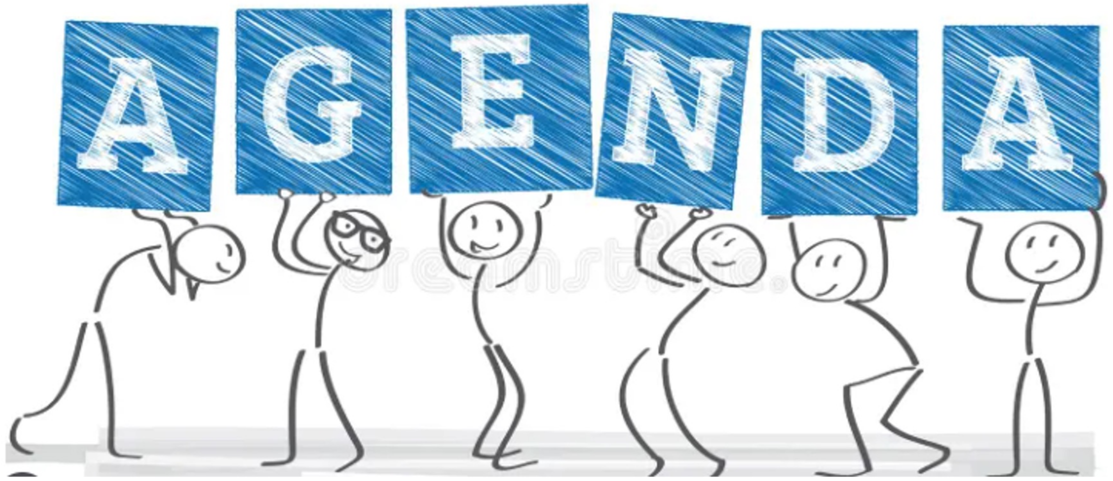
Operating Systems

Lecture 4.3

Message Queues and Shared Memory

Instructor: Muhammad Arif Butt, PhD

Lecture Agenda



Message Queues

- Overview
- Creating/Opening a Message Queue
- Sending/Receiving messages in a message queue
- Control Operations on System V Message Queues

Shared Memory

- Overview
- Creating/Attaching a Shared Memory Segment
- Reading/Writing in a Shared Memory Segment
- Detaching a Shared Memory Segment
- Control Operations on System V Shared Memory

System V vs POSIX IPC

System-V IPC vs POSIX IPC

System V IPC: `$ man 7 sysvipc`

- It is one of the earliest inter-process communication mechanisms introduced in UNIX systems supporting message queues, shared memory segments, and semaphores.
- Resources are identified using integer keys, often generated using the `ftok()` function.
- Resources are memory resident (in kernel space), and can persist in the system after the creating processes terminates. Need to be explicitly removed using control functions like `msgctl()`, `shmctl()`, or `semctl()`.
- System V API is considered less intuitive, with limited error reporting and fewer options for resource naming or permissions management compared to POSIX.

```
$ man 7 mq_overview
$ man 7 shm_overview
$ man 7 sem_overview
```

- **POSIX IPC:**
- It is a modern, user-friendly alternative supporting message queues, shared memory, and semaphores.
- Resources are named using human-readable strings (filenames), making management and debugging easy.
- Resources are memory-resident (in kernel space) and can persist in the system after the creating process terminates. They need to be explicitly removed using `mq_unlink()`, `shm_unlink()`, `sem_close()` or `sem_unlink()`.
- POSIX APIs provide more consistent, standardized interfaces across UNIX-like systems, with improved error handling and richer feature sets (e.g., notification via signals, timeout options).

System V MQ vs POSIX MQ

1. **Message Retrieval by Priority:**
   - In System V message queues, the reader can specify and can retrieve a message of specific priority.
   - The POSIX message queues always returns the oldest message with the highest priority first.

2. **Notification on Message Arrival:**
   - POSIX message queues allows the generation of a signal (a synchronous notification with `mq_notify()`), when a message arrives on an empty queue.
   - System V message queues do not provide built-in notification mechanisms. You must poll the queue or use external signalling techniques.

API for System V IPC

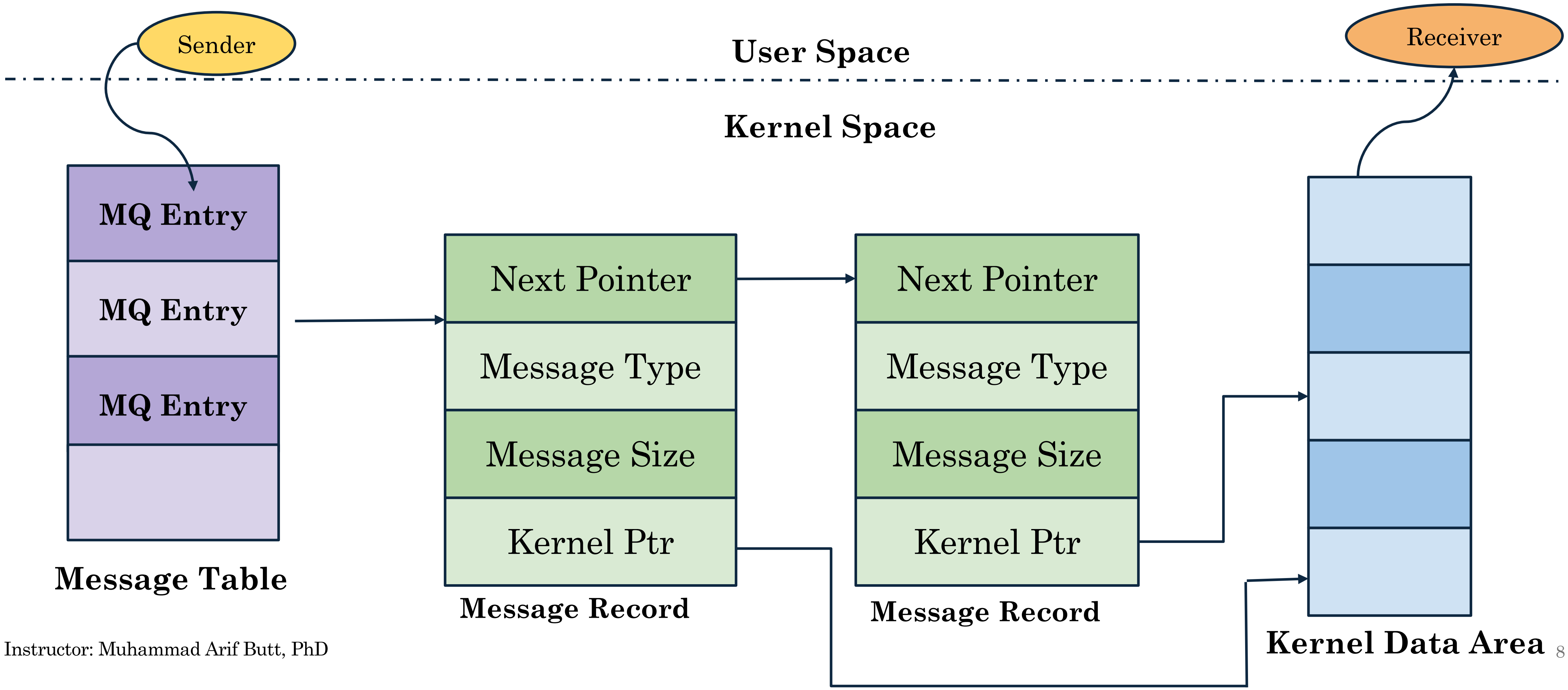
Summary of programming interfaces for **System V IPC** objects:

Interface	Message Queues	Shared Memory	Semaphores
Header file	`<sys/msg.h>`	`<sys/shm.h>`	`<sys/sem.h>`
Associated DS	`msqid_ds`	`shmid_ds`	`semid_ds`
Create/Open object	`msgget()`	`shmget()+shmat()`	`semget()`
Close Object	none	`shmdt()`	none
Control Operations	`msgctl()`	`shmctl()`	`semctl()`
Performing IPC	`msgsnd()`, `msgrcv()`	Access memory in shared region	`semop()`

Overview of Message Queues

Overview of a Message Queue

Message Queue are used to pass messages between related or unrelated processes executing on same machine. It can be thought of like a linked list of messages in kernel space. Processes with adequate permissions can put messages on to the queue and processes with adequate permissions can remove messages from the queue.



Overview of a Message Queue (cont…)

A Message Queue is stored entirely in kernel space and is organized into three main components:

- **Message Table:**
  - Resides in the kernel and contains entries for all active message queues in the system.
  - Each Message Queue Entry holds metadata about the queue and a pointer to the first message record in that queue.
  - This acts as the "directory" for all queues, allowing the kernel to manage multiple queues simultaneously.
- **Message Record:**
  - Represents an individual message stored within the queue.
  - Each record contains:
    - Next Pointer → The address of the next message in the queue, forming a linked list.
    - Message Type → Helps processes selectively read specific messages.
    - Message Size → Indicates the number of bytes in the message's data section.
    - Kernel Data Pointer → Points to the location in kernel memory where the actual message data is stored.
- **Kernel Data Area:**
  - A dedicated space in kernel memory where the actual contents of messages are stored.
  - All queues in the system share this memory pool, but each message record's Kernel Data Pointer ensures the right process accesses the right data.

https://elixir.bootlin.com/linux/v5.19.17/source/include/linux/msg.h#L9

https://elixir.bootlin.com/linux/v5.19.17/source/ipc/msg.c#L48

Named Pipes vs Message Queues

Aspect	Named Pipes (FIFOs)	Message Queues (System V / POSIX)
Persistence	Process-persistent; data disappears when no process has the pipe open.	Kernel-persistent; remains until explicitly removed, even after processes terminate.
Data Structure	Byte stream with no message boundaries; read as continuous bytes.	Maintains discrete, delimited messages; each read retrieves one complete message.
Blocking Behavior	`write()` blocks until a reader is present (unless non-blocking is used).	`msgsnd()` / `mq_send()` does not require a reader; message is enqueued.
Prioritization	No message ordering or priority; strictly FIFO data flow.	Supports message priorities; messages sorted by priority and timestamp.
Monitoring Status	No built-in way to check internal state.	Can query status (e.g., message count, size) via `msgctl()` or `mq_getattr()`.
Communication Mode	Requires both ends (reader/writer) to be active simultaneously.	Supports asynchronous communication; sender and receiver can run independently.

Implementing Message Queues using System V API

Creating/Opening a System V Message Queue

```
int msgget(key_t key, int msgflag);
```

- To create a brand new message queue or to get the identifier of an existing queue we use the `msgget()` system call, which on success returns a unique message queue identifier. This identifier is then used in all later operations on that message queue, allowing multiple processes to communicate or share resources through it.
- If a message queue associated with the first argument (`key`) already exist, the call returns the identifier of the existing message queue, otherwise it creates a new message queue.
- For the first argument **key**, we have two options:
  - Use `IPC_PRIVATE` constant. For related processes, the parent process creates message queue prior to performing a `fork()`, and the child inherits the returned message queue identifier. For unrelated processes we can use this constant, but in that case the creator process has to write the returned message queue identifier in a file that can be read by the other process.
  - Use the `ftok()` library call to generate unique key, and then use that key as first argument to `msgget()` to either generate a new message queue identifier or get an existing one.
- The second argument **`msgflag`** is normally `IPC_CREAT|0666`. If no message queue exists with the given key, create a new one with the specified permission mode.

Creating/Opening a System V Message Queue

```
int ftok(const char* pathname, int proj_id);
```

- The key returned by `ftok()` (file-to-key conversion) is a 32 bit value, created by taking:
  - Least significant 8 bits from `proj` argument.
  - Least significant 8 bits of minor device number of the device containing the filesystem on which the file in the first argument reside.
  - Least significant 16 bits of the inode number of the file referred by first argument `pathname`.

Message Queue Identifier

IPC Identifier

- An IPC identifier is similar to a file descriptor, but instead of referring to files, it refers to an IPC object.
  - **File descriptor** → Exists only within a specific process and is not visible to others.
  - **IPC identifier** → Belongs to the IPC object itself and is visible system-wide, meaning all processes can reference the same object using the same identifier.

Same Identifier for All

- All processes accessing the same IPC object will use the same identifier to interact with it.
- If a process already knows the identifier, it can skip the `get` call entirely and directly access the object.
- The process that creates the object can share the identifier with others, by writing it to a file or passing it through another IPC mechanism. This allows multiple processes to coordinate and use the same resource.

