САМОСТІЙНА РОБОТА №1

Назва роботи: Паралельні асинхронні процеси в ОС UNIX

Мета роботи: Засвоїти основи паралельних обчислень рівня задач.

1. Загальні відомості

1.1. Уся побудова операційної системи UNIX заснована на використанні концепції процесів. Контекст процесу складається з контексту користувача і контексту ядра.

Під контекстом користувача розуміють код і дані, розташовані в адресному просторі процесу.

Під поняттям "контекст ядра" поєднуються системний контекст і реєстровий контекст. Ми будемо виділяти в контексті ядра стек ядра, що використовується при роботі процесу в режимі ядра (kernel mode), і дані ядра, які зберігаються в структурах, що є аналогом блоку керування процесом - PCB. В дані ядра входять: ідентифікатор користувача - UID, груповий ідентифікатор користувача - GID, ідентифікатор процесу - PID, ідентифікатор батьківського процесу - PPID.

1.2. Кожен процес в операційній системі отримує унікальний ідентифікаційний номер - PID (process identificator). При створенні нового процесу операційна система намагається привласнити йому вільний номер більший, ніж у процесу, створеного перед ним. Якщо таких вільних номерів не виявляється (наприклад, ми досягли максимально можливого номера для процесу), то операційна система вибирає мінімальний номер із усіх вільних номерів.

1.3. В операційній системі UNIX усі процеси, крім одного, що створюється при старті операційної системи, можуть бути породжені тільки іншими процесами. Як прабатьки всіх інших процесів у UNIX подібних системах можуть виступати процеси з номерами 1 чи 0.

Таким чином, усі процеси в UNIX зв'язані відносинами процес-батько - процес-син й утворюють генеалогічне дерево процесів. Для збереження цілісності генеалогічного дерева в ситуаціях, коли процес-батько завершує свою роботу до завершення виконання синівського процесу, ідентифікатор батьківського процесу в даних ядра синівського процесу (PPID - parent process identificator) змінює своє значення на значення 1, що відповідає ідентифікатору процесу init, час життя якого визначає час функціонування операційної системи. Тим самим процес init як би усиновляє "осиротілі" процеси.

1.4. В операційній системі UNIX новий процес може бути породжений єдиним способом - за допомогою системного виклику fork(). При цьому створений процес буде практично повною копією батьківського процесу. Батьківський процес та процес-син починають одночасну роботу з точки виклику функції fork(). fork() повертає в батьківський процес PID процесу-сина, а в процес-син - нуль. Звичайно користі від двох однакових працюючих процесів мало. Тому один з процесів заміняють іншим. Робиться це за допомогою функції exec(), котра завантажує програму з диску, та заміщає нею поточний процес.

2. Синтаксис та призначення системних викликів

2.1. Системний виклик `fork`.

```
#include <unistd.h>

int fork ()
```

Виклик fork приводить до створення нового процесу (породженого процесу) – майже точної копії процесу, що зробив виклик

(батьківського процесу). У породженого процесу в порівнянні з батьківським змінюються значення наступних параметрів:

1. ідентифікатор процесу (pid);
2. ідентифікатор батьківського процесу (ppid);
3. час, що залишився до одержання сигналу SIGALRM;
4. сигнали, що очікували доставки батьківському процесу, не будуть доставлятися породженому процесу.

При успішному завершенні породженому процесу повертається 0, а батьківському процесу повертається ідентифікатор породженого процесу. У випадку помилки батьківському процесу повертається -1, не створюється нового процесу і змінній errno присвоюється код помилки.

2.2. Системний виклик `exec`.

```
#include <unistd.h>

int execl  (path, arg0, arg1, ..., argn, (char*) 0)

           char *path, *arg0, *arg1, ..., *argn;

int execv  (path, argv)

           char *path, *argv [];

int execle (path, arg0, arg1, ..., argn, (char*) 0, envp)

           char *path, *arg0, *arg1, ..., *argn, *envp [];

int execve (path, argv, envp)

           char *path, *argv [], *envp [];

int execlp (file, arg0, arg1, ..., argn, (char*) 0)

           char *file, *arg0, *arg1, ..., *argn;

int execvp (file, argv)

           char *file, *argv [];
```

Усі форми системного виклику exec заміщують процес, що викликав, новим процесом, який завантажується зі звичайного виконавчого файлу. Якщо системний виклик exec закінчився успішно, то він не може повернути керування, тому що процес, що викликав, уже замінений новим процесом.

