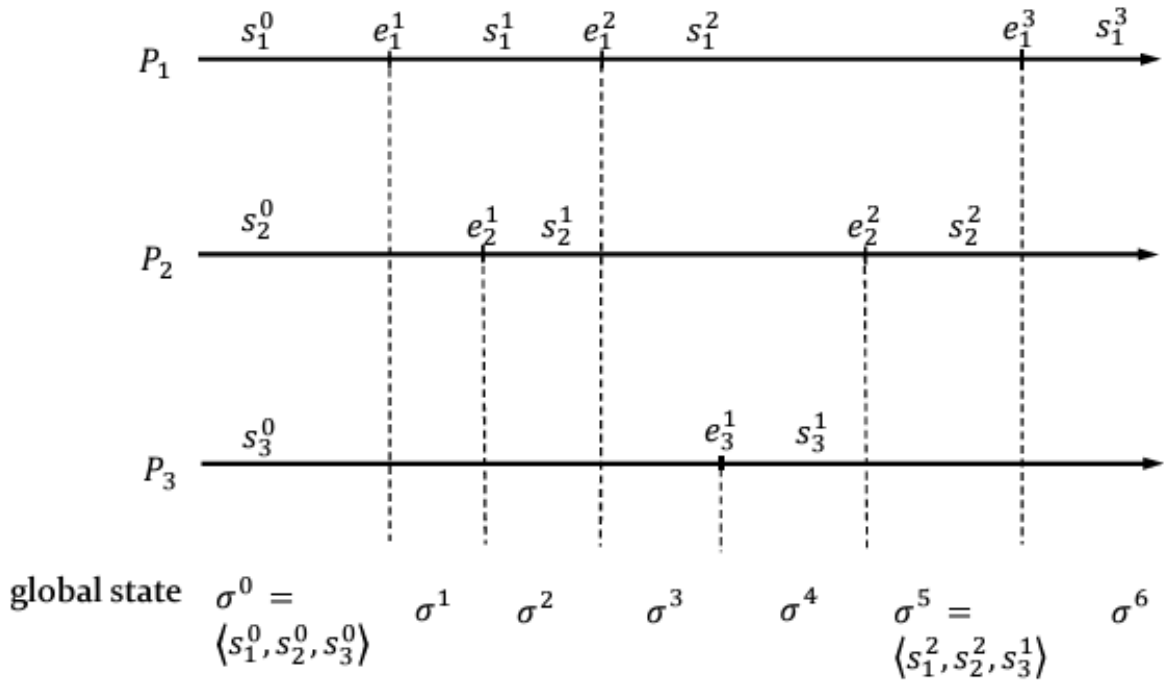
# Concurrency

## Concurrent system model

- A concurrent processing consists of a set of sequential processes (or threads) and shared resources.
- The sequential process is a sequence of states.
- A state of a process is it’s memory contents and registers contents.
- The change of a process state is the effect of instruction execution and is called event (or action)
- The instruction to be executed next depends on the current state and whether the event subsequent to the execution is admissible.
- Atomic instruction — an operation that is indivisible and uninterruptible, meaning it is completed in a single step relative to other threads or processes.

# States & events




# Indeterminism

- Because computations in a concurrent system can interact with each other while being executed, the number of possible execution paths in the system can be extremely large, and the resulting outcome can be indeterminate.
- For each process there is exactly one admissible event that can happen. But there are many admissible events in the entire system.
- Only one of those event will happen depending on the available resources and scheduler's decision. Therefore the next event is unpredictable.

# Interleaving and state reachability

- Interleaving is an ordering of all events in the concurrent system which preserves the local order of events of individual processes. In other words if an event $a$ happens before the event $b$ in one of the processes, then in the

global sequence of events $a$ must also come before $b$. It might not come right before $b$, but the order of local events will be preserved in the global ordering.

- A global state $\sigma'$ is reachable from a global state $\sigma$ (denoted $\sigma \rightarrow \sigma'$) if:
  - they are the same state $\sigma = \sigma'$
  - there is an admissible event $e$ in $\sigma$ resulting in $\sigma'$
  - there is a path of admissible events that lead from $\sigma$ to $\sigma'$
- Example:

```
n: integer := 0; /* shared variable */
```

| instructions \ threads | thread A | thread B |
|---|---|---|
| high-level lang. | `n := n + 1` | `n := n + 1` |
| RISC processor | `load R_A, n`<br>`add R_A, 1`<br>`store R_A, n` | `load R_B, n`<br>`add R_B, 1`<br>`store R_B, n` |
| CISC processor | `inc n` | `inc n` |

