SPL (Simple Programming Language)
Syntax

Basic Statements

- **skip**
- **assignment**
  \[
  (u_1, \ldots, u_k) \leftarrow (e_1, \ldots, e_k)
  \]
  \[
  \text{variables} \quad \text{expressions}
  \]
- **await** \(c\)
  
  (where \(c\) is a boolean expression)

  special case: \(\text{halt} \equiv \text{await } F\)

- Communication by message-passing
  
  \[
  \alpha \leftarrow e \quad \text{(send)}
  \]
  
  \[
  \alpha \Rightarrow u \quad \text{(receive)}
  \]
  
  (where \(\alpha\) is a channel)

- Semaphore operations
  
  \[
  \text{request } r \quad (r > 0 \rightarrow r \leftarrow r - 1)
  \]
  
  \[
  \text{release } r \quad (r \leftarrow r + 1)
  \]
  
  (where \(r\) is an integer variable)
SPL (CON’T)

Schematic Statements

In Mutual-Exclusion programs:

- **noncritical**
  may not terminate
- **critical**
  terminates

In Producer-Consumer programs:

- **produce** \( x \)
  terminates – assign nonzero value to \( x \)
- **consume** \( y \)
  terminates

No program variables are modified by schematic statements. One exception:

“\( x \)” in **produce** \( x \)
SPL (CON’T)

Compound Statements

- **Conditional**
  
  \[
  \text{if } c \text{ then } S_1 \text{ else } S_2 \\
  \text{if } c \text{ then } S
  \]

- **Concatenation**
  
  \[
  S_1; \cdots; S_k
  \]

  Example:
  
  \[
  \text{when } c \text{ do } S \equiv \text{await } c; S
  \]

- **Selection**
  
  \[
  S_1 \text{ or } \cdots \text{ or } S_k
  \]

- **while**
  
  \[
  \text{while } c \text{ do } S
  \]

  Example:
  
  \[
  \text{loop forever do } S \equiv \text{while } \top \text{ do } S
  \]

SPL (CON’T)

Compound Statements (Con’t)

- **Cooperation Statement**
  
  \[
  \ell: [\ell_1: S_1; \hat{\ell}_1:] \parallel \cdots \parallel [\ell_k: S_k; \hat{\ell}_k:]; \hat{\ell}:
  \]

  \[S_1, \ldots, S_k\] are parallel to one another interleaved execution.

  entry step: from \(\ell\) to \(\ell_1, \ell_2, \ldots, \ell_k\),

  exit step: from \(\hat{\ell}_1, \hat{\ell}_2, \ldots, \hat{\ell}_k\) to \(\hat{\ell}\).

- **Block**

  \[
  [\text{local declaration}; \ S]
  \]

  local variable, \ldots, variable : type where \(\varphi_i\)

  \[
  y_1 = e_1, \ldots, y_n = e_n
  \]
SPL (CON’T)

Basic types – boolean, integer, character, …

Structured types – array, list, set, …

Static variable initialization
(variables get initialized at the start of the execution)

Programs

\[
P :: [\text{declaration}; \quad P_1 :: [\ell_1: S_1; \  \hat{\ell}_1: ] \ || \ \cdots \ || \ P_k :: [\ell_k: S_k; \ \hat{\ell}_k: ]]
\]

\[P_1, \ldots, P_k\] are top-level processes

Variables in \( P \) called program variables

Declaration

\[
\text{mode } \{\text{variable, } \ldots, \text{ variable}\} : \text{ type } \quad \text{where } \varphi_i \\
\downarrow \quad \downarrow
\]

in (not modified)
\text{local}
\text{out}

\[\varphi_1 \land \ldots \land \varphi_n\] data-precondition of the program
Channel Declaration

- synchronous channels
  (no buffering capacity)
  
  \[ \text{mode } \alpha_1, \alpha_2, \ldots, \alpha_n: \text{channel of type} \]

- asynchronous channels
  (unbounded buffering capacity)
  
  \[ \text{mode } \alpha_1, \alpha_2, \ldots, \alpha_n: \text{channel [1..] of type} \]
  \[ \text{where } \varphi_i \]
  
  - \( \varphi_i \) is optional
  
  - \( \varphi_i = \Lambda \) (empty list) by default

Foundations for SPL Semantics

Labels

\[ \ell : S \]