Sending Messages

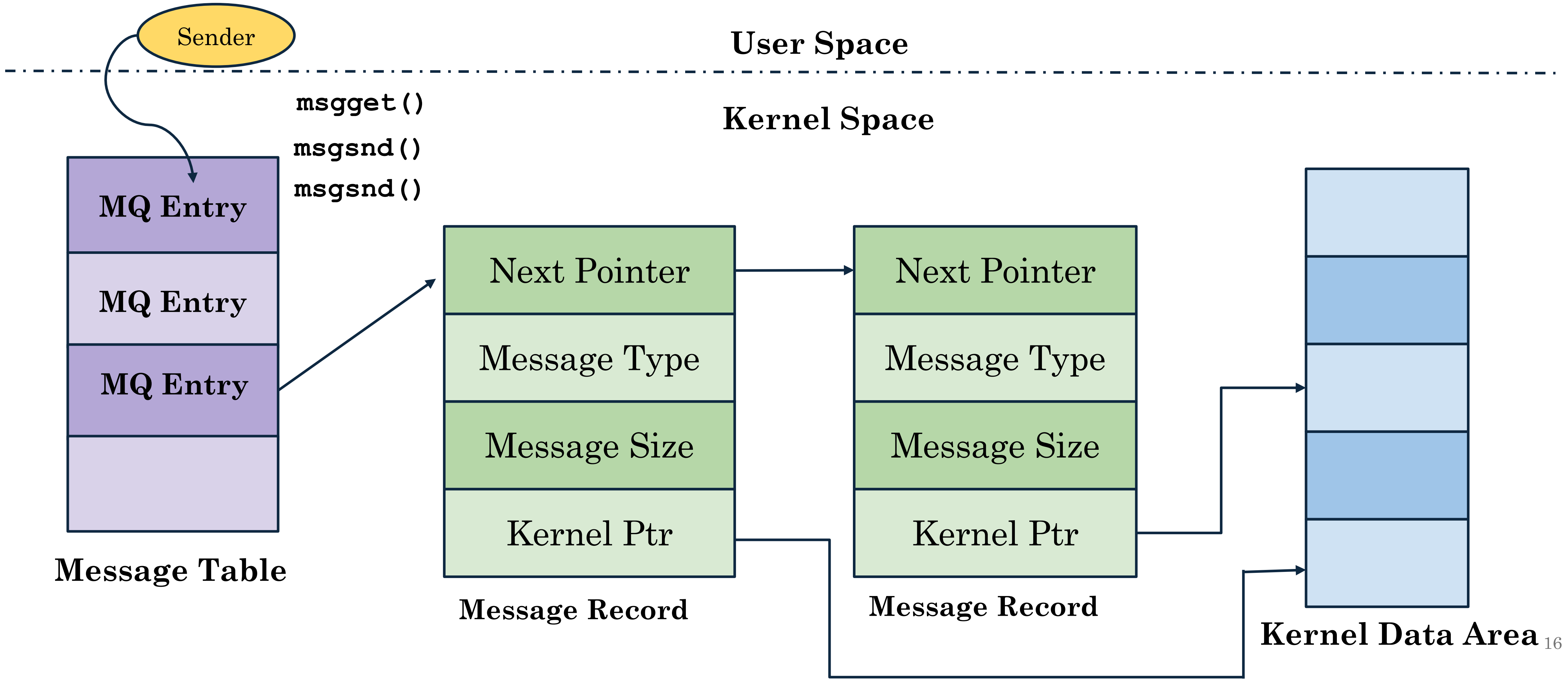
```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg);
```

- The `msgsnd()` system call is used to send a message to the message queue identified by its first argument (`mqid`), which is the message queue identifier, generated using the `msgget()` system call.
- The second argument is a pointer to a structure of type `msgbuf` having following two fields:

```
struct msgbuf{
        long mtype; //used to retrieve a selective message, must be a positive integer
        char mtext[512]; //actual message
}
```

- The third argument `msgsz` is the size of message data (`mtext`), excluding the `mtype` field.
- The fourth argument `msgflag` can be 0 or `IPC_NOWAIT`.
    - 0 → Blocking mode (wait if queue is full).
    - `IPC_NOWAIT` → Non-blocking mode (return error if queue is full).

Illustration of Sending Messages



Example: sender.c

```c
#define SIZE 512
struct msgbuf{
  long mtype;
  char mtext[SIZE];
};
int main(){
  key_t key = ftok("./myfile", 65);
  int qid = msgget(key, IPC_CREAT | 0666);
  struct msgbug msg1;
  msg1.mtype = 10;
  strcpy(msg1.mtext, ”Learning is fun with Arif\n");
  msgsnd(qid, &msg1, sizeof(msg1.mtext), 0);
  return 0;
}
```

```
$ gcc sender.c –o sender
$ ipcs –q
key msqid owner perms used-bytes messages
$ ./sender
$ ipcs –q
key msqid owner perms used-bytes messages
$ ./sender
$ ipcs –q
key msqid owner perms used-bytes messages
$ ipcrm –q <msqid>
$ ipcs –q
key msqid owner perms used-bytes messages
```

Demonstration

Sending Messages in MQ

`Lec4.3/msgq/sender.c`

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Receiving Messages

```
int msgrcv(int msqid, void *msgp, size_t maxmsgsz, long msgtype, int msgflg);
```

The `msgrcv()` system call is used to read and remove a message from the specified message queue, copying its contents into the buffer pointed to by `msgp`.

- **`msqid`** → Message Queue Identifier (returned by **`msgget()`**).
- **`msgp`** → Pointer to the message structure where the received data will be stored.
- **`maxmsgsz`** → Maximum number of bytes available for `mtext`.
- **`msgtype`** → Controls which message to retrieve (see **Message Selection** below).
- **`msgflg`** → Normally kept as IPC_NOWAIT, i.e., return immediately if no matching message is available.

Messages need not to be read in the order in which they are sent. Instead we can select messages accordingly to the value in the `mtype` field of message. This selection is controlled by `msgtype` argument, which can take following values:

msgtype	**Description**
`msgtype == 0`	First message from queue is removed and returned to calling process
`msgtype > 0`	First message from queue whose mtype field equals to msgtype is removed and returned to caller
`msgtype < 0`	First message of the lowest `mtype` field less than or equal to absolute value of `msgtype` is removed & returned to the caller

Receiving Messages

Suppose that we have a message queue containing messages as shown and we perform `msgrcv()` calls of the following form:

`msgrcv(id,&msg,maxmsgsz,0,0);` Would retrieve msgs in following order:

1(mtypr=300)
2(mtypr=100)
3(mtypr=200)
4(mtypr=400)
5(mtypr=100)

`msgrcv(id,&msg,maxmsgsz,100,0);` Would retrieve msgs in following order:

2(mtypr=100)
5(mtypr=100)

Any further calls would block, since no message of type 100 remains.

