Operating System Concepts
Ch. 3: Processes

Silberschatz, Galvin & Gagne

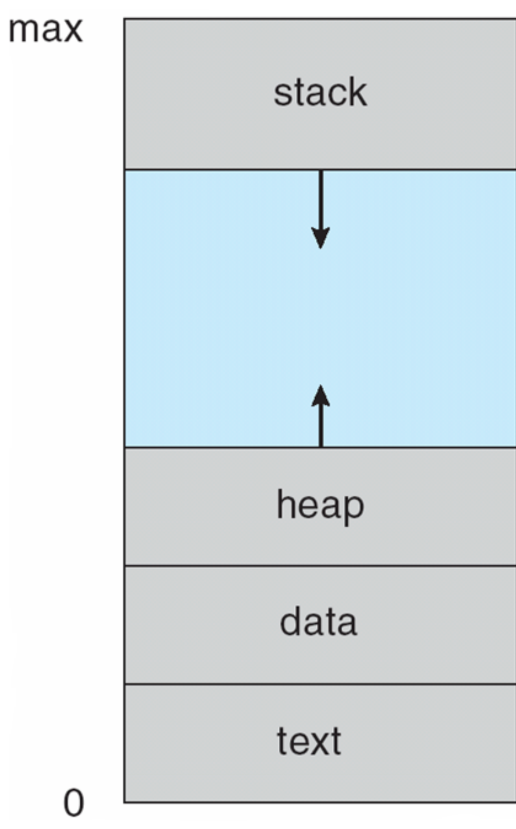
Universiteit Leiden
The Netherlands

Process Concept

- Recall:
  - **Program**: *passive* entity stored on secondary storage (executable files); instructions & initialization data.
  - **Process**: *active* entity; program in execution.
- Programs can be started in various ways:
  - By the system itself (system start up, periodic tasks)
  - By the user through a user interface (command line based, graphical)
  - For batch systems: job submission and the job reaches the front of the queue.

Process Structure

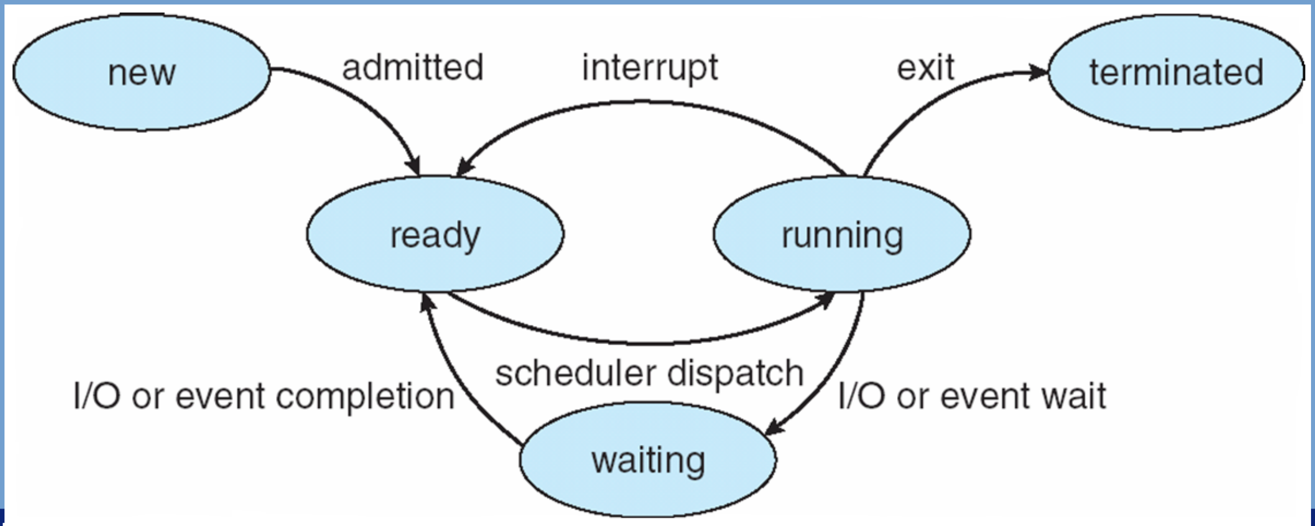
- Processes consist of multiple parts:
  - *Text section*: program code, the instructions.
  - *Data section*: initialization data for global variables
  - *Stack*: contains temporary data, used during program execution to create local variables, pass function arguments, etc.
  - *Heap*: contains dynamic memory allocations (new, malloc)
  - Stack and heap are placed at opposite ends and grow towards each other.
  - Current CPU register state, including program counter.
- All these parts must be prepared when the program is loaded in memory.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Process State

- With each process a state is associated. The following states are (roughly) distinguished:
  - **new**: process is being created
  - **ready**: process is ready to be run (waiting to be put on CPU)
  - **running**: process is running, so executing instructions
  - **waiting**: process is waiting for some event to occur / request to complete
  - **terminate**: process is being terminated
- Note: names and availability of states differs per system.
- Current state of a process can be seen in e.g. top utility.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

