Programowanie równoległe i rozproszone

SKRYPT

Krzysztof Banaś

Wydział Fizyki, Matematyki i Informatyki
Politechniki Krakowskiej
Kraków 2011

Materiały dydaktyczne zostały przygotowane w ramach Projektu „Politechnika XXI wieku - Program rozwojowy Politechniki Krakowskiej – najwyższej jakości dydaktyka dla przyszłych polskich inżynierów" współfinansowanego ze środków Unii Europejskiej w ramach Europejskiego Funduszu Społecznego. Umowa o dofinansowanie nr UDA-POKL.04.01.01-00-029/10-00

Spis treści

Spis wydruków kodu

Spis rysunków

Spis tablic

Wstęp

Niniejszy skrypt przeznaczony jest dla wszystkich chcących poznać podstawy i wybrane bardziej zaawansowane elementy programowania współbieżnego, równoległego i rozproszonego. Tematem głównym są obliczenia równoległe, czyli odpowiedź na pytanie jak w praktyce wykorzystywać współczesny sprzęt komputerowy, taki jak np. mikroprocesory wielordzeniowe i klastry komputerów osobistych. Przetwarzanie współbieżne i rozproszone traktowane są jako ważne obszary, w wielu miejscach pokrywające się z tematyką obliczeń równoległych, jednak skrypt nie jest ich wyczerpującą prezentacją.

Skrypt zakłada u Czytelnika pewien poziom wiedzy dotyczącej architektury procesorów i komputerów, systemów operacyjnych, programowania w klasycznych językach proceduralnych i obiektowych. Zakres omawiany na kursach studiów informatycznych pierwszego stopnia powinien być w zupełności wystarczający. Osoby nie studiujące informatyki, a mające pewne doświadczenie programistyczne, także nie powinny mieć kłopotu z korzystaniem ze skryptu.

Założeniem skryptu jest położenie głównego nacisku na rozumienie omawianych zagadnień oraz przedstawienie szeregu przykładów praktycznych. W wielu miejscach (np. w tytułach rozdziałów) pojawiają się w sposób jawnie sformułowany problemy, jakie napotyka się tworząc oprogramowanie współbieżne, równoległe czy rozproszone. Treść rozdziałów szczegółowo prezentuje problemy, analizuje ich możliwe rozwiązania oraz pokazuje przykłady praktycznych rozwiązań.

Do przedstawienia przykładów kodu rozwiązującego problemy, wybrane zostały popularne środowiska programowania, jednakże tylko w wariancie podstawowym. Prezentacja przykładu zazwyczaj obejmuje kod źródłowy wraz z informacją o jego kompilacji, uruchomieniu programu oraz efekcie działania. Celem głównym jest przedstawienie rozwiązania wybranego problemu programistycznego, dlatego też np. fragmenty kodu nie zawierają szczegółowej obsługi błędów. Odpowiednią obsługę błędów, na podstawie szczegółowych specyfikacji omawianych środowisk, należy dodać w przypadku umieszczania kodu w programach użytkowych.

W skrypcie nie są przedstawiane sposoby obsługi zaawansowanych środowisk programowania, w tym programowania równoległego czy rozproszonego. **Filozofia prezentacji jest taka, aby nauczyć dostrzegania problemów przy programowaniu oraz umiejętności rozwiązywania tych problemów.** Narzędzia używane w skrypcie są proste – ktoś, kto rozumie problem i wie jaka jest istota jego rozwiązania, łatwo stworzy właściwy kod posługując się dowolnym narzędziem. **Z założenia zawartość skryptu ma być pomocna dla osób stosujących dowolne narzędzia i środowiska programowania.**

Układ skryptu pomyślany jest jako szereg rozdziałów, w każdym rozdziale występuje pewna myśl przewodnia, pewien problem programistyczny, który rozdział stara się omówić. W zasadzie możliwe jest korzystanie ze skryptu na zasadzie korzystania z wybranych rozdziałów – zależnie od napotkanych problemów, które chce się rozwiązać. Dla ułatwienia takiego stosowania w skrypcie zawarty jest szereg odniesień do zagadnień omawianych gdzie indziej, tak aby np. można było na bieżąco uzupełniać treść pominiętą wcześniej. Rozdziały oznaczone gwiazdką (*) zawierają materiał bardziej zaawansowany, nie stanowiący niezbędnej podstawy do zrozumienia innych zagadnień. Skrypt zawiera także indeks oraz *Słownik użytych tłumaczeń terminów angielskich* – terminologia angielska jest zazwyczaj bardziej

jednoznaczna i ugruntowana niż terminologia polska.

Najważniejszymi elementami skryptu są fragmenty tekstu wyróżnione pogrubionymi literami oraz przykłady kodu. Reszta służy poszerzeniu i objaśnieniu jednego i drugiego. Każdy rozdział kończy się krótkim quizem w postaci testu uzupełnień i wielokrotnego wyboru, który ma na celu sprawdzenie opanowania problematyki rozdziału. W miejscach wykropkowanych należy wpisać właściwe odpowiedzi, kwadratowe ramki należy zaznaczyć przy każdym prawdziwym stwierdzeniu. W jednym pytaniu może być wiele prawdziwych odpowiedzi, jedna lub żadnej.

W skrypcie położony jest nacisk na aspekty praktyczne wiedzy i praktyczne umiejętności – umiejętności poparte zrozumieniem zagadnień. Służyć temu ma duża liczba przykładów (poza rozdziałem pierwszym o bardziej teoretycznym charakterze) oraz ukierunkowanie na rozwiązywanie problemów. W każdym rozdziale znajduje się lista pytań z nim związanych. Z założenia lista jest otwarta. **Każdy czytelnik może wziąć udział w redagowaniu skryptu poprzez przesłanie swojego pytania** (na adres pobanas@cyf-kr.edu.pl)**, odpowiedź na które powinna prędzej czy później znaleźć się na stronach skryptu.**

Rozdział 1, który mówi o tym, po co stosuje się obliczenia współbieżne, równoległe i rozproszone oraz czy można tego uniknąć

1.1 Wprowadzenie

Rozdział niniejszy wprowadza szereg pojęć wykorzystywanych w dalszych częściach skryptu. Przedstawia także szerszy kontekst, w którym funkcjonują obliczenia równoległe i rozproszone. Zaprezentowane są jedynie podstawowe idee, najważniejsze z nich, zgodnie z konwencją skryptu, zaznaczone drukiem pogrubionym. Więcej o historii i modelach przetwarzania równoległego można znaleźć w Rozdziale **??** omawiającym sprzęt do obliczeń równoległych oraz Dodatku **??**.

1.2 Czym jest przetwarzanie współbieżne, równoległe i rozproszone?

Przetwarzanie współbieżne, równoległe i rozproszone to powszechne dziś formy wykonania programów. Programem jest system operacyjny komputera, kompilator języka programowania, maszyna wirtualna interpretująca języki skryptowe czy dowolny program użytkowy realizujący rozmaite funkcje i zadania. Przedmiotem naszego zainteresowania będzie głównie ta ostatnia grupa – programy użytkowe.

Mówiąc o wykonaniu programu rozważamy realizację zbioru rozkazów i instrukcji, zmierzających do rozwiązania postawionego problemu obliczeniowego (rozkazami będziemy nazywali pojedyncze polecenia wykonywane przez procesory, instrukcjami bardziej złożone zadania, zapisywane w językach programowania i tłumaczone na szereg rozkazów procesora). Realizacja zbioru rozkazów odnosi się do procesu zachodzącego w czasie i nie jest jednoznacznie związana z kodem programu. Zbiór rozkazów oznacza zestaw wykonywany przy konkretnym uruchomieniu programu, dla konkretnych danych wejściowych. Każde wykonanie programu może odpowiadać innemu zbiorowi rozkazów (np. zależnie od danych wejściowych programu). Kod występującej w programie pętli zawierającej kilka instrukcji może być realizowany kilka tysięcy lub kilka milionów razy i wykonanie programu odnosić się będzie wtedy do realizacji zbioru złożonego odpowiednio z kilku tysięcy lub kilku milionów instrukcji.

Będziemy rozważać sytuację, kiedy zbiór rozkazów dzielony jest na podzbiory (na przykład po to, żeby każdy podzbiór uruchomić na innym procesorze). W ramach każdego podzbioru rozkazy wykonywane są w jednoznacznej kolejności określonej przez programistę lub przez kompilator tłumaczący kod źródłowy – takie wykonanie będziemy nazywać sekwencyjnym. **Sekwencyjnie wykonywany zbiór rozkazów będziemy nazywali wątkiem**[1]. **O wykonaniu współbieżnym dwóch wątków**

[1] W kolejnych rozdziałach, m.in. przy omawianiu wsparcia systemów operacyjnych dla obliczeń współbieżnych (rozdział

będziemy mówili wtedy, gdy rozkazy jednego wątku zaczną być wykonywane, zanim zakończy się wykonywanie rozkazów drugiego, uruchomionego wcześniej. Sytuację tę ilustrują rysunki 1.1 i 1.2. Na obu z nich, po lewej stronie, wzdłuż biegnącej pionowo osi czasu, widać kolejno wykonywane rozkazy wątku A uruchomionego wcześniej, po prawej znajdują się kolejne rozkazy z wykonywanego współbieżnie wątku B. Wątki A i B mogą być związane z tym samym programem, mogą też należeć do dwóch różnych programów.

Rysunek 1.1: Współbieżne wykonanie dwóch wątków – w przeplocie

Rysunek 1.2: Równoległe wykonanie dwóch wątków

O przetwarzaniu równoległym będziemy mówili wtedy, kiedy przynajmniej niektóre z rozkazów wątków wykonywanych współbieżnie są realizowane w tym samym czasie (jak to zobaczymy później, im więcej jednocześnie wykonywanych rozkazów, tym lepiej). Rys. 1.1 pokazuje sytuację, kiedy wątki wykonywane są współbieżnie, ale nie równolegle. Wykonywanie sekwencyjne wątku A jest przerywane (w terminologii systemów operacyjnych mówimy o wywłaszczeniu wątku), na pewien czas uruchamiane jest wykonywanie rozkazów wątku B, po czym system wraca do realizacji rozkazów wątku A. Takie wykonanie nazywane jest wykonaniem w przeplocie. Nie wymaga wielu jednostek wykonywania rozkazów, może zostać zrealizowane na pojedynczym procesorze, wystarczą odpowiednie możliwości systemu operacyjnego zarządzającego wykonywaniem wątków. Rys. 1.2 przedstawia wykonanie tych samych rozkazów w sposób równoległy. Takie wykonanie wymaga już specjalnego sprzętu, maszyny równoległej. Może to być maszyna z wieloma procesorami (rdzeniami) pracującymi pod kontrolą jednego systemu operacyjnego, może też być zbiór komputerów połączonych siecią, z których każdy posiada własny system operacyjny.

Wnioskiem z obu definicji jest stwierdzenie, że **każde wykonanie równoległe jest wykonaniem współbieżnym, natomiast nie każde wykonanie współbieżne jest wykonaniem równoległym**. Przetwarzanie współbieżne jest pojęciem bardziej ogólnym i ma też bardziej ugruntowaną pozycję w świecie informatyki niż przetwarzanie równoległe. Od wielu już lat każdy uruchomiony program na dowolnym komputerze jest wykonywany współbieżnie z innymi programami. Dzieje się tak za sprawą wielozadaniowych systemów operacyjnych, które zarządzają wykonaniem programów (każdy z popularnych systemów operacyjnych jest systemem wielozadaniowym). W tym wypadku każdy program jest osobnym zbiorem rozkazów i systemy operacyjne pozwalają na współbieżne wykonywanie tych zbiorów. Przeciętnie w jednym momencie typowy współczesny komputer ma uruchomione kilkadziesiąt do kilkuset programów (zdecydowana większość z nich przez zdecydowaną większość czasu przebywa w stanie uśpienia).

Przetwarzanie współbieżne jest od lat omawiane w ramach prezentacji systemów operacyjnych. Niniejszy skrypt nie zajmuje się zagadnieniami specyficznymi dla systemów operacyjnych. Jednak ze względu na to, że każdy program równoległy (i każdy system rozproszony) stosuje przetwarzanie współbieżne, szereg istotnych problemów współbieżności zostanie omówionych w dalszej części skryptu.

Z przetwarzaniem rozproszonym mamy do czynienia wtedy, gdy wątki, składające się na program, wykonywane są na różnych komputerach połączonych siecią[2]. Podobnie jak w przypadku

2), pojęcie wątku zostanie rozwinięte i uściślone

[2]Często, np. dla celów testowania, takie systemy rozproszone uruchamiane są na pojedynczym komputerze (który, jak wiemy, pozwala na współbieżne wykonanie programów). Istotą systemu rozproszonego pozostaje fakt, że **może** on zostać uruchomiony na różnych komputerach połączonych siecią.

wykonania równoległego, także **wykonanie rozproszone jest szczególnym przypadkiem przetwarzania współbieżnego**. Natomiast to, czy system rozproszony będzie wykonywany równolegle, zależy od organizacji obliczeń i pozostaje w gestii programisty. Często pojedynczy program może zostać zakwalifikowany i jako równoległy, i jako rozproszony. O ostatecznej klasyfikacji może zadecydować cel, dla którego zastosowano taką, a nie inną formę przetwarzania współbieżnego.

1.3 Po co przetwarzanie równoległe i rozproszone?

Istnieje kilka podstawowych celów, dla których stosuje się przetwarzanie równoległe i rozproszone. Pierwszym z nich jest zwiększenie wydajności obliczeń, czyli szybsze wykonywanie zadań przez sprzęt komputerowy. Realizowane zadania są rozmaite, zależą od konkretnego programu i konkretnej dziedziny zastosowania (przetwarzanie plików multimedialnych, dynamiczne tworzenie i wyświetlanie stron internetowych, przeprowadzanie symulacji zjawisk i procesów technicznych, analiza plików tekstowych itp.). W wykonywaniu zadań użytkowych może brać udział wiele urządzeń, takich jak monitory, karty sieciowe, twarde dyski. Zawsze jednak, jeśli podstawowym sprzętem przetwarzania jest komputer, mamy do czynienia z wykonywaniem zbiorów rozkazów i instrukcji. Zwiększenie wydajności przetwarzania będzie więc oznaczało realizację przewidzianych w programie rozkazów i instrukcji w krótszym czasie.

Ten cel – zwiększenie wydajności obliczeń i skrócenie czasu rozwiązania konkretnego pojedynczego zadania obliczeniowego – jest podstawowym celem stosowania przetwarzania równoległego. Intuicyjnie wydaje się naturalne, że mając do wykonania pewną liczbę rozkazów i dysponując możliwością uruchomienia dwóch wątków jednocześnie (czyli posiadając dwa procesory), możemy liczyć na zakończenie działania programu w czasie dwa razy krótszym niż w przypadku użycia tylko jednego wątku (procesora). Podobnie mając do dyspozycji cztery procesory chcielibyśmy zakończyć zadanie w czasie cztery razy krótszym, mając osiem w ośmiokrotnie krótszym itd. Kontynuując ten tok myślenia, możemy zadawać pytania:

- **jak maksymalnie skrócić czas wykonania danego programu?**
- **jak budować i wykorzystywać komputery o wielkiej liczbie procesorów?**
- **jakie teoretycznie największe korzyści możemy mieć z przetwarzania równoległego i rozproszonego?**

Tego typu pytania często stawiane są w dziedzinach związanych z obliczeniami wysokiej wydajności, wykorzystaniem superkomputerów, zadaniami wielkiej skali.

