OPERATING SYSTEMS CSCI-SHU215

Lecture 04 - Processes

Notion of Process

Definition (C. Girault)

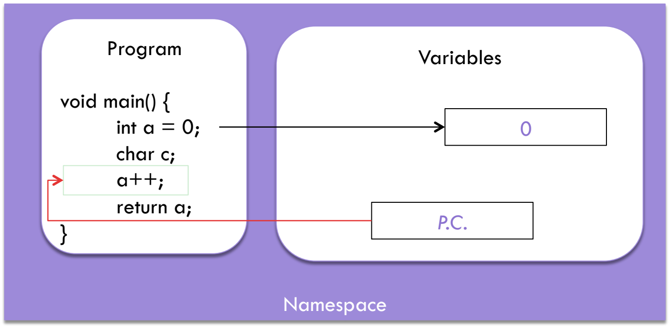
A process is the sequential execution of the instructions of a program in a Namespace

Namespace

- code
- stack
- shared variables and constants
- files

Multiple runs of the same program in different namespaces produce different processes

Notion of Process



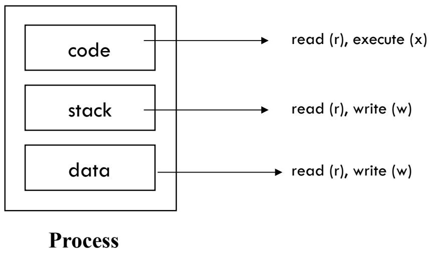
One register, the program counter, stores the current running instruction

Notion of Process

A process is an active entity of the system

- Corresponds to the execution of a binary program
- Identified in a unique way by its pid number
- Has 3 segments: code, data, and stack
- Runs under the identity of a user
- Has a current directory

Notion of Process



code → read (r), execute (x)

stack → read (r), write (w)

data → read (r), write (w)

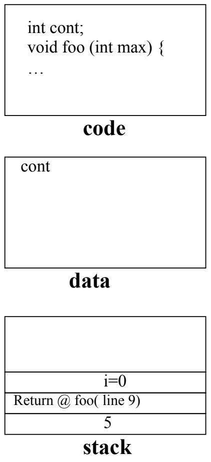
Process

Each process is independent

Two processes can be associated with the same program (code)

Program Execution

```
1:    int cont;

2:    void foo (int max) {
3:            int i;
4:            for (i=0; i++; i<max)
5:                 printf ("%d \n", i);
6:    }

7:    int main (int argc, char* argv []) {
8:            int cont=5;
9:            foo(cont) ;
10:           return EXIT_SUCCESS;
11:    }
```



int cont;
void foo (int max) {
…

code

cont

data

i=0

Return @ foo(line 9)

5

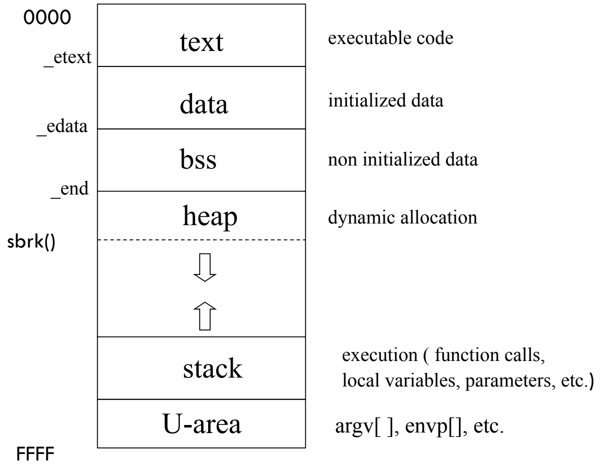
stack

Data Segment

- Data
  - Data initialized upon loading the process
- BSS (Block Started by Symbol)
  - Non initialized data
- Heap
  - Dynamically allocated memory areas
    - eg. `malloc(), calloc()`

The heap and the stack grow in opposite directions

Namespace of a process



0000

_etext

_edata

_end

sbrk()

FFFF

text

executable code

data

initialized data

bss

non initialized data

heap

dynamic allocation

⇩

⇧

stack

execution (function calls, local variables, parameters, etc.)