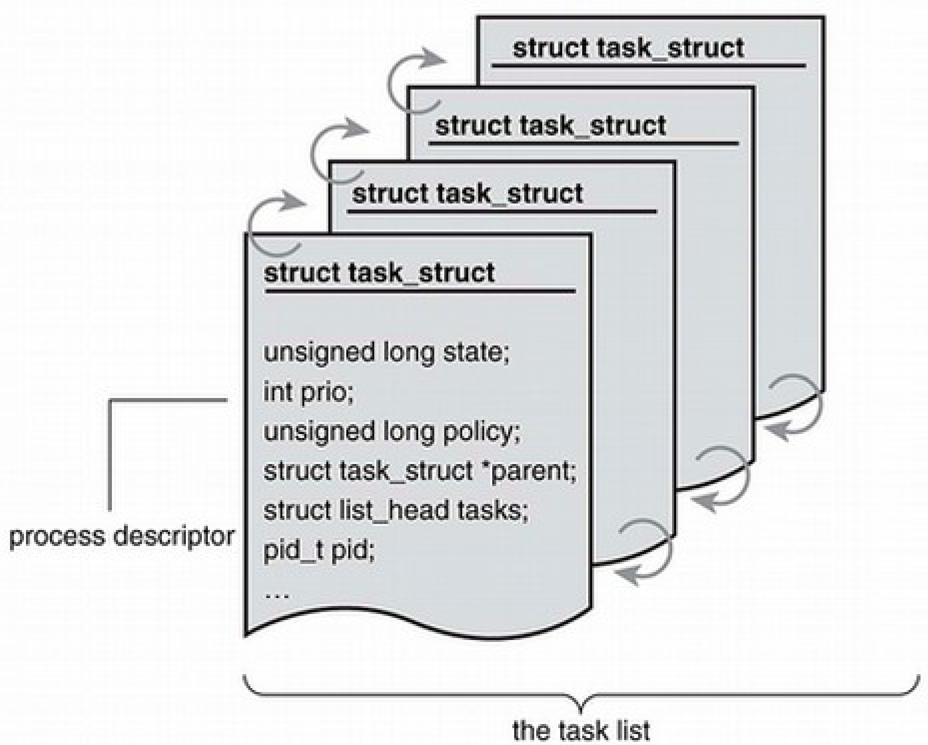
Bookkeeping Processes

It is the responsibility of the Operating System to keep track of:

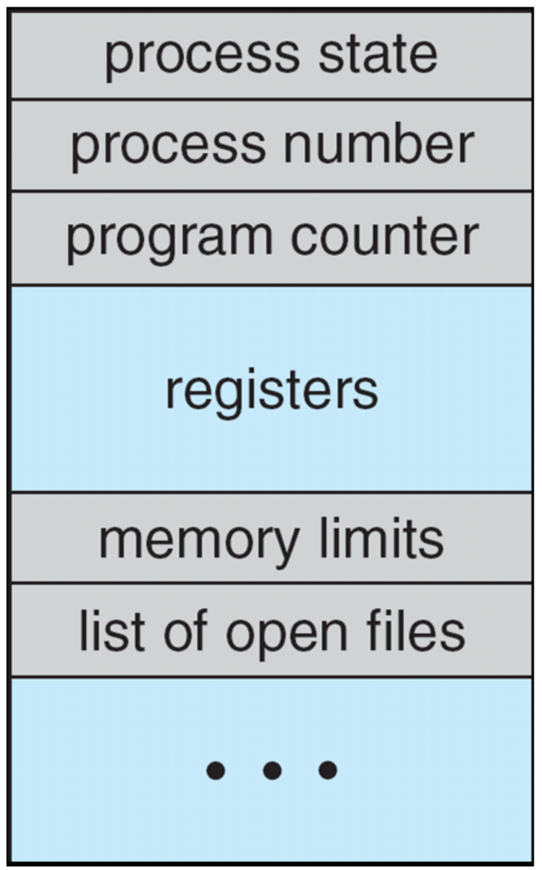
- the processes that are active in the system; for instance a table or linked list of processes is maintained;
- various information associated with a process:
  - allocated memory segments
  - register state when process is not active (suspended/waiting)
  - open files, network connections
  - process identifier (pid)
  - process state
  - process owner
  - scheduling information
  - consumed CPU cycles
  - etc., etc.

Bookkeeping Processes (2)

- The information associated with a process is stored in a Process Control Block.
  - Typically a C-structure.
  - Linux has a `struct task_struct` of approx. 500 lines.



Source: Robert Love, Linux Kernel Development, 2nd Edition

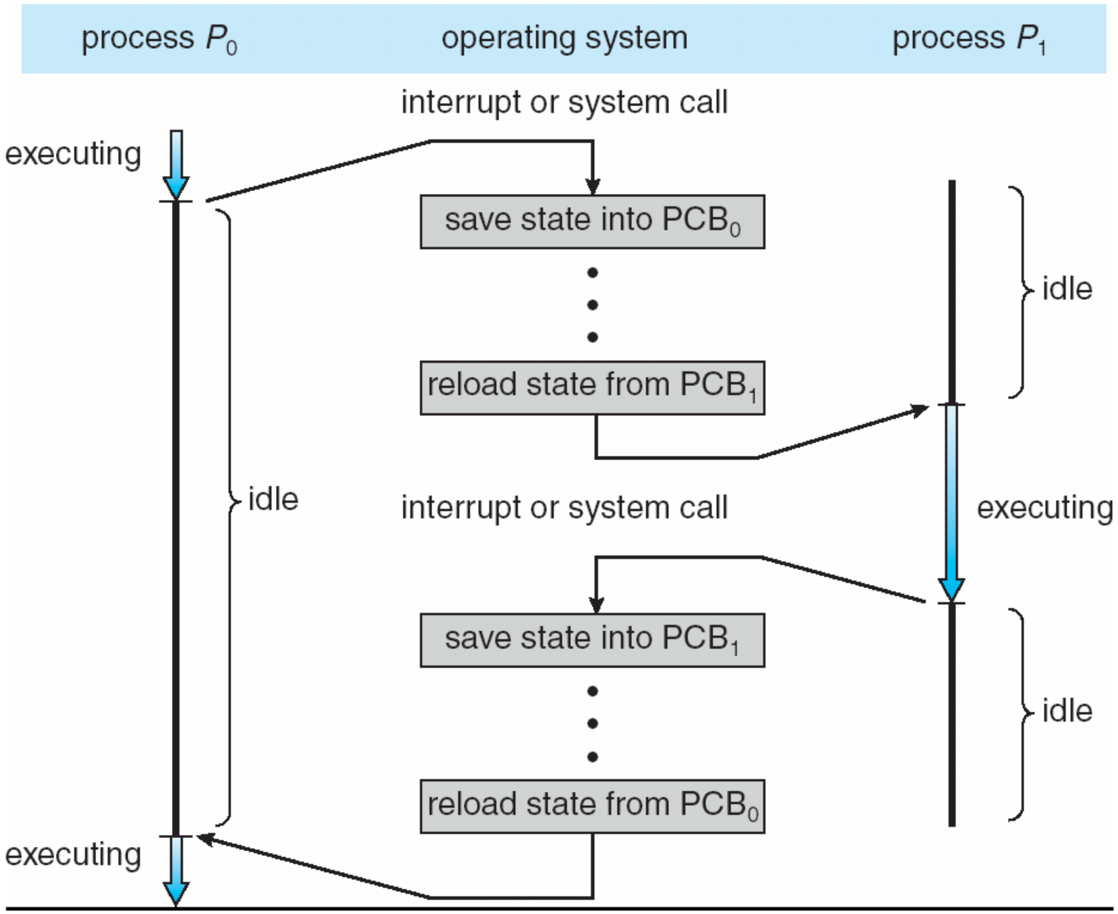


Source: Silberschatz et al., Operating System Concepts, 9th Edition

Switching Between Processes

- An important capability of multiprogramming/timesharing systems is that of *switching* between different processes.
- This entails suspending the process that is currently running on the CPU and resuming another.
  - **Suspend**: storing necessary information ("its state") in the process control block. Think of program counter & register state!
  - **Resume**: restoring information from process control block to CPU register.
  - So, the entire **state** of the process is temporarily stored in memory!
- This is a routine that is implemented in the OS kernel and runs in kernel mode.
- We refer to this procedure as *context switching*.

