CSC 4320/6320: Operating Systems

Chapter 03: Processes – part II

Spring 2025

Instructor: Dr. Md Mahfuzur Rahman
Department of Computer Science, GSU

Disclaimer

The slides to be used in this course have been created, modified, and adapted from multiple sources:

- *The slides are copyright Silberschatz, Galvin and Gagne, 2018. The slides are authorized for personal use, and for use in conjunction with a course for which Operating System Concepts is the prescribed text. Instructors are free to modify the slides to their taste, as long as the modified slides acknowledge the source and the fact that they have been modified.*

Outline

- Process Concept ✓
- Process Scheduling ✓
- Operations on Processes ✓
- Interprocess Communication
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems

Objectives

- Identify the separate components of a process and illustrate how they are represented and scheduled in an operating system. ✓
- Describe how processes are created and terminated in an operating system, including developing programs using the appropriate system calls that perform these operations.✓
- Describe and contrast interprocess communication using shared memory and message passing.
- Design programs that uses pipes and POSIX shared memory to perform interprocess communication.

Orphan Processes

- If a parent process terminates before its child terminates, the child process is automatically adopted by the ***init*** process
- The following program shows this.

Example Program of Orphan

The parent process finishes earlier than its child, making the child an “orphan” process.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
int main () {
    int pid;
    printf ("I'm the original process with PID %d and PPID %d.\n", getpid(), getppid());
    pid = fork (); /* Duplicate. Child & parent continue from here */
    if (pid != 0) /* Branch based on return value from fork () */ {
    /* pid is non-zero, so I must be the parent */
        printf ("I'm the parent process with PID %d and PPID %d.\n",getpid (), getppid ());
        printf ("My child's PID is %d\n", pid);
    }
    else {
      /* pid is zero, so I must be the child */
      sleep(5); /* Make sure that the parent terminates first */
      printf ("I'm the child process with PID %d and PPID %d.\n",getpid (), getppid ());
    }
    printf ("PID %d terminates.\n", getpid() ); /* Both processes execute this */
    return 0;
}
```

Zombie Processes

- A process cannot leave the system until parent process accepts its termination code (returned by exit)
- If parent process is dead; *init* adopts process and accepts code

Orphan process

- If the parent process is alive but is unwilling to accept the child's termination code
  - because it never executes wait()
  - the child process will remain a zombie process.

Zombie process

Zombie Example

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main () {
    pid_t pid;
    pid = fork (); /* Duplicate */
    /* Branch based on return value from fork () */
    if (pid != 0) {
        /* Never terminate, never execute a wait () */
        while (1)
          sleep (1000);
    }
    else {
      exit (42); /* Exit with a silly number */
    }
    return 0;
}
```

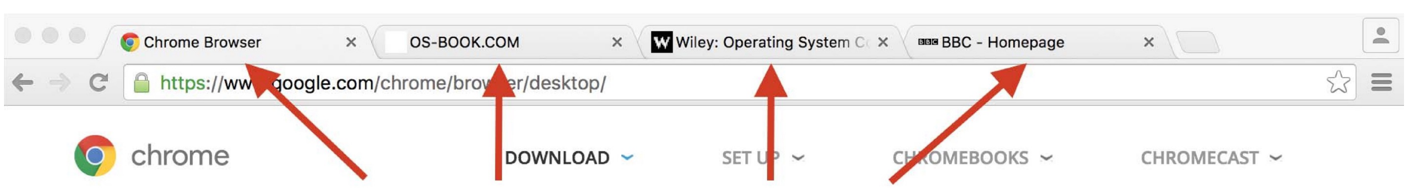
The parent process continues to run but it is not waiting/willing to receive the termination code from its child, eventually making the child a “zombie” process.

Android Process Importance Hierarchy

- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From **most** to **least** important:
  - Foreground process (current process visible on screen)
  - Visible process (not directly interacting, but visible, e.g., YouTube video in Picture-in-Picture mode)
  - Service process (background process, but output is apparent, e.g. downloading files, streaming music)
  - Background process (not apparent, e.g., a message app synching messages)
  - Empty process (no active components. e.g., shopping app not in use)
- Android will begin terminating processes that are least important.

Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
  - 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 (extension, add-on)



