Programming UNIX

Inter-process Communication

Overview

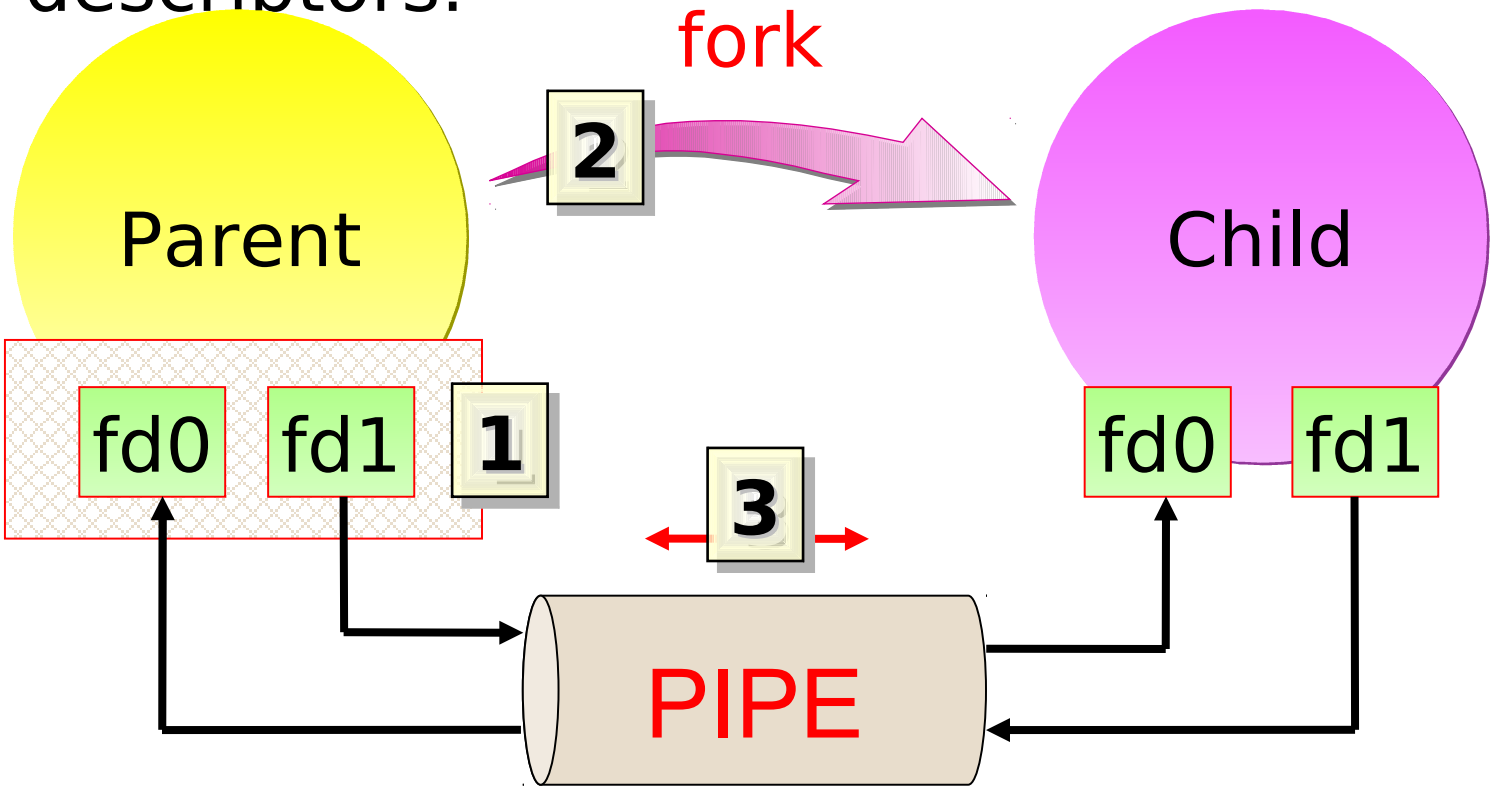
- In this module you will learn about:
  - **Pipes**
  - **FIFOs**
  - **Message Queues**
  - **Shared Memory**
  - **Semaphores**

