Терентьев Алексей Борисович

# Межпроцессное взаимодействие(IPC)

Материал
для лабораторных работ по курсу
ОПЕРАЦИОННЫЕ СИСТЕМЫ

Санкт-Петербургский государственный политехнический университет

# Компиляция и сборка

# Библиотеки

- Статические(.a). Объектный код библиотеки помещается в исполняемый файл программы
- Динамические(.so)
  - Динамически линкуемые. Должны быть доступны во время компиляции/линковки. Не включаются в исполняемый файл, но должны быть доступны для запуска.
  - Динамически загружаемые и линкующиеся во время работы с помощью системных функций [1]
- Линковка со статическими библиотеками:

```
libctest.a # имя содержит префикс lib
gcc -o exec.f file.c libctest.a
gcc -o exec.f file.c -L/path/to/library-directory -lctest
gcc file.c -lm -lrt # libm, librt
```

[1] http://www.yolinux.com/TUTORIALS/

# Опции компилятора

- `-w` - отключить сообщения ворнингов;
- `-Werror` - ворнинги становятся ошибками;
- `-Wfatal-errors` - компиляция прекращается на первой ошибке;
- `-Wall` - включает некоторый набор ворнингов[2];
- `-E` - вывод рез-та работы препроцессора, `-S` - ассемблер, `-save-temps` - сохранить все промежуточные файлы;
- `-O1, -O2, -O3` - уровни оптимизации, по умолчанию -O0;

[2] https://gcc.gnu.org/onlinedocs/gcc/Warning-Options.html

# Makefile

# Утилита make

make [ -f make-файл ] [ цель ] выполняет команды из make-файла для создания указанной цели

- all — выполнить сборку пакета;
- install — установить пакет из дистрибутива (производит копирование исполняемых файлов, библиотек и документации в системные каталоги);
- uninstall — удалить пакет (производит удаление исполняемых файлов и библиотек из системных каталогов);
- clean — очистить дистрибутив (удалить из дистрибутива объектные и исполняемые файлы, созданные в процессе компиляции);

# make-файл

make-файл состоит из правил и переменных. Правила имеют следующий синтаксис:

```
цель1 цель2 ...: реквизит1 реквизит2 ... команда1 команда2 ...
```

- Реквизит - файл, от которого зависит данная цель(проверяется существование)
- Строки, в которых записаны команды, должны начинаться с символа табуляции.
- По умолчанию используется самая первая цель

# Пример 1. make-файл

```
all: hello

first.f: main.o file.o
	g++ main.o file.o -o first.f

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

file.o: file.cpp
	g++ -c file.cpp

clean:
	rm -rf *.o first.f
```

# Переменные в make-файлах

Переменная = значение

```
CC = gcc
CFLAGS = -O2
OBJECTS = main.o foo.o

main.exe : $(OBJECTS)
$(CC) $(CFLAGS) $(OBJECTS) -o main.exe

main.o : main.c
$(CC) $(CFLAGS) -c main.c

foo.o : foo.c
$(CC) $(CFLAGS) -c foo.c
```

# Паттерны в правилах

Переменная = значение

```
# From %.file to %.f
%.f : %.file
cat $< > $@
```

```
TARG_FILES = $(shell find . -name “v1_*.cpp”)
ALL_TARGS=$(TARG_FILES:%.cpp=%.e)
# @echo $(ALL_TARGS)
all: $(ALL_TARGS)
```

# Общая память

# Общая память. shmget. seg

Самый быстрый вид взаимодействия

```
//создание или получение доступа к сегменту
int shmget(key_t key, int size, int shmflg)
// attach
void* shmat(int shmid, const void* shmaddr,
            int shmflg)
// detach
int shmdt(const void* shmaddr)
// операции с сегментом:
удаление, блокировка, изменение
int shmctl(int shmid, int cmd,
           struct shid_ds* buf)
```

# Общая память. shmget

**процесс1**

```
key=1234;
int shmid = shmget(key, bufSize, rights | IPC_CREAT);
//создать сегмент
char *shm = (char*)shmat(shmid, 0, 0);
//use shm
shmdt(shm);
shmctl(shmid, IPC_RMID, NULL);//удалить сегмент
```

**процесс2**

```
int shmid = shmget(key, bufSize, 0);
char *shm = (char*)shmat(shmid, 0, 0);
//use shm
shmdt(shm);
```

# Общие файлы

# Общие файлы. shm_open. shm

```
// создать или открыть shared memory object
int shm_open(const char *name, int flags,
             mode_t mode))
// замапить fd в память
void* mmap(void *start_addr, size_t len,
     int protection, int flags, int fd,
     off_t offset)
// удалить shared memory object
// если он не используется другими процессами
int shm_unlink(const char *name)
```

