Inter Process Communication (IPC)

Giorgio Richelli
giorgio_richelli@it.ibm.com

Contents

- Introduction
- Universal IPC Facilities
- System V IPC

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)

Giorgio Richelli
giorgio_richelli@it.ibm.com

Signal Generation & Handling

- Pending: between generated and delivered.
- System V: 15 signals
- 4BSD/SVR4/POSIX : 31 signals
- POSIX 1b Realtime Signals (at least 8)
- Signal numbers different from those mandated by POSIX might be 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

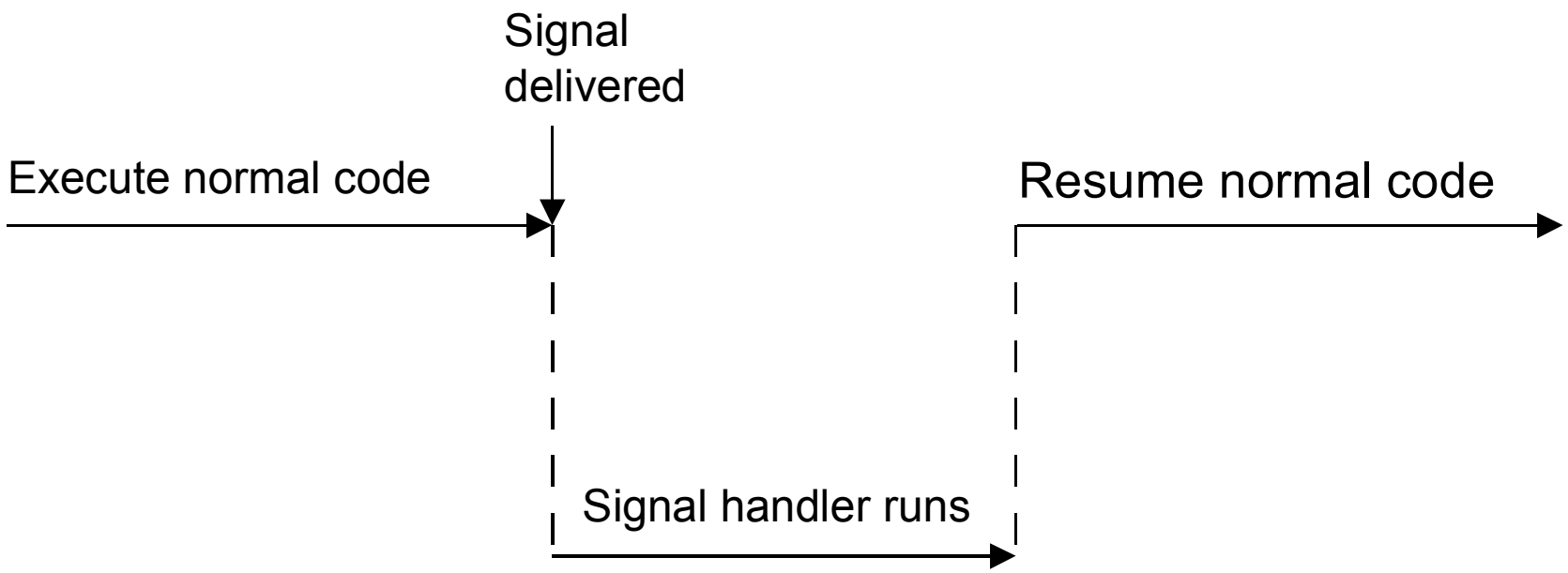
Giorgio Richelli
giorgio_richelli@it.ibm.com

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)
  - VERY OLD & UNLIKELY
- 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 POSIX implementation

```
int sigsetup(int sig, void (*h)())
{
    struct sigaction s;
    sigset_t nmask;
    s.sa_handler = h;
    sigemptyset (&nmask);
    s.sa_mask = nmask;
    s.sa_flags = 0;
   return(sigaction (sig, &s, NULL));
}
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Creating Processes

Giorgio Richelli
giorgio_richelli@it.ibm.com

New Processes & Programs

- `int fork()`:
  - creates a new process.
  - returns 0 to the child, PID to the parent
- `int exec*(..)`:
  - begins to execute a new program

Giorgio Richelli
giorgio_richelli@it.ibm.com

Using fork & exec