Lesson: Pipes

- This lesson describes:
  - **Creating Pipes**
  - **Using Pipes**
  - **Using popen**
  - **Pipe Limitations**
  - **Filters**
  - **Coprocesses**

Creating a Pipe

- The function pipe creates a file with two descriptors:



Creating a Pipe

```
int fd[2];
pid_t   pid;

if (pipe (fd) < 0)                          1
      Error
if ((pid = fork()) < 0) {                   2
      Error
} else if (pid > 0) {
      close (fd[0]);
      write (fd[1], “Guten Tag\n", 10);     3
} else {
      close (fd[1]);
      n = read (fd[0], line, MAXLINE);      3
}
```

popen

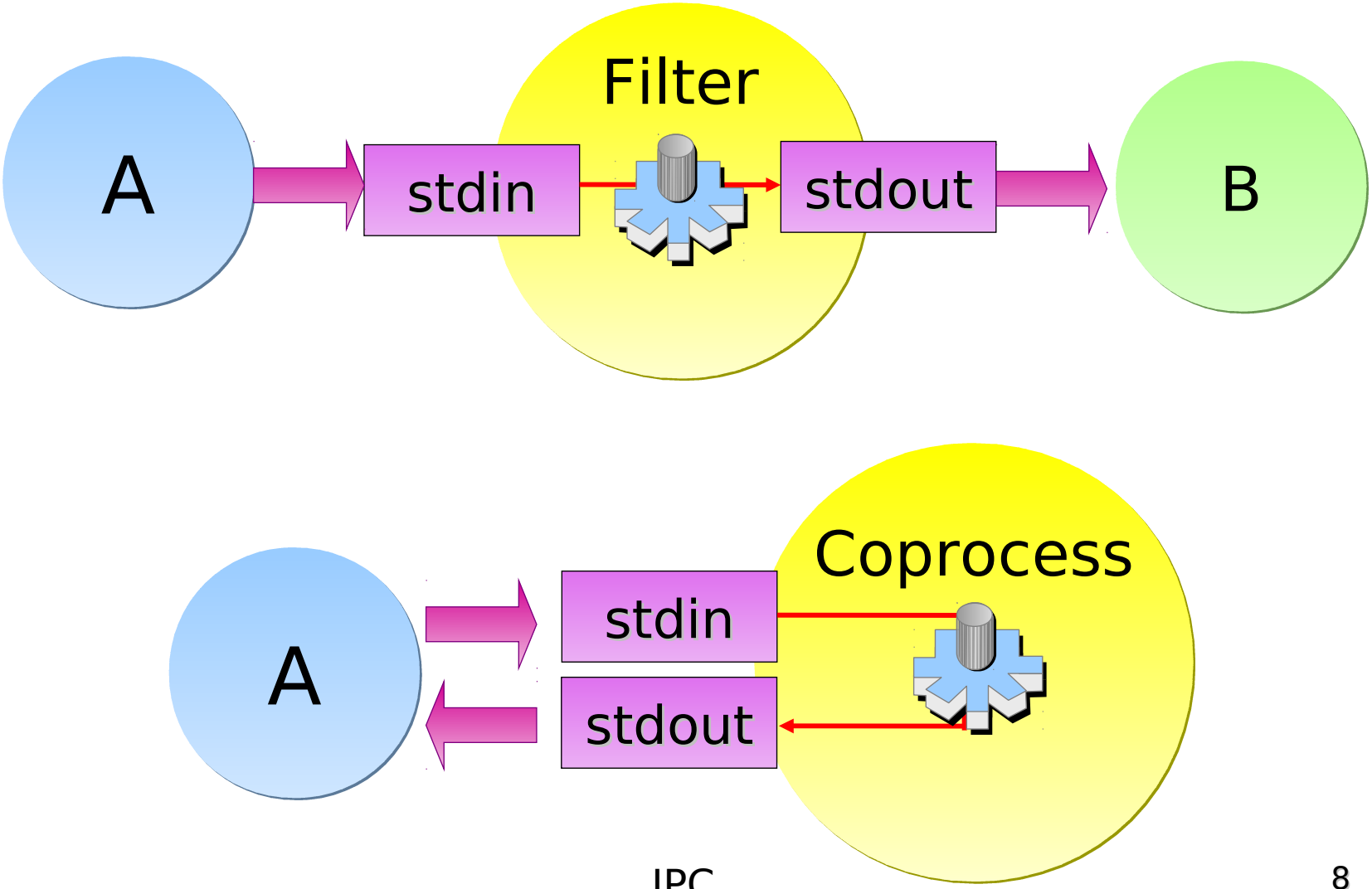
- The popen function simplifies work with pipes:
  - Creates a pipe
  - Forks a child
  - Closes unused ends of the pipe

```
#include <stdio.h>

FILE *popen (const char *cmdstring, const char *type);

int pclose (FILE *fp)
```

