System V IPC

Common Elements:

- Key: resource ID
- Creator: UID/GID
- Owner: UID/GID
- Permissions: r/w/x for owner/group/others

Resources are persistent and not automatically destroyed

Giorgio Richelli
giorgio.richelli@uniroma1.it

Semaphores

- A resource (variable) used for “signaling”
- No relationship with `signal()` IPC
- If a process is waiting for a <signal>, it is suspended until that <signal> is sent
- <wait> and <signal> operations cannot be interrupted (they are atomic)
- Queue is used to hold processes waiting on the semaphore

Giorgio Richelli
giorgio.richelli@uniroma1.it

Semaphore

```
int semget(key_t key, int count, int flag);
```

- Returns the identifier of semaphore <set> associated with key.
- `count:`
  - Number of semaphores in the <set>
- `key :`
  - `ftok()`
  - IPC_PRIVATE
- `flag :`
  - IPC_CREAT, ...
  - Access permissions (least 9 bits)

Giorgio Richelli
giorgio.richelli@uniroma1.it

Semaphore

```
int semop(int semid, struct sembuf *sops, unsigned
nsops);
```

- Performs operations on selected members of the semaphore set indicated by `semid`.
- Each of the `nsops` elements in the array pointed to by `sops` specifies an operation to be performed on a semaphore

```
 struct sembuf {
   unsigned short sem_num;
   short sem_op;
   short sem_flg;
}
```

Giorgio Richelli
giorgio.richelli@uniroma1.it

Semaphore

- The set of operations contained in `sops` is performed in array order, and atomically
- The operations are performed either as a complete unit, or not at all
- The behavior of the system call depends on the presence of the `IPC_NOWAIT` in the individual `sem_flg` field.

Giorgio Richelli
giorgio.richelli@uniroma1.it

Semaphore

- `unsigned short sem_num`
  - semaphore number (in set semid)
- `short sem_flg:`
  - `IPC_NOWAIT:` don't block, returns -1 and set errno to EAGAIN
  - `IPC_UNDO:` undo operation(s) when process exits

Giorgio Richelli
giorgio.richelli@uniroma1.it

Semaphore

- `short sem_op`
  - >0 : add `sem_op` to the value & eventually wake up suspended processes
  - 0 : block until value gets equal to 0 (unless `IPC_NOWAIT`)
  - <0 : block (unless `IPC_NOWAIT`) until the value becomes greater than or equal to the absolute value of `sem_op`, then subtract `sem_op` from that value

Giorgio Richelli
giorgio.richelli@uniroma1.it

Semaphore

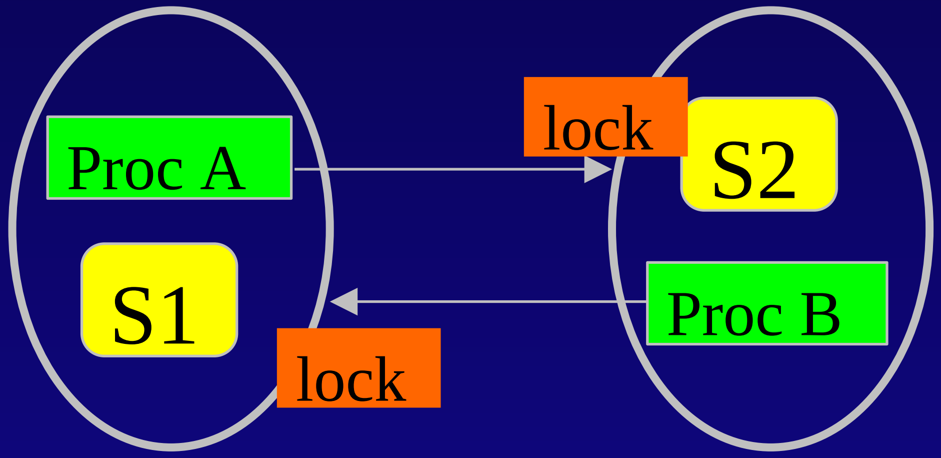
```
int semctl(int semid, int snum, int cmd, ...);
```

- Performs the operation specified by `cmd` on hore set identified by `semid`, or on the `snum-th` semaphore

- E.g.:
  - IPC_SETVAL/IPC_GETVAL: set, get the value of the semaphore
  - IPC_RMID: Remove semaphore set
  - ....

