13. Synchronizacja procesów i wątków

Wstęp

Procesy działające współbieżnie w systemie wielozadaniowym mogą współpracować lub być niezależne od siebie. Współpracujące procesy i wątki wymagają od systemu dostarczenia mechanizmów, które umożliwią im komunikowanie się ze sobą oraz synchronizowanie swoich działań.

Wykład 13 zawiera omówienie podstawowych zagadnień dotyczących synchronizacji procesów i wątków. Prezentujemy kolejno rozwiązania najważniejszych problemów synchronizacji, do których należy zaliczyć:

- problem sekcji krytycznej,
- problem ograniczonego buforowania,
- problem czytelników i pisarzy (pierwszy i drugi),
- problem zakleszczeń.

Klasyczne problemy synchronizacji reprezentują pewne klasy rzeczywistych problemów synchronizacji procesów. Służą zazwyczaj jako problemy testowe dla opracowanych metod synchronizacji. Problemy te zostały już częściowo omówione w ramach przedmiotu „Architektury systemów komputerowych” [5].

W dalszej części wykładu prezentujemy najważniejsze mechanizmy oferowane przez system Linux: semafory IPC Systemu V, semafory POSIX, mutexy i zmienne warunkowe. Omawiamy sposób ich implementacji oraz podstawowe operacje dostępne poprzez funkcje systemowe.

13.1. Problem sekcji krytycznej

Problem sekcji krytycznej jest podstawowym problemem synchronizacji współpracujących procesów i wątków. **Sekcją krytyczną** określany jest fragment kodu procesu wymagający wyłącznego dostępu do określonych zasobów np. plików lub wspólnych struktur danych.

Problem sekcji krytycznej polega na zapewnieniu wzajemnego wykluczania (wzajemnego wyłączania) procesów w dostępie do niepodzielnego zasobu. Problem nabiera jeszcze większego znaczenia w przypadku wątków, które korzystają przecież ze wspólnej przestrzeni adresowej.

Ogólną strukturę kodu programu można przedstawić w następujący sposób:

```
while (true) {
   początek kodu
   ...
   sekcja wejściowa
      sekcja krytyczna
   sekcja wyjściowa
   reszta kodu
   ...
}
```

Sekcję krytyczną otaczają tu dwie dodatkowe sekcje. W **sekcji wejściowej** proces oczekuje na pozwolenie wejścia do sekcji krytycznej. W **sekcji wyjściowej** proces sygnalizuje wyjście z sekcji krytycznej.

Rozwiązanie problemu sekcji krytycznej sprowadza się do prawidłowego skonstruowania sekcji wejściowej i wyjściowej. Każde rozwiązanie powinno spełniać trzy następujące warunki:

1. **wzajemne wyłączanie procesów** - tylko jeden proces może działać w swojej sekcji krytycznej,
2. **postęp** - jeżeli żaden proces nie działa w swojej sekcji krytycznej oraz istnieją procesy oczekujące na wejście do swoich sekcji krytycznych, to wybór procesu następuje w skończonym czasie,
3. **ograniczone czekanie** - dla każdego procesu czas oczekiwania na pozwolenie wejścia do ich sekcji krytycznych jest ograniczony.

Znane rozwiązania problemu sekcji krytycznej można podzielić na czysto programowe oraz takie, które wymagają wsparcia sprzętowego.

Rozwiązanie programowe dla dwóch procesów - algorytm Petersona

W przypadku, gdy sprzęt komputerowy nie daje żadnego wsparcia dla synchronizacji, jedynym rozwiązaniem jest czysto programowa implementacja sekcji wejściowej i wyjściowej. W najprostszym przypadku synchronizacji tylko dwóch procesów poprawne rozwiązanie zapewnia przedstawiony poniżej algorytm Petersona.

Procesy korzystają z dwóch współdzielonych zmiennych:

```
boolean flaga[2];
int numer;
flaga[0]=false; flaga[1]=false;
```

Zmienna `flaga` służy do zasygnalizowania chęci wejścia do sekcji krytycznej. Zmienna `numer` wskazuje proces, który może wejść do sekcji. Jeśli proces P_i chce wejść do sekcji krytycznej, ustawia swoją flagę `flaga[i]` na wartość `true` oraz ustawia numer drugiego procesu `numer=j`. Następnie sprawdza, czy jednocześnie są ustawione numer i flaga procesu P_j, co oznacza, że jest on w sekcji krytycznej. Proces może wejść do swojej sekcji dopiero, gdy co najmniej jeden z tych warunków nie będzie spełniony.

```
while (true) {
      flaga[i] = true;
      numer = j;
      while (flaga[j] && numer==j);
            sekcja krytyczna
      flaga[i] = false;
            reszta kodu
}
```

Rozwiązania ze wsparciem sprzętowym

Większość systemów komputerowych zapewnia jakiś rodzaj wsparcia sprzętowego dla realizacji wzajemnego wyłączania w sekcji krytycznej. Sekcja wejściowa polega wtedy na założeniu blokady a sekcja wyjściowa sprowadza się do jej zwolnienia.

```
while (true) {
      załóż blokadę
            sekcja krytyczna
      zwolnij blokadę
            reszta kodu
}
```

W systemach jednoprocesorowych wsparcie sprzętowe może polegać na wyłączeniu przerwań na czas operacji w sekcji krytycznej. Dopóki do procesora nie docierają przerwania z czasomierza, wykonywany proces nie zostanie wywłaszczony (wykonywany właśnie fragment kodu procesu nie zostanie przerwany), a żaden inny proces nie zostanie wznowiony.

W systemach wieloprocesorowych rozwiązanie takie jest jednak bardzo nieefektywne i właściwie nieprzydatne. Procesy korzystające ze współdzielonych struktur danych mogą być bowiem wykonywane na różnych procesorach, więc wyłączanie przerwań musiałoby przebiegać synchroniczne dla wszystkich procesorów. Konieczne staje się wykorzystanie specjalnych atomowych (nieprzerywalnych) rozkazów sprzętowych. Większość nowoczesnych systemów komputerowych realizuje takie operacje w postaci pojedynczych rozkazów procesora, które pozwalają wykonać niepodzielnie dwie kolejne operacje. Zazwyczaj są to rozkazy jednego z dwóch typów:

- **`testuj_i_ustaw`** - sprawdza i zmienia zawartość jednego słowa pamięci,
- **`zamień`** - zamienia zawartości dwóch słów pamięci.

13.2. Semafory

Podstawowym mechanizmem synchronizacji procesów jest semafor. **Semafor** został klasycznie zdefiniowany jako liczba całkowita, której można nadać wartość początkową i która dostępna jest tylko za pomocą dwóch niepodzielnych operacji: `wait` i `signal`**.** W tablicy 13.1 przedstawione zostały klasyczne definicje operacji na semaforach oraz wyjaśnione ich znaczenie.

