Operating Systems
Lecture

Inter-process Communication

Prof. Mengwei Xu

Goals for Today

- Why Inter-process Communication (IPC)?
- Types of IPCs
  - Pipes
  - Message Queues
  - Shared memory
  - Remote Procedure Calls (RPC)
  - Others..

Goals for Today

- Why Inter-process Communication (IPC)?
- Types of IPCs
  - Pipes
  - Message Queues
  - Shared memory
  - Remote Procedure Calls (RPC)
  - Others..

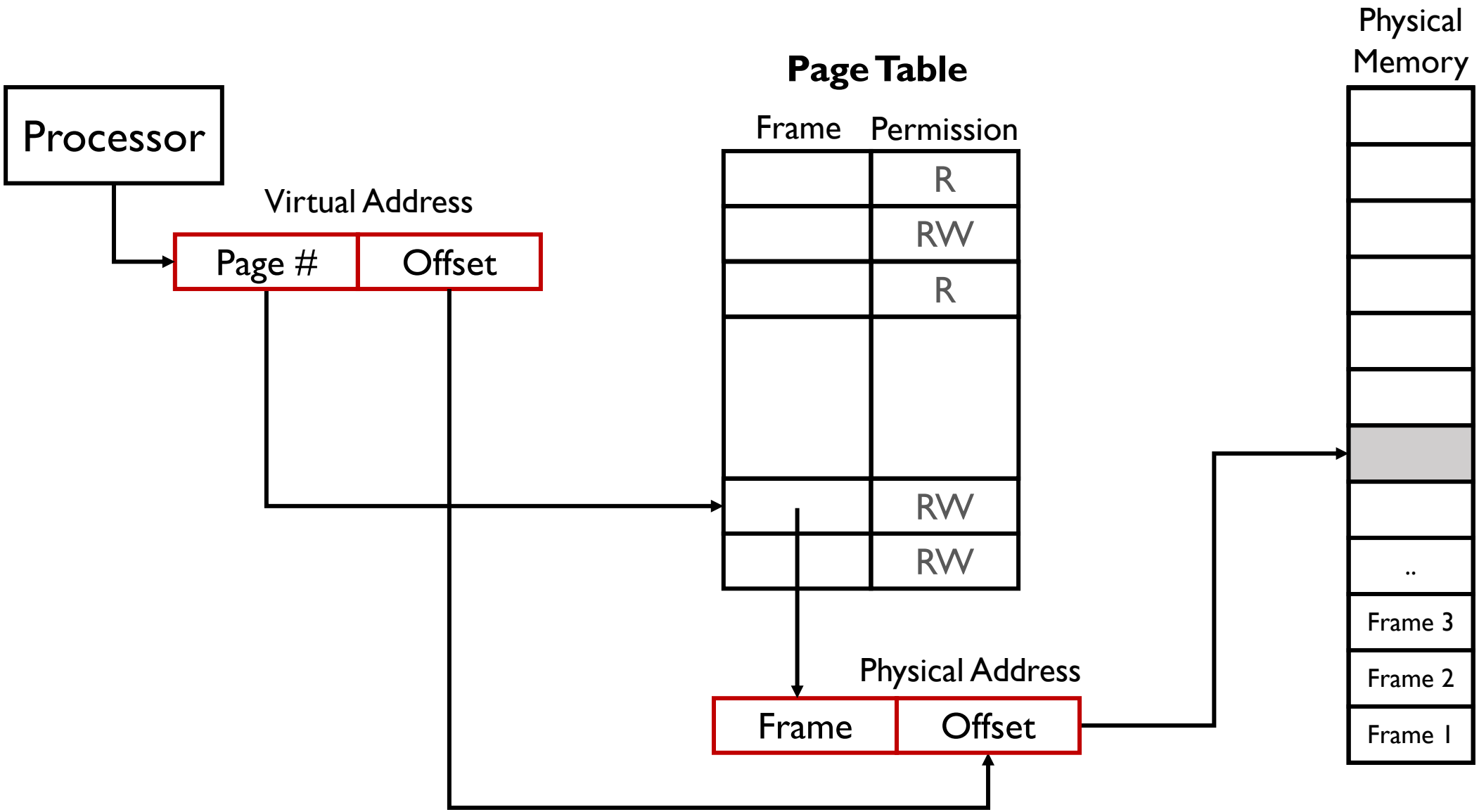
Cooperating Processes

- 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 of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Examples of IPC

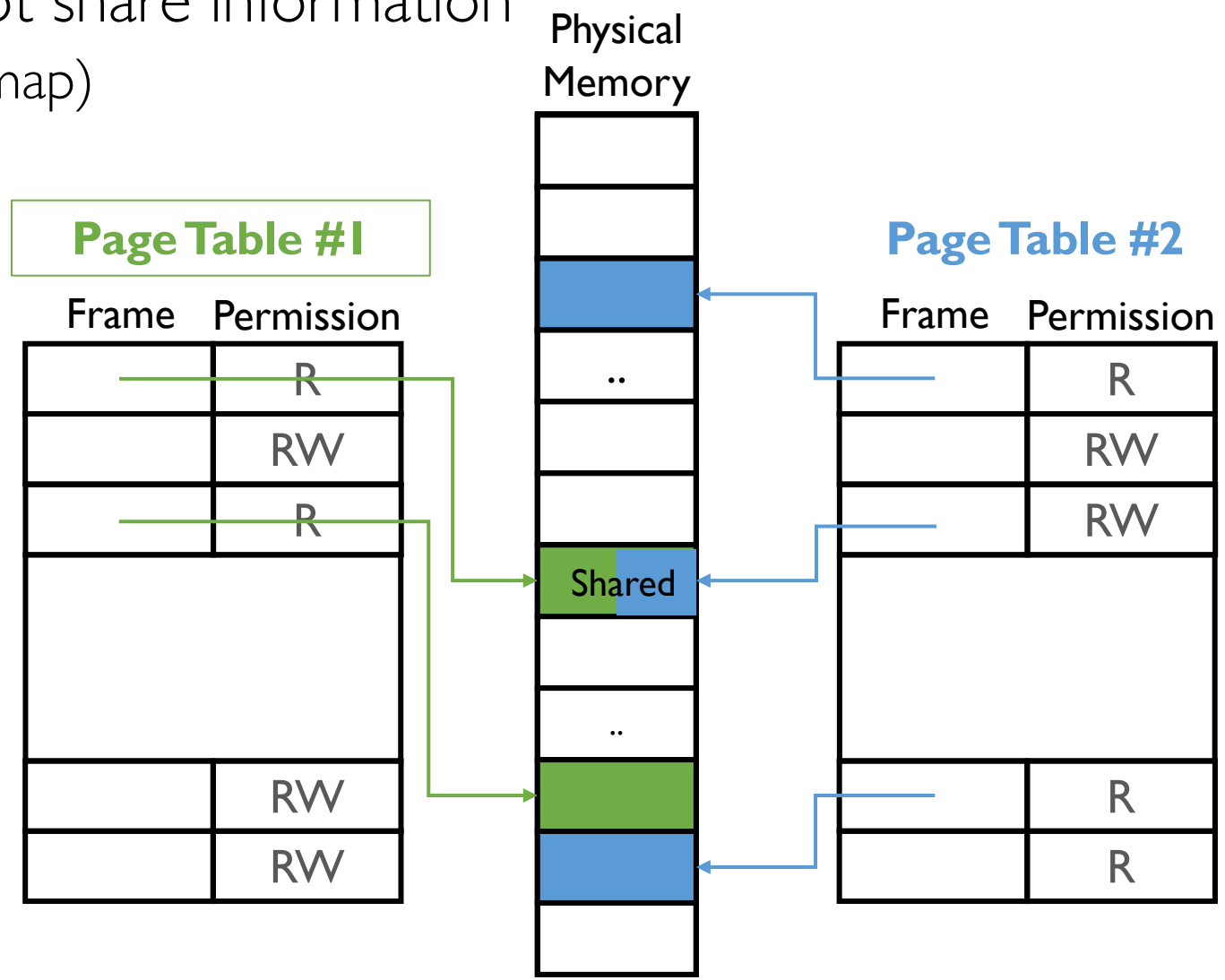
- Web Browsers: Modern web browsers often use multiple processes (one for the main browser interface, others for individual tabs or extensions). IPC allows these processes to share information such as user data, cache, and cookies.
- Online Chat Systems: Applications like Slack or WhatsApp use IPC mechanisms to enable real-time communication between server processes (handling messages) and client processes (user interfaces).
- Video Conferencing Tools: Applications like Zoom or Microsoft Teams use IPC to synchronize video, audio, and data streams, ensuring a cohesive communication experience.
- Gaming: Multiplayer online games use IPC to synchronize game state across client applications and the game server, managing real-time interactions and game physics.
- Automotive Systems: Modern vehicles have multiple electronic control units (ECUs) for engine management, braking systems, infotainment, and more. IPC allows these units to share information and respond to real-time events, like adjusting anti-lock brakes based on speed and traction data.
- Lots more..

