Section 2
Mutual Exclusion

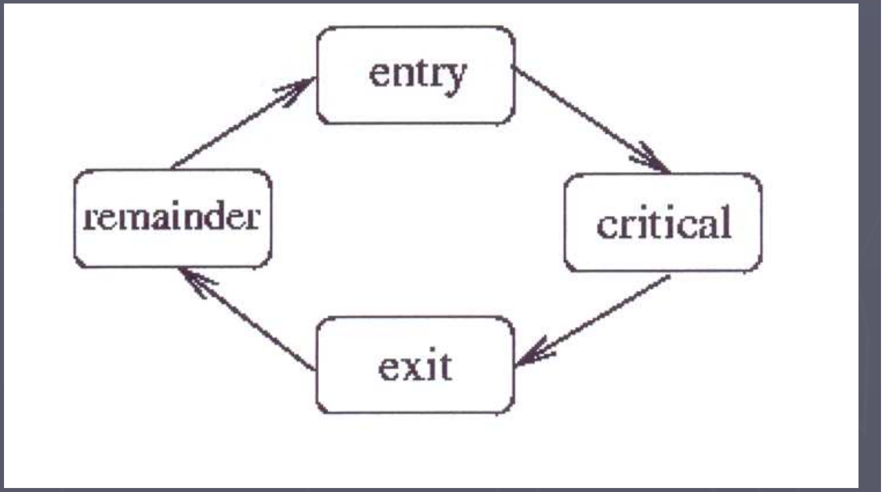
The Mutual Exclusion Problem

- The problem concerns a group of processors which occasionally need access to some resource that cannot be used simultaneously by more than a single processor.

- Examples of what the resource may be are:
  - The printer or any other output device
  - A record of a shared data base or a shared data structure, etc.

- Each processor may need to execute a code segment called critical section, such that at any time:
  - at most one processor is in the critical section
  - If one or more processors try to enter the critical section, then one of them eventually succeeds as long as no processor stays in the critical section forever.

The Mutual Exclusion Problem



Entry (Trying) Section: the code executed in preparation for entering the critical section

Critical Section: the code to be protected from concurrent execution

Exit Section: the code executed upon leaving the critical section

Remainder Section: the rest of the code

Each process cycles through these sections in the order: remainder, entry, critical, exit.

The problem is to design the entry and exit code in a way that guarantees that the mutual exclusion and deadlock-freedom properties are satisfied.

Mutual Exclusion Algorithms

Admissible Executions

- An execution is admissible if for every process p_i, p_i either takes an infinite number of steps or p_i ends in the remainder section.

- An algorithm solves the mutual exclusion problem if the following hold:
    - **Mutual Exclusion**

      In every configuration of every execution, at most one process is in the critical section.
    - **No Deadlock**

      In every execution, if some process is in the entry section in a configuration, then there is a later configuration in which some process is in the critical section.

- **Stronger Progress Property**
    - **No lockout (starvation-free)**

      In every execution, if some process is in the entry section in a configuration, then there is a later configuration in which that same process is in the critical section.

Mutual Exclusion Algorithms

Assumptions

- Any variable that is accessed in the entry or the exit section of the algorithm cannot be accessed in any of the other two sections.
- No process stays in the critical section forever.
- The exit section consists of a finite number of steps.

Useful Definitions

- A waiting process is a process that is busy-waiting on some condition in its entry code (i.e., it is waiting for some other process to do something that will enable it to proceed).
- The code before the waiting statement and the last statement before the waiting statement are sometimes called the doorway.
- **r-bounded-waiting**: A waiting process will be able to enter its critical section before each of the other processes is able to enter its critical section r+1 times.
- **Bounded Waiting**: There exists a positive integer r for which the algorithm is r-bounded waiting. That is, if a given process is in the entry section, then there is a bound on the number of times any other process is able to enter the critical section before the given process does so.
- Do r-bounded waiting and bounded waiting imply deadlock freedom?

- **Linear Waiting**: 1-bounded waiting
- **First-In-First-Out (FIFO)**: The term is used for 0-bounded waiting ⇒ FIFO guarantees that no beginning process can pass an already waiting process.

ME Algorithms that use RW Registers

Algorithms

- Algorithms for two processes
- An algorithm that guarantees mutual exclusion and no lockout but uses O(n) registers of unbounded size.
- An algorithm that guarantees mutual exclusion and no lockout using O(n) registers of bounded size.

Lower Bounds

- Any algorithm that provides mutual exclusion, even with the weak property of no deadlock, must use n distinct RW registers, regardless of the size of these registers.

Proposed solution I

Process p_0

```
while (true} {
        while (turn = 1) {skip} //entry
        critical section
        turn = 1                  // exit
        remainder section
}
```

Process p_1

```
while (true} {
      while (turn = 0) {skip}  //entry
      critical section
       turn = 0                  // exit
       remainder section
}
```

turn

0/1

✓mutual exclusion
X deadlock-freedom

Does it work?

Proposed Solution II

Process p_0

```
while (TRUE) {
        flag[0] = true
        while (flag[1]) {skip}
        critical section
        flag[0] = false
        remainder section
}
```

Process p_1

```
while (TRUE) {
        flag[1] = true
        while (flag[0]) {skip}
        critical section
        flag[1] = false
        remainder section
}
```

	flag
0	false
1	false

✓mutual exclusion
X deadlock-freedom

Does it work?

Transparency made by Gadi Taubenfeld – Synchronization Algorithms and Concurrent Programming

Proposed solution III

Process p_0

```
while (TRUE) {

    while (flag[1]) {skip}
     flag[0] = true
     critical section
     flag[0] = false
     remainder section
}
```

Process p_1

```
while (TRUE) {

    while (flag[0]) {skip}
     flag[1] = true
     critical section
     flag[1] = false
     remainder section
}
```

	flag
0	false
1	false

X mutual exclusion
✓Deadlock-freedom

Does it work?

Transparency made by Gadi Taubenfeld – Synchronization Algorithms and Concurrent Programming

Peterson's algorithm

Process p_0

```
While (TRUE) {
    flag[0] = true
    turn = 1
    while (flag[1] and turn == 1)
        {skip}
    critical section
    flag[0] = false
     remainder section
}
```

Process p_1

```
While (TRUE) {
    flag[1] = true
    turn = 0
    while (flag[0] and turn == 0)
        {skip}
    critical section
    flag[1] = false
     remainder section
}
```

	flag
0	false
1	false

turn 0/1

Transparency made by Gadi Taubenfeld – Synchronization Algorithms and Concurrent Programming

ME Algorithm using Single-Writer binary RW registers

want[0]: SW register written by p_0 and read by p_1 with initial value 0; it is set to 1 to identify that process p_0 wants to enter the critical section

want[1]: symmetric to want[0]

Process p0

```
while (TRUE) {


3.        want[0] = 1;


6.        wait until (want[1] == 0);
          critical section;
8.        want[0] = 0;
          remainder section;
}
```

Process p1

```
while (TRUE) {
1.        want[1] = 0;
2.        wait until (want[0] == 0);
3.        want[1] = 1;
4.        if (want[0] == 1) then
5.                  goto line 1

          critical section;
8.        want[1] = 0;
          remainder section;
}
```

Is this correct?

