howest
university of applied sciences

Practical Reverse Engineering and Malware Analysis (PREMA)

Lecture 2
2025-2026

howest
university of applied sciences

Safe environment - “sandbox”

Virtual Machine as a “safe environment”

- Actually, **not** that safe!
  - Yes, you can “escape” from a virtual machine
  - At the end everything is simply a process running on physical hardware
  - Luckily not that common if you update your software

- Malware can find a way through the network
  - Disable the network (= “pulling the cable”)
  - Host-only network → No internet, but malware can still reach out to your host through the network

Virtual Machine as a “safe environment”

- How to safely transfer files to a virtual machine for malware analysis?
  - Shared folders?
    - Possible, but not advised to have them when you detonate the malware
  - scp (secure copy, sftp, etc)
    - Good, safe and encrypted way over the wire, requires a network connection
    - Still, not advised to have a network connection open to your host
  - Drag & Drop (or copy paste)
    - Requires VMware tools
  - Webserver
    - Seems like a lot of work ☺
    - python -m http.server

howest university of applied sciences

Virtual Machine as a “safe environment”

In other words, there is no “ideal” way to transfer the files safely.

That’s why we create archives (.zip files for example) that are password protected. This encrypts the data, making sure the malware will not cause any harm when the data is in transit.

Once transferred, you can disconnect all network connections, shared folders if you want.

→ We will not create malware that spreads itself over the network in this course

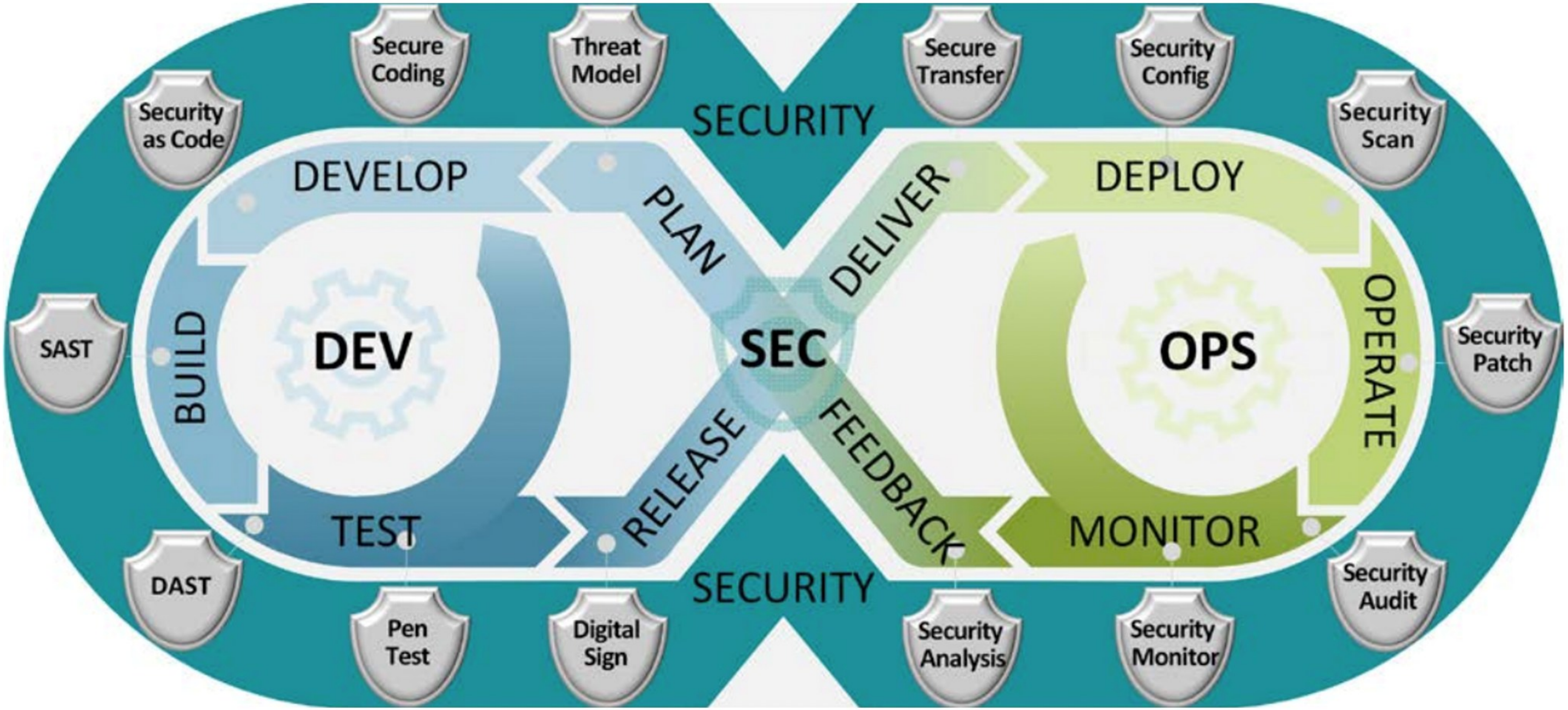
Virtual Machine as a “safe environment”

- Some malware detects if it is running in a virtual environment and acts accordingly
  - Bare metal machine as a sandbox

- Some malware does spread through the network, which you would want to analyse
  - Wireshark and other tools can help (see later)
  - Creating a lab network instead of 1 sandbox

Building software

DevSecOps



howest
university of applied sciences

How developers build software

- **Writing Code**: Using a programming language that fits the task.

- **Compilation or Interpretation**: Turning source code into something the machine can run:
  - **Compiled** → Produces a native executable (C, C++, Rust).
  - **Managed/Intermediate compiled** → Compiles to bytecode, run on a VM (Java, C#).
  - **Interpreted/Scripting** → Run directly by an interpreter (Python, JavaScript), though JIT compilation can boost speed.
  - **Markup/Style (HTML/CSS)** → Not executable by themselves but interpreted by browsers.

- **Linking/Packaging** : Combine code, libraries, and resources into deployable applications.

- **Deployment** : Delivering the software (binaries, bytecode, web apps, containers).

Managed vs unmanaged code

Managed code:

- Runs “on” a runtime environment
  - This handles memory management, garbage collection, exception handling, etc.
- Typically considered “higher” and easier to program

- Examples are Java/Kotlin with JVM, .NET CLR (C#)

Managed vs unmanaged code

Unmanaged code:

- Runs directly on the operating system
  - Developers must handle memory management, garbage collection, exception handling, pointers etc. themselves!
- Typically considered to have more control and higher performance

- Examples are C, C++, Rust,

C(++) Compiling & Linking

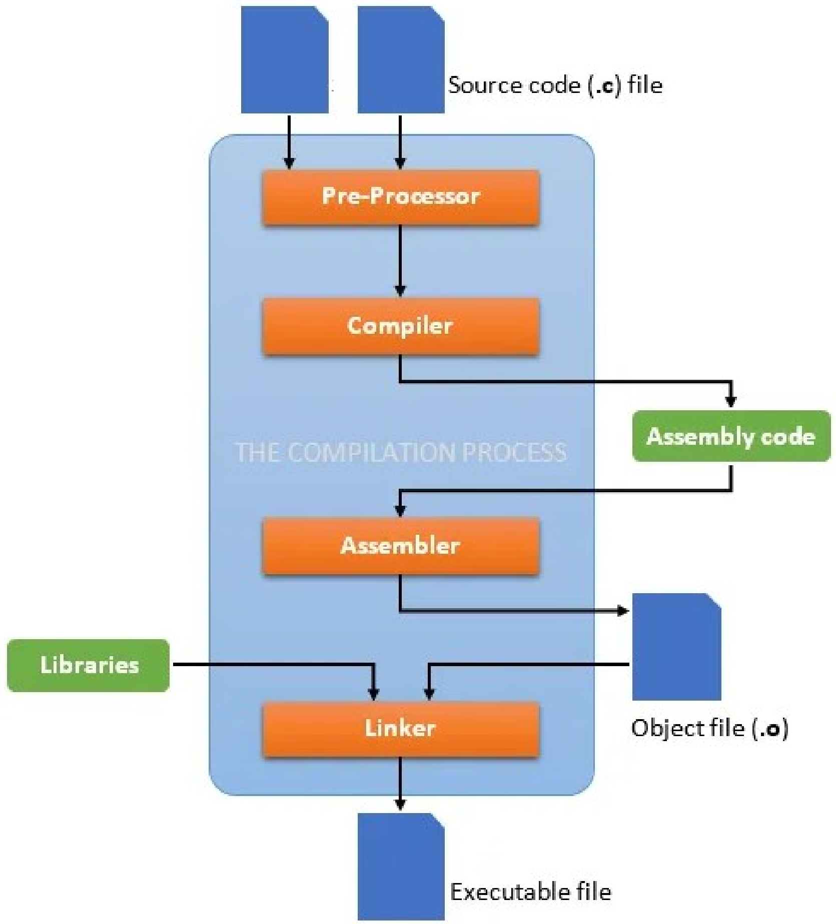
To **compile** a .c-file you can either use:

gcc hello.c -o hello

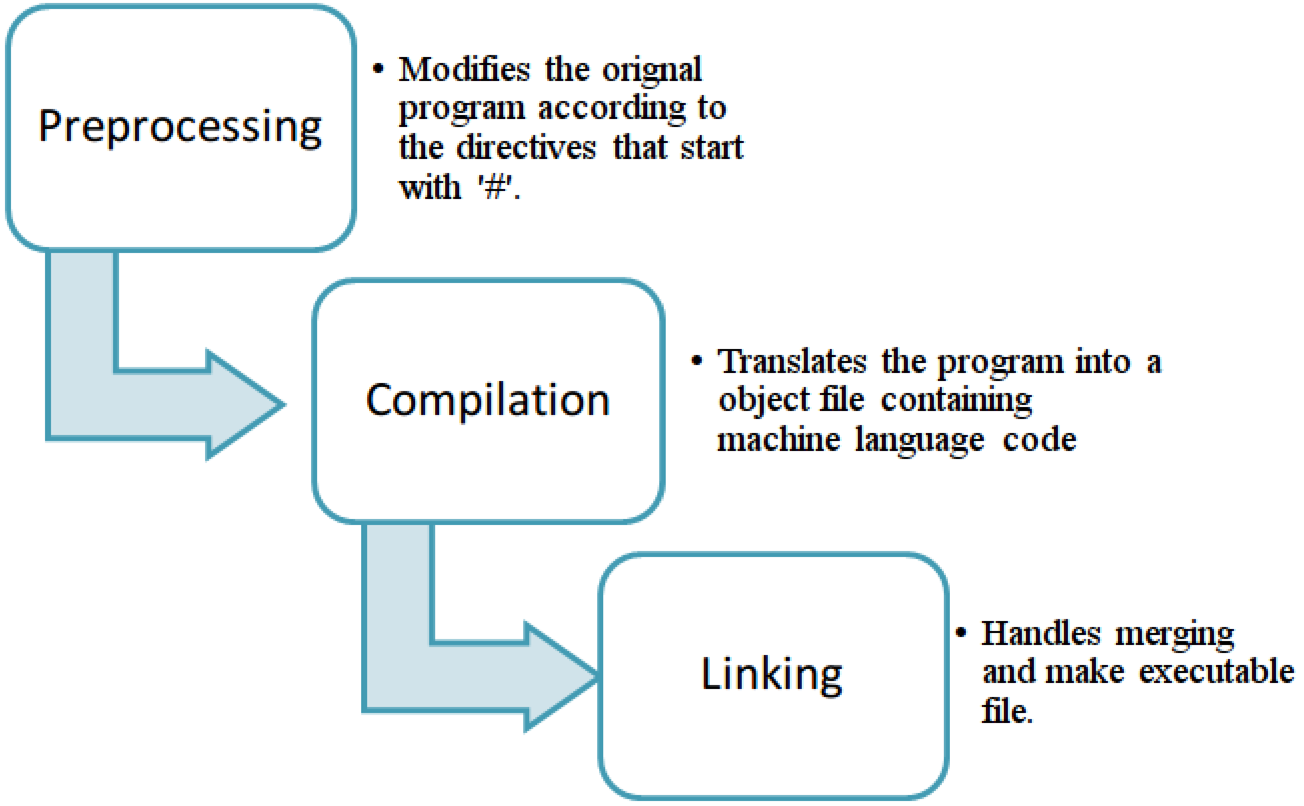
Or

make hello

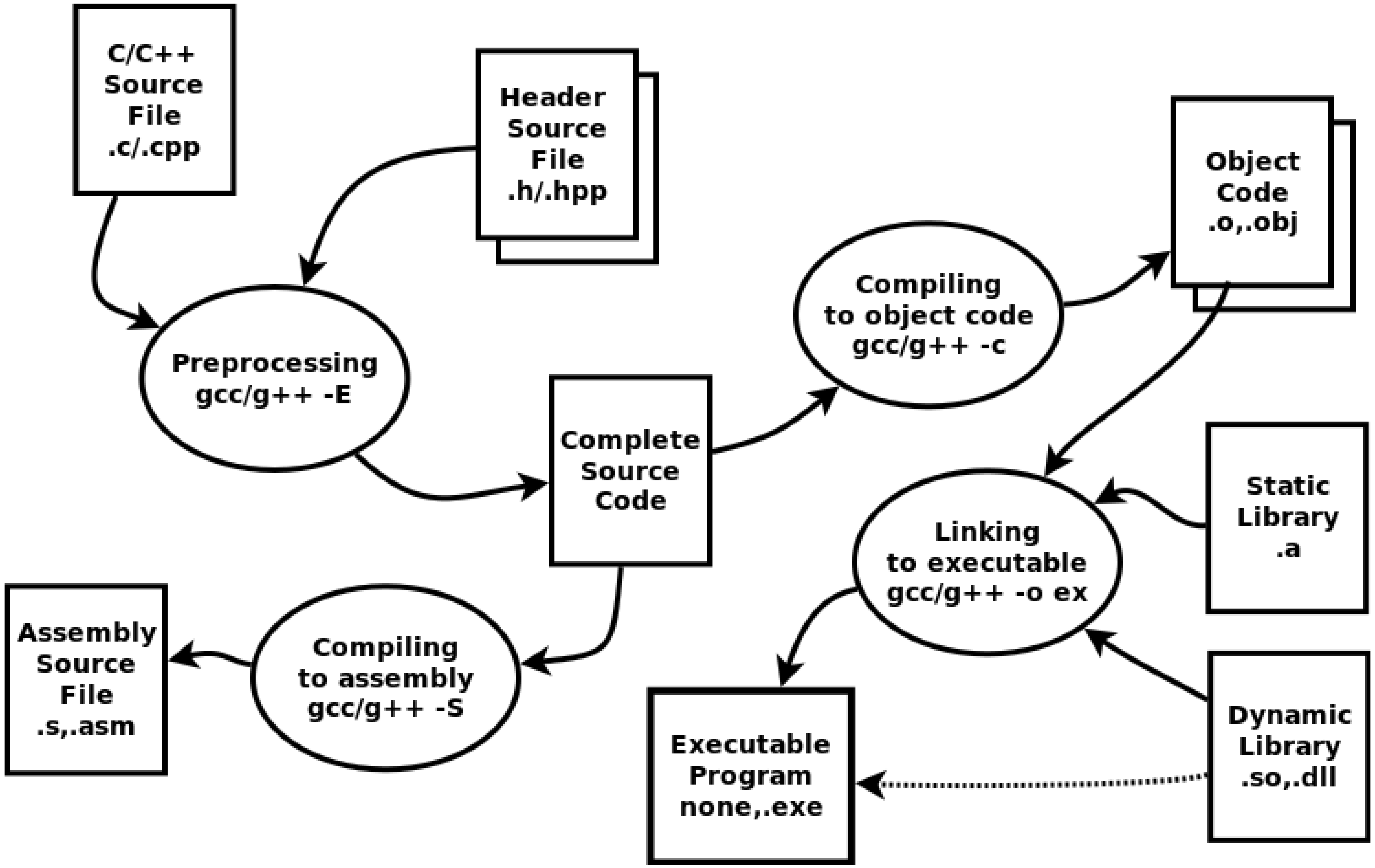
This will create a new exectuable file



C(++) Compiling & Linking



C(++) Compiling & Linking



Statically linked or dynamically linked

File: Dockerfile

```
FROM golang:1.15-alpine as dev

WORKDIR /work

```

File: server.go

```go
package main

import (
    "fmt"
    "net/http"
)

func main() {
    http.HandleFunc("/", Index)
    http.ListenAndServe(":8888", nil)
}

func Index(w http.ResponseWriter, r *http.Request) {
    fmt.Fprintf(w, "Hello, %s!\n", r.URL.Path[1:])
}
```

```
debian@debiandocker:~/dockerdemos/go$ docker run -it -v $(pwd):/work mygo sh
/work #
```

