The Operating System Project

Start Here

Version 4.50: September 2018

Table of Contents

Table of Contents(2)

Brushing up your C programming skills.

While brushing up on your skills, it's highly recommended that you select an IDE for your project. This will save you many many hours of time!

There are many places you can review C programming. The document listed below looks at some of the differences between C and Java. The concept of pointers is where most java programmers have difficulty. If you need more information, the web is your friend.

C_By_Example.ppt can be found on the Project Home Page – the same place you found this Start_Here.ppt.

Compiling the program.

The first thing you need to figure out is the environment where you will be doing this project.

If you're a LINUX fan, then life is easy since gcc is installed on every computer. I would recommend that as a good way to get started. If you've been programming on Windows, then I'd recommend the free Visual Studio version designed for students – it's great though may be a bit formidable to start with.

I've built this code with Eclipse on Windows, with gcc installed on Windows, and with a standard gcc on Linux. It worked for these three environments.

The first thing you need to do is define whether you will be building on Windows or Linux. In reality, your code should be able to run on Linux. Windows, or iOS. You are writing in standard C which is completely portable.

In studentConfiguration.h, there is a C preprocessor statement that determines the underlying OS where you are compiling and sets configuration accordingly.

After you've moved the files from the webpage into a new directory, compile the program:

```
>gcc –g *.c -lm –std=gnu11 -Wall -o z502              ← Windows
>gcc –g *.c -lm –lpthread –std=gnu11 -Wall -o z502    ← Linux
```

```
This will create an executable called z502.  Because of the –g, you can debug
   this program.
```

Compiling the program.

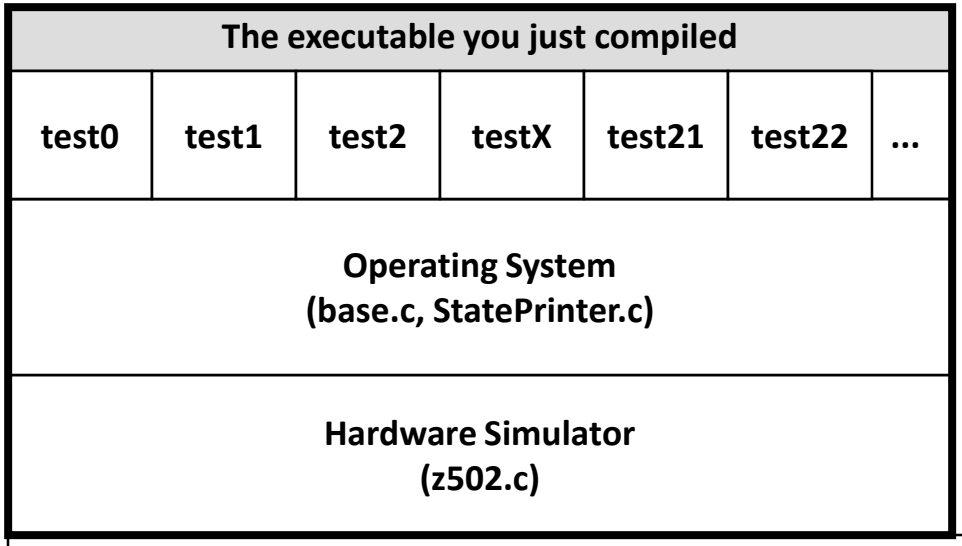
Executing the program will give the following output:

```
This is Simulation Version 4.50 and Hardware Version 4.50.

Program called with 2 arguments: z502 test0
Calling with argument 'sample' executes the sample program.
Simulation is running as a UniProcessor
Add an 'M' to the command line to invoke multiprocessor operation

This is Release 4.50:  Test 0
SVC handler: get_time
Arg 0: Contents = (Decimal)  4300384,  (Hex)   419E60
Time of day is 0
SVC handler: term_proc
Arg 0: Contents = (Decimal)       -1,  (Hex) FFFFFFFF
Arg 1: Contents = (Decimal)  4300388,  (Hex)   419E64
ERROR: Test should be terminated but isn't.
ERROR:  Simulation did not end correctly
```

If you get this result, you know your compilation was successful. Your task now for Test 0 will be to make this code work right, so it doesn't produce the errors you see here.

Compiling the program.

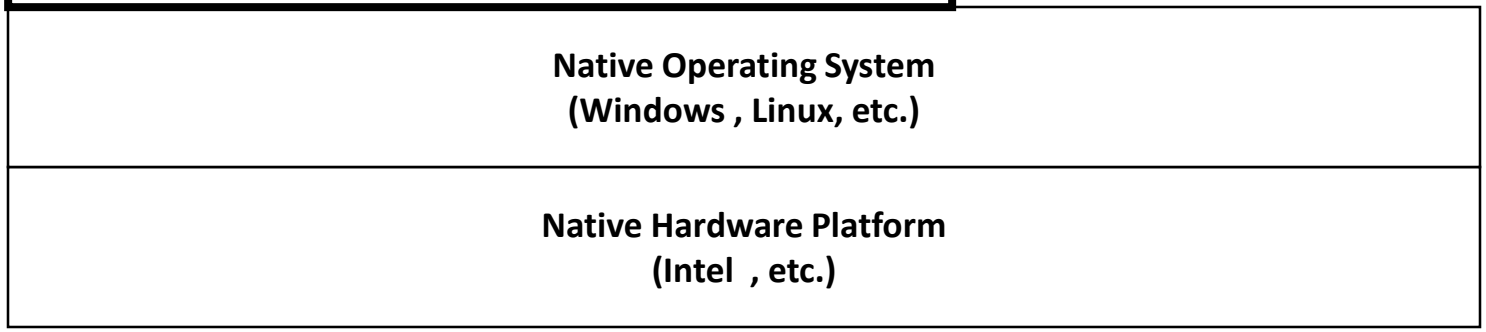
You just compiled the program shown in the solid box. It includes a hardware simulator, the beginnings of an operating system that you will expand, and test cases that drive your development of the OS.



All elements inside the heavy box are in a single process, running *several* threads of execution.

All I/O devices in the program are simulated entities. This includes the timer device and the disk devices.

***Try* to treat the Hardware Simulator as a "black box" and use the architecture specification instead.**



Understanding how the program flows.

