CISC 3320
C09a: Inter-Process Communication

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Acknowledgement

- This slides are a revision of the slides by the authors of the textbook

Outline

- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems

- Examples of IPC Systems
- Communication in Client-Server Systems

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***

Cooperating Processes

- ***Independent*** process cannot affect or be affected by the execution of another process
- ***Cooperating*** process can affect or be affected by the execution of another process
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity

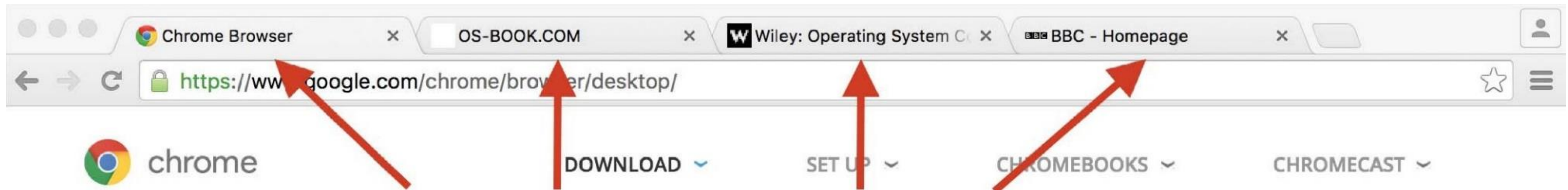
Multiprocess Architecture: Example Applications

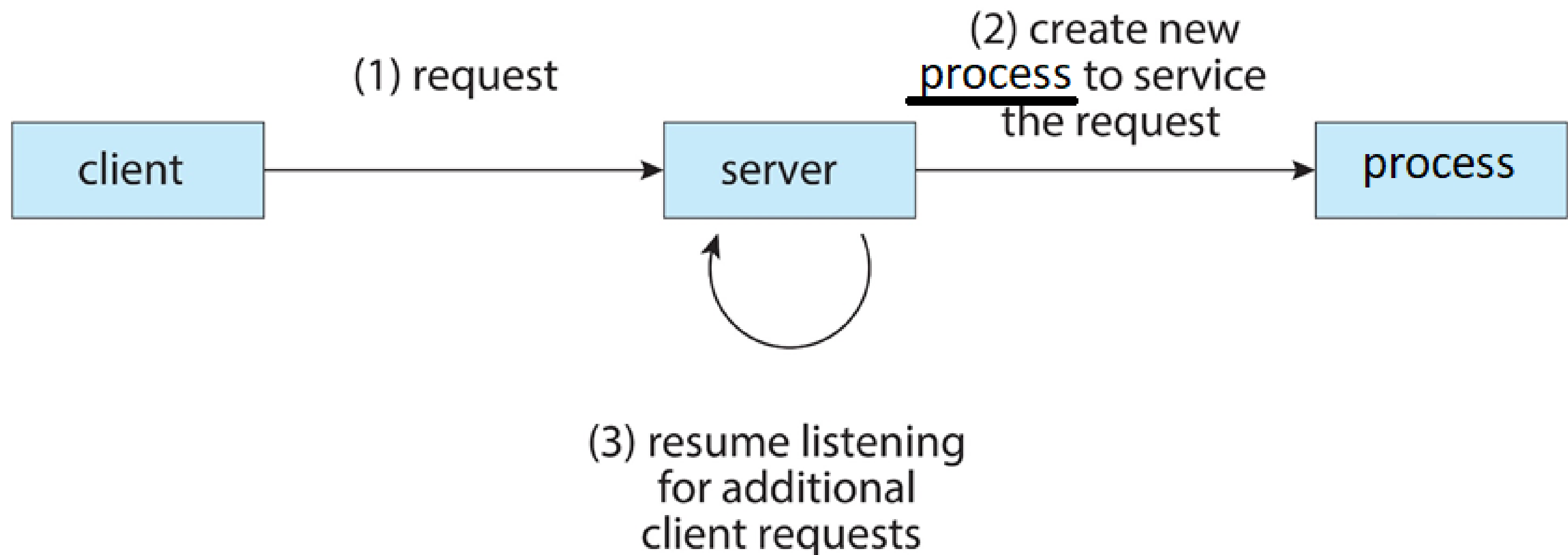
- Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
- Chrome Web browser
- The instructor's Monte Carlo simulation program to estimate π

