ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«МЕТІНВЕСТ ПОЛІТЕХНІКА»

ОПЕРАЦІЙНІ СИСТЕМИ ТА ОСНОВИ СИСТЕМНОГО ПРОГРАМУВАННЯ:

методичні рекомендації до виконання індивідуального розрахункового завдання

Запоріжжя 2024

metinvest polytechnic

УДК 004.45(072)
О60

Рекомендовано Науково-методичною радою
ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«МЕТІНВЕСТ ПОЛІТЕХНІКА»
(протокол № 1 від 27.09.2024 р.)

Укладач:
Шматко О.В., к.т.н.
Гамаюн І.П., д.т.н.
Держевецька М.А., к.е.н.

О60 **Операційні системи та основи системного програмування** : методичні рекомендації до виконання індивідуального розрахункового завдання / уклад.: О. В. Шматко, І. П. Гамаюн, М. А. Держевецька. Запоріжжя : ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ «МЕТІНВЕСТ ПОЛІТЕХНІКА», 2024. 66 с.

Методичні вказівки включають тематику індивідуальних завдань, методичні пояснення щодо порядку їх виконання, критерії оцінювання виконаного індивідуального завдання, вимоги до його оформлення, включаючи зразок звіту.

Рекомендовано для студентів спеціальності 122 «Комп'ютерні науки» першого (бакалаврського) рівня освіти.

УДК 004.45(072)

ЗМІСТ

ВСТУП

Індивідуальне розрахункове завдання (надалі - ІНДРЗ) виконується з актуальних проблем пов'язаних з методами та моделями штучного інтелекту та являє собою самостійне дослідження, яке є важливою ланкою у системі опанування загальних та фахових компетенцій здобувачами ОПП «Комп'ютерні науки» спеціальності 122 Комп'ютерні науки усіх форм навчання.

Методичні вказівки укладено на підставі Стандарту вищої освіти за спеціальністю 122 Комп'ютерні науки галузі знань 12 Інформаційні технології для першого (бакалаврського) рівня вищої освіти, затвердженого Наказом Міністерства освіти і науки України від 28.04.2022 р. № 393.

В процесі виконання ІНДРЗ передбачається поєднання теоретичних знань і практичних умінь, набутих здобувачами освіти в результаті вивчення дисципліни професійної підготовки бакалавра, а саме: Операційні системи та основи системного програмування.

ІНДРЗ – це індивідуальне завдання, яке є творчим та практичним рішенням конкретних завдань з використанням ключових технологій забезпечення організації обчислювальних процесів в інформаційних системах різного призначення з урахуванням архітектури, конфігурування, показників результативності функціонування операційних систем і системного програмного забезпечення, виконане здобувачем вищої освіти самостійно під керівництвом викладача згідно із поставленими завданнями.

Виконання ІНДРЗ сприяє розширенню та поглибленню теоретичних знань, розвитку навичок їх практичного використання, формує вміння самостійного розв'язання конкретних професійних завдань, створює наукове підґрунтя для виконання кваліфікаційної роботи бакалавра.

Мета індивідуального розрахункового завдання – поглиблення теоретичних знань та закріплення практичних навичок використання можливості операційних систем, офісних програмних продуктів, розробки програмних моделей предметних середовищ, вибору парадигм програмування з позицій зручності та якості застосування..

Для досягнення цієї мети необхідно поставити та вирішити такі завдання:

– сформулювати постановку задачі з дослідження проблемної ситуації (практичного завдання) відповідно до обраного варіанту ІНДРЗ;

– вибрати мову системного програмування та методи розробки програм, що взаємодіють з компонентами комп'ютерних систем.

– застосовувати знання методології та CASE-засобів проектування складних систем для проєктування та розробки програмних компонентів згідно обраного варіанту,

– застосувати методи об'єктно-орієнтованого аналізу та проєктування при розробці та дослідженні функціональних моделей розроблених програмних компонентів.

– представити керівнику у встановлений термін результати індивідуального розрахункового завдання, у якому у логічній послідовності відобразити основні етапи і результати дослідження, обґрунтувати методи, засоби, інструменти вирішення проблемної ситуації (практичного завдання), а також очікувані результати і рекомендації і пропозиції вирішення проблемної ситуації (практичного завдання) із застосуванням методів штучного інтелекту;

– підготувати презентацію результатів виконання індивідуального розрахункового завдання у вигляді проєкту і продемонструвати вміння обґрунтовано і коректно викладати та відстоювати власну позицію перед професійною аудиторією та експертами з інших галузей знань під час захисту.

Дисципліна спрямована на отримання здобувачами наступних загальних та спеціальних (фахових) компетентностей:

– ЗК1. Здатність до абстрактного мислення, аналізу та синтезу.

– ЗК2. Здатність застосовувати знання у практичних ситуаціях.

– ЗК3. Знання та розуміння предметної області та розуміння професійної діяльності.

– ЗК6. Здатність вчитися й оволодівати сучасними знаннями.

– ЗК9. Здатність працювати в команді.

– СК3. Здатність до логічного мислення, побудови логічних висновків, використання формальних мов і моделей алгоритмічних обчислень, проєктування, розроблення й аналізу алгоритмів, оцінювання їх ефективності та складності, розв'язності та нерозв'язності алгоритмічних проблем для адекватного моделювання предметних областей і створення програмних та інформаційних систем

– СК7. Здатність застосовувати теоретичні та практичні основи методології та технології моделювання для дослідження характеристик

і поведінки складних об’єктів і систем, проводити обчислювальні експерименти з обробкою й аналізом результатів.

У результаті виконання індивідуального розрахункового завдання здобувач вищої освіти повинен продемонструвати достатній рівень сформованості наступних програмних результатів навчання:

– ПР5. Проєктувати, розробляти та аналізувати алгоритми розв’язання обчислювальних та логічних задач, оцінювати ефективність та складність алгоритмів на основі застосування формальних моделей алгоритмів та обчислюваних функцій.

– ПР9. Розробляти програмні моделі предметних середовищ, вибирати парадигму програмування з позицій зручності та якості застосування для реалізації методів та алгоритмів розв’язання задач в галузі комп’ютерних наук.

1. ЗАГАЛЬНІ МЕТОДИЧНІ РЕКОМЕНДАЦІЇ З ВИКОНАННЯ ІНДИВІДУАЛЬНОГО РОЗРАХУНКОВОГО ЗАВДАННЯ СТУДЕНТА

Самостійна робота студентів (СРС) займає провідне місце у системі сучасної вищої освіти. З усіх видів навчальної діяльності СРС значною мірою забезпечує формування самостійності як провідної риси особистості студента.

Самостійна робота завершує завдання усіх інших видів навчальної діяльності. Адже знання, що не стали об'єктом власної діяльності, не можуть вважатися дійсним надбанням людини. Тому СРС має навчальне, особисте та суспільне значення.

СРС – це багатоаспектне та поліфункціональне явище з двоєдиністю цілей:

- формування самостійності студента;
- розвиток здібностей, вмінь, знань та навичок студентів.

Завдяки СРС відбувається перехід від переважно виконавчої репродуктивної діяльності студентів до пошукового, творчого начала на всіх етапах навчання у ВНЗ.

Індивідуальне розрахункове завдання (ІНДРЗ) з дисципліни «Операційні системи та основи системного програмування» припускає її здійснення в наступних видах: самостійне вивчення теоретичного матеріалу, самостійне виконання практичних завдань для більш глибокого засвоєння матеріалу.

Метою виконання ІНДРЗ є більше глибоке вивчення сфери застосування та можливостей використання мов системного програмування та методів розробки програм, що взаємодіють з компонентами комп'ютерних систем, знання мережевих технології, архітектури комп'ютерних мереж, отримання практичних навичок технології адміністрування комп'ютерних мереж та їх програмного забезпечення.

Правильна організація самостійної роботи необхідна для більш повного оволодіння дисципліною та визначає успішність здачі заліку й наступної практичної діяльності.

Самостійна робота виконується студентами під керівництвом викладача, який здійснює аудиторну роботу в навчальній групі.

Самостійна робота студентів повинна мати такі головні ознаки:

- бути виконаною особисто студентом;
- бути закінченою розробкою, де розкриваються й аналізуються актуальні проблеми з певної теми або її окремих аспектів;

– демонструвати достатню компетентність автора в розкритті питань, що досліджуються;

– мати навчальну, наукову, й/або практичну спрямованість і значимість;

– містити певні елементи новизни;

– самостійна індивідуальна розрахункова робота оформляється відповідно до вимог кафедри.

1.1 Основні вимоги до виконання індивідуального розрахункового завдання

Перед виконанням самостійної роботи потрібно повністю ознайомитися зі змістом завдання, підібрати потрібну літературу, визначити усі параметри виконання індивідуального завдання.

Результатом виконання самостійної роботи є звіт, який виконується з використанням комп'ютерної техніки та надрукований на папері формату А4. Оформлення звіту: шрифт − Arial; розмір шрифту − 14 кегель; інтервал між рядками − півтора; абзац − 12,5 мм, поля: верхнє і нижнє – 20 мм, ліве – 25 мм, праве – 15 мм; нумерація сторінок − по центру нижнього поля. Зразок оформлення звіту наведено у додатку А.

Після перевірки кожного завдання викладачем студент зобов'язаний усунути допущені помилки, інакше він не допускається до виконання наступного завдання.

Усі види самостійної роботи повинні бути здані у встановлений графіком термін. Викладач фіксує факт здачі кожної роботи та виставляє оцінку в журнал.

Поради із планування й організації часу, необхідного для виконання самостійної роботи.

Раціональне планування і організація самостійної роботи студентів є найважливішою умовою її ефективності.

Планування самостійної роботи направлено на формування логічно вибудуваної, прозорої, зрозумілої, доступної і ефективної системи організації самостійної роботи та її оцінки.

При цьому необхідно пам'ятати, що самостійна робота студентів виконує в навчальному процесі кілька функцій:

− розвиваючу (підвищення культури розумової праці, привчання до творчих видів діяльності, вдосконалення інтелектуальних здібностей студентів);

− інформаційно-навчальну (навчальна діяльність на аудиторних заняттях, непідкріплена самостійною роботою, стає мало результативною);

− орієнтуючу і стимулюючу (процесу навчання надається прискорення і мотивація);

− виховну (формуються і розвиваються професійні якості фахівця);

− дослідницьку (новий рівень професійно-творчого мислення).

В основі самостійної роботи студентів лежать наступні принципи: розвиток творчої діяльності, цільове планування, особистісно-діяльнісний підхід.

Самостійну роботу можна назвати ефективною тільки в тому випадку, якщо вона організована і реалізується в освітньому процесі як цілісна система на всіх етапах навчання.

Можна виділити кілька об'єктивних закономірностей організації самостійної роботи студентів:

− творча складова самостійної роботи зростає в міру навчання;

− в процесі організації самостійної роботи виникає потреба в методичному забезпеченні;

− застосування інформаційних технологій стає частиною організації і моніторингу самостійної роботи студентів на всіх її етапах.

У процесі самостійної роботи студент набуває навиків самоорганізації, самоконтролю, самоврядування, саморефлексії і стає активним самостійним суб'єктом навчальної діяльності.