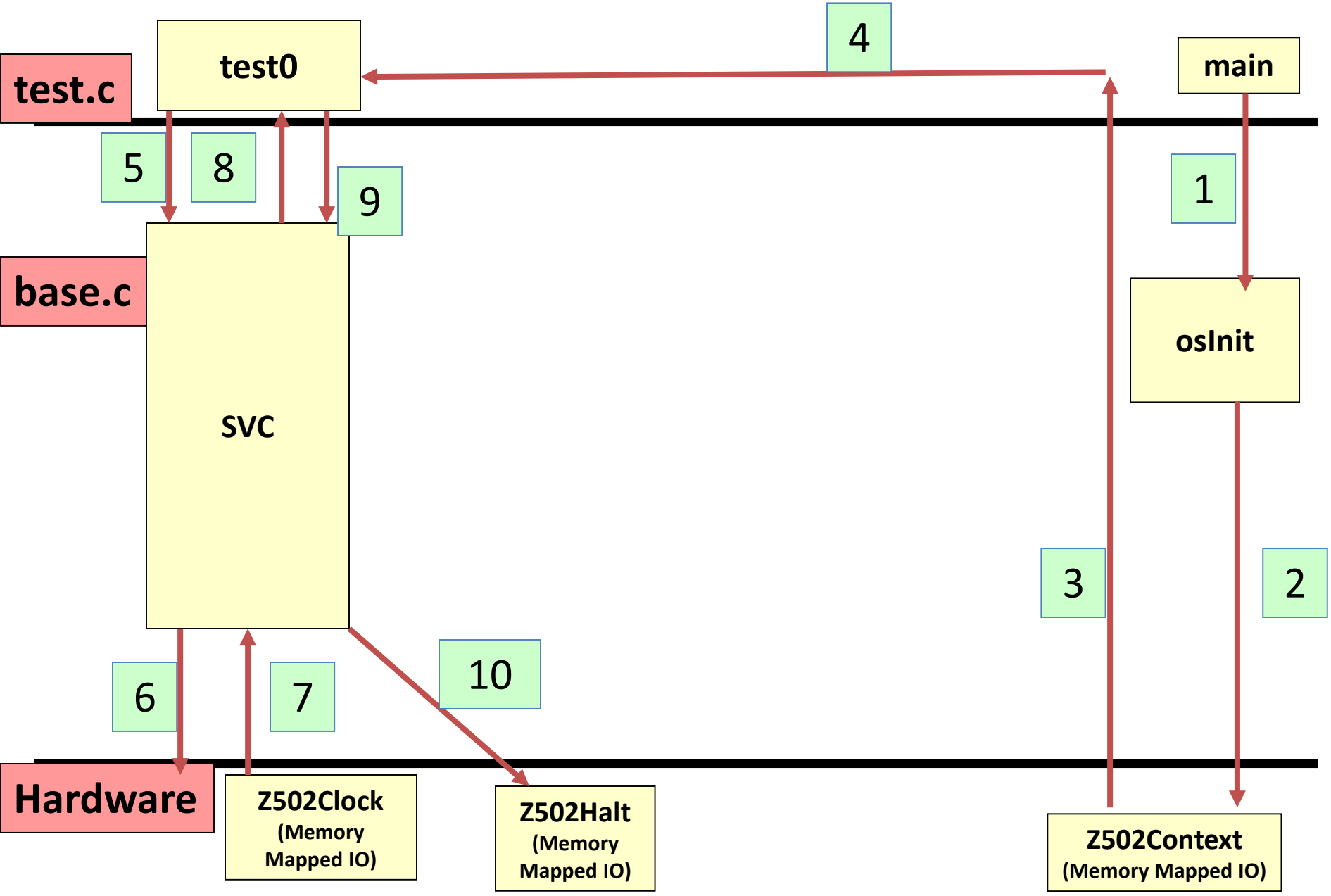
The next slides describe the starting code that's given to you (what you've already compiled). It shows how the program flows. The important actions are:

Test0 in test.c contains system calls – requests for service from the Operating System.

Those system calls come to the routine **svc( )** in base.c. This is in the OS – you're writing the OS so you own this code.

In **svc**, you call (make a subroutine call) to the hardware in order to implement the action requested by Test0.

The Execution of test0



The Execution of test0

1. All C programs start in main(). A temporary context is created and the simulation starts by requesting to run on that context.
2. osInit is a routine in your operating system. For right now, all it does is create a context that will allow test0 to run.
3. We go out to test0. It is time to run the user code.
4. Test0 does a system call GET_TIME_OF_DAY. A system call produces a software interrupt that causes execution to go to svc(), the software service routine.
5. svc must get the time in order to service the system call. It calls the hardware to do that. It passes by reference a variable in which the time can be placed.
6. Z502Clock is a hardware routine that keeps track of the time. It passes back this time to svc.
7. svc passes back the time to test0. test0 prints out that time as part of its code.
8. test0 does a TERMINATE_PROCESS system call – it's all done with its job. It makes this call and again the execution ends up back in svc.
9. svc must handle this terminate_process request. Eventually this code will be more complicated, but for right now, since there's nothing else for the OS to do, it simply ends the simulation by halting the processor.

What does Test0 do?

```
void test0(void) {
    printf("This is Release %s:  Test 0\n", CURRENT_REL);
    GET_TIME_OF_DAY(&ReturnedTime);

    printf("Time of day is %ld\n", ReturnedTime);
    TERMINATE_PROCESS(-1, &ErrorReturned);

    // We should never get to this line since the TERMINATE_PROCESS call
    // should cause the program to end.
    printf("ERROR: Test should be terminated but isn't.\n");
}                                                    // End of test0
```

There are two system calls:

GET_TIME_OF_DAY(& `ReturnedTime`); Get the time the hardware thinks it is. This is NOT in any normal units like seconds or whatever. Note that following the C convention, we're passing the ADDRESS of the variable Z502_REG1 (that's what the "&" does.) Then in the next line

printf("Time of day is %d\n", `ReturnedTime`);

the value that's in the variable is used in the printf statement.

TERMINATE_PROCESS(-1, & `ErrorReturned`); has two arguments. The "-1" says terminate the current process. The & `ErrorReturned` gives the address of a variable that the OS can use to return an error.

System Call Interface document is where you will find a description of the system calls. Syscalls.h contains the macros that implement these system calls.

Test0 - What do you need to change?

