Programowanie równoległe w POSIX API

PRiR, wykład 3

Plan

- Ogólnie o wsparciu PR
- Co to jest POSIX
- POSIX API dla PR
  - Procesy
  - Wątki
  - Komunikacja
- Przykład

Narzędzia wspierające PR

- System operacyjny
  - efektywne uruchamianie zadań
- Języki wysokiego poziomu
  - jasny sposób wyrażania równoległości
- Kompilatory
  - automatyczne zrównoleglanie (dyrektywy)
- Biblioteki procedur numerycznych
  - wbudowana równoległość
- Biblioteki procedur do sterowania zadaniami
  - efektywna implementacja algorytmu równoległego
- Narzędzia do wspomagania programowania
  - projektowanie, uruchamianie, profilowanie

Niezbędne mechanizmy

- Systemy SM: kontrola wspólnej pamięci
  - Synchronizacja operacji dostępu do pamięci przez wiele procesów
- Systemy DM: komunikacja
  - Mechanizmy służące do przesyłania komunikatów
- W obu wypadkach: synchronizacja: bariery

Osiąganie maksymalnej wydajności zależy od:

- możliwie rzadkiego używania barier,
- minimalizacji czasu trwania operacji synchronizacji,
- minimalizacji czasu trwania operacji komunikacji.

Wsparcie PR dla maszyn SM

- Wątek (*thread*) – instancja procedury (metody), posiadająca prywatne dane lokalne oraz mająca dostęp do danych globalnych zgodnie z regułami zasięgu dla danego języka np. przez dziedziczenie przestrzeni adresowej procesu.
  - Tworzenie wątków
  - Niszczenie wątków
  - Identyfikowanie się wątków
  - Ustalanie punktów synchronizacji wątków współbieżnych

Wsparcie PR dla maszyn SM

- Zamek (*mutex lock*) – służy do ochrony fragmentów kodu (tzw. sekcji krytycznych) przed wejściem do nich więcej niż jednego wątku;
  - **lock_init** – inicjalizacja zamka,
  - **lock_on** – zamknięcie zamka,
  - **lock_off** – otwarcie zamka.

Zamki - realizacja

- Hardware
  - 1 procesor: blokada przerwań;
  - wiele procesorów: flaga test-and-set
- Software
  - Algorytmy: Dekker, Lamport, Peterson,…
  - Semafory
  - Monitory
  - Krotki

Zamki, cd

- Zamek typu *jeden pisarz – wielu czytelników* (*readers/writer lock*) – służy do zamknięcia sekcji krytycznej jedynie wtedy, gdy jeden z wątków zmienia dane globalne;
  - w pozostałych okresach możliwe jest jednoczesne czytanie danych przez wiele wątków.

Wsparcie PR dla maszyn SM

- Semafor – mechanizm niższego poziomu wykorzystujący in- i dekrementację pewnej zmiennej *s* (całkowitej, nieujemnej).
  - *wait(s)*: if *s>0* then *s:=s-1* else *wstrzymaj wątek* endif
  - *signal(s)*: if {*wątek W wstrzymany na s*}
    then *przywróć wątek W* else *s:=s+1* endif
  - Dla wartości początkowej *s=1* otrzymujemy semafor binarny (zamek).

Wsparcie PR dla maszyn SM

- Monitor – obiekt łączący dane globalne, zmieniane tylko w sekcji krytycznej, oraz operacje na nich z mechanizmem synchronizacji.
- Bariera – zapewnia dojście wszystkich wątków do punktu jej ustawienia, zanim którykolwiek będzie mógł pójść dalej:
  - **barrier_init(strength)** – inicjalizacja, liczba wątków;
  - **barrier** – ustawienie bariery, punkt synchronizacji.

Wsparcie PR dla maszyn DM

- Część wymagań analogiczna do SM:
  - Tworzenie procesów
  - Niszczenie procesów
  - Identyfikowanie się procesów (SPMD)
  - Komunikacja między procesami
    - wysyłanie komunikatów do jednego / wielu
    - odbiór komunikatów od jednego / wielu

Wsparcie PR dla maszyn DM

- Komunikacja synchroniczna – proces wysyłający jest wstrzymywany dopóki nie będzie gotowy odbierający i na odwrót.
- Komunikacja asynchroniczna – procesy nie są wstrzymywane; buforowanie komunikatów.
- Wysyłanie/obieranie blokujące – następna instrukcja w procesie będzie wykonana dopiero po zakończeniu operacji komunikacji.
- Wysyłanie/odbieranie nieblokujące – przejście do następnej instrukcji zaraz po zainicjowaniu operacji komunikacji.

