Operating Systems and Concurrency

Lecture 7: Concurrency

G52OSC/COMP2007

Geert De Maere
(Isaac Triguero)
{Geert.DeMaere,Isaac.Triguero}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2018

Concurrency

Context

- Threads and processes **execute concurrently** or **in parallel** and can **share resources** (e.g., devices, memory – variables and data structures)
  - Multi-programming/multi-processing **improves system utilisation**
- A process/thread can be **interrupted at any point in time** (timer, I/O)
  - The process "state" (including registers) is **saved** in the **process control block**
- The outcome of programs may become **unpredictable**:
  - Sharing data can lead to **inconsistencies** (e.g., when interrupted whilst manipulating data)
  - The **outcome of execution** may **depend on the order** in which instructions are carried out

Example

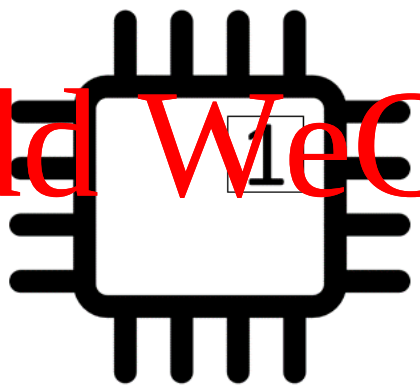
Incrementing a counter

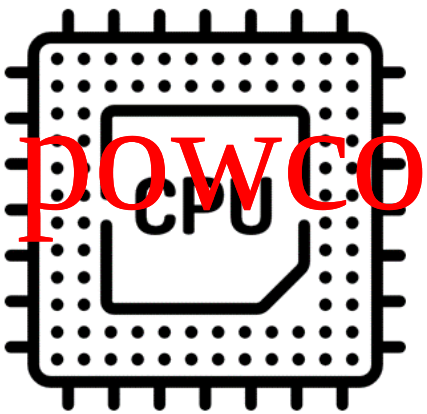
```
#include <stdio.h>
#include <pthread.h>
int counter = 0;
void * calc(void * number_of_increments) {
  int i;
  for(i = 0; i < *((int*) number_of_increments);i++)
    counter++;
}

int main() {
  int iterations = 5000000;
  pthread_t tid1,tid2;
  pthread_create(&tid1, NULL, calc, (void *) &iterations);
  pthread_create(&tid2, NULL, calc, (void *) &iterations);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example[1]