U-area

argv[ ], envp[], etc.

Dynamic allocation of memory

```
#include <stdlib.h>
```

```
void * malloc(size_t size)
```

allocates a block of *size* bytes

```
void * calloc(size_t nb, size_t size)
```

allocates a block of *nb* * *size* bytes (initially set to 0)

```
void * realloc(void* ptr, size_t size)
```

resizes a block of memory (preserves content)

```
void free (void * ptr)
```

frees the allocated memory

States of a process

At runtime, a process switches states

- **Running**

  process instructions are being executed

- **Suspended**

  process is waiting for a resource to become available

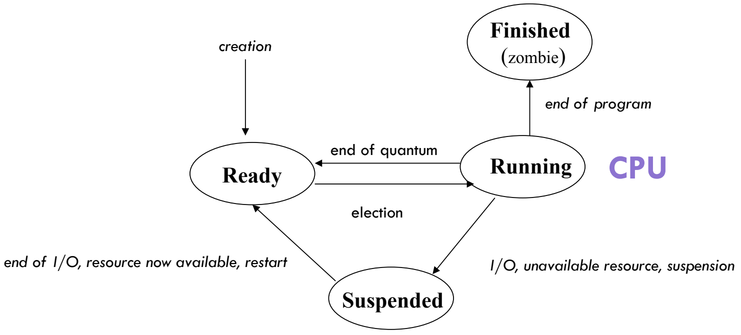
- **Ready**

  process waits to be assigned to a processor

- **Zombie**

  process has finished running, but its parent has yet to acknowledge its termination

States of a process



Quantum: unit of duration (e.g. 100 ms)

Attributes of a process

Identity of a process

pid: positive integer (type POSIX *pid_t*)

```
pid_t getpid (void)
```

returns the pid of a process

Exemple:

```
#define _XOPEN_SOURCE 700
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>

int main(int argc, char **argv) {
  printf (" process pid is: %d \n", getpid()) ;
  return EXIT_SUCCESS;
}
```

Attributes of a process

A process is tied to a user and its group

- UID (User identifier) & GID (group identifier)
  - Identities associated with access permissions by the kernel
- Permissions
  - Rights granted to the user (group) that starts the program
- Effective rights
  - Rights granted to the program itself
    - identity that the kernel takes into account to check the access permissions for operations requiring an identification
    - eg. opening a file, making a system call

Process Control Block (PCB)

Information associated with each process

- State of the process
- ID of the process
- Program Counter (PC)
- CPU registers
- CPU scheduling info
- Memory management info
- Accounting info
- I/O info

Process creation with fork

Fork - creation of a process

Primitive *pid_t fork (void)*

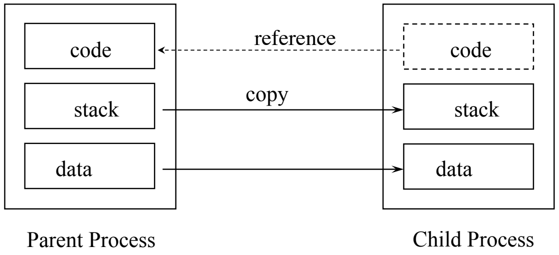
- Dynamic creation of a new process (*child*)
- The child is executed concurrently with the process that created it (*parent*)

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork (void)
```

- Created child process is a copy of the parent process

Fork - creation of a process

- Both processes share the same physical code
- Duplication of the stack and data segment
  - variables of the child have the same values as those of the parent upon *fork* call;
  - any value modification of a variable by one of the processes is not visible to the other



Fork - creation of a process

One single call, but two return values

Each process resumes its execution

Return values differ

0	returned to the child
child process pid	returned to the parent
-1	system call failed

errno <errno.h>:

ENOMEM	not emough memory available
EAGAIN	too many processes created

pid_t getppid (void)

returns the pid of the parent

Fork - creation of a process

- Example 1

test-fork1.c

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

int main (int argc, char* argv []) {
      pid_t pid_child;
      switch (pid_child = fork ( ) ) {
        case (pid_t) -1:
              perror ("fork"); exit (1);
        case  (pid_t) 0:
              printf ("CHILD> pid %d, parent pid %d \n", getpid (), getppid () );
              return EXIT_SUCCESS;
        default:
              printf ("PARENT> pid %d, child pid %d \n", getpid(), pid_child);
              return EXIT_SUCCESS;
      }
}
```

