**Procese UNIX**

Curs 5-7

**Programe C sub Linux**

Se editeaza un program in vim sau alt editor.

Se compileaza cu

gcc -Wall -o numeexecutabil numeprogramsursa.c

Se executa cu

./numeexecutabil

**gcc -Wall -o hello helloworld.c**

To execute:

./hello

|  |
| --- |
| helloworld.c |
| -------------------------------------------- #include <stdio.h> int main(int argc, char\*\* argv) {        printf(“Hello world!\n”);  int count;  int sum=0;  if (argc>1)  for (count=1; count<argc; count++){  printf("argv[%d]= %s \n", count, argv[count]);  sum+=atoi(argv[count]);  }  else  printf("The command has no arguments.\n");  printf("Sum is %d \n", sum);  return 0; } |
|  |

Manual sections can be seen by typing command **man man**

1 User Commands

2 System Calls

3 C Library Functions

...

Or use **whatis printf**

Exemplu program C 2: Citeste nume si le afiseaza.

#include <stdio.h>

int main(int argc, char\*\* argv) {

char name[64];

while(scanf("%s", name) == 1) {

printf("Hello %s!\n", name);

}

return 0;

}

**Utilizare pointeri**

**void \*malloc(size\_t size)**

**Syntax:**

ptr = (cast-type\*) malloc(byte-size)

**Example:**

**int\* ptr; //we want this to be an array of 100 integer numbers**

**ptr = (int\*) malloc(100 \* sizeof(int));**

//Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory.

//And, the pointer ptr holds the address of the first byte in the allocated memory.

**free(ptr);  
  
  
  
Typedef pointer example:**

typedef struct

{  
 char name[21];

char city[21];

char state[3];

} NewType;

typedef NewType \* NewTypePointer;

NewTypePointer r;

r = (NewTypePointer)malloc(sizeof(NewType));

Verificare erori de memorie cu valgrind

valgrind ./executabil

Exemplu: Citeste din fisier o matrice si se afiseaza pe ecran. Alocand memoria dynamic.

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char\*\* argv) {

int\*\* m;

int rows, cols, i, j;

FILE\* f;

f = fopen(argv[1], "r");

fscanf(f, "%d %d", &rows, &cols);

m = (int\*\*)malloc(rows\*sizeof(int\*));

for(i=0; i<rows; i++) {

m[i] = (int\*)malloc(cols\*sizeof(int));

for(j=0; j<cols; j++) {

fscanf(f, "%d", &m[i][j]);

}

}

fclose(f);

for(i=0; i<rows; i++) {

for(j=0; j<cols; j++) {

printf("%3d ", m[i][j]);

}

printf("\n");

}

for(i=0; i<rows; i++) {

free(m[i]);

}

free(m);

return 0;

}

**Read more in man sections 2 and 3 about the following C functions (and corresponding c libraries required - header files):**

* **scanf, printf, fgets, getchar, fprintf, fopen, fclose, open, close, fread, fwrite, read, write**
* **malloc, free**
* **strcmp, strcpy, strstr, strlen, strcat, strncpy, strncmp,**
* **atoi, floor, round, sqrt**

**File system calls**

**• Unix system calls**

call directly to the kernel functions of the OS

(the lower level of file processing)

• **Standard functions in C library**:

fopen, fprintf, fscanf, fclose, fseek, etc

- (the higher level of file processing)

**The open system call (with close, read, write)**

flags: O\_RDONLY, O\_WRONLY, O\_RDWR, O\_NDELAY, O\_APPEND, O\_CREAT, O\_EXCL (with O\_CREAT ), O\_TRUNC

mode: access rights; works in accordance with *umask* **S\_IRUSR, S\_IWUSR , S\_IXUSR**, *S\_IRGRP, S\_IWGRP , S\_IXGRP*, S\_IROTH, S\_IWOTH , S\_IXOTH

- error

- the file descriptor

Return: -1 error

>0 the file descriptor

**man 2 open**

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

int open(const char \*pathname, int flags [,mode\_t mode]);

#include <unistd.h>

int close (int fd) ;

man 2 read

#include <unistd.h>

ssize\_t read(int fd, void \*buf, size\_t count);

ssize\_t write(int fd, void \*buf, size\_t count);

Return number of bytes read/written or -1 on error or 0 on EOF.

