Real-Time Programming TI00AA55

Lecture 13 - 29.04.2015

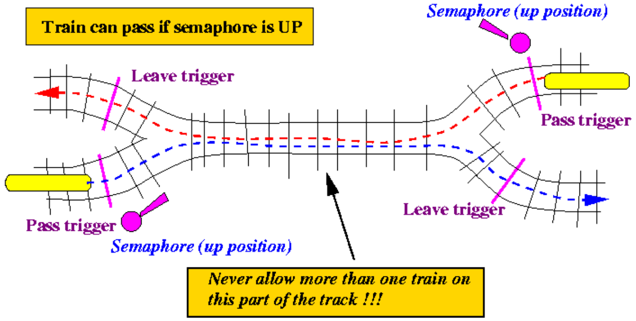
Jarkko.Vuori@metropolia.fi

System V vs. new Posix IPC

- We have recently handled the communication and synchronization of threads
- We have also discussed some inter process communication (IPC) methods like pipe, fifo and socket
- Next we learn semaphore, which is a synchronization method between processes
  - We also learn more IPC tools like shared memory and message queues
- Linux and other Unix systems still have the old System V IPC, that contains semaphore sets, shared memory and message queues
- These are still defined also in the standard
  - In addition to these old methods POSIX defines the new tools for the same purpose
  - These new tools are easier to use, because they have simpler interface
- One difference is that the new instruments can be identified with the (file) name instead of id-number
  - Because of that processes can easily access these instruments
  - File descriptors are used as a handle

Basic idea of semaphore

- A semaphore is used to synchronize the usage of certain resources in the application
  - A semaphore is actually an integer number that tells how many resources are free to use
- If semaphore value is zero, it means that all resources are in use and a process that possibly needs the resource must wait
- If a process needs a resource, it calls the function `sem_wait`
  - If there is a free resource, the function decrements the semaphore value (the number of free resources) and returns immediately
  - The test and the subtraction (read, test, subtract, write) happens atomically
- If the resource is reserved (semaphore is 0), the process is enqueued to the queue of the processes that wait for this semaphore
  - The function blocks and waits until one of the processes has finished the using of the resource
- When the process has finished using the resource, it must call the function `sem_post`
  - This function increments the semaphore value atomically
  - If there are processes that wait for the semaphore, one of them (or all depending on the scheduling) are put in runnable state



Functions sem_wait and sem_post

- It is possible to define one semaphore
  - In System V IPC, you needed to declare a semaphore set even if you needed one semaphore
- The type of the semaphore is sem_t
- POSIX defines two new semaphores types
  - named and
  - unnamed
- The operation functions that are used for the synchronization and that were mentioned on the previous page, are same for unnamed and named semaphores :

```
int sem_wait(sem_t *sem);
int sem_post(sem_t *sem);
```

- Declaration and initialization of unnamed semaphores and named semaphores are different
  - It is easier to use named semaphores for inter process communication
- Lets first study declaration and initialization of named semaphores
  - Named semaphores have a name
- The named semaphore object is owned by the operating system
  - The processes get the handle to the semaphore by opening it
  - The handle is a pointer to the semaphore
  - In Linux it is a file descriptor and for example can be put in the file descriptor set

Opening and closing named semaphores

- Named semaphores have a (file) name and they need to be opened with the function `sem_open`
  - When the semaphore is not needed anymore, the semaphore needs to be closed with the function `sem_close`
  - There is also a function `sem_unlink` that removes the semaphore name from the file system
- The prototypes of these functions are as follows

```
sem_t *sem_open(const char *name, int
oflag, ...);
int sem_close(sem_t *sem);
int sem_unlink(const char *name);
```

- The function `sem_open` works like the ordinary open function in the sense that takes file name and open flags as a parameter
- The principle is that one process creates the semaphore entry in the file system using open flag O_CREAT and giving access rights and initial value to the semaphore
- Other processes can then use the semaphore after opening it for reading or writing or both
- Remark 1. The first parameters of `sem_open` are exactly like the corresponding parameters in the system call open
  - In similar way, the third parameter is similar in the case of the second parameter having the flag O_CREAT
  - In that case there is also fourth parameter that indicates the initial value of the semaphore