Fork - creation of a process

Concurrency

- Child gets to run before its parent

```
$test-fork1
CHILD> pid 1528, parent pid 1476
PARENT> pid 1476, child pid 1528
```

- Parent gets to run before its child

```
$test-fork1
PARENT> pid 1476, child pid 1528
CHILD> pid 1528, parent pid 1
```

Child becomes an orphan, gets adopted by init process (pid = 1)

Fork - data duplication

Example 2:

test-fork2.c

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

int main (int argc, char* argv []) {
        int a= 3; pid_t pid_child;
        a *=2;
        if ( (pid_child = fork ( ) )  == -1 ) {
                perror ("fork"); exit (1);
        } else
                if (pid_child == 0) {
                        a=a+3;
                        printf ("child a=%d \n", a);
        } else
                printf ("parent a=%d \n", a);
        return EXIT_SUCCESS;
}
```

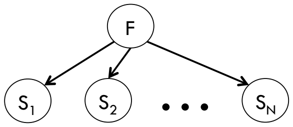
```
$test-fork2
   child a=9
   parent a=6

$test-fork2
   parent a=6
   child a=9
```

Fork - looped calls

Example 3: a process creates N childs

test-fork3.c

```
#include <sys/types.h>
#include <unistd.h>
#define N 3
int main (int argc, char* argv []) {
   int i=0;  pid_t pid;
     while (i <N) {
        if (((pid=fork ( ) )== -1) || (pid==0))
            exit (1);
        i++;
      }
     return EXIT_SUCCESS;
}
```

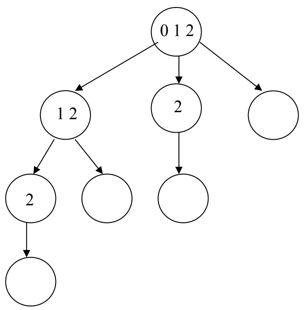


Fork - looped calls

Example 4: How many processes at the end?

test-fork4.c

Creation tree



```
#define _XOPEN_SOURCE 700
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main (int argc, char* argv []) {
    int i =0 ;
    while (i <3) {
        printf ( "%d ", i);
        i ++;
        if (fork ( ) == -1)
                exit (1);
    }
    printf( "\n ");
    return EXIT_SUCCESS;
}
```

Fork - Inheritance

A child process inherits

- User ID and group ID (real and effective)
- Session ID
- Current working directory
- Current umask bits, signal mask, signal routines
- Attached shared memory
- Environment variables
- Open file descriptors
- ***nice*** value
- …

Fork - Inheritance

A child process does not inherit

- Identity (pid) of its parent process
- Execution time
- Pending signals
- File locks held by the parent
- Alarms and timers

functions `alarm, setitimer,...`

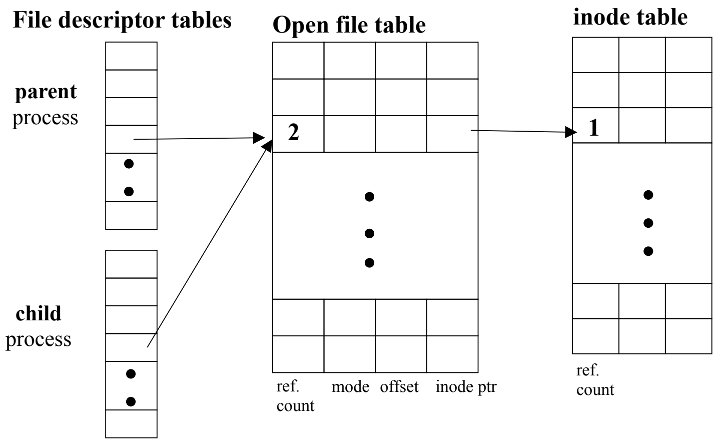
Fork - Inheritance vs dynamic memory

test_fork5.c

