Introduction to Synchronization

Klara Nahrstedt

Lecture 10

2/7/07

CS241 Administrative

This week

- SMP2, Deadline Monday 9am, 2/12/07
- Regular Quiz 3 on Friday, 2/9/07

Reminder:

Read Stallings Chapter 5 and R & R Chapter 13 and 14

Overview

Introduction to synchronization

- Why do we need synchronization?
- What is a data race?
- Critical region, mutual exclusion, progress, bounded waiting

Inter-Process Communication (IPC)

IPC needed in

- multiprogramming – management of multiple processes within a uni-processor system
- multiprocessing – management of multiple processes within a multi-processor system

- Pass information to each other via shared resource

Mutual exclusion & Synchronization

- Proper sequencing

Spooling Example: Correct

Process 1

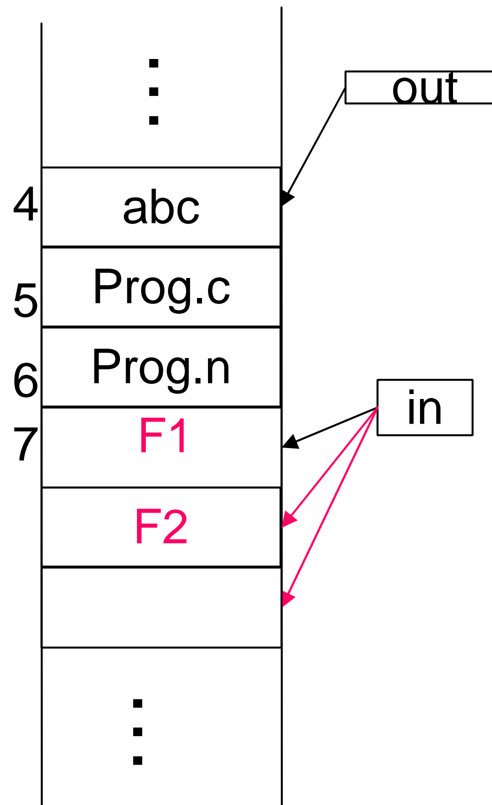
int next_free;

1 next_free = in;

2 Stores F1 into next_free;

3 in=next_free+1

Shared memory



Process 2

int next_free;

4 next_free = in

5 Stores F2 into next_free;

6 in=next_free+1

Spooling Example: Data Races

Process 1

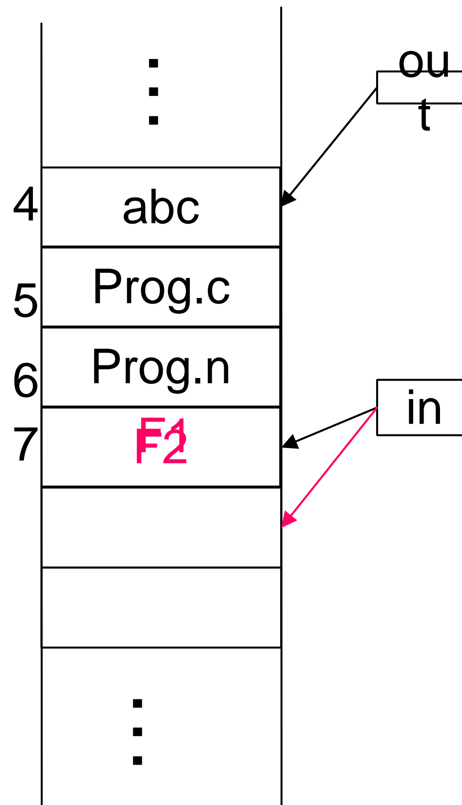
int next_free;

1 next_free = in;

3 Stores F1 into next_free;

4 in=next_free+1

Shared memory



Process 2

int next_free;

2 next_free = in
/* value: 7 */

5 Stores F2 into next_free;

6 in=next_free+1

Critical Region (Critical Section)

```
Process {
    while (true) {
        ENTER CRITICAL SECTION
         Access shared variables; // Critical Section;
        LEAVE CRITICAL SECTION
        Do other work
    }
}
```

Critical Region Requirements

- **Mutual Exclusion**
- **Progress**
- **Bounded Wait**
- **No Speed and Number of CPU assumptions**

Critical Region Requirements 1 of 2

Mutual Exclusion: No other process must execute within the critical section while a process is in it.

Progress: If no process is waiting in its critical section and several processes are trying to get into their critical section, then entry to the critical section cannot be postponed indefinitely.

