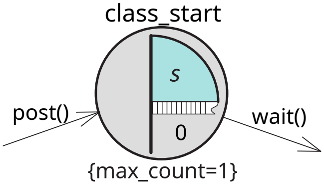
Data exchange in real-time applications using semaphores

Demonstration of basic principles using a practical example

Gökçe Aydos



Learning goals

understand semaphore’s working principle

Case study: Odometry in a robot

Figure 1: A wheeled robot

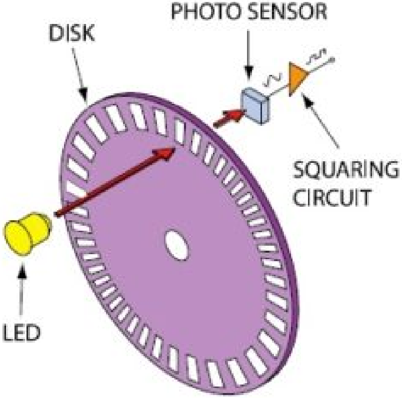
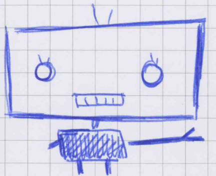


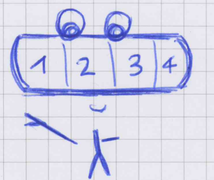
Figure 2: Rotary encoder working principle

```
int encoder_value;
...
void encoder_read() { ...
    encoder_value = ...;
}
void odometry_process() { ...
    odometry += encoder_value;
}
void journal() { ...
    fprintf(..., uptime, odometry);
}
INTERRUPT_FROM(ENCODER) encoder_read();
INTERRUPT_FROM(TIMER_10HZ) { odometry_process; journal; }
```

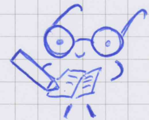
im Robo-Office



Encoder-Emily



Odometry-Otto



Journal-Josef

Demonstration: Emily and Otto exchange `encoder_val`

Do you see any problems?

Demonstration: Emily and Otto exchange `encoder_val`

Do you see any problems?

- data integrity
- data duplication
- data loss

Semaphore

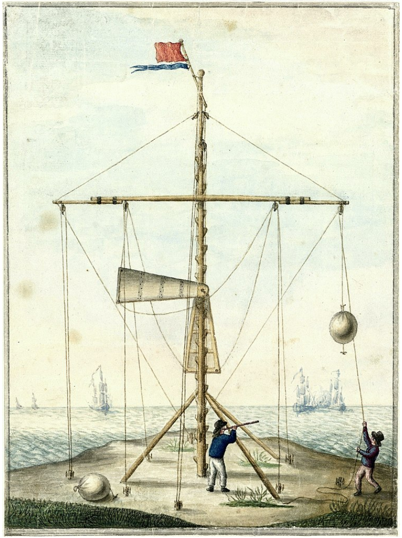


Figure 4: Coastal telegraph, also known as *semaphore*

How does a semaphore work?

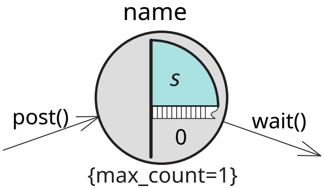


Figure 5: Semaphore with a maximum count of 1 and initial value of 0

Semaphore application patterns

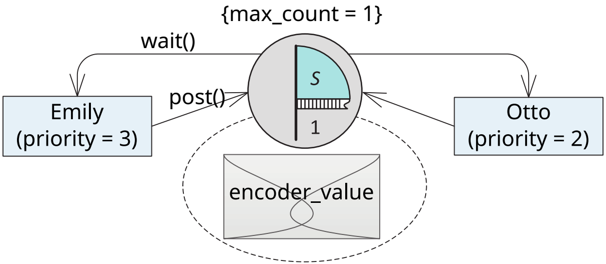


Figure 6: Pattern 1: Resource protection (single)

How would you solve the problem/s we had in the beginning with a semaphore?

Demonstration: Protecting `encoder_value`

How can Emily and Otto meet?

