Western

UNIVERSITY · CANADA

Chapter 3D – Interprocess Communications

Spring 2023

Interprocess Communications

- Interprocess Communications
- IPC in Shared-Memory Systems
- IPC in Message-Passing Systems
- Pipes
- Examples

Interprocess Communications

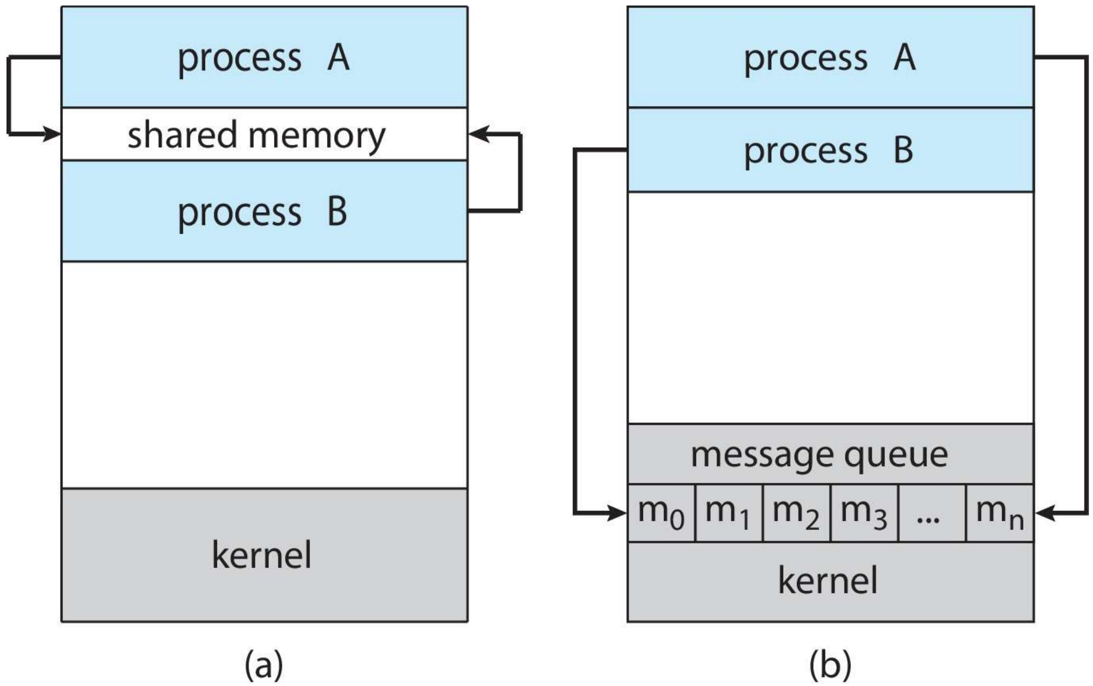
- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes:
  - Information sharing
  - Computation speedup
  - Modularity
  - Convenience

Interprocess Communications

- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - **Shared memory** – Managed by the user processes
  - **Message passing** – Managed by the operating system

Interprocess Communications

- Shared Memory vs. Message Passing



Producer-Consumer problem

- The Producer-Consumer problem. Paradigm for cooperating processes
  - How does the producer produce data that can be consumed by the consumer?
  - How do we prevent data loss?
- E.g.
  - Compiler -> Assembler -> Loader
  - Web server -> Client web browser

Producer-Consumer problem

- 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

Shared-Memory

- 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 greater detail in Chapters 6 & 7.

Shared-Memory

- IPC – Shared Memory
  - ```
    #define BUFFER_SIZE 10
    typedef struct {
    …
    } item;

    item buffer[BUFFER_SIZE];
    int in = 0; //points to the next item to produce
    int out = 0; //points to the next item to consume
    ```
- This circular buffer can only use BUFFER_SIZE-1 elements
  - in == out is, by definition, an empty buffer, so any attempt to fill the last slot will make in == out again. But the buffer is not actually empty!
    ```

Shared-Memory

- IPC – Shared Memory
  - How do we use all slots?
    - Share a counter

Shared-Memory

- 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++;
}
```

Shared-Memory

- Consumer

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

Shared-Memory

- A counter introduces a new problem:
  - `counter` could be set to 5
  - `counter++` could set the value to 6 in one process
  - `counter--` could set the value to 4 in the other process
- Solutions in chapter 6

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

- If processes P and Q wish to communicate, they need to:
  - Establish a communication link between them
  - Exchange messages via send/receive

Message-Passing

- 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?

