Operating Systems

Interprocess communication

Me

November 10, 2016

Назначение IPC

- Потоки vs процессы:
  - потоки одного процесса разделяют все ресурсы - они зависимы;
  - потоки разных процессов не разделяют ресурсов по-умолчанию.
- Процессы по-умолчанию надежны:
  - при правильном дизайне падение одного процесса не потревожит другой;
  - упавший процесс всегда можно перезапустить.

Назначение IPC

- Процессам приходится взаимодействовать:
  - чтобы выполнять общую задачу необходимо взаимодействовать;
  - чтобы иметь доступ к общим ресурсам необходима синхронизация.
- IPC - набор примитивов для взаимодействия процессов:
  - обмен сообщениями;
  - синхронизация.

Виды IPC

- Виды IPC определяются ОС и ее архитектуры:
  - классические UNIX IPC;
  - невнятные Windows IPC;
  - L4 IPC (в каком-то смысле тоже классические).
- Мы будем рассматривать классические UNIX IPC.

Создание процесса

- fork - вызов создающий почти точную *копию* процесса:
  - новый процесс имет свой уникальный идентификатор;
  - в новом процессе будет копия только одного исходного потока (того что вызвал fork);
  - есть некоторые тонкости с открытыми файлоыми дескрипторами.

Пример

```
1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4
5  int main()
6  {
7          const pid_t pid = fork();
8
9          if (pid < 0) {
10                 perror("fork␣failed");
11                 exit(1);
12         }
13
14         if (!pid)
15                 printf("Process␣%d␣says:␣I'm␣child\n",
16                                         getpid());
17         else
18                 printf("Process␣%d␣says:␣I'm␣parent␣of␣%d\n",
19                                         getpid(), pid);
20
21         return 0;
22 }
```

Завершение процессов

- Завершение процесса состоит из двух частей:
  - процесс должен завершиться или совершить ошибку;
  - другой процесс должен дождаться пока процесс завершиться.
- Чтобы завершиться процесс может вызвать exit или _exit:
  - как аргумент принимается некоторый код возврата;
  - return из main за сценой приводит к вызову exit.

wait

- Чтобы дождаться завершения процесс может воспользоваться wait или waitpid:
  - до тех пор пока кто-то не вызовет wait/waitpid процесс находится в состоянии *zombie* - не жив и не мертв;
  - зачастую wait вызывает процесс-родитель;
  - если родитель умер, то ребенка усыновит/удочерит/etc другой процесс системы.
- waitpid на самом деле дожидается не завершения процесса, а изменения его состояния:
  - процессмы могу не только умирать, но и "останавливаться" и "продолжать".

Пример

```
1           if (!pid) {
2                   const int rc = rand() & 0xff;
3
4                   printf("Process␣%d␣says:␣return␣%d\n",
5                                                   getpid(), rc);
6                   exit(rc);
7           } else {
8                   int status;
9
10                  waitpid(pid, &status, 0);
11                  printf("Process␣%d␣says:␣my␣child's␣died␣with␣value␣%d\n
                        ↪ ",
12                                                  getpid(), WEXITSTATUS(status));
13          }
```

Сигналы

- Сигналы - это сообщения, которые получает процесс в некоторых предвиденных и не очень случаях:
  - кто-то послал процессу сигнал с помощью вызова kill (говорящее название);
  - процесс сам себе послал сигнал с помощью вызова abort (опять же говорящее название);
  - процесс сделал какую-то гадость:
    - доступ к не своей/не существующей/не выровненной памяти;
    - попытка выполнить недопустимую инструкцию;
  - другие менее/более противные ситуации.

Сигналы

- При получении сигнала процесс (или скорее от имени процесса) выполняется некоторое действие:
  - типичные действия - проигнорировать сигнал, упасть или упасть красиво с core dump-ом;
  - для некоторых типов сигналов действия можно переопределить: например мы можем перехватить CTRL+C.

