ЦЕНТРАЛЬНОУКРАЇНСЬКИЙ НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ

Кафедра кібербезпеки та програмного забезпечення

Комп'ютерні системи

МЕТОДИЧНІ ПОРАДИ

до самостійної роботи студентів

спеціальності 123 – «Комп’ютерна інженерія»

Кропивницький - 2023 рік

Комп'ютерні системи : метод. поради до самостійної. роботи студентів спеціальності 123 – «Комп’ютерна інженерія» / [уклад. К. М. Марченко] ; М-во освіти і науки України, Центральноукраїн. нац. техн. ун-т, каф. кібербезпеки та програмного забезпечення. – Кропивницький : ЦНТУ, 2023. – 23 с.

Для студентів спеціальності 123 – "Комп’ютерна інженерія"при вивченні навчальної дисципліни “Комп'ютерні системи”. Визначено тематику самостійної роботи, подані короткі теоретичні відомості, приведені приклади та завдання.

Укладач:

Марченко Костянтин Миколайович - канд. техн. наук, доцент кафедри кібербезпеки та програмного забезпечення

Затверджено на засіданні кафедри КбПЗ, протокол №10 від 19.01.2022 р.

Самостійна робота №1

Паралельне програмування. Створення програмних кодів для паралельних алгоритмів з використанням расширення мови С ++

Розглянемо приклад перетворення послідовної програми в паралельну (завдання обчислення певного інтеграла від заданої функції (всі вхідні параметри задані константами)).

Для обчислення наближення до певного інтеграла від функції f по відрізку [a, b] використовуємо складову формулу трапецій:

$$\int_a^b f(x)dx \approx h(f(a)/2 + \sum_{j=1}^{n-1} f(a+jh) + f(b)/2),$$

Где $h = (b-a)/n$, а параметр *n* задает точность вычислений.

Спочатку файл integral.c з текстом послідовної програми, що обчислює певний інтеграл цим способом:

```
#include “integral.h”
/*интегрируемая функция*/
Double a (double x)
{
Return x
}
/*вычислить интеграл по отрезку [a,b] с число точек
разбиения n методом трапеций.  */
{
Double res; /*результат*/
Double h; /*шаг интегрирования*/
Int I;

H=(b-a)/n;
Res=0.5*(f(a)+f(b))*h;
For (i=1; i<n; i++)
  Res +=f(a+i*h)*h;
Return res;
}
```

Відповідний заголовки integral.h:

```
Double integrate (double a, double b, int n);
```

Файл sequential.c з текстом послідовної програми:#include <stdio.h>

```
#include “integral.h”

/ * Всі параметри для простоти задаються константами * /
```

```
Static double a = 0 .; / * Лівий кінець інтервалу * /
Static double b = 1 .; / * Правий кінець інтервалу * /
Static int n = 100000000; / * Число точок розбиття * /

Int main ()
{
Double total = 0 .; / * Результат: інтеграл * /
/ * Обчислити інтеграл * /
Total= integrate (a,b,n);
Printf (“integral from %if to %if = %.18if\n”, a, b, total);

Return 0;
}
```

Компіляція цих файлів:

```
Cc sequential.c integral.c - o sequential
```

і запуск

```
./sequential
```

Самостійна робота №2

Рішення задач з застосуванням синхронізації мови програмування: блоки / розблокування; критична секція, семафори.

Доступ процесів (завдань) до різних ресурсів (особливо розділяються) в багатозадачних системах вимагає синхронізації дій цих процесів (завдань). Способи здійснення поділяють на:

• безпечне взаємодія, коли обмін даними здійснюється за допомогою «об'єктів» взаємодії, що надаються системою; при цьому цілісність інформації та неподільність операцій з нею (тобто відсутність небажаного перемикання завдань) неявно забезпечуються системою; прикладами таких «об'єктів» взаємодії є семафори, сигнали і поштові скриньки;

• небезпечне взаємодія, коли обмін даними здійснюється за допомогою поділюваних ресурсів (наприклад, загальних змінних), незалежних від системних об'єктів взаємодії; при цьому цілісність інформації та неподільність явно забезпечуються самим додатком (в переважній більшості випадків -за того чи іншого системного об'єкта синхронізації та взаємодії).