Giorgio Richelli
giorgio.richelli@uniroma1.it

DeadLock



Giorgio Richelli
giorgio.richelli@uniroma1.it

Producer/Consumer Problem

- One or more producers are generating data and placing these in a buffer
- One or more consumers are taking items out of the buffer one at time
- The buffer is must be kept coherent: only one producer or consumer may access the buffer at any given time

P/V Operations

Wait:

```
P() { s=s-1; if (s<0) block(); }
```

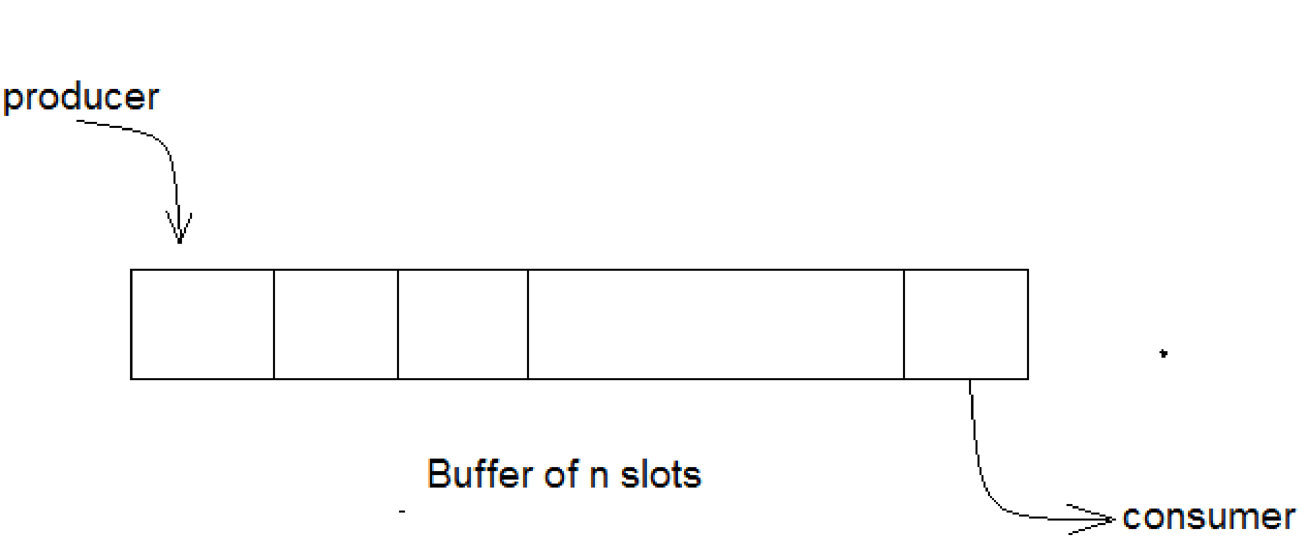
Signal:

```
V() { s=s+1; if (s>=0) wake(); }
```

Giorgio Richelli
giorgio.richelli@uniroma1.it

Producer/Consumer Problem

- How many semaphores ?



Giorgio Richelli
giorgio.richelli@uniroma1.it

Producer/Consumer Problem

- We need three semaphores:
  - Amount of items in the buffer
  - Number of free entries in the buffer
  - Right to use the buffer

Giorgio Richelli
giorgio.richelli@uniroma1.it

Producer Function - Pseudocode

```
semaphore s=1, n=0, e=SIZE
void producer(void)
{
   while (1){
      produce_item();
      P(e);
      P(s);
      enter_item();
      V(s);
      V(n);
   }
}
```

Giorgio Richelli
giorgio.richelli@uniroma1.it

Consumer Function - Pseudocode

```
semaphore s=1, n=0, e=SIZE
void consumer(void)
{
   while (1){
      P(n);
      P(s);
      remove_item();
      V(s);
      V(e);
   }
}
```

Giorgio Richelli
giorgio.richelli@uniroma1.it

Readers/Writers

- Two kinds of threads: readers and writers.
  - Readers can inspect items in the buffer, but cannot change their value.
  - Writers can both read the values and change them.
- The problem allows any number of concurrent reader threads, but a writer thread must have exclusiver access to the buffer.

