Operating Systems

Inpter Process Communication

Me

March 21, 2016

План лекции

- Назначение IPC.
- Создание и завершение процессов.
- Сигналы.
- Pipe.
- Сокеты.
- Разделяемая память.
- ptrace.
- Другие виды IPC.

Назначение IPC

- У нас есть потоки - потоки делят все ресурсы:
  - ошибка в одном потоке может испортить жизнь всем.
- У нас есть процессы - процессы изолированны друг от друга:
  - ошибка в одном процессе не беспокоит других;
  - если мы хотим максимальной надежности - мы используем процессы, а не потоки;
- Если процессы не независимы
  - процессам может потребоваться передавать данные друг другу;
  - процессам может потребоваться синхронизация
    - не всегда можно полагаться на надежность процессов.

Создание процессов

- Способ создания процессов зависит от ОС:
  - Windows использует CreateProcess (есть еще NtCreateProcess и ZwCreateProcess);
  - UNIX-like ОС используют fork (мы тоже будем использовать его):

Создание процессов

- Способ создания процессов зависит от ОС:
  - Windows использует CreateProcess (есть еще NtCreateProcess и ZwCreateProcess);
  - UNIX-like ОС используют fork (мы тоже будем использовать его):
- fork - создает почти точную копию процесса вызвавшего fork
  - идентичная память (у каждого своя копия - не общая);
  - файловые дескрипторы;
  - процессы образуют дерево.

Создание процессов

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main()
6  {
7          const pid_t pid = fork();
8          if (pid < 0) {
9                  perror("failed to fork");
10                 exit(1);
11         }
12
13         if (pid == 0)
14                 printf("PROCESS %d says: I'm child\n", getpid());
15         else
16                 printf("PROCESS %d says: I'm parent of %d\n", getpid(),
                       ↪ pid);
17
18         return 0;
19 }
```

Завершение процессов

- В завершении процесса, как и в создании, участвуют двое:
  - завершающийся процесс, вызывает exit или _exit;
  - процесс дожидающийся завершения вызывает wait или waitpid.

Завершение процессов

- В завершении процесса, как и в создании, участвуют двое:
  - завершающийся процесс, вызывает exit или _exit;
  - процесс дожидающийся завершения вызывает wait или waitpid.
- Чтобы удалить процесс из системы родитель должен вызывать wait:
  - до тех пор, пока не вызван wait, завершившийся процесс находится в состоянии "zombie";
  - если родитель умер раньше ребенка, то ребенка усыновит/удочерит другой процесс системы (например, init);

Завершение процессов

```
1  #include <sys/types.h>
2  #include <sys/wait.h>
3
4  #include <unistd.h>
5  #include <stdlib.h>
6  #include <stdio.h>
7  #include <time.h>
8
9  int main()
10 {
11         const pid_t pid = fork();
12         if (pid < 0) {
13                 perror("failed to fork");
14                 exit(1);
15         }
16
17         srand(time(0));
18         if (pid == 0) {
19                 const int rc = rand() & 0xff;
20                 printf("PROCESS %d says: return %d\n", getpid(), rc);
21                 exit(rc);
22         } else {
23                 int status;
24                 waitpid(pid, &status, 0);
25                 printf("PROCESS %d says: My child's died, with return
                       ↪ value %d\n", getpid(), WEXITSTATUS(status));
26         }
27
28         return 0;
29 }
```

Сигналы

- Сигналы - сообщения о каких-либо исключительных событиях (как правило):
  - SIGSEGV - segmentation fault (не очень оно исклчительное);
  - SIGCHLD - статус дочернего процесса изменился;
  - SIGTERM и SIGKILL - процессу пора на покой;

Сигналы

- Сигналы - сообщения о каких-либо исключительных событиях (как правило):
  - SIGSEGV - segmentation fault (не очень оно исклчительное);
  - SIGCHLD - статус дочернего процесса изменился;
  - SIGTERM и SIGKILL - процессу пора на покой;
- обработчики сигналов асинхронны по отношению к основному коду:
  - в этом смысле они очень походи на прерывания;
  - в отличие от прерываний их можно сделать синхронными;

Сигналы

- Сигналы - сообщения о каких-либо исключительных событиях (как правило):
  - SIGSEGV - segmentation fault (не очень оно исклчительное);
  - SIGCHLD - статус дочернего процесса изменился;
  - SIGTERM и SIGKILL - процессу пора на покой;
- обработчики сигналов асинхронны по отношению к основному коду:
  - в этом смысле они очень походи на прерывания;
  - в отличие от прерываний их можно сделать синхронными;
- не все сигналы можно обработать или проигнорировать:
  - SIGTERM можно перехватить, а SIGKILL нельзя.

Пример обработчика сигналов

