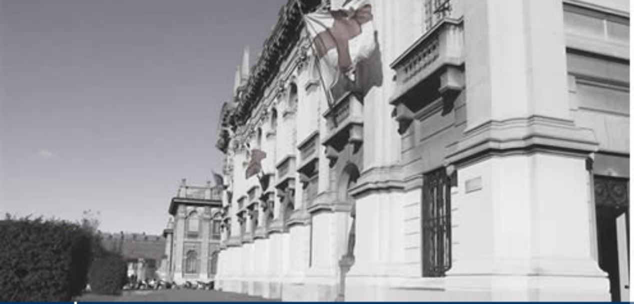


POLITECNICO DI MILANO

Multi-Process Programming in C

Advanced Operating Systems (2016/2017)

Giuseppe Massari
giuseppe.massari@polimi.it

Outline

Multi-process programming

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

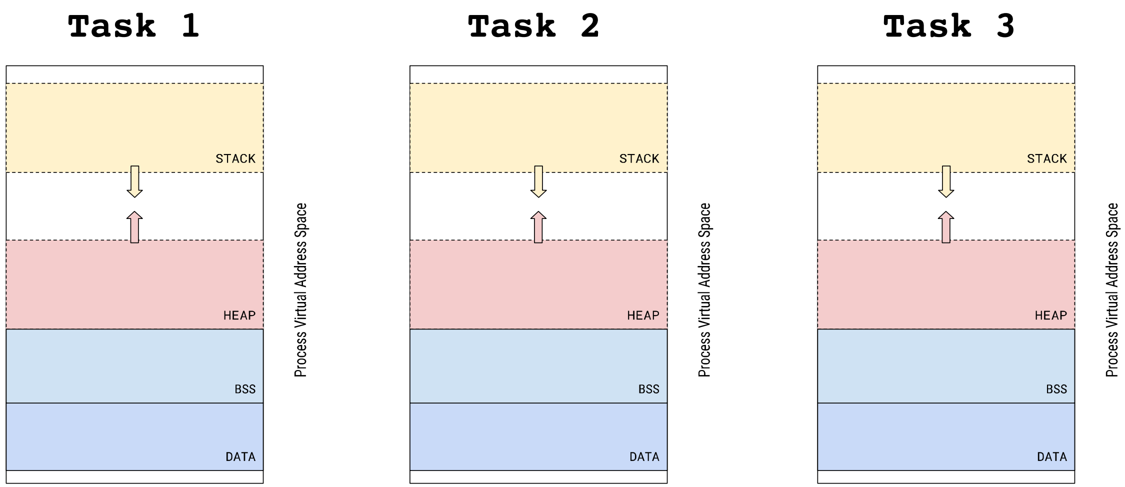
Inter-Process Communication

- Signals
- Pipes
- 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
- 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
  - Tasks featuring high I/O activity (networking, disk accesses, …)



Example 1: *Forking a process*

```c
#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
```

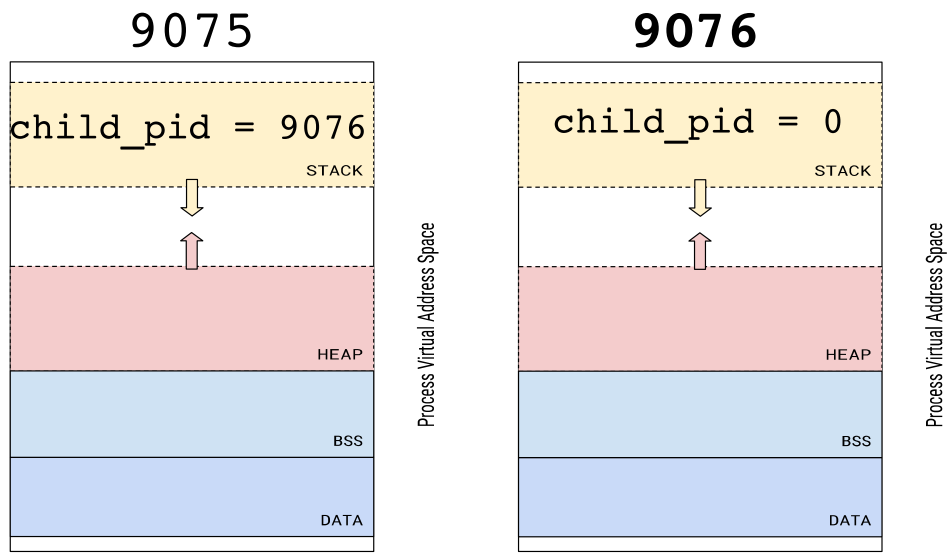
```
Main process id = 9075 (parent PID = 32146)
Parent: child's process id = 9076
Child:  my process id = 9076
```

