Processes

Tran, Van Hoai

Faculty of Computer Science & Engineering
HCMC University of Technology

E-mail: hoai@hcmut.edu.vn
(partly based on slides of Le Thanh Van)

Outline

Outline

What is a process ?

The textbook

A process is a program in execution

Oxford dictionary

A process is a series of actions or steps taken in order to achieve a particular end

What is a process ?

The textbook

A process is a program in execution

Oxford dictionary

A process is a series of actions or steps taken in order to achieve a particular end

- process execution must progress in sequential fashion
- job and process are used interchangeably

Process vs. Program

- Process is not the same as “program”

Process vs. Program

- Process is not the same as "program"
  - Program is a passive executable code on disk; process is an active entity

Process vs. Program

- Process is not the same as "program"
  - Program is a passive executable code on disk; process is an active entity
  - A same program can be executed into multiple processes (normally by mutiple users)

Process vs. Program

- Process is not the same as "program"
  - Program is a passive executable code on disk; process is an active entity
  - A same program can be executed into multiple processes (normally by mutiple users)
  - Components of a process
    - program counter
    - stack
    - data section

Process vs. Program

- Process is not the same as "program"
  - Program is a passive executable code on disk; process is an active entity
  - A same program can be executed into multiple processes (normally by mutiple users)
  - Components of a process
    - program counter
    - stack
    - data section
- User and OS processes
  - jobs (batch system)
  - tasks (time-shared system)
  - process (generic)

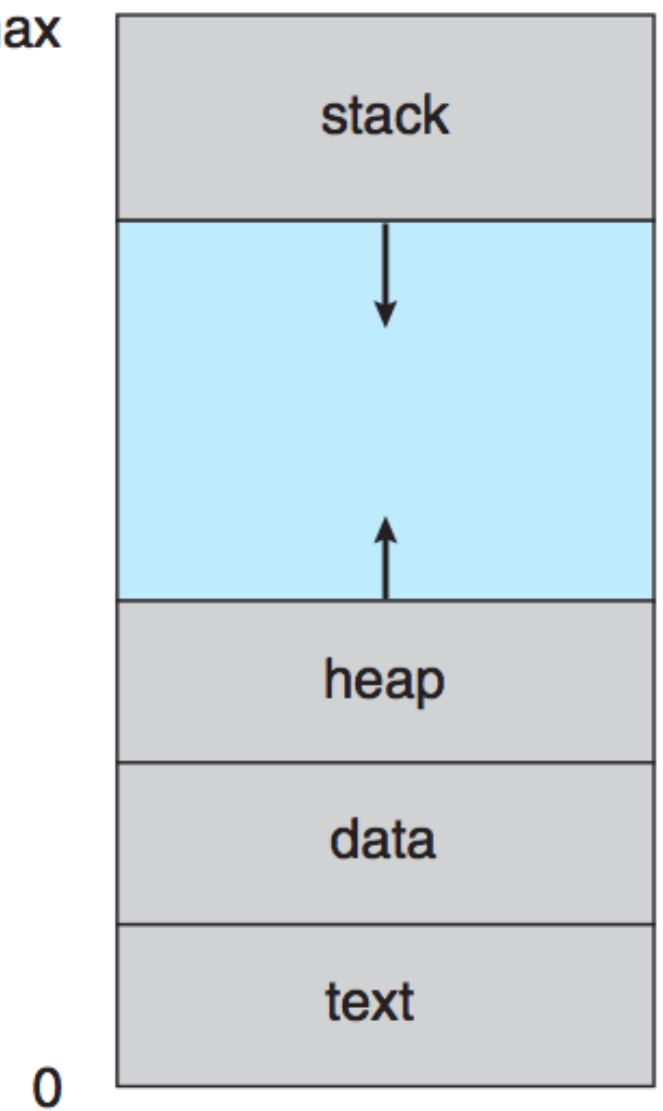
Process vs. Program

- Process is not the same as "program"
  - Program is a passive executable code on disk; process is an active entity
  - A same program can be executed into multiple processes (normally by mutiple users)
  - Components of a process
    - program counter
    - stack
    - data section
- User and OS processes
  - jobs (batch system)
  - tasks (time-shared system)
  - process (generic)