Самостійна робота повинна давати важливий вплив на формування особистості майбутнього фахівця. Кожен, хто навчається самостійно планує режим своєї роботи з урахуванням часу роботи бібліотеки, профільних лабораторій, комп'ютерних класів і т.п. Він виконує самостійну роботу за особистим індивідуальним планом, в залежності від його підготовки, часу та інших умов.

Першим завданням в організації позааудиторної самостійної роботи є складання розкладу, що відображає час занять і їх характер, перерви на обід, вечеря, відпочинок, сон, проїзд і т.ін. Із самого початку студенту не потрібно прагнути робити відразу найважчу її частину. Доцільно вибрати щось середнє за складністю. Після цього, перейти до більш важкої роботи, легке залишивши наостанок. Розумову працю необхідно не тільки правильно організувати, а й стимулювати. Важливо вміти підтримувати стійку увагу до досліджуваного матеріалу.

Вироблення уваги вимагає значних вольових зусиль від студента. Стійка увага з'являється тоді, коли людина ставиться до справи з інтересом.

Слід правильно організувати свої заняття за часом: 50 хвилин − робота, 5-10 хвилин − перерва, після 3 годин роботи перерва − 20-25 хвилин. Інакше наростаюча втома спричинить нестійкість уваги. Організація активного відпочинку передбачає чергування розумової та фізичної діяльності, що відновлює працездатність людини.

1.2 Опис послідовності дій студента при виконанні самостійної роботи

Організацію самостійної роботи можна умовно розділити на три етапи:

- планування навчальної діяльності та її методична підготовка;
- здійснення цієї діяльності та її супровід;
- контроль, аналіз результатів (з можливими змінами в плануванні самостійної роботи).

Рекомендації щодо використання матеріалів навчально-методичного комплексу навчальної дисципліни

Зміст вивчення дисципліни "Операційні системи та основи системного програмування" визначено її робочою програмою.

Інформативну частину навчання складають навчальні посібники, конспекти лекцій у паперовій та електронній формі, план, зміст та методичні рекомендації до проведення лабораторних занять, методичні рекомендації до виконання самостійної роботи, перелік рекомендованої до вивчення літератури, ресурси мережі Інтернет.

У рекомендаціях до виконання індивідуального розрахункового завдання з дисципліни "Операційні системи та основи системного програмування" містяться варіанти індивідуальних завдань та перелік питань для самостійного опрацювання матеріалу. Також зазначається короткий теоретичний коментар до кожної теми, що допомагає студентові ознайомитися із сутністю питань, на основі яких базується виконання завдань.

1.3 Рекомендації щодо роботи з літературою

Найважливішим інформаційним джерелом вивчення навчальної дисципліни «Операційні системи та основи системного програмування» є ресурси мережі Інтернет. Основна частина матеріалу в Інтернеті розрахована на професіоналів, тому при вивченні навчальної дисципліни спочатку необхідно користуватися літературою навчального характеру.

При опрацюванні матеріалу потрібно дотримуватись таких правил:

1. Зосередитися на тому, що читаєш.
2. Виділити головну думку автора.
3. Виділити основні питання тексту від другорядних.
4. Зрозуміти думку автора чітко і ясно, що допоможе виробити власну думку.
5. Уявити ясно те, що читаєш.

У процесі роботи над темою тлумачення незнайомих слів і спеціальних термінів слід знаходити у фаховій літературі, термінологічних словниках. Незрозумілі місця, фрази, вирази доречно перечитувати декілька разів, щоб зрозуміти їх зміст.

Після прочитання тексту необхідно:

1. Усвідомити зв’язок між теоретичними положеннями і практикою.
2. Закріпити прочитане у свідомості.
3. Пов’язати нові знання з попередніми у даній галузі.
4. Перейти до заключного етапу засвоєння і опрацювання – записам.

Записи необхідно починати з назви теми та посібника, прізвища автора, року видання та назви видавництва. Якщо це журнал, то рік і номер видання, заголовок статті. Після чого скласти план, тобто короткий перелік основних питань тексту в логічній послідовності теми.

Складання плану, або тез логічно закінченого за змістом уривка тексту, сприяє кращому його розумінню. План може бути простий або розгорнутий, тобто більш поглиблений, особливо при опрацюванні додаткової літератури за даною темою. Записи необхідно вести розбірливо і чітко. Вони можуть бути короткі або розгорнуті залежно від рівня знань студента, багатства його літературної і професійної лексики, навичок самостійної роботи з книгою.

Для зручності користування записами необхідно залишати поля для заміток і вільні рядки для доповнень. Записи не повинні бути

одноманітними. В них необхідно виділяти важливі місця, головні слова, які акцентуються різним шрифтом або різним кольором шрифтів, підкреслюванням, замітками на полях, рамками, стовпчиками тощо. Записи можуть бути у вигляді конспекту, простих або розгорнутих тез, цитат, виписок, систематизованих таблиць, графіків, діаграм, схем.

1.4 Поради із підготовки до поточного, проміжного та підсумкового контролю

Контрольні заходи включають поточний і підсумковий контроль знань студентів. Поточний контроль є органічною частиною навчального процесу і проводиться під час лекцій та лабораторних занять.

Форми поточного контролю:

- усна співбесіда за матеріалами розглянутої теми на початку лабораторного заняття з оцінкою відповідей студентів (5-10 хв.);
- письмове фронтальне опитування студентів на початку чи в кінці лабораторного заняття (5-10 хв.). Відповіді перевіряються і оцінюються викладачем у поза аудиторний час;
- перевірка виконання завдань лабораторних робіт;
- тестова перевірка знань студентів;
- модульний контроль;
- інші форми.

При кредитно-модульній системі навчання теми самостійної роботи входять у модуль, який контролюються після закінчення логічно завершеної частини лекцій та інших видів занять з дисципліни та їх результати враховуються при виставленні підсумкової оцінки.

2. ОСНОВНІ ТЕОРЕТИЧНІ ВІДОМОСТІ ЩОДО ВИКОНАННЯ ІНДИВІДУАЛЬНОГО РОЗРАХУНКОВОГО ЗАВДАННЯ

2.1 Передача параметрів командного рядка до програми. Робота із процесами. Отримання системної інформації

2.1.1 Передача параметрів командного рядка до програми.

Параметри командного рядка в С передаються через аргументи функції main(), яка може мати синтаксис:

```
int main (int argc, char *argv[]) { ... }
```

argc - задає число параметрів командного рядка (вважаючи саме ім'я програми, тобто якщо програма запущена без параметрів, argc дорівнюватиме 1, якщо з одним параметром, то 2 і т.д.)

argv - це символьний масив, у якому містяться аргументи командного рядка (argv[0] - (зазвичай) ім'я самої програми, argv[1] – перший аргумент тощо.).

Наприклад, під час запуску

```
./lab2 file1 file2
```

Argc 0 дорівнюватиме 3, а **argv** – масиву { "./lab2", "file1", "file2" }

2.1.2. Обробка параметрів командного рядка Linux.

У випадку, коли аргументи командного рядка є, наприклад, іменами файлів, їхня обробка не є складною. Значно більш трудомісткою вона стає тоді, коли програма запускається з прапорами або іменованими аргументами (наприклад, як компілятор gcc -g -o ім'я-виконуваного файлу file.c), оскільки потрібно окремо відстежити, чи задані ті чи інші опції (можливо з параметрами) і т.ін.

Для полегшення обробки в UNIX системах існує функція getopt(), яка здійснює синтаксичний аналіз параметрів командного рядка.

Синтаксис:

```
int getopt (int argc, char *argv[], char *options);
```

Перші два аргументи - це argc і argv, отримані в main(). Третій параметр задає список допустимих (односимвольних) опцій командного рядка. Якщо опція вимагає після себе параметра (як -o для компілятора), то після відповідної літери ставиться ':', наприклад

```
getopt (argc, argv, "lf: h");
```

показує, що програма може викликатися як

```
lab2 -l
lab2 -h
lab2-lh
lab2 -f text.c
lab2 -ftext.c
lab2 -lf text.c
```

getopt() за один раз повертає лише одну (чергову) опцію (у вигляді символу, наприклад, 'l' при виклику lab2 -l), тому її потрібно викликати у циклі. Цикл продовжується доти, доки функція не поверне -1 (EOF), що означає те, що всі опції оброблені. Якщо якась опція потребує параметра (як -f), його значення повертається в зумовленої зовнішньої рядкової змінної optarg. Ще одна зовнішня змінна optind після виходу з циклу дорівнюватиме тому, скільки опцій було задано:

```
1 для lab2 -l
3 для lab2-lhf test1.c
```

Приклад обробки опцій за допомогою getopt() представлений нижче

```
int main (int argc, char *argv[]) {
int c;
while (1) {
c = getopt(argc, argv, "hlf:");
if (c == -1) break;
switch (c) {
case 'h':
printf ("використання: %s [-h] [-l] [-f ім'я]\n",
argv[0]);
break;
case 'l':
printf ("задано -l\n");
break;
case 'f':
printf ("задано -f з параметром: %s\n", optarg);
break;
}
}
}
```

Якщо жодної опції не встановлено, getopt() відразу поверне -1. Якщо встановлені помилкові опції (наприклад, lab2 -k) або опущені параметри при опціях (lab2 -f), видається повідомлення про помилку. Якщо задані додаткові аргументи командного рядка (не опції), наприклад,

```
lab2 -f test1.c test.c
```

після того, як getopt() поверне -1, можна буде отримати до цих аргументів доступ, наприклад, так:

```
if (optind < argc)
{printf ("додаткові аргументи:");
for (int i=optind; i < argc; i++)
printf ("%s", argv[i]);
}
```

2.1.3. Перенаправлення введення-виводу та канали.

У UNIX є можливість будь-яку програму, яка пише на стандартний висновок або чекає на дані зі стандартного введення, змусити працювати з файлами або з іншими такими ж програмами як джерела або приймачі даних.

Для цього використовуються операції перенаправлення введення-виводу та канали. Перенаправлення введення-виводу здійснюється за допомогою операцій

```
> - висновок
< - введення
>> - Висновок з додаванням до кінця (append).
```

Так, наприклад, можна написати так:

```
ls > a.txt або ls >> a.txt
```

і результат виведення опиниться у файлі a.txt (знищивши те, що в ньому було до того при > або додавши в кінець при >>)

```
sort < a.txt введення піде з файлу a.txt, а не з клавіатури.
```

Стандартний потік stderr (повідомлення про помилки) не перенаправляється. Для його перенаправлення використовують такий синтаксис

```
ls 2> a.txt - тільки stderr
ls > a.txt 2>&1 - stderr і stdout разом.
```

Якщо ми не хочемо, щоб висновок взагалі йшов кудись, його можна перенаправити на стандартний пристрій "битовий кошик" /dev/null

```
ls > /dev/null
```

Весь висновок пропаде. Канал (операція |) дозволяє направляти виведення однієї програми на вхід інший, наприклад

```
ls | grep cc
```

видасть усі рядки виведення ls, які містять cc (grep фільтрує вхід, залишаючи лише рядки, що містять задані символи).

