Operating System Concepts

Lecture 7: Interprocess Communication

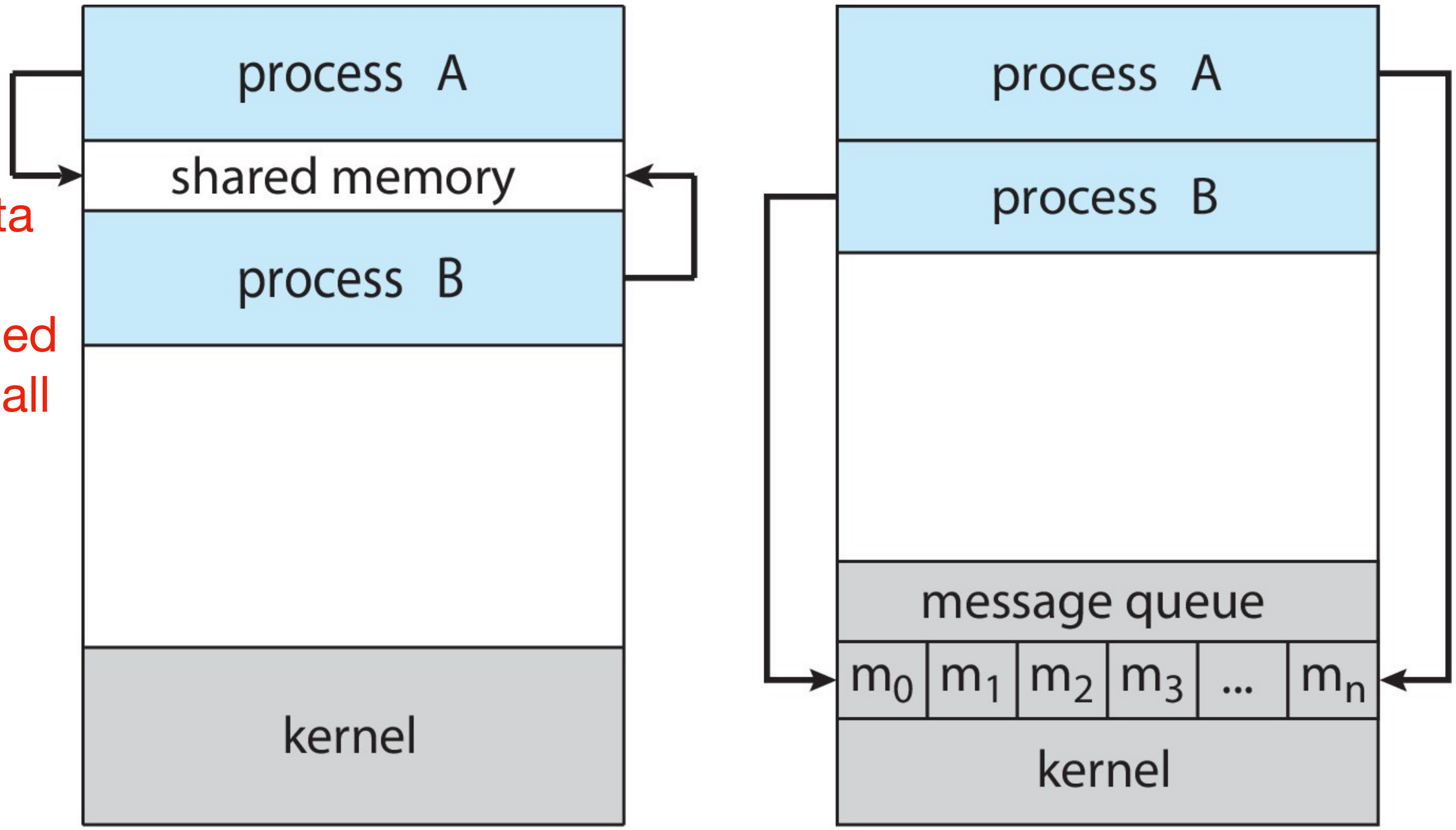
Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today's class

- Fundamental models of interprocess communication
  - Shared memory
  - Message passing

Two fundamental approaches

processes read/write data from/to this shared memory region established by the `mmap()` system call



