

Systemy Operacyjne 2:
„Wątki *pthread*”

dr inż. Arkadiusz Chrobot

18 kwietnia 2013

1. Wprowadzenie

Wątki podobnie jak procesy umożliwiają współbieżną realizację czynności w wykonywanym programie. Domyślnie w ramach każdego procesu działa pojedynczy wątek. Liczba wątków może zostać zwiększona jeśli system operacyjny dostarcza mechanizmów ich obsługi, które mogą być zaimplementowane w przestrzeni użytkownika, w jądrze systemu lub hybrydowo (i w przestrzeni użytkownika, i w jądrze systemu). Nowe wątki tworzone są zawsze w obrębie istniejącego procesu. Każdy wątek współdzieli zasoby z innymi wątkami działającymi w ramach tego samego procesu. Jednym z tych zasobów jest przestrzeń adresowa, dzięki czemu nie trzeba dodatkowych zabiegów, aby zrealizować pamięć dzieloną, jest ona po prostu domyślnie dostępna. Z drugiej strony wymusza to ostrożne korzystanie ze zmiennych, które nie są lokalne. Współdzielenie zasobów eliminuje konieczność ich ochrony przed wątkami, a więc zmniejsza ilość informacji jaką trzeba zapamiętać przy przełączaniu kontekstu, redukując tym samym ilość czasu jaką trzeba poświęcić na tę czynność. Z tego względu wątki są czasem określane mianem „lekkich procesów”¹. Wątek jest mniejszą (bardziej drobnoziarnistą) jednostką pracy niż proces.

2. Wątki w Linuksie

Dla programistów aplikacji system Linux dostarcza API wątków, które jest zgodne w większości szczegółów ze standardem POSIX. W przeciwieństwie do innych systemów operacyjnych (nawet tych, które są zgodne z Uniksem) Linux nie posiada osobnych mechanizmów jądra do obsługi wątków. W jego przypadku wątki są realizowane jako procesy, które współdzielą swoje zasoby. Z tego względu powstały wspomniane odstępstwa od standardu. Wysłanie sygnału w Linuksie z innego procesu do procesu w ramach którego działają wątki jest traktowane jako wysłanie sygnału zawsze do wątku głównego. Wywołanie jednej z funkcji *exec* powoduje automatyczne zakończenie wszystkich wątków, które zostały stworzone w ramach procesu, który wywołał tę funkcję.

3. Oprogramowywanie wątków

Aby obsługiwać wątki POSIX, nazywane w skrócie wątkami *threads* należy w programie włączyć plik nagłówkowy `pthread.h` i skompilować program z dodaną opcją `-lpthread`.

3.1. Obsługa wątków

Poniżej znajduje się lista funkcji związanych z zarządzaniem wątkami.

- Funkcja `pthread_create()` tworzy nowy wątek. Przyjmuje ona cztery argumenty wywołania. Pierwszy jest wskaźnikiem na zmienną typu `pthread_t`, w której zostanie zapisany numer identyfikacyjny nowego wątku. Drugi jest wskaźnikiem na strukturę typu `pthread_attr_t`, która określa atrybuty wątku. Trzeci parametr jest wskaźnikiem na funkcję o prototypie `void *func(void *)`, która ma być realizowana w ramach wątku. Ostatni parametr `pthread_create()` jest wskaźnikiem typu `void *` i służy do przekazania argumentów wywołania funkcji realizowanej w ramach wątku (jest podstawiany jako argument formalny tej funkcji). Jeśli utworzenie wątku się powiedzie `pthread_create()` zwróci zero, w przeciwnym przypadku wartość różną od zera.
Szczegóły: `man pthread_create`.
- Wątek kończy się z chwilą powrotu z funkcji realizowanej w jego ramach lub jeśli bezpośrednio wywoła funkcję `pthread_exit()`. Ta funkcja nie zwraca żadnej wartości, ale przyjmuje wskaźnik typu `void *` na zmienną przechowującą status zakończenia wątku.
Szczegóły: `man pthread_exit`.
- Funkcja `pthread_join()` jest odpowiednikiem funkcji `wait()` dla procesów. Przyjmuje ona dwa argumenty. Pierwszy jest numerem identyfikacyjnym wątku na którego zakończenie ta funkcja czeka, drugi wskaźnikiem typu `void *` na zmienną w której zostanie zapisany status zakończenia wątku. Funkcja zwraca wartość zero, jeśli jej wykonanie zakończyło się prawidłowo lub wartość

¹Pojęcie to nie zawsze jest tożsame z pojęciem wątku, dlatego w tej instrukcji nie będzie stosowane wymiennie.

różną od zera w przeciwnym przypadku.
Szczegóły: `man pthread_join`.

