tabeli wątków, dokładnie w taki sam sposób, w jaki jądro zapisuje informacje o procesach w tabeli procesów.

Kiedy wątek wykona operację, która może spowodować jego lokalne zablokowanie, na przykład oczekuje, aż inny wątek w tym samym procesie wykona jakąś pracę, wykonuje procedurę ze środowiska wykonawczego. Procedura ta sprawdza, czy wątek musi być przełączony do stanu zablokowania. Jeśli tak, to zapisuje rejestry wątku (tzn. własne) w tabeli wątków, szuka w tabeli wątku gotowego do działania i ładuje rejestry maszyny zapisanymi wartościami odpowiadającymi nowemu wątkowi. Po przełączeniu wskaźnika stosu i licznika programu nowy wątek automatycznie powraca do życia. Jeśli maszyna posiada instrukcję zapisującą wszystkie rejestry oraz inną instrukcję, która je wszystkie ładuje, przełączenie wątku można przeprowadzić za pomocą zaledwie kilku instrukcji. Przeprowadzenie przełączania wątków w taki sposób jest co najmniej o jeden rząd wielkości szybsze od wykonywania rozkazu pułapki do jądra. To silny argument przemawiający za implementacją pakietu zarządzania wątkami na poziomie przestrzeni użytkownika.

Jest jednak jedna zasadnicza różnica w porównaniu z procesami. Kiedy wątek zakończy na chwilę działanie, na przykład kiedy wywoła funkcję `thread_yield`, kod funkcji `thread_yield` może zapisać informacje dotyczące wątku w samej tabeli wątków. Co więcej, może on następnie wywołać zarządcę wątków w celu wybrania innego wątku do uruchomienia. Procedura zapisująca stan wątku oraz program szeregujący są po prostu lokalnymi procedurami, zatem wywołanie ich jest znacznie bardziej wydajne od wykonania wywołania jądra. Między innymi nie jest potrzebny rozkaz pułapki, nie trzeba przełączać kontekstu, nie trzeba opróżniać pamięci podręcznej. W związku z tym zarządzanie wątkami odbywa się bardzo szybko.

Implementacja wątków na poziomie przestrzeni użytkownika ma także inne zalety. Dzięki temu każdemu procesowi można przypisać własny, spersonalizowany algorytm szeregowania. W przypadku niektórych aplikacji, na przykład zawierających wątek mechanizmu odśmiecania, brak konieczności przejmowania się możliwością zatrzymania się wątku w nieodpowiednim momencie jest zaletą. Takie rozwiązanie okazuje się również łatwiejsze do skalowania, ponieważ wątki zarządzane na poziomie jądra niewątpliwie wymagają przestrzeni na tabelę i stos w jądrze, a to, w przypadku dużej liczby wątków, może być problemem.

Pomimo lepszej wydajności implementacja wątków na poziomie przestrzeni użytkownika ma również istotne wady. Pierwsza z nich dotyczy sposobu implementacji blokujących wywołań systemowych. Przypuśćmy, że wątek czyta z klawiatury, zanim zostanie wciśnięty jakikolwiek klawisz. Zezwolenie wątkowi na wykonanie wywołania systemowego jest niedopuszczalne, ponieważ spowoduje to zatrzymanie wszystkich wątków. Trzeba pamiętać, że jednym z podstawowych celów korzystania z wątków jest umożliwienie wszystkim wątkom używania wywołań blokujących, a przy tym niedopuszczenie do tego, by zablokowany wątek miał wpływ na inne. W przypadku blokujących wywołań systemowych trudno znaleźć łatwe rozwiązanie pozwalające na spełnienie tego celu.

Wszystkie wywołania systemowe można zmienić na nieblokujące (na przykład odczyt z klawiatury zwróciłby 0 bajtów, gdyby znaki nie były wcześniej zbuforowane),

ale wymaganie zmian w systemie operacyjnym jest nieatrakcyjne. Poza tym jednym z argumentów przemawiających za obsługą wątków na poziomie użytkownika była możliwość wykorzystania takiego mechanizmu w istniejących systemach operacyjnych. Co więcej, zmiana semantyki wywołania `read` wymagałaby modyfikacji wielu programów użytkowych.