Recall: Paged Memory



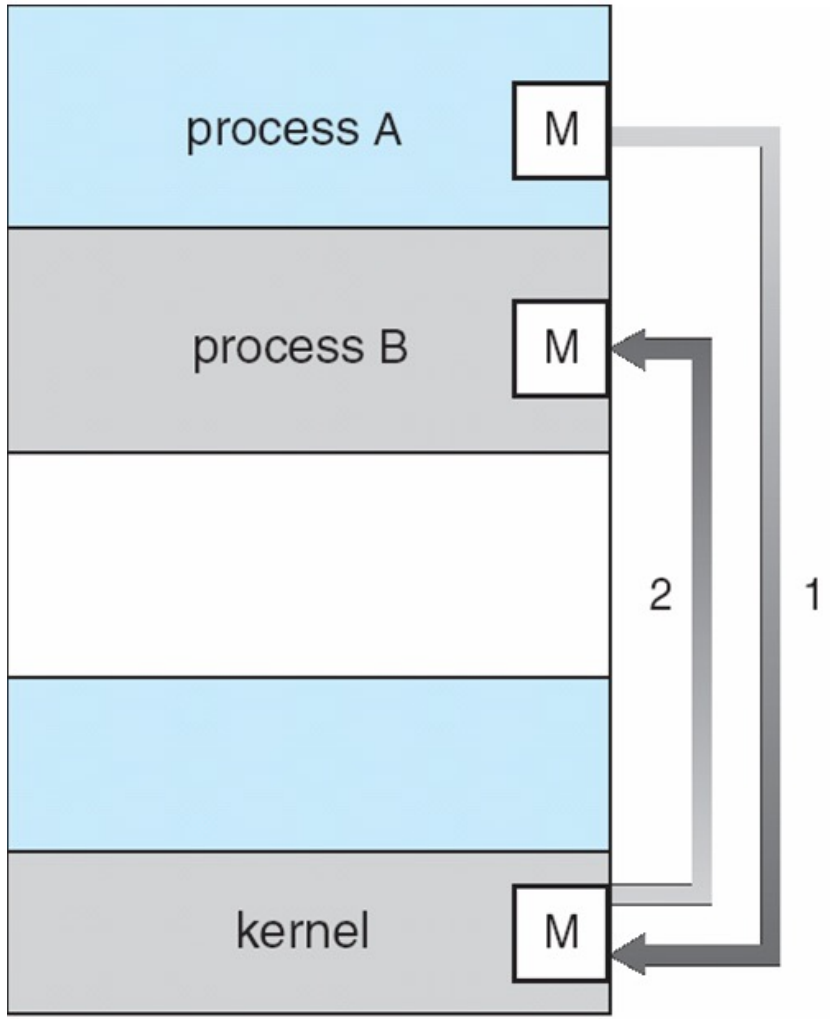
Recall: Processes have independent Addr. Space

- Without OS's help, they cannot share information
  - Except they use file i/o (e.g., mmap)

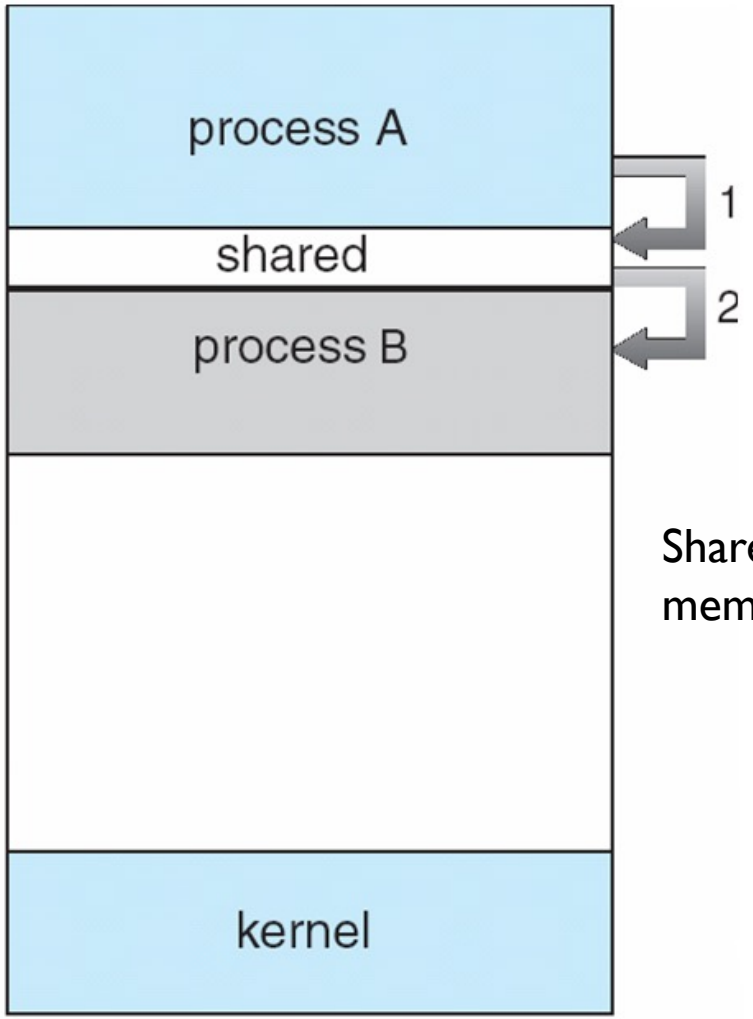


Communication Models

Message passing



(a)



(b)

Shared memory

Goals for Today

- Why Inter-process Communication (IPC)?
- Types of IPCs
  - Pipes
  - Message Queues
  - Shared memory
  - Remote Procedure Calls (RPC)
  - Others..

Unix Pipes

- Unix pipes are a fundamental feature of Unix and Unix-like operating systems, which allow the output of one process to be used as input to another, creating a pipeline that can process data in a sequential and controlled flow.
  - Symbol: In shell commands, a pipe is represented by the vertical bar "|".
  - Unidirectional Data Flow: Unix pipes are unidirectional, meaning data flows in only one direction. If bidirectional communication is needed, two pipes must be set up.
  - In-Memory Buffer: The pipe creates an in-memory buffer which can be written to by the outputting process and read from by the inputting process. The system manages this buffer, providing a form of temporary storage.
  - Synchronous Operations: Reading from and writing to pipes is a synchronous operation. If the pipe's buffer is full, the writing process will be blocked until there is space available. Similarly, if the buffer is empty, the reading process will be blocked until there is data available.

Unix Pipes ("|") in CLI

- Basic usage: command1 | command2 | ..

```
ls | sort
```

```
ls -l | grep ".txt" | sort
```

```
ps aux | grep httpd > processes.txt
```

```
cat access.log | grep "404"
```

```
cat list.txt | while read user; do
  add_user "$user"
done
```

Unix Pipes ("|") in CLI

- Basic usage: command1 | command2 | ..

ls | sort

- lists files in the current directory (ls) and sorts them alphabetically (sort).

ls -l | grep ".txt" | sort

- lists all files with details (ls -l), filters for files with a .txt extension (grep ".txt"), and sorts them (sort).

ps aux | grep httpd > processes.txt

- searches for httpd processes (ps aux | grep httpd) and writes the output to processes.txt.

cat access.log | grep "404"

- searches for "404" in the access.log file.

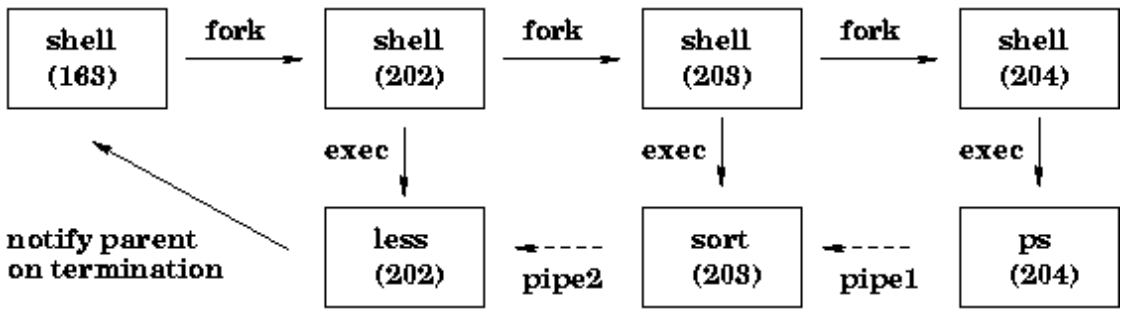
```
cat list.txt | while read user; do
  add_user "$user"
done
```

- reads list.txt and adds each user found in the list with add_user.

Unix Pipes ("|") in CLI

- Basic usage: command1 | command2 | ..
- If any command in a pipeline fails, the entire pipe operation might still succeed. To catch errors, you might need to set the pipefail option in bash (set -o pipefail).
- Piping does not work for commands that require interactive input; it is primarily used for processing static or streaming data.
- Each command in a pipe is executed in its own subshell.

ps | sort | less

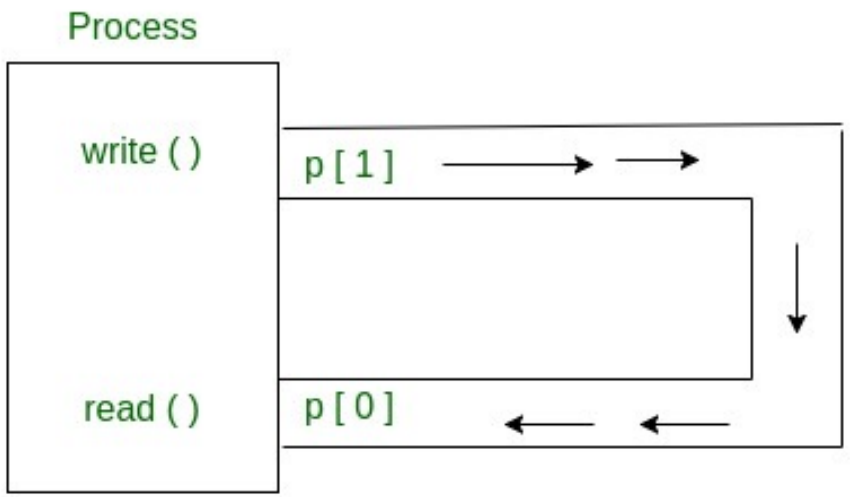


Unix Pipes in C

Syntax in C language:

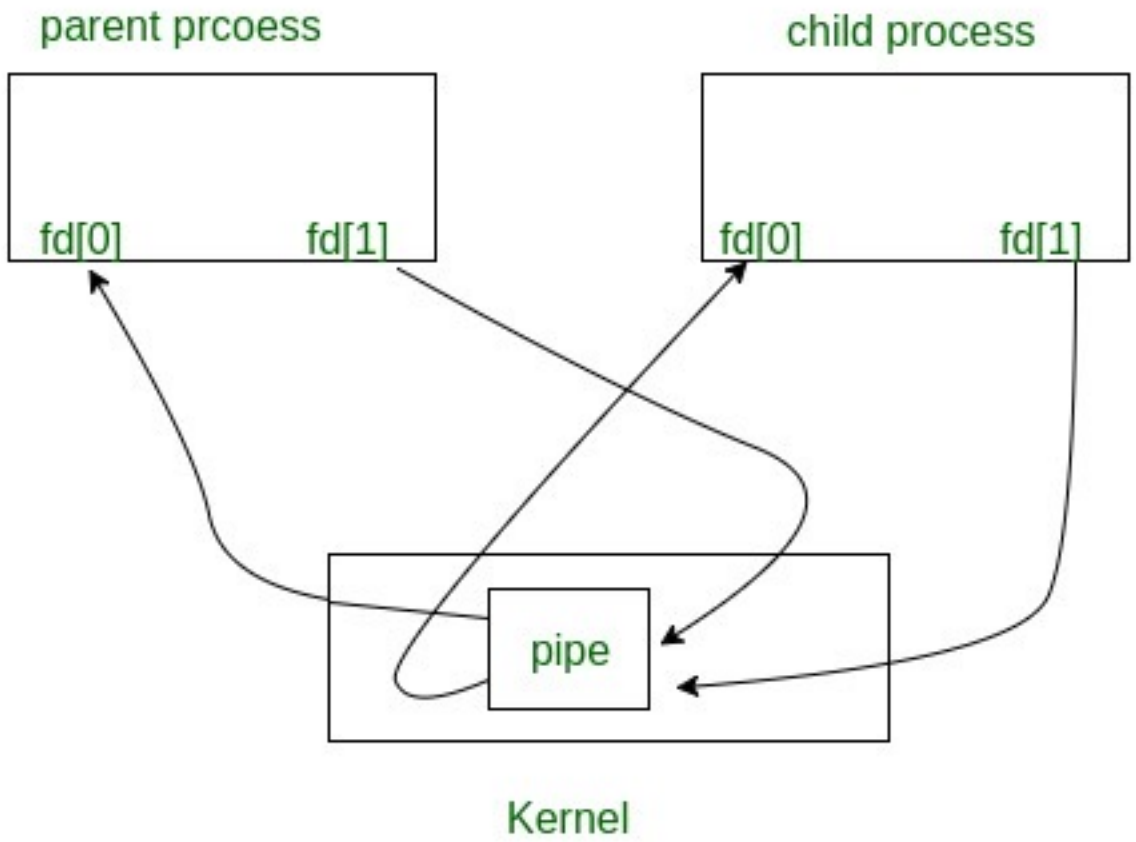
```
int pipe(int fds[2]);

Parameters :
fd[0] will be the fd(file descriptor) for the
read end of pipe.
fd[1] will be the fd for the write end of pipe.
Returns : 0 on Success.
-1 on error.
```



Unix Pipes in C

- When we use fork in any process, file descriptors remain open across child process and also parent process. If we call fork after creating a pipe, then the parent and child can communicate via the pipe.



Unix Pipes in C

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

int main(void) {
    int fds[2];
    char buffer[100];
    ssize_t numRead;

    if (pipe(fds) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    switch (fork()) {
        case -1: // Error
            perror("fork");
            exit(EXIT_FAILURE);

        case 0: // Child - writes to pipe
            close(fds[0]); // Close unused read end
            write(fds[1], "Hello, world!\n", 14);
            close(fds[1]); // Finished writing
            break;

        default: // Parent - reads from pipe
            close(fds[1]); // Close unused write end
            numRead = read(fds[0], buffer, sizeof(buffer));
            if (numRead == -1) {
                perror("read");
                exit(EXIT_FAILURE);
            }
            if (numRead == 0) { // End-of-File
                printf("End-of-File encountered\n");
                exit(EXIT_FAILURE);
            }
            printf("Parent got: %s", buffer);
            close(fds[0]); // Finished reading
            break;
    }

    return EXIT_SUCCESS;
}
```

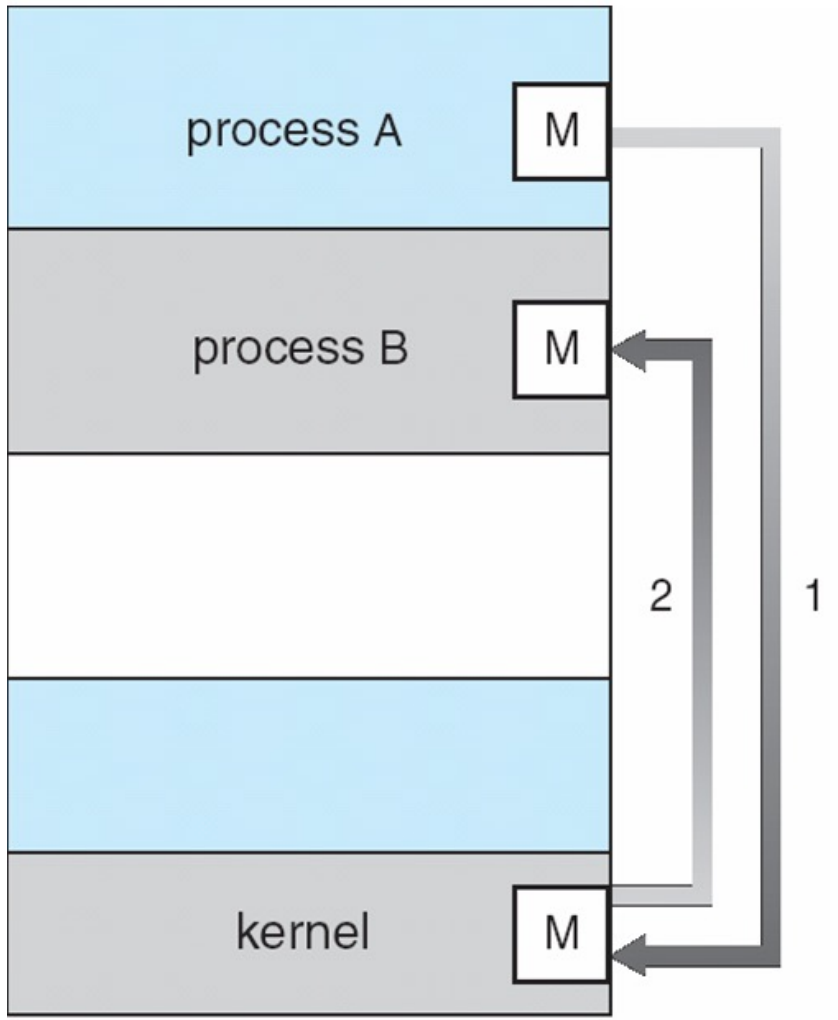
Unix Pipes in Python

```python
import subprocess

# Run a command and pass the output to another command
p1 = subprocess.Popen(['ls', '-l'], stdout=subprocess.PIPE)
p2 = subprocess.Popen(['grep', 'myFile'], stdin=p1.stdout, stdout=subprocess.PIPE)
p1.stdout.close()  # Allow p1 to receive a SIGPIPE if p2 exits.
output = p2.communicate()[0]
print(output.decode())
```

Implementing Pipes

- File Descriptor Table: Every process has a file descriptor table that keeps track of the files and streams it has open. A file descriptor is simply an index into this table.
- Data Buffer: The kernel maintains a buffer for the pipe, which is a block of memory in the kernel space. The size of the buffer is system-dependent.
- Synchronization: Pipes inherently provide synchronization. Writers may block if the pipe is full, and readers may block if the pipe is empty; it allows pipes to be used in producer-consumer scenarios without additional locking mechanisms.



(a)

Pipes in xv6

```
13  struct pipe {
14    struct spinlock lock;
15    char data[PIPESIZE];
16    uint nread;     // number of bytes read
17    uint nwrite;    // number of bytes written
18    int readopen;   // read fd is still open
19    int writeopen;  // write fd is still open
20  };
```

Source: https://github.com/mit-pdos/xv6-public/blob/master/pipe.c

```
78    int
79    pipewrite(struct pipe *p, char *addr, int n)
80    {
81      int i;
82
83      acquire(&p->lock);
84      for(i = 0; i < n; i++){
85        while(p->nwrite == p->nread + PIPESIZE){  //DOC: pipewrite-full
86          if(p->readopen == 0 || myproc()->killed){
87            release(&p->lock);
88            return -1;
89          }
90          wakeup(&p->nread);
91          sleep(&p->nwrite, &p->lock);  //DOC: pipewrite-sleep
92        }
93        p->data[p->nwrite++ % PIPESIZE] = addr[i];
94      }
95      wakeup(&p->nread);  //DOC: pipewrite-wakeup1
96      release(&p->lock);
97      return n;
98    }
```

```
100     int
101     piperead(struct pipe *p, char *addr, int n)
102     {
103       int i;
104
105       acquire(&p->lock);
106       while(p->nread == p->nwrite && p->writeopen){  //DOC: pipe-empty
107         if(myproc()->killed){
108           release(&p->lock);
109           return -1;
110         }
111         sleep(&p->nread, &p->lock); //DOC: piperead-sleep
112       }
113       for(i = 0; i < n; i++){  //DOC: piperead-copy
114         if(p->nread == p->nwrite)
115           break;
116         addr[i] = p->data[p->nread++ % PIPESIZE];
117       }
118       wakeup(&p->nwrite);  //DOC: piperead-wakeup
119       release(&p->lock);
120       return i;
121     }
```

Goals for Today

- Why Inter-process Communication (IPC)?
- Types of IPCs
  - Pipes
  - Message Queues
  - Shared memory
  - Remote Procedure Calls (RPC)
  - Others..

Message Passing

- Mechanism for processes to communicate and to synchronize their actions

- Message system
  - processes communicate with each other without resorting to shared variables

- IPC facility provides two operations:
  - send(message) – message size fixed or variable
  - receive(message)

Message Passing

- 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
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., logical properties)
- Questions
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Direct Communication

- 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

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A

Indirect Communication

- Mailbox sharing
  - P1, P2, and P3 share mailbox A
  - P1, sends; P2 and P3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Synchronization & Asynchronous

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send has the sender block until the message is received
  - Blocking receive has the receiver block until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send has the sender send the message and continue
  - Non-blocking receive has the receiver receive a valid message or null
- Queue of messages attached to the link; implemented in one of three ways
  - Zero capacity – 0 messages
    Sender must wait for receiver (rendezvous)
  - Bounded capacity – finite length of n messages
    Sender must wait if link full
  - Unbounded capacity – infinite length
    Sender never waits

Unix Message Passing

#include <sys/msg.h>

int msgget(key_t *key*, int *msgflg*);

- The **msgget**() system call returns the System V message queue identifier associated with the value of the *key* argument.

int msgsnd(int *msqid*, const void *msgp*[.*msgsz*], size_t *msgsz*, int *msgflg*);

ssize_t msgrcv(int *msqid*, void *msgp*[.*msgsz*], size_t *msgsz*, long *msgtyp*, int *msgflg*);

- The calling process must have write permission on the message queue in order to send a message, and read permission to receive a message.
- The *msgp* argument is a pointer to a caller-defined structure of the following general form. The *mtext* field is an array (or other structure) whose size is specified by *msgsz*

```
struct msgbuf {
    long mtype;       /* message type, must be > 0 */
    char mtext[1];    /* message data */
};
```

sage Passing

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

// Define the message structure
struct my_msg {
    long msg_type;
    char msg_text[100];
};

int main() {
    int msgid;
    struct my_msg message;
    key_t key;

    // Generate a unique key for the message queue (use a file path
    key = ftok("path-to-file", 'a');
    if (key == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }

    // Create a message queue
    msgid = msgget(key, 0666 | IPC_CREAT);
    if (msgid == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
```

```c
    // Send a message
    message.msg_type = 1; // Message type must be a positive number
    strcpy(message.msg_text, "Hello, world!");
    if (msgsnd(msgid, &message, sizeof(message.msg_text), 0) == -1) {
        perror("msgsnd");
        exit(EXIT_FAILURE);
    }
    printf("Sent message: %s\n", message.msg_text);

    // Receive a message
    if (msgrcv(msgid, &message, sizeof(message.msg_text), 0, 0) == -1) {
        perror("msgrcv");
        exit(EXIT_FAILURE);
    }
    printf("Received message: %s\n", message.msg_text);

    // Destroy the message queue after use
    if (msgctl(msgid, IPC_RMID, NULL) == -1) {
        perror("msgctl");
        exit(EXIT_FAILURE);
    }

    return 0;
}
```

Goals for Today

- Why Inter-process Communication (IPC)?
- Types of IPCs
  - Pipes
  - Message Queues
  - Shared memory
  - Remote Procedure Calls (RPC)
  - Others..

