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)
- Solution
  - Establish an **access protocol** that enforces **mutual exclusion** for each CS
    - Before a CS, there should be a **reservation** section
    - After the CS, thre should be a **release** section

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

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
}
```

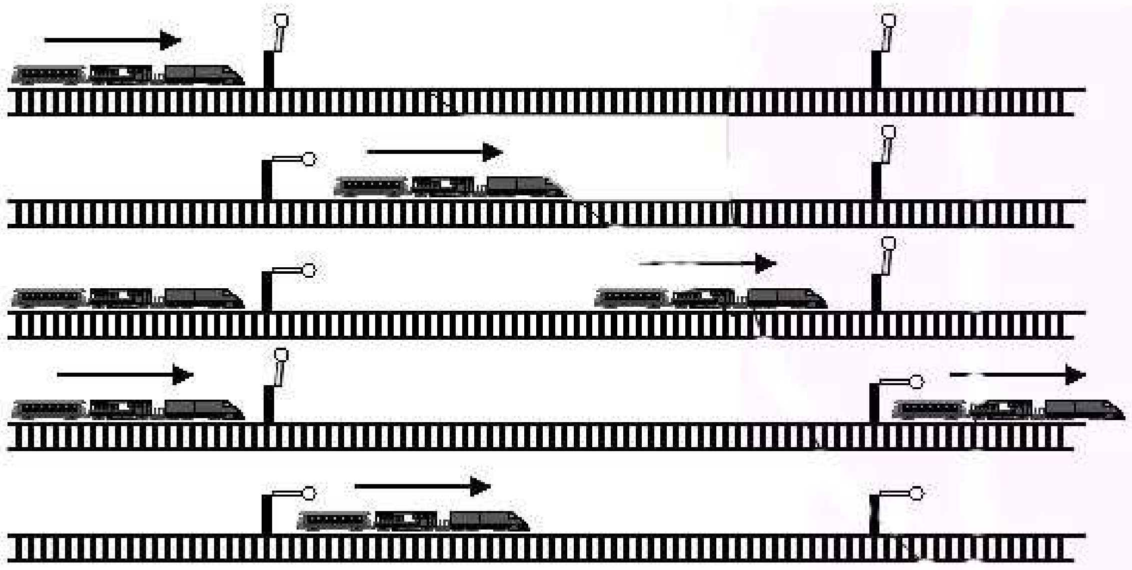
P_j / T_j

```
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
	blocked	wait	-2	T_2, T_3
	blocked	blocked	-2	T_2, T_3
signal	blocked	blocked	-2	T_2, T_3
	CS_2	blocked	-1	T_3
	signal	blocked	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

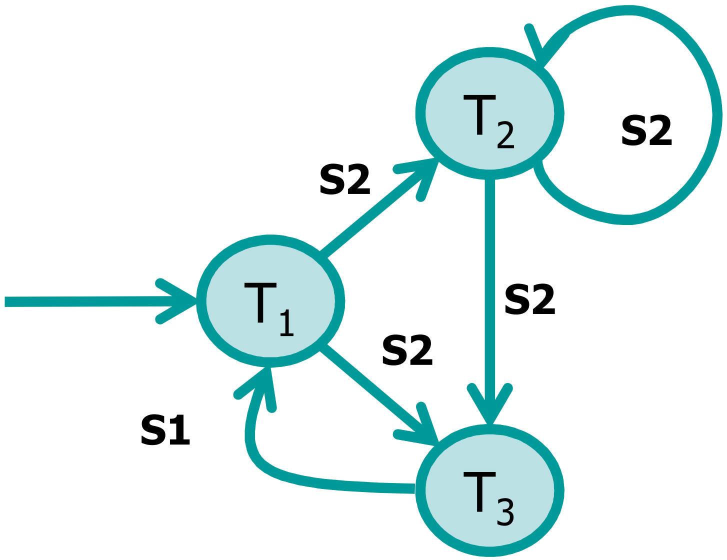
T_1
```
...
while (1) {
   wait (S1);
   T1 code
   signal (S2);
}
...
```

