ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ОБРАЗОВАНИЯ
«СИБИРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ТЕЛЕКОММУНИКАЦИЙ И ИНФОРМАТИКИ»

КАФЕДРА ВЫЧИСЛИТЕЛЬНЫХ СИСТЕМ

ЛАБОРАТОРНАЯ РАБОТА № 4

«ПАРАЛЛЕЛЬНОЕ ВЫПОЛНЕНИЕ ОПЕРАЦИЙ НА УРОВНЕ ОПЕРАЦИОННОЙ СИСТЕМЫ. ПРОЦЕССЫ»

Автор: С.Н. Мамойленко

Новосибирск - 2016

Оглавление

Цель работы

Лабораторная работа направлена на развитие следующих общекультурных, общепрофессиональных и профессиональных компетенций:

- способностью к профессиональной эксплуатации современного оборудования и приборов (ОК-8);
- владением методами и средствами получения, хранения, переработки и трансляции информации посредством современных компьютерных технологий, в том числе в глобальных компьютерных сетях (ОПК-5);
- способностью проектировать системы с параллельной обработкой данных и высокопроизводительные системы, и их компоненты (ПК-9);
- способностью к программной реализации систем с параллельной обработкой данных и высокопроизводительных систем (ПК-14);
- способностью проектировать распределенные информационные системы, их компоненты и протоколы их взаимодействия (ПК-8);
- способностью к программной реализации распределенных информационных систем (ПК-13);

В результате изучения дисциплины должно быть сформировано понимание принципов организации распределённых приложений, использования для этого процессов, стандарта MPI, а также получены умения и навыки по проектированию и разработке распределённого программного обеспечения с использованием стандарта MPI.

Теоретическое введение

1. Базовые сведения о процессах

Совокупность программы и описывающей её информации называется процессом. Информация о программе хранится в операционной системе в виде специальной структуры, называемой дескриптором процесса. Дескриптор процесса и дополнительная информация о программе вместе называются контекстом процесса.

2. Типы процессов

Все запущенные процессы условно (в зависимости от выполняемой ими функции) можно разделить на три типа: системные, процессы-демоны и прикладные процессы.

Системные процессы являются частью ядра и всегда расположены в оперативной памяти. Они часто не имеют соответствующих им программ в виде исполняемых файлов и всегда запускаются особым образом при загрузке ядра системы. Системными процессами являются, например, `keventd` (диспетчер системных событий), `kswapd` (диспетчер страничного замещения - своппинга), `bdflush` (диспетчер буферного кэша) и т.д.

К системным процессам следует отнести процесс `init`, являющийся первым создаваемым процессом в системе и «прародителем» всех остальных процессов.

Процессы-демоны - это не интерактивные (т.е. не взаимодействующие с пользователями) процессы, которые выполняются в фоновом режиме. Обычно обеспечивают работу различных подсистем GNU\Linux, например, системы терминального доступа, системы печати, системы сетевого доступа и сетевых услуг и т.п.

К прикладным относятся все остальные процессы, выполняющиеся в системе. Как правило, это процессы, порожденные в рамках пользовательского сеанса работы.

Пользовательские процессы могут выполняться как в интерактивном, так и в фоновом режиме, но в любом случае время их жизни (и выполнения) ограничено сеансом работы пользователя. При выходе из системы все пользовательские процессы будут уничтожены. Интерактивные процессы связаны с определённым терминалом и через него взаимодействуют с пользователем. Фоновые

процессы выполняются независимо от пользователя и (псевдо)параллельно. Из фонового процесса можно создать процесс-демон, для чего ему следует отключиться от терминала.

Запустить процесс на выполнение в фоновом режиме можно одним из следующих способов:

- указать в конце командной строки символ амперсанд (`&`),
- приостановить выполнение интерактивного процесса (нажать комбинацию клавиш <CTRL+Z>), а затем командой `bg` запустить его на фоновое выполнение.

