Synchronous C + WCRT Algebra 101

Reinhard von Hanxleden
Joint work with Michael Mendler, Claus Traulsen, ...

Real-Time and Embedded Systems Group (RTSYS)
Department of Computer Science
Christian-Albrechts-Universität zu Kiel
www.informatik.uni-kiel.de/rtsys

SYNCHRON 2011, Le Bois du Lys
Overview

**Synchronous C**
- SC Basics
- An Alternative SC Syntax
- Summary

**WCRT Algebra**
The SC-Story From Last Time

How to get deterministic concurrency?
- Deterministic decision on when to perform context switch
- Deterministic decision on what thread to switch to

Idea: Cooperative thread scheduling at application level
- As in co-routines, threads decide *when* to resume another thread (static)
- However, a *dispatcher* decides *what* thread to resume (dynamic)
- Reactive control flow implemented at application level—are still executing a sequential C program!
5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel. The offsets to achieve this are given by deadlines at the beginning of the program. The offset of the producer loop is \( 28 \times 6 = 168 \) cycles, which is 78 cycles less than the offset of \( 41 \times 6 = 246 \) for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the data an earlier rotation of the wheel. Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, each loop iteration takes \( 26 \times 6 = 156 \) cycles: exactly two rotations of the wheel.
5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel. The offsets to achieve this are given by deadlines at the beginning of the program. The offset of the producer loop is $28 \times 6 = 168$ cycles, which is 78 cycles less than the offset of $41 \times 6 = 246$ for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the data an earlier rotation of the wheel. Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, each loop iteration takes $26 \times 6 = 156$ cycles: exactly two rotations of the wheel.
Figure 5: Simple Producer/Consumer Example

5.1 Mutual Exclusion

A general approach to managing shared data across separate threads is to have mutually exclusive critical sections that only a single thread can access at a time. Our memory wheel already guarantees that any accesses to a shared word will be atomic, so we only need to ensure that these accesses occur in the correct order.

Figure 5 shows the C code for the producer, consumer, and an observer all accessing a shared variable (underlined). The producer iterates and writes an integer value to a shared data. The consumer reads this value from this shared data and stores it in an array. For simplicity, our consumer does not perform any other operations on the consumed data or overwrite the data after reading it. The observer also reads the shared data and writes it to a memory-mapped peripheral. We use staggered deadlines to offset the threads to force a thread ordering. The deadline instructions are marked in bold.

As Figure 5 shows, every loop iteration first executes the critical section of the producer, and then the observer and the consumer in parallel. The offsets to achieve this are given by deadlines at the beginning of the program. The offset of the producer loop is $28 \times 6 = 168$ cycles, which is 78 cycles less than the offset of $41 \times 6 = 246$ for the consumer and observer. Since this difference is the same as the frequency with which the wheel schedule repeats, this guarantees the producer thread will access the data an earlier rotation of the wheel. Once inside the loop, deadlines force each thread to run at the same rate, maintaining the memory access schedule. It is important for this rate to be a multiple of the wheel rate to maintain the schedule. In this example, each loop iteration takes $26 \times 6 = 156$ cycles: exactly two rotations of the wheel.

