Lecture 2: Processes
Operating Systems – EDA093/DIT401

Vincenzo Gulisano
vincenzo.gulisano@chalmers.se

UNIVERSITY OF GOTHENBURG

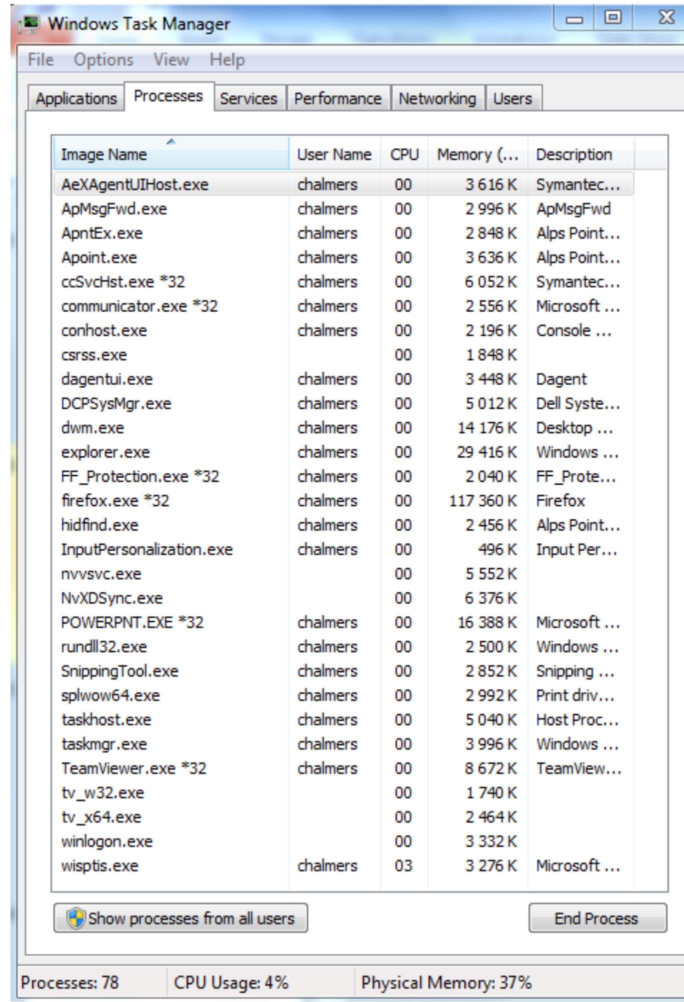
What to read (Main textbook)

- Chapter 2.1

  (extra facultative reading: 3.1-3.4, 3.5.3, 3.6.3 from Silberschatz Operating System Concepts)

Objectives

- To introduce the notion of a process: a program in execution, which forms the basis of all computation
- To describe the various features of processes, including scheduling, creation and termination, and communication
- To explore inter-process communication using shared memory and message passing



AGENDA

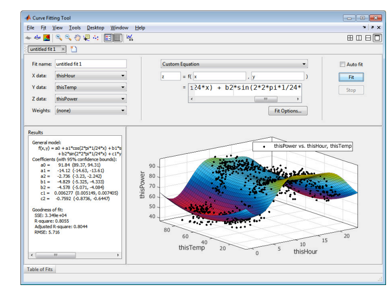
- Processes (Introduction)
- Process Scheduling
- Operations on Processes
- Interprocess Communication (contains self-reading part)

AGENDA

- Processes (Introduction)
- Process Scheduling
- Operations on Processes
- Interprocess Communication (contains self-reading part)

