The SystemC Simulation Engine

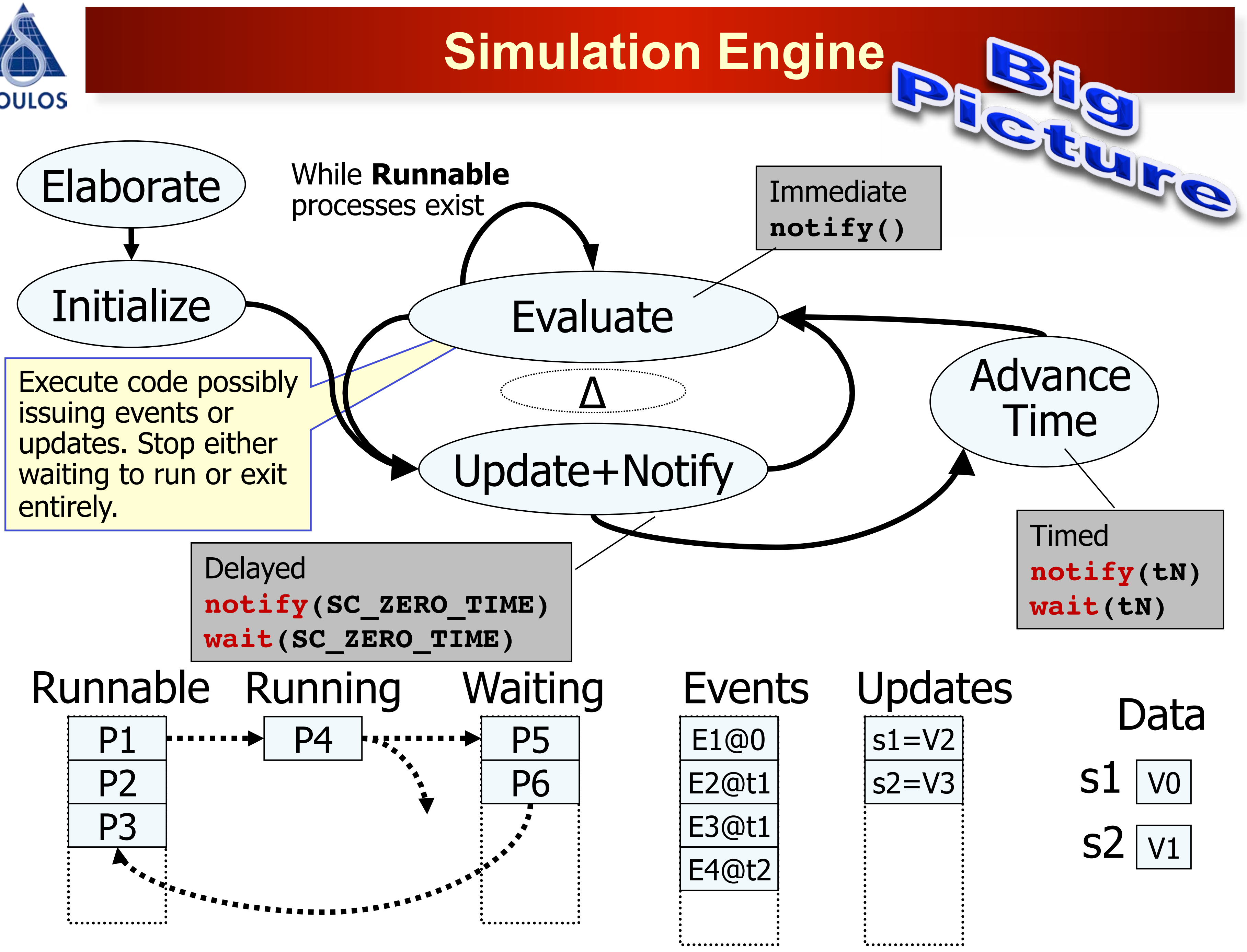
An Interactive Exploration of an Event Driven Simulator

David C. Black

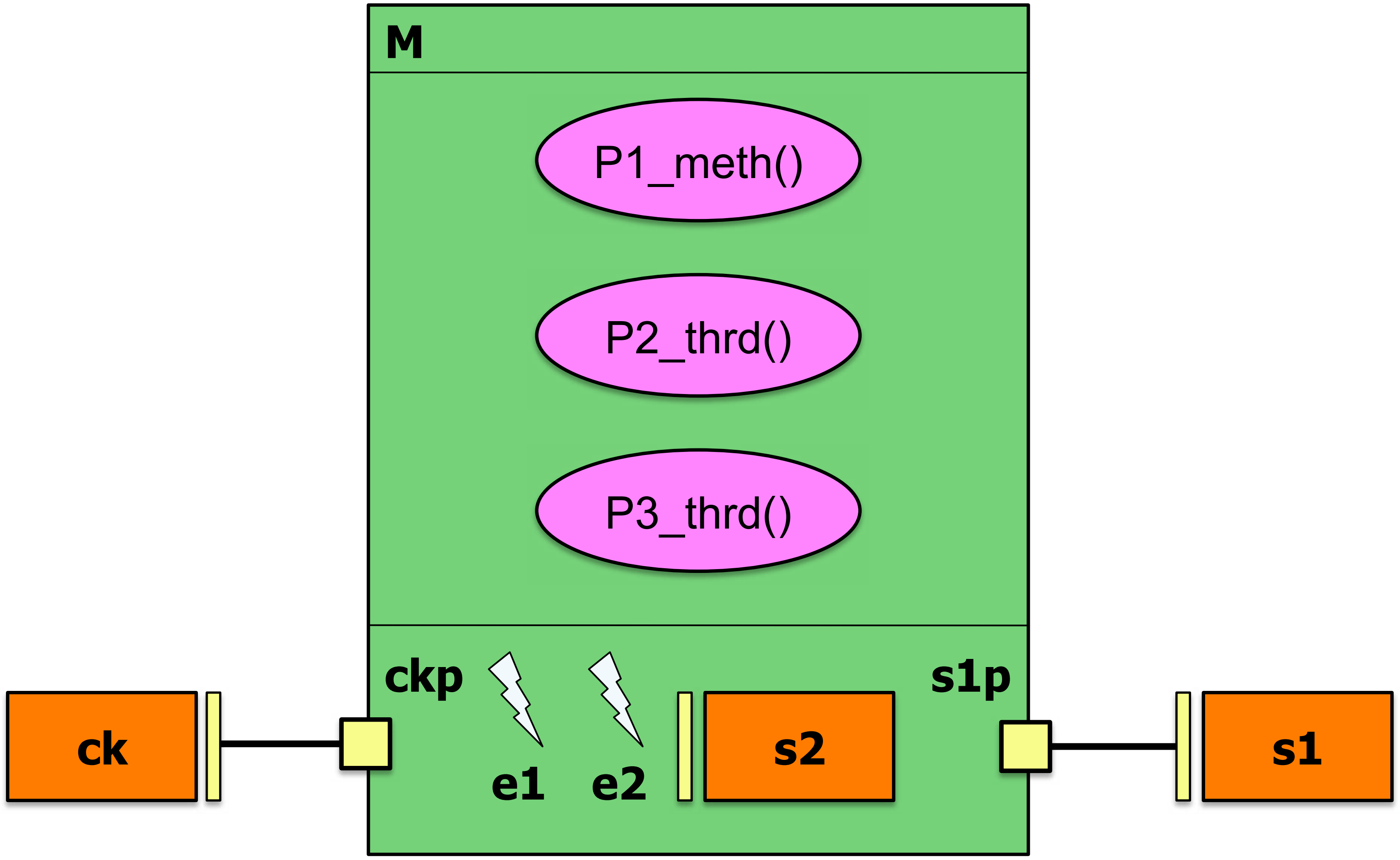
Agenda & Goals

- Overview of flow diagram and queues
- SystemC constructs used
- Example code
- Step-by-step walk-thru
  - Illustrate each step of simulation
  - Understand events & delta cycles
- References

Simulation Engine



Example Design



SystemC constructs used herein

- **SC_MODULE**, **SC_CTOR** - used to create modules
- **SC_THREAD**[1], **SC_METHOD**[1] - types of processes
- **sensitive**, **dont_initialize** - attributes of processes
- **sc_event**, **wait**[2], **notify**[3] - synchronization mechanisms
- **sc_signal**[4], **read**, **write** - primitive channel
- **sc_clock** - **sc_signal**<**bool**> with a generating process
- **sc_in** - **sc_port**<> specialization of type **sc_signal_in_if**<>
- **sc_out** - **sc_port**<> specialization of type **sc_signal_out_if**<>

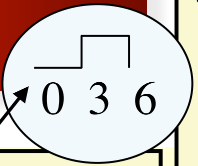
[1] Verilog **initial** or **always** block; VHDL **process** block

[2] Verilog **@**, **#**, or **wait** statement; VHDL **wait** statement

[3] Verilog **->** statement, except SystemC is more flexible

[4] Verilog **wire** or **var** with **<=** type; VHDL **signal** type

Example code



```
sc_clock ck("ck",6,0.5,3);
SC_MODULE(M) {
  sc_in<bool> ckp;//in  port
  sc_out<int> s1p;//out port
  sc_signal<int> s2;
  sc_event e1, e2;
  void P1_meth();
  void P2_thrd();
  void P3_thrd();
  SC_CTOR(M):temp(9){
    SC_THREAD(P3_thrd);
   {SC_THREAD(P2_thrd);
      sensitive<<ckp.pos();
   {SC_METHOD(P1_meth);
      sensitive<<s2;
      dont_initialize();
  }//end SC_CTOR
 private:
  int temp;
};
```

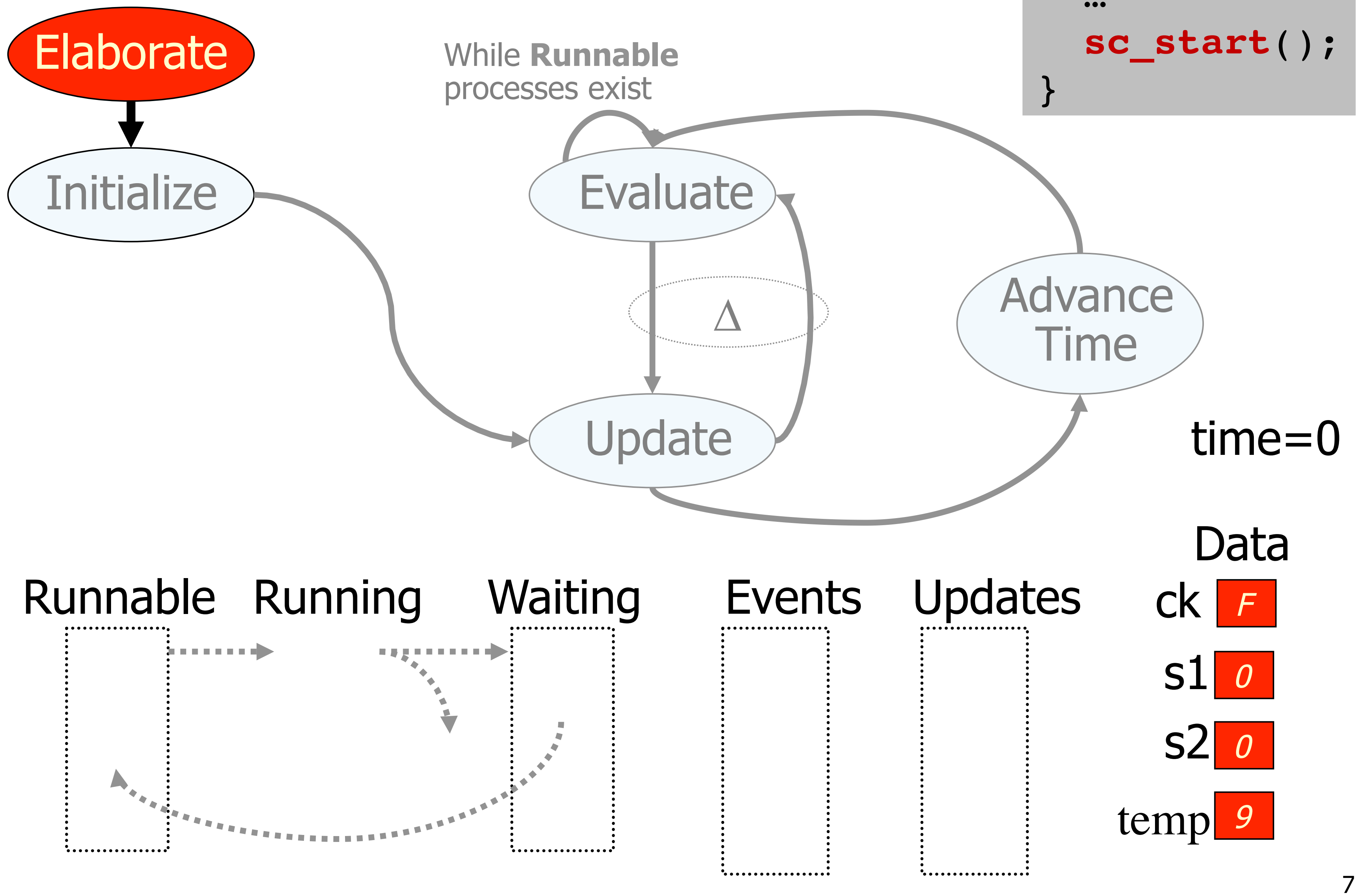
Executed during elaboration

```
void M::P1_meth() {
  temp = s2.read();
  s1p->write(temp+1);
  e2.notify(2,SC_NS);//timed
}
void M::P2_thrd() {
A:s2.write(5);
  e1.notify();//immediate
  wait();
B:for (int i=7;i<9;i++){
    s2.write(i);
    wait(1,SC_NS);        //delayed
C:  e1.notify(SC_ZERO_TIME);
    wait();//static sensitive
  }//endfor
}
void M::P3_thrd() {
D:while(true) {
    wait(e1|e2);
E:  cout << "time "
    <<sc_time_stamp()<<endl;
  }//endwhile
}
```

Simulation

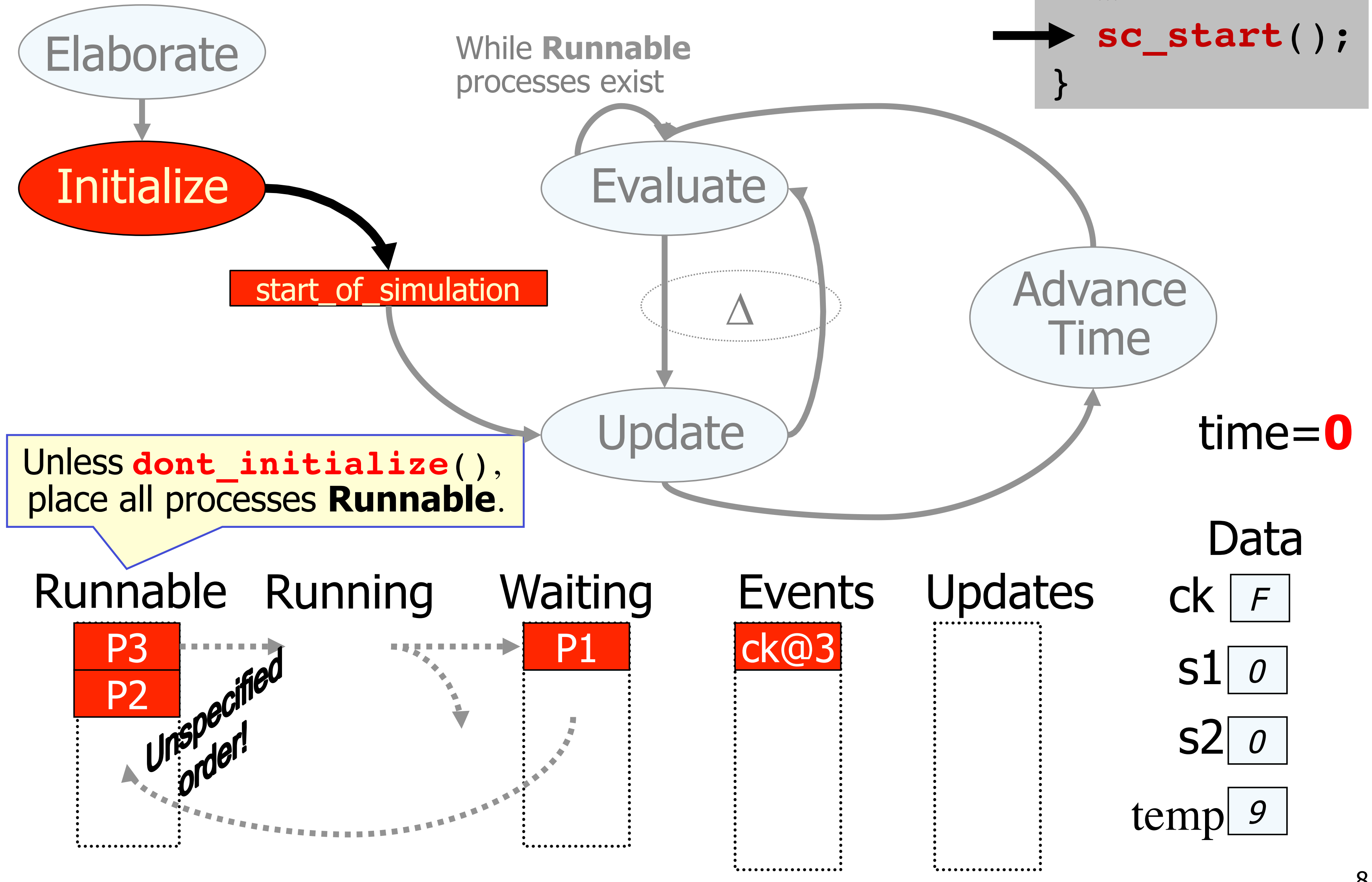
T-1 Elaboration

```
sc_main(){
  M m("m");
  …
  sc_start();
}
```



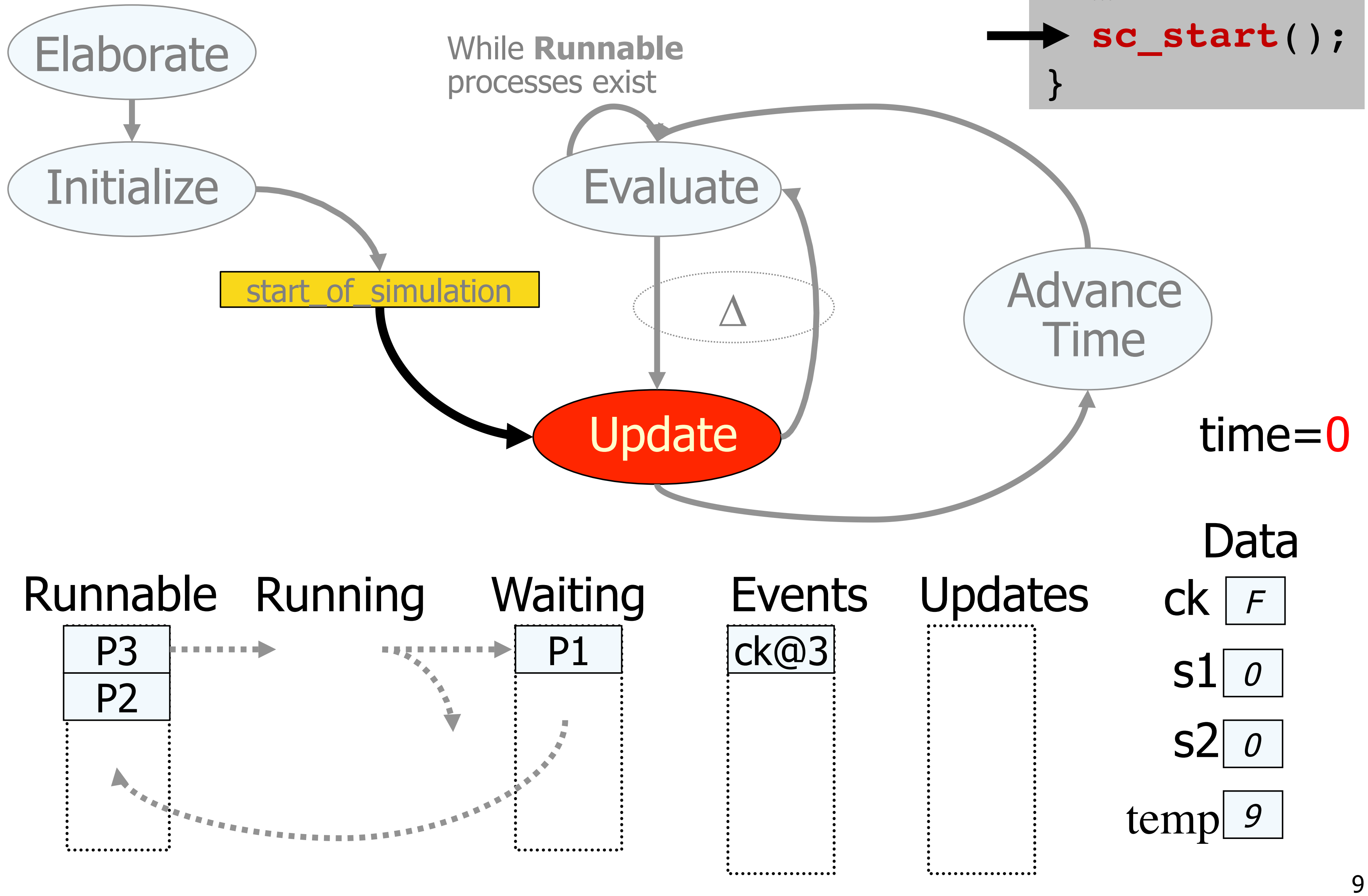
T-0a Initialization

```
sc_main(){
   M m("m");
   …
   sc_start();
}
```



