Systemy operacyjne

Wykład 14
Synchronizacja i wątki

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

Ile blokować? Ziarnistość blokad

Rywalizacja o blokady (ang. *lock contention*) powstaje kiedy zadanie oczekuje na zwolnienie (ang. *release*) blokady założonej (ang. *acquire*) przez inne zadania.

Narzut wydajnościowy (ang. *lock overhead*) to czas jaki zadanie spędza na wykonywanie akcji założenia lub zwolnienia blokady.

Ziarnistość (ang. *granularity*) określa ilość chronionych danych. Inaczej → jak długo zadanie wykonuje się z założoną blokadą?

Ziarnistość blokad duża (ang. *coarse-grained*) → sumarycznie niski narzut, ale wysoka rywalizacja. Ziarnistość mała (ang. *fine-grained*) → sumarycznie duży narzut, ale niska rywalizacja.

Dodatkowe problemy z blokadami

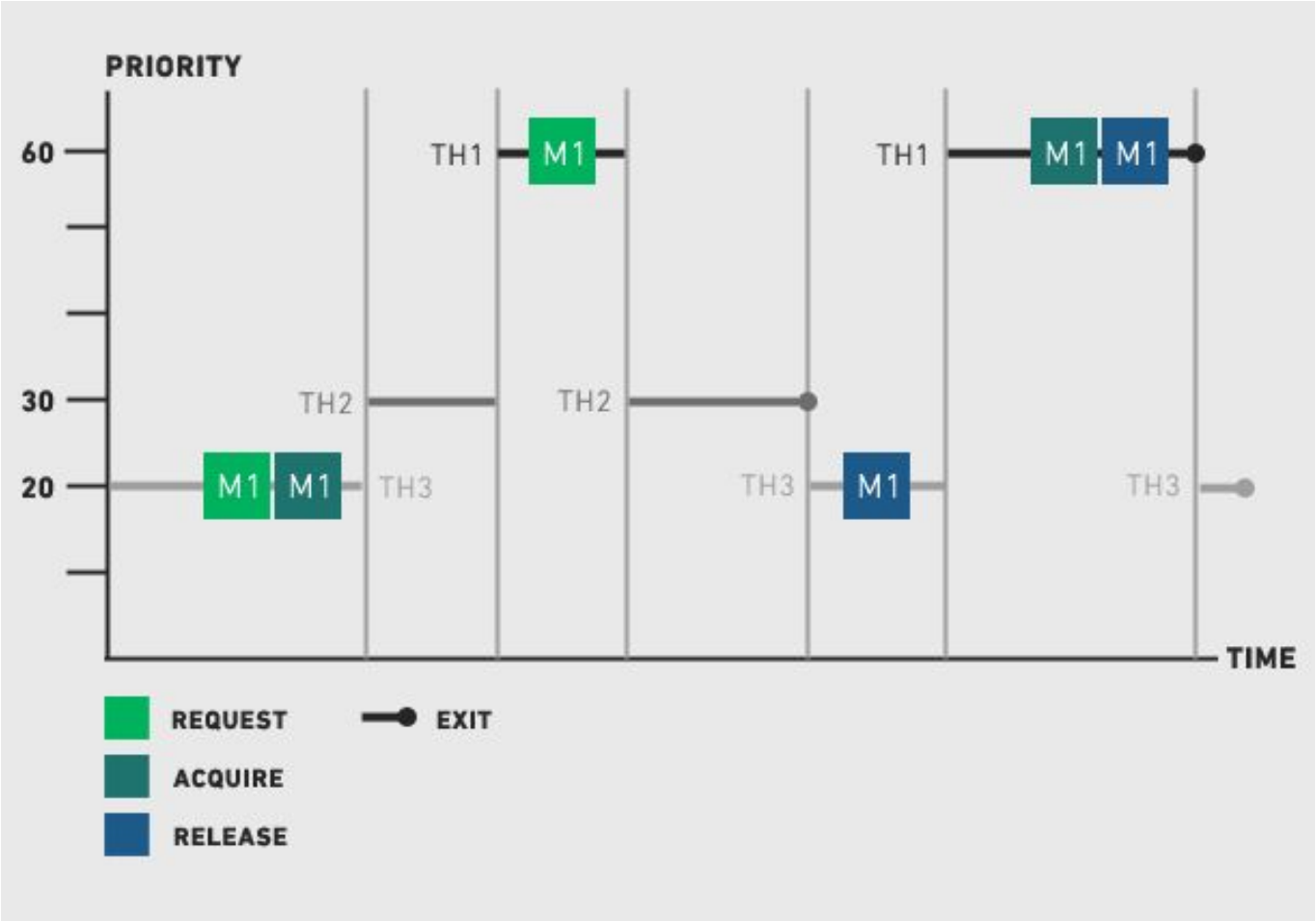
Odpluskwianie Błędy zależne od przeplotu wykonania instrukcji! Debugger może przypadkiem usuwać przeploty kończące się błędem.