1. We run several programs at the same time

2. One CPU can only run one program at the time



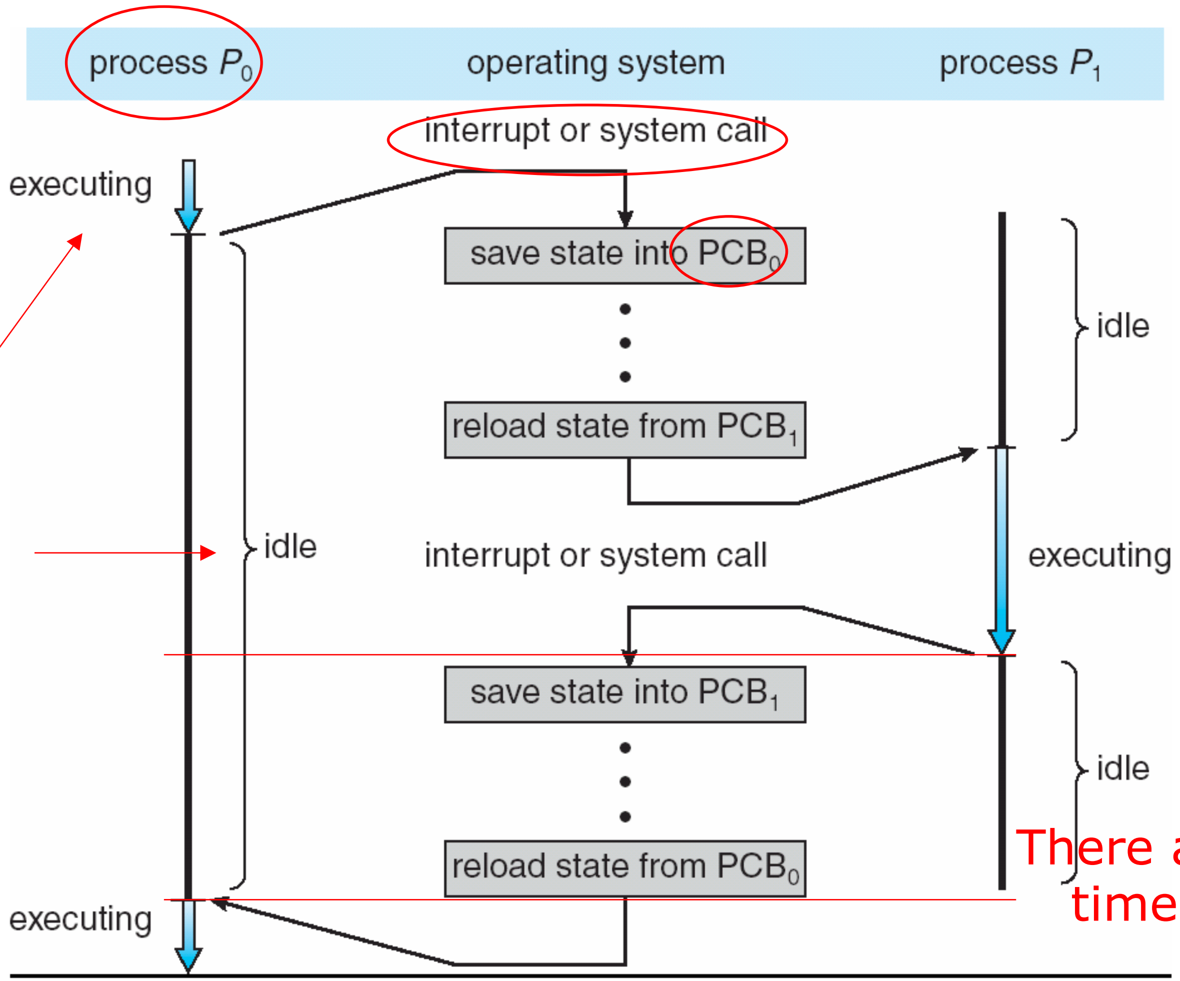
Concurrent vs Parallel execution

1. ~~We run several programs at the same time~~ → 1. We feel several programs run at the same time

2. ~~One CPU can only run one program at the time~~ → 2. Each CPU core can only run one program at the time

(Next lecture)

Process, not program



A process can be in different states, how many?

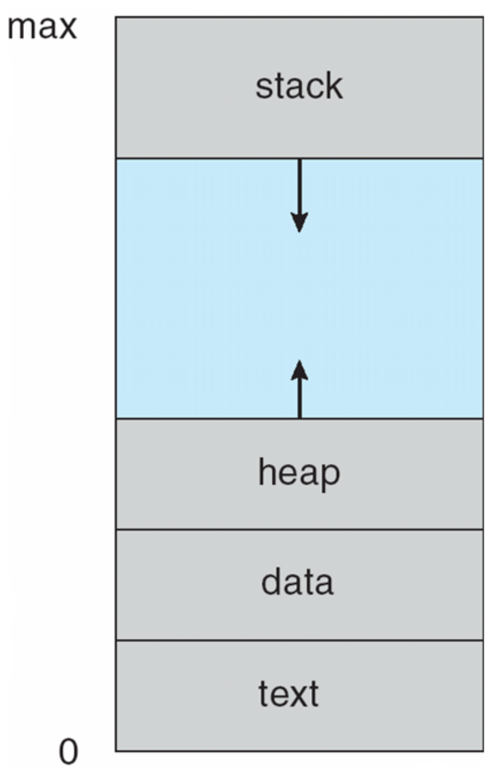
There are periods of time during which no process is running! Overheads should be minimal

Process Concept

- Process (or job, task) – a program in execution; process execution must progress in sequential fashion

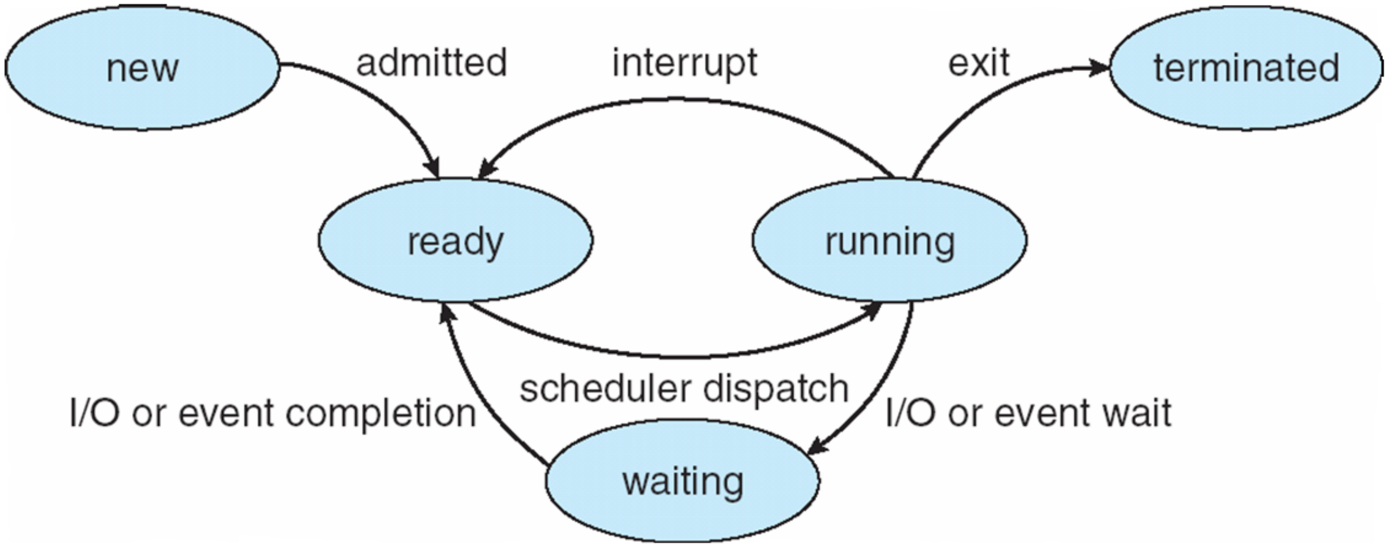
- Program is passive entity stored on disk (executable file), process is active
- Program becomes process when executable file loaded into memory
- Execution of program started via GUI mouse clicks, command line entry of its name, …
- One program can be several processes (consider multiple users executing the same program)
- A process can be the execution environment for other code (e.g., Java Virtual Machine)

Process in Memory



- Multiple parts:
  - The program code, also called text section
  - Data section containing global variables
  - Heap containing memory dynamically allocated during run time
  - Stack containing temporary data
    - Function parameters, return addresses, local variables

Diagram of Process State



- As a process executes, it changes state
  - **new**: The process is being created
  - **ready**: The process is waiting to be assigned to a processor
  - **running**: Instructions are being executed
  - **waiting**: The process is waiting for some event to occur
  - **terminated**: The process has finished execution

Important: only 1 process can be **running** on any processor at any instant.

