Inter Process Communication (IPC)

Giorgio Richelli
giorgio_richelli@it.ibm.com

Contents

Giorgio Richelli
giorgio_richelli@it.ibm.com

Introduction

The purposes of IPC:

- Data transfer
- Sharing data
- Event notification
- Resource sharing
- Process control

Giorgio Richelli
giorgio_richelli@it.ibm.com

Signal Generation & Handling

Signal:

A way to call a procedure when some events occur.

Generation:

when the event occurs.

Delivery:

when the process recognizes the signal’s arrival (handling)

Signal Generation & Handling

Pending: between generated and delivered.

System V: 15 signals

4BSD/SVR4 : 31 signals

Signal numbers: different in different system or versions

Giorgio Richelli
giorgio_richelli@it.ibm.com

Signal Handling

Default actions: each signal has one.

```
Abort: Terminate the process after generating a core dump.
Exit: Terminate the process without generating a core dump.
Ignore: Ignores the signal.
Stop: Suspend the process.
Continue:  Resume the process, if suspended
```

Default actions may be overridden by signal handlers

Giorgio Richelli
giorgio_richelli@it.ibm.com

Signal Handling

`issig()` (Kernel call) : check for signals

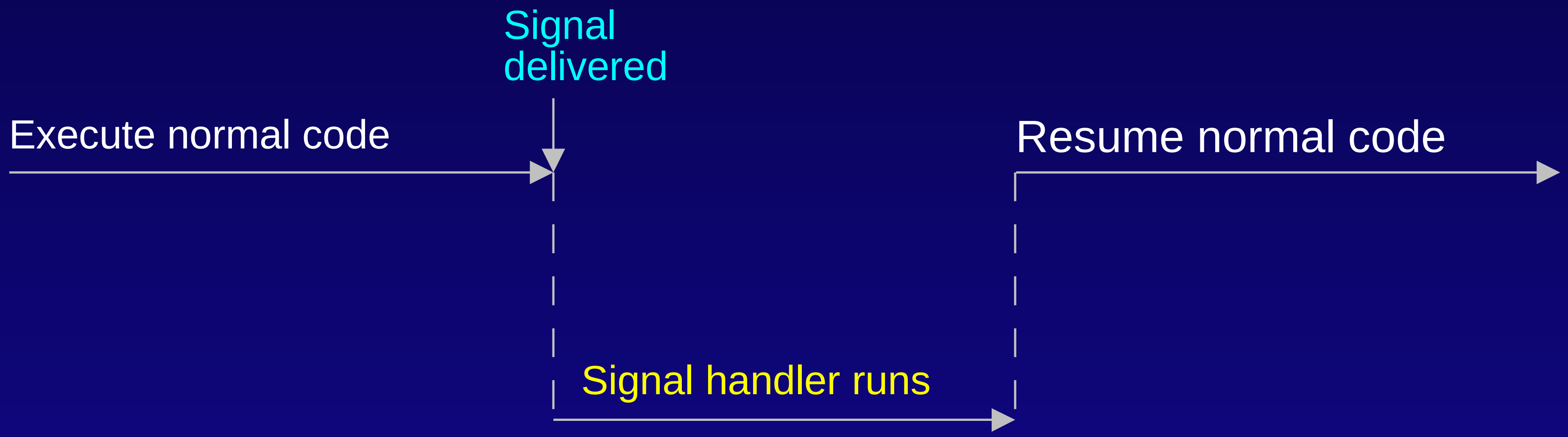
- Before returning to user mode from a system call or interrupt.
- Just before blocking on an interruptible event
- Immediately after waking up from an interruptible event

`psig()`: dispatch the signal

`sendsig()`: invoke the user-defined handler

Giorgio Richelli
giorgio_richelli@it.ibm.com

Signal Handling



Giorgio Richelli
giorgio_richelli@it.ibm.com

Signal Generation

Signal sources:

- Exceptions
- Other processes
- Terminal interrupts
- Job control
- Quotas
- Notifications
- Alarms

Giorgio Richelli
giorgio_richelli@it.ibm.com

Typical Scenarios

^C (Ctrl-c)

Exceptions:

- Trap
- issig(): when return to user mode.

Pending signals

- processed one by one.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Sleep and signals

Interruptible sleep:

- waiting for an event with indefinite time.

Uninterruptible sleep:

- is waiting for a short term event such as disk I/O
  - Pending the signal
  - Recognizing it until returning to user mode or blocking on an event

```
if (issig()) psig();
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Unreliable Signals

- Signal handlers are not persistent and do not mask recurring instances of the same signal (SVR2)
- Race conditions: two ^C.
- Performance: SIG_DFL, SIG_IGN:
  - Kernel does not know the content of `u_signal[];`
  - Awake, check, and perhaps go back to sleep again (waste of time).

Giorgio Richelli
giorgio_richelli@it.ibm.com

Reinstalling a signal handler

```
void sigint_handler(int sig)
{
    signal(SIGINT, sigint_handler);
  …
}
main()
{
    signal(SIGINT, sigint_handler);
    …
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Unreliable Signals

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

int cnt=0;
void handler(int sig)
{
    cnt++;
    printf("In the handler...\n");
    signal(SIGINT,handler);
}
main()
{
    signal(SIGINT,handler);
    while (1) {
         printf("In main\n");
         sleep(1);
    }
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Reliable Signals

Primary features:

- Persistent handlers: need not to be reinstalled.
- Masking: A signal can be temporarily masked (will be delivered later)
- Sleeping processes: let the signal disposition info visible to the kernel (kept in the *proc*)
- Unblock and wait: sigpause()-automatically unmasks a signal and blocks the process.

