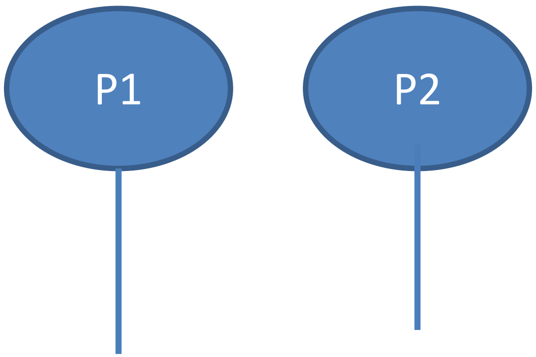
Inter Process Communication

Inter Process Communication

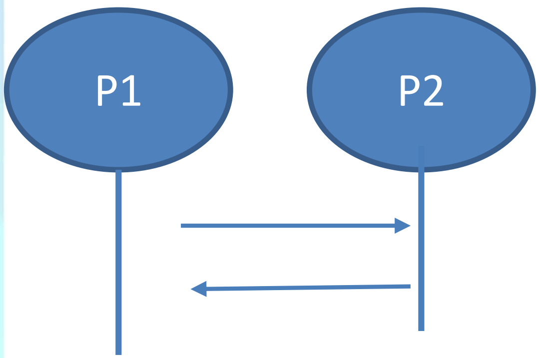
Independent Process:

Doesn’t share any data.



Cooperating Process
(or) Inter process communication

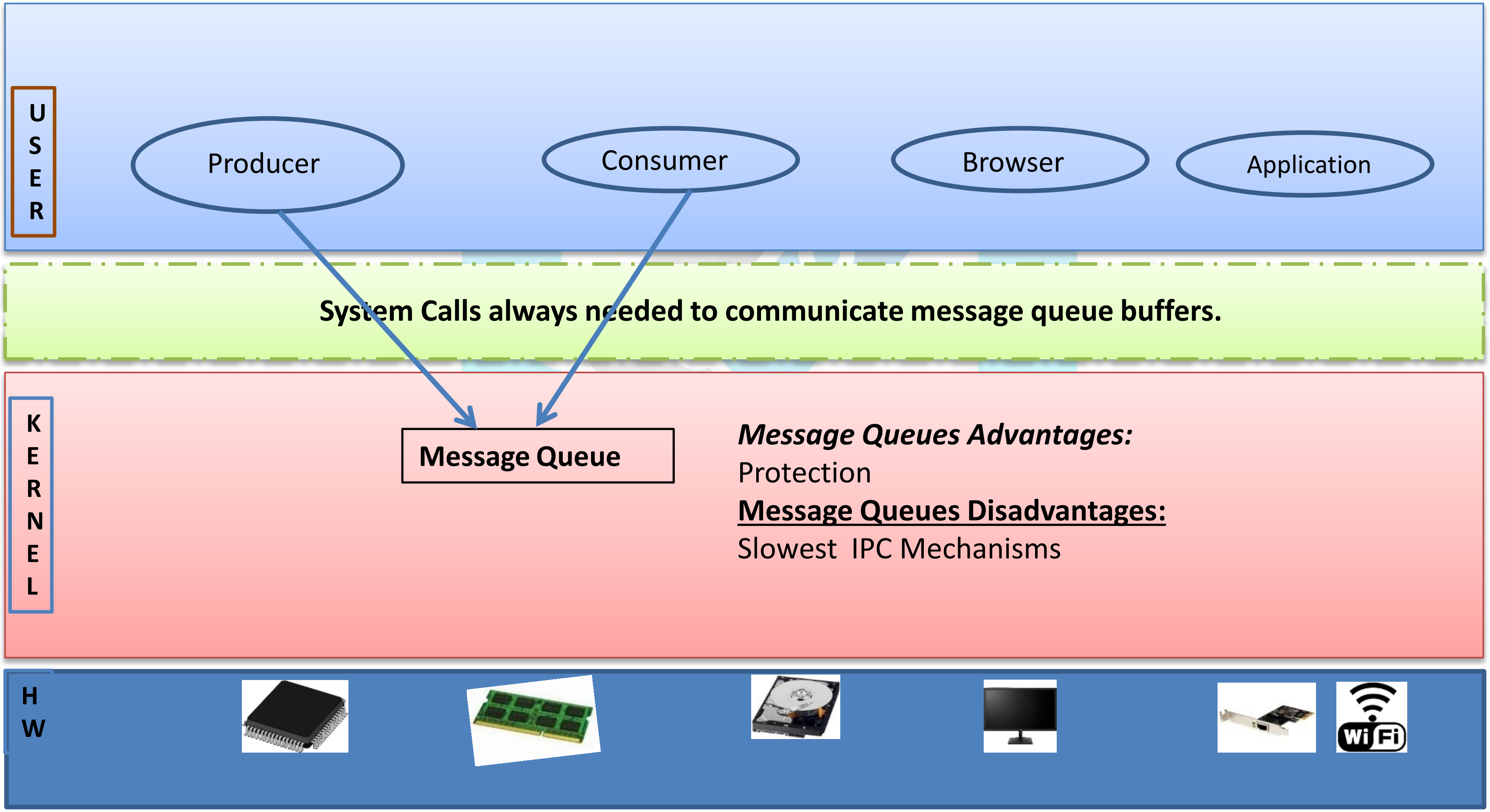
Share any data.