Switching Between Processes (2)



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Switching Between Processes (3)

- In case of multi-core systems, processes can be separately switched per core.
- The *context switch* routine is *hardware-dependent*. It depends on the underlying hardware platform since it has to save/restore specific CPU registers.
  - This implies that the amount of work this routine has to perform and its time duration depend on the hardware platform.
  - On some systems caches need to be (partly) flushed, on others this is not necessary. Need to check architecture reference manuals!
- Context switch time is pure overhead, no useful work is done.
  - So, you don't want to switch too often.
  - But only switching every few minutes leads to non-interactive systems. An important trade-off to make.

Threads

- Up till now, we considered a single process state, program counter and registers to be associated with a process.
  - A single thread of execution.
- When speaking of multiple threads, we have a single process that contains multiple threads of execution.
  - Every thread of execution needs a program counter, register state to be (re)stored and a stack.
  - Process Control Block is extended or organized differently to accommodate this.
- See also Chapter 4 on Multi-Threading.

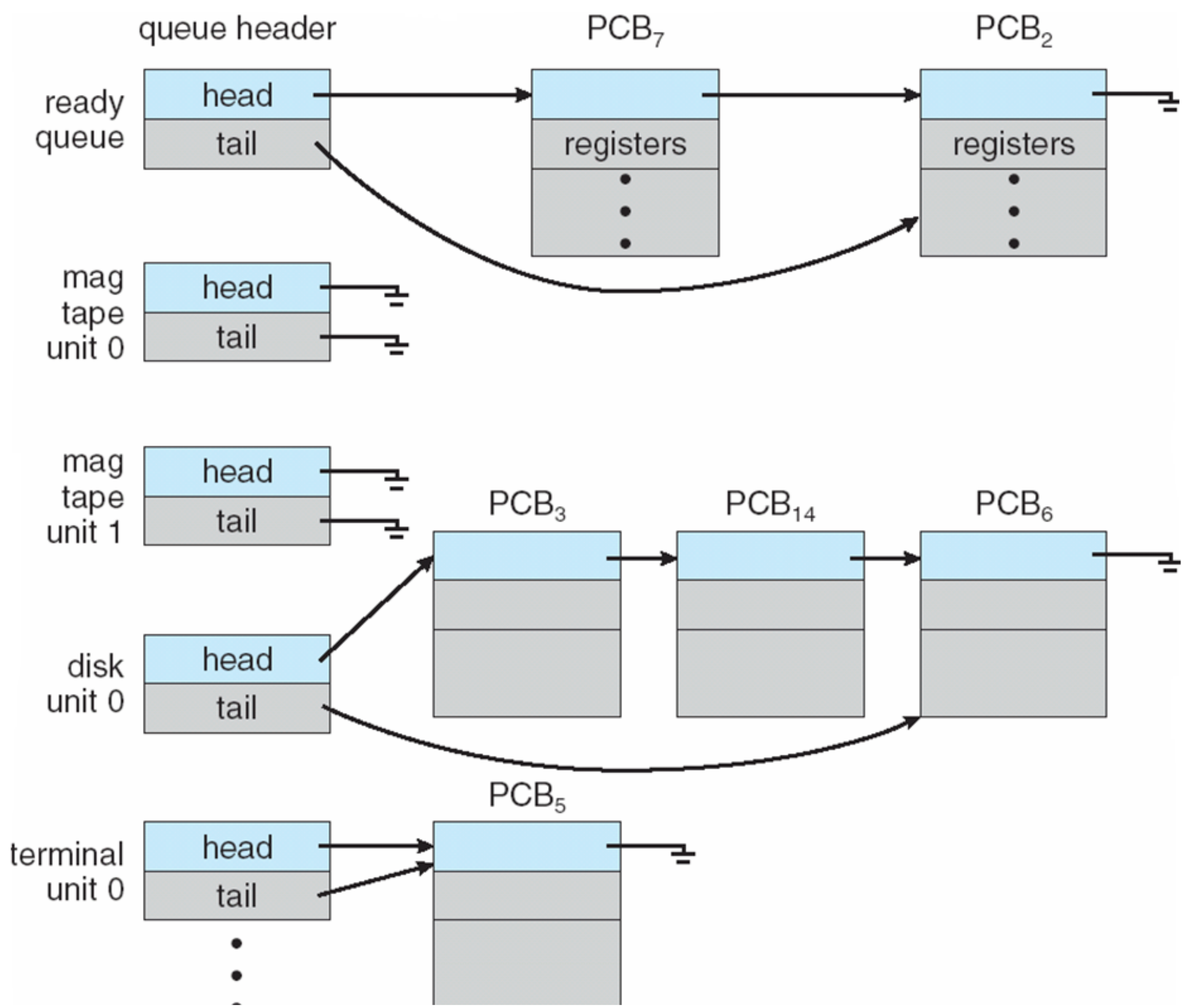
Process Scheduling

- Assume we have 1 CPU available and multiple processes in our process list. Which process do we assign to the CPU? How do we decide? Who decides?
- The kernel decides and uses an algorithm referred to as the CPU scheduling algorithm.
- Objective of this algorithm:
  - We have a resource, the CPU, and we want to maximize the use of this CPU.
  - We have a picky user and we want to maximize responsiveness of the graphical user interface.
  - Sometimes conflicting interests: for super computers differently tuned algorithms are used compared to smart phones.
  - But underlying principles are the same!