- Label \( \ell \) identifies statement \( S \)

- Equivalence Relation \( \sim_L \) between labels:
  
  - For \( \ell \): \( [\ell_1: S_1; \ldots; \ell_k: S_k] \)
    \[ \ell \sim_L \ell_1 \]
  
  - For \( \ell \): \( [\ell_1: S_1 \text{ or } \ldots \text{ or } \ell_k: S_k] \)
    \[ \ell \sim_L \ell_1 \sim_L \cdots \sim_L \ell_k \]
  
  - For \( \ell \): \( [\text{local declaration}; \ell_1: S_1] \)
    \[ \ell \sim_L \ell_1 \]
Note: For $\ell: [\ell_1 : S_1]|| \ldots || \ell_k : S_k$

$\ell \not\sim_L \ell_1 \not\sim_L \ell_2 \not\sim_L \ldots$

because of the entry step

Example: In Figure 0.1

$\ell_0 \sim_L \ell_1$

$\ell_2 \sim_L \ell_3 \sim_L \ell_5$

in $a, b : \text{integer where } a > 0, b > 0$
local $y_1, y_2: \text{integer where } y_1 = a, y_2 = b$
out $g : \text{integer}$

\[
\begin{align*}
\ell_1: \text{while } y_1 \neq y_2 \text{ do} \\
\ell_2: \text{if } y_1 > y_2 \text{ then } y_1 := y_1 - y_2 \\
\ell_3: \text{await } y_1 > y_2 \\
\ell_4: y_1 := y_1 - y_2 \\
\ell_5: \text{if } y_2 > y_1 \text{ then } y_2 := y_2 - y_1 \\
\ell_6: y_2 := y_2 - y_1 \\
\ell_7: g := y_1 \\
\ell_8: 
\end{align*}
\]

Figure 0.1

A Fully Labeled Program GCD-F
Locations

\[ [\ell] \]

Identify site of control

- \([\ell]\) is the location corresponding to label \(\ell\).
- Multiple labels identifying different statements may identify the same location.

\[ [\ell] = \{\ell' \mid \ell' \sim_L \ell\} \]

Example: Fig 0.1: A fully labeled program

\[
\begin{align*}
[\ell_0] &= [\ell_1] = \{\ell_0, \ell_1\} & [\ell_6] &= \{\ell_6\} \\
[\ell_2] &= \{\ell_2, \ell_3, \ell_5\} & [\ell_7] &= \{\ell_7\} \\
[\ell_4] &= \{\ell_4\} & [\ell_8] &= \{\ell_8\}
\end{align*}
\]

Example: Fig 0.2: A partially labeled program

\[
\begin{align*}
\ell_0 \\
\ell_3 \rightarrow \ell_2^a \\
\ell_5 \rightarrow \ell_2^b \\
\text{shortcut: } \text{label } \ell_2 \text{ “represents” } \{\ell_2, \ell_2^a, \ell_2^b\}
\end{align*}
\]

\[
\begin{align*}
\text{in} & \quad a, b : \text{integer where } a > 0, \ b > 0 \\
\text{local} & \quad y_1, y_2 : \text{integer where } y_1 = a, \ y_2 = b \\
\text{out} & \quad g : \text{integer}
\end{align*}
\]

\[
\begin{array}{c}
\ell_1: \text{while } y_1 \neq y_2 \text{ do} \\
\ell_2: \quad \left[ \ell_2^a: \text{await } y_1 > y_2; \ \ell_4: \ y_1 := y_1 - y_2 \right] \\
\quad \text{or} \\
\ell_2^b: \text{await } y_2 > y_1; \ \ell_6: \ y_2 := y_2 - y_1 \\
\ell_7: \ g := y_1 \\
\ell_8:
\end{array}
\]

Figure 0.2

A Partially Labeled Program GCD
Post Location

\[ \ell: S; \hat{\ell}: \quad \text{post}(S) = [\hat{\ell}] \]

• For \([\ell_1: S_1; \hat{\ell_1}: ] \parallel \cdots \parallel [\ell_k: S_k; \hat{\ell_k}: ]\]

\[ \text{post}(S_i) = [\hat{\ell_i}], \text{for every } i = 1, \ldots, k \]

