OPERATING SYSTEMS CSCI-SHU215

Lesson 08 - Synchronization

Context

1

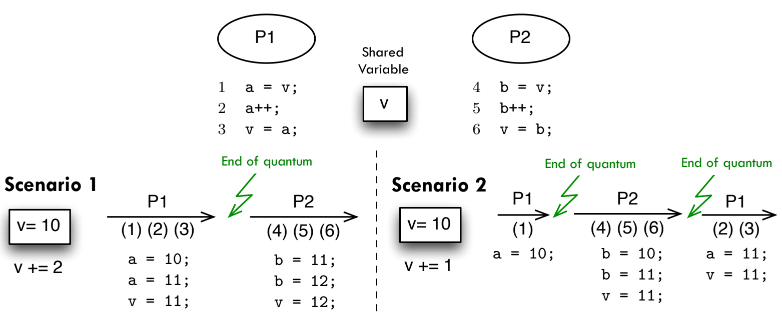
- Context switching
  - Multi-concurrent programming
- Shared Memory
  - Modification of a variable value visible by several processes

Consequence: Nondeterministic executions

ie. 2 runs of the same program can produce ≠ results

Nondeterminism - Example

2



Definitions

3

Critical resource

Resource shared among multiple processes

Consistency => access control (eg. mutual exclusion)

Critical Section (CS)

Portion of code that manipulates one or more critical resources

CS execution must be *indivisible*

Indivisibility

Two CSs on a same critical resource must never be executed in parallel

Definitions

4

Synchronization

- Operation that affects the progress of a set of processes
- Establishment of an order of occurrence between types of instructions
  - *eg. send / receive*
- Enforcement of a condition
  - *eg. rendez-vous, mutual exclusion*

Mutual exclusion

- Control mechanism to access CS
- Special case of synchronization
- Provides exclusive access for ***one*** process at a time
  - *ie. no more than one process in CS at any time*

Definitions

5

Busy waiting

- Repeated execution of a synchronization primitive
- Until condition of synchronization is verified
- => CPU greedy!

Deadlock

- Two (or more) processes wait for each other to carry out competing actions
  - => No escape!

7

Mutual Exclusion

Mutual exclusion

8

Properties

- Safety - No more than one process in CS
- Liveness – Eventually, all access requests to the CS get served

Mutual exclusion


```
EnterCS ();
/ ** CS Instructions ** /
LeaveCS ();
```

`EnterCS ()` and `LeaveCS ()` must be indivisible (atomic)

Naive solution

- `EnterCS ()` and `LeaveCS ()` = System calls
- Hide ITs during execution

Mutual exclusion

Mechanism 1 – Masking of the Clock IT

Disable timesharing when CS gets executed

- `EnterCS` masks Clock IT
- `LeaveCS` unmasks the Clock IT

Wrong answer!

- Risk of monopolizing the processor
- Excludes all processes
- IT Clock mask is internal to the system

Mutual exclusion

Mechanism 2 – Spinlock

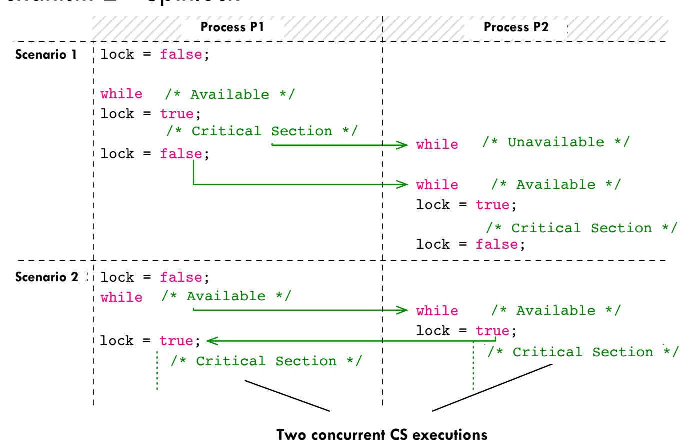
Boolean variable `lock`

- `true` if the resource is locked
- `false` if the resource is available

```
            bool lock = false;
EnterCS()  [while (lock == true);
           [lock = true;
            / ** CS ** /
LeaveCS()  [lock = false;
```

Mutual exclusion

Mechanism 2 - Spinlock



Mutual exclusion

Mechanism 3 – Peterson's algorithm

Mutual exclusion for 2 processes (extensible)

Based on polling and shared variables

`flag`	Array of boolean values shows which processes require the CS
`round`	Integer value indicates the process eligible to access the CS

Code for process i

EnterCS()

```
flag[i] = true
round = j
while (flag[j] == true && round == j)
        /* busy wait */;
```

