

Networking Operating Systems (CO32010)



2. Processes and scheduling



- 2.1 Introduction
- 2.2 Scheduling
- 2.3 Higher-level primitives
- 2.4 Signals, pipes and task switching
- 2.5 Messages
- 2.6 Microsoft Windows scheduling
- 2.7 UNIX process control
- 2.8 Finite-state machines

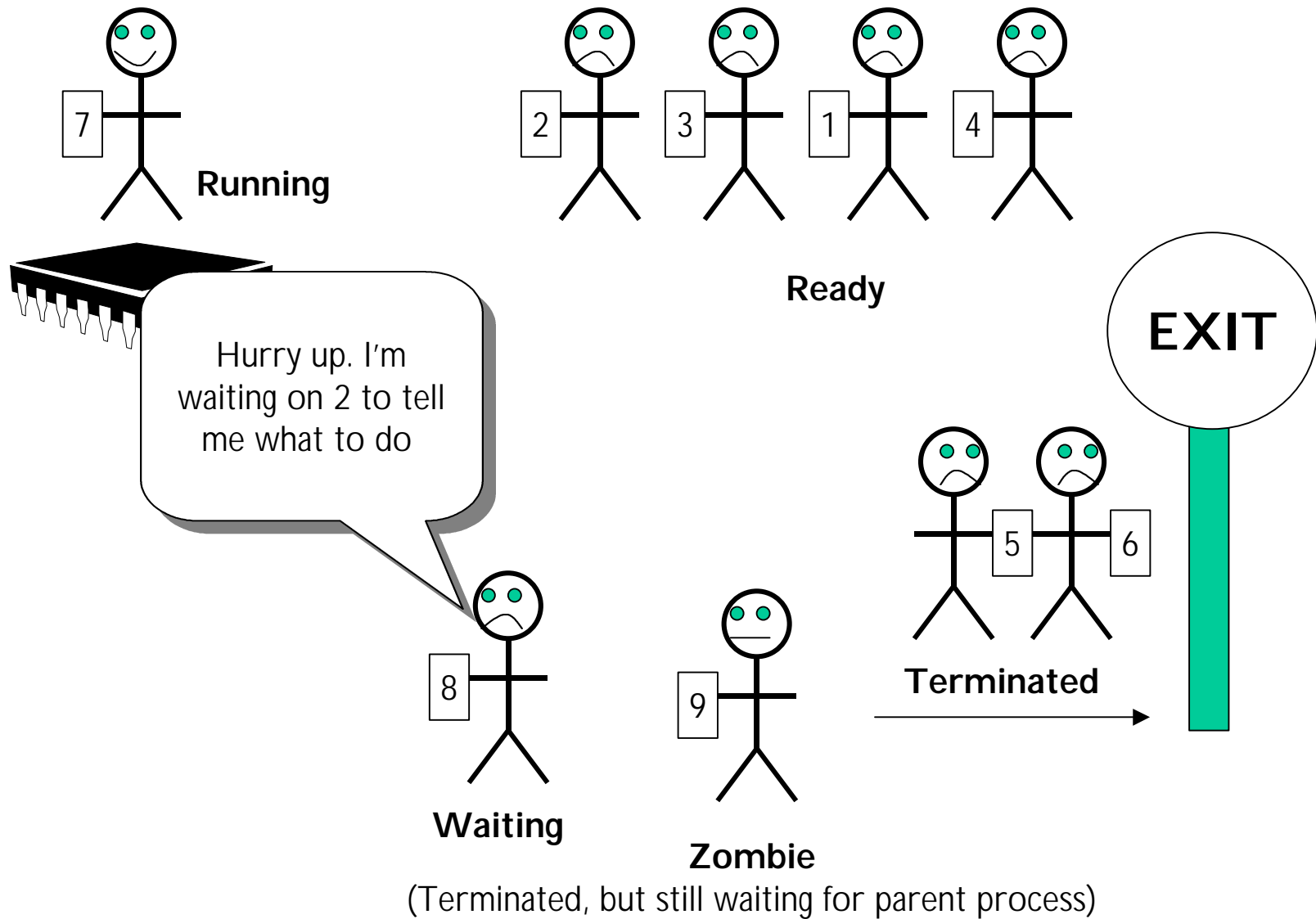
3. Distributed processing



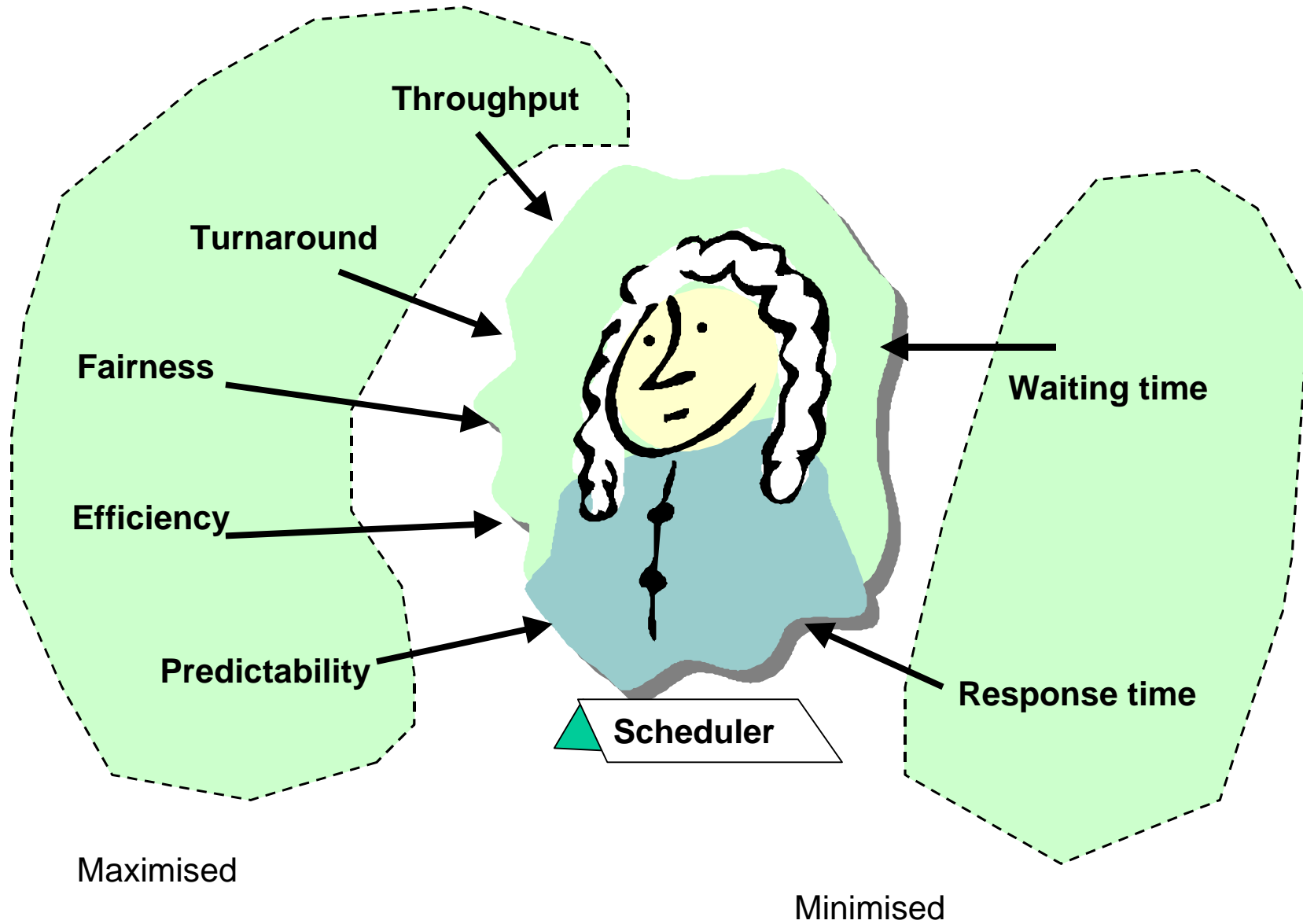
Objectives:

- To define the main parameters used in scheduling.
- To define some of the main scheduling technique and be able to contrast them.
- To briefly define the usage of parallel processing.
- To outline the usage of high-level primitives, such as signals, pipes and task-switching.
- To give examples of practical process control.

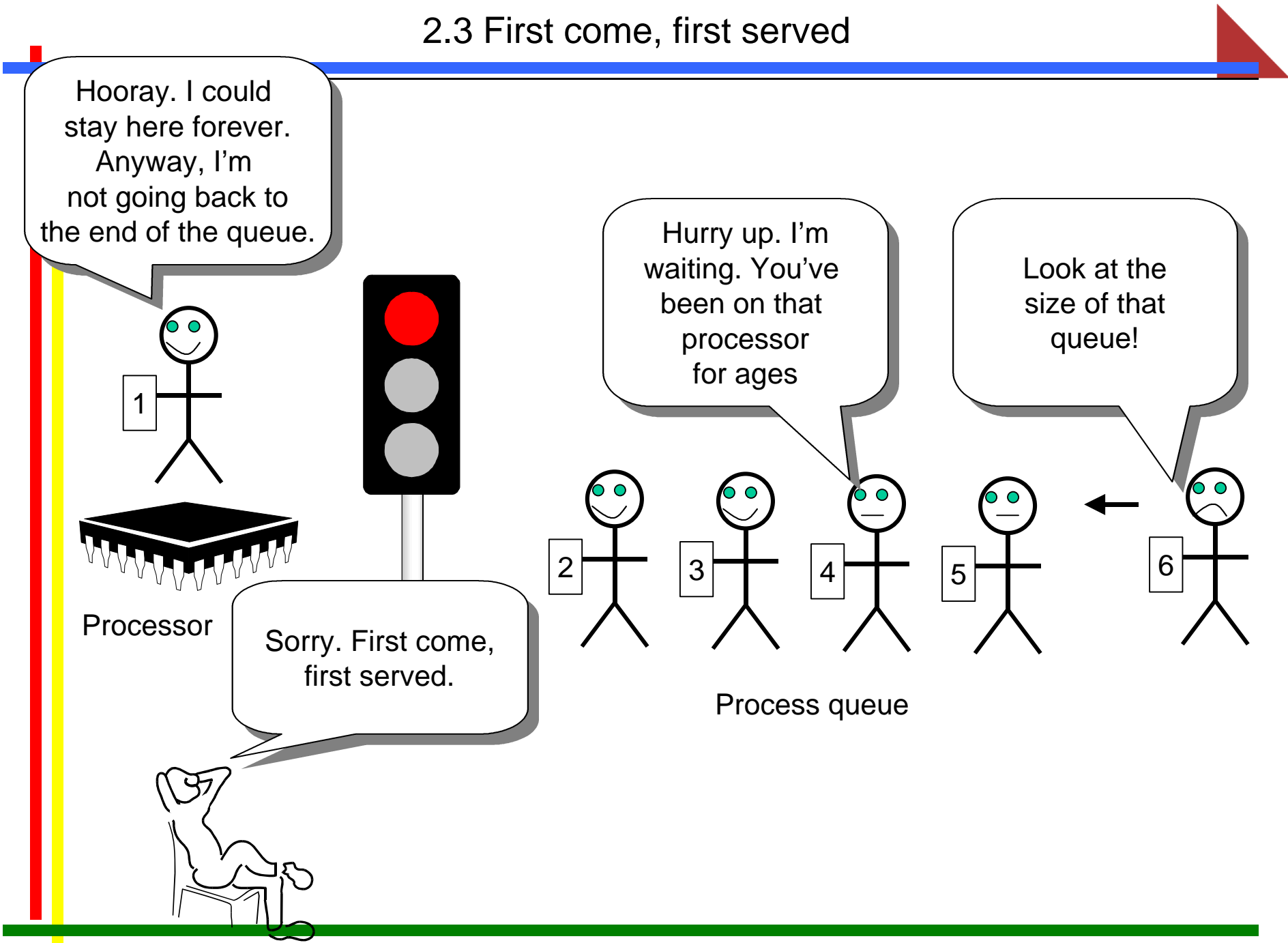
2.1 Running, Ready, Waiting and Terminated