How can we prove it?

Proving Correctness

Theorem

- The algorithm ensures the mutual exclusion property.

Sketch of Proof

- Assume, by contradiction, that at some configuration C, both processes are in the critical section

  ⇒ want[0] = want[1] = 1.
- Case 1: Last write of p_0 to want[0] follows the **last** write of p_1 to want[1].

want[1] = 1

want[0] = 1

p_0 executes line 6 and does not enter the CS

both processes in critical section

Contradiction!

- Case 2: Last write of p_1 to want[1] follows the **last** write of p_0 to want[0].

want[0] = 1

want[1] = 1

p_1 executes line 5 and does not enter the CS

both processes in critical section

Contradiction!

☺ The algorithm ensures the no-deadlock property.

☹ The algorithm does not guarantee lockout freedom.

ME Algorithm using Single-Writer binary RW registers – Symmetric Version

Code for process p_i, i = 0, 1

```
while (TRUE) {
1:      want[i] = 0;
2:      wait until ((want[1-i] == 0) OR (priority == i));
3:      want[i] = 1;
4:      if (priority == 1-i) then {
5:                      if  (want[1-i] == 1) then
                                goto line 1; }
6:      else wait until (want[1-i] == 0);
        critical section;
7:      priority = 1-i;
8:      want[i] = 0;
        remainder section;
}
```

Proving the No-Deadlock Property

Theorem

- The algorithm ensures the no-deadlock property.

Sketch of Proof

- Suppose in contradiction that from some configuration on at least one process is forever in the entry section and no process enters the critical section.
- Case 1: Both processes are forever in the entry section.
  - The value of Priority does not change
  - Assume, wlog, that Priority = 0 (the case where Priority = 1 is symmetric).
  - By the code, it follows that one of the two processes cannot be stuck forever in the critical section! A contradiction!!!
- Case 2: Just one process is forever in the critical section (wlog, assume this holds for p_0) .
  - Critical and exit sections are bounded $\Rightarrow$ after some point want[1] = 0 forever.
  - By the code, it follows that process p_0 does not loop forever in the entry section! A contradiction!!!

Proving Lockout Freedom

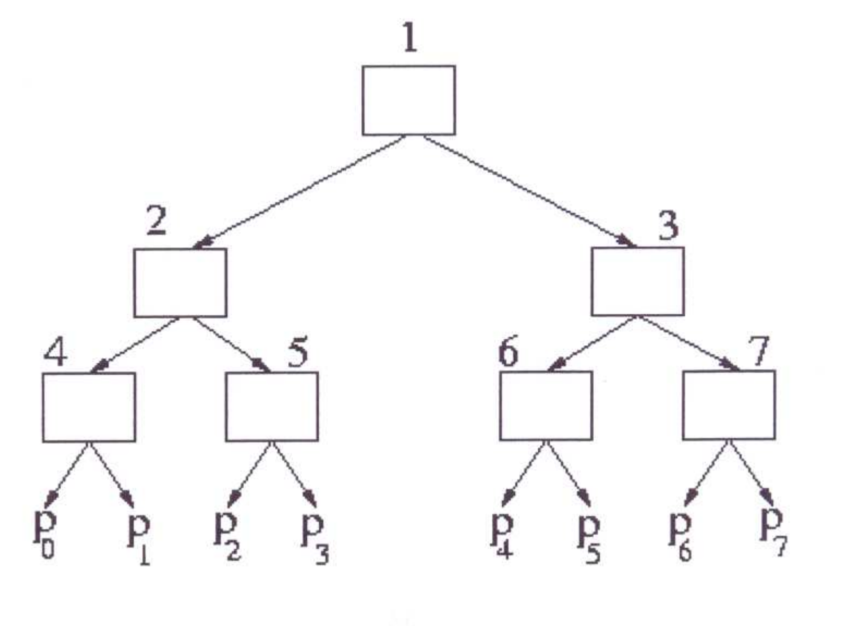
Theorem

- The algorithm ensures lockout freedom.

Sketch of Proof

- Assume, by way of contradiction, that some process (e.g., p_0) is starved $\Rightarrow$ from some configuration on p_0 is forever in the entry section.
- Case 1: Suppose p1 executes line 7 at some later point.
  - Priority = 0 forever after.
  - p_0 is stuck executing line 6
  - Thus, want[1] == 1 each time p_0 checks the condition of line 6. By the code, it follow that this cannot happen. A contradiction!
- Case 2: p1 never executes line 7 at any later point.
  - Since no-deadlock holds, p_1 is forever in the remainder section.
  - Thus, want[1] == 0 henceforth.
  - By the code, it follows that p_0 cannot be stuck in the entry section! A contradiction!!!

ME Algorithms for many processes



❑ Processes compete pairwise, using a two-process algorithm.

❑ The pairwise competitions are arranged in a complete binary tree.

❑ The tree is called the tournament tree.

❑ Each process begins at a specific leaf of the tree

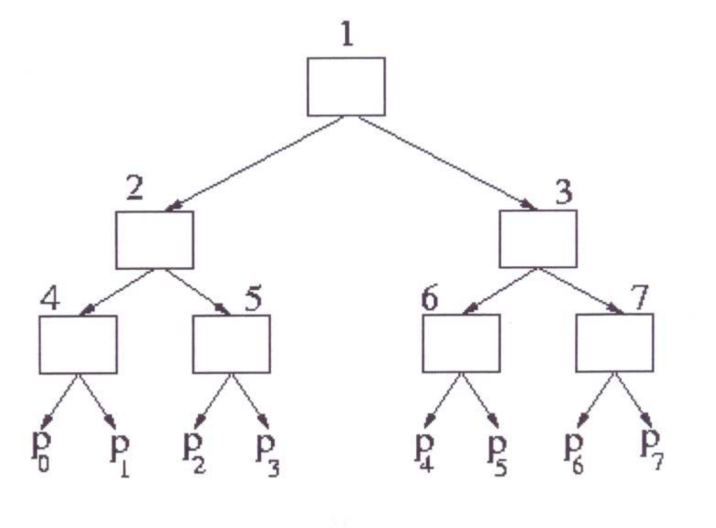
❑ At each level, the winner moves up to the next higher level, and competes with the winner of the competition on the other side.

