

Critical Sections and Semaphores

- A *critical section* is code that contains access to shared resources that can be accessed by multiple processes.
- Critical sections can be managed with semaphores
- This chapter describes POSIX.*SEM*
- semaphores and POSIX.*XSI* semaphores

Protecting CS

- *Mutual Exclusion* – Only one process is in CS at a time
- *Progress* – If no process is in CS, a process that wishes to can get in
- *Bounded Wait* – No process can be postponed indefinitely (starved)

Implementations of: c++;
and c— ;

Non-Atomic Operations

r1 = c;	r2 = c;
r1 = r1 + 1;	r2 = r2 - 1;
c = r1;	c = r2;

c = 6;

Process A

Process B

...; c++; ...

...; c—; ...

At the end of processes A and B we expect c to be incremented and decremented so the final value is 6.

However, if process A is interrupted after completing the instruction r1 = c; and process B executes to completion, c = 5 at the end of B and it is set to 7 after A finishes

test-and-set/swap

- Test-and-Set and Swap are routines implemented in hardware to coordinate lower level critical sections such as the implementing of a semaphore counter
- Review section 8.1 if you are unfamiliar with these operations

Busy-Wait Semaphores

```
wait – while (*s <= 0) noop;
```

```
        (*s)--;
```

```
signal – (*s)++;
```

```
*s = 1;
```

```
Process A
```

```
wait(&s);
```

```
c++;
```

```
signal(&s);
```

```
Process B
```

```
wait(&s);
```

```
c--;
```

```
signal(&s);
```

- Busy-wait implementations waste CPU cycles
- One process can starve the others

Waiting List Semaphores

```
wait – if (sp->value > 0)
        sp->value —;
    else {
        <block the current process and add it to waiting list sp->list>
signal – if (sp->list != NULL)
        <remove process at head of semaphore queue and place it in
            ready queue>
    else
        sp->value++;
```

POSIX.*SEM* Semaphores

- POSIX.*SEM* standard was adopted in 1993
- Since they are new, POSIX.*SEM* semaphores may not be available in all operating systems – even those that claim to be POSIX.*SEM* compliant
- An implementation supports POSIX semaphores if `_POSIX_SEMAPHORES` is defined in `unistd.h`
- It is defined there on the ect-unix machines

POSIX.*SEM* Semaphore Variables

- Semaphore variable is of type `sem_t`
- Atomic operations for initializing, incrementing and decrementing value
- *Unnamed semaphores* – Can be used by a single process or by children of a process that created it
- *Named semaphores* – Can be used by all processes
- Unnamed semaphores are similar in operation to pipes, and named semaphores are similar to named pipes

POSIX.*SEM* Semaphore Declaration

```
#include <semaphore.h>  
sem_t sem;
```

- `sem` is a semaphore variable
- POSIX.*SEM* does not specify underlying type of `sem_t`
- One possibility is for it to act like a file descriptor that points to a local table and the table entries point to entries in a system file table

Semaphore Operations

SYNOPSIS

```
#include <semaphore.h>
```

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

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

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

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

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

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

POSIX.*SEM*

- All semaphore functions return -1 and set `errno` on error
- It is uncertain what semaphore functions return on success, but usually `0`
- `_POSIX_SEMAPHORES` may be defined but system may NOT support POSIX.*SEM* semaphores
- POSIX.*SEM* semaphores are counting semaphores

sem_init

- Initializes semaphore to *value* parameter
- If the value of *pshared* is non-zero, the semaphore can be used between processes (the process that initializes it and by children of that process)
- If the value of *pshared* is zero, the semaphore can only be used by threads of a single process
- Think of *sem* as referring to a semaphore rather than being the semaphore itself

sem_destroy

- Destroys a previously initialized semaphore
- If `sem_destroy` attempts to destroy a semaphore that is being used by another process, it may return `-1` and set `errno` to `EBUSY` – Unfortunately, the specifications do not require that the system detect this