Example of named semaphores

- If the function succeeds, it returns a pointer to the semaphore
  - If it fails it returns SEM_FAILED
    - In this case errno tells the reason for the failure
- Remark. In this case the process A first needs to create the semaphore object
  - Otherwise the open in the process B would fail
  - One process at a time uses the resource

```
// Process A
sem_t *semp;
semp = sem_open("/mysemaphore.sem",
                 O_RDWR | O_CREAT,
                 S_IRUSR | S_IWUSR, 1);
sem_wait(semp);
// use the resource
sem_post(semp);
sem_close(semp);
sem_unlink(/mysemaphore.sem);

// Process B
sem_t *semp;
semp=sem_open("/mysemaphore.sem",O_RDWR);
sem_wait(semp);
// use the resource
sem_post(semp);
sem_close(semp);
```

Initialize and destroy unnamed semaphore 1

- Unnamed semaphores are semaphore variables that are declared in the program
  - Before they can be used, one of the processes need to initialize the semaphore variable with the function `sem_init`
  - When the semaphore is not needed anymore it must be destroyed with the function `sem_destroy` to release the resources
- The prototypes of these functions are

```
int sem_init(sem_t *sem, int pshared,
unsigned int value);
int sem_destroy(sem_t *sem);
```

- These functions return 0 in the success and -1 in the failure
  - In the failure the errno variable indicates the reason for the error
- These functions are called only once in one process even the semaphore is used by two or more processes
- The first parameter of the function sem_init is the address of a semaphore variable
  - The semaphore variable must be allocated (from somewhere) before the sem_init function is called
- **If a semaphore is used by different processes (not only by different threads in the same process), it must be allocated from the shared memory** (from the memory, where all processes have an access)

Initialize and destroy unnamed semaphore 2

- The second parameter pshared of the function `sem_init`, tells whether the semaphore is shared between threads in different processes or only between threads in the same process
  - The parameter value ≠ 0 (true) of pshared means that the semaphore is shared between process
    - In that case the semaphore must be allocated from the shared memory
  - The parameter value 0 (false) means that the semaphore is not shared
    - It is used only between threads in the same process
    - In that case the semaphore can be allocated from the memory area of the process (global variable, dynamic memory (or even a local variable of one thread))
- The third parameter of the function `sem_init` is the initial value of the semaphore
- Remark 1. It is probably most obvious to use named semaphores for the synchronization between threads in different processes and unnamed semaphores if all threads are in the same process
- Remark 2. Mutex is actually a kind of binary semaphore
  - Remember that also mutex can be shared between threads in different processes

Example of unnamed semaphore

```
sem_t sem;  //(global)

// Thread 1
sem_init(&sem, 0, 1); // Not shared
sem_wait(&sem);
// use resource
sem_post)&sem);
sem_destroy(&sem);

// Thread 2
sem_wait(&sem);
// use resource
sem_post (&sem);
sem_destroy(&sem);
```

Remark. This works only with the threads in the same process

POSIX shared memory

- A process is a unit of protection
  - We cannot access the memory area of the process B from the process A
- To pass information from one process to an another we need to use IPC methods provided by the operating system
  - So far we have learned pipe, fifo and sockets
- There is an additional IPC method that is called shared memory
  - The basic idea is that one process requests a shared memory object from the kernel
  - After that processes can map that memory object to the address space of their own and use it either for writing or reading or both
- The usage is two phase operation
  - First we need to create a shared memory object
  - Secondly we need to map the memory object to the process
- The shared memory object is created with the function `shm_open` and it is released with the function `shm_unlink`. The prototypes of these functions are:

```
int shm_open(const char *name, int
oflag, mode_t mode);
int shm_unlink(const char *name);
```