Giorgio Richelli
giorgio.richelli@uniroma1.it

Readers/Writers

```
Writer()
{
   while (1) {
      P(writing);
      <<< perform write >>>
      V (writing);
   }
}
```

Giorgio Richelli
giorgio.richelli@uniroma1.it

Readers/Writers

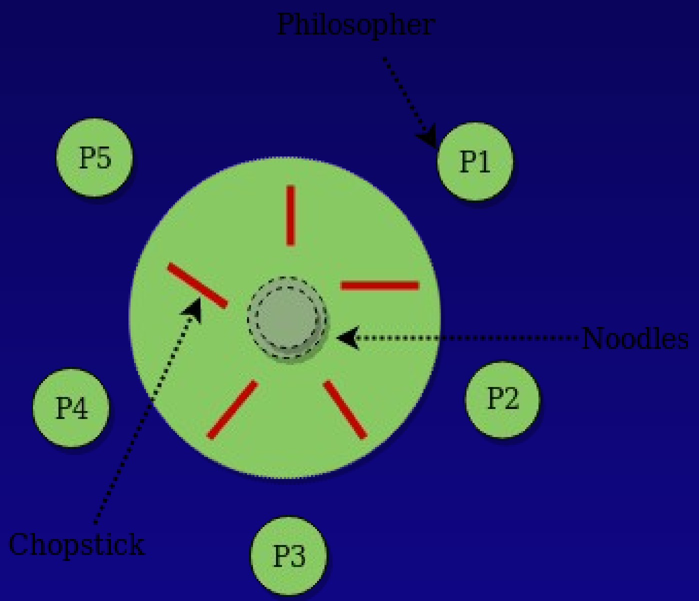
```
Reader()
{
   while (1) {
      P(mutex);
      rd_count++;
      if (rd_count==1) P(writing); /* First reader gets the write lock */
      V(mutex);
      <<< perform read >>>
      P(mutex)
      rd_count--;
      If (!rd_count) V(writing); /* Last reader unlocks writers */
      V(mutex);
    }
}
```

Giorgio Richelli
giorgio.richelli@uniroma1.it

Dining Philosopers

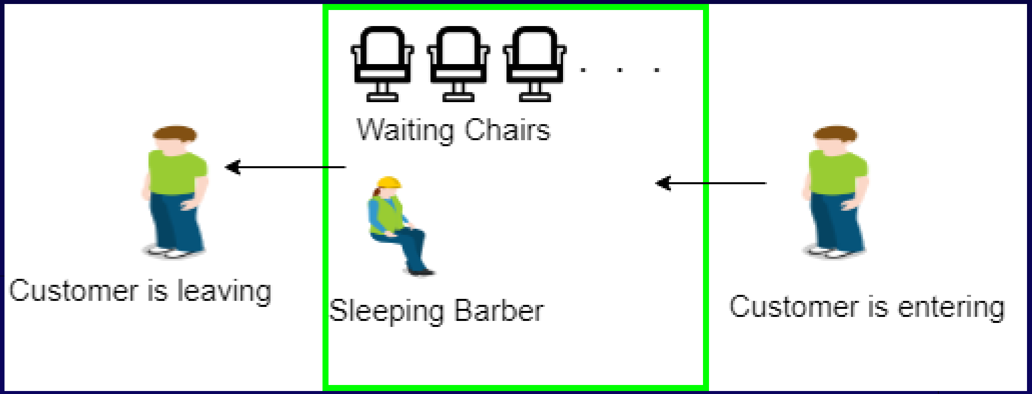
- K philosophers are seated around a circular table with one chopstick between each pair of philosophers.
- There is one chopstick between each philosopher.
- A philosopher may eat if he can pickup the two chopsticks adjacent to him.
- One chopstick may be picked up by any one of its adjacent followers but not both.

