Systems Programming

A3, A4

Luca Mayr / Christian Rieger

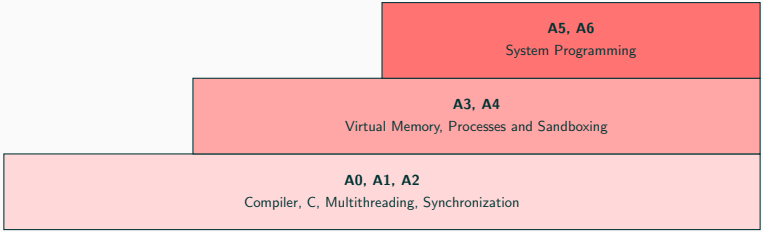
October 21, 2019

IAIK – Graz University of Technology

A3 - Virtual Memory

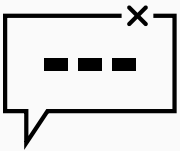
A4 - Interprocess Communication

Course Overview



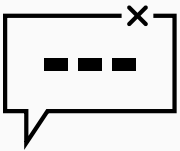
A3 - Virtual Memory

Segmentation fault



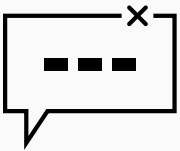
- Common scenario: User program accesses “invalid” memory location

Segmentation fault



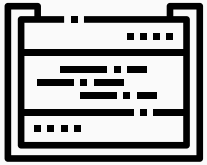
- Common scenario: User program accesses “invalid” memory location
- The OS invokes a fault handler
- This fault handler can abort the user program or **fix the situation** by “making” the address valid
- Efficient: Don’t assign memory until necessary

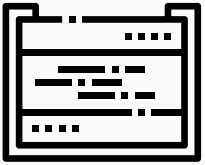
Segmentation fault



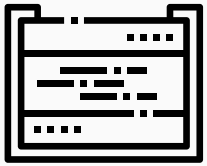
- Common scenario: User program accesses “invalid” memory location
- The OS invokes a fault handler
- This fault handler can abort the user program or **fix the situation** by “making” the address valid
- Efficient: Don’t assign memory until necessary
- Are pointers addresses in physical memory?
  - How can addresses in physical memory be “invalid”?

- **Pointers are not addresses in physical memory**

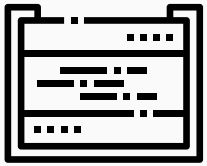




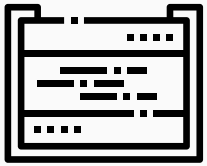
- **Pointers are not addresses in physical memory**
- Pointers are virtual addresses



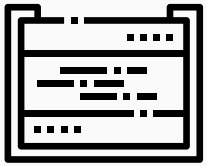
- **Pointers are not addresses in physical memory**
- Pointers are virtual addresses
- Addresses are translated from virtual addresses to real physical addresses transparently



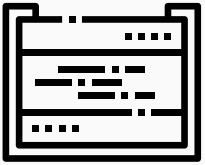
- **Pointers are not addresses in physical memory**
- Pointers are virtual addresses
- Addresses are translated from virtual addresses to real physical addresses transparently
- Therefore, memory is split into parts called “pages”



- **Pointers are not addresses in physical memory**
- Pointers are virtual addresses
- Addresses are translated from virtual addresses to real physical addresses transparently
- Therefore, memory is split into parts called “pages”
- Operating system can “map” pages



- **Pointers are not addresses in physical memory**
- Pointers are virtual addresses
- Addresses are translated from virtual addresses to real physical addresses transparently
- Therefore, memory is split into parts called “pages”
- Operating system can “map” pages
- Operating system can maintain this mapping per process



- **Pointers are not addresses in physical memory**
- Pointers are virtual addresses
- Addresses are translated from virtual addresses to real physical addresses transparently
- Therefore, memory is split into parts called “pages”
- Operating system can “map” pages
- Operating system can maintain this mapping per process

→ different processes can use the same addresses, but “see” different things there

- Experiment with different kinds of variables, which addresses do they get?
- Observe memory usage in practice, when does it really increase?
- **Answer questions from the test system questionnaire!**
- **Register + participate in one of the virtual memory discussions!**

A4 - Interprocess Communication

- **A4 is 25% of all possible points in SLP**

- **A4 is 25% of all possible points in SLP**
- You have to implement some Interprocess Communication Example

- **A4 is 25% of all possible points in SLP**
- You have to implement some Interprocess Communication Example
- We provide you with a basic framework