T-0b Initialization

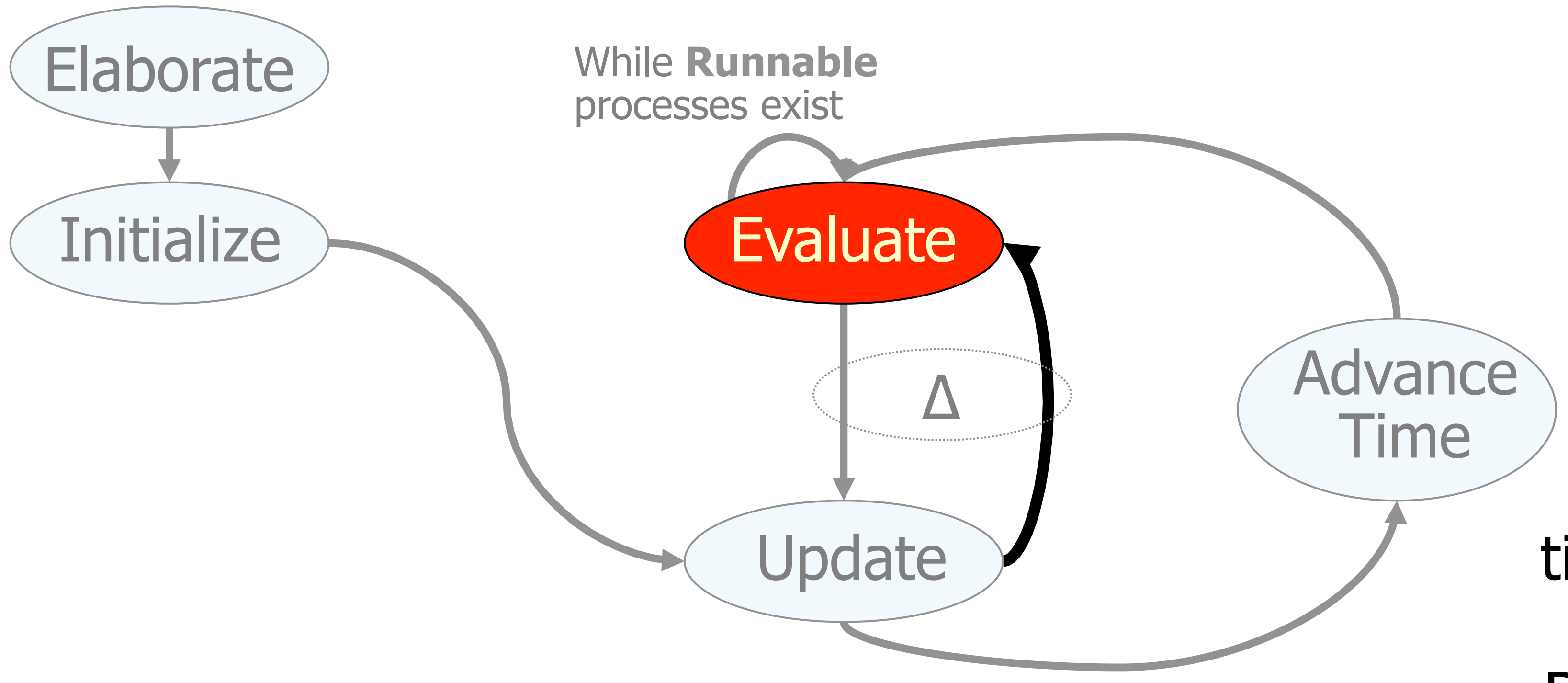
```
sc_main(){
   M m("m");
   …
  sc_start();
}
```



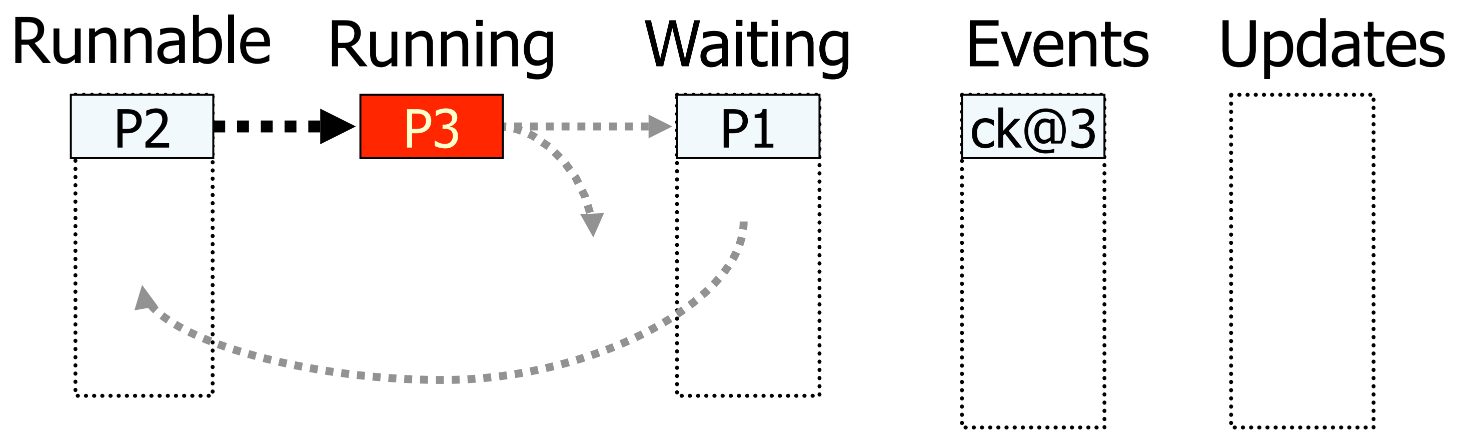
T_1a Evaluate

P3

```
D:while(true) {
   wait(e1|e2);
```



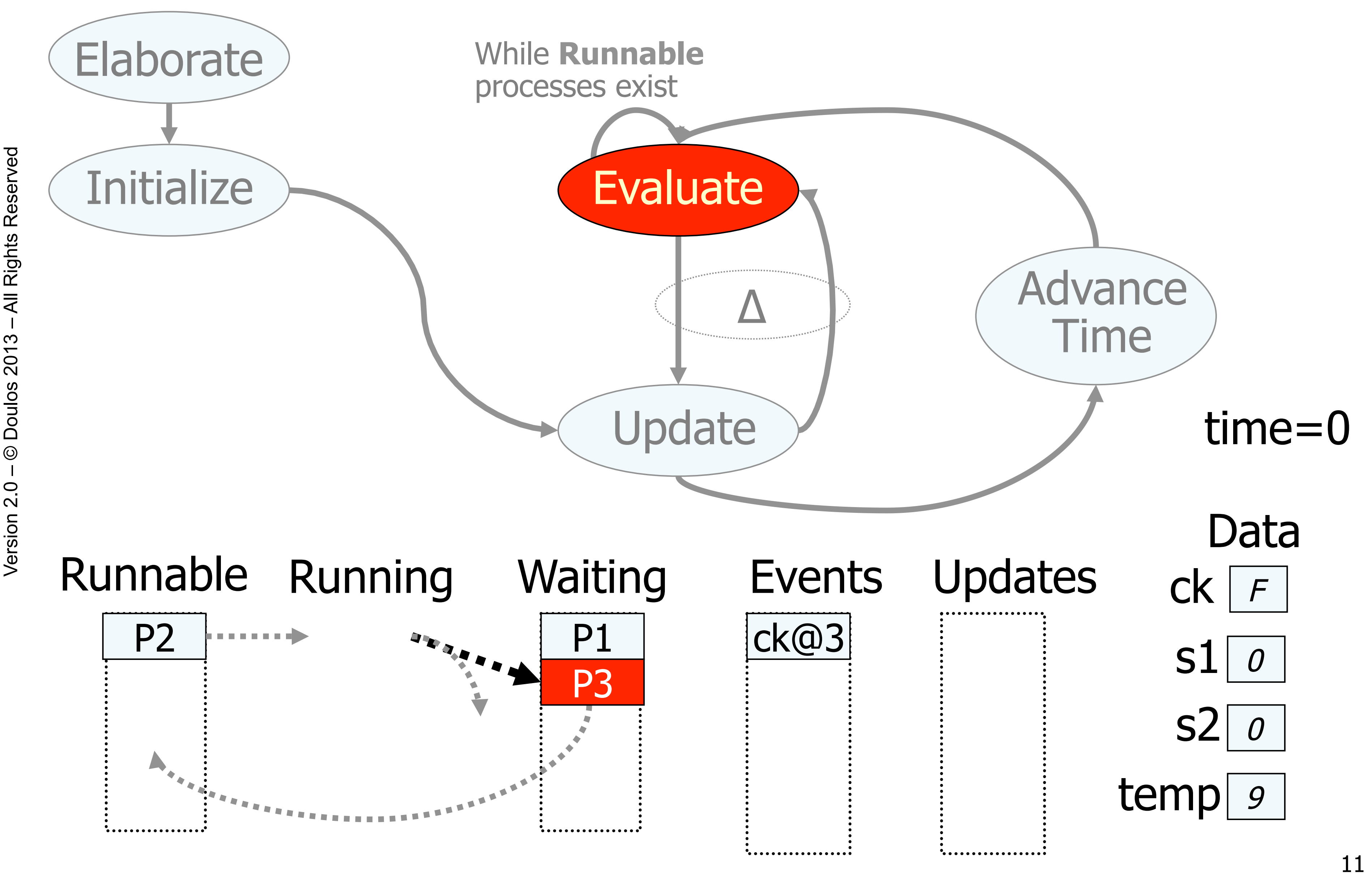
time=0



Data	
ck	*F*
s1	*0*
s2	*0*
temp	*9*

T_1b Evaluate

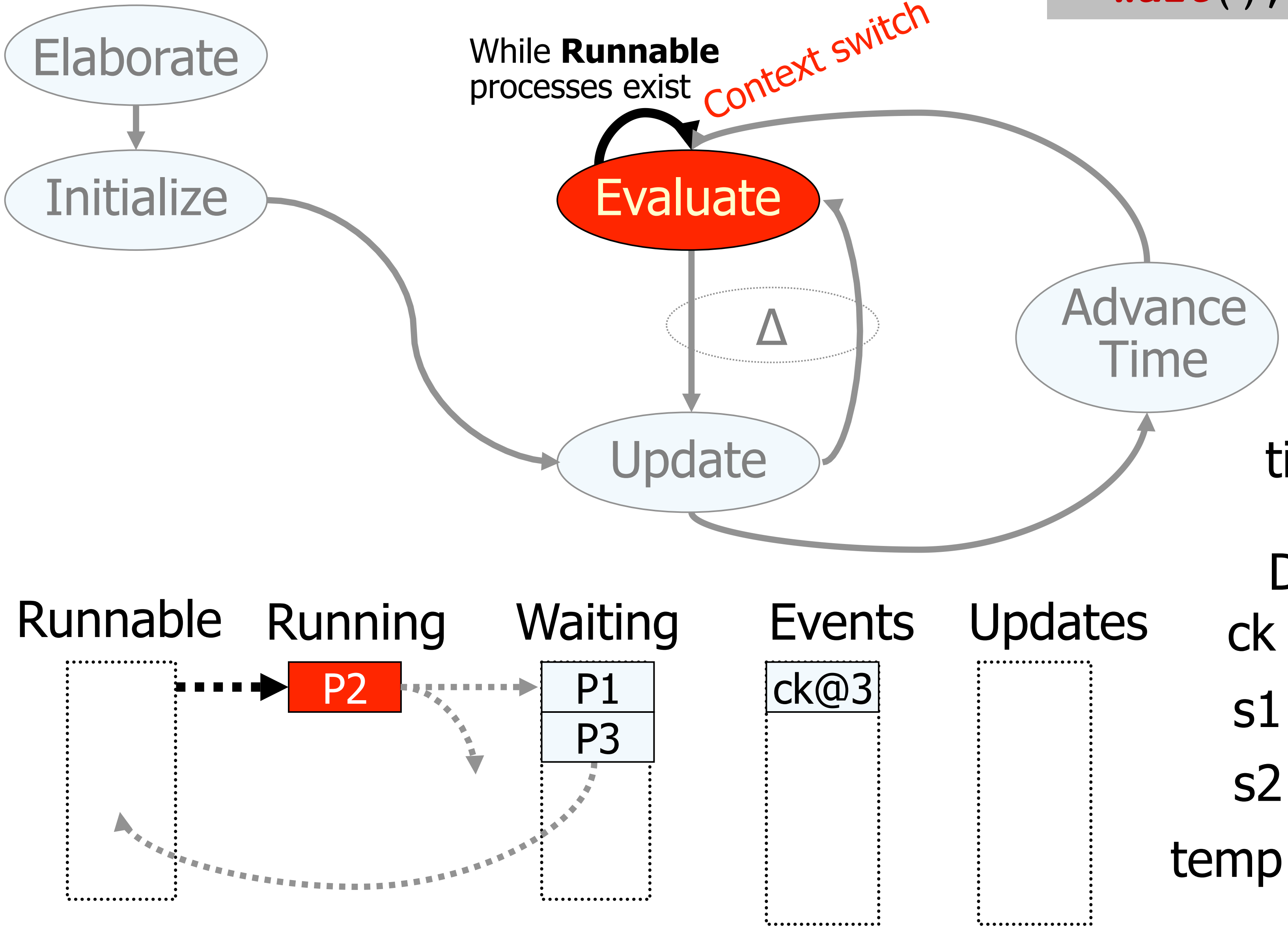
```
D:while(true) {
  wait(e1|e2);
```



T_2a Evaluate

P2

```
A:s2.write(5);
  e1.notify();
  wait();
```



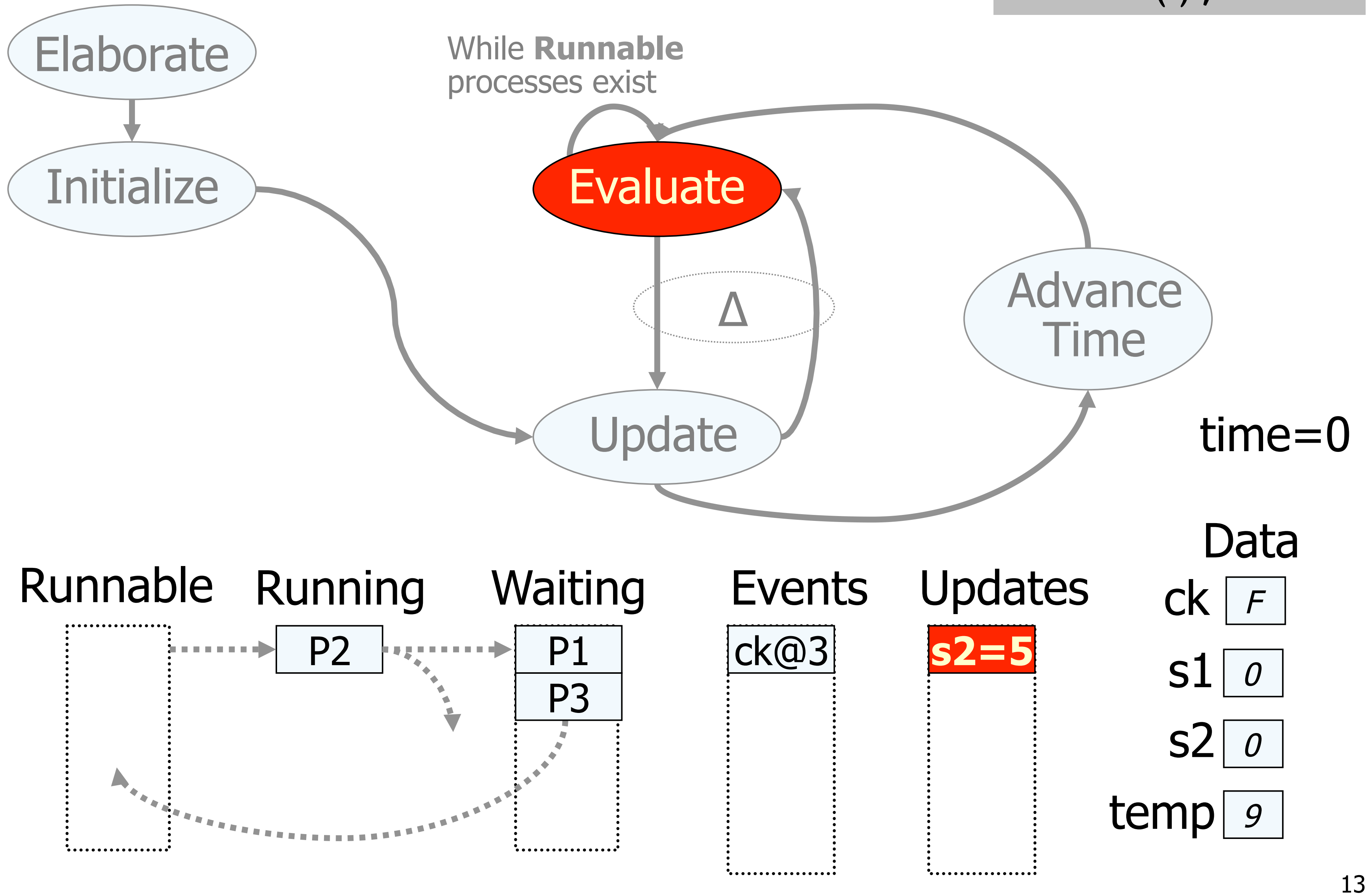
time=0

Data	
ck	*F*
s1	*0*
s2	*0*
temp	*9*

T_2b Evaluate

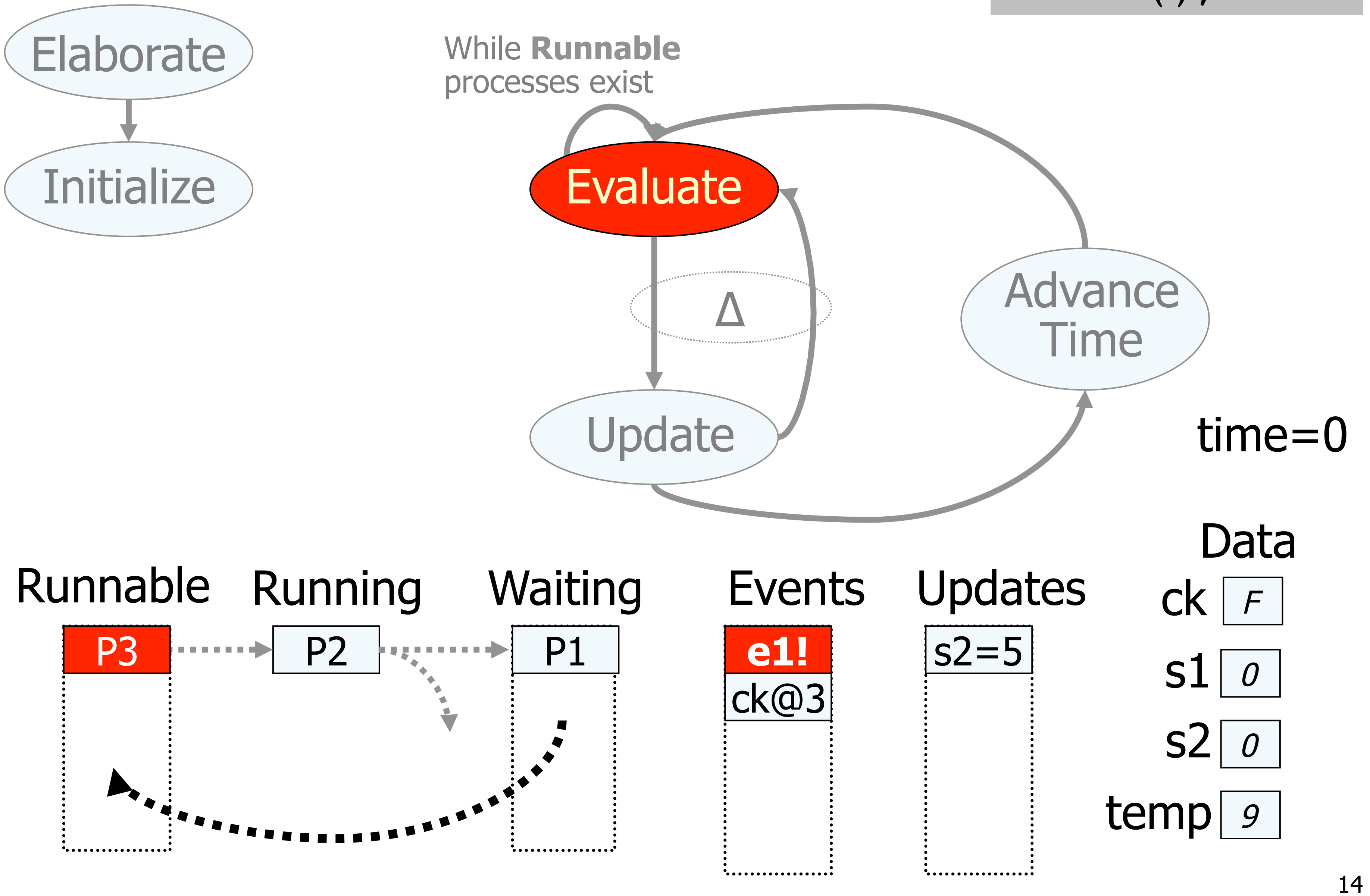
P2 →

```
A:s2.write(5);
  e1.notify();
  wait();
```



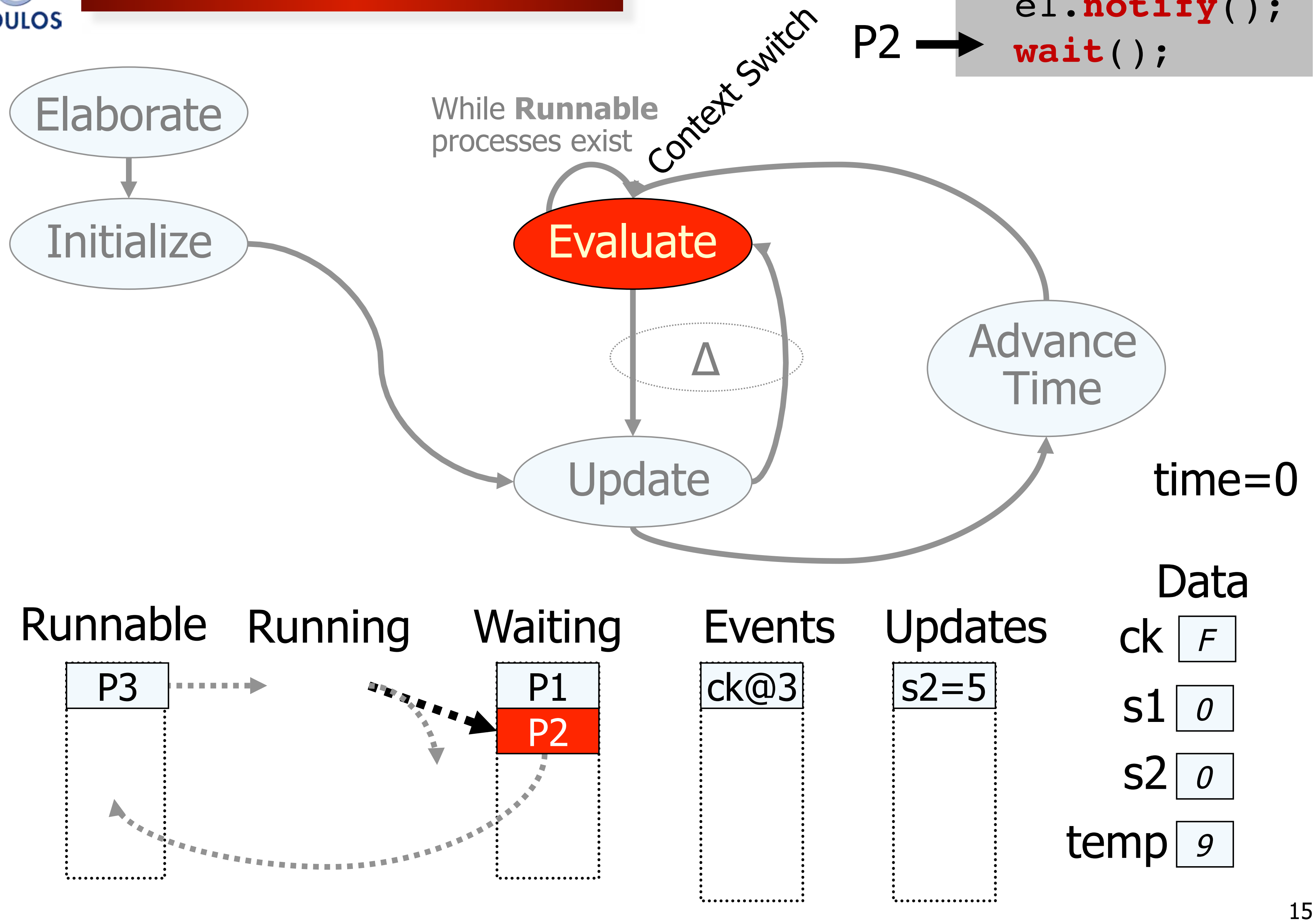
T_2c Evaluate

```
A:s2.write(5);
  e1.notify();
  wait();
```



