2 MULTITASKING

Multitasking can be achieved by multithreading – multiple threads within a single process – or by a multi process application, where the first process creates additional ones which can then run in parallel. Each solution has its merits. With multithreading, threads can easily communicate (for example, with global variables) and system overhead is significantly lower. On the other hand, with multiple processes you can isolate processes so they cannot use each other's (private) data.

When a program is to be started, the operating system must first prepare memory for it, creating a process. Then instructions and data from a program file are loaded into memory, and the process can be started (with a single thread in it). This thread can then request the creation of new threads or processes, as programmed in its instructions. These requests use the operating system API, e.g. `pthread_create` or `fork`, as described in the following text.

2.1 Multitasking with multiple processes

2.1.1 Creating a new process

A new process can be created from an existing process with the `fork()` system call (in UNIX).

```
int fork();
```

The `fork` function duplicates an existing process, called a parent process in this context, into a new process, called a child. Everything from the instructions to the data will be copied – the child will be exactly the same as parent. However, any change that the parent or child process makes from that point on will only be visible in that process; other processes will not see that change.

In the implementation, copying is delayed, since paging is used, which means that only pages that are changed are copied when the change occurs. Since most pages are not changed (instructions, constants), this represents a significant performance gain.

Since fork is not passed an argument, it always creates a duplicate process unless there are no resources available for it (or some other limit has been reached). In case of an error, if no process is created, the fork function returns -1.

If a child process is created, fork returns two values:

- to the child process: 0 (zero)
- to the parent process: child process identification number (PID).

In this way, the program can branch with a return value and execute either the work of the child process or the work of the parent process.

Parent and child are two different processes, they are identical at the time of child creation, but they do not (logically) share memory. However, system resources created by the parent process before the creation of the child process are shared with the child process (e.g. files, shared memory, ...). For example, if a parent process opened a file and then created a child process, that file is shared: Each read/write/* that moves the file pointer is visible to both. If parent and child process want to use "shared memory" feature, as in case of threads within the same process, OS must be asked to create such memory segment which is visible to both processes. More on this later.

2.1.1.1 Functions `wait`, `exit`, `getpid`

The `exit` function is used to terminate the current process.

```
void exit(int status);
```

The status of the terminated process is returned with the `status` argument. Normally, a value of zero is used to indicate that the process has terminated after successful execution. Any other value indicates an error.

Alternatively, a process can be terminated by exiting its start function - `main` - with return status.

The parent process can suspend its execution with the `wait` function until its child terminate.

```
int wait(int *status);
```

With a pointer to `status`, the parent process can get the exit status of the terminated child process. The return value of `wait` is the PID of the child process or -1 if an error occurred (e.g. because there are no children).

Each process is identified by its PID, but its parent PID is also stored. However, if a parent process terminates (dies) before its child, a new parent process is assigned to the child, usually the *init* process (usually with PID 1). Such children are called orphans.

If a child process terminates but the parent process does not wait for it with a `wait` or similar call, the child process becomes a *zombie*. For such a process, not all resources are released until a `wait` call is made and its status collected.

A process can obtain its own PID with a call to function `getpid()`.

```
pid_t getpid()
```

The PID of its parent process can be similarly obtained with `getppid()`.

2.1.1.2 Examples

The use of fork is explained with two examples, a simple one where only a single child process is created, and another where multiple child processes are created (in a loop).

```
if (fork() == 0) {
   child process job – usually a call to some function;
   exit(0);
}
parent process job;
wait(NULL);
```

Code in **blue** is what parent executes; **red** is code that both processes do – compare return value of fork with zero; **green** is what a child process do, since for it zero is returned by fork.

To create more processes previous code should be put in a loop. However, in the next code segment, another use of fork is demonstrated, where all return values are tested.

```
for (i = 0; i < N; i++) {
   pid = fork();
   switch (pid) {
   case 0:
      child "i" job
      exit(0);
   case -1:
      error handling;
   dafault:
      parent code specific to this child;
      e.g. parent can save its PID for later use
   }
}
some other job parent performs
while (i--)
   wait (NULL); /*wait for all children to terminate*/
```

2.1.2 Shared memory

Parent and child processes or any other two or more processes are separated from each other. Each process has its own address space, inaccessible to other processes. This is a feature!

However, if two processes want to use "shared memory", OS can be used to help, i.e. a shared memory segment is created which both processes can access directly.

If a parent and his children want to use the shared memory, the parent creates the shared memory first and only then he creates his children. The children inherit this shared memory at the same addresses where it is accessible in the parent process. However, if two unrelated processes need to use shared memory, they must know a key (or name) for the shared memory.

Shared memory can be created using two different interfaces that create different objects but have the same result. The first interface, `shmget` (and its associated functions), is explained here. The second, `shm_open`, can be found in the man pages, or elsewhere.

2.1.2.1 Functions `shmget`, `shmat`, `shmdt`, `shmctl`

To create or connect with existing shared memory function `shmget` is used.