```
1  #include <sys/types.h>
2  #include <signal.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  static void try_it_again(int sig)
7  {
8          static const char msg[] = "I'm␣still␣alive!\n";
9          write(1, msg, sizeof(msg) - 1);
10 }
11
12 int main(void)
13 {
14         signal(SIGTERM, &try_it_again);
15
16         printf("I'm␣%d,␣kill␣me␣if␣you␣can!\n", (int)getpid());
17
18         while (1);
19
20         return 0;
21 }
```

Сигналы и конкурентность

- Как и прерывания, асинхронные сигналы являются источником конкуренции:
  - обработчик сигнала может прервать исполнение кода;
  - код может быть не готов к тому, что его прервут;

Сигналы и конкурентность

- Как и прерывания, асинхронные сигналы являются источником конкуренции:
  - обработчик сигнала может прервать исполнение кода;
  - код может быть не готов к тому, что его прервут;
- На обработчики сигналов налагаются ограничения:
  - например, в обработчике сигнала нельзя вызывать функцию printf;
  - многие функции стандартной библиотеки могут быть не "signal-safe";
  - создавать "signal-safe" функции возможно.

Маскировка сигналов

```
1  #include <sys/types.h>
2  #include <signal.h>
3  #include <unistd.h>
4  #include <stdio.h>
5
6  static void try_it_again(int sig)
7  {
8          static const char msg[] = "I'm␣still␣alive!\n";
9          write(1, msg, sizeof(msg) - 1);
10 }
11
12 static void block_signals(void)
13 {
14         sigset_t set;
15         sigfillset(&set);
16         sigprocmask(SIG_BLOCK, &set, NULL);
17 }
18
19 int main(void)
20 {
21         signal(SIGTERM, &try_it_again);
22
23         printf("I'm␣%d,␣kill␣me␣if␣you␣can!\n", (int)getpid());
24         block_signals();
25         while (1);
26
27         return 0;
28 }
```

Синхронная обработка сигналов

- Можно погрузить поток в сон в ожидании сигнала:
  - sigwait или sigwaitinfo позволяют дождаться нужных сигналов;
  - в многопоточных программах, не редко, создают поток для обработки сигналов:
    - при старте процесса все сигналы блокируются;
    - выделеный поток с помощью sigwait/sigwaitinfo ожидает сигналов и нотифицирует другие потоки, если нужно;

Синхронная обработка сигналов

- Можно погрузить поток в сон в ожидании сигнала:
  - sigwait или sigwaitinfo позволяют дождаться нужных сигналов;
  - в многопоточных программах, не редко, создают поток для обработки сигналов:
    - при старте процесса все сигналы блокируются;
    - выделеный поток с помощью sigwait/sigwaitinfo ожидает сигналов и нотифицирует другие потоки, если нужно;
- В Linux есть специальный системный вызов signalfd:
  - создает фиктивный файловый дескриптор, из которого можно "читать" сигналы;

Сигналы и IPC

Процессы могут посылать сигналы дургим процессам:

- SIGKILL, SIGTERM - чтобы "попросить" их умереть;
- SIGUSR1 и SIGUSR2 - специальные сигналы, выделенные под нужды приложений.

Отправка сигналов

