POLITECNICO
DI TORINO

Inter Process Communication

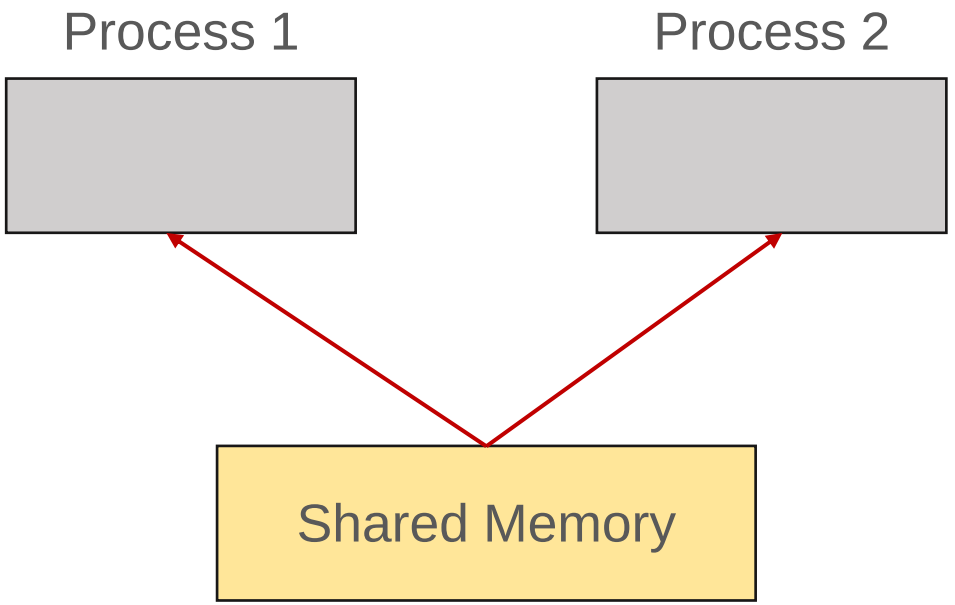
Operating Systems – Sarah Azimi

OUTLINE

- IPC mechanisms:
  - Shared Memory
  - Message Passing
  - Pipes (named and unnamed)
  - Signals

Shared Memory

- Shared memory is a memory shared between two or more processes.



Shared Memory

- Shared memory is a memory shared between two or more processes.

- Create the shared memory segment or use already shared memory segment
  - `shmget()`
- Attached the process to already created shared memory segment
  - `shmat()`
- Detach the process from the already attached shared memory segment
  - `shmdt()`
- Control operations on the shared memory segment
  - `shmctl()`

- `#include <sys/stat.h>, #include <sys/ipc.h>, #include <sys/types.h>`

Creating a Shared Memory Segment

- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id = shmget (key_t key, int size , int flag);
```

Creating a Shared Memory Segment

- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id = shmget (key_t key, int size , int flag);
```

- Key is of type key_t. It is an integer that specifies which segment to create:
  - `IPC_PRIVATE` For creating a new private shared memory segment that is accessible only by its creating process or child processes of that creator.
  - Unrelated processes can access the same public shared segment by specifying the same key value.

Creating a Shared Memory Segment

- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id =  shmget  (key_t  key, int  size  , int   flag);
```

- The argument Size determines the size in byte of the shared memory segment.

Creating a Shared Memory Segment

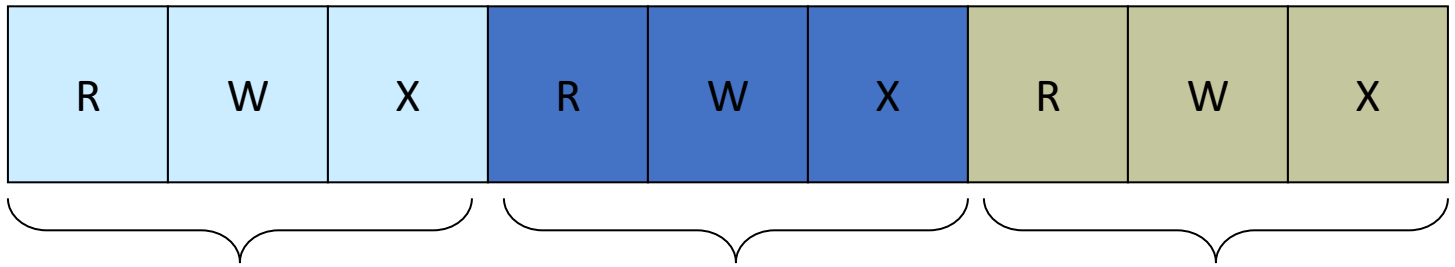
- The system call which requests a shared memory segment is shmget()
  - Syntax:

```
Shm_id = shmget (key_t key, int size , int flag);
```

- The argument flag is used to indicate segment creation conditions and access permissions.
  - IPC_CREATE : A new segment should be created.
  - IPC_EXCL : It causes `shmget` to fail if the specified segment key already exists.
  - Mode flags: This value is made of 9 bits indicating permissions granted to owner, group and world to control access to the segment. Execution bits are ignored.

Mode Flags: Access Control

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



S_IRUSR	S_IRGRP	S_IROTH
S_IWUSR	S_IWGRP	S_IWOTH
S_IXUSR	S_IXGRP	S_IXOTH
S_IRWXU	S_IRWXG	S_IRWXO

Generating a unique Key

- Type `key_t` is an integer number. You can use any number you want.
- A special function, `ftok()` has been introduced to generate a unique key, from two arguments:
  - `key_t ftok(const char *path, int id);`
  - `path` is a file that this process can read
  - `id` is usually just set to some arbitrary char, like 'A'.
- The ftok() function uses information about the named file (in particular its inode number) and the id to generate a probably-unique key for the shared memory segment.

Example:

```
key_t key;
int shmid;
key = ftok ("<somefile>", 'A');
shmid = shmget (key, 1024, 0644 | IPC_CREAT);
```

Attach a shared memory segment

- `shmat` is used to attach the referenced shared memory segment into the calling process's data segment.
  - Syntax:

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

  - `shmid` is a valid shared memory identifier.
  - `shmaddr` allows the calling process some flexibility in assigning the location of the shared memory segment
    - If a nonzero value is given, `shmat` uses this as the attachment address for the shared memory segment
    - If `shmaddr` is 0, the system will choose an available address.
  - `shmflg` is used to specify the access permissions for the shared memory segment and to request special attachment conditions, such as a read-only segment

Attach a shared memory segment

- `shmat` is used to attach the referenced shared memory segment into the calling process's data segment.
  - Syntax:

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

  - Return value:
    - If `shmat` is successful, it returns the address of the actual attachment
    - If `shmat` fails, it returns -1.

Detaching a shared memory segment

- When a process finished with a shared memory segment, the segment should be detached using `shmdt()`.
- In order to detach a segment, it is needed to pass the address returned by `shmat()`.

- Syntax:

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

  - `void *shmaddr`: a reference to an attached memory segment (the shared memory pointer).

- Return value:
  - Success: 0
  - Failure: -1

Controlling shared memory segment

- `shmctl()` call returns information about a shared memory segment and can modify it.

- Syntax:

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

- `shmid` corresponds to the shared memory identifier (i.e., the address returned by `shmat()`)

- To obtain information: pass `IPC_STAT` (see man pages for the details)
- To remove a segment: pass `IPC_RMID` as the second parameter and NULL as the third parameter. The segment is removed when the last process that has attached it, finally detaches it.

Example

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

#define SHMSZ     27
main()
{
    char c;
    int shmid;
    char *shm, *s;
    pid_t pid;

        if ((shmid = shmget(IPC_PRIVATE, SHMSZ, IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR)) < 0) {
                printf("Error in shmget");
                exit(1);
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                 perror("Error in shmat");
                 exit(1);
    }
   pid= fork();
```

