MS Windows Concurrency Mechanisms

Prepared By
SUFIAN MUSSQAA AL-MAJMAIE
163103058

April 2017

Basic of Concurrency

In multiple processor system, it is possible not only to interleave processes/threads but to **overlap** them as well. Both techniques can be viewed as examples of concurrent processing and both present the same problems such as in sharing (global) resources e.g. global variables and in managing the allocation of resources optimally e.g. the request use of a particular I/O channel or device. The following figure try to describe the interleaving the processes. P stands for process and t is time.

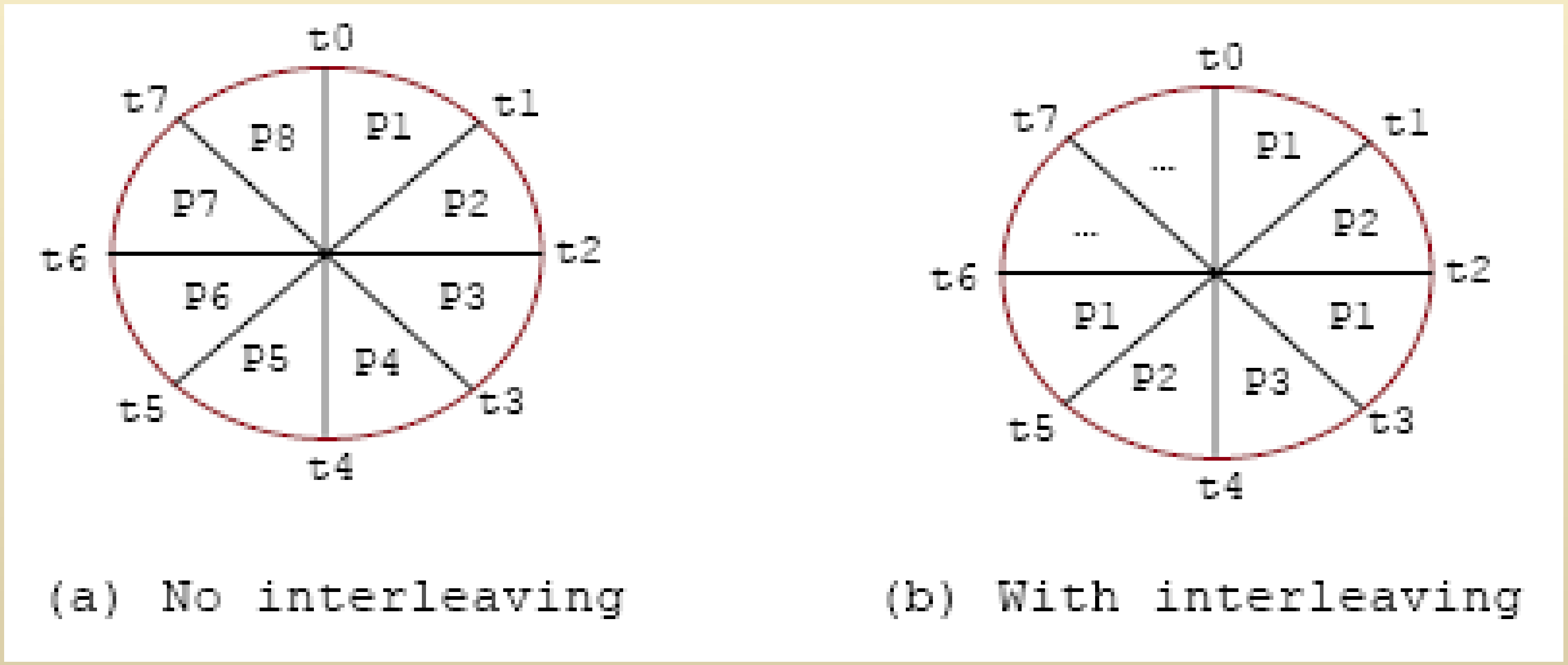


Figure 1: Interleave concept.