Source: www.geeksforgeeks.org



Giorgio Richelli
giorgio.richelli@uniroma1.it

Sleeping Barber

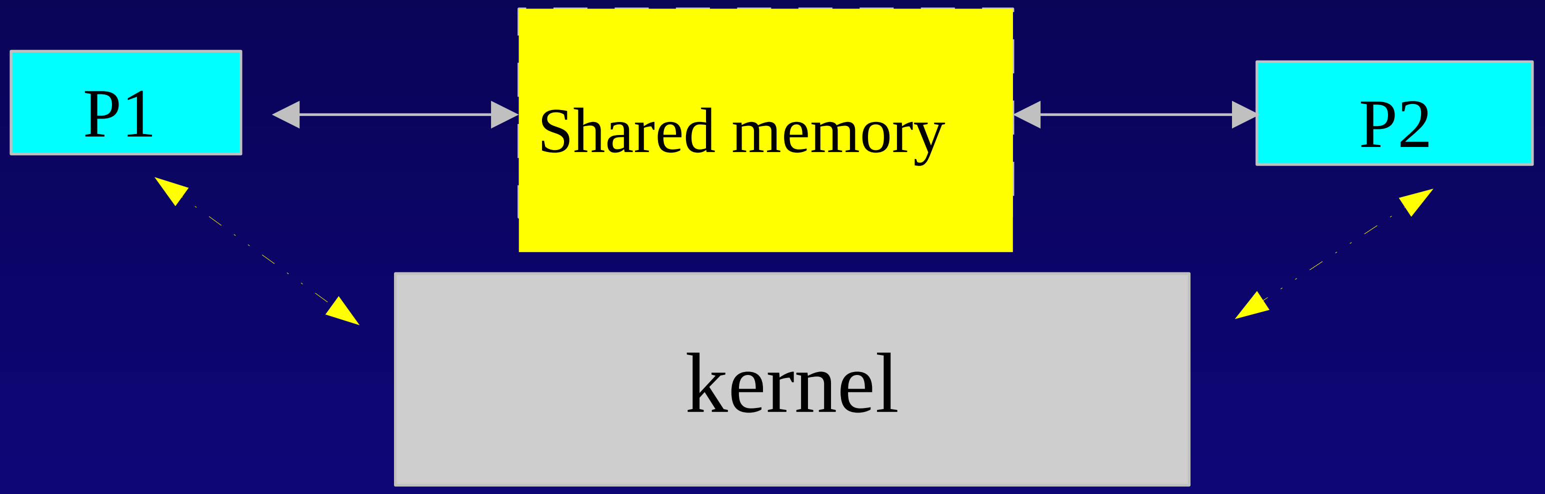


Source: www.geeksforgeeks.org

- The barber shop has:
  - one barber
  - one barber chair
  - N chairs for waiting for customers .
- If there is no customer, then the barber sleeps in his own chair.
- When a customer arrives, he wakes up the barber.
- If there are many customers and the barber is cutting a customer's hair, then the remaining customers either wait if there are empty chairs in the waiting room or they leave if no chairs are empty.

