Processes

Process

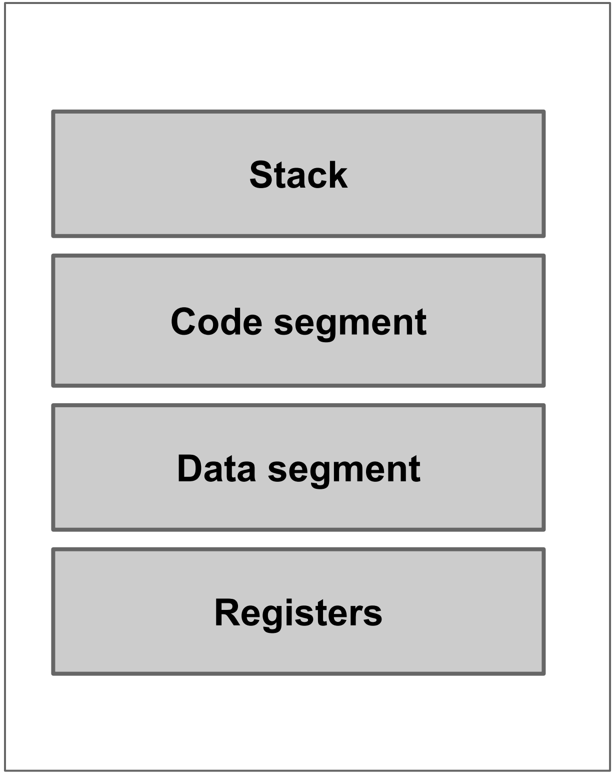
Program - binary file

Process - the program is under execution

Process in UNIX:

- works in user space
- has specific attributes in the kernel

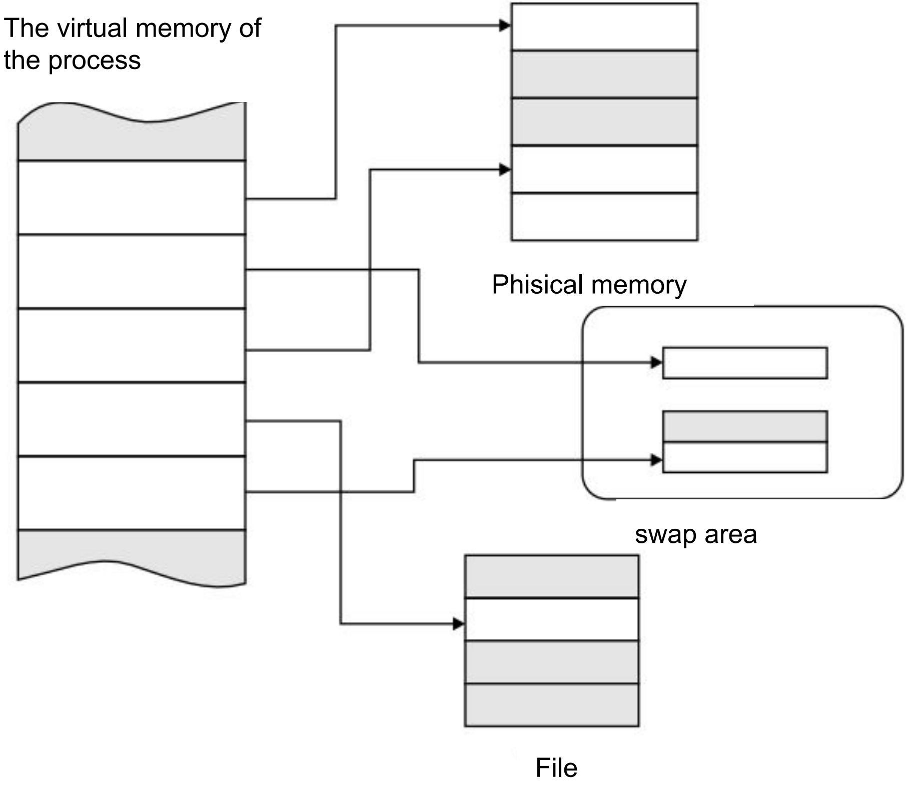
Process



Each process is executed in its own virtual address space

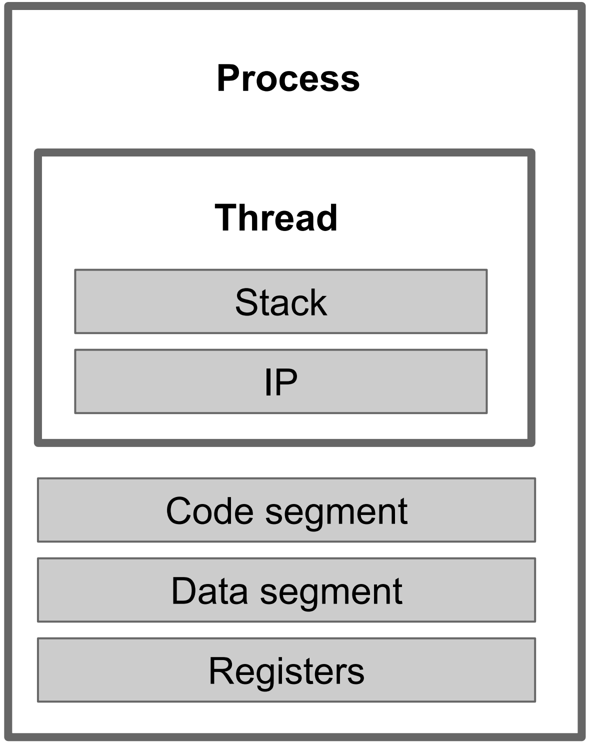
You must use IPC to communicate between multiple processes

The virtual memory of the process



Virtual memory is a method of managing computer memory that allows you to run programs that require more RAM than your computer has by automatically moving parts of the program between the main memory and secondary storage.

Processes and threads



Processes:

- Support for multiple command threads within a single process
- Interaction through shared memory

What to choose?

Processes are generally **more reliable**

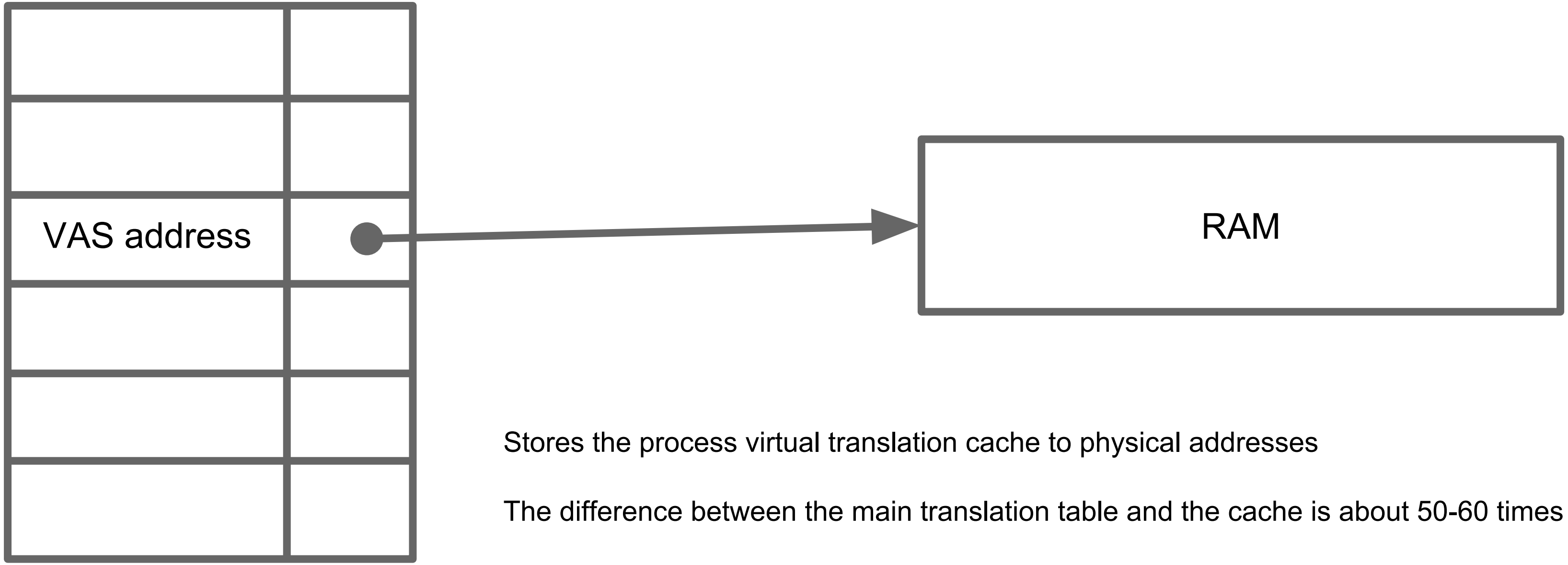
- processes interacting at different times
- distributed application
- data security