Incrementing a counter

- `counter++` consists of three separate actions:
  1. `read` the value of `counter` from memory and **store it in a register**
  2. `add` one to the value in the register
  3. `store` the value of the register **in counter** in memory
- The above actions are **NOT** "atomic", e.g. they can be interrupted by the timer ($\Rightarrow$ **context switch**)



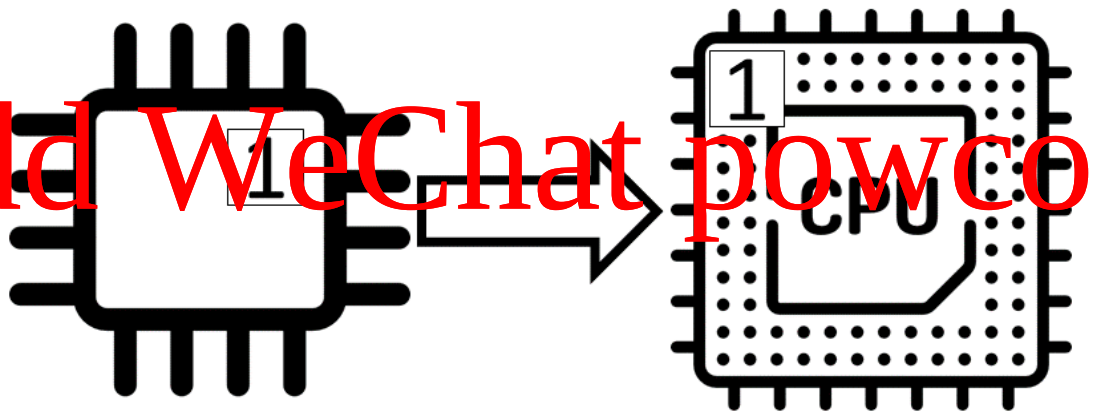


[1] Icons from https://www.flaticon.com/

Example[1]

Incrementing a counter

- `counter++` consists of three separate actions:
  1. `read` the value of counter from memory and **store it in a register**
  2. `add` one to the value in the register
  3. `store` the value of the register **in counter** in memory
- The above actions are **NOT** "atomic", e.g. they can be interrupted by the timer ($\Rightarrow$ **context switch**)

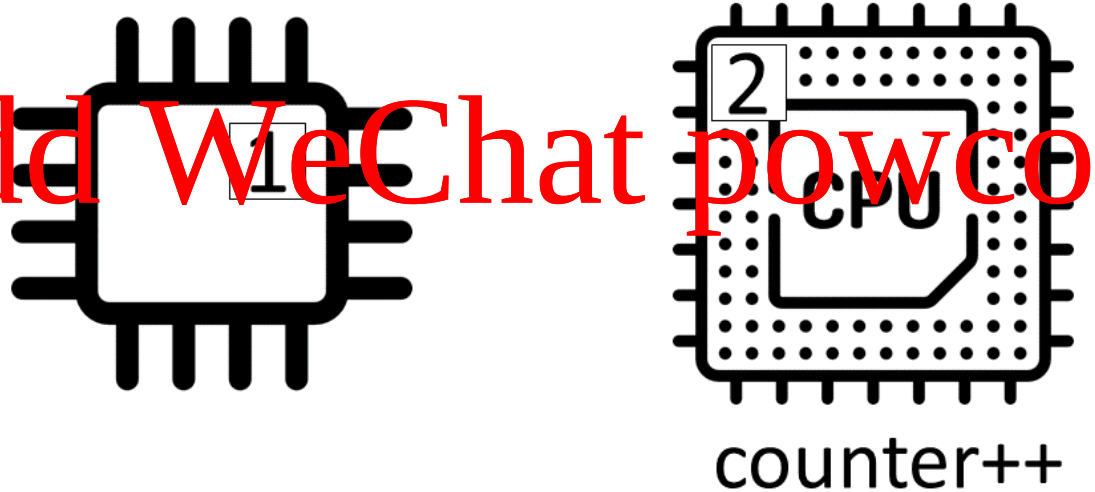


[1] Icons from https://www.flaticon.com/

Example[1]

Incrementing a counter

- `counter++` consists of three separate actions:
  1. `read` the value of counter from memory and **store it in a register**
  2. `add` one to the value in the register
  3. `store` the value of the register **in counter** in memory
- The above actions are **NOT** "atomic", e.g. they can be interrupted by the timer ($\Rightarrow$ **context switch**)

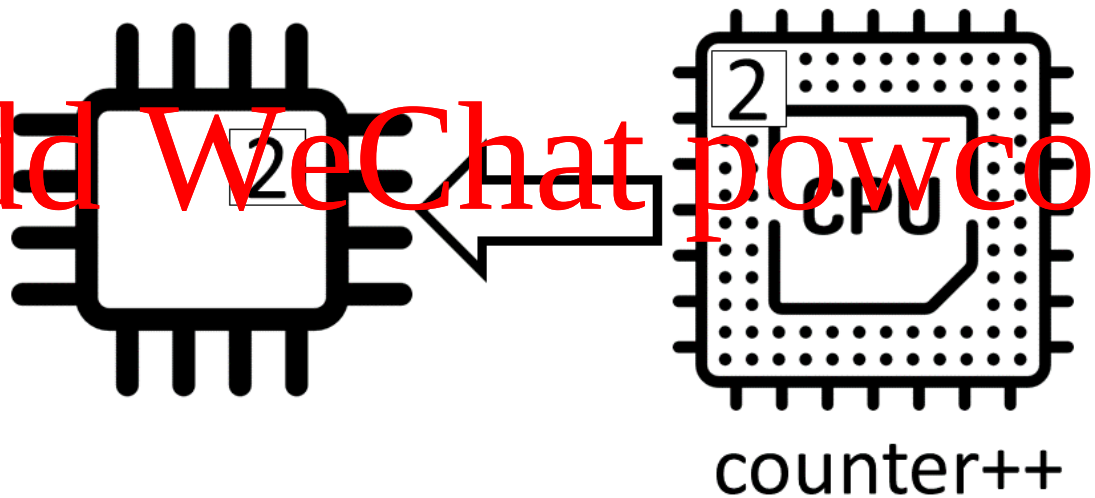


[1] Icons from https://www.flaticon.com/

Example[1]