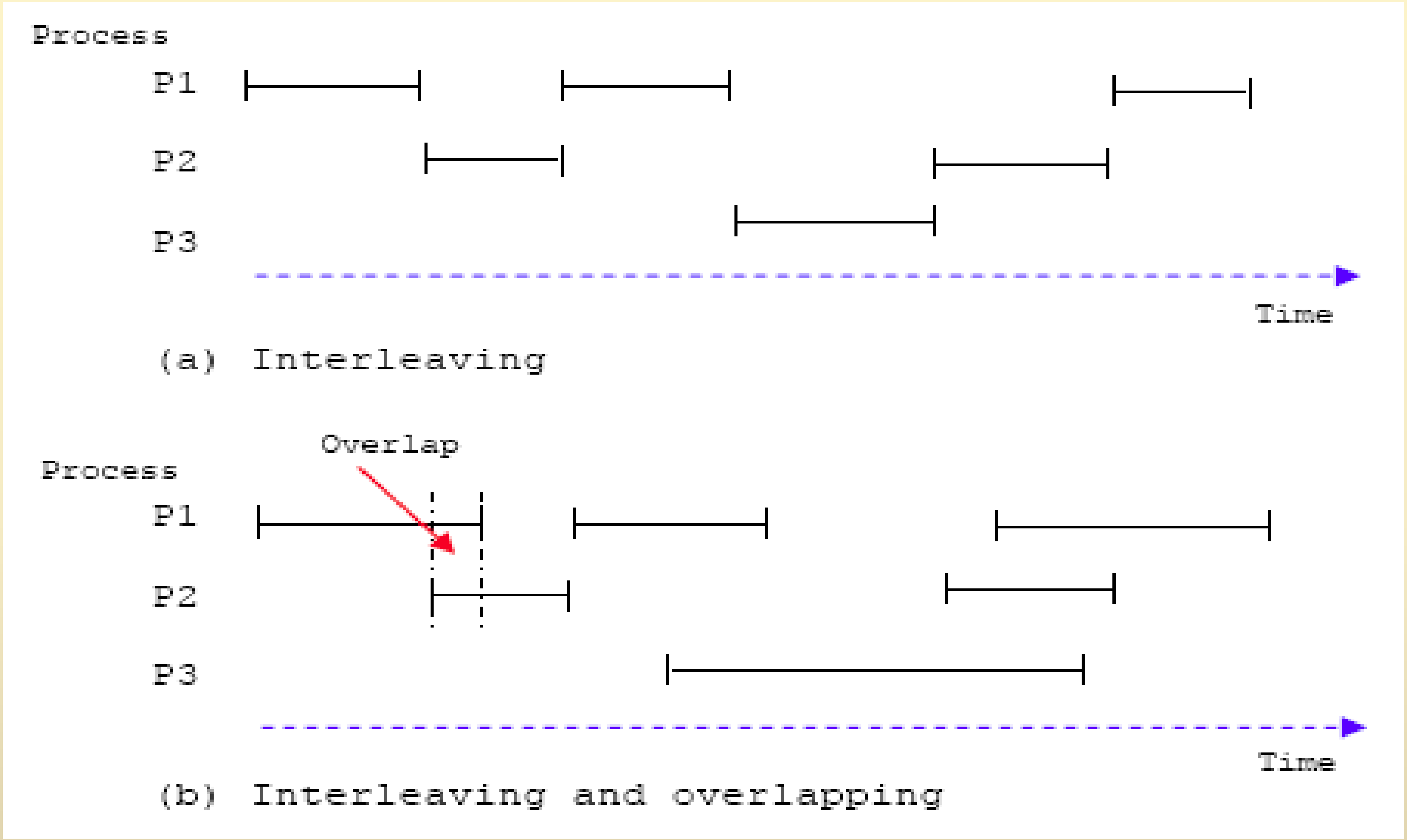
