Linux
Core Dump Analysis
Accelerated

Third Edition

Dmitry Vostokov
Software Diagnostics Services

Prerequisites

GDB Commands

We use these boxes to introduce GDB commands used in practice exercises

WinDbg Commands

We use these boxes to introduce WinDbg commands used in practice exercises

- Basic Linux troubleshooting
- Beneficial to know basics of assembly language (depends on your platform):

Foundations of Linux Debugging, Disassembling, and Reversing

Foundations of ARM64 Linux Debugging, Disassembling, and Reversing

Training Goals

- Review fundamentals
- Learn how to collect core dumps
- Learn how to analyze core dumps

Training Principles

- Talk only about what I can show
- Lots of pictures
- Lots of examples
- Original content

Schedule Summary

Day 1

- Analysis fundamentals (25 minutes)
- Process core dump collection (5 minutes)
- Basic x64 assembly language review (30 minutes)
- Process GDB core dump analysis (1 hour)

Day 2

- Process GDB core dump analysis (2 hours)

Day 3

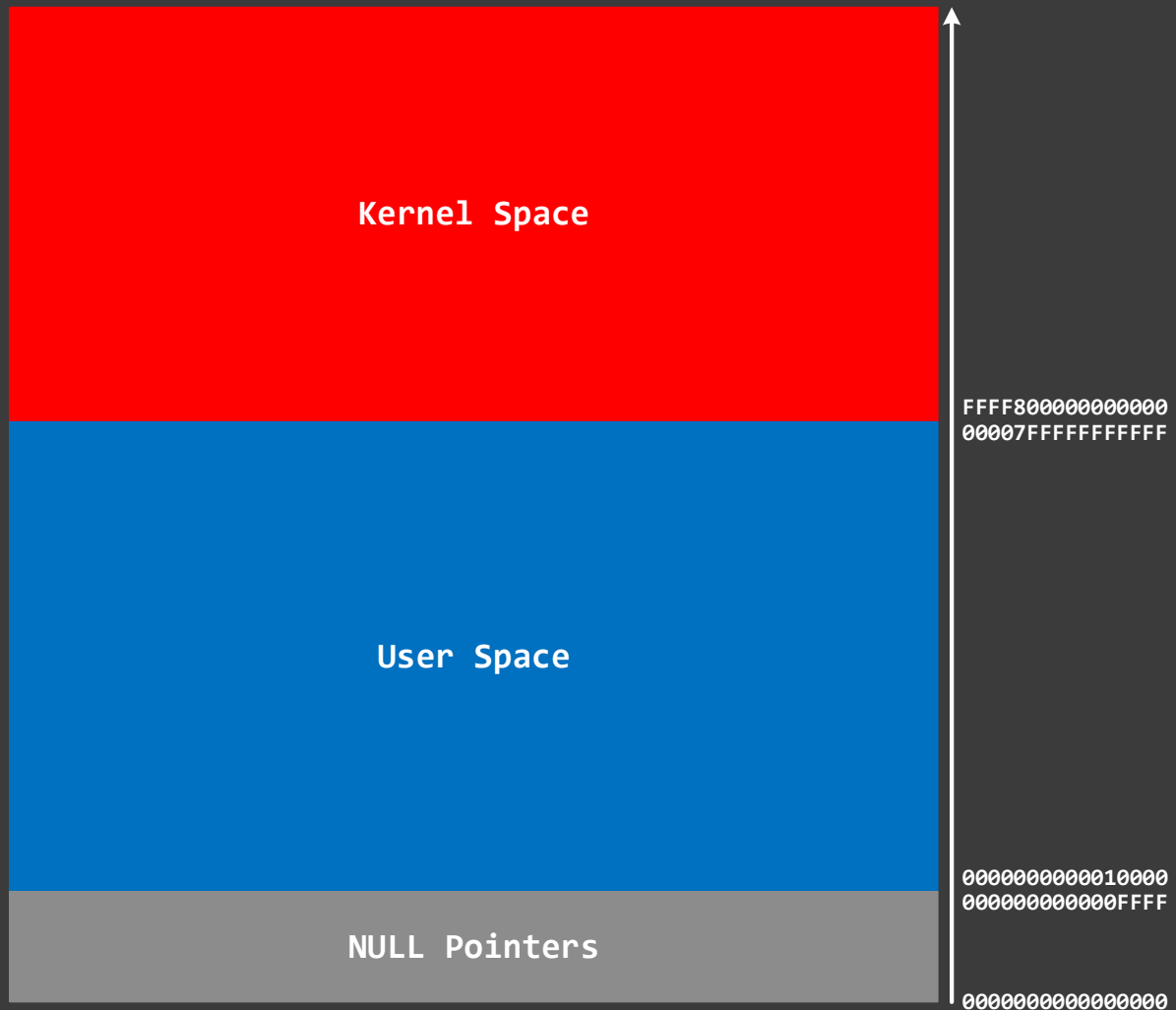
- Kernel core dump collection (5 minutes)
- Kernel core dump analysis (1 hour 55 minutes)

Day 4

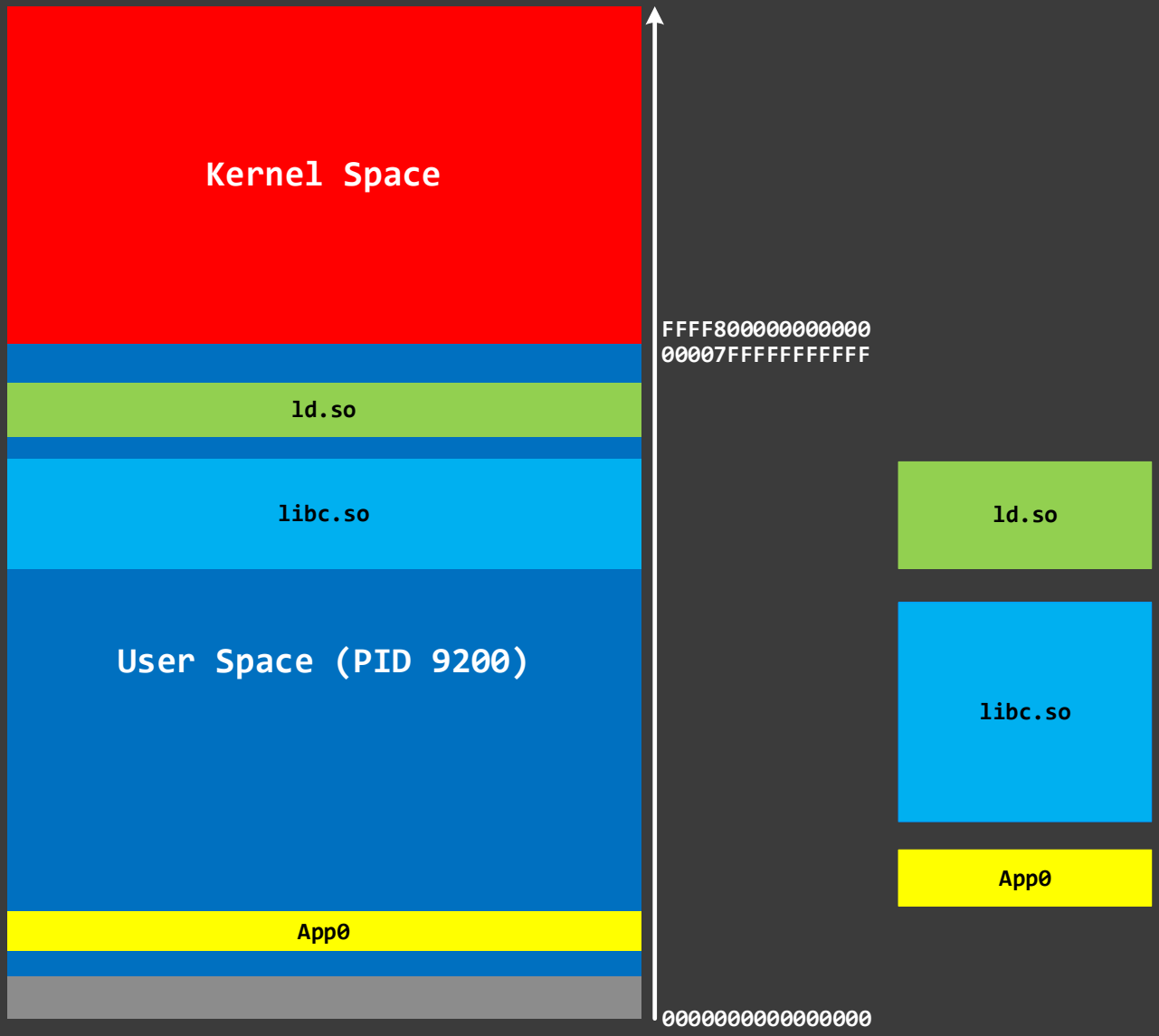
- Basic ARM64 assembly language review (30 minutes)
- Process GDB core dump analysis (1 hour 30 minutes)
- [Optional] Process WinDbg core dump analysis