Tablica 13.1 Klasyczne definicje operacji na semaforach

Operacja	Definicja	Znaczenie
`wait(S)`	`wait(S) { while(S<=0); S=S-1; }`	Proces oczekuje na zwolnienie semafora (zwolnienie zasobu chronionego semaforem), a gdy to nastąpi, zajmuje semafor (zajmuje ten zasób).
`signal(S)`	`signal(S) { S=S+1; }`	Proces zwalnia semafor (zwalnia zasób chroniony semaforem).

Obydwie operacje muszą być niepodzielne, to znaczy, że nie mogą zostać przerwane w trakcie wykonywania. Dzięki temu w każdym momencie tylko jeden proces może wykonywać operację zmiany wartości semafora. Rozdzielenie sprawdzenia wartości semafora i ewentualnej zmiany tej wartości w czasie operacji `wait` mogłoby doprowadzić do sytuacji, gdy kilka procesów równocześnie modyfikuje semafor.

Konkretne realizacje semaforów mogą odbiegać od klasycznej definicji zarówno w zakresie typu obiektu, jak i przebiegu operacji. Typowo stosowane są następujące realizacje:

- semafory binarne, przyjmujące tylko dwie wartości: 0 i 1,
- semafory wielowartościowe z możliwością zmiany o dowolną wartość w jednej operacji,
- semafory realizowane w postaci plików z operacjami:
    - `wait` - otwórz na wyłączność plik o ustalonej nazwie,
    - `signal` - zamknij plik.

Każda realizacja musi jednak zapewniać:

- niepodzielność operacji na semaforze,
- oczekiwanie procesu na osiągnięcie pożądanej wartości semafora.

Niepodzielność operacji na semaforze oznacza, że muszą być one realizowane przez system komputerowy w taki sposób, aby były nieprzerywalne. Niepodzielność może być osiągnięta poprzez:

- zabronienie obsługi przerwań na czas operacji na semaforze,
- wykorzystanie sprzętowych rozkazów synchronizacji, które umożliwiają wykonanie sprawdzenia wartości i jej zmianę w jednym rozkazie procesora,
- wykorzystanie programowego algorytmu synchronizacji dostępu do sekcji krytycznej (w tym przypadku jest nią semafor).

Oczekiwanie procesu na zmianę wartości semafora można zrealizować na dwa sposoby poprzez:

- aktywne czekanie procesu, czyli wirującą blokadę,
- usypianie i budzenie procesu.

Aktywne czekanie polega na tym, że proces w pętli sprawdza stan semafora, czyli wiruje w niewielkim fragmencie programu. Taka wirująca blokada zużywa czas procesora. Jest jednak użyteczna w systemach wieloprocesorowych, gdy przewidywany czas oczekiwania jest krótki, ponieważ nie wymaga przełączania kontekstu procesu. Przy długim czasie oczekiwania na zmianę wartości semafora, zdecydowanie korzystniejsze jest usypianie procesów i budzenie ich, gdy semafor osiągnie oczekiwaną wartość. Wymaga to jednak zmodyfikowania operacji na semaforach. Z każdym semaforem należy związać kolejkę procesów oczekujących na zmianę jego wartości. Semafor musi być zatem opisany strukturą danych zawierającą dwa pola: wartość całkowita semafora i wskaźnik do listy procesów w kolejce. Nowe definicje podano w tablicy 13.2.

Tablica 13.2 Nowe definicje operacji na semaforach z usypianiem

Operacja	Definicja	Znaczenie

wait(S)	wait(S) {
	  S.wartosc--;
	  if (S.wartosc < 0) {
	  dodaj ten proces do S.lista
	  block();
	  }
	}
		Proces zajmuje semafor: zmniejsza jego wartość o 1 i sprawdza, czy jest nieujemna. Jeśli wartość jest ujemna, proces oczekuje na zwolnienie semafora. Zostaje wtedy wstrzymany (zablokowany) i umieszczony w kolejce procesów oczekujących na zmianę wartości semafora.

signal(S)	signal(S) {
	  S.wartosc++;
	  if (S.wartosc <= 0) {
	  usuń jakiś proces P z S.lista
	  wakeup(P);
	  }
	}
		Proces zwalnia semafor. Wartość semafora jest zwiększana o 1. Jeśli wartość jest dodatnia, to jeden z procesów oczekujących jest budzony i przenoszony z kolejki czekających do kolejki procesów gotowych.

Nowe definicje dopuszczają ujemne wartości semafora. W takim przypadku, wartość bezwzględna określa liczbę procesów czekających na zajęcie danego semafora. Obsługa kolejki procesów oczekujących na zmianę wartości semafora powinna być realizowana za pomocą sprawiedliwego algorytmu szeregowania (np. algorytmu FCFS) tak, aby żaden proces nie czekał nieskończenie długo.

Semafory są bardzo uniwersalnym narzędziem synchronizacji udostępnianym programistom przez system operacyjny. Nie posiadają jednakże żadnego wbudowanego mechanizmu kontrolnego, zatem to na programistach spoczywa odpowiedzialność za stworzenie poprawnego schematu synchronizacji z użyciem semaforów Niewłaściwe użycie semaforów może prowadzić do dwóch niebezpiecznych zjawisk: zakleszczenia lub głodzenia procesów.

Do **zakleszczenia** zbioru procesów dojdzie wtedy, gdy każdy proces ze zbioru zajmie jakiś semafor i będzie czekał na zwolnienie innego semafora zajmowanego przez inny z czekających procesów. Zjawisko to ilustruje poniższy przykład:

Proces P1

```
wait(S1);
wait(S2);
      sekcja krytyczna
signal(S1);
signal(S2);
```

Proces P2

```
wait(S2);
wait(S1);
      sekcja krytyczna
signal(S2);
signal(S1);
```

Proces P1 zajmuje semafor S1 a potem próbuje zająć S2, natomiast proces P2 robi to w odwrotnej kolejności. Obydwa procesy zajmą po jednym semaforze i będą czekać w nieskończoność na zwolnienie drugiego. Problem zakleszczeń został szerzej omówiony w punkcie 13.4.

Głodzenie procesów, czyli blokowanie nieskończone polega na tym, że tylko wybrane procesy czekają w nieskończoność pod semaforem (na przykład z powodu niskiego priorytetu). Zwykle dochodzi do tego, gdy zostanie zastosowany niewłaściwy algorytm szeregowanie procesów w kolejce do semafora.

