Computer Architecture and Operating Systems
Lecture 9: Inter-Process Communication

Andrei Tatarnikov

atatarnikov@hse.ru
@andrewt0301

Inter-Process Communication

- Files
- Pipes
- Signals
- Message Queues
- Shared Memory

Signals

- Asynchronous
- One-byte
- Delivered by OS (**kill** system call and utility)
- Can be caught by OS or the process itself
- Examples: Ctrl-C = **SIGINT**, Ctrl-\ = **SIGQUIT**, Ctrl-Z = **SIGTSTP**

Never Ending Program

Example program to be managed by signals:

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

int main(int argc, char *argv[]) {
	int i;
	for(i=0;; i++) {
		sleep(1);
		printf("%d\n", i);
	}
	return 0;
}
```

Kill

- **kill** utility — send a signal to a process
  - kill -l
  - ⇒ (slightly) platform-depended
  - kill -SIGNAL
  - example: suspend (**STOP**) / continue (**CONT**)
  - kill never-ending program with just **kill**, kill -**HUP**, 9, **SEGV** :), **STOP**, and **CONT**
- Types of processes (just a convention, both types runs by **fork**()/**exec**())
  - interactive process: ⩽1 at each terminal can input and output to the terminal
  - background process (runs from shell with '**&**'): any number can only output to the terminal
- Changing type:
  - **^Z** to stop, **fg** to continue, **bg** to continue in background (complex)
  - When background process inputs from tty, in immediately **STOP**ped, we can **fg** it

Sending Signals: System Call Kill

- Send a signal: see **kill** system call at https://www.man7.org/

```
#include <stdio.h>
#include <sys/types.h>
#include <signal.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    if (kill(atoi(argv[1]), atoi(argv[2])))
        perror("Can not kill");
    return 0;
}
```

Try to kill foreign or non-existent process

Handling Signals

```
#include <stdio.h>
#include <unistd.h>
#include <signal.h>

void handler(int sig) {
    printf("Caught %d\n", sig);
}

int main(int argc, char *argv[]) {
    signal(SIGINT, handler);
    signal(SIGSEGV, handler);
    int i;
    for(i=0;; i++) {
      sleep(1);
      printf("%d\n", i);
    }
    return 0;
}
```

- Handler (**signal**):
  - needs to be registered
  - not all signals can be handled (e. g. **9** and **STOP**/**CONT**)
  - permission restrictions (by process UID)

Looking After Child Processes

```
#include <stdio.h>
#include <wait.h>
#include <signal.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int stat;
    pid_t pid;
    if ((pid = fork()) == 0) {
        while(1);
    } else {
        printf("Forking a child: %d\n", pid);
        wait(&stat);
        printf("And finally…\n");
        if (WIFSIGNALED(stat))
            psignal(WTERMSIG(stat), "Terminated:");
        printf("Exit status: %d\n", stat);
    }
    return 0;
}
```

- See wait for **WIFSIGNALED/WTERMSIG** macros
- See **psignal**

Message Queues

- Base manpage: **mq_overview** at https://www.man7.org/
- What we need for messaging:
  - Synchronous
  - Can store content
  - Can be queued
  - Can be prioritized
  - Every message is delivered over certain queue

Creating Message Queue

```
#include <mqueue.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>

int main(int argc, char *argv[]) {
    mqd_t mqd;
    struct mq_attr attr;

    attr.mq_maxmsg = 10;
    attr.mq_msgsize = 2048;
    mqd = mq_open(argv[1], O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR, attr);

    return 0;
}
```

- Queue is for **10** messages **2048** bytes each
- Queue is creating for read/write, if there is no queue with the same name, or else an error is generated
- Omitting **O_EXCL** allows to re-create a queue with the same name, purging all messages, whit is probably not a good idea

Sending Messages

```
#include <mqueue.h>
#include <fcntl.h>
#include <string.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    mqd_t mqd;
    unsigned int prio;

    mqd = mq_open(argv[1], O_WRONLY);
    prio = atoi(argv[2]);
    mq_send(mqd, argv[3], strlen(argv[3]), prio);
    return 0;
}
```

- Priority varies from 0 (lowest) to system-depended maximum (at least **31**, **32767** in Linux)
- Message content is a byte array, it does not have to be zero-terminating string
- POSIX queue provides prioritization mechanism. Earliest massage from higher priority messages subset is to be delivered first.

Receiving Messages

