8 Wątki

8.1 Wprowadzenie

Wiele rozwiązywanych problemów można podzielić na zadania cząstkowe, które dają się wykonać niemal niezależnie. Każde z takich zadań można by powierzyć oddzielnemu procesowi. Jednak tworzenie nowego procesu w systemie jest na ogół dość kosztowne. Dodatkowe narzuty w takim przypadku wiązałyby się z powielaniem wielu struktur danych oraz komunikacją międzyprocesową. Bardziej naturalnym rozwiązaniem jest rozwidlenie procesu w jego przestrzeni adresowej i powierzenie poszczególnych zadań tego typu ,,podprocesom", zwanym powszechnie **wątkami** (ang. *threads*). Istnieją pewne klasy zagadnień, które szczególnie nadają się do rozwiązań w postaci programów wielowątkowych. Przykładami są tu różnego rodzaju monitory i demony obsługujące wiele jednoczesnych połączeń, programy obsługujące równocześnie wiele okien, różnorodne zadania typu producent–konsument itd.

Każdy wątek ma swój własny stos, zestaw rejestrów, licznik programowy, indywidualne dane, zmienne lokalne, maskę sygnałów i informacje o stanie. Wszystkie wątki tego samego procesu mają tę samą przestrzeń adresową, ogólną obsługę sygnałów, pamięć wirtualną, dane oraz wejście/wyjście. W ramach procesu wielowątkowego każdy wątek wykonuje się niezależnie i asynchronicznie. Komunikacja między wątkami jest ułatwiona, ponieważ mają one dostęp do wspólnych danych.

Istnieją dwa podstawowe poziomy tworzenia wątków i zarządzania nimi: **poziom użytkownika** oraz **poziom jądra**. W przypadku wątków **poziomu użytkownika** system operacyjny zawiera podsystem wykonawczy zarządzający działaniem wątków. Jądro systemu nic nie wie o wątkach, a proces wielowątkowy traktuje tak jak każdy inny proces. Wątki zaimplementowane w ten sposób nie wymagają dużego dodatkowego nakładu pracy ze strony systemu operacyjnego i łatwo dają się rozbudowywać. W przypadku wątków **poziomu jądra** system operacyjny zapewnia bezpośrednią obsługę wątków. Zaletą tego typu modelu jest możliwość efektywnej realizacji wielowątkowości w komputerze wieloprocesorowym. Ponieważ jądro systemu bezpośrednio zarządza wątkami, więc może je łatwo rozdzielać między różne procesory, zwiększając tym samym wydajność wykonywania procesu. Przełączanie wątków w tym przypadku jest jednak wolniejsze, gdyż angażuje jądro. W praktyce często spotyka się próby łączenia zalet obu powyższych rodzajów wątków w postaci różnego rodzaju modeli mieszanych. W jednym rodzaju implementacji stosuje się odwzorowanie typu **,,jeden na jeden"**, w którym system operacyjny przyporządkowuje jeden wątek poziomu użytkownika do jednego wątku poziomu jądra (np. systemy Windows NT, OS/2). Innym rozwiązaniem jest relacja typu **,,wiele na jeden"**, gdzie wiele wątków poziomu użytkownika jest odwzorowywanych w jeden wątek poziomu jądra. Wreszcie są też modele mieszane typu **,,wiele na wiele"**, w których istnieje wiele wątków poziomu użytkownika oraz pewna pula wątków poziomu jądra. Wątki jądra działają w obrębie tzw. *procesów lekkich* (ang. *light-weight processes, LWP*) i są obsługiwane przez system operacyjny. Tego typu rozwiązanie zastosowano np. w systemie Solaris firmy Sun Microsystems (jednego z pionierów w dziedzinie wielowątkowości).

Istnieje wiele różnych bibliotek funkcji wątkowych, często związanych z określonymi

systemami operacyjnymi[6]. My zajmiemy się wątkami standardu POSIX (zwanymi również *P-wątkami*, ang. *Pthreads*) i będziemy korzystać z biblioteki funkcji Thread Library (rozdział 3t podręcznika man) zgodnej z tą normą. Zaletą takiego podejścia jest przenośność kodu, jako że wiele współczesnych systemów operacyjnych posiada biblioteki funkcji zgodne ze standardem POSIX. Niemniej jednak niektóre systemy zawierają specyficzne dla swojego środowiska implementacje wątków, które mogą być bardziej wydajne od *P-wątków*.

