Unix Shared Memory
What is Shared Memory?

- The parent and child processes are run in **separate** address spaces.
- A *shared memory segment* is a piece of memory that can be allocated and attached to an address space. Thus, processes that have this memory segment attached will have access to it.
- But, *race conditions can occur!"
Procedure for Using Shared Memory

- Find a *key*. Unix uses this key for identifying shared memory segments.
- Use `shmget()` to allocate a shared memory.
- Use `shmat()` to attach a shared memory to an address space.
- Use `shmdt()` to detach a shared memory from an address space.
- Use `shmctl()` to deallocate a shared memory.
To use shared memory, include the following:

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

A key is a value of type `key_t`. There are three ways to generate a key:

- Do it yourself
- Use function `ftok()`
- Ask the system to provide a private key.
Keys: 2/2

- **Do it yourself:** use
  
  ```c
  key_t    SomeKey;
  SomeKey = 1234;
  ```

- **Use `ftok()` to generate one for you:**
  
  ```c
  key_t = ftok(char *path, int ID);
  ```
  - *path* is a path name (e.g., `./`)
  - *ID* is an integer (e.g., `'a'`)
  - Function `ftok()` returns a key of type `key_t`:
    ```c
    SomeKey = ftok("./", 'x');
    ```

- Keys are **global** entities. If other processes know your key, they can access your shared memory.

- Ask the system to provide a private key using `IPC_PRIVATE`. 
Asking for a Shared Memory: 1/4

- Include the following:

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

- Use `shmget()` to request a shared memory:

```c
shm_id = shmget(
    key_t  key, /* identity key */
    int    size, /* memory size */
    int    flag); /* creation or use */
```

- `shmget()` returns a shared memory ID.

- The flag, for our purpose, is either 0666 (rw) or `IPC_CREAT | 0666`. Yes, `IPC_CREAT`.
The following creates a shared memory of size `struct Data` with a private key `IPC_PRIVATE`. This is a creation (`IPC_CREAT`) and permits read and write (0666).

```c
struct Data { int a; double b; char x; }; int ShmID;

ShmID = shmget(
    IPC_PRIVATE, /* private key */
    sizeof(struct Data), /* size */
    IPC_CREAT | 0666); /* cr & rw */
```
The following creates a shared memory with a key based on the current directory:

```c
struct Data { int a; double b; char x;};
int ShmID;
key_t Key;

Key = ftok("./", 'h');
ShmID = shmget(
    Key, /* a key */
    sizeof(struct Data),
    IPC_CREAT | 0666);
```
When asking for a shared memory, the process that creates it uses `IPC_CREAT | 0666` and the process that accesses a created one uses `0666`.

If the return value is negative (Unix convention), the request was unsuccessful, and no shared memory is allocated.

Create a shared memory before its use!
After the Execution of `shmget()`

Process 1

```
shmget(..., IPC_CREAT | 0666);
```

Process 2

- Shared memory is allocated; but, is not part of the address space
Attaching a Shared Memory: 1/3

- Use `shmat()` to attach an existing shared memory to an address space:

```c
shm_ptr = shmat(
    int  shm_id, /* ID from shmget() */
    char *ptr,  /* use NULL here    */
    int  flag); /* use 0 here       */
```

- `shm_id` is the shared memory ID returned by `shmget()`.
- Use `NULL` and `0` for the second and third arguments, respectively.
- `shmat()` returns a `void` pointer to the memory. If unsuccessful, it returns a negative integer.
struct Data { int a; double b; char x;};
int    ShmID;
key_t  Key;
struct Data *p;

Key = ftok("./", ‘h’);
**ShmID = shmget(Key, sizeof(struct Data), IPC_CREAT | 0666);**
p = (struct Data *) shmat(ShmID, NULL, 0);
if ((int) p < 0) {
    printf("shmat() failed\n"); exit(1);
}
p->a = 1; p->b = 5.0; p->c = ‘.’;
Attaching a Shared Memory: 3/3

Now processes can access the shared memory
Detaching/Removing Shared Memory

- To detach a shared memory, use
  
  ```c
  shmdt(shm_ptr);
  ```
  
  `shm_ptr` is the pointer returned by `shmat()`.

