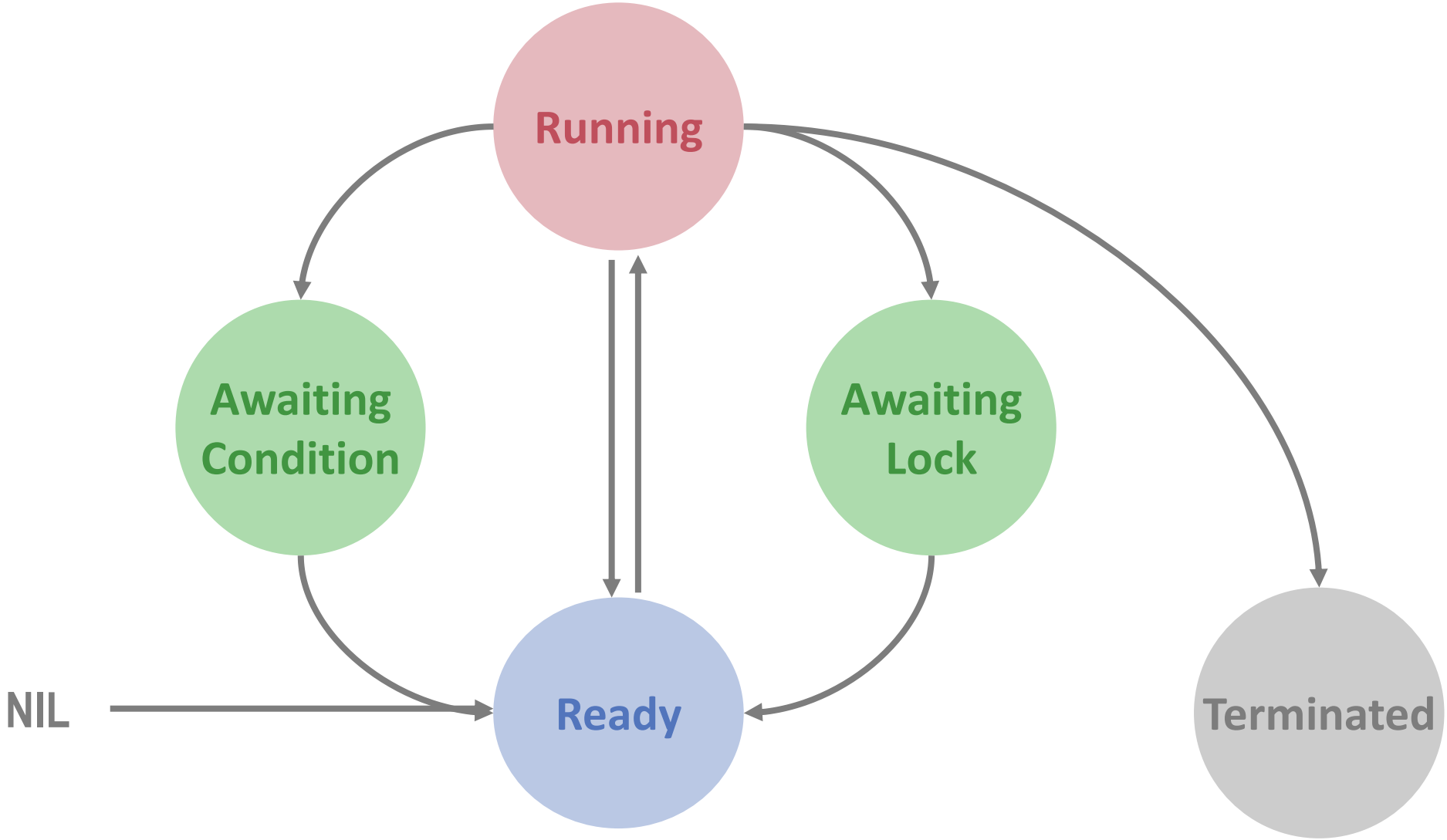
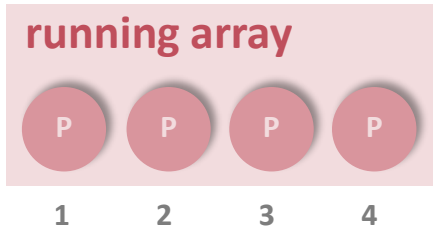
2.3. ACTIVITY MANAGEMENT

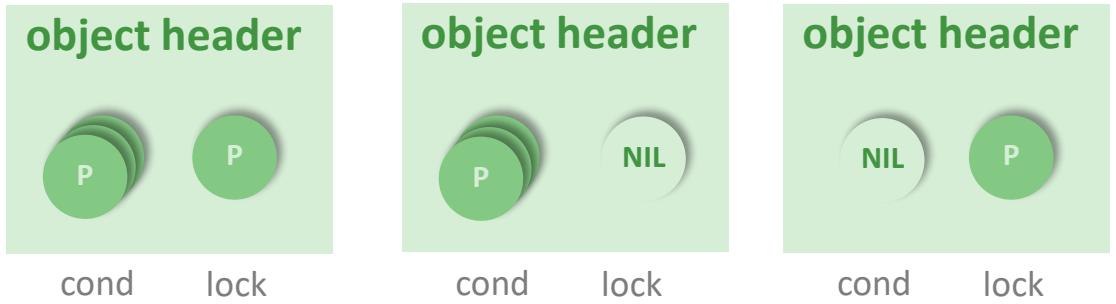
Life Cycle of Activities



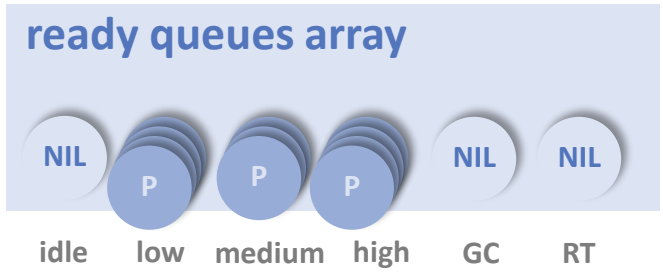
Runtime Data Structures



global



per (monitor) object



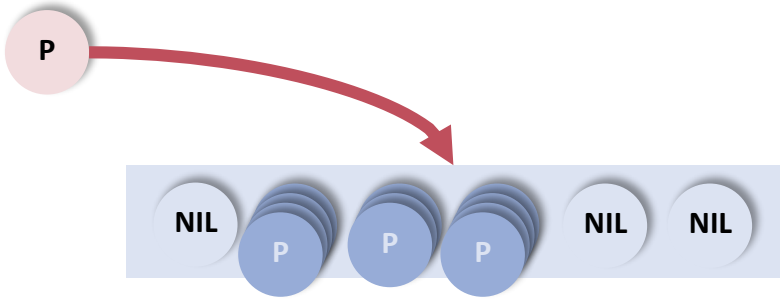
global

Process Descriptors

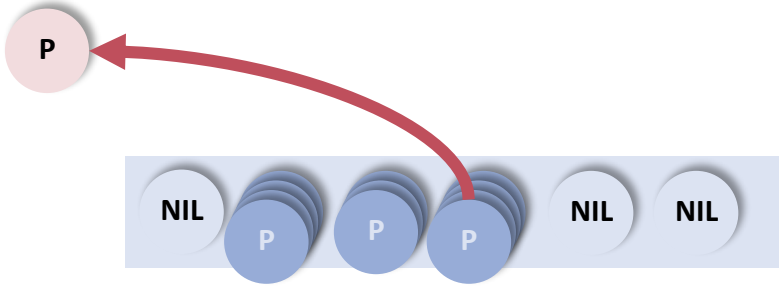
```
TYPE
  Process = OBJECT
    …
    stack: Stack;
    state: ProcessState;
    preempted: BOOLEAN;
    condition: PROCEDURE (slink: ADDRESS);
    conditionFP: ADDRESS;
    priority: INTEGER;
    obj: OBJECT;
    next: Process
  END Process;


  ProcessQueue = RECORD
    head, tail: Process
  END;


VAR
  ready: ARRAY NumPriorities OF ProcessQueue;
  running: ARRAY NumProcessors OF Process;
```

Process Dispatching

```
PROCEDURE Enter (p: Process);
BEGIN
  Put(ready[p.priority], p);
  IF p.priority > maxReady THEN
    maxReady := p.priority
  END
END Enter;
```

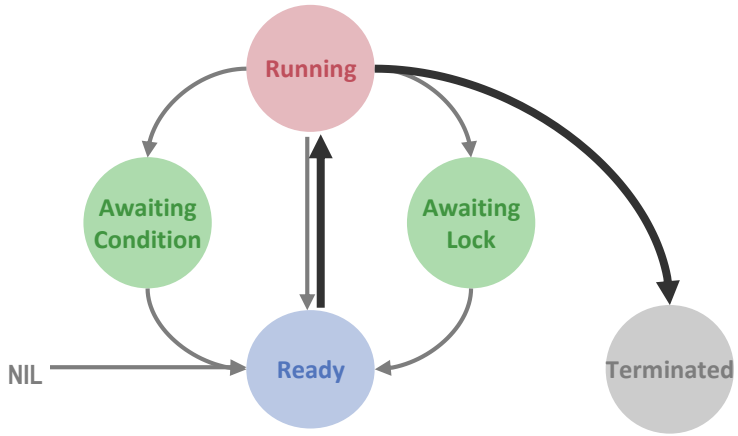



```
PROCEDURE Select (VAR new: Process; priority: integer);
BEGIN
  LOOP
    IF maxReady < priority THEN new := nil; EXIT END;
    Get(ready[maxReady], new);
    IF(new # NIL) OR (maxReady = MinPriority) THEN
      EXIT
    END;
    maxReady := maxReady-1
  END
END Select;
```



Process Creation

```
PROCEDURE CreateProcess (body: ADDRESS; priority: INTEGER; obj: OBJECT);
  VAR p: Process;
BEGIN
  NEW(p); NewStack(p, body, obj);
  p.preempted := false;
  p.obj := obj; p.next := nil;
  RegisterFinalizer(p, FinalizeProcess);
  Acquire(Objects); (* module lock *)
  IF priority # 0 THEN p.priority := priority
  ELSE (* inherit priority of creator *)
    p.priority := running[ProcessorID()].priority
  END;
  Enter(p);
  Release(Objects)
END CreateProcess;
```

