Introduction to Concurrent Programming

Mahendran Velauthapillai
• I don’t think we have found the right programming concepts for parallel computers yet. When we do, they will almost be certainly be very different from anything we know today – Brinch Hansen [1993]

• My only serious debate with your account is with the last sentence. I do not believe there is any “right” collection of programing concepts for parallel or even sequential computers. The design of a language is always a compromise, in which the designer must take into account the desired level of abstraction, the target machine architecture, and proposed range of applications. –Hoare [1993]
• Concurrency in a programing languages and parallelism in the underlying hardware are independent concepts.

• Hardware operations occur in parallel if they overlap in time.

• Operations in source text are *concurrent* if they could be, but need not be, executed in parallel. Operations that occur one after the other, ordered in time, are said to be sequential.
• The fundamental concept of concurrent programming is the notion of process.

• In this section process corresponds to a sequential computation, with its own thread of control.

• The thread of a sequential computation is the sequence of program points that are reached as control flows through the source text of the program.
• **Communication:** Involves the exchange of data between *processes*, either by an explicit message, or through the values of shared variables. A variable is *shared* between processes if it is visible to code for the processes.

• **Synchronization:** relates the thread of one process with that of another. If \( p \) is a point in the thread of a process \( P \), and \( q \) is a point in the thread of a process \( Q \), then synchronization can be used to constrain the order in which \( P \) reaches \( p \) and \( Q \) reaches \( q \). In other words, synchronization involves the exchange of control information between processes.
Parallelism in Hardware

- Dates back to 1950
- Disparity between instruction execution and input/output
- This resulted in the introduction of data channels
- The IBM 709 1958 had one CPU and 6 data channels, hence simultaneous arithmetic operations and input/output possible.
• This raised the problem of synchronized access to shared resources, such as memory devices and input/output devices.

• Eg: How could two data channels be prevented from writing to the same printer and mixing up the output.

• *Busy waiting (or Polling)*: A processor continuously check a condition such as completion of I/O by a data channel.
Interrupts and Time Sharing

• Interrupts: A hardware signal, allowed the activities of CPU to be synchronized with the data channels.

• If a program P needs to read data, the CPU can initiate a read action on a data channel and start executing a another program Q. Once the data has been read, the data channel can send an interrupt to the CPU and it resumes execution of P.
Interrupts + hardware clock
≡
Time sharing (time slicing).

Eg: CPU starts a program, the clock send an interrupt and the CPU suspends the current program and starts another....

NOTE: Program can be suspended by I/O
Multiprocessor Organization

- The Burroughs B5000 allowed two CPU’s and four data channels to access shared memory.
- A shared memory facilitates communication through shared variables that are visible to the communicating processes.
- *Communication is synchronous* if a value is sent and received simultaneously without any delay.
Concurrency and Interleaving

**Definition: Atomic operation**
An indivisible operation performed at the Hardware level.

**Definition: Event**
An operation considered by a language (eg: addition, procedure call) to be atomic.
Definition: Thread of a Process
Corresponds to a sequence of events.
Interleaving Threads

• Is a convenient technical device for studying concurrent execution of processes.

• Suppose a thread T1 consists of two events $a, b$ and T2 consists of events $x, y, z$
  (Note: $a$ must occur before $b$ and similarly $x$ before $y$ and $y$ before $z$) the following are some of the possible concurrent execution of Threads T1 and T2
• a b x y z
• a x b y z
• .......
• .......
• x y z a b
with text_io; use text_io;
procedure identify is
  task p; -- task specification for p
  task body p is
  begin
    put_line("p");
  end p;
  task q; -- task specification for q
  task body q is
  begin
    put_line("q");
  end q;
begin -- procedure body sets up parent of p and q
  put_line("r");
end identify;

Figure 12.6 A procedure with two tasks declared within it.
The calls to \texttt{put\_line} in the procedure body and the two tasks can be interleaved in six possible ways. The procedure and tasks identify themselves by writing a character on a line. The six possible outcomes appear in the following columns:

\begin{center}
\begin{tabular}{cccccc}
p & p & q & q & r & r \\
q & r & p & r & p & q \\
r & q & r & p & q & p \\
\end{tabular}
\end{center}
LIVENESS PROPERTIES

• Concurrent execution raises two kinds of correctness issues:

• **Safety**: Deals with getting the “right” answer

**Liveness**: Deals with the rate of progress of the process. That is the rate at which the Computation proceeds.

Next we study deadlock, livelock and fairness
Resource Sharing Constraints Concurrency

Competition for resources imposes constraints on the interleaving of threads. Consider two processes $A$ and $Z$ that compete for a resource. Suppose that the thread of process $A$ is the sequence of events

$$a \ lock_A(R) \ b \ c \ unlock_A(R) \ d$$

and the thread of process $Z$ is

$$w \ lock_Z(R) \ x \ y \ unlock_Z(R) \ z$$
Between events $\text{lock}_A(R)$ and $\text{unlock}_A(R)$, the resource $R$ is unavailable to process $Z$; that is, event $\text{lock}_Z(R)$ cannot occur while process $A$ maintains a lock on $R$. Similarly, $\text{lock}_A(R)$ cannot occur between $\text{lock}_Z(R)$ and $\text{unlock}_Z(R)$. Competition for the resource therefore constrains the possible interleavings of the threads for $A$ and $Z$.

The competition between $A$ and $Z$ for resource $R$ is illustrated geometrically in Fig. 12.7. The line stepping to the right and up from the origin corresponds to the interleaving

$$a \quad w \quad \text{lock}_A(R) \quad b \quad c \quad \text{unlock}_A(R) \quad \text{lock}_Z(R) \quad x \quad d \quad y \quad \text{unlock}_Z(R) \quad z$$
Competition for a resource constrains interleaving.
The Dining Philosophers

Five philosophers sit at a table, alternating between eating spaghetti and thinking. In order to eat, a philosopher must have two forks (or chopsticks, as in Fig. 12.8). The problem is that there is a single fork between each pair of philosophers, so if one of them is eating, a neighboring one cannot be eating. A philosopher puts down both forks in their place before thinking.
Deadlock: The Inability to Proceed

A concurrent program is in deadlock if all processes are waiting, unable to proceed. For an example of deadlock, suppose that each philosopher executes the following pseudocode:

```
loop
    pick up the fork to the left;
    pick up the fork to the right;
    eat;
    release the forks;
    think;
end;
```
Livelock: No Process Makes Progress

Another situation illustrated by the dining-philosophers problem is *livelock*, in which the system is not in deadlock but no process makes any progress. Suppose that a philosopher’s program is changed so that the left fork is released if the right fork is not available. Livelock occurs if all philosophers go into the infinite loop:

```
  pick up left fork;
  release left fork;
  pick up left fork
  release left fork;
  ...
```
Fairness

This chapter makes the finite-progress assumption; that is, any process that wants to run will be able to do so within a finite amount of time. In other words, a process that wants to run cannot be blocked indefinitely.

The fairness assumption implies that any philosopher who wants to eat eventually will be able to eat. An unfair solution is to let just one philosopher eat all the time, with the other processes perpetually waiting. Fairness is a delicate issue, because processes may run at different rates. Thus, strict alternation between processes P and Q

\[ P, Q, P, Q, P, Q, \ldots \]

may be unfair if P is 10 times as fast as Q and is needlessly delayed until Q takes its turn.
Deadlock Prevention

- Z
- release $R, S$
- pick $R$
- pick $S$
- A
- pick $R$
- pick $S$
- release $R, S$
- Infeasible
- Deadlock
Deadlock can be prevented by ordered resource usage, whereby all processes request resources in the same order. When philosophers A and Z both pick up forks in the same order RS, then deadlock cannot occur. If R is unavailable, then S is left untouched, so the philosopher who gets R will be able to pick up S as well. The corresponding diagram in Fig. 12.9(b) has an infeasible region but no region marked deadlock. Ordered resource usage works for any number of resources.
release $R, S$

pick $S$

pick $R$

$Z$

pick $R$

pick $S$

release $R, S$

Infeasible
We now formulate a notion of safety that imposes constraints on interleaving without insisting on a unique deterministic result.

A critical section in a process is a portion or section of code that must be treated as an atomic event. Two critical sections are said to be mutually exclusive because their executions must not overlap.

