Operating Systems
2010/2011

Processes

Johan Lukkien
Agenda

• Processes, threads
• API for creation and termination
  – POSIX
  – Processes
• Implementation aspects
• Communication and Synchronization
  – Shared Memory
  – Message passing
**Concepts in OS: Process, Thread**

- **Process** ("program in execution")
  - defines a data space (virtualization of the memory)
  - realizes transparency for multiple activities
  - defines ownership of resources
  - has at least one associated thread
  - unit of deployment (distribution)
  - unit of fault containment

- **Thread**
  - unit of concurrency (virtualization of the processor)
  - unit of scheduling
    - though in case of one thread per process, the *process* is often said to be the unit of scheduling
  - operates in an address space
    - i.e., in a process; several threads may share this
  - has an associated *execution state*
    - place where it is in the code, stack image and return addresses of function calls, values of processor registers
Motivation for using processes

- Introduce safety boundaries
  - separate users
  - separate distinct, independent tasks

- Follow solution structure
  - e.g. (older) organization of the Apache web server
    - a managing process
    - client process per connection

- Increase concurrency level

- Allow for distribution
Apache organization (preforking)

Picture from 'multitasking server architecture', Grone, Knopfel, Kugel, Schmidt, Potsdam University
Reasons to introduce threads

- Increase concurrency level
  - performance
    - hide latency
    - increase responsiveness
    - exploit platform concurrency
  - discriminate importance levels in activities
    - e.g. interrupt routines

- Deal with the natural concurrency
  - natural organization, structure ... e.g.
    - thread per event
    - thread per resource
    - thread per (active) external interaction sequence
      - e.g. in user interfaces
Structuring by threads

- This program must deal with three input sources

- Possible program structure:
  - while ... do
    - check keyboard;
    - check network;
    - check mouse;
    - wait a while
  od

- Simpler organization
  - start three threads
  - each thread takes care of one source

- Downside: shared resource (global data)
Two issues

1. How to *use* processes (and threads)?
   - primitives for manipulation: the API provided by the OS
     - creation, termination
     - exchange of information
       - *synchronization*: exchange events
       - *communication*: in addition, exchange data
   - interference of concurrent activities
     - what if one process modifies data another one examines?
   - program design [other courses]
     - typical parallel program structures, design criteria
       - e.g. functional-, data- and result parallelism
     - distributed systems, algorithms

2. How do we implement
   - processes, threads?
   - communication facilities?
Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing
API standardization: POSIX

- Portable Operating System Interface
  - UNIX-like

- Goal: source-code portability of applications
  - in practice: just reduce portability effort

- Standard, IEEE, ANSI, ISO, developed in API chapters:
  - mandatory (‘base’) & optional parts, per chapter
  - the entire set of chapters covers ‘everything’

- Versioning: Posix 1003.1x-year, where x is the chapter
  - x=a, or no x: basic set of systems calls, like UNIX; 1003.1a-2004
  - x=b: real-time extensions (also: POSIX.4); 1003.1b-1993
  - x=c: multi-threading support (also: POSIX.4a); 1003.1c-1995
  - x=d: additional real-time extensions; 1003.1d-1999
  - x=j: advanced real-time extensions; 1003.1j-2000
POSIX compliance

- POSIX standardizes on the API !!
  - and not on the implementation

- A system supporting POSIX provides
  - a host language and compiler (often: C)
  - interface definition files (e.g., C-header files)
    - including standardized ways to define included optional parts
  - interface implementation binary or code (e.g., C-libraries)
  - a run-time system (a platform: OS or the like)

- Notes:
  - POSIX is NOT any particular Unix System
  - Many OS’es have a POSIX face
POSIX processes (1003.1)

• Starting point is an already running process
  – shell (command interpreter) or other program in the GUI (windows, Xwindows)
  – that was started through a login procedure

• POSIX 1003.1 API: \texttt{fork()} – \texttt{exec()} – \texttt{exit()} – \texttt{wait()}

