User Manual

Release 4.50 - Revised 7/2018

PREFACE

1. Purpose and Scope

This document contains all the information necessary to design and implement an operating system for the Z502 computer. It describes the assignment for each part of the project, the development environment, and gives complete details on the machine architecture.

After reading this material the reader should be ready to start working on his/her operating system.

2. Organization of this Document

This manual has three chapters and three appendices.

The first chapter is an Introduction that describes the project goals and introduces the machine and the development environment.

Chapters 2 and Chapter 3 describe each part of the projects. They explain in detail what should be achieved in each phase, what software has been provided, and how to test the resulting system.

The appendices contain (very important) reference material.

Z502Hardware.html describes the functional aspects of the Z502 computer architecture, that is, its interface to software.

CS502SystemCalls.pdf gives definitions of the system calls, while

CS502Output.pdf describes the state printers, SchedulerPrinter and MemoryPrinter, and their interfaces.

Z502 Student Manual gives lots of detail on implementation.

3. Revision History

The Z502 Project, Release 1.0 was written in the Summer of 1990. The inspiration for the project came from the "cSOS Project" developed at Harvard and the Wang Institute. The goals of cSOS are somewhat different from that required at WPI. In particular:

- cSOS is not portable - it's designed to be run on a particular machine, running UNIX, at Harvard. The code for the Z502 project is both compact and portable. It's easily downloadable and runs on LINUX, Windows, MACS and many other platforms.
- Students at Harvard are expected to work in groups of 2 - 3 on the project. You'll be working individually so less is expected in these assignments.
- Over the years, this project has become increasingly complex. Enjoy!!

Chapter 1

Project Introduction

1. Introduction

The course project is the design and implementation of a simple operating system for a simple hypothetical machine: The Z502 computer. You are given a simulator for the Z502 machine, along with a number of test programs that exercise the required features of your operating system. You are required to write your version of the OS502 operating system and, optionally, any special test programs that you desire.

The remainder of this document gradually introduces the components of the project and the minimum requirements of each project part of your operating system. In order to finish each project part, it is only necessary to read this document as far as the assignment for that phase, along with related reading from the course textbook, lecture notes, and the appendices of this document. You may, however, find it interesting to read through the entire document to get a perspective on the whole project, and to plan ahead for later phases.

2. Project Goals

The two project parts cover the following general areas:

1. The basics of an interrupt system in relation to a multiprocess, non-virtual-memory environment and familiarization with the simulator and with programming.
2. Virtual memory support; page fetch, replacement algorithms and shared memory; disk driver support.

Throughout the project you will have considerable flexibility in designing your own algorithms and data structures. There are also many interesting ways in which the minimum project requirements can be extended, and you are welcome to explore in any direction that interests you. When making design decisions, you will often find yourself making efficiency tradeoffs (for example, between the time required to perform an operation and the space required for the associated data structures). You should explain such decisions in your documentation. In some cases, you may be able to imagine an extremely efficient and sophisticated design that would be rather complicated to implement. You should **avoid** such solutions unless you have first **implemented** and **debugged** a straightforward solution. The amount of work

required to design, implement and debug a straightforward solution is already considerable, and you are not required to go beyond the basic requirements. In blunt terms, please resolve efficiency/simplicity tradeoffs in favor of simplicity, even if you are aware that such a choice would not be acceptable in a real operating system. If you wish, and if time and resources allow, you may consider implementing more complex designs after the minimum project is complete. This method of development should also enhance the modularity of your operating system, as once your system is working, you should be able to substitute an easy but inefficient algorithm for a more complicated design without endangering other components of your code.

3. The Z502 Machine

Your operating system will be written for the Z502 computer, a hypothetical machine designed for use in this Operating Systems course. The Z502 machine is a 32bit/64bit word microcomputer. Its advanced multi-register CPU sports numerous instructions, virtual memory support, and a single-level interrupt scheme.

The Z502's advanced architecture has several features that are of particular interest to Operating System designers: (For the details of the Z502 machine, including registers, instruction set, and memory management support, see Z502Hardware.)