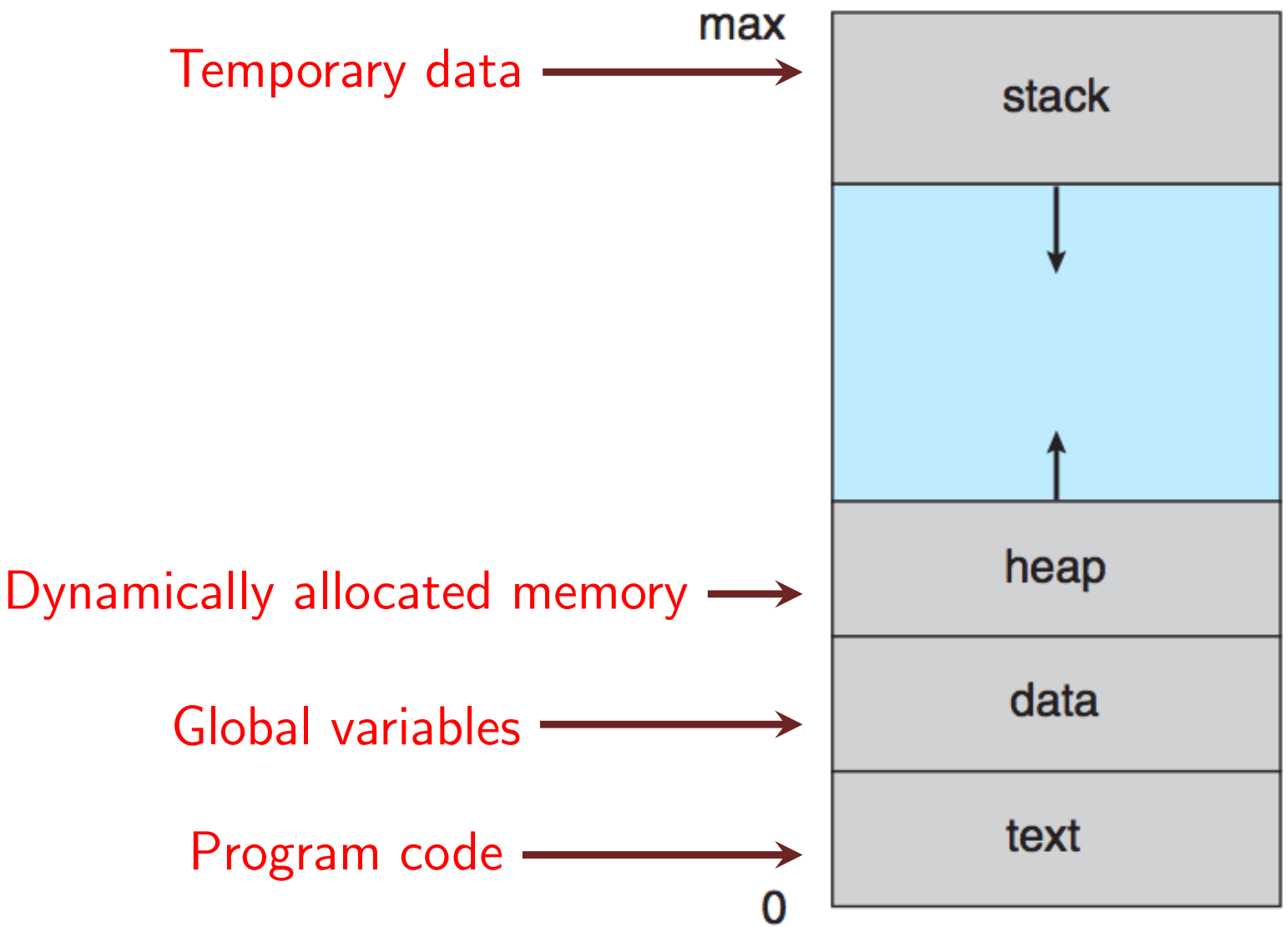
J. Brisendine *(writer)*

Life is a process, not a thing

Process in memory

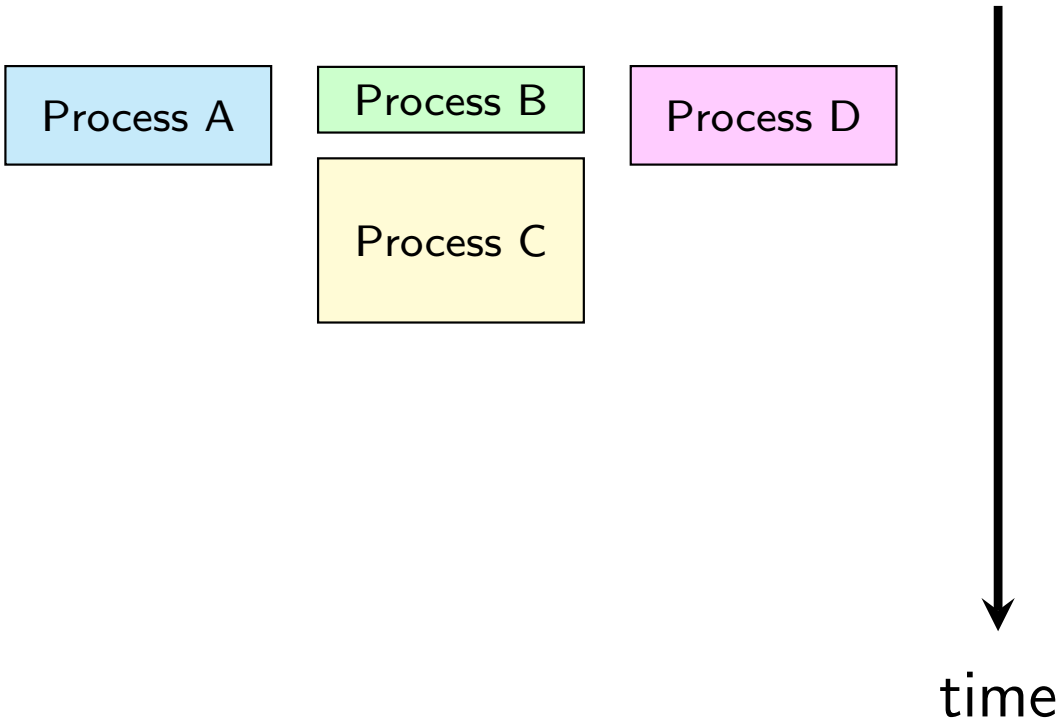


Process in memory



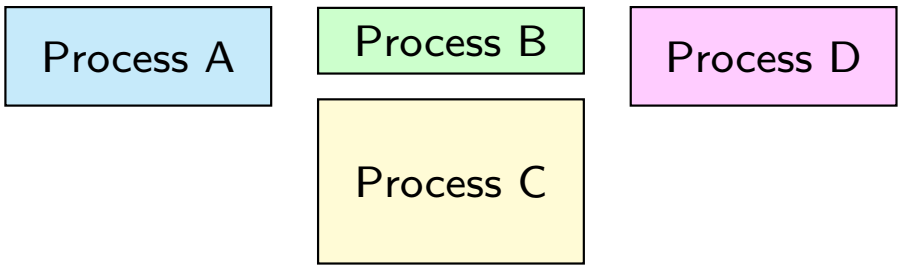
Process in execution

Conceptual model of process execution

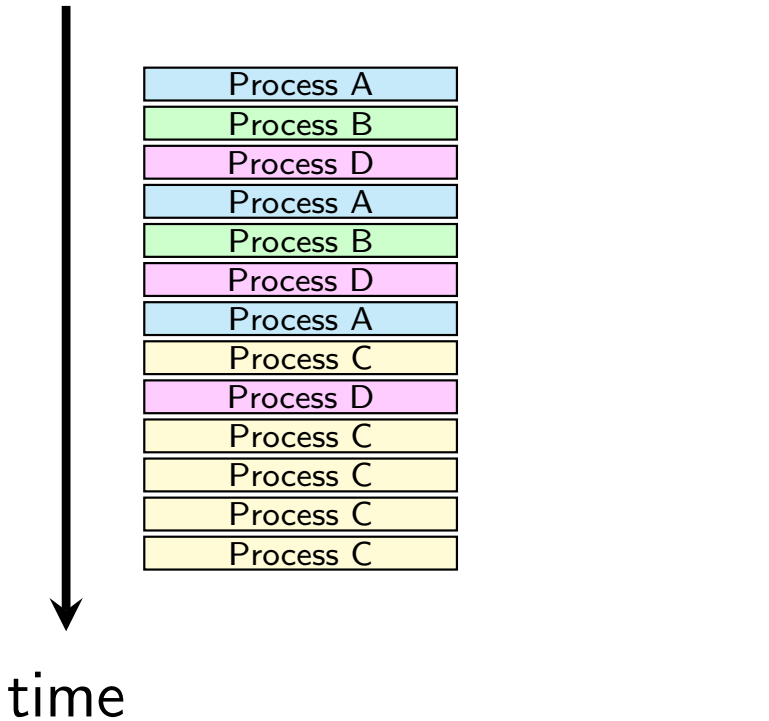


Process in execution

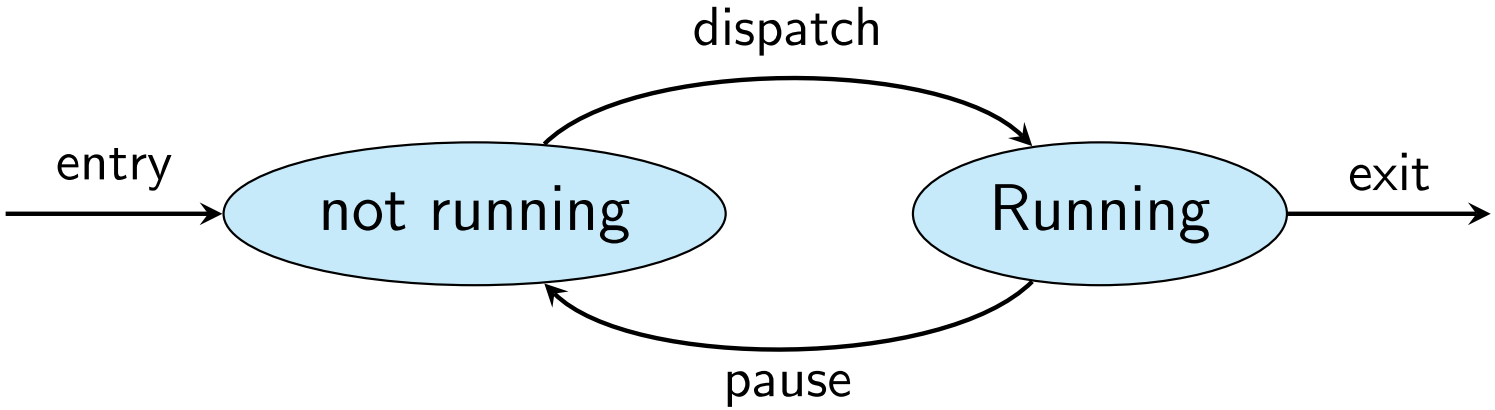
Conceptual model of process execution



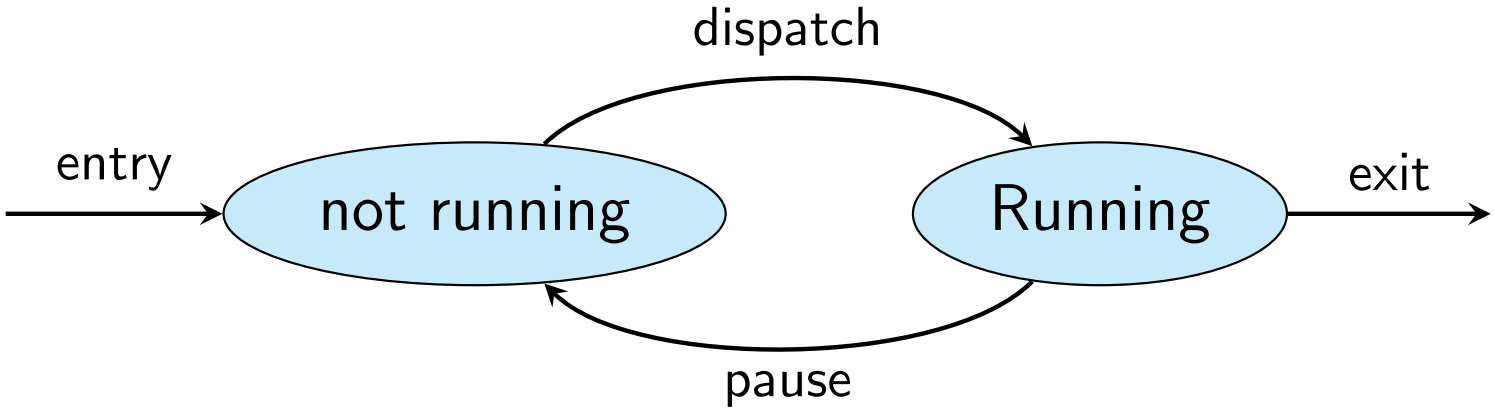
Actual interleaved execution of the processes



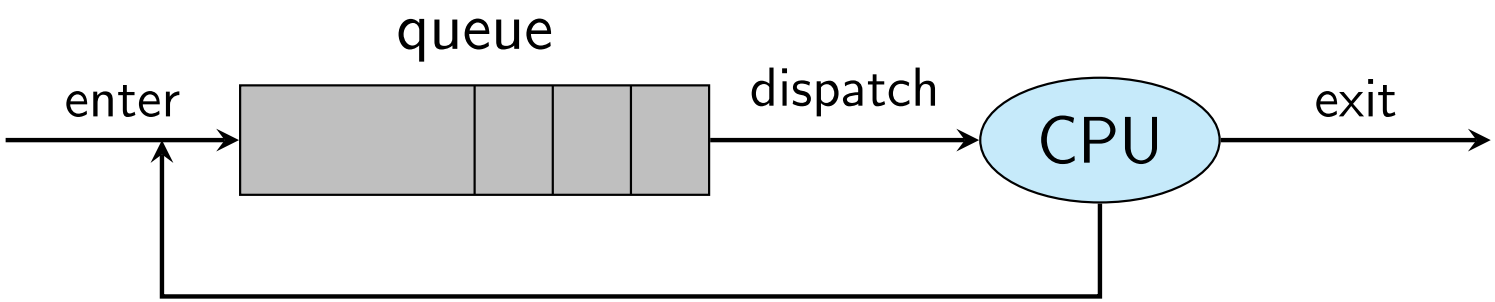
- A process is either "running" or "not running"



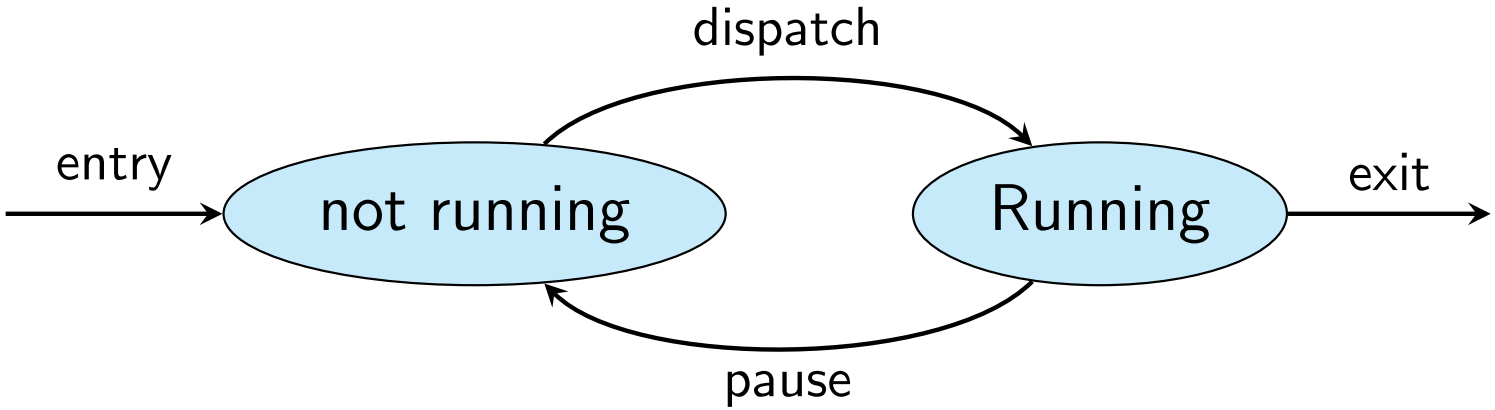
- A process is either "running" or "not running"



- Queueing diagram



- A process is either "running" or "not running"



- Queueing diagram

queue

Weakness

2-state model cannot deal with I/O operations.