Multiprocess Architecture in Chrome Browser

- Some Web browsers ran as single process
  - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
  - **Browser** process manages user interface, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened
    - Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in

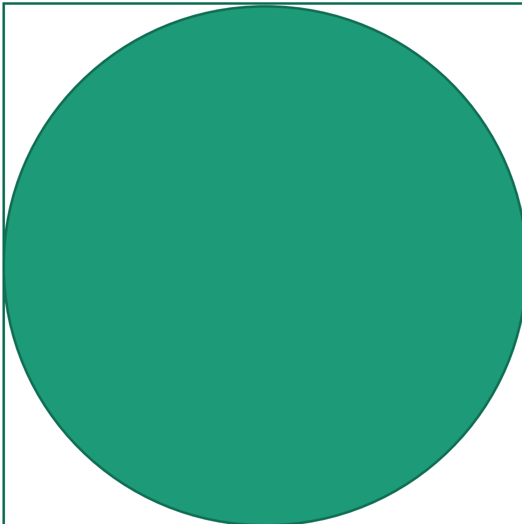
New Renderer Created for Each Website Opened





Multiprocess: Computation Speed-up by Example

- Estimate π using a Monte Carlo simulation
    - What if a machine has multiple CPU cores? Can we take advantage of it?



Multiprocess: Computation Speed-up by Example

- Estimate π using a Monte Carlo simulation
  - What if a machine has multiple CPU cores? Can we take advantage of it?
    - *Case 1*: we try 40000000 trials on 1 CPU core (1 child process)
    - *Case 2*: we try 40000000/2 trials on 2 CPU core (2 child processes)
    - *Case 3*: we try 40000000/4 trials on 4 CPU core (4 child processes)
  - Child processes need to send the parent process the results

Questions?

- Some design consideration for Android?
- Take advantage of multiprocess architecture
  - Information sharing
  - Computation speed-up
  - Modularity