Konwojowanie (ang. *convoying*) Oczekiwanie na blokadę, którą zwolni zadanie wywłaszczone bądź czekające na obsługę błędu strony.

Wrażliwość na zmiany architektury aplikacji. Zmienia się sposób przetwarzania danych → trzeba ponownie przemyśleć ziarnistość blokad!

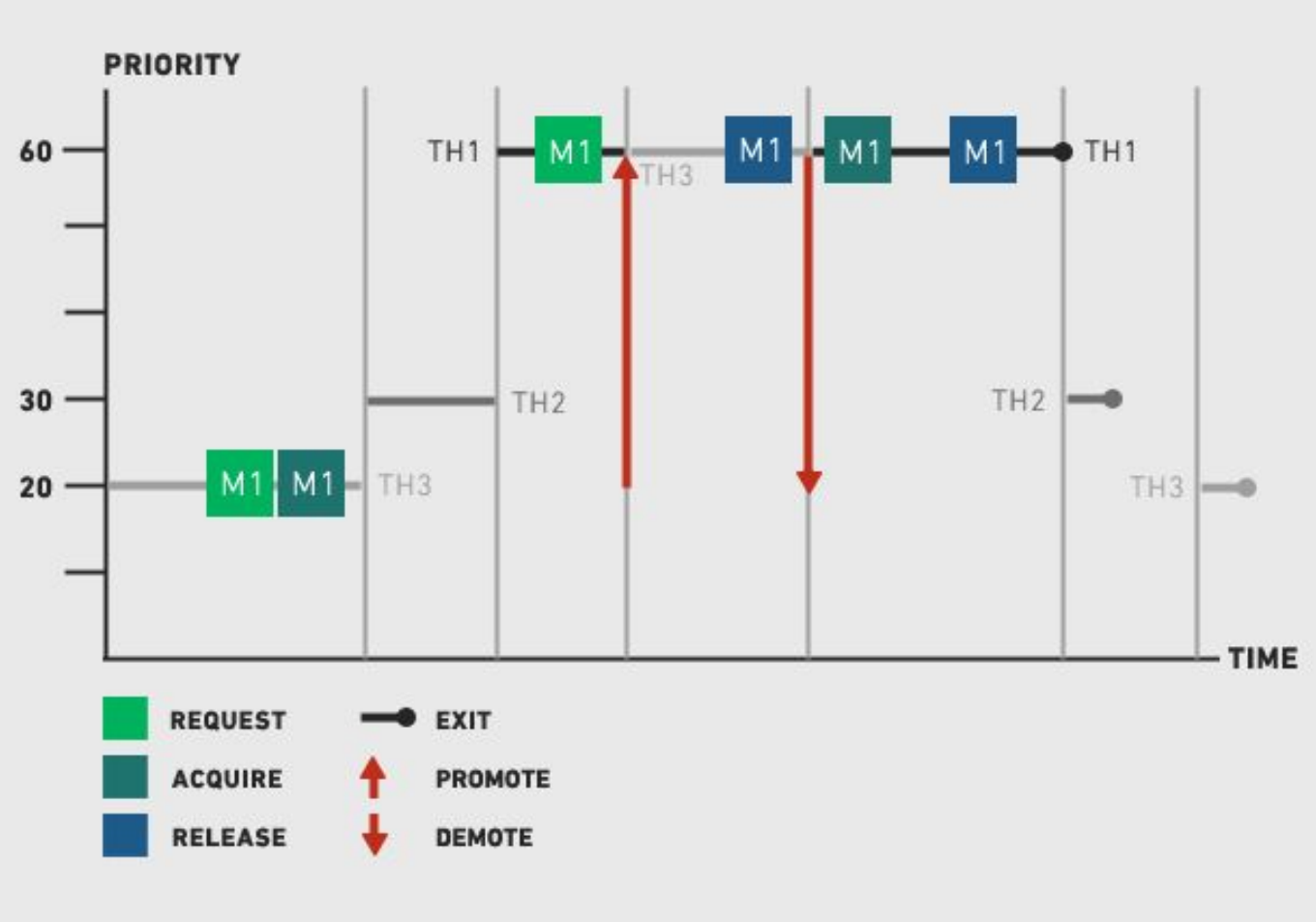
Składanie (ang. *composability*) procedur zakładających blokady wymaga wiedzy o tym jak ich używają. Inaczej możliwe zakleszczenia!