❑ The process on the left side plays the role of p_0, while the process on the right side plays the role of p_1.

❑ The process that wins at the root enters the critical section.

ME Algorithms for many processes

```
procedure **Node**(v: integer, side: 0..1) {
    1: want$^v$[side] = 0;
    2: wait until ((want$^v$[1-side] == 0)
                  OR (priority$^v$ == side));
    3: want$^v$[side] = 1;
    4: if (priority$^v$ == 1-side) then {
    5:        if  (want$^v$[1-side] == 1) then
                  goto line 1; }
    6: else wait until (want$^v$[1-side] == 0);
    8: if (v == 1) then
    9:    critical section;
    10: else Node($\lfloor v/2 \rfloor$, v%2)
    11: priority$^v$ = 1-side;
    12: want$^v$[side] = 0;
}
```



❑ Tree nodes are numbered. The number of the root is 1. The number of the left child node of a node v is 2v, and the number of the right child of v is 2v+1.

❑ want^v[0], want^v[1], priority^v: variables associated to node v for the instance of 2-ME that is executed at this node.

❑ Process p_i begins by calling Node(2k+$\lfloor i/2 \rfloor$, i % 2), where k = $\lceil \log n \rceil$ -1

Tournament ME Algorithm: Correctness Proof

- **Projection** of an execution of the tree algorithm onto some node v
  - We only consider steps that are taken while executing the code in Node(v,0) and Node(v,1)

- **We will show the following:**
- For each node v, the projection of any execution of the tree algorithm onto v is an admissible execution of the symmetric mutual exclusion algorithm for 2 processes, if we view every process that executes Node(v,0) as p_0 and every process that executes Node(v,1) as p_1.

Tournament ME Algorithm: Correctness Proof

More formally:

- Fix an execution $a = C_0\ \varphi_1\ C_1\ \varphi_2\ C_2 \ldots$ of the tournament tree algorithm.
- Let a^v be the subsequence of alternating configurations and events

$$D_0\ \pi_1\ D_1\ \pi_2\ D_2 \ldots$$

defined inductively as follows:

Base Case: D_0 is the initial configuration of the 2 –processor algorithm

Induction Hypothesis: Assume that a^v has been defined up to configuration D_{i-1}.

Induction Step: Let $\varphi_j = k$ be the i-th event of a that is a step in Node(v,0) or Node(v,1) (suppose, wlog, that φ_j is a step in Node(v,0)).

- Let $\pi_i = 0$ (i.e., p_0 takes this step) and let D_i be a configuration such that:
  - The variables' states are those of the variables of node v in C_j
  - The state of p_1 is the same as in D_{i-1}
  - The state of p_0 is the same as the state of p_k in C_j except for the id being replaced with 0.

Process p4

```
procedure **Node**(6, 0) {
    1: $want^6[0] = 0$;                              φ1
    2: wait until (($want^6[1] == 0$)                φ2
               OR ($priority^6 == 0$));              φ3
    3: $want^6[0] = 1$;                              φ4
    4: if ($priority^6 == 1$) then {                 φ5
    5:     if ($want^6[1] == 1$) then goto line 1; }     φ6
    6: else wait until ($want^6[1] == 0$);               φ7
    8: if (6 == 1) then critical section;
       else Node(3, 0)

           $want^3[0] = 0$;                          φ8
           wait until (($want^3[1] == 0$) OR ($priority^3 == 0$));
                          φ9                     φ10

           $want^3[0] = 1$;                          φ14
           if ($priority^3 == 1$) then {             φ15
               if ($want^3[1] == 1$) then goto line 1; }   φ16
           else wait until ($want^3[1] == 0$);    φ17
           if (3 == 1) then critical section;
           else Node(1, 1)

               $want^1[1] = 0$;                      φ25
               wait until (($want^1[0] == 0$) OR ($priority^1 == 0$));
                          φ26                  φ27

               $want^1[1] = 1$;                      φ30
               if ($priority^1 == 1$) then {         φ31
                   if ($want^3[1] == 1$) then goto line 1; }   φ32
               else wait until ($want^3[1] == 0$);   φ33
               if (1 == 1) then critical section;
```

Process p7

procedure **Node**(7,1) {

a = C0, φ1, C1, φ2, C2, φ3, C3, φ4, C4, φ5, C5, φ6, C6, φ7, C7, φ8, C8, φ9, C9, φ10, C10, φ11, C11, φ12, C12, φ13, C13, φ14, C14, φ15, C15, φ16, C16, φ17, C17, φ18, C18, φ19, C19, φ20, C20, φ21, C21, φ22, C22, φ23, C23, φ24, C24, φ25, C25, φ26, C26, φ27, C27, φ28, C28, φ29, C29, φ31, C30, φ31, C31, φ32, C32, φ33, C33 …

