Synchronization

Synchronization Basics

Stefano Quer

Dipartimento di Automatica e Informatica

Politecnico di Torino

Critical sections

- Critical Section (**CS**) or Critical Region (**CR**)
  - A section of code, common to multiple processes (or threads), in which these entities can access (read and **write**) shared objects
  - A section of code in which multiple processes (or threads) are competing for the use (read and **write**) of shared resources (e.g., data or devices)
- Concurrent programming is subject to **race conditions**
  - The result of more concurrent processes working on common data depends on the execution order of the processes instructions

Critical sections

- Race conditions could be prevented if
  - No P (or T) executes in the same CS simultaneously
  - No other P (or T) can execute, when a P (or T) executes in the CS
  - The code in the CS is executed by a single P (or T) at a time
  - The code in the CS is executed in mutual exclusion

In other words, Bernstein's conditions must fulfill

Solution

- Solution
  - Establish 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

Access protocol

P_i / T_i

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

P_j / T_j

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

- Every CS is protected by an
  - Enter code (reservation, or prologue)
  - Exit code (release, or epilogue)
- Non-critical sections should not be protected

Synchronization

- To synchronize entities (Ps or Ts) OSs provide appropriate primitives
- Among these primitives, we have **semaphores**
  - Introduced by Dijkstra in 1965
  - Each semaphore is associated to a queue
    - Semaphores do not busy waiting, therefore they do not waste resources
    - Queues are implemented in kernel space by means of a queue of Thread Control Blocks
    - The kernel scheduler decides the queue management strategy (not necessarily FIFO)

Definition

- A semaphore S is a shared structure including
  - A counter
  - A waiting queue, managed by the kernel
  - Both protected by a lock

```
typedef struct semaphore_tag  {
  char lock;
  int cnt;
  process_t *head;
} semaphore_t;
```

Lock variable
Counter
Semaphore list

- Operations on **S** are **atomic**
  - Atomicity is managed by the OS
  - It is impossible for two threads to perform simultaneous operations on the same semaphore

Manipulation functions

- Typical operations on a semaphore S
  - init (S, k)
    - Defines and initializes the semaphore S to the value k
  - wait (S)
    - Allows (in the reservation code) to obtain the access of the CS protected by the semaphore S
  - signal (S)
    - Allows (in the release code) to release the CS protected by the semaphore S
  - destroy (S)
    - Frees the semaphore S

sleep, down, P

wakeup, up, V

They are not the "wait" and "signal" seen with processes

Semaphore primitives

- init(S, k)
  - Defines and initializes semaphore S to value k
  - Two types of semaphores
    - Binary semaphores
      - The value of **k** is only **0** or **1**
    - Counting semaphores
      - The value of **k** is **non negative**

k is a counter

known as "mutex lock"
(mutex ≡ MUTual EXclusion)

```
init (S, k) {
  alloc (S);
  S=k;
}
```

Logical implementation

Atomic operation

Semaphore primitives

- wait(S)
  - If the counter value of **S** is negative or zero blocks the calling T/P
    - If S is negative, its absolute value S indicates the number of waiting threads
  - The counter is decremented at each call

Logical implementation

In the logical versions S is always positive

```
wait (S) {
  while (S<=0);
  S--;
}
```

Real implementations do **not** use busy waiting

Atomic operation

Other possible (and equivalent) logical implementation

```
wait (S) {
  if (S==0) block();
  else S--;
}
```

Semaphore primitives

- wait(S)
  - Originally called **`P()`** from the Dutch language "probeer te verlagen", i.e., "try to decrease"
  - **Not** to be confused with the **wait** system call used to wait for a child process

Logical implementation

```
wait (S) {
  while (S<=0);
  S--;
}
```

In the logical versions S is always positive

Real implementations do **not** use busy waiting

Atomic operation

Other possible (and equivalent) logical implementation

```
wait (S) {
  if (S==0) block();
  else S--;
}
```

Semaphore primitives

- signal(S)
  - Increases the semaphore **S**
    - If **S** counter is negative or zero some T/P was blocked on the semaphore queue, and it can be wakeup
  - Originally called **V()**, from the Dutch language "verhogen", i.e., "to increment"
  - **Not to be confused** with system call **signal** that is used to declare a signal handler