• For \(S = [\ell_1: S_1; \ldots; \ell_k: S_k]\)

\[ \text{post}(S_i) = [\ell_{i+1}], \text{for } i = 1, \ldots, k-1 \]

\[ \text{post}(S_k) = \text{post}(S) \]

• For \(S = [\ell_1: S_1 \text{ or } \cdots \text{ or } \ell_k: S_k]\)

\[ \text{post}(S_1) = \cdots = \text{post}(S_k) = \text{post}(S) \]

• For \(S = [\text{if } c \ \text{then } S_1 \ \text{else } S_2]\)

\[ \text{post}(S_1) = \text{post}(S_2) = \text{post}(S) \]

• For \([\ell : \text{while } c \ \text{do } S']\)

\[ \text{post}(S') = [\ell] \]

Example: Post Locations of Fig 0.2

\[ \text{post}(\ell_1) = [\ell_7] \]

\[ \text{post}(\ell_2) = \text{post}(\ell_4) = \text{post}(\ell_6) = [\ell_1] \]

\[ \text{post}(\ell^a_2) = [\ell_4] \]

\[ \text{post}(\ell^b_2) = [\ell_6] \]

\[ \text{post}(\ell_7) = [\ell_8] \]
Ancestor

$S$ is an ancestor of $S'$
if $S'$ is a substatement of $S$

$S$ is a common ancestor of $S_1$ and $S_2$
if it is an ancestor of both $S_1$ and $S_2$

$S$ is a least common ancestor (LCA) of $S_1$ and $S_2$
if $S$ is a common ancestor of $S_1$ and $S_2$
and any other common ancestor
of $S_1$ and $S_2$ is an ancestor of $S$

LCA is unique for given statements $S_1$ and $S_2$

Example: $[S_1; [S_2\|S_3]; S_4] \parallel S_5$

<table>
<thead>
<tr>
<th>Statements</th>
<th>LCA</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_2$ parallel to $S_3$</td>
<td>$S_2 \parallel S_3$</td>
</tr>
<tr>
<td>$S_2$ parallel to $S_5$</td>
<td>$S$</td>
</tr>
<tr>
<td>$S_2$ not parallel to $S_4$</td>
<td>$[S_1; \cdots; S_4] \text{ not coop.}$</td>
</tr>
<tr>
<td>$S_2$ not parallel to $S_2 \parallel S_3$</td>
<td>$S_2 \parallel S_3 \text{ same}$</td>
</tr>
</tbody>
</table>

Parallel Labels

$\bullet$ Statements $S$ and $\tilde{S}$ are parallel if
their LCA is a cooperation statement
that is different from statements $S$ and $\tilde{S}$

Example: $S = [S_1; [S_2\|S_3]; S_4] \parallel S_5$

$\bullet$ parallel labels – labels of parallel statements
Conflicting Labels

conflicting labels – not equivalent and not parallel

Example:

\[
\begin{align*}
\ell_1 &: S_1; \\
\ell_2 &: ([\ell_3: S_3; \hat{\ell}_3:] \parallel [\ell_4: S_4; \hat{\ell}_4:]); \parallel [\ell_6: S_6; \hat{\ell}_6:] \\
\ell_5 &: S_5; \hat{\ell}_5;
\end{align*}
\]

\(\ell_3\) is parallel to each of \(\{\ell_4, \hat{\ell}_4, \ell_6, \hat{\ell}_6\}\) and in conflict with each of \(\{\ell_1, \ell_2, \hat{\ell}_3, \ell_5, \hat{\ell}_5\}\)

\(\ell_6\) and \(\hat{\ell}_6\) are in conflict with each other but are parallel to each of \(\{\ell_1, \ell_2, \ell_3, \hat{\ell}_3, \ell_4, \hat{\ell}_4, \ell_5, \hat{\ell}_5\}\)

Critical References

Writing References:

\[
x := \ldots \alpha \Rightarrow u \text{ produce } x \text{ request } r \uparrow \uparrow \uparrow \uparrow \text{ release } r \uparrow
\]

Reading References: all other references

critical reference of a variable in \(S\) if:

- writing ref to a variable that has reading or writing refs in \(S'\) (parallel to \(S\))
- reading reference to a variable that has writing references in \(S'\) (parallel to \(S\))
- reference to a channel
**Limited Critical References (LCR)**

Statement obeys LCR restriction (LCR-Statement) if each test (for await, conditional, while) and entire statement (for assignment) contains at most one critical reference.