POSIX Shared Memory

POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the shm_open() system call, as follows:

```
shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
```

- name: The first parameter specifies the name of the shared-memory object. Processes that wish to access this shared memory must refer to the object by this name.
- O_CREAT | O_RDWR : The subsequent parameters specify that the shared-memory
- object is to be created if it does not yet exist (O_CREAT) and that the object is
- open for reading and writing (O_RDWR).
- The last parameter establishes the directory permissions of the shared-memory object.

POSIX Shared Memory

POSIX shared memory is organized using memory-mapped files, which associate the region of shared memory with a file. A process must first create a shared-memory object using the shm_open() system call, as follows:

```
int ftruncate(int fd, off_t length)
```

- cause the regular file named by path or referenced by fd to be truncated to a size of precisely length bytes.

```
void *mmap(void addr[.length], size_t length, int
prot, int flags, int fd, off_t offset);
```

- mmap() creates a new mapping in the virtual address space of the calling process. The starting address for the new mapping is specified in addr. The length argument specifies the length of the mapping (which must be greater than 0).

```c
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* strings written to shared memory */
    const char* message_0 = "Hello";
    const char* message_1 = "World!";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void* ptr;
    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
    /* write to the shared memory object */
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);
    return 0;
}
```

```c
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void* ptr;
    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    /* read from the shared memory object */
    printf("%s", (char*)ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

gcc producer.c -pthread -lrt -o producer
gcc consumer.c -pthread -lrt -o consumer
./consumer & ./producer &
Output: HelloWorld!

```c
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* strings written to shared memory */
    const char* message_0 = "Hello";
    const char* message_1 = "World!";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void* ptr;
    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
    /* write to the shared memory object */
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);
    return 0;
}
```

```c
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void* ptr;
    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    /* read from the shared memory object */
    printf("%s", (char*)ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

What happened?

Is physical memory allocated during mmap?

Recall: Memory-mapped Files

- Memory-mapped Files (内存映射文件) is a segment of virtual memory that has been assigned a direct byte-for-byte correlation with some portion of a file or file-like resource
  - A special case of demand paging
  - A replacement for syscall read()/write()

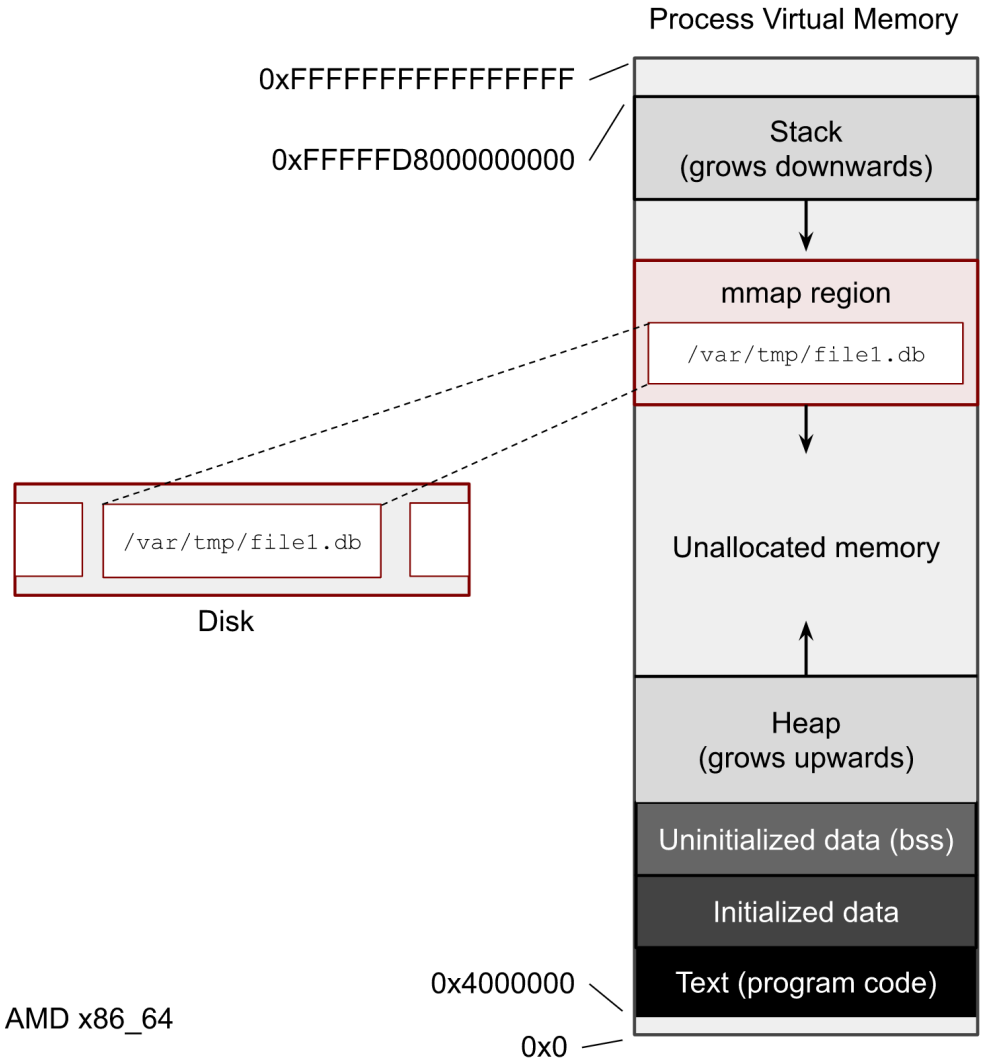
```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags,
           int fd, off_t offset);
int munmap(void *addr, size_t length);
```

mmap(): creates a new mapping in the virtual address space of the calling process. The virtual address starts at *addr* with length *length*. The contents of a file mapping are initialized using length bytes starting at *offset* offset in the file (or other object) referred to by the file descriptor *fd*.

- If addr is NULL, the OS picks a location
- Return value: the address of new mapping

Recall: Memory-mapped Files




```c
int main() {
    int fd;
    char *mapped_data;
    struct stat file_stat;

    // Open the file for reading and writing
    fd = open("example.txt", O_RDWR);

    // Get file size
    if (fstat(fd, &file_stat) < 0) {
        return -1;
    }

    // Map the file into memory
    mapped_data = mmap(NULL, file_stat.st_size, PROT_READ |
PROT_WRITE, MAP_SHARED, fd, 0);

    // Modify the file in memory
    strncpy(mapped_data, "Modified", 8);

    // Sync changes to disk
    if (msync(mapped_data, file_stat.st_size, MS_SYNC) == -1) {
        return -1;
    }

    // Unmap the file and close fd
    if (munmap(mapped_data, file_stat.st_size) == -1) {
        return -1;
    }
    close(fd);
    return 0;
}
```

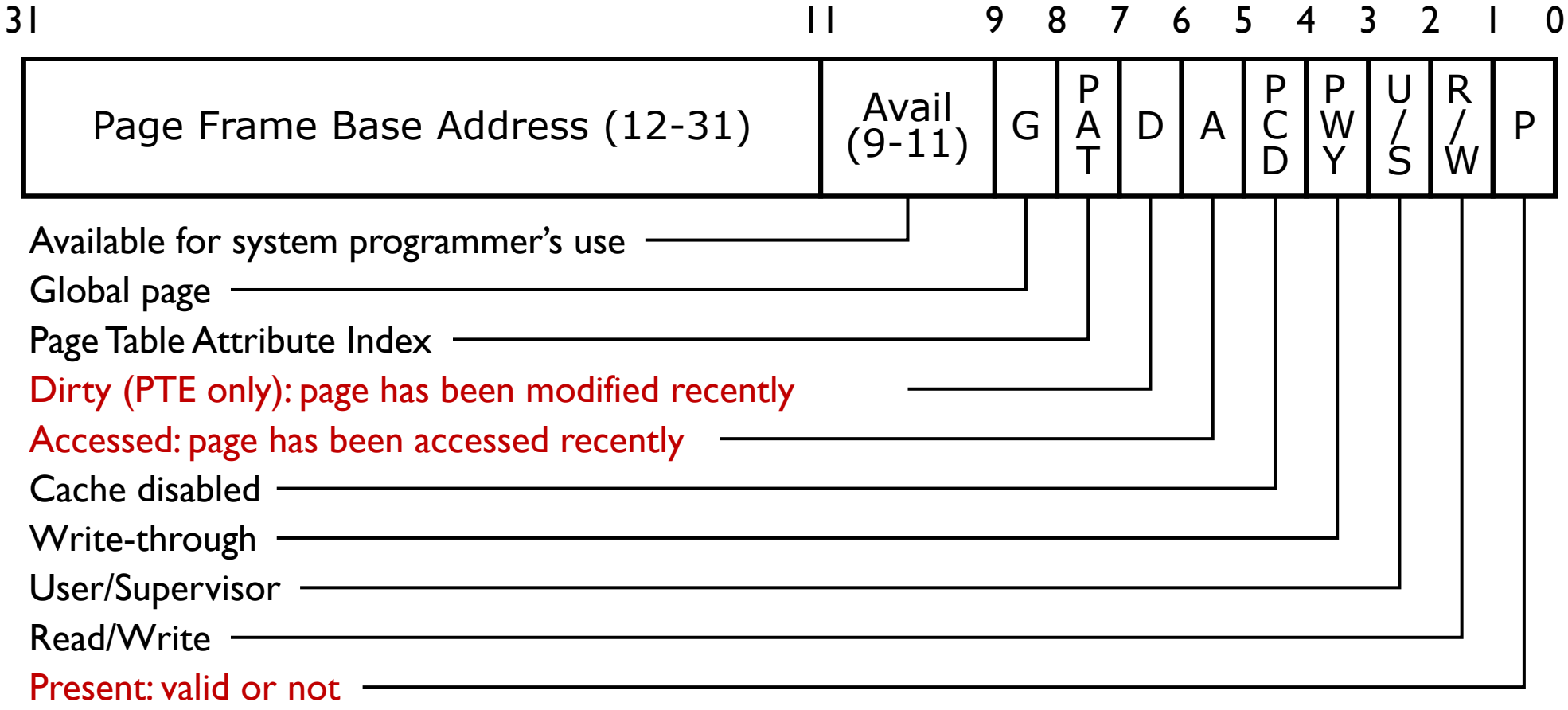
https://biriukov.dev/docs/page-cache/5-more-about-mmap-file-access/

