Multi-Process Programming in C

Advanced Operating Systems

Giuseppe Massari
giuseppe.massari@polimi.it

Outline

Multi-process programming

- Fork processes
- Inter-process synchronization
- Executing other programs

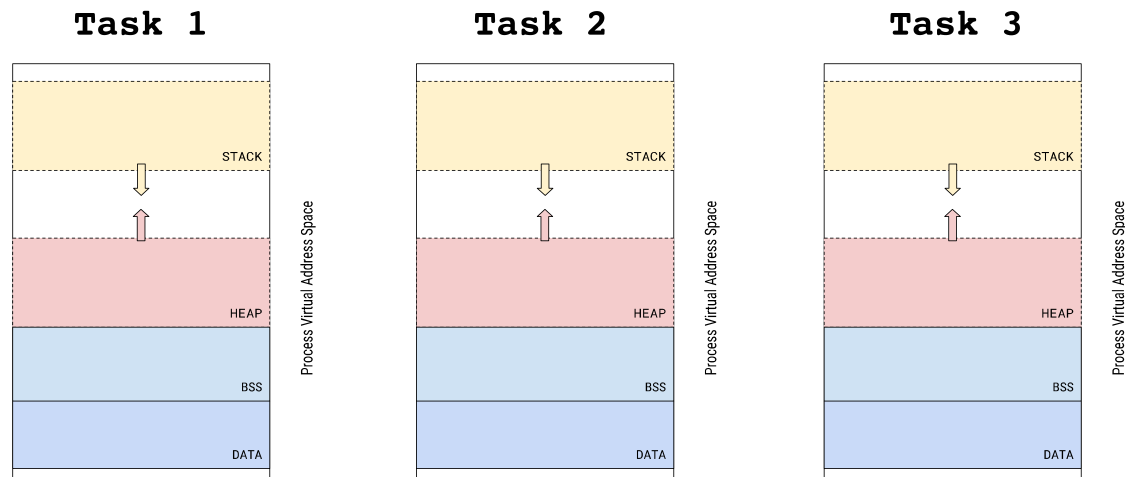
Inter-Process Communication

- Signals
- Pipes and FIFO
- Message queues
- Shared memory
- Synchronization

Multi-process programming

Why multi-process programming?

- Multi-process means that each task has its own *address space*
  - More *task isolation* and independence compared to multi-threading
  - Reliability → one process crashing does not affect the entire program
- Useful choice for multi-tasking application where tasks have significant requirements in terms of *resources*
  - Tasks requiring "long" processing times
  - Tasks processing big data structures



Multi-process programming

Example 1: *Forking a process*

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

int main () {
  pid_t child_pid;
  printf("Main process id = %d (parent PID = %d)\n",
    (int) getpid(), (int)  getppid());

  child_pid = fork();
  if (child_pid != 0)
        printf("Parent: child's process id = %d\n", child_pid);
  else
        printf("Child:  my process id = %d\n", (int) getpid());
  return 0;
}
```

- **fork()** creates a new process duplicating the calling process

Example 1: *Forking a process*

```
$ gcc example1.c -o fork_ex1
$ ./fork_ex1
```

```
Main process id = 9075 (parent PID = 32146)
Parent: child's process id = 9076
Child:  my process id = 9076
```

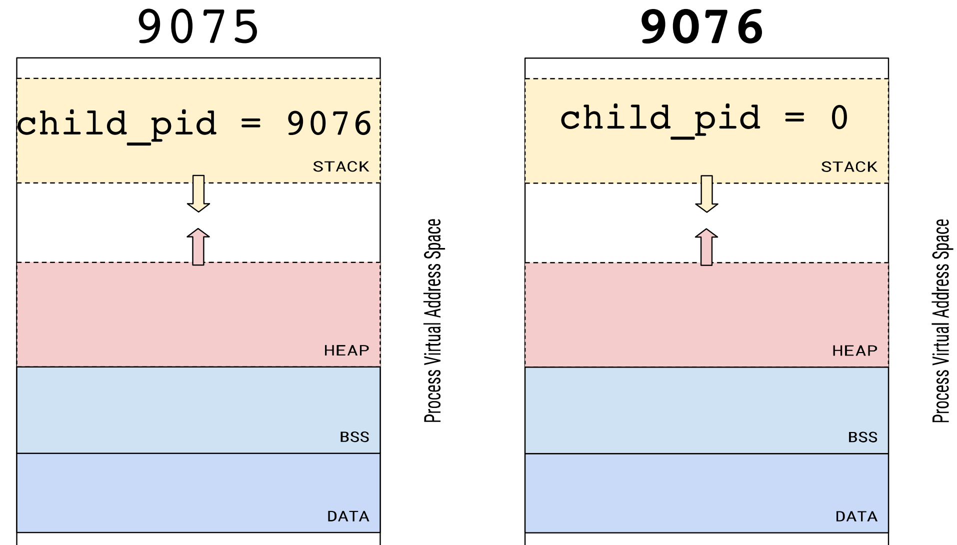
- The main process has PID = 9075. It's parent (PID=32146) is the shell (`echo $$`) from which the executable has been started
- After the **`fork()`** the program concurrently executes two processes
- The `child_pid` variable, in the address space of the *parent process*, is set to the return value of the fork (the child process ID)
- The `child_pid` variable, in the address space of the *child process*, is not set
- The **`getpid()`** returns the current process identifying number

Example 1: *Forking a process*

```
int main ()
{
    pid_t child_pid;
    …



    child_pid = fork();
    …
}
```



- Parent process virtual address space is replicated in the child
  - Including the states of variables, mutexes, condition variables, POSIX objects
- The child inherits copies of the parent's set of open file descriptors
  - As well as status flags and current file offset

Example 2a

- Two processes writing something to the standard output

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

void char_at_a_time( const char * str ) {
  while( *str!= '\0' ) {
    putchar( *str++ );   // Write a char and increment the pointer
    fflush( stdout );    // Print out immediately (no buffering)
    usleep(50);
  }
}
int main() {
  if( fork() == 0 )
    char_at_a_time( "............." );
  else
    char_at_a_time( "|||||||||||||" );
}
```

Example 2a

```
$ gcc forkme_sync1.cpp -o forkme
$ ./forkme

|.|.|.|.|.|.|..||..|.|.|.|
```

- Concurrency leads to unpredictable processes execution order
- The application might need to *synchronize* the execution of two or more processes
- The parent process might need to *wait for* a child process to finish
  - The parent process forks a child process to perform a computation, goes on in parallel, and then it reaches an execution point where it needs to use the output data of the child process
- Considering our example, assume this is the output we want:

```
.............|||||||||||||
```

Multi-process programming

Synchronization

- The parent process can block itself until a status change has occurred in one of the child processes
  - Child process terminated or stopped
  - Child process resumed by a *signal* (see later)
- The status is retrieved from an integer argument passed by pointer

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

  - *waitpid(…)* wait for a state change for a specific child process
  - *wait/waitpid(...)* allows the system to release the resources associated with the child process (e.g., opened files, allocated memory, etc...)

