COS431 - Operating Systems
Semester Projects (Complete) 1 through 5

Larry Latour
Fall, 2002

Project 1: The Unix System Interface

Part A - Processes:

Part A of the first project will be to explore the process abstraction in the UNIX/MINIX system interface. The following programs must be implemented in C:

1. A C program that

- first forks a "background" child process that loops "forever" and prints "it's great to be alive" every 3 seconds at the terminal,
- then loops forever, repeatedly printing "I have 42000 children and not one comes to visit".
- but has a handler that kills the child when "interrupted" by the terminal user, after which the child "apologizes" and shuts down,
- and (the handler, that is) acknowledges the child's apology with a "you're welcome" message before shutting down also.

Make use of the following UNIX/Minix system calls: **fork**, **wait**, **exit**, **signal**, **sigkill**, and **alarm**.

IMPORTANT: sketch a diagram of process interaction that reasonably "captures" the behavior of this concurrent program.

2. A C program that provides the "scaffolding" for linking (piping) together n "filter" processes to form a simple pipelined bubble sort of <= n values. n **must** be an input parameter of your program. A filter is straightforward - it is a C program that reads from standard input and writes to standard output.

The following are input/output specifications for the overall bubble sorter:

INPUT:

A sequence of lines from standard input. Each line is a sequence of integers separated by one or more blanks. Each integer is a sequence of digits.

OUTPUT: Output is a sequence of lines where:

- as many input integers are "packed" onto an output line as possible, each integer separated by at least one blank.
- output lines are 30 characters in length.
- the integers are in ascending order.

The first filter should input lines and output integers, the second through (n-2)nd filter should sort the data and output a stream of integers, 1 integer to a line, the (n-1)st filter should input integers and output unjustified lines of integers, and (optional) the nth filter should input unjustified lines of integers and output justified lines of integers.

The C scaffolding program must

- input the filter names of the filters in your pipeline,
- see that the proper number of processes are forked,
- create pipes to connect the filters, and
- redirect the filter I/O to the pipes.

Make use of at least the following UNIX/Minix system calls: **fork**, **pipe**, **dup** (or **dup2**), and **execl**, a UNIX variant of **execve**.

HINT: most (if not all) of the middle sorting filters can be "instantiated" from 1 basic filter "template".

IMPORTANT: Again, sketch a diagram that captures the behavior of this concurrent program. Is it more or less "complicated" than program 1?

Part B - Files:

3. There is a way to pass parameters to a C program when that program is executed directly from the Shell command line. Figure out how to do it, and explain the technique in your HW. It is described in a number of popular texts on C programming in Unix, including chapter five of the classic Kernighan and Ritchie C text *The C Programming Language* (I have a copy in my office). Use this technique in each of the following programs.

4. Write a C program to create a new link to a previously created file. Enter the simple filename for both the source and target files as command line parameters. Use the **link** system call.

5. Write a C program *CopyAlias* to determine whether or not two filenames refer to two distinct files but have the same contents.

- If they do refer to distinct files, but have the come contents, then delete one of them and alias (**link**) it with the other.

- If they are actually aliases of the same file, delete one alias and create a separate copy of the file.

Figure out what system calls are needed here.

Project 2: A Virtual Machine Interpretor for BRAIN02

Your task is to implement a virtual machine interpretor for the BRAIN02 machine in C, and then test it with at least 3 non-trivial programs written in BRAIN02 and portable to other interpretors in the class. I will randomly choose student programs to test that this portability holds.

The Virtual Machine

The virtual machine viewed by a user program is illustrated in **figure 1**. It first appeared as an assignment in *The Logical Design of Operating Systems*, by Alan C. Shaw, Prentice-Hall, 1974. Storage consists of a maximum of 100 words, addressed from 00 to 99; each word is divided into four one-byte units, where a byte may contain any character acceptable by the host machine. The CPU has three registers of interest: a four-byte general register R, a one-byte "Boolean" toggle C, which may contain either `T' (true) or `F' (false), and a two-byte instruction counter IC.