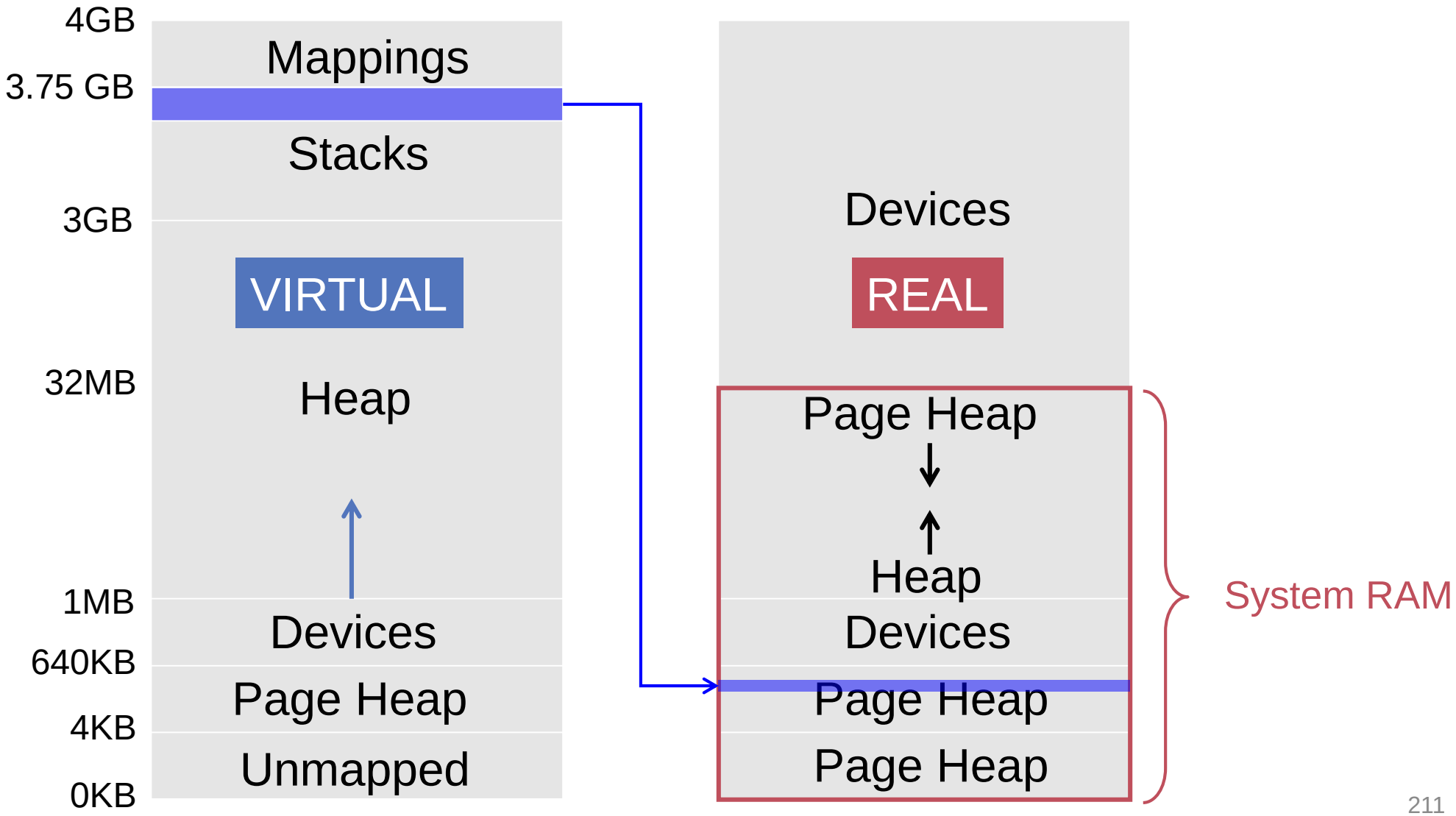


Stack Management

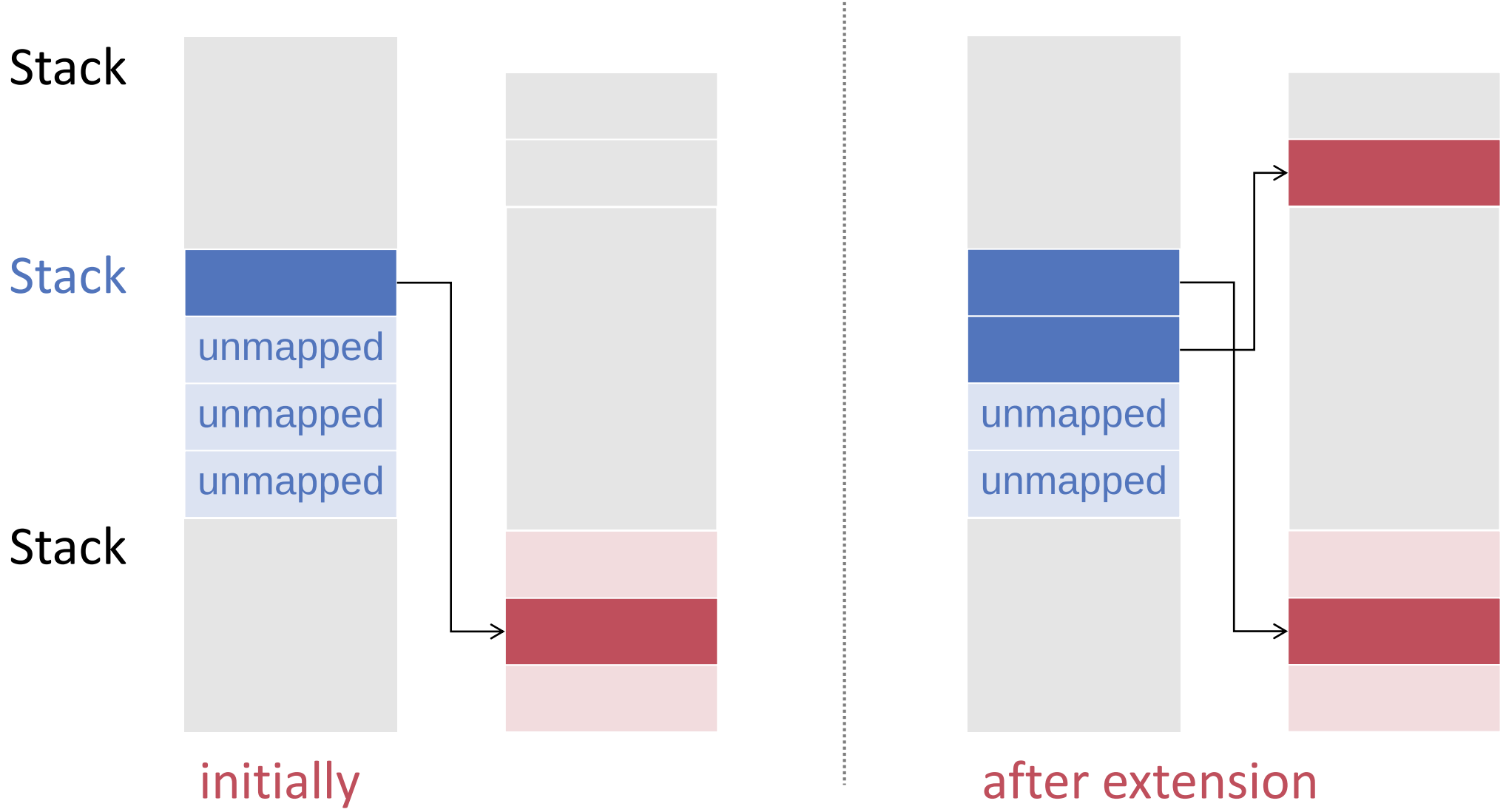
- Use virtual adressing
- Allocate stack in page units
- Use page fault for detecting stack overflow
- Deallocate stack via garbage collector
  (in process finalizer)

CreateProcess	Allocate first frame
Page fault	Allocate another frame
Finalize	Deallocate all frames

