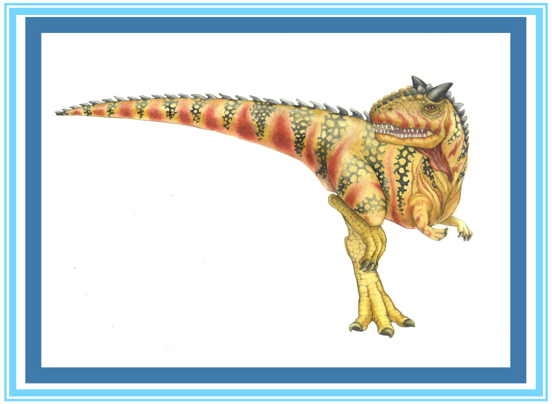
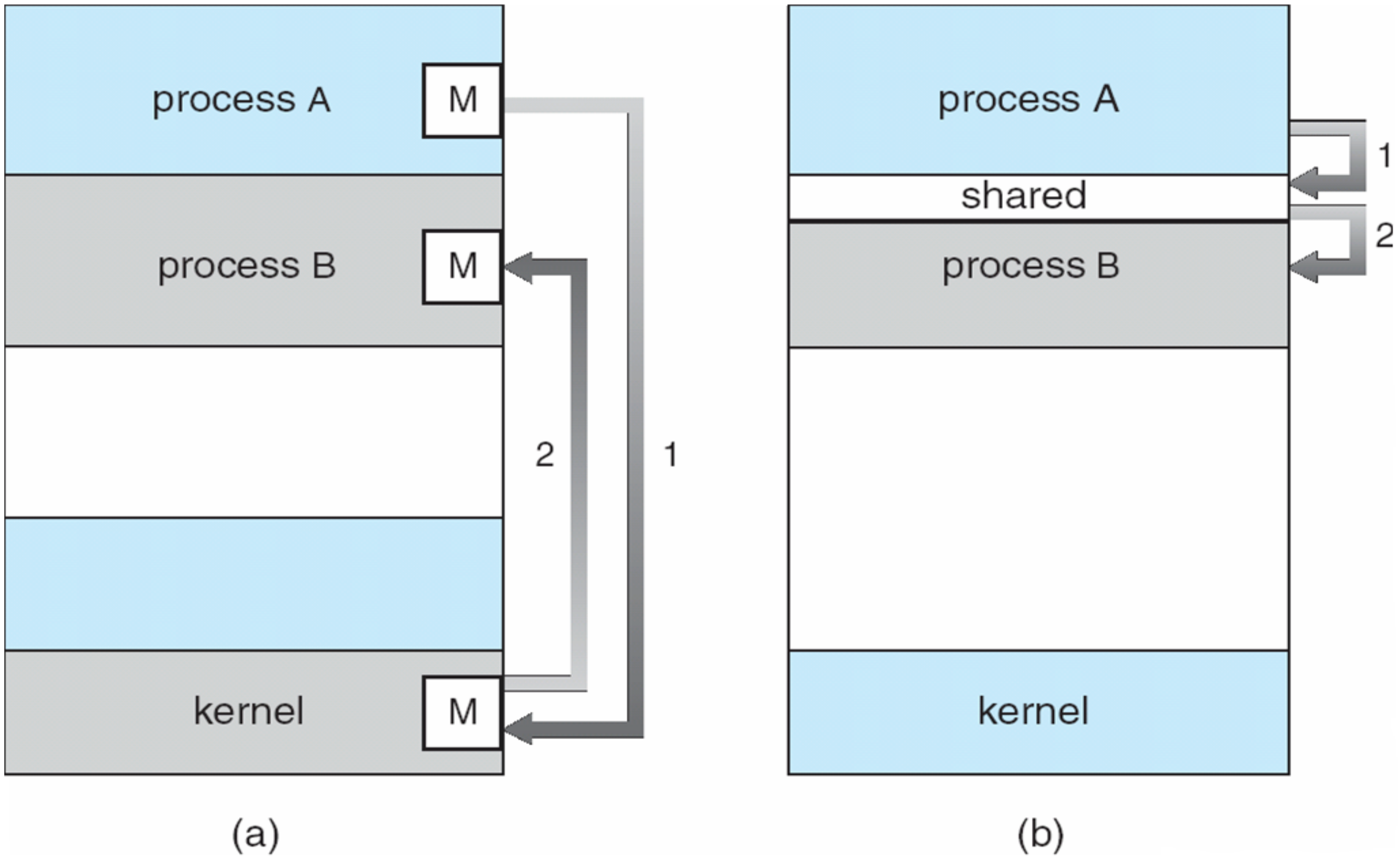
Chapter 3: Processes-IPC