- 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

Multi-process programming

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

```c
#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:

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

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( "|||||||||||||" );
}
```

Synchronization using `wait()`

- The parent process 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
  - `pid_t` **`wait`**`(int * status)`

- The **`waitpid(`*`pid_t,…`*`)`** call allows the caller process to wait for a specific child process
- The **`wait/waitpid`** calls allow the system to release the resources associated with the child process
  - (e.g., opened files, allocated memory, etc...)

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

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

Spawning executor processes

- Function signatures

```
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[]);
```

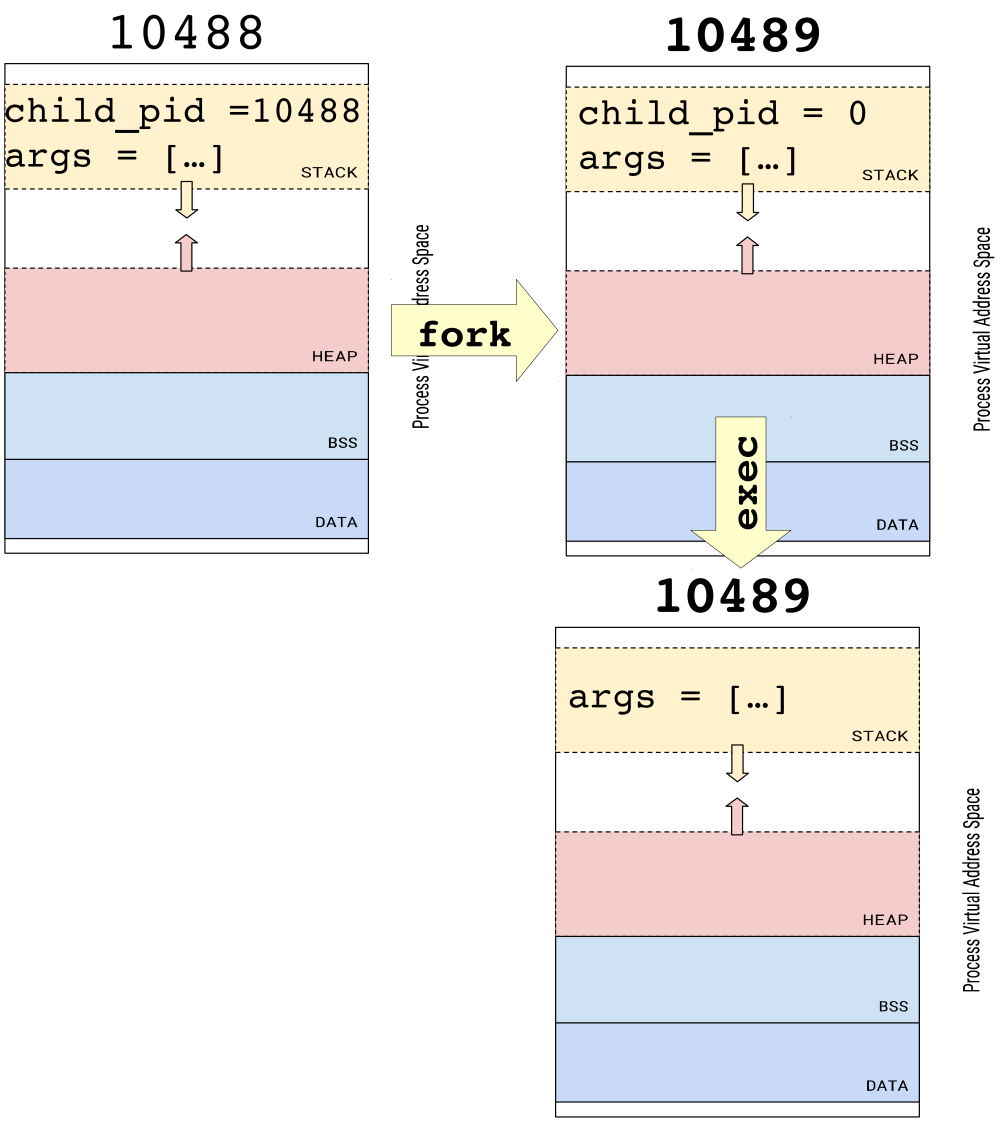
- All the functions take the executable path as first argument
- "`l`" functions accept variable amount of null-terminated `char *`
- "`v`" functions accept the executable path and an array of null-terminated char *
  - Both forward arguments to the executable (arg0 must be set to executable name)
-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