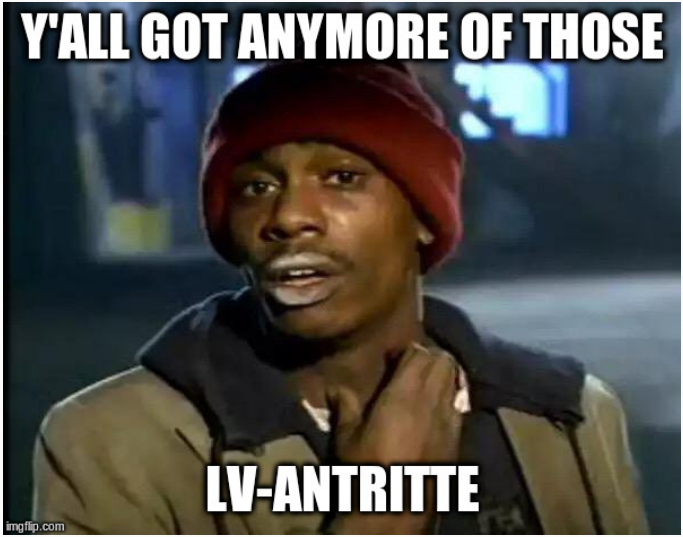
- **A4 is 25% of all possible points in SLP**
- You have to implement some Interprocess Communication Example
- We provide you with a basic framework
- Just change the functions which are marked with TODO

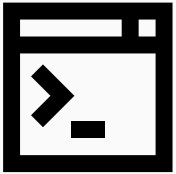
- **A4 is 25% of all possible points in SLP**
- You have to implement some Interprocess Communication Example
- We provide you with a basic framework
- Just change the functions which are marked with TODO
- You need to have a certain understanding of fork, exec and shared memory

- **A4 is 25% of all possible points in SLP**
- You have to implement some Interprocess Communication Example
- We provide you with a basic framework
- Just change the functions which are marked with TODO
- You need to have a certain understanding of fork, exec and shared memory
- Use the man pages to get this information, or just use google

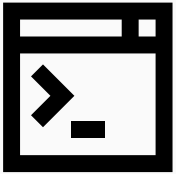
- **A4 is 25% of all possible points in SLP**
- You have to implement some Interprocess Communication Example
- We provide you with a basic framework
- Just change the functions which are marked with TODO
- You need to have a certain understanding of fork, exec and shared memory
- Use the man pages to get this information, or just use google
- Take a look at Assignment A2 you will need to use semaphores again :-)


INTER PROCESS
COMMUNICATION
IGNORE

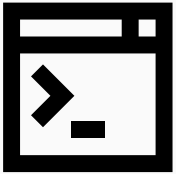

Y'ALL GOT ANYMORE OF THOSE
LV-ANTRITTE
imgflip.com



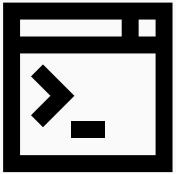
- e.g. using a shell on Linux



- e.g. using a shell on Linux
- the shell is a process itself



- e.g. using a shell on Linux
- the shell is a process itself
- executing a program with ./program



- e.g. using a shell on Linux
- the shell is a process itself
- executing a program with ./program
- $\rightarrow$ the program starts