go build -o server1

Vs

CGO_ENABLED=0 GOOS=linux GOARCH=amd64 go build -a -tags netgo -ldflags '-w' -o server2 *.go

```
/work #
/work #
debian@debiandocker:~/dockerdemos/go$ ls -alh
total 11M
drwxr-xr-x 2 debian debian 4.0K Oct  3 18:48 .
drwxr-xr-x 3 debian debian 4.0K Oct  3 18:30 ..
-rw-r--r-- 1 debian debian   47 Sep 26 21:06 Dockerfile
-rwxr-xr-x 1 root   root   6.2M Oct  3 18:47 server1
-rwxr-xr-x 1 root   root   4.8M Oct  3 18:47 server2
-rw-r--r-- 1 debian debian  229 Oct  3 18:36 server.go
debian@debiandocker:~/dockerdemos/go$ file server*
server1:   ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter /lib/ld-musl-x86_64.so.1
, Go BuildID=xXJREcQntqDhuGE5ZEgM/MxEYLorKJVlyER5ogqgC/iaUUC2xnTQUfY-YU02YT/sgaPNPLX-GThuFhxkEhb, not stripped
server2:   ELF 64-bit LSB executable, x86-64, version 1 (SYSV), statically linked, Go BuildID=Lep-PB8PnQ8WE-lTX01t/Xnu1l
Ry0s0OJ9RaXzVwo/et6NtFn_mkRcTfEdhJAU/sBzj_AMlSaEDFE1xq3Ch, not stripped
server.go: C source, ASCII text
debian@debiandocker:~/dockerdemos/go$ ldd *
Dockerfile:
        not a dynamic executable
server1:
        linux-vdso.so.1 (0x00007ffd3e746000)
        libc.musl-x86_64.so.1 => not found
server2:
        not a dynamic executable
server.go:
        not a dynamic executable
debian@debiandocker:~/dockerdemos/go$
```

howest
university of applied sciences

Executable vs Process

High level overview

Process vs Program

- Program/Binary/Executable (/application/app):
  - passive
  - has no state
  - program code + initialized data
  - (= is not running!)
- Process:
  - active
  - has a state
  - program code + program counter + stack + data section + heap + …

→ *We will look more in depth in future lectures*

If we view memory as a big array, the regions (or ``segments") look as follows:

```
|--------------| 0
|              |
|   void       |
|              |
|--------------| 0x10000
|              |
|  code        |
|              |
|--------------|
|  void        |
|--------------| 0x20000
|              |
|  globals     |
|              |
|--------------|
|              |
|  heap        |
|              |
||||||||||||||||
|vvvvvvvvvvvvvv|
|              |
|              |
|  void        |
|              |
|              |
|^^^^^^^^^^^^^^|
||||||||||||||||
|              |
|  stack       |
|              | 0xefffffff
|--------------|
```

Stack & Heap (more later)

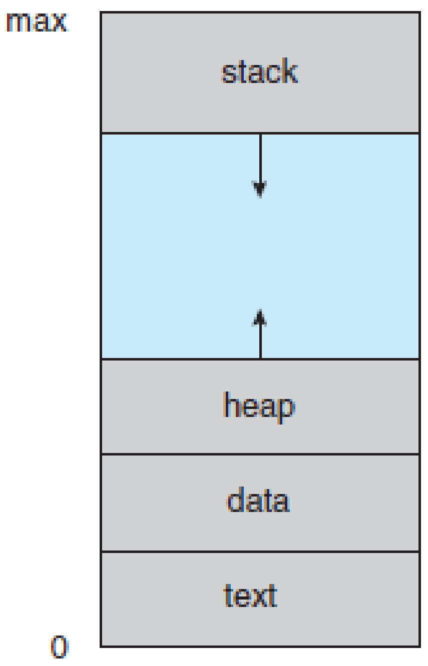
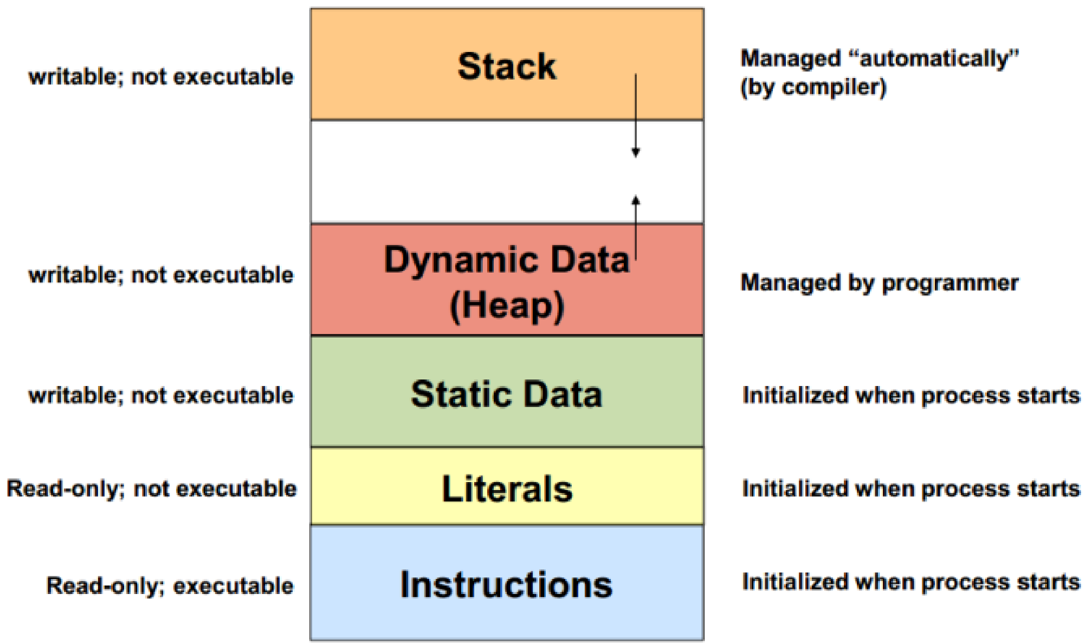


Figure 3.1 Process in memory.



Process states

As a process executes, the **state** changes:

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution

Process states

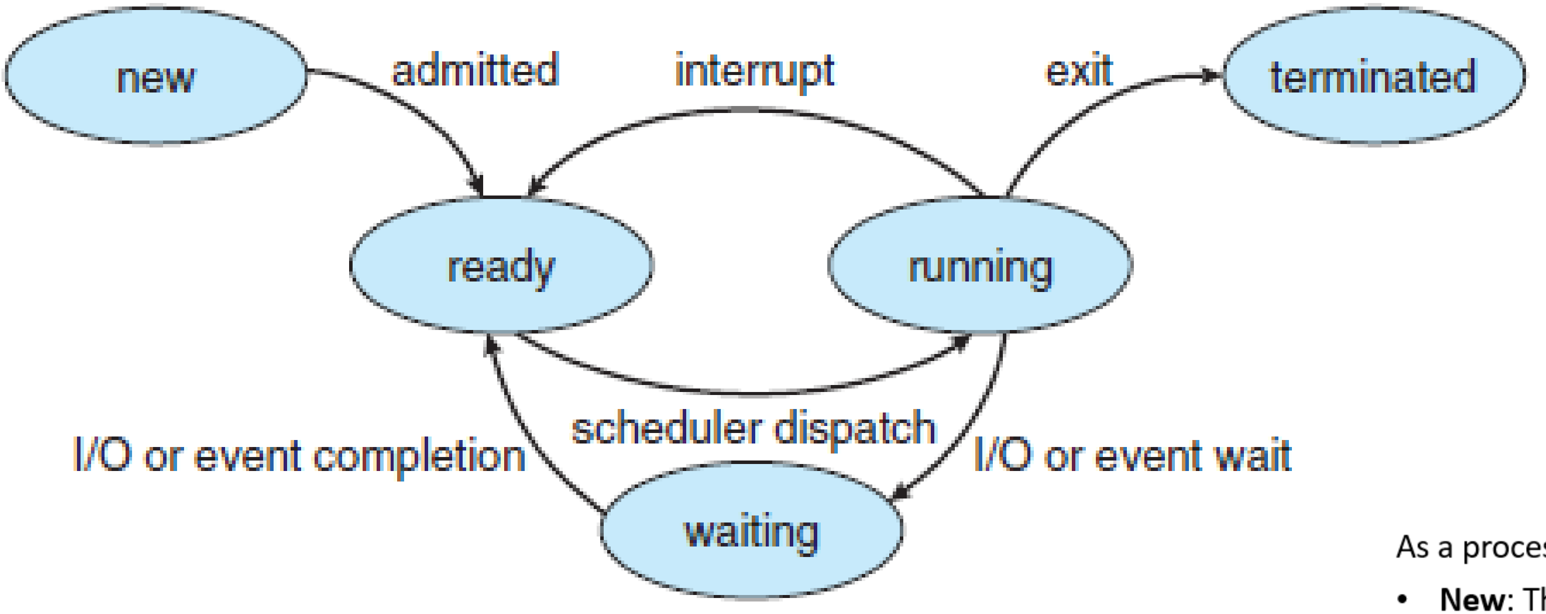


Figure 3.2 Diagram of process state.

As a process executes, the **state** changes:

- **New**: The process is being created
- **Running**: Instructions are being executed
- **Waiting**: The process is waiting for some event to occur
- **Ready**: The process is waiting to be assigned to a processor
- **Terminated**: The process has finished execution

Investigating memory

- We will use a debugger in later lectures
- Memzoom: https://justine.lol/memzoom/index.html
- /proc/<pid> on Linux
- Cheat Engine is actually a memory debugger ☺
  - If you download it, make sure to use the official downloads!
  - Not needed in this course for our samples

memzoom


ELF
/lib64/ld-linux-x86-64.so.2
GNU
__gmon_start__
55ead6620000
55ead6620040
55ead6620080
55ead66200c0
55ead6620100
55ead6620140
55ead6620180
55ead66201c0
55ead6620200
55ead6620240
55ead6620280
55ead66202c0
55ead6620300
55ead6620340
55ead6620380
55ead66203c0
55ead6620400
55ead6620440
55ead6620480
55ead66204c0
55ead6620500

Recover an executable from memory

```
debian@debian:~/memzoom$ md5sum hello
fa73213b1360c722241a0bbd3d25795b  hello
debian@debian:~/memzoom$ ./hello &
[2] 1620
debian@debian:~/memzoom$ Hello

debian@debian:~/memzoom$ rm hello
debian@debian:~/memzoom$ xxd /proc/1620/exe > hello-memory.hex
debian@debian:~/memzoom$ ls -l /proc/1620/exe
lrwxrwxrwx 1 debian debian 0 Oct  1 17:40 /proc/1620/exe -> '/home/debian/memzoom/hello (deleted)'
debian@debian:~/memzoom$ xxd -r hello-memory.hex > hello-memory
debian@debian:~/memzoom$ chmod +x hello-memory
debian@debian:~/memzoom$ ./hello-memory
Hello

^C
debian@debian:~/memzoom$ md5sum hello-memory
fa73213b1360c722241a0bbd3d25795b  hello-memory
debian@debian:~/memzoom$
```

Process control block

The information associated with a process is called a **Process Control Block** (=PCB, also called **task control block**).

- Process state: running, waiting,…
- Program counter: location of instruction to execute next
- CPU registers
- CPU scheduling information
- Memory-management information
- Accounting information
- I/O status and related devices
- List of open files

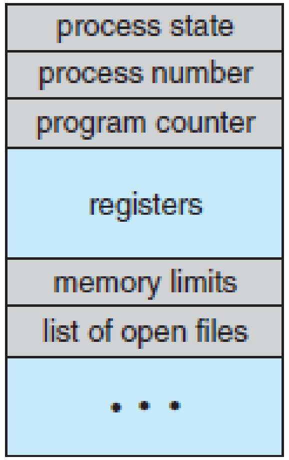


Figure 3.3 Process control block (PCB).

Process control block as a "state saver"

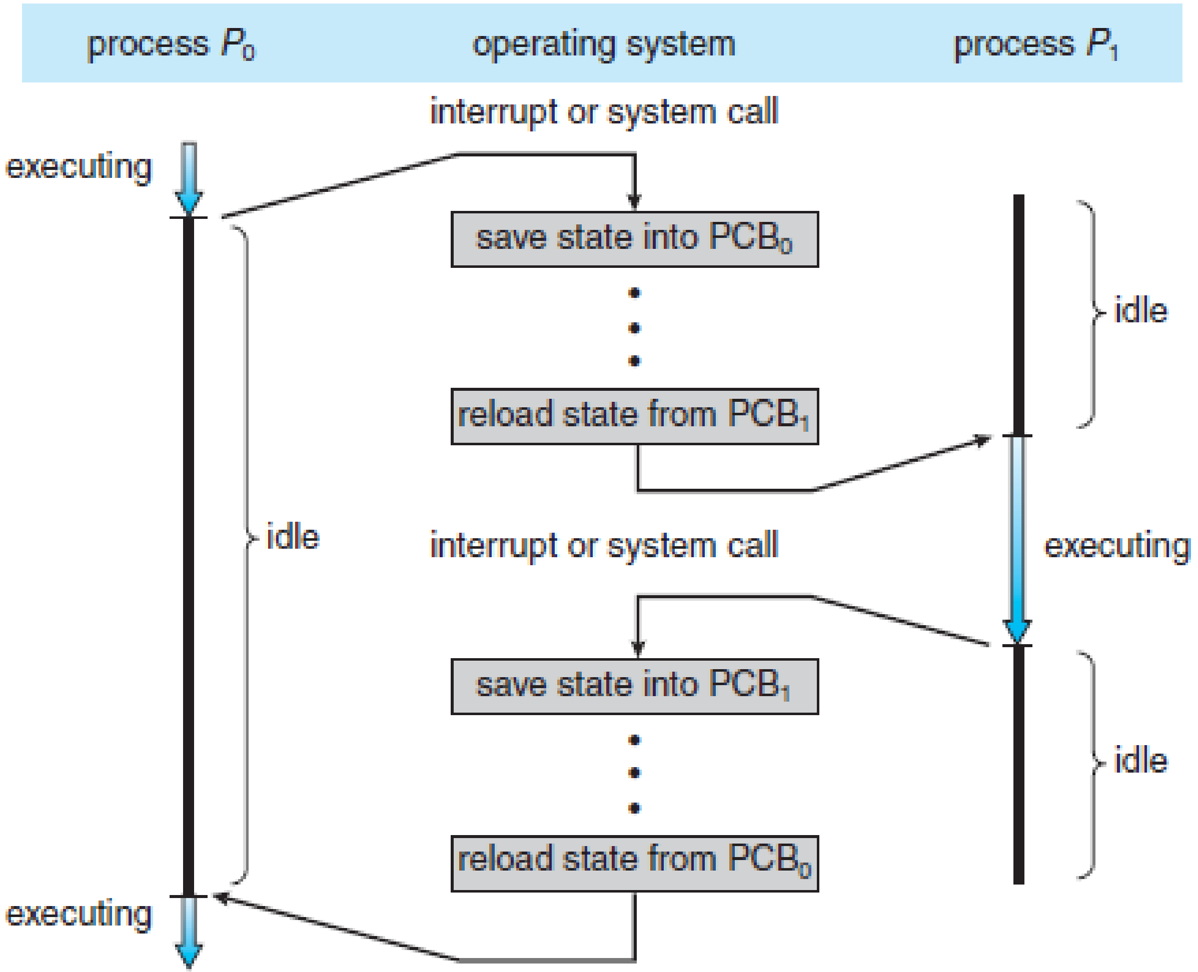


Figure 3.4 Diagram showing CPU switch from process to process.

Context Switch

= Saving/storing the state of a process (or actually thread)

As a result a **single CPU** can **allow multiple processes** -> multitasking operating system!