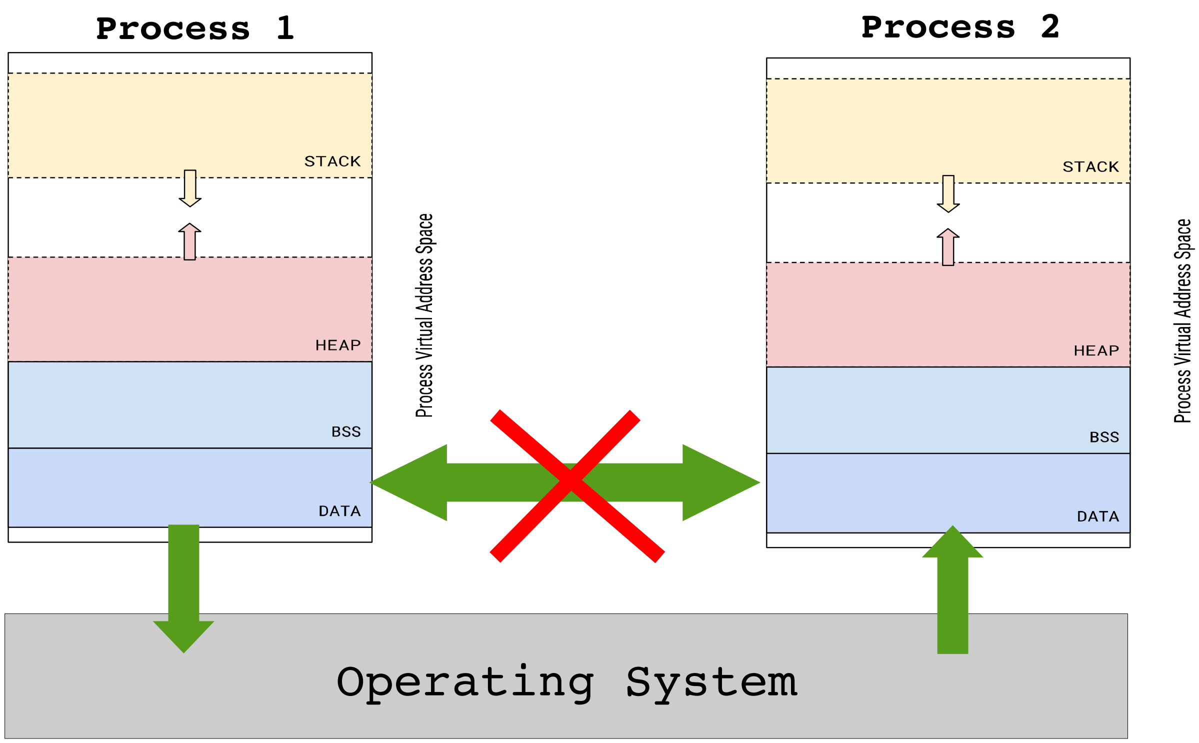
```
int main ()
{
    char * args[] = {
        "ls", "-l", NULL
    };
    pid_t child_pid;

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



Inter-Process Communication

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



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

Signal handling

- Signals may be thought as software equivalent of *hardware interrupts*
- Operating Systems manage a *signal vector table* for each process
  - Conversely, for hardware interrupts there is a single system-wide table
- OS typically defines several signals (name defined as integer macro)
- In Linux, the default action performed to handle a signal is to *terminate* the process
- It is possible to register a custom *signal handler* for each signal
  - Each entry of the signal vector table are
- Signals can be *ignored* at process-level (completely discarded)
- Signals can be *blocked* at process or thread-level
  - Enqueued and managed later, when the process/thread “unmask” the signal

A subset of common POSIX signals

POSIX signals	Portable number	Default action	Description
SIGABRT	6	Terminate	Process abort signal
SIGALRM	14		Alarm clock
SIGCHLD	N/A	Ignore	Child process terminated, stopped or continued
SIGINT	2	Terminate	Terminal interrupt
SIGKILL	9	Terminate	Kill the process
SIGPIPE	N/A	Terminate	Write on a pipe with no one to read it
SIGSEV	N/A	Terminate	Invalid memory reference
SIGUSR1	N/A	Terminate	User-defined signal 1
SIGUSR2	N/A	Terminate	User-defined signal 2
…	…	…	...

Example 4: *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;

void handler (int signal_number) {
  ++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());
  /* Do some lengthy stuff here.  */
  printf ("SIGUSR1 was raised %d times\n", sigusr1_count);
  return 0;
}
```