Example 2b: *Forking a process with synchronization*

- The results can be obtained by exploiting **`wait(...)`** functions

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>

void char_at_a_time( const char * str ) {
  while( *str!= '\0' ) {
    putchar( *str++ );    // Write a char and increment the pointer
    fflush( stdout );     // Print out immediately (no buffering)
    usleep(50);
  }
}
int main() {
  if( fork() == 0 )
    char_at_a_time( "..............." );
  else {
    wait( NULL );
    char_at_a_time( "|||||||||||||" );
}
```

Zombie processes

- If a child terminates, without **`wait()`** performed, it remains in a “zombie” state
- The Linux kernel maintains a minimal set of information about the zombie process
  - (PID, termination status, resource usage information, …)
  - Parent can later perform a wait to obtain information about children
- A zombie process consumes a slot in the *kernel process table*
  - If the table fills, it will not be possible to create further processes
- If a parent process terminates, then its "zombie" children (if any) are adopted by the ***init*** process
  - ***init*** automatically performs a wait to remove the zombies

Multi-process programming

Spawning executor processes

- Process forking basically “clones” the parent process image
  - Same code and same variables
- In a multi-process application we may need to spawn a process to execute a completely different task (program)
  - Load and run another executable
- The `exec()` family of functions allows us to start a program within another program
- The `exec()` family of functions replace the current process image with a new one coming from loading a new program

Multi-process programming

Spawning executor processes

```
#include <unistd.h>
int execl(const char *path, const char *arg, … );
int execlp(const char *file, const char *arg, … );
int execle(const char *path, const char *arg, … ,char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
```

- All the functions take the executable path as first argument
- "**l**" functions accept variable amount of null-terminated `char *`
- "**v**" functions accept the executable path and an array of null-terminated char *
  - Both forward arguments to the executable (arg0 must be set to executable name)
- "**p**" functions access PATH environment variable to find the executable
- "**e**" functions accept also an array of null-terminated char * storing environment variables

Example 3

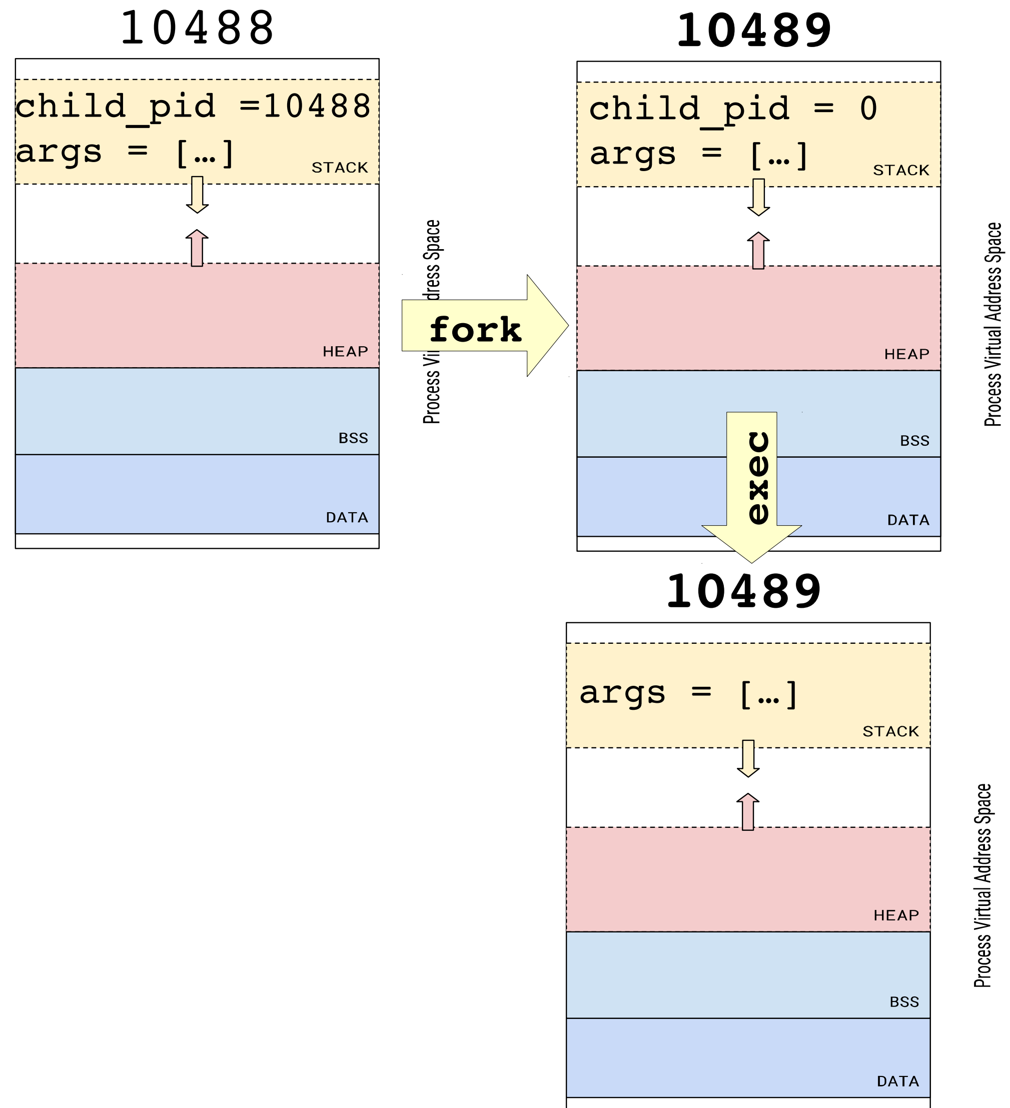
```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int spawn(const char * program, char ** arg_list) {
  pid_t child_pid = fork();
  if (child_pid != 0)
    return child_pid;       /* This is the parent process.  */
  else {
    execvp (program, arg_list);      /* Now execute PROGRAM */
    fprintf (stderr, "An error occurred in execvp\n");
    abort ();
  }
}
int main() {
  char * arg_list[] = { "ls", "-l", "/", NULL };
  spawn("ls", arg_list);
  printf ("Main program exiting...\n");
  return 0;
}
```

Spawning executor process

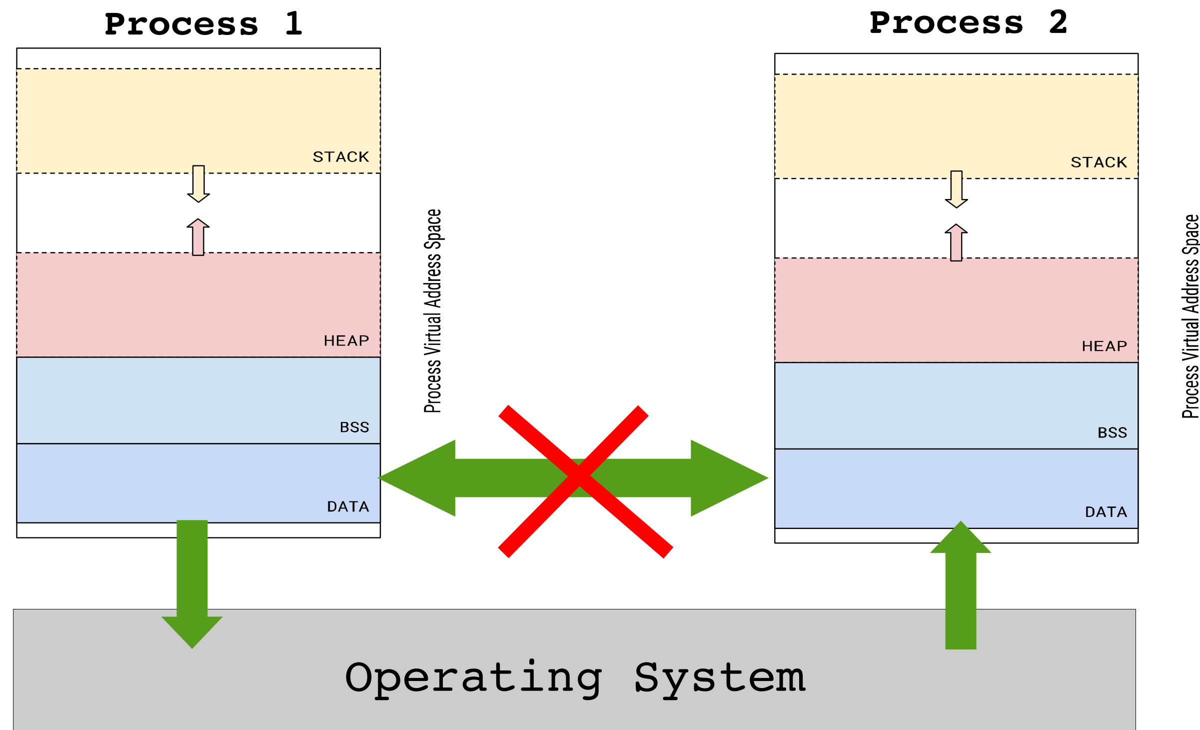
```
int main ()
{
    char * args[] = {
        “ls”, “-l”, NULL
    };
    pid_t child_pid;

