Exercise 3: Shared Memory and Synchronization

Operating Systems UE 2019W

Michael Platzer

Technische Universität Wien
Computer Engineering
Cyber-physical Systems

2019-12-12

Overview

Inter-process communication

Considered so far...

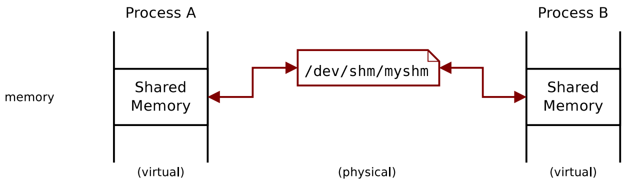
- Implicit Synchronization
  - Blocking read- and write operations
  - Non-related processes via sockets
  - Related processes via unnamed pipes

Today...

- Exchanging data via same memory
  - Memory Mappings
  - POSIX Shared Memory (SHM)
- Explicit synchronization of multiple processes
  - POSIX Semaphore
  - Synchronization tasks

Shared Memory

- Common memory area: Multiple processes (related or unrelated) can access the same region in the physical memory (i.e., share data). This memory region is mapped into the address space of these processes.



- Read and modify by normal memory access operations
- Fast inter process communication[1]

Concurrent access!

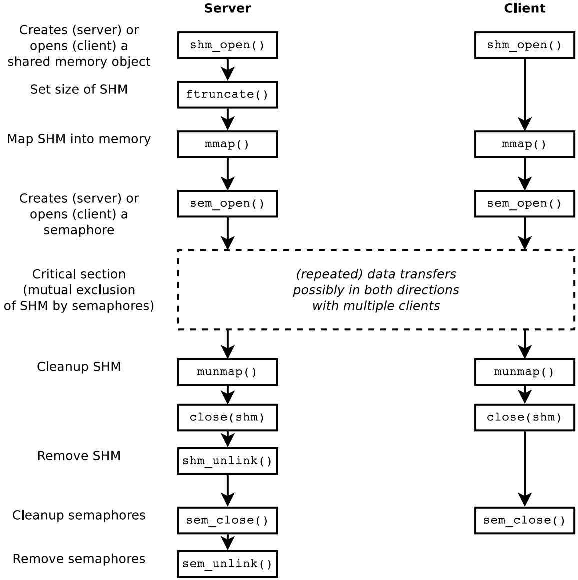
→ Explicit synchronization is necessary

[1] no intervention of the OS kernel/ "zero-copy", see http://www.linuxjournal.com/article/6345

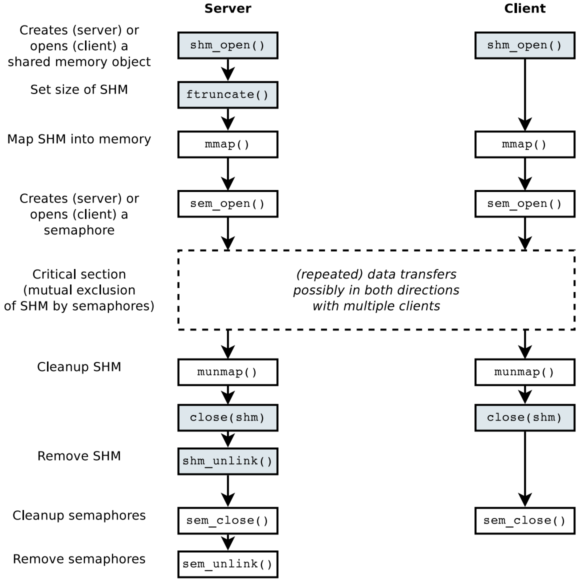
POSIX Shared Memory

- Makes it possible to create shared memory between non-related processes without creating a file
- Shared memory objects identified via names
- Created on file system for volatile memory: tmpfs
- Behaves as a usual file system (e.g. access rights)
- Available as long as system is running
- mmap is used to map it into the virtual memory of a process

Client-Server Example



Client-Server Example



Shared Memory API

Create/Open

- Create and/or open a new/existing object: shm_open(3)

```
#include <sys/mman.h>
#include <fcntl.h>     /* For O_* constants */

int shm_open(const char *name, int oflag,
             mode_t mode);
```

name Name like "/somename"
oflag Bit mask: O_RDONLY or O_RDWR and eventually...
  - O_CREAT: creates an object unless it is created
  - additionally O_EXCL: error if already created

mode Access rights at creation time, otherwise 0

- Return value: file descriptor on success,
  -1 on error (→ errno)
- Linux: Object at `/dev/shm/somename` created