A concurrent program with critical sections is safe if it executes the critical sections contiguously, without interleaving. The two cyclic processes in Fig 12.10 are allowed to execute their critical sections in any order, even if $P$ executes more often than $Q$. 
Figure 12.10  Two cyclic processes with critical sections.
Example 12.2  For a technical example of the need for critical sections, consider the following two assignments:

\[ x := x + 1; \quad x := x + 2; \]

In a sequential language, these assignments increment the value of \( x \) by 3. This behavior remains the same if the order of the assignments is reversed:

\[ x := x + 2; \quad x := x + 1; \]
A concurrent language, however, might not treat an assignment as an atomic event. Suppose that these assignments are split and implemented as two concurrent processes $P$ and $Q$:

**PROCESS P**

\[
\begin{align*}
t & := x; \\
x & := t + 1;
\end{align*}
\]

**PROCESS Q**

\[
\begin{align*}
u & := x; \\
x & := u + 2;
\end{align*}
\]

If the assignments of $Q$ are interleaved between those of $P$,

\[
\begin{align*}
t & := x; \\
u & := x; \\
x & := u + 2; \\
x & := t + 1;
\end{align*}
\]

then $x$ is incremented by 1 instead of 3. Such interleaving can be prevented by treating the assignments as critical sections. $\Box$
SYNCHRONIZED ACCESS TO SHARED VARIABLES

(a) Direct access

(b) Synchronized direct access

(c) Access through a monitor

(d) The buffer as a separate process
with text_io; use text_io;

procedure direct is

    size : constant integer := 5;
    buf : array(0..size-1) of character;
    front, rear : integer := 0;

    function notfull return boolean is ... end notfull;
    function notempty return boolean is ... end notempty;

    task producer;
    task body producer is
        c : character;
        begin
            while not end_of_file loop
                if notfull then
                    get(c);
                    buf(rear) := c;
                    rear := (rear + 1) mod size;
                    end if;
                end loop;
            end producer;

    task consumer;
    task body consumer is
        c : character;
        begin
            loop
                if notempty then
                    c := buf(front);
                    front := (front + 1) mod size;
                    put(c);
                    end if;
                end loop;
            end consumer;

    begin
        null;
    end direct;
Problem!

\begin{align*}
\text{producer} & \quad \text{consumer} \\
\text{notfull} & \text{ returns } \texttt{true;} \\
\text{get}(c); & \\
\text{buf}(\text{rear}) := c; & \\
\text{notempty} & \text{ returns } \texttt{true;} \\
& \text{ c := \text{buf}(\text{front}); } \\
& \text{front := (front + 1) \ mod \ size; } \\
\text{rear} := (\text{rear} + 1) \ mod \ size; & \\
\end{align*}

Note that the consumer touches the buffer between the time the producer assigns a value to \text{buf}(\text{rear}) and the time it updates the index \text{rear}.
Semaphores: Mutual Exclusion

A semaphore is a construct that has an integer variable value and supports two operations:

1. If value $\geq 1$, then a process can perform a $p$ operation to decrement the value by 1. Otherwise, a process attempting a $p$ operation waits until the value becomes greater than or equal to 1.
2. A process can perform a $v$ operation to increment variable value by 1.$^4$

A binary semaphore is a semaphore whose value is constrained to be either 0 or 1. If the value of a binary semaphore is 1, then a process attempting a $v$ operation on it is suspended until its value becomes 0. In other words, the $p$ and $v$ operations on a semaphore must be performed alternately.
Mutual exclusion can be implemented by enclosing each critical section between the operations \( s.p \) and \( s.v \), where \( s \) is a binary semaphore:

\[
\begin{align*}
\text{process } Q & \\
\cdots & \\
\text{critical section for } Q; & \\
s.v & \\
\cdots & \\
\text{process } R & \\
\cdots & \\
s.p; & \\
\text{critical section for } R; & \\
s.v & \\
\cdots &
\end{align*}
\]
Although the producer and consumer can indeed be synchronized by treating the highlighted code segments in Fig. 12.17 as critical sections, the code in the figure has another failing—namely, busy waiting. If the buffer is full, the producer busily loops and tests until the buffer is not full. Similarly, if the buffer is empty, the consumer busily loops and tests until the buffer is not empty. Busy waiting will be avoided in the solutions discussed next.

\[\begin{align*}
\text{producer} \\
\text{...} \\
\text{if notfull then} \\
\text{get}(c); \\
\text{buf}(\text{rear}) := c; \\
\text{update rear;} \\
\text{end if;} \\
\text{...}
\end{align*}\]

\[\begin{align*}
\text{consumer} \\
\text{...} \\
\text{if notempty then} \\
\text{c := buf(front);} \\
\text{update front;} \\
\text{put(c);} \\
\text{end if;} \\
\text{...}
\end{align*}\]

