Operating System Concepts

Lecture 8: Interprocess Communication

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

MWF 12:00-12:50 VVC 2 215

Today's class

- Interprocess communication
  - Shared memory
  - Message Passing

Cooperating processes

- any two processes are either independent or cooperating
  - a process is cooperating if it can affect or be affected by other processes

Cooperating processes

- any two processes are either independent or cooperating
  - a process is cooperating if it can affect or be affected by other processes
- cooperating processes work with each other to accomplish a single task
  - improve performance
    - parallel execution results in computation speedup on multicore systems
  - improve program structure/modularity
    - each cooperating process is smaller than a single monolithic program

Cooperating processes

- any two processes are either independent or cooperating
  - a process is cooperating if it can affect or be affected by other processes
- cooperating processes work with each other to accomplish a single task
  - improve performance
    - parallel execution results in computation speedup on multicore systems
  - improve program structure/modularity
    - each cooperating process is smaller than a single monolithic program
- cooperating processes need to share information
  - OS makes this happen using interprocess communication mechanisms

What are IPC mechanisms in UNIX?

IPC type	SUS	FreeBSD 8.0	Linux 3.2.0	Mac OS X 10.6.8	Solaris 10
half-duplex pipes	•	(full)	•	•	(full)
FIFOs	•	•	•	•	•
full-duplex pipes	allowed	•, UDS	UDS	UDS	•, UDS
named full-duplex pipes	obsolescent	UDS	UDS	UDS	•, UDS
XSI message queues	XSI	•	•	•	•
XSI semaphores	XSI	•	•	•	•
XSI shared memory	XSI	•	•	•	•
message queues (real-time)	MSG option	•	•		•
semaphores	•	•	•	•	•
shared memory (real-time)	SHM option	•	•	•	•
sockets	•	•	•	•	•
STREAMS	obsolescent				•

Two fundamental approaches

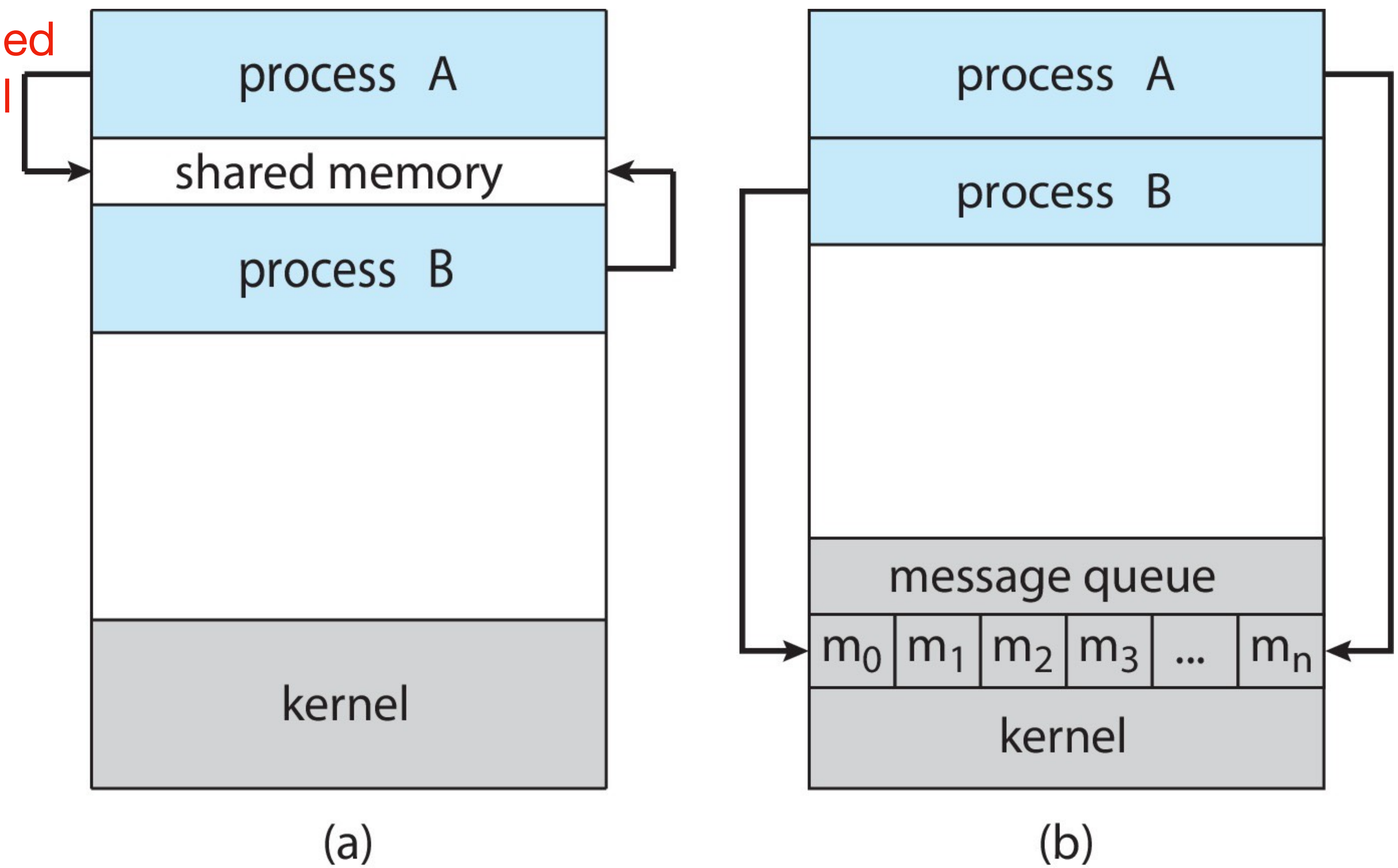
- Message passing
  - kernel intervention (through system calls) is required in every send/receive operation
  - extensible to communication in distributed systems
  - **positive aspects:** all sharing is explicit; less chance for error
  - **negative aspects:** high overhead. data copying, cross protection domains

