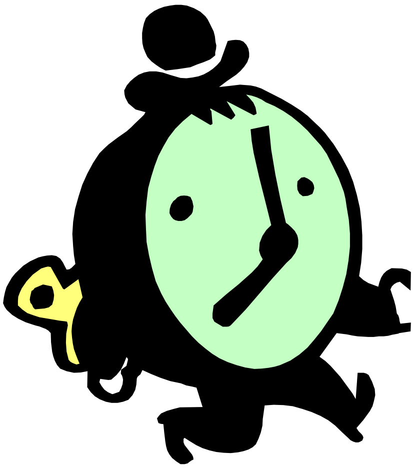
# Real-Time Systems Programming

## Summer-Semester 2002
## Lecture 21
## 27 June 2002



# POSIX Coordination

# *The 5 Minute Review Session*

1) *What is coordination? Which coordination mechanisms do you know?*
2) *What is mutual exclusion?*
3) *What is deadlock? What is livelock?*
4) *What is a conditional critical region? What is a monitor?*
5) *What does Java's* **`synchronized`** *modifier mean?*

# *Coordination in POSIX*

| | *Reliable?* | *Flexible?* | *Fast?* |
|---|---|---|---|
| **Signals** | Sometimes | Very | No |
| **Semaphores** | Yes | Very | Very (w/ shared memory) |
| **Messages** (pipes, fifos, msg queues) | Yes | Not very | Not particularly |
| **Shared Memory** | Yes | Very | Very (w/ semaphores) |



- ***Signals:***
  - An asynchronous event handling mechanism
  - Can also be used as a low-bandwidth communication means
  - Treated in more detail later
- ***Semaphores***
  - Can be used to synchronize access to shared memory
  - However, can also be used as a low-bandwidth communication mechanism in itself
  - ***POSIX.1b*** provides counting semaphore
  - See Lecture 19

# *Where are we?*

- ***POSIX messages***
  - ***Pipes***
  - ***Fifos***
  - ***Msg queues***
- POSIX shared memory

# *POSIX.1 Messages – Pipes and FIFOs*

- An ***indirect*** communication means
- A pipe consists of ***two file descriptors***
  - These are the writing and the reading end
  - Can use standard **`read`** and **`write`** with these file descriptors (fds)
  - File descriptors remain open in processes created via **`fork`** and **`exit`** – thus allowing inter-process communication

# *Pipes and FIFOs cont.*

- Pipes/FIFOs are more appropriate than signals as a communication channel
  - Do not require handler functions, masking, etc.
- Data are queued internally
- Can be ***synchronous*** or ***asynchronous***
  - Switch via **`O_NONBLOCK`** flag in file descriptor

# *Plumbing Hints for Pipes*

- The following is a sequence of steps for setting up pipes between processes:
  1) Call **`pipe`** to create the pipe
  2) Call **`fork`** – both child and parent now have access to the pipe
  3) As the child process will exec a new process with its own memory image (which does not include the fds of the pipe), will have to duplicate the fds into known locations (in the example, **`MY_PIPE_READ`** and **`MY_PIPE_WRITE`**). We can use **`dup2`** for that purpose; but first, close the new fds to make sure that they are available
  4) The child calls **`exec`**

# *Example of POSIX Pipes*

- Server process communicates to terminals via pipes

```
int pipe_ends[2];

// Create a new client process
PIPE(pipe_ends);
child = fork();
if (child) {
  // Parent process
  break;
} else {
  // Child process
  // Make the pipe ends be fds MY_PIPE_READ and MY_PIPE_WRITE
  CLOSE(MY_PIPE_READ);
  CLOSE(MY_PIPE_WRITE);
  DUP2(pipe_ends[0], MY_PIPE_READ);
  DUP2(pipe_ends[1], MY_PIPE_WRITE);
  CLOSE(pipe_ends[0]);
  CLOSE(pipe_ends[1]);
  EXECLP("terminal", arg1, arg2, ..., NULL);
}
```

# *Limitations of Pipes I*

- Pipes are used for communication in a ***"pipelined" manner***
  - This corresponds to familiar shell syntax:
    **`% prog1 | prog2 | prog3 | prog4`**
  - Can alter this linear topology using **`dup2`** and **`close`**
- Pipes are only ***one-directional***
  - Need two pipes for bidirectional communication
- Using pipes for inter-process communicaton requires a ***child-parent relationship*** between the communicating processes
  - What if multiple clients want to communicate with multiple servers?