To make Test0 work, "all" you need do is change code in svc(). Let's start by looking at the original code:

```
void    svc( SYSTEM_CALL_DATA *SystemCallData ) {
    short               call_type;
    static short        do_print = 10;
    short               i;

    call_type = (short)SystemCallData->SystemCallNumber;
    if ( do_print > 0 ) {
        printf( "SVC handler: %s\n", call_names[call_type]);
        for (i = 0; i < SystemCallData->NumberOfArguments - 1; i++ ){
         //Value = (long)*SystemCallData->Argument[i];
            printf( "Arg %d: Contents = (Decimal) %8ld,  (Hex) %8lX\n", i,
            (unsigned long )SystemCallData->Argument[i],
            (unsigned long )SystemCallData->Argument[i]);
        }
    do_print--;
    }
}                                                   // End of svc
```

- SystemCallData - a data structure containing everything we know about this system call.
- Call_type– a variable contains the type of system call that's being passed to svc. In svc, this variable, as well as the arguments requested by the system call in test0, are printed out so you can see them.
- The do_print variable is here simply to do some initial printout, but then not clutter up printouts when there are many system calls. You can see how it works from the code.

Test0 - Step by step example.

```
void    svc SYSTEM_CALL_DATA *SystemCallData ) {
    short               call_type;
    static INT16        do_print = 10;
    INT32               Time;
    MEMORY_MAPPED_IO    mmio;


    call_type = (short)SystemCallData->SystemCallNumber;
    if ( do_print > 0 ) {
        // same code as before
    }
    switch (call_type) {
        // Get time service call
        case SYSNUM_GET_TIME_OF_DAY:   // This value is found in syscalls.h
            mmio.Mode = Z502ReturnValue;
            mmio.Field1 = mmio.Field2 = mmio.Field3 = 0;
            MEM_READ(Z502Clock, &mmio);
            *(long *)SystemCallData->Argument[0] = mmio.Field1;
            break;
        // terminate system call
        case SYSNUM_TERMINATE_PROCESS:
             mmio.Mode = Z502Action;
             mmio.Field1 = mmio.Field2 = mmio.Field3 = 0;
             MEM_WRITE(Z502Halt, &mmio);
             break;
        default:
            printf( "ERROR!  call_type not recognized!\n" );
            printf( "Call_type is - %i\n", call_type);
    }                                                   // End of switch
}                                                       // End of svc
```

Declare the MEMORY_MAPPED_IO structure here.

This is easy – all I did was find the code in sample.c that does this same call to the hardware. Then I copied it here! At this point, it's magic.

We're returning the time to the caller (in test0). The ARG1_PTR could be pointing to 32 bits or 64 bits. We cast it to long since the data value to match the underlying OS. Then the "*" on the front says this is a pointer. (This is not obvious stuff if you're new to C).

In this test, when Test0 says it wants to terminate, there's nothing more to do, so we simply call the hardware to say we're done. Note how this is in a different case statement from the time.

If a illegal system call number comes in here, we want to know about it and report an error.

Test0 - Step by step example.

Here's what the execution looks like after the code has been changed. Note that the time of day is reported as "45" in this case (you're number may be different). Note also that the simulator says that the test ended happily.

```
This is Simulation Version 4.50 and Hardware Version 4.50.

Program called with 2 arguments: Z502.exe test0
Calling with argument 'sample' executes the sample program.
This is Release 4.50:  Test 0
SVC handler: get_time
Arg 0: Contents = (Decimal)  4300384,  (Hex)   419E60
Time of day is 45
SVC handler: term_proc
Arg 0: Contents = (Decimal)       -1,  (Hex) FFFFFFFF
Arg 1: Contents = (Decimal)  4300388,  (Hex)   419E64
Hardware Statistics during the Simulation
Context Switches =     1:  CALLS =    13:  Masks =     0
The Z502 halts execution and Ends at Time 50
Exiting the program
```

Steps For a Perfect Project (1)

- DO NOT modify any of my files. You can change base.c, StudentConfiguration.h, and you can add any other .c and .h files.
- DO NOT use any of the routines in z502.c in any way other than the public interface.

Steps For a Perfect Project (2)

- StudentManual.pdf – contains the output requirements for each of the tests.

- Some people like to keep a source maintenance system such as GitHub. Do NOT make it public – if you do it's the same as giving everyone your code – and that's plagiarism.

- You will get points for my being able to compile and run your code with no problems. This is called code portability.

Steps For a Perfect Project (3)

- Use an IDE. This is a tremendous time saver. I use Eclipse but many people are fond of Microsoft Visual Studio. Any choice is up to you.
- You can develop your code on whatever platform you wish.
- When you hand in your project, I will compile and run it from the command line in Windows or Linux. Those are the ONLY two OSs I will use. It shouldn't matter which I use.
- On the command line, I will compile using the commands given on page 5 of this document. If you want to give me a script (makefile, .bat file) I will use that. If your code does not produce an executable, you will lose points. This is the meaning of "portability."

Test1 - What you need to do Overview

This is a “small” incremental step in your code development. It involves only a couple of pieces:

1. Be able to read test names from the command line and execute the correct test.
2. Create a method osCreateProcess that will enable you to generate and save state for each process.
3. In SVC, build a way for a system call GET_PROCESS_ID() to get information about the running process.

What follows are steps to accomplish this.

Test1 - What you need to do

1. Be able to read test names from the command line and execute the correct test.

In osInit, there's a line of code →

```
if ((argc > 1) && (strcmp(argv[1], "sample") == 0)) {
```

It's purpose is to catch the command line and look at the test you are requesting. If you create the line

```
if ((argc > 1) && (strcmp(argv[1], "test1") == 0)) {,
```

Then it will direct your code instead to start in test1.

Test1 - What you need to do

2. Create a method osCreateProcess that will enable you to generate and save state for each process.

   In Step 1 (the last slide), you learned how to figure out what test you are running. That's only ONE of the characteristics of the process you are creating. To save all these properties, it's easiest to create a structure called a Process Control Block (a PCB).

   AND since you will be creating many more processes, it's easiest to have a separate method that you use for this task.

   You will be asked for the Process ID – this is an Operating System characteristic assigned to a process; you can use whatever value makes sense to you. Store the Process ID in the PCB.

Test1 - What you need to do

3. In SVC, build a way for a system call GET_PROCESS_ID() to get information about the running process.

This will work the same way you did the GET_TIME SYSTEM CALL, but now, instead of asking the Z502 Hardware for the answer, you will ask your own Process Management code to read the PID of the running process.