Part 1: Fundamentals

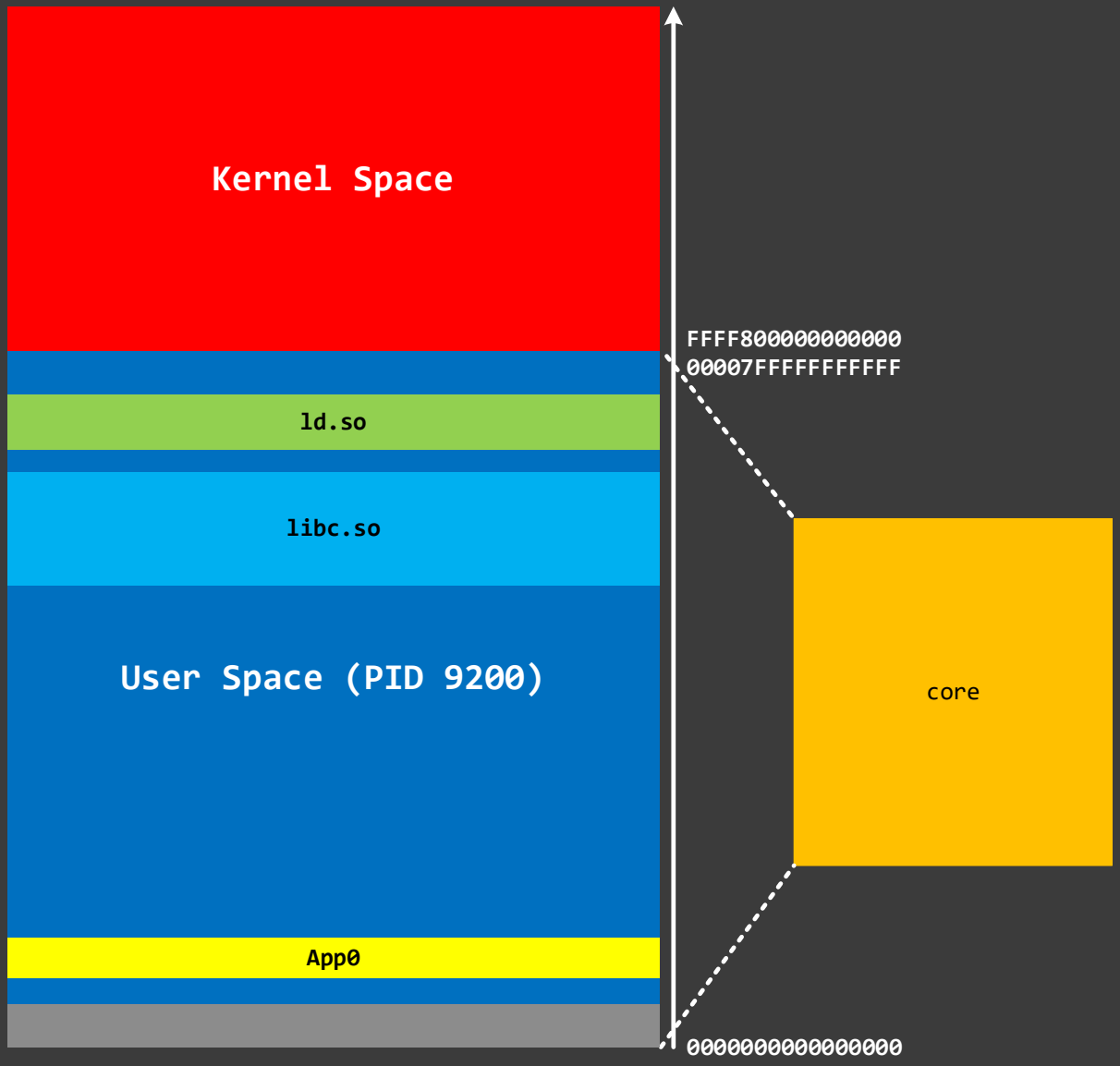
Memory/Kernel/User Space



App/Process/Library



Process Memory Dump



GDB Commands

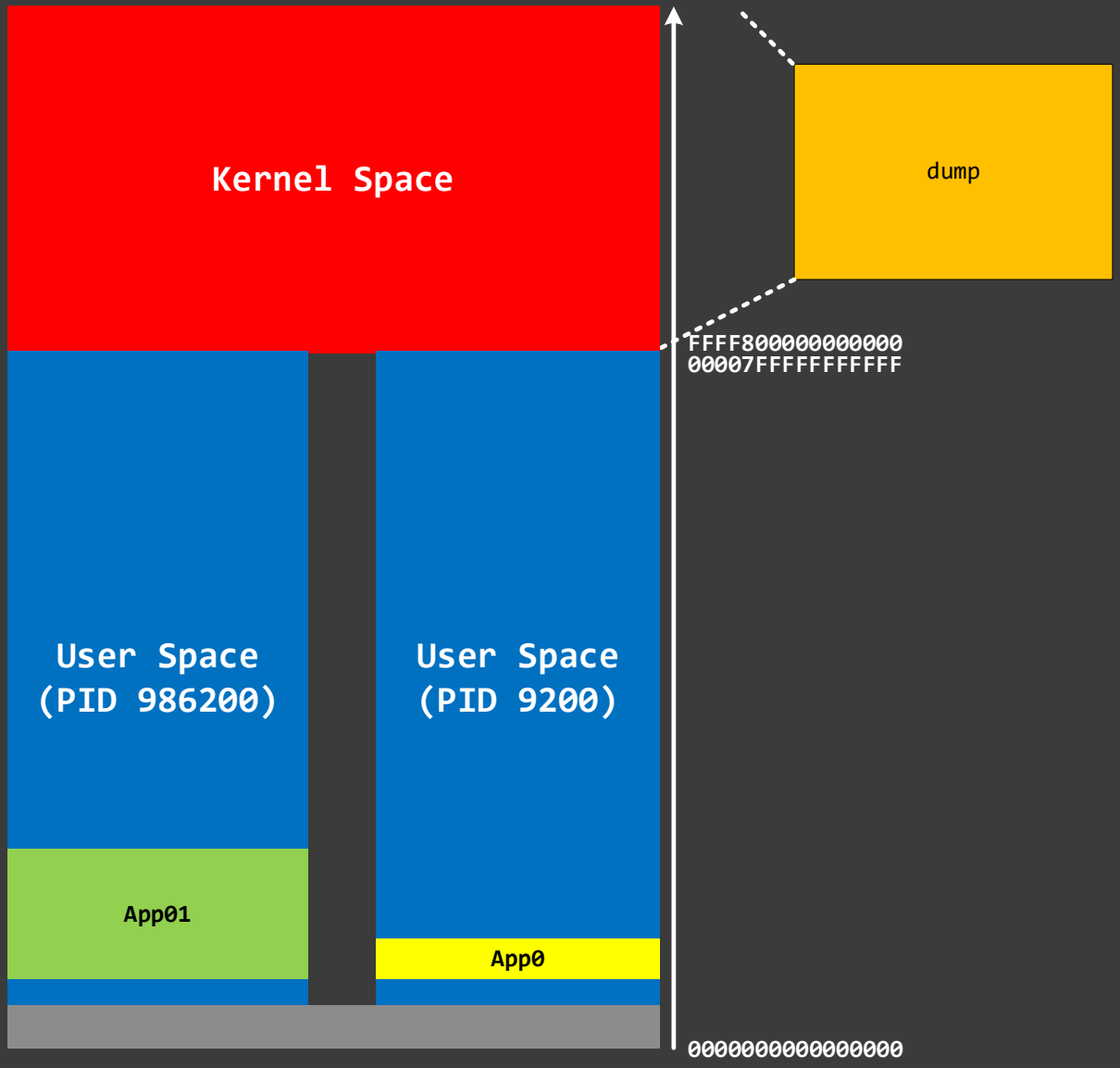
`info sharedlibrary`
Lists dynamic libraries

`maintenance info sections`
Lists memory regions

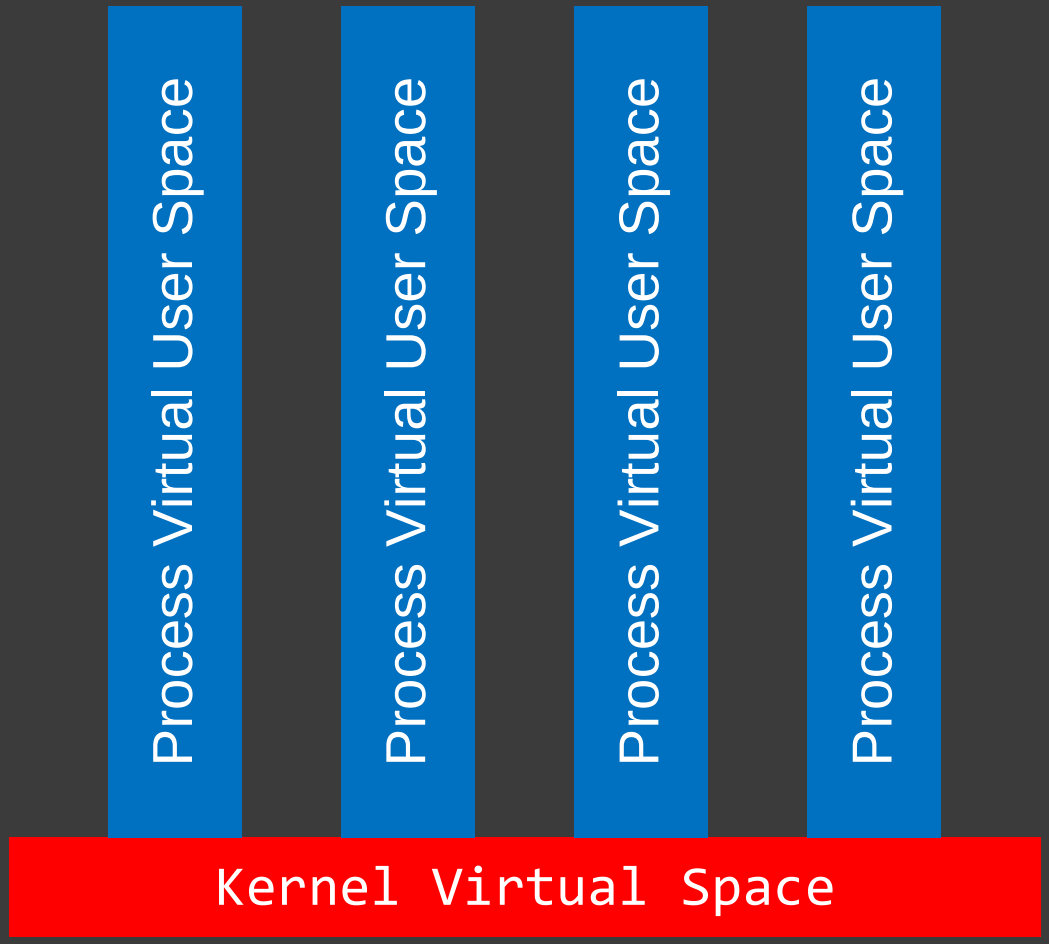
WinDbg Commands

`!address`
Lists memory regions

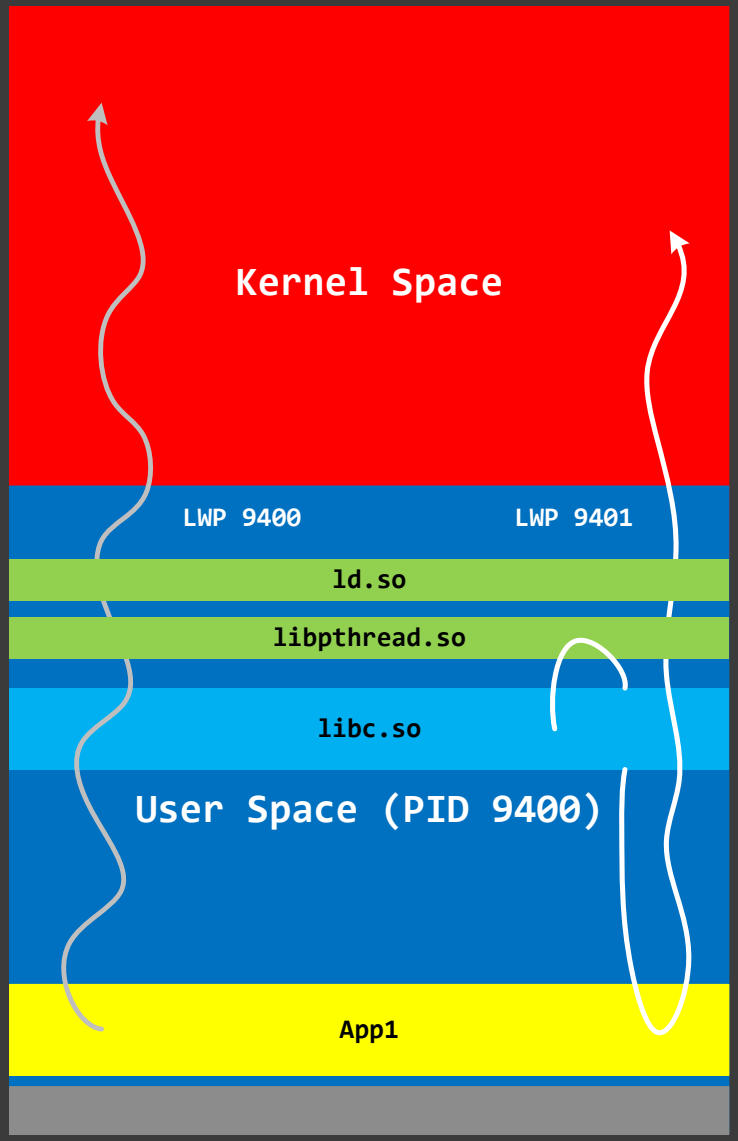
Kernel Memory Dump



Fiber Bindle Memory Dump



Lightweight Processes (Threads)