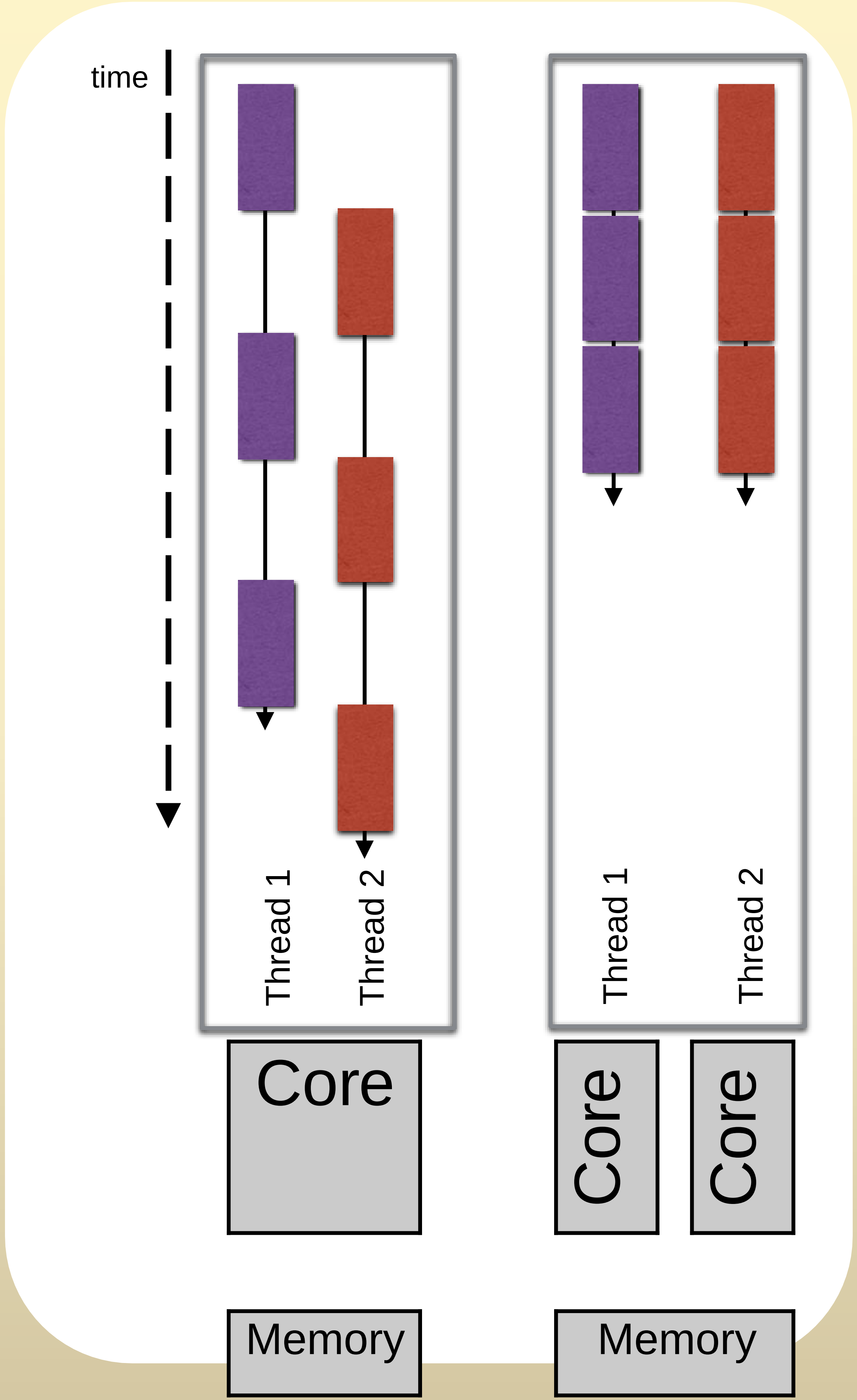


