Processes, Communication, and Synchronization

How do we model computation?

Message-oriented systems
- small # of large, static processes
- communication through send/receive
- synchronization through send/receive
- separate address spaces for processes
- no data shared

Procedure-oriented systems
- many small processes
- monitors
- fork/join used for communication
- most communication through shared data

These classifications come from the duality paper which states that neither approach is better than the other. However, one may be suited more for a specific architecture than the other.

Communicating Sequential Processes
- just about everything can be modeled as concurrent processes
- monitors can be seen as a single process that communicates with many separate processes
- procedures/subroutines may be modeled as processes

Monitors - Hoare semantics
- when you wake up after a wait, you have the lock
- context switches occur -> you're also the running process

How do you minimize overhead?

Monitors - Mesa semantics
- you must reacquire the lock when you wake up
- this reduces the number of context switches

Nonblocking objects (Compare and Swap)
The atomic update of a data object does not prevent others from using
Why are they good? They avoid:

1.) scheduling convoys
2.) priority inversion problem
3.) deadlock

Problem: C&S not supported in hardware and so must be implemented in software by using blocking objects (like TSL).

When constructing C&S from blocking objects, you must use roll-out or roll-forward to achieve the non-blocking quality.
Problems with blocking objects:
Bus contention (invalidates, increased requests)

Invalidates - can be avoided by reading value to make sure it's what you want it to be before trying to update it.

Increased requests - use software queueing so not everyone tries to grab lock when it is updated.

Software queueing results in many processes waiting on incorrect values. An advisory C&S is used so that occasionally processes on queue compare the current value to their expected value and abort if the value is not as expected.

Remote Procedure Calls
RPC is overgeneralized and clunky
The idea is to try to make the common case fast. The common case is RPC on the same machine.
The solution is LRPC

process <-> kernel <-> server
- map addresses of arguments into both process and server contexts
- establish stack for server code to run on
- process thread runs in server domain on this stack

Why is LRPC good?
1.) reduces copying of data (shared on stack)
2.) ? doesn't bother network interface with messages going to NI and then being routed to current node
3.) reduces scheduling frequency. RPC forces the client process to block and selects a server thread for execution.
4.) reduces access validation. LRPC doesn't require validation on a return.
5.) no need for a dispatcher thread as in RPC
6.) reduces the need for locks on shared data structures (we think buffers and such)
LRPC also allowed for the use of domain caching to reduce the overhead of domain switches.

Synthesis
- using kernel code synthesis, you reduce the overhead of kernel calls and context switches
- synchronization is the next big problem
Approaches to reducing synchronization costs:
1.) Code Isolation

- have code work on own pieces of data so you don't need to synchronize access to one large data object

2.) Procedure Chaining

- chain interrupts so you don't have interrupts interrupting interrupts

3.) Optimistic Synchronization

- assume less synchronization is needed so use less heavyweight synchronizing object
- rollback if violated (synch object chosen wasn't enough)
- upgrade synchronization mechanism to next level

Each resource has its own wait queue in Synthesis. This means you don't have to search long list to find associated processes.
Because almost all kernel data structures are implemented as queues, there is very little interlocking in kernel. (reduces synchronization)

Deadlock

Necessary conditions for deadlock:
1.) tasks claim exclusive control of resources they hold
2.) tasks hold resources while waiting for additional resources
3.) resources can't be preempted
4.) circular chain of tasks (and their requests) exists

Prevention of deadlock - attack on of 4 conditions
1.) request all resources at beginning and must get them all before processing (wait for condition)
2.) if task denied resource, must release all resources (no preemption)
3.) ordering of resource types (circular chain)

Avoid deadlock by only entering safe states.

Monitors:
- don't reference data outside of monitor
- use specific sequence of monitor calls

Mesa paper:
- interrupts can be lost due to race condition (naked notifies)
- condition variables must count or "remember" NOTIFY.