Real-Time Programming TI00AA55

Lecture 5 - 15.02.2017

Jarkko.Vuori@metropolia.fi

Process identifiers

- Running process has the following identifiers (IDs):
  - Process id
  - The process id of the parent process
  - Real user id
  - Effective user id
  - Real group id
  - Effective group id
- IDs are unique way to identify an object (e.g. process or user)

- The process can ask these ids with the following functions:

```
<sys/types.h> <unistd.h>
pid_t getpid(void); //process id of calling process
pid_t getppid(void);//process id of parent process
uid_t getuid(void); //real user id of calling process
uid_t geteuid(void);//effective user id of calling
process
gid_t getgid(void); //real group id of calling process
gid_t getegid(void);//effective group id of calling
process
```

- Notice: No error returns
- `pid_t` is defined to be ”signed integer type” which means in most cases, but not guaranteed to be, an int (4 bytes)

Process control primitives

- Several processes are running simultaneously in the computer
- Two or more processes can run the same program code in memory (code is shared)
- Operating system provides the basic process management services:
  - process creation,
  - process termination,
  - waiting for a child process termination and
  - changing the program in the process

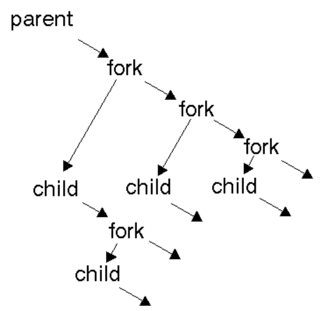
- These services can be used with the following system calls (basic functions for process control):

fork	to create a new process
_exit	to terminate the process (without flushing buffers), or exit to terminate the process (and flushing buffers)
wait	to wait for termination of a child process
waitpid	to wait for termination of specific child process
exec	to initiate a new program in a process

Creation of a new process

- Only a running process can create a new process
- The new process is called a child process of the process that created it
- The process that created a new (child) process is called the parent process of the child process
- The process can create a new process with a system call fork
  `<unistd.h>`
  `pid_t fork(void);`
  This function returns
  - 0 for a child process
  - the process id of the child for the parent (>0)
  - -1 if error

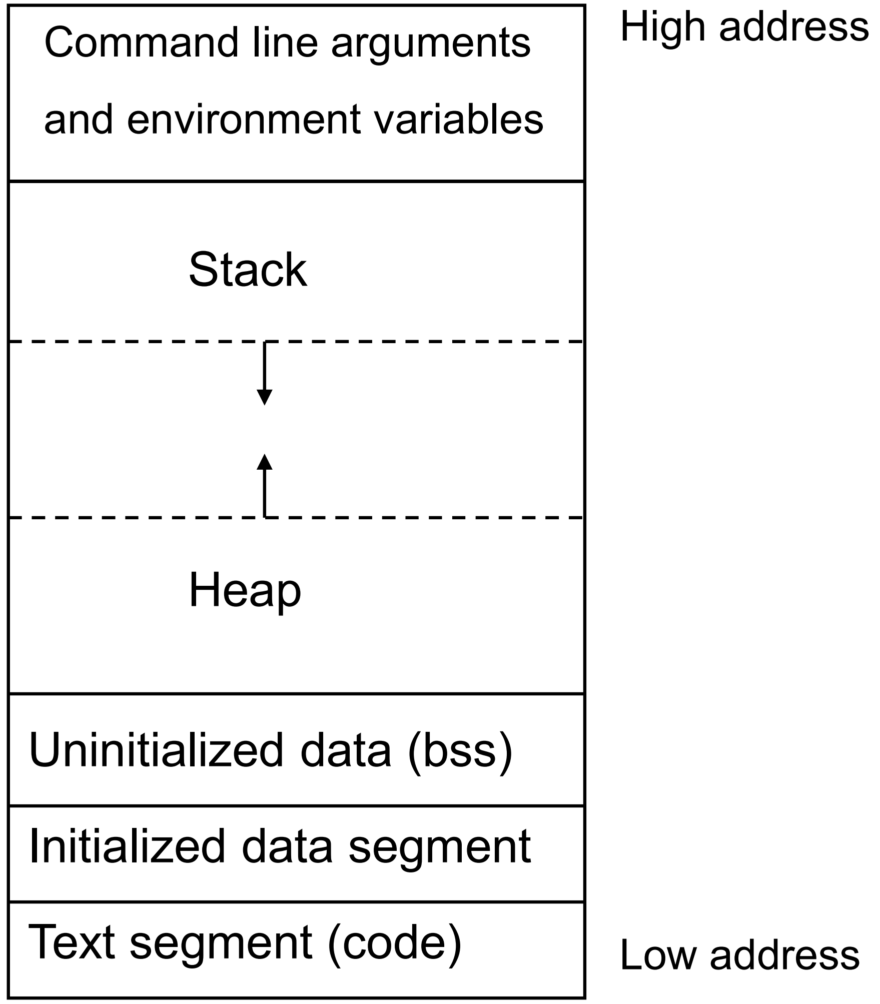
```
if ((pid = fork()) > 0) {
  // parent here
  ...
} else {
  // child here
  ...
}
```