Logical implementation

```
signal (S) {
  S++;
}
```

Atomic operation (register=s;register++;s=register;)

Other possible (and equivalent) logical implementation

```
signal (S) {
  if (blocked())
    wakeup();
  else S++;
}
```

Semaphore primitives

- destroy(S)
  - Release semaphore **S** memory
    - Actual implementations of a semaphore require much more of a simple global variable to define a semaphore
  - This function is often not used in the examples

```
destroy (S) {
  free (S);
}
```

Logical implementation

Synchronization with semaphores

- The use of semaphores is not limited to the critical section access protocol
- Semaphores can be used to solve **any synchronization problem** using
  - An appropriate positioning of semaphores in the code
  - Possibly, more than one semaphore
  - Possibly, additional shared variables

Mutual exclusion with semaphore

```
init (S, 1);
```

P_i / T_i

```
while (TRUE) {
  wait (S);
  CS
  signal (S);
  non critical section
}
```

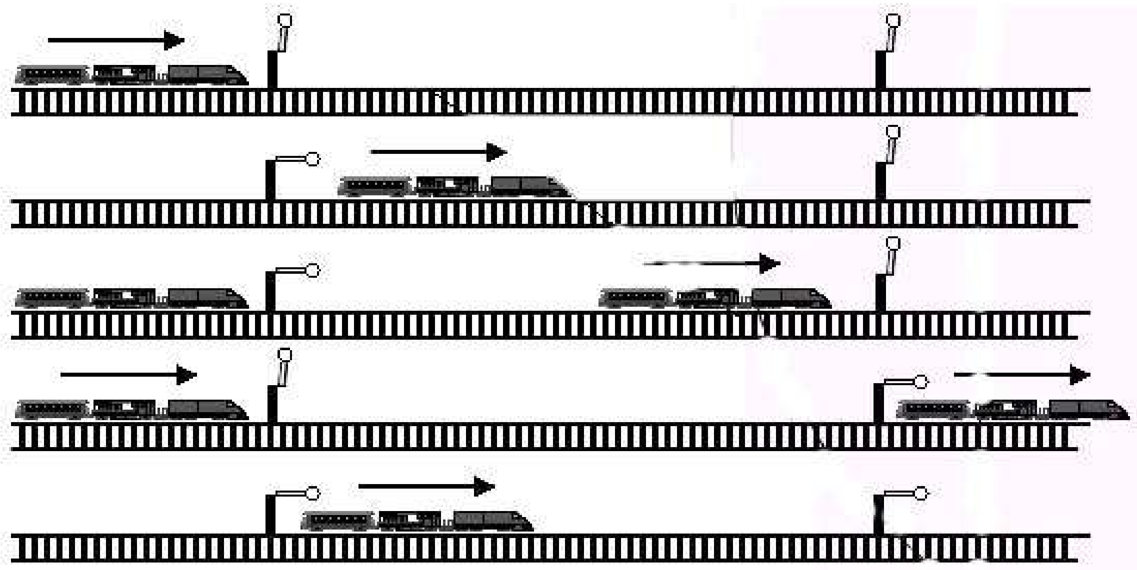
P_j / T_j

```
while (TRUE) {
  wait (S);
  CS
  signal (S);
  non critical section
}
```

Remind:

```
wait (S) {
  while (S<=0);
  S--;
}

signal (S) {
  S++;
}
```



Critical sections of N threads

```
init (S, 1);
...
wait (S);
CS
signal (S);
```

T_1	T_2	T_3	S	queue
			1	
wait			0	
CS_1	wait		-1	T_2
		wait	-2	T_2, T_3
	blocked		-2	T_2, T_3
signal			-2	T_2, T_3
	CS_2	blocked	-1	T_3
	signal		0	
		CS_3	0	
		signal	1	

At most **one** T/P at a time in the critical section

Critical sections of N threads

```
init (S, 2);
...
wait (S);
CS
signal (S);
```