Process Scheduling (2)

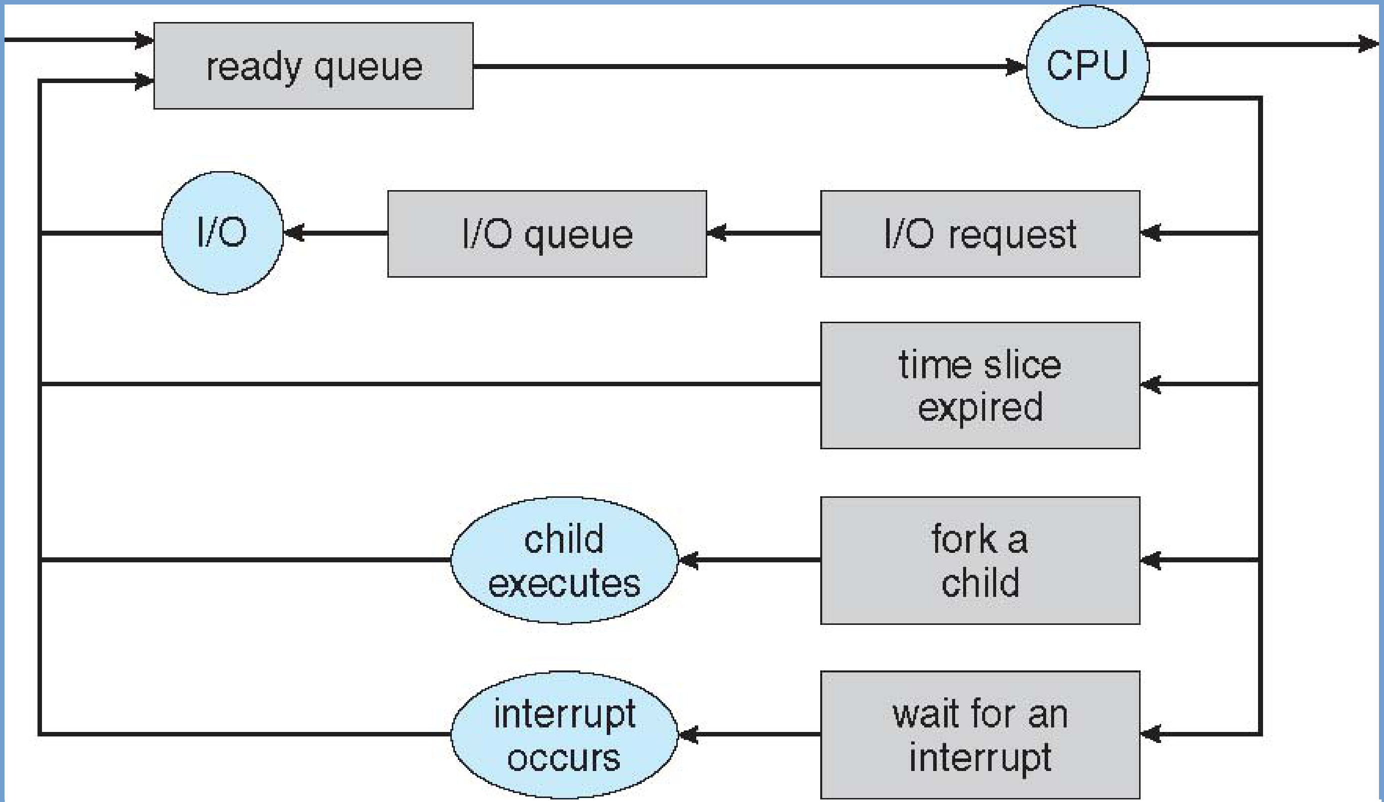
- In an OS we commonly have multiple lists or queues of processes.
  - An overall list of tasks registered in the system (job queue).
  - A queue of processes that are ready for execution and are not blocking on anything (ready queue).
  - A per-device queue of processes waiting for service (device queue, wait queue).
  - A queue of processes that suspended itself (sleeping); in fact these are waiting for an appropriate timer interrupt.
- Processes migrate between the different queues.

Process Scheduling (3)



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Process Scheduling (4)



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Process Scheduling (5)

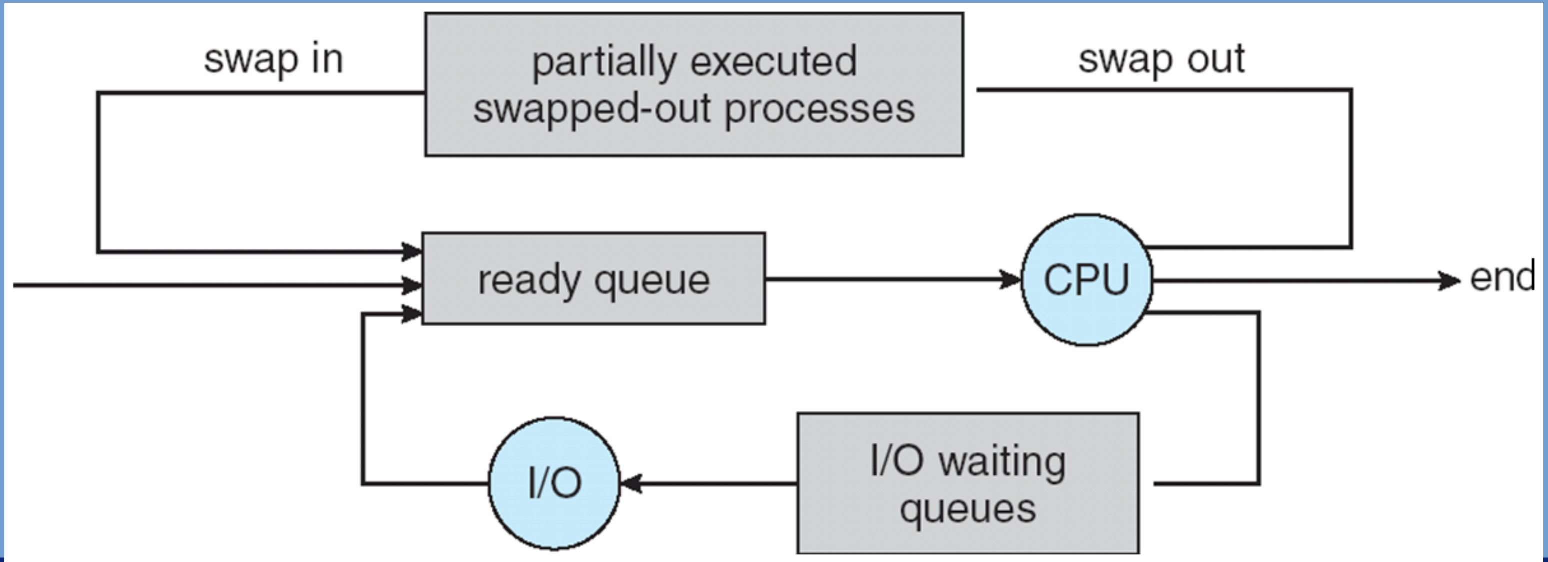
- Within a system different schedulers can be distinguished:
  - **Short-term CPU scheduler:**
    - Present in all timesharing systems.
    - Invoke e.g. when the current process blocks and needs to decide what process from the ready queue is to be assigned to the CPU next.
    - Needs to make a scheduling decision every ~100 ms, so needs to be fast.
  - **Long-term (job) scheduler:**
    - Decides what jobs to be loaded into memory and when.
    - Common in the past and still seen in cluster computer setups: batch job schedulers.
    - Batch jobs run for a long time (hours to weeks/months). Scheduling decisions are infrequent and therefore the algorithm can be more sophisticated / may take more time.
    - Because this scheduler decides how many jobs are brought in memory at the same time, it controls the degree of multiprogramming.