Incrementing a counter

- `counter++` consists of three separate actions:
  1. `read` the value of counter from memory and **store it in a register**
  2. `add` one to the value in the register
  3. `store` the value of the register **in counter** in memory
- The above actions are **NOT** "atomic", e.g. they can be interrupted by the timer ($\Rightarrow$ **context switch**)



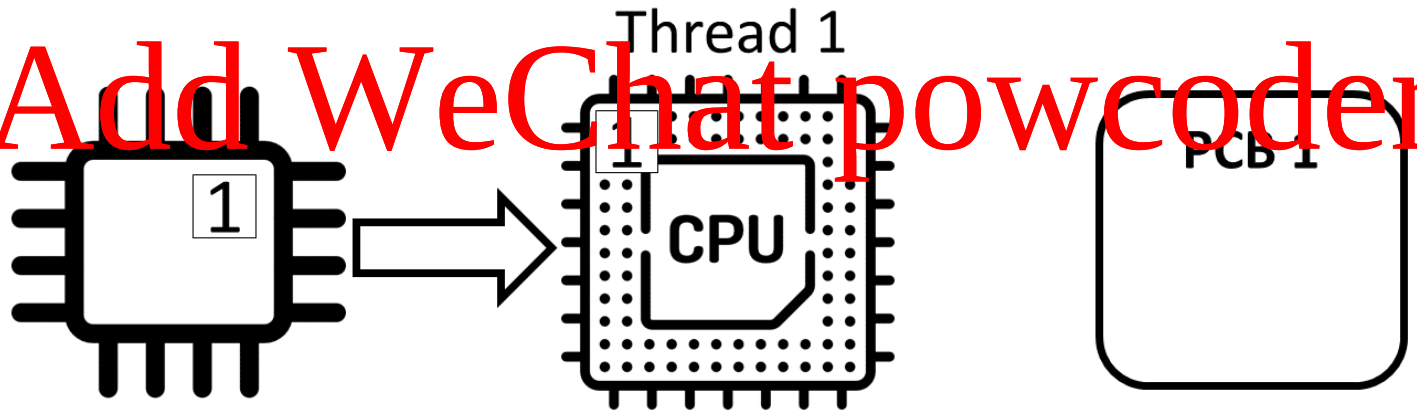
[1] Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

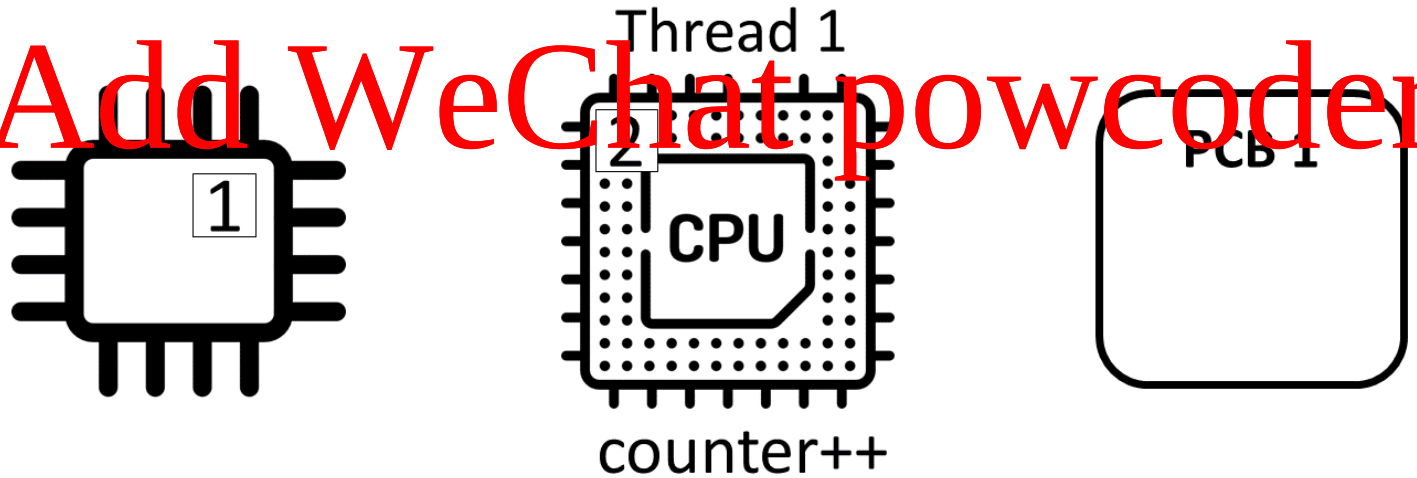
[2]Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

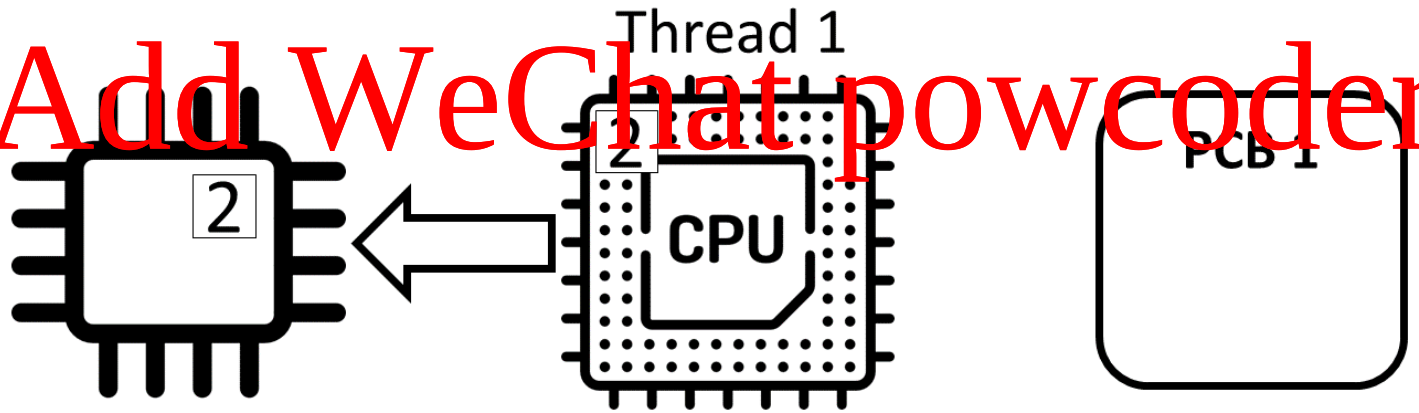
[2] Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

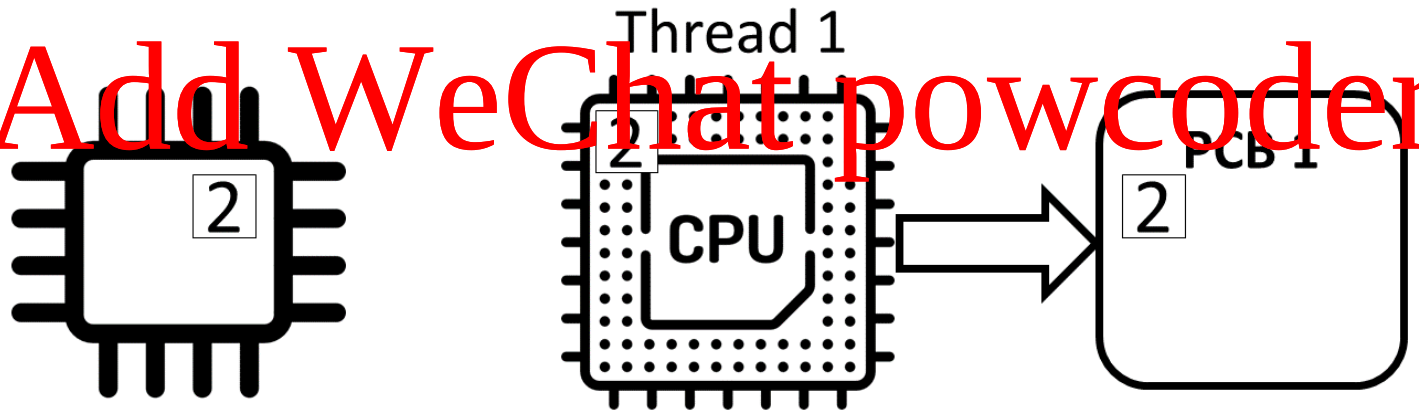
[2]Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

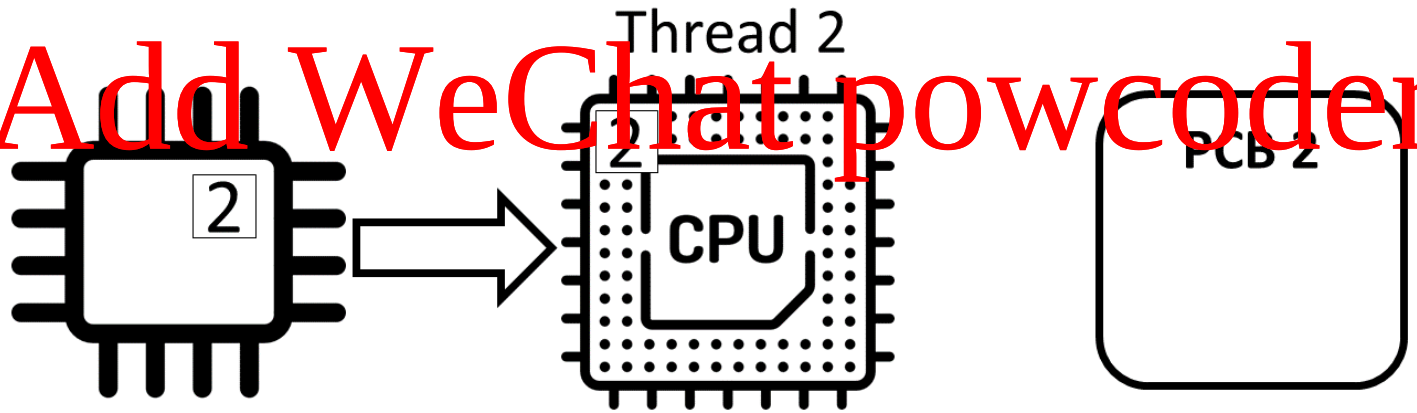
[2]Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

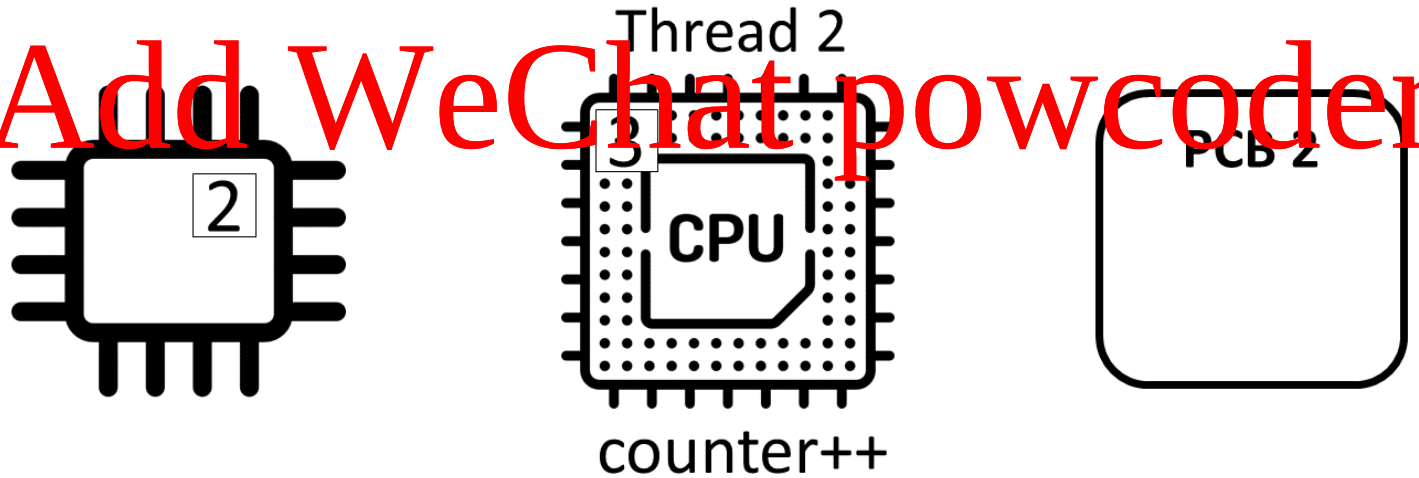
[2]Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

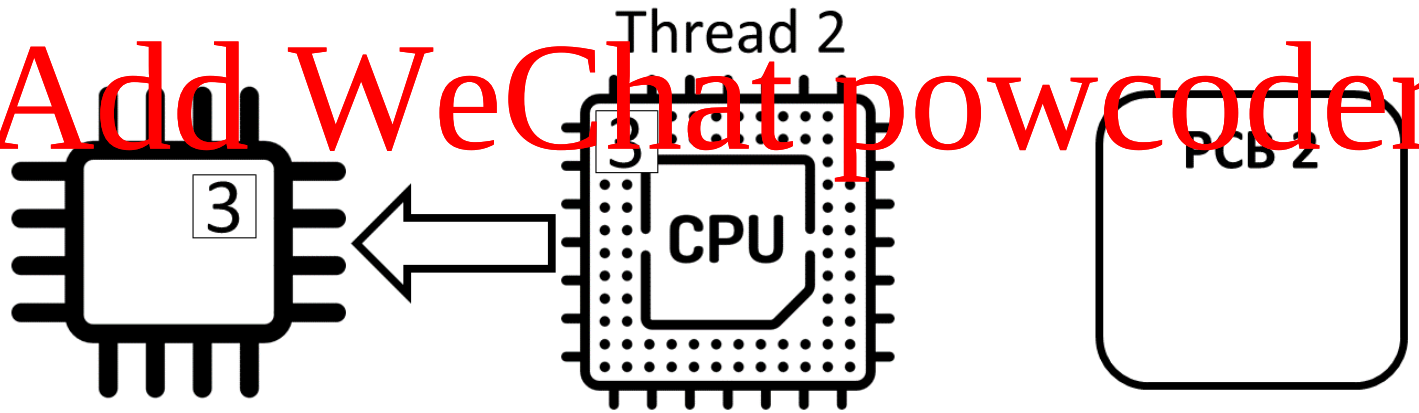
[2]Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

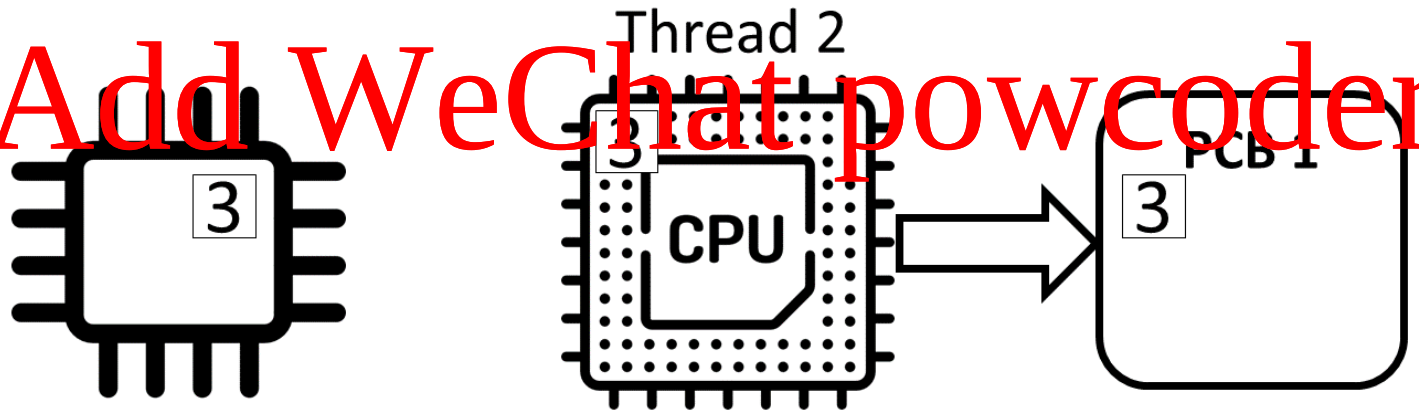
[2] Icons from https://www.flaticon.com/

Example[2]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
Add 1 to register value (= 2)
Store register in counter (= 2)
...
...
...
```

```
Thread 2:
...
...
...
...
Read counter -> register (= 2)
Add 1 to register value (= 3)
Store register in counter (= 3)
```