**Example:** Fig 0.3

\[ \ell_2, m_1, m_3 \text{ are LCR-Statements} \]

\[ \ell_1, m_2 \text{ violate the LCR-requirement} \]

**LCR-Program:** only LCR-statements

**Interleaved vs. Concurrent Execution**

**Claim:** If \( P \) is an LCR program, then the interleaving computations of \( P \) and the concurrent executions of \( P \) give the same results.

**Discussion & explanation:** *Blue Book*.

\[
\begin{align*}
P_1 &:: \\
\ell_1: &\; b := b \cdot y_1 \\
\ell_2: &\; y_1 := y_1 - 1 \\
\ell_3: &
\end{align*}
||
\begin{align*}
P_2 &:: \\
m_1: &\; \text{await} \; y_1 + y_2 \leq n \\
m_2: &\; b := b / y_2 \\
m_3: &\; y_2 := y_2 + 1 \\
m_4: &
\end{align*}
\]

Figure 0.3

Critical references
Given an SPL-program $P$, we can construct the corresponding FTS $\Phi = \langle V, \Theta, T, J, C \rangle$:

- **system variables** $V$
  
  $Y = \{y_1, \ldots, y_n\}$ – program variables of $P$
  
  domains: as declared in $P$
  
  $\pi$ – control variable
  
  domain: sets of locations in $P$

  $V = Y \cup \{\pi\}$

**Comments:**

- For label $\ell$, $at_{-\ell}$: $[\ell] \in \pi$
  
  $at'_{-\ell}$: $[\ell] \in \pi'$

**Note:** When going from an SPL program to an FTS we lose the sequential nature of the program. We need to model control explicitly in the FTS: $\pi$ can be viewed as a program counter.
SPL Semantics (Con’t)

Example: Fig 0.1

\[ V = \{ \pi, a, b, y_1, y_2, g \} \]
\[ \pi \text{ - ranges over subsets of } \{[\ell_1], [\ell_2], [\ell_4], [\ell_6], [\ell_7], [\ell_8]\} \]
\[ a, b, \ldots, g \text{ - range over integers} \]

- Initial Condition \( \Theta \)

For \( P :: [\text{dec}; [P_1 :: [\ell_1: S_1; \hat{\ell}_1: ]] \| \cdots \| P_k :: [\ell_k: S_k; \hat{\ell}_k: ] ] \]

with data-precondition \( \varphi \),
\[ \Theta: \pi = \{[\ell_1], \ldots, [\ell_k]\} \land \varphi \]

Example: Fig 0.1

\[ \Theta: \pi = \{[\ell_1]\} \land \]
\[ a > 0 \land b > 0 \land y_1 = a \land y_2 = b \]

data-precondition

\[ \text{in } a, b : \text{integer where } a > 0, b > 0 \]
\[ \text{local } y_1, y_2 : \text{integer where } y_1 = a, y_2 = b \]
\[ \text{out } g : \text{integer} \]

\[ [\ell_1: \text{while } y_1 \neq y_2 \text{ do} \]
\[ [\ell_2: \text{if } y_1 > y_2 \text{ then } \ell_4: y_1 := y_1 - y_2 \]
\[ \text{or} \]
\[ [\ell_2: \text{if } y_2 > y_1 \text{ then } \ell_6: y_2 := y_2 - y_1 \]
\[ \ell_7: g := y_1 \]
\[ \ell_8: \]

Figure 0.2

A Partially Labeled Program GCD
SPL Semantics (Con’t)

• Transitions $\mathcal{T}$

$$\mathcal{T} = \{\tau_I\} \cup \left\{ \text{transitions associated with the statements of } P \right\}$$

where $\tau_I$ is the “idling transition”

$\rho_I: V' = V$

abbreviation

– $\text{pres}(U): \bigwedge_{u \in U} (u' = u)$ (where $U \subseteq V$) the value of $u \in U$ are preserved

