Processes

Thanks to Fabienne Boyer and Arnaud Legrand and the books (Silberschatz, tanenbaum)

Process (1/2)

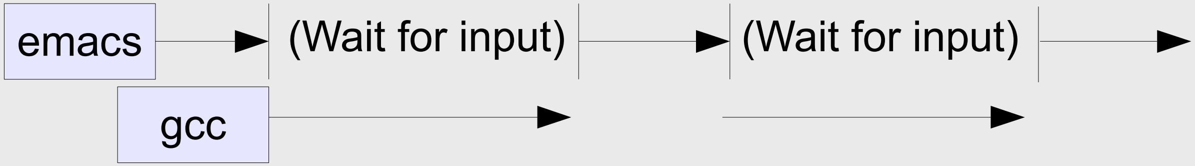
- A process is an instance of a running program
  - Eg : gcc, sh, firefox ...
  - Created by the system or by an application
  - Created by a parent process
  - Uniquely idendified (PID)

- Correspond to two unit :
  - Execution unit
    - Sequential control flow (exécute une suite d'instruction)

  - Addressing unit
    - Each process has its own address space
    - Isolation

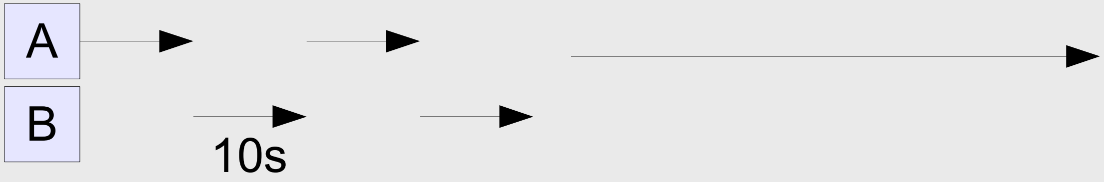
Concurent processes

- **Multiple processes can increase CPU utilization**
  - **Overlap one process's computation with another's wait**



- **Multiple processes can reduce latency**
  - **Running A then B requires 100 secs for B to complete**

  - **Running A and B concurently improves the average response time**



Execution context

- A process is characterized by its context
- Process' current state
  - Memory image
    - Code of the running program
    - Static and dynamic data
  - Register's state
    - Program counter (PC), Stack pointer (SP) ...
  - List of open files
  - Environment Variables
  - …

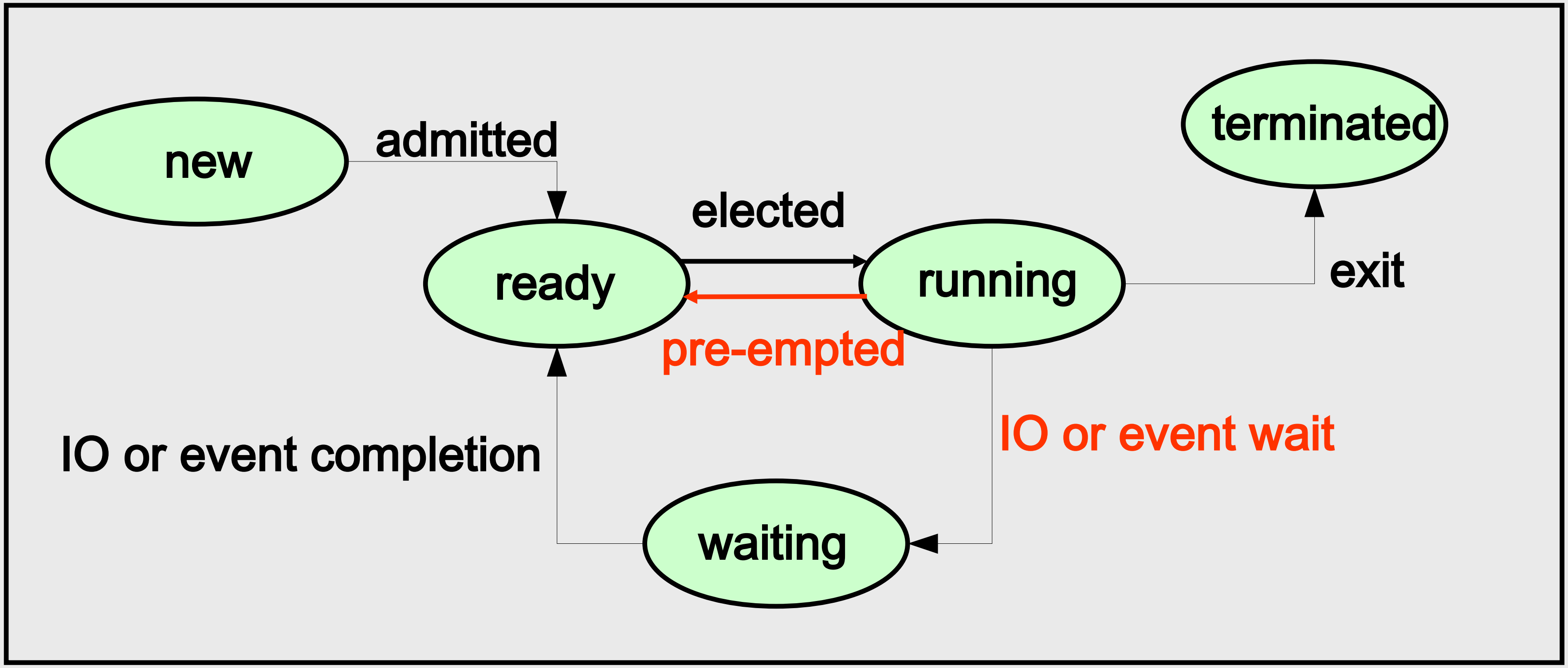
→ To be saved when the process is switched off

→ To be restored when the process is switched on

Running mode

- User mode
  - Restricted access to process own adress space
  - Limited instruction set
- Supervisor mode
  - Full memory access
  - Full access to the instruction set
- Interrupt, trap
  - Asynchronous event
  - Illegal instruction
  - System call request

Process Lifecycle



- Which process should kernel run ?
  - If 0 runnable, run a watchdog, if 1 runnable, run it
  - If n runnable, make scheduling decision

Process management by the OS

- **Process files**
  - Ready queue (ready process)
  - Device queue (Process waiting for IO)
  - Blocked Queue (Process waiting for an event)
  - ...