Nieograniczone czasowo odwrócenie priorytetów



TH2 wykonuje się gdy TH3 jest w sekcji krytycznej i blokuje TH1!

Rozwiązanie: dziedziczenie priorytetów



TH3 dziedziczy priorytet po TH1 na czas wykonania sekcji krytycznej!

Im mniej blokad tym lepiej

Alternatywne rozwiązania:

- **pamięć transakcyjna** Tworzymy transakcję, która może zawieść. Odczytujemy zbiór komórek pamięci S, wykonujemy obliczenia i wdrażamy zmiany pod warunkiem, że nikt nie zmienił S. W przeciwnym wypadku musimy ponowić transakcję!
- **struktury danych bez blokad** (ang. *lock-free data structures*) Wykorzystanie operacji atomowych wbudowanych w procesor do realizacji prostych struktur danych: stos, kolejka, zbiór, …
- **trwałe struktury danych** (ang. persistent data structure) Operacje modyfikacji struktury danych tworzą jej nowe wersje współdzieląc pamięć z poprzednimi wersjami.

Synchronizacja wątków POSIX.1

Mutex (mutual exclusion)

Służą głównie do synchronizacji wątków. Mają dwa stany **zablokowany** (ang. *locked*) i **odblokowany** (ang. *unlocked*).

Każdy muteks ma **właściciela** → wątek który go zablokował. Tylko właściciel może odblokować muteks, w p.p. błąd lub zachowanie niezdefiniowane.

Muteksy mogą być **rekursywne**, tj. zliczają ile razy zostały wzięte.

Z reguły nie są dostępne dla procesów chyba, że przez pamięć dzieloną (*POSIX.1*) lub obiekty nazwane (*WinNT*).

POSIX.1: Muteksy

pthread_mutex_init	inicjalizuje strukturę muteksa podanymi atrybutami
pthread_mutex_destroy	zmienia strukturę muteksa, tak by kolejne operacje zawiodły
pthread_mutex_lock	zakłada blokadę lub zostaje uśpiony, ew. **EDEADLK**
pthread_mutex_timedlock	j.w. ale po upłynięciu terminu zwraca **ETIMEDOUT**
pthread_mutex_unlock	zwalnia blokadę lub zmniejsza licznik, ew. **EPERM**
pthread_mutex_trylock	zakłada blokadę lub zwraca **EBUSY**
pthread_mutexattr_settype	ustawia typ muteksa na **RECURSIVE** lub **ERRORCHECK**
pthread_mutexattr_setrobust	ustawia tryb **ROBUST**, jeśli wątek umrze **EOWNERDEAD**

Dokumentacja w pakietach **manpages-posix-dev**.

W trakcie kompilacji należy użyć opcji konsolidatora **-lpthread**

POSIX.1: zachowanie muteksów

Mutex Type	Robustness	Relock	Unlock When Not Owner
NORMAL	non-robust	deadlock	undefined behavior
NORMAL	robust	deadlock	error returned
ERRORCHECK	either	error returned	error returned
RECURSIVE	either	recursive	error returned
DEFAULT	non-robust	undefined behavior	undefined behavior
DEFAULT	robust	undefined behavior	error returned

W implementacji Linuksowej można zawsze zwolnić muteks typu **DEFAULT** lub **NORMAL**, ale formalnie to **błąd programisty**!