13.3. Wykorzystanie semaforów w typowych problemach synchronizacji

Problem sekcji krytycznej

Typowe zastosowanie semafora do rozwiązania problemu sekcji krytycznej wygląda następująco:

```
while (true) {
   początek kodu
   ...
   wait(S)
      sekcja krytyczna
   signal(S)
   reszta kodu
}
```

W sekcji wejściowej proces zajmuje semafor, następnie wykonuje krytyczne operacje, po czym zwalnia semafor w sekcji wyjściowej. W oczywisty sposób rozwiązanie to zapewnia wzajemne wyłączanie procesów, ponieważ operacje na semaforze są niepodzielne (są realizowane przez system komputerowy w sposób niepodzielny). Spełnienie warunków postępu i ograniczonego czekania zapewnia się poprzez odpowiednią realizację budzenia procesów oczekujących pod semaforem.

Problem ograniczonego buforowania (problem producenta i konsumenta z ograniczonym buforem)

W problemie ograniczonego buforowania występuje wielu producentów i wielu konsumentów, którzy komunikują się przez bufor o ograniczonej długości. Producenci zapisują elementy danych do kolejnych pustych pól bufora, a konsumenci odczytują i usuwają dane z kolejnych pól.

Rozwiązanie wymaga użycia trzech semaforów. Semafor binarny **`mutex`** zapewnia wzajemne wyłączanie procesów w dostępie do bufora:

```
mutex = 1;
```

Semafory wielowartościowe **`miejsce`** i **`element`** określają odpowiednio liczbę wolnych i zajętych pól bufora. Przyjmują wartości początkowe:

```
miejsce = N, element = 0;
```

Struktury procesów producenta i konsumenta wyglądają następująco:

Producent

```
while (true) {
      wytwórz element
      wait(miejsce);
      wait(mutex);
            umieść element w buforze
      signal(mutex);
      signal(element);
}
```

Konsument

```
while (true) {
      wait(element);
      wait(mutex);
            pobierz element z bufora
      signal(mutex);
      signal(miejsce);
      zużyj element
}
```

Proces producenta umieszcza w buforze kolejne wytworzone elementy. Musi w tym celu sprawdzić, czy w buforze jest jeszcze wolne miejsce oraz zarezerwować je, nie blokując przy tym dostępu do bufora. Operacja `wait(miejsce)` powoduje zmniejszenie wartości semafora `miejsce`, wskazującego liczbę wolnych pól bufora. Jeśli bufor jest zapełniony (`miejsce=0`), to proces zostaje zablokowany w oczekiwaniu na zwolnienie miejsca (niezerową wartość semafora). Następnie producent zajmuje semafor `mutex` i w sekcji krytycznej umieszcza element w buforze. W sekcji wyjściowej proces zwalnia `mutex` oraz zwiększa wartość semafora `element`, określającego liczbę elementów zapisanych w buforze.

Proces konsumenta sprawdza, czy w buforze są jakieś elementy do pobrania, wykonując operację `wait(element)`. Jeśli bufor jest pusty, proces zostaje zablokowany w oczekiwaniu na jego wypełnienie. W przeciwnym przypadku wartość semafora element zmniejsza się, informując inne procesy o zamiarze pobrania jednego elementu zapisanego w buforze. Następnie konsument zajmuje `mutex`, w sekcji krytycznej pobiera element z bufora (odczytuje i usuwa) i w sekcji wyjściowej zwalnia `mutex` i zwiększa wartość semafora `miejsce`.

Pierwszy problem czytelników i pisarzy

Kilku czytelników i kilku pisarzy korzysta ze wspólnego obiektu danych. Każdy pisarz musi uzyskać wyłączny dostęp do obiektu co oznacza, że jednocześnie nie może korzystać z obiektu żaden inny pisarz ani czytelnik. Natomiast wielu czytelników może czytać jednocześnie.

W pierwszym problemie uprzywilejowani są czytelnicy. Czytelnik musi czekać na dostęp do obiektu tylko wtedy, gdy pisarz rozpoczął pisanie. Natomiast kolejni czytelnicy mogą w każdej chwili dołączyć do czytelników, którzy już uzyskali dostęp do obiektu. Nie ma przy tym znaczenia, czy czeka jakiś pisarz.

Do rozwiązania problemu dla dowolnej liczby czytelników I pisarzy potrzebne są dwa semafory binarne oraz zmienna współdzielona. Semafor `writer` zapewnia wzajemne wyłączanie pisarzy w dostępie do obiektu, natomiast semafor `mutex` chroni dostęp do sekcji krytycznej, w której czytelnicy modyfikują zmienną `readcount`, będącą licznikiem czytelników korzystających z zasobu. Wartości początkowe ustawione są następująco:

```
mutex=1, writer=1;
int readcount=0;
```

Pisarz

```
while(true) {
      wait(writer);
            pisanie
      signal(writer);
}
```

Czytelnik

```
while(true) {
      wait(mutex);
      readcount++;
      if (readcount== 1)
            wait(writer);
      signal(mutex);
            czytanie
      wait(mutex);
      readcount--;
      if (readcount== 0)
            signal(writer);
      signal(mutex);
}
```

Proces pisarza zajmuje semafor `writer` przed rozpoczęciem pisania, a po zakończeniu zwalnia go. Blokuje w ten sposób innych pisarzy i czytelników przed dostępem do obiektu danych.

Przed rozpoczęciem czytania proces czytelnika zwiększa licznik `readcount` i sprawdza jego wartość. Jeśli jest pierwszym czytelnikiem w czytelni (`readcount=1`), to zajmuje semafor `writer`, żeby zablokować dostęp pisarzom. Operacje te wykonuje w sekcji krytycznej pod osłoną semafora `mutex`. Po zakończeniu czytania czytelnik zmniejsza licznik `readcount`. Jeśli jest ostatnim czytelnikiem (`readcount=0`), to zwalnia semafor `writer`, blokujący pisarzy.

Drugi problem czytelników i pisarzy

W drugim problemie czytelników i pisarzy, uprzywilejowanie uzyskują pisarze. Jeśli jakiś pisarz czeka na dostęp do obiektu, to żaden nowy czytelnik nie może rozpocząć czytania. Pisarz musi tylko zaczekać na wyjście z czytelni wszystkich czytelników, którzy rozpoczęli czytanie przed nim.

Rozwiązanie tego problemu jest znacznie trudniejsze niż pierwszego problemu. Tym razem potrzebnych jest pięć semaforów oraz dwie zmienne współdzielone.