```
1  #include <sys/types.h>
2  #include <signal.h>
3  #include <stdlib.h>
4  #include <stdio.h>
5
6  int main(int argc, char **argv)
7  {
8          if (argc < 2) {
9                  puts("Target process PID expected");
10                 exit(1);
11         }
12
13         for (int i = 1; i != argc; ++i) {
14                 const int pid = atoi(argv[i]);
15
16                 if (!pid) {
17                         printf("Wrong PID: %s\n", argv[i]);
18                         exit(1);
19                 }
20                 kill(pid, SIGKILL);
21         }
22
23         return 0;
24 }
```

Каналы (a. k. a. Pipes)

- Сигналы и коды возврата не позволяют передавать данные
  - в отсуствие общей памяти, это может стать проблемой;

Каналы (a. k. a. Pipes)

- Сигналы и коды возврата не позволяют передавать данные
  - в отсуствие общей памяти, это может стать проблемой;
- Чтобы передавать данные между процессами есть множество способов:
  - каналы или pipes должны быть вам уже хорошо известны;
  - вы можете создать участок общей памяти;
  - вы можете использовать сокеты;
  - вы можете использовать файлы (хотя это очень не удобно);

Pipes

- Pipe - это пара файловых дескрипторов:
  - в один дескриптор можно писать поток байт;
  - из другого млжно читать (очевидно, то что было записано в первый);
  - pipe имеет ограниченный буффер;
  - pipe не сохраняет границы сообщений (по-умолчанию);

Pipes

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

int main(void)
{
        char buf[4096];
        int size, fds[2];

        pipe(fds);
        if (fork()) {
                while ((size = read(0, buf, sizeof(buf) - 1)) > 0) {
                        buf[size] = '\0';
                        printf("Send: %s\n", buf);
                        write(fds[1], buf, size);
                }
        } else {
                while ((size = read(fds[0], buf, sizeof(buf) - 1)) > 0)
                    ↪ {
                        buf[size] = '\0';
                        printf("Received: %s\n", buf);
                }
        }
        close(fds[0]);
        close(fds[1]);

        return 0;
}
```

Named pipes

- Pipe можно сделать перманентным:
  - для этого существует вызов mkfifo;
  - кроме того есть команда mkfifo (можно вызвать из bash);

Named pipes

- Pipe можно сделать перманентным:
  - для этого существует вызов mkfifo;
  - кроме того есть команда mkfifo (можно вызвать из bash);
- Именованный канал выглядит почти как обычный файл:
  - в отличие от файла, чтение из канала удаляет данные;
  - смещение в канале не имеет смысла (seek не работает);
  - именованный канал не хранит данные на диске;

Сокеты

- Pipe-ы не позволяют отличать "отправителей":
  - вы не знаете, кто записал байт в pipe, вы видите только байт;

Сокеты

- Pipe-ы не позволяют отличать "отправителей":
  - вы не знаете, кто записал байт в pipe, вы видите только байт;
- Pipe-ы не сохраняют границы сообщений:
  - у вас есть поток байт, но вам не известно как он был создан;
  - в Linux можно создать pipe, который будет сохранять границы;

Сокеты

- Сокеты предоставляют более широкий выбор возможностей по сравнению с pipe-ами:
  - вы можете выбирать сохранять или нет границы сообщений (DGRAM vs STREAM);
  - вы можете узанать, от кого пришли данные;
  - вы можете передавать данные между компьютерами;

Сокеты

- Сокеты предоставляют более широкий выбор возможностей по сравнению с pipe-ами:
  - вы можете выбирать сохранять или нет границы сообщений (DGRAM vs STREAM);
  - вы можете узанать, от кого пришли данные;
  - вы можете передавать данные между компьютерами;
- В частности, сокеты предоставляют доступ к сетевым сервисам ОС:
  - сокеты это интрефейс, за ним не обязательно должна быть настоящая сеть;
  - но зачастую сокеты ассоциируются именно с сетевым взаимодействием.

Сокеты

- Для создания сокета используется вызов socket
  - при создании нужно указать "домен" или семейство протоколов;
  - нужно указать тип соединения (DGRAM или STREAM);
  - можно укзать конкретный проткол, который будет использован при передаче;

Сокеты

- Для создания сокета используется вызов socket
  - при создании нужно указать "домен" или семейство протоколов;
  - нужно указать тип соединения (DGRAM или STREAM);
  - можно укзать конкретный проткол, который будет использован при передаче;
- Примеры семейств протоколов:
  - AF_UNIX, AF_LOCAL, AF_NETLINK - скорее всего вам не знакомы;
  - AF_INET, AF_INET6 - известны вам под именами IPv4 (TCP/IP) и IPv6;
  - AF_BLUETOOTH - может быть и такое.