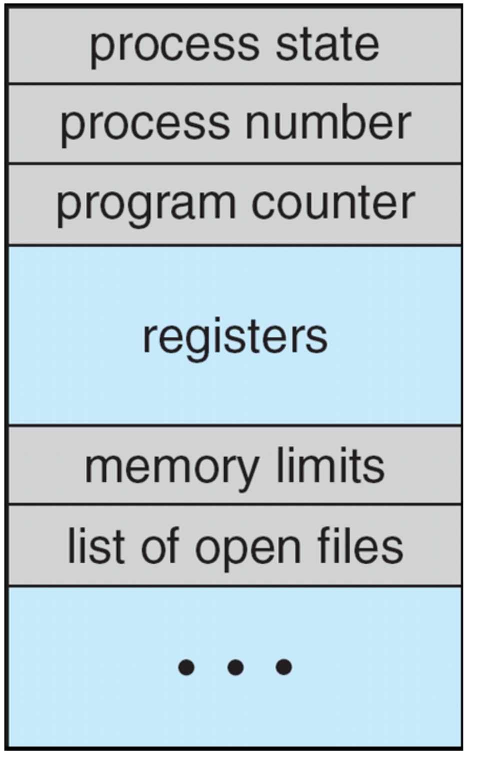
Context Switch

- 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 the 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
- Time dependent on hardware support
  - Some hardware provides multiple sets of registers per CPU ➔ multiple contexts loaded at once

Process Control Block (PCB)

- Information associated with each process in the OS

- Process state: running, waiting, etc.
- Program counter: location of next instruction to execute
- CPU registers: contents of all process-centric registers
- CPU scheduling information: priorities, scheduling queue pointers
- Memory-management information: memory allocated to the process
- Accounting information: CPU used, clock time elapsed since start, time limits
- I/O status information: I/O devices allocated to process, list of open files



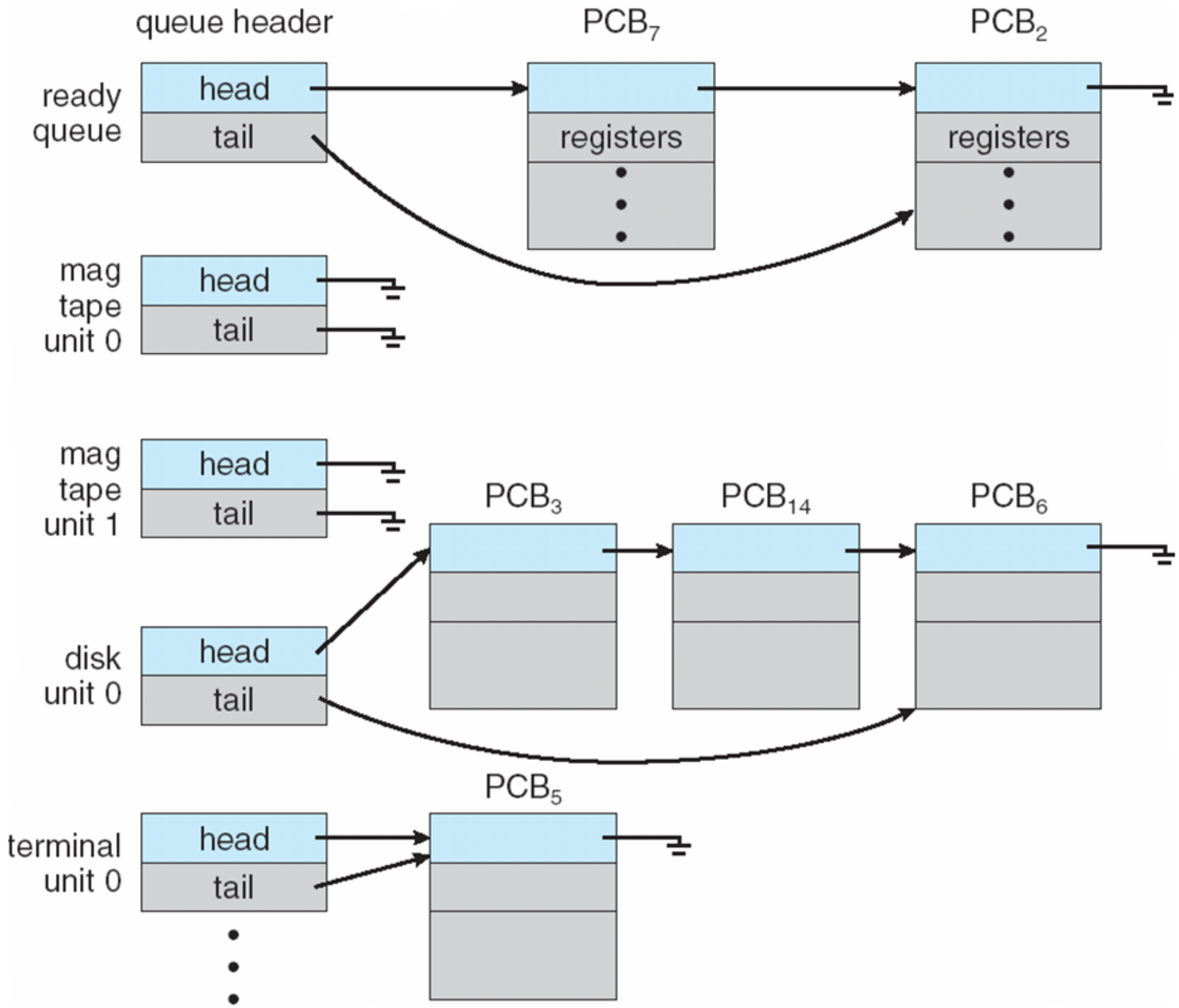
AGENDA

- Processes (Introduction)
- Process Scheduling
- Operations on Processes
- Interprocess Communication (contains self-reading part)