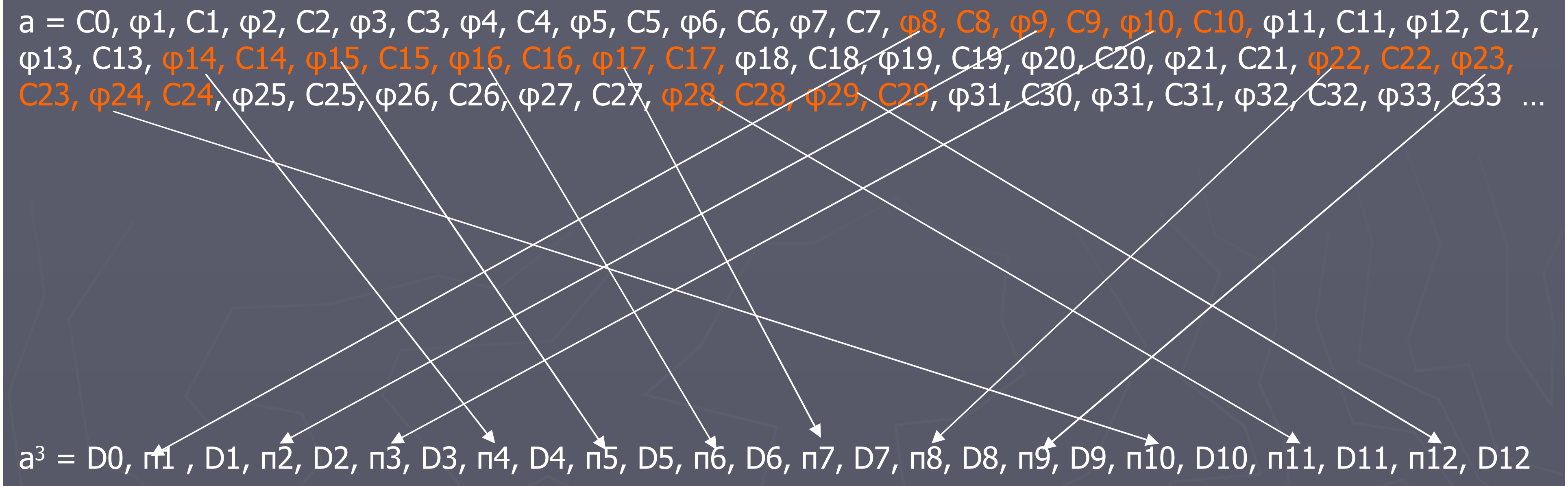
Orange events are steps of Node(3,0) or Node(3,1).

```
    1: $want^7[1] = 0$;                         φ11
    2: wait until (($want^7[0] == 0$) OR ($priority^7 == 1$));
                      φ12                   φ13

    3: $want^7[1] = 1$;                         φ18
    4: if ($priority^7 == 0$) then {            φ19
    5:          if ($want^7[0] == 1$) then goto line 1; } φ20
    6: else wait until ($want^7[0] == 0$);          φ21
    8: if (7 == 1) then critical section;
       else Node(3, 1)
           $want^3[1] = 0$;                          φ22
           wait until (($want^3[0] == 0$) OR ($priority^3 == 1$));
                          φ23                     φ24

           wait until (($want^3[0] == 0$) OR ($priority^3 == 1$));
                          φ28                     φ29
```

Tournament ME: Example Execution



a = C0, φ1, C1, φ2, C2, φ3, C3, φ4, C4, φ5, C5, φ6, C6, φ7, C7, φ8, C8, φ9, C9, φ10, C10, φ11, C11, φ12, C12, φ13, C13, φ14, C14, φ15, C15, φ16, C16, φ17, C17, φ18, C18, φ19, C19, φ20, C20, φ21, C21, φ22, C22, φ23, C23, φ24, C24, φ25, C25, φ26, C26, φ27, C27, φ28, C28, φ29, C29, φ31, C30, φ31, C31, φ32, C32, φ33, C33 …

a^3 = D0, π1 , D1, π2, D2, π3, D3, π4, D4, π5, D5, π6, D6, π7, D7, π8, D8, π9, D9, π10, D10, π11, D11, π12, D12

Tournament ME Algorithm: Correctness Proof

Lemma

For every v, a^v is an execution of the 2-process algorithm.

Proof

- The code of Node(v,i) and the code of the 2-process algorithm for p_i, $i = 0,1$, are the same.
- The only thing to check is that only one process performs instructions of Node(v,i) at a time. We prove this by induction on the level of v, starting at the leaves.

  Base Case: It holds by construction.

  Induction Hypothesis: Let v be any internal node of the tournament tree.

  Induction Step: We prove the claim for v.

  - If a process executes instructions of, e.g., Node(v,0), then it is in the critical section for v's left child.
  - By induction hypothesis and the fact that the 2-process algorithm guarantees mutual exclusion, only one process at a time is in the critical section for v's left child. -> The claim follows.
  - Similarly, only one process at a time executes instructions of Node(v,1).

Tournament ME Algorithm: Correctness Proof

Lemma

- For all v, if a is an admissible execution of the tournament algorithm, then a^v is an admissible execution of the 2-process algorithm.

Proof

- We prove that in a^v no process stays in the critical section forever.
- The proof is performed by induction on the level of v, starting from the root.

Theorem

- The Tournament Algorithm provides mutual exclusion.

Proof

- The restriction of any execution to the root of the tree is an admissible execution of the 2-process algorithm.
- Since this algorithm provides mutual exclusion, the Tournament algorithm also provides mutual exclusion.

The Bakery Algorithm

for each i, $0 \leq i \leq n-1$:

- Choosing[i]: it has the value TRUE as long as p_i is choosing a number
- Number[i]: the number chosen by p_i

Code for process p_i, $0 \leq i \leq n-1$

Initially, Number[i] = 0, και
Choosing[i] = FALSE, for each i, $0 \leq i \leq n-1$

```
Choosing[i] = TRUE;
Number[i] = max{Number[0], …, Number[n-1]}+1;
Choosing[i] = FALSE;
for j = 0 to n-1, j ≠ i, do
    wait until Choosing[j] == FALSE;
    wait until ((Number[j] == 0) OR ((Number[j], j) > (Number[i], i)));

critical section;

Number[i] = 0;

remainder section;
```

The Bakery Algorithm

Lemma

- In every configuration C of any execution a, if p_i is in the critical section, and for some k ≠ i, Number[k] ≠ 0, then (Number[k],k) > (Number[i],i).

Sketch of Proof

- Number[i] > 0
- p_i has finished the execution of the for loop (in particular, the 2nd wait statement for j = k).
- **Case 1**: p_i read that Number[k] == 0
- **Case 2**: p_i read (Number[k],k) > (Number[i],i)

Theorem

- The Bakery algorithm ensures the mutual exclusion property.

The Bakery Algorithm

Theorem

- The Bakery algorithm provides no lockout.

Sketch of proof

- Assume, by the way of contradiction, that there is a starved process.
- All processes wishing to enter the critical section eventually finish choosing a number.
- Let p_j be the process with the smallest (Number[j],j) that is starved.
- All processes entering the critical section after p_j has chosen its number will choose greater numbers, and therefore will not enter the critical section before p_j.
- Each process p_k with Number[k] < Number[j] will enter the critical section and exit it.
- Then, p_j will pass all tests in the for loop and enter the critical section.

