7. Communication and Synchronization

- Message Queues
- Advanced Memory Management
- Shared Memory Segments

7. Communication and Synchronization

- Message Queues
- Advanced Memory Management
- Shared Memory Segments

IPC: Message Queues

Queueing Mechanism for Structured Messages

- Signals
  - Carry no information beyond their own delivery
  - Cannot be queued
- FIFOs (pipes)
  - Unstructured stream of data
  - No priority mechanism
- Message queues offer a loss-less, *structured*, *priority-driven* communication channel between processes
  `$ man 7 mq_overview`

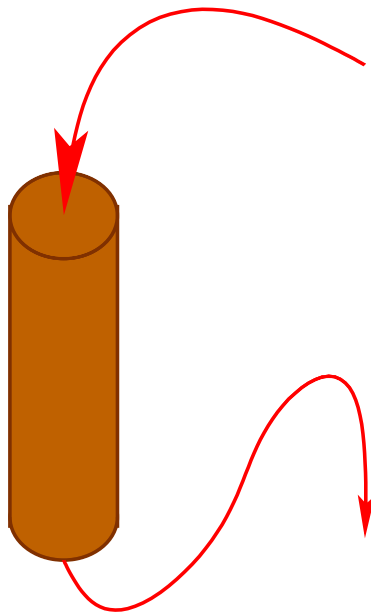
Implementation in Linux

- Message queue files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/mqueue`
- Must link the program with `-lrt` (real-time library)

Structured Communication

Priority, Structured Queues

- Maintain message boundary
- Sort messages by priority



```
mq_send(mqdes, " World!", 7, 20);

mq_send(mqdes, "Hello", 5, 31);


mq_getattr(mqdes, &mq_attr);

msg_len = mq_attr.mq_msgsize;

s = mq_receive(mqdes, buf, msg_len, NULL);

mq_receive(mqdes, buf+s, msg_len, NULL);
```

System Call: mq_open()

Open and Possibly Create a POSIX Message Queue

```
#include <mqueue.h>

mqd_t mq_open(const char *name, int flags);
mqd_t mq_open(const char *name, int flags, mode_t mode,
              struct mq_attr *attr);
```

Description

- Analogous to open(), but not mapped to persistent storage
  - name: must begin with a “/” and may not contain any other “/”
  - flags: only O_RDONLY, O_RDWR, O_CREAT, O_EXCL, O_NONBLOCK; and FD_CLOEXEC flag is set automatically
  - mode: S_IRUSR, S_IWUSR, S_IXUSR, etc.
  - attr: attributes for the queue, see mq_getattr()
    Default set of attributes if NULL or not specified
- Return a message queue descriptor on success, -1 on error

System Call: mq_getattr() and mq_setattr()

Attributes of a POSIX Message Queue

```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes, struct mq_attr *mq_attr)
int mq_setattr(mqd_t mqdes, struct mq_attr *mq_newattr,
               struct mq_attr *mq_oldattr)
```

Description

- The mq_attr structure is defined as

```
struct mq_attr {
  long mq_flags;   // Flags: 0 or O_NONBLOCK
  long mq_maxmsg;  // Maximum # of pending messages (constant)
  long mq_msgsize; // Maximum message size (bytes, constant)
  long mq_curmsgs; // # of messages currently in queue
};
```

- Return **0** on success, **−1** on error

System Call: mq_send()

Send a Message To a POSIX Message Queue

```
#include <mqueue.h>

int mq_send(mqd_t mqdes, char *msg_ptr,
            size_t msg_len, unsigned int msg_prio)
```

Description

- Enqueues the message pointed to by msg_ptr of size msg_len into mqdes
- msg_len must be less than or equal to the mq_msgsize attribute of the queue (see mq_getattr())
- msg_prio is a non-negative integer specifying message priority
  0 is the lowest priority, and **31** is the highest (portable) priority
- By default, mq_send() blocks when the queue is full (i.e., mq_maxmsg currently in queue)
- Return **0** on success, **−1** on error