Jedną z możliwych alternatyw można zastosować w przypadku, gdy można z góry powiedzieć, czy wywołanie jest blokujące. W niektórych wersjach Uniksa istnieje wywołanie systemowe `select`, które pozwala procesowi wywołującemu na sprawdzenie, czy wywołanie `read` będzie blokujące. Jeśli jest dostępne to wywołanie, można zastąpić procedurę biblioteczną `read` nową wersją, która najpierw wykonuje wywołanie `select`, a następnie wywołuje `read` tylko wtedy, gdy jest to bezpieczne (tzn. nie spowoduje zablokowania). Jeżeli wywołanie `read` ma doprowadzić do zablokowania, nie jest wykonywane. Zamiast wywołania `read` uruchamiany jest inny wątek. Następnym razem, kiedy środowisko wykonawcze otrzyma sterowanie, może sprawdzić ponownie, czy wykonanie wywołania `read` jest bezpieczne. Takie podejście wymaga zmiany implementacji części biblioteki wywołań systemowych, jest niewydajne i nieeleganckie, ale możliwości wyboru są ograniczone. Kod wokół wywołania systemowego, który wykonuje test, określa się **osłoną** lub **opakowaniem** (ang. *wrapper*).

W pewnym sensie podobnym problemem do blokujących wywołań systemowych jest problem braku stron w pamięci (ang. *page faults*). Zagadnienie to omówimy w rozdziale 3. Na razie wystarczy, jeśli powiemy, że komputery można skonfigurować w taki sposób, aby w danym momencie w głównej pamięci znajdowała się tylko część programu. Jeżeli program wywoła lub skoczy do instrukcji, której nie ma w pamięci, występuje warunek braku strony. Wtedy system operacyjny jest zmuszony do pobrania brakującej instrukcji (wraz z jej sąsiadami) z dysku. Na tym właśnie polega warunek braku strony. Podczas gdy potrzebna instrukcja jest wyszukiwana i wczytywana, proces pozostaje zablokowany. Jeśli wątek spowoduje warunek braku strony, jądro, które nawet nie wie o istnieniu wątków, blokuje cały proces do czasu zakończenia dyskowej operacji wejścia-wyjścia. Robi to, mimo że nie ma przeszkód, by inne wątki działały.

Inny problem z pakietami obsługi wątków na poziomie użytkownika polega na tym, że jeśli wątek zacznie działać, to żaden inny wątek w tym procesie nigdy nie zacznie działać, o ile pierwszy wątek dobrowolnie nie zrezygnuje z procesora. W obrębie pojedynczego procesu nie ma przerwań zegara, dlatego nie ma możliwości szeregowania procesów w trybie cyklicznym (tzn. po kolei). Jeśli wątek z własnej woli nie przekaże sterowania do środowiska wykonawczego, program szeregujący nigdy nie będzie miał szansy działania.

Jednym z możliwych rozwiązań problemu wątków działających bez przerwy jest zlecenie środowisku wykonawczemu żądania sygnału zegara (przerwania) co sekundę w celu przekazania mu kontroli. Takie rozwiązanie okazuje się jednak toporne i trudne do zaprogramowania. Okresowe przerwania zegara z wyższą częstotliwością nie

zawsze są możliwe, a nawet jeśli tak jest, koszt obliczeniowy takiej operacji może być wysoki. Co więcej, wątek również może potrzebować przerwania zegara, co przeszkadza wykorzystaniu zegara przez środowisko wykonawcze.

Innym, rzeczywiście druzgoczącym argumentem przeciwko wątkom zarządzanym na poziomie przestrzeni użytkownika jest fakt, że programiści, ogólnie rzecz biorąc, potrzebują wątków w aplikacjach, gdzie wątki blokują się często — na przykład w wielowątkowym serwerze WWW. Wątki te bezustannie wykonują wywołania systemowe. Kiedy zostanie wykonany rozkaz pułapki do jądra w celu realizacji wywołania systemowego, jądro nie ma nic więcej do roboty przy przełączaniu wątków, w przypadku gdy stary wątek się zablokował, a zlecenie jądru wykonania tej czynności eliminuje potrzebę ciągłego wykonywania wywołań systemowych `select` sprawdzających, czy wywołania systemowe `read` są bezpieczne. Jaki jest sens istnienia wątków w aplikacjach całkowicie powiązanych z procesorem, które rzadko się blokują? Nikt nie jest w stanie zaproponować sensownego rozwiązania problemu wyliczania liczb pierwszych lub grania w szachy z wykorzystaniem wątków, ponieważ realizacja tych programów w ten sposób nie przynosi istotnych korzyści.

2.2.5. Implementacja wątków w jądrze

Rozważmy teraz sytuację, w której jądro wie o istnieniu wątków i to ono nimi zarządza. Środowisko wykonawcze w każdym z procesów nie jest wymagane, co pokazano na rysunku 2.9(b). Tabela wątków również nie występuje w każdym procesie. Zamiast tego jądro dysponuje tabelą wątków, która śledzi wszystkie wątki w systemie. Kiedy wątek chce utworzyć nowy wątek lub zniszczyć istniejący, wykonuje wywołanie systemowe, które następnie realizuje utworzenie lub zniszczenie wątku poprzez aktualizację tabeli wątków na poziomie jądra.