Чтобы посмотреть список всех фоновых задач, выполняющихся в рамках текущего сеанса пользователя, используется команда `jobs`, которая также выведет и текущее состояние каждой задачи.

Чтобы перевести задачу, выполняющуюся в фоновом режиме, на передний план (в интерактивных режим), надо вызвать команду `fg` и передать ей в качестве параметра номер фоновой задачи (из списка, выдаваемого командой `jobs`).

Любая фоновая задача будет приостановлена, если ей требуется определённый доступ к консоли (например, для ввода информации).

3. Состояние процесса

Каждый процесс в операционной системе GNU\Linux может находиться в одном из четырёх состояний: работоспособный, спящий (или ожидающий), остановленный и завершившийся.

Работоспособный (runnable) процесс. Если процесс в текущий момент выполняет какие-либо действия или стоит в очереди на получение кванта времени на центральном процессоре, он называется работоспособным и обозначается символом R.

Ожидающий (спящий, sleeping) процесс. Это состояние обозначается символом S и возникает после того, как процесс инициирует системную операцию, окончания которой он должен дождаться. К таким операциям относятся ввод/вывод, истечение заданного интервала времени, завершение дочернего процесса и т.д.

Остановленный (stopped) процесс. Остановленный процесс не использует процессор и может быть полностью выгружен из оперативной памяти. Пользователь может остановить процесс, например, чтобы дать возможность другому процессу использовать больший объем оперативной памяти или быстрее завершиться. Обычный пользователь может остановить и продолжить только свой процесс, а суперпользователь – любой процесс в системе. Операционная система автоматически останавливает фоновые процессы в случае, когда они пытаются ввести данные с терминала. Это состояние обозначается символом T.

Завершившийся (зомби, ''zombie'') процесс. После завершения процесса информация о нем должна быть удалена операционной системой из таблицы процессов. В GNU\Linux такая операция возможна только после того, как родительский процесс выполнит системную операцию «ожидания завершения дочер=него процесса» и прочитает статус возврата. До этих пор ОС вынуждена хранить в таблице процессов запись о завершившемся процессе, хотя он реально уже не существует и не потребляет ресурсы ЭВМ. Такое состояние процесса обозначается символом Z.

4. Атрибуты процесса

Любой процесс в операционной системе описывается дескриптором, содержащим ряд параметров, включая:

- уникальный идентификатор процесса (PID);
- идентификатор родительского процесса (PPID);
- связанный с процессом терминал (TTY);
- идентификаторы пользователя (UID, RUID, EUID, FSUID, SUID) и группы (GID, RGID, EGID, FSGID, SGID) от имени которых процесс выполняется и пытается осуществлять действия в рамках операционной системы или файловой системы;
- номер группы процессов (GROUP);
- идентификатор пользовательского сеанса (SESSION);

- и т.д.

Уникальный идентификатор процесса (PID) – это целое число, позволяющее ядру системы различать процессы, т.е. своего рода «имя» процесса. По сути, PID – это номер ячейки в таблице дескрипторов процессов, выполняемых под управлением операционной системы. Когда создается новый процесс, ядро операционной системы использует следующую свободную запись в таблице дескрипторов. Если достигнут конец таблицы, то производится поиск свободной ячейки начиная с начала таблицы. Когда процесс завершает свою работу, ядро освобождает занятый им идентификатор и соответствующий ему дескриптор.

Каждый процесс «запоминает» своего родителя в поле PPID. Зная значения поля PID и PPID можно построить иерархию порождения процессов, начиная с самого первого процесса в системе (он обычно имеет PID = 0). Такая иерархия называется деревом процессов. Значение поля PPID играет особую роль при завершении процесса (см. ниже).

Как известно, каждому интерактивному процессу при создании назначаются потоки для: ввода информации, для её вывода и для вывода ошибок (stdin, stdout, stderr) . Обычно все эти потоки указывают на один терминал, который называется «связанным» с процессом и имя этого терминала (имя файла устройства) помещается в поле «TTY».

