AARHUS
UNIVERSITY

DEPARTMENT OF ENGINEERING

Test of Distributed Systems
Lecture 5

C-- BACI:

Concurrency

Semaphores

Monitors

14/04/14

C-- BACI programs

- Execution of a C-- BACI program starts with
  `int main () { … }` or
  `void main () { … }` or
  `main () { … }`
- I/O is only done by means of
  `cout` the output stream
  `cin` the input stream

C-- BACI variables

- Variables must be declared the beginning of code blocks

```
 {
   int x;
   …
  }
```

- This holds, in particular, for index variables of for loops

C-- BACI strings

- C-- BACI has a predefined string type, e.g.,
  `string[N] name;`
  declares a string of length *N* called *name*
- *N* denotes the number of characters of the string without the trailing *0*
- Attention: string bounds are **unchecked**
- `void proc(string name)`
  passes a string of arbitrary length to `proc`

C-- BACI arrays, typedefs, consts

- C-- BACI arrays use stand C-syntax:

```
int matrix[M][N];
```

- typedefs can be used:

```
typdef int length;
```

constants of simple types are supported:

```
const int MAX = 5;
```

C-- BACI initialisers

- For variables of types `int` and `char` initialisers are supported:

```
const int m = 5;
int j = m;
int k = 3;
int c = ‘a’;
```

C-- BACI functions

- Procedures and functions are supported
- Recursion is supported
- Parameters can be passed by value or by reference:

```
int func( int  a, /* by-value */
          int& b) /* by-ref. */
```

- `main()` must be the last function in the source file; execution starts with a call to `main()`

C-- BACI statements

- The statements
  `if-else, switch-case,`
  `for, while, do-while,`
  `break, continue`
  are as usual in C
- Code should always be bracketed

C-- BACI file inclusion

- A file part.cm can be included by writing

```
#include <part.cm>
```

or

```
#include “part.cm”
```

- They both means the same thing

C-- BACI extern declarations

- Variables can be declared extern
- Such variables can have any valid C-- type
- They cannot have initialisers
- only global variables can be extern
- Examples

```
extern int g;
extern char a[20];
extern int func( int k );
```

C-- BACI concurrency

- A C-- process is a `void` function:

```
void proc( … )
```

- processes `proc1, …, procN` can be run concurrently by enclosing them in a `cobegin` block:

```
cobegin {
  proc1( … ); …; procN( … );
 }
```

C-- BACI concurrency

- `cobegin` blocks cannot be nested
- they must appear in the `main` function
- the processes in the block are executed by interleaving
- the `main` function is suspended until all processes in the `cobegin` block have terminated
- the `main` function then resumes

C-- BACI semaphores

- kinds of semaphores: *N*-ary and binary
- *N*-ary:

```
semaphore s = N;
```

- binary:

```
binarysem b = 0;
```

- within a program semaphores are initialised using the built-in:

```
initialsem(sema, integer_expression)
```

C-- BACI semaphores

- *N*-ary and binary semaphores both have the type:
  `semaphore`
- the only difference is that
  for binary semaphores it is verified that its value is 0 or 1
- whereas for the general kind it only needs to be non-negative

C-- BACI semaphore semantics

- Two built-in functions on semaphores are provided:

```
void wait( semaphore& s );
```

and

```
void signal( semaphore& s );
```

C-- BACI wait

- `wait( sema );`
- if `sema>0`, then decrement `sema` by `1` and return allowing the caller to continue
- if `sema==0`, then suspend the caller
- `wait` is `atomic`

C-- BACI signal

- `signal( sema );`
- if `sema==0` and at least one process is suspended, wake one of the processes for continuation (the choice is non-deterministic)
- if no processes are waiting, then increment `sema` by `1`
- the caller is allowed to continue
- `signal` is `atomic`

C-- BACI monitors

- Syntax:

```
monitor name {
 variable and condition declarations
 function definitions
 init {
  …
 }
}
```