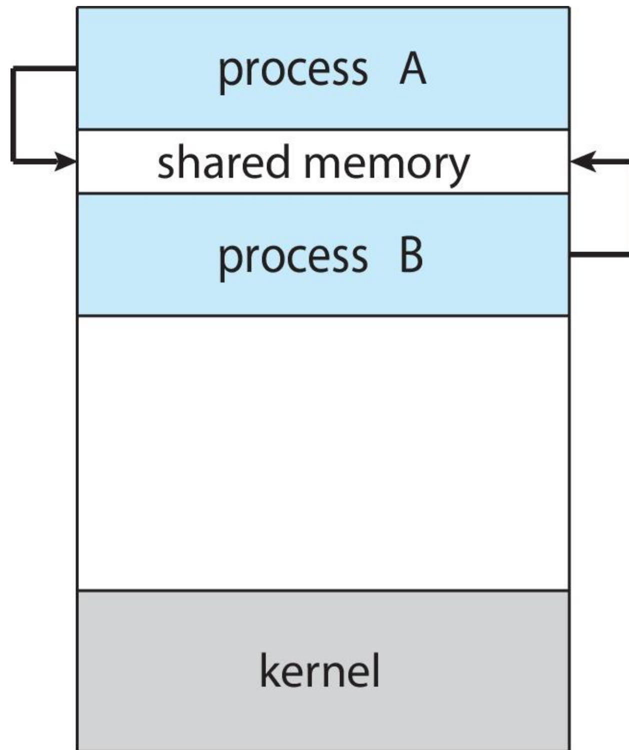
Interprocess Communication

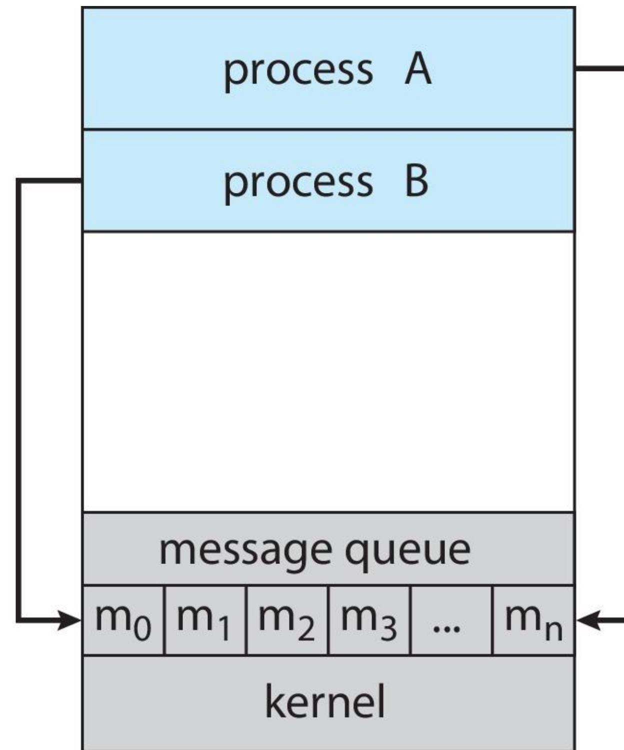
- Cooperating processes need **interprocess communication** (**IPC**). There are two models of IPC:
  - **Shared memory**
    - Processes share a region of memory, and they can read and write to it
    - System calls are required only to establish the shared-memory regions.
    - All accesses are treated as routine memory accesses, and no assistance from the kernel is required. Typically faster than message passing.
    - Conflicts may arise (both processes write to the same area in the region)
  - **Message passing**
    - Processes exchange messages.
    - Good for small messages, and for distributed systems where there is no shared physical memory between processes on multiple hosts.
    - There is no conflict needed to be avoided.

Communications Models

(a) Shared memory.

(b) Message passing.





Questions?

- Concept and benefits of interprocess communication
- Concept of shared memory and message passing

IPC for Shared Memory Systems

- OS must provide a system call to create a shared memory region, and communicating processes must attach this shared memory segment to their address space.
- OS must removes the restriction that normally one process is prevented from accessing another process's memory.
- The processes can then exchange information by reading and writing data in the shared areas.
- The processes are also responsible for ensuring that they are not writing to the same location simultaneously.
- The producer–consumer problem is a common paradigm for cooperating processes.

Process Synchronization

- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to **synchronize** their actions when they access shared memory.
- The producer–consumer problem is a common paradigm for cooperating processes.
- Synchronization is discussed in great details in a few weeks

Producer-Consumer Problem

- Paradigm for cooperating processes
- *Producer* process produces information that is consumed by a *consumer* process
- The information is stored in a memory buffer
  - **unbounded-buffer** places no practical limit on the size of the buffer
  - **bounded-buffer** assumes that there is a fixed buffer size

Bounded-Buffer: Shared-Memory Solution

- Shared data
- Producer
- Consumer
- At present, we do not address concurrent access to the shared memory by the producer and the consumer.

Producer-Consumer: Shared Data via Shared Memory

- Share BUFFER_SIZE – 1 items

```
#define BUFFER_SIZE 10
typedef struct {
  . . .
} item;

// The following are shared among cooperating processes
item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer-Consumer: Buffer Full or Empty?

- The bounded buffer works in a cyclic fashion (cyclic buffer, circular array)
    - Index: `0, 1, 2, … BUFFER_SIZE-1, 0, 1, …`
  - So
    - `(in + 1) % BUFFER_SIZE`
    - `(out + 1) % BUFFER_SIZE`
- Buffer Empty

  When `in == out`
- Buffer Full

  When `(((in + 1) % BUFFER_SIZE) == out`

Producer Process via Shared Memory

```
item next_produced;

while (true) {
  /* produce an item in next produced */
  while (((in + 1) % BUFFER_SIZE) == out)
      ; /* do nothing when buffer is full */
  buffer[in] = next_produced;  /* write it */
  in = (in + 1) % BUFFER_SIZE; /* next slot */
}
```

Consumer Process via Shared Memory