GDB Commands

`info threads`
Lists threads

`thread <n>`
Switches between threads

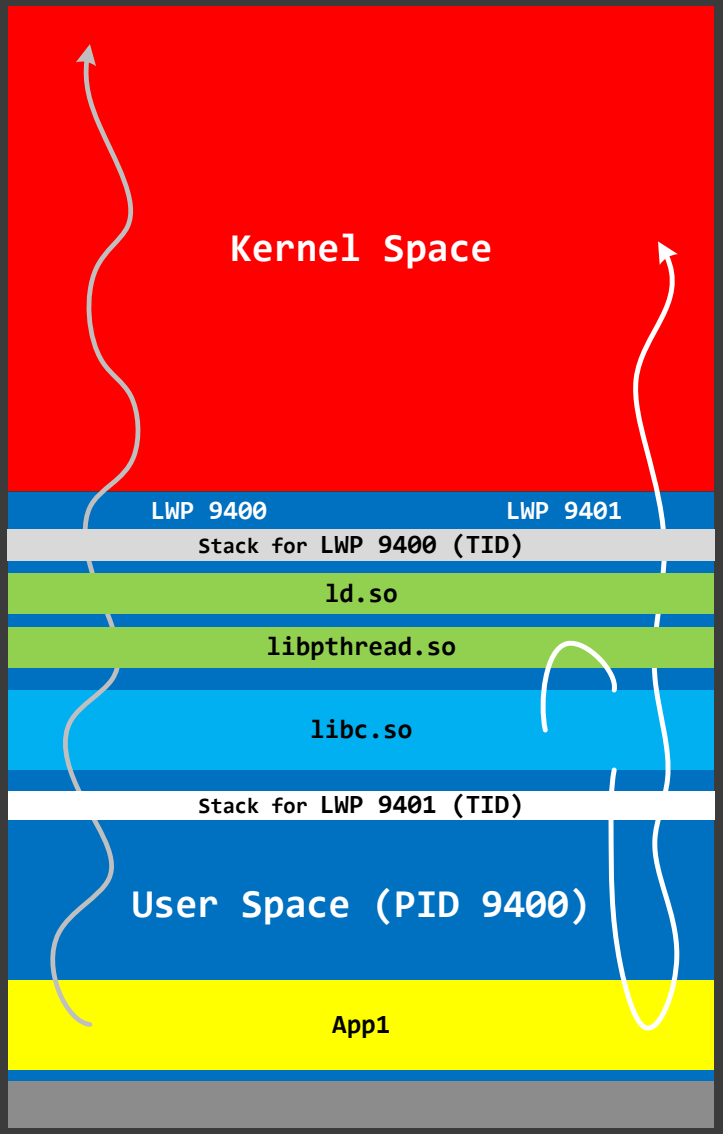
`thread apply all bt`
Lists stack traces from all threads

WinDbg Commands

`~*k`
Lists stack traces from all threads

`~<n>s`
Switches between threads

Thread Stack Raw Data



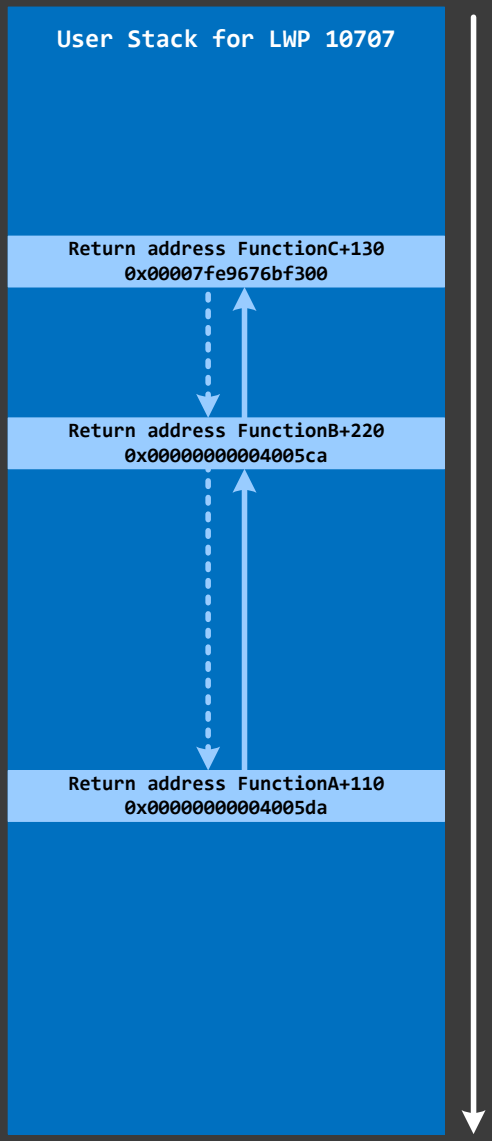
GDB Commands

x/<n>a <address>
Prints n addresses with corresponding symbol mappings if any

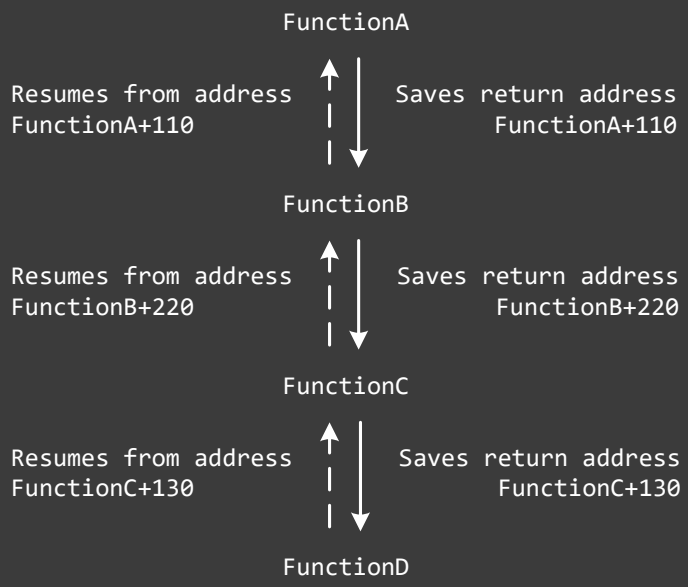
WinDbg Commands

dps <address> L<n>
Prints n addresses with corresponding symbol mappings if any

Thread Stack Trace




```
FunctionA()
{
  ...
  FunctionB();
  ...
}

FunctionB()
{
  ...
  FunctionC();
  ...
}

FunctionC()
{
  ...
  FunctionD();
  ...
}
```



GDB Commands

```
(gdb) bt
#0 0x00007fe9676bf48d in FunctionD ()
#1 0x00007fe9676bf300 in FunctionC ()
#2 0x00000000004005ca in FunctionB ()
#3 0x00000000004005da in FunctionA ()
```

GDB vs. WinDbg vs. LLDB

GDB Commands

```
(gdb) bt
#0 0x00007fe9676bf48d in FunctionD ()
#1 0x00007fe9676bf300 in FunctionC ()
#2 0x00000000004005ca in FunctionB ()
#3 0x00000000004005da in FunctionA ()
```

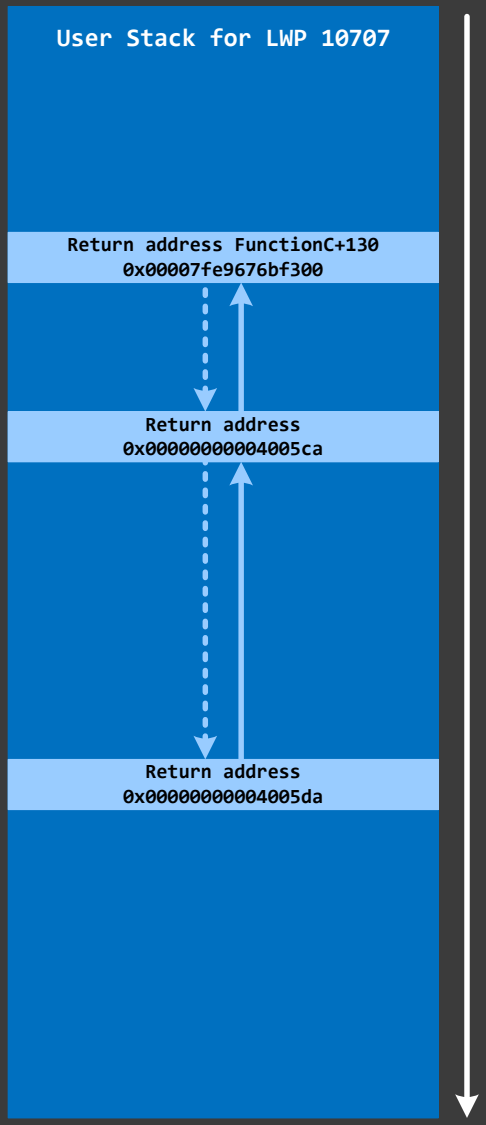
WinDbg Commands

```
0:000> k
00 00007fe9676bf300 Module!FunctionD+offset
01 00000000004005ca Module!FunctionC+130
02 00000000004005da AppA!FunctionB+220
03 0000000000000000 AppA!FunctionA+110
```

LLDB Commands

```
(lldb) bt
frame #0: 0x000000020328982a Module`FunctionD + offset
frame #1: 0x0000000203288a9c Module`FunctionC + 130
frame #2: 0x0000000104da3ea9 AppA`FunctionB + 220
frame #3: 0x0000000104da3edb AppA`FunctionA + 110
```