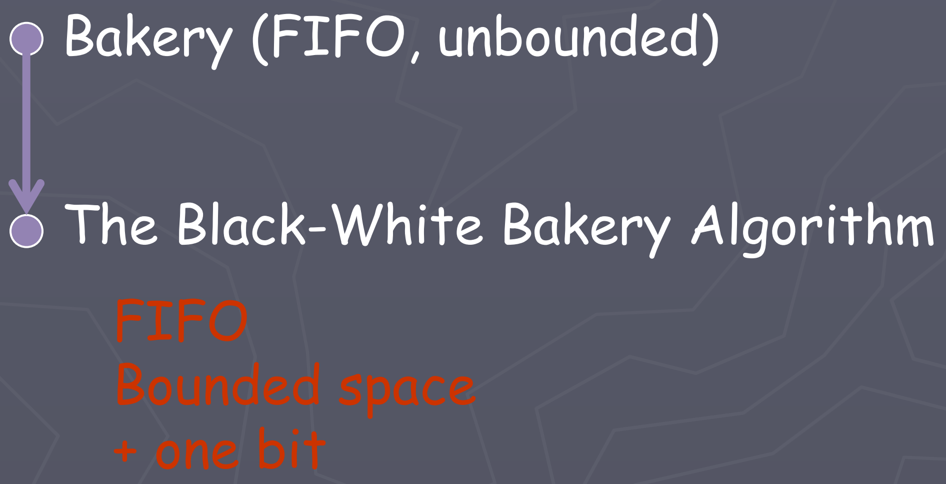
Space Complexity

- The Bakery Algorithm uses 2n single-writer RW registers. The n Choosing[j] variables are binary, while the n Number[j] variables are unbounded, $0 \leq j \leq n-1$.

Bakery Algorithm versus

Properties of the Bakery Algorithm

- The Bakery Algorithm satisfies mutual exclusion & FIFO.
- The size of number[i] is <u>unbounded.</u>



○ Bakery (FIFO, unbounded)

○ The Black-White Bakery Algorithm

FIFO
Bounded space
+ one bit

The Black-White Bakery Algorithm

```
choosing[i] = true;
mycolor[i] = color;
number[i] = 1 + max{number[j] | (1 $\le$ j $\le$ n) $\wedge$ (mycolor[j] = mycolor[i])};
choosing[i] = false;
for j = 0 to n {
    await (choosing[j] == false);
    if (mycolor[j] == mycolor[i])
    then await (number[j] == 0) $\vee$ (number[j],j) $\ge$ (number[i],i) $\vee$
                    (mycolor[j] $\neq$ mycolor[i]);
    else await  (number[j] == 0) $\vee$ (mycolor[i] $\neq$ color) $\vee$
                    (mycolor[j] == mycolor[i]);
}
critical section;
if (mycolor[i] == black) then color = white;
else color = black;
number[i] = 0;
```

Transparency made by Gadi Taubenfeld – Synchronization Algorithms and Concurrent Programming

Tight space bounds for mutual exclusion using atomic registers

- All mutual exclusion algorithms presented so far use at least n shared r/w registers. This is not an accident!
- Any mutual exclusion algorithm using only shared read-write registers must use at least n such registers.
- This is so:
  - even if we require the basic conditions – mutual exclusion and progress, and
  - regardless of the size of the registers.

Tight space bounds for mutual exclusion using r/w registers – Useful Definitions

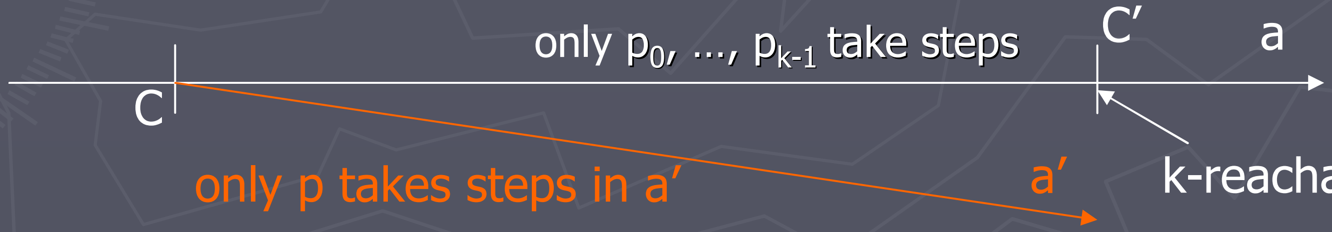
- A configuration C is called idle (or inactive) if no process is in the entry, critical or exit section at C.

all processes are in the
remainder section

- A process p covers some register R at some configuration C, if at its next step, p will perform a write into R, overwriting whatever was written in R before.

p's first step after C is performed just
before C' and it is a write into R

- For any k, $1 \le k \le n$, we say that a configuration C is *k-reachable* from another configuraion C' if there is an execution fragment starting from C and ending at C' which contains steps only of processes $p_0, \ldots, p_{k-1}$.
- An execution fragment a is called p-only if p is the only process taking steps in a. We say that a is S-only (where S is a set of processes) if only processes belonging to S take steps in a.



only $p_0, \ldots, p_{k-1}$ take steps

k-reachable from C

only p takes steps in a'

p-only execution fragment

Lower Bound - Useful Definitions

- The *schedule* of an execution a is the sequence of process indices that take steps in a (in the same order as in a).
- Example
  - a = C_0, i_1, C_1, i_2, C_2, i_3, …
  - $\sigma(a) = i_1, i_2, i_3, \ldots$
- A configuration C and a schedule σ uniquely determine an execution fragment which we denote by exec(C,σ).
- For each configuration C, let mem(C) = $(r_0, \ldots, r_{m-1})$ be the vector of register values in C
- A configuration C is similar with or indistinguishable from some other configuration C′ to some process set S, if each process of S is in the same state at C and C′ and mem(C) = mem(C′).

  If C is similar with C′ to S, we write $C \sim^S C'$.

Lower Bound – Simple Facts

- **Lemma 1**

  Suppose that C is a reachable idle configuration and let p_i be any process. Then, there is an execution fragment starting from C and involving steps of process p_i only, in which p_i enters the critical section.