`typedef key_t int;`

`int shmget(key_t key, int size, int flags) ;`

If a new shared memory is to be created, for `key` a constant `IPC_PRIVATE` should be used. Otherwise, a key, must be somehow obtained (e.g. via environment variable, or hardcoded in program).

The argument size defines the size of the shared memory, while the lowest nine bits of the flags, divided into three groups of three bits, define the access rights. This number is usually written in octal form. For example: 0644 is decoded: The leading zero is interpreted as a prefix for an octal number. The next three octal digits represent access for processes of the same user, access for processes of the same group the user is in (corresponding to the group the program is assigned to), and access for all others. So, the first digit 6 is the binary number 110, which corresponds to the access rw-: 1 for read, 1 for write, 0 for execute: A memory can be read and written, but not used to execute code from. Others can only read this shared memory ($4_8 = 100_2$).

The return value of `shmget` is a segment ID, or -1 if an error occurred.

After a segment is created (or connected with) it must be mapped into address space of the process with a call to `shmat`.

`char *shmat(int segid, char *addr, int flags) ;`

The first argument, `segid`, should be the segment ID returned by `shmget`. The second argument, `addr`, can be used by a program to request that shared memory be mapped at that address. If `NULL` is specified instead, OS selects where the shared memory segment should be stored within that process. Flags are usually set to zero.

When the shared memory is no longer needed by the process, a function `shmdt` should be called to remove mapping (detach) of this segment to the address space of calling process.

`int shmdt(char *addr) ;`

The shared memory segment can be deleted (or just mark for deletion) with `shmctl`.

`int shmctl(int segid, int cmd, struct shmid_ds *sbuf) ;`

When `segid` is shared memory segment ID and `cmd` is `IPC_RMID` then the segment is removed immediately, or just marked for deletion if it is currently in use by and process. Only when all processes that use this segment detach from it the segment will be removed. Third argument isn't relevant, NULL can be passed for this command.

If `shmctl` isn't called, shared memory segment will persist even after all processes that were using it terminate. That is obviously not good, since that memory will be reserved but not used.

2.1.2.2 Example program with processes and shared memory

```
#include <stdio.h>
#include <signal.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

int Id; /* shared memory segment ID */
int *SharedVariable;

void writer(int val);
void reader(void);
void remove_shared_memory(int sig);
```

```
int main(void)
{
   /* obtain shared memory segment */
   Id = shmget(IPC_PRIVATE, sizeof(int), 0600);

   if (Id == -1)
      exit(1);  /* error - no shared memory */

   /* map shared memory to the process */
   SharedVariable = (int *) shmat(Id, NULL, 0);

   /* initialize shared memory with a value */
   *SharedVariable = 0;

   /* if a SIGINT (Ctrl+C) arrives, remove shared memory before
      terminating the process, i.e. call a function for it */
   sigset(SIGINT, remove_shared_memory);

   /* create a reader and writer processes */
   if (fork() == 0) {
      reader();
      exit(0);
   }
   if (fork() == 0) {
      sleep(5);
      writer(123);
      exit(0);
   }
   /* wait for those processes to finish */
   (void) wait(NULL);
   (void) wait(NULL);
   remove_shared_memory(0);
   return 0;
}
void writer(int val)
{
   *SharedVariable = val;
}
void reader(void)
{
   int val;
   do {
      val = *SharedVariable;
      printf("Read %d\n", val);
      sleep(1);
   } while (val == 0);
   printf("Terminating, last read value: %d\n", val);
}
void remove_shared_memory(int sig)
{
   /* detach and delete shared memory segment */
   (void) shmdt((char *) SharedVariable);
   (void) shmctl(Id, IPC_RMID, NULL);
   exit(0);
}
```

2.2 Multitasking with Multithreading

Creating a new process is a heavy task for OS, since a lot of resources must be allocated. Creating a new thread within an existing process, on the other hand, is a lightweight task (and much faster). Therefore, it is recommended to achieve multitasking with multithreading (when no isolation between threads is required). Also, threads within the same process share the same memory – the process address space – and can easily communicate through it (for processes, a shared memory mechanism must be used for this, which has already been explained). In both cases, however, a synchronization mechanism is usually required in addition. More about this in the next exercise.

Originally, "old processes" had only a single thread within them. Multiple threads within the same process is a concept that emerged in the mid-1980s. Figure 1 shows processes with one or more threads. Each thread of a process is a separate "scheduling unit" that is scheduled separately by the OS kernel to the available processors.

