ON PROGRAM TRANSLATION

A priori, we have two translation mechanisms available:

- **Interpretation**
- **Compilation**

On interpretation: Statements are translated one at a time and executed immediately.

- Advantages:
  - Programs are smaller and easier to write
  - Execution starts quickly
  - Debugging is easy
- Disadvantages:
  - Memory must be allocated to the program and to the interpreter
  - Performance penalty

On compilation: Complete machine code of the entire program is generated prior to execution.

- Advantages:
  - Lower memory requirements
  - Faster program execution
- Disadvantages:
  - Execution is delayed (must follow compilation)
  - Debugging is harder

Overall: In parallel computing we tend to use compilers: we are mostly concerned about execution speed.

PARALLELIZING COMPILERS

Goal: Decrease execution time of a program by breaking it into blocks that could be executed simultaneously by different CPUs. (We tend not to worry about increase in compilation time).

Identification of program blocks depends on the computer architecture.

Approaches in compiler construction:

- **Run-Time Partitioning and Run-Time Scheduling**: Practical for specific applications. In general, performance suffers due to partitioning and scheduling overhead at run-time.

- **Compile-Time Partitioning and Run-Time Scheduling**: Most common approach for multiprocessor targets. Partitioning is done by programmer or by compiler. Synchronization and communication mechanisms must be provided.

- **Compile-Time Partitioning and Compile-Time Scheduling**: Biggest challenge in compiler construction. No overhead at run time, but it is VERY difficult (impossible in general!) to estimate the program execution time, so the resulting scheduling may be far from optimal.