Process Scheduling (6)

- Processes can be characterized as follows:
  - ***I/O bound***: regularly blocking on I/O operations or system calls. These are processes that perform many more system calls compared to computations. Many short CPU bursts.
  - ***Compute bound***: processes that mainly perform computations and not much I/O. These are almost always ready to run and do not spend much time in wait-queues. Few long CPU bursts.
- Note that processes sometimes migrate between different phases:
  - For example a process first reads a lot of data into memory (I/O bound).
  - When the data load is completed, it starts the computation (compute bound).
- To maximize use of the available resources, you want a good *process mix* consisting of both I/O-bound and compute-bound processes.

Process Scheduling (7)

- When a system supports *process swapping*, it can temporarily unload a process from memory and store its state on secondary storage.
  - Frees up main memory, decreases degree of multiprogramming.
- Now a *medium-term scheduler* is necessary to decide what process is unloaded and what process is brought back into memory.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Creation of processes

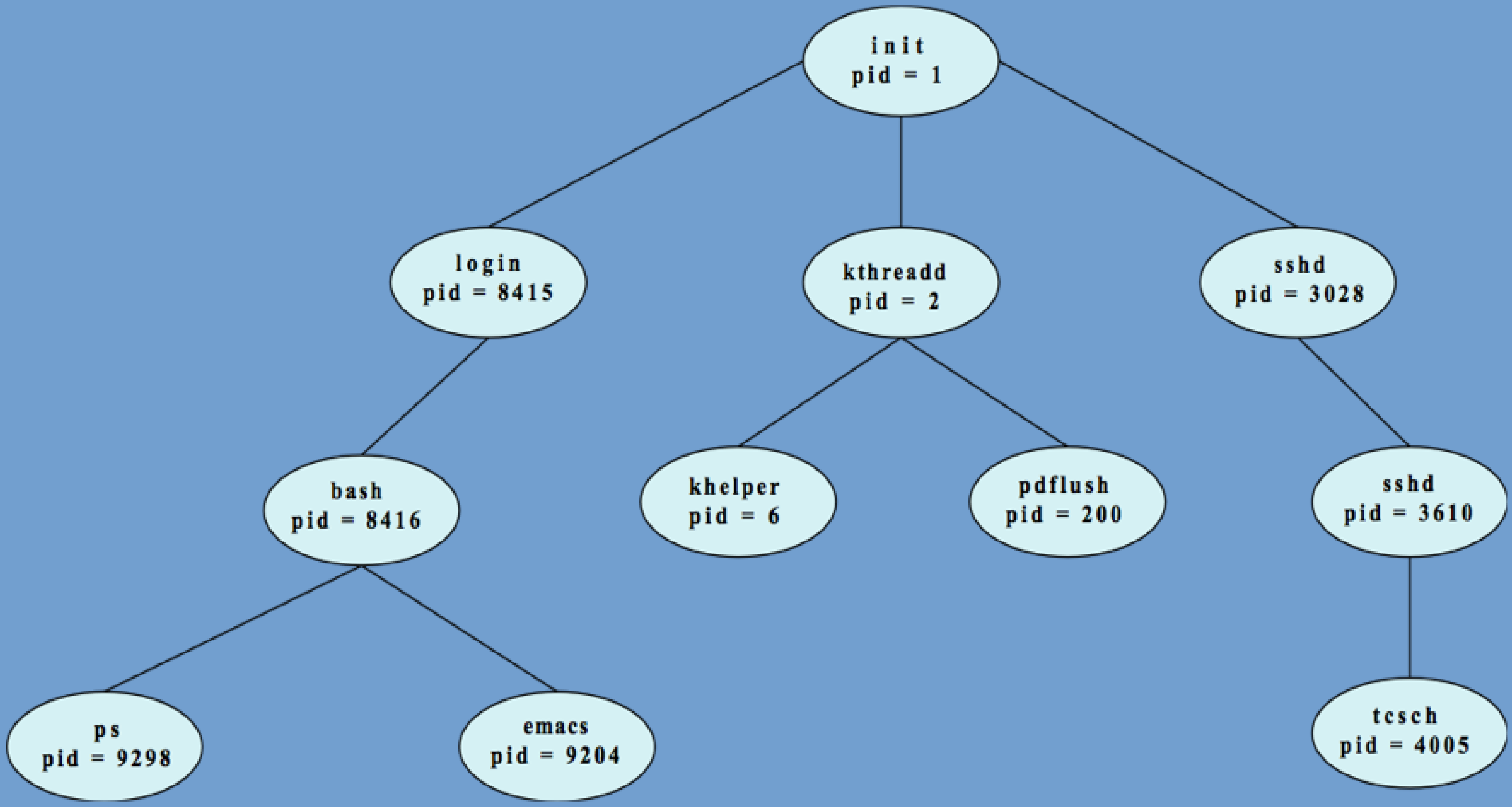
- On creation, each process is given a number: the *process identifier* (short: pid).

- A parent process can create child processes.

  - Who creates the first parent? The kernel does, it creates the first process and loads a program.

