Synchronization

Semaphores

Stefano Quer and Pietro Laface
Dipartimento di Automatica e Informatica
Politecnico di Torino

Introduction

- The previous solutions are not satisfactory because they are either complex or not flexible
- However the hardware solution can be used to implement **system calls** that can be used for solving
  - not only the Mutual Exclusion problem
  - but also any other synchronization problem
  - avoiding the busy form of waiting
- These system calls rely on a data structure called **semaphore**
  - Introduced by Dijkstra in 1965

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;                // Lock variable protects count
                            // and queue management
  int cnt;                  // Counter
  process_t *head;          // Thread list
} semaphore_t;
```

Semaphore primitives

- The kernel offers a set of primitives (i.e., system calls) that allows a thread to be blocked on the semaphore (**wait**) or to wakeup if it was blocked (**signal**)
- Operations on a semaphore are atomic
  - It is impossible for two threads to perform simultaneous operations on the same semaphore

Semaphore primitives

- `init(S, k)`

k is a counter

- Defines and initialize semaphore S counter 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**

"mutex lock"
(mutex ≡ MUTual EXclusion)

Semaphore primitives

k>=0 !!

Cannot be negative because the system calls acting on a semaphore manage the counter so that, if negative, its absolute value is the number of threads waiting on the semaphore queue

```
init (semaphore_t S, int k) {
  alloc S;
  lock(S.lock);
  S.cnt = k;
  S.queue = NULL;
  unlock(S.lock);

}
```

Semaphore primitives

- `wait(S)`
  - Decrement the counter, if the counter value of `S` is negative or zero blocks the calling thread
  - If `S` is negative, the counter absolute value indicates the number of threads blocked on the semaphore queue
  - Originally called `P()` from the Dutch "Probeer te verlagen", i.e., "try to decrease"

Semaphore primitives

- **`signal(S)`**
  - Increases the semaphore **`S`** counter
  - If **`S`** counter is negative or zero some thread was blocked on the semaphore queue, which can be made ready to run
  - Originally called **`V()`**, from the Dutch "verhogen", i.e., "to increment"
  - **Not to be confused** with system call **`signal`** that used to declare a signal handler

Semaphore primitives

```
wait (semaphore_t S) {
  lock(S.lock)
  S.cnt--;
  if (S.cnt<0) {
    insert T to S.queue;
    block T;
     (includes unlock(S.lock))
  }
  else
    unlock(S.lock);
}
```

Waits only if **cnt becomes negative**

```
signal (semaphore_t S) {
  lock(S.lock)
  S.cnt++;
  if (S.cnt<=0) {
    remove T from S.queue;
    wakeup T;
  }
  unlock(S.lock
}
```

If **cnt was negative** before the increment -> some threads are waiting

Semaphore primitives

- **`destroy(S)`**
  - Release semaphore **`S`** memory
    - Often not used in the examples

```
destroy (semaphore_t S){
  lock(S.lock)
  while (S.cnt<=0) {
    free S.queue;
    S.cnt++;
  }
  unlock(S.lock)
}
```

Eliminates all remaining waiting threads

Semaphore primitives

- The semaphore queue
  - Is implemented in kernel space by means of a queue of Thread Control Blocks
  - The kernel scheduler decides the queue management strategy (not necessarily FIFO)

Mutual exclusion with semaphore

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

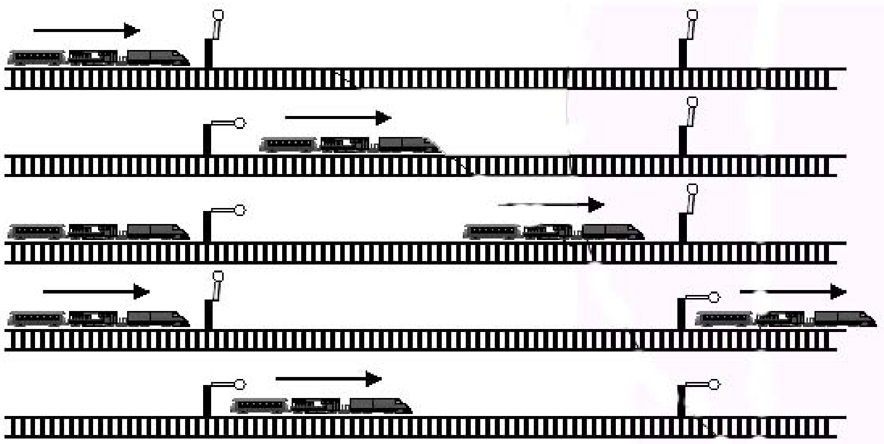
P_i / T_i

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

P_j / T_j

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