Threads - generally **faster**

- easier IPC

TLB (Translation lookaside buffer)

Translation lookaside buffer - Hardware



Stores the process virtual translation cache to physical addresses

The difference between the main translation table and the cache is about 50-60 times

Processes

Several important kernel components are associated with processes:

- memory management
- process planner
- interprocess communication

The start of the process

```
int main(int argc, char *argv[]);
```

argc - number of arguments

argv - array of pointers to arguments

Command line argument

```
int main(int argc, char *argv[]) {
    int i;
    for (i = 0; i < argc; i++)
        printf("argv[%d]: %s\n", i, argv[i]);
    exit(0);
}
```

ISO C and POSIX.1 element of the array argv[argc] must be NULL:

```
for (i = 0; argv[i] != NULL; i++)
```

The completion of the process

Normal ways

- Return from main function
- Call the exit function
- Calling _exit or _Exit
- Returning the pthread_exit function from the last thread

Abnormal ways

- Calling the abort function
- Receive signal

Functions of the exit family

_exit и **_Exit** - instant return of control to the kernel

exit - before transferring control to the kernel, it performs a number of steps to free up resources

```
#include <stdlib.h>

void exit(int status);
void _Exit(int status);

#include <unistd.h>

void _exit(int status);
```

Function atexit

Setting output handler functions.

Up to 32 handlers can be specified according to ISO C.

Handlers are called in LIFO order

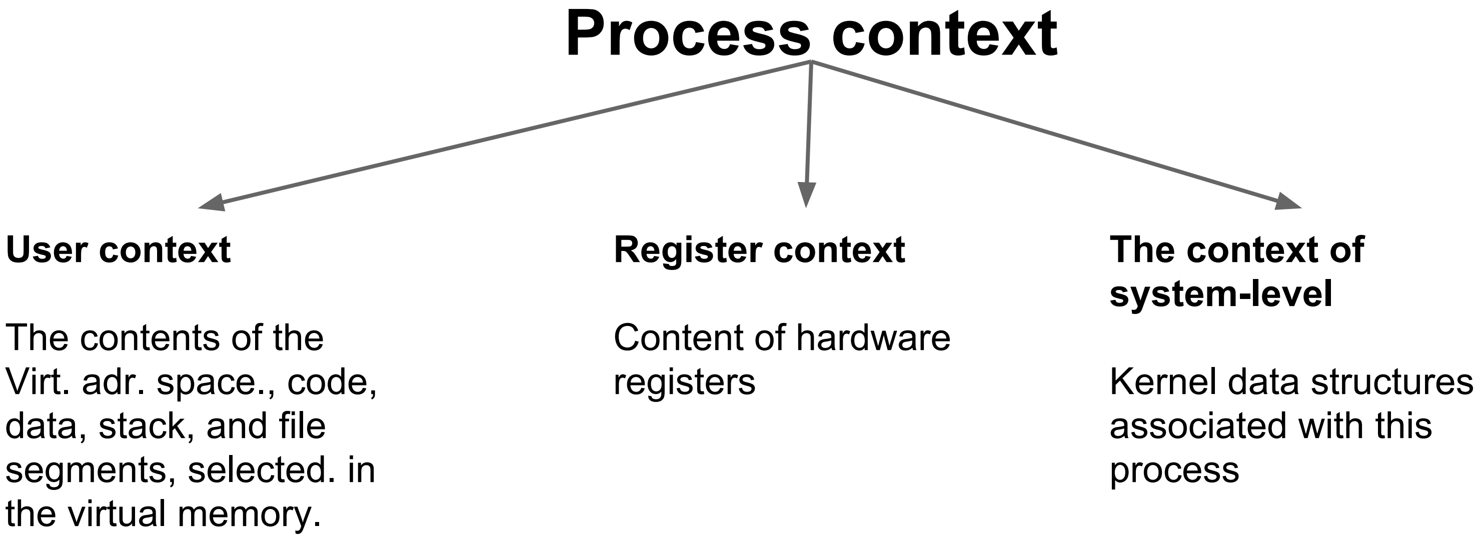
#include <stdlib.h>

```
int atexit(void (*func)(void));
```

