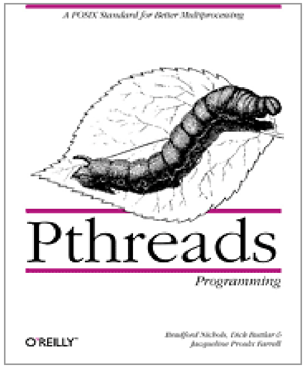
SYSTEM PROGRAMMING

Part 1: C = Language + Environment

Part 2: Linux Programming Interface

Course Material

- Slides & code on Toledo
  - Intro to multi-tasking
    - 1 doc on parallel computing
  - Multi-processing
    - 1 doc on process/fork/exec/wait/zombie
    - 1 doc on signals
    - 2 docs on pipes&fifos
    - 1 doc on design models
    - Beej's Guide to Unix IPC
  - Multi-Threading
    - 3 docs on POSIX thread programming



- Slides contain online references!
  - "Programming in C – Unix system calls and subroutines using C"
    - www.cs.cf.ac.uk/Dave/C/

- A recent book (hardcopy)
  - "The Linux Programming Interface – A Linux and Unix System Programming Handbook", M. Kerrisk

LECTURE S1

Interfacing With Linux

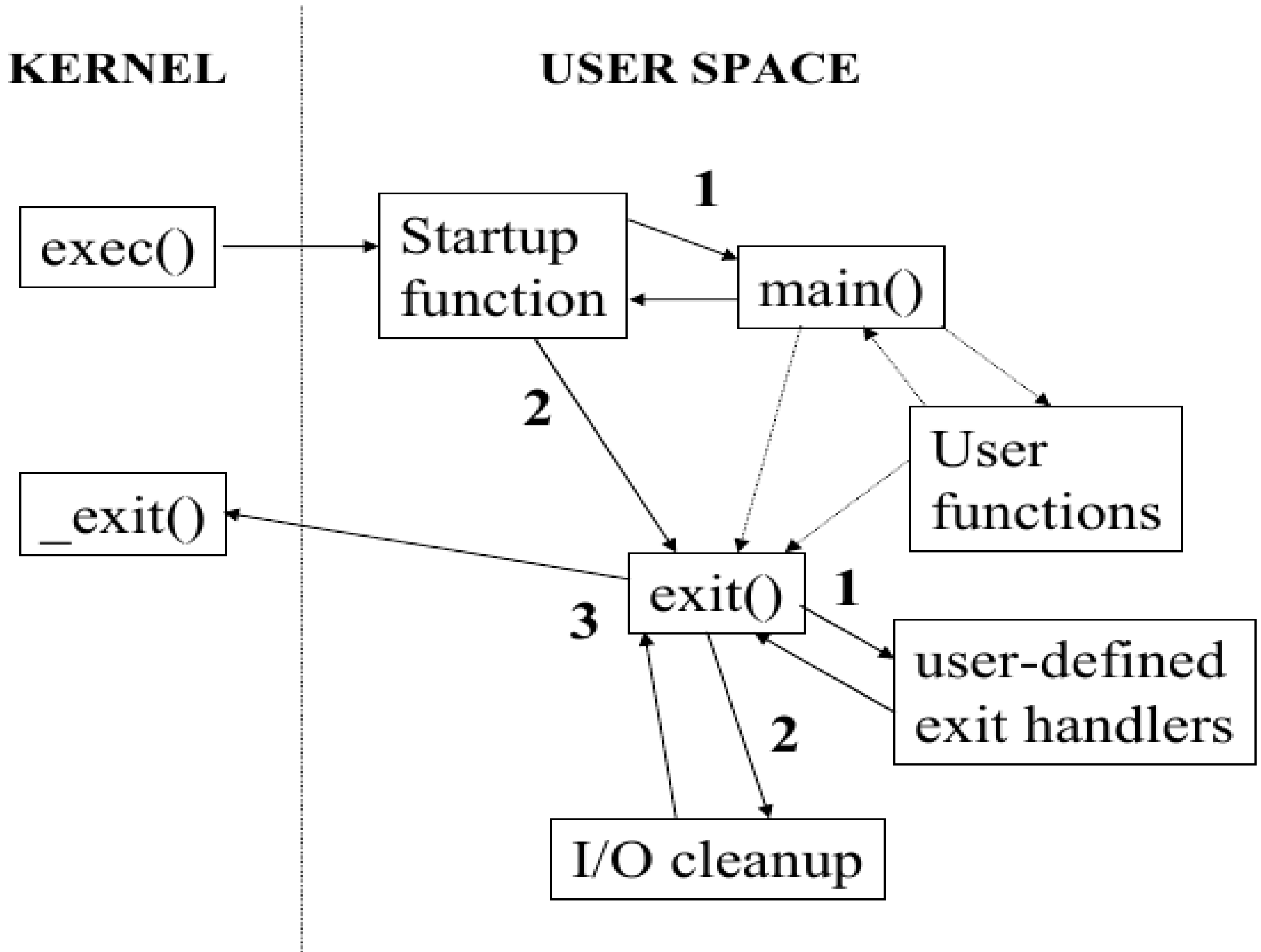
System Calls

- Kernel Mode
  - Executing code has unrestricted access to the underlying hardware
  - Executing code has access to any memory address
  - Kernel mode is reserved for the most trusted tasks of the OS
  - Crashes in kernel mode will most likely halt the entire PC

- User Mode
  - Executing code has no ability to directly access hardware or reference memory.
  - Code running in user mode must use system calls to access hardware or memory
  - Crashes in user mode are recoverable
  - Most code (applications) run in user mode

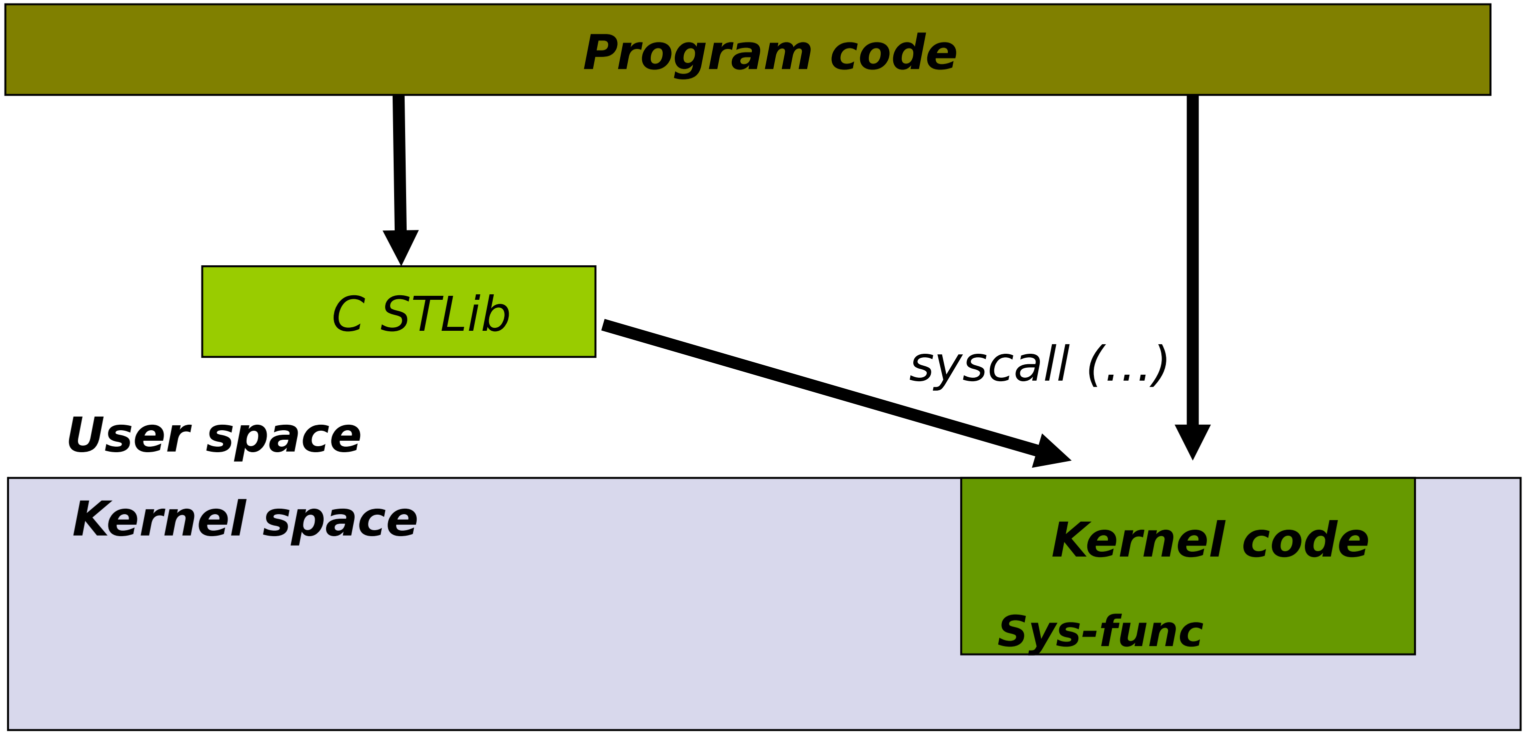
System Calls

Example: execution of a program



System Calls

- System calls allow user programs to call OS services



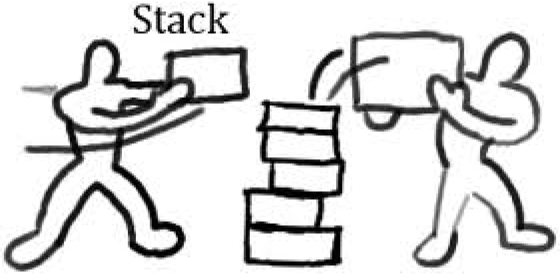
System Calls

- Library code is executed as part of the program code and runs in **user mode**
  - Static library code is compiled as part of the final program code
  - Dynamic library code is loaded at run-time
- System call code is executed as part of the kernel itself and not of the program
  - Runs in **kernel mode**
  - System call requires a switch from user mode to kernel mode
    - This is a costly operation!

System Calls

- System calls in Linux
  - Overview of Linux system calls
    - https://github.com/torvalds/linux/blob/v3.17/arch/x86/syscalls/syscall_64.tbl
    - <sys/syscall.h>
    - Man 2 syscalls
    - Man 2 intro

[DEMO]



Calling System Calls

- Method 1: syscall
  - long int syscall (syscall_number, args);
    - Man syscall
    - See also
      - sys/syscall.h
      - /usr/include/asm-generic/unistd.h

```
#include <unistd.h>
#include <sys/syscall.h>
int rc;
rc = syscall(SYS_chmod, "/etc/passwd", 0444);
if (rc == -1)
  fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

Calling System Calls

- Method 2: wrapper syscall
  - C library has wrappers for most system calls that are easier to use
  - Example

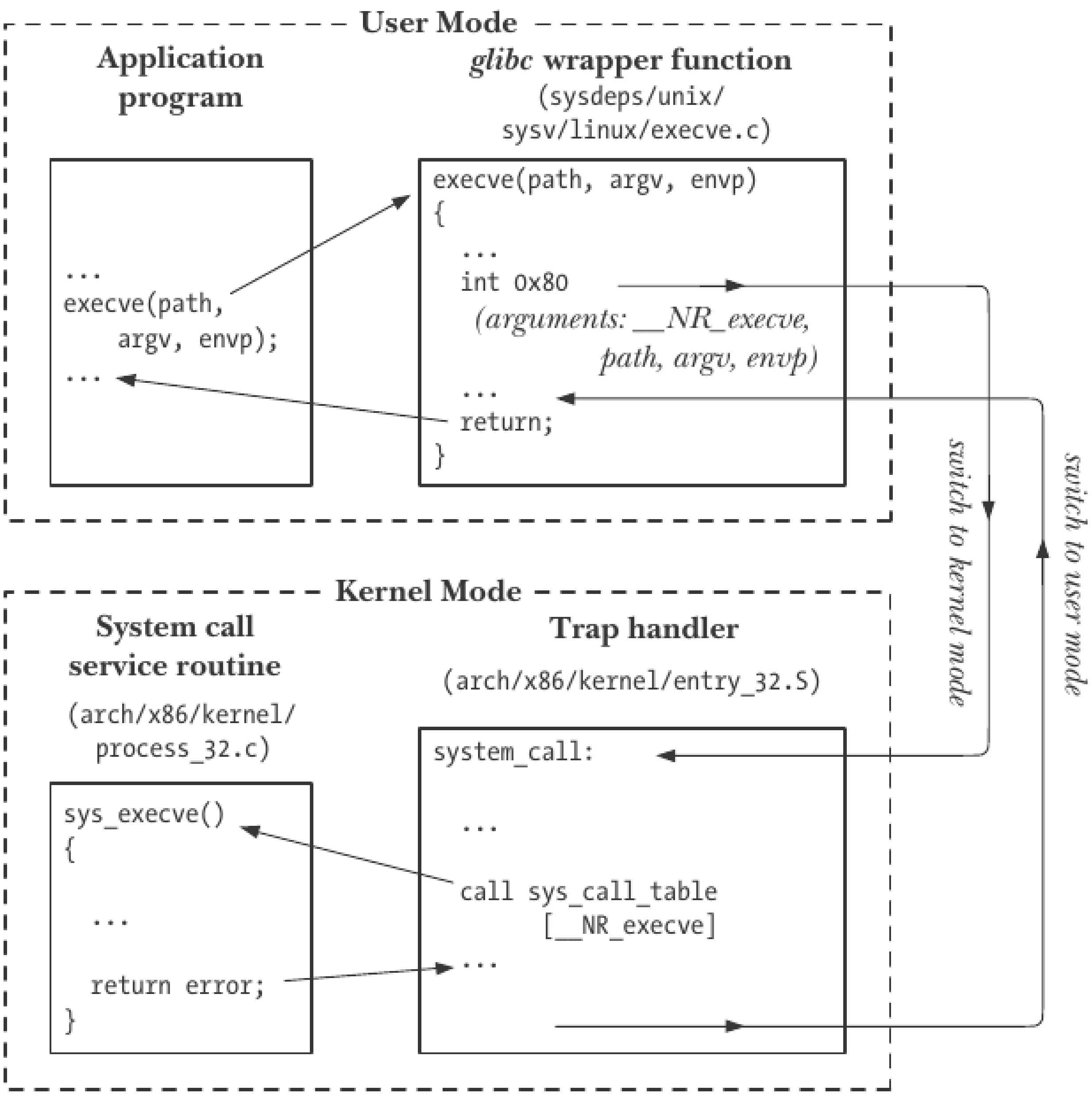
```
#include <sys/types.h>
#include <sys/stat.h>

int rc;
rc = chmod("/etc/passwd", “r--r--r--”);
if (rc == -1)
   fprintf(stderr, "chmod failed, errno = %d\n", errno);
```

Calling System Calls

- Method 3: indirect call using C lib high-level function calls
  - High-level function calls that handle all system call(s) code behind the scene