```
wait (S) {
   S--;
   if(S<=0)
     blok;
}
signal (S) {
   S++;
   if (S<=0)
     wakeup;
}
```



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	-2	
signal			-2	T_2, T_3
	CS_2		-1	T_3
	signal		0	
		CS_3	0	
		signal	1	

Initialization error

```
init (S, 2);
...
wait (S);
SC di Pi
signal (S);
```

T_1	T_2	T_3	S	queue
			2	
wait			1	
SC	wait		0	
	SC	wait	-1	T_3
		blocked		
signal		blocked	0	
		SC		
	signal		1	
		signal	2	

Threads 1 and 2 in their CSs

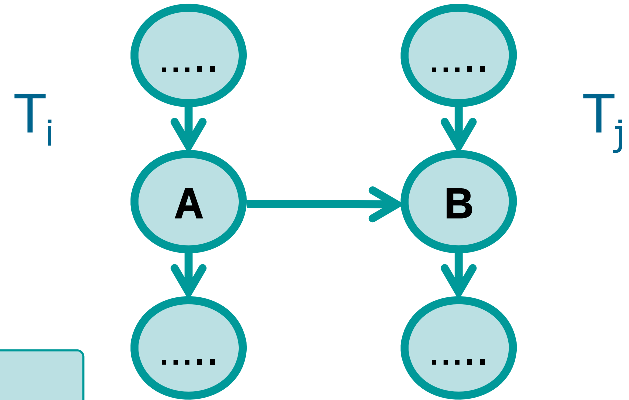
Threads 2 and 3 in their CSs

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 protocol
  - Possibly, more than one semaphore
  - Possibly, additional shared variables

Pure synchronization: Example 1

- Obtain a specific order of execution
  - T_i executes code A before T_j executes code B




```
init (S, 0);
```

T_i

```
……
A;
signal (S);
……
```

T_j

```
……
wait (S);
B;
……
```

Pure synchronization: client-server

- Synchronize two threads so that
  - T_j waits T_i, then T_i waits T_j

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

P_i / T_i

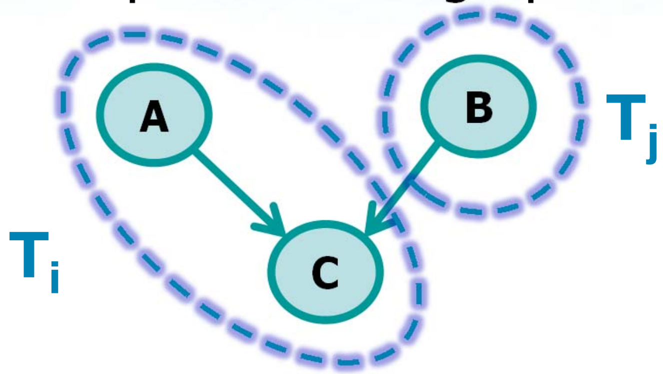
```
while (TRUE) {
  prepare data
  signal (S1);
  wait (S2);
  get processed data
}
```

P_j / T_j

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

Pure synchronization : Precedence graph

- Implement this precedence graph




```
init (S, 0);
```

T_i

```
A
wait (S);
C
```

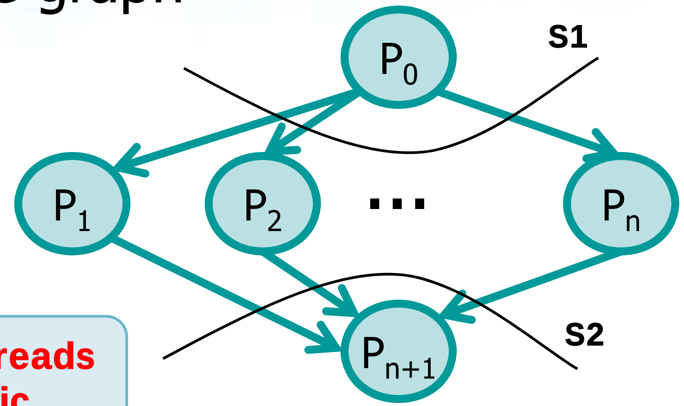
T_j

```
B
signal (S);
```

Pure synchronization : cobegin-coend

- Implement this precedence graph

cobegin-coend
(concurrent begin-end)




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

Note: These threads are not cyclic

T_0