System Call: mq_receive()

Receive a Message From a POSIX Message Queue

```
#include <mqueue.h>

ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
                   size_t msg_len, unsigned int *msg_prio)
```

Description

- Removes the oldest message with the highest priority from mqdes
- Stores it into the buffer pointed to by msg_ptr of size msg_len
- msg_len must be greater than or equal to the mq_msgsize attribute of the queue (see mq_getattr())
- If msg_prio is not null, use it to store the priority of the received message
- By default, mq_receive() blocks when the queue is empty
- Return the number of bytes of the received message on success, -1 on error

System Call: mq_close()

Close a POSIX Message Queue Descriptor

```
#include <mqueue.h>

int mq_close(mqd_t mqdes);
```

Description

- Also remove any notification request attached by the calling process to this message queue
- Return $\mathbf{0}$ on success, $-\mathbf{1}$ on error

System Call: mq_unlink()

Unlink a POSIX Message Queue File

```
#include <mqueue.h>

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

Description

- Message queues have *kernel* persistence
- Similar to unlink()

Other System Calls

- mq_notify(): notify a process with a signal everytime the specified queue receives a message while originally empty
- mq_timedreceive() and mq_timedsend(): receive and send with timeout

7. Communication and Synchronization

- Message Queues
- Advanced Memory Management
- Shared Memory Segments

Memory and I/O Mapping

Virtual Memory Pages

- *Map* virtual addresses to physical addresses
  - Configure MMU for page translation
  - Support growing/shrinking of virtual memory segments
  - Provide a protection mechanism for memory *pages*
- Implement copy-on-write mechanism (e.g., to support `fork()`)

Memory and I/O Mapping

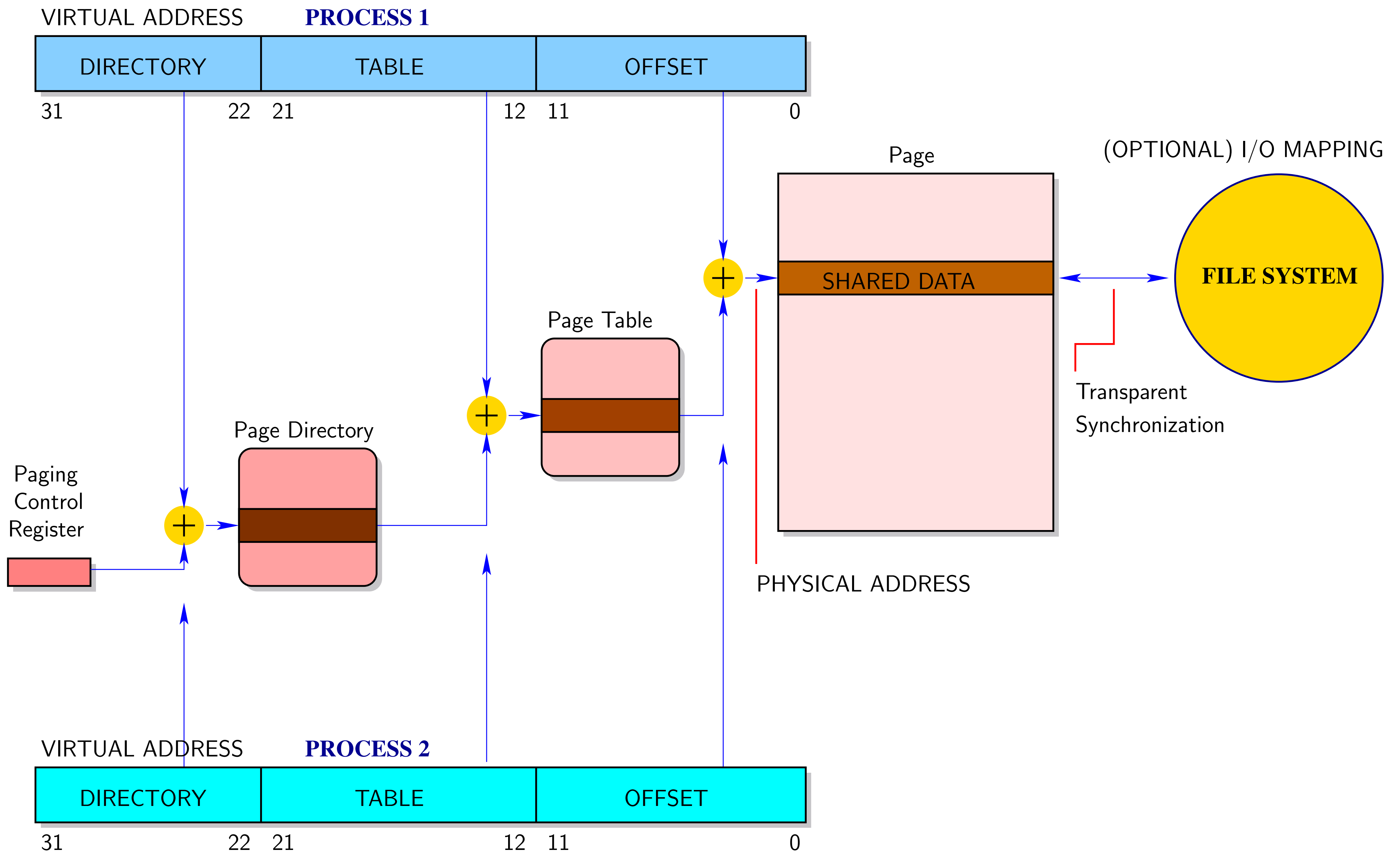
Virtual Memory Pages

- *Map* virtual addresses to physical addresses
  - Configure MMU for page translation
  - Support growing/shrinking of virtual memory segments
  - Provide a protection mechanism for memory *pages*
- Implement copy-on-write mechanism (e.g., to support `fork()`)

I/O to Memory

- *Map* I/O operations to simple memory load/store accesses
- Facilitate sharing of memory pages
  - Use file naming scheme to identify memory regions
  - Same system call to implement private and shared memory allocation

Memory and I/O Mapping



System Call: mmap()

Map Files or Devices Into Memory

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

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

Semantics

- Allocate `length` bytes from the process virtual memory, starting at the `start` address or any fresh interval of memory if `start` is `NULL`
- Map to this memory interval the a file region specified by `fd` and starting position `offset`
- `start` address must be multiple of memory page size; almost always `NULL` in practice
- Return value
  - Start address of the mapped memory interval on success
  - `MAP_FAILED` on error (i.e., `(void*)-1`)

System Call: mmap()

Map Files or Devices Into Memory

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

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