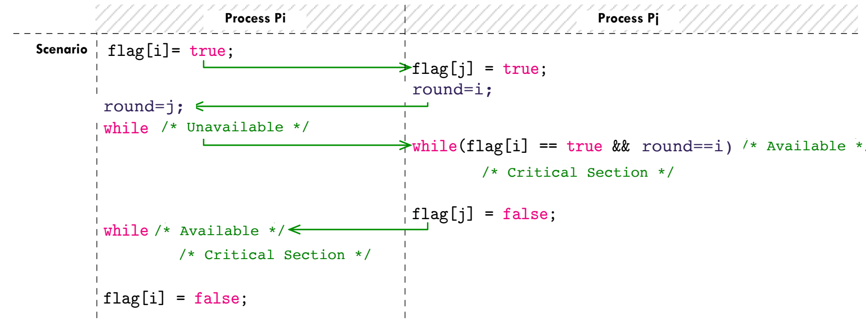
```
/ ** CS ** /
```

LeaveCS()

```
flag[i] = false;
```

Mutual exclusion

Mechanism 3 – Peterson's algorithm



Mutual exclusion

Mechanism 4 - Test-and-Set

Indivisible block of instructions: assignment & test of a variable value

Based on busy waiting

indivisible block (ie. masked interrupts)

```
bool tas(bool v) {
        bool b = v;
        v = true;
        return b;
}
```

```
bool lock = false;
```

EnterCS() busy wait

```
while(tas(lock) == true);
```

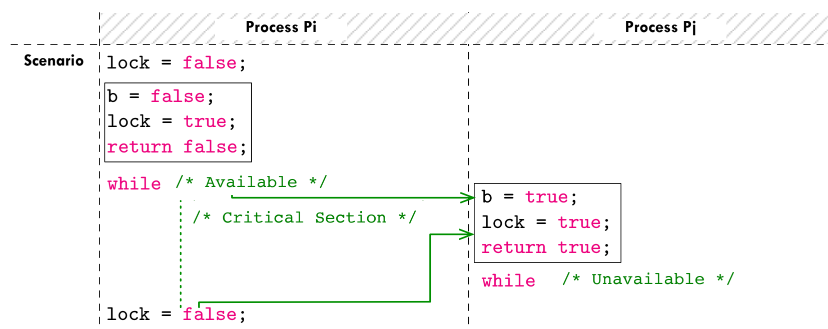
```
    /* Critical Section */
```

LeaveCS()

```
lock = false;
```

Mutual exclusion

Mechanism 4 - Test-and-Set

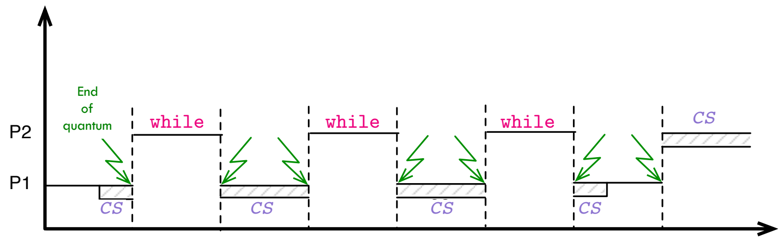


Mutual exclusion

Performance issue raised by busy waiting

busy waiting consumes CPU time

=> solutions without busy waiting are more effective



Semaphores

Semaphores

Principle (Dijkstra)

Synchronization mechanism

- solves mutual exclusion
- structure managed by the system and shared between processes
- no busy waiting

Shared structure

- a queue contains all blocked processes awaiting access
- a counter
  - value initialized to the total number of available accesses (**≥0**)
  - value ≥ 0 — # of accesses available before blocking
  - value < 0 — # of processes stuck in the queue

Semaphores

Mechanism

- Request access (P - *proberen* or *"procure"*)
  - Decrement counter
  - If counter < 0, then block calling process and insert it in the queue
- Release access (V - *verhogen* or *"vacate"*)
  - Increment counter
  - If counter ≤ 0, then extract a process from the queue and unblock it

P and V are indivisible blocks of code (masked ITs)

P *can* be blocking, V is *never* blocking

block, unblock, and queue insertion/extraction are implicit

Semaphores

21

Main drawback

Frankly ***counter intuitive***

Deadlock

- Two processes P and Q are blocked
  - P waits for Q to release its access
  - &
  - Q waits for P to release its access

Famine

- One process is blocked, waiting for a release that will never happen

22

Classic synchronization problems

Classic synchronization problems

23

- Mutual exclusion
- Barrier
- Producer / Consumer
- Reader / Writer
- Swimming pool
- Dining philosophers
- Banker
- Elevator
- Smokers
- ...

Classic problems

24

Mutual exclusion

- Problem

  Allow no more than one process to access the CS at any time

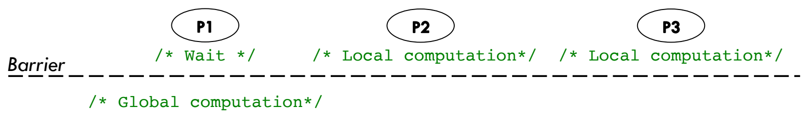
- Semaphore implementation

```
init (MUTEX, 1);
P (MUTEX);
/ ** CS ** /
V (MUTEX);
```

Classic problems

Barrier

- Problem
  Force a process to wait until all others have carried out their action