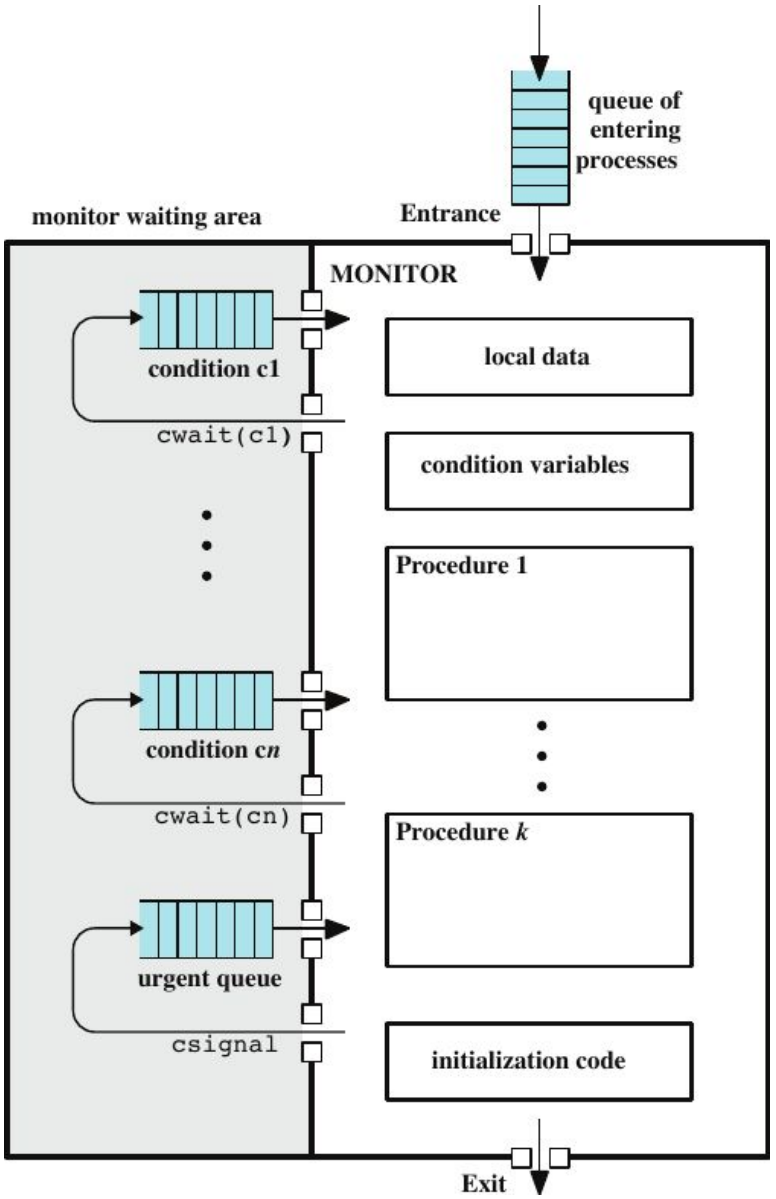
Muteksami można synchronizować procesy pod warunkiem, że reprezentacja muteksów została przydzielona w pamięci współdzielonej → `pthread_mutexattr_setpshared`.