Interprocess Communication

- Processes within a system may be **independent** or **cooperating**
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models



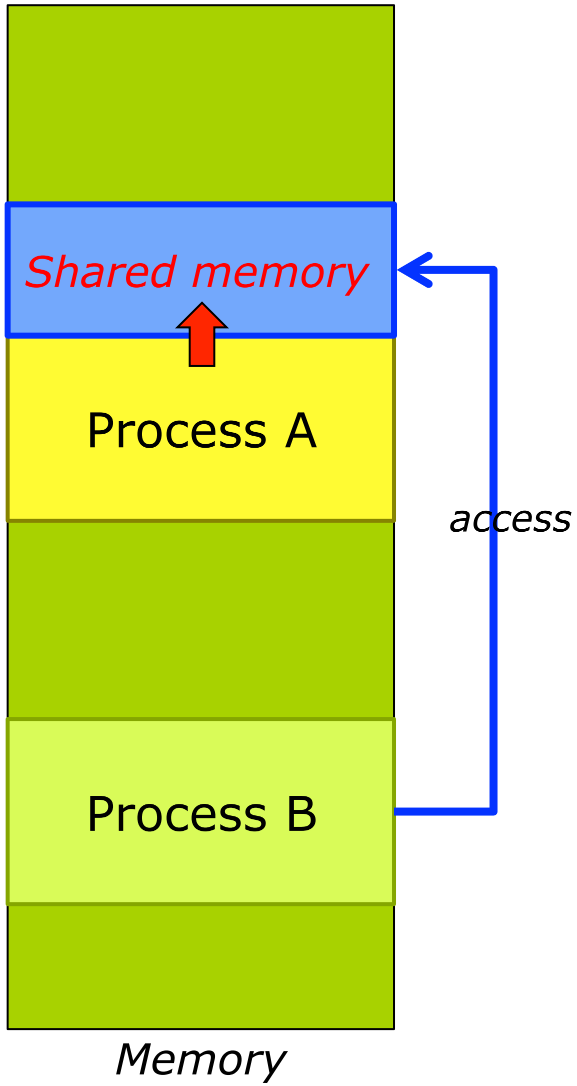
Shared Memory & Message Passing

	Message Passing	Shared Memory
Implementation		
Speed		
Kernel interventi on		
Data size		

Shared Memory & Message Passing

	Message Passing	Shared Memory
Implementation	Easier	Difficult
Speed	Slower	Faster
Kernel intervention	A lot, via system calls	No system calls except set up
Data size	Good for small amount	Good for large amount

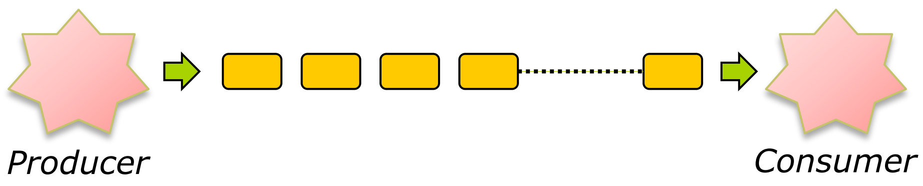
Shared Memory Systems



- Process-A creates a shared memory
  - Shared memory in Process-A's address space
- Allow Process B to access the shared memory
- No predefined data format

Producer-Consumer Model

- Producer-Consumer Model
  - Producer only produces (writes) information and Consumer only consumes (reads) the information