Thread Stack Trace (no symbols)



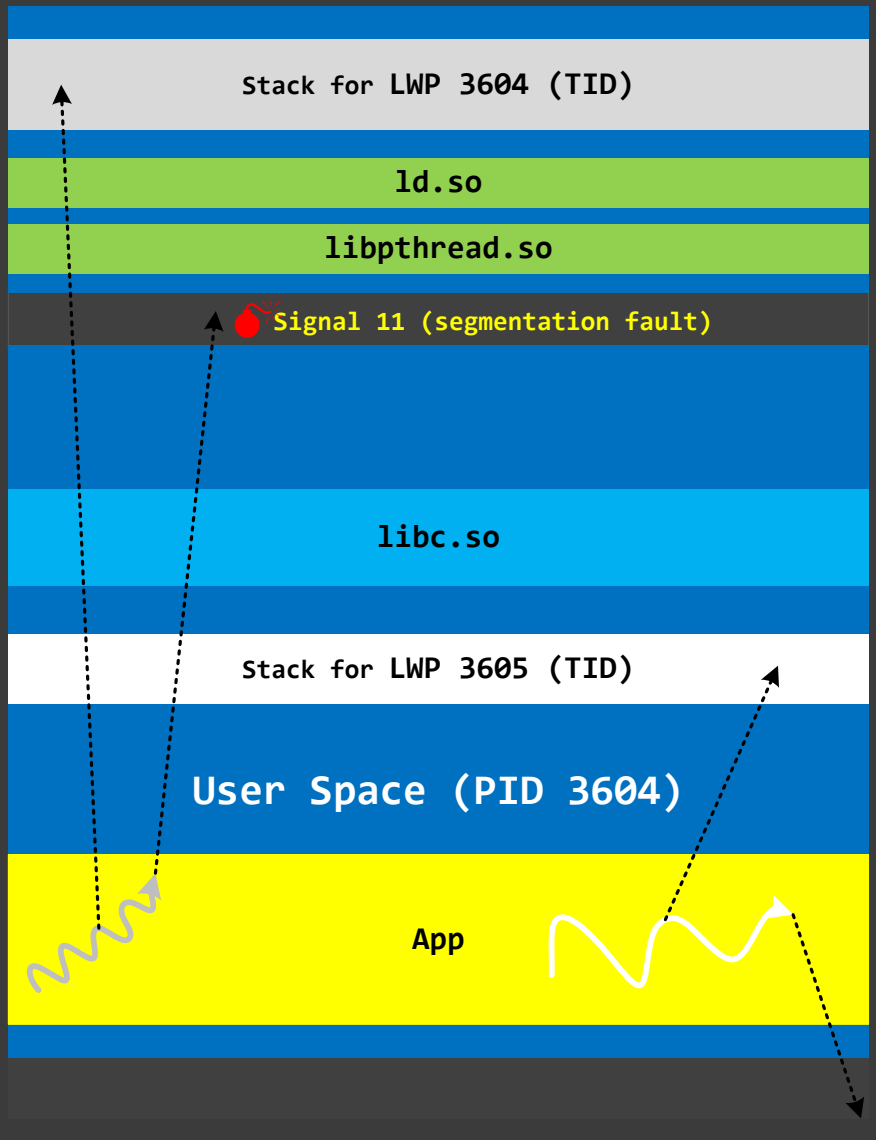
Symbol file App.sym

FunctionA 22000 - 23000
FunctionB 32000 - 33000

GDB Commands

```
(gdb) bt
#0 0x00007fe9676bf48d in FunctionD ()
#1 0x00007fe9676bf300 in FunctionC ()
#2 0x00000000004005ca in ?? ()
#3 0x00000000004005da in ?? ()
```

Exceptions (Access Violation)



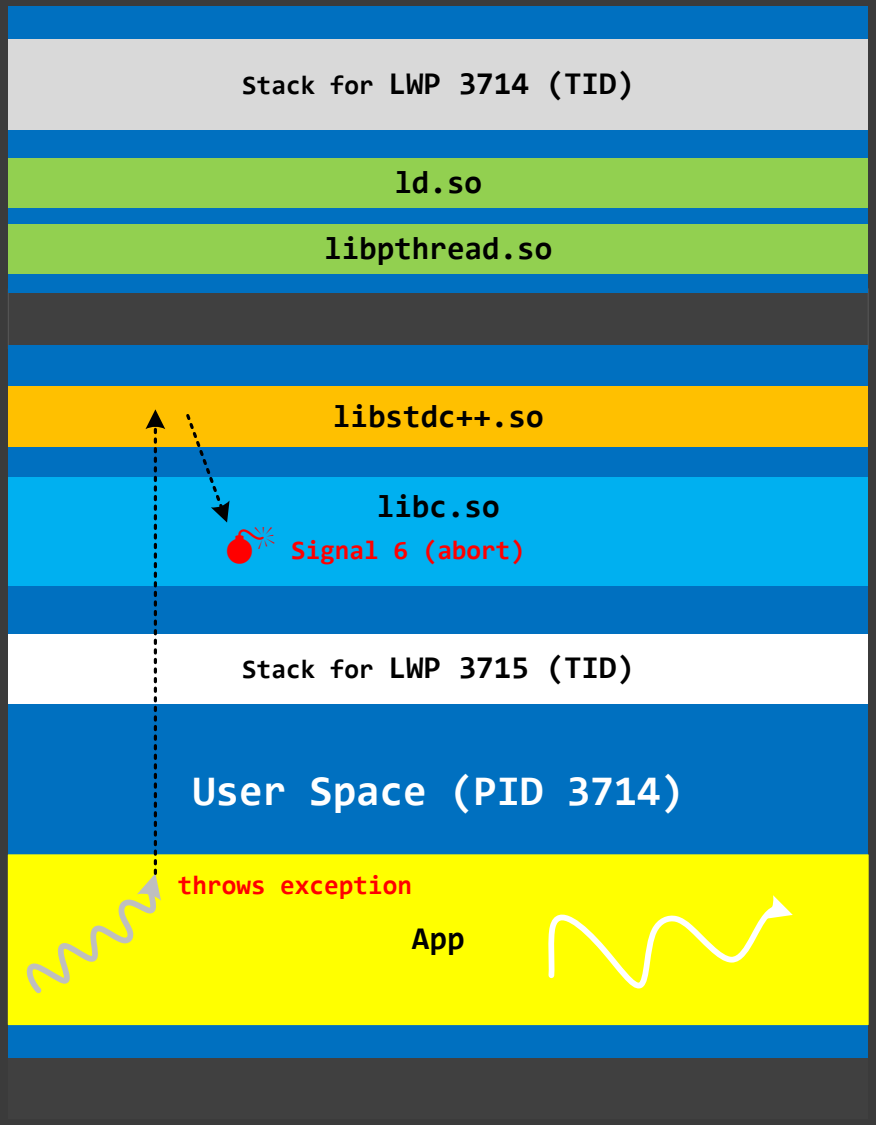
GDB Commands

```
(gdb) x <address>
0x<address>: Cannot access
memory at address 0x<address>
```

WinDbg Commands

```
0:000> dp <address> L1
<address>  ????????`????????
```

Exceptions (Runtime)



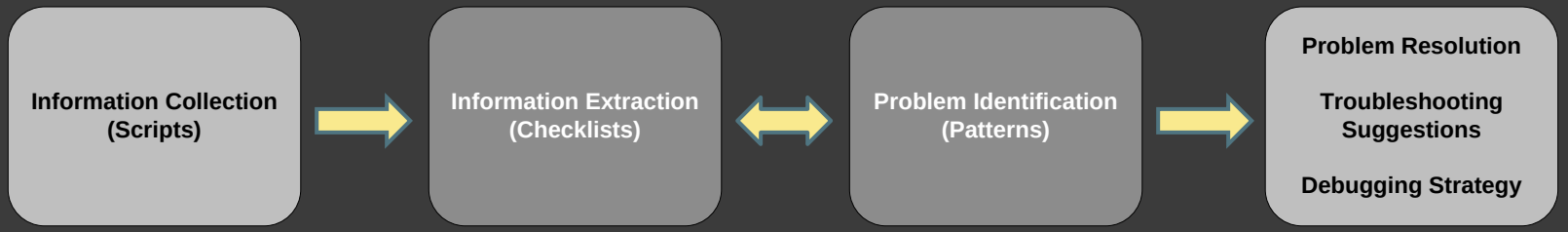
Pattern-Oriented Diagnostic Analysis

Diagnostic Pattern: a common recurrent identifiable problem together with a set of recommendations and possible solutions to apply in a specific context.

Diagnostic Problem: a set of indicators (symptoms, signs) describing a problem.

Diagnostic Analysis Pattern: a common recurrent analysis technique and method of diagnostic pattern identification in a specific context.

Diagnostics Pattern Language: common names of diagnostic and diagnostic analysis patterns. The same language for any operating system: Windows, macOS, Linux, ...



Part 2: Core Dump Collection

Enabling Collection (Processes)

- Temporary for the current user

```
$ ulimit -c unlimited
```

- Permanent for every user except root

Edit the file: `/etc/security/limits.conf`

Add or uncomment the line:

```
*       soft    core    unlimited
```

To limit root to 1GB, add or uncomment this line:

```
*       hard    core    1000000
```

Generation Methods (Processes)

- kill (requires ulimit)

```
$ kill -s SIGQUIT PID
$ kill -s SIGABRT PID
```

- gcore

```
$ gcore [-o filename] PID
```

- procdump

https://github.com/Sysinternals/ProcDump-for-Linux

Finding Core Dumps (Processes)

- Check the current core dump directory and naming pattern

```
$ cat /proc/sys/kernel/core_pattern
```

- Search

```
$ sudo find / -name core.*
```

- Further information

https://man7.org/linux/man-pages/man5/core.5.html

Enabling Collection (Kernel)

- Uncompressed kernel image with symbols:

Debian: `$ sudo apt install linux-image-$(uname -r)-dbg`

Ubuntu: https://wiki.ubuntu.com/Kernel/Systemtap (Where to get debug symbols for kernel X?)