Recall: Memory-mapped Files

- PROS
  - Transparency – the program can use pointers to access those data
  - Zero copy I/O – the OS just changes the page table entries without copying the data into memory; read()/write() needs to copy the data twice (disk-kernel-user)
  - Pipelining – the program can start executing as soon as the page table has been set
  - Interprocess communication – sharing becomes easy
  - Large files – which pages shall be in memory? OS handles it for you
- CONS
  - Frequent page faults
  - A few more..

Recall: Implementation of Memory-mapped Files

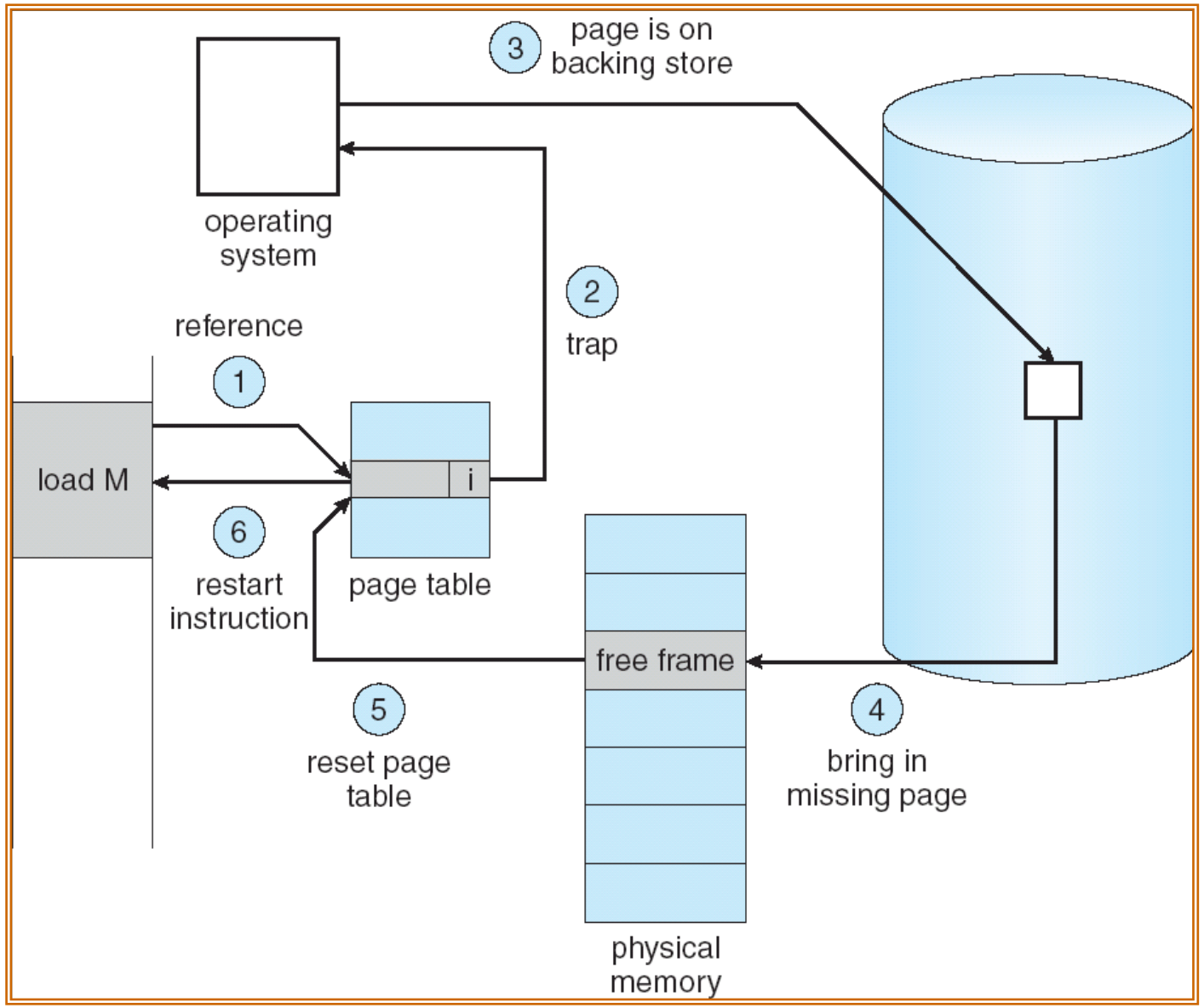
- Set up mapping
  - Initialize the page table entries and setting them to invalid



Recall: Implementation of Memory-mapped Files

- When program accesses an invalid address
  1. [MMU] TLB miss; full page table lookup
  2. [MMU + OS] Trapping into page fault handler
  3. [OS] Convert virtual address to file offset
  4. [OS] Allocate a new page frame in memory
  5. [OS] Read data from disk to the memory (blocked)
  6. [CPU] Disk interrupt when read completes
  7. [OS] Updating page table by marking the entry as valid
  8. [OS] Resume process
  9. [MMU] TLB miss; full page table lookup
  10. [MMU] TLB update

Recall: Implementation of Memory-mapped Files



```c
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* strings written to shared memory */
    const char* message_0 = "Hello";
    const char* message_1 = "World!";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void* ptr;
    /* create the shared memory object */
    shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    /* configure the size of the shared memory object */
    ftruncate(shm_fd, SIZE);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);
    /* write to the shared memory object */
    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);
    return 0;
}
```

```c
int main()
{
    /* the size (in bytes) of shared memory object */
    const int SIZE = 4096;
    /* name of the shared memory object */
    const char* name = "OS";
    /* shared memory file descriptor */
    int shm_fd;
    /* pointer to shared memory object */
    void* ptr;
    /* open the shared memory object */
    shm_fd = shm_open(name, O_RDONLY, 0666);
    /* memory map the shared memory object */
    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    /* read from the shared memory object */
    printf("%s", (char*)ptr);
    /* remove the shared memory object */
    shm_unlink(name);
    return 0;
}
```

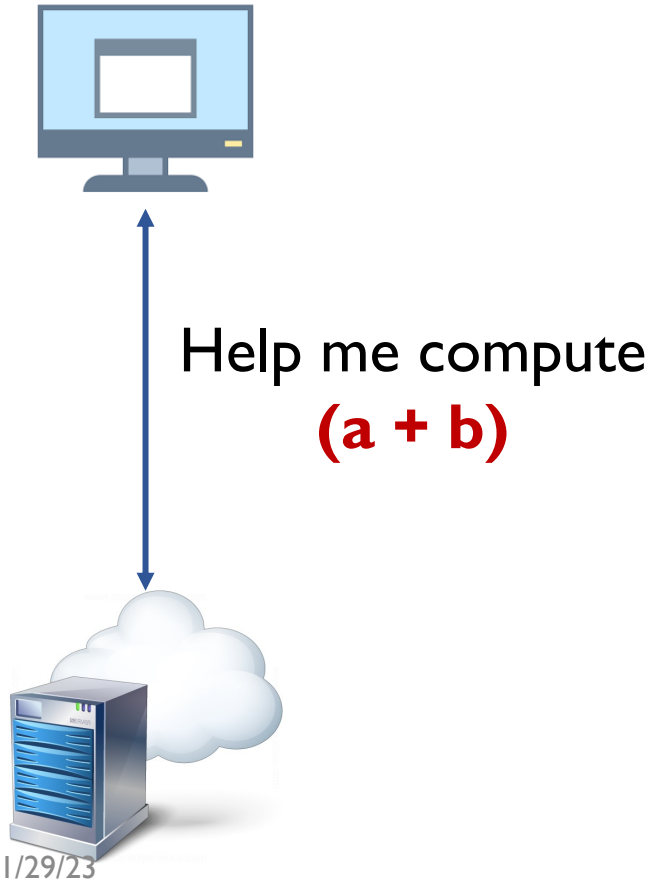
Is there any synchronization?