Figure 12.17 The producer and consumer as cyclic processes with critical sections.
Semaphores and a Bounded Buffer

Solution to the bounded buffer problem uses 3 semaphores.

critical : binary semaphore;  critical.value = 1;
filling: general semaphore;    filling.value = n
emptying:  gs;         emptying.value = 0;
Solution using Semaphores

\[
\begin{align*}
\text{task body } & \text{ producer is} \\
& c : \text{ character;} \\
& \begin{align*}
& \text{begin} \\
& \quad \text{while not } \text{ end\_of\_file } \text{ loop} \\
& \quad \quad \text{get}(c); \\
& \quad \quad \text{filling}\.p; \\
& \quad \quad \text{critical}\.p; \\
& \quad \quad \quad \text{buf}(\text{rear}) := c; \\
& \quad \quad \quad \text{rear} := (\text{rear} + 1) \mod \text{size}; \\
& \quad \quad \text{critical}\.v; \\
& \quad \quad \text{emptying}\.v; \\
& \quad \text{end loop;} \\
& \text{end producer;} \\
\end{align*}
\end{align*}
\]

\[
\begin{align*}
\text{task body } & \text{ consumer is} \\
& c : \text{ character;} \\
& \begin{align*}
& \text{begin} \\
& \quad \text{loop} \\
& \quad \quad \text{emptying}\.p; \\
& \quad \quad \text{critical}\.p; \\
& \quad \quad \quad c := \text{buf}(\text{front}); \\
& \quad \quad \quad \text{front} := (\text{front} + 1) \mod \text{size}; \\
& \quad \quad \text{critical}\.v; \\
& \quad \quad \text{filling}\.v; \\
& \quad \quad \text{put}(c); \\
& \quad \text{end loop;} \\
& \text{end consumer;} \\
\end{align*}
\end{align*}
\]

\textbf{Figure 12.19} Use of the semaphores \textit{filling, emptying, and critical}.
A Brief Look at Monitors

Critical sections and semaphores represent an early approach to providing exclusive access to shared data. Another approach is to encapsulate the shared data in a construct called a monitor, which is a generalization of the class construct of Chapters 6 and 7. A monitor object is a collection of shared variables and procedures with the constraint that only one process is allowed to execute a monitor procedure at a time. In other words, the thread of at most one process can be within a monitor at a time.
monitor buffer is
  buf : ...;

  procedure enter(c : in character);
  begin
    if buffer full then wait(filling); -- block producer
    enter c into buffer;
    ... signal(emptying); -- unblock consumer
  end enter;

  procedure leave(c : out character);
  begin
    if buffer empty then wait(emptying); -- block consumer
    c := next character;
    ... signal(filling); -- unblock producer
  end leave;

begin
  initialize private data;
end buffer;

Figure 12.20  A monitor for a bounded buffer.
If the producer process is executing procedure `enter`, then the consumer process is not allowed to execute procedure `leave`, and vice versa. Monitors allow a language to check and enforce synchronized access to shared data.

The main problem with monitors is that a process may block within a monitor. For example, suppose that a producer grabs a buffer monitor and starts to execute the `enter` procedure. What if the buffer is full? We face a problem because the producer has exclusive use of the monitor and finds it cannot complete execution of the procedure it has started.

Processes that block inside a monitor can be handled by maintaining queues of blocked processes. Execution by a process $P$ of

$$\text{wait}(q);$$

blocks $P$ on the queue $q$. Subsequently, if a process $R$ executes

$$\text{signal}(q);$$

then a blocked process, if any, is taken off the queue $q$ and restarted.
Procedure *enter* uses a queue *filling* to hold a producer that blocks when the buffer is full. Symmetrically, procedure *leave* uses a queue *emptying* to hold a consumer that blocks when the buffer is empty.

To see how queues are used, suppose that the buffer is not empty, and that a producer executes *enter*. After character *c* is entered into the buffer, the procedure body executes

\[
\text{signal(emptying)}
\]

This signal has no effect if there is no blocked process in queue *emptying*; otherwise, a blocked process is taken off the queue and restarted.

Similarly, just before control exits from procedure *leave*,

\[
\text{signal(filling)}
\]

allows a blocked producer, if any, to resume execution.