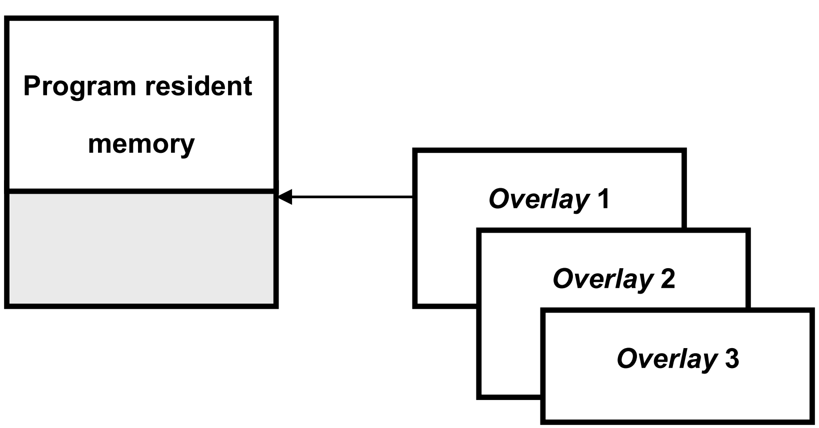
Memory management

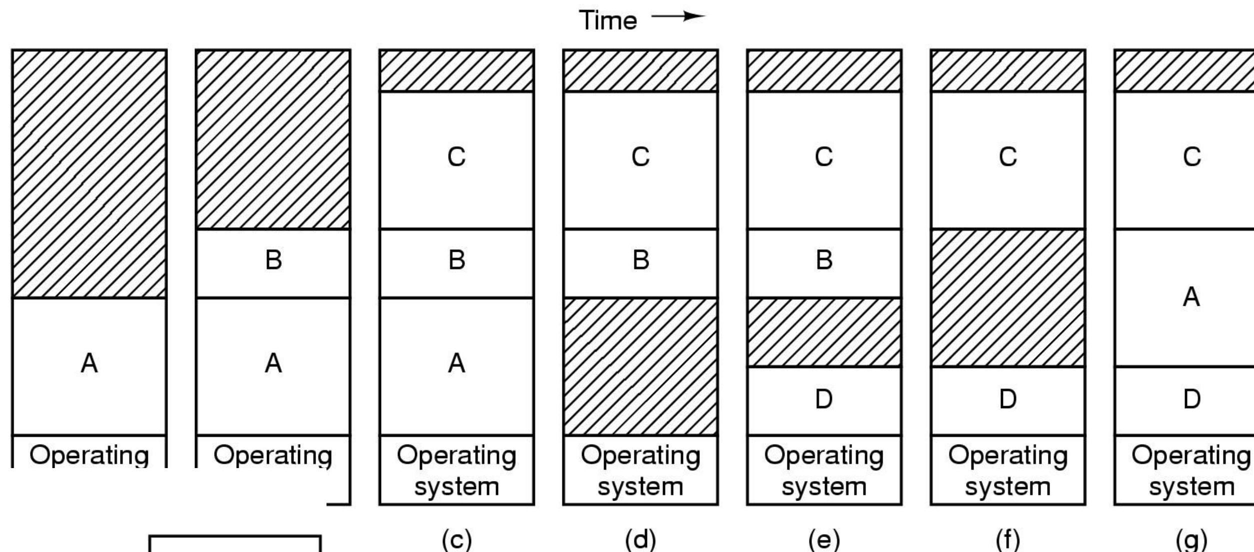
- Single process
  - All memory assigned to the process
  - Addresses defined at compile time
- Multiple processes. How to:
  - assign memory
  - manage addresses?
  - manage relocation?
  - manage program grow?

