Critical Section Problem: Semaphore-based Solutions

23-Oct-2018 (part-2)
CS303
Autumn 2018

Critical Section Problem Solutions

- We have seen two types:
  - Software-based alone (NO SUPPORT from OS was mandated)
  - Hardware-based (provided by HW (delegated via OS)) support
  - The key concept here is some steps are executed in an atomic fashion
  - Semaphores are a special category of variables
    - Which can be initialised to certain value
    - And they can be modified/accessed only by certain methods that are **guaranteed to execute atomically**

Semaphores: Introduction

- Attributed to the dutch scientist Prof. Dijkstra
- Two types:
  - Counting Semaphores
  - Binary Semaphores
- GLIBC includes an implementation of POSIX specification of semaphores in “semaphore.h”
- Counting Semaphores
  - Semaphore S;
    - **Atomic Methods:**
      - wait(semaphore s)   // probe(s) or p(s)
      - signal(semaphore s) // v(s) or increase(s)
- Atomic method: No interleaving or interruptions from the invocation/called point till return from the methods

Semaphore and associated operations

- semaphore s; //initialised to a postive value
- Initialisation value defines the mode of application
- wait(s) signal(s)

```
wait(s){
while (s =< 0) {}
s = s – 1;
}
```

```
signal(s){
s = s + 1;
}
```

- The initialisation value defines the number of resources available for access and being prtected by this semaphore
- Mode of application: either for CS guarding or Synchronisation

Semaphore-based solution for Critical section problem

```
wait(s){
while (s =< 0) {}
s = s – 1;
}
```

```
signal(s){
s = s + 1;
}
```

- Shared variable/s

  semaphore s=1; //initial value

```
wait(s);
  cs
signal(s);
.
```

Semaphore-based solution for Critical section problem

```
wait(s){
while (s =< 0) {}
s = s – 1;
}
```

```
signal(s){
s = s + 1;
}
```

- Shared variable/s

semaphore s=1; //initial value

```
wait(s);
  cs
signal(s);
.
```

ENTRY SECTION

EXIT SECTION

REMAINDER SECTION

Semaphore-based solution: problems

- Recall the essential criteria
  - Mutual Exclusion, Progress and Bounded Waiting
- Progress is there but NOT BW in this solution!
  - Because possibility for alternation is there and a process might be ignored indefinitely!
    - So BW is not satisfied

Process Synchronisation Techniques based on Semaphores

```
Pi:
    .
    .
    Sp;
    .
    .
    .
```

```
Pj:
    .
    .
    Sq;
    .
    .
    .
```

- If P_j shall continue beyond S_q only after P_i Pfinishes S_p, how to implement it?
- Semaphore based solution is ideal here!
  - **P_i** : Synchroniser Process with Synchroniser point at S_p
  - **P_j** : Synchronised Process with Synchronised point at S_q

Process Synchronisation Semaphore Solution

Shared variables:
semaphore s = 0;

```
Pi:
    .
    .
    Sp;
    signal(s);
    .
    .
```

```
Pj:
    .
    wait(s);
    Sq;
    .
    .
    .
```

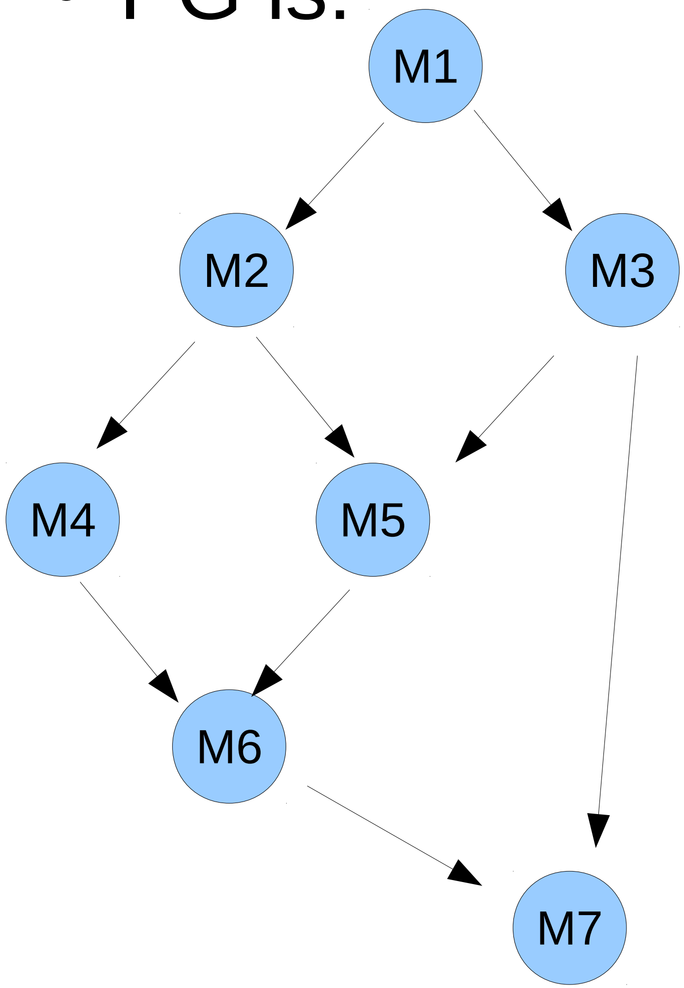
- Init. Semaphore to ZERO
- After syncer point call signal(s)
- Before syncing point call wait(s)

