Programming Languages

Concurrency

Benjamin J. Keller

Department of Computer Science, Virginia Tech
Overview

- Classification of machine architectures
- Thread Creation
- Language mechanisms to support concurrency
Notation: S – single, M – multiple, I – instruction, D – data

- SISD — traditional von Neuman architecture
- SIMD — typical supercomputer (synchronous) prior to clusters
  - Instruction pipeline to interleave operations for speed
  - Array-processor — do same thing to all elements of an array
- MIMD — asynchronous operation
Shared Memory

- Synchronization issues: must wait for process to finish and produce output before next can execute
- Generalization of sequential algorithms not necessarily fastest parallel algorithm
- Problems with resource contention (e.g., reader-writer problem)
- Cache coherence problem — different copies of variables saved in processor caches
Distributed Computing System

- Usually no global memory — each processor has its own memory
- Shared common communications channel
Terminology

- *Concurrency* — having two or more execution contexts active at the same time
- May be provided by OS that supports multiprogramming, or parallel hardware
- *Process* — basic execution context supported by OS
- Typically instance of program scheduled for independent execution
- *Thread* — separate execution context of concurrent program
- Implemented on one or more processes
- heavyweight vs lightweight — whether has own address space or not
Thread Creation Mechanisms

- Special control flow:
  - Co-begin
  - Parallel loops
- Subroutine-like:
  - Launch at elaboration — declaration comes into scope
  - fork (and join)
  - implicit receipt — bind communication channel to receiving code
  - early reply — subroutine that can return result before done
Algod 68 Co-Begin

- Variations on block: semicolon for sequential, comma for nondeterminism
- Precede block with `par` for concurrent semantics

```plaintext
par begin
  p (a,b,c), # concurrent #
begin
  d := q(e,f); # sequential #
  r(d,g,h);
end,
s(i,j)
end
```

- In Occam label block as either `par` or `seq`
Parallel Loops

- Loop with concurrent iterations
- Occam
  
  \[
  \text{par } i = 5 \text{ for } 6 \\
  p(a,b,i)
  \]
- Also common in dialects of Fortran
Launch-At-Elaboration

- Declare thread of control as a parameterless procedure
- Starts when declaration is elaborated
- Example: Ada task
  
  procedure P is
    task T is
      ...
      end T;
    begin -- P
      ...
      end P;
  
  - When control enters P instance of task T is started
  - May be many instances executing
  - P may not exit until corresponding instance of T ends


Language Mechanisms

- Three major mechanisms:
  - Semaphores (for mutual exclusion)
  - Monitors (for mutual exclusion)
  - Message passing (using “tasks”)

- Producer/Consumer or Bounded Buffer problem:
  Two processes cooperating, one by adding items to a buffer, the other removing items. Prevent removing when nothing there, and buffer overflow as well.
Semaphores

- Support mutual exclusion and synchronization in shared-memory model.
- Three operations:
  - InitSem(S: Semaphore; value: integer);
  - Wait(S: Semaphore);
  - Signal(S: Semaphore);
- InitSem initializes semaphore with (non-negative) value
Semaphores

- **Wait(S):** If $S > 0$ then $S := S - 1$ else suspend self

- **Signal(S):** if processes are waiting, then wake up a process, else $S := S + 1$;

- Think of $\text{Wait}(S)$ as claiming a resource so that no one else can get it, while $\text{Signal}(S)$ releases the resource.

- In order to solve mutual exclusion problem, must ensure that $\text{Wait}$ and $\text{Signal}$ execute atomically (i.e., cannot be interrupted and no one else can execute at same time).
Critical Section

- If start with $S = 1$ then protect a critical region by:
  
  \[
  \text{Wait}(S); \quad \text{-- grab token}
  \]
  
  \{
  \text{Critical region}
  \}
  
  \[
  \text{Signal}(S); \quad \text{-- release token}
  \]

- Can also start with other values of $S$

- Example: if start with $S = 0$ and call \text{Wait}(S) then suspend execution until another process executes \text{Signal}(S)
Bounded Buffer Solution

- Suppose also have procedures:
  - \texttt{CreateProcess(p:PROC; workspacesize: CARDINAL);}
    \quad --- Creates nameless process
  - \texttt{StartProcesses; -- starts all processes which have been created}
  - \texttt{Terminate; -- stop execution of process}

- When all processes are terminated control returns to unit calling \texttt{StartProcesses}. 
**Bounded Buffer Main Program**

CreateProcess(Producer, WorkSize); -- create producer
CreateProcess(Consumer, WorkSize); -- create consumer
BufferStart := 1; BufferEnd := 0
InitSem(NonEmpty, 0) -- initial to 0 for empty
InitSem(NonFull, MaxBuffSize) -- init to size of buffer
InitSem(MutEx, 1) -- used for mutual exclusion
StartProcesses
end;
Producer

Procedure Producer;
begin
  loop
    read(ch)
    Wait(NonFull);
    Wait(MutEx);
    BufferEnd := BufferEnd MOD MaxBuffSize + 1;
    Buffer[BufferEnd] := ch;
    Signal(MutEx);
    Signal(NonEmpty);
  end loop;
end;
Procedure Consumer;
begin
  loop
    Wait(NonEmpty);
    Wait(MutEx);
    ch := Buffer[BufferStart];
    BufferStart := BufferStart MOD MaxBuffSize + 1;
    Signal(MutEx);
    Signal(NonFull);
    Write(ch)
  end loop
end;
Bounded Buffer Solution

