Operating Systems and Concurrency

Concurrency 1

COMP2007

Dan Marsden
(Geert De Maere)
{Geert.DeMaere,Dan.Marsden}@Nottingham.ac.uk

University Of Nottingham
United Kingdom

2023

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 thread can be **interrupted at any point in time** (timer, I/O)
  - The process state is **saved** in the **process control block**
- The outcome of programs may become **unpredictable**:
  - Sharing data can lead to **inconsistencies** - we can be interrupted part way through doing something.
  - The **outcome of execution** may **depend on the order** in which code gets to run on the CPU.

Example

Incrementing a counter

```c
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```c
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```c
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

Incrementing a counter

```
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

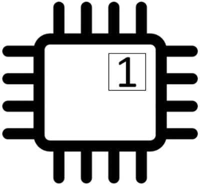
Incrementing a counter

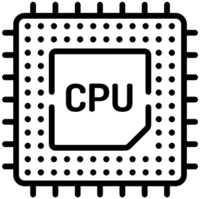
```c
#include <stdio.h>
#include <pthread.h>

int counter = 0;
void* calc(void* param) {
  int const iterations = 50000000;
  for(int i = 0; i < iterations; i++)
    counter++;
  return 0;
}

int main() {
  pthread_t tid1 = 0,tid2 = 0;
  pthread_create(&tid1, NULL, calc, 0);
  pthread_create(&tid2, NULL, calc, 0);
  pthread_join(tid1,NULL);
  pthread_join(tid2,NULL);
  printf("The value of counter is: %d\n", counter);
}
```

Example

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"[1]



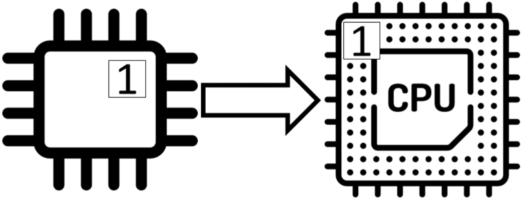


[1] Icons from `https://www.flaticon.com/`

Example

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"[1]

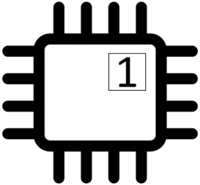


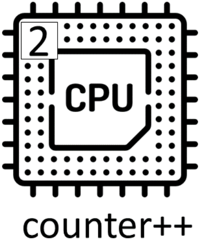
[1] Icons from `https://www.flaticon.com/`

Example

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”[1]



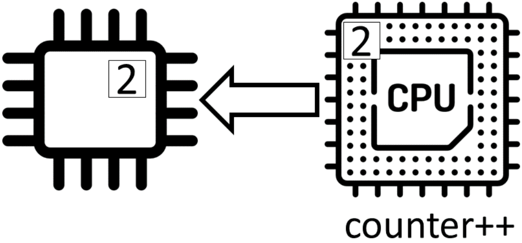


[1] Icons from `https://www.flaticon.com/`

Example

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"[1]