```
#include <stdio.h>

int i;
double d;
scanf("%d %g", &i, &d);
   ==> reads input bytes from keyboard buffer and
   converts these bytes into, respectively, an integer
   and a double
```

System Call Execution



©The Linux Programming Interface, M. Kerrisk

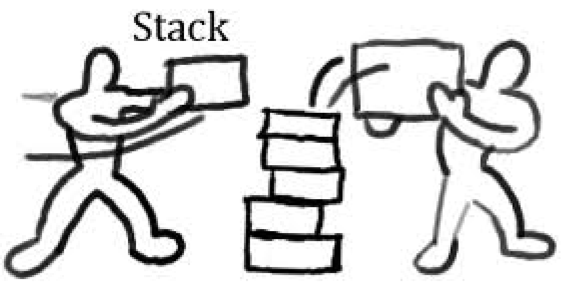
System Calls and Errors

- System calls can fail due to some error
  - Typically, a return value of 0 indicates success and -1 a failure
    - But there are exceptions … ==> always check the man pages!
  - If an error occurred, the global variable 'errno' will be set to the error code
    - Check the man pages for all error codes and there meaning!
  - Example: ‘open(…)’
    - Errors
      - EACCES
        - The requested access to the file is not allowed, or search permission is denied for one of the directories in the path prefix of pathname, or the file did not exist yet and write access to the parent directory is not allowed. (See also path_resolution(7).)
      - And many more ….
  - This is also true for malloc, read, write, select, fseek, …
    - Check the man pages!

System Calls and Errors

- #include <errno.h>
  - To get access to the global variable ‘errno’ …
  - Errno contains the ID of an implementation-defined error message
  - char * strerror(int ID)
    - Returns pointer to implementation-defined error message corresponding to ID
  - void perror(char *s)
    - Prints s and an implementation-defined error message corresponding to errno

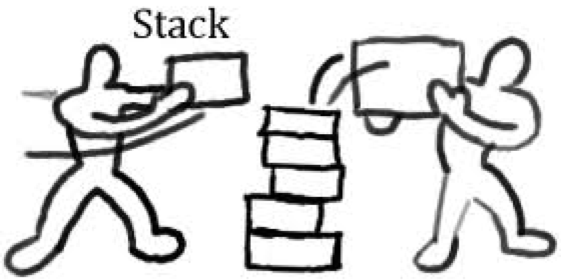
[DEMO]



Strace Utility

- Diagnostic and debugging tool to record all system calls of an application
  - Strace: system calls
  - Ltrace: library calls

[DEMO]

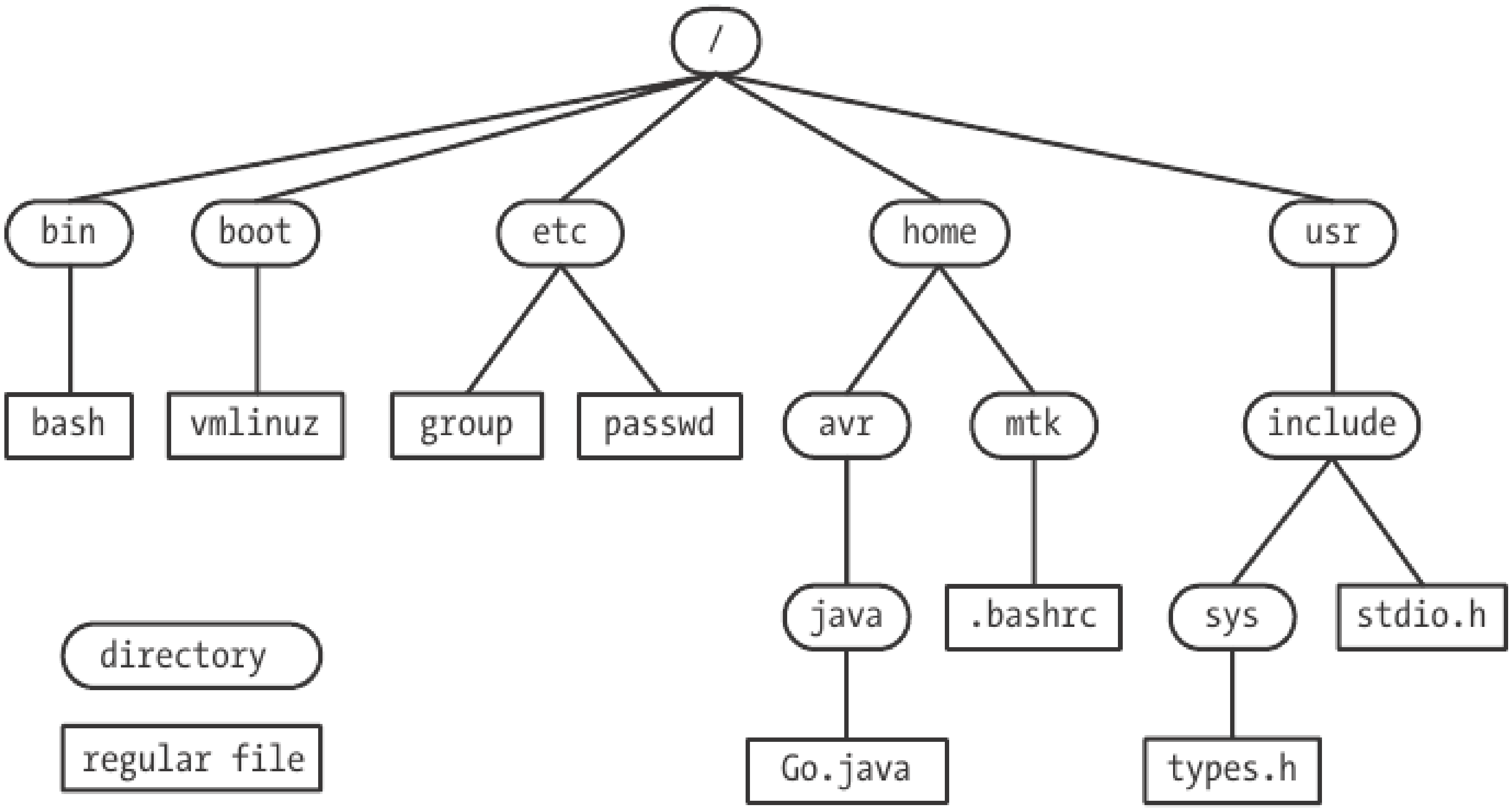


LECTURE S2

FILE I/O

Linux File Basics

- Kernel maintains 1 hierarchical file/directory structure



[Shell cmd: cd, pwd, ...]

©The Linux Programming Interface, M. Kerrisk

Linux File Basics

- Several 'file' types exist
  - Data files are regular files
  - Directories are files containing a list of 'links' between file/directory names and references to the appropriate files/directories
    - . = link to this directory
    - .. = link to the parent directory
    - Hard link = normal link
    - Soft link or symbolic link = a file name and reference to a special file containing the name of the file/directory for which the symbolic link defines an alternative name
    - Pathname describes the location of the file within the directory structure
      - Absolute pathname: location relative to root
      - Relative pathname: location relative to current working dir
  - Devices (mouse, keyboard, serial ports, ...), pipes, sockets, … are also 'files'

[Shell cmd: ls -a, ln -s, ...]

Linux File Basics

- Users / Groups / Superuser
  - Every user of the system has a login name, userID, home dir, login shell, etc.
  - Users are organized in groups to easily control access to files and system resources
    - Group name, group id, list of users, …
  - Superuser (userID 0 – login = root) bypasses all permission checks and can access all resources
    - System administration

[Shell cmd: chown, groups, id, su, ...]

Linux File Basics

- File permissions in Linux
  - 3 types of access control
    - Read
    - Write
    - Execute
  - 3 types of users
    - Owner
    - Group
    - World (other)

> ls –l main.c

-rw-rw-r--

d = file type: [d]irectory, [l]ink, [-]ordinary, …

[Shell cmd: chmod]

Linux File Basics

- File permissions in Linux
  - Textual or octal representation

Example:

Owner has all rights, group and world only read permission:

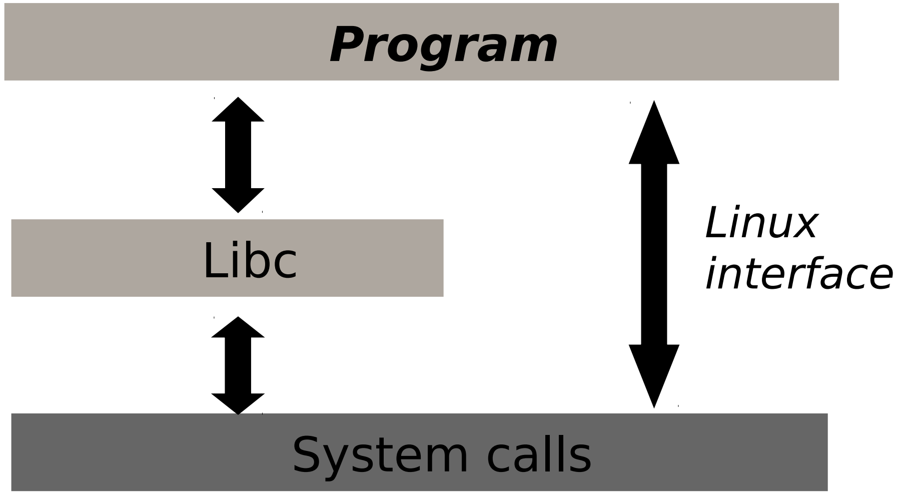
rwxr--r--

0744

0	- - -
1	- - x
2	- w -
3	- w x
4	r - -
5	r - x
6	r w -
7	r w x

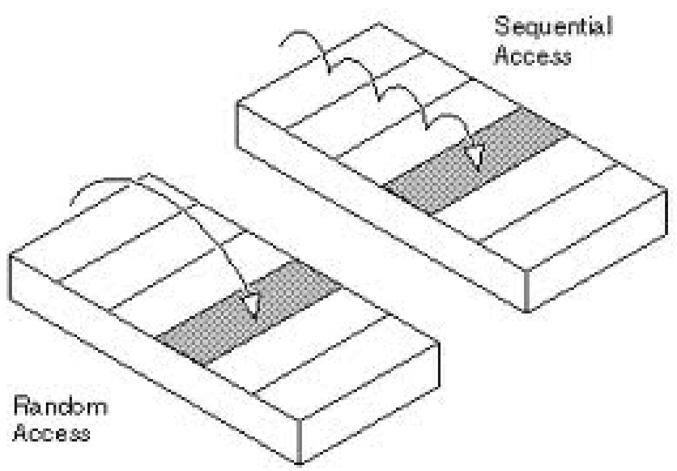
File I/O

- I/O not part of C language
- Stdin
  - Standard input
  - e.g. keyboard
- Stdout
  - Standard output
  - e.g. screen
- Stderr
  - Standard error output
  - e.g. to file or screen

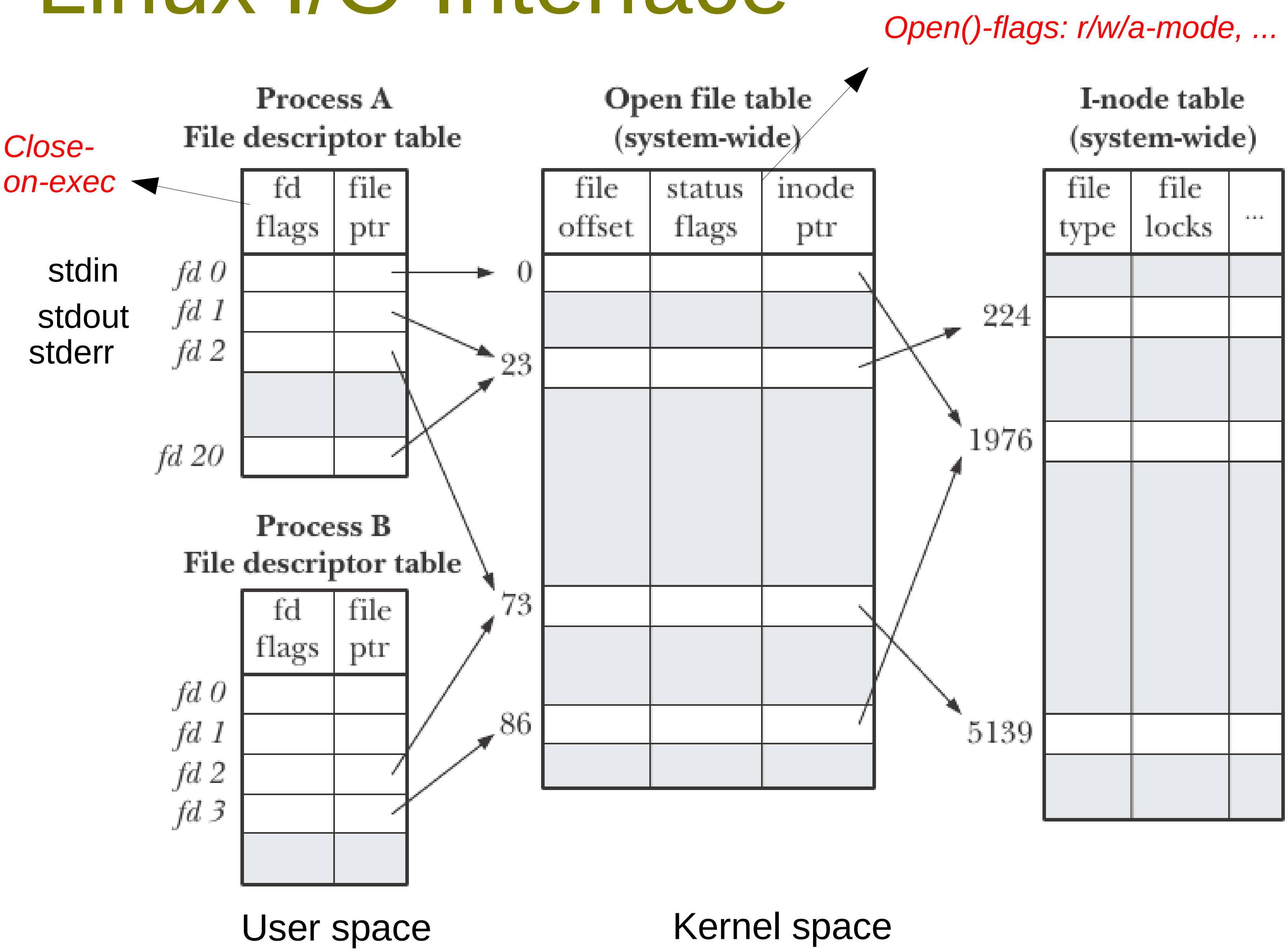


Sequential vs. Random Access

- Files keep track of a current “file offset”
  - Read/write always happens at “file offset”
  - After read/write operation, file offset is moved up 1 record
  - File open in r/w mode: file offset = start of file
  - File open in a mode: file offset = end of file
- Sequential access files
  - File offset can only move forward, record by record
  - Rewind allows to return to the beginning of the file and start over
  - Example: A/V tape
- Random access (direct access) files
  - File offset can be set to an arbitrary position
  - Example: HDD



Linux I/O Interface



[Shell cmd: ls -i]

©The Linux Programming Interface, M. Kerrisk

Linux I/O Interface

- Int open(char *pathname, int flags, mode_t mode)
  - Pathname: absolute / relative path + filename
  - Flags
    - O_RDONLY | O_WRONLY | O_RDWR
    - Optional: O_APPEND | O_CREAT | O_NONBLOCK | ...
  - Mode: is optional parameter used with O_CREAT to define file access permissions
  - Returns new fd or -1 on error

A call to open() creates a new open file description, an entry in the system-wide table of open files!

Read man pages for details!

[Shell cmd: lsof = list all open files]

Linux I/O Interface

- Int close(int fd)
  - Associated file record locks are freed; if fd is the last file descriptor to the file, recources are freed
  - Returns 0 on success; on error -1
- Size_t read(int fd, void *buf, size_t size)
  - Reads at the current file offset
  - After reading, the file offset is updated
  - Returns #bytes read; on error -1

Read man pages for details!

Linux I/O Interface

- Size_t write(int fd, void *buf, size_t size);
  - Writes at the current file offset
  - After writing the file offset is updated
  - Returns #bytes written; on error -1

- Off_t lseek(fd, off_t offset, int startpos)
  - Random access file
    - Set file position to “startpos+offset”
    - startpos =
      - 0 = SEEK_SET: start from beginning of file
      - 1= SEEK_CUR: start from current position
      - 2 = SEEK_END: start from end of file (offset <=0)

Read man pages for details!

Linux I/O Interface

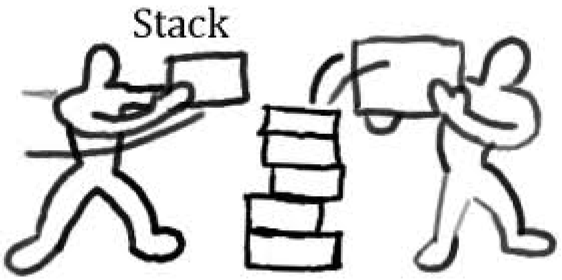
- Example: sequential access

```
int filecopy(char *inf, char *outf)
{
  char buffer[512];
  int inhandle,outhandle,bytes;

  inhandle = open(inf,O_RDONLY);
  outhandle = open(outf,O_CREAT|O_WRONLY);
  if ((inhandle==-1) || (outhandle==-1)) return –1;

  while((bytes=read(inhandle,buffer,512))>0)
      write(outhandle,buffer,bytes);

  close(inhandle); close(outhandle);
  return 0;
}
```

Linux I/O Interface

- Study the code ...

[Toledo: StackS2A]



Linux I/O Interface

- Other useful syscalls
  - Fd duplication to share the file offset – opening the same file twice doesn't!

    ```
    int dup2(int oldfd, int newfd);
    ```

  - File control operations: set blocking/non-blocking, get/set file descriptor/status flags, fd duplication, signal control, ...

    ```
    int fcntl(int fd, int cmd, ...);
    ```

Read man pages for details!

Linux I/O Interface

- Other useful syscalls
  - Read/write from a specified offset without changing the 'file offset'
    - Application: avoiding race conditions in multithreaded apps

```
ssize_t pread(int fd, void *buf, size_t count, off_t offset);

ssize_t pwrite(int fd, const void *buf, size_t count,
                                                      off_t offset);
```

Read man pages for details!

Linux I/O Interface

- Read/write syscall performance
  - Using read()/write() with small vs. large byte blocks

Copy 100MB file

BUF_SIZE	Time (seconds)			
	Elapsed	Total CPU	User CPU	System CPU
1	107.43	107.32	8.20	99.12
2	54.16	53.89	4.13	49.76
4	31.72	30.96	2.30	28.66
8	15.59	14.34	1.08	13.26
16	7.50	7.14	0.51	6.63
32	3.76	3.68	0.26	3.41
64	2.19	2.04	0.13	1.91
128	2.16	1.59	0.11	1.48
256	2.06	1.75	0.10	1.65
512	2.06	1.03	0.05	0.98
1024	2.05	0.65	0.02	0.63
4096	2.05	0.38	0.01	0.38
16384	2.05	0.34	0.00	0.33
65536	2.06	0.32	0.00	0.32

Libc Interface

- Library calls: stdio.h
  - Accessing files through FILE *
  - Default FILE *
    - stdin
    - stdout
    - stderr

```
typedef struct _iobuf
{
    char*   _ptr;
    int  _cnt;
    char*   _base;
    int  _flag;
    int  _file;
    int  _charbuf;
    int  _bufsiz;
    char*   _tmpfname;
} FILE;
```

Libc Interface

- Library calls: stdio.h
  - FILE *fopen(char *fname, char *mode)
    - Opens the file <fname> in <mode>; returns NULL if this is not possible
    - Mode
      - r : read-only – file must exists
      - w : write-only – file is created if it doesn’t exist
      - a : append – file is created if it doesn’t exist
      - r+ | w+ | a+ : reading and writing, but …
  - fclose(FILE *)

Libc Interface

- Library calls: stdio.h
  - Text file: formatted I/O
    - int fprintf(file, “Date = %u/%u\n”, day, month)
      - Returns # chars printed
      - Remark: printf(…) == fprintf(stdout, …)
    - int fscanf(file, “%f%d”, &x, &i)
      - Returns # matched and assigned input items or EOF
      - Remark: scanf(…) == fscanf(stdin, …)

Libc Interface

- Library calls: stdio.h
  - Text file: single char I/O
    - int fgetc(FILE *)
    - int fputc(int, FILE *)
    - int ungetc(int, FILE *)
      - putting char back in input stream

>> return EOF if not succesful

Libc Interface

- Library calls: stdio.h
  - Text file: line I/O
    - char * fgets(char *, int size , FILE *)
      >> return NULL if not succesful
    - int fputs(char *, FILE *)
      >> return EOF if not succesful
    - UNSAFE: gets(str) and puts(str)

Libc Interface

- Library calls: stdio.h
  - Binary I/O (unformatted)
    - size_t fread(void *buf, size_t elem_size, size_t num_elem, FILE *)
      - Returns # elements read (which might be zero)
    - size_t fwrite(void *buf, size_t elem_size, size_t num_elem, FILE *)
      - Returns # elements written

Libc Interface

- Library calls: stdio.h
  - For each file, flags are maintained concerning file errors and EOF status
    - int ferror(FILE *)
      - Returns non-zero if an error occured on fp
    - int feof(FILE *)
      - Returns non-zero if EOF occured on fp
  - Once the flags are set, they stay that way
    - void clearerr(file *)
      - clear eof and error flags for the file

Libc Interface

- Example: sequential acces

