Systemy operacyjne

Wykład 5–6:
Współbieżność, synchronizacja, komunikacja

Współbieżność

Po co nam komunikacja?

- **dzielenie informacji** – programy mogą wymieniać się danymi by realizować bogatszy zestaw zadań (potoki, wrappery GUI) lub współtworzyć dane (bazy danych)
- **szybsze przetwarzanie** – podział obliczeń na zadania pozwala wykonywać je z użyciem dodatkowych zasobów sprzętowych (inne rdzenie, GPU, chmura)
- **modularność** – podział dużego monolitycznego programu na podprocesy → system jest elastyczny i bezpieczniejszy (izolacja!), ale trzeba wymieniać dane (umiejętnie!)

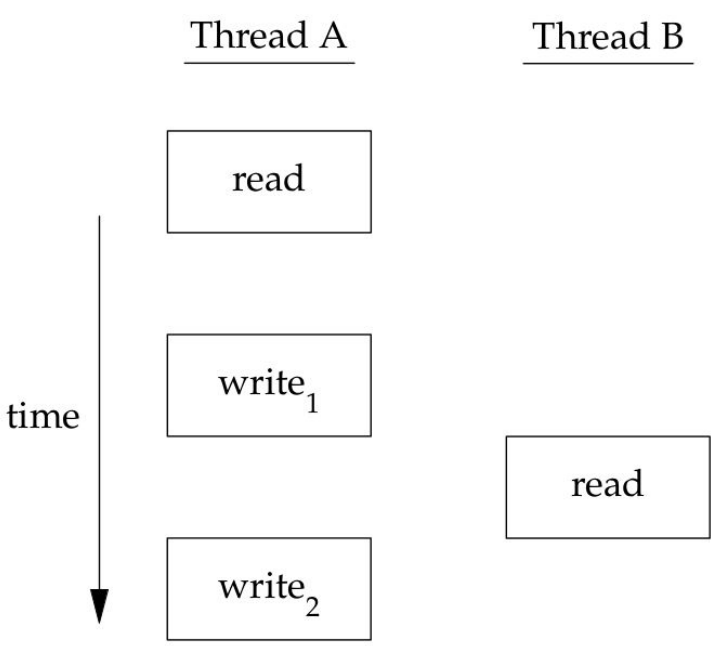
Narzędzia IPC

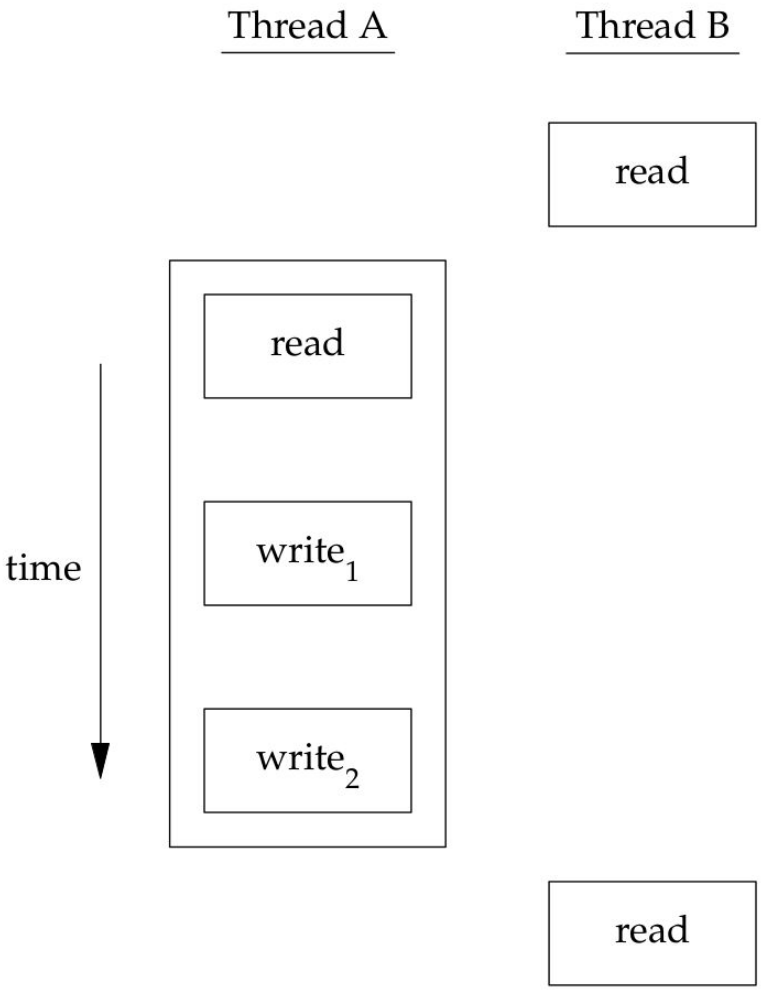
- **pamięć współdzielona** lokalnie, wymaga koordynacji dostępu → inaczej możliwe naruszenie spójności danych
- **wymiana komunikatów** lokalnie i sieciowo, przesyłanie datagramów lub strumieni danych, adresowanie, buforowanie, semantyka operacji **`send`** i **`recv`**

Komunikacja i synchronizacja wiąże się z problemami:

- **sytuacja wyścigu** (ang. *race condition*)
- **zakleszczenie** (ang. *deadlock*)
- **głodzenie** (ang. *starvation*)
- **uwięzienie** (ang. *livelock*)

