COMP3511 Operating Systems

Topic 3: Process Concept

Dr. Desmond Tsoi

Department of Computer Science & Engineering
The Hong Kong University of Science and Technology
Hong Kong SAR, China

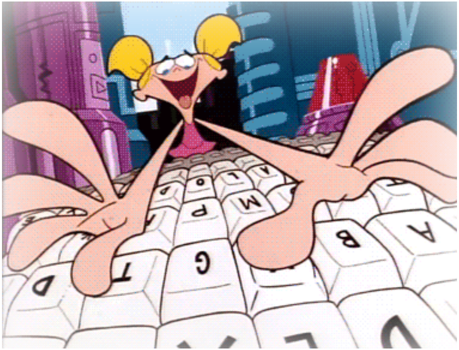
Acknowledgment: The lecture notes are based on various sources on the Internet

Fundamental Concepts of Process

- Introduction
- Definition of Process
- I/O-bound Process vs. CPU-bound Process
- Program vs. Process
- Process States and Diagram
- Process Control Block (PCB) and CPU Switch
- Threads

Introduction

- An OS executes a variety of programs
  - In batch systems, referred to as jobs
  - In time shared systems, referred to as user programs or tasks
  - So far pretty informally referred to as programs in execution, processes, jobs, tasks...
  - From now on, we will try to use the term process synonymously with the above terms and really mean...

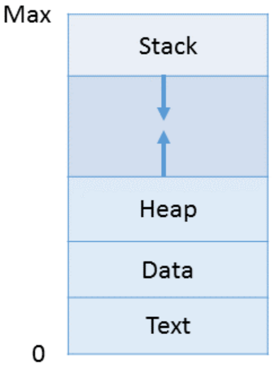


Definition of Process

Process

A program in execution; process execution must progress in sequential fashion

- A process include
  - The program code, also called text section
  - Currently activity, represented by the program counter, processor registers
  - Stack containing temporary data (such as function parameters, return addresses, local variables)
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time

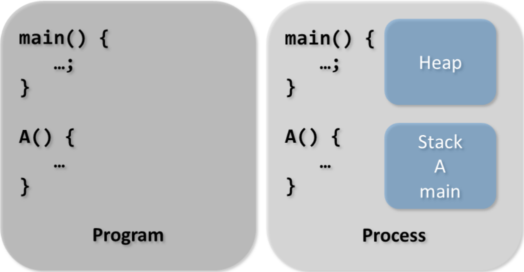


I/O-bound Process vs. CPU-bound Process

Processes can be described as either

- I/O-bound process: spends more time doing I/O than computations, many short CPU bursts
- CPU-bound process: spends more time doing computations; few very long CPU bursts

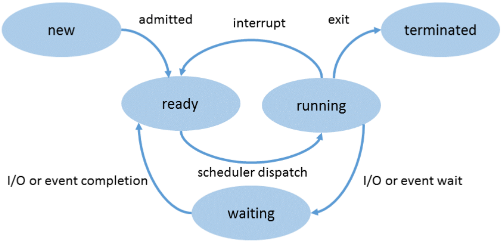
Program vs. Process



- A program is a passive entity such as a file containing a list of instructions stored on disk (executable file)
- A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources
- A program becomes a process when an executable file is loaded into memory. Execution of program started via
  - GUI mouse clicks
  - Command line entry of its name
- One program can be used by several processes
  - Consider multiple users executing the same program

Process States and Diagram

- As a process executes, it changes state, which is defined by the current activity
  - New: The process is being created
  - Running: Instructions are being executed
  - Waiting: The process is waiting for some event to occur (such as I/O completion)
  - Ready: The process is waiting to be assigned to a processor (CPU)
  - Terminated: The process has finished execution



- Could have multiple processes at ready / waiting state
- Only one process at running state at any instant