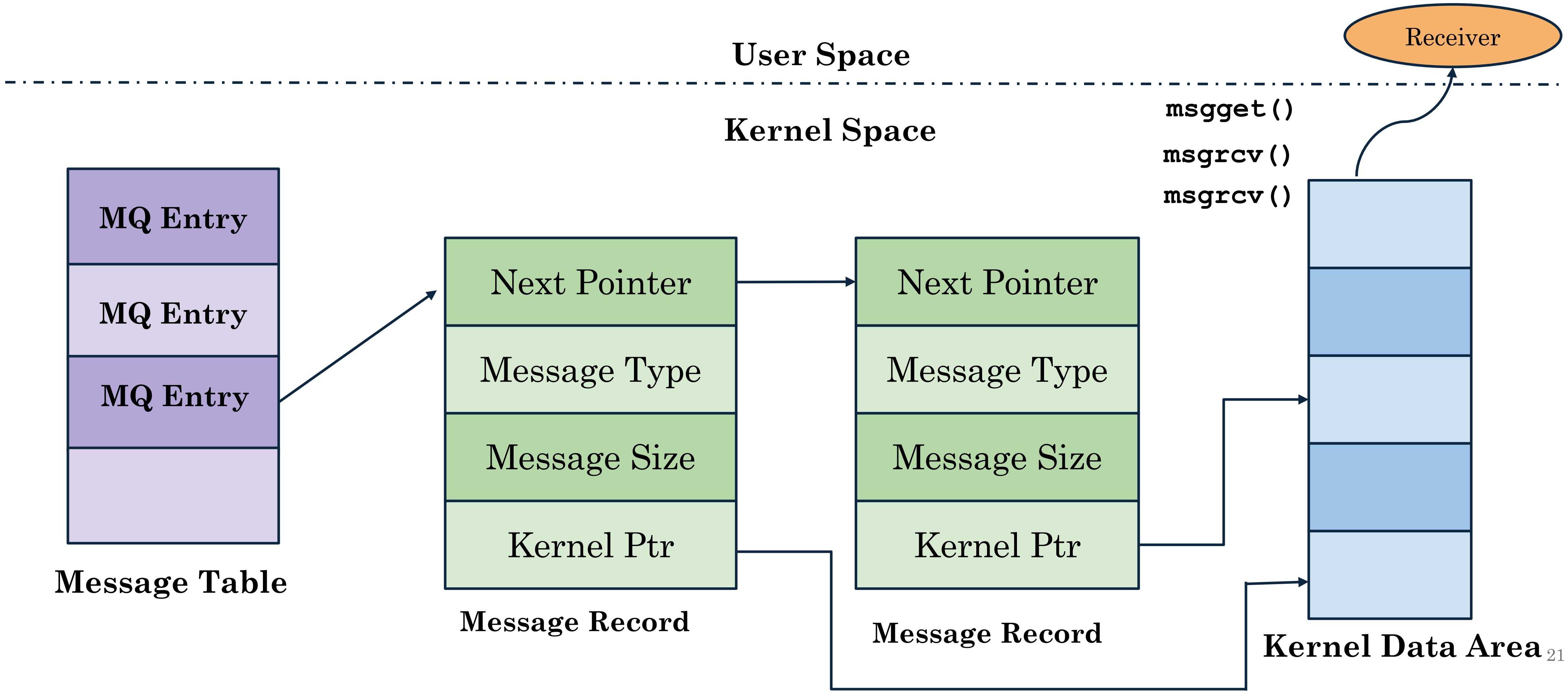
Queue position	Msg type	Msg Body
1	300	….
2	100	….
3	200	….
4	400	….
5	100	….

`msgrcv(id,&msg,maxmsgsz,-300,0);` Would retrieve msgs in following order:

2(mtypr=100)
5(mtypr=100)
3(mtypr=200)
1(mtypr=300)

Any further call would block, since type of the remaining message(400) exceeds 300

Illustration of Receiving Messages



Example: `receiver.c`

```c
#define SIZE 512
struct msgbuf{
  long mtype;
  char mtext[SIZE];
};
int main(){
  key_t key = ftok("./myfile", 65);
  int qid = msgget(key, IPC_CREAT | 0666);
  struct msgbuf msg;
  msgrcv(qid, &msg, SIZE, 0, IPC_NOWAIT);
  printf("Message Received: %s\n",msg.mtext);
  return 0;
}
```

```
$ ./sender
$ ipcs -q
$ ./sender
$ ipcs -q
key msqid owner perms used-bytes messages


$ ./receiver
Message Received: ...............
$ ipcs -q
key msqid owner perms used-bytes messages
$ ./receiver
$ ipcs -q
key msqid owner perms used-bytes messages
$ ipcrm -q <msqid>
$ ipcs -q
key msqid owner perms used-bytes messages
```

Demonstration

Receiving Messages from MQ

`Lec4.3/msgq/receiver.c`

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

System V Message Queue Control Operations

The ipcs and ipcrm Commands

- The **`ipcs`** command is used to display information about the active System V IPC objects on a Linux system, including message queues, shared memory segments, and semaphores. It provides details such as the key, identifier, owner, permissions, size, and status of these IPC resources. This command is particularly useful for system administrators and developers to monitor IPC usage and diagnose IPC. It can be customized with options like **`-q`** (message queues), **`-m`** (shared memory), **`-s`** (semaphores), **`-p`** (permissions) and **`-c`** (creator details) to filter the output.

- The **`ipcrm`** command is used to remove System V IPC objects from the system, such as message queues, shared memory segments, and semaphores, that are no longer in use or have been orphaned. Resources can be removed by their identifier (ID) or by their key. This is crucial for freeing up kernel memory and ensuring that unused IPC resources do not persist indefinitely. Common options include **`-m`** for shared memory, **`-q`** for message queues, and **`-s`** for semaphores, followed by the corresponding ID of the object to be deleted.

Limits Related to System V Message Queues

- In Linux operating system, the `/proc/sys/kernel/` directory contains tunable parameters related to kernel's behaviour, including limits, scheduling, messaging, and system control settings.
- Some important limits related to System V message queues are contained in following files. These control the kernel's behaviour regarding System V message queues and can be viewed or changed at runtime (as root):
  - `/proc/sys/kernel/msgmax` – Maximum size (in bytes) of a single message that can be sent to a System V message queue.
  - `/proc/sys/kernel/msgmnb` – Maximum number of bytes allowed in a single System V message queue (i.e., the queue size limit).
  - `/proc/sys/kernel/msgmni` – Maximum count of System V message queues allowed system-wide.