- Function is called once, but there are two returns from it (one in the parent and another in the child)
- Both processes continue from the statement after the fork-call
- The parent process gets the process id of the child from the return value of fork
- The child process can ask the process id of the parent with the function `getppid`
- Nobody knows which process continues execution first
- Parent process and child process shares the program code, but they have their own data areas (data segment, heap and stack segment)

Memory area of the new process

- The memory area of the new process (child process) is a copy of the memory area of the process that created it (the parent process)
- This means that initialized data segment, uninitialized data segment, heap, stack, command line parameters and environment variables have the same contents immediately after the fork
- Code segment can be common for both processes
- Remark. Copy On Write –feature
- Only text segment and initialized data segment are stored on the disk (in the executable file)
- Function exec constructs other segments when the program is started



Parent process creates one child

```
int main(void) {
   pid_t pid_of_child;
   int a = 5;
   pid_of_child = fork();
   if (pid _of_child > 0) {
         // this is parent code for parent process
        // tasks of parent process are performed

        // parent could wait for the child here
        exit(0); // parent terminates
   }
   if (pid_of_child == 0) {
         // this is child code for child process
        // tasks of child process are performed

        exit(5); // Child terminates
   }
   // This part of code is never executed
}
```

Waiting a process

- Often parent process and child process co-operate to do some task
  - i.e. parent process can create a child for doing some subtask
- Therefore we need some kind of synchronization so that parent process can know when this subtask is ready
- The simplest possible synchronization method is waiting
  - The parent process can wait for the termination of the child
- The simplest way to wait a child process is to call the function wait that is described below:

  ```
  <sys/wait.h>
  pid_t wait(int *status);
  ```

  Return value
  - Process id of the child that is terminated, if ok
  - -1, if error (if no children for example)
- Function wait blocks, if no children of the process have terminated
- If process has many child processes, wait function returns, when one (whatever) child process terminates (child calls exit or _exit)
  - Function wait returns the process id of the terminated child
- Function wait returns immediately and returns the termination status of the child if child has terminated already before than wait was called
- Function wait returns immediately and returns –1, if the process has no children

Passing the termination status

- The child process passes the exit status to the kernel using the function exit (or _exit)
- The parent process receives the so called termination status of the child from the kernel using function wait (or waitpid) and passing the address of an int parameter
- This termination status contains the reason of termination that can be normal (the child has called exit or _exit) or abnormal (termination signal is sent to the child)
  - In addition to the termination reason, the status contains in the normal case the exit-status, that child has put as a parameter to the exit or in the case of abnormal termination the number of the signal that caused the termination of the child
- When child process terminates, the kernel also sends a signal SIGCHLD to the parent
  - The default handling of this signal is to ignore it
- Notice the difference between termination status and exit status
  - When wait function is returned we can find out the reason of termination and depending on the reason either exit-status or signal number
    - This can be done with macros on the next page
  - Exit status is used to indicate the success (0) or failure (all other values) of the child doing it’s task
  - See the picture on the page 12

Process termination functions exit and _exit

- exit <stdlib.h> (Ansi)
  - Prototype: `void exit(int status);`
  - Makes clean up procedures in standard i/o-library:
    - Clears the buffers (fflush)
    - Releases the space allocated for buffers
    - Closes the files (close)
  - Calls the function _exit (and passes the exit-status as a parameter)
- _exit <unistd.h> (Posix)
  - Prototype: `void _exit(int status);`
  - Passes the termination status to the kernel
  - Terminates the process (where it is called)
  - Closes file descriptors
  - Returns to kernel
- Exit status
  - Both functions take one parameter that is exit status
  - Parent process can get the exit status of it's child
  - If the program has been started from the (command) shell, shell receives the exit status, because shell process is the parent