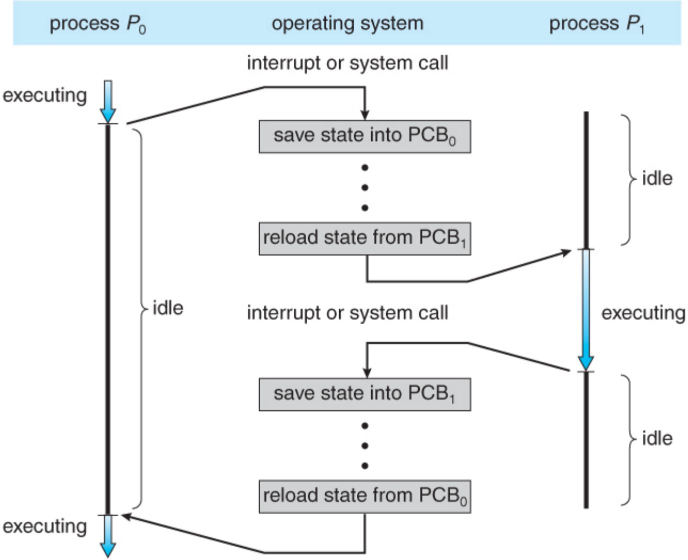
Process Control Block (PCB)

Each process is represented by a process control block in the operating system (also called task control block)

- Process state: running, waiting, ready, halted, and so on
- Process ID: A unique identification number for each process in the OS
- Program counter: location of instruction to next execute
- CPU registers: contents of all process-centric registers
- Memory-management information: memory allocated to the process
- I/O status information: I/O devices allocated to process, list of open files
- CPU scheduling information: priorities, scheduling queue pointers
- Accounting information: CPU used, clock time elapsed since start, time limits

Process state
Process ID (number)
Program counter
Registers
Memory limits
List of open files
...

CPU Switch from Process to Process



Threads

- So far, process has a single thread of execution
  - This single thread of control allows a process to perform only one task at a time
  - If this is the case, a word-processor program cannot simultaneously type in characters and run the spell checker at the same time
  - Most modern OS allows a process to have multiple threads of execution, thus to perform more than one task at a time
  - This can best take advantage of the multicore systems, where multiple threads of one process can run in parallel
- PCB has to be expanded to include information for each thread
  - Multiple locations can execute at once
  - Multiple program counters, one for each thread