Тип соединения

- При создании сокета вы указываете тип соединения:
  - тип соединения позволяет выбрать нужный протокол передачи из семейства протоколов;
  - если с семейством протоколов AF_INET вы укажите тип SOCK_STREAM, то будет использован протокол TCP;
  - если укажите SOCK_DGRAM, то будет использован UDP.

Тип соединения

- При создании сокета вы указываете тип соединения:
  - тип соединения позволяет выбрать нужный протокол передачи из семейства протоколов;
  - если с семейством протоколов AF_INET вы укажите тип SOCK_STREAM, то будет использован протокол TCP;
  - если укажите SOCK_DGRAM, то будет использован UDP.
- В общем случае тип соединения описывает гарантии:
  - гарантии сохранения границ сообщений;
  - гарантии доставки - можем отправить и забыть, а можем ждать подтверждения доставки и переотправлять сообщения по таймаутам;

Адресация

- Чтобы отправлять и получать сообщения нам нужны адреса:
  - чтобы отправить сообщение именно на тот сокет, который нужно;
  - чтобы кто-то мог отправить сообщение на наш сокет;

Адресация

- Чтобы отправлять и получать сообщения нам нужны адреса:
  - чтобы отправить сообщение именно на тот сокет, который нужно;
  - чтобы кто-то мог отправить сообщение на наш сокет;
- Адрес, ествественно, зависит от семейства протоколов:
  - вы не можете полагаться на то, что bluetooth использует тот же формат адресов что и IP;
  - в частности для TCP/IP адрес состоит из IP адреса и порта;

Адресация

- Для указания адресов есть два вызова:
  - bind задает "наш" адрес - адрес, по которму могут связаться с нашим процессом;
  - connect задачет "чужой" адрес - адрес процесса, с которым мы хотим связаться;
  - сокет, таким образом, описывает соединение и определяется парой адресов и протоколом;

Адресация

- Для указания адресов есть два вызова:
  - bind задает "наш" адрес - адрес, по которму могут связаться с нашим процессом;
  - connect задачет "чужой" адрес - адрес процесса, с которым мы хотим связаться;
  - сокет, таким образом, описывает соединение и определяется парой адресов и протоколом;
- bind и connect нужны не всегда:
  - если вы инициируете соединение, то bind может быть не обязателен, а такой сокет часто называют клиентским;
  - если вы создаете сокет, чтобы кто-то мог подключиться к вам, то connect не нужен, а такой сокет часто называют серверным;

Пример: серверный сокет

- Создадим сокет использующий протокол TCP для соединения;
  - в качестве порта будем использовать 2016;
  - в качестве IP адреса возьмем 127.0.0.1;
  - не каждый порт можно использовать на свое усмотрение;

Пример: серверный сокет

- Создадим сокет использующий протокол TCP для соединения;
  - в качестве порта будем использовать 2016;
  - в качестве IP адреса возьмем 127.0.0.1;
  - не каждый порт можно использовать на свое усмотрение;
- серверный сокет используется для приема входящих соединений:
  - чтобы указать, что сокет используется для приема входящих соединений используется вызов listen;
  - чтобы дождаться входящего соединения используется вызов accept;
  - accept возвращает новый сокет, который описывает установленное соединение;

Пример: серверный сокет

```
1   struct sockaddr_in local;
2   const int port = 2016;
3   const int sk = socket(AF_INET, SOCK_STREAM, 0);
4
5   if (sk == -1) {
6           perror("failed to create socket");
7           exit(1);
8   }
9
10  local.sin_family = AF_INET;
11  local.sin_port = htons(port);
12  local.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
13
14  if (bind(sk, (const struct sockaddr *)&local, sizeof(local)) ==
        ↪ -1) {
15          perror("failed to bind socket");
16          exit(1);
17  }
18
19  if (listen(sk, 0) == -1) {
20          perror("listen failed");
21          exit(1);
22  }
```