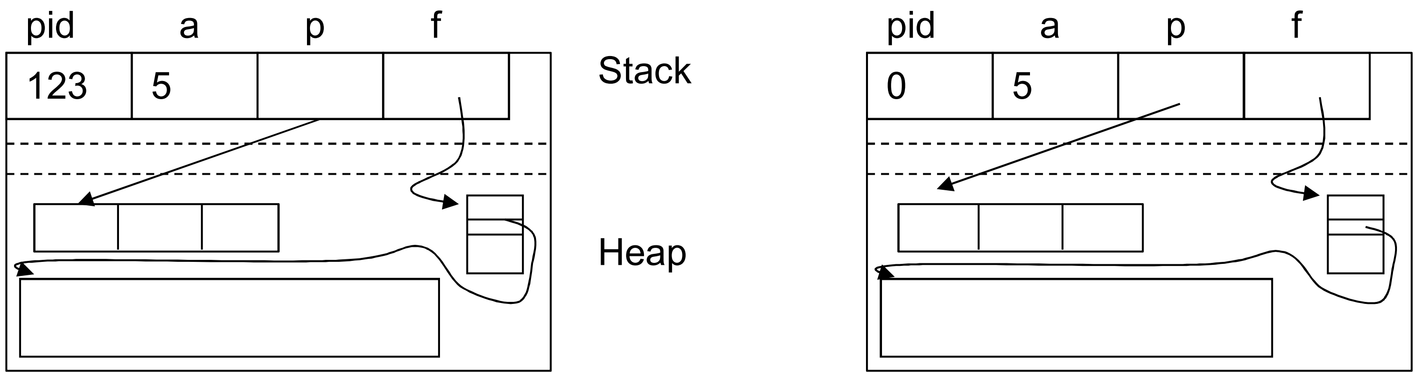
Termination status / exit status

- Posix defines macros we can use to extract termination reason and exit status from the status got from the wait function
  `<sys/wait.h>`
  `WIFEXITED(status)` True if exited normally
  `WIFSIGNALED(status)` True if terminated by signal
- If child process is terminated normally so that it has called exit or _exit (macro WIFEXITED returned true), it's exit status can be extracted with the macro WEXITSTATUS(status)
- If child process is terminated abnormally with a signal that had no handler (WIFSIGNALED returned true), the signal number of that signal can be extracted with the macro WTERMSIG(status)
- We have seen that parent can wait for a child using a function `wait` (or `waitpid`)
- A child can wait the termination of the parent like this:

```
while (getppid() != 1)
     sleep(1);
```

- This solution is based on the fact that init process (id 1) becomes a parent for a child when it's own parent terminates
  – This way of waiting uses polling and wastes resources
  – Later we use signals to inform the child

Parent and child process after fork



Executed once. Results usable in both.

Executed only in parent.

Executed only in child.