- Kdump (and kexec):

```
$ sudo apt install kdump-tools kexec-tools
```

Generation Methods (Kernel)

- Manual

```
$ sudo echo 1 > /proc/sys/kernel/sysrq
$ sudo echo c > /proc/sysrq-trigger
```

- Kernel modules

Finding Core Dumps (Kernel)

- Core dumps

```
/var/crash
```

- vmlinux

```
/usr/lib/debug
```

Enabling Analysis (Kernel)

- Install crash tool (depends on distribution)

```
$ sudo apt install crash
```

- Compile crash tool from source

```
$ git clone https://github.com/crash-utility/crash.git
$ sudo apt install bison
$ cd crash
$ make
$ sudo make install
```

Part 3: x64 Disassembly

CPU Registers (x64)

- **RAX** ⊃ **EAX** ⊃ **AX** ⊇ {**AH**, **AL**}
- ALU: **RAX**, **RDX**
- Counter: **RCX**
- Memory copy: **RSI** (src), **RDI** (dst)
- Stack: **RSP**, **RBP**
- Next instruction: **RIP**
- New: **R8** – **R15**, **Rx**(**D**|**W**|**B**)

GDB Commands

```
info registers
```

WinDbg Commands

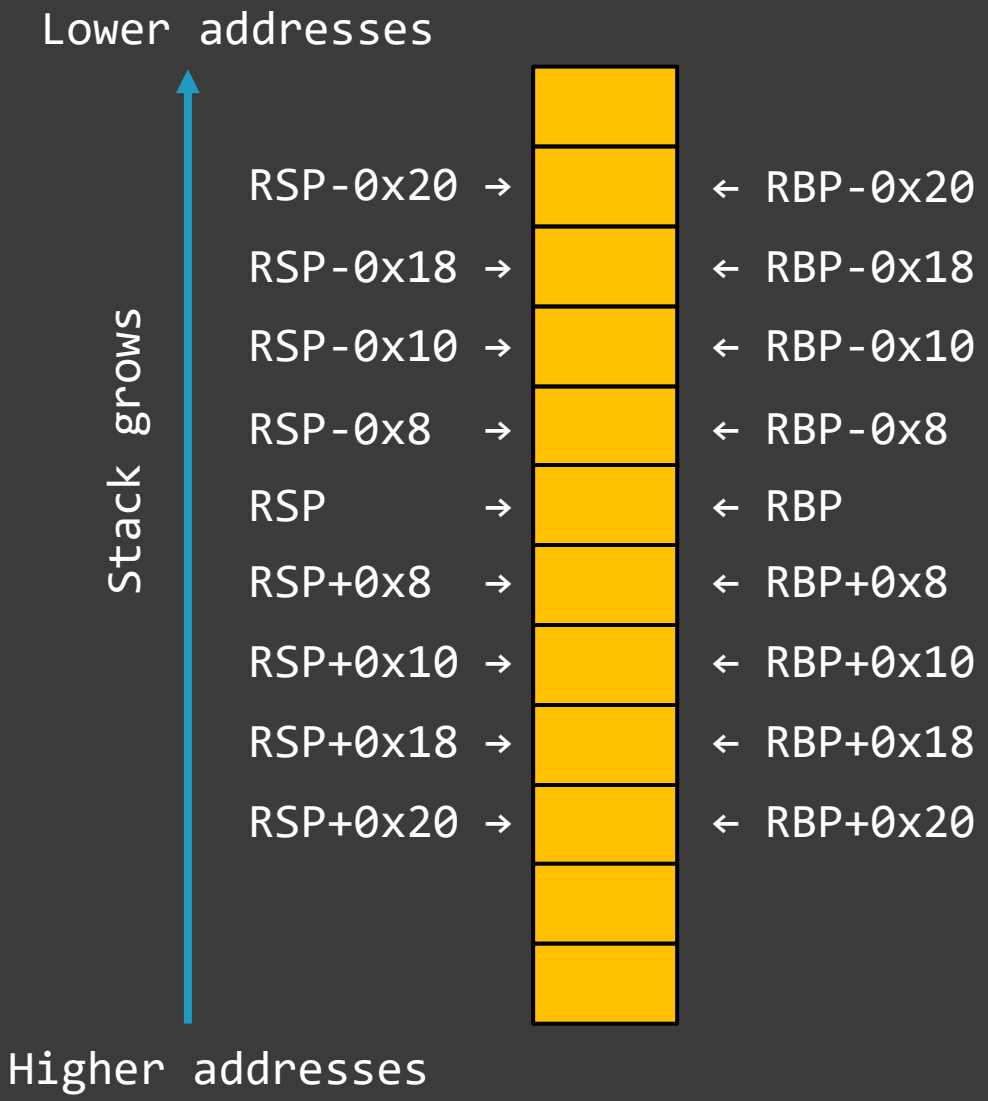
```
r
```

Instructions: registers (x64)

- `Opcode SRC, DST # default AT&T flavour`
- Examples:

```
mov   $0x10, %rax          # 0x10 → RAX
mov   %rsp, %rbp           # RSP → RBP
add   $0x10, %r10          # R10 + 0x10 → R10
imul  %ecx, %edx           # ECX * EDX → EDX
callq *%rdx                # RDX already contains
                           #    the address of func (&func)
                           # PUSH RIP; &func → RIP
sub   $0x30, %rsp          # RSP-0x30 → RSP
                           # make a room for local variables
```

Memory and Stack Addressing



Instructions: memory load (x64)

- `Opcode Offset(SRC), DST`
- `Opcode DST`
- Examples:

```
mov   0x10(%rsp), %rax       # value at address RSP+0x10 → RAX
mov   -0x10(%rbp), %rcx      # value at address RBP-0x10 → RCX
add   (%rax), %rdx           # RDX + value at address RAX → RDX
pop   %rdi                   # value at address RSP → RDI
                             # RSP + 8 → RSP
lea   0x20(%rbp), %r8        # address RBP+0x20 → R8
```

Instructions: memory store (x64)

- `Opcode SRC, Offset(DST)`
- `Opcode SRC|DST`
- Examples:

```
mov   %rcx, -0x20(%rbp)      # RCX → value at address RBP-0x20
addl  $1, (%rax)             # 1 + 32-bit value at address RAX →
                             #     32-bit value at address RAX
push  %rsi                   # RSP - 8 → RSP
                             # RSI → value at address RSP
inc   (%rcx)                 # 1 + value at address RCX →
                             #     value at address RCX
```

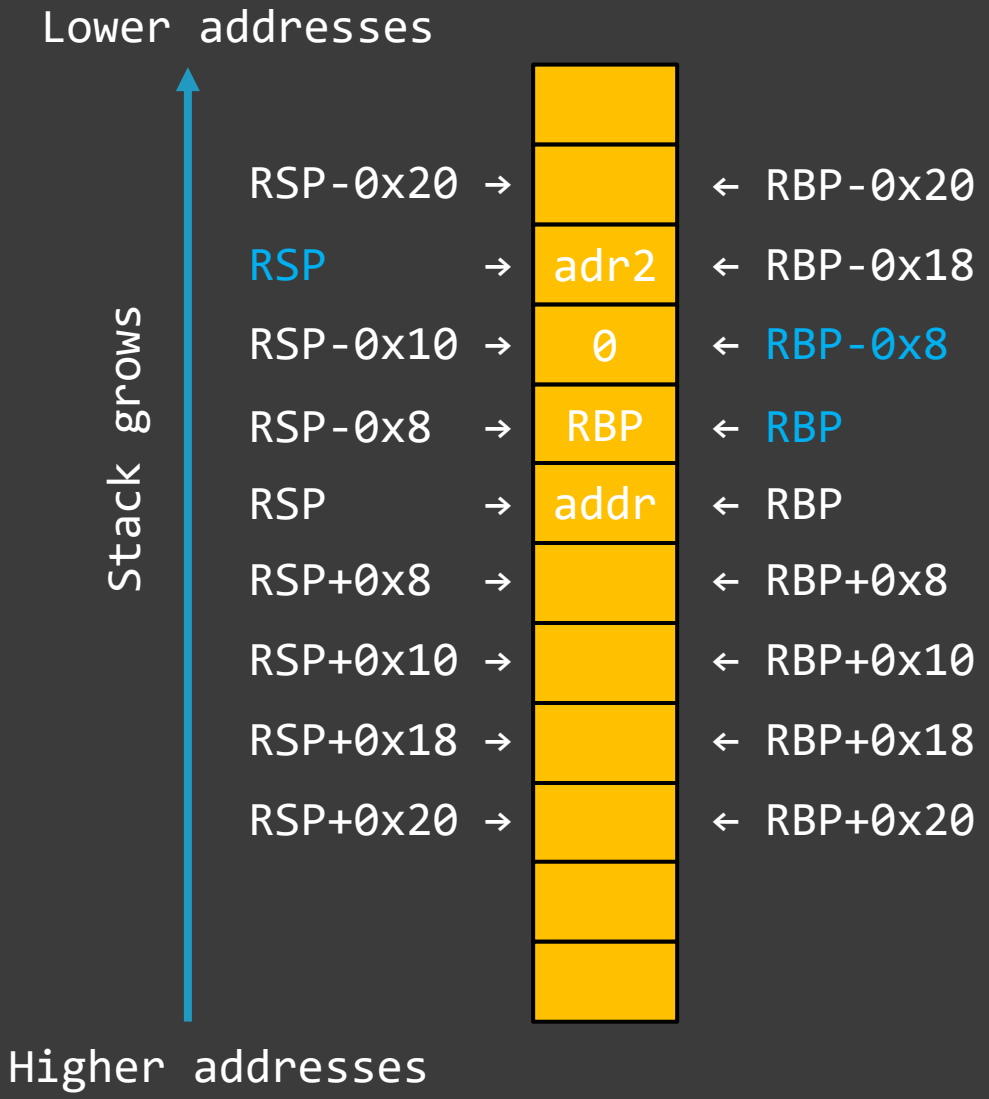
Instructions: flow (x64)

- `Opcode DST`
- Examples:

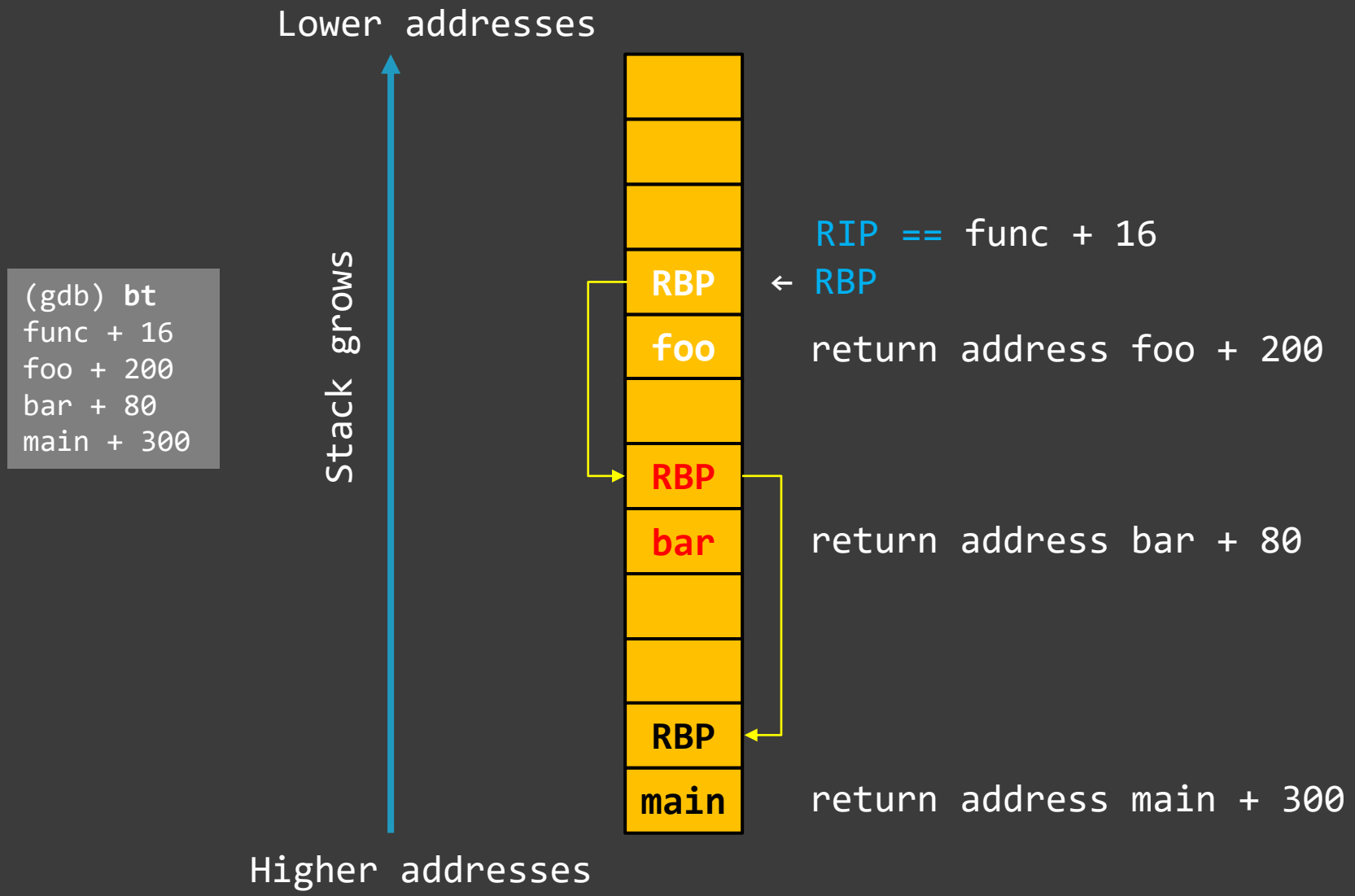
```
jmp    0x10493fc1c        # 0x10493fc1c → RIP
                          # (goto 0x10493fc1c)

call   0x10493ff74        # RSP - 8 → RSP
0x10493fc14:              # 0x10493fc14 → value at address RSP
                          # 0x10493ff74 → RIP
                          # (goto 0x10493ff74)
```

Function Call and Prolog (x64)

```
# void proc(int p1, long p2);
mov  $0x1, %edi
mov  $0x2, %rsi
call proc
addr:


# void proc2();
# void proc(int p1, long p2) {
#   long local = 0;
#   proc2();
# }
proc:
push %rbp
mov  %rsp, %rbp
sub  $0x8, %rsp
mov  $0, -0x8(%rbp)
call proc2
adr2:
...
```



Stack Trace Reconstruction (x64)



Part 4: ARM64 Disassembly

CPU Registers (ARM64)

- **X0** – **X28**, **W0** – **W28**
- Stack: **SP**, **X29** (**FP**)
- Next instruction: **PC**
- Link register: **X30** (**LR**)
- Zero register: **XZR**, **WZR**
- 64-bit floating point registers **D0** – **D31**

GDB Commands

```
info registers
```

WinDbg Commands

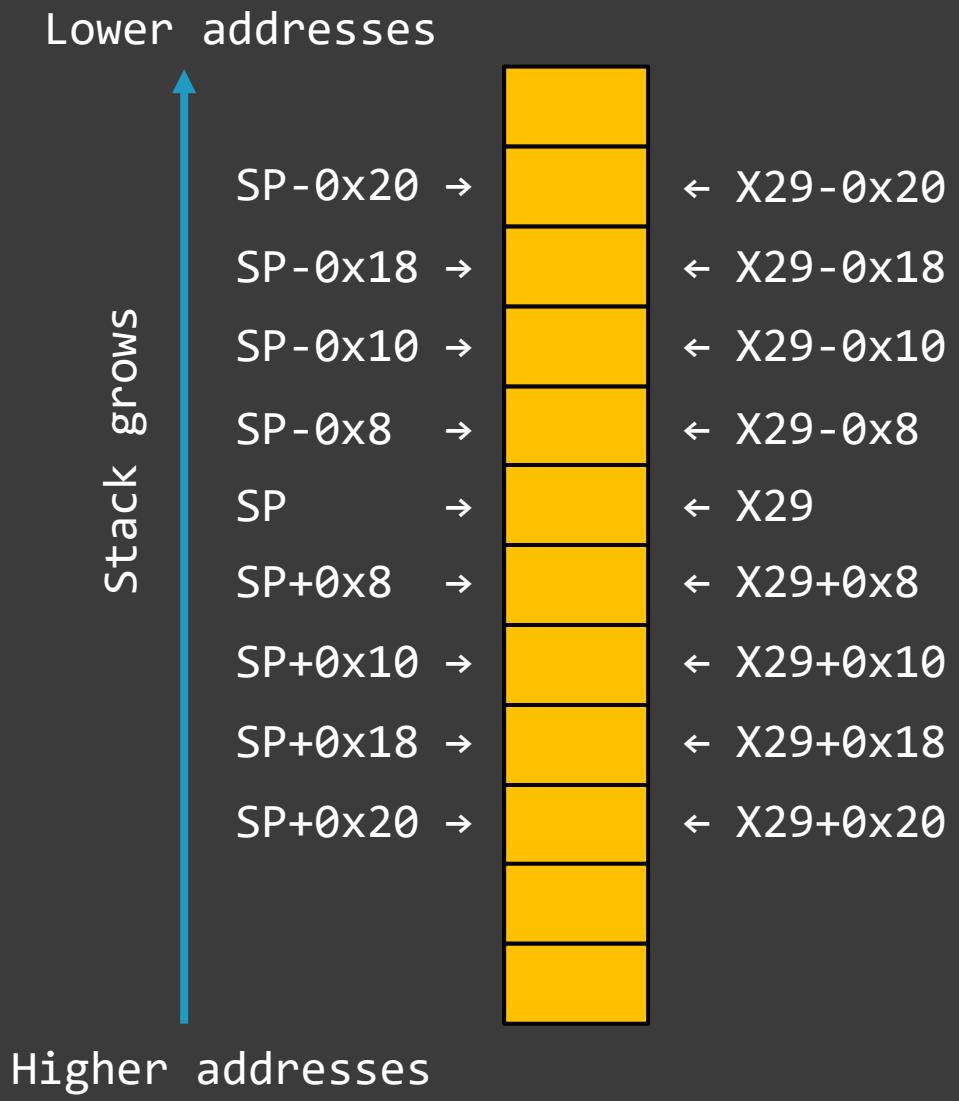
```
r
```

Instructions: registers (ARM64)

- `Opcode DST, SRC, SRC2`
- Examples:

```
mov   x0, #16              // X0 ← 16 (0x10)
mov   x29, sp              // X29 ← SP
add   x1, x2, #16          // X1 ← X2+16 (0x10)
mul   x1, x2, x3           // X1 ← X2*X3
blr   x8                   // X8 already contains
                           //    the address of func (&func)
                           // LR ← PC+4; PC ← &func

sub   sp, sp, #48          // SP ← SP-48 (-0x30)
                           // make a room for local variables
```

Memory and Stack Addressing



Instructions: memory load (ARM64)

- `Opcode DST, DST2, [SRC, Offset]`
- `Opcode DST, DST2, [SRC], Offset // Postincrement`
- Examples:

```
ldr   x0, [sp]              // X0 ← value at address SP+0
ldr   x0, [x29, #-8]        // X0 ← value at address X29-0x8
ldp   x29, x30, [sp, #32]   // X29 ← value at address SP+32 (0x20)
                            // X30 ← value at address SP+40 (0x28)
ldp   x29, x30, [sp], #16   // X29 ← value at address SP+0
                            // X30 ← value at address SP+8
                            // SP ← SP+16 (0x10)
```

Instructions: memory store (ARM64)

- `Opcode SRC, SRC2, [DST, Offset]`
- `Opcode SRC, SRC2, [DST, Offset]! // Preincrement`
- Examples:

```
str   x0, [sp, #16]          // x0 → value at address SP+16 (0x10)
str   x0, [x29, #-8]         // x0 → value at address X29-8
stp   x29, x30, [sp, #32]    // x29 → value at address SP+32 (0x20)
                             // x30 → value at address SP+40 (0x28)
stp   x29, x30, [sp, #-16]! // SP ← SP-16 (-0x10)
                             // x29 → set value at address SP
                             // x30 → set value at address SP+8
```

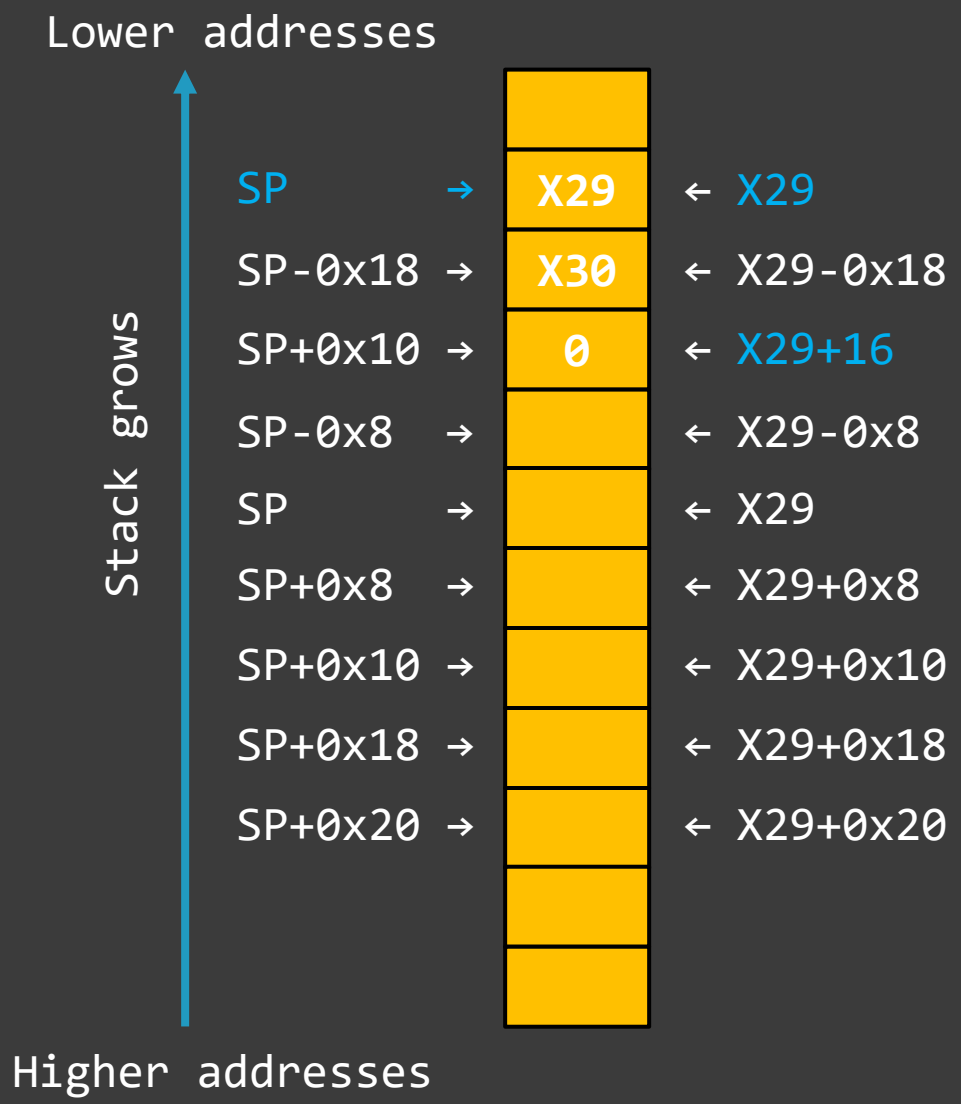
Instructions: flow (ARM64)