Critical Region Requirements

Mutual Exclusion

Progress

Bounded Wait: A process requesting entry to a critical section should only have to wait for a bounded number of other processes to enter and leave the critical section.

Speed and Number of CPUs: No assumption may be made about speeds or number of CPUs.

Bounded Wait **Mutual Exclusion** **Progress**

Good concrete example...

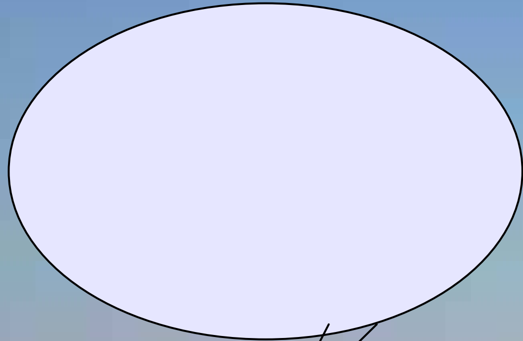
Oversimplifying Assumptions

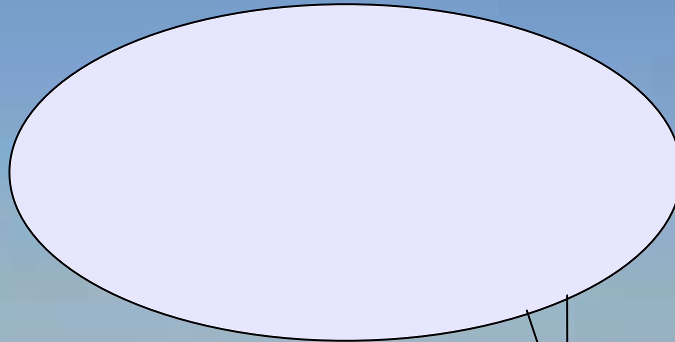
*visual learning aid: '**bmp**' in alphabetical order*

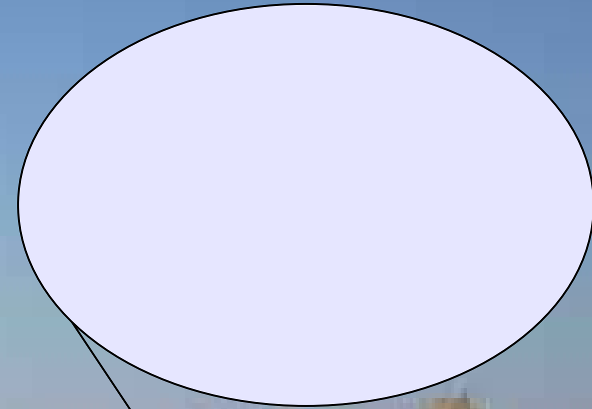
Bounded Wait **Mutual Exclusion** **Progress**

