Introduction into Multicore Programming

Károly Bósa
(Karoly.Bosa@jku.at)

Research Institute for Symbolic Computation
(RISC)

Dates of the Final Exam

- There will be a **written exam**.

- Dates:
  - 10th of February (Thursday) 10:15 – 11:45 K 224B
  - 24th of February (Thursday) 10:15 – 11:45 K 224B

Topic

Communication (IPC and ITC)

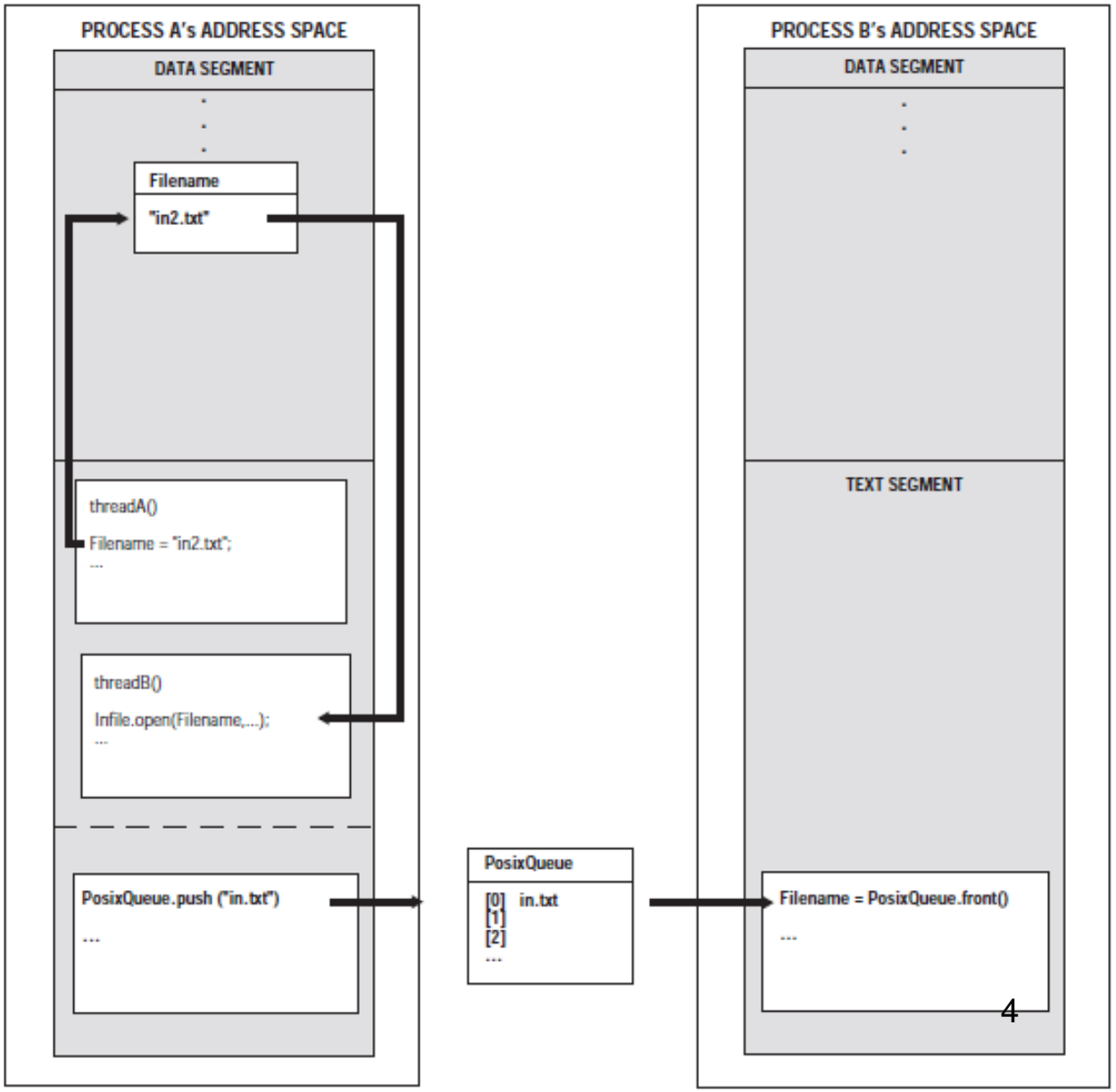
- Interprocess Communication (IPC)
  - Persistence of IPC
  - Types of IPC
  - Names of POSIX Objects
  - Environment Variables and Command Line Arguments
  - Using Files for Communication
  - Shared Memory
  - Pipes
  - Message Queues
- Interthread Communications (ITC)
  - Usage of Global Variables
  - Parameters for Interthread Communication
- An addition: Signals

Unidirectional Communication

Karoly.Bosa@jku.at

- An example: Process A sends a message to B which contains the names of the files that B intends to process.

