**NOTATKA: SYNCHRONIZACJA**

**wywołania sleep i wakeup** – operacje, które w momentach, kiedy procesy nie mogą wejść do swoich sekrytów, blokują je, zamiast marnotrawić czas procesora. sleep to wywołanie systemowe, które powoduje zablokowanie procesu wywołującego – zawieszenie go do czasu, kiedy inny proces go obudzi. wakeup wybudza ze sleep.

**semafor** – zmienna służąca do zliczania liczby zapisanych sygnałów wakeup.

* wartości semafora:
  + 0 – brak zapisanych sygnałów wakeup
  + > 0 – gdy istnieje jeden lub więcej zaległych sygnałów wakeup
* operacje na semaforze (niepodzielne):
  + up – inkrementuje wartość wskazanego semafora. Jeśli na tym semaforze był uśpiony jeden proces lub więcej procesów, które nie mogły wykonać wcześniejszej operacji down, to system wybiera jeden z nich (np. losowo) i zezwala na dokończenie operacji down. Więc po wykonaniu operacji up na semaforze, na którym były uśpione procesy, semafor w dalszym ciągu będzie miał wartość 0, ale będzie na nim uśpiony o jeden proces mniej.
  + down – sprawdza wartość zmiennej
    - = 0 – proces jest przełączany na chwilę w stan uśpienia bez wykonywania operacji down
    - > 0 – dekrementuje wartość (tzn. wykonuje operację up z argumentem 1 dla zapisanych sygnałów wakeup) i kontynuuje.
* system operacyjny na czas sprawdzania semafora powinien zablokować przerwania, zaktualizować semafor i jeśli trzeba – przełączyć proces do stanu uśpienia.
* rodzaje semaforów:
  + **semafor binarny (mutex) –** semafor inicjowany wartością 1 i używany przez dwa lub więcej procesów po to, by uzyskać pewność, że tylko jeden z nich może wejść do sekrytu w tym samym czasie - wzajemne wykluczanie
  + **semafor zliczający - synchronizacja** – licznik zestawu dostępnych zasobów
* problem semaforów: trzeba przyjąć założenie, że przynajmniej do fragmentu współdzielonej pamięci ma dostęp wiele procesów. Semafory mogą być przechowywane w jądrze, a dostęp do nich być możliwy tylko za pomocą wywsysów. W większości sysopków istnieje mechanizm pozwalający procesom współdzielić pewną część swojej przestrzeni adresowej z innymi procesami.

**mutex** – uproszczona wersja semafora; nadaje się do zarządzania wzajemnym wykluczaniem niektórych współdzielonych zasobów lub fragmentów kodu (wykorzystywane, gdy nie jest potrzebna właściwość zliczania)

* zmienna, która może znajdować się w jednym z dwóch stanów: odblokowany lub zablokowany. Do jego zaprezentowania jest potrzebny jeden bit. W praktyce 0 – odblokowany, cokolwiek innego – zablokowany.
* procedury:
  + mutex\_lock – kiedy wątek lub proces potrzebuje dostępu do sekryta. Jeśli muteks jest już odblokowany (sekryt dostępny), to wywołanie kończy się sukcesem i wątek wywołujący może wejść do sekrytu. Jeśli muteks jest zablokowany, wątek wywołujący zablokuje się do czasu, kiedy wątek znajdujący się w sekrycie zakończy w nim działania i wywoła mutex\_unlock. Jeśli na muteksie zablokowanych jest wiele wątków, losowo wybieramy jest jeden z nich i otrzymuje zgodę na założenie blokady.
  + mutex\_unlock – kiedy wątek zakończy swoje działania w sekrycie
* różnica między enter\_region: kiedy mutex\_lock nie uda się ustawić blokady, wywołuje funkcję thread\_yield po to, by przekazać procesor do innego wątku. W konsekwencji nie ma aktywnego oczekiwania (powtarzanie testowania blokady). Kiedy wątek uruchomi się następnym razem, ponownie analizuje blokadę.

**L6.Z1.**

**Opisz semantykę operacji FUTEX\_WAIT i FUTEX\_WAKE mechanizmu futex(2) wykorzystywanego w systemie Linux do implementacji środków synchronizacji w przestrzeni użytkownika. Podaj w pseudokodzie implementację funkcji lock i unlock semafora binarnego korzystając wyłącznie z futeksów i atomowej instrukcji compare-and-swap. Odczyty i zapisy są atomowe.**

**futex (*fast user space mutex*)**

* składa się z dwóch części:
  + usługa jądra – zapewnia kolejkę oczekiwania, która
* idea: blokady pętlowe są szybkie, jeśli czas oczekiwania jest krótki, w przeciwnym razie powodują marnotrawienie cykli procesora. Z tego powodu, w przypadku gdy rywalizacja jest duża, najbardziej wydajne jest zablokowanie procesu i zlecenie jądru, aby odblokowanie go nastąpiło dopiero wtedy, gdy blokada zostanie zwolniona. Powoduje to odwrotny problem: sprawdza się to w przypadku dużej rywalizacji, ale ciągłe przełączanie do jądra jest kosztowne, gdy rywalizacji nie ma zbyt wiele. To, ile będzie rywalizacji, nie jest do przewidzenia. Futeks łączy zalety obu światów. Jest własnością Linuksa, która implementuje podstawowe blokowanie (podobnie jak muteks), ale unika odwoływania się do jądra, jeśli nie jest to konieczne.
* składa się z dwóch części:
  + przestrzeń jądra: kolejka oczekiwania, która umożliwia oczekiwanie na blokadę wielu procesom. Procesy nie będą działać, jeśli jądro wyraźnie ich nie odblokuje. Operacje na kolejce są drogie, wymagają wywsysów: waking up waiting proces, umieszczenie procesu w kolejce
  + przestrzeń użytkownika: int

W przypadku braku rywalizacji, futex działa w całości w przestrzeni użytkownika, nie wykonuje wywsysów.