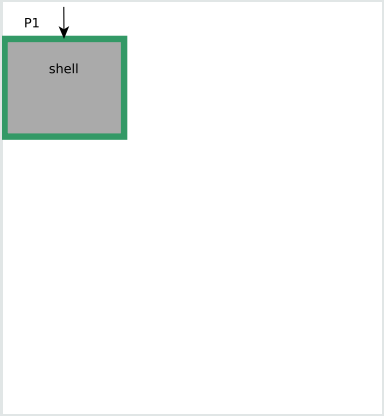
Code

```
$ ls ~/
```

Code

```
//shell stuff
```

Image

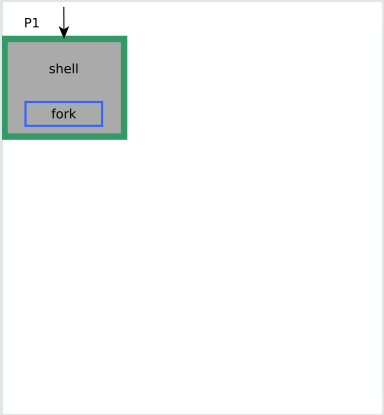


Code

```
//shell stuff

pid_t pid = fork();
```

Image

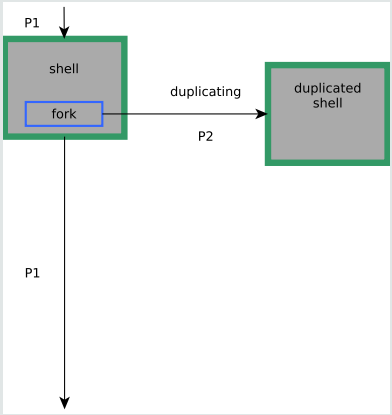


Code

```
//shell stuff

pid_t pid = fork();
if(pid == 0)
```

Image

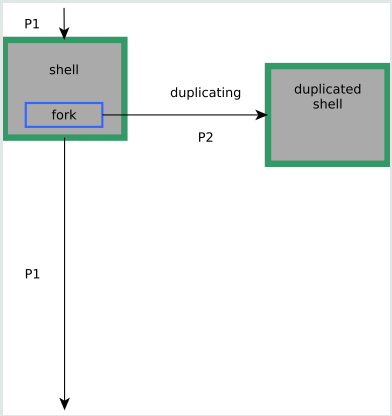


Code

```
//shell stuff

pid_t pid = fork();
if(pid == 0)
{
  const char* args[] = {"~/"};
}
else
{
  //do further shell stuff
}
```

Image

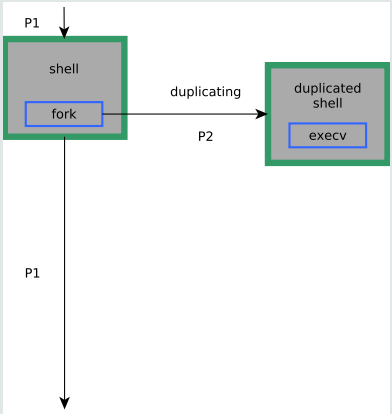


Code

```
//shell stuff

pid_t pid = fork();
if(pid == 0)
{
  const char* args[] = {"~/"};
  execv("/bin/ls", args);
}
else
{
  //do further shell stuff
}
```

Image

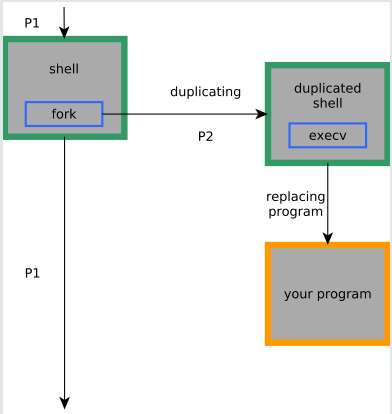


Code

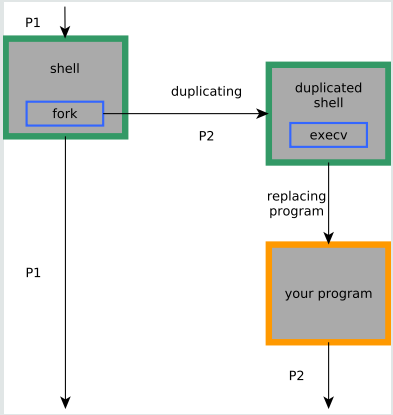
```
//shell stuff

pid_t pid = fork();
if(pid == 0)
{
  const char* args[] = {"~/"};
  execv("/bin/ls", args);
}
else
{
  //do further shell stuff
}
```

Image



Image


P1
shell
fork
duplicating
P2
duplicated shell
execv
replacing program
your program
P1
P2

1. the shell gets forked respectively "cloned"

1. the shell gets forked respectively "cloned"
2. two identical processes exist simultanely independently

Fork and Exec summary

1. the shell gets forked respectively "cloned"
2. two identical processes exist simultanely independently
3. the forked process will be emptied

Fork and Exec summary

1. the shell gets forked respectively "cloned"
2. two identical processes exist simultanely independently
3. the forked process will be emptied
4. your program will be filled into the empty process

Fork and Exec summary

1. the shell gets forked respectively "cloned"
2. two identical processes exist simultanely independently
3. the forked process will be emptied
4. your program will be filled into the empty process

Fork and Exec summary

1. the shell gets forked respectively "cloned"
2. two identical processes exist simultanely independently
3. the forked process will be emptied
4. your program will be filled into the empty process
5. execution of your main starts

- `fork` followed by an `exec`
  - Worse than just creating a process

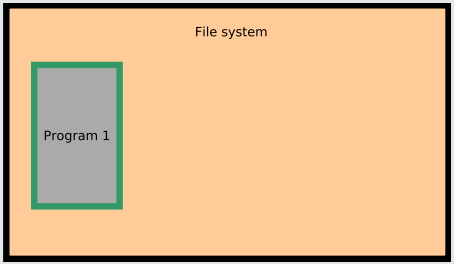
- `fork` followed by an `exec`
  - Worse than just creating a process
  - No, because of COW:
    - Forked process shares memory with the old process
    - Memory is copied upon write access

    → Almost nothing copied if followed by an `exec`!