Example (cont.)

```
if (pid < 0) exit(-1);
else if (pid == 0) { /* child process */
                    while (shm[SHMSZ-2] != 'z')
                            sleep(1);
                  printf("Reading: %s\n", shm);
                    for (s = shm; *s != NULL ; s++)
                            putchar(*s);
                    putchar('\n');

                    exit (0);
                    }
else { /* parent process */
              s = shm;
              for (c = 'a' ; c <= 'z' ; c++)
                      *s++ = c;
              *s = NULL;
        printf("Writing: %s\n", shm);
              wait(NULL);
              shmdt(shm);
              shmctl(shmid, IPC_RMID, 0);
              exit(0);
         }
}
```

Client – Server version

SERVER

```
#define SHMSZ 1
main(int argc, char **argv)
{
        char c, tmp;
        int shmid;
        key_t key;
        char *shm;
        key = 1234; /* Shared memory segment at 1234 */
        if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
                perror("shmget");
                return 1;
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                return 1;
        }
        *shm = 0;
         tmp = *shm;
        while (*shm != 'q'){
                sleep(1);
                if(tmp == *shm) continue;
                fprintf(stdout, "You pressed %c\n",*shm);
                tmp = *shm;
        }
        if(shmdt(shm) != 0)  fprintf(stderr, "Could not close memory segment.\n");
        shmctl(shmid, IPC_RMID, 0);
        return 0;
}
```

Client – Server version

CLIENT

```
#define SHMSZ     1
main()
{       int shmid;
        key_t key;
        char *shm;
        key = 1234; /* We need to get the segment named "1234", created by the server. */
        if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
                perror("shmget");
                return 1;
        }
        if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
                perror("shmat");
                return 1;
        }
        for(;;){
                char tmp = getchar();
                getchar(); // Eat the enter key
                *shm = tmp;
                if(tmp == 'q')
                    break;
        }
        if(shmdt(shm) != 0) fprintf(stderr, "Could not close memory segment.\n");
        return 0;
}
```

Shared Memory in POSIX

- Shared memory is a memory shared between two or more processes.

- Create or open the existing shared memory object
  - `Shm_open()`
- Map the shared memory region in the address space of a process
  - `mmap()`
- Delete the map
  - `munmap()`
- Remove the shared memory object
  - `Shm_unlinck()`

- `#include <sys/stat.h>, #include <fcntl.h>, #include <sys/types.h>`

Shared Memory in POSIX

- `shm_open()` creates and opens a new, or opens an existing, POSIX shared memory object. It returns a file descriptor to the shared memory.
- A POSIX shared memory object is a handle which can be used by unrelated processes to the same region of shared memory.

  - Syntax:

    int shm_open (const char * name, int oflag, mode_t mode)

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

  - Note that when you compile your code, real time library should be used

    gcc -o output filename.c –lrt

Shared Memory in POSIX

- `shm_open()` creates and opens a new, or opens an existing, POSIX shared memory object. It returns a file descriptor to the shared memory.
- A POSIX shared memory object is a handle which can be used by unrelated processes to the same region of shared memory

  - Syntax:

    int fd = int shm_open (const char * name, int oflag, mode_t mode)

  - `Name` is of the shared memory object (`/name`)
  - `Oflag` is a bit mask : `O_RDONLY |O_RDWR | O_CREATE | …`
  - `Mode` established the permissions of the shared memory object (0666)
  - If successful, returns the file descriptor for the shared memory object (`fd`). If it fails, it returns -1.

Shared Memory in POSIX

- To set the size of shared memory object
  - Syntax:

    ```
    int ftruncate (int fd, off_t length);
    ```

  - `ftruncate` function causes the regular file referenced by `fd` to be truncated to a size of precisely length bytes. If the file previously was larger than this size, the extra data is lost. If the file previously was shorter, it is extended, and the extended part reads as zero bytes. The file pointer is not changed.