1. The CPU may operate in one of two modes: **user** and **kernel**. In user mode, the machine executes the C language and user system calls as defined in CS502SystemCalls. In kernel mode, the C code that you write and link with the simulator is executed. In addition, programs running in kernel mode are able to manipulate the machine in privileged ways that programs running in user mode cannot (e.g. issue I/O instructions, change address translation tables, etc.). Specialized instructions exist in both modes to switch from one mode to another, and in kernel mode from one program to another. Since a program in kernel mode is given control of the machine, it is the logical choice for most of an operating system.
2. The Z502 includes the hardware and CPU features necessary to support two types of memory system: identity translation, and full page translation. Thus, the system can be configured for many applications. With the identity translation, programs are non-relocatable and statically loaded. With page translation, a full virtual memory operating system is possible. All references to memory addresses are interpreted as virtual addresses and are translated to physical addresses before each memory access takes place. The conversion from virtual to physical addresses is performed automatically by the hardware via a **Page Table** that is completely under the control of the operating system.
3. To allow control of processor allocation and measurement of resource consumption, the Z502 has both a **timer** and a **clock**. The timer allows the

operating system to gain control after a certain period has elapsed, while the clock allows the operating system to provide time-of-day functions as well as the time-stamping of events.

4. The Z502 comes with a disk drive as well as the timer and clock. Device integrity is protected by only allowing transactions with the devices in kernel mode, thus placing them under the control of the operating system. I/O transfers proceed in parallel with the central processor to achieve maximum processor use. Communication from the devices to the CPU is implemented via an interrupt system and specialized registers.

Because the Z502 is intended as a general purpose machine, it is available in several configuration options. The amount of memory, address translation scheme, and number of disks can all be varied. An operating system for the Z502 should be tailored for the address translation scheme, but should, of course, run with any number of disks and amount of memory.

4. The Development Environment

To aid your development of an operating system we provide a simulator (not hypothetical, we hope) for the Z502 machine on your "real" machine OS. The combination of Your "Real" Machine Operating System (YRMOS) and the simulator constitutes the environment for your development effort.

The Z502 simulator is a program that provides a thorough simulation of our hypothetical Z502 computer. Your operating system will run under the control of the simulator, while the simulator runs under the control of YRMOS. The simulator simulates privileged instructions, Z502 devices, and interrupts. Upon system startup, i.e., when the simulator is first invoked from YRMOS, the simulator performs necessary initialization (i.e., simulating hardware bootstrap) and transfers control to the entry point of the operating system (i.e., the function osInit() in the operating system). From the point of view of your operating system, the behavior of the simulator is virtually identical to that of the Z502 machine described in the previous subsection and detailed in Appendix A.

The development environment comes with test programs designed to measure your success in implementing OS502. (described in Appendix B).

5. Unrealistic Aspects of the Project

The unrealistic aspects of the project fall into two categories: hardware simplifications in the Z502 that remove certain complexities that occur in real systems, and unrealistically simple requirements for the operating system functionality.

1. *Hardware*

    1. The Z502 interrupt system guarantees that no interrupts will be missed or lost IF you design your interrupt handler correctly. In a real system lost interrupts are possible if the software is not properly designed and the latency associated with responding to an interrupt has the potential for being too long.
    2. The Z502 hardware is completely error-free (we hope). In real operating systems, a large part of the code is concerned with error recovery and diagnostics. In other words, the hardware breaks and the OS must be prepared to handle that failure.
    3. In the Z502 kernel mode (i.e. from the point of view of your code), the amount of memory available for operating system data structures can be considered to be unlimited. In a real operating system, much design is concerned with the allocation of space for operating system data areas. Operating system performance is usually strongly affected by the amount of space it consumes. In systems that do not page the kernel, less is better.

1. *Functionality*

    1. You are allowed to assume a non-hostile user environment. This allows protection and security issues to be largely ignored.
    2. You are allowed to assume that all processes require a fixed, statically allocated amount of virtual memory. If processes dynamically allocate and free space, it is from within a fixed area allocated at compile time. You may assume that the process makes no memory allocation requests to the operating system at run time. Thus, no dynamic virtual memory allocation issues need be addressed.
    3. You are not required to support a general inter-process communication mechanism, that is included in most real operating systems. Support of system calls that give a subset of inter-process communication is a part of Project 1.
    4. In order to be portable, the Z502 assumes the underlying Operating System is multitasking and may contain multiple processors. But there is no assumption about clocks, so a timing mechanism is simulated by the Z502.
        1. References to all subroutines within your OS must use the CALL mechanism explained in Appendix C.
        2. Follow these simple rules: ALWAYS just "return" from an interrupt; NEVER return from a fault, but ultimately cause a context switch, starting a different (or the same) process.

