DAT103
Datamaskiner og operativsystemer

Processes

Chapter 3 in [B1]

Violet.Ka.I.Pun@hvl.no

Høgskulen på Vestlandet

What do we cover?

- What a process is
- Characteristics of processes
  - Scheduling
  - Creation
  - Termination
  - Communication
- Communication with shared memory and messages
- Communication in client-server systems

Process concept (1)

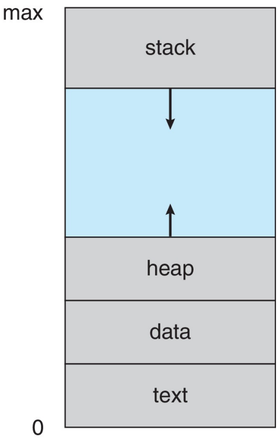
- OS runs many different programs
  - Batch jobs
  - Time shared programs or tasks
- The terms jobs and processes are used interchangeably

Program is a passive entity on the disk, while process is active

- A program becomes a process when it is loaded into the memory
- A **process** is a program in execution
- Process execution is sequential

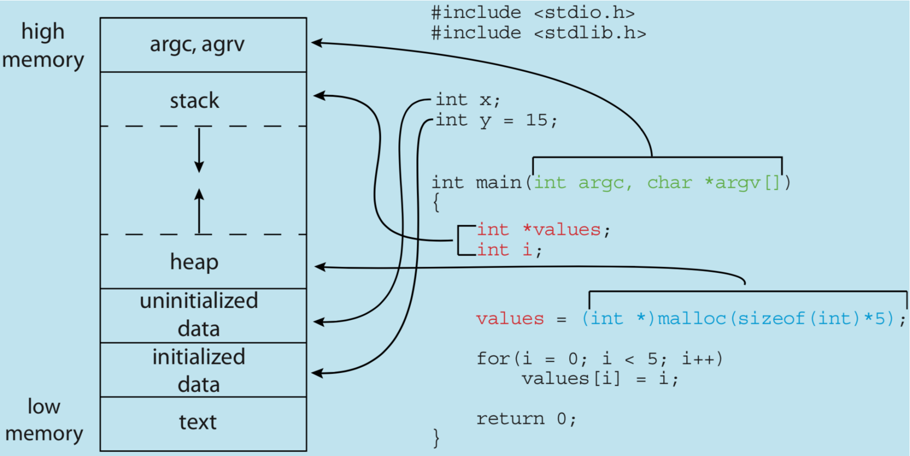
A process contain several parts

- **Program code**: called text section
- **Current activity**: program counter and content of processor's registers
- **Process stack**: temporary data (parameters, return address, local variables)
- **Data section**: global variables
- **Heap**: dynamically located memory during runtime



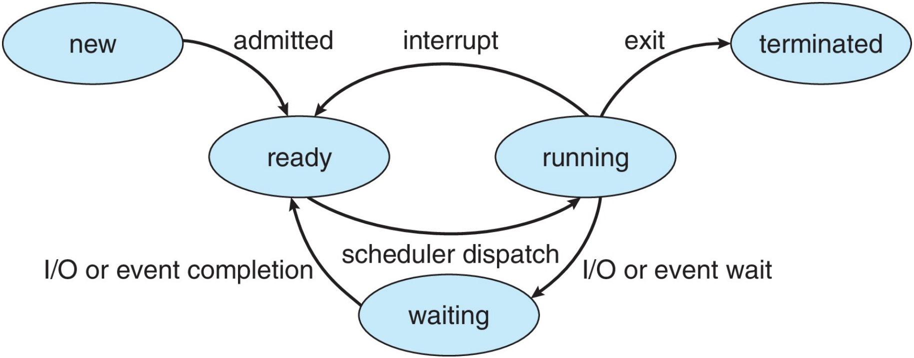
- Execution starts via mouse-click and command line
- A process itself can be an execution environment for other code