# Общие файлы с shm_open

```
int fd = shm_open( c_fname, O_CREAT | O_EXCL | O_RDWR,
S_IRWXU | S_IRWXG);
ftruncate( fd, seg_size );
addr = (char*)mmap( 0, seg_size, PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0 );
//use addr
shm_unlink(c_fname)

int fd = shm_open( c_fname, O_RDWR, S_IRWXU | S_IRWXG);
char *addr =(char*)mmap( 0, c_seg_size,
PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0 );
//use addr
```

# Anonymous mapping. fls

- Подходит только для процессов-родственников
- Не требует наличия разделяемого объекта
- Два способа:
  - **BSD**. Передавать в *mmap* -1 в качестве *fd* и использовать флаг *MAP_ANON*. Не использует файловую систему.
  - **System V**. Открыть файл */dev/zero* и передавать его в *mmap*. Не работает в OS X.

# Anonymous mapping

```
//первый способ, MAP_ANON
addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED | MAP_ANON, -1, 0);

//второй способ, /dev/zero
fd = open("/dev/zero", O_RDWR);
addr = mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE,
MAP_SHARED, fd, 0);
```

# Очереди сообщений

# Очереди сообщений. mq

```
// создает или открывает очередь
mqd_t mq_open(const char *name, int flags,
  ... [ mode_t mode, struct mq_attr *mq_attr ])
// закрывает очередь
int mq_close(mdq_t mqdes)
// посылает некоторое сообщение с
 определенным приоритетом
int mq_send(mqd_t mqdes, const char *msgbuf,
size_t len, unsigned int prio)
// берет из очереди самое старое сообщение с
наивысшим приоритетом
ssize_t mq_recieve(mqd_t mqdes, char *buf,
size_t len, unsigned *prio)
```

```
int flags = O_RDWR | O_CREAT;
mode_t mode = rights;
struct mq_attr attr;
attr.mq_flags = 0;
attr.mq_maxmsg = 10;
attr.mq_msgsize = sizeof(Msg);
attr.mq_curmsgs = 0;
mqd_t qid = mq_open(c_name, flags, mode, &attr);
Msg msg = init msg;
mq_send(qid, (char*) &msg, sizeof(Msg), 0);

mqd_t msgq_id = mq_open(c_name, O_RDWR);
struct mq_attr msgq_attr;
mq_getattr(msgq_id, &msgq_attr);
msgsz = mq_receive(msgq_id, (char*)&msg, sizeof(msg), &sender);
```

# Программные каналы

# Программные каналы. pipe

- Самый простой и самый быстрый в реализации способ межпроцессного взаимодействия.
- Как и все безымянные виды взаимодействия, подходит лишь для процессов-родственников.
- Работают только в одном направлении
- Атомарность гарантируется системой, если размер буфера не превышает *PIPE_BUF* байт
- Наполовину открытые pipe считаются *овдовевшими, сломанными(widowed, broken)*. Запись туда приводит к ошибке, чтение возвращает 0 байт.

# Программные каналы. pipe

Родственные процессы

```
pipe(fd);//common

close(fd[0]);
Msg msg;
write(fd[1], (char*)&msg, sizeof(msg));
close(fd[1]);

close(fd[1]);
Msg msg;
int nbytes=read(fd[0], (char*)&msg, sizeof(msg));
close(fd[0]);
```

# Именованные каналы

# Именованные каналы. mkfifo

- Работают практически так же, как и pipes, но поддерживают взаимодействие между независимыми процессами.
- Переживают смерть создателя, необходимо явное удаление

```
// создает файл для работы
int mkfifo(char *path, mode_t mode)
```

# Именованные каналы

Может работать с помощью команды mkfifo оболочки.

```
mkfifo("myfifo rights);

int fp;
fp = open(“myfifo”,O_WRONLY);
write(fp,r,size);
close(fp);

fz = open(“myfifo”,O_RDONLY);
read(fz,p,size);
close(fz);
```

# Дополнительно

# Дополнительно изучить

- Команды bash:
  - ipcs
  - mkfifo
- errno, strerror( err )

# Полезные ссылки

Книги:

- Gray, J. *Interprocess Communications in Linux®: The Nooks & Crannies*. Prenties Hall PTR. 2003
- Полезные примеры.
  `http://mij.oltrelinux.com/devel/unixprg/`
- По поводу makefile'ов.
  `http://mrbook.org/blog/tutorials/make/`