Completing the Inductive Step

For every $k$, from every quiescent configuration $C$, we can reach a configuration $D$, by steps of $p_0,\ldots,p_{k-1}$ only, s.t.

(a) $p_0,\ldots,p_{k-1}$ cover $k$ distinct variables in $D$
(b) $D$ is $\{p_k,\ldots,p_{n-1}\}$-quiescent

Repeat until the sets are repeated ($W_i = W_j$), and then apply the previous argument

Optimizing Memory Locality
Memory Access, in Formal Model

In Reality: Memory Interconnect

Memory is accessed through an interconnection network (e.g., a bus)
Local Memory: CC model

Interconnect traffic is expensive
Store copies of data in local memory (cache)
Keep caches coherent with memory and each other (cache coherence model)

Local Memory: DSM model

Larger memory banks are located at the processors (distributed shared memory model)
Local & Remote Accesses

- An access to $v$ by $p$ is **remote** if it is:
  - $p$'s first access to $v$ or
  - $v$ has been written by another process since $p$'s previous access

---

Local Spinning

- An algorithm is **local-spin** if all busy waiting is in read-only loops of local-accesses, which do not cause interconnect traffic

- An algorithm may be local-spin on one model (DSM or CC) and not local-spin on the other!

- The **remote memory references (RMR)** complexity of an algorithm is the number of remote accesses
R / W 2-Process Mutex

```
Want[i] = 0
wait until Want[1-i] == 0 or Priority == 1
Want[i] = 1
if (Priority == 1-i) then
    if (Want[1-i] == 1) then goto Line 1
else wait until (Want[1-i] == 0)
```

• Is this algorithm local-spin?
  – In the DSM model? No
  – In the CC model? Yes
• What is its RMR complexity?
  – In the DSM model? Unbounded
  – In the CC model? Constant

Test&Set Mutex

Entry section
```
wait until test&set(lock) == 0
```

Exit section:
```
reset(lock)
```

• Is this algorithm local-spin?
  – No (in both models)
• What is its RMR complexity?
  – Unbounded (in both models)
Test&Test&Set Mutex

Entry section

\[
\text{while ( test\&set(lock) == 1 )} \\
\text{wait until (lock == 0)}
\]

Exit section:

\[
\text{reset(lock)}
\]

Less traffic in CC model, still not local-spin.

Recall Anderson’s Algorithm

entry section:

\[
\text{myPlace} = \text{rmw(}\text{Last,}\text{Last+1 \mod n}) \\
\text{wait until Flags[myPlace] == 1} \\
\text{Flags[myPlace] = 0}
\]

exit section:

Is this algorithm local-spin? 
- In the CC model? Yes
- In the DSM model? No

Uses modulo 
Fetch & Inc
Local-Spin Mutex w/ Swap

Atomic register-to-memory swap operations, also called fetch-and-store
More common

Each process spins on its own location in array

Array contains the queue of waiting processes
Each entry in the array holds a pointer to the next process in line.

Shared variables:
Flags[0..n-1], binary; all initially 1
Tail {binary, {0,..,n-1}}, initially {0,0}

Local variables:
myRecord, prev {binary, {0,..,n-1}}, temp binary

entry section:
myRecord.value = Flags[i]
myRecord.slot = i
prev = swap(tail, myRecord)
wait until(Flags[prev.slot] ≠ prev.value)

exit section:
Flags[i]= 1 - Flags[i]
Local-Spin Mutex w/ Swap

**Shared variables:**
- Flags[0..n-1], binary; all initially 1
- Tail {binary, {0,..,n-1}}, initially {0,0}

**Local variables:**
- myRecord, prev {binary, {0,..,n-1}},
- temp binary

entry section:

```
myRecord.value = Flags[i]
prev = swap(tail, myRecord)
wait until(Flags[prev.slot] ≠ prev.value)
Flags[i] = 1 - Flags[i]
```

Is this algorithm local-spin?
- In the CC model? Yes
- In the DSM model? No

---

CLH Lock

[Craig 1993] and [Landin & Hagers, 1994]

- Also a queue, but does not allocate space for all processes
- Instead, “thread” records in a (virtual) linked list
CLH Lock