W tabeli wątków w jądrze są zapisane rejestry, stan oraz inne informacje dla każdego wątku. Informacje są takie same, jak w przypadku wątków zarządzanych na poziomie użytkownika, z tą różnicą, że są one umieszczone w jądrze, a nie w przestrzeni użytkownika (wewnątrz środowiska wykonawczego). Informacje te stanowią podzbiór informacji tradycyjnie utrzymywanych przez jądro na temat jednowątkowych procesów — czyli stanu procesów. Oprócz tego jądro utrzymuje również tradycyjną tabelę procesów, która służy do śledzenia procesów.

Wszystkie wywołania, które mogą zablokować wątek, są implementowane jako wywołania systemowe znacząco większym kosztem niż wywołanie procedury środowiska wykonawczego. Kiedy wątek się zablokuje, jądro może uruchomić wątek z tego samego procesu (jeśli jakiś jest gotowy) lub wątek z innego procesu. W przypadku wątków zarządzanych na poziomie przestrzeni użytkownika środowisko wykonawcze uruchamia wątki z własnego procesu do czasu, aż jądro zabierze mu procesor (lub nie będzie wątków gotowych do działania).

Ze względu na relatywnie większy koszt tworzenia i niszczenia wątków na poziomie jądra niektóre systemy przyjmują rozwiązanie „ekologiczne” i ponownie wykorzystują

swoje wątki. W momencie niszczenia wątku jest on oznaczany jako niemożliwy do uruchomienia, ale poza tym struktury danych jądra pozostają bez zmian. Kiedy później trzeba utworzyć nowy wątek, stary wątek jest reaktywowany, co eliminuje konieczność wykonywania pewnych obliczeń. Recykling wątków jest również możliwy w przypadku wątków zarządzanych w przestrzeni użytkownika, ale ponieważ koszty zarządzania wątkami są znacznie niższe, motywacja do korzystania z tego mechanizmu jest mniejsza.

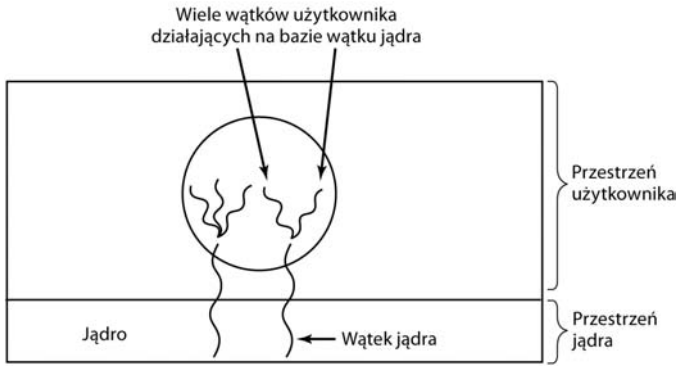
Wątki jądra nie wymagają żadnych nowych nieblokujących wywołań systemowych. Co więcej, jeśli jeden z wątków w procesie spowoduje warunek braku strony, jądro może łatwo sprawdzić, czy proces zawiera inne wątki możliwe do uruchomienia. Jeśli tak, uruchamia jeden z nich w oczekiwaniu na przesłanie z dysku wymaganej strony. Główną wadą tego rozwiązania jest fakt, że koszty wywołania systemowego są znaczące. W związku z tym, w przypadku dużej liczby operacji zarządzania wątkami (tworzenia, niszczenia itp.), ponoszone koszty obliczeniowe okazują się wysokie.

O ile wykorzystanie zarządzania wątkami na poziomie jądra rozwiązuje niektóre problemy, o tyle nie rozwiązuje ich wszystkich. Na przykład co się stanie, jeśli wielowątkowy proces wykona wywołanie `fork`? Czy nowy proces będzie miał tyle wątków, co stary, czy tylko jeden? W wielu przypadkach najlepszy wybór zależy od tego, do czego będzie służył nowy proces. Jeśli zamierza skorzystać z wywołania `exec` w celu uruchomienia nowego programu, prawdopodobnie właściwe będzie stworzenie procesu z jednym wątkiem, jeśli jednak ma on kontynuować działanie, reprodukcja wszystkich wątków wydaje się właściwsza.