Lickly et al., Predictable Programming on a Precision Timed Architecture, CASES'08
Producer-Consumer-Observer in SC

```c
int tick ()
{
    static int BUF, fd, i, j,
    k = 0, tmp, arr[8];

    TICKSTART(1);

    PCO:
    FORK(Producer, 3);
    FORK(Consumer, 2);
    FORKE(Observer);

    Producer:
    for (i = 0; ; i++) {
        PAUSE;
        BUF = i;
    }

    Consumer:
    for (j = 0; j < 8; j++)
        arr[j] = 0;
    for (j = 0; ; j++) {
        PAUSE;
        tmp = BUF;
        arr[j % 8] = tmp;
    }

    Observer:
    for ( ; ; ) {
        PAUSE;
        fd = BUF;
        k++; }

    TICKEND;
}
```
```c
int tick ()
{
    static int BUF, fd, i, j, k = 0, tmp, arr [8];
    TICKSTART(1);

    PCO:
    FORK(Producer, 3);
    FORK(Consumer, 2);
    FORKE(Observer);

    Producer:
    for (i = 0; ; i++) {
        PAUSE;
        BUF = i;
    }
    Observer:
    for ( ; ; ) {
        PAUSE;
        fd = BUF;
        k++;
    }
    TICKEND;

    Consumer:
    for (j = 0; j < 8; j++)
        arr [j] = 0;
    for (j = 0; ; j++) {
        PAUSE;
        tmp = BUF;
        arr [j % 8] = tmp;
    }
}
```
```c
int tick () {
    static int BUF, fd, i, j,
    k = 0, tmp, arr[8];
    TICKSTART(1);
    PCO:
    FORK(Producer, 4);
    FORK(Consumer, 3);
    FORK(Observer, 2);
    FORKE(Parent);
    Producer:
    for (i = 0; ; i++) {
        BUF = i;
        PAUSE;
    }
    Consumer:
    for (j = 0; j < 8; j++) {
        arr[j] = 0;
        for (j = 0; ; j++) {
            tmp = BUF;
            arr[j % 8] = tmp;
            PAUSE; }
            fd = BUF;
            k++;
            PAUSE; }
            Observer:
            for ( ; ; ) {
                fd = BUF;
                k++;
                PAUSE; }
                Parent:
                while (1) {
                    if (k == 20)
                        TRANS(Done);
                    if (BUF == 10)
                        TRANS(PCO);
                    PAUSE;
                }
                Done:
                TERM;
                TICKEND;
```
Going on the Road . . .
Criticisms of Original SC

- Hard to understand what’s going on
- Thread structure gets lost
- What does FORK/FORKE mean?
Proposed Alternative Syntax

- Replace labels by explicit “Thread” and “State” declarations
- Add syntactic scopes (braces)
- Predefined FORK1/FORK2/... operators
```c
int tick ()
{
    static int BUF, fd, i, j,
    k = 0, tmp, arr [8];

    TICKSTART(1);

    PCO:
    FORK(Producer, 4);
    FORK(Consumer, 3);
    FORK(Observer, 2);
    FORK(Parent);

    Producer:
    for (i = 0; ; i++) {
        BUF = i;
        PAUSE;
    }

    Consumer:
    for (j = 0; j < 8; j++) {
        arr [j] = 0;
        for (j = 0; ; j++) {
            tmp = BUF;
            arr [j % 8] = tmp;
            PAUSE;
        }
    }

    Observer:
    for ( ; ; ) {
        fd = BUF;
        k++;
        PAUSE;
    }

    Parent:
    while (1) {
        if (k == 20)
            TRANS(Done);
        if (BUF == 10)
            TRANS(PCO);
        PAUSE;
    }

    Done:
    TERM;
    TICKEND;
}
```
Transforming the PCO-Example (1)

1. Separate threads

```c
1 int tick ()
2 {
3     static int BUF, fd, i, j,
4         k = 0, tmp, arr [8];
5
6     TICKSTART(1);
7
8     PCO:
9         FORK(Producer, 4);
10        FORK(Consumer, 3);
11        FORK(Observer, 2);
12        FORKE(Parent);
13
14     Parent:
15         while (1) {
16             if (k == 20)
17                 TRANS(Done);
18             if (BUF == 10)
19                 TRANS(PCO);
20             PAUSE;
21         }
22
23     Done:
24         TERM;
25
26     Producer:
27         for (i = 0; i++ ) {
28             BUF = i;
29             PAUSE;
30         }
31
32     Consumer:
33         for (j = 0; j < 8; j++)
34             arr[j] = 0;
35         for (j = 0; j++ ) {
36             tmp = BUF;
37             arr[j % 8] = tmp;
38             PAUSE;
39         }
40
41     Observer:
42         for ( ; ; ) {
43             fd = BUF;
44             k++;
45             PAUSE;
46         }
47
48     TICKEND;
49```
Transforming the PCO-Example (2)

```c
1 int tick ()
2 {
3   static int BUF, fd, i, j,
4   k = 0, tmp, arr [8];
5
6   TICKSTART(1);
7
8   PCO:
9   FORK3(Producer, 4,
10      Consumer, 3,
11      Observer, 2);
12
13   while (1) {
14     if (k == 20)
15       TRANS(Done);
16     if (BUF == 10)
17       TRANS(PCO);
18     PAUSE;
19   }
20
21   Done:
22   TERM;
23
24 Producer:
25     for (i = 0; i++ ) {
26       BUF = i;
27       PAUSE; }
28
29 Consumer:
30     for (j = 0; j < 8; j++)
31       arr [j] = 0;
32     for (j = 0; j++ ) {
33       tmp = BUF;
34       arr [j % 8] = tmp;
35       PAUSE; }
36
37 Observer:
38     for ( ; ; ) {
39       fd = BUF;
40       k++;
41       PAUSE; }
42
43 TICKEND;
44}
```

1. Separate threads
2. Consolidate FORK
Transforming the PCO-Example (3)