T_1	T_2	T_3	S	queue
			2	
wait			1	
CS_1	wait		0	
	CS_2	wait	-1	T_3
		blocked	-1	T_3
signal		blocked	0	
		CS_3	0	
	signal		1	
		signal	2	

Threads 1 and 2 in their CSs

Threads 2 and 3 in their CSs

At most **two** T/P at a time in the critical section

Pure synchronization

- Synchronize two T/P so that
  - T_j waits T_i
  - Then, T_i waits T_j
  - It is a client-server schema

```
init (S1, 0);
init (S2, 0);
```

T_i / P_i

```
while (TRUE) {
  prepare data
  signal (S1);
  wait (S2);
  get processed data
}
```

T_j / P_j

```
while (TRUE) {
  wait (S1);
  process data
  signal (S2);
  ...
}
```

Exercise

- Given the code of these three threads

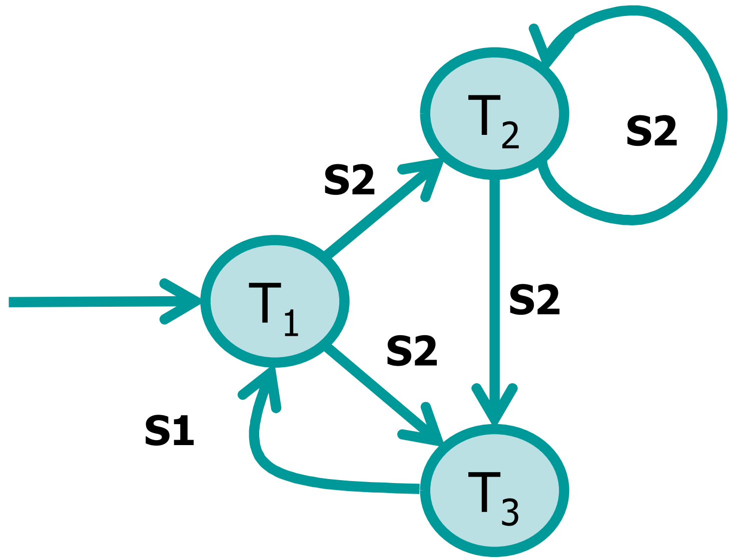
T_1
```
...
while (1) {
  wait (S1);
  T1 code
  signal (S2);
}
...
```

T_2
```
...
while (1) {
  wait (S2);
  T2 code
  signal (S2);
}
...
```

T_3
```
...
while (1) {
  wait (S2);
  T3 code
  signal (S1);
}
...
```

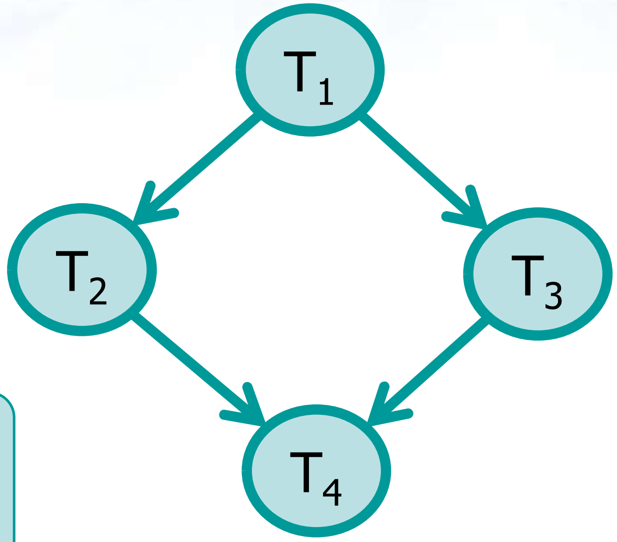
```
init (S1, 1);
init (S2, 0);
```

- Which is the possible execution order?