→ **OS migrates processes across queues**

Process Control Structure

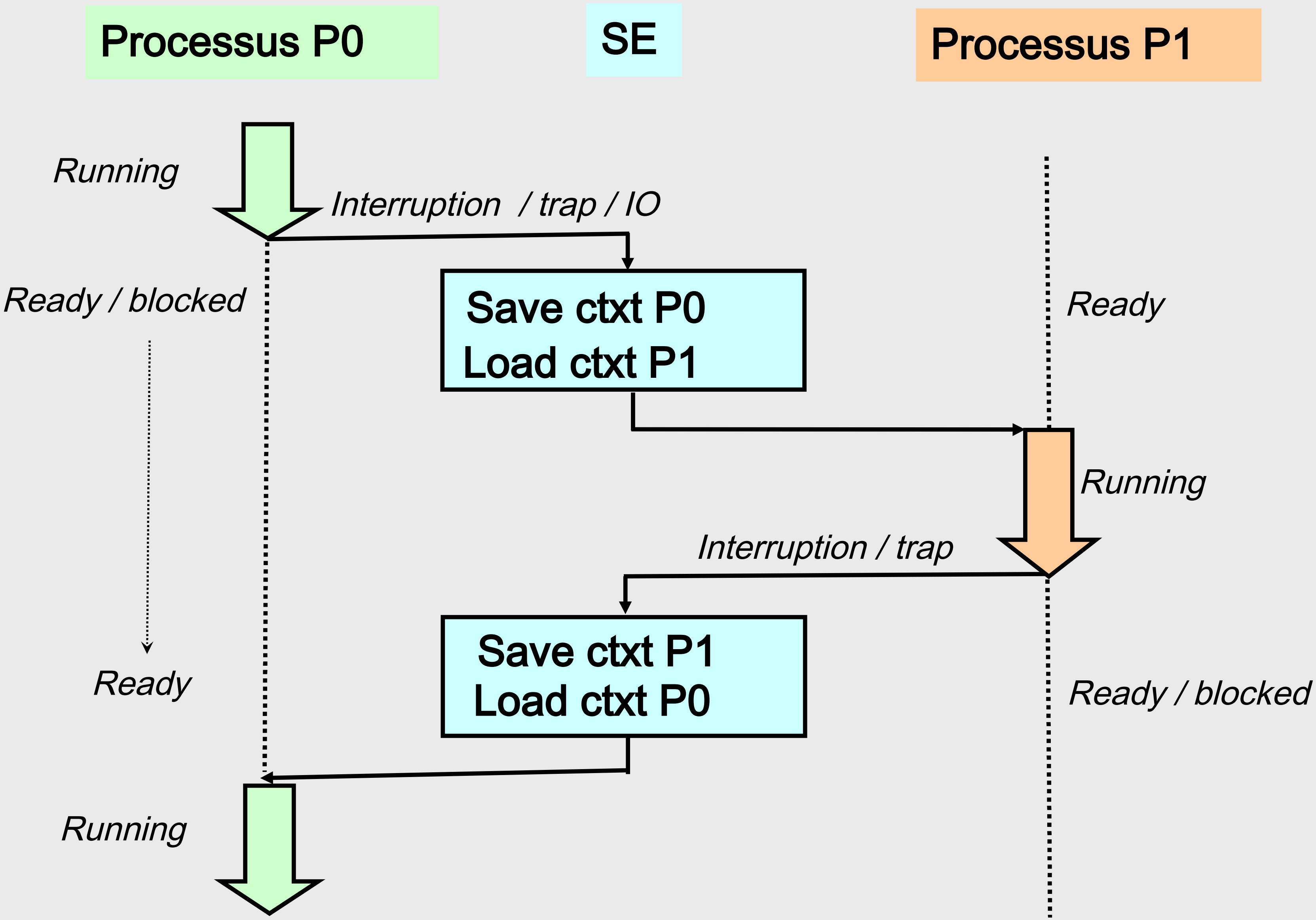
Hold a process execution context

PCB (Process Control Block):
Data required by the OS to manage process

Process tables:
PCB [MAX-PROCESSES]

- Process state (ready, ...)
- Process ID
- User ID
- Registers
- Address space
- Open files
- …

Process context switch



Processus P0

SE

Processus P1

Running

Interruption / trap / IO

Ready / blocked

Save ctxt P0
Load ctxt P1

Ready

Running

Interruption / trap

Save ctxt P1
Load ctxt P0

Ready

Ready / blocked

Running

CPU Allocation to processes

- The scheduler is the OS's part that manage CPU Allocation

- Criteria / Scheduling Algorithm
  - Fair (no starvation)
  - Minimize the waiting time for processes
  - Maximize the efficiency (number of jobs per unit of time)

Scheduling criteria

- Algorithm with/without Pre-emption
  - A process can be interrupted if pre-emption (time sharing) or forbidden (multiprogramming only)
- Choice of the Quantum
- Priority management
  - System Process
  - User process
    - Very interactive
    - Few interactive

Simple scheduling algorithms (1/2)

- **Non-pre-emptive scheduler (multiprogramming)**
  - FCFS (First Come First Served)
    - Fair
- **Pre-emptive scheduler (multiprogramming+timesharing)**
  - SJF (Shortest Job First)
    - Priority to shortest task
    - Require to know the execution time (model estimated from previous execution)
    - Unfair but optimal in term of response time
  - Round Robin (fixed quantum)
    - Each processus is affected a CPU quantum (10-100 ms) before pre-emption
    - Efficient (unless the quantum is too small), fair / response time (unless the quantum too long)

Simple scheduling algorithms (2/2)

Round robin with static priority

- A priority is associated with a quantum number (1,2,4, etc)
- High priority induces small quantum
- Processes are elected according to their priority
- Good response time (priority to interactive process)
- Starvation possible
  - Problem ≡ low priority processes may never be elected
  - Solution ≡ "Aging" – increasing a process priority according to its age → dynamic priority

Round robin with dynamic priority

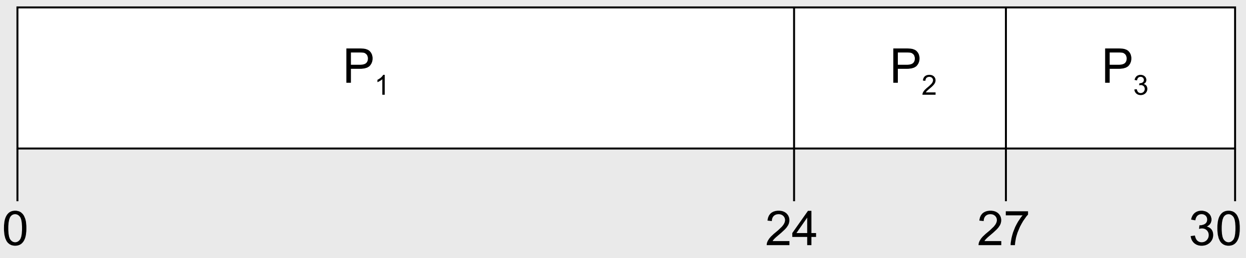
- An additional parameter (e.g. a duration and an interrupt count, or ages) allows to increase/decrease a process priority.
- Fair

First-Come, First-Served (FCFS)--non pre-emptive

Process's execution time

P_1	24
P_2	3
P_3	3

- Let's these processes come in this order : P1,P2,P3



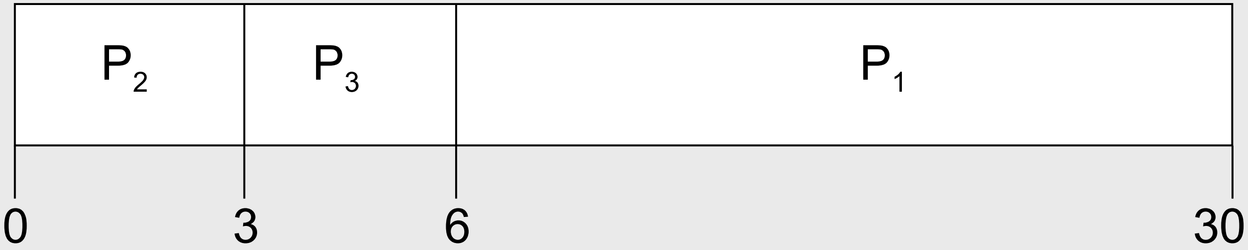
- Response time of P_1 = 24; P_2 = 27; P_3 = 30
- Mean time : (24 + 27 + 30)/3 = 27

(c) Sylberschatz

First-Come, First-Served (FCFS) (2/2)

Let's these processes come in this order :

P_2 , P_3 , P_1 .



- **Response time :** $P_1 = 30; P_2 = 3; P_3 = 6$
- **Mean time : (30 + 3 + 6)/3 = 13**
- **Better than the precedent case**

➔ ***Schedule short processes before***

(c) Sylberschatz

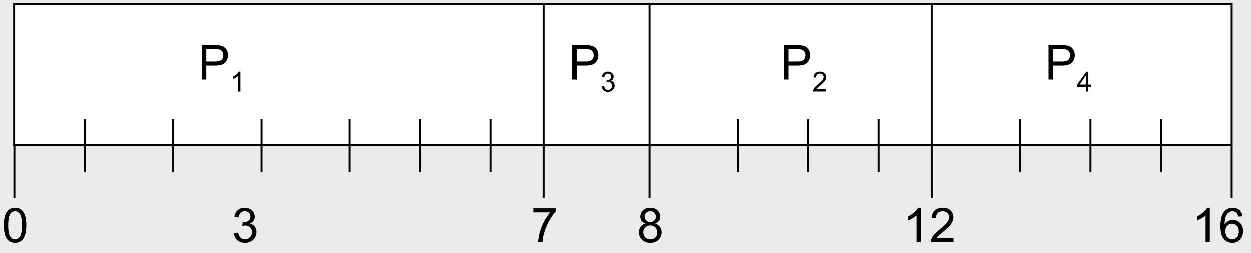
Shortest-Job-First (SJR)