Memory Layout



Stack Allocation

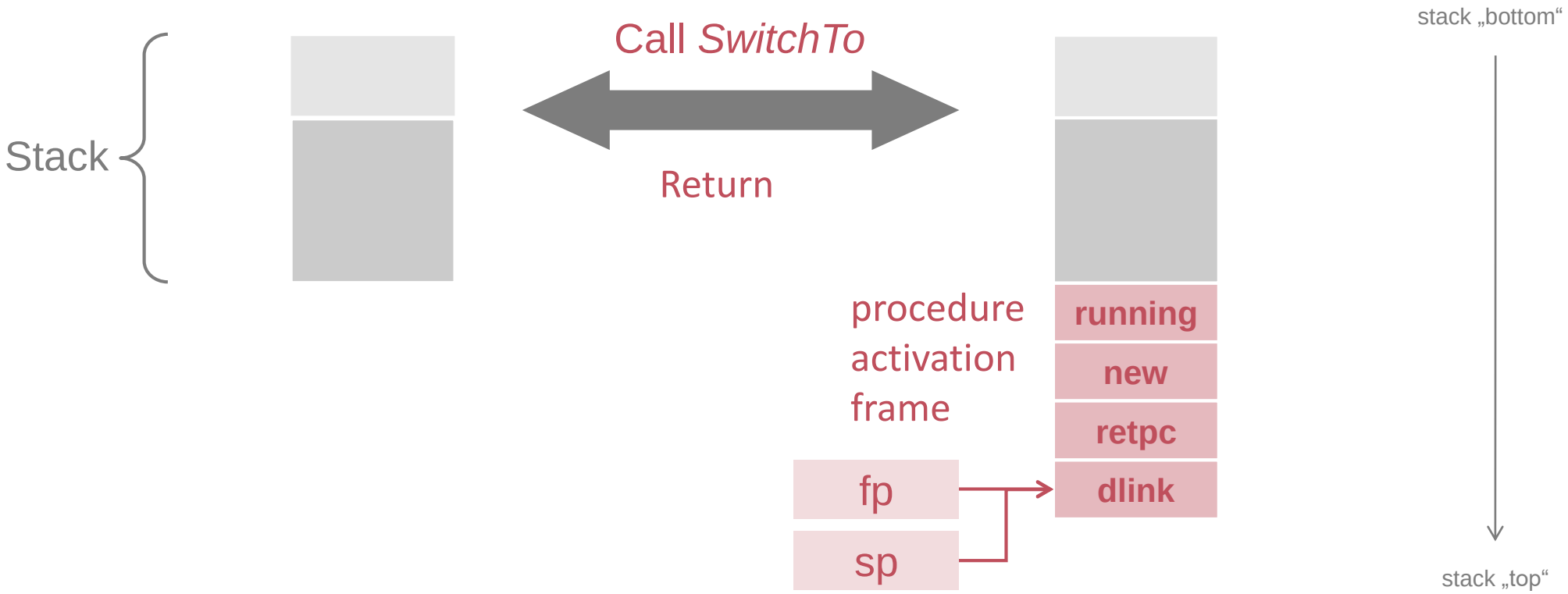


Context Switch

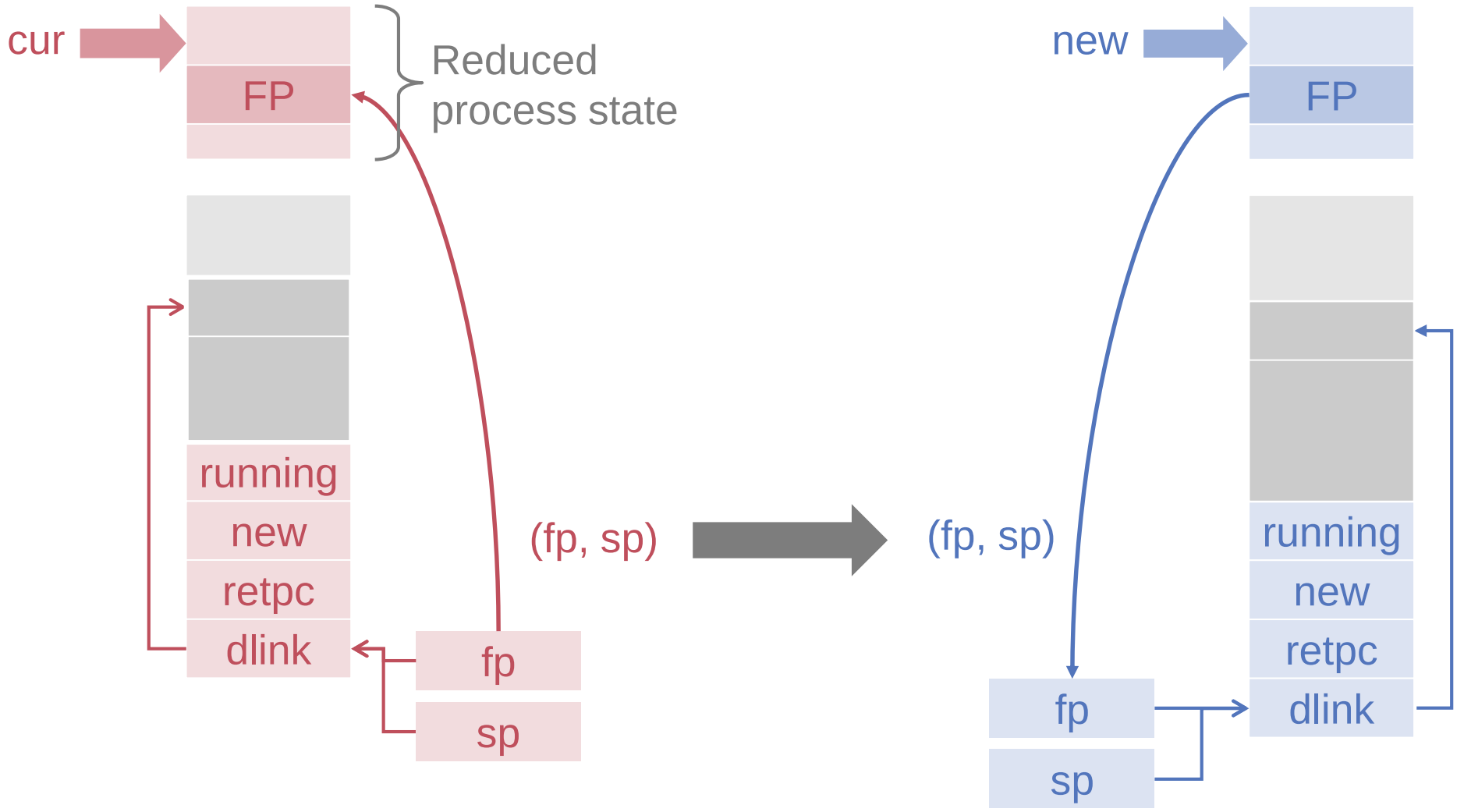
- Synchronous
  - Explicit
    - Terminate
    - *Yield*
  - Implicit
    - Awaiting condition
    - Mutual exclusion
- Asynchronous
  - Preemption
    - Priority handling
    - Timeslicing

Synchronous Context Switch (1)

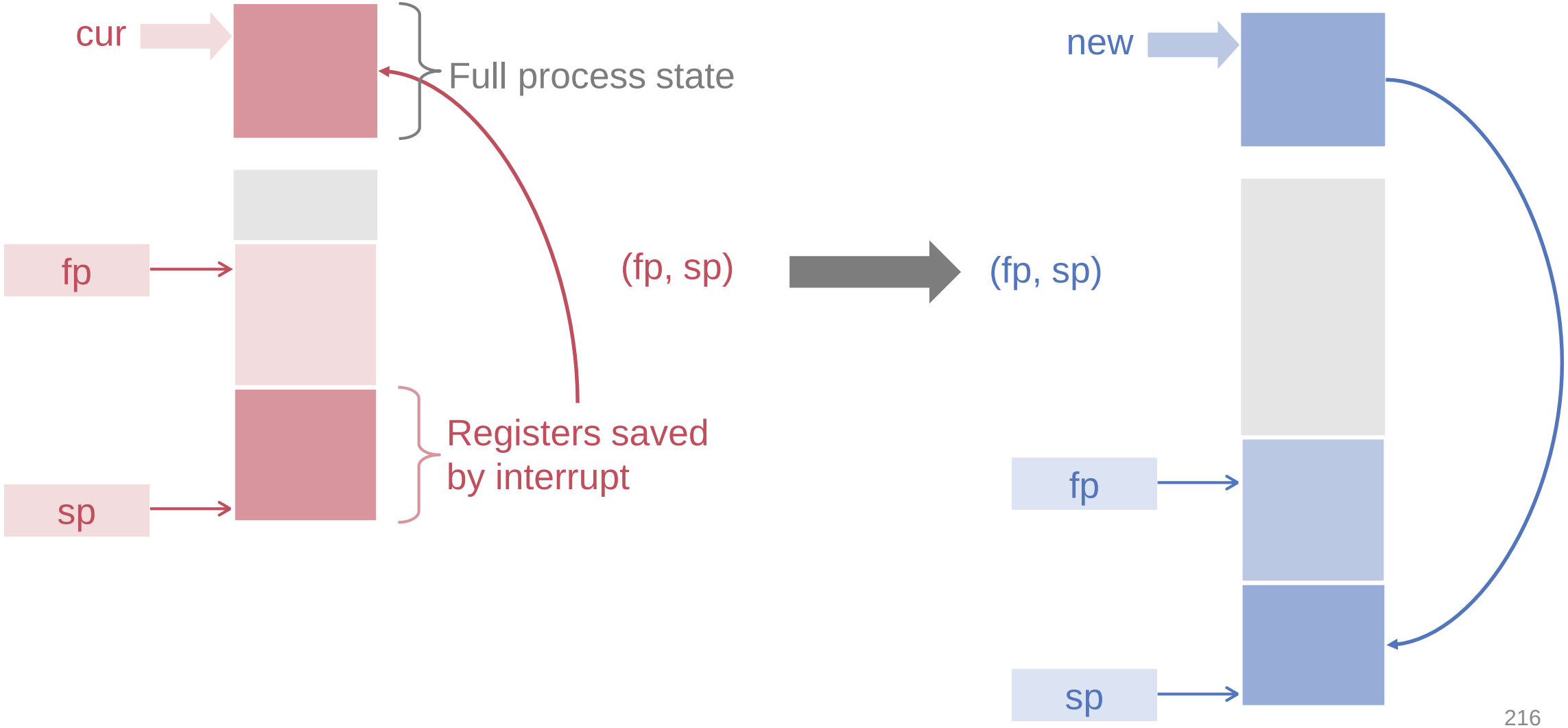
```
PROCEDURE SwitchTo (VAR running: Process; new: Process);
```



Synchronous Context Switch (2)



Asynchronous Context Switch



Stack Layout after Interrupt

```
72  .GS
68  .FS
64  .DS
60  .ES ; or haltcode
56  .SS
52  .ESP    ; or haltcode
48  .EFLAGS
44  .CS
40  .EIP
36  .EBP   ; err pushed by processor or glue code, exchanged by glue code popped by POP EBP
32  .INT   ; int pushed by glue code, popped by POP EBP
28  .EAX
24  .ECX
20  .EDX
16  .EBX
12  .ESP0
08  .ERR   ; (was: EBP, exchanged by glue code)
04  .ESI
00  .EDI
```

ESP0

state: State

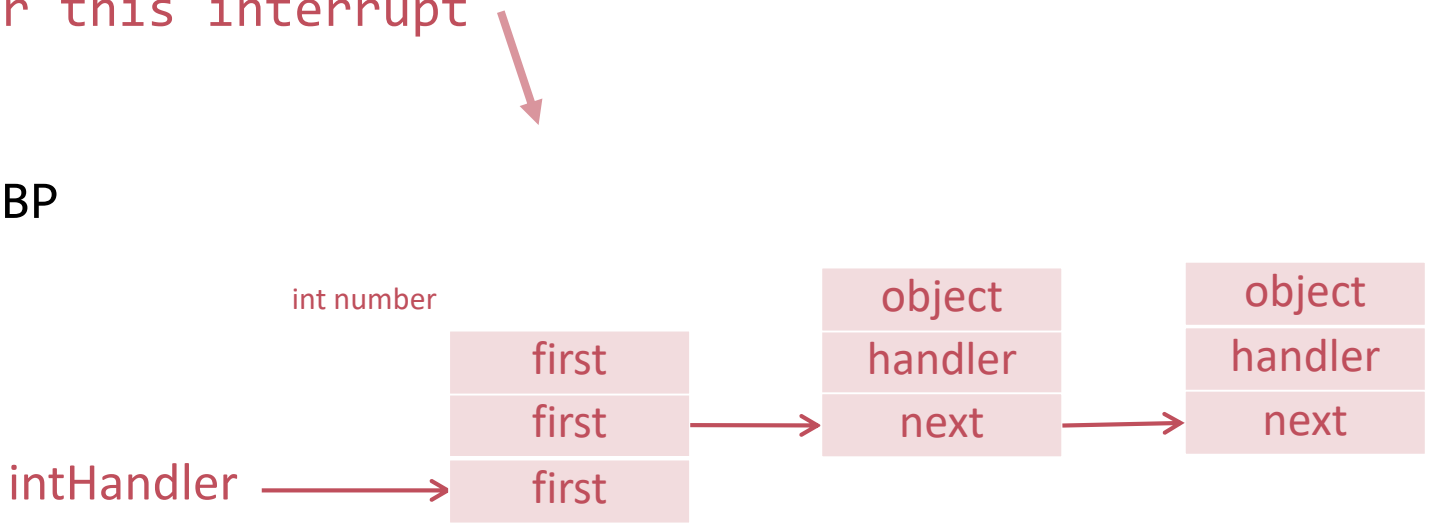
via interrupt/ IRETD

via Glue Code

via PUSHAD/ POPAD

First Level Interrupt Code (1)

```
PROCEDURE Interrupt;
CODE {SYSTEM.i386}
; called by interrupt handler (= glue code)
   PUSHAD ; save all registers (EBP = error code)
   LEA EBP, [ESP+36] ; procedure link
   MOV EBX, [ESP+32] ; EBX = int number
   LEA EAX, intHandler
   MOV EAX, [EAX][EBX*4]
   … ; now call all handlers for this interrupt
   POPAD ; now EBP = error code
   POP EBP ; now EBP = INT
   POP EBP ; now EBP = caller EBP
   IRETD
END Interrupt;
```



Switch Code (1)

```
PROCEDURE Switch (VAR cur: Process; new: Process);
BEGIN
  cur.state.SP := SYSTEM.GETREG(SP);
  cur.state.FP := SYSTEM.GETREG(FP);
  cur := new;
  IF ~cur.preempted then (* return from call *)
    SYSTEM.PUTREG(SP, cur.state.SP);
    SYSTEM.PUTREG(FP, cur.state.FP)
    Release(Objects);
  ELSE (* return from interrupt *)
    cur.preempted := FALSE;
    SYSTEM.PUTREG(SP, cur.state.SP);
    PushState(cur.state.EFLAGS, cur.state.CS,
      cur.state.EIP, cur.state.EAX, cur.state.ECX,
      cur.state.EDX, cur.state.EBX, 0,
      cur.state.EBP, cur.state.ESI, cur.state.EDI
    );
    Release(Objects);
    JumpState
  END
END Switch;
```