```
#DEFINE SIZE 100
char* ptr1=NULL;
char* ptr2 =NULL;
int main (int argc, char* argv []) {
   pid_t pid;
   ptr1= (char*)malloc(SIZE);
   memcpy(ptr1, (char*)"toto", strlen("toto"));
      if ((pid=fork ( ) )== -1)
         exit (1);
      else
        if (pid != 0){
          /*parent */
          ptr2= (char*) malloc(SIZE);
         memcpy(ptr2, (char*) "titi", strlen("titi"));
         printf ("PARENT - ptr1:%s ; ptr2:%s \n", ptr1, ptr2);
        }
       else
           printf ("CHILD - ptr1:%s ; ptr2:%s \n", ptr1, ptr2);

      return 0;
}
```

```
$test_fork5
PARENT - ptr1: toto; ptr2:titi
CHILD - ptr1: toto; ptr2:
```

Fork - Inheritance vs file descriptors



End of a process

Process termination

exit(int val) or *return val*

- val is a value that the parent can acquire

- Pre-defined constants
  - EXIT_SUCESS
  - EXIT_FAILURE

- If a process started by the shell terminates with an error
  - error code available in variable $?
    - *echo $?*

Process termination

When a process terminates correctly

- All I/O streams get closed
  - Buffers are emptied

- Call to _*exit()* (system call)
  - closes open file descriptors
  - parent process received a signal SIGCHILD

- The terminated process becomes a *zombie*

Process termination

Zombie process

- Temporary state of a process until its parent acknowledges its termination

Parent/Child Synchronization

- Upon termination, either with an *exit* call or with a *return* instruction in the main function, a process assigns a value to its return code
- The parent process can access this value by calling function *wait*/*waitpid*

Basic parent/child synchronization

Basic parent/child synchronization

Primitive pid_t wait (int* status)

```
#include <sys/types.h>
#include <sys/wait.h>
 pid_t wait (int* status)
```

If the calling process

- has at least one zombie child
  - call returns the identity of one of the zombies
  - recovery of the return code value in variable `status`
- has a child, but none is in a zombie state
  - process is blocked until one of its children becomes zombie
  - notification by a signal `SIGCHLD`
- does not have any children
  - the call returns-1 and the value of `errno` becomes `ECHILD`

Basic parent/child synchronization

Interpretation of the return value *`int *status`*

Pre-defined ***macros*** to solve portability issues

Type of termination

- WIFEXITED: not NULL if the child process terminated normally
- WIFSIGNALED: not NULL if the child process terminated because of a signal
- WIFSTOPPED: not NULL if the child process is stopped (option WUNTRACED waitpid)

Information on the return value or the signal

- WEXITSTATUS: return code if the process ended normally
- WTERMSIG: value of the signal that caused the process termination
- WSTOPSIG: value of signal that stopped the process

Basic parent/child synchronization

include files: stdio.h, sys/types.h, sys/wait.h, unistd.h, stdlib.h

test-wait.c

```
int main(int argc, char **argv) {
    pid_t pid_child; int status;
    if (fork () == 0) {
        printf ("CHILD:  pid = %d \n", getpid());
        exit (2);
    } else {
        pid_child = wait(&status);
        if (WIFEXITED (status) ) {
            printf("PARENT: child %d ended with status %d \n",
                                 pid_child, WEXITSTATUS (status));
            return EXIT_SUCCESS;
        } else
          return EXIT_FAILURE;
    }
}
```

```
$test-wait
CHILD: pid = 3254
PARENT: child 3254 ended with status 2
```

Basic parent/child synchronization

include files: stdio.h, sys/types.h, sys/wait.h, unistd.h, stdlib.h

test-wait2.c