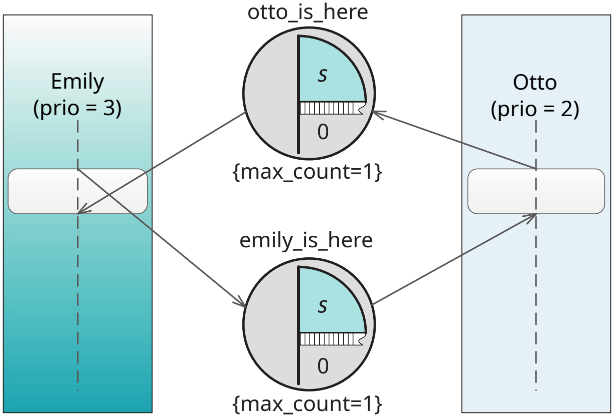


Figure 7: Pattern 2: Rendezvous synchronization

```
Emily:
post(emily_is_here);
wait(otto_is_here);
```

```
Otto():
post(otto_is_here);
wait(emily_is_here);
```

How can Emily and Otto work one after another?

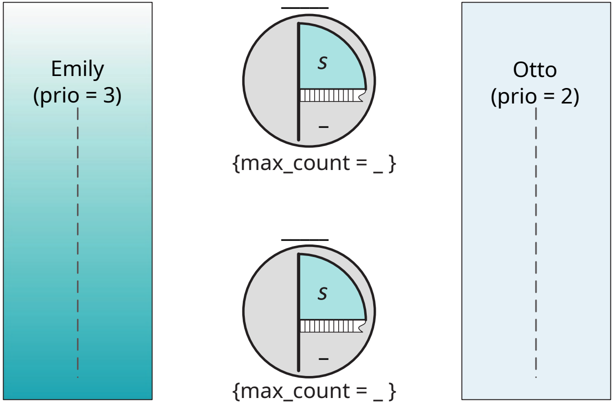
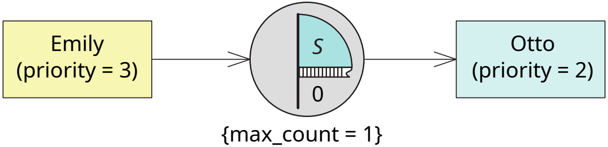


Figure 8: Fill the gaps!

Demonstration: Ensuring that Otto works after Emily

Quiz

Would the following solution work in the last problem?



A) Yes

B) No

C) I don’t know

Problem

Imagine we modified `encoder_value` to a FIFO with a capacity of 3. How can we leverage a semaphore that there are no more than 3 `encoder_value`s in the FIFO?

Problem 2

Instead of using a semaphore, we could use a loop like:

```
int encoder_read_done;

void* odometry_process() {
    ...
    while (!encoder_read_done);
    ...
}
```

What are the pros/cons?

Summary

Where can I find semaphores?

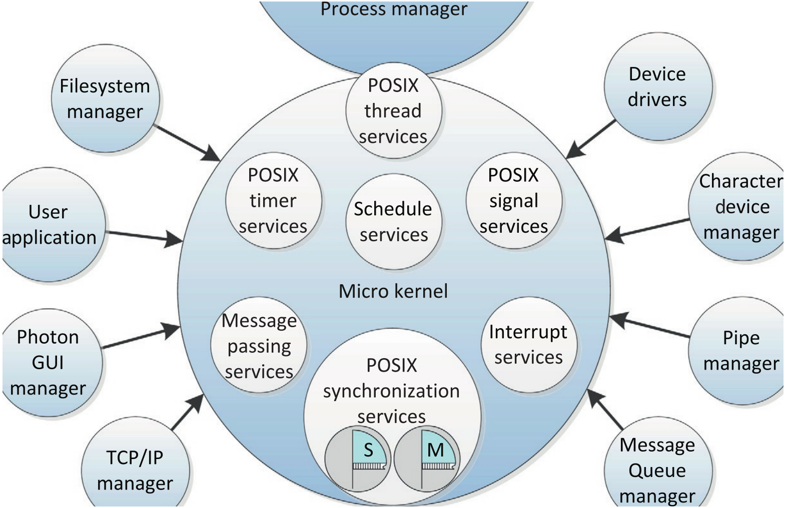


Figure 9: Embedded Real-time microkernel Blackberry QNX Neutrino

Further resources I

- Demonstrator code
- Fan, Real-time Embedded Systems, 2015

  Very suitable for introduction, *includes many code examples*. Many of the resources in this work is based on this book.
- POSIX.1-2017

  The standard document. Most of the `man` pages are based on this doc. Includes the rationale behind some concepts. Especially relevant: Realtime services index

More resources I skimmed, but did not use thoroughly:

- Arpaci-Dusseau et.al., Operating Systems: Three Easy Pieces, 2018

  Enjoyed reading. Contains a chapters about concurrency.