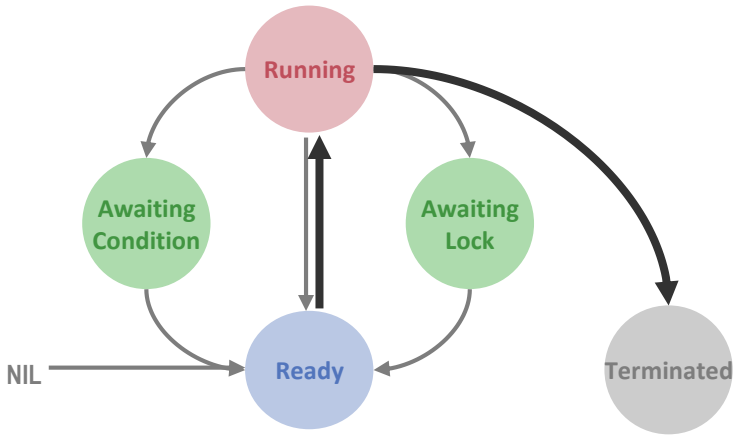
synchronous → synchronous/ asynchronous

Switch Code (2)

```
(* parameters are pushed on the stack in this inline procedure by the caller *)

PROCEDURE -PushState(EFLAGS: SET;
  CS, EIP, EAX, ECX, EDX, EBX,
  ESP, EBP, ESI, EDI: LONGINT);
CODE {SYSTEM.i386}
      (* to omit call protocol, parameters stay on stack *)
END PushState;


PROCEDURE -JumpState;
CODE {SYSTEM.i386}
  POPAD
  IRETD
END PopState;
```

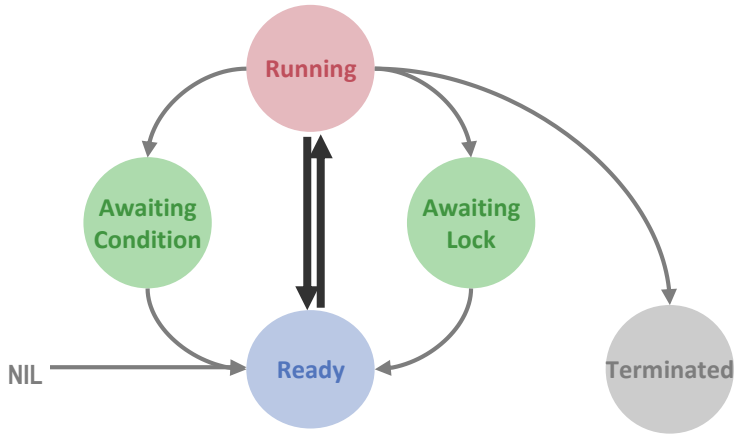
Example 1: Termination

```
PROCEDURE Terminate;
  VAR new: Process;
BEGIN
  Acquire(Objects);
  Select(new, MinPriority);
  Switch(running[ProcessorID()], new)
END Terminate;
```



Example 2: *Yield*

```
PROCEDURE Yield;
VAR id: INTEGER; new: Process;
BEGIN
  Acquire(Objects);
  id := ProcessorID();
  Select(new, running[id].priority);
  IF new # NIL THEN
    Enter(running[id]);
    Switch(running[id], new)
  ELSE
    Release(Objects)
  END
END Yield;
```



Idle Activity in Objects

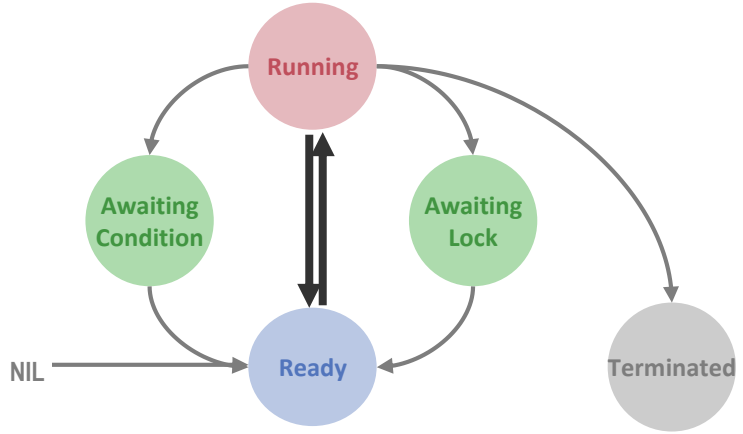
```
Idle = OBJECT
  BEGIN{ ACTIVE, SAFE, PRIORITY(PrioIdle)}
    LOOP
      REPEAT
         Machine.SpinHint
      UNTIL maxReady > MinPriority;
      Yield
    END
  END Idle;
```

Example: Timeslicing

```
PROCEDURE Timeslice (VAR state: ProcessorState);
VAR id: integer; new: Process;
BEGIN Acquire(Objects);
  id := ProcessorID();
  IF running[id].priority # Idle THEN
    Select(new, running[id].priority);
    IF new # NIL THEN
      running[id].preempted := true;
      CopyState(state, running[id].state);
      Enter(running[id]);
      running[id] := new;
      IF new.preempted then
          new.preempted := false;
          CopyState(new.state, state)
      ELSE
          SwitchToState(new, state)
      END
    END
  END;
  Release(Objects)
END Timeslice;
```

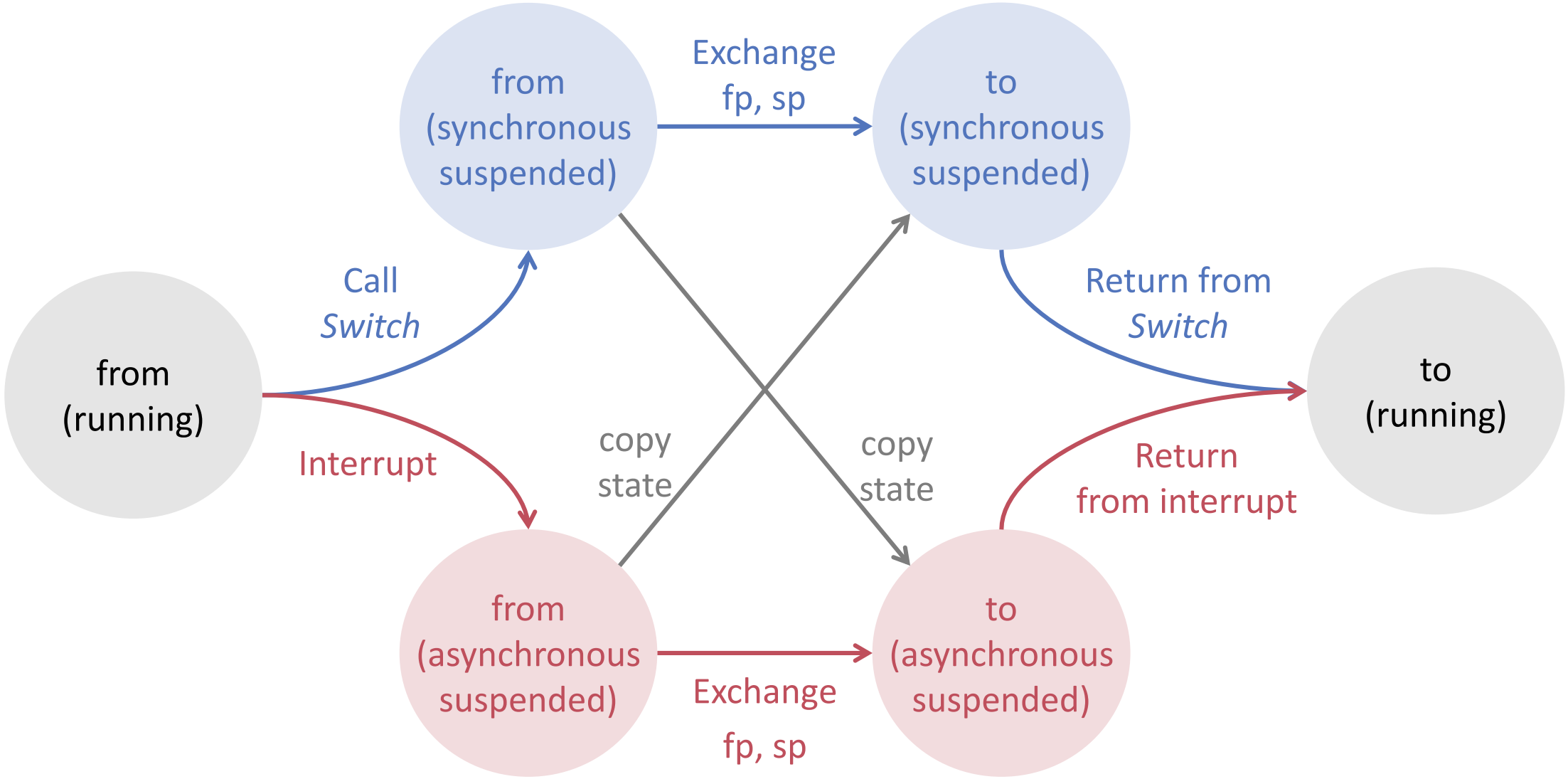
return from interrupt of new process

simulate return from procedure switch



asynchronous → synchronous/ asynchronous

Context Switching Scenarios

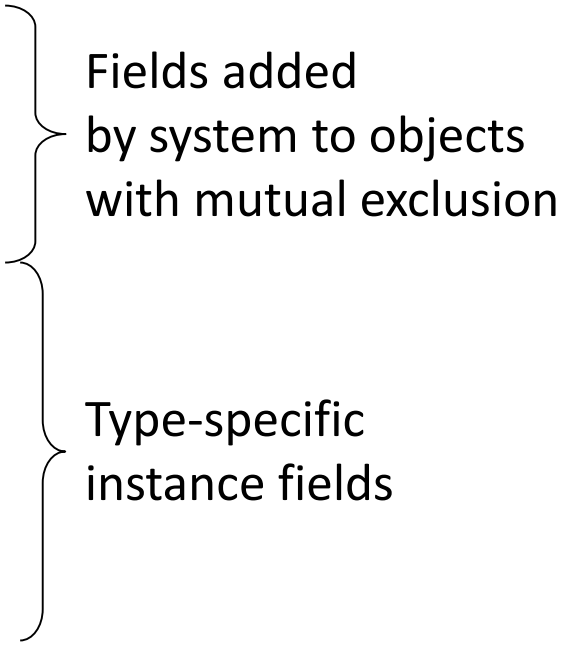


Synchronization

- Object locking
- Condition management

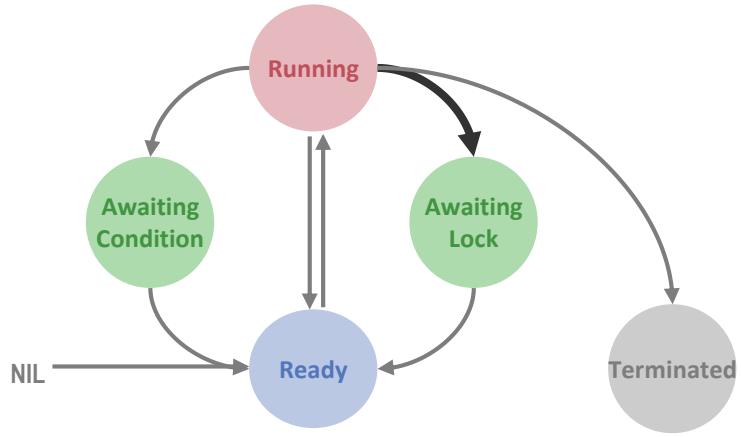
Object Descriptors

```
ObjectHeader = RECORD
  headerLock: BOOLEAN;
  lockedBy: Process;
  awaitingLock: ProcessQueue;
  awaitingCondition: ProcessQueue;

  …

END;
```



