Distributed Systems

Winter Term 2024/25

Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 28, 2024

Distributed Systems

Winter Term 2024/25

5 Process Management

Contents

- Distributed process scheduling
- Code migration

Literature

- Tanenbaum, van Steen: Ch. 3
- Stallings: Ch. 14.1

5.1 Distributed Process Scheduling

- Typical: middleware component that
  - decides on which node a process is executed
  - and probably migrates processes between nodes
- Gloals:
  - balance the load between nodes
  - maximize the system performance (average response time)
    - also: minimize the communication between nodes
  - meet special hardware / resource requirements
- Load: typically the length of the process queue (ready queue)
  - sometimes resource consumption and communication volume are considered, too

Approaches to distributed scheduling

- Static scheduling
  - mapping of processes to nodes is defined before execution
  - NP-complete, therefore heuristic methods
- Dynamic load balancing, two variants:
  - execution location of a process is defined during creation and is not changed later
  - execution location of a process can be changed at runtime (several times, if necessary)
    - preemptive dynamic load balancing, **process migration**

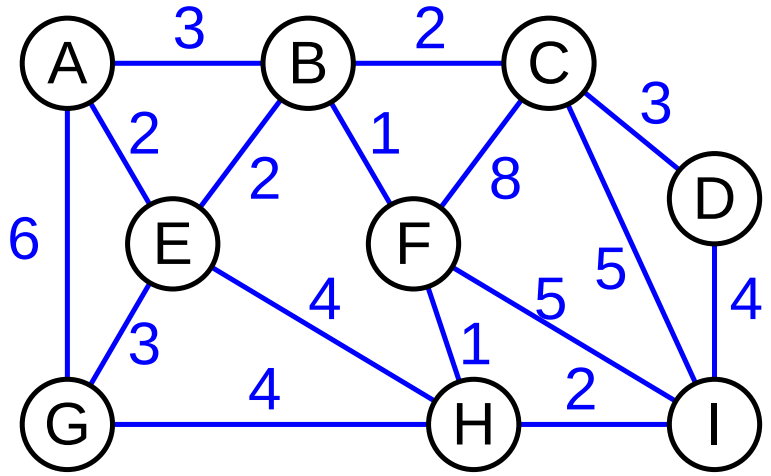
5.1.1 Static Scheduling

- Procedure dependent on the structure / the modelling of a job
  - jobs always consist of several processes
  - differences in communication structure
- Examples:
  - communicating processes: graph partitioning
  - non-communicating tasks with dependencies: list scheduling

Scheduling through graph partitioning

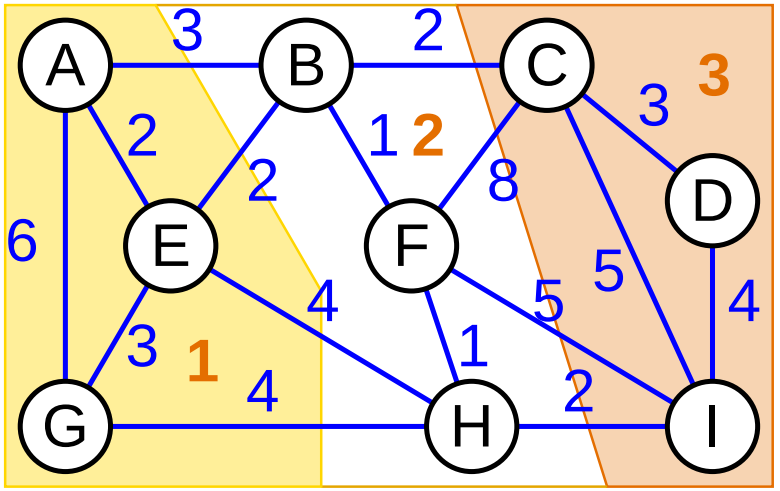
- Given: process system with
  - CPU / memory requirements
  - specification of communication load between each pair of processes

  usually represented as a graph



- Wanted: partitioning of the graph in such a way that
  - CPU and memory requirements are met for each node
  - partitions are about the same size (load balancing)
  - weighted sum of cut edges is minimal
    - i.e. as little communication as possible between nodes
- NP-complete, therefore many heuristic procedures

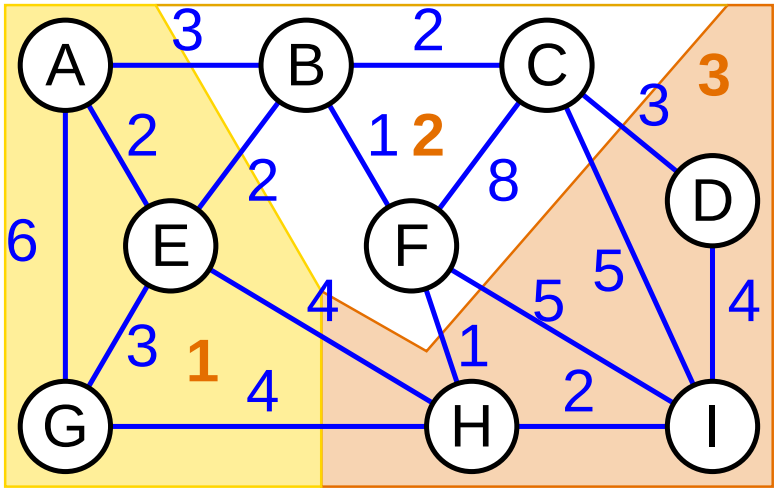
Scheduling through graph partitioning



- Given: process system with
  - CPU / memory requirements
  - specification of communication load between each pair of processes

  usually represented as a graph
- Wanted: partitioning of the graph in such a way that
  - CPU and memory requirements are met for each node
  - partitions are about the same size (load balancing)
  - weighted sum of cut edges is minimal
    - i.e. as little communication as possible between nodes
- NP-complete, therefore many heuristic procedures

Scheduling through graph partitioning

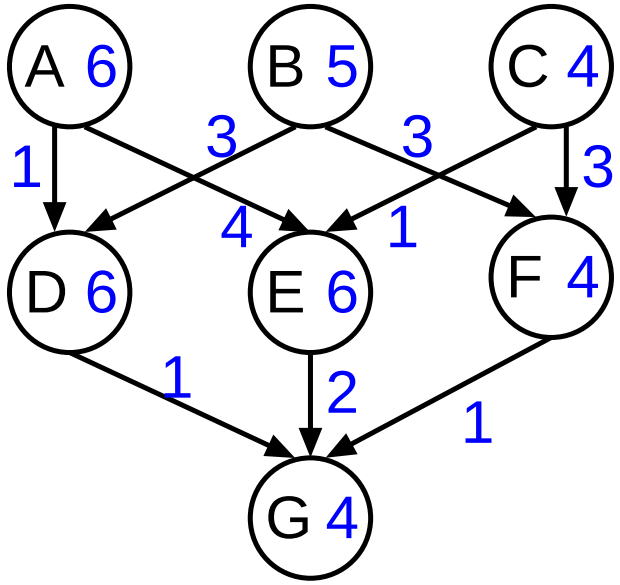


- Given: process system with
  - CPU / memory requirements
  - specification of communication load between each pair of processes

  usually represented as a graph
- Wanted: partitioning of the graph in such a way that
  - CPU and memory requirements are met for each node
  - partitions are about the same size (load balancing)
  - weighted sum of cut edges is minimal
    - i.e. as little communication as possible between nodes
- NP-complete, therefore many heuristic procedures

List scheduling

- Tasks with dependencies, but without communication during execution
  - tasks work on results of other tasks
- Modelling
  - program represented as a DAG
  - nodes: tasks with execution times
  - edges: communication with transfer time



Distributed Systems

Winter Term 2024/25

21.11.2024

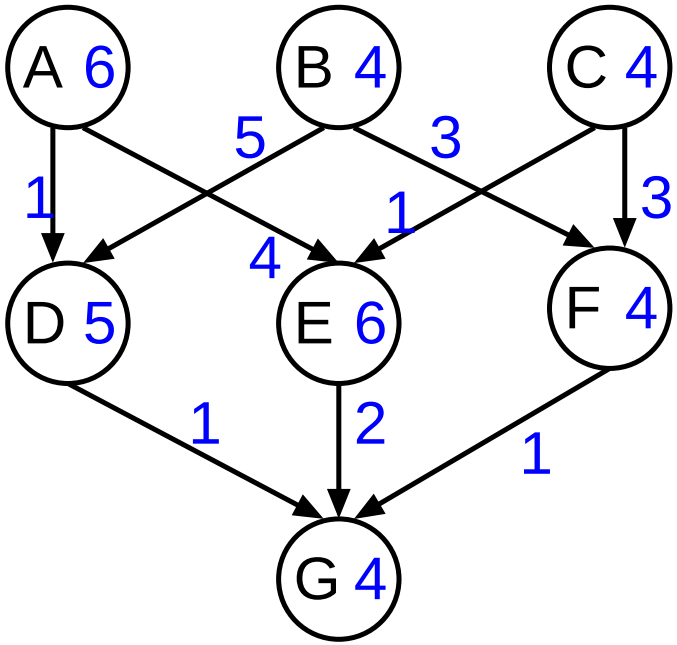
Roland Wismüller
Universität Siegen
roland.wismueller@uni-siegen.de
Tel.: 0271/740-4050, Büro: H-B 8404