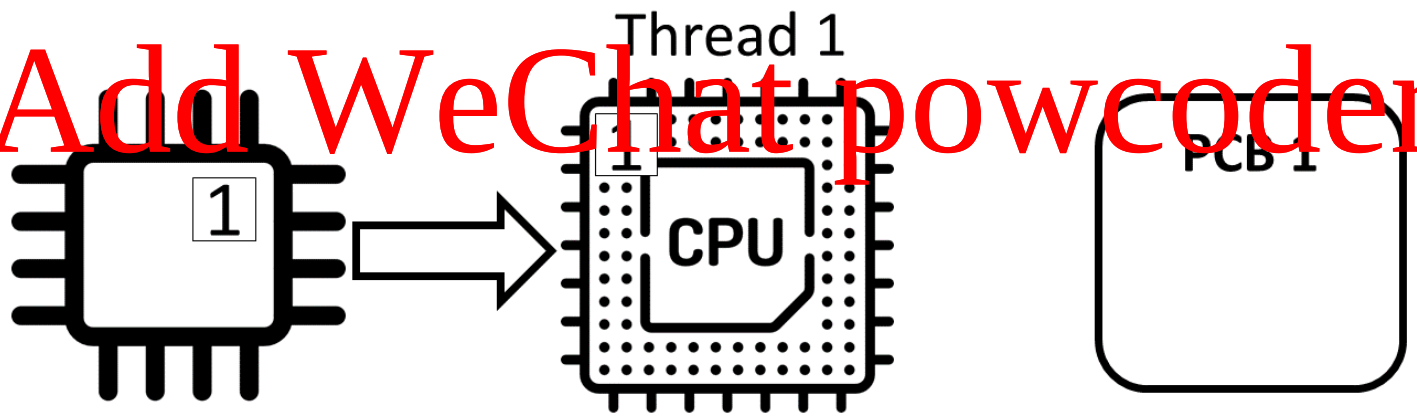
[2]Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