Figure 2: Process/thread Interleaving and overlapping

Concurrency

- Management of **concurrent activities** in Windows operating System
  - Multiple applications *in progress* at the same time, non-sequential operating system activities
  - Time sharing for interleaved execution
  - Demands **dispatching** and **synchronization**
- **Parallelism:** Actions are executed *simultaneously*
  - Demands **parallel hardware**
  - Relies on a concurrent application



Concurrency is Hard

- Sharing of global resources
  - Concurrent reads and writes on the same variable makes order critical
- Optimal management of resource allocation
  - Process gets control over a I/O channel and is then suspended before using it
- Programming errors become non-deterministic
  - Order of interleaving may / may not activate the bug
- Happens all with **concurrent** execution, which means even on uniprocessors
- **Race condition**
  - The final result of an operation depends on the order of execution
  - Well-known issue since the 60‘s, identified by E. Dijkstra

Race Condition

```
void echo() {
  char_in = getchar();
  char_out = char_in;
  putchar(char_out);
}
```

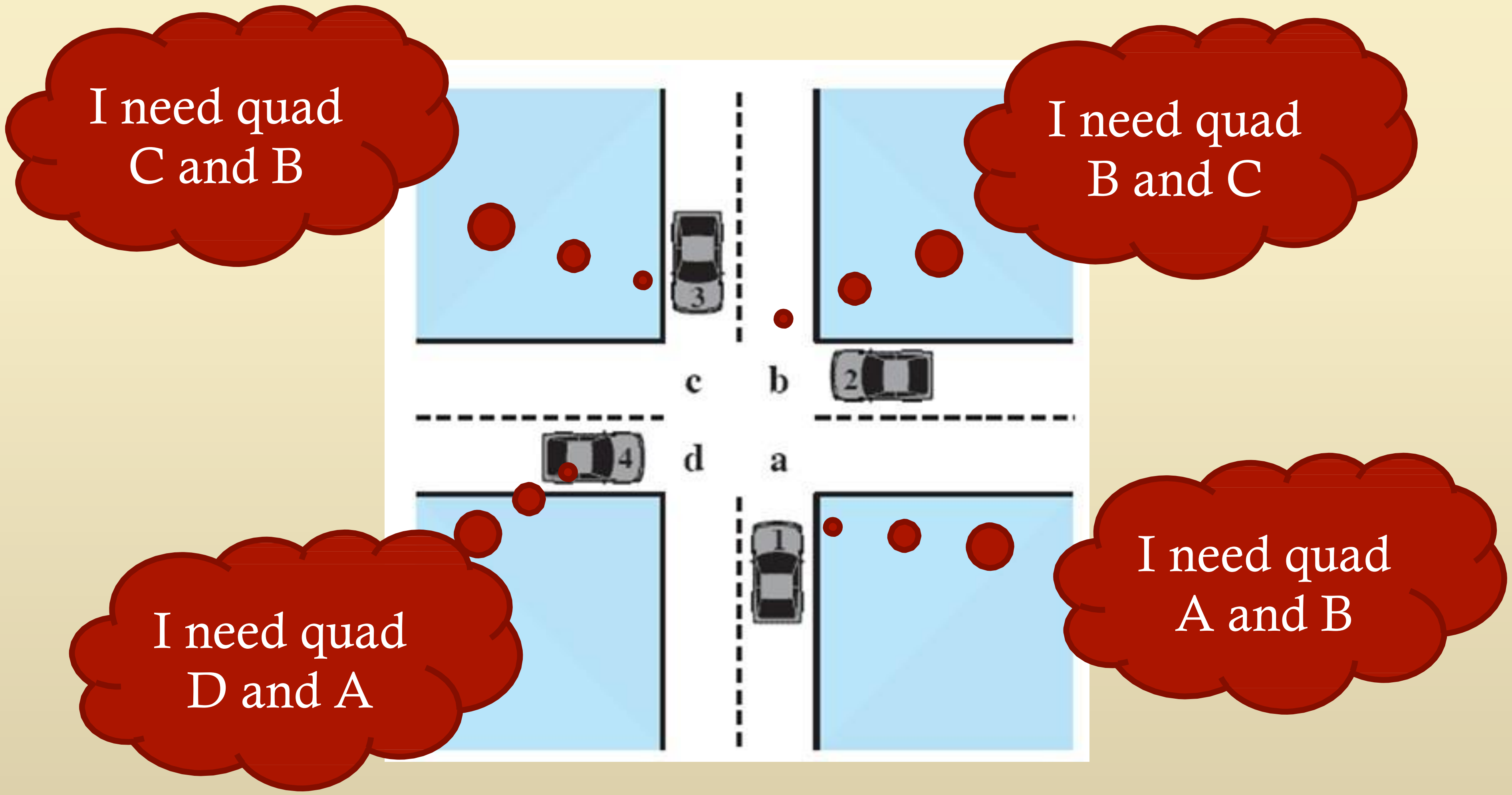
This is a „critical section“

- Executed by two threads on uniprocessor
- Executed by two threads on multiprocessor
- What happens ?