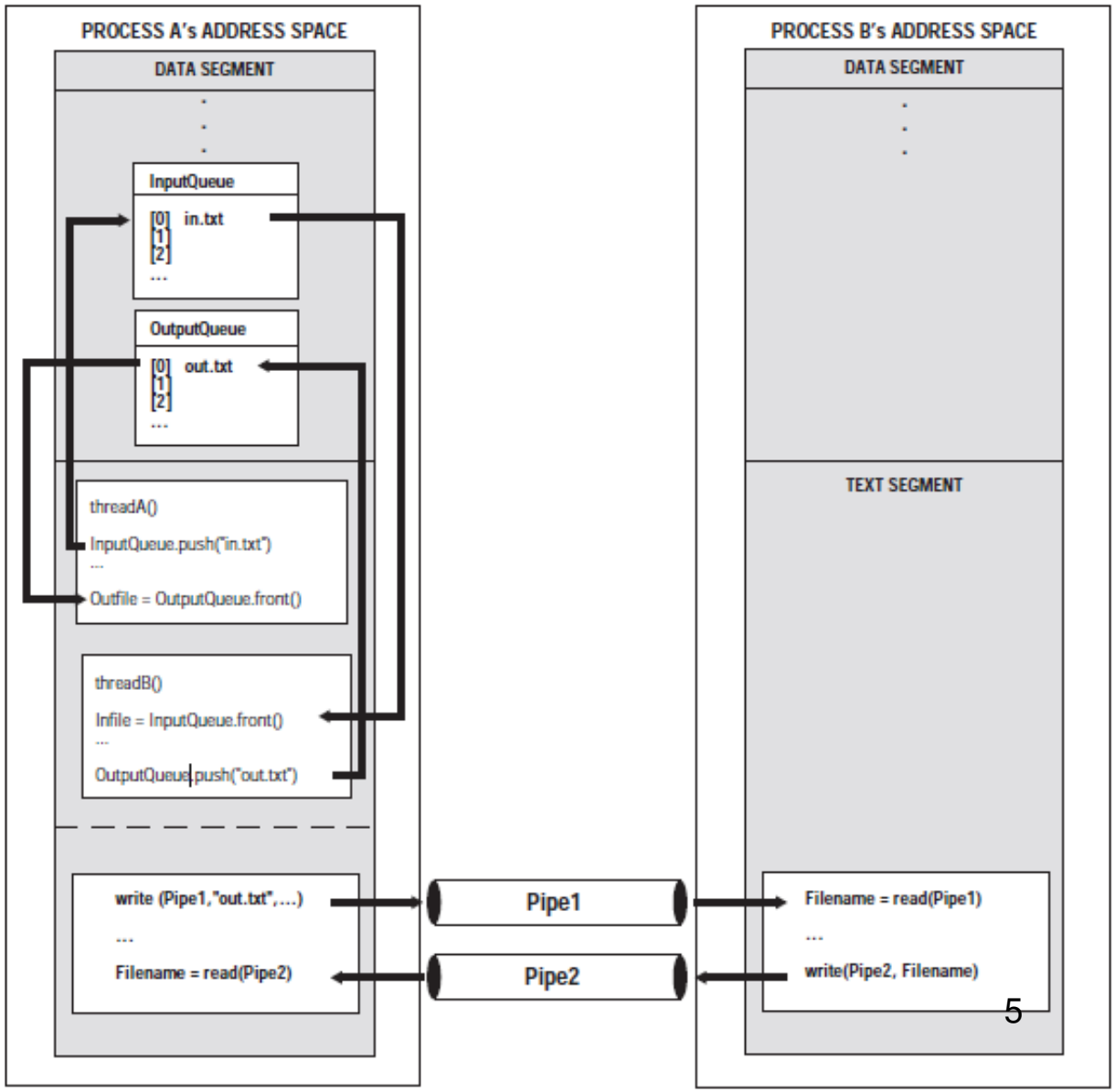
- Another example: thread A would write the name of the file to a global variable, and thread B would simply read that variable.



Bidirectional Communication

Karoly.Bosa@jku.at

- An example: Process
  - A use pipe 1 to send the name of the file that process B has to process.
  - Outcome is written into another file whose name is sent back to process via pipe 2.
- Another example: Thread A and thread B can use two global data structures for the same purpose.



Interprocess Communication (IPC)

- Processes have their own address space.
  - Data that is declared in one process is not available in another process.
  - Events that happen in one process are not known to another process.
  - The data in the stack and data segments or is dynamically allocated in memory protected from other processes.
- The operating system APIs by which a process can sends data or events to another process is **called *Interprocess Communication (IPC)*** mechanisms**.**
- The operating system's kernel acts as the communication channel between the processes, e.g.:
  - posix_queue
  - Files for communication
  - Shared memory

Persistence of IPC

- It *refers to the existence of an object during or beyond the execution of the* program, process, or thread that created it.
- IPC entities reside in the filesystem, in kernel space, or in user space, and persistence is also defined the same way:
  - **Filesystem persistence:** IPC object exist until they are deleted explicitly. If the kernel is rebooted, the objects will keep its value.
  - **Kernel persistence**: IPC objects remain in existence until the kernel is rebooted or the object is deleted explicitly.
  - **Process persistence**: IPC objects exists until the process that created the object finishes its running.

Types of IPC

Type of IPC	Name space	Persistence	Process
Pipe	unnamed	process	Related
FIFO	pathname	process	Both
Mutex	unnamed	process	Related
Condition variable	unnamed	process	Related
Read-write locks	unnamed	process	Related
Message queue	Posix IPC name	kernel	Both
Semaphore (memory-based)	unnamed	process	Related
Semaphore (named)	Posix IPC name	kernel	Both
Shared memory	Posix IPC name	kernel	Both

- For processes that are not related (e.g.: parent-child), named IPC objects are available.
- For IPCs that require a POSIX IPC name, that name must begin with a slash and contain no other slashes.

- To create the IPC object, one must have write permissions for the directory.

Environment Variables and Command Line Arguments

Karoly.Bosa@jku.at

- Parent processes share their resources with child processes.
- By using posix_spawn, or the exec functions, the parent process can create the child process:
  - with exact copies of its environment variables or
  - initialize them with new values.

```
int posix_spawn(pid_t *restrict pid, const char *restrict path,
                const posix_spawn_file_actions_t *file_actions,
                const posix_spawnattr_t *restrict attrp,
                char *const argv[restrict], char *const envp[restrict]);
```

- *argv[] and envp[]* are used to pass a list of command line argument and environment variables to the new process.
- This is one-way, one-time communication.