```
item next_consumed;

while (true) {
      while (in == out)
       ; /* do nothing when buffer is empty */
      next_consumed = buffer[out];
      out = (out + 1) % BUFFER_SIZE;

      /* consume the item in next_consumed */
}
```

Producer-Consumer: Require Synchronization

- Both producer and consumer may read and write to the shared memory concurrently (future lessons)

```
item next_produced;

while (true) {

  while (((in + 1) % BUFFER_SIZE)
  == out)

        ;

  buffer[in] = next_produced;

  in = (in + 1) % BUFFER_SIZE;

}
```

```
item next_consumed;

while (true) {
        while (in == out)

            ;
        next_consumed = buffer[out];

        out = (out + 1) % BUFFER_SIZE;


        /* consume next_consumed */

}
```

Questions?

- Producer-consumer problem for shared memory
- Bounded-buffer producer and consumer problem for shared memory
  - What data are shared in the shared memory?
  - How do we know the buffer is full?
  - How do we know the buffer is empty?
  - Why process synchronization is needed?

Message Passing

- Mechanism for processes to communicate and to synchronize their actions
- Message system – processes communicate with each other without resorting to shared variables
- IPC facility provides two operations:
  - **`send`**(*message*)
  - **`receive`**(*message*)
- The *message* size is either fixed or variable

Message Passing: Implementation Issues

- If processes *P* and *Q* wish to communicate, they need to:
  - Establish a ***communication link*** between them
  - Exchange messages via send/receive
- Implementation issues:
  - How are links established?
  - Can a link be associated with more than two processes?
  - How many links can there be between every pair of communicating processes?
  - What is the capacity of a link?
  - Is the size of a message that the link can accommodate fixed or variable?
  - Is a link unidirectional or bi-directional?

Communication Link

- Implementation of communication link
  - Physical:
    - Shared memory
    - Hardware bus
    - Network
  - Logical:
    - Direct or indirect
    - Synchronous or asynchronous
    - Automatic or explicit buffering

Message Passing: Design Consideration

- Naming
  - Direction and indirect communication
- Synchronization
  - Blocking vs. non-blocking; synchronous vs. asynchronous
- Buffering

Questions?

- Concept of message passing
- General issues and design considerations

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 and Primitives

- Operations
  - create a new mailbox (port)
  - 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*

Indirect Communication: Mailbox Sharing?

- Mailbox sharing
  - P_1, P_2, and P_3 share mailbox *A*
  - P_1, sends; P_2 and P_3 receive
  - Who gets the message?
- Solutions
  - Allow a link to be associated with at most two processes
  - Allow only one process at a time to execute a receive operation
  - Allow the system to select arbitrarily the receiver. Sender is notified who the receiver was.

Questions?

- Direction and indirect communications
- Comparison between Direction and indirect communications

Message Passing: Synchronization

- Message passing may be either blocking or non-blocking
  - Blocking is considered synchronous
  - Blocking send -- the sender is blocked until the message is received
  - Blocking receive -- the receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send -- the sender sends the message and continue
  - Non-blocking receive -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations of blocking and non-blocking possible
- If both send and receive are blocking, we have a rendezvous

Producer-Consumer via Blocking Sender and Receiver

- Producer

```
message next_produced;

while (true) {
    /* produce an item in next_produced */
   send(next_produced); /* blocking */

}
```

- Consumer

```
message next_consumed;

while (true) {
    receive(next_consumed); /* blocking */

    /* consume the item in next_consumed */

}
```

Questions?

- Blocking vs non-blocking
- The producer-consumer problem for blocking sender and blocking receiver (i.e., rendezvous)
- Synchronous vs. asynchronous

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  - Zero capacity – no messages are queued on a link.
    Sender must wait for receiver (rendezvous)
  - Bounded capacity – finite length of n messages
    Sender must wait if link full
  - Unbounded capacity – infinite length
    Sender never waits

Questions?

- Buffering for message passing
- When must the sender wait?