Giorgio Richelli
giorgio.richelli@uniroma1.it

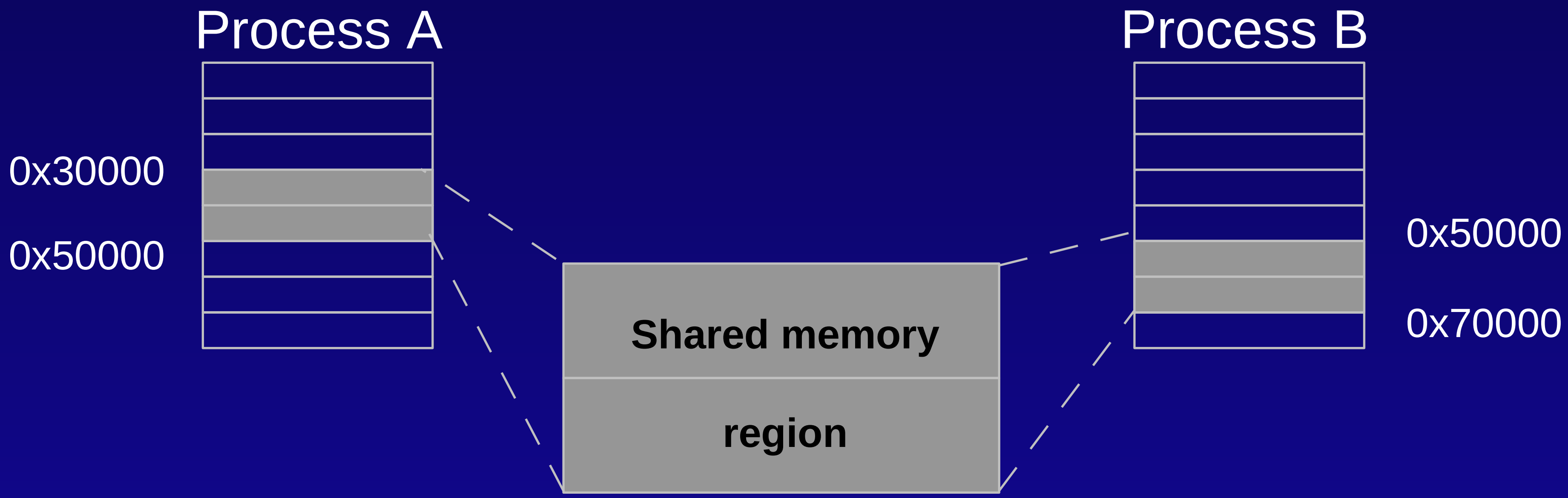
IPC with shared memory



Giorgio Richelli
giorgio.richelli@uniroma1.it

Shared Memory

A portion of physical memory shared between multiple processes.



Giorgio Richelli
giorgio.richelli@uniroma1.it

Shared Memory API

```
int shmget(key_t key, size_t size, int flag);
```

- returns the identifier of the shared memory segment associated with key
    - `key`: IPC_PRIVATE, ...
    - `size`: size of shared area
    - `flag`: IPC_CREATE, permissions, ..

Giorgio Richelli
giorgio.richelli@uniroma1.it

Shared Memory

Shared memory segments are:

- inherited after fork()
- detached but not destroyed, after `exec()` or `exit()`

`Use specific command for manage resources:`

- `ipcs, ipcrm, ..`

Giorgio Richelli
giorgio.richelli@uniroma1.it

Shared Memory API

```
void *shmat(int shmid, void * shmaddr, int shmflag);
```

- attaches the shared memory segment identified by `shmid` to the address space of the calling process
- does not modify the `brk`

- `shmaddr` : usually NULL, otherwise address requested for segment
- `shmflag`: SHM_RDONLY, SHM_RND, ...

Giorgio Richelli
giorgio.richelli@uniroma1.it

Shared Memory API

```
int shmdt(void *shmaddr);
```

- Detaches the shared memory segment at `shmaddr` from address space of calling process.

Giorgio Richelli
giorgio.richelli@uniroma1.it

Shared Memory API

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

- performs operation indicated by cmd on shared memory segment identified by `shmid`

- `cmd`: IPC_RMID, ...
- `buf`: address of struct to hold information about segment

Giorgio Richelli
giorgio.richelli@uniroma1.it

Shared Memory API

- Shared memory segments must be explicitly removed (IPC_RMID)
- The segment is then marked as removed, but it will be destroyed only when the last process call `shmdt()`

- So it is common to:
  - create the segment (one process)
  - map the shared memory region (all processes)
  - remove the segment (one process)
- In order to avoid to leave unused segments, e.g. in case of crashes

ftok

IPC key can be correlated to a file name

```
key_t ftok(char *pathname, int ndx)
```

- builds a key based on:
  - `pathname:` an existing, accessible file
  - `ndx:` least significant 8 bits
-

Giorgio Richelli
giorgio.richelli@uniroma1.it

Security

If a process holds the key, it might access the resource.