• POSIX process:
  – an executing program
  – unit of distribution, concurrency, \textit{scheduling}
  – single address space: memory management
Create new process: code fragment

- `fork()` creates an identical copy of the caller in a separate memory space; only the return value stored in variable ‘child’ differs between the two: 0 for the child, child-processid for the parent

- `execlp()` overwrites the calling process with its argument (file with an executable program)
  - This means that code following `execlp` is never reached

- `perror()` is a generic error printing routine

```c
pid_t child;

......
child = fork();
if (child<0) /* error occurred */  { perror ("fork"); exit (-1); } 
if (child == 0) /* the child */ {
    execlp ("/bin/ls", "ls", arg0, arg1, ..., NULL); /* this place is reached only in case of error */
    perror ("execlp"); exit (-1);
}
else /* the parent; child == process id of child */ {
    /* do whatever you want, e.g., just return from this routine */
}
......
```
Before and after a `fork()`

Parent process about to perform `child = fork()`

Variable `child` of child process equals 0

Variable `child` in parent points to child process

Parent and child process after `fork()`

`child` is a literal copy of parent
Termination of children: fragment

- Use `exit(status)` to terminate

- Need to wait for children to free child’s resources
  - on many systems at least

- Functions `wait()`, `waitpid()`
  - `wait()` blocks until a child returns

- Alternative: (not shown here): use asynchronous notification: signals

child:

..... exit (23);

parent:

pid_t child, terminated; int status;

/* blocking wait */
while (child != wait (&status)) /* nothing */;

/* or polling wait */
terminated = (pid_t) 0;
while (terminated != child) {
    terminated = waitpid (child, &status, WNOHANG);
    /* other useful activities */
}
/* both cases: status == 23 */
The life of a process
POSIX/UNIX processes in operation