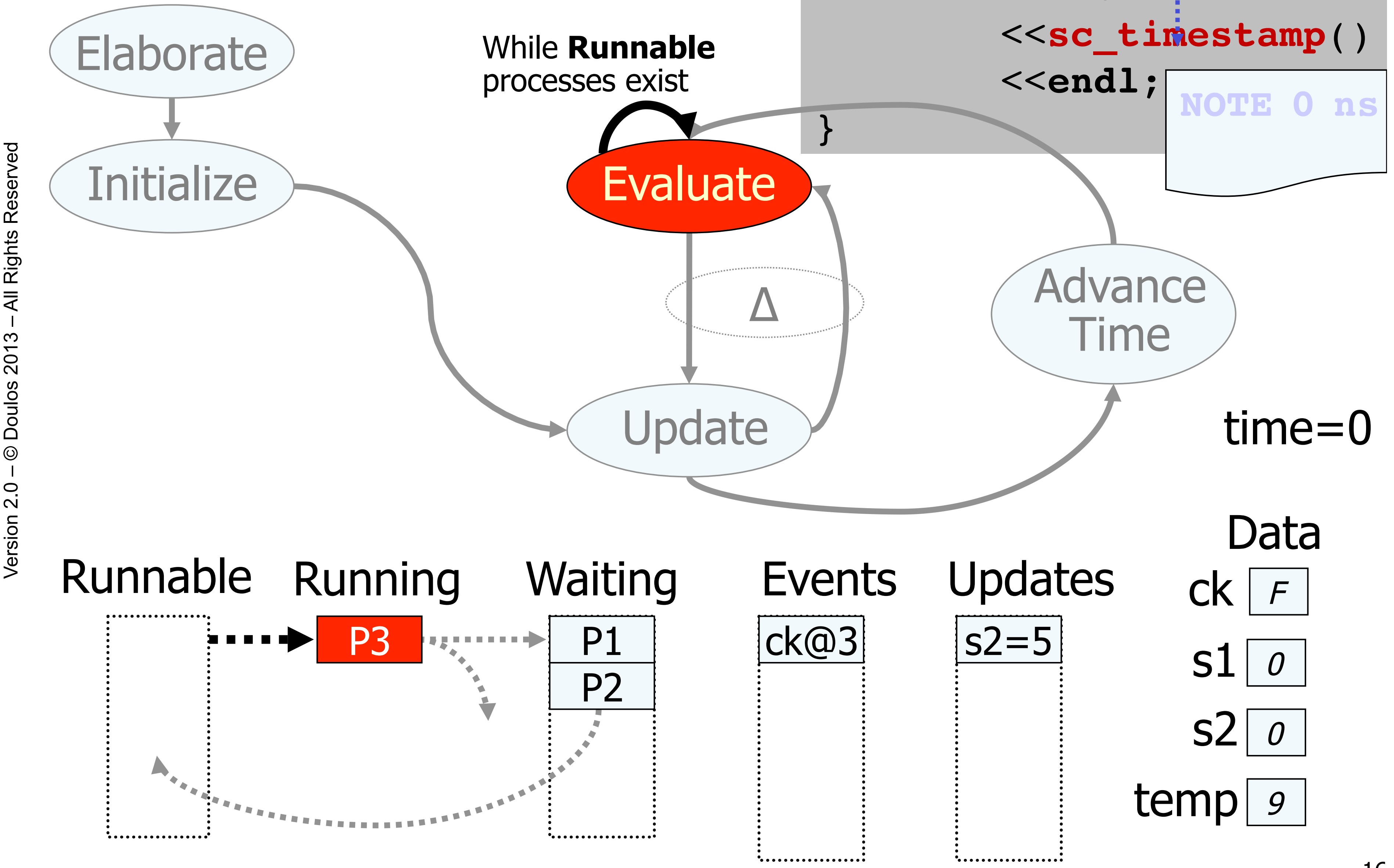
T_2d Evaluate

```
A:s2.write(5);
  e1.notify();
  wait();
```



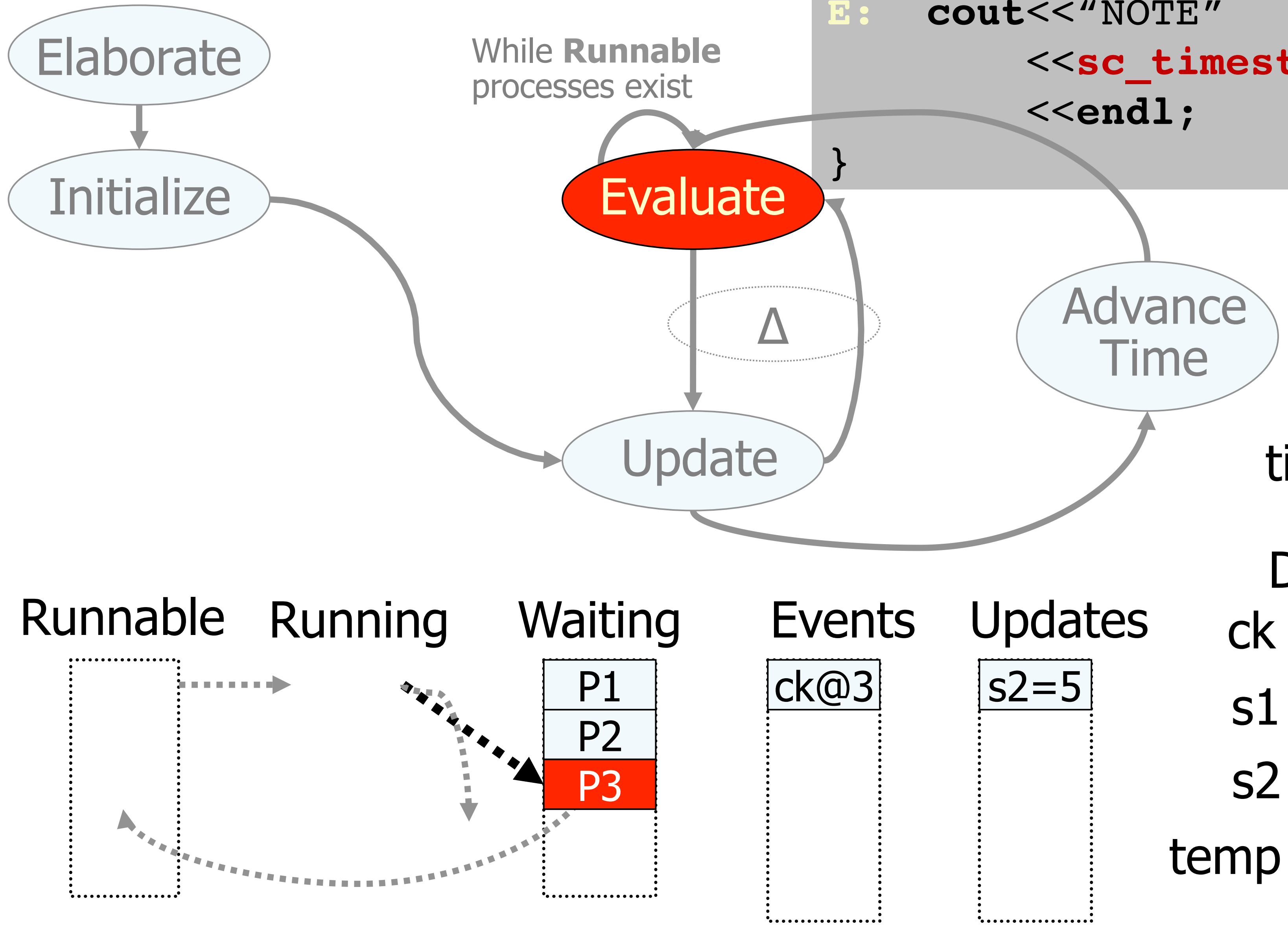
T_3a Evaluate

```
D:while(true){
     wait(e1|e2);
E:   cout<<"NOTE"
         <<sc_timestamp()
         <<endl;
}
```

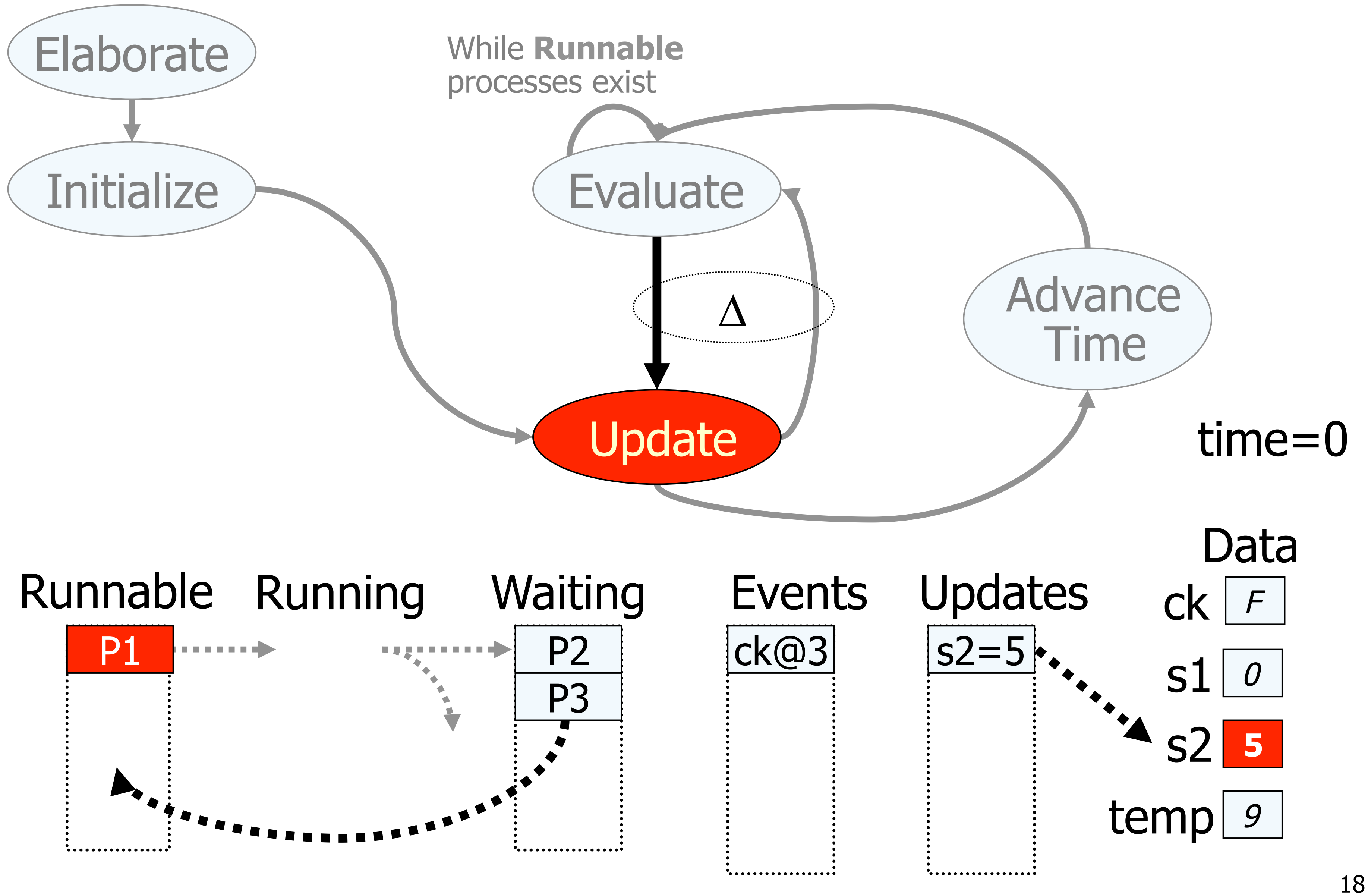


T_3b Evaluate

```
D:while(true){
P3 ──▶ wait(e1|e2);
E:   cout<<"NOTE"
         <<sc_timestamp()
         <<endl;
}
```

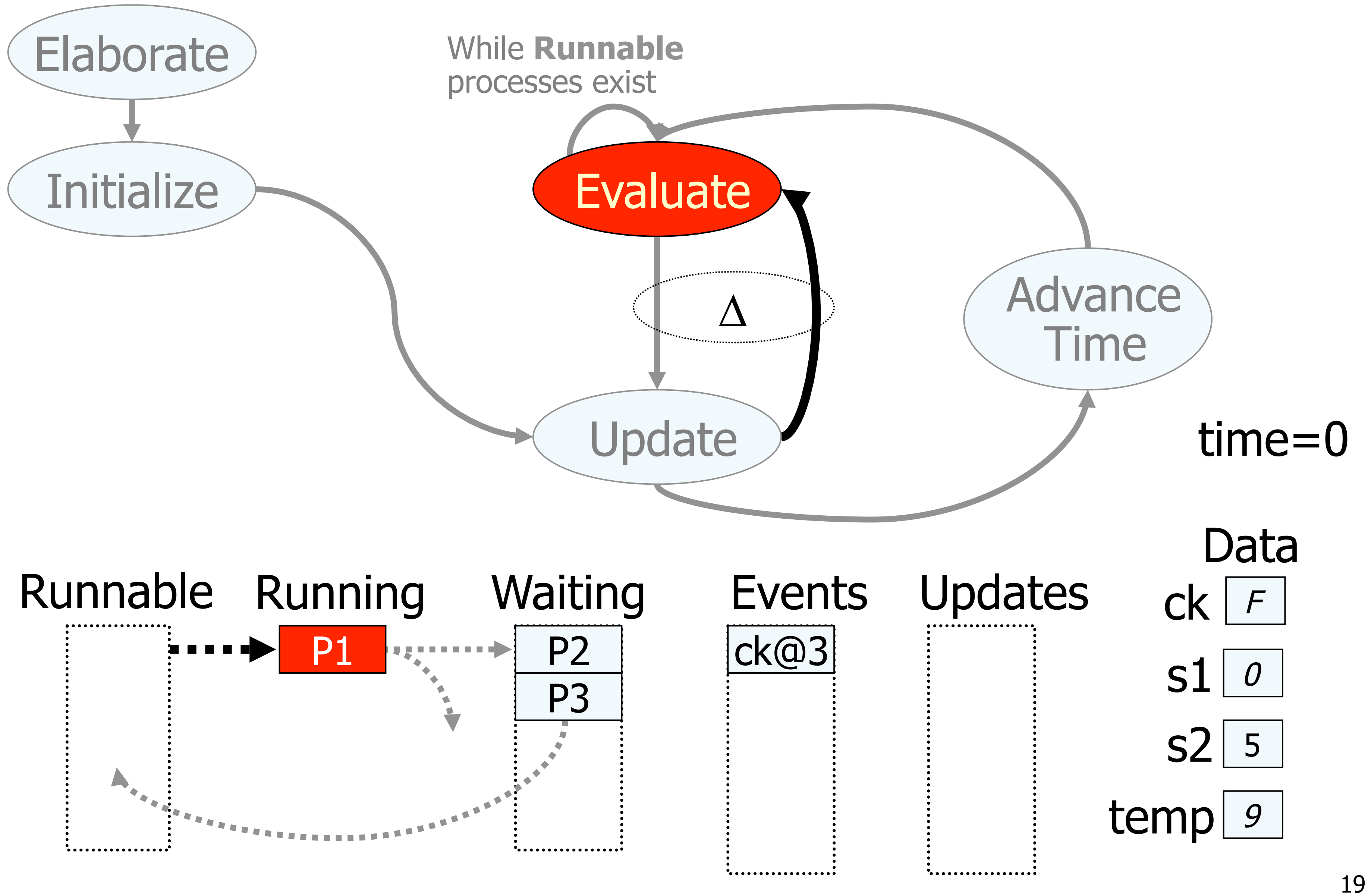


T_4 Update


Elaborate
Initialize
While **Runnable** processes exist
Evaluate
Δ
Update
Advance Time
time=0
Runnable
Running
Waiting
Events
Updates
Data
P1
P2
P3
ck@3
s2=5
ck F
s1 0
s2 5
temp 9