Stand: November 28, 2024

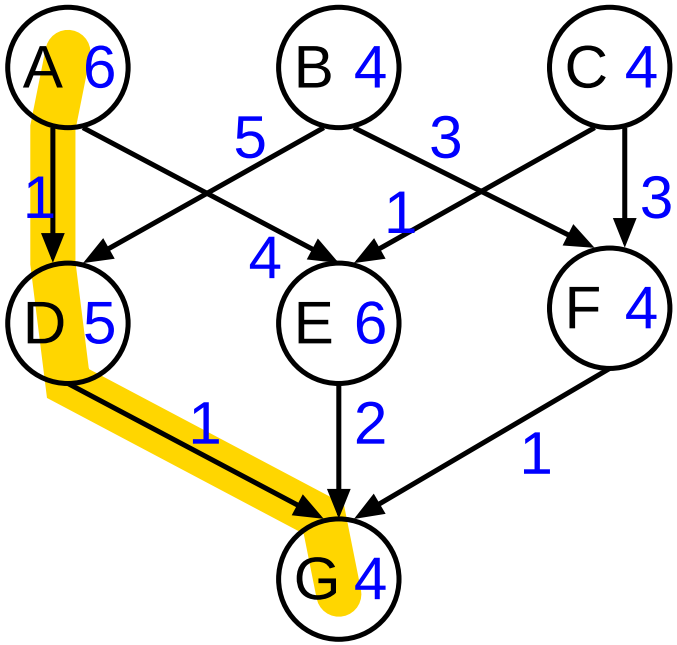
Method

- Create prioritized list of all tasks
  - many different heuristics to determine the priorities, e.g. according to:
    - length of the longest path (without communication) from the node to the end of the DAG (*High Level First with Estimated Time*, HLFET).
    - earliest possible start time (*Earliest Task First*, ETF)
- Process the list as follows:
  - assign the first task to the node that allows the earliest start time
  - remove the task from the list
- Creation and processing of the list can also be interleaved

Example: List Scheduling with HLFET

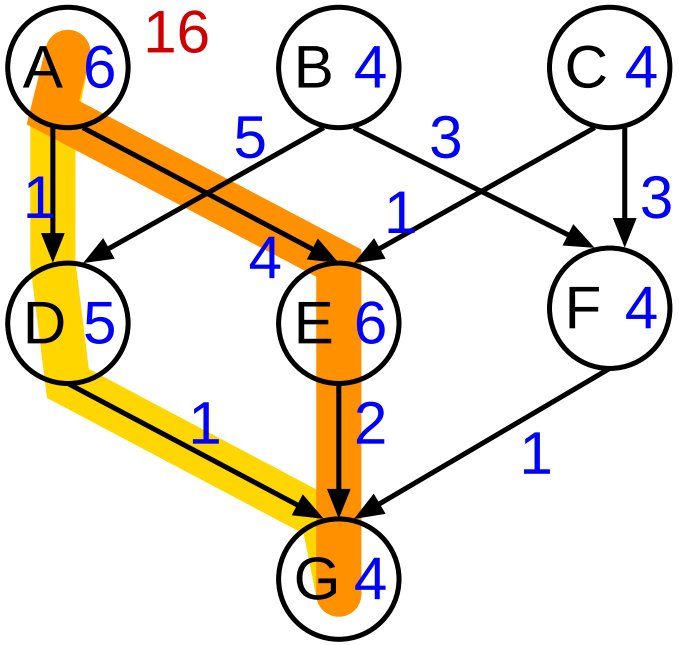


Example: List Scheduling with HLFET



Static level (without comm.):
6+5+4 = 15

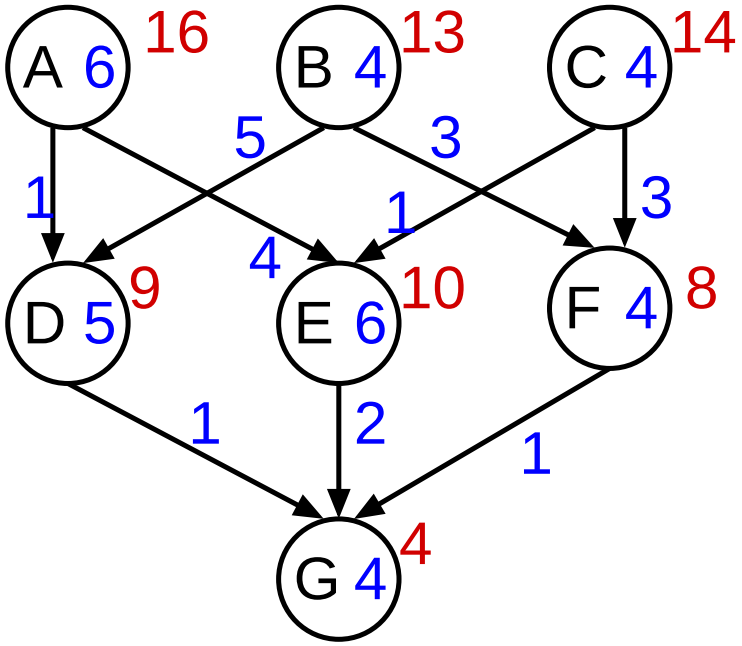
Example: List Scheduling with HLFET



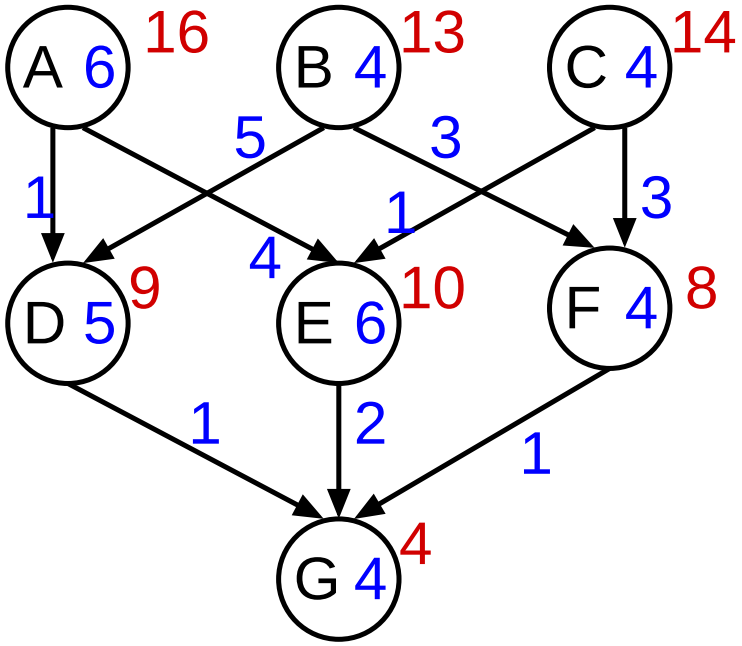
Static level (without comm.):

6+5+4 = 15 6+6+4 = 16

Example: List Scheduling with HLFET



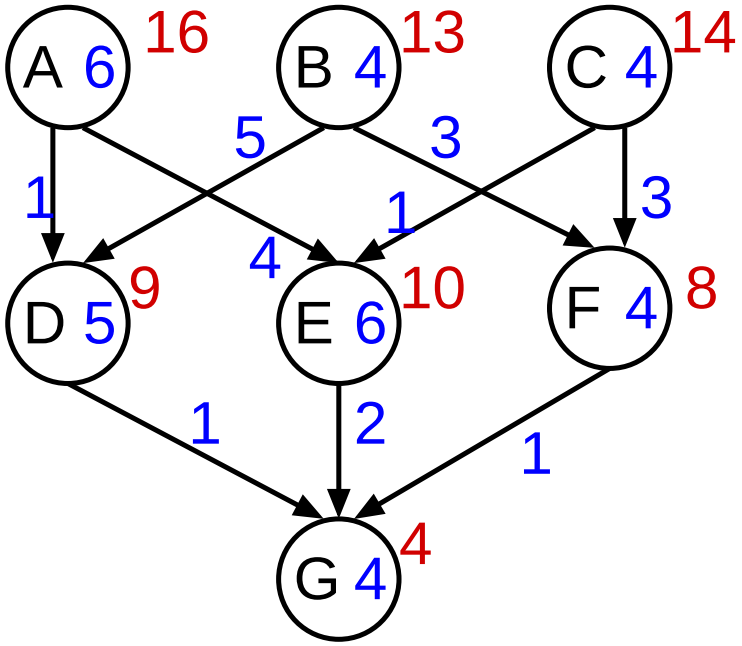
Example: List Scheduling with HLFET



List:

A	C	B	E	D	F	G

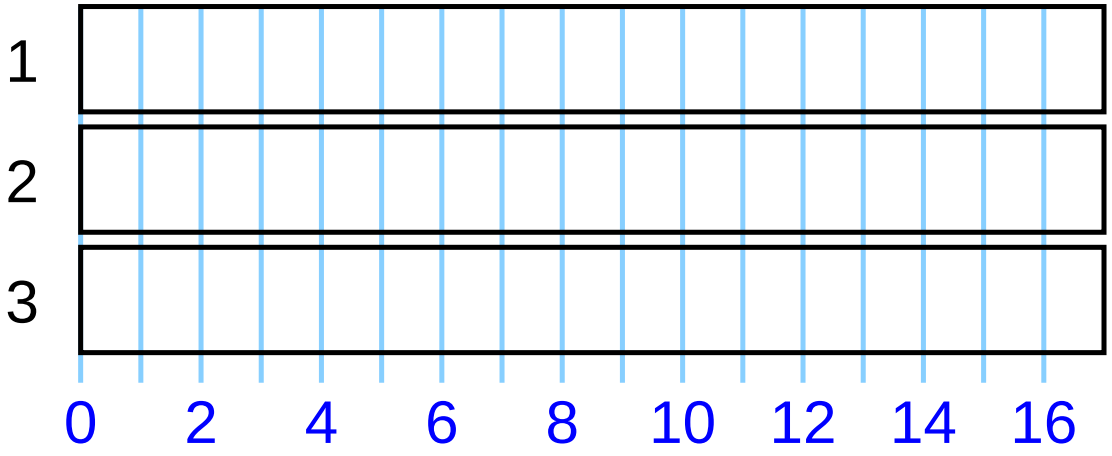
Example: List Scheduling with HLFET



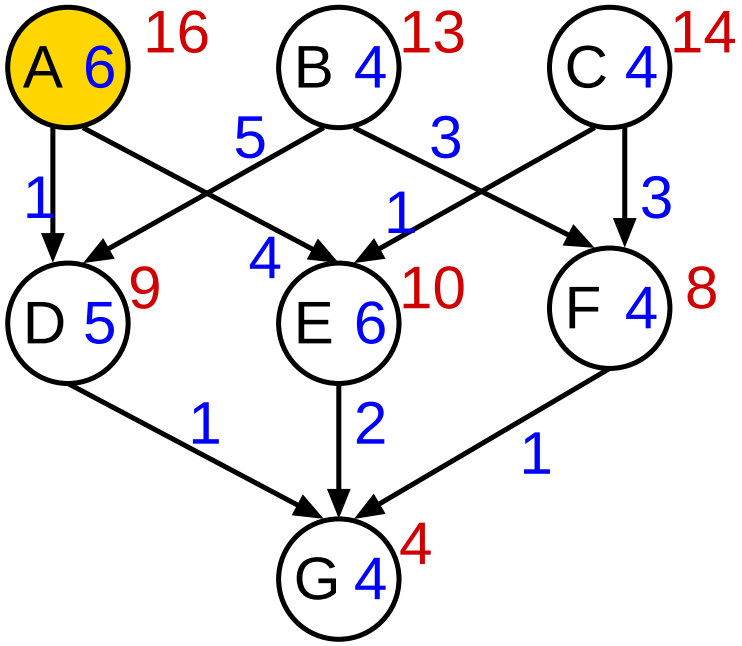
List:

A	C	B	E	D	F	G

Schedule with 3 nodes:



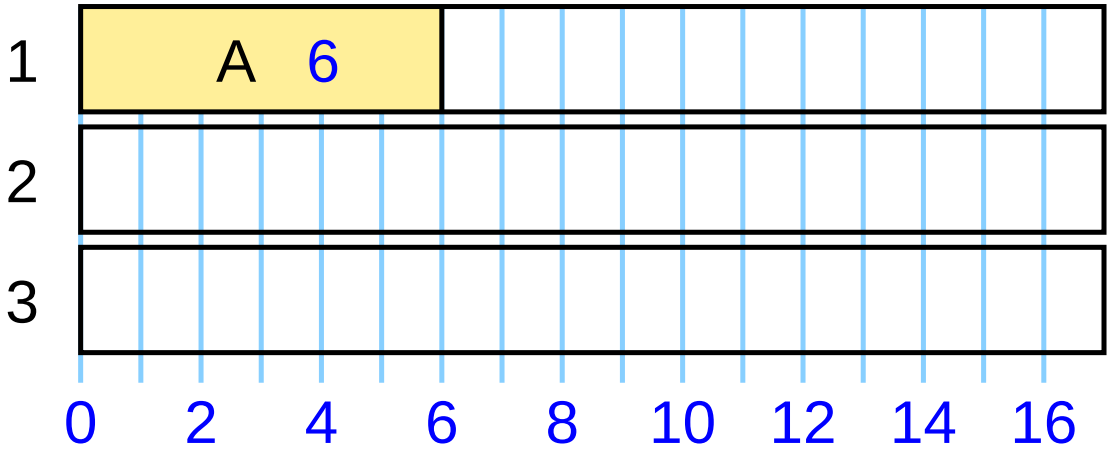
Example: List Scheduling with HLFET



List:

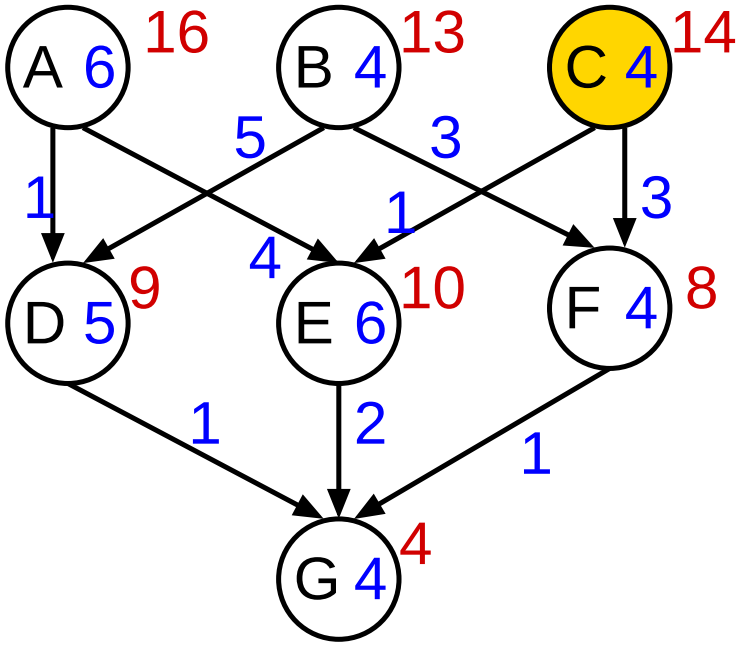
C	B	E	D	F	G

Schedule with 3 nodes:



➥ Assumption: local communication does not cost any time

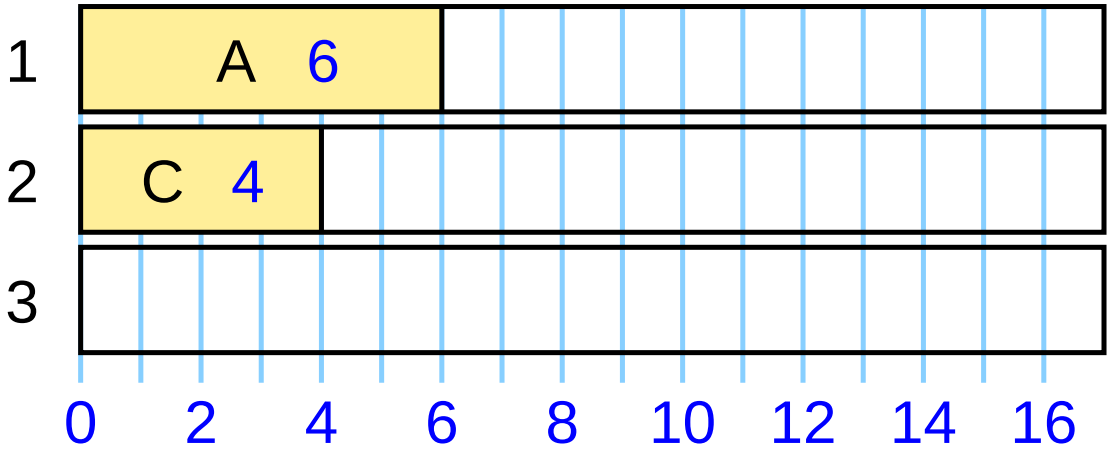
Example: List Scheduling with HLFET



List:

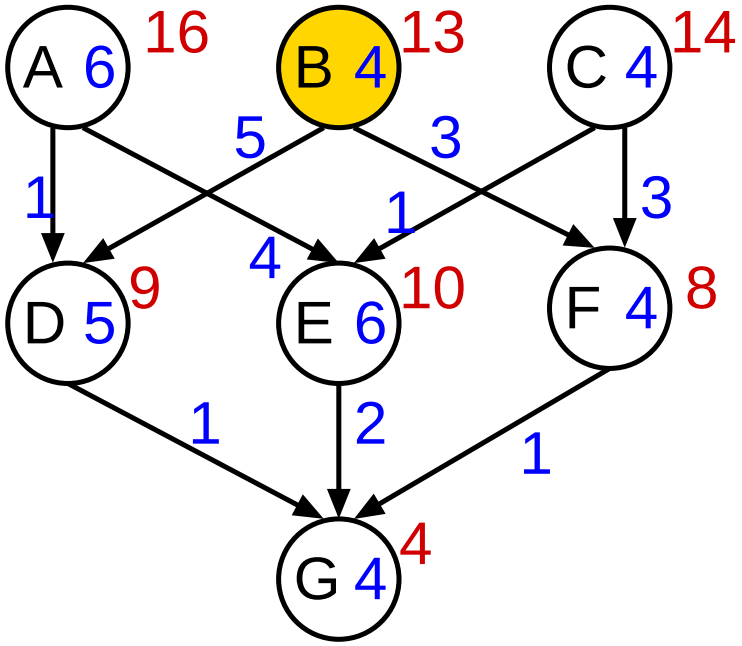
B	E	D	F	G

Schedule with 3 nodes:

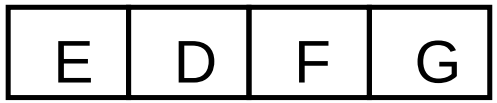


➥ Assumption: local communication does not cost any time

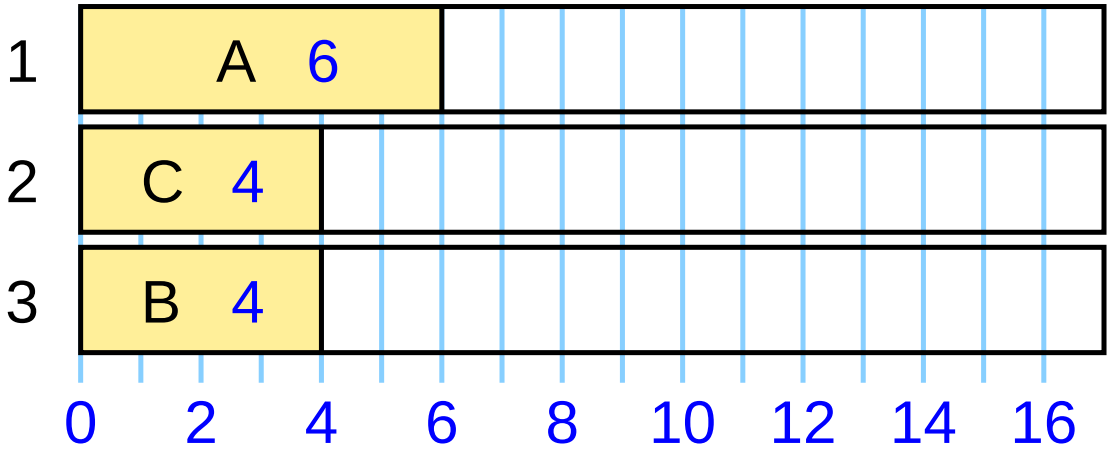
Example: List Scheduling with HLFET



List:

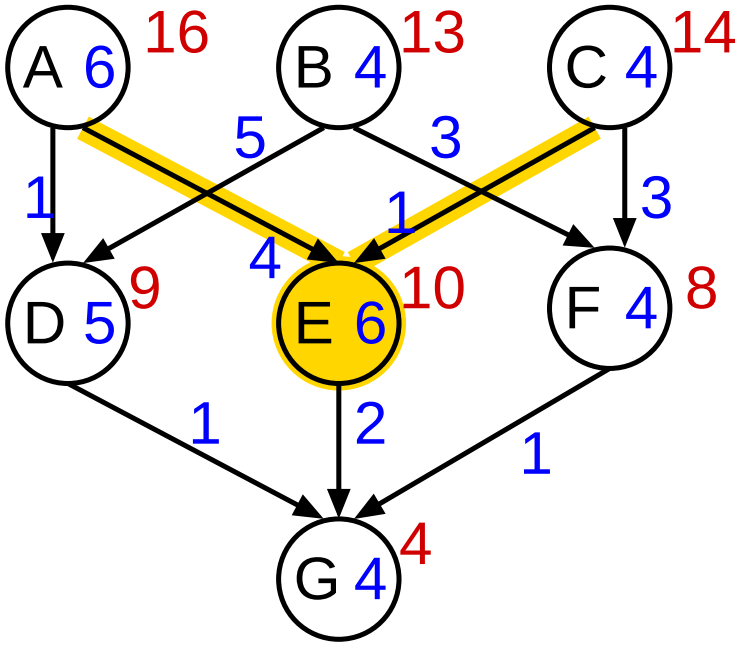


Schedule with 3 nodes:



➥ Assumption: local communication does not cost any time

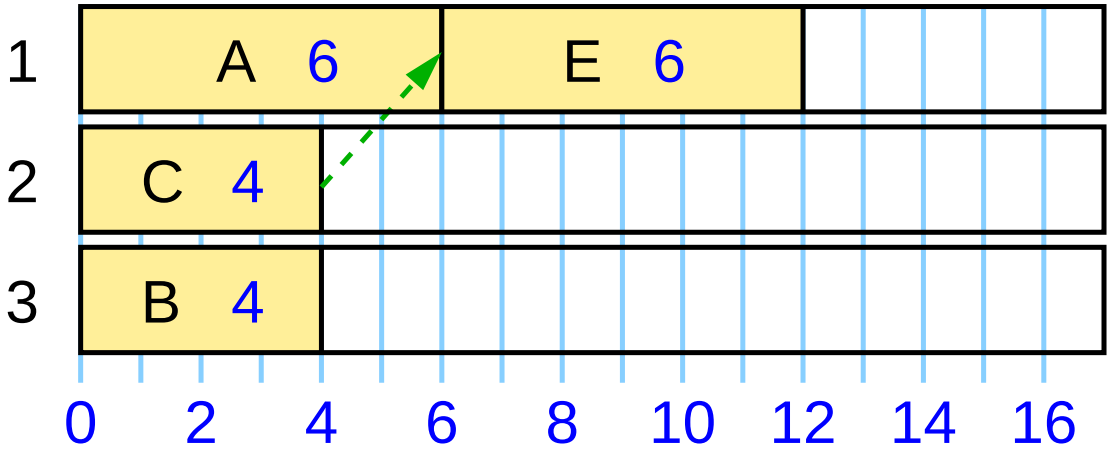
Example: List Scheduling with HLFET



List:

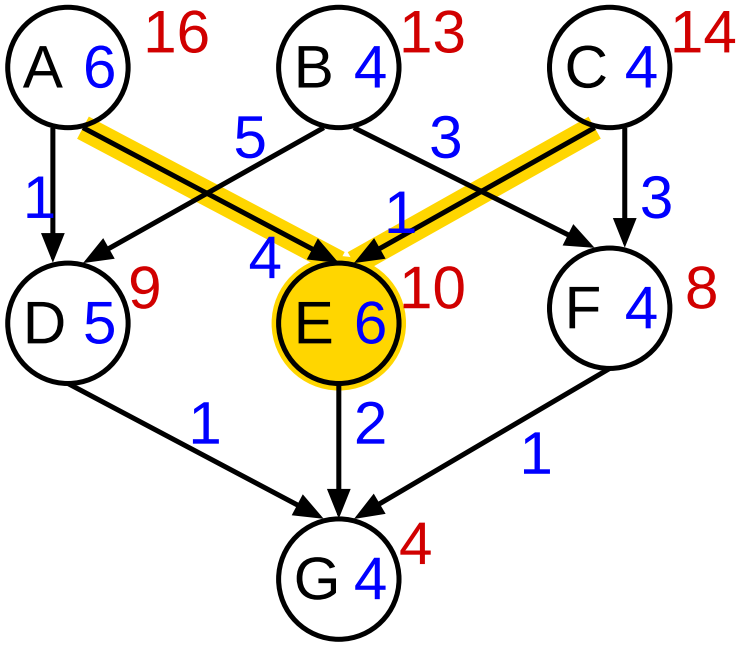
E	D	F	G

Schedule with 3 nodes:



➥ Assumption: local communication does not cost any time

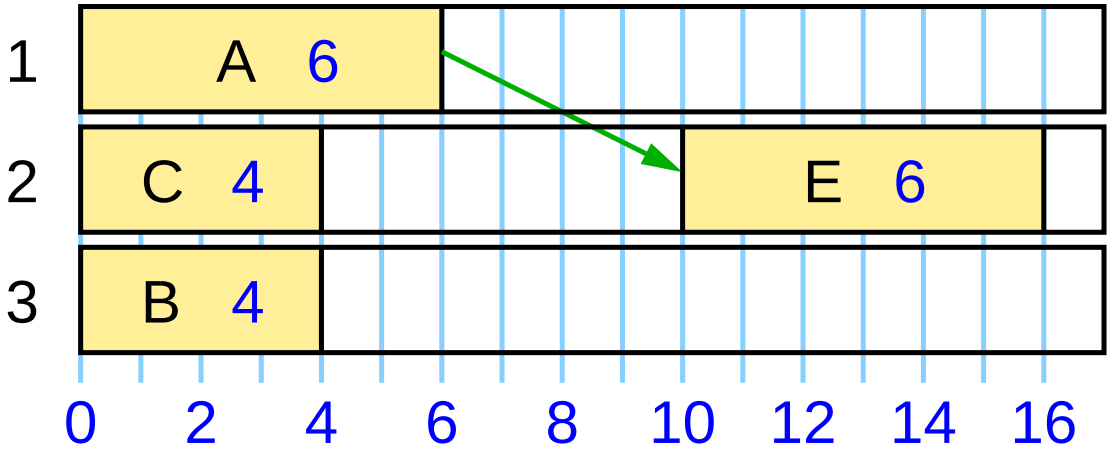
Example: List Scheduling with HLFET



List:

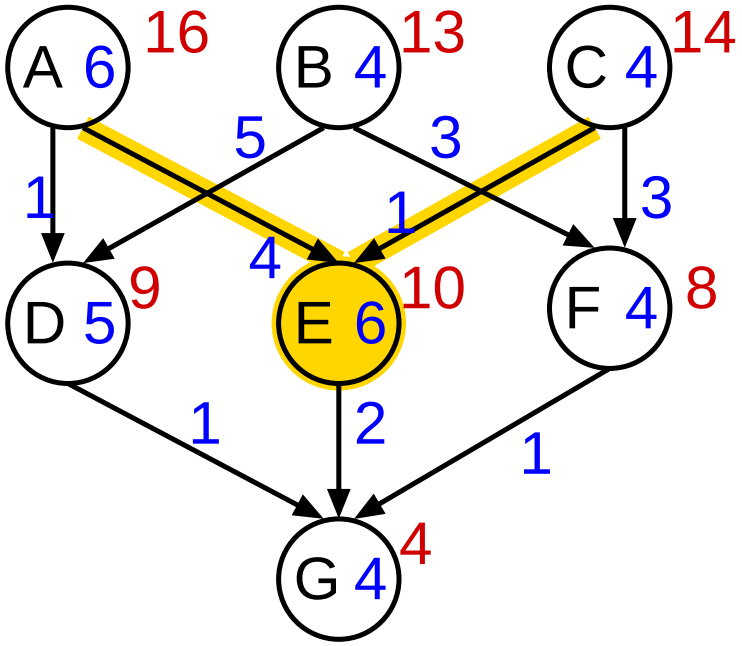
E	D	F	G

Schedule with 3 nodes:



➥ Assumption: local communication does not cost any time

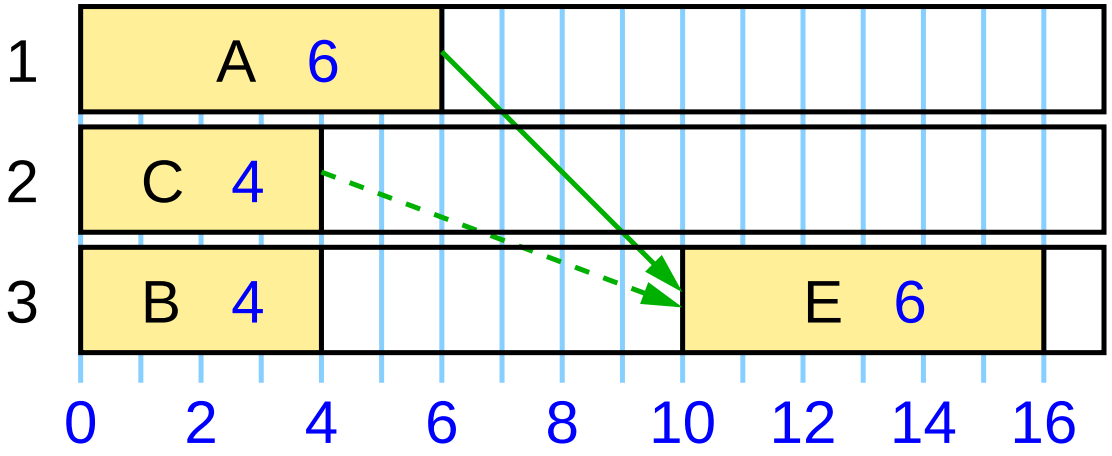
Example: List Scheduling with HLFET



List:

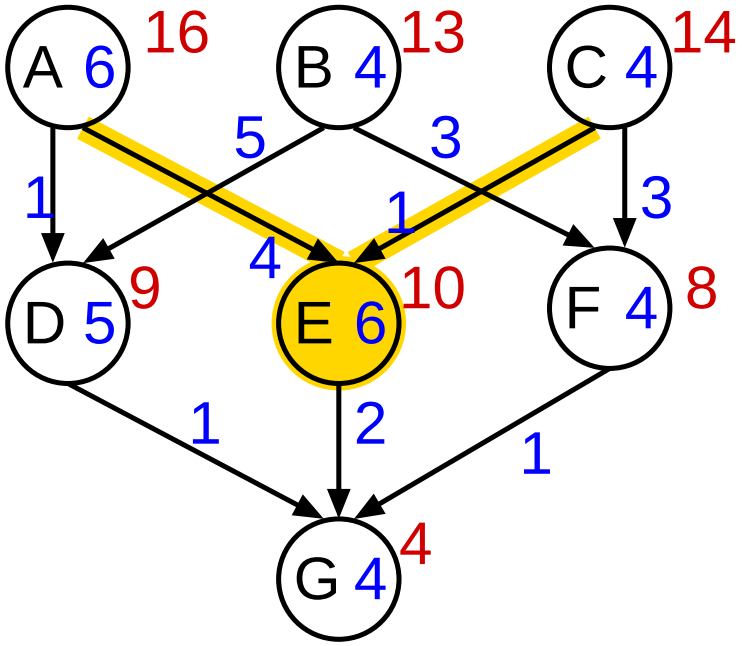
E	D	F	G

Schedule with 3 nodes:



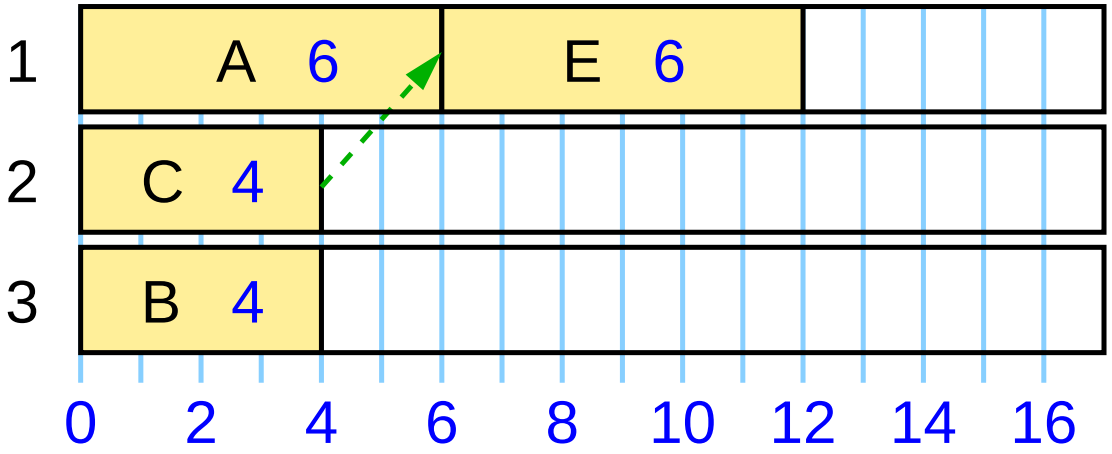
➥ Assumption: local communication does not cost any time