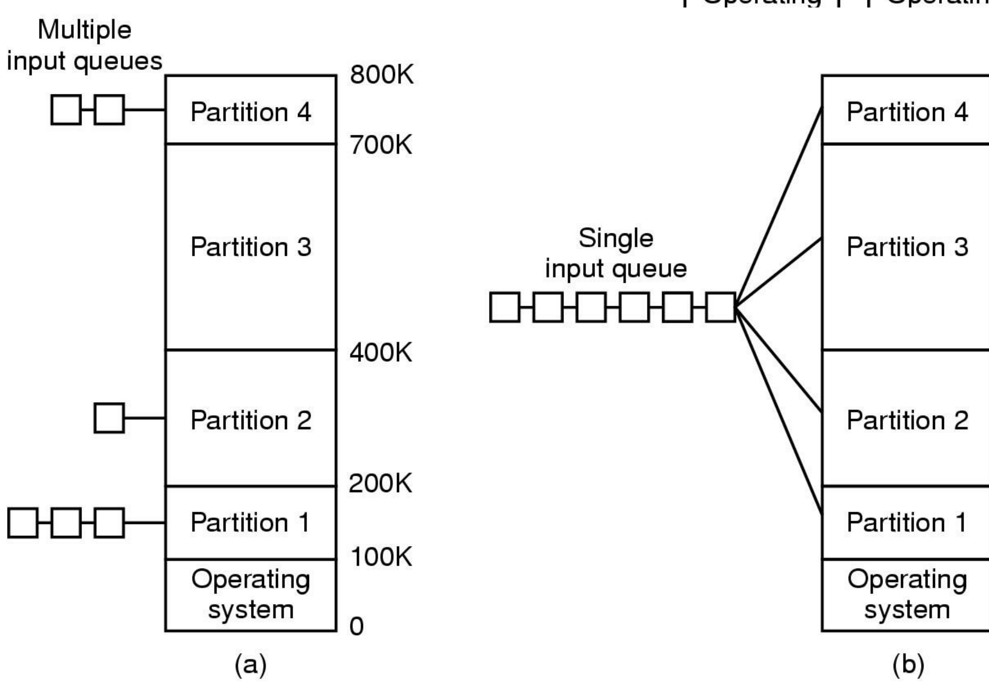
manage program grow?

- overlay
  - Statically defined memory zone
  - loaded on demand
    - By the program
  - may be replaced



Memory management



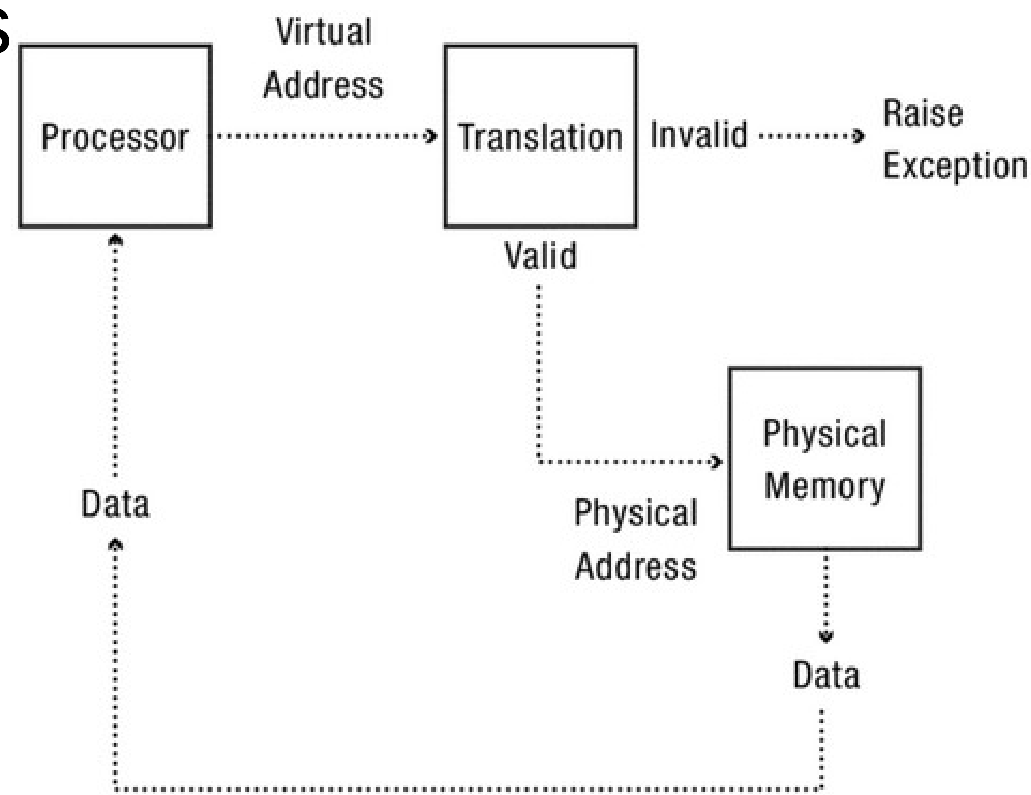


```c
for(i = 0; i< 1000; i++){
    vect[i] = c;
}
```