Giorgio Richelli
giorgio_richelli@it.ibm.com

The SVR3 implementation

```
int sig_received = 0;
void handler (int sig)
{
    sig_received++;
}
main()
{
    sigset (SIGQUIT, handler);
    /* sighold(SIGQUIT);  */
    while (sig_received ==0) sigpause(SIGINT);
    ....
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Signals in SVR4

- `sigprocmask(how, setp, osetp)`
    - `SIG_BLOCK, SIG_UNBLOCK, SIG_SETMASK`
- `sigaltstack(stack, old_stack):`
    - Specify a new stack to handle the signal
- `sigsuspend(sigmask)`
    - Set the blocked signals mask to *sigmask* and puts the process to sleep
- `sigpending(setp)`
    - *setp* contains the set of signals pending to the process

Giorgio R
giorgio_richelli@it.ibr

Signals in SVR4

- `sigsendset(procset, sig)`
  - Sends the signal sig to the set of processes procset
- `sigaction(signo, act, oact)`
  - Specify a handler for signal signo.
  - act, oact pointers to sigaction structure
  - oact is the previous sigaction data
- Compatibility interface:
  - `signal, sigset, sighold, sigrelse, sigignore, sigpause`

Signal flags

- SA_NOCLDSTOP: Do not generate SIGCHLD when a child is suspended
- SA_RESTART: Restart system call automatically if interrupted by this signal
- SA_ONSTACK: Handle this signal on the alternate stack, if one has been specified by sigaltstack
- SA_NOCLDWAIT: sleep until all terminate
- SA_SIGINFO: additional info to the handler.
- SA_NODEFER: do not block this signal
- SA_RESETHAND: reset the action to default

Universal IPC Facilities

- Signals
  - Kill
  - Sigpause
  - ^C
- Expensive
- Limited: only 31 signals.
- Signals are not enough.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Pipes

A unidirectional, FIFO, unstructured data stream of fixed maximum size.

```
int pipe (int * filedes)
```

Data flow through a pipe.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Pipes

- Write to `filedes[1]`
- Read from `filedes[0]`
- Write to a pipe could block for large I/O sizes

Giorgio Richelli
giorgio_richelli@it.ibm.com

Named Pipes

Aka 'FIFO's

Identified by their access point (filename)

```
int mkfifo(char *path, mode_t mode);
```

Can be opened/read/written as normal files

Giorgio Richelli
giorgio_richelli@it.ibm.com

Named Pipes

A named pipe cannot be opened for both reading and writing.

Read and write operations to a named pipe are blocking,by default.

Seek operations (lseek) cannot be performed on named pipes

Giorgio Richelli
giorgio_richelli@it.ibm.com

System V IPC

Common Elements

- Key: resource ID
- Creator: Ids
- Owner: Ids
- Permissions: r/w/x for owner/group/others

Giorgio Richelli
giorgio_richelli@it.ibm.com

Semaphores

Special variable called a semaphore is used for “signaling”

If a process is waiting for a “signal”, it is suspended until that “signal” is sent

“Wait” and “signal” operations cannot be interrupted (e.g. they are atomic)

Queue is used to hold processes waiting on the semaphore

Giorgio Richelli
giorgio_richelli@it.ibm.com

P/V Operations

```
P(wait):
    s=s-1;
    if (s<0) block();

V(signal):
    s= s+1;
    If (s>=0) wake();
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Producer/Consumer Problem

One or more producers are generating data and placing these in a buffer

- A single consumer is taking items out of the buffer one at time
- Only one producer or consumer may access the buffer at any one time
- Three semaphores are used:
  - Amount of items in the buffer
  - Number of free entries in the buffer
  - Right to use the buffer

Producer Function - Pseudocode

```
#define SIZE 100
semaphore s=1
semaphore n=0
semaphore e= SIZE
void producer(void)
{
   while (TRUE){
      produce_item();
      wait(e);
      wait(s);
      enter_item();
      signal(s);
      signal(n);
   }
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Consumer Function

```
void consumer(void)
{
    while (TRUE){
        wait(n);
        wait(s);
        remove_item();
        signal(s);
        signal(e);
    }
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Semaphore

int semget(key_t key, int count, int flag);

- Returns the id. of semaphore set (*count* elements) associated with *key*.
- *key* :
  - IPC_PRIVATE
- *flag* :
  - IPC_CREAT, ...
  - Access permissions

Giorgio Richelli
giorgio_richelli@it.ibm.com

Semaphore

int semop(int semid, struct sembuf *sops, unsigned nsops);

performs operations on selected members of the semaphore set indicated by *semid*. Each of the *nsops* elements in the array pointed to by *sops* specifies an operation to be performed on a semaphore by a

Operations are performed atomically and only if they can all be simultaneously performed