Wsparcie PR dla maszyn DM

- Bariera – podobnie jak w systemach SM
- Brak sekcji krytycznych – zamki zbędne
- Pożądana jest symulacja środowiska pamięci wspólnej na maszynie DM:
  - *data parallelism* – High Performance Fortran,
  - przestrzeń krotek (rekordów z globalnej bazy danych) – Linda.
- Zdalne wywoływanie procedur (RPC)

Elementy wsparcia specyficzne dla klastrów

- Mechanizmy dodatkowe w stosunku do maszyn DM:
  - Umieszczanie i uruchamianie procesu (programu) na zdalnym komputerze;
  - Konwersja typów danych między różnymi architekturami.
- Architektury:
  - *Master* (koordynator) – *slaves* (wykonawcy),
  - *Peer-to-peer* – węzły równouprawnione (asynchroniczne).

Elementy wsparcia specyficzne dla RPC

- Zrównoleglanie zdalnych procedur:
  - Procedury zdalne są uruchamiane z niezależnych wątków i komunikują się między sobą za pośrednictwem komunikatów.
  - Procedury zdalne uruchamiane są sekwencyjnie w sposób nie blokujący; dane wymieniają za pomocą komunikatów.

POSIX

- POSIX – Portable Operating System Interface
- Implementowany przez systemy uniksopodobne (linux, FreeBSD)
- Drogi (w odp. Powstał Single UNIX Specification)
- P1003.1b,c,d (POSIX.4) – wsparcie PR

Uniksowe wsparcie PR na maszynach wieloprocesorowych z pamięcią współdzieloną

- Procesy
- Identyfikatory i uchwyty
- Wątki
- Komunikacja i synchronizacja między procesami
- Synchronizacja między wątkami jednego procesu
- Przykłady – IPC, wątki

Tworzenie procesów - fork

- System Unix, funkcje w języku C
- Tworzenie procesu: fork

```
int fork()
```

  - powstaje proces potomny (kopia),
  - oba procesy wykonują ten sam kod (wychodzą z fork), operują na kopiach tych samych danych,
  - w procesie macierzystym zwracany jest identyfikator procesu potomnego, a w potomnym zero.

Różne zadania - execl

```
int execl(char *path, char *arg0, ..., char *argn, NULL)
```

- Inicjowanie procesów wykonujących różne zadania:
  - funkcja execl wywoływana przez proces potomny (po fork),
  - proces potomny rozpoczyna wykonywanie programu o nazwie arg0 z katalogu path,
  - pozostałe argumenty execl określają argumenty wywoływanego programu, ostatni powinien mieć wartość NULL,
  - po uruchomieniu programu proces zwalnia dotychczas używane zasoby.

Przykład – powołanie procesu

```
/* proces.c */

int id, status;

switch(id=fork())
{
  case 0:              /* Proces potomny                    */
     execl("./path", "test", "arg1", NULL);
  case -1:
     perror("powielenie procesu niemozliwe");
  default:             /* Proces macierzysty                */

     wait(&status); /* oczekuje zakonczenia procesu  */
}
```

Przykład – powołanie wielu procesów

```
/* procesy.c */

int i, p = 10;

for(i = 0; i < p; i++)
  if(fork() == 0)
     execl("./path", "test", "arg1", NULL);

for(i = 0; i < p; i++)
  wait(&status);
```

Zakończenie procesu - exit

- Funkcja exit kończy działanie procesu:
  - zwracany status zakończenia wynosi 0xFF & status,

```
int exit(int status)
```

  - status zakończenia może odczytać tylko proces macierzysty funkcją wait, która pobiera pierwszy, dostępny status zakończenia procesu potomnego i zapisuje pod adresem status, zwraca identyfikator (PID) zakończonego procesu.

```
int wait(int *status)
```

Zakończenie procesu - kill

- Zakończenie procesu przez inny proces:
  - funkcja kill wysyła sygnał sig do procesu lub grupy procesów,

```
int kill(int pid, int sig)
```

  - jeśli pid ≥ 0 to sygnał jest wysyłany do procesu o tym identyfikatorze, a jeśli pid < 0 to do grupy procesów o identyfikatorze |pid|,

Obsługa sygnałów - SysV

- procedura obsługi sygnału może być wskazana funkcją signal (fun – nowa procedura obsługi sygnału sig),
- funkcja signal zwraca wskazanie na poprzednią procedurę obsługi.

```
int(*signal(int sig, int (*fun)()))()
```

Obsługa POSIX

- Obsługa zbiorów sygnałów

int sigemptyset(sigset_t *zbior) --- zeruje zbiór sygnałów wskazywany przez parametr zbior.

int sigfillset(sigset_t *zbior) --- dodaje do zbioru wszystkie dostępne sygnały.

int sigaddset(sigset_t *zbior, int sygnal) --- dodaje sygnał do zbioru.

int sigdelset(sigset_t *zbior, int sygnal) --- usuwa sygnal ze zbioru.

int sigismember(const sigset_t *zbior, int sygnal) --- przekazuje w wyniku wartość niezerową, jeżeli sygnał jest w zbiorze, a w przeciwnym wypadku 0.