# *FIFOs*

- A ***FIFO*** is a pipe that ***has a name in the file system***
- Any process with the appropriate permissions can access the pipe
- ***This removes the restrictions on the topology of communicating processes***
- FIFOs are created with **`mkfifo`**
- FIFOs are opened with **`open`**
  - This now (unlike pipe) returns only one file descriptor
  - Opening with **`O_RDONLY`** returns the reading end
  - Opening with **`O_WRONLY`** returns the writing end
  - Results of opening with **`O_RDWR`** are undefined

# Limitations of Pipes II

- ***Prioritization***
  - A pipe is strictly FIFO
  - There is no built-in means for message prioritization
  - Could prioritize at application level – but this would require significant overhead
- ***Asynchronicity***
  - In principle, do not have to wait for somebody reading what we have written if we use **`O_NONBLOCK`**
  - However, there is only limited buffer space – when this fills up, the writer process starts blocking again
  - There is no portable way to control (or just know) the amount of buffer space for a given pipe!

# *Limitations of Pipes III*

- ***Lack of control over pipe structure***
  - Pipes cannot provide any information about their state – i.e., how many data have been written into a pipe
- ***Lack of structure of the data stream***
  - Pipes are fairly pure instances of the (very powerful) UNIX file abstraction; a pipe is nothing more than a stream of bytes
  - Can read or write an arbitrary number of bytes at a time
    - This can be problematic if variable-sized messages are passed
    - A single offset error can corrupt all subsequent messages

# *System V UNIX Message Queues*

- System V UNIX provides message queues that provide inter-procedure calls (IPCs)
- However, are ***clumsy*** to use:
  - Named by numbers, rather than strings
  - **`ipcs`** and **`ipcrm`** maintain separate "namespace"
- Are ***very slow***
- Hence, POSIX.1b working group abandoned System V message queues (as well as e.g. its semaphores and shared memory) in favor of something new

# *POSIX Message Queues*

- POSIX message queues (henceforth abbreviated MQs) are named objects which operate basically as pipes do – however:
  - An MQ can have ***many readers*** and ***many writers***
  - ***Priority*** may be associated with messages
- Intended for communication between processes (not threads)
- Early draft proposals were very flexible
  - Allowed for example the passing of messages without copying (instead just mapping from address space to address space)
  - However, the resulting complexity lead to very slow implementations

# POSIX MQs

- The final definition of MQs leans more toward simplicity
- MQs have attributes which indicate their maximum size, the size of each message, the number of messages currently queued etc.
- An attribute object is used to set the queue attributes when the MQ is created

# *POSIX MQs*

- MQs are given a name when they are created
- To gain access to the queue, requires an **mq_open name**
- **mq_open** is used to both create and open an already existing MQ

```
#include <mqueue.h>
mqd_t              child_mq;
struct mq_attr     child_mq_attr;

child_mq= mq_open("/terminal.0",
    O_CREAT|O_EXCL|O_RDWR,
    MS_IRWXU, &child_mq_attr);
if (child_mq == mqd_t - 1) {
  perror("mq_open");
}
```

# *Naming and Accessing MQs*

- MQs are named like files, and opening them does look a lot like opening a file
- However, unlike FIFOs and pipes, message queues are not accessed using **`open`**, **`read`** and **`write`**; instead, sending and receiving messages is done via **`mq_send`** and **`mq_receive`**