Все процессы в операционной системе запускаются каким-либо пользователем. Идентификатор пользователя, запустившего процесс, помещается в поле «реальный идентификатор пользователя» (UID или RUID, Real UID). Часто возникает необходимость, чтобы процесс был запущен одним пользователем, а имел возможность получать доступ к ресурсам, принадлежащим другому пользователю. Контроль доступа к ресурсам системы осуществляется исходя из значения поля «эффективный идентификатор пользователя» (EUID, Effective UID). Примером ситуации выполнения действий от имени другого пользователя может служить смена пользовательского пароля с помощью команды passwd. Очевидно, что если база паролей (файлы `/etc/passwd` или `/etc/shadow`) будет доступна на запись кому угодно, тогда нельзя будет говорить ни о какой безопасности системы. Поэтому passwd настроена таким образом, что независимо от того, какой пользователь её выполняет, действия будут производиться всегда от имени суперпользователя (root, UID=0). При этом ответственность за правильную и допустимую смену пароля несёт passwd. «Настроена» - означает, что владельцем файла является пользователь root и файл имеет специальное право «Установить эффективного пользователя». При запуске такого файла в поле EUID операционная система поместит идентификатор владельца файла и все действия будут осуществляться уже от его имени. Очевидно, что значения полей RUID и EUID могут и совпадать (когда пользователь, запустивший процесс, выполняет все действия от своего имени). В ходе выполнения пользовательский процесс имеет возможность изменять значения RUID и EUID (менять их местами или делать одинаковыми). При изменении значения поля EUID его изначальное значение сохраняется в поле «сохраненный идентификатор пользователя» SUID (Saved UID). Кроме эффективного идентификатора в дескрипторе процессов в ОС Linux имеется нестандартное поле FSUID (File System UID), которое используется при обращении процесса к файловой системе. При запуске процесса FSUID эквивалентен EUID. Все сказанное про идентификаторы пользователей справедливо и для основной группы пользователя (поля GID или RGID, EGID, SGID, FSGID).

В операционной системе Linux процессы, выполняющие одну задачу, объединяются в группы. Например, если в командной строке задано, что процессы связаны при помощи программного канала, они обычно помещаются в одну группу процессов. Каждая группа процессов обозначается собственным идентификатором. Процесс внутри группы, идентификатор которого совпадает с идентификатором группы процессов, считается лидером группы процессов.

В свою очередь, каждая группа процессов принадлежит к сеансу пользователя. Сеанс обычно представляет собой набор из одной группы процессов переднего плана, использующей терминал, и одной или более групп фоновых процессов. Каждый сеанс также обозначается идентификатором, который совпадает с PID процесса лидера. При завершении пользовательского сеанса все принадлежащие к нему процессы автоматически завершаются системой. Именно поэтому процессы-демоны обязательно должны иметь идентификатор сеанса, отличный от сеанса пользователя, который их запустил.

5. Получение информации о состоянии и атрибутах процессов

5.1. Из командной строки

Чтобы получить информацию о том, какие процессы запущены в данный момент в системе и в каком состоянии они находятся можно использовать команду ps (process status) с определённым набором опций (таблица 1).

Таблица 1. Опции команды `ps`.

Опция	Действие
-a	Выдать все процессы системы, включая лидеров сеансов
-d	Выдать все процессы системы, исключая лидеров сеансов
-e	Выдать все порцессы системы
-x	Выдать процессы системы не имеющие контрольного терминала
-o	Определяет формат вывода. Формат задается как параметр опции в виде символьной строки, поля которой (см. таблицу 2) разделяются символом ',' (запятая)
-u	Выдать процессы, принадлежащие указанному пользователю. Пользователь задается в парамете опции в символьном виде.