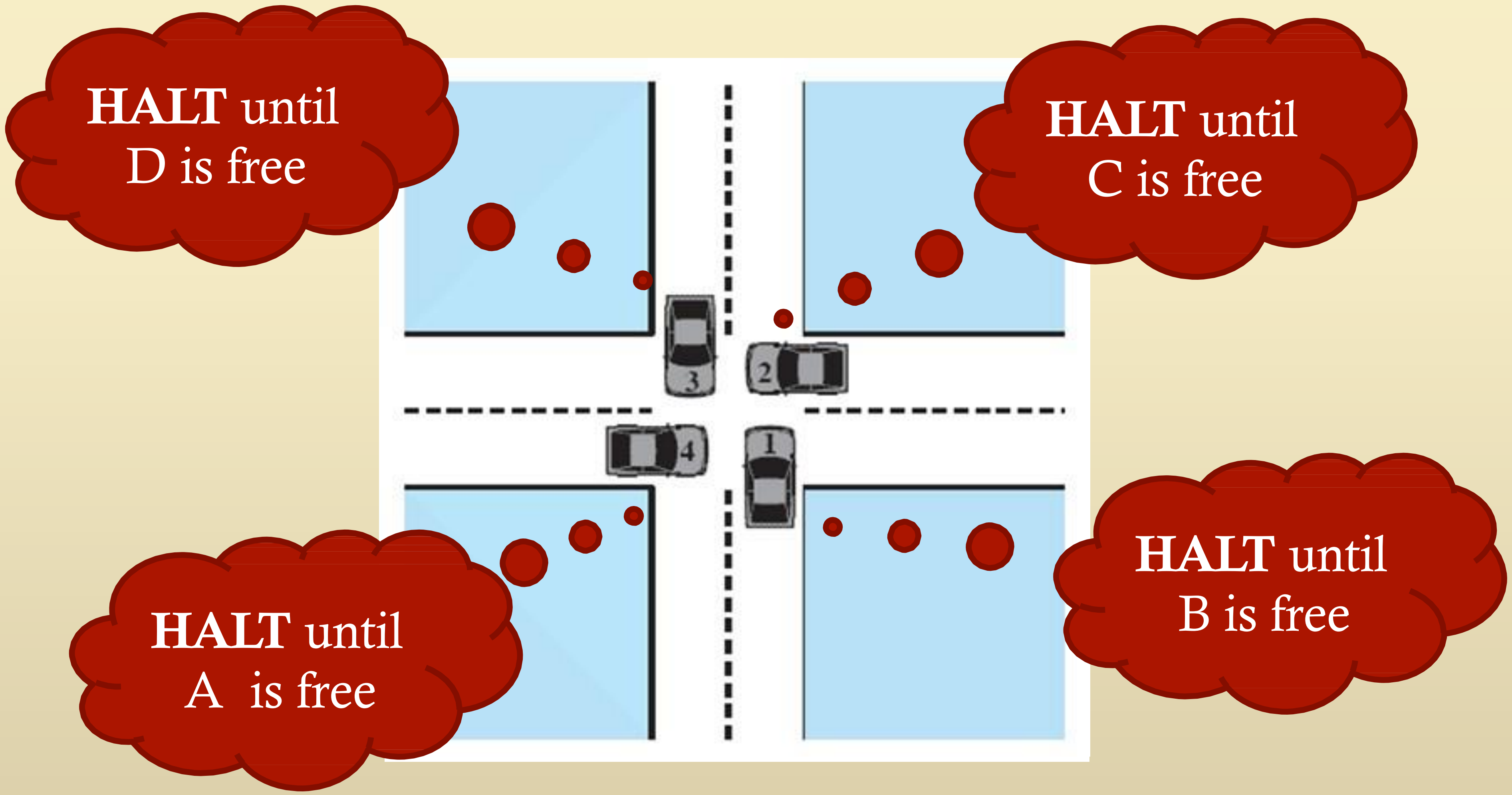
Terminology

- **Deadlock**
  - Two or more processes / threads are unable to proceed
  - Each is waiting for one of the others to do something
- **Livelock**
  - Two or more processes / threads continuously change their states in response to changes in the other processes / threads
  - No global progress for the application
- **Race condition**
  - Two or more processes / threads are executed concurrently
  - Final result of the application depends on the relative timing of their execution

Potential Deadlock



Actual Deadlock



Terminology

- **Starvation**
  - A runnable process / thread is overlooked indefinitely
  - Although it is able to proceed, it is never chosen to run (dispatching / scheduling)
- **Atomic Operation**
  - Function or action implemented as a sequence of one or more instructions
  - Appears to be indivisible - no other process / thread can see an intermediate state or interrupt the operation
  - Executed as a group, or not executed at all
- **Mutual Exclusion**
  - The requirement that when one process / thread is using a resource,
    no other shall be allowed to do that

Critical Section

- n threads all competing to use a shared resource (i.e.; shared data, spaghetti forks)
- Each thread has some code - **critical section** - in which the shared data is accessed
- **Mutual Exclusion** demand
  - Only **one thread at a time** is allowed into its critical section, among all threads that have critical sections for the same resource.
- **Progress** demand
  - If no other thread is in the critical section, the **decision for entering should not be postponed indefinitely**. Only threads that wait for entering the critical section are allowed to participate in decisions. (deadlock problem)
- **Bounded Waiting** demand
  - It must **not be possible** for a thread requiring access to a critical section to be **delayed indefinitely by other threads** entering the section.   (starvation problem)

Critical Section

- Only 2 threads, T0 and T1
- General structure of thread T_i (other thread T_j)
- Threads may share some common variables to synchronize their actions