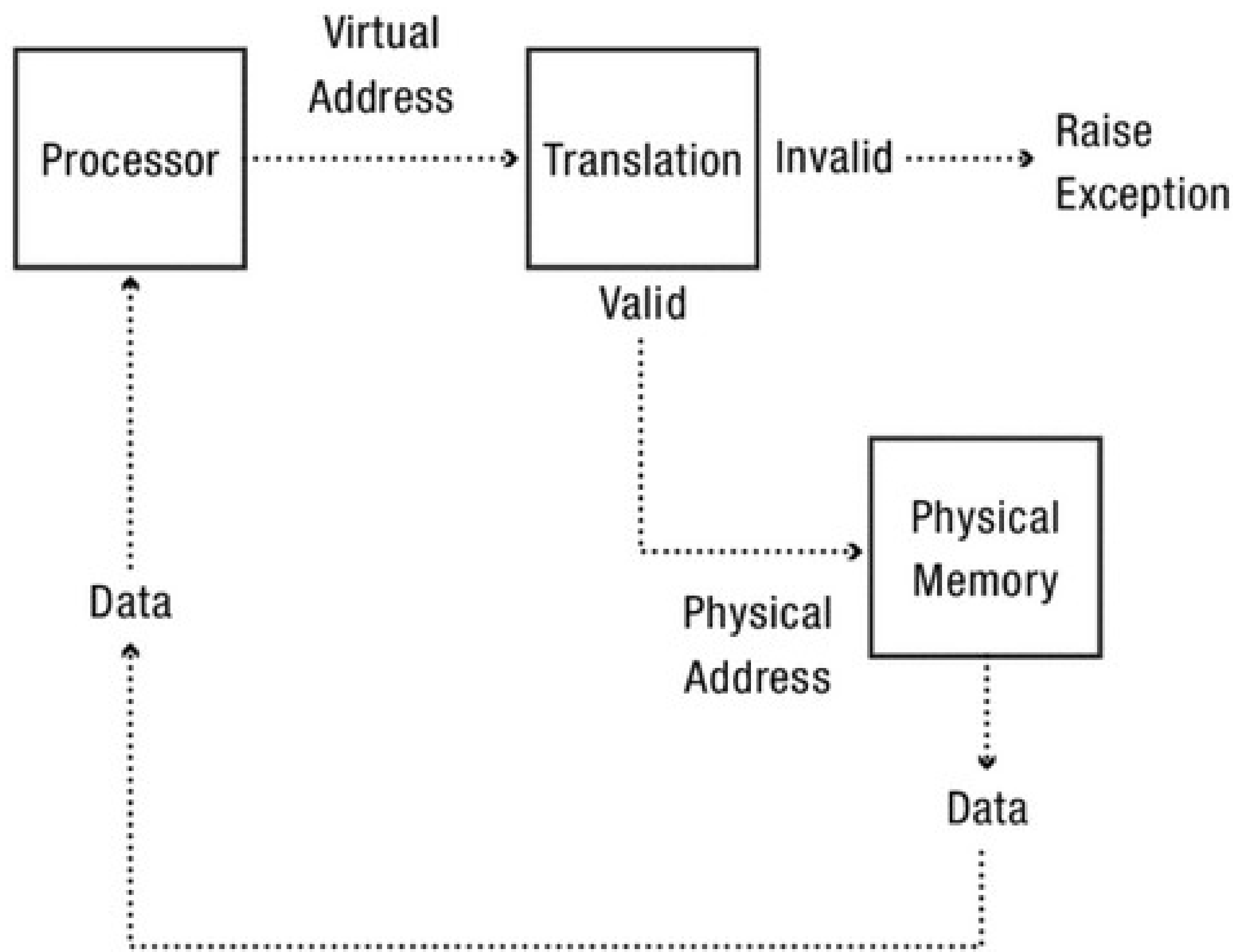
```asm
.L3:
movl    -4(%rbp), %eax
cltq
movzbl  -5(%rbp), %edx
movb    %dl, -1008(%rbp,%rax)
addl    $1, -4(%rbp)
.L2:
cmpl    $999, -4(%rbp)
jle     .L3
```