Semantyka:

każda operacja futeksa startuje z user space, ale może być konieczna komunikacja z kernelem, używając wywsysa futex(2). Futex ma licznik domyślnie ustawiony na 1.

futex\_wait zmniejsza o 1 licznik

* jeśli licznik jest nieujemny, to oznacza, ze nie ma rywalizacji i można przejść do sekcji krytycznej.
* jeśli jest ujemny, to oznacza, ze jest rywalizacja i za pośrednictwem system wywsysa futex() wysyła się proces do kolejki zablokowanych.

futex\_wake zwiększa o 1 licznik.

* jeśli zwiększono licznik z 0 do 1, to oznacza, ze nie było rywalizacji, więc nic nie trzeba robić.
* jeśli licznik nie jest dodatni, to trzeba wybudzić dowolną liczbę uśpionych procesów. Używa się do tego system wywsysa futex(2).

**semafor binarny –** semafor inicjowany wartością 1 i używany przez dwa lub więcej procesów po to, by uzyskać pewność, że tylko jeden z nich może wejść do sekrytu w tym samym czasie

**instrukcja atomowa –** obserwator nie może zobaczyć wyników pośrednich

futex:

0 - unlocked

1 - locked & waiters = 0

2 - locked & waiters >= 0

void lock(int\* semaphore) {

while (not cas(\*semaphore, 0, 1)) {

futex(FUTEX\_WAIT, semaphore, 1);

}

}

void unlock(int\* semaphore) {

if (cas(\*semaphore, 1, 0))

return;

if (cas(\*semaphore, 2, 0))

futex(FUTEX\_WAKE, semaphore, 1)

}

**L6.Z2.**

**Podaj w pseudokodzie implementację semafora z operacjami init, wait i post używając wyłącznie muteksów i zmiennych warunkowych. Dopuszczamy ujemną wartość semafora.**

**monitor** – kolekcja procedur, zmiennych i struktur danych pogrupowanych ze sobą w specjalnym rodzaju modułu lub pakietu.

* procesy mogą wywoływać procedury w monitorze, kiedy tylko tego chcą, ale z poziomu procedur zadeklarowanych poza monitorem nie mogą bezpośrednio korzystać z wewnętrznych struktur danych monitora
* w dowolnym momencie w monitorze może być aktywny tylko jeden proces
* zmienne warunkowe - sposób na to, by procesy blokowały się w czasie, gdy nie mogą kontynuować działania

**zmienna warunkowa**

* ma dwie operacje: wait i signal.
* nie są licznikami – nie akumulują sygnałów, jeśli zostanie wysłany sygnał do zmiennej warunkowej, na który nikt nie czeka, zostanie utracony na zawsze – operacja wait musi być wykonana przed operacją signal
* kiedy procedura monitora wykryje, że nie może kontynuować działania (np. producent wykryje, że bufor jest pełny), wykonuje operację wait na wybranej zmiennej warunkowej, np. full. operacja ta powoduje zablokowanie procesu wywołującego. Pozwala również innemu procesowi, który wcześniej nie mógł wejść do monitora, aby teraz do niego wszedł.
* inny proces, np. konsument, może obudzić swojego uśpionego partnera poprzez przesłanie sygnału z wykorzystaniem zmiennej warunkowej, na którą jego partner oczekuje. Aby uniknąć jednoczesnego występowania dwóch aktywnych procesów w monitorze, potrzebna jest reguła, która informauje o tym, co się dzieje po wykonaniu operacji signal.
  + Hoare zaproponował umożliwienie działania przebudzonemu procesowi i zawieszenie drugiego z nich
  + Hansen: wymaganie od procesu wykonującego operację signal natychmiastowego opuszczenia monitora
* różnica między sleep i wakeup a wait i signal: te pierwsze zawodzą, gdy jeden proces próbuje przejść w stan uśpienia, natomiast drugi próbuje go obudzić. W przypadku monitorów to nie może się zdarzyć – automatyczne wzajemne wykluczanie procedur.

void wait(semaphore \*s) {

mutex\_lock(s->mutex);

s->value--;

if (s->value < 0) {

do {

condvar\_wait(s->cond, s->mtx);

} while (s->wakeups < 1);

s->wakeups--;

}

mutex\_unlock(s->mutex);

}

void post(semaphore \*s) {

mutex\_lock(s->mutex);

s->value++;

if (s->value <= 0) {

s->wakeups++;

condvar\_signal(s->cond);

}

mutex\_unlock(s->mutex);

}

struct semaphore {

int value;

int wakeups;

mutex\* mtx;

condvar\* cond;

}

void init(semaphore\* s) {

s->value = 0;

s->pending\_wakeups = 0;

mutex\_init(s->mtx);

condvar\_init(s->cond)