**entry section:**
```java
new myNode
pred = getAndSet(tail,myNode)
wait until ¬ pred
```

**exit section:**
```java
myNode = false
```

Pointers are kept in local memories

Is this algorithm local-spin?
- In the CC model? Yes
- In the DSM model? No

©Hagit Attiya
236358 (2018) Mutual Exclusion
85

©Hagit Attiya
236358 (2018) Mutual Exclusion
86
MCS Lock

[Mellor-Crummey and Scott, 1991]

- Maintain a more explicit queue of waiting processes
- Small space overhead
- Local spinning in CC & DSM models
  - Each process has a dedicated record that is enqueues and dequeues

MCS Lock: Enqueing for the lock

- Set tail to point to your record (with compare&set)
MCS Lock: Enqueing for the lock

• Set tail to point to your record (with compare&set)
• Make last element point to your record

MCS Lock: Enqueing for the lock

• Set tail to point to your record (with compare&set)
• Make last element point to your record
• Spin on your own record
MCS Lock: Unlock

• Notify next in line that is can go into the critical section
  – $p_i$ sets $p_j$’s flag to false
• Dequeue own record from the list
  – clear the next pointer
MCS Lock: Unlock Subtleties

• Another thread might be joining the list at the same time
  – No thread will be enabled for the critical section
  – Exception ($p_k$ accesses $p_i$’s reclaimed memory)

MCS Lock: Unlock Subtleties

• Another thread might be joining the list at the same time
  • Can be detected since tail is not null
    – Wait for next to be filled before proceeding
MCS Lock: Unlock Subtleties

- Another thread might be joining the list at the same time
- Can be detected since tail is not null
  - Wait for next to be filled before proceeding to set its flag to false

\[
\text{tail} \quad \text{\texttt{false}} \quad \text{\texttt{false}}
\]

\[
\text{\texttt{pi}} \quad \text{\texttt{false}} \quad \text{\texttt{pk}}
\]

MCS Queue-Based Algorithm

- Shared Qnode nodes[0..n-1]
- Shared Qnode *tail initially null
- Local Qnode *myNode, initially &nodes[i]
- Local Qnode *successor

acquire-lock

myNode->next := null // prepare to be last in queue
pred=swap(&tail, myNode) // tail now points to myNode
if (pred ≠ null) // should wait for a predecessor
  myNode->locked := true // prepare to wait
  pred->next := myNode // let predecessor know to unlock me
  wait until myNode.locked = false

release-lock

if (myNode.next = null) // not sure there is successor
  if (compare-and-swap(&tail, myNode, null) = false)
    wait until (myNode->next ≠ null) // wait for successor id
  successor := myNode->next // get pointer to successor
  successor->locked := false // unlock successor
else // for sure, there is successor
  successor := myNode->next // get pointer to successor
  successor->locked := false // unlock successor
MCS Queue-Based Algorithm

Shared Qnode nodes[0..n-1]
Shared Qnode *tail initially null
Local Qnode *myNode, initially &nodes[i]
Local Qnode *successor

acquire-lock
myNode->next := null               // prepare to be last in queue
pred=swap(&tail, myNode )         // tail now points to myNode
if (pred ≠ null)                   // should wait for a predecessor
    myNode->locked := true           // prepare to wait
    pred->next := myNode            // let predecessor know to unlock me
    wait until myNode.locked = false

release-lock
if (myNode.next = null)           // not sure there is successor
    if (compare-and-swap(&tail, myNode, null) = false)
        wait until (myNode->next ≠ null) // wait for successor id
        successor := myNode->next       // get pointer to successor
        successor->locked := false      // unlock successor
else                              // for sure, there is successor
    successor := myNode->next       // get pointer to successor
    successor->locked := false      // unlock successor

Uses swap and CAS
Is this algorithm local-spin? Yes
In the CC model? Yes
In the DSM model? Yes

Local-Spin Mutex
without Strong Primitives
Local-Spin Tournament-Tree Mutex

$O(\log n)$ RMR complexity for CC model (this is optimal)