Innym problemem są sygnały. Jak pamiętamy, sygnały są przesyłane do procesów, a nie do wątków (przynajmniej w modelu klasycznym). Który wątek ma obsłużyć nadchodzący sygnał? Można wyobrazić sobie rozwiązanie, w którym wątki rejestrują swoje zainteresowanie określonymi sygnałami. Dzięki temu w przypadku nadejścia sygnału mógłby on być skierowany do wątku, który na ten sygnał oczekuje. Co się jednak stanie, jeśli dwa wątki (lub większa liczba wątków) zarejestrują zainteresowanie tym samym sygnałem? To tylko dwa problemy, jakie stwarzają wątki. Jest ich jednak więcej.

2.2.6. Implementacje hybrydowe

Próbowano różnych rozwiązań mających na celu połączenie zalet zarządzania wątkami na poziomie użytkownika oraz zarządzania nimi na poziomie jądra. Jednym ze sposobów jest użycie wątków na poziomie jądra, a następnie zwielokrotnienie niektórych lub wszystkich wątków jądra na wątki na poziomie użytkownika. Sposób ten pokazano na rysunku 2.10. W przypadku skorzystania z takiego podejścia programista może określić, ile wątków jądra chce wykorzystać oraz na ile wątków poziomu użytkownika ma być zwielokrotniony każdy z nich. Taki model daje największą elastyczność.



Rysunek 2.10. Zwielokrotnianie wątków użytkownika na bazie wątków jądra

Przy tym podejściu jądro jest świadome istnienia *wyłącznie* wątków poziomu jądra i tylko nimi zarządza. Niektóre spośród tych wątków mogą zawierać wiele wątków poziomu użytkownika, stworzonych na bazie wątków jądra. Wątki poziomu użytkownika są tworzone, niszczone i zarządzane identycznie, jak wątki na poziomie użytkownika działające w systemie operacyjnym bez obsługi wielowątkowości. W tym modelu każdy wątek poziomu jądra posiada pewien zbiór wątków na poziomie użytkownika. Wątki poziomu użytkownika po kolei korzystają z wątku poziomu jądra.

2.2.7. Mechanizm aktywacji zarządcy

Chociaż zarządzanie wątkami na poziomie jądra jest lepsze od zarządzania nimi na poziomie użytkownika pod pewnymi istotnymi względami, jest ono również bezdyskusyjnie wolniejsze. W związku z tym poszukiwano sposobów poprawy tej sytuacji bez konieczności rezygnacji z ich dobrych właściwości. Poniżej opiszemy jedno z zaproponowanych rozwiązań, opracowane przez zespół kierowany przez Andersona [Anderson et al., 1992] i nazywane **aktywacją zarządcy** (ang. *scheduler activations*). Podobne prace zostały opisane w [Edlera et al., 1988] oraz [Scott et al., 1990].

Celem działania mechanizmu aktywacji zarządcy jest naśladowanie funkcji wątków jądra, ale z zapewnieniem lepszej wydajności i elastyczności — cech, które zwykle charakteryzują pakiety zarządzania wątkami zaimplementowane w przestrzeni użytkownika. W szczególności wątki użytkownika nie powinny wykonywać specjalnych, nieblokujących wywołań systemowych lub sprawdzać wcześniej, czy wykonanie określonych wywołań systemowych jest bezpieczne. Niemniej jednak, kiedy wątek zablokuje się na wywołaniu systemowym lub sytuacji braku strony, powinien mieć możliwość uruchomienia innego wątku w ramach tego samego procesu, jeśli jakiś jest gotowy do działania.

Wydajność osiągnięto dzięki uniknięciu niepotrzebnych przejść pomiędzy przestrzenią użytkownika a przestrzenią jądra. Jeśli na przykład wątek zablokuje się w oczekiwaniu na to, aż inny wątek wykona jakieś działania, nie ma powodu

informowania o tym jądra. Dzięki temu unika się kosztów związanych z przejściami pomiędzy przestrzeniami jądra i użytkownika. Środowisko wykonawcze przestrzeni użytkownika może samodzielnie zablokować wątek synchronizujący i zainicjować nowy.

Kiedy jest wykorzystywany mechanizm aktywacji zarządcy, jądro przypisuje określoną liczbę procesorów wirtualnych do każdego procesu i umożliwia środowisku wykonawczemu (przestrzeni użytkownika) na przydzielanie wątków do procesorów. Mechanizm ten może być również wykorzystany w systemie wieloprocesorowym, w którym zamiast procesorów wirtualnych są procesory fizyczne. Liczba procesorów wirtualnych przydzielonych do procesu zazwyczaj początkowo wynosi jeden, ale proces może poprosić o więcej, a także zwrócić procesory, których już nie potrzebuje. Jądro może również zwrócić wirtualne procesory przydzielone wcześniej, w celu przypisania ich procesom bardziej potrzebującym.