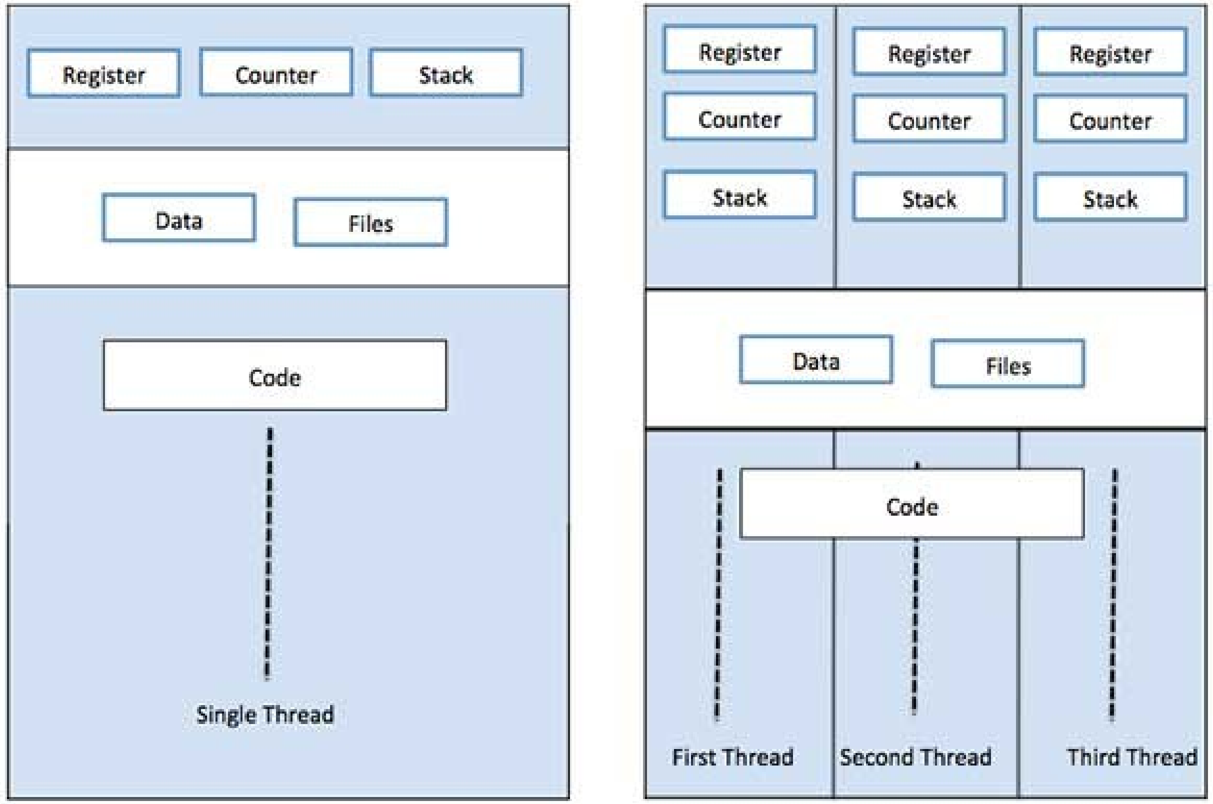
When a context switch occurs depends on multiple things:

- Type of OS
- One or more register set
- Sometimes it is needed to go from user mode to kernel mode
- Sometimes as a result of interrupts
- …

Thread

Often called a “**lightweight process**”

→ Minimize **context switching** time
→ A “blocking” thread does not block other threads



Single Process P with single thread

Single Process P with three threads

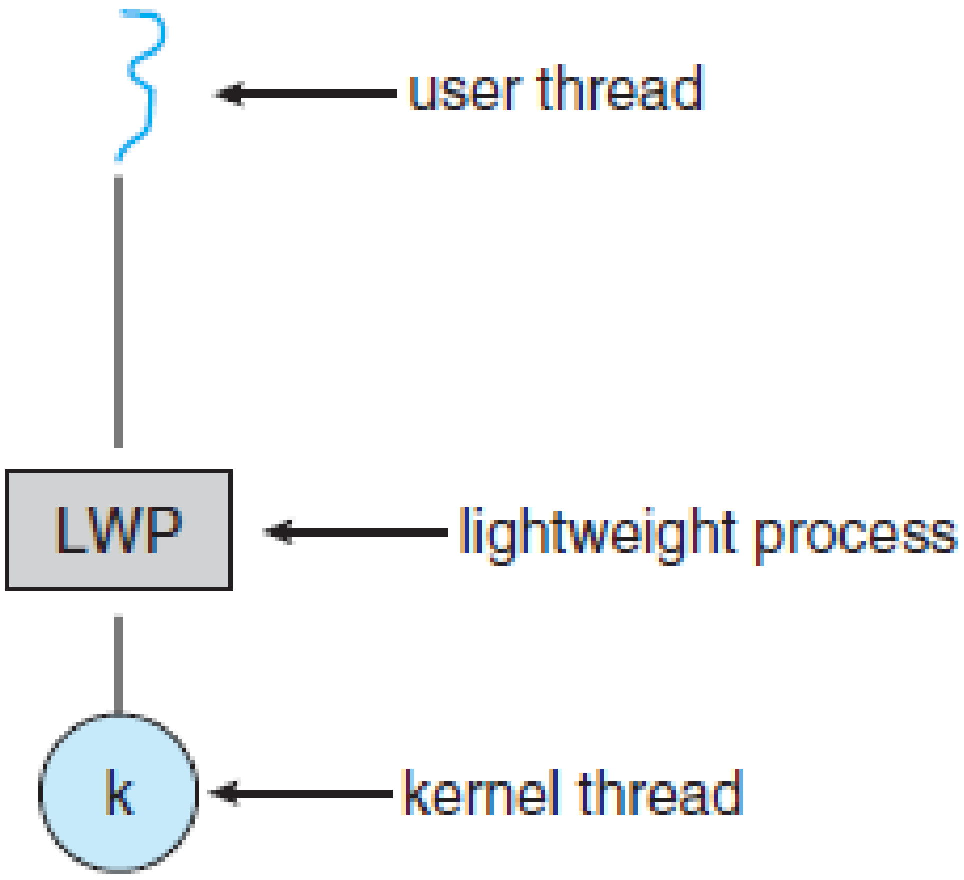


Figure 4.13 Lightweight process (LWP).