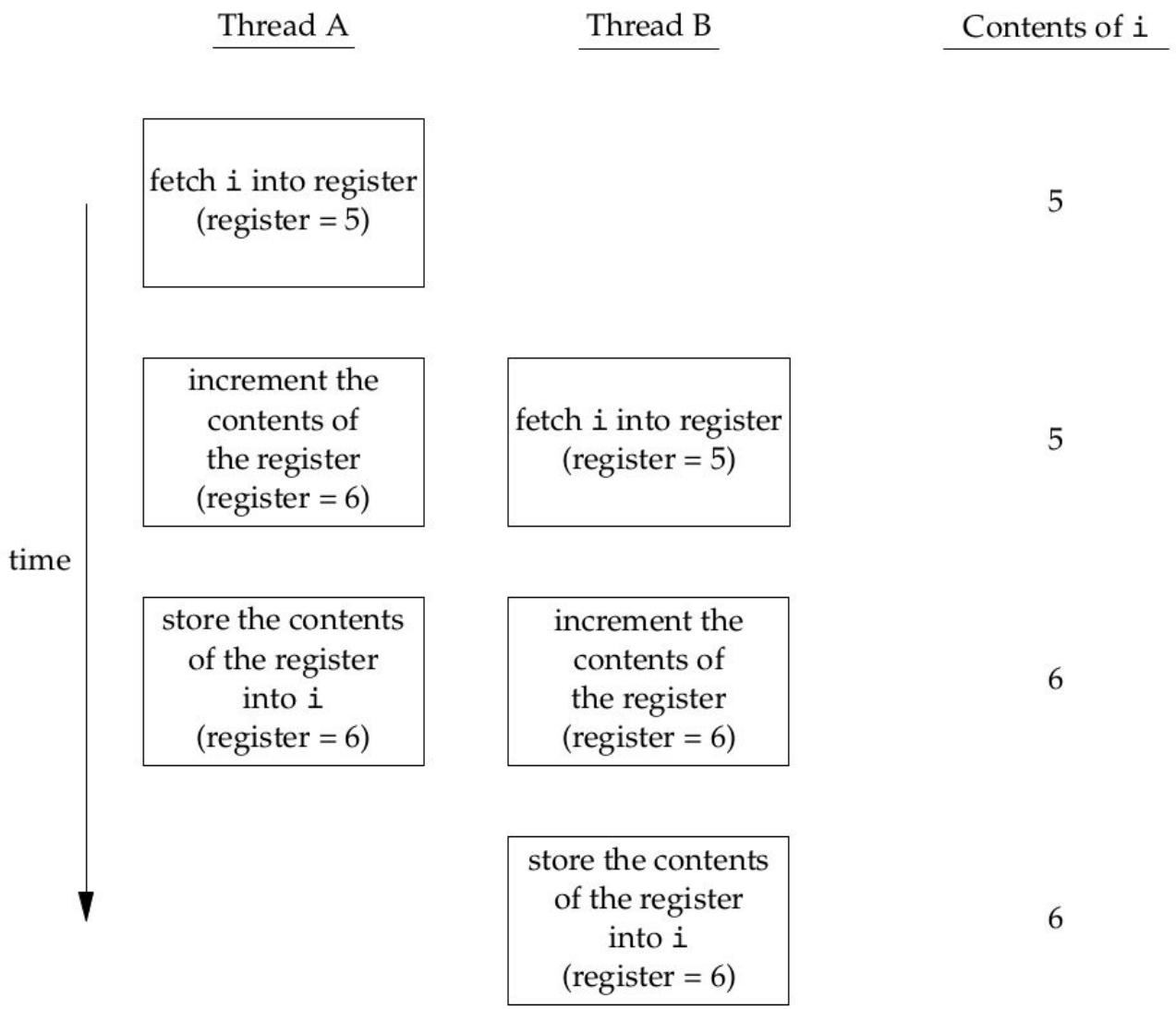
Naruszenie spójności danych



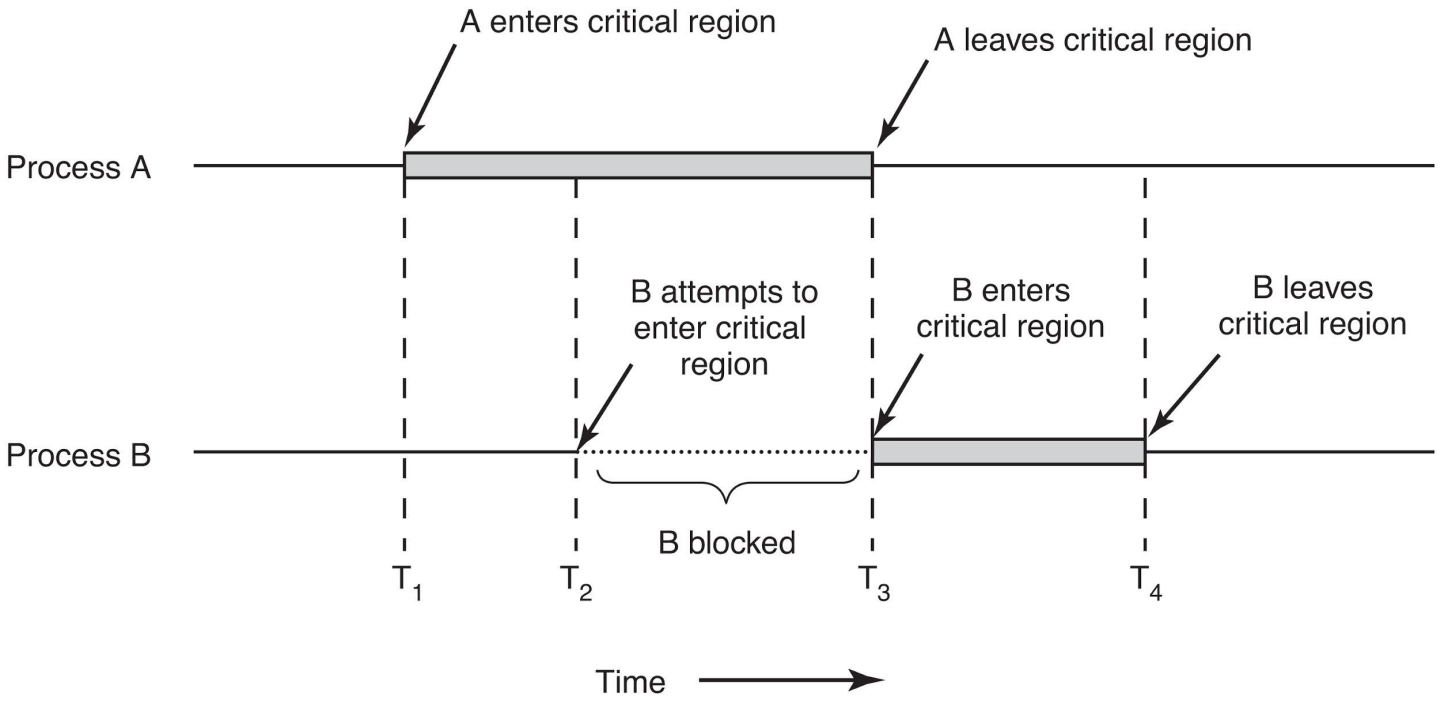


Operacja jest **atomowa**
jeśli obserwator nie może
zobaczyć wyników pośrednich.

Przykład wyścigu



Sekcja krytyczna



Fragment programu korzystający ze współdzielonych zasobów to **sekcja krytyczna**. Potrzebujemy mechanizmu **wzajemnego wykluczania** (ang. *mutual exclusion*) by zapobiec wyścigom!