Giorgio Richelli
giorgio_richelli@it.ibm.com

Semaphore

```
struct sembuf {
        unsigned short sem_num;
        short sem_op;
        short sem_flg;
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Semaphore

- unsigned short sem_num
  - semaphore number (in set *semid*)
- short sem_flg
  - IPC_NOWAIT
    - Don't block, but returns *-1* and set *errno* to *EAGAIN*
  - IPC_UNDO
    - undo operation(s) when process exits

Giorgio Richelli
giorgio_richelli@it.ibm.com

Semaphore

short sem_op

- when >0
  - Add sem_op to the value; eventually wake up suspended processes
- when == 0
  - Block until value == 0 (unless IPC_NOWAIT)
- when <0
  - Block (unless IPC_NOWAIT) until the value becomes greater than or equal to the absolute value of sem_op, then subtract sem_op from that value

Giorgio Richelli
giorgio_richelli@it.ibm.com

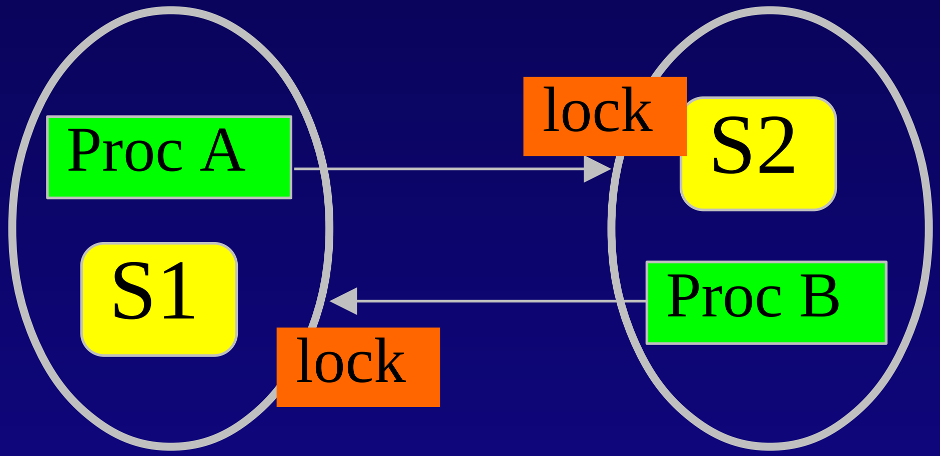
Semaphore

int semctl(int semid, int snum, int cmd, ...);

- Performs the control operation specified by *cmd* on the semaphore set identified by *semid*, or on the *snum*-th semaphore
- IPC_SETVAL/IPC_GETVAL
  - Set, Get value of semaphore
- IPC_RMID
  - Remove semaphore set
- ....

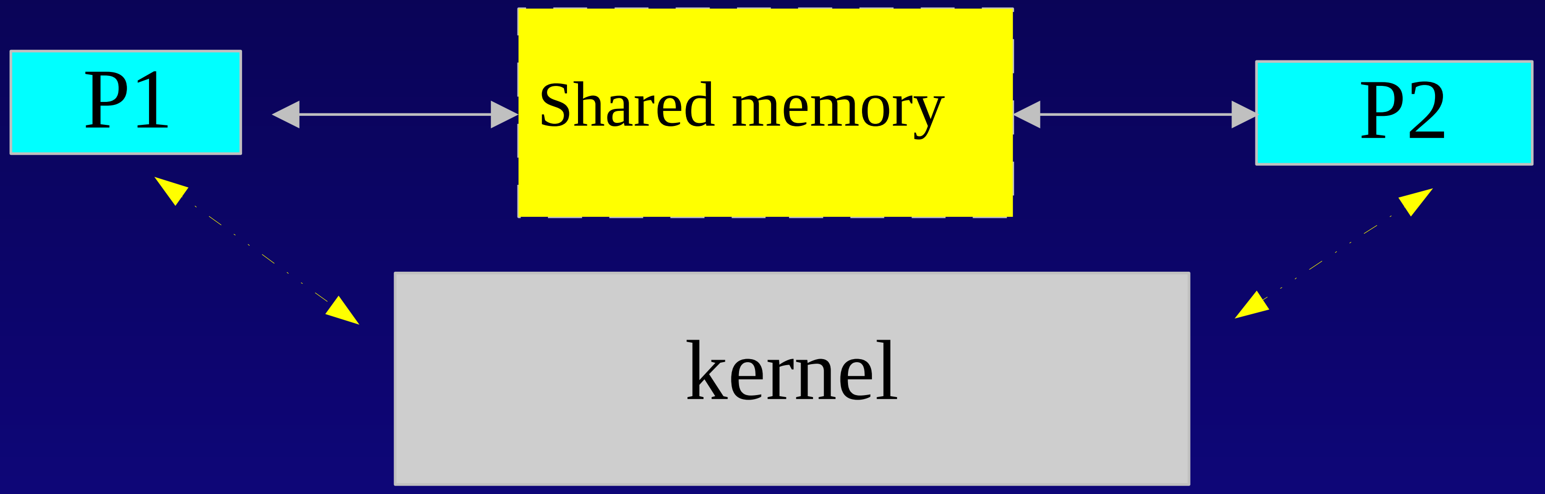
Giorgio Richelli
giorgio_richelli@it.ibm.com

DeadLock



Giorgio Richelli
giorgio_richelli@it.ibm.com

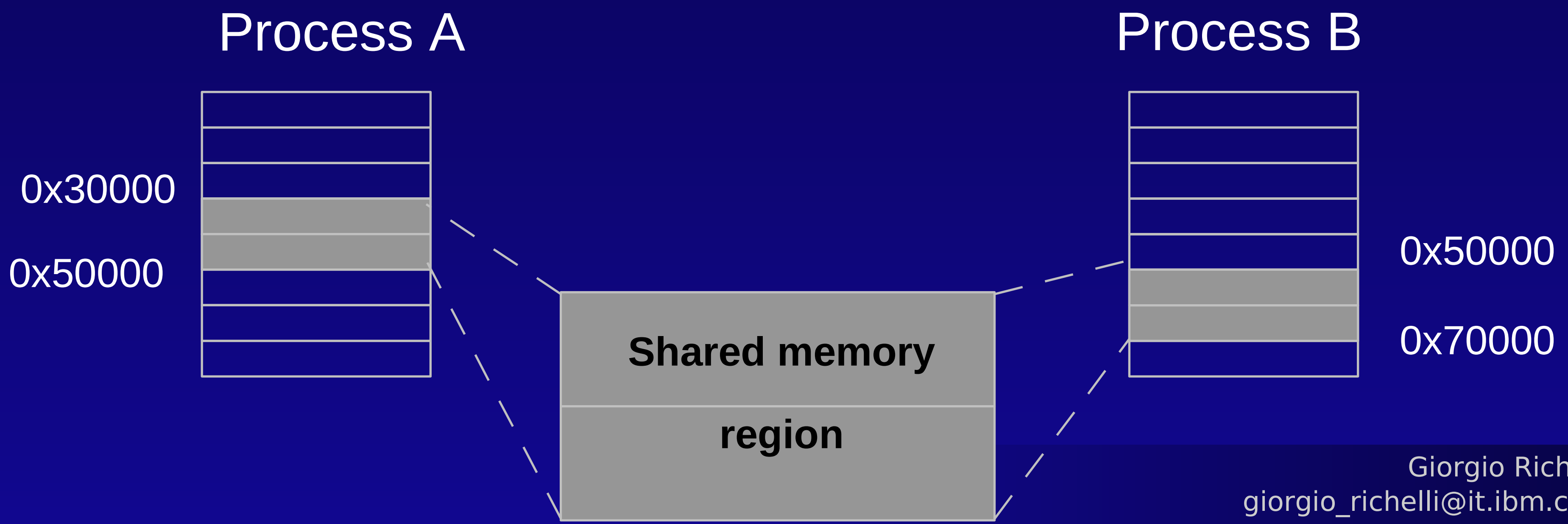
IPC with shared memory



Giorgio Richelli
giorgio_richelli@it.ibm.com

Shared Memory

A portion of physical memory that is share by multiple processes.