Канал може складатися з кількох програм

```
ls | grep cc | sort
```

Програма, яка чекає на свій висновок тільки зі стандартного введення і пише тільки на стандартний висновок, у термінології UNIX називається фільтром. Потужність та гнучкість UNIX пов'язана з тим, що в ньому є безліч фільтрів для різних цілей і механізми для зв'язку їх один з одним (в першу чергу, канали і переадресація введення-виведення).

Серед фільтрів:

sort- Сортує введення
wc- підраховує кількість слів у введенні
grep- фільтрує введення на основі шаблону
tail, head- видають останні чи перші n рядків введення
perl- Інтерпретатор потужної мови Perl теж може використовуватися як фільтр.

Ще одна цікава особливість UNIX - можливість підстановки в командний рядок результатів виведення програми за допомогою операції (т.зв. лапки, backticks).

Ось приклад: програма, якаповертає повний шлях до виконуваного файлу, якщо він є в path, тобто. which sort поверне /usr/bin/sort

Припустимо, що ми хочемо подивитися характеристики файлу для sort, що виконується, але не пам'ятаємо, де він знаходиться. Тоді можна написати так:

```
ls -l `which sort`
```

Спочатку виконається команда which sort, та був її результат підставиться командний рядок для ls -l, тобто. далі виконається

```
ls -l /usr/bin/sort
```

з результатом

```
-rwxr-xr-x 1 root bin 35560 Feb 2 1997 /usr/bin/sort*
```

2.2 Основи роботи з процесами в Linux.

2.2.1. Середа виконання процесу

Процеси в UNIX-подібних системах можуть мати доступ до різних характеристик середовища виконання. Серед них:

1) параметри командного рядка, описані вище (argc та argv)

2) змінні оточення (environment variables). Ці змінні мають вигляд name=value і доступні всередині програми (наприклад, HOSTNAME=asu).

Особливо важлива змінна PATH, в якій міститься список каталогів (розділених двокрапками ":"), в яких система повинна шукати файли, що виконуються, якщо до них не вказаний повний шлях.

Для того, щоб переглянути змінні оточення, використовується команда set.

Для доступу до оточення у програмі використовується зумовлена змінна environ, яка описана так:

```
extern char **environ;
```

і є масив покажчиків на рядки, останній з яких дорівнює NULL. Так можна побачити все оточення:

```
for (char **e = environ; *e; e++)
printf ("%s", * e);
```

Для доступу до окремих змінних або їх встановлення використовуються функції

```
char * getenv (const char * name)
```

і

```
int putenv(const char *string),
```

описані в ***stdlib.h***:

```
char *hostname = getenv("HOSTNAME");
putenv ("MYVAR = myvalue");
```

3) різні ідентифікатори, пов'язані з процесом (ідентифікатор користувача, під яким він запущений – uid, групи цього користувача – gid, ідентифікатор процесу – pid, ідентифікатор предка – ppid тощо).

Для доступу до цих ідентифікаторів використовуються відповідні функції, для роботи з якими потрібно підключати заголовний файл ***unistd.h***(Типи uid_t і т.п. описані там же і позначають цілі).

Процес може дізнатися про свій ідентифікатор (PID), а також батьківський ідентифікатор (PPID) за допомогою системних викликів getpid() і getppid().

Системні виклики getpid() та getppid() мають такі прототипи:

```
pid_t getpid (void);
pid_t getppid (void);
```

Для використання getpid() і getppid() у програму повинні бути включені директивою #include заголовні файли **unistd.hisys/types.h**(Для типу pid_t). виклик getpid() повертає ідентифікатор поточного процесу (PID), а getppid() повертає ідентифікатор батька (PPID). pid_t це цілий тип, розмірність якого залежить від конкретної системи. Значеннями цього можна оперувати як звичайними цілими числами типу int.

Розглянемо тепер просту програму, яка виводить на екран PID і PPID, та був " завмирає " до того часу, поки користувач не натисне <Enter> (рис.2.1).

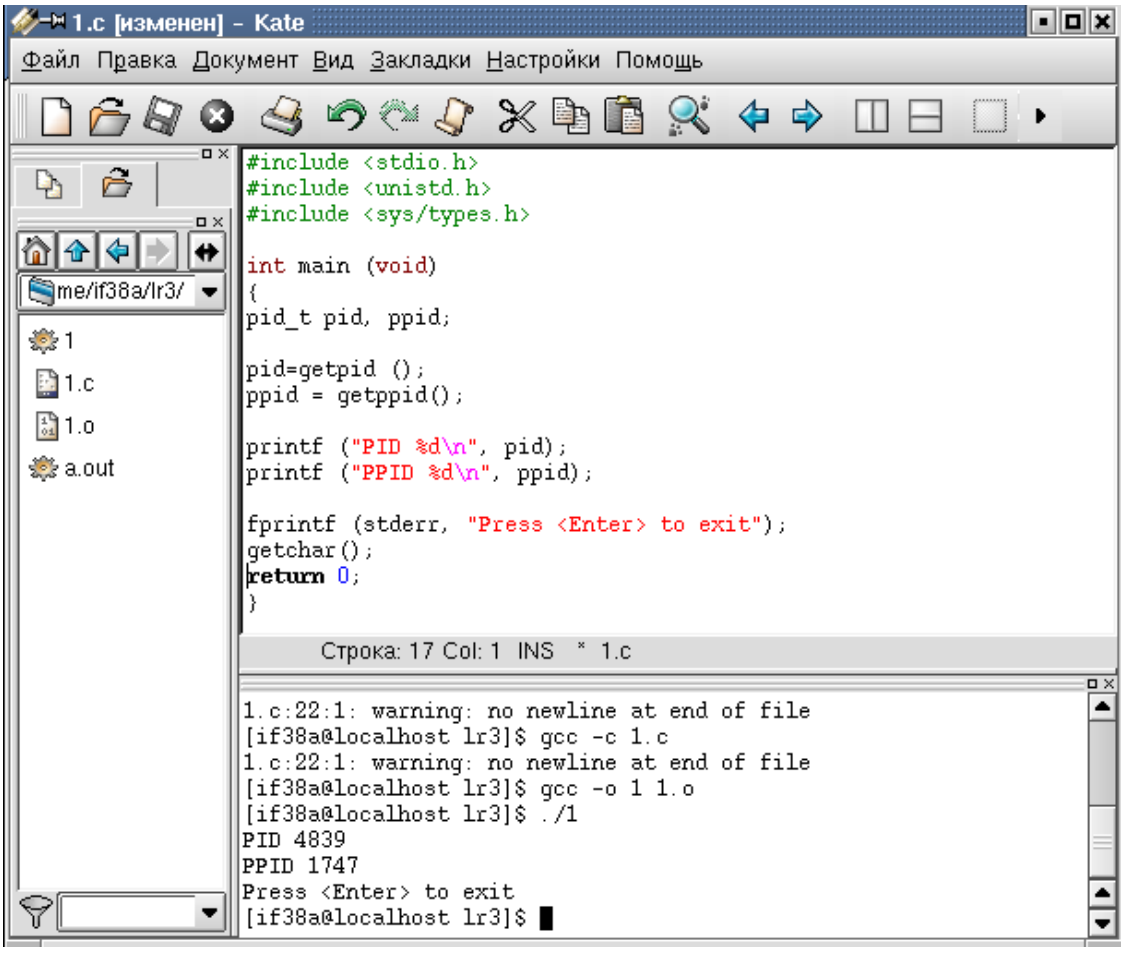


Рисунок 2.1. Програми перегляду інформації про процес

2.2.1.1. Доступ до інформації про користувачів та групи з файлів /etc/passwd та /etc/group

Інформація про користувачів UNIX міститься у файлі /etc/passwd. Рядок цього файлу має вигляд:

```
ім'я-користувача:пароль-або-x:uid:gid:відомості:home-каталог:shell
```

Пароль міститься в зашифрованому вигляді або як x, що означає, що він знаходиться в іншому місці. uid і gid визначають чисельні ідентифікатори користувача та його групи (кожен користувач належить до певної групи, наприклад код 102 позначає групу students).

Відомості позначають додаткову інформацію про користувача (його повне ім'я тощо)

home-каталог позначає розташування home-каталогу користувача.

Shell позначає версію командного інтерпретатора, яка запускається користувача, коли він входить у систему. Стандартний shell у Linux - /bin/bash (є й інші, наприклад, C shell - tcsh)

Приклад:

```
i38a1:x:540:102:i38a - варіант 1:/usr/students/i38a1:/bin/bash
```

Інформація про групи міститься у файлі /etc/group.

Рядок цього файлу має вигляд:

```
ім'я-групи:пароль-групи:gid:список-членів
```

пароль зазвичай опускається.

gid- чисельний ідентифікатор групи (той, що використовується в /etc/passwd)

список-членів- Список всіх користувачів, які входять до групи, розділений комами.

До групи входять: а) всі користувачі, для яких gid в/etc/passwdдорівнює gid групи та б) усі користувачі зі списку список-членів з файлу/etc/group

Приклад:

```
students::102:i38a1,i38a2,...
```

Для доступу до інформації з файлу/etc/passwd можна використовувати такі функції (з**pwd.h**):

```
struct passwd *getpwuid(uid_t uid);
```

дає доступ до інформації для заданого uid

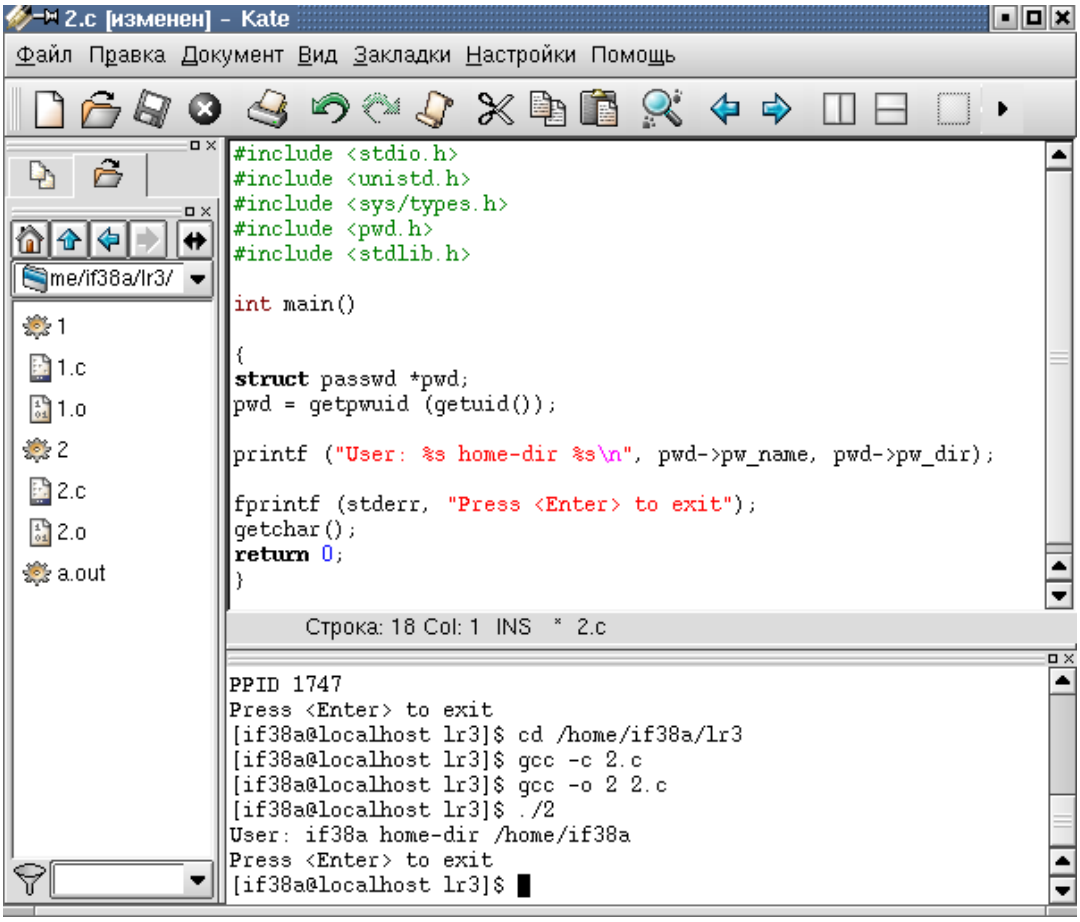
```
struct passwd *getpwnam(char *name);
```