- Funkcja `pthread_self()` zwraca ID wątku, który ją wywołał. Nie pobiera żadnych argumentów wywołania.
Szczegóły: `man pthread_self`.
- Funkcja `pthread_equal()` służy do porównywania ID dwóch wątków. Zwraca zero jeśli są one różne, lub wartość różną od zera jeśli są równe.
Szczegóły: `man pthread_equal`.
- Funkcja `pthread_attr_init()` służy do inicjalizacji struktury atrybutów wątku. Zwykle taką strukturę inicjalizuje się przed utworzeniem wątku, zmienia się wartości domyślne atrybutów, przekazuje się je do nowo tworzonego wątku i wykonuje deinicjalizację struktury atrybutów. Funkcja `pthread_attr_init()` przyjmuje wskaźnik na strukturę atrybutów i zwraca zero w przypadku prawidłowego zakończenia lub wartość różną od zera w przeciwnym przypadku.
Szczegóły: `man pthread_attr_init`.
- Funkcja `pthread_attr_destroy()` dokonuje deinicjalizacji struktury atrybutów wątku. Przyjmuje ten sam parametr co funkcja `pthread_attr_init()` i zwraca te same wartości.
Szczegóły: `man pthread_attr_init`.
- Do obsługi struktury typu `pthread_attr_t` zdefiniowano wiele funkcji, które pozwalają odczytywać lub zmieniać atrybuty wątków. Takimi funkcjami są `pthread_attr_setdetachstate()` oraz `pthread_attr_getdetachstate()`. Pierwsza ustawia atrybut odpowiedzialny za tworzenie wątku łączonego lub rozdzielnego. Druga pobiera wartość tego atrybutu. Obie funkcje zwracają zero jeśli zakończą się prawidłowo lub wartość różną od zera w przeciwnym przypadku. Obie również jako pierwszy parametr pobierają wskaźnik do struktury atrybutów. Funkcja `pthread_attr_getdetachstate()` jako drugi argument wywołania pobiera wskaźnik do zmiennej typu `int`, natomiast komplementarna do niej funkcja `pthread_attr_setdetachstate()` wartość atrybutu `detached`. Domyślnie tą wartością jest `PTHREAD_CREATE_JOINABLE`. Wątek z takim atrybutem po zakończeniu pozostaje w systemie w stanie będącym odpowiednikiem stanu zombie dla procesów, do momentu wywołania dla niego funkcji `pthread_join()`. Jeśli wątek zostanie utworzony z atrybutem `PTHREAD_CREATE_DETACHED` to jest całkowicie usuwany z systemu po zakończeniu. Inne funkcje tego typu są związane z parametrami polityki szeregowania wątków i nie będą tu opisywane.
Szczegóły: `man pthread_attr_init`.
- Funkcja `pthread_cancel()` służy do anulowania (ang. *cancel*) wątku. Jeśli wątek otrzyma informację, że jest anulowany, to jego zachowanie zależne jest od ustawień dokonywanych za pomocą jednej z dwóch funkcji opisanych niżej. Funkcja zwraca zero jeśli wykona się prawidłowo, lub wartość różną od zera w przeciwnym przypadku.
Szczegóły: `man pthread_cancel()`.
- Funkcja `pthread_setcanceltype()` pozwala oznaczyć wątek, który ją wywoła jako anulowany asynchronicznie (`PTHREAD_CANCEL_ASYNCHRONOUS`) lub synchronicznie (`PTHREAD_CANCEL_DEFERRED`). W pierwszym przypadku wątek może być anulowany w dowolnym momencie wykonania, w drugim wątek może być anulowany dopiero wtedy, gdy jego wykonanie osiągnie punkt anulowania (ang. *cancelation point*). Opisywana funkcja przyjmuje dwa argumenty, pierwszym jest jedną z dwóch stałych podanych wyżej, a drugim wskaźnikiem na zmienną typu `int`. Zwraca zero w przypadku pomyślnego wykonania lub wartość różną od zera w przeciwnym przypadku.
Szczegóły: `man pthread_setcanceltype`.
- Funkcja `pthread_setcancelstate()` przyjmuje parametry tego samego typu co opisywana wyżej. Włącza (`PTHREAD_CANCEL_ENABLE`) lub wyłącza (`PTHREAD_CANCEL_DISABLE`) możliwość anulowania wątku. Wartości zwraca według tego samego schematu co funkcje opisane wyżej.
Szczegóły: `man pthread_setcancelstate`.
- Funkcja `pthread_testcancel()` służy do tworzenia punktu anulowania, jeśli wątek jest anulowany synchronicznie. Nie przyjmuje żadnego argumentu, ani nie zwraca żadnej wartości. Jej działanie

polega na zakończeniu bieżącego wątku. Niektóre z funkcji biblioteki *threads* zawierają wywołanie tej funkcji.

