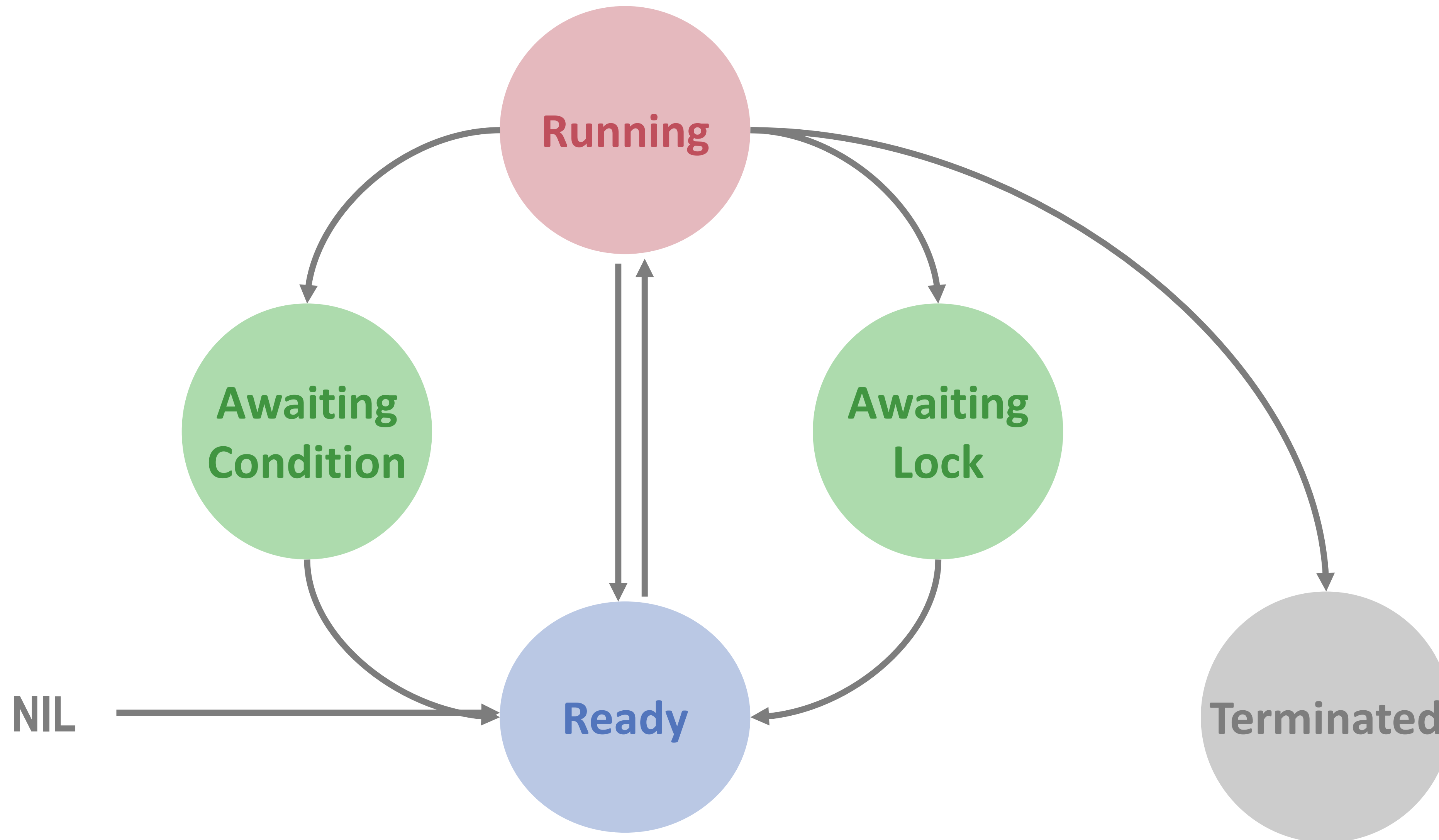
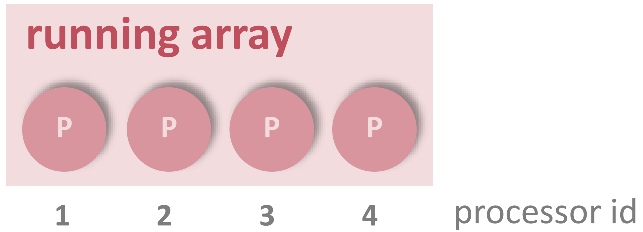
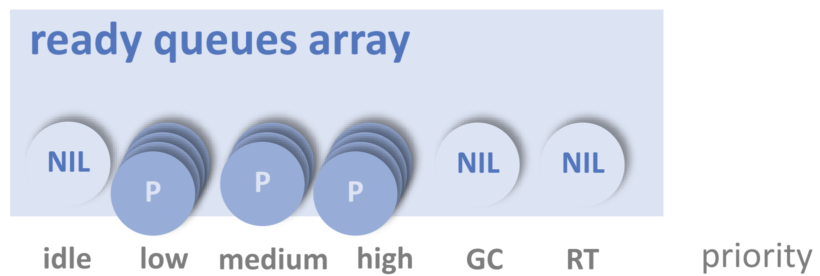
2.3. ACTIVITY MANAGEMENT

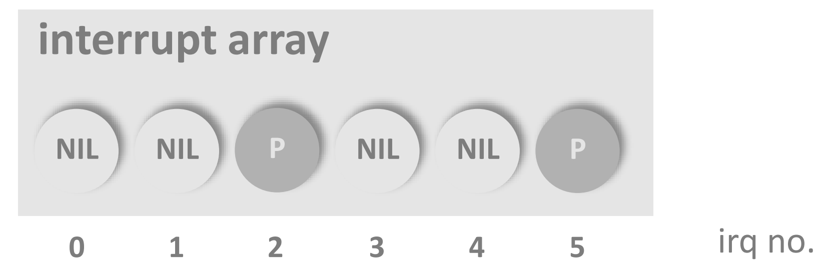
Life Cycle of Activities



Runtime Data Structures





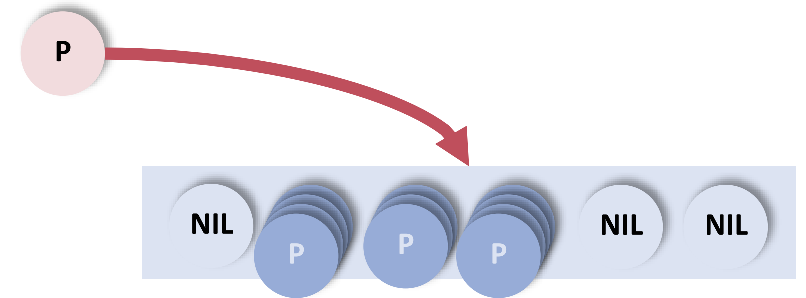


Process Descriptors

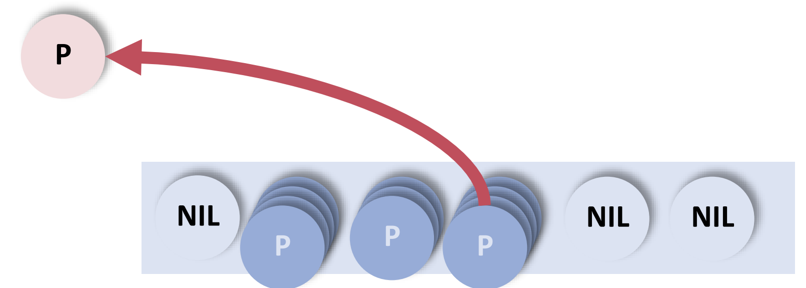
```
TYPE
  Process = OBJECT
    …
    stack: Stack;
    state: ProcessState;
    preempted: BOOLEAN;
    condition: PROCEDURE (slink: ADDRESS);
    conditionFP: ADDRESS;
    priority: INTEGER;
    obj: OBJECT;
    next: Process
  END Process;

  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;
```

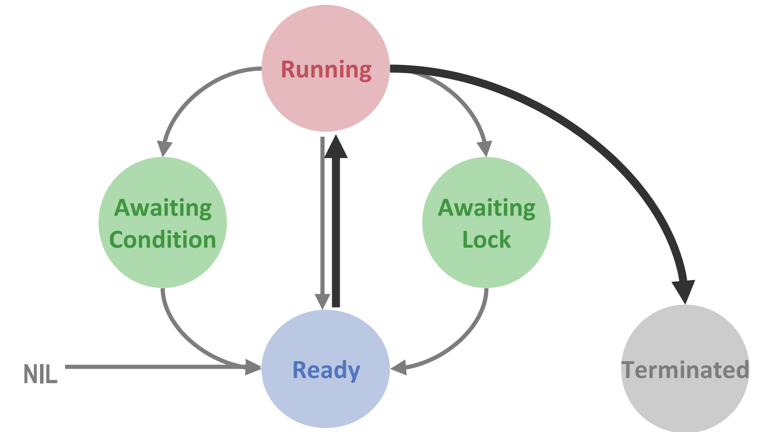



```
PROCEDURE Select (VAR new: Process; priority: integer);
BEGIN
  LOOP
    IF maxReady < priority THEN new := nil; EXIT END;
    Get(ready[maxReady], new);
    IF(new # NIL) OR (maxReady = MinPriority) THEN
      EXIT
    END;
    maxReady := maxReady-1
  END
END Select;
```