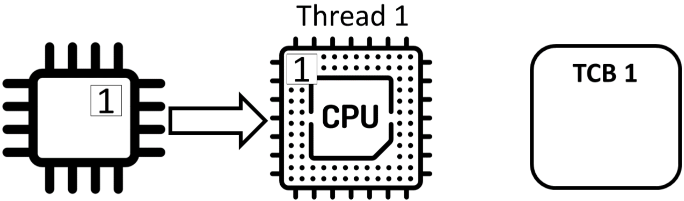


[1] Icons from `https://www.flaticon.com/`

Example

Incrementing a counter

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

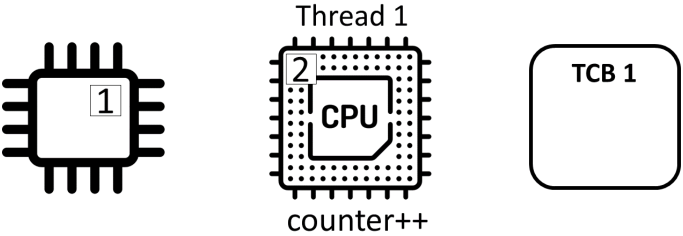


Example

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)
```

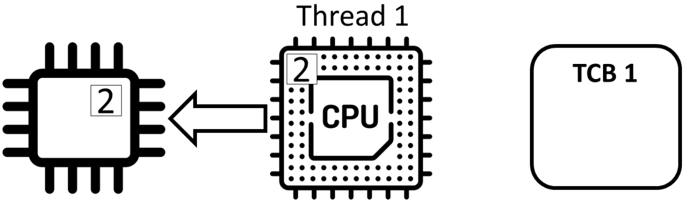


Example

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)
```

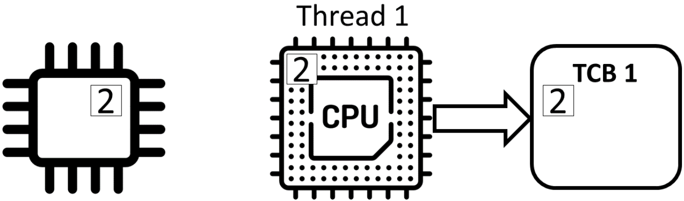


Example

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)
```

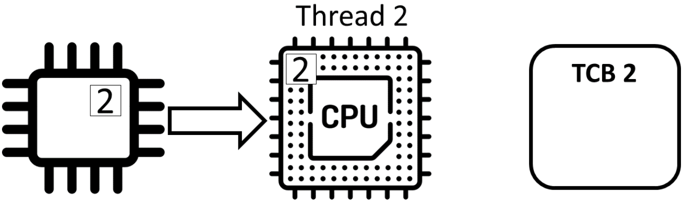


Example

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)
```

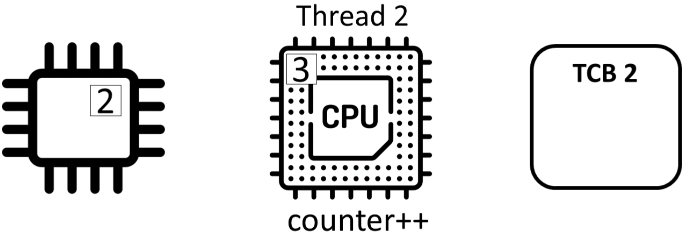


Example

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)
```

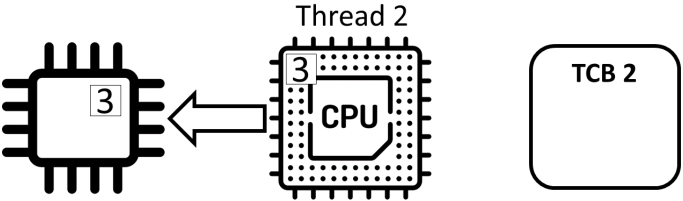


Example

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)
```

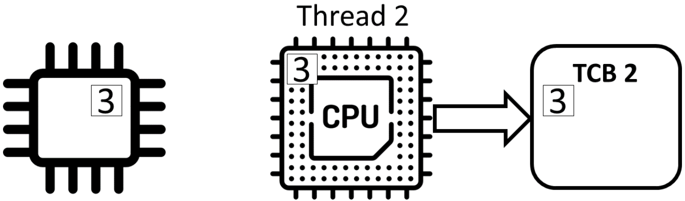


Example

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)
```

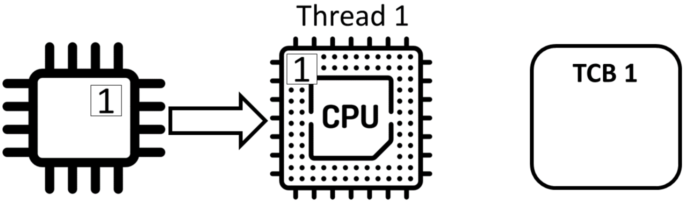


Example

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)
```

