UNIX inter-process communication

Tamás Mészáros
http://www.mit.bme.hu/~meszaros/

Department of Measurement and Information Systems
Budapest University of Technology and Economics
Previously....

- **The kernel**
  - handles processes, manages resources
  - separates processes from each other (“virtual machine”)

- **Process**
  - a running user program
  - communicates with the kernel via system calls
  - has a family (parent and children)

- **Communication in theory**
  - shared memory region, messages, remote procedure calls (RPC)
  - data channel (resource) sharing, critical region, semaphore
  - blocking (synchronized) and non-blocking (asynchronous)
UNIX IPC examples

• Processing output of a program
  
  \texttt{ps -ef | more \ ps -ef | wc -l}  
  \texttt{ps -ef | cut -d \ -f 1 | sort | uniq | wc -l}
  
  – The “|” symbol represents a \textbf{pipe}: the output of the first command 
    (process) is redirected to the input of the second one.

• For independent processes
  
  \texttt{prw-rw-r-- 1 demo demo 8 márc 26 10:46 /tmp/named_pipe}
  
  – The “p” marks a special pipe: \textbf{named pipe} or \textbf{FIFO}

• Signals
  
  \texttt{CTRL + C \hfill CTRL + Z \hfill \textbf{kill} \hfill <SIG> \hfill <PID>}
  
  – Interrupt, suspend and continue
  – Notifies a process about an event (e.g. child process dies)

• Other simple examples: \textbf{Speaking UNIX: !$#@*%}
Examples (2)

- Spam and virus filtering in a UNIX system

```
srw-r--r-- 1 clamav clamav 0 Nov 27 11:38 /var/clamav/clmilter.socket
srwxr-xr-x 1 sa-milt sa-milt 0 Nov 27 11:38 /var/run/spamass-milter/spamass-milter.socket
```

- “s” denotes a **socket**, which is a “network” communication interface.

- Database systems use **shared memory** extensively
  - They have many processes which needs fast and simple communication
  - Typical usage:
    - static data
    - locking
    - data buffers
  - It should be properly configured during database installation
    It is not uncommon to assign half of the physical memory as shared
  - For more info, see e.g. Oracle System Global Area
    [http://docs.oracle.com/cd/B19306_01/server.102/b14220/memory.htm](http://docs.oracle.com/cd/B19306_01/server.102/b14220/memory.htm)
UNIX process communication – an overview

• Signal
  – event handling (raise and handle)

• Pipe
  – data flow FIFO, communication mainly in the family

• Semaphore

• Message queue
  – has a type and clear boundary

• Shared memory
  – several processes use the same physical memory region

• “network” (socket) communication

• Remote procedure call: the UNIX way
UNIX Signals

• Goals
  – notify a process about events raised by other processes or the kernel
  – synchronize processes (we have a better solution for this now)

• Signals have type (SIGINT, SIGCHLD, SIGKILL, … See: `kill -l`)
  – system: exceptions (errors), quota, alarm, notices (e.g. zombie child)
  – user: stop, kill, user defined, etc.

• How it works
  – generation: it is generated by a system call or some event)
  – delivery: the kernel notifies the target process about the signal
  – processing: the target process does something (or nothing) in the signal handler

• Problems
  – generation and delivery in (quite) different times
  – several different implementations, some of them are not too good
Generation and delivery

• Generation (by a process)
  ```c
  #include <signal.h>  /* kill() */
  kill(pid, SIGUSR1);  /* send the signal */
  ```

• Delivery and handling
  - Several signal handlers exists
    • Core: core dump and stop (`exit()`)  
    • Term: stop (`exit()`)  
    • Ign: ignore
    • Stop: suspend
    • Cont: return from suspended states
  - processes can define their own handlers
    ```c
    signal(SIGALRM, alarm);  /* set the handler */
    void alarm(int signum) { ... }  /* the handler */
    ```
  - it depends on the signal type which handler could be used
    • e.g. SIGKILL can not be ignored or handled by a process function
UNIX Signals: examples

```c
#include <signal.h>       /* signal(), kill() */
#include <unistd.h>                /* getpid() */
#include <sys/types.h>             /* pid_t */
pid_t pid = getpid();              /* own PID */

kill(pid, SIGSTOP);       /* send the STOP signal */
You can do the same in the command line: kill -STOP <PID>

signal(SIGCLD, SIG_IGN);  /* ignore child stops events */

signal(SIGINT, SIG_IGN);  /* ignore the CTRL+C */

signal(SIGINT, SIG_DFL);  /* set the default handler */

signal(SIGALRM, alarm);   /* set a special handler function */
void alarm(int signum) { ... } /* the handler... */
alarm(30);                 /* set and ALARM signal for 30 secs */
```
**man -s 7 signal** (excerpt)