Aby móc korzystać z biblioteki funkcji wątkowych standardu POSIX należy użyć następujących dyrektyw:

```
#define _REENTRANT
#include <pthread.h>
```

Dyrektywa `#define _REENTRANT` musi wystąpić przed wszystkimi dyrektywami `#include`. Oznacza ona dany kod jako *kod wielokrotnego użytku* (ang. *reentrant code*), czyli taki, do którego można wiele razy wchodzić (ang. *enter*). Dodatkowo podczas linkowania (konsolidacji) programu należy użyć opcji `-lpthread`, powodującej dołączenie odpowiedniej biblioteki funkcji wątkowych.

8.2 Tworzenie wątków

Każdy proces zawiera przynajmniej jeden główny wątek początkowy, tworzony przez system operacyjny podczas powoływania procesu do życia. Aby do procesu dodać nowy wątek wykonania, należy użyć funkcji pthread_create. Nowy wątek będzie działał razem

Pliki włączane	`<pthread.h>`		
Prototyp	`int pthread_create(pthread_t *pthreadID, const pthread_attr_t *attr, void * (*start_fun) (void *), void *arg);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	$\neq 0$	Nie

z poprzednio utworzonymi wątkami danego procesu (może także, w razie potrzeby, działać razem z innymi wątkami innych procesów).

Pierwszy parametr funkcji pthread_create, o nazwie pthreadID, jest wskaźnikiem na obiekt typu pthread_t (w rzeczywistości jest to liczba całkowita bez znaku), który w przypadku pomyślnego wywołania funkcji będzie niepowtarzalnym identyfikatorem wątku (w przypadku argumentu równego `NULL` identyfikator nie zostanie zwrócony). Drugi parametr, o nazwie attr, wskazuje na dynamicznie rezerwowaną strukturę atrybutów wątku (rozmiar stosu, adres, polityka planowania przydziału procesora, stan priorytetu, stan odłączenia itp.). Aby nadać wątkowi domyślne atrybuty systemu wystarczy jako drugi

[6]Na przykład w systemie Linux wątki można tworzyć przy pomocy funkcji systemowej clone (patrz np. podręcznik man).

argument tej funkcji przekazać wskaźnik `NULL`. Do zmiany atrybutów wątku służy funkcja pthread_attr_init (patrz podręcznik man). Trzeci parametr pthread_create jest wskaźnikiem na definiowaną przez użytkownika funkcję, która będzie wykonana jako nowy wątek. Funkcja ta powinna mieć jeden parametr w postaci wskaźnika na void i zwracać wskaźnik na void. Jeżeli zwracana przez funkcję wartość jest wskaźnikiem na inny typ niż void, to należy zastosować rzutowanie: (void * (*) ()). Wskaźnik na rzeczywisty argument przekazywany do funkcji definiowanej przez użytkownika jest czwartym parametrem pthread_create – jest on również wskaźnikiem na void. Aby przekazać kilka argumentów do funkcji użytkownika, należy zdefiniować odpowiednią strukturę, zadeklarować ją jako static i zainicjować. Następnie w wywołaniu pthread_create wskaźnik na tę strukturę należy zrzutować na wskaźnik na void.

Nowo utworzony wątek zaczyna się od wykonania funkcji użytkownika wywołanej przez funkcję pthread_create i działa do czasu aż:

- zakończy się funkcja (jawnie bądź niejawnie),
- zostanie wywołana funkcja pthread_exit (omawiana poniżej),
- wątek zostanie anulowany za pomocą funkcji pthread_cancel (patrz man),
- zakończy się proces macierzysty wątku (jawnie bądź niejawnie),
- jeden z wątków wykona funkcję exec.

W przypadku niepowodzenia funkcja pthread_create zwraca wartość niezerową oznaczającą kod błędu, np.

`EAGAIN (11)`: przekroczony został limit systemowy liczby wątków lub procesów LWP;
`ENOMEM (12)`: brak pamięci do utworzenia nowego wątku;
`EINVAL (22)`: nieprawidłowa wartość argumentu attr.

8.3 Kończenie działania wątków

Do kończenia pracy wątku służy funkcja biblioteczna pthread_exit, która działa podobnie do standardowej funkcji exit. Ma ona tylko jeden parametr będący wskaźnikiem na

Pliki włączane	`<pthread.h>`		
Prototyp	`void pthread_exit(void *status);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
			Nie

wartość stanu wątku. Wskaźnik ten jest zwracany, jeśli kończony wątek nie jest wątkiem odłączonym (ang. *detached*). W chwili zakończenia wątek zwraca swoje zasoby. Jeżeli funkcja wykonywana w ramach wątku zakończy się (jawnie lub niejawnie), to funkcja pthread_exit zostanie automatycznie wywołana przez system.

8.4 Podstawowe operacje zarządzania wątkami

Procesowi, w którym działa wątek utworzony za pomocą funkcji pthread_create można nakazać czekanie na zakończenie tego wątku – ale tylko takiego, który nie został zadeklarowany jako wątek odłączony (ang. *detached*). Do tego celu służy funkcja pthread_join.