Example: List Scheduling with HLFET



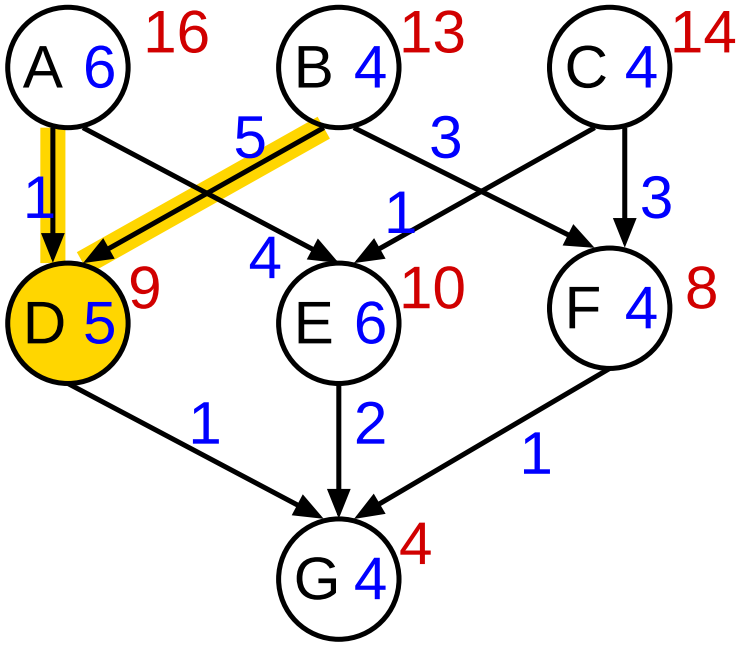
List:

Schedule with 3 nodes:



➥ Assumption: local communication does not cost any time

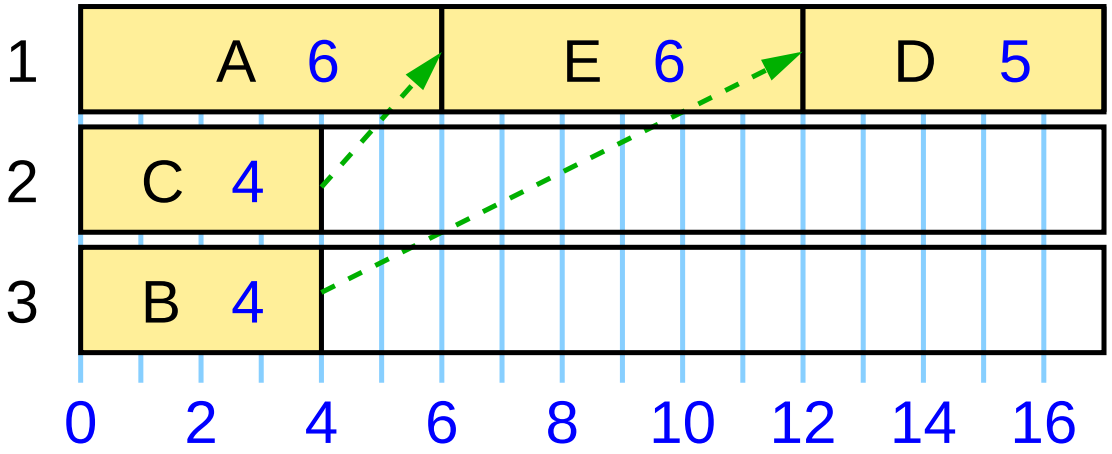
Example: List Scheduling with HLFET



List:

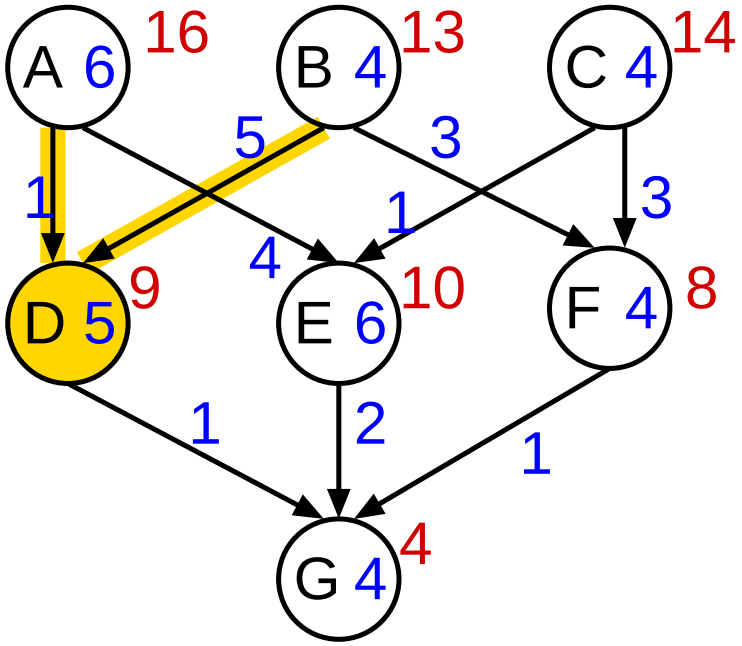
D	F	G

Schedule with 3 nodes:



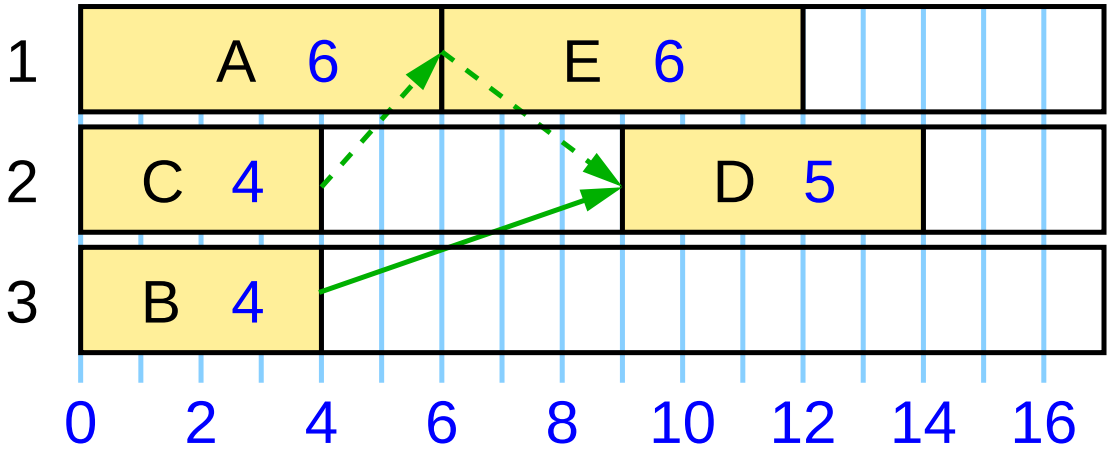
➥ Assumption: local communication does not cost any time

Example: List Scheduling with HLFET



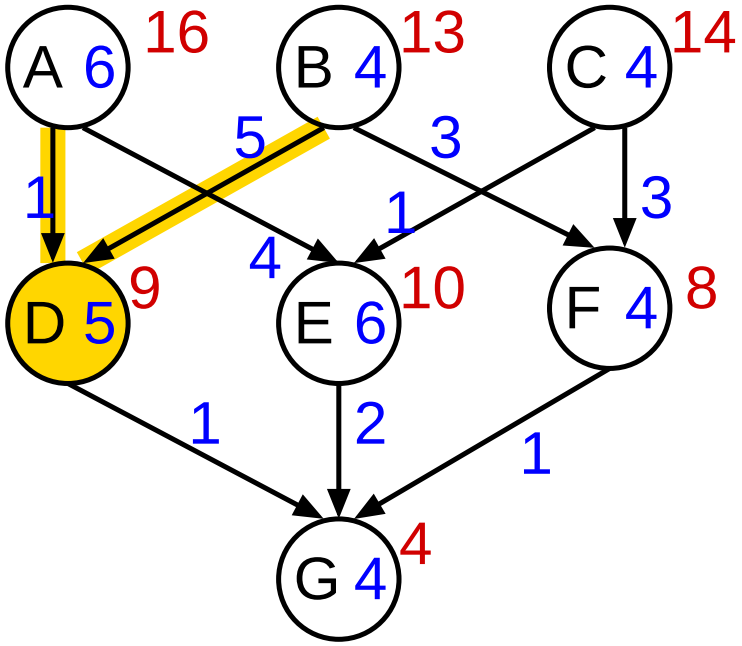
List:

Schedule with 3 nodes:



➥ Assumption: local communication does not cost any time

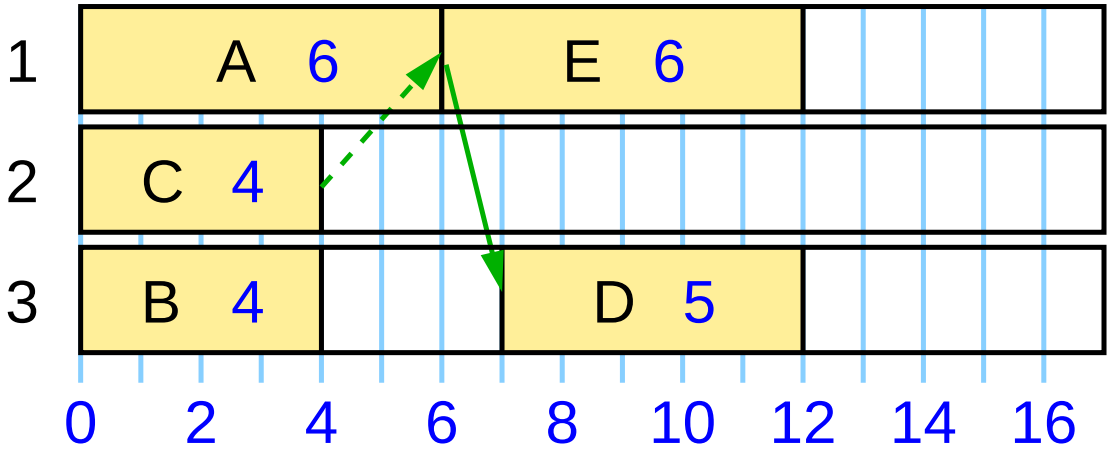
Example: List Scheduling with HLFET



List:

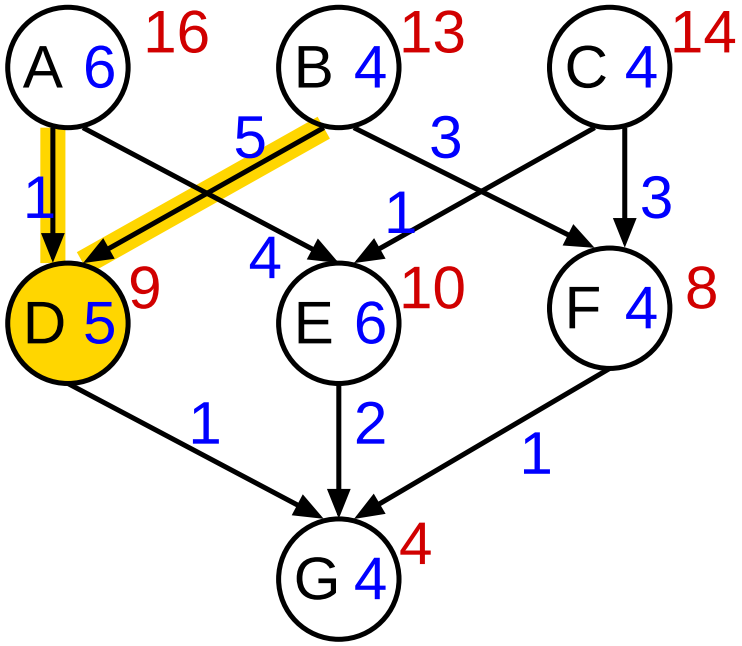
D	F	G

Schedule with 3 nodes:

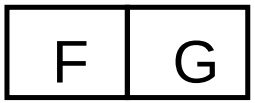


➥ Assumption: local communication does not cost any time

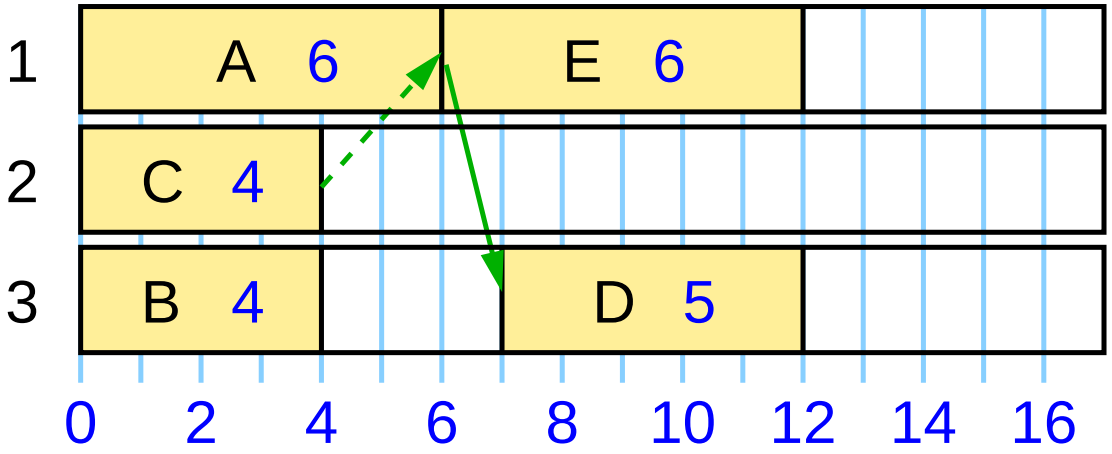
Example: List Scheduling with HLFET



List:

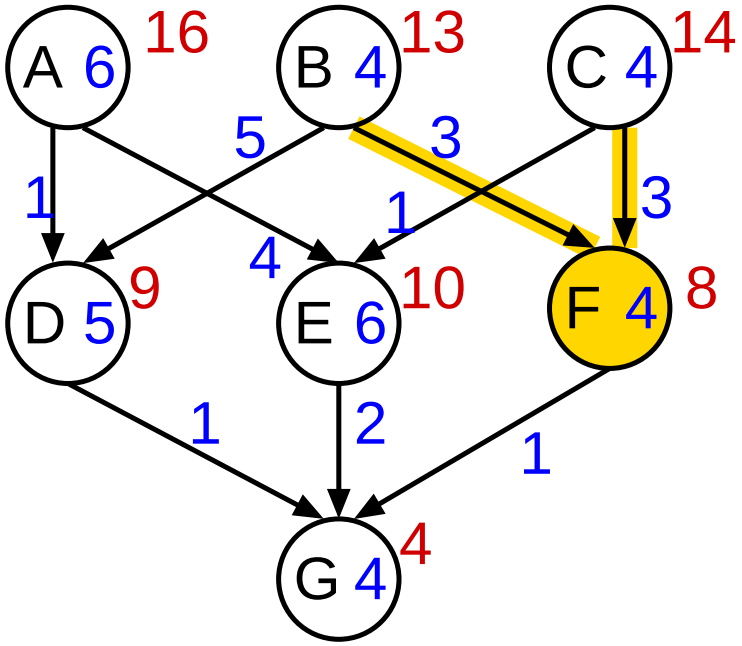


Schedule with 3 nodes:



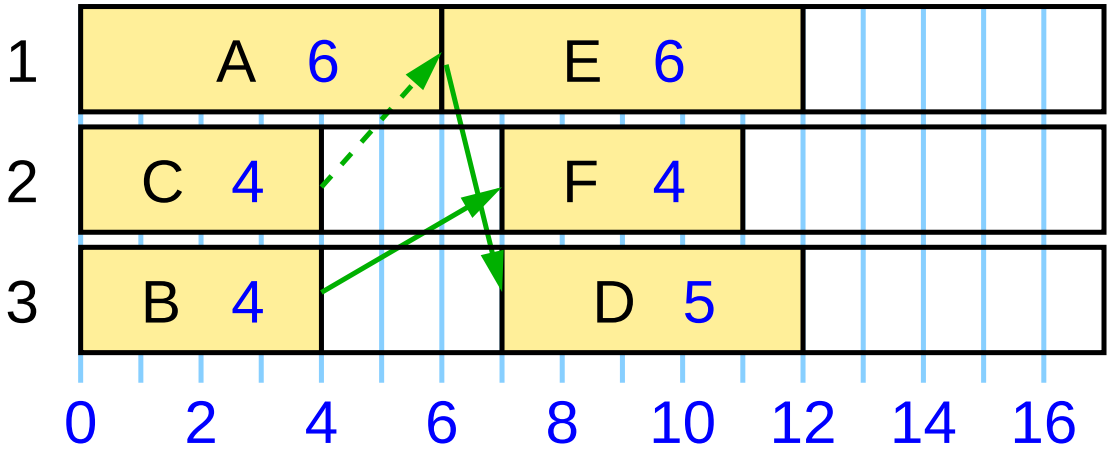
➥ Assumption: local communication does not cost any time