Two fundamental approaches

- Message passing
  - kernel intervention (through system calls) is required in every send/receive operation
  - extensible to communication in distributed systems
  - **positive aspects:** all sharing is explicit; less chance for error
  - **negative aspects:** high overhead. data copying, cross protection domains
- Shared memory: processes (or threads) can read and write a set of shared memory locations
  - system calls are only required to establish shared memory regions
  - processes are responsible for ensuring that they do not access a location concurrently (synchronization is explicit)
  - difficult to provide across machine boundaries
  - **positive aspects:** faster: set up shared memory once, then access w/o crossing protection domains
  - **negative aspects:** things might change behind your back; error prone

Two fundamental approaches

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



Communication using message passing

- distributed systems typically communicate using message passing
  - each process needs to be able to name the other process or name the mailbox/message queue (POSIX implementation)

Communication using message passing

- distributed systems typically communicate using message passing
  - each process needs to be able to name the other process or name the mailbox/message queue (POSIX implementation)
- a common system message queue is
  - a linked list of messages stored within the kernel and identified by a message queue identifier
  - this identifier is shared between the processes to access to the queue (**HOW?**)

Communication using message passing

- distributed systems typically communicate using message passing
  - each process needs to be able to name the other process or name the mailbox/message queue (POSIX implementation)
- a common system message queue is
  - a linked list of messages stored within the kernel and identified by a message queue identifier
  - this identifier is shared between the processes to access to the queue (**HOW?**)
- OS keeps track of 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)

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: message boundaries
  - byte stream model: no boundaries

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: message boundaries
  - byte stream model: no boundaries
- Connection model
  - connection-oriented model: recipient is specified at connection time, hence it does not need be specified for individual send operations
  - connectionless models: recipient is specified as a parameter to each send operation

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: message boundaries
  - byte stream model: no boundaries
- Connection model
  - connection-oriented model: recipient is specified at connection time, hence it does not need 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

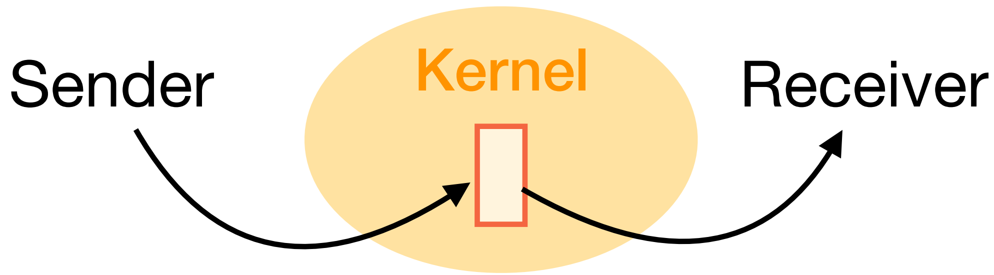
Message passing issues

Does a send/receive operation block?

Message passing issues

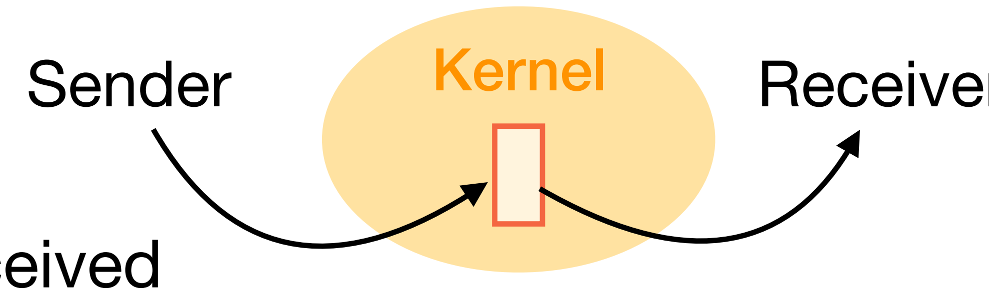
Does a send/receive operation block?

- blocking operations



Message passing issues

Does a send/receive operation block?

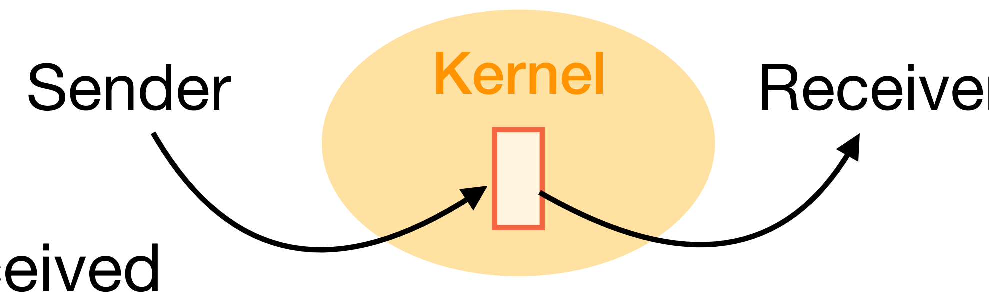


- blocking operations
  - sender has to wait until its message is received
  - receiver has to wait if no message is available