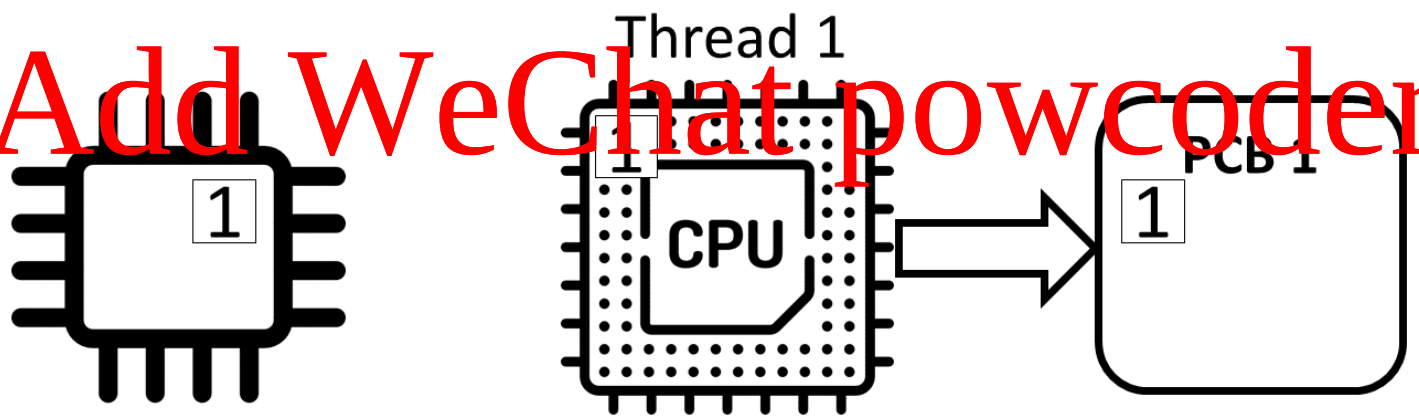
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

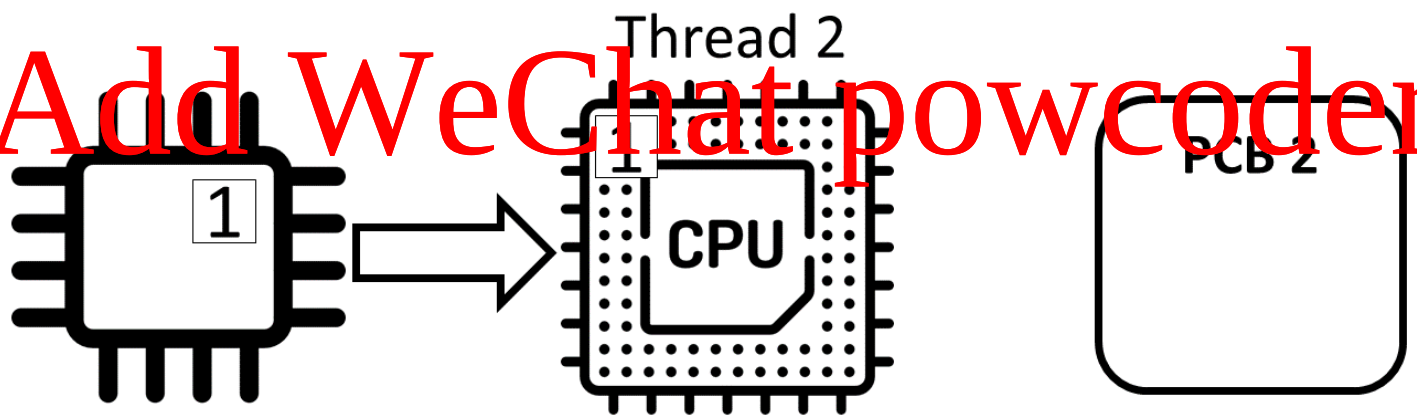
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

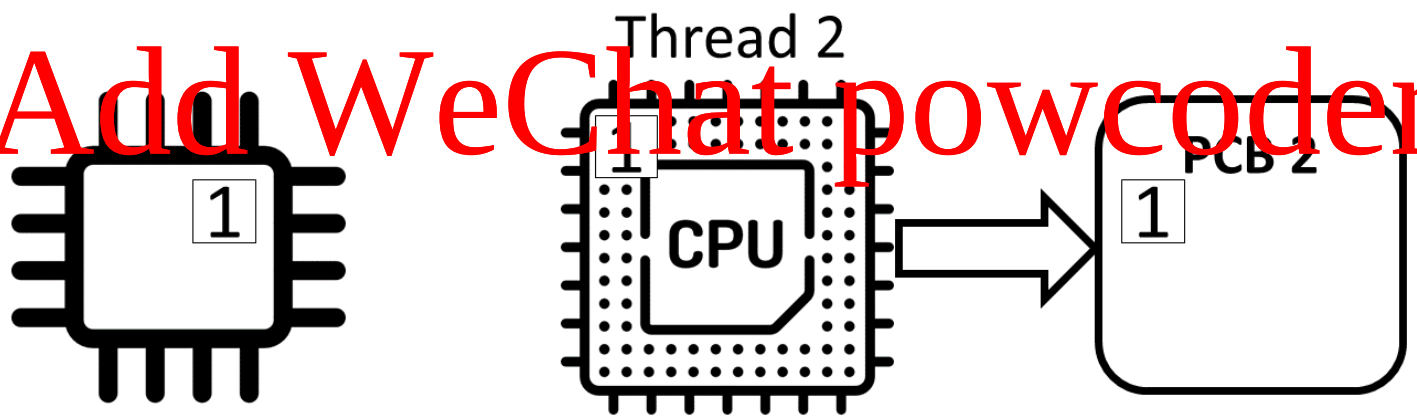
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