Повернення із системного виклику exec свідчить про помилку. У такому випадку результат дорівнює -1, а змінній errno присвоюється код помилки.

2.3. Системні виклики `getpid` та `getppid`.

```
#include <unistd.h>

int getpid ( )

int getppid ( )
```

Системний виклик getpid повертає ідентифікатор поточного процесу.

Системний виклик getppid повертає ідентифікатор батьківського процесу.

2.4. Системні виклики waitpid та wait.

```
#include <wait.h>

int waitpid(pid, stat_loc, options)

            int pid;

            int* status;

            int options;

int wait (stat_loc)

          int* status;
```

Системний виклик waitpid() блокує виконання поточного процесу доти, поки не завершиться породжений їм процес, обумовлений значенням параметра pid, або поточний процес не одержить сигнал, для якого встановлена реакція за замовчуванням "завершити процес" чи реакція обробки функцією користувача. Якщо породжений процес, заданий параметром pid, до моменту системного виклику закінчив виконання, то системний виклик повертається негайно без блокування поточного процесу.

Параметр pid визначає породжений процес, завершення якого чекає процес-батько, у такий спосіб:

1. Якщо pid > 0 очікуємо завершення процесу з ідентифікатором pid.
2. Якщо pid = 0, то очікуємо завершення будь-якого породженого процесу в групі, до якої належить процес-батько.
3. Якщо pid = -1, то очікуємо завершення будь-якого породженого процесу.
4. Якщо pid < 0, але не -1, то очікуємо завершення будь-якого породженого процесу з групи, ідентифікатор якої дорівнює абсолютному значенню параметра pid.

Статус завершення породженого процесу містяться в молодших 16 біт слова, на яке вказує параметр status. За допомогою статусу можна довідатися, зупинено чи завершено виконання породженого процесу. Якщо породжений процес завершився, то статус вказує причину завершення. Статус трактується в такий спосіб:

1. Якщо породжений процес зупинено, старший 8 біт статусу містять номер сигналу, що став причиною зупинки, а молодші 8 біт встановлюються рівними 0177.
2. Якщо породжений процес завершився за допомогою системного виклику exit, то молодші 8 біт статусу будуть рівні 0, а старші 8 біт будуть містити молодші 8 біт аргументу, що породжений процес передає системному виклику exit.
3. Якщо породжений процес завершився через одержання сигналу, то старші 8 біт статусу будуть рівні 0, а молодші 8 біт будуть містити номер сигналу,

що викликав завершення процесу. Крім того, якщо молодший сьомий біт (біт 0200) дорівнює 1, буде зроблений дамп оперативної пам'яті.

Параметр status може бути заданий рівним 0, якщо ця інформація не має для нас значення.

Параметр options задає деякі опції виконання системного виклику. Якщо значення options дорівнює WNOHANG повернення з виклику відбувається негайно без блокування поточного процесу в будь-якому випадку.

При виявленні процесу, що завершився, системний виклик повертає його ідентифікатор. Якщо виклик був зроблений із встановленою опцією WNOHANG, і породжений процес, специфікований параметром pid, існує, але ще не завершився, системний виклик поверне значення 0. В всіх інших випадках він повертає негативне значення. Якщо виконання системного виклику waitpid завершилося внаслідок одержання сигналу, то результат буде дорівнювати -1, а змінній errno буде присвоєно значення EINTR (переривання системного виклику)..

Системний виклик wait є синонімом для системного виклику waitpid зі значеннями параметрів pid = -1, options = 0.