Пример: серверный сокет

```
1         while (1) {
2                 struct sockaddr_in remote;
3                 socklen_t size = sizeof(remote);
4
5                 const int client = accept(sk, (struct sockaddr *)&remote
                      ↪ , &size);
6
7                 if (client == -1) {
8                         perror("accept failed");
9                         exit(1);
10                }
11
12                printf("Connection from %s:%d\n",
13                        inet_ntoa(remote.sin_addr),
14                        ntohs(remote.sin_port));
15
16                handle(client);
17                close(client);
18        }
```

Пример: клиентский сокет

- У нас есть сервер, пусть наш клиент подключается к нему:
  - т. е. в качестве удаленного адреса будем использовать 127.0.0.1:2016
  - локальный адрес указывать не обязательно;
- передавать будем параметры командной строки;

Пример: клиентский сокет

```
1   struct sockaddr_in remote;
2   const int port = 2016;
3   const int sk = socket(AF_INET, SOCK_STREAM, 0);
4
5   if (sk == -1) {
6           perror("failed to create socket");
7           exit(1);
8   }
9
10  remote.sin_family = AF_INET;
11  remote.sin_port = htons(port);
12  remote.sin_addr.s_addr = htonl(INADDR_LOOPBACK);
13
14  if (connect(sk, (const struct sockaddr *)&remote, sizeof(remote)
        ↪ ) == -1) {
15          perror("failed to connect socket");
16          exit(1);
17  }
18
19  for (int i = 0; i != argc; ++i)
20          write(sk, argv[i], strlen(argv[i]));
21
22  close(sk);
```

Разделяемая память

- Рассмотренные варианты IPC обладают рядом недостатков:
  - они либо не позволяют передавать данные вообще либо требуют копирования;
  - они требуют взаимодействия с ядром (системного вызова) на передачу и на получение данных
    - системные вызовы довольно дешевы;
    - но плохо, если приходится делать их слишком часто;

Разделяемая память

- Рассмотренные варианты IPC обладают рядом недостатков:
  - они либо не позволяют передавать данные вообще либо требуют копирования;
  - они требуют взаимодействия с ядром (системного вызова) на передачу и на получение данных
    - системные вызовы довольно дешевы;
    - но плохо, если приходится делать их слишком часто;
- Адресные пространтсва процессов по умолчанию отделены друг от друга
  - но мы можем попросить ОС создать общий участок памяти;
  - общая память самый дешевый способ взаимодействия.

Разделяемая память

- Мы можем попросить ОС выделить именованный участко памяти
  - по имени, этот участок памяти смогут найти другие процессы;
  - для создания участка разделенной памяти используется shm_open;
  - для удаления используется shm_unlink - вся память хранится до удаления участка или перзагрузки;

Разделяемая память

- Мы можем попросить ОС выделить именованный участко памяти
  - по имени, этот участок памяти смогут найти другие процессы;
  - для создания участка разделенной памяти используется shm_open;
  - для удаления используется shm_unlink - вся память хранится до удаления участка или перзагрузки;
- процессы могут просить ОС отобразить этот участок памяти в адресное пространство процесса
  - чтобы отобразить регион в адресное пространство используется mmap;
  - чтобы убрать отображение используется munmap;
  - с помощью mmap можно также отображать файлы в память - очень быстрый способ чтения/записи;

Разделяемая память

```
1   const char *mem_name = "shared_mem_name";
2   const int mem_size = 4096;
3   const int fd = shm_open(mem_name, O_CREAT | O_RDWR,
4           S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP);
5
6   if (ftruncate(fd, mem_size) == -1) {
7           perror("failed to truncate shared mem");
8           exit(1);
9   }
10
11  void *ptr = mmap(0, mem_size,
12          PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
13  if (ptr == MAP_FAILED) {
14          perror("mmap failed");
15          exit(1);
16  }
```

Process Trace

- Process Trace - не совсем IPC:
  - он позволяет передавать информацию из одного процесса в другой;

Process Trace

- Process Trace - не совсем IPC:
  - он позволяет передавать информацию из одного процесса в другой;
  - совершенно всю информацию - даже ту, которую процесс и не собирался передавать;

Process Trace

- Process Trace - не совсем IPC:
  - он позволяет передавать информацию из одного процесса в другой;
  - совершенно всю информацию - даже ту, которую процесс и не собирался передавать;
- Зачем нам нужен механизм, который раскрывает все внутренности процесса?

