Inter-Process Communication

2501ICT Nathan

René Hexel

School of Information and Communication Technology
Griffith University

Semester 1, 2011

Outline

1. Inter-Process Communication Introduction
   - Shared Memory
   - Messages
   - Signals
2. Advanced IPC
   - Pipes
   - Sockets
   - Networked Sockets

Inter-Process Communication

- Shared Memory
  - Shared Address Space
  - Implicit (Threads) or Explicit (Processes)
- Semaphores
  - Simple Integer (mainly for synchronisation)
- Pipes, Sockets
  - FIFO queue between Tasks
- Signals
  - Asynchronous Events

Shared Memory

- Fastest Form of IPC
  - Threads: simple shared objects or variables
  - Processes: explicitly shared memory (SysV) or memory mapped files (BSD)
- Explicitly Shared Memory
  - `shmget()`
    - get a new shared memory area identifier
  - `shmat()` / `shmdt()`
    - attach/detach shared memory to local address space
  - `shmctl()`
    - configure shared memory (e.g. permissions)
    - gather statistics

Shared Memory Writer Example

Example (Write `Hello, world` to shm 12345)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>

int main(int argc, char *argv[])
{
        size_t size = 1024;
        int shmid = shmget(12345, size, 0644 | IPC_CREAT);

        if (shmid == -1) { perror("shmget failed"); return EXIT_FAILURE; }

        char *data = shmat(shmid, NULL, 0);      /* attach shared memory */

        if (data == (char *) -1) { perror("shmat"); return EXIT_FAILURE; }

        sprintf(data, "Hello, %s", "world");     /* write important data */

        shmdt(data);                              /* done, detach shared memory */

        return EXIT_SUCCESS;
}
```

Shared Memory Reader Example

Example (Reads, prints, and deletes shm 12345)

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/shm.h>

int main(int argc, char *argv[])
{
        size_t size = 1024;
        int shmid = shmget(12345, size, 0644);

        if (shmid == -1) { perror("shmget failed"); return EXIT_FAILURE; }

        char *data = shmat(shmid, NULL, SHM_RDONLY);    /* attach read-only */

        if (data == (char *) -1) { perror("shmat"); return EXIT_FAILURE; }

        printf("data is: %s!\n", data);                 /* read data */

        shmdt(data);                                    /* detach */

        shmctl(shmid, IPC_RMID, NULL);                  /* deallocate memory */

        return EXIT_SUCCESS;
}
```

Memory-Mapped Files

- Similar to Shared Memory
  - BSD origins
  - More flexible: uses file names instead of `int` IDs
  - Allows writing data to disk as well
- Memory Mapping Functions
  - `mmap()`
    - map an already-opened file into memory
  - `munmap()`
    - unmap file from memory
  - `msync()`
    - synchronise mapped region with disk
  - `unlink()`
    - delete file from disk

Memory Mapped Writer Example

Example (Write `Hello, world` to `/tmp/test`)

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

int main(int argc, char *argv[])
{
        size_t size = 8192;
        int file = open("/tmp/test", O_RDWR | O_CREAT| O_TRUNC, 0644);

        if (file == -1) { perror("open"); return EXIT_FAILURE; }

        if (ftruncate(file, size) == -1) return EXIT_FAILURE;

        char *data = mmap(NULL, size, PROT_WRITE, MAP_FILE | MAP_SHARED,
                                 file, 0);        /* memory-map file */

        if (data == MAP_FAILED) { perror("mmap"); return EXIT_FAILURE; }

        sprintf(data, "Hello, %s", "world");     /* write important data */

        munmap(data, size);                      /* done, unmap file */
        close(file);                             /* close the file */

        return EXIT_SUCCESS;
}
```

Memory Mapped Reader Example

Example (Reads, prints, and deletes /tmp/test)

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

int main(int argc, char *argv[])
{
        size_t size = 8192;
        int file = open("/tmp/test", O_RDONLY);

        if (file == -1) { perror("open"); return EXIT_FAILURE; }

        char *data = mmap(NULL, size, PROT_READ, MAP_FILE | MAP_SHARED,
                          file, 0);        /* memory-map file */

        if (data == MAP_FAILED) { perror("mmap"); return EXIT_FAILURE; }

        printf("data is: %s!\n", data);          /* print shared data */

        munmap(data, size);                      /* done, unmap file */
        close(file);                             /* close the file */
        unlink("/tmp/test");                     /* delete the file */

        return EXIT_SUCCESS;
}
```