```
...
for(i=1;i<=n;i++)
  signal (S1);
...
```

T_i

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

T_{n+1}

```
...
for(i=1;i<=n;i++)
  wait (S2);
...
```

Errors using semaphores: Example 1

- Just **a single** thread is **incorrect**

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

T_1

```
while (TRUE) {
   ...
   signal (S); !!
   CS1
   wait (S);   !!
   ...
}
```

T_2

```
while (TRUE) {
   ...
   wait (S);
   CS2
   signal (S);
   ...
}
```

T_3

```
while (TRUE) {
   ...
   wait (S);
   CS3
   signal (S);
   ...
}
```

Enters its CS and makes possible that the two other threads enter their CSs

Errors using semaphores: Example 2

- Just **a single** thread is **incorrect**

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

T_1

```
while (TRUE) {
   ...
   wait(S);
   CS1
   wait (S);   !!
   ...
}
```

T_2

```
while (TRUE) {
   ...
   wait (S);
   CS2
   signal (S);
   ...
}
```

T_3

```
while (TRUE) {
   ...
   wait (S);
   CS3
   signal (S);
   ...
}
```

When the second wait is executed all thread are in deadlock

Errors using semaphores: Example 3

- Just **a single** thread is **incorrect**

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

T_1

```
while (TRUE) {
   ...
   signal(S); !!
   CS1
   signal(S);
   ...
}
```

T_2

```
while (TRUE) {
   ...
   wait (S);
   CS2
   signal (S);
   ...
}
```

T_3

```
while (TRUE) {
   ...
   wait (S);
   CS3
   signal (S);
   ...
}
```

When the second signal is executed , if T_1 is fast, all threads can enter their CSs

Errors using semaphores: Example 4

- Just **a single** thread is **incorrect**

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

T_1

```
while (TRUE) {
   ...
   wait(S);
   CS1
   !! no signal(S)
   ...
}
```

T_2

```
while (TRUE) {
   ...
   wait (S);
   CS2
   signal (S);
   ...
}
```

T_3

```
while (TRUE) {
   ...
   !! no wait(S)
   CS3
   signal (S);
   ...
}
```

After T_1 exit its CS, all threads will be in deadlock

If T_3 is fast, all threads can enter their CSs

Errors using semaphores: Example 5

Acquiring two resources

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

T_1

```
while (TRUE) {
  ...
  wait (S);
  ... Use S
  wait (Q);
  ... Use S and Q
  signal (Q);
  signal (S);
  ...
}
```

Access to pen-drive, then to DVD

T_2

