Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Why semaphores?

- In concurrent programming (many processes running simultaneously), the output depends of the input **and on the scheduling decisions**
- synchronization primitives are used to constrain the possible schedules
- *semaphores* are synchronization primitives

How does a semaphore work?

- For any semaphore s, the kernel records a *value* denoted by $v(s)$ **always** ≥ 0
- The value $v(s)$ of a semaphore represents the number of available accesses to the resource protected by s
- Any process can perform the following actions on a semaphore s:

1 1. Initialize the value $v(s)$ with some integer a (number of allowed concurrent accesses to the resource)
    - ⋆ $v(s) \leftarrow a$

2 2. Use, if available, the shared resource protected by the semaphore s
    - ⋆ if $v(s)$ equals 0, block the process until $v(s) > 0$
    - 3 ⋆ decrement $v(s)$ and use the resource
3. Release a resource being used
    - ⋆ increment $v(s)$ (it is never blocking)
4. Wait until $v(s)$ equals zero. A process waiting until $v(s)$ is zero can be used to
    - ⋆ "refill" the resource
    - ⋆ have many processes waiting for the same "green light"

Creating/accessing an System V semaphore

- System V implements an **array of semaphores**: each operation onto an array of semaphores is **atomic**
  - an array is useful for code fragments that need more than one resource
- The system call

```
int semget(key_t key, int nsems, int semflag);
```

dd

returns the identifier of an array of `nsems` semaphores associated to `key`

  - `semflag` is a list of ORed ("|") options including:
    - ⋆ read/write permissions (least significant 9 bits)
    - ⋆ `IPC_CREAT`:
      (1) create a new semaphore associated to the `key`, if it doesn't exist
      (2) return the existing semaphore associated to the `key`, if it exists
    - ⋆ `IPC_EXCL` (used only with `IPC_CREAT`): the call fails (with `errno=EEXIST`) if the semaphore exists
- When created semaphores are not initialized to any value: they must be explicitly initialized by `semctl(...)` (see later)
- Semaphores are persistent object: they will survive to the process death, they must be erased expicitly

Operation on a single semaphore

- Access/realease of a semaphore are called *semaphore operations*
- An operation on **a single semaphore** (in an array) is described by a dedicated data structure sembuf

```
struct sembuf {
  unsigned short sem_num;     /* sem number */
  short          sem_op;   /* sem operation */
  short          sem_flg;          /* flags */
}
```

- An **array of operations** over the semaphore s_id are performed by

```
int semop(int s_id, struct sembuf * ops, size_t nops);
```

  - ops, array of the operations
  - nops, number of the operations in ops ($\leq$ size of semaphores' array)
  - **Notice**: the nops operations are made all together atomically
  - The call blocks if **any** of the operations cannot be made

Semaphore operations to access/release a resource

```
struct sembuf {
  unsigned short sem_num;      /* sem number */
  short          sem_op;    /* sem operation */
  short          sem_flg;          /* flags */
}
```

- To access a resource protected by semaphore the value of the semaphore must be **decremented** by setting `my_op.sem_op = -<num-res>;`
  - Important: the process **blocks** if `<num-res>` resources are not available
- To release a resource protected by semaphore the value of the semaphore must be **incremented** by setting `my_op.sem_op = <num-res>;`
  - Important: the process **never blocks** when increasing the resources
- `sem_num`, indicates the index of the semaphore in the array

Basic usage: protecting a critical section

- Example: more processes need to write to the same shared memory area
- To avoid that inconsistent data, **only one process** is allowed in **the critical section** of code modifying the data

```
struct sembuf my_op;
int sem_id;
...
sem_id = semget(IPC_PRIVATE /*key*/, 1 /*nsems*/, 0600 /*flags*/);
/* Initialize the semaphore allowing one access only */
semctl(sem_id, 0, SETVAL , 1);   /* later details of semctl call */
/* sharing sem_id with all processes accessing the resource */
...
/* now trying to access critical section */
my_op.sem_num = 0;  /* only one semaphore in array of semaphores */
my_op.sem_flg = 0;                         /* no flag: default behavior */
my_op.sem_op = -1;                            /* accessing the resource */
semop(sem_id , &my_op, 1);   /* blocking if others hold resource */
/* NOW IN CRITICAL SECTION */
...
my_op.sem_op = 1;                             /* releasing the resource */
semop(sem_id , &my_op, 1);                       /* may un-block others */
```