Оскільки для будь-якого типу взаємодії потрібно системні об'єкти синхронізації, то всі наявні операційні системи надають додаткам деякий набір таких об'єктів. Нижче ми розглянемо найпоширеніші з них.

Колективна пам'ять - це область пам'яті, до якої мають доступ кілька процесів. Взаємодія через пам'ять, що розділяється є базовим механізмом взаімодествія процесів, до якого зводяться всі інші.

Семафор - це об'єкт синхронізації, що задає кількість користувачів (завдань, процесів), що мають одночасний доступ до деякого ресурсу. З кожним семафором пов'язані лічильник (значення семафора) і черга очікування (процесів, завдань, які очікують прийняття лічильником певного значення).

Подія - це логічний сигнал (сповіщення), який надходить асинхронно по відношенню до перебігу процесу. З кожним подією пов'язані булевська змінна E, приймаюча два значення (0- подія не прийшло, і 1 - подія прийшло), і черга очікування (процесів, завдань, які очікують приходу події).

Взаємне блокування процесів може виникнути через блокування файлів. Нехай, наприклад, процес 1 намагається встановити блокування в деякому файлі dead.txt в позиції 10.

Інший процес з 2 організовує блокування того ж самого файлу в позиції 20. До сих пір ситуація ще керована. Далі, процес 1 хоче організувати наступну блокування в позиції 20, де вже стоїть блокування процесу 2. При цьому використовується команда F_SETLKW. При цьому процес 1 призупиняється до тих пір, поки процес 2 знову не звільнить зі свого боку блокування в позиції 20. Тепер процес 2 намагається організувати в позиції 10, де процес 1 вже поставив свій блокування, таку ж блокування командою F_SETLKW, і також припиняється і чекає, поки процес 1 зніме блокування. Тепер обидва процеси, 1 і 2, припинені і обидва чекають один одного (F_SETLKW), утворюючи тупик. Жоден із процесів не може відновити своє виконання.

Причини виникнення цієї ситуації багато в чому викликані невдалим проектуванням алгоритмів. У Linux не передбачені механізми визначення та запобігання глухого кута, оскільки присутність цих механізмів суттєво впливає на продуктивність системи. Відповідальність за запобігання глухого кута цілком лягає на програміста.

Приклад програми, що викликає глухий кут:

```
#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>
```

```
#include <stdio.h>
#include <errno.h>
extern int errno;
void status(struct flock *lock)
{
  printf("Status: ");
  switch(lock->l_type)
  {
    case F_UNLCK: printf("F_UNLCK\n"); break;
    case F_RDLCK: printf("F_RDLCK (pid: %d)\n",
         lock->l_pid); break;
    case F_WRLCK: printf("F_WRLCK (pid: %d)\n",
         lock->l_pid); break;
    default : break;
  }
}
void writelock(char *process, int fd, off_t from,
     off_t to) {
  struct flock lock;
  lock.l_type = F_WRLCK;
```

```
  lock.l_start=from;
  lock.l_whence = SEEK_SET;
  lock.l_len=to;
  lock.l_pid = getpid();
  if (fcntl(fd, F_SETLKW, &lock) < 0)
  {
    printf("%s : Ошибка
       fcntl(fd, F_SETLKW, F_WRLCK) (%s)\n",
    process,strerror(errno));
    printf("\nВозник DEADLOCK (%s - process)!\n\n",
          process);
    exit(0);
  }
  else
    printf("%s : fcntl(fd, F_SETLKW, F_WRLCK)
         успешно\n", process);
  status(&lock);
}

int main()
{
  int fd, i;
```

```
pid_t pid;
if(( fd=creat("dead.txt", S_IRUSR |
            S_IWUSR | S_IRGRP | S_IROTH))<0)
{
  fprintf(stderr, "Ошибка при создании.....\n");
  exit(0);
}
/*Заповнюємо dead.txt 50 байтами символа X*/
for(i=0; i<50; i++)
  write(fd, "X", 1);
if((pid = fork()) < 0)
{
  fprintf(stderr, "Ошибка fork().....\n");
  exit(0);
}
else if(pid == 0) //Нащадок
{
  writelock("Нащадок", fd, 20, 0);
  sleep(3);
  writelock("Нащадок" , fd, 0, 20);
```

```
  }
  else //Батько
  {
    writelock("Батько", fd, 0, 20);
    sleep(1);
    writelock ("Батько", fd, 20, 0);
  }
  exit(0);
}
```