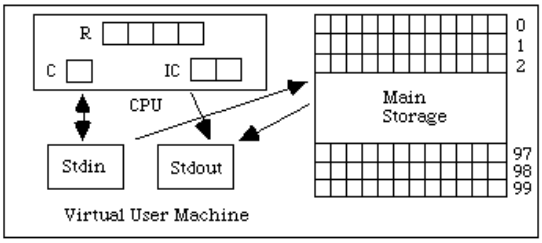

figure 1

A storage word may be interpreted as an instruction or data word. The operation code of an instruction occupies the two high-order bytes of the word, and the operand address appears in the two low-order bytes. **figure 2** gives the format and interpretation of each instruction. Note that the input instruction (GD) reads 10 data items into 10 successive storage locations, and that the output instruction prints 10 storage locations at a time. The first instruction of a program must always appear in location 00, which the virtual machine will assume is the first program instruction.

Instruction Set of Virtual Machine

The following is the instruction set for your BRAIN02 virtual machine. You may not, **under any circumstances**, add additional instructions to the set.

Operator	Operand	Interpretation
LR	x1x2	R := [a];
LL	x1x2	R := zz[a]3[a]4;
LH	x1x2	R := [a]1[a]2zz
SR	x1x2	[a] := R;
CE	x1x2	if R = [a] then C := 'T' else C := 'F';
CL	x1x2	if R < [a] then C := 'T' else C := 'F';
BT	x1x2	if C = 'T' then IC := a;
BU	x1x2	IC := a;
GD	x1x2	Read([b+i], i = 0,...,9);
PD	x1x2	Print([b+i], i = 0,...,9);
AD	x1x2	R := R + [a];
SU	x1x2	R := R - [a], R >= [a];
MU	x1x2	R := R * [a];
DI	x1x2	R := R / [a], [a] > 0;
NP	x1x2	null operation (no-op);
H		halt;

figure 2

Notes

1. x1 and x2 are digits in the range 0-9
2. a = 10x1 + x2
3. [a] means "the contents of location a"
4. [a]i means "the contents of the ith position of location a"
5. b = 10x1
6. z = the contents of the register byte (positional) before execution of the instruction

Arithmetic Operations (AD,SU,MU,DI)

1. An arithmetic op on non-numeric data fields is *undefined* .
2. Divide by zero is *undefined*.
3. A negative result (for SU) is *undefined*.
4. Overflow values (for AD and MU) are lost.

Important: *Undefined* behavior can be handled by the implementor in any reasonable way. This is course means that programs with *undefined* behavior most likely will not be portable. There might very well be other unforseen undefined behavior in this language specification, which will undoubtedly surface during the semester and will most certainly cause portability problems.

Enter instructions one per line, starting in the first character of the line, with no blanks between the operator and operand. After at least one blank, a non-interpretable comment may complete the line.

Form of a BRAIN02 Program file:

Line 1: **BRAIN02**
Note: **BRAIN02** starts in the first character of the input line.
Lines 2 - (n-1):
each of these lines contains information to be loaded into a BRAIN02 virtual memory storage location. Line 2 is loaded into storage location 0, line 3 into location 1, and so on, until the DATA line is reached.
Line n: **DATA**
Note: **DATA** starts in the first character of the input line.
Lines n+1 - m:
each of these lines contains 10 input items to be loaded into main storage by the GD (Read) instruction
Line m+1 ("last line"): **END**
Note: **END** starts in the first character of the input line.
Subsequent lines:
as many more sets of the above lines as the user would like.

Project 3: The Execution of Parallel BRAIN02 Programs

Project 2 required that you implement a virtual machine simulator for the **BRAIN02** language. Project 3 requires that you implement the underlying "virtual physical machine" and the kernel that will allow you to run up to ten preloaded BRAIN02 programs in parallel. Included in this parallel implementation are facilities for message passing, and facilities for accessing a special shared storage area using semaphores.