```
#include <mqueue.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
    mqd_t mqd;
    unsigned int prio;
    void *buf;
    struct mq_attr attr;
    ssize_t n;

    mqd = mq_open(argv[1], O_RDONLY);
    mq_getattr(mqd, &attr);

    buf = malloc(attr.mq_msgsize);
    n = mq_receive(mqd, buf, attr.mq_msgsize, &prio);
    printf("Read %ld bytes; priority = %u\n", (long) n, prio);
    free(buf);
    return 0;
}
```

- Knowing nothing about message size, program must retrieve this value from queue attributes to provide an appropriate space in read buffer.
- There's no mechanism of message typification, so only size is printed
- To remove a queue call **mq_unlink(name)**
- POSIX message API is implemented in **librt** library, so compile program with -**lrt** option.

Notifying

- Every **mq_receive** call returns a message if there's one. If queue is empty, **mq_receive**() can wait for message or return with fail status, depending on O_NONBLOCK flag.

- There's alternate method to notify program by signal: a program calls **mq_notify** to subscribe on certain queue. Every time message is arrived in queue, the program gets a signal described in **mq_notify**() and can handle message asynchronously.

Memory Mapping

- Kernel has a paging mechanism. When memory is limited, some memory pages can be swapped out. When a program needs one of them:
  - TLB produces page miss (no physical memory is provided for the virtual address);
  - Kernel a loads corresponding page from disk and links to virtual memory page.
- If paging out a .text section, there is no need to provide a space on swap, because this data is already on disk — e. g. in the binary program file, from which the process was started.
- More general process of mapping file to memory is called memory map.
- System call **mmap** asks kernel to map selected file to the virtual memory address range. After this done, the range can be used as an ordinary array filled with file's contents. The file has not to be read into memory completely, Linux use paging mechanism to represent corresponded file parts.

Memory Mapping : System Calls

Example of simple cat analog, that memory-maps file and than just writes it to **stdout**:

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <stdio.h>
#include <fcntl.h>

int main(int argc, char *argv[]) {
    char *addr;
    int fd;
    struct stat sb;
    fd = open(argv[1], O_RDONLY);
    fstat(fd, &sb);
    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
    fwrite(addr, 1, sb.st_size, stdout);
    return 0;
}
```

- **PROT_READ** means that memory-mapped pages can only be read by the program
- **MAP_PRIVATE** means the program observe some fixed state of the file
- write to memory-mapped area does not change the file itself
- program supposes file can not be changed while memory-mapped in **MAP_PRIVATE** mode
- **fstat** is used to determine file size (it discovers other file properties as well)

Shared Memory

- See page **shm_overview** at https://man7.org
- Multiple processes can have some of their virtual memory pages translated to the same physical page. Then they can communicate through this shared area called shared memory.
- POSIX shared memory implemented over memory-mapped file abstraction.
- First we need to open named shared memory object (shared memory analog of queue, shmobj for short). Programs can memory-map this object, read and write to it.

Shared Memory : Create

Open named *shared memory object* (shared memory analog of queue, *shmobj* for short). Programs can **mmap** this object, read and write to it.

```
#include <stdio.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <stdlib.h>

int main(int argc, char *argv[]) {
  int fd;
  size_t size;
  void *addr;

  fd = shm_open(argv[1], O_RDWR|O_CREAT|O_EXCL, S_IRUSR|S_IWUSR);
  size = atol(argv[2]);
  ftruncate(fd, size);

  addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);
  return 0;
}
```

- Modes and permissions of the object are the same as when creating a queue.
- There is no sense in having newly created shmobj size other than zero, so **ftruncate**() call.
- We call **mmap**() for declaring the object shared (with **MAP_SHARED**, of course).

Shared Memory : Write

To write to the shared memory, program opens shmobj, mmaps it and uses the memory as ordinary array

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd;
    size_t len;
    char *addr;

    fd = shm_open(argv[1], O_RDWR, 0);
    len = strlen(argv[2]);
    ftruncate(fd, len);

    addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
    close(fd);

    printf("Copying %d bytes\n", len);
    memcpy(addr, argv[2], len);
    return 0;
}
```

- There is no difference if you open shmobj for reading/writing or just writing, it is memory
- The smobj descriptor only needed when opening shmobj, we can close it just after **mmap**()

Shared Memory : Read

To read from shared memory, the program opens shmobj and tread it like mmapped file:

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>

int main(int argc, char *argv[]) {
    int fd;
    char *addr;
    struct stat sb;

    fd = shm_open(argv[1], O_RDONLY, 0);
    fstat(fd, &sb);
    addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
    close(fd);

    fwrite(addr, 1, sb.st_size, stdout);
    printf("\n… Done");
    return 0;
}
```

- Note **fstat** can be used to determine shared memory size as well as to determine file size
- As usual, to stop using shmobj, one shall unlink it with **shm_unlink**(name)

Any Questions?

```
                .text
__start:        addi t1, zero, 0x18
                addi t2, zero, 0x21
cycle:          beq t1, t2, done
                slt t0, t1, t2
                bne t0, zero, if_less
                nop
                sub t1, t1, t2
                j cycle
                nop
if_less:        sub t2, t2, t1
                j cycle
done:           add t3, t1, zero
```