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

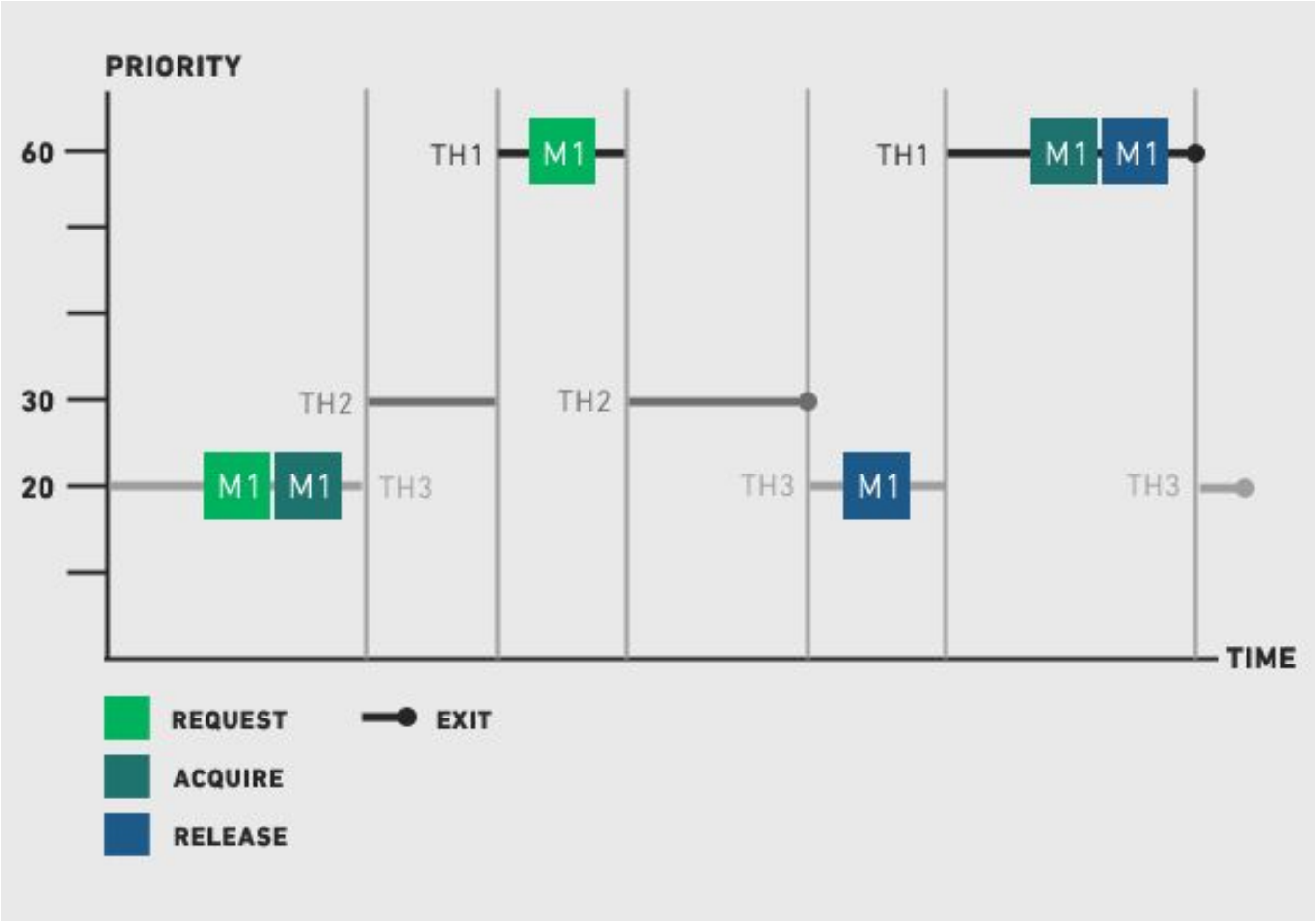
Odplukswianie Błędy zależne od przeplotu wykonania instrukcji! Debugger może przypadkiem usuwać przeploty kończące się błędem.

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!

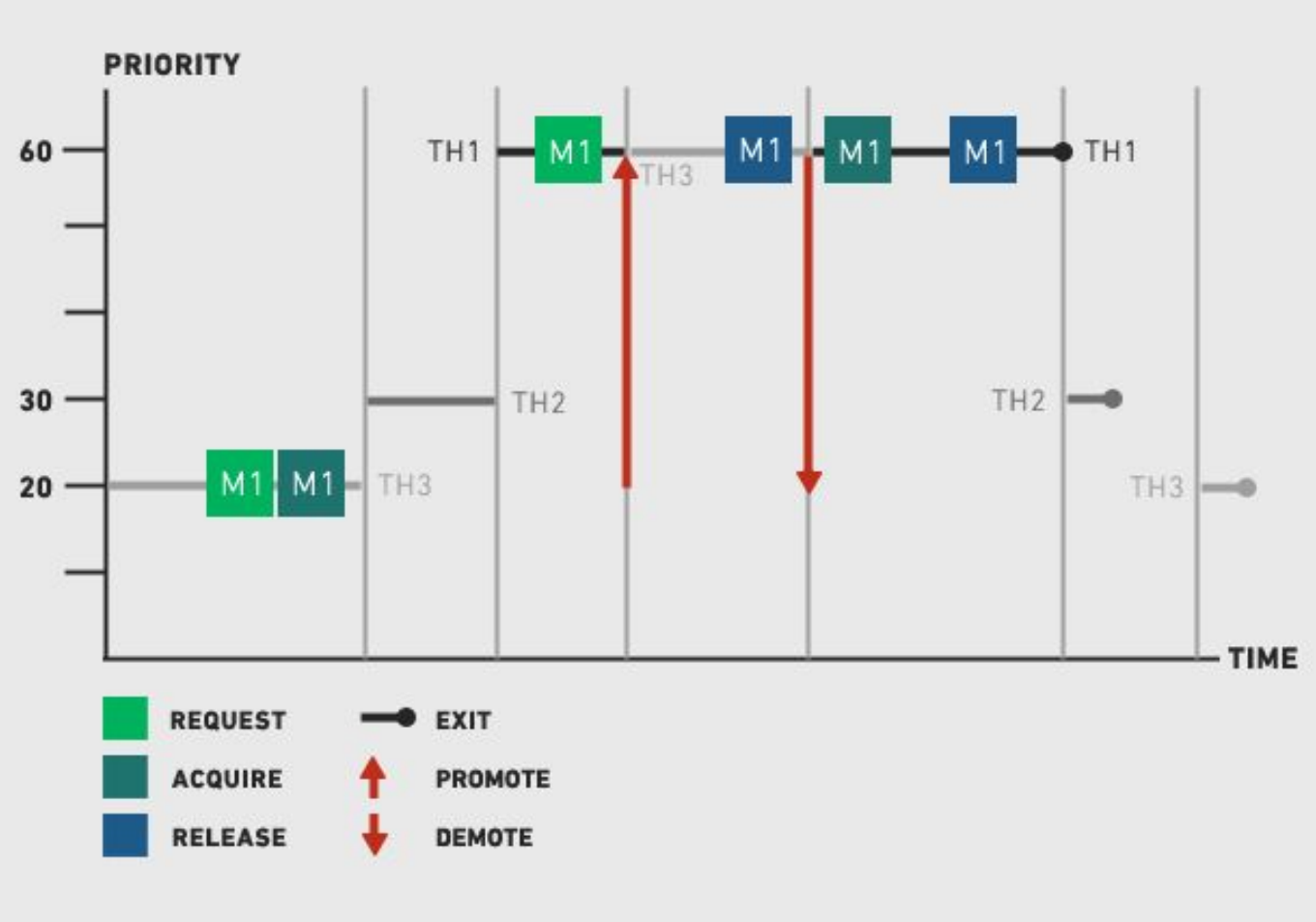
Składanie (ang. *composability*) procedur zakładających blokady wymaga wiedzy o tym jak ich używają. Inaczej możliwe zakleszczenia!

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.