Pipe Limitations

- Pipes are the oldest form of IPC on UNIX.
- They have two limitations:
  - Pipes have been half duplex (1)
    - Data flows in only one direction
    - Some systems now provide full-duplex pipes
      - UNIX domain Sockets are full-duplex
  - Pipes can be used only between processes that have a common ancestor (2)
    - FIFOs and UNIX Domain Sockets can be established between processes not in the parent-child hierarchy

Filters and Coprocesses



Lesson: FIFOs

- This lesson describes:
  - **Creating FIFOs**
  - **Using FIFOs**

FIFOs

- FIFOs allow unrelated processes to exchange data.
  - The st_mode member of the stat structure

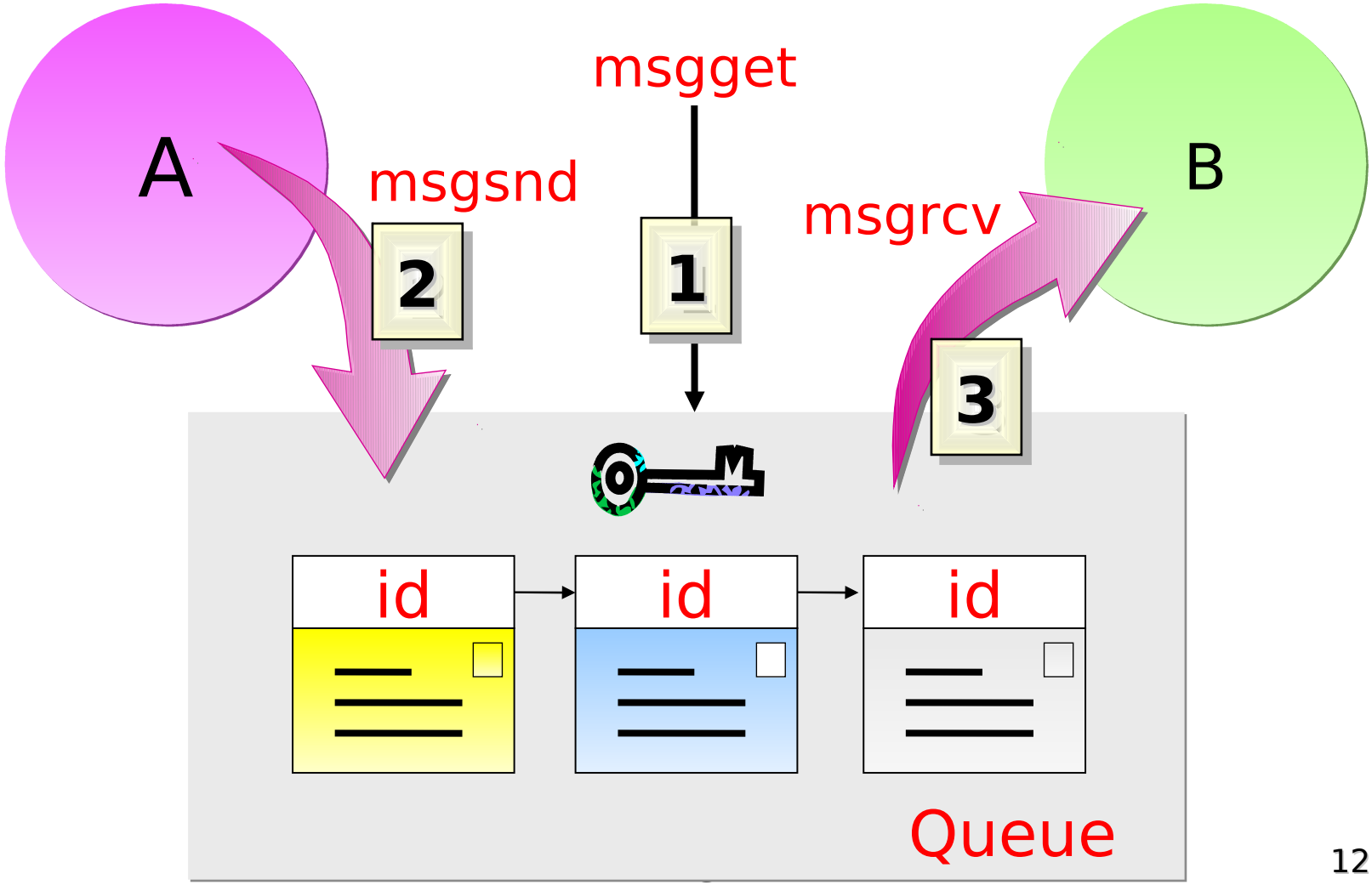
```
#include <sys/stat.h>

int mkfifo (const char *pathname, mode_t mode);
```

- FIFO is used for:
  - By shell commands to pass data from one shell pipeline to another without creating intermediate temporary files
  - As rendezvous points in client/server

Lesson: Message Queues

- This lesson describes:
  - **Creating Message Queues**
  - **Using Message Queues**