Example 4: *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

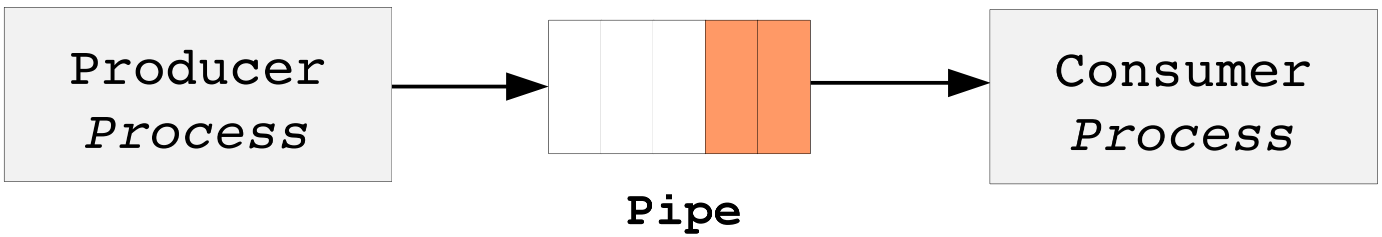
```
$ gcc example4.cpp -o sig_example
$ ./sig_example
Running process... (PID=16151)
```

```
$ kill -SIGUSR1 16151
```

```
SIGUSR1 was raised 1 times
```

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

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

```c
#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;
}
```

Pipes

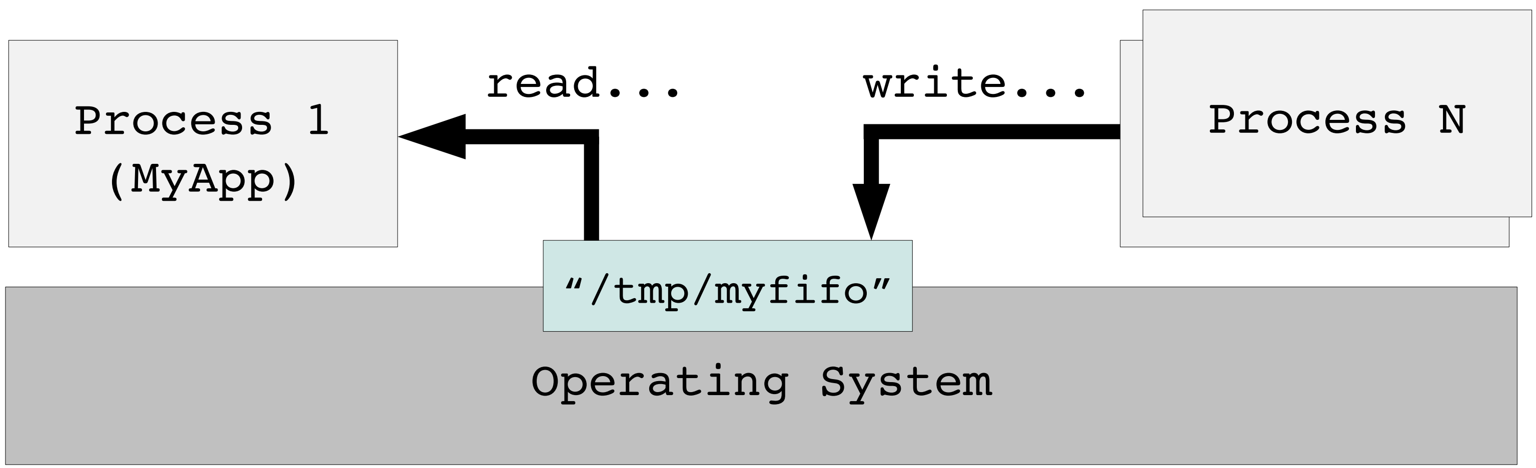
Example 5: *Simple unnamed pipe based messaging*

- 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)

- 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
- Suitable for *unrelated processes* communication