- Function `shm_open` returns a file descriptor to the memory object
- The memory object is then mapped to the address space of the process with the function `mmap`

Mapping shared memory object

- When shared memory object has been created, it must be mapped to the address space of the process. This is done with the function `mmap`. The prototype of this function is

```
<sys/types.h>  <sys/mman.h>
void* mmap(void* addr, size_t
len, int prot, int flags, int
fildes, off_t off);
```

- The function returns a pointer which the process can use to access the shared memory object (or -1 in the failure)
- Parameter addr can be used to advice the system to select the address from the address space of the process
  - Most often this parameter is 0 meaning that kernel can select freely the address the `mmap` function returns
- Parameter fildes determines the shared memory object we want to map (or real disk file)
- The parameter len specifies the size of the memory area
- Parameter offset determines the offset in the shared memory object or in the file where the mapping starts from (this is usually 0)

Mapping shared memory object (cont)

- The parameter prot determines the protection of the mapped memory
- Possible options that can be inclusively be or’ed together are

– PROT_READ	can be read
– PROT_WRITE	can be written
– PROT_EXEC	can be executed
– PROT_NONE	No access

- The options must be in harmony with the flags that are used when the shared memory object has been created

- The parameter flags determines the following things:
  - MAP_FIXED The starting address of the memory area must be exactly as specified with the parameter addr
  - MAP_SHARED Writing modifies the file (real file, see next page)
  - MAP_PRIVATE If the program writes to the address, a copy of the file is created (real file, see next page)
- Signal SIGSEGV tells that a process tried to write to the object even it has no access rights to it
- Signal SIGBUS tells that a process tries to write to the memory address that has no memory object anymore (shared memory object or area in real file)

Memory mapped i/o to the disk file

- The function `mmap` is old unix function
- Memory mapped i/o initially has meant that a certain address area is mapped to the disk file so that always when data is stored (assigned) to that address it actually is written to the disk file
- If data is read from that memory address the data actually comes from the disk file
- The functions write and read are not needed to write to or read from the disk file
- It was possible to use memory mapped i/o also without the file!
  - This provided a method to use shared memory in older unix systems
- The concept of shared memory object and the functions `shm_open` and `shm_unlink` are newer features in POSIX
  - It is now recommended to use this new method

Example of POSIX shared memory

```
// Process A
// (Creates and initializes counter in shm)
long int* pcounter_in_shared_mem_area;
int fd_shared_int;

fd_shared_int = shm_open("/sharedmem.shm",
          O_CREAT|O_RDWR,
          S_IRUSR|S_IWUSR);
ftruncate(fd_shared_int, sizeof(long int));
pcounter_in_shared_mem_area=(long int*)mmap(
          0  sizeof(int),
           PROT_READ | PROT_WRITE,
          MAP_SHARED, fd_shared_int, 0);
*pcounter_in_shared_mem_area = 0;
close(fd_shared_int);
shm_unlink("/sharedmem.shm");
```

```
// Process B
// (Increments the counter in shared mem)
long int* pcounter_in_shared_mem_area;
int fd_shared_int;

fd_shared_int = shm_open("/sharedmem.shm",
          O_RDWR);
pcounter_in_shared_mem_area=(long int*)mmap(
          0, sizeof(int),
          PROT_READ | PROT_WRITE,
          MAP_SHARED, fd_shared_int, 0);
*pcounter_in_shared_mem_area++;
close(fd_shared_int);
```

Posix message queue basics

- We have learned IPC methods pipe, fifo and socket
  - They have no message boundaries
  - They are streams of bytes
  - Message queues have clearly defined message boundaries
  - When message has been sent it is received exactly of the length as is specified in the attribute of the message
- System V has message queues with similar interface than System V semaphores
  - Newer Posix version defines new message queues with a more modern and simpler interface
- These new message queues also utilize real time signals
  - It is possible for a process to request a signal when message has arrived
- Message queues have a name in the file system
  - This means that all processes having access rights to that name can use the message queue (read messages from the message queue or write messages to the message queue)