Zwiększanie wydajności obliczeń dzięki przetwarzaniu równoległemu ma także aspekt bardziej codzienny, związany ze znaczeniem praktycznym. Chodzi o to, **w jak wielu przypadkach i w jak dużym stopniu przeciętny użytkownik komputerów może uzyskać znaczące zyski z zastosowania obliczeń równoległych?** Odpowiedź na te pytania zależy od tego, jak wiele osób ma dostęp do sprzętu umożliwiającego efektywne przetwarzanie równoległe. Na potrzeby naszych rozważań przyjmiemy, że **przetwarzanie równoległe można uznać za efektywne, jeśli pozwala na co najmniej kilku-, kilkunastokrotne skrócenie czasu realizacji zadań w stosunku do przetwarzania sekwencyjnego**. Czy sprzęt dający takie możliwości jest powszechnie używany?

W ostatnich latach dostępność wysoko efektywnego sprzętu równoległego znacznie się zwiększyła i, jak wiele na to wskazuje, kolejne lata będą przynosiły dalszy postęp. Kilka, kilkanaście lat temu sprzęt równoległy był na tyle drogi, że wykorzystywały go głównie instytucje rządowe, wielkie firmy i centra naukowe. **Pierwszym krokiem na drodze do upowszechnienia sprzętu równoległego było wprowadzenie klastrów**, zespołów komputerów osobistych połączonych siecią, relatywnie tanich w

porównaniu z komputerami masowo wieloprocesorowymi[3] i dających porównywalne zyski czasowe. Obecnie klastry stają się standardowym wyposażeniem także w małych i średnich firmach, znacząco rośnie liczba korzystających z nich osób. Takie małe klastry zawierają kilkanaście, kilkadziesiąt procesorów i pozwalają na przyspieszanie obliczeń (skrócenie czasu wykonywania zadań) w podobnym zakresie – kilkanaście, kilkadziesiąt razy. Na bardzo zbliżonych zasadach konstrukcyjnych i programowych opierają się wielkie superkomputery pozwalające na uzyskiwanie przyspieszeń rzędu dziesiątek i setek tysięcy.

Drugim, znacznie ważniejszym z punktu widzenia popularyzacji obliczeń równoległych, procesem zachodzącym w ostatnich latach jest zmiana architektury mikroprocesorów.

Krótkie przypomnienie – architektura von Neumanna. Sposób przetwarzania rozkazów przez procesory, tak dawne jak i współczesne, najlepiej charakteryzowany jest przez model maszyny von Neumanna. Wymyślony w latach 40-tych XX wieku, stał się podstawą konstrukcji pierwszych komercyjnie sprzedawanych komputerów i do dziś jest podstawą budowy wszelkich procesorów oraz stanowi fundament rozumienia metod przetwarzania rozkazów we wszystkich komputerach. Konkretny sposób realizacji rozkazów przez procesory z biegiem lat stawał się coraz bardziej złożony[4], jednak jego istota nie zmieniła się i pozostaje taka sama jak w pierwotnej architekturze von Neumanna.

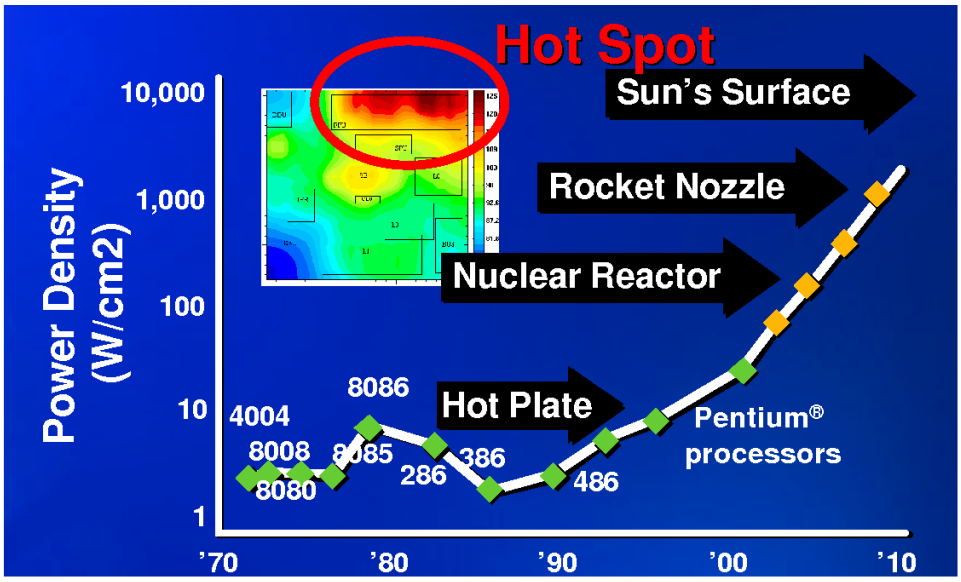
Istotą maszyny von Neumanna, odróżniającą ja od innych współczesnych jej modeli przetwarzania rozkazów, jest wykorzystanie jednej pamięci operacyjnej, w której znajdują się zakodowane (obecnie zawsze w sposób binarny) i kod programu, i dane na których operuje kod. Pamięć operacyjna składa się z komórek przechowujących dane, do których dostęp możliwy jest poprzez jednoznaczny adres, jaki posiada konkretna komórka. Adres jest dodatnią liczbą całkowitą, złożoną z takiej liczby bitów, którą wygodnie jest przetwarzać procesorowi. W miarę rozwoju procesorów długość tej liczby rosła (jest to związane m.in. z liczba bitów podstawowych rejestrów procesora i z szerokością magistrali łączącej procesor z pamięcią), zaczynając od kilku bitów, aż po dominujące dziś procesory 64-bitowe.

Wykonanie programu to przetwarzanie rozkazów zapisanych w pamięci operacyjnej komputera. Przetwarzanie pojedynczego rozkazu składa się z szeregu faz. Liczba faz zależy od konkretnego rozkazu, np. od tego czy operuje na danych pobieranych z pamięci, czy zapisuje wynik w pamięci itp. Kilka faz występuje we wszystkich rozkazach. Są to:

- pobranie rozkazu z pamięci (ang. *fetch*)
- dekodowanie rozkazu (ang. *decode*)
- wykonanie rozkazu (ang. *execute*)
- pewna forma zapisu efektu realizacji rozkazu (ang. *write back*)

W przypadku rozkazów operujących na danych z pamięci, dochodzi do tego jeszcze dostęp do pamięci: pobranie argumentów lub ich zapis (zazwyczaj współczesne procesory nie wykonują rozkazów, które pobierałyby argumenty z pamięci **i** zapisywałyby wynik do pamięci). Jako wsparcie działania systemów operacyjnych pojawia się jeszcze faza sprawdzenia, czy nie wystąpiło przerwanie.

[3] Komputerami masowo wieloprocesorowymi (ang. *massively parallel processors, MPP*) nazywamy komputery wyposażone w dużą liczbę (co najmniej kilkaset) procesorów, posiadających własne, niezależne pamięci operacyjne i połączonych szybką, zaprojektowaną specjalnie na potrzeby danego komputera, siecią.



Rysunek 1.3: Gęstość wydzielania ciepła [źródło: Intel, 2001]

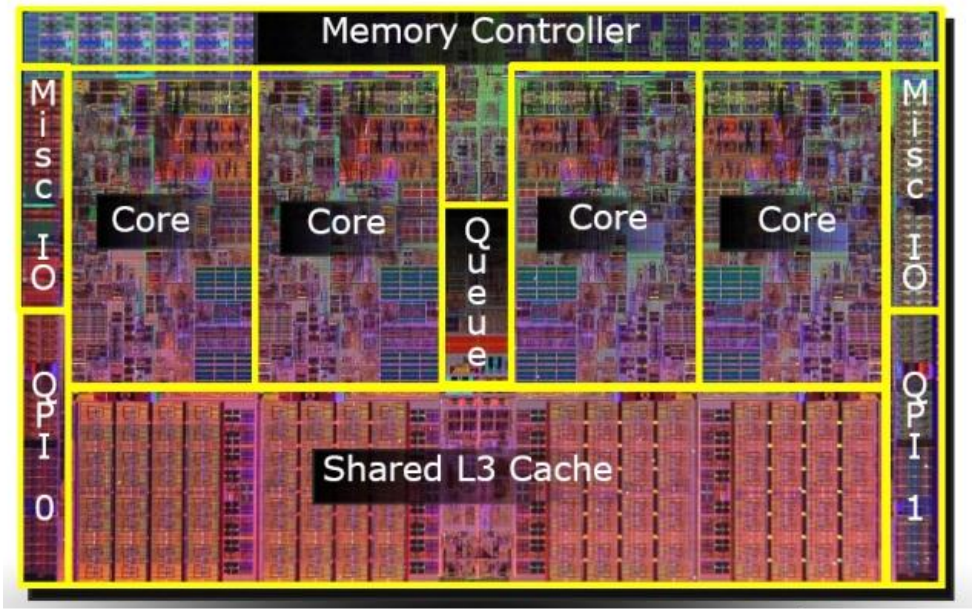
Zanim omówimy następstwa zmiany architektury mikroprocesorów w ostatnich latach, konieczne jest poczynienie kilku uwag odnośnie terminologii. Pierwotnie **procesorem nazywano jednostkę centralną komputera, która w sposób sekwencyjny wykonuje rozkazy pobierane z pamięci operacyjnej**. Tak też stosowane było to określenie dotychczas w skrypcie – procesor służył w pewnej konkretnej chwili do realizacji pojedynczego wątku.

W późniejszych latach nazwą procesor objęto także **mikroprocesory – pojedyncze układy scalone realizujące funkcje procesora**. Wprowadzone w latach 70-tych XX wieku, jako konkurencja dla układów stosujących obwody drukowane, są dziś jedyną formą procesorów. Będące początkowo prostymi układami zawierającymi kilka tysięcy tranzystorów, mikroprocesory w miarę upływu lat stawały się coraz bardziej złożone. Postęp elektroniki powodował, że rozmiar pojedynczego tranzystora w układzie scalonym stawał się coraz mniejszy, a ich liczba coraz większa. W pewnym momencie, w pojedynczym układzie scalonym zaczęto umieszczać, oprócz układów bezpośrednio realizujących przetwarzanie rozkazów, nazywanych rdzeniami mikroprocesora, także układy pamięci podręcznej czy układy komunikacji mikroprocesora ze światem zewnętrznym. Przez ponad trzydzieści lat rdzenie stawały się coraz bardziej złożone, a mikroprocesory coraz bardziej wydajne.

Na początku XXI-go wieku okazało się, że budowanie jeszcze bardziej złożonych rdzeni prowadzi do nadmiernego wydzielania ciepła przez mikroprocesory. Ilustruje to rys. 1.3, na którym porównane są gęstości wytwarzania ciepła procesorów i kilku wybranych urządzeń, takich jak płyta kuchenki elektrycznej, reaktor nuklearny i dysza silnika rakietowego, a także gęstość wydzielania ciepła na powierzchni słońca.

W konsekwencji nierozwiązania problemów z odprowadzaniem ciepła z coraz szybszych mikroprocesorów jednordzeniowych, producenci zdecydowali się na umieszczenie w pojedynczym układzie scalonym wielu rdzeni[5]. Powstały mikroprocesory wielordzeniowe, które dziś są już praktycznie jedyną formą mikroprocesorów. Rys. 1.4 przedstawia typowy współczesny mikroprocesor wielordzeniowy. Widać na nim rdzenie oraz układy pamięci podręcznej, a także inne elementy, w skład których wchodzą między innymi układy sterowania dostępem do pamięci oraz komunikacji ze światem zewnętrznym.

[5]Pierwszym wielordzeniowym mikroprocesorem ogólnego przeznaczenia był układ Power4 firmy IBM z 2001 roku

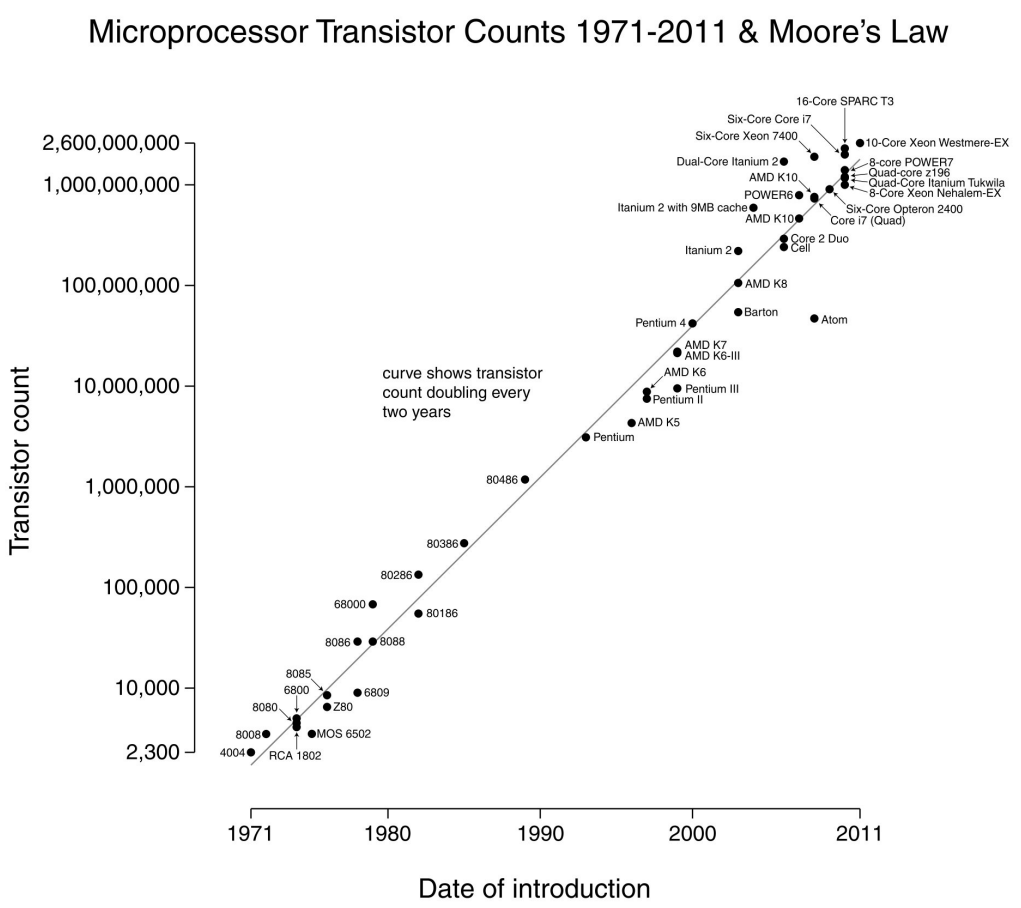


Rysunek 1.4: Procesor wielordzeniowy - widoczne rdzenie, układy pamięci podręcznej oraz inne elementy, m.in. układy sterowania dostępem do pamięci oraz komunikacji ze światem zewnętrznym. [źródło: Intel, 2010]

Określenia *rdzeń* i *mikroprocesor wielordzeniowy* mają ustalone, powszechnie przyjęte znaczenia. Natomiast pierwotny termin *procesor* bywa używany dwojako – w sensie pierwotnym, a więc jako synonim rdzenia, pojedynczej jednostki przetwarzania rozkazów, lub w sensie mikroprocesora, a więc obecnie jako synonim mikroprocesora wielordzeniowego. Pierwsze znaczenie odnosi się w większym stopniu do przetwarzania, a więc i do programowania, natomiast to drugie do elektroniki i budowy układów scalonych. Dlatego **w niniejszym skrypcie określenie *procesor* stosowane będzie głównie jako synonim określenia *rdzeń* – układ do przetwarzania ciągów (sekwencji) rozkazów, a więc układ sekwencyjny. Równoległym układem będzie natomiast mikroprocesor wielordzeniowy, jako w rzeczywistości system wieloprocesorowy w pojedynczym układzie scalonym**[6].

