METODE AVANSATE DE GESTIUNE A DOCUMENTELOR ȘI A SISTEMELOR DE CALCUL
- CURS 6

Asist. Diana – Florina Șotropa

www.cs.ubbcluj.ro/~diana.sotropa

Expresii regulare – Caractere speciale

- `.`
  - `a.c` => *abc adc aec a=c a:c*
  - `x..x` => *xaax xavx x=kx*
- `*`
  - `ab*c` => *ac abc abbc abbbbbbbbbbbbbbbc*
  - `a*` => "" *a aa aaaaaaaaaa*
  - `a*b*c*` => ?
  - `.*` => ?

Expresii regulare – Caractere speciale

- []
  - `[Mm]ark` => *mark Mark*
  - `t[aeiou]x` => *tax tex tix tox tux*
  - `[abc].*` => orice incepe cu a or b or c
  - `[a-z][a-z]` => orice sir de caractere care contine cu doua litere mici
  - `[a-zA-Z]*` => orice sir de caractere format doar din litere mari si mici
  - `[^abc].*` => orice sir de caractere care contine alte caractere in afara de a,b,c
  - `[a-zA-Z0-9_]*` => ?

Expresii regulare – Caractere speciale

- ^
  - `^T` => liniile care incep cu T
  - `^[0-9]` => ?
- $
  - `T$` => liniile care se termina cu T
  - `^$` => ?
- \
  - `\.` => .
  - `a\*b` => a*b

Exemple

- `[0-9][0-9][0-9]-[0-9][0-9]-[0-9][0-9][0-9][0-9]` =>?
- `[a-zA-Z_][a-zA-Z0-9_]*` =>?

- This (rug) is not what it once was (a long time ago), is it?
  - `Th.*is` =>?
  - `(.*)` =>?

Expresii regulare complexe – folosite cu egrep sau grep -E

- +
  - `ab+c` => *abc abbc abbbc*, dar nu si pe *ac*
  - `..* = .+`
- ?
  - `ab?c` => *ac abc*
- |
  - `abc|def` => *abc def*
- ( )
  - `ab(c|d)ef` => *abcef abdef*
  - `ab(cd|de)fg` => *abcdfg abdefg*

Expresii regulare complexe – folosite cu egrep sau grep -E

- \{ \}
  - `[0-9]\{3\}-[0-9]\{2\}-[0-9]\{4\}` => 3 cifre – 2 cifre – 4 cifre
  - `a\{4,\}` => cel putin 4 de a
  - `[a-z]\{3,5\}` => cel putin 3 litere mici si cel mult 5

Exemplu

Structura de directoare:

```
a
aa
aaa
ab
aba
abb
abc
abd
abe
ac
aca
ad
ada
ae
aea
b
c
d
e
bba
aaabbbb
```

```
ls | grep 'abc'

ls | grep 'a..'

ls | grep 'a.*'

ls | grep 'a[ab].?'

ls | egrep 'a[ab].?'

ls | grep '[^a]'

ls | grep '^[^a]$'
```

Exemple

- Considerand textul scris pe cele 4 linii:

```
Flip is a file interchange program that converts text file
formats between **IX and MS-DOS.  It converts lines ending
with carriage-return (CR) and linefeed (LF) to lines ending
with just linefeed, or vice versa.
```

- Ce se afiseaza in urma unui grep care foloseste urmatoarele expresii regulare?

- in
- [R-Z]
- ^[Ff]
- .$
- ee*
- *
- lines\{0,\}
- [Cc].*[Ff]
- \(.\{2\}\)
- [Ii][acX][^a-f]
- F[^]+
- Line\(s|[^s]+\)
- v.*e
- [a-z]*[e.]$
- *+

Exemple Sed

- `sed 's/index1/index2/g' main.c`
- `sed -n '20,30p' file`
- `sed '1,10d' file`
- `sed '$d' file`
- `sed 's/^\([A-Z][a-z-]*\)[,][ ]\([A-Z][a-z-]*\)$/\2 \1/' file`
- `sed '10,20w newfile' file`
- `sed '1,/^$/d' file`
- `sed -n '/^$/,/^end/p' file`
- `sed '/one/d   /two/d' file`