# *MQ Naming cont.*

- Naming MQs like files allows vendors to implement message queues using the UNIX file system
- However, the system calls associated with the file system have a significant ***overhead***
- Therefore, MQs are often implemented w/o accessing the file system – try **`ls`** to find out!
- ***Recall***: UNIX pathnames may be
  - ***absolute***: beginning with a slash ("/")
  - ***relative*** to the current working directory: without leading /
- Whether or not MQs are implemented using the file system may ***affect portability***

# MQ Naming cont.

- ***Example I:***
  - Assume the current working directory is **`/home/joe`**
  - Assume that we create MQs named **`"/home/joe/my_mq"`** and **`"my_mq"`**
  - An implementation using the file system treats these MQs as identical
  - Otherwise, the MQs are treated as being different
- ***Example II***:
  - Assume that different processes in different working directories each create MQs named **`"my_mq"`**
  - An implementation using the file system treats these MQs as different
  - Otherwise, the MQs are identical

# *Portable MQ Naming*

- Even if MQs are not using the file system, ***implementations may differ how they interpret slashes***
- To avoid the aforementioned portability problems, one should follow these two rules in naming POSIX MQs:

1. ***Start the name with a /***
2. ***Do not use any other / characters***



- An implementation may interpret a slash as a special character, separating components as directories, and performing proper permission checks
- Other implementations may treat slashes like any other characters, and use the whole name just as a hash key
- The conformance document of the product should tell you what is actually happening on your system

# *POSIX MQs*

- Data are read/written from/to a character buffer.
- If the buffer is full or empty, the sending/receiving process is blocked unless the attribute **O_NONBLOCK** has been set for the queue (in which case an ***error return*** is given)
- If senders and receivers are waiting when an MQ becomes unblocked, it is not specified which one is woken up unless the ***priority scheduling*** option is specified

# *POSIX MQs*

- A process can also indicate that a signal should be sent to it when an empty queue receives a message and there are no waiting receivers
- In this way, a process can ***continue executing*** whilst waiting for messages to arrive or one or more message queues
- It is also possible for a process to wait for a ***signal*** to arrive; this allows the equivalent of ***selective waiting*** to be implemented
- If the process is multi-threaded, each thread is considered to be a potential sender/receiver in its own right

# *Robot Arm Example I*

```
typedef enum {xplane, yplane, zplane} dimension;

void move_arm(int D, int P);

#define DEFAULT_NBYTES 4
int nbytes = DEFAULT_NBYTES;

#define MQ_XPLANE  "/mq_xplane" // MQ name
#define MQ_YPLANE  "/mq_yplane" // MQ name
#define MQ_ZPLANE  "/mq_zplane" // MQ name
#define MODE ...                // Mode info for mq_open
```

# Robot Arm Example II

```
void controller(dimension dim) {
  int    position, setting;
  mqd_t  my_queue;       // Message queue
  struct mq_attr ma;     // Attributes
  char   buf[DEFAULT_NBYTES];
  ssiz_t len;

  position = 0;
  switch(dim)  {         // open appropriate message queue
    case xplane:
      my_queue = MQ_OPEN(MQ_XPLANE, O_RDONLY, MODE, &ma);
      break;
    case yplane:
      my_queue = MQ_OPEN(MQ_YPLANE,...);
      break;
    case zplane:
      my_queue = MQ_OPEN(MQ_ZPLANE,...);
      break;
    default:
      return;
  };
```

# Robot Arm Example III

```
  while (1) {
    // read message
    len = mq_receive(my_queue, &buf[0], nbytes, null);
    setting = *((int *)(&buf[0]));
    position = position + setting;
    move_arm(dim, position);
  };
}
```

# *Robot Arm Example IV*

- The main program that creates the controller processes and passes them the appropriate coordinates:

```
int main(int argc, char **argv) {
  mqd_t           mq_xplane, mq_yplane, mq_zplane;
  struct mq_attr  ma; // queue attributes
  int             xpid, ypid, zpid;
  char            buf[DEFAULT_NBYTES];


  // Set MQ attributes
  ma.mq_flags = 0;    // No special behaviour
  ma.mq_maxmsg = 1;
  ma.mq_msgsize = nbytes;

  mq_xplane = MQ_OPEN(MQ_XPLANE, O_CREAT|O_EXCL, MODE, &ma);
  mq_yplane = ...;
  mq_zplane = ...;
```