Using Files for Communication

- Using files to transfer data between processes is one of the simplest and most flexible means of transferring or sharing data.

- When you use files to communicate between processes, you follow seven basic steps :

  1. The name of the file has to be communicated.
  2. You must verify the existence of the file.
  3. Be sure that the correct permission are granted to access to the file.
  4. Open the file.
  5. Synchronize access to the file.
  6. While reading/writing to the file, check to see if the stream is good and that it's not at the end of the file.
  7. Close the file.

- **Remark:** files have filesystem persistence; in this case, the persistence can survive a system reboot.

File Descriptors

- File descriptors are unsigned integers used by a process to identify an open file.
- They are indices to the file descriptor table, a block maintained by the kernel for each process.
- When a child process is created, the descriptor table is copied for the child process,
- which allows the child process to have equal access to the files used by the parent.
- The number of file descriptors that can be allocated to a process is restricted by a resource limit.
- The limit can be changed by setrlimit().

Shared Memory

- A block of shared memory can be used to transfer information between processes.
- The shared block of memory is separate from the address space of the processes.
- A process gains access to the shared memory by temporarily connecting the shared memory block to its own memory block.
- Shared memory can be written to and read by only a single process as well and held open by that process.
- Other processes can attach to and detach from the shared memory any time.
- The shared memory related function calls can map either **files** or **internal memory** to the shared memory region.
- In case of usage of shared memory, synchronization is required (otherwise data race can occur).

Using POSIX Shared Memory

Karoly.Bosa@jku.at

```
#include <sys/mman.h>

void  *mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
int mumap(void *addr, size_t len);
```

- The function maps *len* bytes starting at offset *offset* in the file or other object specified by the file descriptor *fd* into memory, preferably at address *addr*.
- The *addr* is usually specified as 0. The actual place where the object is mapped in memory is returned and is never 0.
-

Flag Arguments for mmap	Description
prot	Describes the protection of the memory-based region.
PROT_READ	Data can be read.
PROT_WRITE	Data can be written.
PROT_EXEC	Data can be executed.
PROT_NONE	Data is inaccessible.
flags	Describes how the data can be used.
MAP_SHARED	Changes are shared.
MAP_PRIVATE	Changes are private.
MAP_FIXED	addr is interpreted exactly.

An Example of Memory Mapping with a File


```
fd = open(file_name,O_RDWR);
ptr = casting<type>(mmap(NULL,sizeof(type),PROT_READ,MAP_SHARED,fd,0));
```

- To create a shared memory:
  - open a file and store the file descriptor;
  - then call mmap() with the appropriate arguments and store the returning void * .
- Use a semaphore when accessing the variable.
- The void * may have to be type cast. depending on the data you are trying to manipulate.

Using Shared Memory with Internal Memory I.

Karoly.Bosa@jku.at

- A function that creates a shared memory object is used instead of a function that opens a file:

```
#include <sys/mman.h>

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

- The shm_open() creates and opens a new or opens an existing POSIX shared memory object.
- To ensure the portability of name use an initial slash (/) and don't use embedded slashes.
- shm_open() returns a new file descriptor referring to the shared memory object.
- E.g.:

```
fd = sh_open(memory_name,O_RDWR, MODE);
ptr = casting<type>(mmap(NULL,sizeof(type),PROT_READ,MAP_SHARED,fd,0));
```

Using Shared Memory with Internal Memory II.

Karoly.Bosa@jku.at

Shared Memory Arguments	Description
oflag	Describes how the shared memory will be opened.
O_RDWR	Opens the object for read or write access.
O_RDONLY	Opens the object for read-only access.
O_CREAT	Creates the shared memory if it does not exist.
O_EXCL	Checks for the existence and creation of the object. If O_CREAT is specified and the object exists with the specified name, then returns an error.
O_TRUNC	If the shared memory object exists, then truncates it to zero bytes.
mode	Specifies the permission.
S_IRUSR	User has read permission.
S_IWUSR	User has write permission.
S_IRGRP	Group has read permission.
S_IWGRP	Group has write permission.
S_IROTH	Others have read permission.
S_IWOTH	Others have write permission.

Remark: Definitions of these flag values can be obtained by including <fcntl.h> and mode constant by including <sys/stat.h>

Creating POSIX Shared Memory


```
#include <sys/mman.h>
int shm_open(const char *name, int flags, mode_t mode);
int shm_unlink(const char *name);
```

- *shm_open()* open (or create) a shared memory object with the given POSIX name.
- The flags argument instructs on how to open the object: the most relevant ones are
  - O_RDONLY xor O_RDWR for access type, and
  - O_CREAT for create if object doesn't exist.
- mode is only used when the object is to be created, and specifies its access permission.
- *shm_unlink()* removes a shared memory object specified by name.
- If some process is still using the shared memory segment associated to *name*, the segment is not removed until all references to it have been closed.

Shared Memory Example - Server

Karoly.Bosa@jku.at

```
#include <stdio.h>
 /* shm_* stuff, and mmap() */
#include <sys/mman.h>
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdlib.h>
#include <time.h>

#define SHMOBJ_PATH        "/foo1234"
#define MAX_MSG_LENGTH     50
/* how many types of messages we
recognize (fantasy) */
#define TYPES              8

struct msg_s {
   int type;
   char content[MAX_MSG_LENGTH];
};

int main(int argc, char *argv[]) {
   int shmfd;
   int shared_seg_size = (1 * sizeof(struct msg_s));
   struct msg_s *shared_msg;
```