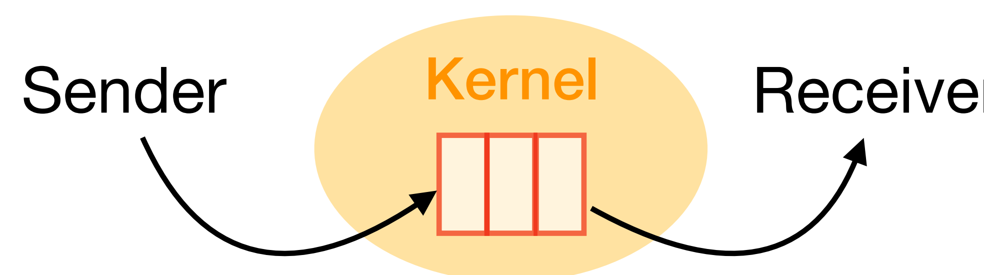
Message passing issues

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



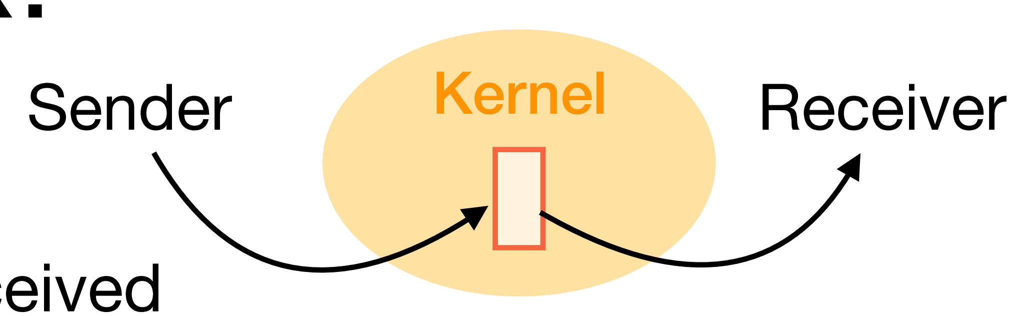
- non-blocking operations



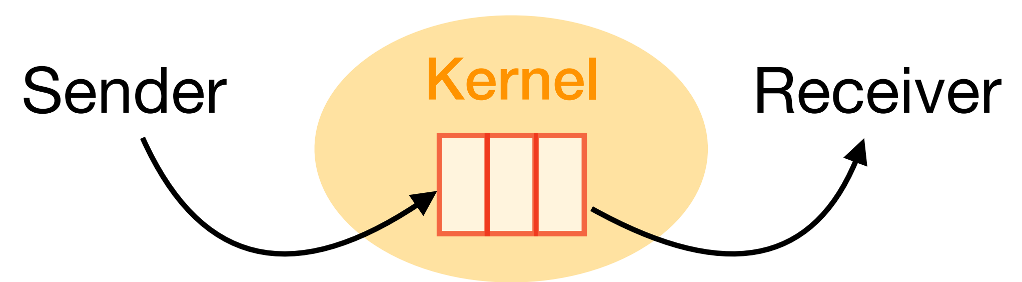
Message passing issues

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



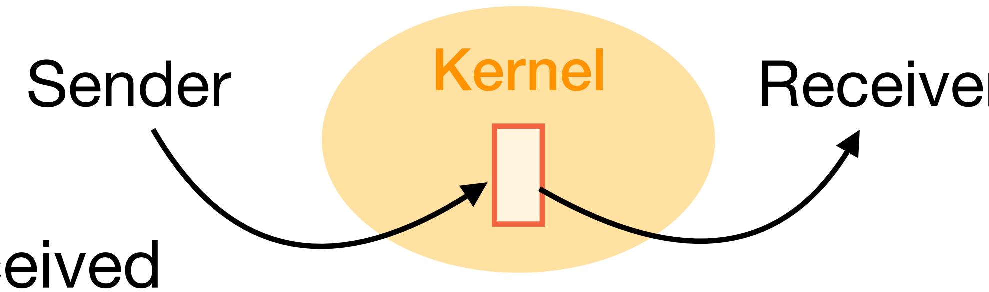
- non-blocking operations
  - send operation returns immediately
  - receive operation returns if no message is available