IPC Mechanisms:

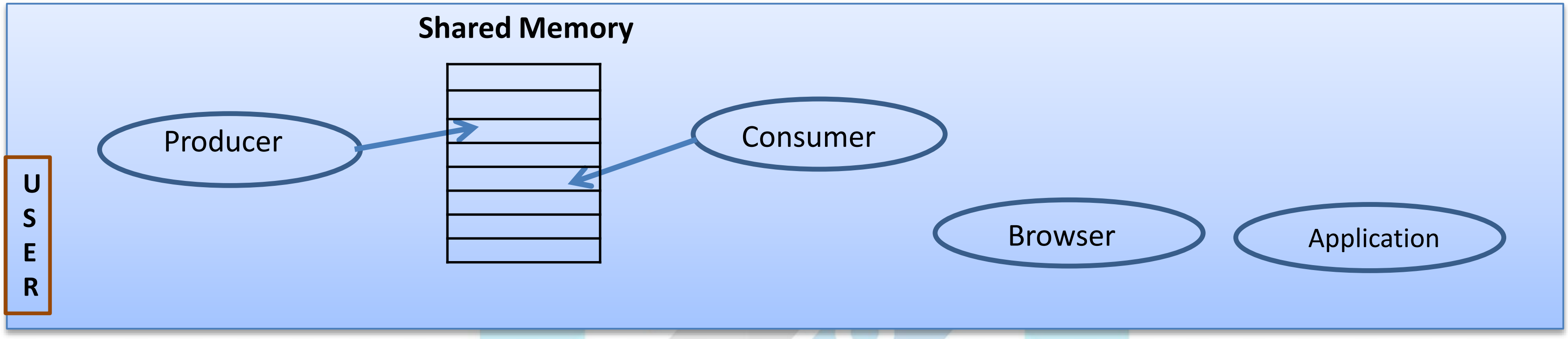
1. Pipes
2. Message Queues
3. Shared Memory
4. Sockets

IPC: Message Queues



KERNEL MASTERS

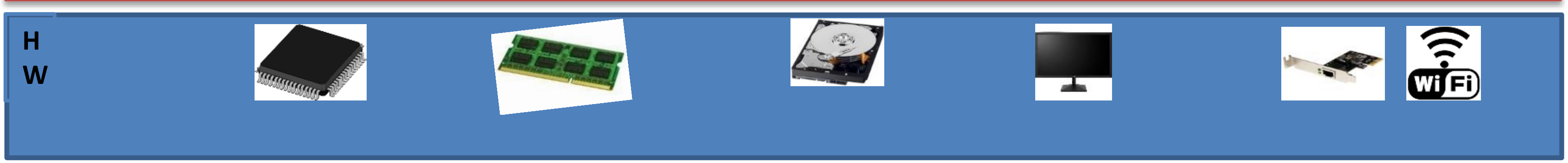
IPC: Shared Memory



System Calls always needed to establish communication

K
E
R
N
E
L

<u>Shared Memory Advantages:</u>
Fastest IPC Mechanisms
<u>Shared Memory Disadvantages:</u>
Doesn’t Support Synchronization Mechanisms



Pipes

Simplest IPC Mechanism

Pipes

- **<u>Connection oriented:</u>** Writing or reading is possible only after connection is established between two processes.
- Simples form of IPC.
- Unidirectional Communication.