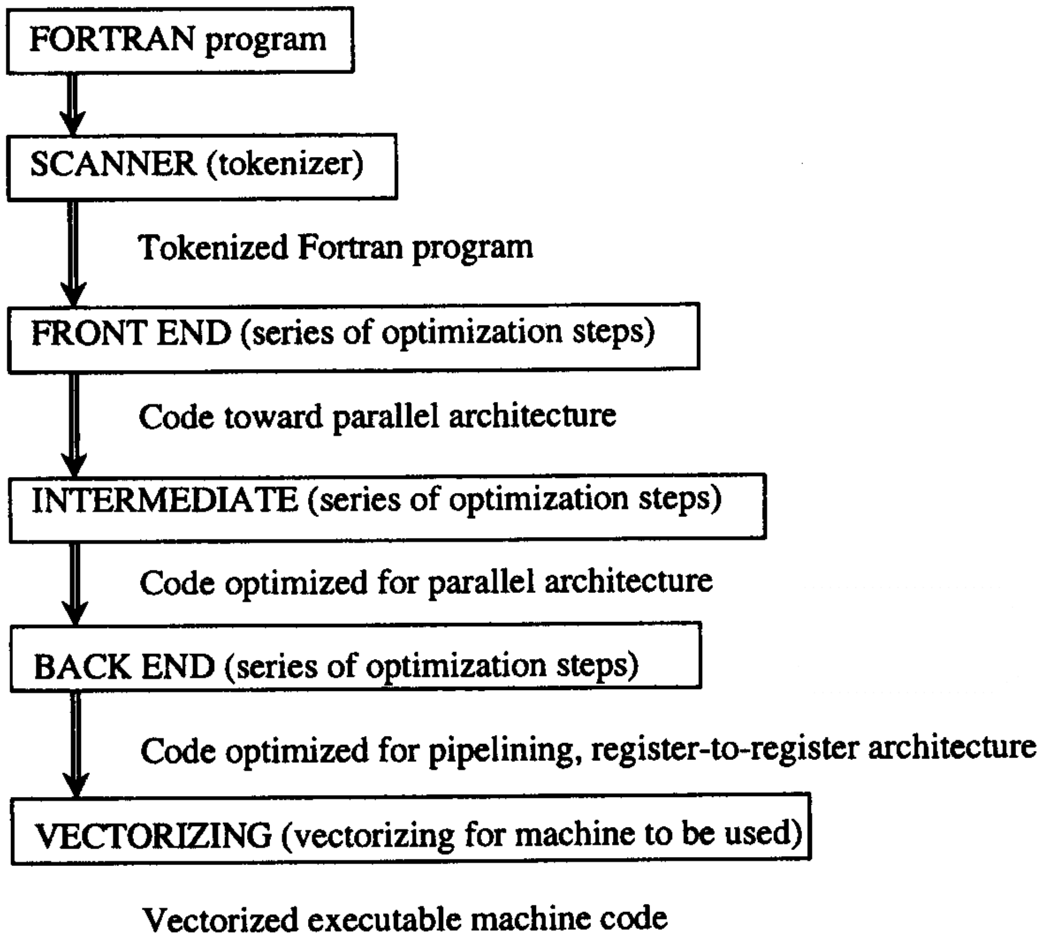
CASE STUDY: PARAPHRASE COMPILER

A FORTRAN-to-FORTRAN (i.e. source to source) compiler developed at U. of Illinois for the Cedar multiprocessor, also used successfully on vector processor machines such as Cray X/MP.

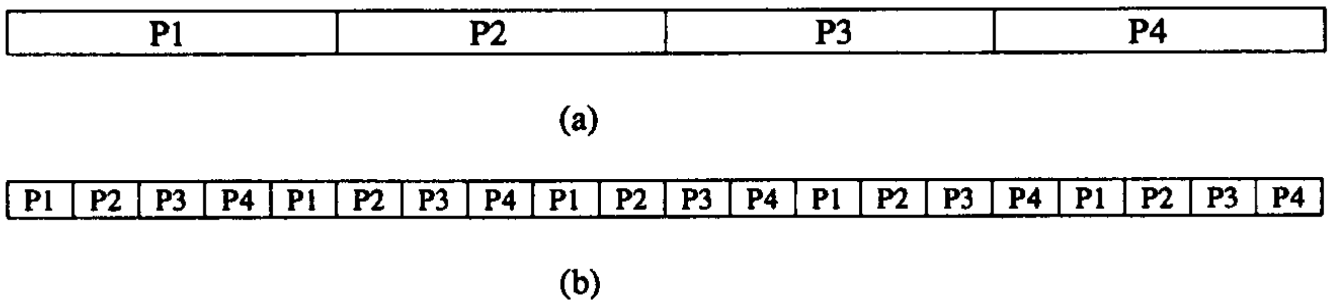
Main idea: Using data dependence graph, transform a traditional, sequential FORTRAN program into another FORTRAN program suitable for parallel execution. Done in two phases:

Perform machine-independent transformations and equivalent changes to the program code to obtain an intermediate program expressing the parallelism,
Map the intermediate program onto the machine architecture.

Diagram follows: Paraphrasing a FORTRAN program to work on pipelined vector processors.


FORTRAN program
SCANNER (tokenizer)
Tokenized Fortran program
FRONT END (series of optimization steps)
Code toward parallel architecture
INTERMEDIATE (series of optimization steps)
Code optimized for parallel architecture
BACK END (series of optimization steps)
Code optimized for pipelining, register-to-register architecture
VECTORIZING (vectorizing for machine to be used)
Vectorized executable machine code

CASE STUDY: ARRAY PARTITIONING BETWEEN PROCESSORS



(a) Block partitioning, (b) cyclic partitioning

OPERATING SYSTEMS

Traditional (centralized) operating systems for uniprocessors have the luxury of perfect knowledge of the state of the machine.

Operating systems for parallel computers usually do not have an up-to-date knowledge of the state of the parallel machine. Their primary goal is to integrate the computing resources and processors into one unified, cohesive system.

In order of increasing complexity: Operating systems for the parallel architectures can be classified into one of three categories:

1. Simple modifications of uniprocessor Oses, like VMS or UNIX. Usually run on selected architectures, typically of master / slave configuration.

2. Oses designed specifically for selected architectures. Examples: Hydra OS for C.mmp or Medusa OS for the Cm* machine.

3. General-purpose Oses designed from the start to run on different parallel architectures. Example: MACH OS, a descendant of UNIX.