Code for a thread A and a thread B

| | interleaving 1 | interleaving 2 |
|---|---|---|
| flow of control | `{A} load R_A, n`<br>`{A} add R_A, 1`<br>`{A} store R_A, n`<br>`// n = 1`<br>`{B} load R_B, n`<br>`{B} add R_B, 1`<br>`{B} store R_B, n` | `{A} load R_A, n`<br>`{A} add R_A, 1`<br>`// n = 0`<br>`{B} load R_B, n`<br>`{B} add R_B, 1`<br>`{A} store R_A, n`<br>`{B} store R_B, n` |
| final value of *n* | 2 | 1 |

Example of two interleavings of the same code

# Mutual exclusion

- Problem formulation:
  - The system consists of $n$ processes, $P_0, P_1, \ldots, P_{n-1}$.
  - The program of each process has a code segment, called critical section
  - Critical section code contains access operations to shared resources, in practise shared data
  - Critical section can be executed by at most one process at a time
- General structure of mutual exclusion algorithm:
  1. Remainder section: this is the part of the code where the process of thread performs operations that do not require mutual exclusion.

2. Entry section: it contains the code that a thread must execute to request access to the critical section. It typically involves some mechanism (e.g., locks, semaphores, flags) to check and set the conditions for entering the critical section.
3. Critical section: this is the part of the code where the shared resources are accessed or modified.
4. Exit section: it contains the code that a thread executes to release control of the critical section.
5. Repeat from 1

- For a mutual exclusion algorithm to be correct the following properties must be fulfilled:
  - Mutual exclusion — no more than one process can execute in its critical section at one time.
  - Progress — if there are some processes in their entry sections, then one of them is eventually allowed to enter the critical section.
  - Lockout-freedom — if any process tries to enter its critical section, then that process must eventually succeed. This property is also termed „bounded waiting" or „freedom from individual starvation".

# Bakery algorithm

- Bakery algorithm is mutual exclusion algorithm designed to prevent concurrent threads entering critical sections of code concurrently to eliminate the risk of data corruption.
- The Bakery algorithm is one of the simplest known solutions to the mutual exclusion problem for the general case of $N$ processes.

```
# Global shared variables
choosing = [False] * n
number = [0] * n

# Method for locking. i is the number of the thread
```

```
def lock(i):
    choosing[i] = True
    number[i] = max(number) + 1
    choosing[i] = False
    for j in range(n):
        if j == i: continue
        while choosing[j]:
            wait()
        while number[j] != 0 and (number[j], j) < (number[i], i
            wait()

# Method for unlocking
def unlock(i):
    number[i] = 0

# Then for every thread
def thread_func():
    lock(i);
    critical_section()
    unlock(i)
    remainder_section()
```

- Explanation:

  This algorithm works by assigning a number for each thread (process). A number 0 means that the process is done using the critical section, a number greater than 0 means that the process is waiting to get access to the critical section. The process with the lowest (non zero) number gets to the critical section first. It could happen that two numbers will have the same number assigned, this issue is resolved by giving higher priority to the process with the lower process identifier (`i` in my code). If a process wants to enter a critical section it gets assigned a number which is greater than any other processes' number. It then checks the number of every other process, waiting until all other processes are either done with the critical section (number equal to 0) or they have a smaller priority (their number is greater than ours). When a process is getting assigned a number, another variable called choosing

protects this operation, so that when a different process wants to check our processes number it must wait until we are done with the assignment operation (i.e. the choosing variable for our process is `False`).

# Mutual exclusion with test&set

- The test&set instruction is an instruction used to write (set) 1 to a memory location and return its old value as a single atomic operation.
- We can use this instruction to implement mutual exclusion:

```
shared bool_t lock = false;

while (test_and_set(&lock)) // entry section
    do_nothing();
critical_section();
lock = false;           // exit section
remainder_section();
```

  - If the lock variable has the value `true` when we wait until another process manually sets this variable to `false`. If it is `false` then we atomically set it to `true` (acquire the lock) and proceed to the critical section.
- This code guarantees the mutual exclusion property and the progress property, but it doesn't guarantee lockout-freedom since one process may always be the last one to try to lock the variable, and will never enter the critical section.

# Mutual exclusion with exchange

- The exchange instruction takes two boolean variables and exchanges their values atomically.

```
shared lock: Boolean := false;
local key: Boolean;
```

```
key := true;                // entry section
repeat                      // entry section
    exchange(lock, key);    // entry section
until key = false;          // entry section
critical_section();
lock := false;              // exit section
remainder_section();
```

- If the variable `lock` is set to `false` then our exchange operation atomically sets it to `true` and then we proceed to the critical section. If it's `true` then we wait until another process sets it to `false`.