Shared Memory API

Set Size

- The creating process normally sets the size (in bytes) based on the file descriptor: `ftruncate(2)`

```
#include <unistd.h>
#include <sys/types.h>

int ftruncate(int fd, off_t length);
```

- Return value: 0 on success, -1 on error ($\rightarrow$ errno)
- Then the file descriptor can be used to create a common mapping (mmap(2)) and finally it can be closed (close(2))

Shared Memory API

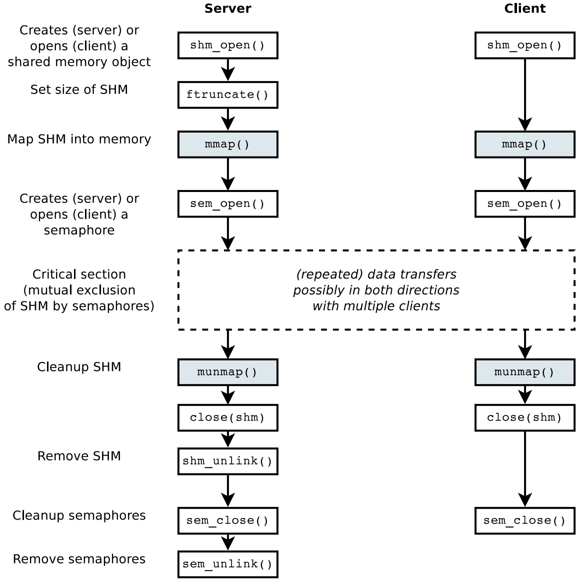
Remove

- Remove a shared memory object name: shm_unlink(3)

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

- Name, which was specified at creation
- Return value: 0 on success, -1 on error ($\rightarrow$ errno)
- Further shm_open() with the same name raises an error (unless a new object is created by specifying `O_CREAT`)
- The memory is released when the last process has closed the file descriptor with close() and released any mappings with munmap()
- Common commands (`ls`, `rm`) can be used to list and remove `/dev/shm/` (e.g. if program crashes)

Client-Server Example



Memory Mapping

Recall: mmap(2)

mmap(2)

= maps a file into the virtual memory of a process

- Multiple processes can access the underlying memory
- Shared memory is based on sharing a resource (a file) "shared file mapping"

Memory Mapping

Create

- Create a mapping: mmap(2)

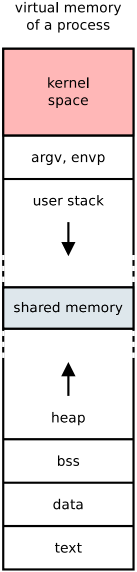
```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot,
           int flags, int fd, off_t offset);
```

addr	Suggestion for starting address, should be NULL
length	Size of the mapping in bytes, often the size of a file (see fstat(2))
prot	Bit mask for memory protection: PROT_NONE (no access allowed), PROT_READ, PROT_WRITE
flags	Bit mask, e.g., MAP_PRIVATE, MAP_SHARED, MAP_ANONYMOUS
fd	The file descriptor to be mapped
offset	Offset in the file (multiple of page size), 0

- Return value: Starting address of the mapping (aligned to page limit), MAP_FAILED on error (errno)

Memory Mapping

Virtual Address Space



- Mappings in different processes are created at different addresses
- Take care by storing pointers!

Memory Mapping

Comments

- The file descriptor (e.g. of a shared memory) can be closed after the creation of the mapping
- In Linux, mappings are listed under `/proc/PID/maps`
- Disadvantages of actual file mappings (not a virtual file) for shared memory: Persistent → costs for disk I/O
- For related processes: shared, anonymous mappings (`MAP_SHARED | MAP_ANONYMOUS`)
  - No underlying file, not even a virtual file
  - Create mapping before fork():
    → child processes can access the mapping at the same address

Memory Mapping

Release

- Releasing a mapping: `munmap()`

```
#include <sys/mman.h>

int munmap(void *addr, size_t length);
```

- Removes whole memory pages from the given space, starting address has to be page-aligned
- Return value: 0 on success, -1 on error ($\rightarrow$ errno)