Test2 - What you need to do

What does Test2 do? This section contains:

- Summary information.
- Starting Architecture of the Simulator Environment; the InterruptHandler
- Implementation of Test2:
  - Step 1
  - Step 2
  - Step 3

What does Test2 do?

```
void test2(void) {
    long  SleepTime = 100;
    INT32 time1, time2;

    aprintf("This is Release %s:  Test 1\n", CURRENT_REL);
    GET_TIME_OF_DAY(&time1);

    SLEEP(SleepTime);

    GET_TIME_OF_DAY(&time2);

    aprintf("Sleep Time = %d, elapsed time= %d\n", SleepTime, time2 - time1);
    GET_PROCESS_ID("", &MyProcessID, &ErrorReturned);
    GET_TIME_OF_DAY(&Time2);
    aprintf("Test 4, PID %ld, Ends at Time %ld\n", MyProcessID, Time2);
    TERMINATE_PROCESS(-1, &ErrorReturned);

    printf("ERROR: Test should be terminated but isn't.\n");
}                                                    // End of test2
```

What does Test2 do?

Let's look at this code. A lot is the same as test0. There are two calls to GET_TIME_OF_DAY, and one call to TERMINATE_PROCESS. One new piece is the SLEEP. The call GET_PROCESS_ID was used in test1.

There is a new system call:

SLEEP(TimeToSleep); With this call, we're not getting a value returned to us – we're simply passing to the OS, the amount of time we want to "sleep". We don't want control to come back to this code for a least TimeToSleep time units.

There is a repeat system call:

GET_PROCESS_ID("", &MyProcessID, &ErrorReturned); With this call, we're getting a value returned to us – we give to the call the address of where we want it to return the process ID of the current process – the one making the call. The Process ID (or PID) is a property of the process – it is kept in the Process Control Block.

The document *CS502SystemCalls* is where you will find a description of the system calls. Syscalls.h contains the macros that implement these system calls.

Test2 – Summary Information
Interrupt Handling

An Operating System is just a program waiting for someone to give it something to do. It's the hardware that transfers control into the OS. There are three ways to do this:

- Interrupts **(starts executing at InterruptHandler in base.c)**
  - TIMER_INTERRUPT from the delay timer
  - DISK_INTERRUPT from disk 1, 2, ...
- Faults **(starts executing at fault_handler in base.c)**
  - INVALID_MEMORY fault
  - CPU_ERROR fault
  - PRIVILEGED_INSTRUCTION fault
- Traps **(starts executing at svc in base.c)**
  - SOFTWARE_TRAP for each system call

Test2 – Summary Information System Modes

Modes have to do with privileges. The code executing in User mode has access to the code in Test.c and access to data associated with the test. In Kernel Mode, the code can see, touch, smell, and modify ANYTHING!

- User Mode
  - Address space for user programs is divided into
    - C code "program" memory for instructions and for local variables.
    - User "data" memory, referenced through a virtual address space, and called MEMORY. You don't need to know this until Test21.
- Kernel Mode
  - Instruction set includes C language instructions, plus
    - access to all the Z502 registers
    - access to the privileged instructions of the Z502 instruction set
      - » I/O primitives
      - » memory primitives
      - » context switching primitives
    - These are all available through provided macros

Test2 – Summary Information Hardware Actions on Interruption

- User registers are saved in Z502 *Hardware Context – this is done by the hardware so you don't have to worry about it.*

- The InterruptHandler queries the hardware to find out about the interrupt. There are three requests to the hardware. These are explained in excruciating detail in *Z502 Architecture Specification* – see Section 5.3.
- *The calls:*

  a) ask for the device that caused the interrupt and also get it's status.

- Execution mode is set to *kernel –we're now running in the OS!*
- Hardware begins execution at InterruptHandler when the hardware has something to communicate (i.e., it took an error, it's successfully completed its work, etc.)

Test2 – The InterruptHandler

```
void    InterruptHandler( void ) {
        INT32 DeviceID;
        INT32 Status;

        MEMORY_MAPPED_IO mmio;   // Enables communication with Z502

        // Get cause of interrupt
        mmio.Mode = Z502GetInterruptInfo;
        mmio.Field1 = mmio.Field2 = mmio.Field3 = 0;
        MEM_READ(Z502InterruptDevice, &mmio);
        DeviceID = mmio.Field1;
        Status = mmio.Field2;

        // Check the status of the interrupt to make sure no
        // error occurred.

        // Do Whatever Work You Want Here

}            // End of InterruptHandler
```