Pliki włączane	`<pthread.h>`		
Prototyp	`int pthread_join(pthread_t threadID, void **status);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	$\neq 0$	Nie

Pierwszym jej parametrem jest identyfikator wątku, zwracany przez funkcję pthread_create. Drugi parametr jest wskaźnikiem na statyczne miejsce w pamięci, w którym zostanie zapisany stan zakończenia wątku. Stan ten to argument, który będzie przekazany funkcji pthread_exit lub wartość `PTHREAD_CANCELED`, w przypadku gdy wątek zostanie anulowany. Użycie `NULL` jako drugiego argumentu funkcji pthread_join spowoduje zignorowanie informacji o stanie. W przypadku niepowodzenia funkcja pthread_join zwraca wartość niezerową będącą odpowiednim kodem błędu (patrz man pthread_join).

Wątek może być wcielany (ang. *join*) tylko przez jeden inny wątek. Wcielanie jest podobne do czekania (wait) w procesie macierzystym na proces potomny (utworzony przy pomocy funkcji fork). Istotną różnicą jest jednak to, że wątek można w dowolnej chwili odłączyć wywołując funkcję pthread_detach, podczas gdy z procesem potomnym nie da się tego zrobić. Funkcja biblioteczna pthread_detach ma tylko jeden parametr w postaci

Pliki włączane	`<pthread.h>`		
Prototyp	`int pthread_detach(pthread_t threadID);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	$\neq 0$	Nie

identyfikatora wątku. Zakończona pomyślnie spowoduje odłączenie wątku. W przypadku niepowodzenia zwróci odpowiedni kod błędu (patrz man pthread_detach).

W momencie zakończenia wątku odłączonego jego zasoby są automatycznie zwracane systemowi. Natomiast wątek nie odłączony, który nie zostanie wcielony w inny wątek, nie zwalnia zasobów po zakończeniu swojego działania. Zasoby te zostaną zwolnione dopiero z chwilą zakończenia jego procesu macierzystego.

8.5 Synchronizacja wątków

Ponieważ wątki w ramach procesu operują na wspólnych strukturach danych, dlatego, aby nie dopuścić do niespójności danych, potrzebny jest odpowiedni mechanizm synchronizacji. Wątki standardu POSIX można synchronizować na wiele sposobów. Jedną z najprotszych metod jest zastowanie blokad wzajemnie wykluczających, tzw. **muteksów** (ang. *mutual exclusion*). Muteks jest rodzajem semafora dwustanowego, który wątki

mogą ,,posiadać". Wartość `0` oznacza, że muteks jest otwarty, tzn. zezwala na dostęp, a wartość `1`, że muteks jest zamknięty, tzn. zabrania dostępu (konwencja odwrotna niż dla semaforów). Muteks może zostać zamknięty przez dowolny wątek będący w jego zasięgu, natomiast może (i powinien) zostać *otwarty tylko przez wątek, który go zamknął*. Operacje wykonywane na muteksach są niepodzielne (atomowe). Muteks może być **wewnątrzprocesowy** (ang. *intra-process*) – do synchronizowania wątków w obrębie jednego procesu, lub **międzyprocesowy** (ang. *inter-process*) – do synchronizowania wątków różnych procesów. Muteks międzyprocesowy należy dodatkowo odwzorować w obszar pamięci wspólnej (dzielonej) odpowiednich procesów.

Z muteksem związany jest pewien zbiór atrybutów, który może być modyfikowany za pośrednictwem funkcji bibliotecznych: pthread_mutex_init, pthread_mutexattr_init lub pthread_mutexattr_setpshared (szczegóły można znaleźć na stronach podręcznika man). W dalszej części będziemy zajmować się tylko muteksami wewnątrzprocesowymi o domyślnych atrybutach. Aby utworzyć taki muteks o nazwie `myMutex` i zainicjować go jako ,,otwarty" (tzn. o wartości `0`), wystarczy następująca definicja:

```
pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
```

gdzie `pthread_mutex_t` jest typem zmiennych muteksowych (w rzeczywistości pewna struktura), a predefiniowana stała `PTHREAD_MUTEX_INITIALIZER` służy do inicjowania muteksu jako ,,otwarty". Powyższą operację można też zrealizować w inny sposób:

```
pthread_mutex_t myMutex;
pthread_mutex_init(&myMutex, NULL);
```

Pliki włączane	`<pthread.h>`		
Prototyp	`int pthread_mutex_init(pthread_mutex_t *pmutex, const pthread_mutexattr_t *attr);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	$\neq 0$	Nie

Funkcja pthread_mutex_init służy do inicjowania muteksów. Pierwszy jej parametr jest wskaźnikiem na muteks, a drugi wskaźnikiem na przygotowany wcześniej obiekt atrybutów. Jeżeli jako drugi argument zostanie użyty wskaźnik `NULL`, to muteks zostanie zainicjowany wartościami domyślnymi dla systemu. W przypadku niepowodzenia, funkcja zwraca kod `EINVAL`, oznaczający podanie niewłaściwej wartości któregoś z argumentów.