Giorgio Richelli
giorgio_richelli@it.ibm.com

Shared Memory API

int shmget(key_t key, size_t size , int flag);

- returns the identifier of the shared memory segment associated with *key*
- *key*
  - IPC_PRIVATE, ...
- *size*
  - size of shared area
- *flag*
  - IPC_CREATE, permissions, ..

Giorgio Richelli
giorgio_richelli@it.ibm.com

Shared Memory

Segments are:

- inherited after *fork()*
- detached, not destroyed, after *exec()* or *exit()*

Giorgio Richelli
giorgio_richelli@it.ibm.com

Shared Memory API

void *shmat(int shmid, void * shmaddr, int shmflag);

- attaches the shared memory segment identified by *shmid* to the address space of the calling process
- *shmaddr*
  - Usually NULL, otherwise address requested for segment
- *shmflag*
  - SHM_RDONLY, SHM_RND, ...

Does not modify the *brk*

Shared Memory API

int shmdt(void *shmaddr);

Detaches the shared memory segment at *shmaddr* from address space of calling process.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Shared Memory API

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- performs operation indicated by *cmd* on shared memory segment identified by *shmid*
- *cmd*
  - IPC_RMID, ...
- *buf*
  - address of struct to hold information about segment

Giorgio Richelli
giorgio_richelli@it.ibm.com

Shared Memory API

Shared memory segments must be explicitly removed (IPC_RMID)

The segment is marked as removed, but it will be destroyed when the last process call *shmdt()*

Giorgio Richelli