Example: Memory of a process for a C program



Process state

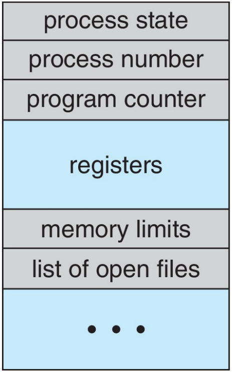
- A process can change its state during its execution
- The state of a process is defined partly by the current activity of that process
- A process may be in one of these states
  - New
  - Running
  - Waiting
  - Ready
  - Terminated
- Only one process can be running on any processor at anytime


new
admitted
interrupt
exit
terminated
ready
running
scheduler dispatch
I/O or event completion
I/O or event wait
waiting

Process control block (PCB)

Or, task control block

- Represents a process in OS
- Contains information associated with the process
- Includes
  - Process state
  - Program counter
  - CPU registers
  - CPU-scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information



Threads

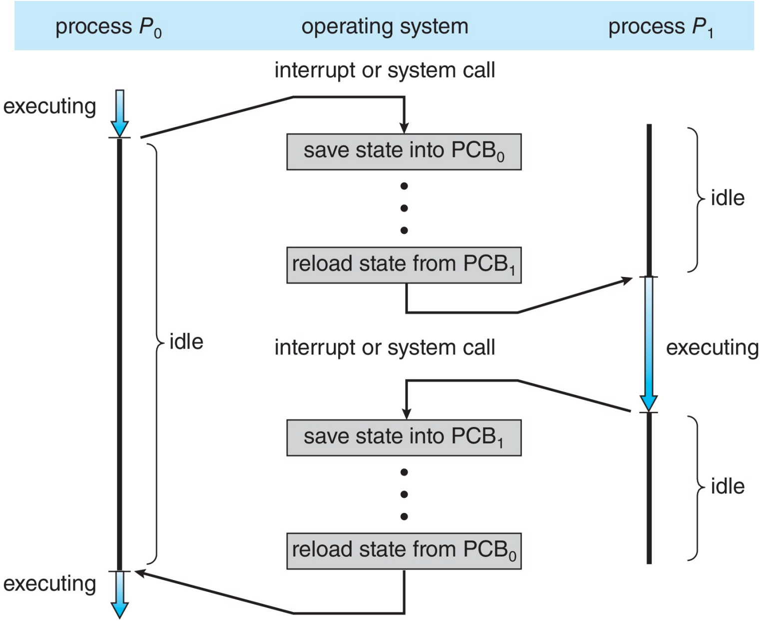
- Each process in the previous slides are assumed to have only one thread
- With multiple threads in a process
  - Each thread has a different program counter
  - Each program counter must be saved in PCB
- Must also include thread information in PCB

Thread Control Block (TCB)

- Contains information about:
  - Thread Identifier
  - CPU registers, including stack pointer, program counter, etc.
  - Thread state
  - Pointer to the Process control block (PCB) of the process to which the thread belongs

More details about threads come in Chapter 4

Switching CPU from process to process



Context switch

- When CPU switches process, the system must save the state of the current process and load the state of the new one