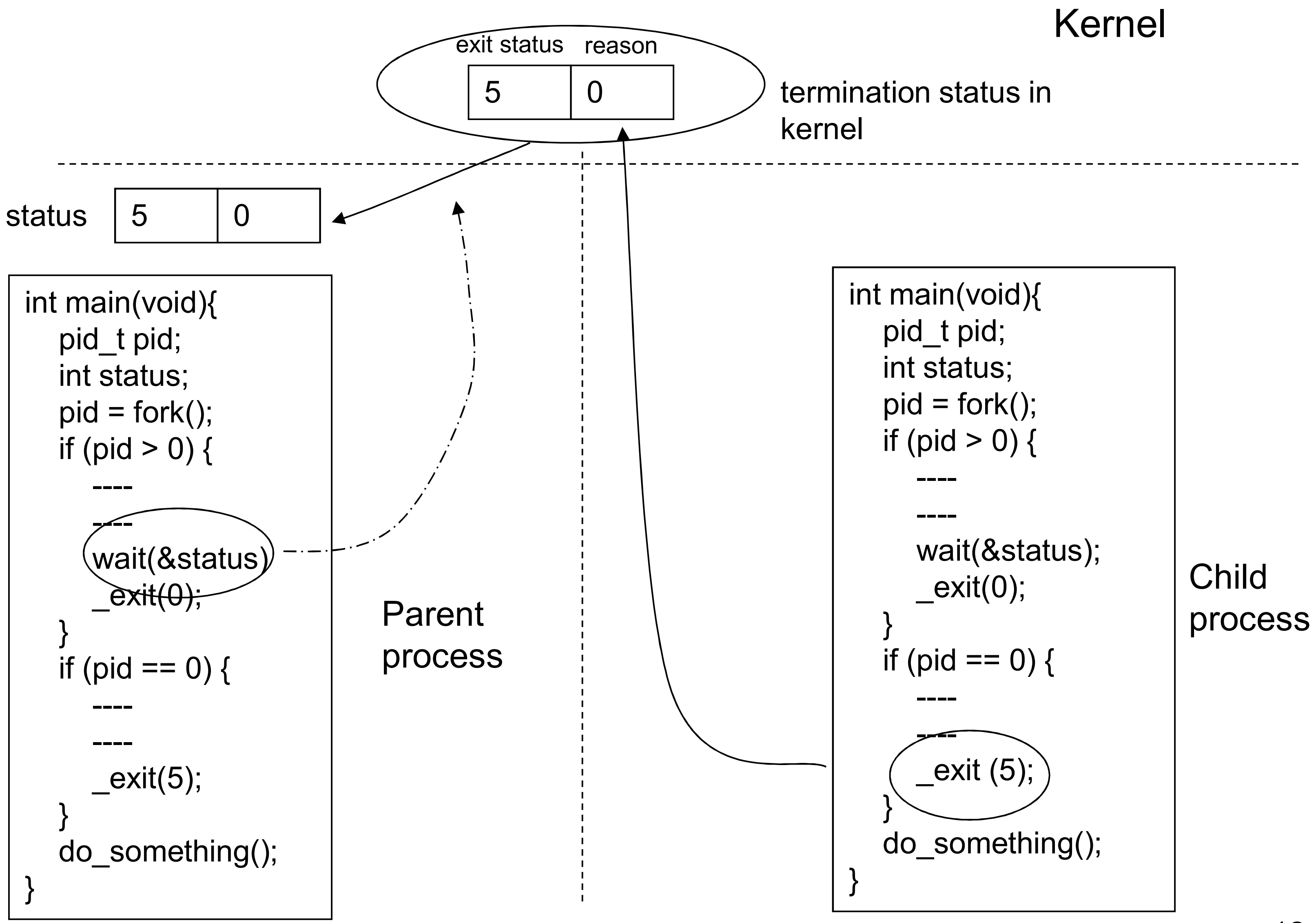
Executed in both

```
int main(void){
    pid_t pid;
    int a;
    double *p =
    malloc(3*sizeof(double);
    a = 5;
    pid = fork();
    if (pid > 0) {
        ----
        ----
    }
    if (pid == 0) {
        ----
        ----
    }
    do_something();
    return 0;
}
```

Text segment is actually shared but thinking it as a copy helps to understand things

```
int main(void){
    pid_t pid;
    int a;
    double *p =
    malloc(3*sizeof(double);
    a = 5;
    pid = fork();
    if (pid > 0) {
        ----
        ----
    }
    if (pid == 0) {
        ----
        ----
    }
    do_something();
}
```

Passing termination status from the child to the parent



```
int main(void){
    pid_t pid;
    int status;
    pid = fork();
    if (pid > 0) {
        ----
        ----
        wait(&status)
        _exit(0);
    }
    if (pid == 0) {
        ----
        ----
        _exit(5);
    }
    do_something();
}
```

```
int main(void){
    pid_t pid;
    int status;
    pid = fork();
    if (pid > 0) {
        ----
        ----
        wait(&status);
        _exit(0);
    }
    if (pid == 0) {
        ----
        ----
        _exit (5);
    }
    do_something();
}
```

Parent creates two children (Solution 1)

```
int main(void) {
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 > 0)  {   // this is parent
      pid2 = fork();
      if (pid2 > 0) {  // this is parent
         // tasks of parent process  are performed
         …
         wait(NULL); // waiting for the child to terminate
         wait(NULL); // waiting for the second child to terminate
         exit(0);
      }
      if (pid2 == 0) {   // this is child 2
         // tasks of child process are performed
         …
         exit(0);
      }
    }
    else {  // this is child 1
         // tasks of child process are performed
         …
         exit(0);
    }
}
```