Exemple Sed

faculty.details:

```
Name: Mehdi Zargham Office: 139 Anderson Hall Course: ASI 150

Name: Raghava Gowda Office: 142 Anderson Hall Course: CPS 310

Name: James P. Buckley Office: 146 Anderson Hall Course: CPS
430/530

Name: Dale Courte Office: 144 Anderson Hall Course: CPS 387

Name: Saverio Perugini Office: 145 Anderson Hall Course: CPS
444/544

Name: Zhongmei Yao Office: 150 Anderson Hall Course: CPS 341
```

-
  ```
  sed -n '/CPS/p' faculty.details
  # same as grep CPS faculty.details
  # same as sed '/CPS/!d' faculty.details
  ```

-
  ```
  sed -n '/[/]/p' faculty.details
  # prints lines with a cross-listed course;
  # same as sed -n '/\//p' or grep '\/' faculty.details
  ```

-
  ```
  sed '/\//d' faculty.details
  # print lines containing a non-cross-listed course;
  # same as grep -v '\/' faculty.details
  ```

-
  ```
  sed 's/^Name:[ ]//' faculty.details
  # removes "Name: " from file faculty.details
  ```

-
  ```
  sed 's/^Name:[ ]//' faculty.details | sed 's/Office:[ ]//'
  # removes "Name: " & "Office: " from faculty.details
  ```

-
  ```
  sed 's/[A-Za-z][A-Za-z]*: //g' faculty.details
  # purge all attribute labels (i.e., "Name: ", "Office: ")?
  ```

Exemple Sed

faculty.details:

```
Name: Mehdi Zargham Office: 139 Anderson Hall Course: ASI 150

Name: Raghava Gowda Office: 142 Anderson Hall Course: CPS 310

Name: James P. Buckley Office: 146 Anderson Hall Course: CPS
430/530

Name: Dale Courte Office: 144 Anderson Hall Course: CPS 387

Name: Saverio Perugini Office: 145 Anderson Hall Course: CPS
444/544

Name: Zhongmei Yao Office: 150 Anderson Hall Course: CPS 341
```

-
  ```
  sed 's/[A-Za-z]\{1,\}: //g' faculty.details
  ```

-
  ```
  sed 's/^Name:[ ]//' faculty.details | sed 's/Office:[ ]//' |
  sed 's/Course:[ ]//'
  # purges all attribute labels
  ```

-
  ```
  sed 's/^Name:[ ]//;
       s/Office:[ ]//;
       s/Course:[ ]//' faculty.details
  ```

-
  ```
  cat sedfile
  s/^Name:[ ]//
  s/Office:[ ]//
  s/Course:[ ]//
  ```

-
  ```
  sed -f sedfile faculty.details
  ```

-
  ```
  sed 's/^Name:[ ]\(.*\)Office:[ ]\(.*\)Course:[
  ]\(.*\)$/\1\2\3/' faculty.details
  ```

-
  ```
  sed 's/[A-Za-z][A-Za-z]*:[ ]//g' faculty.details
  ```

Exemple Sed

faculty.details:

```
Name: Mehdi Zargham Office: 139 Anderson Hall Course: ASI 150

Name: Raghava Gowda Office: 142 Anderson Hall Course: CPS 310

Name: James P. Buckley Office: 146 Anderson Hall Course: CPS
430/530

Name: Dale Courte Office: 144 Anderson Hall Course: CPS 387

Name: Saverio Perugini Office: 145 Anderson Hall Course: CPS
444/544

Name: Zhongmei Yao Office: 150 Anderson Hall Course: CPS 341
```

- `sed 'd' faculty.details`
  `# reads in one line at a time into a buffer (work space), deletes it, and prints the contents of the buffer (in this case, empty)`
- `sed '1d' faculty.details`
  `# reads in one line at a time into the buffer, deletes it if it is line 1, and prints the buffer contents onto output (in this case, all lines except 1 would be output)`
- `sed '$d' faculty.details`
  `# does the same, but for the last line`
- `sed '2,4d' faculty.details`
  `# deletes lines from 2 up to and including line 4`
- `sed '/Yao/,/ran/d' faculty.details`
  `# deletes lines starting from one which matches Yao up to and including one which matches ran`
- `sed '/Yao/,/ran/!d' faculty.details`
  `# negates the address (i.e., do not delete these lines, and delete others)`