# Robot Arm Example V

```
  // Duplicate the process to get three controllers
  switch (xpid = FORK()) {
    case 0: controller(xplane); exit(0); // child
    default:                             // parent
     switch (ypid = FORK()) {
        case 0: controller(yplane); exit(0);
        default:                         // parent
         switch (zpid = FORK()) {
            case 0: controller(zplane); exit(0);
            default:                     // parent
              break;
          }
      }
  }

  while (1) {
    // set up buffer to transmit each co-ordinate
    //   to the controllers, for example
    MQ_SEND(mq_xplane, &buf[0], nbytes, 0);
  }
}
```

# *Closing and Unlinking MQs*

- If we can open MQs, we must be able to close them again
- Similarly, if we can create MQs, we must be able to delete them again
- The functions **`mq_close`** and **`mq_unlink`** provide these functionalities, mimicking their corresponding file-based calls, **`close`** and **`unlink`**

# *Closing MQs*

- Calling **`mq_close`** has no effect on the contents of the MQ
  - We can open an MQ, send ten messages to the queue, close the queue, open it again the next day, and retrieve those ten messages again
- MQs are closed when the process creating them calls **`exit`**, **`_exit`**, or one of the **`exec`** functions
- This is somewhat different from files, where a file can be prevented from being closed upon calling an exec function (by clearing the **`FD_CLOEXEC`** flag, using **`fcntl`**)

# *Unlinking MQs*

- Calling **`mq_unlink`** does for message queues what **`unlink`** does for files
- If no one has the MQ open when **`mq_unlink`** is called, then the MQ is immediately deleted, and all the messages in it are lost
- Otherwise, the destruction of the MQ is delayed until the last processes closes the MQ
  - However, the name of the MQ is removed immediately
  - After **`mq_unlink`** is called, the only processes that can access the MQ are the ones that had it open before **`mq_unlink`** was called
  - Then again, if those processes **`fork`**, their children can also access the MQ, as MQs are inherited across **`fork`**, like files

# *Cleaning Up After Yourself*

- Message queues, like files, are ***persistent***
  - they continue to exist even if no one has the queue open
  - they are not eliminated when the process creating them ceases to exist
  - they are only eliminated when the system goes down
- However, as message queues do not necessarily exist in the file space, we may not be able to remove the message queues – ***there is no POSIX equivalent of `rm` for MQs!***
- Therefore, unless we still want those MQs for debugging purposes, it is recommended to ***unlink the MQs as soon as everybody who needs them has opened them***

# *Limitations of POSIX MQs*

MQs have some definite advantages over pipes and FIFOs, but some limitations remain:

- They ***are*** queues after all
  - This may be clumsy for some apps
  - ***Examples***: stack-based operations; general data sharing
- ***Efficiency***
  - MQs always require two copy operations – from the sending app to the OS, from the OS to the receiving app
  - It would be faster to let the OS decide about the target memory location – but this is outside of POSIX.1b
- An alternative here is ***shared memory***

## *Where are we?*

- POSIX messages
- ***POSIX shared memory***
  - ***Naming, opening***
  - ***Memory mapping***
  - ***Aligning***
  - ***For regular files: synchronization***

# *Shared Memory*

- Very low-level means of communication
- ***Pro***: Flexibility, Speed
- ***Con***:
  - More difficult to use than signals or message queues – access has to be ***explicitly synchronized*** (e.g. with semaphores, mutexes, condition variables)
  - Unlikely to perform well in a distributed environment (unless we have an architecture that explicitly supports distributed shared memory)
- ***POSIX.1b*** defines a shared memory interface