giorgio_richelli@it.ibm.com

Ftok

IPC key can be correlated to a file name

key_t ftok(char *pathname, int ndx)

builds a key based on *pathname* and *ndx*

Giorgio Richelli
giorgio_richelli@it.ibm.com

Security

If a process holds the key, it might access the resource.

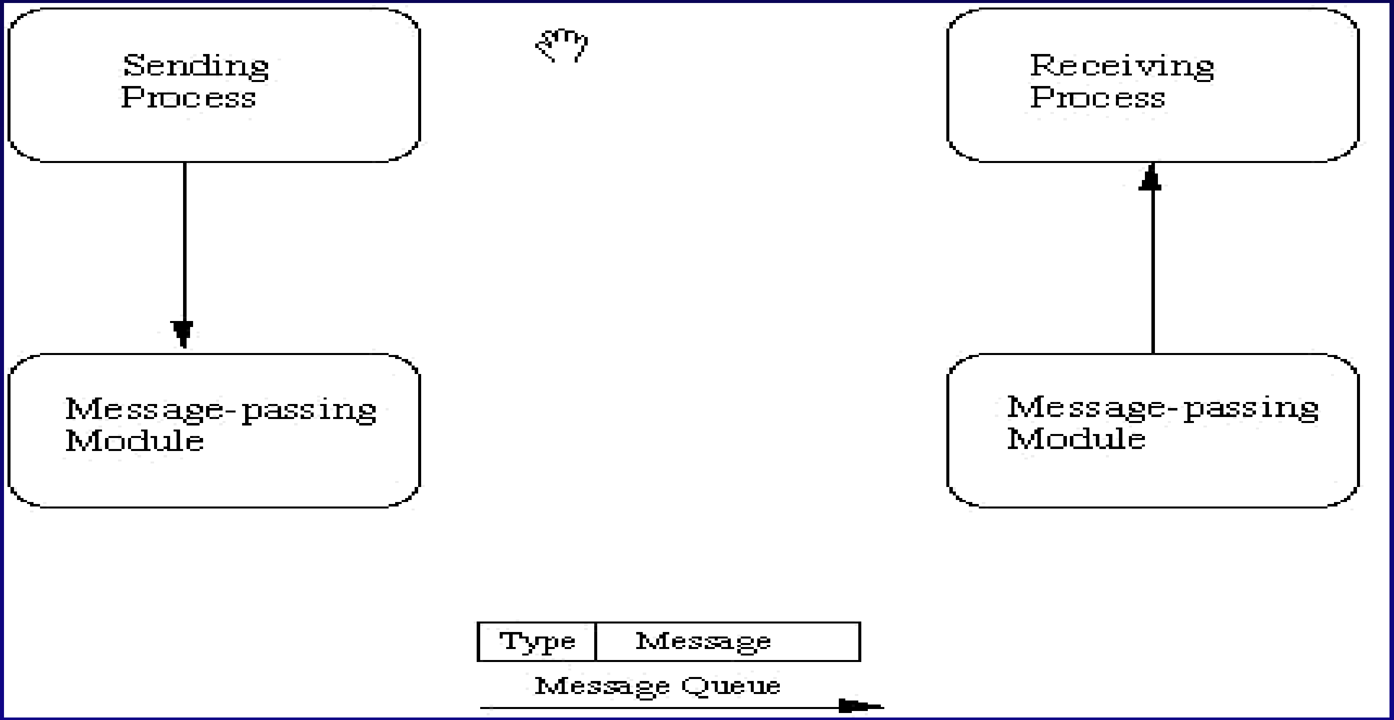
Giorgio Richelli
giorgio_richelli@it.ibm.com

Message Queues

Processes can send and receive messages in an arbitrary order.
Unlike pipes, each message has an explicit length.
Messages can be assigned a specific type.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Message Queues



Giorgio Richelli
giorgio_richelli@it.ibm.com

Message Queue

int = msgget(key_t key, int flag);

- returns the message queue identifier associated with the value of the *key* argument.
- *key:* IPC_PRIVATE, ..
- *flag:* IPC_CREAT, ...

Giorgio Richelli
giorgio_richelli@it.ibm.com

Message Queue

int msgsnd(int msgqid, struct msgbufp *msgp, size_t size, int flag)

appends a copy of the message pointed to by *msgp* to the message queue whose identifier is specified by *msqid*

flag: IPC_NOWAIT, ..

Message Queue

count =msgrcv(int msgqid, struct msgbuf *msgp, size_t size, long type, int flag)

- reads a message from the message queue specified by *msqid* into the buffer pointed to *msgp*
- *size:* maximum size (in bytes) for the mtext member of msgp
- *type:* 0, [type], - [type]
- *flag:*
  - IPC_NOWAIT, MSG_NOERROR, MSG_EXCEPT

Giorgio Richelli
giorgio_richelli@it.ibm.com

Message Queue