    child_pid = fork();
    execvp(“ls”, arg);
    …
}
```



Overview

- Each process has its own address space → How can we exchange information between different processes?
- Operating systems provide system calls on top of which communication mechanisms and API are built



POSIX vs System V

- Both provides the same set of mechanisms
- POSIX born after System V for standardization purposes
- POSIX aimed at simplify and improve System V
- POSIX functions are thread safe
- *The code in the next slides is based on POSIX IPC*

Signals

Characteristics

- A single bit length “message”
- No data exchange
- Information content implicitly provided by the signal type
- Mechanism for asynchronous event notification

Examples

- Elapsed timer
- I/O operation completion
- Program exceptions
- User-defined events

Synchronization

- Asynchronous interaction between a sender and a receiver

Overview

- A signal is a notification of an event
  - OS defines a set of macros bound to signal numbers
- Processes can receive signals to *asynchronously* react to requests from software or unexpected hardware events
  - Other processes calling a function like *kill()*
  - The process itself calling a function like *abort()*
  - A child process exiting (SIGCHLD)
  - User interrupts program from the keyboard (SIGINT)
  - Program behaving incorrectly (SIGILL, SIGFPE, SIGSEGV)
  - Program accessing unavailable mapped memory (SIGSEV)
  - Program sends data via *write()* and nobody receives it (SIGPIPE)
  - ...

Most common POSIX signals

POSIX signals	Number	Default action	Description
SIGHUP	1	Terminate	Controlling terminal disconnected
SIGINT	2	Terminate	Terminal interrupt
SIGILL	4	Terminate and dump	Attempt to execute illegal instruction
SIGABRT	6	Terminate	Process abort signal
SIGKILL	9	Terminate	Kill the process
SIGSEGV	11	Terminate and dump	Invalid memory reference
SIGSYS	12	Terminate and dump	Invalid system call
SIGPIPE	13	Terminate	Write on a pipe with no one to read it
SIGTERM	15	Terminate	Process terminated
SIGUSR1	16	Terminate	User-defined signal 1
SIGUSR2	17	Terminate	User-defined signal 2
SIGCHLD	18	Ignore	Child process terminated, stopped or continued
SIGSTOP	23	Suspend	Process stopped

Signals

Signal handler registration

- OS manages a *signal vector table* for each process
- A process can register a custom signal han*dler* for each signal
  - In Linux OS, the default handler behaviour is in the most of the cases to *terminate* the process
- **sigaction(…)** function

```
#include <signal.h>
int sigaction(
    int signum, const struct sigaction *act, struct sigaction *oldact);
```

  - *signum*: number of signal to handle
  - *act*: new settings to apply to register a handler function
  - *oldact*: previous settings
    - set to NULL if you are not interested in saving them

Signal handler registration

- **struct sigaction**

```
struct sigaction {
        void     (*sa_handler)(int);
        void     (*sa_sigaction)(int, siginfo_t *, void *);
        sigset_t   sa_mask;
        int        sa_flags;
        void     (*sa_restorer)(void);
};
```

- *sa_handler*: pointer to your handler function
- *sa_sigaction*: alternative handler function with two additional arguments
  - Provides more information about the received signal...
- *sa_mask*: set signals to be blocked during the execution of the handler
- *sa_flags*: allow to modify the behavior of the signal handling process
  - set to SA_SIGINFO in order to use *sa_sigaction* as handler

Example 4a: *User-defined signal handling*

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
sig_atomic_t sigusr1_count = 0;  // Use atomic variable for safety

void handler(int sig_num) {
  ++sigusr1_count;
}

int main() {
  struct sigaction sa;
  memset(&sa, 0, sizeof(sa));
  sa.sa_handler = &handler;
  sigaction (SIGUSR1, &sa, NULL);
  fprintf(stderr, "Running process... (PID=%d)\n", (int) getpid());
  /* ...  */
  getchar();
  printf("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

Signals

Example 4a: *User-defined signal handling*

- Include `<signal.h>` header file
- Declare a data structure of type **`sigaction`**
- Clear the **`sigaction`** data structure and then set **`sa_handler`** field to point to the **`handler()`** function
- Register the signal handler for signal `SIGUSR1` by calling the **`sigaction()`** function

```
$ gcc example4.cpp -o sig_example
$ ./sig_example
Running process... (PID=16151)
```

```
$ kill -SIGUSR1 16151
```

```
SIGUSR1 was raised 1 times
```

Sending a signal

- Functions **kill(..)** and **sigqueue(..)**

```
#include <signal.h>
#include <sys/types.h>
int kill(pid_t pid, int sig);
```

  - *pid*: Target/destination process ID
  - *sig*: Signal number

```
#include <signal.h>
int sigqueue(pid_t pid, int sig, const union sigval value);
```

  - *value*: Data item to append

```
union sigval {
    int     sival_int;    // Integer value
    void   *sival_ptr;    // Pointer to other type value
};
```

Example 4b: *Signal sender program (SIGUSR1)*

- A SIGUSR1 signal is sent to a target process, appending an integer data

```
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
    pid_t target_pid = atoi(argv[1]);
    const union sigval value = { atoi(argv[2]) };

    printf("Sending USR1 to process %d...\n", target_pid);
    int ret = sigqueue(target_pid, SIGUSR1, value);
    if (ret != 0) {
        perror("sigqueue: ");
        exit(1);
    }
    return 0;
}
```

Signal information

- A data structure is defined to be passed to the signal handler when the SA_SIGINFO flag is specified

```
struct siginfo_t {
    int       si_signo;     /* Signal number */
    int       si_errno;     /* An errno value */
    int       si_code;      /* Signal code */
    int       si_trapno;    /* Trap number that caused
                               hardware-generated signal
                               (unused on most architectures) */
    pid_t     si_pid;       /* Sending process ID */
    uid_t     si_uid;       /* Real user ID of sending process */
    int       si_status;    /* Exit value or signal */
    ...
};
```

Example 4c: *Accessing signal information*