Exemple Sed

faculty.details:

```
Name: Mehdi Zargham Office: 139 Anderson Hall Course: ASI 150

Name: Raghava Gowda Office: 142 Anderson Hall Course: CPS 310

Name: James P. Buckley Office: 146 Anderson Hall Course: CPS
430/530

Name: Dale Courte Office: 144 Anderson Hall Course: CPS 387

Name: Saverio Perugini Office: 145 Anderson Hall Course: CPS
444/544

Name: Zhongmei Yao Office: 150 Anderson Hall Course: CPS 341
```

- `sed 'p' faculty.details`
  `# reads in one line at a time into the buffer and prints each. Notice that by default sed prints what is in the buffer. Therefore, you will get two copies of each line.`
- `in sed -n 'p' faculty.details`
  `# the -n suppresses the default print action of sed. Therefore, this is the equivalent of doing a cat.`
- `sed -n 4,6 'p' faculty.details`
  `# we can use the same addressing commands as before (e.g., prints lines 4 through 6).`

PROCESE IN UNIX

Procese in UNIX

- Program vs. Proces
  - Programul este reprezentat de un grup de instructiuni care rezolva un anumit task
  - Procesul este un program in executie
- Un program poate invoca mai multe procese
- Program vs. Proces
  - Programul e stocat pe disk intr-un fisier si nu necesita resurse suplimenare
  - Procesul necesita resurse suplimentare (CPU, memorie, I/O)

Procese in UNIX

- Crearea unui proces nou:
  - `fork()`
- Comunicare intre procese
  - `pipe()`
    - se foloseste pentru a transmite informatia de la un proces la altul
    - pipe este unidirectional
    - pentru a comunica in ambele sensuri trebuie sa fie definite doua pipe-uri
  - `mkfifo()`
    - Orice process poate deschide fifo-ul pentru a scrie sau a citi exact in acelasi fel in care se deschide un fisier
    - However, it has to be open at both ends simultaneously before you can proceed to do any input or output operations on it.

Procese in UNIX – fork()

- `int fork()`
  - Creeaza un nou proces, numit proces copil care ruleaza in paralel cu procesul parinte;
  - Dupa ce s-a creat un process copil, ambele procese vor executa instructiunea urmatoare apelului system fork()
  - Apelul system fork() nu are parametri si returneaza un intreg
    - <0 – procesul copil nu a putut fi creat
    - =0 – sunt in procesul fiu
    - >0 – sunt in procesul parinte
  - Proces parinte => PID proces parinte
  - Proces fiu => PID proces fiu

Procese in UNIX – fork()

- Ce se va afisa in cazul executiei urmatorului program?

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    fork();

    printf("Hello world!\n");
    return 0;
}
```

```
Hello world!
Hello world!
```

Procese in UNIX – fork()

- getpid() – returneaza pid-ul procesului copil

  pid_t getpid(void);

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

int main()
{
    int pid = fork();
    if (pid == 0)
printf("\nCurrent process id of Process : %d",getpid());
    return 0;
}
```

Procese in UNIX – fork()

- getppid() – returneaza pid-ul procesului parinte

  `pid_t getppid(void)``;`

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

int main()
{
    int pid;

    pid = fork();
    if (pid == 0)
    {
printf("\nChild Process id : %d ", getpid());
printf("\nChild Process with parent id : %d", getppid());
    }
    else {
printf("\nParent Process id : %d ", getpid());
printf("\nParent Process with parent id : %d", getppid());
}
    return 0;
}
```

Procese in UNIX – fork()

- getppid() – returneaza pid-ul procesului parinte

  `pid_t getppid(void);`

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

int main()
{
    int pid;

    pid = fork();
    if (pid == 0)
    {
printf("\nChild Process id : %d ", getpid());
printf("\nChild Process with parent id : %d", getppid());
    }
    else {
printf("\nParent Process id : %d ", getpid());
printf("\nParent Process with parent id : %d", getppid());
wait(10);}
    return 0;
}
```

Procese in UNIX – fork()

```
#include<stdio.h>
#include <unistd.h>
main()
{
    pid_t pid;
    printf("Hello World1\n");
    pid=fork();
    if(pid==0)
    {
        printf("I am the child\n");
        printf("The PID of child is %d\n",getpid());
        printf("The PID of parent of child is %d\n",getppid());
    }
    else
    {
        printf("I am the parent\n");
        printf("The PID of parent is %d\n",getpid());
        printf("The PID of parent of parent is %d\n",getppid());
    }
}
```

Procese in UNIX – fork()

- De cate ori se afiseaza cuvantul hello?