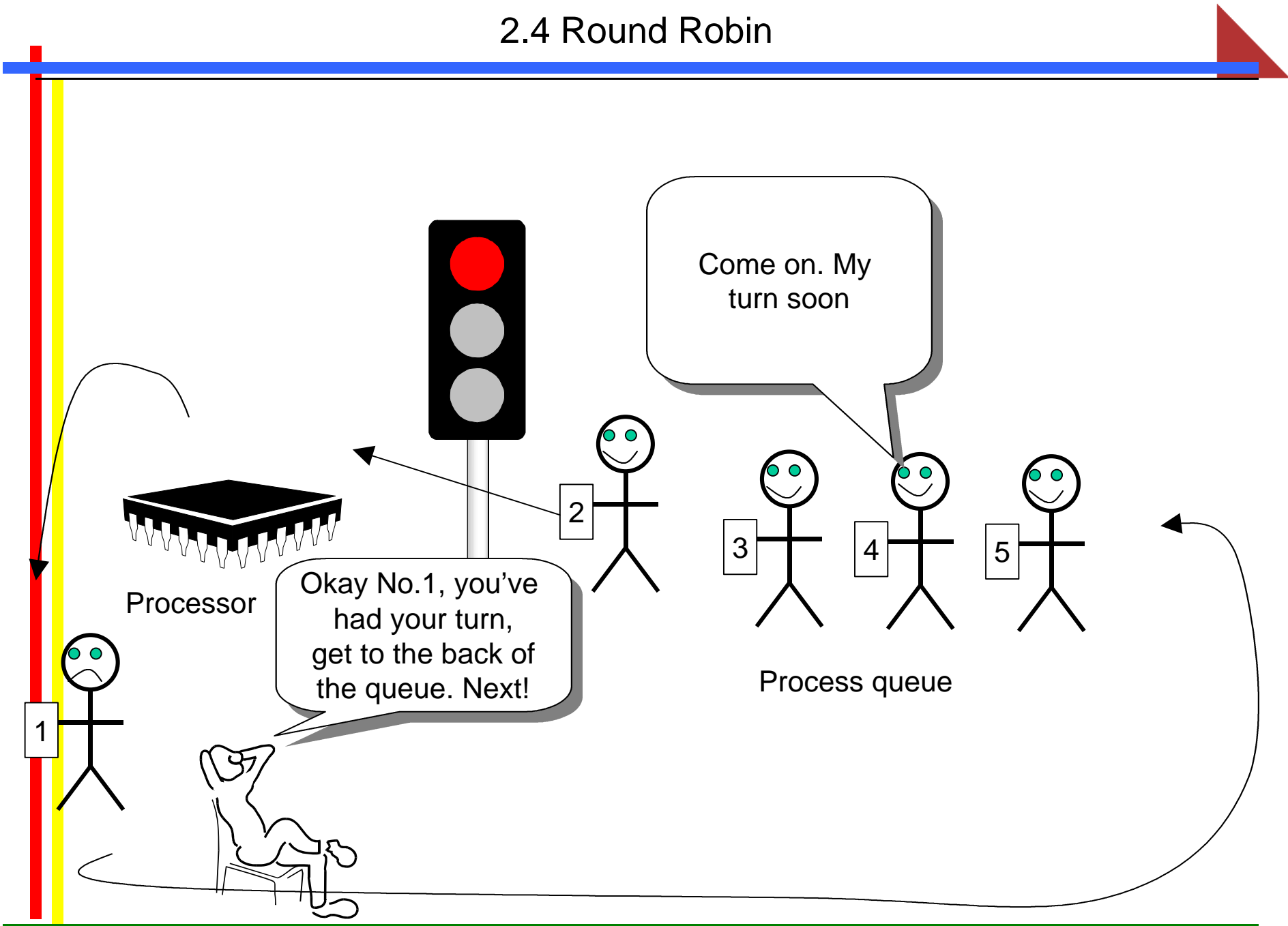
2.2 Decisions for the Scheduler



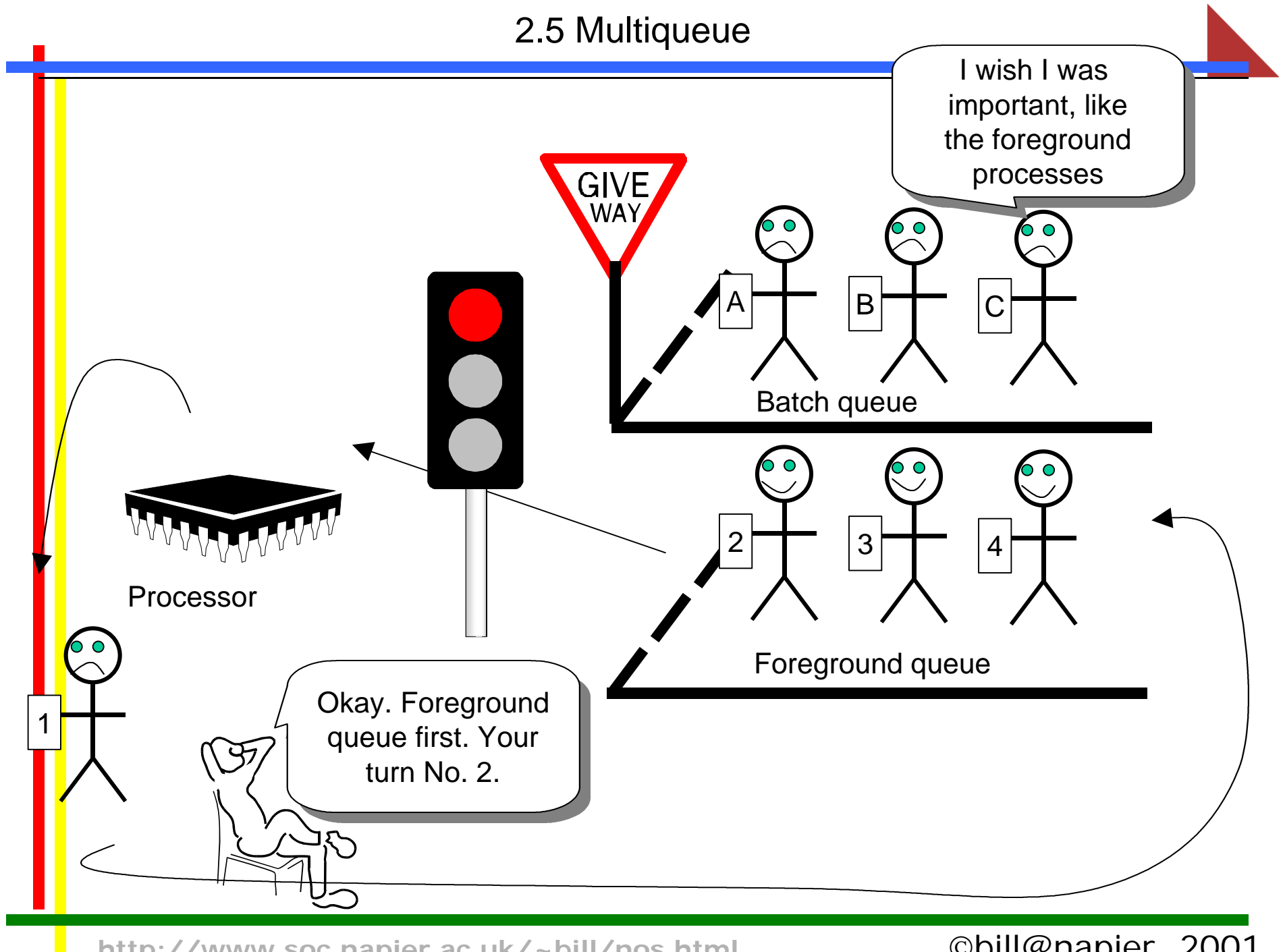