$O(n \log n)$ registers

Key is to find the right 2-process mutex

Local-Spin 2-Process Mutex: 1st Try

Shared variables:
- Want[0], Want[1]: initially \bot
- Spin[0], Spin[1]: initially \bot

acquire-lock(side)
- Want[side] = 1 // announce
- Spin[side] = 0
- opponent = Want[1-side] // read other side
- if ( opponent <> \bot )
    - wait until ( Spin[side] <> 0 ) // spin

release-lock(side)
- Want[side] = \bot // cancel announcement
- Spin[1-side] = 1 // release other
Local-Spin 2-Process Mutex: 1st Try

Shared variables:
Want[0], Want[1]: initially \bot
Spin[0], Spin[1]: initially \bot

acquire-lock(side)
Want[side] = 1    // announce
Spin[side] = 0
opponent = Want[1-side]    // read other side
if ( opponent <> \bot )
   wait until ( Spin[side] <> 0 )  // spin
release-lock(side)
Want[side] = \bot    // cancel announcement
Spin[1-side] = 1 // release other

Ensures mutual exclusion
But may deadlock

Local-Spin 2-Process Mutex: Avoid Deadlock

Shared variables:
Tie, Want[0], Want[1]: initially \bot
Spin[0], Spin[1]: initially \bot

acquire-lock(side)
Want[side] = 1    // announce
Tie = i    // tie breaker
Spin[side] = 0
opponent = Want[1-side]    // read other side
if ( opponent <> \bot ) and ( Tie == i )
   if ( Spin[1-side] == 0 ) Spin[1-side] = 1
   wait until ( Spin[side] <> 0 )  // spin
   if ( Tie == i ) wait until ( Spin[side] > 1 )
release-lock(side)
Want[side] = \bot    // cancel announcement
if ( Tie <> i ) Spin[1-side] = 2 // release other
Local-Spin 2-Process Mutex: Avoid Deadlock

Shared variables:

Tie, Want[0], Want[1]: initially ⊥
Spin[0], Spin[1]: initially ⊥

acquire-lock(side)

Want[side] = 1    // announce
Tie = i    // tie breaker
Spin[side] = 0
opponent = Want[1-side]    // read other side
if ( opponent <> ⊥ ) and ( Tie == i )
  if ( Spin[1-side] == 0 ) Spin[1-side] = 1
  wait until ( Spin[side] <> 0 )  // spin
  if ( Tie == i ) wait until ( Spin[side] > 1 )

release-lock(side)

Want[side] = ⊥    // cancel announcement

if ( Tie <> i ) Spin[1-side] = 2 // release other

Local-Spin 2-Process Mutex

Shared variables:

Tie, Want[0], Want[1]: initially ⊥
Spin[0,…,n-1]: initially ⊥

acquire-lock(side)

Want[side] = i    // announce your identity
Tie = i    // tie breaker
Spin[i] = 0
opponent = Want[1-side]    // who's competing
if ( opponent <> ⊥ ) and ( Tie == i )
  if ( Spin[opponent] == 0 ) Spin[opponent] = 1
  wait until ( Spin[i] <> 0 )  // spin
  if ( Tie == i ) wait until ( Spin[i] > 1 )

release-lock(side)

Want[side] = nil
opponent = Tie    // who's competing
if ( opponent <> i ) Spin[opponent] = 2
Example (for processes 3 and 7)

Want[0] = 3
Tie = 3
Spin[3] = 0
opponent <> ⊥ and Tie <> 3
opponent = 7
CRITICAL
Spin[7] = 1
Spin[7] = 2

Want[1] = 7
Tie = 7
Spin[7] = 0
opponent <> ⊥ and Tie == 7
opponent = 3
CRITICAL
CRITICAL
WAIT until Spin[7] <> 0
CRITICAL
WAIT
CRITICAL
Wait until Spin[7] > 1
CRITICAL

Local-Spin 2-Process Mutex

Shared variables:
Tie, Want[0], Want[1]: initially ⊥
Spin[0,...,n-1]: initially ⊥

acquire-lock(side)
Want[side] = i
Tie = i
Spin[i] = 0
opponent = Want[1-side]
if (opponent <> nil) and (Tie == i)
if (Spin[opponent] == 0) Spin[opponent] = 1
wait until (Spin[i] <> 0)
if (Tie == i) wait until (Spin[i] > 1)

release-lock(side)
Want[side] = nil
opponent = Tie
if (opponent <> i) Spin[opponent] = 2