  $\rightarrow$ Need **context switch**

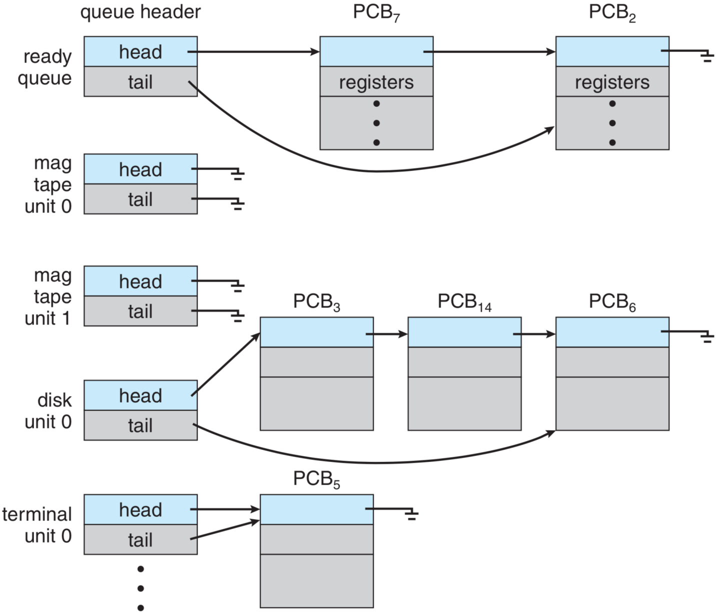
- Context is represented in PCB
- Time for context switch creates overhead
  - The system does not do useful work while switching
  - Switching speed varies from machine to machine
    - Factors may include: the memory speed, the number of registers that must be copied, ...
    - Typical speed: a few milliseconds instructions
  - The more complex the OS and the PCB
    $\rightarrow$ the longer the context switch

- Time taken also depends on hardware
  - Some hardware allow loading multiple contexts at once

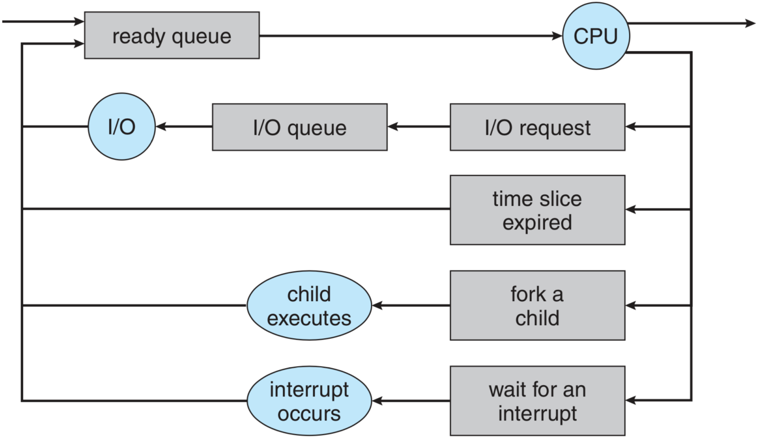
Process scheduling

- Maximise CPU utilisation by switching CPU among processes
- Process scheduler selects one out of all available processes for execution on CPU
- All other processes have to stay in some queues
  - job queue: all processes in the system
  - ready queue: processes in main memory, ready and wait to execute – usually stored as a linked list
  - device queue: processes waiting for I/O device
    - each device has its own device queue

Example: ready queue and device queues



Queueing-diagram representation of process scheduling



circles: resources

Schedulers (1)

- To select processes from various queues appropriately
- **Long-term scheduler** (job scheduler)
  - Select processes for disk to loads into memory
  - Control the **degree of multiprogramming**
    i.e., the number of processes in the main memory
  - Selection is perform less frequently and can be slow
- **Short-term scheduler** (CPU scheduler)
  - Select a process from the ready queue and allocate the CPU
  - Selection is perform frequently and needs to be fast

Schedulers (2)

Long-term scheduler

- Needs to select a good mix of I/O-bound and CPU-bound processes

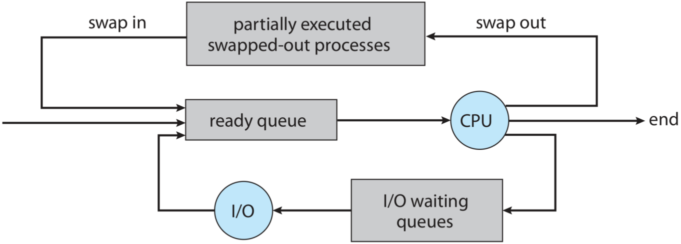
Process can be described as either

- I/O bound, or
  - More time for I/O and computations
- CPU bound
  - Mostly computations, infrequent I/O requests