Shared Memory in POSIX

- Creating a new mapping in the virtual address space of the calling process.
  - Syntax:
    void *mmap (void *addr, size_t length, int prot, int flags, int fd, off_t offset);
  - `Addr` is the address at which the shared memory should be mapped (NULL).
  - `Length` is the length of the shared memory object that should be mapped.
  - `Prot` can have the values, `PROT_EXEC`, `PROT_READ`, `PROT_WRITE` and `PROT_NONE`.
  - `Flags` There are several flags, but `MAP SHARED` is essential for shared memory
  - `Fd` is the file descriptor for the shared memory received from `shm_open` call.
  - `Offset` is the point where the mapping begins in the shared memory entity (the 0 offset value can also be used)
- On success, `mmap` returns the pointer to the location where the shared memory object has been mapped. In case of error, -1 is returned

Shared Memory in POSIX

- Deleting the mappings for the specified address range and causes further references to addresses within the range to generate invalid memory references. The region is also automatically unmapped when the process is terminated. On the other hand, closing the file descriptor does not unmap the region.

  - Syntax:

    int munmap (void *addr, size_t length);

  - `addr` is the address at which the shared memory should be mapped.
  - `length` is the length of the shared memory object that should be mapped.

- On success, `munmap` returns 0. In case of error, -1 is returned

Shared Memory in POSIX

- Removing the shared memory object
  - Syntax:
    int shm_unlinck (const char * name)
  - `name` is the name of the shared memory object
  - If successful, returns 0. If it fails, it returns -1.
  - Note that when you compile your code, real time library should be used

    gcc -o output filename.c –lrt

Example

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
#define DEATH(mess) { perror(mess); exit(errno); }
#define SIZE 8196
#define NAME "/my_shm"

int main (int argc, char *argv[])
{
    if (argc > 1) {
        if (!strcasecmp ("create", argv[1]))
            create_it ();
        if (!strcasecmp ("remove", argv[1]))
            remove_it ();
        if (!strcasecmp ("send", argv[1]))
            send_it ();
        if (!strcasecmp ("receive", argv[1]))
            receive_it ();
}
    printf ("Usage: %s  create | remove | receive | send \n", argv[0]);
    exit (-1);
}
```

Example (cont.)

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
void create_it (void)
{
    int shm_fd;

    if ((shm_fd = shm_open (NAME, O_RDWR | O_CREAT | O_EXCL, 0666)) == -1)
        DEATH ("shm_open");
    ftruncate (shm_fd, SIZE);

    printf ("Shared Memory Region successfully created\n");
    exit (EXIT_SUCCESS);
}

void remove_it (void)
{
    int shm_fd;
    if ((shm_fd = shm_open (NAME, O_RDWR, 0) == -1))
        DEATH ("shm_open");
    if (shm_unlink (NAME))
        DEATH ("shm_unlink");
    printf ("Shared Memory Region successfully destroyed\n");

    exit (EXIT_SUCCESS);
}
```

Example (cont.)

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
void send_it (void)
{
    int shm_fd, iflag = 1;
    void *shm_area;

    if ((shm_fd = shm_open (NAME, O_RDWR, 0)) == -1)
        DEATH ("shm_open");

    shm_area = mmap (NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);

    if (shm_area == MAP_FAILED)
        DEATH ("mmap");

    printf ("SEND: Memory attached at %lX\n", (unsigned long) shm_area);
    memcpy (shm_area, &iflag, sizeof(int));

    if (munmap (shm_area, SIZE))
        DEATH ("munmap");

    printf ("SEND has successfully completed\n");
    exit (EXIT_SUCCESS);
}
```

Example (cont.)

- Write a C program to pass messages to each other in shared memory. Receiving program terminates when it receives the message.

```
void receive_it (void)
{
    int shm_fd, iflag = 8;
    void *shm_area;
    if ((shm_fd = shm_open (NAME, O_RDWR, 0)) == -1)
        DEATH ("shm_open");
    shm_area = mmap (NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, shm_fd, 0);
    if (shm_area == MAP_FAILED)
        DEATH ("mmap");
    printf ("RCV: Memory attached at %lX\n", (unsigned long)shm_area);
    memcpy (shm_area, &iflag, sizeof(int));
    printf ("iflag is now = %d\n", iflag);
    while (iflag == 8) {
        memcpy (&iflag, shm_area, sizeof(int));
        sleep (1);
    }
    printf ("RCV has successfully completed\n");
    printf ("iflag is now = %d\n", iflag);
    if (munmap (shm_area, SIZE))
        DEATH ("munmap");
    exit (EXIT_SUCCESS);
}
```