– $\text{move}(L, \hat{L}): L \subseteq \pi \land \pi' = (\pi - L) \cup \hat{L}$

where $L, \hat{L}$ are sets of locations

– $\text{move}(\ell, \hat{\ell}): \text{move}(\{[\ell]\}, \{[\hat{\ell}]\})$
SPL Semantics (Con’t)

Basic Statements (Con’t)

ℓ: \text{await } c; \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land c \land \text{pres}(Y)

ℓ: \text{request } r; \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land r > 0 \land r' = r - 1 \land \text{pres}(Y - \{r\})

ℓ: \text{release } r; \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land r' = r + 1 \land \text{pres}(Y - \{r\})

asynchronous send

ℓ: \alpha \leftarrow e; \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land \alpha' = \alpha \cdot e \land \text{pres}(Y - \{\alpha\})

asynchronous receive

ℓ: \alpha \Rightarrow u; \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land |\alpha| > 0 \land \alpha = u' \cdot \alpha' \land \text{pres}(Y - \{u, \alpha\})

synchronous send-receive

ℓ: \alpha \leftarrow e; \hat{\ell}: \quad m: \alpha \Rightarrow u; \hat{m}:

\text{move}(\{\ell, m\}, \{\hat{\ell}, \hat{m}\}) \land u' = e \land \text{pres}(Y - \{u\})
SPL Semantics (Con’t)

Schematic Statements

\( \rho_\ell \)

\( \ell: \) noncritical; \( \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land \text{pres}(Y) \)

(nontermination modeled by \( \tau_\ell \notin J \))

\( \ell: \) critical; \( \hat{\ell}: \rightarrow \text{move}(\ell, \hat{\ell}) \land \text{pres}(Y) \)

Compound Statements

\( \ell: \) [if \( c \) then \( \ell_1: S_1 \) else \( \ell_2: S_2 \)]; \( \hat{\ell}: \rightarrow \)

\( \rho_\ell: \rho_\ell^T \lor \rho_\ell^F \) where

\( \rho_\ell^T: \text{move}(\ell, \ell_1) \land c \land \text{pres}(Y) \)

\( \rho_\ell^F: \text{move}(\ell, \ell_2) \land \neg c \land \text{pres}(Y) \)

\( \ell: \) [while \( c \) do \( \tilde{\ell}: \tilde{S} \)]; \( \hat{\ell}: \rightarrow \)

\( \rho_\ell: \rho_\ell^T \lor \rho_\ell^F \) where

\( \rho_\ell^T: \text{move}(\ell, \tilde{\ell}) \land c \land \text{pres}(Y) \)

\( \rho_\ell^F: \text{move}(\ell, \hat{\ell}) \land \neg c \land \text{pres}(Y) \)

\( \ell: \) \( [\ell_1: S_1; \hat{\ell}_1:] \parallel \cdots \parallel [\ell_k: S_k; \hat{\ell}_k:] \); \( \hat{\ell}: \rightarrow \)

\( \rho_\ell^E: \text{move}(\{\ell\}, \{\ell_1, \ldots, \ell_k\}) \land \text{pres}(Y) \) (entry)

\( \rho_\ell^X: \text{move}(\{\hat{\ell}_1, \ldots, \hat{\ell}_k\}, \{\hat{\ell}\}) \land \text{pres}(Y) \) (exit)
Grouped Statements \( (S) \)
executed in a single atomic step

Example:
\[ \langle x := y + 1; z := 2x + 1 \rangle \]
\[ x' = y + 1 \land z' = 2y + 3 \]
the same as \( (x, z) := (y + 1, 2y + 3) \)

Example:
\[ \langle a := 3; a := 5 \rangle \]
\[ a' = 5 \]
\[ a = 3 \] is never visible to the outside world, nor to other processes

SPL Semantics (Con’t)

- Justice Set \( \mathcal{J} \)
  All transitions except \( \tau_I \) and all transitions associated with noncritical statements

- Compassion Set \( \mathcal{C} \)
  All transitions associated with send, receive, request statements
Computations of Programs

local $x$: integer where $x = 1$

$P_1 ::$

$$
\begin{array}{c}
\ell_0: \\
\ell_0^a: \text{await } x = 1 \\
\text{or} \\
\ell_0^b: \text{skip} \\
\ell_1:
\end{array}
$$