Giorgio Richelli
giorgio.richelli@uniroma1.it

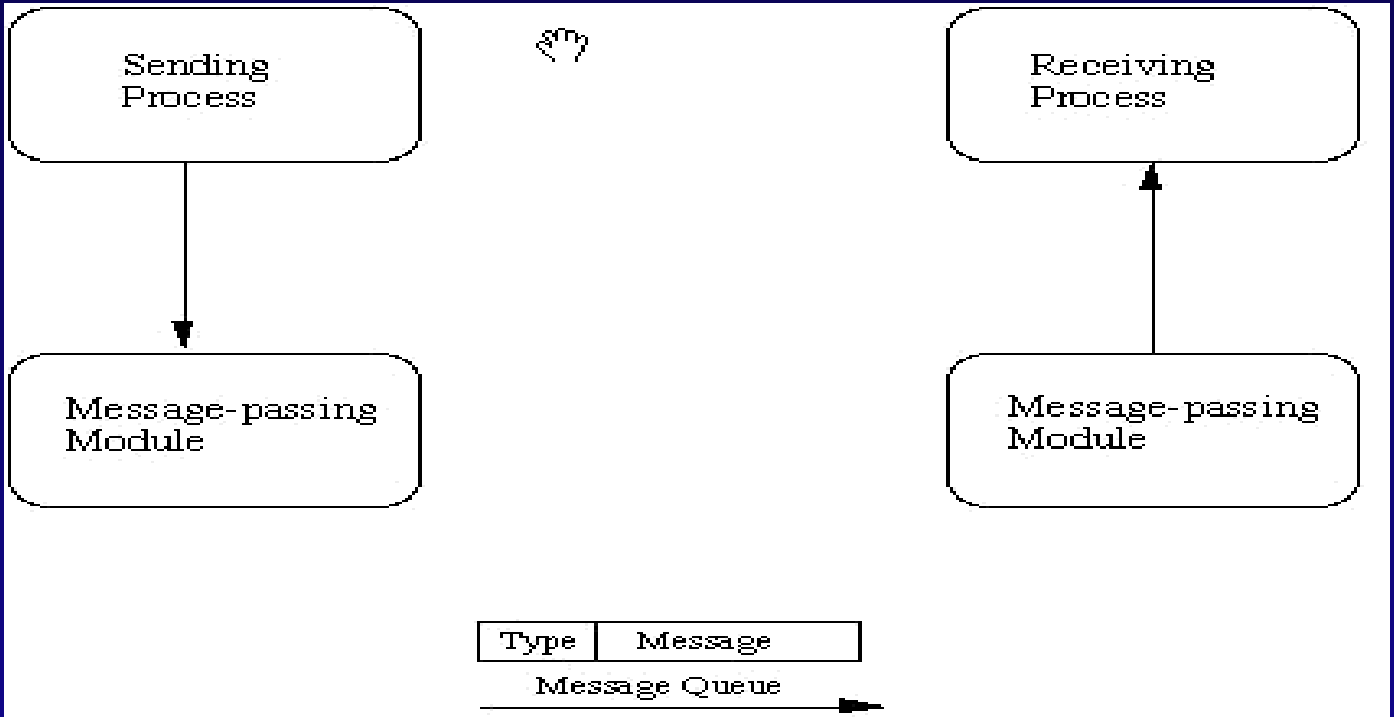
Message Queues

- Somewhat similar to pipes, but (unlike pipes):
  - processes can send and receive messages in an arbitrary order
  - each message has an explicit length
  - messages can be assigned a specific type