```
int main( void )
{
   int c;  /* don't use unsigned char */
   FILE *ifp, *ofp;

   ifp = fopen( “in.txt”, “r” );
   ofp = fopen( “out.txt”, “w” );
   if ( (ifp == NULL) || (ofp == NULL) )
     return -1;

    while ( ( c = fgetc( ifp ) ) != EOF )
     fputc( toupper( c ), ofp );

   fclose( ifp ); fclose( ofp );
   return 0;
}
```

Libc Interface

- Library calls: stdio.h
  - void rewind(FILE *)
    - Reset file position to beginning of file
  - long ftell(FILE *)
    - Returns the current file position as #chars since start of file
      - Only limited to files with size no more than ‘long’ ➔ fgetpos()!
    - Return -1 on error

Libc Interface

- Library calls: stdio.h
  - Int fseek(FILE *, long offset, int startpos)
    - Set file position to “startpos+offset”
    - startpos =
      - 0 = SEEK_SET: start from beginning of file
      - 1= SEEK_CUR: start from current position
      - 2 = SEEK_END: start from end of file (offset <=0)

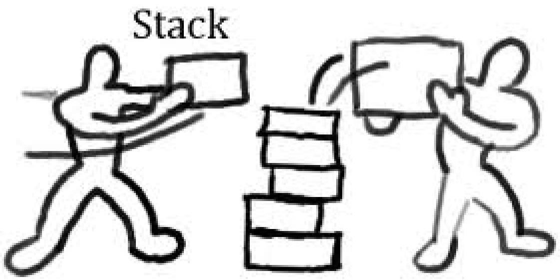
Libc Interface

- Library calls: stdio.h
  - Int fgetpos(FILE *, fpos_t *)
  - Int fsetpos(FILE *, fpos_t *)
    - Stores/Sets the current file position
      - File position is stored as a ‘magic number’!
      - No limitations on file size
    - Returns 0 on success, non-zero on error
  - Get file descriptor from file pointer
    - Int fileno(FILE * fp)
  - Open file pointer from file descriptor
    - FILE * fdopen(int fd, char * mode)

Libc Interface

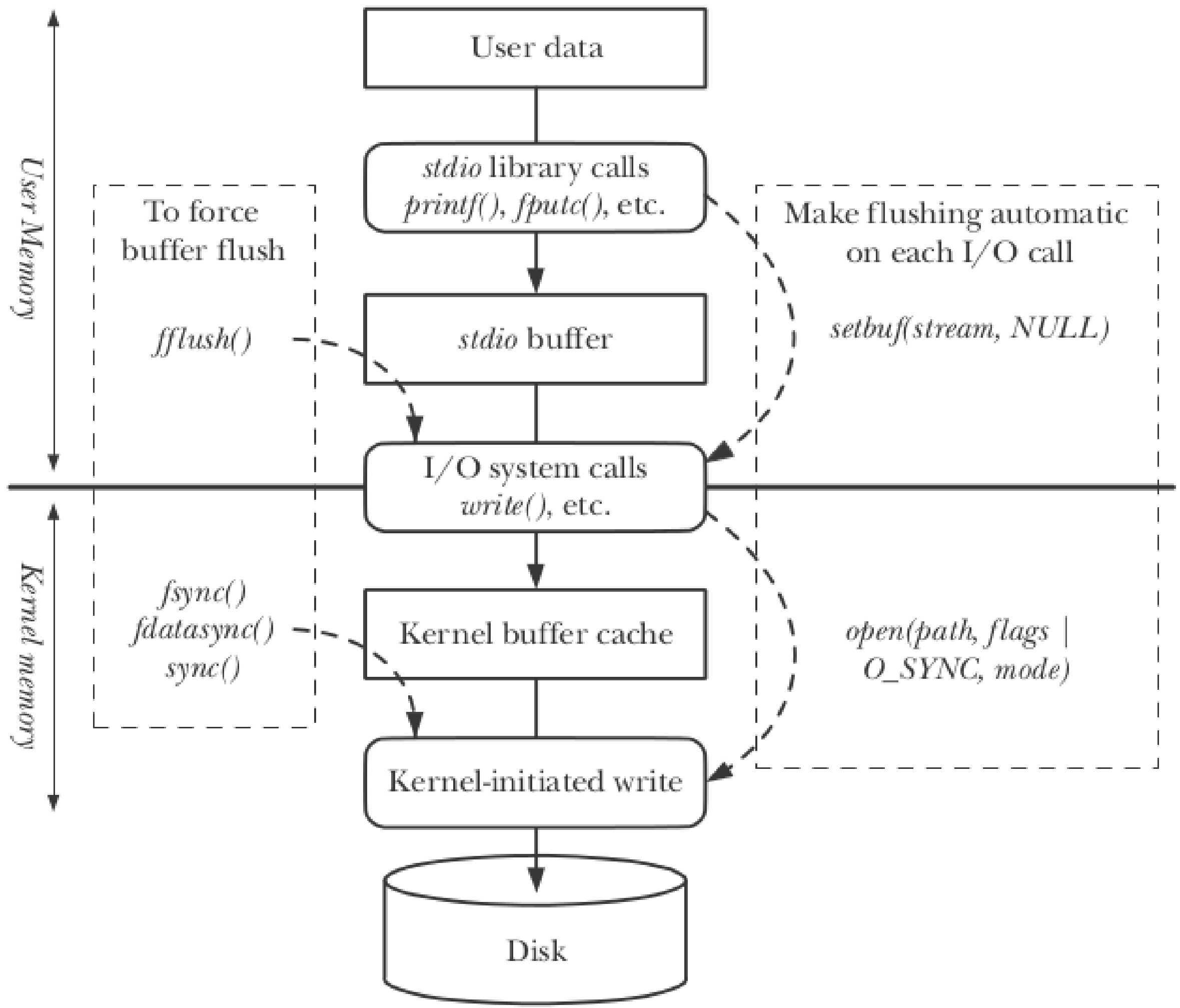
- Study the code …

[Toledo: StackS2B]



Advanced I/O

- User and kernel buffering



Reduce #syscalls

Reduce #I/O calls to device

©The Linux Programming Interface, M. Kerrisk

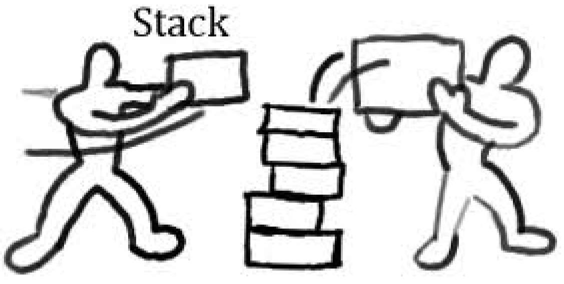
Advanced I/O

- Controlling Libc buffering
  - Unbuffered (e.g. stderr)
  - Block buffered (default mode)
  - Line buffered ➔ newline! (e.g. Stdin)

- Int setvbuf(FILE *, char *buf, int mode, size_t size)
  - Mode
    - _IONBF (unbuffered)
    - _IOLBF (line buffered)
    - _IOFBF (fully buffered)

- Int fflush(FILE *)
  - Write data from buffer to file

[demo]



Advanced I/O

- Controlling kernel buffering
  - Synchronized I/O
    - I/O operation after which the [data/metadata] on disk is really updated
    - Syscalls to control this
      - int fsync(int fd); ==> data+metadata
      - int fdatasync(int fd); ==> data only
      - void sync(void); ==> flush all kernel buffers
    - Use flags in file open call
      - O_SYNC, O_DSYNC, O_RSYNC
      - See man pages for details!

Advanced I/O

- Performance
  - Impact of O_SYNC flag

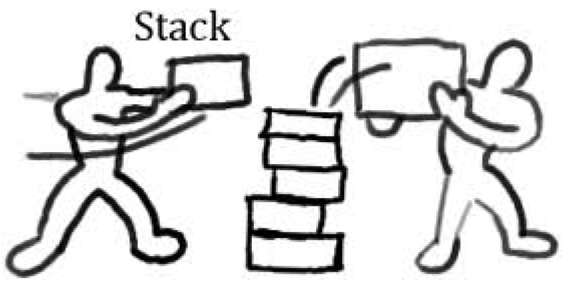
Write 1MB

BUF_SIZE	Time required (seconds)			
	Without O_SYNC		With O_SYNC	
	Elapsed	Total CPU	Elapsed	Total CPU
1	0.73	0.73	1030	98.8
16	0.05	0.05	65.0	0.40
256	0.02	0.02	4.07	0.03
4096	0.01	0.01	0.34	0.03

Advanced I/O

- Blocking / non-blocking
  - Blocking I/O: if an I/O syscall (open, read, …) can't be completed immediately, the syscall waits until it can be completed, hence, blocks the calling program
    - E.g.: a file is locked by another program, stdin, sockets, pipes (see later)
  - Non-blocking I/O syscall always returns immediately but the operation might not or only partially executed
  - How to control this?
    - Syscall
      - int fcntl(int fd, int cmd, ...);
    - Use O_NONBLOCK flag in file open call
      - See man pages for details!

[DEMO]

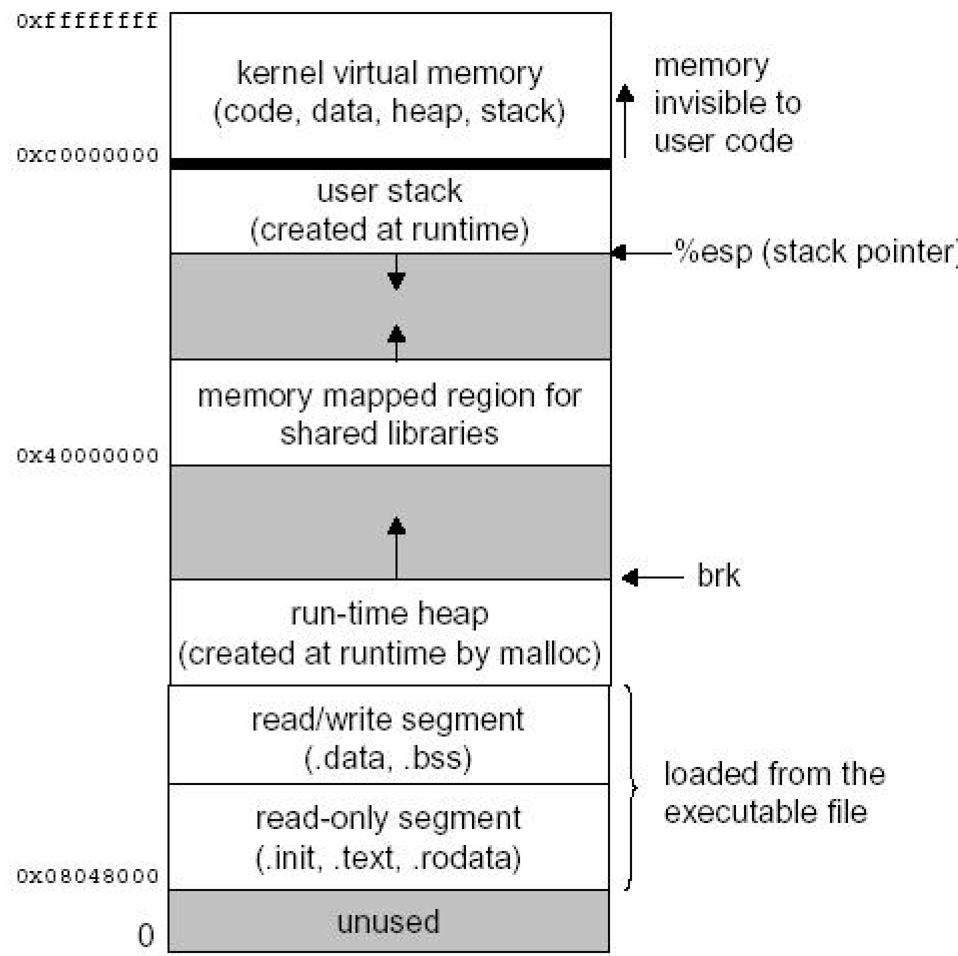


LECTURE M1

Programming With Processes

Process

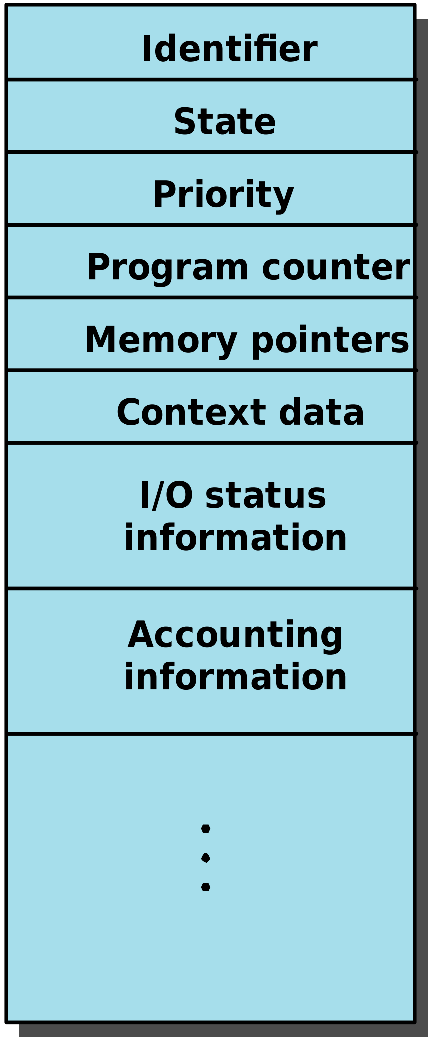
- Process is an OS concept
  - OS creates a process environment in which instances of executable code can run



Process

- A process is identified by a process ID (=PID)
  - Process can obtain its PID with:
    - *pid_t getpid(void)*
- Every process has its own (copy of the) code, PC, SP, registers, data, I/O handlers, …
    - A process runs 'independent' of other processes
- For every process, the OS needs to keep track of process information
  - OS implements a list of process control blocks (PCB)

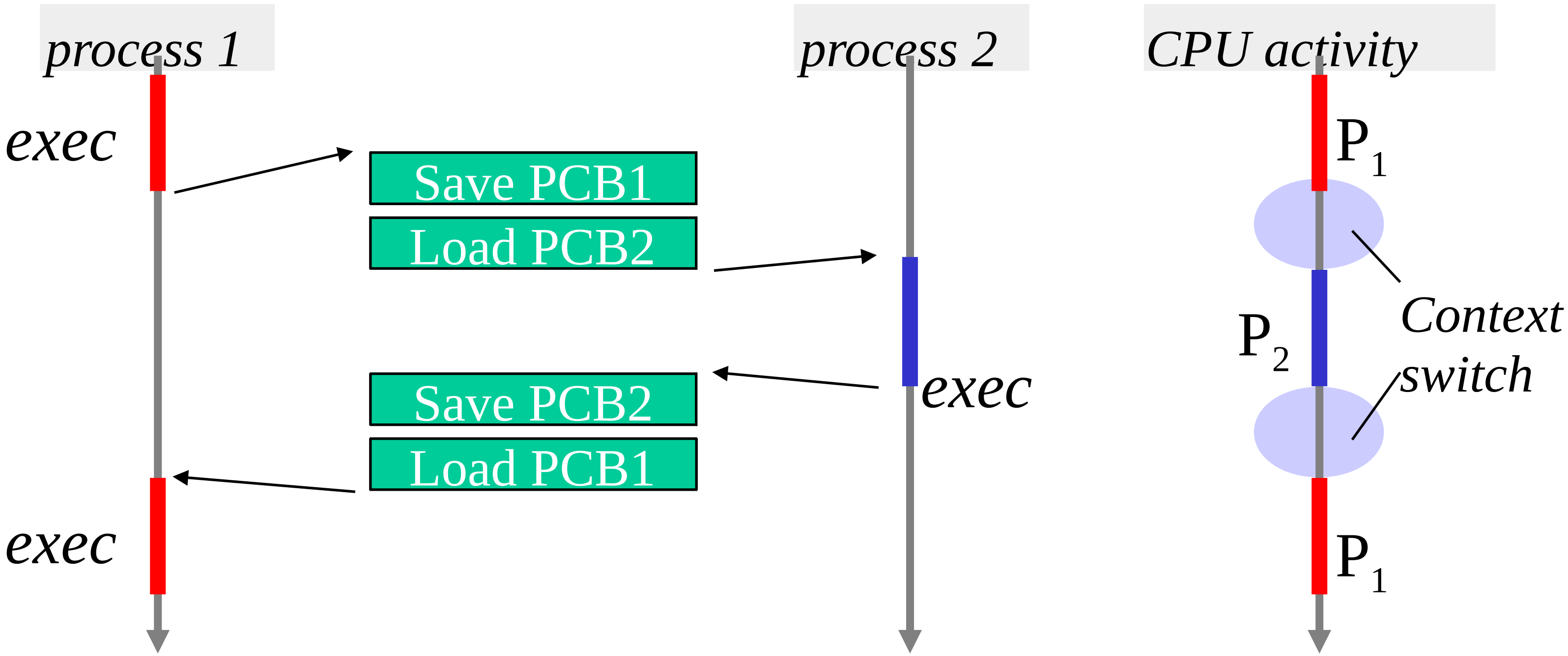
PCB



Details at: www.tldp.org/LDP/tlk/kernel/processes.html

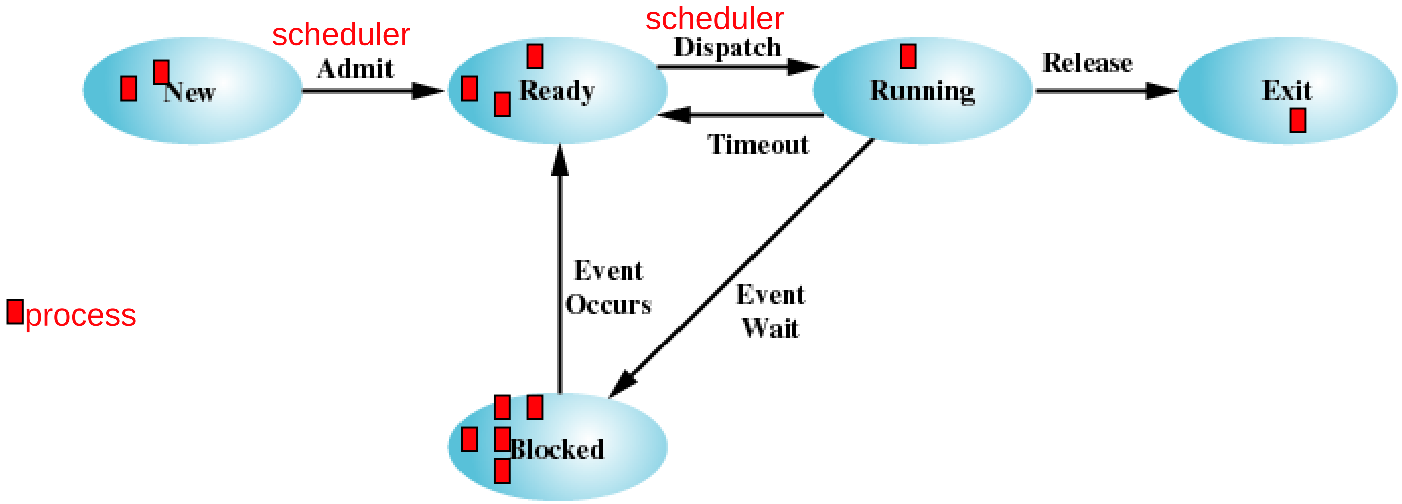
Context Switch

- In general, #processes that wish to run is larger than #CPUs
  - How does the OS deal with this?
    - Processes are put in a ready queue (waiting for CPU)
    - OS implements a process scheduler (dispatcher) that assigns every process to a CPU for a short time quantum
    - OS scheduler implements a **context switch**



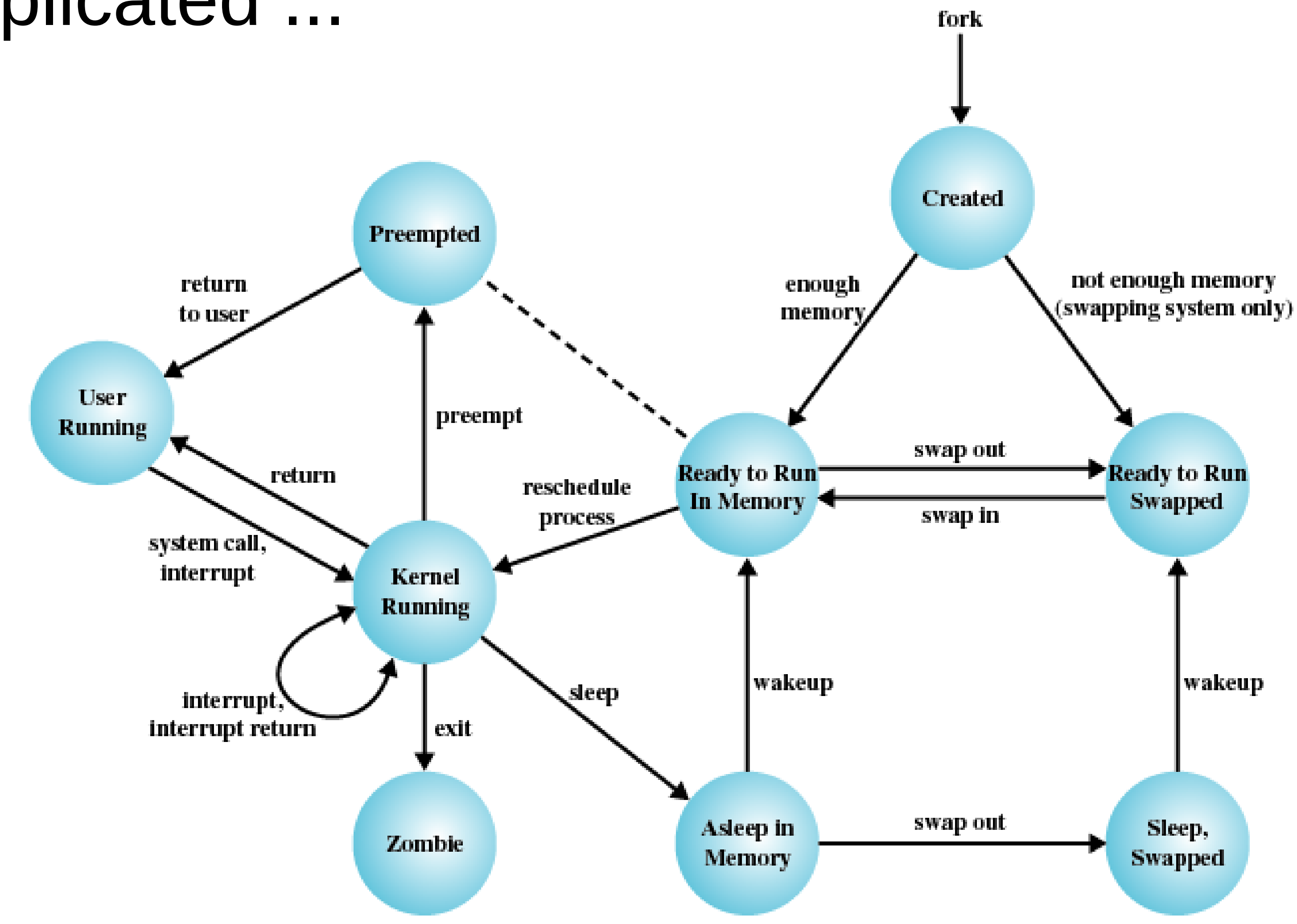
Process States

- Process can be in several process states
  - Running: has the CPU and runs
  - Ready: waiting on CPU to run
  - Blocked: waiting on some event (e.g. user input)
- OS keeps track of a process state model
  - Example: 5-state model



Process States

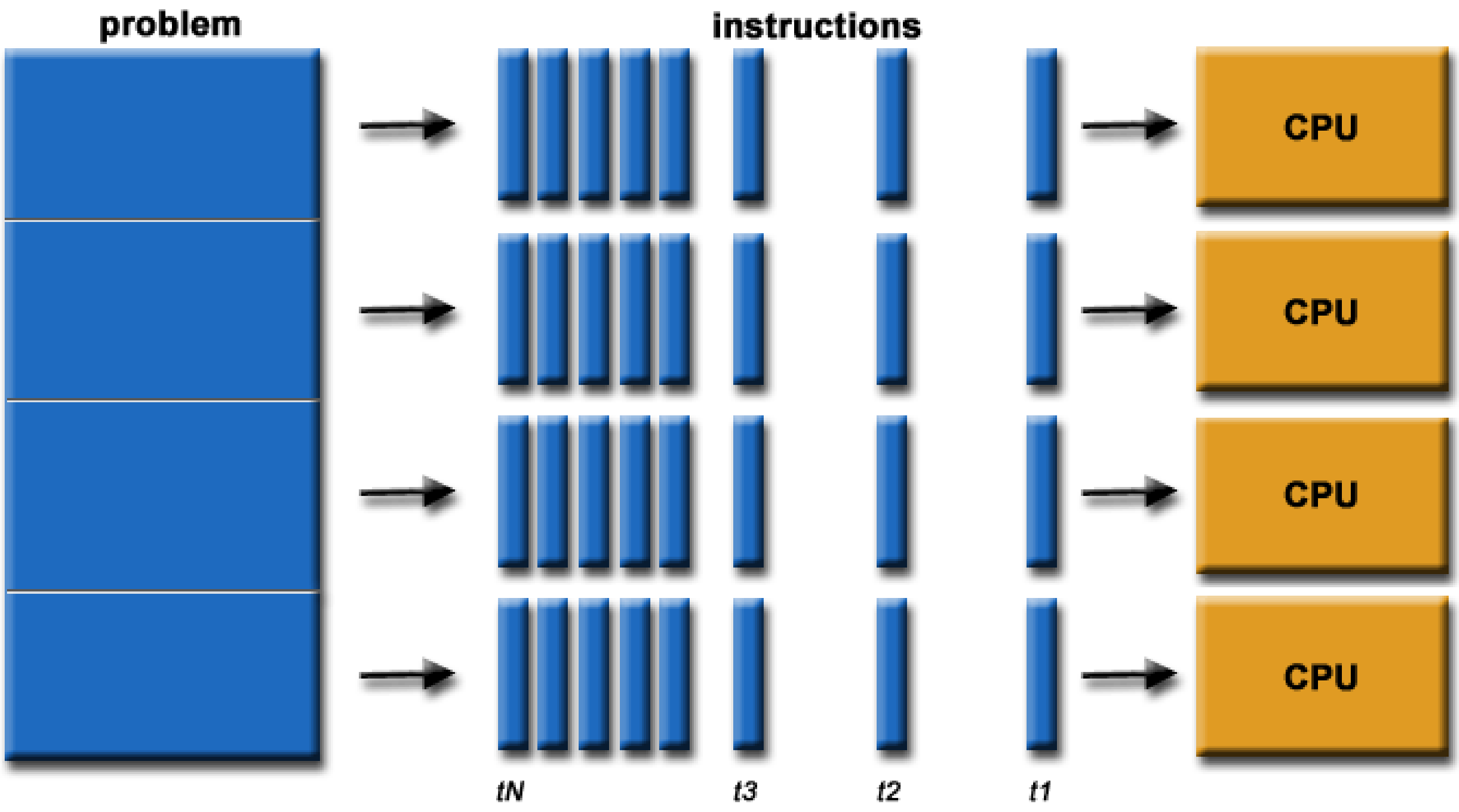
- In real OS process state model is more complicated ...



Why Multiple Processes?

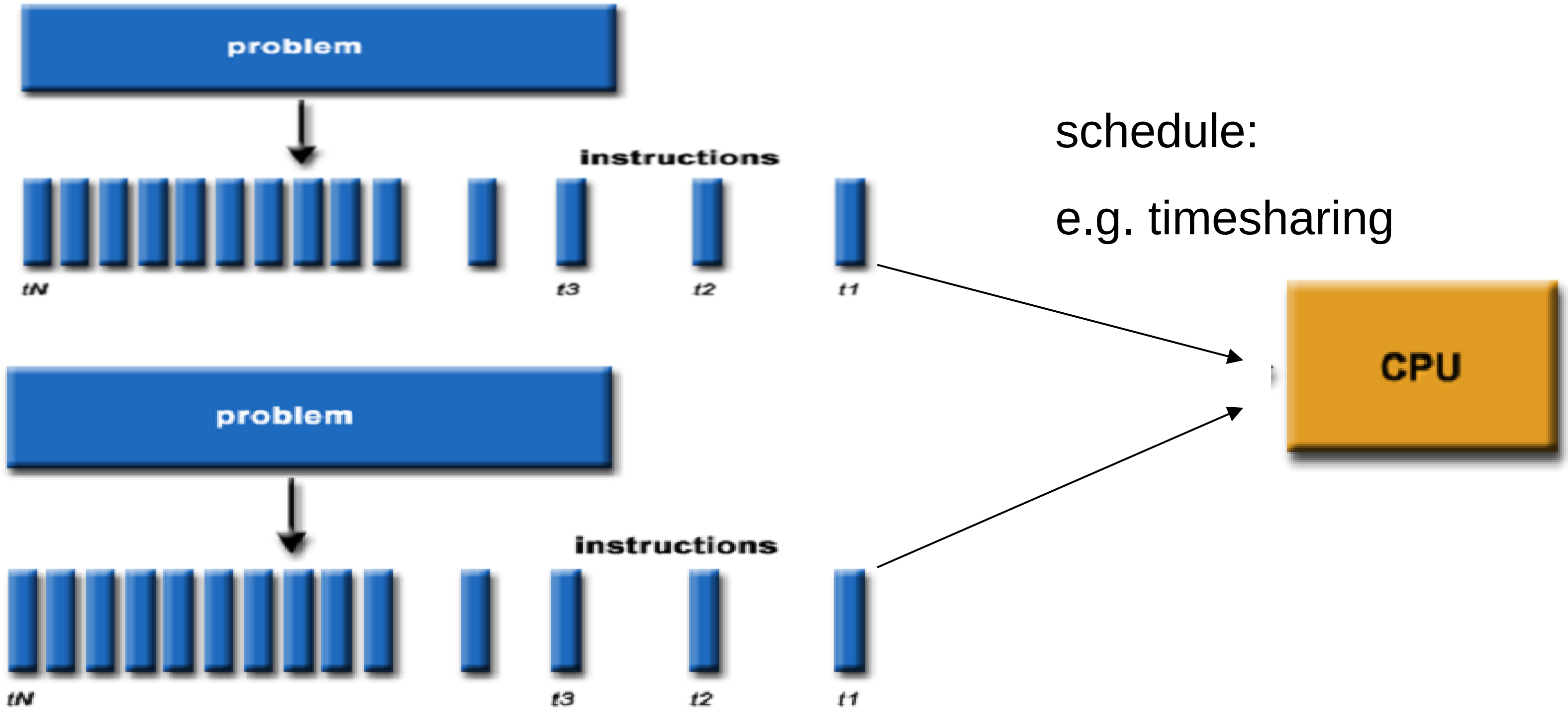
- A method to run multiple programs on one or more processors/cores and share system resources
  - Process
    - Can be an entire 'program'
    - BUT: many 'programs' can also be divided in several 'processes (or threads)' that could be executed simultaneously
      - E.g. web/SQL/mail/... -server handling several requests at the same time
      - E.g. find, sort (cf. qsort), … algorithms, matrix calculations, ...

Why Multiple Processes?



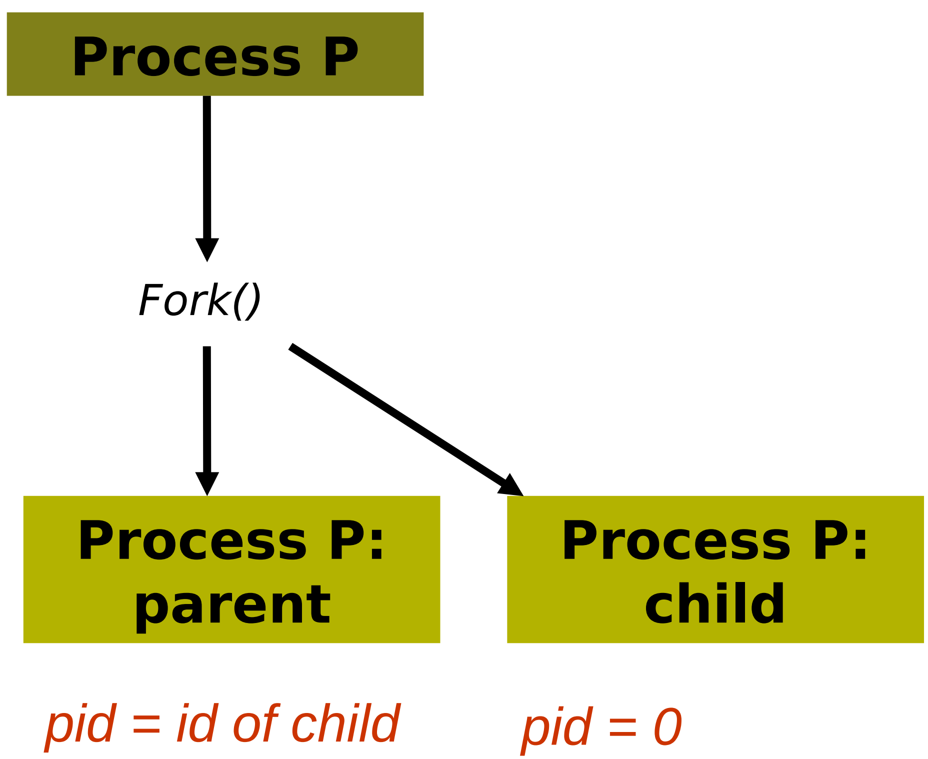
Why Multiple Processes?

- Notice that …
  - Even if there is only CPU it could be beneficial to use multiple processes ….
    - More efficient use of resources



Fork

- A process can create new processes with fork()
  - Creates a duplicate, identical process (child) of the calling process (parent)