Producent-konsument z błędami

```
queue = Queue(100) # kolejka ograniczonej długości

1  def producer():
2   forever:
3     item = produce()
4     if queue.full():
5       sleep()
6     queue.push(item)
7     if not queue.empty():
8        wakeup(consumer)

9  def consumer():
10  forever:
11    if queue.empty():
12      sleep()
13    item = queue.pop()
14    if not queue.full():
15      wakeup(producer)
16    consume(item)
```

Synchronizacja
w przestrzeni użytkownika

Semafory

Środek synchronizacji, z którym skojarzono kolejkę zadań oczekujących i liczbę całkowitą (wartość semafora).

`sem_wait` Wartość zmiennej większa od 0?
Nie → zadanie zostaje uśpione! Tak → zmniejszamy o 1!

`sem_post` Jest zadanie uśpione i zmienna równa 0?
Tak → wybudzamy wątek! Nie → zwiększamy o 1!

`POSIX.1` udostępnia semafory:

- nazwane (`sem_open`, `sem_close`, `sem_unlink`),
- nienazwane (`sem_init`, `sem_destroy`).

Producent-konsument: semafory (1)

```
queue = Queue(N)
critsec = Semaphore(1) # sekcja krytyczna
empty = Semaphore(N)   # liczba wolnych miejsc
taken = Semaphore(0)   # liczba zajętych miejsc
```

```
  def producer():
1   forever:
2     item = produce()
3     empty.wait()
4     critsec.wait()
5     queue.push(item)
6     critsec.post()
7     taken.post()
```

```
  def consumer():
8   forever:
9     taken.wait()
10    critsec.wait()
11    item = queue.pop()
12    critsec.post()
13    empty.post()
14    consume(item)
```

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`.

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.

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, &mutex);
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, &mutex);
 6     int item = queue_pop(&q);
 7     pthread_cond_signal(&non_full);
 8     pthread_mutex_unlock(&critsec);
 9     consume(item);
10   }
11 }
```

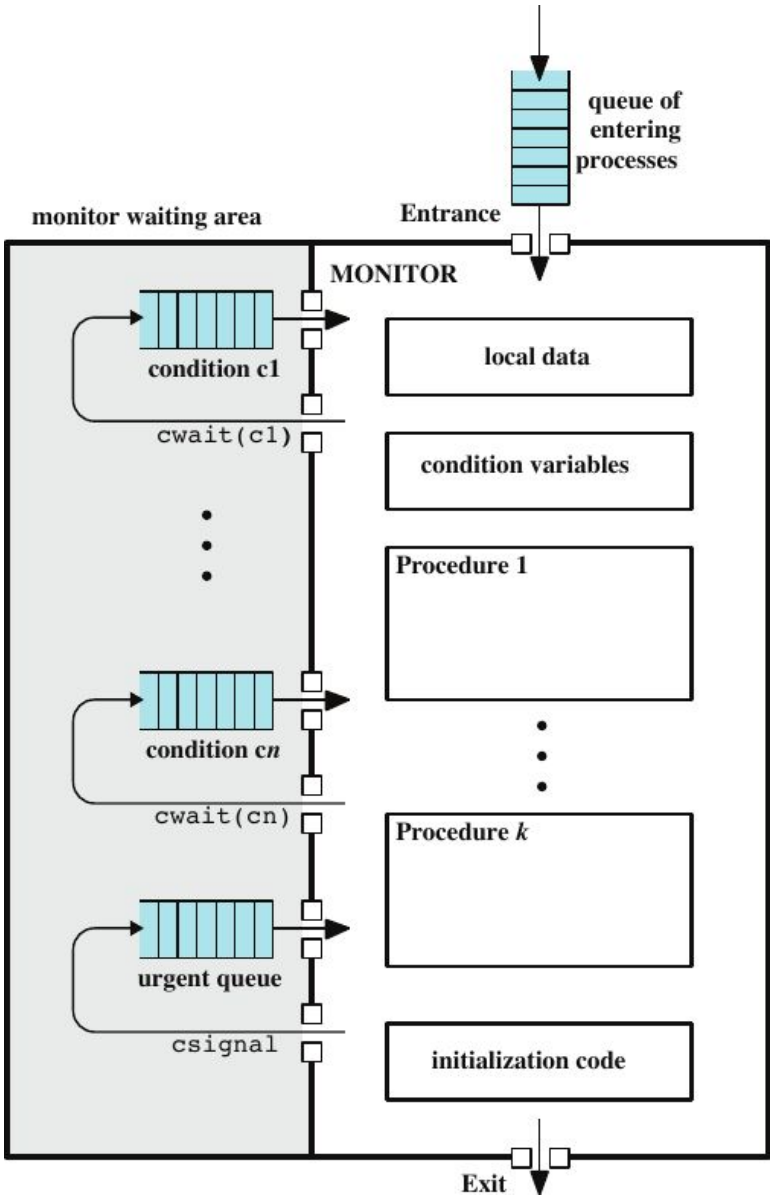
Producent-konsument: wątki POSIX (3)