Podstawowa zasada działania tego mechanizmu polega na tym, że jeśli jądro dowie się o blokadzie wątku (na przykład z powodu uruchomienia blokującego wywołania systemowego lub braku strony), to powiadamia o tym środowisko wykonawcze procesu. W tym celu przekazuje na stos w postaci parametrów numer zablokowanego wątku oraz opis zdarzenia, które wystąpiło. Powiadomienie może być zrealizowane dzięki temu, że jądro uaktywnia środowisko wykonawcze znajdujące się pod znanym adresem początkowym. Jest to mechanizm w przybliżeniu analogiczny do sygnałów w Uniksie. Mechanizm ten określa się terminem **wezwanie** (ang. *upcall*).

Po uaktywnieniu w taki sposób środowisko wykonawcze może zmienić harmonogram działania swoich wątków. Zazwyczaj odbywa się to poprzez oznaczenie bieżącego wątku jako zablokowany oraz pobranie innego wątku z listy wątków będących w gotowości, ustawienie jego rejestrów i wznowienie działania. Później, kiedy jądro dowie się, że poprzedni wątek może ponownie działać (na przykład potok, z którego czytał dane, zawiera dane lub brakująca strona została pobrana z dysku), wykonuje kolejne wezwanie do środowiska wykonawczego w celu poinformowania go o tym zdarzeniu. Środowisko wykonawcze może wówczas we własnej gestii natychmiast zrestartować zablokowany wątek lub umieścić go na liście wątków do późniejszego uruchomienia.

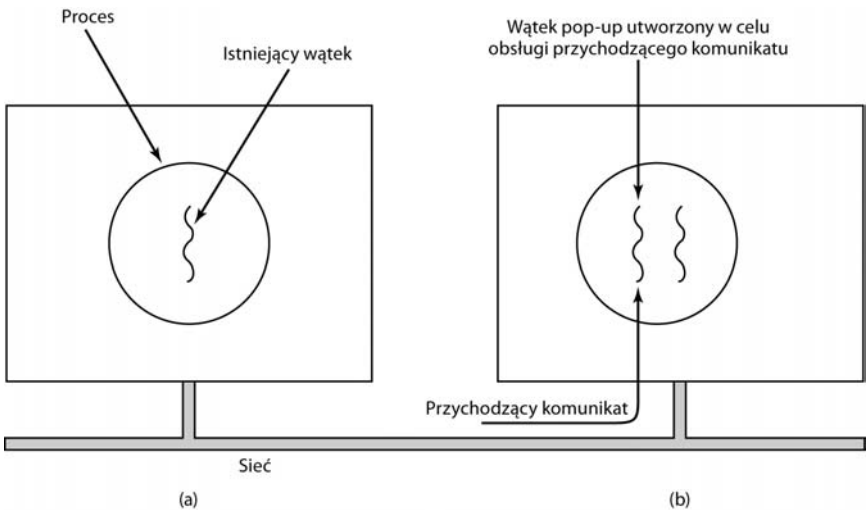
Jeżeli wystąpi przerwanie sprzętowe, gdy działa wątek użytkownika, procesor przełącza się do trybu jądra. Jeśli przerwanie jest spowodowane przez zdarzenie, którym przerwany proces nie jest zainteresowany — na przykład zakończenie operacji wejścia-wyjścia innego procesu — to kiedy procedura obsługi przerwania zakończy działanie, umieszcza przerwany wątek w tym samym stanie, w jakim znajdował się on przed wystąpieniem przerwania. Jeśli jednak proces jest zainteresowany przerwaniem — na przykład nadejście strony wymaganej przez jeden z wątków procesu — przerwany proces nie jest wznawiany. Zamiast tego jest on zawieszany, a na tym samym wirtualnym procesorze zaczyna działać środowisko wykonawcze — stan przerwanego wątku jest w tym momencie umieszczony na stosie. W tym momencie środowisko wykonawcze podejmuje decyzję o tym, jakiemu wątkowi przydzielić dany procesor: przerwanemu, nowo przygotowanemu do działania czy jakiemuś innemu.

Wadą mechanizmu aktywacji zarządcy jest całkowite poleganie na wezwaniach — jest to pojęcie, które narusza wewnętrzną strukturę każdego systemu warstwowego. Zwykle warstwa n oferuje określone usługi, które może wywołać warstwa $n+1$, warstwa n nie może jednak wywoływać procedur w warstwie $n+1$. Mechanizm wezwań narusza tę podstawową zasadę.

