# Operating Systems 2023 Spring Term
# Week 4

Dr. Emrah İnan (emrahinan@iyte.edu.tr)

Interprocess Communication II - Threads I

March 30, 2023

# Week 3: Sample Glossary

- **producer:** A process role in which the process produces information that is consumed by a consumer process. (on Page 1264)
- **consumer:** A process role in which the process consumes information produced by a producer process. (on Page 1243)
- **shared-memory model:** An interprocess communication method in which multiple processes share memory and use that memory for message passing. (on Page 1269)
- **bounded buffer:** A buffer with a fixed size. (on Page 1240)

# Bounded-Buffer – Shared-Memory Solution

Implemented as a circular array with two logical pointers
in -> the next free position in the buffer
out -> the first full position in the buffer
Solution is correct, but can only use BUFFER_SIZE-1 elements

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

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

# 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;
}
```

# 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 */
}
```

# 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 and is decremented by the consumer after it consumes a buffer.

# Producer-Consumer Example: 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++;
}
```

# Producer-Consumer Example: 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} |

# IPC – Message Passing I

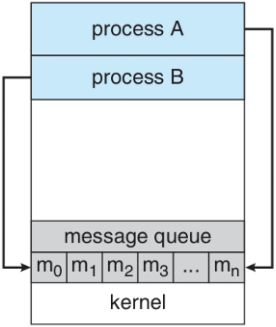
Processes communicate with each other without resorting to shared variables

IPC facility provides two operations:

send(message)

receive(message)

The message size is either fixed or variable



# Message Passing II

- 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

# 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 I

- 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 II

- 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 III

- 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.

# 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

# Examples of IPC Systems - POSIX

- POSIX Shared Memory
  Process first creates shared memory segment
  shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
  Also used to open an existing segment
  Set the size of the object
  ftruncate(shm_fd, 4096);
  Use mmap() to memory-map a file pointer to the shared memory object
  Reading and writing to shared memory is done by using the pointer returned by mmap().

# IPC POSIX Producer

```
#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;
}
```

# IPC POSIX Consumer

```
#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 of IPC Systems - Mach

- Mach communication is message based
  Even system calls are messages
  Each task gets two ports at creation - Kernel and Notify
  Messages are sent and received using the mach_msg() function
- Ports needed for communication, created via mach_port_allocate()
  Send and receive are flexible; for example four options if mailbox full:
  Wait indefinitely
  Wait at most n milliseconds
  Return immediately
  Temporarily cache a message

# Mach Messages

```
#include<mach/mach.h>

struct message {
        mach_msg_header_t header;
        int data;
};

mach port t client;
mach port t server;
```

# Mach Message Passing - Client

```
/* Client Code */

struct message message;

// construct the header
message.header.msgh_size = sizeof(message);
message.header.msgh_remote_port = server;
message.header.msgh_local_port = client;

// send the message
mach_msg(&message.header, // message header
  MACH_SEND_MSG, // sending a message
  sizeof(message), // size of message sent
  0, // maximum size of received message - unnecessary
  MACH_PORT_NULL, // name of receive port - unnecessary
  MACH_MSG_TIMEOUT_NONE, // no time outs
  MACH_PORT_NULL // no notify port
);
```

# Mach Message Passing - Server

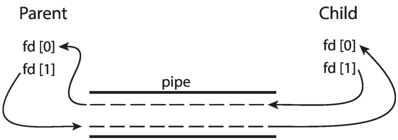
```
/* Server Code */

struct message message;

// receive the message
mach_msg(&message.header, // message header
  MACH_RCV_MSG, // sending a message
  0, // size of message sent
  sizeof(message), // maximum size of received message
  server, // name of receive port
  MACH_MSG_TIMEOUT_NONE, // no time outs
  MACH_PORT_NULL // no notify port
);
```

# 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 – 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

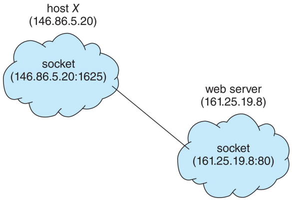


# Named Pipes

- 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

# Sockets

- A socket is defined as an endpoint for communication
- Concatenation of IP address and port – a number included at start of message packet to differentiate network services on a host
- The socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between a pair of sockets
- All ports below 1024 are well known, used for standard services
- Special IP address 127.0.0.1 (loopback) to refer to system on which process is running