дає доступ до інформації для заданого імені користувача

Обидві функції повертають покажчик на системну структуру passwd, описану в pwd.h. У ній є такі поля:

```
char * pw_name; // Ім'я користувача
char * pw_passwd; // Зашифрований пароль (або x)
uid_t pw_uid; // uid
gid_t pw_gid; // gid
char * pw_gecos; // відомості
char * pw_dir; // home-каталог
char * pw_shell; // shell
```

Приклад програми, яка повертає інформацію про користувача, його home-каталогу (рис.2.2).



Рисукнок 2.2. Програма виведення інформації про користувачів.

Відводити пам'ять наперед під структуру не треба. Аналогічно можна отримувати доступ до елементів файлу `/etc/group`.

```
#include<grp.h>
…
struct group * getgrnam (const char * name);
```

дає доступ до інформації для заданого імені групи

```
#include<grp.h>
…
struct group *getgrgid(gid_t gid);
```

дає доступ до інформації для заданого gid

```
struct group містить такі поля:
char *gr_name; // ім'я групи
char *gr_passwd; // пароль
gid_t gr_gid; // gid
char **gr_mem; // Список-членів (масив рядків)
```

Приклад програми:

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

int main()
{
struct group *grp;
grp = getgrgid(getgid());
printf ("user's group:%s\n", grp->gr_name);
return 0;
}
```

2.2.1.2. Обмеження на ресурси

```
getrlimit, setrlimit, prlimit –
```

зчитує/встановлює обмеження використання ресурсів

Приклад програми:

```
#include <sys/time.h>
#include <sys/resource.h>

int getrlimit(int resource, struct rlimit *rlim);
int setrlimit (int resource, const struct rlimit * rlim);

int prlimit(pid_t pid, int resource, const struct rlimit
*new_limit,
struct rlimit *old_limit);
```

Системні виклики getrlimit() та setrlimit(), відповідно, отримують та встановлюють обмеження використання ресурсів. Кожному ресурсу призначається м'яке та жорстке обмеження, яке визначається структурою rlimit:

```
struct rlimit {
rlim_t rlim_cur; /* м'яке обмеження */
rlim_t rlim_max; /* жорстке обмеження (максимум для
rlim_cur) */
};
```

М'яким обмеженням є значення, яке примусово встановлюється ядром для відповідного ресурсу. Жорстке обмеження працює як максимальне значення для м'якого обмеження: непривілейовані процеси можуть визначати лише свої м'які обмеження в діапазоні від 0 до жорсткого обмеження, тобто однозначно менше за жорстке обмеження.

Ресурсами можуть бути:

```
RLIMIT_CPU – на процесорний час
RLIMIT_FSIZE – на максимальний розмір файлу
RLIMIT_NPROC - на кількість процесів
RLIMIT_NOFILE - на кількість відкритих файлів
```

Приклад:

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

int main()
{
struct rlimit flim;
getrlimit(RLIMIT_NOFILE, &flim);
printf ("max open files:%d\n", flim.rlim_max);
return 0;
}
```

2.2.2. Управління процесами

Насамперед слід зазначити, що у UNIX створення нового процесу завантаження програми відокремлені друг від друга. Для створення

процесу використовується виклик fork(), а для завантаження програми – сімейство функцій exec().

2.2.2.1. Клонування процесів та fork()

Єдиний спосіб створити новий процес у UNIX-подібних системах – за допомогою знаменитого системного виклику fork().

Синтаксис**:**

```
#include <sys/types>
#include <unistd.h>
pid_t fork (void);
```

pid_t є примітивним типом даних, що визначає ідентифікатор процесу чи групи процесів.

При виклику fork() породжується новий процес (процес-нащадок), який майже ідентичний процесу-батькові, що породжує. Процес-нащадок успадковує такі ознаки батька:

- сегменти коду, даних та стеку програми;
- таблицю файлів, в якій знаходяться стани прапорів дескрипторів файлу, що вказують, чи читається чи пишеться файл.
- Крім того, у таблиці файлів міститься
- поточна позиція покажчика запису-читання;
- робочий та кореневий каталоги;
- реальний та ефективний номер користувача та номер групи;
- пріоритети процесу (адміністратор може змінити їх через nice);
- контрольний термінал;
- маску сигналів;
- обмеження щодо ресурсів;
- відомості про середовище виконання;
- сегменти пам'яті, що розділяються.

Нащадок не успадковує від батька наступних ознак:

- ідентифікатор процесу (PID, PPID);
- витраченого часу ЦП (воно обнулюється);
- сигналів процесу-батька, які потребують відповіді;

– блокованих файлів (record locking).

При виклику fork() виникають два повністю ідентичні процеси. Весь код після fork() виконується двічі, як у процесі-нащадку, так і в процесі-батьку.

Процес-нащадок і процес-батько отримують різні коди повернення після виклику fork(). Процес-батько отримує ідентифікатор (PID) нащадка. Якщо це значення буде негативним, отже, при породженні процесу відбулася помилка. Процес-нащадок отримує як код повернення значення 0, якщо виклик fork() виявився успішним (рис.2.3).

Таким чином, можна перевірити, чи було створено новий процес:

```
switch(ret=fork())
{
case -1: /{*}при викликі fork() виникла помилка{*}/
case 0 : /{*}це код нащадка{*}/
default : /{*}це код батьківського процесу{*}/
}
```

Приклад породження процесу через fork() наведено нижче:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{pid_t pid;
if ((pid = fork())==-1)
{printf ("fork error\n"); exit(-1);
}
if (pid == 0)
{// це нащадок
}
else
{// це предок
printf ("нащадок запущений з кодом %d\n", pid);
}
return 0;}
```

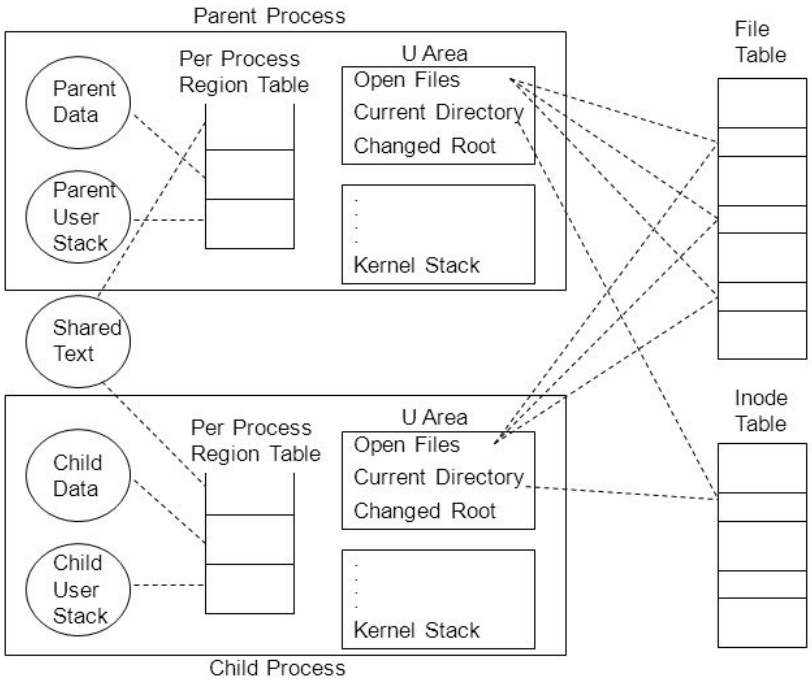


Рисунок 2.3 - Створення контексту нового процесу під час виконання функції fork

Хто перший виконає цю наступну інструкцію, не визначено (на машинах із кількома процесорами це може бути й одночасно).

Нащадок і предок можуть йти своїми шляхами відповідно до логіки програми. У нащадку доступні такі властивості, як ppid - ідентифікатор предка (за допомогою функції getppid()). Предок та його нащадки зазвичай становлять одну групу процесів з єдиним pgid - ідентифікатором групи процесів. Предок і нащадок поділяють такі характеристики, як дескриптори відкритих файлів (тобто вони можуть писати в той самий файл), поточний каталог, змінні оточення і т.д.

Предок може завершити виконання до нащадка, у разі предком останнього стає init (тобто. ppid == 1).

Так звані процеси-зомбі виникають, якщо нащадок завершився, а батьківський процес не викликав wait(). Для завершення процесів використовують оператор повернення, або виклик функції exit() зі значенням, яке потрібно повернути операційній системі. p align="justify"> Операційна система залишає процес зареєстрованим у своїй внутрішній таблиці даних, поки батьківський процес не отримає коду повернення нащадка, або не закінчиться сам. У разі процесу-зомбі його код повернення не передається батькові, і запис про цей процес не видаляється з таблиці процесів операційної системи. При подальшій

роботі та появі нових зомбі таблиця процесів може бути заповнена, що призведе до неможливості створення нових процесів.

Ми вже бачили у прикладі, як завершувати процес. І тому існує два шляху: повернення з main() - return 0; та виклик exit().

Синтаксис

```
void exit (int status);
```

де status визначає код повернення.

exit (-1); // вихід із кодом повернення -1

2.2.2.2. Запуск інших програм (функції сімейства exec())

Зазвичай породження нових процесів пов'язане із завантаженням інших програм із файлів на диску. У UNIX, як згадувалося, ці дві функції розділені. Завантаження програм на згадку здійснюється з допомогою сімейства функцій exec().

Таких функцій кілька, вони різняться методами передачі властивостей, але не важливо (в Linux це різні методи доступу до того самого системного виклику execve()). Серед них виділяються дві групи: execvx() та execlx(), де x може бути p, e або порожньо. p означає те, що пошук здійснюється в каталогах, зазначених у PATH, якщо не заданий повний шлях, e - що додатково передається покажчик на набір змінних оточення (тобто можна задати нове оточення для програми, що завантажується).

Групи execvx() і execlx() відрізняються передачею параметрів програмі, що завантажується (execvx() - через масив, execlx() - через список параметрів змінної довжини, як у printf()).

Ми розглянемо групу execvx() з прикладу функції execvp().

Синтаксис:

```
int execvp (const char * path, const char * argv []);
```

Першим параметром іде ім'я файлу програми (шлях або просто ім'я, у другому випадку пошук буде зроблено в каталогах, вказаних у PATH).

Другим параметром йде масив аргументів командного рядка (покажчиків на рядки), останнім елементом цього масиву має йти NULL.

Основна особливість функцій exec() полягає в тому, що після завантаження в пам'ять програма, що викликається, повністю заміщає процес, що викликав exec(), причому pid залишається тим же, тобто. новий процес не створюється, а нова програма завантажується в адресний простір поточного процесу. Отже, якщо fork() відразу два виходу, то exec() при успішному виконанні - жодного, тобто. управління будь-коли повернеться у викликає процес, т.к. він вже не знаходиться в пам'яті – на його місці новий, завантажений у exec().

Повернення з exec() буде лише в тому випадку, якщо завантажити програму не вдалося (наприклад, файл не знайдено). І тут функція поверне -1.

Приклад формування параметрів та виклику:

```
if (execvp (prog, args) == -1) {
printf ("error executing the program: %s\n", prog);
exit (-1);
}
```

Хоча створення процесів - fork() і завантаження програм - exec() у UNIX розділені, вони часто використовуються разом. Технологія така:

1) предок створює нащадка за допомогою fork()

2) нащадок відразу завантажує іншу програму замість себе за допомогою exec(). Така технологія називається fork-and-exec.

Приклад коду:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
pid_t pid;

if ((pid = fork()) < 0) exit (-1);
if (pid == 0) { // Нащадок завантажує замість себе іншу
програму
     char *prog = "./child";
     char *args[] = {"./child", "arg1", NULL};
     if (execvp (prog, args) == -1) {
          printf("error starting the child\n");
          exit (-1);
     }
}
else {
// предок продовжує роботу (наприклад, чекає закінчення
нащадка)
printf ("нащадок запущений з кодом %d\n", pid);
}


return 0;
}
```

Предок зазвичай чекає закінчення нащадка з допомогою wait() чи waitpid() - див. 2.2.2.3.

2.2.2.3. Синхронне виконання, очікування, wait() та waitpid().

Дуже часто предок, запустивши нащадка, хоче дочекатися його завершення, а потім продовжити своє виконання (як при виклику процедури в мовах програмування), можливо, отримавши деяку

інформацію про нащадок, що завершився (наприклад, код повернення). Таке виконання ще називають синхронним виконанням чи виконанням з очікуванням.

Синтаксис:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int * status);
pid_t waitpid(pid_t pid, int * status, int options);
```

Функція wait зупиняє виконання поточного процесу доти, доки дочірній процес не завершиться, або до появи сигналу, який або завершує поточний процес, або вимагає викликати функцію-обробник. Якщо дочірній процес на момент виклику функції вже завершився (так званий "зомбі" ("zombie")), то функція негайно повертається. Системні ресурси, пов'язані з дочірнім процесом, звільняються. Функція waitpid призупиняє виконання поточного процесу доти, доки дочірній процес, вказаний у параметрі pid, не завершить виконання, або доки з'явиться сигнал, який або завершує поточний процес або вимагає викликати функцию-обработчик. Якщо цей дочірній процес до моменту виклику функції вже завершився (так званий "зомбі"), то функція негайно повертається. Системні ресурси,

Параметр pid може набувати кількох значень:

– < -1 означає, що слід чекати будь-якого дочірнього процесу, ідентифікатор групи процесів якого дорівнює абсолютному значенню pid.

– -1 означає очікування будь-якого дочірнього процесу; функція wait веде себе так само.

– 0 означає очікування будь-якого дочірнього процесу, ідентифікатор групи процесів якого дорівнює ідентифікатору поточного процесу.

– 0 означає очікування дочірнього процесу, ідентифікатор якого дорівнює pid.

Значення options створюється шляхом логічного складання кількох наступних констант:

– **WNOHANG** означає негайне повернення управління, якщо жоден дочірній процес не завершив виконання.

– **WUNTRACED** означає повернення управління та для зупинених (але не відстежуваних) дочірніх процесів, про статус яких ще не було повідомлено. Статус для зупинених підпроцесів, що відстежуються, також забезпечується без цієї опції. Якщо status не дорівнює NULL, то функції wait і waitpid зберігають інформацію про статус змінної, яку вказує status. Цей статус можна перевірити за допомогою нижченаведених макросів (вони приймають як аргумент буфер (типу int), --- а не покажчик на буфер!):

– **WIFEXITED(status)** не дорівнює нулю, якщо дочірній процес успішно завершився.

– **WEXITSTATUS(status)** повертає вісім молодших бітів значення, яке повернув дочірній процес, що завершився. Ці біти могли бути встановлені в аргументі функції exit() або аргументі оператора return функції main(). Цей макрос можна використовувати лише тоді, коли WIFEXITED повернув ненульове значення.

– **WIFSIGNALED(status)** повертає справжнє значення, якщо дочірній процес завершився через необроблений сигнал.

– **WTERMSIG(status)** повертає номер сигналу, який спричинив завершення дочірнього процесу. Цей макрос можна використовувати лише тоді, коли WIFSIGNALED повернув ненульове значення.

– **WIFSTOPPED(status)** повертає справжнє значення, якщо дочірній процес, через який функція повернула управління, зараз зупинено; це можливо, лише якщо використовувався прапор WUNTRACED або коли підпроцес відстежується.

– **WSTOPSIG(status)**повертає номер сигналу, через який дочірній процес було зупинено. Цей макрос можна використовувати лише тоді, коли WIFSTOPPED повернув ненульове значення.

Приклад синхронного виконання з очікуванням:

```
pid_t pid;
if ((pid = fork())==-1) { printf ("fork error\n"); exit(-
1); }
if (pid == 0) {
// Нащадок - виклик exec()
}
else {
// предок - Чекати нащадка
int status;
waitpid (pid, &status, 0);
// продовжувати виконання
}
```

Приклад того, як отримати код повернення нащадка:

```
waitpid (pid, &status, 0);
if (WIFEXITED(status))
printf("нащадок завершився з кодом %d\n",
WEXITSTATUS(status));
```

Якщо exec() завершився невдало, має сенс повертати з нащадка -1 і перевіряти це значення так:

```
waitpid(pid, &status, 0);
if (WIFEXITED(status) && ((k = WEXITSTATUS(status)) !=
255))
printf("нащадок завершився з кодом %d\n", k);
```

Приклад породження процесу через fork() наведено нижче:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
pid_t pid;
int rv;
switch(pid=fork()) {
case -1:
perror("fork"); /* Виникла помилка */
exit(1); /*вихід із батьківського процесу*/
case 0:
printf(" CHILD: Це процес-нащадок!\n");
printf(" CHILD: Мій PID - %d\n", getpid());
printf(" CHILD: PID мого батька -- %d\n",
getppid());
printf(" CHILD: Введіть мій код для повернення
(якомога менше):");
scanf("%d");
printf("CHILD: Вихід!\n");
exit(rv);
default:
printf("PARENT: Це процес-батько!\n");
printf("PARENT: Мій PID - %d\n", getpid());
printf("PARENT: PID мого нащадка %d\n", pid);
printf("PARENT: Я чекаю, поки нащадок
не викличе exit()...\n");
wait();
printf("PARENT: Код повернення нащадка:%d\n",
WEXITSTATUS(rv));
printf("PARENT: Вихід!\n");
}
}
```

Перегляд процесів у системі командою ps. Для перегляду процесів UNIX використовується команда ps. Вона може викликатися так:

```
ps -aux або ps -jx або ps
```

u та **j** - основні формати виведення,
a - вивести процеси всіх користувачів, а не тільки поточного,
x- вивести та процеси, для яких немає керуючого терміналу

У форматі **u** висновок буде таким:

```
USER PID %CPU %MEM SIZE RSS TTY STAT START TIME COMMAND
shekvl 123 0.0 2.2 1200 684 1 S 23:38 0:00 -bash
```

- **PID** - код процесу,
- **%CPU** - відсоток процесорного часу,
- **%MEM** - відсоток пам'яті,
- **SIZE** - розмір в Kb,
- **RSS** - розмір у пам'яті в даний час,
- **TTY** - контролюючий термінал,
- **START** - час запуску,
- TIME - загальний час, який виконується процес,
- **STAT**- стан процесу (S - очікування (sleeping), R - виконання (running), Z - зомбі, T - перерваний (terminated))

У форматі **j** висновок буде таким:

```
PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND
1 123 123 123 1 139 S 501 0:00 -bash
```

- **PPID**- код предка,
- **PGID**- код групи процесів,
- **SID**- код сесії,
- **TPGID**- код процесу, що контролює термінал.

Без формату (просто ps) висновок буде таким:

```
PID TTY STAT TIME COMMAND
123 1 S 0:00 -bash
```

Для нашої роботи нам найзручніше працювати з форматом –jx (т.к. нам потрібна інформація

1) про коди предків-нащадків

2) у тому числі про демони теж.

Для зручності отримання інформації про той чи інший процес можна обмежувати виведення ps, переправляючи його каналом в grep:

```
ps-jx | grep 'PID\|parent'
```

відфільтрує лише рядки, що містять PID (тобто заголовок) або parent (тобто що відносяться до цього додатку).

2.2.2.4. Асинхронне виконання та зомбі.

Якщо предок викликає wait() чи waitpid() своїх нащадків, він припиняє своє виконання, доки закінчиться виконання нащадків. Якщо нам потрібно запустити нащадок асинхронно (тобто. продовжити виконання предка, не чекаючи завершення нащадка, наприклад, як із запуску у фоновому режимі з shell: $ls -l&), то природним видається просто викликати wait(), а продовжувати виконання далі:

```
if ((pid = fork()) < 0) exit (-1);
if (pid == 0) {
// нащадок - запустити по exec()
}
else {
// предок - продовжувати виконання
for(; ;);
}
```

Але у випадку, коли предок продовжує свою роботу (як у нашому випадку) і коли може статися так, що він буде продовжувати її довше за нащадка, так і не викликавши для нього wait() або waitpid(), є ймовірність того, що нащадок перетвориться у процес-зомбі (zombie).

Процес перетворюється на зомбі у разі, що він завершив своє випонування, та його предок не викликав йому wait() чи waitpid(), тобто. не рахував інформацію про нього з таблиці процесів. Такого процесу в системі немає (він завершений), є тільки інформація про нього в системній таблиці процесів, яка чекає на те, що її хтось вважає, викликавши wait() або waitpid().

Приклад створення зомбі:

```
#include <unistd.h>
main () {
int pid;
if ((pid = fork()) < 0) exit(-1);
if (pid == 0) exit (0); // Нащадок
else
for(; ;); // предок
}
```

Тут нащадок завершує свою роботу негайно, а предок залишається в нескінченному циклі.

Приклад запуску програми:

```
$ ./myzombie &

ps-j | grep 'PID\|myzombie'
```

Результат виконання

```
PPID PID PGID SID TTY TPGID STAT UID TIME COMMAND
236 2430 2430 236 p1 2437 R 501 0:26 ./myzombie
2430 2431 2430 236 p1 2437 Z 501 0:00 (myzombie <zombie>)
```

Нащадок, як бачимо, став зомбі.