```
if ((ChildPid = fork())==0){
   /* child code*/
   … …
  if (execve(“new program”),…)<0) {
      perror(“execve failed.”);
      exit(-1)
  }
} else if (ChildPid <0) {
      perror(“fork failed”);
      exit(-1)
}
/*parent continues here*/
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

MGT - [~/MasterINFN]

File New Tab Edit Settings Help

```
[giorgio@gastone MasterINFN]$ cat p2.c
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    int pid, ppid, pid1;

    pid=fork();
    if (pid==0) {
        printf("Child Process - My PID:%d, Parent PID:%d\n", getpid(), getppid());
    }
    else {
        printf("Parent Process- My PID:%d, Parent PID:%d\n", getpid(), getppid());
    }
    return 0;
}
[giorgio@gastone MasterINFN]$ ps
  PID TTY          TIME CMD
 8582 pts/2    00:00:00 bash
18351 pts/2    00:00:00 ps
[giorgio@gastone MasterINFN]$ cc p2.c
[giorgio@gastone MasterINFN]$ a.out
Parent Process- My PID:18364, Parent PID:8582
Child Process - My PID:18365, Parent PID:18364
[giorgio@gastone MasterINFN]$
```

~/MasterINFN Root ~/MasterINFN

Giorgio Richelli
giorgio_richelli@it.ibm.com

Process Creation

- Almost an exact clone of the parent.
  - Reserve swap space for the child
  - Allocate a new PID and proc structure for the child
  - Initialize proc structure
  - Allocate ATM (address translation map)
  - Allocate *u* area and copy
  - Update the *u* area to refer to the new ATM & Swap space
  - Add the child to the set of processes sharing the text region of the program
  - Duplicate the parent's data and stack regions update ATM to refer to these new pages.
  - Acquire references to shared resources inherited by the child
  - Initialize the hardware context
  - Make the child runnable and put it on a scheduler queue
  - Arrange to return with 0
  - Return the PID to the parent

Giorgio Richelli
giorgio_richelli@it.ibm.com

Fork Optimization

- It is wasteful to make an actual copy of the address space of the parent
  - Copy-on-write:

    only the pages that are modified must be copied.(SYSV)
  - vfork() (BSD):

    The parent loans the address space and blocks until the child returns to it.

    dangerous

    (csh exploits it)

Giorgio Richelli
giorgio_richelli@it.ibm.com

Invoking a New Program

- Process address space
  - Text: code
  - Initialized data
  - Uninitialized data(bss)
  - Shared memory(SYSV)
  - Shared libraries
  - Heap: dynamic space
  - User stack: space allocated by the kernel

Giorgio Richelli
giorgio_richelli@it.ibm.com

Awaiting Process Termination

```
wait(statusp);/* SV, BSD & POSIX*/
wait3(statusp,options,rusagep);/*BSD*/
waitpid(pid,statusp,options);/*POSIX*/
waitid(idtype,id,infop,options);/*SVR4*/
```

Giorgio Richelli
giorgio_richelli@it.ibm.com

Zombie Processes

- Only holds proc structure.
- `wait()` frees the proc
  - parent or the init process.
- When child dies before the parent & parent doesn't wait for all childs, then the proc is never released.

Giorgio Richelli
giorgio_richelli@it.ibm.com

Zombie Processes

- Scenario:
  - Child exits -> [Defunct]
  - Parent doesn't wait for child and SIGCHLD ignored
  - Parent exits -> Child is inherited by *init*

