Concurrency and Concurrent Programming

- Early computers operates 1 bit of information at a time. Performance are drastically improved by hardware that can process data in words of several bits at once.
- Further improvements were achieved by arranging for I/O and CPU operations to proceed in parallel; thus the need of an OS to coordinate all these time-sensitive and critical activities between the CPU and the various I/O devices.
- Thus, OSs are the earliest concurrent programs.
- Multiprogramming systems attempt to utilize resources that would otherwise be wasted, by running two or more jobs concurrently.
- Multiprogramming shares the CPU and the I/O devices, while multiaccess systems allow many jobs to be run, each on behalf of a user at an interactive terminal or a remote client computer. These systems must also provide main storage for a large number of jobs whose combined demand may exceed the physical capacity of the computer.
- Multiprocessor systems have several CPUs operate simultaneously on separate jobs in a shared main store. Multicore systems provide parallelism within a CPU.
- The recent development of fast networks has boosted the interested of distributed computer systems, consisting of several complete computers that can operate independently but also intercommunicate efficiently.

- New computer architectures, such as *array processors*, *dataflow computers*, *connectionism*, and *massively parallel computers*, also sparkle hopes for previously unachievable performance by exploiting novel computation models.
- These hardware and OS developments open up myriad opportunities for injecting concurrency into application programs. At the same time, extra language features are called for to allow the programming of concurrent programs.
- Early examples of concurrent programs at the application level include simulation and iconic user interfaces.
- Execution of conventional sequential programs entails a single thread of control, while execution of concurrent programs involves interaction among a dynamic pool of multiple processes or threads.
- A sequential process is a totally ordered set of events, each event being a change of state in (some component of) a computing system. A sequential program is a text that specifies the possible state changes of a sequential process.
- A concurrent program specifies the possible state changes of two or more, often cooperating, sequential processes. *In general*, no ordering needs be defined between the state changes of any pair of processes.
- If the set of concurrent processes are to cooperate to achieve a common goal, they must (1) *communicate* in order to exchange information and (2) *synchronize* at certain critical points to ensure proper merging of control.

- Concurrent programming brings a new dimension of issues.
  - Nondeterminism

    随机数

    - Sequential processes are nearly always deterministic, the results of which are completely *reproducible* in multiple executions with the same input.
    - A concurrent program is likely to be *nondeterministic*. The order of execution of processes is often unpredictable, even under a particular language processor, since it may be influenced by run-time conditions and the specific OS scheduling policies.
    - Usually we try to write programs that are *effectively deterministic*, so their outcomes are predictable.
  - Speed dependence
    - A sequential program's correctness does not depend on the rate at which it is executed, while a concurrent program's outcome may depend on the *relative* speeds of execution of its component sequential processes.
    - When outcomes are speed-dependent, a *race* condition is said to exist.

      E.g. Suppose that two processes, P and Q, update the same string variable as follows.

      Race condition

      In P: `s := "ABCD";`

      In Q: `s := "EFGH";`

- The problem in the previous example can be resolved by declaring variables to be *atomic*, *i.e.* it must be inspected and updated as a whole, and not piecemeal. *E.g.* Suppose that two processes, P and Q, update the same integer variable, which is declared to be atomic and have initial value 0.
  In P: `i := i + 1;`
  In Q: `i := 2 * i;`
- Concurrent processes can share and update common resources, such as memories, screen, other I/O devices, *etc*. Care must be taken in coordinating the read/write accesses so as to ensure consistency of states.

◦ Deadlock

- During deadlock, a set of processes are prevented from making further progress by their mutually incompatible demands for additional resources. Deadlock can occur *iff* the following conditions all hold: *mutual exclusion*, *wait and hold*, *no preemption*, and *circular wait*.
- Solutions: ignore, detect and recover, prevent, smart OS scheduling (such as the banker's algorithm)

◦ A concurrent program has the property of finite progress (or *liveness* if it is guaranteed that every process will make nonzero progress over a sufficiently long (but finite) span of time. In the absence of deadlock, starvation takes place when a process is prevented indefinitely from running by unfair scheduling.