- **Lemma 2**

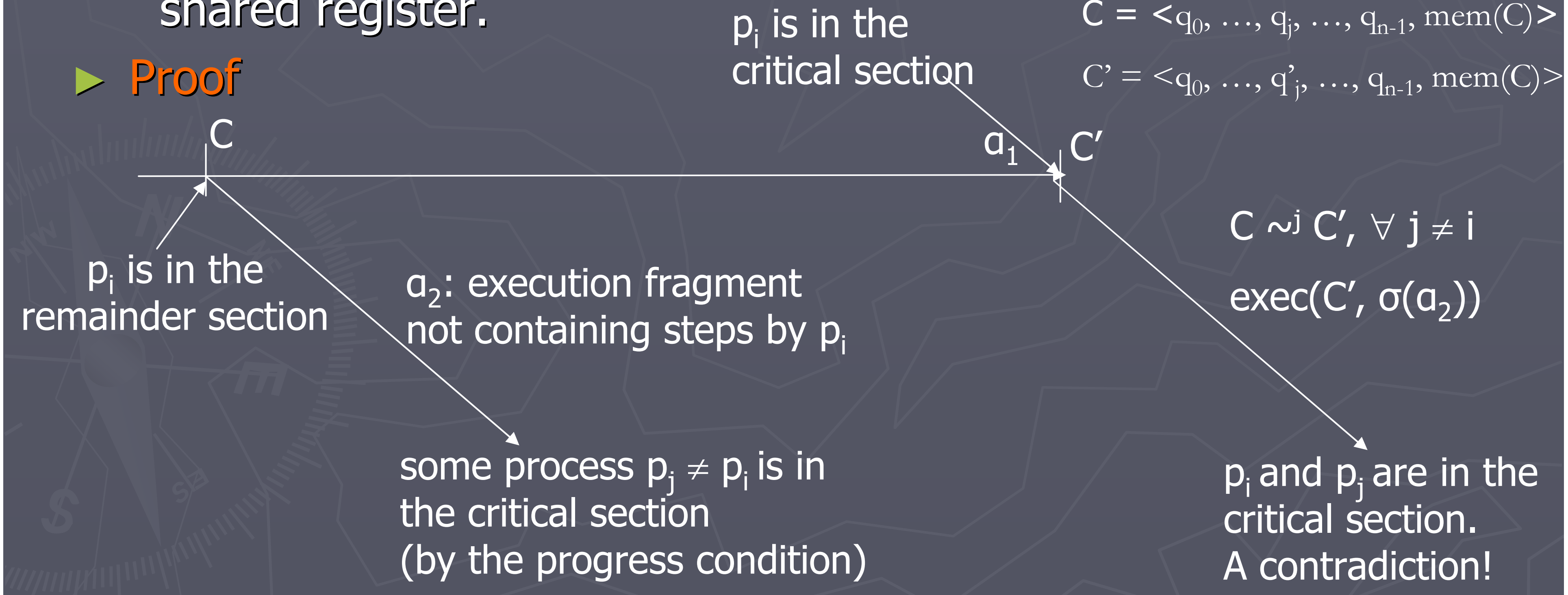
  Suppose that C and C′ are reachable configurations that are indistinguishable to some process p_i and suppose that C′ is an idle configuration. Then, there is an execution fragment starting from C and involving steps of process p_i only, in which p_i enters the critical section.

Lower Bound – Simple Facts

- Lemma 3

  Suppose that C is a reachable configuration where some process p_i is in the remainder section. Consider an execution fragment α_1 starting from C such that (1) α_1 involves steps of p_i only and (2) p_i is in the critical section in the final configuration of α_1. Then, α_1 contains a write by p_i to some shared register.

- Proof



p_i is in the critical section

$C = <q_0, \ldots, q_j, \ldots, q_{n-1}, mem(C)>$

$C' = <q_0, \ldots, q'_j, \ldots, q_{n-1}, mem(C)>$

C

α_1

C′

$C \sim^j C', \forall j \neq i$

p_i is in the remainder section

α_2: execution fragment not containing steps by p_i

exec(C′, $\sigma(\alpha_2)$)

some process $p_j \neq p_i$ is in the critical section (by the progress condition)

p_i and p_j are in the critical section. A contradiction!

Lower Bound

Definition

- A register is called single-writer if it can be written by only one process.

Theorem 1 (Lower Bound for Single-Writer Multi-Reader R/W Registers)

- If algorithm A solves the mutual exclusion problem for n > 1 processes, using only single-writer r/w shared registers, then A must use at least n shared registers.

Proof

- Immediate from Lemma 3

Theorem 2 (Lower Bound for Multi-Writer R/W Registers)

- If algorithm A solves the mutual exclusion problem for n > 1 processes, using only r/w shared registers, then A must use at least n shared registers.

Lower Bound

Lemma 4 (Generalized Version of Lemma 3)

- Let C be a reachable configuration in which process p_i is in the remainder section. Consider an execution fragment α_1 starting from C such that (1) α_1 involves steps of p_i only and (2) p_i is in the critical section in the final configuration of α_1. Then, α_1 contains a write by p_i to some shared register **that is not covered by any other process in C**.

Proof

Left as an exercise!

Lower Bound - Two processes

Theorem 2.1 (Special Case: just two processes)

- There is no algorithm that solves the mutual exclusion problem for two processes using only one R/W shared register.

Proof

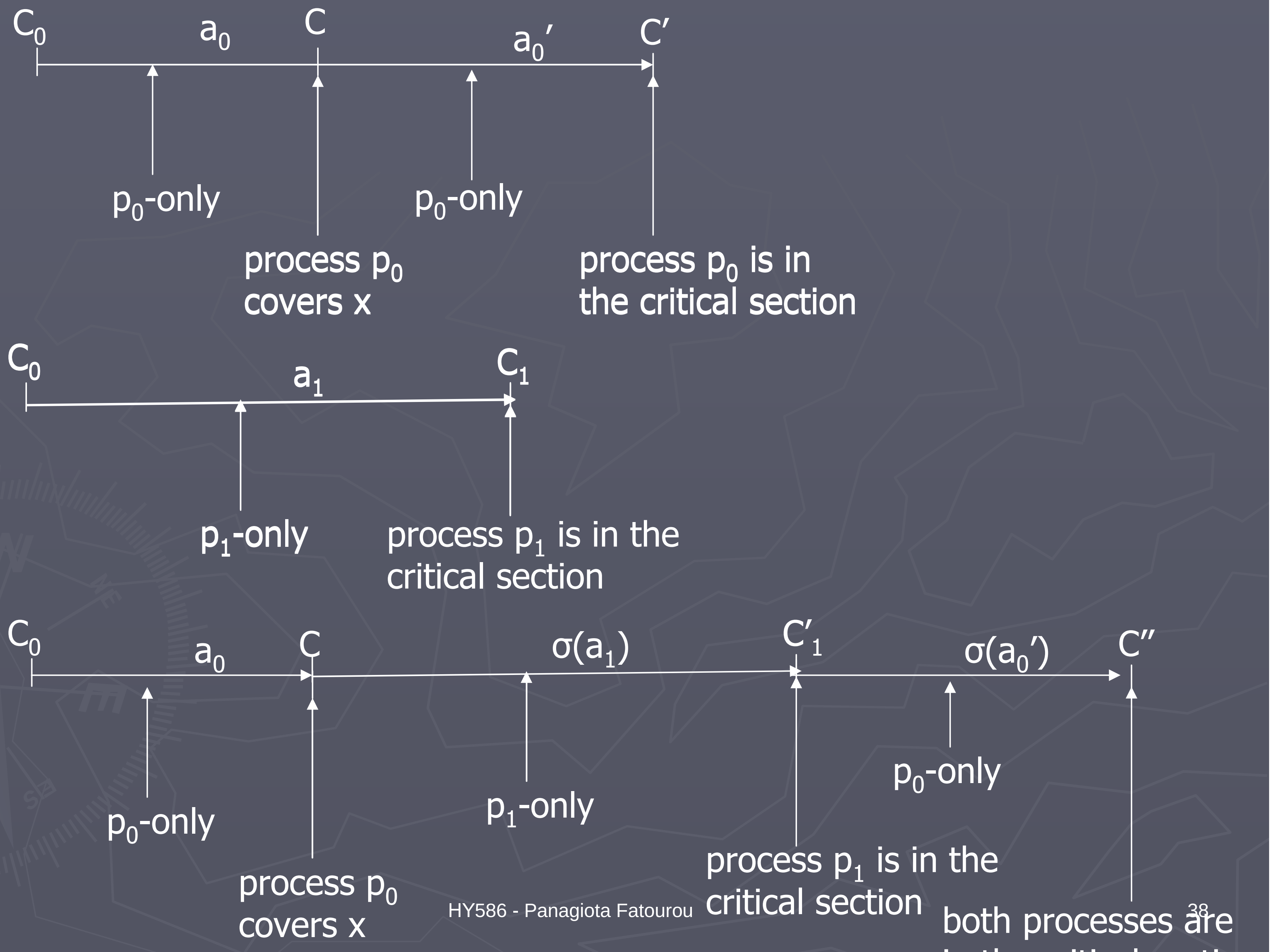
Assume, by contradiction, that A is such an algorithm.