- Technically MutEx is not necessary here since Producer only changes BufferEnd, while Consumer only changes BufferStart.
- Would be important if they both changed a count of the number of items in the buffer.
- What would go wrong if you changed the order of the two Wait’s at the beginning of either Producer or Consumer?
- Semaphores are too low level and unstructured. Accidentally reversing order or Wait’ing twice could be disastrous.
Monitors

- Monitors are a much higher-level construct to support mutual exclusion and synchronization

- ADT mechanism encapsulating shared data and operations. At most one process can be in monitor at a time

- Provides condition variables, each with a waiting queue of processes and associated suspend and continue operations

- Suspend puts process on queue, continue results in removing next process from queue for execution
Monitors in Concurrent Pascal

- Continue by process in monitor causes it to be immediately suspended.
- Each queue can only hold one suspended process at a time (error otherwise)
Bounded Buffer Problem

type buffer = monitor;

var store: array[1..MaxBuffSize] of char;
  BufferStart, BufferEnd, BufferSize: integer
  nonfull, nonempty: queue;

procedure entry insert(ch: char);
begnin
  if BufferSize = MaxBuffSize then delay(nonfull);
  BufferEnd := BufferEnd mod MaxBuffSize + 1;
  store[BufferEnd] := ch;
  BufferSize := BufferSize + 1;
  continue(nonempty)
end;
**Bounded Buffer Problem**

procedure entry delete(var ch: char);
begin
    ifBufferSize = 0 then delay(nonempty);
    ch := store[BufferStart];
    BufferStart := BufferStart mod MaxBuffSize + 1;
    BufferSize := BufferSize - 1;
    continue(nonfull);
end;

begin (* initialization *)
    BufferEnd := 0;
    BufferStart := 1;
    BufferSize := 0
end;
type producer = process (b: buffer);
  var ch: char;
  begin
    while true do begin
      read(ch);
      b.insert(ch)
    end;
  end

type consumer = process(b: buffer);
  var ch: char;
  begin
    while true do begin
      b.delete(ch);
      write(ch)
    end
  end;
Bounded Buffer Problem

```plaintext
var p: producer;
  q: consumer;
  b: buffer;
begin
  init b, p(b), q(b)
end.
```
Message Passing

- With distributed systems, cannot rely on shared variables for synchronization and communication

- With message passing, rely on send and receive operations:
  - Procedure Send(To: Process; M: Message);
  - Procedure Receive(From: Process; M: Message);

- Variants exist in which drop “To” and “From” fields (broadcast messages)
Synchronization

- Synchronization questions:
  - Does sender have to wait for receiver to accept message or keep in buffer?
  - Does receiver have to wait for message or can just check to see if any?
- If wait for each other, called “rendezvous”. Otherwise often use a mailbox to store messages.
Ada Tasks

- Ada’s tasks combine features of monitors and message passing.
- Process in Ada is called a task (defined much like a package).
- Task exports entry names to world (like a monitor) which can be “called”
  - Entries can have parameters as well.
  - Each entry has associated queue (managed as FIFO)
Ada Tasks

- Task accepts an entry call via `accept` statement (not name sender).

- Select allows choice of which accept is selected.

  ```
  select
      [when <cond> =>] <select alternative>
      {or [when <cond> =>] <select alternative>}
      [else <statements>]
  end select
  ```

  when statements form guards.

- System selects one of `when` conditions that is true and has an `accept` process waiting.

- Caller is blocked only during body of `accept` clause
Example

task Buffer is       -- specification
    entry insert(ch: in CHARACTER);
    entry delete(ch: out CHARACTER);
end;
Example

task body Buffer is
MaxBuffSize: constant INTEGER := 50;
  Store: array(1..MaxBuffSize) of CHARACTER;
BufferStart: INTEGER := 1;
BufferEnd: INTEGER := 0;
BufferSize: INTEGER := 0;
begin
  loop
    select
      when BufferSize < MaxBufferSize =>
        accept insert(ch: in CHARACTER) do
          BufferEnd := BufferEnd mod MaxBuffSize + 1;
          Store(BufferEnd) := ch;
        end;
        BufferSize := BufferSize + 1;
      or when BufferSize > 0 =>
        accept delete(ch: out CHARACTER) do
          ch := Store(BufferStart);
        end;
        BufferStart := BufferStart mod MaxBuffSize + 1;
        BufferSize := BufferSize - 1;
    end select;
  end loop
end Buffer;
task type Producer;
task body Producer is
  ch: CHARACTER;
begin
  loop
    TEXT_IO.GET(ch);
    Buffer.insert(ch);
  end loop;
end Producer;

task type Consumer;
task body Consumer is
  ch: CHARACTER;
begin
  loop
    Buffer.delete(ch);
    TEXT_IO.PUT(ch);
  end loop;
end Consumer;
Bounded Buffer Problem

- Note Producer/Consumer run forever - no entries.
- Exit from task when at end and no open subprocesses or waiting with terminate alternative open and parent has executed to completion. If this true of all children of process then all terminate simultaneously.
Comparison

- Semaphores very low level.
- Monitors are passive regions encapsulating resources to be shared (mutual exclusion). Cooperation enforced by delay and continue statements.
- Everything active in Ada tasks (resources and processes)
- Monitors and processes can easily be structured as Ada tasks and vice-versa.
- Ada tasks better represents behavior of distributed system.