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;
```

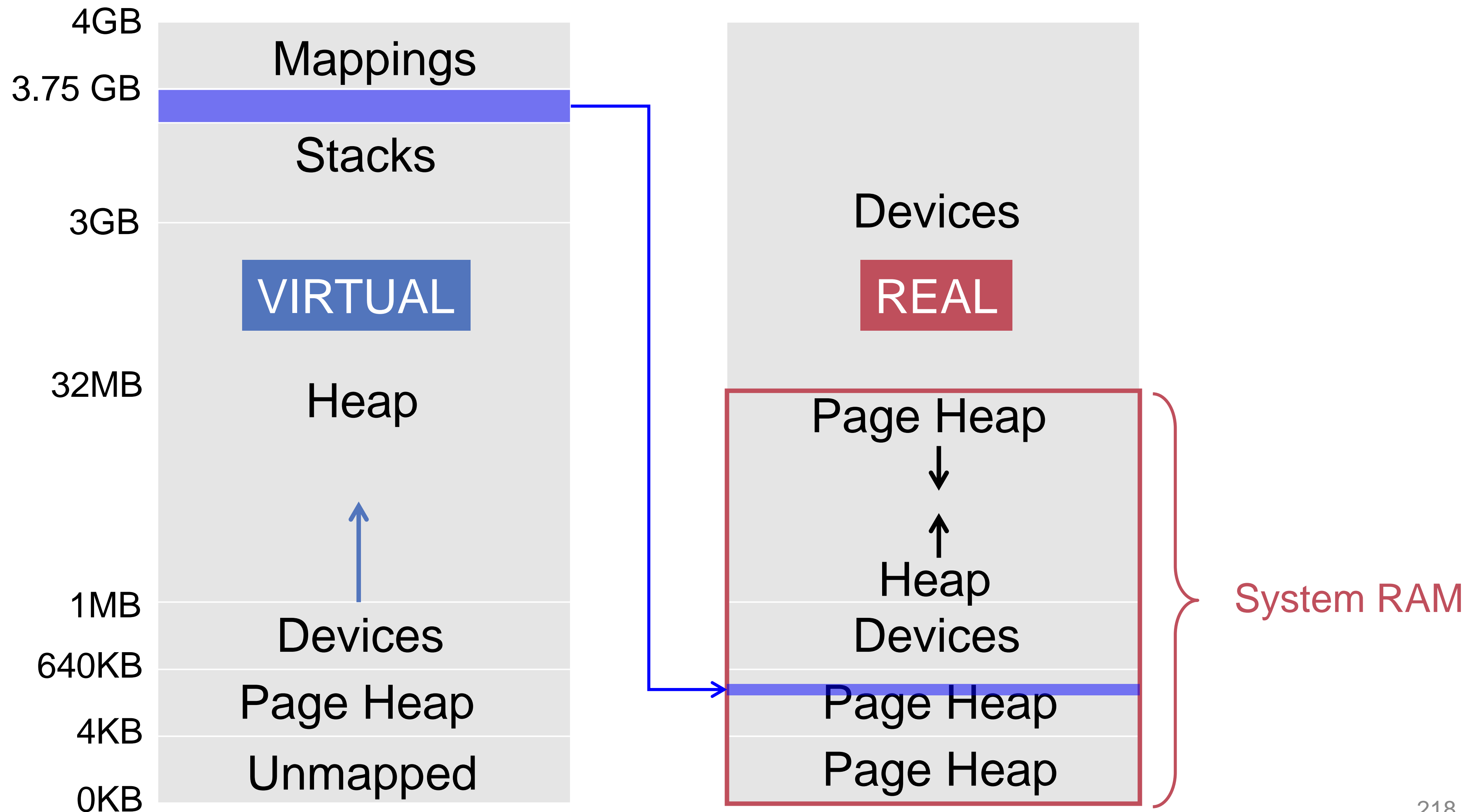


Stack Management

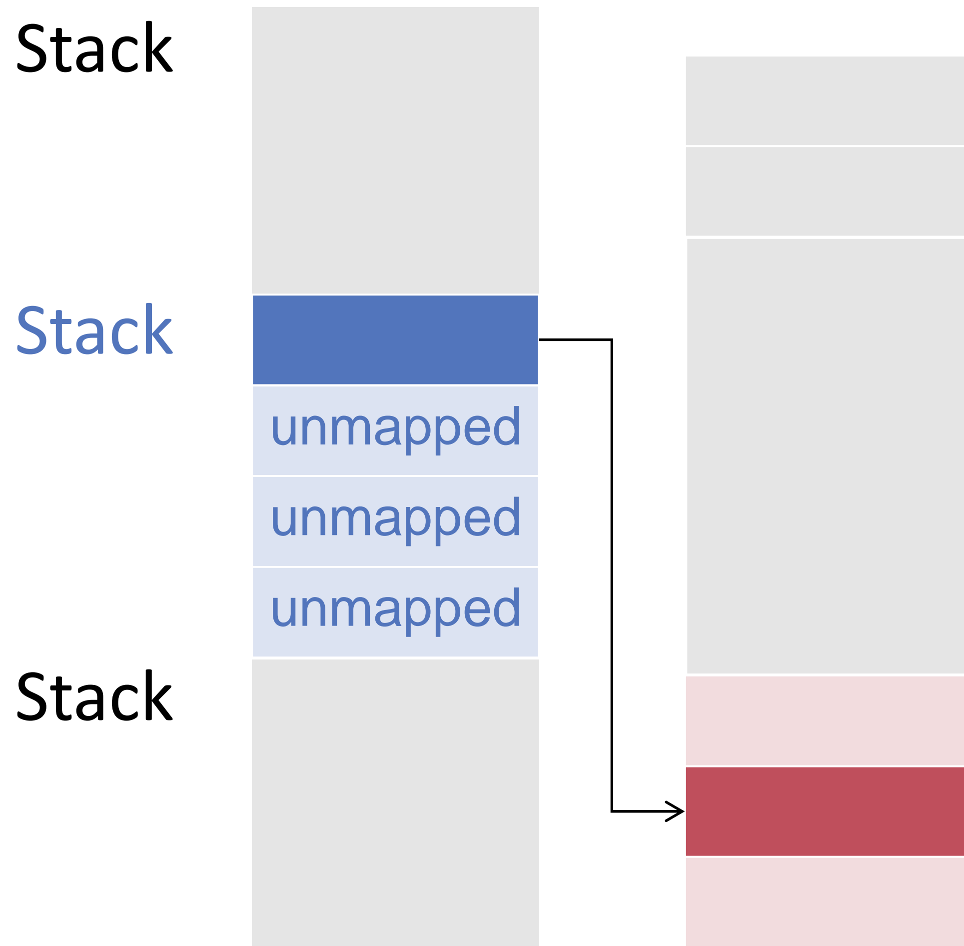
- Use virtual adressing
- Allocate stack in page units
- Use page fault for detecting stack overflow
- Deallocate stack via garbage collector
  (in process finalizer)

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

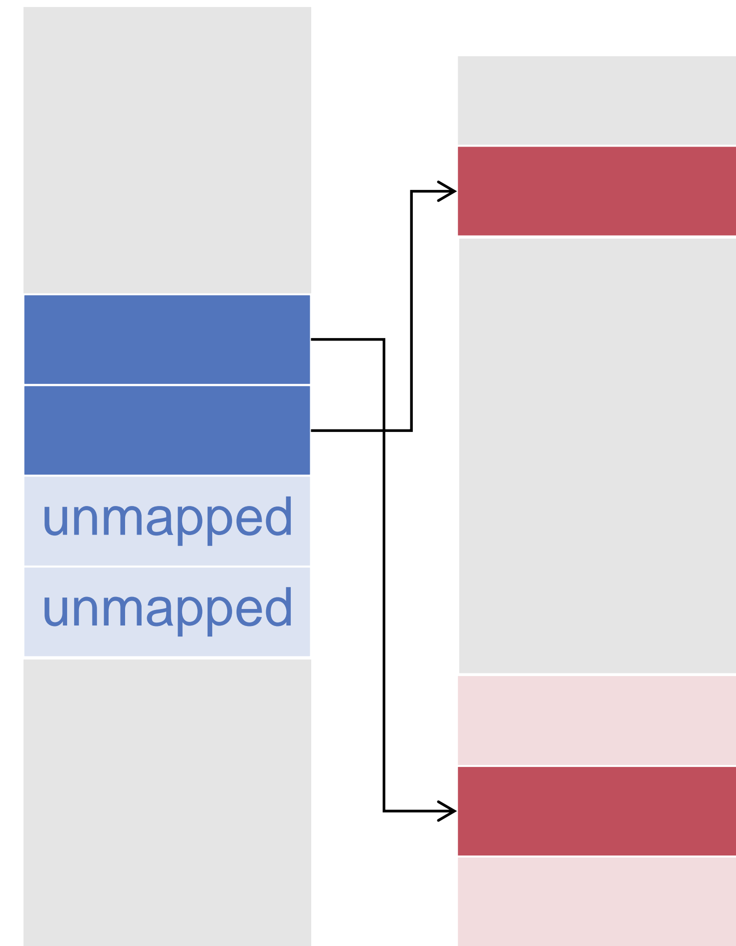
Memory Layout



Stack Allocation



initially



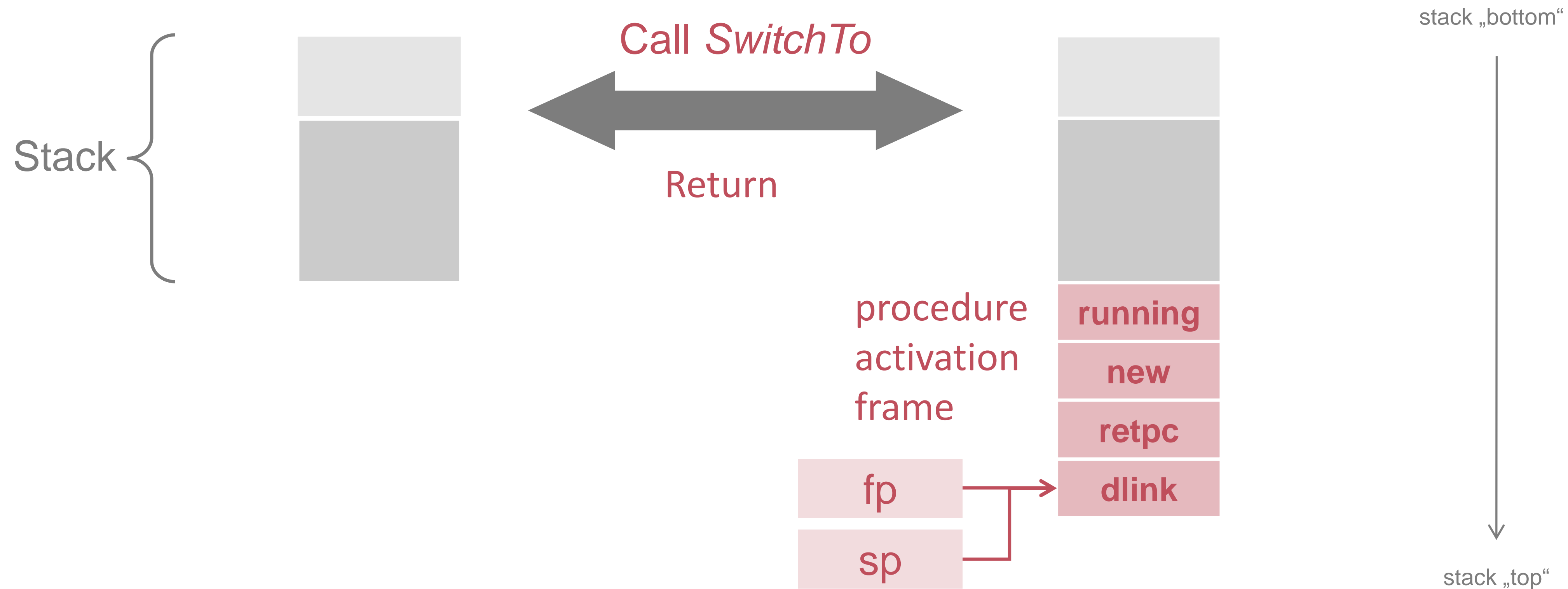
after extension

Context Switch

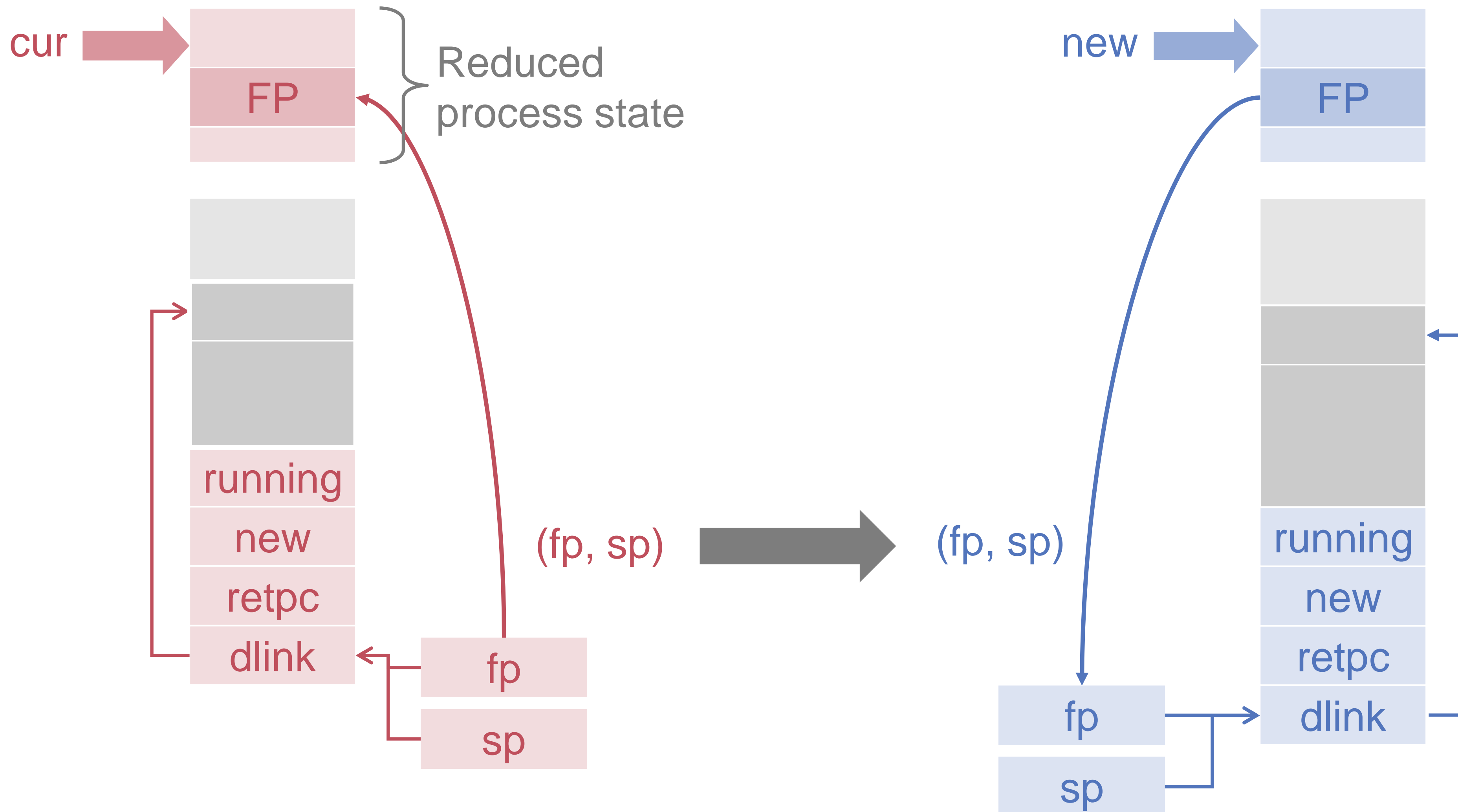
- Synchronous
  - Explicit
    - Terminate
    - *Yield*
  - Implicit
    - Awaiting condition
    - Mutual exclusion
- Asynchronous
  - Preemption
    - Priority handling
    - Timeslicing

Synchronous Context Switch (1)

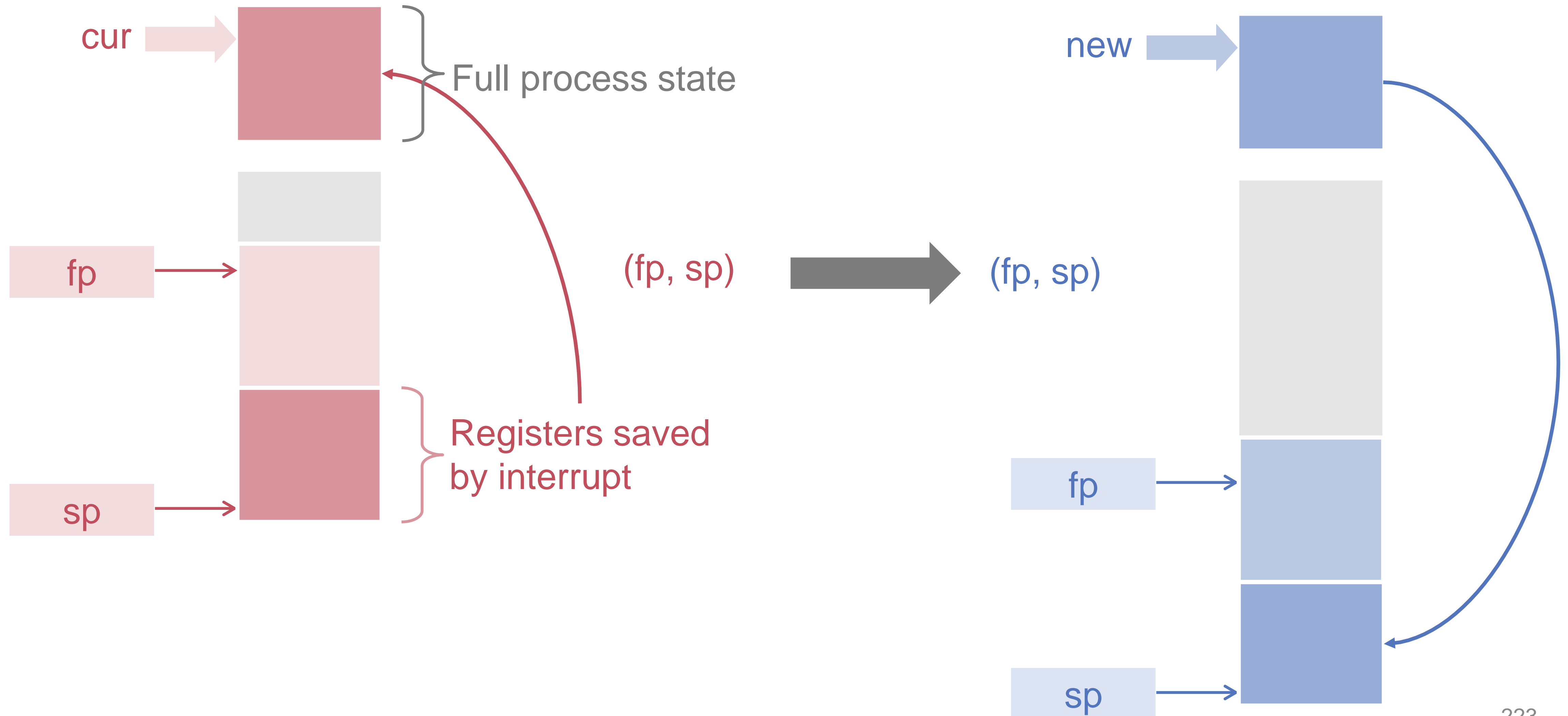
```
PROCEDURE SwitchTo (VAR running: Process; new: Process);
```



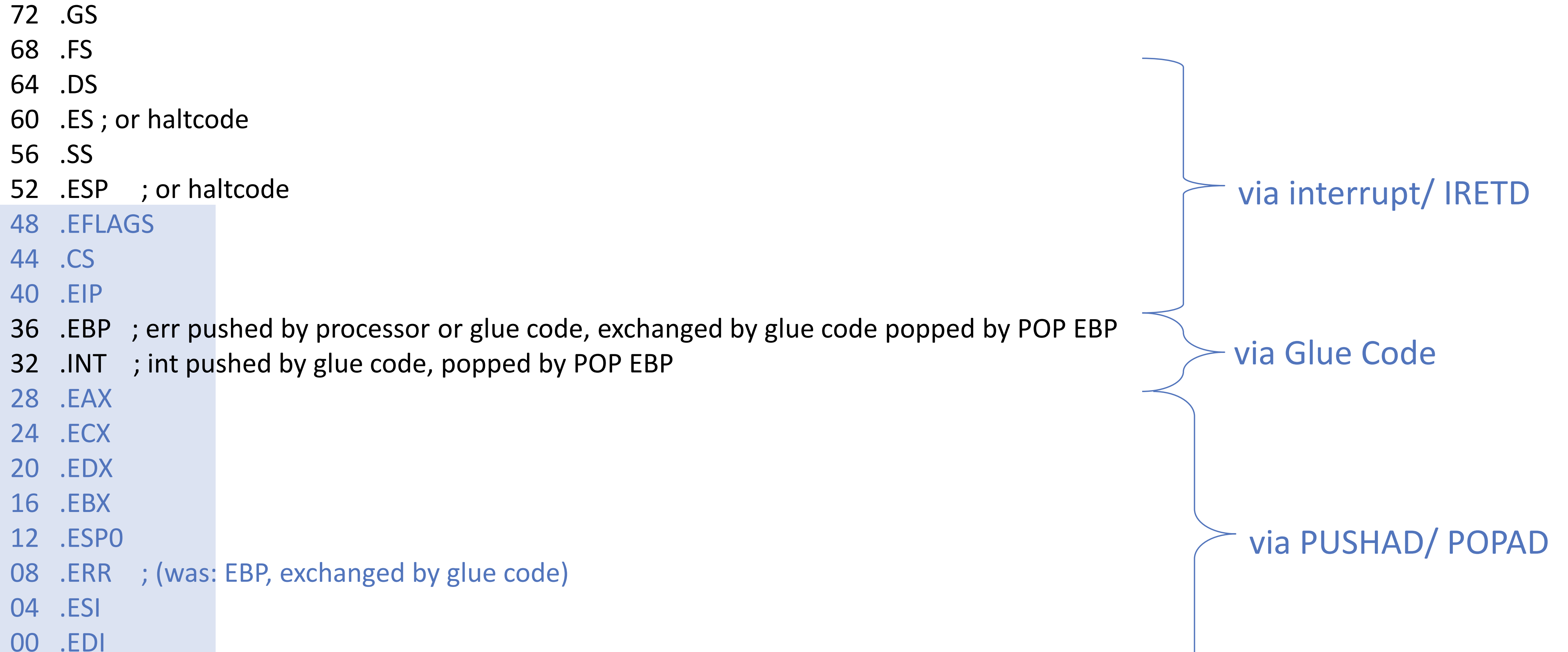
Synchronous Context Switch (2)



Asynchronous Context Switch

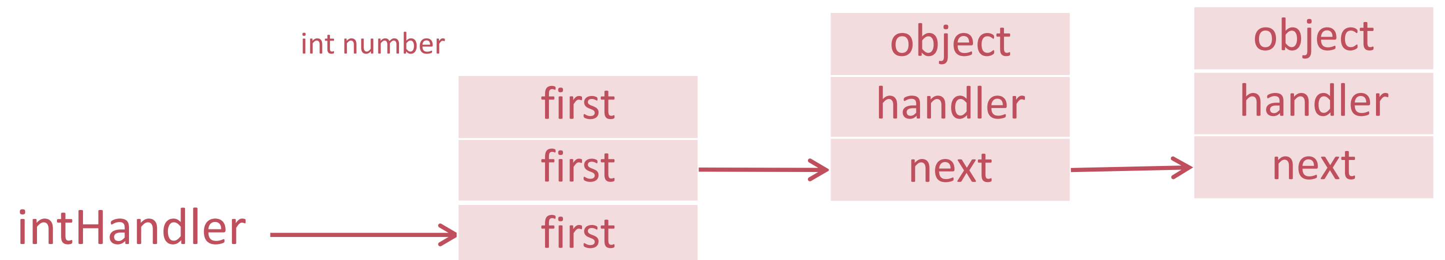


Stack Layout after Interrupt



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;
```