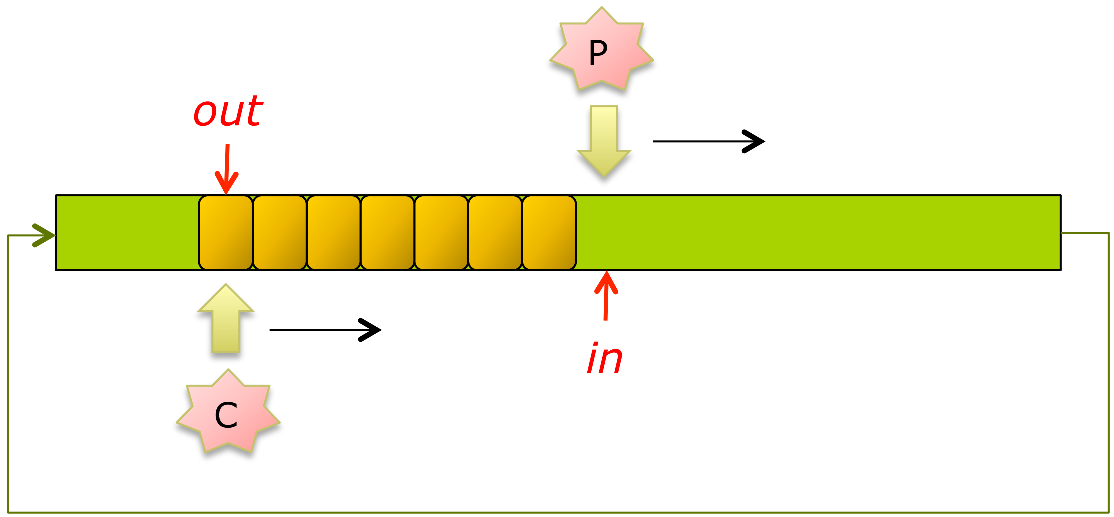


  - Use *Buffer* to deliver information from producer to consumer

Shared Buffer Model

- Unbounded Buffer
  - There is no limit in the buffer size
  - Producer can always create data
  - Consumer cannot consume data if the buffer is empty
- Bounded Buffer
  - There is a limit in the buffer size
  - *Producer cannot create data if the buffer is full*
  - *Consumer cannot consume data if the buffer is empty*
- In practice, we have only bounded buffer

Shared Buffer by Circular Array




```
#define BS 100
typedef struct {…} item;

item buf[BS]
int in = 0
int out = 0
```

* Buffer is empty if
  i == j
* Buffer is full if
  (in+1)%BS == out
* Maximum items count
  BS-1

Bounded-Buffer – Producer

```
while (true) {
   /* Produce an item */
      while (((in = (in + 1) % BUFFER SIZE
count)  == out)
     ;   /* do nothing -- no free buffers */
    buffer[in] = item;
    in = (in + 1) % BUFFER SIZE;
   }
```

Bounded Buffer – Consumer

```
while (true) {
        while (in == out)
               ; // do nothing --
nothing to consume

     // remove an item from the buffer
     item = buffer[out];
     out = (out + 1) % BUFFER SIZE;
return item;
   }
```

Message Passing Systems

- IPC provides two operations:
  - **send**(*message*) – message size fixed or variable
  - **receive**(*message*)
- If *P* and *Q* wish to communicate,
  - establish a *communication link* between them
  - exchange messages via send/receive
- Methods
  - Direct / Indirect Communication
  - Synchronous / Asynchronous Communication

Direct Communication

- Processes must name each other explicitly:
  - **send** (*P, message*) – send a message to process P
  - **receive**(*Q, message*) – receive a message from process Q

- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - The link may be unidirectional, but is usually bi-directional

Indirect Communication

- Messages are directed and received from mailboxes (also referred to as ports)
  - Each mailbox has a unique id
  - Processes can communicate only if they share a mailbox
- Properties of communication link
  - Link established only if processes share a common mailbox
  - A link may be associated with many processes
  - Each pair of processes may share several communication links
  - Link may be unidirectional or bi-directional

Indirect Communication

- Operations
  - create a new mailbox
  - send and receive messages through mailbox
  - destroy a mailbox

- Primitives are defined as:

  send(*A, message*) – send a message to mailbox A

  receive(*A, message*) – receive a message from mailbox A

Synchronization

- Message passing may be either blocking or non-blocking
- **Blocking** is considered **synchronous**
  - **Blocking send** has the sender block until the message is received
  - **Blocking receive** has the receiver block until a message is available
- **Non-blocking** is considered **asynchronous**
  - **Non-blocking** send has the sender send the message and continue
  - **Non-blocking** receive has the receiver receive a valid message or null

Buffering

- Queue of messages attached to the link; implemented in one of three ways
  1. Zero capacity – 0 messages
     Sender must wait for receiver (rendezvous)
  2. Bounded capacity – finite length of *n* messages
     Sender must wait if link full
  3. Unbounded capacity – infinite length
     Sender never waits