Process Scheduling

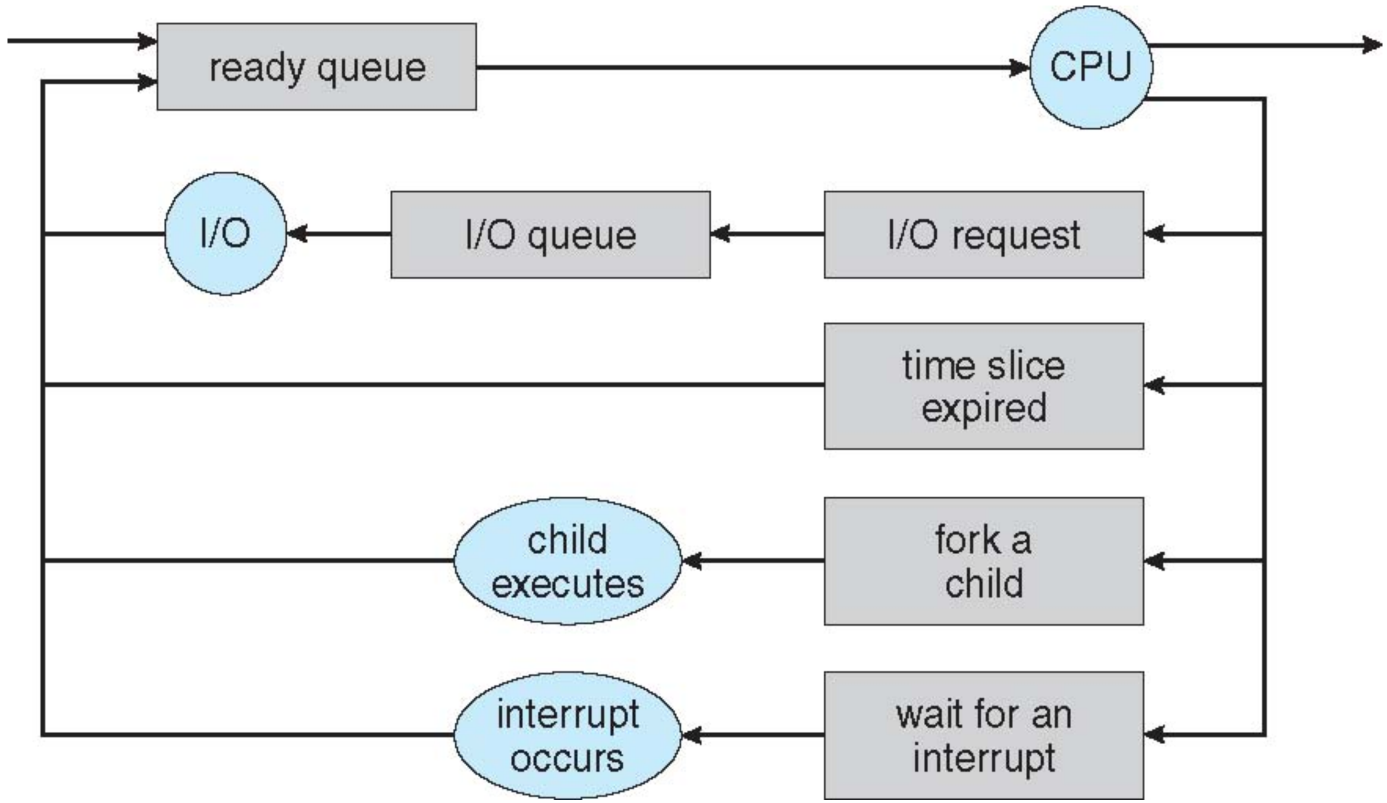
- Maximize CPU use, quickly switch processes onto CPU for time sharing
- Process scheduler selects among available processes for next execution on CPU
- Maintains scheduling queues of processes
  - Job queue – set of all processes in the system
  - Ready queue – set of all processes residing in main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

Ready Queue And Various I/O Device Queues



Representation of Process Scheduling

Queueing diagram represents queues, resources, flows



Schedulers

- Short-term scheduler (or CPU scheduler) – selects which process should be executed next and allocates CPU
  - Sometimes the only scheduler in a system
  - Short-term scheduler is invoked frequently (milliseconds) $\Rightarrow$ (must be fast)

- Long-term scheduler (or job scheduler) – selects which processes should be brought into the ready queue
  - Long-term scheduler is invoked infrequently (seconds, minutes) $\Rightarrow$ (may be slow)
  - The long-term scheduler controls the degree of multiprogramming

AGENDA

- Processes (Introduction)
- Process Scheduling
- Operations on Processes
- Interprocess Communication (contains self-reading part)

Operations on Processes

- System must provide mechanisms for:
  - process creation,
  - process termination,
  - and so on as detailed next