- After a shared memory is detached, it is still there. You can re-attach and use it again.

- To remove a shared memory, use
  
  ```c
  shmctl(shm_ID, IPC_RMID, NULL);
  ```
  
  `shm_ID` is the shared memory ID returned by `shmget()`. After a shared memory is removed, it no longer exists.
Communicating with a Child: 1/2

```c
void main(int argc, char *argv[]) {
    int ShmID, *ShmPTR, status;
    pid_t pid;

    ShmID = shmget(IPC_PRIVATE, 4*sizeof(int), IPC_CREAT|0666);
    ShmPTR = (int *) shmat(ShmID, NULL, 0);
    ShmPTR[0] = atoi(argv[0]);  ShmPTR[1] = atoi(argv[1]);
    if ((pid = fork()) == 0) {
        Child(ShmPTR);
        exit(0);
    }
    wait(&status);
    shmdt((void *) ShmPTR);  shmctl(ShmID, IPC_RMID, NULL);
    exit(0);
}
```
Communicating with a Child: 2/2

```c
void Child(int SharedMem[]) {
    printf("%d %d %d %d\n", SharedMem[0],
         SharedMem[1], SharedMem[2], SharedMem[3]);
}
```

Why are `shmget()` and `shmat()` unnecessary in the child process?
Communicating Among Separate Processes: 1/5

- Define the structure of a shared memory segment as follows:

```c
#define NOT_READY (-1)
#define FILLED (0)
#define TAKEN (1)

struct Memory {
    int status;
    int data[4];
};
```
The “Server”

```c
void main(int argc, char *argv[])
{
    key_t            ShmKEY;
    int             ShmID, i;
    struct Memory   *ShmPTR;
    ShmKEY = ftok("./", 'x');
    ShmID = shmget(ShmKEY, sizeof(struct Memory),
                    IPC_CREAT | 0666);
    ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
}
```

Prepare for a shared memory
Communicating Among Separate Processes: 3/5

```c
ShmPTR->status = NOT_READY;
for (i = 0; i < 4; i++)
    ShmPTR->data[i] = atoi(argv[i]);
ShmPTR->status = FILLED;
while (ShmPTR->status != TAKEN)
    sleep(1);  /* sleep for 1 second */
shmdt((void *) ShmPTR);
shmctl(ShmID, IPC_RMID, NULL);
exit(0);
}
```

- **shared memory not ready**
- **filling in data**
- **wait until the data is taken**
- **detach and remove shared memory**
void main(void) {
    key_t ShmKEY;
    int ShmID;
    struct Memory *ShmPTR;

    ShmKEY=ftok("./", ‘x’);
    ShmID = shmget(ShmKEY, sizeof(struct Memory), 0666);
    ShmPTR = (struct Memory *) shmat(ShmID, NULL, 0);
    while (ShmPTR->status != FILLED)
        printf("%d %d %d %d\n", ShmPTR->data[0],
               ShmPTR->data[1], ShmPTR->data[2], ShmPTR->data[3]);
    ShmPTR->status = TAKEN;
    shmdt((void *) ShmPTR);
    exit(0);
}
Communicating Among Separate Processes: \(5/5\)

- The “server” must run first to prepare a shared memory.
- Try run the server in one window, and run the client in another a little later.
- Or, run the server as a background process. Then, run the client in the foreground:
  ```
  server 1 3 5 7 &
  client
  ```
- This version uses busy waiting.
- One may use Unix semaphores for mutual exclusion.
Important Notes

- If you did not remove your shared memory segments (e.g., program crashes before the execution of `shmctl()`), they will be in the system forever. This will degrade the system performance.
- Use the `ipcs` command to check if you have shared memory segments left in the system.
- Use the `ipcrm` command to remove your shared memory segments.