Example: List Scheduling with HLFET



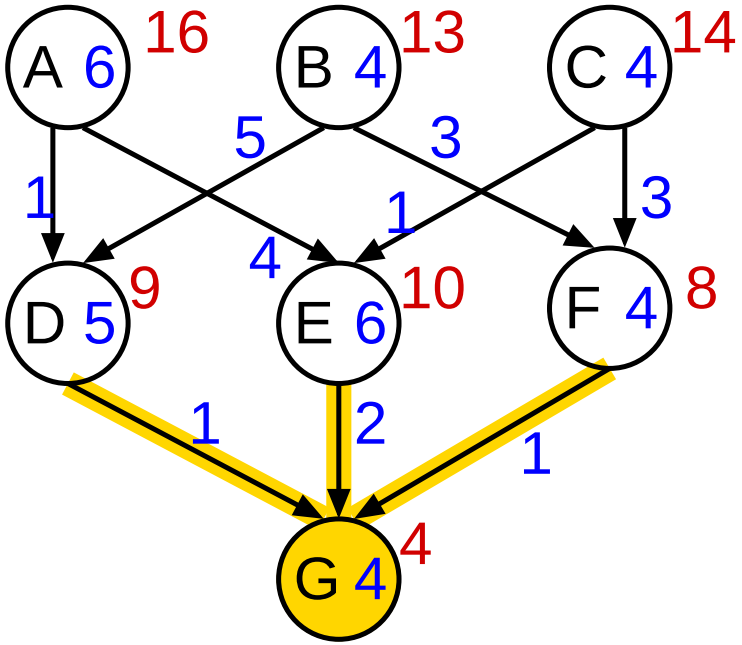
List:

Schedule with 3 nodes:



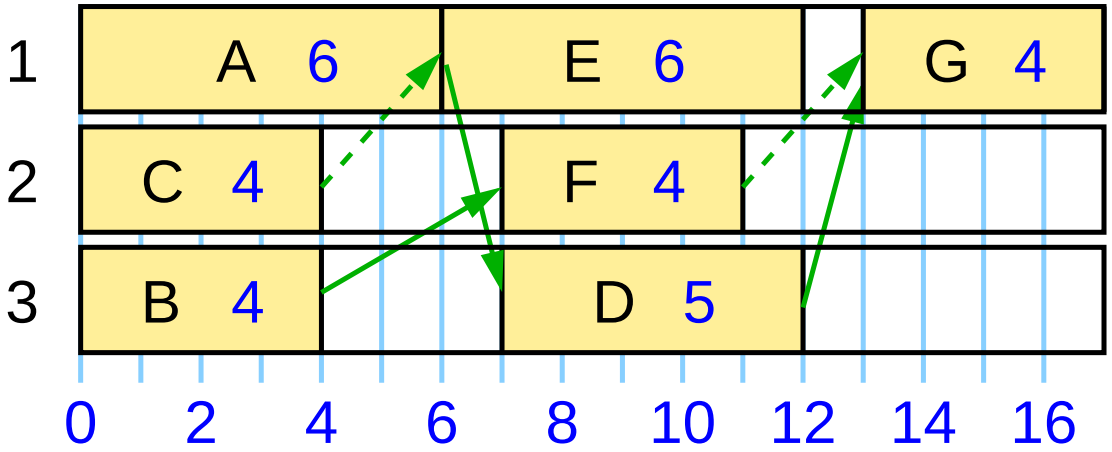
➥ Assumption: local communication does not cost any time

Example: List Scheduling with HLFET



List:

Schedule with 3 nodes:



➥ Assumption: local communication does not cost any time

5.1.2 Dynamic Load Balancing

- Components of a load balancing system
  - *Information policy* – when is load balancing triggered?
    - on demand, periodically, in case of state changes, ...
  - *Transfer policy* – under which condition is load shifted?
    - often: decision with the help of threshold values
  - *Location policy* – how is the receiver (or sender) found?
    - polling of some nodes, broadcast, ...
  - *Selection policy* – which tasks are moved?
    - new tasks, long tasks, location-independent tasks, ...

Typical approaches to dynamic load balancing

- Sender initiated load balancing
  - new process usually start on the local node
  - if node is overloaded: determine load of other nodes and start process on low-loaded node
    - e.g. ask randomly selected nodes for their load, send process if load $\leq$ threshold, otherwise: next node
  - disadvantage: additional work for already overloaded node!
- Receiver initiated load balancing
  - when scheduling a process: check whether the node has still enough work (processes)
  - if not: ask other nodes for work
- Similar also for preemptive dynamic load balancing

5.2 Code Migration

[Tanenbaum/Steen, 3.4]

- In distributed systems, in addition to data also programs are transfered between nodes
  - partly also during their execution
- Motivation: performance and flexibility
  - preemptive dynamic load balancing
  - optimization of communication (move code to data or highly interactive code to client)
  - increased availability (migration before system maintenance)
  - use of special HW or SW resources
  - use / evacuation of unused workstation computers
  - avoid code installation on client machines (dynamic loading of code from server)

Models for Code Migration

- Conceptual model: a process consists of three “segments”:
  - code segment
    - the executable program code of the process
  - execution segment
    - complete execution status of the process
      - virtual address space (data, heap, stack)
      - processor register (incl. instruction counter)
      - process / thread control block
  - resource segment
    - contains references to external resources required by the process
      - e.g. files, devices, other processes, mailboxes, ...

Models for Code Migration ...

- **Weak mobility**
  - only the code segment is transferred
    - including initialization data if necessary
  - program is always started from initial state
  - examples: remotely loaded classes in Java, Java Script
- **Strong mobility**
  - code and execution segment are transferred
  - migration of a process in execution
  - examples: process migration, agents
- Sender- or receiver-initiated migration

Code Migration Issues and Solutions

- Security: target computer executes unknown code
  - restricted environment (sandbox)
  - signed code
- Heterogeneity: code and execution segment depend on CPU and operating system
  - use of virtual machines (e.g. JVM, XEN)
  - migration points at which state can be stored and read in a portable way (possibly supported by compiler)
- Access to (local) resources
  - remote access with a global reference
  - move or copy the resource
  - new binding to resource of the same type

[Stallings, 14.1]

Process migration

- Migration of a process that is already running
  - triggered by OS or the process itself
  - mostly for dynamic load balancing
- Sometimes combined with *checkpoint*/*restart* function
  - instead of transferring the status of the process, it can also be stored persistently
- Design goals of migration procedures:
  - low communication effort
  - only short blocking of the migrated process
  - no dependency on source computer after migration

Process Flow of a Process Migration

- Creating a new process on the target system
- Transfer the code and execution segment (process address space, process control block), initialization of the target process
  - required: identical CPU and OS or virtual machine
- Update all connections to other processes
  - communication links, signals, ...
  - during migration: buffering at source
  - then: forwarding to target computer
- Delete the original process
  - if necessary, retain a “shadow process” for redirected system calls, e.g. file accesses

Transferring the process address space

- ***Eager (all)***: transfer the entire address space
  - no traces of the process remain on source nodes
  - very expensive for large address space (especially if not all pages are used)
  - often together with checkpoint/restart function
- ***Precopy***: process continues to run on source node during transfer
  - to minimize time in which the process is blocked
  - pages modified while the migration is in progress must be sent again

Transferring the process address space ...

- ***Eager (dirty)***: transfer only modified pages that are in main memory
  - all other pages are only transferred when accessed
    - integration with virtual memory management
  - motivation: quickly “flush” main memory of the source node
  - source node may remain involved until the end of the process
- ***Copy-on-reference***: transfer each page only when accessed
  - variation of *eager (dirty)*
  - lowest initial costs
- ***Flushing***: move all pages to disk before migration
  - after that: *copy-on-reference*
    - advantage: main memory of the source node is relieved