```
 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 }
```

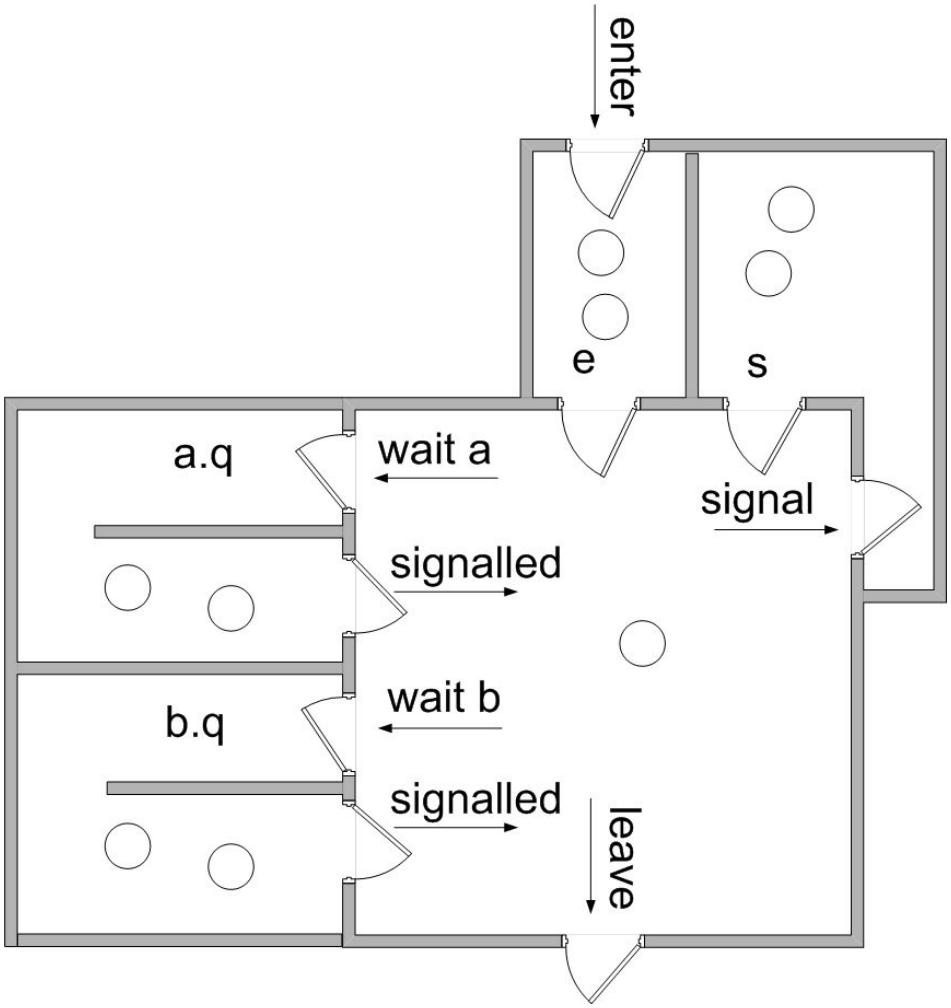
Monitory

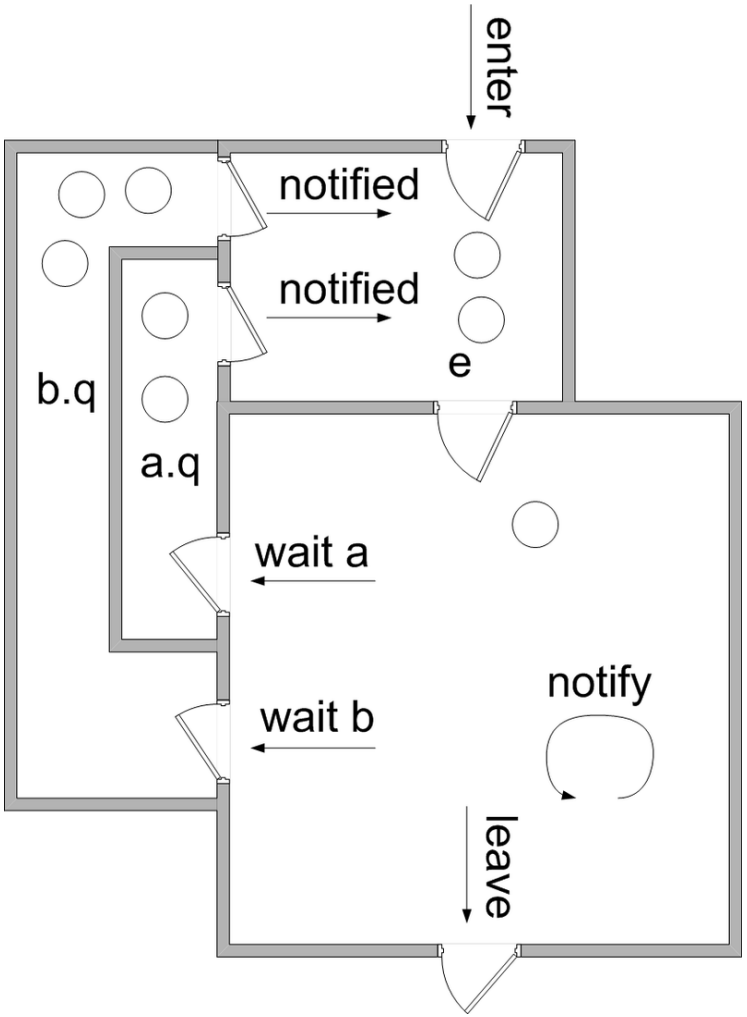
Monitor narzędzie języka programowania lub wzorzec projektowy. Zawiera metody, zmienne warunkowe i zmienne lokalne. Tylko jeden wątek może modyfikować stan wewnętrzny monitora.

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



Monitory Hoare'a vs. Mesa





Monitor: Problem producent-konsument

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

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

  fn get() -> T:
    if queue.is_empty():
      nonempty.wait()
    T item = queue.pop()
    nonfull.signal()
    return item
```

```
monitor ProducerConsumerMesa:
  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
```

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.

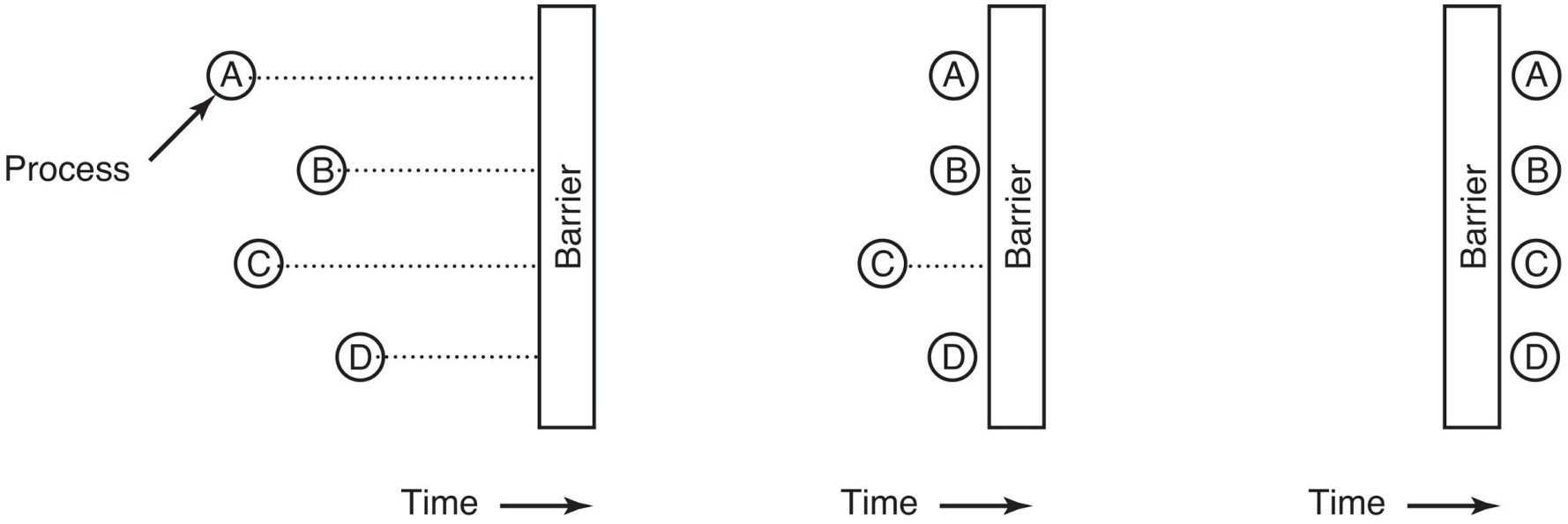
Możemy nadać priorytet czytelnikom (ang. *read-preferring RW-lock*) lub pisarzom (ang. *write-preferring RW-lock*). Kto wyczuwa głodzenie?

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.

Komunikacja

Pamięć dzielona (lokalnie)

Rodzic może utworzyć blok pamięci dzielonej wywołaniem `mmap` z flagą **`MAP_SHARED`** i utworzyć podproces. Ograniczone!