Virtual memory

- Program addresses
  - are independent on the physical location
- memory manager
  - translates addresses
    - virtual -> physical
  - verifies permitions
- Address Translation



Address Translation Concept

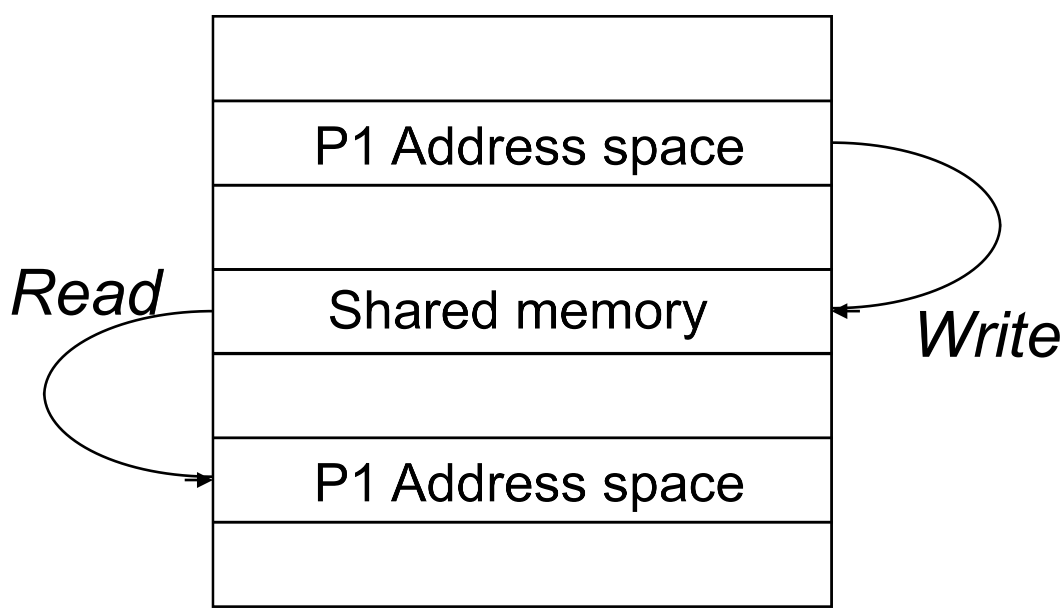


Adress translation

- Process isolation
- Interprocess communication
- Shared code
- Program debugging
- Efficient I/O
- Memory mapped files
- Virtual memory
- Checkpointing
- Process migration
- Information flow control
- DSM