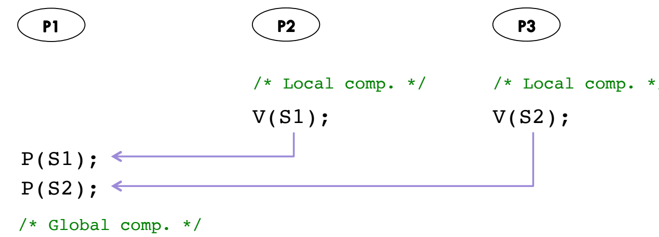


Classic problems

Barrier

- Implementation

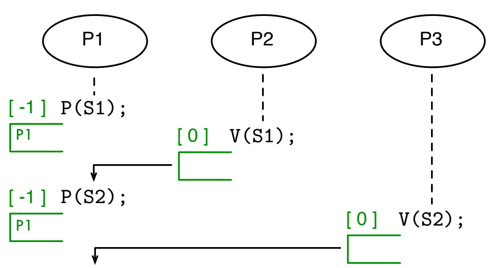
```
init(S1, 0); init (S2, 0);
```



Classic problems

Barrier

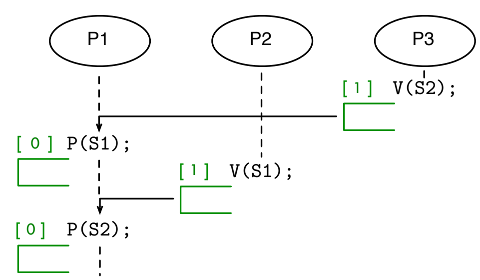
- Example scenario 1



Classic problems

Barrier

- Example scenario 2

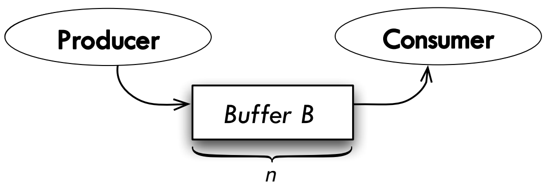


Classic problems

Producer / Consumer (aka. Bounded Buffer)

- Problem

  Exchange of data in a shared buffer



Unix command

`$ ls -l | wc -l`

producer consumer

Classic problems

Producer / Consumer

- Implementation (single value buffer)

```
init (sprod, 1); init (SCons, 0);
```

Producer

```
P(Sprod);
put(m, B);
V(Scons);
```

Consumer

```
P(Scons);
m = get(B);
V(Sprod);
```

Classic problems

Reader / Writer

- Problem

  Share data in a file

  Access constraints

  Either N simultaneous read accesses (no write)

  Or a single write access (no read)

Classic problems

Reader / Writer

- Implementation (*Imperfect: possibility of famine*)

  1 semaphore to enforce file access

  1 shared counter (init 0)

  stores the current number of read accesses

  1 mutex to protect the shared counter

33 POSIX Semaphores

Programming with semaphores

2 types of POSIX semaphores:

- Named semaphores
  - Scope: all processes of the system
  - Basic primitives: **sem_open**, **sem_close**, **sem_unlink**, sem_post, sem_wait
- Unnamed semaphores (*memory-based*)
  - Scope: process with filiation, only threads in linux
  - ***Requires a memory space that is shared between processes / threads***
  - Basic primitives: **sem_init**, **sem_destroy**, sem_post, sem_wait

Included in the `pthread` library

```
$ gcc -Wall -o myprog monprog.c -lpthread
```

Named semaphore creation


```
#include <semaphore.h>
sem_t* sem_open(const char *name, int oflag, mode_t mode, int value);
```

Creates or opens the semaphore called *name*

- *oflag*
  - **O_RDONLY, O_WRONLY , O_RDWR** : read/write permissions
  - **O_CREAT** : create semaphore if it does not exist
  - **O_EXCL** : error if O_CREAT & semaphore already exists
- *mode* read/write/execute permissions (O_CREAT)
- *value* initial value of the counter (O_CREAT)

Returns a pointer to the semaphore, NULL on error

Ex: Creation of a semaphore initialized with value 10

```
sem_t * s;
s = sem_open ( "mysem", O_CREAT | O_RDWR, 0600, 10);
```

Semaphore operations

"P"

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

There is a "P" non-blocking ...

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

"V"

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

Return -1 on error, 0 otherwise

Closing / Destruction

Named semaphore

1) Close the semaphore

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

2) Destroy the semaphore

```
int sem_unlink(const char *name);
```

Sample code with a named semaphore


```
...
int main () {
        * sem_t smutex;
    / * Creation of a semaphore mutex initialized to 1 * /
    if ((smutex = sem_open ( "mysem" O_CREAT | O_EXCL | O_RDWR, 0666, 1)) == SEM_FAILED) {
        if (errno != EEXIST) {
                perror ( "sem_open"); exit (1);
        }
         / * Semaphore already created, open without O_CREAT * /
        smutex = sem_open("mysem", O_RDWR);
      }

      / * P * on smutex /
    sem_wait (smutex);

    / * V * smutex /
    sem_post (smutex);

    / * Close the semaphore * /
     sem_close (smutex);

    / * Destroy the semaphore * /
    sem_unlink ("mysem");
    return 0;
}
```