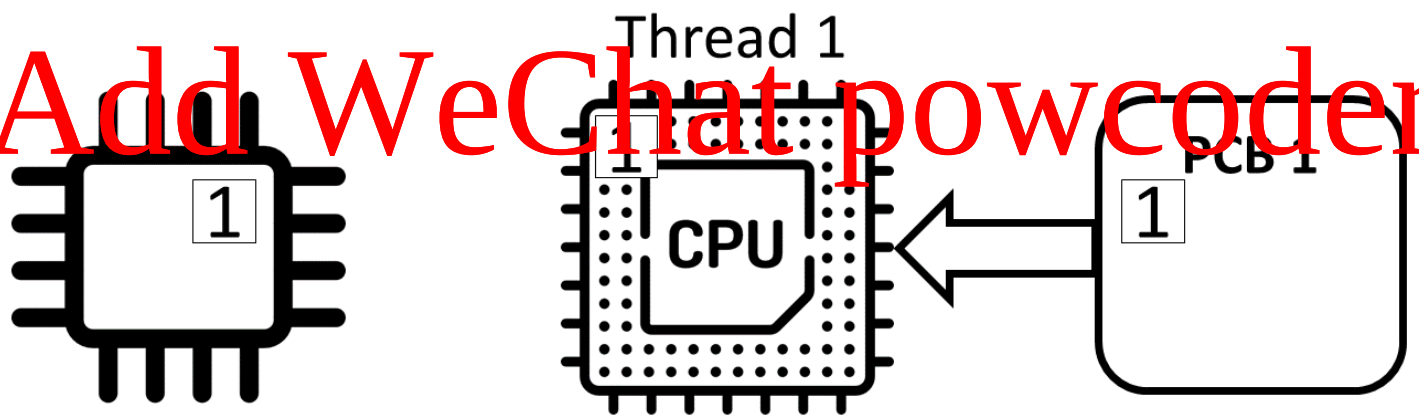
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

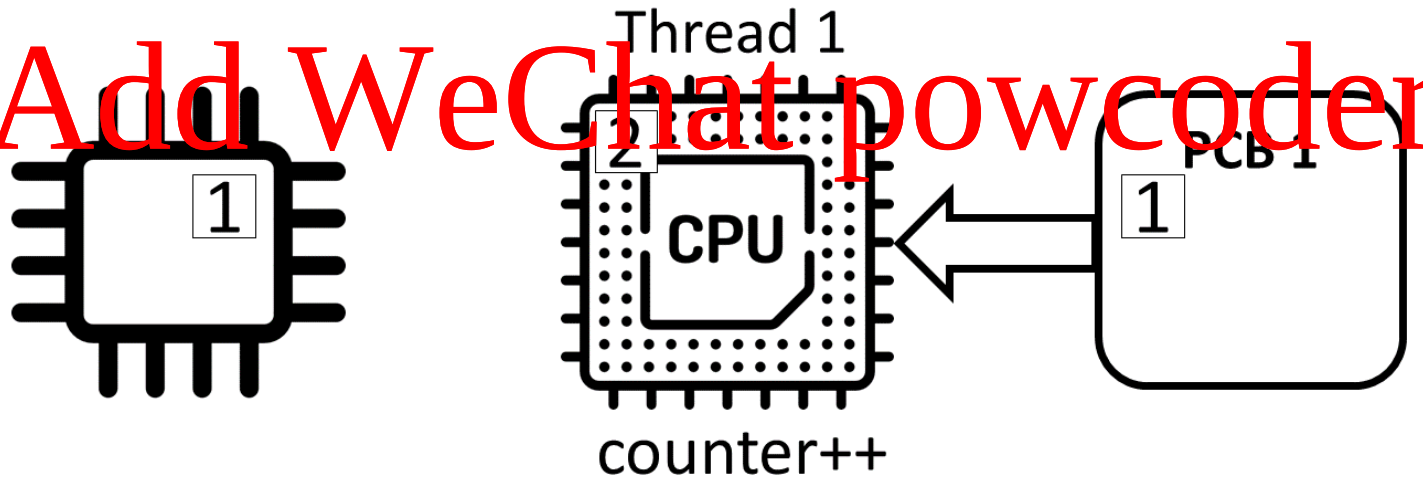
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

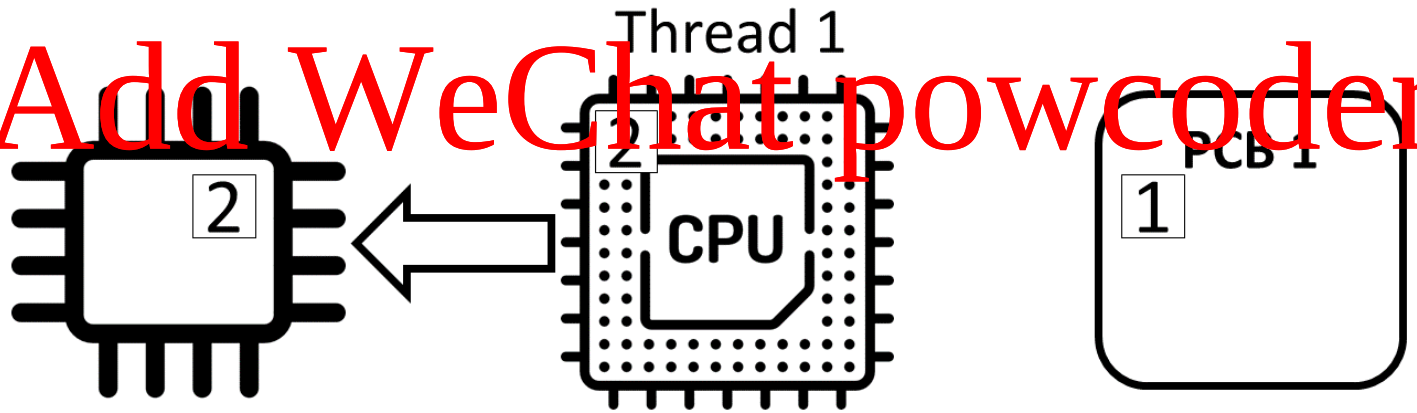
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