Giorgio Richelli
giorgio_richelli@it.ibm.com

Universal IPC Facilities

- Signals

  Kill

  Sigpause

  ^C

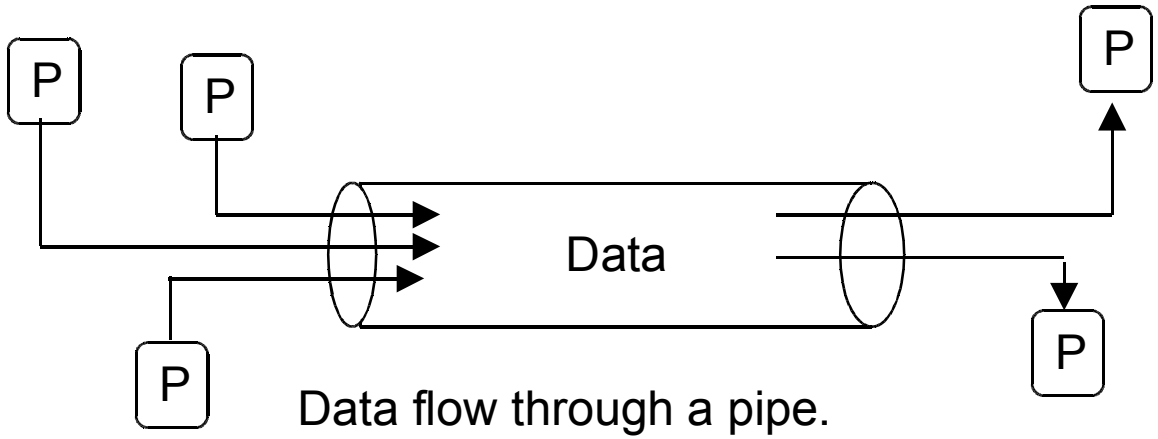
  - Expensive
  - Limited: only 31 signals.
  - Signals are not enough.

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

Giorgio Richelli
giorgio_richelli@it.ibm.com

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 <u>atomically</u> and only if they can <u>all</u> be <u>simultaneously</u> 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

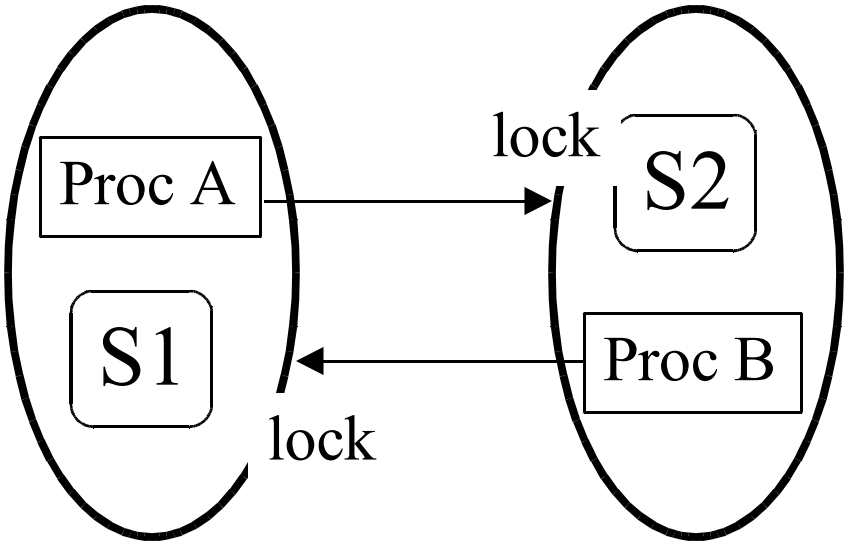
- int semctl(int semid, int snum, int cmd, ...);
  - Performs the control operation specified by *cmd* on the semaphore set identified by *semid*, or on the *snum*-th semaphore
  - IPC_SETVAL/IPC_GETVAL

    Set, Get value of semaphore
  - IPC_RMID

    Remove semaphore set
  - ....