Test2 – Summary Information Hardware Context

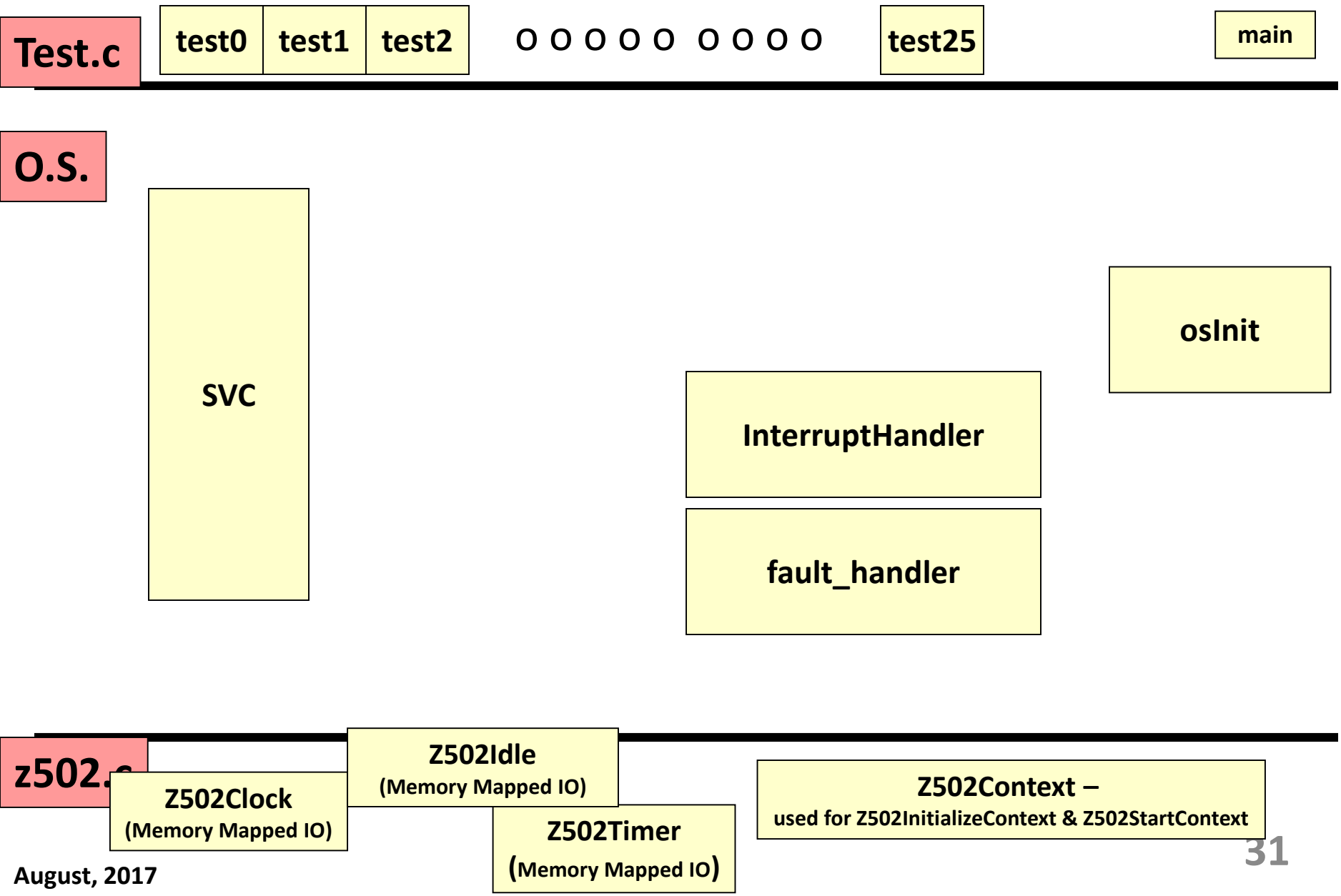
- The *context* is the state of the executing CPU; essentially its registers.
- The Hardware context is really just the set of registers , plus an entry address.
- The OS only deals with the handle to a context. Typically this is stored in the process control block. You don't EVER need to know what's in that context.
- Z502 Operations for manipulating contexts
  - Z502InitializeContext
  - Z502StartContext

Writing Test2

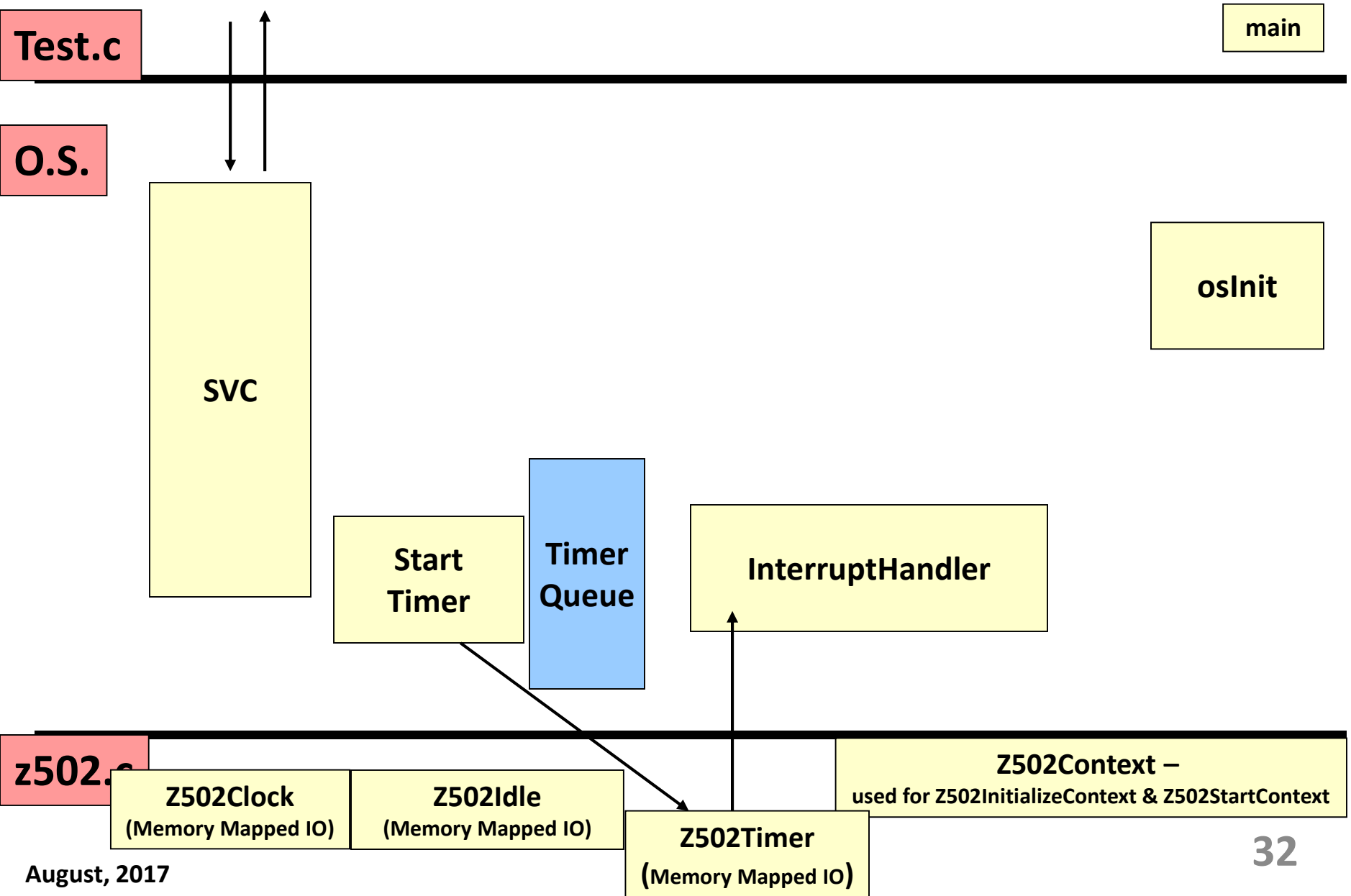
Write this test in multiple stages – get each stage working before you start the next one; take baby steps.