Pipes

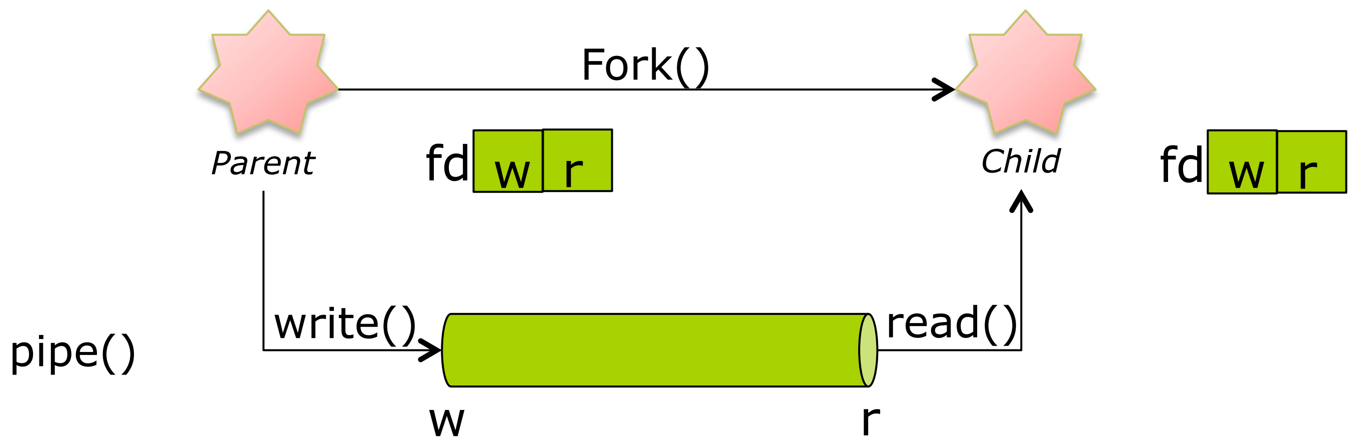
- Acts as a conduit allowing two processes to communicate
- **Issues**
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex?
  - Must there exist a relationship (i.e. parent-child) between the communicating processes?
  - Can the pipes be used over a network?

Ordinary Pipes

- **Ordinary Pipes** allow communication in standard producer-consumer style
- Producer writes to one end (the *write-end* of the pipe)
- Consumer reads from the other end (the *read-end* of the pipe)
- Ordinary pipes are therefore unidirectional
- Only between parent and child processes

Ordinary Pipes: Example

- Parent process wants to send a message "Greetings" to a child process
- When creating a pipe, it returns two file descriptors
  - One for writing, one for reading
- Parent process writes to the writing file descriptor
- Child process reads from the reading file descriptor



Ordinary Pipes: Code in Unix

```
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1

int main(void)
{
char write_msg[BUFFER_SIZE] = "Greetings";
char read_msg[BUFFER_SIZE];
int fd[2];
pid_t pid;

 /* create the pipe */
 if (pipe(fd) == -1) {
    fprintf(stderr,"Pipe failed");
    return 1;
 }

 /* fork a child process */
 pid = fork();

 if (pid < 0) { /* error occurred */
    fprintf(stderr, "Fork Failed");
    return 1;
 }

if (pid > 0) { /* parent process */
   /* close the unused end of the pipe */
   close(fd[READ_END]);

   /* write to the pipe */
   write(fd[WRITE_END], write_msg, strlen(write_msg)+1);

   /* close the write end of the pipe */
   close(fd[WRITE_END]);
}
else { /* child process */
   /* close the unused end of the pipe */
   close(fd[WRITE_END]);

   /* read from the pipe */
   read(fd[READ_END], read_msg, BUFFER_SIZE);
   printf("read %s",read_msg);

   /* close the write end of the pipe */
   close(fd[READ_END]);
}

return 0;
```

Named Pipes

- Ordinary pipe disappears when the process terminates

- Named Pipes are more powerful than ordinary pipes
  - Communication is bidirectional
  - No parent-child relationship is necessary
  - Several processes can use it (ex: many writers)
  - Continue to exist after a process terminates
  - Provided on both UNIX and Windows systems

Named Pipes

- Unix
  - Called FIFO
  - Once created (mkfifo()), appear as a file (use open(), read(), write(), close())
  - Exists until deleted from the file system
  - Bidirectional, half-duplex
  - Only within a system
- Windows
  - Bidirectional, full-duplex
  - Within or between systems
  - CreateNamedPipe(), ConnectNamedPipe(), ReadFile(), WriteFile()
- ls | more, dir | more

End of Chapter 3