Exercise

- Implement this precedence graph using semaphores
  - Ts/Ps are not **cyclic**




```
init (S1, 0);
init (S2, 0);
```

T_1
```
T1 code
signal (S1);
signal (S1);
...
```

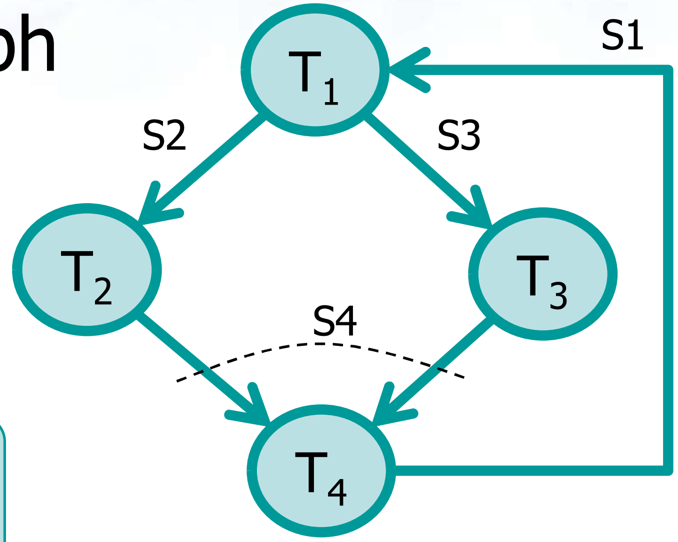
T_2
```
...
wait (S1);
T2 code
signal (S2);
...
```

T_3
```
...
wait (S1);
T3 code
signal (S2);
...
```

T_4
```
...
wait (S2);
wait (S2);
T4 code
```

Exercise

- Implement this precedence graph using semaphores
  - **All** Ts/Ps are cyclic




```
init (S1, 1);
init (S2, 0);
init (S3, 0);
init (S4, 0);
```

T_1
```
while (1) {
  wait (S1);
  T1 code
  signal (S2);
  signal (S3);
}
```

T_2
```
while (1) {
  wait (S2);
  T2 code
  signal (S4);
}
```

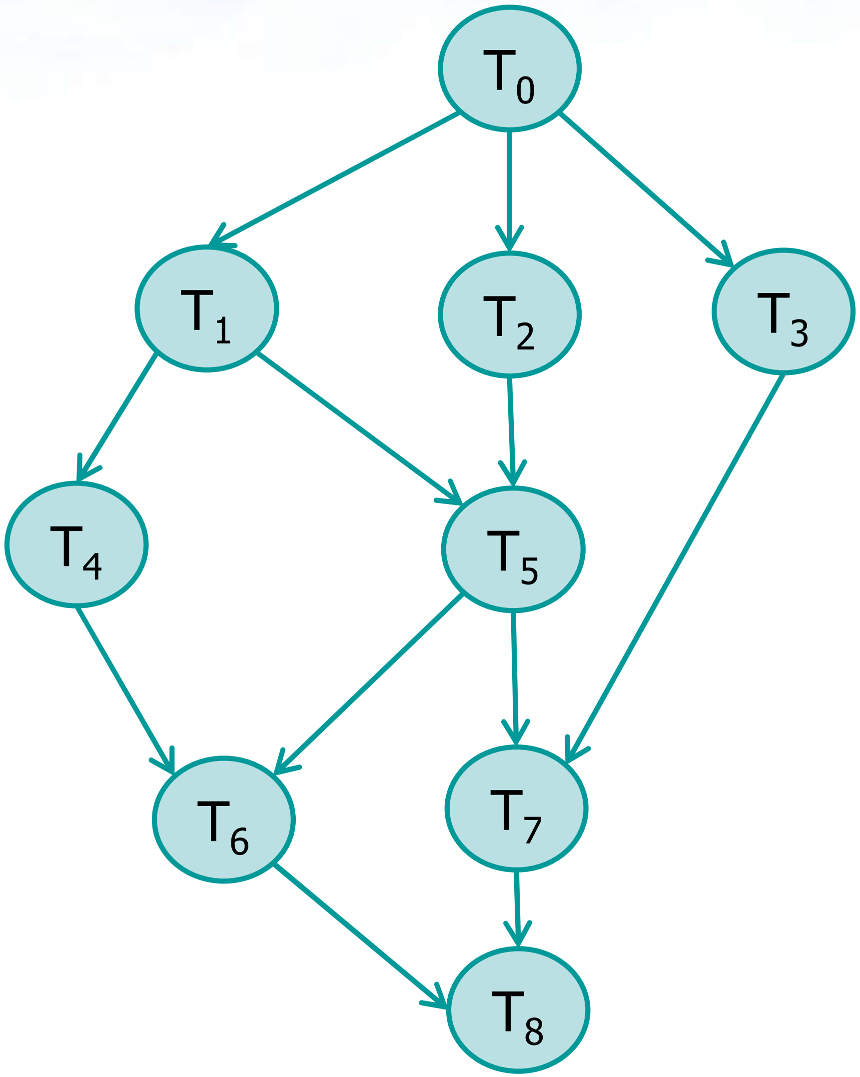
T_3
```
while (1) {
  wait (S3);
  T3 code
  signal (S4);
}
```