- **Stage 1:** In svc for the SLEEP system call, you should:
  a) Change osInit so it will execute test2.
  b) Start the clock (see sample.c for an example of this – see also Z502Hardware.html for the API for the timer.
  c) Wait for a timer interrupt by generating a Memory Mapped IO → Z502Idle
  d) Control will not pass back from IDLE to it's caller until the timer has completed its delay.
- **Stage 2:** In method OSCreateProcess ()
  a) Ask the Hardware for the Context for this process. You already know how to create the PCB where you store that Context.
- **Stage 3:** Timer Queue is an object that contains an ordered list of the processes waiting for or currently being handled by the timer.
  a) Your Svc calls AddToTimerQueue()
  b) Your InterruptHandler → TimerInterrupt → RemoveFromTimerQueue();

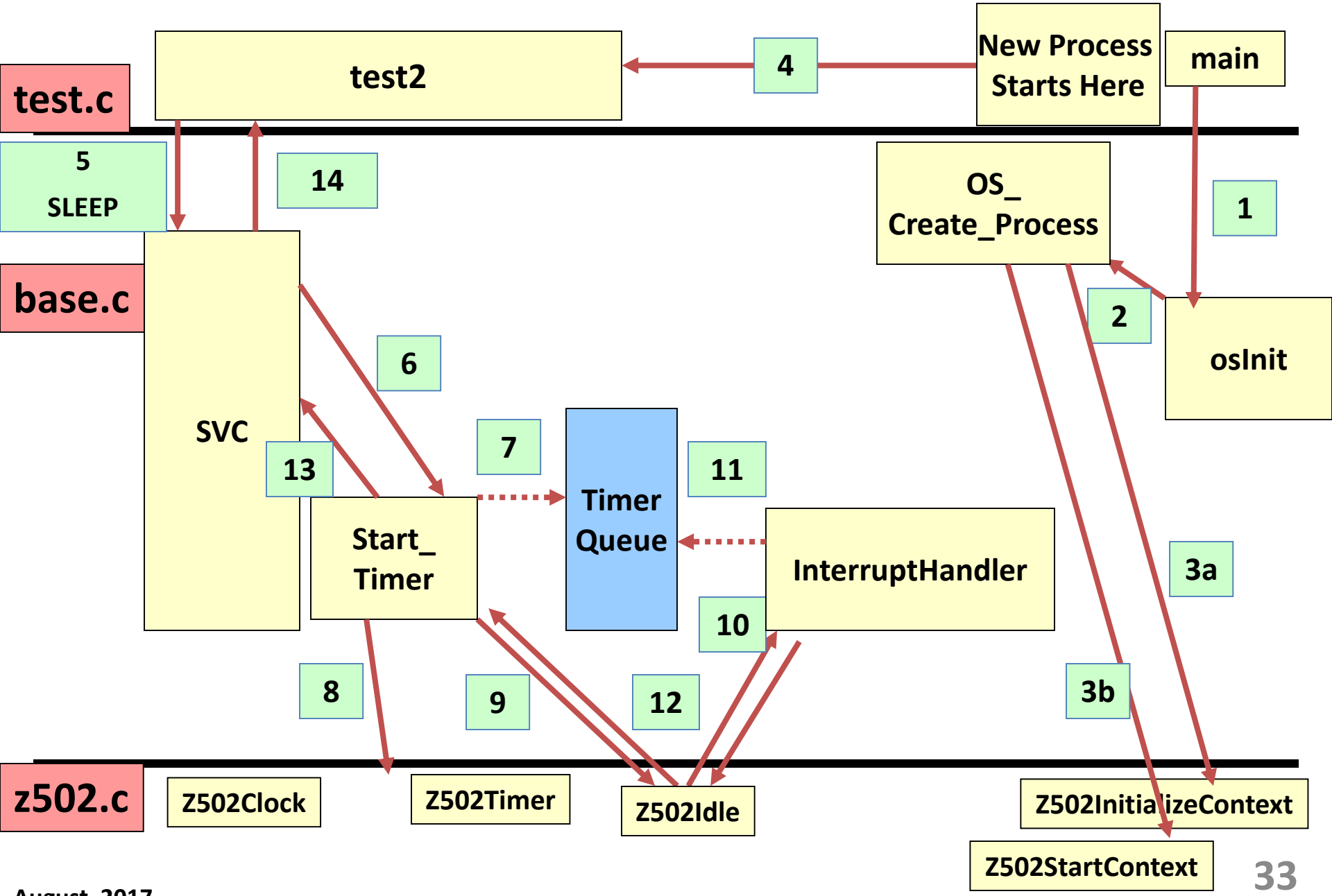
Components In The Starter Code



OS Components – What you need to Build for test 2



The Execution of test2



The Execution of test2

1. The program starts in main(), and passes control to osInit.

2. osInit figures out what test you want to run. It passes the identifier for that test to os_create_process.

3. We come to os_create_process, a routine YOU write. Here we ask the hardware for a context(Z502InitializeContext) , create the PCB, and then call Z502StartContext.

4. Z502StartContext causes control to be passed to a new thread which transfers control to test2.

5. Note: Test2 does various system calls, but we're looking only at SLEEP in this picture. Test2 does a system call SLEEP transferring control to svc.

6. svc hands control of the SLEEP request to start_timer, a routine YOU write.

7. start_timer, enqueues the PCB of the running process onto the timer_queue.

8. Start_timer calls the Z502Timer to give the request for a future interrupt. The timer starts thinking about the time, but interrupts in the future!!

9. Start_timer realizes there's nothing else to do and so calls Z502Idle. This routine says to idle the processor until an interrupt occurs.

10. Svc must handle this terminate_process request. Eventually this code will be more complicated, but for right now, since there's nothing else for the OS to do, it simply ends the simulation by halting the processor.

The Execution of test2

10 When the delay timer expires, an interrupt is generated. This causes the processor to go to the interrupt handler.

11 In the interrupt handler, take the PCB off the timer queue. This is the process that has been sleeping!

12 When you return from the InterruptHandler, execution returns back to start_timer, to the line AFTER your call to Z502Idle.

13 Start_timer returns to svc.

14 svc returns to test2.

Test5 - What you need to do

- What's the goal of Test5?
- Summary information.
- Where do you find resources to help you?