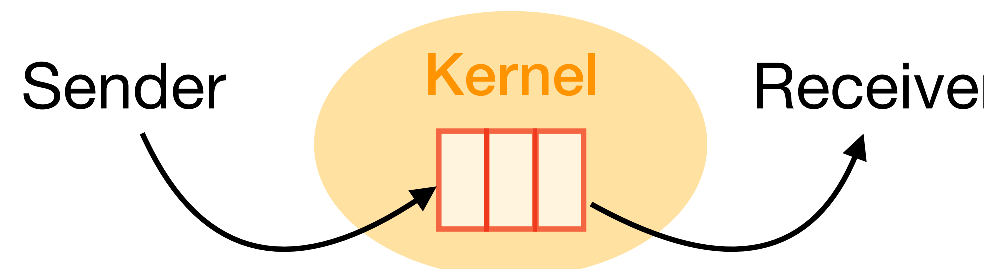
Message passing issues

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

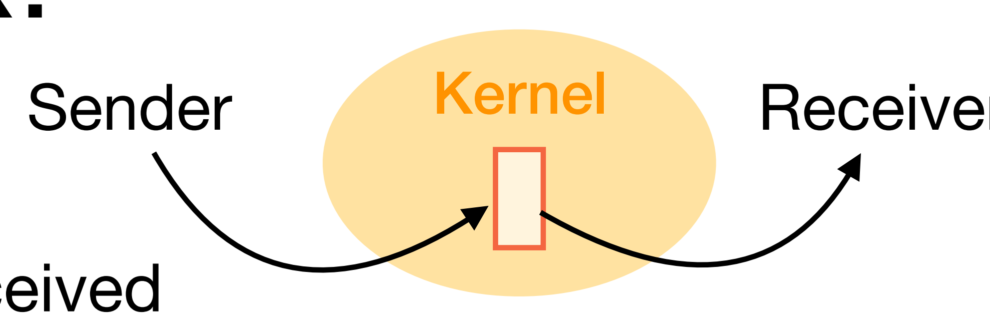


- partially blocking/non-blocking

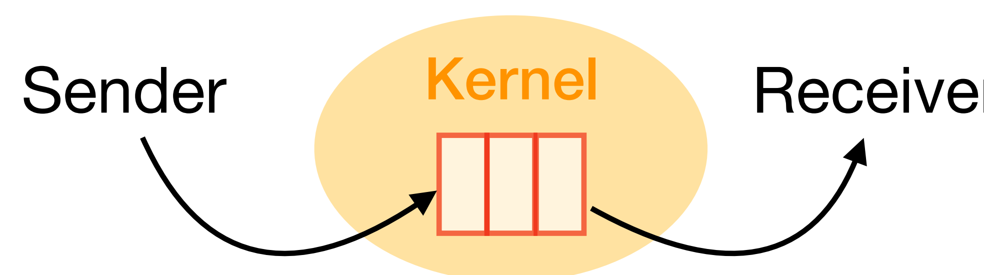
Message passing issues

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

- need to establish a mapping between the process's address space to a named memory object that may be shared across processes

Communication using shared memory

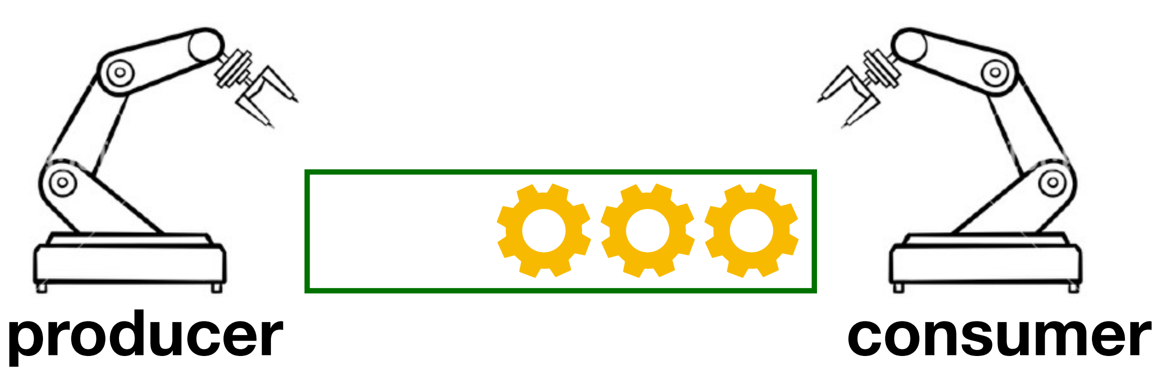
- need to establish a mapping between the process's address space to a named memory object that may be shared across processes
- the `mmap( )` systems call performs this function

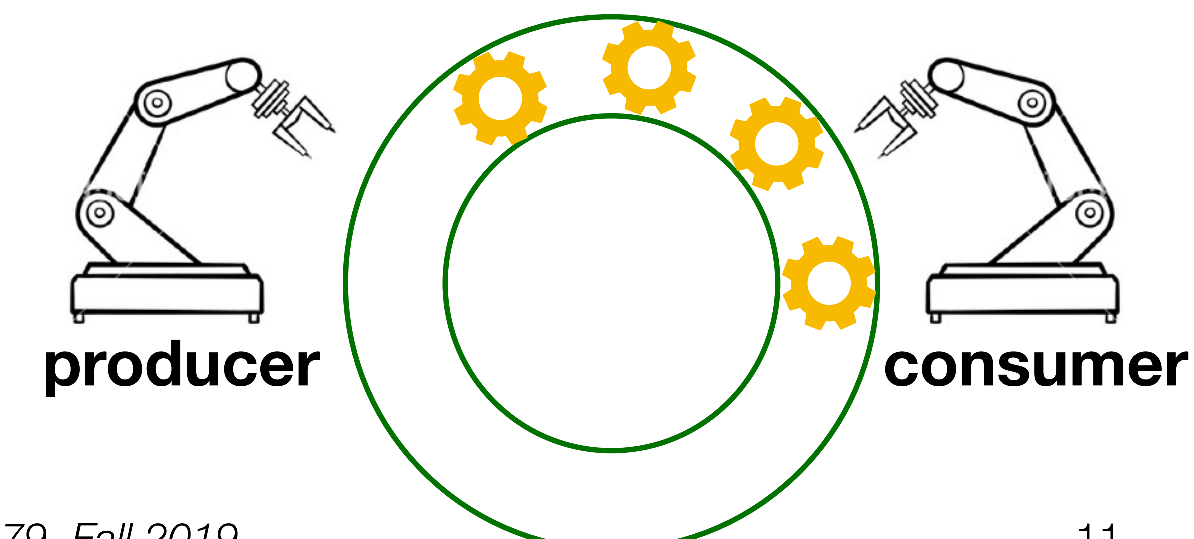
Communication using shared memory

- need 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 performs this function
- can fork processes that need to share the data structure so that they know the name of the shared object
    - **any other solution?**