Як уникнути появи зомбі? Для цього потрібно використовувати той факт, що прямі нащадки init зомбі ніколи не стають – init автоматично зчитує їхню інформацію після завершення. Тому якщо асинхронно занедбаний нащадок став нащадком init, то зомбі він не стане.

Для цього використовується технологія подвійного fork(). Справа в тому, що процес, чий предок завершив виконання, стає нащадком init (ppid = 1). Тому дії відбуваються так:

```
if ((pid = fork()) < 0) exit (-1); // (1) перший fork()
if (pid == 0) { // Нащадок 1
if ((pid2 = fork()) < 0) exit (-1); // (3) другий fork()
if (pid2 == 0) { // (5) нащадок 2
// запуск exec() асинхронно
}
else { // (4) нащадок 1, він же предок 2
printf ("спроба запуску нащадка з pid=%d\n", pid2);
exit(0); // вихід із нащадка 1
}
}
else { // предок 1
waitpid(pid, NULL, 0); // (2) очікування завершення
нащадка 1
}
```

- предок1 викликає fork()
- після fork() предок1 чекає завершення потомка1, а потомок1 ще раз викликає fork()
- після другого fork() нащадок1 стає предком2, зчитує pid2 і завершує виконання (при цьому waitpid(), який на нього чекає на кроці (2) спрацьовує і зомбі не буде)
- а потомок2 запускає процес exec(). Оскільки його предок завершився (на кроці (4)), то предком цього процесу (потомок2) стає init (ppid==1) і зомбі він вже не зможе.

3. ЗМІСТ ІНДИВІДУАЛЬНОЇ РОЗРАХУНКОВОЇ РОБОТИ СТУДЕНТА І МЕТОДИЧНІ РЕКОМЕНДАЦІЇ ЩОДО ЇЇ ВИКОНАННЯ

3.1 Основне завдання

1. Розробити дві програми для Linux, перша з яких:

а) при завданні ключа -w ім'я-виконуваного-файлу - запускає заданий додаток (з пошуком його в системному шляху, заданому змінною оточення PATH) - чекає завершення програми та виводить повідомлення про завершення на екран разом із кодом повернення

б) при завданні ключа -f ім'я-виконуваного файлу - запускає заданий додаток (з пошуком його в системному шляху, заданому змінною оточення PATH) - негайно продовжує виконання (асинхронний запуск)

Другий додаток має виводити на екран:

- значення аргументів командного рядка;
- ідентифікатори: процесу (pid), сесії (sid), групи процесів (pgid)
- та процесу-предка (ppid)
- ім'я користувача, під яким процес запущено та його групу
- (отримані з файлів /etc/passwd та /etc/group)

3.2 Варіанти індивідуальних завданнь

Варіант №1

Розробити дві програми. Перша обчислює суму та добуток чисел відL до U, де L – це нижня межа діапазону, U – верхня межа діапазону, межі вводяться користувачем, та виводить отримані значення на екран. Друга програма запускає першу як новостворений процес.

Варіант №2

Розробити дві програми. Перша обчислює число Фібоначчі за номером, введеним користувачеві, та формулоюFi = Fi-1 + Fi-2, F0 = F1 = 1 і виводить його на екран. Друга програма запускає першу як новостворений процес.

Варіант №3

Розробити дві програми. Перша приймає від користувача рядок, що зберігає ціле число знакового, і виводить на екран рядковий

еквівалент цього числа прописом (наприклад, введення «-1211» повинен приводити до висновку «мінус тисяча двісті одинадцять»). Друга програма запускає першу як новостворений процес.

Варіант №4

Розробити дві програми. Перша приймає від користувача рядок, що зберігає число зі знаком і плаваючою точкою, і виводить на екран рядковий еквівалент цього числа прописом (наприклад, введення «-12.11» має приводити до висновку «мінус дванадцять одинадцять сотих»). Друга програма запускає першу як новостворений процес.

Варіант №5

Розробити дві програми. Перша приймає від користувача два рядки. Далі, якщо обидві рядки зберігають цілі числа зі знаком, то екран виводиться сума чисел, інакше – конкатенація двох введених рядків. Друга програма запускає першу як новостворений процес.

Варіант №6

Розробити дві програми. Перша приймає від користувача дві прямокутні матриці, а потім виводить на екран їхню суму та добуток. Друга програма запускає першу як новостворений процес.

Варіант №7

Розробити дві програми. Перша приймає від користувача одновимірний цілий масив, упорядковує його за зростанням будь-яким з так званих «покращених алгоритмів» сортування масивів і виводить на екран. Друга програма запускає першу як новостворений процес.

Варіант №8

Розробити дві програми. Перша приймає від користувача одновимірний масив чисел з плаваючою точкою, впорядковує його за спаданням будь-яким з так званих «покращених алгоритмів» сортування масивів і виводить на екран. Друга програма запускає першу як новостворений процес.

Варіант №9

Розробити дві програми. Перша приймає від користувача одновимірний масив рядків, упорядковує його будь-яким із так званих «покращених алгоритмів» сортування масивів та виводить на екран. Друга програма запускає першу як новостворений процес.

Варіант №10

Розробити дві програми.Перша приймає від користувача квадратну матрицю, обчислює суму елементів, що лежать на головній

та побічній діагоналях, та виводить на екран. Друга програма запускає першу як новостворений процес.

Варіант №11

Розробити дві програми.Перша приймає від користувача квадратну матрицю, обчислює суму елементів, що не лежать на головній та побічну діагоналі, і виводить на екран. Друга програма запускає першу як новостворений процес.

Варіант №12

Розробити дві програми. Перша приймає від користувача дві дати – рядки виду ЦЦ..ЦЦЦЦ, де Ц - це будь-яка цифра з діапазону [0-9]. Далі вона обчислює повну кількість днів, що пройшли між двома введеними датами, та виводить його на екран. Друга програма запускає першу як новостворений процес.

Варіант №13

Розробити дві програми. Перша приймає від користувача два значення часу - рядки виду ЦЦ..ЦЦ, де Ц - це будь-яка цифра з діапазону [0-9]. Далі вона обчислює повну кількість секунд, що пройшли між двома значеннями часу і виводить його на екран. Друга програма запускає першу як новостворений процес.

Варіант №14

Розробити дві програми. Перша приймає від користувача два рядки, здійснює пошук входження другого рядка в першу будь-яким відомим методом, крім прямого, і виводить на екран значення індексу елемента першого рядка, з якого почався збіг, або -1 в іншому випадку. Друга програма запускає першу як новостворений процес.

Варіант №15

Розробити дві програми. Перша приймає від користувача два рядки, здійснює пошук кількості входжень другого рядка в перший будь-який відомий метод,крім прямого, та виводить на екран отримане значення. Друга програма запускає першу як новостворений процес.

4. КОНТРОЛЬНІ ПИТАННЯ

4.1 Контрольні питання до індивідуальної роботи №1

1. Який ключ використовується для синхронного запуску програми в першому додатку, і що має відбуватися після завершення її виконання?
2. Що робить програма при запуску з ключем `-f` у першому додатку, і яка відмінність цього режиму від запуску з ключем `-w`?
3. Як перший додаток визначає, де знаходиться виконуваний файл, якщо йому передано ім’я файлу для запуску?
4. Що таке змінна оточення `PATH`, як вона використовується в цьому завданні, і чому вона важлива для пошуку виконуваного файлу?
5. Яким чином перший додаток має відображати код повернення після завершення виконання програми, якщо було задано ключ `-w`?
6. Що таке асинхронний запуск процесу і як його можна реалізувати в Linux, щоб перший додаток міг негайно продовжувати виконання після запуску програми?
7. Яку інформацію має вивести на екран другий додаток щодо командного рядка, і яким чином вона передається процесам у Linux?
8. Як у другому додатку отримати ідентифікатори процесу (`pid`), сесії (`sid`), групи процесів (`pgid`), і процесу-предка (`ppid`) в Linux?
9. Які файли системи (`/etc/passwd` і `/etc/group`) потрібні для отримання інформації про ім’я користувача та його групу, і як вони використовуються у другому додатку?
10. Що означає поняття «процес-предок», і як його ідентифікатор (`ppid`) допомагає в розумінні структури взаємозв’язків між процесами в Linux?

4.2 Контрольні питання до індивідуальної роботи №2

1. Що таке процес в операційній системі Linux і які основні стани процесу існують?
2. Поясніть різницю між системними викликами fork() та exec() в Linux.
3. Що таке змінна оточення PATH в Linux і яке її призначення?
4. Як працює механізм міжпроцесної комунікації (IPC) в Linux? Наведіть приклади методів IPC.

5. Що таке сигнали в Linux і як вони використовуються для управління процесами?

6. Поясніть концепцію файлових дескрипторів в Linux та їх роль у роботі з файлами та пристроями.

7. Що таке демон-процеси в Linux і як вони відрізняються від звичайних процесів?

8. Як працює планувальник процесів в Linux? Які основні алгоритми планування використовуються?

9. Що таке системні виклики в Linux і як вони відрізняються від звичайних функцій?

10. Поясніть концепцію віртуальної пам'яті в Linux та як вона реалізується за допомогою сторінкової організації пам'яті.

5. КРИТЕРІЇ ОЦІНЮВАННЯ

Виконання та захист індивідуального завдання:

Підготовлений звіт у вигляді файлу *pdf розміщується у відповідному розділі дисципліни в Moodle і перевіряється протягом тижня після завершення терміну подачі. Оскарження оцінки може бути здійснене на останньому практичному занятті модуля.

Max 15 балів:

– студент підготував звіт за індивідуальним завданням, в якому: правильно визначив мету, описав програму, обґрунтував використання різних даних, виконав необхідні тести програми, представив висновок та додаток з кодом програми, викладено діловим, науковим або публіцистичним стилем української (10 балів);

– студент під час захисту індивідуального завдання демонструє володіння термінологічним апаратом, відповідає на запитання, здатний продемонструвати робочу програму (5 балів).

6. СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

Базові

1 Голубничий Д. Ю., Холодкова А. В., Шматко О. В., Козуля М. М. Операційні системи : лабораторний практикум. Харків : НТУ “ХПІ”, 2019. 336 с. URL: http://library.kpi.kharkov.ua/files/new_postupleniya/opsilp.pdf.

2 Голубничий Д. Ю., Холодкова А. В. Операційні системи. Харків : ХНЕУ ім. С. Кузнеця, 2018. 317 с. URL: http://repository.hneu.edu.ua/handle/123456789/23844.

3 Операційні системи : навчальний посібник. / І. М. Федотова-Півень та ін. ; за ред. В.М. Рудницького. Харків : ТОВ «ДІСА ПЛЮС», 2019. 216 с.

4 Зайцев В. Г., Дробязко І. П. Операційні системи : навч. посіб. для студ. Київ : КПІ ім. Ігоря Сікорського, 2019. 240 с.

5 Рукін М., Григор'єв М., Балалаєва Т. Операційні системи. Вінниця : Ліброком, 2016. 350 с.

