BITS Pilani
Pilani Campus

OS Tutorial

Interprocess communication (IPC) (shared memory)

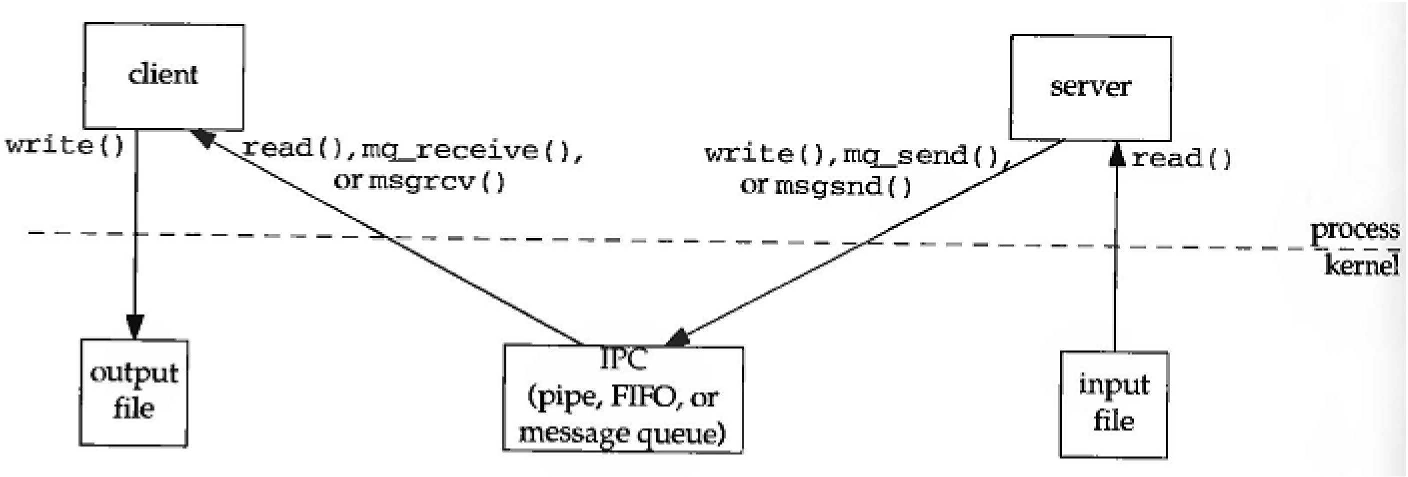
IPC methods

- Signal
- Pipe
- FIFO
- Shared Memory
- Message passing
- Semaphores

Objectives

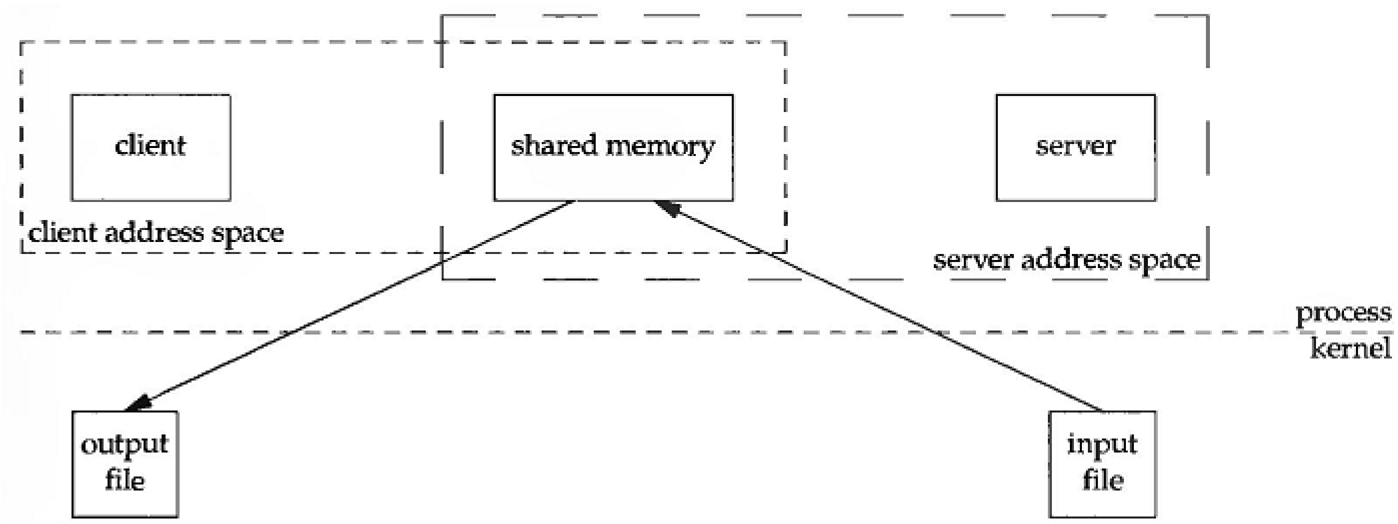
- How the processes can communicate among themselves using the Shared Memory.
- Creating a Shared Memory Segment.
- Controlling a Shared Memory Segment.
- Attaching and Detaching a Shared Memory Segment.