(a) (b)

queue identifier must be known by both processes

Two fundamental approaches

- Message passing
  - kernel intervention (through system calls) required for every send/receive message operation
  - **pros:** no conflicts; extensible to communication in distributed systems
  - **cons:** high overhead for large data: copying data, crossing protection domains
- Shared memory: a memory segment is attached to the address space of each process
  - system calls are only required to establish shared memory regions; processes must ensure that they do not access a location concurrently
  - **pros:** **fastest** form of interprocess communication: set up shared memory once, then access w/o crossing protection domains
  - **cons:** **synchronizing** access to the shared region is necessary; error prone; difficult to support across machine boundaries

Communication using message passing

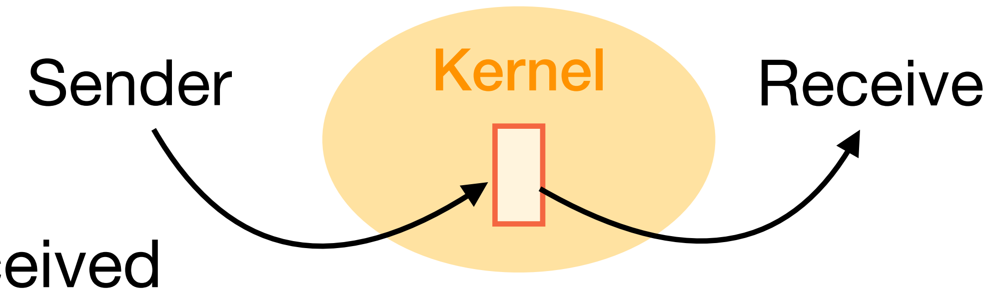
- Distributed systems typically communicate using message passing
  - each process needs to be able to name the other process or the mailbox/port/message queue (POSIX implementation)
- A common system message queue is a linked list of messages stored within the kernel address space and identified by a message queue **identifier,** which is shared with the cooperating processes to access the queue
- OS is responsible for handling the messages
  - copies them, notifies receiving process, etc.

Properties of message passing systems

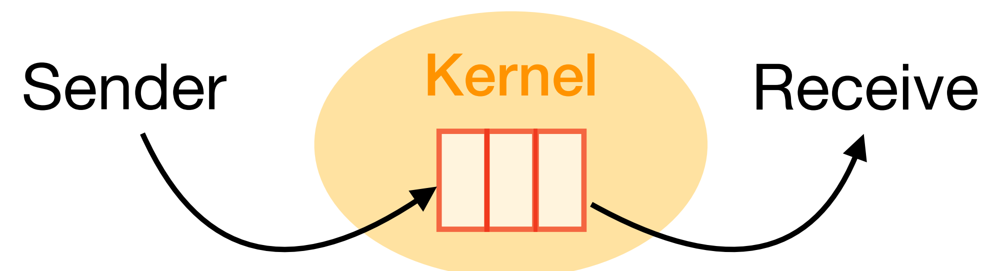
- Direction
  - simplex (one-way)
  - half-duplex (two-way, but only one-way at a time)
  - full-duplex (two-way)
- Message boundaries
  - datagram model: with message boundaries
  - byte stream model: no message boundaries
- Connection model
  - connection-oriented model: recipient is specified at connection time, so it does not need to be specified for individual send operations
  - connectionless models: recipient is specified as a parameter to each send operation
- Reliability
  - messages can get lost, corrupted, or reordered

Message passing issues

Does a send/receive operation block?



- blocking operations
  - sender has to wait until its message is received
  - receiver has to wait if no message is available