```
 /* creating the shared memory object    --  shm_open()  */
shmfd = shm_open(SHMOBJ_PATH, O_CREAT |
            O_EXCL | O_RDWR, S_IRWXU | S_IRWXG);
if (shmfd < 0) { perror("In shm_open()"); exit(1); }
fprintf(stderr, "Created shared memory object %s\n",
            SHMOBJ_PATH);
/* adjusting mapped size (make room for the whole */
ftruncate(shmfd, shared_seg_size);

/* requesting the shared segment    --  mmap() */
shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size,
     PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
if (shared_msg == NULL) { perror("In mmap()"); exit(1); }
fprintf(stderr, "Shared memory segment allocated correctly (%d
            bytes).\n", shared_seg_size);

 srandom(time(NULL));
shared_msg->type = random() % TYPES;
snprintf(shared_msg->content, MAX_MSG_LENGTH, "My
    message, type %d.", shared_msg->type);

/* [uncomment if you wish] requesting the removal of the shm
object    --  shm_unlink() */
/* if (shm_unlink(SHMOBJ_PATH) != 0) {
      perror("In shm_unlink()"); exit(1); } */
return 0;
}
```

Shared Memory Example - Client

Karoly.Bosa@jku.at

```
#include <stdio.h>
#include <unistd.h>          /* exit() etc */
#include <sys/mman.h>    /* shm_* stuff, and mmap() */
#include <sys/types.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdlib.h>          /* for random() stuff */
#include <time.h>

#define SHMOBJ_PATH        "/foo1423"
#define MAX_MSG_LENGTH      50
#define TYPES               8

struct msg_s {                  /* message structure */
   int type;
   char content[MAX_MSG_LENGTH];
};

int main(int argc, char *argv[]) {
   int shmfd;

 /* want shared segment capable of storing 1 message */
   int shared_seg_size = (1 * sizeof(struct msg_s));
   struct msg_s *shared_msg;

   /* openning the shared memory object    --  shm_open()  */
   shmfd = shm_open(SHMOBJ_PATH, O_RDWR, S_IRWXU |
               S_IRWXG);
   if (shmfd < 0) { perror("In shm_open()"); exit(1); }
   printf("Created shared memory object %s\n",
               SHMOBJ_PATH);

   /* requesting the shared segment    --  mmap() */
   shared_msg = (struct msg_s *)mmap(NULL, shared_seg_size,
     PROT_READ | PROT_WRITE, MAP_SHARED, shmfd, 0);
   if (shared_msg == NULL) { perror("In mmap()"); exit(1); }
   printf("Shared memory segment allocated correctly (%d
               bytes).\n", shared_seg_size);

   printf("Message type is %d, content is: %s\n",
               shared_msg->type, shared_msg->content);
   return 0;
}
```

Compile with:

```
gcc --ansi --Wall -o shm_msgserver shm_msgserver.c
gcc --ansi --Wall -o shm_msgclient shm_msgclient.c
```

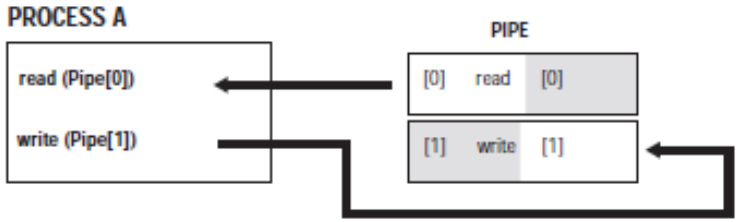
Pipes I.

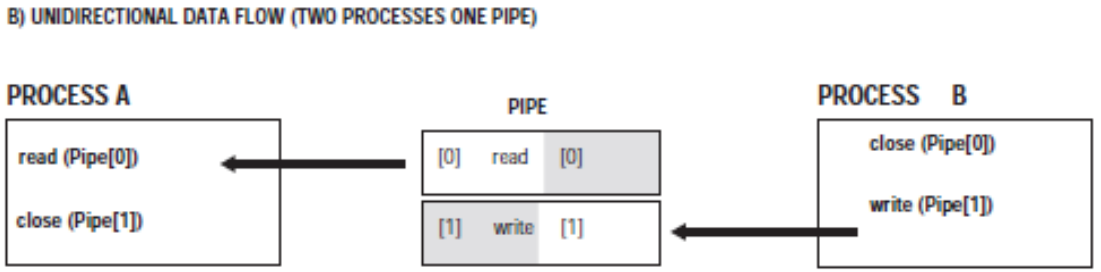
- Pipes are communication channels used to transfer data between two processes.
- data transfer using pipes require the sending and receiving of data to be active at the same time.
- The general rule is:
  - One process (the writer) opens or creates the pipe and
  - then blocks until another process (the reader) opens the same pipe for reading.
- There are two kinds of pipes:
  - Anonymous (only to transfer data between related processes(e.g.: parent-child relationship)).
  - Named (also called FIFO).
- Named pipes have process persistence(the data itself), but the file structure has filesystem persistence).

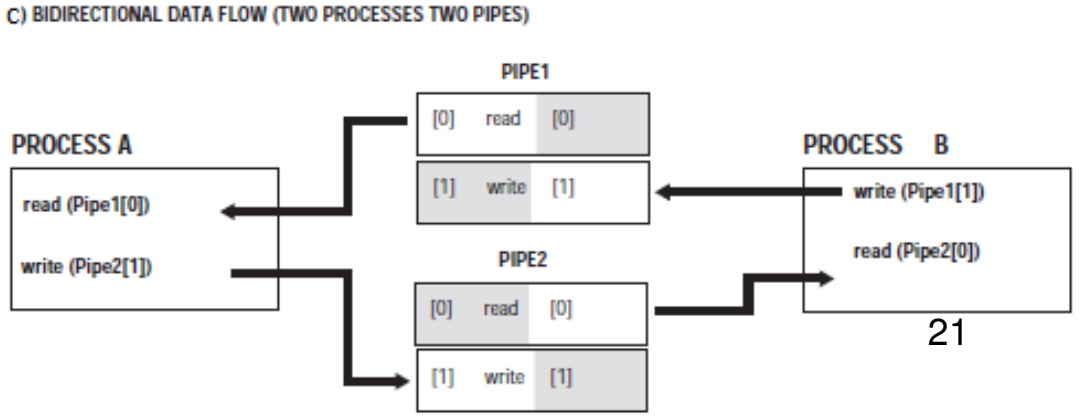
Pipes II.

Karoly.Bosa@jku.at

- Pipes create a flow of data from one end in one process (input end) to the other end that is in another process (output end).
- The data becomes a stream of bytes that flows in one direction through the pipe.
- Two pipes can be used to create bidirectional flow of communication between the processes.







Pipes III.

- Anonymous pipes are temporary and exist only while the process that created them has not terminated.

- Named pipes (FIFO) are special types of files and exist in the filesystem:
  - A program that creates it can finish executing and leave it in the filesystem,

  - but the data that was placed in the pipe will not be present.

  - Future programs and processes can then access the named pipe later, writing new data to the pipe.

- Named pipes have file permission settings associated with them and anonymous pipes do not.

Using Named Pipes (FIFO)