System V MQ Control Operations in C

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

The **msgctl()** is typically used for querying queue status, modifying permissions, or deleting System V message queues to free system resources.

- `msqid` → Message queue identifier returned by `msgget()`.
- `cmd` → Control command specifying the operation to perform (`IPC_STAT`, `IPC_SET`, `IPC_RMID`).
- `buf` → Pointer to a `struct msqid_ds` used to get or set message queue attributes. For deleting a message queue the third argument is set to `NULL`.

```
struct msqid_ds {
    struct ipc_perm msg_perm;  // Ownership and permissions (UID, GID, etc.)
    time_t          msg_stime; // Time of last msgsnd()
    time_t          msg_rtime; // Time of last msgrcv()
    time_t          msg_ctime; // Time of last change
    msgqnum_t       msg_qnum;  // Number of messages in the queue
    msglen_t        msg_qbytes;// Max number of bytes allowed in the queue
    pid_t           msg_lspid; // PID of last msgsnd()
    pid_t           msg_lrpid; // PID of last msgrcv()
};
```

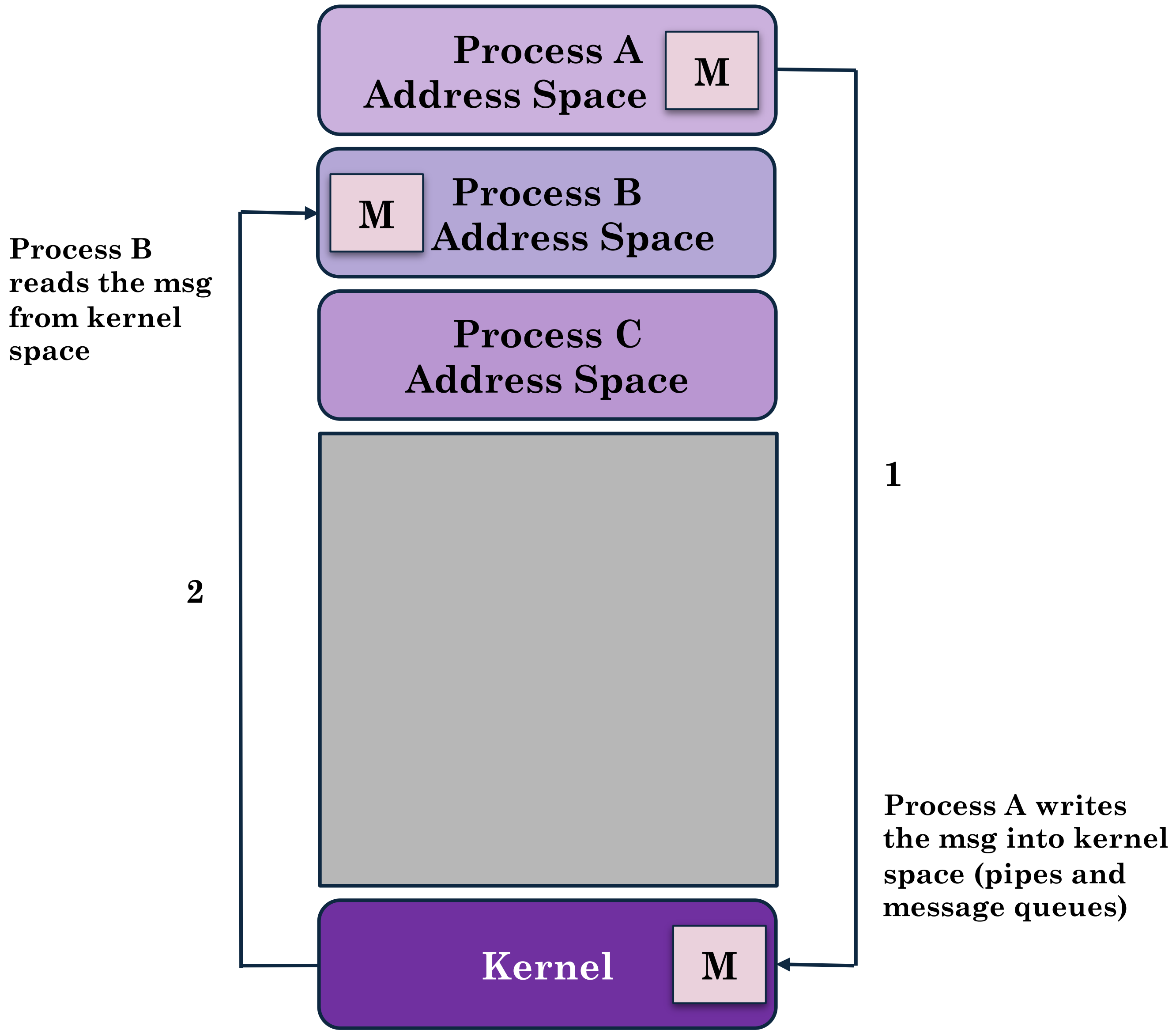
Overview of Shared Memory