Monitory

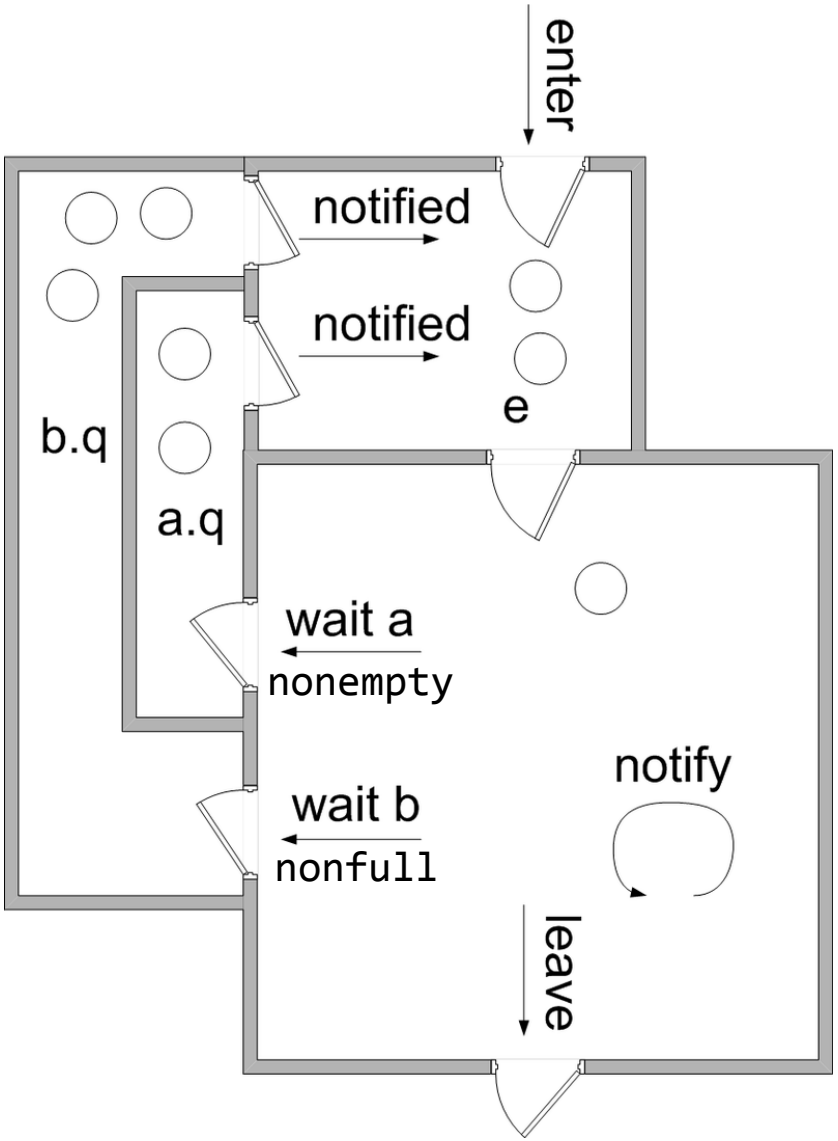
Monitor narzędzie języka programowania lub wzorzec projektowy. Zawiera metody, zmienne warunkowe i zmienne lokalne. Tylko jeden wątek może modyfikować stan wewnętrzny monitora.

Zmienna warunkowa to kolejka uśpionych wątków.

Monitor to w pewnym sensie *“zsynchronizowana klasa”*.



Monitor Mesa: problem producent-konsument




```
monitor ProducerConsumer:
  nonempty, nonfull: CondVar
  queue: Queue<T>

  fn put(T item) -> unit:
    while queue.full():
      nonfull.wait()
    queue.push(item)
    nonempty.notify()

  fn get() -> T:
    while queue.empty():
      nonempty.wait()
    T item = queue.pop()
    nonfull.notify()
    return item
```

Zmienne warunkowe

Kodują zdarzenie spełnienia określonego warunku. Sprawdzenie predykatu nie jest elementem tego narzędzia. Zmiennych warunkowych używa się w sekcji krytycznej używającej muteksa.

Kiedy warunek nie jest spełniony wątek woła **`wait`**. Następnie w jednym kroku (atomowo) wychodzi z sekcji krytycznej i zostaje uśpiony na kolejce wątków oczekujących na spełnienie warunku.

Jeśli wątek przebywający w sekcji krytycznej swym działaniem spełnił warunek, to woła **`signal`** lub **`broadcast`**. To wybudza wątki oczekujące, które natychmiast blokują się na ponownym wejściu do sekcji krytycznej (`mutex`)!

POSIX.1: Zmienne warunkowe

`pthread_cond_init`	inicjalizuje zmienną warunkową podanymi atrybutami
`pthread_cond_destroy`	zabrania kolejnych operacji na zmiennej warunkowej lub **`EBUSY`**
`pthread_cond_wait`	oczekuje na wybudzenie
`pthread_cond_timedwait`	j.w. ale po upłynięciu terminu zwraca **`ETIMEDOUT`**
`pthread_cond_signal`	wybudza dokładnie jeden oczekujący wątek
`pthread_cond_broadcast`	wybudza wszystkie oczekujące wątki
`pthread_condattr_setpshared`	dzielenie zmiennej warunkowej między procesy

W przypadku **`pthread_cond_timedwait`** nieścisłość w specyfikacji!