```
...
void handler2(int sig_num, siginfo_t * info, void * extra) {
    printf(" Signal number:   %d\n", info->sig_no);
    printf(" Signal code:     %d\n", info->sig_code);
    printf(" Sender process:  %d\n", info->si_pid);
    printf(" Sender user ID:  %d\n", info->si_uid);
    printf(" Signal value:    %d\n", info->si_value.sival_int);
}

int main(){
  struct sigaction sa;
  memset(&sa, 0, sizeof(sa));
  sa.sa_handler = &handler2;
  sa.sa_flags   = SA_SIGINFO;
  sigaction (SIGUSR1, &sa, NULL);
  fprintf(stderr, "Running process... (PID=%d)\n", (int) getpid());
  /* ...  */
  getchar();
  printf("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

Blocking signals

- Signals can be *blocked* at process-level or thread-level
  - Apart from SIGKILL and SIGSTOP (attempts are ignored)
  - Enqueued and managed later, when the process/thread “unmask” the signal

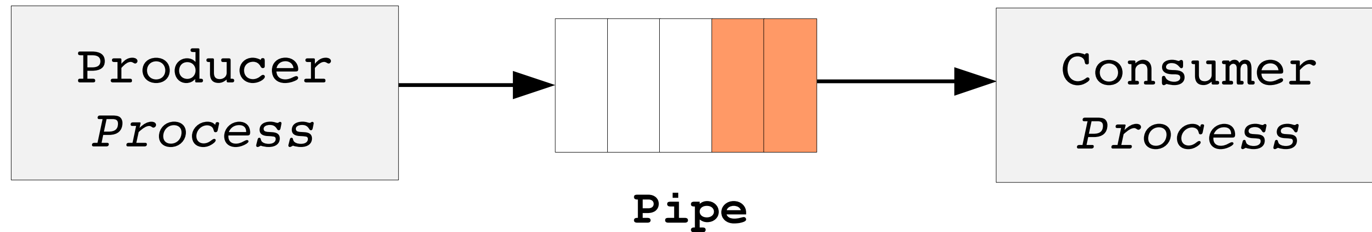
```
#include <signal.h>
...

int main() {
     sigset_t curr, old;
     sigemptyset(&set);
     sigaddset(&set, SIGINT);
     int ret = sigprocmask(SIG_BLOCK, &set, &old); // set new mask
     …
     ret = sigprocmask(SIG_UNBLOCK, &set, NULL);
     // alternative: sigprocmask(SIG_SETMASK, &old, NULL);
     return 0;
}
```

  - For threads use function *pthread_sigmask()* from POSIX thread library

Unnamed pipes

- Based on the producer/consumer pattern
  - A producer write, a consumer read
- Data are written/read in a First-In First-Out (FIFO) fashion



- In Linux, the operating system guarantees that only one process per time can access the pipe
- Data written by the producer (sender) are stored into a buffer by the operating system until a consumer (receiver) read it

Unnamed pipes

- POSIX provide function call *pipe(…)* to create a unidirectional FIFO communication channel between related processes

```
#include <unistd.h>
int pipe(int pipefd[2]);
#include <fcntl.h>              /* Obtain O_* constant definitions */
#include <unistd.h>
int pipe2(int pipefd[2], int flags);
```

  - *pipefd*: array of integers to be filled with two file descriptors
    - pipefd[0] : read end of the pipe
    - pipefd[1] : write end of the pipe
  - *flags*: if =0 same of pipe()
    - O_CLOEXEC – close file descriptors in case of *exec(…)* call
    - O_DIRECT – perform I/O in “packet” mode
    - O_NONBLOCK – avoid blocking read/write in case of empty/full pipe
  - We can then use common functions to access files

Example 5: *Simple unnamed pipe based messaging (1/2)*

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>

/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each.  */
void writer(const char * message, int count, FILE * stream) {
  for(; count > 0; --count) {
    fprintf(stream, "%s\n", message);
    fflush(stream);
    sleep(1);
  }
}
void reader(FILE * stream) {
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while(!feof(stream) && !ferror(stream)
        && fgets(buffer, sizeof(buffer), stream) != NULL)
    fputs(buffer, stdout);
}
```

Example 5: *Simple unnamed pipe based messaging (2/2)*

```
int main () {
  FILE * stream;
  /* Create pipe place the two ends pipe file descriptors in fds */
  int fds[2];

  pipe(fds);
  pid_t pid = fork();
  if(pid == (pid_t) 0) {     /* Child process (consumer) */
    close(fds[1]);            /* Close the copy of the fds write end */
    stream = fdopen(fds[0], "r");
    reader(stream);
    close(fds[0]);
  }
  else {                     /* Parent process (producer)  */
    close(fds[0]);           /* Close the copy of the fds read end */
    stream = fdopen(fds[1], "w");
    writer("Hello, world.", 3, stream);
    close(fds[1]);
  }
  return 0;
}
```

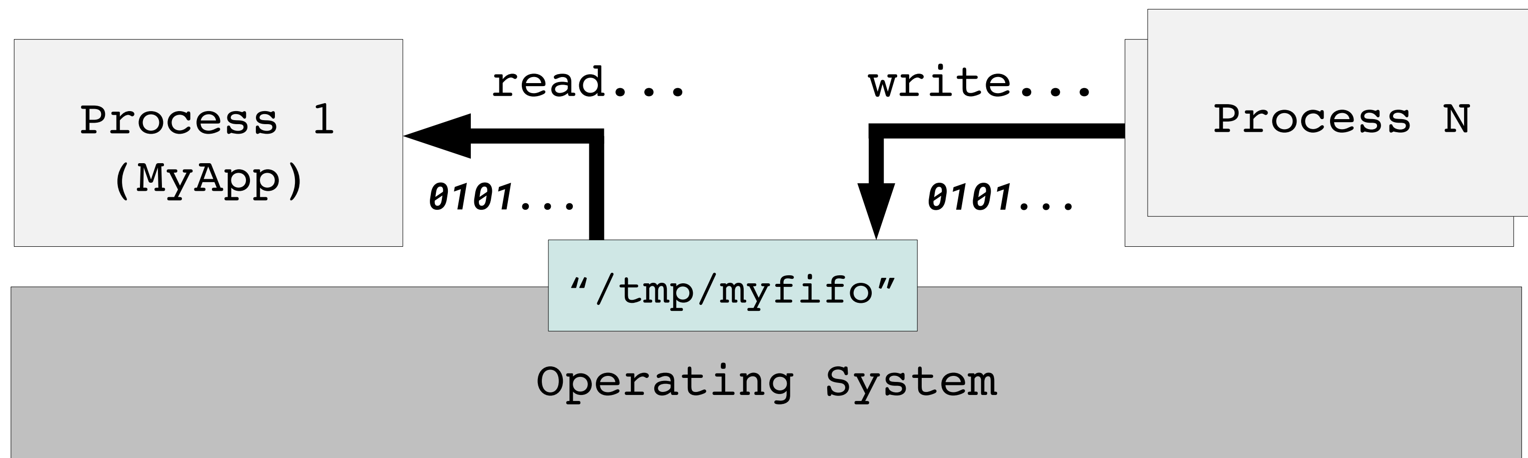
Example 5: *Simple unnamed pipe based messaging*

- Create a pipe with **`pipe()`** call and initialize the array of file descriptors "`fds`"
- Fork a child process that will behave as consumer
  - Close the write end of the pipe file descriptors array
  - Open the read end of the pipe file descriptors array
  - Call the `reader()` function to read data from the pipe
- Parent process acts as producer
  - Close the read end of the pipe file descriptors array
  - Open the write end of the pipe file descriptors array
  - Call the `writer()` function to write 3 times "*Hello, world.*"