Sem. op. to wait until semaphore is zero

```
struct sembuf {
  unsigned short sem_num;      /* sem number */
  short          sem_op;    /* sem operation */
  short          sem_flg;          /* flags */
}
```

- If processes $A_1, A_2, \ldots, A_n$ must wait that another process B reaches some given point, then
  1. a semaphore is initialized with value 1
  2. all processes $A_1, A_2, \ldots, A_n$ "waits for zero" with a semaphore operation
     `my_op.sem_op = 0;`
  3. process B decrements the same semaphore by one by
     `my_op.sem_op = -1;`
     - the value of the semaphore becomes zero and the processes $A_1, A_2, \ldots, A_n$ will be unblocked

Don't wait forever

- If a resource protected by a semaphore is unavailable, a process may:
  1. wait until the resource is available again (as seen before), or
  2. decide to do something else
- The flag IPC_NOWAIT may be set in a semaphore operation

```
struct sembuf sop;
sop.sem_flg = IPC_NOWAIT;
semop(.., &sop, 1);/*dont wait*/
```

- When executing the semop()
  - if the resource is available, get it as usual
  - if unavailable don't wait, return −1, and errno set to EAGAIN

timeout

- If only waiting for some time is desired

```
#include <time.h>
struct timespec {
    time_t tv_sec; /* seconds */
    long tv_nsec; /* nanoseconds */ }
semtimedop(/*same as semop*/, struct timespec * timeout);
```

Controlling (and initializing) a semaphore

- The system call `semctl()` enables several actions to be performed on semaphores

```
int semctl(int s_id,int i,int cmd);
int semctl(int s_id,int i,int cmd, /* arg */);
```

  - `s_id`, is the ID of the semaphore set
  - `i`, is the index of the semaphore in the set
  - `cmd`, describes the action to be taken over the semaphore
  - the optional fourth argument depends on the type of command

- To set (initialize) or get the value of the `i`-th semaphore in a set

```
int semctl(int s_id,int i,SETVAL,int val);
int semctl(int s_id,int i,GETVAL);
```

  - if `GETVAL`, the value of the `i`-th semaphore is returned;

Semaphore: getting information, removing

- To know how many processes are blocked

```
int semctl(int s_id,int i,GETNCNT);
```

returns the number of processes waiting for the `i`-th semaphore to increase

- To know the process who last accessed a resource

```
int semctl(int s_id,int i,GETPID);
```

returns the PID of the last processes who executed a `semop(s_id,...)` operation on the `i`-th semaphore

- To deallocate the semaphore `s_id`

```
int semctl(int s_id,/*ignored*/,IPC_RMID);
```

  - when a process is blocked on a `semop(id,...)` and the sempahore is removed by `semctl(id,...,IPC_RMID)` the process is unblocked with return value −1 and `errno` is set to `EIDRM`

Semaphores and signals

- When a process is blocked on a `semop(...)` and it receives a non-masked signal
    1. the handler is executed
    2. the `semop()` system call returns `-1` and `errno` is set to `EINTR`
- Even if the flag `SA_RESTART` was set in the signal handler by the `sigaction(...)`, an interrupted `semop(...)` will always fail with `errno` set to `EINTR`

Wrong ways to wait for a semaphore

- Do not loop forever testing the value of a semaphore

```
sop.sem_flg = IPC_NOWAIT;
do {
    semop(.., &sop, 1);
while (errno == EAGAIN);
```

Semaphores: Examples

1. Processi figli che scrivono nella pipe in modo ordinato
   test-pipe-round.c
   which uses a small module for handling semaphores
   - *my_sem_lib.h* (header file)
   - *my_sem_lib.c* (implementation of the functions)
2. Tanti processi che vogliono cucinare condividendo le risorse di una cucina
   test-sem-cook.c

Semaphores: POSIX APIs

- For historical reasons, the course follows the System V API
- However, today the POSIX standard is dominant
- Here is a one slide overview
  - `man sem_overview` for an overview of POSIX semaphores
  - `sem_open(...)`, `sem_init(...)`, and `sem_destroy(...)` to create, initialize and destroy a semaphore
  - `sem_post(...)` to increment by one, `sem_wait(...)` to decrement by one. `sem_timedwait(...)` to wait at most a given timeout
  - no "wait-for-zero" interface
  - not possible to increment by more than one
  - no array of semaphore operations
  - simpler interface (which is good and bad)