```
while (TRUE) {
  ...
  wait (Q);
  ... Use Q
  wait (S);
  ... Use Q and S
  signal (S);
  signal (Q);
  ...
}
```

Access to DVD, then to pen-drive

Exercise

- Given the code of these three threads
  - Which is the possible execution order?

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

T1

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

T2

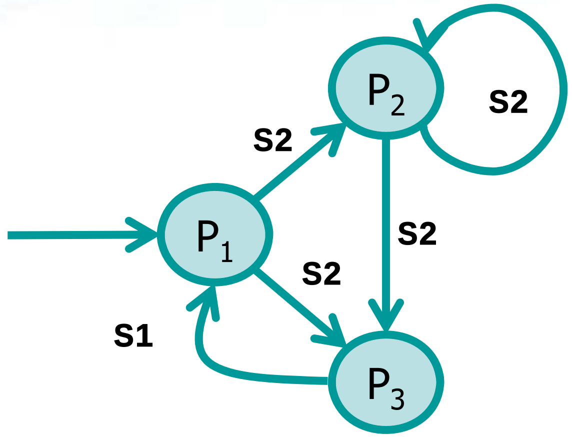
```
...
while (1) {
  wait (S2);
  T2 code
  signal (S2);
}
...
```

T3

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

Solution

- It is a peculiar synchronization example !!




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

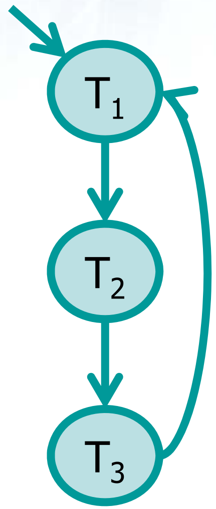
T1
```
...
while (1) {
  wait (S1);
  T1 code
  signal (S2);
}
...
```

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

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

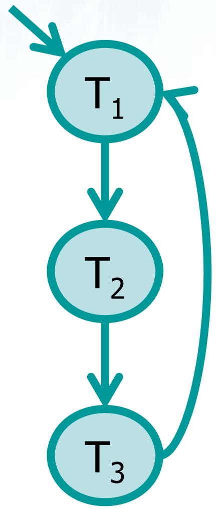
Exercise

- Implement this precedence graph using semaphores



Solution

- Implement this precedence graph using semaphores




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

T_1

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

T_2

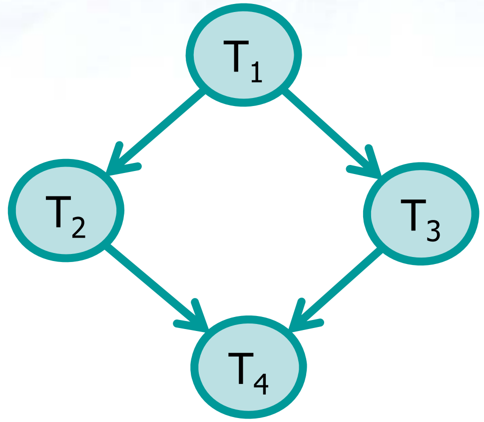
```
...
while (1) {
  wait (S2);
  T2 code
  signal (S3);
}
...
```

T_3

```
...
while (1) {
  wait (S3);
  T3 code
  signal (S1);
}
...
```

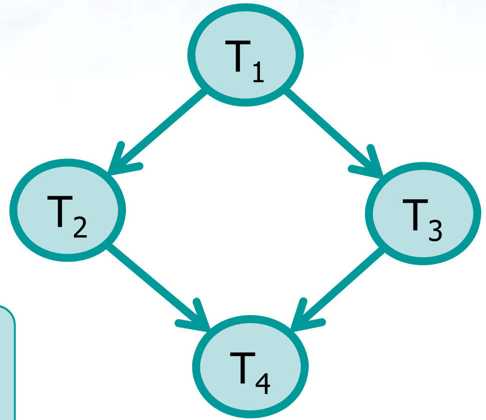
Exercise

- Implement this precedence graph using semaphores



Solution

- Implement this precedence graph using semaphores




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

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

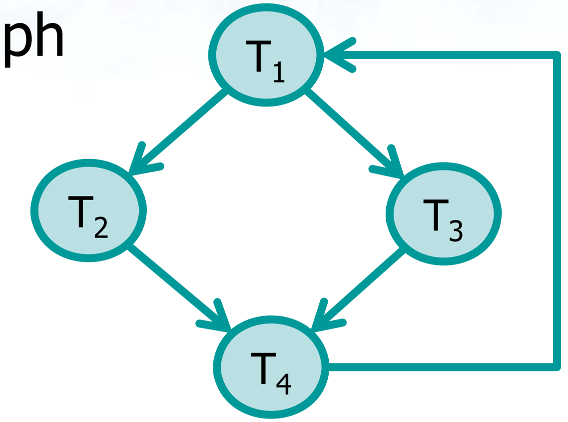
T_2
```
...
wait (S1);
T2 code
signal (S2);
...
```

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

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

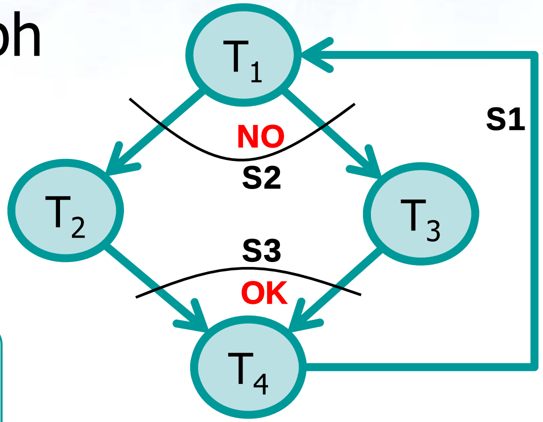
Exercise

- Implement this precedence graph using semaphores
  - **All threads are cyclic**



Erroneous solution

- Implement this precedence graph using semaphores
  - **All threads are cyclic**




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

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

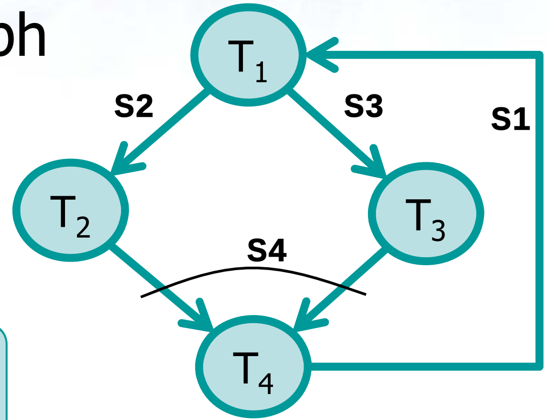
T_2
```
while (1) {
  wait (S2);
  T2 code
  signal (S3);
}
```

T_3
```
while (1) {
  wait (S2);
  T3 code
  signal (S3);
}
```

T_4
```
while (1) {
  wait (S3);
  wait (S3);
  T4 code
  signal (S1);
}
```

Solution

- Implement this precedence graph using semaphores
  - **All threads 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);
}
```