T_2 and T_3 cannot use the same semaphore

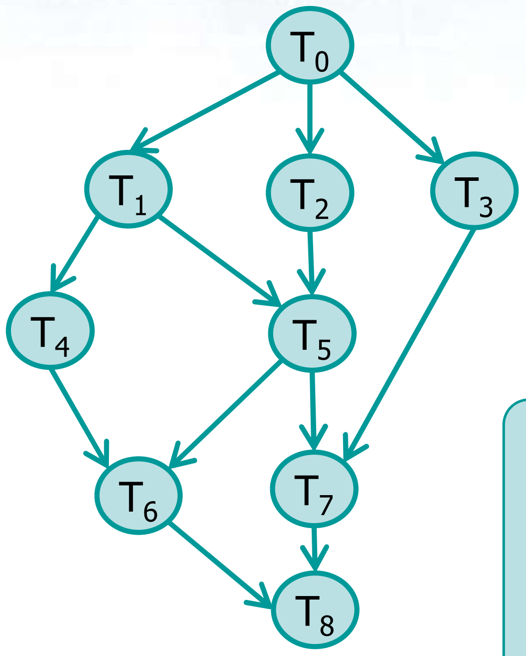
T_4
```
while (1) {
  wait (S4);
  wait (S4);
  T4 code
  signal (S1);
}
```

Exercise

- Implement this precedence graph using semaphores
  - Ts/Ps are **not cyclic**



Solution



T_0
```
T0 code
signal(S1);
signal(S2);
signal(S3);
```

T_1
```
wait(S1);
T1 code
signal(S4);
signal(S5);
```

T_2
```
wait(S2);
T2 code
signal(S5);
```

T_3
```
wait(S3);
T3 code
signal(S7);
```

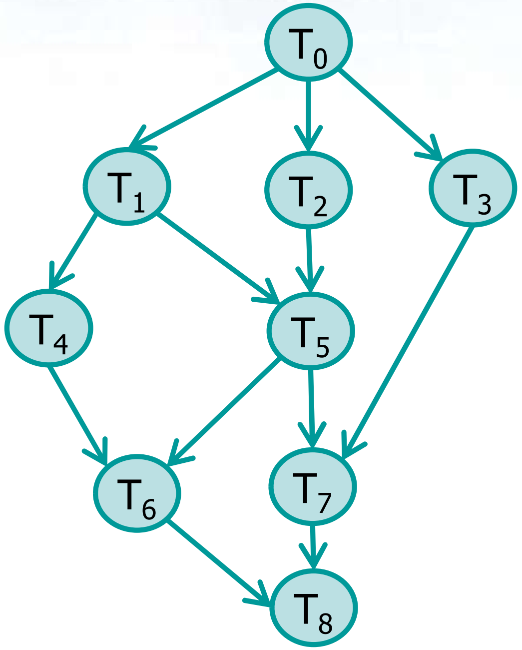
```
init (S1, 0);
init (S2, 0);
init (S3, 0);
...
```

T_4
```
wait(S4);
T4 code
signal(S6);
```

T_5
```
wait(S5);
wait(S5);
T5 code
signal(S6);
signal(S7);
```

...

Solution




```
T6
wait(S6);
wait(S6);
T6 code
signal(S8);
```

```
T7
wait(S7);
wait(S7);
T7 code
signal(S8);
```

```
T8
wait(S8);
wait(S8);
T8 code
```

This solution is correct, but the number of semaphores is **not minimal**