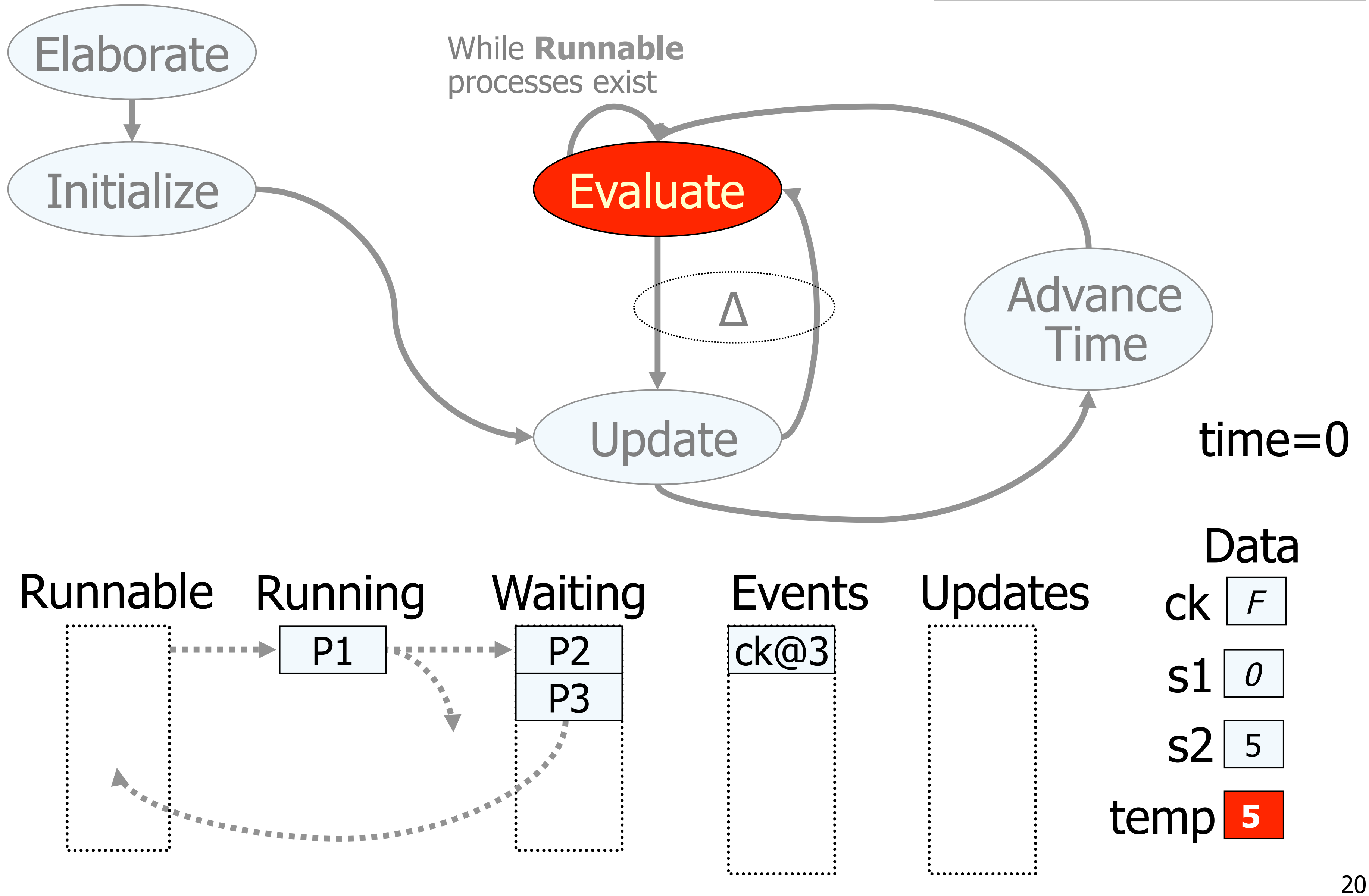
T_5a Evaluate

P1 →
```
temp = s2.read();
s1->write(temp+1);
e2.notify(2,SC_NS);
```



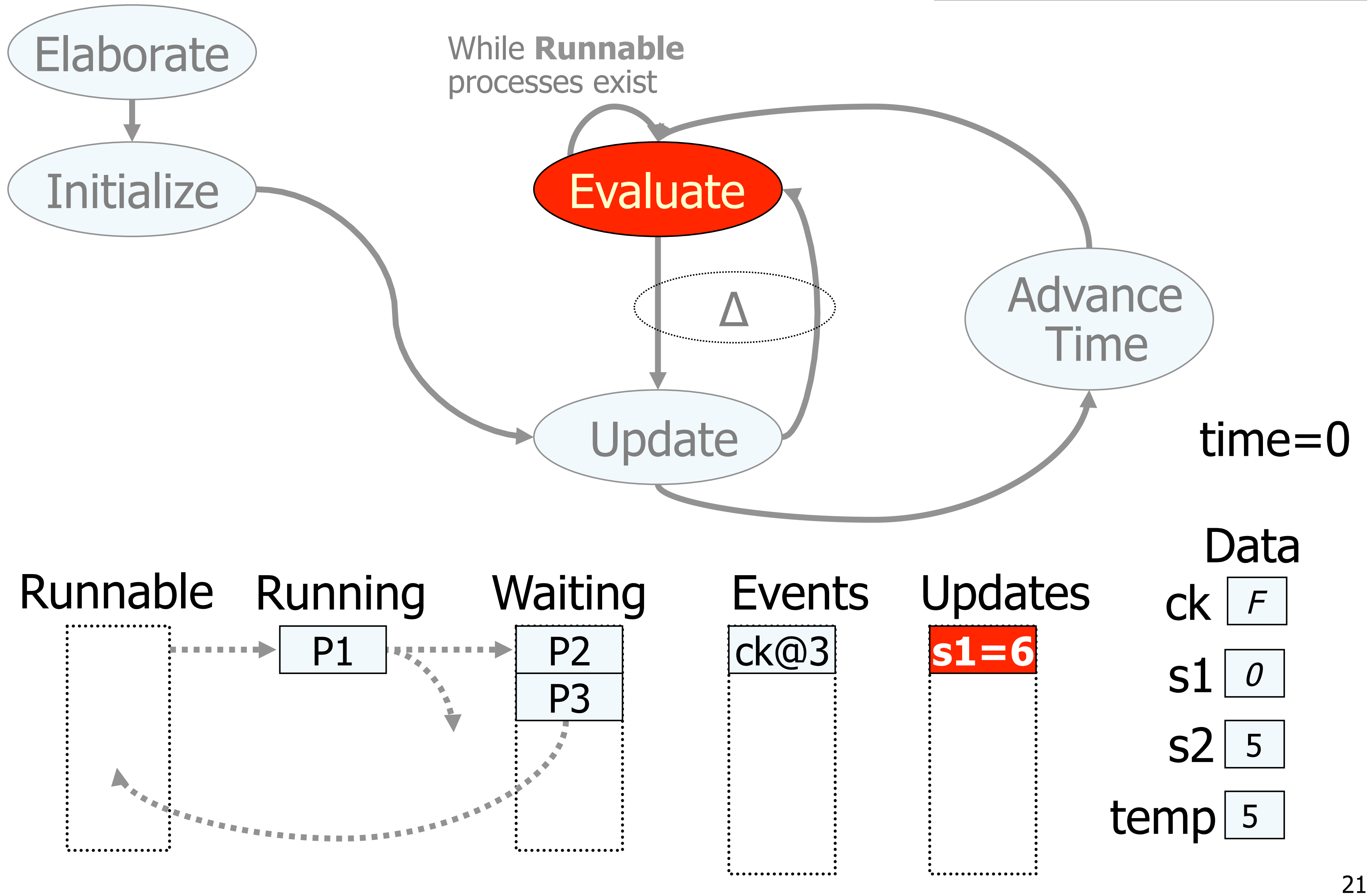
T_5b Evaluate

P1 →
```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
```



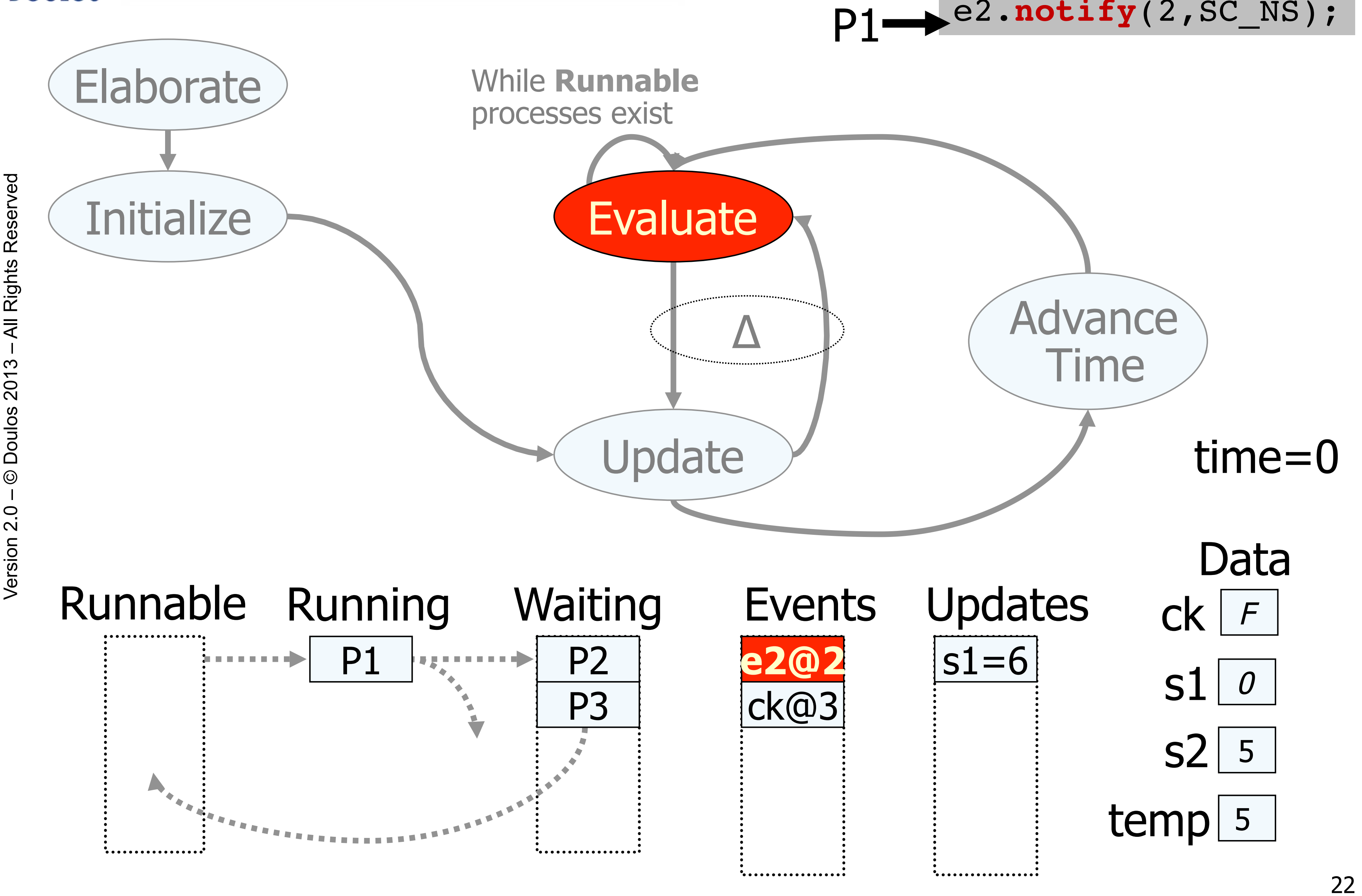
T_5c Evaluate

```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
```



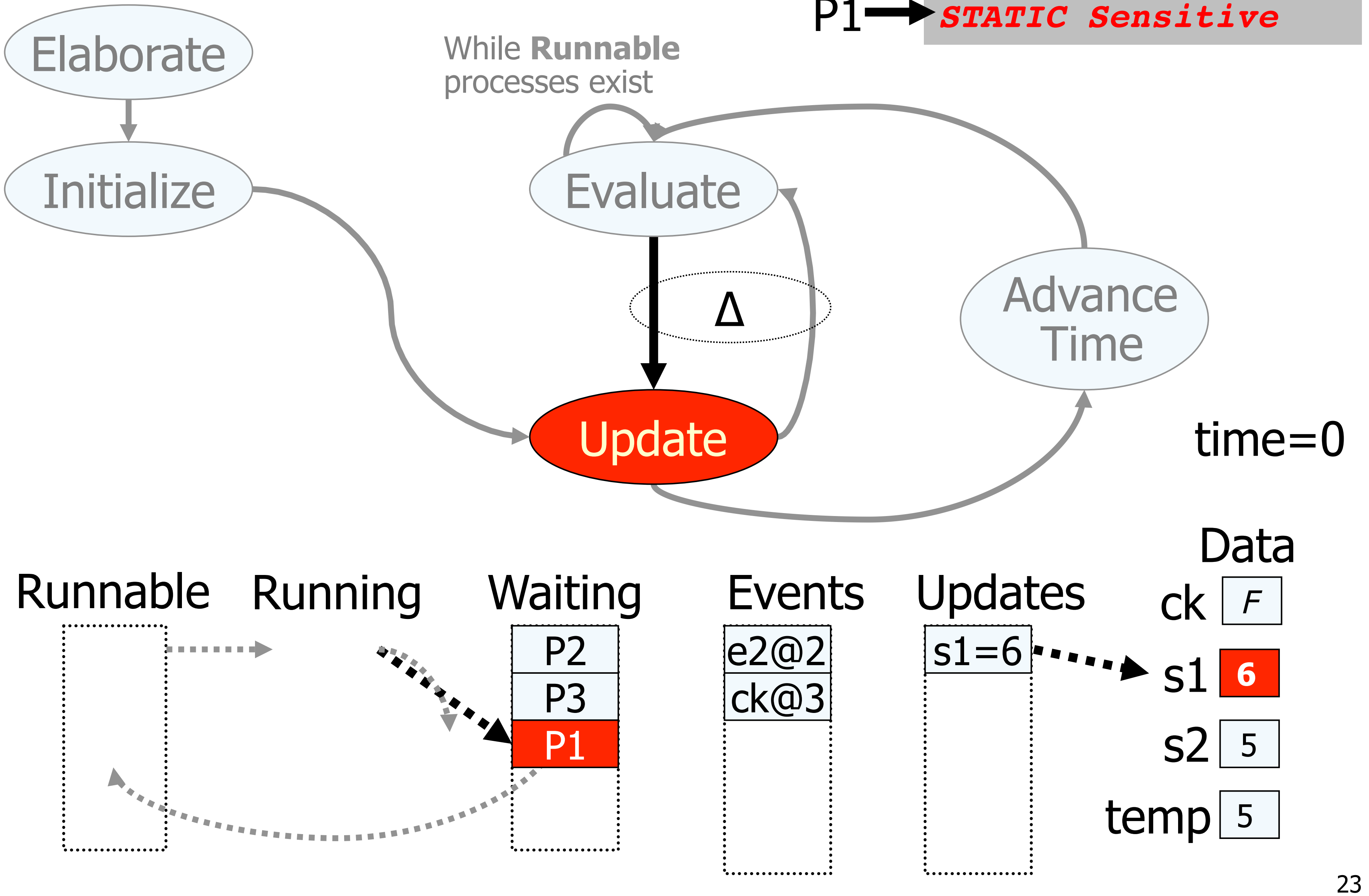
T_5d Evaluate

```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
```

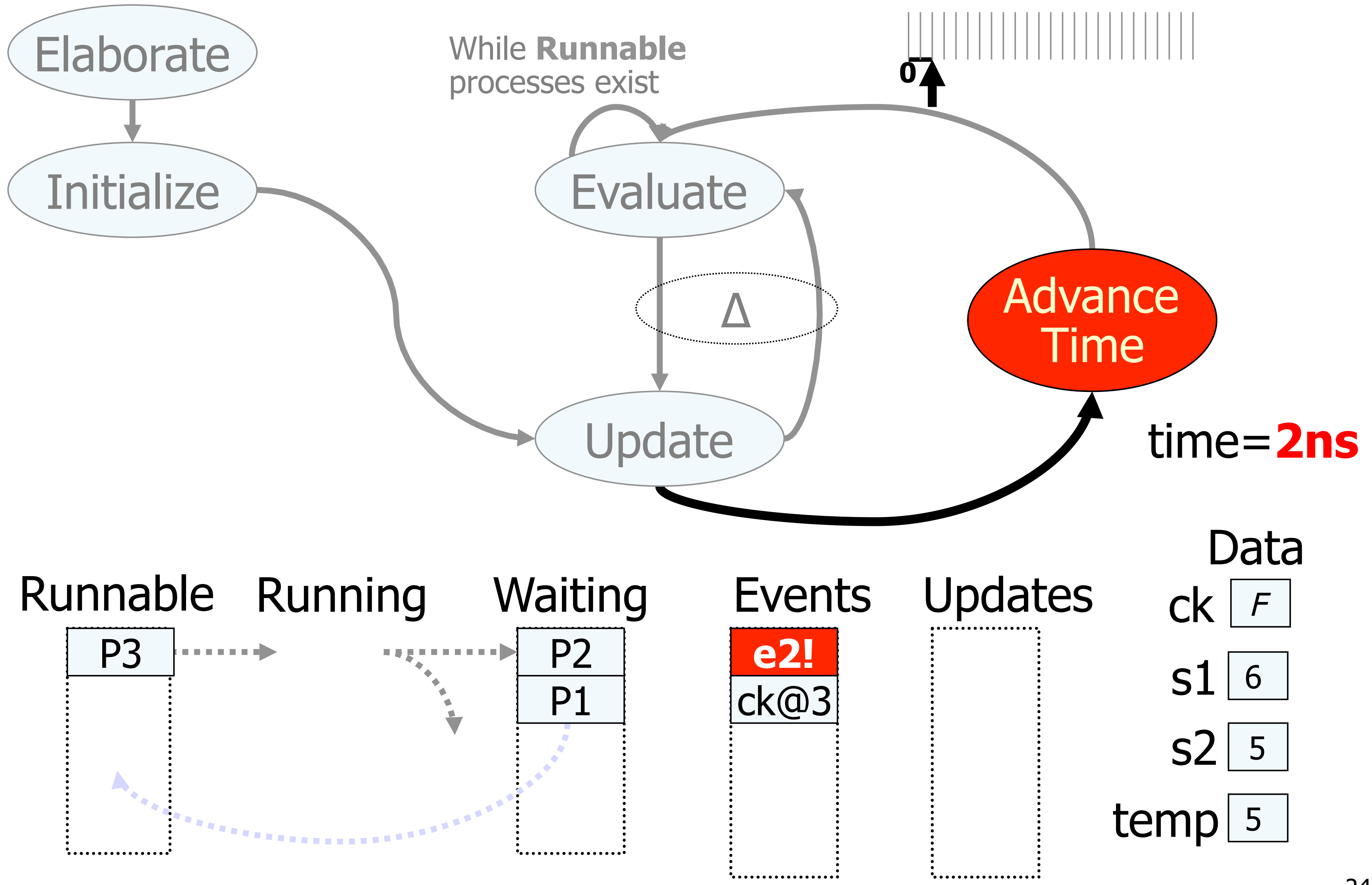


T_6 Update

```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
STATIC Sensitive
```



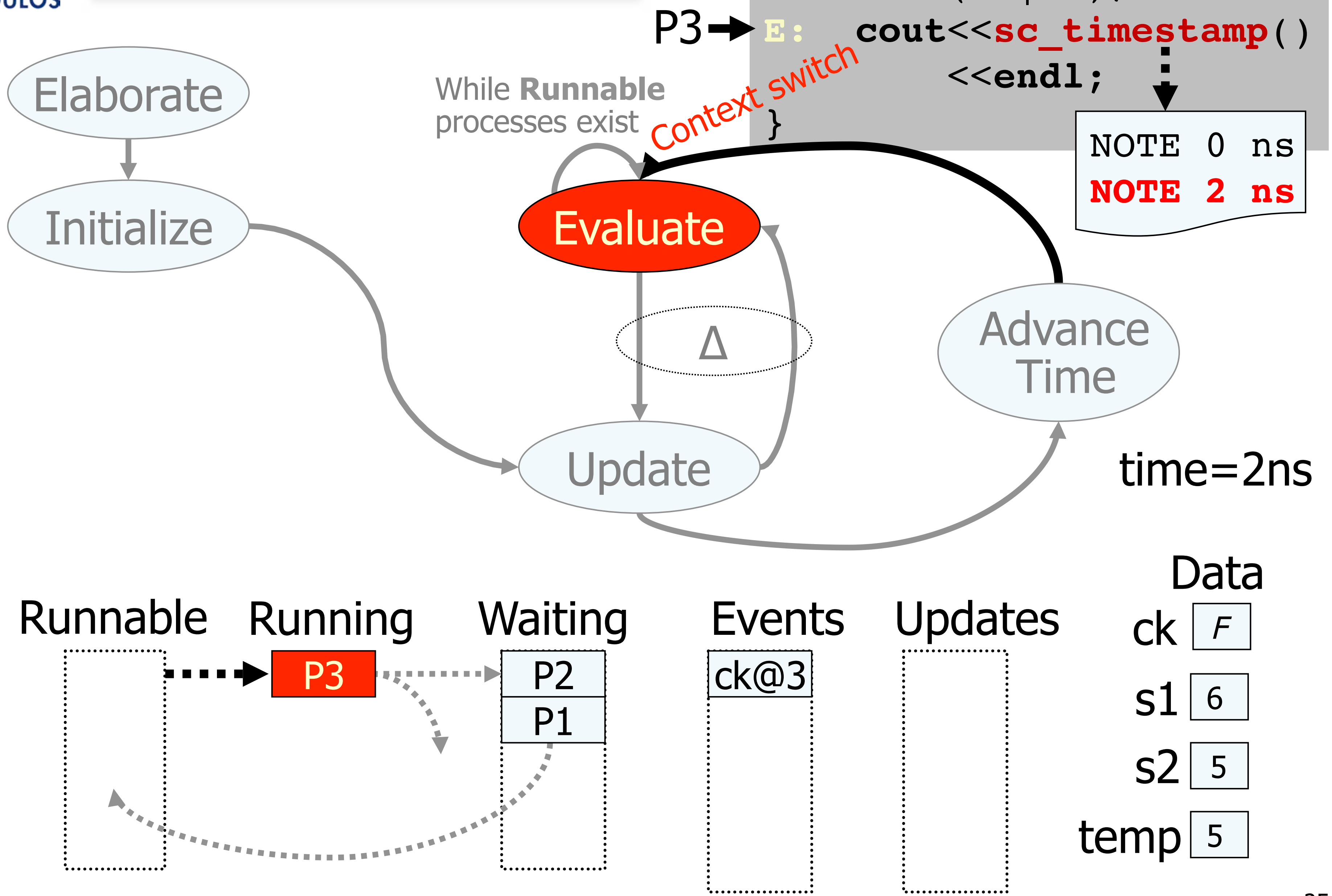
T_7a Advance Time



T_7b Evaluate

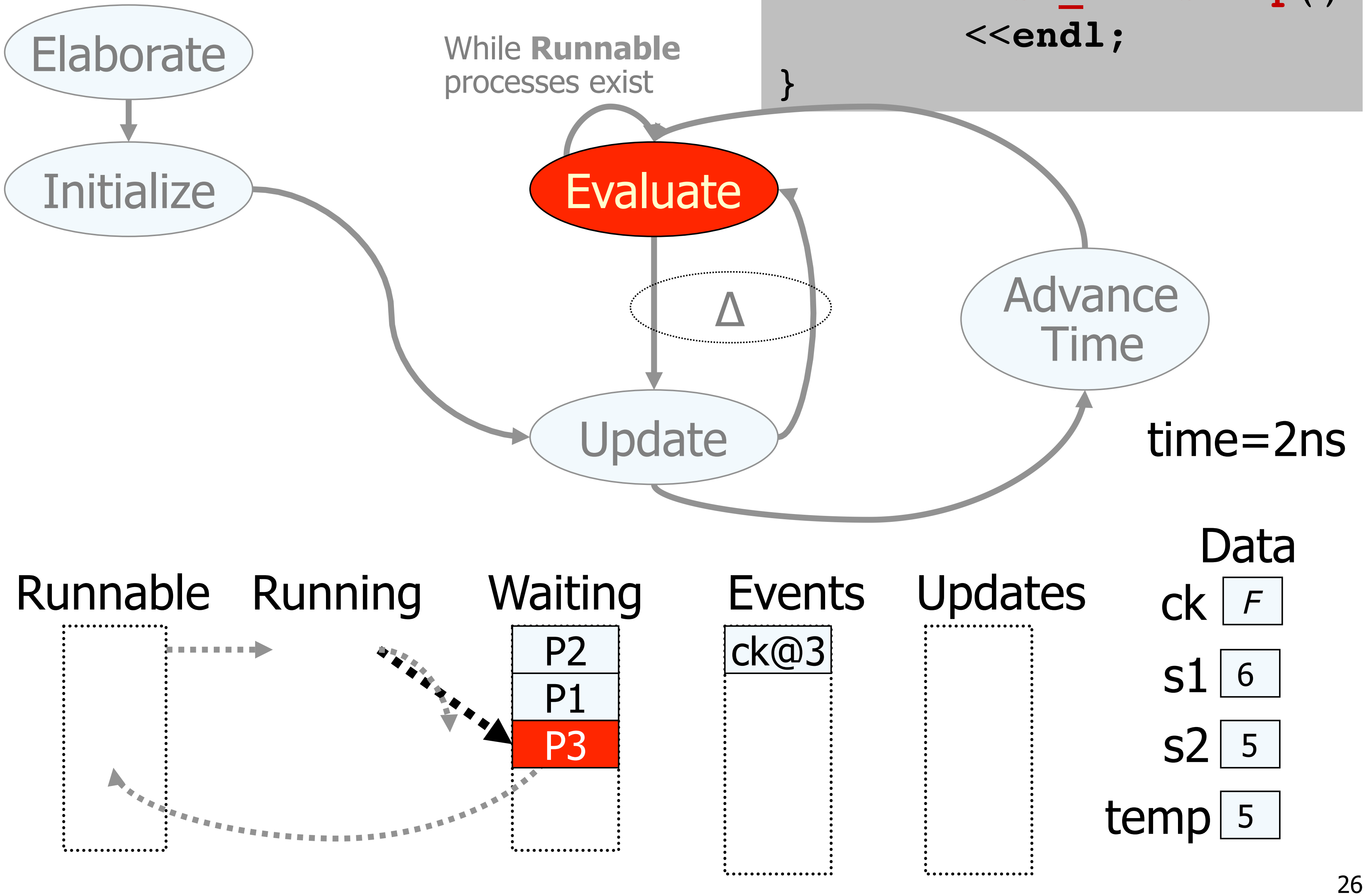
```
D:while(true){
    wait(e1|e2);
E:  cout<<sc_timestamp()
        <<endl;
  }
```

```
NOTE 0 ns
NOTE 2 ns
```



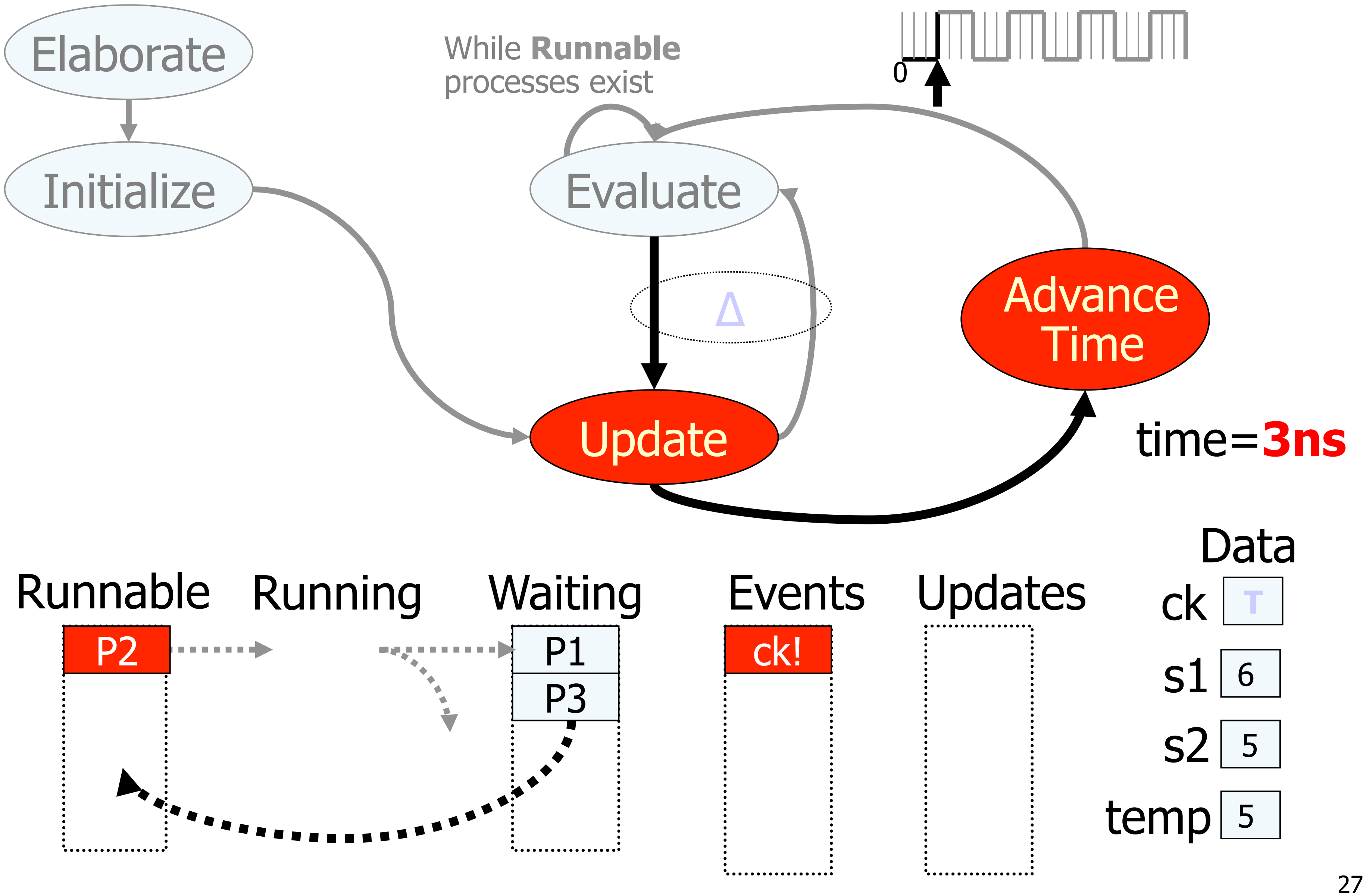
T_7c Evaluate

```
D:while(true){
P3 →  wait(e1|e2);
E:    cout<<sc_timestamp()
          <<endl;
}
```



