Processes

Fall 2016

What a process is

- A process is an instance of a program in execution. If 10 users are running vi, then we have 10 processes of vi.
- Obviously, a process is an entity that represents a program that a CPU is running.
- What does this entity hold?
- It holds the code of the program being run, the files that were opened and the memory that was allocated during the execution.
- Of course it should keep in some way where the CPU has reached in its execution.
- To understand how we keep the place where the CPU had reached we first have to understand how the CPU operates.
- The CPU keeps its data in registers. A register is an internal variable of the CPU.
- The CPU uses registers as temporal variables for operation it performs, or to keep certain values it needs to keep.
- One of these registers keeps the next operation the CPU needs to perform

How the OS keeps processes

- Following the previous section it is clearer how the OS represents a process. A process is represented by a structure containing:
- **Pid** - process id - a unique number given to each process.
- **Open Files** - a list of opened files.
- **State** - The state of the process. Is it running? Is it waiting for something?
- **Owner** - The user who created the process.
- **The CPU registers**
- **Process memory** - Every process has its own memory space. (We will learn more about the memory in the next few lectures)
- A system that has several processes keeps them in a linked list of structures as explained before, where each process is represented by a structure in the list. The OS chooses a process to run from this list.
- Actually the OS keeps several lists. One list is of processes that are ready to run. Processes might not be ready to run because they are waiting for something (IO, mutex, semaphore, etc.). These processes are kept in different lists. We'll talk about these lists in the next weeks. The list of processes that are read to run is called the **ready queue**.

How parallelism is achieved

- How does a computer run several processes simultaneously?
- First we have to understand that a computer with one CPU runs only one process at a time. The illusion of concurrency is achieved by running a process for a little time and then switching to run another process for a little time and then switching to another process and so on.
- Switching from one process to another is called **context-switching**.

How context-switch is performed

- As you remember the attributes of a process are kept in the process structure. The opened files, the memory and the rest of the attributes are maintained through the process structure. For example, when a process opens a file the file is added to the opened files list. Actually, the file is considered opened only when the appropriate data is added to the opened file list. The only attribute that is not updated through the process structure is the CPU-registers. It is impossible to update the CPU registers in the process structure as they change.
- So how do we perform context switch?
- It follows that if we "read" the registers from the CPU to the process structure we'll have all the information we need to reconstruct a process. To perform a context switch the OS reads the CPU registers and stores them in the process structure. This way we can suspend a process and "store" it. When we want to restore a process, we read the registers from the process structure and load them to the CPU.

How context-switch is performed (con’t)

- To understand exactly how an operating system does that we have to distinguish between two types of OS-s: preemptive systems and non-preemptive systems.
- **Non-preemptive** systems wait for a process to declare that it is willing to give up the CPU. Then the OS stores the CPU registers, thus stop running the process and find another process to schedule.
- **Preemptive** systems initiate a context switch themselves. Such systems do not wait until the process gives up the CPU but decide themselves when to change the running process. These systems use a mechanism called "interrupt" to wake up every some amount of time (few milliseconds) and change the running process if they decide to do so. Changing the running process, is again done by reading the registers from the CPU and storing them in the process structure.

Deciding what process to run

- When a system performs a context switch, it needs to determine which process is going to be scheduled instead the one that is running. There are several policies for this decision. Linux supports three policies that are the three common ones:
- **Round-Robin**. In this policy the OS keeps the ready processes in a queue. The process chosen to be scheduled is the process at the head of the queue. The process we suspend is moved to the tail of the queue.
- **Real-time**. In this policy each process is given a number (priority). The process chosen to be scheduled is the one with the highest priority.
- **Conventional**. In this policy each process is given a number (priority), but we hold another number (dynamic priority) and the process chosen to run is the one with the highest dynamic priority. Initially the dynamic priority is set to the value of the static priority. Every time a scheduling is performed and the process is not chosen to run its dynamic priority increases, when it is chosen to run, its dynamic priority decreases. This is a compromise between the real-time policy and the Round-Robin policy. Processes with higher priorities run more than processes with lower priorities, but process with lower priorities still get to run.