Pipe is called **<u>Unnamed Pipe</u>**

<u>Use:</u> Command line

$ cat hello.c | grep hello

<u>Named Pipe</u>

/dev/fb0

<u>Use:</u> Device file

/dev/fb0 → monitor

/dev/sda → harddisk

Pipes

Simplest IPC Mechanism

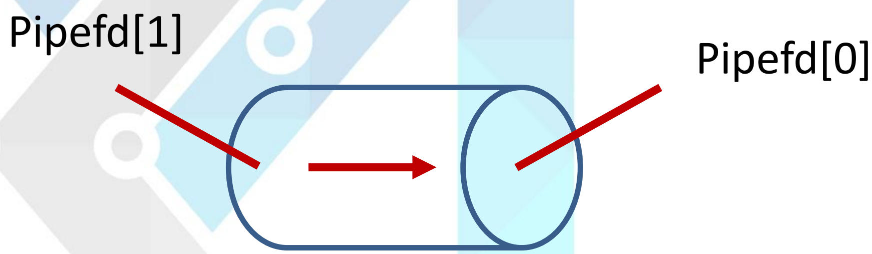
Unnamed pipe

Unnamed Pipe – Creating a Pipe

int pipefd[2];

int pipe(int pipefd[2]);

Returns 0
On SUCCESS
-1 on ERROR



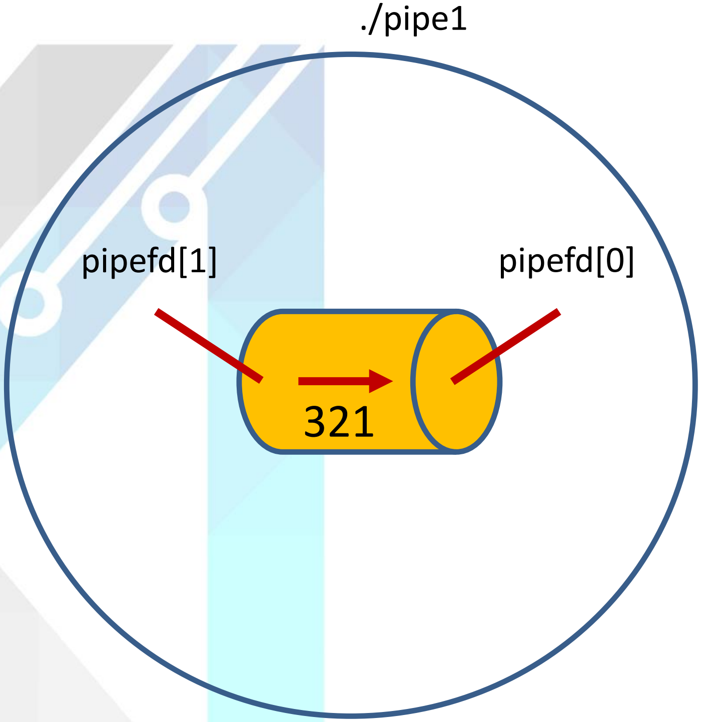
Unnamed Pipe – Examples

pipe1.c: [NOT a perfect IPC Example]

```
int main()
{
   int file_pipes[2];
   const char data[] = "123";
   char buffer[BUFSIZ + 1];

 if (pipe(file_pipes) == 0) {
write(file_pipes[1],data,strlen(data));

read(file_pipes[0], buffer, BUFSIZ);
printf("Read %s\n", buffer);
exit(EXIT_SUCCESS);
   }
```