- The most basic command for specifying concurrency is the parallel command, "$C||K$." While the "$||$" symbol suggests parallelism, the command does not enforce simultaneity, but merely permits it. Note thus that collateral and sequential executions are special cases of concurrency.
- Concurrent programs are distinguished from sequential programs by the presence of operations that cause *interactions* between processes.
  - Commands C and K are independent if no step of C can affect the benhavior of any component step of K and *vice versa*.
    - → sequential
      Consequently, the compositions "$C;K$," "$K;C$," "C,K," and "$C||K$" are all equivalent.
      intermediate parallel
    - Thus, independent commands can be executed either collaterally or concurrently without special precautions. Concurrent composition of such commands is deterministic.
    - It is undecidable in general to determine if two commands are independent. A sufficient condition, however, is that neither commands updates a "variable" that the other inspects or updates.
  - Commands C and K compete if each must gain *exclusive* access to the same resource r for some of their actions.
    - Let $C = C_1; C_2; C_3$ and $K = K_1; K_2; K_3$. None of C_1, C_3, K_1, and K_3 access r. C_1 and the K_i's are independent. So are C_3 and the K_i's.

- C_2 and K_2 must acquire exclusive access to r, and thus cannot be executed in parallel. They are called critical sections with respect to the r. In the execution of "$C||K$," either C_2 precedes K_2, or *vice versa*.
- The execution of "$C||K$" is equivalent to that of either "$C; K$" or "$K; C$" but which is the actual outcome is in general unpredictable.

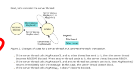
◦ Assuming the same structures of C and K as before, there is communication from C to K if the action C_2 must entirely precede the action K_2.

- C_2 must precede K_2 since it produces information that K_2 consumes.
- "$C||K$" must have the same outcome as "$C; K$."
- The situation becomes more complex if C and K can *intercommunicate*.

○ Analogous to jump in sequential control flow, the parallel command is too *primitive* and yet *powerful*. Maintaining the intellectual manageability of our programs is fallen upon the application programmer's shoulders, especially in the presence of complicated intercommunication among processes.

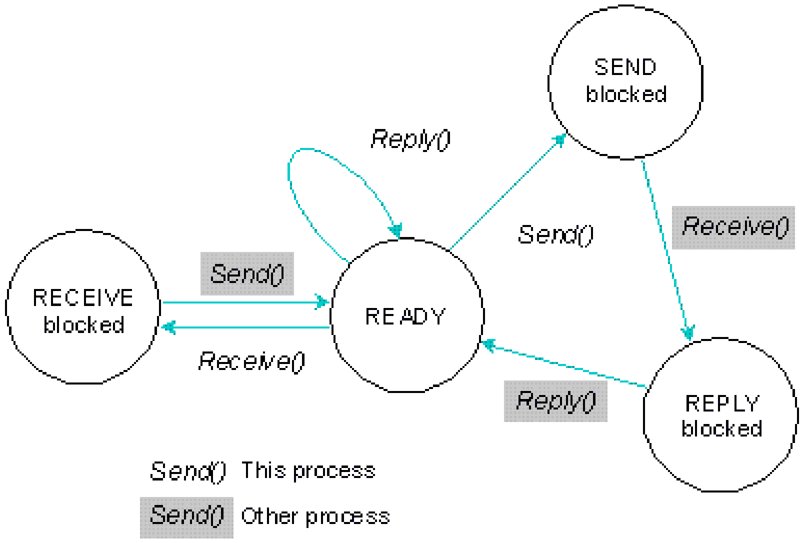
○ Concurrent language designers came up with a proliferation of notations of concurrent structures that are thought to be beneficial. Unfortunately, none correspond to the seminal theorem of Böhm and Jacopini in 1966, which assures that a small, fixed set of control structures is capable of expressing all possible sequential algorithms.

- Various low-level concurrent constructs were designed, including: `parbegin-parend`, semaphores, mutexes, message passing, and remote procedure calls.
- Higher level structured concurrent programming constructs also emerged, such as conditional critical regions, monitors (Concurrent Pascal and Modula), and rendezvous (Ada).
- With carefully designed semantics, message passing can be used for, in addition to efficient inter-process communication, process synchronization. QNX
- The message passing system consists of a blocking *Send()*, a blocking *Receive()*, and a non-blocking *Reply()*.
  - *Send()*—the sender process to send a message to another process, and is blocked until the receiver replies.
  - *Receive()*—the receiver process to receive a message from another process, and is blocked until a message arrives.

