Inter-Process Communication

Heechul Yun

Disclaimer: some slides are adopted from the book authors' slides with permission

Inter-Process Communication (IPC)

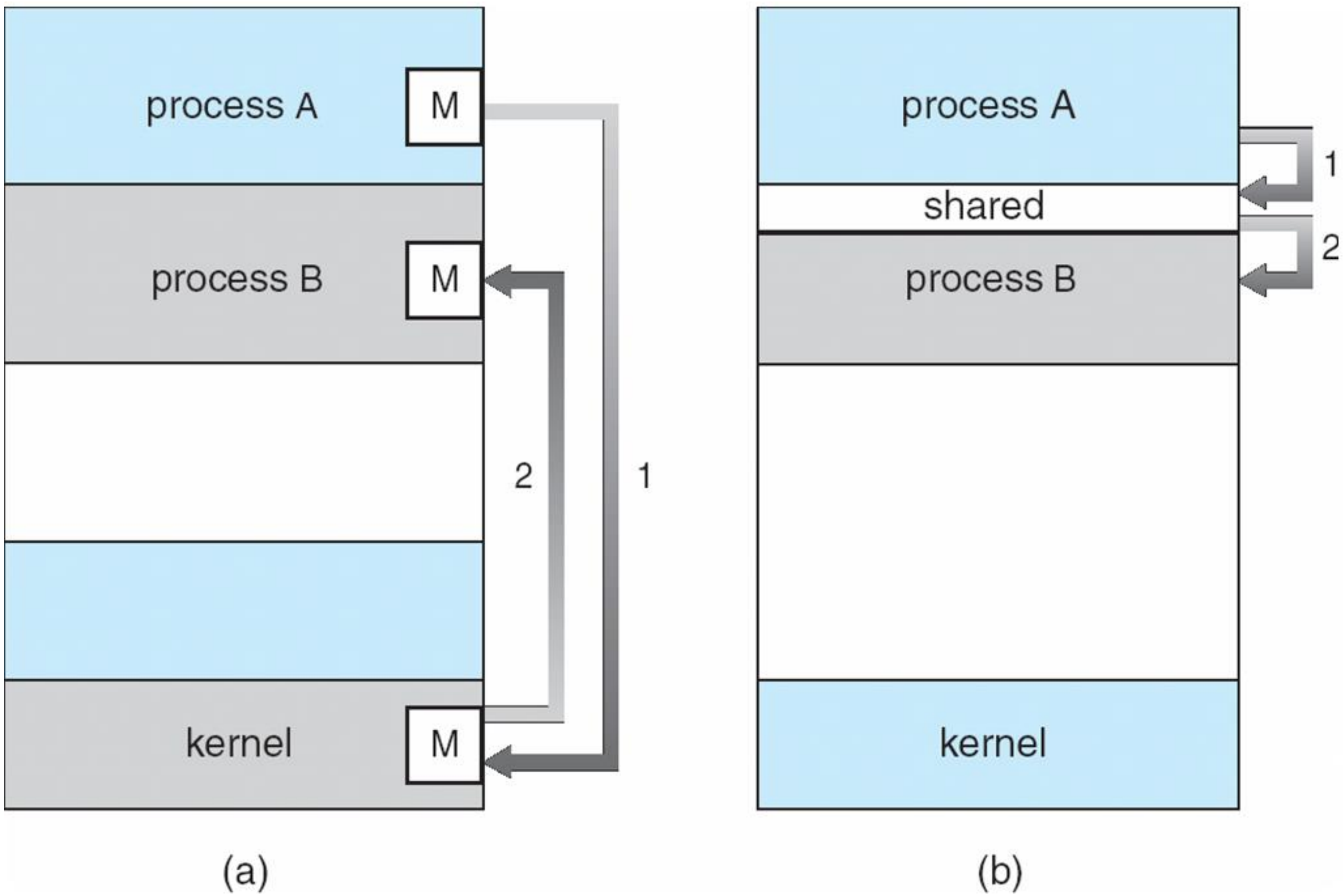
- What is it?
  - Communication among processes

- Why needed?
  - Information sharing
  - Modularity
  - Speedup

Chrome Browser

- Multi-process architecture
- Each tab is a **separate** process
  - Why?
  - How to communicate among the processes?