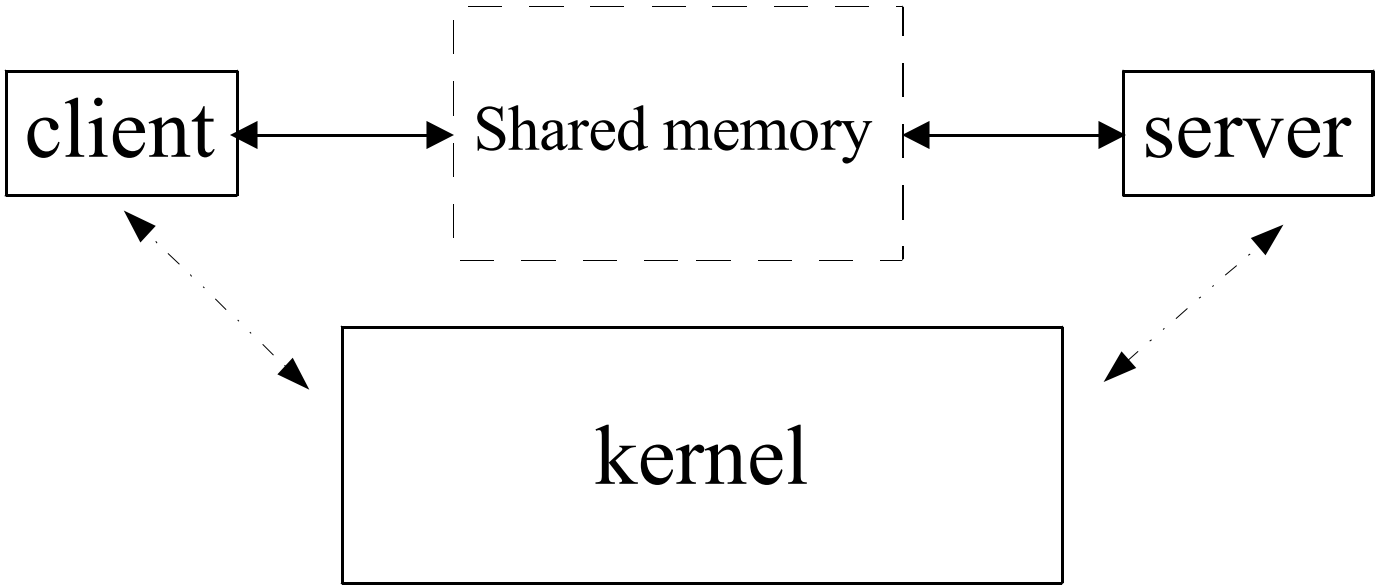
Giorgio Richelli
giorgio_richelli@it.ibm.com