- Associate to each process its execution time
- Two possibilities :
  - Non pre-emptive – When a CPU is allocated to a process, it cannot be pre-empted
  - Pre-emptive – if a new process comes with a shorter execution time that the running one, this last process is pre-empted
  - Alternate solution : Shortest-Remaining-Time-First (SRTF) algorithm
- SJF is optimal / mean response time

Example: Non Pre-emptive SJF

Process	Come in	Exec. time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (non pre-emptive)

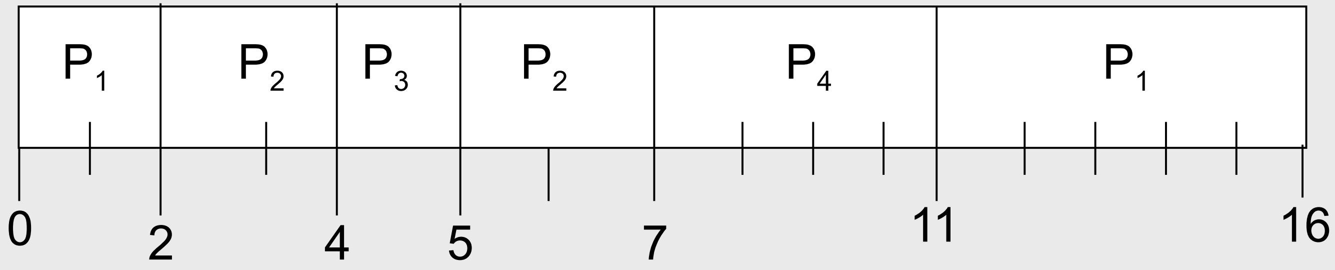


- Mean response time = (7 + 8 + 12 + 16)/4 = 10,75

Pre-emptive SJF (Shortest Remaining Time Next)

Process	Come in	Exec time.
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

- SJF (pre-emptive)

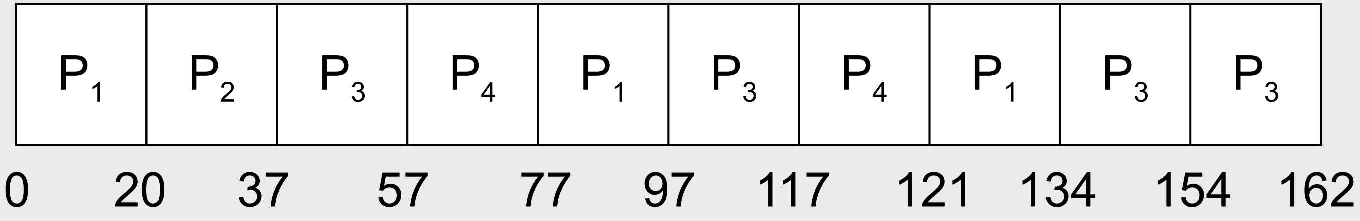


- Mean response time = (16 + 7 + 5 + 11)/4 = 8,25

Round Robin (Quantum = 20ms)

Process	Exec Time.
P_1	53
P_2	17
P_3	68
P_4	24

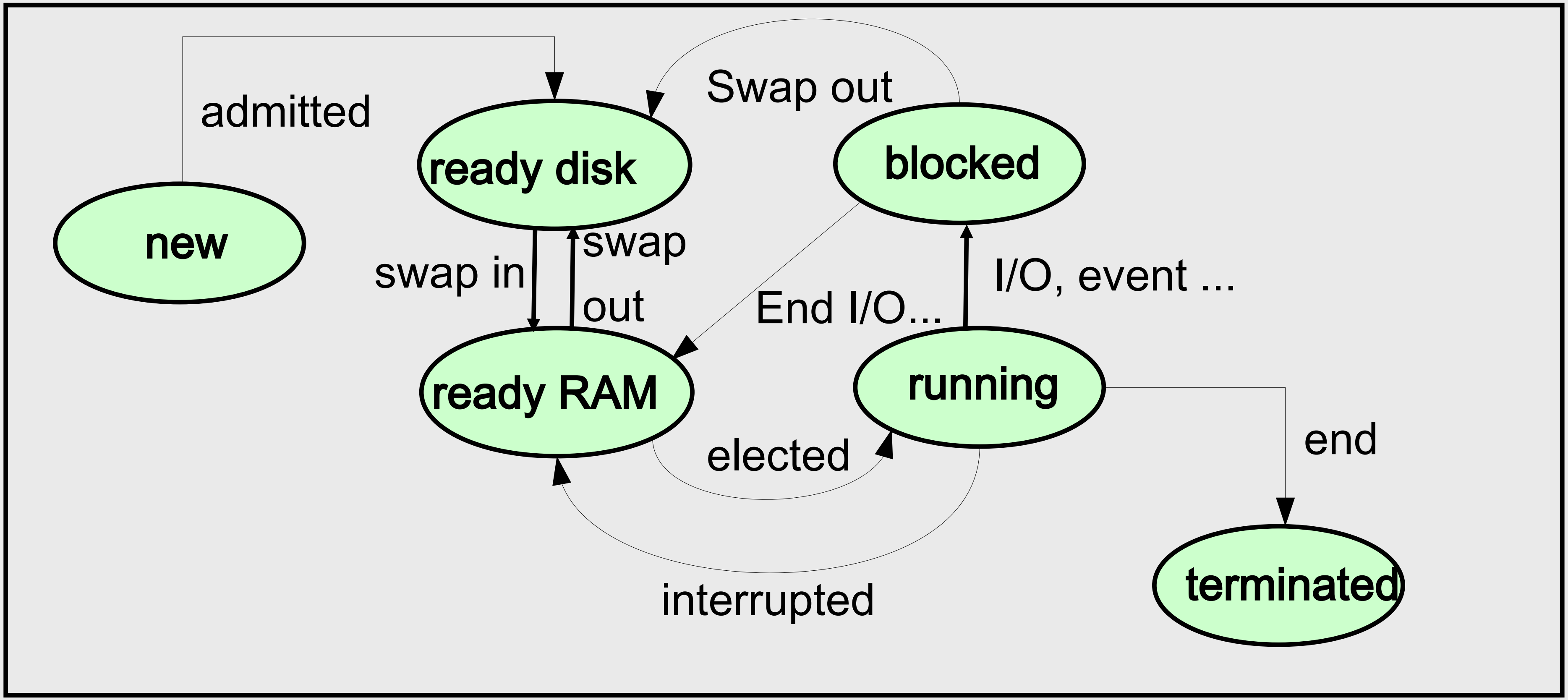
- **Efficiency and mean response worse than SJF**



Multiple level scheduling algorithm

- The set of ready processes too big to fit in memory
- Part of these processes are swapped out on the disk. This increase their activation time
- The elected process is always choosen from those that are in memory
- In parallel, another scheduling algorithm is used to manage the migration of ready process between disk and memory

Two level scheduling



Process SVC overview

- **int fork (void);**
  - Create new process that is exact copy of current one
  - Returns process ID of new process in “parent”
  - Returns 0 in “child”
- **int waitpid (int pid, ...);**
  - pid – process to wait for, or -1 for any
  - Returns process ID or -1 on error
- **Hierarchy of processes**
  - run the pstree -p command

Process SVC overview

- void exit (int status);
  - Current process ceases to exist
  - status shows up in waitpid (shifted)
  - By convention, status of 0 is success, non-zero error
- int kill (int pid, int sig);
  - Sends signal sig to process pid
  - SIGTERM most common value, kills process by default (but application can catch it for “cleanup”)
  - SIGKILL stronger, kills process always
- When a parent process terminates before its child, there are two options:
  - Cascading termination (VMS).
  - Re-parent the orphan (UNIX).