More details on Thread will be discussed in the next topic

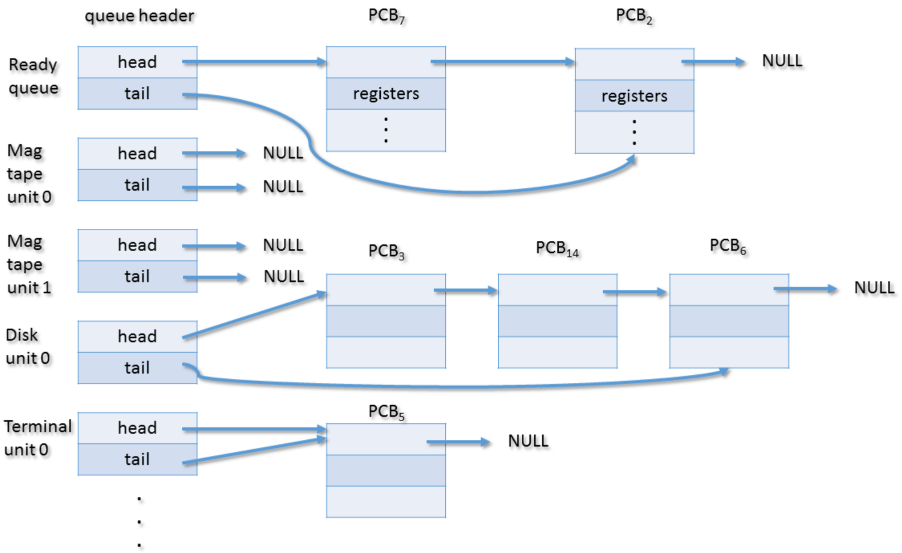
Process Scheduling

- Motivation
- Queues of Processes
- Types of Scheduler
- Context Switch

Motivation

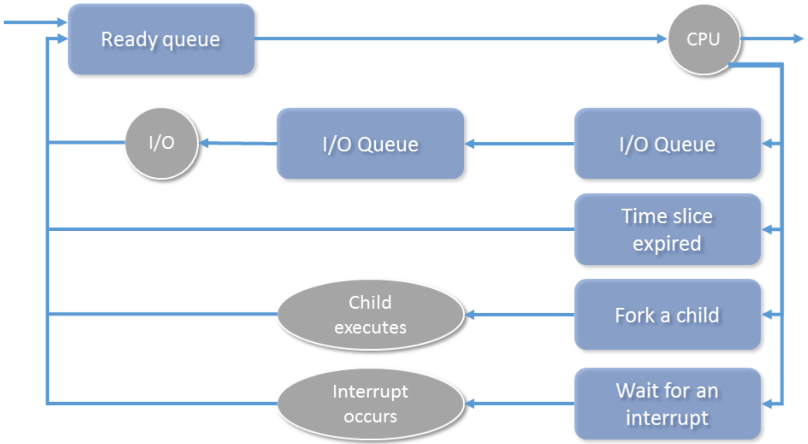
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - Job queue: set of all processes in the system
  - Ready queue: set of all processes residing in main memory, ready and waiting to execute
  - Device queues: set of processes waiting for an I/O device
  - Processes migrate among the various queue

Ready Queue and Various I/O Device Queue



Representation of Process Scheduling

Queuing diagram represents queues, resources, flows

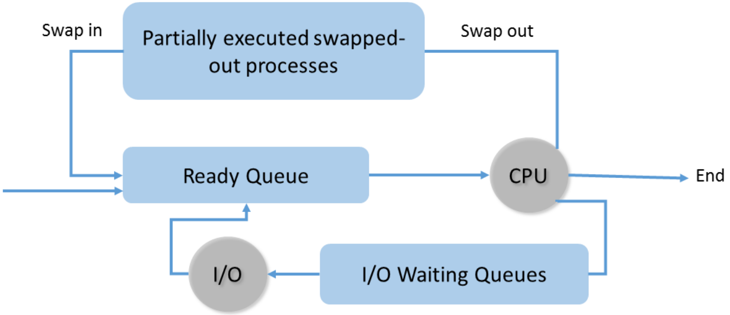


Types of Scheduler

- Long-term scheduler (or job scheduler)
  selects which processes should be brought into the ready queue
  - It is invoked very infrequently (seconds, minutes) $\rightarrow$ (must be slow)
  - It controls the degree of multiprogramming
  - It strives for good process mix, i.e. I/O-bound processes and CPU-bound processes
- Short-term scheduler (or CPU scheduler)
  selects which process should be executed next and allocates CPU
  - It is invoked very frequently (milliseconds) $\rightarrow$ (must be fast)

Types of Scheduler (Cont'd)

- Medium-term scheduler
  - removes a process from the memory, store on disk, bring back later in from disk to continue execution - swapping
  - added if degree of multiple programming needs to be decreased



Context Switch

Context Switch

It is the switching of the CPU from one process or thread to another

- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process via a context switch
- Context of a process represented in PCB
- Context-switch time is overhead; the system does no useful work while switching
  - The more complex the OS and the PCB, the longer the context switch
  - Typical speed is a few milliseconds
- Context-switch times are highly dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU (such as SUN UltraSPARC), multiple contexts loaded at once

Operations on Processes

- Introduction
- Process Creation
- Process Termination
  - Zombie
  - Orphans

Introduction

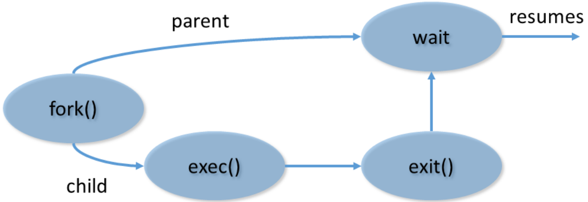
- The processes in most systems can execute concurrently, and they may be created and deleted dynamically
- System must provide mechanisms for
  - process creation
  - process termination
  - ...