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

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

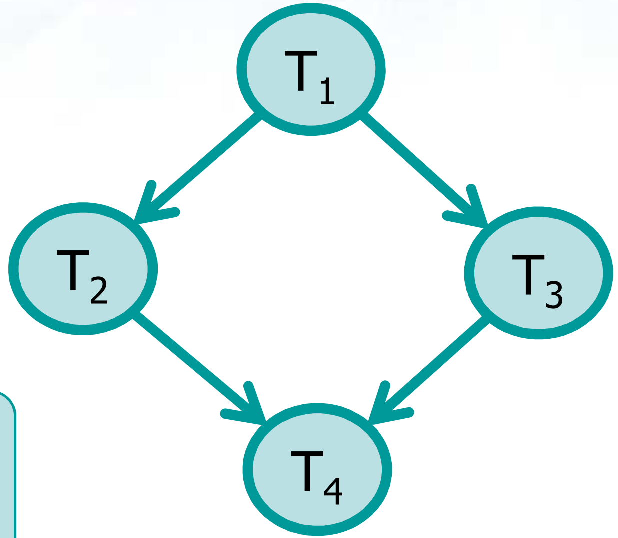
```
init (S1, 1);
init (S2, 0);
```

- 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_2

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

T_1

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

T_3

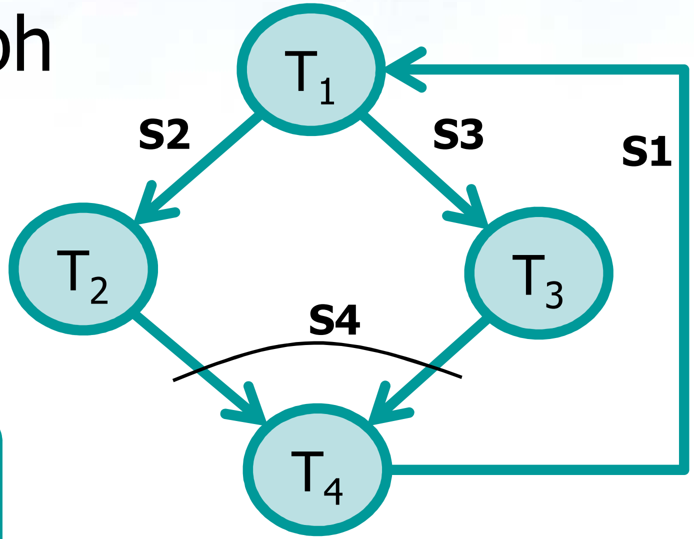
```
...
wait (S1);
T3 code
signal (S2);
...
```

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);
```

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

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

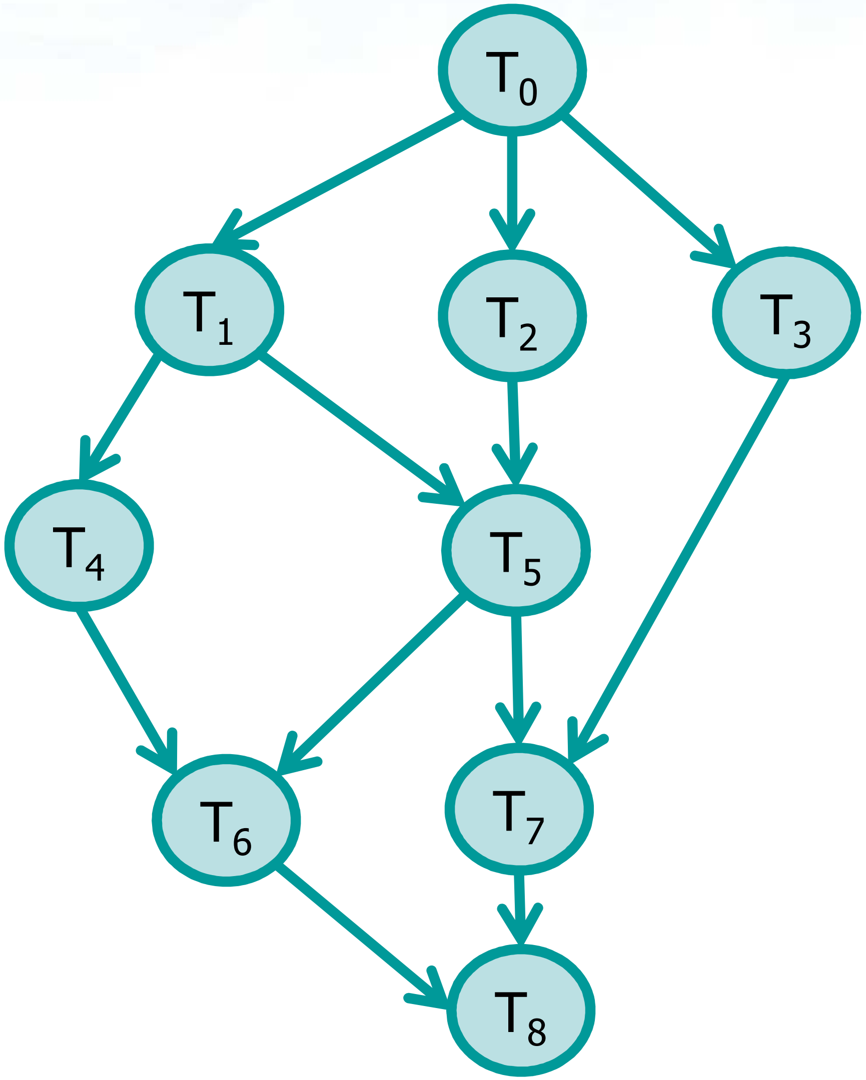
T3
```
while (1) {
  wait (S3);
  T3 code
  signal (S4);
}
```

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

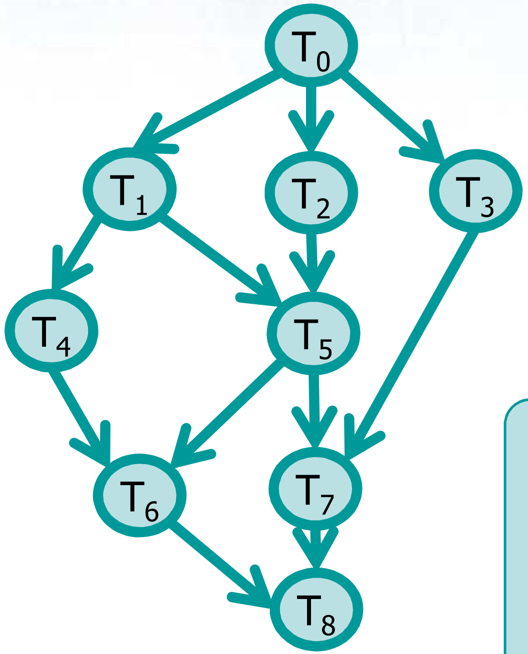
T_2 and T_3 cannot use the same semaphore

Exercise

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



Solution




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

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

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

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

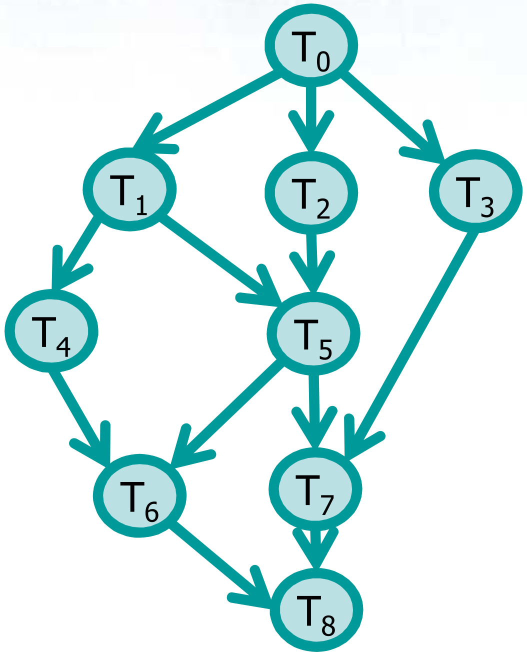
```
init (S1, 0);
init (S2, 0);
init (S3, 0);
...
```

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