6 Волох С. В. UЬuntu Linux з нуля. Київ : Видавнича група BHV, 2018. 400 с.

7 Таненбаум Э., Бос Х. Сучасні операційні системи. Київ : Пітер в Україні, 2018. 1120 с.

8 Граннеман С. Linux. Кишеньковий довідник. Київ : Діалектика, 2019. 464 с.

9 Погребняк Б. І., Булаєнко М. В. Операційні системи : навч. посібник. Харків : ХНУМГ ім. О. М. Бекетова, 2018. 104 с.

10 Костогриз В. Метод використання подвійного завантаження та мультизавантаження операційних систем сімейства Microsoft Windows із зовнішнього системного диску. *Електроніка та інформаційні технології*. 2018. Випуск 10. С. 109–120.

11 Операційна система Kolibri : веб-сайт. URL: http://kolibrios.org/ (дата звернення: 09.11.2024).

12 Windows Sysinternals : веб-сайт. URL: http://technet.microsoft.com/ru-ru/sysinternals (дата звернення: 09.11.2024).

13 Windows : веб-сайт. URL: http://windows.microsoft.com/ru-ru/windows/home (дата звернення: 09.11.2024).

ДОДАТОК А. ПРИКЛАД ОФОРМЛЕННЯ ЗВІТУ

ТОВ «ТЕХНІЧНИЙ УНІВЕРСИТЕТ
«МЕТІНВЕСТ ПОЛІТЕХНІКА»
Факультет автоматизації виробництва
та цифрових технологій
Кафедра цифрових технологій
та проєктно-аналітичних рішень

ЗВІТ З ІНДИВІДУАЛЬНОГО ЗАВДАННЯ

з дисципліни: «ОПЕРАЦІЙНІ СИСТЕМИ ТА ОСНОВИ СИСТЕМНОГО ПРОГРАМУВАННЯ»

на тему
«Передача параметрів командного рядка до програми. Робота із процесами. Отримання системної інформації»

Роботу виконав Іван ФЕЩЕНКО

Роботу перевірив Олександр ШМАТКО

ЗАПОРІЖЖЯ 2024

metinvest polytechnic

РЕФЕРАТ

Пояснювальна записка: 24 сторінок, 5 рисунків, 2 таблиць, 3 додатки, 13 використаних джерел.

Мета проектування – розробка системних програмних застосунків для демонстрації роботи механізмів міжпроцесорної взаємодіє та механізмів управління процесами в ОС Лінукс.

Робота складається з двох розділів.

Перший розділ присвячений детальному опису процесу створення системних програмних застосунків з використанням хмарного середовища розробки Repl.it. Наведено детальний опис процесу створення додатків в середовищі Repl, їх компіляції та тестування. Розроблено програмний застосунок, що демонструє основні механізми створення та управління процесами в ОС Лінукс.

У другому розділі, згідно з індивідуальним завданням, розроблено та протестовано програмний застосунок, що демонструє механізми міжпроцсерної взаємодії

Ключові слова: СИСТЕМНЕ ПРОГРАМУВАННЯ, ОПЕАРЦІЙНА СИСТЕМА ЛІНУКС, МІЖПРОЦЕСОРНА ВЗАЄМОДІЯ, МЕХАНІЗМИ УПРАВЛІННЯ ПРОЦЕСАМИ.

ABSTRACT

Explanatory note: 24 pages, 5 figures, 2 tables, 3 appendices, 13 references.

The purpose of the project is to develop system software applications to demonstrate the operation of interprocessor interaction mechanisms and process control mechanisms in Linux.

The work consists of two sections.

The first section is devoted to a detailed description of the process of creating system software applications using the Repl.it cloud development environment. It provides a detailed description of the process of creating applications in the Repl environment, compiling and testing them. A software application has been developed that demonstrates the basic mechanisms for creating and managing processes in the Linux OS.

In the second section, according to the individual task, a software application demonstrating the mechanisms of interprocess interaction is developed and tested

Keywords: SYSTEM PROGRAMMING, LINUX OPERATING SYSTEM, INTERPROCESSOR INTERACTION, PROCESS CONTROL MECHANISMS.

ЗМІСТ

ВСТУП

Системне програмування – це різновид програмування, що полягає у розробці програм, які взаємодіють з системним програмним забезпеченням (ПЗ), операційною системою (ОС) або апаратним забезпеченням комп'ютера. Головною відмінністю системного програмування в порівнянні з прикладним програмуванням є те, що прикладне програмне забезпечення призначене для кінцевих користувачів (наприклад, текстові та графічні редактори), тоді як результатом системного програмування є програми, які обслуговують апаратне забезпечення або операційну систему (наприклад, дефрагментація диска), що обумовлює значну залежність такого типу ПЗ від апаратної частини.

Слід зазначити, що «звичайні» прикладні програми можуть використовувати у своїй роботі фрагменти коду, характерні для системних програм, і навпаки; тому чіткої межі між прикладним та системним програмуванням немає. Оскільки різні операційні системи відрізняються як внутрішньою архітектурою, так і способами взаємодії з апаратним та програмним забезпеченням, то принципи системного програмування для різних ОС є відмінними.

Тому розробка прикладних програм, які здійснюватимуть одні і ті ж дії на різних ОС, може суттєво відрізнятися..

1 ДОСЛІДЖЕННЯ МЕТОДІВ ПЕРЕДАЧІ ПАРАМЕТРІВ КОМАНДНОГО РЯДКА ДО ПРОГРАМИ

1.1. Основне завдання

1. Розробити дві програми для Linux, перша з яких:

а) при завданні ключа -w ім'я-виконуваного-файлу - запускає заданий додаток (з пошуком його в системному шляху, заданому змінною оточення PATH) - чекає завершення програми та виводить повідомлення про завершення на екран разом із кодом повернення

б) при завданні ключа -f ім'я-виконуваного файлу - запускає заданий додаток (з пошуком його в системному шляху, заданому змінною оточення PATH) - негайно продовжує виконання (асинхронний запуск)

Другий додаток має виводити на екран:

- значення аргументів командного рядка;
- ідентифікатори: процесу (pid), сесії (sid), групи процесів (pgid)
- та процесу-предка (ppid)
- ім'я користувача, під яким процес запущено та його групу
- (отримані з файлів /etc/passwd та /etc/group)

1.2. Створення програмного додатку згідно основного завдання

Для створення додатку використаємо онлайн середовище розробки Repl.it (https://replit.com/). Після завантаження онлайн середовища створимо новий проєкт repl, обравши мову програмування C++ (рис.1.1).

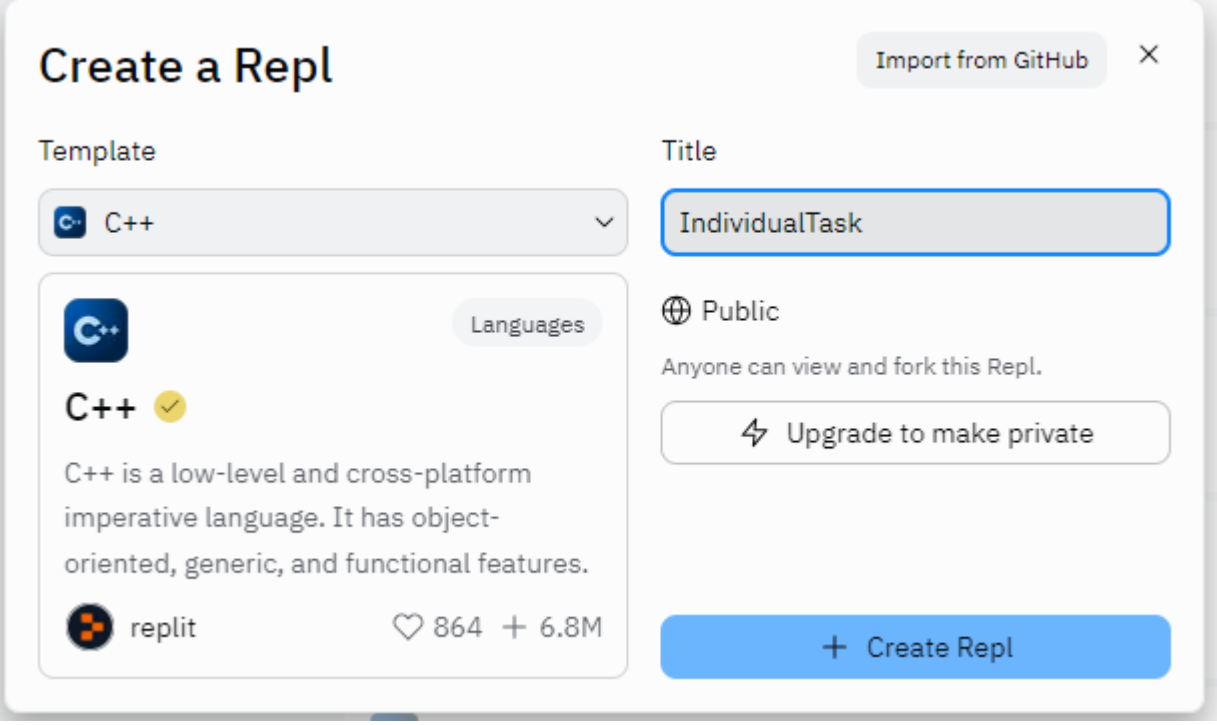


Рисунок 1.1 – Створення нового проєкту Repl

Після створення проєкту потрібно створити програмний код для першої головної програми. Для цього перейдемо на вкладку програми main.cpp та введемо програмний код, як показано на рисунку 1.2. Повний програмний код наведено в Додатку А.

main.cpp | 2.cpp | Makefile | 3.cpp

main.cpp > ...

```cpp
// Program 1
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " [-w|-f] executable-file" << std::endl;
        return 1;
    }

    std::string option = argv[1];
    std::string executable = argv[2];

    if (option == "-w") {
        pid_t child_pid = fork();

        if (child_pid == 0) {
            // Child process
            execlp(executable.c_str(), executable.c_str(), nullptr);
            std::cerr << "Failed to execute the program." << std::endl;
            return 1;
        } else if (child_pid > 0) {
            // Parent process
            int status;
            waitpid(child_pid, &status, 0);
            std::cout << "Program has finished with return code: " << WEXITSTATUS(status) <<
std::endl;
        } else {
            std::cerr << "Error creating child process." << std::endl;
            return 1;
```

Рисунок 1.2 – Створення програмного коду головної програми

Для реалізації другої програми з основного завдання створимо новий файл 2.cpp та введемо програмний код, який наведено в Додатку Б. Результати створення програми для виведення інформації про:

– значення аргументів командного рядка;
– ідентифікатори: процесу (pid), сесії (sid), групи процесів (pgid)
– та процесу-предка (ppid)
– ім'я користувача, під яким процес запущено та його групу
– (отримані з файлів /etc/passwd та /etc/group)

на екран представлено на рисунку 1.3.