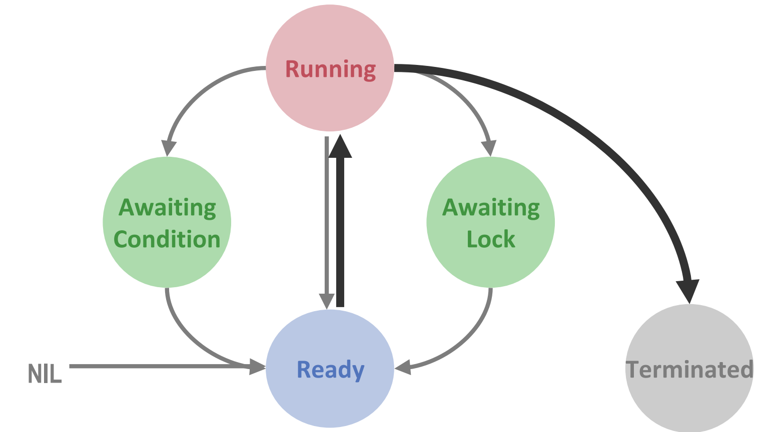
synchronous synchronous/ asynchronous

Switch Code (2)

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

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

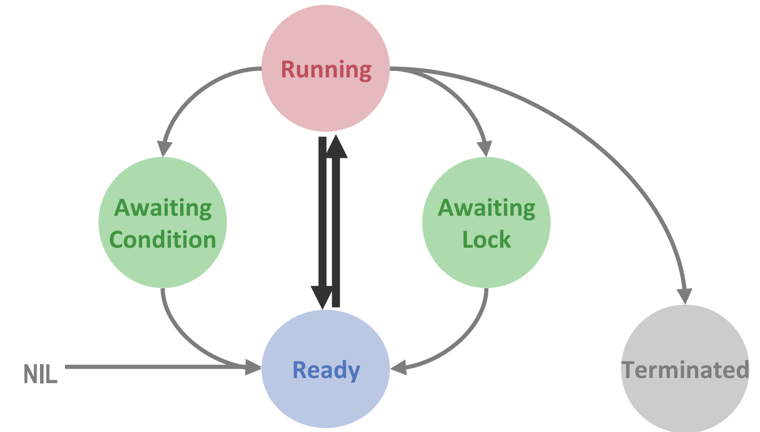
Example 1: Termination

```
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

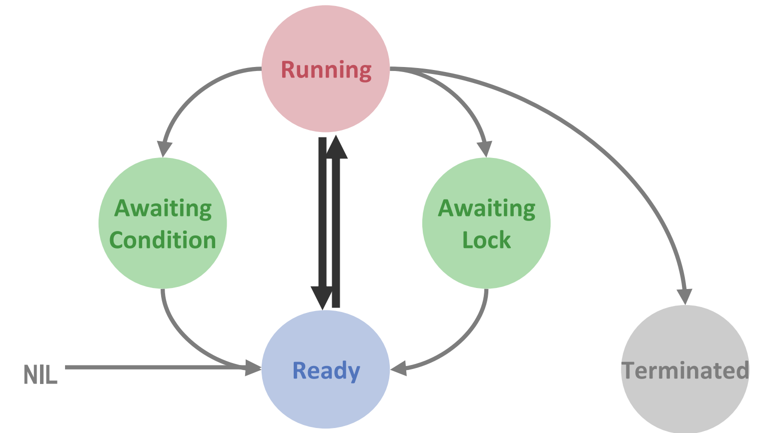
```
Idle = OBJECT
  BEGIN{ ACTIVE, SAFE, PRIORITY(PrioIdle)}
    LOOP
      REPEAT
         Machine.SpinHint
      UNTIL maxReady > MinPriority;
      Yield
    END
  END Idle;
```

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;
```

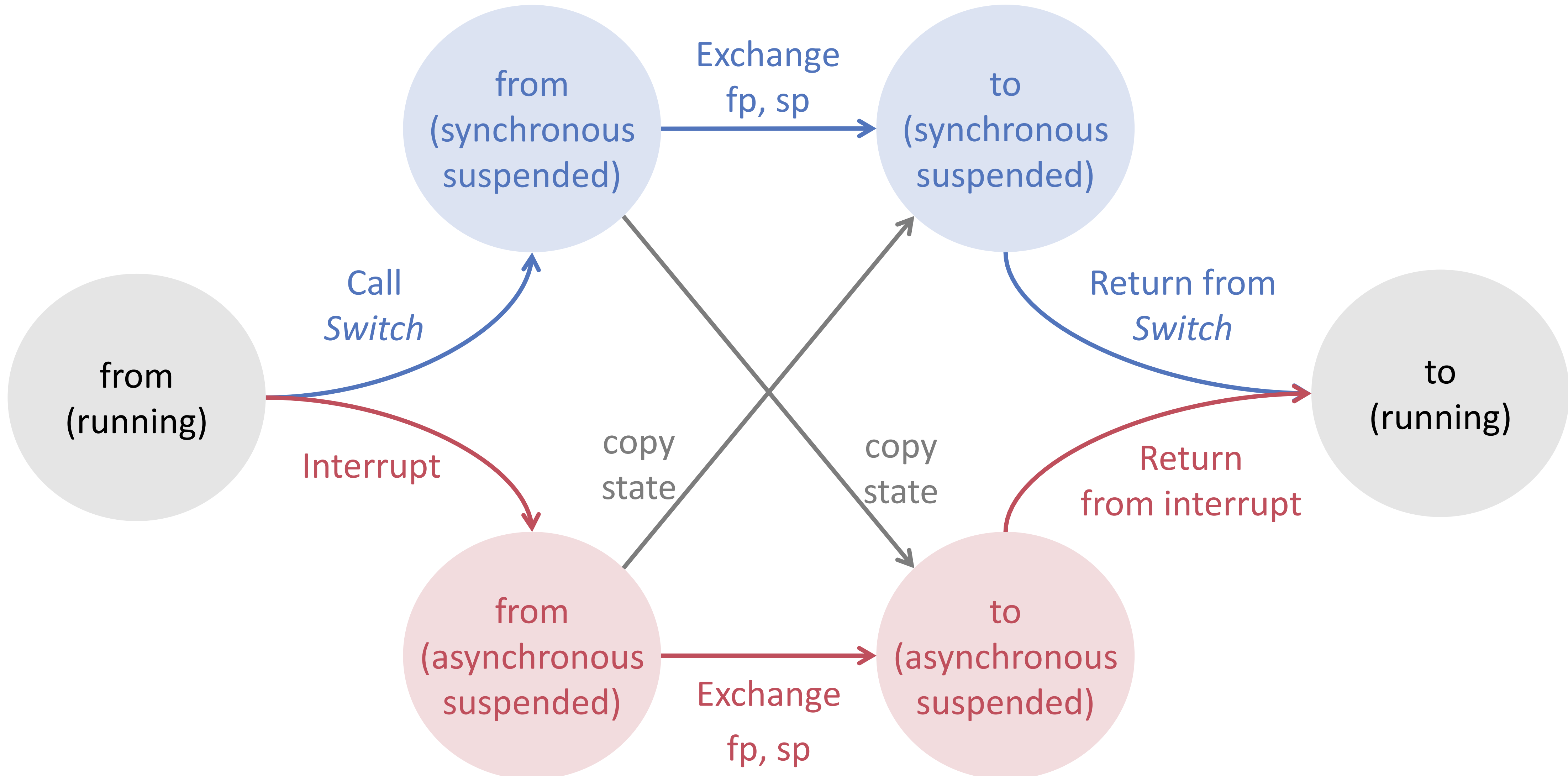
return from interrupt of new process

simulate return from procedure switch



asynchronous synchronous/ asynchronous

Context Switching Scenarios



Synchronization

- Object locking
- Condition management

Object Descriptors

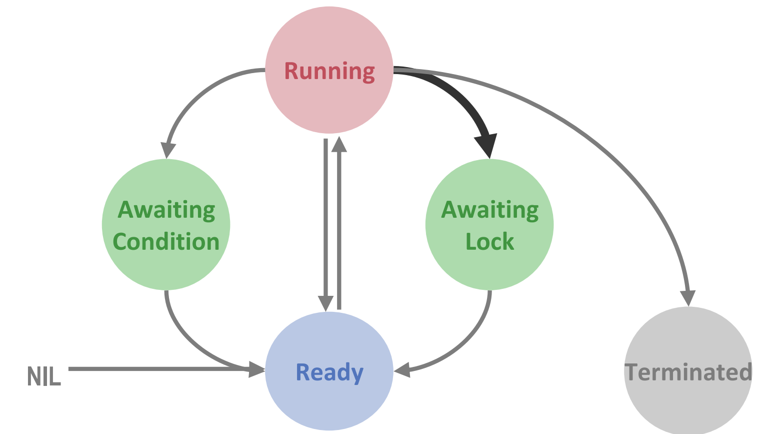
```
ObjectHeader = RECORD
  headerLock: BOOLEAN;
  lockedBy: Process;
  awaitingLock: ProcessQueue;
  awaitingCondition: ProcessQueue;

  …




END;
```