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

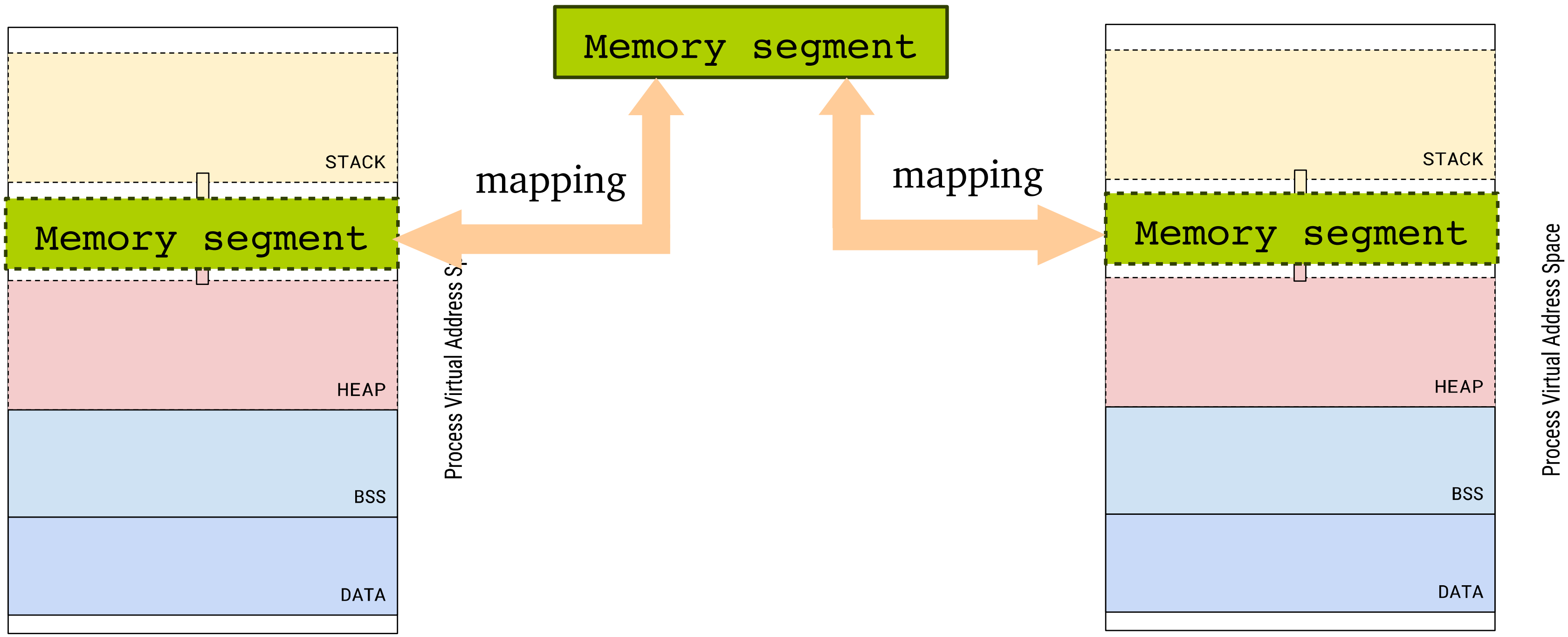
- Low overhead
- Simplicity
- Mutual access solved in kernel-space

Cons

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

Memory mapping

- Shared memory in Linux/UNIX operating systems is based on the concept of memory mapping
- A memory segment can be memory mapped in the address space of multiple processes



Shared memory

Memory mapping

- A POSIX standard has been defined to implement memory mapping application program interfaces
- **shm_open()** – opening/creation of a shared memory segment referenced by a name
  - A special file will appear in the file-system under "/dev/shm/" with the provided name
  - The special file represents a POSIX object and it is created for persistence
  - **ftruncate(…)** function resize to memory region to the correct size
- **mmap()** – mapping of the memory segment referenced by the file descriptor returned by shm_open()
- **munmap()** – unmapping of the memory segment
- **shm_unlink()** – removal of shared memory segment object if nobody is referencing it

- Link to POSIX real-time extension library to build (gcc … **-lrt**)