Further resources II

- Tian et.al., (ed.), Handbook of real-time computing, 2022

  Based on the latest research, written by many experts. Targeted at researchers.

- Kopetz et.al, Real-Time Systems - Design Principles for Distributed Embedded Applications, 2022

  Based on the lecture notes in Vienna University

- Hüning, Echtzeitbetriebssystem, Embedded Systems für IoT, 2018

  Principles of real-time OS, a case study based on Renesas Synergy RTOS

- Seck, Aufbau Echtzeitbetriebssystem OS9000, 2014

  German case study of OS-9 RTOS, part of lecture series about real-time systems.

Further resources III

- Introduction to real-time systems

  ROS is popular framework for robotics. Covers ROS programming related aspects. Real-time tutorial: ROS2 demo: Understanding real-time programming

Appendix

Semaphore pattern

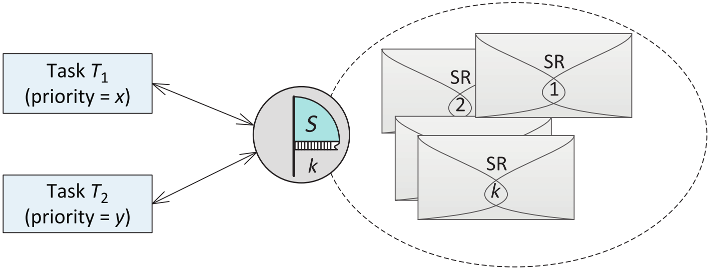
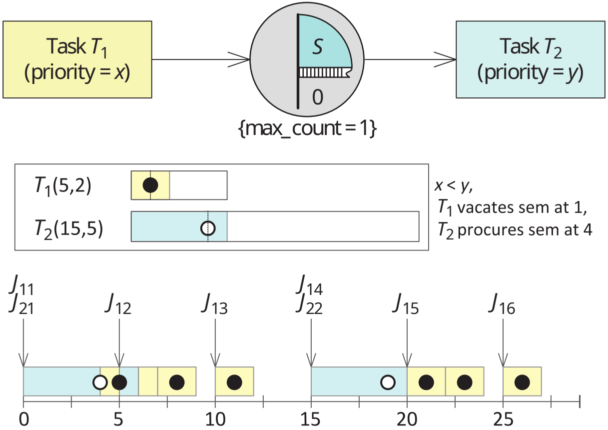


Figure 10: Semaphore pattern: Resource protection (multiple)

if a consumer task T2 must wait for the producer T1:



$x < y$,
T_1 vacates sem at 1,
T_2 procures sem at 4

Figure 11: Semaphore pattern: Task synchronization

if a consumer task T2 must wait for the producer T1:

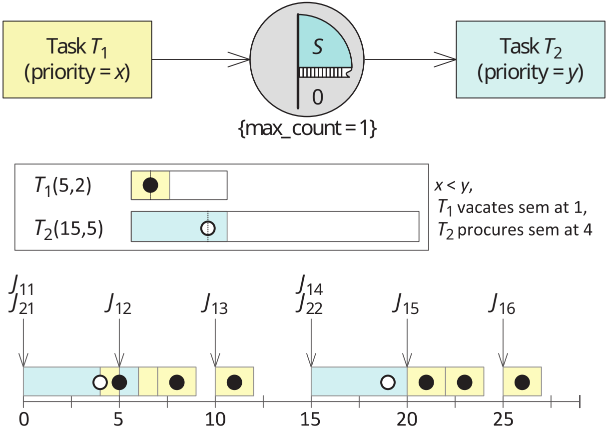


Figure 11: Semaphore pattern: Task synchronization

Would a single semaphore with `max_count = 2` not suffice?