Code

```
/* something in the main or whatever
   */
```

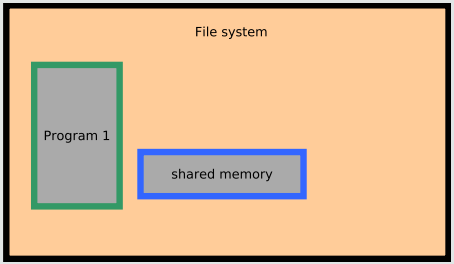
Image

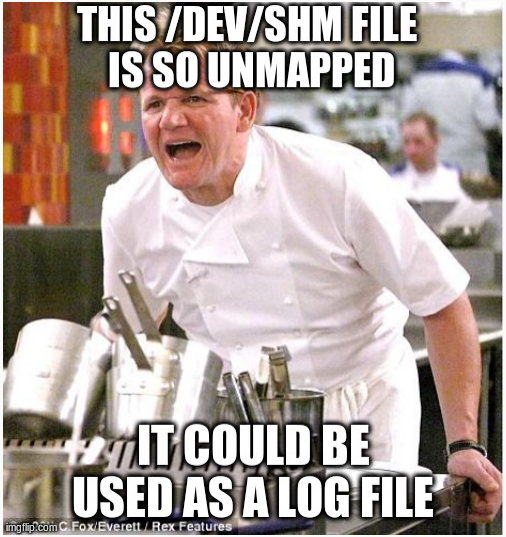


Code

```
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644)
  ;
```

Image




THIS /DEV/SHM FILE
IS SO UNMAPPED
IT COULD BE
USED AS A LOG FILE
imgflip.com
C. Fox/Everett / Rex Features

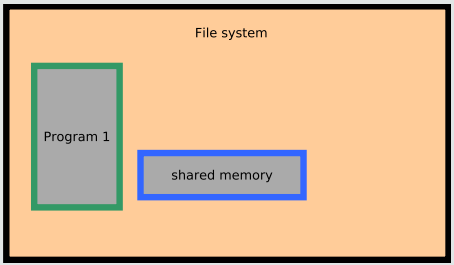

WHY DON'T WE
TAKE THE SHM FILE
AND MAP IT INTO
THE VIRTUAL MEMORY

Code

```
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644)
  ;

/* enlarge the shared memory object
  */
ftruncate(fd, 1000);
```

Image



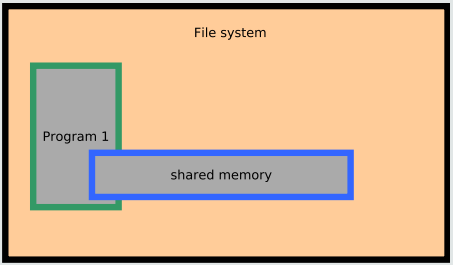
Code

```
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644)
  ;

/* enlarge the shared memory object
  */
ftruncate(fd, 1000);

/* now map the shared object */
char* ptr = (char*) mmap(NULL, 1000,
   PROT_READ | PROT_WRITE, MAP_SHARED
  , fd, 0);
```

Image



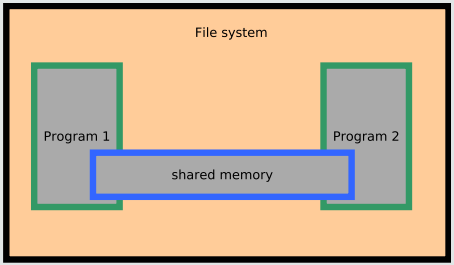
Code

```
/* found in (/dev/shm/obj)  */
int fd = shm_open("obj",O_RDWR,0644)
  ;

/* enlarge the shared memory object
  */
ftruncate(fd, 1000);

/* now map the shared object */
char* ptr = (char*) mmap(NULL, 1000,
   PROT_READ | PROT_WRITE, MAP_SHARED
  , fd, 0);
  pid_t pid = fork();
```

Image



Thanks for your attention!