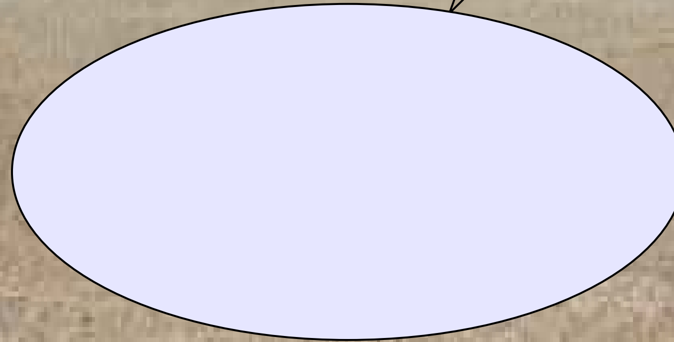
Oversimplifying Assumptions

Bounded Wait **Mutual Exclusion** **Progress**









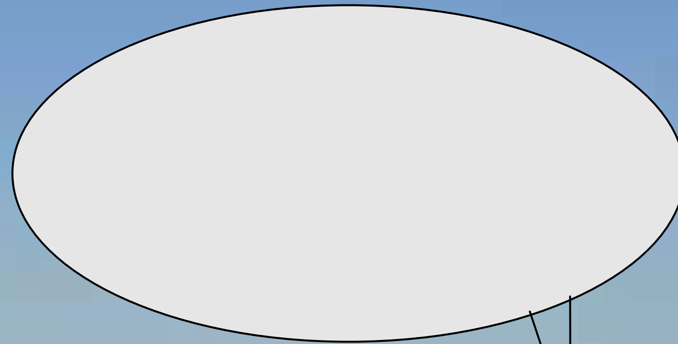
Oversimplifying Assumptions

Bounded Wait

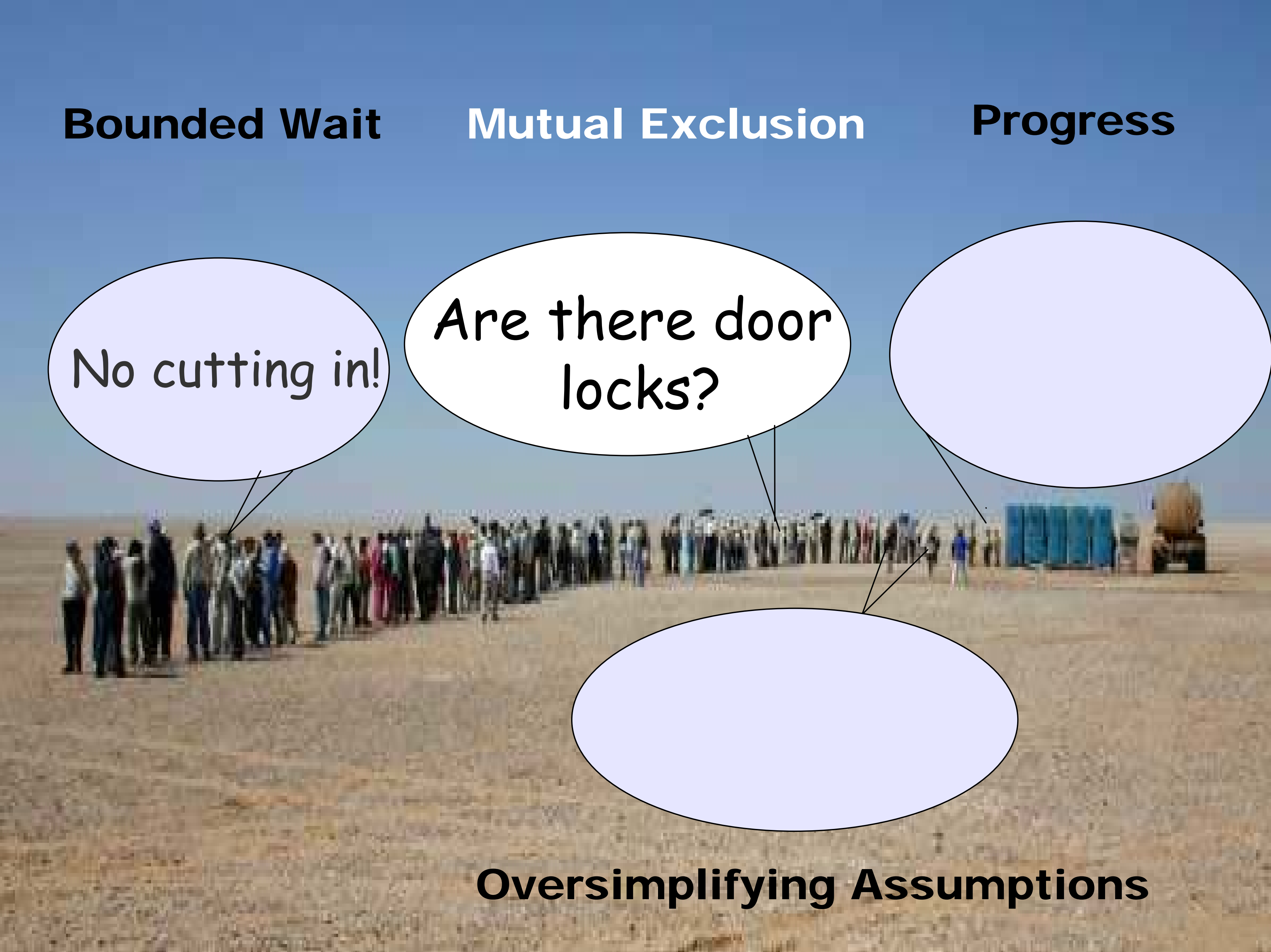
Mutual Exclusion

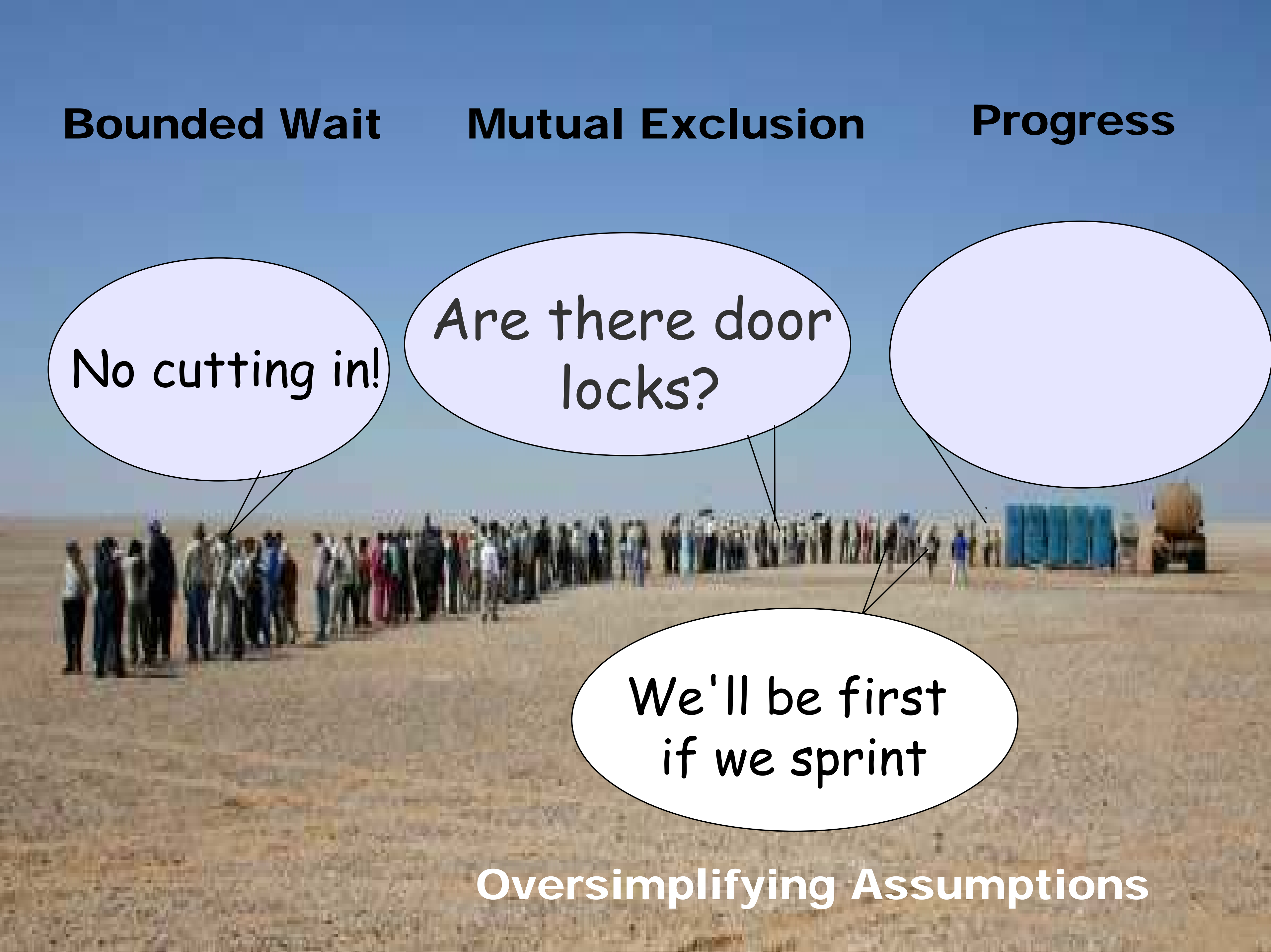
Progress

No cutting in!



Oversimplifying Assumptions


Bounded Wait
Mutual Exclusion
Progress
No cutting in!
Are there door locks?
Oversimplifying Assumptions


Bounded Wait
Mutual Exclusion
Progress
No cutting in!
Are there door locks?
We'll be first if we sprint
Oversimplifying Assumptions

Bounded Wait
Mutual Exclusion
Progress
No cutting in!
Are there door locks?
Well, Did you see anybody go in?
We'll be first if we sprint
Oversimplifying Assumptions

Progress?

Bounded Wait?

What's the difference?