2.3 First come, first served



2.4 Round Robin

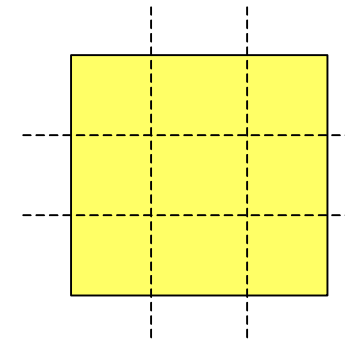
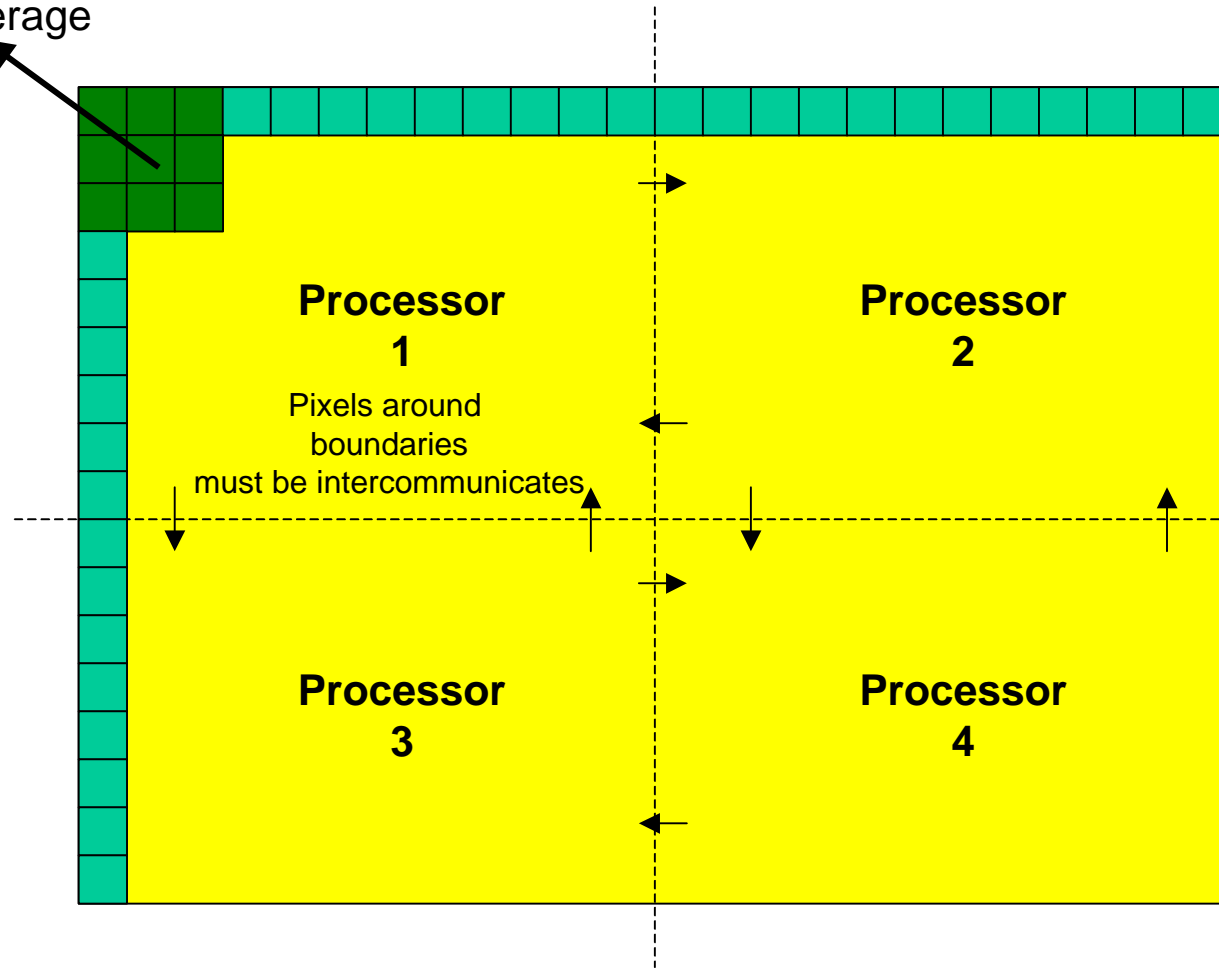