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

- Procesul parinte P

`fork()`

- se executa in parinte => se creeaza copilul C1

`fork()`

- se executa in parinte si in C1 => se creeaza copiii C21 si C22

`fork()`

- se executa in parinte si in copiii C1, C21 si C22 => se creeaza copiii C31, C32, C33, C34

Prin urmare exista procesele P, C1, C21, C22, C31, C32, C33, C34

Procese in UNIX – fork()

- De cate ori se afiseaza cuvantul hello?

```
#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

```
hello
hello
hello
hello
hello
hello
hello
hello
```

- Numarul de cuvinte hello afisate este egal cu numarul de procese create.
- Numarul total de procese este 2^n, une n este numarul de apeluri system fork()
- In acest caz n = 3, 2^3 = 8

Procese in UNIX – fork()

- Ce se afiseaza?

```
Hello from Child!
Hello from Parent!
     (or)
Hello from Parent!
Hello from Child!
```

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    if (fork() == 0)
        printf("Hello from Child!\n");
    else
        printf("Hello from Parent!\n");
}
int main()
{
    forkexample();
    return 0;
}
```

Procese in UNIX – fork()

- Ce se afiseaza?

```
Parent has x = 0
Child has x = 2
     (or)
Child has x = 2
Parent has x = 0
```

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

void forkexample()
{
    int x = 1;

    if (fork() == 0)
        printf("Child has x = %d\n", ++x);
    else
        printf("Parent has x = %d\n", --x);
}
int main()
{
    forkexample();
    return 0;
}
```

Procese in UNIX – fork()

- Ce se afiseaza?

```
fork
..before fork
..after fork, a = 7, b = 89
..after fork, a = 6, b = 88
```

```c
#include <sys/types.h>
#include <stdio.h>
int a = 6;

int main(void)
{
 int b;
 pid_t pid;
 b = 88;
 printf("..before fork\n");
 pid = fork();
 if (pid == 0)
     {a++; b++;}
 else wait(pid);
 printf("..after fork, a = %d, b = %d\n", a, b);
 return 0;
}
```

Procese in UNIX – fork()

- Cate procese copil se creeaza?

```
for (i = 0; i < n; i++) fork();
```

a) n
b) 2^n -1
c) 2^n
d) 2^(n+1)-1

- Procesul parinte P

`fork() => 1 copil`

- se executa in parinte => se creeaza copilul

`fork() => 2 copii`

- se executa in parinte si in C1 => se creeaza copiii C21 si C22

`fork() => 4 copii`

- se executa in parinte si in copiii C1, C21 si C22 => se creeaza copiii C31, C32, C33, C34

...

Prin urmare exista procesele P, C1, C21, C22, C31, C32, C33, C34, ... adica 2^n procese din care 2^n -1 sunt copii

Procese in UNIX – fork()

- Procese zombie
  - Proces care si-a terminat executia dar inca are o intrare in tabela de procese deoarece raporteaza procesului parnte
  - Un process fiu intotdeauna devine zombie inainte sa fie scos din tabela de procese
  - Procesul parinte citeste statusul de exit al copilului si scoate procesul fiu din tabela de procese

```
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    pid_t child_pid = fork();

    if (child_pid > 0)
        sleep(50);
    else
        exit(0);