Mesage Passing



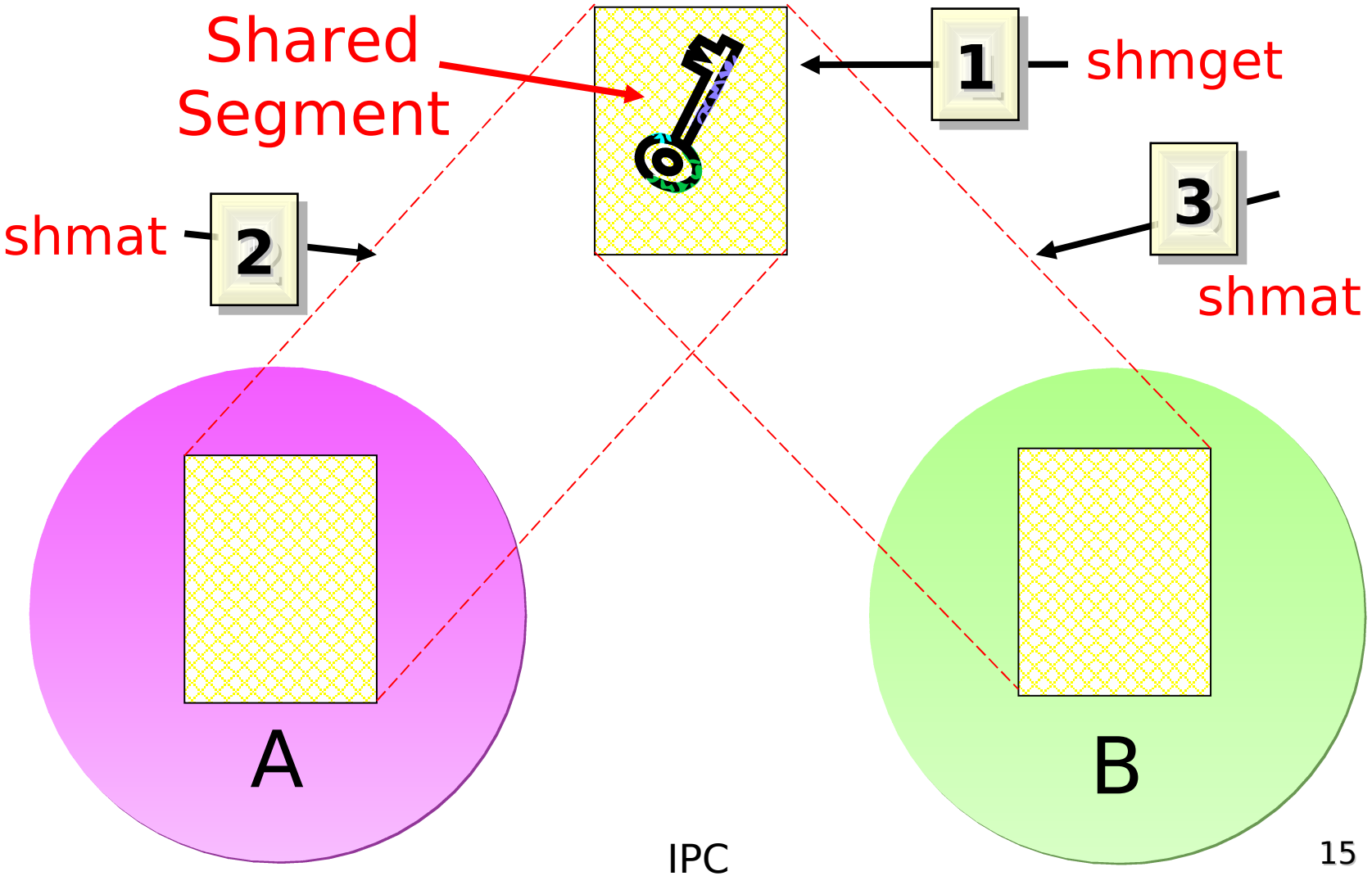
Message Queues

- A message queue is a list of messages stored in kernel
  - Each queue has ID called key
  - Each message has a message type.
- Operations on message queues
  - The msgget function - creates a new queue or opens an existing one
  - The msgsnd function - adds a message to the end of a queue.
  - The msgrcv function - fetches messages from a queue
    - we can fetch messages based on their type field

Lesson: Shared Memory

- This lesson describes:
  - **Creating Shared memories**
  - **Using Shared Memories**