Process SVC overview

- **int execve (const char *prog, const char **argv, char **envp;)**
  - prog – full pathname of program to run
  - argv – argument vector that gets passed to main
  - envp – environment variables, e.g., PATH, HOME
- **Generally called through a wrapper functions**
  - int execvp (char *prog, char **argv);
    - Search PATH for prog, use current environment

Fork and Exec

- The fork system call creates a copy of the PCB
  - Open files and memory mapped files are thus similar
  - Open files are thus opened by both father and child. They should both close the files
  - The pages of many read only memory segments are shared (text, r/o data)
  - Many others are lazily copied (copy on write)
- The exec system call replaces the address space, the registers, the program counter by the one of the program to exec.
  - Open files are thus inherited

Why fork

- **Most calls to fork followed by execvp**
- **Real win is simplicity of interface**
  - **Tons of things you might want to do to child:**
  - **Yet fork requires no arguments at all**
  - **Without fork, require tons of different options**
  - **Example: Windows CreateProcess system call**

```
Bool CreateProcess(
      LPCTSTR lpApplicationName, //pointer to a name to executable module
      LPTSTR lpCommandLine, // pointer to a command line string
      LPSECURITYATTRIBUTES lpProcessAttributes, //process security attr
      LPSECURITYATTRIBUTES lpThreadAttributes, // thread security attr
      BOOL bInheritHandles, //creation flag
      DWORD dwCreationFlags, // creation flags
      LPVOID lpEnvironnement, // pointer to  new environment block
      LPCTSTR lpCurrentDirectory, // pointer to crrent directory name
      LPSTARTUPINFO lpStartupInfo, //pointer to STARTUPINFO
      LPPROCESSINFORMATION lpProcessInformaton // pointer to PROCESSINFORMATION );
```

Fork example

- **Process creation**
  - **Done by cloning an existing process**
    - **=>Duplicate the process**
  - **Fork() system call**
    - **Return 0 for the child process**
    - **Retour the child's pid to the father**
    - **Return -1 if error**

```
#include <unistd.h>

pid_t fork(void)
```

```
r = fork();
if (r==-1) ...       /* error */
else if (r==0) ... /* child's code */
else ...             /* father's code */
```

Fork example

- **How many processes are created ?**

```
fork();
fork();
fork();
```

```
for (i=0; i<3;i++){
   fork();
}
```

- What are the possible different traces

```
int i = 0;
switch((i=fork()) {
            case -1 : perror(“fork‟); break;
            case  0 : i++; printf(“child I :%d‟,i); break;
            default : printf(“father I :%d‟,i);

}
```

Exec example

- Reminder : main function definition
  - int main(int argc, char *argv[]);
- Execvp call
  - Replace the process's memory image
  - int execvp(const char *ref, const char *argv[])
    - ref : file name to load
    - argv : process parameters
  - execvp calls main(argc, argv) on the process to launch

Example

```
char  * argv[3];

argv[0] = “ls “;
argv[1] = “-al “;
argv[2] = 0;
execvp(“ls“, argv);
```

Father/child synchronization

- The father process wait for the terminaison of one of its child
  - pid_t wait(int *status)
    - The father wait for the terminaison of one of its child
      - pid_t : dead child's pid or -1 if no child
      - status : information on the child's death
  - pid_t waitpid(pid_t pid, int *status, int option)
    - Waint for a specific child's death
    - Option : non blocking … see man

Example wait

```
#include <sys/types.h>
#include <sys/wait.h>

main(){
  int spid, status;
  switch(spid = fork()){
             case -1 : perror(…); exit(-1);
             case 0  :  // child's code
                            break;
             default : // the father wait for this child's terminaison
                            if (waitpid(spid,&status,0)==-1) {perror(…);exit(-1);}
                            …
  }
}
```

Example : minishell

```
pid t pid;
char **av;
void doexec() {
  execvp(av[0], av);
  perror(av[0]);
  exit(1);
}
/* ... main loop: */
for (;;) {
  parse_next_line_of_input(&av, stdin);
  switch (pid = fork()) {
   case -1: perror("fork"); break;
   case 0:
   doexec();
   default:
   waitpid(pid, NULL, 0);
   break;
   }
}
```

I/O redirection

- **All file is adressed through a descriptor**
  - **0, 1 et 2 correspond to standard input, standard output, and standard error**
  - **The file descriptor number is return by the open system call**

I/O redirection

- Basic operation
  - int open(const char *ref, int mode);
    - O_RDONLY, O_WRONLY, O_RDWR …
  - int creat(const char *ref,mode_t droit);
  - int close(int desc)
  - ssize_t read(int desc, void *ptr,size_t nb_octet);
  - ssize_t write(int desc, void *ptr, size_t nb_octet);

I/O redirection

- **Descriptor duplication**
  - **dup(int desc); dup2(int desc_src, int desc_dest);**
  - **Used to redirect standard I/O**

```
#include <stdio.h>
#include <unistd.h>
int f;
/* redirect std input to the file */

…
close(STDIN_FILENO); // close std input
dup(f);                    // dupliquate f on the first free descriptor (i.e. 0)
close(f);      // free f
…
```

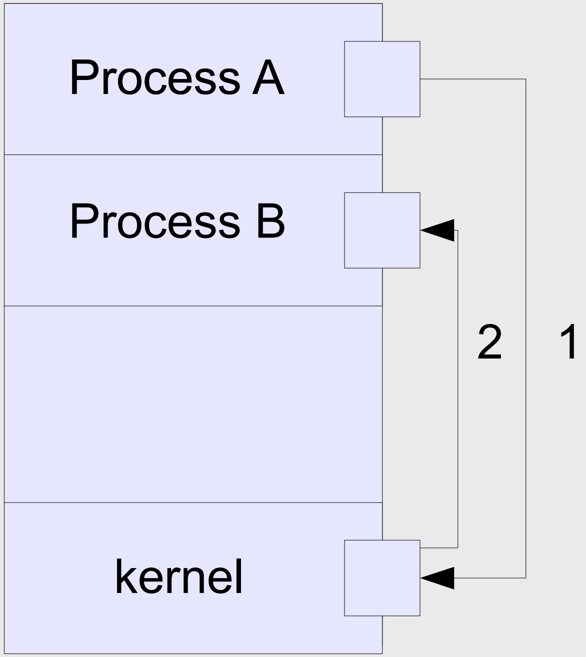
```

dup2(f,STDIN_FILENO);
close(f);
```

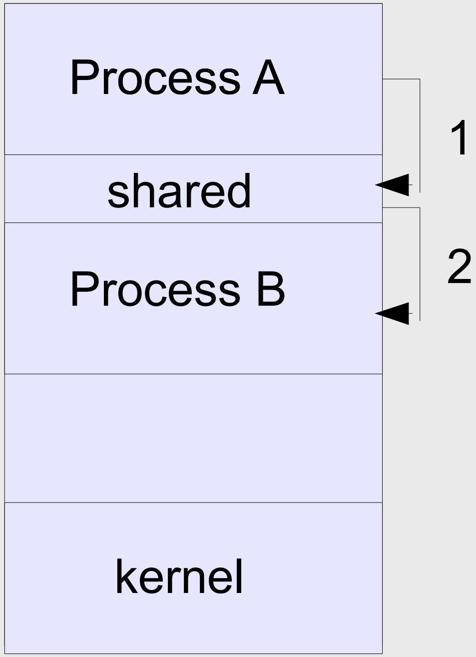
Cooperation between processes

- Independent process cannot affect or be affected by the execution of another process
- Cooperating process can affect or be affected by the execution of another process. Advantages:
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Process Interaction

- **How can processes interact in real time?**
  - **(1) Through files but it’s not really “real time”.**
  - **(2) Through asynchronous signals or alerts but again, it’s not really**
  - **“real time”.**
  - **(3) By sharing a region of physical memory**
  - **(4) By passing messages through the kernel**





Asynchronous notification (Signal)

- **A process may send a SIGINT, SIGSTOP, SIGTERM, SIGKILL signal to CTRL-C, suspend (CTRL-Z), terminate or kill a process using the kill function:**

  `int kill (int pid, int sig);`

  - A lot of signals …
  - Some signals cannot be blocked (SIGSTOP and SIGKILL)
- **Upon reception of a signal, a given handler is called. This handler can be obtained and modified using the signal function:**
  - `typedef void (*sighandler t)(int); //  handler`
  - `sighandler t signal(int signum, sighandler t handler); // set a handler`

Example

```
void handler(int signal_num) {
    printf("Signal %d => ", signal_num);
    switch (signal_num) {
    case SIGTSTP:
        printf("pause");
        break;
    case SIGINT:
    case SIGTERM:
        printf("End of the program");
        exit(EXIT_SUCCESS);
        break;
    }
}
```

```
int main(void) {
  signal(SIGTSTP, handler);
  /* if control-Z */
  signal(SIGINT, handler);
  /* if control-C */
  signal(SIGTERM, handler);
  /* if kill processus */
  while (1) {
    sleep(DELAI);
    printf(".");
    fflush(stdout);
   }
printf("fin");
exit(EXIT_SUCCESS);
}
```