Message Passing



*Takes 4 **copies** to transfer data between two processes*

Shared Memory



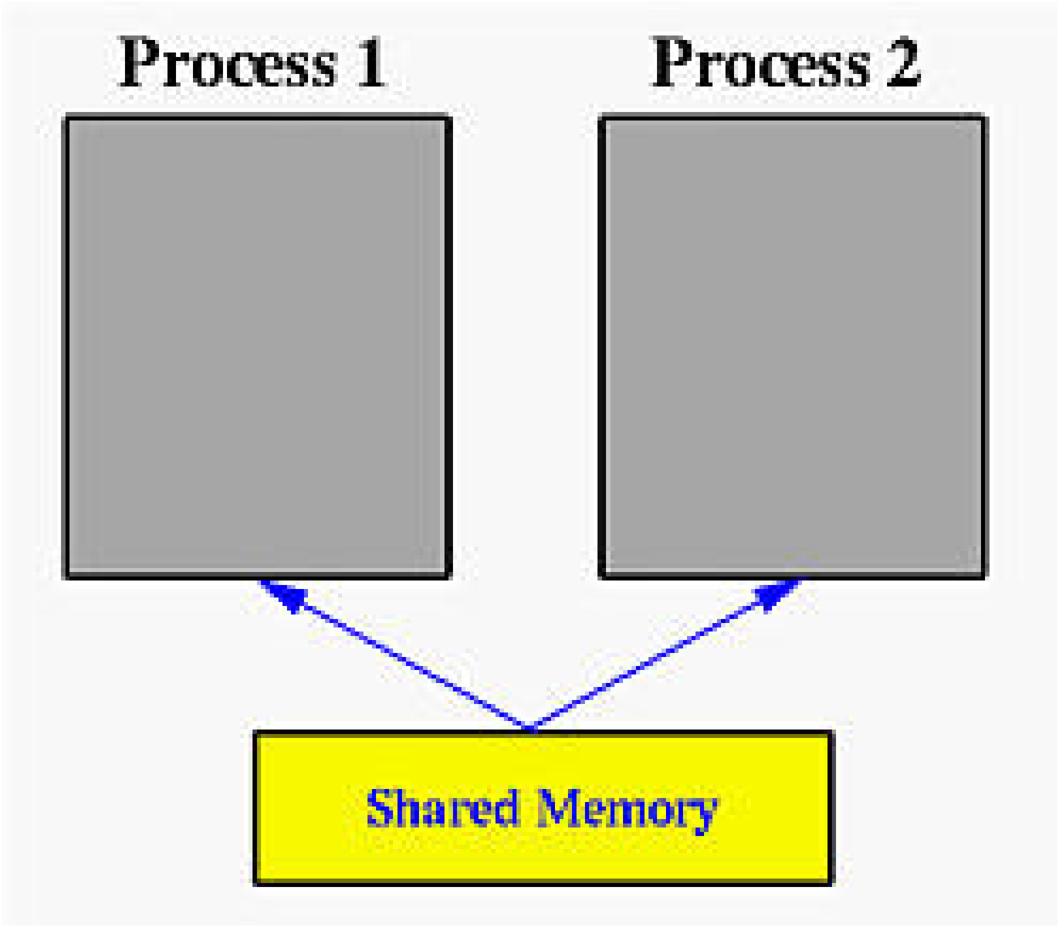
✓*Takes only two steps*

✓*Kernel is not involved in transferring data but it is involved in creating shared memory*

Shared Memory

- ✓ Shared Memory is an efficient means of passing data between programs.
- ✓ One program will create a memory portion, which other processes (if permitted) can access.
- ✓ A shared segment can be attached multiple times by the same process.
- ✓ A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory.