You are required to write this program in C/C++ BUT you may write the program on any platform (gcc/g++ on Unix, Borland C++ or Visual C++, Metroworks Codewarrior, etc.). A diagram of the virtual and physical virtual machines appears in **figure 3** below:

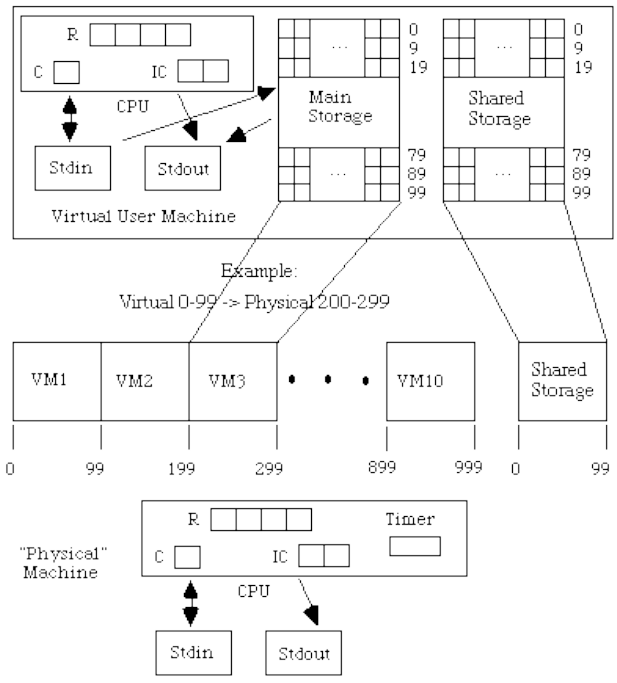

figure 3

Input Data Format:

Since you may pre-load up to 10 BRAIN02 programs, the input "data" (BRAIN02 programs and data that THEY will read) should be in the following format:

```
BRAIN02 (start of first BRAIN02 program)
info for storage location 0
info for storage location 1
.
.
.
info for storage location < 99
BRAIN02 (start of second BRAIN02 program)
info for storage location 0
info for storage location 1
.
.
.
info for storage location < 99
BRAIN02 (start of third BRAIN02 program)
.
.
.
BRAIN02 (start of nth BRAIN02 program, where n <= 10)
info for storage location 0
info for storage location 1
.
.
.
info for storage location < 99
DATA
data lines in same format as in project 2
END
```

Important: The process id of each BRAIN02 program is assigned in the order in which the program is read in above, starting from 0 (so n BRAIN02 programs will be numbered 0 through n-1).

Message Passing Instructions for BRAIN02 Process Communication:

Add the following additional message passing instructions to your BRAIN02 virtual machine.

1. Add the two communication operations:

    SD x1x2

    RC x1x2

where SD sends a message to the BRAIN02 process with process id x1x2, and RC receives a message from the BRAIN02 process with process id x1x2. The special command RCXX is a "receive from any process" command.

For a send message SD, the source of the message should be in the ten memory locations including and following the memory address in the register R, and for a receive message RC, the target of the message is defined similarly.

These operations should be blocking sends and receives respectively. That is, the sender should be blocked until the receiver receives the message, and the receive likewise should be blocked until the sender send the message.

2. Add the following "identification" operation:

GP x1x2

GP puts the process id of the process into register R.

Shared Storage and a Semaphore Facility for Shared Memory Synchronization.

Create a special 100 location shared storage area that can be accessed by any process using only the following two new operations:

Operator	Operand	Interpretation
LS	**x1x2**	**R := [a];**
SS	**x1x2**	**[a] := R;**

LS stands for "load shared", and SS stands for "store shared". These operations are executed atomically.

In addition to the shared memory area, create an array of 100 semaphores numbered 00 - 99 and always available to any process. Provide the following two operations to operate on these semaphores:

PE x1x2

VE x1x2

SI x1x2

PE acts atomically as a P, or "down" operation on the semaphore x1x2, VE acts as a V, or "up" operation, and SI initializes the designated semaphore (x1x2) to the current value of the register R. By default all semaphores have an initial value of 1.