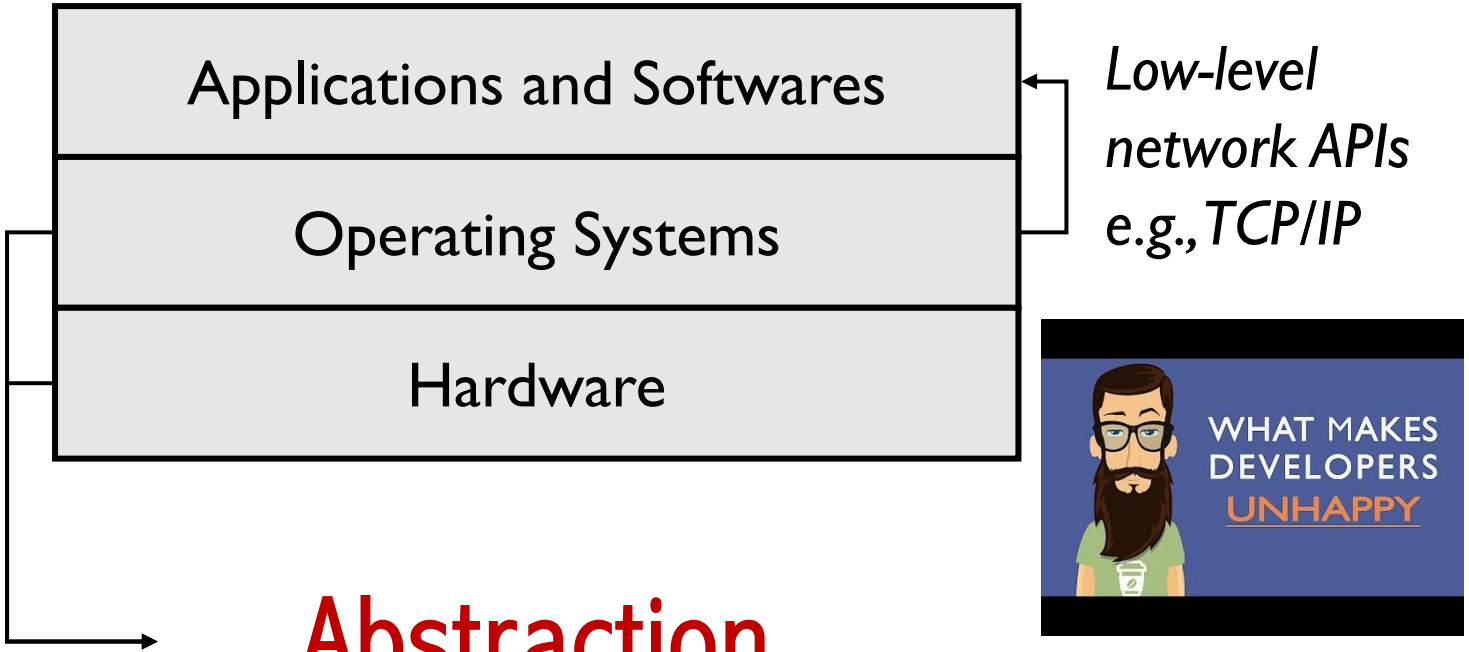
Goals for Today

- Why Inter-process Communication (IPC)?
- Types of IPCs
  - Pipes
  - Message Queues
  - Shared memory
  - Remote Procedure Calls (RPC)
  - Others..

Communications in Distributed Systems

- Inter-process or cross-machine communications



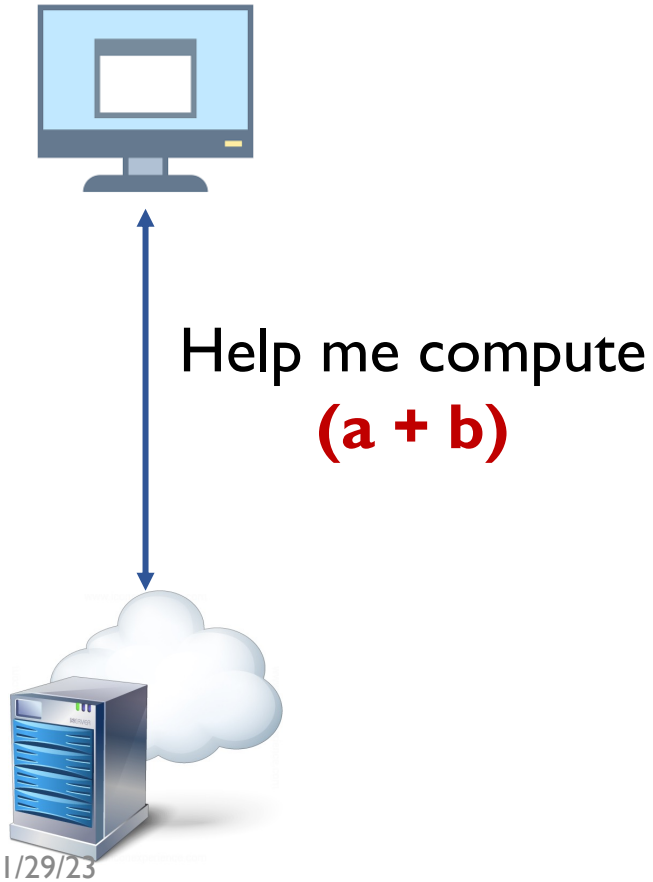


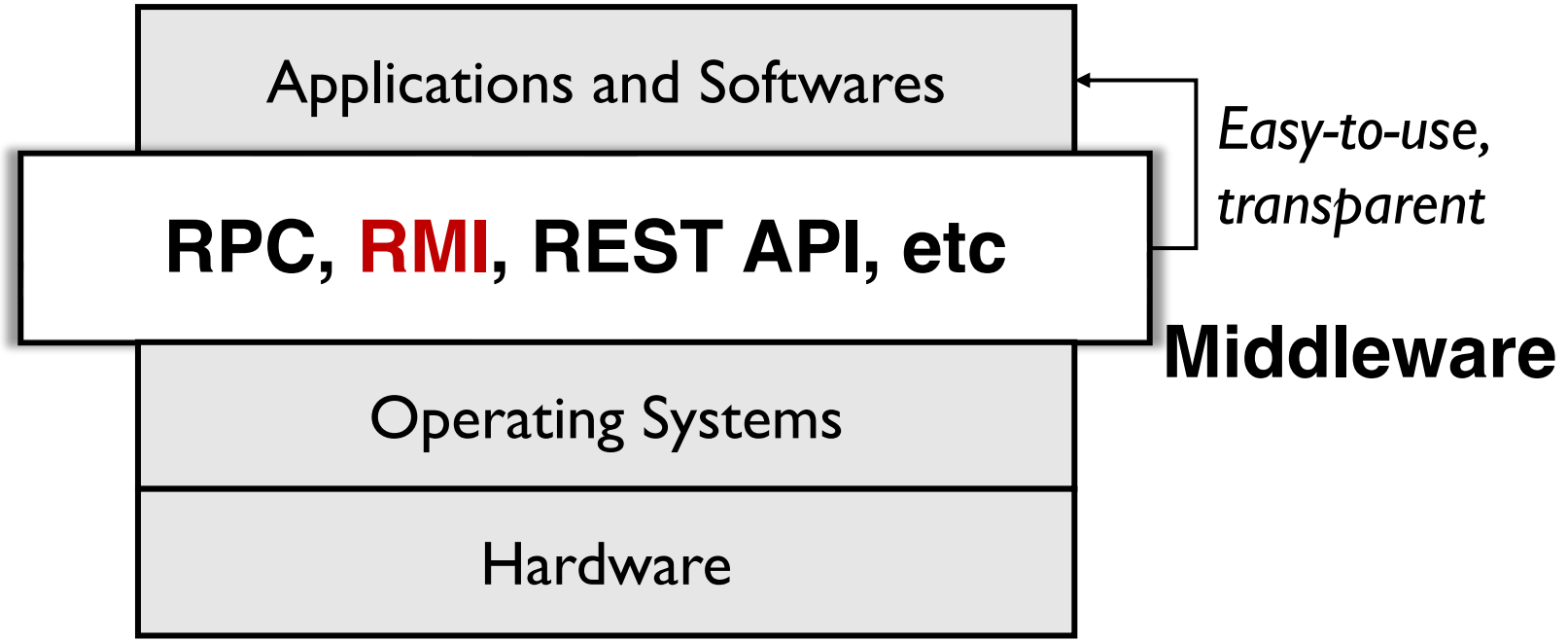
Abstraction

Cleaner APIs and Transparency

Communications in Distributed Systems

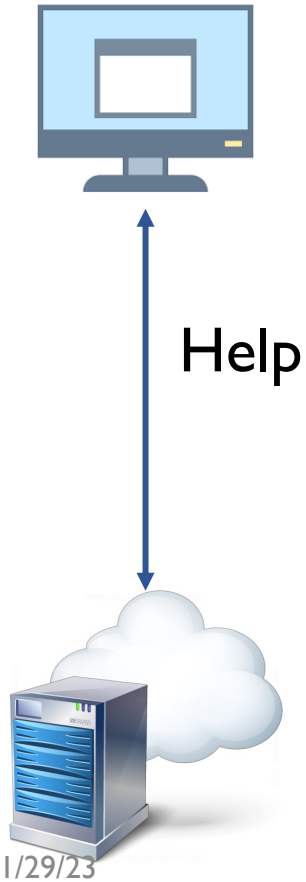
- Inter-process or cross-machine communications





Communications in Distributed Systems

- Inter-process or cross-machine communications

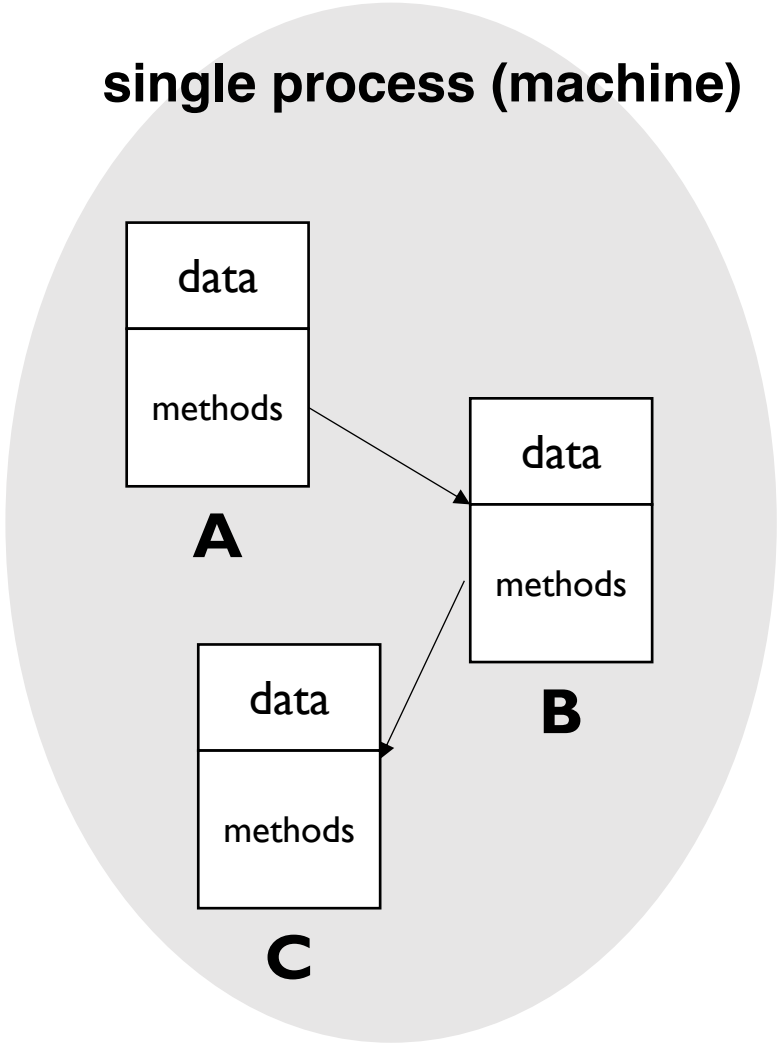


RPC	c = remote_add(a, b)	C
RMI	adder = new remote_Adder() c = adder.add(a, b)	Java
REST	req = "ip:host/adder?a=a&b=b" c = Http.send(req)	any lang