Un program C care citeste 7 caractere dintr-un fisier si le scrie in alt fisier.

**Example 1:**4

#include <fcntl.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <stdio.h>

#include <unistd.h>

int main(){

int fisier1 = open("f1.txt", O\_WRONLY);

int fisier2 = open("f2.txt", O\_RDONLY);

printf("%d and %d", fisier1, fisier2); // descriptorii de fisiere

char buff[100];

read(fisier2,buff, 7);

write(fisier1, buff, 7);

return 0;

}

Example 2:

// read write example

int main(){

int fisier = open("cont1", O\_RDONLY);

char buff[256];

read(fisier, buff, 7);

write(1, buff, 7); // afiseaza la stdout (in consola)

return 0;

}

**Function lseek**

**#include <sys/types.h>**

#include <unistd.h>

off\_t lseek(int fd, off\_t offset, int whence);

offset: the number of bytes on which the move is made;

whence: position relative to:

SEEK\_SET - the beginning of the file;

SEEK\_CUR - current position;

SEEK\_END - end of file.

Example 1:

#include <stdio.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <unistd.h>

int main(){

int f=open("f.txt", O\_RDWR);

char buff[1];

lseek(f, 7, SEEK\_CUR);

read(f, buff, 1);

printf("%c\n", \*buff);

lseek(f, -1, SEEK\_CUR);

read(f, buff, 1);

printf("%c\n", \*buff);

lseek(f, 0, SEEK\_SET);

read(f, buff, 1);

printf("%c\n", \*buff);

return 0;

}

Example 2:

#include <fcntl.h>

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <stdio.h>

#include <stdio.h>

int main(){

int f = open("f.txt", O\_RDWR); //change flag to O\_APPEND

printf("%ld\n", lseek(f, 0, SEEK\_SET));

char a='A';

printf("%ld\n", write(f, &a, 1));

perror("Error:");

return 0;

}

**Standard functions in the C library**

man 3 fopen

#include <stdio.h>

FILE \*fopen(const char \*path, const char \*mode);

int fclose(FILE \*fp);

int fprintf(FILE \*stream, const char \*format, ...);

int fscanf(FILE \*stream, const char \*format, ...);

size\_t fread(void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

size\_t fwrite( void \*ptr, size\_t size, size\_t nmemb, FILE \*stream);

int fseek(FILE \*stream, long offset, int whence);

**Procese**

**Process = a running program that uses a system resource set (memory, processor, disk,**

**network interface, etc.)**

**The image of a process in memory**

**User context** – the portion of the address space accessible during execution

by that process in user mode;

•

**Kernel context** – maintained by the kernel and accessible only through

specific system calls.

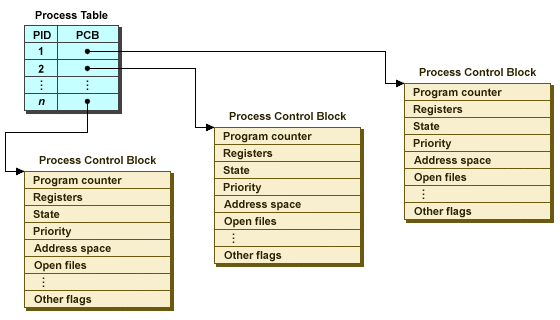
Entry in the process table

* process state
* pointer to the User area and the Process Memory Regions Table
* process size in memory
* PID, PPID
* UID, EUID, GID, EGID
* process priority (number between 1 and 39 used for scheduling)
* signals sent to the process
* time statistics (e.g., using the processor)
* process memory status (if the process image is in the main memory or in the swap memory)
* pointer to the next process in the process queue that is in the READY state
* event descriptors that occurred while the process was in SLEEP state

