Synchronization

Introduction to Synchronization

Stefano Quer
Dipartimento di Automatica e Informatica
Politecnico di Torino

License Information

This work is licensed under the license

CC BY-NC-ND 4.0

Attribution-NonCommercial-NoDerivatives 4.0 International

This license requires that reusers give credit to the creator. It allows reusers to copy and distribute the material in any medium or format in unadapted form and for noncommercial purposes only.

BY: Credit must be given to you, the creator.

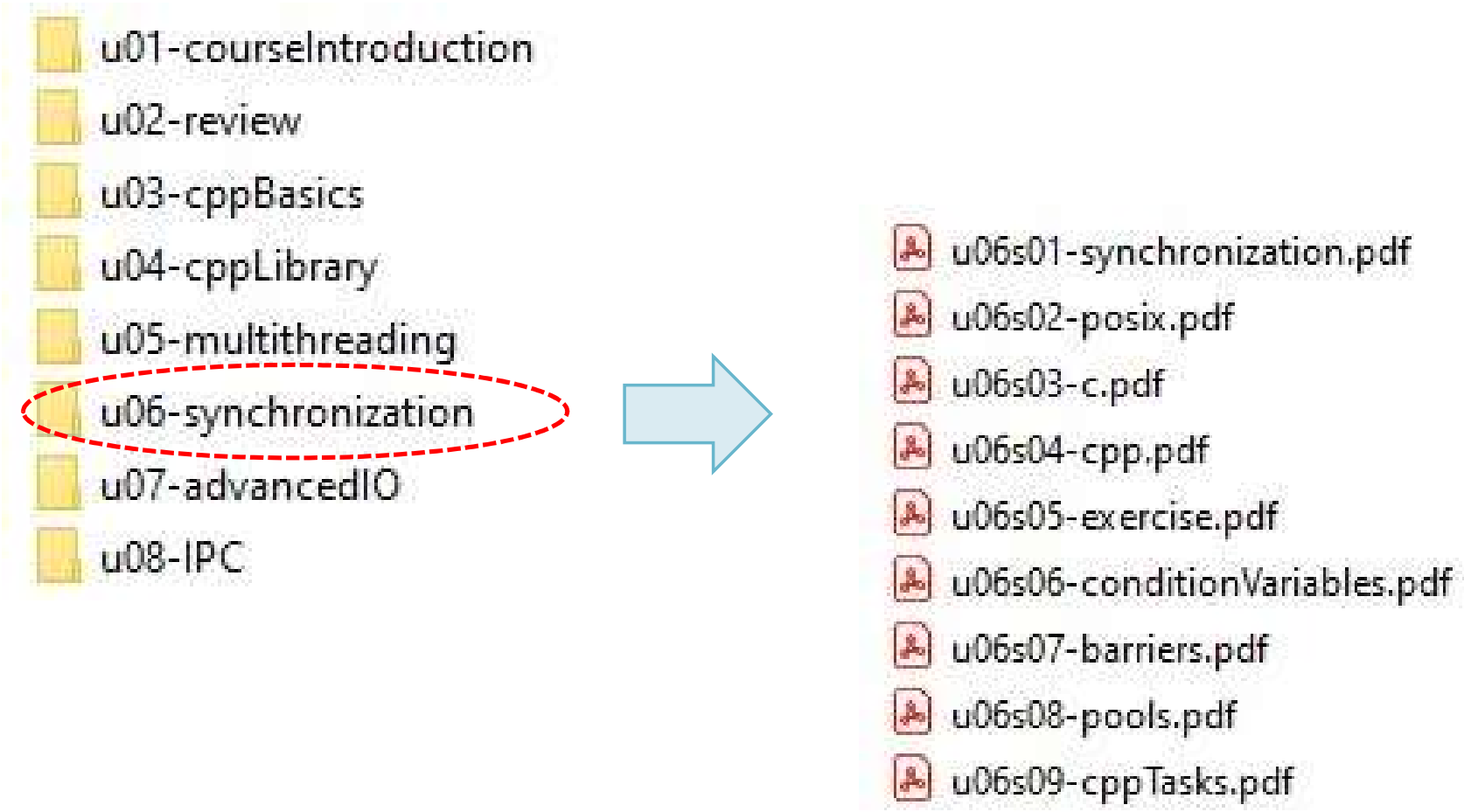
NC: Only noncommercial use of your work is permitted.
Noncommercial means not primarily intended for or directed towards commercial advantage or monetary compensation.