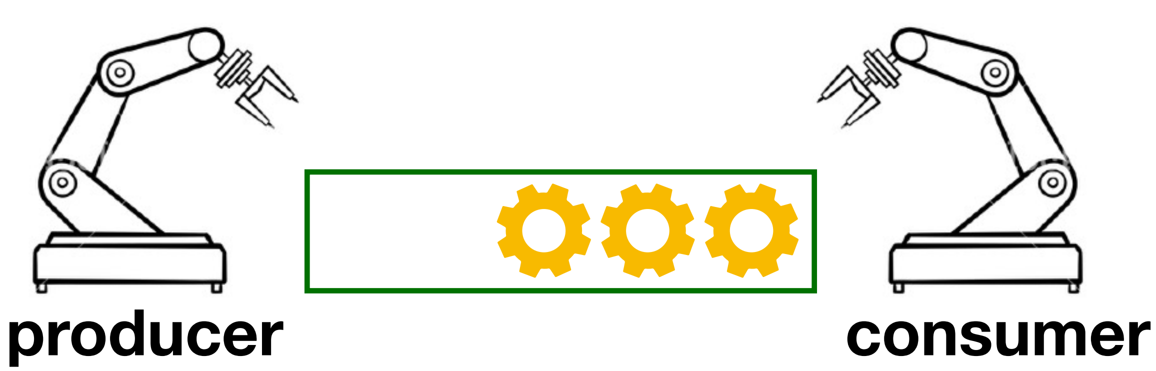
- non-blocking operations
  - send operation returns immediately
  - receive operation returns if no message is available
- partially blocking/non-blocking
  - send( )/receive( ) with timeout

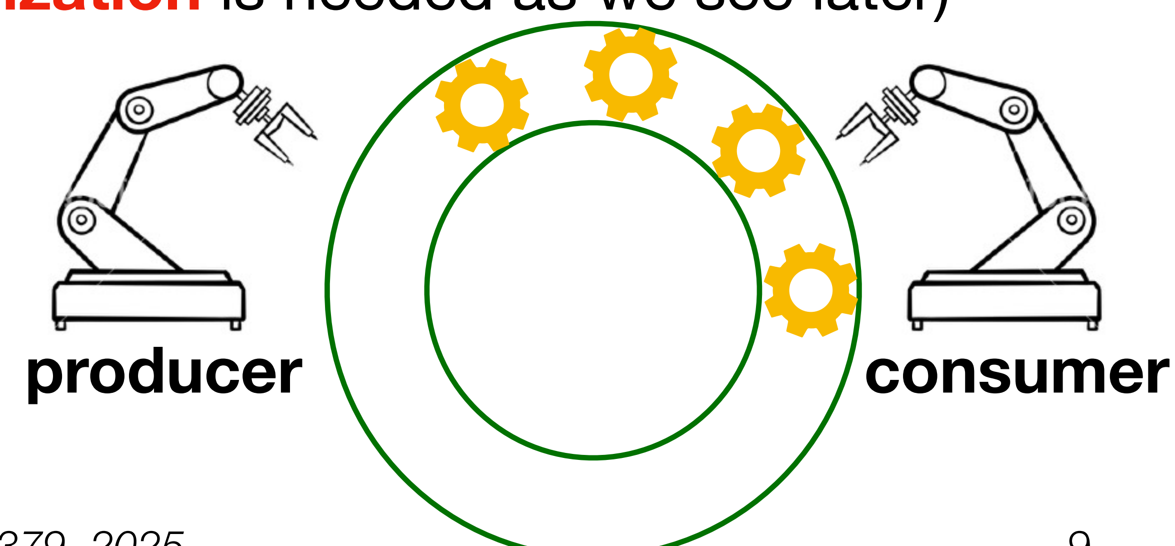
Communication using shared memory

- It is required to establish a mapping between the process's address space to a named memory object that may be shared across processes
- The `mmap( )` systems call does this
  - there is a backing file for the shared memory segment
- Can create the named memory object, then fork processes so that the forked processes know the name of this object
  - **what are the other solutions?**

Producer-Consumer problem

- Definition: producers puts data into a shared buffer; consumers takes it out
  - also known as the bounded-buffer problem
  - examples:
    - web servers (server:producer & client:consumer)
    - compiling your code w/ gcc: `cpp | cc1 | as | ld`
- For implementation, **interprocess communication** is necessary
- What should the producer do when the buffer is full?
- What should the consumer do when the buffer is empty?
- How to maintain data integrity? (**synchronization** is needed as we see later)





Producer-Consumer problem

- Implementation based on message passing (loose syntax)
  - using send(c_pid, nextp) and receive(p_pid, nextc)
    - each process must be able to name the other process
  - the kernel manages the buffer (message queue)

Main

```
int main( ) {
   …
   if (fork( ) != 0) producer( );
   else consumer( );
   …
}
```

Producer

```
int producer( ) {
  …
  while(true) {
     …
     nextp = produced item
     send(C_pid, nextp)
     …
  }
}
```