```
Hello, world.
Hello, world.
Hello, world.
```

Named pipes (FIFO)

- Same behaviour of unnamed pipes but for *unrelated processes* communication
- Pipe-based mechanism accessible through file-system
- The pipe appears as a special FIFO file
- The pipe must be opened on both ends (reading and writing)
- OS passes data between processes without performing real I/O



Named pipes (FIFO)

- POSIX provides function mkfifo() to create a named pipe accessible via filesystem as a special file

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

int mkfifo(const char *pathname, mode_t mode);
```

  - *pathname*: path of the special file created
  - *mode*: R/W/X permissions (.e.g. S_IRWXU, S_IRUSR, S_IWUSR,...)
    - More info in the man page: `$ man 2 open`
- Once the FIFO has been created we can open and access it as a normal file
  - It has to be open at both ends before proceeding with I/O operations
    - Opening a FIFO for reading normally blocks until some other process opens the same FIFO for writing, and vice versa

Example 6a: *External interfacing through named pipe*

- *fifo_writer.c*

```
int main () {
  struct datatype data;
  char * myfifo = "/tmp/myfifo";
  if (mkfifo(myfifo, S_IRUSR | S_IWUSR) != 0)
      perror("Cannot create fifo. Already existing?");

  int fd = open(myfifo, O_RDWR);
  if (fd == 0) {
      perror("Cannot open fifo");
      unlink(myfifo);
      exit(1);
  }
  int nb = write(fd, &data, sizeof(struct datatype));
  if (nb == 0)
      fprintf(stderr, “Write error\n”);

  close(fd);
  unlink(myfifo);
  return 0;
}
```

Example 6a: *External interfacing through named pipe*

- *fifo_reader.c*

```
int main () {
  struct datatype data;
  char * myfifo = "/tmp/myfifo";

  int fd = open(myfifo, O_RDONLY);
  if (fd == 0) {
      perror("Cannot open fifo");
      unlink(myfifo);
      exit(1);
  }

  read(fd, &data, sizeof(struct datatype));
  …

  close(fd);
  unlink(myfifo);
  return 0;
}
```

Example 6a: *External interfacing through named pipe*

- The writer
  - Creates the named pipe (`mkfifo`)
  - Open the named pipe as a normal file in read/write mode (`open`)
  - Write as many bytes as the size of the data structure
    - The reader must be in execution (otherwise data are sent to no nobody)
  - Close the file (`close`) and then release the named pipe (`unlink`)

- The reader
  - Open the named pipe as a normal file in read only mode (`open`)
  - The read() function blocks waiting for bytes coming from the writer process
  - Close the file (`close`) and then release the named pipe (`unlink`)

Example 6b: *External interfacing through named pipe*

- *message-reader.c*
  - message-writer: the user sends char strings from the shell

```
int main () {
  char data = ' ';
  char * myfifo = "/tmp/myfifo";

  int fd = open(myfifo, O_RDWR);
  if (fd == 0) {
      perror("Cannot open fifo");
      unlink(myfifo);
      exit(1);
  }
  while (data != '#') {
    while (read(fd, &data, 1) && (data != '#'))
        fprintf(stderr, "%c", data);
  }
  close(fd);
  unlink(myfifo);
  return 0;
}
```

Example 6: *External interfacing through named pipe*

```
$ gcc example7.cpp -o ex_npipe
$ ./ex_npipe
Hello!
My name is
Joe
Communication closed
```

```
$ echo "Hello!" > /tmp/myfifo
$ echo "My name is" > /tmp/myfifo
$ echo "Joe" > /tmp/myfifo
$ echo "#" > /tmp/myfifo
```

- The (a priori known) named pipe location is opened as a regular file (`open`) to read and write
  - Write permission is required to flush data from pipe as they are read
- Blocking `read()` calls are performed to fetching data from the pipe
- The length of the text string not known a priori
  - `'#'` is used as special END character
- Close (`close`) and release the pipe (`unlink`) when terminate

Pipes and FIFO

Pros

- Low overhead
- Simplicity
- Mutual access solved in kernel-space

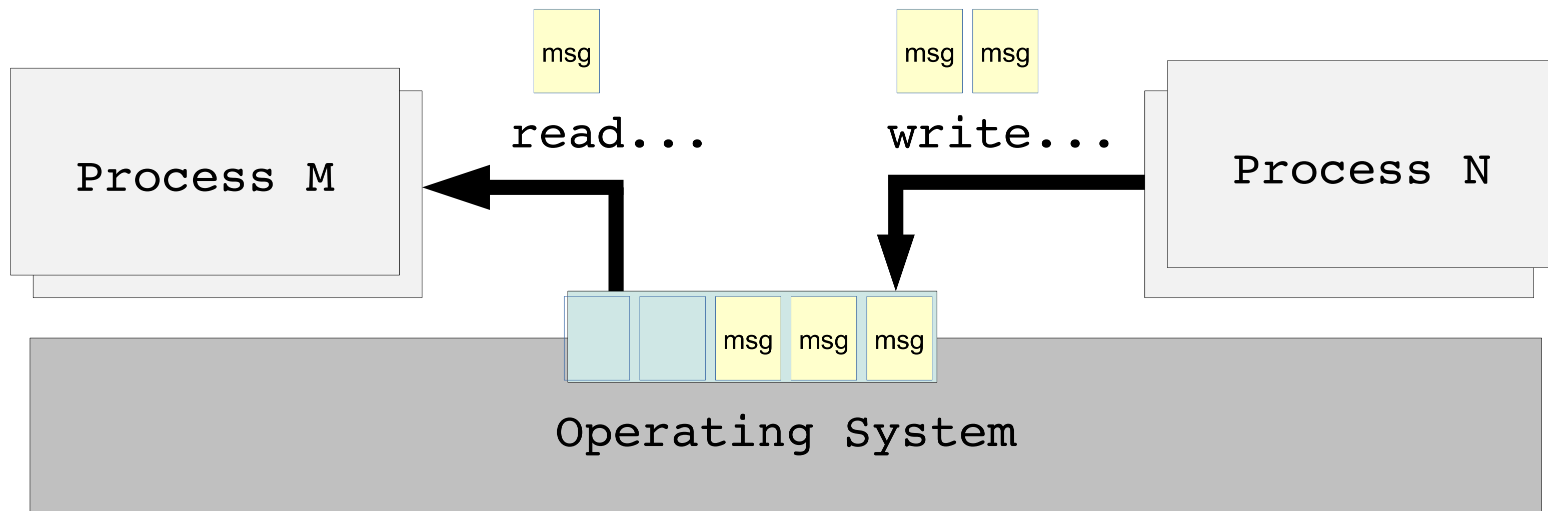
Cons

- No broadcast
- Unidirectional
- No message boundaries, data are managed as a stream
- Poor scalability

Message Queues

POSIX Message Queue

- Suitable for multiple readers and multiple writers
- Priority driven data exchange
- Message queue and message size are managed by the programmer
- The status of the message queue can be observable
- Link to POSIX real-time extension library to build (gcc … **-lrt**)