Пример

```
 1  int main()
 2  {
 3          struct sigaction sa;
 4
 5          memset(&sa, 0, sizeof(sa));
 6          sa.sa_sigaction = &handle_term;
 7          sa.sa_flags = SA_SIGINFO;
 8
 9          if (sigaction(SIGTERM, &sa, NULL) < 0) {
10                  perror("sigaction␣failed");
11                  exit(1);
12          }
13
14          if (sigaction(SIGINT, &sa, NULL) < 0) {
15                  perror("sigaction␣failed");
16                  exit(1);
17          }
18
19          while (1)
20                  sleep(1);
21
22          return 0;
23  }
```

Пример

```
1  static void handle_term(int no, siginfo_t *info, void *context)
2  {
3          static const char msg[] = "Stil␣alive!\n";
4
5          (void) no;
6          (void) info;
7          (void) context;
8
9          write(1, msg, sizeof(msg) - 1);
10 }
```

Ограничения на обработку сигналов

- Обработчики сигналов вызываются асинхронно (в некотором смысле) относительно кода потока:
  - не все функции можно безопасно вызывать из обработчика сигнала;
  - например, printf нельзя, по крайней мере так говорит POSIX.
- POSIX требует чтобы довольно много функций были signal-safe
  - но разные библиотеки и ОС позволяют себе всякие вольности;
  - например, glibc разрешает longjmp из обработчика сигнала - можно использовать, чтобы реализовать потоки на уровне приложения, а не ОС.

Блокировка сигналов

- Некоторые сигналы можно блокировать
  - сигнал не будет доставлен процессу пока его не разблокируют.
- Для блокировки сигналов рекомендуется использовать sigprocmask
  - функция принимает битовую маску и действие (например, заблокировать все сигналы в маске, или разблокировать).

Пример

```
1         sigset_t mask, orig;
2
3         sigemptyset(&mask);
4         sigaddset(&mask, SIGTERM);
5
6         if (sigprocmask(SIG_BLOCK, &mask, &orig) < 0) {
7                 perror("sigprocmask␣failed");
8                 exit(1);
9         }
```

```
1         if (sigprocmask(SIG_SETMASK, &orig, NULL) < 0) {
2                 perror("sigprocmask␣failed");
3                 exit(1);
4         }
```

Каналы (Pipes)

- Сигналы и коды возврата не позволяют передавать произвольные данные
  - память у каждого процесса своя по-умолчанию;
  - не пригодны для передачи данных - нет возможности обмениваться данными.
- Для передачи данных между процессами UNIX IPC предоставляет много способов:
  - можно создать участок общей памяти разделяемый процессами;
  - сокеты - зачастую сеть, но есть специальные UNIX сокеты;
  - файлы - можно читать/писать одни и те же файлы;
  - каналы.

Каналы

- pipe - это пара файловых дескрипторов:
  - один дескриптор можно использовать для записи;
  - другой дескриптор можно использовать для чтения, того что было записано в первый;
  - pipe имеет ограниченный буффер, так что записать в pipe не блокируясь произвольное количество данных не получится;
  - pipe не сохраняет границы сообщений (по-умолчанию).

Пример

```
1         int fd[2];
2
3         if (pipe(fd) < 0) {
4                 perror("pipe␣failed");
5                 exit(1);
6         }
```

```
1                 while ((size = read(0, buf, sizeof(buf) - 1)) > 0) {
2                         buf[size] = 0;
3                         printf("Send:␣%s\n", buf);
4                         write(fd[1], buf, size + 1);
5                 }
```

```
1                 while ((size = read(fd[0], buf, sizeof(buf) - 1)) > 0)
2                         printf("Received:␣%s\n", buf);
```

Разделяемая память

- Каждый процесс имеет собственное адресное пространство по-умолчанию
  - в этом и заключается основа безопасности процессов.
- Однако по требованию процессы могут создавать участки общей памяти:
  - обмен данными через общую память позволяет избежать копирования при передаче данных;
  - однако не предоставляет по-умолчанию возможности синхронизации (вам нужно реализовать синхронизацию самостоятельно).

Разделяемая память

- Чтобы попросить ОС создать/удалить именованный участок памяти используются shm_open/shm_unlink (open/unlink - создание/удаление памяти)
  - участок памяти хранится до удаления или перезагрузки.
- Чтобы иметь доступ к разделяемой памяти как к обычной памяти необходимо отобразить ее в адресное пространство процесса
  - для отображения региона используется mmap;
  - munmap - обратная к mmap операция (удаляет отображение);
  - участок разделяемой памяти в разных процессах может быть отображен в разные места адресного пространства.

