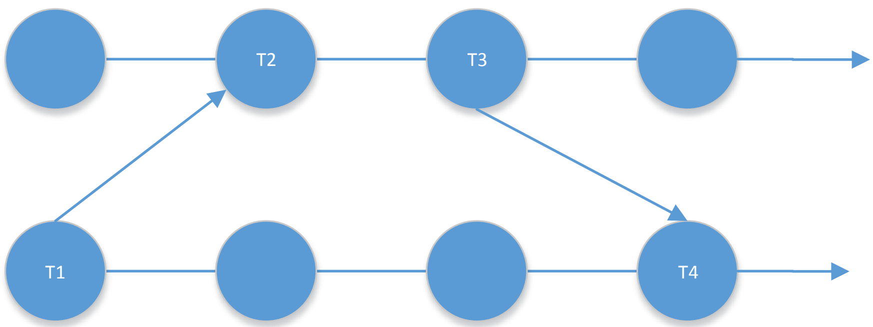

Figure 10: Time stamps used by NTP

- several threads sharing CPU
- thread context has little memory information
- threads avoid blocking application
- thread switch is fast
- user level threads allows parallel computation
- I/O or other blocking system calls block all threads

$\longrightarrow$ lightweight processes aim at combining the advantages of both.

8 Programs in distributed systems (week 7)

Recall advantages of threads:

- a blocking system call blocks only thread, not process
  $\Rightarrow$ system call for communication in distributed system.

9 Multiple threads in clients and servers

Clients

- multiple threads may hide communication delay (= distribution transparency)
- web browser opens several connections to load parts of a page.
- web server may be replicated in same or different location
  $\Rightarrow$ truly parallel access to items and per all download

Servers

- single-threaded, e.g. file-server
  - thread serves incoming requests, waits for disk, returns document
  - serves next request
- multi-threaded
  - dispatcher thread receives request
  - hands it over to a worker, which waits for disk etc.
  - dispatcher handles next request

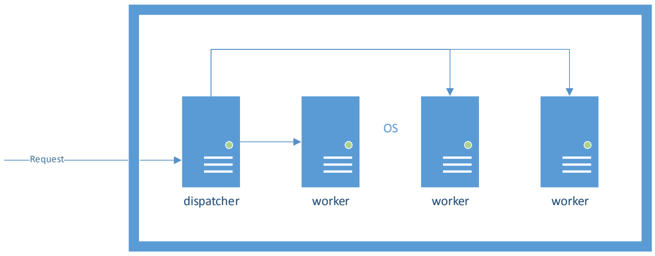

Figure 11: Multi-threaded architecture of dispatcher and worker threads

- finite state machine mode
  - only one thread
  - examines request, either reads from cache or from disk
  - during wait times, stores request in table
  - serves next request
  - message can be either new request or reply from disk - act accordingly
  - process acts as finite state machine that receives messages and acts/changes state

model	characteristics
single thread	no parallelism , blocking system calls
multiple threads	parallelism, blocking system calls
finite state-machine	parallelism, no blocking system call

Table 2: Properties of the server models

9.1 Virtualization

Virtualisation pretends there are more resources than there are in fact available.
Threads create "virtual processor".
⇒ general "resource virtualization"

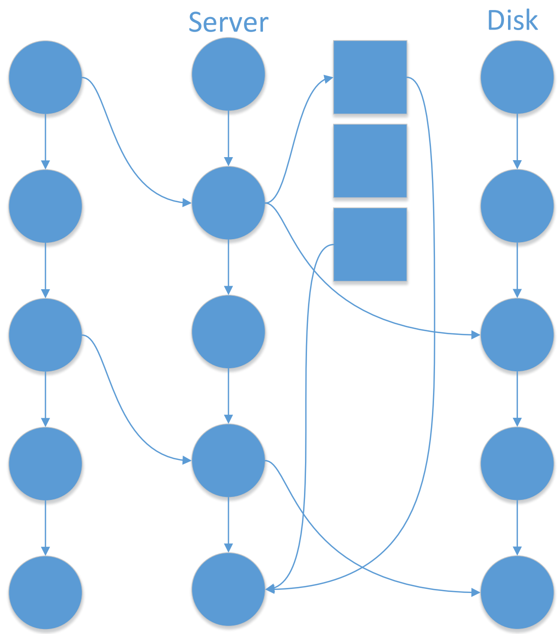

Figure 12:

Virtualisation in general:

- Layered system offers interfaces to layer above e.g. programming interface

Program
Interface A
Hardware system A

If B is mainframe

Program
Interface A
Implementation of Mimicking A on B
Interface B
Hardware B

Reasons for the need of virtualization:

- hardware changes much faster than software. Virtualization improves portability
- heterogeneous systems: networks consist of different types of switches, routers, servers,..
  $\Rightarrow$ virtualization enables portability of programs for all usage(distributed applications, network)

9.2 Code Migration

Code migration or running processes migration

Why do it?