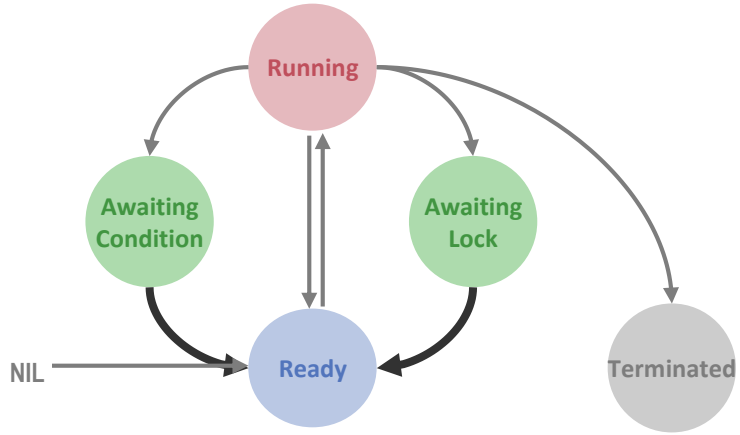
Object Locking

```
PROCEDURE Lock (obj: object);
  VAR r, new: Process;
BEGIN
   r := running[ProcessorID()];
   AcquireObject(obj.hdr.headerLock);
   IF obj.hdr.lockedBy = nil THEN
      obj.hdr.lockedBy := r;
      ReleaseObject(obj.hdr.headerLock);
   ELSE
      Acquire(Objects);
      Put(obj.hdr.awaitingLock, r);
      ReleaseObject(obj.hdr.headerLock);
      Select(new, MinPriority);
      SwitchTo(running[ProcessorID()], new)
   END
END Lock;
```



Object Unlocking

```
PROCEDURE Unlock (obj: object);
VAR c: Process;
BEGIN
   c := FindCondition(obj.hdr.awaitingCondition)
   AcquireObject(obj.hdr.headerLock);
   IF c = NIL THEN
      Get(obj.hdr.awaitingLock, c);
   END;
   obj.hdr.lockedBy := c
   ReleaseObject(obj.hdr.headerLock);
   IF c # NIL THEN
      Acquire(Objects); Enter(c); Release(Objects)
   END;
END Unlock;
```



Atomic Lock Transfer
Eggshell-Model !

Condition Management

Condition Type

```
TYPE
  Condition = PROCEDURE(fp: ADDRESS): BOOLEAN;
```

Condition Boxing

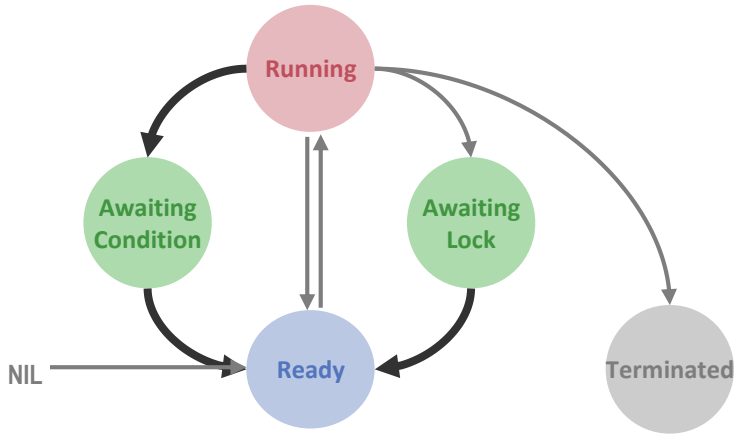
```
PROEDURE $Condition(fp: ADDRESS): BOOLEAN;
BEGIN
  RETURN ”condition from await statement”
END $Condition;
```

Await Code

```
IF ~$Condition(FP) THEN
  Await($Condition, FP, SELF)
END
```

AWAIT Code

```
PROCEDURE Await (condition: Condition; fp: address; obj: object);
VAR r, t, new: Process;
BEGIN
   AcquireObject(obj.hdr.headerLock);
   c := FindCondition(obj.hdr.awaitingCondition);
   IF c = NIL THEN
     Get(obj.hdr.awaitingLock, c);
   END;
   obj.hdr.lockedBy := c
   Acquire(Objects);
   IF c # NIL THEN Enter(c) END;
   r := running[ProcessorID()];
   r.condition := condition;
   r.conditionFP := fp;
   Put(obj.hdr.awaitingCondition, r);
   ReleaseObject(obj.hdr.headerLock);
   Select(new, MinPriority);
   SwitchTo(running[ProcessorID()], new)
END Await;
```

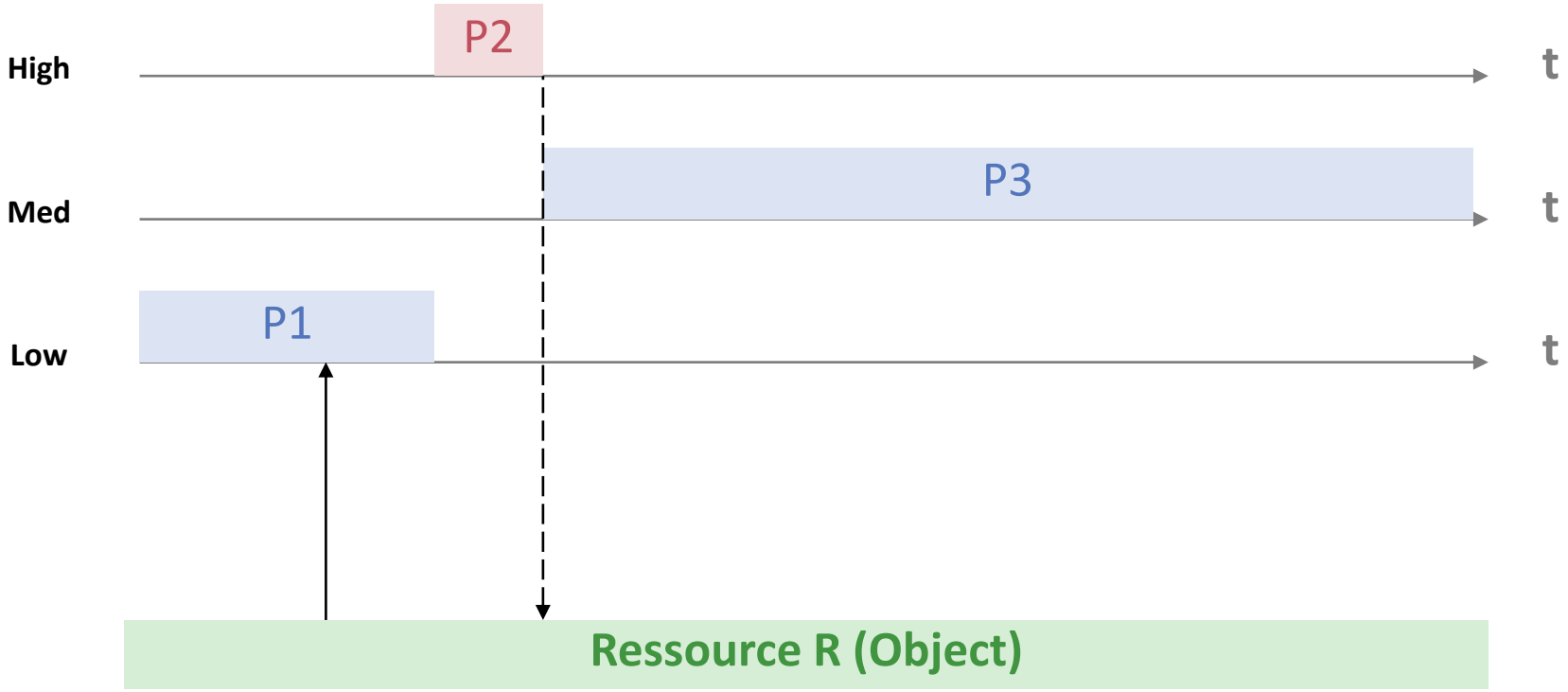


Condition Evaluation

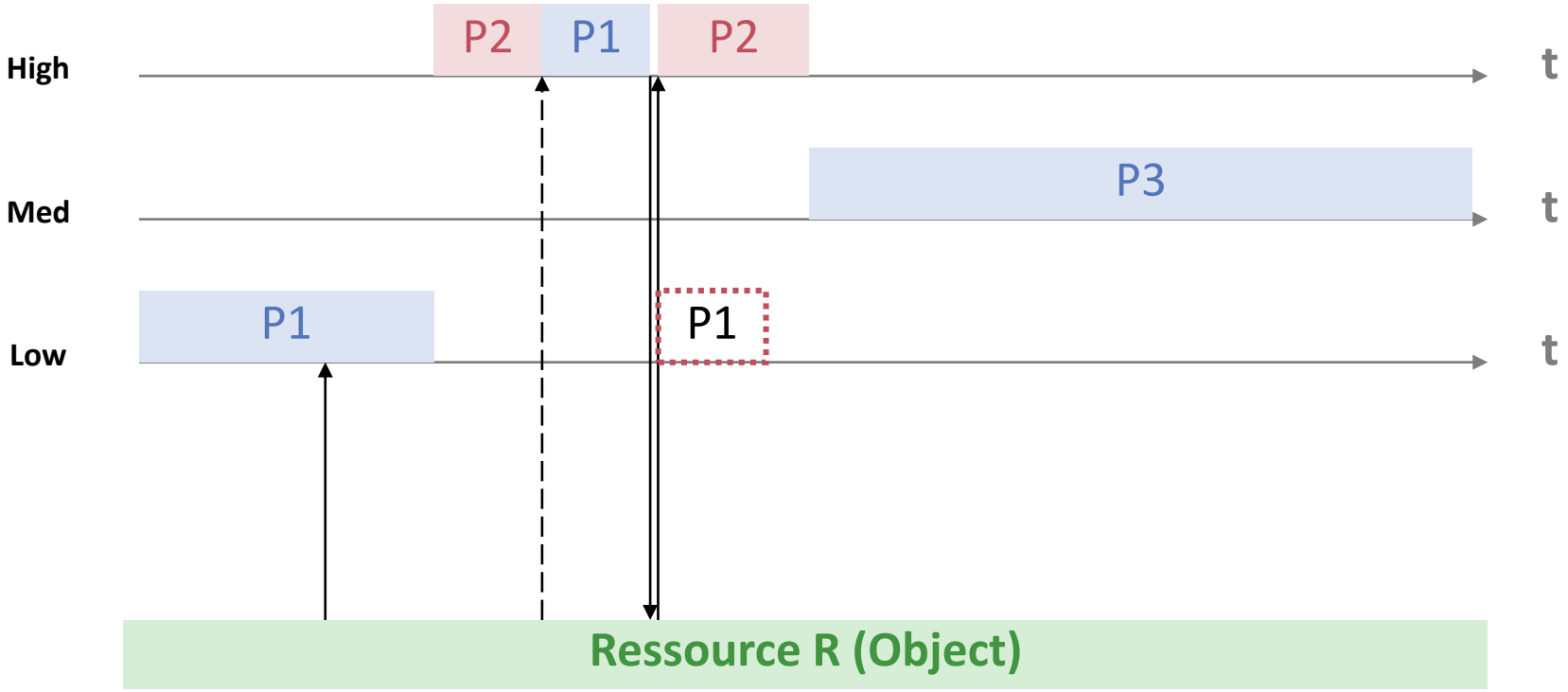
```
PROCEDURE FindCondition (VAR q: ProcessQueue): Process;
VAR first, c: Process;
BEGIN
   Get(q, first);
   IF first.condition(first.conditionFP) THEN
      RETURN f
   END;
   Put(q, first);
   WHILE q.head # first DO
      Get(q, c);
      IF c.condition(c.conditionFP) THEN
         RETURN c
      END;
      Put(q, c)
   END;
   RETURN NIL
END FindCondition;
```