Creating a new process

- A new process is created with the function call "fork". Fork duplicates the process, and creates an exactly the same process. The processes are identical in the sense that they both have the same memory allocation, the same values in every variable, the same opened files. Both of the processes run the same code and both of the processes will continue running from the same place, i.e. they will both continue running by returning from the fork system call.
- The only two difference between these two process is, their PIds and the value returned by fork. One process (the new one) gets 0. The other (the older one) get a positive number which is the pid of the new process.
- We usually call the new process the child and the older process the parent.

Code Example

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv) {
        int res;
                /* duplicate the current process */
        res = fork();
                /* *******
                 * If fork was successful there are now two processes at this point
                 **** */
                /* check whether fork is successful */
        if (res < 0) {
                perror("fork");
                exit();
        }
```

```
/* Check whether this is the father or the child */
                    /* the child got 0 from fork. */
                    /* the parent got a positive number from fork */
        if (res == 0) {
                            /* The child enters this block */
                    printf("I am the child. My pid is %d\n", getpid());
        } else {
                    int child_pid = res;
                            /* The parent enters this block */
                            /* print the parent pid */
                    printf("I am the father my pid is %d\n", getpid());
                    printf("Father is waiting for child to terminate %d\n",
child_pid);
```

```
/* Wait for the child to terminate */
                waitpid(child_pid, NULL, 0);
                        /* If the parent exits before the child, the child will become
                         * a child of process 0 */
                        /* A child that finishes its execution is in zombie mode,
                          * until its parent performs the wait function call */
                        /* Instead of wait we could set a signal handler for SIGCHLD
                         * to be notified of a child termination */
                printf("Father has seen that the child (%d) exited\n", child_pid);
        }
}
```

How fork creates a new process.

- To create a new process, fork duplicates the process structure. Duplicates the memory, duplicates the open-file list etc.
- Fork also duplicates the CPU registers, which means that the new process will continue its run from the same point as the old process (after the fork sys. call).
- There are two differences between the two processes.
- The First difference is that their Pids are different.
- The second difference is the return value of fork.
- After the process structure was created it is added to the process list.

How to make a process run a different code

- When we create a new process, we don't necessarily want it to run the same code as the old process.
- Sometimes we want to create a process that will run a different code.
- For example we may want to create a new process that will run "ls".
- Unfortunately the only way to create a new process is by duplicating the current one.
- However we can make our process run a different code.
- Function calls like execv, execl, execlp, etc. replaces the process structure with a new structure.
- These functions destroy the memory of the process, close the opened files etc. Then they create a new memory space and load a new code to run.

Code Example

```
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdio.h>
int main(int argc, char **argv) {
        int res;
                /* duplicate the current process */
        res = fork();
                /* *******
                 * If fork was successful there are now two processes at this point
                 **** */
                /* check whether fork is successful */
        if (res < 0) {
                perror("fork");
                exit();
        }
```

```
/* Check whether this is the father or the child */
                    /* the child got 0 from fork. */
                    /* the parent got a positive number from fork */
        if (res == 0) {
                    char *args[] = {"ls", "-l", NULL};
                            /* The child enters this block */
                    printf("I am the child. My pid is %d\n", getpid());
                            /* Make this process run the code of ls. */
                            /* this function will destroy the current process
and
                             * it will be no more! */
                    res = execv("/bin/ls", args);
```

```
/* the only way we can return from execv is if there was an error */
                if (res == -1) {
                        perror("execv");
                        exit(2);
                }
                        /* This code is never reached */
                printf("This will never be printed\n");

                  }
```

```
    else {
                int child_pid = res;
                        /* The parent enters this block */
                        /* print the parent pid */
                printf("I am the father my pid is %d\n", getpid());
                printf("Father is waiting for child to terminate %d\n", child_pid);
                        /* Wait for the child to terminate */
                waitpid(child_pid, NULL, 0);
                        /* If the parent exits before the child, the child will become
                          a child of process 0 */
                        /* The child is in zombie mode, until its parent performs
                          the wait function call */
                        /* Instead of wait we could set a signal handler for SIGCHLD
                          to be notified of a child termination */
                printf("Father has seen that the child (%d) exited\n", child_pid);
        }
}
```