Overview of Shared Memory

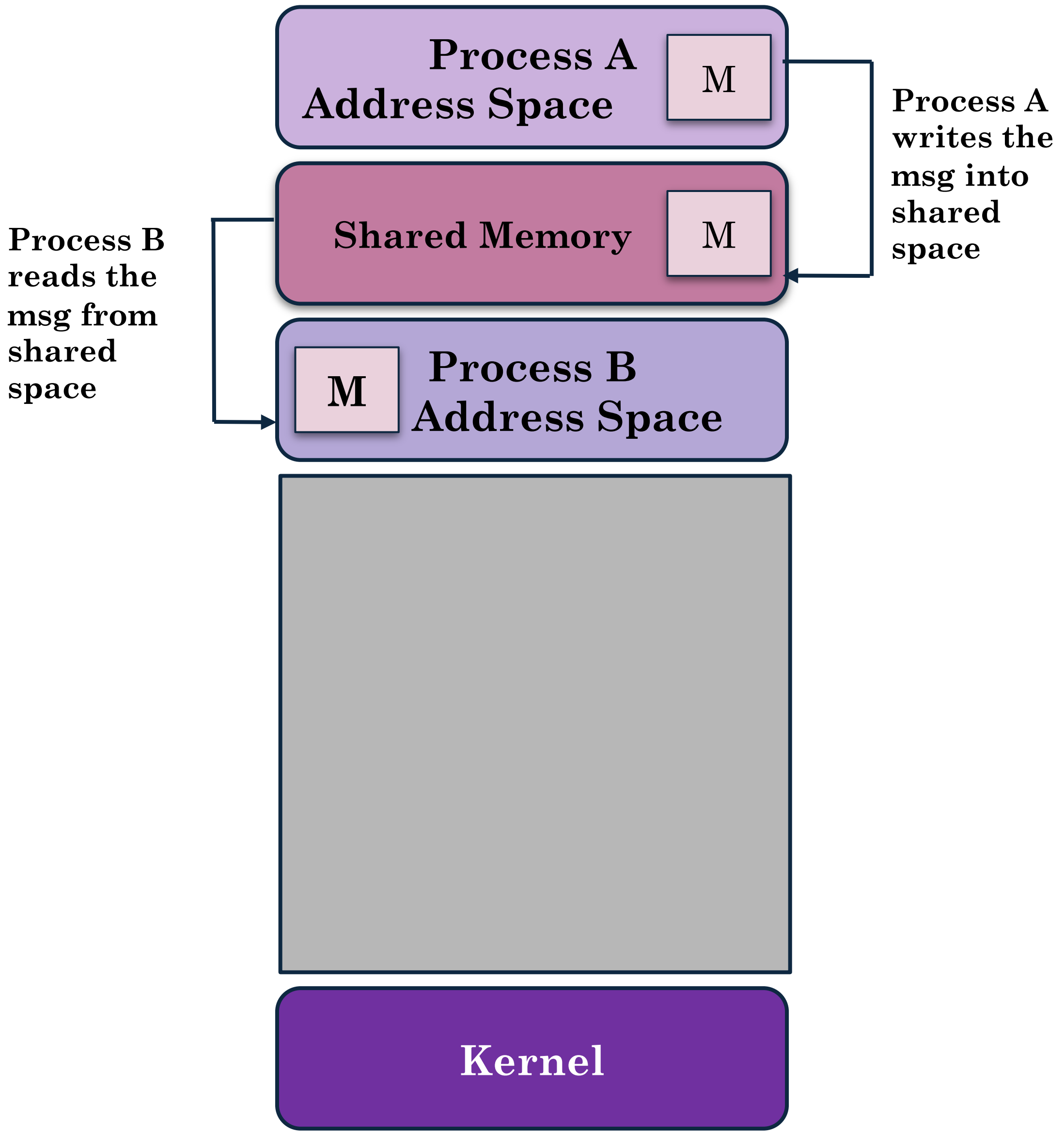
- Shared memory is an inter-process communication (IPC) mechanism that allows multiple processes to access a common region of memory, enabling high-speed data exchange. The kernel sets up this region by mapping the same physical memory pages into the address space of each participating process using page tables.
- Once the memory is mapped, no further kernel involvement is needed during access — making shared memory one of the fastest IPC mechanisms, as it avoids copying data between user and kernel space.

- **Key Characteristics of Shared Memory:**
  - **Fast Communication:** Processes can read from and write to shared memory directly, offering performance benefits for large or frequent data exchange.
  - **Non-Destructive Reads:** Unlike pipes or message queues (which remove data when read), shared memory allows reading without consuming the data.
  - **Requires Synchronization:** Due to concurrent access, proper synchronization (e.g., using semaphores or mutexes) is essential to avoid race conditions when one process is modifying data while another reads it.