```c
1  int tick ()
2  {
3     static int BUF, fd, i, j,
4         k = 0, tmp, arr[8];
5
6     MainThread (1) {
7         PCO:
8         FORK3(Producer, 4,
9             Consumer, 3,
10            Observer, 2);
11
12         while (1) {
13             if (k == 20)
14                 TRANS(Done);
15             if (BUF == 10)
16                 TRANS(PCO);
17             PAUSE;
18         }
19
20     Done:
21         TERM;
22 }
23
24     Thread (Producer) {
25         for (i = 0; ; i++) {
26             BUF = i;
27             PAUSE; }
28 }
29
30     Thread (Consumer) {
31         for (j = 0; j < 8; j++)
32             arr[j] = 0;
33         for (j = 0; ; j++) {
34             tmp = BUF;
35             arr[j % 8] = tmp;
36             PAUSE; }
37 }
38
39     Thread (Observer) {
40         for ( ; ; ) {
41             fd = BUF;
42             k++;
43             PAUSE; }
44 }
45
46     TICKEND;
47 }
```

1. Separate threads
2. Consolidate FORK
3. Add Thread scopes
Transforming the PCO-Example (4)

1. Separate threads
2. Consolidate FORK
3. Add Thread scopes
4. Add States

```c
int tick ()
{
    static int BUF, fd, i, j, k = 0, tmp, arr [8];

    MainThread (1) {
        State (PCO) {
            FORK3(Producer, 4, Consumer, 3, Observer, 2);
            while (1) {
                if (k == 20)
                    TRANS(Done);
                if (BUF == 10)
                    TRANS(PCO);
                PAUSE;
            }
        }
        State (Done) {
            TERM;
        }
    }

    Thread (Producer) {
        for (i = 0; ; i++) {
            BUF = i;
            PAUSE; }
    }

    Thread (Consumer) {
        for (j = 0; j < 8; j++)
            arr[j] = 0;
        for (j = 0; ; j++) {
            tmp = BUF;
            arr[j % 8] = tmp;
            PAUSE; }
    }

    Thread (Observer) {
        for ( ; ; ) {
            fd = BUF;
            k++; PAUSE; }
    }

TICKEND;
```
int tick ()
{
    static int BUF, fd, i, j, k = 0, tmp, arr[8];

    MainThread (1) {
        State (PCO) {
            FORK3(Producer, 4,
                Consumer, 3,
                Observer, 2);
            while (1) {
                if (k == 20)
                    TRANS(Done);
                if (BUF == 10)
                    TRANS(PCO);
                PAUSE;
            }
            Thread (Producer) {
                for (i = 0; ; i++) {
                    BUF = i;
                    PAUSE;
                }
            }
            Thread (Consumer) {
                for (j = 0; j < 8; j++)
                    arr[j] = 0;
                for (j = 0; ; j++) {
                    tmp = BUF;
                    arr[j % 8] = tmp;
                    PAUSE;
                }
            }
            Thread (Observer) {
                for ( ; ; ) {
                    fd = BUF;
                    k++;
                    PAUSE;
                }
            }
            State (Done) {
                TERM;
            }
        }
    }
    TICKEND;
    }

Still a sequential C program!
SC Summary

- Embeds reactive control flow constructs into C
  (Not discussed today: Synchronous Java)
- Light-weight + deterministic
- Multi-threaded, priority-based, co-routine like
- Full range of concurrency, preemption, signal handling

- Inspired by Kiel Esterel Processor (KEP)
  → Next part
References

Claus Traulsen, Torsten Amende, Reinhard von Hanxleden. Compiling SyncCharts to Synchronous C. DATE’11, Grenoble

Reinhard von Hanxleden. SyncCharts in C—A Proposal for Light-Weight, Deterministic Concurrency. EMSOFT’09, Grenoble
Outline

Synchronous C

WCRT Algebra
  WCRT Problem Statement
  WCRT Algebra
Worst Case Reaction Time (WCRT)

- Defined as **upper bound** for longest instantaneous path
- Measured e.g. in KEP instruction cycles
- Maximum time to react to given input with according output

Usage of WCRT:

- Safely determine whether deadlines are met
- Can eliminate reaction time jitter of KEP by setting variable _TICKLEN according to WCRT
WCRT vs. WCET

Worst Case Execution Time

- Compute maximal execution time for piece of code

Worst Case Reaction Time

- Compute maximal time to react: one valid program state to another
- Similar to stabilization time of circuits
**WCRT as Longest Path**

- Compute longest path between delay-nodes
- Abstract data-dependencies
- *Ad-hoc* optimizations