Shared memory

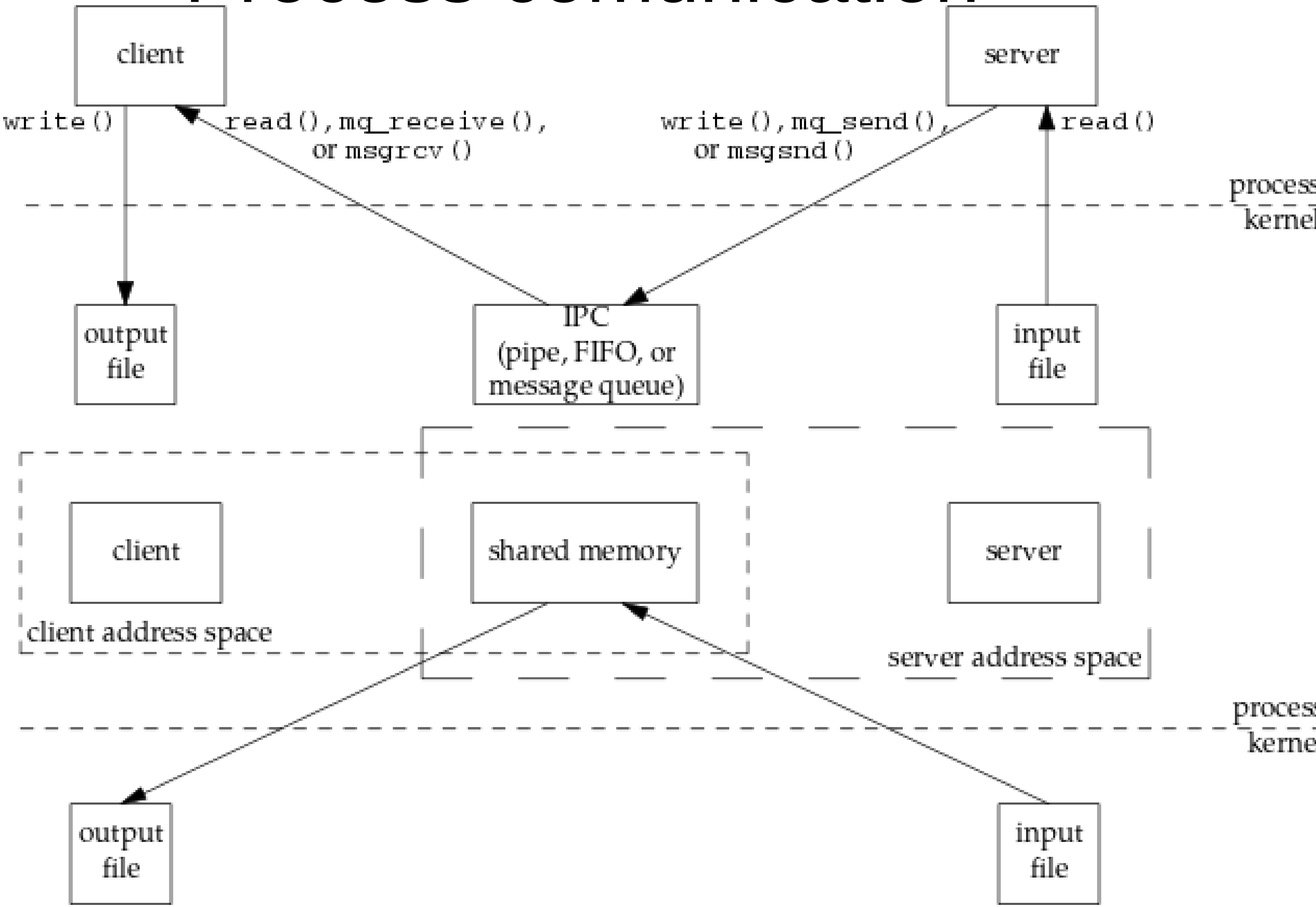
- Processes use regular variables/vector to communicate
  - Available in shared memory
- Should be explicitly created.
  - Each process has its own address space



Shared memory Advantages

- Random Access
  - you can update a small piece in the middle of a data structure, rather than the entire structure
- Efficiency
  - unlike message queues and pipes,
    - copy data between user memory ↔ kernel memory
  - shared memory is directly accessed
  - Shared memory resides in the user process memory
    - Is shared among other processes

Process comunication



Shared memory Disadvantages

- No automatic synchronization
  - In pipes or message queues
  - Programmer has to provide synchronization
    - Semaphores or signals.
- Pointers are only valid within a given process.
  - Pointer offsets cannot be assumed to be valid across inter-process boundaries.
  - This complicates the sharing of linked lists or binary trees.
- Variables are “produced by the compiler”
  - Names can not be used to access shared memory