# *POSIX Shared Memory*

- Per default, different processes operate on different pieces of memory
- POSIX.1b defines a mechanism that allows processors to share memory
- If one process writes a value into a particular byte of shared memory, the other process sees this ***"immediately"***
  - On ***multiprocessor systems*** – ruled outside of the scope of POSIX.1b considerations – the actual delay depends on the underlying hardware memory coherence system

# *Complications with Shared Memory*

- The OS is not involved in your use of this memory – there are no run-time checks, no copy operations
- The free-form nature of shared memory allows flexibility and efficiency – ***and also gives you the freedom to hang yourself pretty badly!***
- ***Example***: implementing a circular linked list in shared memory

```
dequeue(element *e)
{
  e->prev->next = e->next;
  e->next->prev = e->prev;
  e->next = e->prev = NULL;
}
```

*What may happen if two processors dequeue at once?*

# *More Complications: File Mapping*

- Another complications is related to how POSIX.1b standardized shared memory – which is intertwined with another facility: *file mapping*
- ***File mapping*** allows an app to map a file into its address space and then access it as if it were memory
- File mapping works for ***any*** type of file:
  - Disk files
  - Frame buffers
  - ... and shared memory
- This complicates the shared memory interface somewhat, as file mapping and shared memory have somewhat diverging requirements

# *What does My System Provide?*

- In principle, an OS indicates the presence of shared memory by defining **`_POSIX_SHARED_MEMORY_OBJECTS`**
- Shared memory requires file mapping – but a vendor may be interested in just file mapping, without providing shared memory
- Hence, the following feature test macro zoo:

| **Function** | **Present According to Which Options?** |
|---|---|
| `mmap, munmap, ftruncate` | `_POSIX_MAPPED_FILES` or `_POSIX_SHARED_MEMORY_OBJECTS` |
| `shm_open, shm_close, shm_unlink` | `_POSIX_SHARED_MEMORY_OBJECTS` |
| `mprotect` | `_POSIX_MEMORY_PROTECTION` |
| `msync` | `_POSIX_MAPPED_FILES` and `_POSIX_SYNCHRONIZED_io` |

# *Memory is a File ... Sort Of*

- Shared memory, message queues, and semaphores are all cousins in the POSIX world and have interfaces similar to ordinary UNIX disk files
  - Similar naming
  - Similar schemes for creation and deletion, opening and closing
- However, each interface differs according to the specific requirements of the functionalities
- Message queues are the simplest of the lot – open the queue, send a message
- Shared memory requires file mapping as additional step

# *Opening Shared Memory Objects*

- Shared memory objects (SMOs) are created with **`shm_open`**:

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

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

- **`shm_open`** takes exactly the same arguments as open, and also returns a file descriptor
- In this, **`shm_open`** is different from **`mq_open`**, which returned an object of the type **`mqd_t`**, which is not necessarily a file descriptor

# *Naming SMOs*

- **`shm_open`** operates on a name that looks like a file name, but ***may not exist in the file system***
- Like message queues, SMOs may not show up in the output of an **`ls`**
- For ***portability***, thus the same naming rules apply as for message queues (recall ...)
- Even though **`shm_open`** returns a normal file descriptor, we ***cannot use most standard file operations*** (**`read`**, **`write`**, **`lseek`**)
- ***Exceptions:*** **`ftruncate`** (which set the size) and **`close`** can both be used on shared memory objects

# *Creating SMOs*

- The **oflag** parameter of **shm_open** can be set to **O_RDONLY** or **O_RDWR** to indicate the desired access types
- **oflag** also determines whether an SMO is created, by setting **O_CREAT**
- If we try to create an SMO that already exists, **shm_open** fails silently – unless we also set **O_EXCL**
- Unlike message queues, ***no additional parameters*** are needed to describe the newly created SMO – the relevant parameters are set when the SMO is mapped into your address space, using **mmap**