Models of IPC



message passing shared memory

Models of IPC

- Shared memory
  - share a region of memory between co-operating processes
  - read or write to the shared memory region

  ++ **fast** communication

  -- **synchronization** is very **difficult**

- Message passing
  - exchange messages (*send* and *receive*)
  - typically involves data copies (to/from buffer)

  ++ **synchronization** is **easier**

  -- **slower** communication

KU THE UNIVERSITY OF KANSAS

Interprocess Communication in Unix (Linux)

- **Pipe**
- **FIFO**
- **Shared memory**
- **Socket**
- Message queue
- …

Pipes

- Most basic form of IPC on all Unix systems
  - Your shell uses this a lot (and your 1st programming project too)

```
ls | more
```

- Characteristics
  - Unix pipes only allow **unidirectional** communication
  - Communication **between parent-child**
  - Processes must be in the **same OS**
  - Pipes exist only until the processes exist
  - Data can only be collected in FIFO order

IPC Example Using Pipes

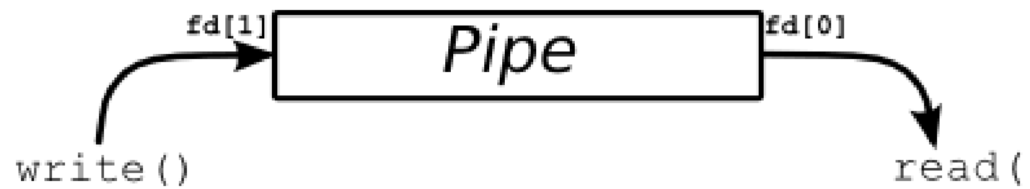
```
main()
{
  char *s, buf[1024];
  int fds[2];
  s  = “Hello World\n";

  /* create a pipe */
  pipe(fds);

  /* create a new process using fork */
  if (fork() == 0) {

    /* child process. All file descriptors, including
       pipe are inherited, and copied.*/
    write(fds[1], s, strlen(s));
    exit(0);
  }

  /* parent process */
  read(fds[0], buf, strlen(s));
  write(1, buf, strlen(s));
}
```



(*) Img. source: http://beej.us/guide/bgipc/output/html/multipage/pipes.html

Pipes Used in Unix Shells

- Pipes commonly used in most Unix shells
  - output of one command is input to the next command
  - example: `ls| more`
- How does the shell realize this command?
  - create a pipe
  - create a process to run `ls`
  - create a process to run `more`
  - the standard output of the process to run `ls` is redirected to a pipe streaming to the process to run `more`
  - the standard input of the process to run `more` is redirected to be the pipe from the process running `ls`

Named Pipes (FIFO)

- Pipe with a name !
  - More powerful than anonymous pipes
  - no parent-sibling relationship required
  - FIFOs exists even after creating process is terminated
- Characteristics of FIFOs
  - appear as typical *files*
  - communicating process must reside on the same machine

KU THE UNIVERSITY OF KANSAS

Example: Producer

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // create FIFO file
  fd = open(FIFO_NAME, O_WRONLY); // open FIFO for writing