Shared memory in *NIX

- System V shared memory
  - Original shared memory mechanism, still widely used
  - Sharing between unrelated processes
- Shared mappings – mmap
  - Shared file mappings
  - Sharing between unrelated processes, backed by filesystem
  - Shared anonymous mappings
    - Sharing between related processes only (related via fork())
- POSIX shared memory
  - Sharing between unrelated processes, without overhead of filesystem I/O
  - Intended to be simpler and better than older APIs

Shared memory in *NIX

- Programming steps
  - Define Shared data structure
  - Creation of memory segment
  - Configuration
  - Assignment to address
  - Access
  - Disconnection
  - Destruction

In multiple processes

System V shared memory

- Shared memory operations
  - shmget
    - allocates a shared memory segment
  - shmctl
    - allows the user to receive information on a shared memory segment,
    - set the owner, group, and permissions of a shared memory segment,
    - destroy a segment
  - shmat
    - attaches the shared memory segment (identified by shmid) to the address space of the calling process
  - shmdt
    - detaches the shared memory segment (located at the address specified by shmaddr) from the address space of the calling process

System V shared memory

- int shmget(key_t key, size_t size, int shmflg);
  - key known by all processes
  - Flags - IPC_CREAT | 0666

- void *shmat(int shmid, const void *shmaddr, int shmflg);
  - Shmid – returned by shmget
  - shmaddr – NULL or other address
  - Shmflg - SHM_EXEC   SHM_RDONLY