```
struct msgbuf {
    long mtype;    /* message type */
    char mtext[MSGSZ]; /* message text of length MSGSZ */
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

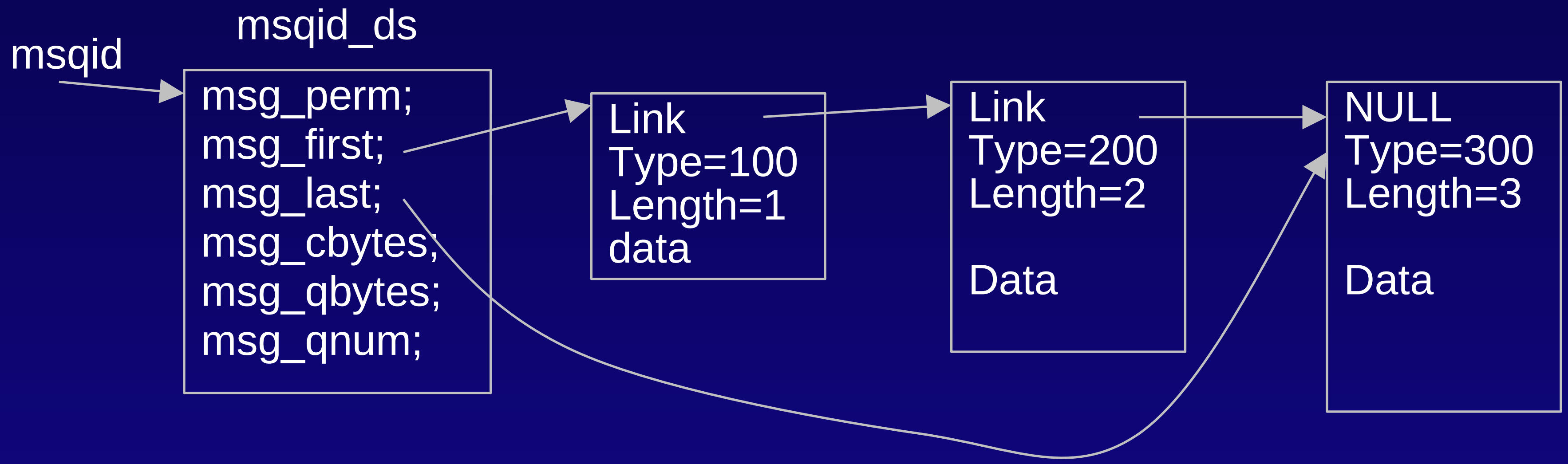
Message Queue

int msgctl(int msqid, int cmd, struct msqid_ds *buf);

- performs the control operation specified by *cmd* on the message queue with identifier *msqid*
- *cmd*
  - IPC_RMID, ...

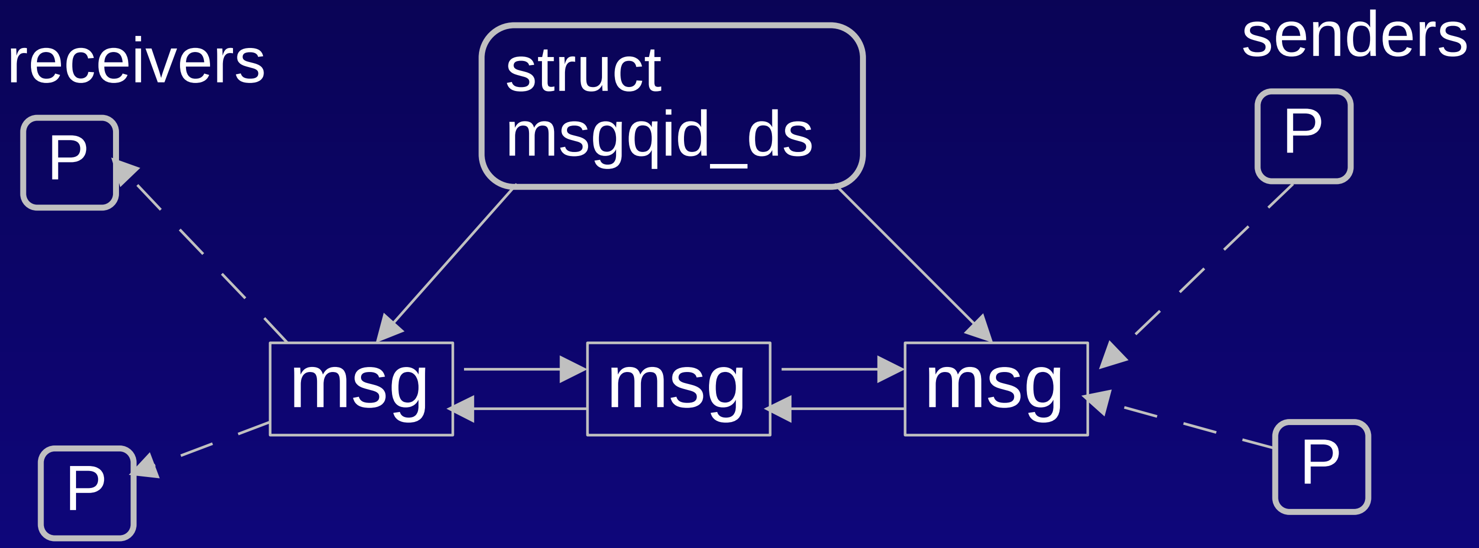
Giorgio Richelli
giorgio_richelli@it.ibm.com

