# Operating Systems And Applications For Embedded Systems

Processes and Threads

KATEDRA INŻYNIERII KOMPUTEROWEJ

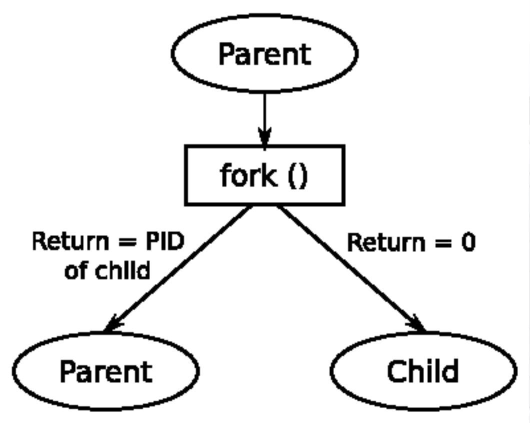
# Plan

- Processes
  - Process definition
  - Creating a new process
  - Output
  - Running a different program
- Threads
  - Thread definition
  - Creating a new thread
  - Terminating a thread
  - Compiling a program with threads
  - Partitioning the problem
  - Scheduling
  - Further reading

# Process definition

A process is a memory address space and a thread of execution, as shown in the following diagram. The address space is private to the process and so threads running in different processes. cannot access it. This memory separation is created by the memory management subsystem in the kernel, which keeps a memory page mapping for each process and re-programs the memory management unit on each context switch.

# Creating a new process I



# Creating a new process II

Here is a simple example, showing process creation and termination:

Listing 1: Listing

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
 int main(void)
 {
 int pid;
 int status;
 pid = fork();
 if (pid == 0) {
    printf("I am the child, PID %d\n", getpid());
```

# Creating a new process III

```
        sleep(10);
        exit(42);
    } else if (pid > 0) {
        printf("I am the parent, PID %d\n", getpid());
        wait(&status);
        printf("Child terminated, status %d\n",
       WEXITSTATUS(status));
    } else
        perror("fork:");
return 0;
}
```

KATEDRA INŻYNIERII KOMPUTEROWEJ

# Output

```
I am the parent, PID 13851
I am the child, PID 13852
Child terminated with status 42
```

# Running a different program I

1. int execl(const char *path, const char *arg, ...);
2. int execlp(const char *file, const char *arg, ...);
3. int execle(const char *path, const char *arg, ..., char * const envp[]);
4. int execv(const char *path, char *const argv[]);
5. int execvp(const char *file, char *const argv[]);
6. int execvpe(const char *file, char *const argv[], char *const envp[]);

# Running a different program II

Listing 2: Listing

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>
int main(int argc, char *argv[])
{
char command_str[128];
int pid;
int child_status;
int wait_for = 1;
while (1) {
    printf("sh> ");
```

# Running a different program III

```
    scanf("%s", command_str);
    pid = fork();
    if (pid == 0) {
        /* child */
        printf("cmd '%s'\n", command_str);
        execl(command_str, command_str, (char *)NULL);
        /* We should not return from execl, so only get
        to this line if it failed */
        perror("exec");
        exit(1);
    }
    if (wait_for) {
        waitpid(pid, &child_status, 0);
        printf("Done, status %d\n", child_status);
    }
```

# Running a different program IV

```
   }
return 0;
}
```

# Thread definition I

A thread is a thread of execution within a process. All processes begin with one thread that runs the main() function and is called the main thread. You can create additional threads using the POSIX threads function pthread_create(3), causing additional threads to execute in the same address space, as shown in the following diagram. Being in the same process, they share resources with each other. They can read and write the same memory and use the same fle descriptors, and so communication between threads is easy, so long as you take care of the synchronization and locking issues.

# Creating a new thread I

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

# Creating a new thread II

Listing 3: Listing

```c
#include <stdio.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/syscall.h>
static void *thread_fn(void *arg)
{
    printf("New thread started, PID %d TID %d\n",
    getpid(), (pid_t)syscall(SYS_gettid));
    sleep(10);
    printf("New thread terminating\n");
    return NULL;
}
int main(int argc, char *argv[])
{
```

# Creating a new thread III

```
    pthread_t t;
    printf("Main thread, PID %d TID %d\n",
    getpid(), (pid_t)syscall(SYS_gettid));
    pthread_create(&t, NULL, thread_fn, NULL);
    pthread_join(t, NULL);
    return 0;
}
```

# Terminating a thread

1. It reaches the end of its start_routine
2. It calls pthread_exit(3)
3. It is canceled by another thread calling pthread_cancel(3)
4. The process which contains the thread terminates, for example, because of a thread calling exit(3), or the process receiving a signal that is not handled, masked or ignored

# Compiling a program with threads

The support for POSIX threads is part of the C library, in the library libpthread.so.
When building a threaded program, you must add the switch –pthread at the compile and link stages.

# Partitioning the problem

1. Keep tasks that have a lot of interaction.
   Minimize overheads by keeping closely inter-operating threads together in one process.
2. Don't put all your threads in one basket.
   On the other hand, try and keep components with limited interaction in separate processes, in the interests of resilience and modularity.
3. Don't mix critical and non-critical threads in the same process.
   This is an amplification of Rule 2: the critical part of the system, which might be the machine control program, should be kept as simple as possible and written in a more rigorous way than other parts. It must be able to continue even if other processes fail. If you have real-time threads, they, by definition, must be critical and should go into a process by themselves.
4. Threads shouldn't get too intimate.
   One of the temptations when writing a multi-threaded program is to intermingle the code and variables between threads because it is all in one program and easy to do. Don't keep threads modular with well-defined interactions.
5. Don't think that threads are for free.
   It is very easy to create additional threads but there is a cost, not least in the additional synchronization necessary to coordinate their activities.
6. Threads can work in parallel.
   Threads can run simultaneously on a multi-core processor, giving higher throughput. If you have a large computing job, you can create one thread per core and make maximum use of the hardware. There are libraries to help you do this, such as OpenMP. You probably

# Scheduling

1. A thread blocks by calling sleep() or in a blocking I/O call
2. A timeshare thread exhausts its time slice
3. An interrupt causes a thread to be unblocked, for example, because of I/O completing

# Further reading

1. The Art of Unix Programming, by Eric Steven Raymond, Addison Wesley; (23 Sept. 2003) ISBN 978-0131429017
2. Linux System Programming, 2nd edition, by Robert Love, O'Reilly Media; (8 Jun. 2013) ISBN-10: 1449339530
3. Linux Kernel Development, 3rd edition by Robert Love, Addison-Wesley Professional; (July 2, 2010) ISBN-10: 0672329468
4. The Linux Programming Interface, by Michael Kerrisk, No Starch Press; (October 2010) ISBN 978-1-59327-220-3
5. UNIX Network Programming: v. 2: Interprocess Communications, 2nd Edition, by W. Richard Stevens, Prentice Hall; (25 Aug. 1998) ISBN-10: 0132974290
6. Programming with POSIX Threads, by Butenhof, David R, Addison-Wesley, Professional
7. Scheduling Algorithm for multiprogramming in a Hard-Real-Time Environment, by C. L. Liu and James W. Layland, Journal of ACM, 1973, vol 20, no 1, pp. 46-61

# References

C. Simmonds.
*Mastering Embedded Linux Programming*.
Packt Publishing, 2015.

# The End