POSIX, cd

Rejestrowanie funkcji obsługi

```
struct sigaction {
   sighandler_t sa_handler;
   sigset_t sa_mask;
   unsigned long sa_flags;
   void (*sa_restorer)(void);  /*not used*/
};
```

POSIX, cd

Funkcja obsługi

- sighandler_t
  - void handler(int numer_sygnalu);
  - SIG_IGN
  - SIG_DFL

POSIX, cd

Flagi

* SA_NOCLDSTOP - dla sygnału SIGCHLD – sygnał jest generowany tylko wtedy, gdy proces zakończył działanie, a wstrzymanie procesów potomnych nie powoduje wysłania żadnego sygnału.
* SA_NOMASK - w trakcie wykonywania funkcji obsługi sygnału, sygnał nie jest automatycznie blokowany.
* SA_ONESHOT - kiedy sygnał jest wysłany, obsługa sygnału jest zerowana do SIG_DFL.
* SA_RESTART - kiedy sygnał jest wysłany do procesu, w czasie, gdy ten wykonuje wolną funkcję systemową, funkcja systemowa jest wznawiana po powrocie z funkcji obsługi sygnału.

POSIX, cd

Blokowanie sygnałów

int sigprocmask(int co, const sigset_t * zbior, sigset_t * stary_zbior);

sigprocmask(SIG_BLOCK, NULL, &aktualnaMaska);

* SIG_BLOCK --- sygnały ze zbioru są dodawane do aktualnej maski sygnałów.
* SIG_UNBLOCK --- sygnały ze zbioru są usuwane z aktualnej maski sygnałów.
* SIG_SETMASK --- tylko sygnały należące do zbioru są blokowane.

int sigpending(sigset_t * zbior);

Komunikacja

- Identyfikator – unikalna liczba mająca znaczenie nazwy przydzielanej przez system podczas tworzenia obiektu (strumienia).
- Uchwyt (*handle*) – liczba identyfikująca obiekt (strumień), używana przez wszystkie funkcje operujące na obiekcie;
  - ma znaczenie tylko w obrębie procesu,
  - uchwyty mogą być dziedziczone przez procesy potomne

Operacje na uchwytach

- pipe – tworzenie strumienia komunikacyjnego int pipe(int fd[2]);
- dup – tworzenie nowego uchwytu do tego samego obiektu co fd i zwrócenie uchwytu: `int dup(int fd)`
  - nie pozwala na tworzenie kopii uchwytu z innego procesu, ani dla innego procesu (i zwraca tę wartość lub -1). `int dup2(int fs, int fd2)`
  - dup2 – kopii uchwytu fs nadaje wartość fd2 (i zwraca tę wartość lub -1).
- execl – także: zamykanie uchwytów w procesie potomnym z ustawionym atrybutem zamknięcia.
- fcntl – zmiana wartości atrybutu zamknięcia:

```
int fcntl(int des,
          int cmd,
          int arg)
```

  - des – modyfikowany uchwyt, cmd – stała F_SETFD, arg – nowa wartość atrybutu FD_CLOEXEC (0 lub 1) ;
  - odczyt atrybutu: cmd – stała F_GETFD, zwraca w/w atrybut.
- close – zamknięcie uchwytu. `int close(int des)`

Wady obliczeń równoległych w oparciu o procesy

- Udział jądra systemu w tworzeniu i przełączaniu procesów – opóźnienia.
- Powielanie wszystkich danych procesu (fork):
  - segment danych rodzica nie musi być wykorzystywany przez potomka.
- Procesy mają niezależne przestrzenie adresowe:
  - dostęp do segmentu pamięci współdzielonej wymaga wywołania (na początku) dwu funkcji systemowych, odwzorowujących ten segment na przestrzeń adresową procesu.