Rozwój mikroprocesorów przez ostatnie ponad czterdzieści lat możliwy był dzięki postępowi w dziedzinie elektroniki, postępowi, który najlepiej charakteryzuje prawo Moore'a (rys. 1.5). Zostało ono sformułowane przez założyciela firmy Intel, na podstawie zaobserwowanych tendencji, już w roku 1965. W formie odzwierciedlającej rzeczywisty rozwój technologii w ostatnich kilkudziesięciu latach, można je sformułować następująco: **liczba tranzystorów standardowo umieszczanych w pojedynczym układzie scalonym podwaja się w okresie każdych dwóch lat**. Obecnie postęp technologiczny w tworzeniu układów scalonych, który najprawdopodobniej trwać będzie w tempie zgodnym z prawem Moore'a jeszcze przez najbliższe kilka, kilkanaście lat, przekłada się na zwiększanie liczby rdzeni w pojedynczym mikroprocesorze wielordzeniowym. Można więc spodziewać się, że jeśli dziś mamy powszechnie do czynienia z mikroprocesorami czterordzeniowymi, to za dwa lata powszechne będą mikroprocesory ośmiordzeniowe. To oznacza, że najprawdopodobniej już **niedługo nastąpi era**

[6]Procesor jest także określeniem używanym w kontekście działania systemów operacyjnych, jako pojedyncza jednostka, której przydzielany jest do wykonania strumień rozkazów. W tym przypadku nawet mikroprocesory jednordzeniowe, ale wyposażone w tzw. *simultaneous multithreading*, zwany także *hyperthreading*, widziane są przez system operacyjny jako dwa lub więcej procesory (więcej na ten temat w skrypcie "Obliczenia wysokiej wydajności").



Rysunek 1.5: Prawo Moore'a i liczba tranzystorów rzeczywistych mikroprocesorów [źródło: Wikipedia, 2011]

mikroprocesorów masowo wielordzeniowych, mikroprocesorów z liczbą rdzeni rzędu kilkunastu, kilkudziesięciu i więcej.

W efekcie **każdy właściciel komputera będzie posiadał do dyspozycji maszynę zdolną do efektywnej pracy równoległej**. Czy można się spodziewać, że w takiej sytuacji ktoś może zdecydować się na korzystanie z programów sekwencyjnych zamiast programów równoległych, dobrowolnie zrezygnować z możliwości wykonywania programów kilkanaście razy szybciej? Nie jest to prawdopodobne i częściowo odpowiada na postawione wcześniej pytanie o przydatność przetwarzania równoległego. **Skoro sprzęt równoległy staje się tak masowy, równie masowe powinny być efektywne programy równoległe**. Z tej perspektywy wydaje się, że **żaden twórca oprogramowania nie może uniknąć konieczności pisania programów równoległych**. [7]

Jak zwykle, w przypadku kiedy pojawia się konieczność stosowania określonego paradygmatu

[7] Tylko w jednym przypadku zrozumiałe jest korzystanie z programów sekwencyjnych – wtedy kiedy nie da się stworzyć algorytmu równoległego do rozwiązania danego problemu obliczeniowego. Ale także w tym przypadku, żeby jednoznacznie dojść do takiego wniosku, trzeba rozumieć na czym polegają obliczenia równoległe i dlaczego konkretny problem nie pozwala na rozwiązanie równoległe. Co oznacza, że także w tym przypadku trzeba uprzednio przynajmniej podjąć próbę stworzenia programu równoległego.

programowania, pojawiają się także służące do tego narzędzia. Niniejszy skrypt omawia wiele takich narzędzi, koncentruje się jednak na narzędziach relatywnie niskiego poziomu, których użycie najczęściej wymaga zrozumienia funkcjonowania systemów równoległych. Istnieją środowiska programowania oferujące narzędzia (najczęściej biblioteki) służące tworzeniu programów równoległych, przy jak najmniejszym wysiłku ze strony programisty. Wielokrotnie jednak przy ich stosowaniu ujawnia się jedna z zasadniczych cech programowania równoległego: **nie jest trudno stworzyć program równoległy – program, w którym występuje wiele jednocześnie wykonywanych strumieni rozkazów. Znacznie trudniej jest stworzyć *poprawny* program równoległy – program, który przy każdym wykonaniu będzie zwracał poprawne rezultaty. Najtrudniej jest stworzyć *wysoko wydajny* program równoległy – program, który w swoim działaniu będzie osiągał skrócenie czasu przetwarzania zbliżone do liczby zastosowanych procesorów**. Do realizacji tego ostatniego celu, nawet w przypadku stosowania narzędzi tworzenia programów równoległych wysokiego poziomu, przydatne, a czasem konieczne jest dogłębne zrozumienie zasad działania programów i komputerów równoległych.

Drugim z celów stosowania obliczeń równoległych i rozproszonych jest zwiększenie niezawodności przetwarzania. Dotyczy to głównie dużych systemów, w szczególności takich, które powinny pracować w sposób ciągły. Standardowym sposobem przeciwdziałania skutkom rozmaitych awarii jest powielanie – tworzenie kopii zapasowych, zapewnianie powtarzalności operacji. Przetwarzanie równoległe może służyć zapewnieniu niezawodności – np. poprzez wykorzystanie kilku pracujących jednocześnie i wykonujących te same funkcje wątków, w taki sposób, że gdy jeden ulega awarii pozostają inne, świadcząc odpowiednie usługi, aż do czasu naprawy pierwszego wątku. Do zapewnienia niezawodności szczególnie dobrze nadaje się przetwarzanie rozproszone – jeśli mamy kilka wątków na kilku komputerach, awaria jednego z komputerów nie musi prowadzić do zaprzestania dostarczania usług.

Wreszcie trzecim z celów stosowania obliczeń równoległych i rozproszonych, tym razem dotyczącym głównie przetwarzania rozproszonego, jest zwiększenie elastyczności wykorzystania dostępnych zasobów komputerowych. Koronnym przykładem takiego sposobu funkcjonowania programów są systemy w ramach tzw. ”Grid Computing” i ”Cloud Computing”. W systemach tych użytkownik, korzystający ze swojej lokalnej maszyny połączonej siecią z innymi zasobami, zleca realizację pewnej usługi. System sam dobiera jakie konkretnie oprogramowanie i na jakim sprzęcie zrealizuje usługę.

Zasady działania powyższych systemów omówione są w tomie II niniejszego skryptu. W tomie niniejszym przedstawione są tylko podstawy funkcjonowania systemów rozproszonych, podstawowe problemy z tym związane i wybrane sposoby rozwiązywania tych problemów.

1.4 Pytania

Nie planuję programować równolegle – będę korzystać ze środowisk dostarczających biblioteki procedur wielowątkowych, czy wiedza o obliczeniach równoległych jest mi potrzebna?

Korzystanie z gotowych procedur wielowątkowych lub obiektowych struktur danych obsługiwanych przez procedury wielowątkowe staje się coraz popularniejsze, ze względu na konieczność stosowania obliczeń wielowątkowych i powszechne przekonanie o dużym stopniu trudności programowania równoległego. Nawet taka realizacja obliczeń równoległych wymaga pewnej wiedzy. Pojęciem często pojawiającym się przy opisie bibliotek wielowątkowych jest pojęcie *bezpieczeństwa wielowątkowego* (ang. *thread safety*), czyli zagwarantowania poprawności programu w przypadku wykonania wielowątkowego. Fragment kodu nazywać będziemy *wielowątkowo bezpiecznym*, jeżeli może on być wykonywany przez wiele współbieżnie pracujących wątków bez

ryzyka błędnej realizacji programu lub programów, w ramach których funkcjonują wątki. Nie tylko procedury zawierające jawnie kod wielowątkowy, ale także procedury wywoływane przez współbieżnie wykonywane wątki, powinny być bezpieczne wielowątkowo. Po to, aby wiedzieć, kiedy wykonanie danego fragmentu kodu jest bezpieczne, tzn. kiedy nie prowadzi do błędów wykonania, należy znać zasady realizacji obliczeń wielowątkowych i ogólniej współbieżnych.

Drugim przypadkiem, kiedy wiedza o przetwarzaniu równoległym może okazać się przydatna, jest sytuacja, niestety bardzo częsta, kiedy zastosowany kod wielowątkowy okazuje się być znacznie wolniejszy niż było to planowane (w ostateczności może okazać się wolniejszy niż kod sekwencyjny). Wiedza o zasadach realizacji obliczeń równoległych może pomóc odpowiednio skonfigurować środowisko wykonania tak, aby zdecydowanie zmienić wydajność wykonania równoległego, także w przypadku dostarczanych, gotowych procedur.

1.5 Test

- Przetwarzanie sekwencyjne w standardowych współczesnych systemach komputerowych:
  - ☐ jest sposobem przetwarzania w pojedynczym wątku
  - ☐ jest sposobem przetwarzania w pojedynczym procesie
  - ☐ może być realizowane tylko na mikroprocesorach jednordzeniowych
  - ☐ nie może być założone (przy braku jawnych mechanizmów synchronizacji) jako sposób realizacji procesu wielowątkowego
- Przetwarzanie współbieżne w standardowych współczesnych systemach komputerowych:
  - ☐ oznacza wykonywanie dwóch zbiorów rozkazów w taki sposób, że czasy wykonania nakładają się (nowe zadania zaczynają się zanim stare zostaną zakończone)
  - ☐ jest synonimem przetwarzania równoległego
  - ☐ nie daje się zrealizować w systemach jednoprocesorowych (jednordzeniowych)
  - ☐ służy głównie zwiększeniu stopnia wykorzystania sprzętu
- Przetwarzanie równoległe w standardowych współczesnych systemach komputerowych:
  - ☐ jest synonimem przetwarzania współbieżnego
  - ☐ jest synonimem przetwarzania wielowątkowego
  - ☐ nie daje się zrealizować w systemach jednoprocesorowych (jednordzeniowych)
  - ☐ służy głównie zwiększeniu wydajności przetwarzania
  - ☐ służy głównie zwiększeniu niezawodności przetwarzania
- Przetwarzanie rozproszone w standardowych współczesnych systemach komputerowych:
  - ☐ umożliwia zwiększenie wydajności przetwarzania
  - ☐ umożliwia zwiększenie niezawodności przetwarzania
  - ☐ oznacza uruchamianie programów na różnych komputerach połączonych siecią
  - ☐ jest szczególnym przypadkiem przetwarzania współbieżnego
  - ☐ jest szczególnym przypadkiem przetwarzania równoległego

- Przetwarzanie w przeplocie oznacza sytuacje kiedy:
  - □ system operacyjny przydziela wątki tego samego zadania różnym rdzeniom (procesorom)
  - □ system operacyjny realizuje przetwarzanie współbieżne na jednym procesorze (rdzeniu)
  - □ procesor (rdzeń) stosuje *simultaneous multithreading*
  - □ pojedynczy procesor (rdzeń) na przemian wykonuje fragmenty wielu wątków
  - □ pojedynczy proces korzysta na przemian z wielu rdzeni
- Zwiększenie wydajności obliczeń (skrócenie czasy realizacji zadań przez systemy komputerowe) jest głównym celem przetwarzania:
  - □ współbieżnego
  - □ równoległego
  - □ rozproszonego
  - □ wielowątkowego
- Zgodnie z prawem Moore'a liczba tranzystorów umieszczanych w pojedynczym układzie scalonym podwaja się co ... miesięcy.
- Prawo Moore'a stwierdza, że co stałą liczbę miesięcy podwaja się:
  - □ liczba tranzystorów umieszczanych w pojedynczym układzie scalonym
  - □ wydajność mikroprocesorów
  - □ częstotliwość taktowania zegara mikroprocesora
  - □ rozmiar pamięci podręcznej mikroprocesora
- Zgodnie z prawem Moore'a i współczesnymi kierunkami rozwoju architektur mikroprocesorów liczba rdzeni standardowego mikroprocesora w roku 2022 osiągnie:
  □ kilka □ kilkanaście □ kilkadziesiąt □ kilkaset (ponad 100)
- Zgodnie z prawem Moore'a i współczesnymi kierunkami rozwoju architektur mikroprocesorów liczba rdzeni standardowego mikroprocesora osiągnie kilkadziesiąt (ponad 20) około roku:
  □ 2013 □ 2016 □ 2019 □ 2022
- Producenci procesorów ogólnego przeznaczenia przestali zwiększać częstotliwość pracy procesorów z powodu:
  - □ zbyt wielu etapów w potokowym przetwarzaniu rozkazów, utrudniających zrównoleglenie kodu
  - □ zbyt wysokiego poziomu wydzielania ciepła
  - □ barier technologicznych w taktowaniu układów elektronicznych
  - □ niemożności zagwarantowania odpowiednio szybkiej pracy pamięci podręcznej
- Płyta kuchenki elektrycznej produkuje ok. 10 W/cm2 natomiast współczesne procesory około ... W/cm2
- Rdzeń mikroprocesora wielordzeniowego:
  - □ jest częścią mikroprocesora odpowiedzialną za pobieranie i dekodowanie rozkazów, ale nie wykonuje rozkazów (jednostki funkcjonalne znajdują się poza rdzeniem)

- ☐ jest bardzo zbliżony do dawnych procesorów jednordzeniowych, ale nie potrafi wykonywać bardziej złożonych rozkazów
- ☐ jest bardzo zbliżony do dawnych procesorów jednordzeniowych, ale bez pamięci podręcznej L2 i L3
- ☐ wydziela znacznie mniej ciepła niż procesor jednordzeniowy o tej samej częstotliwości pracy

- Bezpieczeństwo wielowątkowe (*thread safety*) procedury oznacza, że (zaznacz tylko odpowiedzi ujmujące istotę bezpieczeństwa wielowątkowego, a nie każde prawdziwe stwierdzenie):
  - ☐ może ona być wykonywana przez dowolnie dużą liczbę wątków
  - ☐ można w niej tworzyć wątki
  - ☐ może być wykonywana przez wiele współbieżnych wątków bez wprowadzania błędów wykonania
  - ☐ może być procedurą startową wątków
  - ☐ inne procedury z innych wątków nie mogą zakłócić jej prawidłowego przebiegu

Rozdział 2, który przedstawia wsparcie ze strony systemów operacyjnych dla obliczeń współbieżnych, równoległych i rozproszonych

2.1 Wprowadzenie

Niniejszy rozdział przedstawia kilka podstawowych mechanizmów, za pomocą których systemy operacyjne umożliwiają tworzenie programów równoległych i rozproszonych. Mechanizmami tymi są tworzenie nowych procesów, nowych wątków, a także zarządzanie procesami i wątkami. Ważnym elementem wsparcia obliczeń równoległych i rozproszonych przez systemy operacyjne jest umożliwienie komunikacji pomiędzy procesami i watkami. Zagadnienia prezentowane w niniejszym rozdziale nie wyczerpują całości tematyki, wykorzystanie mechanizmów systemowych w przetwarzaniu równoległym i rozproszonym będzie pojawiać się także w dalszych rozdziałach skryptu.