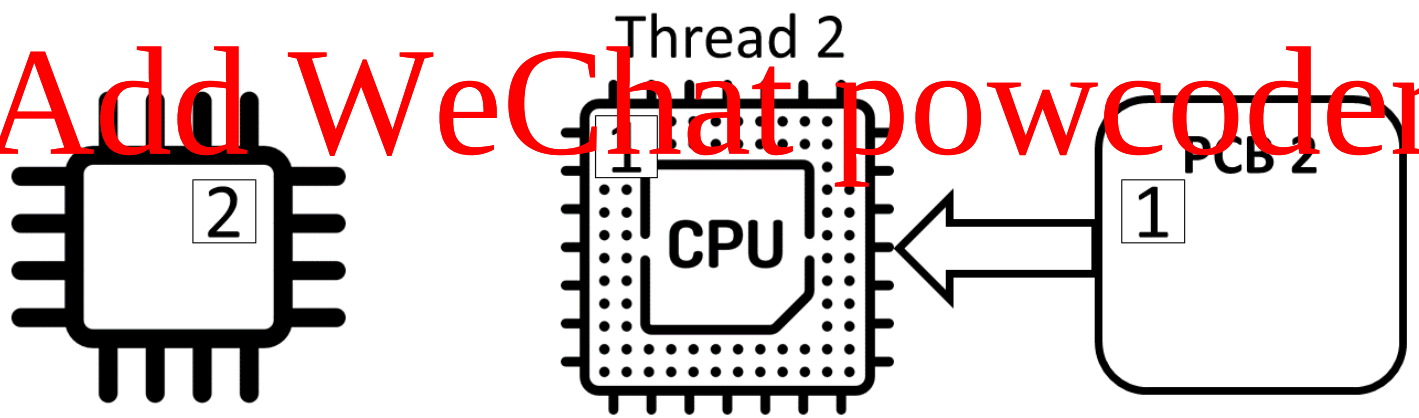
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

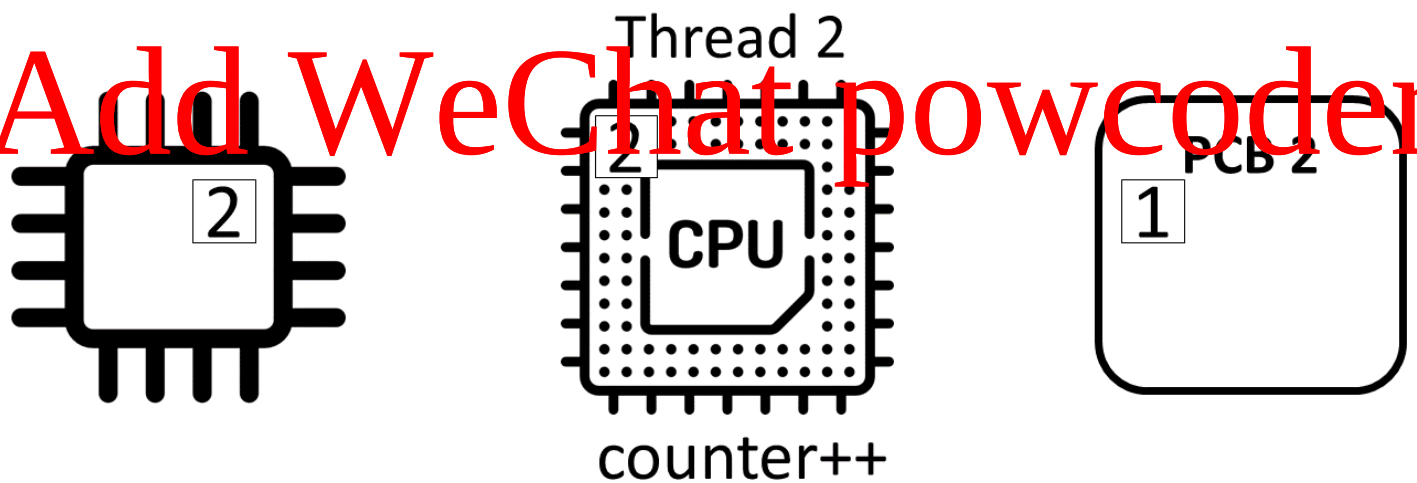
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

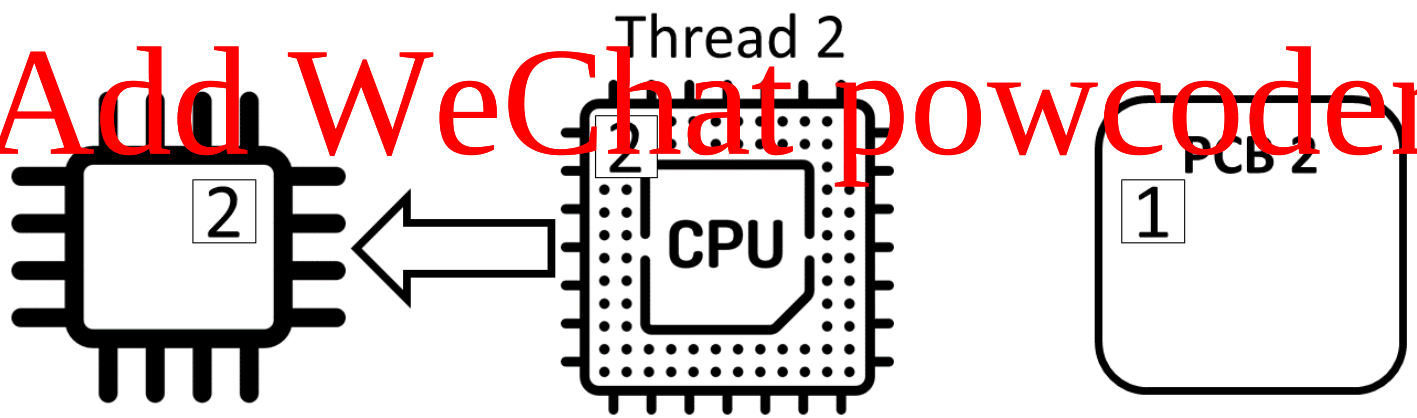
[3] Icons from https://www.flaticon.com/

Example[3]