BRAIN02 Process Scheduling

The intention is that BRAIN02 process scheduling should be similar to the model of process scheduling presented in class. That is, ready BRAIN02 processes should be linked up in a circular queue (the ready list) and chosen one after another in a *round robin* manner.

A question arose in class as to how and when the kernel would switch between BRAIN02 processes. The answer was *in two ways*.

1. A send, receive, P, or V operation would give the kernel not only the opportunity to record the sending/receipt of a message or semaphore operation, but also the opportunity to *context switch* to another BRAIN02 process, the next ready process in the ready list.
2. A process would *time out* after running for a fixed *time slice.* The difference between your simulator and an actual kernel is that you will not be using real time with a real clock, but simulated time with a clock that "ticks" once for every BRAIN02 instruction executed. You decide the number of clock "ticks" that make up a time slice. Vary this number keeping everything else the same in order to see how the parallel execution changes.

Translating Program 2 to Program 3

Do you have to rewrite your program 2 for program 3? The answer is "yes, to an extent", **but** you should be able to encapsulate the virtual machine accesses inside procedure calls, after which only the procedure implementations have to change. For example, if you addressed your 100 locations of memory with direct memory accesses such as:

MEM[99] = R and

R = MEM[30]

You would then have to rewrite these calls as:

PHYSICAL_MEM[99 + offset] = R and

R = PHYSICAL_MEM[30 + offset]

where offset is the position (0,100,200,300,etc.) of the currently executing BRAIN02 process.

On the other hand, if you *encapsulated* this memory access inside procedure and function calls, the calls might look like this in both cases:

STORE_MEM(99,R) where the value of R is stored in virtual location 99, and

R = LOAD_MEM(30) where R gets the value of the function LOAD_MEM from virtual location 30.

In these cases the value of offset can be a global variable that is managed by the kernel and that corresponds to the memory map stored in each process slot of the kernel discussed in class.

It is **very much** worthwhile to think first about the virtual machine implementation, **and then** think about how such a virtual machine runs on the physical machine. For this reason I broke the problem into 2 and 3.

In each of these cases, don't get carried away. We have lots of other things to do.

BRAIN02 Program Implementation and Analysis:

Implement and analyze the following BRAIN02 programs:

1. Using message passing, implement a series of filters connected in a "pipeline" architecture. One choice of such a pipeline might be a parallel bubblesort. Run your implementation with a variety of timeslices, demonstrating graphically how the amount of overhead (context switching) varies. You may output your numerical results to an external graphing program, or you can write your own graphing program (but be aware of time limitations).
2. Using message passing, implement both a "non-solution" and "solution" to the dining philosophers problem. Run your algorithm with a variety of timeslices, graphing and analyzing your observations.
3. Using semaphores, re-implement both a "non-solution" and "solution" to the dining philosophers problem. Again run your algorithm with a variety of timeslices, graphing and analyzing your observations, and comparing them to the observations of the message passing implementation.
4. In both the message passing and semaphore versions of the "non-solution" to the dining philosopher problem, you might or might not achieve deadlock. Document the conditions under which deadlock does or does not occur.
5. Compare the number of context switches in your message passing and semaphore "solutions" to the dining philosopher problem. Is there a discrepancy? Why?/Why not?
6. Using semaphores, implement the producer/consumer problem. Analyze how your program behaves with varying timeslices and varying buffer sizes.
7. Using message passing, implement one or more I/O server processes. It is important that there be only one input "stream" and one output "stream", accessible by all of your BRAIN02 processes. This is very much like the one stdin and stdout file available to a parent and its children in Unix. You have already seen what happens when both a parent and child write to stdout - output is interleaved.

   Attach a unique process number to each output line, designating which process is printing which output. Designate a special process to handle input, another to handle output, and then require any process who wants to do input and output to send input and/or output messages to these "server" processes. These processes may then do a limited amount of buffering.

