# Concurrent programming

IPC Inter-process communication
Semaphores

# Semaphores

Semaphores are a primary concept in the problems of synchronization. They are not used to exchange information, but to synchronize access to resources

# Semaphores

An example of a P(S) semaphore operation in pseudo code:

```
for (;;)
   {
   if (Semaphor > 0){
      Semaphor--;
      break;}
   }
```

Unfortunately, there is no guarantee of indivisibility here. There is also active checking.

# Semaphores

The kernel maintains some information structure for each set of semaphores on the system (structure example).

```
#include <sys/types.h>
#include <sys/ipc.h> /* ipc_perm structure
definition*/
struct semid_ds
   {
   struct ipc_perm sem_perm;
   struct sem *sem_base; /* pointer to the first
semaphore in the set */
   ushort sem_nsems; /* number of semaphores */
   time_t sem_otime; /* time of the last operation
*/
   time_t sem_ctime; /* time of the last change */
   };
```

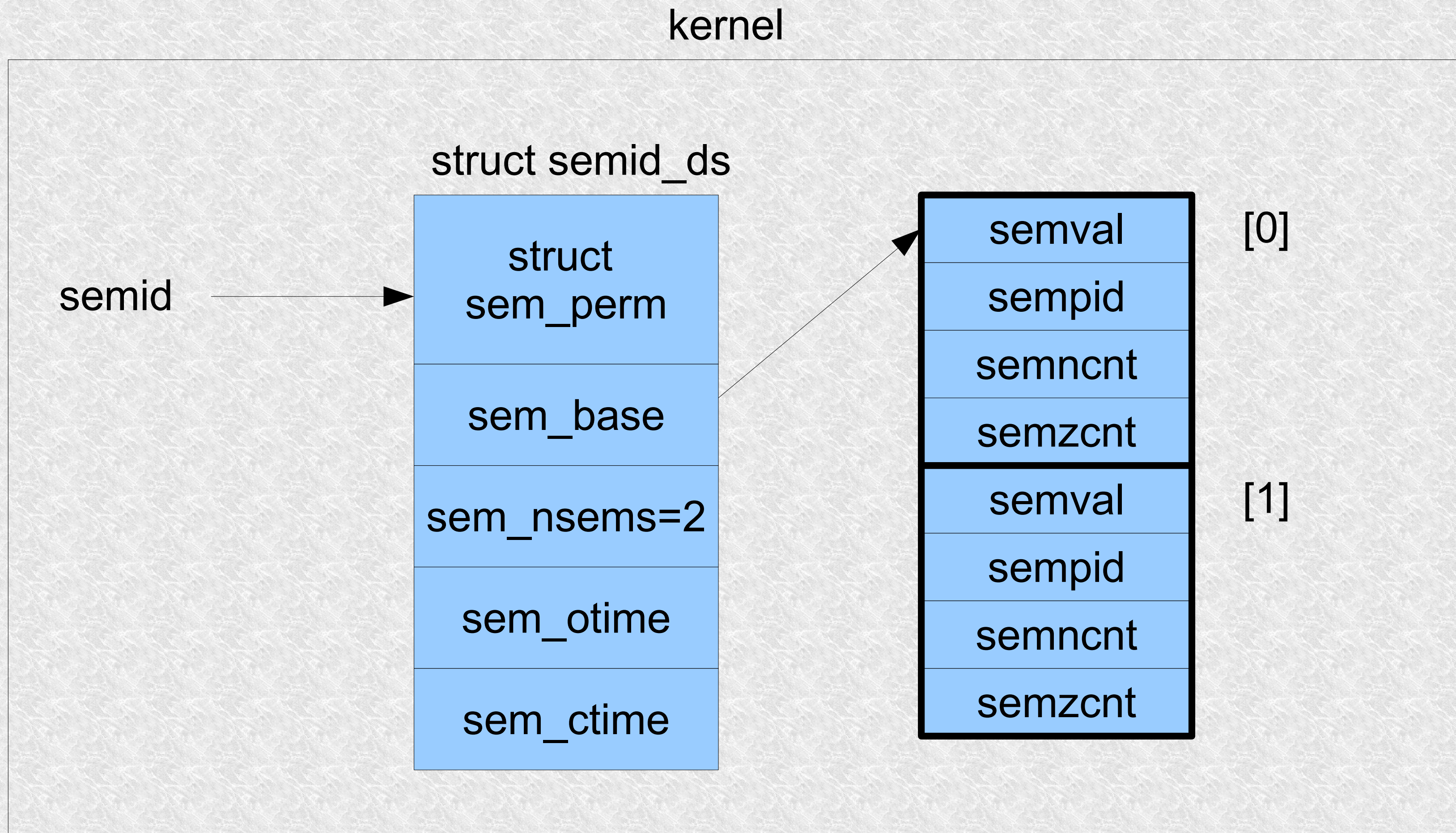
# Semaphores

sem is the internal data structure

```
struct sem {
   ushort semval; /* non-negative semaphore value
*/
   short sempid; /* the process id for the last
operation */
   ushort semncnt /* number of pending semaphore
values> current value */
   ushort semzcnt; /* number of waiting semaphore
values = 0 */
```

# Semaphores

**An example of the structure of a 2-element semaphore**



# Semaphores

Use a function to create or open a semaphore

```
int semget (key_t key, int nsems, int semflag);
```

- returns the semaphore id or –1
- *nsem* – how many semaphores are in the set, if we do not create but open a given set of semaphores, you can give 0 here
- the number of semaphores in the created set cannot be changed
- *semflag* – is a combination of the following symbolic constants:

# Semaphores

| Numerical value | symbolic constant | meaning |
|---|---|---|
| 0400 | *SEM_R* | reading by the owner |
| 0200 | *SEM_A* | change by owner |
| 0040 | SEM_R >> 3 | reading by the group |
| 0020 | *SEM_A* >> 3 | changing by group |
| 0004 | *SEM_R* >> 6 | reading by others |
| 0002 | *SEM_A* >> 6 | changing by others |
| 1000 | *IPC_CREAT* | |
| 2000 | *IPC_EXCL* | |

# Semaphores

The function semop is used to perform operations on semaphores

*int semop(int semid, struct sembuf *opstr, unsigned int nops);*

- returns 0 on success, or -1 on error

- *semid* – semaphore identifier
- *nops* – number of elements in the *sembuf* structure array pointed to by opstr
- *opstr* - points to an array of the following structures:

```
struct sembuf {
   ushort sem_num; /* semaphore number*/
   short sem_op; /* ooperation on the semaphore
*/
   short sem_flag; /* operation flag */
};
```

# Semaphores

- each element of this array specifies an operation on the value of one semaphore from the set of semaphores
  - *sem_num* – determines which semaphore (counting from 0)
  - *sem_op*
    - *>0 add this value to the current semaphore (release resources) operation V(s)*
    - *= 0 the calling process semop wants to wait until the semaphore value becomes zero.*
    - *<0 the calling process waits for the value of the semaphore to become greater than (or equal to) the absolute value of this field. Then they will be summed, i.e. resource allocation operation P (s). Np. s=1+(-1). s=0 semaphore down.*
  - *sem_flg – has several options, e.g. undoing changes made by the process on this semaphore if the process "crashes"*
    - *IPC_NOWAIT to sem_flg tells the system that we do not want to wait for the operation to be completed.*

# Semaphores

The function is used for controlling operations on the semaphore

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

- semun – is constructed as follows

```
union semun {
int val; /* only used for SETVAL */
struct semid_ds. *buff; /* used for IPC_STAT and IPC_SET */
ushort *array; /* used for GETVAL and SETVAL */
} arg;
```

- cmd – command
  - IPC_RMID – removing a semaphore
  - GETVAL – get the value of a semaphore, the function will return its value
  - SETVAL – setting the value to the semophore val in the semun union;
- semnum – which semaphore is concerned

# Semaphores

Seizing resources using semaphores

- semaphores can be treated as a synchronization mechanism
- suppose that the value of semaphore 1 will be the resource busy and 0 the resource free
- this assumption is opposite to our semaphore logic but in some systems it is impossible to initialize a semaphore other than 0

# Semaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
#define SEMKEY 123456L /* key value for the system function
semget() */
#define PERMS 0666

static struct sembuf op_lock[2] = {
   0,0,0, /* wait until semaphore 0 becomes zero */
   0,1,0 /* then increase that semaphore 1*/
};

static struct sembuf op_unlock[1] = {
   0, -1, IPC_NOWAIT /* decrease semaphore 0 by 1 without
waiting because it is release resources */
};

int semid = -1; /* semaphore identifier */
```

# Semaphores

```
my_lock()
{
if (semid <0)
    {
    if ((semid = semget(SEMKEY, 1, IPC_CREAT| PERMS)) < 0)
       perror(„error creating semaphore”);
    }
if (semop(semid, &op_lock[0], 2) < 0)
    perror(„error of seizing a semaphore”);
}

my_unlock()
{
if (semop (semid, &op_unlock[0], 1) < 0)
    perror(„unlocking error”);
}
```

# Semaphores

What if any of the processes "fails"?

- You can "arm" the process occupying the semaphore to handle most signals that may come, and elegantly release the semaphore on each call. Unfortunately, this does not work for SIG_KILL

# Semafory

- *my_lock can determine the IPC_NOWAIT flag in the first operation in the op_lock array. If the operation function returns -1 and errno = EAGAIN, the process may call semctl and examine the value of the sem_ctime field of the semid_ds structure. for that sempore.* If it turns out that a predetermined time has elapsed (e.g. 10s) since the last change, the process may take the resource, assuming that another process no longer needs it, and has forgotten or failed to release it

The disadvantage is that you have to constantly call additional functions when the resource is busy and you have to accept a TIMEOUT

# Semaphores

- The third best solution is to notify the kernel when seizing a resource that if the process terminates before the resource is released, the kernel is to release it.

# Semaphores

***Semaphore Adjustment Value*** **–** for each semaphore value in the system, you can specify a second associated value.

- When you specify the initial value of a semaphore, then set the value to set that semaphore to 0.
- For each operation used in the call to the semop function and with the SEM_UNDO flag set, if the value of a semaphore is increased, the semaphore adjustment value will also be decreased by the same amount.
- When the process finishes then the kernel will automatically use all adjustment values for that process. (will be summed with the value of the semaphore).

# Semaphores

An example for the previous seizng:

```
static struct sembuf op_lock[2] = {
   0,0,0,
/* wait until semaphore 0 becomes zero */
   0,1,SEM_UNDO
/* then increase that semaphore  1*/
};