2.5 Multiqueue

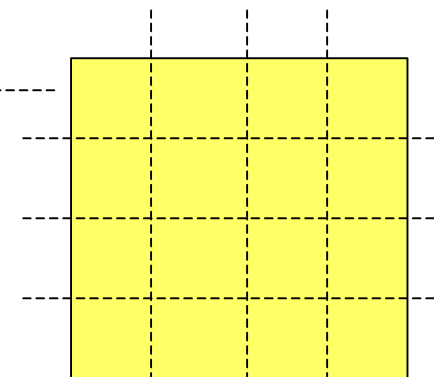


2.6 Inter-communication between processors

Find average



3x3 processing array
4 interfaces

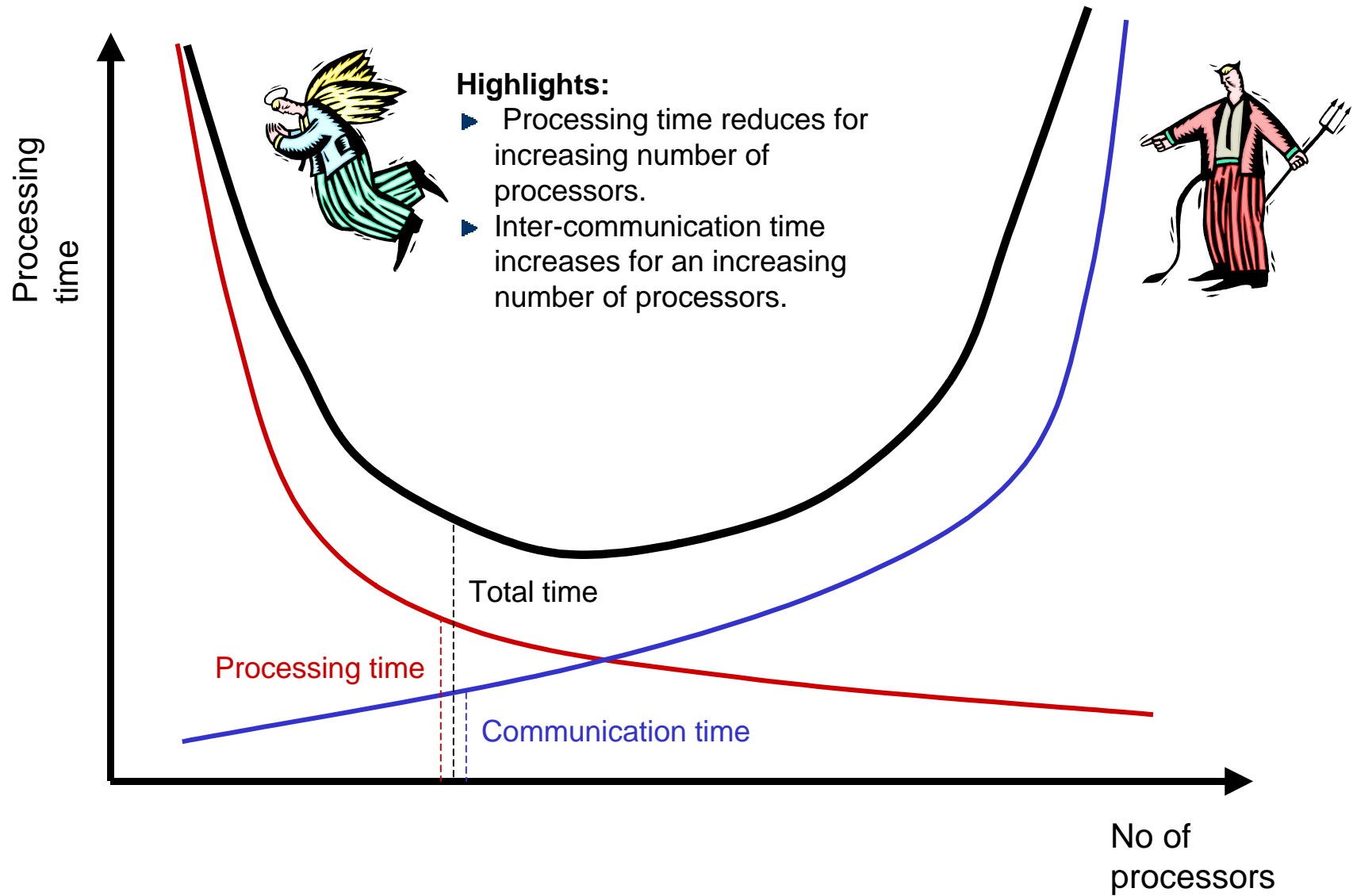