Example

Define Structure of the shared memory

```c
#include <fcntl.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>

#define SHM_NAME "/myshm"
#define MAX_DATA (50)

struct myshm {
  unsigned int state;
  unsigned int data[MAX_DATA];
};
```

Example

Create and map the shared memory

```
// create and/or open the shared memory object:
int shmfd = shm_open(SHM_NAME, O_RDWR | O_CREAT, 0600);
if (shmfd == -1)
    ... // error

// set the size of the shared memory:
if (ftruncate(shmfd, sizeof(struct myshm)) < 0)
    ... // error

// map shared memory object:
struct myshm *myshm;
myshm = mmap(NULL, sizeof(*myshm), PROT_READ | PROT_WRITE,
             MAP_SHARED, shmfd, 0);

if (myshm == MAP_FAILED)
    ... // error

if (close(shmfd)) == -1)
    ... // error
```

Example

Cleanup

```
// unmap shared memory:
if (munmap(myshm, sizeof(*myshm)) == -1)
    ... // error

// remove shared memory object:
if (shm_unlink(SHM_NAME) == -1)
    ... // error
```

Semaphores

Synchronization

= control access of concurrent processes to a critical section

- Conditional synchronization: In which order is a critical section accessed: A before B? B before A?
- **Mut**ual **ex**clusion: Ensure that only one process is accessing a shared resource ().
  Not necessarily fair/alternating.

Example (1)

Thread A:

```
a1: print ''yes''
```

Thread B:

```
b1: print ''no''
```

- No deterministic sequence of “yes” and “no”. Depends on, e.g., the scheduler.
- Multiple calls might cause different outputs. Are other outputs possible?

Example (2)

Thread A:

```
a1: x = 5
a2: print x
```

Thread B:

```
b1: x = 7
```

- Path to output "5" and in the end $x = 5$?
- Path to output "7" and in the end $x = 7$?
- Path to output "5" and in the end $x = 7$?
- Path to output "7" and in the end $x = 5$?

Example (3)

Thread A:

```
a1: x = x + 1
```

Thread B:

```
b1: x = x + 1
```

- Assumption: x is initialized with 1. What are possible values for x after execution?
- Is x++ atomic?

Semaphores

Functions

Semaphore

= "Shared variable" used for synchronization

- 3 basic operations:
  - S = Init(N)
    create semaphore S with value N
  - P(S), Wait(S), Down(S)
    decrement S and block when S gets negative
  - V(S), Post(S), Signal(S), Up(S)
    increment S and wake up waiting process

Example - Serialization

Thread A:

```
statement a1
```

Thread B:

```
statement b1
```

How to guarantee that $a1 < b1$ (a1 before b1)?

Example - Serialization

Initialization:

```
S = Init(0)
```

Thread A:

```
statement a1
V(S) // post
```

Thread B:

```
P(S) // wait
statement b1
```

Example - Mutex

Thread A:

```
x = x + 1
```

Thread B:

```
x = x + 1
```

How to guarantee that only one thread is entering the critical section?

Example - Mutex

Initialization:

```
mutex = Init(1)
```

Thread A:

```
P(mutex) // wait
x = x + 1
V(mutex) // post
```

Thread B:

```
P(mutex) // wait
x = x + 1
V(mutex) // post
```

$\Rightarrow$ Critical section seems to be atomic

Example - Alternating Execution

Thread A:

```
for(;;) {
  x = x + 1
}
```

Thread B:

```
for(;;) {
  x = x + 1
}
```

How to achieve that A and B are called alternately?