```
#include <sys/types.h>
#include <sys/stat.h>

int mkfifo(const char *pathname, mode_t mode);
int unlink(const char *pathname);
```

- mkfifo() creates a named pipe using pathname as the name of the FIFO with permission specified by mode.
- It is always created with O_CREAT | O_EXCL flags, which means
  - it creates a new named pipe with the name specified if it does not exist;
  - if is does exist, an error EEXIST is returned.
- So, if you want to open an already existing named pipe,
  - call the function, and check for this error;
  - if the error occurs then use open() instead of mkfifo().
- The unlink() removes the filename *pathname* from the filesystem.

Example for FIFO – A Writer Code

- The program creates a named pipe.
- then opens the pipe with an fstream object called Pipe.
- Pipe has been opened for output using the ios::out flag.
- Although Pipe is ready for input, it blocks (waits) until another process has opened for reading.

```
1  using namespace std;
2  #include <iostream>
3  #include <fstream>
4  #include <sys/wait.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7
8  int main(int argc,char *argv[],char *envp[])
9  {
10
11    fstream Pipe;
12
13    if(mkfifo("Channel-one",S_IRUSR | S_IWUSR
14                                    | S_IRGRP
15                                    | S_IWGRP) == -1){
16       cerr << "could not make fifo" << endl;
17    }
18
19    Pipe.open("Channel-one",ios::out);
20    if(Pipe.bad()){
21       cerr << "could not open fifo" << endl;
22    }
23    else{
24          Pipe << "2 3 4 5 6 7 " << endl;
25
26    }
27
28    return(0);
29 }
```

Example for FIFO – A Reader Code

- Pipe opening for input and output as well (in line 16) will prevent deadlock (in case of the unsafe of fstream).

```
1  using namespace std;
2  #include <iostream>
3  #include <fstream>
4  #include <string>
5  #include <sys/wait.h>
6  #include <sys/types.h>
7  #include <sys/stat.h>
8
9
10 int main(int argc, char *argv[])
11 {
12    int type;
13    fstream NamedPipe;
14    string Input;
15
16    NamedPipe.open("Channel-one",ios::in | ios::out);
17
18    if(NamedPipe.bad()){
19       cerr << "could not open Channel-one" << endl;
20    }
21
22    while(!NamedPipe.eof() && NamedPipe.good()){
23
24       getline(NamedPipe,Input);
25       cout << Input << endl;
26    }
27    NamedPipe.close();
28    unlink("Channel-one");
29    return(0);
30
31 }
```

Message Queues

- A message queue is a linked list of strings or messages
- This IPC mechanism allows processes with the adequate permissions to the queue to write or remove messages.
- With a message queue, a writer process can write to the message queue and then terminate. The data is retained in the queue.
- The message queue has kernel persistence.
- When reading a message from the queue, the oldest message with the highest priority is returned.
- Each message in the queue has these attributes:
  - A priority
  - The length of the message
  - The message or data

Using Message Queue I.


```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag,mode_t mode,struct mq_attr *attr);
int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

- mq_open() creates a message queue with the specified name .
- The message queue uses *oflag* with these possible values to specify the access modes:
  - O_RDONLY : Open to receive messages
  - O_WRONLY : Open to send messages
  - O_RDWR : Open to send or received messages
- Any combination of the remaining flags may be specified in the value of *oflag*:
  - O_CREAT : Create a message queue.
  - O_EXCL : The function fails if the pathname already exists.
  - O_NONBLOCK : Determines if queue waits for resources or messages that are not currently available.

Using Message Queue II.


```
#include <mqueue.h>
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag,mode_t mode,struct mq_attr *attr);
int mq_close(mqd_t mqdes);
int mq_unlink(const char *name);
```

- The last parameter is an attribute structure that describes the properties of the message queue :

```
struct mq_attr {
        long mq_flags; //flags
        long mq_maxmsg; //maximum number of messages allowed
        long mq_msgsize; //maximum size of message
        long mq_curmsgs; //number of messages currently in queue
}
```

- mq_close() closes the message queue, but the message queue still exists in the kernel.

- unlink() removes the message queue specified by name from the system.