Process state

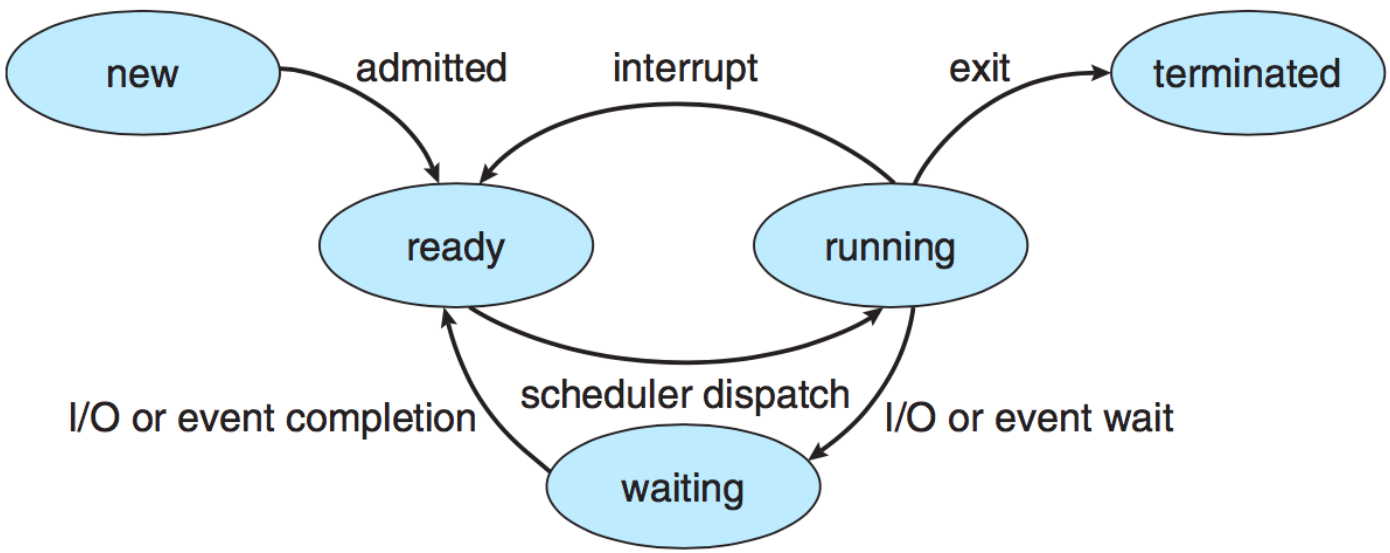
As a process executes, it changes state

- new: the process is being created
- running: its instructions are being executed
- waiting: the process is waiting for some event to occur
- ready: the process is waiting to be assigned to CPU
- terminated: the process has finished execution

Process state

As a process executes, it changes state

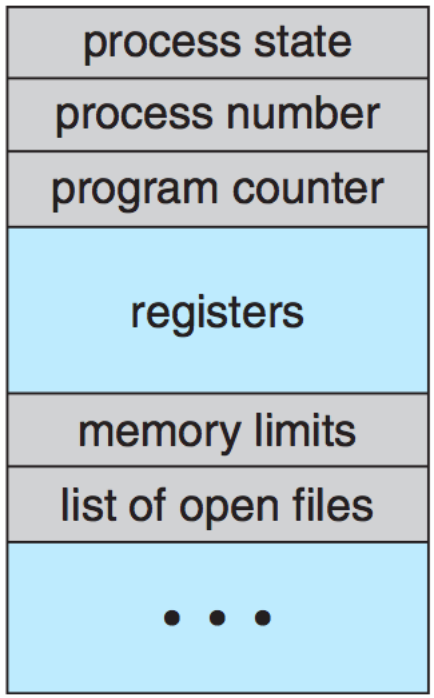
- new: the process is being created
- running: its instructions are being executed
- waiting: the process is waiting for some event to occur
- ready: the process is waiting to be assigned to CPU
- terminated: the process has finished execution



Process control block (PCB)

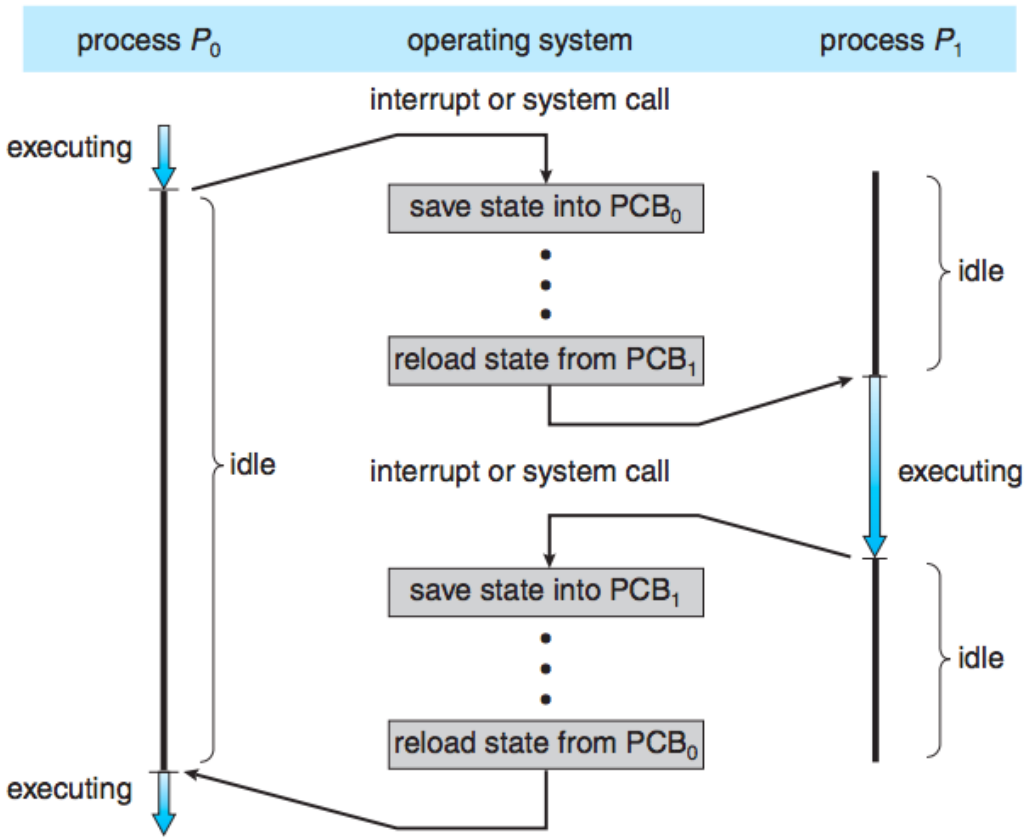
Following information is for a process in operating system, stored in process control block

- Process state
- Program counter
- CPU registers
- CPU scheduling information (e.g., process priority, pointers to scheduling queues)
- Memory management information (e.g., base/limit registers, segment tables)
- Accounting information
- I/O status information (e.g., open files)



Process switching

PCBs are used for process switching in a mutiple tasking operating system



Context switching

- 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
- Context-switch time is overhead. The system does no useful work while switching
  - Time dependent on hardware support
    Ex.: UltraSPARC uses multiple register sets

Outline

Process scheduling queues

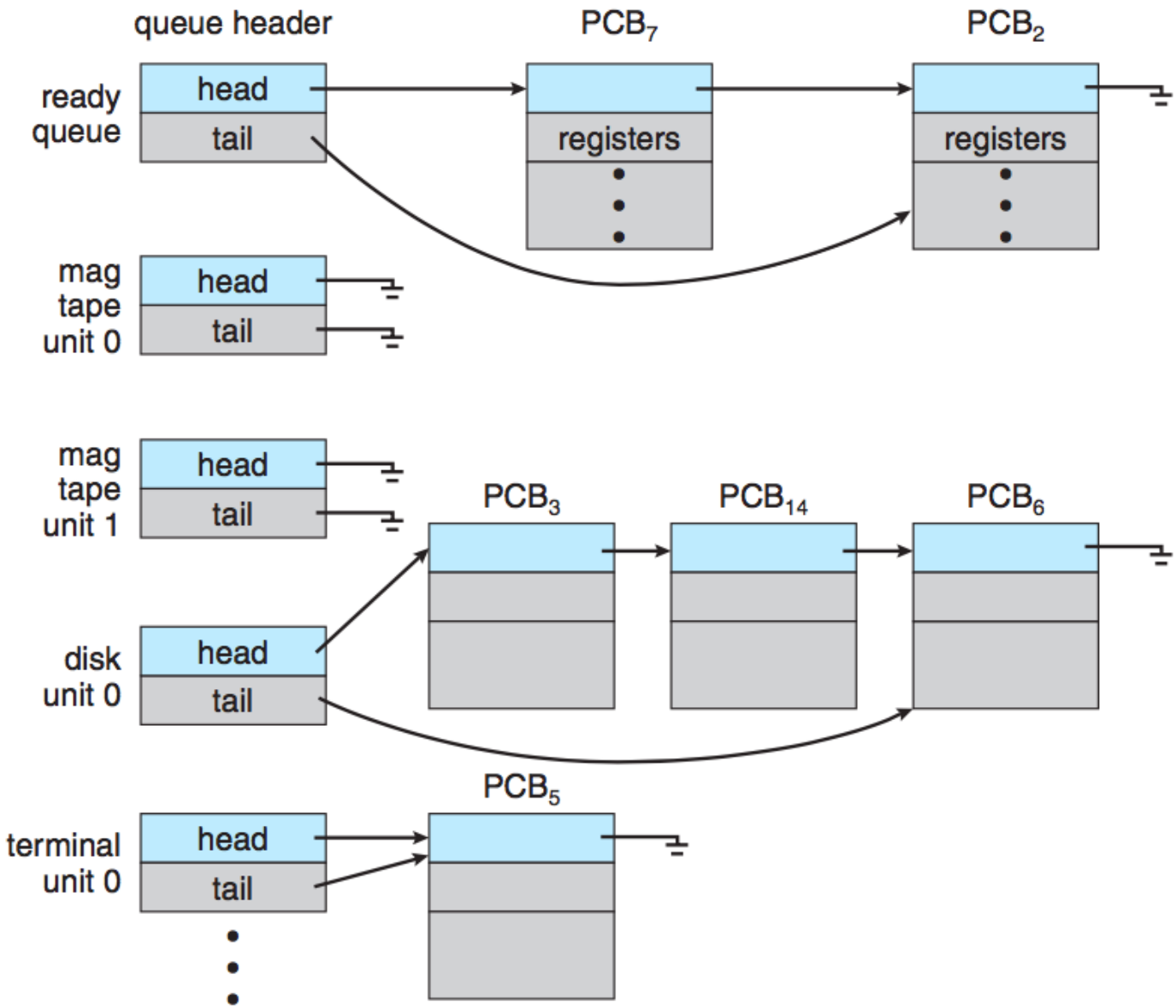
In 5-state process model, we need more than 1 queue to store jobs