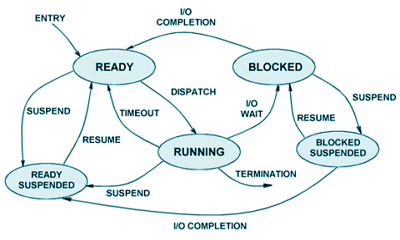
User area

* pointer to the entry in the process table that corresponds to this user area
* UID, EUID, GID, EGID - used to determine process access rights
* timers - time spent in user mode and in kernel mode
* vector of actions to handle signals
* process control terminal - the one from which it was launched
* return values and possible errors after making system calls
* the current directory and the root directory
* environmental variables
* possible kernel restrictions imposed on the process (e.g., process size in memory)
* file descriptor table (data about all open process files)
* umask access rights mask for newly created files by this process

**Process Control Block**

****

**The states of a process**

****

ready (the process is ready to execute when a processor becomes available),

running (the process is currently being executed by a processor)

blocked (the process is waiting for a specific event to occur before it can proceed)

suspended (execution is suspended either because it waits for a resource or other cases)

**fork**

The UNIX way of creating a process is to duplicate the current process, by making a copy of it using the system call fork. This leads to another unfamiliar aspect: source code that looks simple and appears to start at the beginning and finish at the end, will actually execute along split paths. Take for instance the code below. It will print "after" twice. That's because after calling fork, there will be two processes: the original one (called parent), and the copy of it (called child). Both processes continue from the instruction after fork, and thus will both print "after".

Processes that run simultaneously are said to be concurrent A quick check shows that a system usually has significantly more processes than CPUs and cores. SO how can all those processes be active at the same time since there are not enough processing units?

a. The system gives each process a quanta of time on the CPU and switches between processes after each quanta

b. These time quants are imperceptibly small, so all processes seem to make progress simultaneously.

Problem: processes may yield wrong results when working on the same resources as other processes, even though they would be otherwise correct. Such a situation is called race condition.

#include <unistd.h>

pid\_t fork();

Return: -1 on error;

0 in child process;

PID of the child in parent process

#include <sys/types.h>

#include <sys/wait.h>

pid\_t wait(int \*status);

pid\_t waitpid(pid\_t pid, int \*status, int options);

Return: -1 on error ; the PID of the terminated child;

* *fork* creates a child process that is a *clone* of the parent
* child has a (virtual) copy of the parent's virtual memory
* child is running the same program as the parent
* child *inherits* open file descriptors from the parent
* child begins life with the same register values as parent
* the child process may execute a different program in its context with a separate *exec* system call

Using fork:

if (fork() == 0) {

/\* child code \*/

exit(0);

}

else {

/\* parent code \*/

wait(0);

}

Example: Executia urmatorului program va produce 3 linii de output.

#include<stdio.h>

#include<unistd.h>

int main(int argc, char\*\* argv) {

printf(“before\n”);

fork(); //din acest moment avem 2 procese, cel initial (parinte) si cel nou creat (copil)

printf(“after\n”); //ambele procese executa acest cod

return 0;

}

Exemplu: Pentru a separa cadrul procesului parinte initial de cel nou creat (copil).

#include<stdio.h>

#include<unistd.h>

#include<stdlib.h>

#include<sys/types.h>

#include<sys/wait.h>

int main(int argc, char\*\* argv) {

int pid;

pid = fork();

if(pid == 0) {

printf(“Child-only code\n”);

exit(0);

}

printf(“Parent-only code\n”);

wait(0);

return 0;

}

**Zombie process**

It is natural that a parent process be interested in the execution status (exit code) of a child process it created). To get that state, the parent process uses wither the wait or waitpid system calls. What if the child process ends before the parent calls wait? The system keeps the child process in a zombie state: it doesn't execute anything, but it appears in the list of processes. A zombie process stops when the parent calls wait or waitpid. Zombie processes are problematic because if they are not "waited" they will grow in number, occupying PIDs and bringing the system to a point where no other processes can be created because of lack of PIDs.

**To avoid zombie processes**, always call **wait** for each child process you create. The system call wait waits for any child process to end. If you want to wait for a specific one, use waitpid. The argument for wait is a pointer to int, where it will return the child process exit code. But since we do not care for that value here, we just pass a zero (NULL).