```
#include <unistd.h>
int main( void )
{
   pid_t pid, parentPid;
   pid = fork();
   if ( pid == -1 )
      // some error
   else if ( pid == 0  )
      // child’s code
      parentPid = getppid(void);
   else
      // parent’s code
}
```

Fork

- Parent and child both run the same code
- Parent and child each have a private copy of the process environment
  - Program code
  - Variables (copy of stack and shared heap)
  - Including shared input and output file descriptors

- Distinguish parent from child by the return value from fork
  - Returns 0 to the child process
  - Returns child’s pid to the parent process
  - There is no function to get all pids of your children!

Fork

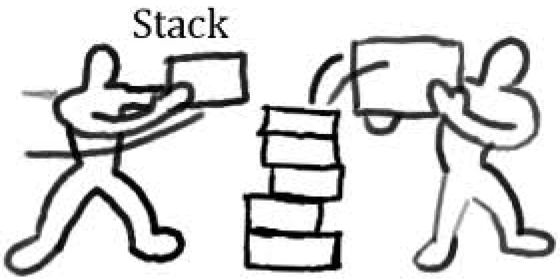
- A call to exit(status_code) terminates the process
  - Atexit(…) can be used to install exit handling functions
  - Normal return with status 0 (success)

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

Fork

- Study the code ...

[Toledo: fork]



Exit

- Parent process should wait on the 'exiting' of all its child-processes
  - Pid_t wait (int *childStatus)
    - Suspends current process until one of its children terminates
      - If multiple children completed, they will be taken in arbitrary order
    - Return value is pid of child process that terminated
    - If child_status != NULL, then integer it points to will be set to indicate why child terminated
      - WIFEXITED and WEXITSTATUS macros can be used to get information about exit status

Exit

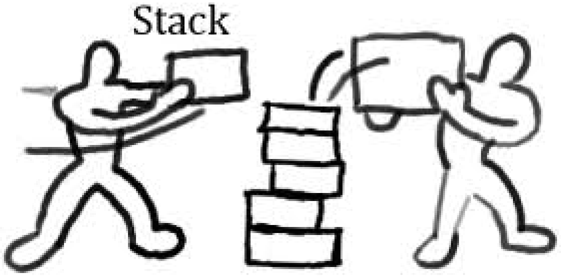
- waitpid(pid_t pid, int *childStatus, int options)
  - Can wait for termination of child process with PID in argument
  - Various options available (see man page)
    - WNOHANG-option
      - Poll if child is exiting and return immediately
      - If child is not exiting, 0 is returned as result of waitpid

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminaed with exit status 103
Child 3572 terminated with exit status 104
```

Exit

- Study the code …

[Toledo: wait / waitpid]



Zombies

- What if child exits when its parent is not executing ‘wait’?
  - Child becomes ‘zombie’ process: it occupies a slot in the process table and finally exits when the parent executes ‘wait’
- What if parent exits before child(ren)?
  - Child(ren) will be adopted by the ‘init’ process!

```
linux> ./fork7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 fork7
 6640 ttyp9    00:00:00 fork7 <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

Exec

End of arg list

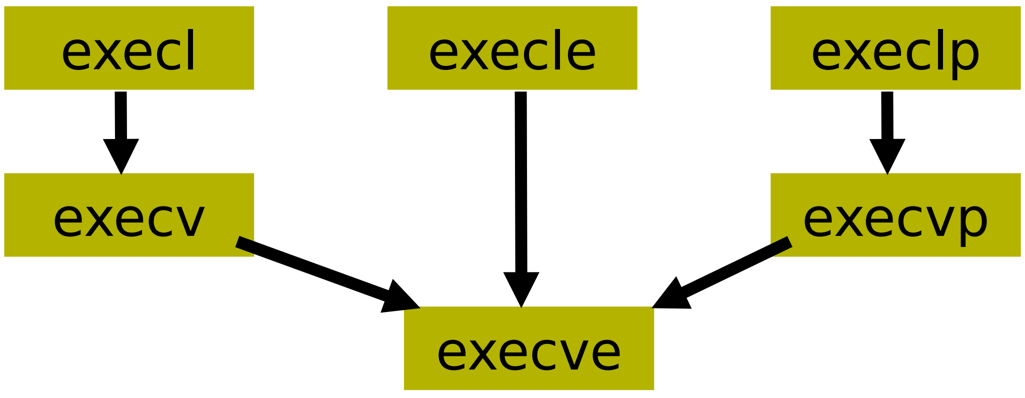
```
#include <unistd.h>
int main( void )
{
    int status;
    status = execl( “ls”, “ls”, “-l”, (char*)0 );

    // should never come here!
    if ( status == -1 )
     printf(“execl failed to run ls\n”);
    return 0;
}
```

Exec

- Exec-call loads and executes a new program in the memory space of the calling process
  - ‘Overlays’ the calling process
    - Replaces the calling program by a new program using the existing process environment
  - Process id remains the same
  - Exec-call doesn’t return, unless an error happened
  - Open files remain open
    - Set the close-on-exec flag for files, sockets, etc.
    - unflushed I/O is lost …
  - …

Exec

- Exec-family: only different in arg passing

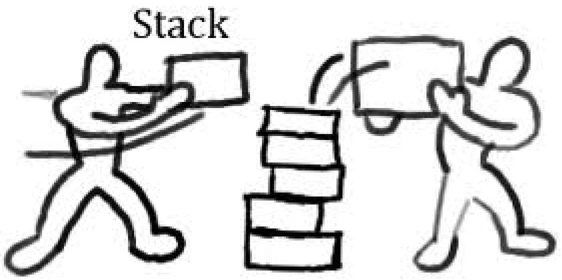