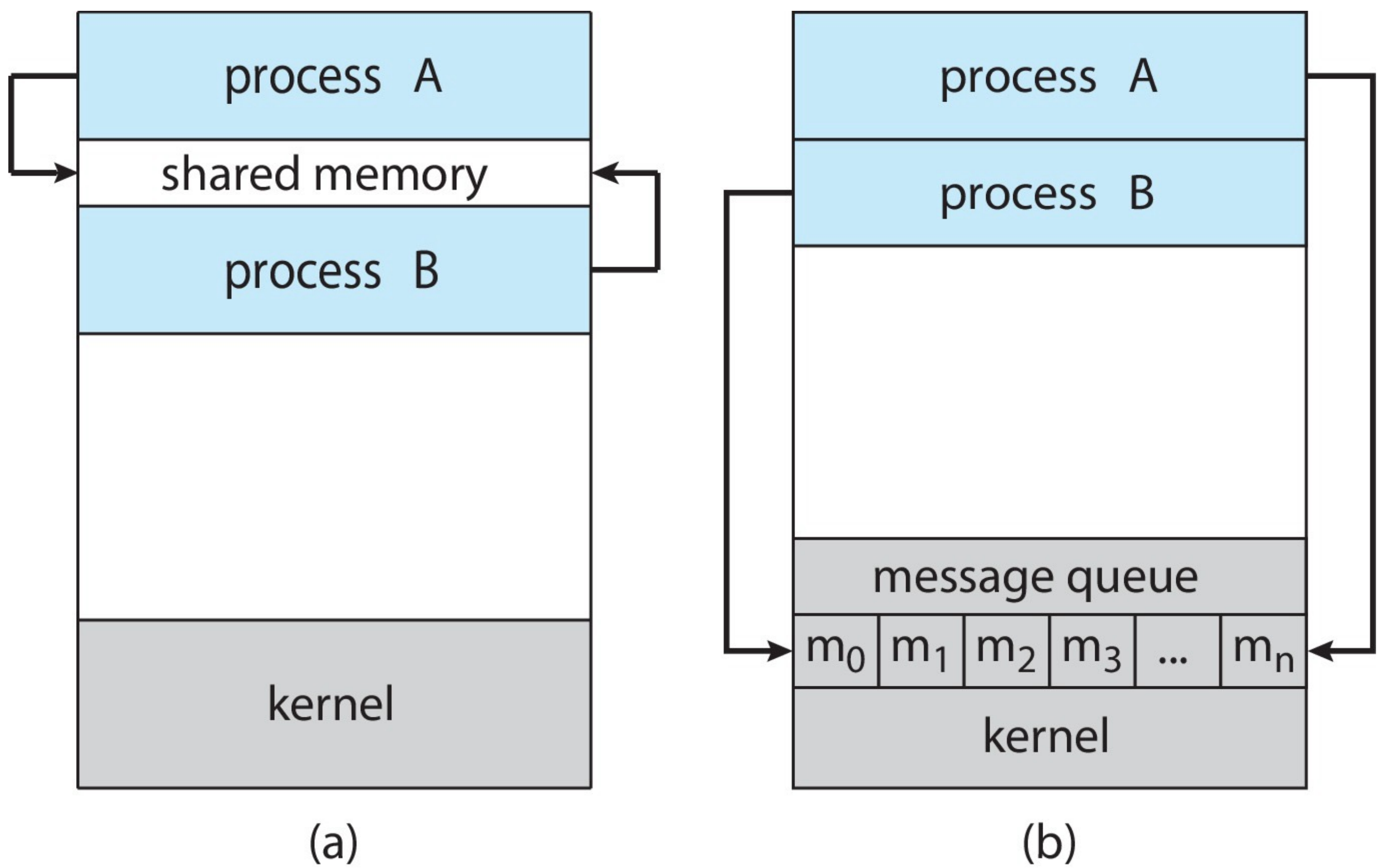
Each tab represents a separate process.

Interprocess Communication

- Processes within a system may be ***independent*** or ***cooperating***
- Cooperating process can affect or be affected by other processes (e.g., processes sharing data are cooperating)
- Reasons for cooperating processes:
  - Information sharing (e.g., same file sharing)
  - Computation speedup (e.g., subtasks via multicore processing)
  - Modularity (e.g., modularizing via threads)
  - Convenience
- Cooperating processes need **interprocess communication** (**IPC**)
- Two models of IPC
  - **Shared memory**
  - **Message passing**

Communications Models

(a) Shared memory. (b) Message passing.



Message Passing vs. Shared Memory

Message passing:
- useful for exchanging smaller amounts of data, because no conflicts need be avoided.
- Message passing is also easier to implement in a distributed system than shared memory.
- Message passing model requires the more time-consuming task of kernel intervention.

Shared memory:
- faster than message passing, since message-passing systems are typically implemented using system calls
- system calls are required only to establish shared-memory regions.
- Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Questions

Which of the following statements is true?

A) Shared memory is typically faster than message passing. ✓
B) Message passing is typically faster than shared memory.
C) Message passing is most useful for exchanging large amounts of data.
D) Shared memory is far more common in operating systems than message passing.

Shared memory is a more appropriate IPC mechanism than message passing for distributed systems.

True
False ✓

Producer-Consumer Problem

- Paradigm for cooperating processes:
  - *producer* process produces information that is consumed by a *consumer* process
- Two variations:
  - **unbounded-buffer** places no practical limit on the size of the buffer:
    - Producer never waits
    - Consumer waits if there is no buffer to consume
  - **bounded-buffer** assumes that there is a fixed buffer size
    - Producer must wait if all buffers are full
    - Consumer waits if there is no buffer to consume

IPC – Shared Memory

- An area of memory shared among the processes that wish to communicate
- 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.
- Synchronization is discussed in great details in Chapters 6 & 7.