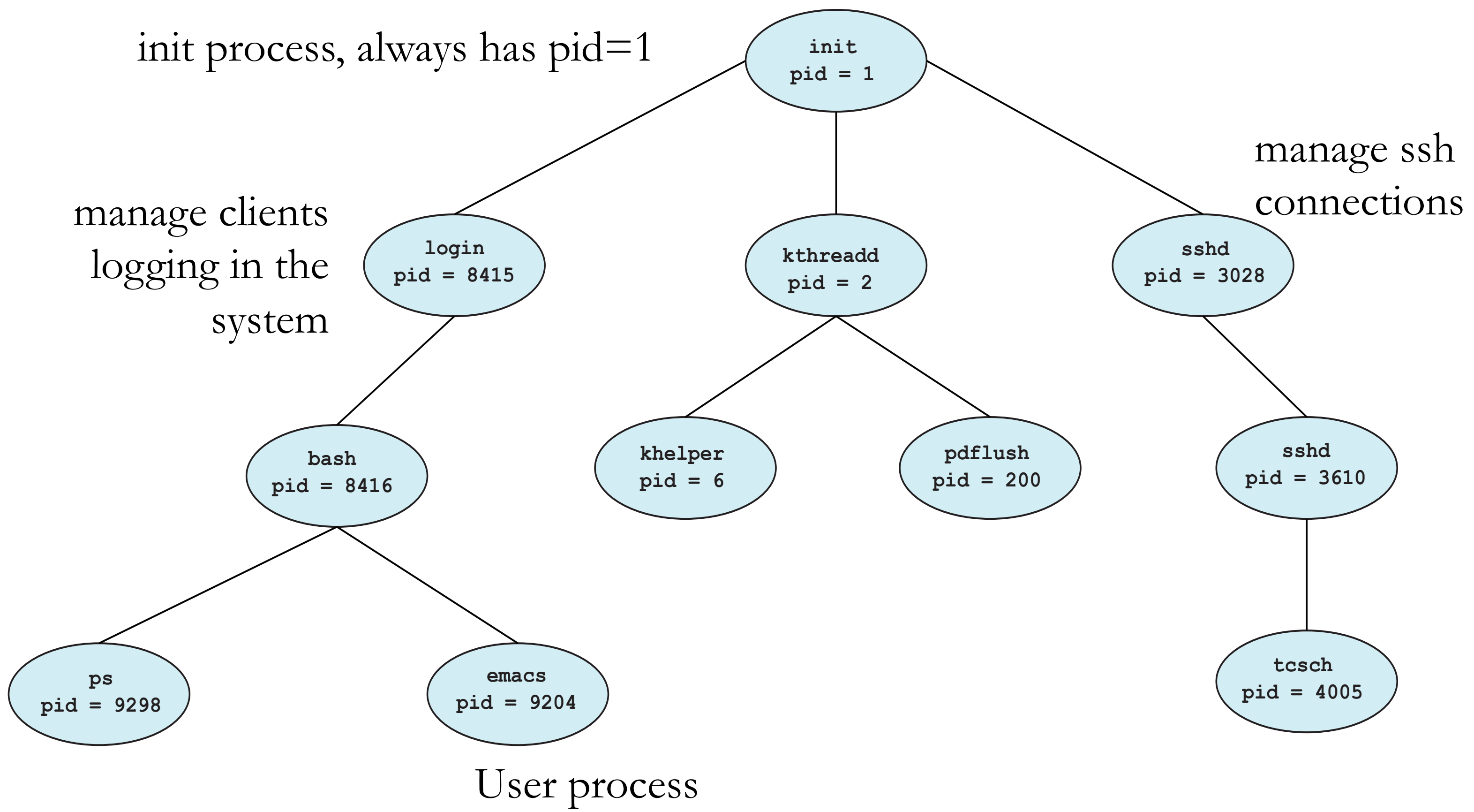
Process Creation

- Parent process create children processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via a process identifier (pid)

- Resource sharing options
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and child share no resources

- Execution options
  - Parent and children execute concurrently
  - Parent waits until children terminate

A Tree of Processes in Linux

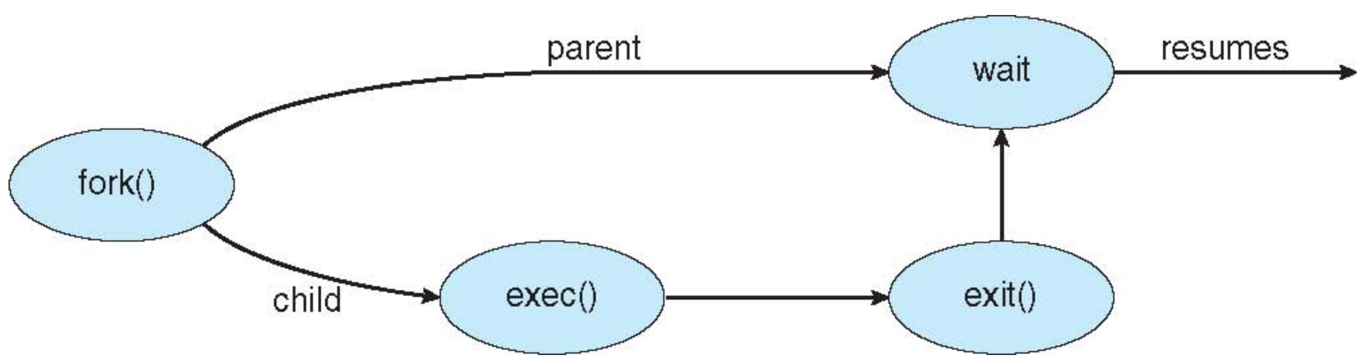


ps –el (UNIX)

```
XXXXXXXXXX:~$ pstree -pl
init(1)-+-NetworkManager(1215)-+-dhclient(3386)
        |                      |-dnsmasq(3390)
        |                      |-{NetworkManager}(1224)
        |                      `-{NetworkManager}(2164)
        |-accounts-daemon(1527)---{accounts-daemon}(1530)
        |-acpid(1360)
        |-apache2(16861)-+-apache2(2458)
        |                |-apache2(2460)
        |                |-apache2(2461)
        |                |-apache2(2462)
        |                |-apache2(2463)
        |                |-apache2(2464)
        |                |-apache2(4079)
        |                |-apache2(4082)
        |                |-apache2(5464)
        |                |-apache2(5465)
        |                `-apache2(5466)
        |-at-spi-bus-laun(3570)-+-dbus-daemon(3576)
        ...
```

Process Creation (Cont.)

- Address space
  - Child duplicate of parent
  - Child has a program loaded into it
- UNIX examples
  - fork() system call creates new process
  - exec() system call used after a fork() to replace the process' memory space with a new program



C Program Forking Separate Process

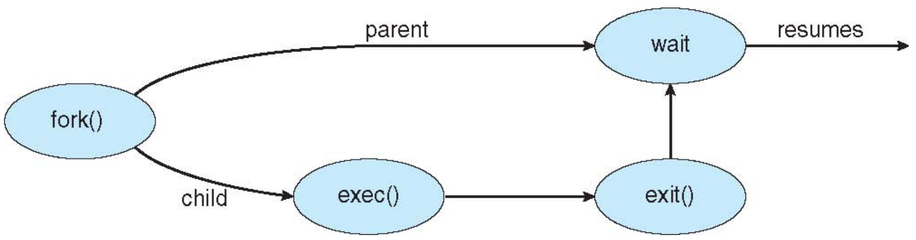
```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
      fprintf(stderr, "Fork Failed");
      return 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");
    }

    return 0;
}
```