The Object Model

- **Object** = data + methods
  - logical and physical encapsulation
  - instantiated from a *class*
  - accessed via *object references*

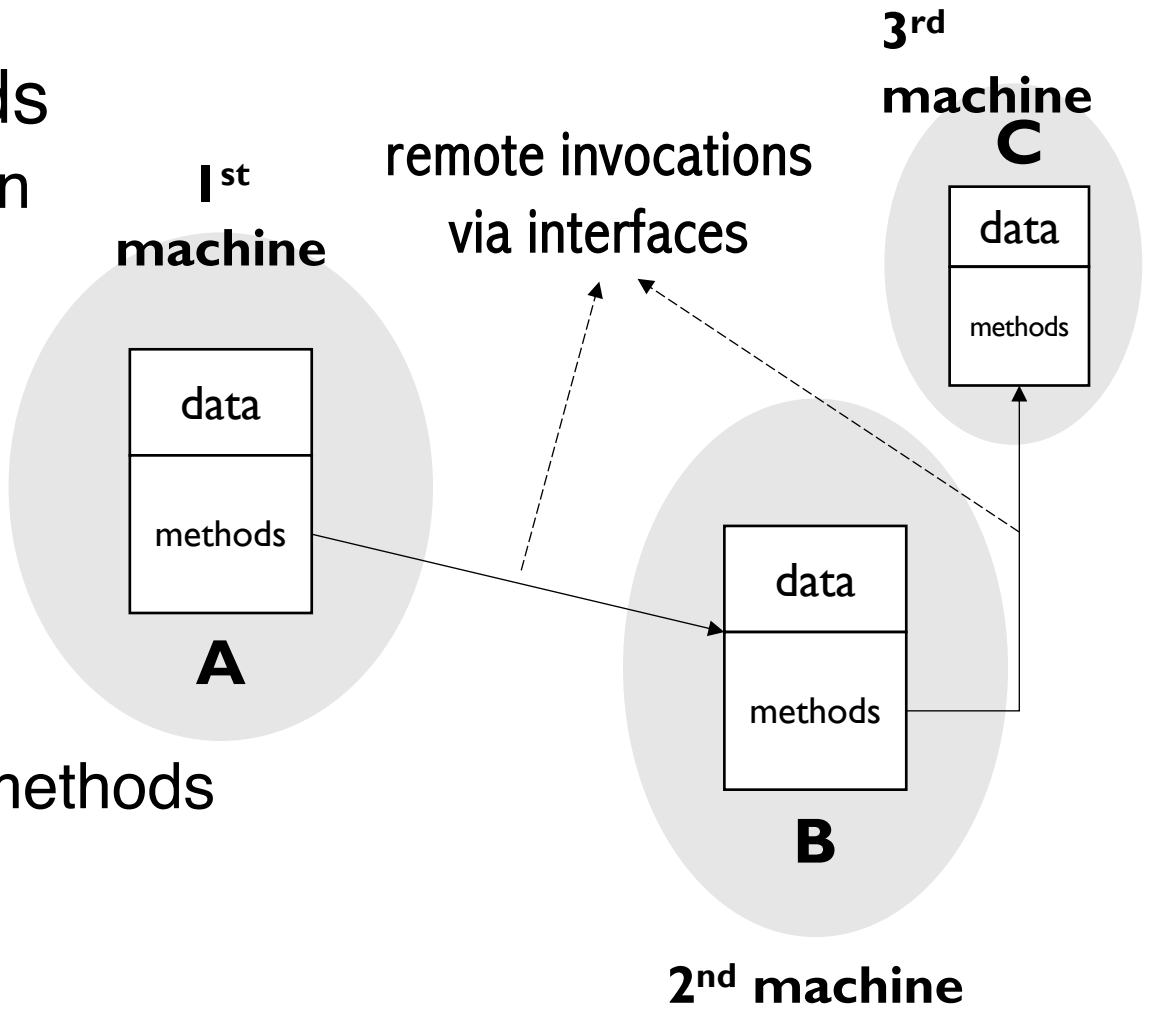
- **Interface**
  - abstract of classes (objects)
  - specify the behaviors (methods)



The Remote Object Model

- Remote Object = data + methods
  - logical and physical encapsulation
  - accessed via object references
  - distributed on different machines
  - remote object references

- Remote Interface
  - abstraction of remote objects
  - specify the remotely-accessible methods
  - in Java: extends class ***Remote***



Java Example

Client

Server

Java Example

Client

Server


```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

Java Example

Client

Server

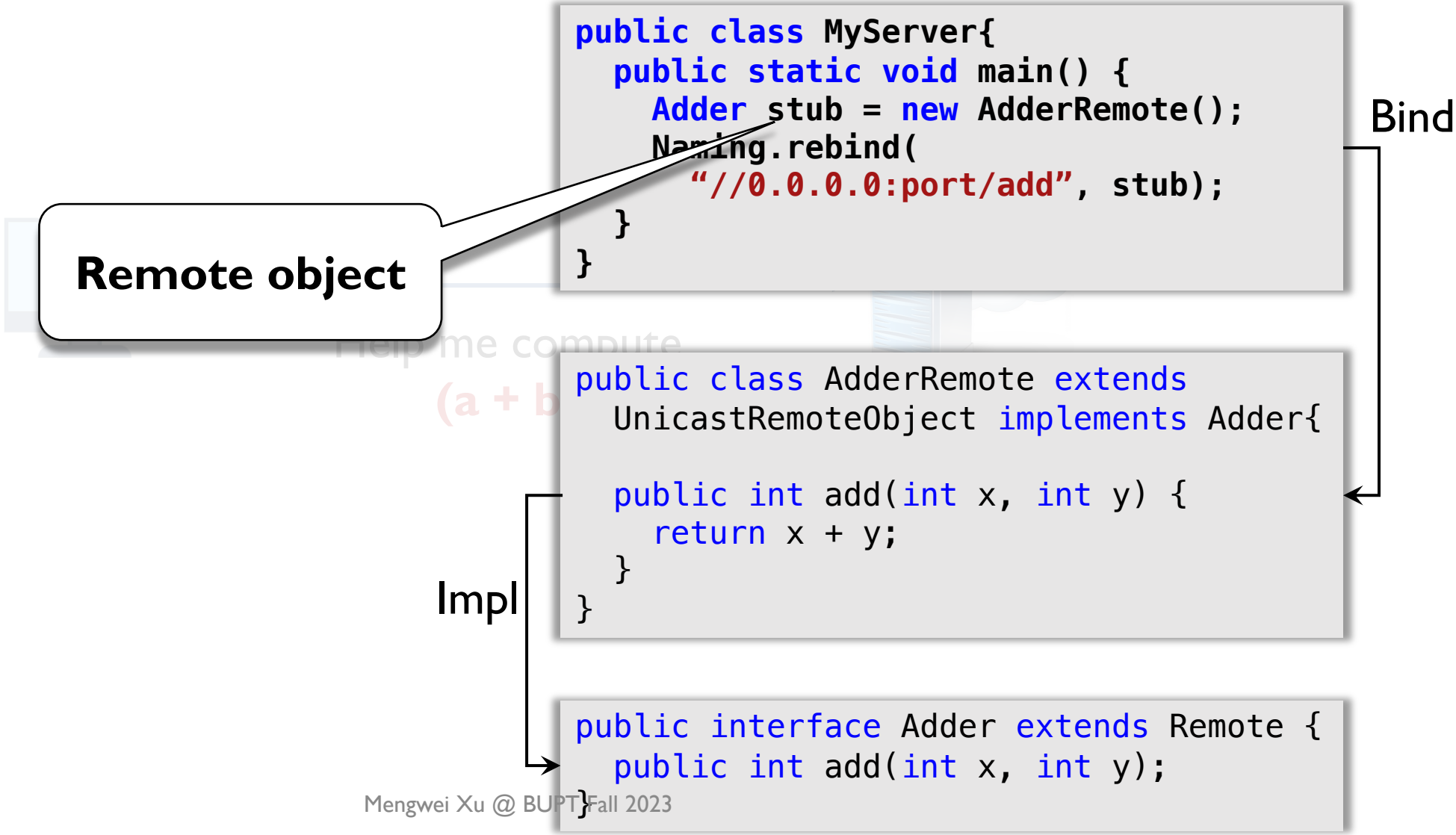

```
public class AdderRemote extends
  UnicastRemoteObject implements Adder{

  public int add(int x, int y) {
    return x + y;
  }
}
```

Impl

```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

Client Java Example Server



Java Example

Client

Server

```
public class MyServer{
  public static void main() {
    Adder stub = new AdderRemote();
    Naming.rebind(
      “//0.0.0.0:port/add”, stub);
  }
}
```

Bind

Help me compute

(a + b

```
public class AdderRemote extends
  UnicastRemoteObject implements Adder{

  public int add(int x, int y) {
    return x + y;
  }
}
```

Impl

```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

Java Example

Client

Server

```
public class MyClient{
  public static void main() {
    Adder stub = (Adder) Naming.lookup(
        "//ip:port/add");
    int c = stub.add(a, b));
  }
}
```

```
public class MyServer{
  public static void main() {
    Adder stub = new AdderRemote();
    Naming.rebind(
      "//0.0.0.0:port/add", stub);
  }
}
```

Bind

remote invoke

```
public class AdderRemote extends
  UnicastRemoteObject implements Adder{

  public int add(int x, int y) {
    return x + y;
  }
}
```

Impl

```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

Java Example

Client

Server

```
public class MyClient{
  public static void main() {
    Adder stub = (Adder) Naming.lookup(
        "//  port/add");
    int c      b.add(a, b));
  }
}
```