Parent creates two children (Solution 2)

```
int main(void) {
    pid_t pid1, pid2;
    // This part is done only by parent
    pid1 = fork();
    if (pid1 == 0)  {
         // this is child 1
         // tasks of child process are performed
         …
         exit(0);
    }
    pid2 = fork();
    if (pid2 == 0) {
         // this is child 2
         // tasks of child process are performed
         …
         exit(0);
    }
    // parent continues from here
    // tasks of parent process are performed

   wait(NULL);
   wait(NULL);
   exit(0);
}
```

Parent creates 5 (n) children

```
int main(void) {
   pid_t pid;
   int i = 0;
   // This part is done only by parent but "inherited" by children
   while (i < 5) {
      pid = fork();
      i++;
      if (pid == 0) {
         // this is child
         // tasks of child process are performed

         exit(0); // child terminates
      }
      // Parent continues from here after creating a child
   }
   // Parent continues from here after creating all children
   // Tasks of parent process are performed
   while(wait(NULL) > 0); // wait for all children to terminate
   exit(0);
}
```

Parent/child process features

- Remember that the execution of code in the child continues from the statement immediately after the fork:

```
int main(void) {
    int i;
    for (i = 0 ; i < 3 ; i++) {
        fork();
        do_something();
    }
}
```

  a) How many processes do exist?
  b) How many times do_something is called?
  c) If do_something is replaced with printf(“%d”, i), what is the output from the program?

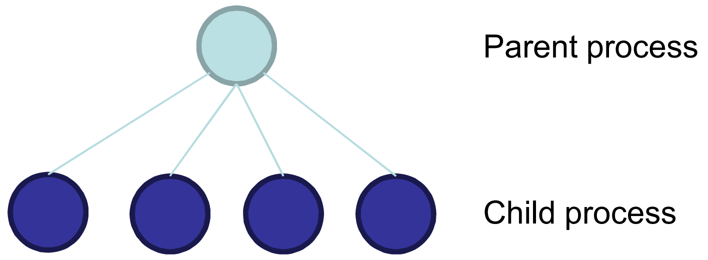
- It is not possible to know in advance which process runs first after the fork
- The open file descriptors of parent are inherited by child (file descriptor is duplicated)
- It is possible to synchronize the interaction between parent and child in different ways (using signals)
- It is possible that fork call can fail, if
  - The are too many processes in the system
  - The resource limit of a running processes for the user is exceeded (resource limit RLIMIT_NPROC)

Common and non-common things of parent and child

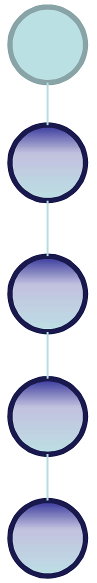
- The child process inherits the following things:
  - Real user ID, real group ID
  - Effective user ID and effective group ID
  - Current working directory
  - File mode creation mask
  - Signal mask and signal handlers
  - Close-on-exec flag for open file descriptors
  - Environment
  - Resource limits
- The differences are:
  - The return value of fork function
  - Process id
  - Record locks of parent are not inherited
  - Pending signals are cleared in child process

Different process relation patterns

- In our earlier system the parent process created n children. The relations between processes can be described with the figure below:



- There are other options to create a multi-process system. One of them is such that parent process creates a child (child1). This child creates a child (child2) and again this child creates the next child (child3) and so on. In that case the relations between process are like below:



On the following page you can find an example of this kind of chain of three processes

Parent process creates a child that creates a child (Solution 1)

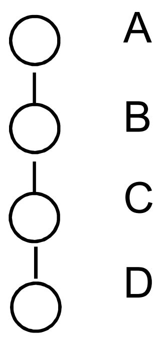
```
int main(void) {
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 == 0) { // This is child 1
          // Code for child process 1
          pid2 = fork();
          if (pid2 == 0) { // This is child 2
                    // Parent of this child is child 1
                    // Tasks of child 2 are performed
                    exit(0); // Child 2 terminates
          } // Child 1 continues from here
          // Tasks of child 1 are performed
          wait(NULL); // Wait for child 2
          exit(0);  // Child 1 terminates
    }
    // Parent (grand parent) continues from here
    // Code for parent process
    // Tasks of parent are performed
    wait(NULL); // Wait for the child 1
    exit(0);
}
```