Process Creation

- One process can create another process
  - the original process is called the parent
  - the new process is called child
  - the child is an identical copy of the parent (same code, same data) but has a new process ID (pid)
- Resource sharing options
  - Parent and child share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources
- Execution options
  - Parent and children execute concurrently (i.e. in parallel)
  - Parent waits until children terminate

Useful System Calls in UNIX

- fork()
  - If fork() returns a negative value (e.g. -1), the creation of a child process was unsuccessful
    Note: errno is also set to indicate the error
  - In parent process, fork() returns a positive value, which is the process ID of child
  - In child process, fork() returns 0
- execl(), execlp(), execle(), execv(), execvp(), execvpe()
  - Child can overwrite its remaining programs with a new one and start a completely different program
- wait(pid)
  - Parent, if desired can wait until child completes



Example of UNIX Process Creation using fork()

- Suppose the following program executes up to the point of the call to fork() (marked in red)

```
#include <stdio.h>
#include <sys/types.h>

int a = 6;
int main() {
  int b;                        // local variable
  pid_t mypid, childpid;        // process ids
  b = 99;
  printf("Before fork\n");
  childpid = fork();
  mypid = getpid();
  if(childpid==0) {             // child
    a++; b++;
  }
  else wait(childpid);          // parent
  printf("After fork\n");
  printf("me=%d, ", mypid);
  printf("mychild=%d, ", childpid);
  printf("a=%d, b=%d\n", a, b);
  return 0;
}
```

If the call to fork() is executed successfully, UNIX will

- make two identical copies of address space, one for the parent and the other for the child
- Both process will start their execution at the next statement following the fork() system call
- In this case, both processes will start their execution at the assignment statement as shown on the next page

Example of UNIX Process Creation using fork() (Cont'd)

Parent

```
#include <stdio.h>
#include <sys/types.h>

int a = 6;
int main() {
  int b;
  pid_t mypid, childpid;
  b = 99;
  printf("Before fork\n");              // #1
  childpid = fork();                    childid > 0
  mypid = getpid();     // mypid is parent's pid
  if(childpid==0) {                     // false
    a++; b++;                           // SKIPPED
  }
  else wait(childpid);                  // EXECUTED
  printf("After fork\n");               // #6
  printf("me=%d, ", mypid);             // #7
  printf("mychild=%d, ", childpid);     // #8
  printf("a=%d, b=%d\n", a, b);         // #9
  return 0;                             // EXECUTED
}
```

Child

```
#include <stdio.h>
#include <sys/types.h>

int a = 6;
int main() {
  int b;
  pid_t mypid, childpid;
  b = 99;
  printf("Before fork\n");
  childpid = fork();                    childid = 0
  mypid = getpid();     // mypid is child's pid
  if(childpid==0) {                     // true
    a++; b++;                           // EXECUTED
  }
  else wait(childpid);                  // SKIPPED
  printf("After fork\n");               // #2
  printf("me=%d, ", mypid);             // #3
  printf("mychild=%d, ", childpid);     // #4
  printf("a=%d, b=%d\n", a, b);         // #5
  return 0;                             // EXECUTED
}
```

Example of UNIX Process Creation using fork() (Cont'd)

- Output:

```
Before fork
After fork
me=27687, mychild=0, a=7, b=100
After fork
me=27686, mychild=27687, a=6, b=99
```

Another Example of UNIX Process Creation using fork() and execlp()

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

int main() {
  pid_t pid;
  pid = fork();                                    // fork another process
  if(pid < 0) {                                    // error occurred
    fprintf(stderr, "Fork failed");
    exit(-1);
  }
  else if (pid == 0) {                             // child process
    execlp("/bin/ls", "ls", NULL);
  }
  else {                                           // parent process
    // parent will wait for the child to complete
    wait(NULL);
    printf("Child Complete");
    exit(0);
  }
}
```

What does it take to create a process?

- Create new PCB (Inexpensive)
- Setup new page tables for address space (More expensive)
  - More details about page tables later when we talk about memory management
- Copy data from parent process (UNIX fork()) (Very expensive)
  - Semantics of UNIX fork() are that the child process gets a complete copy of the parent memory and I/O state
- Copy I/O state (file handles, etc.) (Medium expensive)

fork(): Parent vs. Child

- Duplicated
  - Address space
  - Global and local variables
  - Current working directory
  - Root directory
  - Process resources
  - Resource limits
- Different
  - PID
  - Running time
  - Running state
  - Return values from fork()

Creating a Separate Process via Windows API

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

int main(VOID)
{
STARTUPINFO si;
PROCESS_INFORMATION pi;

    /* allocate memory */
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    /* create child process */
    if (!CreateProcess(NULL, /* use command line */
     "C:\\WINDOWS\\system32\\mspaint.exe", /* command */
     NULL, /* don't inherit process handle */
     NULL, /* don't inherit thread handle */
     FALSE, /* disable handle inheritance */
     0, /* no creation flags */
     NULL, /* use parent's environment block */
     NULL, /* use parent's existing directory */
     &si,
     &pi))
    {
      fprintf(stderr, "Create Process Failed");
      return -1;
    }
    /* parent will wait for the child to complete */
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Child Complete");

    /* close handles */
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}
```