Chapter 2

Project Phase 1

1. Introduction

In this phase, you are given a skeleton source code of a simple operating system and asked to complete it. This implementation assumes many programs are running on the computer at a time. The implementation works with a version of the Z502 machine with no address translation hardware of any kind since memory management is only implemented by you in Project 2. The functions that the kernel supports in Project 1 are multiprocessing, interaction between these processes, disk IO, and some simple error handling.

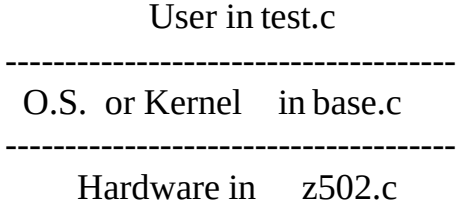
This chapter contains a vast amount of information about operating systems in general, and your system in particular. It concludes with a description of what, specifically, you are to do.

2. The Basic Interface

In Project 1 your operating system will provide several services that fall into two categories: those that the user program requests and those that the hardware signals. In fact, real OSs react in exactly this same fashion - they service user requests and hardware requests.

The Operating System is sandwiched between these two requestors - the Hardware and the User.

This is the system you'll be using:



It's also possible to run the system in "sample mode". This is a way to see how the hardware calls are implemented - the code in sample.c is well worth studying. In this mechanism the structure looks like this:

Kernel in sample.c

Hardware in Z502.c

2.1 Kernel Service Requests

As far as the code you must write for Project 1 is concerned, all services the user program can request act practically the same. Consequently, we only describe one such request in detail here.

What must a user program do when it wants to get the current time? Notice that there is no instruction in the user mode instruction set (see Appendix C) that causes hardware access directly (or any I/O for that matter). This is because hardware access is only managed by the operating system. In fact, the I/O instructions are only part of the kernel instruction set. Consequently, the ability to read the clock is considered a service that the operating system must perform on behalf of the user program.

To request the service of reading the clock, the user program communicates its request by executing a **SOFTWARE_TRAP** instruction. This is also referred to as a **SYSTEM CALL.** When the Z502 CPU encounters such an instruction, it generates a trap and switches to some operating system specified routine to handle it. Let's explain a little more about what happens when a trap occurs.

OH - note that the words "trap" and "exception" are used interchangeably both in these pages and in the OS world in general.

Whenever a trap or exception occurs, the Z502 hardware always switches control of the CPU to an exception handler routine that is part of the operating system. Which routine to switch to depends on the type of exception. The addresses of all the exception handler routines are stored in the set of **TO_VECTOR[ ]** registers. (These registers are described in Appendix A, section 3.) The TO_VECTOR is established in the osInit routine of base.c so you don't need to set it up - you just need to understand what it does.

Now before you get too frantic about the TO_VECTOR[ ], you should understand that it's all set up for you. It good for you to comprehend what it's doing for you, but you won't have to modify it at all.

When the **SOFTWARE_TRAP** exception occurs, the Z502 hardware invokes the exception handler. It finds the address of this exception handler by looking in the register/location **TO_VECTOR[TO_VECTOR_TRAP_HANDLER_ADDR]**. In your base.c code, this exception handler is the routine svc(). (The hardware also does

other things when an exception occurs. Consult Appendix A, section 6 for details.) Part of what you have to do in Project 1 is to write this exception handler routine that is invoked when a **SOFTWARE_TRAP** instruction is encountered.

What this exception handler routine must first do is determine what kind of service is being requested by the user. The operand field of the STAT_VECTOR register contains a number that indicates the service requested (a complete list is in Appendix C). You can look at the code in svc() that already exists in base.c to understand how this works.

Based on this code, the SOFTWARE_TRAP handler (svc()) should simply call the proper service handler routine to provide that service. Those service handler routines that are required in Project 1 must be written by you to support various system calls. Let's go through one of those calls in detail.