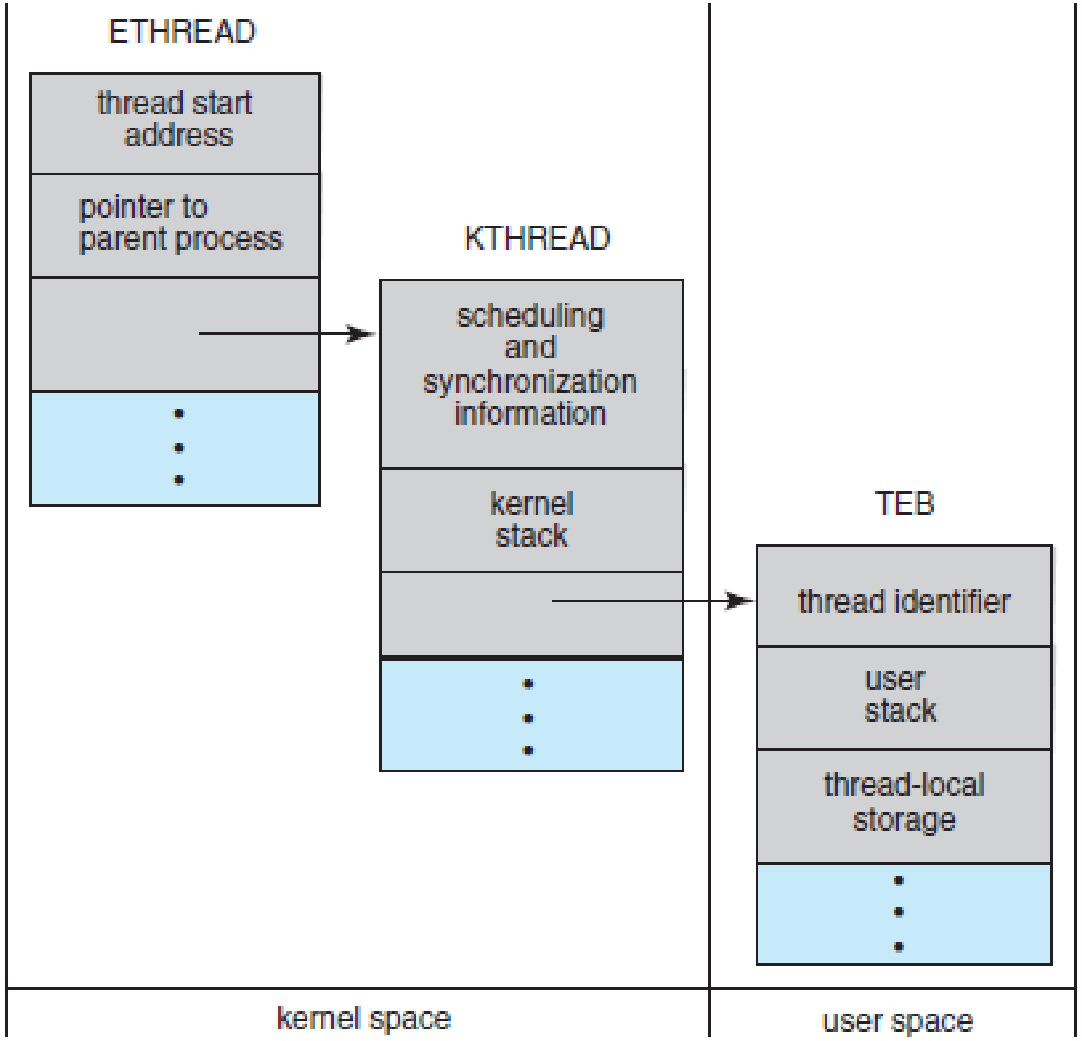
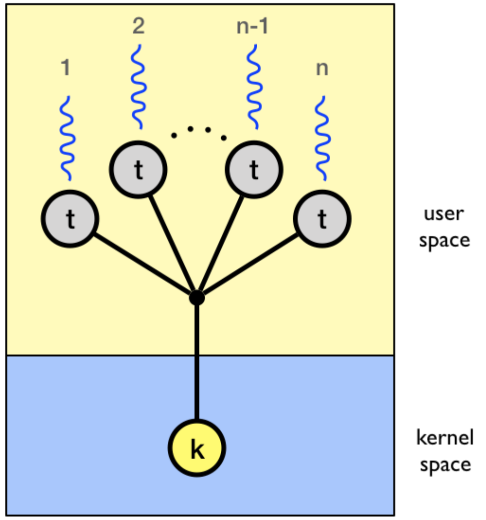
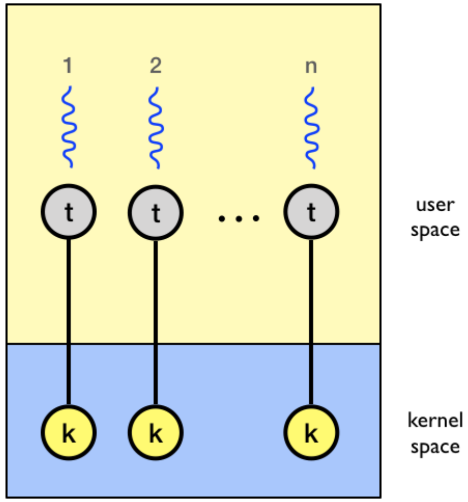


Figure 4.14 Data structures of a Windows thread.

User- level vs kernel-level threads





User-level threads

All **code** and **data structures** for the library exist in user space.

Invoking a function in the API results in a **local function call** in user **space** and not a system call.

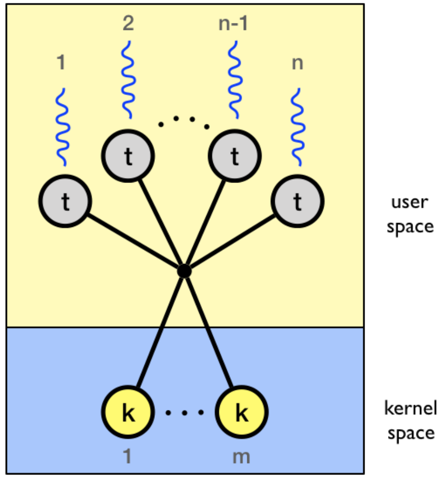
user mode

Kernel-level threads

All **code** and **data structures** for the library exists in **kernel space**.

Invoking a function in the API typically results in a **system call** to the kernel.

kernel mode



howest university of applied sciences

Linux example

Some interesting things:

- htop
    - nlwp via f2 setup
- ps -e -T
- /proc/<pid-nr>/status
- pstree -p

```
                 0.0%]   Tasks: 26, 25 thr; 1 running
           226M/1.92G]   Load average: 0.04 0.02 0.00
             0K/2.00G]   Uptime: 05:55:03

M%   TIME+  NLWP Command
.5  0:03.28    1 /sbin/init
.8  0:00.02    3 ├─ /usr/lib/packagekit/packagekitd
.8  0:00.00    3 │  ├─ gdbus
.8  0:00.00    3 │  └─ gmain
.1  0:00.00    1 ├─ nginx: master process /usr/sbin/nginx -g daemon on; master_process on;
.3  0:00.37    1 │  └─ nginx: worker process
.4  0:00.12    3 ├─ /usr/lib/policykit-1/polkitd --no-debug
.4  0:00.07    3 │  ├─ gdbus
.4  0:00.00    3 │  └─ gmain
.4  0:00.03    1 ├─ /lib/systemd/systemd --user
.1  0:00.00    1 │  └─ (sd-pam)
.3  0:03.06    8 ├─ /usr/bin/dockerd -H fd:// --containerd=/run/containerd/containerd.sock
```

```
Name:   containerd
Umask:  0022
State:  S (sleeping)
Tgid:   412
Ngid:   0
Pid:    412
PPid:   1
TracerPid:      0
Uid:    0       0       0       0
Gid:    0       0       0       0
FDSize: 128
Groups:
NStgid: 412
NSpid:  412
NSpgid: 412
NSsid:  412
VmPeak:   811796 kB
VmSize:   747668 kB
VmLck:         0 kB
VmPin:         0 kB
VmHWM:     50120 kB
VmRSS:     50120 kB
RssAnon:           25160 kB
RssFile:           24960 kB
RssShmem:              0 kB
VmData:   170056 kB
VmStk:       132 kB
VmExe:     17172 kB
VmLib:         0 kB
VmPTE:       252 kB
VmSwap:        0 kB
HugetlbPages:          0 kB
CoreDumping:    0
Threads:        9
SigQ:   0/7767
SigPnd: 0000000000000000
ShdPnd: 0000000000000000
SigBlk: fffffffe3bfa2800
SigIgn: 0000000000000000
/proc/412/status
```

Thread example

File: printthread.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    sleep(200);
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");

    exit(0);
}
```

936	943	1	943	debian	20	0	8836	5576	3820	S	0.0	0.1	0:00.14	└─ -bash
943	1379	2	1379	debian	20	0	10648	540	464	S	0.0	0.0	0:00.00	└─ ./printthread
943	1379	2	1380	debian	20	0	10648	540	464	S	0.0	0.0	0:00.00	└─ printthread