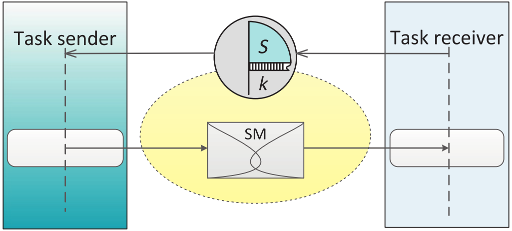


Figure 12: Semaphore pattern: Flow control

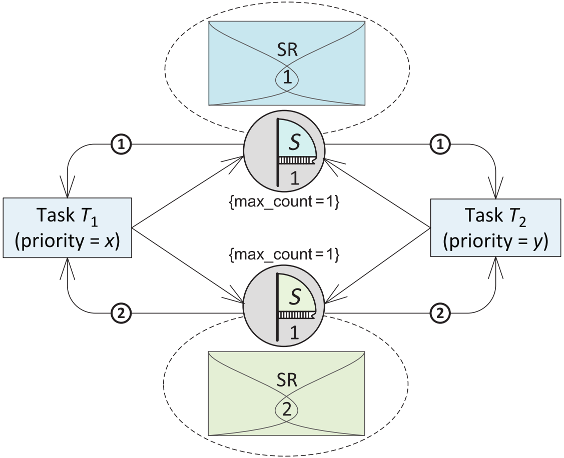


Figure 13: Semaphore pattern: Deadlock (circular wait) avoidance

```
T1() {
    ...
    procure(sem🔵);
```

```
T2() {
    ...
    // same as T1()
```

Do you see a problem below?

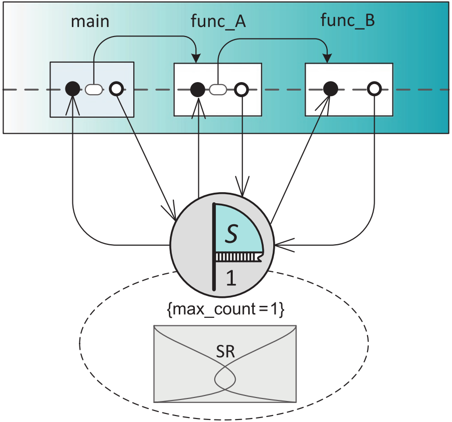


Figure 14: Example showing the disadvantage of the semaphore in case of recursive requests

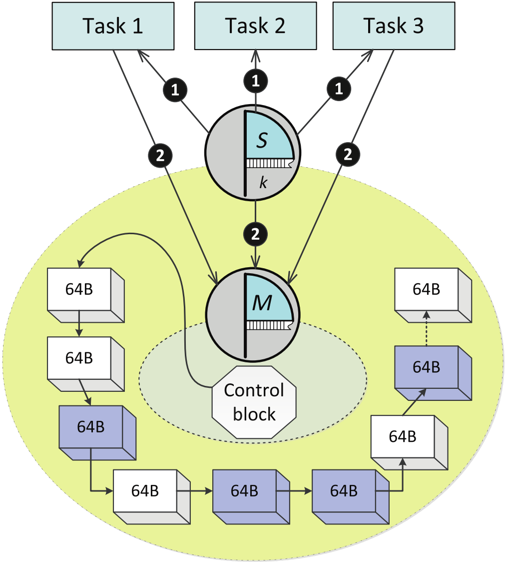


Figure 15: Semaphore + mutex pattern: memory management and exclusive access to control block

Now you are armed with mutex. How would you solve the problem we had in the beginning using a mutex?

Now you are armed with mutex. How would you solve the problem we had in the beginning using a mutex?