Producer-Consumer problem

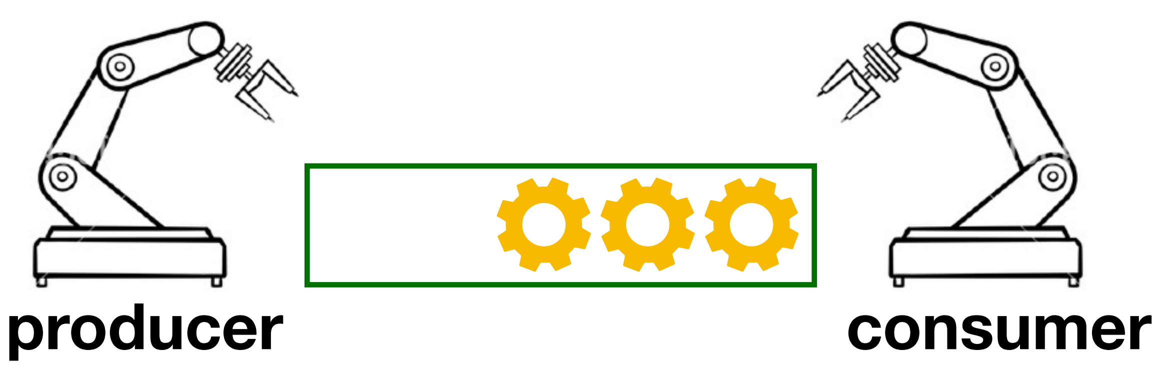
- Definition: producers puts information into a shared buffer; consumers takes it out

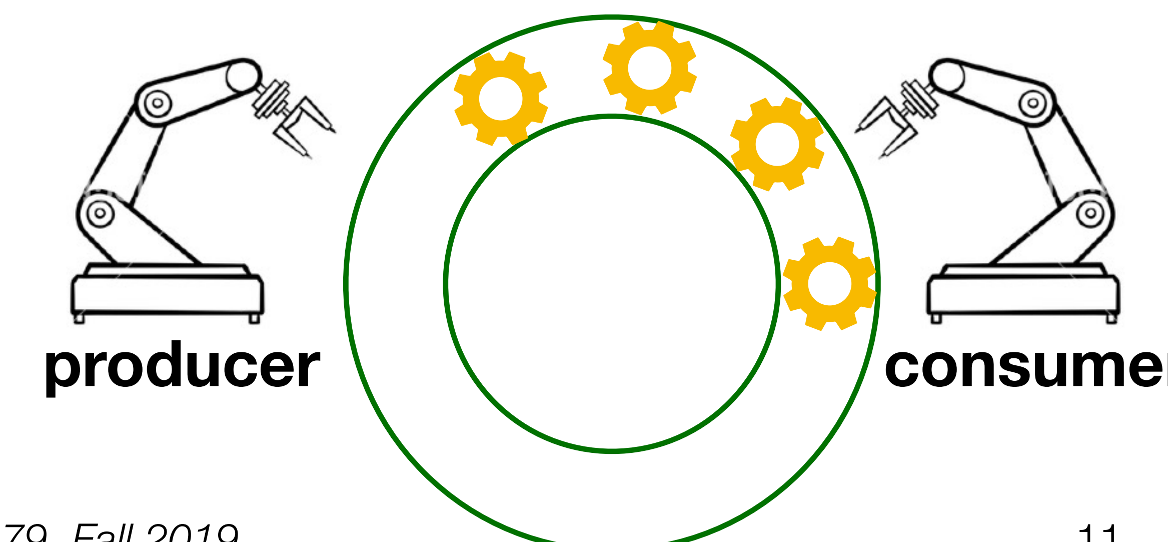




Producer-Consumer problem

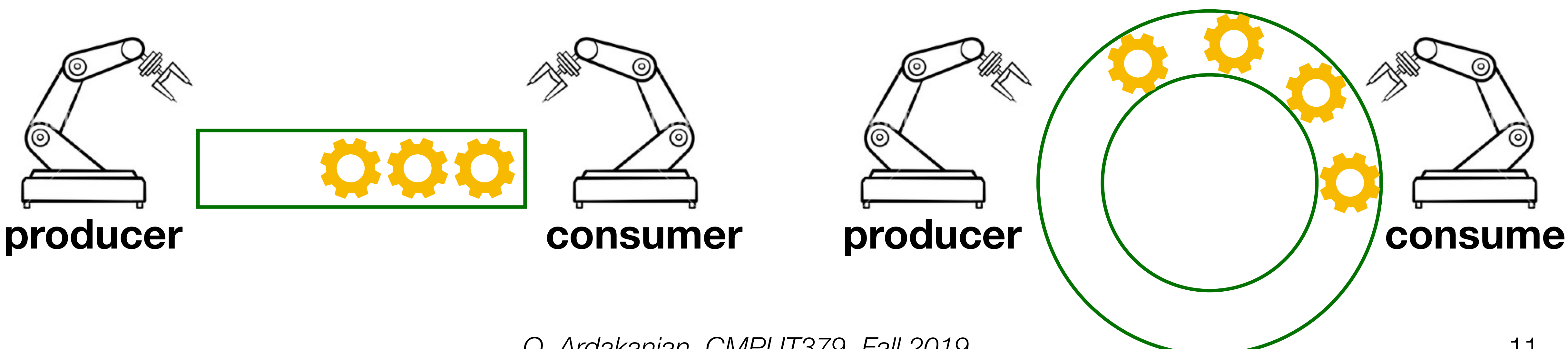
- Definition: producers puts information into a shared buffer; consumers takes it out
- it is also known as the bounded-buffer problem
  - examples:
    - web servers (server:producer & client:consumer)
    - compiling your code w/ gcc: `cpp | cc1 | as | ld`