  - The children can in turn create processes too, leading to a tree of processes.

An example tree of processes



Source: Silberschatz et al., Operating System Concepts, 9th Edition

Linux “pstree” command output

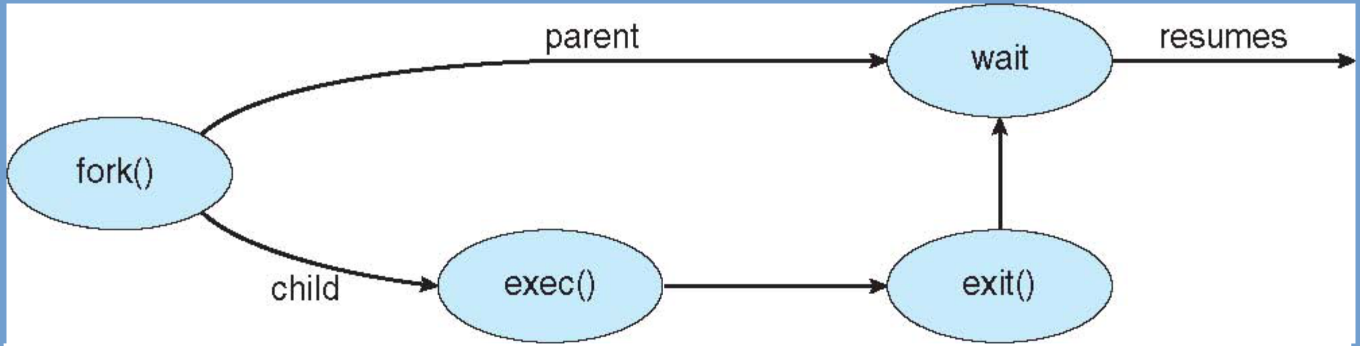
```
$ pstree
init-+-acpid
     |-auditd---{auditd}
     |-automount---4*[{automount}]
     |-avahi-daemon
     |-console-kit-dae---64*[{console-kit-da}]
     |-cron
     |-cupsd
…
     |-rpc.statd
     |-rpcbind
     |-rsyslogd---4*[{rsyslogd}]
     |-screen---2*[tcsh]
     |-ssh-agent
     |-sshd-+-2*[sshd---sshd---bash]
     |      |-sshd---sshd---tcsh---less
     |      |-sshd---sshd---tcsh---telnet
     |      |-sshd---sshd---bash---pstree
     |      `-sshd---sshd
     |-udevd---2*[udevd]
     |-udisks-daemon-+-udisks-daemon
     |               `-{udisks-daemon}
     |-upowerd---{upowerd}
     `-ypbind---2*[{ypbind}]
```

Creation of processes (2)

- Many choices can be made when creating new processes:
  - Should all resources of the parent be shared with the child? Or only a subset, or nothing?
  - Should the parent wait (block) until the child has finished? Or may both processes execute concurrently?
  - What about open files? Network connections?
  - What if the parent terminates while the child is still active?

Process creation on UNIX

- Process creation on UNIX is done through the fork() and execv() system calls.
  - fork() creates a new process and sets up a copy of the parent's address space (so running the **same** program).
  - execv() replaces the executable image loaded into the address space.



Source: Silberschatz et al., Operating System Concepts, 9th Edition

fork() system call

- fork() creates a new process.
  - "The child process is an exact copy of the calling process."
  - "Except for process ID, parent process ID".
- Return value of fork():
  - < 0: operation failed.
  - == 0: returned to the child process.
  - > 0: returned to the parent process, indicates process ID of child.

exec() system call

- exec(): “replace the process image”.
  - Text, data segment, stack, heap.
  - File descriptor state not modified!
- So, for instance, load the program “/bin/ls” in memory.

Process creation on UNIX (2)

```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;
}
```

Source: Silberschatz et al., Operating System Concepts, 9th Edition

Process Termination

- Process termination is invoked either voluntarily or involuntarily.
  - **Voluntarily**: process performs `exit()` system call.
    - OS kernel will deallocate all resources held by this process and free the task struct.
    - If a parent process was waiting (`wait()` system call) it is informed of the termination and the return value (status code) is communicated.
      - If no parent is waiting the process becomes a *zombie process* until it is cleaned up by the parent.
  - In many systems return from `main` will return to a special routine in the startup code (e.g. `_start`) from which the `exit()` system call will be performed.

Process Termination (2)

- Process termination is invoked either voluntarily or involuntarily.
  - **Involuntarily**: a parent process request a child to be terminated. This can be done using the `kill()` system call.
  - Some reasons for doing so:
    - Task is no longer needed (user quit the program).
    - Task is behaving incorrectly (when debugging).
    - Task has exhausted assigned/admitted resources.
    - The parent is exiting (or being involuntarily terminated itself) and the system does not support child processes without a parent to continue execution (cascading termination)
      - If supported, a child without parent is called an orphan.

Interprocess Communication

- Processes are either:
  - **Independent**: cannot affect or be affected by other processes in the system.