Таблица 2. Поля форматного вывода команды ps

Формат	Значение
F	статус процесса
S	состояние процесса
UID	идентификатор пользователя
PID	идентификатор процесса
PPID	идентификатор родительского процесса
PRI	текущий динамический приоритет процесса
NI	значение приоритета процесса
TTY	управляющий терминал процесса
TIME	суммарное время выполнения процесса процессором
STIME	время создания процесса
COMMAND	имя команды, соответсвующей процессу
SESS	идентификатор сеанса процесса
PGRP	идентификатор группы процесса

Например, список запущенных процессов вашей системы можно получить, используя команду `ps -e`. Информацию о группах и сессиях процессом можно получить так: `ps -axo pid,pgrp,session,command.`

Можно также посмотреть "дерево" процессов используя команду `pstree`.

Чтобы посмотреть использование ресурсов «в динамике» можно использовать команду `top`. Выход из просмотра осуществляется нажатием <q>.

5.2. При выполнении процессов

Для получения значений идентификаторов процесс может использовать системные вызовы, описание которые представлено на рисунке 1.

Назначение функция ясно из того, что в их названии следует после `get`. Например, функция `getpid` – вернёт идентификатор процесса (PID). Все функции, кроме `getsid` не принимают ни каких параметров и возвращают значение соответствующих полей из дескриптора текущего процесса (вызывающего функцию). Функция `getsid` позволяет узнать идентификатор сеанса для любого процесса в системе (если это позволено соответствующему пользователю), идентификатор которого

передаётся в качестве параметра. Если передать вызову `getsid` значение 0, то он вернет идентификатор сеанса вызывающего процесса (т.е. это эквивалентно записи `getsid(getpid());`).

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

pid_t getpid  (void);
pid_t getppid (void);
uid_t getuid  (void);
uid_t geteuid (void);
gid_t getgid  (void);
gid_t getegid (void);
pid_t getpgrp (void);
pid_t getsid  (pid_t pid);
```

Рисунок 1 – Заголовки функций, возвращающих значения идентификаторов

Процесс может создать новую группу или присоединиться к существующей, а также определить новый сеанс при помощи системных вызовов, заголовки который представлены на рисунке 2.

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

int setpgid ();
int setsid  ();
```

Рисунок 2 – Функции по управлению членством в группах

Вызов setpgid устанавливает идентификатор группы процесса с идентификатором равным `pid` текущего процесса. В случае ошибки возвращается `-1`.

Изменить значение полей RUID, EUID, FSUID, RGID, EGID, FSGID можно при помощи системных вызовов, заголовки которых представлены на рисунке 3.

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

int setuid   (uid_t uid);
int seteuid  (uid_t euid);
int setreuid (uid_t ruid, uid_t euid);
int setfsuid (uid_t fsuid);
int setgid   (gid_t gid);
int setegid  (gid_t egid);
int setfsgid (gid_t fsgid);
int setregid (gid_t rgid, gid_t egid);
```

Рисунок 3 – Функции по управлению значениями идентификаторов процесса

Назначение функций очевидно из их названия. Пояснений требуют лишь функции `setreuid` и `setregid`. Обе эти функции предназначены для изменения полей соответственно RUID, EUID и RGID, EGID в ситуации, когда операционная система не поддерживает дополнительных полей SUID, SGID. Эти функции включены в библиотеку для реализации совместимости с такими операционными системами как 4.3 UNIX BSD, которые не имеют в дескрипторах процессов указанных полей. За подробной информацией об использовании этих функции следует обратиться к электронной справочной системе `man`.

Очевидно, что процесс может получить информацию только о своих атрибутах и при этом процесс всегда находится в состоянии Running (R).

Использование функции установки значений полей дескриптора процесса рассмотрим на примере создания процесса-демона (рисунок 4).