Example - Alternating Execution

Initialization:

```
S1 = Init(1)
S2 = Init(0)
```

Thread A:

```
for(;;) {
  P(S1) // wait
  x = x + 1
  V(S2) // post
}
```

Thread B:

```
for(;;) {
  P(S2) // wait
  x = x + 1
  V(S1) // post
}
```

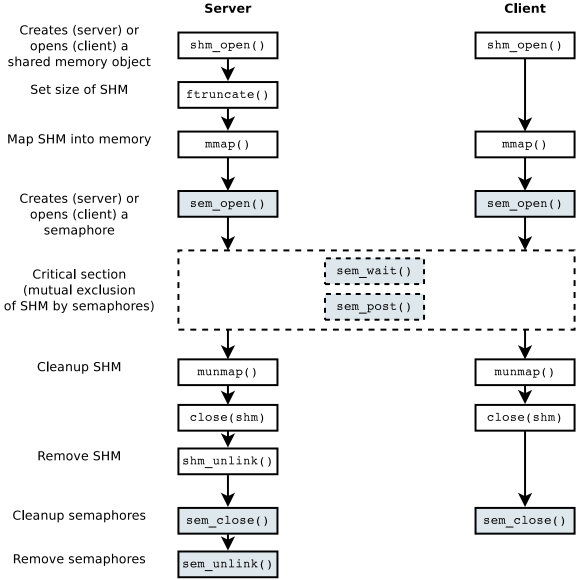
$\Rightarrow$ 2 semaphores are necessary!

How does the synchronization look like for 3 threads that should work alternately? How about N threads?

POSIX Semaphore

- Synchronization of processes
  - Non-related processes: named semaphores
  - (Related processes or threads within a process: unnamed semaphores)
- Similar to POSIX shared memory...
  - Identified by name
  - Created on dedicated file system for volatile memory: tmpfs
  - Lifetime limited to system runtime
- Linked with `-pthread`
- See also `sem_overview(7)`
- Linux: object is created at `/dev/shm/sem.`<u>`somename`</u>

Client-Server Example



Semaphore API

Create/Open

- Create/open a new/existing semaphore: sem_open(3)

```
#include <semaphore.h>
#include <fcntl.h>     /* For O_* constants */

/* create a new named semaphore */
sem_t *sem_open(const char *name, int oflag,
                mode_t mode, unsigned int value);

/* open an existing named semaphore */
sem_t *sem_open(const char *name, int oflag);
```

name	Name of the form "/somename"
oflag	Bit mask: O_CREAT, O_EXCL
mode	Access rights (at creation time only)
value	Initial value (when creating)

- Return value: Semaphore address on success, SEM_FAILED on error (→ errno)

Semaphore API

Close and Remove

- Close a semaphore: sem_close(3)

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

- Remove a semaphore: sem_unlink(3)

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

  Is released after all processes have closed it.

- Return value: 0 on success, -1 on error ($\rightarrow$ errno)

Semaphore API

Wait, P()

- Decrement a semaphore: `sem_wait(3)`

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

- If the value > 0, the method returns immediately
- It blocks the function until the value gets positive otherwise
- Return value: 0 on success, -1 on error ($\rightarrow$ errno) and the value of the semaphore is not changed

Signal Handling

The function `sem_wait()` can be interrupted by a signal (errno == EINTR)!

Semaphore API

Post, V()

- Increment a semaphore: sem_post(3)

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

- If the value of a semaphore gets positive, a blocked process will continue
- If multiple processes are waiting: the order is not defined (= weak semaphore)
- Return value: 0 on success, -1 on error ($\rightarrow$ errno) and the semaphore value is not changed

Example - Alternating Execution

Process A (code without error handling)

```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

#define SEM_1   "/sem_1"
#define SEM_2   "/sem_2"

int main(int argc, char **argv) {
  sem_t *s1 = sem_open(SEM_1, O_CREAT | O_EXCL, 0600, 1);
  sem_t *s2 = sem_open(SEM_2, O_CREAT | O_EXCL, 0600, 0);

  for(int i = 0; i < 3; ++i) {
    sem_wait(s1);
    printf("critical: %s: i = %d\n", argv[0], i);
    sleep(1);
    sem_post(s2);
  }
  sem_close(s1); sem_close(s2);

  return 0;
}
```

Example - Alternating Execution

Process B (code without error handling)

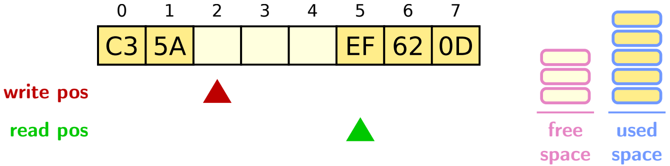
```
#include <stdio.h>
#include <unistd.h>
#include <semaphore.h>
#include <fcntl.h>

#define SEM_1   "/sem_1"
#define SEM_2   "/sem_2"

int main(int argc, char **argv) {
  sem_t *s1 = sem_open(SEM_1, 0);
  sem_t *s2 = sem_open(SEM_2, 0);

  for(int i = 0; i < 3; ++i) {
    sem_wait(s2);
    printf("critical: %s: i = %d\n", argv[0], i);
    sleep(1);
    sem_post(s1);
  }
  sem_close(s1); sem_close(s2);
  sem_unlink(SEM_1); sem_unlink(SEM_2);
  return 0;
}
```