```
EMIT _TICKLEN,#11
A0: ABORT R,A1
    PAR 1,A2,1
    PAR 1,A3,2
    PARE A4,1
A2: AWAIT A
A3: AWAIT B
A4: JOIN 0
    EMIT 0
    HALT
A1: GOTO A0
```

▶ Compute longest path between delay-nodes
▶ Abstract data-dependencies
▶ *Ad-hoc* optimizations
 Interfaces

- **Approach:** use interface algebra to express WCRT
- **Solid theoretical basis**
- **Allows refinement,** eg. considering data-dependencies
- **Modular computation** (dynamic programming)
- **Computation:** \((\max, +)\)-algebra on timing matrix
Node Types

\[ T = \begin{pmatrix} d_{\text{thr}} & d_{\text{src}} \\ d_{\text{snk}} & d_{\text{int}} \end{pmatrix} : (\zeta \lor \text{active}) \supset (\odot \xi \ominus \odot \text{wait}) \]
Interface Types

\[ D : \phi \supset \psi \]

- Delay Matrix

\[ D = \begin{pmatrix}
  d_{11} & \cdots & d_{1n} \\
  \vdots & \ddots & \vdots \\
  d_{m1} & \cdots & d_{mn}
\end{pmatrix} \]

- Input Control

\[ \phi = \zeta_1 \lor \zeta_2 \lor \cdots \lor \zeta_m \]

- Output Control

\[ \psi = \circ \xi_1 \oplus \circ \xi_2 \oplus \cdots \oplus \circ \xi_n \]
Expressing the WCRT

- (6) : \( G_0 \supset \circ L_{11} \)
- (6, 4, 3, 1) :
  \( (G_0 \lor G_1 \lor G_3 \lor G_2) \supset \circ L_{11} \)
- (5, 5, 3, 4, 3, 1) :
  \( ((G_0 \land I) \lor (G_0 \land \neg I) \lor (G_1 \land I) \lor (G_1 \land \neg I) \lor G_3 \lor G_2) \supset \circ L_{11} \)
- (5, 3, 4, 3, 1) :
  \( (G_0 \lor (G_1 \land I) \lor (G_1 \land \neg I) \lor G_3 \lor G_2) \supset \circ L_{11} \)
- (5, 3, 4, 1) :
  \( (G_0 \lor ((G_1 \land I) \oplus G_3) \lor (G_1 \land \neg I) \lor G_2) \supset \circ L_{11} \)
- (5) : \( G_0 \supset \circ L_{11} \)
Summary/Outlook

- Algebraic approach allows to trade off precision and efficiency
- So far geared towards reactive processing ISA (KEP)
- Would like to lift this to higher level and mixed models (e.g. C/SC)

- To be continued ... → talk by M. Mendler
References

- Michael Mendler, Claus Traulsen, Reinhard von Hanxleden
  WCRT Algebra and Interfaces for Esterel-Style Synchronous Processing
  DATE’09, Nice

- Partha S. Roop, Sidharta Andalam, Reinhard von Hanxleden, Simon Yuan, Claus Traulsen
  Tight WCRT Analysis for Synchronous C Programs
  CASES’09, Grenoble

- Xin Li, Reinhard von Hanxleden
  Multi-Threaded Reactive Programming—The Kiel Esterel Processor
  IEEE Transactions on Computers 2011
Thanks!
Questions/Comments?
Appendix
Overview

Background
Explaining the (Original) Title
Inspiration: Reactive Processing

SC Operators
SC Thread Operators
SC Signal Operators
Further Operators

Thread Synchronization and Signals
Experimental Results
Explaining the (Original) Title: SyncCharts . . .

Reactive control flow:
- Sequentiality
- + Concurrency
- + Preemption

Statecharts [Harel 1987]:
- Reactive control flow
- + Visual syntax

SyncCharts [André 1996]:
- Statecharts concept
- + Synchronous semantics
... in C

Today’s Scenario 1: Develop model in SyncCharts, synthesize C
- Can use visual syntax
- Need special modeling tool
- Cannot directly use full power of classical imperative language

Today’s Scenario 2: Program “State Machine Pattern” in C