Mid-term scheduler

- Can remove a process from memory/CPU
  $\rightarrow$ reduce the degree of multiprogramming
- Can reintroduce the process into memory
- Called **swapping**[1]

- Are used to improve the process mix; or
- To free up memory due to e.g., a change in memory requirements that has overcommitted available memory



[1] Discussed later in Chapter 8

Multitasking in mobile systems

- Some mobile systems (e.g., early version of iOS) allow only one process to run at one time, while others suspended
- **iOS** currently allows:
  - Single foreground process: the one that is open and on the display
  - Multiple background processes: runs in memory and not on the display, and with limits
  - Background processes are limited to
    – single, short task, receiving notification of events and specific long-running tasks like audio player
- **Android** has fewer limits on background processes

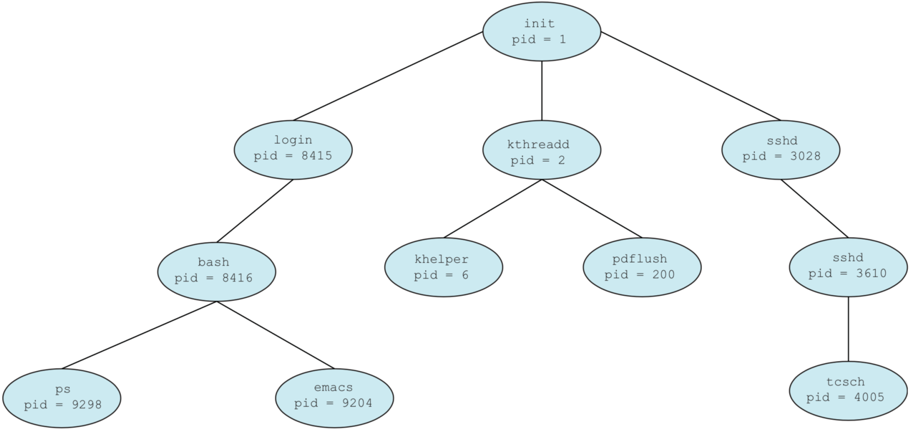
Operations on processes – Creation

- A parent process creates child processes
- Identify and mange a process with a unique process identifier (pid)

Relationship between parent and child processes

- Resource sharing
  - Parent and children share all resources, or
  - Children share subset of parent's resources, or
  - Parent and children do not share resources
- Execution
  - Parent and children run simultaneously, or
  - Parent waits until children are finished
- Address space
  - Child process is a duplicate of the parent process, or
  - Child process has a new program loaded into it

A typical process tree for Linux



Process creation – Unix (example)

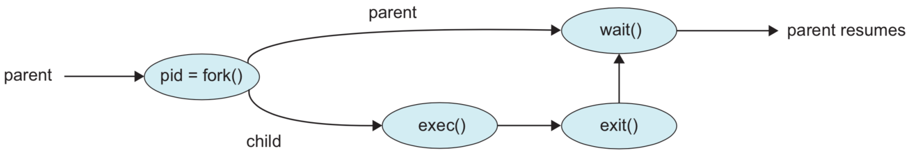
- `fork()` system call creates new process
  - Child process has a copy of the address space of the parent process
- `exec()` system call used after a `fork()`
  - To replace the process' memory space with a new program
- Parent process may call `wait()` to move itself off the ready queue until the child terminates

Forking Separate Process in C

```
#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;
}
```

Operations on processes – Termination

- Process executes last statement, and then asks OS to delete it (`exit()`)
  - Returns status data from child to parent (via `wait()`)
  - Process' resources are deallocated by OS
- Parent may terminate the execution of children processes (`abort()`)
- Why?
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - If the parent terminates, the child also terminates (depends on OS)