Opening and closing message queues 1

- Before the message queue can be used it must be created in the file system
  - The procedure is like the creation of the disk file
  - One process creates the message queue and gives it a name and access rights
  - The message queue is created with the function `mq_open`
- Other processes can then open the message queue for writing and/or reading
  - The function is again `mq_open`
- Opening the message queue returns a message queue descriptor that is used to identify it
  - The type of message queue descriptor is mqd_t
- When the process does not need the message queue any more, it closes the message queue with the function `mq_close`
  - Message queue name can be removed from the file system calling `mq_unlink`
- See the prototypes on the next page
- Remark. In Linux message queue descriptor (mqd_t) is like ordinary file descriptor
  - It can be put in a file descriptor set that is passed to the select function
  - This is not necessarily portable for all Unix systems

Opening and closing message queues 2

- The prototypes of these functions are
```
mqd_t mg_open(const char *name, int
oflag, ...);
int mq_close(mqd_t mqdesc);
int mq_unlink(const char *name);
```
- The first two parameters of `mq_open` are exactly like the corresponding parameters in the function `sem_open`
- In similar way the third parameter (access rights) is same in the case of the second parameter having the flag O_CREAT
  - If O_CREAT flag is set, the function has also fourth parameter that is a pointer to the attribute structure of type struct mq_attr
- Attribute structure specifies certain features and properties of the message queue (see the page 19)
- If you pass NULL as an attribute parameter, the message queue has default properties (see page 19)
  - The default properties are usually not suitable for the application
- The function `mq_open` returns the descriptor of the message queue in the success
- The function `mq_open` returns -1 in the failure

Sending and receiving messages

- When the message queue has been opened and a process has a valid message queue descriptor, it can send or receive messages with functions `mq_send` and `mq_receive`. The prototypes of these functions are

```
int mq_send(mqd_t mqd, const char
*msg_ptr, size_t msg_len, unsigned int
msg_prio);

int mq_receive(mqd_t mqd, char
*msg_ptr, size_t msg_len, unsigned int
*msg_prio);
```

- The prototypes are quite self evident. The only thing that needs explanation is the last parameter msg_prio
- Messages have priorities
  - Priority is given when the message is sent
- When a process reads from the message queue, messages are received in priority order
  - When you read a message, you always get the message with the highest priority
  - The priority of the message can be found using the last parameter of the `mq_receive` function

Message queue attribute 1

- When the message queue is created, we can pass a parameter struct mq_att* to the open function that specifies the attributes of the message queue
  - It is also later possible to find out and modify the properties of the message queue
- It is possible to get and set the attributes of a message queue with the functions `mq_getattr` and `mq_setattr`. The prototypes are

```
int mq_getattr(mqd_t mq,  struct
mq_attr *attr);
int mq_setattr(mqd_t mq, const struct
mq_attr *attr, struct mq_attr
*oldattr);
```

- The definition of the structure struct mq_attr is as follows:

```
struct mq_attr {
  long mq_flags;  //Flags: 0 or O_NONBLOCK
  long mq_maxmsg; //Max # of messages on queue
  long mq_msgsize;//Message size (bytes)
  long mq_curmsgs;//# of msg currently in queue
};
```

Message queue attribute 2

- If you pass NULL as the last parameter to the mq_open, attribute takes default values
  - Default value for the message size is 8192 and default value for the maximum number of messages is 10
- If the message queue is full the `mq_send` blocks until there is room in the message queue

- These are experiences in Linux:
  - If you send a shorter message that is indicated in the attribute `mq_send` succeeds, but the rest in the message is rubbish
  - If you send a longer message that is indicated in the attribute `mq_send` succeeds, but the `mq_receive` blocks?
  - If you try to read a longer message that is indicated in the attribute, `mq_receive` succeeds, but you get the message of the length indicated in the attribute
  - If you try to read a shorter message that is indicated in the attribute, `mq_receive` causes an error “Message too long”