```
debian@debian:~/threads$ gcc printthread.c -lpthread -o printthread
```

Java example

File: HelloWorld.java

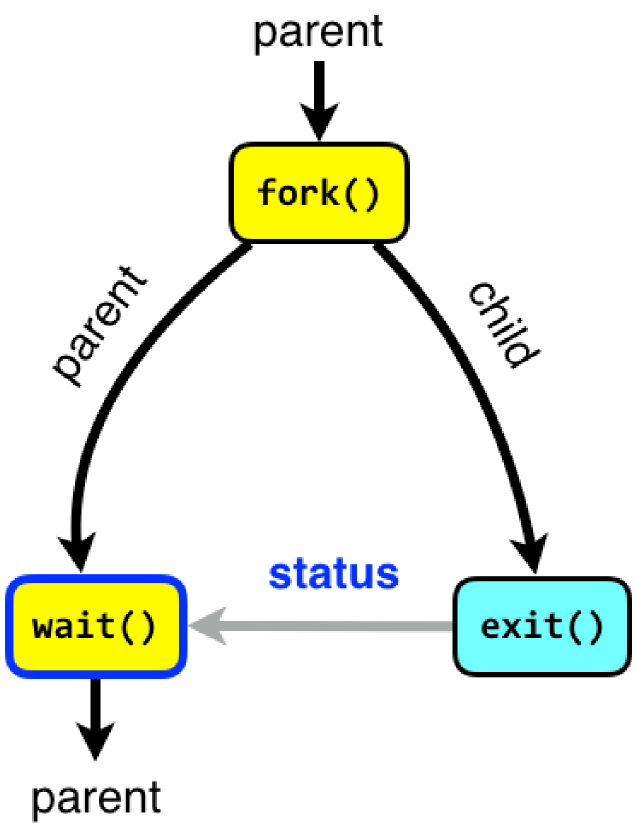
```java
import java.util.*;

public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
        Scanner sc= new Scanner(System.in);
        System.out.print("Enter a string: ");
        String str= sc.nextLine();
    }
}
```

```
 936   943   1   943 debian  20  0  8820  5800  3884 S 0.0 0.1 0:00.18    └─ -bash
 943  1486  12  1486 debian  20  0 2957M 40664 25396 S 0.7 1.0 0:00.13       └─ java HelloWorld
 943  1486  12  1487 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.04          ├─ java
 943  1486  12  1488 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.00          ├─ VM Thread
 943  1486  12  1489 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.00          ├─ Reference Handl
 943  1486  12  1490 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.00          ├─ Finalizer
 943  1486  12  1491 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.00          ├─ Signal Dispatch
 943  1486  12  1492 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.00          ├─ Service Thread
 943  1486  12  1493 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.01          ├─ C2 CompilerThre
 943  1486  12  1494 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.02          ├─ C1 CompilerThre
 943  1486  12  1495 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.00          ├─ Sweeper thread
 943  1486  12  1496 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.03          ├─ VM Periodic Tas
 943  1486  12  1497 debian  20  0 2957M 40664 25396 S 0.0 1.0 0:00.00          └─ Common-Cleaner
```

Fork

- Fork -> System call that creates a new child **process**
  - If parent ends before child, the child becomes orphaned
- Pthread_create -> All "part of" the same process
  - There is no exit as virtual memory does not need clean-up



Fork example

How many times will “hello” be printed?

```
File: hello.c

#include <stdio.h>
#include <sys/types.h>
int main()
{
    fork();
    fork();
    fork();
    printf("hello\n");
    return 0;
}
```

```
File: 2fork.c

#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
void forkexample()
{
    // child process because return value zero
    if (fork() == 0) {
        printf("Hello from Child!\n");
        sleep(200);
    }
    // parent process because return value non-zero.
    else {
        printf("Hello from Parent!\n");
        sleep(200);
    }
}
int main()
{
    forkexample();
    return 0;
}
```