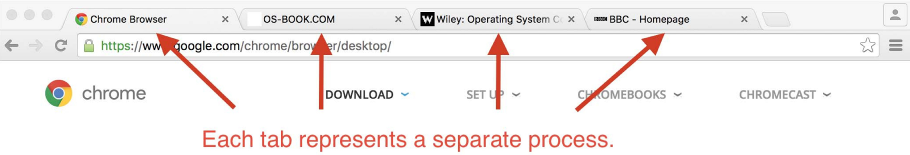
Creating a Separate Process via Windows API

```
#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);
}
```

Multiprocess architecture – Chrome (example)

- Many web browsers ran as single process (some still do)
- If one web site causes trouble, entire browser can hang or crash

- Can use multiprocess architecture to tackle this, e.g., **Chrome**
- 3 different types of processes
  - Browser: only one, created when Chrome starts
    – User interface, disk and network, etc.
  - Renderer: rendering webpages; generally one for each tab
    – Run in a sandbox
  - Plug-in: one for each type of plug-in (such as Flash or QuickTime) in use



Communications between processes

- Processes can be either independent or cooperating
- Independent processes
  - Cannot affect or be affected by the other processes
  - Does not share data with any other process
- Cooperating processes
  - Can affect or be affected by the other processes
  - Shares data with other processes
- Advantages of cooperation
  - Information sharing
  - Computation speed up
  - Modularity
  - Convenience

Communications between processes

- Cooperating processes need interprocess communication (IPC)
- Two models of IPC
  - **Shared memory**
    - A region of memory is established for sharing data
    - Information is exchanged by reading and writing to the shared memory
  - **Message passing**
    - Communication by exchanging messages

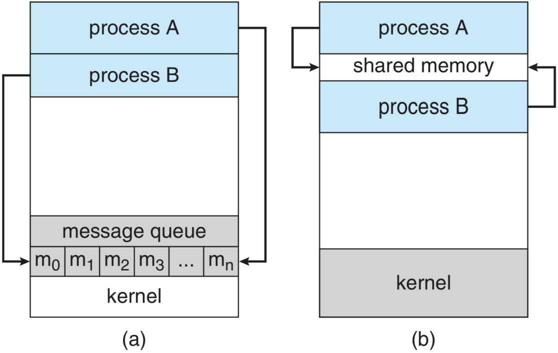


Figure 3.12 Communications models. (a) Message passing. (b) Shared memory.

A common paradigm for cooperating processes

- A producer process produces information that is consumed by a consumer process

Shared-memory solution:

→ Buffer

- Bounded: fixed buffer size
  - Empty buffer: consumer must wait
  - Full buffer: producer must wait
- Unbounded: no limit on the buffer size
  - Empty buffer: consumer must wait
  - Producer can always produce

Example: bounded buffer

```
#define BUFFER_SIZE 10

typedef struct {
   ...
}item;

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

Producer and consumer problem – Shared memory (2)

One possible implementation

Producer

```
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;
}
```

- next_produced is a local variable

Consumer