Priority Inversion

Inversion caused by an immediately shared resource

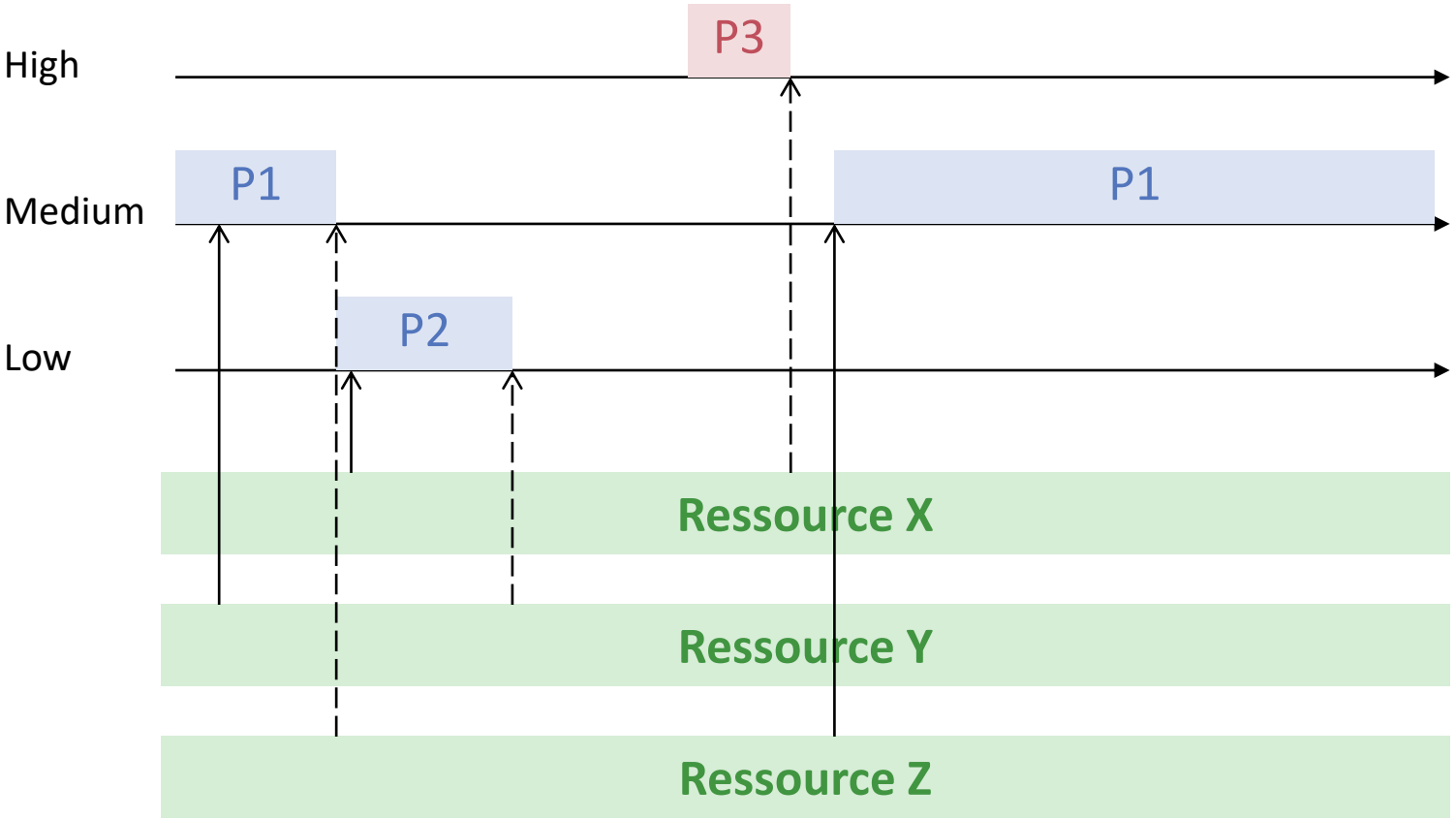


Priority Inheritance

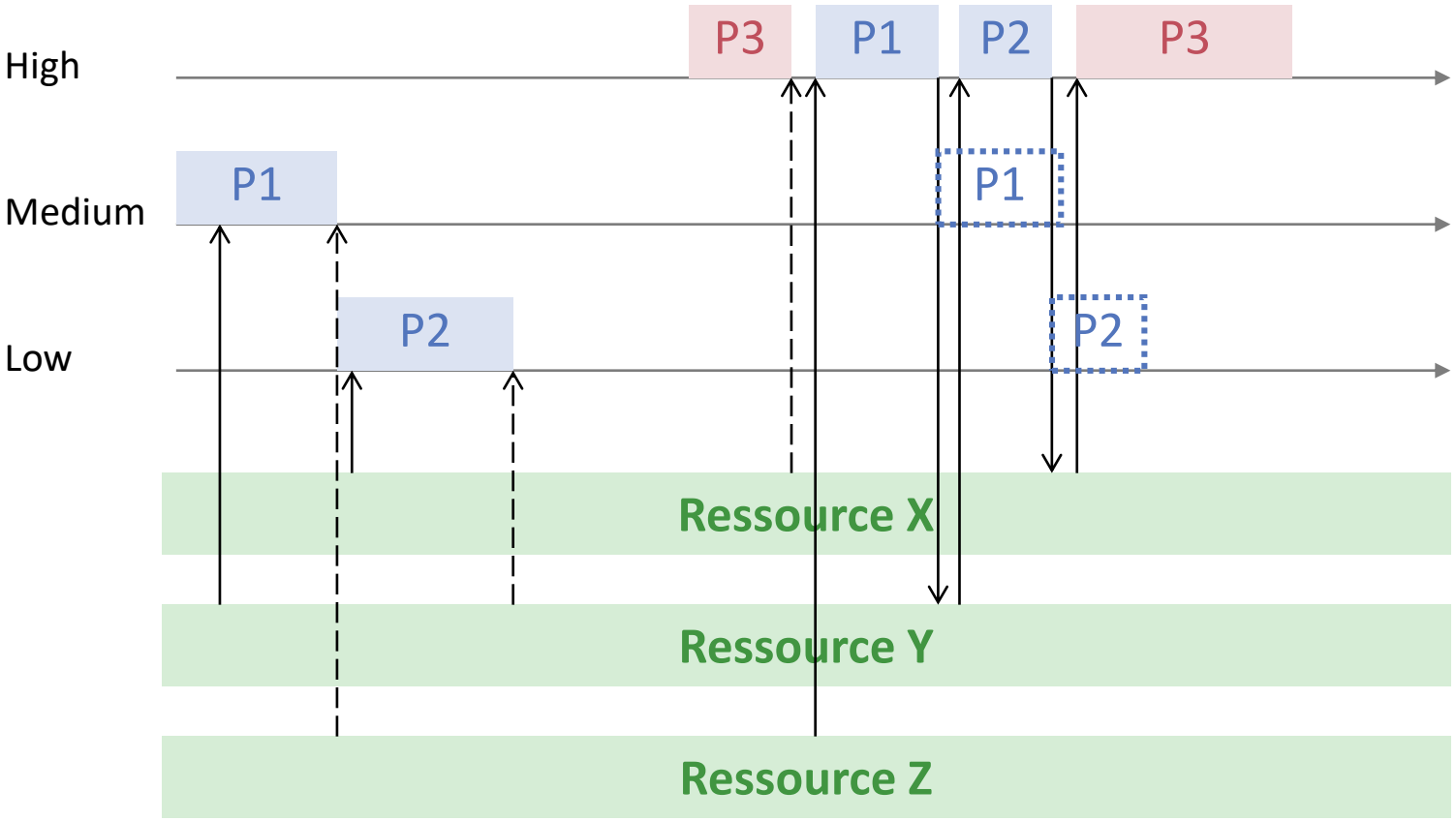


Priority Inversion

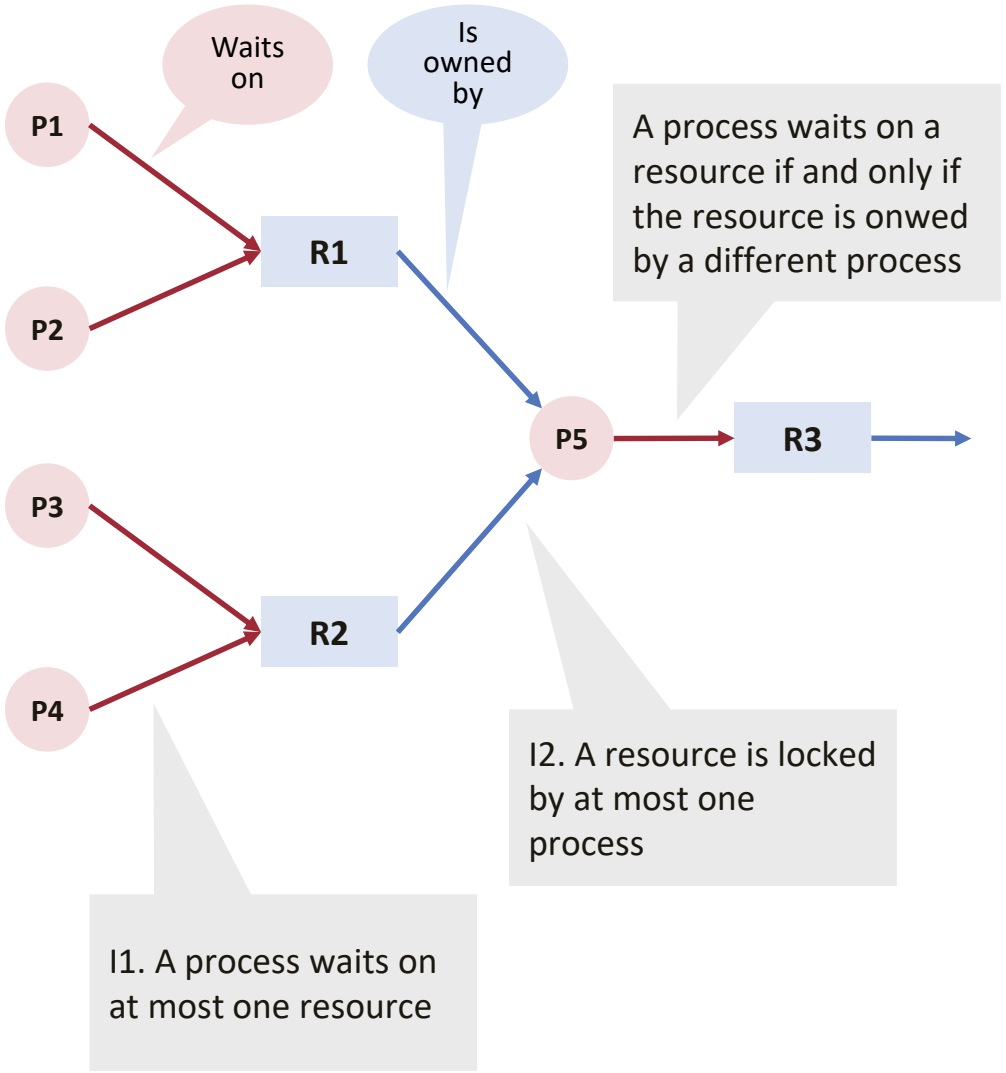
Inversion caused by not immediately shared resource



Priority Inheritance



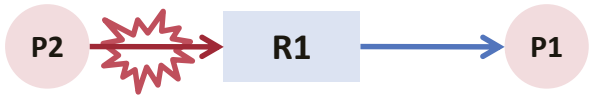
Invariants


Waits on
Is owned by
P1
P2
P3
P4
R1
R2
P5
R3
A process waits on a resource if and only if the resource is onwed by a different process
I2. A resource is locked by at most one process
I1. A process waits on at most one resource

Possible Transitions

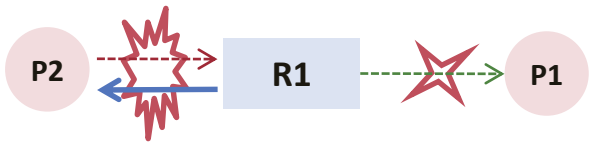