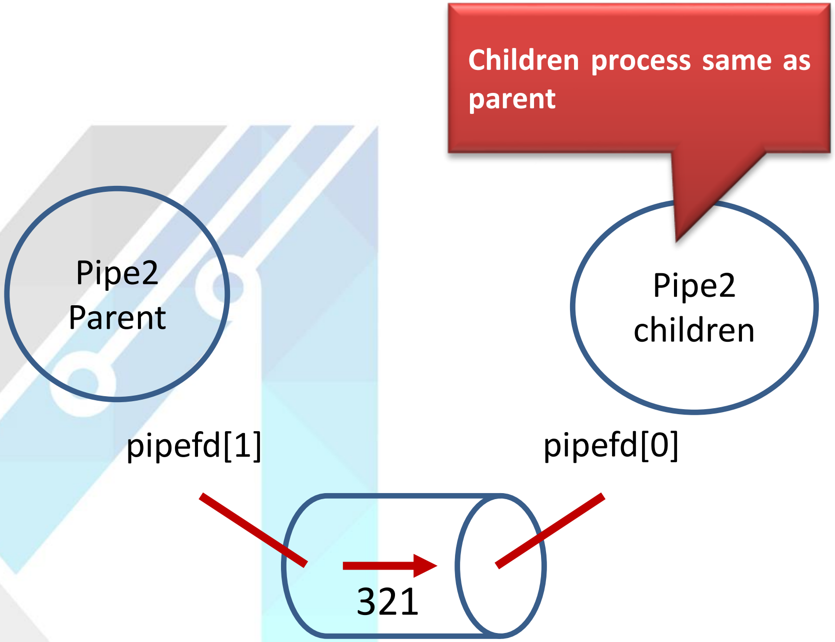


Unnamed Pipe – Examples

pipe2.c: [Perfect IPC Example]

```
int main()
{
   int file_pipes[2];
   char buffer[BUFSIZ + 1];
   pid_t fork_result;

   if (pipe(file_pipes) == 0)
   {
         fork_result = fork();
         if (fork_result == 0)
         {
                read(file_pipes[0], buffer, BUFSIZ);
                printf("Read %s\n", buffer);
                exit(EXIT_SUCCESS);
          }
      else
         {
                write(file_pipes[1],"123",3);
         }
   }
}//main()
```

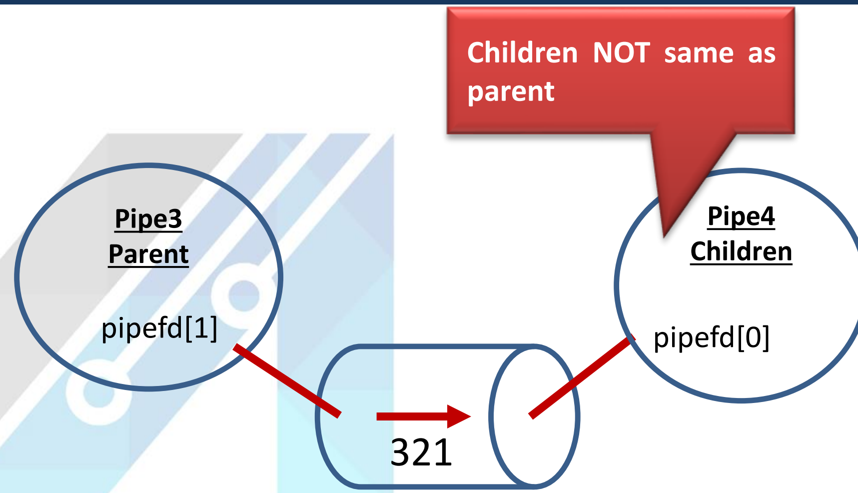


Unnamed Pipe – Examples

pipe3.c: [fork() + execve()]

```
int main()
{
   int file_pipes[2];
   char buffer[BUFSIZ + 1];
   pid_t fork_result;

   if (pipe(file_pipes) == 0)
   {
         fork_result = fork();
         if (fork_result == 0)
         {
         sprintf(buffer, "%d", file_pipes[0]);
        (void)execl("pipe4", "pipe4", buffer, (char *)0);
        exit(EXIT_FAILURE);
        }
      else
         {
            write(file_pipes[1],"123",3);
         }
   }
}//main()
```



Pipe4.c:

```
int main(int argc, char *argv)
{
   char buffer[BUFSIZ + 1];
   int file_descriptor;
   sscanf(argv[1], "%d", &file_descriptor);
    read(file_descriptor, buffer, BUFSIZ);

   printf("read %s\n", buffer);
   exit(EXIT_SUCCESS);
}
```

Unnamed Pipe

<u>Disadvantages:</u>

– Only for related process.

– Pipe is useful only among processes related as parent/child.

<u>Solution:</u> Named Pipe

Named Pipe (FIFO)

- Named pipes have an entry in the file system.
  - Have ownership and permissions like any file.

Advantages:

– Two unrelated processes can use FIFO unlike plain pipe.

– Can be used for non-related processes

– No message boundaries. Data is treated as stream of byte.