Process Termination

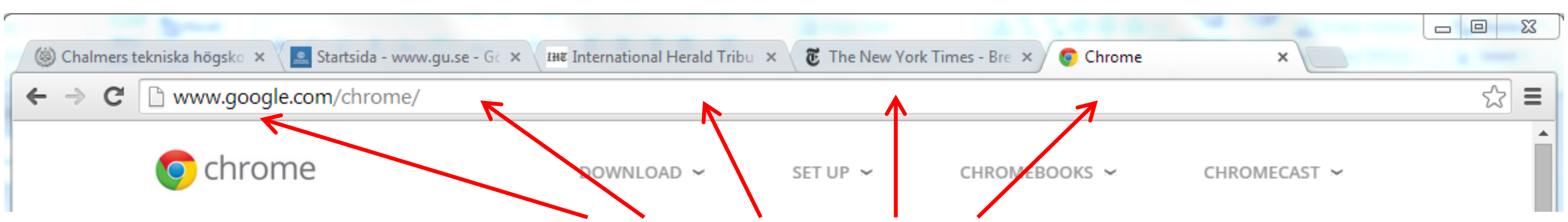
- Process executes last statement and then asks the operating system to delete it using the exit() system call.
  - Returns status data from child to parent (via wait())
  - Process' resources are deallocated by operating system
- Parent may terminate the execution of children processes using the abort() system call. Some reasons for doing so:
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - The parent is exiting and the operating systems does not allow a child to continue if its parent terminates

Process Termination

- Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.
  - cascading termination. All children, grandchildren, etc. are terminated.
  - The termination is initiated by the operating system.
- The parent process may wait for termination of a child process by using the wait()system call. The call returns status information and the pid of the terminated process
- pid = wait(&status);
- If no parent waiting (did not invoke wait()) process is a zombie
- If parent terminated without invoking wait , process is an orphan

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 multi process 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



Each tab represents a process

AGENDA

- Processes (Introduction)
- Process Scheduling
- Operations on Processes
- Interprocess Communication (contains self-reading part)