Fields added by system to objects with mutual exclusion

Type-specific instance fields

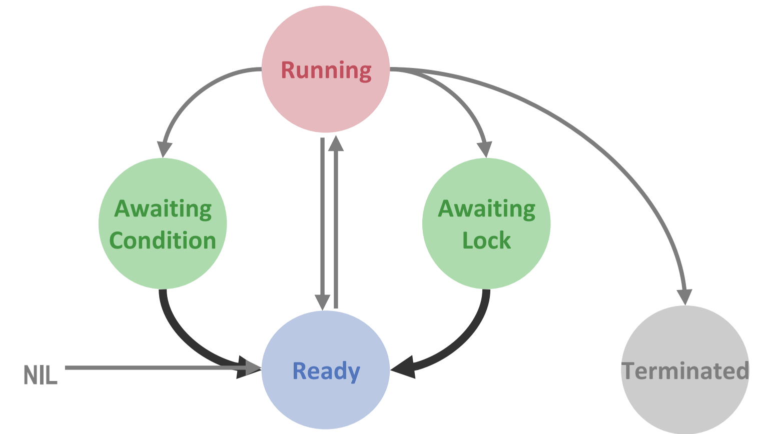
Object Locking

```
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

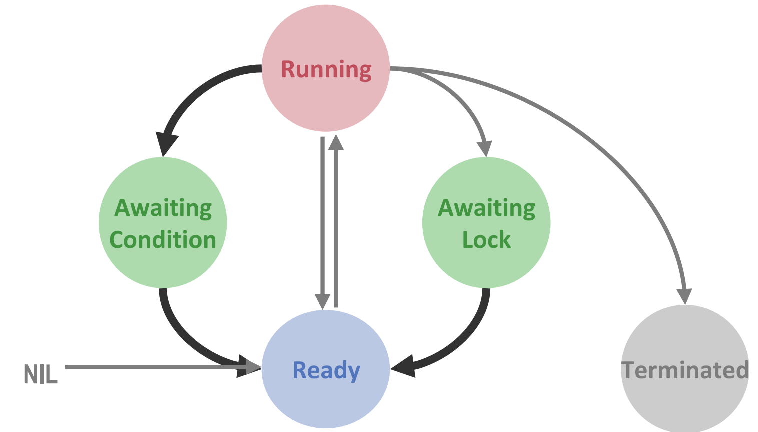
```
PROEDURE $Condition(fp: ADDRESS): BOOLEAN;
BEGIN
  RETURN ”condition from await statement”
END $Condition;
```

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;
```