Parent process creates a child that creates a child (Solution 2)

```
int main(void) {
    pid_t pid1, pid2;
    pid1 = fork();
    if (pid1 > 0) { // This is the "original parent"
        // Parent (grand parent) continues from here
        // Code for parent process
        wait(NULL); // Wait for child 1
        exit(0);
    }
    // Code for child process 1
    pid2 = fork();
    if (pid2 > 0) { // This is child 1 that is a parent of child 2
        // Child 1 continues from here
        // Code for parent process
        wait(NULL); // Wait for child 2
        exit(0);
    }
    // Code for child process 2
    // Only the last child continues from here
    // Code for child process
    exit(0);
}
```

Same thing with a loop

```
int main(void) {
    pid_t pid;
    int i;
    for (i = 0; i < 3; i++) {
         pid = fork();
         if (pid > 0) { // This is a parent
              // task of parent process
              do1(); // A, B and C
              wait(NULL);  // wait for your child
              exit(0);
         }
         // else
              do2(); // B, C and D
    }
    // Only the last child reaches this
    exit(0);
}
```

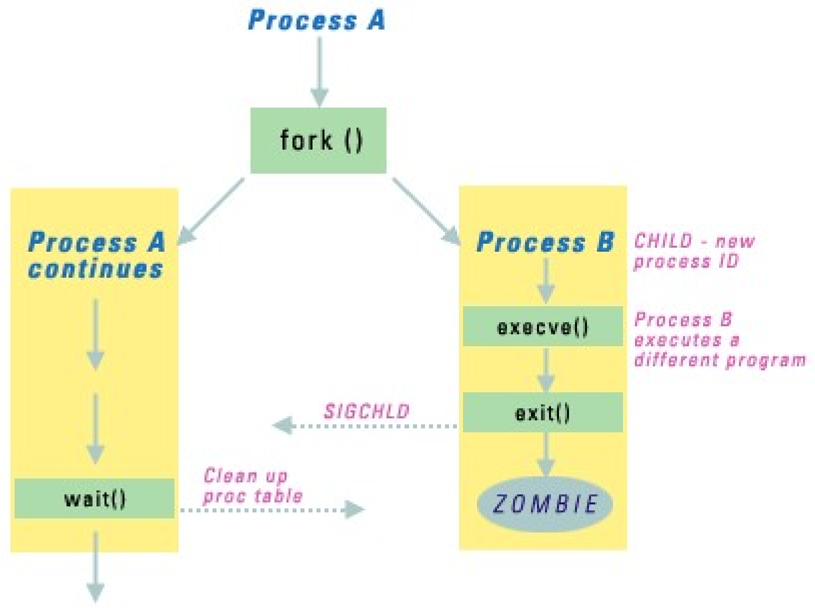


Remark. The process A does the task do1. The processes B and C do the tasks do2 and do1. The process D does the task do2.

Synchronizing terminations

- Case 1a. Parent process is waiting and child still continues. Wait function blocks in this case. Parent continues when child terminates
- Case 1b. Parent has something to do but child terminates. The parent process starts to wait for the child later. In this case kernel stores the exit status of the child so that parent can "ask" it later. When parent calls function wait, it returns immediately and gives the termination status of the child. Then it is not anymore stored by the kernel. The kernel must save some information of terminated child until parent ask it's termination status with function wait (or waitpid). This kind of process is called "zombie". Zombies waste resources of the computer (they are still processes in the system)
- Case 2a. The parent does not call wait function and terminates before the child. In this case the init-process (ID 1) becomes the parent of the child
  - init process fetches the exit status of the child when it terminates, so that it does not become a zombie
- Case 2b. The parent does not call wait function and terminates later than the child. Kernel stores the termination status and discards it when the parent terminates

- Often (for example in servers that run infinitely) we don't want to wait for children or fetch their termination status (case 1a). Then we need to prevent zombies from being born. The solution for this is **forking twice** or **double forking**



Zombie problem in servers

This example is used to demonstrate the zombie problem in server

```
int main(void) {
    pid_t pid;
    int running = 1;
    while (running != 0) {
         wait_client();
         pid = fork();
         if (pid == 0)  { // child
             serve_client(); // this can take more or less time
             exit(0);
         }
         // wait(NULL); // server process cannot block, because
                       // it must be ready to accept new clients

    }  // end of while
}  // end of main
```