```
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 */
}
```

- This allows only at most `BUFFER_SIZE - 1` items in the buffer
- How do we have `BUFFER_SIZE` items?

- Provides a mechanism for process communicating and synchronising
- No shared memory
  Useful in a distributed environment
- Provides at least two operations
  - `send(message)`
  - `receive(message)`
- Message size can be either fixed or variable

Message passing

- If processes P and Q want to communicate, they need to:
  - Establish a communication link between them, then
  - Exchange messages via send/receive operations
- We focus on logical implementation, not physical one (like bus or shared memory), possible methods:
  - Direct or indirect communication
  - Synchronous or asynchronous communication
  - Automatic or explicit buffering
- Implementation issues one needs to consider:
  - 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

- Processes must name each other explicitly
  - `send (P, message)` – send a message to P
  - `receive(Q, message)` – receive a message from Q
  - The above is called symmetry addressing
- For asymmetry addressing, only the sender names recipient
  - For recipient: use `receive(`*id*`, message)`
- Properties of communication link:
  - Link is established automatically
  - A link associate to one and only one pair of communicating processes
  - Exactly one link between each pair
  - The link can be unidirectional, but usually bi-directional

Message passing – Indirect communication (1)

- Messages are sent to and received from mailboxes, or ports.
- Each mailbox has a unique identifier
- Messages can be placed or removed from the mailbox by processes
- Two processes can communicate only if they have a shared mailbox
- `send()` and `receive()` are defined as
  - `send(`A`, message)` – send a message to mailbox A
  - `receive(`A`, message)` – receive a message from mailbox A
- Properties of communication link:
  - Link can only be established if processes share a mailbox
  - A link can associate to many processes
  - Each pair of processes can communicate via a number of different mailboxes
  - The link can be unidirectional, but usually bi-directional

Message passing – Indirect communication (2)

Example: assume P_1, P_2 and P_3 share the mailbox A

- P_1 sends a message to mailbox A
- P_2 and P_3 execute `receive()`
- Who will receive the message?
- Some possible answers:
  - Allow a link to associate with at most two processes
  - Allow at most one process to execute `receive()` at one time
  - Let the system choose randomly the recipient and notify the sender who it is

Message passing – Indirect communication (3)

A mailbox can be owned by either a process or OS

If a mailbox is owned by a **process**

- Owner: can only receive message through the mailbox
- User: can only send message to the mailbox
- If the owner process terminates, the mailbox will disappear

If a mailbox is owned by the **OS**

- The mailbox is not attached to any process
- OS provides a mechanism to processes to:
  - Create a new mailbox
  - Send and receive messages through a mailbox
  - Delete a mailbox
- Owner is the creating process by default
- Ownership can be passed to other processes $\Rightarrow$ multiple receivers

Message passing – Synchronisation

Message passing can be either

- Blocking = Synchronous
  - **send**: sender is blocked until the message is received, either by receiving processes or mailbox
  - **receive**: receiver is blocked until a message is available
- Nonblocking = Asynchronous
  - **send**: sender resumes operation after sending a message
  - **receive**: receiver retrieves either a valid message or a null

Producer and consumer problem – Synchronisation

- When both `send()` and `receive()` are blocking
  → a rendezvous between the sender and the receiver
  → producer and consumer problem become trivial

Producer

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

  send(next_produced);  /* send produced_element */
}
```

Consumer

```
message next_consumed;
while (true) {
  receive(next_consumed); /* receive produced element */

  /* consume the item in next_consumed */
}
```

A message queue is required for messages

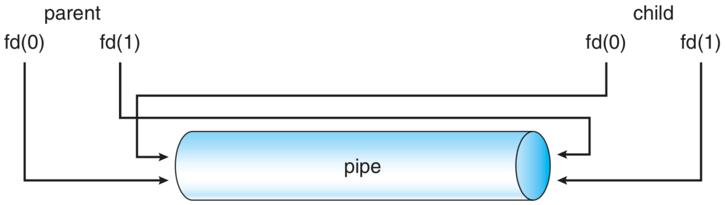
- Zero capacity: sender must wait for receiver (rendezvous)
- Bounded capacity: sender must wait if it is full
- Unbounded capacity: sender never waits

Ordinary pipes

- Communication in ordinary producer-consumer fashion
  - Producer: writes to one end of the pipe (write end)
  - Receiver: reads from the other end (read end)
- Unidirectional
- **Two** pipes are needed for **bi**directional communication

Ordinary pipe between parent and child in UNIX

- An ordinary pipe cannot be accessed from outside the process that created it
  - It is typical a parent process creates a pipe to communicate with its child process
- A pipe can be seen a special type of file
- In UNIX, a pipe that is accessed through the int `fd[]` file descriptors
- `fd[0]` is the read-end of the pipe, and `fd[1]` is the write-end



Named pipes

- More powerful than ordinary pipe
- Bidirectional
- Require no parent-child relationship
- One pipe can be used by several processes for communication