An example of a msq



Giorgio Richelli
giorgio_richelli@it.ibm.com

Message Queue



Giorgio Richelli
giorgio_richelli@it.ibm.com

Sockets

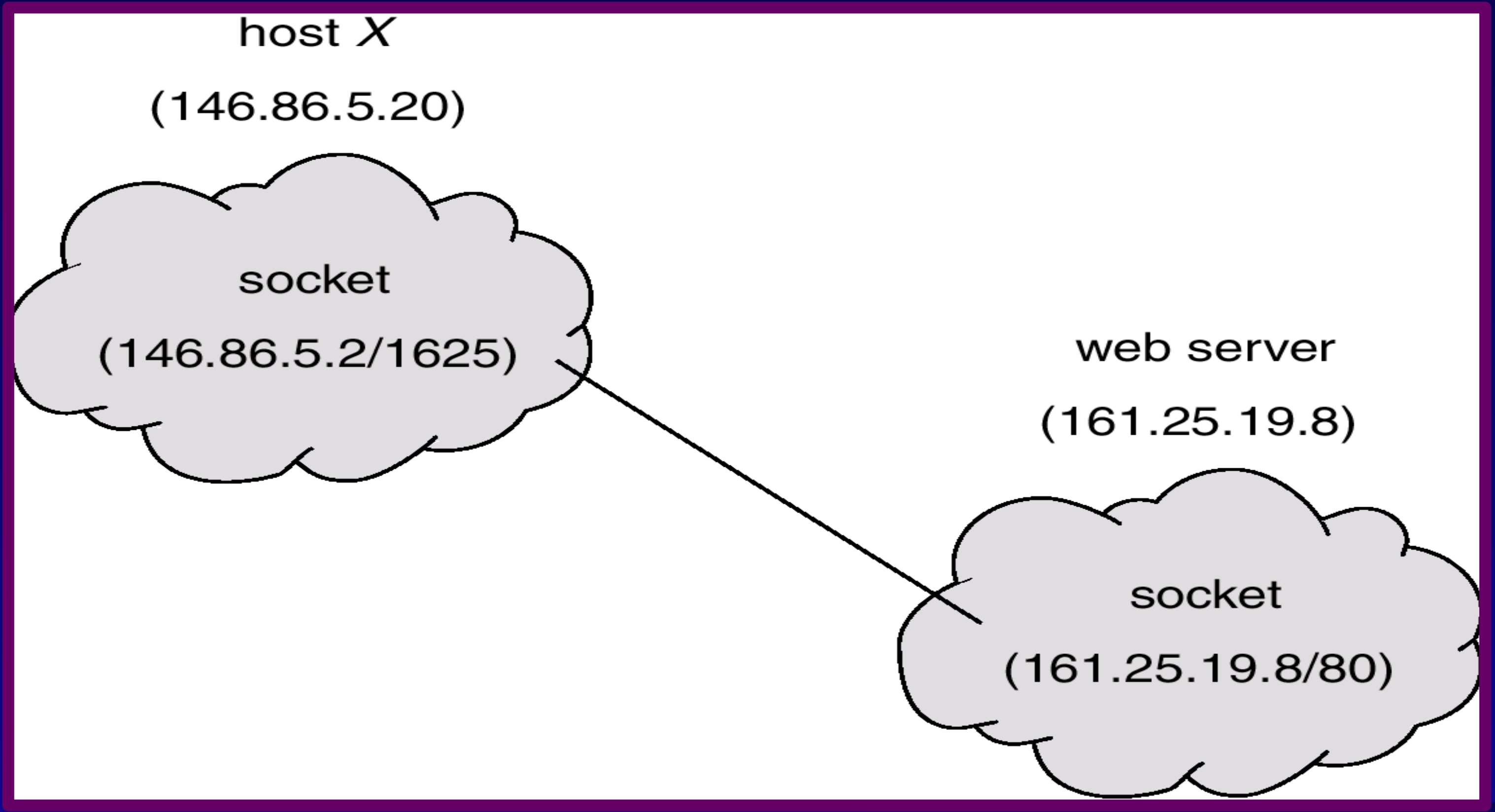
A socket is an endpoint of communication.

An in-use socket it usually bound with an address; the nature of the address depends on the communication domain of the socket.

e.g. *161.25.19.8:1625* refers to port 1625 on host 161.25.19.8

Giorgio Richelli
giorgio_richelli@it.ibm.com

Sockets



Giorgio Richelli
giorgio_richelli@it.ibm.com

Sockets

Communication consists between a pair of sockets.

A characteristic property of a domain is that processes communication in the same domain use the same address format.

- protocol domain
- address domain

Giorgio Richelli
giorgio_richelli@it.ibm.com

Sockets

A single socket can communicate in only one domain

Commonly implemented domains:

- UNIX (PF_LOCAL)
- Internet (PF_INET)
- .... (lots) ....

Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket Types

Stream

- Reliable, duplex, sequenced data streams.
- Supported in Internet domain by the TCP protocol.
- In UNIX domain, pipes are implemented as a pair of communicating stream sockets.

Sequenced packet

- Provide similar data streams, except that record boundaries are provided.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket Types

Datagram:

- Transfer messages of variable size in either direction.
- Supported in Internet domain by UDP protocol

Reliably delivered message:

- Transfer messages that are guaranteed to arrive.
- Almost unsupported.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket Types

Raw:

- allow direct access by processes to the protocols that support the other socket types.
  - E.g., in the Internet domain, it is possible to reach TCP, IP beneath that, or a deeper Ethernet protocol.
- Useful for developing new protocols.

Socket System Calls

The *socket()* call creates a socket

A name/address is bound to a socket by *bind()*

The *connect()* system call is used to initiate a connection

Socket System Calls (Cont.)

close()

terminates a connection and destroys the associated socket

select()

multiplex data transfers on several file descriptors and /or socket descriptors

Socket System Calls

A server process usually calls:

- *socket()* to create a socket
- *bind()* to bind an address
- *listen()* to indicate willingness to accept connections from clients
- *accept()* to accept an individual connection
  - eventually, *fork()* a new process after the *accept()*
- *send() & recv()* to move data
- *close()* when all is done on the connection