Szczegóły: `man pthread_testcancel`.

- Funkcja `pthread_key_create()` tworzy klucz służący do odwoływania się do zmiennych nielokalnych, ale należących do danego wątku (tzn. takich do których może się odwoływać tylko ten wątek). Takie zmienne nazywane są zmiennymi prywatnymi wątków. Jako pierwszy parametr przyjmuje wskaźnik do zmiennej typu `pthread_key_t`, jako drugi wskaźnik na tzw. funkcję sprzątającą. Zwraca zero, jeśli wykonała się poprawnie, lub inną wartość w przypadku niepowodzenia.
Szczegóły: `man pthread_key_create`.
- Funkcja `pthread_key_destroy()` niszczy klucz służący do odwołań do zmiennych prywatnych wątku. Jako argument wywołania przyjmuje ten klucz i zwraca zero w przypadku powodzenia lub wartość różną od zera w przeciwnym razie.
Szczegóły: `man pthread_key_destroy`.
- Funkcja `pthread_setspecific()` ustawia wartość zmiennej prywatnej wątku. Jako argumenty przyjmuje klucz do zmiennej własnej i wskaźnik do zmiennej zawierającej wartość jaka ma być nadana zmiennej prywatnej.
Szczegóły: `man pthread_setspecific`.
- Funkcja `pthread_getspecific()` zwraca wskaźnik typu `void *` na zmienną prywatną wątku, a jako argument wywołania przyjmuje klucz do tej zmiennej.
Szczegóły: `man pthread_getspecific`.
- Funkcja `pthread_cleanup_push()` służy do rejestracji funkcji sprzątających. Te funkcje są wywoływane jeśli wątek zostanie anulowany lub wywoła `pthread_exit()`. Jeśli wątek zarejestruje więcej niż jedną funkcję sprzątającą, to będą one wywołane w porządku odwrotnym do kolejności ich rejestracji. Opisująca funkcja przyjmuje dwa argumenty. Pierwszym jest wskaźnikiem na funkcję sprzątającą o prototypie `void *func(void *)`, a drugim wskaźnikiem typu `void *` na argument dla tej funkcji. Funkcja nic nie zwraca.
Szczegóły: `man pthread_cleanup_push`.
- Funkcja `pthread_cleanup_pop()` służy do wyrejestrowania funkcji sprzątającej, która została zarejestrowana jako ostatnia. Nie zwraca żadnej wartości, a jako argument wywołania pobiera wartość typu `int`. Jeśli jest ona różna od zera, to funkcja sprzątająca przed wyrejestrowaniem jest wykonywana, w przeciwnym razie jest tylko wyrejestrowywana.
Szczegóły: `man pthread_cleanup_pop`.
- Funkcja `pthread_kill()` służy do wysyłania sygnałów do wątków **wyłącznie wewnątrz procesu w którym te wątki działają**. Przyjmuje ona dwa parametry. Pierwszym jest ID wątku, drugim numer sygnału. Zwraca zero jeśli wykona się poprawnie, lub wartość różną od zera w przeciwnym przypadku.
Szczegóły `man pthread_kill`.

3.2. Muteksy

Muteksy są zmiennymi synchronizującymi podobnymi do semaforów, ale mogą przyjmować tylko dwie wartości. W bibliotece obsługi wątków *threads* są zdefiniowane za pomocą typu `pthread_mutex_t`. Oto funkcje związane z ich obsługą:

- Funkcja `pthread_mutex_init()` służy do inicjalizowania muteksów. Przyjmuje dwa argumenty. Pierwszym jest wskaźnik do muteksu, a drugim wskaźnik na strukturę atrybutów muteksu typu `pthread_mutexattr_t`. Funkcja ta zwraca zawsze wartość zero. Mutex może być zainicjalizowany bezpośrednio, np. można mu przypisać wartość `PTHREAD_MUTEX_INITIALIZER`.
Szczegóły: `man pthread_mutex_init`.
- Funkcja `pthread_mutex_lock()` zajmuje mutex lub zawiesz wykonanie wątku jeśli był on już zajęty. Jako argument przyjmuje wskaźnik na mutex. Zwraca zero w przypadku wykonania poprawnego lub wartość różną od zera w przeciwnym przypadku.