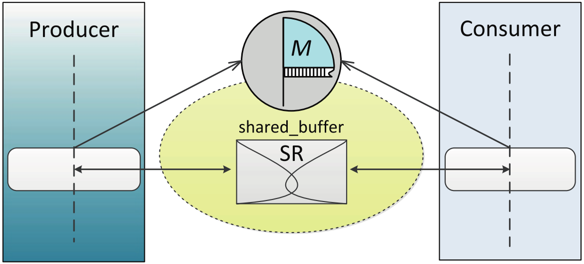


Figure 16: Mutex enables exclusive access

Condition variable

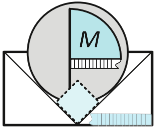


Figure 17: Condition variable: Guarding mutex for exclusive access + a condition

Condition variable application pattern

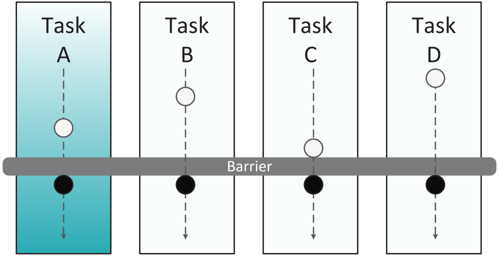


Figure 18: Barrier synchronization

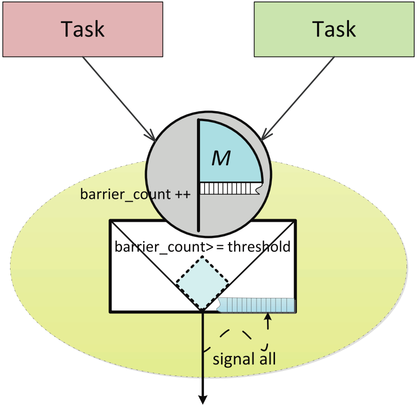


Figure 19: Barrier synchronization

Previous problem revisited

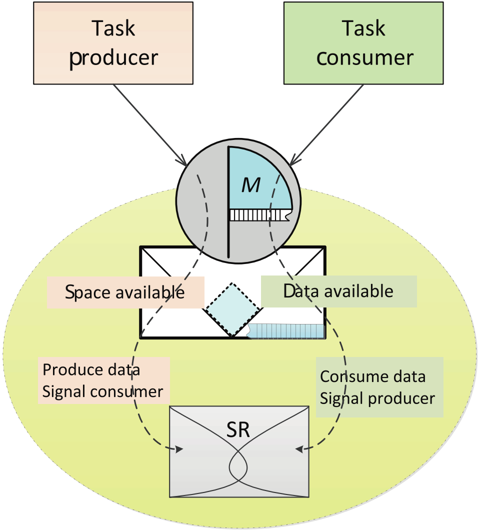
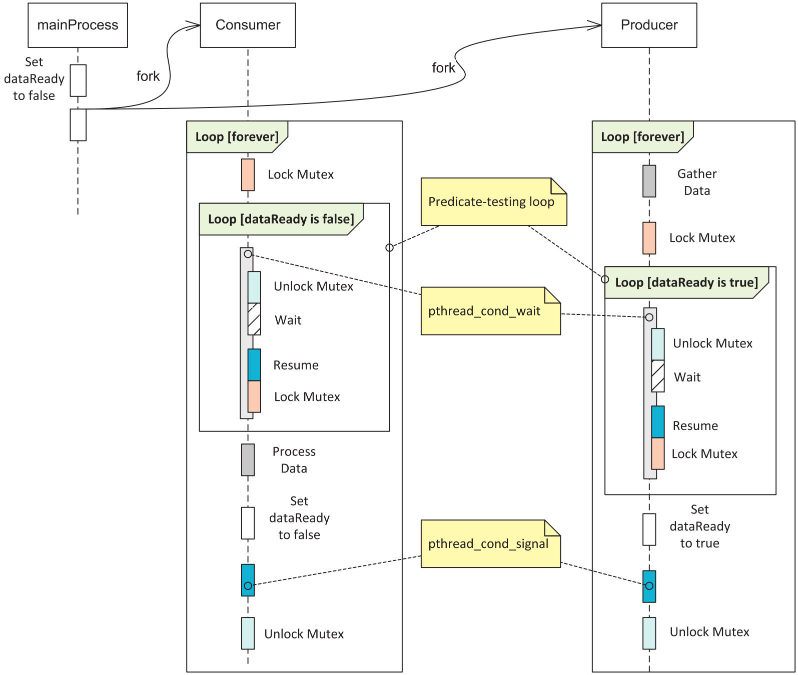


Figure 20: Producer consumer problem solved with condition variables


mainProcess
Consumer
Producer
Set
dataReady
to false
fork
fork
Loop [forever]
Lock Mutex
Loop [dataReady is false]
Unlock Mutex
Wait
Resume
Lock Mutex
Process
Data
Set
dataReady
to false
Unlock Mutex
Predicate-testing loop
pthread_cond_wait
pthread_cond_signal
Loop [forever]
Gather
Data
Lock Mutex
Loop [dataReady is true]
Unlock Mutex
Wait
Resume
Lock Mutex
Set
dataReady
to true
Unlock Mutex