Let x be the unique shared r/w register that it uses.

Denote by C_0 the initial state of the algorithm.

We construct an execution α that violates mutual exclusion!

Lower Bound - Two processes



C_0 a_0 C a_0' C'

p_0-only

process p_0 covers x

p_0-only

process p_0 is in the critical section

C_0 a_1 C_1

p_1-only

process p_1 is in the critical section

C_0 a_0 C $\sigma(a_1)$ C'_1 $\sigma(a_0')$ C''

p_0-only

process p_0 covers x

p_1-only

process p_1 is in the critical section

p_0-only

both processes are in the critical section

Lower Bound - Three processes

Theorem 2.2 (Special Case: three processes)

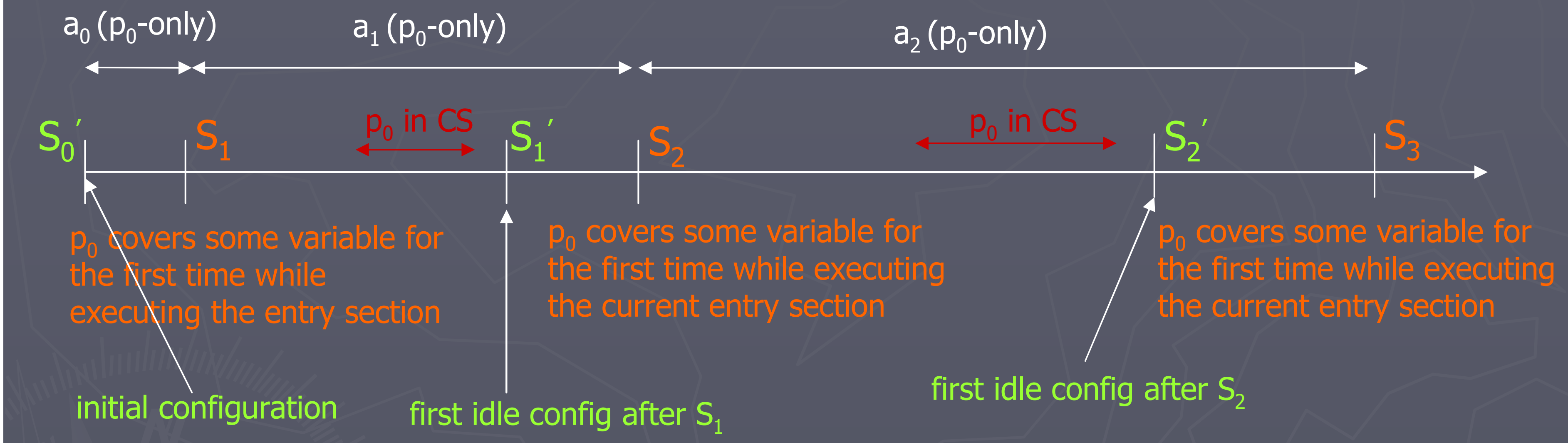
- There is no algorithm that solves the mutual exclusion problem for three processes using only two R/W shared register.

Proof

- Assume, by contradiction, that A is such an algorithm.
- Let x, y be the shared r/w registers that it uses.
- We construct an execution α that violates mutual exclusion!
- Strategy
  1. Starting from C_0, we will maneuver processes p_0 and p_1 to a point where each covers one of the two variables x and y. Moreover, the resulting configuration C′ will be indistinguishable to process p_2 from some reachable idle state.
  2. We run process p_2 on its own from C′ until it reaches the critical section.
  3. We let each of processes p_0 and p_1 take a step. Since each covers one of the two variables, they can eliminate all traces of process p_2’s execution.
  4. Then, we let p_0 and p_1 continue taking steps until one of them enters the critical section.
  5. At this point we have two processes in the critical section, which is a contradiction!

Lower Bound - Three processes

How can we construct an execution such that at its final configuration C_2 processes p_0 and p_1 cover both registers x and y, yet C′ is indistinguishable to an idle configuration to p_2?



a_0 (p_0-only)

a_1 (p_0-only)

a_2 (p_0-only)