- **Signal handling is vulnerable to race conditions:** another signal (even of the same type) can be delivered to the process during execution of the signal handling routine.
- **The sigprocmask() call can be used to block and unblock delivery of signals.**

Shared memory segment

- A process can create a shared memory segment using:
  - int shmget(key t key, size t size, int shmflg);
  - The returned value identifies the segment and is called the shmid
  - The key is used so that process indeed get the same segment.
- The original owner of a shared memory segment can assign ownership to another user with shmctl().
  - It can also revoke this assignment.
- Once created, a shared segment should be attached to a process address space using
  - void *shmat(int shmid, const void *shmaddr, int shmflg);
- It can be detached using int shmdt(const void *shmaddr);
- Can also be done with the mmap function
- Example

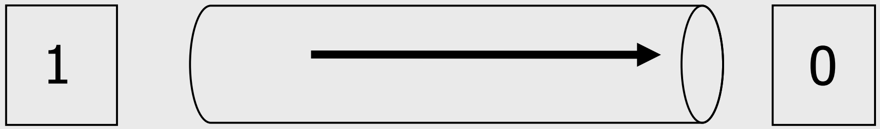
Example

```
char c;
int shmid;
key t key;
char *shm, *s;
key = 5678;
/* Create the segment */
if ((shmid = shmget(key, SHMSZ,
IPC_CREAT | 0666)) < 0) {
perror("shmget");
exit(1);
}
/* Attach the segment */
if ((shm = shmat(shmid, NULL, 0)) ==
(char *) -1) {
perror("shmat");
exit(1);
}
```

```
int shmid;
key t key;
char *shm, *s;
key = 5678;
/* Locate the segment */
if ((shmid = shmget(key, SHMSZ, 0666))< 0) {
perror("shmget");
exit(1);
}
/* Attach the segment */
if ((shm = shmat(shmid, NULL, 0)) ==
(char *) -1) {
perror("shmat");
exit(1);
}
```

Pipe

- Communication mechanism between processes
  - Fifo structure
  - Limited capacity
  - Producer/consumer synchronization



pipe

Pipes

- int pipe (int fds[2]);
  - Returns two file descriptors in fds[0] and fds[1]
  - Writes to fds[1] will be read on fds[0]
  - When last copy of fds[1] closed, fds[0] will return EOF
  - Returns 0 on success, -1 on error
- Operations on pipes
  - read/write/close – as with files
  - When fds[1] closed, read(fds[0]) returns 0 bytes
  - When fds[0] closed, write(fds[1]):
  - Kills process with SIGPIPE, or if blocked
  - Fails with EPIPE

Example

```
void doexec (void) {
    int pipefds[2];

        pipe (pipefds);
        switch (fork ()) {
             Case -1:         perror ("fork"); exit (1);
             case 0:
                              dup2 (pipefds[1], 1);
                              close (pipefds[0]); close (pipefds[1]);
                              execvp(...);
                              break;
            default:
                              dup2 (pipefds[0], 0);
                              close (pipefds[0]); close (pipefds[1]);
                              break;
       }
/* ... */
}
```

IPC

- Mechanism for processes to communicate and to synchronize their actions
- Message system for processes to communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - send(message) message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to:
  - establish a communication link between them
  - exchange messages via send/receive
  - Implementation of communication link shared memory, hardware bus ...

IPC

- How are links established?
- Can a link be associated with more than two processes?
- How many links can there be between every pair of comunicating processes?
- What is the capacity of a link?
- Is the size of a message that the link can accommodatefixed or variable?
- Is a link unidirectional or bi-directional?
- Detailled during practical sessions

Direct link

- Processes must name each other explicitly:
  - send (P, message) send a message to process P
  - receive(Q, message) receive a message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
- Between each pair there exists exactly one link.
- The link may be unidirectional, but is usually bi-directional

Indirect communication

- Messages are directed and received from mailboxes (also referred to as ports)
- Each mailbox has a unique id
- Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional
- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox
  - send(A, message) send a message to mailbox A
  - receive(A, message) receive a message from mailbox A

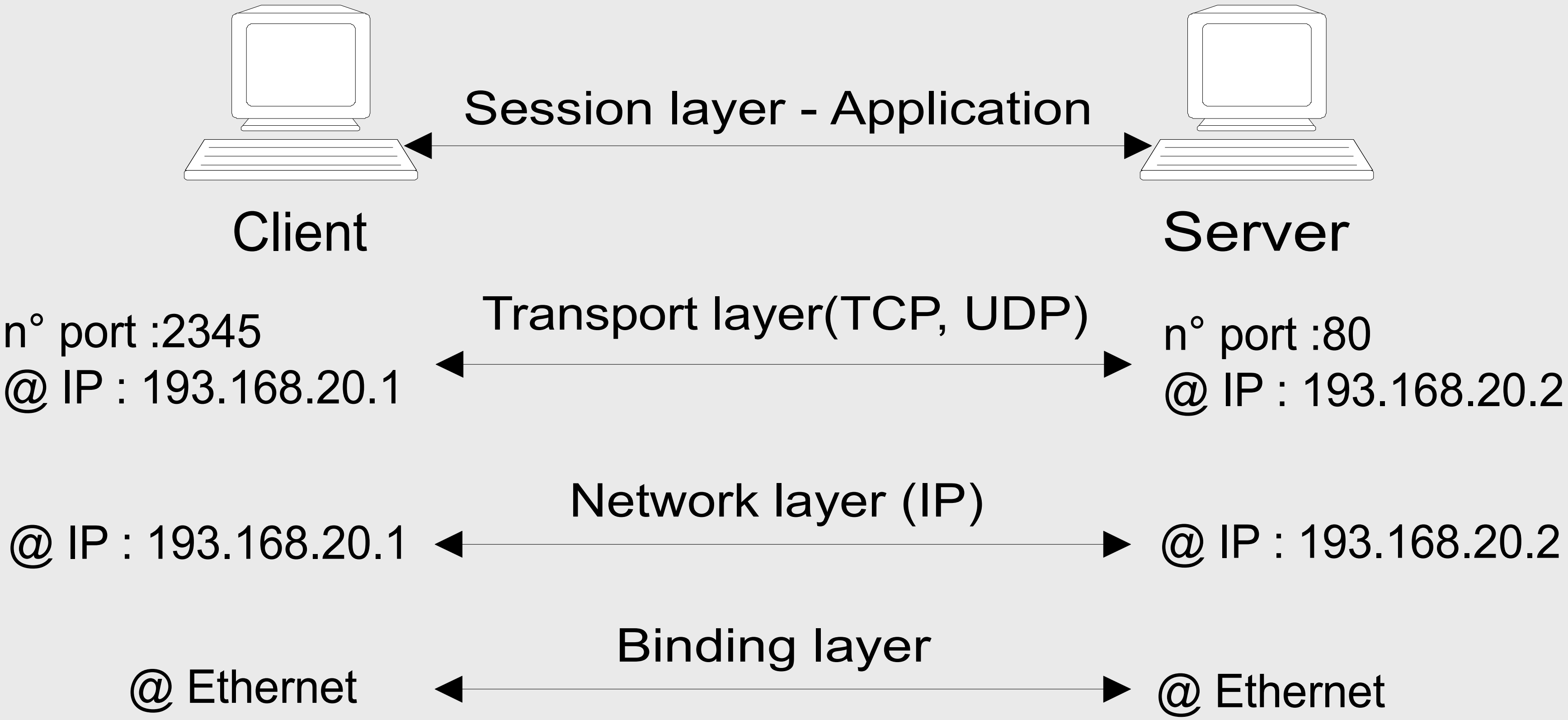
Socket

- A socket is defined as an endpoint for communication
- Basic message passing API
- Identified by an IP address and port
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets and is bidirectionnal

Socket

- Connected mode (TCP) :
  - Communication problem are managed automatically
    - Message lost, duplication, delivery order
  - Connexion and reliability overhead
- Non connected mode (UDP) :
  - Consume less system's resources
  - Message can be lost or duplicated
  - Manage error by hand