- Processes receive a unique number through which they can be referred
- Parent-child relationships are recorded
- Orphans are attached to the system ‘grandfather’ called *init* (process # 1)
- *Init* is also the process that initiates the login of new users
- System services are processes started by *init*, by themselves capable of starting new ones
- Network services are executed under control of *inetd* (the internet daemon)
  - e.g. RPC support, ftp, telnet, rlogin etc.
Some processes on the SANDPIT cluster

```bash
/home/johanl> ps uax

<table>
<thead>
<tr>
<th>USER</th>
<th>PID</th>
<th>%CPU</th>
<th>%MEM</th>
<th>VSZ</th>
<th>RSS</th>
<th>TTY</th>
<th>STAT</th>
<th>START</th>
<th>TIME</th>
<th>COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>1</td>
<td>0.0</td>
<td>0.0</td>
<td>1344</td>
<td>424</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:26</td>
<td>init [3] --in</td>
</tr>
<tr>
<td>root</td>
<td>4</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:03</td>
<td>[keventd]</td>
</tr>
<tr>
<td>root</td>
<td>7</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>1:58</td>
<td>[kswapd]</td>
</tr>
<tr>
<td>root</td>
<td>8</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:37</td>
<td>[bdflush]</td>
</tr>
<tr>
<td>root</td>
<td>9</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>1:07</td>
<td>[kupdatd]</td>
</tr>
<tr>
<td>root</td>
<td>10</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:00</td>
<td>[mdrecoveryd]</td>
</tr>
<tr>
<td>root</td>
<td>11</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:00</td>
<td>[kcopyd]</td>
</tr>
<tr>
<td>root</td>
<td>12</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:00</td>
<td>[kmirrord]</td>
</tr>
<tr>
<td>root</td>
<td>190</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:00</td>
<td>[khubb]</td>
</tr>
<tr>
<td>root</td>
<td>8554</td>
<td>0.0</td>
<td>0.0</td>
<td>1700</td>
<td>760</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>0:00</td>
<td>/sbin/devfsd /d</td>
</tr>
<tr>
<td>root</td>
<td>8849</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>3:20</td>
<td>[kjournald]</td>
</tr>
<tr>
<td>root</td>
<td>9874</td>
<td>0.0</td>
<td>0.0</td>
<td>1688</td>
<td>648</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>1:36</td>
<td>/usr/sbin/synlo</td>
</tr>
<tr>
<td>root</td>
<td>10000</td>
<td>0.0</td>
<td>0.0</td>
<td>1480</td>
<td>520</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>0:00</td>
<td>/sbin/hcpcd -H</td>
</tr>
<tr>
<td>named</td>
<td>10129</td>
<td>0.0</td>
<td>0.0</td>
<td>11868</td>
<td>1232</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:02</td>
<td>/usr/sbin/named</td>
</tr>
<tr>
<td>root</td>
<td>10214</td>
<td>0.0</td>
<td>0.0</td>
<td>3172</td>
<td>860</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>0:03</td>
<td>/usr/sbin/sshd</td>
</tr>
<tr>
<td>rpc</td>
<td>10350</td>
<td>0.0</td>
<td>0.0</td>
<td>1576</td>
<td>620</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>1:09</td>
<td>/sbin/portmap</td>
</tr>
<tr>
<td>root</td>
<td>10498</td>
<td>0.0</td>
<td>0.0</td>
<td>3944</td>
<td>624</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>3:37</td>
<td>/usr/sbin/ypser</td>
</tr>
<tr>
<td>root</td>
<td>10512</td>
<td>0.0</td>
<td>0.0</td>
<td>5848</td>
<td>680</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:25</td>
<td>/usr/sbin/ypbin</td>
</tr>
<tr>
<td>nobody</td>
<td>10569</td>
<td>0.0</td>
<td>0.0</td>
<td>1592</td>
<td>632</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>0:00</td>
<td>/sbin/rpc.statd</td>
</tr>
<tr>
<td>root</td>
<td>10579</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>3:13</td>
<td>[nfsd]</td>
</tr>
<tr>
<td>root</td>
<td>10580</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:00</td>
<td>[lockd]</td>
</tr>
<tr>
<td>root</td>
<td>10581</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>0:06</td>
<td>[rpciod]</td>
</tr>
<tr>
<td>root</td>
<td>10582</td>
<td>0.0</td>
<td>0.0</td>
<td>0</td>
<td>0</td>
<td>?</td>
<td>S</td>
<td>Feb14</td>
<td>2:48</td>
<td>[nfsd]</td>
</tr>
<tr>
<td>root</td>
<td>10590</td>
<td>0.0</td>
<td>0.0</td>
<td>1548</td>
<td>592</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>0:28</td>
<td>/usr/sbin/rpc.m</td>
</tr>
<tr>
<td>root</td>
<td>11266</td>
<td>0.0</td>
<td>0.0</td>
<td>1976</td>
<td>624</td>
<td>?</td>
<td>Ss</td>
<td>Feb14</td>
<td>0:03</td>
<td>/usr/sbin/xinetd</td>
</tr>
</tbody>
</table>

-pidfile /var/run/xinetd.pid -stayalive -reuse
johanl 16966 0.0 0.0 2872 1804 pts/12 Ss 14:45 0:00 -tcsh
johanl 17221 0.0 0.0 2348 848 pts/12 R+ 15:06 0:00 ps uax
```
Simple command interpreter

```c
int main(void)
{
    char buf[MAXLINE];
    pid_t pid;
    int status;

    printf("%s "); /* print prompt (printf requires %s to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {
        buf[strlen(buf) - 1] = 0; /* replace newline with null */
        if ( (pid = fork()) < 0)
            err_sys("fork error");
        else if (pid == 0) { /* child */
            execvp(buf, buf, (char *)0);
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }
        else { /* parent */
            if ( (pid = waitpid(pid, &status, 0)) < 0)
                err_sys("waitpid error");
            printf("%s ");
        }
    }
    exit(0);
}
```
Explanation

1. **fgets** reads one line at a time.
   - It returns a null pointer for a single EOF (end-of-file) character.

2. **strlen** gives the length of a string (excluding the last \0).