Shared Memory



Shared Memory

- ✓ A shared memory is an extra piece of memory that is attached to some address spaces for their owners to use.
- ✓ As a result, all the processes share the same memory segment and have access to it.

Shared Memory

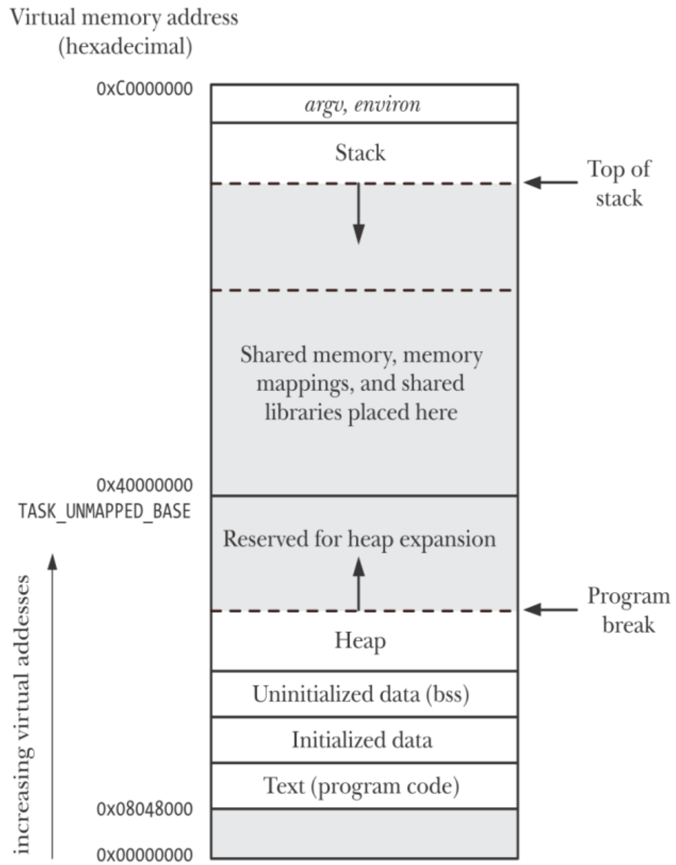
- ✓ Shared memory is a feature supported by UNIX System V, including Linux, SunOS and Solaris.
- ✓ One process must explicitly ask for an area, using a key, to be shared by other processes. This process will be called the server. All other processes, the clients that know the shared area can access it.

Shared Memory

- ✓ However, there is no protection to a shared memory and any process that knows it can access it freely.
- ✓ To protect a shared memory from being accessed at the same time by several processes, a synchronization protocol must be setup.

Location of Shared Memory in Virtual Memory

- When shared memory segment is mapped on the process address space using recommended method, it is attached as shown.



Shared Memory Limits

- SHMMNI
  - System limit on no of shared memory identifies
- SHMMIN
  - Minimum size of a shared memory segment
- SHMAX
  - Maximum size of shared memory segment.
- SHMALL
  - System limit of total number of pages of shared memory.

```
$ cd /proc/sys/kernel
$ cat shmmni
4096
$ cat shmmax
33554432
$ cat shmall
2097152
```

Table 48-2: System V shared memory limits

Limit	Ceiling value (x86-32)	Corresponding file in `/proc/sys/kernel`
SHMMNI	32768 (IPCMNI)	shmmni
SHMMAX	Depends on available memory	shmmax
SHMALL	Depends on available memory	shmall

Shared Memory

✓ The shared memory itself is described by a **structure of type** **shmid_ds** in header file **sys/shm.h**.

```
struct shmid_ds {
    struct ipc_perm shm_perm;    /* Ownership and permissions */
    size_t   shm_segsz;          /* Size of segment in bytes */
    time_t   shm_atime;          /* Time of last shmat() */
    time_t   shm_dtime;          /* Time of last shmdt() */
    time_t   shm_ctime;          /* Time of last change */
    pid_t    shm_cpid;           /* PID of creator */
    pid_t    shm_lpid;           /* PID of last shmat() / shmdt() */
    shmatt_t shm_nattch;         /* Number of currently attached processes */
};
```

Shared Memory

- ✓ The shared memory itself is described by a **structure of type shmid_ds** in header file **sys/shm.h**.
- ✓ To use this file, following files must be included:

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

Shared Memory

- For a server, it should be started before any client.
- The server should perform the following tasks:
  - Ask for a shared memory with a memory key and memorize the returned **shared memory ID**. This is performed by **system call shmget()**.
  - Attach this shared memory to the server's address space with **system call shmat()**.
  - Initialize the shared memory, if necessary.
  - Do something and wait for all clients' completion.
  - Detach the shared memory with **system call shmdt()**.
  - Remove the shared memory with system call **shmctl()**.