Exemplu de zombie:

|  |
| --- |
| #include <stdio.h>  #include <string.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  int main(int argc, char\* argv[]) {  int i, pid;  if (fork()==0)  exit(0);  sleep(10);  return 0;  } |

Se ruleaza programul, apoi Ctr+Z (muta procesul parinte in background), apoi ps aux ne arata un process T (timeout) si unul Z (zombie sau <defunct>) pe langa procesul ps care are starea R (run).

Alte exemple:

|  |  |
| --- | --- |
| for(i=0; i<3; i++) {  pid = fork();  if(pid == 0) {  // do some stuff  exit(0);  }  }  // do some other stuff  for(i=0; i<3; i++) {  wait(0);  } | Ierarhia de procese:  P  / | \  C0 C1 C2  Procesul parinte creeaza 3 procese copil. Pot rula simultan 4 procese cu parintele. |
| void f(int n) {  if(n > 0) {  if(fork() == 0) {  f(n-1);  }  wait(0);  }  exit(0);  }  f(3); | Ierarhia de procese:  P  |  C3  |  C2  |  C1  Pot rula simultan 4 procese cu procesul parinte. |
| int main() {    printf("A %d %d\n", getppid(), getpid());    if(fork() == 0) {       printf("B %d %d\n", getppid(), getpid());       if(fork() == 0) {           printf("C %d %d\n", getppid(), getpid());           if(fork() == 0) {               printf("D %d %d\n", getppid(), getpid());               exit(0);           }           wait(0);           exit(0);      }      wait(0);      exit(0);  }  wait(0);  return 0;} | A (parent)  |  B (child)  |  C (grandchild to P)  |  D (grand-grandchild of P)  Fiecare process asteapta dupa procesul copil pe care l-a creat pentru a evita formarea proceselor zombie. Pot rula simultan 4 procese cu procesul parinte. |
| for(i=0; i<3; i++) {  pid = fork();  if(pid == 0) {  // do some stuff  exit(0);  }  else {  // do some other stuff  wait(0);  }  } | Ierarhia de procese:  P  / | \  C0 C1 C2  Atentie, procesele NU ruleaza simultan. Fiecare process copil se termina (parintele asteapta dupa finalizarwa lui) inainte de a se crea alt process copil. Maxim 2 procese ruleaza simultan. |
| **for(i=0; i<3; i++) {**  **pid = fork();**  **}** | Se creeaza 2^3=8 procese care pot rula simultan. Nu avem wait, vor fi zombies. |

**Calcul in parallel cu 2 procese, cu transmiterea rezultatelor partiale prin codul de exit.**

**Procesul parinte comunica cu procesul copil prin intermediul codului de retur.**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <unistd.h>  #include <sys/types.h>  #include <sys/wait.h>  int main(int argc, char\* argv[])  {  int a[] = {1,2,3,4};  int pid = fork(); // try to create a child process  if (pid == -1) // fork() has failed  {  perror("fork() error\n");  exit(EXIT\_FAILURE); //exit(1);  }  if (pid == 0) // in the child process  {  printf("[In CHILD] My PID is %d. My parent PID is %d.\n", getpid(), getppid());  int ps=a[0]+a[1];  printf("[In CHILD] Suma partiala este %d \n", ps);  exit(ps);  }  else // in the parent process  {  printf("[In PARENT] My PID is %d. My child PID is %d.\n", getpid(), pid);  int ps=a[2]+a[3];  printf("[In PARENT] Suma partiala este %d\n",ps);  int status;  wait(&status);  int suma=ps+WEXITSTATUS(status);//  printf("Suma totala calculata in paralel este %d .\n", suma);  }  return 0;  } |

**Execution of an external program. System calls exec**

#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[]);

#include <stdlib.h>

int system(const char \*command); //system launches a program on the disk, using fork() followed by exec() together with waitpid() in the parent.