Wątki

- Wątek (*thread*) nie posiada własnego segmentu danych.
- Wszystkie wątki działają w obrębie pewnego procesu i mają dostęp do jego segmentu danych i sterty (*heap*).
- Każdy wątek wykonuje określony fragment kodu procesu:
  - posiada własny stos (*stack*) i zmienne lokalne.
- Wszystkie uchwyty stosowane w procesie mogą być używane przez wątki bez ponownego otwierania.
- Proces po utworzeniu ma jeden wątek, główny (wyjątki: wersje fork kopiujące wątki):
  - wątek główny powołuje inne wątki, a one następne.
- Standard POSIX

Tworzenie wątku

- Funkcja – pthread_create
  - utworzenie wątku wywołującego funkcję o adresie start_func i przekazanie jej argumentu arg,
  - argument attr określa atrybuty przekazane nowemu wątkowi (jeśli NULL to domyślne),
  - Jeśli wątek zostanie utworzony jego identyfikator jest wstawiany pod adres wskazany przez thread a funkcja zwraca 0, w przeciwnym wypadku zwracany jest kod błędu.

```
int  pthread_create(pthread_t *thread, pthread_attr_t
    *attr, void * (*start_func)(void *), void *arg);
```

Zakończenie wątku

- Wątek może się zakończyć powrotem ze swojej funkcji.
- Funkcja pthread_exit:

```
void pthread_exit(void *status)
```

  - wątek może ją wywołać w celu zakończenia swojego działania,
  - wartość argumentu status może być odczytana przez inny wątek procesu.

Zakończenie wątku

```
int pthread_join(pthread_t target_thread, void **thread_return)
```

- Funkcja pthread_join:
  - zawiesza wykonywanie bieżącego wątku w oczekiwaniu na zakończenie wątku target_thread,
  - jeśli thread_return ≠ NULL to pod ten adres wpisywany jest status zakończonego wątku (lub jeśli został przerwany - PTHREAD_CANCELED), identyfikator wątku jest zwalniany a funkcja zwraca zero lub kod błędu,
  - wątek target_thread musiał być w stanie *joinable*,
  - dopiero wywołanie funkcji pthread_join zwalnia zasoby zakończonego wątku (unikanie wycieku pamięci – *memory leak*).

Zakończenie wątku

```
int pthread_kill(pthread_t thread, int sig)
```

- Zakończenie wątku przez wywołanie w innym wątku funkcji pthread_kill:
  - wysłanie sygnału sig do wątku o identyfikatorze thread,
  - obsługa sygnału następuje wg działania funkcji kill,
  - sposób obsługi sygnałów jest wspólny dla wszystkich wątków.

Zmienne lokalne wątku

- Umożliwiają posługiwanie się wspólnym identyfikatorem (globalnym) zawierającym różne wartości w różnych wątkach.
- Unikanie rozrostu liczby argumentów.
- Zmienna lokalna (*thread local storage*):
  - „wektor” o rozmiarze równym ilości wątków,
  - każdy wątek operuje na jednym elemencie,
  - operacje wymagają jednego argumentu – nazwy zmiennej lokalnej,
  - Proces główny tworzy obiekt, a jego identyfikator zapisuje jako zmienną globalną, wątki inicjują tą zmienną swoje pozycje w zmiennej lokalnej.

Tworzenie zmiennej lokalnej

```
int pthread_key_create(pthread_key_t *key,
    void (*destr_function) (void *))
```

- Funkcja pthread_key_create:
  - pod adres key wpisuje identyfikator utworzonej zmiennej,
  - argument destr_function powinien wskazywać funkcję, która ma być wywołana w chwili zakończenia wątku (lub mieć wartość NULL),
  - argumentem funkcji destr_function będzie aktualna wartość pola przydzielonego niszczonemu wątkowi,
  - funkcja zwraca zero lub kod błędu.

Nadawanie i odczytywanie wartości zmiennej lokalnej

```
int pthread_setspecific(pthread_key_t key,
    const void  *pointer);
```

- Funkcja pthread_setspecific:
  - argument key – identyfikator zmiennej lokalnej,
  - argument pointer wskazuje na nową wartość pola zmiennej dla danego wątku,
  - funkcja zwraca zero lub kod błędu.

```
void * pthread_getspecific(pthread_key_t key)
```

- Funkcja pthread_getspecific:
  - zwraca wartość wskazywaną przez zmienną lokalną key, lub NULL (w razie błędu).

Deklarowanie ilości wątków

```
int pthread_setconcurrency(int new_level)
```

- Funkcja pthread_setconcurrency pozwala poinformować implementację wątków o maksymalnej ilości new_level aktywnych wątków.
- Bardziej oszczędne gospodarowanie zasobami systemowymi.
- Reakcja implementacji nie jest określona.
- Funkcja zwraca 0 (sukces) lub kod błędu.

Mechanizmy komunikacji i synchronizacji między procesami

- Wymiana danych bez synchronizacji:
  - wspólna pamięć.
- Synchronizacja:
  - semafory,
  - obiekty *mutex* (zamki).
- Komunikacja, powodująca także synchronizację:
  - potoki,
  - kolejki komunikatów.