Using Message Queue III.


```
#include <mqueue.h>

int mq_getattr(mqd_t mqdes,struct mq_attr *attr);
int mq_setattr(mqd_t mqdes,struct mq_attr *attr,struct mq_attr *oattr);
```

- When you are setting the attribute with mq_setattr , only the mq_flags are set in the attr structure.
- Other attributes are not affected. mq_maxmsg and mq_msgsize are set when the message queue is created.
- mq_curmsg can be returned and not set.
- oattr contains the previous values for the attributes.

Using Message Queue IV.


```
#include <mqueue.h>

int mq_send(mqd_t mqdes, const char *ptr, size_t len, unsigned int prio);
ssize_t mq_receive(mqd_t mqdes, const char *ptr, size_t len,unsigned int priop);
```

- The message is stored in *ptr.
- The prio argument is a non-negative integer that specifies the priority of this message:
  - Messages are placed on the queue in decreasing order of priority,
  - with newer messages being placed after older messages with the same priority.
- If the message queue is already full then, mq_send() blocks until sufficient space becomes available.
- For mq_receive(), the len must be at least the maximum size of the message. If prio is not NULL, it is used to return the priority associated with the received message.

Message Queue Example - Sender

Karoly.Bosa@jku.at

```
#include <stdio.h>
#include <mqueue.h>  /* mq_* functions */
#include <sys/stat.h>
#include <stdlib.h>      /* exit() and atoi() */
#include <unistd.h>      /* getopt() */
#include <time.h>        /* ctime() and time() */
#include <string.h>      /* strlen() */

#define MSGQOBJ_NAME    "/myqueue123"
#define MAX_MSG_LEN     70

int main(int argc, char *argv[]) {
   mqd_t msgq_id;
   unsigned int msgprio = 0;
   pid_t my_pid = getpid();
   char msgcontent[MAX_MSG_LEN];
   int create_queue = 0;
   char ch;            /* for getopt() */
   time_t currtime;
   while ((ch = getopt(argc, argv, "qp:")) != -1) {
      switch (ch) {
         case 'q':   create_queue = 1;
            break;
         case 'p':   /msgprio = atoi(optarg);
            break;
         default:
                  printf("Usage: %s [-q] -p msg_prio\n“,
                  argv[0]);
                  exit(1);
         }
   }
   if (create_queue) {
         msgq_id = mq_open(MSGQOBJ_NAME,
         O_RDWR | O_CREAT | O_EXCL, S_IRWXU |
         S_IRWXG, NULL);
   } else {
         msgq_id = mq_open(MSGQOBJ_NAME,
         O_RDWR);
   }
   if (msgq_id == (mqd_t)-1) {
         perror("In mq_open()");
         exit(1);
   }
   currtime = time(NULL);
    snprintf(msgcontent, MAX_MSG_LEN, "Hello from
   process %u (at %s).", my_pid, ctime(&currtime));

   mq_send(msgq_id, msgcontent, strlen(msgcontent)+1,
   msgprio);

   mq_close(msgq_id);
    return 0;}
```

Message Queue Example - Receiver

Karoly.Bosa@jku.at

```
#include <stdio.h>
#include <mqueue.h> /* mq_* functions */
#include <stdlib.h> /* exit() */
#include <unistd.h> /* getopt() */
#include <string.h>  /* strlen() */

#define MSGQOBJ_NAME    "/myqueue123"
#define MAX_MSG_LEN     10000

int main(int argc, char *argv[]) {
   mqd_t msgq_id;
   char msgcontent[MAX_MSG_LEN];
   int msgsz;
   unsigned int sender;
   struct mq_attr msgq_attr;

   msgq_id = mq_open(MSGQOBJ_NAME,
O_RDWR);
   if (msgq_id == (mqd_t)-1) {
      perror("In mq_open()");
      exit(1);
   }

   mq_getattr(msgq_id, &msgq_attr);

 printf("Queue \"%s\":\n\t- stores at most %ld
messages\n\t- large at most %ld bytes each\n\t-
currently holds %ld messages\n",
MSGQOBJ_NAME, msgq_attr.mq_maxmsg,
msgq_attr.mq_msgsize, msgq_attr.mq_curmsgs);

   /* getting a message */
   msgsz = mq_receive(msgq_id, msgcontent,
MAX_MSG_LEN, &sender);
   if (msgsz == -1) {
      perror("In mq_receive()");
      exit(1);
   }
   printf("Received message (%d bytes) from %d:
   %s\n", msgsz, sender, msgcontent);

   mq_close(msgq_id);
   return 0;
}
```

Notify Process That a Message Is Available (**REALTIME**)

#include <mqueue.h>
int mq_notify(mqd_t *mqdes*, const struct sigevent **notification*);

- If the argument *notification* is not NULL, this function shall register the calling process to be notified of message arrival at an empty message queue *mqdes*.
- If *notification* is NULL and the process is currently registered for notification by the specified message queue, the existing registration shall be removed.
- At any time, only one process may be registered for notification by a message queue.
- The example program:
  - Registers a notification request for the message queue named in its command-line argument.
  - Notification is performed by creating a thread.
  - The thread executes a function which reads one message from the queue and then terminates the process.

mq_notify Example


```
#include <pthread.h>
#include <mqueue.h>
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

static void                     /* Thread function */
tfunc(union sigval sv) {
    struct mq_attr attr;
    ssize_t nr;
    void *buf;
    mqd_t mqdes = *((mqd_t *) sv.sival_ptr);

    if (mq_getattr(mqdes, &attr) == -1) {
        perror("mq_getattr");
        exit(EXIT_FAILURE);
    }
    buf = malloc(attr.mq_msgsize);
    nr = mq_receive(mqdes, buf, attr.mq_msgsize,
NULL);
    if (nr == -1) {
        perror("mq_receive");
        exit(EXIT_FAILURE);
 }
 printf("Read %ld bytes from message
queue\n",(long) nr);
    free(buf);
    exit(EXIT_SUCCESS);         /* Terminate the process */
}

int main(int argc, char *argv[]) {
    mqd_t mqdes;
    struct sigevent not;

    assert(argc == 2);
    mqdes = mq_open(argv[1], O_RDONLY);
    if (mqdes == (mqd_t) -1) {
        perror("mq_open");
        exit(EXIT_FAILURE);
    }

    not.sigev_notify = SIGEV_THREAD;
    not.sigev_notify_function = tfunc;
    not.sigev_notify_attributes = NULL;
    not.sigev_value.sival_ptr = &mqdes;   /* Arg. to
thread*/
    if (mq_notify(mqdes, &not) == -1) {
        perror("mq_notify");
        exit(EXIT_FAILURE);
    }
    pause();    /* Process will be terminated by thread */
}
```

What Are Interthread Communications?