Consumer

```
int consumer( ) {
  …
  while(true) {
     …
     receive(P_pid, &nextc)
     consume nextc
     …
  }
}
```

Producer-Consumer problem

- Implementation based on shared memory (loose syntax)
  - `n` is the size of the buffer
  - `in` points to the next free location, `out` points to the first full location
  - `in` and `out` are shared between producer and consumer
  - this way we can have at most `n - 1` items in the buffer (**why?**)

Main

```
int main( ) {
  …
  mmap(..., PROT_WRITE, PROT_SHARED, …);
  // define in, out, buffer in the shared region
  if (fork( ) != 0) producer( );
  else consumer( );
  …
}
```

Producer

```
int producer( ) {
  …
  while(true) {
    …
    nextp = produce item
    while (in+1 mod n == out) { }
    buffer[in] = nextp
    in = in+1 mod n
  }
}
```

Consumer

```
int consumer( ) {
  …
  while(true) {
    …
    while (in == out) { }
    nextc = buffer[out]
    out = out+1 mod n
    consume nextc
  }
}
```

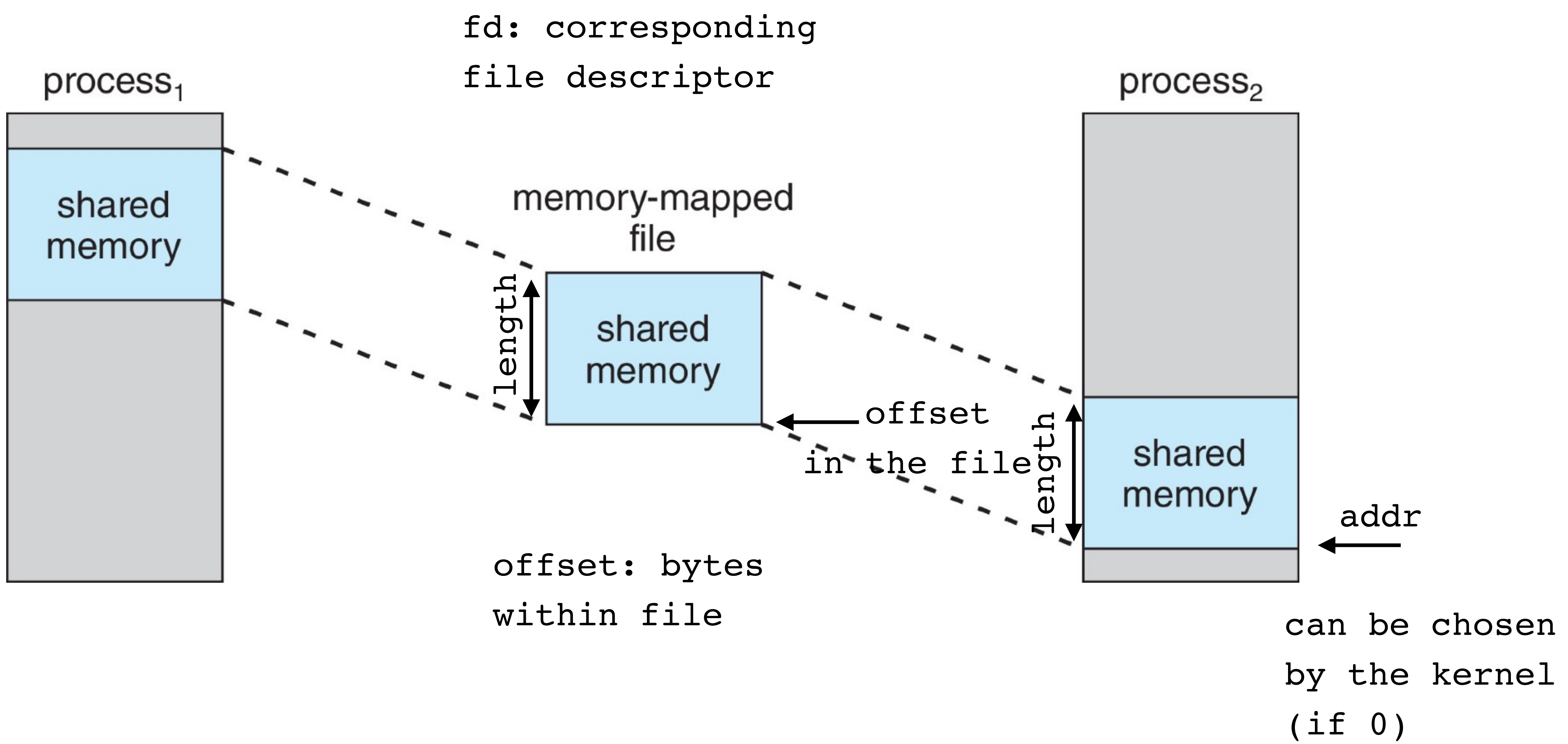
POSIX shared memory support