    return 0;
}
```

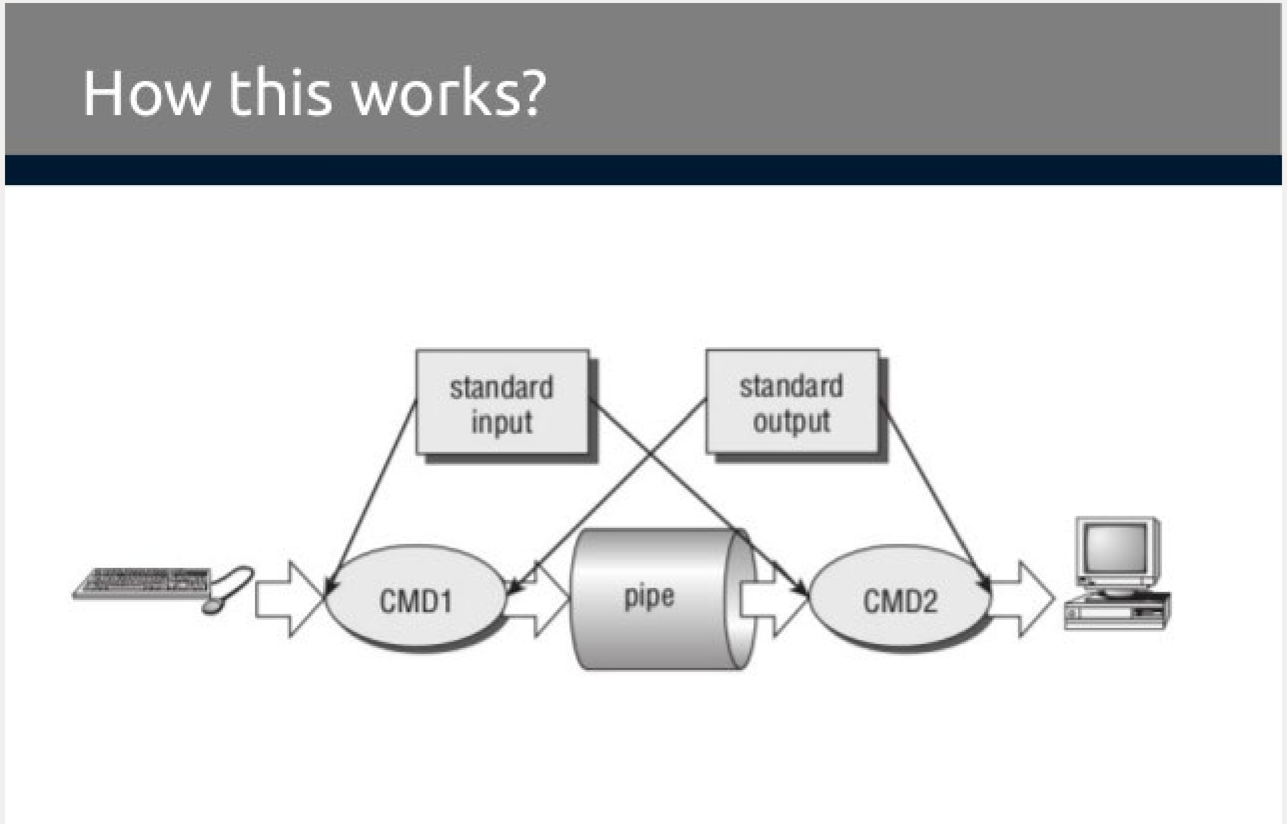
Procese in UNIX – fork()

- Procese orfan
  - Proces al carui parinte nu mai exista, adica si-a terminat executia fara sa astepte ca procesul copil sa isi termine executia

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

int main()
{
    int pid = fork();

    if (pid > 0)
        printf("in parent process");

    else if (pid == 0)
    {
        sleep(30);
        printf("in child process");
    }

    return 0;
}
```

Procese in Unix – pipe()

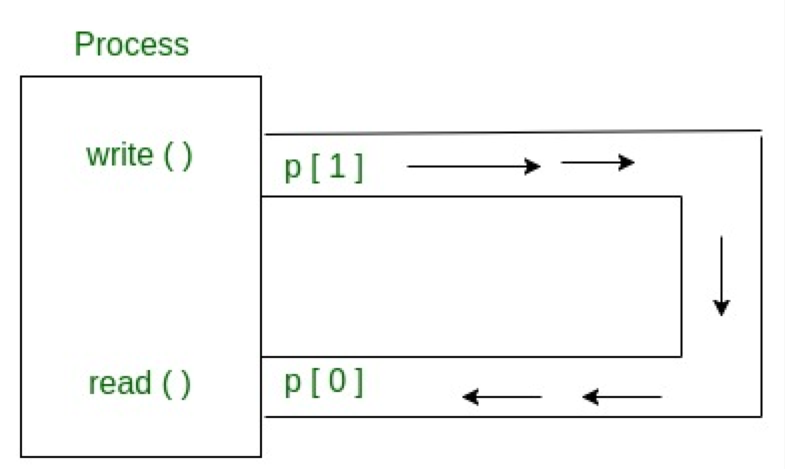
- Termenul de pipe se foloseste pentru a atasa output-ul unui process ca input pentru altul
- `Ex. cmd1 | cmd2`
- `Ex. ls | wc`
- `Ex. who | sort`
- `Ex. cat file.txt | sort | wc`



How this works?