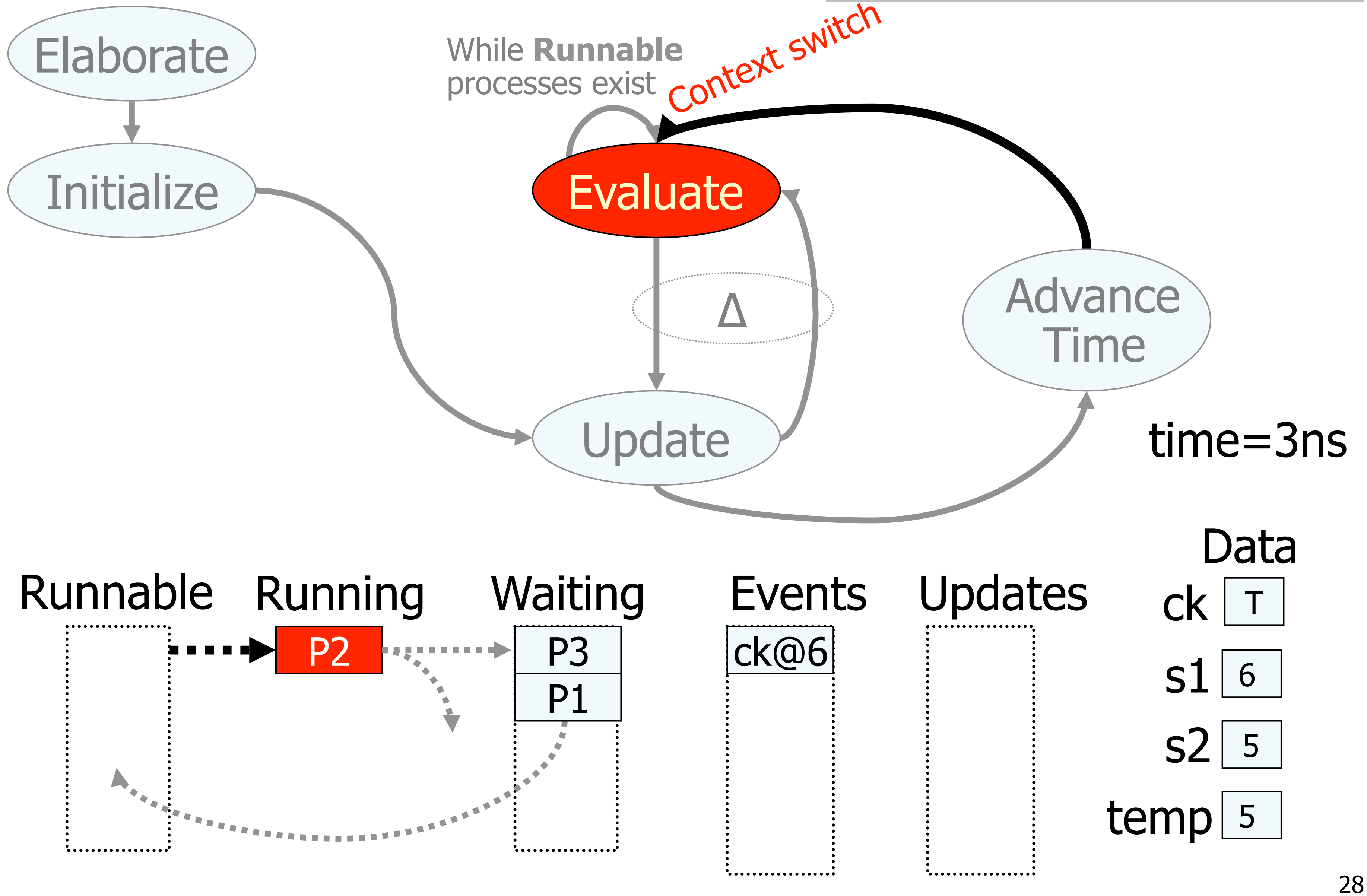
T_8/9 Advance Time

```
sc_clock ck("ck",6,0.5,3);
```



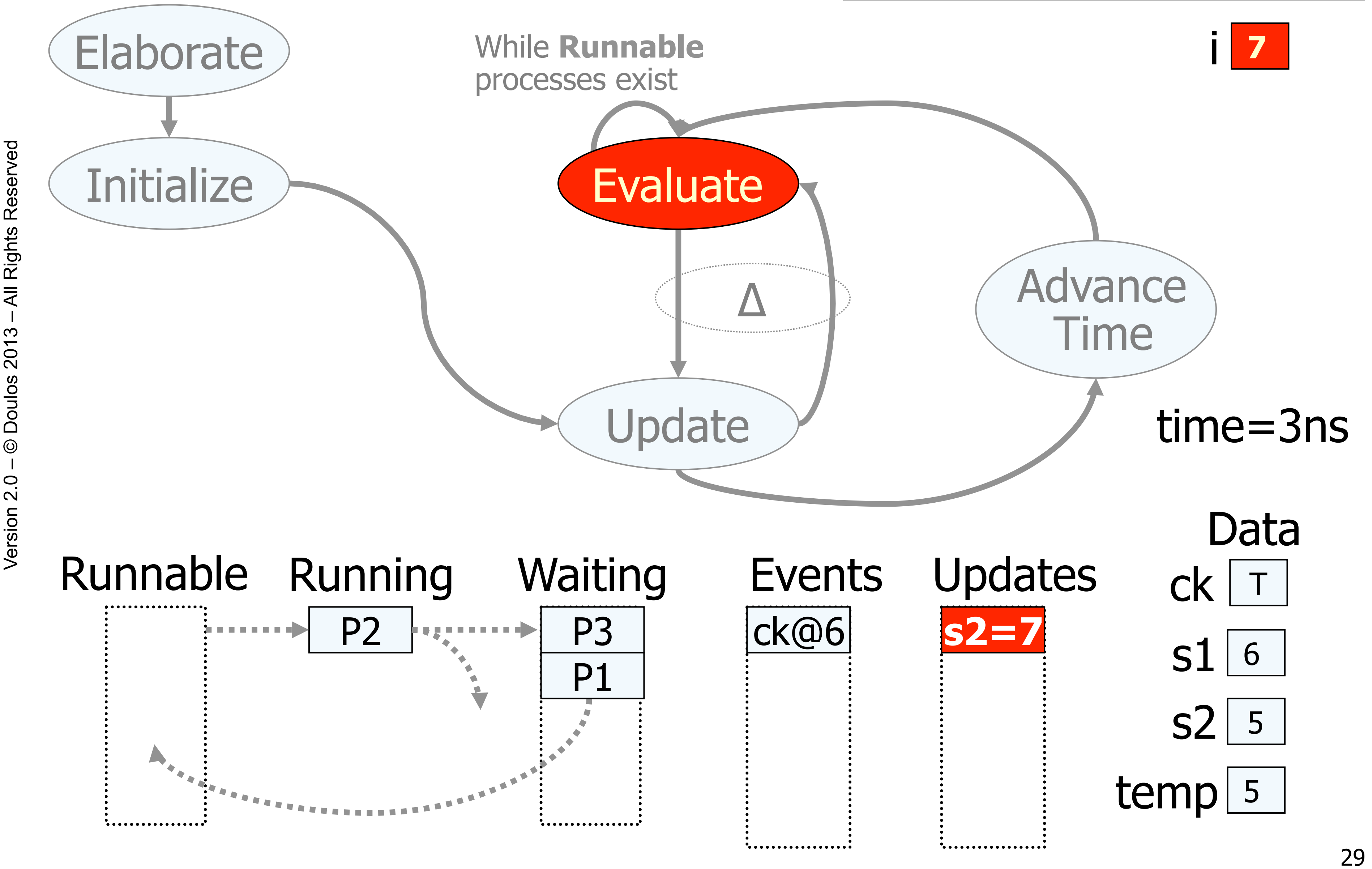
T10a Evaluate

P2 →
```
B:for (int i=7,i<9;i++){
     s2.write(i);
     wait(1,SC_NS);
```



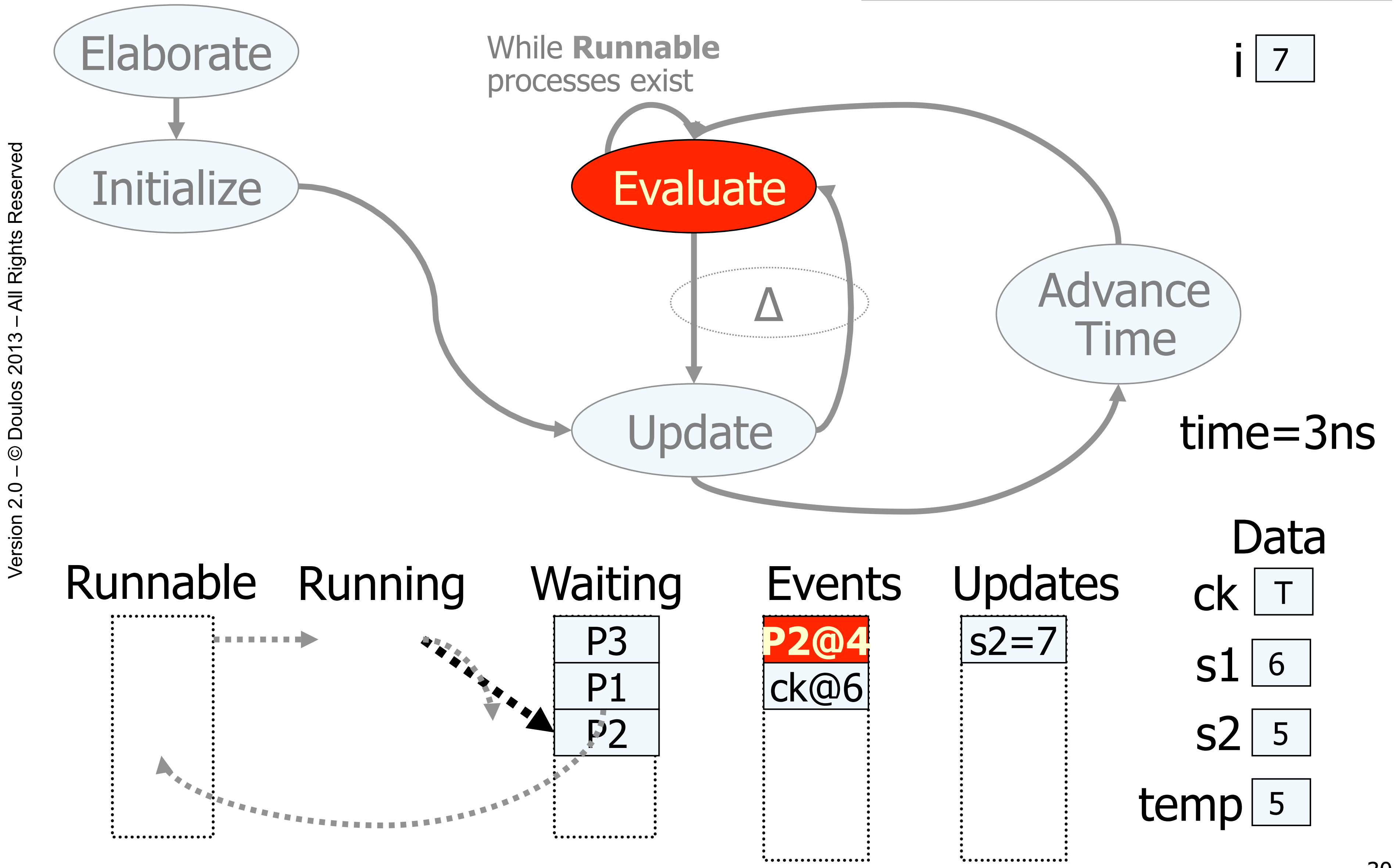
T10b Evaluate

```
B:for (int i=7,i<9;i++){
    s2.write(i);
    wait(1,SC_NS);
```

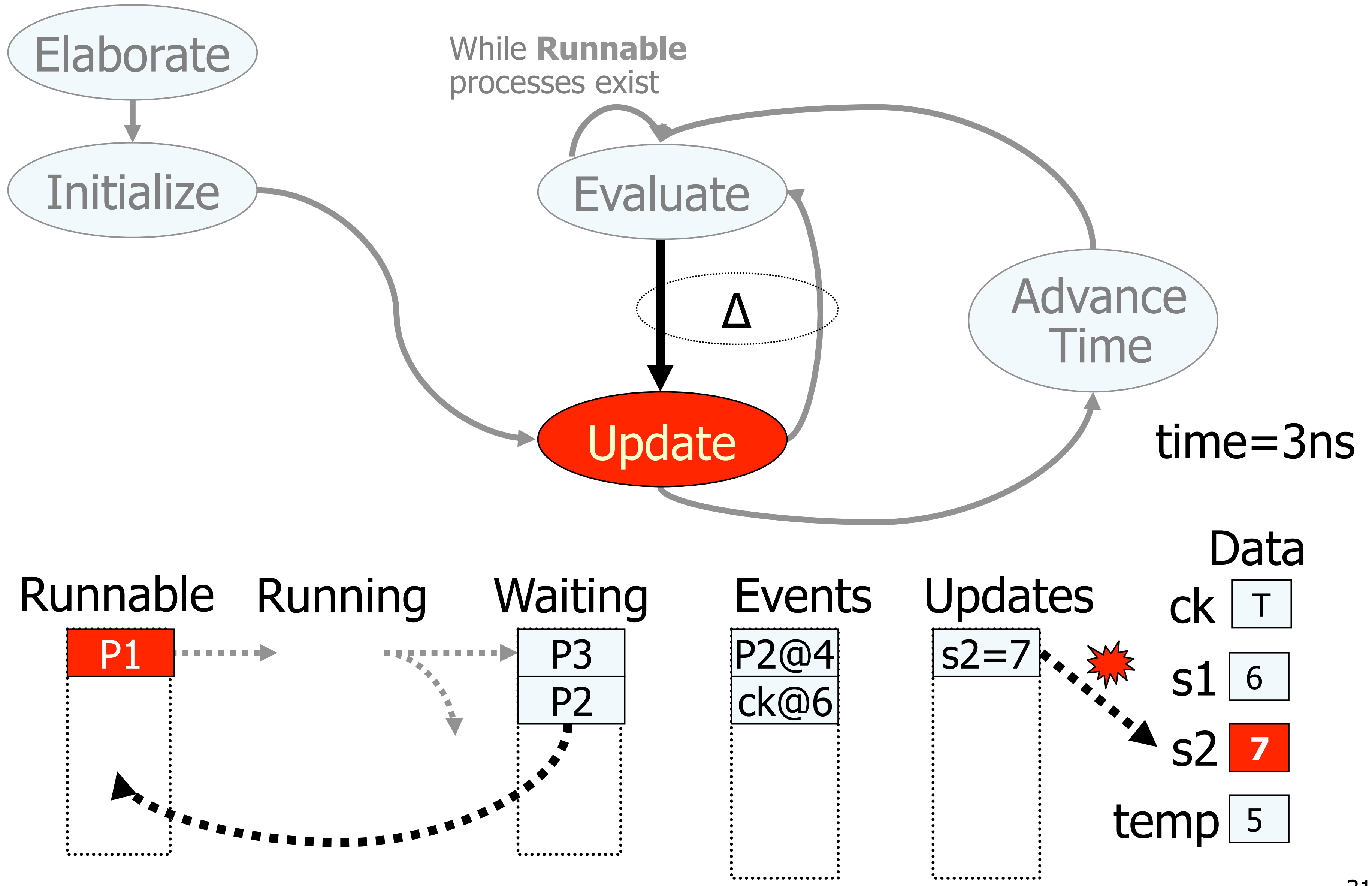


T10c Evaluate

```
B:for (int i=7,i<9;i++){
    s2.write(i);
    wait(1,SC_NS);
```



T11 Update

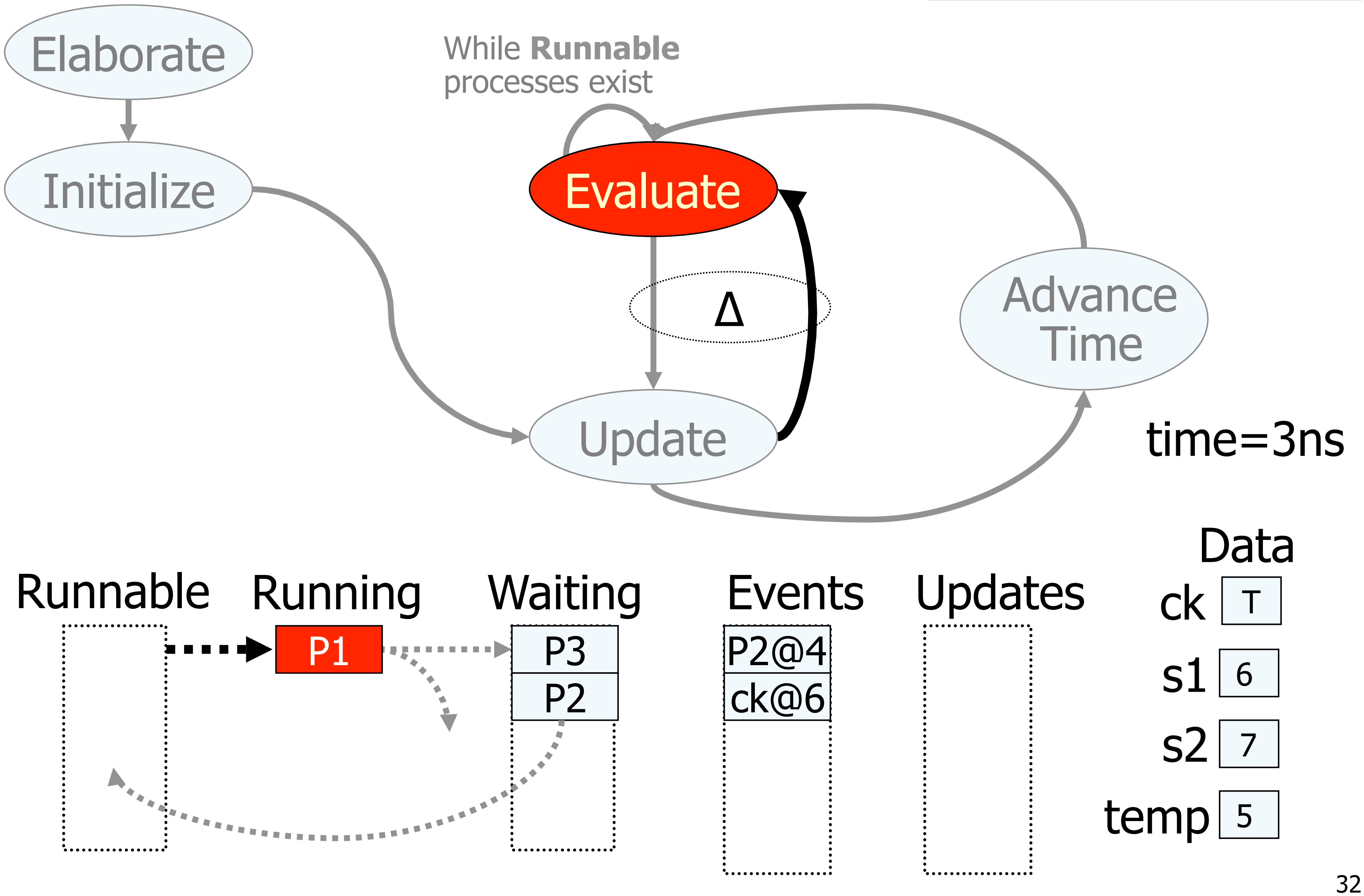


While **Runnable** processes exist
Elaborate
Initialize
Evaluate
Δ
Update
Advance Time
time=3ns
Runnable
P1
Running
Waiting
P3
P2
Events
P2@4
ck@6
Updates
s2=7
Data
ck
T
s1
6
s2
7
temp
5