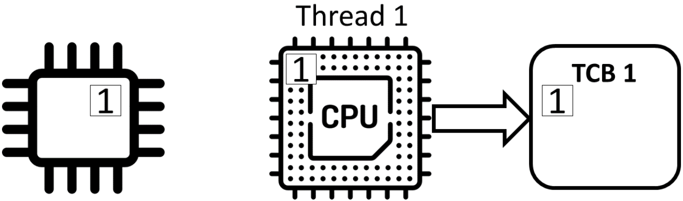


Example

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)
```

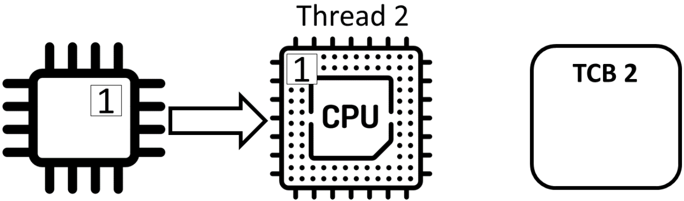


Example

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)
```

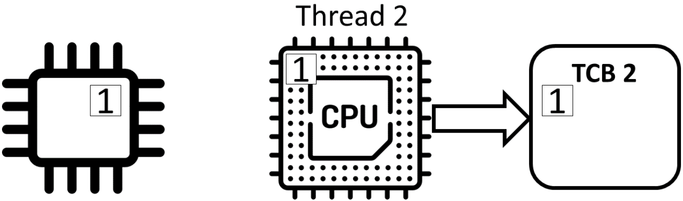


Example

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)
```

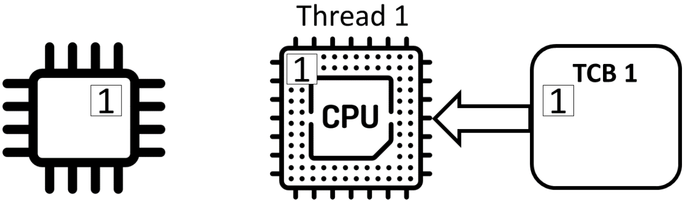


Example

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)
```

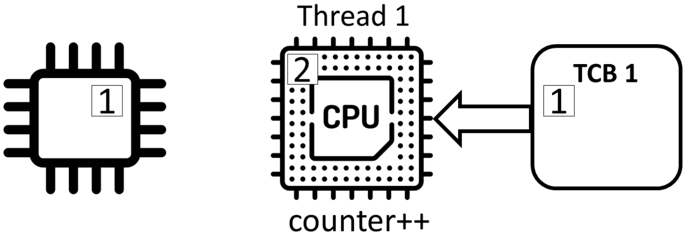


Example

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)
```

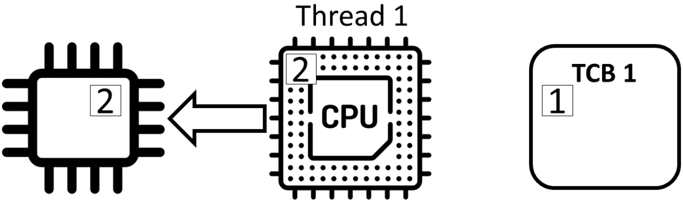


Example

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)
```