Shared Memory

- For the client part, the procedure is almost the same:
  - Ask for a shared memory with the same memory key and memorize the returned shared memory ID.
  - Attach this shared memory to the client's address space.
  - Use the memory.
  - Detach all shared memory segments, if necessary.
  - Exit.

shmget()

- This system call requests a shared memory segment.
- It is defined as follows:
- **shm_id** = **shmget (**

```
key_t   k,    /* the key for the segment */
int     size, /* the size of the segment */
int     flag  /* create/use flag */
);
```

- If it is successful, it returns a non-negative integer, the shared memory ID; otherwise, the function value is negative.

shmget()

- **k** is of type **key_t** or **IPC_PRIVATE**. It is the numeric key to be assigned to the returned shared memory segment.
- **size** is the size of the requested shared memory.
- **flag** is used to specify the way that the shared memory will be used.
  - Only the following two values are important:
    - **IPC_CREAT | 0666** for a server (i.e., creating and granting read and write access to the server).
    - 0666 for any client (i.e., granting read and write access to the client)

shmget()

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
.....
int   shm_id; /* shared memory ID */
.....
   shm_id = shmget (IPC_PRIVATE, 4*sizeof(int), IPC_CREAT | 0666);
   if (shm_id < 0)
   {
   printf("shmget error\n");
   exit(1);
   }
```

key

- UNIX requires a key of type **key_t** defined in file **sys/types.h** for requesting shared memory segments.
- There are three different ways of using keys, namely:
  - **A specific integer value (e.g., 123456)**
  - **A key generated with function ftok()**
  - A uniquely generated key using IPC_PRIVATE (i.e., a private key).

key

- The first way is the easiest one; however, its use may be very risky since a process can access your resource as long as it uses the same key value to request that resource.
- The following example assigns 1234 to a key:

  key_t SomeKey;

  SomeKey = 1234;

- **The ftok() function has the following prototype:**

```
key_t   ftok (
                const char *path,    /* a path string */
                int   id             /* an integer value */
              );
```

key

- ftok() takes a character string that identifies a path and an integer (usually a character) value, and generates an integer of type key_t based on the first argument with the value of id in the most significant position.

- Example: if the generated integer is $35028A5D_{16}$ and the value of id is 'a' (ASCII value = 61_{16}), then ftok() returns $61028A5D_{16}$. That is, 61_{16} replaces the first byte of $35028A5D_{16}$,generating 61028A5D16.

key

- ✓ If a processes use the same arguments to call ftok(), the returned key value will always be the same.
- ✓ The most commonly used value for the first argument is ".", the current directory.
- ✓ If all related processes are stored in the same directory, the following call to **ftok() will generate the same key value.**