- `Opcode DST, SRC`
- Examples:

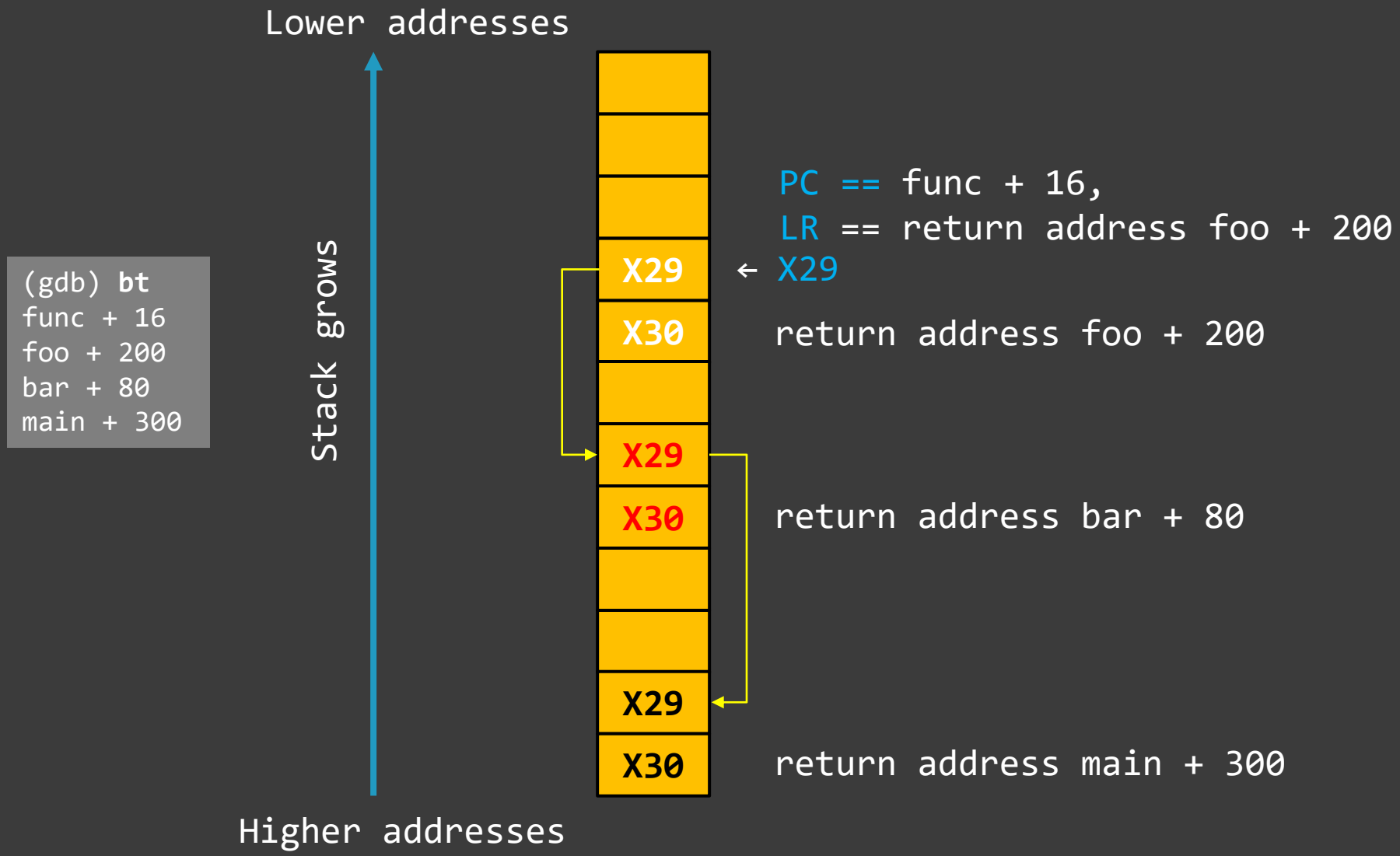
```
adrp   x0, 0x420000       // x0 ← 0x420000

b      0x10493fc1c        // PC ← 0x10493fc1c
                          // (goto 0x10493fc1c)
br     x17                // PC ← the value of X17

0x10493fc14:              // PC == 0x10493fc14
bl     0x10493ff74        // LR ← PC+4 (0x10493fc18)
                          // PC ← 0x10493ff74
                          // (goto 0x10493ff74)
```

Function Call and Prolog (ARM64)

```
// void proc(int p1, long p2);
mov  w0, #0x1
mov  x1, #0x2
bl   proc


// void proc2();
// void proc(int p1, long p2) {
//   long local = 0;
//   proc2();
// }
proc:
stp  x29, x30, [sp, #-32]!
mov  x29, sp
str  zxr, [x29, #16]
bl   proc2
...
```



Stack Trace Reconstruction (ARM64)



Part 5: Practice Exercises

Links

- Memory Dumps:

Included in Exercise 0

- Exercise Transcripts:

Included in this book

Exercise 0

- **Goal:** Install GDB and check if GDB loads a core dump correctly
- **Goal:** Install WinDbg Preview or Debugging Tools for Windows, or pull Docker image, and check that symbols are set up correctly
- **Patterns:** Stack Trace; Incorrect Stack Trace
- \ALCDA-Dumps\Exercise-A0-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A0-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A0-A64-WinDbg.pdf

Process Core Dumps

Exercises A1 – A12

Exercise A1

- **Goal:** Learn how to list stack traces, disassemble functions, check their correctness, dump data, get environment
- **Patterns:** Manual Dump (Process); Stack Trace; Stack Trace Collection; Annotated Disassembly; Paratext; Not My Version; Environment Hint
- \ALCDA-Dumps\Exercise-A1-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A1-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A1-A64-WinDbg.pdf

Exercise A2D

- **Goal:** Learn how to identify exceptions, find problem threads and CPU instructions
- **Patterns:** NULL Pointer (Data); Active Thread (x64, GDB)
- \ALCDA-Dumps\Exercise-A2D-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A2D-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A2D-A64-WinDbg.pdf

Exercise A2C

- **Goal:** Learn how to identify exceptions, find problem threads and CPU instructions
- **Patterns:** NULL Pointer (Code); Missing Frame (WinDbg)
- \ALCDA-Dumps\Exercise-A2C-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A2C-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A2C-A64-WinDbg.pdf

Exercise A2S

- **Goal:** Learn how to use external debugging information
- \ALCDA-Dumps\Exercise-A2S-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A2S-A64-GDB.pdf

Exercise A3

- **Goal:** Learn how to identify spiking threads
- **Patterns:** Active Thread; Spiking Thread
- \ALCDA-Dumps\Exercise-A3-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A3-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A3-A64-WinDbg.pdf

Exercise A4

- **Goal:** Learn how to identify heap regions and heap corruption
- **Patterns:** Dynamic Memory Corruption (Process Heap); Regular Data
- \ALCDA-Dumps\Exercise-A4-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A4-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A4-A64-WinDbg.pdf

Exercise A5

- **Goal:** Learn how to identify stack corruption
- **Patterns:** Local Buffer Overflow (User Space); Execution Residue (User Space)
- \ALCDA-Dumps\Exercise-A5-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A5-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A5-A64-WinDbg.pdf

Exercise A6

- **Goal:** Learn how to identify stack overflow, stack boundaries, reconstruct stack trace
- **Patterns:** Stack Overflow (User Mode)
- \ALCDA-Dumps\Exercise-A6-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A6-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A6-A64-WinDbg.pdf

Exercise A7

- **Goal:** Learn how to identify active threads
- **Patterns:** Divide by Zero (User Mode, x64); Invalid Pointer (General); Multiple Exceptions (User Mode); Near Exception
- \ALCDA-Dumps\Exercise-A7-x64-GDB.pdf

Exercise A8

- **Goal:** Learn how to identify runtime exceptions, past execution residue and stack traces, identify handled exceptions
- **Patterns:** C++ Exception; Execution Residue (User Space); Past Stack Trace; Coincidental Symbolic Information; Handled Exception (User Space)
- \ALCDA-Dumps\Exercise-A8-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A8-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A8-A64-WinDbg.pdf

Exercise A9

- **Goal:** Learn how to identify heap leaks
- **Patterns:** Memory Leak (Process Heap); Module Hint
- \ALCDA-Dumps\Exercise-A9-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A9-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A9-A64-WinDbg.pdf