Process scheduling queues

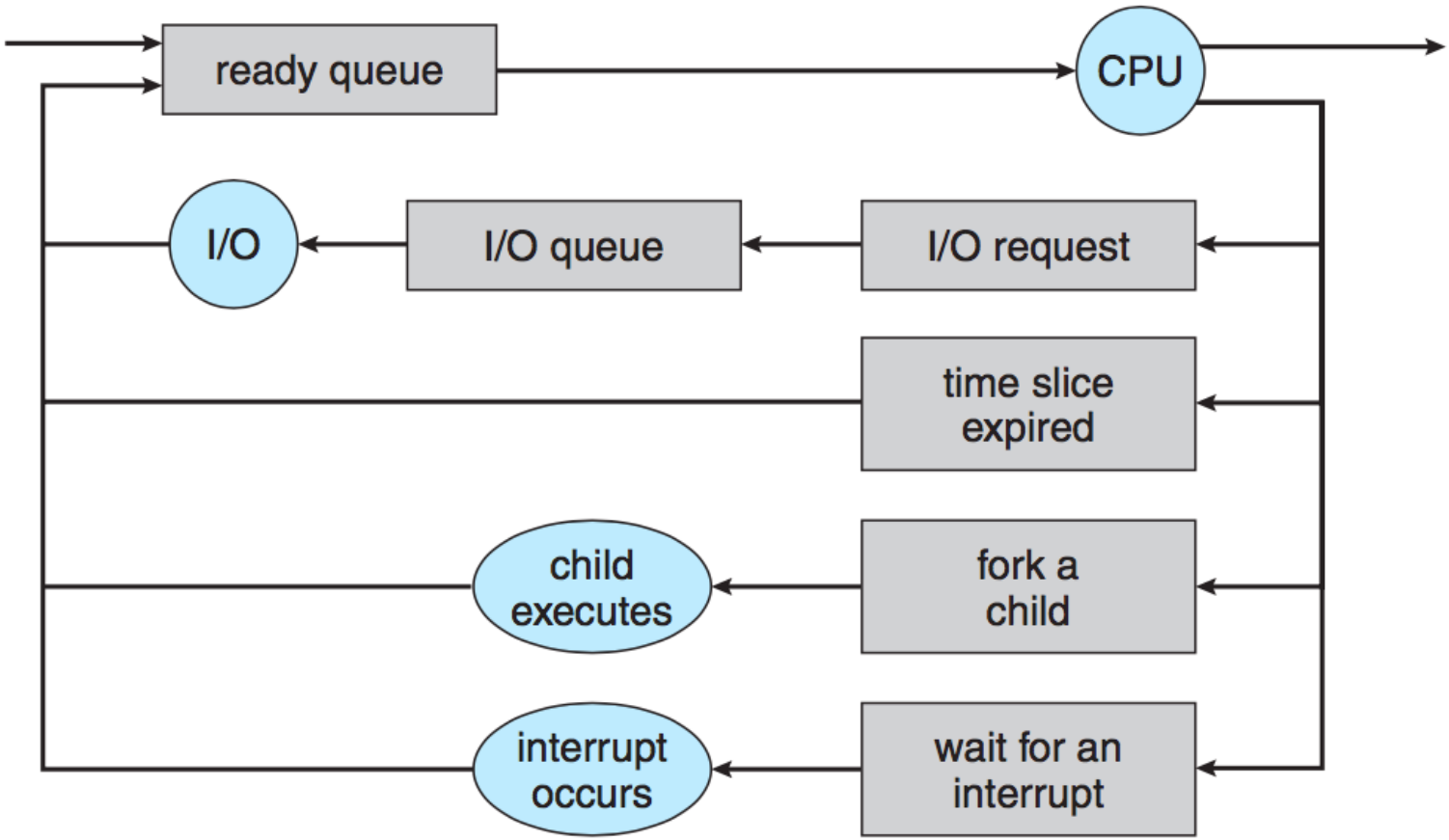
In 5-state process model, we need more than 1 queue to store jobs

- Queue types:
  - 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
- Process migration between the various queues

Process queues as linked lists



Queueing diagram



Schedulers

Processes are selected from the queues for migration, under some selection strategies managed by schedulers.

Schedulers

Processes are selected from the queues for migration, under some selection strategies managed by schedulers.

- Long-term scheduler (or job scheduler): selects which processes should be brought into the ready queue
  - Frequency: only necessary when a process leaves
  - Efficiency depends strongly on I/O bound or CPU bound

Schedulers

Processes are selected from the queues for migration, under some selection strategies managed by schedulers.

- Long-term scheduler (or job scheduler): selects which processes should be brought into the ready queue
  - Frequency: only necessary when a process leaves
  - Efficiency depends strongly on I/O bound or CPU bound
- Short-term scheduler (or CPU scheduler): selects which process should be executed next and allocates CPU
  - Frequency: at least once every 100 milliseconds (*quantum time*)
  - Effciency depends strongly on process switching time

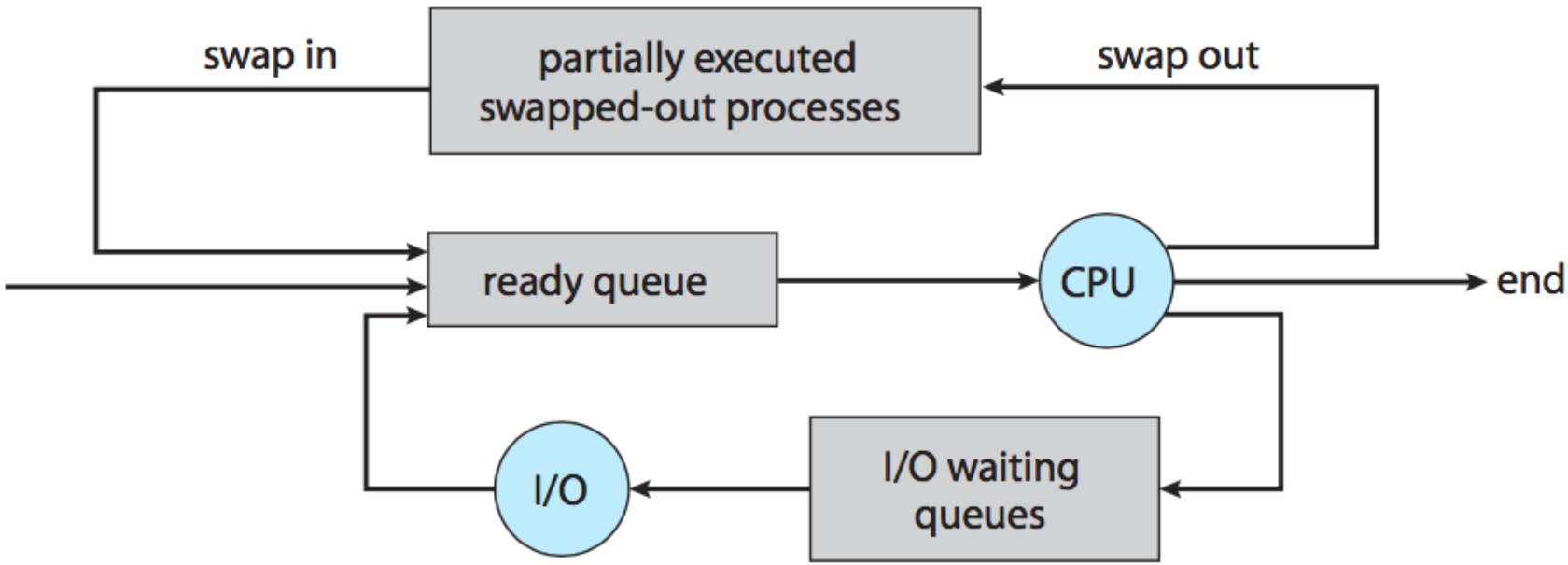
Schedulers

Processes are selected from the queues for migration, under some selection strategies managed by schedulers.

- Long-term scheduler (or job scheduler): selects which processes should be brought into the ready queue
  - Frequency: only necessary when a process leaves
  - Efficiency depends strongly on I/O bound or CPU bound
- Short-term scheduler (or CPU scheduler): selects which process should be executed next and allocates CPU
  - Frequency: at least once every 100 milliseconds (*quantum time*)
  - Effciency depends strongly on process switching time

- Job scheduler is for batch systems
- CPU scheduler is for time-sharing systems

Medium-term scheduler

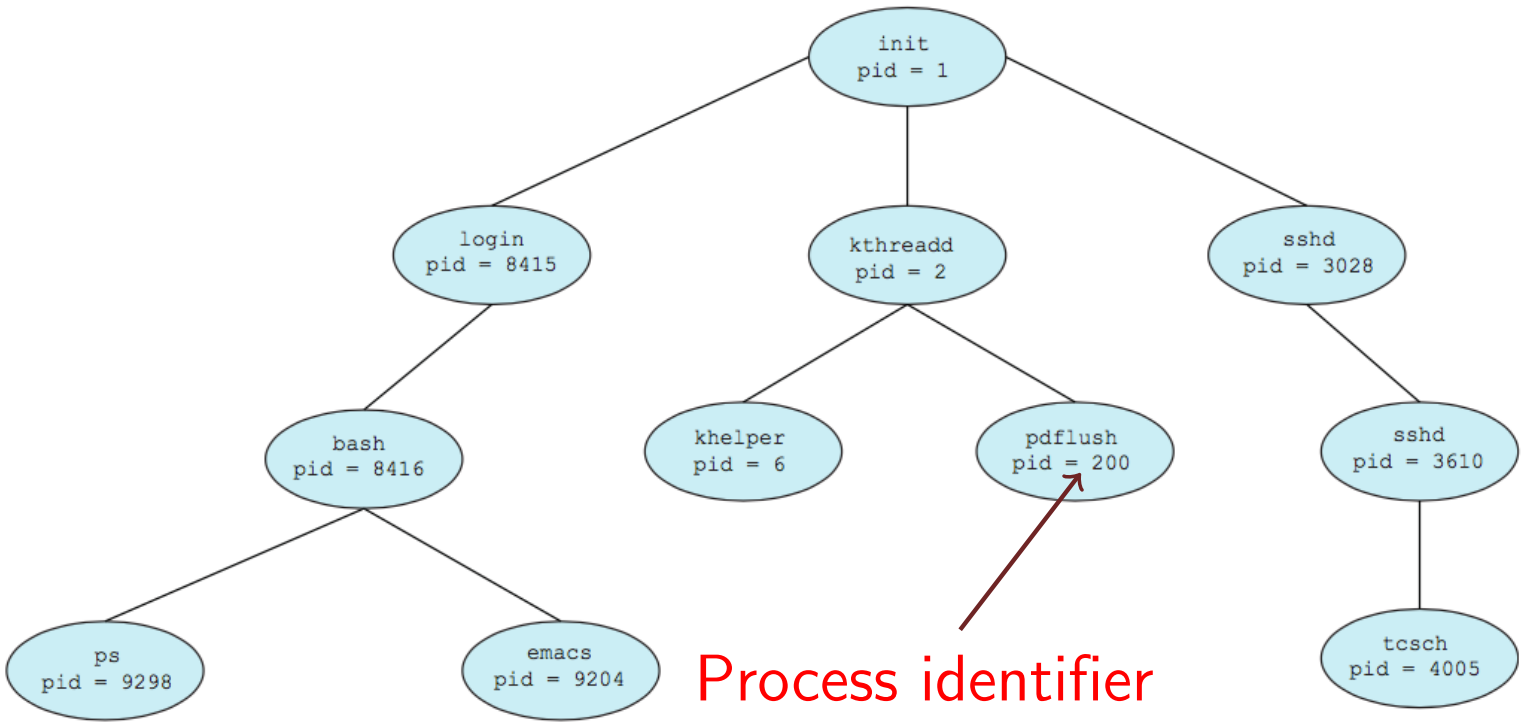


- Swaping is useful to release some resources (for other ready processes)
  - Mobile OSs utilize swaping to save memory and power