Спочатку створюється файл даних dead.txt, в який записується 50 символів X. Потім батьківський процес організовує блокування від байта 0 до байта 19, а Нащадок - блокування від байта 20 до кінця файлу (EOF). Нащадок "засинає" на 3 сек., А Батько тепер встановлює блокування від байта 20 до байта EOF і призупиняється, так як байти від 20 до EOF блоковані в даний момент нащадком, а Батько використовує команду F_SETLKW. Нарешті, Нащадок намагається встановити блокування на запис від байта 0 до байта 19, причому він також припиняється, так як в цій області вже встановлено блокування батька і використовується команда F_SETLKW. Тут виникає глухий кут, що підтверджується видачею коду помилки для errno = EDEADLK (виникнення тупика по ресурсам). Тупик може виникнути тільки при використанні команди F_SETLKW. Якщо застосовувати команду F_SETLK, видається код помилки для errno = EAGAIN (ресурс тимчасово недоступний).

Для створення безлічі семафорів служить системний виклик semget. Синтаксис даного системного виклику:

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget (key_t key, int nsems, int semflg);
```

Цілочисельне значення, що повертається в разі успішного завершення, тобто ідентифікатор множини семафорів (semid). У разі невдачі результат дорівнює -1.

Сенс аргументів key і semflg той же, що і у відповідних аргументів системного виклику msgget. Аргумент nsems задає число семафорів в множині. Якщо запитується ідентифікатор існуючого безлічі, значення nsems не повинно перевищувати числа семафорів в множині.

Перевищення системних параметрів SEMMNI, SEMMNS і SEMMSL при спробі створити нове безліч завжди веде до невдалого завершення. Системний параметр SEMMNI визначає максимально допустиму кількість унікальних ідентифікаторів множин семафорів в системі. Системний параметр SEMMNS визначає максимальна загальна кількість семафорів в системі. Системний параметр SEMMSL визначає максимально допустиму кількість семафорів в одному безлічі.

Богачев К.Ю. Основи паралельного програмування 97-122 с.

Самостійна робота №3

Створення програм під Linux з використанням потоків

У системах Linux концепція сигналів була розширена. Це викликано наступними недоліками старого підходу ANSI-C:

- відсутність можливості опитати актуального статусу сигналів.
- виникнення конфліктів між двома однаковими сигналами, наприклад, SIGINT, в одній програмі. Один обробник сигналу повинен був ігнорувати його, а інший - виконувати певну функцію, що могло заплутати програму.

У новій концепції сигналів спочатку вводиться змінна примітивного типу даних:

sigset_t signal_set;

Безліч сигналів инициализируется функцією sigemptyset ():

```
#include <signal.h>

int sigemptyset(sigset_t *sig_m);
```

Тепер можна додавати нові сигнали для конкретного процесу за допомогою функції:

```
#include <signal.h>

int sigaddset (sigset_t * sig_m, int signr);
```

де signr є номерами сигналів, які додаються в безліч. Також можна використовувати символічне ім'я, наприклад:

sigaddset (& signal_set, SIGINT);

За допомогою функції

#include <signal.h>

int sigdelset (setsig_t * sig_m, int signr);

сигнал signr видаляється з безлічі сигналів sig_m. Щоб перевірити, чи є деякий сигнал в безлічі, можна використовувати наступну функцію:

#include <signal.h>

int sigismember(sigset_t sig_m,int signr);

Якщо сигнал присутній у великій кількості, функція повертає 1, інакше 0.

Існує функція для збереження або зміни маски сигналів:

#include <signal.h>

int sigprocmask (int mode, const sigset_t * sig_m,

sigset_t * alt_sig_m);

Застосовуються три режими використання цієї функції:

sigprocmask (mode, NULL, alt_sig_m). У поточному процесі безліч сигналів записується за адресою alt_sig_m. mode в цьому випадку не діє;

sigprocmask (mode, sig_m, NULL). Маска сигналів змінюється на нову маску sig_m;

sigprocmask (mode, sig_m, alt_sig_m). Спочатку актуальна, яка використовується в поточному процесі маска сигналів записується в alt_sig_m, тобто зберігається. Потім встановлюється безліч сигналів sig_m.

Функція дозволяє використовувати три зумовлені константи:

SIG_BLOCK. До безлічі сигналів додаються всі встановлені в sig_m сигнали;

SIG_UNBLOCK. Сигнали, встановлені в sig_m, видаляються;

SIG_SETMASK. Встановлюється нова маска сигналів з сигналами, зазначеними в sig_m.

Якщо потрібно змінити маску сигналів або заборонити всі сигнали під час виконання певної частини коду, застосовується функція:

```
#include <signal.h>

int sigsuspend (const sigset_t * sig_m);
```

За допомогою цієї функції можна блокувати процес до тих пір, поки не прийде потрібний сигнал.

Самостійна робота №4

Програмування передачі повідомлень (MPI)

Першою програмою на C, яку пише більшість початківців програмістів, є програма, що виводить повідомлення "Привіт, світ!". Вона просто друкує повідомлення "Привіт, світ!" На термінал. Багатопроцесорний варіант містить процеси, кожен з яких посилає вітання іншому.

У MPI процеси, які беруть участь у виконанні паралельної програми, ідентифікуються послідовністю невід'ємних цілих чисел. Якщо у виконанні програми беруть участь P процесів, то вони будуть мати номери (ранги) 0, 1, ..., P-1. У наступній програмі кожен процес з рангом, не рівним 0, посилає повідомлення в процес 0, а процес 0 виводить всі повідомлення, які він отримав:

```
#include <stdio.h>

#include " mpi.h "