- Just need regular compiler
- Relies on scheduler of run time system—no determinism
- Typically rather heavyweight

SyncCharts in C scenario: Use SC Operators
- Light weight to implement and to execute
- Just need regular compiler
- Semantics grounded in synchronous model
The inspiration: Reactive processing

- SC multi-threading very close to Kiel Esterel Processor
- **Difference:** KEP dispatches at every instrClk, SC only at specific SC operators (such as PAUSE, PRIO)
### SC Thread Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TICKSTART</strong>*(init, p)**</td>
<td>Start (initial) tick, assign main thread priority p.</td>
</tr>
<tr>
<td><strong>TICKEND</strong></td>
<td>Return true (1) iff there is still an enabled thread.</td>
</tr>
<tr>
<td><strong>PAUSE</strong>++</td>
<td>Deactivate current thread for this tick.</td>
</tr>
<tr>
<td><strong>TERM</strong>*</td>
<td>Terminate current thread.</td>
</tr>
<tr>
<td><strong>ABORT</strong></td>
<td>Abort descendant threads.</td>
</tr>
<tr>
<td><strong>TRANS(l)</strong></td>
<td>Shorthand for ABORT; GOTO(l).</td>
</tr>
<tr>
<td><strong>SUSPEND</strong>*(cond)**</td>
<td>Suspend (pause) thread + descendants if cond holds.</td>
</tr>
<tr>
<td><strong>FORK(l, p)</strong></td>
<td>Create a thread with start address l and priority p.</td>
</tr>
<tr>
<td><strong>FORKE</strong>*(l)**</td>
<td>Finalize FORK, resume at l.</td>
</tr>
<tr>
<td><strong>JOINELSE</strong>++*(l_else)**</td>
<td>If descendant threads have terminated normally, proceed; else pause, jump to l_else.</td>
</tr>
</tbody>
</table>
## SC Signal Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SIGNAL($S$)</strong></td>
<td>Initialize a local signal $S$.</td>
</tr>
<tr>
<td><strong>EMIT($S$)</strong></td>
<td>Emit signal $S$.</td>
</tr>
<tr>
<td><strong>PRESENT($S$, $l_{else}$)</strong></td>
<td>If $S$ is present, proceed normally; else, jump to $l_{else}$.</td>
</tr>
<tr>
<td><strong>EMITINT($S$, $val$)</strong></td>
<td>Emit valued signal $S$, of type integer, with value $val$.</td>
</tr>
<tr>
<td><strong>EMITINTMUL($S$, $val$)</strong></td>
<td>Emit valued signal $S$, of type integer, combined with multiplication, with value $val$.</td>
</tr>
<tr>
<td><strong>VAL($S$, $reg$)</strong></td>
<td>Retrieve value of signal $S$, into register/variable $reg$.</td>
</tr>
<tr>
<td><strong>PRESENTPRE($S$, $l_{else}$)</strong></td>
<td>If $S$ was present in previous tick, proceed normally; else, jump to $l_{else}$. If $S$ is a signal local to thread $t$, consider last preceeding tick in which $t$ was active, i.e., not suspended.</td>
</tr>
<tr>
<td><strong>VALPRE($S$, $reg$)</strong></td>
<td>Retrieve value of signal $S$ at previous tick, into register/variable $reg$.</td>
</tr>
</tbody>
</table>
## Further Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GOTO($l$)</td>
<td>Jump to label $l$.</td>
</tr>
<tr>
<td>CALL($l$, $l_{ret}$)</td>
<td>Call function $l$ (e.g., an on exit function), return to $l_{ret}$.</td>
</tr>
<tr>
<td>RET</td>
<td>Return from function call.</td>
</tr>
<tr>
<td>ISAT($id$, $l_{state}$, $l$)</td>