0 in case of success,

not 0 - otherwise

Process management



Process context

User context

The contents of the Virt. adr. space., code, data, stack, and file segments, selected. in the virtual memory.

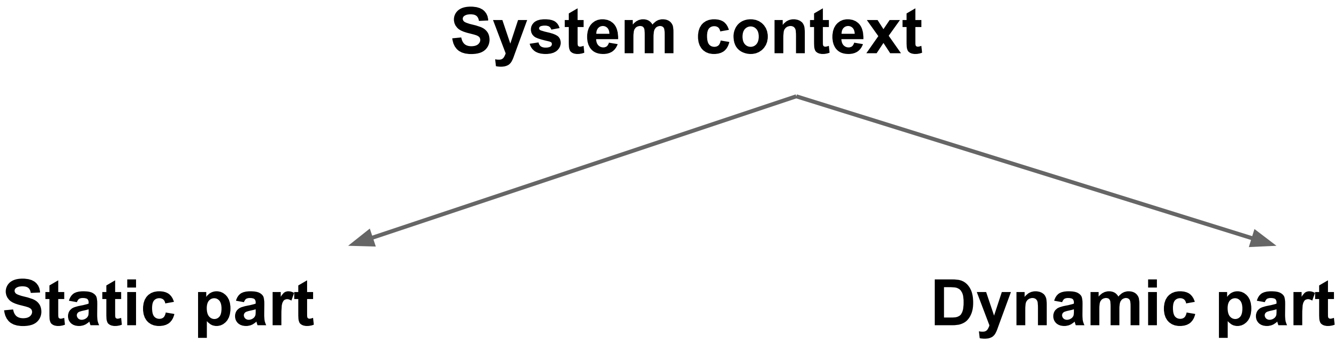
Register context

Content of hardware registers

The context of system-level

Kernel data structures associated with this process

System process context



System context

Static part

Dynamic part

One or more stacks that are used by a process when running in user mode and kernel mode (during interrupts and system calls)

Static part

- Process identifier (PID)
- The ID of the parent process (PPID)

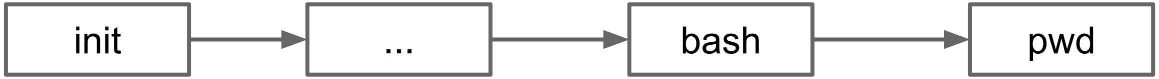
#include <unistd.h>

pid_t getpid(void);

Returns the ID of the calling process

pid_t getppid(void);

Returns the ID of the parent process



example: ps -ef --forest, pstree

PID

By default, the maximum identifier in the system **32768 (16 bits)**

/proc/sys/kernel/pid_max

Assigned by the system linearly

Static part

- Real user ID of the process
- The real group ID of the process

```
#include <unistd.h>

uid_t getuid(void);

gid_t getgid(void);
```