ND: No derivatives or adaptations of your work are permitted.

To view a copy of the license, visit:
https://creativecommons.org/licenses/by-nc-nd/4.0/?ref=chooser-v1

Introduction

- Where are we?



Introduction

- Critical Section (**CS**) or Critical Region (**CR**)
  - A section of code, common to multiple threads, in which each thread can read and **write** shared objects
- Access to CS is subject to **race conditions**
  - The result depends on the execution order of the processes instructions

These are sections of code or variables that are accessed by multiple threads. Proper synchronization ensures that only one thread accesses these sections at a time to prevent inconsistencies.

Critical Sections (CS): These are the parts of your program that access shared resources and must be executed by only one thread at a time.

Race Conditions: This happens when the outcome of a program depends on the sequence or timing of uncontrollable events such as the scheduling of threads.

Example

FIFO, Queue, Circular Buffer

Code Explanation:
The enqueue function adds an element to the queue.
The dequeue function removes an element from the queue.

T_i

```
void enqueue (int val) {
  if (n>SIZE) return;
  queue[tail] = val;
  tail=(tail+1)%SIZE;
  n++;
  return;
}
```

register = n
register = register + 1
n = register

T_j

```
int dequeue (int *val) {
  if (n<=0) return;
  *val=queue[head];
  head=(head+1)%SIZE;
  n--;
  return;
}
```

register = n
register = register - 1
n = register

- Even if **enqueue** and **dequeue** operate on the different ends of the queue, the variable **n** is shared

Shared Variable (n):
Even though enqueue and dequeue operate on different ends of the queue, they both modify the shared variable n, which tracks the number of elements in the queue.
A race condition can occur if both functions try to modify n simultaneously, leading to incorrect updates.

Race condition:
Increments and decrements can be lost

If enqueue and dequeue are called simultaneously:
enqueue reads n, increments it, and stores it back.
dequeue reads n, decrements it, and stores it back.
Without proper synchronization, both may read the same value of n and update it incorrectly.

Critical sections

Preventing Race Conditions: Use access protocols to enforce mutual exclusion (ensuring only one thread accesses the critical section at a time).

Release Section: Code that runs after leaving the critical section to signal that the critical section is now free for other threads to use.

- Race conditions could be prevented with an access protocol that enforces mutual exclusion for each CS
  - Before a CS, there should be a **reservation** section
    - The reservation code must block (lock out) the P (or T) if another P (or T) is using its CS
  - After the CS, there should be a **release** section
    - The release possibly unlocks another P (or T) which was waiting in the ”reservation” code of its CS

Reservation Section:

Code that runs before entering the critical section to check if the critical section is available. If another thread is using it, the current thread waits.

Mutex (Mutual Exclusion Object):
A common tool used to implement critical sections. It locks the critical section so that only one thread can access it at a time.

Access protocol

T_i

```
while (TRUE) {
   ...
   reservation code
   Critical Section
   release code
   ...
   non critical section
}
```

Structure of Critical Section Protection:
Reservation Code: Check and wait if necessary until the critical section is free.
Critical Section Code: The actual code that accesses shared resources.
Release Code: Signal that the critical section is now free.

T_j

```
while (TRUE) {
   ...
   reservation code
   Critical Section
   release code
   ...
   non critical section
}
```

Non-critical Sections:These are parts of your code that do not access shared resources and do not require synchronization

- Every Critical Section is protected by an
  - Enter code (reservation, or prologue)
  - Exit code (release, or epilogue)
- Non-critical sections should not be protected
- OSs provide appropriate primitives

Locks and Unlocks:
In the reservation code, threads try to acquire a lock (using mutex.lock()). If the lock is already taken, they wait.

In the release code, threads release the lock (using mutex.unlock()), allowing other waiting threads to proceed.