Messages

- Block of data with accompanying type
- Process Message queue (mailbox)
- Message Queue functions
  - `msgget()`
    - get a new message queue
  - `msgsnd()` / `msgrcv()`
    - transmit / receive a message
  - `msgctl()`
    - configure mailbox (e.g. permissions)
    - gather statistics

Message Sender Example

Example (Write `Hello message` to mailbox 12345)

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msgbuf { long mtype; char mtext[80]; };  /* message type and data */

int main(int argc, char *argv[])
{
        int msgq = msgget(12345, 0644 | IPC_CREAT);     /* get queue 12345 */

        if (msgq == -1) { perror("msgget failed"); return EXIT_FAILURE; }

        struct msgbuf data;                             /* date to write */

        sprintf(data.mtext, "Hello message\n"); /* create message */
        data.mtype = 1234;                      /* message type ( > 0 ) */

        msgsnd(msgq, &data, sizeof(data), 0);   /* send message */

        return EXIT_SUCCESS;
}
```

Message Receiver Example

Example (Reads, prints, and deletes mailbox 12345)

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

struct msgbuf { long mtype; char mtext[80]; };  /* message type and data */

int main(int argc, char *argv[])
{
        int msgq = msgget(12345, 0644);         /* get mailbox 12345 */

        if (msgq == -1) { perror("msgget failed"); return EXIT_FAILURE; }

        struct msgbuf data;                     /* data to read */

        /* receive message: IPC_NOWAIT means don't wait if no message */
        if (msgrcv(msgq, &data, sizeof(data), 1234, IPC_NOWAIT) < 0)
                perror("no message of type 1234 for me");
        else
                printf("Message type %ld: %s", data.mtype, data.mtext);

        msgctl(msgq, IPC_RMID, NULL);           /* remove mailbox */

        return EXIT_SUCCESS;
}
```

Signals

- Inform aprocess about an event
  - Similar to a hardware interrupt
- Signals can be sent . . .
  - . . . by the Kernel
  - . . . by a Process
    - `kill()`
- Signals are *Sets*

  ⇒ they are either present or not present

  ⇒ their number is not counted

  ⇒ they cannot be queued (order is not guaranteed)

Signal Sender Example

Example (Send given signals to given processes)

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

int main(int argc, char *argv[])
{
    do
    {
        int sig   = argc > 1 ? atoi(argv[1]) : SIGKILL; /* get signal */
        pid_t pid = argc > 2 ? atoi(argv[2]) : getpid();/* get process */

        if (kill(pid, sig) == -1)
        {
                perror("kill did not work");            /* can't send signal */
                return EXIT_FAILURE;
        }
        argv += 2;
        argc -= 2;
    }
    while (argc > 1);

    return EXIT_SUCCESS;
}
```

Intercepting Signals

- Default Action
  - *Ignore* signal or *terminate* the program
  - Depends on signal type (`SIGTERM`, `SIGKILL`, `SIGHUP`, `SIGCHLD`, `SIGUSR1`, . . .)
- Behaviour can be modified
  - A function gets exectuted at signal arrival
  - `signal()`
    - sets the action or function that should be performed

Signal Interceptor Example

Example (Prints message, waits for 1 second, exits with error)

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

void handler(int sig)
{
        printf("I don't want to exit!\n");
        sleep(1);
}

int main(int argc, char *argv[])
{
        signal(SIGINT, handler);        /* install signal handler */

        if (sleep(3600) > 0)             /* wait for one hour */
        {
                perror("sleep");
                return EXIT_FAILURE;
        }

        return EXIT_SUCCESS;
}
```

Pipes

- Circular FIFO Buffer
  - acts like a file
  - unidirectional (one reading and one writing end)
  - Producer/Consumer model between two tasks
- Unnamed pipes
  - `pipe()` function, `NSPipe` class
  - Works only between related processes
- Named Pipes
  - have a name on the file system (like memory mapped files)
  - but data are never actually written to the file system

    $\rightarrow$ memory only
  - `mkfifo()` function, together with `open()`, `close()`, ...