```
#include <sys/types.h>
#include <sys/ipc.h>

key_t   SomeKey;
SomeKey = ftok(".", 'x');
```

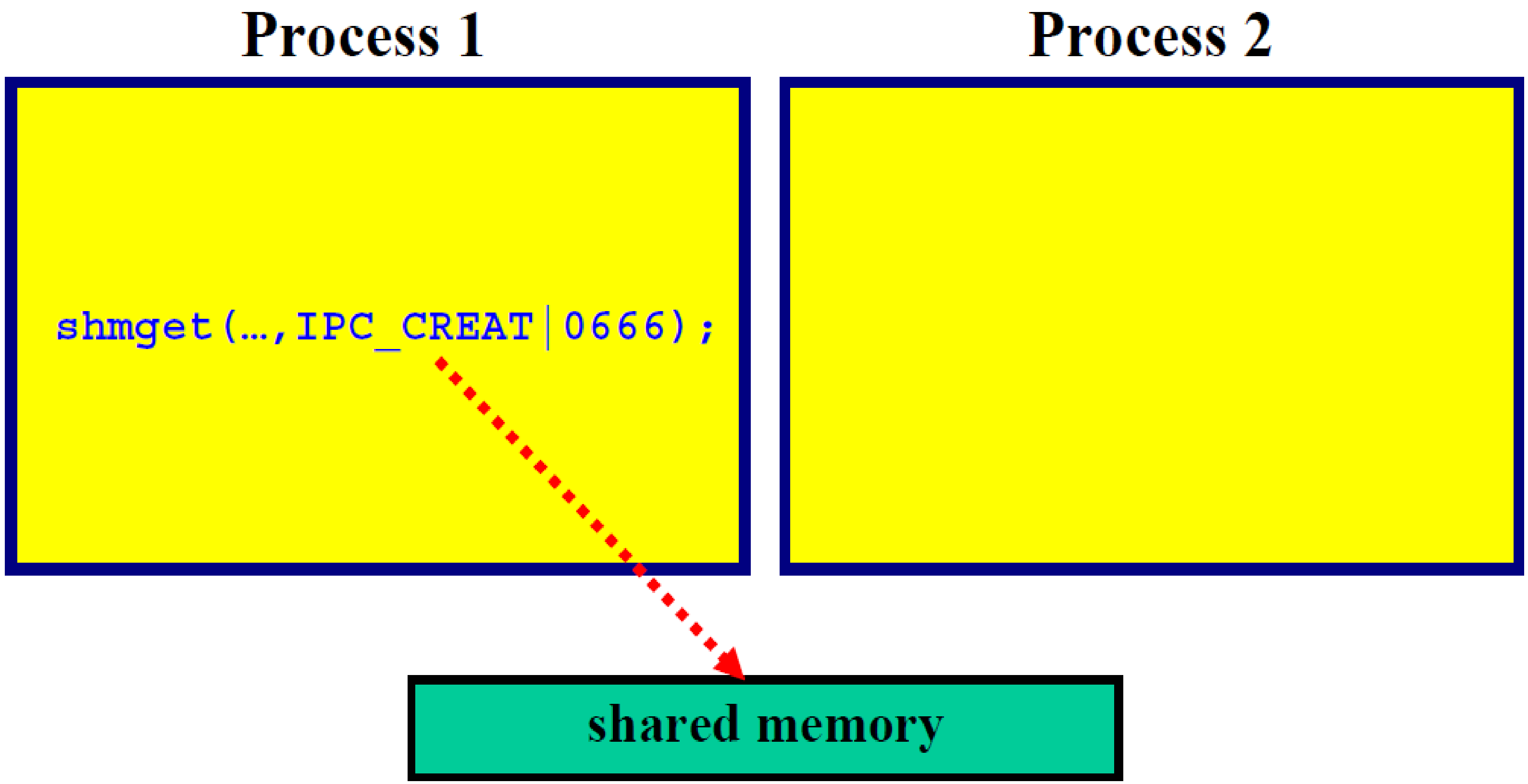
key

- After obtaining a key, it can be used in any place where a key is required.
- Moreover, the place where a key is required accepts a special parameter, IPC_PRIVATE.
  - the system will generate a unique key and guarantee that no other process will have the same key.

key

- If a resource is requested with IPC_PRIVATE in a place where a key is required, that process will receive a unique key for that resource.
  - Since that resource is identified with a unique key unknown to the outsiders, other processes will not be able to share that resource and, as a result, the requesting process is guaranteed that it owns and accesses that resource exclusively.

After the Execution of shmget()



Shared memory is allocated; but, is not part of the address space

key

- Once a shared memory segment has been created, a process attaches it to its address space by calling shmat.