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(0, 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;
}
```

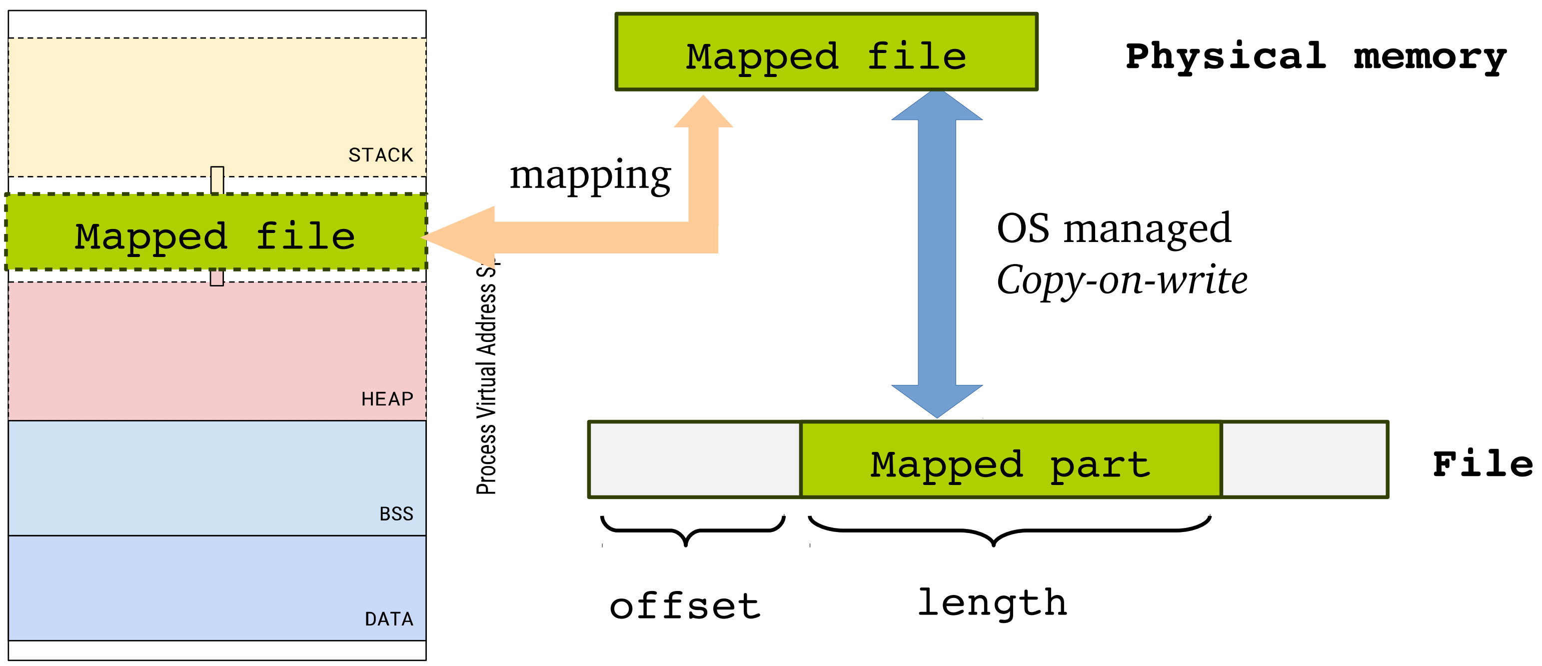
- The client opens the memory segment “AOS” in read-only mode
- The client maps the memory segment in read-only mode
- The client write the memory segment content to console

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;
}
```

Semaphores

- Concurrency in multi-tasking applications may introduce race conditions → we need to protect shared resource
- *Semaphores* are examples of structure aiming at solving such a problem in multi-process applications
- Semaphores are usually system objects managed by the OS kernel
- Semaphores can be thought as counters that we can manipulate by performing two actions: *wait* and *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

Shared memory

POSIX semaphores

- **sem_open()** – opening/creation of a *named* semaphore
  - Useful for synchronization among *unrelated processes*
- **sem_wait()** – Decrement the counter and lock if counter = 0
  - Initial counter value can be set to > 1
- **sem_post()** – Increment the count and unlock the critical section if counter > 0

- **sem_close()** – Close all the references to the named semaphore
- **sem_unlink()** – Destroy semaphore object
  - if all the references have been closed

- Link to POSIX real-time and threads extension library to build (gcc … **-lrt -pthread**)

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_close(sem_id);
    sem_unlink(SEM_PATH);
    return 0;
}
```

Shared memory

Example 9: *Using 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 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 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_close(sem_id);
    sem_unlink(SEM_PATH);
    return 0;
}
```

Shared memory

Example 9: *Using 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`)

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