Bounded-Buffer – Shared-Memory Solution

- Shared data

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

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

- Solution is correct, but can only use **`BUFFER_SIZE-1`** elements

Producer Process – Shared Memory

```
item next_produced;

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

Circular array – the last location will not be used.
`in` points to the next free location
`out` points to the first filled location

`(in + 1) % buffer_size == out` means, buffer is full

Consumer Process – Shared Memory

```
item next_consumed;

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

    /* consume the item in next consumed */
}
```

Question

The shared buffer is implemented as a circular array with two logical pointers: `in` and `out`. The variable `in` points to the next free position in the buffer; `out` points to the first full position in the buffer. The buffer is full when `in == out`.

True

False ✓

What about Filling all the Buffers?

- Suppose that we wanted to provide a solution to the consumer-producer problem that fills **all** the buffers.
- We can do so by having an integer `counter` that keeps track of the number of full buffers.
- Initially, `counter` is set to 0.
- The integer `counter` is incremented by the producer after it produces a new buffer.
- The integer `counter` is decremented by the consumer after it consumes a buffer.

Producer

```
while (true) {
    /* produce an item in next produced */

    while (counter == BUFFER_SIZE)
        ; /* do nothing */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Consumer

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;
    /* consume the item in next consumed */
}
```

Race Condition

- `counter++` could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- `counter--` could be implemented as

```
register2 = counter
register2 = register2 - 1
counter = register2
```

- Consider this execution interleaving with "count = 5" initially:

S0: producer execute `register1 = counter`	{register1 = 5}
S1: producer execute `register1 = register1 + 1`	{register1 = 6}
S2: consumer execute `register2 = counter`	{register2 = 5}
S3: consumer execute `register2 = register2 - 1`	{register2 = 4}
S4: producer execute `counter = register1`	{counter = 6 }
S5: consumer execute `counter = register2`	{counter = 4}

Race Condition

- A **race condition** in a shared memory solution occurs when multiple processes or threads access and manipulate shared data concurrently

- The final result depends on the order in which the operations are executed.

- This situation can lead to **unpredictable behavior** and **data corruption**, as the processes "race" to access the shared resource without proper synchronization.

Race Condition (Cont.)

- Question – why was there no race condition in the first solution (where at most N – 1) buffers can be filled?
- More in Chapter 6.

IPC – Message Passing

- 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 (Cont.)

Georgia State University

- 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?

Implementation of Communication Link

- Physical:
  - Shared memory
  - Hardware bus
  - Network
- Logical:
  - Direct or indirect
  - Synchronous or asynchronous
  - Automatic or explicit buffering (bounded/unbounded capacity of queue)

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 (each referring to one mailbox)
  - Link may be unidirectional or bi-directional

Indirect Communication (Cont.)

- 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.

Indirect Communication (Cont.)

- Operations
  - Create a new mailbox (port) (by owner)
  - Send and receive messages through mailbox
  - Delete 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** -- 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 possible
  - If both send and receive are blocking, we have a **rendezvous**

Producer-Consumer: Message Passing

- Producer

```
message next_produced;

while (true) {

  /* produce an item in next_produced */

 send(next_produced);
}
```

- Consumer

```
message next_consumed;

while (true) {
  receive(next_consumed)

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

Buffering

- Queue of messages attached to the link.
- Implemented in one of three ways
  1. Zero capacity – no messages are queued on a link.
     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

Questions

Q1. A blocking `send()` and blocking `receive()` is known as a(n) ________________

A) synchronized message

B) rendezvous ✓

C) blocked message

D) asynchronous message

Q2. In a(n) ____ temporary queue, the sender must always block until the recipient receives the message.

A) zero capacity ✓

B) variable capacity

C) bounded capacity

D) unbounded capacity

Shared Memory Model - Example

producer.c

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

int main()
{

/* the size (in bytes) of shared memory object */
const int SIZE = 4096;

/* name of the shared memory object */
const char *name = "OS";

/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int fd;

/* pointer to shared memory obect */
char *ptr;

/* create the shared memory object */
fd = shm_open(name,O_CREAT | O_RDWR,0666);

/* configure the size of the shared memory object */
ftruncate(fd, SIZE);
/* memory map the shared memory object */
ptr = (char *)
    mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

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

return 0;
}
```

consumer.c

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

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;

/* name of the shared memory object */
const char *name = "OS";

/* shared memory file descriptor */
int fd;

/* pointer to shared memory obect */
char *ptr;

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

/* memory map the shared memory object */
ptr = (char *)
mmap(0, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);

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

/* remove the shared memory object */
shm_unlink(name);

return 0;
}
```

Executing the shared memory model

```
gcc -o producer producer.c -lrt
```

(on snowball linux, `-lrt` is required: lrt stands for real time library handling shared memory)

```
gcc -o consumer consumer.c -lrt
./producer
./consumer
```

Each memory map has a corresponding file location in the file system.

Use: `ls -l /dev/shm/` to see the object OS