Socket layer



Session layer - Application

Client

Server

n° port :2345
@ IP : 193.168.20.1

Transport layer(TCP, UDP)

n° port :80
@ IP : 193.168.20.2

@ IP : 193.168.20.1

Network layer (IP)

@ IP : 193.168.20.2

@ Ethernet

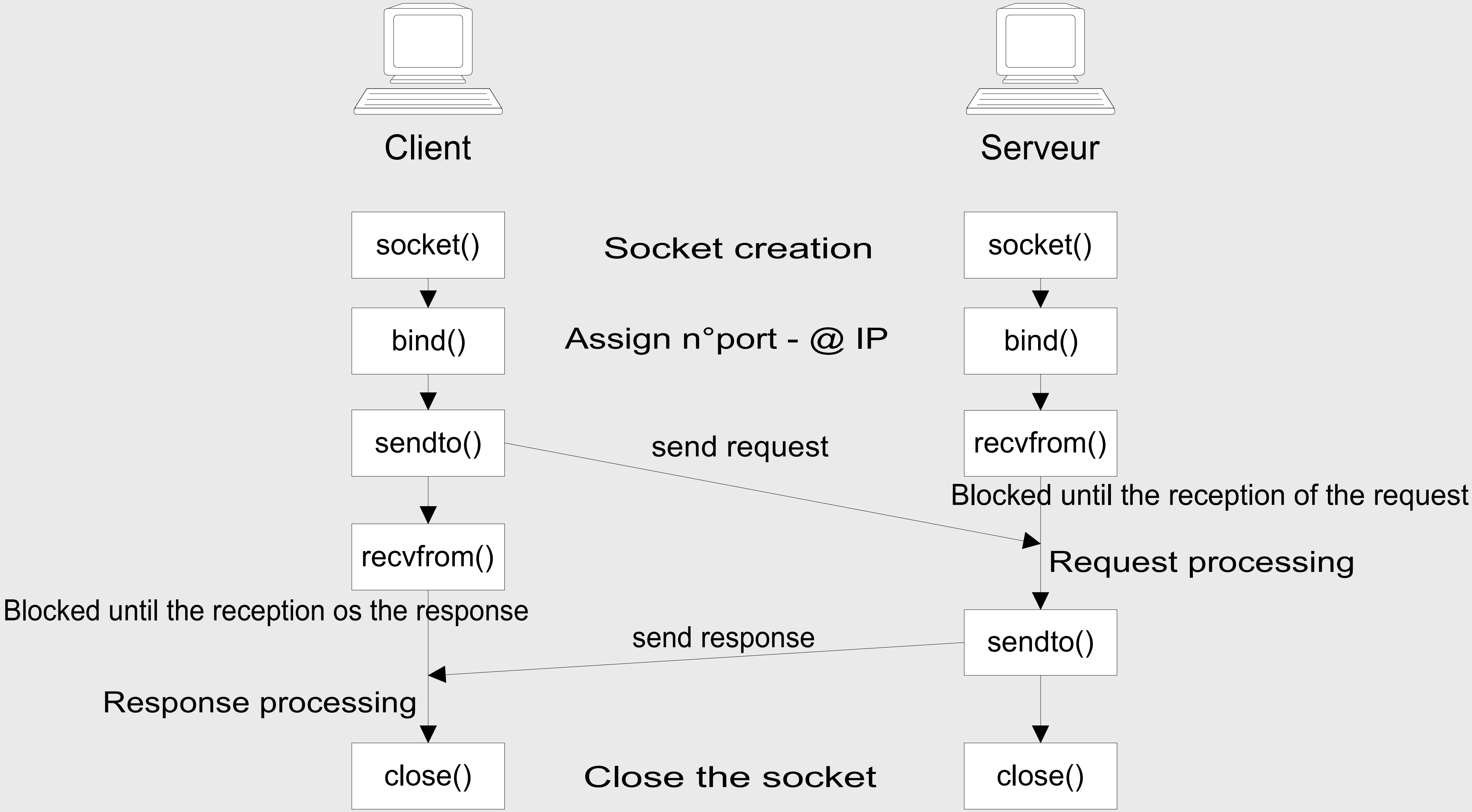
Binding layer

@ Ethernet

The socket API

- Socket creation: socket(..., protocol)
- Open the dialogue :
  - client : connect(...)
  - serveur : bind(..), listen(...), accept(...)
- Data transfert :
  - Connected mode : read(...), write(...), send(...), recv(...)
  - Not connected mode : sendto(...), recvfrom(...), sendmsg(...), recvmsg(...)
- Close the dialogue :
  - close(...), shutdown(...)

Client/Server in not connected mode



Client

Serveur

socket()

Socket creation

socket()

bind()

Assign n°port - @ IP

bind()

sendto()

send request

recvfrom()

Blocked until the reception of the request

recvfrom()

Request processing

Blocked until the reception os the response

send response

sendto()

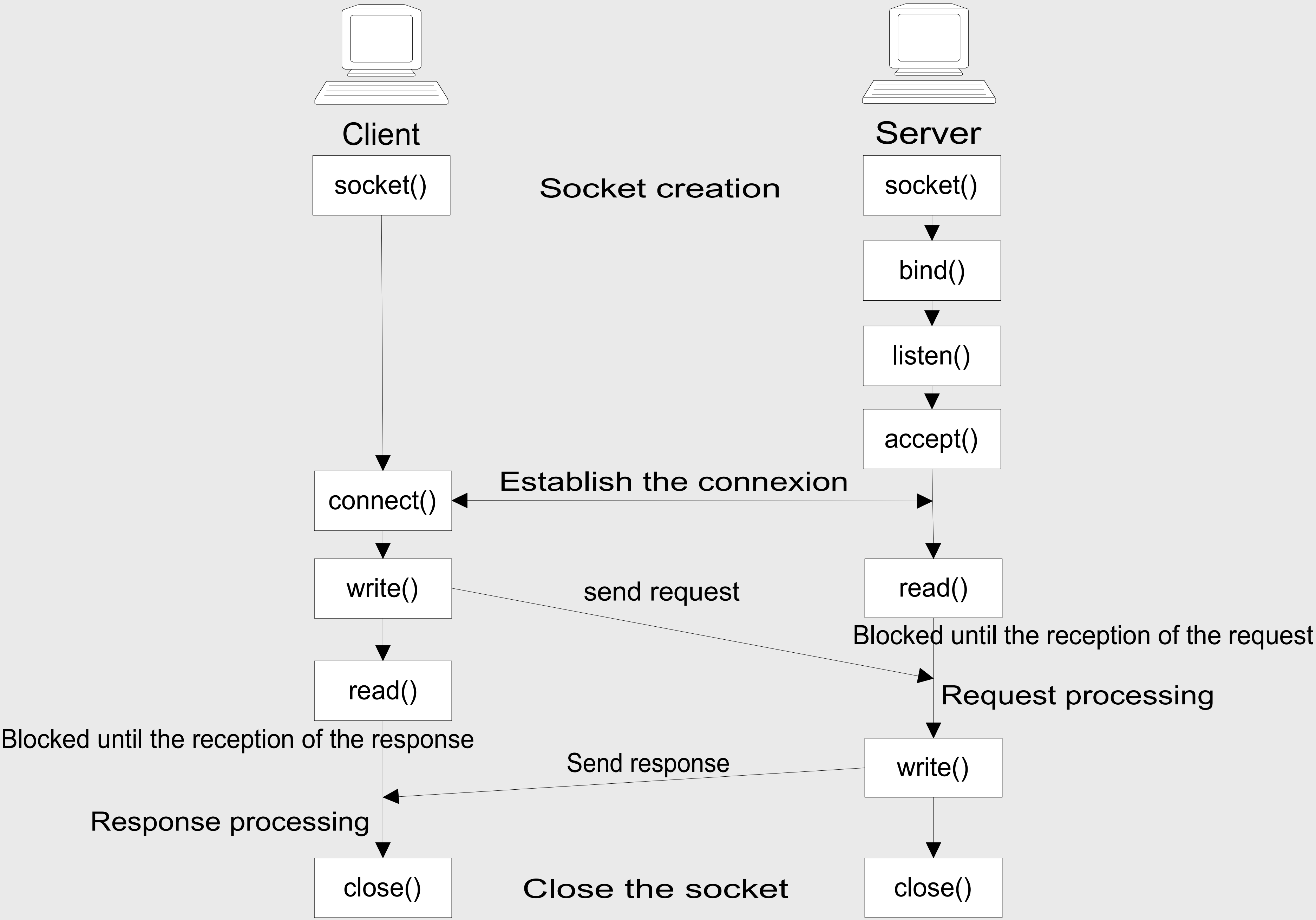
Response processing

close()