Zmienna współdzielona `readcount` służy do zliczania czytelników, którzy uzyskali dostęp do danych, a zmienna `writecount` zlicza pisarzy oczekujących na dane:

```
int readcount = 0;
int writecount = 0;
```

Dostęp do zmiennej `readcount` odbywa się w sekcji krytycznej pod ochroną semafora `mutex_r`, natomiast dostęp do `writecount` jest chroniony przez semafor `mutex_w`. Semafor `writer` zapewnia wzajemne wyłączanie pisarzy, podczas gdy semafor `reader` powstrzymuje nowych czytelników przed dołączaniem do czytania danych, gdy przynajmniej jeden pisarz czeka na możliwość dostępu do danych. Konieczne jest jeszcze ograniczenie liczby czytelników oczekujących w kolejce na semafor `reader`, aby nie dopuścić do głodzenia procesów pisarzy. Czytelnicy czekają zatem w kolejce związanej z dodatkowym semaforem `mutex_q`.

Semafory przyjmują następujące wartości początkowe:

```
mutex_w = 1; mutex_r = 1; mutex_q = 1;
```

```
writer = 1; reader = 1;
```

Poniżej przedstawiono typowe rozwiązanie.

Pisarz

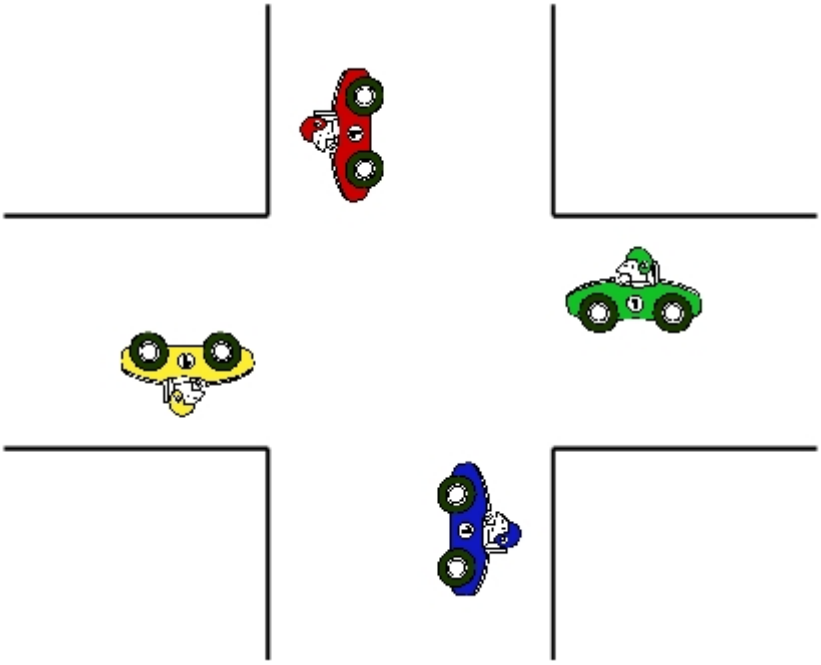
```
while(true) {
      wait(mutex_w);
      writecount++;
      if (writecount== 1)
            wait(reader);
      signal(mutex_w);

      wait(writer);
            pisanie
      signal(writer);

      wait(mutex_w);
      writecount--;
      if (writecount== 0)
            signal(reader);
      signal(mutex_w);
}
```

Czytelnik

```
while(true) {
      wait(mutex_q);
      wait(reader);
      wait(mutex_r);
      readcount++;
      if (readcount== 1)
            wait(writer);
      signal(mutex_r);
      signal(reader);
      signal(mutex_q);
            czytanie
      wait(mutex_r);
      readcount--;
      if (readcount== 0)
            signal(writer);
      signal(mutex_r) ;
}
```

13.4. Problem zakleszczeń

W punkcie 13.2 opisaliśmy przykład zakleszczenia dwóch procesów w wyniku niewłaściwego użycia dwóch semaforów. Przykład zakleszczenia z zupełnie innej dziedziny przedstawiamy na rysunku 13.1. Cztery samochody wjeżdżają jednocześnie na skrzyżowanie dróg równorzędnych i blokują sobie wzajemnie przejazd. Dochodzi do zakleszczenia a krytycznym, niepodzielnym zasobem okazuje się środek skrzyżowania.



Rys. 13.1 Przykład zakleszczenia na skrzyżowaniu

Zbiór współpracujących procesów pozostaje w stanie **zakleszczenia**, jeżeli każdy proces przetrzymuje jakiś zasób i jednocześnie oczekuje na przydział innego zasobu przetrzymywanego przez inny proces ze zbioru. Aby doszło do zakleszczenia muszą zostań spełnione cztery warunki konieczne:

1. **wzajemne wyłączanie procesów** – zasoby są niepodzielne, więc procesy nie mogą korzystać z nich jednocześnie,
2. **przetrzymywanie i oczekiwanie** – każdy proces ma przydzielony jakiś zasób o oczekuje na przydział innego zasobu,
3. **brak wywłaszczeń** – nie można odebrać procesowi przydzielonego zasobu,
4. **czekanie cykliczne** – istnieje cykliczna kolejka procesów czekających na zasoby zajmowane przez procesy z tego zbioru.

Ostatni z warunków można zilustrować w następujący sposób:

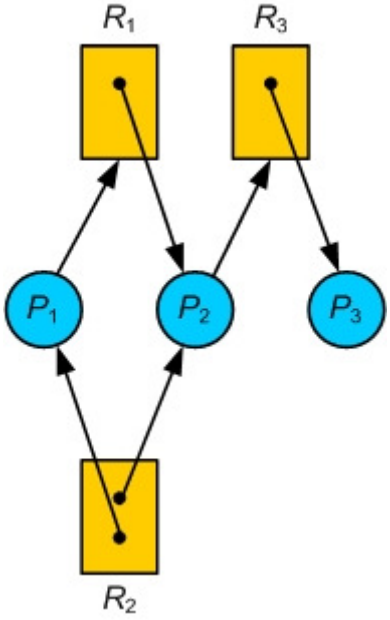
$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_2 \rightarrow P_3 \rightarrow \ldots \rightarrow P_n \rightarrow R_n \rightarrow P_1$$

gdzie P_i oznacza proces, a R_j – zasób.