POSIX shared memory is organized using **memory-mapped files**, i.e., associating the region of shared memory with a file

- the `shm_open( )` and `ftruncate( )` system calls are used to create a shared memory object with a specified name and to set the size of this object, respectively
  - **shared memory object**: a handle which can be used by unrelated processes to memory map the same region of shared memory
  - `shm_open( )` creates/opens the shared memory object (i.e. the backing file with the given name) returning a file descriptor referring to the newly created shared memory object
- the `mmap( )` system call establishes a memory-mapped file containing this object and returns a pointer to this file;
  - the file pointer is used to write or read from this shared memory object
  - `munmap( )` unmaps the mapped region
- the `shm_unlink( )` system call removes the shared memory object
  - once all processes have unmapped the object, it de-allocates and destroys the contents of the associated memory region

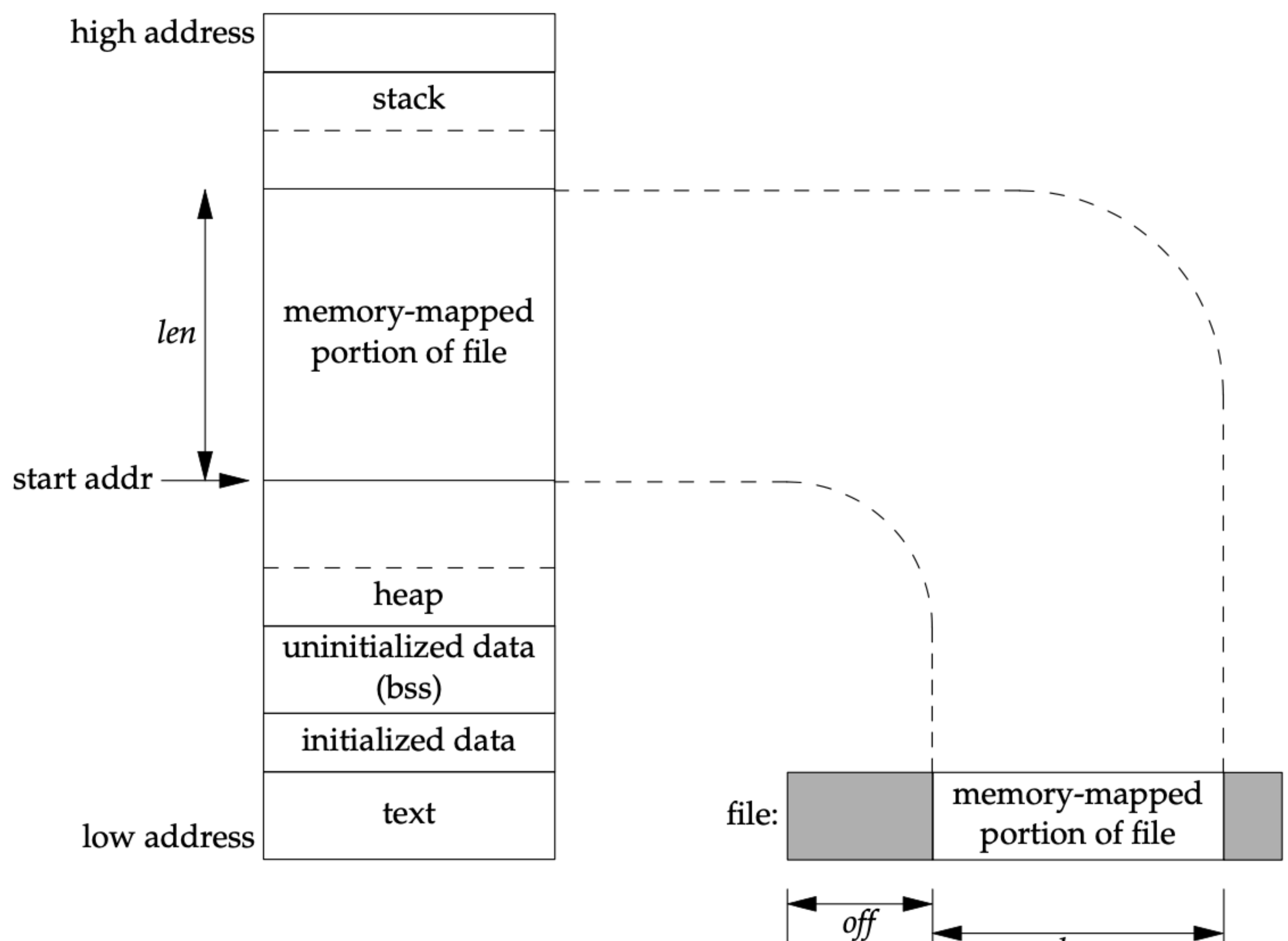
The mmap system call

```
void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset)
```