Remark. All children are left in the system as zombies. Finally server cannot create a new children (and serve new clients) any more

Forking twice (to prevent zombie)

```
int main(void) {
    pid_t c1, c2;
    int running = 1;
    while (running != 0) {
         wait_client();
         c1 = fork();
         if (c1 == 0) { // child 1
             c2 = fork();
             if (c2 > 0)
                 exit(0); // child 1 (parent of child 2) terminates
             // this is child 2 that serves the client
             serve_client();
             exit(0); // Parent of this child is init
                      // because the original parent has been terminated
         }
         wait(NULL); // fetch termination status of c1
                     // this does not block, because c1
                     // has been terminated
    } // end of while
} // end of  main
```

Remark. Double forking is also used in daemon processes

Function waitpid

- The easiest way to wait for a child is function `wait`. Function `waitpid` provides more options like:
  1. You can specify the child or children you want to wait and
  2. You don't need necessarily to block if the child is not ready
- The prototype of the function `waitpid`

```
<sys/types.h>  <sys/wait.h>
pid_t waitpid(pid_t pid, int
*status, int options);
```

- This is the way you can determine the child (children) to wait:
  – pid > 0, wait a child whose process id is pid
  – pid = 0, wait a child whose process group id is same than that of caller
  – pid < 0, wait a child whose process group id is |pid|
  – pid = -1, wait whatever child first terminates (like wait)
- You can choose non-blocking wait by passing WNOHANG as a third parameter
  – Then `waitpid` does not block, if child is not ready (the return value in that case is 0)
- Return values of function `waitpid`:
  – The process id in "normal case"
  – Value –1 in error situations. These are:
    1) There is no child of specified type
    2) System call wait is interrupted with a signal
  – 0, when child is not terminated and we have used WNOHANG mode

Waiting for several children

```
// Example 1. (Waiting for children in a specific order)

int main(void) {
    pid_t childs[3];
    int i;
    for (i = 0; i < 3 ; i++) {
        childs[i] = fork();
        if (childs[i] == 0) { //this is child
                do_something();
                sleep(i+1);
                exit(0);
        }
    }
    i = 0;
    while ((pid = waitpid(childs[i], NULL, 0)) > 0) {
        i++;
        printf("Child %d has terminated", pid);
    }
}
```

Waiting for several children

```
// Example 2. (Waiting for each children in the order they terminate)

int main(void) {
    pid_t pid;
    int i;
    for (i = 0; i < 3 ;i++) {
        pid = fork();
        if (pid == 0) { // this is child
                do_something();
                sleep(i+1);
                exit(0);
        }
    }
    while ((pid = waitpid(-1, NULL, 0)) > 0) {
         printf("Child %d has terminated", pid);
    }
}
```

or `wait(NULL)`

Non blocking waiting

```
// Example 3. (In this example we prevent zombies by "polling"
// the termination of children.)

int main(void) {
    pid_t pid;
    int server_is_running = 1;
    while (server_is_running != 0) {
        wait_client();
        pid = fork();
        if (pid == 0) { // this is child
               serve_client();
               exit(0);
        }
        while (waitpid(-1, NULL, WNOHANG) > 0);
        // the server fetches the status of every
        // child that is ready to avoid zombies
    }
}
```

Remark. This is not actually real polling. The process is looping in the inner loop only as long as there are terminated childrens. After that it returns to wait for new clients. When next client arrives, the server process again retrieves the returns status of all children that have possibly terminated during the waiting.