Data Transfer vs Shared Memory

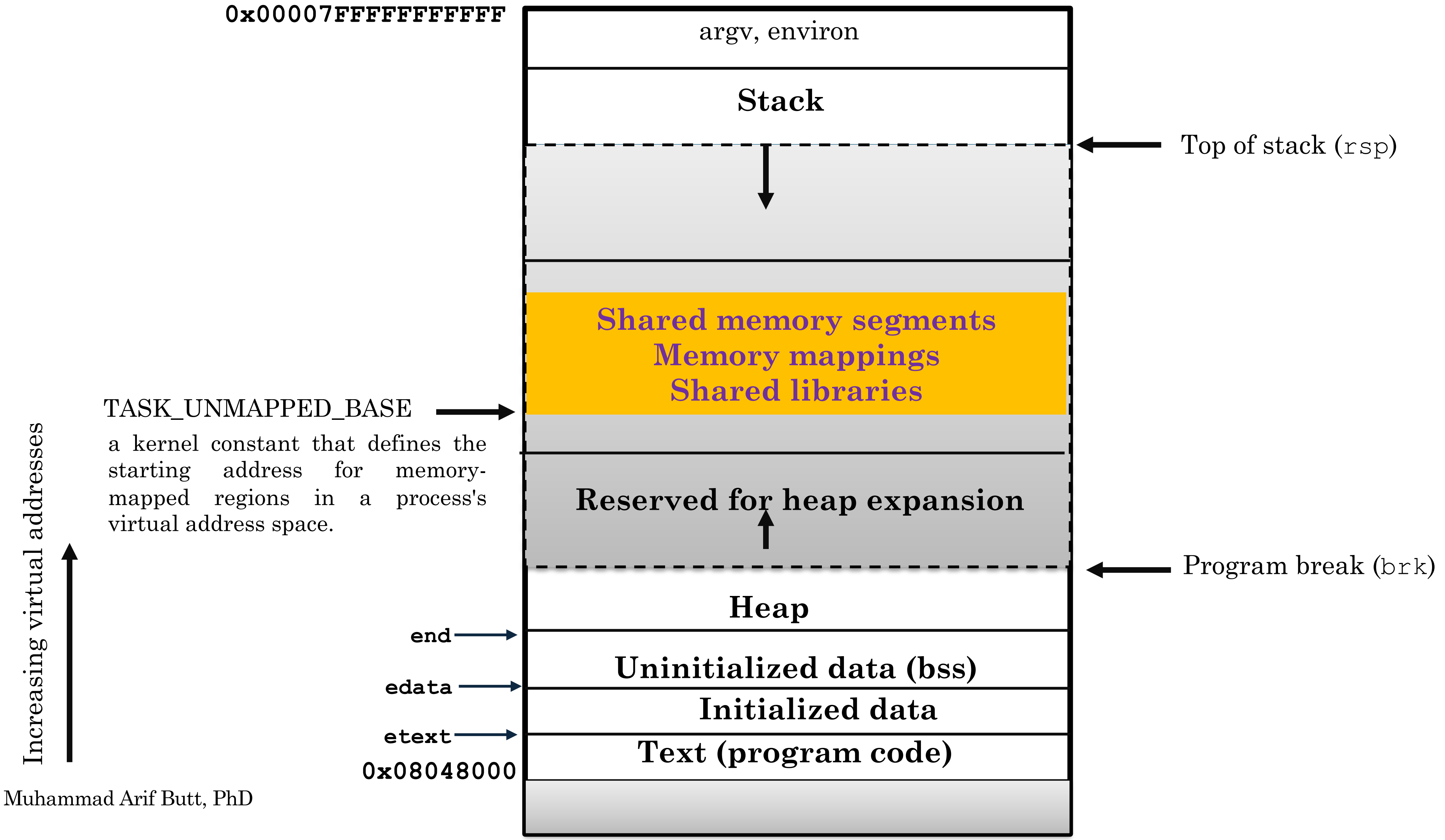


Process A Address Space
M
Process B Address Space
M
Process C Address Space
Kernel
M
1
2
Process B reads the msg from kernel space
Process A writes the msg into kernel space (pipes and message queues)



Process A Address Space
M
Shared Memory
M
Process B Address Space
M
Kernel
Process A writes the msg into shared space
Process B reads the msg from shared space

Shared Memory in Process Logical Address Space



a kernel constant that defines the starting address for memory-mapped regions in a process's virtual address space.

System V vs POSIX Shared Memory

Name and Identification:

- System V shared memory segments are identified using integer keys (often generated with `ftok`)
- POSIX shared memory objects are identified by string names, resembling filesystem paths.

Creation and Access:

- System V uses `shmget()`, `shmat()`, and `shmdt()` to create, attach, and detach shared memory segments.
- POSIX uses `shm_open()` and `mmap()` to create and map shared memory into a process's address space.

Clean-up and Deletion:

- System V shared memory must be manually deleted using `shmctl(..., IPC_RMID, ...)`.
- POSIX shared memory must be unlinked using `shm_unlink()` to remove the named object.

File System Integration:

- System V shared memory is not visible in the file system.
- POSIX shared memory objects appear under the `/dev/shm/` directory on most Linux systems, allowing easier inspection.

Implementing Shared Memory using System V API

Creating/Opening Shared Memory Segment

```
void* shmget(int key_t key, size_t size, int shmflag);
```

- To create a brand new shared memory segment or to get the identifier of an existing one we use the `shmget()` system call, which on success returns a unique shared memory identifier used in all later operations on that segment.
- If a shared memory segment with the given `key` already exists, the call returns its identifier, otherwise it creates a new segment.
- For the first argument **`key`**, we have two options:
  - Use `IPC_PRIVATE` constant to create a new segment. For related processes, the parent creates the segment before `fork()` and the child inherits the identifier. For unrelated processes, the creator must write the identifier to a file for others to read.
  - Use `ftok()` to generate a unique key for creating new or accessing existing segments.
- The second argument **`size`** specifies the segment size in bytes, rounded up to PAGE_SIZE multiples. It can be 0 when accessing existing segments but must specify the desired size when creating new ones.
- The third argument **`shmflg`** is normally `IPC_CREAT|0666`. In case if you want to access an existing segment, you can keep this argument as zero.

Attaching a Shared Memory Segment

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

- The `shmat()` system call is used to attach (map) a shared memory segment to the calling process's address space, which on success returns the address at which the shared memory segment has been attached. This allows the process to read from and write to the shared memory segment using regular memory operations.
- The first argument **`shmid`** is the shared memory identifier returned by `shmget()`. This identifies which shared memory segment to attach to the process.
- The second argument **`shmaddr`** is the address where the memory segment will be attached. The recommended and portable way is the specify NULL over here and let the OS Kernel select a suitable address.
- The third argument **`shmflg`** can be `SHM_RDONLY` to attach the shared memory segment for read-only access. We can place a zero over there for giving both read and write access.
- On success `shmat()` returns the address at which the shared memory segment is attached, which can be treated like a normal C pointer. We can assign the return value from `shmat()` to a pointer of some intrinsic data type or a programmer defined structure.

Detaching a Shared Memory Segment

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

- To detach a shared memory segment from the calling process's address space we use the `shmdt()` system call, which on success returns 0. This removes the shared memory segment from the process's virtual address space but does not destroy the segment itself. Deletion can be performed using the `shmctl()` system call.
- The only argument **`shmaddr`** must be the address returned by a previous `shmat()` call, specifying which shared memory segment to detach from the process. After successful detachment, the shared memory segment is no longer accessible from the calling process. Any attempt to access memory at that address will result in a segmentation fault.
- Detaching a shared memory segment does not affect other processes that may still have the same segment attached to their address spaces. However, the system decrements the attach count for the shared memory segment. When the attach count reaches zero and the segment is marked for deletion, and is actually removed from the system

System V Shared Memory Control Op

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

The **`msgctl()`** is typically used for querying queue status, modifying permissions, or deleting System V message queues to free system resources.

- `shmid` → Shared memory identifier returned by `shmget()`.
- `op` → Control command specifying the operation to perform (`IPC_STAT, IPC_SET, IPC_RMID`).
- `buf` → Pointer to a `struct shmid_ds` used to get or set shared memory segment attributes. For deleting a shared memory segment the third argument is set to `NULL`.

```
struct shmid_ds {
   struct ipc_perm shm_perm;  // Ownership and permissions
   size_t          shm_segsz; // Size of segment (bytes)
   time_t          shm_atime; // Last attach time
   time_t          shm_dtime; // Last detach time
   time_t          shm_ctime; // Creation time / last modification time via shmctl
   pid_t           shm_cpid;  // PID of creater
   …
   };
```

Example: writer.c

```
int main(){
   key_t key = ftok("./f1.txt", 65); //generate a unique key
//use above key and create a segment of size 1 KiB having read/write pmns to all
   int shmid = shmget(key, 1024, IPC_CREAT | 0666);
   char *buffer = (char*)shmat(shmid, NULL, 0); //attach shared memory to its address space
   printf(“Please enter a string to be written in shared memory:\n”);
   fgets(buffer, 512, stdin); //read from stdin and write to shared memory pointed to by buf
   printf(“\nData has been written in shared memory. Bye\n”);
   shmdt(buffer); //process detach the shared memory segment and exits
   return 0;
}
```

```
$ gcc. writer.c –o writer
$ ipcs –m
key shmid owner perms bytes nattch status
$ ./writer
```

```
$ ipcs –m
key shmid owner perms bytes nattch status
$ ipcrm –m <shmid>
$ ipcs –m
```