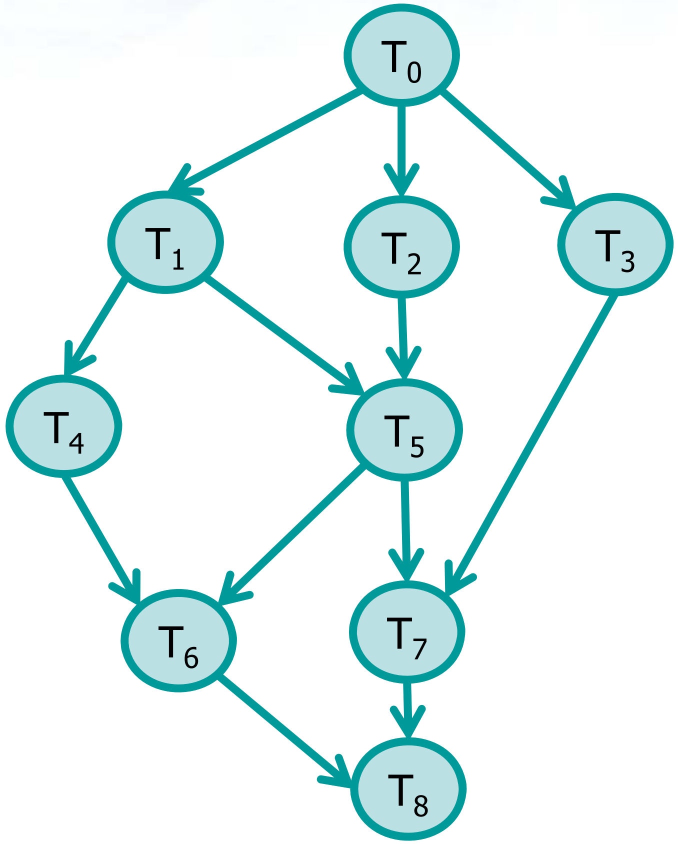
T_2
```
while (1) {
   wait (S2);
   T2 code
   signal (S4);
}
```