Message queue and signals

- It is possible to ask the message queue to send a signal to the process when a message has arrived
- This is done with the function `mq_notify`. The prototype of this function is

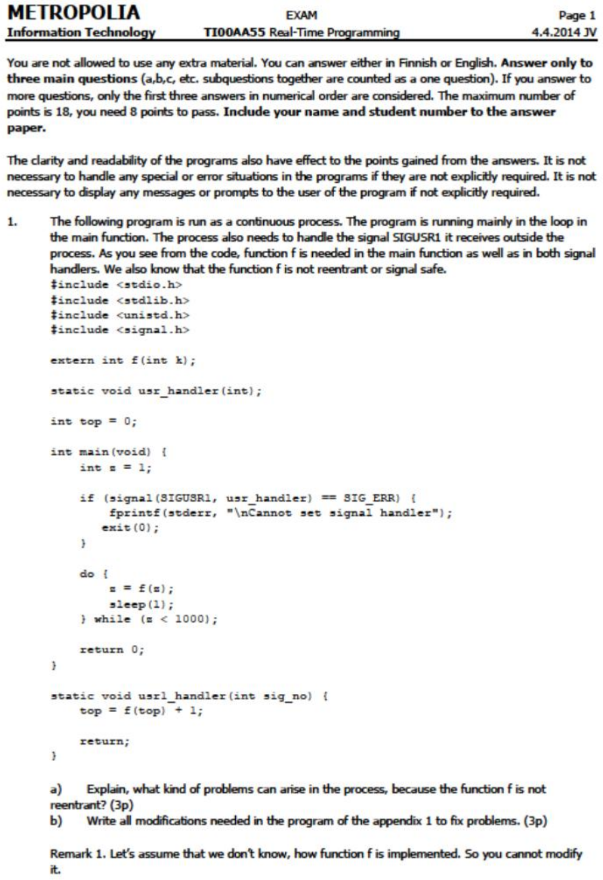
  ```
  int mq_notify(mqd_t  mqdesc,
  const struct sigevent
  *notification);
  ```

```
// Signal handler
void ready_handler(int signo,
                   siginfo_t* siginfo,
                     void*) {
// read a message from the msg queue
}
// Open the message queue
// Install the signal handler
// Ask a notification about data in message queue
struct sigevent sigevent;
sigevent.sigev_notify = SIGEV_SIGNAL;
sigevent.sigev_signo = SIGRTMIN;
sigevent.sigev_value.sival_int = mq;// descriptor
mq_notify(mq, &sigevent);
```

The exam

- The examination is held on Thu 13.5.2015 10:00 – 12:00 ETYA1103, the exam will last two hours (2h)
  - After the exam we will have lab session normally in order to check all the remaining labs
- After 30 min from the exam beginning you cannot enter the classroom
- You can leave after 45 min from the exam beginning
- The exam is arranged in which the understanding of the topics and the ability to apply them is tested
- Final grade is based on the exam, look for more details at Lecture 1, Slide 5
- The problems to be solved in the exam have similarities in nature with the problems worked out in labs, but labs don't give straight answers to questions in the exam
  - On the next slide we have a demo exam (in the real exam, we'll have four questions available, and you need to answer three of them)
  - Small syntax errors are not penalized, e.g. `stdev = sqrt(accum/(running_times.size()-1);`
  - Additional handnotes (exam_handnote1.pdf and exam_handnote2.pdf) are given at the exam (you don't need to memorize all those nasty details of programmer's API)
- Good luck (for the exam)!

The exam



METROPOLIA EXAM Page 1

Information Technology **TI00AA55** Real-Time Programming 4.4.2014 JV

You are not allowed to use any extra material. You can answer either in Finnish or English. **Answer only to three main questions** (a,b,c, etc. subquestions together are counted as a one question). If you answer to more questions, only the first three answers in numerical order are considered. The maximum number of points is 18, you need 8 points to pass. **Include your name and student number to the answer paper.**

The clarity and readability of the programs also have effect to the points gained from the answers. It is not necessary to handle any special or error situations in the programs if they are not explicitly required. It is not necessary to display any messages or prompts to the user of the program if not explicitly required.