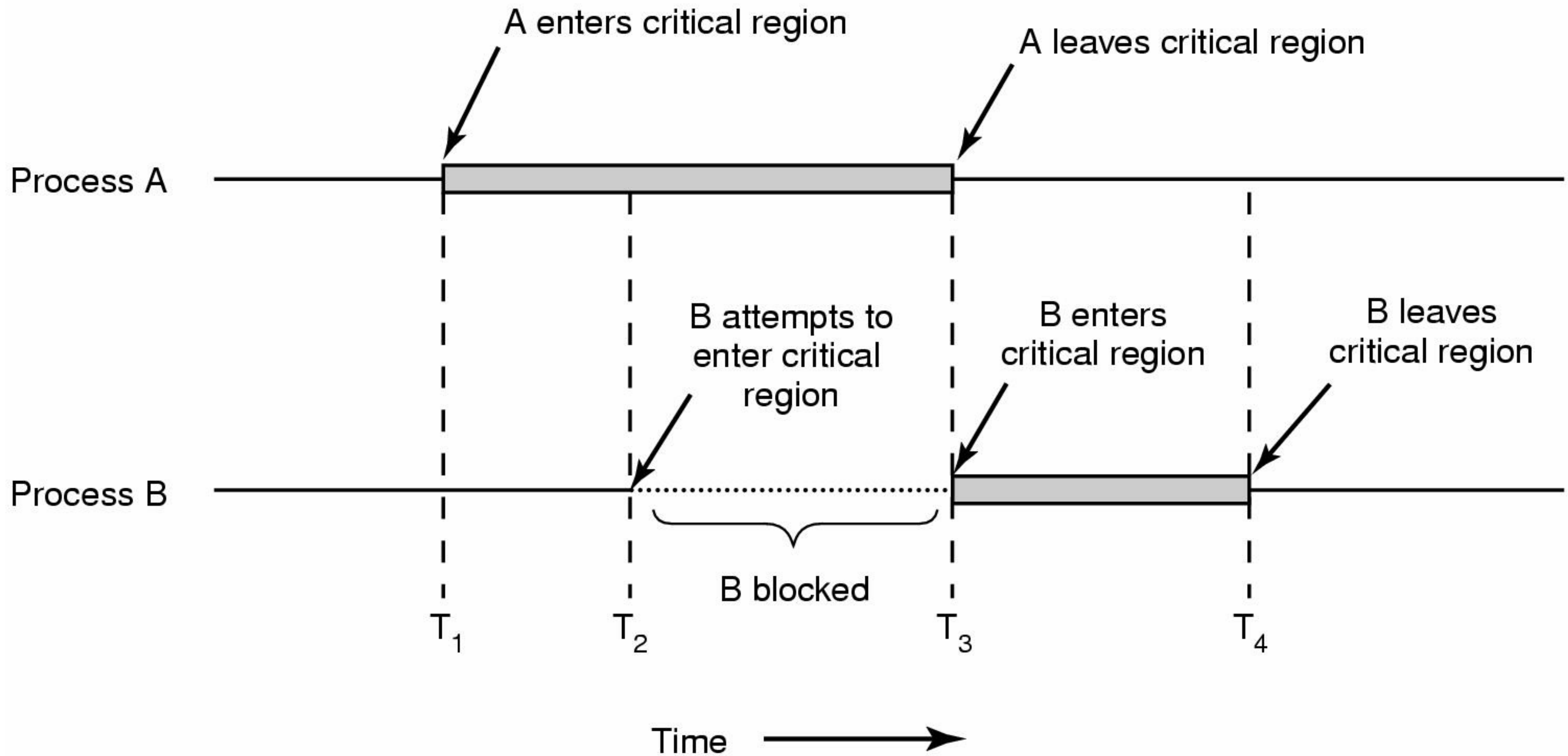
Progress?

- If ***no process is waiting in its critical section*** and several processes are trying to get into their critical section, then entry to the critical section **cannot be** postponed indefinitely

Bounded Wait?

A process requesting entry to a critical section should only have to wait for a *bounded number* of other processes to enter and leave the critical section.

Critical Regions (2)



Mutual exclusion using critical regions

Mutual Exclusion With Busy Waiting

- Possible Solutions
  - Software-only candidate solutions (Two-Process Solutions)
    - Lock Variables
    - Turn Mutual Exclusion
    - Other Flag Mutual Exclusion
    - Two Flag Mutual Exclusion
    - Two Flag and Turn Mutual Exclusion
  - Hardware solutions
    - Disabling Interrupts; Test-and-set; Swap (Exchange)
  - Semaphores

Lock Variables

```
while (lock) {/* spin spin spin spin */}
lock = 1;
/* EnterCriticalSection; */
  access shared variable;
/* LeaveCriticalSection; */
lock = 0;
```

What's the problem?