Static part

- Process priority
- Open file descriptor table

Example

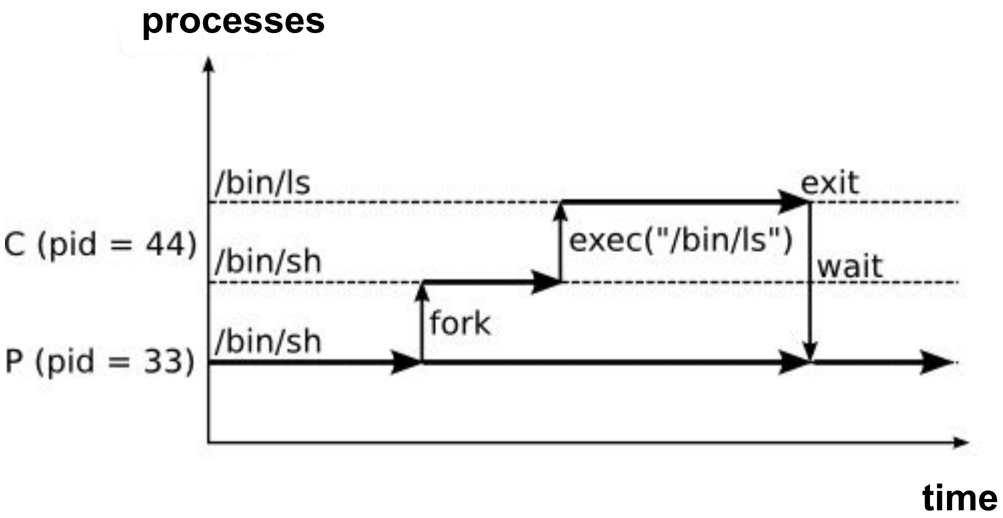
Output of the following process characteristics:

- PID
- PPID
- UID
- GID

example1.c

The creation of new processes

- fork
- exec
- wait



Creating a process and running the program are separated

Fork system call

There are two main cases when the fork function is used:

1. When a process wants to duplicate itself so that the parent and child processes can run different parts of the program at the same time. Used, for example, in network servers. The parent process waits for the service request from the client, when it receives it calls the fork and passes the service request to the child process, and then returns to wait for the next request.

2. When a process wants to run another program. This is commonly used in command shells. In this case, the child process calls the exec function as soon as the fork returns control.

System call: fork

#include <unistd.h>

pid_t fork(void);

Returns 0 in the child process,

ID child process - in parent,

-1 in case of error

Possible errors

EAGAIN — the kernel is unable to allocate certain resources, such as a new pid

ENOMEM — insufficient kernel memory resources

Example

Create a child process

example2.c

Isolated processes

Each process is in its own isolated address space

Duplicating memory on a “copy-on-write basis”

wait and waitpid

Waiting for process to complete

The parent is notified of the child's completion by a signal SIGCHLD

Parent can:

- Redefine
- Ignore (default)

Often you need to know exactly how the process ended

System call: wait

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

pid_t wait (int *status);
```

Returns the pid of the completed child process, or -1 if an error occurs

System call: wait()

Status pointer - additional information about the child process.

status - int with bits set depending on the type of completion

Macros for interpretation of the completion status

#include <sys/wait.h>

int WIFEXITED (status);	returns true if the process terminates by calling _exit(), as usual
int WEXITSTATUS (status);	Exit code
int WIFSIGNALED (status);	returns true if the interruption of the process caused the alarm
int WTERMSIG (status);	the number of the signal that caused the interrupt
int WCOREDUMP (status);	returns true if the process dumped core in response to the signal

Macros for interpretation of the completion status

int WIFSTOPPED (status);	return true if the process was stopped or continued, respectively
int WIFCONTINUED (status);	
int WSTOPSIG (status);	number of signal that stopped the process

System call: wait

On error **errno**:

- ECHILD — the calling process has no children
- EINTR — signal was received while waiting

System call: waitpid()

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

pid_t waitpid (pid_t pid, int *status, int options);
```

Parameter pid

-1	Wait for any child process; behavior similar to wait()
> 0	Wait for any child process whose pid is exactly equal to the specified value; for example, if 500, a child process with pid equal to 500 is expected

Parameter options

The value obtained using logical or:

- **WNOHANG** — do not block the call, return the result immediately if no suitable process has completed (stopped or continued);

- **WUNTRACED** — selecting it sets the WIFSTOPPED parameter even if the calling process does not monitor its child; this property helps to implement more General job management, as it does in the shell;

- **WCONTINUED** — if set, the WIFCONTINUED bit in the returned status parameter will be set even if the calling process does not track its child; as with WUNTRACED, the parameter is useful for implementing the shell.