```
Char * arg[] = { “ls”, “-l”, (char *)0 };
Execv( “ls”, arg );
```

Exec+fork

- Study the code …

[Toledo: execfork]



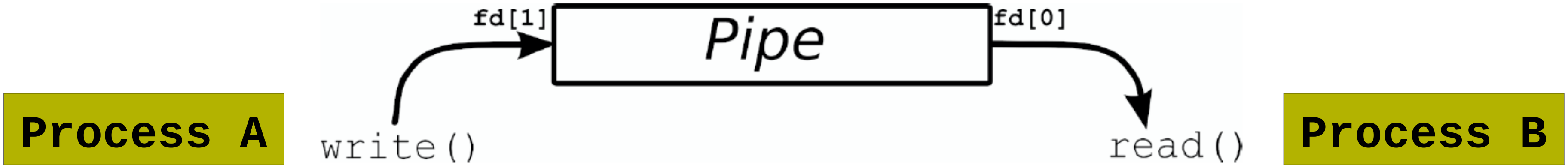
LECTURE M2

IPC

[Inter-Process Communication]

Pipes

- Pipes offer one-way, byte stream-oriented communication



  - A pipe is defined by a pair of pipe descriptors
  - Pipes are created with the system call pipe()
  - Use read() and write() system calls to get/put data from/into the pipe
  - Close all pipe descriptors to close the pipe

Pipe+Fork

- Pipes can only be accessed by the pipe descriptors
  - Parent process creates the pipe … but how do the child processes know the pipe descriptors????

Parent makes a pipe

Parent forks

Data structure (including pipe descriptors) is copied

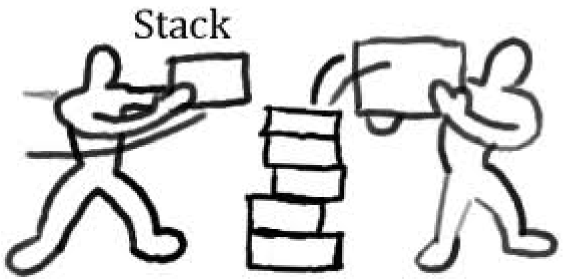
Child can access the pipe (write on it)

Parent can access the pipe (read from it)

Pipes

- Study the code ...

[Toledo: pipe]



Pipes

- Summary
  - Easy to use between parent/child related processes
  - Read/write-fd is closed …
    - Reading results in ‘end-of-pipe’ when ALL write-fds are closed (i.e. read() returns 0)
    - Writing results in ‘sigpipe’ signal when ALL read-fds are closed
      - Ignoring ‘sigpipe’ signal, fails write with EPIPE error
  - Blocking / non-blocking read/write
    - Reading from an empty pipe results in blocking
    - Writing to a full pipe results in blocking

Pipes

- Summary
  - Atomic/non-atomic write()
    - What?
      - An 'atomic write' can not be interrupted by a write action of another process
      - If write is not atomic, data blocks from several process will be merged in the pipe which makes it rather impossible to correctly read these blocks
    - Write() of less than PIPE_BUF bytes must be atomic
  - Pipes are meant to be used between 1 reader process and 1 writer process
    - Multiple readers/writers are possible but hard to implement
  - Pipe has limited capacity
    - Size depends on implementation, e.g. Linux 2.6.11 uses max. 65536 bytes
    - See man 7 pipe

FIFO

- FIFO
  = named, permanent pipe
  - FIFO is created and deleted like a file, but behaves like a pipe when reading / writing
    - Also half-duplex, stream-oriented
  - Easy to use between processes that are not within the ancestory of the pipe-creater process
  - FIFO can survive the processes using it

FIFO

- Pipe resides in the kernel but a FIFO is part of the file system

- FIFO creation
  - On the command line
    - mknod my_fifo p
    - mkfifo --mode=0666 my_fifo
    - rm my_fifo

FIFO

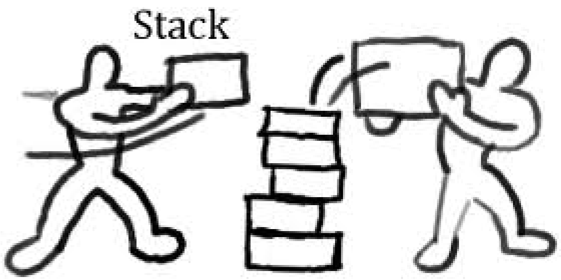
- FIFO creation
  - In program code
    - Int mknod(char *pathname, mode_t mode, dev_t dev)
      - E.g.: err =mknod(“my_fifo", S_IFIFO|0666, 0);
    - Int mkfifo(char *pathname, mode_t mode)
      - E.g.: err = mkfifo(“my_fifo", 0666);
    - Int remove(“my_fifo”);

Err==0: success | *Err==-1: error*

FIFO

- Study the code …

[Toledo: fifo]



FIFO

- Synchronization between reader/writer
  - When FIFO is opened for reading (writing), it will block until another process opens the FIFO for writing (reading)

Blocking and Non-Blocking

- Recall blocking and non-blocking I/O on files ...
  - E.g. ‘read(...)’ blocks until data becomes available
- In most cases, blocking is the **default** behavior!
  - But then, how to read/write/... on multiple descriptors (=pipe, fifo, file, socket, ...) at the same time?

BUT: if data is available on pipe2 but not on pipe1, read blocks on pipe1

```
While ( … ) {
  if ( ( bytes = read(pipe1[0], buf, size) ) > 0 )
    write( stdout, …);
  if ( ( bytes = read(pipe2[0], buf, size) ) > 0 )
    write( stdout, …);
  …
}
```

Blocking and Non-Blocking

- Solution 1: use non-blocking mode
  - FIFOs, files, …
    - int open(const char *pathname, int flags);
      - Use 'O_NONBLOCK' flag
  - Pipes
    - int pipe2(int pipefd[2], int flags);
      - Use 'O_NONBLOCK' flag
  - *Fcntl()* and *ioctl()* allow to retrieve and modify flags of an **open** fd

Read man pages for details!

Blocking and Non-Blocking

- Solution 1: use non-blocking mode
  - Read()/write() on non-blocking fds will return –1 if no data is available; ‘errno’ indicates if an error occured or the next read()/write() might successful

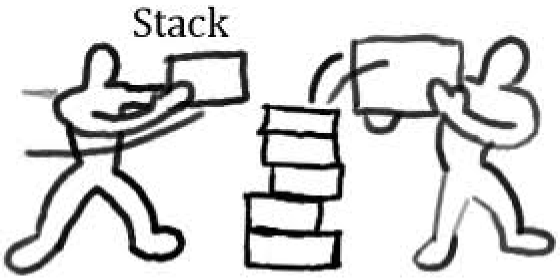
```
Int bytes;
Bytes = read( fd, buff, size);
if ( bytes == -1 ) {
  if ( errno == EAGAIN ) {
     // no data available – not really an error
  } else {
    //handle error and terminate reading;
  }
} else if ( bytes == 0 ) {
   // eof is reached: terminate reading
} else if ( bytes > 0 ) {
   //new data was delivered - process the data
}
```

Read man pages for details!

Blocking and Non-Blocking

- Solution 2: multiplexing I/O on blocking fds
  - Use select(), poll() or Linux specific epoll() system call

[Toledo: Poll]



Other IPC Methods

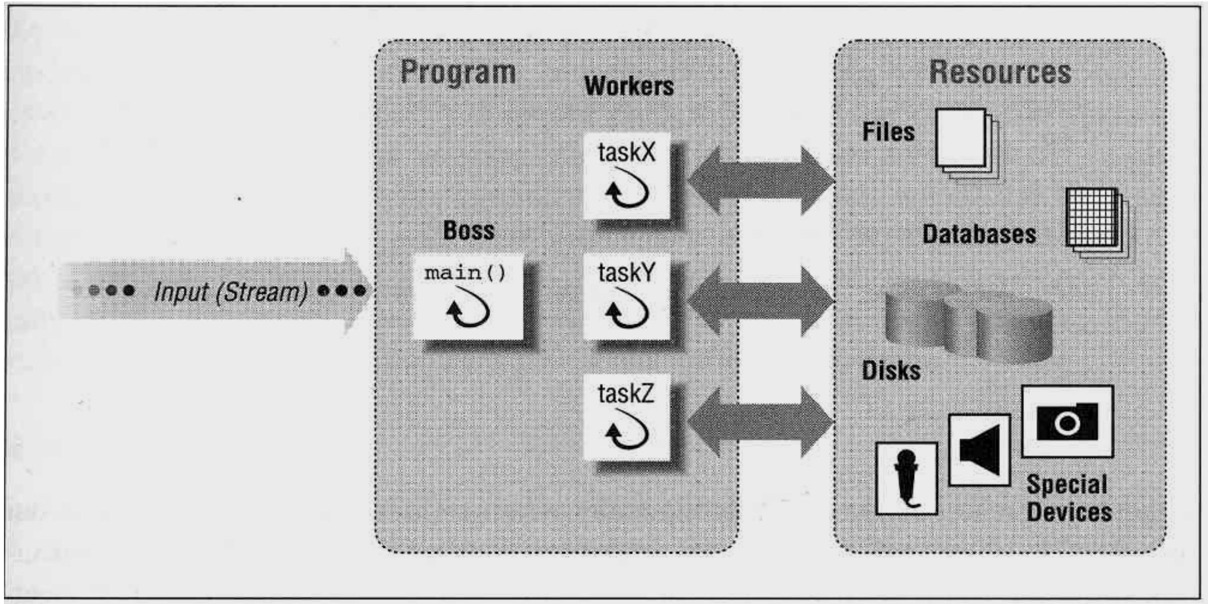
- Message queues
- Mailboxes
- TCP/IP and Unix sockets
- Shared memory
- Remote Procedure Calls (RPC, SOAP, ...)
- …

LECTURE M3

Design Patterns

Design Patterns

- Manager/worker model
  - Manager
    - Collect input
    - Start up workers
  - Workers handle output or synchronize with the manager to let it handle its output
  - Examples
    - Web server
    - File server
    - Database server
    - Window managers
    - …



Design Patterns

- Manager/worker with dynamic worker pool

```
Main() // manager
{
    Forever
   {
        get request;
        create_worker(...);
        ....
    }
}
```

```
main() // worker
{
    perform task;
    synchronize as needed;
    done;
}
```

Design Patterns

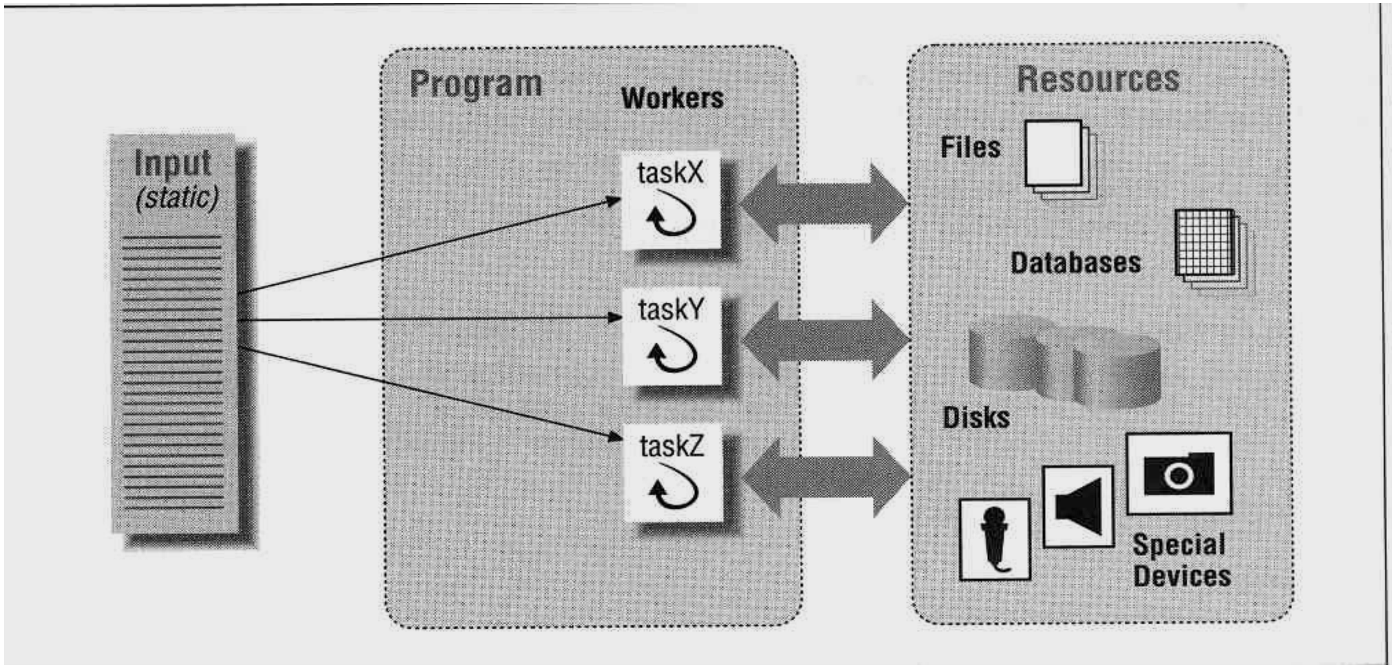
- Manager/worker with static worker pool

```
Main() // manager
{   // create all workers
    For all workers
        create_worker(...);

    Forever
   {
        get request;
        put request in worker queue;
        ...
    }
}
```

```
main() // worker
{
    Forever {
        listen to queue for request; //block
        dequeue (request);
        perform task;
        synchronize as needed;
    }
}
```

Design Patterns

- Peer-to-peer model
  - Manager/worker model but without a manager: in this model, the ‘manager’ creates all peers and then acts as any other peer
  - Each peer must handle its own input/output



Design Patterns

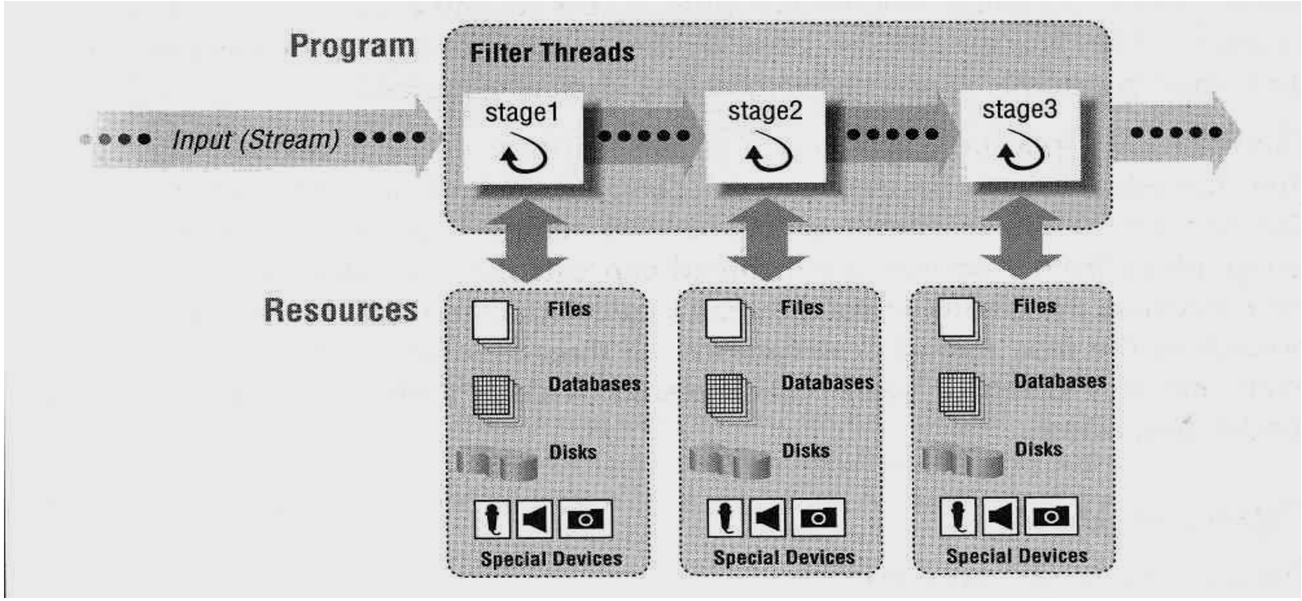
Peer-to-peer model

```
main()  // starting peer
{
    create_peer(... , peer 1);
    create_peer(... , peer 2);
    ...
    signal all peers to start;
    // become also a peer
    while ( ! finished )
        perform task;
    do any clean up;
}
```

```
main() // peer
{
    wait for start;
    while ( ! finished )
        perform task;
    done;
}
```

Design Patterns

- Pipeline model
  - Apply 'filters' on a stream of input ‘packets’
  - Interconnect 'filters' to obtain the desired output
    - Cf.: car assembly line
  - Example
    - Multi media streaming frameworks, e.g. DirectX, gstreamer, image processing, ...



Design Patterns

- Pipeline model

```
main()
{
    create_filter(... , filter 1);
    create_filter(... , filter 2);
    interconnect all filters;
    wait for filters to finish;
    do any clean up;
}
```