Turn-based Mutual Exclusion with strict alternation

```
ProcessMe/* For two processes */
{
  while (true)
  {  while ( turn != my_process_id) { }
     access shared variables
     turn = other_process_id
     do other work
  } }
```

Turn-based Mutual Exclusion with strict alternation

```
ProcessMe/* For two processes */
{
  while (true)

  {  while ( turn != my_process_id) { }
     access shared variables
     turn = other_process_id
     do other work
  } }
```

Satisfies Mutual Exclusion?

Turn-based Mutual Exclusion with strict alternation

```
ProcessMe/* For two processes */
{
  while (true)
  {  while ( turn != my_process_id) { }
     access shared variables
     turn = other_process_id
     do other work
  } }
```

Satisfies Bounded Waiting?

Turn-based Mutual Exclusion with strict alternation

```
ProcessMe /* For two processes */
{
  while (true)
  {  while ( turn != my_process_id) { }
     access shared variables
     turn = other_process_id
     do other work
  } }
```

Satisfies Progress?

Other Flag Mutual Exclusion

```
int owner[2] = {false, false}
ProcessMe
{  while (true)
   {   while (owner[other_process_id] ) { }
       owner[my_process_id] = true
       access shared variables
       owner[my_process_id] = false
       do other work
   }  }
```

Progress?

Other Flag Mutual Exclusion

```
int owner[2] = {false, false}
ProcessMe
{  while (true)
   {   while (owner[other_process_id] ) { }
       owner[my_process_id] = true
      access shared variables
      owner[my_process_id] = false
      do other work
   }  }
```

Mutual exclusion?

2 Flag Mutual Exclusion

```
int owner[2]= {false, false}
ProcessMe {
  while (true) {
    owner[my_process_id] = true
    while (owner[other_process_id] ) { }
    access shared variables
    owner[my_process_id] = false
    do other work
  } }
```

Mutual Exclusion?

2 Flag Mutual Exclusion

```
int owner[2]= {false, false}
ProcessMe {
  while (true) {
    owner[my_process_id] = true
    while (owner[other_process_id] ) { }
    access shared variables
    owner[my_process_id] = false
    do other work
  } }
```

Progress?

2 Flag Mutual Exclusion

```
int owner[2]= {false, false}
ProcessMe {
  while (true) {
    owner[my_process_id] = true
    while (owner[other_process_id] ) { }
    access shared variables
    owner[my_process_id] = false
    do other work
  } }
```

Bounded Waiting?

2 Flag and Turn Mutual Exclusion

Problems?

```
   int owner[2]={false, false}
   int turn;
ProcessMe {   while (true)
   { owner[my_process_id] = true;
     turn = other_process_id;
     while (owner[other_process_id]
            and turn == other_process_id ) { }
     access shared variables
     owner[my_process_id] = false
     do other work  } }
```

2 Flag and Turn Mutual Exclusion

Peterson Solution

```
int owner[2]={false, false}
int turn;
ProcessMe {   while (true)
  { owner[my_process_id] = true;
    turn = other_process_id;
    while (owner[other_process_id]
          and turn == other_process_id ) { }
    access shared variables
    owner[my_process_id] = false
    do other work  } }
```

Discussion

- In uni-processor
  - Concurrent processes cannot be overlapped, only interleaved
  - Process runs until it invokes system call, or is interrupted
  - To guarantee mutual exclusion, hardware support could help by allowing disabling interrupts

```
While(true) {
    /* disable interrupts */
    /* critical section */
    /* enable interrupts */
    /* remainder */
}
```

What's the problem with this solution?

Discussion

- In multi-processors
  - Several processors share memory
  - Processors behave independently in a peer relationship
  - Interrupt disabling will not work
  - We need hardware support where access to a memory location excludes any other access to that same location
  - The hardware support is based on execution of multiple instructions atomically
  - Atomic execution of a set of instructions means that these instructions are treated as a single step that cannot be interrupted

Summary

- Synchronizations are important for correct multi-threading programs
- Data races
- Critical regions
- Solutions to protect critical regions
  - Software-only approaches
- Next lecture:
  - Multi-processor hardware solutions;
  - semaphores