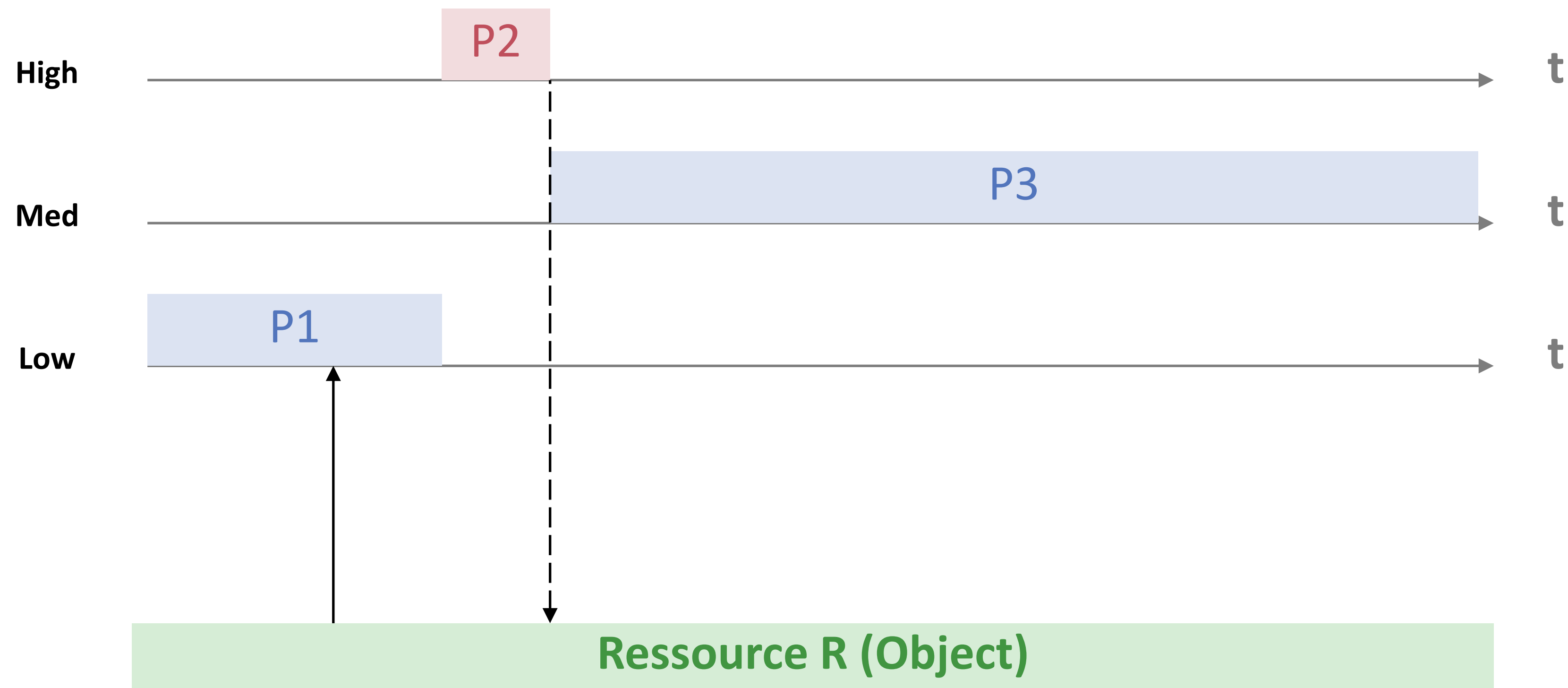


Condition Evaluation

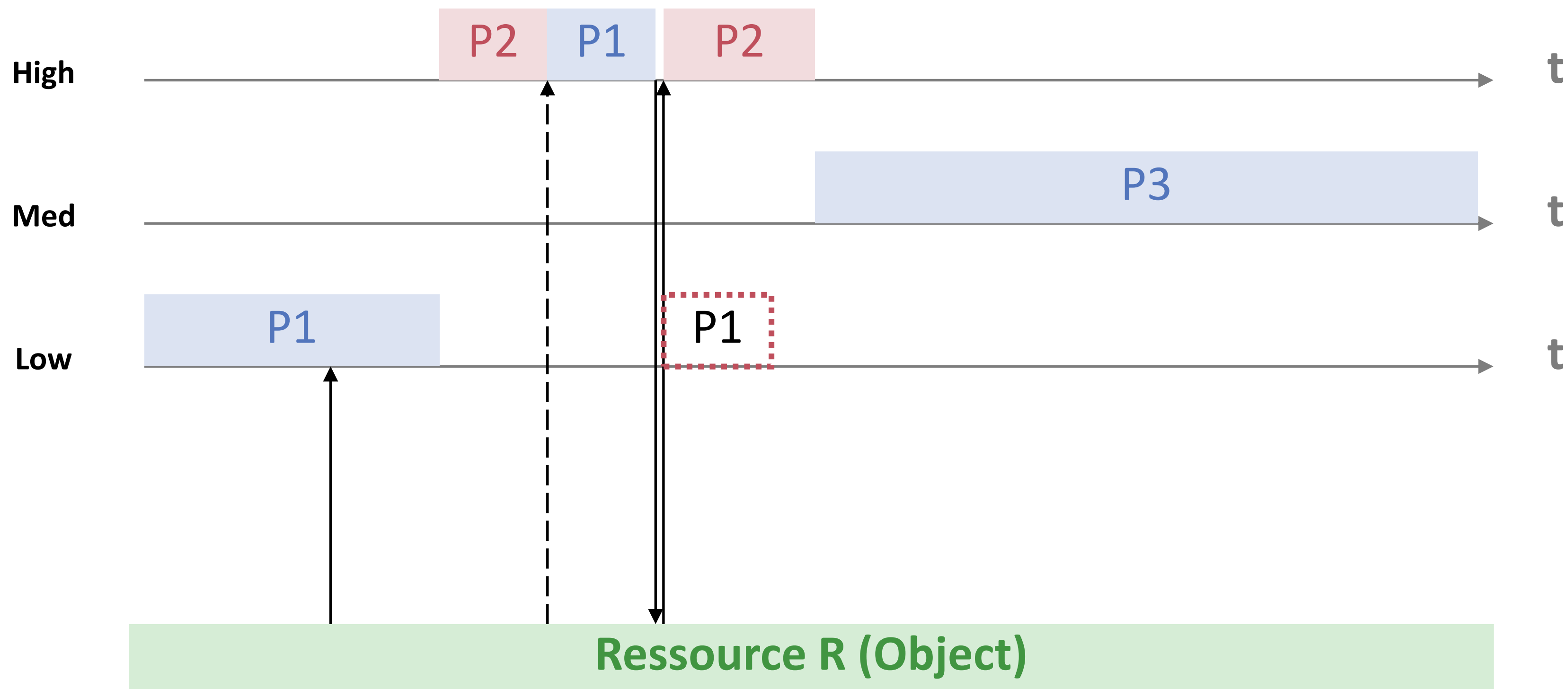
```
PROCEDURE FindCondition (VAR q: ProcessQueue): Process;
VAR first, c: Process;
BEGIN
   Get(q, first);
   IF first.condition(first.conditionFP) THEN
      RETURN f
   END;
   Put(q, first);
   WHILE q.head # first DO
      Get(q, c);
      IF c.condition(c.conditionFP) THEN
         RETURN c
   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

Priority Inversion

Inversion caused by an immediately shared resource

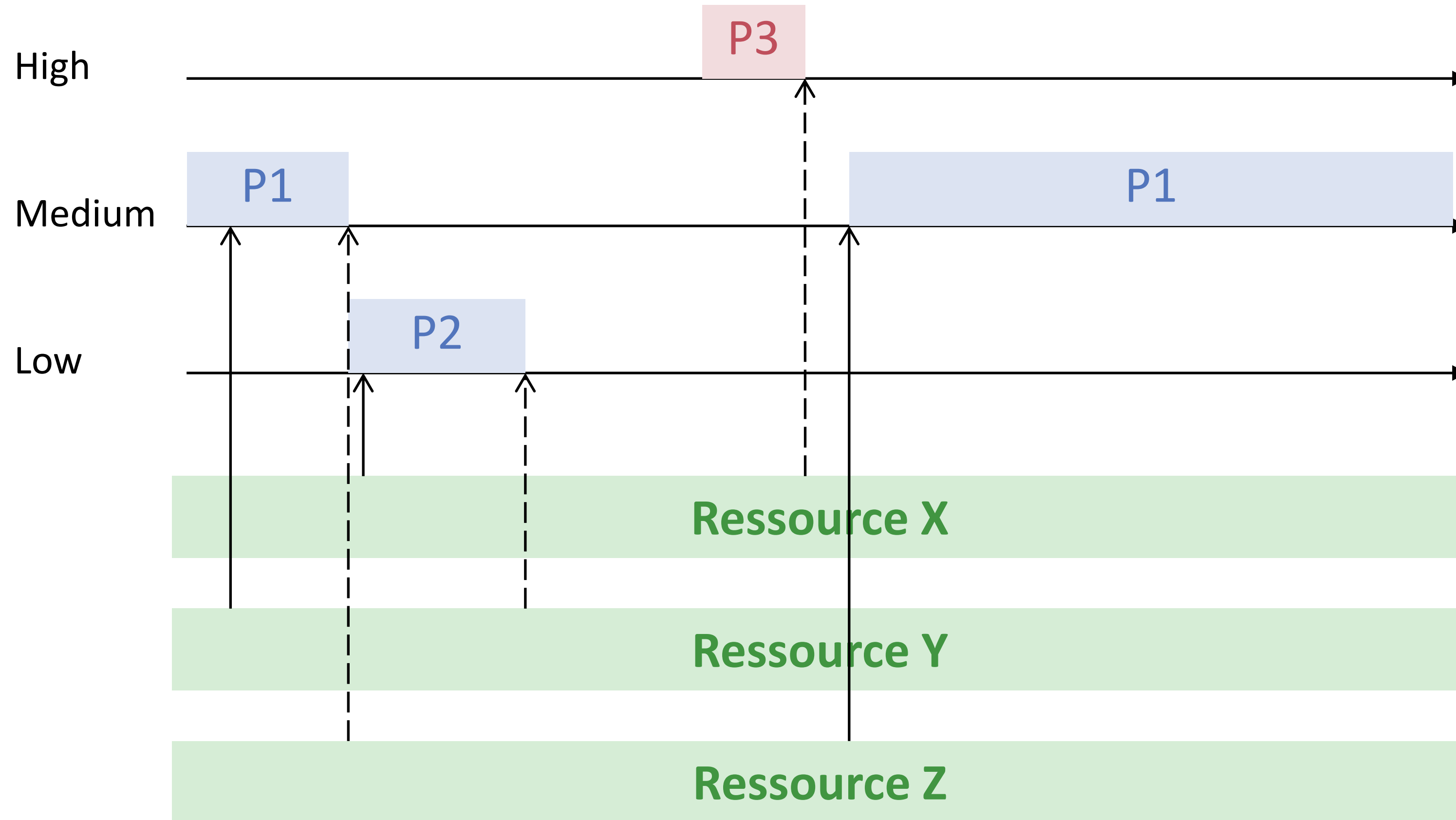


Priority Inheritance

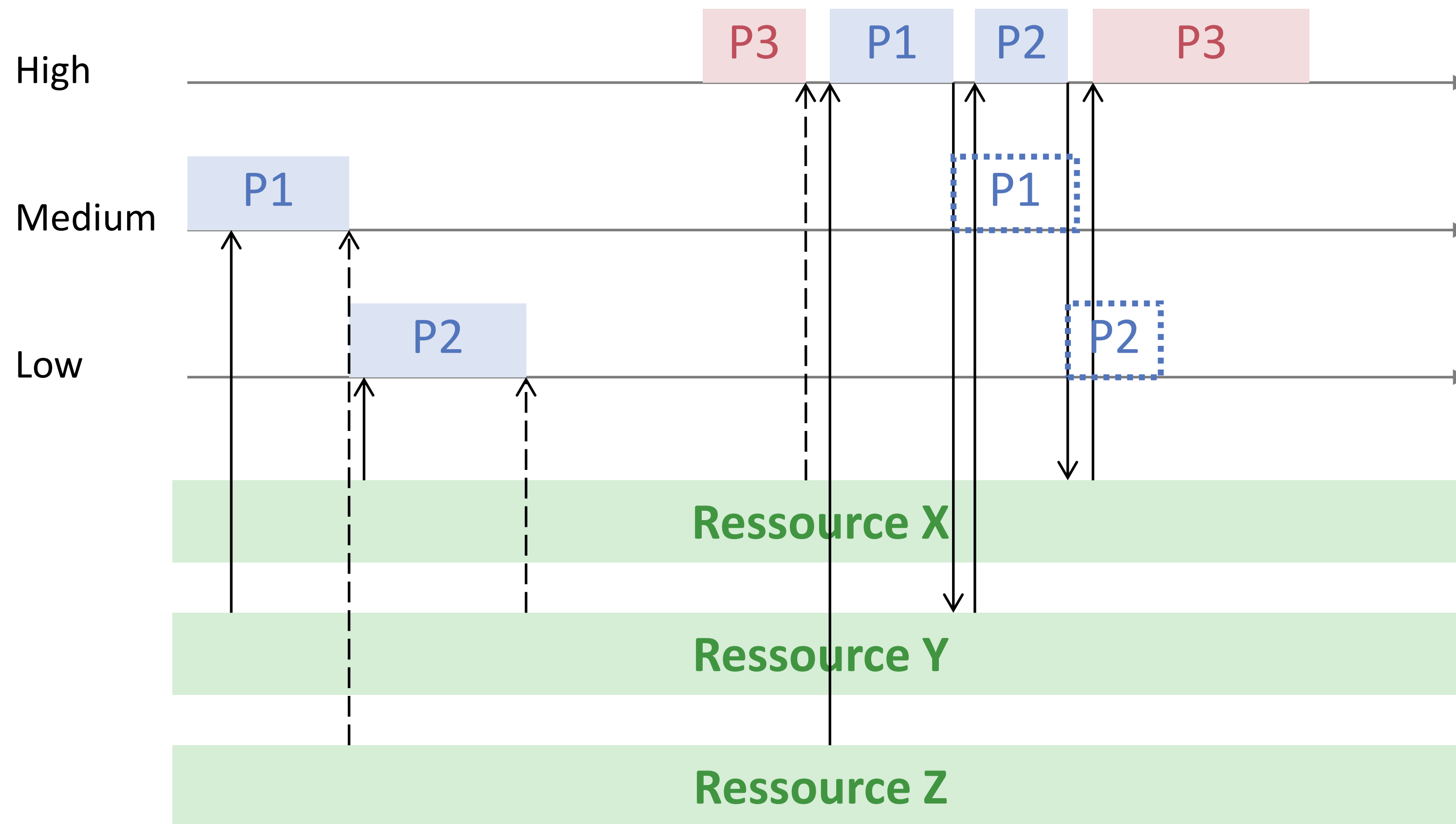


Priority Inversion

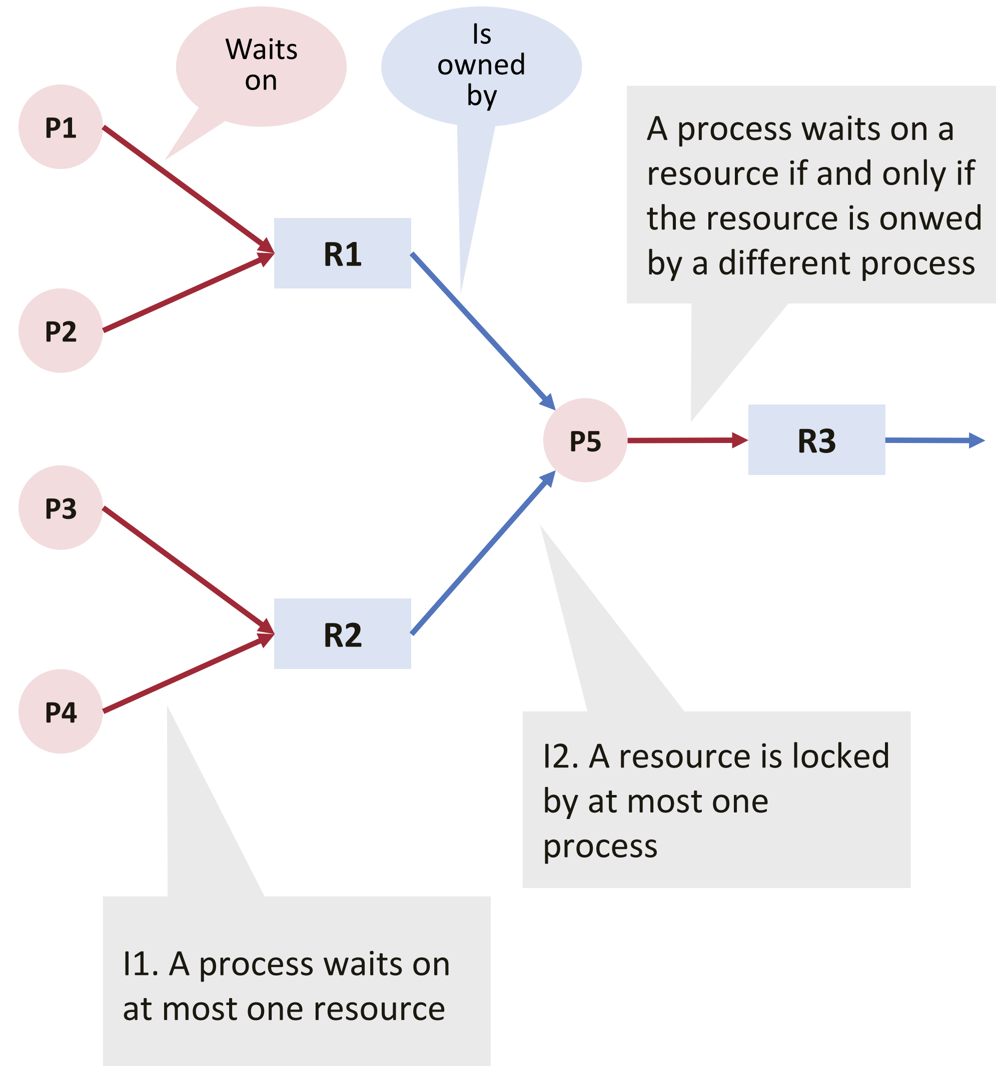
Inversion caused by not immediately shared resource



Priority Inheritance

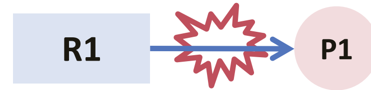


Invariants

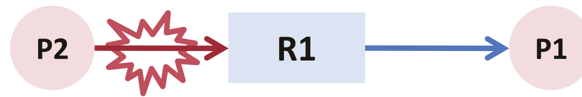


Possible Transitions

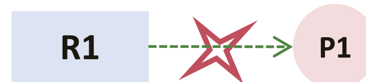
process
acquires and receives
free resource



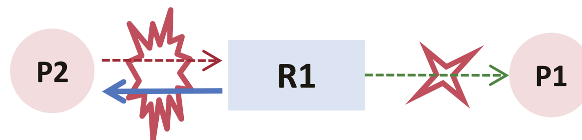
Process tries to
acquire locked
resource



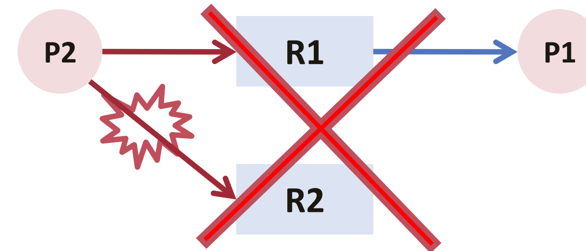
process
releases resource

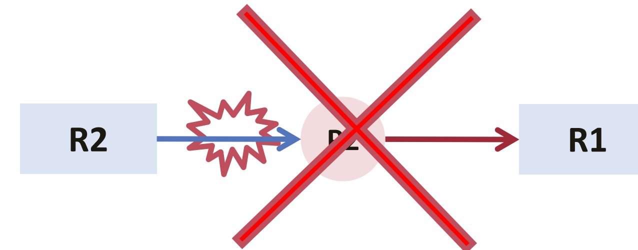


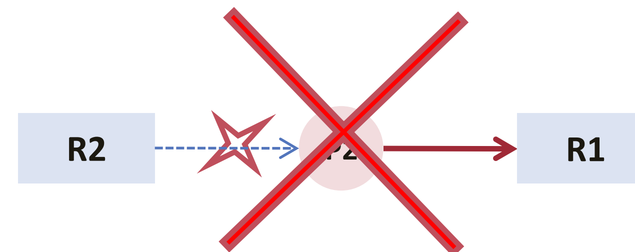
waiting process
receives resource



waiting processes can
neither acquire
nor release resources

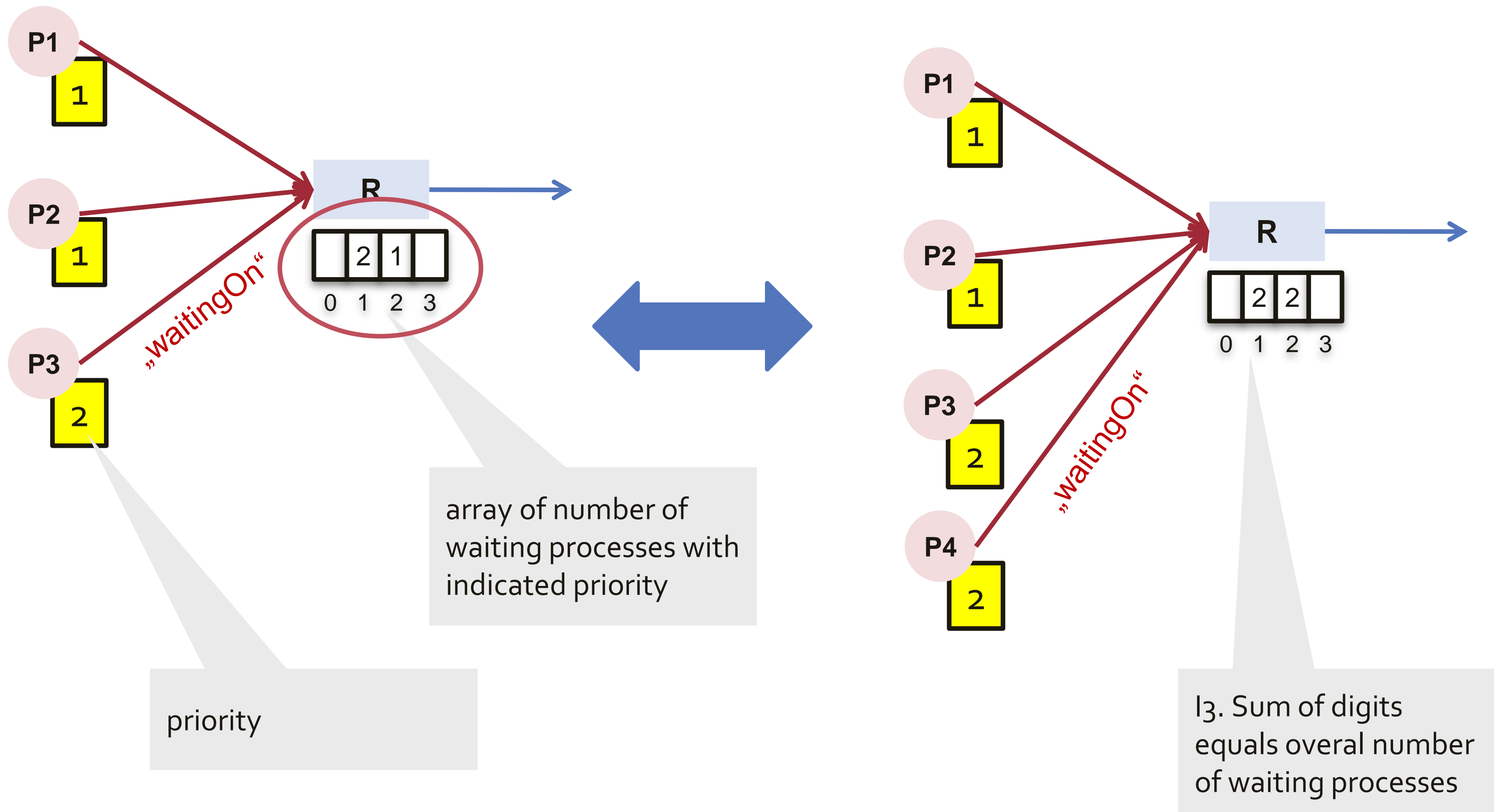






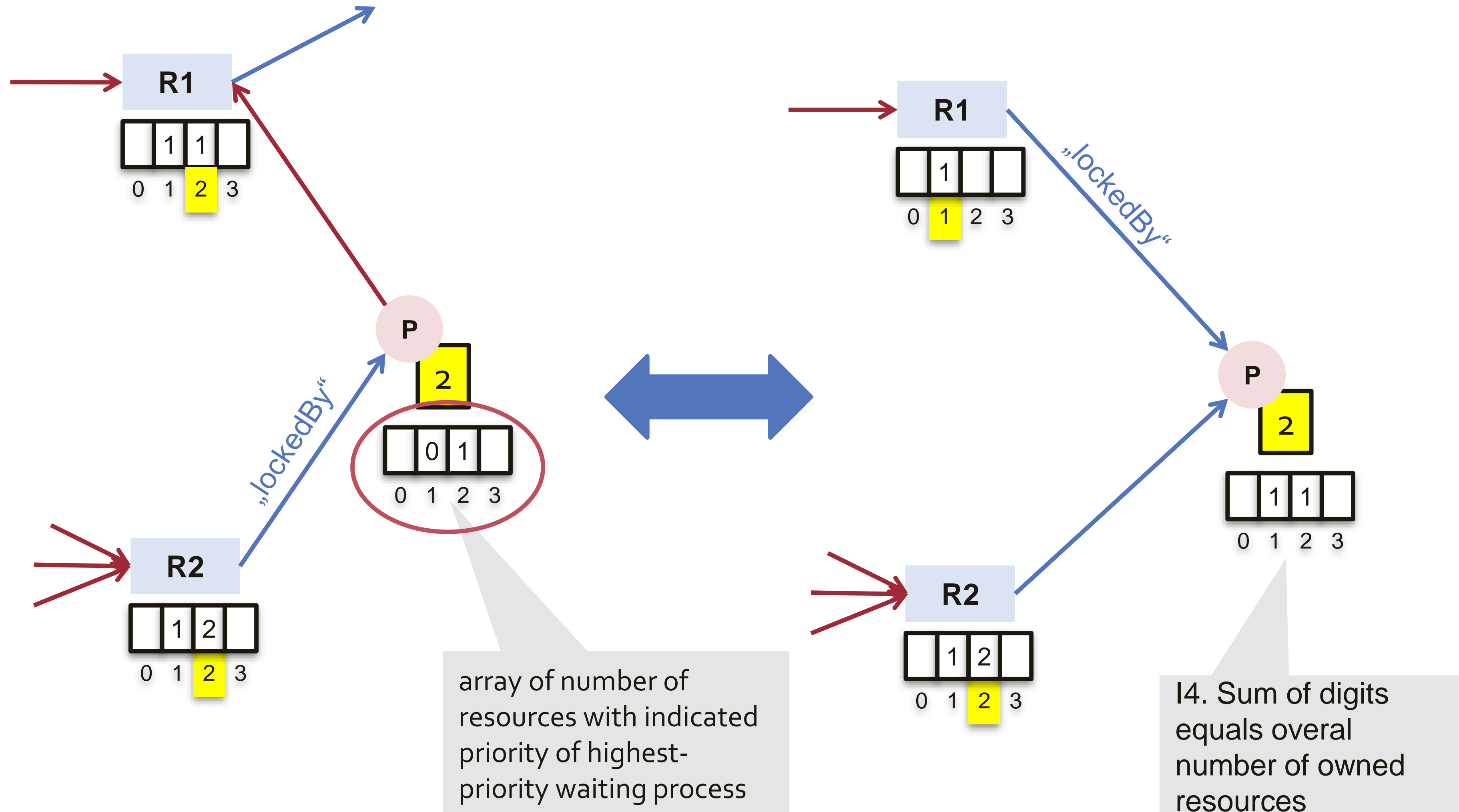
WaitingPriorities List

(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.

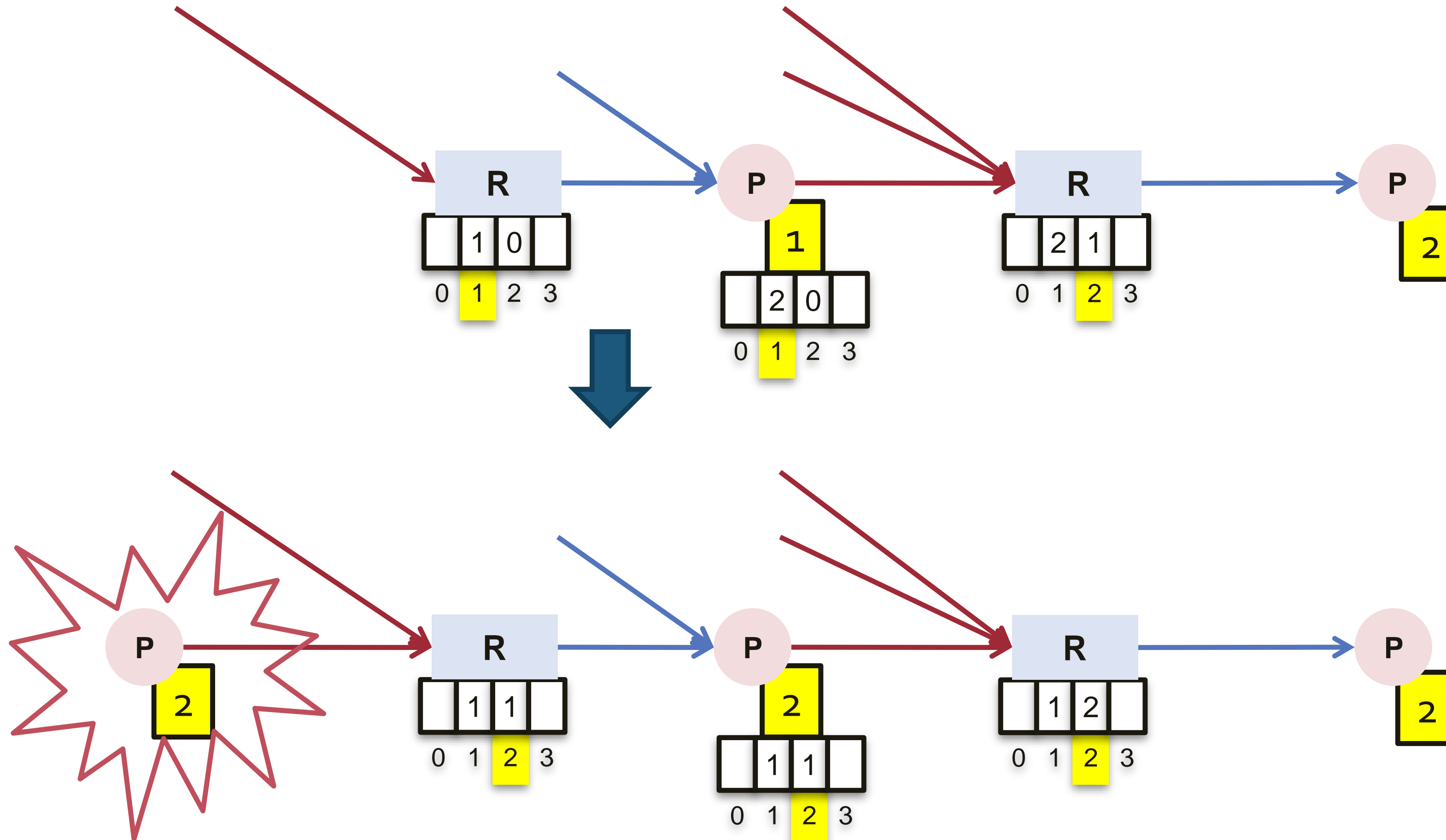
- invariant held during (re-)setting the „waitingOn“ pointer

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

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

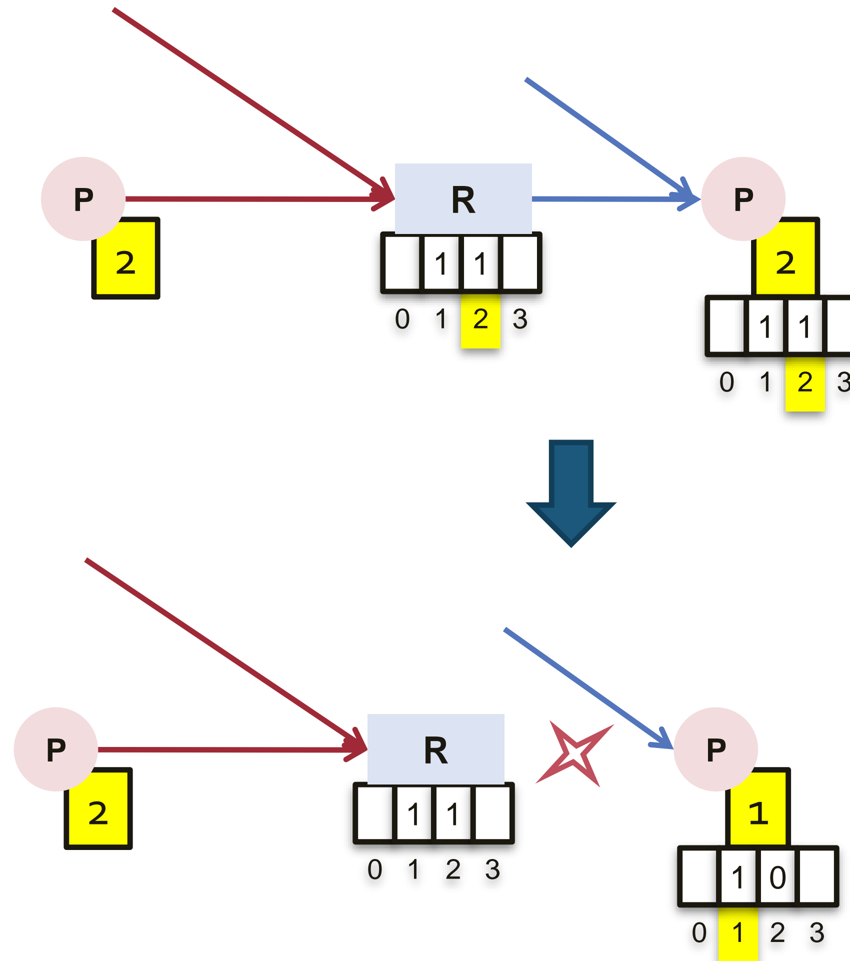
Handling Priority Inversion

(Setting „WaitingOn “)



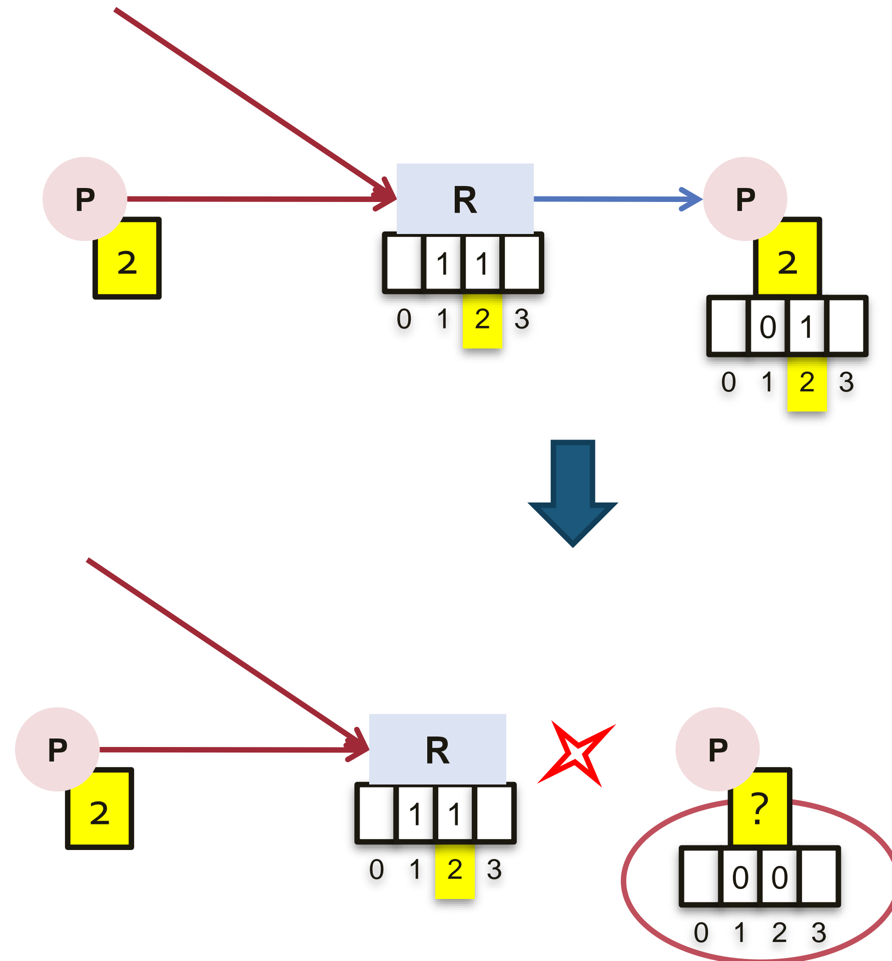
Handling Priority Inversion