DeadLock



Giorgio Richelli
giorgio_richelli@it.ibm.com

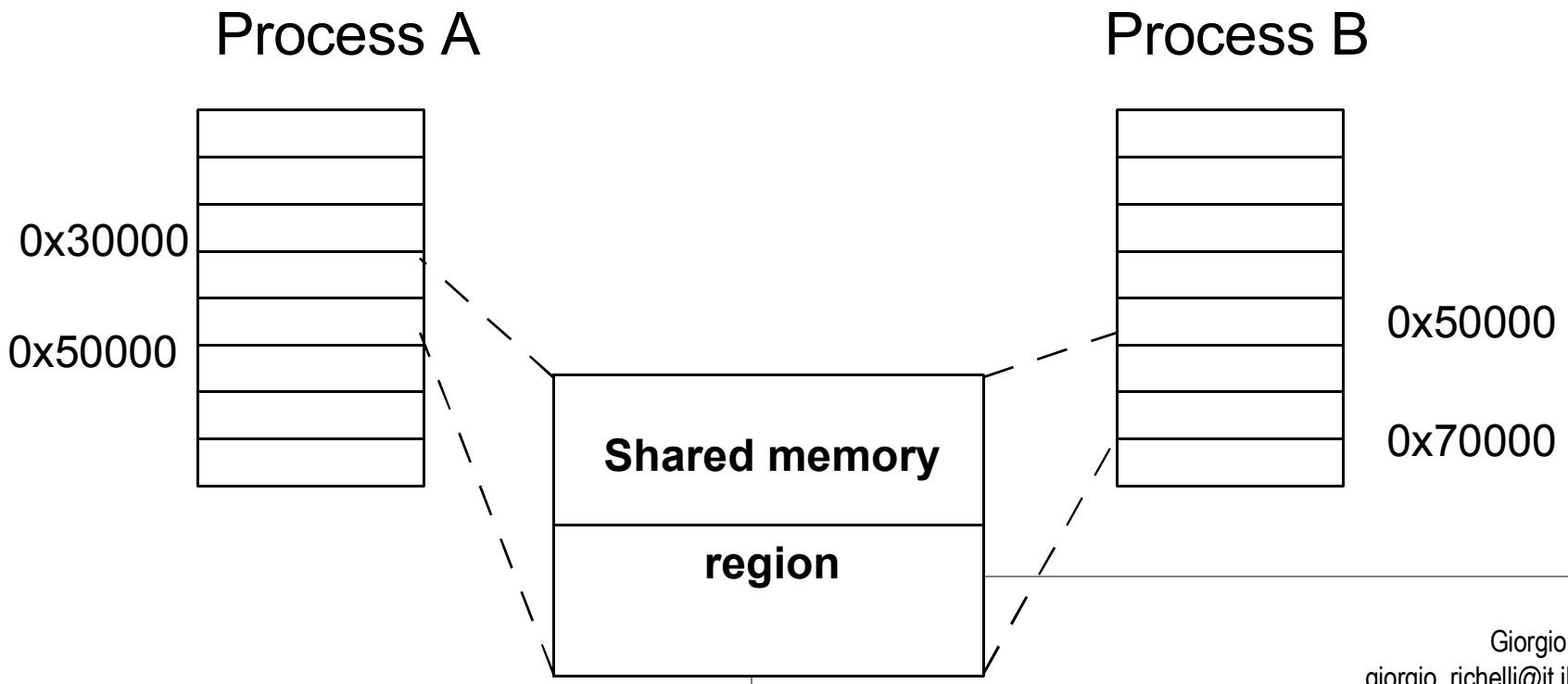
Client/server with shared memory



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, <u>not destroyed</u>, 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*