static struct sembuf op_unlock[1] = {
   0, -1, IPC_NOWAIT | SEM_UNDO
   /* decrease semaphore 0 by 1 without waiting because it
is freeing resources */
};
```

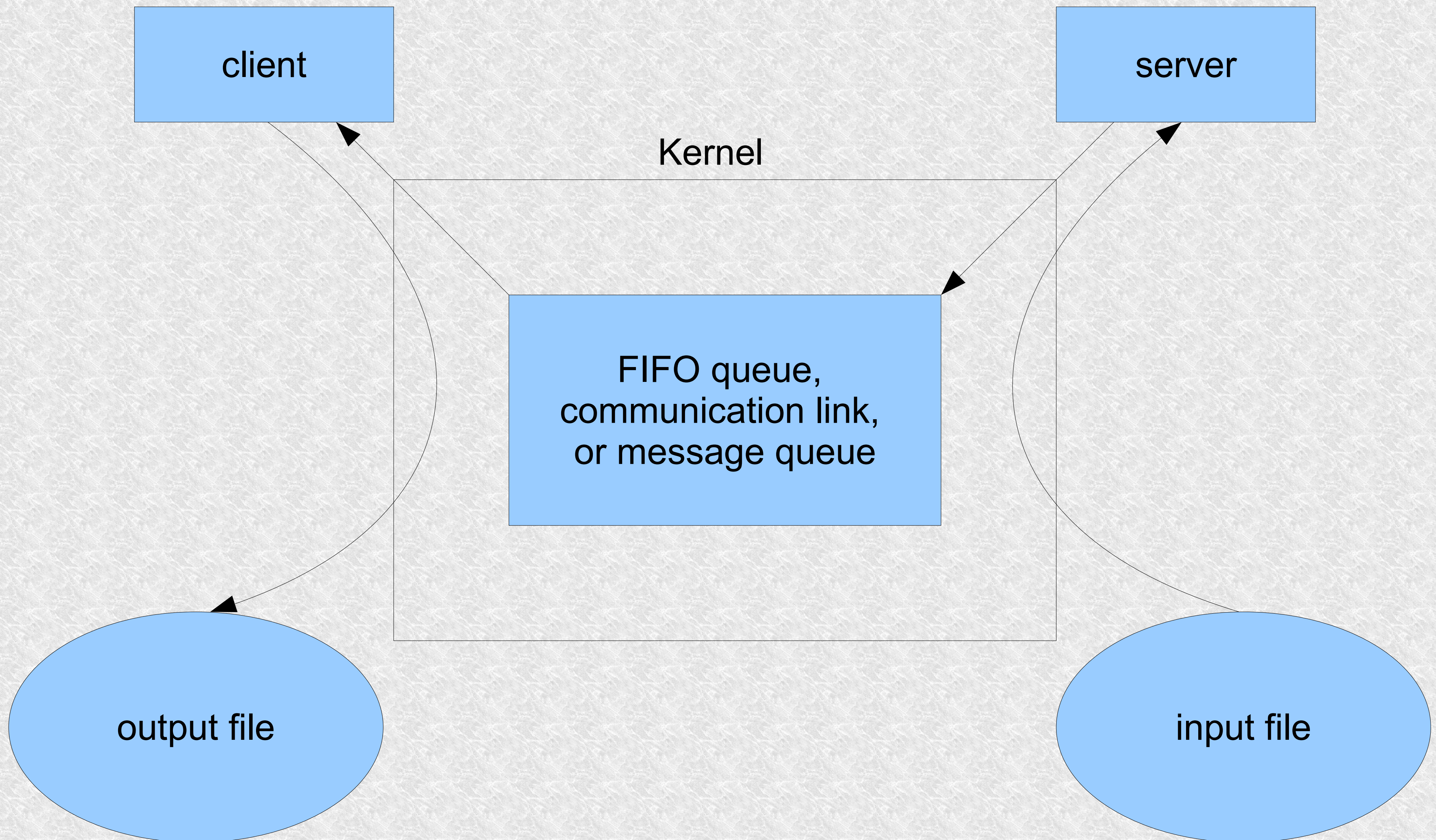
Also, remember that the last process using the semaphore should remove it from the system

# Shared Memory

Once again the program (client-server) that copies files

- The server reads a file, usually the kernel copies from disk to some buffer
- from this buffer it goes to our server's buffer specified as the second argument of the read function
- The server writes this data to a FIFO, unnamed link or message queue, again it is copying from the user buffer to the kernel
- The client reads data from the IPC channel, it requires a copy from the kernel buffer to the client's buffer
- finally copies from client buffer to output buffer (write)