Uniwersalne rozwiązanie? POSIX.1 *shared memory* (`shm_overview`)
Nazwana pamięć dzielona (`shm_open`) istnieje póki nie zostanie usunięta (**`shm_unlink`**) albo do restartu. Początkowo długość zero, trzeba określić z użyciem **`ftruncate`**. Zasób plikopodobny (deskryptor pliku) odwzorowany w pamięci wywołaniem **`mmap`**.

Można również efektywnie dzielić pamięć przez plik → znów **`mmap`**.

Można też tworzyć anonimowe pliki odwzorowane w pamięci memfd_create i przesyłać je między procesami (o tym za chwilę).

Domena komunikacji

Nazewnictwo punktów końcowych, tj. czy adres oznacza:

- zasób sprzętowy (adres MAC karty sieciowej),
- zasób programowy (skrzynka pocztowa, gniazdo),
- proces, który może migrować między maszynami,
- brokera, który zna prawdziwy adres docelowy,
- jeden zasób (ang. *unicast*) czy wiele zasobów (ang. *multicast*).

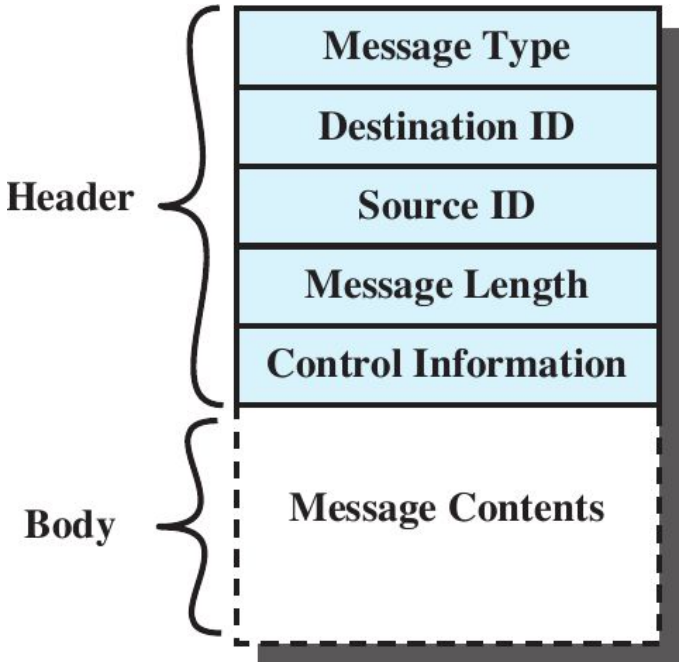
Semantyka operacji:

- zachowanie porządku przesyłania danych,
- eliminowanie duplikatów,
- niezawodność dostarczania danych (np. retransmisja, korekcja błędów)
- skojarzenie danych z sesją (protokoły połączeniowe)
- zachowanie granic między komunikatami,
- wsparcie dla komunikacji pozapasmowej (ang. *out-of-band*)

Przesyłanie komunikatów

Lokalne → kopiowanie danych między procesami: potoki, gniazda domeny uniksowej, skrzynki pocztowe, …

Rozproszone → protokoły sieciowe IPv4 lub IPv6: gniazda TCP, UDP, SCTP, RAW, …; zdalne wywołania procedur, ...



Potoki (rury?)

Jedno- lub dwukierunkowe strumieniowe przesyłanie danych + buforowanie. Nienazwane → potoki, nazwane → FIFO (mkfifo).

Q: Zapisujemy do potoku pakiety z wielu procesów.
Jaka jest gwarancja, że nie zostaną pofragmentowane?
A: *POSIX.1* mówi, że zapis pakietu < **`PIPE_BUF`** jest atomowy!
`PIPE_BUF` co najmniej 512, w Linuksie 4096.

Q: Ile danych może zbuforować jądro zanim nadawca się zablokuje?
A: Prezentacja zmiennych jądra: **`sysctl -a -r pipe`**

Potoki występują również w WindowsNT.

Potoki: przykład

```
int main(void) {
  int fd[2]
  if (pipe(fd) < 0)
    err_sys("pipe");

  pid_t pid = fork()
  if (pid < 0)
    err_sys("fork");

  if (pid > 0) {
    /* parent */
    close(fd[0]);
    write(fd[1],
          "hello world\n", 12);
  }
  if (pid == 0) {
    /* child */
    char line[MAXLINE];
    close(fd[1]);
    int n = read(fd[0], line,
                 MAXLINE);
    write(STDOUT_FILENO,
          line, n);
  }

  return EXIT_SUCCESS;
}
```

Gniazda BSD

Dwukierunkowa metoda komunikacji lokalnej lub sieciowej (do pewnego stopnia przezroczyste).

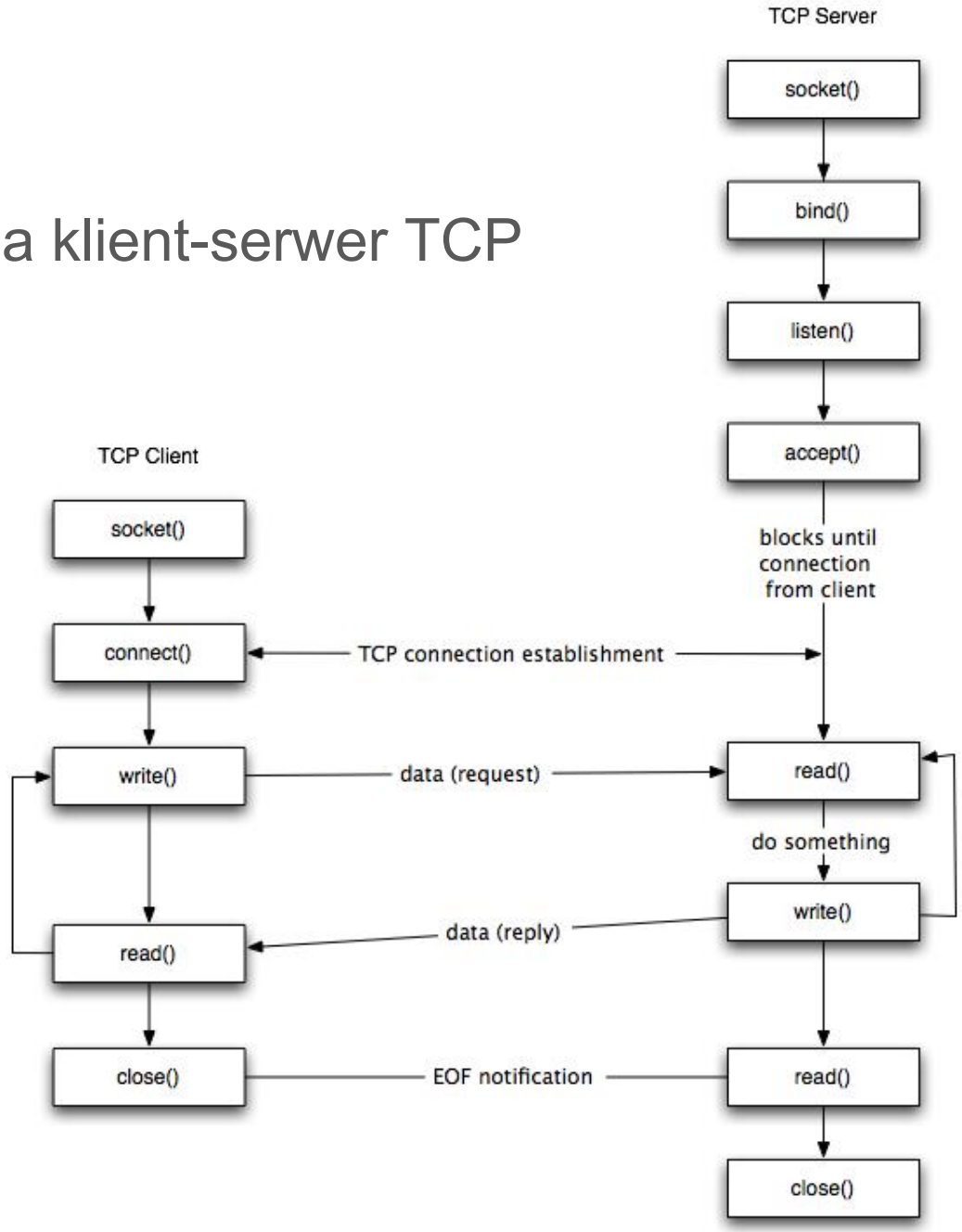
Protokół TCP → strumieniowy, połączeniowy, niezawodny i samoregulujący (kontrola przepływu, unikanie zatorów).