$\parallel P_2 ::$

$$
\begin{array}{c}
m_0: \text{while } T \text{ do} \\
\ell_1: \\
\end{array}
$$

$\ell_1^a: [m_1: x := \ell_1^b: \text{skip}]$

$\ell_1^b: \text{await } x = 1$

$P_1 ::$

$$
\begin{array}{c}
\ell_0: \\
\ell_0^a: \text{await } x = 1 \\
\text{or} \\
\ell_0^b: \text{skip} \\
\ell_1:
\end{array}
$$

$\parallel P_2 ::$

$$
\begin{array}{c}
m_0: \text{while } T \text{ do} \\
\ell_1: \\
\end{array}
$$

$\ell_1^a: [m_1: x := \ell_1^b: \text{skip}]$

$\ell_1^b: \text{await } x \neq 1$

\[\sigma: \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: 1 \rangle \xrightarrow{m_1} \langle \pi: \{\ell_0, m_0\}, x: -1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: -1 \rangle \xrightarrow{m_1} \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \cdots \]

$\sigma$ is not a computation. Unjust towards $\ell_0^b$

(enabled on all states but never taken)

\[\sigma: \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: 1 \rangle \xrightarrow{m_1} \langle \pi: \{\ell_0, m_0\}, x: -1 \rangle \xrightarrow{m_0} \langle \pi: \{\ell_0, m_1\}, x: -1 \rangle \xrightarrow{m_1} \langle \pi: \{\ell_0, m_0\}, x: 1 \rangle \xrightarrow{m_0} \cdots \]

$\sigma$ is a computation –

since none of the just transitions are continually enabled.

Fig 0.4 Process $P_1$ terminates in all computations.

Fig 0.5 $\text{skip } \rightarrow \text{await } x \neq 1$
Computations of Programs (Con’t)

local $x$: integer where $x = 1$

$$P_1 :: \begin{cases}
\ell_0: \text{if } x = 1 \text{ then } \\
\ell_1: \text{skip } \\
\ell_2: \text{skip } \\
\ell_3: \text{else } \\
\end{cases} \parallel P_2 :: \begin{cases}
m_0: \text{while } T \text{ do } \\
m_1: x := -x \\
\end{cases}$$

Fig 0.6 Process $P_1$ terminates in all computations.

Control Configurations

$L = \{[\ell_1], \ldots, [\ell_k]\}$ of $P$ is called conflict-free
if no $[\ell_i]$ conflicts with $[\ell_j]$, for $i \neq j$.

$L$ is called a (control) configuration of $P$
if it is a maximal conflict-free set.

Example:

local $x$: integer where $x = 0$

$$P_1 :: \begin{cases}
\ell_0: x := 1 \\
\ell_1: \\
\end{cases} \parallel P_2 :: \begin{cases}
m_0: \text{await } x = 1 \\
m_1: \text{ } \\
\end{cases}$$

Configurations

$\{[\ell_0],[m_0]\}$, $\{[\ell_0],[m_1]\}$,
$\{[\ell_1],[m_0]\}$, $\{[\ell_1],[m_1]\}$

$\sigma$ is not a computation –
since $\ell_0$ is continually enabled,
but not taken.
SPL Semantics (Con’t)

accessible configuration – appears as value of $\pi$ in some accessible state

Example:

$\{[\ell_0],[m_1]\}$ does not appear in any accessible state

Is a given configuration accessible? Undecidable

The Mutual-Exclusion Problem

loop forever do

\[
\begin{bmatrix}
\text{noncritical} \\
\hdots \\
\text{critical} \\
\hdots \\
\end{bmatrix}
\]

||

\[
\begin{bmatrix}
\text{noncritical} \\
\hdots \\
\text{critical} \\
\hdots \\
\end{bmatrix}
\]

Requirements:

- **Exclusion**
  While one of the processes is in its critical section, the other is not

- **Accessibility**
  Whenever a process is at the noncritical section exit, it must eventually reach its critical section

Example: mutual exclusion by semaphores

Fig. 0.7
local $y$: integer where $y = 1$

$P1$

$P2$

Fig. 0.7 Program MUX-SEM

Message-Passing Programs

Example: Producer-Consumer

assumption:

channel $send \leq N$ values

Fig. 0.9

Example: Producer-Consumer

assumption:

channel $send \leq N$ values

Fig. 0.9 Program PROD-CONS