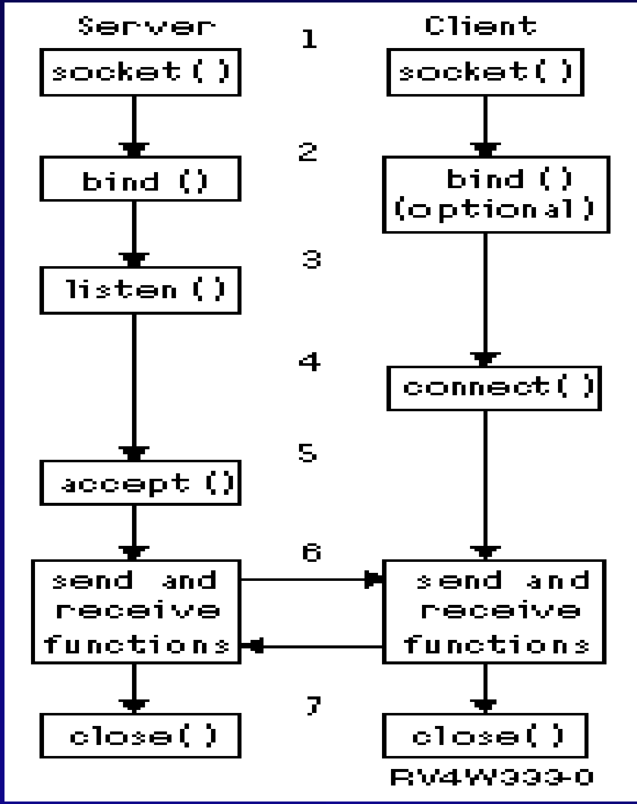
Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket System Calls

A client process usually calls:

- *socket()* to create a socket
- *connect()* to estabilish a connection with server
- *send(), recv()* to move data
- *close()* to close the connection

Socket Calls Flow



Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket System Calls

int socket(int domain, int type, int protocol);

- creates an endpoint for communication and returns a descriptor
- *domain:* PF_UNIX, PF_INET, ...
- *type:* SOCK_STREAM, SOCK_DGRAM, ...
- *protocol:* 0, IPPROTO_TCP, IPPROTO_UDP, ...

Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket System Calls

int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);

gives the socket *sockfd* the local address *my_addr* (*addrlen* bytes long)

Socket System Calls

int listen(int s, int backlog);

- specify willingness to accept incoming connections and a queue limit (for pending connections)
- *s:* socket
- *backlog:* maximum length for the queue

Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket System Calls

int accept(int s, struct sockaddr *addr, socklen_t *addrlen);

extracts the first connection request on the queue of pending connections on *s*, creates a new connected socket with (mostly) the same properties

s: socket

addr: will contain the “from” address

addrlen: bytes available in *addr*

Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket System Calls

ssize_t send(int s, const void *buf, size_t len, int flags);

- transmit a message to another socket
- *s* must be “connected”
- almost identical to *write()*, execpt for *flags*
- *flags:* MSG_DONTWAIT, MSG_DONTROUTE, ...

Socket System Calls

ssize_t recv(int s, void *buf, size_t len, int flags);

- receive messages from a (connected) socket
- almost identical to a *read()*, except for *flags*
- *flags:*
  - MSG_PEEK, MSG_TRUNC, MSG_WAITALL, ...

Socket System Calls

int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);

attempts to connect *sockfd* to another socket, specified by *serv_addr*, which is an address (of length *addrlen*) in the communications space of the socket.

returns: 0 / -1

Giorgio Richelli
giorgio_richelli@it.ibm.com

Socket System Calls

int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds, struct timeval *timeout);

- wait on a number of file descriptors (until, eventually, a timeout occurs)
- Three sets of descriptors are watched.
  - *readfds* see if characters become available for reading *writefds* will be watched to see if a write will not block *exceptfds* will be watched for exceptions
- Macros to manipulate the sets:
  - FD_ZERO, FD_SET, FD_CLR, FD_ISSET

Giorgio Richelli
giorgio_richelli@it.ibm.com

Getting Host Name & Address(es)

struct hostent *gethostbyname(const char *name);

returns a structure of type *hostent* for the given host name (either an host name, or an IP address)

```
struct hostent {
    char    *h_name;       /* official name of host */
    char    **h_aliases;   /* alias list */
    int     h_addrtype;    /* host address type */
    int     h_length;      /* length of address */
    char    **h_addr_list; /* list of addresses */
}
#define h_addr  h_addr_list[0]  /* backw. compatibility */
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Getting Host Name & Address(es)

struct hostent *gethostbyaddr(const char *addr, int len, int type);

- returns a structure of type *hostent* for the given host address addr of length len and address type type.
- *type:* AF_INET, AF_INET6

Giorgio Richelli
giorgio_richelli@it.ibm.com

Endiannes

Big Endian

- the most significant byte of any multibyte data field is stored at the lowest memory address

Little Endian

- the least significant byte of any multibyte data field is stored at the lowest memory address

Giorgio Richelli
giorgio_richelli@it.ibm.com

Host Independent Formats

Intel CPUs are *Little Endian*, while the network byte order is *Big Endian*

uint32_t htonl(uint32_t hostlong);

uint16_t htons(uint16_t hostshort);

from host byte order to network byte order

uint32_t ntohl(uint32_t netlong);

uint16_t ntohs(uint16_t netshort);

from network byte order to host byte order

Giorgio Richelli
giorgio_richelli@it.ibm.com