Research in OS design for parallel architectures focus on the following:

1. **Resource Sharing**: sharing resources among processors.
   Example: Sharing of a global memory of a SIMD machine.

2. **Extensibility:** possibility of addition (static or dynamic) of extra processors.

3. **Availability:** fault tolerance in face of the component (say, processor) failure.

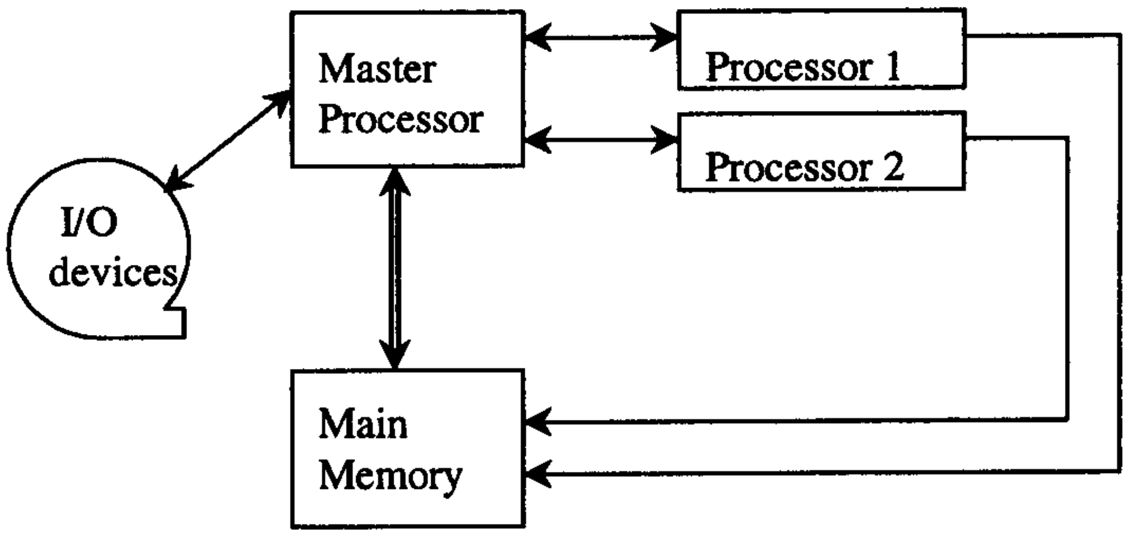
OPERATING SYSTEMS ORGANIZATIONS:
MASTER / SLAVE

One processor is designated as the master, all others are slaves.
Only the master may execute the OS, slaves execute user programs only.
Only the master can perform I/O on behalf of the slaves.

Problems:

- Everything fails if the master fails.
- Slaves can run effectively CPU-bound jobs, but I/O bound jobs generate frequent interrupts of the master.
- The OS code is simple (no need to be reentrant) but offers poor degradation. Poor fault tolerance.
- Works well only for specific applications with well defined, static slave loads.
- Does not respond well to dynamically changing loads.

Typical master / slave multiprocessor configuration is shown below:



OPERATING SYSTEMS ORGANIZATIONS:
SEPARATE EXECUTIVE

Each processor runs its own copy of the OS and responds to interrupts generated by processes running on that processor.

A process is assigned to run on a particular processor.

Each processor has its own dedicated resources, such as peripherals, files, etc.

The processors do not cooperate on execution of an individual process. This may result in some processors being overloaded, while other processors may be idle.

A failure of a single processor usually does not cause the system failure, but restarting a processor may be difficult.

In general, separate executive Oses are more reliable than master / slave Oses.

In some approaches some information tables contain global data, that must be protected by mutual exclusion. Alternatively, one may replicate all tables so that each processor has its own copy, but keeping the tables up-to-date and consistent is difficult.

OPERATING SYSTEMS ORGANIZATIONS:
SYMMETRIC ORGANIZATION

One processor at any time may be the master, but this role floats from one processor to another.

Each processor may control I/O devices or shared memory unit.