2.2.8. Wątki pop-up

Wątki często się przydają w systemach rozproszonych. Istotnym przykładem może być sposób postępowania z nadchodzącymi komunikatami — na przykład żądaniami obsługi. Tradycyjne podejście polega na tym, że na komunikat oczekuje proces lub wątek zablokowany na wywołaniu systemowym `receive`. Kiedy nadejdzie komunikat, wątek ten przyjmuje go, rozpakowuje, analizuje zawartość i przetwarza.

Możliwe jest jednak całkiem inne podejście — po dotarciu komunikatu system tworzy nowy wątek do jego obsługi. Taki wątek określa się jako **wątek pop-up**. Zilustrowano go na rysunku 2.11. Zasadnicza zaleta wątków pop-up polega na tym, że ponieważ są one zupełnie nowe, nie mają żadnej historii — rejestrów, stosu, czegokolwiek, co musiałoby być odtworzone. Każdy wątek rozpoczyna się jako nowy i wszystkie są identyczne. Dzięki temu takie wątki można tworzyć szybko. Nowy wątek otrzymuje przychodzący komunikat do przetworzenia. Dzięki zastosowaniu wątków pop-up opóźnienie pomiędzy przybyciem komunikatu a rozpoczęciem jego przetwarzania jest bardzo niewielkie.



Rysunek 2.11. Tworzenie nowego wątku po przybyciu pakietu: (a) zanim nadejdzie komunikat; (b) po nadejściu komunikatu

W przypadku wykorzystania wątków pop-up potrzebne jest pewne zaawansowane planowanie. Na przykład w którym procesie działa wątek? Jeśli system obsługuje wątki

działające w kontekście jądra, to wątek może tam działać (dlatego właśnie nie pokazaliśmy jądra na rysunku 2.11). Umieszczenie wątku pop-up w przestrzeni jądra zazwyczaj jest łatwiejsze i szybsze niż umieszczenie go w przestrzeni użytkownika. Ponadto wątek pop-up w przestrzeni jądra może łatwo uzyskać dostęp do wszystkich tabel i urządzeń wejścia-wyjścia jądra, co może być potrzebne do przetwarzania przerwań. Z drugiej strony działający błędnie wątek jądra może zrobić więcej szkód niż błędnie działający wątek przestrzeni użytkownika. Jeśli na przykład działa zbyt długo i nie ma możliwości jego wywłaszczenia, wchodzące dane mogą zostać utracone.

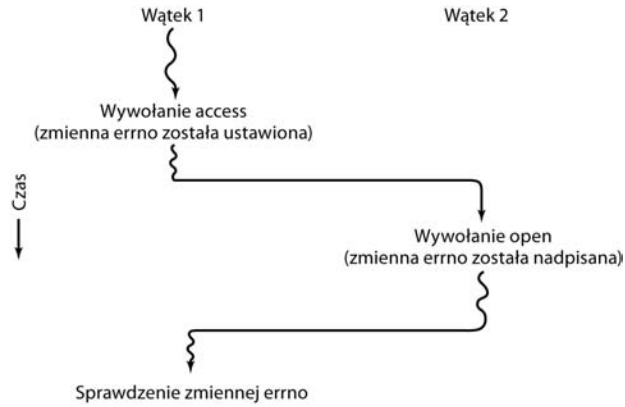
2.2.9. Przystosowywanie kodu jednowątkowego do obsługi wielu wątków

Dla procesów jednowątkowych napisano wiele programów. Ich konwersja na postać wielowątkową jest znacznie trudniejsza, niż mogłoby się wydawać na pierwszy rzut oka. Poniżej zaprezentujemy kilka problemów, które mogą wystąpić podczas takiej konwersji.

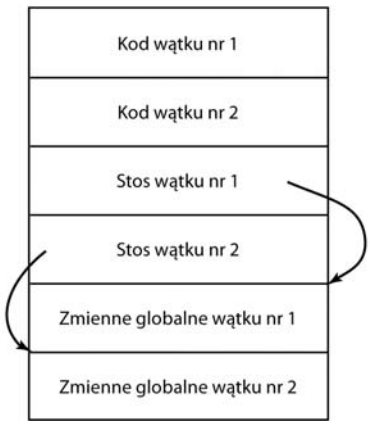
Na początek należy sobie uświadomić, że wątek, tak jak proces, zwykle składa się z wielu procedur. Mogą one mieć zmienne lokalne, zmienne globalne i parametry. Zmienne lokalne i parametry nie powodują żadnych problemów, tymczasem zmienne, które są globalne dla wątku, ale nie są globalne dla całego programu, sprawiają problem. Są to zmienne, które są globalne w tym sensie, że używa ich wiele procedur w obrębie wątku (ponieważ mogą one wykorzystywać dowolne zmienne globalne), ale inne wątki nie powinny z nich korzystać.