Giorgio Richelli
giorgio_richelli@it.ibm.com

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 <u>marked</u> as removed, but it will be destroyed when the last process call *shmdt()*

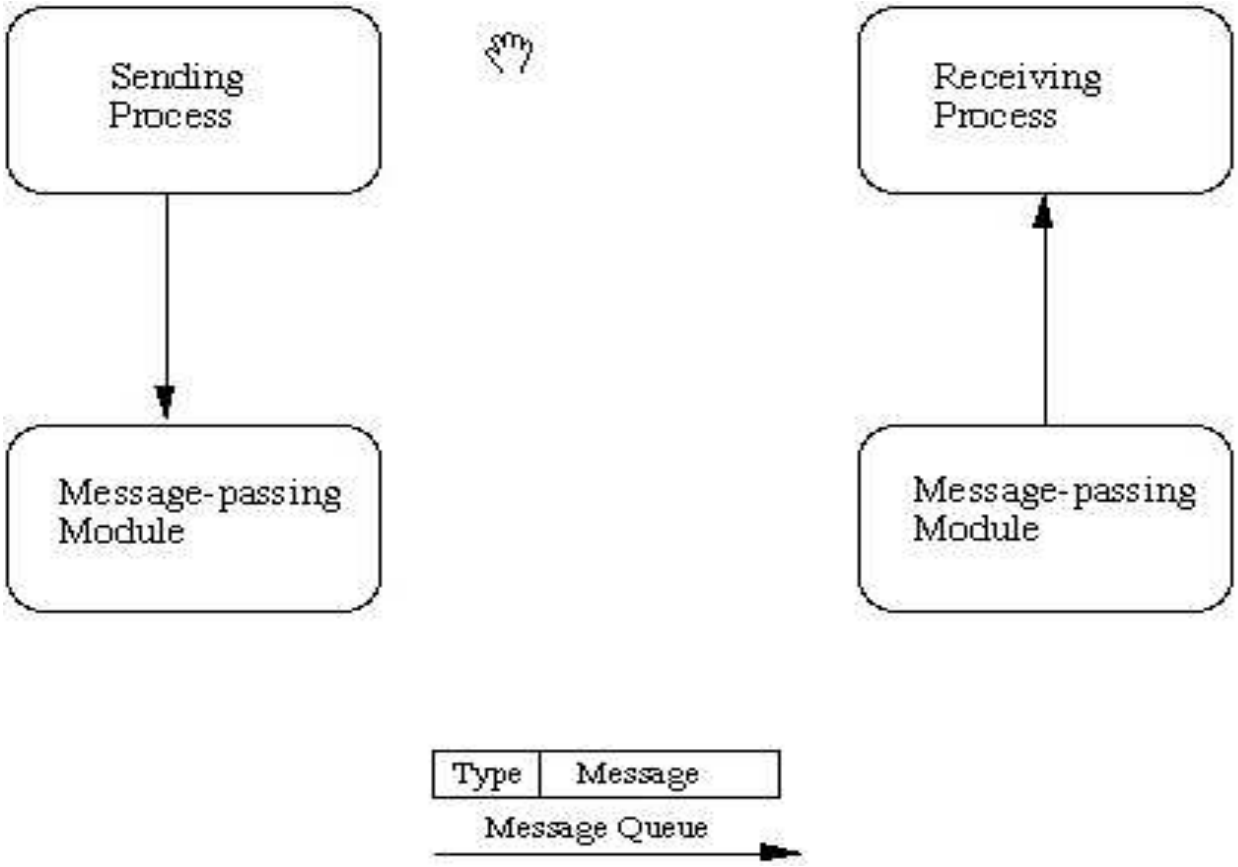
Giorgio Richelli
giorgio_richelli@it.ibm.com