Shared Memory



Shared Memory

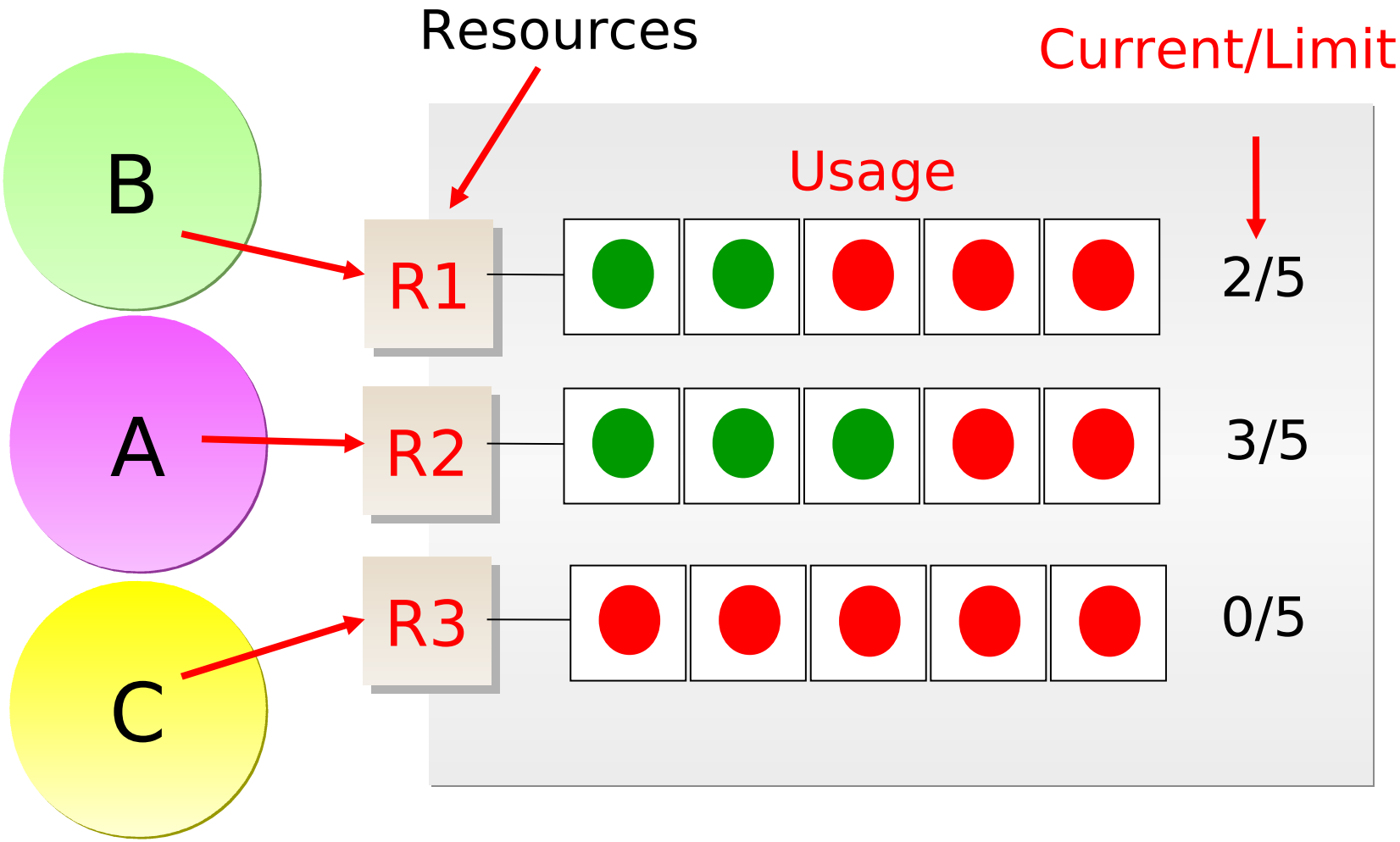
- Shared memory allows two or more processes to share a given region of memory
  - This is the fastest form of IPC
  - ! Accesses must be synchronized

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

int shmget (key_t key, size_t size, int flag);
int shmctl (int shmid, int cmd, struct shmid_ds *buf);
void *shmat (int shmid, const void *addr, int flag);
int shmdt (void *addr);
```

Lesson: Semaphores

- This lesson describes:
  - **Creating Semaphores**
  - **Restricting Accesses via Semaphores**

Semaphores



Semaphores

- A semaphore is a counter used to provide access to a shared data resource for multiple processes
  - You can specify set of operations on semaphore to be executed atomically

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

int semget (key_t key, int nsems, int flag);
int semctl (int semid, int semnum, int  cmd,
    ... /* union semun arg */);
int semop (int semid, struct sembuf semoparray[],  size_t nops);
```