Procese in Unix – pipe()

- Pipe reprezinta o conexiune intre doua procese si se utilizeaza pentru comunicarea intre procese
- Pipe se foloseste pentru comunicare uni directionala, astfel incat un proces sa scrie in pipe si celalalt sa citeasca
- Un pipe este o zona de memorie tratata ca un fisier virtual
- Pipe-ul poate fi folosit de catre procesul parinte sau de catre procesul fiu
- Daca un process incearca sa citeasca din pipe inainte sa fie scris ceva in pipe procesul se suspenda pana cand se scrie ceva in pipe
- Se poate folosi doar intre procese inrudite (care au un stramos comun)



Procese in Unix – pipe()

- `int pipe(int fd[2])`
  - Parametri:
    - fd[o] – descriptorul de fisiere pentru capatul de citire din pipe
    - fd[1] – descriptorul de fisiere pentru capatul de scriere in pipe
  - Trebuie inclus header-ul #include <unistd.h>
  - Datele sunt procesate pe principiul FIFO (first in first out), adica daca se scriu octetii 1,2,3 in fd[1] atunci se vor citi din fd[o] octetii 1,2,3

Procese in Unix – pipe()

- `size_t write(int fildes, const void *buf, size_t nbytes)`
  - `#include <unistd.h>`
  - Primii nbytes din buf vor fi scrisi in fisierul care are descriptorul fildes
  - Functia returneaza numarul de octeti scrisi

Procese in Unix – pipe()

- `size_t read(int fildes, void *buf, size_t nbytes)`
  - `#include <unistd.h>`
  - nbytes din buf vor fi cititi din fisierul care are descriptorul fildes
  - Functia returneaza numarul de octeti cititi

Procese in UNIX – pipe()

- Ce se afiseaza?

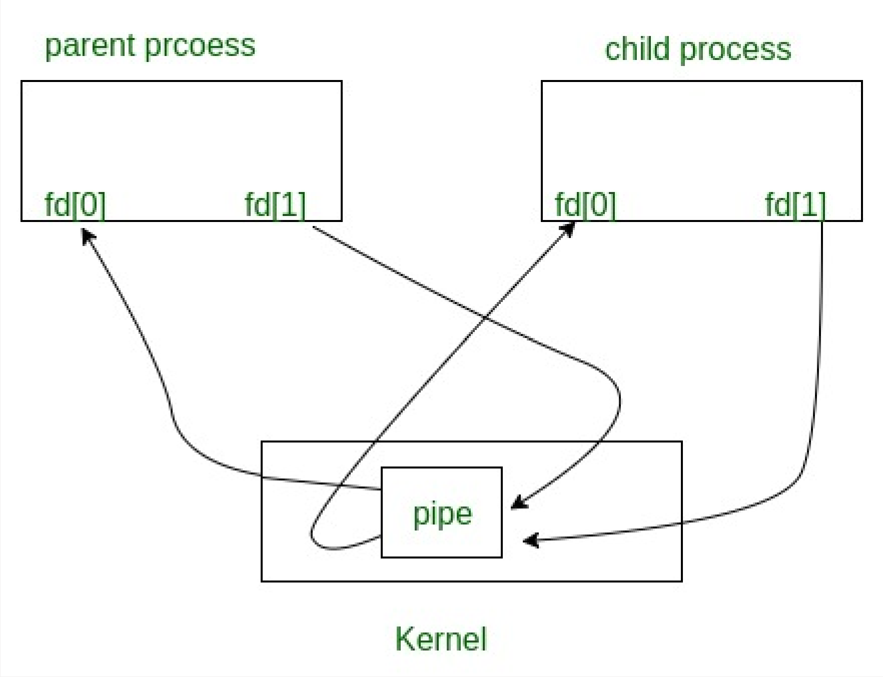
```
hello, world #1
hello, world #2
hello, world #3
```

```c
#include <stdio.h>
#include <unistd.h>
#define MSGSIZE 16
char* msg1 = "hello, world #1";
char* msg2 = "hello, world #2";
char* msg3 = "hello, world #3";

int main()
{
    char inbuf[MSGSIZE];
    int p[2], i;

    if (pipe(p) < 0)
        exit(1);

    write(p[1], msg1, MSGSIZE);
    write(p[1], msg2, MSGSIZE);
    write(p[1], msg3, MSGSIZE);
```

```c
for (i = 0; i < 3; i++) {
        read(p[0], inbuf, MSGSIZE);
        printf("%s\n", inbuf);
    }
    return 0;
}
```