S_0' S_1 p_0 in CS S_1' S_2 p_0 in CS S_2' S_3

p_0 covers some variable for the first time while executing the entry section

p_0 covers some variable for the first time while executing the current entry section

p_0 covers some variable for the first time while executing the current entry section

initial configuration

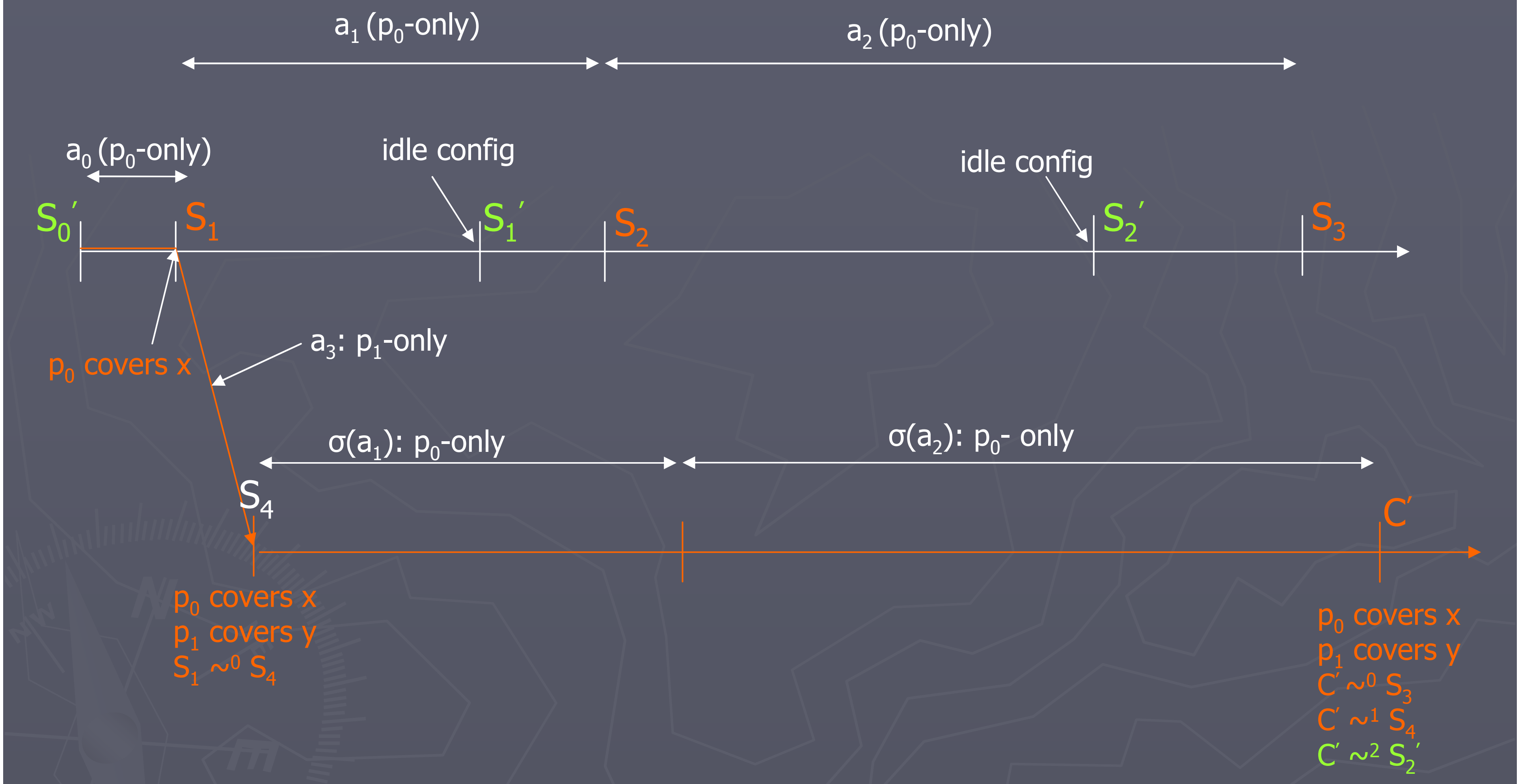
first idle config after S_1

first idle config after S_2

In two out of the three configurations S_1, S_2, S_3, process p_0 covers the same register. Wlog, assume that in S_1 and S_3, p_0 covers register x. Let $S_1' = C_0$.

If we run p_1 alone starting from S_1, p_1 will enter its critical section since $S_1 \sim^1 S_0'$. By Lemma 4, in this execution, p_1 writes to y.

Lower Bound - Three processes



a_1 (p_0-only)

a_2 (p_0-only)

a_0 (p_0-only)

idle config

idle config

S_0' S_1 S_1' S_2 S_2' S_3

p_0 covers x

a_3: p_1-only

$\sigma(a_1)$: p_0-only

$\sigma(a_2)$: p_0- only

S_4

C'

p_0 covers x
p_1 covers y
$S_1 \sim^0 S_4$

p_0 covers x
p_1 covers y
$C' \sim^0 S_3$
$C' \sim^1 S_4$
$C' \sim^2 S_2'$

C' is the configuration at which (1) p_0 and p_1 cover x and y, respectively, and (2) C' is indistinguishable from an idle reachable configuration (S_2') to p_2.

We now apply steps 2,3,4 and 5 of our strategy to derive a contradiction!

Lower Bound – The General Case

Lemma 5

Suppose A solves the mutual exclusion problem for n > 1 processes using exactly n-1 r/w shared registers. Let C be any reachable idle configuration. Suppose $1 \leq k \leq n-1$. Then, there are two configurations C′ and C″, each k-reachable from C, satisfying the following properties:

1. k distinct registers are covered by processes $p_0, \ldots, p_{k-1}$ in C′,
2. C″ is an idle configuraton
3. $C' \sim^i C''$, for all i, $k \leq i \leq n-1$

Proof: By induction on k.

Base Case: We run process p_0 alone until it first covers a shared register. Let C′ be the resulting configuration and $C'' = C_0$. Then, all properties hold.

Induction Step: Natural generalization of the proof of Theorem 2.2, where similar arguments as those for proving the first step of the employed strategy are used.

Proof of Theorem 2:

- By Lemma 5, there are two configurations C′ and C″, each (n-1)-reachable from C_0, such that:
    - all n-1 shared r/w registers are covered by processrs p0, ..., pn-2 in C′
    - C″ is an idle configuration
    - $C' \sim^{n-1} C''$.
- There exists an (n-1)-only execution fragment α from C′ in which p_{n-1} ends up in the critical section
- In α, p_{n-1} must write into some register which is not covered in C′
- However, all n-1 are covered in C′. This is a contradiction!

A Tight Upper Bound - The One-Bit Algorithm

Code of process p_i , $i \in \{1 ,..., n\}$

```
repeat {
    b[i] = true; j = 1;
        while (b[i] == true) and (j < i) {
            if (b[j] == true) {
                b[i] = false; await (b[j] ==false);
            }
            j = j+1
        }
}
until (b[i] == true);
for (j = i+1 to n)
    await (b[j] == false);
critical section
b[i] = false;
```

Properties of the One-Bit Algorithm

- Satisfies mutual exclusion and deadlock-freedom
- Starvation is possible
- It is not symmetric
- It uses only n shared bits and hence it is space optimal

Transparency made by Gadi Taubenfeld – Synchronization Algorithms and Concurrent Programming