C-- BACI monitor constraints

- all functions in the monitor are visible outside the monitor
- all variables and conditions can only be accessed from within the monitor
- monitors can only be declared at the outmost level
- they cannot be nested

C-- BACI concurrency support

- Three constructs can be used by functions of a monitor:

```
condition ,
```

```
void wait( condition c );
```

and

```
void signal( condition c );
```

C-- BACI conditions

- Conditions can only be declared in monitors
- They should only be used as parameters to waitc and signalc

```
condition c;
```

C-- BACI wait condition

- `void waitc( condition c );`
  the caller is blocked (until `c` is signaled)
- `void waitc( condition c, int p );`
  the caller is blocked with priority `p` for being woken up
- the smaller `p`, the higher the priority

C-- BACI signal condition

- `void signalc( condition c );`
  wake some process waiting on `c` with highest priority
- if no process is waiting for `c`, then this function does nothing
- `wait/signal` semaphores are different from `waitc/signalc` conditions!
- the function `int empty( cond )` returns 1 if there are no processes waiting and 0 otherwise

C-- BACI immediate resumption

- The **immediate resumption requirement** says that a process waiting on a condition that has just been signaled should have priority in re-entering the monitor over new calls to monitor processes.
- It is implemented by suspending the signaler of a condition and picking one of the waiters on the condition with the appropriate priority to run.
- Because of this, monitor procedures that `signalc` a condition typically do so as their last instruction.
- *(Why is this important? Fairness?)*

C-- BACI monitor example

```
monitor monSemaphore {
 int semvalue;
 condition notbusy;

 void monP() {
  if (semvalue == 0) waitc(notbusy);
  else semvalue--;
 }

 void monV() {
  if (empty(notbusy)) semvalue++;
  else signalc(notbusy);
 }

 init{ semvalue = 1; }
}
```

Test-and-set

test-and-set

in C:

```
#define LOCKED 1
 int TestAndSet(int* lockPtr) {
     int oldValue;
     oldValue = SwapAtomic(lockPtr, LOCKED);
     return oldValue == LOCKED;
 }
```

Test-and-set

mutual exclusion:

in C:

```
volatile int lock = 0;

void Critical() {
    while (TestAndSet(&lock));
    critical section
//only one process can be in this section at a time
    lock = 0
//release lock when finished with critical section
}
```

(also see compare-and-swap)

C-- BACI atomic functions

```
atomic int test_and_set(int& target) {
 int u;
 u = target;
 target = 1;
 return u;
}

int lock = 0;

void proc(int id) {
 inti=0;
 while(i < 10) {
  while (test_and_set(lock)) /* wait */ ;
  cout << id;
  lock = 0;
  i++;
 }
}

main() {
 cobegin { proc(1); proc(2); proc(3); }
}
```

Is this the same `test_and_set` as the C-program before?

C-- BACI example: program 1

```
int sum = 0;

void add10() {
 int i;
 int tmp;
 for (i = 1; i <= 10; i++) {
  tmp = sum;
  sum = tmp + 1;
 }
}

void main() {
 cobegin {
  add10();
  add10();
 }
 cout << "Sum = " << sum << endl;
}
```

C-- BACI example: program 1

```
int sum = 0;

void add10() {
 int i;
 int tmp;
 for (i = 1; i <= 10; i++) {
  tmp = sum;
  sum = tmp + 1;
 }
}

void main() {
 cobegin {
  add10();
  add10();
 }
 if (sum == 20)
   cout << ”OK!” << endl;
 else
   cout << “!OK” << endl;
}
```

C-- BACI example 1

- The intention is that the result produced by program 1 is 20
- How can we verify that this holds?

```
byte sum = 0;
proctype add10() {
  byte i;
  byte tmp;
  i = 1; do :: i > 10 -> break
  :: else ->
     tmp = sum;
     sum = tmp+1;
     i = i+1
  od
}
```

```
init {
  atomic {
    run add10();
    run add10()
  }

  (_nr_pr == 1) ->
    assert sum == 20
}
```