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

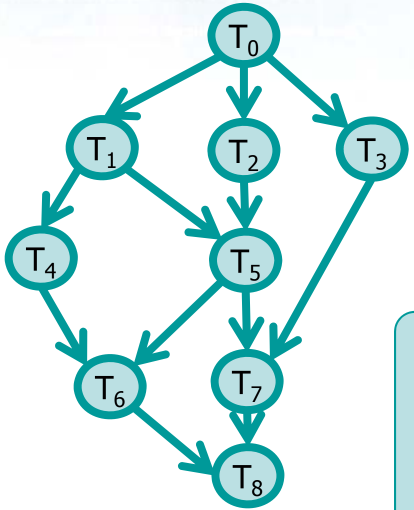
T_4
```
while (1) {
   wait (S4);
   wait (S4);
   T4 code
   signal (S1);
}
```

Exercise

- Implement this precedence graph using semaphores
  - Threads 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);
```

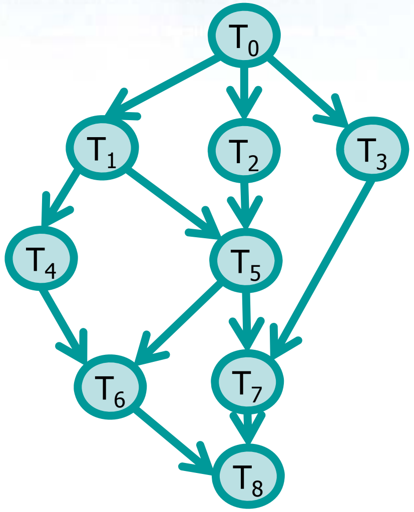
```
for(i=1;i<=7;i++)
  init (Si, 0);
```

```
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);
```

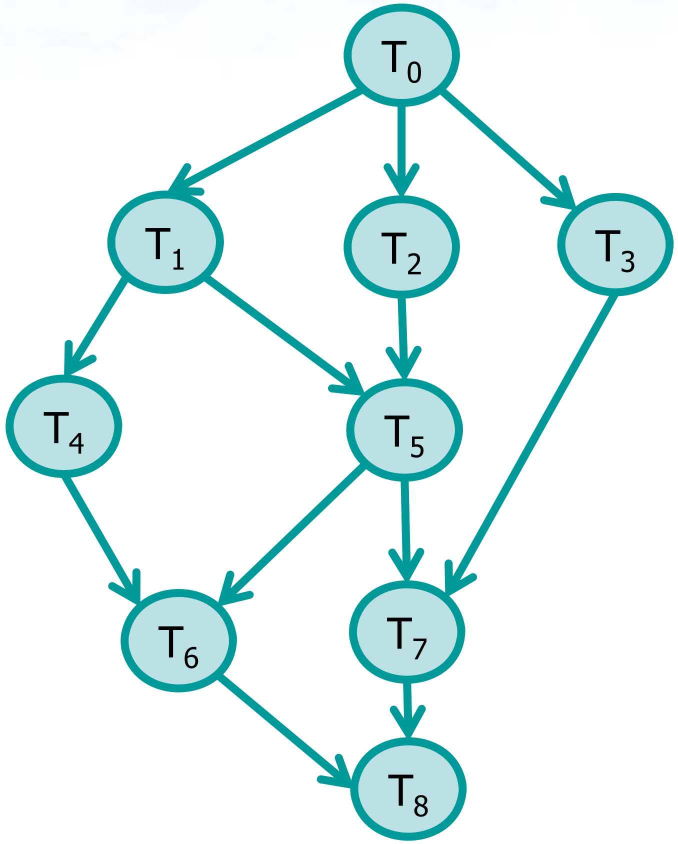
```
T7
wait(S7);
wait(S7);
T7 code
signal(S8);
```

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

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

Exercise

- Implement this precedence graph using semaphores
  - Version A: Threads are **not cyclic, but** use the **minimum number of semaphores**
  - Version B: Threads are **cyclic**



Semaphore implementation

- **Several synchronization structures**
  - **POSIX** Pthread
    - **Mutex** (Mutual exclusion)
    - **Semaphore**
    - **Condition Variable**

System calls:
`pthread_cond_init`
`pthread_cond_wait`
`pthread_cond_signal`
`pthread_cond_broadcast`
`pthread_cond_destroy`

- Please notice that
  - These are share objects
  - They are allocated by a thread, but they are kernel objects

POSIX semaphores

- Kernel 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_xxxx**
  - On error returns -1

System calls:
`sem_init`
`sem_wait`
`sem_try`
`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 `pshared` value identifies the type of semaphore
  - If equal to `0`, the semaphore is local to the **threads of current process**
  - Otherwise, the semaphore can be **shared between different processes** (parent that initializes the semaphore and its children)

sem_wait ()

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

- Standard wait
  - If the semaphore counter is <=0 the calling thread is blocked

sem_post ()

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

- Standard signal
  - Increments the semaphore counter
  - Wakeup a blocking thread is the counter is <= 0

sem_getvalue ()

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

- 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 to 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

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

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 counter is ≤ 0, returns -1 (instead of blocking the caller as `sem_wait` does)
  - `EAGAIN` error if the counter is ≤ `0`

sem_trywait ()