```
T5
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**

Real implementations

- There are several semaphores implementations
  - **Semaphores by means of a pipe**
  - **POSIX Pthread**
    - Condition variables
    - **Semaphores**
      - The most important
    - Mutex (for mutual exclusion)
    - ...
  - Linux semaphores

System call: semget, semop, semctl (in sys/sem.h) they are complex to use

- Notice that semaphores are
  - Global share objects (see **sem_init**)
  - They are allocated by a thread, but they are kernel objects

POSIX semaphores

- Kernel and OS independent system calls (POSIX)
  - Header file
    - #include <semaphore.h>
- A semaphore is a type sem_t variable
  - sem_t *sem1, *sem2, ...;
- All semaphore system calls
  - Have name sem_*
  - On error, they return the value -1

System calls:
sem_init
sem_wait
sem_trywait
sem_post
sem_getvalue
sem_destroy

sem_init ()

```
int sem_init (
  sem_t *sem,
  int pshared,
  unsigned int value
);
```

- Initializes the semaphore counter at value **value**
- The value **pshared** identifies the semaphore type
  - If equal to **0**, the semaphore is local to the **threads of the current process**
  - Otherwise, the semaphore can be **shared between different processes** (parent that initializes the semaphore and its children)

Linux does not currently support shared semaphores

sem_wait ()

```
int sem_wait (
  sem_t *sem
);
```

- Standard wait
  - If the semaphore is equal to 0, it blocks the caller until it can decrease the value of the semaphore

sem_trywait ()

```
int sem_trywait (
  sem_t *sem
);
```

- Non-blocking wait
  - If the semaphore counter has a value greater than 0, perform the decrement, and returns 0
  - If the semaphore is equal to 0, returns -1 (instead of blocking the caller as **sem_wait** does)

sem_post ()

```
int sem_post (
  sem_t *sem
);
```

- Standard signal
  - Increments the semaphore counter, or wakes up a blocked thread if present

sem_getvalue ()

```
int sem_getvalue (
   sem_t *sem,
   int *valP
);
```

Better not use this function. From Linux manual: "The value of the semaphore may already have changed by the time sem_getvalue() returns"

- Allows obtaining the value of the semaphore counter
  - The value is assigned to *valP
  - If there are waiting threads
    - 0 is assigned to *valP (Linux)
    - or a negative number whose absolute value is equal to the number of processes waiting (POSIX)

sem_destroy ()

```
int sem_destroy (
  sem_t *sem
);
```

- Destroys the semaphore at the address pointed by sem
  - Destroying a semaphore that other threads are currently blocked on produces undefined behavior (on error, -1 is returned)
  - Using a semaphore that has been destroyed produces undefined results, until the semaphore has been reinitialized

Example

Use of sem_* primitives to synchRonize threads

```
...
#include "semaphore.h"
...
sem_t *sem;
...
sem = (sem_t *) malloc(sizeof(sem_t));
sem_init (sem, 0, 1);
...
... create threads ...
...
sem_wait (sem);
... CS ...
sem_post (sem);
...
sem_destroy (sem);
```