An array for each level
O(n log n) total

n registers per node?
Another Approach: Elevator Algorithm

When exiting the critical section, a process checks who’s waiting and promote them

Place in a queue of waiting processes
Ensure that waiting processes are promoted, and hence, not starved
Easier because promotion queue is handled in exclusion
Elevator Algorithm: Data Structures

- lock ∈ \{P_i, ⊥\}
- Promotion Queue
- apply[1...n]
- signal[1...n]
- exits
- inPromQ[1...n]

Elevator Algorithm: Exit section

- Promotion Queue
- q → r → s

Elevator Algorithm: Exit section

Exit section: Scenario 1
Exit section: Scenario 1

Promotion Queue

Exit section: Scenario 2

Promotion Queue
Exit section: Scenario 2

Promotion Queue

Elevator Algorithm: Entry Section

signal[i] := false  // initialize signal entry
apply[i] := true    // promote me
For each node n on the path from leaf to root
  n := i           // make yourself visible
if (not trylock(root))  // try to capture lock
  e := exits      // wait before re-trying
  wait until ((exits - e ≥ 2) or lock in {u, l})
  if (not trylock(root))  // try to capture lock
    await (signal[i])    // if failed, await signal
Elevator Algorithm: Entry Section

```plaintext
signal[i] := false  // initialize signal entry
apply[i] := true   // promote me
For each node n on the path from leaf to root
    n := i         // make yourself visible
if (not trylock(root)) // try to capture lock
    e := exits    // wait before re-trying
    wait until (exits – e ≥ 2) or lock ∈ {u, ¬}
if (not trylock(root)) // try to capture lock
    await (signal[i]) // if failed, await signal
```

What if we remove these lines?

Elevator Algorithm: Exit Section

```plaintext
apply[i] := false // not applying for help
exits := exits+1  // increment exits counter
for each node n on the path from root to i’s leaf
    // promote processes along path
    q1 := n, q2 := n.left, q3 := n.right
    for q in {q1,q2,q3} // parent and both children
        if apply[q] and ¬inPromQ[q]
            promQ.enqueue(q), inPromQ[q] := true
if promQ.isEmpty() freeLock
else
    next := Q.dequeue()
    inPromQ[next] := false
    lock := next // hand lock to promoted process
    signal[next] := true
```
Elevator Algorithm: Exit Section

apply[i] := false // not applying for help
exits := exits+1 // increment exits counter
foreach node n on the path from root to i’s leaf
  // promote processes along path
  q1 := n, q2 := n.left, q3 := n.right
  for q in {q1,q2,q3} // parent and both children
    if apply[q] and ¬inPromQ[q]
      promQ.enqueue(q), inPromQ[q] := true
  if promQ.isEmpty() freeLock
  else
    next := Q.dequeue()
    inPromQ[next] := false
    lock := next // hand lock to promoted process
    signal[next] := true

Argument for starvation-freedom

Why p is guaranteed to be promoted?
Argument for starvation-freedom

Assume a set of processes starve in some execution.
For a starved process $p$, let $l_p$ be the highest level in which $p$ is visible.

Let $l_p$ be non-decreasing, and stabilizes for all starved processes after some execution prefix $E$. 

Argument for starvation-freedom

Prove by induction on $l_p$ (w.r.t $E$) that processes do not starve.

Level 0

Level 1

Level 2

Level 3

Assume the claim holds for all levels up to $k$ and let $l_p = k + 1$
Argument for starvation-freedom

Assume the claim holds for all levels up to $k$ and let $l_p = k+1$
$p$'s writes before tryLock

Let $q$ be the last process to overwrite $p$ on level $k$
Argument for starvation-freedom

By induction, q cannot be starved and hence, it promotes p in its exit section

Optimizing for No Contention

In a well-designed system, most of the time only a single process wants the critical section...