Disadvantages

– Less secure than pipe, since any process with valid access permission can access them.

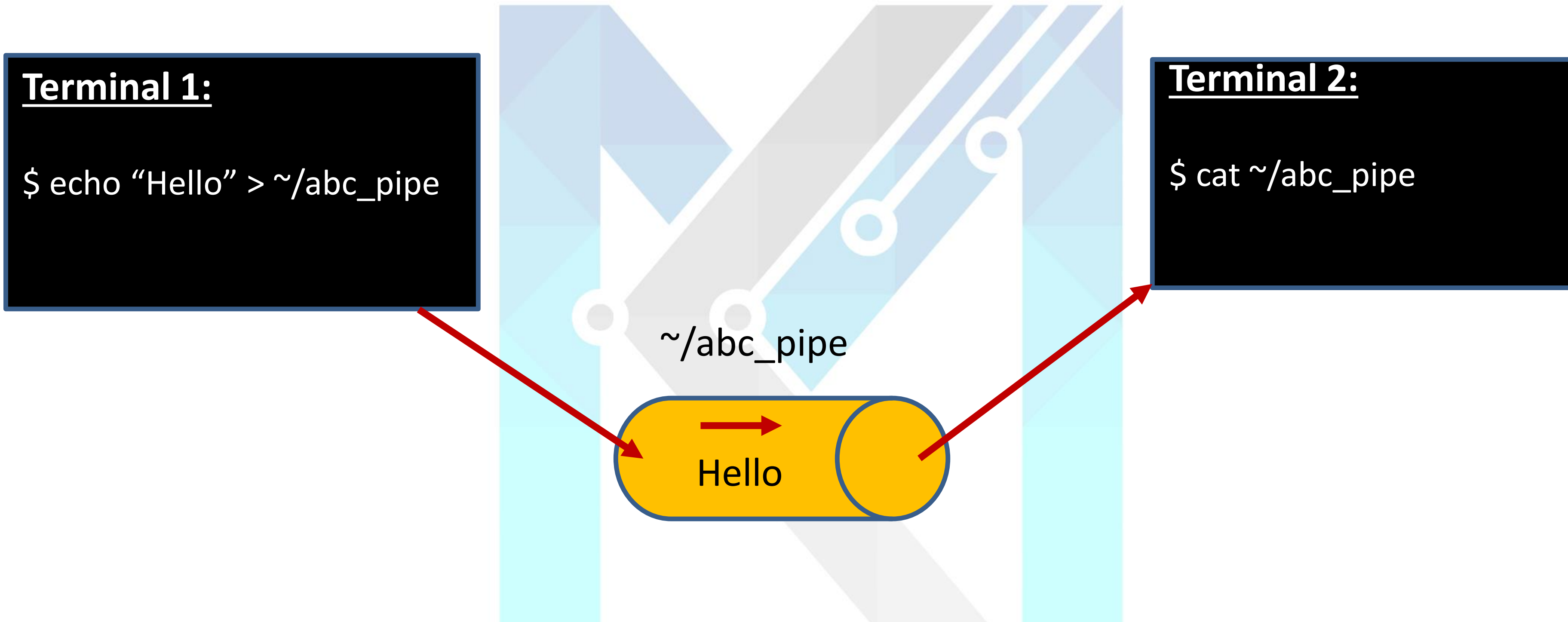
KERNEL MASTERS

Create a Named Pipe

Create a Named Pipe:

1. On the command line, created with **mknod** /tmp/mypipe.
2. mnod() system call
3. mkfifo() library call

Named Pipe Example



Connection oriented:
Writing or reading is possible only after connection is established between two processes.

Blocking Call vs Non Blocking Call

Blocking Call Example:

read() call will block; it will not return until a process opens the same FIFO for writing.