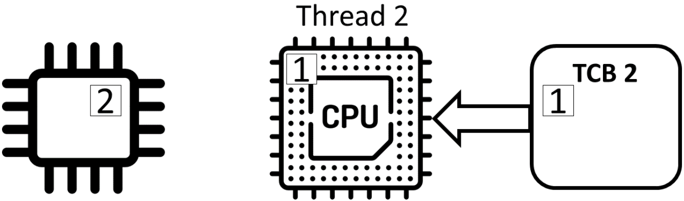


Example

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)
```



Example

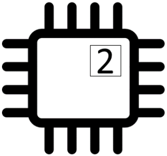
Incrementing a counter

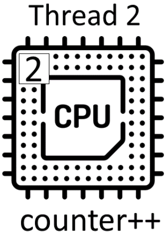
Thread 1:

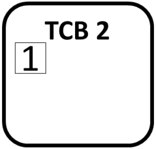
```
Thread 1:
...
Read counter -> register (= 1)
...
Add 1 register value (= 2)
Store value in counter (= 2)
...
...
```

Thread 2:

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





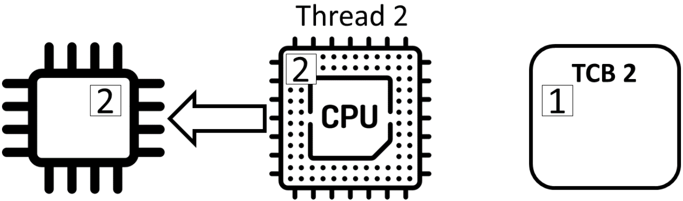


Example

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)
```



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:
...
chin = getchar(); ...
...
chout = chin;
putchar(chout);
...
...
```

```
Thread 2:
...

chin = getchar();
...
...
chout = chin;
putchar(chout);
```

- 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 when multiple threads read and write to the bounded buffer.

Example 3

Bounded Buffers – Producer/Consumer

```
// producer
while (true) {
  while (counter == BUFFER SIZE);
  buffer[in] = new_item;
  in = (in + 1) % BUFFER_SIZE;
  counter++;
}


// consumer
while (true) {
  while (counter == 0);
  consumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter--;
 }
```

Race Conditions

Definition

- Code has a **race condition** if its behaviour is dependent on the timing of when computation is performed.
- A race condition typically occurs when multiple threads **access shared data** and the result is dependent on **the order in which the instructions are interleaved**.
- We will be interested in **mechanisms** to provide **synchronised** access to data and **avoid race conditions**.

Concurrency within the OS

Data Structures

- **Kernels are preemptive** these days
  - **Multiple threads are running** in the kernel.
  - Kernel **code can be interrupted** at any point.
- The kernel maintains **data structures** such as process tables and open file lists:
  - These data structures may be accessed **concurrently**.
  - These can be subject to **concurrency issues**.
- The OS must make sure that interactions within the OS **do not result in race conditions.**

Critical Sections and Mutual Exclusion

- As race conditions can lead to wild, unpredictable and plain wrong behaviour, we want software abstractions help to prevent them.
- A **critical section** is a section of code that can only be run by one thread at a time. This property is referred to as **mutual exclusion**.
- The question then becomes how to enforce mutual exclusion? Potentially:
  - The O/S and compiler could provide **direct support for critical sections** as a programming primitive.
  - The O/S and compiler provide **locks** which can be held by at most one thread at a time.

Mutual Exclusion

Definition

```
do {
  ...
    // ENTRY to critical section

    critical section, e.g.counter++;

  // EXIT critical section

  // remaining code
    ...
} while (...);
```

Critical Sections, Mutual Exclusion

Definition

- A **solution to the critical section problem** should 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 computations are executed

Enforcing Mutual Exclusion

Approaches

- A standard approach to enforcing mutual exclusion is via locks known as **mutexes**. These can be implemented in various ways:
  - **Software based**: Peterson's solution
  - **Hardware based**: `test_and_set()`, `swap_and_compare()`
  - **O/S based** - the operating system blocks processes waiting for the lock.
- Unfortunately, **mutexes** and other concurrency primitives such as **semaphore** introduce a new problem - **deadlocks**.

Deadlocks

Example

- Assume that `X` and `Y` are **mutexes.**
- Thread `A` and `B` need to **acquire both mutexes**, and request them in **opposite orders**.
- The following **sequence of events** could occur in a **multi-programmed system**:

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

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

Tanenbaum

“A set of threads is deadlocked if ***each thread*** *in the set is waiting for* ***an event*** *that only the* ***other thread*** *in the set can cause”*

- Each **deadlocked thread** is **waiting for** a resource held by **an other deadlocked thread** (which cannot run and hence cannot release the resources)
- This can happen between **any number of 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 whilst 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**

- **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 whilst 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**

Test your understanding

- The code `x != y` doesn't modify anything. Is it certain to occur atomically?
- Can race conditions or deadlocks occur in practice on a machine with a single hardware thread?
- Can two threads running the same function deadlock against each other?