fork() vs. CreateProcess()

- fork() has the child process inheriting the address space of its parent, while CreateProcess() requires loading a specified program into the address space of the child process at process creation
- fork() is passed no parameters, CreateProcess() expects no fewer than 10 parameters. In the example above, application mspaint.exe is loaded

Process Termination

- Process executes last statement and asks the operating system to delete it (exit())
  - Output data from child to parent (via wait())
  - Process' resources are deallocated by operating system
- Parent may terminate execution of children processes (abort())
  - Child has exceeded its usage of some of the resources that has been allocated
  - Task assigned to child is no longer required
  - If parent is exiting, some operating systems do not allow child to continue if its parent terminates (all children terminated - cascading termination)
- A parent process may wait for the termination of a child process by using wait() system call, returning the pid, so the parent process can tell which of its children has terminated

```
pid_t pid;
int status;
pid = wait(&status);
```

Zombie and Orphans

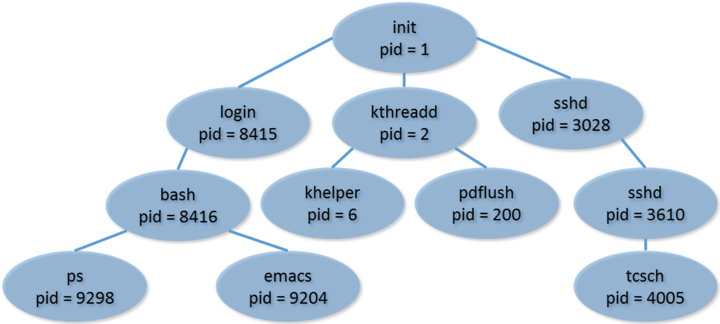
- If no parent waiting, then terminated process is a zombie. Once the parents calls wait(), the process identifier of the zombie process and its entry in the process table are released
- If parent terminated without calling wait(), the child processes are orphans. Linux and UNIX assign the init process as the new parent to orphan processes

Zombie

Orphan

A Tree of Processes in Linux

- A parent process create children processes, which, in turn create other processes, forming a tree of processes



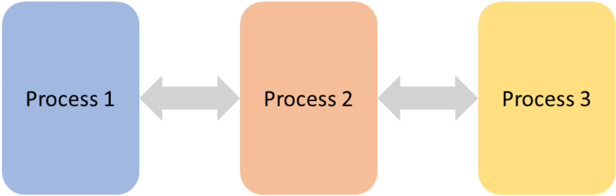
Interprocess Communication (IPC)

- Introduction
- Mechanisms of Multiple Processes Collaboration
- Producer-Consumer / Bounded-Buffer Problem
- Message Passing
  - Direct Communication
  - Indirect Communication
- Synchronization
- Buffering
- Communications in Client-Server Systems
  - Sockets
  - Pipes

Introduction to Interprocess Communication (IPC)