- Since threads reside in the address space of their process, that communication between threads seem not be difficult.
- The most important issue that has to be dealt with when peer threads require communication with each other is **synchronization**.
- Communication between threads is used to:
  - Share data
  - Send a message
- Threads cannot communicate with threads outside their process…
- …unless you are referring to primary threads of processes. In that case, you refer to them as two processes.
- In most cases, the cost of Interprocess Communication is higher than Interthread Communication.

The basic Interthread Communications

Karoly.Bosa@jku.at

Types of ITC	Description
Global data, variables, and data structures	Declared outside of the main function or have global scope. Any modifications to the data are instantly accessible to all peer threads.
Parameters	Parameters passed to threads during creation. The generic pointer can be converted any data type.
File handles	Files shared between threads. These threads share the same read-write pointer and offset of the file.

Example for the Usage of Global Variables

Karoly.Bosa@jku.at

```
// thread_tasks.h.

 1
 2  void *task1(void *X);
 3  void *task2(void *X);
 4  void *task3(void *X);
 5
```

```
// thread_tasks.cc.

 1  extern int Answer;
 2
 3  void *task1(void *X)
 4  {
 5     Answer = Answer * 32;
 6  }
 7
 8  void *task2(void *X)
 9  {
10     Answer = Answer / 2;
11  }
12
13  void *task3(void *X)
14  {
15     Answer = Answer + 5;
16  }
```

```
 1  using namespace std;
 2  #include <iostream>
 3  #include <pthread.h>
 4  #include "thread_tasks.h"
 5
 6  int Answer = 10;
 7
 8
 9  int main(int argc, char *argv[])
10  {
11
12     pthread_t ThreadA, ThreadB, ThreadC;
13
14     cout << "Answer = " << Answer << endl;
15
16     pthread_create(&ThreadA,NULL,task1,NULL);
17     pthread_create(&ThreadB,NULL,task2,NULL);
18     pthread_create(&ThreadC,NULL,task3,NULL);
19
20     pthread_join(ThreadA,NULL);
21     pthread_join(ThreadB,NULL);
22     pthread_join(ThreadC,NULL);
23
24     cout << "Answer = " << Answer << endl;
25
26     return(0);
27
28  }
```

- **Synchronization is missing!**
- It is not guaranteed that the correct answer (165).

Parameters for Interthread Communication

- The thread creation API supports thread parameters. The parameter is in the form of a void pointer:

```
int pthread_create(pthread_t *threadID,const pthread_attr_t *attr,
                   void *(*start_routine)(void*),
                   void *restrict parameter);
```

- The void pointer can be used to point to any kind of data type.

Example – Prg. 1.


```
1  using namespace std;
2  #include <queue>
3  #include <string>
4  #include <iostream>
5
6  extern queue<string> SourceText;
7  extern queue<string> FilteredText;
8
9  void *task1(void *X)
10 {
11    char Token = '?';
12
13    queue<string> *Input;
14
15
16    Input = static_cast<queue<string> *>(X);
17    string Str;
18    string FilteredString;
19    string::iterator NewEnd;
20
21    for(int Count = 0;Count < 16;Count++)
22    {
23       Str = Input->front();
24       Input->pop();
25       NewEnd = remove(Str.begin(),Str.end(),Token);
26       FilteredString.assign(Str.begin(),NewEnd);
27       SourceText.push(FilteredString);
28
29    }
30
31
32 }
33
34
35 void *task2(void *X)
36 {
37    char Token = '.';
38
39    string Str;
40    string FilteredString;
41    string::iterator NewEnd;
42
43    for(int Count = 0;Count < 16;Count++)
44    {
45       Str = SourceText.front();
46       SourceText.pop();
47       NewEnd = remove(Str.begin(),Str.end(),Token);
48       FilteredString.assign(Str.begin(),NewEnd);
49       FilteredText.push(FilteredString);
50
51
52    }
53
54 }
```

Example – Prg. 2.


```
 1  using namespace std;
 2  #include <iostream>
 3  #include <pthread.h>
 4  #include "thread_tasks.h"
 5  #include <queue>
 6  #include <fstream>
 7  #include <string>
 8
 9
10
11
12  queue<string> FilteredText;
13  queue<string> SourceText;
14
15  int main(int argc, char *argv[])
16  {
17
18     ifstream Infile;
19     queue<string> QText;
20     string Str;
21     int Size = 0;
22
23
24     pthread_t ThreadA, ThreadB;
25
26     Infile.open("book_text.txt");
27     for(int Count = 0;Count < 16;Count++)
28     {
29        getline(Infile,Str);
30        QText.push(Str);
31
32     }
33
34     pthread_create(&ThreadA,NULL,task1,&QText);
35     pthread_join(ThreadA,NULL);
36
37     pthread_create(&ThreadB,NULL,task2,NULL);
38     pthread_join(ThreadB,NULL);
39
40     Size = FilteredText.size();
41
42     for(int Count = 0;Count < Size ;Count++)
43     {
44        cout << FilteredText.front() << endl;
45        FilteredText.pop();
46
47     }
48
49     Infile.close();
```

Compile and Link Instructions

- c++ -o prg_arg prg_arg.cc thread_tasks.cc -lpthread

An Addition: Signals

- A **signal** is a limited form of inter-process communication used in Unix, Unix-like, and other POSIX-compliant operating systems.
- Essentially it is an asynchronous notification sent to a process in order to notify it of an event that occurred.
- When a signal is sent to a process, the operating system interrupts the process's normal flow of execution.
- Execution can be interrupted during any non-atomic instruction.
- If the process has previously registered a **signal handler**, that routine is executed. Otherwise the default signal handler is executed.