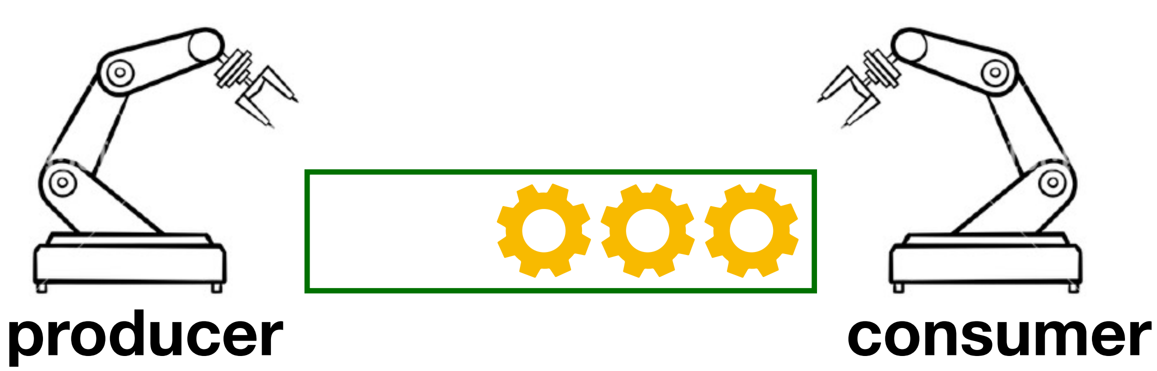
Producer-Consumer problem

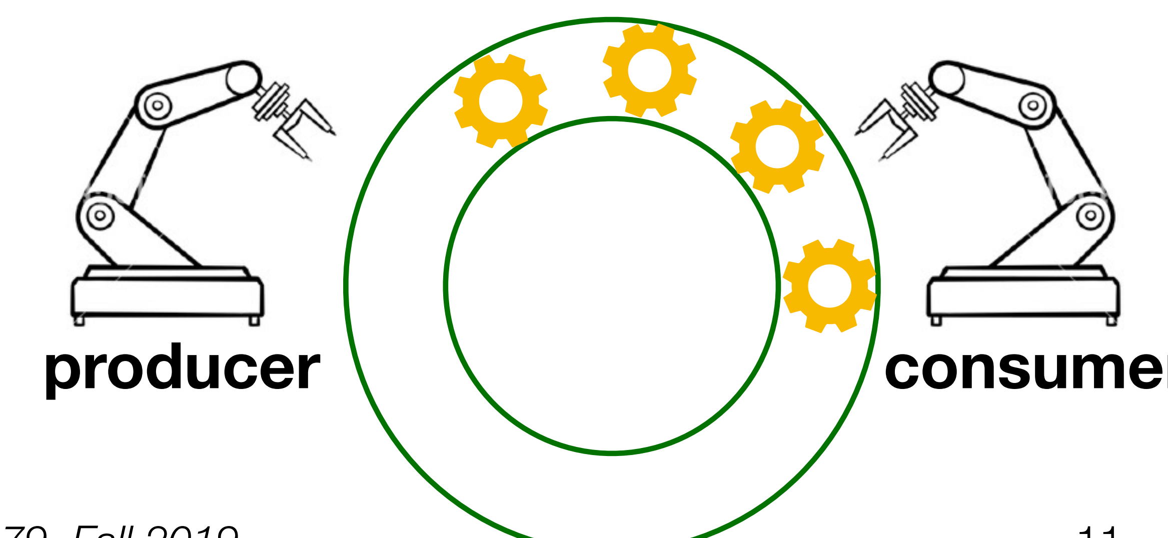
- Definition: producers puts information into a shared buffer; consumers takes it out
- it is also known as the bounded-buffer problem
  - examples:
    - web servers (server:producer & client:consumer)
    - compiling your code w/ gcc: `cpp | cc1 | as | ld`
- For implementation, **interprocess communication** is necessary



Producer-Consumer problem

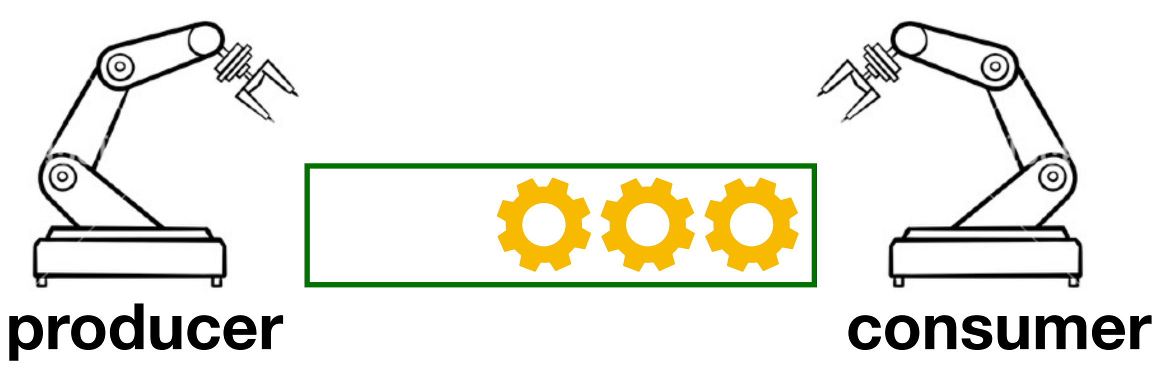
- <u>Definition</u>: producers puts information into a shared buffer; consumers takes it out
- it is also known as the bounded-buffer problem
  - examples:
    - web servers (server:producer & client:consumer)
    - compiling your code w/ gcc: `cpp | cc1 | as | ld`
- For implementation, **interprocess communication** is necessary
- What should the producer do when the buffer is full?