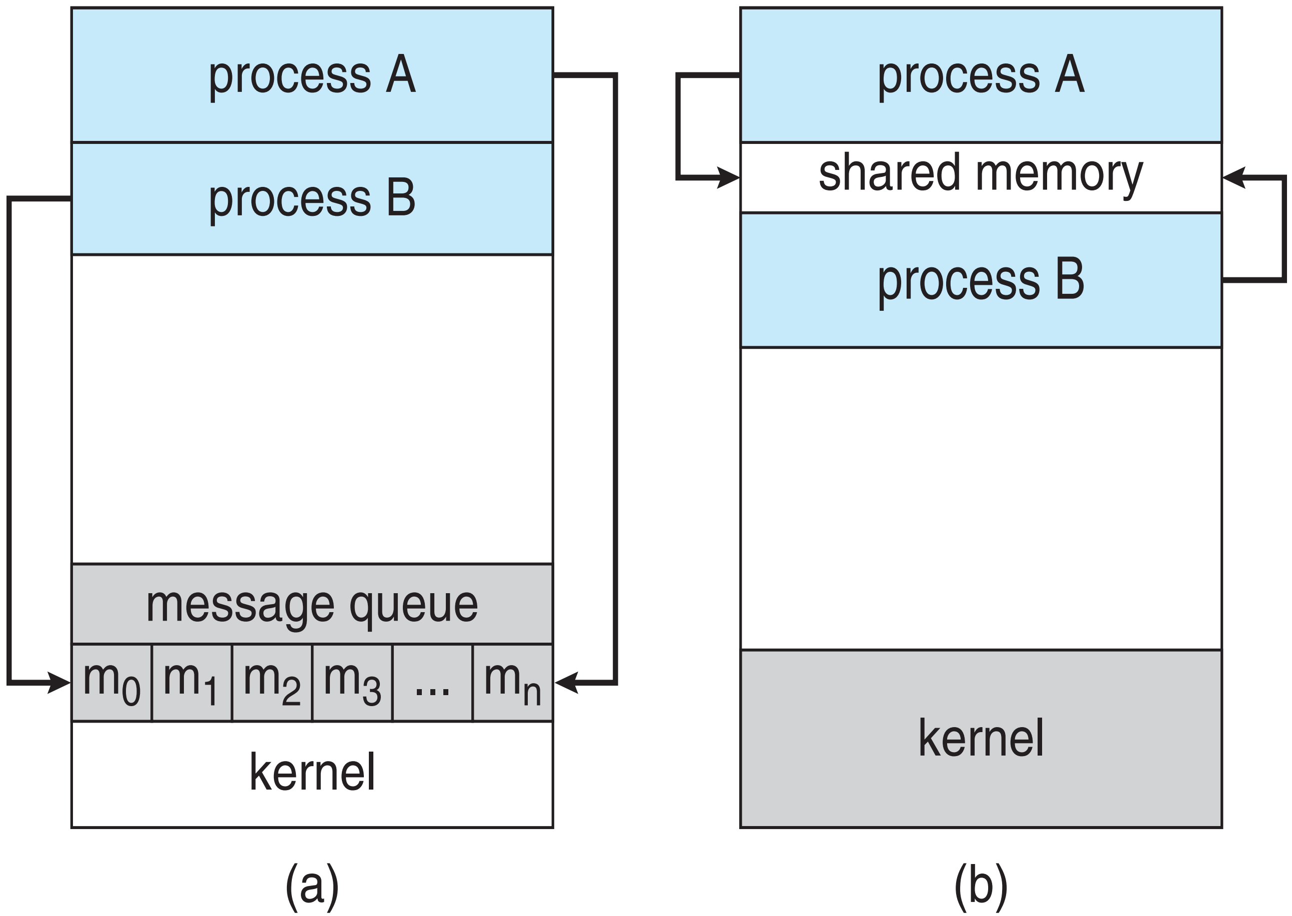
Interprocess Communication

- 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

- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models

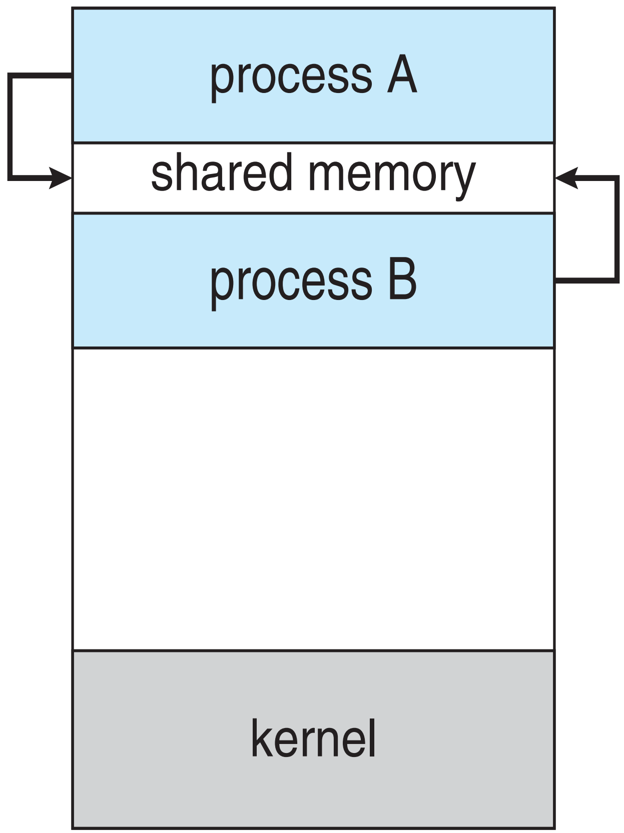
(a) Message passing. (b) shared memory.



Cooperating processes
Producer-Consumer Problem

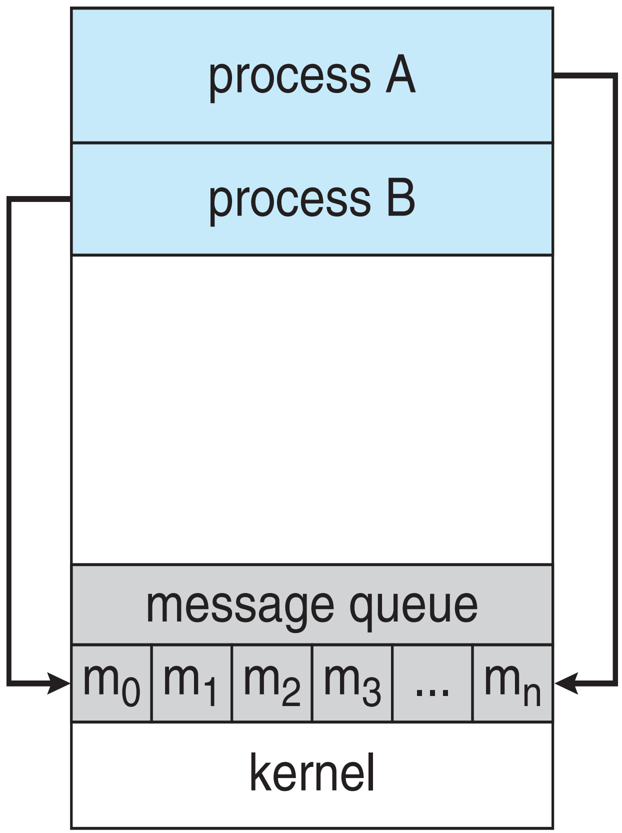
- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
- **unbounded-buffer:** places no practical limit on the size of the buffer
- **bounded-buffer:** assumes that there is a fixed buffer size

Interprocess Communication – Shared Memory



- 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 detail in following lessons

Interprocess Communication – Message Passing



- 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?

Logical implementation – Issues [Self-reading section]

- Naming, Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

Logical implementation - Issues

- Naming, Direct or indirect communication
- Synchronous or asynchronous communication
- 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

- Operations
  - create a new mailbox (port)
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as:
  - send(A, message) – send a message to mailbox A
  - receive(A, message) – receive a message from mailbox A

Logical implementation - Issues

- Naming, Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

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 continues
  - 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

Synchronization (Cont.)

- Producer-consumer becomes trivial for blocking send/receive:

```
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 */
}
```

Logical implementation - Issues

- Naming, Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

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)

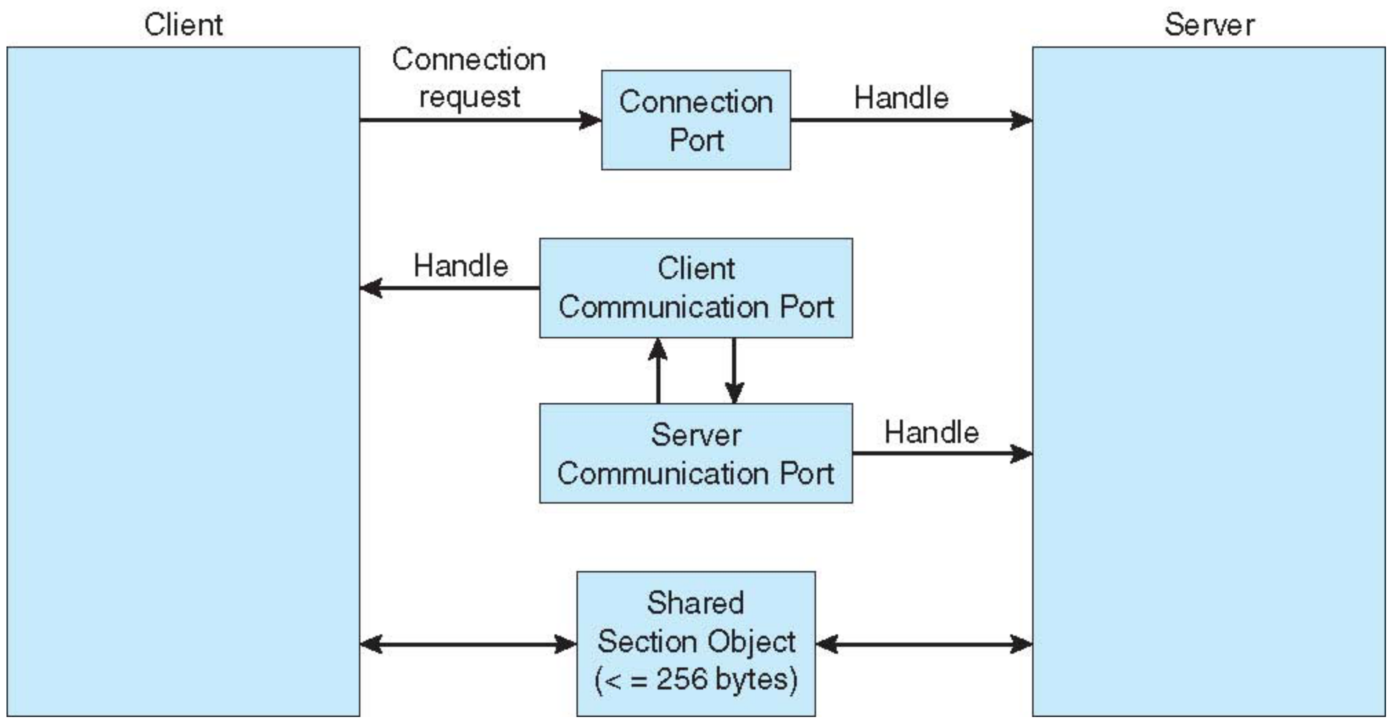
  - 2.Bounded capacity – finite length of n messages
    Sender must wait if link full