Incrementing a counter

```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

```
Thread 2:
...
...
Read counter -> register (= 1)
...
...
Add 1 to register value (= 2)
Store register in counter (= 2)
```

[3] Icons from https://www.flaticon.com/

Example 2

Shared procedures

- Consider the following **code shared** between threads/processes
- chin and chout **shared global variables**


```
void print()
{
    chin = getchar();
    chout = chin;
    putchar(chout);
}
```

Example 2

Shared procedures

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they do **NOT** interact):


```
Thread 1                    Thread 2
...                         ...
chin = getchar(); ...
chout = chin;               ...
putchar(chout);             ...
...                         chin = getchar()
...                         chout = chin;
...                         putchar(chout);
```

Example 2

Shared procedures

- Consider **two processes/threads** and the following **interleaved sequence of instructions** (they **DO** interact):


```
Thread 1                    Thread 2
...                         ...
chin = getchar(); ...
...                                 chin = getchar();
chout = chin;               ...
putchar(chout);             ...
...                                 chout = chin;
...                                 putchar(chout);
```

Example 3
Bounded Buffers

- Consider a **bounded buffer** in which *N* **items** can be stored
- A **counter** is maintained to count the number of items currently in the buffer
  - **Incremented** when an item is **added**
  - **Decremented** when an item is **removed**
- Similar **concurrency problems** as with the calculation of sums happen in the bounded buffer (producer/consumer) problem

Example 3

Bounded Buffers – Producer/Consumer

```
// producer
while (true) {
        // while buffer is full
        while (counter == BUFFER_SIZE); /* do nothing */
        // Produce item
        buffer[in] = new_item;
        in = (in + 1) % BUFFER_SIZE;
        counter++;
}
```

```
// consumer
while (true) {
        // wait until items in buffer
        while (counter == 0);  /* do nothing */
        // Consume item
        consumed = buffer[out];
        out = (out + 1) % BUFFER_SIZE;
        counter--;
}
```

Race Conditions

Definition

- A **race condition occurs** when multiple threads/processes **access shared data** and the result is dependent on **the order in which the instructions are interleaved**
- We will discuss **mechanisms** to provide **controlled/synchronised** access to data and **avoid race conditions**

Concurrency within the OS

Data Structures

- **Kernels are preemptive** these days ($\Leftrightarrow$ non-preemptive)
  - **Multiple processes/threads are running** in the kernel
  - I.e. kernel **processes can be interrupted** at any point
- The kernel maintains **data structures**, e.g. process tables, memory structures, open file lists, etc.
  - These data structures are accessed **concurrently/in parallel**
  - These can be subject to **concurrency issues**
- The OS must make sure that interactions within the OS **do not result in race conditions**

- Processes **share resources**, including memory, files, processor time, printers, I/O devices, etc.
- The operating system must:
  - The operating system must provide **locking mechanisms** to implement/support **mutual exclusion** (and **prevent starvation and deadlocks**)
  - **Allocate and deallocate** these resources safely (i.e. avoid interference, deadlocks and starvation)

- A **critical section** is a set of instructions in which **shared resources** between processes/threads (e.g. variables) **are changed**
- **Mutual exclusion** must be enforced **for critical sections**
  - Only **one process at a time** should be in the critical section (mutual exclusion)
  - Processes have to **get "permission"** before entering their critical section (e.g. **request** a lock, **hold** the lock, **release** the lock)

Mutual Exclusion

Definition


```
do
{
        ...
        // ENTRY to critical section
        critical section, e.g.counter++;

        // EXIT critical section

        remaining code
        ...

} while (...);
```

Critical Sections, Mutual Exclusion

Definition

- Any **solution to the critical section problem** must satisfy the following **requirements**:
  - **Mutual exclusion**: only one process can be in its critical section at any one point in time
  - **Progress**: any process must be able to enter its critical section at some point in time
    - Processes/threads in the **"remaining code" do not influence** access to critical sections
  - **Fairness/bounded waiting**: fairly distributed waiting times/processes cannot be made to wait indefinitely
- These requirements have to be satisfied, **independent of the order** in which sequences are executed

- **Approaches** for mutual exclusion can be:
  - **Software based**: Peterson's solution
  - **Hardware based**: `test_and_set()` `swap_and_compare()`
  - **Based on**:
    - Mutexes
    - Semaphores
    - Monitors (software construct within the programming languages)
- In addition to mutual exclusion, **deadlocks** have to be prevented

Deadlocks

Example

- Assume that `X` and `Y` are **mutually exclusive resources** (for instance, "locks" that can only be held by one process/thread at a time)
- Thread `A` and `B` need to **acquire both resources** ("locks"), and request them in **opposite orders**

- The following **sequence of events** could occur in a **multi-programmed system**:


```
THREAD A:
request resource X
acquire resource X
...
...
request resource Y
...
```

```
THREAD B:
...
...
request resource Y
acquire resource Y
...
request resource X
...
```

Deadlocks
Definition

Tanenbaum

*"A set of processes/threads is deadlocked if **each process/thread** in the set is waiting for **an event** that only the **other process/thread** in the set can cause"*

- Each **deadlocked process/thread** is **waiting for** a resource held by **another deadlocked process/thread** (which cannot run and hence cannot release the resources)
- This can happen between **any number of processes/threads** and for **any number of resources**

Figure: Deadlocks

Deadlocks

Minimum Conditions

- **Four conditions** must hold for deadlocks to occur (Coffman et al. (1971)):
  - **Mutual exclusion**: a resource can be assigned to at most one process at a time
  - **Hold and wait condition**: a resource can be held while requesting new resources
  - **No preemption**: resources cannot be forcefully taken away from a process
  - **Circular wait**: there is a circular chain of two or more processes, waiting for a resource held by the other processes

- **No deadlocks** can occur if one of the conditions is **not satisfied**

Deadlocks

Minimum Conditions

- **Four conditions** must hold for deadlocks to occur (Coffman et al. (1971)):
  - **Mutual exclusion**: a resource can be assigned to at most one process at a time
  - **Hold and wait condition**: a resource can be held while requesting new resources
  - **No preemption**: resources cannot be forcefully taken away from a process
  - **Circular wait**: there is a circular chain of two or more processes, waiting for a resource held by the other processes

- **No deadlocks** can occur if one of the conditions is **not satisfied**
- If your **coursework solution deadlocks**, check for the **order in which resources are requested**

- **Problems with synchronisation** and concurrent/parallel code
- Concurrency from an **OS perspective**
- Concept of **mutual exclusion** and **deadlocks**
- Requirements and **approaches** for mutual exclusion