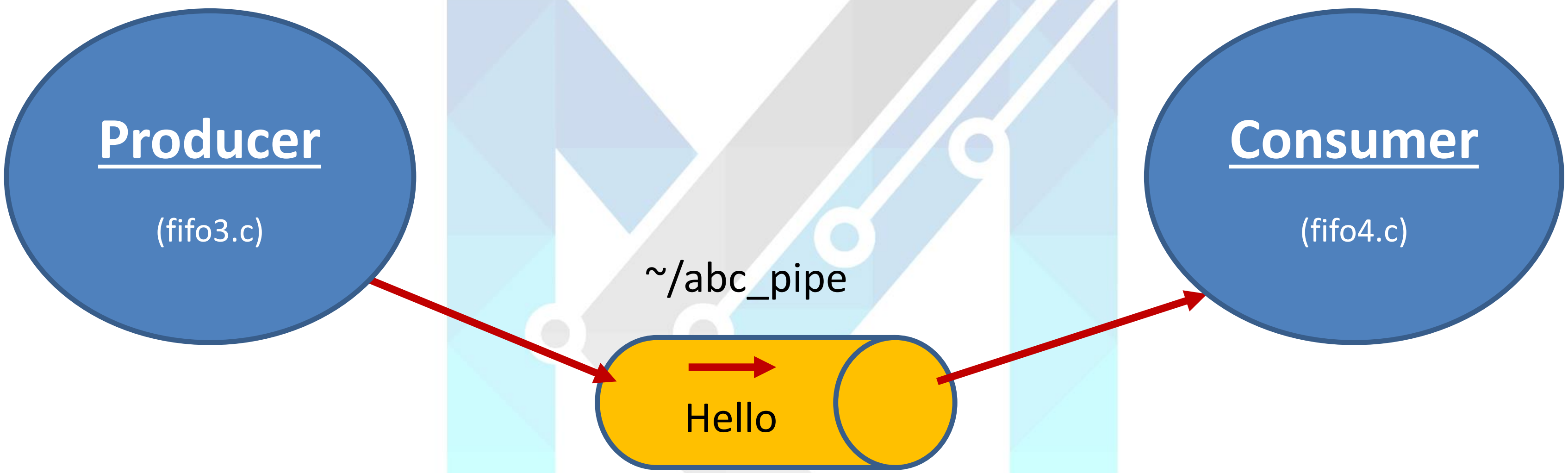
int fd; char buff[1024];
fd = open(“abc_pipe”,O_RDONLY);
read(fd, buff, sizeof(buff)); // read system call goes to blocking mode.

Non Blocking Call Example:

read() call will now succeed and return immediately, even if the FIFO has not been opened for writing by any process.

int fd; char buff[1024];
*fd = open(“abc_pipe”,O_RDONLY | **O_NONBLOCK**);*
read(fd, buff, sizeof(buff)); // read system call goes to non blocking mode.

Producer Consumer Problem using Named Pipes



Linux guarantees that at least **PIPE_BUF** bytes can be written to a pipe atomically.
(PIPE_BUF = 4096)

Message Queues

Connection less IPC Mechanism

Message Queues

Connection less: A process can send data without caring for the connection between two process. So sender can send message, even if no receiver is present is connection less mechanism.

Can direct and store data.