# Sockets in JAVA

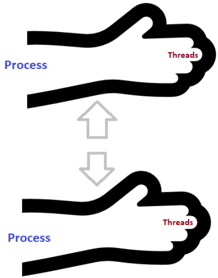
- Three types of sockets
  Connection-oriented (TCP)
  Connectionless (UDP)
  MulticastSocket class– data can be sent to multiple recipients
- Consider this “Date” server in Java:

```
import java.net.*;
import java.io.*;

public class DateServer
{
   public static void main(String[] args) {
      try {
         ServerSocket sock = new ServerSocket(6013);

         /* now listen for connections */
         while (true) {
            Socket client = sock.accept();

            PrintWriter pout = new
             PrintWriter(client.getOutputStream(), true);

            /* write the Date to the socket */
            pout.println(new java.util.Date().toString());

            /* close the socket and resume */
            /* listening for connections */
            client.close();
         }
      }
      catch (IOException ioe) {
         System.err.println(ioe);
      }
   }
}
```

# Sockets in Java: Date client

```
import java.net.*;
import java.io.*;

public class DateClient
{
  public static void main(String[] args) {
    try {
      /* make connection to server socket */
      Socket sock = new Socket("127.0.0.1",6013);

      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      /* read the date from the socket */
      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      /* close the socket connection*/
      sock.close();
    }
    catch (IOException ioe) {
      System.err.println(ioe);
    }
  }
}
```

# Process and Threads

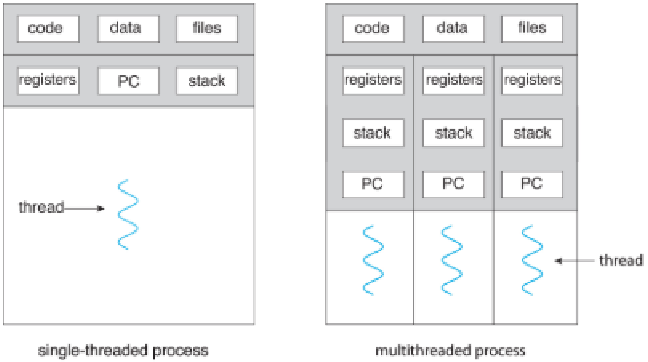



Sophie Wilson (https://en.wikipedia.org/wiki/Sophie_Wilson)
Steve Furber (https://en.wikipedia.org/wiki/Steve_Furber)
(https://community.arm.com/arm-community-blogs/b/architectures-and-processors-blog/posts/a-brief-history-of-arm-part-1)

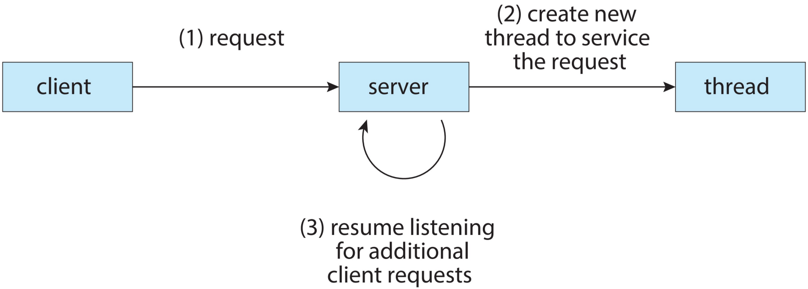
# Threads & Concurrency

- Most modern applications are multithreaded
- Threads run within application
- Multiple tasks with the application can be implemented by separate threads
  Update display
  Fetch data
  Spell checking
  Answer a network request
- Process creation is heavy-weight while thread creation is light-weight
- Can simplify code, increase efficiency
- Kernels are generally multithreaded