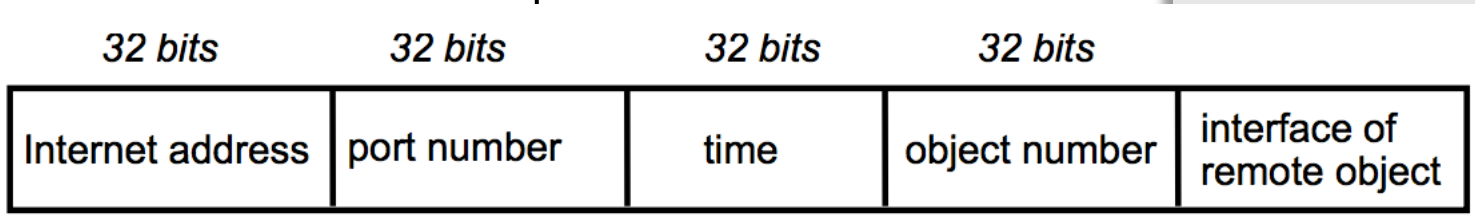
```
public class MyServer{
  public static void main() {
    Adder stub = new AdderRemote();
    Naming.rebind(
      "//0.0.0.0:port/add", stub);
  }
}
```

Bind

Remote object reference

(proxy)

- Object identifiers in a DS
- Unique in space and time




```
public class AdderRemote extends
  UnicastRemoteObject implements Adder{

  add(int x, int y) {
  + y;
```

```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

```
public interface Adder extends Remote {
  public int add(int x, int y);
}
```

RMI Challenges

- Exceptions and Garbage Collection
  - What's changed in distributed systems?

RMI Challenges

- Exceptions and Garbage Collection
- Request-reply protocols (failure strategy)
  - What if requests/replies are delayed or lost during transmission?

✓ ***Retry request message:*** whether to retransmit the request message until either a reply is received or the server is assumed to have failed.

✓ ***Duplicate filtering:*** when retransmissions are used, whether to filter out duplicate requests at the server.

✓ ***Retransmission of results:*** whether to keep a history of result message to enable lost results to be retransmitted without re-executing the operations at the server.

RMI Challenges

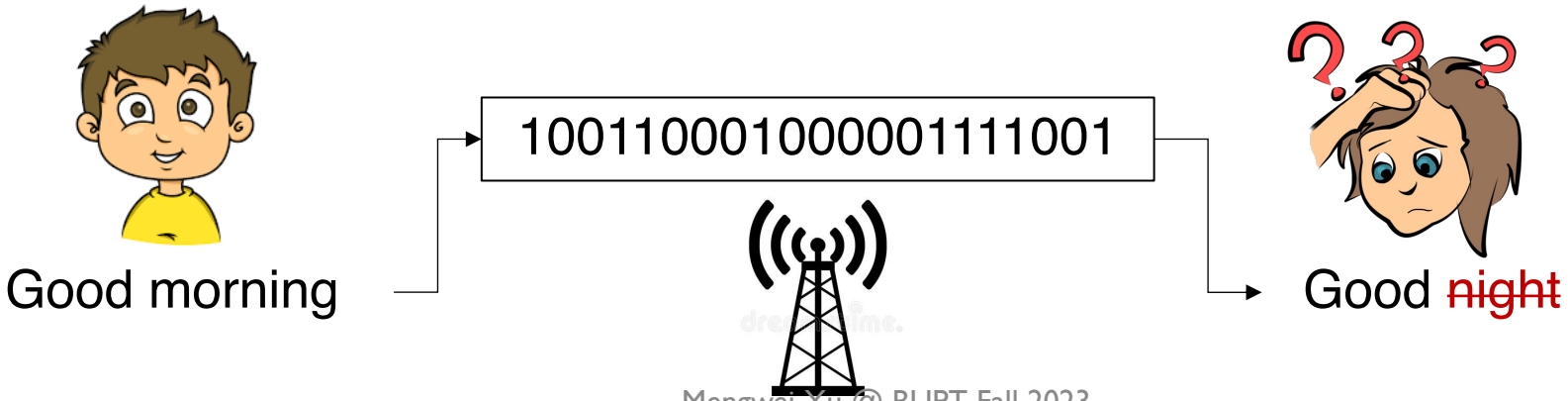
- Exceptions and Garbage Collection
- Request-reply protocols (failure strategy)
  - What if requests/replies are delayed or lost during transmission?

Fault tolerance measures			*Invocation semantics*
Retransmit request message	*Duplicate filtering*	*Re-execute procedure or retransmit reply*	
No	Not applicable	Not applicable	*Maybe*
Yes	No	Re-execute procedure	*At-least-once*
Yes	Yes	Retransmit reply	*At-most-once*

- **What are their applicable scenarios?**
- **How to implement?**
- **Which semantic is used in Java?**

RMI Challenges

- Exceptions and Garbage Collection
- Request-reply protocols
- Data Interoperability
  - Big-endian or little-endian?
  - UTF-8 or ASCII?
  - Floating point representations?



RMI Challenges

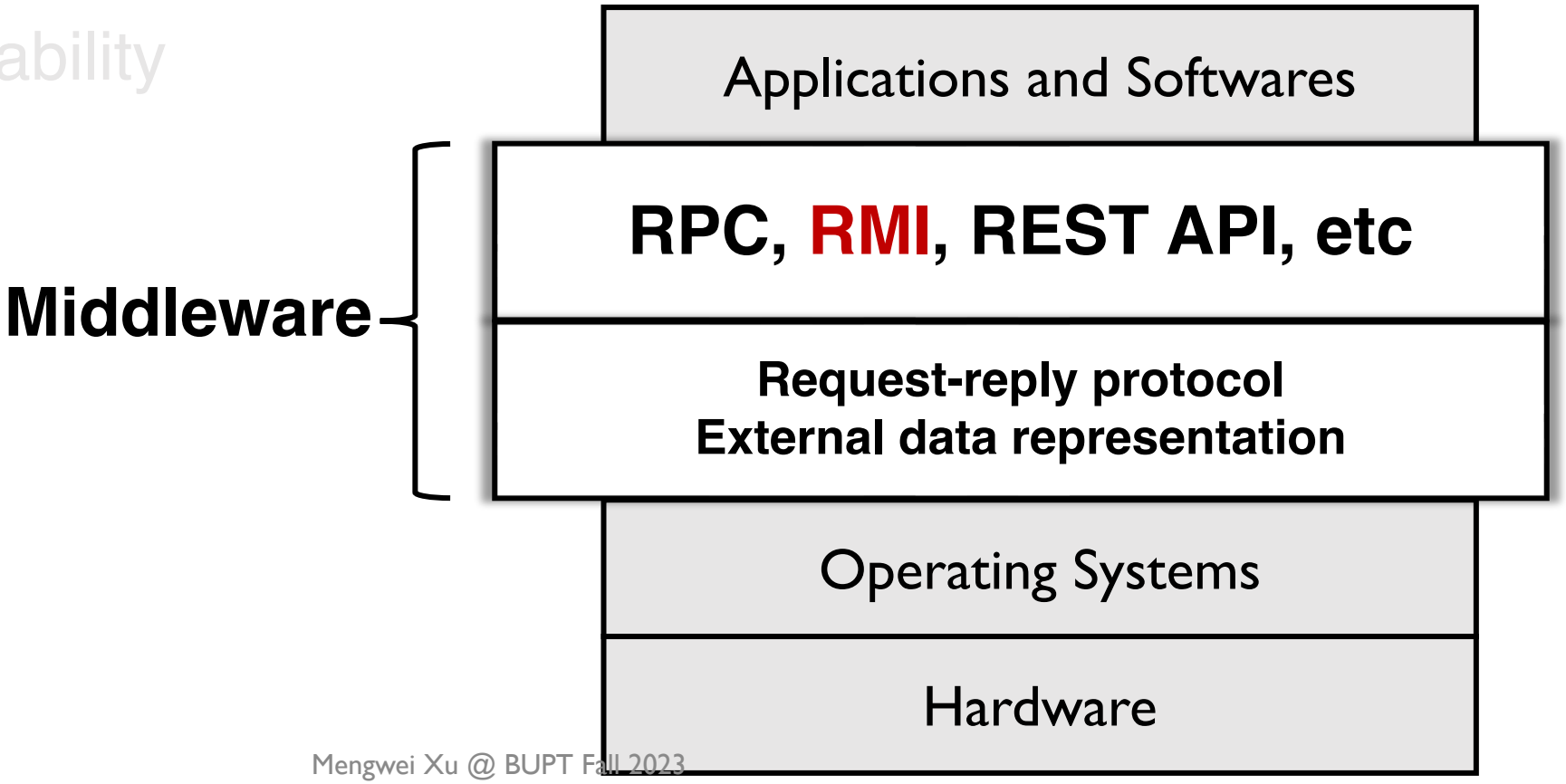
- Exceptions and Garbage Collection
- Request-reply protocols
- Data Interoperability
  - External data representation (XDR)
  - Marshalling and Unmarshalling

XDR data types [edit]

- boolean
- int – 32-bit integer
- unsigned int – unsigned 32-bit integer
- hyper – 64-bit integer
- unsigned hyper – unsigned 64-bit integer
- IEEE float
- IEEE double
- quadruple (new in RFC1832)
- enumeration
- structure
- string
- fixed length array
- variable length array
- union – discriminated union
- fixed length opaque data
- variable length opaque data
- void – zero byte quantity

RMI Challenges

- Exceptions and Garbage Collection
- Request-reply protocols
- Data Interoperability



Which one shall I use..

- Pipes: Command-Line Interface (CLI) in Unix/Linux Systems
  - Combining grep, sort, and awk commands to filter, sort, and process text data.
- Message Queues: Decoupled Communication in Distributed Systems
  - Message queues are ideal in distributed systems where different services or applications run on separate machines or environments. For example, a web application sending tasks to a background worker process for asynchronous processing. Message queues allow these components to communicate reliably and asynchronously, ensuring that messages are not lost even if some parts of the system temporarily fail.
- Shared Memory: High-Performance Computing (HPC) Applications
  - In high-performance computing, where processes running on the same machine need to exchange large amounts of data quickly, shared memory is extremely effective. It is used to avoid the overhead of data copying between processes,.
- RPC: Client-Server Architectures in Network Applications
  - RPC is particularly useful in client-server architectures, such as a networked database system where a client application needs to perform operations on a database server. The client can invoke procedures (functions) on the server as if they were local procedures, abstracting the complexity of the network communication.