Outline

Process tree

Processes are organized in a form of tree



Process creation

- Parent creates children processes

Process creation

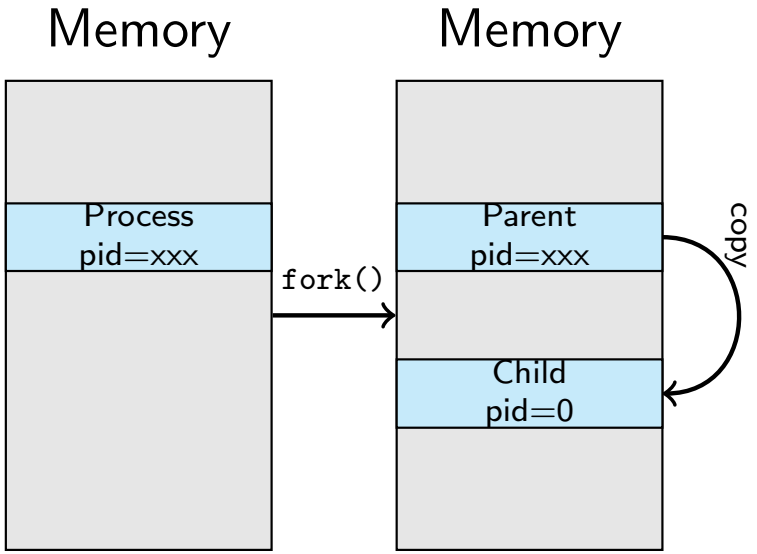
- Parent creates children processes
- Resource sharing: 3 possibilities
  - Parent and children share all resources
  - Children share subset of parent's resources
  - Parent and children share no resources
- Execution: 2 possibilities
  - Parent and children execute concurrently
  - Parent waits until children terminate
- Address-space: 2 possibilities
  - Children duplicate of the parent (program & data)
  - Children load new program

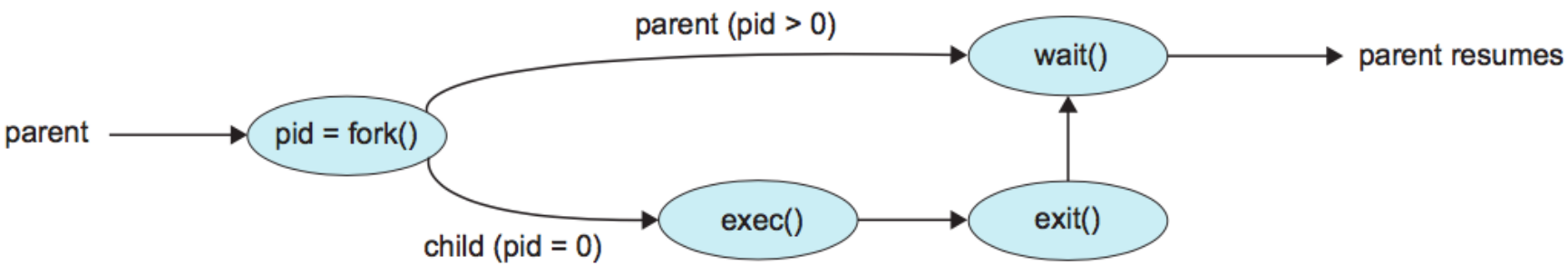
fork() system call

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

int main()
{
  pid_t pid;

  /* fork a child */
  pid = fork();
  if ( pid < 0 ){          /* error occurs */
    fprintf( stderr, "Fork failed\n" );
    return 1;
  } else if ( pid == 0 ){ /* child process  */
    execlp( "/bin/ls", "ls", NULL );
  } else {                 /* parent process */
    wait(NULL);
    printf( "Child complete!\n" );
  }
  return 0;
}
```





Process termination

- Process executes the last statement and asks OS to decide it (`exit()`)
  - Output data from child to parent (via `wait()`)
  - Process's resources are deallocated by OS

Process termination

- Process executes the last statement and asks OS to decide it (`exit()`)
  - Output data from child to parent (via `wait()`)
  - Process's resources are deallocated by OS
- Parent may terminate execution of children processes (`abort()`)
  - Child has exceeded allocated resources
  - Task assigned to child is no longer required
  - Parent is exiting
    - OS does not allow child to continue if its parent terminates $\Rightarrow$ Cascading termination
    - Parent did not invoke `wait()` and instead terminated, thereby leaving its child processes as orphans

Outline

Process relationship

- Independent process: cannot affect or be affected by others
  - Independent process does not share data
- Cooperating process: can affect or be affected by others
  - Cooperating process does share data

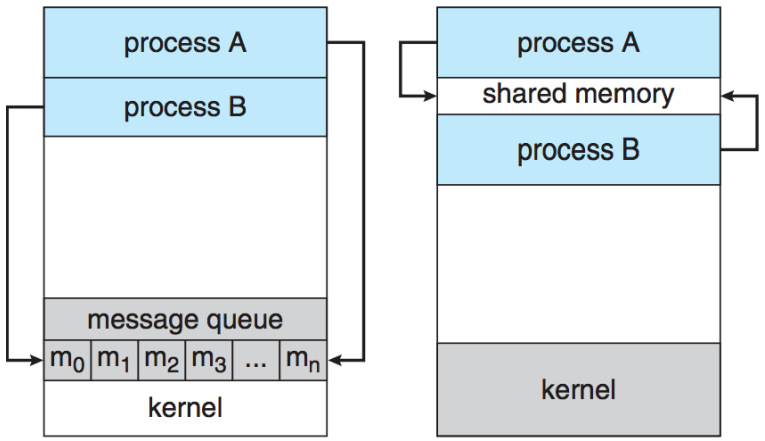
Reasons for process cooperation

- Information sharing
- Computation speedup
- Modularity
- Convenience

Interprocess communication (IPC)

IPC

- IPC is used to exchange data and information
- 2 IPC models: shared-memory and message-passing



- Shared-memory: fast
- Message-passing: no conflict ; suitable for distributed system; better performance on multiple core architecture

Shared-memory

- All things for IPC are decided by processes in communication themselves, not under OS's control

Producer

```
item next_produced;
while (true){
  while ((( in + 1 ) % BUFFER_SIZE ) == out )
    ; /* do nothing */
  buffer[ in ] = next_produced;
  in = ( in + 1 ) % BUFFER_SIZE;
}
```

Consumer

