Systemy operacyjne

Wykład 4: Wątki

Motywacja: prostszy (?) model programowania

Model	Characteristics
Threads	Parallelism, blocking system calls
Single-threaded process	No parallelism, blocking system calls
Finite-state machine	Parallelism, nonblocking system calls, interrupts

Robimy serwer plików – chcemy realizować wiele połączeń.

- **Jednowątkowy proces** Tylko jedno połączenie :(
- **Skończone automaty** Trzeba wyrazić wszystkie stany i procedury przejść. Nieliniowa struktura programu. Centralne zarządzanie stanem automatów. Błąd w procedurze?
- **Wątki** Programy liniowe. Stan połączenia wyrażony przez instrukcje sterowania. Błąd nie musi się propagować!

Wątki są łatwiejsze w użyciu… ale tylko na początku!

Motywacja: wydajność

- **Współpracujące procesy** Współdzielą duże regiony pamięci? Często się komunikują? Są ściśle powiązane? Płacimy za koszt tworzenia / niszczenia procesu, zmiany kontekstu, komunikację i synchronizację...
- **Maszyna wieloprocesorowa** Jak w jednym procesie zrobić użytek z wielu procesorów? Rozdzielić obliczenia między wątki!
- **Operacje asynchroniczne** Połączenie z wieloma serwerami? Jeśli nie musimy szeregowo, to równoległe przetwarzanie będzie szybsze (**fork-join**). Inny wariant? Jeden wątek ładuje dane z dysku, drugi przetwarza, trzeci zapisuje (**pipeline**).

Każda z opcji zamienia jeden zbiór problemów na drugi!

Motywacja: responsywność

- **procesor tekstu** obciążające zadania (formatowanie książki) wykonujemy w tle, pozwalając użytkownikowi edytować stronę
- **program graficzny** można zmieniać parametry filtru obrazu nie czekając na zakończenie generowania podglądu
- **przeglądarka WWW** przewijanie strony zanim doczytają się wszystkie obrazki

W przypadku interfejsów użytkownika normalnie korzysta się z szeregowego przetwarzania zdarzeń (reakcja na naciśnięcie przycisku). Wątki są użytecznym dodatkiem.

Czy wątki to dobre narzędzie?

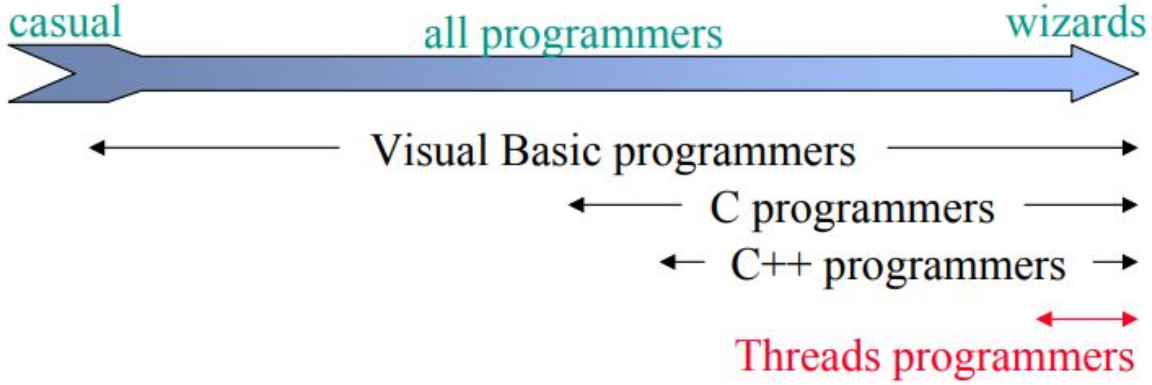
Although threads seem to be a small step from sequential computation, in fact, they represent a huge step. They discard the most essential and appealing properties of sequential computation: understandability, predictability, and determinism. Threads, as a model of computation, are wildly nondeterministic, and the job of the programmer becomes one of pruning that nondeterminism.