Message queue lifecycle

```
#include <fcntl.h>           /* For O_* constants */
#include <sys/stat.h>        /* For mode constants */
#include <mqueue.h>
int mq_open(const char *name, int oflag, mode_t mode,
        struct mq_attr *attr);
```

- *name* – the POSIX object name
- *oflag* – Opening flag (O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_EXCL, ...)
- *mode* – R/W/X permissions (.e.g. S_IRWXU, S_IRUSR, S_IWUSR,...)
- *attr* – set of attributes (e.g., message size, queue length, etc...)
- The function returns a message queue descriptor

```
int mq_close(mqd_t mqdes);
```

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

- *mq_close* closes the queue referenced by the descriptor returned by mq_open()
- *mq_link* removes the message queue and destroy it when closed by all the processes

Message queue attributes

- Attributes data structure

```
struct mq_attr {
    long mq_flags;        /* Flags: 0 or O_NONBLOCK */
    long mq_maxmsg;       /* Max nr of messages on queue */
    long mq_msgsize;      /* Max message size (bytes) */
    long mq_curmsgs;      /* Nr of messages currently in queue */
 };
```

- Function for manipulation

```
#include <mqueue.h>
int mq_getattr(mqd_t mqdes, struct mq_attr *attr);
int mq_setattr(mqd_t mqdes, struct mq_attr *newattr,
    struct mq_attr *oldattr);
```

  - *mqdes* – message queue descriptor
  - *newattr* – new set of attributes
  - *oldattr* – previous set of attributes

Message queue input/output

- Sending a message

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

  - *mqdes* – message queue descriptor
  - *msg_ptr* – pointer to the message to send
  - *msg_len* – message length
  - *msg_prio* – non-negative priority value (in POSIX: 0..31)

- Messages are queued in decreasing order of priority
  - Same priority messages → newer messages placed after older messages
- Full queue → the function call blocks
  - If O_NONBLOCK is specified as flag in the attributes it returns an error

Message queue input/output

- Receiving a message

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

  - *mqdes* – message queue descriptor
  - *msg_ptr* – message buffer to fill
  - *msg_len* – message buffer length
  - *msg_prio* – non-negative priority value (in POSIX: 0..31) associated to the received message
  - Returns the number of bytes in the received message, -1 in case of error

- Function call blocks until a message in available on the queue
  - If O_NONBLOCK is specified as flag in the attributes it returns an error

Message Queues

Pros

- Simplicity of the API
- Message packing
- Data can have different priorities
- Mutual access solved in kernel-space
- Can exploit notification mechanisms (not seen)

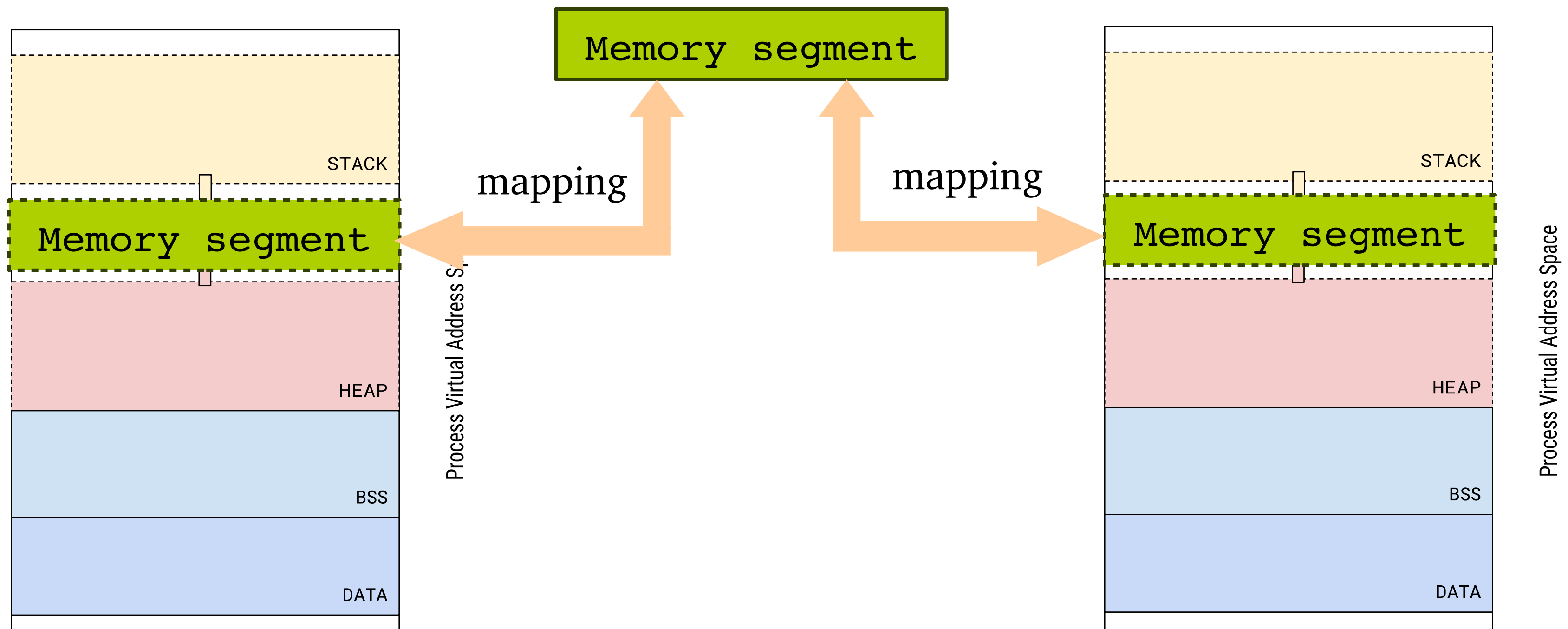
Cons

- Lower performance with respect to pipes
  - Sending a message actually implies writing into a file
- Unidirectional

Shared memory

Memory mapping

- Shared memory in Linux/UNIX OS is based on memory mapping
- A memory segment can be mapped in the address space of multiple processes
- POSIX implementation: link to real-time extension (gcc … **-lrt**)



Memory mapping

- Opening/creation of a shared memory object referenced by a name
  - A special file appears in "*/dev/shm/<name>*" with the provided name
    - This is a POSIX object, an handle that can be used by unrelated processes

```
#include <sys/mman.h>
#include <sys/stat.h>        /* For mode constants */
#include <fcntl.h>           /* For O_* constants */
int shm_open(const char *name, int oflag, mode_t mode);
```

  - *name* – the POSIX object name
  - *oflag* – Opening flag (O_RDONLY, O_RDWR, O_CREAT, O_EXCL, O_TRUNC)
  - *mode*: R/W/X permissions (.e.g. S_IRWXU, S_IRUSR, S_IWUSR,…)
  - The function returns a file descriptor

Memory mapping

- Once we allocate a memory object, before actually allocating the memory region, we must specify the size of the special file

```
#include <unistd.h>
#include <sys/types.h>          /* For mode constants */

int truncate(const char *path, off_t length);
int ftruncate(int fd, off_t length);
```

  - *path/fd* – File path or file descriptor
  - *off_t* – Opening flag (O_RDONLY, O_RDWR, O_CREAT, O_EXCL, O_TRUNC)