```
...
#include "semaphore.h"
...
sem_t *sem;
...
sem = (sem_t *) malloc(sizeof(sem_t));
sem_init (sem, 0, 1);
sem_getvalue(&sem, &value);     // 1
printf("Initial value of the sem: %d\n", value);
sem_wait(&sem);
sem_getvalue(&sem, &value);     // 0
printf("sem value after wait is %d\n", value);
rc = sem_trywait(&sem);         // 0
if ((rc == -1) && (errno == EAGAIN)) {
  printf("trywait did not decrement the sem");
```

Pthread mutex

- Binary semaphores (mutex)
- A mutex is of type **pthread_mutex_t**
- System calls
  - pthread_mutex_init
  - pthread_mutex_lock
  - pthread_mutex_trylock
  - pthread_mutex_unlock
  - pthread_mutex_destroy

pthread_mutex_init ()

```
int pthread_mutex_init (
  pthread_mutex_t *mutex,
  const pthread_mutexattr_t *attr
);
```

- Initializes the mutex referenced by **`mutex`** with attributes specified by **`attr`** (default=NULL)
- Return value
  - 0 on success
  - Error code otherwise

pthread_mutex_lock ()

```
int pthread_mutex_lock (
  pthread_mutex_t *mutex
);
```

- Blocks the caller if the mutex is locked
- Acquire the mutex lock if the mutex is unlocked

- Return value
  - 0 on success
  - Error code otherwise

pthread_mutex_trylock ()

```
int pthread_mutex_trylock (
  pthread_mutex_t *mutex
);
```

- Similar to pthread_mutex_lock, but returns without blocking the caller if the mutex is locked
- Return value
  - 0 if the lock has been successfully acquired
  - **EBUSY** error if the mutex was already locked

pthread_mutex_unlock ()

```
int pthread_mutex_unlock (
  pthread_mutex_t *mutex
);
```

- Release the mutex lock (typically at the end of a Critical Section)
- Return value
  - 0 on success
  - Error code otherwise

pthread_mutex_destroy ()

```
int pthread_mutex_destroy (
  pthread_mutex_t *mutex
);
```

- Free mutex memory
- The mutex cannot be used any more
- Return value
  - 0 on success
  - Error code otherwise

Exercise

- A file contains a list of integers of indefinite length
- Write a program that, given an integer **`k`** and a file name on the command line, generates **`k`** threads, and then wait their termination
- Each thread
  - Reads the file in concurrency with the other threads, and sums the read values
  - At EOF it displays the number of rows read and the sum of the read values

Exercise

- When all threads complete their job, the main thread displays the total number of rows, and the total sum of the values read by the treads.
- Example

Format of file **file.txt**

```
7
9
2
-4
15
0
3
```

Execution example

```
> pgrm 2 file.txt
Thread 1: Sum=18 #Lines=3
Thread 2: Sum=14 #Lines=4
Total   : Sum=32 #Lines=7
```

Solution

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <semaphore.h>
#include <pthread.h>

#define L 100
typedef struct threadData {
  pthread_t threadId;
  int id;
  FILE *fp;
  int lines;
  int sum;
} threadData ;
static void *readFile (void *);
sem_t sem;
```

Includes, variables and prototypes

Solution

Main
Part 1

```
int main (int argc, char *argv[]) {
  int i, nT, total, lines;
  threadData *td;
  void *retval;
  FILE *fp;

  nT = atoi (argv[1]);
  td = (threadData *) malloc(nT * sizeof (threadData));
  fp = fopen (argv[2], "r");
  if (fp==NULL) {
    fprintf (stderr, "Error Opening File.\n");
    exit (1);
  }
  sem_init (&sem, 0, 1);
```