Message-Passing

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

Message-Passing

- 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

Message-Passing

- Properties of direct communication links
  - 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

Message-Passing

- 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

Message-Passing

- Properties of indirect communication links
  - 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

Message-Passing

- Indirect communication
  - Operations
    - Create a new mailbox (port)
    - 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

Message-Passing

- Indirect communication
  - Mailbox sharing
    - Suppose P1, P2, and P3 share mailbox A. P1, sends; P2 and P3 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.

Message-Passing

- Synchronization
  - **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

Message-Passing

- Synchronization
  - **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**

Message-Passing

- 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

Pipes

- Pipes provide a simple one-to-one communication channel between processes
- Issues:
  - Is communication **unidirectional** or **bidirectional**?
  - In the case of two-way communication, is it half (one direction at a time) or full-duplex (both directions at any time)?
  - Must there exist a relationship (i.e., parent-child) between the communicating processes?
  - Can the pipes be used over a network?

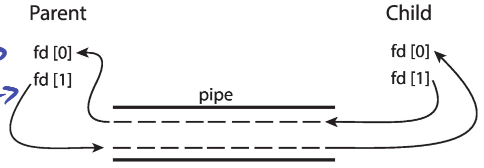
Pipes

- **Ordinary pipes** – cannot be accessed from outside the process that created it. Typically, a parent process creates a pipe and uses it to communicate with a child process that it created.
- **Named pipes** – can be accessed without a parent-child relationship.

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** (use 2 pipes for bi-directional e.g. fork())
- Require parent-child relationship between communicating processes
- Windows calls these **anonymous pipes**

read end ->
write end ->



Pipes

- Named Pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating processes (could be used over networks)
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems

Pipes

- When the pipe is **full**: By default, if a writing process attempts to write to a full pipe
    - The system will automatically block the process until the pipe is able to receive the data
    - The OS has a limit on the buffer space used by the pipe and if you hit the limit, write will be blocked
- When the pipe is **empty**: if a read is attempted on an empty pipe, the process will block until data is available

Examples

- Shared-memory: Use `shm_open()`, `ftruncate()`, `mmap()`, `shm_unlink()` to map a portion of a file descriptor in memory. Then use `sprintf()` to write to the shared memory.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;

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

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_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;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.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 shm_fd;
/* pointer to shared memory obect */
void *ptr;

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

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

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

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

   return 0;
}
```

Examples

- Pipes: Use `pipe()` to create a pipe, then use `fork()`

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>
int main(void){
    int n;
    int status;
    char c;
    int port[2];
    pid_t pid;
```

Examples

- Pipes: Use `pipe()` to create a pipe, then use `fork()`

```
if (pipe(port) < 0){
    perror("pipe error");  exit(0);
}
pid = fork();
if (pid < 0) {
    perror("fork error");
    exit(0);
}
```

Examples

- Pipes: Use `write()` to send data to `port[1]` (the writing end of the pipe)

```
if(pid > 0){ //parent
    printf("\n From parent: writing ABCD to pipe now..");
    write(port[1],"ABCD",4);
    printf("\n From parent: waiting for child to complete..\n");
    wait(NULL);
}
```

writing end

number of byte.

wait for the child
to continue.

Examples

- Pipes: Use `read()` to read data from `port[0]` (the reading end of the pipe)

```
    else { //child
        printf("\n From Child: reading A from the pipe now..");
        read (port[0],&c,1);
        printf("\n from child: this is what I read %c\n", c);
    }
    return 0;
}
```

read one char from the pipe.

size_t

Examples

- Pipes: Writing and reading different types

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <errno.h>

int main(void) {
    int port[2];
    if (pipe(port) < 0){
        perror("pipe error");
        exit(1);
    }
    pid_t pid = fork();
```

Examples

- Pipes: Writing and reading different types

```
if (pid<0){
     perror("fork error");
     exit(1);
}
if (pid>0){ //parent
    char c = 'A';
    char s[4] = "ABC"; //Note: sizeof(s) == 4
    int a = 1234;
    write(port[1],&c,sizeof(c)); // write char
    write(port[1],s,sizeof(s)); // write string
    write(port[1],&a,sizeof(a)); // write int
    wait(NULL);
}
```

Examples

- Pipes: Writing and reading different types

```
    else{ //child
        char d;
        char t[4]; //Note: sizeof(t) == 4
        int b;
        read(port[0],&d,sizeof(d)); //read char
        read(port[0],t,sizeof(t)); //read string
        read(port[0],&b,sizeof(b)); //read int
        printf("%c %s %d\n", d, t, b);
    }
    return 0;
}
```

1878

Western

UNIVERSITY · CANADA