```
item next_consumed;
while (true){
  while ( in == out )
    ; /* do nothing */
  next_consumed = buffer[ out ];
  out = ( out + 1 ) % BUFFER_SIZE;
}
```

- At most `BUFFER_SIZE-1` items in the buffer at the same time

Message-passing

- IPC facility provides at least 2 operations
  - `send(message)`
  - `receive(message)`
- If P and Q wish to communicate, they need to
  - establish a *communication link* between them
  - exchange a message via `send()/receive()`
- Methods to implement communication link
  - physical link: shared memory, hardware bus, network, ...
  - logical link: by following methods
    - Direct or indirect communication
    - Synchronous or asynchronous communication
    - Automatic or explicit buffering

- How are links establised ?
- 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 ?

Explicit name of sender and receiver must be given

- `send(P,message)`
- `receive(Q,message)`

- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- The link may be unidirectional, but is usually bi-directional

Explicit name of sender and receiver must be given

- `send(P,message)`
- `receive(Q,message)`

- Links are established automatically
- A link is associated with exactly one pair of communicating processes
- The link may be unidirectional, but is usually bi-directional

Disadvantage

Direct communication has poor modularity

Messages are sent to or received from a mailbox or port

- `send(A,message)`
- `receive(A,message)`

`A` is a mailbox with unique identification

- A link between 2 processes is established only if both have a shared mailbox
- A link may be associated with more than 2 processes
- Between each pair, several different links may exist

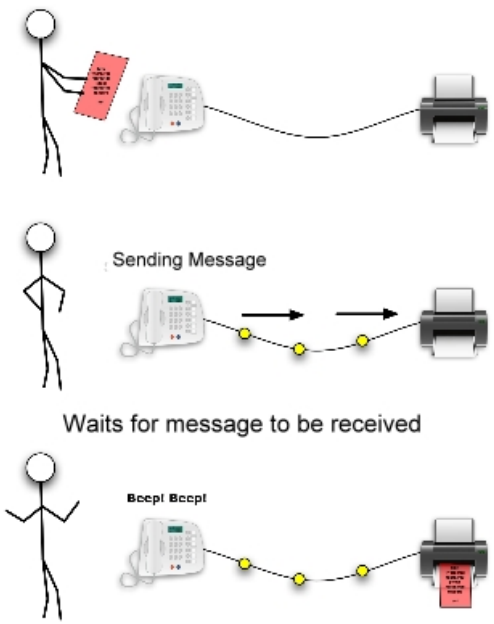
- P_1, P_2 and P_3 share a mailbox.
  - P_1 sends a message
  - P_2 and P_3 receive
  - Who gets the message ?

- P_1, P_2 and P_3 share a mailbox.
  - P_1 sends a message
  - P_2 and P_3 receive
  - Who gets the message ?
- It depends on which of the following methods is chosen
  - A link associated with 2 processes at most
  - Allowing one process to perform `receive()` at a time
  - An algorithm to choose which process to perform `receive()` (e.g., round-robin)