Suppose that the service requested was a SLEEP. The service handler does some argument shuffling and some poking about in its data structures and finally decides to initiate the I/O instruction. It does this by calling **DELAY_TIMER**. The DELAY_TIMER is a kernel I/O instruction that merely **starts** the I/O operation, thus when the function (the DELAY_TIMER instruction) returns, the operation has not necessarily been completed (refer to Appendix A, section 5 for a detailed description of the DELAY_TIMER). Most I/O operations occur **asynchronously** and in **parallel** with respect to the central processor. An I/O exception will be generated by the Z502 I/O circuitry at some later time, when the I/O operation has completed.

Meanwhile, the operating system has control over the CPU. The service handler now has nothing to do until the time interval completes. (The definition of the "interval timer" service is such that the USER program may assume that when it gains control over the CPU again the service has been completed.) Consequently, the service handler calls the routine

WAIT_FOR_INTERRUPT (<reason>)

where <reason> indicates what is being waited for (in this case the symbolic constant TIMER_INTERRUPT, the only possibility in Project 1). WAIT_FOR_INTERRUPT is written by you. It is a routine within the operating system that, when called, takes the CPU away from the caller, after remembering where to come back to the caller when the event indicated by <reason> does occur.

What should WAIT_FOR_INTERRUPT do then? If there is anything else that can be done while waiting for the timer to finish, then WAIT_FOR_INTERRUPT should

ensure that it gets done (waiting for the timer to finish is a perfect time to wash a few windows).

We now digress for a moment to discuss one of the most important aspects of most operating systems: sharing of resources among more than one process. When one process has to wait for any reason (e.g. for an I/O operation to complete) the operating system should allow another process to run if possible. Changing from one process to another is called a **process switch**. Switching processes involves saving all information not normally saved by a context switch, e.g. a process id number, and restoring this same information for the new process. The place where process state information is stored is often called a **process control block** or PCB. In the Z502 machine, the routine **switch_context** automatically saves all of the register information described in the context structure. You are responsible for anything else that describes the process. While one process is waiting for an I/O interrupt, another process can be switched in to make use of the idle CPU. You must be careful to save all of the information associated with the process waiting for the interrupt, so that when it finally occurs, that process can be properly restarted from where it left off. Your operating system must also be able to handle the case where a process switch from process A to process B occurs, and then another switch occurs from process B to process C, and then finally the CPU gets switched back to process A.

Remember that I/O operations may not be serviced in the order requested. Matching an I/O response with a request is a device specific task.

WAIT_FOR_INTERRUPT must determine if some other process is ready to run by calling the dispatcher. If there's no job to be done, then WAIT_FOR_INTERRUPT should call **the Memory Mapped IO instruction for Idle.** The idea is to switch the CPU into idle context, which causes neither the user program nor the operating system to execute until some external event (for instance an interrupt) occurs. Let's now make a small diversion to explain this instruction. (It is also described in Appendix A, section 5.)

Whenever the Z502 CPU is executing code, either the user's or the operating system's, we say it does so in some *context*. A context is a state of the CPU. It includes the registers, the program counter, the local variables, and the status. While in the middle of performing a task, perhaps a service for an exception or trap, it is often nice to go off and do something else while waiting for things to complete. (This is when WAIT_FOR_INTERRUPT should be called.) At some later time you would like to come back and complete the service at the appropriate time. Thus the Z502 kernel instruction set includes an instruction that lets one context put itself on hold (so to speak), later to be restarted from where it left off.

The Z502 CPU maintains a register called **Z502_REG_CURRENT_CONTEXT** that contains a pointer to an image of the context for the currently running code. It is this pointer that the operating system must save if it ever wants to restart the context after it has been stalled. To cause a context to be put on hold you should call the function **switch_context**. For example, if a context A wants to be put on hold then it should store a copy of the value of Z502_REG_CURRENT_CONTEXT (that points to its own context) somewhere before it calls **switch_context**. That way, when later on some other context, say context B, decides that it is time to resume context A then it can call **switch_context** using the value of Z502_REG_CURRENT_CONTEXT stored by context A (which points to context A). Your routines WAIT_FOR_INTERRUPT and RESUME_PROCESS should make use of the **switch_context** instruction to implement this type of processing. (RESUME_PROCESS is discussed later.) Appendix A, section 5.3, explains in detail the steps taken during a context switch, and describes the primitives to handle contexts. We encourage you to read and understand that section after reading this chapter.