- Since the *shm_open(…)* returns a file descriptor, we will use *ftruncate(...)*

Memory mapping

- New mapping in the (virtual) address space of the calling process

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

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

- *addr* – Starting address. If NULL the Linux kernel will choose
- *length* –The mapped content is initialized using *length* bytes, in case of “file mapping”, it starts from *offset* in the file referenced by *fd*
- *prot* – Memory protection of the mapping, i.e., what the process can do
  - PROT_EXEC, PROT_READ, PROT_WRITE, PROT_NONE
- *flags* – Visibility of the updates coming from other processes
  - MAP_SHARED : The mapped memory is shared. Updates are visible to other processes, and carried through the underlying file
  - MAP_PRIVATE: Private copy-on-write. Updates not visibile to others and not carried through the underlying file
- It returns a pointer to the mapped area

Shared memory

Memory mapping

- Unmapping the region

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

  - Unmap all the memory pages in the give range
    - *addr* must be a multiple of the memory page size (in Linux often 4K)
  - Once performed, subsequent accesses to the region will generate a SIGSEV

- Release of the shared memory object

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

  - It removes the POSIX object
  - Once all processes have unmapped the object, de-allocates and destroys the contents of the associated memory area

Example 7: *Simple shared memory mapping*

- posix-shm-server.c (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main (int argc, char *argv[]) {
    const char * shm_name  = "/AOS";
    const int SIZE = 4096;
    const char * message[] = {"This ","is ","about ","shared ","memory"};
    int i, shm_fd;
    void * ptr;
    shm_fd = shm_open(shm_name, O_CREAT | O_RDWR, 0666);
    if (shm_fd == -1) {
        printf("Shared memory segment failed\n");
        exit(-1);
    }
…
```

Example 7: *Simple shared memory mapping*

- posix-shm-server.c (2/2)

```
    ftruncate(shm_fd, sizeof(message));
    ptr = mmap(NULL, SIZE, PROT_READ | PROT_WRITE, MAP_SHARED,shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        return -1;
    }
    /* Write into the memory segment */
    for (i = 0; i < strlen(*message); ++i) {
        sprintf(ptr, "%s", message[i]);
        ptr += strlen(message[i]);
    }
    mummap(ptr, SIZE);
    return 0;
}
```

- The server creates a shared memory referenced by “/AOS”
- The server writes some data (a string) into the memory segment
- The pointer ptr is incremented after each char string writing

Example 7: *Simple shared memory mapping*

- posix-shm-client.c (1/2)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main (int argc, char *argv[]) {
    const char * shm_name  = "/AOS";
    const int SIZE = 4096;
    int i, shm_fd;
    void * ptr;

    shm_fd = shm_open(shm_name, O_RDONLY, 0666);
    if (shm_fd == -1) {
        printf("Shared memory segment failed\n");
        exit(-1);
    }
```

Example 7: *Simple shared memory mapping*

- posix-shm-client.c (2/2)

```
…   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);
    if (ptr == MAP_FAILED) {
        printf("Map failed\n");
        return -1;
    }
    printf("%s", (char *) ptr);

    if (shm_unlink(shm_name) == -1) {
        printf("Error removing %s\n", shm_name);
        exit(-1);
    }
    return 0;
}
```

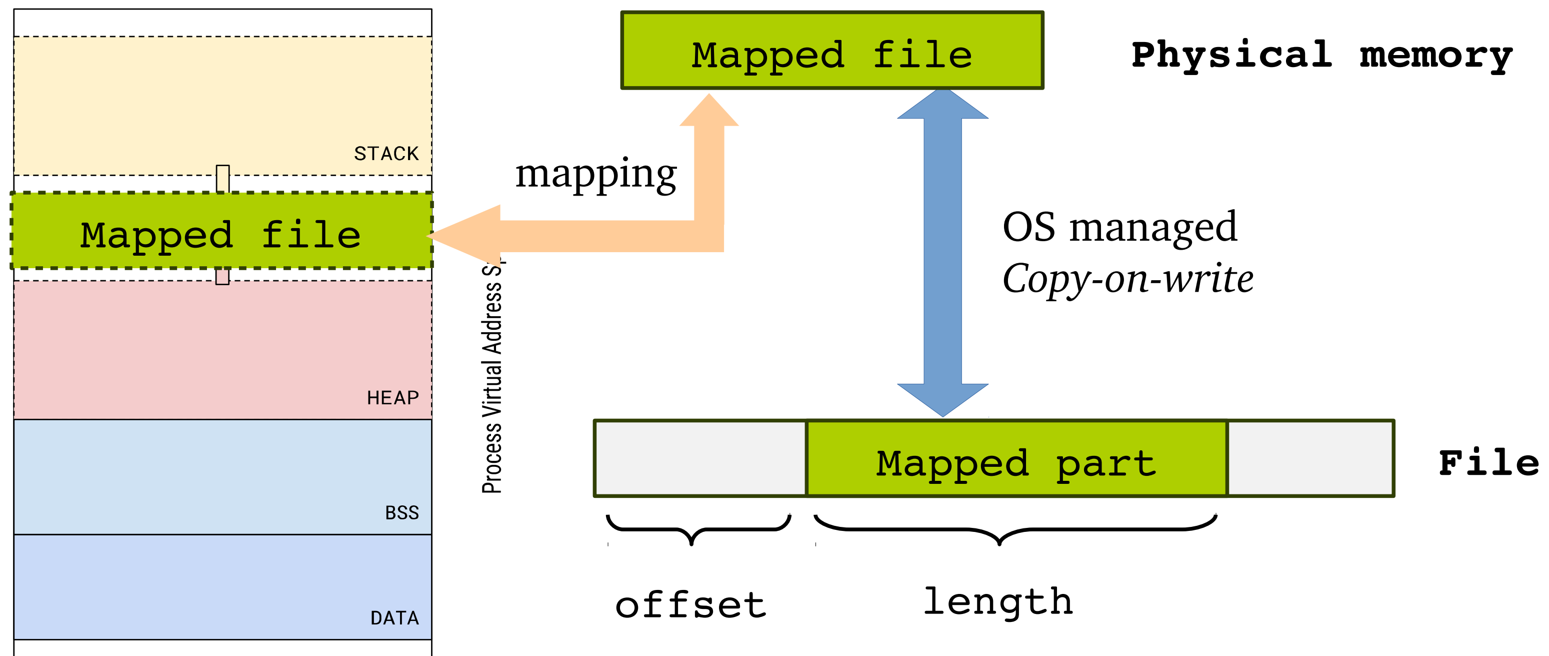
- The client opens the memory segment “AOS” in read-only mode
- The client maps the memory segment in read-only mode
- The client write the memory segment content to console

Example 7: *Simple shared memory mapping*

```
$ gcc posix-shm-server.c -o shm_server -lrt
$ gcc posix-shm-client.c -o shm_client -lrt
$ ./shm_server
$ ./shm_client
This is about shared memory
```

Memory mapping file

- Memory mapping allows us to logically insert part or all of a named *binary* file into a process address space



Example 8: *Simple I/O mapping*

- The file, passed at command-line (`argv[1]`), is opened and then memory mapped using the `mmap()` system call
- **mmap()** needs the address, the region size (file length), the permissions, the scope flags, file descriptor and offset