  printf("Enter text to write in the FIFO file: ");
  fgets(str, MAX_LENGTH, stdin);
  while(!(feof(stdin))){
    if ((num = write(fd, str, strlen(str))) == -1)
      perror("write");
    else
      printf("producer: wrote %d bytes\n", num);
    fgets(str, MAX_LENGTH, stdin);
  }
}
```

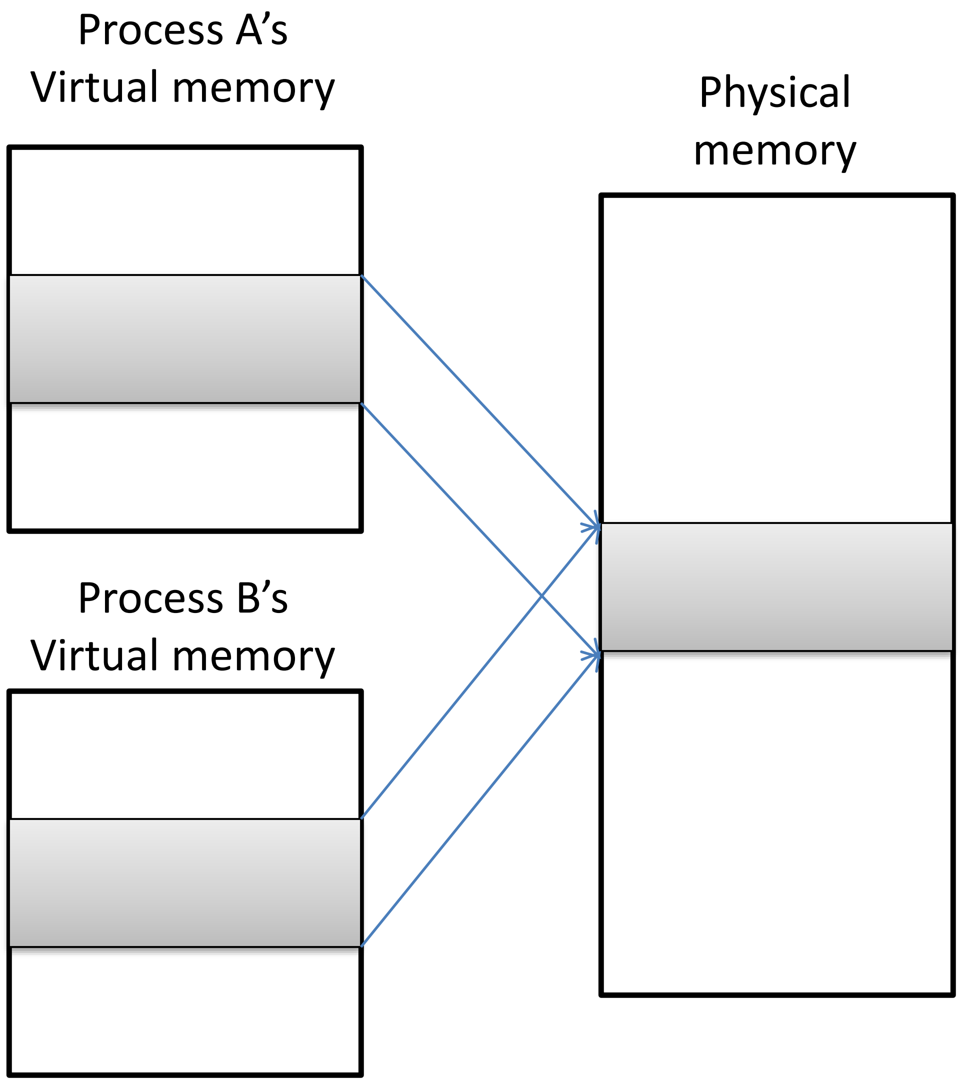
Example: Consumer