process
acquires and receives
free resource

Process tries to
acquire locked
resource

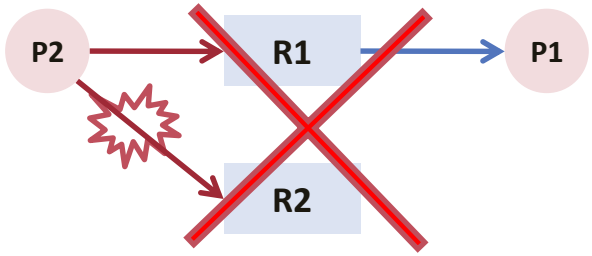


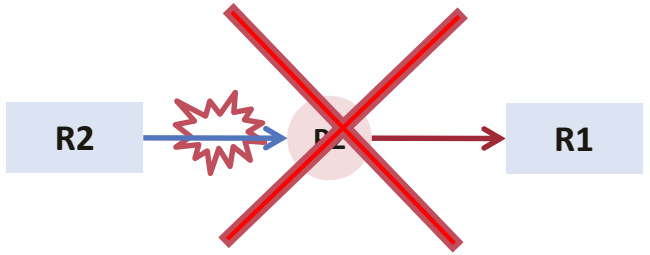
process
releases resource

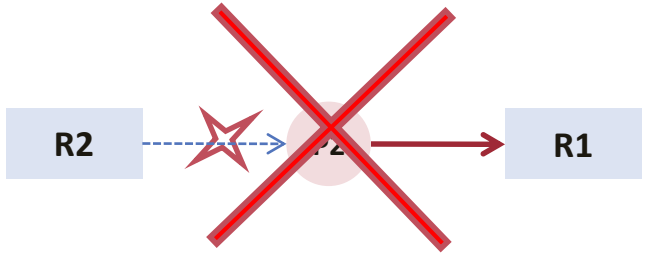
waiting process
receives resource



waiting processes can
neither acquire
nor release resources

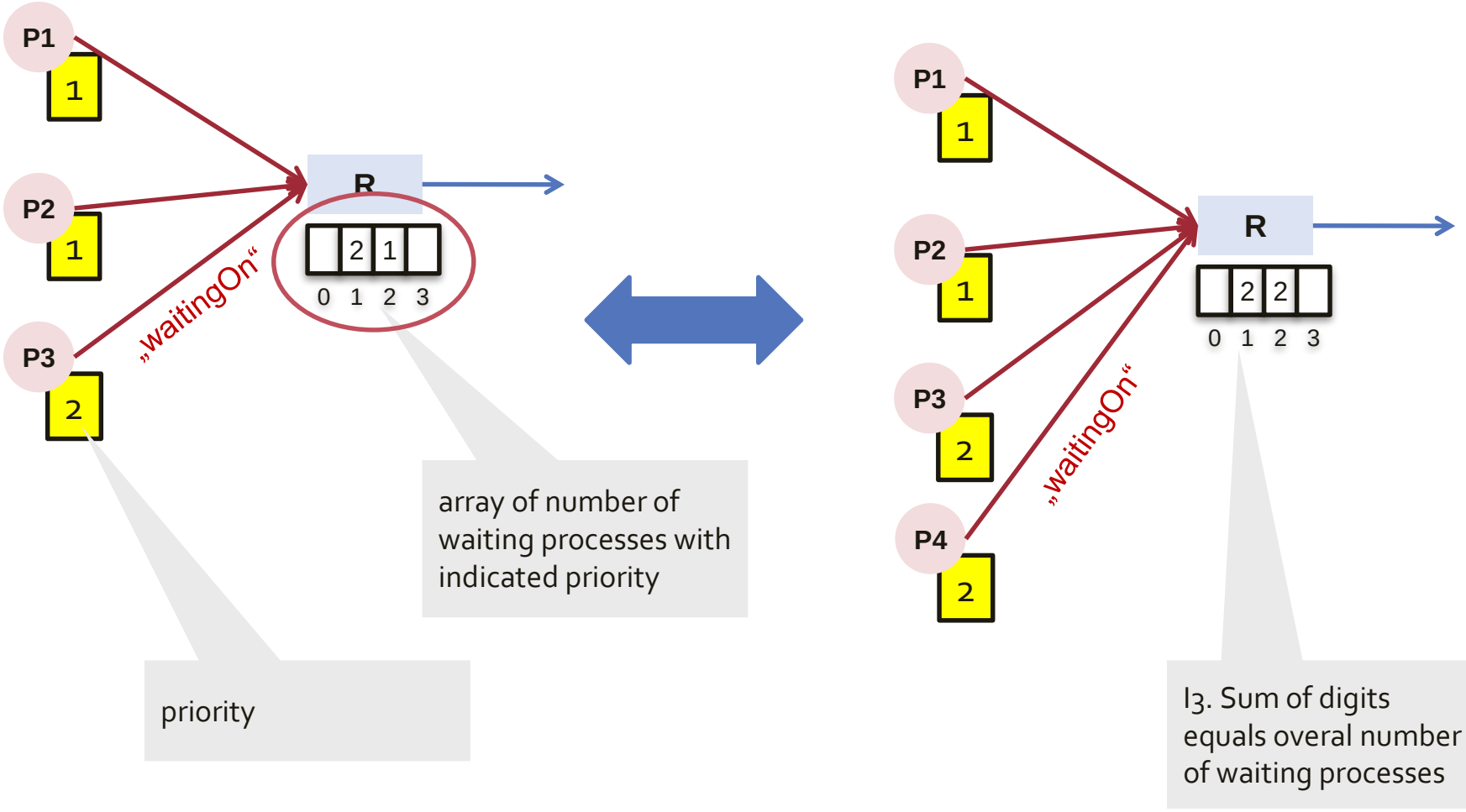






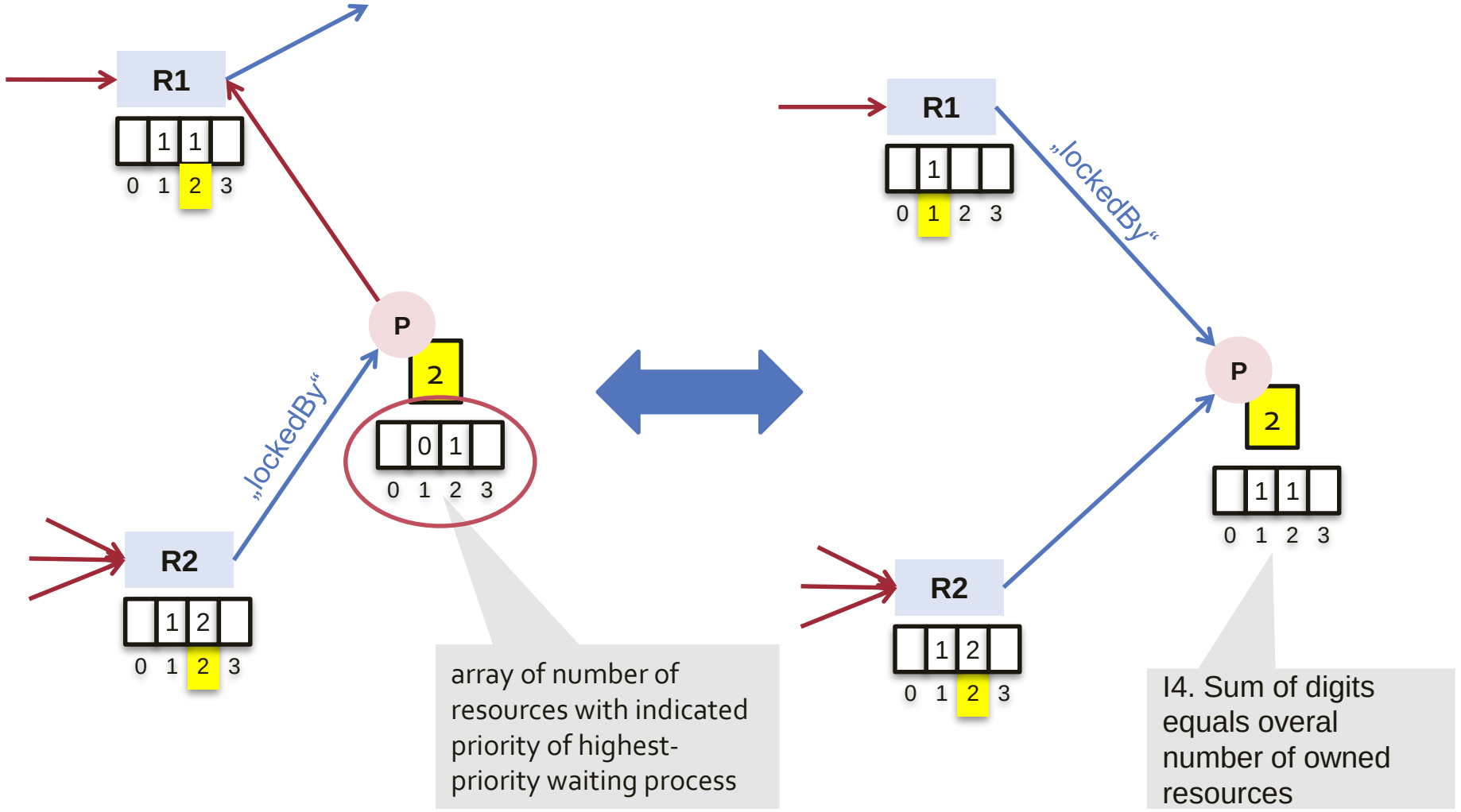
WaitingPriorities List

(Re-)Setting the „waitingOn“ pointer



LockedByPriorities List

(Re-)Setting the „lockedBy“ pointer



Further Rules / Invariants

A process that releases a resource does not simultaneously wait on a resource.