Project 4: Dynamic Process Creation and Memory Management

Your task is to modify project 3 to allow for the dynamic creation and deletion of BRAIN02 processes that use a prespecified address space from addresses 0 up to n.

Change in BRAIN02 Program Format

Recall that you entered BRAIN02 programs in project 3 in batch format. That is, you entered your BRAIN02 programs all at once from standard input before starting execution. For this project, store each BRAIN02 program on a disk file, and include at the beginning of this file a requested upper bound on your address space. This value may be any value from 0 up to the value of the upper bound on your physical address space.

Addition of "Spawn" instruction

Add a SPAWN instruction to your BRAIN02 interpretor.

```
SP x1x2
```

where x1x2 is a two character filename residing locally with respect to the BRAIN02 executable.

This instruction will retrieve a named BRAIN02 program from disk, allocate memory for it, load it, and add it to the ready processes. Upon normal completion of the spawn operation, the parent R = the process ID of the child and the child R = the process ID of the parent. If no memory is available, then R = 0 upon spawn completion. **Note**: this means that the system doesn't halt if a process cannot be spawned. Instead, the process that is attempting to spawn is "told" of the problem (with an R = 0).

To test the "R=0" feature, create a BRAIN02 program RR that spawns itself as follows:

```
SPRR
SPRR
SPRR
SPRR
H
```

This program has very interesting *(emergent?)* characteristics, which can vary depending on how you implement your scheduler:

- There is a possibility inherent in **some** scheduler (not necessarily in yours) that this parallel program will halt. Give a rationale for such a situation arising.
- Keep track of the number and pattern of successful and unsuccessful spawn operations, and keep track of the memory location of successfully spawned processes. What do you notice?

Memory Management Protocol

Implement your memory management scheme using two different memory allocation schemes (i.e., first fit, next fit, worst fit, best fit, etc...). You can do this with two different implementations, or by integrating them into one implementation and allowing the user to choose which method to use when the system is "booted up".

- Design a mix of BRAIN02 programs that works well with respect to each of the schemes, but worse with respect to the other.
- Provide mechanisms to properly monitor how memory is allocated and deallocated. These may be textual or graphical in nature.
- Carefully explain your observations.

Project 5: Virtual Memory Management

Break your internal (RAM) memory into n m-location page frames, where n*m can be any value >= 1, not necessarily 1000. Allocate your programs in m location blocks in "external" memory. For purposes of your project, "external" memory should actually be a separate internal memory data structure. The purpose of project 5 is primarily to monitor page faults.

n and m should be parameters of your system to test the effect of varying page frame size. For example choose n=200 page frames and m=5 locations per page frame, then choose n=100 and m=10. Re-implement your memory management scheme of **project 4** using a virtual memory scheme discussed in class.

- Document your observations on how your system behaves under various values of m and n. Are some BRAIN02 programs more resiliant to changes in m and n than others? Why?
- It has been observed that the number of instructions executed between page faults is directly proportional to the number of page frames allocated to a program. If the available memory is doubled, the mean interval between page faults is also doubled. Use your implementation to verify this observation.
- Does choosing m=1 and n=1 cause any problems with your system? Why? Why not?

Reporting Responsibilities - All Projects

- **Individual Project (Project 1)**
  - Each student is responsible for implementing and demonstrating all aspects of project 1 individually.

- **Team Project (Projects 2 through 5)**
  - Each student will be responsible for implementing **projects 2 through 5** with a fellow classmate. Each team must demonstrate a working version of each step of their project in the instructors office in a timely manner, and must deliver project 5 as a working system on the public presentation day (in the Neville PC and Workstation cluster, during the last class period).

- **Deliverables**:
  - a working system.
  - documentation describing the group's efforts to significantly address each step in the project (projects 2 through 5).
  - a 22x28 inch poster summarizing the information included in the documentation, for presentation on the public presentation day.

Last Updated: 9/03/02