Although only one processor is the master, other processors may be concurrently executing supervisory services.

OS must be reentrant and mutual exclusion must be provided to protect global data structures. Conflicts arising from processors trying to reference the same memory locations are resolved by hardware.

This organization is very fault tolerant.

Effective use of resources.

Easy recovery in case of single processor failure.

OPERATING SYSTEMS ORGANIZATIONS: DISTRIBUTED ORGANIZATION

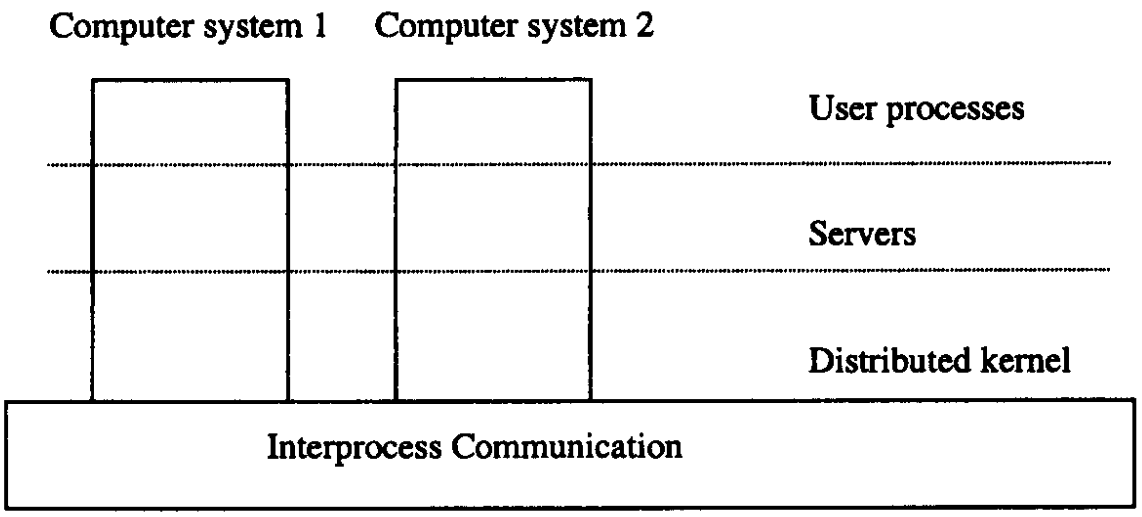
Distributed OS looks to the user like the familiar, centralized OS.
It provides a virtual machine abstraction of a distributed system, thus providing an unified interface for resource access regardless of location.

Key objective here is transparency.

Each processor in a distributed system may have assigned different functions to perform. Example: Various OS functions and utilities are allocated to different processors. A given processor is responsible only for its OS function or utility.

Consequently, each processor executes the OS code segments residing in its own local memory. This approach removes the needs for global memory access (and resulting conflicts). OS code need not be reentrant.

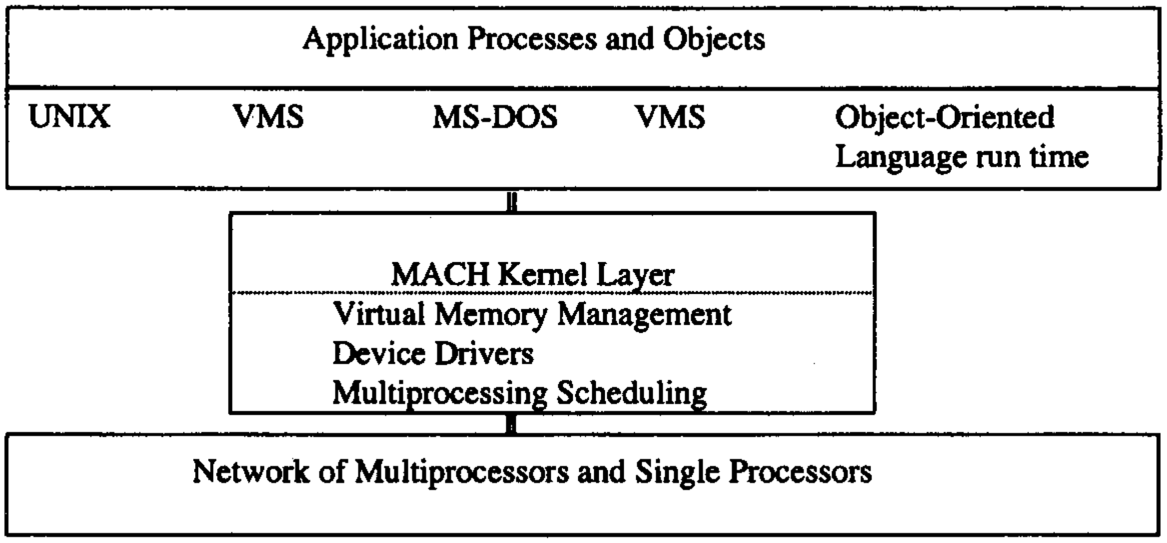
OS must provide interprocess communication mechanism (message passing).