3. **fork** creates an exact copy of the old process.
   - It returns 0 to the child (= new) process.
   - It returns the process ID of the child process to the parent (= old) process.

4. In the child process **execvp** replaces the process by a new program.

5. The parent calls **waitpid** to wait for the child (**pid**) to terminate.
   - Otherwise the two processes run concurrently.
   - **waitpid** also returns the termination status of the child process; in the example this is ignored.

6. Main limitation of this example:
   - Commands with arguments are not possible.
Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing
Process State

• A process is represented by a *Process Control Block* (PCB) “inside” the kernel

  – all info needed to run it
  – identified by an ID or a pointer to this block
    • dependent on representation details, e.g. table, linked list
    • we use: *self* to refer to the current process PCB
  – *state*: ready, waiting, ...
    • + perhaps the queue where the process is on
  – *scheduling queues*
    • queues where this process is part of
  – “...” includes
    • quota limits
    • accounting information
      • e.g. times in user, kernel mode
    • user identification
    • permissions

Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking
Typical memory layout for a process

• In user mode, the process can access just its private data and stack

• In kernel mode, the process can access kernel functions and data
  – it uses a new stack for that purpose, stack pointer loaded as part of the trap
  – the kernel runs, in fact, on behalf of the process
    • “the process runs in kernel mode”
  – this space is shared by all processes

• No other processes are visible
  – hence, in order to deliver data to another process, this data must be copied in between
  – the kernel cannot deliver data to just any process; it first must change the memory settings such that this process becomes visible

• Many system calls are just a trap
Context switch

process $P_0$  operating system  process $P_1$

interrupt or system call

save state into PCB$_0$

idle

reload state from PCB$_1$

executing

idle

interrupt or system call

save state into PCB$_1$

idle

reload state from PCB$_0$
Blocking events, queues
Link PCB at place of blocking

![Diagram showing the link PCB at place of blocking](image-url)
Example: process creation

• Executed
  – in response to an application call (e.g. \texttt{fork()} )

• Create new PCB, fresh process number, ...

• Fill in:
  – parent: \texttt{self} (caller)
  – initial CPU and memory state vector (defaults)
  – other control fields based on defaults and parent

• Add to ready queue

• Return status (process id) to caller
  – i.e., put this at the right place in memory

• Transfer control to scheduler
  – may need to reschedule, depending on priority
    • otherwise, just return, only mode switch needed
Example: process deletion

- Executed
  - in response to a call
    - from process itself (\textit{exit()} ) or from other one (kernel, or higher in hierarchy)

- Delete all descendents
  - may need to stop several processors!

- Return all resources
  - close open files
    - references to administration in other kernel datastructures
  - reclaim memory resources
  - \textit{if parent waiting}, move parent to ready queue
  - \textit{otherwise} store status and move to ‘zombie’ queue
  - notice: \textit{resources referenced through the private space of a process must be returned by the process itself (or: within the context of that process)}
    - we come back to this point later

- Transfer control to scheduler
  - the current process may have been removed
  - waiting processes (parent) may have become unblocked
Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing
Concurrent systems

- components in a PC

- processes in a multi-tasking environment

- threads within a process
Major issues in concurrency

- **co-operation:**
  - *sharing resources* (hardware like printer, scanner, disk, ..., or most likely: memory and processors)
  - *transfer information*: *synchronization* and *communication* (needs shared resources)

- **interference**, mutual influence (good or bad):

  Assumptions or knowledge that one process has about the state, are disturbed by actions of another process

- *good interference*: wait for another process to set a boolean to *true* indicating delivery of a value in a variable
  - \( x := \text{false}; \{ \neg x \} \text{while } \neg x \text{ do skip od}; \) “use \( y \)” \( || \) .... \( y := E; x := \text{true} ..... \)
  - Question: are there ‘tricky’ interleavings?