main (int argc, char ** argv) {

  int my_rank; / * Ранг процесу * /

  int p; / * Кількість процесів * /

  int source; / * Ранг посилає * /

  int dest; / * Ранг приймає * /

  int tag = 50; / * Тег повідомлень * /

  char message [100]; / * Пам'ять для повідомлення * /

  MPI_Status status; / * Статус повернення * /
```

```
MPI_Init (& argc, & argv);
MPI_Comm_rank (MPI_COMM_WORLD, & my_rank);
MPI_Comm_size (MPI_COMM_WORLD, & p);
if (my_rank! = 0) {
  sprintf (message, " Привіт з процесу% d! ", my_rank);
  dest = 0;
  / * Використовується strlen (message) +1
    щоб включити '\ 0' * /
  MPI_Send (message, strlen (message) +1,
    MPI_CHAR, dest, tag, MPI_COMM_WORLD);
} Else {/ * my_rank == 0 * /
  for (source = 1; source <p; source ++) {
  MPI_Recv (message, 100, MPI_CHAR, source, tag,
    MPI_COMM_WORLD, & status);
  printf ( "% s \ n ", message);
  }
}
MPI-Finalize ();
} / * Main * /
```

Якщо програму відкомпілювати і запустити для чотирьох процесів, вона виведе:

Привіт з процесу 1!

Привіт з процесу 2!

Привіт з процесу 3!

Деталі механізму запуску програми змінюються в залежності від архітектури комп'ютера, але основні дії будуть одні і ті ж на всіх машинах, якщо один процес працює на одному процесорі.

Користувач дає команду операційній системі, яка поміщає копію виконуваної програми на кожен процесор.

Кожен процесор починає виконання його копії програми.

Різні процеси можуть в кожен момент часу виконувати різні оператори, здійснюючи переходи в межах програми. Зазвичай перехід залежить від рангу процесу.

Наведена програма використовує парадигму ОПМД (одна програма - множинні дані). При цьому різні програми, що працюють на різних процесорах, реалізуються через розгалуження в межах однієї загальної програми на основі рангів процесу. Оператори, виконані процесом 0, відрізняються від виконаних іншими процесами навіть в разі, коли всі процеси підкоряються одній і тій же програмою. Це найбільш поширений метод написання програм ОПМД.

Кожна програма MPI містить директиву препроцесора:

#include " mpi.h "

Файл mpi.h містить визначення, макроозначення і прототипи функцій, необхідних для компіляції програм MPI. Перш ніж викликати будь-які інші функції MPI, потрібно одноразово викликати функцію MPI_Init (). Її аргументи - це покажчики на параметри функції main () - argc і argv. Вони дозволяють системі виконувати будь-яку спеціальну настройку, щоб використовувати бібліотеку MPI. Після того, як програма, що використовує бібліотеку MPI, закінчилася, необхідно викликати MPI_Finalize (). Ця функція завершує всі незавершені дії MPI - наприклад, нескінченне очікування передач. Типова програма MPI має наступну структуру:

```
#include "mpi.h"
. . .
main(int argc, char** argv) {
. . .
/ * Функції MPI можна викликати до цього моменту * /
```

```
MPI_Init (& argc, & argv);
. . .
MPI_Finalize ();

/ * Функції MPI можна викликати після цього моменту * /
. . .
```

Функції передачі повідомлень
MPI_Send () і MPI_Recv ()

Синтаксис функцій наведено нижче:

```
int MPI_Send(void* message, int count,
   MPI_Datatype datatype, int dest, int tag,
   MPI_Comm comm)