2.2 Procesy

Proces to jedno z podstawowych pojęć w dziedzinie systemów operacyjnych. **Najogólniejsza definicja mówi, że proces jest to program w trakcie wykonania.** Analizując bardziej szczegółowo, można powiedzieć, że pojecie procesu zawiera w sobie wiele mechanizmów, za pomocą których systemy operacyjne zarządzają wykonaniem programów. W niniejszym skrypcie mechanizmy te nie zostaną szczegółowo omówione, skupimy się tylko na kilku aspektach najbardziej interesujących z punktu widzenia praktyki programowania równoległego i rozproszonego.

Każdy proces związany jest z wykonywalnym plikiem binarnym zawierającym kod programu oraz z wątkami wykonującymi rozkazy z tego pliku[1]. Jeszcze do niedawna zdecydowana większość procesów była procesami jednowątkowymi, istniał tylko jeden ciąg wykonywanych rozkazów związany z realizacją danego programu. Pojęcie wątku praktycznie nie istniało, wystarczało pojęcie procesu. Obecnie rozkazy z pliku wykonywalnego są coraz częściej wykonywane przez wiele współbieżnie pracujących wątków. Dlatego pojęcie procesu przestaje być jednoznaczne i zazwyczaj wymaga dalszej charakterystyki – czy mamy na myśli proces jednowątkowy czy wielowątkowy? Odpowiedź na to pytanie decyduje o szczegółach sposobu zarządzania realizacją procesu przez systemy operacyjne.

System operacyjny zarządza realizacją procesu m.in. poprzez przydzielenie odpowiedniego obszaru pamięci operacyjnej komputera, zarządzanie dostępem do pamięci, dostępem do urządzeń

[1] Możliwe jest zawarcie kodu wykonywanego w ramach pojedynczego procesu w wielu plikach binarnych, jak to ma miejsce np. w przypadku użycia bibliotek ładowanych dynamicznie. Szczegóły tego zagadnienia należą do dziedziny systemów operacyjnych.

wejścia-wyjścia, a także poprzez nadzorowanie przydziału procesora. W celu realizacji powyższych zadań system operacyjny tworzy dla każdego procesu złożoną strukturę danych, zwaną blokiem kontrolnym procesu, zawierającą między innymi:

- identyfikator procesu – jednoznaczny numer procesu w ramach systemu operacyjnego
- informacje o plikach, z których korzysta proces, o połączeniach sieciowych nawiązanych przez proces oraz o innych urządzeniach wejścia-wyjścia powiązanych z procesem
- informacje o zasobach pamięci operacyjnej przydzielonych procesowi
- zawartość rejestrów wykorzystywanych przez proces (wartości te zapisywane są w strukturze danych w momencie kiedy proces jest wywłaszczany i z powrotem kopiowane do rejestrów w momencie kiedy proces ponownie uzyskuje dostęp do procesora)
- szereg innych informacji związanych z wykonaniem procesu (m.in. stan procesu, jego priorytet, ilość zużytego czasu procesora i czasu rzeczywistego itp.)

Szczególnie istotny jest sposób zarządzania korzystaniem z pamięci operacyjnej. **W celu zapewnienia bezpieczeństwa wielu współbieżnie wykonywanym procesom, system operacyjny przydziela każdemu z nich nie zachodzące na siebie obszary pamięci operacyjnej.** Sposób zarządzania korzystaniem przez proces z pamięci operacyjnej może być różny dla różnych systemów operacyjnych (najpopularniejszą strategią jest wirtualna pamięć stronicowana, systemy operacyjne stosują także segmentację pamięci). Zazwyczaj całość pamięci przydzielonej procesowi składa się z szeregu drobnych fragmentów, dynamicznie alokowanych w pamięci fizycznej, a także na dysku twardym. W niniejszym skrypcie fakt ten będzie pomijany i w celu ilustracji działania procesów i wątków, całość pamięci przydzielonej procesowi traktowana będzie jako pojedynczy blok, nazywany przestrzenią adresową procesu. **Każdy proces posiada odrębną przestrzeń adresową, każdy proces może dokonywać odczytu i zapisu komórek pamięci tylko w swojej przestrzeni adresowej, próba dostępu do przestrzeni adresowej innego procesu kończy się przerwaniem wykonania procesu** (częsty błąd *segmentation fault* w systemie Unix).

Każda komórka w przestrzeni adresowej procesu posiada swój adres wykorzystywany w trakcie wykonania programu (dotyczy to tak komórek zawierających kod programu, jak i komórek zawierających dane programu). W dalszych analizach będziemy pomijać fakt czy adres komórki, o której mówimy jest adresem wirtualnym czy rzeczywistym - translacja adresów jest zadaniem systemu operacyjnego współpracującego z odpowiednimi układami procesora. **Z punktu widzenia wykonywanego programu, chcąc uzyskać dostęp do pamięci stosujemy adres komórki pamięci.** Szczegóły dostępu zaczną odgrywać istotną rolę dopiero wtedy, gdy będziemy dążyć do optymalizacji szybkości wykonania programu.

Co zawiera przestrzeń adresowa procesu? Składa się ona z obszarów o różnym charakterze, które w przybliżeniu (nie oddając całej złożoności wykorzystania pamięci przez proces, który może np. korzystać z współdzielonych biliotek) można sklasyfikować jako:

- obszar kodu programu, czyli zawartość pliku wykonywalnego
- obszar danych statycznych – istniejących przez cały czas trwania procesu
- stos
- stertę – obszar danych w sposób jawny dynamicznie alokowanych przez program (np. funkcje *malloc* lub *new*)

Krótkie przypomnienie – stos. Stos pełni szczególną rolę w trakcie wykonania programu. Jest obszarem o zmiennym rozmiarze i umożliwia sprawną realizacje wywoływania procedur (funkcji), w tym także procedur rekurencyjnych. Jego szczegółowa budowa jest różna w różnych systemach operacyjnych, jednak zasada funkcjonowania jest zawsze taka sama. **Na stosie umieszczane są wartości argumentów wykonywanych procedur oraz wartości ich zmiennych lokalnych (automatycznych).** Każda procedura w klasycznych językach programowania, posiada w kodzie źródłowym nagłówek zawierający listę argumentów. Kod źródłowy procedury określa także, zgodnie z zasadami użytego języka programowania, które zmienne w kodzie są zmiennymi lokalnymi procedury, czyli zmiennymi, których użycie możliwe jest tylko w obrębie danej procedury, zmiennymi, które w programie nie są widoczne poza ta procedurą (w niektórych językach programowania istnieją zmienne, których tak określony zasięg widoczności – i co za tym idzie, czas życia – jest jeszcze mniejszy, ograniczony do fragmentu procedury).

W momencie wywołania procedury, system zarządzający wykonaniem programu alokuje dodatkowy obszar na stosie, przeznaczony do obsługi wywołania. Do obszaru tego kopiowane są wartości argumentów procedury, rezerwowane jest miejsce dla zmiennych automatycznych, niektóre z nich inicjowane są odpowiednimi wartościami. Wartości argumentów kopiowane są z lokalizacji określonej w procedurze wywołującej – mogą to być np. zmienne z obszaru statycznego lub zmienne z innego obszaru stosu. Dzięki kopiowaniu argumentów w momencie wywołania, procedura wywoływana może pracować na swojej kopii danych, odrębnej od zmiennych użytych w trakcie wywołania przez procedurę wywołującą. O argumentach procedury i jej zmiennych lokalnych można powiedzieć, że są zmiennymi "prywatnymi" procedury - pojęcie to ma szczególne znaczenie przy wykonaniu wielowątkowym i zostanie później uściślone.

Na mocy konwencji przyjętej przez większość języków programowania, w trakcie realizacji procedury proces ma dostęp do zmiennych w związanym z procedurą fragmencie stosu (dostęp do tych zmiennych możliwy jest tylko w trakcie wykonania tej właśnie procedury) oraz do zmiennych, które nazywać będziemy wspólnymi (do tych zmiennych dostęp możliwy jest w trakcie wykonania różnych procedur, zmienne te obejmują zmienne statyczne i globalne programu). Sytuacja komplikuje się, kiedy mamy do czynienia ze wskaźnikami[2]. Wskaźnikiem jest zmienna zawierająca adres innej zmiennej, a więc dająca dostęp do tej zmiennej. Przesłanie wskaźnika jako argumentu pozwala jednej procedurze na uzyskanie dostępu do dowolnych danych, w tym do danych lokalnych innej procedury. Ta komplikacja obrazu wykonania programu jest jednak konieczna – bez niej elastyczność funkcjonowania programów byłaby znacznie mniejsza. Przesyłanie wskaźników (referencji) jest np. podstawowym, jeśli praktycznie nie jedynym, sposobem dostępu do sterty – do zmiennych jawnie, dynamicznie alokowanych przez program.

2.3 Tworzenie procesów i wielozadaniowe systemy operacyjne

Wszystkie współczesne systemy operacyjne są systemami wielozadaniowymi. Sam system operacyjny, też będący programem, wykonywany jest jako proces. Po to, aby system operacyjny mógł rozpocząć wykonanie dowolnego programu musi posiadać zdolność utworzenia nowego procesu, powiązania go z plikiem wykonywalnym i przekazania do niego sterowania, czyli przydzielenia procesowi dostępu do procesora. Nie wchodząc w szczegóły realizacji tych zadań, możemy zobaczyć jak tworzenie nowego procesu można zrealizować w dowolnym programie. Posłużymy się do tego kodem w języku C i procedurami systemowymi dostępnymi w rozmaitych wariantach systemu operacyjnego Unix.

Kod 2.1: Prosty program, w którym proces nadrzędny uruchamia proces potomny

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

int zmienna_globalna=0;

main(){

  int pid, wynik;

  pid = fork();
  if(pid==0){

    zmienna_globalna++;
    printf("Proces potomny: zmienna globalna = %d\n", zmienna_globalna);

    wynik = execl("/bin/ls", "ls", ".", (char *) 0);
    if(wynik == -1) printf("Proces potomny nie wykonał polecenia ls\n");
  } else {

    wait(NULL);
    printf("Proces nadrzędny: proces potomny zakończył działanie\n");