4x4 processing array
6 interfaces

2x2 processing array
2 interfaces

$n \times n$ processing array
 $2(n-1)$ interfaces

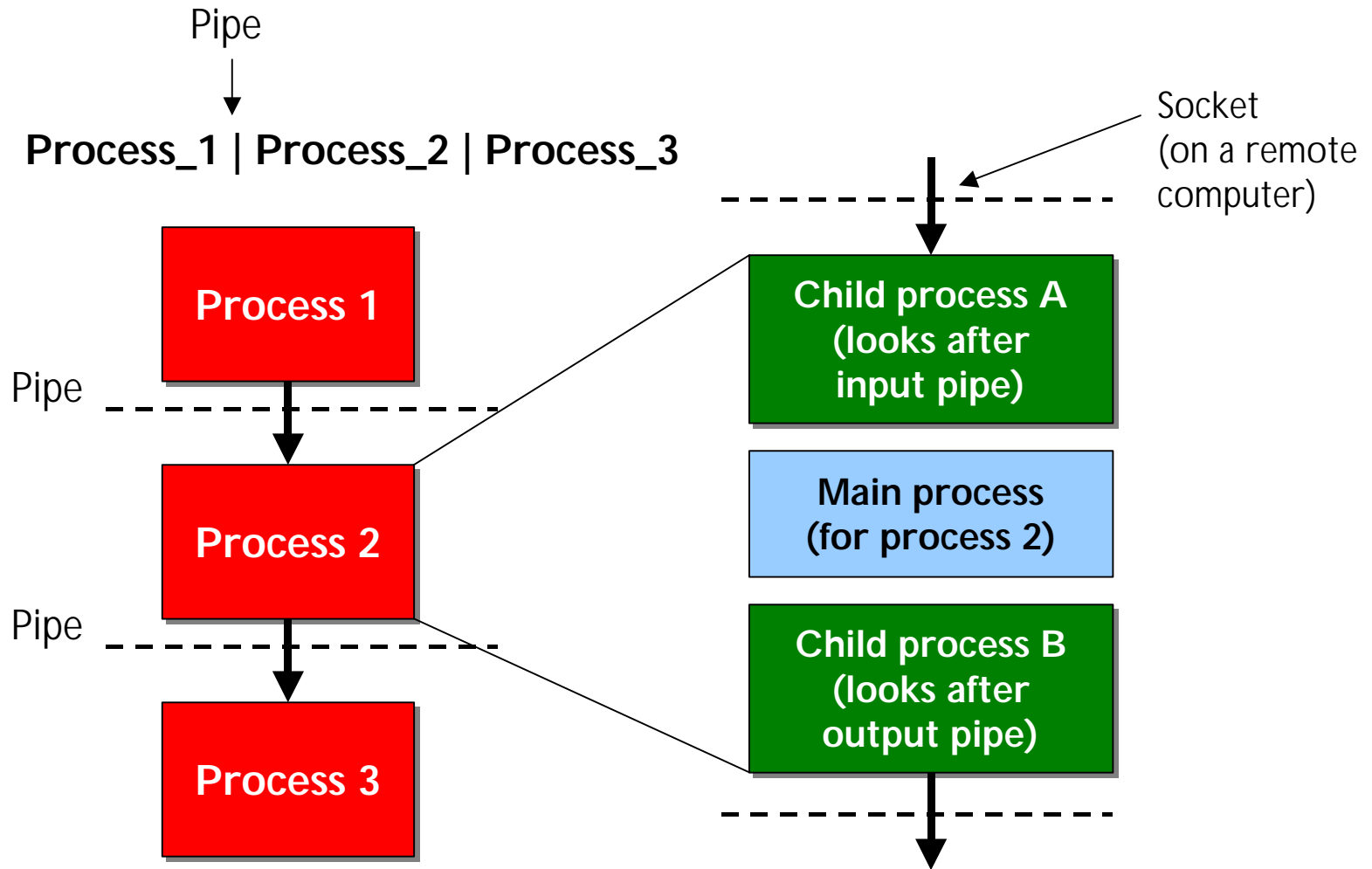
2.7 Communication time and processing time