int MPI_Recv(void* message, int count,
   MPI_Datatype datatype, int source, int tag,
   MPI_Comm comm, MPI_Status* status)
```

Вміст повідомлення зберігається в блоці пам'яті, на який вказує аргумент message. Наступні два аргументи, count та datatype, дозволяють системі визначити кінець повідомлення: воно містить послідовність count значень, кожне з яких має тип даних datatype, який не є вбудованим типом C, хоча більшість визначених типів відповідає типам C. Визначені типи MPI і відповідні типи C представлені в табл. 8.

Таблиця 8. Відповідність між типами MPI і C.

Тип MPI	Відповідно тип C
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	-

MPI_PACKED -

Останні два типи, MPI_BYTE і MPI_PACKED, не відповідають стандартним типам С. Тип MPI_BYTE використовується, якщо система не повинна виконувати перетворення між різними уявленнями даних (наприклад, в гетерогенної мережі робочих станцій, які використовують різні представлення даних). Особливість типів MPI в тому, що вони дозволяють додаткам на гетерогенних архітектур взаємодіяти одноманітно, виконуючи необхідні перетворення типів прозоро для користувача.

Слід зазначити, що кількість пам'яті, виділеної для буфера отримання, може точно не відповідати розміру отримуваного повідомленні. Наприклад, при роботі програми розмір повідомлення, що посилається процесом 1 дорівнює strlen (message +1) = 28 символів, а процес 0 отримує повідомлення в буфер, який має розмір 100 символів. У будь-якому випадку процес-одержувач не може знати точного розміру посланого повідомлення. Тому MPI дозволяє приймати повідомлення, поки є місце в виділеної пам'яті. Якщо місця недостатньо, виникає помилка виходу за межі пам'яті.

Аргументи dest і source відповідають рангах процесів прийому і посилки. MPI дозволяє source приймати значення зумовленої константи MPI_ANY_SOURCE, яке може використовуватися, якщо процес готовий отримати повідомлення від будь-якого іншого процесу. Для dest подібної константи немає.

MPI має два механізми, призначені для поділу просторів повідомленні - теги і комунікатори. Аргументи tag і comm відповідно визначають тег і комунікатор. Існує груповий тег MPI_ANY_TAG, що визначає будь-який тег для повідомлення.

Останній аргумент MPI_Recv (), status, повертає інформацію, що відноситься до фактично отриманими даними. Він посилається на запис з двома полями: одне - для джерела, інше - для тега. Наприклад, якщо в якості джерела був зазначений MPI_ANY_SOURCE, то status буде містити ранг процесу, який надіслав повідомлення.

Щоб мати змогу здійснювати повідомлень також можна використовувати варіанти функцій MPI_Send (). Ці варіанти визначають різні режими передачі повідомлень (стандартний, синхронний, буферізованние і режим передачі по готовності). Інформація про режими приведена в табл. 9.

Таблиця 9. Комунікаційні режими MPI.

Назва режиму Умова завершення Функція
Стандартна передача Як для синхронної або буферизує MPI_SEND
Синхронна передача Завершується, коли завершено прийом MPI_SSEND
Буферизованная передача Завжди завершується MPI_BSEND
Передача по готовності Завжди завершується MPI_RSEND
Прийом Завершується, коли повідомлення було доставлене MPI_RECV

Програміст, який використовує стандартний режим передачі, повинен дотримуватися наступних рекомендацій:

Він не повинен розраховувати, що передача завершиться перед початком прийому повідомлення. Якщо передача є блокує, то в деяких випадках може виникнути тупик.

Чи не повинен розраховувати, що передача завершиться після початку прийому повідомлення. У цьому випадку порядок прийому послідовності повідомлень може бути порушений.

Процеси повинні приймати і обробляти всі повідомлення, адресовані їм.

Синхронна передача може бути істотно повільніше стандартної. Однак вона не призведе до перевантаження комунікаційної мережі повідомленнями і забезпечить детермінована поведінка програми. Використання цього режиму передачі також полегшує налагодження паралельного додатка.

Буферизованная передача гарантує негайне завершення, оскільки повідомлення спочатку копіюється в системний буфер, а потім доставляється. Недоліком її є необхідність виділення спеціальних буферів, які споживають ресурси системи.

Передача по готовності передбачає ініціювання передачі в момент, коли приймач викликає відповідний їй прийом. Цей режим гарантує відсутність в комунікаційної мережі блукаючих повідомлень.

Самостійна робота №5

Робота на PVM. Створення потоків і паралельна обробка даних в PVM

Програмне забезпечення PVM надає уніфіковані

структури, за допомогою яких паралельні програми можуть розроблятися ефективним і цілеспрямованим способом з використанням існуючого обладнання. PVM дозволяє групі гетерогенних комп'ютерних систем сприйматися як одна паралельна віртуальна машина. PVM прозоро керує

обробкою всіх повідомлень, перетворенням даних і виконанням завдань в межах мережі, що включає несумісні комп'ютерні архітектури.

Обчислювальна модель PVM є простою, вельми узагальненої, тому пристосовується до широкого спектру програмних структур додатків. Програмний інтерфейс навмисно зроблений `` цільовим '', що дозволяє доступ до простих програмним структурам осуществляеть інтуїтивним способом. Користувач пише свою програму у вигляді групи взаємопов'язаних `` завдань ''. Завдання отримують доступ до ресурсів PVM за допомогою бібліотеки підпрограм зі стандартизованим інтерфейсом. Ці підпрограми дозволяють ініціювати і завершити завдання в мережі, а також забезпечити зв'язок між завданнями і їх синхронізацію. Примітиви обміну повідомленнями PVM орієнтовані на гетерогенні операції, що включають певні конструкції для буферизації і пересилання. Комунікаційні конструкції містять їх для передачі і прийому структур даних, також, як і високорівневі примітиви, такі як широкомовна передача, бар'єрна синхронізація і глобальне підсумовування.

Завдання PVM можуть містити структури для забезпечення необхідних рівнів контролю і залежності. Іншими словами, в будь-який `` точці '' виконання взаємопов'язаних програм будь-яка можлива завдання може запускати або зупиняти інші завдання, додавати або видаляти комп'ютери з віртуальної машини. Кожен процес може взаємодіяти і / або синхронізуватися з будь-яким іншим. Кожна специфічна структура для контролю і залежності може бути реалізована в системі PVM адекватним використанням конструкцій PVM і керуючих конструкцій головного (хост) мови системи.

Всеосяжна природа (специфічна для концепції віртуальної машини), а також її простий, але функціонально повний програмний інтерфейс, забезпечили системі PVM широке визнання, в тому числі і в науковому співтоваристві, пов'язаному з високошвидкісними обчисленнями.

Нижче наведена програма PVM hello - простий приклад, який ілюструє базову концепцію програмування PVM. Ця програма розглядається як запускається вручну; після виведення на екран свого ідентифікатора завдання (отриманого за допомогою pvm_mytid ()) вона породжує копію іншої програми під назвою hello_other, використовуючи функцію pvm_spawn (). Успішне породження змушує програму виконати блокуючий прийом за допомогою pvm_recv (). Після прийому повідомлення програма виводить на екран повідомлення, надіслане їй абонентом - так само як і свій ідентифікатор завдання; вміст буфера витягується з повідомлення застосуванням pvm_upsksrt (). Заключний виклик pvm_exit `` виводить '' програму з системи PVM:

```
#include "pvm3.h"

main()

{

  int cc, tid, msgtag;

  char buf[100];

  printf("Это программа t%x\n", pvm_mytid());

  cc = pvm_spawn("hello_other", (char**)0, 0, "",

      1, &tid);

  if (cc == 1) {

    msgtag = 1;

    pvm_recv(tid, msgtag);

    pvm_upkstr(buf);

    printf("Виведення з t%x: %s\n", tid, buf);

  } else

    printf("Неможливо запустити hello_other\n");

  pvm_exit();

}
```

Нижче наведено лістинг `` відомою " або породжується програми; її першою дією в PVM є отримання ідентифікатора `` провідною " завдання викликом pvm_parent (). Потім ця програма отримує власне ім'я хоста і передає його провідною, використовуючи послідовність з трьох викликів: pvm_initsend () - для ініціалізації буфера передачі; pvm_pkstr () - для розміщення рядка, навмисно введеної в архітектурно-незалежному стилі, в буфері передачі; і

pvm_send () - для її пересилки в запитувач процес, який визначається за допомогою ptid, і маркування повідомлення числом 1:

```
#include "pvm3.h"

main()

{

  int ptid, msgtag;

  char buf[100];

  ptid = pvm_parent();

  strcopy(buf, "hello, world from");

  msgtag = 1;

  gethostname(buf + strlen(buf), 64);

  msgtag = 1;

  pvm_initsend(PvmDataDefault);

  pvm_pkstr(buf);

  pvm_send(ptid, msgtag);

  pvm_exit();