Istnieją cztery funkcje biblioteczne do wykonywania operacji na utworzonych i zainicjowanych muteksach. Każda z nich ma pojedynczy parametr będący wskaźnikiem na muteks. Funkcja pthread_mutex_lock służy do zamykania muteksu. Wywołanie jej dla muteksu, który jest już zamknięty spowoduje zablokowanie wątku do czasu otwarcia muteksu. Natomiast wywołanie jej dla muteksu otwartego spowoduje jego zamknięcie i zawłaszczenie przez wywołujący wątek. *Uwaga: Ponowne wywołanie funkcji zamykającej przez właściciela muteksu może doprowadzić do zakleszczenia!* Funkcja pthread_mutex_unlock służy do otwierania muteksu, ale tylko przez wątek, który zamknął dany muteks – *wywołanie*

jej przez inny wątek może mieć nieprzewidywalne skutki! Funkcja pthread_mutex_trylock działa podobnie jak pthread_mutex_lock, ale z taką różnicą, iż nie powoduje blokowania wywołującego ją wątku, jeśli dany muteks jest już zamnięty (tzn. jest to nieblokujące zamykanie muteksu). Wreszcie funkcja pthread_mutex_destroy jest przeznaczona do usuwania wskazanego muteksu – faktycznie sprawia, że muteks staje się ,,niezainicjowany", użytkownik natomiast musi zadbać o zwolnienie pamięci wskazywanej przez wskaźnik muteksu. Wszystkie te funkcje w przypadku niepowodzenia zwracają odpowiedni kod błędu. W szczególności pthread_mutex_destroy zwraca wartość `EBUSY (16)`, jeżeli wskazywany muteks jest już zamknięty.

Pliki włączane	`<pthread.h>`		
Prototyp	`int pthread_mutex_lock(pthread_mutex_t *pmutex);`		
Prototyp	`int pthread_mutex_unlock(pthread_mutex_t *pmutex);`		
Prototyp	`int pthread_mutex_trylock(pthread_mutex_t *pmutex);`		
Prototyp	`int pthread_mutex_destroy(pthread_mutex_t *pmutex);`		
Zwracana wartość	Sukces	Porażka	Czy zmienia `errno`
	0	$\neq 0$	Nie

Oprócz muteksów istnieje jeszcze kilka innych mechanizmów do synchronizacji wątków, w tym semafory. Informacje na ten temat można znaleźć m.in. na stronach podręcznika systemowego man (patrz np. man sem_init).

ĆWICZENIE 9: Wzajemne wykluczanie dla Wątków: Muteksy

Przy pomocy **muteksów** zaimplementować zadanie **wzajemnego wykluczania** dla **wątków**. Do demonstracji działania programu można użyć sekwencji sterujących konsoli (patrz np. plik `hello.c` w katalogu `StartSO`). Niech na przykład wątek wykonując swoją sekcję prywatną wypisuje odpowiedni komunikat po lewej stronie okna konsoli, natomiast będąc w sekcji krytycznej drukuje informacje po prawej stronie (w tym samym wierszu). Każdy wątek może kilka razy powtarzać powyższy cykl. Przy poprawnie zrealizowanym zadaniu wzajemnego wykluczania, po prawej stronie okna konsoli w danym momencie powinien zgłaszać się co najwyżej jeden wątek, pozostałe natomiast powinny zgłaszać się po lewej stronie.

Do zademonstrowania operacji na zasobie dzielonym użyć np. wspólnej (globalnej) zmiennej licznikowej, zainicjowanej wartością `0`. Niech każdy z wątków na początku sekcji krytycznej przypisuje jej wartość swojemu prywatnemu licznikowi, następnie zwiększa wartość tego prywatnego licznika o `1`, a po pewnym czasie (użyć np. funkcji sleep) przypisuje jego wartość wspólnemu licznikowi. Sprawdzić, czy po zakończeniu działania wszystkich wątków wartość tego wspólnego licznika jest taka jaka powinna być.

ĆWICZENIE 10: Algorytm Piekarni: P-wątki

Przy pomocy *P-wątków* zaimplementować **algorytm piekarni** podany na wykładzie [15]. Poprawność działania programu zademonstrować jak w ćw. 9.