Memory Protection: the prot Argument

- It may be PROT_NONE: access forbiden
- Or it may be built by *or'ing* the following flags

  PROT_EXEC: data in pages may be executed as code
  PROT_READ: pages are readable
  PROT_WRITE: pages are writable

System Call: mmap()

Map Files or Devices Into Memory

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

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

Memory Protection: the flags Argument

- Either

  MAP_PRIVATE: create a private, copy-on-write mapping; writes to the region do not affect the mapped file

  MAP_SHARED: share this mapping with all other processes which map this file; writes to the region affect the mapped file

  MAP_ANONYMOUS: mapping not associated to any file (fd and offset are ignored); underlying mechanism for growing/shrinking virtual memory segments (including stack management and malloc())

System Call: mmap()

Map Files or Devices Into Memory

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

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

Error Conditions

EACCESS: fd refers to non-regular file or prot incompatible with opening mode or access rights
Note: modes O_WRONLY, O_APPEND are forbidden

ENOMEM: not enough memory

Error Signals

SIGSEGV: violation of memory protection rights

SIGBUS: access to memory region that does not correspond to a legal position in the mapped file

System Call: munmap()

Delete a Memory Mapping for a File or Device

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

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

Semantics

- Delete the mappings for the specified address and range
- Further accesses will generate invalid memory references
- Remarks
  - `start` must be multiple of the page size (typically, an address returned by `mmap()` in the first place)
    Otherwise: generate `SIGSEGV`
  - All pages containing part of the specified range are unmapped
  - Any pending modification is synchronized to the file
    See `msync()`
  - Closing a file descriptor does not unmap the region
- Return **0** on success, **−1** on error

7. Communication and Synchronization

- Message Queues
- Advanced Memory Management
- Shared Memory Segments

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem
- Solution: use a *file name* as a meeting point

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem
- Solution: use a *file name* as a meeting point
- Slight problem... one may not want to waste disk space for transient data (not persistent accross system shutdown)
  - `MAP_ANONYMOUS` solves this problem... but looses the association between the file and memory region to implement the rendez-vous

IPC: Shared Memory Segments

Naming Shared Memory Mappings

- Question: how do processes *agree* on a *sharing* a *physical memory* region?
  - *Sharing* is easy: call `mmap()` with `MAP_SHARED` flag
  - *Agreeing* is the problem
- Solution: use a *file name* as a meeting point
- Slight problem... one may not want to waste disk space for transient data (not persistent accross system shutdown)
  - `MAP_ANONYMOUS` solves this problem... but looses the association between the file and memory region to implement the rendez-vous