Graf przydziału zasobów

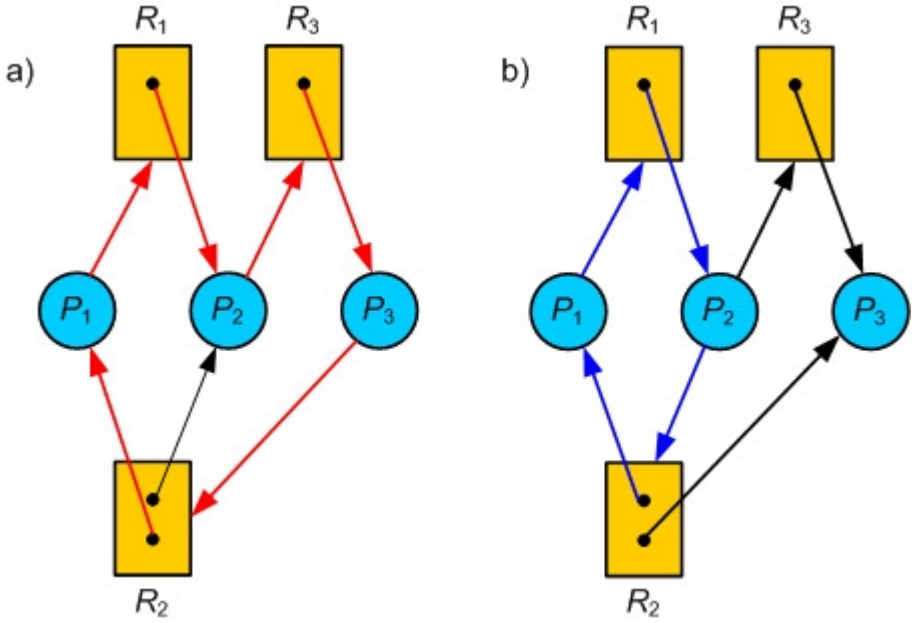
Zjawisko zakleszczenia można obrazowo przedstawić za pomocą grafu przydziału zasobów. Składa się on ze zbioru wierzchołków, reprezentujących procesy P_i i zasoby systemu R_j, oraz zbioru krawędzi. Krawędź skierowaną $P_i \rightarrow R_j$ nazywa się krawędzią zamówienia zasobu, a krawędź skierowaną $R_j \rightarrow P_i$ nazywa się krawędzią przydziału zasobu. Przykład takiego grafu dla trzech procesów i trzech zasobów przedstawia rysunek 13.2. Dostępne są 2 egzemplarze zasobu R_2 oraz

po jednym egzemplarzu zasobów R_1 i R_3. Proces P_1 ma zasób R_2 i czeka na R_1, proces P_2 dostał już dwa zasoby R_1 i R_2, a wciąż czeka na R_3, który został przydzielony procesowi P_3. W grafie nie ma cyklu.



Rys. 13.2 Graf przydziału zasobów

Jeżeli w takim grafie wystąpi cykl, to może to oznaczać, ale nie musi, że doszło do zakleszczenia procesów. Rysunek 13.3 przedstawia obydwa możliwe przypadki.



Rys. 13.3 Cykl w grafie przydziału zasobów: a) z zakleszczeniem, b) bez zakleszczenia

Rozwiązania problemu zakleszczeń

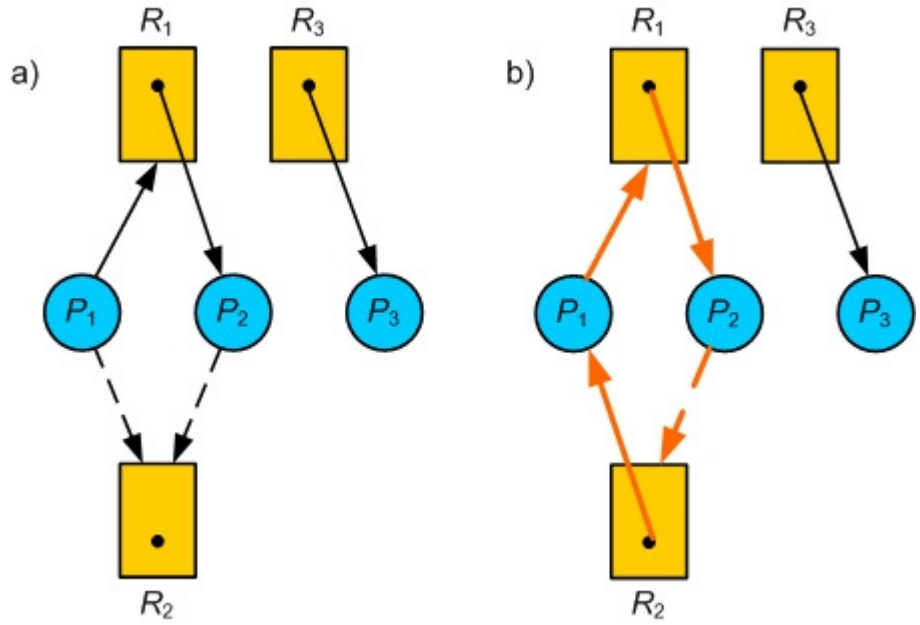
Rozwiązania problemu zakleszczeń można podzielić na trzy kategorie:

1. niedopuszczanie do powstania zakleszczenia,
2. wykrywanie i wychodzenie z zakleszczenia,
3. ignorowanie problemu i okresowe restartowanie systemu w przypadku zakleszczenia.

Pierwsza kategoria obejmuje metody zapobiegania zakleszczeniom i unikania zakleszczeń.

Metody **zapobiegania zakleszczeniom** koncentrują się na zapewnieniu, że co najmniej jeden z warunków koniecznych zakleszczenia nie będzie spełniony. Uzyskuje się to poprzez narzucenie ograniczeń m.in. na liczbę i kolejność zamawiania zasobów oraz wymuszanie ich zwalniania. W rezultacie znacząco spada efektywność wykorzystania zasobów systemu komputerowego.

Metody **unikania zakleszczeń** pozostawiają procesom swobodę zamawiania zasobów i koncentrują się na monitorowaniu stanu systemu. Dopóki system pozostaje w stanie bezpiecznym, wszystkie zamówienia na zasoby są realizowane. Jeśli realizacja jakiegoś zamówienia może doprowadzić do stanu zagrożenia zakleszczeniem, to zamówienie zostanie odrzucone. Konieczna jest jednak informacja *a priori* o planowanym wykorzystaniu zasobów przez wszystkie procesy. Algorytm unikania zakleszczeń, nazywany algorytmem grafu przydziału zasobów, został zilustrowany na Rys. 13.4. Wprowadza się tu dodatkowy typ krawędzi - krawędź deklaracji. Określa ona, że proces P_i może w przyszłości zamówić zasób R_j. Procesy P_1 i P_2 planują zamówienie zasobu R_2 (Rys. 13.4a). Jeśli proces P_1 złoży zamówienie i zostanie ono zrealizowane, to powstanie cykl w grafie, wskazując na stan zagrożenia (Rys. 13.4b). Złożenie zamówienia przez P_2 doprowadzi bowiem do stanu zakleszczenia. Zadaniem algorytmu będzie zatem odmowa realizacji poprzedniego zamówienia, które proces P_1 złożył na wolny zasób R_2 i niedopuszczenie do powstania cyklu w grafie.