The Problem with Threads

Edward A. Lee; University of California at Berkeley

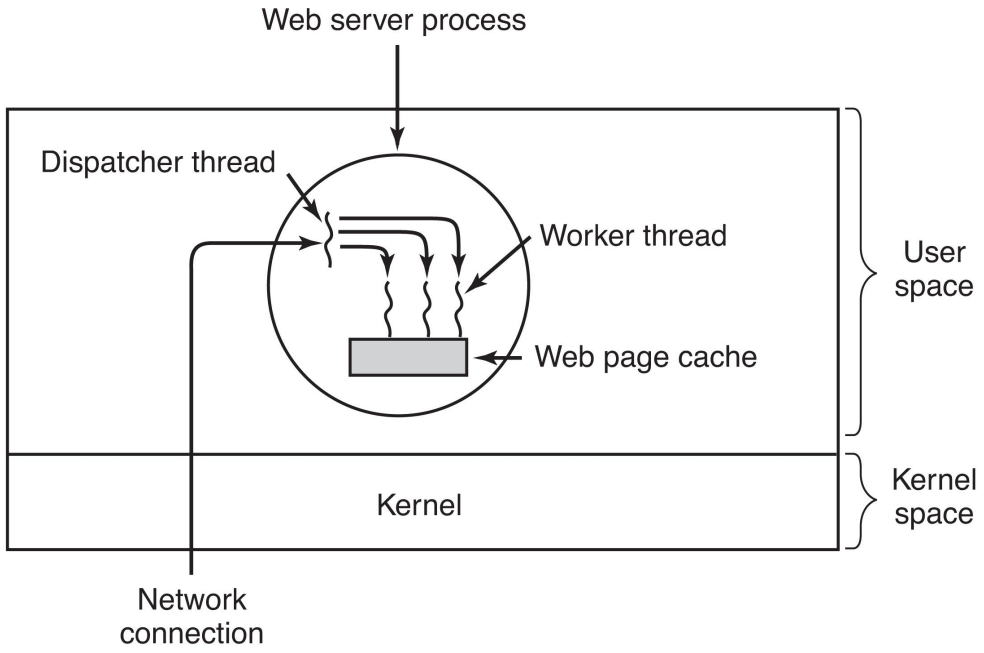
Sytuacja jest dynamiczna, wciąż powstają nowe narzędzia i techniki, ale problem nadal koncentruje się wokół niedeterminizmu.

Programowanie wielowątkowe: wyzwania



- Identyfikacja zadań i podział kodu
- Równomierne rozłożenie zadań między wątki
- Podział danych między zadania
- Zależności między danymi
- Komunikacja i synchronizacja
- Testowanie i odpluskwianie

Przykład: serwer HTTP (thread pool)




```
while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}
```

```
while (TRUE) {
    wait_for_work(&buf)
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}
```