Do komunikacji wystarczą wywołania `read` (`send`) / `write` (`recv`).
Utworzenie gniazda serwera (`socket` → `bind` → `listen`).
Nawiązanie połączenia: klient → `connect`, serwer → `accept`.

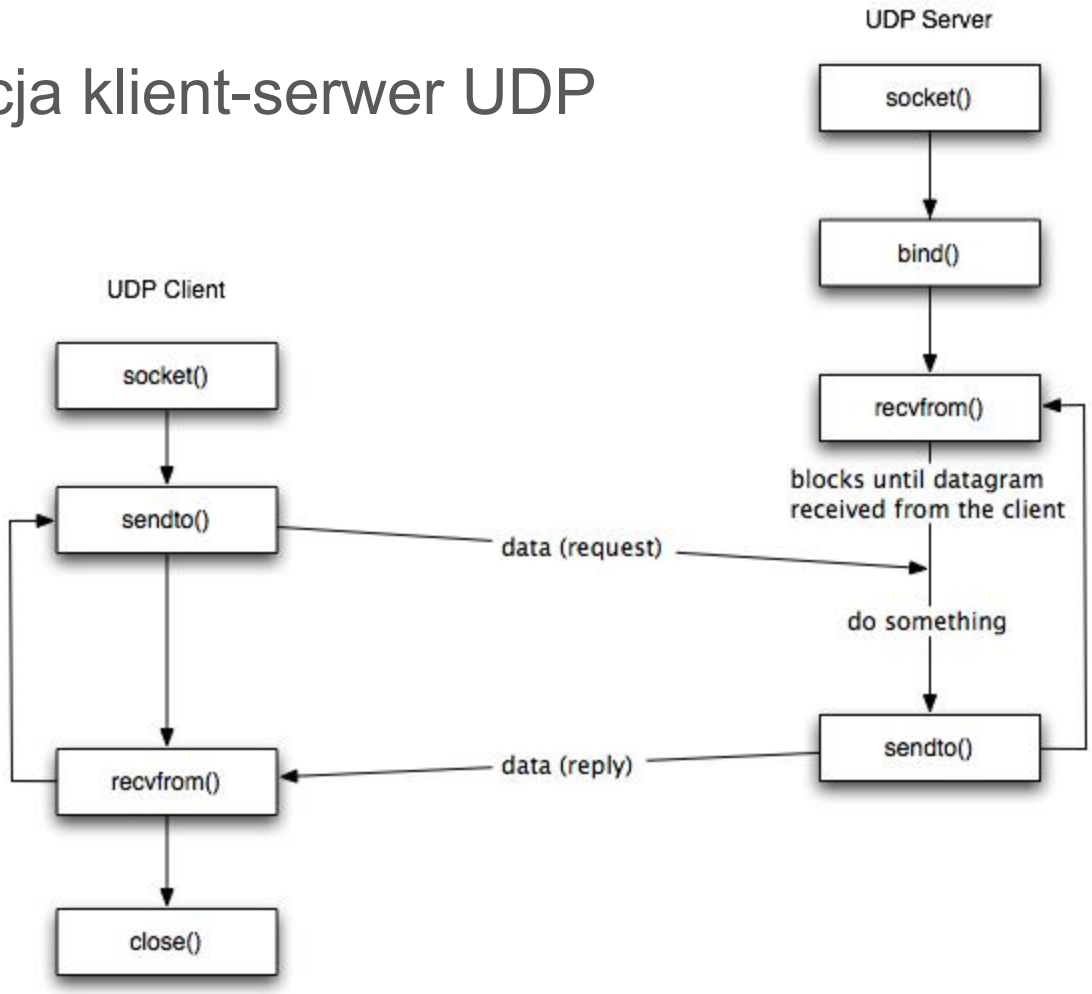
Protokół UDP → datagramowy, bezpołączeniowy, zawodny (możliwe: zagubienie, duplikacja, zmiana kolejności).

Podanie / pobranie adresu odbiorcy / nadawcy → `sendto` / `recvfrom`.

Komunikacja klient-serwer TCP



Komunikacja klient-serwer UDP



Gniazda domeny uniksowej

Dwukierunkowe przesyłanie strumieniowe (**`SOCK_STREAM`**), datagramowe (**`SOCK_DGRAM`**) lub sekwencyjne pakietowe (**`SOCK_SEQPACKET`**). Nienazwane gniazda tworzymy `socketpair`.

Q: Czy jądro może zmieniać kolejność pakietów?
A: Może **`SOCK_DGRAM`** (ale nie robi), nie może **`SOCK_SEQPACKET`**.

Q: Czy pakiety mogą być pofragmentowane?
A: Nie, do przekroczenia pewnego limitu (**`SO_SNDBUF`**, **`SO_RCVBUF`**). Można to sprawdzić `{get,set}sockopt` lub podejrzeć zmienne jądra poleceniem: **`sysctl -a -r 'net.core.[rw]mem.*'`**

Brak odpowiednika w WindowsNT!

Przenośna implementacja dwukierunkowego potoku

```c
#include <sys/socket.h>

/*
 * Returns a full-duplex pipe (a UNIX domain socket) with
 * the two file descriptors returned in fd[0] and fd[1].
 */
int fd_pipe(int fd[2])
{
  return socketpair(AF_UNIX, SOCK_STREAM, 0, fd);
}
```

Komunikaty pomocnicze

Dodatkowa funkcja gniazd domeny uniksowej →
przesyłanie między procesami zasobów i tożsamości (`cmsg`).