- traditionally, load balancing in multi-processor machines, cluster
- service placement in distributed systems
  ⇒ minimize communication
- security is a big issue! Not addressed here.

9.2.1 Models for Code Migration

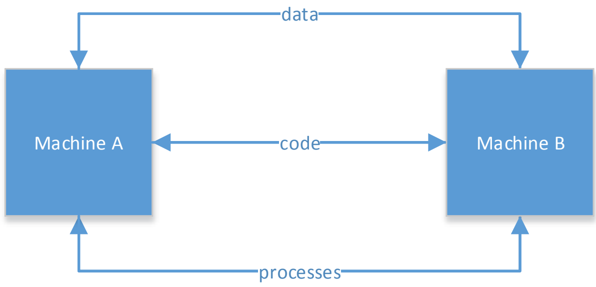

Figure 13: Migration from machine A to machine B

Or process model:

1. code segment, instructions
2. resource segment, references to external resources, i.e. file, printer, devices, other processes
3. execution segment, execution state of process, stack, private data, program counter, etc.

9.2.2 Migration types

- weak mobility, transfer code,(1) maybe 3)) which executes from beginning (Java applet)
- strong mobility, transfer 1) and 2), stop execution, transfer, resume
- cloned process is executed in parallel to original process.
  Migrating resource segment 2) is difficult

Consider process-to-resource binding

1. binding by identifier: URL, FTP-server
2. binding by value libraries for programming, value is needed but precise file
3. binding by type local devices, monitor, printer, etc. that can be easily replaced

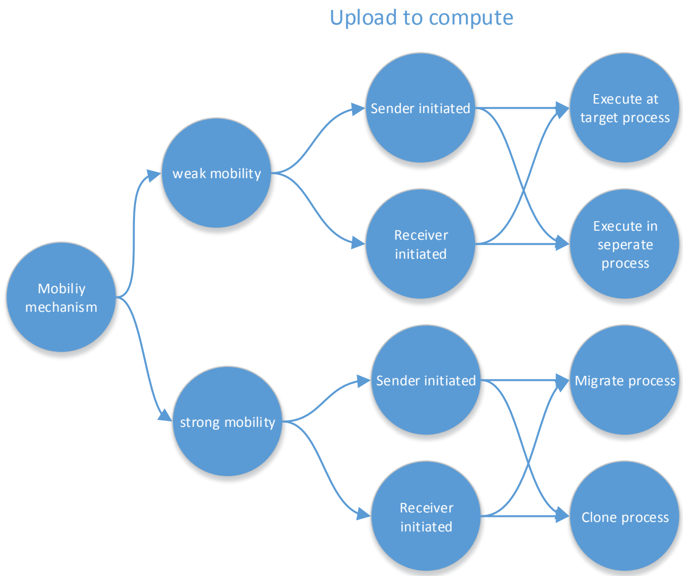

Figure 14: Migration classification

Resource-to-machine binding

1. unattached, easily movable (e.g. files)
2. fastened resources, difficult to move (database, websites,..)
3. fixed resources, not movable (local device, communication endpoint)

process-to-resource binding	resource-to-machine binding		
	unattached	fastened	fixed
by identifier	MV	GR(or MV)	GR
by value	CP	GR(or CP)	GR
by type	RB	RB(GR,CP)	RB(GR)

MV = move, GR = global reference, CP = copy, RB = rebind to locally available resource

Remote procedure calls: ordinary procedure Call
e.g. count = read($\underbrace{fd}_{\text{int indicating file}}$, $\underbrace{buf}_{\text{array pointer}}$, $\underbrace{nbytes}_{\text{int ?roof bytes}}$)

Remote procedure call uses stubs to pack parameters in message

parameter passing:

- packing of parameter in message is called parameter marshalling
- value parameters are easily packed in message; messages are transferred by byte; different machines can be a problem: little endian versus big endian.

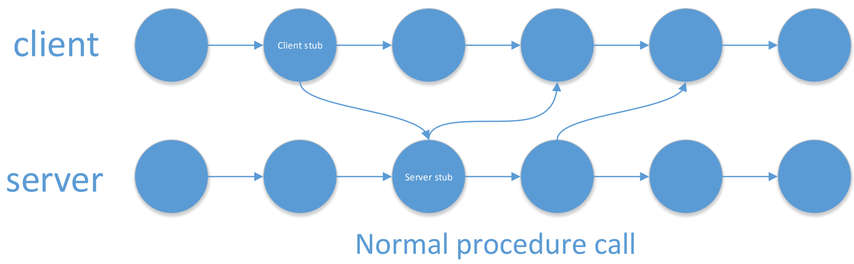

Figure 15: Remote procedure call

- reference parameters
  - passing pointers in extremely difficult
  - create array and pass as value, how to handle linked list, graph?

RPC protocol implementation needs interfaces (=stubs). Interface definition language (IDL) defines interfaces for stub implementation.

Asynchronous RPC hides communication ⇒ communication transparency.