Example - Handling Signals

```c
volatile sig_atomic_t quit = 0;

void handle_signal(int signal) { quit = 1; }

int main(void)
{
    sem_t *sem = sem_open(...);

    struct sigaction sa = { .sa_hander = handle_signal; };
    sigaction(SIGINT, &sa, NULL);

    while (!quit) {
        if (sem_wait(sem) == -1) {
            if (errno == EINTR) // interrupted by signal?
                continue;

            error_exit(); // other error
        }

        ...
    }
}
```

Circular Buffer

= simple FIFO implementation with shared memory and semaphores




```c
int wr_pos = 0;
void write(int val) {
    sem_wait(free_sem);
    buf[wr_pos] = val;
    sem_post(used_sem);
    wr_pos += 1;
    wr_pos %= sizeof(buf);
}
```

```c
int rd_pos = 0;
int read() {
    sem_wait(used_sem);
    int val = buf[rd_pos];
    sem_post(free_sem);
    rd_pos += 1;
    rd_pos %= sizeof(buf);
    return val;
}
```

Circular Buffer (code without error handling)

```c
#define BUF_LEN 8
int *buf; // points to shared memory mapped with mmap(2)
sem_t *free_sem, // tracks free space, initialized to BUF_LEN
      *used_sem; // tracks used space, initialized to 0
```

```c
int write_pos = 0;
void circ_buf_write(int val) {
    sem_wait(free_sem); // writing requires free space
    buf[write_pos] = val;
    sem_post(used_sem); // space is used by written data
    write_pos = (write_pos + 1) % BUF_LEN;
}
```

```c
int read_pos = 0;
int circ_buf_read() {
    sem_wait(used_sem); // reading requires data (used space)
    int val = buf[read_pos];
    sem_post(free_sem); // reading frees up space
    read_pos = (read_pos + 1) % BUF_LEN;
    return val;
}
```

Reading and writing can happen simultaneously!

Resource Management

Who creates a resource?

→ depends on the calling sequence of the processes!

- Undefined
- Fixed sequence (e.g., client-server systems)

Who removes resources?

- No errors during program execution
- On errors
  - Unsynchronized cleanup:
    Process that has errors removes resources
  - Synchronized cleanup:
    Special communication is necessary (expensive)

Resource Allocation

... at undefined calling sequence

- Creates it, unless it exists
- O_CREAT flag without O_EXCL flag (no error if the SHM already exists → SHM will be opened only)
- e.g., shared memory:

```
shmfd = shm_open(SHM_NAME,
                 O_CREAT | O_RDWR, PERM);

if (shmfd == -1) ... /* error */
```

Remove Resources

... at fixed calling sequence

Assumption: Synchronization ok (correctly implemented), i.e., processes execute the critical section in the correct sequence
→ Process responsible (the last one accessing the critical section) is able to remove the resources

- Remove the resources local to the process as usual (e.g., close opened log file)
- Remove the kernel persistent resources (e.g., shared memory, semaphores) on normal process termination but also on errors
- Help: atexit(3)
  - Register a function that gets called on normal process termination
  - Multiple functions: in reversed sequence of the registration
  - Registered functions are not called on `_exit()` (cf. signal handling)

Remove Resources

Example

```
static int shmfd = -1;

void allocate_resources(void) {
  shmfd = shm_open(SHM_NAME, O_CREAT ...);
  ...
}

void free_resources(void) {
  ...
  if (shmfd != -1) {
    if (shm_unlink(SHM_NAME) == -1)
      /* print error message, DON'T CALL EXIT */;
  }
}

void main(void) {
  if (atexit(free_resources) != 0)
    /* error */

  allocate_resources();
  ...
}
```

Summary

- Shared memory is a fast method for IPC
- Explicit synchronization with semaphores
- Synchronization tasks
- Strategies to resource (de-)allocation

Material

- Michael Kerrisk: A Linux and UNIX System Programming Handbook, No Starch Press, 2010.
- Linux implementation of shared memory/tmpfs: `http://www.technovelty.org/linux/shared-memory.html`
- Richard W. Stevens: UNIX Network Programming, Vol. 2: Interprocess Communications