Close the socket

close()

Client/Server in connected mode



Client

Server

socket()

Socket creation

socket()

bind()

listen()

accept()

connect()

Establish the connexion

write()

send request

read()

Blocked until the reception of the request

read()

Request processing

Blocked until the reception of the response

Send response

write()

Response processing

close()

Close the socket

close()

Server model

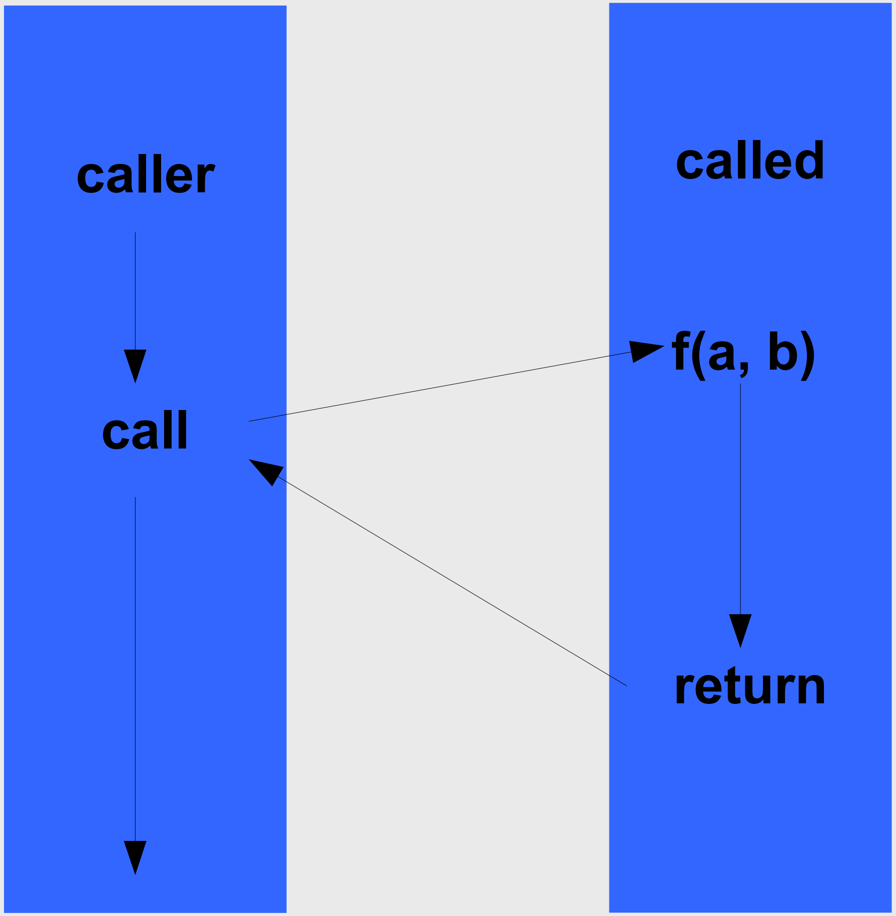
- Simple
- Master/slave
  - On demand creation of processes/threads
  - Pool of processes/threads
- Duplicated
  - Request load balancer
  - Primary/secondary replication
  - Active replication

Publish/Subscribe (1)

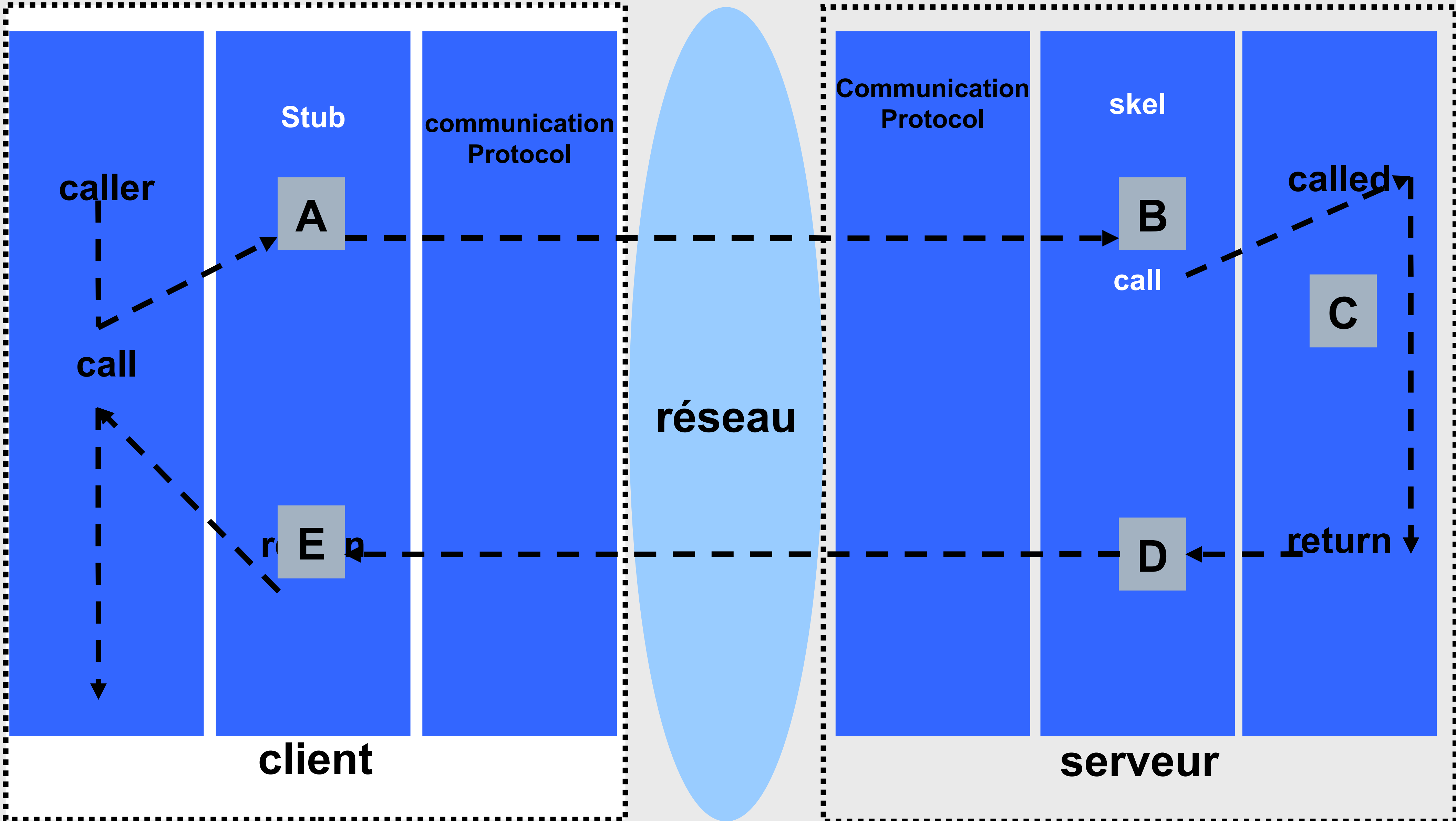
- Anonymous sender/receiver
  - Sender send a message (subject-based or content-based)
  - Receiver subscribe (to a subject or to a content)
- Communication 1-N
  - Multiple receivers can subscribe