Some POSIX.1 Signals

SIGABRT	Abnormal termination signal caused by the abort() function. A portable program should void catching SIGABRT.
SIGALRM	The timer set by the alarm() function has timed-out.
SIGINT	Interrupt special character typed on controlling keyboard.
SIGKILL	Termination signal. This signal cannot be caught or ignored.
SIGPIPE	Write to a pipe with no readers.
SIGQUIT	Quit special character typed on controlling keyboard.
SIGTERM	Termination signal.
SIGUSR1	Application-defined signal 1.
SIGUSR2	Application-defined signal 2.

Some POSIX.1 Signals 2 – Job Controls

Karoly.Bosa@jku.at

SIGCHLD	Child process terminated or stopped. By default, this signal is ignored.
SIGCONT	Continue the process if it is currently stopped; otherwise, ignore the signal.
SIGSTOP	Stop signal. This signal cannot be caught or ignored.
SIGTSTP	Stop special character typed on the controlling keyboard.
SIGTTIN	Read from the controlling terminal attempted by a member of a background process group.
SIGTTOU	Write to controlling terminal attempted by a member of a background process group

Example for Signals

- Typing Ctrl-C sends an INT signal (SIGINT); by default, this causes the process to terminate.
- Typing Ctrl-Z sends a TSTP signal (SIGTSTP); by default, this causes the process to suspend execution.
- Typing Ctrl-\ sends a QUIT signal (SIGQUIT); by default, this causes the process to terminate and dump core.
- The kill system call will send the specified signal to the process, if permissions allow.
- The kernel can generate a signal to notify the process of an event. For example, SIGPIPE will be generated when a process writes to a pipe which has been closed by the reader.

Sending Signals and Signal Handlers


```
#include <signal.h>

int kill(pid_t pid, int sig);
void signal(int sig_type, void (*sig_handler)(int signal_type));
```

- Processes send signals with the *kill()* function.
- A process may associate an handler function to a specific signal type with the *signal()* function.
- A process can signal another process only if they belong to the same user.
- Processes run by a superuser can signal every process.

How the Signals Are Treated

- For some of these signals, the OS inhibits custom handlers: SIGSTOP and SIGKILL will always make the process respectively stop and die.

- Two standard handlers have been provided with the standard C library:
  - SIG_DFL causing process termination, and
  - SIG_IGN causing the signal to get ignored.

- When an handler fires for a signal, it won't be interrupted by other handlers if more signals arrive.

- Signals are not queued (somehow, "best effort"). In order to notify signals, the OS holds a bitmask for every process.

Block and Unblock Signal Delivery I.


```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

- The *sigprocmask()* call can be used to block and unblock delivery of signals (in a single-threaded process).
- If the argument *set* is not a null pointer, it points to a set of signals to be used to change the currently blocked set.
- The argument *how* indicates the way in which the set is changed:
  - **SIG_BLOCK**: The resulting set will be the union of the current set and the signal set pointed to by *set*.
  - **SIG_SETMASK** The resulting set will be the signal set pointed to by *set*.
  - **SIG_UNBLOCK** The resulting set will be the intersection of the current set and the complement of the signal set pointed to by *set*.
- If the argument *oset* is not a null pointer, the previous mask is stored in the location pointed to by *oset*.

Block and Unblock Signal Delivery II.


```
#include <signal.h>

int sigprocmask(int how, const sigset_t *set, sigset_t *oset);
int pthread_sigmask(int how, const sigset_t *set, sigset_t *oset);
```

- The use of the *sigprocmask()* function is unspecified in a multi-threaded process.
- The *pthread_sigmask()* function is used to examine or change (or both) the calling thread's signal mask, regardless of the number of threads in the process
- **Question:** Which thread receive the signal address to the process?

Signal Example


```
#include <signal.h>
#include <pthread.h>
/*…*/
void  signal_handler(int sig);

int main(int argc, char *argv[]) {
sigset_t set;

/* create signal processing thread */
pthread_create(&thread_id, NULL, signal_processor,
                NULL);

sigemptyset(&set);
sigaddset(&set, SIGHUP);
sigaddset(&set, SIGINT);
sigaddset(&set, SIGUSR1);
sigaddset(&set, SIGUSR2);
sigaddset(&set, SIGALRM);

/* block out these signals */
pthread_sigmask(SIG_BLOCK, &set, NULL);

/* ... */
return 0;
}
```

```
// implementation of the signal processing thread
void *signal_processor(void *arg) {
  sigset_t set;

  signal(SIGHUP, &signal_handler);
  signal(SIGINT, &signal_handler);
  signal(SIGUSR1, &signal_handler);

  sigfillset(&set);
  for (;;) {
    sigwait(&set, &sig);
    switch(sig) {
        case SIGTERM: exit(1);

        default: printf("caught signal %d\n", sig);
    }
  }
   ...
}
void signal_handler(int sig) {
   /*...*/
   ;
}
```