(Resetting „LockedBy“)



Handling Priority Inversion

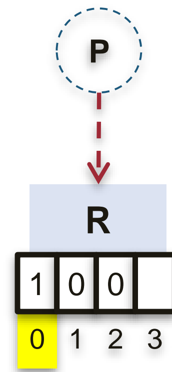
Resetting „LockedBy“: how to preserve initial priorities?

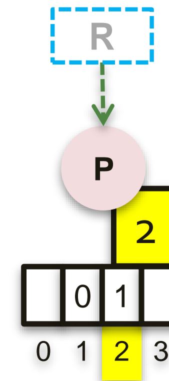


Handling Priority Inversion

Resource and Process Initialization

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





COMPILER-SUPPORTED UNIFICATION OF STATIC AND DYNAMIC LOADING

Contents of an Object File

- Machine code and data
- Metadata, i.e. information about
  - relocation
  - reflection
  - garbage collection
  - debugging
  - exception handling
  - runtime type system
  - etc.

Examples

- PE - Portable Executable / COFF – Common Object File Format
  - MS Windows .EXE and .DLL files, EFI applications
  - Header describes physical contents of the file and metadata
  - Metadata: import / export tables, relocation, debug information, target architecture.

- ELF - Executable and Linkable Format
  - Posix Executables
  - Different section types for code, data and metadata

- Common properties:
  - Metadata stored separate from code and data using a special format
  - Linking is more complex than just resolving references.
  - Not particularly designed for / with programming language support.

Motivation

- Modifications of toolchain and runtime support of programming language became more and more difficult over time.
- This was partially due to an increasingly complex object file format that became
  - hard to understand
  - hard to maintain
  - hard to extend
- Wanted to exploit co-design of language, compiler and runtime system to improve the situation.

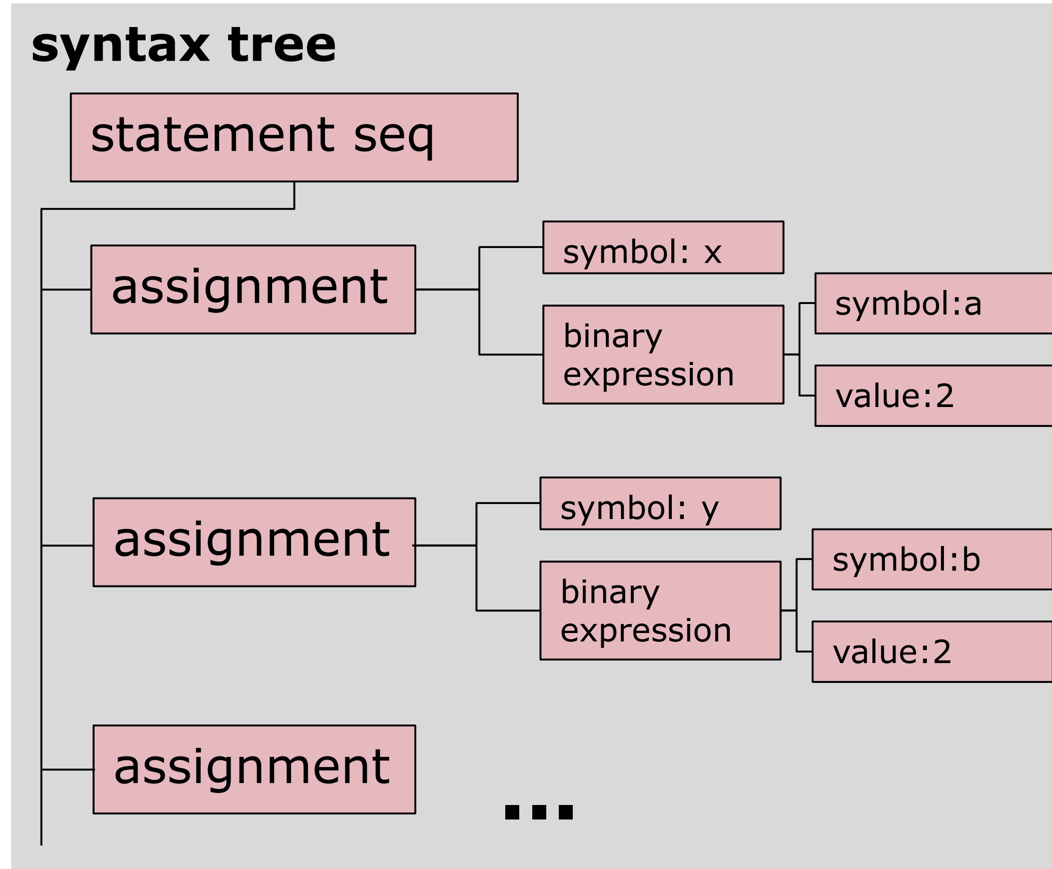
Compilation Process

Parsing

source code

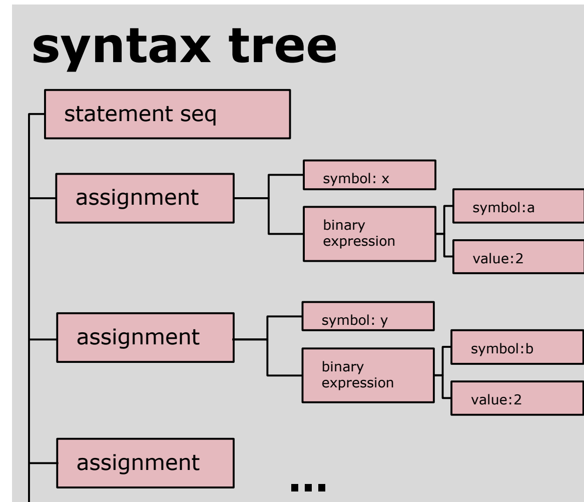
```
x := a * 2;
y := b * 2;
z := ...
```

scanning,
parsing,
checking
(resolving)



Compilation Process

Frontend



syntax tree traversal

intermediate code