Rys. 13.4 Ilustracja algorytm grafu przydziału zasobów: a) stan bezpieczny, b) stan zagrożenia

W przypadku, gdy każdy typ zasobu ma po kilka egzemplarzy, konieczne jest zastosowanie innego, bardziej złożonego algorytmu unikania zakleszczenia, znanego jako algorytm bankiera. Bliższe informacje na ten temat można znaleźć w [1].

13.5. Synchronizacja w systemach wieloprocesorowych

W systemach wieloprocesorowych problem synchronizacji nabiera nowego znaczenia, ponieważ procesory mogą jednocześnie wykonywać kod jądra. Oznacza to, że wiele wątków ma jednoczesny dostęp do danych jądra, a przy tym żaden pojedynczy wątek nie może wyłączyć przerwań na wszystkich procesorach. Konieczne jest więc stosowanie innych mechanizmów synchronizacji. Typowe mechanizmy to:

- operacje atomowe,
- blokady wyłączności dostępu (muteksy),
- zmienne warunkowe,
- blokady typu czytelnicy – pisarze,
- semafory.

Blokada wyłączności dostępu (ang. ***mut****ual* ***ex****clusion lock*), w skrócie **mutex**, to mechanizm skonstruowany na wzór semafora binarnego. Zasadnicza różnica polega na tym, że mutex może być zwolnione tylko przez wątek, który go zajął. Mutex może być realizowany jako:

- **blokada wirująca** (prosta) - wątek wykonuje aktywne czekanie i wykorzystuje sprzętowe rozkazy procesora,
- **blokada wstrzymująca** - wątek jest usypiany,
- **blokada adaptacyjna** - wątek czeka aktywnie, gdy wątek blokujący jest aktywny, wątek jest wstrzymywany, gdy wątek blokujący jest wstrzymany.

Blokady wyłączności dostępu wykorzystywane są przede wszystkim do ochrony krótkich sekcji kodu. W dłuższych sekcjach stosuje się dodatkowo zmienne warunkowe, które umożliwiają wątkom oczekiwanie na spełnienie określonych warunków pod ochroną muteksu. Wątek zajmuje mutex i sprawdza warunek. Jeśli warunek nie jest spełniony, to wątek zwalnia czasowo mutex i blokuje się na zmiennej warunkowej. Inny wątek może zająć mutex, zmienić warunek i zasygnalizować jego zmianę oraz zwolnić mutex. Czekający wątek jest wtedy budzony, zajmuje mutex, sprawdza ponownie warunek i zwalnia mutex.

13.6. Semafory IPC Systemu V

Utworzenie nowego zestawu semaforów lub dostęp do już istniejącego zapewnia funkcja systemowa **semget**().

```
int semget(key_t key, int nsems, int semflg);
```

gdzie:

key - klucz,

nsems - liczba semaforów w zbiorze,

semflg - flagi.

Funkcja zwraca identyfikator zbioru semaforów związanego z podaną wartością klucza **key**. Szczegółowy sposób działania wynika z flag ustawionych w argumencie **semflg**:

0 - funkcja udostępnia istniejący zbiór semaforów z podanym kluczem lub zwraca błąd, jeśli zbiór nie istnieje,

IPC_CREAT - funkcja tworzy nowy zbiór semaforów lub udostępnia istniejący zbiór z podanym kluczem,

IPC_EXCL | IPC_CREAT - funkcja tworzy nowy zbiór semaforów lub zwraca błąd, jeśli zbiór z podanym kluczem już istnieje.

Argument **semflg** może również opcjonalnie zawierać maskę praw dostępu do zbioru semaforów. Podawany jest wtedy jako suma bitowa stałych symbolicznych określających flagi oraz maski praw dostępu w kodzie ósemkowym np.:

```
IPC_CREAT | 0660
```

Operacje na semaforach umożliwia funkcja **semop**():

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

gdzie:

semid - identyfikator zbioru semaforów,

sops - wskaźnik do tablicy operacji, które mają być wykonane na zbiorze semaforów,

nsops - liczba operacji w tablicy.

Funkcja udostępnia trzy rodzaje operacji:

1. zajęcie zasobu chronionego semaforem,
2. zwolnienie zasobu,
3. oczekiwanie na 100% zużycie zasobu.

W jednym wywołaniu funkcji można zrealizować kilka operacji na semaforach ze zbioru. Jądro zapewnia niepodzielność realizacji wszystkich operacji, zarówno każdej z osobna, jak i zestawu operacji jako całości. Każda operacja dotyczy tylko jednego semafora wybranego ze zbioru i jest opisana w oddzielnej strukturze **sembuf**:

```
struct sembuf {
        short sem_num;  - indeks semafora w tablicy
        short sem_op;   - operacja na semaforze
        short sem_flg;  - flagi
};
```

Dla **sem_op** < 0 podana wartość bezwzględna zostanie odjęta od wartości semafora. Ponieważ wartość semafora nie może być ujemna, to proces może zostać zablokowany (uśpiony) do momentu uzyskania przez semafor odpowiedniej wartości, która umożliwi wykonanie operacji. Odpowiada to zajęciu zasobu.

Dla **sem_op** > 0 podana wartość zostanie dodana do wartości semafora. Tę operację można zawsze wykonać. Odpowiada to zwolnieniu zasobu.

Przy **sem_op** = 0 proces zostanie uśpiony do momenu, gdy semafor osiągnie wartość zerową. Oznacza to oczekiwanie na 100% zużycie zasobu.

Dla każdej operacji można ustawić dwie flagi:

IPC_NOWAIT - powoduje, że operacja wykonywana jest bez blokowania procesu,

SEM_UNDO - powoduje, że operacja zostanie odwrócona, jeśli proces się zakończy.

Funkcja **semctl**() umożliwia wykonywanie różnorodnych operacji sterujących na zbiorze semaforów obejmujących m.in. usuwanie całego zbioru, pobieranie informacji o semaforach oraz ustawianie ich wartości.

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

gdzie:

semid - identyfikator zbioru semaforów,

semnum - indeks semafora w tablicy,

scmd - operacja sterująca,

arg - unia **semun** zawierająca różne typy danych dla różnych operacji sterujących.

Składniki unii **semun** są następujące:

```
union semun {
        int val;                    - wartość dla SETVAL
        struct semid_ds *buf;       - bufor dla IPC_STAT i IPC_SET
        unsigned short int *array;  - tablica dla GETALL i SETALL
        struct seminfo *__buf;      - bufor dla IPC_INFO
};
```

Funkcja **semctl**() oferuje następujące operacje sterujące:

IPC_STAT - zapis struktury **semid_ds** do bufora **buf**,

IPC_SET - ustawienie w strukturze **ipc_perm** maski praw dostępu do zbioru semaforów,

IPC_RMID - usunięcie zbioru semaforów,

GETALL - pobranie wartości wszystkich semaforów ze zbioru do tablicy **array**,

GETVAL - pobranie wartości pojedynczego semafora,

GETNCNT - pobranie liczby procesów oczekujących na zasoby,

GETPID - pobranie identyfikatora PID procesy, który ostatni wykonał ostatnią operację na zbiorze semaforów,

GETZCNT - pobranie liczby procesów oczekujących na zerową wartość semafora,

SETALL - ustawienie wartości wszystkich semaforów ze zbioru na podstawie tablicy **array**,

SETVAL - ustawienie wartości pojedynczego semafora na podstawie wartości zmiennej **val**.