CASE STUDIES:
MACH

One of the first general-purpose OSes designed specifically for SMP machines. Emulates UNIX, VMS, OS/2, MS-DOS, etc. Developed at Carnegie-Mellon University. Runs on SUN machines, IBM RT, VAX, Encore and Sequent multiprocessors, BBN Butterfly and others. Recent MAS OS is based on MACH.

MACH organization:


Application Processes and Objects
UNIX
VMS
MS-DOS
VMS
Object-Oriented Language run time
MACH Kernel Layer
Virtual Memory Management
Device Drivers
Multiprocessing Scheduling
Network of Multiprocessors and Single Processors

Services provided by MACH kernel:
Interprocess communication
Virtual memory management (paging, sharing of memory between tasks)
Resource allocation and management
Controlled access to peripherals

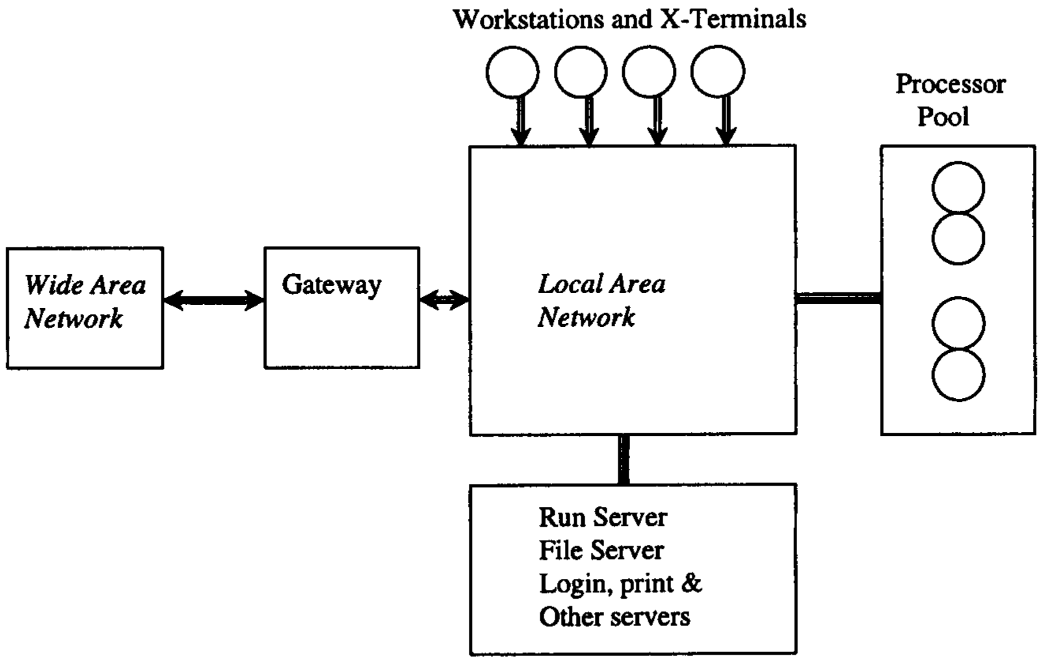
There exists ann-kernel level of user services like synchronization, semaphores, file servers, etc.