At this point in our discussion of a service request we are now, in the simplest case, in the idle context with the SOFTWARE_TRAP service context currently on hold. What happens next? Well. . .

2.2 Hardware Signals

Hardware signals occur asynchronously and are termed **exceptions** or **interrupts.** You will also have to handle **software traps** that occur synchronously in software as explained above. A hardware signal causes the Z502 CPU to invoke your operating system via the routines you specify in the vector **TO_VECTOR**. We have already seen one such signal, SOFTWARE_TRAP, that occurs when the user program executes a SOFTWARE_TRAP instruction.

Another interrupt in our example is DELAY_TIMER. This happens when the requested I/O operation to the timer completes its action. For instance when the timer initiated with the **DELAY_TIMER** o-so-many paragraphs ago finishes, your operating system routine in InterruptHandler (in base.c) is invoked . You will have to supply this routine that directs what happens when the exception is received.

Your interrupt handler routine at this point must decide which device interrupted and call a handler for that device. That handler in turn determines which context is waiting for this exception and sets up that context so it may complete its service. This should be done by calling

RESUME_PROCESS (<reason>)

where <reason> is the same type of argument as for WAIT_FOR_INTERRUPT. Remember, RESUME_PROCESS is also written by you. RESUME_PROCESS must decide which context is to be resumed, what its value of Z502_REG_CURRENT_CONTEXT was and then must do the resumption. Resumption means making a process ready to run so that later on the dispatcher will run it. Once the dispatcher does run, the resumed context continues onward. Unfortunately the user process starts up at its initial entry (at the start of the routine; sorry that's one of the hacks in the Z502 hardware.)

When the operating system finally switches to the user's program's context, the user's program resumes execution assuming that its service request has been satisfied. Gee, an awful lot of things happened simply to SLEEP!

The following list shows the major steps taken to service a SOFTWARE_TRAP request. Let us explain those steps briefly to review what was said in sections 2.1 and 2.2.

1. The user's code is running and Z502_REG_CURRENT_CONTEXT points to an image of the user's context.
2. The user issues a system call that causes a SOFTWARE_TRAP. The address of the SOFTWARE_TRAP handler was obtained from TO_VECTOR[TO_VECTOR_TRAP_HANDLER_ADDR].)
3. The trap handler initiates the I/O service and calls WAIT_FOR_INTERRUPT. WAIT_FOR_INTERRUPT stores a pointer to the handler's context in a variable and calls switch_context to put the CPU into some other process' context. Z502_REG_CURRENT_CONTEXT now points to the newly-running context.
4. When the I/O service is completed, the device causes an interrupt. The device handler (its address was stored in TO_VECTOR), starts running. Z502_REG_CURRENT_CONTEXT continues to point to the context of the process that was interrupted (WATCH OUT - this could be dangerous.)
5. The device handler calls RESUME_PROCESS that puts the context on the ready queue.

3. Resource Sharing

Since more than one process can be active in the system, it's possible for more than one process to request service concurrently. Your kernel must make sure that:

1. No more than **one** request for the disk can be issued at a time; otherwise, a fatal I/O error occurs, and simulation aborts.

2. The CPU is shared equitably among ready processes. The CPU is allowed to idle **only** if there are no processes ready to run. This means for instance that given a scheduling choice of running another process or waiting around for a timer or disk to complete, you should choose running another process.

Because of the first requirement, you have to design and implement a basic synchronization mechanism and to maintain a queue of processes waiting for access to each device. For the second requirement, you have to maintain a queue of processes waiting to run on the CPU. You should use the Z502 timer to prevent any process from hogging the CPU. The DELAY_TIMER can be used to set an interval timer on user processes for cpu scheduling. A **DELAY_TIMER** interrupt is generated when the value of TIMER reaches zero. You are free to design your own dispatching policy.

One last note. While it might seem that the Z502's very high level instructions for context switching are not very realistic, they do exist on real machines. At the heart of almost all operating systems is a layer just above the hardware layer that provides some set of concepts and instructions. These extended services are often "hardware assisted" through careful design of the instruction set, and on some machines fully implemented in micro-code.

4. Schedulers

There are lots of possible variations on schedulers you could include in this Project. Required are only two:

1. First Come First Served (FCFS): services next the process that has been waiting the longest.
2. Simple priority scheduling: services next that process having the most favorable priority.