<table>
<thead>
<tr>
<th>Signal</th>
<th>Value</th>
<th>Action</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>1</td>
<td>Term</td>
<td>Hangup detected on controlling terminal or death of controlling process</td>
</tr>
<tr>
<td>SIGINT</td>
<td>2</td>
<td>Term</td>
<td>Interrupt from keyboard</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>3</td>
<td>Core</td>
<td>Quit from keyboard</td>
</tr>
<tr>
<td>SIGILL</td>
<td>4</td>
<td>Core</td>
<td>Illegal Instruction</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>6</td>
<td>Core</td>
<td>Abort signal from abort(3)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>8</td>
<td>Core</td>
<td>Floating point exception</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>9</td>
<td>Term</td>
<td>Kill signal</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>11</td>
<td>Core</td>
<td>Invalid memory reference</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>13</td>
<td>Term</td>
<td>Broken pipe: write to pipe with no readers</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>14</td>
<td>Term</td>
<td>Timer signal from alarm(2)</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>15</td>
<td>Term</td>
<td>Termination signal</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>30,10,16</td>
<td>Term</td>
<td>User-defined signal 1</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>31,12,17</td>
<td>Term</td>
<td>User-defined signal 2</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>20,17,18</td>
<td>Ign</td>
<td>Child stopped or terminated</td>
</tr>
<tr>
<td>SIGCONT</td>
<td>19,18,25</td>
<td>Cont</td>
<td>Continue if stopped</td>
</tr>
<tr>
<td>SIGSTOP</td>
<td>17,19,23</td>
<td>Stop</td>
<td>Stop process</td>
</tr>
<tr>
<td>SIGTSTOP</td>
<td>18,20,24</td>
<td>Stop</td>
<td>Stop typed at tty</td>
</tr>
<tr>
<td>SIGTIN</td>
<td>21,21,26</td>
<td>Stop</td>
<td>tty input for background process</td>
</tr>
<tr>
<td>SIGTTOU</td>
<td>22,22,27</td>
<td>Stop</td>
<td>tty output for background process</td>
</tr>
</tbody>
</table>
UNIX Pipes: pipe()

- Goal: transfer data between processes (ls -la | more)

- Features
  - works within the family (parent – children, or between children)
  - it is a data flow (no message boundary)
  - it is a directed data flow (writer → reader) (there could be several readers and writers on the same channel!)
  - it has a limited capacity: e.g.. 4k (Linux < 2.6.11), 65k (Linux >= 2.6.11)

- How does it work?
  - a process creates a pipe (pipe())
  - the kernel creates the pipe data structures and returns file descriptors
  - the process creates child processes, they inherit these descriptors
  - these processes communicate via the descriptors (read(), write())

- Limitations
  - no addressing, no boundary, works in the “family”
UNIX Named Pipe

• Goals / problems to solve
  – How to communicate between independent processes?
  – How to access a pipe that was created by an independent process? (naming or identification problem)

• Features
  – it is the same as a normal Pipe
  – but it works outside the family
  – the file system helps in the identification of the pipe

• Example: communication with the *init* process (PID 1)
  – we can see its pipe interface in the file system:

```bash
prw------- 1 root root 0 Jan 1 12:38 /dev/initctl
```

(we can open this “file” and then we can send commands in the pipe)
UNIX System V IPC

• Goal: unified communication interface between processes
  – data transfer
  – synchronization

• Common foundation (and notation)
  – resources: means of communication (see below)
  – key: resource identifier (a number)
  – functions for control and access: *ctl(), *get( ... key ...)
  – access management:
    • creator, owner and their groups
    • the usual UNIX access management system works here too

• Resources
  – semaphores
  – message queues see these man pages: man svipc ipc ipcs
  – shared memory
UNIX System V IPC: semaphores

- Goal: synchronization between processes
  - P() and V() operators
  - to handle multiple semaphores at once

- How does it work?

  \[
  \text{sem\_id = semget(key, num, options);} \\
  \text{access a number of semaphores identified by the key} \\
  \text{(they will be created if needed)}
  \]

  \[
  \text{perform the operations defined in the ops structure (see man semop):} \\
  \text{status = semop(sem\_id, ops, ops\_méret);} \\
  \]

- multiple operations on multiple semaphores
- blocking and non-blocking P()
- there is also a simple transaction managements (undo)
UNIX System V IPC: message queues

- **Goal**: data exchange between processes
  - messages with clear boundary
  - a message type helps in filtering

- **How does it work?**

  ```c
  msgq_id = msgget(key, options);
  - access a message queue identified by the key
    (it will be created if needed)

  - send messages (the msg contains a type identifier):
    ```c
    msgsnd(msgq_id, msg, size, options);
    ```

  - receive: ```c
    msgrcv(msgq_id, msg, size, type, options);
    ```

  - the **type** (number) can be used to filter out messages
    ```c
    = 0 any message
    > 0 a message with the specified type
    < 0 a message with the same or “lower” type (“importance”)
  ```
UNIX System V IPC: shared memory

• **Goal:** simple and fast data exchange between processes
  - a special area of the system memory reserved for this purpose
  - there is no kernel overhead on data transfer (shared physical memory)

• **How does it work?**

  \[
  \text{shm\_id} = \text{shmget(key, size, options)};
  \]

  - access a shared memory region identified by the key
    (it will be created if needed)

  - bind this region to a virtual address:
    \[
    \text{var} = (\text{type}) \text{shmat}(\ldots);
    \]

    We can access the memory via the variable.