Wspólna pamięć

- Obszar pamięci dostępny dla wielu procesów.
- Utworzenie segmentu wspólnej pamięci w systemie;
  - zainicjowanie pamięci fizycznej.
- Odwzorowanie segmentu wspólnego w przestrzeń adresową danego procesu:
  - bądź przez podanie jawnie adresu początku obszaru wspólnego (może się nie powieść),
  - bądź przez pozwolenie systemowi na przydzielenie adresu obszaru wspólnego (jako argument określający adres należy podać 0).

Tworzenie segmentu wspólnej pamięci

- Funkcja shmget: `int shmget(key_t key, int size, int shmflg)`
  - zwraca identyfikator wspólnego segmentu lub -1 (błąd),
  - argument key jest liczbą unikalną skojarzoną z segmentem (lub IPC_PRIVATE),
  - argument size określa rozmiar tworzonego segmentu (w jednostkach PAGE_SIZE),
  - argument shmflg określa flagi:
    - IPC_CREAT – jeśli segment o danym kluczu key nie istnieje należy go utworzyć,
    - IPC_EXCL – w połączeniu z IPC_CREAT gwarantuje błąd jeśli segment już istnieje,
    - mode_flags – 9 najmniej znaczących bitów określa uprawnienia użytkownika, grupy i innych do segmentu.

Podłączenie segmentu wspólnego do procesu

```
void *shmat (int shmid,
     const void *shmaddr,
     int shmflg)
```

- Funkcja shmat:
  - shmid – identyfikator segmentu pamięci wspólnej;
  - shmaddr – żądany adres początkowy,
    - jeśli 0 to alokowany jest obszar w zakresie 1-1.5G od góry,
    - Jeśli ≠0 to alokowany jest obszar od tego adresu;
  - shmflg – dodatkowe flagi, m.in.:
    - SHM_RDONLY – segment dołączony tylko do odczytu,
  - Funkcja zwraca adres dołączonego segmentu lub -1.

Likwidowanie segmentu wspólnego

- Funkcja shmdt: `int shmdt (const void *shmaddr)`
  - odłącza segment o adresie shmaddr,
  - zwraca 0 (sukces) lub -1 (błąd).
- Funkcja shmctl - tutaj: `shmctl(shmid,IPC_RMID,NULL)`
  - usuwa wspólny segment pamięci shmid,
  - może być wywołana tylko w jednym procesie,
  - zwraca 0 (sukces) lub -1 (błąd).

Przykład

```
#define key 1001

 int   xsize,
       id_seg;
 float *y;

 xsize = sizeof(float)*300;
 id_seg = shmget(key, xsize, IPC_CREAT);
 y = (float *)shmat(id_seg, 0, 0)

 y[0]=1.0;
 y[1]=2.0;
```

Potoki anonimowe (nienazwane)

- Mechanizm przekazywania danych między spokrewnionymi procesami:
  - macierzystym a potomnym,
  - dwoma mającymi wspólnego przodka.
- Wspólny przodek tworzy potok anonimowy uzyskując dwa uchwyty (do pisania i czytania),
  - przekazuje te uchwyty procesom potomnym.

Tworzenie potoku anonimowego

- Funkcja pipe:

```
int pipe(int filedes[2])
```

  - uchwyt otwarty do czytania jest umieszczany w filedes[0], a uchwyt do pisania w filedes[1],
  - funkcja zwraca 0 (sukces) lub -1 (błąd).
  - uchwyty mogą być dziedziczone przez procesy potomne i służyć do przekazywania informacji między tymi procesami.

Pisanie do i czytanie z potoku

```
ssize_t write(int fd,
    const void *buf,
    size_t count)
```

- Pisanie – funkcja write:
  - zapisuje count bajtów z bufora wskazywanego przez buf do uchwytu fd,
  - funkcja zwraca count (sukces) lub -1 (błąd).

```
ssize_t read(int fd,
    void *buf,
    size_t count)
```

- Czytanie – funkcja read:
  - czyta do count bajtów z uchwytu fd do bufora buf,
  - funkcja zwraca liczbę przeczytanych bajtów (może być zero) (sukces) lub -1 (błąd).