You can write other schedulers if you want, but these are required. The toughest part of designing a scheduler is in writing the testing and reporting mechanism which will assure to the reader that the schedulers are operating as expected.

3. Provided Routines

Some routines reside in the supplied code. The base operating system provided to you consists of base.c and StatePrinter.c.

The code for the Z502 simulator can be found in z502.c Read it and enjoy it if you wish. When the written instructions and descriptions differ from the code, believe the code. It is hoped, however, that you won't need to spend much time reading the

simulator code since this document should describe everything you need for your interface.

1. Initialization Routines

The operating system is initialized by a call to **OSInit**, a semi-provided routine residing in base.c. OSInit sets up the TO_VECTOR exception vector registers, and defines a single user process as the first to be run. You may well want to add to this code to get it to do more. A brief description of the initialization steps follows:

1. The TO_VECTOR exception handling vector is filled with the names of routines to call for each possible exception. (Names and routines to handle three possible exception types must be supplied by the student.)
2. Structures to support a scheduler, etc. are created.
3. A context switch is done to the user process, causing it to begin execution.

Each of these steps is conceptually very simple, but the process as a whole needs to be well understood in order to implement later Projects. See also Appendix A, section 10.

2. Termination Routines

There are two types of termination: termination of the user process and termination of the system. A process termination can be due to either a TERMINATE_PROCESS Software Trap or a process error (illegal instruction, illegal page reference, etc.). In either of these situations, an exception is generated and the handler gives control to the provided routine. It is NOT legal in a user program to simply "return" or reach the end of a routine; the simulation will end if you do so.

A call to Z502_HALT () halts the simulation. This might occur when all user contexts have been terminated, and is under OS502 control.

3. Scheduler Printer and Memory Printer

Routines has been provided that prints what's going on with the processes in various states within the scheduling mechanism. In essence, you make numerous calls to the printer manager, defining the various items you want to print. Then, when you've defined everything, you give the command to actually do the print. You should understand that this routine may be very painful for you to use as is; it's supplied so as to be an example of a detailed printout - modify it any way you wish to make it easier to fit into your implementation. Details on its usage are given in Appendix D.

4. Project Phase 1 Assignment

Your assignment, which you've already decided to accept, is to provide a design document, source code, and test document as defined in the Student Manual. The subject of these documents is described below. The correct approach here is to do what it takes to get the test programs running. For instance, test1a requires that you provide several system call handlers; as you go on to other tests you will need interrupt and fault handlers, and also a scheduler. Specific milestones include:

1. Understand the Z502 machine architecture and study the code provided for you in this phase. Pay special attention to how the SWITCH_CONTEXT kernel instruction works and what exactly the machine does when an exception occurs. You will be writing a scheduler that can do both FCFS and priority scheduling; it will be based on this SWITCH_CONTEXT instruction.
2. Modify and add to the given code to dispatch SOFTWARE_TRAP's and to deal with I/O exceptions, initializing the TO_VECTOR register elements corresponding to these exceptions with the entry names of your own handler functions You are required to write functions that will handle:
    - The hardware interrupts generated by the completion of a DELAY_TIMER.
    - The faults generated by the Z502 processor.
    - The execution of a SOFTWARE_TRAP instruction by a user program.
3. Support for the process related user system calls; the memory commands are part of Project 2.
4. Write WAIT_FOR_INTERRUPT and RESUME_PROCESS, described above, and other pieces in order to make a scheduler. You will also need a dispatcher to implement both FIFO and priority driven schedulers.
5. Run supplied tests and write some of your own.

5. Some Final Advice

Project Phase 1 requires a fairly substantial piece of implementation. It might well require several thousand lines of code (not counting blank lines or comments) in addition to the base operating system given to you in Project Phase 1. We encourage you to get started early, both on your preliminary design specification and your coding.

Chapter 3

Project Phase 2

1. Introduction

At this point, we introduce the most complex portion of the Z502 simulator: the page translation hardware. Now you can (and must) implement virtual memory support in your operating system.