Semaphores: Application in Multi Process Synchronisation

- Recall from initial lectures on the “concurrent specification” via precedence graphs (PG)!
- In essence, PG give module/statement dependency.
- Consider the following graph showing the dependency among different modules:

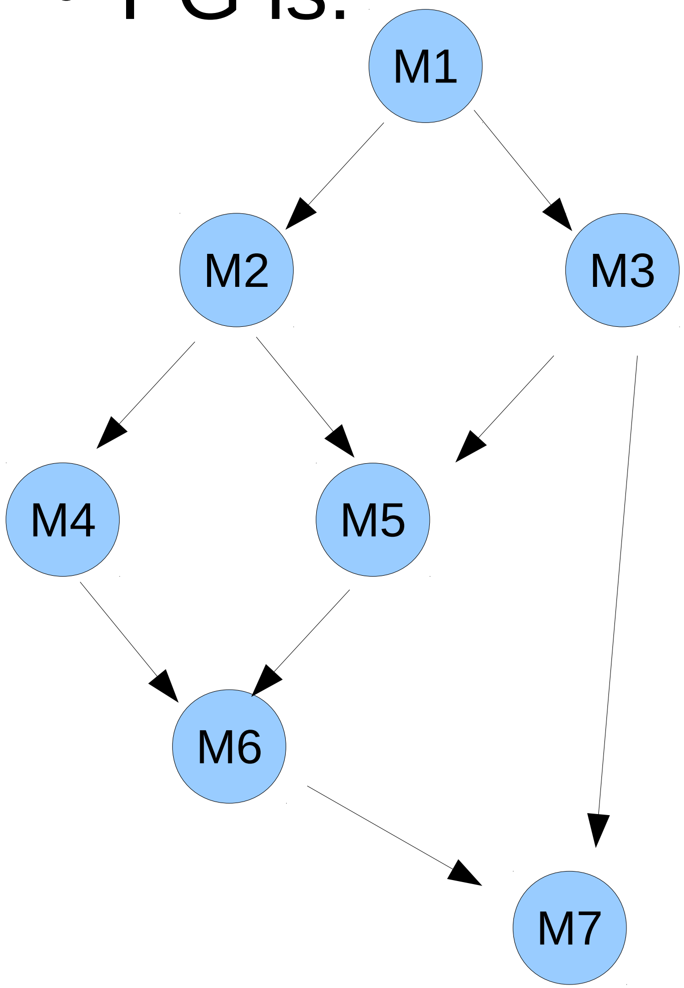
Semaphores: Application in Multi Process Synchronisation

- PG is:



Semaphores: Application in Multi Process Synchronisation

- PG is:

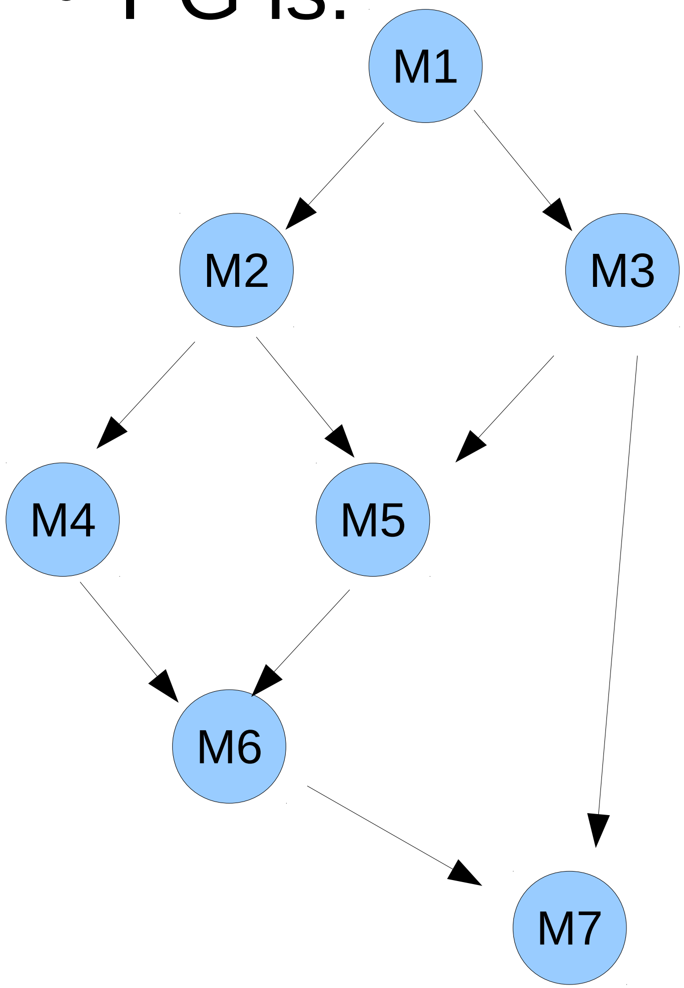


Semaphores:

semaphore s12, s13, s24, s25, s56, s67, s35, s37

Semaphores: Application in Multi Process Synchronisation

- PG is:




```
Semaphores: //init to zero

Begin
    COBEGIN
        M1; signal(s12); signal(s13);
        wait(s12); M2; signal(s24);
                signal(s25);
        wait(s13); M3; signal(s35);
                        signal(s37);
        wait(s24); M4; signal(s46);
        wait(s25); M5; signal(s56);
        wait(s56); M6; signal(s67);
        wait(s37); wait(s67); M7
COEND
End
```

Semaphore-based solution for Critical section problem

```
wait(s){
while (s =< 0) {}
s = s – 1;
}
```

```
signal(s){
s = s + 1;
}
```

- Shared variable/s

  semaphore s=1; //initial value

```
wait(s);
  cs
signal(s);
.
```

Semaphores: for non-waiting type

- Semaphore was a kind of simple numeric/bool variable type: binary vs. coutning
- Whose methods are guaranteed to execute atomically
- Disadvantage: BUSY-WAITING
  - bleeds power and CPU is working
  - Even void skip also takes CPU cycles!
  - SOLUTION: Go for semaphores that let you achieve the same behaviour but with NO-BUSY waiting.

Semaphores: that are of structure type

- Recall that a Semaphore was a kind of simple numeric/bool variable type: binary vs. coutning
- `signal(sem)` and `wait(sem)` methods that execute atomically
  - Recall wait(sem) called before entering the SC
  - signal(sem) while exiting the CS
  - A synchronising task calls signal(sem) : busy-waiitng
  - Synchronised task calls wait(sem): busy-waiting
- To get blocking i.e. non-busy-waiting style synch.
  - Use structure-type semaphore
  - Wait call moves the calling process into BLOCKED state

Semaphores: Of structure type

- Semaphore variant of structure type with richer `wait` and `signal` methods accordingly
- Recall the structure/s from C
  - structure definition for semaphore

```
struct Semaphore {
   int  val;
   int list[50]; // to record the procIDs that are waiting
} sem;
```

- Methods:

```
wait(*s){
s-> value--;
if (s->value < 0) {append(s.list,pid)};
block;}
}
```

```
signal(*s){
s->value++;
if (s->value < 0)
{remove(s.list,pid)}, //process at head
is unblocked
wakeup(pid);}
}
```

Deadlock Problem

- Deadlock: A set of processes is said to be in DL if every process is waiting on an event on another process in that set itself
- To solve DL, we need to identify the criteria for DL occurrence and avoid all such scenarios
- The necessary criteria for DL are:
  - ME
  - Hold and Wait
  - No premption or **resources**
  - Circular waiting