- Funkcja `pthread_mutex_unlock()` zwalnia mutex. Jako argument przyjmuje jego wskaźnik. Zwraca wartości według tego samego schematu co `pthread_mutex_lock()`.
Szczegóły: `man pthread_mutex_unlock`.
- Funkcja `pthread_mutex_trylock()` działa jak `pthread_mutex_lock()`, ale jeśli mutex jest zajęty, to nie blokuje wątku, tylko zwraca błąd `EBUSY`.
Szczegóły: `man pthread_mutex_trylock`.

3.3. Semafory

W systemie Linux oprócz implementacji semaforów System V dostępna jest też implementacja zgodna ze standardem POSIX. Pierwsza dostępna jest wyłącznie dla procesów, a z drugiej mogą korzystać zarówno procesy jak i wątki. W tym rozdziale zostanie opisane użycie semaforów POSIX do synchronizacji wątków. Nie będzie przedstawiony sposób korzystania z tych semaforów przez procesy. Aby posłużyć się semaforami POSIX należy do programu włączyć plik nagłówkowy `semaphore.h` i skompilować go z flagą `-lrt`. Semafor jest zmienną typu `sem_t`. Do obsługi semaforów używane są następujące funkcje:

- Funkcja `sem_init()` służy do inicjalizacji semafora. Przyjmuje trzy argumenty wywołania. Pierwszym jest wskaźnik na semafor, drugim flaga określająca, czy semafor będzie dostępny dla wątków, czy dla procesów. Aby był dostępny dla wątków wartość tego argumentu musi wynosić zero. Trzecim argumentem jest początkowa wartość semafora (typu `int`). Funkcja zwraca zero jeśli wykona się prawidłowo lub `-1` w przeciwnym przypadku. Wtedy także ustawia wartość zmiennej `errno`.
Szczegóły: `man sem_init`.
- Funkcja `sem_post()` służy do zwalniania semafora poprzez zwiększenie jego wartości o jeden. Jeżeli wartość wynikowa będzie większa od zera a inny wątek był uspijony na semaforze, to zostanie on obudzony. Funkcja jako argument przyjmuje wskaźnik na semafor. Wartości zwraca według schematu opisanego wyżej.
Szczegóły: `man sem_post`.
- Funkcja `sem_wait()` zajmuje semafor. Jako argument przyjmuje wskaźnik na semafor. Usypia wątek, jeśli wartość semafora jest równa zero. Zwraca wartości w ten sam sposób jak poprzednio opisywane funkcje.
Szczegóły: `man sem_wait`.
- Funkcja `sem_trywait()` zajmuje semafor, ale nie usypia wątku kiedy wartość semafora jest równa zero, tylko zwraca błąd (nadaje zmiennej `errno` wartość `EAGAIN`).
Szczegóły: `man sem_trywait`.
- Funkcja `sem_getvalue()` zwraca wartość semafora zapisując ją w zmiennej typu `int`, do której wskaźnik przekazywany jest jej jako drugi argument wywołania. Jako pierwszy argument przekazywany jest wskaźnik na semafor. Wartości zwracane są przez tę funkcję według schematu opisanego wyżej.
Szczegóły: `man sem_getvalue`.
- Funkcja `sem_destroy()` usuwa semafor, który został zainicjalizowany przy pomocy `sem_init()`. Przyjmuje wskaźnik na semafor jako argument wywołania, a wartości zwraca według tego samego schematu co pozostałe funkcje obsługujące semafory dla wątków.
Szczegóły: `man sem_destroy`.

3.4. Zmienne warunkowe

Zmienne warunkowe są trzecim środkiem synchronizacji dostępnym dla wątków *threads*. Służą do sygnalizowania ukończenia wykonania pewnego zdania. Przykładowo jeden z wątków może prowadzić obliczenia, a drugi czekać na ich wynik na zmiennej sygnałowej. Zmienne te są określone typem `pthread_cond_t` i obsługiwane za pomocą następujących funkcji:

- Funkcja `pthread_cond_init()` inicjalizuje zmienną warunkową. Przyjmuje dwa wskaźniki jako argumenty wywołania. Pierwszy jest wskaźnikiem na zmienną warunkową, a drugi na strukturę atrybutów zmiennej. Drugi argument jest ignorowany przez system Linux i powinien mieć wartość `NULL`. Ta funkcja, tak jak wszystkie inne związane z obsługą zmiennych warunkowych zwraca zero, jeśli wykona się poprawnie lub wartość różną od zera w przeciwnym przypadku.
Szczegóły: `man pthread_cond_init`.
- Funkcja `pthread_cond_signal()` budzi pojedynczy wątek uśpiony na zmiennej warunkowej, do której wskaźnik jest przekazany tej funkcji.
Szczegóły: `man pthread_cond_signal`.
- Funkcja `pthread_cond_broadcast()` budzi wszystkie wątki uśpione na zmiennej warunkowej, do której wskaźnik jest jej przekazany.
Szczegóły: `man pthread_cond_broadcast`.
- Funkcja `pthread_cond_wait()` umożliwia wątkowi oczekiwanie w uśpieniu na spełnienie warunku. Funkcja przyjmuje dwa argumenty, jeden jest wskaźnikiem do zmiennej warunkowej, drugi do zajętego muteksu.
Szczegóły: `man pthread_cond_wait`.

3.5. Uwagi końcowe

Przedstawiony opis nie wyczerpuje całej listy funkcji związanych z obsługą wątków *threads*. Zainteresowani powinni sięgnąć do innych źródeł. Oprócz biblioteki obsługi wątków *threads* w Linuksie dostępne są też inne biblioteki implementujące wątki, jak np. *pth* (`man pth`, o ile biblioteka jest zainstalowana w systemie).

4. Zadania

1. Stwórz w programie dwa wątki, które wypiszą swój identyfikator i identyfikator procesu.
2. Stwórz dwa wątki w programie. Każdemu z nich przekaz przez parametr funkcji dwie liczby. Pierwszy wątek niech policzy sumę tych liczb, drugi różnicę. Obie wartości należy wypisać na ekran w wątkach.
3. Zmodyfikuj program opisany wyżej tak, aby wątki zwracały jak rezultaty swojego działania wyliczone wyniki do wątku głównego, który będzie wypisywał je na ekran.
4. Napisz program, w którym stworzysz jeden wątek łączny i jeden wątek rozdzielny oraz zademonstrujesz różnicę w działaniu tych wątków.
5. Zademonstruj działanie funkcji `pthread_kill()` wysyłając do wątku sygnał, dla którego będzie on miał własną procedurę obsługi. Do podmiany procedury obsługi wykorzystaj funkcję `sigaction()`.
6. Zademonstruj działanie funkcji `sigwait()`.
7. Zademonstruj działanie funkcji sprzątających.
8. Napisz program, w którym stworzysz 20 wątków wykonujących tę samą czynność. W momencie kiedy jeden z nich ją zakończy pozostałe powinny być anulowane w sposób asynchroniczny.
9. Napisz program, który zademonstruje różnicę między anulowaniem asynchronicznym i synchronicznym wątku. **Uwaga:** funkcje `write()`, `printf()` i `puts()` są punktami anulowania!
10. Napisz program, który zademonstruje działanie włączania i wyłączania anulowania wątków.
11. Napisz program z dwoma wątkami i zademonstruj w nim użycie zmiennych prywatnych.
12. Napisz program z dwoma wątkami, z których każdy posiada swoją zmienną prywatną. Klucze do tych zmiennych zapisz w zmiennych globalnych. Sprawdź co się stanie, jeśli po wątki „zamienią” się kluczami do zmiennych prywatnych.

13. Poszukaj innych funkcji do zarządzania atrybutami wątków niż te, które zostały opisane w instrukcji. Napisz program, który zademonstruje ich działanie.
14. Rozwiąż problem producenta i konsumenta za pomocą semafora.
15. Rozwiąż problem producenta i konsumenta za pomocą mutexa.
16. W programowaniu współbieżnym wykorzystywana jest czasem architektura procesów lub wątków, która określana jest mianem *farmer-worker*. Napisz program w oparciu o tę architekturę, który będzie sprawdzał, które z liczb naturalnych, mniejszych od 32 są liczbami pierwszymi. Wątków-robotników powinno być pięciu, a każdemu z nich będzie przyporządkowana jedna z następujących liczb pierwszych: 2,3,5,7,11. Wątek-farmer będzie przydzielał każdemu z nich tę samą liczbę, dla której będą oni wyznaczać resztę z dzielenia przez ich liczbę pierwszą. Farmer po zakończeniu badania liczby przez wszystkich robotników, na podstawie wyników ich pracy powinien orzec, czy dana liczba jest pierwsza, czy też nie. Oczekiwanie robotników na liczbę do sprawdzenia zrealizuj za pomocą zmiennych warunkowych, a oczekiwanie farmera na wyniki za pomocą semafora `System V`.