```
1  #include <unistd.h>
2
3  int main(void){
4    close (0);
5    close (1);
6    close (2);
7    setsid(0);
8    setpgrp(0,0);
9
10   /* тело процесса-демона */
11   sleep(10);
12   return (0);
13 }
```

Рисунок 4– Создание процесса-демона

В первой строке подключается необходимый заголовочный файл. В данном случае нам нужны только функции по работе с аттрибутами процесса, файловыми дескрипторами и функции sleep, которая добровольно переводит процесс в состояние ожидания. Далее в строках 4-6 процесс отключается от терминала (вспомним, что демон не имеет «присоединенного» терминала). После чего создаёт новый сеанс и группу (строки 7-8). После этого процесс может выполняться независимо от того, находится ли запустивший его пользователь в системе или уже завершил свой сеанс. В нашем случае демон всего лишь 10 секунд «спит» (строка 11) и затем завершается (строка 12).

6. Порождение процессов

Новый процесс порождается с помощью системного вызова fork (см. рисунок 6).

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

pid_t fork(void);
```

Рисунок 5 – Функции порождения нового процесса

Порожденный, или дочерний процесс, является точной копией родительского процесса, за исключением следующих атрибутов:

- идентификатор процесса (PID);
- идентификатор родительского процесса (PPID). У дочернего процесса он приравнивается PID родительского процесса (того, кто выполнил вызов fork);
- дочерний процесс не имеет очереди сигналов, ожидающих обработки.

После завершения вызова fork, оба процесса начинают выполнять одну и ту же инструкцию – следующую за вызовом fork.

Чтобы процесс мог определить кем он является (дочерним или родительским) используется возвращаемое значение вызова fork. В первом случае, когда процесс является дочерним, fork вернет 0, во-втором - PID вновь созданного процесса. Если fork возвращает 1, то это свидетельствует об ошибке (естественно, в этом случае возврат происходит только в процесс, выполнивший системный вызов и нового процесса не создаётся).

Пример программы, порождающей новый процесс приведён на рисунке 6. В строках 1-2 подключаются необходимые заголовочные файлы. В данном случае нам требуются функции ввода-вывода (<stdio.h>) и функции по работе с процессами (<unistd.h>). Порождение процесса

происходит в строке 7, после чего проверяется возвращаемое значение функции `fork`. Если новый процесс создан, то условие в строке 8 не выполнится и каждый процесс перейдет на строку 12. Далее, проверив значение переменной `pid`, дочерний процесс будет выполнять строки 13-17 (так как в его случае `pid = 0`), а родительский – строки 19-24. Затем оба процесса завершаться с кодом 0 (строка 25).

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main(void) {
5    int pid;
6    pid = fork();
7    if (pid == -1) {
8      perror("fork") ; exit(1);
9    }
10   if (pid == 0) { /*Эта часть кода выполняется дочерним процессом*/
11         printf("Потомок\n");
12     sleep(1);
13   } else { /*Эта часть кода выполняется родительским процессом*/
14     printf("Родитель. Создан потомок %u\n", pid);
15     sleep(1);
16   }
17   return (0);
18 }
```

Рисунок 6 – Исходный текст программы, использующей процессы

Особое внимание следует обратить на строки 16 и 22, в которых родительский и дочерний процессы добровольно переходят в «спящее» состояние на 1 секунду. Для чего это сделано? Ответ прост - для того чтобы оба процесса смогли вывести информацию в поток вывода, и она была отображена на экране терминала. Если этого не сделать, то может возникнуть ситуация, когда один из процессов (в силу алгоритмов планирования) завершит свою работу быстрее, чем другой начнет выводить информацию в поток вывода. В этом случае завершившийся процесс автоматически закроет все открытые файлы, которыми в нашем случае являются потоки ввода-вывода. В силу того, что файловые дескрипторы у родственных процессов одинаковые, ещё не завершившийся процесс начнет выдавать информацию в поток, у которого соответствующий файл уже закрыт. И на экран никакой информации выдано не будет.

7. Завершение процесса

Существует несколько способов завершения программы: возврат из функции `main()` (оператор `return`) и вызов функций `exit()`. Заголовок системного вызова `exit()` представлен на рисунке 7.