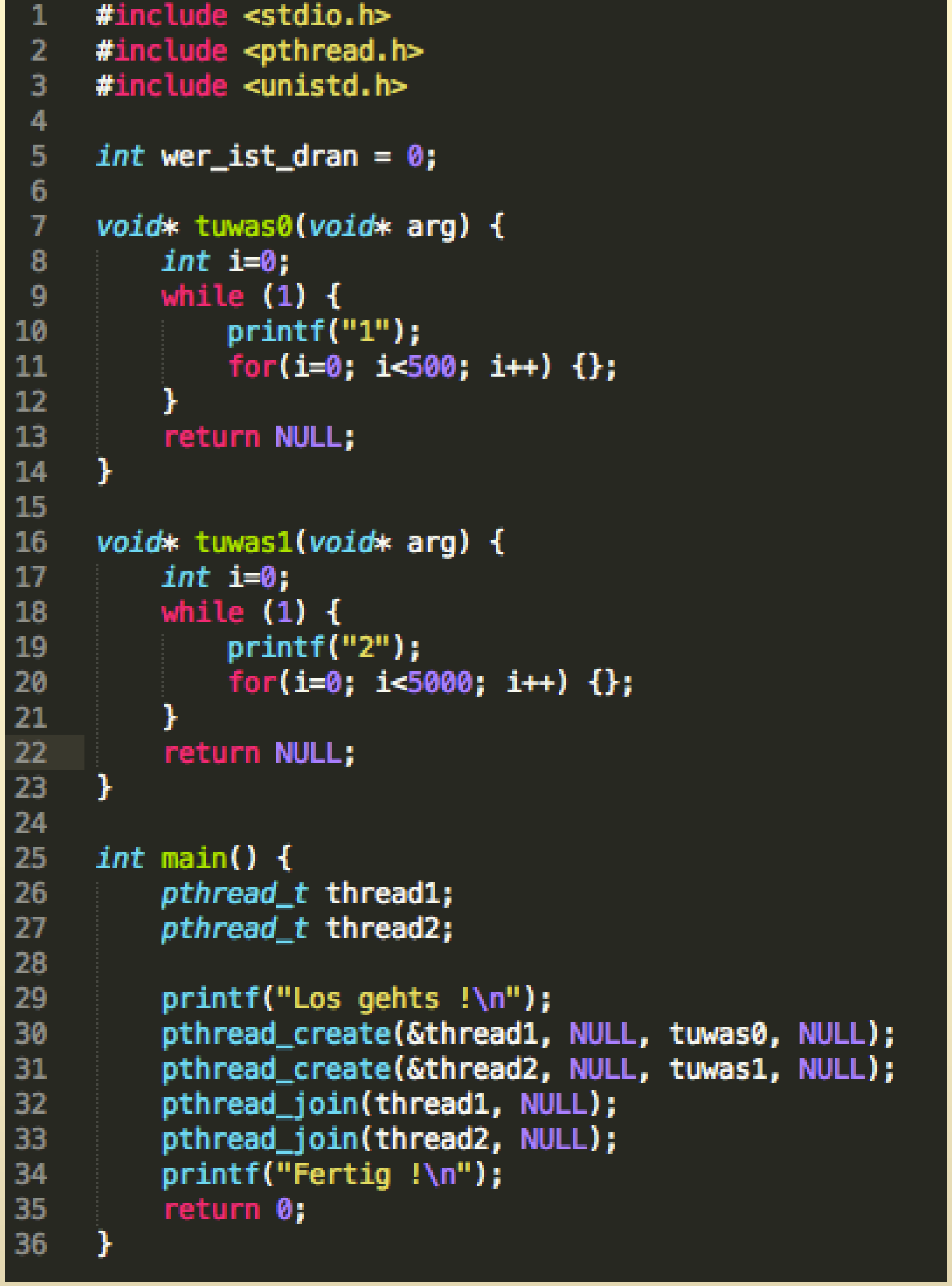
```
do {
    enter section
      critical section
    exit section
      reminder section
  } while (1);
```

Critical Section Protection with Hardware

- Traditional solution was **interrupt disabling**, but works only on multiprocessor
  - Concurrent threads cannot overlap on one CPU
  - Thread will run until performing a system call or interrupt happens
- Software-based algorithms also do not work, due to missing atomic statements
- Modern architectures need **hardware support** with atomic machine instructions
  - **Test and Set** instruction - read & write memory at once
  - If not available, **atomic swap** instruction is enough
- **Busy waiting**, starvation or deadlock are still possible