Process Trace

- Process Trace - не совсем IPC:
  - он позволяет передавать информацию из одного процесса в другой;
  - совершенно всю информацию - даже ту, которую процесс и не собирался передавать;
- Зачем нам нужен механизм, который раскрывает все внутренности процесса?
  - системный вызов ptrace существует для дебага.

Process Trace

- Вызов ptrace позволяет:
  - читать память процесса и состояние его регистров;
  - писать в память процесса или изменить значения регистров;
  - заставить останавливаться поток исполнения:
    - на входе/выходе в/из syscall-а;
    - после каждой инструкции;
- Но предварительно к процессу нужно подключиться:
  - мы можем подключиться к процессам, которые сами создали;
  - для подключения к другим процессам нужны привелегии (обычно);

Пример: отслеживаем системные вызовы

- Попытаемся отследить системные вызовы, которые выполняет программа
  - для этого мы запустим процесс и подключимся к нему;
  - попросим процесс оснавливаться на каждый системный вызов;
  - на входе в системный вызов номер вызова хранится в регистре rax (x86-64 only);
- Все это делается через один единственный вызов ptrace с разными параметрами;

Пример: отслеживаем системные вызовы

```
1         const pid_t child = fork();
2         if (!child) {
3                 char **args = calloc(argc, sizeof(char *));
4                 int i;
5
6                 for (i = 0; i != argc - 1; ++i)
7                         args[i] = strdup(argv[1 + i]);
8                 args[argc - 1] = 0;
9
10                ptrace(PTRACE_TRACEME, 0, 0, 0);
11                if (execvp(args[0], args) == -1)
12                        return 1;
13                return 0;
14        }
```

Пример: отслеживаем системные вызовы

```
1         if (child > 0) {
2                 const int syscall = SIGTRAP | 0x80;
3                 int status;
4
5                 waitpid(child, &status, 0);
6                 ptrace(PTRACE_SETOPTIONS, child, 0,
                      ↪ PTRACE_O_TRACESYSGOOD);
7                 while (!WIFEXITED(status)) {
8                         long rax;
9
10                        ptrace(PTRACE_SYSCALL, child, 0, 0);
11                        waitpid(child, &status, 0);
12
13                        if (!WIFSTOPPED(status) || WSTOPSIG(status) !=
                            ↪ syscall)
14                                continue;
15
16                        rax = ptrace(PTRACE_PEEKUSER, child, 8 *
                            ↪ ORIG_RAX, 0);
17                        printf("syscall␣%ld\n", rax);
18                        ptrace(PTRACE_SYSCALL, child, 0, 0);
19                        waitpid(child, &status, 0);
20                }
21        }
```

Другие виды IPC

- Есть другие классические UNIX-овые IPC:
  - семафоры - позволяют реализовать взаимное исключение;
  - очереди сообщений - как pipe-ы, но сохраняют границы сообщений + сообщения имеют "тип";
  - файловые блокировки - позволяют защитить доступ к файлу или его участкам;

Другие виды IPC

- Есть другие классические UNIX-овые IPC:
  - семафоры - позволяют реализовать взаимное исключение;
  - очереди сообщений - как pipe-ы, но сохраняют границы сообщений + сообщения имеют "тип";
  - файловые блокировки - позволяют защитить доступ к файлу или его участкам;
- Есть и не классические и не UNIX-овые варианты:
  - Windows Mailslots и Windows RPC;

Другие виды IPC

- Есть другие классические UNIX-овые IPC:
  - семафоры - позволяют реализовать взаимное исключение;
  - очереди сообщений - как pipe-ы, но сохраняют границы сообщений + сообщения имеют "тип";
  - файловые блокировки - позволяют защитить доступ к файлу или его участкам;
- Есть и не классические и не UNIX-овые варианты:
  - Windows Mailslots и Windows RPC;
- Есть целые библиотеки обмена сообщениями между процессами:
  - ZeroMQ, RabbitMQ;