2.cpp

```cpp
// Program 2
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

int main() {
    pid_t pid = getpid();
    pid_t sid = getsid(0);
    pid_t pgid = getpgid(0);
    pid_t ppid = getppid();

    std::cout << "PID: " << pid << std::endl;
    std::cout << "SID: " << sid << std::endl;
    std::cout << "PGID: " << pgid << std::endl;
    std::cout << "PPID: " << ppid << std::endl;

    struct passwd *pw = getpwuid(getuid());
    struct group *gr = getgrgid(getgid());

    if (pw != nullptr) {
        std::cout << "User name: " << pw->pw_name << std::endl;
        std::cout << "Group name: " << gr->gr_name << std::endl;
    }

    return 0;
}

```

Рисунок 1.3 – Створення програмного коду другої програми

Для компіляції та запуску програмних застосунків необхідно розробити Makefile. На рисунку 1.4 представлено зміст такого файлу для розроблених програмних файлів.

```
all: program1 program2

program1: main.cpp
  g++ -o program1 main.cpp

program2: 2.cpp
  g++ -o program2 2.cpp
```

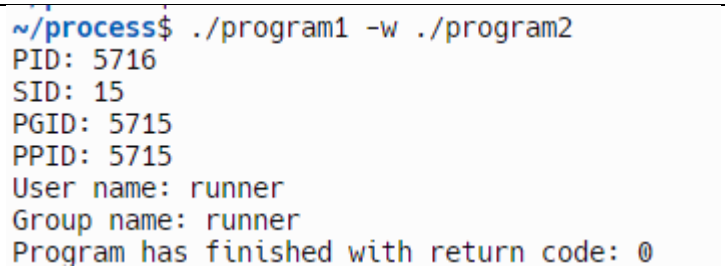
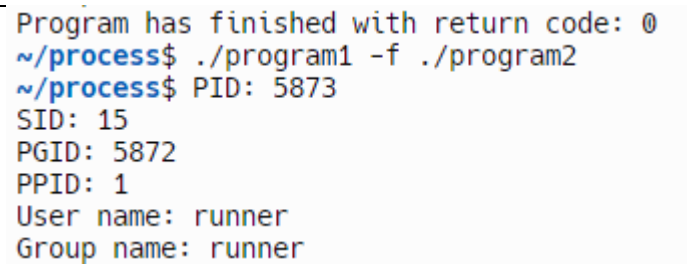
Рисунок 1.4 – Створення програмного коду другої програми

1.3 Тестування розробленого програмного забезпечення

Для тестування розроблених програмних застосунків створимо та опишемо тестові сценарії. В таблиці 1.1 представлено опис тестових сценаріїв для функціонального тестування розроблених програмних застосунків.

Таблиця 1.1 – Тестові сценарії

Назва тестового сценарію	Сценарій 1
Опис тестового сценарію	Запуск основної програми
Передумова	Програма скомпільована та готова до запуску
Кроки для виконання	Ввести в командному рядку команду ./program1
Результат, що очікується	Виведеться повідомлення: Usage: ./program1 [-w|-f] executable-file
Отриманий результат	~/process$ ./program1 Usage: ./program1 [-w|-f] executable-file
Висновок	Тест пройдено
Назва тестового сценарію	Сценарій 2
Опис тестового сценарію	Запуск основної програми з ключем -w
Передумова	Програма скомпільована та готова до запуску
Кроки для виконання	Ввести в командному рядку команду ./program1 -w ./program2
Результат, що очікується	Виведеться повідомлення: PID: 5716 SID: 15 PGID: 5715 PPID: 5715 User name: runner

	Group name: runner Program has finished with return code: 0
Отриманий результат	
Висновок	Тест пройдено
Назва тестового сценарію	Сценарій 3
Опис тестового сценарію	Запуск основної програми з ключем -f
Передумова	Програма скомпільована та готова до запуску
Кроки для виконання	Ввести в командному рядку команду ./program1 -f ./program2
Результат, що очікується	Виведеться повідомлення: SID: 15 PGID: 5872 PPID: 1 User name: runner Group name: runner
Отриманий результат	
Висновок	Тест пройдено

2. РОЗРОБКА СИСТЕМНОГО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ ЗГІДНО ІЗ ІНДИВІДУАЛЬНИМ ЗАВДАННЯМ

2.1 Індивідуальне завдання

Варіант №2

Розробити дві програми. Перша обчислює число Фібоначчі за номером, введеним користувачеві, та формулою Fi = Fi-1 + Fi-2, F0 = F1 = 1 і виводить його на екран. Друга програма запускає першу як новостворений процес.

2.2 Розробка програмного забезпечення згідно із індивідуальним завданням

Для розробки програмного забезпечення згідно із індивідуальним завданням використаємо середовище розробки Repl.it. На рисунку 2.1 представлено програмний код для індивідуального завдання:

3.cpp

```cpp
#include <iostream>

int calculateFibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    int a = 1, b = 1, temp;
    for (int i = 2; i <= n; ++i) {
        temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}

int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    int result = calculateFibonacci(num);
    std::cout << "The Fibonacci number is: " << result << std::endl;
    return 0;
}

```

Рисунок 2.1 – Програмний код

2.3 Тестування розробленого програмного забезпечення

Для тестування розроблених програмних застосунків створимо та опишемо тестові сценарії. В таблиці 2.1 представлено опис тестових сценаріїв для функціонального тестування розроблених програмних застосунків.

Таблиця 2.1 – Тестові сценарії

Назва тестового сценарію	
Опис тестового сценарію	
Передумова	
Кроки для виконання	
Результат, що очікується	
Отриманий результат	
Висновок	

ВИСНОВКИ

У цій роботі були розроблені системні програмні застосунки для демонстрації роботи механізмів міжпроцесорної взаємодії та механізмів управління процесами в операційній системі Linux. Проект включав дві ключові частини: розробку програми для передачі параметрів командного рядка та програми для взаємодії між процесами.

В роботі розглянуто процес створення програм, які ефективно виконують задані завдання, використовуючи хмарне середовище розробки Repl.it.

Проведене тестування підтвердило, що розроблені програми коректно виконують свої функції та відповідають поставленим вимогам.

Застосунки, що були розроблені, мають практичне значення у сфері системного програмування і можуть бути використані як основа для подальших розробок.

Перспективи подальших досліджень: Цей проект відкриває шлях для подальших досліджень у сфері системного програмування. Можливі напрямки включають розширення функціональності існуючих програм, інтеграцію з іншими системами та платформами, а також оптимізацію продуктивності та безпеки розроблених застосунків.

ПЕРЕЛІК ПОСИЛАНЬ

1 Голубничий Д. Ю., Холодкова А. В., Шматко О. В., Козуля М. М. Операційні системи : лабораторний практикум. Харків : НТУ “ХПІ”, 2019. 336 с. URL: http://library.kpi.kharkov.ua/files/new_postupleniya/opsilp.pdf.

2 Голубничий Д. Ю., Холодкова А. В. Операційні системи. Харків : ХНЕУ ім. С. Кузнеця, 2018. 317 с. URL: http://repository.hneu.edu.ua/handle/123456789/23844.

3 Операційні системи : навчальний посібник. / І. М. Федотова-Півень та ін. ; за ред. В.М. Рудницького. Харків : ТОВ «ДІСА ПЛЮС», 2019. 216 с.

4 Зайцев В. Г., Дробязко І. П. Операційні системи : навч. посіб. для студ. Київ : КПІ ім. Ігоря Сікорського, 2019. 240 с.

5 Рукін М., Григор'єв М., Балалаєва Т. Операційні системи. Вінниця : Ліброком, 2016. 350 с.

6 Волох С. В. UЬuntu Linux з нуля. Київ : Видавнича група BHV, 2018. 400 с.

7 Таненбаум Э., Бос Х. Сучасні операційні системи. Київ : Пітер в Україні, 2018. 1120 с.

8 Граннеман С. Linux. Кишеньковий довідник. Київ : Діалектика, 2019. 464 с.

9 Погребняк Б. І., Булаєнко М. В. Операційні системи : навч. посібник. Харків : ХНУМГ ім. О. М. Бекетова, 2018. 104 с.

10 Костогриз В. Метод використання подвійного завантаження та мультизавантаження операційних систем сімейства Microsoft Windows із зовнішнього системного диску. *Електроніка та інформаційні технології*. 2018. Випуск 10. С. 109–120.

11 Операційна система Kolibri : веб-сайт. URL: http://kolibrios.org/ (дата звернення: 09.11.2024).

12 Windows Sysinternals : веб-сайт. URL: http://technet.microsoft.com/ru-ru/sysinternals (дата звернення: 09.11.2024).

13 Windows : веб-сайт. URL: http://windows.microsoft.com/ru-ru/windows/home (дата звернення: 09.11.2024).

ДОДАТОК А. Програмний код для головної програми

```
// Program 1
#include <iostream>
#include <cstdlib>
#include <cstring>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char* argv[]) {
    if (argc != 3) {
        std::cerr << "Usage: " << argv[0] << " [-
w|-f] executable-file" << std::endl;
        return 1;
    }
    std::string option = argv[1];
    std::string executable = argv[2];
    if (option == "-w") {
        pid_t child_pid = fork();
        if (child_pid == 0) {
            // Child process
            execlp(executable.c_str(),
executable.c_str(), nullptr);
            std::cerr << "Failed to execute the
program." << std::endl;
            return 1;
        } else if (child_pid > 0) {
            // Parent process
            int status;
```

```
                waitpid(child_pid, &status, 0);
                std::cout << "Program has finished with
return code: " << WEXITSTATUS(status) << std::endl;
            } else {
                std::cerr << "Error creating child
process." << std::endl;
                return 1;
            }
        } else if (option == "-f") {
            pid_t child_pid = fork();

            if (child_pid == 0) {
                // Child process
                execlp(executable.c_str(),
executable.c_str(), nullptr);
                std::cerr << "Failed to execute the
program." << std::endl;
                return 1;
            } else if (child_pid < 0) {
                std::cerr << "Error creating child
process." << std::endl;
                return 1;
            }
            // Continue parent process execution
without waiting for completion
        } else {
            std::cerr << "Invalid argument. Use -w or -
f" << std::endl;
            return 1;
        } return 0;}
```

ДОДАТОК Б. Програмний код для другої програми

```
// Program 2
#include <iostream>
#include <unistd.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

int main() {
    pid_t pid = getpid();
    pid_t sid = getsid(0);
    pid_t pgid = getpgid(0);
    pid_t ppid = getppid();
    std::cout << "PID: " << pid << std::endl;
    std::cout << "SID: " << sid << std::endl;
    std::cout << "PGID: " << pgid << std::endl;
    std::cout << "PPID: " << ppid << std::endl;
    struct passwd *pw = getpwuid(getuid());
    struct group *gr = getgrgid(getgid());
    if (pw != nullptr) {
        std::cout << "User name: " << pw->pw_name
<< std::endl;
        std::cout << "Group name: " << gr->gr_name
<< std::endl;
    }

    return 0;
}
```

ДОДАТОК В. Програмний код для програми із індивідуального завдання

```
#include <iostream>

int calculateFibonacci(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    int a = 1, b = 1, temp;
    for (int i = 2; i <= n; ++i) {
        temp = a + b;
        a = b;
        b = temp;
    }
    return b;
}

int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    int result = calculateFibonacci(num);
    std::cout << "The Fibonacci number is: " <<
result << std::endl;
    return 0;
}.
```