- *bad*: access a resource (e.g. a printer) after checking its availability; in between the check and the use the resource is accessed by another process
  - \( \text{if avail then} \{ \text{avail} \} \text{avail := false; “use resource”; avail := true; fi } || \) ...same
Looking forward: communication & synchronization primitives

**Synchronization**
- just shared variables, e.g., event flags
  - in combination with *busy waiting*, see previous examples
- semaphores, mutexes
- condition variables
- readers/writers locks
- monitors

**Communication**
- shared memory
- message passing (asynchronous, buffered or synchronous)
  - streaming: pipes, fifos, sockets
  - structured: message queues
  - has *implicit* synchronization (why?)
- remote procedure calls (combines functionality and messaging)
- signals
  - asynchronous messages, i.e., interrupt the flow of control
  - may regard the *signal handler* as a concurrent thread, that is (synchronously) waiting for an event
Shared memory between processes

- A memory segment is reserved in the kernel memory
- It is added to the space addressable in user mode
- Since it is in the shared space, it is available to other processes
- *No kernel involvement is required upon subsequent reading and writing*
POSIX: Shared Memory API (1003.1)

```c
int shmget(key_t key, size_t size, int shmflg);
/* key: an identifier, for reference */
    * size: in #bytes
    * shmflg: indicates how to open, e.g. readonly, create if does not exist
    * shmget() returns an id for manipulation */

void *shmat(int shmid, const void *shmaddr, int shmflg);
/* shmid: from shmget() */
    * shmaddr: place in process space where shared mem should be referenced
    * shmflg: detail and steer the operation
    * shmat() returns a pointer to a shared byte array of size bytes (see shmget()) */

int shmdt(const void *shmaddr);
/* shmaddr: shared memory to detach, obtained from shmat() */
    * shmdt returns just error condition */

int shmctl(int shmid, int cmd, struct shmid_ds *buf);
/* control operations, cmd = IPC_RMID removes the shared memory */
```
Example


- Server
  - creates shared memory
  - writes ‘ab...z’ into it
  - (busy) waits until ‘a’ is replaced by ‘*’

- Client
  - attaches to (not: creates) the created shared memory segment
  - reads and prints the content
  - puts a ‘*’ at the expected place
Server

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
define SHMSZ 27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* arbitrary name */
    /* Create the segment */
    if ((shmid = shmget(key, SHMSZ, 0666)) < 0) {
        perror("shmget"); exit(1);
    }

    /* Now we attach the segment to our data space */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1);
    }

    /* Now put some things into the memory for the
     * other process to read.
     */
    s = shm;
    for (c = 'a'; c <= 'z'; c++) *s++ = c;
    *s = NULL;

    /* wait until the other process changes the first
     * character of our memory to '*'
     */
    while (*shm != '*') /* sleep(1) */;

    exit(0);
}
Client

/* attach the segment to our data space */
if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat"); exit(1);
}

/* Now read what the server put in the memory. */
for (s = shm; *s != NULL; s++) putchar(*s);
putchar('n');

/* Change the first character of the segment to '*' */
*shm = '*';

exit(0);
• Notes:

- busy waiting (in this example as well as in the book) is not an acceptable synchronization method at the level of the OS
  - it uses the entire timeslice and destroys performance
  - the `sleep()` avoids continuous checking at the expense of a one second delay
- instead, synchronization methods provided by the OS should be used
  - message passing, or one of the methods discussed later
- since the shared memory is not removed it remains after termination of server and client
  - to remove: add `shmctl (shmid, IPC_RMID, NULL)`
- there is a newer version of these primitives that should be used
Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing
POSIX: pipes & Fifo’s (1003.1)

- **Pipe:**
  - connected pair of file(-descriptor)s
  - buffered, one-directional *streams* between parent and child
  - process
    - no message structure imposed

- **Fifo (“named pipe”):**
  - same as a pipe, but connection setup via name in kernel
  - create a unique name, e.g., “/tmp/fifo”, using
    - *mknfifo* (“/tmp/fifo”, *mode*);
  - Now use “/tmp/fifo” as a file: *open(), read(), write()*
  - Admits multiple writers
    - check the manual for the rules on atomicity of write operations
Pipes & Fifo’s (cnt’d)