Unnamed Pipe Example

Example (Child process prints `hello child`)

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

int main(void) {
        char line[12];                                          // string buffer
        int n, fd[2];                                           // pipe descriptors
        pid_t pid;

        if (pipe(fd) < 0) { perror("pipe"); return EXIT_FAILURE; }

        if ((pid = fork()) == -1) { perror("fork"); return EXIT_FAILURE; }
        if (pid != 0) {                                         // parent
                close(fd[0]);                                   // close reading end
                write(fd[1], "hello child", sizeof(line));
                wait(&n);                                       // wait for child
        } else {                                                // child
                close(fd[1]);                                   // close writing end
                n = read(fd[0], line, sizeof(line));            // block until written
                printf("%s\n", line);
        }
        return EXIT_SUCCESS;
}
```

FIFO Writer Example

Example (Writes `Hello fifo` to fifo)

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

const char data[] = "Hello fifo";
const char fifo[] = "/tmp/fifo";

int main(int argc, char *argv[])
{
        if (mkfifo(fifo, 0644) == -1) { perror(fifo); return EXIT_FAILURE; }

        int file = open(fifo, O_WRONLY);
        if (file == -1) { perror("open"); return EXIT_FAILURE; }

        if (write(file, data, sizeof(data)) != sizeof(data))
        {
                perror("error writing to FIFO");
                return EXIT_FAILURE;
        }
        close(file);                            /* close the fifo */

        return EXIT_SUCCESS;
}
```

FIFO Reader Example

Example (Read and print fifo data)

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

int main(int argc, char *argv[])
{
        const char fifo[] = "/tmp/fifo";
        int file = open(fifo, O_RDONLY);
        if (file == -1) { perror("open"); return EXIT_FAILURE; }

        char data[80];
        if (read(file, data, sizeof(data)) <= 0)
        {
                perror("error reading from FIFO");
                return EXIT_FAILURE;
        }

        printf("data: %s\n", data);

        close(file);                                /* close the fifo */
        unlink(fifo):                               /* remove the fifo */

        return EXIT_SUCCESS;
}
```

Sockets

- Similar to Pipes
  - but allow finer control over protocol, timeouts, etc.
  - can be used across machines (over networks)
- `socketpair()`
  - like `pipe()`, but creates an unnamed pair of sockets
  - allows specifying the protocol
  - bi-directional

Socket Pair Example

Example (hello parent)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>

int main(void) {
        char line[20];
        int n, fd[2];                           // pipe descriptors
        pid_t pid;
        if (socketpair(AF_UNIX, SOCK_STREAM, 0, fd) < 0) return EXIT_FAILURE;

        if ((pid = fork()) == -1) { perror("fork"); return EXIT_FAILURE; }
        if (pid != 0) {                                 // parent
                close(fd[0]);                           // close child end
                read(fd[1],  line, sizeof(line));       // read from child
                strcat(line, " parent");                // add to original string
                write(fd[1], line, strlen(line) + 1);
                wait(&n);                               // wait for child
        } else {                                        // child
                close(fd[1]);                           // close parent end
                write(fd[0], "hello", 6);               // write to parent
                read(fd[0], line, sizeof(line));        // wait for response
                printf("%s\n", line);
        }
        return EXIT_SUCCESS;
}
```

Networked Sockets

- Allow data to be sent across the Internet
  - unreliable
    - network segments might be down
    - routing problems
    - bandwidth bottlenecks
    - delays and jitter

  $\Rightarrow$ error checking becomes paramount
- Clients and Servers
  - model for handling network connections
  - isolate network specific tasks from other tasks

Clients and Servers

- Applications that can run across a network connection or locally
- Server
  - Process on a (networked) computer that ...
    - ...accepts requests from other, local or remote, programs
    - ...performs services according to such requests
- Client
  - process that issues such requests
  - typically waits for results from server