```
main()
{
  char str[MAX_LENGTH];
  int num, fd;

  mkfifo(FIFO_NAME, 0666); // make fifo, if not already present
  fd = open(FIFO_NAME, O_RDONLY); // open fifo for reading

  do{
    if((num = read(fd, str, MAX_LENGTH)) == -1)
      perror("read");
    else{
      str[num] = '\0';
      printf("consumer: read %d bytes\n", num);
      printf("%s", str);
    }
  }while(num > 0);
}
```

Shared Memory



Shared Memory

- Kernel is not involved in data transfer
  - No need to copy data to/from the kernel
    - Very fast IPC
  - Pipes, in contrast, need to
    - Send: copy from user to kernel
    - Recv: copy from kernel to user
  - BUT, you have to *synchronize*
    - Will discuss in the next week

POSIX Shared Memory

- Sharing between unrelated processes
- APIs
  - shm_open()
    - Open or create a shared memory object
  - ftruncate()
    - Set the size of a shared memory object
  - mmap()
    - Map the shared memory object into the caller's address space

Example: Producer

```
$ ./writer /shm-name "Hello"


int main(int argc, char *argv[])
{
  char str[MAX_LENGTH];
  int fd;
  size_t len;

  fd = shm_open(argv[1], O_CREAT | O_RDWR, S_IRWXU | S_IRWXG);
  len = strlen(argv[2]);
  ftruncate(fd, len);
  addr = mmap(NULL, len, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
  close(fd);

  memcpy(addr, argv[2], len);
  return 0;
}
```

http://www.ittc.ku.edu/~heechul/courses/eecs678/shm-writer.c

Example: Consumer

```
$ ./reader /shm-name


int main(int argc, char *argv[])
{
  char *addr;
  int fd;
  struct stat sb;

  fd = shm_open(argv[1], O_RDWR, 0);
  fstat(fd, &sb);
  addr = mmap(NULL, sb.st_size, PROT_READ, MAP_SHARED, fd, 0);
  close(fd);

  printf("%s\n", addr);
  return 0;
}
```

http://www.ittc.ku.edu/~heechul/courses/eecs678/shm-reader.c

Sockets

- Sockets
  - two-way communication pipe
  - Backbone of your internet services
- Unix Domain Sockets
  - communication between processes on the same Unix system
  - special file in the file system
- Client/Server
  - client sending requests for information, processing
  - server waiting for user requests
- Socket communication modes
  - connection-based, TCP
  - connection-less, UDP

Example: Server

```
int main(int argc, char *argv[])
{
    int listenfd = 0, connfd = 0;
    struct sockaddr_in serv_addr;
    char sendBuff[1025];
    time_t ticks;

    listenfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, '0', sizeof(serv_addr));
    memset(sendBuff, '0', sizeof(sendBuff));

    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(5000);

    bind(listenfd, (struct sockaddr*)&serv_addr, sizeof(serv_addr));
    listen(listenfd, 10);

    while(1)
    {
        connfd = accept(listenfd, (struct sockaddr*)NULL, NULL);
        snprintf(sendBuff, "Hello. I'm your server.");
        write(connfd, sendBuff, strlen(sendBuff));
        close(connfd);
     }
}
```

Example: Client

```
int main(int argc, char *argv[])
{
    int sockfd = 0, n = 0;
    char recvBuff[1024];
    struct sockaddr_in serv_addr;

    sockfd = socket(AF_INET, SOCK_STREAM, 0);
    memset(&serv_addr, '0', sizeof(serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(5000);

    inet_pton(AF_INET, argv[1], &serv_addr.sin_addr);
    connect(sockfd, (struct sockaddr *)&serv_addr, sizeof(serv_addr));