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

int main (int argc, char *argv[]) {
    int * p;
    int fd = open(argv[1], O_RDWR);
    p = mmap(0, sizeof(int), PROT_READ | PROT_WRITE , MAP_SHARED, fd, 0);
    (*p)++;
    munmap(p, sizeof(int));
    close (fd);
    return 0;
}
```

Shared memory

Pros

- Can reduce memory usage

  A big data structure can be mapped and shared to provide the same input set to multiple processes

- I/O mapping can be very efficient
  - Memory accesses instead of I/O read/write
  - Memory pages written back by OS only if the content has been modified
  - Seeking into the file performed with pointer arithmetic instead of "`lseek`"

Cons

- Linux map memory with a granularity of *memory page size*
  - Linux memory page size is typically 4 KB
  - Use memory mapping to map big files or share big data structures
- Can lead to *memory fragmentation*
  - Especially on 32-bits architectures
- Multiple small mappings can weight in terms of OS overhead

Semaphores

- Concurrency in multi-tasking applications may introduce race conditions → we need to protect shared resource
- Semaphores are usually system objects managed by the OS kernel
- Semaphores act as counters that we can manipulate by performing two actions: *increment* (wait) and *decrement* (post)
- If counter value > 0, *wait* decrements the counter and allows the task to enter the critical section
- If counter value = 0, *wait* blocks the tasks in a waiting list
- *post* increments the counter value
  - If the previous value was 0, a task is woken up from the waiting list
- Binary semaphores (value = 0 or 1) implements the same behaviour of mutex
- Not binary semaphores (value 0..n) are suitable for protecting access to pool of resources

POSIX semaphores

- *Named* and *unnamed* variety for using with unrelated and related processes
- Used as synchronization mechanims for both multi-process and multithreaded programs
- Link to POSIX threads extension library to build (gcc … **-pthread**)

Named semaphore opening/creation and release

- Function *sem_open(...)*

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>
sem_t * sem_open(const char *name, int oflags);
sem_t * sem_open(const char *name, int oflags, mode_t mode,
    unsigned int value);
```

  - *name* – the POSIX (semaphore) object name (under “*/dev/shm/sem.<name>*”)
  - *oflag* – Opening flag (O_CREAT, O_EXCL)
  - *mode* – R/W/X permissions (.e.g. S_IRWXU, S_IRUSR, S_IWUSR,…)
  - *value* – initial value
  - It returns the address of the new semaphore or SEM_FAILED in case of error
- Release of the named semaphore

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

Unnamed semaphore initialization and destruction

- Function *sem_init(...)*

```
#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

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

- This function can be used for multi-process or multi-threaded applications
  - *sem* – the semaphore data structure to initialize
  - *pshared* – if 0 shared among threads, if NOT 0 shared among processes
  - *value* – initial value
  - The function returns 0 for success, -1 for error
- An initialized unnamed semaphore can then be destroyed when done

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

Locking and unlocking functions

- Decrement/lock functions

```
#include <semaphore.h>
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *timeout);
```

  - *sem* – the semaphore data structure to decrement (lock)
  - *timeout* – set the limit of time that the call should block in case of value == 0
- *try_wait(...)* returns an error instead of blocking in case value == 0

- Increment/unlock function

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

- All functions return 0 for success and -1 for error

Example 9: *Using semaphores with shared memory*

- *posix-shm-sem-writer.c (1/2)*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>

#define SHMOBJ_PATH  "/shm_AOS"
#define SEM_PATH     "/sem_AOS"

struct shared_data {
    char var1[10];
    int var2;
};

int main(int argc, char *argv[]) {
    int shared_seg_size = (1 * sizeof(struct shared_data));
…
```

Example 9: *Using semaphores with shared memory*

- *posix-shm-sem-writer.c (2/2)*

```
    int shmfd  = shm_open(SHMOBJ_PATH, O_CREAT | O_RDWR, S_IRWU | S_IRWG);
    ftruncate(shmfd, shared_seg_size);
    struct shared_data * shared_msg = (struct shared_data *)
        mmap(NULL, shared_seg_size, PROT_READ | PROT_WRITE, MAP_SHARED,
        shmfd, 0);

    sem_t * sem_id = sem_open(SEM_PATH, O_CREAT, S_IRUSR | S_IWUSR, 1);
    struct shared_data out_msg = { “John”, 23 };

    sem_wait(sem_id);
    /* Update shared data */
    memcpy(shared_msg, &out_msg, sizeof(struct shared_data);
    sem_post(sem_id);

    shm_unlink(SHMOBJ_PATH);
    sem_unlink(SEM_PATH);
    return 0;
}
```

Example 9: *Using named semaphores with shared memory*

- The writer process
  - Maps a memory region
  - Creates a named semaphore and initialize it to 1 (`sem_open`)
  - Decrements the semaphore counter acquiring an exclusive access to the shared memory region (`sem_wait`)
  - Write into the memory region (`memcpy`)
  - Decrements the semaphore counter and releases the access to the memory region (`sem_post`)
  - Releases the shared memory region (`shm_unlink`)
  - Close and release the semaphore object (`sem_unlink`)

Example 9: *Using named semaphores with shared memory*

- *posix-shm-sem-reader.c (1/2)*

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <semaphore.h>

#define SHMOBJ_PATH  "/shm_AOS"
#define SEM_PATH     "/sem_AOS"

struct shared_data {
    char var1[10];
    int var2;
};

int main(int argc, char *argv[]) {
    int shared_seg_size = (1 * sizeof(struct shared_data));
…
```

Example 9: *Using named semaphores with shared memory*

- *posix-shm-sem-reader.c (2/2)*

```
    int shmfd  = shm_open(SHMOBJ_PATH, O_RDONLY, 0666);

    struct shared_data * shared_msg = (struct shared_data *)
        mmap(NULL, shared_seg_size, PROT_READ, MAP_SHARED, shmfd, 0);

    sem_t * sem_id = sem_open(SEM_PATH, 0);
    struct shared_data in_msg;
    sem_wait(sem_id);
    /* Update shared data */
    memcpy(&in_msg, shared_msg, sizeof(struct shared_data);
    sem_post(sem_id);
    /* Process data… */

    shm_unlink(SHMOBJ_PATH);
    sem_unlink(SEM_PATH);
    return 0;
}
```

Example 9: *Using named semaphores with shared memory*

- The reader process
  - Maps a memory region (read-only access)
  - Open the semaphore object, already initialized (`sem_open`)
  - Decrements the semaphore counter acquiring an exclusive access to the shared memory region (`sem_wait`)
  - Copy the data fron the memory region to a local variable (`memcpy`)
  - Decrements the semaphore counter and releases the access to the memory region (`sem_post`)
  - Process the data
  - Releases the shared memory region (`shm_unlink`)
  - Close and release the semaphore object (`sem_unlink`)

Choosing a IPC mechanism

Factors

- *“IPC performance is a complex and multivariate problem”*
- Machine architecture
- Data size and data locality
- Presence of virtualization
- Primitives implementation
- Current system workload

Benchmarks

- Cambridge University developed ***ipc-bench*** benchmark and provides a public database of results
  - http://www.cl.cam.ac.uk/research/srg/netos/projects/ipc-bench/

Resources

Weblinks

- https://www.safaribooksonline.com/library/view/linux-system-programming/0596009585/ch04s03.html
- http://www.linuxprogrammingblog.com/all-about-linux-signals?page=3
- https://www.softprayog.in/programming/posix-semaphores