Attaching a shared region

```
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset)
```



POSIX producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main() {
  const int SIZE = 4096;                                    /* size (B) of shared memory object */
  const char* name = "/prog-shm";                           /* name of the shared memory object */

  const char* message_0 = "Hello";
  const char* message_1 = "World!";

  int shm_fd;                                               /* shared memory file descriptor */
  void* ptr;                                                /* pointer to shared memory object */

  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);          /* create the shared memory object */
  ftruncate(shm_fd, SIZE);                                  /* configure size of the shared memory object */
  ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);  /* memory map the shared memory object */
  // ptr is the starting address of the mapped area

  sprintf(ptr, "%s", message_0);                            /* write to the shared memory object */
  ptr += strlen(message_0);
  sprintf(ptr, "%s", message_1);
  ptr += strlen(message_1);

  return 0;
}
```

POSIX consumer

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

int main() {
  const int SIZE = 4096;                      /* size (B) of shared memory object */
  const char* name = "/prog-shm";             /* name of the shared memory object */

  int shm_fd;                                 /* shared memory file descriptor */
  void* ptr;                                  /* pointer to shared memory object */

  shm_fd = shm_open(name, O_RDONLY, 0666); /* open the shared memory object */

  /* memory map the shared memory object */
  ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
  // ptr is the starting address of the mapped area

  printf("%s", (char*)ptr);                   /* read from the shared memory object */
  shm_unlink(name);                           /* remove the shared memory object */

  return 0;
}
```

Message passing support

- `ftok( )` generates a unique key (of type `key_t`) from a pathname
  - useful when there's no other way for the cooperating processes to share the identifier
- `msgget( )` returns the message queue identifier associated with the unique key
  - if the queue does not currently exist, it creates a new queue and returns its identifier. Otherwise, it returns the identifier of an existing queue
- `msgsnd( )` appends a message to a queue given its identifier
- `msgrcv( )` retrieves a message from a queue given its identifier
- `msgctl( )` performs various operations on a queue
  - can be used to destroy a message queue, change max number of bytes allowed in the queue, etc.

Example — writer.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mymesg {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    key = ftok("progfile", 65);             // generates a unique key
    msgid = msgget(key, 0666 | IPC_CREAT); // creates a message queue
    message.mesg_type = 1;

    printf("Write Data: ");
    scanf("%s", message.mesg_text);

    msgsnd(msgid, &message, sizeof(message), 0); // sends the message

    printf("Data sent is: %s \n", message.mesg_text);

    return 0;
}
```

Example — reader.c

```
#include <stdio.h>
#include <sys/ipc.h>
#include <sys/msg.h>

// structure for message queue
struct mymesg {
    long mesg_type;
    char mesg_text[100];
} message;

int main()
{
    key_t key;
    int msgid;

    key = ftok("progfile", 65);                        // generates a unique key

    msgid = msgget(key, 0666 | IPC_CREAT);             // creates a message queue

    msgrcv(msgid, &message, sizeof(message), 1, 0); // receives the message

    printf("Data Received is : %s \n", message.mesg_text);

    msgctl(msgid, IPC_RMID, NULL);                     // destroys the message queue

    return 0;
}
```