  - **Unbind:** \[
    \text{shmdt} (\text{var});
    \]

  - **Note:** mutual exclusion have to be guaranteed using e.g. semaphores
UNIX “network” (socket) communication

• Goal: data transfer that supports addressing and several protocols
  – between any processes, even on different computers
  – supports many protocols (e.g. the TCP/IP family)
  – provides several addressing methods

• Basic notation
  – socket: the communication endpoint (an identifier)
  – address and port number (see computer networks)

• Usage
  sfd = socket(domain, type, protocol);
  server: bind(sfd, address, ...);
  client: connect(sfd, address, ...);
  server: listen(sfd, queue_size);
  server: accept(sfd, address, ...);
  send(sfd, message, ...);
  recv(sfd, message, ...);
  shutdown(sfd);
How does it work in client-server architecture?

Client program

```
socket()
connect()
send()
recv()
close()
```

Server program

```
sfd1 = socket()
bind(sfd1)
listen(sfd1)
while
  sfd2 = accept(sfd1)
  fork()
  parent: go back to the cycle
  child: recv(sfd2)
         send(sfd2)
         close(sfd2)
         exit()
```
(Sun) RPC (remote procedure call)

- A *distributed system* architecture based on socket communication
- Goals:
  - high level communication between processes
  - calling functions in different processes (even on a different machine)
  - help the programmers: interface specification + code generation

- Basic notation
  - RPC language: a method to describe callable interfaces
  - identifier: program and function identifiers in the interface description
  - portmapper: mapping between network ports and identifiers
  - rpcgen: generates C code from the interface description

- Sun RPC has
  - a method to describe interfaces
  - a program code generator to create client and server code from interface descriptions
  - a communication infrastructure to do the underlying work
RPC interface description and code generation

• RPC language (example: date.x)

    program DATE_PROG {
        version DATE_VERS {
            long BIN_DATE(void) = 1;    /* function identifier = 1 */
            string STR_DATE(long) = 2;  /* function identifier = 2 */
        } = 1;                          /* version = 1 */
    } = 0x31237;                       /* program identifier = 0x31237 */

• Code generation using `rpcgen`
  - `rpcgen date.x` will create several files
    • `date.h`: definitions of data types
    • `date_clnt.c`: client stub that contains the functions called by client programs
    • `date_srv.c`: server skeleton that contains functions to be implemented
    • (...)
How to choose the right one (for a purpose)?

• (Implementation bindings: programming language, environment, etc.)
• Communication endpoints
  – on a single computer: all methods, RPC might not work (portmapper)
  – networked: socket, RPC, distributed filesystems (see next UNIX lecture)
• The nature of the communication
  – notifying about events: signals (SIGUSR1)
  – synchronization: semaphores (signals)
  – data stream (pipe, socket) vs. data messages (message queue)
  – message types, filtering (message queue)
  – amount of data (shared memory: small, pipe, socket)
• Performance
  – speed: shared mempry, pipe, socket (PF_UNIX)
  – resource consumption: all but shared memory
• Convenience
  – RPC, shared memory (but semaphores often needed)
• Programming examples: [http://beej.us/guide/bgipc/](http://beej.us/guide/bgipc/)
Summary: UNIX inter-process communication

• Classical forms of communication:
  – Signals, pipes and named pipes (FIFO)

• System V IPC: a unified communication framework
  – Semaphores
  – Message queues
  – Shared memory

• (not just) Network communication methods
  – socket communication
  – Sun RPC – remote procedure call, the UNIX way

• Standards: IEEE Posix, System V, BSD

• There are other possibilities
  – more speed: e.g. MegaPipe
  – higher abstraction level: CORBA, DCOM, SOAP, REST, etc.
Roll your own... Web server (homework)

- Pick a virtual machine (e.g. CentOS 7)
- Implement the HTTP GET protocol
  
  \[\text{GET } /\text{directory/file.html}\]

- Your server should handle multiple clients at the same time.
  
  \text{fork()}

- Test your server with a Web browser
- Stress test your server
  
  \text{ab -n 100 \textit{-c} 10 http://localhost/}
  
  100 requests, 10 at a time
- Compare your results to a real server

Program skeleton

\begin{verbatim}
  sfd1 = socket()
  bind(sfd1)
  listen(sfd1)
  while
    sfd2 = accept(sfd1)
    fork()
    parent:
      back to accept
    child:
      recv(sfd2)
      analyse input
      load the req'd file
      send(sfd2)
      close(sfd2)
      exit()
\end{verbatim}