- P_1, P_2 and P_3 share a mailbox.
  - P_1 sends a message
  - P_2 and P_3 receive
  - Who gets the message ?
- It depends on which of the following methods is chosen
  - A link associated with 2 processes at most
  - Allowing one process to perform `receive()` at a time
  - An algorithm to choose which process to perform `receive()` (e.g., round-robin)
- A mailbox may be owned by a process or OS. If it owned by OS, system calls must be provided to
  1. Create a new mailbox
  2. Send and receive messages through the mailbox
  3. Delete a mailbox

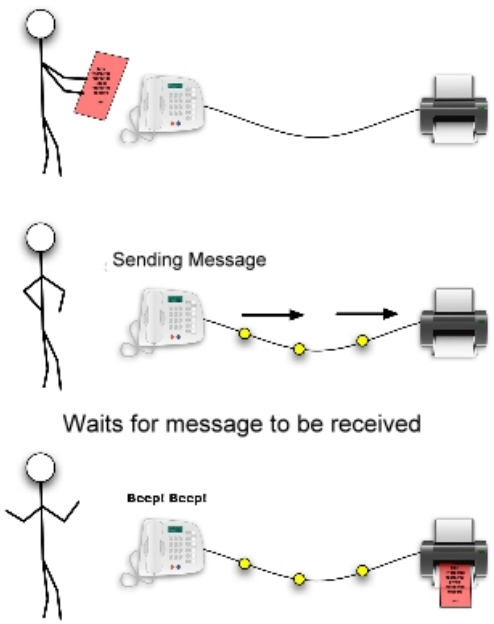
`send()` and `receive()` can be blocking (synchronous) or non-blocking (asynchronous)

`send()` and `receive()` can be blocking (synchronous) or non-blocking (asynchronous)

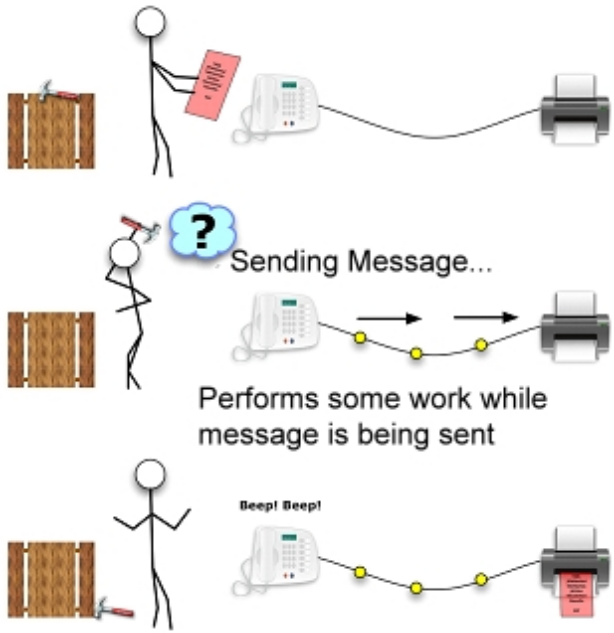


(source: cfd-online.com)

`send()` and `receive()` can be blocking (synchronous) or non-blocking (asynchronous)



(source: cfd-online.com)

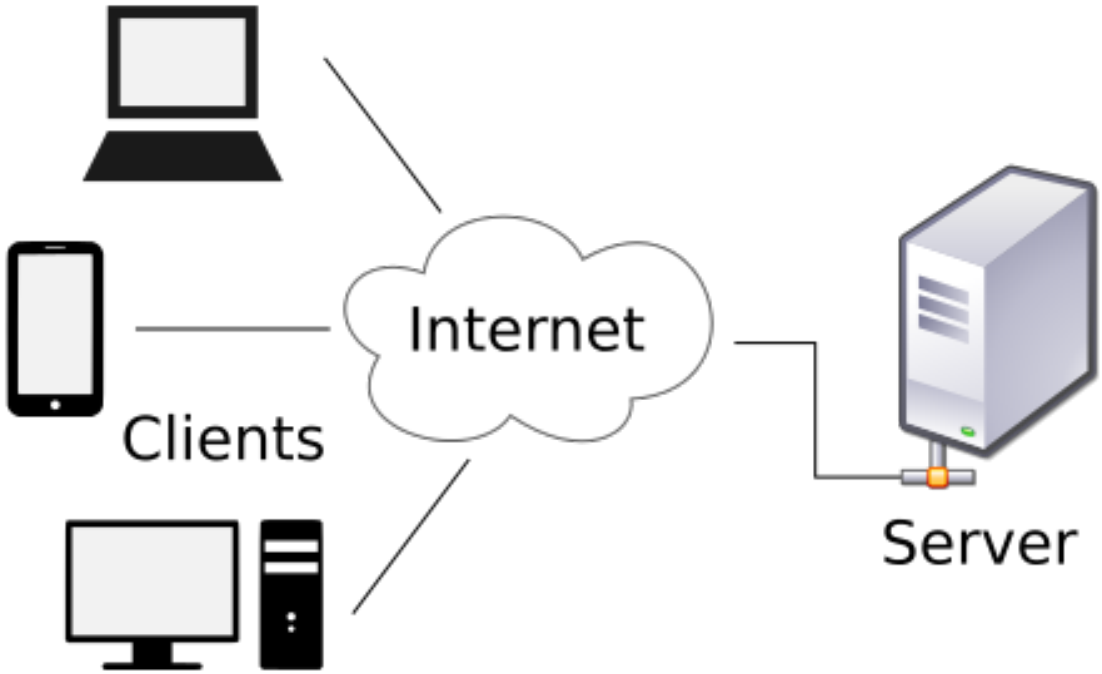


(source: cfd-online.com)

- Message queues attached to the link; implemented in one of 3 ways
  - Zero capacity: 0 message; sender must wait for receiver
  - Bounded capacity: finite length of n; sender must wait if link is full
  - Unbounded capacity: infinite length; sender never waits

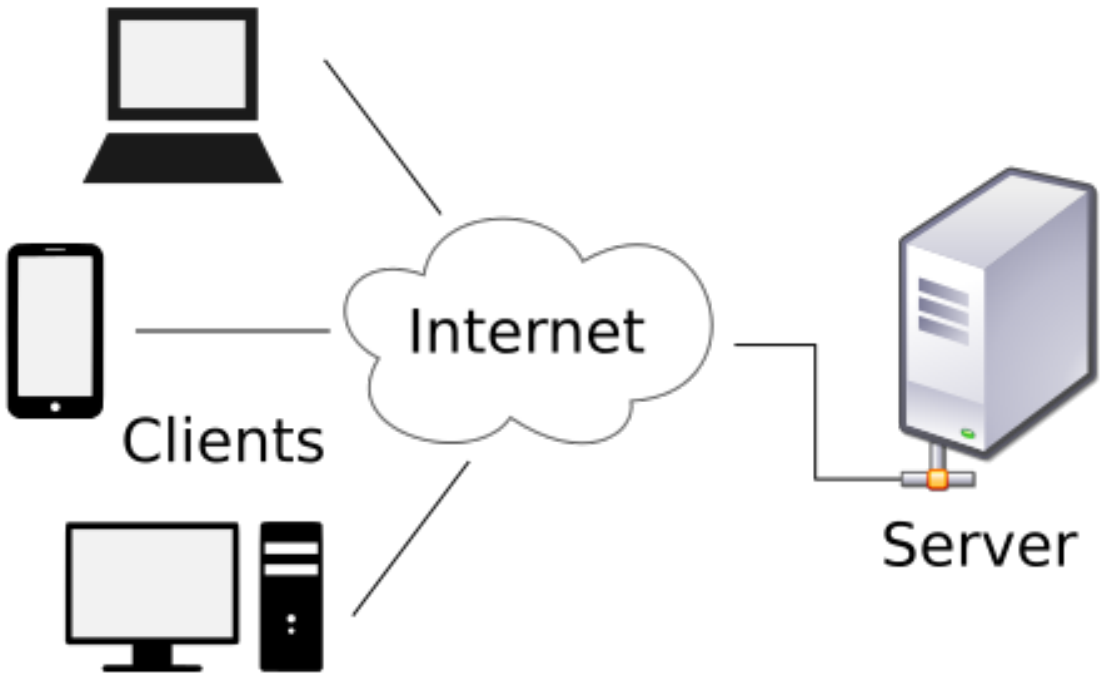
Outline

Client-server communication



(source: wikipedia)

Client-server communication



(source: wikipedia)

- Sockets
- Remote Procedure Calls
  - Remote Method Invocation (Java)
- Pipes

Socket

Socket is an endpoint, consisting of

- IP address
- port

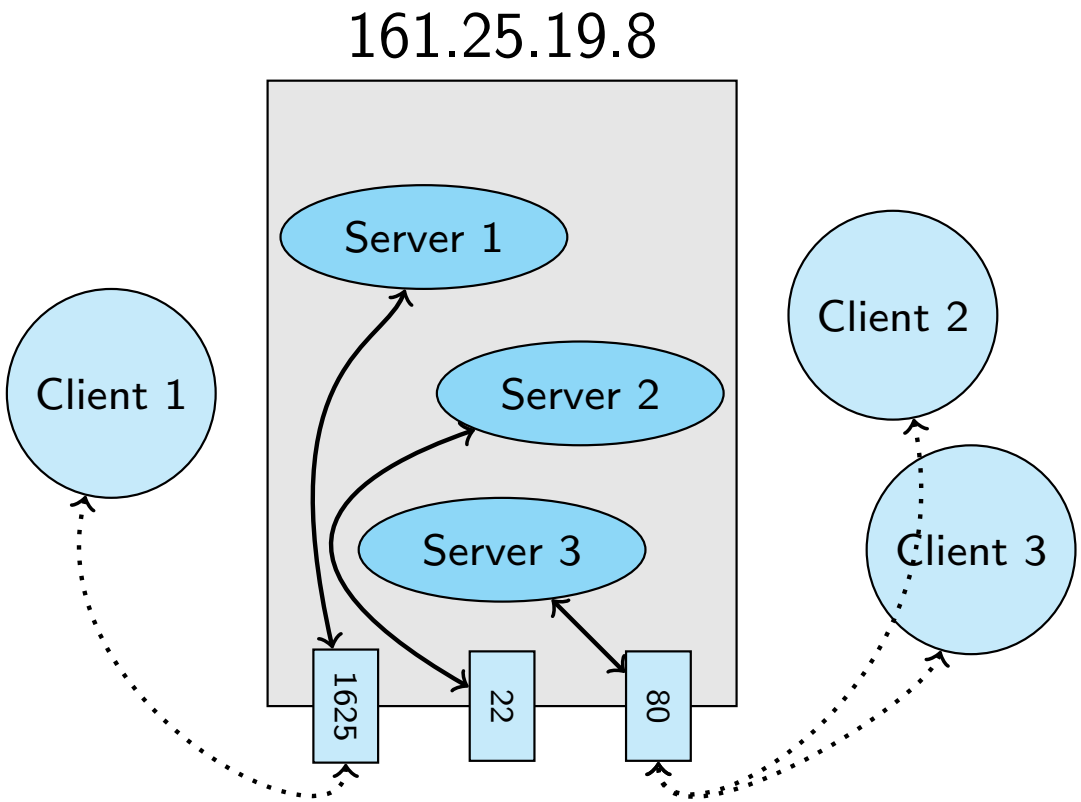
- Socket
  161.25.19.8:1625
  refers to port 1625
  on host 161.25.19.8

Socket

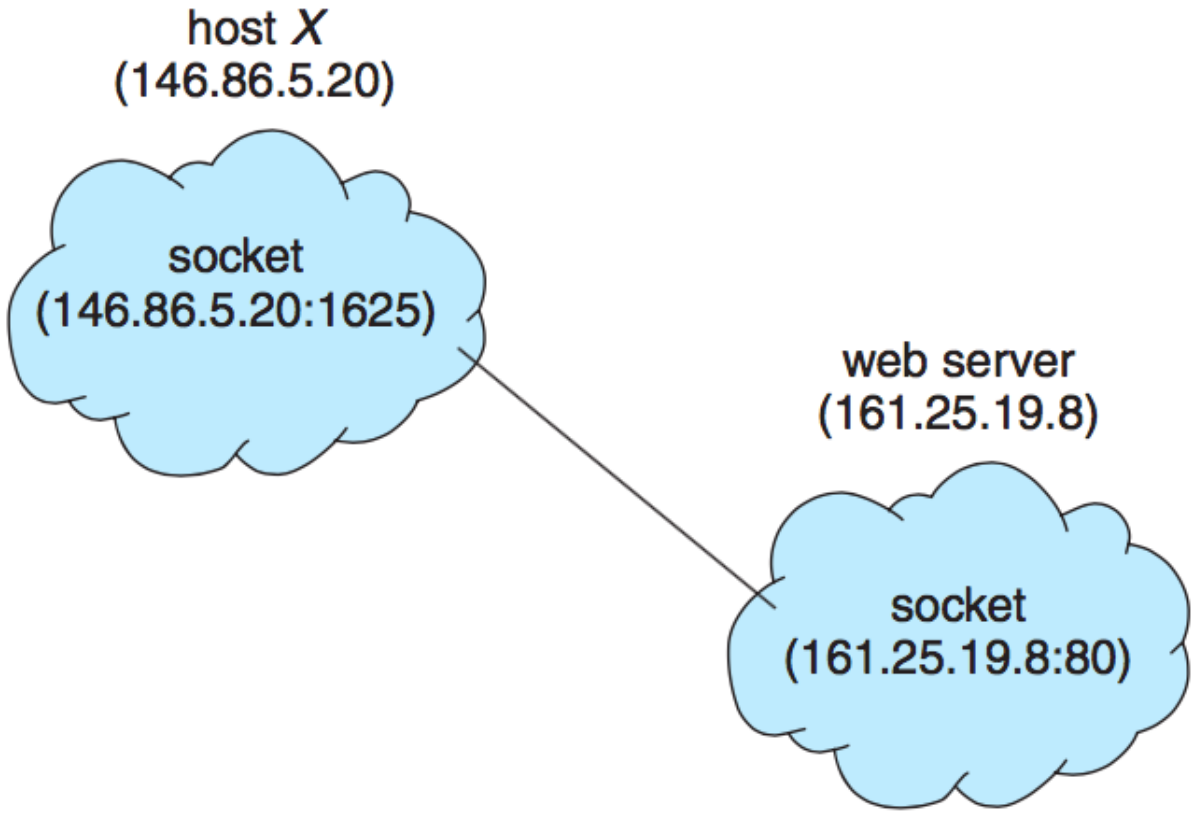
Socket is an endpoint, consisting of

- IP address
- port

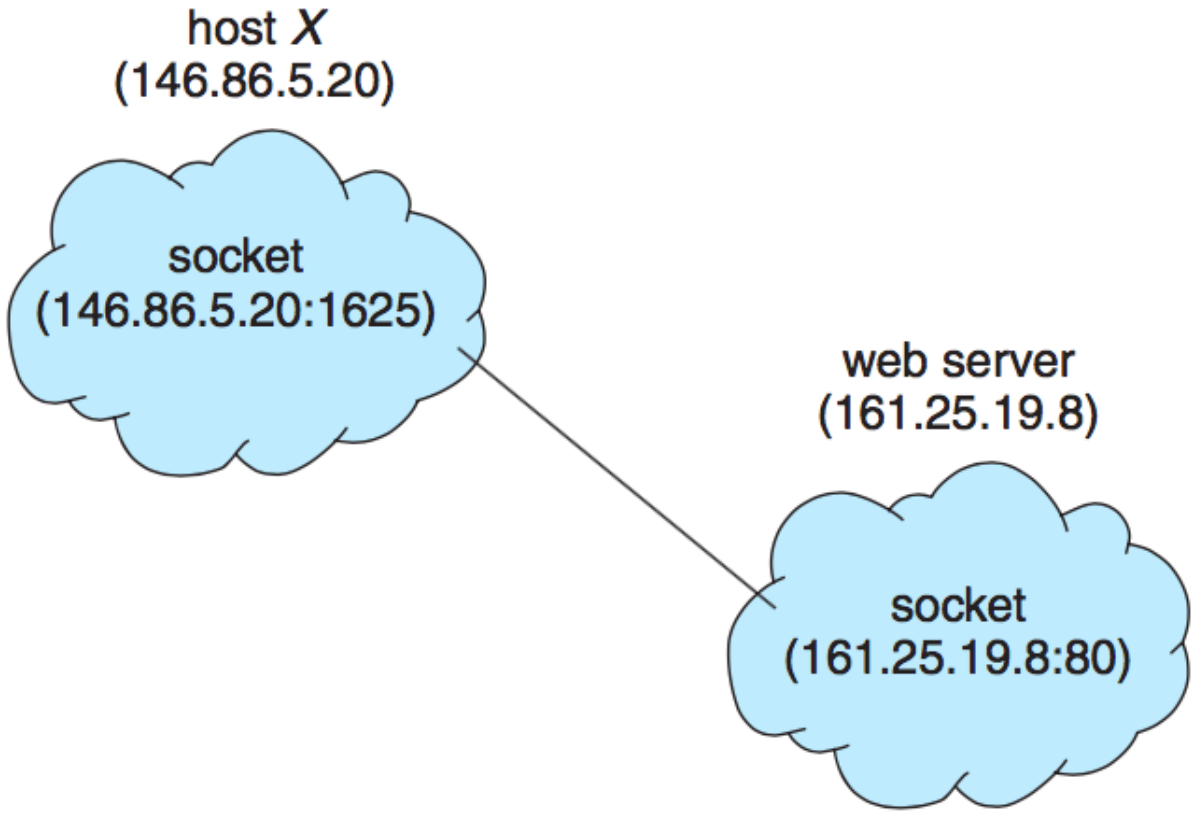
- Socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8



Socket communication



Socket communication



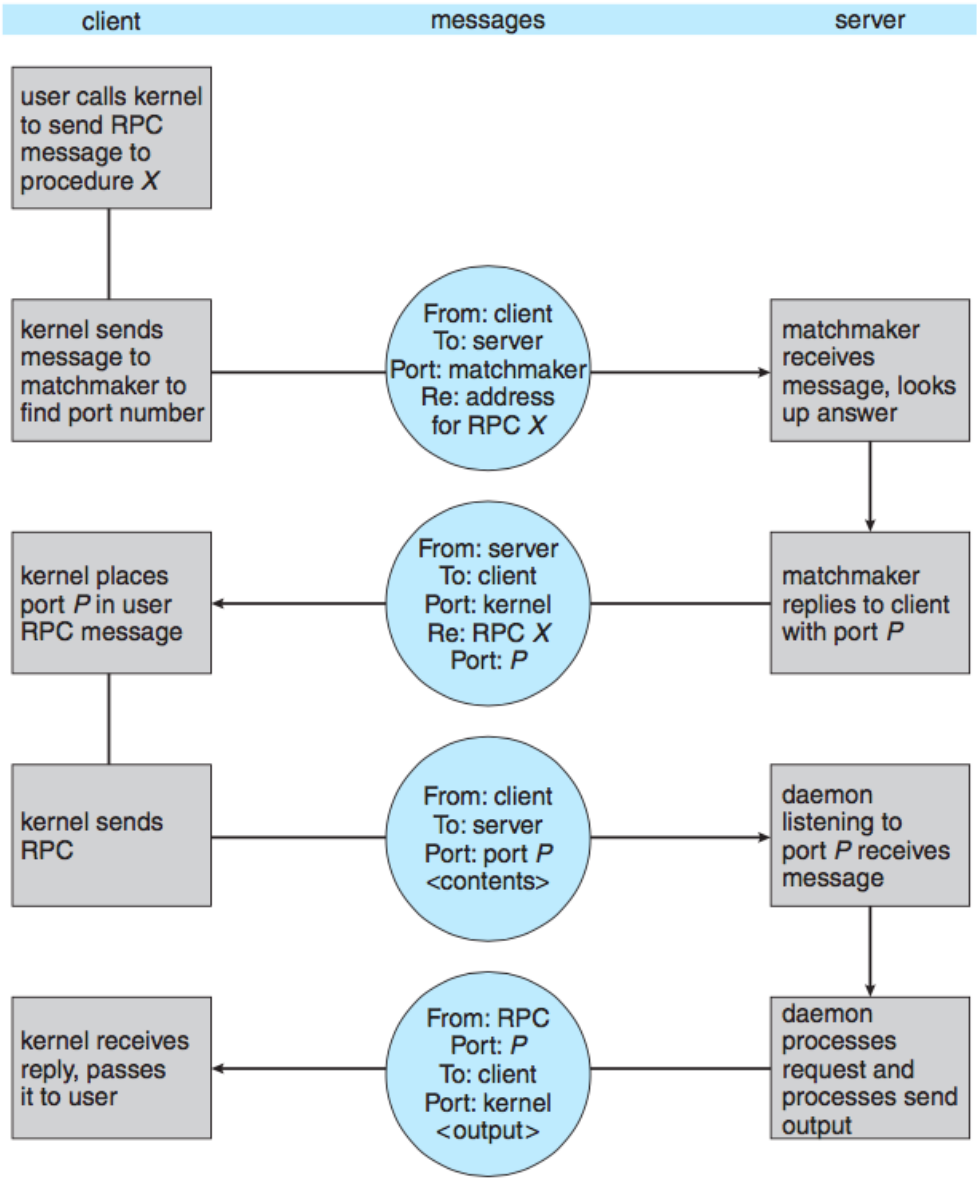
Socket is low-level form of communication in which data is transferred in unstructured stream.

Remote procedure call (RPC)