Podręcznik Linuksa mówi, że może zwrócić **`EINTR`** jeśli w międzyczasie obsłużono sygnał. Specyfikacja POSIX.1 zabrania takiego zachowania.

Szkic implementacji zmiennych warunkowych

```
def pthread_cond_wait(condvar c, mutex m):
  atomic:
    m.unlock()
    c.pushThread(self)
    block
  m.lock()

def pthread_cond_signal(condvar c):
  atomic:
    wakeup(c.popThread())

def pthread_cond_broadcast(condvar c):
  atomic:
    while t = c.popThread():
      wakeup(t)
```

Producent-konsument: wątki POSIX (1)

```
 1 pthread_mutex_t critsec;
 2 pthread_cond_t non_empty, non_full;
 3 queue_t q;
 4
 5 void *producer(void *ptr) {
 6   for (int i = 1; i <= NITEMS; i++) {
 7     int item = produce();
 8     pthread_mutex_lock(&critsec);
 9     while (queue_full(&q))
10       pthread_cond_wait(&non_full, &critsec);
11     queue_push(&q, item);
12     pthread_cond_signal(&non_empty);
13     pthread_mutex_unlock(&critsec);
14   }
15 }
```

Producent-konsument: wątki POSIX (2)

```
 1 void *consumer(void *ptr) {
 2   for (int i = 1; i <= NITEMS; i++) {
 3     pthread_mutex_lock(&critsec);
 4     while (queue_empty(&q))
 5       pthread_cond_wait(&non_empty, &critsec);
 6     int item = queue_pop(&q);
 7     pthread_cond_signal(&non_full);
 8     pthread_mutex_unlock(&critsec);
 9     consume(item);
10   }
11 }
```

Producent-konsument: wątki POSIX (3)

```
 1 int main(int argc, char **argv) {
 2   pthread_t pro, con;
 3   queue_init(&q, 100);
 4   pthread_mutex_init(&mutex, 0);
 5   pthread_cond_init(&non_full, 0);
 6   pthread_cond_init(&non_empty, 0);
 7   pthread_create(&con, 0, consumer, 0);
 8   pthread_create(&pro, 0, producer, 0);
 9   pthread_join(pro, 0);
10   pthread_join(con, 0);
11   pthread_cond_destroy(&non_full);
12   pthread_cond_destroy(&non_empty);
13   pthread_mutex_destroy(&mutex);
14   queue_destroy(&q);
15 }
```

Blokady współdzielone

Znane jako ***reader-write lock***, albo ***shared-exclusive lock***.

Synchronizacja dostępu do struktury danych → w jednej chwili #R ⩾ 0 wątków może ją czytać, albo #W ⩽ 1 wątków może ją modyfikować.