Demonstration

Writing to Shared Memory

Lec4.3/shm/writer.c

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

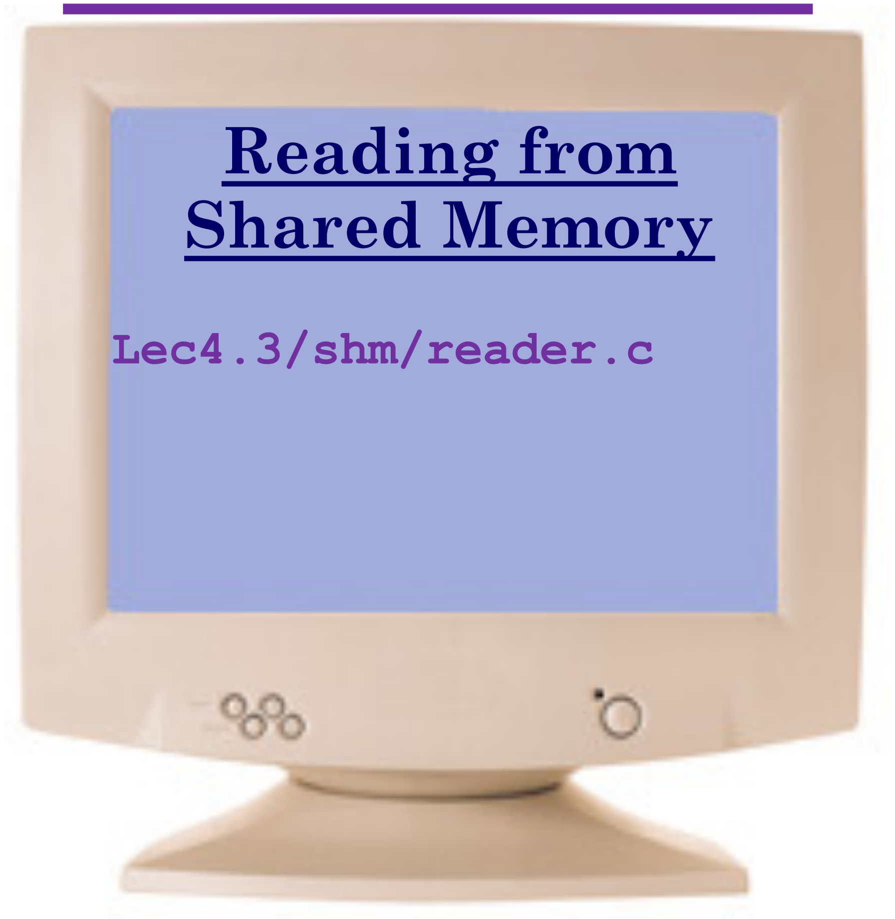
Example: reader.c

```
int main(){
   key_t key = ftok("./f1.txt", 65); //generate key with same parameters as in writer
//use above key and create/get segment ID of size 1 KiB having read/write pmns to all
   int shmid = shmget(key, 1024, IPC_CREAT | 0666);
   char *buffer = (char*)shmat(shmid, NULL, 0); //attach shared memory to its address space
   printf(“Data read from memory: %s\n”, buffer);
   shmdt(buffer); //process detach the shared memory segment and exits
   //shmctl(shmid, IPC_RMID, NULL); //destroy the shared memory from kernel
   return 0;
}
```

```
$ gcc. reader.c –o reader
$ ./reader
Data read from memory: Learning is fun with Arif
$ ./reader
Data read from memory: Learning is fun with Arif
```

```
$ ipcs –m
key shmid owner perms bytes nattch status
$ ipcrm –m <shmid>
$ ./reader
Data read from memory:
```

Demonstration



Reading from Shared Memory

Lec4.3/shm/reader.c

GitHub Code Repository Link: *https://github.com/arifpucit/OS-Codes*

Example: Race Condition

```
void inc();
void dec();
long *balance;
int main(){
  key_t key1 = ftok("file1", 65);
  int shm_id1 = shmget(key1, 8, IPC_CREAT | 0666);
  balance = (long*)shmat(shm_id1, NULL, 0);
  *balance=0;
  printf(”Value of balance is: %ld\n", *balance);
  int cpid = fork();
  if (cpid == 0){
      inc();
      shmdt(balance);
      exit(0);
  }else{
      dec();
      waitpid(cpid,NULL,0);
      printf(”Value of balance is: %ld\n", *balance);
      shmdt(balance);
      shmctl(shm_id1, IPC_RMID, NULL);
      return 0; }}
```

```
void inc(){
   long i;
   for (i = 0; i < 100; i++)
       *balance = *balance + 1;
}
void dec(){
   long i;
   for (i = 0; i < 100; i++)
       *balance = *balance - 1;
}
```

Demonstration

Race Condition
Shared Memory

Lec4.3/shm/cs-problem.c

GitHub Code Repository Link: https://github.com/arifpucit/OS-Codes

Limits Related to System V Shared Memory

- In Linux operating system, the `/proc/sys/kernel/` directory contains tuneable parameters related to kernel's behaviour, including limits, scheduling, messaging, and system control settings.
- Some important limits related to System V shared memory are contained in following files. These control the kernel's behaviour regarding System V shared memory and can be viewed or changed at runtime (as root):
  - `/proc/sys/kernel/shmmax` – Maximum size (in bytes) of a single System V shared memory segment. This determines the largest shared memory segment that can be created with `shmget()`.
  - `/proc/sys/kernel/shmall` – Maximum total amount of shared memory (in pages) that can be allocated system-wide. This is the cumulative limit across all shared memory segments.
  - `/proc/sys/kernel/shmmni` – Maximum number of System V shared memory segments allowed system-wide. This controls how many separate shared memory segments can exist simultaneously (4096).
- Some other files related to System V Shared memory inside `/proc/` are:
  - `/proc/sysvipc/shm` – Displays comprehensive details of all existing shared memory segments including IDs, keys, ownership, size, process information (creator and last accessor), attach count, and timestamps.
  - `/proc/<PID>/maps` – Shows memory mappings for a specific process, revealing which shared memory segments are attached with their virtual addresses and access permissions.
  - `/proc/meminfo` – Provides system-wide memory statistics including overall shared memory usage.

To Do

- Watch SP video on Message Queues: https://www.youtube.com/watch?v=UAbMS3kYV5s
- Watch SP video on Shared Memory: https://www.youtube.com/watch?v=IzhnAW8u1iQ

Coming to office hours does NOT mean that you are academically weak!