    printf("Proces nadrzędny: zmienna globalna = %d\n", zmienna_globalna);
  }
}
```

Kod 2.1 wykorzystuje obecną od wielu lat w systemie Unix procedurę fork. Co dzieje się w momencie wywołania funkcji fork w programie? Funkcja wywołana zostaje w ramach procesu realizującego skompilowany kod 2.1 (po kompilacji kodu, uruchomienie programu związane jest z utworzeniem procesu, którego wątek główny, i w tym momencie jedyny, rozpoczyna realizacje funkcji main kodu). Proces ten jest nazywany *procesem nadrzędnym* lub *procesem macierzystym*. W momencie wywołania funkcji fork system operacyjny tworzy nowy proces, *proces potomny*[3]. Proces ten jest w zasadzie kopią procesu nadrzędnego – posiada wprawdzie własny identyfikator, ale jego blok kontrolny powstaje zazwyczaj przez skopiowanie (przynajmniej częściowe, niektóre elementy zostają wyzerowane) bloku kontrolnego procesu nadrzędnego. Proces potomny teoretycznie posiada własną przestrzeń adresową, będącą kopią przestrzeni adresowej procesu nadrzędnego. W rzeczywistości w celu zwiększenia efektywności wykonania, kopiowanie nie zawsze jest realizowane i procesy mogą współdzielić pewne obszary pamięci. System operacyjny musi jednak tak zarządzać korzystaniem z pamięci, aby utrzymać wrażenie posiadania odrębnych, niezależnych przestrzeni adresowych przez oba procesy.

W końcowym efekcie działania funkcji fork w systemie pojawiają się dwa procesy o różnych identyfikatorach, różnych przestrzeniach adresowych, realizujące ten sam kod (obszary pamięci zawierające kod są współdzielone lub zostały skopiowane) i współdzielące szereg zasobów (pliki, połączenia sieciowe itp.). Jaki jest sens utworzenia dwóch procesów realizujących ten sam program? Co można zrobić w tej sytuacji oprócz, jak się wydaje, bezsensownego powielania operacji jednego procesu przez drugi proces?

[3]W przykładzie tym założone jest, że procesy wykonujące program są procesami jednowątkowymi, stąd w niektórych kontekstach określenia proces i wątek stosowane są zamiennie.

Odpowiedź na to pytanie daje wprowadzenie jednej z podstawowych klasyfikacji modeli przetwarzania równoległego. **Pierwszym z modeli w ramach tej klasyfikacji jest model SPMD – *Single Program Multiple Data* – jeden program wiele danych. Dwa wykonania tego samego fragmentu programu mogą się różnić, jeżeli związane są z innymi danymi wejściowymi.** Ten sam kod może posłużyć do współbieżnego wykonania przez różne wątki, a programista może tak skonstruować program, że każdy wątek będzie operował na innych danych, zrealizuje sobie wyznaczoną część zadania, a całkowity efekt działania wątków będzie stanowił rozwiązanie problemu obliczeniowego postawionego programowi.

W jaki sposób wprowadzić różne dane do dwóch procesów, powstałych w efekcie działania funkcji fork, jeśli procesy te są praktycznie identyczne? W tym celu należy wykorzystać drugi efekt działania funkcji fork. Zwraca ona wartość, która jest inna w procesie nadrzędnym, a inna w procesie potomnym. Dzięki tym różnym zwracanym wartościom, programista może zlecić zupełnie inne zadania procesowi nadrzędnemu i inne procesowi potomnemu.

W momencie wywołania funkcji fork musimy zmienić sposób patrzenia na kod źródłowy. Nie jest to już kod, który zostanie przez każdy wątek wykonany tak samo. Pojawiają się dwa różne sposoby wykonania tego kodu, i to sposoby realizowane współbieżnie! Musimy myśleć o dwóch jednocześnie pracujących wątkach, realizujących ten sam kod, choć na różnych danych. To istotna komplikacja, zmieniająca sposób myślenia o kodzie w stosunku do standardowego wykonania jednowątkowego. Dodatkowo musimy uwzględnić naturę wszystkich obliczeń wielowątkowych realizowanych przez standardowe systemy operacyjne na standardowych maszynach jedno- i wieloprocesorowych (wielordzeniowych) – każdy wątek w dowolnej chwili może zostać przerwany i tylko od systemu operacyjnego zależy kiedy zostanie wznowiony.

Oznacza to, że w trakcie wykonania wielu wątków, jeśli nie zostaną wprowadzone specjalne mechanizmy synchronizacji pracy wątków, kolejność realizacji rozkazów różnych wątków może być dowolna. Każdy wątek pracuje sekwencyjnie, kolejność realizacji rozkazów pojedynczego wątku wynika jednoznacznie z kodu źródłowego, jednakże sposób w jaki przeplatają się wykonania różnych wątków nie jest deterministycznie określony i za każdym razem może być inny. Może być tak, że najpierw jeden wątek wykona wszystkie swoje rozkazy, a dopiero potem drugi, może być, że najpierw wszystkie rozkazy wykona ten drugi, może zajść też dowolny przeplot rozkazów obu wątków. To czy mamy do czynienia z wątkami jednego czy wielu procesów nic nie zmienia w tym obrazie. Różnica pomiędzy wykonaniem na pojedynczym procesorze, a wykonaniem na maszynie wieloprocesorowej, polega na tym, że w przypadku maszyny wieloprocesorowej trzeba dopuścić dodatkowo możliwość, że dwa rozkazy wykonywane są jednocześnie. Zasada zawsze pozostaje taka sama: **analizując realizację współbieżną wielu wątków, przy braku mechanizmów synchronizacji, musimy uwzględniać sekwencyjny charakter pracy każdego wątku oraz dowolny przeplot wykonania rozkazów przez różne wątki.**

Patrząc na kod 2.1 widzimy jak model SPMD realizowany jest w praktyce. W momencie powrotu z procedury fork mamy do czynienia z dwoma procesami, jednak każdy z nich posiada inną wartość zmiennej pid, w procesie nadrzędnym jest ona równa identyfikatorowi procesu potomnego, w procesie potomnym wynosi 0. Za pomocą instrukcji warunkowej if wykonanie programu rozdziela się na dwa odrębne strumienie. **Wprowadzenie pojedynczej liczby, identyfikatora, różnego dla każdego z wątków współbieżnie realizujących ten sam kod, a następnie wykorzystanie tego identyfikatora do zróżnicowania zadania wykonywanego przez każdy wątek, jest jednym z najczęściej wykorzystywanych mechanizmów do praktycznej implementacji modelu przetwarzania SPMD.**

W kodzie 2.1 znajduje się jeszcze pewna zmienna globalna (zmienna_globalna). Warto zwrócić uwagę na fakt, że jest to zmienna globalna dla programu, tzn. dostępna z dowolnej procedury programu, jednak w momencie wykonania funkcji fork zostaje ona powielona w dwóch egzemplarzach i istnieje zupełnie niezależnie dla procesu nadrzędnego i procesu potomnego.

W następującej po wywołaniu procedury fork części programu 2.1 zilustrowano sposób w jaki wielozadaniowy system operacyjny może uruchamiać programy użytkowników. Proces potomny wywołuje procedurę systemową execl, jedną z szeregu procedur z rodziny exec, służącą do powiązania procesu z plikiem wykonywalnym. W następstwie wywołania tej procedury, proces ją realizujący zaczyna wykonywać kod z pliku wskazanego jako argument procedury. Oznacza to umieszczenie w przestrzeni adresowej procesu potomnego nowego kodu z pliku wykonywalnego oraz zmianę bloku kontrolnego procesu, tak aby odpowiadał realizacji nowo wczytanego pliku binarnego.

Pomijając szczegóły realizacji procedury execl, należy wspomnieć tylko, że proces ją wywołujący praktycznie "znika" z kodu źródłowego 2.1. Wątek wykonujący execl nie powinien wykonać żadnej innej instrukcji z kodu 2.1. Jedynym wyjątkiem jest sytuacja kiedy wykonanie procedury execl nie powiedzie się i sterowanie wróci do funkcji main. W tym momencie kod 2.1 przewiduje wyświetlenie odpowiedniego komunikatu dla użytkownika.

W praktyce funkcjonowania wielozadaniowych systemów operacyjnych, program wykonywany przez proces potomny jest najczęściej niezwiązany z programem wykonywanym przez proces nadrzędny. Jeśli jednak nowo utworzony, np. poleceniem fork, proces zaczyna wykonywać program będący inną częścią rozwiązania tego samego problemu obliczeniowego, mamy do czynienia z modelem przetwarzania równoległego MPMD – *Multiple Program Multiple Data*. **W modelu MPMD wiele procesów wykonuje wiele plików binarnych, które jednak tworzą jeden program, rozwiązujący jeden problem obliczeniowy.**

Kod 2.1 zawiera jeszcze jedną interesującą konstrukcję. Proces nadrzędny bezpośrednio po funkcji fork wywołuje funkcję wait(). W najprostszym zastosowaniu, użytym w kodzie, funkcja wait() powoduje zawieszenie pracy wątku nadrzędnego, do czasu zakończenia pracy wątku potomnego. Jest to pierwszy, który poznajemy, i jeden z najważniejszych, mechanizmów synchronizacji pracy wątków. W momencie wykonania instrukcji printf mamy pewność, że wątek, na który oczekujemy, zakończył pracę. Innymi słowy instrukcja printf zostanie wykonana po wszystkich instrukcjach realizowanych przez wątek, na który czekamy.

2.4 Wątki

Wątki najwygodniej jest scharakteryzować poprzez odniesienie do procesów. Wątek jest częścią procesu. **Najważniejszym elementem wątku jest wykonywany strumień rozkazów.** Praktycznie o wątkach sens jest mówić tylko w przypadku procesów wielowątkowych. Wtedy w ramach pojedynczego procesu istnieje wiele strumieni rozkazów, najczęściej wykonywanych współbieżnie lub, w szczególnym przypadku realizacji na maszynach wielordzeniowych (wieloprocesorowych), równoległe. Podobnie jak procesy, wątki w ostateczności[4] zarządzane są przez system operacyjny – przetwarzanie wielowątkowe jest sposobem realizacji obliczeń równoległych w ramach pojedynczego systemu operacyjnego.

Tak jak było to opisywane wcześniej, w przypadku pracy wielowątkowej będziemy zakładać, że zbiór wszystkich rozkazów wykonywanych w ramach procesu jest dzielony na podzbiory. Pojedynczy wątek wykonuje rozkazu pojedynczego podzbioru. **W ramach każdego wątku rozkazy wykonywane są sekwencyjnie. Będziemy przyjmować, że w przypadku braku synchronizacji, dopuszczalne jest dowolne uporządkowanie (przeplatanie) rozkazów wykonywanych przez różne watki.**

Jak widać każdy wątek realizuje swój ciąg, swoją sekwencję rozkazów. Jednak rozkazy te nie są zapisane w prywatnej przestrzeni adresowej wątku – kod dla wszystkich wątków pozostaje w jednym obszarze pamięci, w obszarze przydzielonym przez system operacyjny procesowi. Obszar ten jest współdzielony przez wszystkie wątki.

[4]W dalszej części krótko omówione zostaną także wątki zarządzane przez specjalne środowiska uruchomieniowe, co nie zmienia faktu, że wszystkie watki konkretnego procesu w ostateczności pracuja pod kontrolą jednego systemu operacyjnego

Pojedynczy wątek korzysta z przestrzeni adresowej procesu nie tylko dla wykonywanego przez siebie kodu, ale także dla danych na których operuje. **Przestrzeń adresowa jest cechą procesu, a nie wątku – wątek nie posiada własnej przestrzeni adresowej. Przestrzeń adresowa jest przydzielana procesowi, wszystkie wątki korzystają z przestrzeni adresowej procesu, w ramach którego funkcjonują. Fakt współdzielenia przestrzeni adresowej przez wątki ma kluczowe znaczenie dla modelu programowania i przetwarzania. Przetwarzanie wielowątkowe będzie dla nas zawsze oznaczało przetwarzanie z wykorzystaniem pamięci wspólnej.**

Nie oznacza to, że wszystkie wątki współdzielą na równych prawach cały obszar danych w przestrzeni adresowej procesu. Dane przechowywane są w komórkach pamięci. Odpowiednikiem konkretnej danej w kodzie źródłowym programu jest zmienna[5]. Współdzielenie całej przestrzeni adresowej danych przez wszystkie wątki oznaczałoby, że każdy symbol, każda zmienna o tej samej nazwie, odnosi się we wszystkich wątkach do tej samej komórki pamięci.

Praktyka programowania równoległego pokazuje, że takie założenie doprowadziłoby do znacznego skomplikowania programów wielowątkowych. Intuicyjnie jest oczywiste, że wątki w swojej pracy potrzebują wielu zmiennych tymczasowych, lokalnych. Umożliwienie dostępu do tych zmiennych innym wątkom jest niepotrzebne, a może być wręcz szkodliwe. Ponadto, trzeba uwzględnić fakt, że wątek realizuje typowe sekwencje rozkazów, a więc sekwencje zawierające wywołania procedur. **Każdy wątek działa niezależnie od innych, więc musi posiadać niezależną obsługę wywołań. Wywołanie procedur, podobnie jak funkcjonowanie zmiennych tymczasowych (automatycznych) procedur, jest wspierane przez mechanizm stosu. Prowadzi to do wniosku, że wygodnym sposobem wprowadzenia zmiennych prywatnych wątków jest tworzenie odrębnego stosu dla każdego wątku.** W wątkach, o których mówimy w tym rozdziale, wątkach zarządzanych przez systemy operacyjne (w odróżnieniu od wątków w specjalnych środowiskach programowania równoległego, w których funkcjonowanie wątków może się różnić od funkcjonowania wątków systemowych) przyjęta jest ta właśnie cecha charakterystyczna wątków: **każdy wątek posiada swój własny stos.**

Niezależnie od tego czy powiązane jest to z pojęciem stosu, czy nie, **istnienie zmiennych prywatnych wątków jest założeniem przyjętym w praktycznie wszystkich środowiskach programowania wielowątkowego. Zmienna prywatna wątku to zmienna, która jest niedostępna dla innych wątków** (przynajmniej wprost – poprzez odwołanie do jej nazwy, zależnie od środowiska programowania może być dopuszczalne odwołanie się z poziomu jednego wątku do zmiennej prywatnej innego wątku poprzez wskaźnik/referencję). Nazwa zmiennej prywatnej w jednym wątku odnosi się do innej komórki pamięci, niż ta sama nazwa zastosowana do zmiennej prywatnej w innym wątku. **Jeśli mamy do czynienia z modelem SPMD i wiele wątków wykonuje ten sam kod, to ta sama zmienna prywatna w każdym wątku odnosi się do innej komórki pamięci. Innymi słowy każda zmienna prywatna istnieje w wielu kopiach, po jednym egzemplarzu dla każdego wątku.** Inaczej jest ze zmiennymi, które nie są prywatne – będziemy je nazywać zmiennymi wspólnymi. Nazwa zmiennej wspólnej w każdym wątku odnosi się do tej samej komórki pamięci. Jednym z najważniejszych zadań programisty przy programowaniu wielowątkowym jest odpowiednie rozdzielenie zmiennych w kodzie, na zmienne prywatne i zmienne wspólne, oraz umiejętne zarządzanie zmiennymi stosownie do ich charakteru.

Dotychczasowy opis wskazuje, że wątek jest charakteryzowany przez wykonywany ciąg rozkazów oraz fakt posiadania, w ramach przestrzeni adresowej procesu, prywatnego obszaru pamięci. W przypadku wątków systemowych prywatnym obszarem pamięci jest prywatny stos. Tak więc **zmiennymi prywatnymi wątków systemowych są zmienne lokalne i argumenty wykonywanych przez wątek procedur.**

Z punktu widzenia systemu operacyjnego część zasobów przydzielonych procesowi pozostaje

[5] W prezentowanych tu ogólnych analizach pomijać będziemy fakt, że zmienne różnych typów zajmują obszary pamięci różnej wielkości. Dla uproszczenia opisu przyjmiemy konwencję zakładającą, że w jednej komórce pamięci przechowywana jest jedna zmienna.

wspólna dla wszystkich jego wątków. Pozostałe zasoby są traktowane jako prywatne dla każdego wątku i dla każdego wątku tworzone są osobne kopie odpowiednich struktur danych. Szczegóły zależą od konkretnego systemu operacyjnego. **Z faktu niezależnego wykonywania odrębnych sekwencji rozkazów wynika, że odrębna dla każdego wątku musi pozostać zawartość rejestrów procesora. Natomiast praktycznie zawsze wspólne dla wszystkich wątków pozostają otwarte pliki i połączenia sieciowe.** Wynika z tego, że całość struktur danych związanych z pojedynczym wątkiem jest mniejsza niż całość struktur danych związanych z pojedynczym procesem. Stąd też wątki bywają czasem nazywane lekkimi procesami (ang. *lightweight processes*)[6].

Zarządzając wykonaniem procesów i wątków, system operacyjny (lub specjalne środowisko uruchomieniowe, jak np. maszyna wirtualna) często dokonuje wywłaszczenia – przerwania działania wątku/procesu, w celu przydzielenia czasu procesora innemu procesowi/wątkowi. Po to, aby można było po pewnym czasie powrócić do realizacji przerwanego procesu/wątku pewne informacje dotyczące procesu/wątku (np. zawartość rejestrów) muszą zostać zapisane w odpowiednich strukturach danych (np. bloku kontrolnym). Dane te są kopiowane z powrotem w momencie powrotu do wykonania procesu/wątku. **Całość operacji związanych z wywłaszczeniem jednego procesu/wątku i z przydzieleniem procesora innemu procesowi/wątkowi, nazywana jest przełączaniem kontekstu.** Dzięki temu, że pojedynczy wątek związany jest z mniej rozbudowanymi strukturami danych, **przełączanie kontekstu pomiędzy wątkami odbywa się zazwyczaj szybciej niż pomiędzy procesami**.

Wątki – użytkownika, jądra, lekkie i zielone. Wątki będące głównym obiektem naszych zainteresowań to wątki tworzone w ramach systemów operacyjnych, które są przez systemy operacyjne zarządzane. Takie wątki nazywane są wątkami jądra (ang. *kernel threads*) lub wątkami w przestrzeni jądra (ang. *kernel space threads*). **W kontekście obliczeń równoległych istotne jest, że to system operacyjny decyduje o przydzieleniu procesora (rdzenia) pojedynczemu wątkowi jądra. Stąd wynika, że system operacyjny może w dowolnym momencie wywłaszczyć wątek, ale też że wątek może zostać przeznaczony do wykonania na dowolnym procesorze/rdzeniu dostępnym systemowi operacyjnemu.** Na dzień dzisiejszy wątki jądra są najlepszym sposobem na uzyskiwanie wydajnych programów wielowątkowych.

Wątki jądra nie są jedyną możliwością. W przypadku korzystania z pewnych środowisk uruchomieniowych (jak na przykład Java Runtime Environment) lub odpowiednich bibliotek wątków, można tworzyć tzw. wątki użytkownika (ang. *user threads*) zwane także wątkami w przestrzeni użytkownika (ang. *user space threads*). Dla systemu operacyjnego wątki takie są niewidoczne, widoczny jest tylko proces, w ramach którego zostały utworzone (np. proces maszyny wirtualnej). Zarządzaniem wątkami zajmuje się środowisko uruchomieniowe, ono decyduje m.in. o wywłaszczaniu wątków i przełączaniu kontekstu. Niekiedy rozważa się mechanizmy zarządzania wątkami, w których to same wątki decydują o przerwaniu pracy (np. poprzez wywołanie komendy sleep) i oddaniu procesora innym wątkom. Wątki takie bywają nazywane włóknami (ang. *fibers*).

Nazwy wątków – użytkownika i jądra, związane są z techniczną realizacją przełączania kontekstu między wątkami. Przełączenie kontekstu między wątkami jądra wymaga przejścia w tryb pracy jądra systemu operacyjnego, w którym realizowany jest algorytm przydziału procesora. W przypadku wątków użytkownika wszystko odbywa się w trybie pracy użytkownika.

Wątki w przestrzeni użytkownika mogą mieć zalety w stosunku do wątków jądra. Przełączanie kontekstu może być szybsze, korzystanie z wspólnych zasobów wielu wątków może być sprawniejsze. Mają też istotna wadę – zazwyczaj nie pozwalają na pracę równoległą, tylko współbieżną w przeplocie. Praca równoległa pojawia się wtedy, kiedy środowisko uruchomieniowe

[6] Pojęcie lekkiego procesu nie jest jednoznaczne i może mieć różne znaczenie w różnych systemach operacyjnych i środowiskach wykonania równoległego.

dysponuje kilkoma wątkami jądra, którym przydziela zarządzane przez siebie wątki użytkownika. Jednak w takim przypadku często pojawiają się kłopoty z uzyskiwaniem wysokiej wydajności, ze względu na konflikty dwóch mechanizmów zarządzania przydziałem procesorów/rdzeni wątkom – mechanizmu związanego z systemem operacyjnym (wątki jądra) i mechanizmu związanego ze środowiskiem uruchomieniowym (powiązanie wątków jądra z wątkami użytkownika).

Kiedy korzystamy z rozbudowanego środowiska uruchomieniowego, może zdarzyć się, że środowisko podejmuje decyzje o wyborze mechanizmu przełączania kontekstu. Mechanizm ten może być złożony, obejmując pewne działania w trybie pracy użytkownika oraz działania w trybie pracy jądra. Różne środowiska mogą stosować różne szczegółowe rozwiązania. Warto je znać jeśli dąży się do uzyskania maksymalnej wydajności przetwarzania wielowątkowego.

Konsekwencją rozmaitości implementacji działania wątków jest też rozmaitość nazw jakimi się posługuje na określanie różnych typów wątków. Poza wątkami jądra i użytkownika, wyróżnia się także lekkie procesy i wątki zielone (tak często nazywa się wątki zarządzane przez maszyny wirtualne). Z punktu widzenia użytkownika programów wielowątkowych, ważne jest w zasadzie tylko to, czy konkretny stosowany mechanizm pozwala na równoległa pracę wątków oraz jakie są parametry wydajnościowe tego mechanizmu. Programista może być zainteresowany także tym czy sposób zarządzania wątkami narzuca jakieś ograniczenia związane z synchronizacja pracy wątków (w przypadku wątków, które same decydują o swoim wywłaszczeniu, mogą pojawić się zakleszczenia w sytuacjach, w których zarządzanie przez system operacyjny nie prowadzi do wstrzymania pracy programu, patrz np. [2] str. 160).

2.5 Tworzenie wątków i zarządzanie wykonaniem wielowątkowym – biblioteka Pthreads

W punkcie 2.3 pokazane zostało jak proces może utworzyć nowy proces. W podobny sposób przebiega tworzenie nowego wątku. Tym razem pokażemy sytuację, w której jeden wątek procesu tworzy nowy wątek tego samego procesu. Zamiast narzędzi konkretnego systemu operacyjnego, użyjemy środowiska programowania wielowątkowego Pthreads. Tym określeniem będziemy nazywali wszelkie środowiska zgodne ze standardem zarządzania wątkami zawartym w specyfikacji POSIX.

POSIX jest zbiorem standardów opracowanym w celu ujednolicenia interfejsu programistycznego rozmaitych wersji systemu operacyjnego Unix. Standard dotyczący wątków, *POSIX.1c, Threads extensions (IEEE Std 1003.1c-1995)*, jest jednym z najpopularniejszych spośród dokumentów POSIX. Implementuje go wiele systemów operacyjnych, także tych nie w pełni zgodnych z całością specyfikacji POSIX, a także wiele środowisk programowania, np. będących nakładkami na systemy operacyjne (jak środowisko Cygwin dla systemu Microsoft Windows). Opiekę nad specyfikacją POSIX sprawuje obecnie organizacja ”The Open Group” i standardy są w całości dostępne w sieci WWW (wcześniej specyfikacja POSIX była firmowana przez stowarzyszenie IEEE, jest ona także standardem ISO).

Sposobem prezentacji interfejsu programowania Pthreads, podobnie jak innych opisywanych później środowisk programowania równoległego, nie będzie wyliczenie kolejnych typów zmiennych i procedur składających się na środowisko. Omawiane w skrypcie środowiska są najczęściej środowiskami otwartymi, zgodnymi z powszechnie dostępnymi standardami i osoby poszukujące specyfikacji poszczególnych funkcji i typów zmiennych mogą je łatwo znaleźć w Internecie lub korzystając z systemów

pomocy w ramach systemów operacyjnych (np. stron podręcznika w systemach Unix dostępnych za pomocą polecenia man). Zgodnie z ideą skryptu przedstawionych zostanie kilka przykładów, mających zilustrować najważniejsze cechy środowiska i najważniejsze sposoby radzenia sobie z problemami programowania równoległego. W opisach towarzyszących przykładom znajdą się też dodatkowe informacje na temat środowiska i innych sposobów rozwiązywania rozpatrywanych problemów.

2.5.1 Tworzenie i uruchamianie wątków

Kod 2.2 przedstawia, w jaki sposób można utworzyć program wielowątkowy oraz jak zrealizować model SPMD w środowisku Pthreads. Podobnie jak w przykładowym kodzie 2.1 program napisany jest w języku C. Chcąc korzystać z procedur zdefiniowanych w standardzie należy do pliku źródłowego dołączyć plik nagłówkowy pthread.h. Przy kompilacji należy wskazać lokalną ścieżkę do pliku nagłówkowego (opcja -I) i do biblioteki (opcja -L) oraz podać nazwę biblioteki (najczęściej -lpthread). Przykładowa komenda kompilacji w systemie Linux może wyglądać następująco:

```
$ gcc -I/usr/include -L/usr/lib64 kod_zrodlowy.c -lpthread
```

Opcje -I i -L mogą zostać pominięte przy korzystaniu ze standardowych lokalizacji plików. W wielu przypadkach opcja kompilacji -pthread powoduje automatyczne wyszukanie pliku nagłówkowego i biblioteki. Także przy korzystaniu ze zintegrowanych środowisk programowania powyższe operacje mogą zostać zrealizowane automatycznie.

W standardowy dla języka C sposób, wykonanie skompilowanego kodu 2.2 związane jest z utworzeniem nowego procesu, którego jedyny w tym momencie wątek rozpoczyna realizację procedury main. Wątek ten nazywany będzie wątkiem głównym. Wątek główny w kodzie 2.2 wywołuje procedurę tworzenia wątków pthread_create. Jej sygnatura jest nastepująca:

```
int pthread_create(pthread_t *thread_id, const pthread_attr_t *attr, void *(*start_routine)
(void *), void *arg);
```

Wywołanie procedury pthread_create powoduje utworzenie nowego wątku w procesie[7]. Procedurę może wywołać dowolny wątek, niekoniecznie wątek główny. Pierwszym argumentem pthread_create jest zwracany przez procedurę identyfikator nowo tworzonego wątku (w zadeklarowanej wcześniej zmiennej typu pthread_t). Drugim argumentem procedury jest wskaźnik do struktury zawierającej atrybuty wątku. W przypadku przesłania wartości NULL tworzone są watki o własnościach domyślnych.

Twórcy standardu Pthreads zdecydowali, że utworzenie nowego wątku jest zawsze związane z jednoczesnym uruchomieniem wątku. Jak było to już opisywane, rozpoczęcie pracy wątku związane jest z rozpoczęciem wykonania wybranej funkcji kodu. Wskaźnik do tej funkcji umieszczany jest jako trzeci argument procedury pthread_create. W programach źródłowych języka C, wskaźnik do funkcji jest tożsamy z nazwą funkcji. Czwartym argumentem jest wskaźnik (jeden!) do argumentów procedury zlecanej do wykonania wątkowi.

Procedura zwraca kod sukcesu lub błędu. W programie 2.2, podobnie jak w innych programach w skrypcie, kody błędów nie są szczegółowo sprawdzane (patrz uwaga o fragmentach kodu zamieszczanych w skrypcie na stronie 1). Pomyślne zakończenie procedury pthread_create oznacza, że utworzony

[7]Implementacje procedury pthread_create w systemie Linux korzystają najczęściej z funkcji systemowej clone o podobnej składni.

został nowy wątek, stworzony jego prywatny stos oraz zlecone zostało rozpoczęcie funkcji wskazanej jako trzeci argument procedury pthread_create.

Kod 2.2: Prosty program, w którym tworzone są dwa nowe wątki procesu i przekazywane im argumenty

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

void * zadanie_watku (void * arg_wsk);

int zmienna_wspolna=0;

main(){
  int wyn_1, wyn_2;
  int arg_1=1, arg_2=2;
  pthread_t tid_1, tid_2;

  wyn_1 = pthread_create(&tid_1, NULL, zadanie_watku, &arg_1);
  wyn_2 = pthread_create(&tid_2, NULL, zadanie_watku, &arg_2);
  wyn_1 = pthread_join( tid_1, NULL);
  wyn_2 = pthread_join( tid_2, NULL);

  printf("Wątek główny: zmienna wspólna = %d\n", zmienna_wspolna);
}

void * zadanie_watku (void * arg_wsk){
  int moj_arg;
  moj_arg = *( (int *) arg_wsk );
  zmienna_wspolna++;
  printf("Watek otrzymal argument %d. Wartosc zmiennej wspolnej wynosi %d\n",
          moj_arg, zmienna_wspolna);
  return(NULL);
}
```

Program 2.2 jest niezwykle prosty. Wątek główny tworzy dwa nowe wątki, zleca do wykonania funkcję zadanie_watku i przekazuje każdemu z nich odpowiedni argument. Każdy z wątków wykonuje procedurę zadanie_watku, która polega na wypisaniu na ekranie wartości otrzymanego argumentu. Kilka elementów programu warte jest zwrócenia uwagi.

Procedura zlecana do wykonania wątkom ma ściśle określoną sygnaturę. Posiada jeden argument, którym jest wskaźnik do pewnej komórki pamięci o nieokreślonym typie (symbolicznie oznaczanym jako wskaźnik do typu void). W programie 2.2 wątek główny przekazuje jako argumenty wskaźniki do swoich zmiennych lokalnych typu int. Formalnie powinno się dokonać ich rzutowania na typ void*, co spowodowałoby jednak komplikacje kodu. Rzutowanie w tę stronę nie jest konieczne i kompilator jest w stanie jednoznacznie odczytać instrukcję kodu.

Inaczej jest w przypadku przejmowania argumentu. Typ w sygnaturze funkcji void* jest bardziej ogólny niż int* i kompilator może domagać się jednoznacznego wskazania dopuszczalności rzutowania z typu void* na typ int*. W kodzie 2.2 zmiana typu argumentu połączona jest ze skopiowaniem wartości wskazywanej przez argument wsk do zmiennej lokalnej procedury zadanie_watku, a więc do zmiennej prywatnej wątku. Tym sposobem wewnątrz procedury zadanie_watku każdy wątek operuje na swojej

prywatnej zmiennej moj_arg (umieszczonej na stosie danego wątku), która jest niezależna od zmiennych arg_1 i arg_2 umieszczonych na stosie wątku głównego.

W kodzie 2.2 pokazane jest także funkcjonowanie zmiennych wspólnych w kodzie. Każda zmienna globalna w programie umieszczana jest w obszarze statycznym pamięci, a w procesach wielowątkowych staje się zmienną wspólną wszystkich wątków. Utworzenie nowych wątków nie wpływa na jej funkcjonowanie, przez cały czas programu istnieje w jednej kopii. Zapis dokonany na tej zmiennej przez jeden wątek staje się widoczny dla wszystkich wątków. Widać to w programie 2.2, gdzie zmiany zmiennej wspólnej dokonywane są przez wątki potomne, a jej wartość odczytywana przez wszystkie wątki. Widać jak wygodnym mechanizmem komunikacji pomiędzy wątkami jest posiadanie zmiennych wspólnych.

Podobnie jak w kodzie tworzenia procesów 2.1, także i w kodzie tworzenia wątków 2.2 zastosowano prosty mechanizm synchronizacji. Wątek główny wywołuje procedurę pthread_join, która powoduje wstrzymanie jego pracy do momentu kiedy wątek, którego identyfikator umieszczony jest jako pierwszy argument pthread_join nie zakończy swojej pracy. W programie 2.2 zastosowana synchronizacja gwarantuje, że wątek główny nie zakończy swojej pracy przed zakończeniem pracy obu wątków potomnych.

Procedura pthread_join posiada jeszcze drugi argument. Jest nim wskaźnik do wskaźnika zwracanego ewentualnie przez funkcje wykonywaną przez wątki (argument instrukcji return lub, w tym kontekście, równoważnego mu wywołania pthread_exit). Wątek, zgodnie ze swoją sygnaturą, musi kończąc wykonanie zleconej funkcji zwracać wskaźnik do zmiennej (dokładniej wskaźnik typu void*). Nie może to być adres zmiennej lokalnej, gdyż ta znika wraz z zakończeniem pracy wątku, w procedurze musi być dostępna zmienna będąca wskaźnikiem. Ze względu na złożoność tego sposobu przekazywania informacji z wątków (formalnie ponownie wymagającego konwersji typu argumentu – i w procedurze wątku, i procedurze main) jest on rzadziej stosowany. Zazwyczaj prościej jest przekazać informacje z wątku za pomocą zmiany wartości dowolnej wybranej zmiennej wspólnej. Dlatego jako drugi argument pthread_join (oraz argument zwracany przez watki w instrukcji return) będziemy zazwyczaj używać NULL (wyjątkiem jest przykładowy kod 2.4, gdzie wątki potomne przekazują specjalny sygnał do wątku głównego).