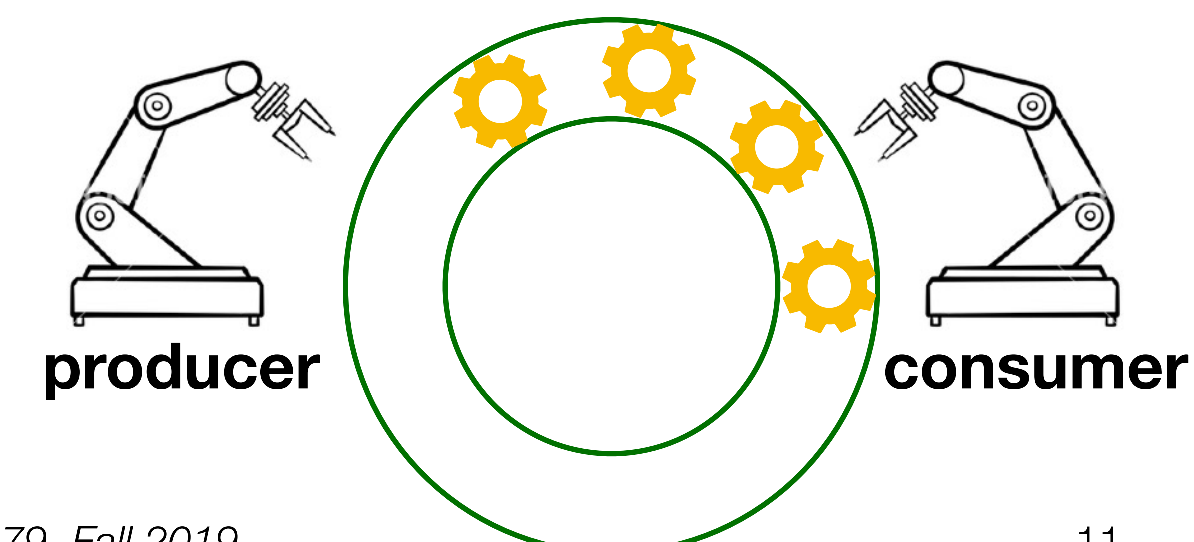




Producer-Consumer problem

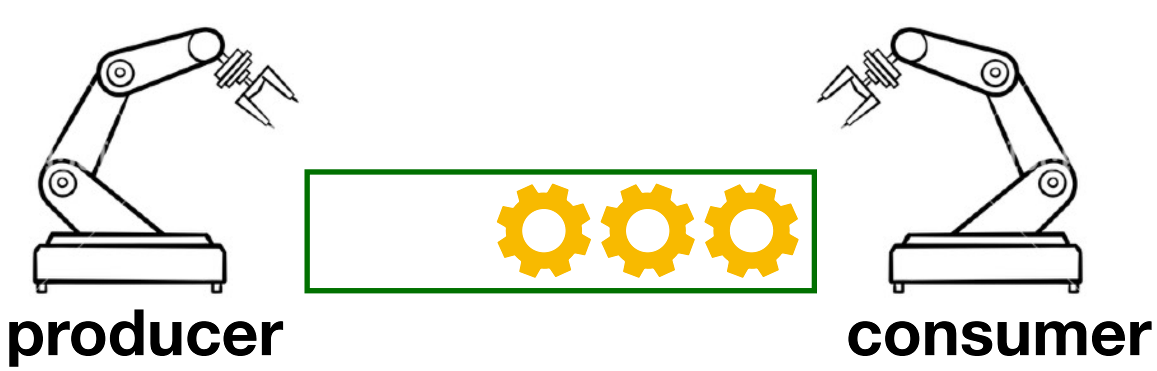
- Definition: producers puts information into a shared buffer; consumers takes it out
- it is also known as the bounded-buffer problem
  - examples:
    - web servers (server:producer & client:consumer)
    - compiling your code w/ gcc: `cpp | cc1 | as | ld`
- For implementation, **interprocess communication**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

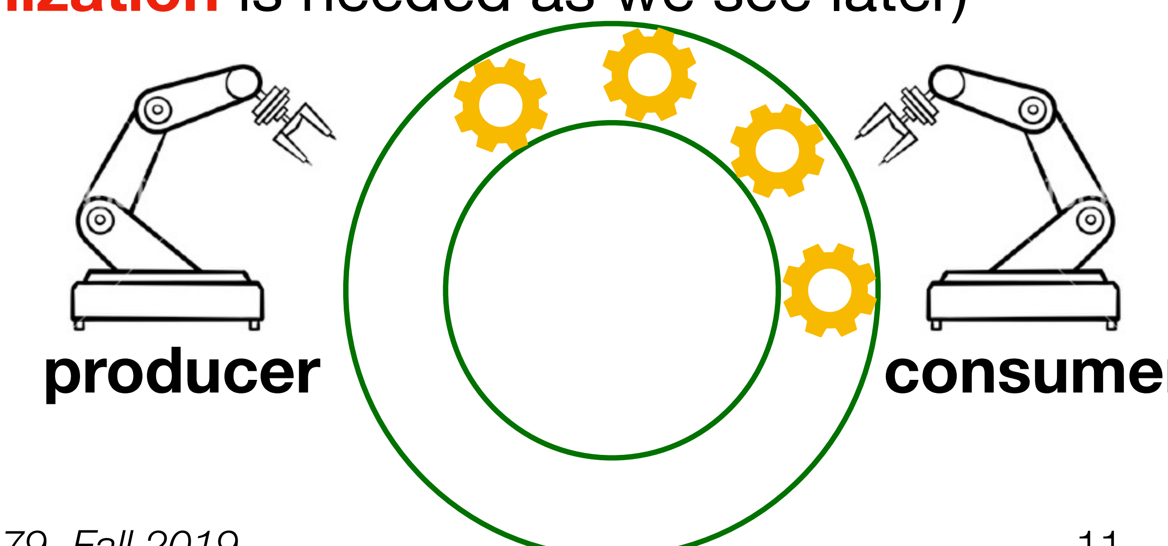




Producer-Consumer problem

- Definition: producers puts information into a shared buffer; consumers takes it out
- it is 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
  - using send(c_pid, nextp) and receive(p_pid, nextc)
  - each process needs to be able to name the other process
  - the kernel manages the buffer (message queue)