Zmiana trybu dostępu

- Poprzez ustawienie atrybutu O_NDELAY za pomocą funkcji fcntl można zmienić zachowanie funkcji write i read:
  - podając jako fd uchwyt do potoku, jako cmd stałą F_SETFL, a jako arg liczbę z ustawionym bitem O_NDELAY; funkcja zwraca 0 (sukces) lub -1 (błąd).
  - funkcja write nigdy nie będzie czekać (zapisze tyle danych ile jest miejsca i zwróci tę liczbę),
  - funkcja read w przypadku braku danych zwróci natychmiast zero.

```
int fcntl(int fd, int cmd,
          long arg)
```

Potoki nazwane

- Named pipes – (FIFOs)
- Potoki nazwane są identyfikowane przez nazwę – można za ich pomocą tworzyć połączenia między procesami.
- Tworzenie potoku – mkfifo (lub mknod):
  - pathname - ścieżka dostępu do pliku,
  - mode – tryb dostępu (jak chmod, umask),
  - funkcja zwraca 0 (sukces) lub -1 (błąd).
- Usuwanie potoku – unlink:
  - funkcja zwraca 0 (sukces) lub -1 (błąd).

```
int mkfifo(const char *pathname,
           mode_t mode)
```

```
int unlink(const char *pathname)
```

Otwieranie potoku nazwanego

- Oba końce (procesy) muszą otworzyć potok aby go używać – funkcja open zwraca uchwyt:
  - pathname – ścieżka dostępu do pliku potoku,
  - flags – binarne flagi określające tryb dostępu np.:
    - O_RDONLY – otwarcie tylko do odczytu,
    - O_WRONLY – otwarcie tylko do zapisu,
    - O_NDELAY – otwarcie bez blokowania.
- Po otwarciu czytanie/pisanie realizują funkcje read i write.

```
int open(const char *pathname,
         int flags)
```

Kolejki komunikatów

- Komunikacja przez potoki ma charakter strumieniowy – nie określona struktura.
- Każdy komunikat może mieć inną liczbę danych (funkcja odczytująca poznaje tę liczbę przed odczytaniem komunikatu).
- Brak połączenia między nadawcą i odbiorcą;
  - w chwili wysłania komunikatu odbiorca może nie istnieć bądź być ich kilku,
  - komunikat przesyła się określając kolejkę.

Tworzenie kolejki komunikatów

- Funkcja msgget: `int msgget(key_t key, int msgflg)`
  - tworzy kolejkę o kluczu key i zwraca jej identyfikator,
  - argument msgflg określa flagi, analogicznie jak przy funkcji shmget:
    - IPC_CREAT – jeśli kolejka o danym kluczu key nie istnieje należy ją utworzyć,
    - IPC_EXCL – w połączeniu z IPC_CREAT gwarantuje błąd jeśli kolejka już istnieje,
    - mode_flags – 9 najmniej znaczących bitów określa uprawnienia użytkownika, grupy i innych do kolejki.

Wysyłanie komunikatów

```
int msgsnd(int msqid,
  struct msgbuf *buf,
          size_t size,
           int msgflg)
```

- Funkcja msgsnd:
  - wysyła komunikat o długości size (bez pola określającego typ) wskazywany przez buf do kolejki o identyfikatorze msqid,
  - deklaracja struktury msgbuf:
    - pole mtype – typ komunikatu,
    - pole mtext – treść (1 znak)

```
struct msgbuf {
  long mtype;     /* message type, must be > 0 */
  char mtext[1]; /* message data */
};
```

Dłuższe komunikaty

- W celu przesłania komunikatu dłuższego niż 1 znak należy zdefiniować własną strukturę:
  - wywołując msgsnd zrzutować wskazanie na strukturę mymsg na wskazanie na strukturę msgbuf;
  - argument size funkcji msgsnd określa długość komunikatu, nie wliczając pola określającego typ;
  - dla mymsg będzie to sizeof(struct mymsg) - sizeof(long);
  - flagi msgflg umożliwiają m.in. włączanie sygnalizacji przepełnienia kolejki (IPC_NOWAIT).

```
struct mymsg {
  long mtype;
  short mshort;
  char mchar[8];
};
```

Odbieranie komunikatów

```
ssize_t msgrcv (int msqid, struct msgbuf *buf,
    size_t size, long type, int msgflg)
```

- Funkcja msgrcv:
  - odczytuje komunikat typu z kolejki o identyfikatorze msqid do struktury typu msgbuf wskazywanej przez buf,
  - argument size określa maksymalny rozmiar komunikatu (pole mtext struktury buf),
  - argument type:
    - = 0 – czytany jest pierwszy komunikat (najstarszy),
    - < 0 – czytany jest komunikat, którego typ jest najmniejszy spośród spełniających ≤ |type|,
    - > 0 – czytany jest pierwszy komunikat o typie type.
  - flagi msgflg: **IPC_NOWAIT – błąd dla pustej kolejki**, **IPC_NOERROR – obcięcie za długiego komunikatu**.