```
int main(int argc, char **argv) {
    pid_t pid_child; int status;
    if (fork () == 0) {
       printf ("CHILD:  pid = %d \n", getpid());
       pause ();
       exit (2);
    } else {
       pid_child = wait(&status);
       if (WIFEXITED (status) ) {
           printf ( "PARENT: child %d ends with status %d \n",
                           pid_child, WEXITSTATUS (status));
           return EXIT_SUCCESS;
       } else
           if (WIFSIGNALED (status) ) {
               printf ( "PARENT: child %d ended by signal %d \n",
                                 pid_child, WTERMSIG (status));
               return EXIT_SUCCESS;
           }
       return EXIT_FAILURE;
    }
}
```

```
$test-wait2 &
CHILD: pid= 4897
$kill –KILL 4987
PARENT: child 4987 ended by signal 9
```

Basic parent/child synchronization

test-wait3.c

```
#define N 3
int cont = 0;
int main (int argc, char* argv []) {
   int i=0;  pid_t pid;
    while (i <N) {
       if ((pid=fork ( ) )== 0) {
         cont++;
         break;
      }
       i++;
     }

    if (pid != 0) {
      /* parent */
       for (i=0; i<N; i++)
        wait (NULL);
    printf ("cont:%d \n", cont);
   }
      return EXIT_SUCCESS;
}
```

What is the value displayed for *cont* ?

Basic parent/child synchronization

- Primitive *pid_t waitpid (pid_t pid, int* status, int opt)*

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid (pid_t pid, int* status, int opt )
```

Option `opt` of function `waitpid` allows a parent process to check the termination of a child

- with a specific pid value
- or that belongs to a group |pid|

Basic parent/child synchronization

- Value of parameter *pid*

> 0	PID of child process
0	any process that belongs to the same group as the caller
-1	any child process
< -1	any child process in group |pid|

- Value of parameter *opt*

WNOHANG	non blocking call
WUNTRACED	check for stopped processes

- Return code

-1	error
0	(non blocking mode) process is still running
pid	of the zombie process

Code replacement

Code replacement

Primitive `exec` replaces the code that is executed by a new program

Arguments: name of the new program + parameters

The new program will be executed in the namespace of the calling process

If the call *succeeds*, the new program neverhands control back to the calling process

Examples of error (`errno`)

EACCES	*no permission to access the file*
ENOENT	*file not found…*

Code replacement

- □ argv in list format

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

- □ argv in array format

```
int execv  (const char *path, char * const argv[]);
```

Last argument value *must always* be `NULL`

Code replacement

Example - execl

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

int main(int argc, char **argv) {
  execl ("/usr/bin/wc","wc", "-w", "/tmp/fichier1", NULL);
  perror ("execl");
  return EXIT_SUCCESS;
}
```

Code replacement

Example - execlp

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

int main(int argc, char **argv) {
  execlp ("wc","wc", "-w", "/tmp/fichier1", NULL);
  perror ("execlp");
  return EXIT_SUCCESS;
}
```

Code replacement

Example - execv

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

int main(int argc, char **argv) {
  char *arg_vect [4];
  arg_vect[0] ="wc";
  arg_vect[1] ="-w";
  arg_vect[2] ="/tmp/fichier1";
  arg_vect[3] = NULL;

  execv ("/usr/bin/wc",arg_vect);
  perror ("execv");
  return EXIT_SUCCESS;
}
```

Code replacement

include files: stdio.h, sys/types.h, stdio.h, unistd.h, stdlib.h

sleep_exec.c

```
int main(int argc, char **argv) {
    char* argv_sleep[]= {"sleep_prog", "2", (char *) NULL };
    printf ("Begin - pid:%d \n", getpid ());
    execv ("./sleep_prog", argv_sleep);
    printf("End – pid:%d \n", getpid ());
    return EXIT_SUCCESS;
}
```

sleep_prog.c

```
int main(int argc, char **argv) {
  printf("sleep_prog> begin - pid:%d\n ", getpid ());
  sleep (atoi(argv[1]));
  printf("sleep_prog> end - pid:%d\n ", getpid ());
  return EXIT_SUCCESS;
}
```

```
$ sleep_exec
Begin – pid: 356
sleep_prog> begin – pid: 356
sleep_prog> end - pid:356
```