Procese in Unix – pipe()

- Daca se foloseste fork() intr-un proces, descriptorii de fisier raman deschisi atat in procesul fiu cat si in procesul parinte
- Daca apelul fork() are loc dupa crearea pipe-ului, atunci parintele si copilul pot comunica prin pipe



Procese in Unix – pipe()

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <string.h>
#include <sys/wait.h>
int main()
 {
  int fd1[2], fd2[2];

  char fixed_str[] = " la GDSC";
  char input_str[100];
  pid_t p;

  if (pipe(fd1) == -1) {
    fprintf(stderr, "Pipe Failed");return 1;
  }

  if (pipe(fd2) == -1) {
    fprintf(stderr, "Pipe Failed");return 1;
  }

  scanf("%[a-zA-Z0-9 ]s", input_str);
  p = fork();
  if (p < 0) {
    fprintf(stderr, "fork Failed");
    return 1;
  }
```

```
// Parent process
  else if (p > 0) {
    char concat_str[100];
    close(fd1[0]);
    write(fd1[1], input_str, strlen(input_str)+1);
    close(fd1[1]);
    wait(NULL);
    close(fd2[1]);
    read(fd2[0], concat_str, 100);
    printf("Concatenated string %s\n", concat_str);
    close(fd2[0]);
  }

 // child process
  else {
    close(fd1[1]);
    char concat_str[100];
    read(fd1[0], concat_str, 100);
    int k = strlen(concat_str);
    int i;
    for (i = 0; i < strlen(fixed_str); i++)
      concat_str[k++] = fixed_str[i];
    concat_str[k] = '\0';
    close(fd1[0]);
    close(fd2[0]);
    write(fd2[1], concat_str, strlen(concat_str) + 1);
    close(fd2[1]);
    exit(0);
  }}
```

Procese in Unix – mkfifo()

- `int mkfifo(const char *pathname, mode_t mode);`
- mkfifo() creeaza un fisier special numit FIFO cu numele ***pathname***.
- Parametrul ***mode*** specifica permisiunile asupra fisierului
- Fisierul FIFO se foloseste ca orice alt fisier; prin urmare apelurile system de lucru cu fisiere pot fi folosite: *open*, *read*, *write*, *close*.
- Comunicarea poate fi bidirectionala;
- Nu este necesara prezenta unui process fiu si a unui process parinte, comunicarea putandu-se face intre mai mult de doua procese

Proces in Unix – mkfifo()

```
// scrie primul si apoi citeste
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main()
{
int fd;

char * myfifo = "tmp/myfifo";
mkfifo(myfifo, 0666);
char arr1[80], arr2[80];
while (1) {
  fd = open(myfifo, O_WRONLY);
  fgets(arr2, 80, stdin);
  write(fd, arr2, strlen(arr2) + 1);
  close(fd);

  fd = open(myfifo, O_RDONLY);
  read(fd, arr1, sizeof(arr1));
  printf("User2: %s\n", arr1);
  close(fd);
}
```

```
// citeste primul si apoi scrie
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

int main() {

    int fd1;

    char * myfifo = "/tmp/myfifo";
    mkfifo(myfifo, 0666);
    char str1[80], str2[80];
    while (1) {
      fd1 = open(myfifo, O_RDONLY);
      read(fd1, str1, 80);
      printf("User1: %s\n", str1);
      close(fd1);
      fd1 = open(myfifo, O_WRONLY);
      fgets(str2, 80, stdin);
      write(fd1, str2, strlen(str2) + 1);
      close(fd1);
    }
```