If users write programs with large address spaces, the availability of main memory can very quickly become the limiting factor keeping users from accessing the computer's resources. This is rather unfortunate, since programs rarely need access to all of their address space at the same time, and more effective use of the system could be made if processes could run with only part of their address space actually resident in physical memory. The technique that allows this to take place is called **virtual memory**. Under virtual memory, address spaces are divided into blocks called **pages**. Processes are allowed to run with only some of the pages in their address spaces actually resident in the physical computer memory. As long as they only reference the code and data that is resident, there is no problem. As soon as they reference a "virtual memory" location that is not resident in physical memory, the operating system must bring in the page that was referenced, replacing some other page if necessary. This operating system function is called **paging**, and a reference to a non-resident page results in a **page fault**.

2. Paging

2.1 Page Faults

Since physical memory is limited, we now begin using the paging mechanism to implement virtual address spaces that are larger than the physical memory they occupy. The mechanism used to accomplish this is called **demand paging**. Any page that is not resident in real memory is stored in auxiliary memory (disk) and the valid bit in the corresponding page table entry is cleared. If the process references this page in any way, a page, or invalid memory, fault occurs, with the virtual page number of the page that was referenced stored as the status.

In order to resolve a page fault, before allowing the process to re-execute the instruction, the operating system must:

1. Allocate a physical memory page (called a **page frame**) for the page that was referenced. This usually involves "stealing" a page, unless there happens to be an available page frame (from a recently terminated process, or during system startup). The rule for choosing which page to steal is called the **page replacement algorithm**. The stolen page can either come from the same process (**local page replacement**) or from an arbitrary process (**global page replacement**). The algorithm tries to choose the "best" page to steal, usually one that the algorithm thinks will not be needed for a while.
2. Initiate an I/O operation to read the page from its slot on auxiliary storage into the allocated page frame.
3. Guarantee that pages that have been modified in physical memory are written out onto auxiliary storage before being stolen. This can be done either on demand when the page is needed, or periodically, as a background activity of the operating system.

Note: There are a number of subtle interactions between the demand paging mechanism and normal I/O. An I/O buffer may not have all of its pages memory resident. In this case, the operating system must simulate page faults in order to bring the pages into memory. Secondly, beware of stealing pages that are involved in a pending I/O operation. Remember that if the buffer spans a page boundary then the buffer may not be contiguous in physical memory, but must be if the I/O is to be performed in a single operation. Options include shuffling pages in memory or breaking the I/O operation up into several operations.

Numerous data structures are involved in supporting the demand paging mechanism. You should choose a reasonable page replacement policy and carefully design the data structures required to implement it efficiently.

2.2 Page Fetch Algorithms

The simplest page fetch algorithm is pure demand paging: a non-resident page is brought into memory only when a page fault occurs. Some operating systems use a prepaging algorithm; pages that are believed to be likely to be referenced are pre-fetched. Such a mechanism is not required for this project (but is an excellent additional feature option).

A related issue arises with regard to the creation of new processes. At one extreme, it is possible to use an approach in which all pages are brought into memory at the time of process creation. This can be considered a crude form of prepaging. At the other extreme, it is possible to initiate a new process without any of its pages resident in memory. The process immediately begins to demand page the memory pages requested during execution. Intermediate approaches are also possible. You should

discuss the advantages and disadvantages of whatever method you choose to implement.

3. Page Replacement Algorithms

You are free to implement any page replacement algorithm. However, you should be prepared to discuss the advantages and disadvantages of the method you choose.

The hardware mechanism that you have available to support your paging algorithm consists of the referenced bit and the changed bit in the page table entry. These bits are automatically set by the Z502 machine whenever a page is referenced or modified, respectively. You may also clear and set these bits within the operating system.

If your page replacement algorithm chooses a page that has not been modified in memory since it was last stored on auxiliary storage, it can be stolen by simply marking it as invalid in the corresponding page table entry. If it has been modified, then it must first be written out to auxiliary storage before it is available for the new page to be read in. This occurrence is not desirable, because it doubles the amount of time required to resolve a page fault. Some operating systems attempt to avoid this case by ensuring that a pool of unmodified pages always exists. Candidates for page stealing are taken preferably from this pool. The pool is maintained by periodically writing out modified pages to auxiliary storage, and then marking them as unmodified. Such a scheme is **not** required as part of the basic requirements.