sem_wait and sem_trywait

- *sem_wait* is a standard semaphore wait operation
- If the semaphore value is 0, *sem_wait* blocks until it can successfully decrement value or when interrupted such as by SIGINT
- *sem_trywait* is similar to *sem_wait* except instead of blocking on 0, it returns -1 and sets `errno` to `EAGAIN`

sem_post

- *sem_post* increments the semaphore value and is the classical semaphore signal operation
- *sem_post* must be *async_signal_safe* and may be invoked from a signal handler

sem_getvalue

- Allows the user to examine the value of a named or unnamed semaphore
- Sets the integer referenced by sval to the value of the semaphore
- If there are processes waiting for the semaphore, POSIX.1b allows setting sval to either 0 or a negative number whose absolute value is equal to the number of waiting processes – ambiguity!
- Returns 0 on success and -1 and sets errno on error

```
#include <semaphore.h>
```

```
...
```

```
void main();
```

```
{
```

```
...
```

```
if (sem_init(&my_lock, 1, 1) {
```

```
    perror("could not initialize my_lock semaphore);
```

```
...
```

```
for (i = 1; i < n; ++i)
```

```
    if (childpid = fork()) break;
```

```
...
```

```
if (sem_wait (&my_lock) == - 1) {
```

```
    perror ("semaphore invalid); exit (1); }
```

Critical Section

```
if (sem_post (&my_lock) == - 1) {
```

```
    perror ("semaphore done"); exit(1); }
```

```
...
```

```
}
```

Unnamed Semaphore Example

Named Semaphores

- Named semaphores can synchronize processes that do not have common ancestors
- Have a name, user ID, group ID and permissions just like files do
- `POSIX.SEM` does not require name to appear in file system nor does it specify consequences of having two processes refer to same name
- If name begins with a slash (/), two processes (or threads) open the same semaphore

sem_open

SYNOPSIS

```
#include <semaphore.h>
```

```
sem_t *sem_open(const char *name, int oflag);
```

```
sem_t *sem_open(const char *name, int oflag, mode_t mode,  
                unsigned int value);
```

POSIX.*SEM*

- `sem_open` establishes a connection between a named semaphore and a `sem_t` value
- `sem_open` returns a pointer identifying the semaphore

sem_open oflag parameter

- oflag determines whether sem_open access a previously defined semaphore or creates a new one
- If oflag is 0 the semaphore is previously defined with the same name – If no such name is found sem_open returns -1 and sets errno to ENOENT
- oflag of O_CREAT or O_CREAT|O_EXCL means the semaphore is not previously defined and requires the second form of sem_open that includes permissions and semaphore value
- oflag of O_CREAT|O_EXCL opens a semaphore if one of that name does not exist or returns -1 if one DOES exist and sets errno to EEXIST

sem_close

SYNOPSIS

```
#include < semaphore.h>
```

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

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

POSIX.*SEM*

- A process calls `sem_close` to deallocate system resources allocated to the user of the semaphore
- `sem_close` does not necessarily remove the semaphore, but makes it inaccessible to the process
- `_exit` and `exec` system calls also deallocate process semaphores

sem_unlink

- `sem_unlink` removes a named semaphore from the system
- If there is still a reference to the semaphore, destruction is delayed until the other references are closed by `sem_close`, `_exit` or `exec`
- Calls to `sem_open` with the same name after `sem_unlink` will refer to a different semaphore

Named Semaphore Example – (top)

```
#include <semaphore.h>
#define S_MODE S_IRUSR | S_IWUSR
...
void main();
{
    if ((my_lock = sem_open ("my.dat", O_CREAT|O_EXCL,
        S_MODE, 1) == - 1) && errno == ENOENT) {
        perror("semaphore open failed"); exit(1); }
    ...
    for (i = 1; i < n; ++i)
        if (childpid = fork()) break;
```

Named Semaphore Example – (bottom)

...

```
if (sem_wait (&my_lock) == - 1) {  
    perror ("semaphore invalid); exit (1); }
```

Critical Section

```
if (sem_post (&my_lock) == - 1) {  
    perror ("semaphore done"); exit(1); }
```

...

```
if (sem_close (&my_lock) == - 1) {  
    perror("semaphore close failed"); exit(1); }
```

```
}
```