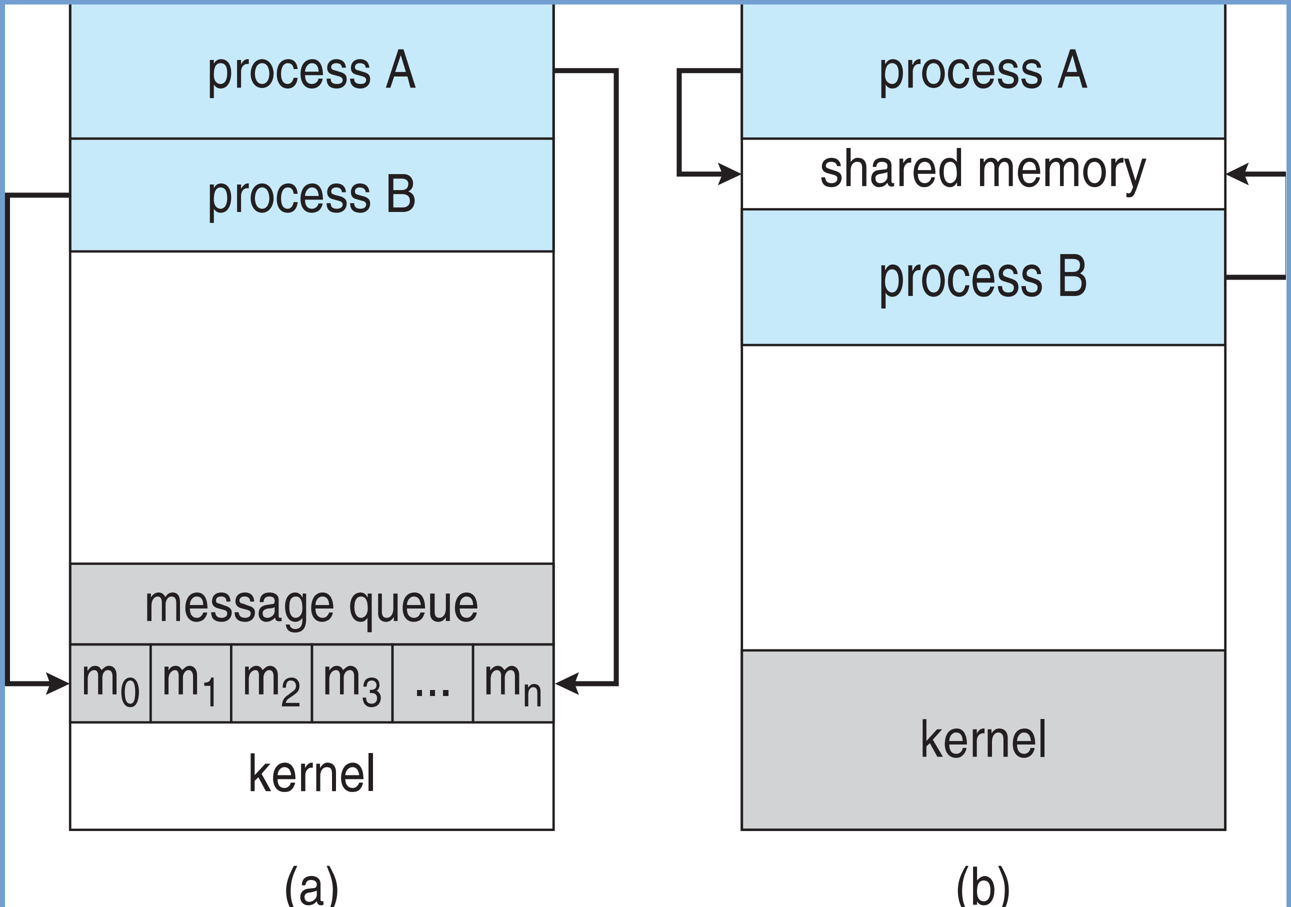
    Example: process which does not share data with any other process. The control flow of this process cannot be influenced by other processes.
  - **Cooperating**: the opposite, so a process that can affect / be affected by others. Broad consequences: any process that shares data is cooperating.
- Communication between cooperating processes is required:
  - Information sharing: control concurrent access to files.
  - Computation speedup: divide the work, merge the results.
  - Modularity: communication between modules (e.g. pipelining).
- Processes may exchange information through Interprocess Communication (IPC) mechanisms.
  - Two important models are: ***shared memory*** and ***message passing.***

IPC examples

- UNIX pipelines: process A sends data to B through a pipe. A pipe can be seen as IPC mechanism.
  - Example of a producer – consumer system.
- Modern web browser implementation:
  - In the past web browsers were a single process: if a tab crashed, the entire browser crashed.
  - These days a separate process per tab, if a tab crashes, only that tab crashes.
  - Tab processes communicate with the master process through IPC mechanisms.
- Apache web server can start multiple processes to serve incoming requests; takes advantages of multi-processor systems.

Interprocess Communication (2)

Two models: *message passing* (a) vs. *shared memory* (b)



Source: Silberschatz et al., Operating System Concepts, 9th Edition

IPC: shared memory

- Idea: allocate a block of memory that is accessible by multiple processes.
  - How this can be done with respect to isolation will be discussed in a later chapter.
- Processes can then communicate through this shared memory.
  - Who writes what and where? This is all under the control of the processes themselves, without OS kernel involvement.
  - What if multiple processes write to the same location at the same time?
    - Not the problem of the OS kernel.
    - The OS kernel does provide mechanisms to help with this: Synchronization primitives which are covered in Chapter 5.

IPC: Message Passing

- Idea: provide system calls to send and receive messages.
  - No shared memory needed.
  - Because system calls are used, the actual copying of the data from one process to the other is performed by the OS kernel.
  - Besides communication, message passing is also used for process synchronization.
- Typically two calls are present:
  - `send(message)`
  - `recv(message)`
  - `message` is either fixed size or variable-size.

IPC: Message Passing (2)

- Various choices can be made when providing message passing primitives:
  - Direct vs. indirect communication
  - Synchronous vs. asynchronous communication
    - Blocking vs. non-blocking
  - Bounded vs. unbounded buffers

IPC: Direct Communication

- With direct communication the sender must explicitly name the recipient, and the recipient must name the sender:
  - `send(P, message)` send message to P
  - `recv(Q, message)` receive message from Q
- This results in a communication link with the following properties:
  - Matching send/receive calls automatically establish a link
  - A link always consists of (exactly one) pair of processes
  - The link may the unidirectional as well as bi-directional

IPC: Indirect Communication

- In this case processes do not name each other explicitly, but communication is done (indirectly) through a mailbox.
- The mailbox has an ID. Processes can only communicate when they share the mailbox with the same ID.
  - Note that more than two processes can take part in this communication.
- Properties of the communication link:
  - A link is established once processes share a common mailbox.
  - More than two processes may be associated with a link.
  - A pair of processes can communicate through more than one mailbox.
  - Again the link may be unidirectional as well as bi-directional.

IPC: Indirect Communication (2)

- Example:
  - `create(A)` create a mailbox A
  - `send(A, message)` send a message to mailbox A
  - `recv(A, message)` receive a message from mailbox A
- Problem!!
  - L and M are trying to receive a message from A. K sends a single message to A. Who receives the message?

IPC: Indirect Communication (2)

- Example:
  - `create(A)` create a mailbox A
  - `send(A, message)` send a message to mailbox A
  - `recv(A, message)` receive a message from mailbox A
- Problem!!
  - L and M are trying to receive a message from A. K sends a single message to A. Who receives the message?
  - Implementor must choose (and make clear in documentation):
    - Allow at most two processes to be associated with a mailbox.
    - Allow at most one process to perform a recv() on a mailbox at the same time.
    - The OS kernel arbitrarily choses a recipient.