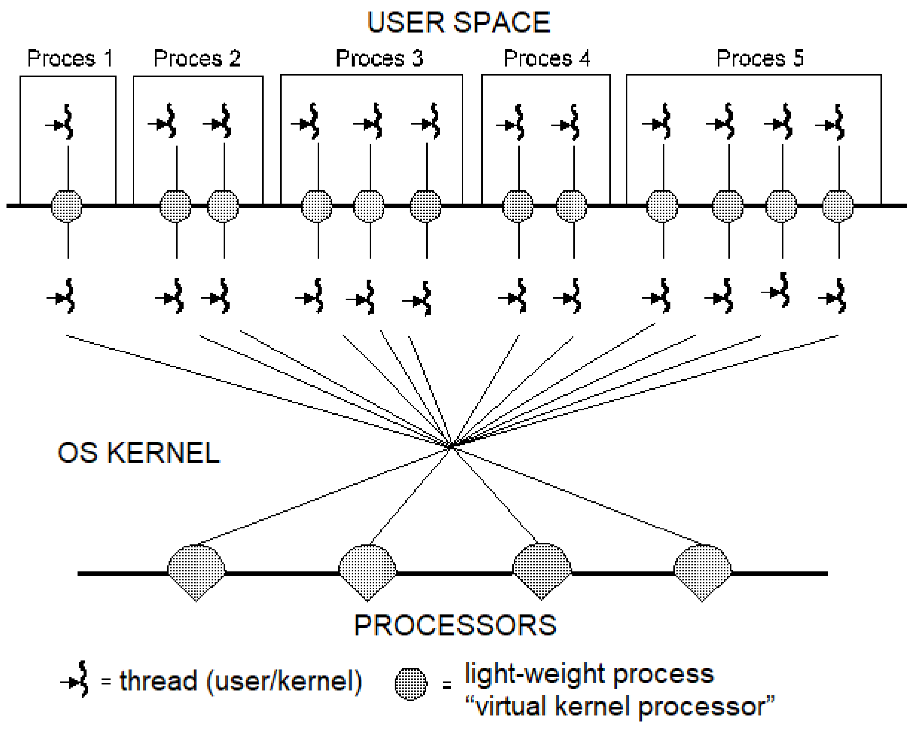

Figure 1. Examples of processes with various number of threads on multiprocessor system

Normally, each thread within a process is scheduled directly by the kernel. However, this is not always the case. In Figure 1, there is a Lightweight Process (LWP) element that maps a process thread to a kernel thread. Some systems and libraries provide thread scheduling within a process, where multiple threads within the process can be scheduled internally to a single LWP - which the kernel sees. This feature is called unbound thread in Solaris OS or fibers in Windows operating systems. This feature is not discussed further.

In the following, POSIX interface will be used to describe thread management.

2.2.1 *Creating a new thread*

Withing existing thread, a new thread can be created with `pthread_create`.

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine)(void *), void *arg);
```

with arguments:

- `thread` – pointer to `pthread_t` element where user-level thread descriptor will be saved when a thread is created
- `attr` – pointer to a structure which can be used to define some thread specific elements, for example, thread priority, scheduling policy, stack address; use `NULL` for default values
- `start_routine` – starting function for new thread – this is where created thread starts its execution
- `arg` – pointer which will be passed as single argument to starting function

If thread is successfully created, its descriptor is saved into `thread`, and function returns zero. Any other return value means that the thread is not created.

2.2.2 *Terminating a thread, waiting for thread termination*

When thread finishes its starting function, when thread exit this function, it has completed assigned job and its terminated. However, thread can be terminated before with a call to `pthread_exit`.

```
int pthread_exit(void *status);
```

Thread exit status is a pointer, contrary to process exit status (which is 1-byte number).

A thread can suspend its execution until another thread terminates with `pthread_join`.

```
int pthread_join(pthread_t thread, void **status);
```

Argument `thread` is a descriptor of thread which termination is waited for. Status of that thread is saved into variable `status`. It's a double pointer since thread exit status is pointer. If `NULL` is provided, no status is received.

2.2.2.1 Example program with threads

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

```
int SharedVariable;

void *writer(void *x);
void *reader(void *x);

int main(void)
{
   int val;
   pthread_t thr_desc[2];

   SharedVariable = 0;
   val = 123;

   /* create threads */
   if (pthread_create(&thr_desc[0], NULL, reader, NULL) != 0) {
      printf("Error with pthread_create!\n");
      exit(1);
   }
   sleep(5);
   if (pthread_create(&thr_desc[1], NULL, writer, &val) != 0) {
      printf("Error with pthread_create!\n");
      exit(1);
   }

   pthread_join(thr_desc[0], NULL);
   pthread_join(thr_desc[1], NULL);

   return 0;
}
void *writer(void *x)
{
   SharedVariable = *((int*)x);
}
void *reader(void *x)
{
   int val;

   do {
      val = SharedVariable;
      printf("Read %d\n", val);
      sleep(1);
   } while (val == 0);

   printf("Terminating, last read value: %d\n", val);
}
```

If POSIX threads are used, flag `-pthread` should be used when compiling. For example:

```
$ gcc lab2.c -pthread -o lab2
```

2.2.3 Main thread

When the process is started, a single thread is created, starting with the function `main` (in C). This thread is the main thread of the process, and when it terminates, the process terminates and so do all other threads within the same process. To test this, try commenting both `pthread_join` in the previous example. Therefore, the main thread normally waits for all other created threads before terminating.

This behavior is typical for programs that are compiled into an executable version (C, C++). Other programming languages, such as Python, may not have this "main thread" feature and the process may remain alive until its last thread is terminated.