Dla przykładu przeanalizujmy zmienną `errno` występującą w systemie UNIX: kiedy proces (lub wątek) wykonuje wywołanie systemowe, które kończy się niepowodzeniem, do zmiennej `errno` jest zapisywany kod błędu. Na rysunku 2.12 wątek nr 1 wykonuje wywołanie systemowe `access` po to, aby się dowiedzieć, czy ma uprawnienia dostępu do określonego pliku. System operacyjny zwraca odpowiedź w zmiennej globalnej `errno`. Po zwróceniu sterowania do wątku 1., ale jeszcze przed przeczytaniem przez niego zmiennej `errno`, program szeregujący zdecydował, że wątek nr 1 miał przydzielony procesor wystarczająco długo i zdecydował przełączyć go do wątku 2. Wątek 2. uruchomił wywołanie systemowe `open`, które się nie powiodło. Spowodowało to nadpisanie zmiennej `errno`, a kod dostępu wątku 1. został utracony na zawsze. Kiedy wątek 1. później się uruchomi, przeczyta nieprawidłową wartość i będzie działał nieprawidłowo.

Możliwych jest wiele rozwiązań tego problemu. Jedno z nich polega na całkowitym wyłączeniu zmiennych globalnych. Choć mogłoby się wydawać, że jest to rozwiązanie idealne, koliduje ono z większością istniejących programów. Inne rozwiązanie to przypisanie każdemu wątkowi własnych, prywatnych zmiennych globalnych, tak jak pokazano na rysunku 2.13. W ten sposób każdy wątek będzie miał własną, prywatną kopię zmiennej `errno` i innych zmiennych globalnych, co pozwoli na uniknięcie konfliktów. Przyjęcie tego rozwiązania tworzy nowy poziom zasięgu:



Rysunek 2.12. Konflikty pomiędzy wątkami spowodowany użyciem zmiennej globalnej



Rysunek 2.13. Wątki mogą mieć prywatne zmienne globalne

zmienne widoczne dla wszystkich procedur wątku. Poziom ten występuje obok istniejących poziomów: zmienne widoczne tylko dla jednej procedury oraz zmienne widoczne w każdym punkcie programu.

Dostęp do prywatnych zmiennych globalnych jest jednak nieco utrudniony, ponieważ większość języków programowania zapewnia sposób wyrażania zmiennych lokalnych i zmiennych globalnych, ale nie ma form pośrednich. Można zaalokować fragment pamięci na zmienne globalne i przekazać go do każdej procedury w wątku w postaci dodatkowego parametru. Chociaż nie jest to zbyt eleganckie rozwiązanie, okazuje się jednak skuteczne.

Alternatywnie można stworzyć nowe procedury biblioteczne do tworzenia, ustawiania i czytania tych zmiennych globalnych na poziomie wątku. Pierwsze wywołanie może mieć następującą postać:

```
create_global("bufptr");
```

Wywołanie to alokuje pamięć dla wskaźnika o nazwie `bufptr` na stercie lub w specjalnym obszarze pamięci zarezerwowanym dla wywołującego wątku. Niezależnie od tego, gdzie jest zaalokowana pamięć, tylko wywołujący wątek ma dostęp do zmiennej globalnej. Jeśli inny wątek utworzy zmienną globalną o tej samej nazwie, otrzyma inną lokalizację w pamięci — taką, która nie koliduje z istniejącą.

Do dostępu do zmiennych globalnych potrzebne są dwa wywołania: jedno do ich zapisywania i drugie do odczytu. Do zapisywania potrzebne jest wywołanie postaci:

```
set_global("bufptr", &buf);
```

Wywołanie to zapisuje wartość wskaźnika w lokalizacji pamięci utworzonej wcześniej przez wywołanie do procedury `create_global`. Wywołanie do przeczytania zmiennej globalnej może mieć następującą postać:

```
bufptr = read_global("bufptr");
```

Zwraca ono adres zapisany w zmiennej globalnej. Dzięki temu można uzyskać dostęp do jej danych.

Następny problem podczas przekształcania programu jednowątkowego na wielowątkowy polega na tym, że wiele procedur bibliotecznych nie pozwala na tak zwaną wielobieżność. Oznacza to, że nie ma możliwości wywołania innej procedury, jeśli poprzednie wywołanie się nie zakończyło. Na przykład wysyłanie wiadomości w sieci można by z powodzeniem zaprogramować w taki sposób, aby wiadomość była tworzona w ustalonym buforze w obrębie biblioteki, a następnie był wykonywany rozkaz pułapki do jądra w celu jej wysłania. Co się stanie, jeśli jeden wątek utworzył swoją wiadomość w buforze, a następnie przerwanie zegara wymusiło przełączenie do drugiego wątku, który natychmiast nadpisze bufor własną wiadomością.