Korzystając z omówionych funkcji systemowych można łatwo zrealizować podstawowe operacje na semaforach: zainicjowanie wartości początkowej, zajęcie i zwolnienie semafora.

Ustawienie wartości początkowej semafora

```
union semun semarg;
semarg.val = 1;
semctl(semid, 0, SETVAL, semarg);
```

Zajęcie semafora

```
struct sembuf sem_lock = {0, -1, 0};
if((semop(semid, &sem_lock, 1)) == -1){
      fprintf(stderr, "Lock failed\n");
}
```

Zwolnienie semafora

```
struct sembuf sem_unlock = {0, 1, 0};
if((semop(semid, &sem_unlock, 1)) == -1){
      fprintf(stderr, ”Unlock failed\n");
}
```

Operacje te najlepiej zdefiniować jako funkcje we własnym programie. Przykład zastosowania semaforów IPC do synchronizacji dostępu do sekcji krytycznej zamieszczono poniżej.

Przykład

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/sem.h>
#define SEGSIZE 80

int main(int argc, char *argv[]) {
      key_t key1, key2;
      int   shmid, semid, semval, i;
      char *adres, *adres2;
      union semun {
            int val;                    /* wartość dla SETVAL */
            struct semid_ds *buf;       /* bufor dla IPC_STAT i IPC_SET */
            unsigned short *array;      /* tablica dla GETALL i SETALL */
                                        /* Część specyficzna dla Linuksa:*/
            struct seminfo *__buf;      /* bufor dla IPC_INFO */
      } semarg;

      struct sembuf sem_lock = {0, -1, 0};
      struct sembuf sem_unlock = {0, 1, 0};

      if (argc < 2) {
            printf("Poprawne wywolanie: %s wiadomosc ...\n", argv[0]);
            exit(1);
      }

      key1 = ftok(".", 'p');
      if ((shmid = shmget(key1, SEGSIZE, IPC_CREAT|0666)) == -1) {
            perror("shmget");
                        exit(1);
      }

      key2 = ftok(".", 's');
      if ((semid = semget(key2, 1, IPC_CREAT|IPC_EXCL|0666)) == -1) {
            printf("Semafor juz istnieje. Otwieram ...\n");
            if ((semid = semget(key2, 1, 0666)) == -1) {
                  perror("semget");
                        exit(1);
```

```
            }
            if ((semval = semctl(semid, 0, GETVAL, 0)) == -1) {
                  perror("semctl");
                  exit(1);
            }
            printf("Wartosc semafora S=%d\n", semval);
      }
      else {
            semarg.val = 1;
            semctl(semid, 0, SETVAL, semarg);
            printf("Inicjowanie semafora S=1\n");
      }
      if ((adres = shmat(shmid, 0, 0)) == -1) {
            perror("shmat");
            exit(1);
      }

      adres2 = adres;

      if ((semop(semid, &sem_lock, 1)) == -1)
            fprintf(stderr, "Lock failed\n");
      printf("Semafor zajety\n");

      for (i=1; i<argc; i++) {
            strcpy(adres2, argv[i]);
            sleep(3);
            adres2 = adres2 + strlen(argv[i]);
      }

      if ((semop(semid, &sem_unlock, 1)) == -1)
            fprintf(stderr, "Unlock failed\n");
      printf("Semafor zwolniony\n");

      if (shmdt(adres) == -1) {
            perror("shmdt");
            exit(1);
      }
}
```

13.7. Semafory POSIX

Wielowartościowe semafory POSIX pełnią rolę liczników zasobów współdzielonych przez wątki lub procesy (stąd inna nazwa: semafory liczące). Interfejs IPC POSIX udostępnia dwa typy semaforów liczących, różniące się sposobem tworzenia i inicjowania: semafory nienazwane i semafory nazwane.

Semafory liczące POSIX (nienazwane)

Semafor

```
int sem_init(sem_t *sem, int pshared, unsigned int value);

int sem_destroy(sem_t * sem);
```

Funkcja **sem_init**() inicjuje semafor **sem** jako obiekt typu sem_t* z wartością początkową **value**. Jeżeli argument **pshared = 0** , to semafor może być współdzielony tylko przez wątki jednego procesu. Przy niezerowej wartości **pshared** semafor może być współdzielony przez różne procesy.

Funkcja **sem_destroy**() usuwa wskazany semafor, nie zwalnia jednak przydzielonej mu pamięci.

Semafory nazwane

Funkcja **sem_open**() tworzy nowy semafor i ustanawia połączenie między semaforem a procesem/wątkiem.

```
sem_t *sem_open(const char *name, int  oflag, /*  unsigned long mode,
unsigned int value */ ...);
```

gdzie:

name - nazwa obiektu,

oflag - flagi,

mode - maska praw dostępu,

value - wartość początkowa semafora.

Funkcja zwraca identyfikator semafora związanego z podaną nazwą **name**. Szczegółowy sposób działania wynika z flag ustawionych w argumencie **oflag**:

0 - funkcja ustanawia połączenie między istniejącym semaforem a procesem lub zwraca błąd, jeśli semafor nie istnieje,

IPC_CREAT - funkcja tworzy nowy semafor lub ustanawia połączenie między istniejącym semaforem a procesem,

IPC_EXCL | IPC_CREAT - funkcja tworzy nowy semafor i ustanawia połączenie lub zwraca błąd, jeśli semafor już istnieje.

Tworzenie semafora wymaga podania dwóch dodatkowych argumentów: maski praw dostępu **mode** i wartości początkowej **value**.

Operacje na semaforach

Niezależnie od sposobu utworzenia, poniższe funkcje umożliwiają podstawowe operacje na semaforach.