  - 3.Unbounded capacity – infinite length
    Sender never waits

Examples of IPC Systems – Windows

- Message-passing centric via advanced local procedure call (LPC) facility
- Only works between processes on the same system
- Uses ports (like mailboxes) to establish and maintain communication channels

Local Procedure Calls in Windows

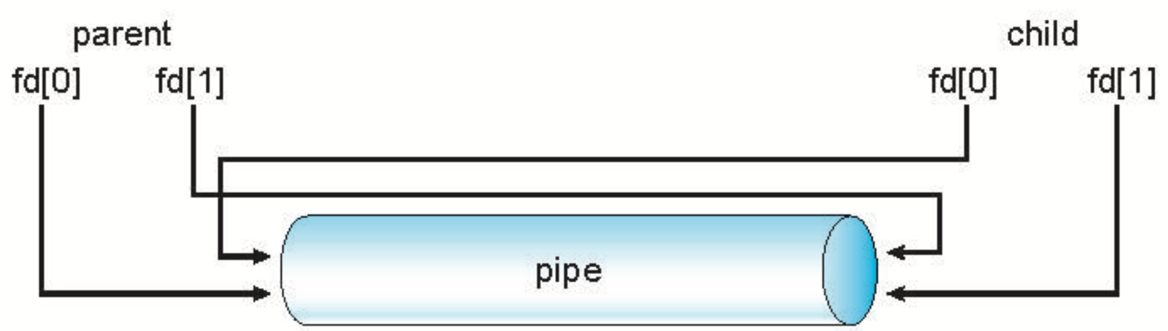


Pipes

- Acts as a conduit allowing two processes to communicate
- 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**
- See Unix and Windows code samples in textbook

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

Pipes in UNIX

- Serve output of one command → input of another command

use a long listing format

pipe

ls -la | grep <pattern>

list directory contents

print lines matching a pattern

do not ignore entries starting with .

Pipes in UNIX

ls -la

```
vincenzo[redacted]:~$ ls -la
total 132
drwxr-xr-x 13 vincenzo vincenzo  4096 Nov  1 15:04
drwxr-xr-x 14 root     root      4096 Oct 28  2013
-rw-------  1 vincenzo vincenzo 52882 Oct 31 22:27
-rw-r--r--  1 vincenzo vincenzo   220 Aug  7  2013
-rw-r--r--  1 vincenzo vincenzo  3785 Jun 23 10:10
drwx------  2 vincenzo vincenzo  4096 Feb 18  2014
drwx------  3 vincenzo vincenzo  4096 Jul 18 13:10
drwxrwxr-x  5 vincenzo vincenzo  4096 Jul 16 08:51
drwxrwxr-x  6 vincenzo vincenzo  4096 Sep  7 19:12
drwxrwxr-x  3 vincenzo vincenzo  4096 Feb 20  2014
drwxrwxr-x 12 vincenzo vincenzo  4096 Oct 24 15:11
-rw-------  1 vincenzo vincenzo    35 Nov  1 15:04
drwxrwxr-x 17 vincenzo vincenzo  4096 Jul 21 10:30
drwxrwxr-x  3 vincenzo vincenzo  4096 Mar 26  2014
drwxrwxr-x 14 vincenzo vincenzo  4096 Apr 27  2014
-rw-r--r--  1 vincenzo vincenzo   675 Aug  7  2013
drwx------  2 vincenzo vincenzo  4096 Feb 20  2014
drwxrwxr-x  3 vincenzo vincenzo  4096 Jul 15 09:02
-rw-------  1 root     root       737 Apr  8  2014
-rw-------  1 vincenzo vincenzo    51 Jun 25 08:58
```

ls -la | grep Apr

```
vincenzo[redacted]:~$ ls -la | grep Apr
drwxrwxr-x 14 vincenzo vincenzo  4096 Apr 27  2014
-rw-------  1 root     root       737 Apr  8  2014
```

Ordinary Pipes - example

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

#define BUFFER_SIZE 25
#define READ_END    0
#define WRITE_END   1
```

Ordinary Pipes - example

```
int main(void)
{
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];
  pid_t pid;
  int fd[2];

  /* create the pipe */
  if (pipe(fd) == -1) {
        fprintf(stderr,"Pipe failed");
        return 1;
  }

  /* now fork a child process */
  pid = fork();
```

```
if (pid < 0) {
         fprintf(stderr, "Fork failed");
         return 1;
  }

  if (pid > 0) {  /* parent process */
         /* close the unused end of the pipe */
         close(fd[READ_END]);
         /* write to the pipe */
         write(fd[WRITE_END], write_msg, strlen(write_msg)+1);
         /* close the write end of the pipe */
         close(fd[WRITE_END]);
  }
  else { /* child process */
         /* close the unused end of the pipe */
         close(fd[WRITE_END]);
         /* read from the pipe */
         read(fd[READ_END], read_msg, BUFFER_SIZE);
         printf("child read %s\n",read_msg);
         /* close the write end of the pipe */
         close(fd[READ_END]);
  }
  return 0;
}
```

Thank you for your attention!

Questions?

Please evaluate the lecture!

https://forms.gle/eH96qJEpatDEQBBz7