```
debian      937  0.0  0.1  8128 4900 pts/1   Ss   10:36   0:00  \_ -bash
debian      970  0.0  0.0  2304  560 pts/1   S+   10:38   0:00  |   \_ ./2fork
debian      971  0.0  0.0  2304   68 pts/1   S+   10:38   0:00  |       \_ ./2fork
```

Child processes vs Parent processes

Why is this important?

- What happens with the parent process if a child process get’s killed?
- **What happens with the child process if a parent process get’s killed?**
  - → Actually OS dependend

    Multiple options are possible:

    → Child process get’s killed as soon as parent get’s killed (behind the scenes “kill signals are send to the child process)

    → Child becomes an “orphan”

    → Child get’s to live on without any issues

    → Etc.

→ Might be interesting for “reverse shells” when hacking for example ☺

howest
university of applied sciences

User space vs Kernel space

The kernel

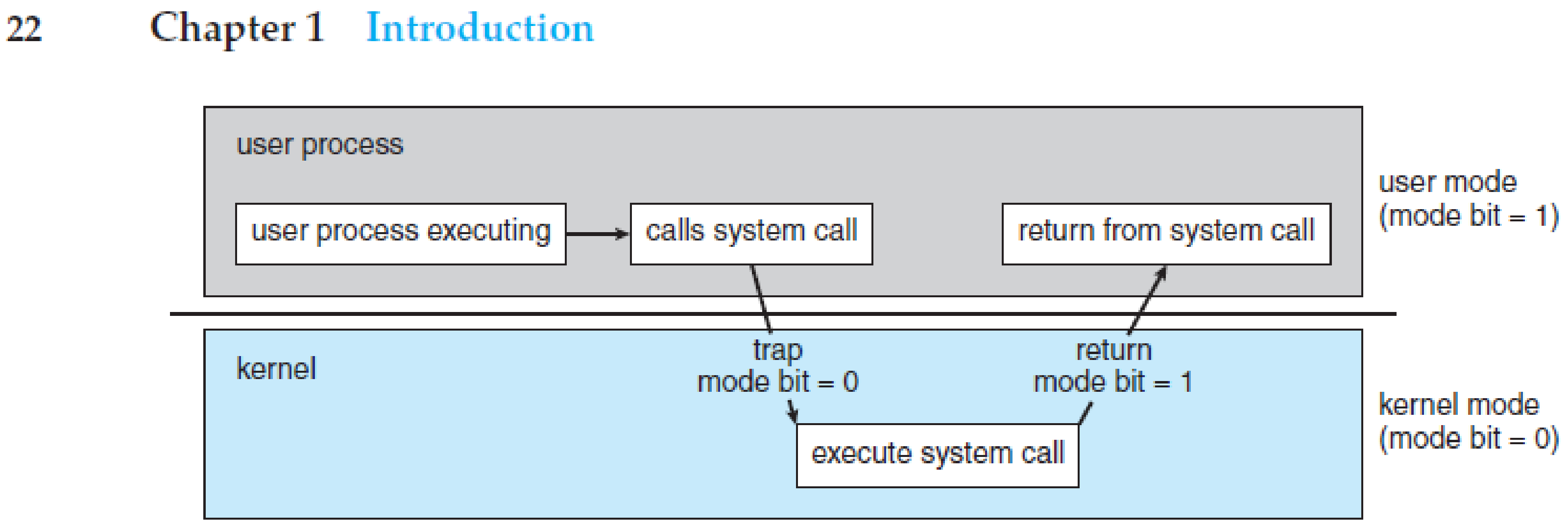


Figure 1.10 Transition from user to kernel mode.

Kernel space vs **User space!**

- Implemented with a **bit**
- **Kernel** has complete control and handles (almost) everything

Kernelspace = system mode - priviliged mode - supervisor mode - secure mode - unrestricted mode

Userspace = ordinary mode - user mode - restricted mode

System calls

2.3 System Calls 65

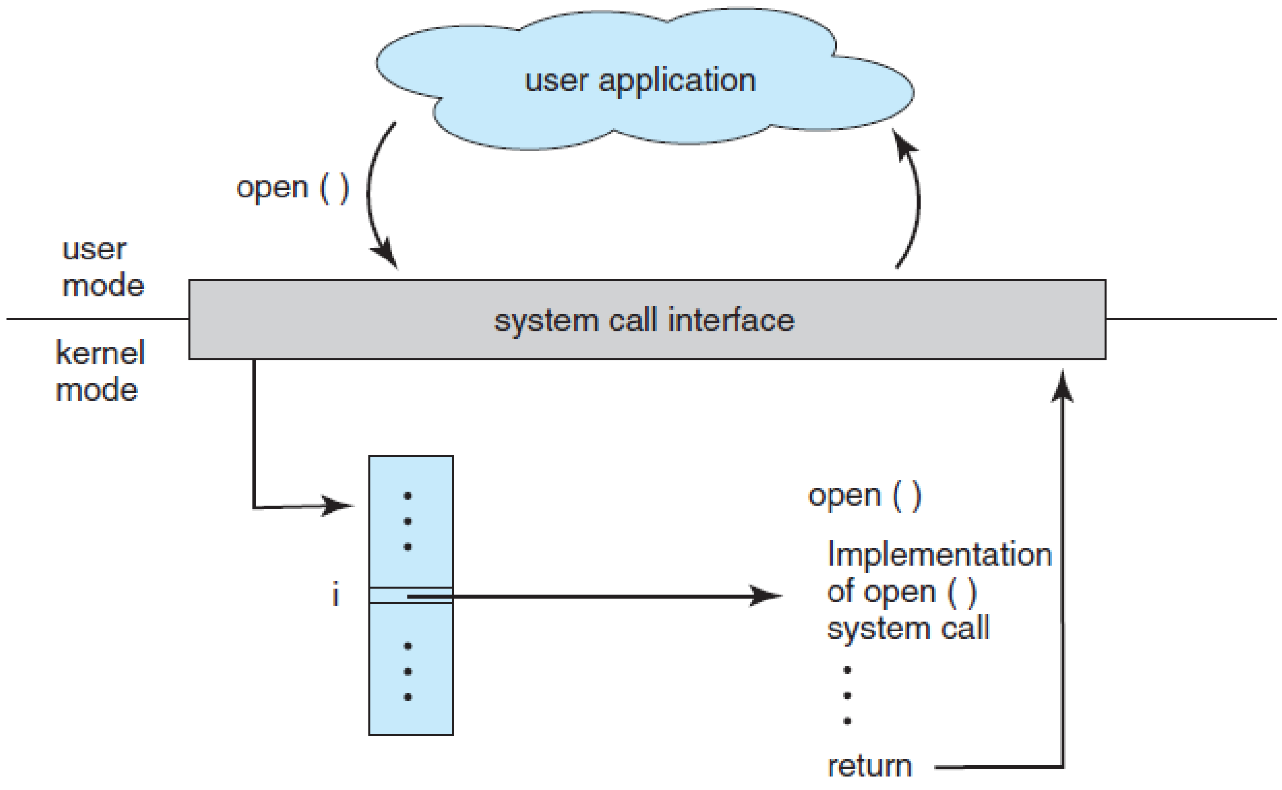
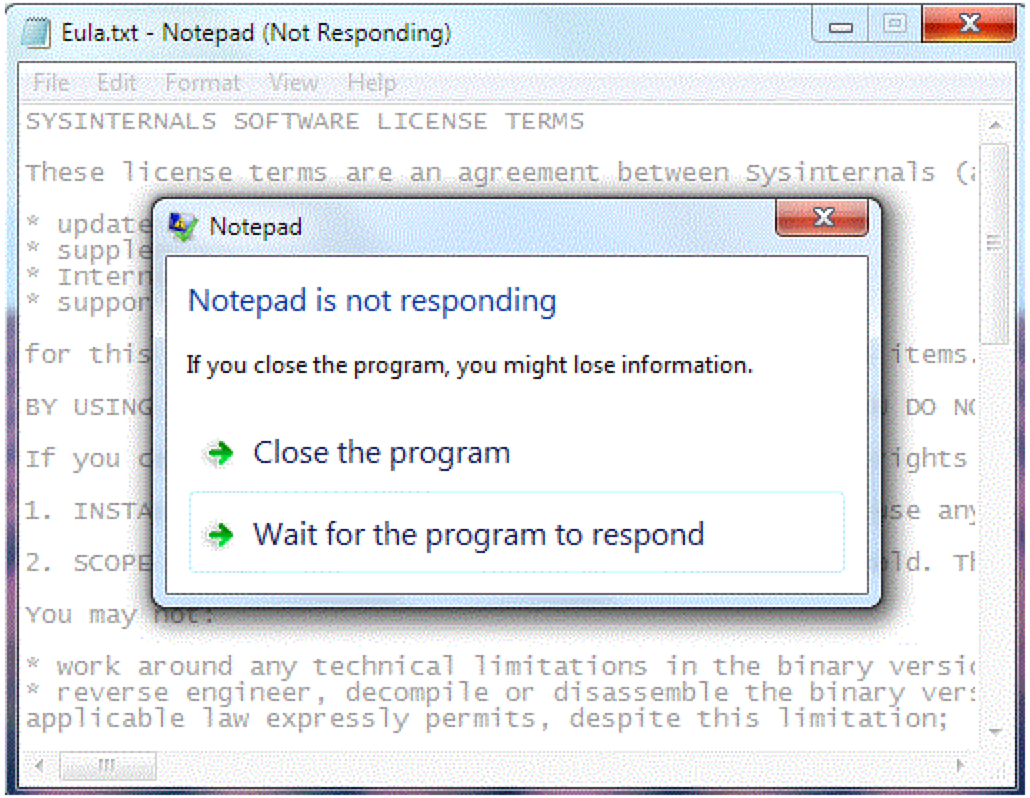


Figure 2.6 The handling of a user application invoking the `open()` system call.

A Windows crash in user space

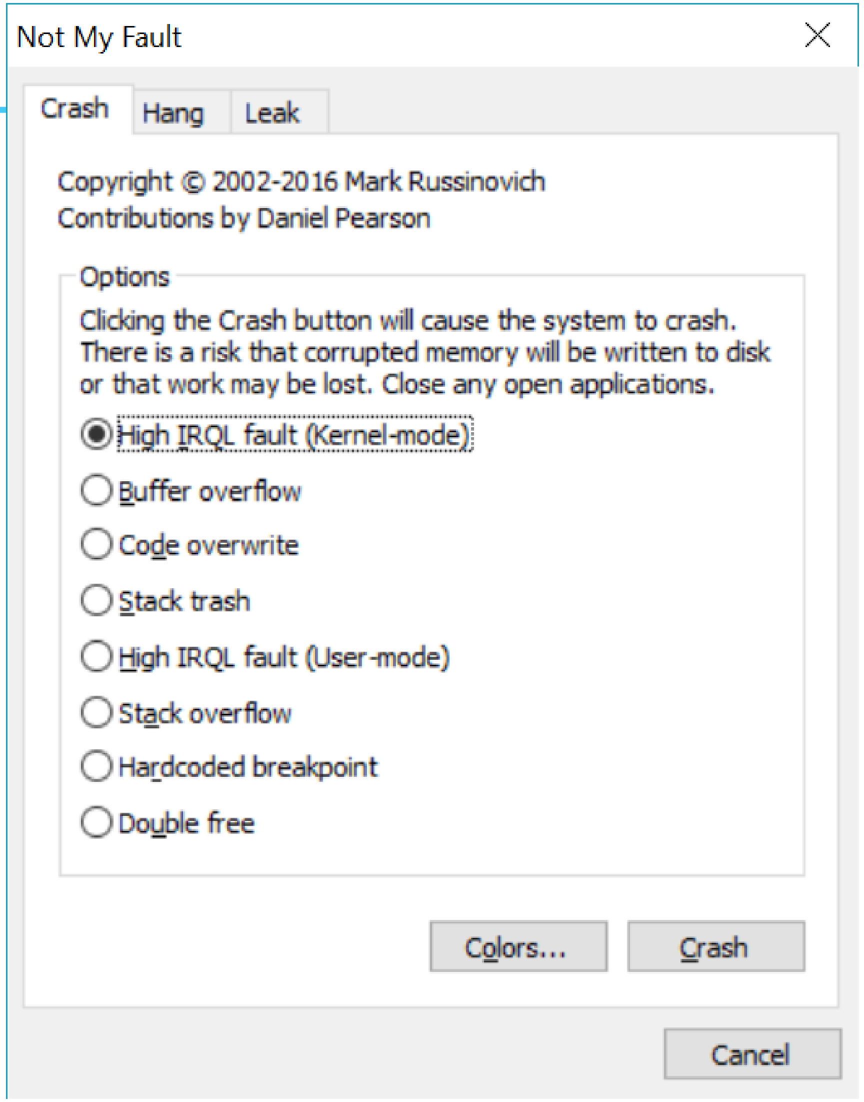


Notmyfault (Windows)

Remember user space vs. kernel space, and what happens when something goes wrong in the kernel?

“Notmyfault”-tool from sysinternals

https://docs.microsoft.com/en-us/sysinternals/downloads/notmyfault



Let’s get practical

On Linux

- We will build, compile and link C(++) software
- We will look at user space & kernel space
- Let’s create some C(++) programs
  - Let’s crash some programs (both in user and kernel space)
- 2 markdown files on leho containing (guided) lab instructions