# Sizing SMOs

- However, there is one aspect of SMOs that neither **`shm_open`** nor **`mmap`** address: the size of the SMO
- An SMO has zero size when it is first created, or when somebody opens it with the **`O_TRUNC`** flag set
- POSIX.1b provides the (confusingly named) **`ftruncate`** for changing an SMO's size:

```
#include <unistd.h>

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

- If an existing SMO is reduced in size, the truncated data are lost
- If one process calls **`ftruncate`**, all processes see this

# *Closing SMOs*

- If we open something, we have to be able to close it again
- However, as **`shm_open`** returns a normal file descriptor, and there is no other information needed for closing a file, we can use the normal **`close`** operation for closing an SMO

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

int fd;   // File descriptor for SMO
int i;

fd = shm_open("/shared_memory", ...);
...
i = close(fd);
```

# *Closing SMOs cont.*

- Recall that creating SMOs is a two-step procedure:
  - The SMO has to be opened (**`shm_open`**)
  - The SMO has to be mapped into memory (**`mmap`**)
- **`close`** ***only undoes the first of these two steps***, it recycles the file descriptor
- Once the memory is mapped in, one can safely close the file descriptor – in fact, one should do so for tidiness

## *Destroying SMOs*

- To really remove an SMO, we have to delete it using **shm_unlink**

```
int i;

i = shm_unlink("/shared_memory");
```

- Unlinking an SMO has the same semantics as unlinking a file or message queue



- If any processes are still using the SMO when it is unlinked, then those instances of the SMO remain viable until each individual process ceases to use the SMO – each process has to **close**, and also **munmap**, their shared memory before it truly ceases to exist
- **exit** and **exec** implicitly close SMOs

# *Mapping SMOs Into Memory*

- To get the SMO into the address space of a process, one has to map the file descriptor returned by **`shm_open`** into the processes address space using **`mmap`**

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

void *mmap(void   *where_i_want_it,
           size_t length,
           int    memory_protections,
           int    mapping_flags,
           int    fd,
           off_t  offset_within_shared_memory);
```

- **`mmap`** returns the address where the SMO is actually placed

# *Where the SMOs Will Be Placed*

- The first parameter to **`mmap`**, **`where_i_want_it`**, is an address that we can pass in as a hint to the system on where in memory the SMO shall be placed
- However, a POSIX compliant system is not obliged to obey this hint
- If we want to insist on the hint being used, we can set **`MAP_FIXED`** in the mapping flags
  - If the system is still unable to use the hint, **`mmap`** fails
  - Implementations do not have to support **`MAP_FIXED`**

# *Aligning SMOs Among Processes*

- Sometimes we do not really care where in memory the SMO is placed – but we do care about different processes having the SMO mapped to the same addresses
  - For example, to share pointers among processes
- One fairly portable way of aligning SMO objects in the memory areas of different processes is the following:
  - When the first maps in the SMO, let the system choose the address, by giving a hint of zero
  - Then communicate the resulting address to the other processes, and let the other processes use this address as a hint

# *Where the System Places SMOs*

- Usually, the SMOs have to start at a ***page boundary***
- Typical page sizes are 4096 and 8192 – check **`PAGESIZE`** to find the page granularity of your system
- For portable use of **`mmap`**, one has to make sure that any specified address hint, as well as the file offset and length, are multiples of **`PAGESIZE`**
  - Some systems check that this is the case
  - Others perform ***silent alignment!***

# *An Example of Silent Alignment*

- Assume the following scenario:
  - **`PAGESIZE`** = 4096
  - **`length`** = 5
  - **`offset_within_shared_memory`** = 10
- **`mmap`** returns a pointer equal to 5 mod 4096 (5001, 8197, ...) - say it returns 5001
  - Then the address range 5001 ... 5010 constitutes our shared memory
  - However, the remaining address ranges of that page – 4096 ... 5000 and 5011 ... 8191 ***– are now also shared memory!***
- ***Any writes to these remaining ranges, outside of what we asked for, are also visible to all processes!!***

# *Unmapping Shared Memory*

- ***Recall:*** once an SMO is **`mmap`**ed, we can close the file descriptor, and still retain the mapping
  - Mappings are inherited across a **`fork`**
  - Mappings are removed upon **`exit`** or **`exec`**
- **`munmap`** explicitly unmaps SMOs

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

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

- **`munmap`** removes the mappings for the pages containing the address rage begin through **`begin`** + **`length`** – 1
- Again, better make sure that **`begin`** and **`length`** are multiples of **`PAGESIZE`**

# *Shared Memory Persistence*

- Like message queues, POSIX.1b shared memory is persistent: it remains around until it is explicitly removed (or until the system shuts down)
- Therefore, we can fill an SMO with data, unmap the SMO, finish all processes, start a process again, and retrieve those data again (⇨ Homework)
- This persistence also means that we have to be careful about the SMO contents when we first open them, as they may already contain some (invalid) data
- It is therefore prudent to first unlink and then re-create SMOs before using them

# *Backing Store*

- ***Recall:*** **`mmap`** can be used for any file object – shared memory for example, but also physical files
  - We can refer to the underlying object that we have mapped in as ***backing store***
  - The memory range in which we have mapped the file object is merely a ***shadow image*** of that backing store
- In the case of SMOs, the shadow image and the backing store are one and the same: the physical memory to share information
- With disk files, however, there is a dichotomy – at any point in time, the backing store and its shadow image do not have to be identical, and they have very different access times

# *For Regular Files: Synchronization*

- The contents of the backing store and our shadow image of it are only guaranteed to be identical
  - when the mappings are removed (**munmap**), or
  - when we explicitly synchronize the mapping
- **msync** performs this synchronization:

```
#include <unistd.h>
#ifdef _POSIX_MEMORY_PROTECTION
#ifdef _POSIX_SYNCHRONIZED_IO
#include <sys/mman.h>

void    *begin;
size_t  length;
int     i, flags;

i = msync(begin, length, flags);
#endif
#endif
```

# *Summary POSIX Pipes and FIFOs*

- The main communication categories provided by ***POSIX*** are signals, messages, and shared memory/semaphores
- ***POSIX*** message types are pipes, FIFOs, and message queues
- ***FIFOs*** are basically ***pipes*** with a name in the file system – they are thus better suited for non-pipelined topologies than pipes
- However, both pipes and FIFOs are limited wrt
  - ***Prioritization*** (cannot assign priorities)
  - ***Asynchronicity*** (no control over buffer space)
  - ***Interrogability*** (cannot inquire about internal state)
  - Message ***structure*** (just a stream of bytes)

# *Summary POSIX Message Queues I*

- ***POSIX*** message queues (MQs) allow ***asynchronous***, ***many to man***y communication, that also allows the definition of ***priorities***
- The handling of MQs looks very much like the handling of files
- MQs may or may not be implemented using the UNIX file system
- Care must be taken with ***naming MQs*** in a portable fashion

# *Summary POSIX Message Queues II*

- MQs are ***persistent***
  - The application has to pay attention to cleaning up after itself
  - There is no MQ-equivalent to what **`rm`** does for files
  - By themselves, MQs only go away when the system goes down
- Overall, MQs give more ***structure and control*** than pipes and FIFOs, and they can easily be extended across machine boundaries
- However, MQs have some limitations as well
  - Even with priorities, MQs are still queues
  - The run-time overhead of the OS calls and copy operations

# *Summary Shared Memory*

- Shared memory is a low-level, bare bones means of exchanging information between processes
- Shared memory is very ***efficient***, but has to be handled with care – i.e., ***explicit synchronization*** (semaphores, mutexes, etc.)
- Using shared memory is a two-stage process:
  - ***Creation*** of the shared memory object (SMO)
  - ***Mapping*** the SMO into the processes memory
- Like MQs, SMOs are ***persistent***
- SMOs are always aligned to page boundaries – we should set lengths and offsets accordingly, to avoid implicit alignments