Implementation in Linux

- Shared memory files are single inodes located in a specific *pseudo-file-system*, mounted under `/dev/shm`
- Must link the program with `-lrt` (real-time library)

System Call: shm_open()

Open and Possibly Create a POSIX Shared Memory File

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

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

Description

- Analogous to open(), but for files specialized into "shared memory rendez-vous", and not mapped to persistent storage

  name: must begin with a "/" and may not contain any other "/"

  flags: only O_RDONLY, O_RDWR, O_CREAT, O_TRUNC, O_NONBLOCK; and and FD_CLOEXEC flag is set automatically

  mode: S_IRUSR, S_IWUSR, S_IXUSR, etc.

System Call: shm_open()

Open and Possibly Create a POSIX Shared Memory File

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

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

Allocating and Sizing a Shared Memory Segment

- The first `mmap()` on a shared memory descriptor allocates memory and maps it to virtual memory of the calling process
- Warning: the size of the allocated region is not yet stored in the descriptor
  - Need to *publish* this size through the file descriptor
  - Use a generic file-sizing system call

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

System Call: shm_unlink()

Unlink a POSIX Shared Memory File

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

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

Description

- Shared memory files have *kernel* persistence
- Similar to `unlink()`
- `close()` works as usual to close the file descriptor after the memory mapping has been performed
- Neither `close()` nor `unlink()` impact shared memory mapping themselves

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
    - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
3. Big problem for *linked data structures* and function pointers

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
3. Big problem for *linked data structures* and function pointers
4. Mapping to a specified address is a fragile solution
   - The `start` argument of `mmap()`

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
3. Big problem for *linked data structures* and function pointers
4. Mapping to a specified address is a fragile solution
   - The `start` argument of `mmap()`
5. Pointers relative to the base address of the segment is another solution (cumbersome: requires extra pointer arithmetic)

About Pointers in Shared Memory

Caveat of Virtual Memory

1. The value of a pointer is a *virtual memory address*
2. Virtual memory is *mapped differently* in every process
   - In general, a pointer in a shared memory segment does not hold a valid address for all processes mapping this segment
3. Big problem for *linked data structures* and function pointers
4. Mapping to a specified address is a fragile solution
   - The `start` argument of `mmap()`
5. Pointers relative to the base address of the segment is another solution (cumbersome: requires extra pointer arithmetic)
6. Note: the problem disappears when forking *after* the shared memory segment has been mapped