Porównanie zasobów procesu i wątku

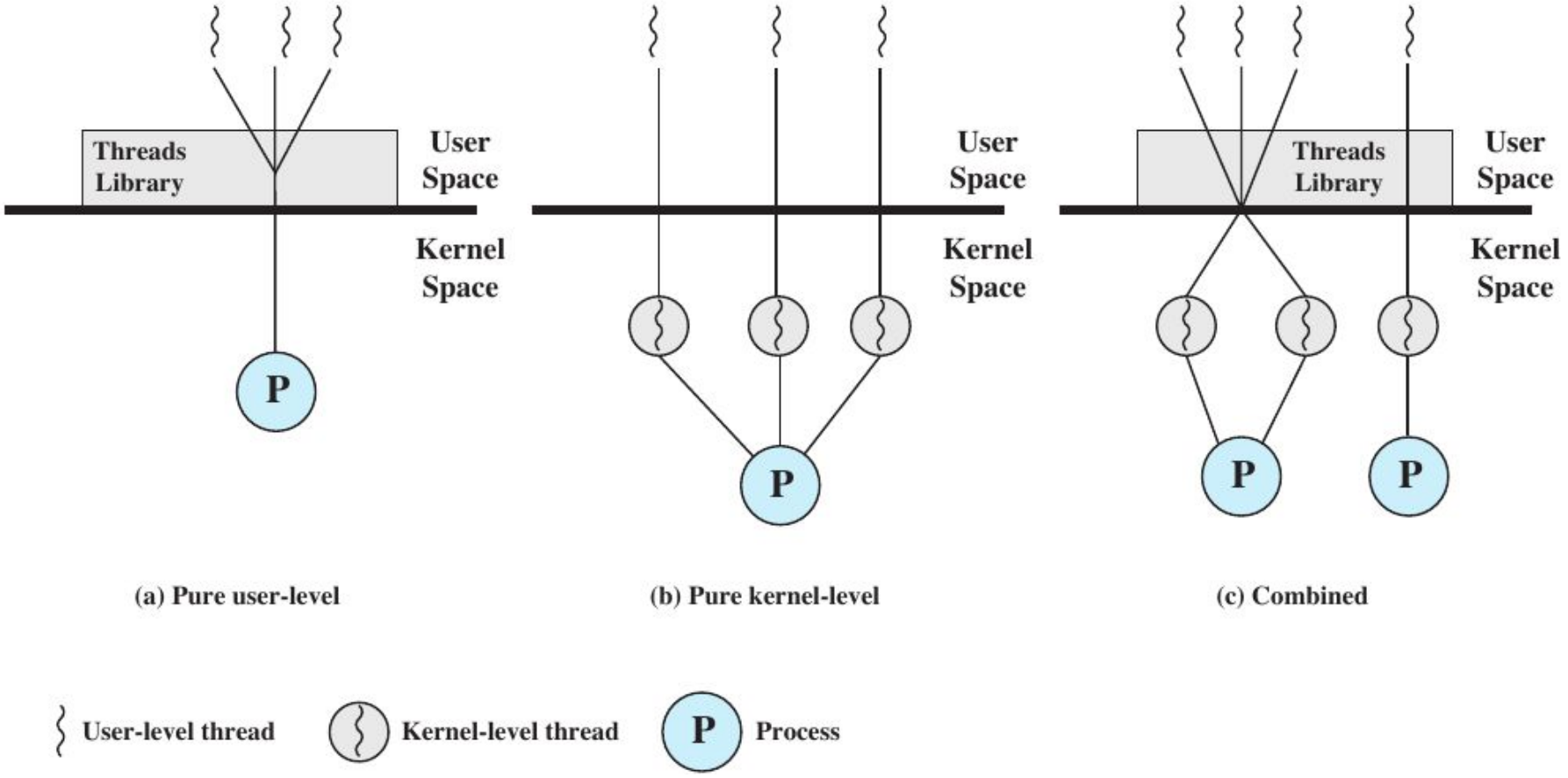
Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Koszt przełączenia wątku jest mały w porównaniu do zmiany kontekstu. Zmiana przestrzeni adresowej powoduje, że:

- TLB zawiera obce wpisy tabeli stron,
- pamięć podręczna zawiera obce bloki pamięci,
- predyktor skoków ma wiedzę o instrukcjach, które zniknęły.

Koszt przełączenia wątku zależy od ich implementacji (KLT / ULT).

Modele wątków: ULT, KLT, hybrydowe



User-Level Threads (model N:1)

Czasami zwane też włóknami (ang. fiber), zielonymi wątkami (ang. green threads) lub współprogramami (ang. coroutines).

+ przełączanie wątków nie wymaga przejścia do jądra
+ aplikacja wie jak efektywnie planować wykonanie wątków
+ wielowątkowość kooperacyjna (mniej problemów z synchronizacją)
+ niezależne od systemu operacyjnego
− większość wywołań systemowych blokuje
− jeden ULT potrafi wstrzymać wykonanie pozostałych w procesie
− błąd strony blokuje wszystkie wątki
− trzeba zaprogramować środowisko wykonawcze (ang. *runtime library*)

Biblioteka ULT zawiera opakowanie (ang. *wrapper*) blokujących wywołań systemowych oraz zarządzanie i przełączanie wątków.