}
```

Рекомендована література

Базова

1. **Марченко К.М.**, Оришака О.В., Босько В.В., Собінов О.Г. Комп'ютерні системи : навч. посіб. М-во освіти і науки України, Центральноукраїн. нац. техн. ун-т, каф. кібербезпеки та програмного забезпечення. – Кропивницький : ЦНТУ, – Кропивницький: 2022. – 130 с.
http://dspace.kntu.kr.ua/jspui/handle/123456789/11956

2. Паралельні та розподілені обчислення: навчальний посібник для вищих закладів освіти / К.Т. Кузьма, О.В. Мельник. – Миколаїв: ФОП Швець В.М., 2020. – 172 с.

3. Ясько, М.М. Навчальний посібник до вивчення курсів “Паралельна обробка даних” та “Мови обчислень та кластерні системи” [Текст] /М.М.Ясько. – Д.: РВВ ДНУ, 2010. – 76с.
http://repository.dnu.dp.ua:1100/upload/0437891d38e3501a2c067570a5fcea63PP6.pdf

4. Аксак Н. Г. Паралельні та розподілені обчислення: підруч. / Н. Г. Аксак,О. Г. Руденко, А.М. Гуржій. – Х.: Компанія СМІТ, 2009. – 480 с.

5. Комп’ютерні системи реального часу, навчальний посібник [Електронний ресурс]: навч. посіб. для здобувачів ступеня магістра за освітньою програмою ”Системне програмування та спеціалізовані комп’ютерні системи” спеціальності 123 «Комп’ютерна інженерія» / В. Г. Зайцев, Є. І. Цибаєв; КПІ ім. Ігоря Сікорського. - Електронні текстові дані (1 файл: 4Мбайт). -Київ: КПІ ім. Ігоря Сікорського, 2019.-162 с.

6. Паралельні та розподілені обчислення [Текст] : навч. підруч. для студентів вищ. навч. закл. / А. Луцків, С. Луценко, В. Пасічник. – Львів : Магнолія 2006, 2017. – 565, [1] с. : схеми

7. Steen M. van, Tanenbaum A.S. Distributed Systems, 3rd ed. – Pearson, 2017. – 596 p.

8. Burns B. Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services – O'Reilly Media, 2018. – 166 p. – ISBN 9781491983645.

9. Newman S. Building Microservices – O'Reilly Media, 2015. – 304 с. – ISBN: 9781491950357.

Допоміжна

10. Мельник А. О. «Архітектура комп'ютера». Наукове видання. — Луцьк: 2008. 470 с.

11. Інструктивно-методичні рекомендації з дисципліни «Технології розподілених систем та паралельних обчислень» / уклад.: Оксана Наконечна, Тетяна Ярмоленко, Вікторія Алексеєнко, Богданна Якимчук. Житомир: Житомир: Вид-во ЖДУ ім. Івана Франка, 2023. 74 с.

http://eprints.zu.edu.ua/35948/1/інст-метод_технології%20розпод%20систем%20%284%29.pdf

12. **К.М. Марченко**, О.В. Оришака, А.К. Марченко, А.М. Мельник. Ризики впровадження штучного інтелекту в комп'ютерні системи / Центральноукраїнський науковий вісник: Технічні науки, вип. № 4 (32), ч. 1. – Кропивницький, ЦНТУ, 2022 - с. 119-124. http://mapiea.kntu.kr.ua/archive/36_I.html

13. Ian Gorton. Foundations of Scalable Systems. O'Reilly Media. 2022. 379 с.

14. Horstmann Cay S. Core Java, Volume II--Advanced Features, 11th Edition, – Pearson, 2019. – 1040 p.

15. Kasun Indrasiri, Prabath Siriwardena, Microservices for the Enterprise – San Jose, CA, USA, 2018. – 434 с.

16. Christian Posta, Microservices for Java Developers – O'Reilly Media, 2015. – 129 с.

Корисні ресурси

17. Підтримка OpenMP в Lazarus/Freepascal: http://freepascal.ru/forum/viewtopic.php?f=9&t=25006

18. Бібліотека OpenMP. Паралельний цикл: http://ccfit.nsu.ru/arom/data/openmp

19. Оцінка продуктивності обчислювальних систем: http://citforum.ck.ua/hardware/svk/glava_3.shtml.

20. Науковий журнал "Комп'ютерні системи та мережі" http://csn.lpnu.ua/ua/magazine/details.

21. The TOP500 table – the 500 most powerful commercially available computer systems [Electronic resource]. – Access mode: http://www.top500.org/

22. Дистанційна освіта ЦНТУ. URL: http://moodle.kntu.kr.ua/course/view.php?id=620