Operating System
CS-2006
Lecture 5

Mahzaib Younas

Lecturer Department of Computer Science

FAST NUCES CFD

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

A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced

Bonded Buffer

- 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

Bounded Buffer (Cont…)

- 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 empty

when in == out

- the buffer is full

when ((in + 1) % BUFFER SIZE) == out.

Producer Process in 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;
}
```

Consumer Process in 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 */
}
```

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}

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

Implementation Issues in 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**

Implementation of Communication Link

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

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 (Cont…)

- 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**

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?

Solution:

- 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.**

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
- **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

Pipes

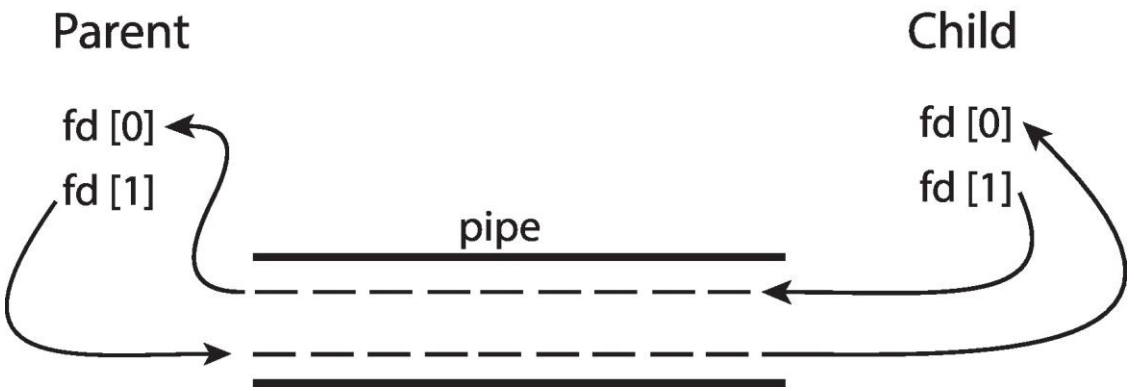
- Acts as a **conduit allowing two processes to communicate**
- Issues:
  - Is communication **unidirectional or bidirectional**?
  - In the case of two-way communication, is it **half or full-duplex**?
  - Must there exist a relationship (i.e., ***parent-child***) between the **communicating processes?**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

Types of 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**

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**
- Require **parent-child relationship between communicating processes**



- Windows calls these **anonymous pipes**

Ordinary Pipes

Creating a pipe

#include<unistd.h>

int pipe(int pipefd[2]);

This pipe() system call creates a pipe with two file descriptors:

one for **writing fd[1],**

Second for **reading fd[0].**

Writing pipe

#include<unistd.h>

ssize_t write(int pipefd, void* buff, size_t s);

The **file descriptor** of the write end of the pipe, pipefd, is returned by the pipe() system call.

A **buffer, buff,** contains data that needs to be written to the pipe end.

The buffer can be **dynamic or static in size**, and the size is sent as a parameter, s.

Ordinary Pipes

Reading pipe

#include<unistd.h>

ssize_t returnsize = read(int pipefd, void* buff, size_t s);

The **file descriptor** of the read end of the pipe, pipefd, is returned by the pipe() system call.

A **buffer, buff,** is used to store the data read from the pipe.

The number of bytes that needs to be read from the pipe is sent as a **parameter, s.**

Close pipe

#include<unistd.h>

int close(int pipefd);

Sample Code

Libraries

```
#include <stdio.h>
#include <unistd.h>

int main() {
  int pipefd[2];
  char msg1[20] = "Message 1";
  char msg2[20] = "Message 2";
  char readmsg[20] = "";
```

Pipe Closing

```
// Create a pipe
  int returnstatus = pipe(pipefd);
  // Check for errors...
```

Process Working

```
// Create a child process
int pid = fork();
// Check for errors...

if (pid == 0) { // Child process
  close(pipefd[1]);
  read(pipefd[0], readmsg, 20);
  // Process data...
  close(pipefd[0]);
} else { // Parent process
  close(pipefd[0]);
  // Write data...
  close(pipefd[1]);
}

  return 0;
}
```

Named Pipes

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