`SCM_RIGHTS` duplikowanie i przesyłanie deskryptorów tj. otwartych plików, gniazd, potoków, semaforów, pamięci dzielonej, urządzeń, …

`SCM_CREDENTIALS` wysyłamy identyfikator procesu, numer użytkownika i grupy. Jądro weryfikuje tożsamość i dostarcza pakiet.

Gniazdo klienta może przechodzić przez różne procesy serwera w zależności od etapu przetwarzania. Izolacja → bezpieczeństwo!

Unikanie kopiowania

Serwer WWW → kopiujemy dane binarne z pliku do gniazdka (np. wysyłamy obrazki, albo statyczne strony HTML5).

Q: Czy to niezbędne, żeby najpierw kopiować dane z pliku do przestrzeni użytkownika, a potem do gniazda sieciowego?
A: Całe szczęście nie → `sendfile` (różnice między uniksami)!

Nieustandaryzowany zestaw wywołań w Linuksie:

- `splice` (pipe ↔ fd),
- `vmsplice` (memory → pipe),
- `tee` (duplikowanie),
- `copy_file_range` (file → file),
- send + `MSG_ZEROCOPY` (memory → socket).

Skrzynki pocztowe

POSIX.1: `mq_overview`. Lokalne, nazwane i trwałe. Komunikaty mają priorytety. Asynchroniczne powiadamianie o komunikacie przychodzącym → sygnałem lub w wątku *pop-up*. Ograniczony rozmiar kolejki i komunikatu: **`sysctl -a -r mqueue`**.

WindowsNT: `Mailslots`. Nazwane i tymczasowe. Kto ma adres może wysyłać, ale tylko właściciel może odbierać. Działają sieciowo, ale tylko do 424 bajtów. Możliwy broadcast w domenie.

Skrzynki pocztowe: Problem producent-konsument

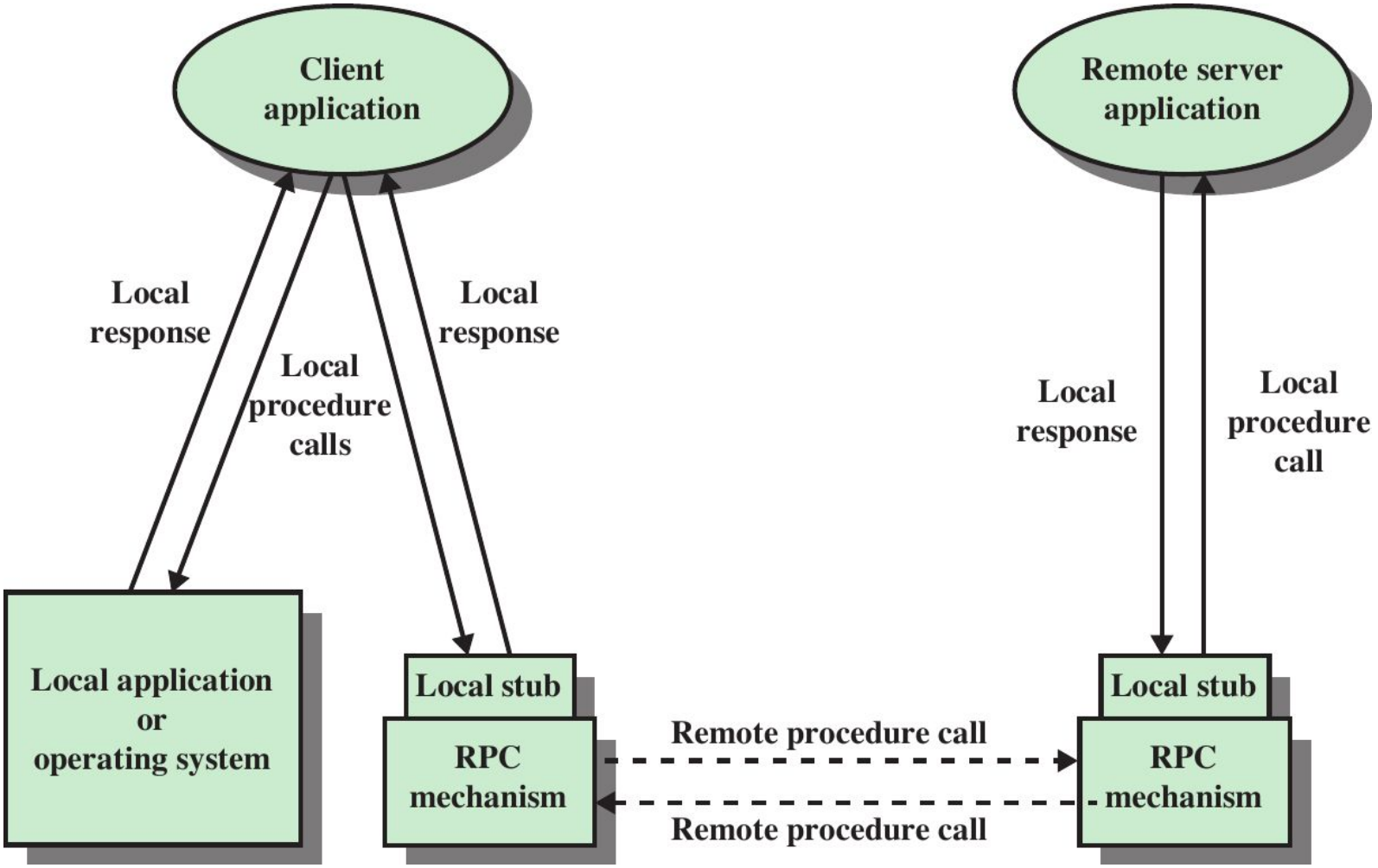
```
prod_mbox = Mailbox()
cons_mbox = Mailbox()

def producer():
 while True:
   item = produce()
   msg = prod_mbox.recv()
   msg.payload = item
   cons_mbox.send(msg)
```

```
def consumer():
 for i in range(100):
   prod_mbox.send(
        Msg(payload = None))

 while True:
   msg = cons_mbox.recv()
   item = msg.payload
   msg.payload = None
   prod_mbox.send(msg)
   consume(item)
```

Zdalne wywołanie procedur



Problemy z RPC

Klient i serwer RPC w różnych przestrzeniach adresowych, potencjalnie na różnych maszynach. Należy **przetoczyć** (ang. *marshalling*) argumenty i wyniki. Co z różnicami w kolejności bajtów (ang. *endianness*), szerokością typów, wskaźnikami, niejawnie współdzielonymi danymi?

Z pomocą przychodzi **IDL** (ang. Interface Definition Language). Niestety RPC nie jest **przezroczyste** jak obiecywano.

Środowisko Uniksowe nie przepada za RPC: The Art of Unix Programming: Problems and Methods to Avoid i Tanenbaum też…

Nie jest źle, rozwiązano część klasycznych problemów o czym piszą w A Critique of the Remote Procedure Call Paradigm - 30 years later. Idea ma się dobrze: Google Protocol Buffers, Google gRPC.

Pytania?