    while ( (n = read(sockfd, recvBuff, sizeof(recvBuff)-1)) > 0)
    {
        recvBuff[n] = 0;
        printf("%s\n" recvBuff);
    }
    return 0;
}
```

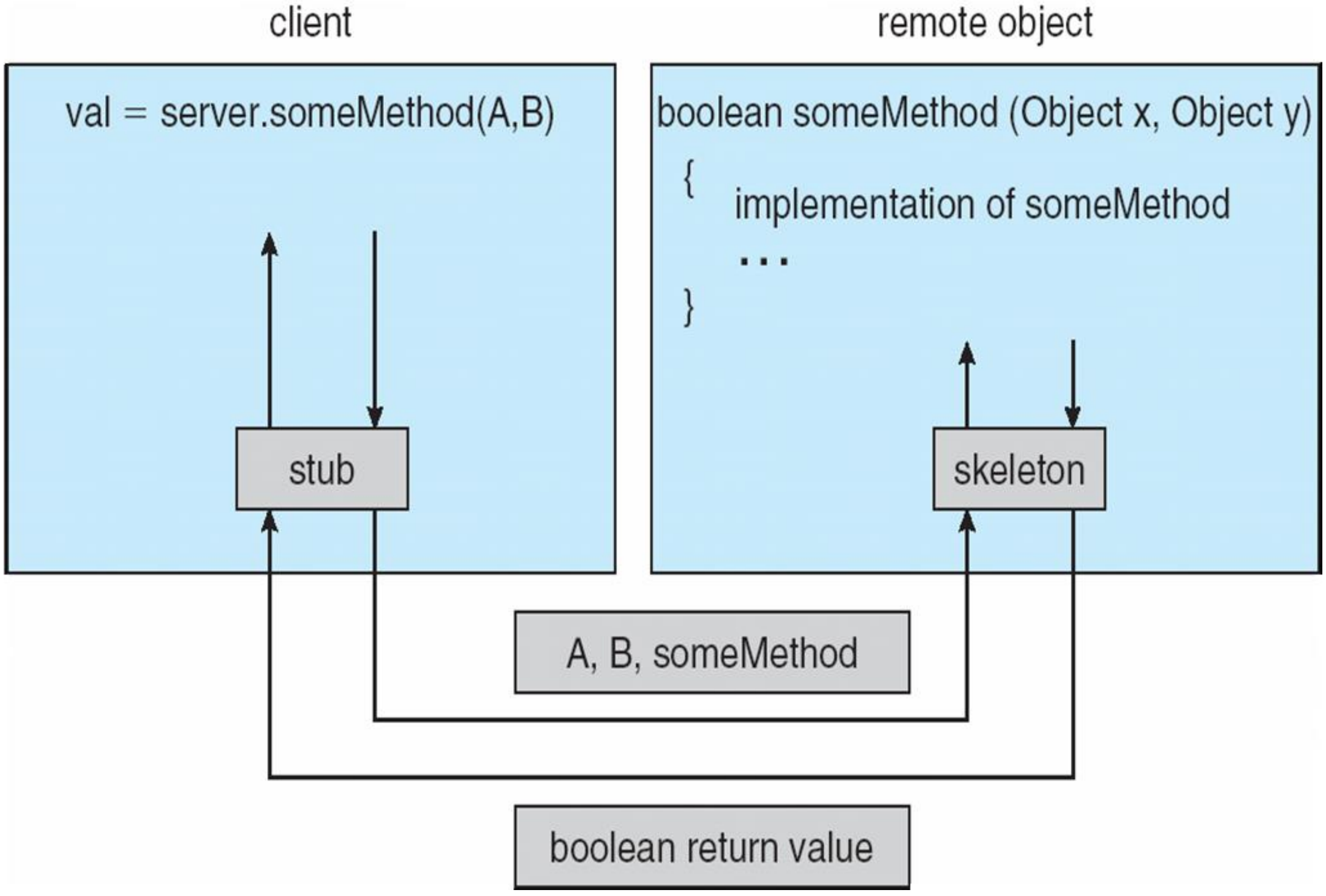
$ ./client 127.0.0.1

Hello. I’m your server.

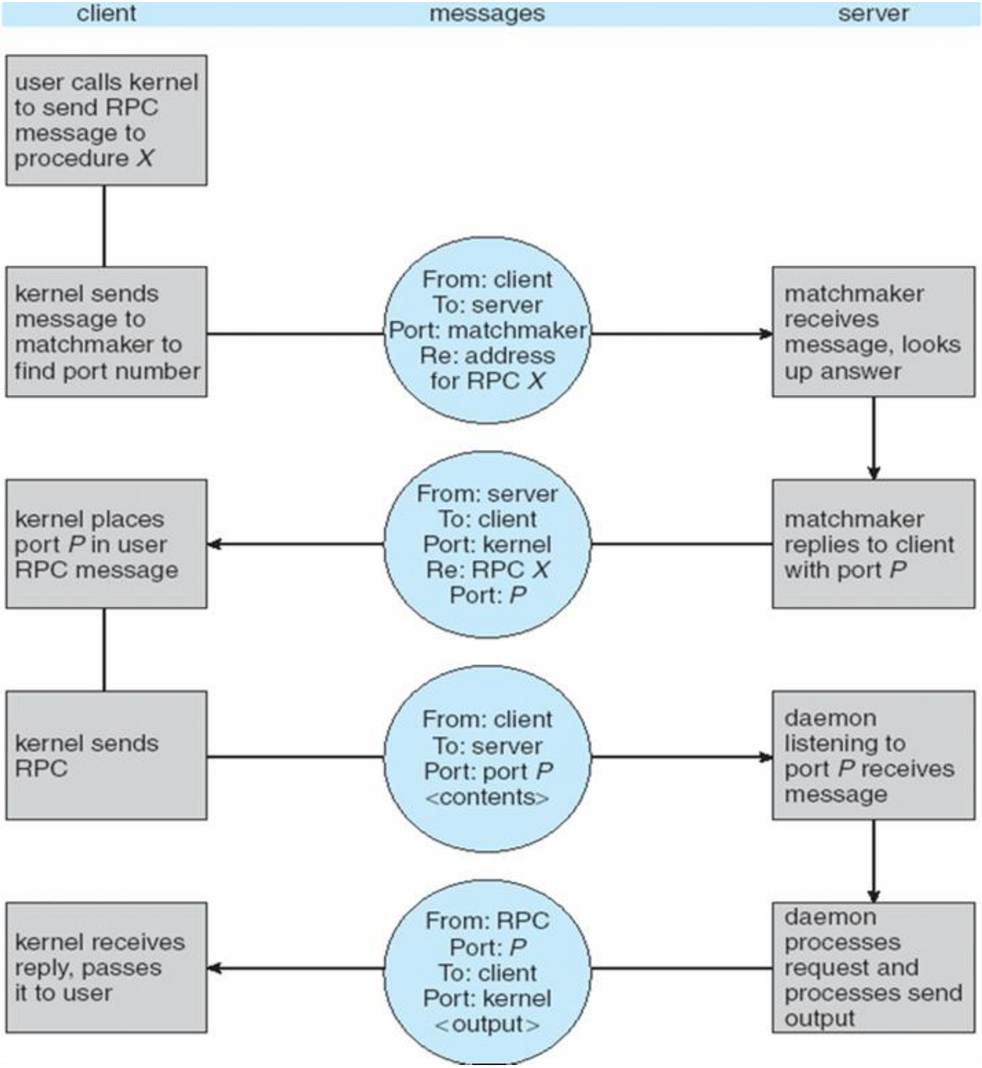
Remote Procedure Calls

- Remote procedure call (RPC) abstracts subroutine calls between processes on networked systems
  - subroutine executes in another address space
  - uses message passing communication model
  - messages are well-structured
  - RPC daemon on the server handles the remote calls
- Client-side *stub*
  - proxy for the actual procedure on the server
  - responsible for locating correct port on the server
  - responsible for *marshalling* the procedure parameters
- Server-side stub
  - receives the message
  - unpacks the marshalled parameters
  - performs the procedure on the server, returns result

Marshalling Parameters



Execution of RPC



Quiz

- A process produces 100MB data in memory. You want to share the data with two other processes so that each of which can access half the data (50MB each). What IPC mechanism will you use and why?