```
.module A
 ....

.code A.P
  ....
  1:  mul s32 [A.x:0], [A.a:0], 2
  2:  mul s32 [A.y:0], [A.y:0], 2
   ....
```

Compilation Process

Backend

intermediate code

```
.module A
 ....

.code A.P
  ....
  1:  mul s32 [A.x:0], [A.a:0], 2
  2:  mul s32 [A.y:0], [A.y:0], 2
  ....
```

IR code traversal

binary code

Fixups

```
fixup 2 <-- A.a (displ=0) [3+7 abs]
fixup 6 <-- A.x (displ=0) [3+7 abs]
fixup 7 <-- A.b (displ=0) [3+7 abs]
...
[  2] 30807 ; LD R1, [0]
[  3] 00401 ; MOV R0, R1
[  4] 2801F ; ROR R0, 31
[  5] 14001 ; BIC R0, 1
[  6] 34007 ; ST R0, [0]
....
```

Symbolic References

Source Code / Interface (Symbol File)

```
MODULE A;
VAR a*: LONGINT;

PROCEDURE P*;
BEGIN a := a*10 END P;
END A.
```

```
MODULE A;
    VAR a*: LONGINT;
    PROCEDURE P*;
```

```
MODULE B;
IMPORT A;
BEGIN A.a := 1; A.P() END
B.
```

```
MODULE B;
    IMPORT A;
```

Intermediate Code

```
.module A
.const A.@moduleSelf offset=0
  0:  data u32 0
.var A.a offset=-4
  0:  reserve 4
.code A.P offset=0
  0:  enter 0, 0
  1:  mul s32 [A.a:0], [A.a:0], 10
  2:  leave 0
  3:  return 0
```

```
.module B
.imports A
.const B.@moduleSelf offset=0
    0:  data u32 0
.code B.$$Body offset=0
  0:  enter 0, 0
  1:  mov s32 [A.a:0], 1
  2:  call u32 A.P:0, 0
  3:  leave 0
  4:  return 0
```

Fixups

Object File*

```
const A.@moduleSelf 651605535 8 aligned 4 0 4
    00000000
data A.a 471859334 8 aligned 4 0 4
    00000000
code A.P 2046820492 8 aligned 0 2 24
    abs 12 A.a 471859334 0 0 1 0 32
    abs 18 A.a 471859334 0 0 1 0 32
    8C0000008BA0000000F0FA5000000000985000000009C3C
code A.$$BODY -278328333 8 aligned 0 0 6
    8C0000009C3C
```