```
#include <unistd.h>
void exit (int);
int atexit(void (*func)(void));
```

Рисунок 7 – Заголовок функций завершения системного вызова и установки обработчика выхода

Аргумент `status`, передаваемый функции `exit`, возвращается родительскому процессу и представляет собой код возврата программы. При этом, считается, что программа возвращает `0` в случае «успеха» и другую величину в случае неудачи. Значение кода неудачи может иметь допол-

нительную трактовку, определяемую самой программой. Наличие кода возврата позволяет организовать условный запуск программ.

При завершении программы важную роль играет значение поля `PPID` дескриптора процесса. Если родительский процесс ещё не завершен, то операционная система не имеет права полностью удалить информацию о процессе, а вынуждена хранить статус завершения до тех пор, пока его не прочитает процесс-родитель. Как было сказано выше, такая ситуация приводит к появлению процессов-зомби.

Помимо передачи кода возврата, функция `exit` производит ряд действий, в частности выводит буферизированные данные и закрывает потоки ввода/вывода. Альтернативой ей является функция `_exit`, которая не производит вызовов библиотеки ввода/вывода, а сразу вызывает завершение процесса. Пример использования функции `_exit` приведён на рисунке 8. В приведённом примере функция `printf` ничего не выдаст на экран. Это связано с тем, что вывод на терминал в каноническом режиме по умолчанию буферизируется. Фактический вывод символов на экран осуществляется только после помещения в буфер символа конца строки или выполнения вызова `fflush()`.

