

# 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 wątkami. 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 pojęcie 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<sup>1</sup>. 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**

<sup>1</sup>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 bibliotek) 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 realizację 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 tą 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<sup>2</sup>. 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 realizację funkcji `main` kodu). Proces ten jest nazywany *procesem nadrzędnym* lub *procesem macierzystym*. W momencie wywołania funkcji `fork` system operacyjny tworzy nowy proces, *proces potomny*<sup>3</sup>. 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?

<sup>3</sup>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 zwracanim wartościom, programista może zlecić zupełnie inne zadania procesowi nadrzédnemu 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 przepływ 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 uwzględnić sekwencyjny charakter pracy każdego wątku oraz dowolny przepływ 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 nadrzédnego 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 (wieloprocessorowych), równoległe. Podobnie jak procesy, wątki w ostateczności<sup>4</sup> 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 rozkazy 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 wątki.**

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.

<sup>4</sup>W dalszej części krótko omówione zostaną także wątki zarządzane przez specjalne środowiska uruchomieniowe, co nie zmienia faktu, że wszystkie wątki konkretnego procesu w ostateczności pracują 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<sup>5</sup>. 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

<sup>5</sup>W prezentowanych tu ogólnych analizach pomijając będziemy fakt, że zmienne różnych typów zajmują obszary pamięci różnej wielkości. Dla uproszczenia opisu przyjmujemy 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*)<sup>6</sup>.

**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 przerwanej 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 istotną wadę – zazwyczaj nie pozwalają na pracę równoległą, tylko współbieżną w przeplocie. Praca równoległa pojawia się wtedy, kiedy środowisko uruchomieniowe

<sup>6</sup>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 różnorodności implementacji działania wątków jest też różnorodność nazw jakimi się posługuje na określenie 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 równoległą pracę wątków oraz jakie są parametry wydajnościowe tego mechanizmu. Programista może być zainteresowany także tym czy sposób zarządzania wątkami narzuca jakieś ograniczenia związane z synchronizacją pracy wątków (w przypadku wątków, które same decydują o swoim wyłączeniu, 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 znajdują 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 następująca:

```
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<sup>7</sup>. 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

<sup>7</sup>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("Wątek otrzymał argument %d. Wartość zmiennej wspólnej 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 warto jest zwrócić uwagi.

Procedura zlecona 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 w 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 wątki 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 zleconej 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 się poza jej stosiem, a więc są dostępne także innym wątkom.

W drugim przypadku procedura dokonuje kopiowania wartości na swój stos. W tym celu deklaruje zmienną 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**<sup>8</sup>. W programie może to zostać zrealizowane poprzez jawne przesłanie wątkowi odpowiedniego sygnału (procedura `pthread_kill` – odpowiednik

<sup>8</sup>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. Wątek zakończy swoją 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 pracę sam czy został anulowany, ale tylko w przypadku jeśli wątek 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 wątek główny w swoim działaniu jest skoordynowany z działaniem wątku potomnego i np. oczekuje na wynik jego działania, aby móc kontynuować swoją pracę.

Kiedy wątek główny nie jest związany w swojej pracy z działaniem wątku pobocznego, wywołanie `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 zapytania. 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 przyjęcie, 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 mają 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
        PRAWDOPODOBNIENIE zostal zabity\n");
else
    printf("\twatek glowny: odlaczony watek potomny
        PRAWDOPODOBNIENIE 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 wątek kończy pracę. W momencie kończenia pracy na skutek anulowania, proces potomny przesyła odpowiednią wartość zwrrotną, 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
```

wątek główny: wysłanie sygnału zabicia wątku odłączonego  
 wątek główny: czy wątek potomny został zabity?  
 wątek główny: sprawdzanie wartości zmiennej wspólnej  
 wątek potomny: umożliwienie zabicia  
 wątek główny: odłączony wątek potomny PRAWDOPODOBNIEM został 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 niszczone 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:

wątek główny: tworzenie odłączonego wątku potomnego nr 3  
 wątek główny: koniec pracy, wątek odłączony pracuje dalej  
 wątek potomny: uniemożliwienie zabicia  
 wątek potomny: umożliwienie zabicia  
 wątek potomny: zmiana wartości zmiennej wspólnej

## 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 przysyłanie odpowiednich sygnałów – np. zgodnie z ujednoliconym interfejsem POSIX.

Prostotę komunikacji między 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 wątek, 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 liczbą 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.