Protocols

- The “language” clients and servers use to talk to each other
- HTTP
    - HyperText Transfer Protocol
        - Client: Web Browser
        - Server: Web Server
- FTP
    - File Transfer Protocol
- SMTP
    - Simple Mail Transfer Protocol

Internet

- Heterogenous Network, connecting . . .
  - . . . Local Area Networks (LANs), and
  - . . . Wide Area Networks (WANs)
- Standardised set of protocols
  - ⇒ allows heterogenous devices to talk to each other
  - Internet Protocol (IP)
    - base protocol that all Nodes must speak
    - IPv4: current version – IPv6: upcoming standard
  - Internet Control Message Protocol (ICMP)
    - controls resource availability and policy

Transport Protocols

- sit on top of IP
- TCP
  - Transmission Control Protocol
  - reliable streaming protocol

    → order and integrity of data are guaranteed!
  - connection-oriented protocol
- UDP
  - User Datagram Protocol
  - packet-oriented, unreliable protocol
  - connectionless protocol
- Application-Level Protocols (FTP, HTTP, SMTP, etc.)
  - sit on top of TCP (or UDP, depending on the protocol)

  ⇒ layered structure
    - each layer only needs to deal with its own concerns

    ⇒ simplifies program structure!

IP Addressing

- How to connect from one computer to another?

→ every network node (computer) has an *IP Address*
  - universal on the Internet: unique identifier
- IPv4
  - *4 bytes* in dotted decimal notation *a.b.c.d*, e.g.:

  → `127.0.0.1`

  → `132.234.34.2`
  - Problem: *shortage* of IP Addresses
- IPv6
  - *16 bytes* in hexadecimal notation, e.g.:

    → `fe80:0000:0000:0000:02e0:98ff:fe85:6ec5`

    → shorter: `fe80::2e0:98ff:fe85:6ec5`

Special IPv4 Addresses

- Local Computer: `127.x.y.z`
  → `127.0.0.1` is always the `localhost`
- Private Networks
  - not visible from the Internet (but can be used privately)
  → `192.168.x.y`
  → `172.16/12`
  → `10.a.b.c`
- Multicast Addresses
  - address a large number of listeners with only one address
  → `224.0.0.0 - 239.255.255.255`

Name Resolution

- IP Addresses
  - generally tied to (physical) network locations
  - hard to remember

→ Assign *Symbolic Names* to IP addresses

  → e.g. `localhost` instead of `127.0.0.1`

- Locally
  - `hosts` file
- Globally
  - Domain Name resolution System (DNS)

Domain Name Resolution

- DNS is a global, distributed service
  - numerous servers worldwide
  - → each responsible for a subset of names
- Hierarchical naming system
  - one *host* name
  - a list of *domain* names
  - → `dwarf.cit.griffith.edu.au`
    - host name: `dwarf`
    - domain: `cit.griffith.edu.au`
    - top-level domain: `.au`

Ports

- IP address only represents a node (computer) on the network
  - ⇒ not enough to identify multiple individual services on one machine!
- Port
  - specific 16 bit address that identifies a service
  - must be agreed on by client and server
- Services
  - operate on well-known ports, e.g.
  - → port 80 – HTTP
  - → port 22 – ssh
  - → port 25 – smtp

Socket API

- Originally developed for BSD Unix
  - now a standard on almost all of today's operating systems
- Two fundamental message communication operations
  - send
  - receive

Programming UDP Sockets

- no connection required
- Sender
  - set up a socket, then send packets to destination
  - unreliable: no guarantee that packets will actually arrive!
- Receiver
  - set up a socket, then wait for packets
- Server
  - loop to receive packets
  - send responses
- Client
  - send request to server
  - wait for response

    → timeouts possible!