```
1  #include <stdio.h>
2  #include <unistd.h>
3
4  int main (void){
5    printf ("Строка, которая не будет выведена.");
6    _exit(0);
7  }
```

Рисунок 8 – Исходный текст программы, использующей вызов `_exit()` для завершения процесса

Вызов `exit()` может быть вызван в любом месте программы. В некоторых ситуациях неожиданное завершение процесса может привести к необратимым последствиям. Например, если процесс занимается обработкой некоторой информации, начинает процедуру записи и завершается, не закончив её, то обрабатываемые данные могу быть испорчены. Чтобы избежать подобной ситуации процесс может зарегистрировать обработчики выхода (exit handler) - функции, которые вызываются функцией `exit()`, после выполнения стандартных действий, но до окончательного завершения процесса (вызова функции `_exit()`). Обработчики выхода регистрируются с помощью функции `atexit()`. Функцией `atexit()` может быть зарегистрировано до 32 обработчиков. На рисунке 9 проиллюстрированы возможные варианты завершения программы.

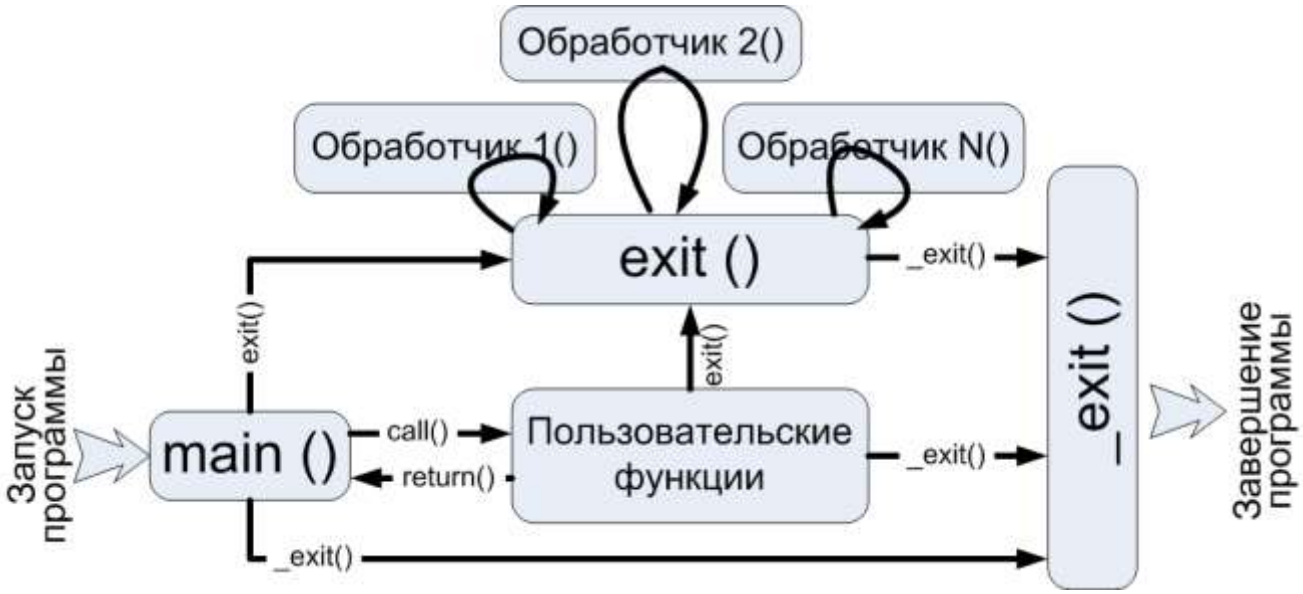

Рисунок 9 Механизмы завершения процессов

Обработчики выхода вызываются по принципу LIFO (последний зарегистрированный обработчик будет вызван первым) только при использовании вызова `exit()`. Если процесс завершается с использованием вызова `_exit()`, то обработчики выхода не вызываются.

8. Распределённые приложения. Технология MPI

Понятие процесса, как независимой сущности, выполняемой на вычислительных ресурсах, легла в основу технологии распределённых приложений. В таких приложениях, все элементы, по сути, являются процессами. При этом где выполняются эти процессы особого значения не имеют (они могут выполняться как на одной ЭВМ, так и на разных).

Для разработки распределённых приложений, объединяющих множество процессов, занимающихся вычислениями, стандартом стал MPI (англ. Message Passing Interface). Несмотря на то, что идея крупноблочного распараллеливания была предложена в СССР ещё в 1960-х годах, первая версия этого стандарта вышла в 1994 году.

8.1. Запуск распределённых приложений, использующих MPI

Для запуска приложений в пакетах MPI используется утилита `mpirun`. Задачами этой утилиты являются запуск заданного количества процессов на указанных узлах вычислительной системы, определить каждому из запущенных процессов свой логический номер и создать среду для передачи сообщений между этими процессами.

Чтобы указать утилите `mpirun` используется опция командной строки `-np`, а чтобы определить на каких узлах вычислительной системы необходимо запустить процессы используется опция `-H`. В вычислительных системах, использующих подсистемы управления ресурсами, утилита mpiexec конфигурируется таким образом, чтобы автоматически получить сведения о выделенных для выполнения программы ресурсах и поэтому указывать какие-либо параметры (кроме запускаемого файла) нет необходимости. Пример такого использования утилиты `mpiexec` представлен на рисунке 10.

```
[user@jet mpi]$ cat job.job
#PBS -N first-MPI
#PBS -l nodes=2:ppn=1
#PBS -j oe
cd $PBS_O_WORKDIR
mpiexec /usr/bin/hostname
[user@jet mpi]$ qsub job.job
10885.jet.cluster.local
[user@jet mpi]$ cat first-MPI.o10885
cn5
cn4
[user@jet mpi]$
```

Рисунок 10 – Пример запуска распределённого приложения с использованием MPI

8.2. Настройка приложения для использования MPI

Для того, чтобы приложение могло использовать стандарт MPI для обмена сообщения между процессами, каждый процесс должен «запустить» библиотеку. Для этого используется функция `MPI_init ()`, а для завершения работы с библиотекой используется функция `MPI_Finalize()`. Заголовки основных функций, настраивающих библиотеку и позволяющих получить процессу информацию о себе как о члене распределённого приложения, представлены на рисунке 11.