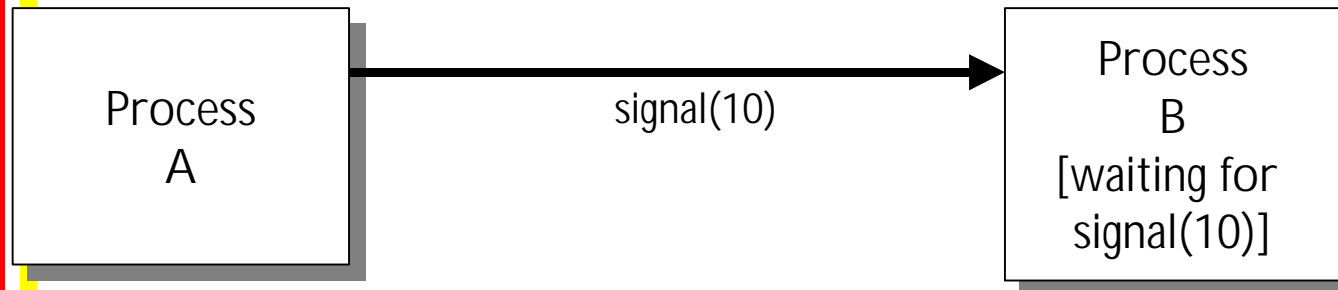
2.8 Semaphores, signals and pipes

- ▶ **Semaphores.** This involves setting flags, which allow or bar other processes from access certain resources. An analogy of a semaphore is where two railway trains are using a single-track railway line. When one train enters the single-track line, it sets a semaphore which disallows the other train from entering the track. Once the train on the single track has left the single track, it resets the semaphore flag, which allows the other train to enter the single track.
- ▶ **Signals.** Signals are similar to interrupts, but are implemented in software, rather than hardware. This is a primitive interrupt handler and involves a signal handler which controls process signals.
- ▶ **Pipes.** Pipes allow data to flow from one process to another, in the required way. Typically they are implemented with a fixed size storage area (a buffer) in which one process can write to it, while the other reads from it (when the data is available). UNIX implements pipes with a file-like approach, and uses the same system calls to write data to a pipe and read data from a pipe as those for reading and writing files. Each process which creates a pipe receives two identifiers: one for the reading and one for the writing. Typically, the creating process forks-off two child processes, one of which looks after one end of the pipe, and the other on the other end of the pipe. The two child processes can then communicate.

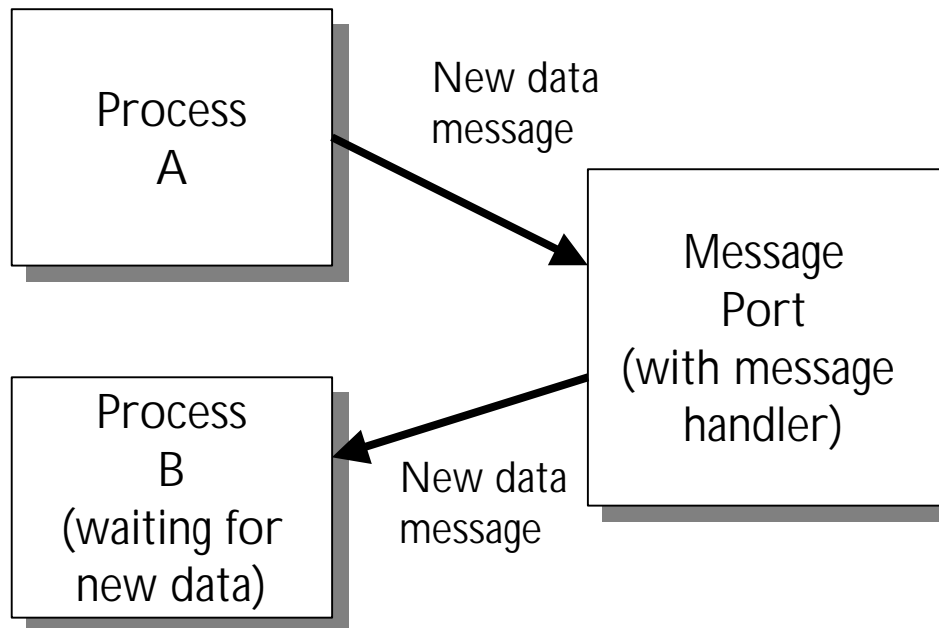
2.9 Pipes (in Unix)



2.10 Signals .v. Messages



Signals
(a simple method which allows processes to identify events)



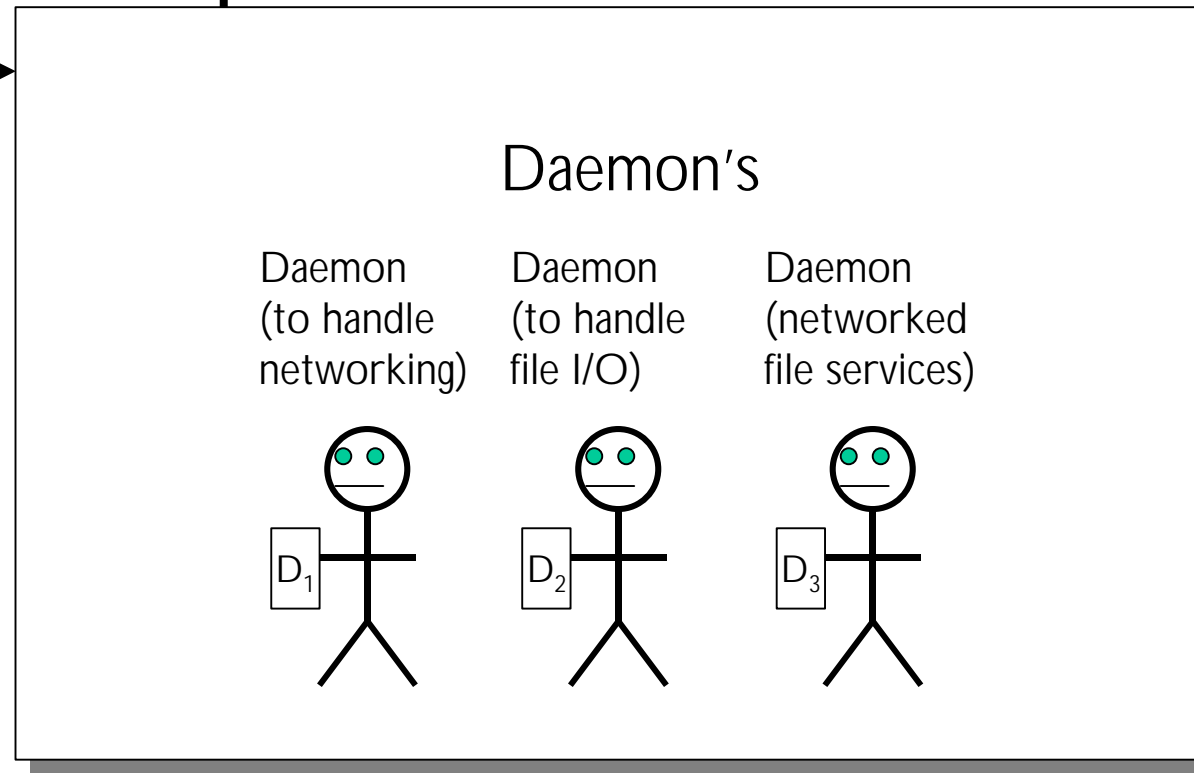
Message
(an improved method where an indication of the actual event can be passed between processes)