Usunięcie kolejki komunikatów

- Funkcja msgctl:
  - pozwala m.in. na usunięcie kolejki,
  - w tym celu należy wysłać komendę IPC_RMID,
  - argument msqid określa identyfikator usuwanej kolejki:

```
msgctl(msqid, IPC_RMID, NULL)
```

Semafory

- Semafor to obiekt synchronizacji blokujący dostęp do pewnego zasobu.
  - Posiada licznik jednostek danego zasobu.
- Podstawowe operacje na semaforze:
  - zmniejszenie liczby jednostek zasobu:
    - w momencie próby zajęcia zasobów przez wątek,
    - jeśli liczba jednostek jest już za mała wątek czekać na jej odpowiedni wzrost;
  - zwiększenie liczby jednostek zasobu:
    - w momencie zwolnienia zasobów przez wątek.
- Licznik ma górne ograniczenie,
  - Semafor binarny (dopuszczalne wartości: 0 i 1).

Tworzenie semaforów

```
int semget(key_t key, int semcount, int flags)
```

- Funkcja semget:
  - tworzy grupę złożoną z semcount semaforów o indeksach od 0 do (semcount – 1),
  - grupie jest przypisany klucz key,
  - argument flags określa uprawnienia dostępu do grupy (jak dla funkcji shmget),
  - funkcja zwraca identyfikator grupy lub –1.

Operacje na semaforach

```
int semop(int semid, struct sembuf *semtab, unsigned count)
```

- Funkcja semop:
  - wykonuje count różnych operacji na semaforach z grupy o identyfikatorze semid,
  - parametr semtab wskazuje na tablicę struktur zawierających trzy liczby określające każdą operację,
  - funkcja zwraca 0 (sukces), -1 (błąd).

Struktura sembuf

```
struct sembuf
{
  short sem_num;
  short sem_op;
  short sem_flg;
}
```

- Struktura sembuf:
  - sem_num – numer semafora w grupie,
  - sem_op – rodzaj operacji:
    - >0 – zwiększenie licznika, <0 – zmniejszenie licznika (z ew. czekaniem), =0 – czekanie aż licznik osiągnie 0,
  - sem_flg – dodatkowe opcje operacji (flagi):
    - IPC_NOWAIT – operacja bez oczekiwania (jeśli byłoby wymagane zgłaszany jest błąd),
    - SEM_UNDO – operacja zostanie odwołana jeśli proces zostanie zamknięty (automatyczne zwalnianie semaforów).

Operacje na semaforach

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

- Funkcja semctl:
  - nadaje wartości licznikowi semafora,
  - odczytuje te wartości,
  - usuwa grupę semaforów,
  - semid – identyfikator grupy semaforów,
  - cmd – operacja, m.in.:
    - IPC_RMID – usunięcie grupy semaforów,
    - SETVAL – nadanie wartości licznika semafora semnum, pobranej z pola val unii arg,
    - GETVAL – zwrócenie wartości licznika semafora semnum.

```
union semun
{
    int val;
    struct semid_ds *buf;
    unsigned short int *array;
    struct seminfo *__buf;
}
```

Operacje na semaforach

- W wielu przypadkach wygodnie jest korzystać z funkcji zmieniającej tylko jeden element tablicy operacji.
  - Można zdefiniować funkcję:

```
void semcall(int sem_id, int op, int nr)
{
        struct sembuf buf;

        buf.sem_num = nr;
        buf.sem_op = op;
        buf.sem_flg = 0;
        semop(sem_id, &buf, 1);
}
```

Przykład

- Implementacja zamka, czyli ochrony sekcji krytycznej z wykorzystaniem funkcji semcall.

```
sem = semget(klucz, 1, IPC_CREAT);
semcall(sem, 1, nr);

semcall(sem, -1, nr); /* zajecie semafora     */

                      /* wykonanie operacji
                      na zasobie              */
semcall(sem, 1, nr);  /* zwolnienie semafora */
```

Mechanizmy synchronizacji między wątkami jednego procesu

- Ponieważ wątki mają dostęp do segmentu danych procesu i sterty nie ma potrzeby tworzenia dodatkowych mechanizmów komunikacji między wątkami.
- Zachodzi potrzeba synchronizacji wątków.

Obiekty typu *mutex*

- Obiekt typu *mutex* (*mutual exclusion*) jest zbliżony do semafora binarnego – pamięta dodatkowo wątek, który go zajął – i tylko ten wątek może go zwolnić.
- Pozwala blokować dostęp do współdzielonych struktur danych, tworzyć sekcje krytyczne i monitory.
- Może on być używane do synchronizacji wątków.

Tworzenie obiektów *mutex*

- Obiekt *mutex* można utworzyć przez statyczne zainicjowanie zmiennej typu *pthread_mutex_t* stałą:
  - **PTHREAD_MUTEX_INITIALIZER – "fast" mutex,**
  - **PTHREAD_RECURSIVE_MUTEX_INITIALIZER_NP – "recursive" mutex,**
  - **PTHREAD_ERRORCHECK_MUTEX_INITIALIZER_NP – "error checking" mutex.**
- Dynamiczne inicjowanie – funkcja pthread_mutex_init:
  - mutex – wskazanie na tworzony obiekt *mutex*,
  - mutexattr – wskazanie na atrybut obiektu *mutex* (lub NULL – ozn. domyślny typ "fast"),
  - funkcja zwraca zawsze 0.