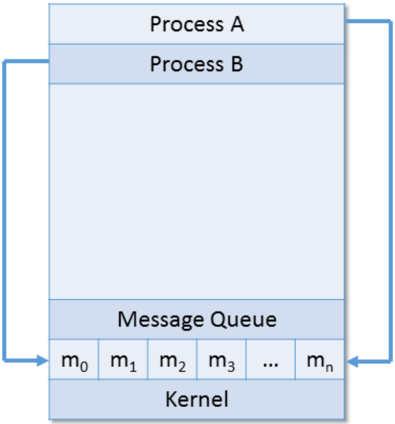
- Processes within a system may be independent or cooperating
- Cooperating process can affect or be affected by other processes, including sharing data
- Reasons for cooperating processes
  - Information sharing, for instance a shared file
  - Computation speedup: subtasks of a task execute in parallel on multicore
  - Modularity: system functions are divided into multiple processes or threads
  - Convenience: users may work on multiple tasks in parallel
- Cooperating processes need an interprocess communication (IPC) mechanism that allow them to exchange data and information
- Two models of IPC, both common in operating systems
  - Shared memory
  - Message passing

Mechanisms of Multiple Processes Collaboration

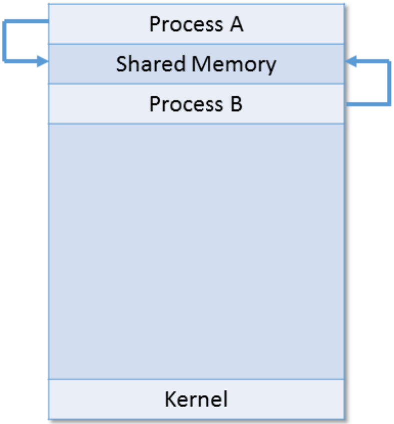


- Need communication mechanisms
  - Different processes, separate address spaces
  - Shared-memory mapping
    - Accomplished by mapping addresses to shared-memory regions
    - System calls such as read() and write() through memory
    - This suffers from cache coherency issues in multicore (with multiple cache)
  - Message passing
    - send() and receive() message
    - Can work across network
    - Better performance in multicore systems

Mechanisms of Multiple Processes Collaboration (Cont'd)



Message Passing



Shared Memory

Producer-Consumer / Bounded-buffer Problem

Producer-consumer problem

Producer-consumer problem (also known as bounded-buffer problem) is a classic example of a multi-process (cooperating processes) synchronization problem. It describes two processes, namely the producer and the consumer, who share a common, fixed-size buffer used as a queue.

- The producer's job is to generate data, put it into the buffer, and start again
- at the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time.

The problem is to make sure that the producer won't try to add data into the buffer if it is full and that the consumer won't try to remove data from an empty buffer.

- Bounded-buffer assumes that there is a fixed buffer size
- Unbounded-buffer places no practical limit on the size of the buffer

Bounded-buffer - Shared-memory solution

```
#define BUFFER_SIZE 10

typedef struct {
   ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;                    // the next free position
int out = 0;                   // the first full position
```

Solution is correct, but can only use BUFFER_SIZE elements

Bounded-buffer - Producer & Consumer

- Producer

```
item next_produced;
while(true) {
  // produce an item in next_produced
  while(((in + 1) % BUFFER_SIZE) == out)          // no space
    ;                                             // do nothing
  buffer[in] = next_produced;
  in = (in + 1) % BUFFER_SIZE;
}
```

- Consumer

```
item next_consumed;
while(true) {
  while(in == out)                                // no data
    ;                                             // do nothing
  next_consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  // consume the item in next_consumed
}
```

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 at least two operations
  - send(message) - message size fixed or variable
  - receive(message)
- If P and Q wish to communicate, they need to
  - establish a communication link between them
  - exchange messages via send/receive
- Implementation of communication link
  - physical (e.g., shared memory, hardware bus)
  - logical (e.g., direct or indirect, synchronous or asynchronous, automatic or explicit buffering)

Implementation Questions

- 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 - Direct Communication

- Each process that wants to communicate must explicitly name the recipient or sender of the communication
  - send(P, message) - send a message to process P
  - receive(Q, message) - receive a message from process Q
- Properties of a communication link
  - A link is established automatically (processes only need to know each other's identity)
  - A link is associated with exactly two processes
  - Between each pair there exists exactly one link
  - The link is usually bi-directional but can be unidirectional

```
while(true) {
  produce an item
  send(B, item)
}
```

```
while(true) {
  receive(A, item)
  consume item
}
```