In the algorithms so far, requires O(f(n)) steps:

- O(n) for the Bakery algorithm
- O(log(n)) for the tournament tree algorithm
Fast Mutex

Algorithm is **fast** if a process enters CS in $O(1)$ steps, when there is no competition

Must use multi-writer shared variables

Detecting Contention: Splitter

A process wins if it is alone in the splitter
$O(1)$ step complexity
Splitter Implementation: Race Variable

Shared variable: race, initially -1
1. \( \text{race} = \text{id}_i \)
2. if \( \text{race} == \text{id}_i \) then win
3. else lose

If a process is alone, clearly wins

But it is possible that two processes win

\[
\begin{align*}
p_1: & \quad \text{write} \quad \text{race} = 1 \\
& \quad \text{read} \quad \text{race} == 1
\end{align*}
\]

\[
\begin{align*}
p_2: & \quad \text{write} \quad \text{race} = 2 \\
& \quad \text{read} \quad \text{race} == 2
\end{align*}
\]

Doorway Mechanism

- Wrap a doorway mechanism around race
- Only a process in the first set of processes to concurrently access race may win
- After writing to race, check the doorway and if open, close it
- race chooses a unique one of the captured processes to "win"
Splitter Implementation

Shared variables
  door, initially false
  race, initially -1

1. race = id_i // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == id_i) // check race variable
   then return( win )
5. else return( lose )

Requires ≤ 5 read / write operations, and two shared registers.
Splitter Implementation: Doorway

Shared variables

door, initially false
race, initially -1

1. race = idᵢ // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == idᵢ) // check race variable
   then return( win )
5. else return( lose )

Correctness of the Splitter

A process wins when executing the splitter by itself

Simply the code when there is no concurrency

1. race = idᵢ // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == idᵢ) // check race variable
   then return( win )
5. else return( lose )
Correctness of the Splitter

At most one process wins the splitter

P: processes that read false from door (Line 2)

pj: last process to write to race
before door is set to true

No process \( p_i \neq p_j \) can win:
– \( p_i \notin P \) loses in Line 2.
– \( p_i \in P \) writes to race before \( p_j \) but checks again (Line 5) after \( p_j \)'s write and loses

1. \( race = id_i \) // write your identifier
2. if door then return( lose )
3. door = true
4. if (race == id_i ) // check race variable
   then return( win )
5. else return( lose )

©Hagit Attiya 2018 Mutual Exclusion 137

Correctness of the Splitter

At most one process wins the splitter

P: processes that read false from door (Line 2)

pj: last process to write to race
before door is set to true

No process \( p_i \neq p_j \) can win:
– \( p_i \notin P \) loses in Line 2.
– \( p_i \in P \) writes to race before \( p_j \) but checks again (Line 5) after \( p_j \)'s write and loses

\( p_i \)
write race = i
set door
read race == i

\( p_j \)
write race = j
read door
read race == j

©Hagit Attiya 2018 Mutual Exclusion 138
Ensuring No Deadlock

In case of concurrency, it is possible that no process wins the splitter

- Nodes losing the splitter enter $n$-process mutex
- Winner of $n$-process mutex competes with winner of splitter using 2-process mutex
- Winner enters CS

Exiting Fast Mutex

- Nodes losing the splitter enter 2-process exit
- Winner of splitter uses 2-process mutex

Exiting Fast Mutex

- n-process exit
  - was slow
  - fast
- 2-process exit
  - was fast
- n-process exit
  - was slow
  - was fast

2-process exit

Releasing the Splitter: Take 1

- \( p_1 \) and \( p_2 \) both lose the splitter
  - But “close” it
- Leave to the remainder
- \( p_3 \) arrives alone
  - Splitter still closed

- Release splitter
  - n-process exit
  - was slow
  - fast
  - 2-process exit
  - critical section

- must release splitter also after slow path

© Hagit Attiya
Releasing the Splitter: Take 2

A process from the slow path may release the splitter while another process is in the fast path.