```
int sem_wait(sem_t * sem);

int sem_trywait(sem_t * sem);

int sem_post(sem_t * sem);

int sem_getvalue(sem_t * sem, int * sval);
```

Funkcje te wykonują:

- zwalnianie semafora, czyli zwiększanie wartości semafora o 1 - **sem_post**(),
- zajmowanie semafora, czyli oczekiwanie na niezerową wartość semafora i zmniejszanie tej wartości o 1
    - z blokowaniem wątku - **sem_wait**(),
    - bez blokowania wątku - **sem_trywait**(),
- pobieranie wartości semafora - **sem_gatvalue**().

13.8. Inne mechanizmy synchronizacji wątków POSIX

Blokady wyłączności dostępu - mutexy

Mutex zapewnia wzajemne wyłączanie wątków w dostępie do sekcji krytycznej. Mutex jest odpowiednikiem semafora binarnego dla tradycyjnych procesów. Poniższy zestaw funkcji umożliwia inicjowanie, zajmowanie i zwalnianie oraz usuwanie mutexów.

Funkcja **pthread_mutex_init()** inicjuje nowy mutex z podanym zestawem atrybutów i zwraca jego identyfikator.

```
int pthread_mutex_init(pthread_mutex_t *mutex, const
pthread_mutexattr_t *mutexattr);
```

Przy **mutexattr=NULL** stosowane są atrybuty domyślne, przedstawione w tablicy 13.3.

Tablica 13.3 Wybrane atrybuty mutexu

Atrybut	Wartości	Znaczenie
pshared	PTHREAD_PROCESS_PRIVATE (domyślnie)	mutex lokalny - może być wykorzystywany tylko przez wątki jednego procesu
	PTHREAD_PROCESS_SHARED	mutex globalny - może być wykorzystywany przez wszystkie wątki, które mają dostęp do pamięci (wątki różnych procesów)
type	PTHREAD_MUTEX_NORMAL	nie wykrywa zakleszczenia przy próbie powtórnego zajęcia mutexu bez zwolnienia
	PTHREAD_MUTEX_ERRORCHECK	wykrywa zakleszczenie i zwraca błąd
	PTHREAD_MUTEX_RECURSIVE	pozwala ponownie zająć mutex bez zwolnienia
	PTHREAD_MUTEX_DEFAULT	

Operacja zajmowania mutexu może być blokująca (funkcja **pthread_lock**()) lub nieblokująca (funkcja **pthread_trylock**()).

```
int pthread_mutex_lock(pthread_mutex_t *mutex);

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Funkcja **pthread_mutex_lock()** zajmuje mutex lub blokuje wołający wątek do momentu zwolnienia mutexu. Jest to zatem operacja blokująca. Funkcja **pthread_mutex_trylock()** zajmuje mutex lub zwraca błąd, jeśli mutex jest już zajęty. Operacja staje się nieblokująca. Wątek, który zajmuje mutex, staje się jego właścicielem i tylko właściciel może zajęty mutex zwolnić.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Zwolniony mutex można usunąć funkcją **pthread_mutex_destroy()**. Dla zajętego mutexu wynik operacji jest nieokreślony.

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

Zmienne warunkowe

Zmienne warunkowe umożliwiają wątkom oczekiwanie na spełnienie określonych warunków pod ochroną mutexu. Wykorzystywane są do synchronizacji zdarzeń, które współdzielą dostęp do mutexu, ale nie koniecznie do danych.

Po zaalokowaniu pamięci (statycznie lub dynamicznie) można zainicjować zmienną warunkową za pomocą funkcji pthread_cond_init().

```
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t
*cond_attr);
```

Użycie zmiennej warunkowej musi być zawsze otoczone mutexem (zajęcie mutexu przed i zwolnienie po). Odpowiedni fragment kodu powinien wyglądać następująco:

```
pthread_mutex_lock(…);
while(warunek_nie_spełniony)
     pthread_cond_wait(…);
pthread_mutex_unlock(…);
```

Funkcja **pthread_cond_wait**() powoduje czasowe zwolnienie mutexu i wstrzymanie wątku w oczekiwaniu na zasygnalizowanie spełnienia warunku. Obydwie operacje wykonywane są niepodzielnie. Gdy warunek zostanie spełniony i zasygnalizowany, mutex jest ponownie zajmowany a wątek wznawiany. Funkcja **pthread_cond_timedwait**() wprowadza dodatkowo ograniczony czas oczekiwania na zasygnalizowanie spełnienia warunku.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);

int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t
*mutex, const struct timespec *abstime);
```

Funkcja **pthread_cond_signal**() powoduje wznowienie jednego z wątków, oczekujących na danej zmiennej warunkowej. Funkcja **pthread_cond_broadcast**() wznawia wszystkie wątki oczekujące na danej zmiennej.

```
int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

Funkcja **pthread_cond_destroy**() usuwa zmienną warunkową, ale nie zwalnia zaalokowanej pamięci.

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Bibliografia

[1] Silberschatz A., Galvin P.B.: Podstawy systemów operacyjnych, WNT 2000

[2] Johnson M.K., Troan E.W.: Oprogramowanie użytkowe w systemie Linux, WNT 2000 (rozdziały: 10.4, 10.6)

[3] Rochkind M.J.: Programowanie w systemie UNIX dla zaawansowanych, WNT 2007 (rozdziały: 5.17, 7.8-7.10)

Słownik

Termin	Objaśnienie
głodzenie procesu	nieskończone blokowanie procesu przez inne procesy; może polegać na tym, że tylko wybrane procesy czekają w nieskończoność pod semaforem (na przykład z powodu niskiego priorytetu)
mutex	blokada wyłączności dostępu (ang. ***mut****ual* ***ex****clusion lock*), mechanizm synchronizacji wątków
problem sekcji krytyczej	problem zapewnienia wzajemnego wykluczania (wzajemnego wyłączania) procesów w dostępie do niepodzielnego zasobu
sekcja krytyczna	fragment kodu procesu wymagający wyłącznego dostępu do określonych zasobów np. plików lub wspólnych struktur danych
sekcja wejściowa	fragment kodu, w którym proces oczekuje na pozwolenie wejścia do sekcji krytycznej
sekcja wyjściowa	fragment kodu, w którym proces sygnalizuje wyjście z sekcji krytycznej
semafor	obiekt (zwykle liczba całkowita), któremu można nadać wartość początkową i który dostępny jest tylko za pomocą dwóch niepodzielnych operacji: **czekaj** i **sygnalizuj**
wirująca blokada	proces lub wątek sprawdza w pętli jakiś warunek (np. stan semafora), czyli wiruje w niewielkim fragmencie programu
zakleszczenie	zbiór współpracujących procesów pozostaje w stanie zakleszczenia, jeżeli każdy proces przetrzymuje jakiś zasób i jednocześnie oczekuje na przydział innego zasobu przetrzymywanego przez inny proces ze zbioru

Zadania do wykładu 13

Zadanie 1

Zmodyfikować program z zadania 2 do wykładu 12. Zrealizować schemat synchronizacji dostępu do tablicy w pamięci dzielonej, odpowiadający pierwszemu problemowi czytelników i pisarzy.