- Disadvantage:
  - Limited modularity: Changing the name of a process means changing every sender and receiver process to match
  - Need to know process names

Message Passing - Indirect Communication

- Messages are sent to and received from mailboxes (or ports)
  - Mailboxes can be viewed as objects into which messages placed by processes and from which messages can be removed by other processes
  - Each mailbox has a unique ID
  - Processes can communicate only if they have a shared mailbox
  - send(A, message) - send a message to mailbox A
    receive(A, message) - receive a message from mailbox A
- Properties of communication link
  - A communication link is only established between a pair of processes if they have a shared mailbox
  - A link may be associated with more than two processes
  - A pair of processes can communicate via several different communication links (i.e.,mailboxes) if desired
  - A link can be either unidirectional or bidirectional
- Operations
  - Create a new mailbox
  - Send and receive messages through mailbox
  - Destroyed a mailbox
- Allows one-to-many, many-to-one, many-to-many communications

Message Passing - Indirect Communication (Cont'd)

- One-to-many: any of several processes may receive from the mailbox
  - Example: a broadcast of some sort
  - Which of the receives gets the message?
    - arbitrary choice of the scheduling system if many waiting? (e.g., round-robin)
    - only allow one process at a time to wait on a receive
- Many-to-one: many processes sending to one receiving process
  - Example: A server (file server, network server, mail server, etc.) providing service to a collection of processes
  - Receiver can identify the sender form the message header contents
- Many-to-many
  - Example: multiple senders requesting service and a pool of receiving servers offering service (a server farm)

Synchronization

- Message passing may be either blocking or non-blocking
- Blocking is considered synchronous
  - Blocking send: The sending process is blocked until the message is received by the receiving process or by the mailbox
  - Blocking receive: The receiver blocks until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send: The sending process sends the message and resumes its operation
  - Non-blocking receive: The receiver retrieves a valid message or null
- Synchronized communications: Both send and receive are blocking
- Asynchronous communications: Both send and receive are non-blocking

Synchronized Communications

- Properties
  - A rendezvous with effective confirmation of receipt for sender
  - At most one message can be outstanding for any process pair (no buffer space problems), producer-consumer becomes trivial

```
message next_produced;
while(true) {
  // produce an item
  // in next_produced
  send(next_produced);
}
```

```
message next_consumed;
while(true) {
  receive(next_consumed);
  // consume the item
  // in next_consumed
}
```

  - easy to implement, with low overhead
- Disadvantage
  - Sending process might want to continue after its send operation without waiting for confirmation of receipt
  - Receiving process might want to do something else if no message is waiting to be received

Asynchronous Communications

- Properties
  - Messages need to be buffered until they are received
    - Amount of buffer space to allocate can be problematic
    - A process running amok could clog the system with messages if not careful
  - Often very convenient rather than forced to wait, particularly for senders
  - Can increase concurrency
  - Some awkward kernel decisions avoid, e.g., whether to swap a waiting process out to disc or not
  - Receivers can poll for messages, i.e., do a test-receive every so often to see if any messages waiting, but interrupt and signal programming more difficult

Question: How about other combinations?

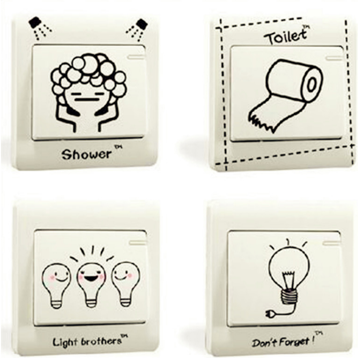
- Non-blocking send + blocking receive
- Blocking send + non-blocking receive

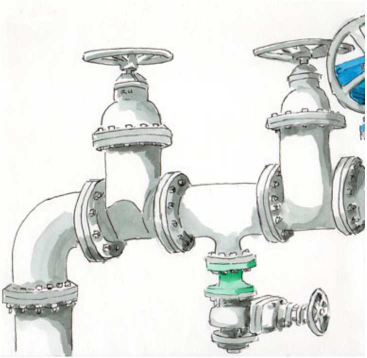
Buffering

- Queue of messages attached to the link (direct or indirect) is implemented in one of three ways
  - Zero capacity (0 messages)
    - 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