Example 1:  
#include <stdio.h>  
#include <unistd.h>  
int main() {  
    char\* argv[3];  
    argv[0] = "/bin/ls";  
    argv[1] = "-l";  
    argv[2] = NULL;  
    execv("/bin/ls", argv);  
}

V-vector; L-list; p-absolute path;

Example 2:  
#include <stdio.h>  
#include <unistd.h>  
#include <stdlib.h> // system  
int main() {  
     //uncomment each of these lines one by one - only one of the 5  
     //execl("/bin/ls", "/bin/ls", "-l", NULL);  
     // execlp("ls", "ls", "-l", NULL);  
     // execl("/bin/ls","/bin/ls","-l","p1.c","execl.c", "fork1.c", "xx", NULL);  
     // execl("/bin/ls","/bin/ls","-l","\*.c", NULL); //will not be interpreted as expected    
     system("ls -l \*.c");  
}

|  |
| --- |
| Write a program that reads from the command line (arguments) a list of text files. Modify these files such that each word is capitalized. Use a separate process for each file. |
| #include <stdio.h>  #include <string.h>  #include <unistd.h>  #include <stdlib.h>  #include <sys/wait.h>  int main(int argc, char\* argv[]) {  int i, pid;  for (i=1; i<argc; i++) {  pid = fork();  if (pid == 0) {  execl("./capit.sh", "./capit.sh", argv[i], NULL);  printf("Error\n");  exit(1);  } else  printf("Parent launched child: %d> %s \n", pid, argv[i]);  }  for (i=1; i<argc; i++)  wait(0);  printf("Launched simultaneously %d processes for capitalization \n",  argc - 1);  return 0;  } |
| The capit.sh script is an executable, receives as first argument a filename:  #!/bin/sh sed "s/\<\([a-z]\)/\u\1/g" $1 > ${1}\_temp |
| Alternatively, we can write a C program to do the capitalization, compile it and launch the executable for each argument. |

**Program C care numara vocalele argumentelor primite la linia de comanda, lansand pentru fiecare argument un proces. Toate vor rula in paralel.**

alina@alina-Lenovo-IdeaPad-Y580:~/MIR/procese$ cat vocale.sh

#!/bin/bash

nr=`echo $1 | grep -E -o "[aeiou]" | wc -l`

echo $nr vocale in $1

-----------------------

alina@alina-Lenovo-IdeaPad-Y580:~/MIR/procese$ cat numaravocale.c

#include <stdio.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/wait.h>

int main(int argc, char \*\*argv){

int i;

for (i=1; i<argc; i++){

int pid=fork();

if (pid==0){

execl("./vocale.sh", "./vocale.sh", argv[i], NULL);

perror("Problema la exec ");

exit(0);

}

}

for(i=1; i<argc; i++)

wait(0);

return 0;

}

**Comunicarea intre procese folosind semnale**

**Functia kill trimite un semnal unul proces**

**Functia signal seteaza un handler la un semnal (schimba ce se intampla cand apare un semnal).**

alina@alina-Lenovo-IdeaPad-Y580:~/MIR/procese$ cat fork1.c

#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>

#include <stdlib.h>

#include <sys/wait.h>

void f(int s){ // functia handler, se executa cand se primeste semnalul

printf("Procesul copil nu se poate inchide cu SIGINT HAHAHA\n");

}

int main(){

int p=fork(); // in parinte fork returneaza pid-ul copilului; in procesul copil fork returneaza 0

if (p==-1){ // in caz de eroare fork returneaza -1

perror("Eroare fork");

exit(1);

}

if (p>0){

// procsul parinte

printf("Procesul parinte cu id %d si parintele id %d.\n", getpid(), getppid());

}

else {

//procesul copil

//printf("Procesul copil cu id %d si parintele id %d. \n", getpid(), getppid());

//sleep(5);

signal(SIGINT, f); //seteaza handlerul de semnal SIGINT (echivalent CTR+C)

while (1){

sleep(1);

printf("Procesul copil ruleaza non stop haha \n");

}

exit(0);

}

sleep(2);

kill(p, SIGINT); // trimite semnalul de intrerupere catre procesul copil

wait(0);

printf("Hello\n");

return 0;

}