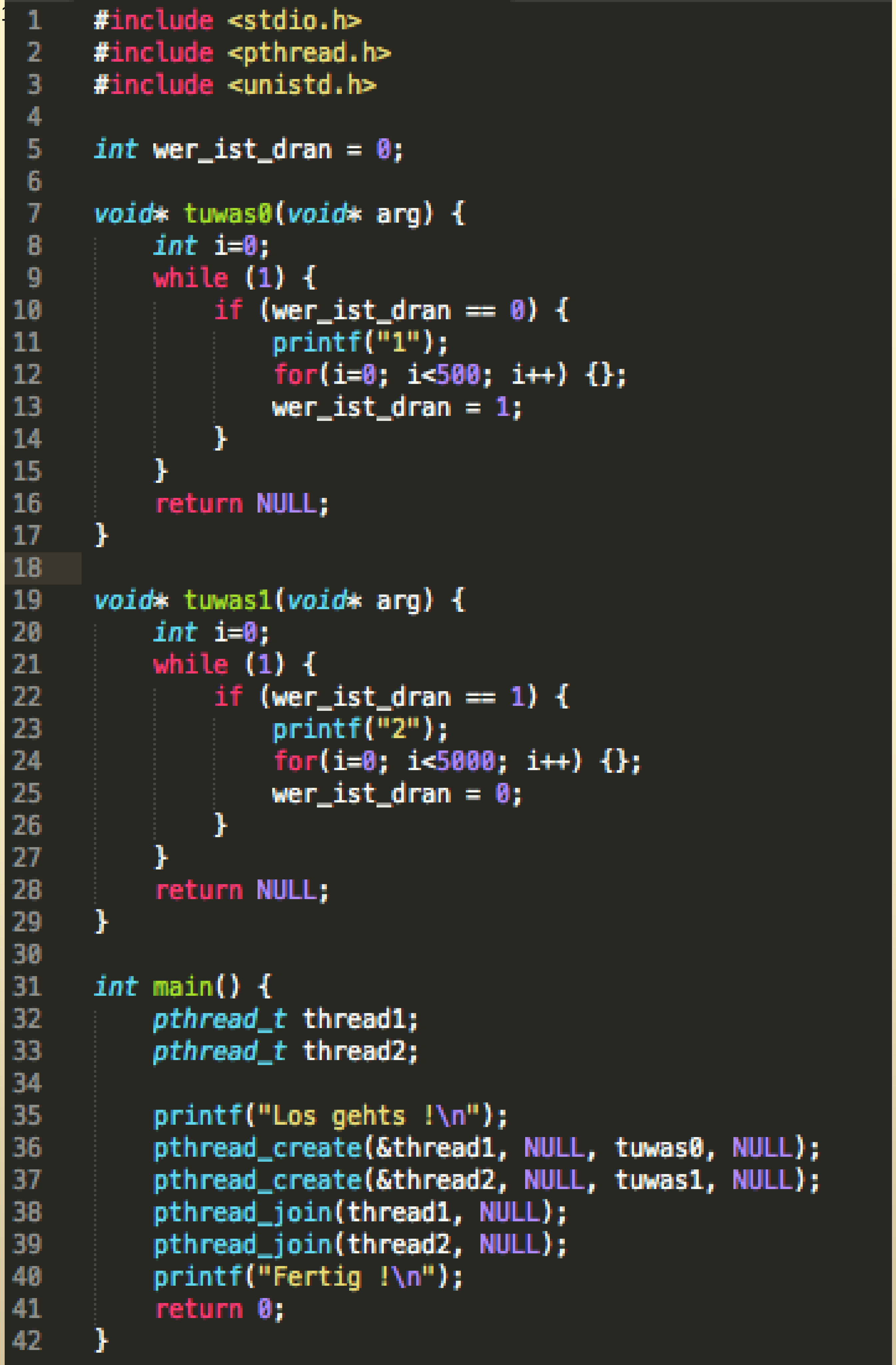
```
#define LOCKED 1
int TestAndSet(int* lockPtr) {
    int oldValue;
    oldValue = SwapAtomic(lockPtr, LOCKED);
    return oldValue;
}
```

```
function Lock(int *lock) {
    while (TestAndSet (lock) == LOCKED);
}
```



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

int wer_ist_dran = 0;

void* tuwas0(void* arg) {
    int i=0;
    while (1) {
        printf("1");
        for(i=0; i<500; i++) {};
    }
    return NULL;
}

void* tuwas1(void* arg) {
    int i=0;
    while (1) {
        printf("2");
        for(i=0; i<5000; i++) {};
    }
    return NULL;
}

int main() {
    pthread_t thread1;
    pthread_t thread2;

    printf("Los gehts !\n");
    pthread_create(&thread1, NULL, tuwas0, NULL);
    pthread_create(&thread2, NULL, tuwas1, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Fertig !\n");
    return 0;
}
```

„Manual“ implementation of a critical section for interleaved output



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

int wer_ist_dran = 0;

void* tuwas0(void* arg) {
    int i=0;
    while (1) {
        if (wer_ist_dran == 0) {
            printf("1");
            for(i=0; i<500; i++) {};
            wer_ist_dran = 1;
        }
    }
    return NULL;
}

void* tuwas1(void* arg) {
    int i=0;
    while (1) {
        if (wer_ist_dran == 1) {
            printf("2");
            for(i=0; i<5000; i++) {};
            wer_ist_dran = 0;
        }
    }
    return NULL;
}

int main() {
    pthread_t thread1;
    pthread_t thread2;

    printf("Los gehts !\n");
    pthread_create(&thread1, NULL, tuwas0, NULL);
    pthread_create(&thread2, NULL, tuwas1, NULL);
    pthread_join(thread1, NULL);
    pthread_join(thread2, NULL);
    printf("Fertig !\n");
    return 0;
}
```

Binary and General Semaphores [Dijkstra]

- Find a solution to allow waiting processes to ‚sleep'
- Special purpose integer called **semaphore**
    - **P**-operation: Decrease value of its argument semaphore by 1 as atomic step
        - Blocks if the semaphore is already zero - **wait** operation
    - **V**-operation: Increase value of its argument semaphore by 1 as atomic step
        - Releases one instance of the resource for other processes - **signal** operation

```
wait (S):
 while (S <= 0);
 S--;   // atomic
signal (S):
 S++;   // atomic
```

```
do {
 wait(mutex);
   critical section
   signal(mutex);
     remainder section
} while (1);
```