Communications in client-server systems

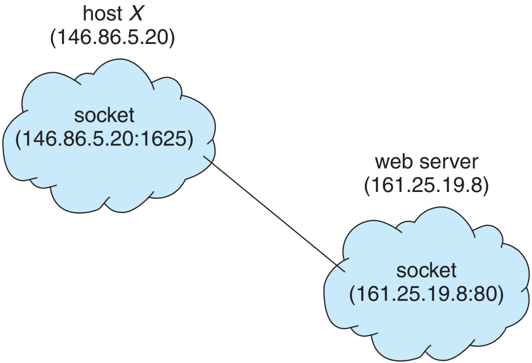
- Both shared memory and message passing can be used
- Also pipes
- Other strategies:
  - Sockets
  - Remote procedure calls (RPC)

Sockets

- Are defined as an endpoint for communication
- Use client-server architecture
- Two processes use a pair of sockets to communicate over a network
- **Socket**: a concatenation of IP address and port

  Port: A number included at start of message packet to differentiate network services on a host

  Socket example: 161.25.19.8 : 1625
- Ports under 1024 is well-known (or defined)



Read the socket example in Java in the textbook
(date server in Figure 3.27 & date client in Figure 3.28)

Remote procedure call (RPC) (1)

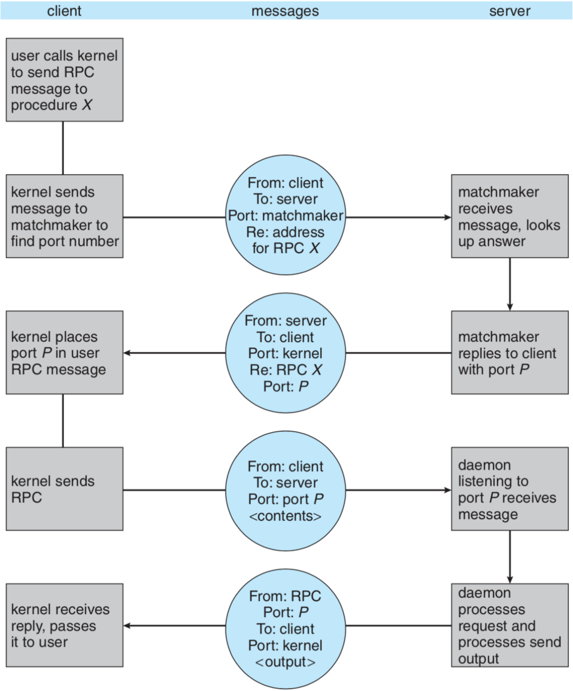
- Abstracts procedure calls between processes on networked systems
- Also uses ports to differentiate services
- Each message is addressed to an RPC daemon listening to a port on the remote system
- Each message contains
  - identifier specifying the function to execute, and
  - the parameters to pass to that function
- RPC system uses a stub on the client side to hide the details that enable the communications
- Typically, each procedure has a stub,
- When the client invokes a remote procedure
  1. The RPC system calls the appropriate stub, and passes it the parameters for the remote procedure
  2. The stub locates the port on the server and arranges the parameters

Remote procedure call (RPC) (2)

- Problem: Data representation can be different in the remote system
- Solution: interface definition language (IDL)

- Remote communication has higher potential to fail than local
  - Messages can be duplicated and executed more than once
  - One solution: OS has to ensure each message is executed at most once, or exactly once

- OS typically provides a matchmaker service to connect client and server


client
messages
server
user calls kernel to send RPC message to procedure X
kernel sends message to matchmaker to find port number
From: client To: server Port: matchmaker Re: address for RPC X
matchmaker receives message, looks up answer
kernel places port P in user RPC message
From: server To: client Port: kernel Re: RPC X Port: P
matchmaker replies to client with port P
kernel sends RPC
From: client To: server Port: port P <contents>
daemon listening to port P receives message
kernel receives reply, passes it to user
From: RPC Port: P To: client Port: kernel <output>
daemon processes request and processes send output