- Architecture of the Simulator Environment
- Z502 Hardware Organization and Architecture
- Generic Operating System Structure

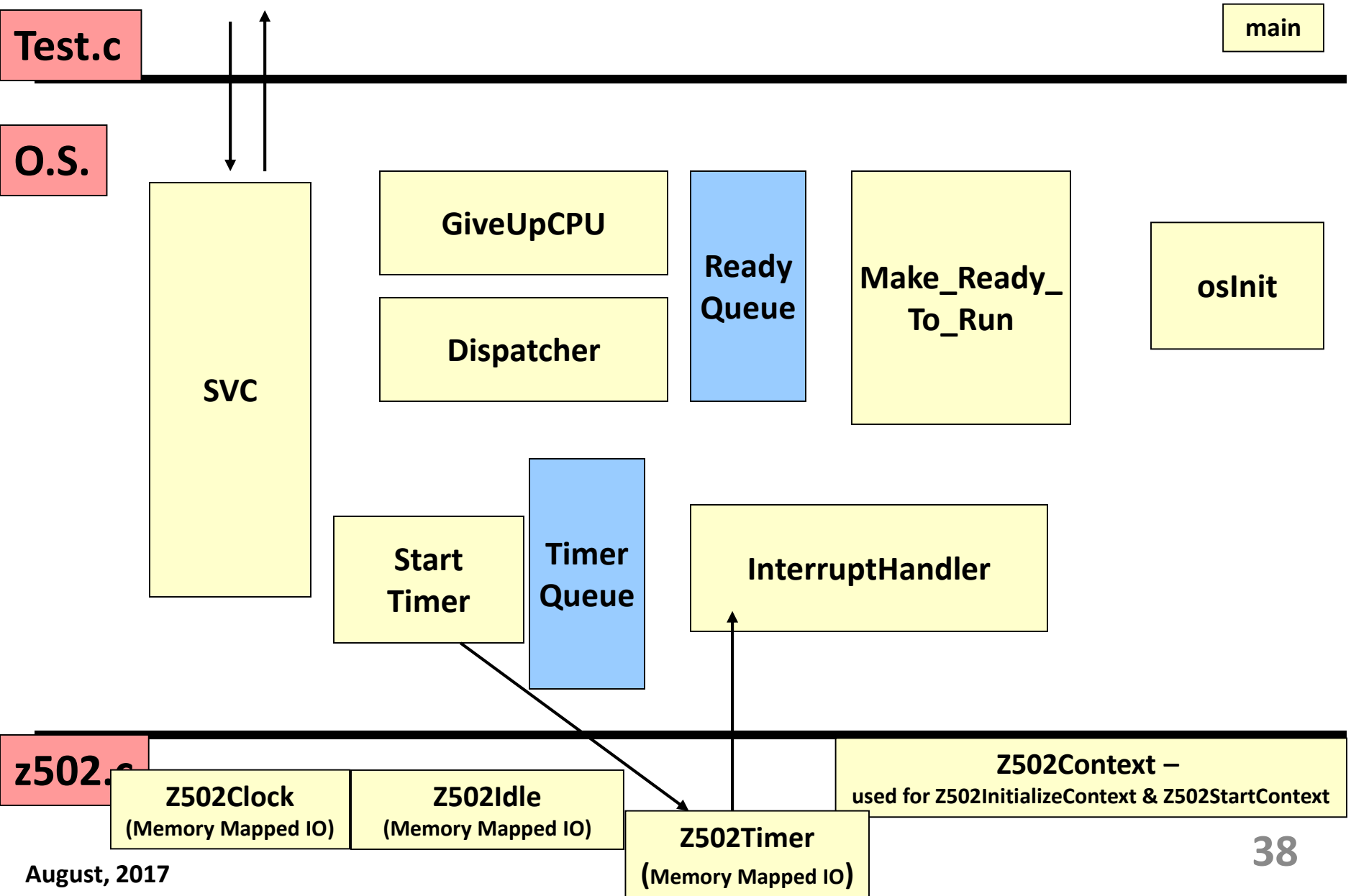
Steps For a Perfect Project

- When creating a process, put it on the ready Q but don't run it yet.
- How to use up time in the dispatcher while waiting for a process to be on the ready Q:

```
void dispatcher() {
        while( ReadyQueueFront( ) == NULL ) {
            CALL();   ← CALL is a C macro.
        }
        // Other dispatcher code
}
```

This has the effect of advancing the simulation time until the next interrupt occurs.

OS Components – What you need to Build (eventually)



Testx m - Multiprocessors

The z502 system can be run in multiprocessor mode.

Do this by executing a test including an “m” as the second argument.

```
“z502 test6 m”
```

What you must do to make this work:

1. Make your code reentrant – now multiple threads will be executing your dispatcher and other OS simultaneously.
2. In single processor mode, a StartContext assumes that the caller will be suspended – you’re using START_NEW_CONTEXT_AND_SUSPEND.
3. In Multiprocessor mode, the dispatcher starts EVERY process that’s on the Ready Q (Using START_NEW_CONTEXT_ONLY) and then when there are none present, suspends itself using SUSPEND_CURRENT_CONTEXT_ONLY.
4. The hardware provides the support you need, providing you with the current context so you can determine a process’ PID in an easy way.

Testx m - Multiprocessors

- In the mode we've been using for "single processor", what we mean by that is that there are TWO processors running; one processor is used to execute all the user processes. This processor must be shared among all the processes. So we use the flag START_NEW_CONTEXT_AND_SUSPEND to accomplish this; every time we start a new context (process) on a processor, we have to suspend the process that's currently running there. The second processor is for the exclusive use of the interrupt handler.

- This is what we call "single processor".

- In "multiple processor" mode, there are many processors to run the processes in a test.

- In the tests we have, there are more processors than processes, so we don't really need to share processors - each process can have its own processor - it's what's called "processor affinity". In real life this approach produces the best performance because the state of the process is maintained in the caches and registers of the processor.

Testx m - Multiprocessors

- Since each processor runs only one process, the hardware scheduling is concerned with keeping the processor idle (if there's nothing for the process to do - the process is waiting) or the processor is running (it's doing work for that process.)

- Initially the simulation starts up running just one processor.

- When we create a new process, we need to place it on a processor. We do that with a START_NEW_CONTEXT_ONLY -- we start the new context on its own processor, and we continue running the current context on its current processor. When we do this action, we add one more active processor.

- OK - so now we have all the processors initialized, and all containing/executing/implementing a process. When a process is waiting for an event (disk, timer), it tells its processor to SUSPEND_CURRENT_CONTEXT_ONLY. So that process ceases execution right there.