- The pair of file descriptors has a read (0) and a write (1) side that are connected.
  - closing one of the two on both sides creates a channel

- Synchronization:
  - `read()` blocks when pipe (buffer) is empty
  - `write()` blocks when buffer is full
  - can choose also non-blocking behavior

```c
int fd[2];
pid_t child;

if (pipe (fd)<0) perror (“pipe”);  

child = fork();
if (child<0) perror (“fork”);

if (child != 0) /* parent: writer */  
    close (fd[0]);
    write (fd[1], ......);  
else{ /* child: reader */  
    close (fd[1]);
    read (fd[0], ......);  
}
```
Distributed channels

- The Berkeley Socket Interface
  - setup a connection using *ports* on both machines
    - setup: client (active, trying) and server (passive, waiting) roles
    - endpoints identified by (machine, port) pair
    - connections identified by two such pairs
  - communicate using regular read/write or send/receive operations, similar as file operations
  - .... discussed in detail in ‘computer networks’

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOCKET</td>
<td>Create a new communication end point</td>
</tr>
<tr>
<td>BIND</td>
<td>Attach a local address to a socket</td>
</tr>
<tr>
<td>LISTEN</td>
<td>Announce willingness to accept connections; give queue size</td>
</tr>
<tr>
<td>ACCEPT</td>
<td>Block the caller until a connection attempt arrives</td>
</tr>
<tr>
<td>CONNECT</td>
<td>Actively attempt to establish a connection</td>
</tr>
<tr>
<td>SEND</td>
<td>Send some data over the connection</td>
</tr>
<tr>
<td>RECEIVE</td>
<td>Receive some data from the connection</td>
</tr>
<tr>
<td>CLOSE</td>
<td>Release the connection</td>
</tr>
</tbody>
</table>
Usage of the interface (C binding)

- Note: send/receive is same as read/write
POSIX: message queues (1003.1b)

- A named priority queue of messages
  - *name*: system-wide
  - *priority*: support discrimination based on importance
  - *messages*: structure is maintained: a `receive()` is coupled with a `send()`

```c
struct mq_attr attr;
mqd_t mq;

attr.mq_maxmsg = 100; attr.mq_msgsize = 128;
attr.mq_flags = 0;
mq = mq_open(“/MyQ”, how, mode, &attr);

if (mq == (mqd_t)-1) perror (“mq_open”);

/* how: e.g. O_CREAT | O_RDWR
 * mode: e.g. S_IRUSR | S_IWUSR | S_IXUSR */
```
POSIX: message queues (cnt’d)

• Manipulation through
  – `mq_setattr()`, `mq_getattr()`
    • queue size, message size – set upon creation only
    • current number of messages, (un)blocking use of communication functions
  – `mq_close()`, `mq_unlink()`
    • unlinking possible only after closing by all partners

• Communication:
  – `status = mq_send(mq, “hello world”, 12, prio)`
    • `prio`: between 0 and MQ_PRIO_MAX (≥ 32)
  – `nbytes = mq_receive(mq, buf, maxsize, &prio)`
    • (oldest, highest prio) first
Message passing primitives properties

- **Source / destination**
  - send to / receive from process, or object
    - object: channel, mailbox

- **Synchronization**
  - blocking / non blocking
  - synchronous, buffered, asynchronous
    - respectively: send/receive simultaneously, excess of sends allowed, send never blocked

- **Delivery order**
  - fifo, arbitrary (receiver determined)

- **Numbers of sender / receiver**
  - point-to-point
  - multiple readers, writers
    - multicast: group of receivers
    - anycast: one receiver out of a group

- **Structure**
  - message oriented, byte oriented

**Examples**

- **Pipe, socket**
  - Object
    - Buffered, blocking
    - Fifo
    - Multiple writers
      - (to receiving side)
      - Byte oriented

- **Mailbox**
  - Object
  - Buffered, non-blocking
  - Receiver-determined order
  - Multiple readers, writers
  - Message oriented, usually