Пример

```
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 (fd < 0) {
7           perror("shm_open failed");
8           exit(1);
9   }
10
11  if (ftruncate(fd, mem_size) == -1) {
12          perror("ftruncate failed");
13          exit(1);
14  }
15
16  void *ptr = mmap(/* addr hint = */0,
17                   /* size =  */mem_size,
18                   /* protection = */PROT_READ | PROT_WRITE,
19                   /* flags (private/shared) = */MAP_SHARED,
20                   /* file descriptor */fd,
21                   /* offset = */0);
22  if (ptr == MAP_FAILED) {
23          perror("mmap failed");
24          exit(1);
25  }
```

Process Trace (ptrace)

- Process Trace - не совсем IPC:
  - ptrace позволяет оожному процессу отслеживать другой процесс;
  - отслеживать состояние его памяти и регистров;
  - менять состояние его памяти и регистров;
  - т. е. ptrace нарушает изоляцию процессов.
- Зачем нам нарушать изолцию процессов?
  - очевидно, что это нужно для отладки;
  - обычно кто попало не может отслеживать другой процесс.

Пример

```
1         const pid_t pid = fork();
2
3         if (pid < 0) {
4                 perror("fork failed");
5                 exit(1);
6         }
7
8         if (!pid) {
9                 char **args = calloc(argc, sizeof(char *));
10
11                for (int i = 0; i != argc - 1; ++i)
12                        args[i] = strdup(argv[i + 1]);
13                args[argc - 1] = 0;
14
15                ptrace(PTRACE_TRACEME, 0, 0, 0);
16                if (execvp(args[0], args) < 0)
17                        exit(1);
18                return 0;
19        }
```

Пример

```
1         const int syscall = SIGTRAP | 0x80;
2         int status;
3
4         waitpid(pid, &status, 0);
5         ptrace(PTRACE_SETOPTIONS, pid, 0, PTRACE_O_TRACESYSGOOD);
```

```
1         while (!WIFEXITED(status)) {
2                 struct user_regs_struct regs;
3
4                 ptrace(PTRACE_SYSCALL, pid, 0, 0);
5                 waitpid(pid, &status, 0);
6
7                 if (!WIFSTOPPED(status) || WSTOPSIG(status) != syscall)
8                         continue;
9
10                if (ptrace_get_regs(pid, &regs) < 0) {
11                        kill(pid, SIGKILL);
12                        waitpid(pid, &status, 0);
13                        exit(1);
14                }
15                printf("syscall␣regs:\n");
16                ptrace_dump_x86_64_regs(&regs);
17
18                ptrace(PTRACE_SYSCALL, pid, 0, 0);
19                waitpid(pid, &status, 0);
20        }
```

Пример

```
1  static int ptrace_get_regs(pid_t pid, struct user_regs_struct *regs)
2  {
3          static const size_t word_size = sizeof(unsigned long);
4          static const size_t data_size = sizeof(*regs);
5
6          unsigned long *ptr = (unsigned long *)regs;
7
8          for (size_t word = 0; word != sizeof(*regs)/word_size; ++word) {
9                  errno = 0; /* thanks to stupid glibc wrapper */
10                 ptr[word] = ptrace(PTRACE_PEEKUSER, pid, word *
                       ↪ word_size, 0);
11                 if (errno) {
12                         perror("ptrace␣PEEKUSER␣failed");
13                         return -1;
14                 }
15         }
16         return 0;
17 }
```

Другие виды IPC

- Механизмы UNIX IPC не ограничиваются представленными вариантами:
  - очереди сообщений;
  - семафоры (что-то вроде межпроцессных локов);
  - файлы и файловые локи;
  - сокеты (зачастую для общения по сети);
  - UNIX domain socket-ы специальный вид сокетов, который позволяет, например, передавать файловые дескрипторы между процессами.
- Различные библиотеки обмена сообщениями:
  - ZeroMQ;
  - MPI;
  - любая другая библиотека/реализация (как правило они будут пользоваться сетью).

Q&A