```
main()  //filter
{
    while ( ! finished )
    {
        get input packet;
        do filtering on packet;
        pass resulting packet to next filter;
    }
}
```

Design Patterns

- Pipeline model
  - Pipeline model can be static or dynamic
  - Pipeline can be also a tree/graph structure
  - Load balancing across the filter threads is very important
    - Pipeline throughput is limited by the slowest filter

LECTURE M4

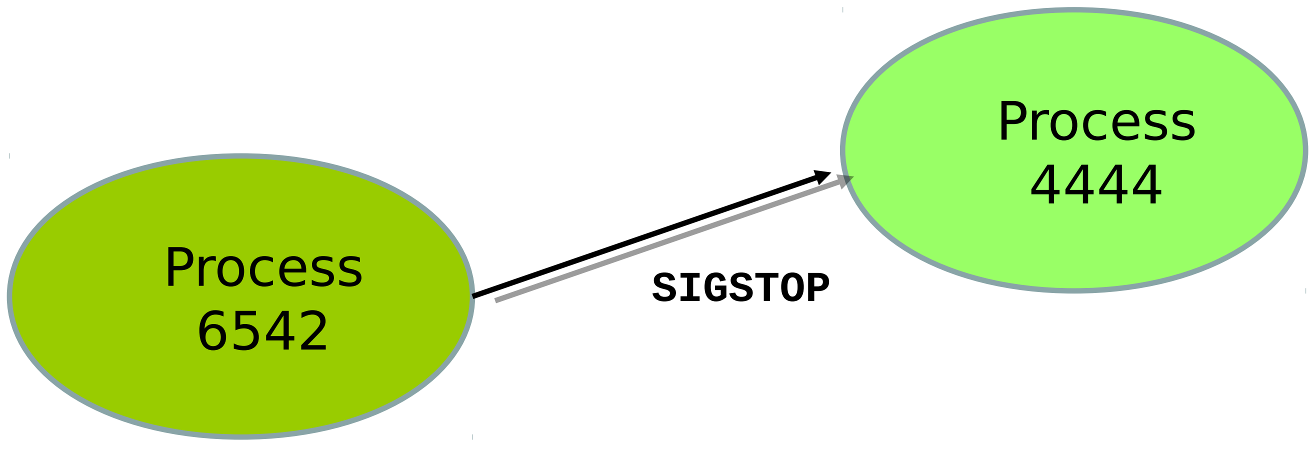
Asynchronous Process Events

Synchronous and Asynchronous

- Synchronous events
  - Events are only notified when asked for
    - E.g. select
- Asynchronous events
  - System notifies when I/O event happens
    - E.g. signals

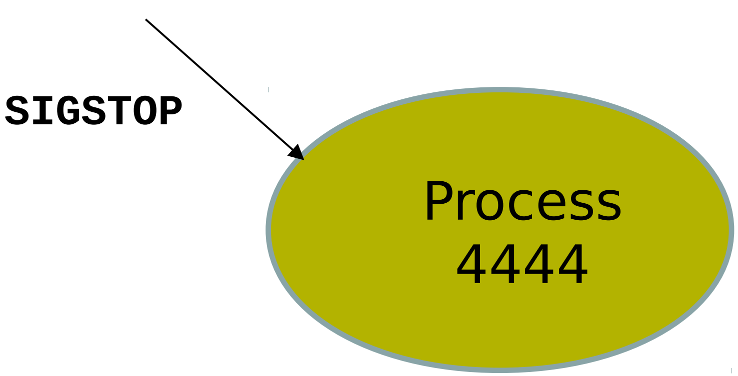
Signals

- Signals are used to notify a program of a particular event
  - ~ software interrupts
  - Signals are asynchronous I/O
  - No data is exchanged!



Signals

- Principle
  - OS or process ‘raises’ a signal to some process
  - The receiving process’ signal handler handles the signal



Signal handler

SIGSTOP	*Ref to handler code*
SIGABRT	*Ref to handler code*
SIGINT	*Ref to handler code*
…	*Ref to handler code*

Signals

Signal constants are defined in signal.h

Signal	Description
SIGABRT	Process abort signal.
SIGALRM	Alarm clock.
SIGFPE	Erroneous arithmetic operation.
SIGHUP	Hangup.
SIGILL	Illegal instruction.
SIGINT	Terminal interrupt signal.
SIGKILL	Kill (cannot be caught or ignored).
SIGPIPE	Write on a pipe with no one to read it.
SIGQUIT	Terminal quit signal.
SIGSEGV	Invalid memory reference.
SIGTERM	Termination signal.
SIGUSR1	User-defined signal 1.
SIGUSR2	User-defined signal 2.
SIGCHLD	Child process terminated or stopped.

SIGCONT	Continue executing, if stopped.
SIGSTOP	Stop executing (cannot be caught or ignored).
SIGTSTP	Terminal stop signal.
SIGTTIN	Background process attempting read.
SIGTTOU	Background process attempting write.
SIGBUS	Bus error.
SIGPOLL	Pollable event.
SIGPROF	Profiling timer expired.
SIGSYS	Bad system call.
SIGTRAP	Trace/breakpoint trap.
SIGURG	High bandwidth data is available at a socket.
SIGVTALRM	Virtual timer expired.
SIGXCPU	CPU time limit exceeded.
SIGXFSZ	File size limit exceeded.

Signals

- The shell command
  - Kill –SIGKILL <pid>
    - Sends ‘SIGKILL’ (signal #9) to process <pid>
- Send a signal to another process
  - int kill(pid, signal_id)
  - Examples
    - Kill(1234, SIGTERM)
      Send the termination signal to the process with id 1234
- Send a signal to ‘yourself’
  - int raise(signal_id)

Signals

- Set your own handler for a signal
  - typedef void (*sighandler_t)(int);

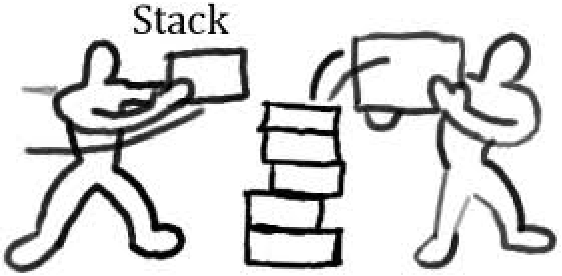
    signal(int signalID, sighandler_t handler);

  - Default handler functions defined in signal.h
    - SIG_IGN: handler that ignores the signal
    - SIG_DFL: handler that implements the default behavior for this signal

Signals

- Study the code ...

[Toledo: Signals]



Signals

- Remark 1
  - If a parent process doesn't want to wait on the exit of its children, then use signal(SIGCHLD, SIG_IGN);
- Remark 2: *check with your Linux if needed*
  - When the signal handler is called, the signal handler for that signal is reset to the default handler!
    So ^C will be handled by sigint_handler only the first time, unless we change the handler to this code:

```
void sigint_handler(int sig) {
  signal(SIGINT, sigint_handler);
  // do real signal handling here
 }
```

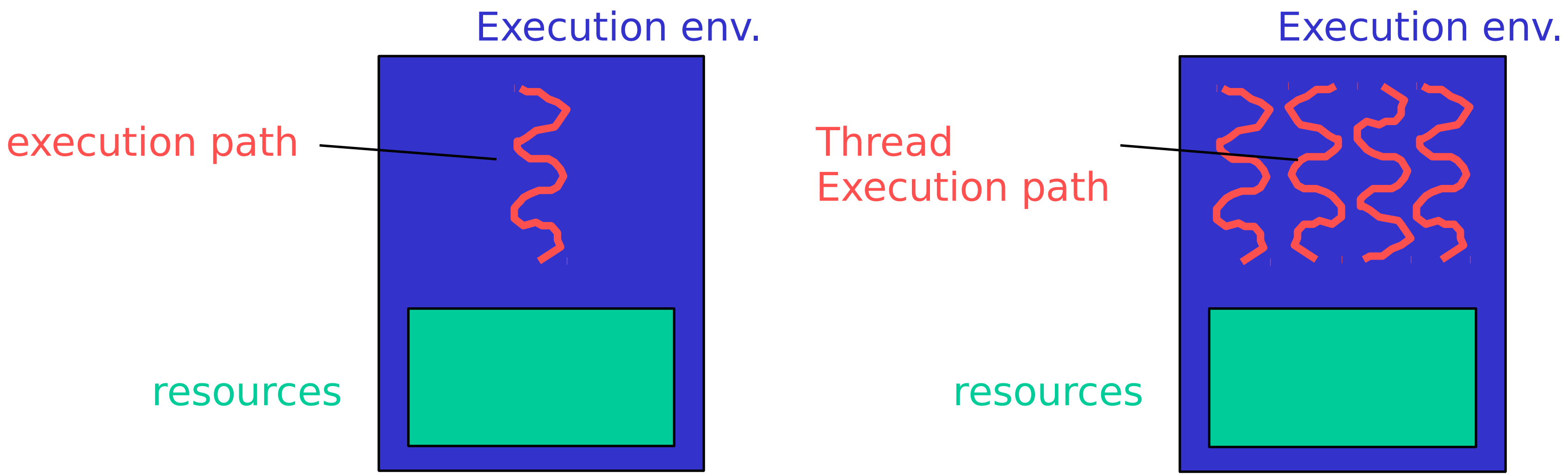
  - Final problem: what happens when many SIGINT may occur -> race condition!

LECTURE M5

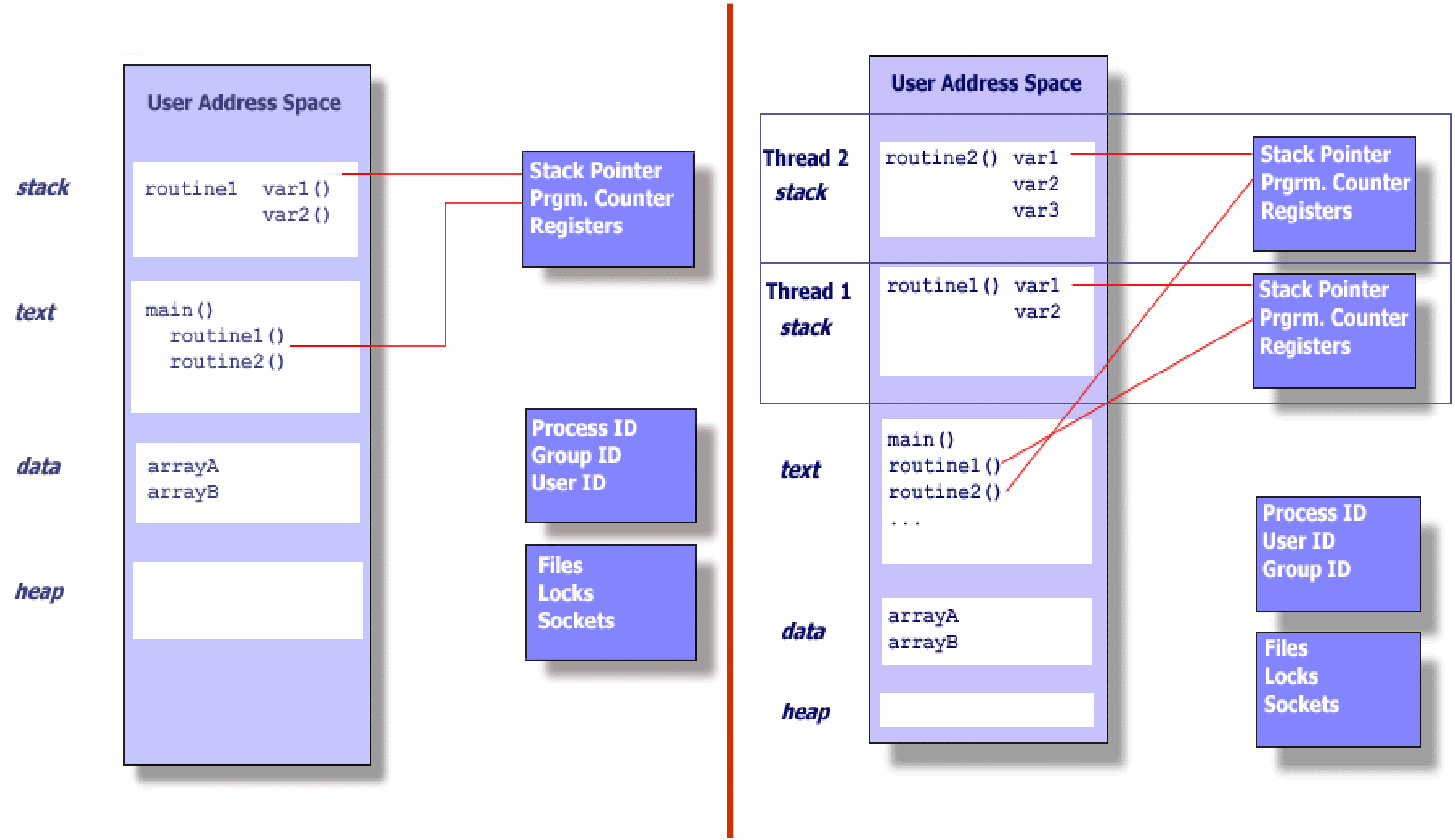
Programming With Threads

Thread

- Processes have
  - Own resources (memory, I/O handlers, …)
  - 1 sequential set of instructions (= execution path)
- Threads
  - Share resources
  - Allow to run multiple execution paths concurrently within a single process

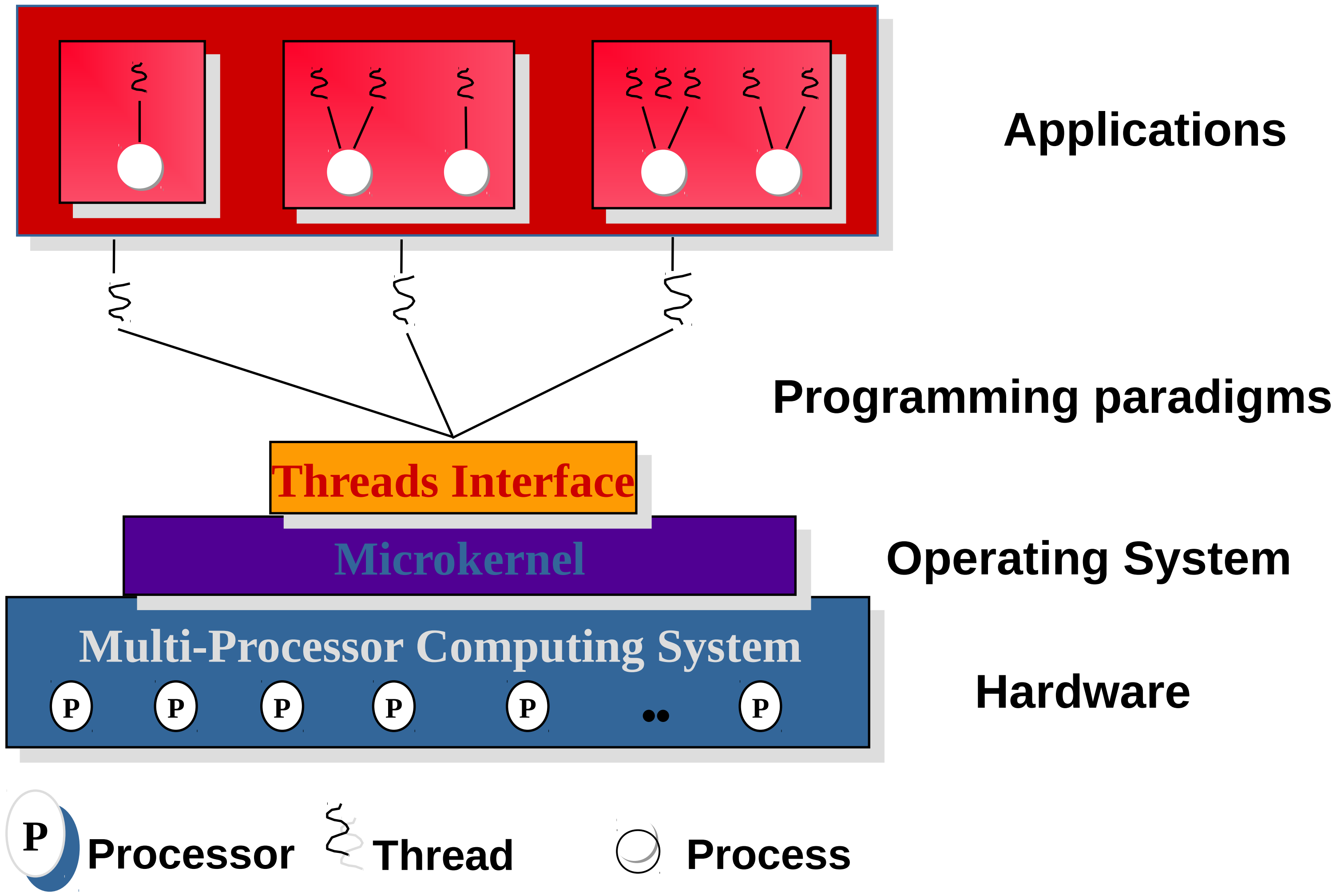


Thread



User Address Space
stack
routine1 var1() var2()
text
main()
routine1()
routine2()
data
arrayA
arrayB
heap
Stack Pointer
Prgm. Counter
Registers
Process ID
Group ID
User ID
Files
Locks
Sockets
User Address Space
Thread 2 stack
routine2() var1 var2 var3
Stack Pointer
Prgrm. Counter
Registers
Thread 1 stack
routine1() var1 var2
Stack Pointer
Prgrm. Counter
Registers
text
main()
routine1()
routine2()
...
data
arrayA
arrayB
heap
Process ID
User ID
Group ID
Files
Locks
Sockets

Thread


Applications
Programming paradigms
Threads Interface
Microkernel
Operating System
Multi-Processor Computing System
P
Hardware
P
Processor
Thread
Process

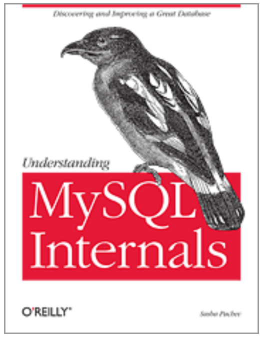
Thread

Thread ~ lightweight process

- Why threads?
  - Creating and managing threads costs less OS overhead
  - Threads require fewer resources (e.g. Memory)
  - Sometimes it is interesting to have different “*processes*” sharing the same resources
  - Inter-thread communication is more efficient (shared address space)

POSIX Threads

- PTHREADS
  - POSIX Threads - IEEE 1003.1C international standard
    - Part of the Linux Distribution
  - Usage: #include <pthread.h> and compile with -pthread flag
- Examples of successful Pthreads applications:
  - Apache
  - MySQL



- Tutorial: many resources available on-line
    - www.llnl.gov/computing/tutorials/workshops/workshop/pthreads

POSIX Threads

```
Int pthread_create( pthread_t *thread, pthread_attr_t attr,
void *(*start_routine)(void*), void *arg )
```

Attr = attributes defining the thread characteristics
    NULL = default values
Return value = 0 : successful / Return value != 0 : failure
    Error number look up in <errno.h>

```
void pthread_exit( void *value )
```

Terminate the thread with return value 'value'
Notice that a call to exit() terminates the entire program (=process and all its threads), not only the thread calling exit!

POSIX Threads

```
Int pthread_join( pthread_t thread, void **result )
```

Caller thread waits on termination of the specified thread

Return value = 0 : successful

Result of multiple simultaneous calls to *pthread_join()* specifying the same target thread are undefined

```
int pthread_tryjoin_np( pthread_t thread, void **result )
```

This is a GNU extension – not POSIX!

Non-blocking pthread_join()

Returns EBUSY if thread had not yet terminated at the time of this call

```
Int pthread_timedjoin_np( pthread_t thread, void **result,
Struct timespec *time )
```

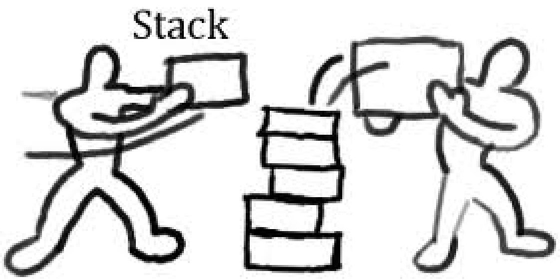
```
Int pthread_detach( pthread_t thread )
```

A detached thread cleans up its resources automatically and doesn’t require a ‘join’ by another thread

POSIX Threads

- Study the code …

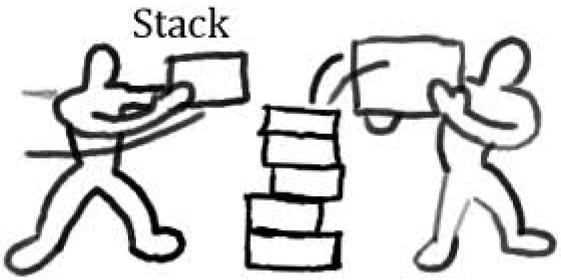
[Toledo: threads]



POSIX Threads

- Thread argument passing: BE CAREFUL!
  - Study the code ...

[Toledo: argPassing]



POSIX Threads

```
Int pthread_yield( void )
```

Yield the CPU: the calling thread is placed at the end of the run-queue and another thread is scheduled to run