```c
#include <sys/types.h>          /* For portability */
#include <sys/shm.h>
void *shmat(int  shmid , const void * shmaddr , int  shmflg );
//Returns address at which shared memory is attached on success,
//or (void *) -1 on error
```

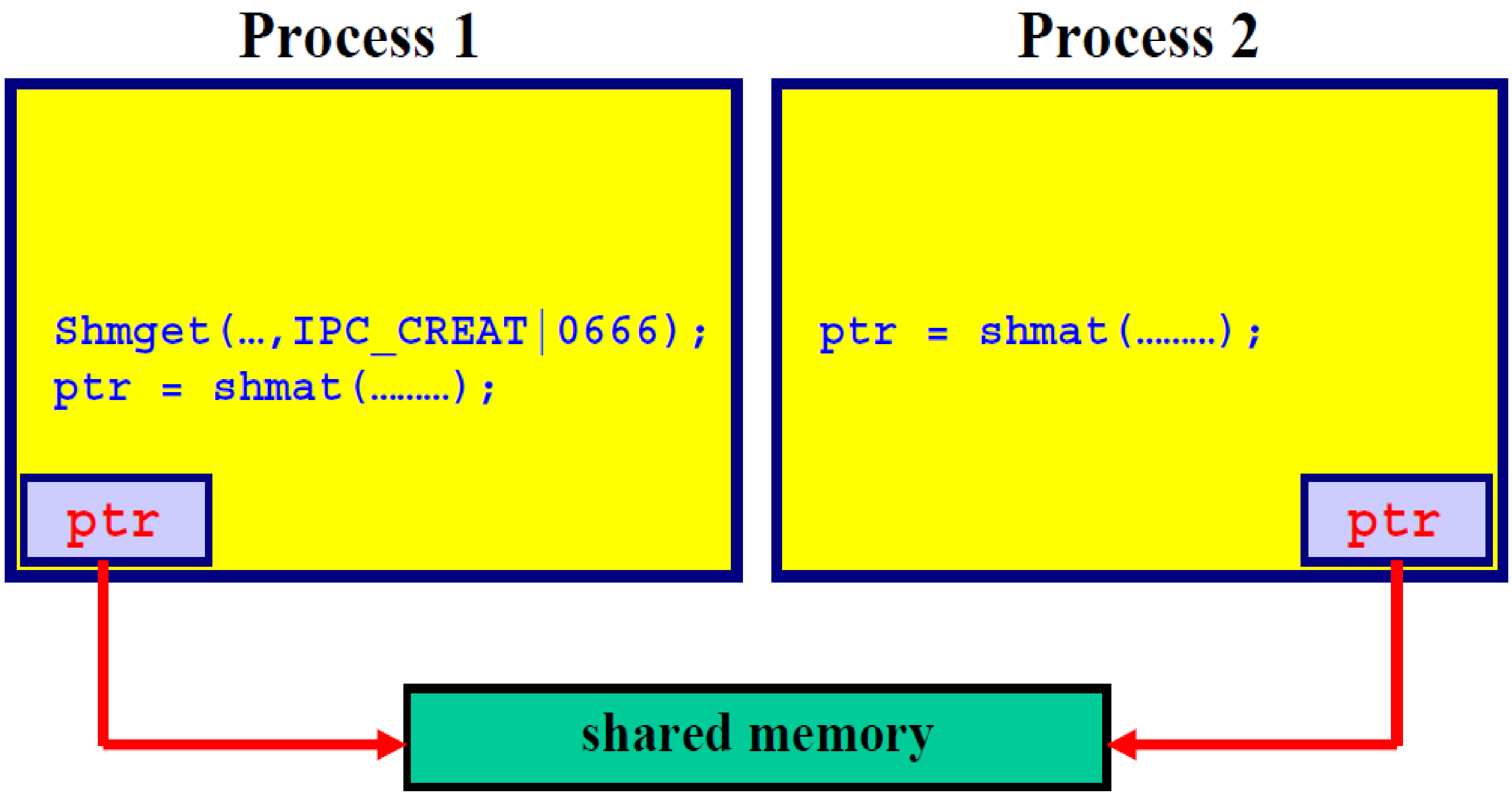
The address in the calling process at which the segment is attached depends on the addr argument.

- If addr is 0, the segment is attached at the first available address selected by the kernel.
- This is the recommended technique.

Table 48-1: *shmflg* bit-mask values for *shmat()*

Value	Description
SHM_RDONLY	Attach segment read-only
SHM_REMAP	Replace any existing mapping at *shmaddr*
SHM_RND	Round *shmaddr* down to multiple of SHMLBA bytes

After the Execution of shmat()



Now processes can access the shared memory

Server Program

```
#include <sys/types.h>
 #include <sys/ipc.h>
 #include <sys/shm.h>
 #include <stdio.h>
int shm_id;
key_t mem_key;
 int *shm_ptr;
mem_key = ftok(".", 'a');
shm_id = shmget(mem_key,
    4*sizeof(int), IPC_CREAT | 0666);
if (shm_id < 0) {
printf("*** shmget error (server) ***\n");
exit(1);
}
shm_ptr = (int *) shmat(shm_id, NULL, 0);
    /* attach */
if ((int) shm_ptr == -1) {
printf("*** shmat error (server) ***\n");
exit(1);
}
```

innovate achieve lead

Client Program

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
int shm_id;
key_t mem_key;
int *shm_ptr;
mem_key = ftok(".", 'a');
shm_id = shmget(mem_key,
    4*sizeof(int), 0666);
if (shm_id < 0) {
printf("*** shmget error (server) ***\n");
exit(1);
}
shm_ptr = (int *) shmat(shm_id, NULL, 0);
    /* attach */
if ((int) shm_ptr == -1) {
printf("*** shmat error (server) ***\n");
exit(1);
}
```

shmdt() and shmctl()