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

Zamknięcie obiektu *mutex*

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

- Funkcja pthread_mutex_lock:
  - zamyka mutex przez wywołujący ją wątek,
  - jeśli mutex był już zamknięty przez ten sam wątek dojdzie do blokady (tryb domyślny),
  - próba zamknięcia wykonana przez inny wątek wstrzymuje go aż do zwolnienia zamka.
  - funkcja zwraca 0 (sukces) lub kod błędu.
  - wersja nie blokująca: pthread_mutex_trylock.

Otwarcie obiektu *mutex*

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

- Funkcja pthread_mutex_unlock:
  - otwiera zamek mutex (tryb domyślny "fast"),
  - zakłada się, że zamek był zamknięty przez ten sam wątek,
  - inne wątki także mogą otworzyć zamek (działanie niestandardowe LinuxThreads)
    - rezultat zależny od implementacji,
  - funkcja zwraca 0 lub kod błędu.

Usuwanie obiektu *mutex*

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

- Funkcja pthread_mutex_destroy:
  - usuwa wskazywany obiekt mutex,
  - zamek musi być otwarty przy jej wywołaniu,
  - funkcja zwalnia tylko zasoby związane z obiektem,
  - jeśli zmienna była alokowana dynamicznie należy ją dodatkowo usunąć,
  - funkcja zwraca 0 lub kod błędu.

Przykładowe zadanie obliczeniowe – IPC i wątki

- Rozwiązanie układu równań liniowych

$$Ax = b$$

dla danej macierzy A o wymiarach $n \times n$ i wektora $b \in \Re^n$.

- Jeśli A jest zdominowana diagonalnie (czyli $\forall i \; |a_{ii}| > \sum_{(j \neq i)} |a_{ij}|$), to rozwiązanie daje procedura iteracyjna:

$$x := x - D(Ax - b)$$

gdzie D jest taką macierzą diagonalną, że $d_{ii} = 1/a_{ii}$.

Dekompozycja

- Wektor $x \in \Re^n$ będzie podzielony na *p* rozłącznych części

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{bmatrix}$$

gdzie $x_i \in \Re^{n_i}$, $\sum^{p}_{(i=1)} n_i = n$.

- Niech *i*-ty procesor może zmieniać tylko podwektor x_i : $x_i = f_i(x_1,\ldots,x_{i-1},x_i,x_{i+1},\ldots,x_p)$

Implementacja asynchroniczna

- Jeden proces inicjujący obliczenia:
  - wczytanie danych,
  - dekompozycja problemu,
  - uruchomienie i inicjalizacja procesów obliczeniowych.
- Wiele procesów obliczeniowych wykonujących iteracje.

Realizacja algorytmu - IPC

- Komunikacja z użyciem pamięci dzielonej
- Synchronizacja za pomocą semaforów
- Proces inicjujący *mainIPC.c*:
  - alokacja A, b, xo – wartość pośrednia x, size – segment przechowujący rozmiar zadania, local_stop – tablica znaczników osiągnięcia lokalnego warunku stopu,
  - przekazanie procesom obliczeniowym jako argumenty wywołania: p – liczba procesów, n – numer procesu, beg, end – współrzędne fragmentu wektora, wyznaczane przez proces.
- Proces obliczeniowy *workIPC.c*:
  - dołącza się do pamięci wspólnej i wykonuje iteracje,
  - po każdej iteracji sprawdza warunek stopu (ustawia znacznik w local_stop jeśli $|| x_i^{(k)} - x_i^{(k+1)} || < \varepsilon$),
  - następnie sprawdza, czy pozostałe procesy również ustawiły znaczniki w local_stop – jeśli tak to kończy działanie.

Plik nagłówkowy defsIPC.h

Defsipc.h

Operacje semaforowe semIPC.c

Semipc.c

Proces główny mainIPC.c

Mainipc.c

Proces obliczeniowy workIPC.c

Workipc.c

Realizacja algorytmu – wątki

- Komunikacja z użyciem pamięci dzielonej
- Wykorzystane wątki LinuxThreads (POSIX threads – *pthreads*)
- Procedury inicjujące *mainW.c*
- Procedura wątku obliczeniowego *workW.c*

Plik nagłówkowy defsW.h

Defsw.h

Procedury inicjujące mainW.c

Mainw .c

Procedura obliczeniowa workW.c

Workw.c