Various MACH components may run on different processors.

CASE STUDIES: AMOEBA

A distributed OS designed under the supervision of A. Tanenbaum at the Vrije Universiteit in Amsterdam. Goal: Provide all the basic facilities of a conventional OS, assuming that memory and processors are going to become cheap and each user will be allocated many processors and lots of memory.

AMOEBA organization:

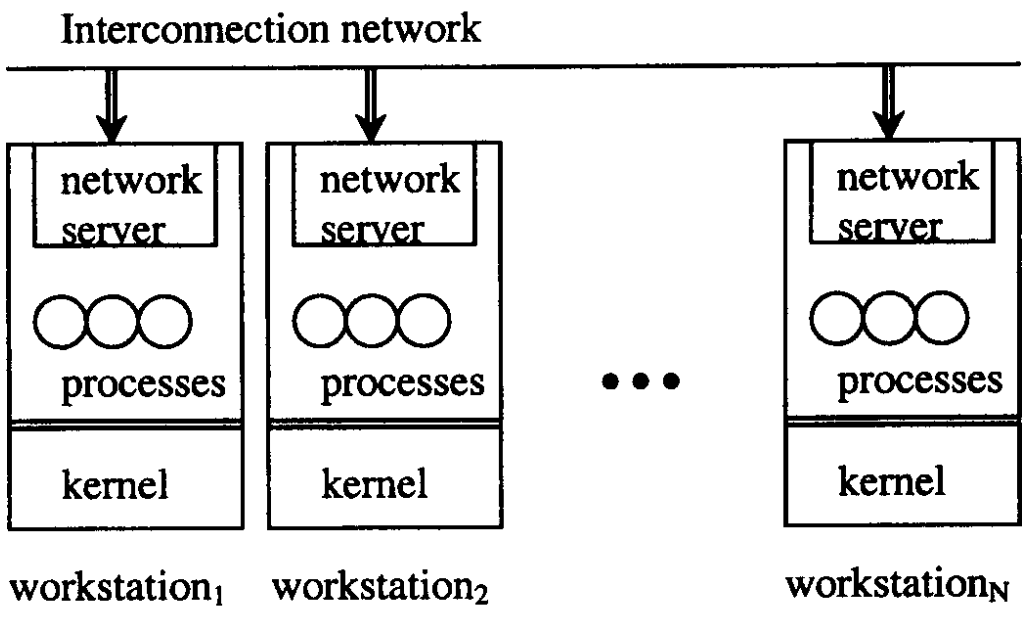


Design principles:

- **Network transparency:** all resource access is net transparent.
- **OO environment**: all resources (files, directories, disk blocks, processes, devices, etc.) are objects, named in an uniform manner and managed by servers by calling server methods.
- **Capability:** all system objects are named and protected by secure capabilities. There is a uniform interface to all objects. User view: a system is a collection of objects named by capabilities
- **RPC:** interprocess communication mechanism involves a sender and receiver and is somewhat similar to rendezvous.
- **Kernel server:** micro-kernel supports a uniform model for accessing resources using capabilities. Basic kernel abstractions are: processes, threads and ports (for communication).

CASE STUDIES:
ACCENT

Another OS from Carnegie-Mellon University, intended to support large network of computers. It is communication-oriented.



Memory management is effectively integrated with interprocess communication system, so that all access to all resources and services is provided via communication requests.

Design principles:

- **Fork and terminate primitives** for process creation and destruction.
- **Protection** of all interactions to avoid conflicts.
- **Programming language support** (many languages).
- **Network transparency.**
- **Fault recovery, debugging and monitoring** (tools available).
- **Problem decomposition**: the system supports modular decomposition of large problems into smaller subproblems.
- **Micro-kernel** consists of language-independent microcode support, low-level scheduling, I/O interrupt support and virtual address translation.