4. Page Tables

Page tables are easy in our implementation. Allocate regular program memory for these, just as you've been allocating space all along. Remember, this structure has one element for each virtual page; each of these slots contains validity/modified information, as well as a pointer to physical memory. The OS502 reads and writes this structure, while the hardware only reads it. Thus both parties must agree on its contents; for this reason, the page table is a hardware-defined entity - see Appendix A, Section 8, for detailed information on this table.

There may also be a structure needed for "shadow page tables", known only to the Operating System. These structures are used to describe how to find a virtual page of memory when it's not in physical memory. Such a structure would hold disk information, for instance, indicating where to find the virtual contents should they need to be paged in.

5. Frame Tables

The Page Tables we've just discussed are used to describe how the virtual memory for each process is assigned to physical memory. In a similar fashion, we need to provide descriptors for each of the physical pages on the machine; these are used only by the Operating System, and thus are not defined as part of the machine architecture.

The frame table indicates what pages are being shared among several processes, stores information about page usage, etc. Your implementation may survive without a Frame Table, or its equivalent, but it's not likely.

6. Shared Memory Mechanism

Generally, implementations of shared memory require some structures in addition to those necessary to hold page information. There needs to be some way of connecting the physical page to each of the virtual pages as seen by the number of processes that are sharing that page. The frame table is generally used as an anchor for a linked list of structures, each of which describes the physical to virtual mapping for an individual process.

3. Disk Drives

You will need to write code to talk to the disk drives. Your goal in this project is very simple; move data back and forth between physical memory and the disk. To do this, you will need a number of structures to maintain the disk system.

1. Bit Map

You must maintain a record of what sectors on your disk are occupied; otherwise, you will try to allocate new information onto sectors that are already in use. Bit maps provide a simple way of doing this.

2. Other

You may want to have a structure that defines for each sector, the data residing in that sector. However, this may be a duplication of what's in the Shadow Page Table.

4. Provided Routines

There's one piece of code that's been given to you for this Project. memory_printer is a tool that enables visualization of what's happening to the physical memory in the system. Here's a description.

4.1 Memory Printer

A routine has been provided that prints lots of information about what's happening to the physical memory on the Z502. You set up information for memory_printer, and then call the printer to output all this data in a nice pretty form. Modify this code any way you want, but realize that it's important to have some kind of way to communicate that you have done a successful job of memory management. Details on its usage are given in Appendix D.

Note that project 2 requires the use of the **page table registers**. These registers are named Z502_REG_PAGE_TBL_ADDR and Z502_REG_PAGE_TBL_LENGTH and are described in Appendix A.

5. Project Phase 2 Assignment

In this project, you are required to implement a demand paging algorithm to support virtual memory and swapping. The minimum required implementation must include design, specification and implementation of:

1. A page fault handler.
2. A page replacement algorithm.
3. Disk allocation mechanism.
4. Shared memory mechanism.

As always, you are encouraged to choose a relatively simple method that works, and you need not feel obliged to resolve all issues in the best way.

6. Additional Features

In Phase 2, you may choose to implement additional features rather than code for test2g - you should do ONLY one or the other. Some people enjoy the structure of getting a pre-defined test to run; others prefer the independence of developing their own test; the choice is yours.

Here are some possible additional projects - you are certainly welcome to think of other areas that are of interest to you:

1. A file system (open, close, read, write). User code (you would develop test2h) has new system calls with which to call the OS. The OS then calls the hardware as necessary to implement these calls.

2. Mirrored disks. In this case, data is written to both disks and read from either of the pair. An additional possibility is to develop a RAID disk from the hardware available in the simulator.
3. Approximate LRU. Test2f generates a sequence of page references that should be an excellent test for smart page replacement algorithms. Can you produce an algorithm that uses fewer faults than FIFO?
4. Disk Seek Algorithms. The hardware simulator is defined to accept only one request per disk at a time. But what if numerous processes are making request to that same disk. You can improve the disk efficiency by the use of algorithms that reduce the seek time. But what about fairness? There are numerous interesting problems to be solved here.

These tests may NOT be simple. You not only need to write OS code, but you may require additional code in test.c that drives your OS code. You may also need code that will DISPLAY the result of your work. Demonstrating success can be as difficult as making it operate in the first place.

7. Final Advice

Project Phase 2 also requires a fair amount of code. You could easily write several thousand lines of code. Naturally, we recommend an early start.