- However, they are not much used in the real world

Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queues



Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue

```
int = msgget(key_t key, int flag);
```

returns the message queue identifier associated with the value of the key argument

- `key: IPC_PRIVATE, ..`
- `flag: IPC_CREAT, ...`

Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue

```
int msgsnd(int msgqid, struct msgbufp *msgp, size_t
size, int flag)
```

appends a copy of the message pointed to by msgp to the message queue whose identifier is specified by msqid

- `msgqid: message queue identifier`
- `msgp, size: pointer and size of message to send`
- `flag: IPC_NOWAIT, ..`

Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue

```
struct msgbuf {
      long mtype;    /* message type */
      char mtext[MSGSZ]; /* message text of length MSGSZ */
};
```

Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue

```
count =msgrcv(int msgqid, struct msgbuf *msgp, size_t size, long type, int flag)
```

reads a message from the message queue specified by msqid into the buffer pointed to msgp

- `size: maximum size (in bytes) for the mtext member of msgp`
- `type: 0, [type], - [type]`
- `flag: IPC_NOWAIT, MSG_NOERROR, MSG_EXCEPT`

Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue

- If msgtyp:
  - ==0 → the first message in the queue is read.
  - >0 → the first message in the queue of type `msgtyp` is read, unless `msgflg==MSG_EXCEPT`, in which case the first message in the queue of type not equal to `msgtyp` will be read.
  - <0 → the first message in the queue with the lowest type less than or equal to the absolute value of `msgtyp` will be read.

Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue

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

performs the control operation specified by cmd on the message queue with identifier msqid

- `msgqid: msg queue identifier`
- `cmd: IPC_RMID, ….`
- `buf: address of buffer`

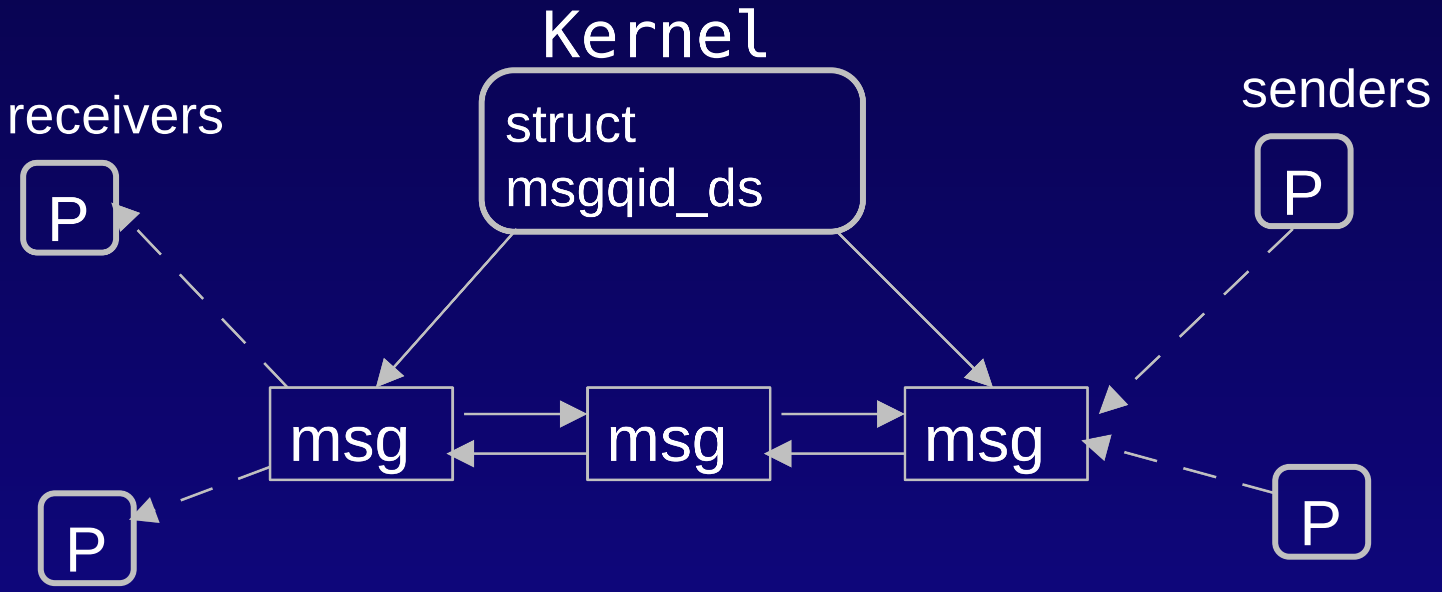
Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue

```
struct msqid_ds {
  struct ipc_perm msg_perm;     /* Ownership and permissions */
  time_t     msg_stime;    /* Time of last msgsnd(2) */
  time_t     msg_rtime;    /* Time of last msgrcv(2) */
  time_t     msg_ctime;    /* Time of last change */
  unsigned long   __msg_cbytes; /* Current number of bytes in queue */
  msgqnum_t msg_qnum;    /* Current number of messages in queue */
  msglen_t  msg_qbytes;  /* Maximum number of bytes allowed in queue */
  pid_t     msg_lspid;  /* PID of last msgsnd(2) */
  pid_t     msg_lrpid;    /* PID of last msgrcv(2) */
};
```