Possible error

errno:

- ECHILD — the process or processes specified with the pid argument do not exist or are not descendants of the caller;

- EINTR — signal was received while waiting;

- EINVAL — the options argument is not specified correctly.

Special case 1

The parent process terminates before the child process

The parent of the child process will be the process **init** (pid = 1).

Special case 2

The child process terminates before the parent knows it

The child process becomes a process ***zombie*** ***(ps -aux)***

Special case 3

Case 2 with the difference that the parent of the process was the process **init**

The **init** process works in a special way, and when the **child** ends, it calls the wait function itself. Thus preventing the appearance of **zombies**.

Call family exec

```
#include<unistd.h>

int execl (const char *path,
           const char *arg,
           ...);
```

Example

```
ret = execl ("/bin/ls", "ls", NULL);

if (ret == –1)
    perror ("execl");
```

Another call: exec

- execlp
- execle
- execv
- execvp
- execve

l	Arguments are passed as a list
v	Arguments are passed as a array
p	Search for a file using a custom path
e	A new environment is created

Example

```
const char *args[] = {
   "vi",
   "/home/user/hooks.txt",
   NULL
};

int ret;

ret = execv ("/bin/vi", args);  // execlp("/bin/vi", "vi", "/home/user/hooks.txt", (char *)NULL);

if (ret == –1)
         perror ("execvp");
```

Do not change

- PID
- PPID
- Priority
- UID
- GID

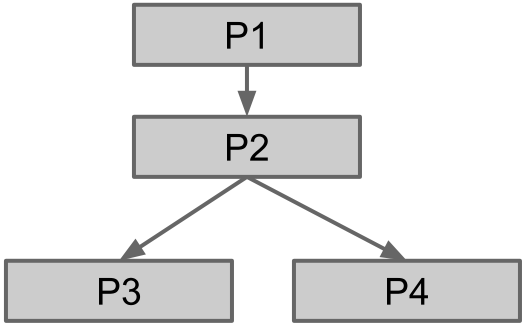
Example

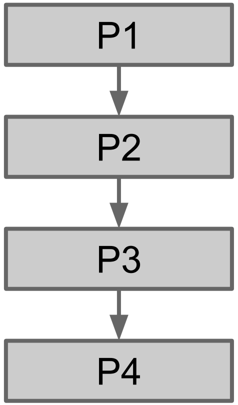
A simple command interpreter

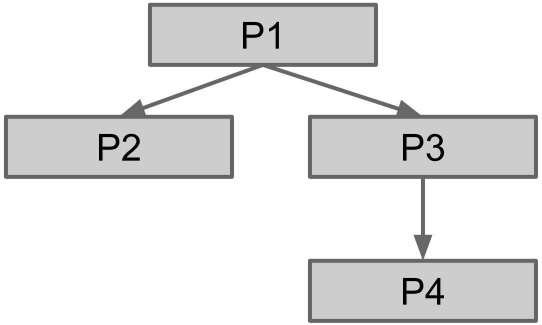
example3.c

Exercise №1

Implement the following process family tree:







For each process, display the PID and PPID. All parents should wait for their children to finish before finishing itself

Exercise №2a

Find all Prime numbers from 1 to 10 000 000

Run N processes. Each process takes its own part of the range (10 000 000 / P)

Output all found Prime numbers to console

Exercise №2b

Parallel computation of PI based on the following series:

*Pi=4 − 4/3 + 4/5 − 4/7 + … + ((-1)^(n+1)*4)/(2*n-1)*

Divide the range by the number of processes P (N/P). Create P processes. Each process calculates its part of the sum and displays it on the screen