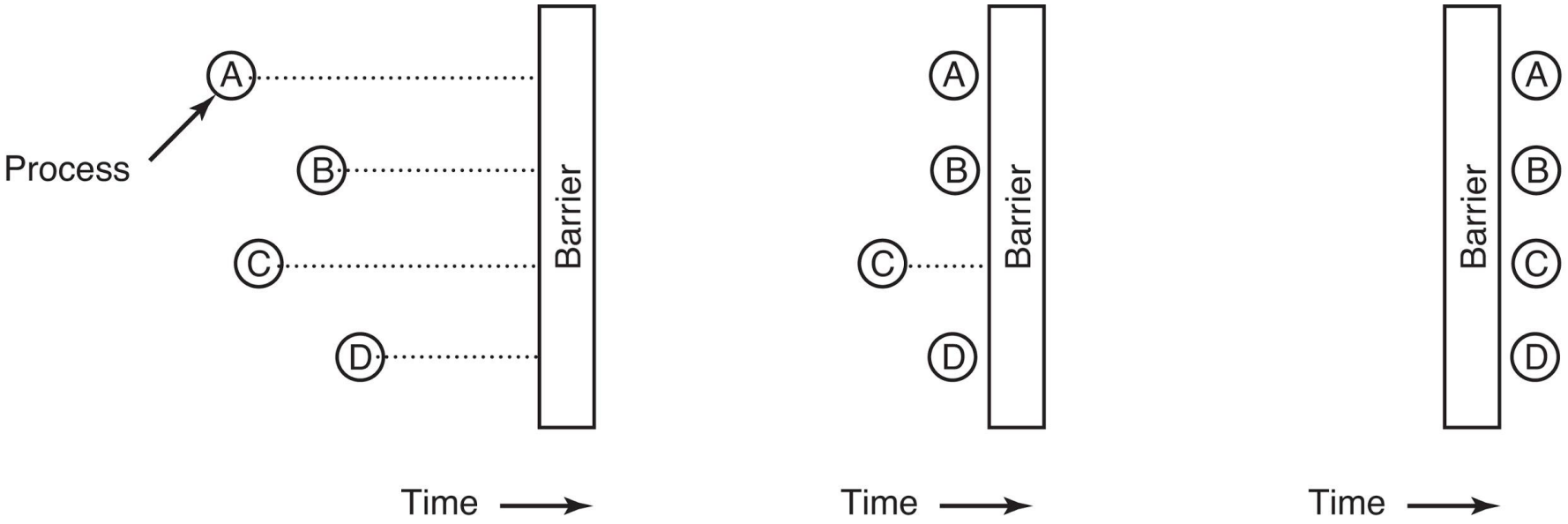
Możemy nadać priorytet czytelnikom (ang. *read-preferring RW-lock*) lub pisarzom (ang. *write-preferring RW-lock*). Kto wyczuwa głodzenie?

Niektóre implementacje blokad współdzielonych dopuszczają operacje:

- zdegradowania (ang. *downgrade*) (W → R),
- awansowania (ang. upgrade) (R → W),
- opróżniania (ang. drain) (#R + #W → 0).

POSIX.1: `pthread_rwlock_rdlock`, `pthread_rwlock_wrlock`.

Bariery synchronizacyjne POSIX.1



Obliczenia postępujące w fazach: symulator procesora potokowego, rendering klatki gry komputerowej. Wszystkie podzadania muszą się zakończyć (`pthread_barrier_wait`) zanim przejdziemy do następnej fazy. Z barierą kojarzymy liczbę zadań (`pthread_barrier_init`).
Po przejściu zadań bariera nadaje się do ponownego użytku.

Unix i wątki

Wątki i sygnały

Sygnały synchroniczny (`SIGSEGV`, `SIGFPE`) spowodowało wykonanie instrukcji wątku, zatem są kierowane do tegoż wątku. **Sygnał asynchroniczny** (`SIGINT`, `SIGHUP`) obsłuży pierwszy wątek, który będzie powracał do przestrzeni użytkownika.

Informacje dot. obsługi sygnału (procedura, `SIG_IGN`, `SIG_DFL`) są współdzielone między wątkami. Każdy wątek ma swoją maskę sygnałów (`pthread_sigmask`) i maskę sygnałów oczekujących (`pending`). Przy rozpatrywaniu sygnałów do dostarczenia wątkowi jądro sumuje `pending` dla procesu i wątku.

(Więcej w APUE §12.8)

Wątki i fork

Sklonowanie procesu z wątkami przed utworzeniem dziecka musiałoby zatrzymać wątki w dobrze zdefiniowanym stanie (przez aplikację!) Jądro nie ma takiej wiedzy!

Wybrana strategia → klonujemy wątek który zawołał `fork`!

Po `fork` dziecko zawiera dokładną kopię przestrzeni adresowej wraz ze stanem wszystkich środków synchronizacyjnych: mutex, semafor, zmienne warunkowe, itd. Kto zwolni te blokady?

Można próbować z `pthread_atfork`…

(Więcej w APUE §12.9)

Wątki i I/O

Co jeśli dwa wątki czytają z tego samego pliku?

1. W jakiej kolejności wykonują się operacje na plikach?
2. Co stanie się z kursorem?

Aplikacja musi rozwiązać problem dostępu do współdzielonych fragmentów pliku samodzielnie albo z użyciem blokad na rekordach → `fcntl` (działa również między procesami).

Jeśli używamy rozłącznych fragmentów pliku to można użyć `pread` i `pwrite`, które nie modyfikują kursora.

(Więcej w APUE §14.3)

Pytania?