UDP Receiver Example

Example (Receive and print UDP message)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>

int main(void)
{
        struct sockaddr_in local;                          // local address
        char msg[50];
        int sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);

        if (sock == -1) { perror("socket"); return EXIT_FAILURE; }

        memset(&local, 0, sizeof local);                   // clear local
        local.sin_family        = AF_INET;                 // IPv4 address
        local.sin_port          = htons(1234);             // listen on port 1234
        local.sin_addr.s_addr   = INADDR_ANY;              // whichever IP we have
        if (bind(sock, (void *) &local, sizeof local) != 0)  return EXIT_FAILURE;

        if (recvfrom(sock, msg, sizeof msg, 0, NULL, 0) < 0) return EXIT_FAILURE;

        printf("got: '%s'\n", msg);                        // assume it's a string
        close(sock);
        return EXIT_SUCCESS;
}
```

UDP Sender Example

Example (Send a UDP message)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(void)
{
        char *msg = "hello UDP!";                       // message to send
        struct sockaddr_in remote;                      // remote address

        int sock = socket(PF_INET, SOCK_DGRAM, IPPROTO_UDP);
        if (sock == -1) { perror("socket"); return EXIT_FAILURE; }

        memset(&remote, 0, sizeof remote);              // clear remote address
        remote.sin_family      = AF_INET;               // IPv4 address
        remote.sin_port        = htons(1234);           // send to port 1234
        remote.sin_addr.s_addr = inet_addr("127.0.0.1");// "remote" IP address

        if (sendto(sock, msg, strlen(msg) + 1, 0, (void *) &remote, sizeof remote)
            != strlen(msg) + 1) { perror("send"); return EXIT_FAILURE; }

        close(sock);
        return EXIT_SUCCESS;
}
```

Programming TCP Sockets

- connection between client and server needs to be established first!
- Client
    - connect to server first
    - exchange data
- Server
    - set up a socket, then listen for connections
    - accept connection
    - exchange data over accepted connection
    - handling of multiple exceptions requires ...
        - → ... multiple tasks
        - → ... use of `poll()` or `select()` to wait for data on multiple connections

TCP Client Example

Example (Simple TCP Client)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
int main(void) {
        char msg[80] = "Hello void!\n";
        struct sockaddr_in remote;                            // remote address

        int n, sock = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (sock == -1) { perror("socket"); return EXIT_FAILURE; }

        memset(&remote, 0, sizeof remote);                    // clear remote
        remote.sin_family      = AF_INET;                     // IPv4 address
        remote.sin_port        = htons(4242);                 // send to port 4242
        remote.sin_addr.s_addr = inet_addr("127.0.0.1");// "remote" IP address
        if (connect(sock, (void *)&remote, sizeof remote) < 0) return EXIT_FAILURE;

        if (write(sock, msg, strlen(msg)) != strlen(msg))
                { perror("write"); return EXIT_FAILURE; }
        while ((n = read(sock, msg, sizeof(msg))) > 0)
                write(STDOUT_FILENO, msg, n);
        close(sock);
        return EXIT_SUCCESS;
}
```

TCP Server Example

Example (Simple TCP Server)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
int main(void)
{
        struct sockaddr_in local;                          // local address
        char msg[50];
        int n, c, s = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
        if (s == -1) { perror("socket"); return EXIT_FAILURE; }

        memset(&local, 0, sizeof local);                   // clear local
        local.sin_family        = AF_INET;                 // IPv4 address
        local.sin_port          = htons(4242);             // listen on port 4242
        local.sin_addr.s_addr   = INADDR_ANY;              // whichever IP we have
        if (bind(s, (void *) &local, sizeof local) != 0) return EXIT_FAILURE;
        if (listen(s, 4) == -1) return EXIT_FAILURE;       // listen for connections
        while ((c = accept(s, NULL, 0))) {                 // accept a connection
                if ((n = read(c, msg, sizeof(msg))) > 0)
                        write(c, msg, n);                  // just echo back msg
                close(c);
        }
        close(s);
        return EXIT_SUCCESS;
}
```

File/Stream TCP API

- TCP sockets are treated in the same way as local files (streams)
  - → allows use of higher level, buffered I/O functions
- C
  - `fdopen()`
    - create `FILE *` from socket
    - allows use of `stdio` functions such as `fprintf()`, `fgets()`, etc.
- Objective-C
  - `NSFileHandler *fh = [[NSFileHandle alloc] initWithFileDescriptor: mySocket];`
  - → see example code
- C++
  - no standard, cross-platform socket API
  - → Boost: `ip::udp::socket`, `ip::tcp::socket`, `ip::tcp::acceptor`, ...