# Single and Multithreaded Processes



# Multithreaded Server Architecture



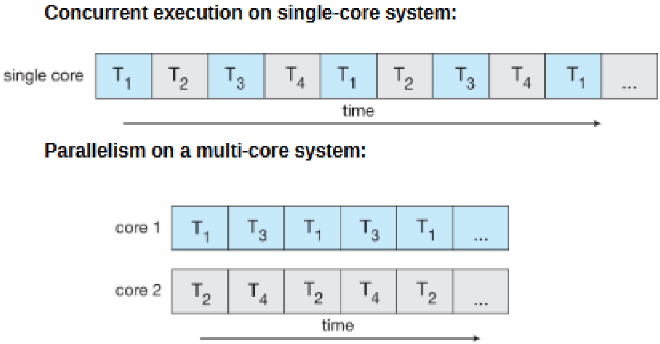
# Threads & Concurrency

- **Responsiveness** – may allow continued execution if part of process is blocked, especially important for user interfaces
- **Resource sharing** – threads share resources of process, easier than shared memory or message passing
- **Economy** – cheaper than process creation, thread switching lower overhead than context switching
- **Scalability** – process can take advantage of multicore architectures

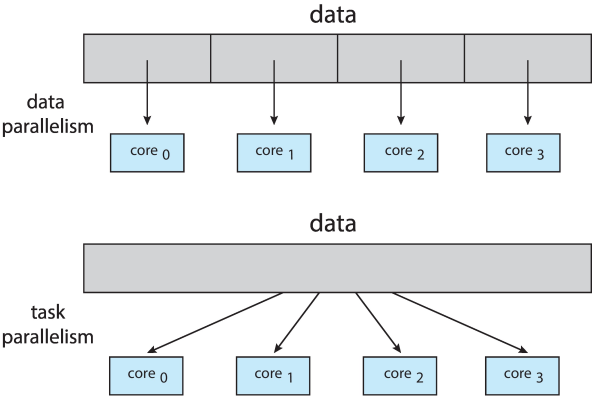
# Multicore Programming

- **Multicore or multiprocessor** systems puts pressure on programmers, challenges include:
  Dividing activities
  Balance
  Data splitting
  Data dependency
  Testing and debugging
- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  Single processor / core, scheduler providing concurrency
- Types of parallelism
  **Data parallelism** – distributes subsets of the same data across multiple cores, same operation on each
  **Task parallelism** – distributing threads across cores, each thread performing unique operation

# Concurrency vs. Parallelism



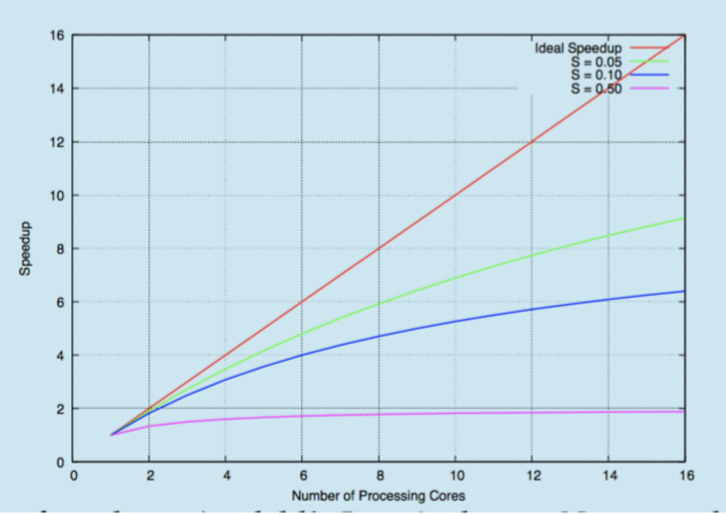
# Data and Task Parallelism



# Amdahl's Law

- Identifies performance gains from adding additional cores to an application that has both serial and parallel components
- S is serial portion
- N processing cores $$speedup \le \frac{1}{S + \frac{(1-S)}{N}}$$
- That is, if application is 75% parallel / 25% serial, moving from 1 to 2 cores results in speedup of 1.6 times
- As N approaches infinity, speedup approaches 1 / S
- Serial portion of an application has disproportionate effect on performance gained by adding additional cores
- But does the law take into account contemporary multicore systems?

# Amdahl's Law



Ideal Speedup
S = 0.05
S = 0.10
S = 0.50
Speedup
Number of Processing Cores
0
2
4
6
8
10
12
14
16