Solution

Main
Part 2

```
  for (i=0; i<nT; i++) {
    td[i].id = i;
    td[i].fp = fp; // Same fp for all Threads
    td[i].lines = td[i].sum = 0;
    pthread_create (&(td[i].threadId),
      NULL, readFile, (void *) &td[i]);
  }
  total = lines = 0;
  for (i=0; i<nT; i++) {
    pthread_join (td[i].threadId, &retval);
    total += td[i].sum;
    lines += td[i].lines;
  }
  fprintf (stdout, "Total: Sum=%d #Lines=%d\n",
    total, lines);
  sem_destroy (&sem);
  fclose (fp);
  return (1);
}
```

Solution

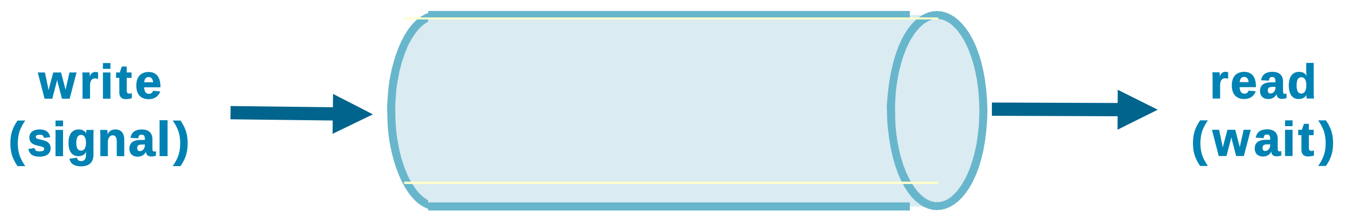
Thread function

```
static void *readFile (void *arg){
  int n, retVal;
  threadData *td;

 td = (threadData *) arg;
 while (1) {
    sem_wait (&sem);
    retVal = fscanf (td->fp, "%d", &n);
    sem_post (&sem);
    if (retVal == EOF)
      break;
    td->lines++;
    td->sum += n;
    sleep (1);  // Delay Threads
  }
  fprintf (stdout, "Thread: %d Sum=%d #Lines=%d\n",
    td->id, td->sum, td->lines);
  pthread_exit ((void *) 1);
}
```

Semaphore by means of a pipe

- Given a pipe
  - The counter of a semaphore is achieved by means of tokens
  - Signal writes a token on the pipe (non-blocking)
  - Wait reads a token from the pipe (blocking)



semaphoreInit (s)

- Semaphore initialization

```
#include <unistd.h>

void semaphoreInit (int *S, int k) {
  char ctr = 'X';
  int i;
  if (pipe (S) == -1) {
    printf ("Error");
    exit (-1);
  }
  for(i=0;i<k,i++)
    if (write(S[1], &ctr, sizeof(char)) != 1) {
      printf ("Error");
      exit (-1);
    }
  return;
}
```

Writes k characters, i.e., initializes the semaphore counter to k

semaphoreSignal (s)

```
#include <unistd.h>

void semaphoreSignal (int *S) {
  char ctr = 'X';
  if (write(S[1], &ctr, sizeof(char)) != 1) {
    printf ("Error");
    exit (-1);
  }
  return;
}
```

Writes a single character, i.e., increments the semaphore counter k

- Writes a character (any) on a pipe
  - Suppose the number of writes (signals) before a read (wait) not exceed the dimension of the pipe

semaphoreWait (s)

```
#include <unistd.h>

void semaphoreWait (int *S) {
  char ctr;
  if (read (S[0], &ctr, sizeof(char)) != 1) {
    printf ("Error");
    exit (-1);
  }
  return;
}
```

If the pipe is empty, read() waits

- Reads a character from a pipe (read is blocking)

Example

```
int main() {
  int S[2];
  pid_t pid;
  semaphoreInit (S, 0);
  pid = fork();
  // Check for correctness
  if (pid == 0) {                    // child
    semaphoreWait (S);
    printf("Wait done.\n");
  } else {                           // parent
    printf("Sleep 3s.\n");
    sleep (3);
    semaphoreSignal (S);
    printf("Signal done.\n");
   }
   return 0;
}
```