Communications in Client-Server Systems

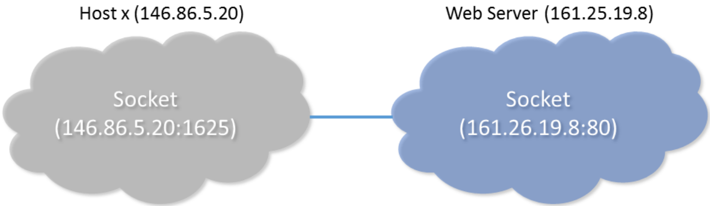
- Typically, communications in client-server systems are done through
  - Sockets
  - Pipes





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 IP address 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 the system on which process is running, i.e. localhost

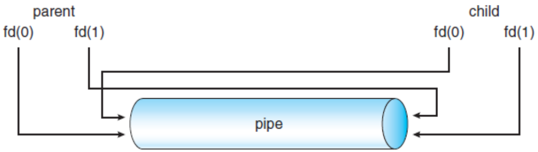


Pipes

- Pipes provide a mechanism by which two processes exchange information and coordinate activities
- It acts as a conduit allowing two processes to communicate
- Pipes were one of the first IPC mechanisms in early UNIX systems
- Four issues must be considered
  - Is communication unidirectional or bidirectional?
  - In the case of two-way communication, is it half or full-duplex
  - Must there exist a relationship (such as parent-child) between the communicating processes?
  - Can the pipes be used over a network or must reside on the same machine?
- Two types
  - Ordinary pipes
  - Named pipes

Ordinary Pipes

- Ordinary pipes allow communication in standard producer-consumer style through system call pipe(int fd[])
  - int pipe(int fd[2]);
    return 0 upon success, -1 upon failure
    fd[0] is open for reading, fd[1] is open for writing
  - Producer writes to one end (the write-end of the pipe), i.e. fd[1]
  - Consumer reads from the other end (the read-end of the pipe), i.e. fd[0]



- Ordinary pipes are therefore unidirectional, UNIX treats a pipe as a special type of file
- Require parent-child relationship between communicating processes on the same machine. Ordinary pipe ceases to exist after the processes have finished communicating and terminated
- Windows calls these anonymous pipes

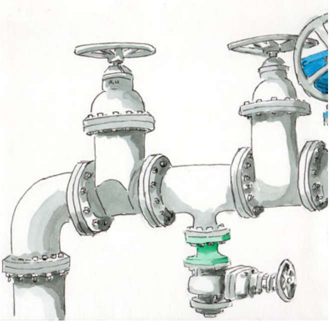
```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>

#define MSGLEN 64

int main(){
    int fd[2];
    pid_t pid;
    int result;

    // create a pipe
    result = pipe(fd);
    if(result < 0) {          // failure
        perror("pipe");
        exit(1);
    }

    // create a child process
    pid = fork();
    if(pid < 0) {             // failure
        perror("fork");
        exit(2);
    }
    if(pid == 0) {            // child process
        char message[MSGLEN];
        while(1) {
            // clear the message
            memset(message, 0, sizeof(message));
            fgets(message, 64, stdin);
            // write the message to the pipe
            write(fd[1], message, strlen(message));
        }
    }
    else {                    // parent process
        char message[MSGLEN];
        while(1) {
            // clear the message
            memset(message, 0, sizeof(message));
            // read message from the pipe
            read(fd[0], message, sizeof(message));
            printf("Message entered: %s\n",message);
        }
    }
    exit(0);
}
```

Input and Output

Welcome to COMP3511 :)
Message entered: Welcome to COMP3511 :)

Named Pipes

- Named pipes are more powerful than ordinary pipes
- Communication is bidirectional
- No parent-child relationship is necessary between the communicating process
- Several processes can use the named pipe for communication
- Provided on both UNIX and Windows systems
- Name pipes continue to exist after communicating processes have finished



That's all!

Any questions?