T12a Evaluate

P1 →

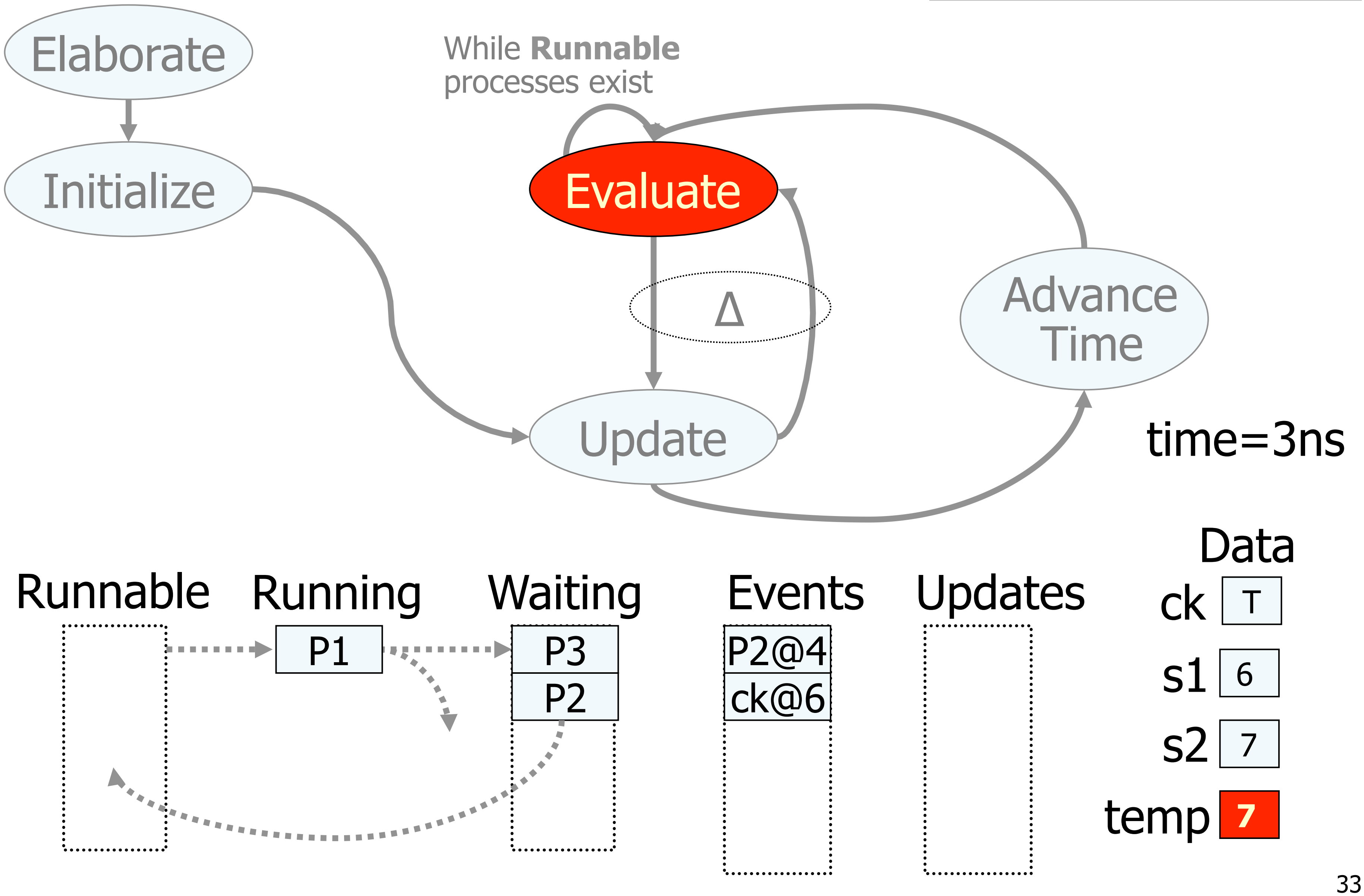
```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
```



T12b Evaluate

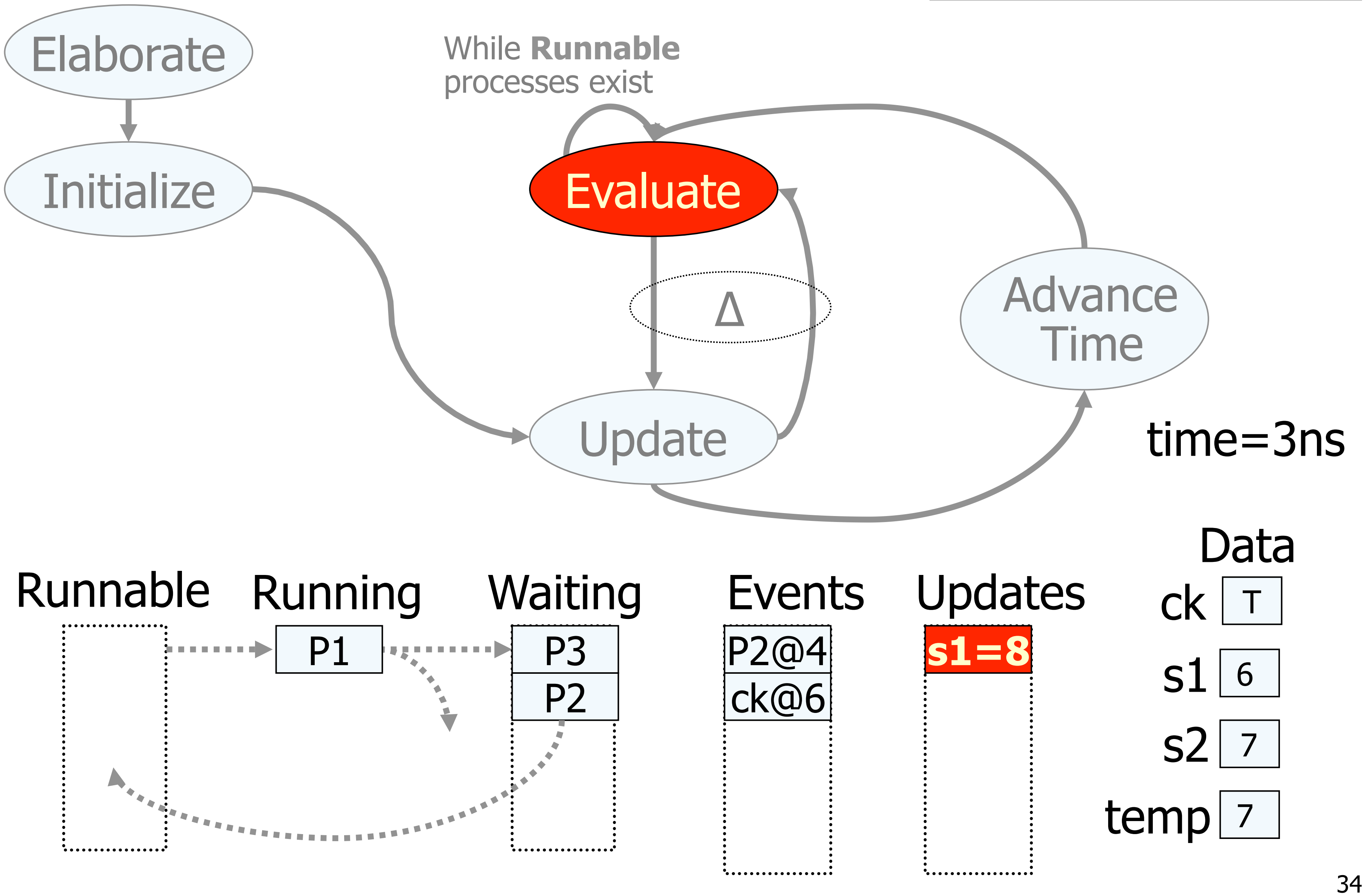
P1 →

```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
```



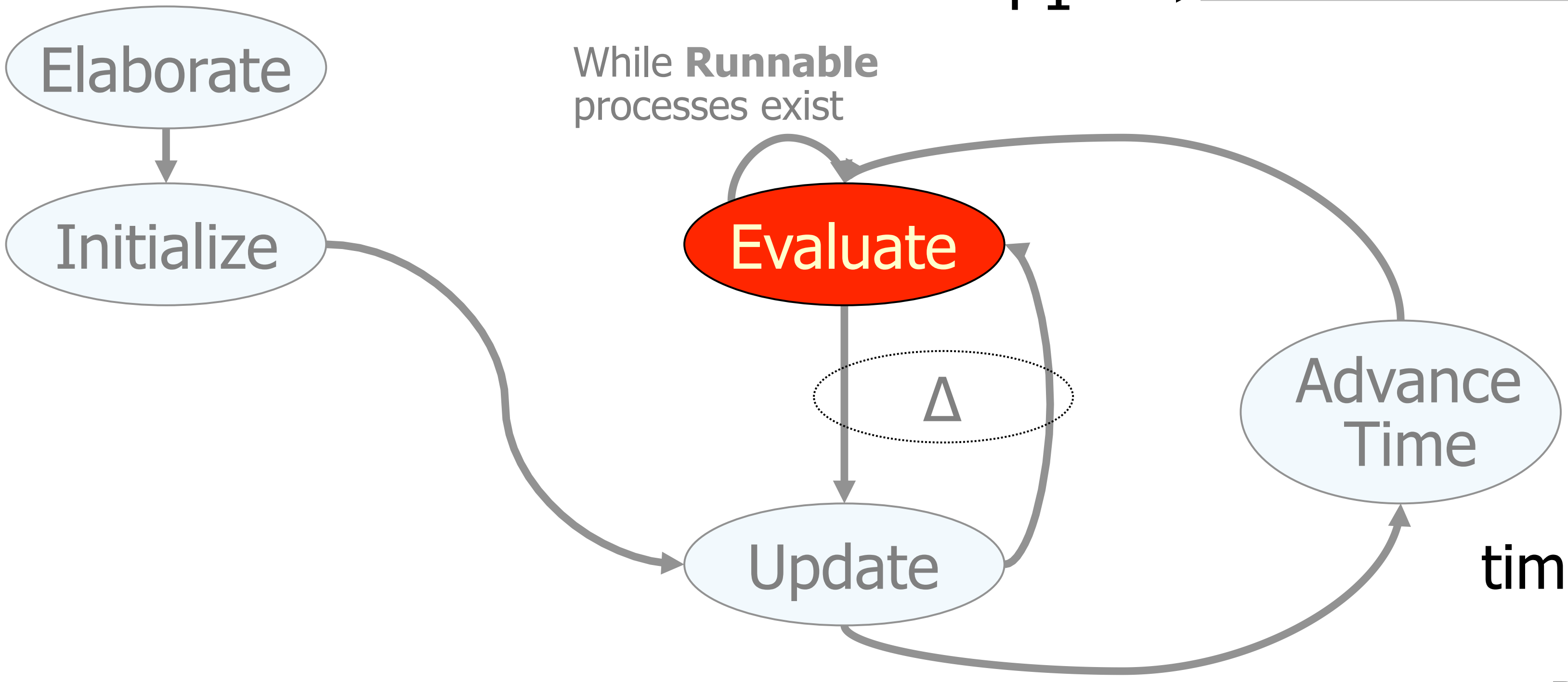
T12c Evaluate

P1 →
```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
```

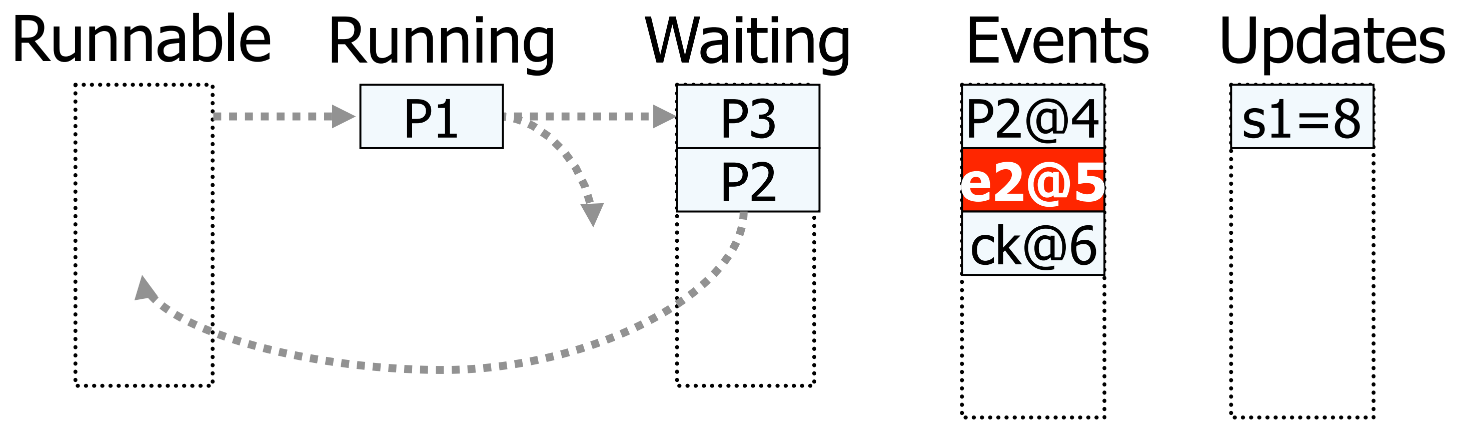


T12d Evaluate

P1 →
```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
```



time=3ns

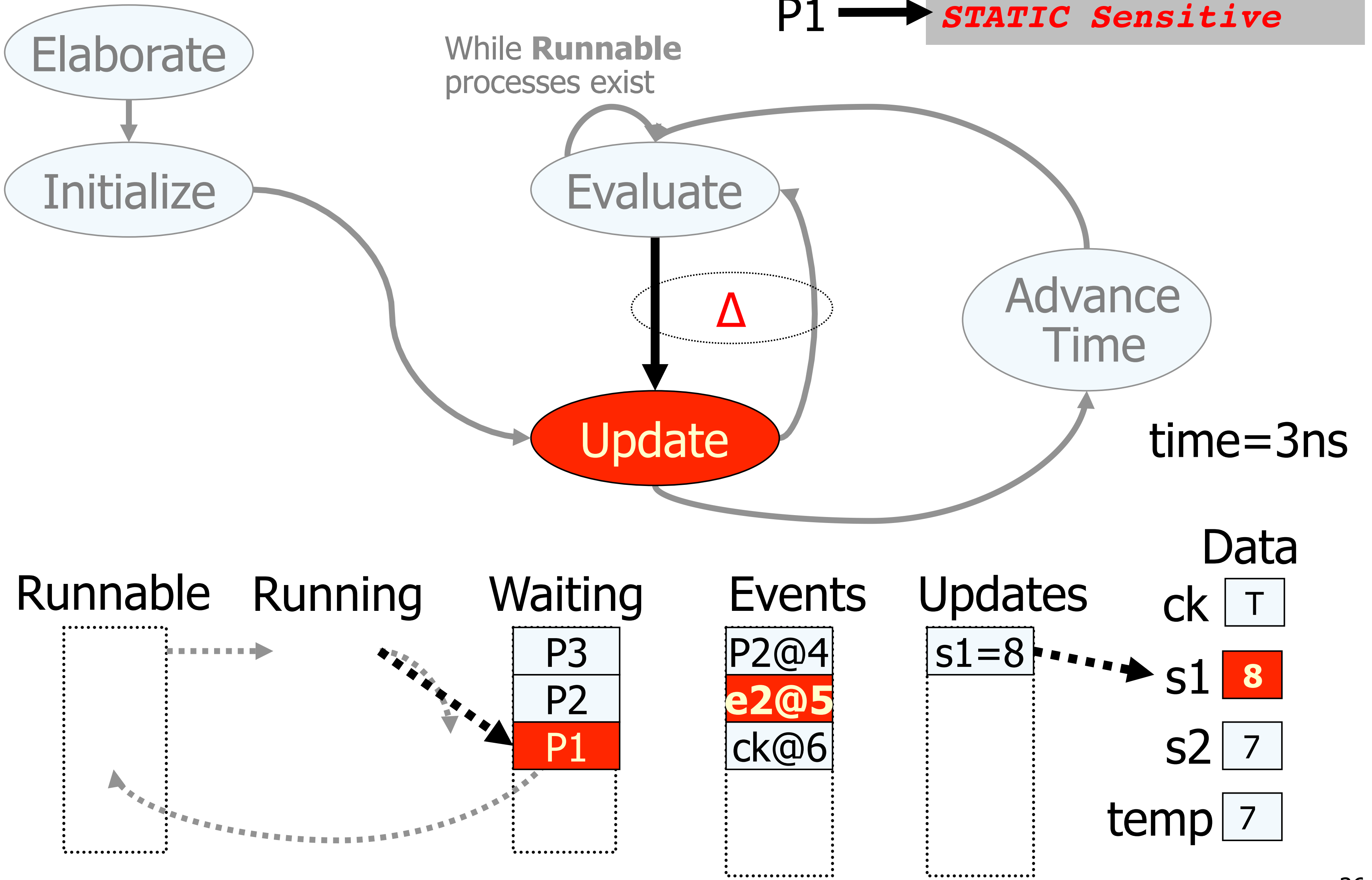


Data

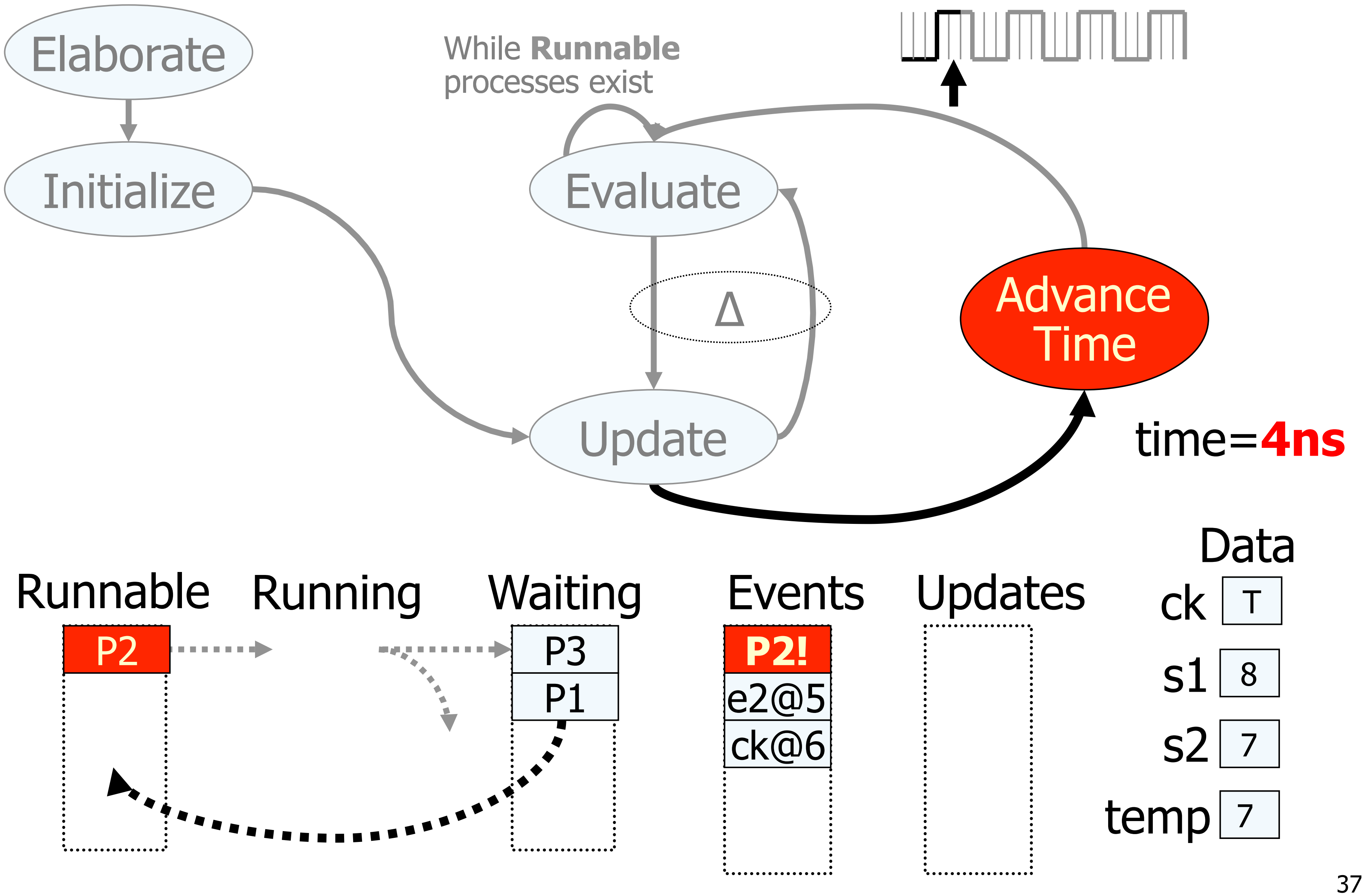
ck	T
s1	6
s2	7
temp	7

T13 Update

```
temp = s2.read();
s1p->write(temp+1);
e2.notify(2,SC_NS);
STATIC Sensitive
```