```
#include <mpi.h>

int MPI_Init      (int *, char **);
int MPI_Finalize  (void);
int MPI_Finalized (int *);
```

```
int MPI_Comm_rank (MPI_Comm, int *);
int MPI_Comm_size (MPI_Comm, int *);
int MPI_Get_processor_name (char *, int *);
```

Рисунок 11 – Заголовки функций библиотеки MPI по её настройке и некоторых функций получению информации о процессе

В качестве параметров для функции инициализации библиотеки передаются параметры командной строки запуска приложения. Это делается для того, чтобы пользователь имел возможность передать в командной строке дополнительные параметры для конфигурирования библиотеки MPI. В функцию `MPI_Comm_rank()` первым параметром передается указатель на структуру, описывающую пространство обмена информации, применяемой в MPI. По умолчанию в каждой программе создается одно единой для всех процессов пространство обмена сообщения (для ссылки на него используется макрос `MPI_COMM_WORLD`). Вторым параметром передается ссылка на переменную, в которую необходимо записать логический номер процесса. Аналогичным образом работает функция `MPI_Comm_size()`, возвращающая общее количество процессов в приложении. Функция `MPI_Get_processor_name()` возвращает имя узла, на котором выполняется процесс.

```
1  #include <mpi.h>
2  #include <stdio.h>
3
4  int main(int argc, char** argv) {
5      MPI_Init(NULL, NULL);    /* Initialize the MPI environment */
6
7      // Get the number of processes
8      int world_size;
9      MPI_Comm_size(MPI_COMM_WORLD, &world_size);
10
11     // Get the rank of the process
12     int world_rank;
13     MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
14
15     // Get the name of the processor
16     char processor_name[MPI_MAX_PROCESSOR_NAME];
17     int name_len;
18     MPI_Get_processor_name(processor_name, &name_len);
19
20     // Print off a hello world message
21     printf("Hello world from processor %s, rank %d"
              " out of %d processors\n",
              processor_name, world_rank, world_size);
22
23     // Finalize the MPI environment.
24     MPI_Finalize();
25 }
```

Рисунок 12 – Пример простейшей программы, использующей MPI

Задание на лабораторную работу

Базовые задания.

1. Продемонстрировать запуск фонового процесса в системах GNU\Linux.

2. Разработать приложение, порождающее несколько процессов и выводящих информацию о каждом из них. В каждом процессе должны быть выведены значения идентификаторы: PID, PPID, GID, EGID, UID, EUID и т.п.

3. Подготовить приложение, реализующее программу, представленную на рисунке 12. Разработать описание задачи, которой требуется для выполнения 3 вычислительных ядра и необходим запуск созданной программы. Запустите задачу на выполнение. Убедитесь, что приложение выдало правильные значения.

Основные задания.

1. Доработать программу, выполненную на этапе 3 простого задания так, чтобы выводилась информация о каждом процессе в MPI приложении (информация аналогична пункту 2 простого задания).

2. Разработайте программу умножения матриц с использованием библиотеки MPI. Каждая ветвь распределённого приложения должна выполнять умножение матриц (одинаковых). Продемонстрируйте, что результат умножения матриц во всех ветвях получился правильным. Сравните время расчета по всем ветвям.

Задания повышенной сложности.

1. Разработайте гибридное приложение (MPI+OpenMPI), реализующее алгоритм умножения матриц. Программа должна работать аналогично той, что разработана в п. 2 основных заданий и плюс к тому, использовать параллельную версию для каждой ветви. Сравните результаты с полученными при выполнении п. 2 основного задания.

Контрольные вопросы

1. Что такое процесс? Чем он отличается от потока выполнения?
2. Какие типы процессов (по способу выполнения) можно выделить в операционной системе?
3. Что такое «зомби»? Когда и как он появляется?
4. Каким образом пользователь операционной системы может узнать информацию о процессах? Как это можно сделать в ходе выполнения процесса?
5. Как создать новый процесс?
6. Как завершить процесс? Зачем и как используются обработчики завершения?
7. Что такое MPI? Зачем он используется?