2.11 Daemon's

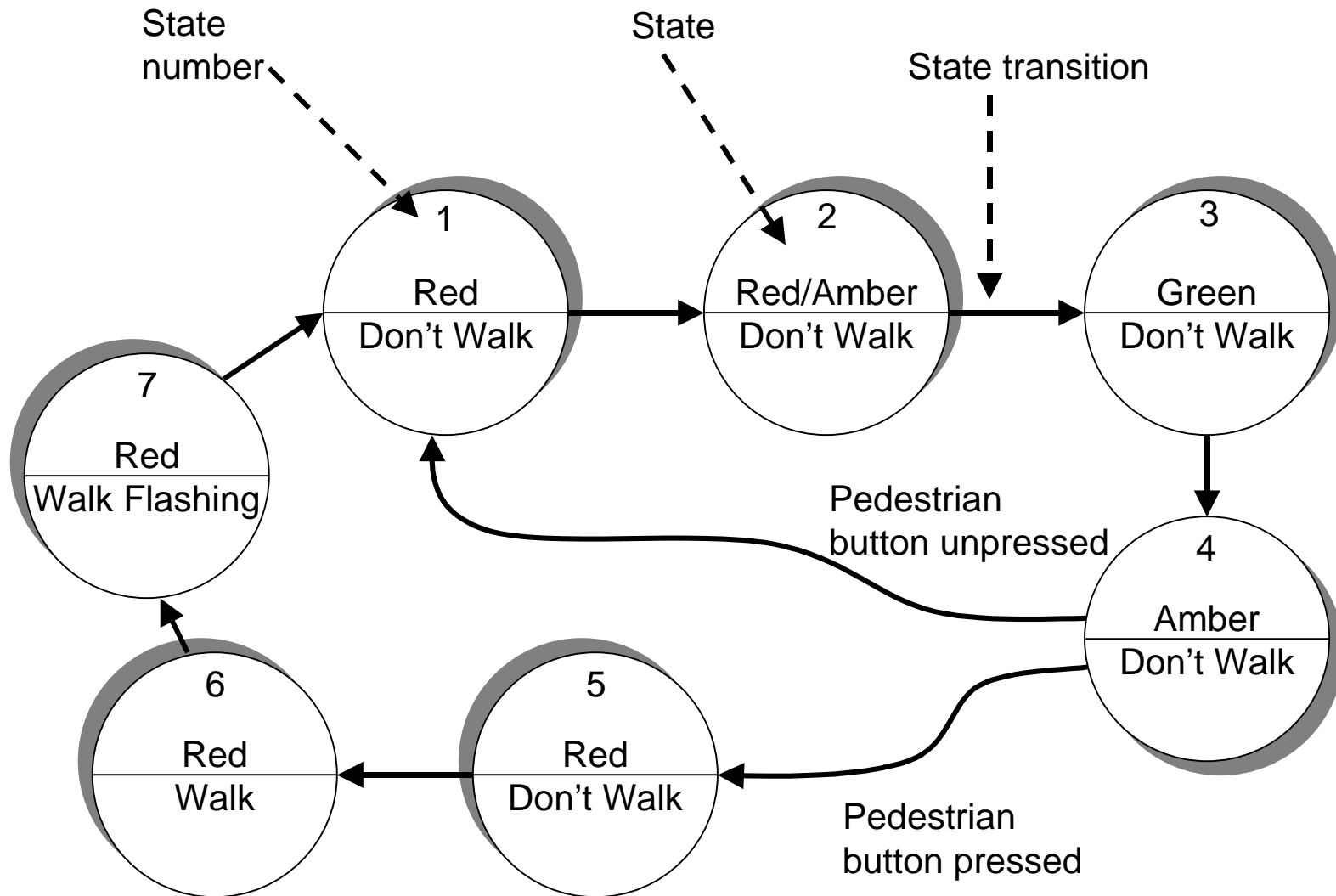
Unix start-up
(reads **rc** file
and initiates
daemon processes)

Single-user
Mode (only
system administrator
can login)

Multi-user
mode (all
users can login)



2.12 Traffic Light Sequence



2.13 Traffic Light Sequence