- and from the output buffer e.g. to the screen.

# Shared Memory



# Shared Memory

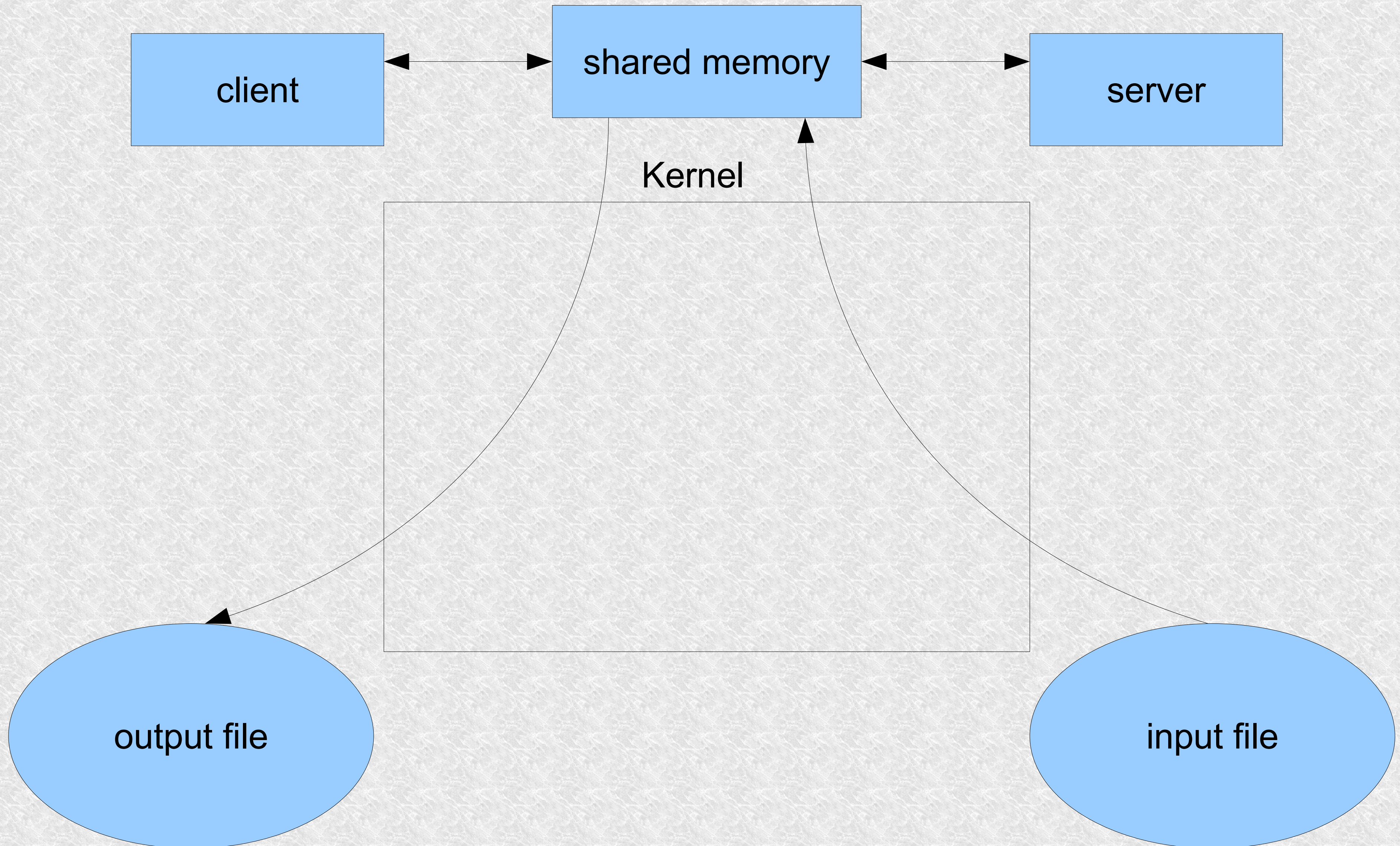
- Shared memory avoids the inconvenience of too many data copies by allowing two or more processes to use the same memory segment.
- Sharing memory is similar to using a shared file. Additional synchronization mechanisms, e.g. semaphores, must be used.

# Shared Memory

The algorithm of action will be as follows:

- The server accesses the shared memory segment using a semaphore
- The server reads to shared memory
- After reading is complete, the server notifies the client using a semaphore that the data is ready to be retrieved from memory
- The client reads from shared memory and writes to the result file (for example stdout)

# Shared Memory



# Shared Memory

for each shared memory segment, the system kernel maintains the following information structurei:

```
struct shmid_ds {
struct ipc_perm shm_perm; / * structure of access
rights to operations */
int shm_segsz; /* segment size */
struct XXX shm_YYY; /* implementation dependent on
information */
ushort shm_lpid; /* the process id for the last
operation */
ushort shm_cpid; /* Process ID creator */
ushort shm_nattch; /* attached running number */
ushort shm_cnattch; /* attached number in internal
memory */
time_t shm_atime; /* time of last attachement */
time_t shm_dtime; /* time of last disattachement */
time_t shm_ctime; /* time of the last change */
};
```

# Shared Memory

**a function is used to create a shared memory segment:**

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget (key_t key, int size , int shmflag);
```

- returns the memory identifier or -1 on error
- *key* key created by ftok or invented by us
- size – memory size in bytes
- shmflag – combination of tags

# Shared Memory

| Numerical value | symbolic constant | meaning |
|---|---|---|
| 0400 | *SHM_R* | reading by the owner |
| 0200 | *SHM_A* | change by owner |
| 0040 | SHM_R >> 3 | reading by the group |
| 0020 | *SHM_A* >> 3 | changing by group |
| 0004 | *SHM_R* >> 6 | reading by others |
| 0002 | *SHM_A* >> 6 | changing by others |
| 1000 | *IPC_CREAT* | |
| 2000 | *IPC_EXCL* | |

# Shared Memory

Attach a memory segment:

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

- passes the start address of the shared memory segment
- *shmid* – the memory id returned by shmget
- *shmflag* may have a flag *(SHM_RDONLY)*
- the address is determined according to the following rules:

# Shared Memory

- if shmaddr = 0, the system chooses the address itself (works well in most of usage).
- if it is! = 0 then the forwarded address depends on whether the flag *SHM_RND* is set
    - if not set, the shared memory segment will be connected from the address specified by the shmaddr argument
    - if set, it will start with an address rounded down by the value of the SHMLBA (Lower Boundary Address) constant

# Shared Memory

common memory disconnection is done with

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

this function does not delete the memory segment!

# Shared Memory

To delete a shared memory segment, use

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

with an argument cmd as IPC_RMID