- **shmdt() is used to detach a shared memory.**
- After a shared memory is detached, it cannot be used. However, it is still there and can be re-attached back to a process's address space.
- **To remove a shared memory, use shmctl().**
- The only argument to shmdt() is **the shared memory address returned by shmat()**.

shmdt() and shmctl()

- **shmdt (shm_ptr), it will detached the shared memory.**
- **shm_ptr** is the pointer to the shared memory, returned by **shmat()** during shared memory attachment.
- **If shmdt() fails, the returned value is non-zero.**
- To remove a shared memory segment, use the following **shmctl (shm_id, IPC_RMID, NULL)**.
- **shm_id is the shared memory ID**.
- **IPC_RMID indicates this is a remove operation**.

- Two different processes communicating via shared memory

Server Process

```
main()
{
char c;
int shmid;
key_t key;
char *shm, *s;
key = 5678;
/* * create the segment.* */
if ((shmid = shmget(key, SHMSIZE, IPC_CREAT | 0666)) < 0)
{
perror("shmget");      exit(1);
}
/** Now we attach the segment to our data space.*/
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
{
perror("shmat");
exit(1);
}
```

```
/** Now put some things into the memory for the other process to read. */
s = shm;
for (c = 'a'; c <= 'z'; c++)
*s++ = c;
*s = NULL;
/** Finally, we wait until the other process
* Changes the first character of our memory
* to '*', indicating that it has read what
* we put there.
*/
while (*shm != '*')
sleep(1);
exit(0);
}
```

Server Process

Client Process

```
main()
{
int shmid;
key_t key;
char *shm, *s;
/*
* We need to get the segment named
* "5678", created by the server.
*/
key = 5678;
/*
* Locate the segment.
*/
if ((shmid = shmget(key, SHMSIZE, 0666)) < 0) {
perror("shmget");
exit(1);
}
```

Client Process

```
/* Now we attach the segment to our data space*/
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1)
 {
perror("shmat");
exit(1);
}
/* Now read what the server put in the memory*/
for (s = shm; *s != NULL; s++)
putchar(*s);
putchar('\n');
/* Finally, change the first character of the
* segment to '*', indicating we have read
* the segment.
*/
*shm = '*';
printf ("\nIts done from client.\n\n\n");
exit(0);
}
```

- Parent and Child processes communicating via shared memory

```c
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int main(void)
{
int shmid;
char *shmPtr;
int n;
if (fork( ) == 0)
 {
sleep(5);
if( (shmid = shmget(2041, 32, 0)) == -1 )
{
exit(1);
}
```

```
shmPtr = shmat(shmid, 0, 0);
if (shmPtr == (char *) -1)
exit(2);
printf ("\nChild Reading ....\n\n");
for (n = 0; n < 26; n++)
putchar(shmPtr[n]);
putchar('\n');
}
else
{
if( (shmid = shmget(2041, 32, 0666 | IPC_CREAT)) == -1 )
 {
exit(1);
}
shmPtr = shmat(shmid, 0, 0);
```

```
if (shmPtr == (char *) -1)
exit(2);
for (n = 0; n < 26; n++)
shmPtr[n] = 'a' + n;
printf ("Parent Writing ....\n\n") ;
for (n = 0; n < 26; n++)
 putchar(shmPtr[n]);
putchar('\n');
wait(NULL);
shmdt(NULL);
 if( shmctl(shmid, IPC_RMID, NULL) == -1 )
{
perror("shmctl");
exit(-1);
}
}
exit(0);
}
```

```
if (shmPtr == (char *) -1)
exit(2);
for (n = 0; n < 26; n++)
shmPtr[n] = 'a' + n;
printf ("Parent Writing ....\n\n") ;
for (n = 0; n < 26; n++)
 putchar(shmPtr[n]);
putchar('\n');
wait(NULL);
shmdt(NULL);
 if( shmctl(shmid, IPC_RMID, NULL) == -1 )
{
perror("shmctl");
exit(-1);
}
}
exit(0);
}
```