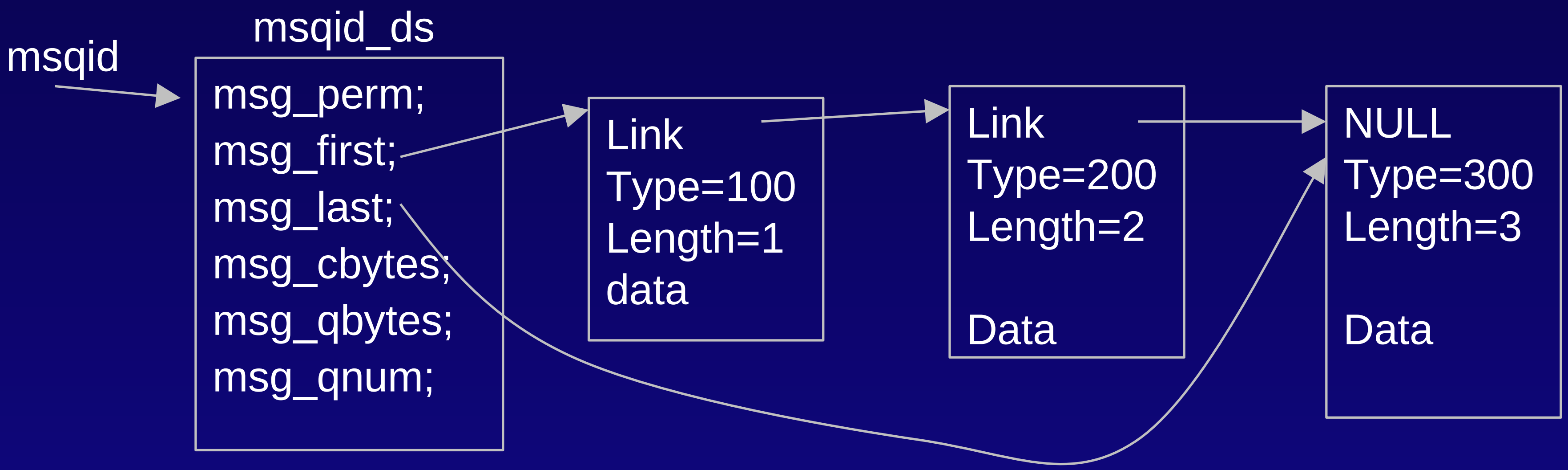
Giorgio Richelli
giorgio.richelli@uniroma1.it

Message Queue



Giorgio Richelli
giorgio.richelli@uniroma1.it

Kernel Message Queue Data Structure



Giorgio Richelli
giorgio.richelli@uniroma1.it