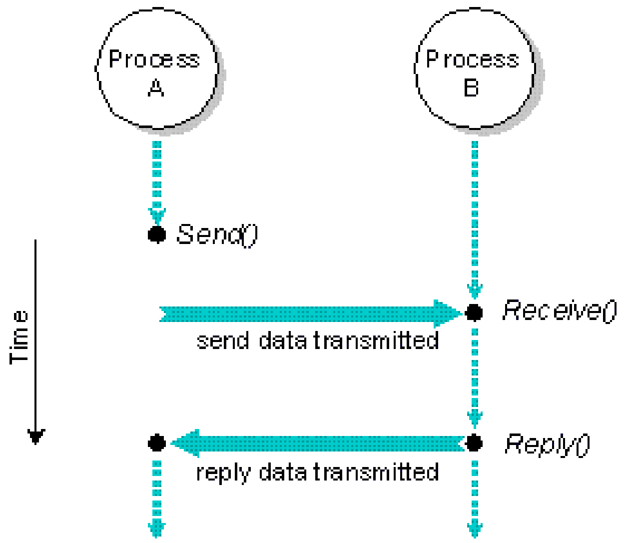
    

    前提: no message is waiting, 若有 message is waiting, 不block
  - *Reply()*—the receiver of a message to issue a reply to the sender, and to move on; the sender is unblocked.
- The message passing protocol synchronizes process execution by setting processes in various blocked states.
  - The sender process is SEND-blocked when the message it sent has not yet been received by the receiver process.
  - The sender process is REPLY-blocked when the message it sent is received by the receiver process, but the receiver has not replied yet.
  - The receiver process is RECEIVE-blocked when the process has not received a message yet.

- The following state diagram summarizes well how a process undergoes state changes in a typical send-receive-reply transaction.



- The following illustration outlines a simple sequence of events: Process A sends a message to Process B, which subsequently receives, processes, then replies to the message.



- Process A sends a message to Process B by issuing a *Send()* request to the OS. At this point, Process A becomes SEND-blocked until Process B issues a *Receive()* to receive the message.
- Process B issues a *Receive()* and receives Process A's waiting message. Process A changes to a REPLY-blocked state. Since a message was waiting, Process B is not blocked.

  (Note that if Process B had issued the *Receive()* before a message was sent, it would become RECEIVE-blocked until a message arrived. In this case, the sender would immediately go into the REPLY-blocked state when it sent its message.)
- Process B completes the processing associated with the message it received from Process A and issues a *Reply()*. The reply message is copied to Process A, which is made ready to run again. A *Reply()* does not cause blocking, so that Process B is also ready to run. Who runs first depends on the relative priorities of Processes A and B.

◯ Note how message passing not only allows processes to pass data to each other, but also provides a means of synchronizing the execution of several cooperating processes.

- Once Process A issues a *Send()* request, it is unable to resume execution until it has received the reply to the message it sent.

- This ensures that the processing performed by Process B for Process A is completed before A can resume executing.
- Moreover, once Process B has issued its *Receive()* request, it cannot continue processing until it receives a message.

- The message passing model just described supports a new paradigm for concurrent programming: the Administrator-and-Worker paradigm, which achieves a high degree of concurrency and parallelism, and uses the available resources efficiently and effectively.
  - Two types of processes: administrators and workers.
  - An administrator owns one or more worker processes.
  - An administrator provides public services by (1) *receiving* requests from clients and possibly queuing up the requests and (2) delegating jobs to its worker processes, which perform the actual computation and processing.
  - A worker *sends* a message to its administrator to report results of last job, if any, and indicates its availability of further tasks.
  - An administrator, upon *receiving* a message from a worker, inspects if there are existing work requests.
    - If yes, the administrator *replies* to the worker with work order.
    - If no, the administrator remembers that the worker is ready for work. At some future time when work is requested, the administrator initiates work by *replying* to the available workers.

- Workers are either blocked waiting for jobs or performing work.
- Administrators issues only *Receive()* and *Reply*, and never *Send()* requests. Thus, they are never blocked, unless they are waiting for requests from clients or reports from workers. This allows the administrators to attend to events and requests instantaneously. 即刻
  no send
  Receive blocked
- A courier 快递员 is a special type of worker processes that sends messages on behalf of its administrator to another administrator.
  若不用courier而是直接由administrator直接send，administrator会被send blocked

- Note how it is unnecessary for an administrator to complete the servicing of one request before accepting the next.
- The Administrator-and-Worker paradigm allows the decomposition of a system by the functionalities and services provided, which is similar to object-oriented decomposition.
- As in the object-oriented approach, an administrator hides its workers, in particular, the number of instances of each kind of worker; thus hiding the working details of the services provided.