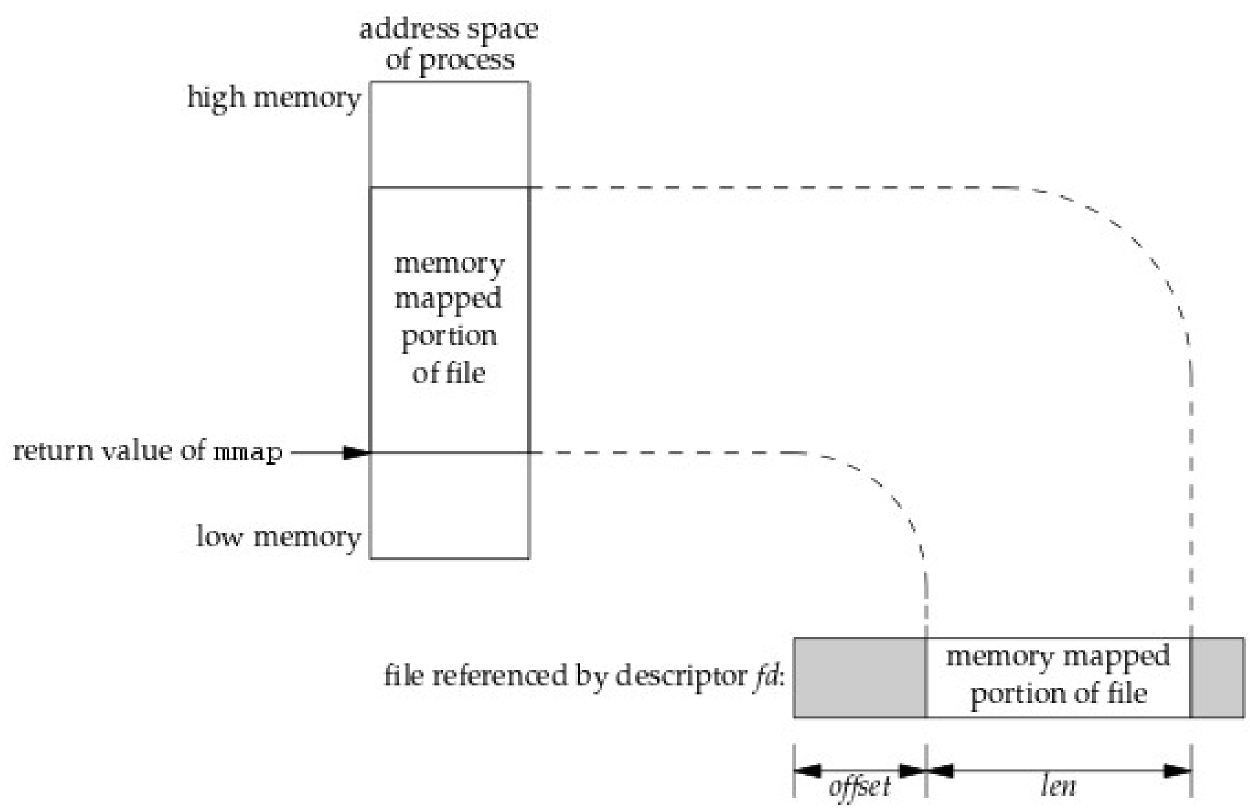
System V shared memory

- char * shm;
- key = 5678;
- /* Create the segment */
- if ((shmid = shmget(key, SHMSZ, 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);
- }

- Repeated on several processes
  - Key must be known
- Followed by fork
  - Shm value is shared

Memory mapped files

- Access a file contect
  - With memory access operations
  - Assignments e accesses
-



Memory mapped files

- void *mmap(void * addr, size_tlen ,

  int prot, int flags,

  int fd, off_toffset);

  - Prot –
    - PROT_READ PROT_WRITE PROT_EXEC PROT_NONE
  - Flags
    - MAP_SHARED MAP_PRIVATE MAP_FIXED
- int munmap(void * addr, size_t len);

Memory access

- After mmap a variable contains a pointer to region
  - Programmer can access that memory as a
    - Pointer to variable
    - Vector
- Is it possible to create linked lists in shared memory?
  - No. mmap in different processes returns different addresses

Memory mapped files

- Int * ptr;
- fd = open(argv[1], O_RDWR | O_CREAT, FILE_MODE);
- ptr = Mmap(NULL, sizeof(int), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
- Close(fd);

Mmap and fork

- Open could be avoided
  - Parent and son share the address to memory
  - Memory is shared among processes
- 4.4BSD introduced anonymous memory sharing
  - Flag – MAP_SHARED|MAP_ANON
  - Fd -1
- ptr = Mmap(NULL, sizeof(int),

```
                    PROT_READ | PROT_WRITE,
                    MAP_SHARED | MAP_ANON,
                    -1, 0);
```

POSIX shared memory

- Mmap
  - File system incurrs overhead
- Sharing between unrelated processes, without overhead of filesystem I/O
- Intended to be simpler and better than older APIs
- New function to create memory regions

POSIX shared memory

- Create/opens a shared memory space
  - fd_mem = shm_open("/myregion", /*region name*/
  - O_CREAT | O_RDWR, 0600);
  - Memory regions are created with size 0
- Assign a size
  - ftrucate (fd_mem, sizeof(int))
  - If the object has already been sized by another process, you can get its size with the fstat function
- A global region has been created
  - Data stored is kernel persistent
  - But still inaccessible by processes → use mmap with fd_mem

- int fd = shm_open(memname,
- O_CREAT | O_TRUNC | O_RDWR, 0666);
- if (fd == -1)
- error_and_die("shm_open");
- int r = ftruncate(fd, sizeof(int));
- if (r != 0)
- error_and_die("ftruncate");
- int *v_int = mmap(0, sizeof(int),
- PROT_READ | PROT_WRITE, MAP_SHARED,
- fd, 0);

Shutdown

- Close the shared memory object
  - close(fd_mem)
- Unmap the shared memory object:
  - munmap (pointer, SHM_SIZE);
  - The address become free to be used.
- At this stage if other processes open and map a memory region
  - They can access data previously written
- To remove permanently the shared memory object:
  - shm_unlink (SHARED_MEMORY_NAME);
  - The object is effectively deleted after the last call to shm_unlink

Synchronization on shared memory

- Multiple processe accesing same variable
- Requirtes synchronization
-
- Mutexes / condition variables
- Semaphores

Mutexes across processes

```
shm = (buffer_t *)mmap(NULL, sizeof(buffer_t),
       PROT_READ|PROT_WRITE, MAP_SHARED, zfd, 0);
pthread_mutex_attr_init(&mattr);
pthread_mutexattr_setpshared(&mattr,
PTHREAD_PROCESS_SHARED);
pthread_mutex_init(&shmm->lock, &mattr);
fork()
```

- `shmm->lock` is in shared memory