Remote Procedure Call

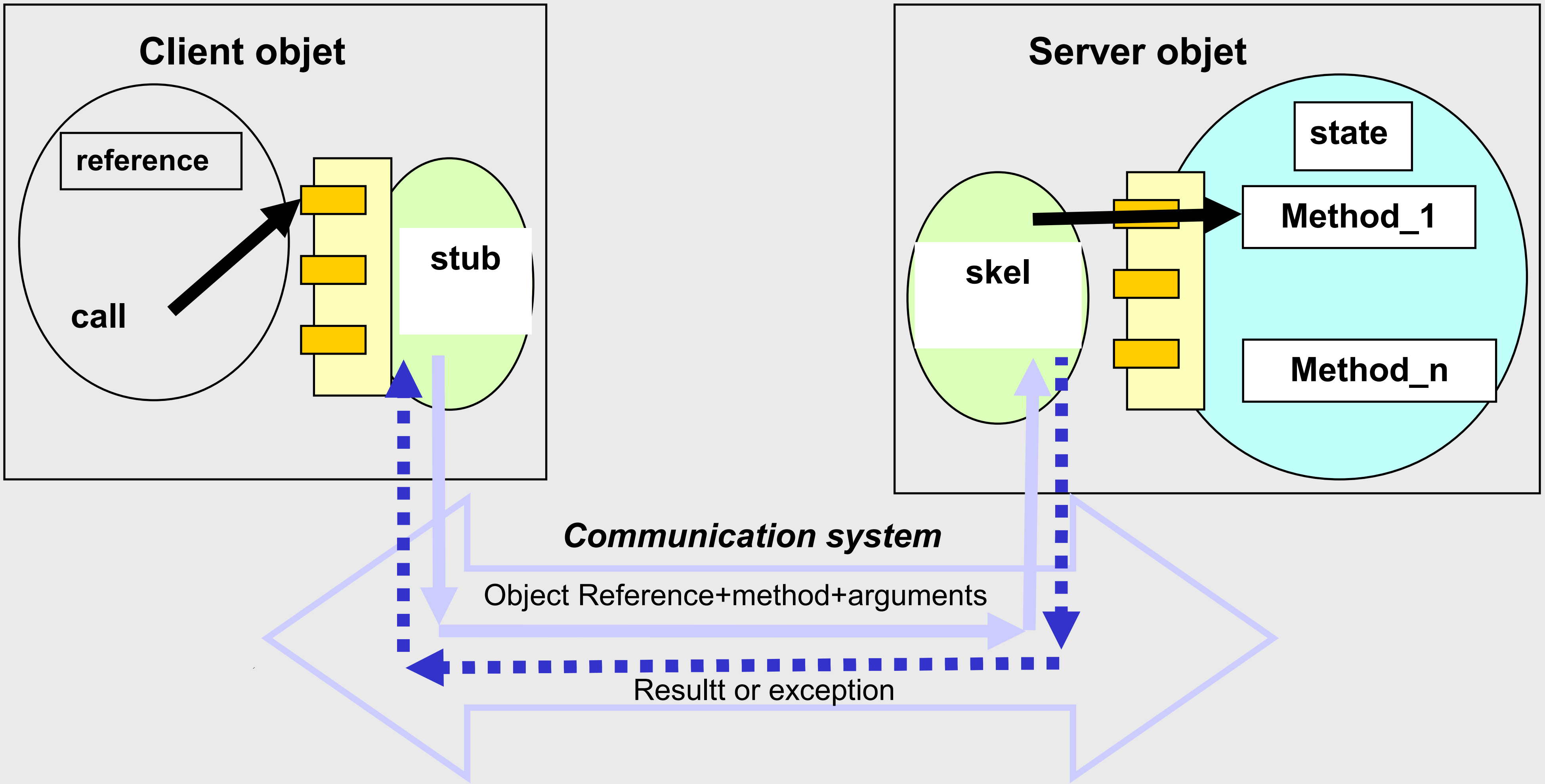
- **Allow to call procedure in other address space**
- **Easy to program : RPC call looks like local procedure call**



RPC [Birrel & Nelson 84]



Remote Method Invocation (RMI)



Client objet

reference

call

stub

Server objet

state

Method_1

skel

Method_n

Communication system

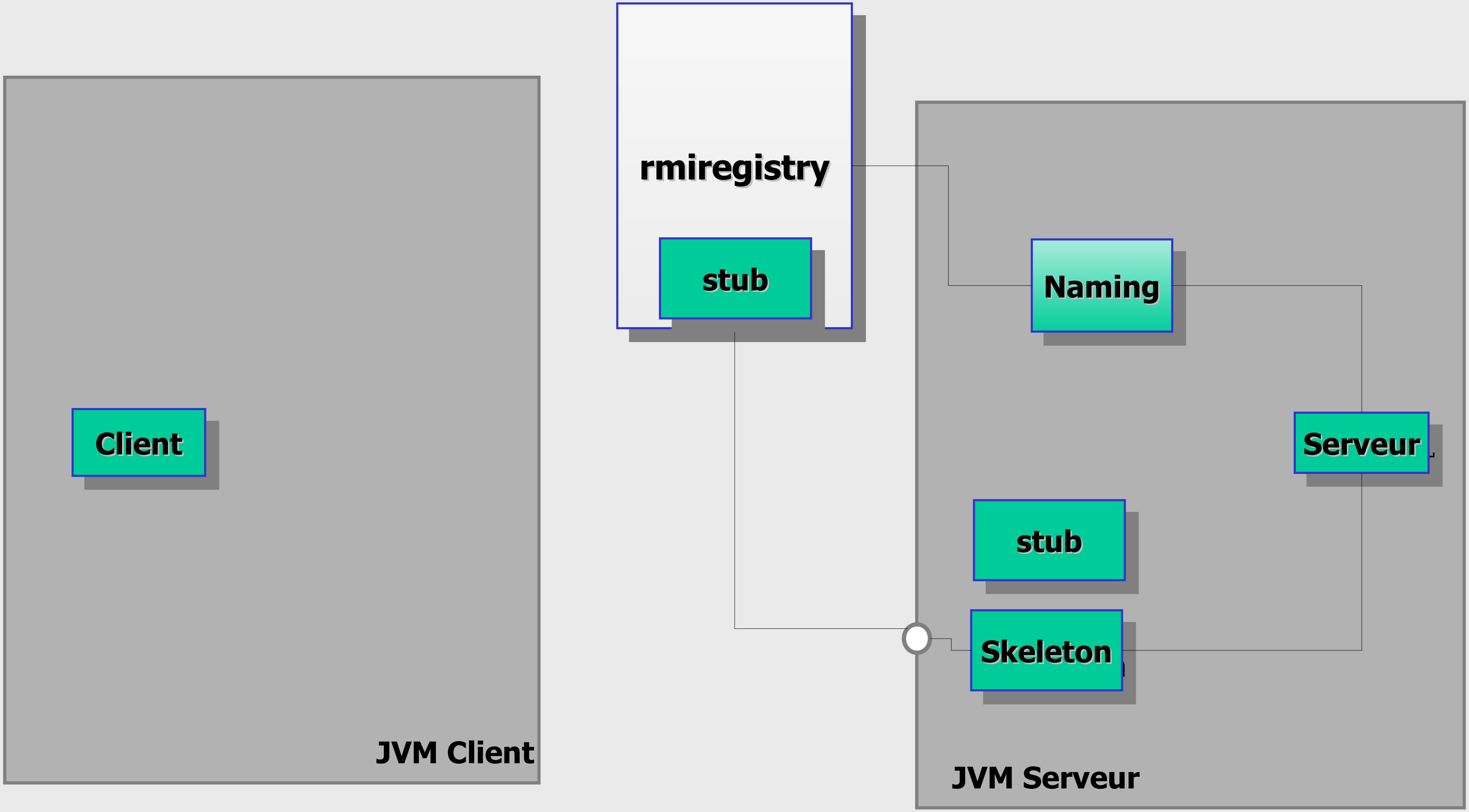
Object Reference+method+arguments

Resultt or exception

Java RMI
Server side

- 0 – When the object is created, a stub and a skeleton (with communication) are created on server side
- 1 – The server object register through the RMI's registry (method *rebind of the naming class*). The object stub is registered in the registry.
- 2 – The registry is ready to provide remote reference to object server

Java RMI Architecture



Java RMI
Client side

- 4 – The client object use the Naming class to locate the server object in the registry (lookup method)
- 5 - The registry provides the stub to the server object
- 6 – install the stub object and return its reference to the client
- 7 – The client calls the remote object through the stub

Java RMI Architecture