Producer-Consumer problem

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

Main

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

Producer-Consumer problem

- implementation based on message passing
  - using send(c_pid, nextp) and receive(p_pid, nextc)
  - each process needs to 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)
     …
  }
}
```

Producer-Consumer problem

- implementation based on message passing
  - using send(c_pid, nextp) and receive(p_pid, nextc)
  - each process needs to 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
  - `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?**)

Producer-Consumer problem

- implementation based on shared memory
  - `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(..., in, out, PROT_WRITE, PROT_SHARED,
    …);
    in = 0
    out = 0
    if (fork( ) != 0) producer( );
    else consumer( );
    …
}
```

Producer-Consumer problem

- implementation based on shared memory
  - `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(..., in, out, PROT_WRITE, PROT_SHARED,
    ...);
    in = 0
    out = 0
    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
  }
}
```

Producer-Consumer problem

- implementation based on shared memory
  - `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(..., in, out, PROT_WRITE, PROT_SHARED,
    ...);
    in = 0
    out = 0
    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 support

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

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

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

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( )` returns a new file descriptor referring to the newly created shared memory object

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( )` returns a new 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;

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( )` returns a new 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 to or read from this shared memory object

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( )` returns a new 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 to or read from this shared memory object
- the `shm_unlink( )` system call removes the shared memory object

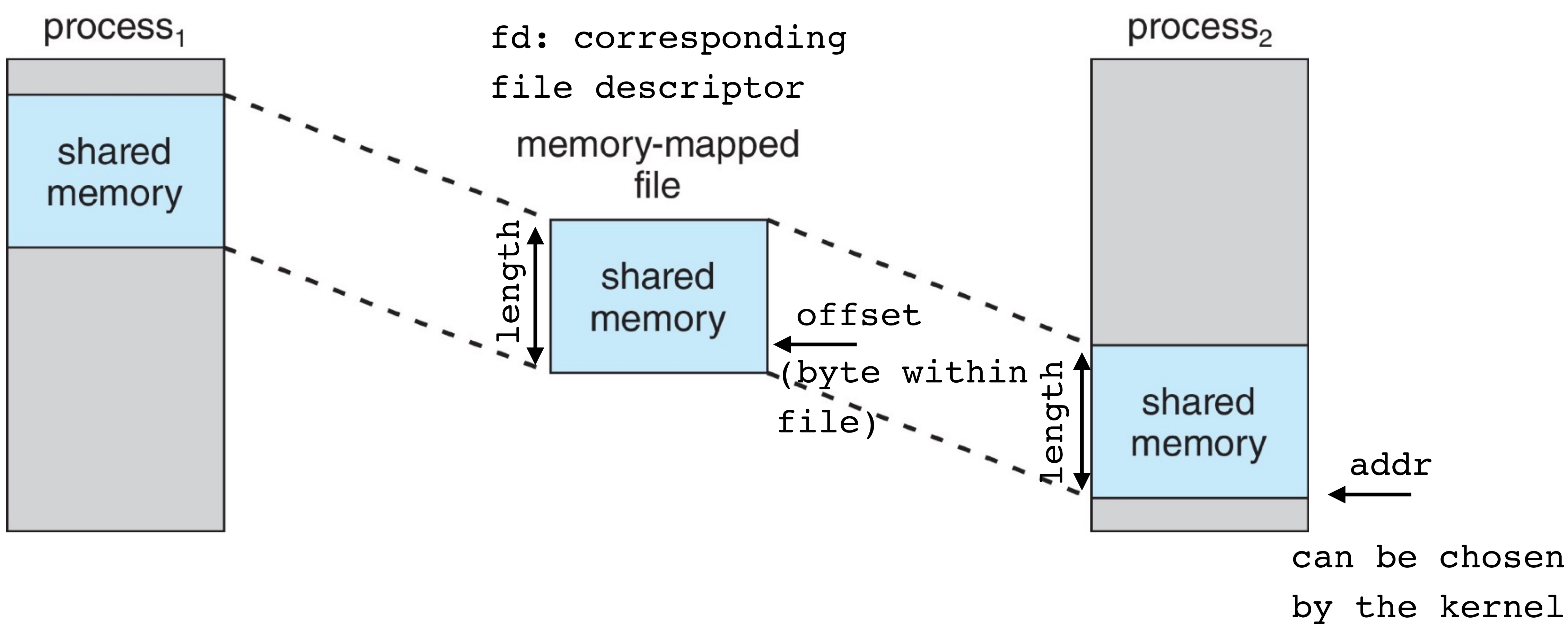
POSIX shared memory support

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( )` returns a new 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 to or read from this shared memory object
- 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)
```



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 */

  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);

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

  return 0;
}
```

System V message passing support

- `ftok( )` generates a unique key

System V message passing support

- `ftok( )` generates a unique key
- `msgget( )` either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value

System V message passing support

- `ftok( )` generates a unique key
- `msgget( )` either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value
- `msgsnd( )` places data onto a message queue

System V message passing support

- `ftok( )` generates a unique key
- `msgget( )` either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value
- `msgsnd( )` places data onto a message queue
- `msgrcv( )` retrieves messages from a queue

System V message passing support

- `ftok( )` generates a unique key
- `msgget( )` either returns the message queue identifier for a newly created message queue or returns the identifiers for a queue which exists with the same key value
- `msgsnd( )` places data onto a message queue
- `msgrcv( )` retrieves messages from a queue
- `msgctl( )` performs various operations on a queue.
  - is generally used to destroy message queue

Example — writer.c

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

// structure for message queue
struct mesg_buffer {
    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 mesg_buffer {
    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;
}
```