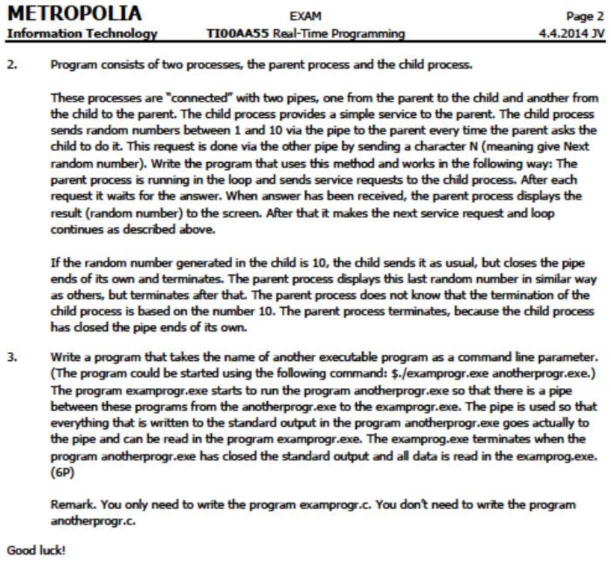
1. The following program is run as a continuous process. The program is running mainly in the loop in the main function. The process also needs to handle the signal SIGUSR1 it receives outside the process. As you see from the code, function f is needed in the main function as well as in both signal handlers. We also know that the function f is not reentrant or signal safe.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>

extern int f(int k);

static void usr_handler(int);

int top = 0;

int main(void) {
    int s = 1;

    if (signal(SIGUSR1, usr_handler) == SIG_ERR) {
        fprintf(stderr, "\nCannot set signal handler");
       exit(0);
    }

    do {
        s = f(s);
        sleep(1);
    } while (s < 1000);

    return 0;
}

static void usr1_handler(int sig_no) {
    top = f(top) + 1;

    return;
}
```

a) Explain, what kind of problems can arise in the process, because the function f is not reentrant? (3p)

b) Write all modifications needed in the program of the appendix 1 to fix problems. (3p)

Remark 1. Let's assume that we don't know, how function f is implemented. So you cannot modify it.



METROPOLIA EXAM Page 2

Information Technology **TI00AA55** Real-Time Programming 4.4.2014 JV

2. Program consists of two processes, the parent process and the child process.

   These processes are "connected" with two pipes, one from the parent to the child and another from the child to the parent. The child process provides a simple service to the parent. The child process sends random numbers between 1 and 10 via the pipe to the parent every time the parent asks the child to do it. This request is done via the other pipe by sending a character N (meaning give Next random number). Write the program that uses this method and works in the following way: The parent process is running in the loop and sends service requests to the child process. After each request it waits for the answer. When answer has been received, the parent process displays the result (random number) to the screen. After that it makes the next service request and loop continues as described above.

   If the random number generated in the child is 10, the child sends it as usual, but closes the pipe ends of its own and terminates. The parent process displays this last random number in similar way as others, but terminates after that. The parent process does not know that the termination of the child process is based on the number 10. The parent process terminates, because the child process has closed the pipe ends of its own.

3. Write a program that takes the name of another executable program as a command line parameter. (The program could be started using the following command: $./examprogr.exe anotherprogr.exe.) The program examprogr.exe starts to run the program anotherprogr.exe so that there is a pipe between these programs from the anotherprogr.exe to the examprogr.exe. The pipe is used so that everything that is written to the standard output in the program anotherprogr.exe goes actually to the pipe and can be read in the program examprogr.exe. The examprog.exe terminates when the program anotherprogr.exe has closed the standard output and all data is read in the examprog.exe. (6P)

   Remark. You only need to write the program examprogr.c. You don't need to write the program anotherprogr.c.

Good luck!