```
int pthread_once( pthread_once_t *once_control,
Void (*init_function)(void) )
```

The ‘init-function’ will only executed once, no matter how many threads do subsequent calls to this function

Use as ‘synchronization’ solution of initialization code between multiple threads

Resource Sharing

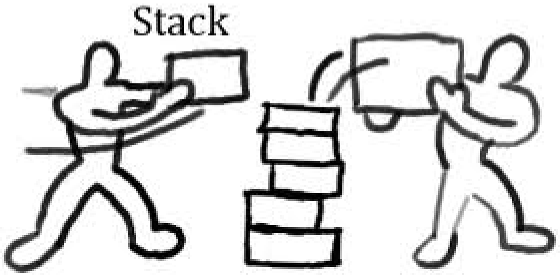
- Threads in the same process share resources
  - E.g. memory (variables), I/O descriptors, …

- What if ...
  - 2 threads open the same file for writing?
  - 1 thread is reading a shared variable while another thread is writing to this variable?
  - 2 threads write output to screen or read input from the keyboard?
  - …

Race Conditions

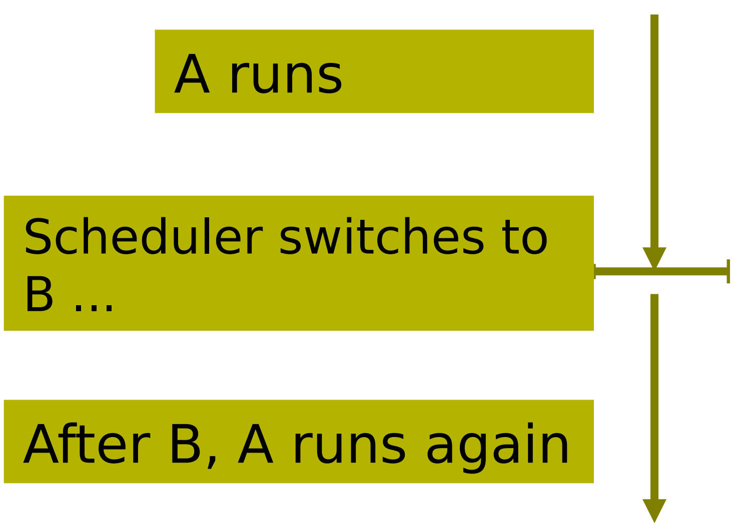
- Study the code …

[Toledo: data sharing]



Reentrancy

- Shared functions / library calls must be reentrant
  - Function is reentrant if it can be interrupted in the middle of its execution and safely called again before its previous invocations are completely executed
  - Example: thread A and thread B both call the function “count”




```
int avg;

int avg( int a[], int length )
{
  avg = 0;
  while (length > 0)
    avg += a[--length];
  ...
  return avg;
}
```

Reentrancy

- Many libc functions are not reentrant!
  - Examples
    - Printf, malloc, … are not reentrant
    - If a reentrant version of a libc function exists, it's often called *func_r*
      - srandom_r(), random_r() : reentrant random number generator
      - strtok_r() : reentrant version of strtok()
      - qsort_r() : reentrant version of qsort()

- Reentrant function should not
  - Use static / global variables
  - Call non-reentrant functions
  - Modify its own code

Synchronization

- How to protect a shared variable between threads of the same process?
  - Only 1 thread may modify the shared data at the same time!
  - Multiple threads may ‘read-without-modifying’ the shared data at the same time!
  - Be aware that single C instructions don’t have to be ‘atomic’
    - An atomic operation is a sequence of machine instructions that are executed sequentially without interruption
    - But, for example, an increment like i++ results into several machine instructions, that can be interrupted
      - Load i into register;
      - Increment register value;
      - Save register value in i;

Synchronization

- How to protect a shared variable between threads of the same process?
  - Access this variable in a **'critical section' (CS)**

```
Int some_shared_data[10];
```

```
Thread A:

Repeat {
  ...
                    Enter CS
// access some_shared_data

  ...               Exit CS
Until ...
```

```
Thread B:

Repeat {
  ...
                    Enter CS
  // access some_shared_data

  ...               Exit CS
Until ...
```

Synchronization

- How to protect a shared variable between threads of the same process?
  - Access this variable in a **'critical section' (CS)**

```
Int some_shared_data[10];
Boolean in_cs = false;
```

```
Thread A:

Repeat {
   ...
  while ( in_cs )
    do nop;
  in_cs = true;            Enter CS

  // access some_shared_data

  in_cs = false;           Exit CS
  ...
Until ...
```

```
Thread B:

Repeat {
   ...
  while ( in_cs )
    do nop;
  in_cs = true;            Enter CS

  // access some_shared_data

  in_cs = false;           Exit CS
  ...
Until ...
```

Synchronization

- How to protect a shared variable between threads of the same process?
  - THIS SOLUTION IS NOT CORRECT!

Thread A:

```
Repeat {
  ...
  while ( in_cs )
    do nop;
  in_cs = true;
  // access some_shared_data
  in_cs = false;
  ...
Until ...
```

Context switch

Thread B:

```
Repeat {
  ...
  while ( in_cs )
    do nop;
  in_cs = true;
  // access some_shared_data
  in_cs = false;
  ...
Until ...
```

Synchronization

- To deal with these problems, synchronization primitives where introduced …
  - Mutex
  - Semaphores
  - Reader/writer locks
  - Barriers
  - Condition variables
  - (spin) locks, ...

Mutex

- POSIX synchronization primitive
  - Mutex variables
    - Mutually exclusive lock
    - Data protected by a mutex allows only one thread at a time to control the data
    - Code protected by a mutex is also called a ‘critical section’

Mutex

```
Int pthread_mutex_init( pthread_mutex_t *mutex,
                        pthread_mutexattr_t *attr )
```

```
Int pthread_mutex_lock( pthread_mutex_t *mutex )
```

Caller blocks until mutex is available

```
Int pthread_mutex_trylock( pthread_mutex_t *mutex )
```

If the mutex is not available, the call immediately returns to the caller ➔ polling style

```
Int pthread_mutex_unlock( pthread_mutex_t *mutex )
```

```
Int pthread_mutex_destroy( pthread_mutex_t *mutex )
```

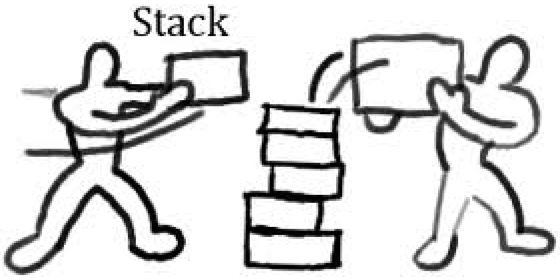
Mutex

- Static versus dynamic initialization of a mutex
  - Pthread_mutex_t myMutex = PTHREAD_MUTEX_INITIALIZER;
  - Pthread_mutex_init(myMutex, NULL);
- Mutex attributes
  - Control process shared or private state
  - Control scheduling and priority related properties

Mutex

- Study the code …

[Toledo: data sharing with mutex]



Semaphore

- Synchronization primitive for processes as well as threads
- Semaphore = variable (integer) that is used to start/stop the execution of processes to implement critical sections. They can only be changed by one of these two ‘atomic’ functions:
  - Wait(...)
  - Post(...)

Semaphore

- Example:
  - Semaphore S = 2 : two more processes/threads may enter CS
    if S becomes 0 : no process/thread can enter CS

```
wait(s)
<critical section>
post(s)
```

- wait(s):

```
while S <=0 do nop;        // Waiting process/thread is blocked
S --;
```

- post(s):

```
S++;        // unlock semaphore
            // Any waiting process/thread is unblocked
```

Semaphore

```
#include <semaphore.h>
sem_t *semaphore;
```

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

Use 'pshared = 0' to indicate that the semaphore is used between threads of a process

```
int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem,
                  const struct timespec *abs_timeout);
```

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

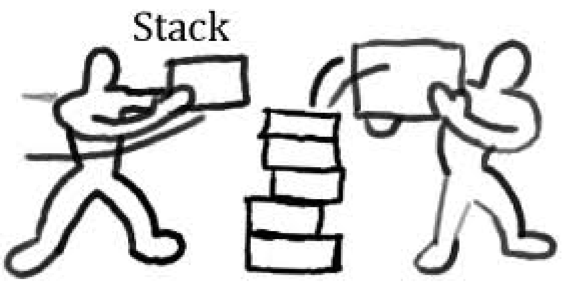
```
int sem_destroy(sem_t *sem);
```

[See man pages for code examples and more info]

Semaphore

- Study the code …

[Toledo: data sharing with semaphore]



Reader/writer locks

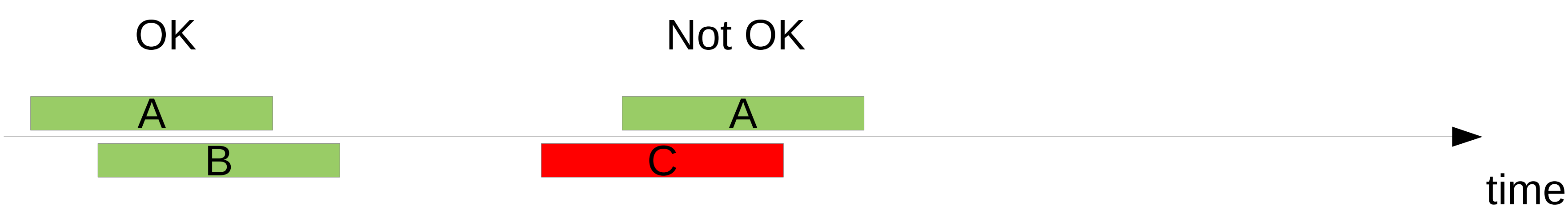
- Assume multiple threads access shared data
  - A writer thread modifies the shared data
  - A reader thread doesn’t modify the shared data
- A simplistic solution:

```
pthread_mutex_lock()
    access shared data (read or write)
pthread_mutex_unlock()
```

- But: multiple readers accessing data doesn’t create synchronization problems, hence, the solution above is suboptimal!

Reader/writer locks

- Reader/writer locks allow multiple readers to access shared data / critical code section at the same time



Thread: reader A & B

```
reader_lock(…)
      Access data
      without
      modification
unlock(…)
```

Thread: writer C

```
writer_lock(…)
      Access data
      with
      modification
unlock(…)
```

Reader/writer locks

```
pthread_rwlock_t *lock;
```

```
int pthread_rwlock_init(pthread_rwlock_t *lock,
                        pthread_rwlockattr_t *attr)
```

```
int pthread_rwlock_destroy(pthread_rwlock_t *lock)
```

```
int pthread_rwlock_rdlock(pthread_rwlock_t *lock)
```

Lock for reading access only

```
int pthread_rwlock_wrlock(pthread_rwlock_t *lock)
```

Lock for writing access

```
int pthread_rwlock_unlock(pthread_rwlock_t *lock)
```

Reader/writer locks

```
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock)
```

```
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock)
```

```
int pthread_rwlock_timedrdlock(pthread_rwlock_t *lock,
                               struct timespec timeout)
```

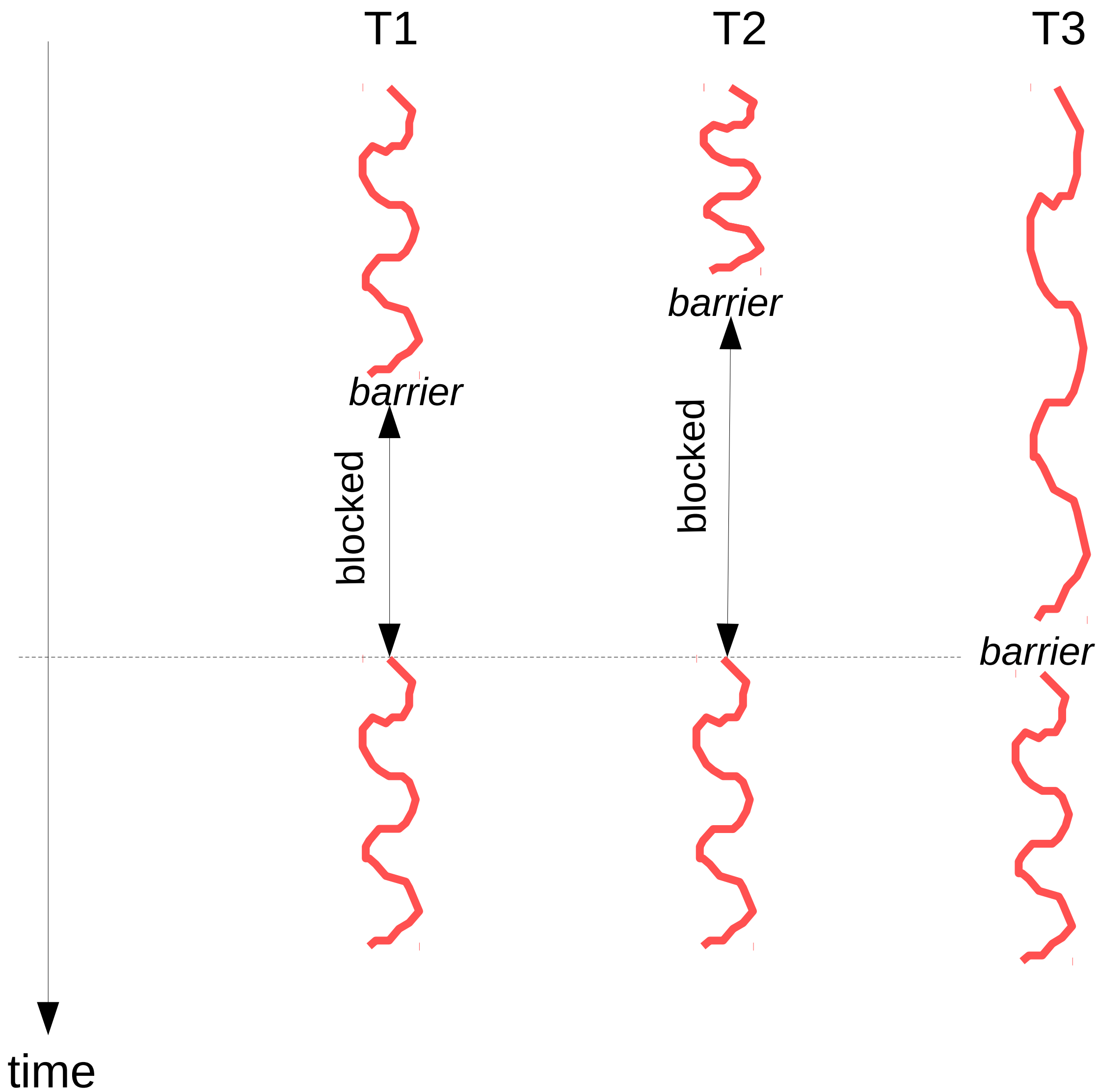
```
int pthread_rwlock_timedwrlock(pthread_rwlock_t *lock,
                               struct timespec timeout)
```

[See man pages for code examples and more info]

Barrier

- POSIX synchronization primitive that forces a defined number of threads to reach an instruction in the program before they can continue
  - Threads that arrive at the barrier wait (block!) until the last thread arrives at the barrier
  - Then all threads are released and can concurrently continue

Barrier



Barrier

```
Int pthread_barrier_init( pthread_barrier_t * barrier,
        pthread_barrierattr_t *attr, unsigned num_threads )
```

Init a barrier that shall ‘wait’ on ‘num_threads’ threads

```
Int pthread_barier_wait( pthread_barrier_t * barrier )
```

Block the calling thread until the required number of threads have called a ‘wait’ on ‘barrier’

```
Int pthread_barrier_destroy( pthread_barrier_t * barrier )
```

[See man pages for code examples and more info]

Condition Variable

- POSIX synchronization primitives
  - Condition variables
    - Used to signal threads that some event has happened; interested threads are waiting (blocking) on this event to happen
      - An event can be a bit flag or a counter that reaches a threshold or …
    - It is used in combination with a mutex

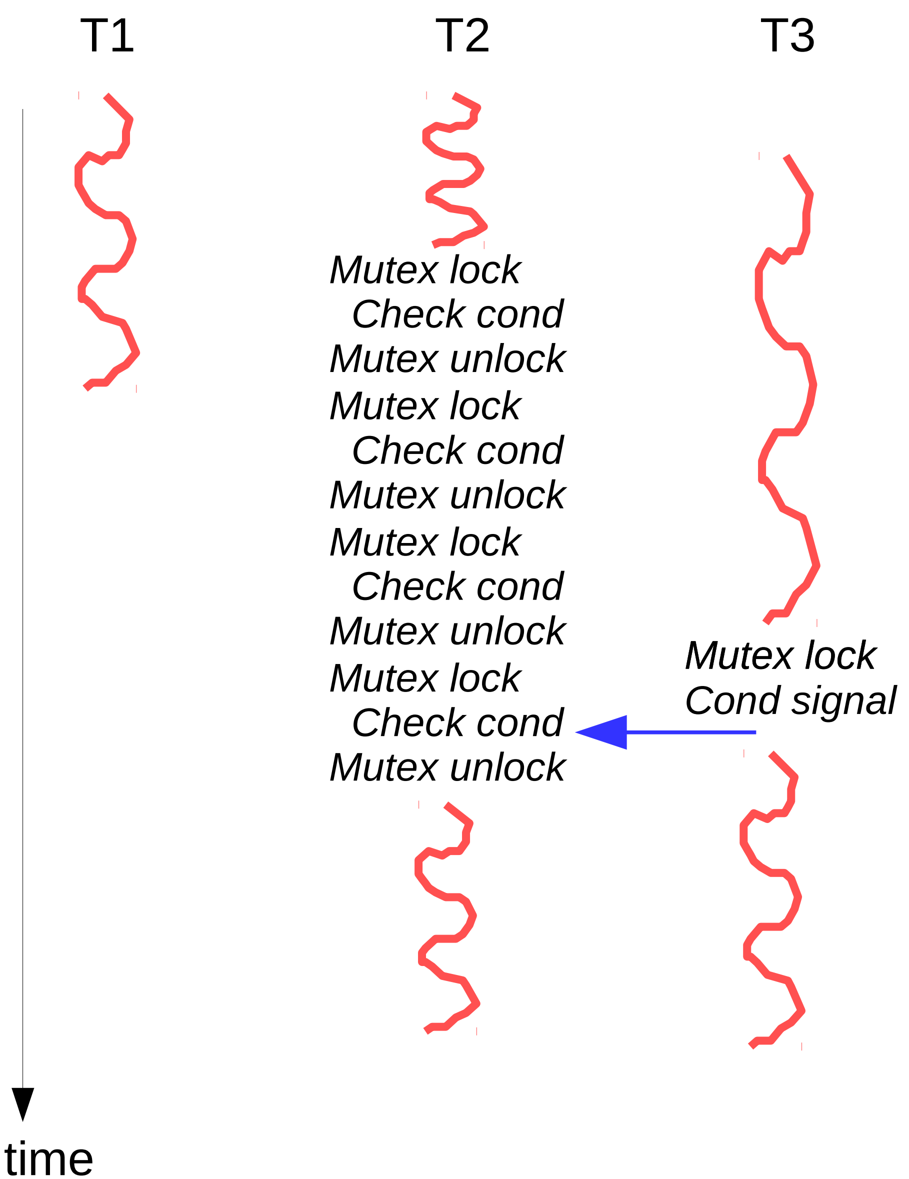
Condition Variable

- Condition variables
  - Why?
    - E.g. assume a thread has to wait until a shared variable has a certain value
  - Using mutex results in an inefficient polling design
  - The system should indicate when this value is reached ==> condition variables in POSIX
  - Mutex allows threads to synchronize by controlling access to variables – a condition variable allows threads to synchronize on the value of a variable
- POSIX condition variables are synchronous

Condition Variable

- Example