Releasing the Splitter: Take 3

Different release code for processes from the fast path and the slow path.
Fast Mutex: Overall Structure

2-process exit

n-process exit

reset inside flag if none inside, open door

open door reset inside flag

was slow was fast

was fast was slow

splitter

& set inside flag

win

lose

play role of p0

play role of p1

reset inside flag

if none inside, open door

reset inside flag

2-process exit

2-process mutex

n-process mutex

n-process exit

Long-Lived Fast Mutex

Shared variables:
race: initially \( \perp \)
doors: initially false
inside[0,...,n-1]: all initially false

procedure slow-path
15: <n-process entry code>
16: <2-process entry code (1)>
17: <critical section>

19: inside[i] = false
20: if for all j, inside[j] == false
21: door = false

23: <2-process exit code (1)>
24: <n-process exit code> & exit

1: race = id
2: inside[i] = true
3: if door == true slow-path()
4: door = true
5: if race == id
6: <2-process entry code (0)>
7: <critical section>
8: door = false
9: inside[i] = false
10: <2-process exit code (0)>
11: else slow-path()
### Long-Lived Fast Mutex

**Shared variables:**
- race: initially $\bot$
- door: initially false
- inside[0,...,n-1]: all initially false
- checking: initially false

**procedure slow-path**
- n-process entry code
- <2-process entry code (0)>
- <critical section>
- door = false
- checking = true
- inside[i] = false
- if for all j, inside[j] == false
- door = false
- checking = false
- <2-process exit code (1)>
- n-process exit code & exit

**Additional Code**
1: race = id
2: inside[i] = true
3: if door == true slow-path()
4: door = true
5: if race == id
6: <2-process entry code (0)>
7: <critical section>
8: door = false
9: inside[i] = false
10: <2-process exit code (0)>
11: else slow-path()
### Long-Lived Fast Mutex: Correctness

At each time, at most one process is in Lines 6-10
\( \Rightarrow \) Mutual exclusion and no starvation

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>race = id</td>
</tr>
<tr>
<td>2</td>
<td>inside[i] = true</td>
</tr>
<tr>
<td>3</td>
<td>if door or checking == true slow-path()</td>
</tr>
<tr>
<td>4</td>
<td>door = true</td>
</tr>
<tr>
<td>5</td>
<td>if race == id</td>
</tr>
<tr>
<td>6</td>
<td>&lt;2-process entry code (0)&gt;</td>
</tr>
<tr>
<td>7</td>
<td>&lt;critical section&gt;</td>
</tr>
<tr>
<td>8</td>
<td>door = false</td>
</tr>
<tr>
<td>9</td>
<td>inside[i] = false</td>
</tr>
<tr>
<td>10</td>
<td>&lt;2-process exit code (0)&gt;</td>
</tr>
<tr>
<td>11</td>
<td>else slow-path()</td>
</tr>
</tbody>
</table>

\[\] procedure slow-path
  15: <n-process entry code>
  16: <2-process entry code (1)>
  17: <critical section>
  18: checking = true
  19: inside\[i\] = false
  20: if for all j, inside\[j\] == false
  21: door = false
  22: checking = false
  23: <2-process exit code (1)>
  24: <n-process exit code> & exit

At each time, at most one process is in Lines 16-23
\( \Rightarrow \) checking == true when a process is in Lines 19-21

### Long-Lived Fast Mutex: Complexity

A process running solo executes \( O(1) \) steps
Otherwise, execute \( O(n) \) steps (regardless of \( n \)-process mutex algorithm)

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>race = id</td>
</tr>
<tr>
<td>2</td>
<td>inside[i] = true</td>
</tr>
<tr>
<td>3</td>
<td>if door or checking == true slow-path()</td>
</tr>
<tr>
<td>4</td>
<td>door = true</td>
</tr>
<tr>
<td>5</td>
<td>if race == id</td>
</tr>
<tr>
<td>6</td>
<td>&lt;2-process entry code (0)&gt;</td>
</tr>
<tr>
<td>7</td>
<td>&lt;critical section&gt;</td>
</tr>
<tr>
<td>8</td>
<td>door = false</td>
</tr>
<tr>
<td>9</td>
<td>inside[i] = false</td>
</tr>
<tr>
<td>10</td>
<td>&lt;2-process exit code (0)&gt;</td>
</tr>
<tr>
<td>11</td>
<td>else slow-path()</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
</table>
| 15   | <n-process entry code>
| 16   | <2-process entry code (1)>
| 17   | <critical section>
| 18   | checking = true
| 19   | inside\[i\] = false
| 20   | if for all j, inside\[j\] == false
| 21   | door = false
| 22   | checking = false
| 23   | <2-process exit code (1)>
| 24   | <n-process exit code> & exit

©Hagit Attiya