2.5.2 Przekazywanie argumentów do wątków potomnych

Pewnym utrudnieniem stosowania wątków Pthreads jest fakt, że do procedury wykonywanej przez uruchamiane wątki można przekazać tylko jeden argument. Jednak typ tego argumentu jest dobrany w taki sposób, żeby można było za jego pomocą przekazać wskaźnik do zmiennej dowolnego typu, a więc także wskaźnik do dowolnej zdefiniowanej przez użytkownika struktury, która z kolei może w sobie zawierać dowolną liczbę zmiennych różnych typów.

Przykład takiego przekazania szeregu argumentów zawarty jest w kodzie 2.3. W programie tym, w wątku głównym definiowana i następnie inicjowana jest pewna struktura. Przy tworzeniu nowego wątku wskaźnik do tej struktury jest przekazywany jako argument dla funkcji zlecanej do wykonania wątkowi.

Wykonanie funkcji, jak zwykle, zaczyna się od rzutowania argumentu na odpowiedni typ. Wewnątrz procedury przekazane argumenty wykorzystywane są na dwa alternatywne sposoby. W pierwszym, funkcja dokonuje tylko rzutowania wskaźnika i następnie korzysta z tego wskaźnika, co oznacza że poprzez wskaźnik uzyskuje dostęp do zmiennych, które znajdują sie poza jej stosem, a więc są dostępne także innym wątkom.

W drugim przypadku procedura dokonuje kopiowania wartości na swój stos. W tym celu deklaruje zmienna lokalną o typie odpowiadającym przesyłanemu wskaźnikowi i następnie przypisuje odpowiednie wartości zmiennym tworzącym strukturę. W tym momencie wszystkie przekazane w strukturze argumenty stają się zmiennymi prywatnymi pracującego wątku.

Kod 2.3: Program, w którym do tworzonego nowego wątku przekazywany jest zbiór argumentów

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

struct struktura_t { int a; double b; char c; };

void * zadanie_watku (void * arg_wsk);

main(){
  pthread_t tid;
  struct struktura_t struktura_main = { 1, 3.14, 'c' };

  pthread_create(&tid, NULL, zadanie_watku, &struktura_main);
  pthread_join( tid, NULL);
}

void * zadanie_watku (void * arg_wsk){

  struct struktura_t *wskaznik_do_struktury_main = arg_wsk;

  struct struktura_t struktura_lokalna;
  struktura_lokalna = *( (struct struktura_t *) arg_wsk );

  printf("Dostep do wartosci z procedury main: a = %d, b = %lf, c = %c\n",
       wskaznik_do_struktury_main->a, wskaznik_do_struktury_main->b,
       wskaznik_do_struktury_main->c);

  printf("Dostep do skopiowanych lokalnych wartosci: a = %d, b = %lf, c = %c\n",
       struktura_lokalna.a, struktura_lokalna.b, struktura_lokalna.c);

  return(NULL);
}
```

2.5.3 Czas życia wątków*

Ostatnim przykładem ilustrującym zarządzanie wątkami Pthreads jest kod 2.4, który pokazuje w jaki sposób może odbywać się kończenie pracy wątku.

Najczęstszym sposobem organizacji pracy wątków jest ten pokazany w przykładzie 2.2: wątek główny tworzy wątek poboczny, ewentualnie wykonuje jakieś własne zadanie, a następnie czeka na zakończenie pracy wątku pobocznego. W takiej sytuacji **zakończenie pracy wątku następuje w momencie kiedy wątek wywołuje w dowolnej procedurze funkcję pthread_exit lub w momencie powrotu z funkcji, której wykonywanie rozpoczął w chwili tworzenia**. W kodzie 2.2 koniec życia wątku następuje wraz z wykonaniem instrukcji return(NULL).

Wątek może zostać także anulowany przez inne wątki[8]. W programie może to zostać zrealizowane poprzez jawne przesłanie wątkowi odpowiedniego sygnału (procedura pthread_kill – odpowiednik

[8]Potocznie mówi się często o zabijaniu wątków, co ma swoje uzasadnienie w fakcie, że w ramach systemów Unixowych proces anulowania wątku (czy procesu) realizowany jest poprzez przesłanie wątkowi (procesowi) sygnału KILL (SIGKILL).

Unixowej funkcji kill) lub przez wywołanie funkcji pthread_cancel. To drugie rozwiązanie jest prostsze i na nim skupimy uwagę.

Wywołanie pthread_cancel przesyła do wątku sygnał anulowania. Watek zakończy swoja pracę po odebraniu sygnału. Powstaje pytanie kiedy wątek odbiera sygnały? Rozwiązanie, w którym wątek odbierałby sygnały w sposób ciągły, np. po każdym rozkazie, prowadziłoby do znacznego obciążenia procesora, redukcji efektywnego czasu pracy wątku i obniżenia wydajności. Dlatego wątek odbiera sygnały tylko w tzw. punktach anulowania (*cancellation points*). Punktami anulowania są m.in. wywołania szeregu funkcji systemowych (np. sleep). Istnieje też specjalna funkcja do testowania czy do wątku nie został wysłany sygnał anulowania – pthread_testcancel.

Programista piszący kod wątku może uzyskać dodatkową kontrolę nad anulowaniem wątku. W standardzie Pthreads istnieje funkcja, pthread_setcancelstate, ustalająca czy wątek będzie przyjmował sygnał anulowania czy nie. Wywołanie funkcji pthread_setcancelstate z argumentem PTHREAD_CANCEL_DISABLE powoduje, że wątek nie będzie przyjmował sygnału anulowania, nawet w punktach anulowania. Natomiast wywołanie pthread_setcancelstate z argumentem PTHREAD_CANCEL_ENABLE spowoduje, że wątek w najbliższym punkcie anulowania przyjmie przesłany mu sygnał i zakończy pracę.

Zakończenie pracy wątku oznacza zwolnienie zasobów przydzielonych wątkowi. Jednak nie wszystkich – część zasobów jest utrzymywana do momentu kiedy wątek główny wywołuje funkcję pthread_join dla wątku, który zakończył pracę. Dzieje się tak dla każdego wątku, niezależnie czy zakończył prace sam czy został anulowany, ale tylko w przypadku jeśli watek jest tzw. wątkiem przyłączalnym (*joinable*). Wątek przyłączalny to wątek, na zakończenie pracy którego można oczekiwać wywołując procedurę synchronizacji pthread_join. Jest to wygodne w sytuacji, kiedy watek główny w swoim działaniu jest skoordynowany z działaniem wątku potomnego i np. oczekuje na wynik jego działania, aby móc kontynuować swoja pracę.

Kiedy wątek główny nie jest związany w swojej pracy z działaniem wątku pobocznego, wywoływanie pthread_join i utrzymywanie zasobów wątku pobocznego może stać się niepotrzebnym obciążeniem i komplikacją dla programu. Dzieje się tak np. wtedy kiedy zadaniem wątku głównego jest tylko przyjmowanie żądań przychodzących przez połączenia sieciowe, a wątki potomne w sposób całkowicie niezależny od wątku głównego obsługują żądania. W takim przypadku wątki potomne mogą funkcjonować jako wątki odłączone (*detached*).

Wątek może znaleźć się w stanie odłączonym na dwa sposoby. Może zostać od razu utworzony w takim stanie, ewentualnie może zostać odłączony przez wątek główny. Drugie rozwiązanie jest prostsze, gdyż pozwala na stworzenie wątku w stanie domyślnym (stan bycia przyłączalnym jest zgodnie ze specyfikacją POSIX stanem domyślnym) i następnie wywołanie pojedynczej funkcji pthread_detach.

Utworzenie wątku w stanie odłączonym jest bardziej skomplikowane. Omawiamy ten mechanizm, dlatego że w podobny sposób ustawia się inne atrybuty wątku oraz atrybuty innych tworzonych w ramach specyfikacji Pthreads obiektów, takich jak mutex'y i zmienne warunku, które zostaną omówione w kolejnych rozdziałach.

Zgodnie ze specyfikacją POSIX wymienione powyżej obiekty są tworzone lub inicjowane za pomocą wywołania odpowiednich procedur. Jednym z argumentów takich procedur jest obiekt zawierający informację dotyczącą atrybutów tworzonych elementów. **Obiekt ten jest tzw. obiektem nieprzenikalnym (nieprzeźroczystym, *opaque*), obiektem którego struktura pozostaje niejawna i którym można tylko manipulować poprzez wywołania odpowiednich funkcji.** W przypadku wątków procedurą tworzącą jest pthread_create, a obiekt przekazujący atrybuty ma typ pthread_attr_t. Obiekt ten musi zostać zainicjowany za pomocą funkcji pthread_attr_init. Następnie szereg rozmaitych funkcji pozwala na ustalenie atrybutów nowo tworzonego wątku.

Jakie atrybuty, jakie własności wątku zostają poddane kontroli programisty? Jedną z grup atrybutów są parametry określające stos wątku – obszar jego prywatnych zmiennych. Programista może ustalić

położenie (adres początkowy) stosu, a także jego rozmiar. Druga z grup atrybutów określa sposób szeregowania (*scheduling*) wątków. Zazwyczaj domyślne mechanizmy systemowe są wystarczające, jednak np. w przypadku systemów czasu rzeczywistego programista może chcieć posiadać większą kontrolę nad szeregowaniem. Najczęściej stosowanym w implementacjach rozwiązaniem jest przyjecie, że każdy wątek tworzony przez pthread_create jest odrębnym wątkiem systemowym, w standardowy sposób, na równi z innymi, konkurującym o zasoby komputera.

Poza kilkoma jeszcze mniej znaczącymi parametrami jest wreszcie parametr określający czy wątek ma być przyłączalny czy odłączony. Jego wartość ustala się wywołując, z odpowiednimi argumentami, procedurę pthread_attr_setdetachstate.

Kod 2.4 jest bardziej od poprzednich rozbudowanym przykładem, w którym wątek główny tworzy trzy wątki pochodne, a następnie testowany jest ich sposób funkcjonowania, inny dla każdego wątku. Różnice wynikają z różnych atrybutów wątków, kod realizowany przez każdy z nich jest taki sam.

Kod 2.4: Program, w którym wątki potomne maja różne atrybuty i są anulowane przez wątek główny

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

void * zadanie_watku (void * arg_wsk);

int zmienna_wspolna=0;

main(){
  pthread_t tid;
  pthread_attr_t attr;
  void *wynik;
  int i;

  printf("watek glowny: tworzenie watku potomnego nr 1\n");
  pthread_create(&tid, NULL, zadanie_watku, NULL);
  sleep(2); // czas na uruchomienie watku
  printf("\twatek glowny: wyslanie sygnalu zabicia watku\n");
  pthread_cancel(tid);

  pthread_join(tid, &wynik);
  if (wynik == PTHREAD_CANCELED)
    printf("\twatek glowny: watek potomny zostal zabity\n");
  else
    printf("\twatek glowny: watek potomny NIE zostal zabity - blad\n");


  zmienna_wspolna = 0;

  printf("watek glowny: tworzenie watku potomnego nr 2\n");
  pthread_create(&tid, NULL, zadanie_watku, NULL);
  sleep(2); // czas na uruchomienie watku
  printf("\twatek glowny: odlaczenie watku potomnego\n");
  pthread_detach(tid);

  printf("\twatek glowny: wyslanie sygnalu zabicia watku odlaczonego\n");
  pthread_cancel(tid);
```

```
  printf("\twatek glowny: czy watek potomny zostal zabity \n");
  printf("\twatek glowny: sprawdzanie wartosci zmiennej wspolnej\n");
  for(i=0;i<10;i++){
    sleep(1);
    if(zmienna_wspolna!=0) break;
  }

  if (zmienna_wspolna==0)
    printf("\twatek glowny: odlaczony watek potomny
          PRAWDOPODOBNIE zostal zabity\n");
  else
    printf("\twatek glowny: odlaczony watek potomny
          PRAWDOPODOBNIE NIE zostal zabity\n");


  pthread_attr_init(&attr);
  pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

  printf("watek glowny: tworzenie odlaczonego watku potomnego nr 3\n");
  pthread_create(&tid, &attr, zadanie_watku, NULL);
  pthread_attr_destroy(&attr);

  printf("\twatek glowny: koniec pracy, watek odlaczony pracuje dalej\n");
  pthread_exit(NULL); // co stanie sie gdy uzyjemy exit(0)
}

void * zadanie_watku (void * arg_wsk){

  pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, NULL);
  printf("\twatek potomny: uniemozliwione zabicie\n");
  sleep(5);

  printf("\twatek potomny: umozliwienie zabicia\n");
  pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, NULL);

  pthread_testcancel();

  zmienna_wspolna++;
  printf("\twatek potomny: zmiana wartosci zmiennej wspolnej\n");

  return(NULL);
}
```

Procedura wykonywana przez wątki zaczyna się od wyłączenia przyjmowania sygnału anulowania, dzięki czemu np. pewne działania rozpoczęte przez wątek na pewno zostaną zakończone. W kodzie 2.4 tym ”działaniem” wątku jest przebywanie w stanie uśpienia, którego nie można przerwać poleceniem pthread_cancel. Po drzemce wątek postanawia przyjmować sygnały anulowania i od razu sprawdza czy jakieś nie pojawiły się, wywołując pthread_testcancel. Jeśli inny wątek uprzednio wysłał sygnał anulowania, wątek realizujący funkcję zadanie_watku w tym momencie zakończy pracę.

Pierwszy wątek potomny jest najbardziej standardowy. Wątek główny tworzy go ze wszystkimi parametrami posiadającymi wartości domyślne i taki pozostaje do końca swojej pracy. Po wydaniu

polecenia utworzenia pierwszego wątku (powtarza się to także dla kolejnych wątków potomnych) wątek główny odczekuje dwie sekundy, dając systemowi czas na uruchomienie nowego wątku, po czym wysyła sygnał anulowania wątku. Sygnał najprawdopodobniej dociera do wątku potomnego, gdy ten przebywa w stanie uśpienia, z wyłączonym przyjmowaniem sygnałów. Dopiero włączenie przyjmowania sygnałów i wywołanie pthread_testcancel powoduje, że watek kończy pracę. W momencie kończenia pracy na skutek anulowania, proces potomny przesyła odpowiednią wartość zwrotną, która może zostać przejęta przez wątek główny wywołujący funkcję pthread_join. W kodzie 2.4 wątek główny wywołuje pthread_join z odpowiednim argumentem (będącym adresem wskaźnika) i następnie sprawdza wartość wskaźnika. Jeśli jest ona równa PTHREAD_CANCELED oznacza to, że wątek potomny, na którego zakończenie oczekiwał wątek główny został anulowany. W kodzie 2.4, ze względu na przewidywany przebieg zdarzeń w czasie, założone jest, że zwrócenie innej wartości oznacza błąd w działaniu systemu zarządzania wątkami.