<td>If thread $id$ is at state $l_{state}$, then proceed to next instruction (e.g., an on exit function associated with $id$ at state $l_{state}$). Else, jump to label $l$.</td>
</tr>
<tr>
<td>PPAUSE*($p$, $l$)</td>
<td>Shorthand for PRIO($p$, $l'$); $l'$: PAUSE($l$) (saves one call to dispatcher).</td>
</tr>
<tr>
<td>JPPAUSE*($p$, $l_{then}$, $l_{else}$)</td>
<td>Shorthand for JOIN($l_{then}$, $l$); $l$: PPAUSE($p$, $l_{else}$) (saves another call to dispatcher).</td>
</tr>
<tr>
<td>ISATCALL($id$, $l_{state}$, $l_{action}$, $l$)</td>
<td>Shorthand for ISAT($id$, $l_{state}$, $l$); CALL($l_{action}$, $l$)</td>
</tr>
</tbody>
</table>
Thread Synchronization and Signals

Recall: Threads may also communicate via signals

- In addition to thread operators, S provides signal operators (EMIT, PRESENT, PRE, valued/combined signals)
- Can handle signal dependencies and instantaneous communication via dynamic priorities
Edwards et al., JES'07; Prochnow et al., LCTES'06
FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORK(TMain);

T1: if (PRESENT(A)) {
    EMIT(B);
    PRI0(4);
    if (PRESENT(C))
        EMIT(D);
    PRI0(2);
    if (PRESENT(E)) {
        EMIT(T_);
        TERM;
    }
}
PAUSE;
EMIT(B);
TERM;

T2: if (PRESENT(B))
    EMIT(C);
TERM;

T3: if (PRESENT(D))
    EMIT(E);
TERM;
FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORK(TMain);

T1: if (PRESENT(A)) {
  EMIT(B);
  PRI0(4);
  if (PRESENT(C))
    EMIT(D);
  PRI0(2);
  if (PRESENT(E)) {
    EMIT(T_);
    TERM; }
}
PAUSE;
EMIT(B);
TERM;

T2: if (PRESENT(B))
  EMIT(C);
  TERM;

T3: if (PRESENT(D))
  EMIT(E);
  TERM;

TMain: if (PRESENT(T_)) {
  ABORT;
  TERM; }
JOINELSE(TMain);
TICKEND;
FORK(T1, 6);
FORK(T2, 5);
FORK(T3, 3);
FORKE(TMain);

T1: if (PRESENT(A)) {
    EMIT(B);
    PRIO(4);
    if (PRESENT(C))
        EMIT(D);
    PRIO(2);
    if (PRESENT(E)) {
        EMIT(T_);
        TERM;
    }
}
PAUSE;
EMIT(B);
TERM;

T2: if (PRESENT(B))
    EMIT(C);
    TERM;

T3: if (PRESENT(D))
    EMIT(E);
    TERM;

TMain: if (PRESENT(T_)) {
    ABORT;
    TERM;
}
JOINELSE(TMain);
TICKEND;

--- Sample Execution ---

TICK 0 STARTS, inputs = 01, enabled = 00

Inputs (id/name): 0/A

Enabled (id/state): <init>

FORKE: 1/_L_INIT continues at TMain

PRESENT: 6/T1 determines A/0 present
EMIT: 6/T1 emits B/1
PRIO: 6/T1 set to priority 4
PRESENT: 5/T2 determines B/1 present
EMIT: 5/T2 emits C/2
TERM: 5/T2 terminates, enabled = 073
PRESENT: 4/_L72 determines C/2 present
EMIT: 4/_L72 emits D/3
PRIO: 4/_L72 set to priority 2
PRESENT: 3/T3 determines D/3 present
EMIT: 3/T3 emits E/4
TERM: 3/T3 terminates, enabled = 017
PRESENT: 2/_L75 determines E/4 present
EMIT: 2/_L75 emits T_/5
TERM: 2/_L75 terminates, enabled = 07
PRESENT: 1/TMain determines T_/5 present

RESULT: Outputs OK.
Conciseness

Size of tick function in C source code, line count without empty lines and comments
Code Size

Size of tick function object code, in Kbytes
Code Size

Size of executable, in Kbytes
Performance

Accumulated run times of tick function, in thousands of clock cycles
Operator Density

SC operations count, ratio to clock cycles