2.5. Системний виклик `exit`.

```
#include <stdlib.h>

void exit (status)

          int status;
```

Системний виклик exit завершує процес, що звернувся до нього, при цьому послідовно виконуються наступні дії:

1. У процесі, що викликав, закриваються всі дескриптори відкритих файлів.
2. Якщо батьківський процес знаходиться в стані виклику wait, то системний виклик wait завершується, видаючи батьківському процесу як результат ідентифікатор завершеного процесу і молодші 8 біт коду його завершення.
3. Якщо батьківський процес не знаходиться в стані виклику wait, то процес, що викликав exit, переходить у стан зомбі. Це такий стан, коли процес займає тільки елемент у таблиці процесів і не займає пам'яті ні в адресному просторі користувача, ні в адресному просторі ядра.

2.6. Системний виклик `sleep`.

```
#include <unistd.h>

unsigned sleep (seconds)

                unsigned seconds;
```

Виконання процесу припиняється на задане аргументом seconds число секунд. Час фактичного припинення може виявитися менше заданого з двох причин:

1. Плановані пробудження процесів відбуваються у фіксовані секундні інтервали часу, відповідно до внутрішнього годинника.
2. Будь-який перехоплений сигнал перериває "сплячку", після чого спрацьовує реакція на сигнал.

З іншого боку, фактичний час припинення може виявитися більше запитаного через те, що система зайнята іншою, більш пріоритетною діяльністю. Результат функції sleep є час "недосипання" (запитаний час мінус фактичний).

3. Приклад програми

Лістинг 3.1. Приклад синівської програми (child.c):

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define SLEEP_TIME    5
#define ITERATION_NUM 7

void main(void)
{
  int i = 0;
  int mPid  = getpid();
  int pPid = getppid();
  printf("Child: My PID = %d, My parent's PID = %d\n\n", mPid, pPid);
  for(i; i < ITERATION_NUM; i++)
  {
    printf("CHILD now is working.\n");
    sleep(SLEEP_TIME);
    //for(j=0;j<1000000;j++);
  }
  exit(0);

}
```

3.2. Приклад батьківської програми (parent.c):

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

#define SLEEP_TIME    5
#define ITERATION_NUM 7

void main(void)
{
  int i = 0;
  int cPid = fork();
  int errFlag;
  int mPid;
  int pPid;
  switch(cPid)
  {
    case -1:
      fprintf(stderr,"Can't fork for a child.\n");
      exit(1);
    case 0:
      errFlag = execl("./child","child",0);
      if(errFlag == -1)
      {
        fprintf(stderr, "Can't execute the external child.\n");
        exit(1);
```

```
        }
        break;
      default:
        mPid  = getpid();
        pPid  = getppid();
        printf("Parent: My Pid = %d, "
              "Parent Pid = %d, "
              "Child Pid = %d\n", mPid, pPid, cPid);
        for(i; i < ITERATION_NUM; i++)
        {
          printf("PARENT now is working.\n");
          sleep(SLEEP_TIME);
          //for(j=0;j<1000000;j++);
        }
        int status;
        wait(&status);
        printf("Parent: Child exit code is %d.\n",(status&0xff00)>>8);
        exit(0);
    }

}
```

4. Варіанти завдань

n%4 + 1	Опис завдання
1	Написати батьківську програму (parent.c) яка буде запускати N синівських процесів кожен з яких буде знати свій порядковий номер запуску.
2	Написати батьківську програму (parent.c) яка буде запускати синівський процес, який в свою чергу буде запускати ще один синівський (по відношенню до нього) процес і так далі. В результаті повинно запуститися ланцюжок з N синівських процесів, кожен з яких повинен знати свою глибину у дереві процесів.
3	Написати батьківську програму, яка зможе викликати себе у якості синівського процесу (при цьому обов‘язково використати системний виклик exec). Батьківський процес повинен породити N синівських процесів кожен з яких буде знати свій порядковий номер запуску.
4	Написати батьківську програму, яка зможе викликати себе у якості синівського процесу (при цьому обов‘язково використати системний виклик exec). Батьківський процес повинен породити ланцюжок з N синівських процесів, кожен з яких повинен знати свою глибину у дереві процесів.
де n – порядковий номер у журналі	

5. Зміст звіту

- Титульний лист;
- Завдання;
- Алгоритм рішення завдання;
- Код програми;
- Екранна форма з результатами роботи програми;
- Висновки.