Kod 2.4 zawiera szereg wydruków w trakcie działania wątku głównego i wątków potomnych. Przewidywany wydruk na ekranie w trakcie manipulowania pierwszym wątkiem potomnym wygląda następująco:

```
$ a.out
watek glowny: tworzenie watku potomnego nr 1
    watek potomny: uniemozliwione zabicie
    watek glowny: wyslanie sygnalu zabicia watku
    watek potomny: umozliwienie zabicia
    watek glowny: watek potomny zostal zabity
```

Drugi wątek potomny, realizujący ten sam kod co pierwszy, jest także tworzony z domyślnymi atrybutami, jednak w chwilę po utworzeniu jest odłączany poprzez wywołanie przez wątek główny procedury pthread_detach. W tym momencie wątek główny traci możliwość prostej synchronizacji z wątkiem potomnym, za pomocą wywołania pthread_join. W kodzie założono, że wątek główny chciałby mimo to zsynchronizować swoją pracę z wątkiem potomnym (stoi to w pewnej sprzeczności z przeprowadzonym przed chwilą odłączeniem wątku potomnego, niemniej ilustruje sytuację, która może się przytrafić w innych kontekstach – chcemy zsynchronizować dwa wątki nie posiadając narzędzia typu pthread_join). W tym celu definiowana jest zmienna wspólna dla obu wątków, której wartość jest zmieniana przez wątek potomny bezpośrednio przed zakończeniem pracy. Wątek główny sprawdza, w pętli z poleceniem sleep, wartość zmiennej i wyciąga odpowiednie wnioski. W momencie kiedy stwierdza, że wartość została zmieniona uznaje, że wątek potomny normalnie zakończył pracę.

W kodzie założone jest, że system jest synchroniczny, tzn. że istnieją limity czasowe na wykonanie konkretnych operacji. Jeśli limit zostaje przekroczony, uprawnione jest wnioskowanie, że nastąpiła awaria. W przypadku ilustrowanym w programie 2.4, wątek główny po odczekaniu ok. 10 sekund (10 pętli z uśpieniem w każdej iteracji na 1 sekundę) zakłada, że wątek potomny uległ awarii – w tym przypadku, że został skutecznie anulowany.

Sekwencja komunikatów wyświetlanych na ekranie w przypadku wątku potomnego 2 powinna wyglądać następująco:

```
watek glowny: tworzenie watku potomnego nr 2
    watek potomny: uniemozliwione zabicie
    watek glowny: odlaczenie watku potomnego
```

```
watek glowny: wyslanie sygnalu zabicia watku odlaczonego
watek glowny: czy watek potomny zostal zabity?
watek glowny: sprawdzanie wartosci zmiennej wspolnej
watek potomny: umozliwienie zabicia
watek glowny: odlaczony watek potomny PRAWDOPODOBNIE zostal zabity
```

Ostatni z tworzonych przez wątek główny wątków potomnych w kodzie 2.4 odchodzi od przyjętej dotąd praktyki tworzenia wątków z atrybutami domyślnymi. Przy jego tworzeniu wątek główny stosuje zmienną typu pthread_attr_t, jako drugi argument funkcji pthread_create. Zmienna ta jest uprzednio inicjowana w procedurze pthread_attr_init, a następnie jedna z procedur z rodziny pthread_attr_set... zostaje użyta do ustawienia odpowiedniego atrybutu tworzonego wątku potomnego. W przypadku kodu 2.4 użytą procedurą jest pthread_attr_setdetachstate, która dzięki przekazanemu parametrowi PTHREAD_CREATE_DETACHED powoduje utworzenie wątku w stanie odłączonym. Bezpośrednio po utworzeniu wątku zmienna typu pthread_attr_t jest niszczona w celu odzyskania zasobów (jej wartość po utworzeniu wątku nie ma już żadnego znaczenia).

W dalszym ciągu pracy programu 2.4 zakłada się, że tym razem wątek główny nie chce w żaden sposób synchronizować swojej pracy z wątkiem potomnym i nie realizuje żadnej formy oczekiwania na trzeci wątek potomny. Naturalnym jest, że w takiej sytuacji pojawia się możliwość, że wątek główny kończy pracę, a wątek potomny dalej ją kontynuuje, aż do wykonania całości zadania. Standardowym sposobem kończenia pracy przez wątek główny jest powrót z procedury main – realizowany przez jawne wywołanie poleceń return lub exit, ewentualnie niejawnie przez dotarcie do końca bloku kodu procedury. Jednakże w każdym z tych przypadków zakończenie pracy wątku głównego oznacza zakończenie pracy całego procesu, a więc zakończenie pracy wszystkich jego wątków, niezależnie od stanu w jakim się znajdują (podobnie działa wywołanie funkcji exit w dowolnym wątku potomnym). Jedynym sposobem zakończenia pracy wątku głównego z pozostawieniem pracujących wątków potomnych jest wywołanie przez wątek główny procedury pthread_exit. Tak też dzieje się w kodzie 2.4. Po wywołaniu przez wątek główny funkcji pthread_exit, trzeci wątek potomny pracuje dalej – po raz pierwszy wątek w kodzie 2.4 dokonuje zmiany wartości zmiennej wspólnej i dociera do końca procedury zadanie_watku.

Zakończenie pracy procesu następuje w momencie kiedy wszystkie wątki kończą swoją pracę, ewentualnie jeśli choć jeden z nich wywołuje funkcję exit. Sposób działania kodu 2.4 w przypadku manipulowania trzecim wątkiem ilustruje sekwencja komunikatów wyświetlanych na ekranie:

```
watek glowny: tworzenie odlaczonego watku potomnego nr 3
  watek glowny: koniec pracy, watek odlaczony pracuje dalej
  watek potomny: uniemozliwione zabicie
  watek potomny: umozliwienie zabicia
  watek potomny: zmiana wartosci zmiennej wspolnej
```

2.6 Komunikacja międzyprocesowa

Pojawienie się wielu wątków lub procesów w ramach pojedynczego programu rozwiązującego pewien określony problem powoduje, że koniecznym staje się wprowadzenie mechanizmów komunikacji –

środków, za pomocą których jeden proces (wątek) może przekazać informacje innemu procesowi (wątkowi). Także w tym względzie konieczne jest wsparcie ze strony systemów operacyjnych. Programiści tworzący programy równoległe i rozproszone korzystają z narzędzi dostarczanych przez systemy operacyjne, a także oprogramowanie wspomagające przetwarzanie równoległe i rozproszone, nazywane oprogramowaniem warstwy pośredniej. Wybrane narzędzia dostarczane przez takie oprogramowanie omówione zostaną przy prezentacji konkretnych środowisk programowania, w niniejszym punkcie opisane zostaną, także wybrane, podstawowe narzędzia systemowe.

Zagadnienie komunikacji międzyprocesowej było rozwiązywane na wiele rozmaitych sposobów od czasu wprowadzenia wielozadaniowych systemów operacyjnych w latach 60-tych XX wieku. Powstał szereg mechanizmów o różnym stopniu ogólności, różnej wydajności i, w efekcie, różnej popularności. Szerszy przegląd tych mechanizmów (takich jak np. sygnały, łącza, kolejki komunikatów) należy do dziedziny systemów operacyjnych. W niniejszym punkcie krótko wspomnimy o pamięci wspólnej, w kolejnych rozdziałach wrócimy jeszcze do sposobów komunikacji za pomocą semaforów oraz gniazd.

Pamięć wspólna jako mechanizm komunikacji międzyprocesowej pojawiła się jeszcze w czasach, kiedy nie funkcjonowały procesy wielowątkowe. Procesy, na skutek posiadania rozłącznych przestrzeni adresowych, w momencie gdy chcą skorzystać z pamięci wspólnej, muszą najpierw, za pomocą odpowiednich narzędzi systemowych, utworzyć dodatkowy obszar pamięci i sprawić, aby system operacyjny umożliwił każdemu z procesów korzystanie z tego obszaru. Później system operacyjny nie ingeruje w sam proces komunikacji, który polega na zapisie przez jeden z procesów do określonych komórek pamięci, a następnie odczycie z tych komórek przez inne procesy. Pamięć wspólna jest uznawana za najszybszy mechanizm komunikacji między procesami.

Obecnie pamięć wspólna jest rzadziej używana w programach użytkowych. Powodem jest jej niepełna przenośność pomiędzy systemami operacyjnymi oraz relatywnie skomplikowany sposób używania. W systemach Unixowych po utworzeniu obszaru pamięci przeznaczonego do współdzielenia (czyni to jeden z procesów posługując się ustaloną nazwą obszaru – jak zwykle w Unixie jest to liczba całkowita), procesy zlecają systemowi operacyjnemu dołączenie obszaru do ich przestrzeni adresowej (posługując się nazwą obszaru). System operacyjny zwraca adres obszaru wspólnego w postaci wskaźnika, po czym procesy, posługując się tym wskaźnikiem, dokonują zapisu i odczytu z obszaru pamięci wspólnej.

Fakt, że procesy komunikują się ze sobą sugeruje, że wspólnie rozwiązują pewien problem. Korzystając z pamięci wspólnej pracują współbieżnie – dlaczego więc nie umieścić ich w jednym procesie, jako dwa różne wątki? Co możemy zyskać? Najważniejszym zyskiem jest niezwykła prostota komunikacji za pomocą przestrzeni adresowej procesu, do którego należą wątki. Drugim zyskiem jest prostota zarządzania wątkami, poprzez ustawianie wartości ich atrybutów oraz przesyłanie odpowiednich sygnałów – np. zgodnie z ujednoliconym interfejsem POSIX.

Prostotę komunikacji miedzy wątkami za pomocą zmiennych wspólnych, które automatycznie przez system zarządzania wykonaniem programu umieszczane są w obszarze pamięci dostępnym dla wszystkich wątków, widać w przykładach z poprzedniego punktu (programy 2.2, 2.4). Dowolna zmienna globalna staje się nośnikiem komunikatów pomiędzy wątkami. Zapis do zmiennej globalnej dokonany przez jeden watek, staje się widoczny dla wszystkich wątków. Zupełnie inaczej niż w przypadku wielu procesów (np. kod 2.1), gdzie zmienne globalne funkcjonują w wielu kopiach, zmiana ich wartości przez jeden proces nie jest widziana przez inne procesy, a korzystanie z pamięci wspólnej wymaga szeregu dodatkowych operacji.

Komunikacja za pomocą pamięci wspólnej, tak jak ujmowaliśmy to dotychczas, oparta jest na jednym kluczowym założeniu – wszystkie wątki (ewentualnie procesy) pracują pod kontrolą jednego systemu operacyjnego. To pojedynczy system operacyjny odpowiedzialny jest za realizację komunikacji przy użyciu pamięci wspólnej. Czy możliwe jest wykorzystanie pamięci wspólnej w przypadku procesów i ich wątków uruchomionych na systemach równoległych zbudowanych z różnych

komputerów połączonych siecią, z których każdy pracuje pod kontrolą innego systemu operacyjnego?

Taka możliwość istnieje, ale interfejs programowania stosowany w takim przypadku nie jest typowym interfejsem programowania wielowątkowego, np. w stylu specyfikacji POSIX. Omawiana w następnych rozdziałach specyfikacja OpenMP programowania w modelu pamięci wspólnej, posiada implementacje dla systemów równoległych z wieloma komputerami pod kontrolą wielu instancji systemów operacyjnych. Rozwiązanie to boryka się jednak z wieloma problemami wydajnościowymi i implementacyjnymi, należąc wciąż do rozwiązań wyjątkowych i mało popularnych. W dalszej części skryptu przyjmować będziemy założenie, że programowanie i przetwarzanie wielowątkowe odbywa się z wykorzystaniem pamięci wspólnej pod kontrolą jednego systemu operacyjnego.

2.7 Zadania

1. Na podstawie kodów 2.1 i 2.2 napisz dwa programy, w których w pętli tworzone będzie 10000 nowych procesów i 10000 nowych wątków (żeby nie przekroczyć zasobów systemowych każdy proces i wątek powinien zostać zakończony przed utworzeniem następnego). Porównaj czas tworzenia i niszczenia procesu do tego samego czasu dla wątku (do pomiaru czasu można użyć np. Unixowego polecenia time).

2. Na podstawie kodu 2.3 napisz program, w którym tworzonych będzie w pętli wiele wątków (z liczba wątków określaną np. przez użytkownika) i każdy z nich otrzyma jako argument wskaźnik do pojedynczego elementu z dowolnie zaprojektowanej tablicy struktur.

3. Skompiluj i uruchom kod 2.4. Przetestuj działanie odrywania wątków i przesyłania sygnałów do wątków w Twoim środowisku uruchomieniowym.

Bibliografia

[1] George Coulouris, Jean Dollimore, Tim Kindberg, and Gordon Blair. *Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair Distributed Systems: Concepts and Design*. Pearson Education, 2012.

[2] C. Severance and K. Dowd. *High Performance Computing*. CONNEXIONS, Rice University, Houston, Texas, 2010. Available from: `http://cnx.org/content/col11136/1.5/`.

Słownik użytych tłumaczeń terminów angielskich

bezpieczeństwo wielowątkowe *thread safety*

blok kontrolny procesu *process control block*

instrukcja *statement*

jednostka centralna *central processing unit, CPU*

komputery masowo równoległe *massively parallel computers, massively parallel processors*

komunikacja międzyprocesowa (międzywątkowa) *inter-process (inter-thread) communication*

mikroprocesor masowo wielordzeniowy *many-core processor, massively multi-core processor*

mikroprocesor wielordzeniowy *multi-core processor*

mikroprocesor *microprocessor*

obiekt nieprzenikalny *opaque object*

obiekt nieprzeźroczysty *opaque object*

obliczenia *computing*

pamięć wspólna *shared memory*

polecenie *command*

proces (wątek) nadrzędny *parent process (thread)*

proces (wątek) potomny *child process (thread)*

procesor *processor, central processing unit, CPU*

przestrzeń adresowa procesu *process address space*

przestrzeń jądra *kernel space*

przetwarzanie masowo równoległe *massively parallel processing*

przetwarzanie *processing*

punkty anulowania *cancellation points*

rdzeń *core*

rozkaz *instruction*

rozproszony *distributed*

równoległy *parallel*

sekwencyjny *sequential*

stos *stack*

szeregowanie *scheduling*

uruchomić *to run*

wielozadaniowy system operacyjny *multitasking operating system*

współbieżność *concurrency*

współbieżny *concurrent*

wydajność *performance*

wykonanie *execution*

wątek główny *main thread, master thread*

wątek odłączony *detached thread*

wątek przyłączalny *joinable thread*

wątek *thread*

wątki (przestrzeni) jądra *kernel (space) threads*

wątki (przestrzeni) użytkownika *user (space) threads*

zrównoleglenie *parallelization*

środowisko uruchomieniowe *runtime environment*

Indeks