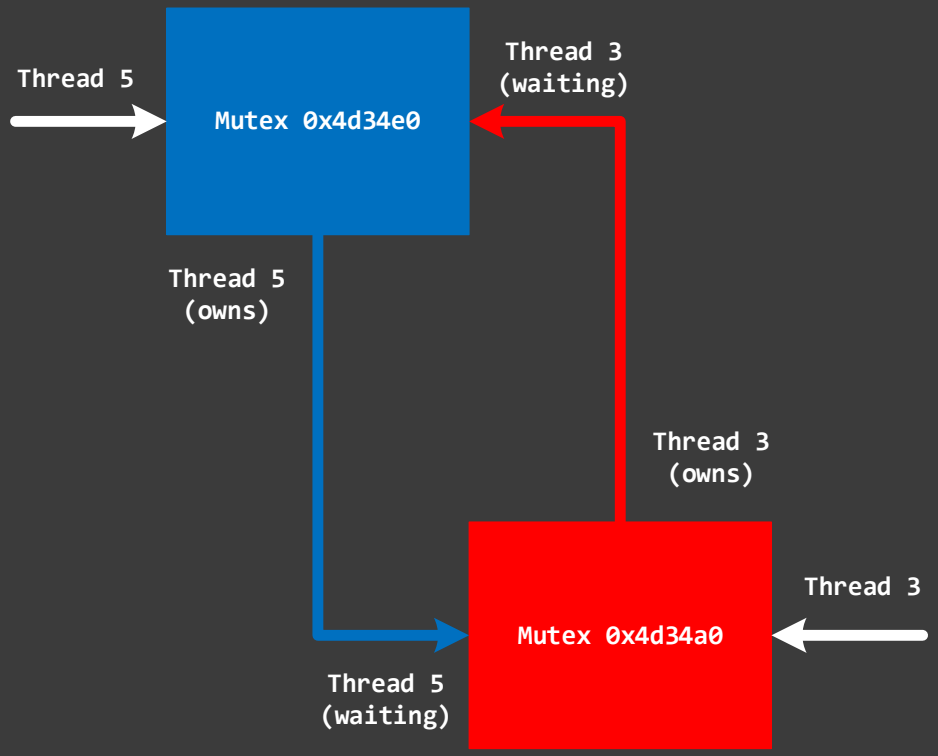
Exercise A10

- **Goal:** Learn how to identify heap contention wait chains, synchronization issues, advanced disassembly, dump arrays
- **Patterns:** Double Free (Process Heap); High Contention (Process Heap); Wait Chain (General); Critical Region; Self-Diagnosis (User Mode)
- \ALCDA-Dumps\Exercise-A10-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A10-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A10-A64-WinDbg.pdf

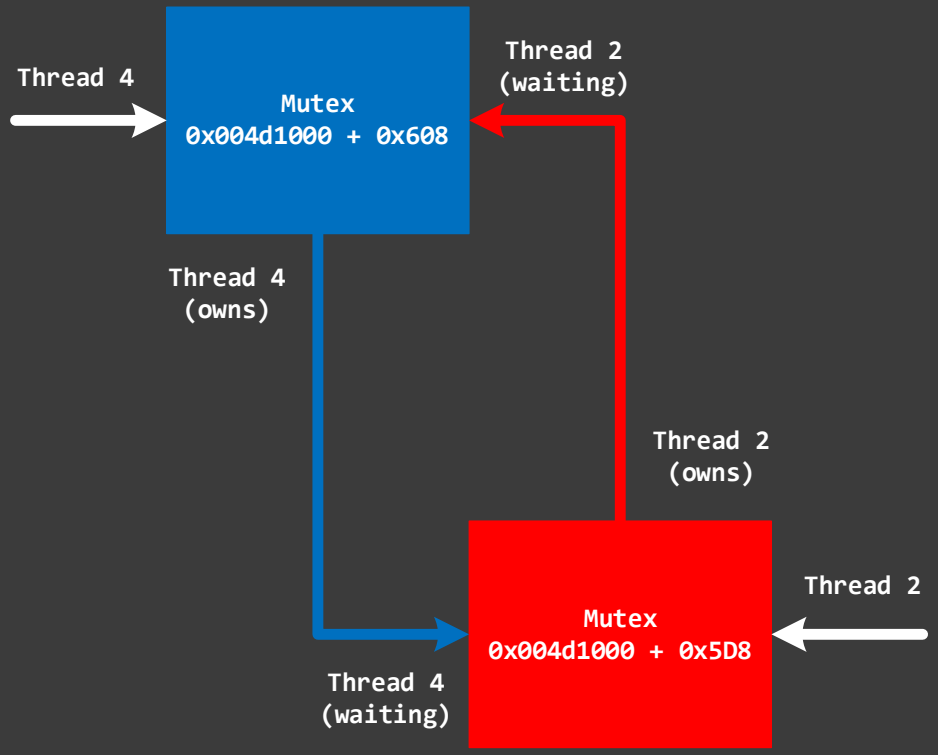
Exercise A11

- **Goal:** Learn how to identify synchronization wait chains, deadlocks, hidden and handled exceptions
- **Patterns:** Wait Chain (Mutex Objects); Deadlock (Mutex Objects, User Space); Disassembly Hole (WinDbg)
- \ALCDA-Dumps\Exercise-A11-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A11-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A11-A64-WinDbg.pdf

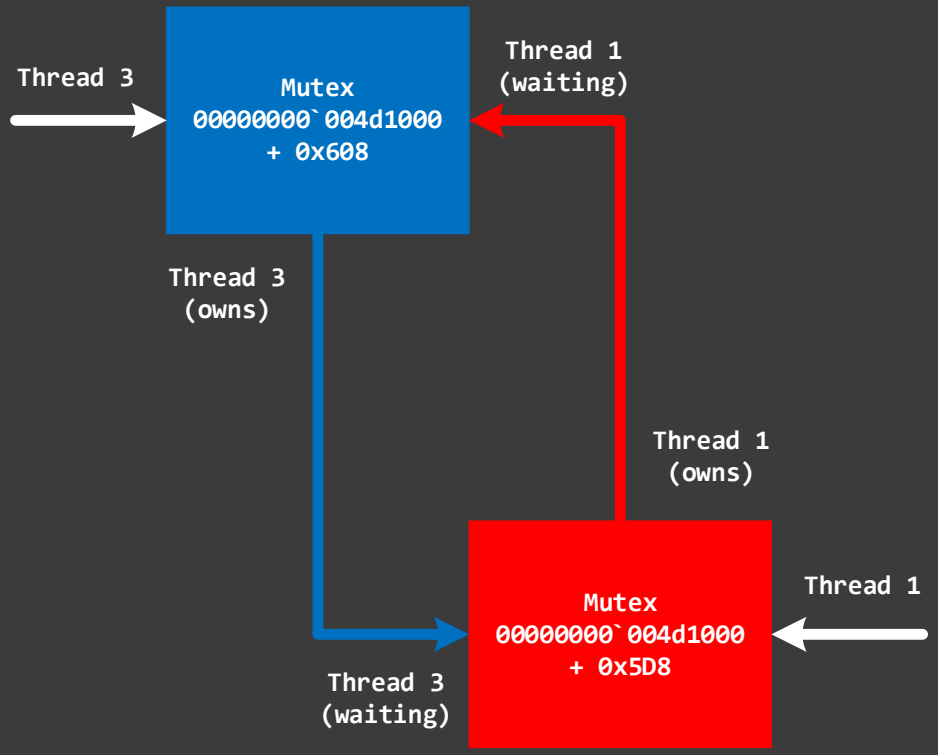
Deadlock (x64, GDB)



Deadlock (A64, GDB)



Deadlock (A64, WinDbg)



Exercise A12

- **Goal:** Learn how to dump memory for post-processing, get the list of functions and module variables, load symbols, inspect arguments and local variables
- **Patterns:** Module Variable
- \ALCDA-Dumps\Exercise-A12-x64-GDB.pdf
- \ALCDA-Dumps\Exercise-A12-A64-GDB.pdf
- \ALCDA-Dumps\Exercise-A12-A64-WinDbg.pdf

Kernel Core Dumps

Exercises K1 – K5

Exercise K1

- **Goal:** Learn how to navigate a normal kernel dump
- **Patterns:** Manual Dump (Kernel); Stack Trace Collection
- \ALCDA-Dumps\Exercise-K1-x64-GDB.pdf

Exercise K2

- **Goal:** Learn how to navigate a problem kernel dump
- **Patterns:** Exception Stack Trace; NULL Pointer (Data); Execution Residue (Kernel Space); Value References
- \ALCDA-Dumps\Exercise-K2-x64-GDB.pdf

Exercise K3

- **Goal:** Learn how to recognize problems with kernel threads, identify their owner module, and follow call chains
- **Patterns:** Origin Module; NULL Pointer (Code); Hidden Call
- \ALCDA-Dumps\Exercise-K3-x64-GDB.pdf

Exercise K4

- **Goal:** Learn how to identify spiking kernel threads
- **Patterns:** Stack Trace Collection (CPUs); Interrupt Stack; Spiking Thread
- \ALCDA-Dumps\Exercise-K4-x64-GDB.pdf

Exercise K5

- **Goal:** Learn how to identify kernel stack overflow and kernel stack boundaries
- **Patterns:** Stack Overflow (Kernel Mode)
- \ALCDA-Dumps\Exercise-K5-x64-GDB.pdf

Follow-up Courses

Advanced
Linux Core Dump Analysis
with Data Structures

Accelerated Linux Disassembly,
Reconstruction, and Reversing

Accelerated Linux Debugging[4]

Pattern Links (Linux and GDB)

Active Thread
C++ Exception
Critical Region
Divide by Zero
Execution Residue
High Contention
Memory Leak
Local Buffer Overflow
Module Hint
Not My Version
NULL Pointer (Data)
Self-Diagnosis
Stack Overflow (User Mode)
Stack Trace Collection
Regular Data
Near Exception
Invalid Pointer
Past Stack Trace
Exception Stack Trace
Value References
Hidden Call
Interrupt Stack

Annotated Disassembly
Coincidental Symbolic Information
Deadlock (Mutex Objects, User Space)
Environment Hint
Handled Exception
Dynamic Memory Corruption
Lateral Damage
Manual Dump (Process) / (Kernel)
Module Variable
NULL Pointer (Code)
Paratext
Spiking Thread
Stack Trace
Wait Chain (General)
Multiple Exceptions
Wait Chain (Mutex Objects)
Missing Frame
Disassembly Hole
Deadlock (Mutex Objects, Kernel Space)
Origin Module
Stack Trace Collection (CPUs)
Stack Overflow (Kernel Mode)

Resources

- DumpAnalysis.org / SoftwareDiagnostics.Institute / PatternDiagnostics.com
- Debugging.TV / YouTube.com/DebuggingTV / YouTube.com/PatternDiagnostics
- Rosetta Stone for Debuggers
- Accelerated macOS Core Dump Analysis (also covers LLDB)
- GDB Pocket Reference
- A64 Instruction Set Architecture
- A64 Base Instructions
- Encyclopedia of Crash Dump Analysis Patterns, Third Edition
- Debugging, Disassembly & Reversing in Linux for x64 Architecture
- Foundations of Linux Debugging, Disassembling, and Reversing
- Foundations of ARM64 Linux Debugging, Disassembling, and Reversing
- Memory Dump Analysis Anthology (some articles in volumes 1, 7, 9A cover GDB)

Q&A

Please send your feedback using the contact form on PatternDiagnostics.com

Thank you for attendance!