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, ..

Giorgio Richelli
giorgio_richelli@it.ibm.com

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

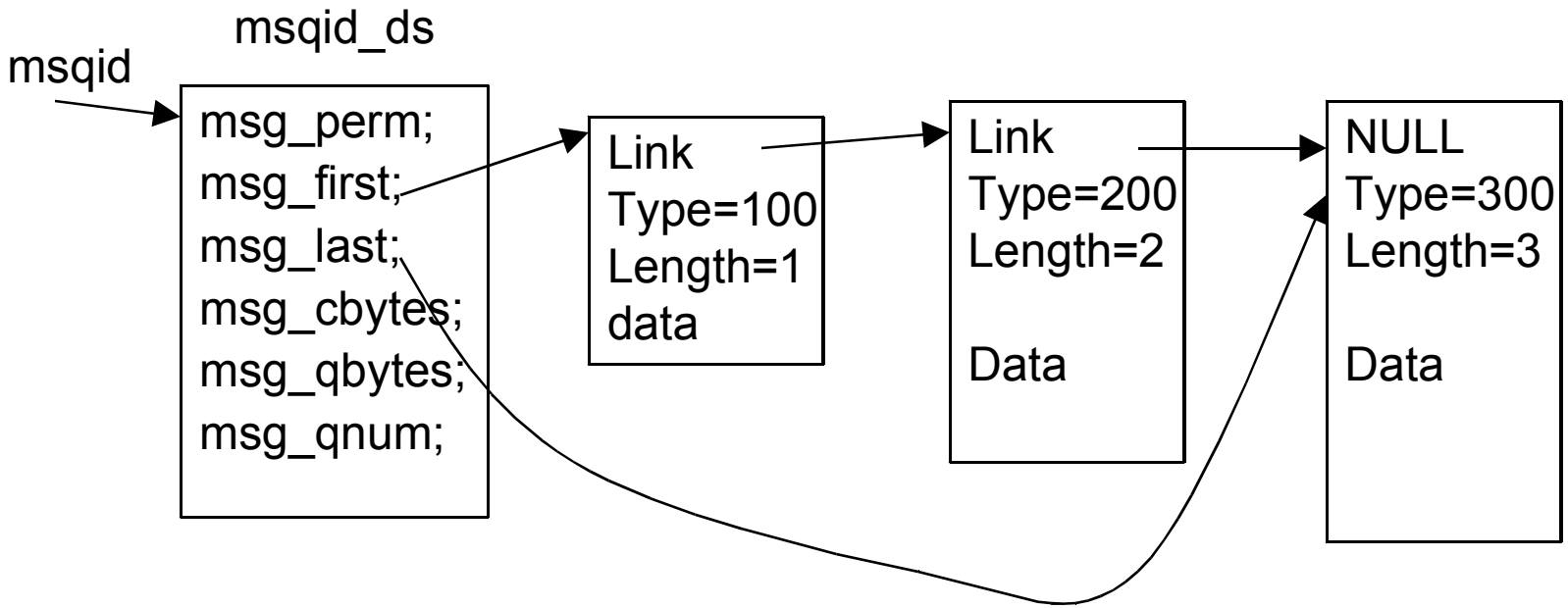
Message Queue

- 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, ...

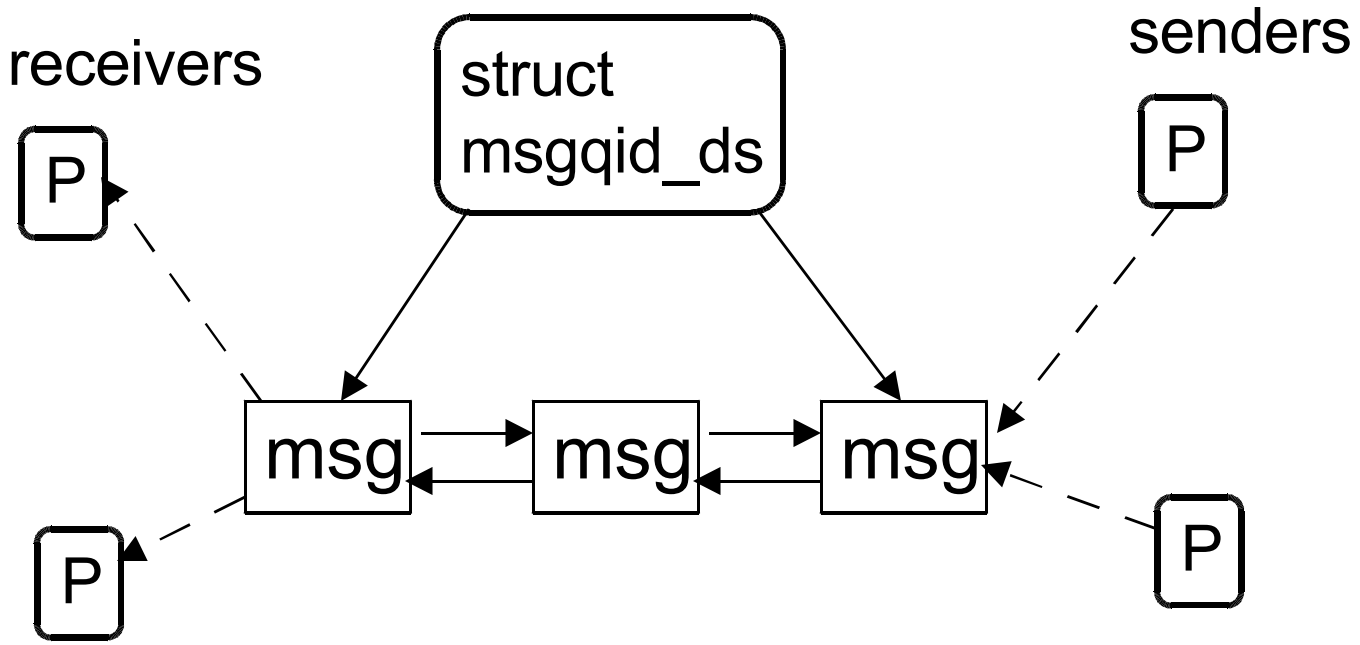
Giorgio Richelli
giorgio_richelli@it.ibm.com

An example of a msq



Giorgio Richelli
giorgio_richelli@it.ibm.com

Message Queue



Giorgio Richelli
giorgio_richelli@it.ibm.com

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.
- For shared memory, remove the segment after (all) *shmat*

Giorgio Richelli
giorgio_richelli@it.ibm.com