Aktywacje planisty

Biblioteka ULT udostępnia środki synchronizacji i wątki blokują się w przestrzeni użytkownika. Potrzebna obsługa akcji blokujących w jądrze.

Aplikacja zgłasza jądru liczbę wirtualnych procesorów, których chce używać. Kiedy wirtualny procesor się blokuje aplikacja dostaje wezwanie (ang. *upcall*) – coś jak obsługa sygnału. Wezwanie wzywa procedurę planowania środowiska uruchomieniowego i informuje o zmianie stanu wirtualnych procesorów. Tablica zdarzeń typów:

```
SA_UPCALL_(NEWPROC | PREEMPTED | BLOCKED | UNBLOCKED | SIGNAL)
```

… i do tego zapisany kontekst przerwanego wątku.

An Implementation of Scheduler Activations on the NetBSD Operating System

Interfejs `pthreads`

`pthread_create`	`fork`	podajemy procedurę do uruchomienia w wątku
`pthread_exit`	`exit`	domyślnie wołane po powrocie z wątku
`pthread_join`	`waitpid`	pobiera kod wyjścia (ale nie kody błędów)
`pthread_cleanup_push`	`atexit`	rejestracja procedur do wykonania przed wyjściem
`pthread_self`	`getpid`	identyfikator wątku
`pthread_cancel`	`kill` (?)	wysłanie prośby o zakończenie do wątku

Atrybuty wątków używane w trakcie ich tworzenia: stos, stan separowania (ang. *detached*), parametrów planisty i przypisanie do zbioru procesorów.

Punkty przerwań to ustalone miejsca, gdzie wykonywanie wątku może zostać przerwane.

Problemy z wątkami KLT...

Nie możemy usunąć z naszych programów wszystkich zmiennych globalnych. Co jeśli jakaś jest nieświadomie współdzielona?

W każdym wątku potrzebujemy prywatnej kopii `errno`, żeby odczytać kod błędu ostatniego wywołania systemowego!

```
/* Function to get address of global `errno' variable. */
extern int *__errno_location (void);
/* When using threads, errno is a per-thread value. */
#define errno (*__errno_location ())
```

MT-Safe vs. MT-Unsafe functions

Czy funkcja biblioteczna przechowuje stan w zmiennej globalnej?

```
long int random(void);
void srandom(unsigned int seed);
```

This function should not be used in cases where multiple threads use random() and the behavior should be reproducible. Use random_r(3) for that purpose.

A może zwraca wskaźnik do statycznie przydzielonego bufora?

```
char *strsignal(int sig);
```

Interface	Attribute	Value
strsignal()	Thread safety	MT-Unsafe race:strsignal locale

Thread Local Storage

Prywatny globalny licznik per wątek programu (GCC):

```
__thread int counter = 0;
```

Skąd wątek wie gdzie w pamięci są jego prywatne zmienne?
Zależne od ABI! Na **`x86-64`** w rejestrze segmentowym **`%fs`**.

Program zyskuje dodatkowe sekcje: **`.tdata`** oraz **`.tbss`**.
Gdy tworzymy nowy wątek ktoś musi utworzyć kopie tych sekcji!
A co jeśli posiadamy biblioteki współdzielone z sekcjami TLS?

Biblioteka standardowa musi współpracować z dynamicznym konsolidatorem! ELF Handling For Thread-Local Storage

Wątki i systemy uniksowe

Sygnały Z synchronicznymi jest prosto (`SIGSEGV`, `SIGFPE`) bo idą bezpośrednio do wątku. Co się stanie jeśli do procesu przyjdzie sygnał asynchroniczny (`SIGINT`, `SIGHUP`)? Który go obsłuży?

Fork Co się dzieje z pozostałymi wątkami przy klonowaniu? Tylko aktywny przechodzi! Lądujemy w nowej przestrzeni adresowej z założonymi blokadami, których nie ma kto zwolnić!
Można próbować z `pthread_atfork`...

I/O Co jeśli dwa wątki czytają z tego samego pliku? W jakiej kolejności wykonują się operacje? Używać `pread` i `pwrite`!

Wątki w systemach uniksowych są ciałem obcym!

Pytania?