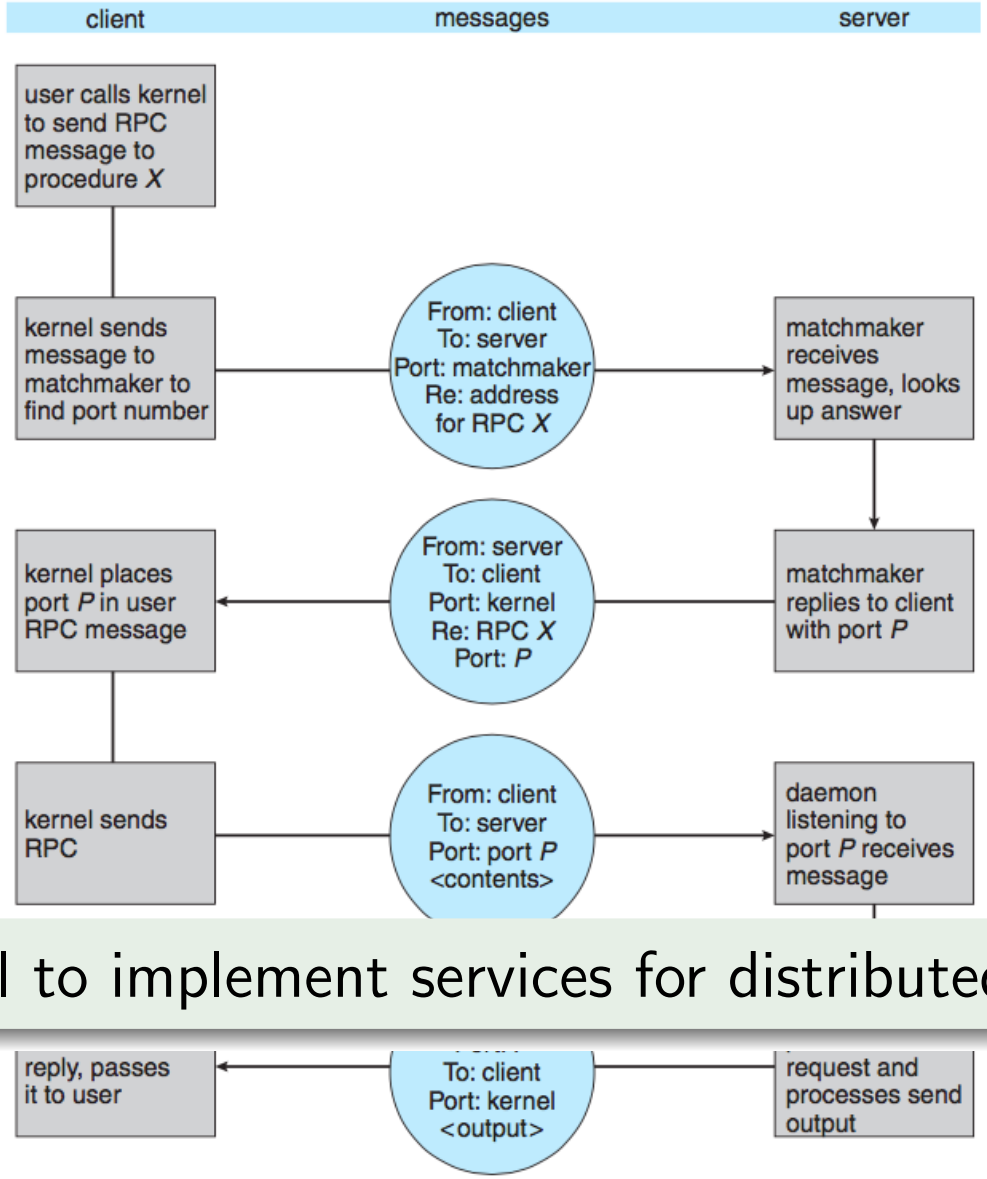
Remote procedure call (RPC) abstracts procedure calls between processes on networked systems

- Messages exchanged in RPC are well-structured (function name, parameters)
- Stubs: client-side proxy for the actual procedure on the server
  - separate stub for each separate remote procedure
  - stub locates port on server and marshalls parameters into a package (by a mechanism of External Data Representation (XDR))
- The server-side stub receives this message, unpacks the marshalled parameters, and peforms the procedure on the server

Execution of RPC



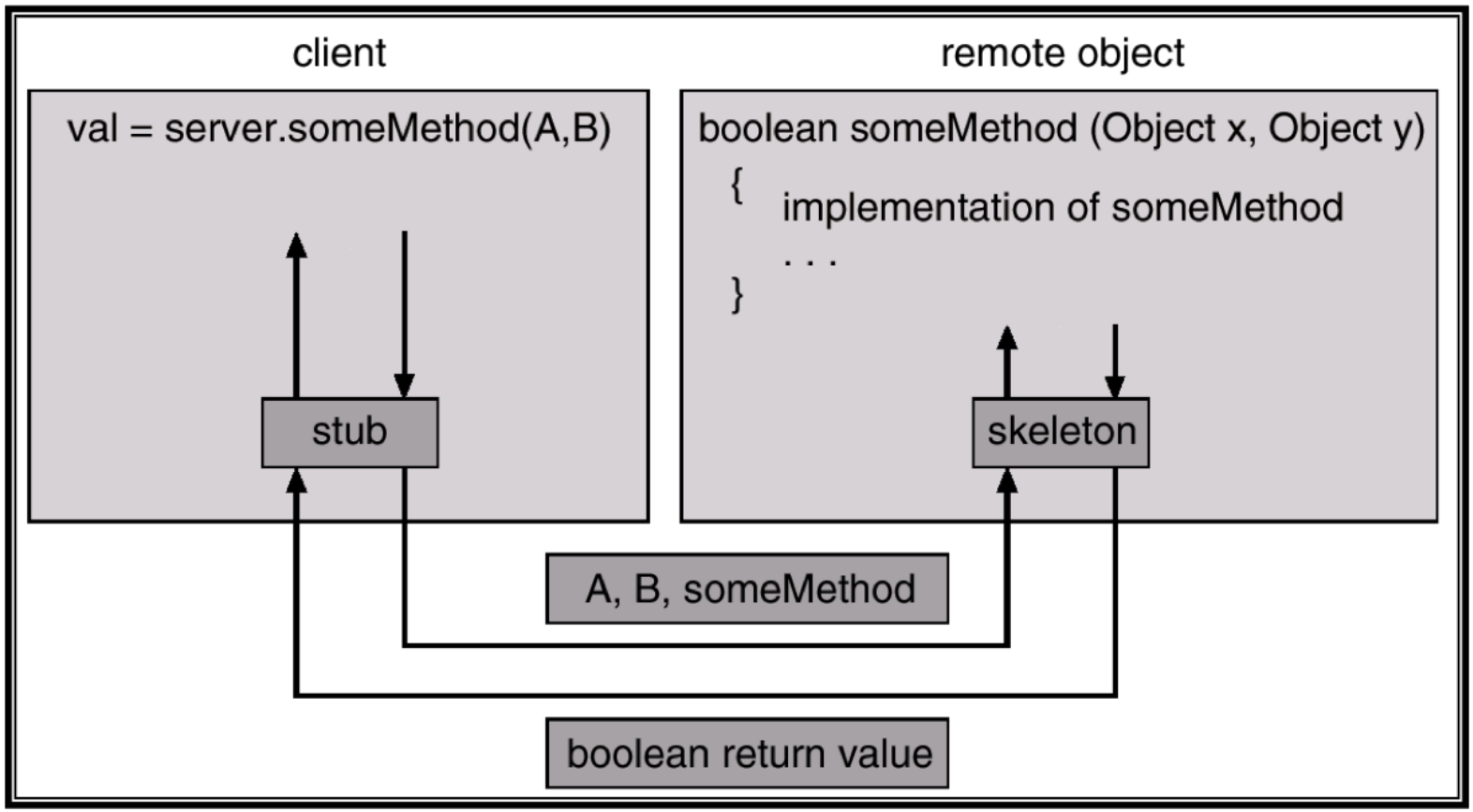
Execution of RPC



RPC is useful to implement services for distributed systems

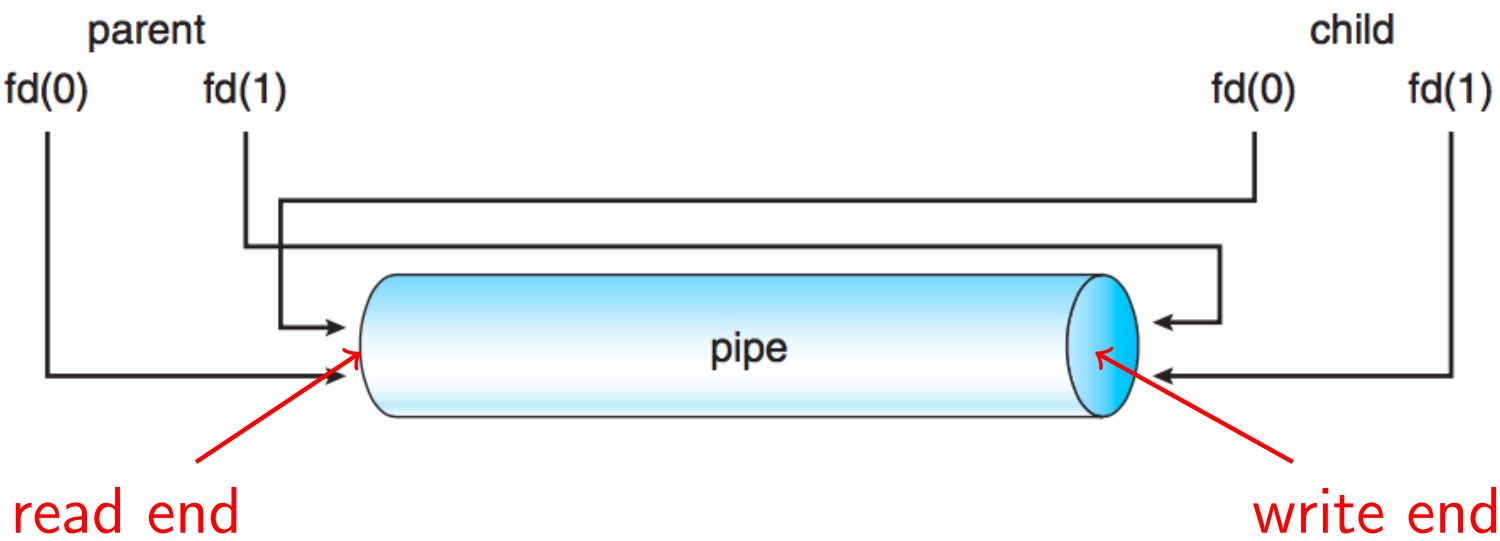
Remote method invocation

- A Java mechanism and quite similar to RPC
- RMI allows a Java program on one machine to invoke a method on a remote object



Pipe

- Pipe is one of very first IPC mechanisms in early UNIX
- Ordinary pipe is unidirectional
- Pipes can be treated as a special type of file



Anonymous pipe

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

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

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

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

  /* fork a child process */
  pid = fork();
  if ( pid < 0 ){
    fprintf( stderr, "Fork failed!\n" );
    return 1;
  }
  if ( pid > 0 ){
    /* close unused end of pipe */
    close( fd[READ_END] );

    /* write to the pipe */
    write( fd[WRITE_END], write_msg,
             strlen(write_msg) + 1 );

    /* close the write end */
    close( fd[WRITE_END] );
  }
  else {
    /* close unused end of pipe */
    close( fd[WRITE_END] );

    /* write to the pipe */
    read( fd[READ_END], read_msg, BUFFER_SIZE );

    /* close the read end */
    close( fd[READ_END] );
  }

  return 0;
}
```

Homeworks

1. Read materials on Thread: textbook, slides (Le Thanh Van)
2. There is quiz in April 1, 2021) on Process & Thread

Homeworks

1. Read materials on Thread: textbook, slides (Le Thanh Van)
2. There is quiz in April 1, 2021) on Process & Thread
3. Quiz grade (of students in 2017)