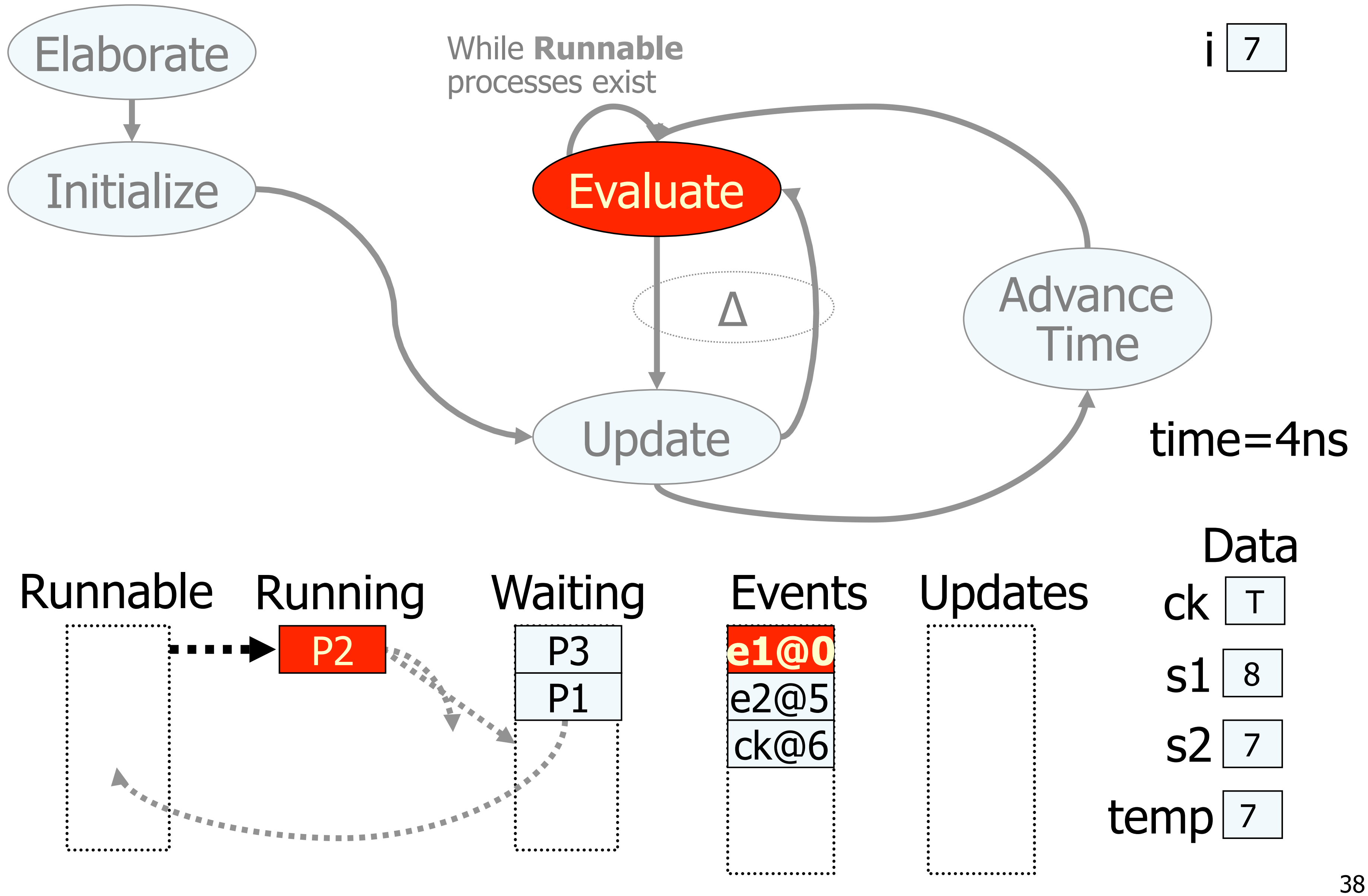


T14a Advance Time



T14b Evaluate

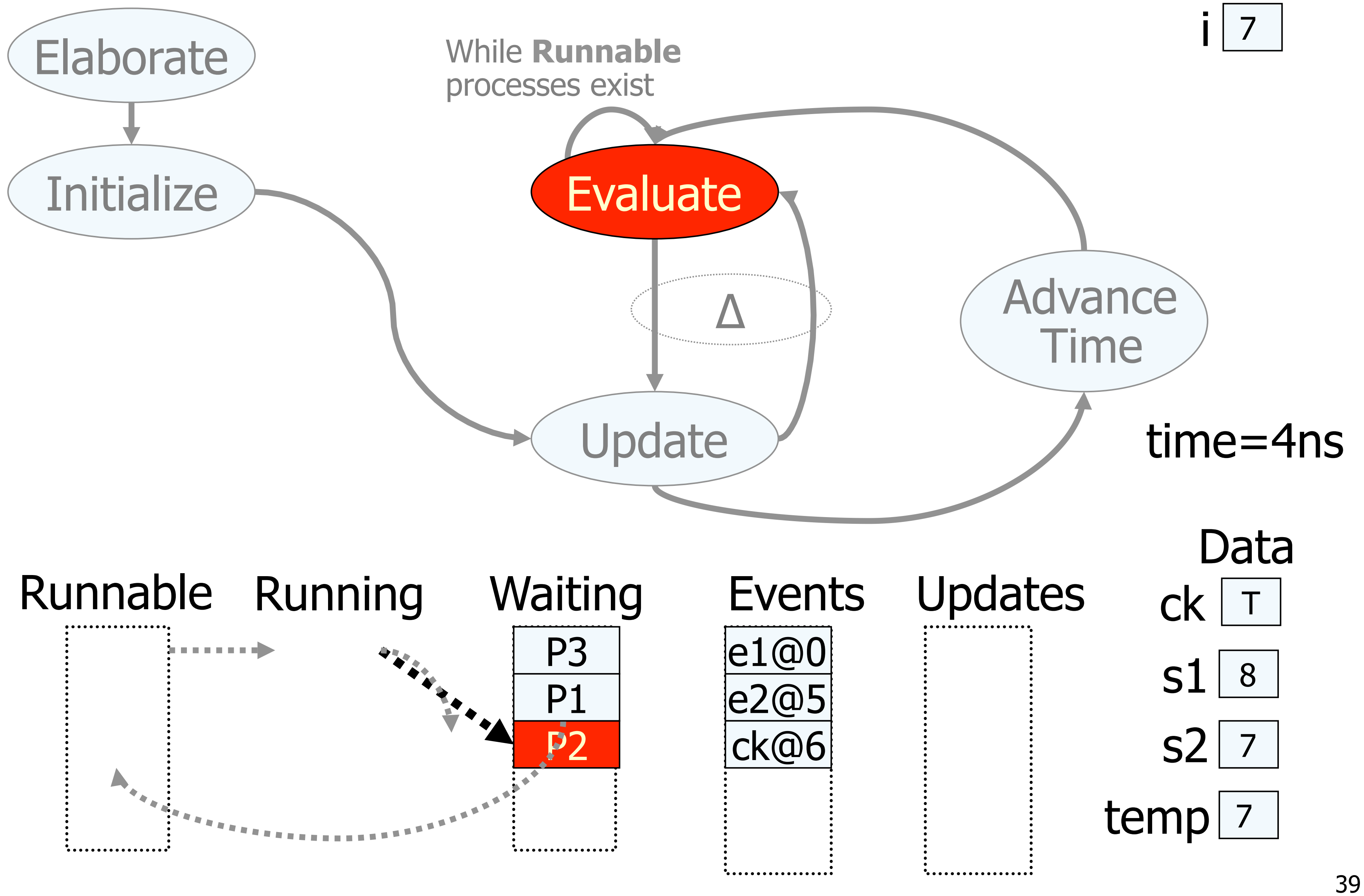
P2 →
```
C:e1.notify(SC_ZERO_TIME);
  wait();//static sensitive
```



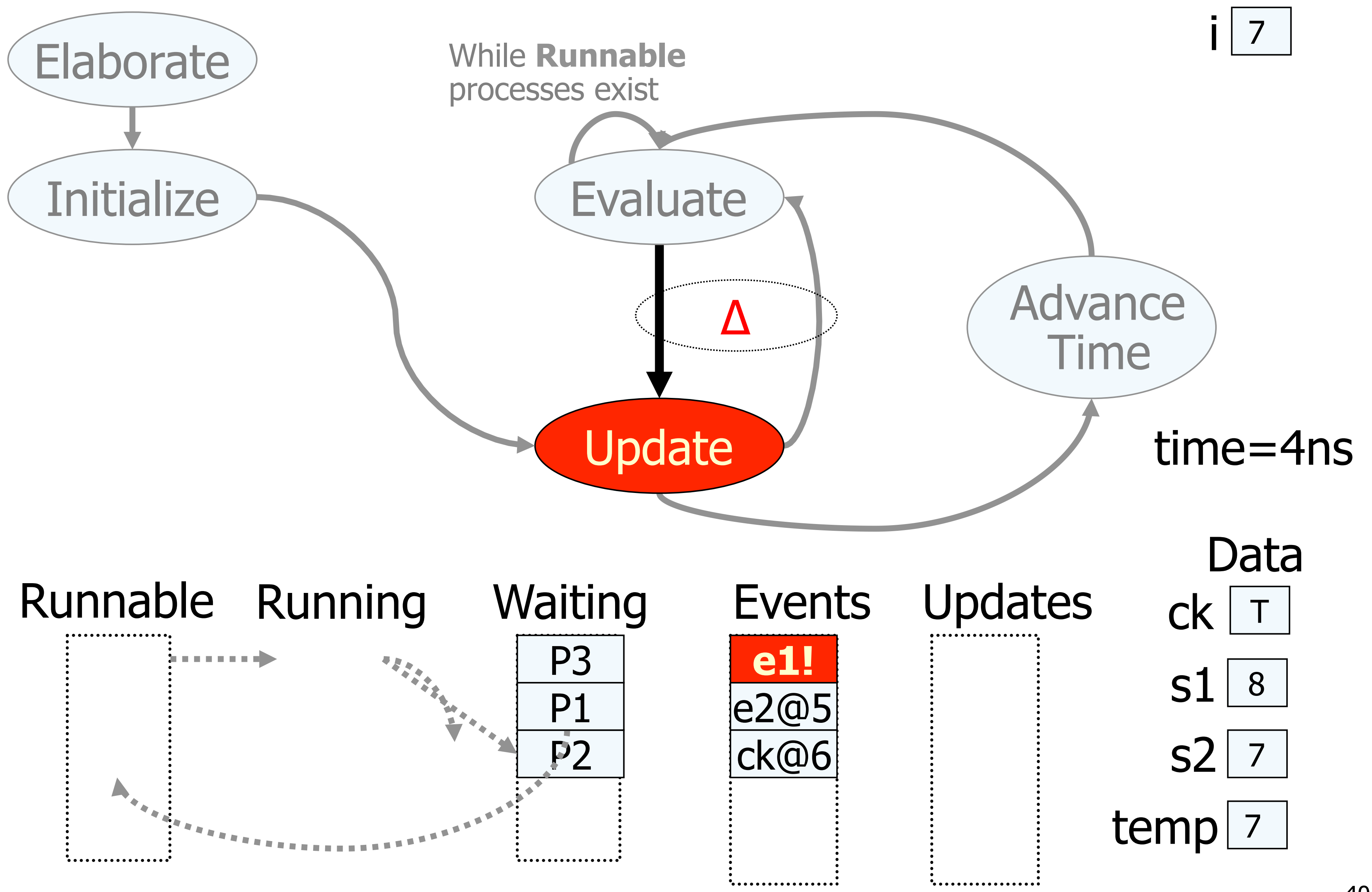
T14c Evaluate

P2 →

```
C:e1.notify(SC_ZERO_TIME);
  wait();//static sensitive
```



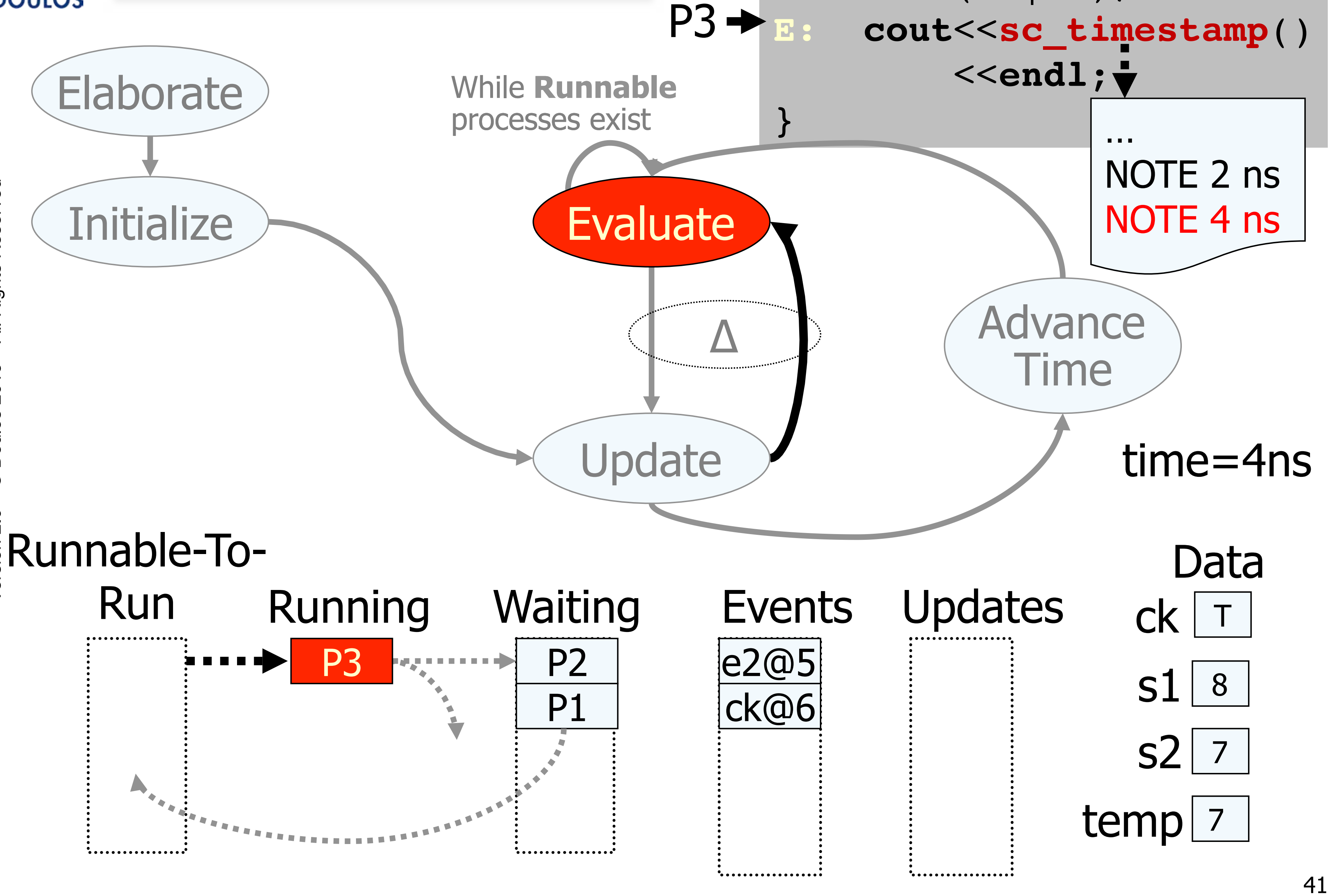
T15 Update



i 7
Elaborate
Initialize
While **Runnable** processes exist
Evaluate
Δ
Update
Advance Time
time=4ns
Data
Runnable
Running
Waiting
P3
P1
P2
Events
e1!
e2@5
ck@6
Updates
ck T
s1 8
s2 7
temp 7

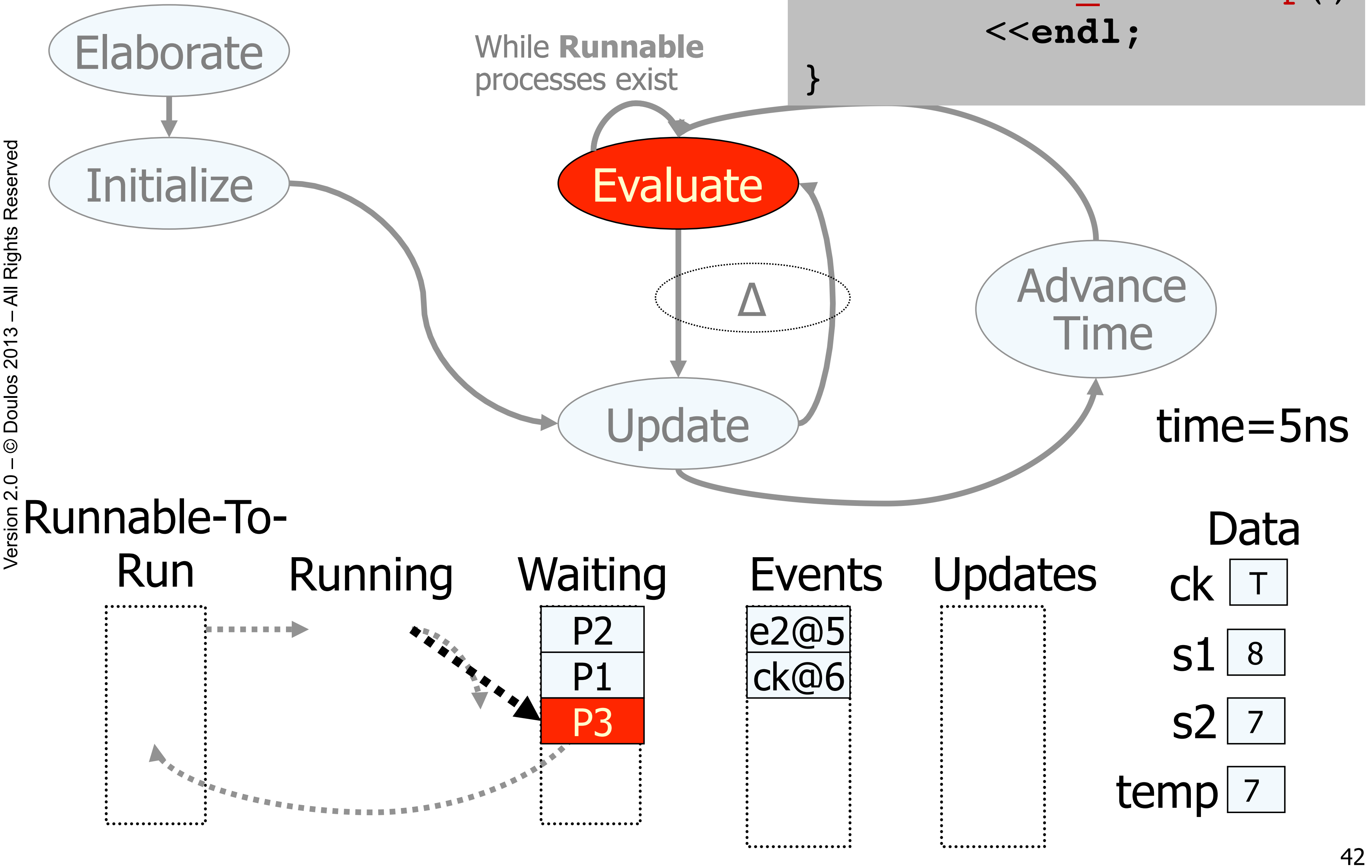
T16a Evaluate

```
D:while(true){
     wait(e1|e2);
E:   cout<<sc_timestamp()
         <<endl;
}
```

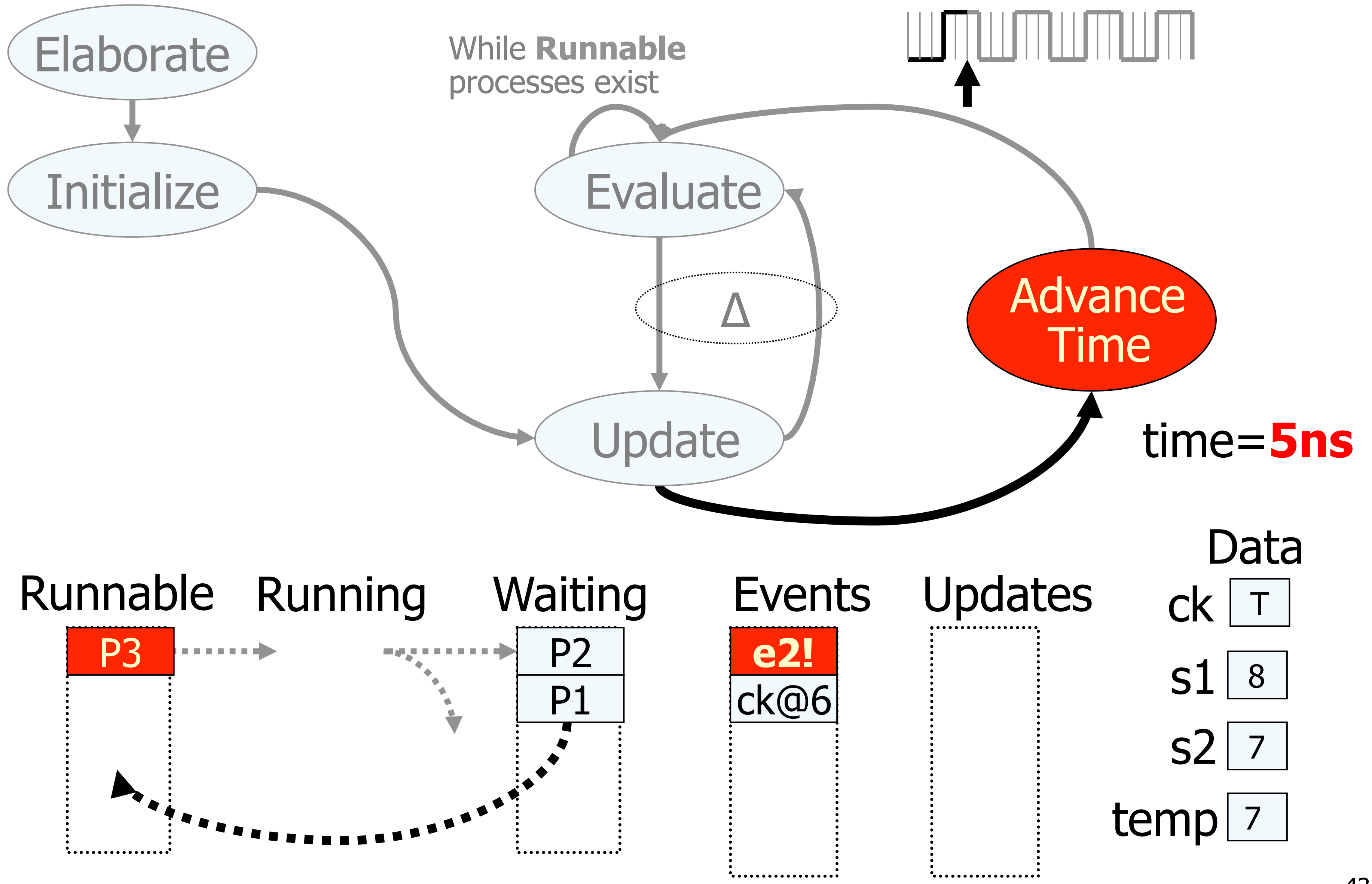


T16b Evaluate

```
D:while(true){
    wait(e1|e2);
E:   cout<<sc_timestamp()
         <<endl;
}
```