Last examples of waitpid

- Let’s assume we have several (n) childrens
  - Compare the two different ways of waiting
- Waiting in way 1

```
…
while((pid=waitpid(-1, NULL, 0)) >= 0) {
    printf("Do it"); // How many times?
                     // How long?
}
```

- Waiting in way 2

```
…
while((pid=waitpid(-1, NULL, WNOHANG)) >=0)
{
    printf("Do it");  // How many times?
                      // How long?
}
```

- We could write the second way in different way

```
…
while((pid=waitpid(-1, NULL, WNOHANG)) >= 0) {
    if (pid == 0)
         printf("Do it");
    if (pid > 0)
         printf("Child terminated\n");
}
```

Resource limits

- A process can allocate only a limited amount of resources when it is running
  - Each process has it’s own resource limits for each resource
- There are two sorts of limits, so called soft limit and hard limit
- On the right you can see some examples of resource limits
  - The identifier that is used to find out (or set) the limit during runtime is also given

- Examples of resource limits:
  - Amount of CPU time in seconds (RLIMIT_CPU)
  - Size in bytes of Data segment (Initialized data, Uninitialized data and Heap together) (RLIMIT_DATA)
  - Maximum size of file in bytes that can be created (RLIMIT_FSIZE)
  - Maximum number of open files per process (RLIMIT_NOFILE)
  - **Maximum number of child processes per user ID (RLIMIT_NPROC)**
  - Maximum size of stack in bytes (RLIMIT_STACK)
  - Maximum size of mapped address space in bytes (RLIMIT_VMEM)

Getting and setting resource limits

- Functions `getlimit` and `setlimit` can be used to ask and set the resource limits
- Both of these functions have a parameter of type `struct rlimit` (pointer to this structure)

```
struct rlimit {
  rlimit_t rlim_cur; //soft limit = current limit
  rlimit_t rlim_max; //hard limit = max value for 
current limit
}
```

```
<sys/resource.h>
int getrlimit(int resource, struct 
rlimit *reslimp);
```

  – returns 0, if OK, non zero if error

- `int setrlimit(int resource, const struct rlimit *reslimp);`
  – returns 0, if OK, non zero if error

- The resource identifier (the first parameter) is a constant defined in <sys/resource.h> (see the previous page)
- The rules of modifying resource limits:
  1. You can change the soft limit as long as it stays between hard limits
  2. Process can make it's hard limits smaller (but it cannot anymore restore them)
  3. Only the superuser can increase hard limit
- Constant RLIM_INFINITY means unlimited

Resource limits (Example)

- In the example below we change the maximum number of open files to maximum possible value (we set the soft limit to the value of hard limit):

```
struct rlimit resource_limit;
getrlimit(RLIMIT_NOFILE, &resource_limit);
resource_limit.rlim_cur = resource_limit.rlim_max;
setrlimit(RLIMIT_NOFILE, &resource_limit);
```

- Remark. Remember what you have learned earlier about different limits
  - For example, the function sysconf you have learned earlier gives the same result than the function getrlimit for the maximum number of processes per user
- Find the values received in different ways in the Edunix computer below:

_POSIX_CHILD_MAX	25
CHILD_MAX	not defined
Sysconf with const _SC_CHILD_MAX	1024
getrlimit with const RLIMIT_NPROC (soft limit)	1024
getrlimit with const RLIMIT_NPROC (hard limit)	1024

setjmp and longjmp functions

- This is another example of functions, that is called once and that returns twice
- Can be used to return from deeply nested function calls directly to higher level
- Most often used in error situations
- Prototypes of functions:

```
<setjmp.h>
int setjmp(jmp_buf return_info);
void longjmp(jmp_buf return_info, int
return_value)
```

- How function setjmp works
  - Function setjmp stores all information needed to return to this location in the buffer return_info
  - Function returns 0, when it returns from this call, i.e. from the call of function setjmp
  - It looks like we had returned from this function, when function longjmp is called. In that case the return value of setjmp is what is passed to the function longjmp as a second parameter
  - The buffer (type jmp_buf) is a kind of array that contains information needed to restore the stack in the state where it was after the call to setjmp. The programmer is responsible for allocating that buffer as a global variable
- How function longjmp works
  - When function longjmp is called the function returns to the location following the function call setjmp. The information where to return and how to clean up stack is given in the parameter return_info. The second parameter is a return value for a function setjmp

Try, throw, catch using setjmp and longjmp

```
jmp_buf buf;
int main(void) {
    int r;
    // try
    r = setjmp(buf);
    if (r == 0)
        do_it();
    else {
        // catch an error
        printf("Error no %d\n", r);
        // everything is as before
        // calling do_it
    }
        ...
}

void do_it() {
    ...
    f1();
    ...
}
```

```
void f1() {
    ...
    f2();
    ...
}

void f2() {
    ...
    if (some unrecoverable error)
        // throw an exception
        longjmp(buf, 1); // error code is 1
    ...
}
```