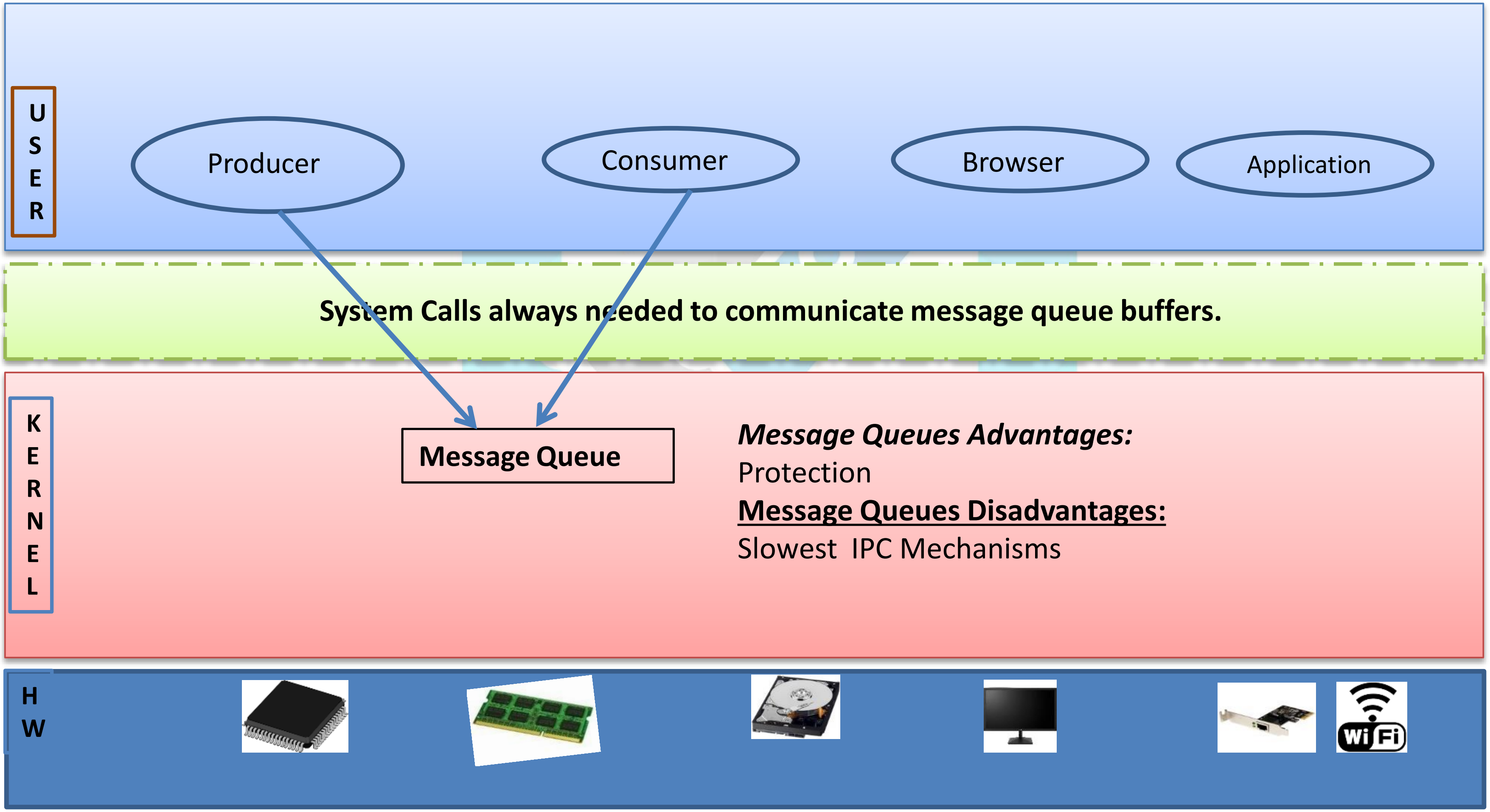
Advantages:

- Message queue is a dedicated IPC without any file management.
- Message queues and signals can be great for hard real-time applications, but they are not as flexible.
- Effective for small amount of data.

Drawbacks:

- Very expensive for large data, as transferred through kernel buffer and each data transfer involves 2 data copy operations.
- Message queue is a slowest IPC mechanisms.
- Message boundary is defined.
- Message is deleted after reading

IPC: Message Queues



KERNEL MASTERS

System V Message Queues

Create a message Queue

int msgget(key_t key, int msgflg);

Send and receive message queue

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

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

Message control operations

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

Shared Memory

Fastest IPC Mechanism

Shared Memory

- Fastest communication among all the IPC events.
- No synchronization like pipe or FIFO.
- Race condition is possible so locking is required.
- When next message will come with different id, then the previous message will be deleted.

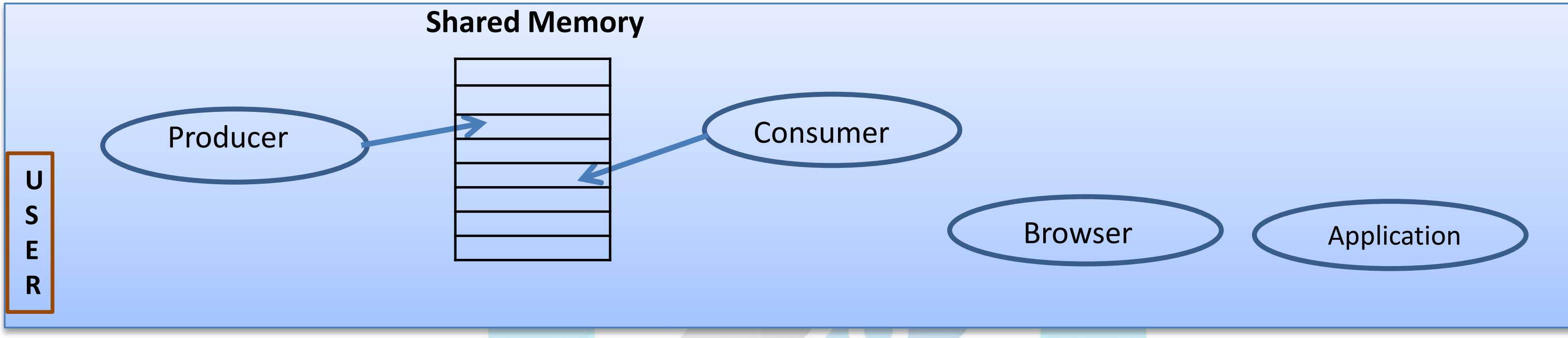
Advantages:

- This can be a very fast method of information transfer because RAM can be used.
- Data are not removed after reading.

Drawbacks:

- Synchronization is a major problem - if the first process keeps writing data,
- how can it ensure that the second one reads it?
- Data can either read or written only. Append is not possible

IPC: Shared Memory



System Calls always needed to communicate message queue buffers.

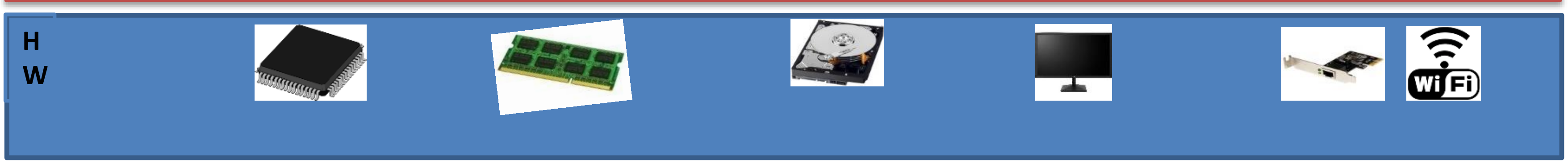
K E R N E L

<u>Shared Memory Advantages:</u>

Fastest IPC Mechanisms

<u>Shared Memory Disadvantages:</u>

Doesn’t Support Synchronization Mechanisms



System V Shared Memory

Create a shared memory

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

Attach/Dettach shared memory segment

void *shmat(int shmid, const void *shmaddr, int shmflg);

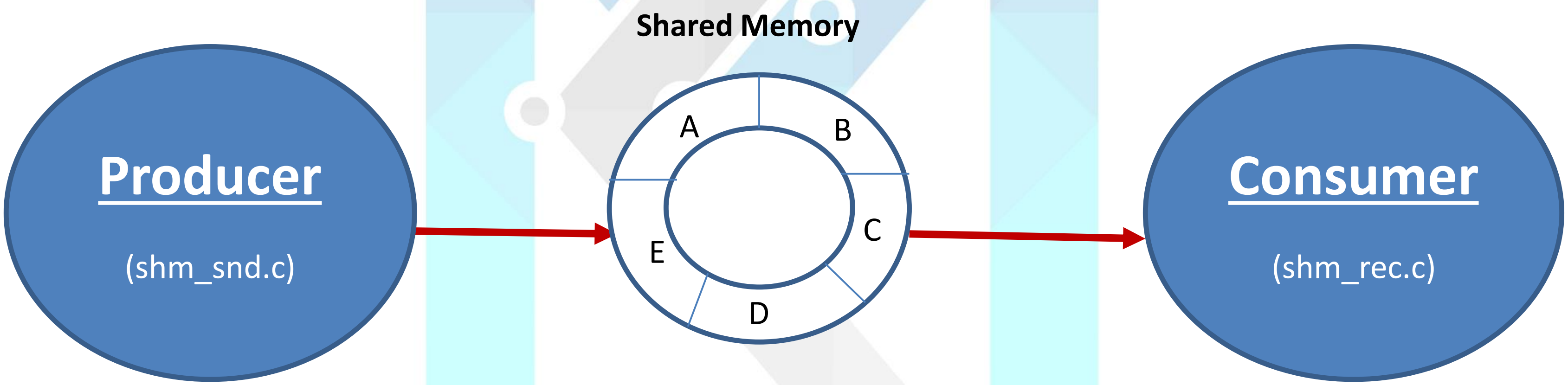
int shmdt(const void *shmaddr);

Shared memory control operations

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

Message Queues

Producer and consumer problem using shared memory without synchronization.



Shared memory doesn’t support synchronization Mechanism.
It’s the responsibility of the programmer to synchronize access.