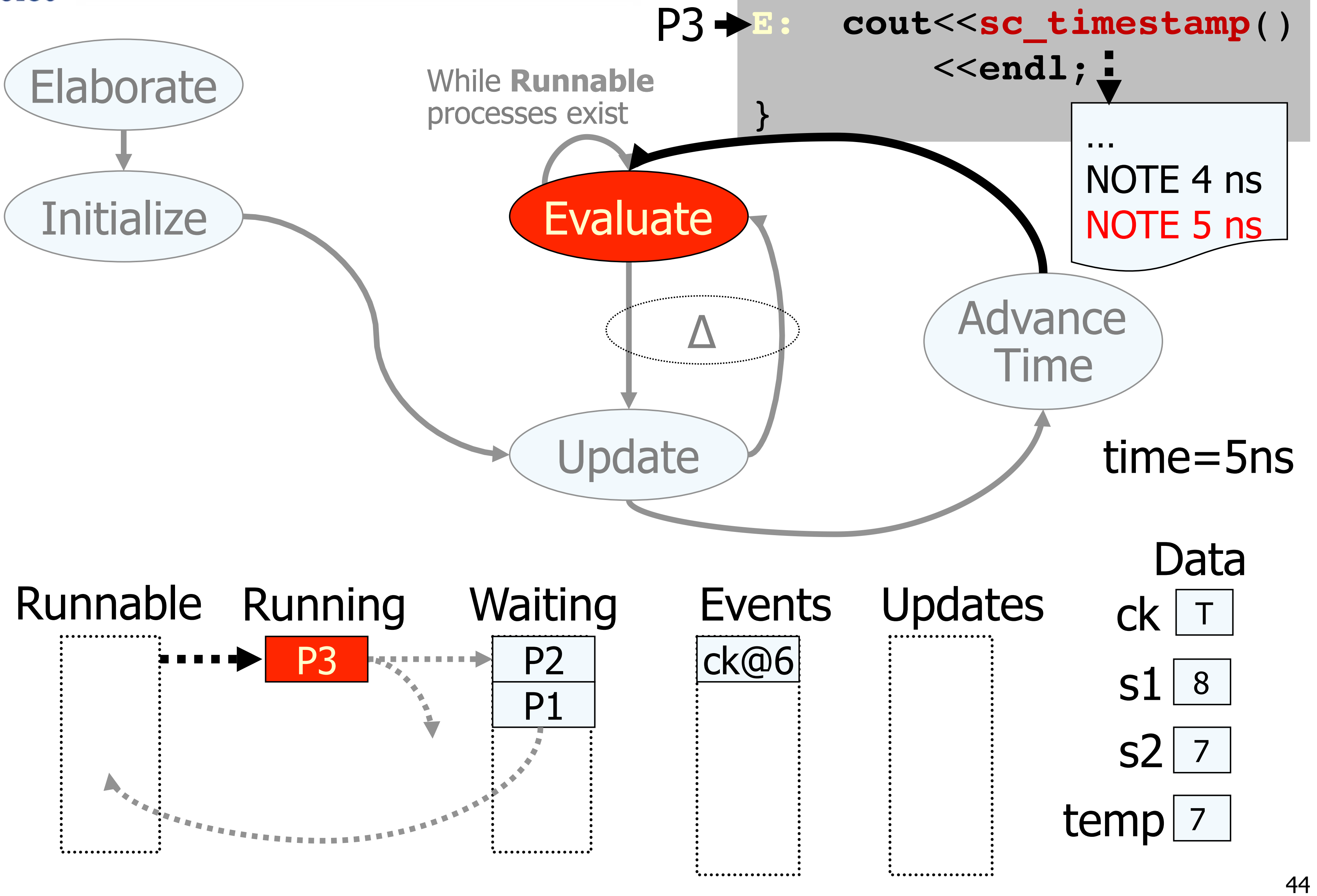


T17 Advance Time



T18a Evaluate

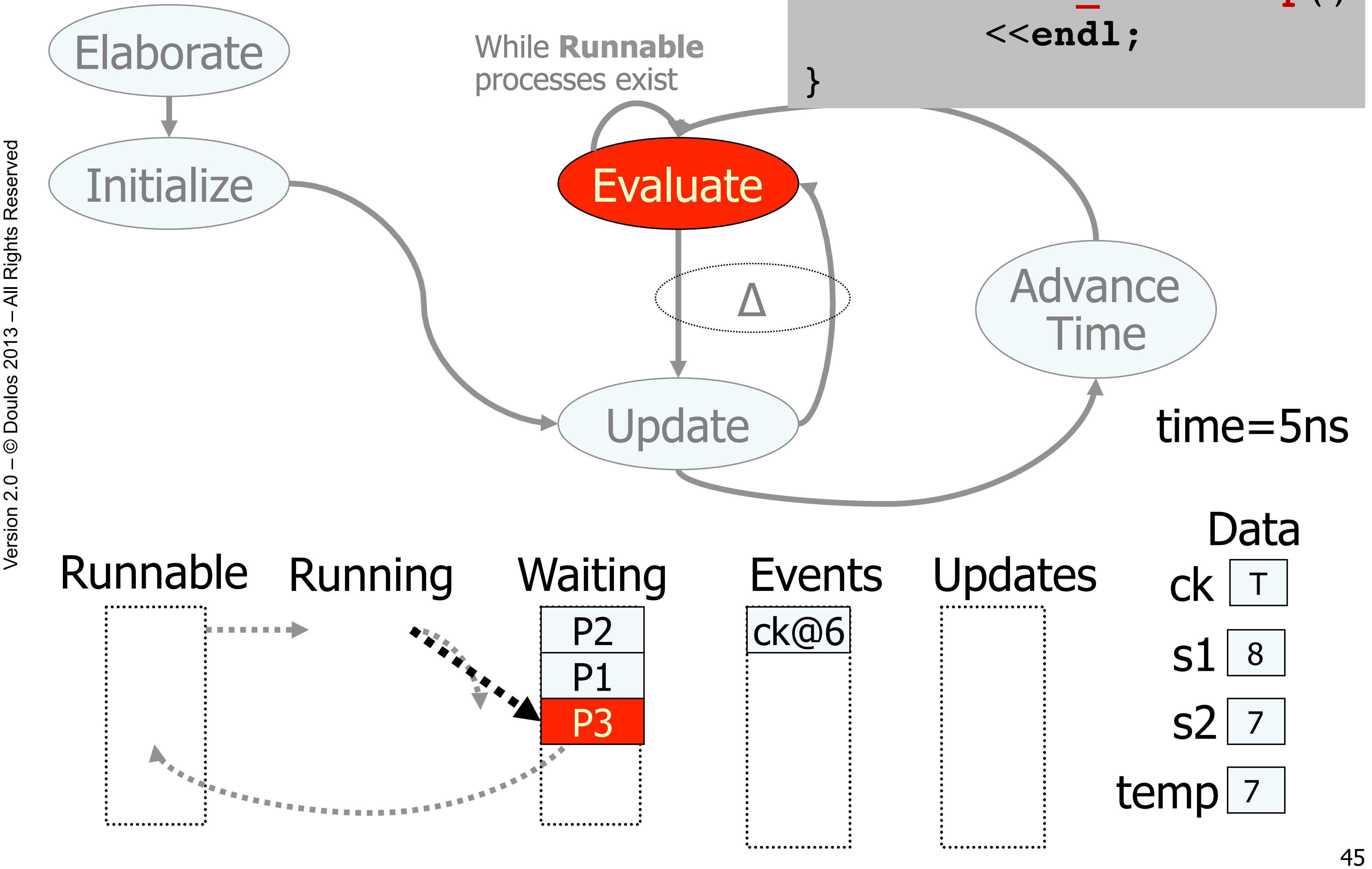
```
D:while(true){
     wait(e1|e2);
E:   cout<<sc_timestamp()
            <<endl;
}
```



T18b Evaluate

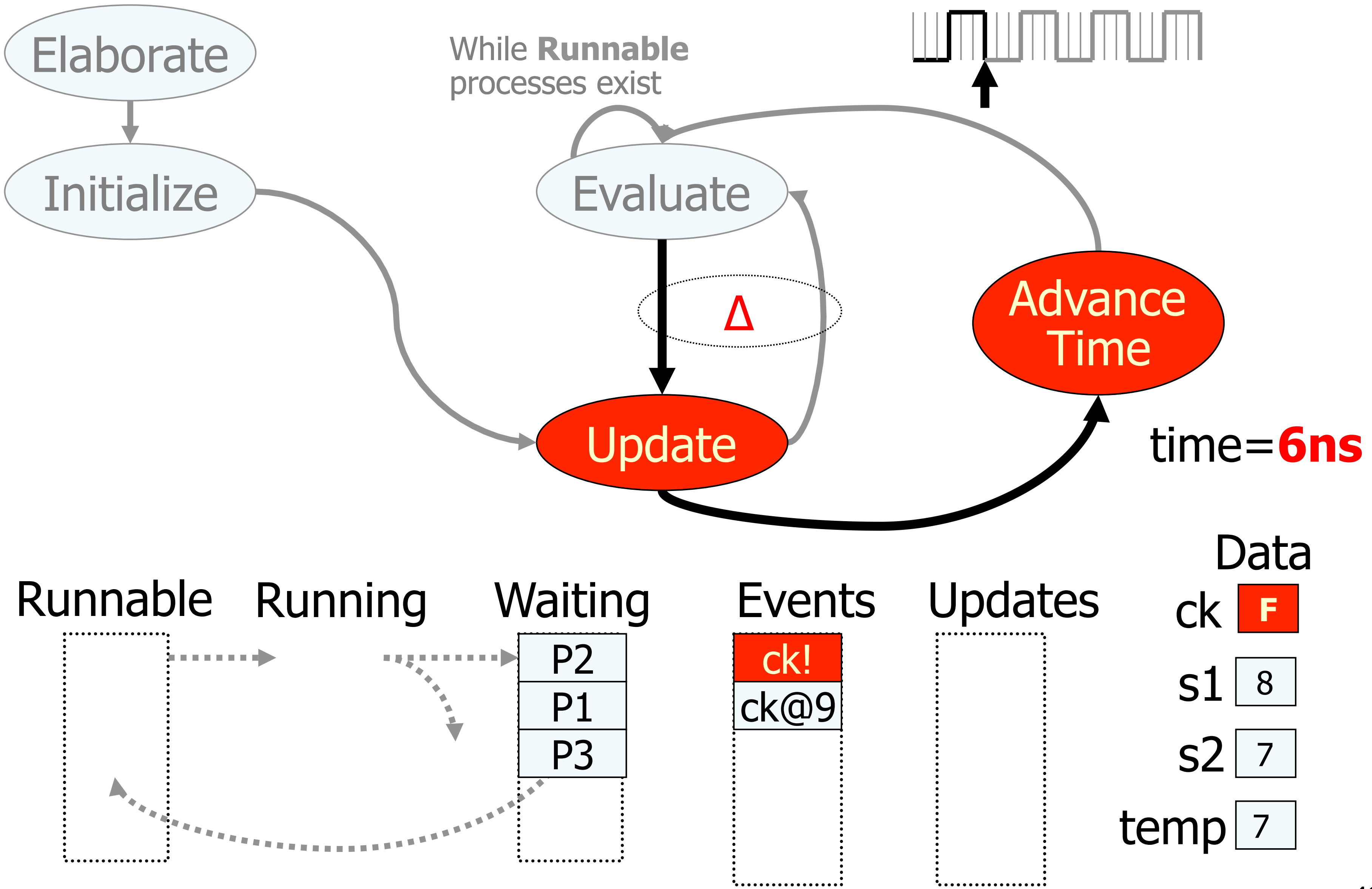
P3 →

```
D:while(true){
    wait(e1|e2);
E:  cout<<sc_timestamp()
        <<endl;
}
```



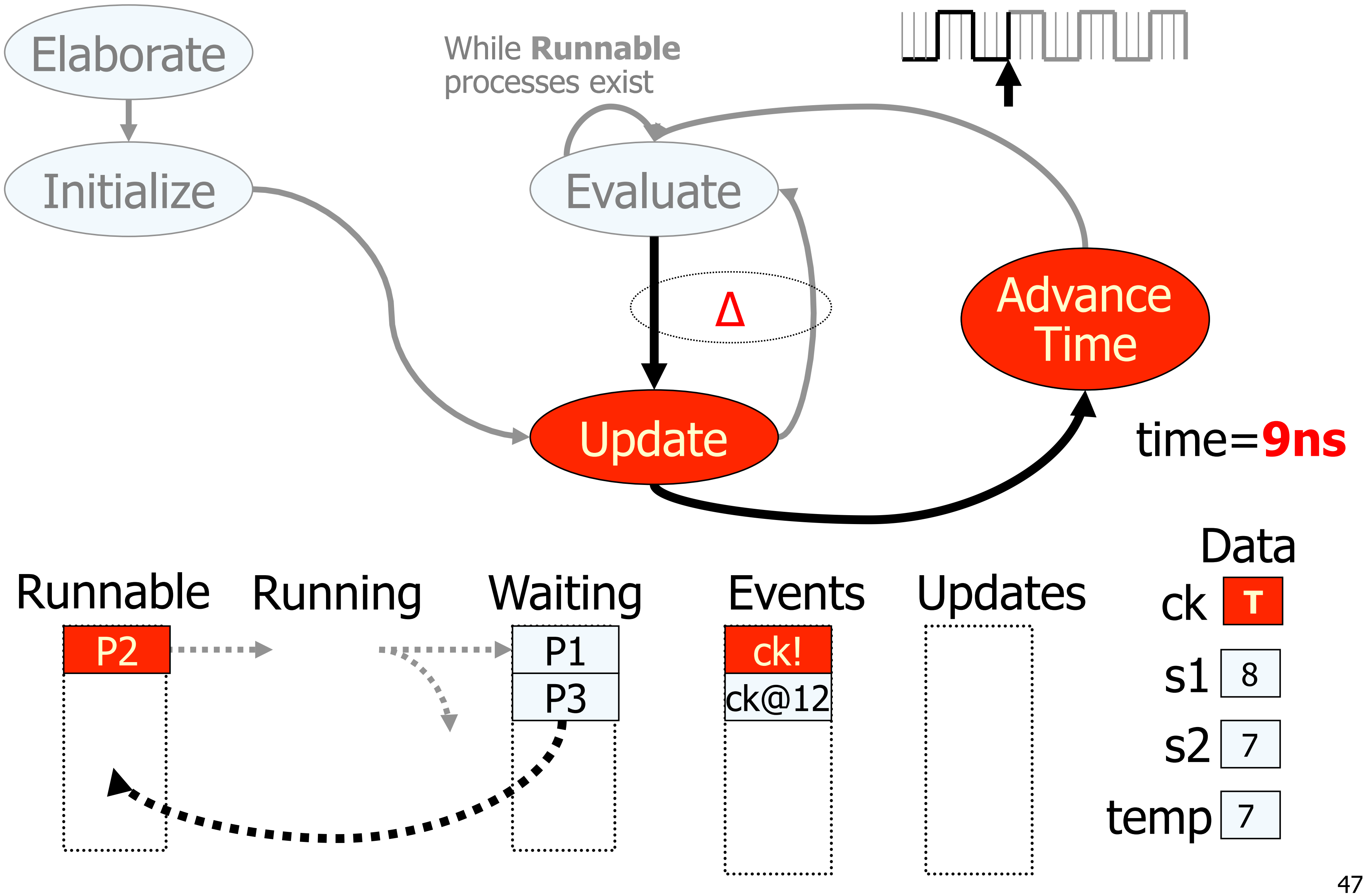
T19/20 Advance Time

```
sc_clock ck("ck",6,0.5,3);
```



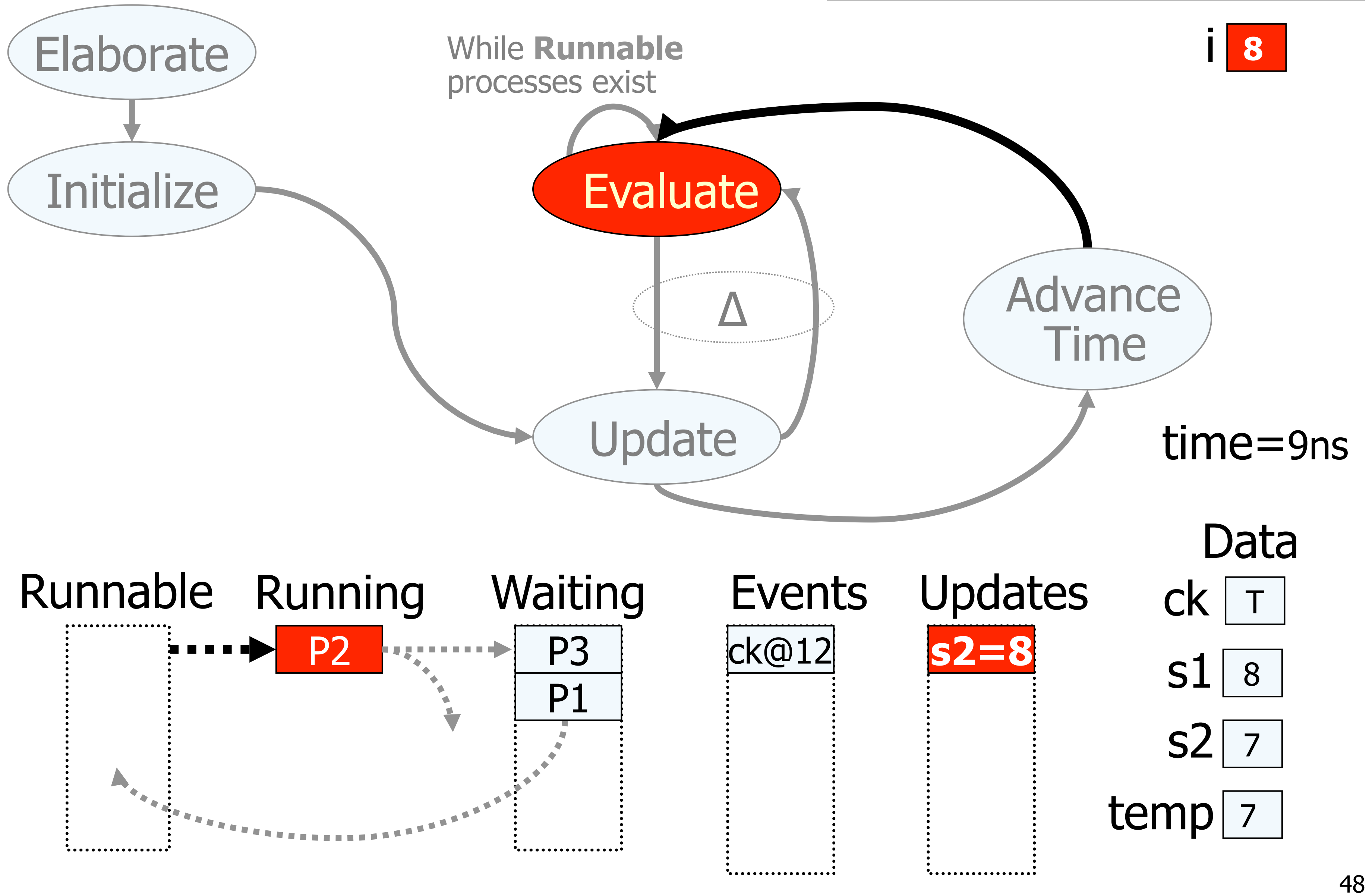
T21/22 Advance Time

```
sc_clock ck("ck",6,0.5,3);
```



T23a Evaluate

```
B:for (int i=7,i<9;i++){
    s2.write(i);
    wait(1,SC_NS);
```



…and on and on it goes…

- Simulation stops when
  - No more **Runnable** nor **Waiting** processes
  - Encounters `sc_stop()` or `SC_FATAL(…)`
  - Simulated time exceeds specified `sc_start(tMAX)` time
  - Simulated time exceeds 64 bits
    - 2^{64} ps = 30 weeks 3 days 12 hours 5 min 44 sec
  - Bad stuff
    - Explicit `exit()` or `abort()`
    - unhandled exception (control-C)
    - bad memory access (pointer gone wrong)
    - out of memory (leak)
    - stack overflow

Observations

- Sample code does not follow best practices
  - designed to be short enough to fit on one page
  - only a single module
  - sc_in, sc_out discouraged
    - prefer sc_port< sc_signal_inout_if< T > >
    - unless you need the event finder specialization
  - sc_signal is a low-level construct (RTL)
  - sc_clock slows simulation - too much context switching
- Serious ESL simulations
  - involve 10's to 1,000's of processes
  - contain lots of module hierarchy and interconnect
  - use higher level of abstraction (TLM)
  - don't have explicit clocks

Process order of execution

- Processes model simulated concurrency (parallel execution)
- Each SystemC implementation has own initial ordering
  - important to allow reproducible results
  - should not depend on this behavior
- Can change this ordering by shuffling process registration
- Design required dependencies should be dictated by explicit events and handshakes

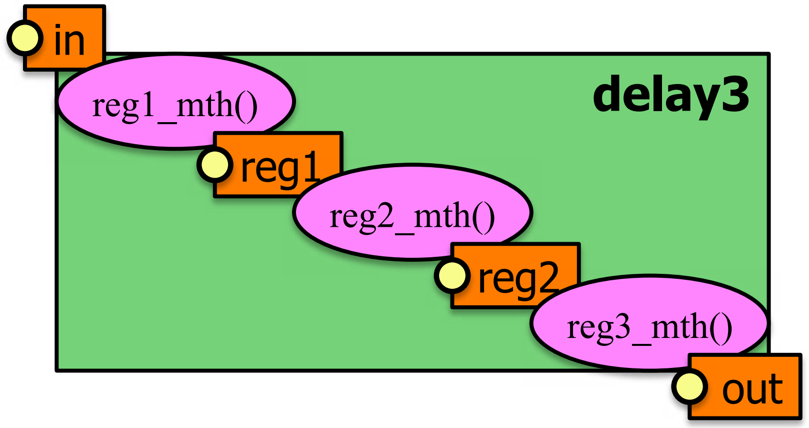
```
SC_CTOR(M)//force random
 …
 if (randomize) {
  switch (random()%3) {
   case 0: SC_THREAD(P3_thrd);
      break;
   case 1: SC_THREAD(P2_thrd);
      sensitive<<ckp.pos();
      break;
   case 2:
    SC_METHOD(P1_meth);
      sensitive<<s2;
      dont_initialize();
  }//endcase
 }//endif
}//end SC_CTOR
```

Guaranteeing order

- Would using simple int instead of `sc_signal<int>` work?

```
SC_MODULE(delay3) {
 sc_in<int> in_port;
 sc_out<int> out_port;
 sc_signal<int> reg1, reg2;
 void reg1_mth(void) {
   v1.write(in_port->read());
 }
 void reg2_mth(void) {
   v2.write(v1.read());
 }
 void reg3_mth(void) {
   out_port->write(v2.read());
 }
```

```
SC_CTOR(delay3) {
   SC_METHOD(reg2_mth);
   sensitive << in_port;
   SC_METHOD(reg1_mth);
   sensitive << in_port;
   SC_METHOD(reg3_mth);
   sensitive << in_port;
  }
};
```



How sc_signal works

- During update kernel calls update() method on all objects that did request_update() in preceding delta cycle
- write() is equivalent to verilog non-blocking assignment

```
mysig.write(expr);
myvar <= expr;
```

```
struct buffer
: sc_prim_channel, buffer_if {

 int read(void){return curr; }

 void write(int v)  {
   next = v; request_update();
 }
 void update(void)  {
   curr = next; evt.notify();
 }
 sc_event& written_evt(void)  {
   return evt;
 }
private:
  int curr, next;
  sc_event evt;
};
```

For further study

- Download
  - Examine the basic code discussed
  - Examine the output log file
  - Examine the instrumented code used for the above
  - Includes the source to reproduce (gzip'd tar)
  - https://www.dropbox.com/s/zqe64ujvqbtve5w/engine.tgz
- Includes this presentation
- Try to get different results
  - Reorder the process registration code
  - Try a different platform, simulator vendor, or version

Common C++ Coding Pitfalls

- Forgetting semi-colon (;) on class/struct
- Forgetting to tag **`private`**, **`protected`** or **`public`**
- Copy-paste error on header guard (or completely forgetting)
- Failure to implement code separate from declaration
- Attempting construction in a declaration
- Forgetting class-name qualifier when implementing separately
- Direct instantiation of class members leading to excess header needs
- Failure to use pass-by-reference (esp. for polymorphism)
- Abuse of pointers leading to memory faults
- Omitting the constructor or destructor
- Omitting copy-constructor or **`operator=`**
- Using **`printf`** instead of **`boost::format`**
- Failure to use STL or boost
- Not using const, and/or using #define

Common SystemC Coding Pitfalls

- Templating `sc_port` on non-`sc_interface` type
- Forgetting to bind (connect) all ports
- Ports use dot (.) operator instead of arrow (->) operator
- Using blocking functions inside SC_METHOD processes
- Incorrectly locating channels relative to `sc_port` or `sc_export`
- Coding at RTL level - too much detail
- Infinite loop path missing `wait` in SC_THREAD process
- Attempting to `sc_stop` and restart with `sc_start`
- Simulation phase actions during elaboration phase
- Using `std::cout` instead of `SC_REPORT_INFO`
- Using `std::cerr` instead of `SC_REPORT_ERROR`
- Elaboration phase actions during simulation phase
- Excessive context switching or I/O causing simulation to crawl
- Converting `SC_THREAD`s to `SC_METHOD`s to gain performance without first profiling code to determine real cause of slowdowns

SystemC Guidelines

- Abstract as high as possible
- Code as simply and cleanly as possible
  - State machines and wires are messy and hard to debug
- Avoid too much context switching (think)
- Be careful with immediate notification (delayed is safer)
- Communicate between processes with channels
- Communicate across module boundaries with ports
- Use indirect instantiation for flexibility
- Limit details - Model to requirements
- Improve your C++ coding skills

DOULOS

Questions

DOULOS

www.doulos.com