```
const B.@moduleSelf 651605535 8 aligned 4 0 4
    00000000
code B.$$Body -345437455 8 aligned 0 2 21
    abs 6 A.a 471859334 0 0 1 0 32
    rel 15 A.P 2046820492 -4 0 1 0 32
    8C0000007C5000000000100000008EDEFFFFFF9C3C
```

Fingerprint

consistency

Fingerprint (must match)

(relative) Fixup

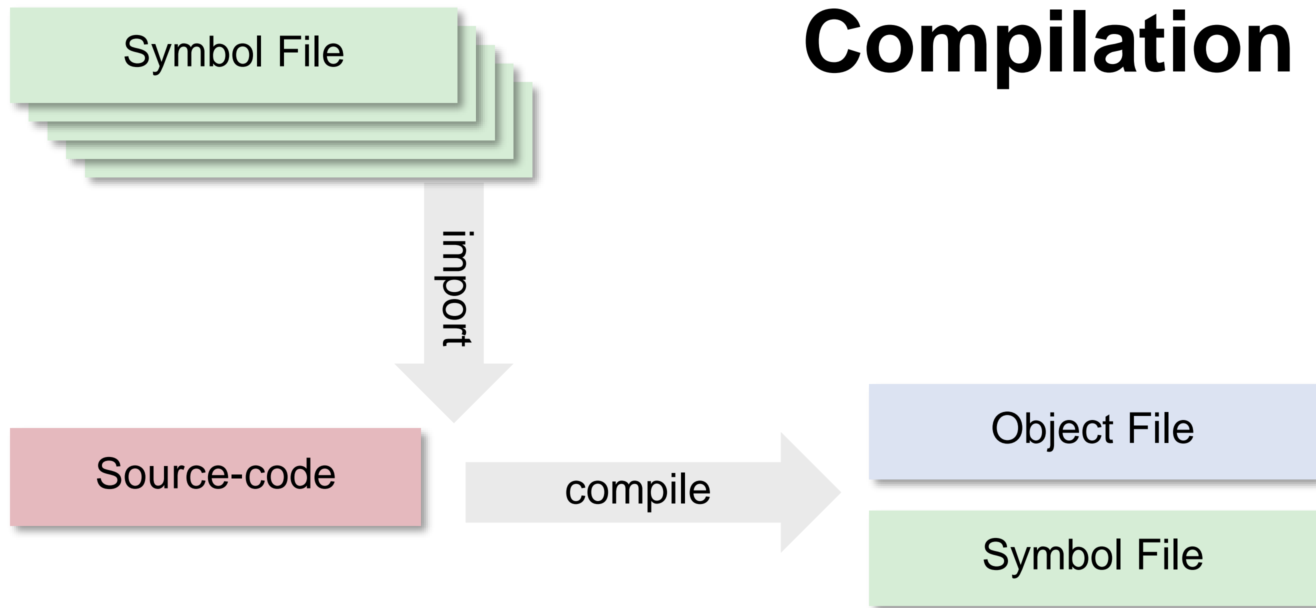
261

*here: Intel x86

Recap: The role of the Object File

in a system with dynamic loading

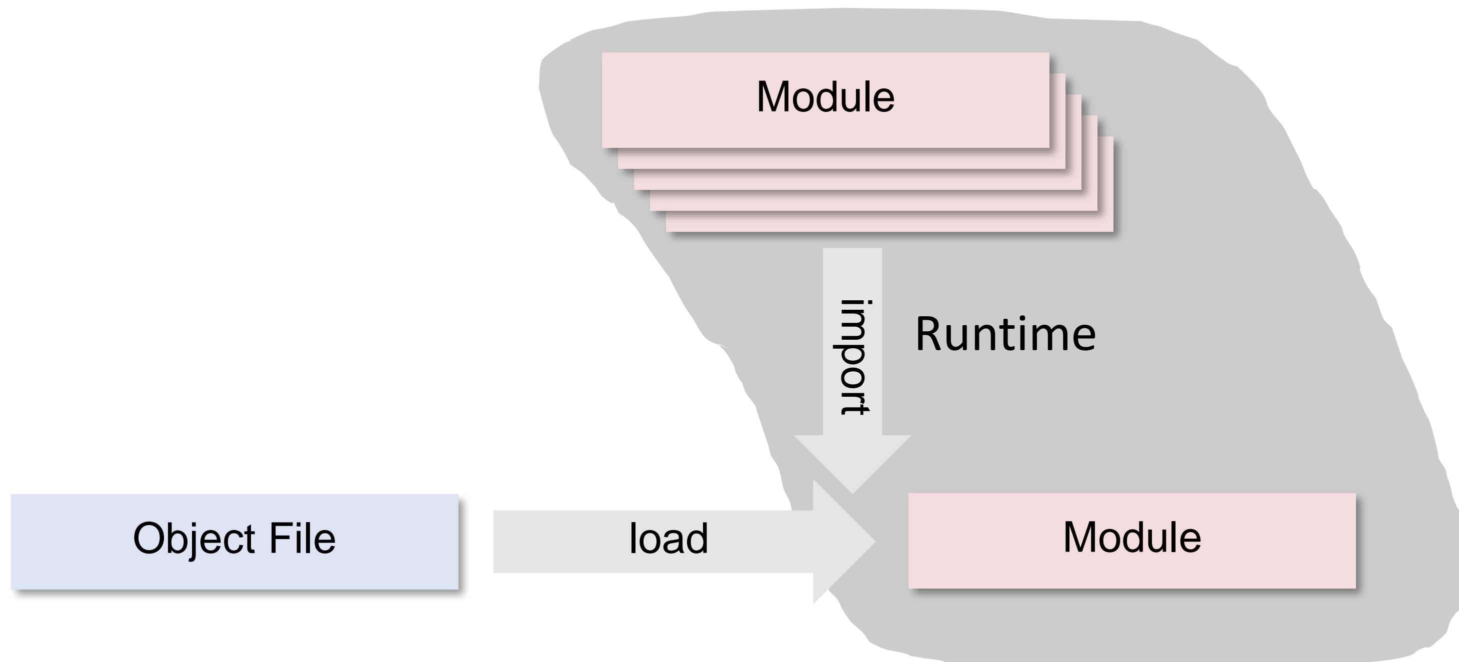
Compilation



Recap: The role of the Object File

in a system with dynamic loading

Static Linking



Dynamic Linking

Required Metadata

Language Features

- type-safe
- object oriented
- modular
- garbage collected
- exception handling
- process synchronisation
- post-mortem debugger
- …

Required Metadata

- type descriptor
- method table
- module descriptor
- pointer offsets
- exception pointers
- process queues
- debugging info
- …

The Linking Process

- Linker or loader must provide at least a mechanism to resolve references when arranging sections of an object file in memory.

- Metadata are usually generated from designated parts of the object file stored in a proprietary format.

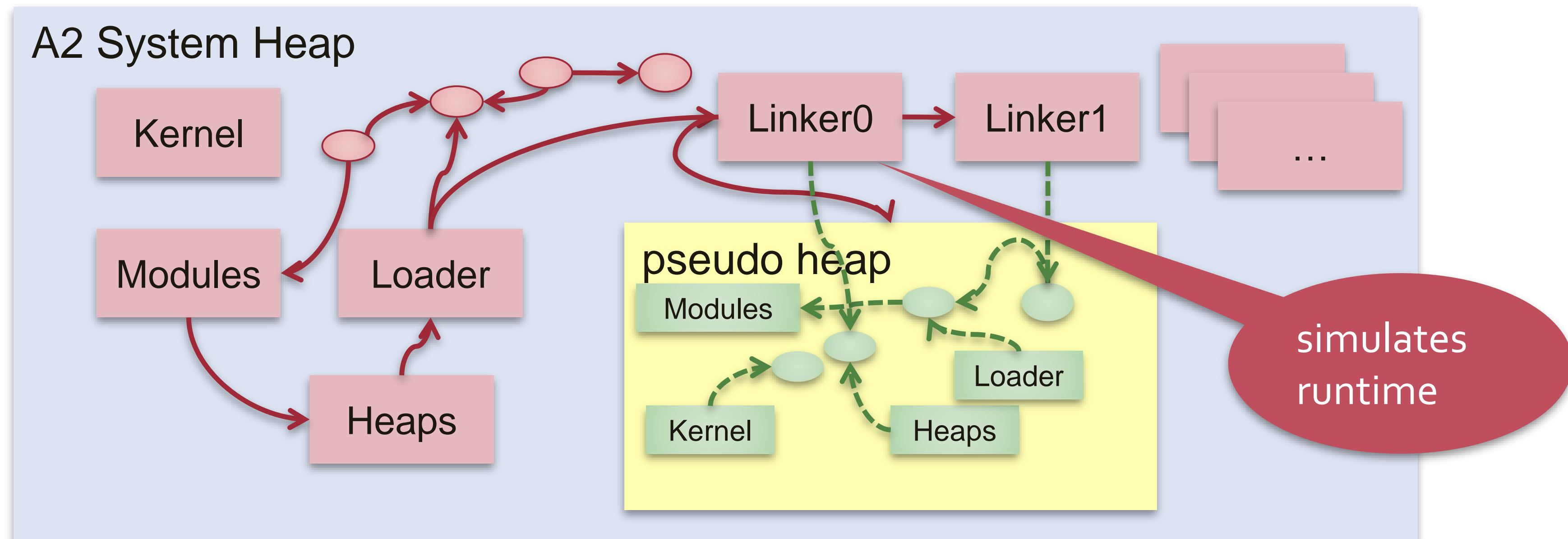
Traditional Object File in Oberon

SectionName	Meaning	Data stored
Commands	list of commands	name, argType, retType, codeOffset
Pointers	list of addresses of global variable pointers	pointerOffset
Imports	imported [illegible]	module Name
Links	fixup lis[illegible]	code and data offsets of fixup queues / case tables
Constants	constant section	[illegible]
Exports	exported symbols	[illegible]gerprint, entry , exportType
Code	code section	b[illegible]
Use	references to imported modules and system calls	moduleNa[illegible]gerprint
Types	description of types	name, baseModule , base[illegible]ntry, methods etc.
Refs	debugging information	(long proprietary format)
ExceptionTable	list of finally sections	pcFrom, pcTo, pcHandler
PtrsInProcs	pointers on stack frames	codeOfs, beginOfs, endOfs, numberPointers list of pointers

Too complex

Traditional Oberon Linking Approach

- Allocate a pseudo-heap and use a simulated module loading to place modules and all necessary data structures in this heap.
- Then store heap as an array of bytes.
- To solve the Hen-and-Egg problem, the linker imitates behaviour of the runtime system and thus generates a 1:1 image.



Disadvantages of Traditional Approach

- Static Linker and Dynamic Loader and all data structures duplicated.
- A modification of the kernel implies a lot of work and is error-prone.
- A reduction of complexity is hardly possible.
- **Definitive Showkiller:** No crosslinking possible!

A different approach

The compiler generates all **metadata as ordinary data sections** and uses the fixup mechanism to establish the necessary links.

- For a modification of the runtime structures only the compiler and little parts of the runtime modules have to be adapted.
- Loader and linker do only need to arrange data in memory / boot image and patch the fixups
- Loader and linker are nearly identical and can use the same code base for patching fixups
- Optimizations, such as sorting and generation of hash-tables for addresses or fingerprints may still be performed by the loader

Object File Format

Object File consists of a list of sections

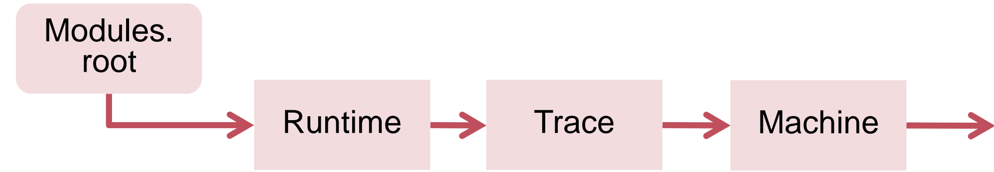
- Section Types
  - Code Sections
  - Data Sections
- Section Attributes
  - Identifier: globally unique name + fingerprint
  - Relocatability: aligned or fixed
  - Unit in bits, Size in units

Each section contains

- A list of fixups
- A sequence of bits representing code or data

Metadata Hook:

Modules.Module




```
Module* = OBJECT (Heaps.RootObject)
VAR
        next*: Module;         (** once a module is published, all fields are read-only *)
        name*: Name;
        init, published: BOOLEAN;
        refcnt*: LONGINT; (* counts loaded modules that import this module *)
        sb*: ADDRESS; (* reference address between constants and local variables *)
        entry*: POINTER TO ARRAY OF ADDRESS;
        command*: POINTER TO ARRAY OF Command;
        ptrAdr*: POINTER TO ARRAY OF ADDRESS; (* traced explicitly in FindRoots *)
        ...

END Module;
```

Metadata: Modules.Module

Intermediate code

```
.const Test.@Module
0:  data u32 0    ; headerAdr
1:  data u32 0    ; typeDesc
2:  data s32 -1    ; mark: LONGINT;
3:  data u32 Test.@Module:21  ; dataAdr
4:  data u32 0    ; size-: SYSTEM.SIZE
5:  data s32 0    ; count*: LONGINT
6:  data s32 0    ; locked*: BOOLEAN
7:  data u32 0    ; awaitingLock*: ProcessQueue
8:  data u32 0
…
19: data u32 Test.@Module:2    ; HeapBlock
20: data u32 0    ; TypeDescriptor
21: data u32 0    ; nextRoot*: RootObject
22: data u32 0    ; next*: Module
23: data u8 84    ; name*: Name
24: data u8 101
….
62: data u32 Test.@CommandArray:7;
….
```

Layout of a Heap Block

Modules.Module

Object File

```
module Module;

var variable: integer;

procedure Procedure;
begin
    variable := 10;
    trace(variable);
end Procedure;

end Module.
```

section= type name fingerprint unit-size alignment #fixups size

...

data Module.variable 1053397876 8 aligned 2 0 2

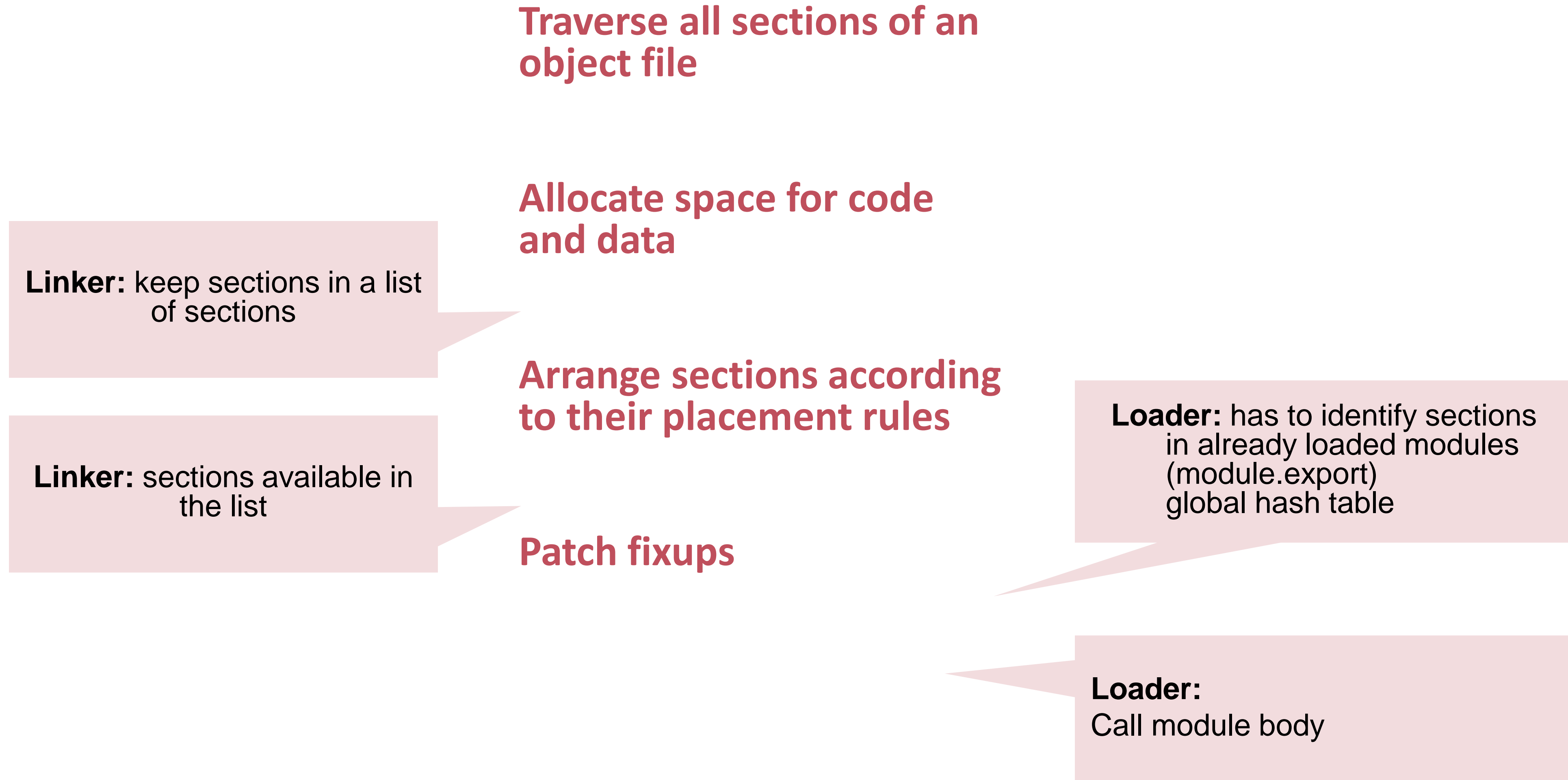
code Module.Procedure -1838848940 8 aligned 1 6 60

fixup = name fingerprint #patches {type displacement shift #patterns {offsets bits} {offsets} }

```
Module.variable 1053397876 1 abs 0 0 1 0 32 2 6 27
Module.@const0 -1762689000 1 abs 0 0 1 0 32 1 15
KernelLog.String 1370406993 1 rel -4 0 1 0 32 2 20 47
KernelLog.Int 828103137 1 rel -4 0 1 0 32 1 35
Module.@const1 846277559 1 abs 0 0 1 0 32 1 42
KernelLog.Ln -1372435551 1 rel -4 0 1 0 32 1 52
55985E667C5000000000A000A66186000000008E8E
FFFFFFF0FBD30000000075A6108E9DFFFFFFA6308
6000000008EDCFFFFFF8E8CFFFFFF98CED53C
```

....

Object File Loading / Linking



Some numbers

Traditional object file format

Module	Lines of code	Number Characters	Code size
Linker0	1554	57k	26k
Linker1	887	28k	17k
Linker	95	3k	3k
Loader	891	28k	18k
Compiler ObjFileWriter	2009	71k	37k
Sum	**5456**	**187 k**	**101 k**

Generic object file format

Module	Lines of code	Number Characters	Code size
ObjFileWriter	278	8k	6k
GenericLinker	234	8k	8k
StaticLinker*	400	16k	10k
Loader	380	12k	6k
Compiler ObjFileWriter	135	5k	6k
Sum	**1427**	**49 k**	**36 k**

*including PE32, PE64, ELF and native format