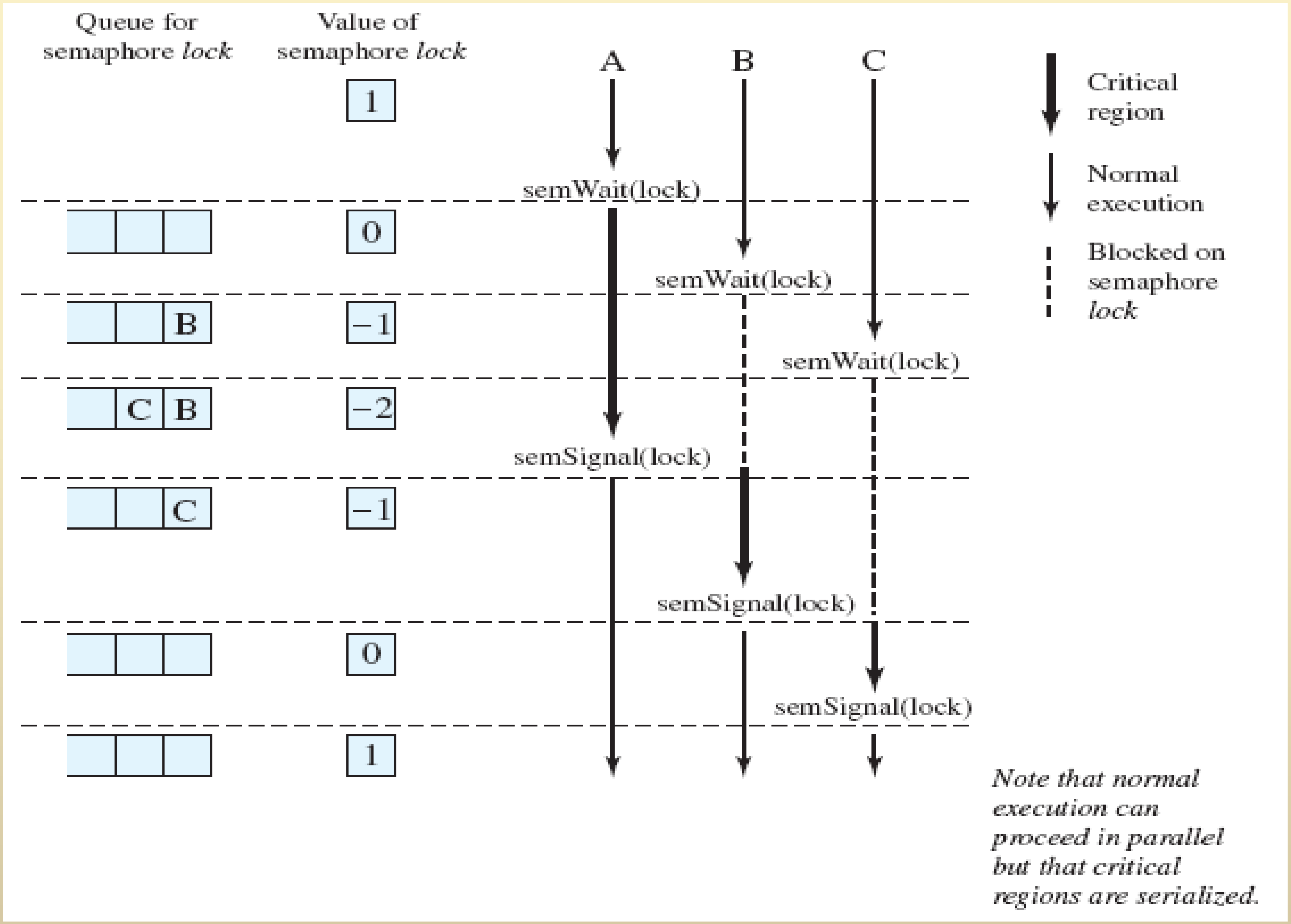
- Solution for critical section shared between N processes
- **Binary semaphore** has initial value of 1, **counting semaphore** of N

Semaphores and Busy Wait

- Semaphores may suspend/resume threads to avoid busy waiting
  - On **wait** operation
    - Decrease value
    - When value <= 0, calling thread is suspended and added to waiting list
    - Value may become negative with multiple waiters
  - On **signal** operation
    - Increase value
    - When value <= 0, one waiting thread is woken up and remove from the waiting list

```
typedef struct {
 int value;
 struct thread *L;
} semaphore;
```

Shared Data Protection by Semaphores


Queue for semaphore *lock*
Value of semaphore *lock*
A
B
C
Critical region
Normal execution
Blocked on semaphore *lock*
1
semWait(lock)
0
semWait(lock)
B
–1
semWait(lock)
C B
–2
semSignal(lock)
C
–1
semSignal(lock)
0
semSignal(lock)
1
Note that normal execution can proceed in parallel but that critical regions are serialized.

<u>References</u> :

- http://www.tenouk.com
- https://computing.llnl.gov
- Modern Operating, 3rd ed., by Andrew Tanenbaum
- Operating System ConcSystemsepts, 7th ed., by Silbershatz, Galvin, & Gagne