C-- BACI example 1 (attempt 1)

- Need to instrument the program

```
byte sum = 0;
byte old = 0;
byte ok;
proctype add10() {
  byte i;
  byte tmp;
  i = 1; do :: i > 10 -> break
  :: else ->
     tmp = sum;
     sum = tmp+1;
     ok = (old <= sum);
     i = i+1
  od
}
```

```
init {
  atomic {
    run add10();
    run add10()
  }

  (_nr_pr == 1) ->
    assert (ok == (sum == 20))
}
```

C-- BACI example 1 (attempt 7)

- Need to instrument the program (use aspects?)

```
byte sum = 0;
byte ok = 1;

proctype add10() {
  byte i;
  byte tmp;
  i = 1; do :: i > 10 -> break
  :: else ->
     ok = ok && (sum <= tmp);
     tmp = sum;
     sum = tmp+1;
     i = i+1
  od
}
```

```
init {
  atomic {
    run add10();
    run add10()
  }

  (_nr_pr == 1) ->
    assert (!ok || (sum == 20))
}
```

C-- BACI example: program 1

- the instrumented program (what have we achieved?)

```
int sum = 0;
int ok = 1;

void add10() {
 int i;
 int tmp;
 for (i = 1; i <= 10; i++) {
  ok = ok && (sum <= tmp);
  tmp = sum;
  sum = tmp + 1;
 }
}
```

```
void main() {
 cobegin {
  add10();
  add10();
 }
 if (ok) {
  cout << “Test succeeded”;
 } else {
  cout << “Test failed”;
 }
 cout << "Sum = “
      << sum << endl;
}
```

C-- BACI example: program 1

- Claim: the following program is correct!

```
int sum = 0;
binarysem s = 1;

void add10() {
  int i;
  int tmp;
  for (i = 1; i <= 10;i++) {
    wait(s);
    tmp = sum;
    sum = tmp + 1;
    signal(s);
  }
}
```

```
void main() {
  cobegin {
   add10();
   add10();
  }
  cout << "Sum = " << sum << endl;
}
```

C-- BACI example: program 1

- Claim: the following program is correct!
- Try in Spin → yes!

```
byte sum = 0;
byte sem = 1;
proctype add10() {
  byte i;
  byte tmp;
  i = 1; do :: i > 10 -> break
  :: else ->
     atomic {
       sem>0;
       sem--
     }
     tmp = sum;
     sum = tmp+1;
     sem++;
     i = i+1
  od
}

init {
  atomic {
    run add10();
    run add10()
  }

  (_nr_pr == 1) ->
    assert sum == 20
}
```

C-- BACI example: program 2

- Analyse the following program!

```
const int M = 5;

binarysem fork[M];

void phil(int N) {
  int i;
  for (i=1; i<=10; i++) {
    wait(fork[N]);
    wait(fork[(N+1) % M]);
    cout << 'P' << N <<
            " is eating\n";
    signal(fork[(N+1) % M]);
    signal(fork[N]);
  }
}
```

```
void main() {
  int k;
  for (k=0; k<M; k++) {
    initialsem(fork[k], 1);
  }
  cobegin {
    phil(0); phil(1);
    phil(2); phil(3); phil(4);
  }
  cout << "finished dining" << endl;
}
```

C-- BACI example: program 3

- Analyse the following program!

```
monitor monSemaphore {
  int semvalue;
  condition notbusy;

  void monP() {
    if (!semvalue)
      waitc(notbusy);
    else semvalue--;
  }

  void monV() {
    if (empty(notbusy))
      semvalue++;
    else signalc(notbusy);
  }

  init{ semvalue = 1; }
} // end of monSemaphore monitor

int n;

void inc(int i) {
  int t;
  monP();
  t = n; t = t + 1; n = t;
  monV();
}

void main() {
  cobegin { inc(1); inc(2); }
}
```