Testx m - Multiprocessors

When the event for which that process is waiting does occur, then we need to wake up that process AND that processor in order to continue execution. For that we use START_NEW_CONTEXT_ONLY since the waker-upper (probably the dispatcher) will want to continue looking on the ready queue for processes to run. Some people might want to implement a dispatcher as a separate process for use in this way.

When a process terminates, it does one last SUSPEND_CURRENT_CONTEXT_ONLY and NO ONE ever wakes it up! From the hardware's point of view, that process (and that processor) are no longer in use.

SUSPEND_CURRENT_CONTEXT_ONLY - Suspend the current context but don't start anyone
START_NEW_CONTEXT_ONLY - Start the context but don't suspend
START_NEW_CONTEXT_AND_SUSPEND - Both start the context AND suspend

Before You Hand In Your Code

What do you need to do to assure the highest possible grade?

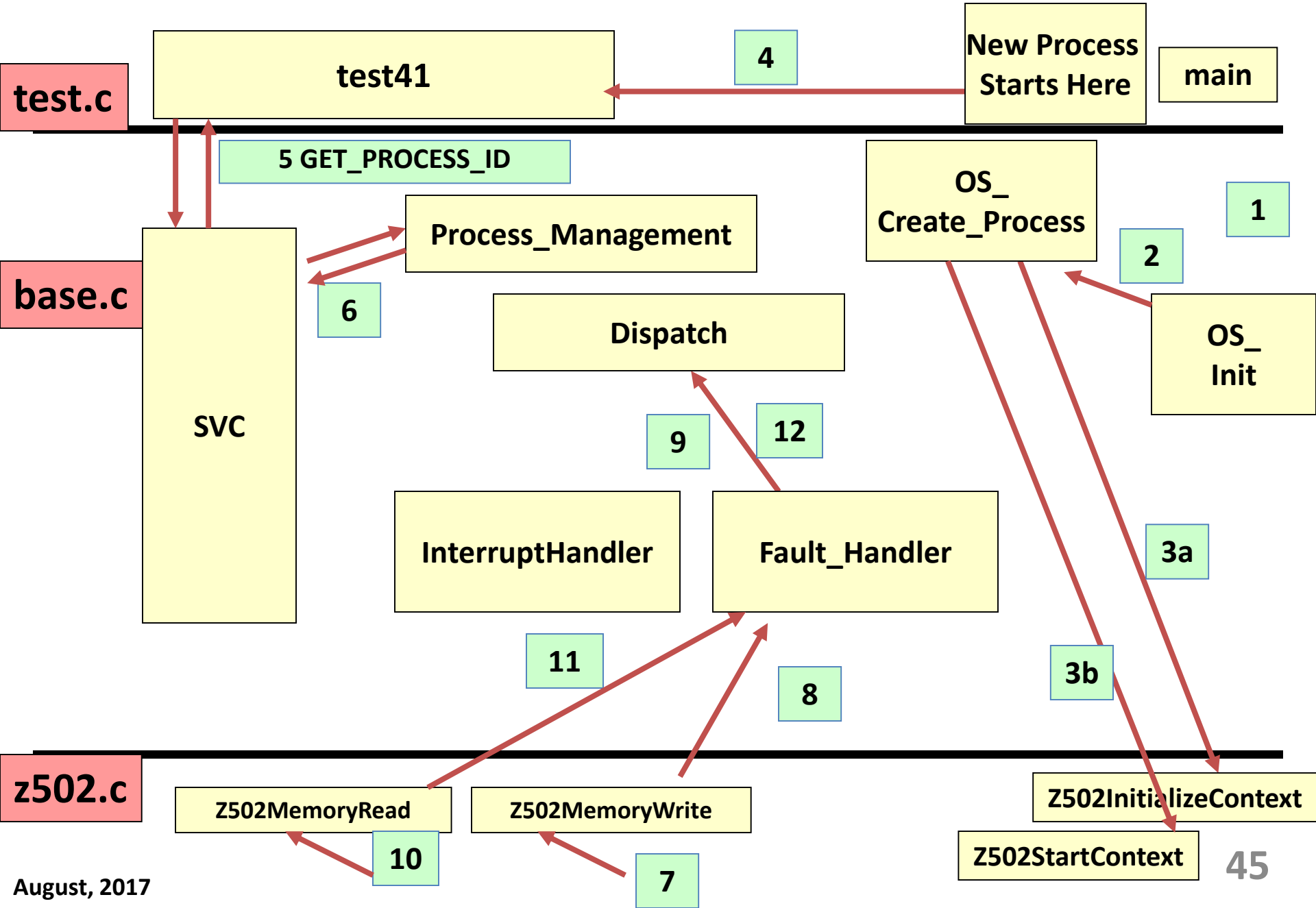
- Make sure your code is portable. I need to be able to compile your code on Linux or Windows using the command line. You will need to export your code OUT of an IDE to make sure this happens.
- I need to be able to say “z502 test7”. You will lose points if I have to recompile your code for every test.
- You must use the SchedulerPrinter correctly. The guidelines are in the Student Manual.
- Every test must finish in less than 30 seconds. If this isn’t true, it means you have too many print statements or too many sleeps.

Test41 - What you need to do

- What's the goal of Test41?
- Summary information.
- Where do you find resources to help you?

- Architecture of the Simulator Environment
- Z502 Hardware Organization and Architecture
- Generic Operating System Structure

The Execution of test41 and test42



The Execution of test41 and test42

1. The program starts in main(), and passes control to osInit.
2. osInit figures out what test you want to run. It passes the identifier for that test to os_create_process.
3. We come to os_create_process, a routine YOU write. Here we ask the hardware for a context(Z502InitializeContext) , create the PCB, and then call Z502StartContext.
4. Z502StartContext causes control to be passed to a new thread which transfers control to test21.
5. The test may do system calls as we saw in test2 – test15. The example we see here is GET_PROCESS_ID.
6. svc hands control of the system call to the appropriate handler.
7. The test does a Memory Request (either read or write). That request ends up in the hardware. If the hardware can handle it, you're done.
8. If hardware can NOT handle the call, then a page fault is generated. You do the work in your fault handler to make the memory access successful.
9. After completing the page_fault work, always call your dispatcher to schedule the same or a new process. NEVER return from the fault handler.
10. Reads and writes are handled the same way.