I5. For each resource, the number of waiting processes and their priorities are available at all times.

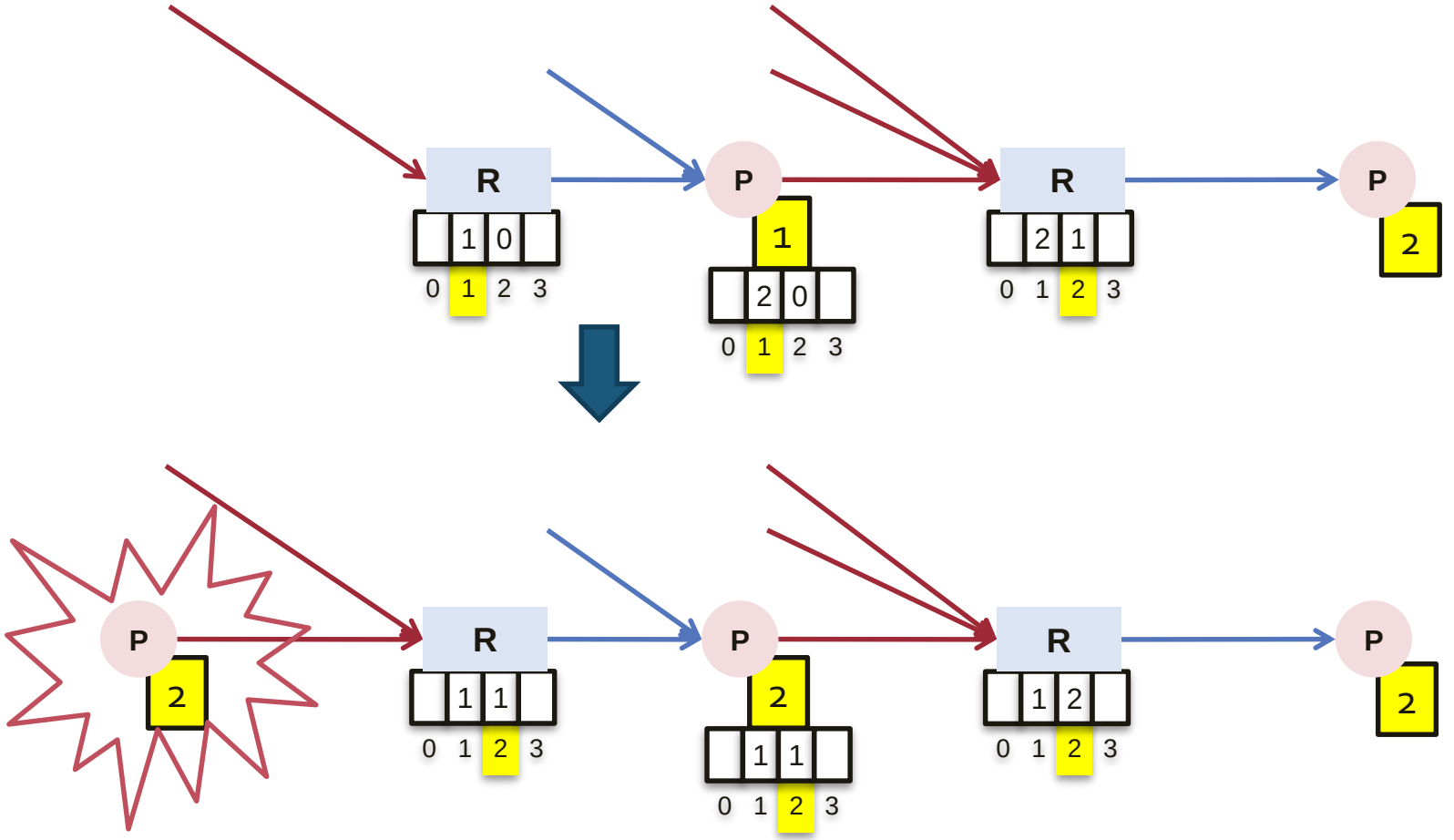
- invariant held during (re-)setting the „waitingOn“ pointer

I6. For each process, the highest priority of indirectly waiting processes is known at all times

- invariant held during (re-) setting the „lockedBy“ pointer

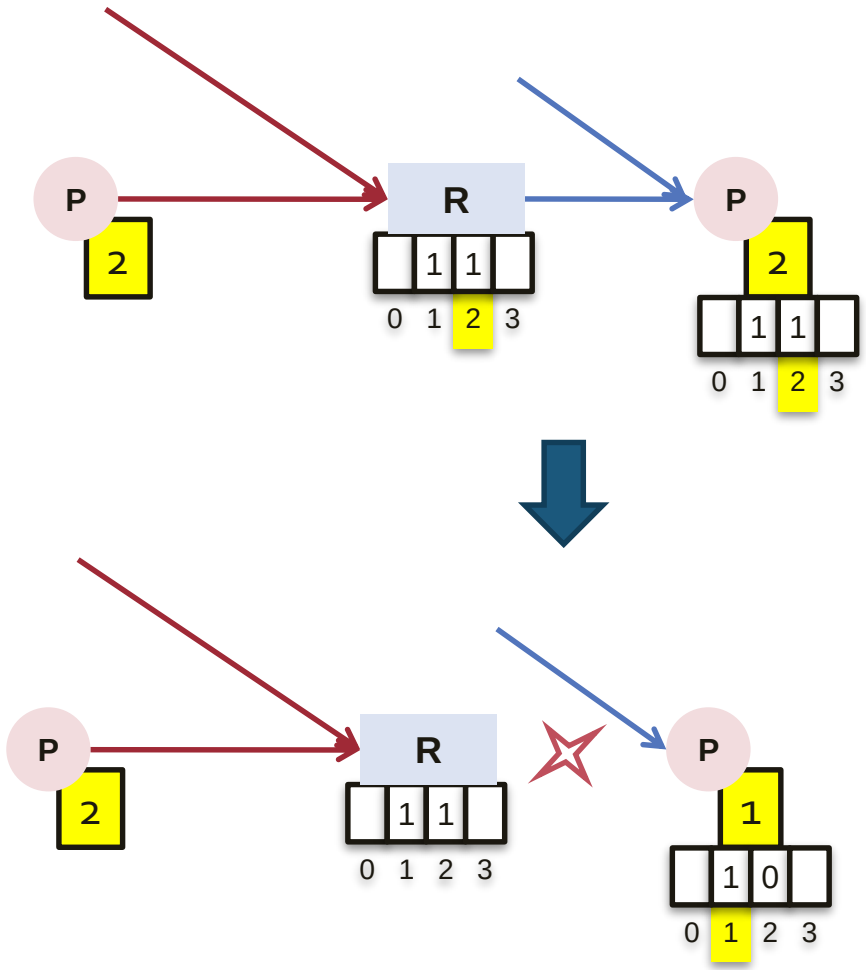
Handling Priority Inversion

(Setting „WaitingOn “)



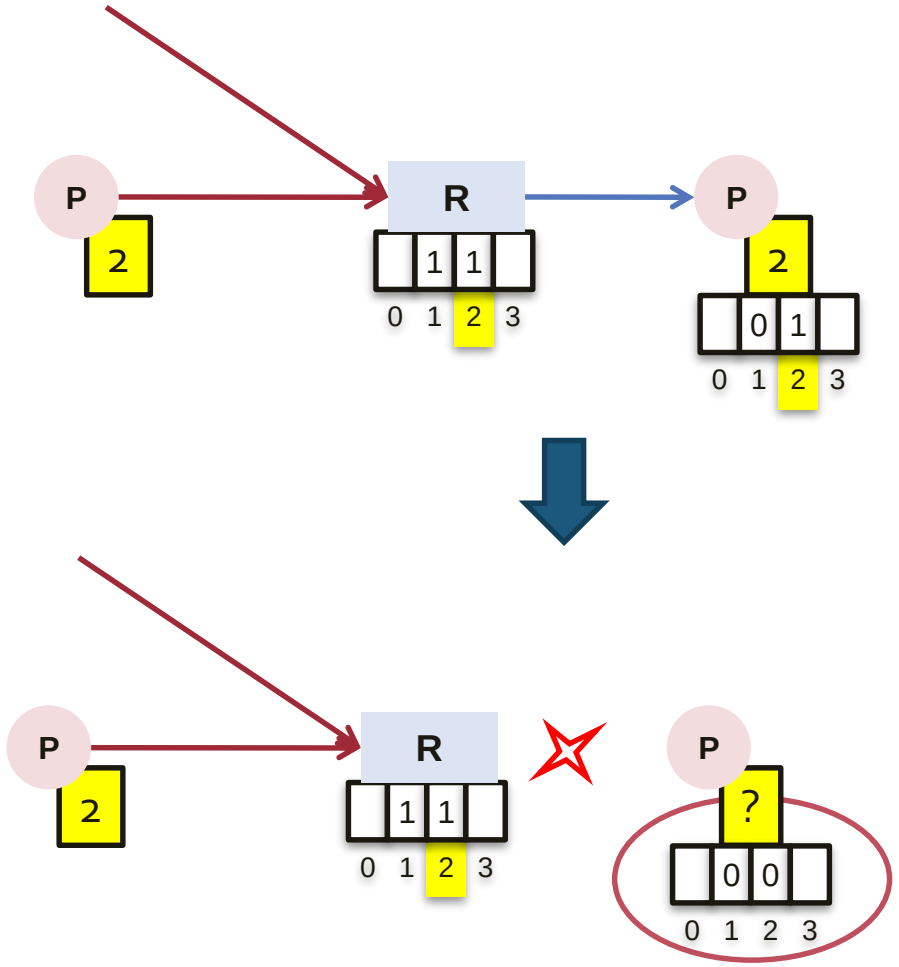
Handling Priority Inversion

(Resetting „LockedBy“)



Handling Priority Inversion

Resetting „LockedBy“: how to preserve initial priorities?



Handling Priority Inversion

Resource and Process Initialization

- Resource is allocated with virtual idle process
- Process is allocated with virtual resource of process initial priority