IPC: Synchronization

- *Synchronous* or *blocking* communication:
  - With a *blocking send*, the sender blocks until the recipient has received the message (using a recv() call).
  - *Blocking receive*: block until a sender sends a message.
  - By pairing blocking send with blocking receive a synchronization primitive can be built: rendezvous messaging.
    - A process can only continue execution from the rendezvous point if the other process has reached that point as well. So they must meet before either can continue.

IPC: Synchronization (2)

- On the other hand we have *asynchronous* or *non-blocking* communication:
  - *Non-blocking send*: send the message and continue.
  - *Non-blocking receive*: try to receive, if a message is waiting then this message is received otherwise an empty message.
    - Often associated with a timeout: wait for a period of time, if no message comes in, return an empty message.
- Some systems support various combinations, you can perform non-blocking sends and blocking receives, and so on.

IPC: Buffering

- In the case of non-blocking communication, the OS kernel must buffer the messages.
- Three options:
  - *Unbounded buffer*: the buffer is "unlimited" in size (of course until system memory is full).
  - *Bounded buffer*: the buffer has a fixed set. A non-blocking send to a full buffer is turned into a blocking send (or send failure).
  - *Zero capacity buffer* (or no buffer): in this case send and receive calls must match up (rendezvous messaging).

IPC across the network

- Naturally, IPC can be extended to involve processes running on different systems.

- These systems may even run different operating systems, as long as they agree on a set of (network) protocols.

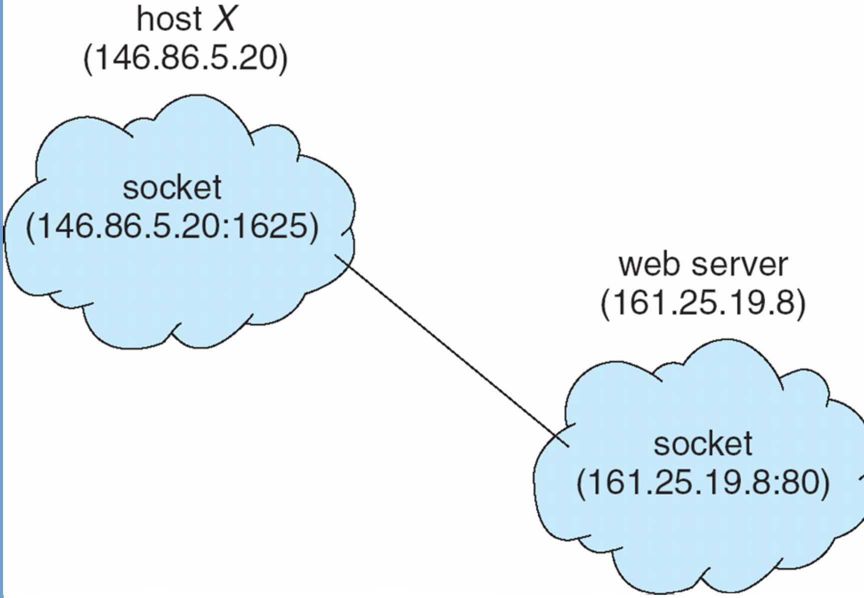
- Low-level network communication is done using the TCP/IP and UDP/IP protocols.

  - Other network protocols are built on top of this: HTTP, SMTP, SSH, IRC.

  - See also the bachelor course “Netwerken”, 3rd year.

IPC across the network (2)

- Networking is typically defined in terms of sockets.
  - A socket is a communication endpoint. A connection can be “plugged in”.
  - It consists of an IP address and a port number (2 bytes).
  - You can either have a listening socket or connecting socket. You can use a connecting socket to connect to a listening socket. In case of TCP/IP, a reliable connection between the two sockets is formed.
  - UDP is a datagram protocol and does not support the notion of established connections.

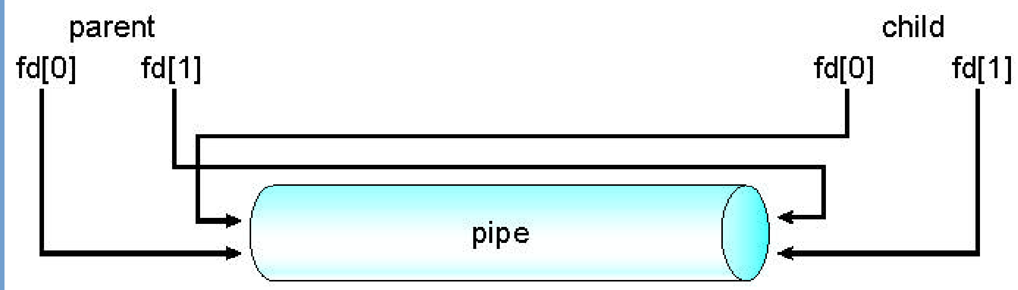


Source: Silberschatz et al., Operating System Concepts, 9th Edition

Remote Procedure Calls

- Easy way to do IPC over the network.
- Instead of local procedure call, call a function on a different machine.
- Transfer of function arguments, return value over the network all handled for you.
- Structured messages, structure already defined.
- Also frequently used to implement “web services”: XML-RPC, SOAP, JSON-RPC.

Pipes



Source: Silberschatz et al., Operating System Concepts, 9th Edition

- Pipes are commonly used as a local IPC mechanism.
- *Ordinary pipes* support producer-consumer communication and provide a *unidirectional* link.
  - Everything that is written to the *write-end* of the pipe and be read from the *read-end*.
  - An ordinary pipe only exists within the process in which it was created.
  - How to use a pipe with multiple processes? We fork! This duplicates the parent process **including** any pipes that were created. Parent can write to write-end, child can read from read-end.
  - Implication: parent-child relationship required.
  - In Windows systems these are referred to as *anonymous pipes*.

Named Pipes

- Next to ordinary/anonymous pipes, some systems also support named pipes.
- These pipes are accessible through a file created on the file system.
- More than two processes can access this pipe.
- Communication is bidirectional.
- Named pipes are for example used to communicate with a database daemon (DBMS) that is running on the local machine.
  - In such a case, we do not have to wrap all our queries (and results) in TCP/IP packets.

End of Chapter 3.