Podobnie procedury alokacji pamięci, na przykład `malloc` w Uniksie, utrzymują kluczowe tabele dotyczące wykorzystania pamięci — na przykład powiązaną listę dostępnych fragmentów pamięci. Podczas gdy procedura `malloc` jest zajęta aktualizacją tych list, mogą one czasowo być w niespójnym stanie — zawierać wskaźniki donikąd. Jeśli nastąpi przełączenie wątku w chwili, gdy tabele będą niespójne i nadejdzie nowe wywołanie z innego wątku, może dojść do użycia nieprawidłowego wskaźnika, co w efekcie może doprowadzić do awarii programu. Skuteczne wyeliminowanie wszystkich tych problemów oznacza konieczność przepisania od nowa całej biblioteki. Wykonanie takiego przedsięwzięcia nie jest proste.

Innym rozwiązaniem jest wyposażenie każdej procedury w kod opakowujący, który ustawia bit do oznaczenia biblioteki tak, jakby była używana. Każda próba innego wątku skorzystania z procedury bibliotecznej, podczas gdy poprzednie wywołanie nie zostało zakończone, jest blokowana. Chociaż takie rozwiązanie jest wykonalne, w dużym stopniu eliminuje ono możliwość wykorzystania współbieżności.

Inną opcją jest wykorzystanie sygnałów. Niektóre sygnały z logicznego punktu widzenia są specyficzne dla wątku, a inne nie. Jeśli na przykład wątek wykonuje wywołanie `alarm`, logiczne jest, aby wynikowy sygnał został przesłany do wątku, który wykonał wywołanie. Jeśli jednak wątki są zaimplementowane w całości w prze-

strzeni użytkownika, jądro nie wie nawet o istnieniu wątków, a zatem trudno mu skierować sygnał do właściwego wątku. Dodatkowe komplikacje występują w przypadku, gdy proces pozwala na występowanie tylko jednego nieobsłużonego alarmu w danym momencie, a kilka wątków niezależnie wykonuje wywołanie `alarm`.

Inne sygnały, na przykład przerwanie klawiatury, nie są specyficzne dla wątku. Co powinno je przechwycić? Wyznaczony wątek? Wszystkie wątki? Nowo utworzony wątek pop-up? Ponadto co się stanie, jeśli jeden wątek zmieni procedury obsługi sygnałów bez informowania pozostałych wątków? A co się wydarzy, kiedy jeden wątek będzie chciał przechwycić określony sygnał (na przykład wciśnięcie przez użytkownika kombinacji *Ctrl+C*), a inny wątek będzie potrzebował tego sygnału do zakończenia procesu? Taka sytuacja może wystąpić, jeśli jeden wątek lub kilka wątków korzysta ze standardowych procedur bibliotecznych, a inne są napisane przez użytkownika. Jest oczywiste, że życzenia tych wątków kolidują ze sobą. Ogólnie rzecz biorąc, sygnały są trudne do zarządzania w środowisku jednowątkowym. Przejście do środowiska wielowątkowego w żaden sposób nie ułatwia zarządzania nimi.

Ostatnim problemem związanym z wątkami jest zarządzanie stosem. W wielu systemach, w przypadku wystąpienia przepełnienia stosu, jądro automatycznie dostarcza takiemu procesowi więcej miejsca na stosie. Jeśli proces ma wiele wątków, musi również mieć wiele stosów. Jeśli jądro nie posiada informacji o wszystkich tych stosach, nie może ich automatycznie rozszerzać, gdy wyczerpie się na nich miejsce. W rzeczywistości jądro może nawet nie wiedzieć, że brak strony w pamięci jest związany z rozszerzeniem się stosu jakiegoś wątku.

Problemy te nie są oczywiście nie do rozwiązania, ale pokazują, że wprowadzenie wątków do istniejącego systemu bez znaczącej jego przebudowy nie zadziała. Trzeba co najmniej zmodyfikować definicję semantyki wywołań systemowych oraz biblioteki. Wszystkie te czynności trzeba dodatkowo wykonać tak, aby zachować wsteczną zgodność z istniejącymi programami, przy założeniu, że wykorzystują one procesy zawierające po jednym wątku. Więcej informacji na temat wątków można znaleźć w następujących pozycjach [Hauser et al., 1993] oraz [Marsh et al., 1991].