```
Pthread_mutex_t dbLock =
PTHREAD_MUTEX_INITIALIZER;
Int dbInitialized;
...
Pthread_mutex_lock( &dbLock );
While ( !dbInitialized ) {
    pthread_mutex_unlock( &dbLock );
    sleep( 1 );
    pthread_mutex_lock( &dbLock );
}
Pthread_mutex_unlock( &dbLock );
```



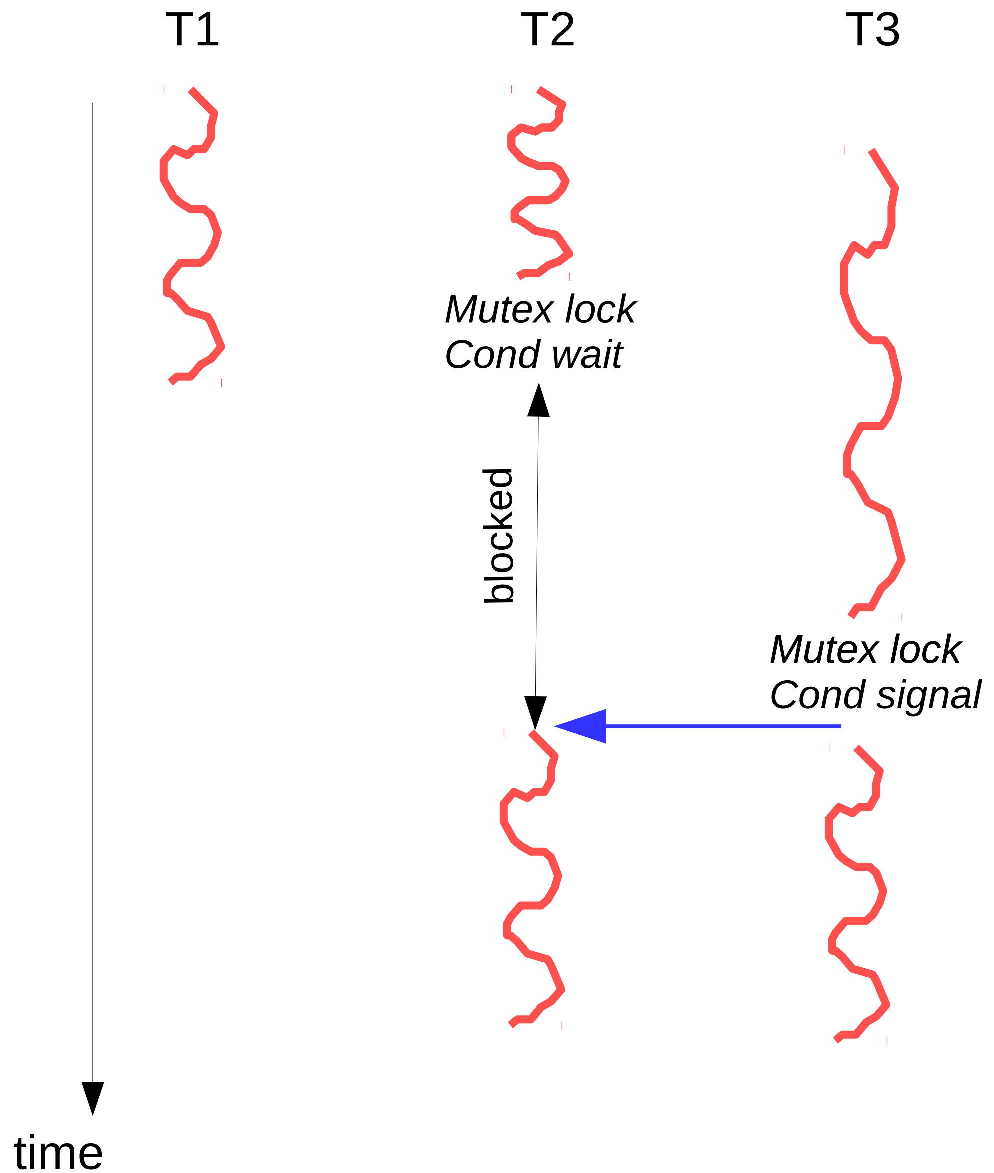
Condition Variable

Example

```
Pthread_mutex_t dbLock =
PTHREAD_MUTEX_INITIALIZER;

Pthread_cond_t  dbInit =
PTHREAD_COND_INITIALIZER;

Int dbInitialized;

...

Pthread_mutex_lock( &dbLock );

pthread_cond_wait( &dbInit, &dbLock );

// do stuff

Pthread_mutex_unlock( &dbLock );
```

Wait until some other thread executes:

pthread_cond_signal(&dbInit);



Condition Variable

```
Int pthread_cond_init( pthread_cond_t *cond, pthread_condattr_t *attr )
```

Attr: Indicates if the condition variable can be shared between processes

```
Int pthread_cond_signal( pthread_cond_t *cond )
```

Unblocks one waiting thread

```
Int pthread_cond_broadcast( pthread_cond_t *cond )
```

Unblocks all waiting threads

```
Int pthread_cond_wait( pthread_cond_t *cond, pthread_mutex_t *mutex )
```

```
Int pthread_cond_timedwait( pthread_cond_t *cond,
     pthread_mutex_t *mutex, struct timespec *abstime )
```

```
Int pthread_cond_destroy( pthread_cond_t *mutex )
```

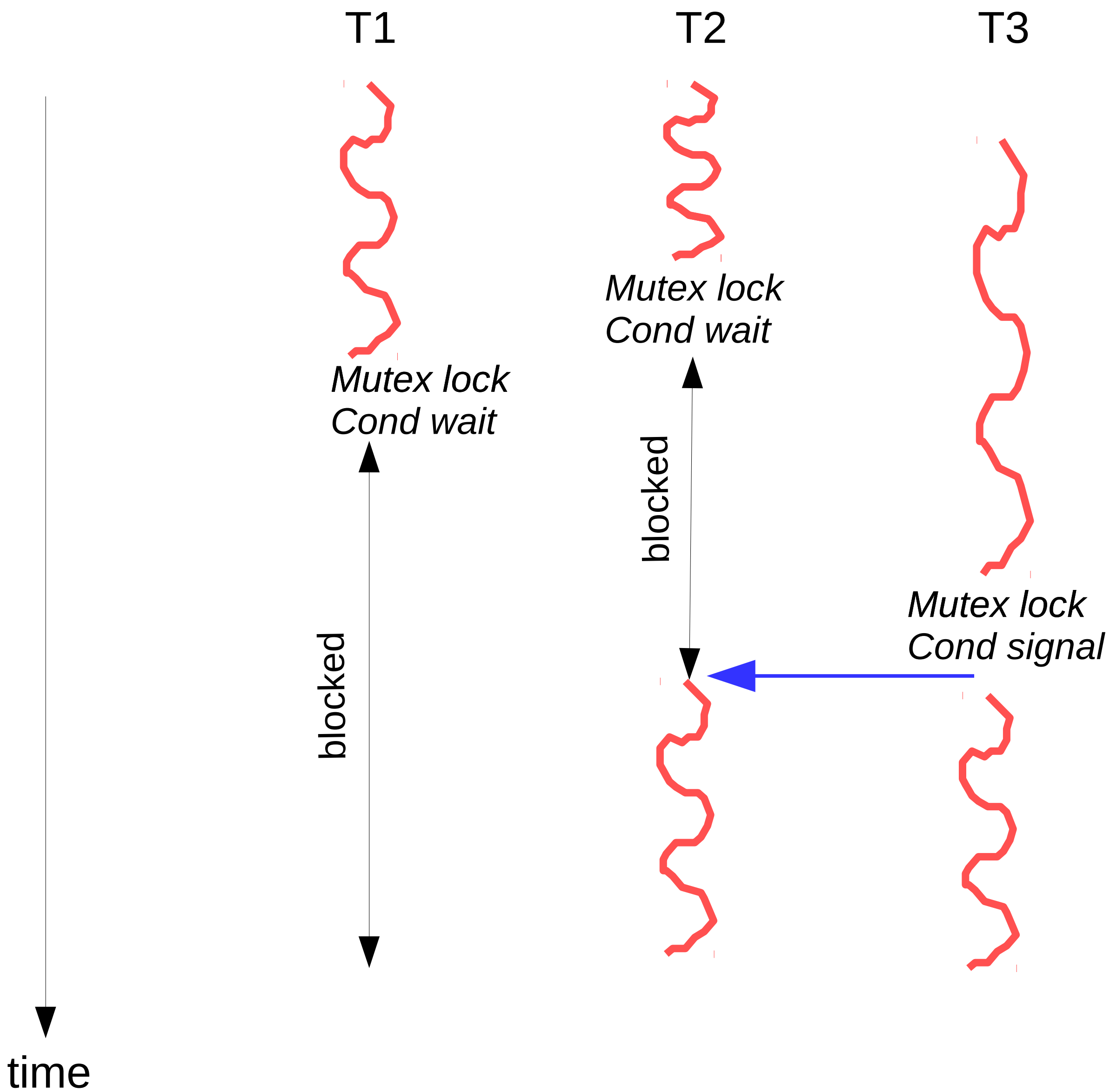
Init and destroy are only for dynamically allocated condition variables

[See man pages for code examples and more info]

Condition Variable

- What if more than one thread is waiting on a mutex / condition variable?
  - Choice is made according to the scheduling priorities of the threads
  - Choice can be randomly or FIFO-style between threads with equal priorities
  - Priorities might result into the ‘priority inversion’ problem

Condition Variable



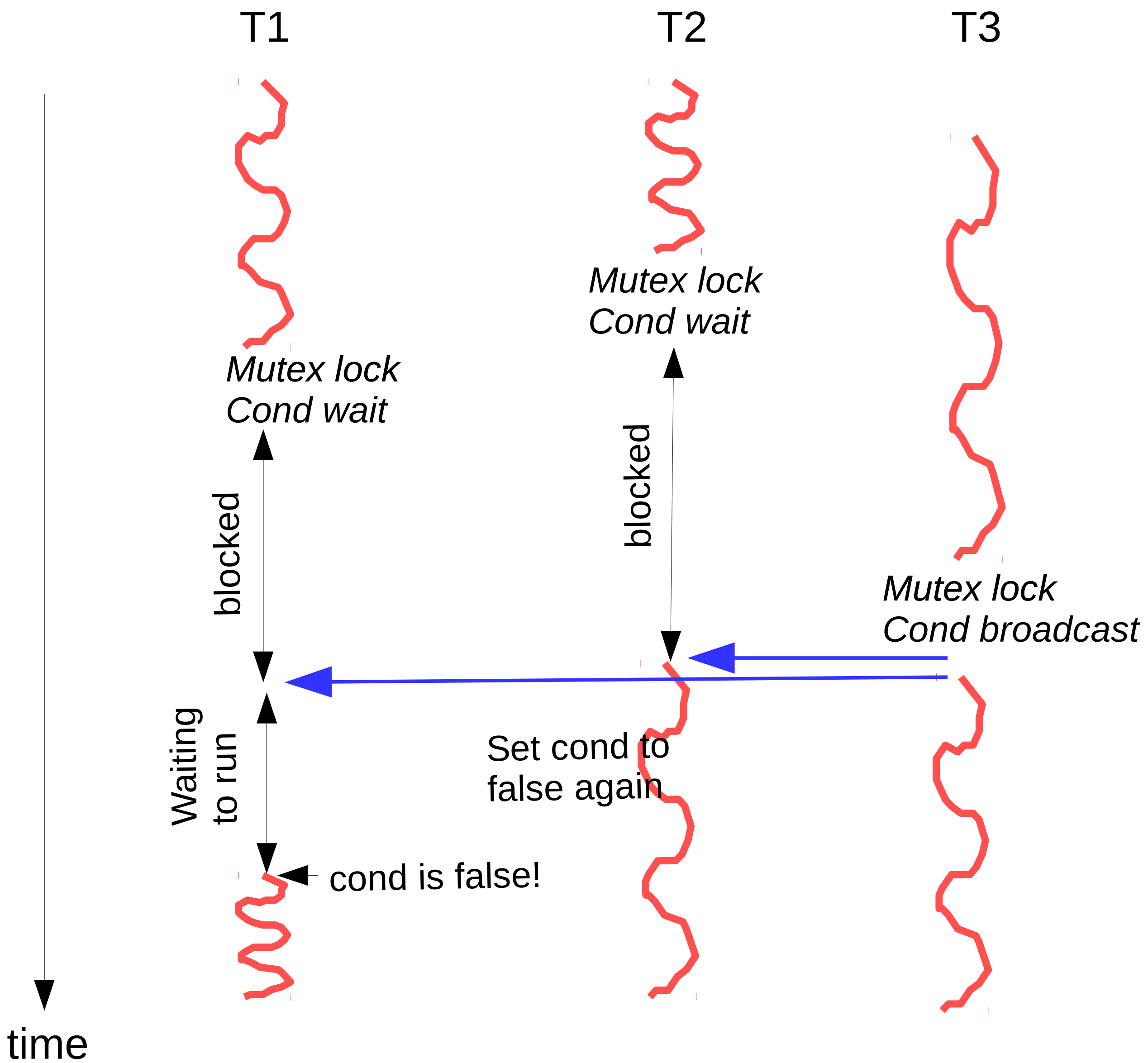
Condition Variable

- Better: use a spurious wake up

```
pthread_mutex_lock( &dbLock );
while ( !dbInitialized )
{
   pthread_cond_wait( &dbInit, &dbLock );
}
...
pthread_mutex_unlock( &dbLock );
...
```

Spurious wake up: after awakening, the thread must check again the while condition because another waiting thread could be awakened before this thread.

Condition Variable



Synchronization Problems

- Thread must be cancellation-safe
  - Example
    - A thread is cancelled while resources are locked
    - A thread is executing library calls (i.e. code not under control) and is cancelled
  - Cancellation is postponed until a ‘cancellation point’ is reached
  - Define a clean up stack for a thread: set of routines that do some final processing before the thread terminates

Synchronization Problems

```
Int pthread_cancel( pthread_t * thread );
```

Send cancellation request to 'thread'

```
Int pthread_testcancel( pthread_t * thread );
```

Create cancellation point in 'thread'

```
void pthread_cleanup_push( void (* function)(void *) );
```

Pushes clean-up handler 'function' on thread's cancellation stack

```
void pthread_cleanup_pop( int execute );
```

Pops clean-up handler 'function' from thread's cancellation stack and executes it if 'execute' is nonzero

[See man pages for code examples and more info]

Synchronization Problems

- Performance issues
  - Lock granularity: the level at which locks are applied on shared data
    - Coarse grain locking: use a lock on the shared data
    - Fine grain locking: use locks on individually accessible pieces of the shared data
  - Examples
    - A shared database: each request might lock the entire database, but then all other request are blocked …
      - Only relevant tables and table rows should be locked
    - Reader/writer locks
      - Readers can share the same data at the same time as long as no writer is active
      - Writer must have exclusive access to the data

Synchronization Problems

```
Pthread_mutex_t countLock =
PTHREAD_MUTEX_INITIALIZER;
Int count = 0;
Void f( /* parameters */ )
{
    // declarations …
    pthread_mutex_lock( &countLock );
    for ( i = start; i < end ; i++ ) {
        …  // computations
            count++;  // update counter
        …  // more computations
    }
    pthread_mutex_unlock( &countLock );
}
```

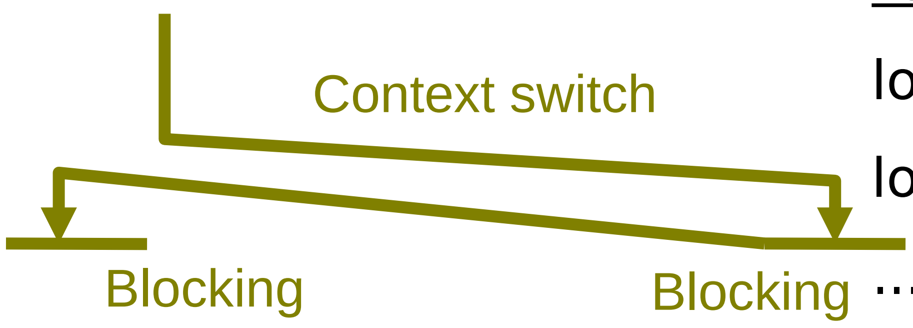
Better?

```
Pthread_mutex_t countLock =
PTHREAD_MUTEX_INITIALIZER;
Int count = 0;
Void f( /* parameters */ )
{
    // declarations …
    for ( i = start; i < end ; i++ ) {
        …  // computations
        pthread_mutex_lock( &countLock );
            count++;  // update counter
        pthread_mutex_unlock( &countLock );
        …  // more computations
    }
}
```

Synchronization Problems

- Deadlock (“deadly embrace”)
  - Example: task A and B are competing for the same resources S and T using locks

<u>task A</u>	<u>task B</u>
lock(S);	lock(T);
lock(T);	lock(S);
…	…
unlock(T);	unlock(S);
unlock(S);	unlock(T);



Objectives Of This Course Part

- Understand the concept of system calls.
- Be able to look up system calls (man pages) and be able to use system calls in C software.
- Be able to work with files (sequential and random access) and program file I/O operations using the Linux system as well as the C library interface.
- Be able to use the strace/ltrace utility.
- Be able to critically reflect on a multi-tasking solution for a given application problem.
- Know and be able to implement the multi-tasking design patterns for a given application.
- Be able to explain the properties of processes and threads and the differences between them. Be able to critically choose between processes or threads for a given application.
- Understand how Linux deals with processes and threads (process and thread model, context switches, process states, …) and be able to work with process/thread-related Linux tools (e.g. ps, kill, …).

Objectives Of This Course Part

- Be able to implement processes (e.g. fork, exec) and IPC (e.g. pipe, fifo, signal) on Linux. Be able to implement blocking as well as non-blocking, and synchronous as well as asynchronous I/O, and critically choose between these I/O techniques.
- Be able to implement threads on Linux.
- Recognize synchronization problems (e.g. shared resources, race conditions, reentrancy) and be able to use synchronization primitives (e.g. criticial sections, mutex, semaphore, cond. var.) in implementations to avoid them.
- Recognize synchronization problems (e.g. cancellation-safety, lock performance, deadlock) introduced by synchronization primitives and be able to solve them.
- Understand and be able to use the terminology introduced in this course part.