CS 111

Scribe Notes for 4/11/05

by Marina Cholakyan, Hyduke Noshadi, Sepehr Sahba and Young Cha

Processes

What is a process?

A process is a *running instance* of a program. The Web browser you're using to read these notes, for example, is a program: a sequence of machine instructions. The program becomes a process when it starts running, and develops some execution state. A computer can have multiple *processes* running the same *program* simultaneously.

A process has a *program counter* (or, equivalently, *instruction pointer*) specifying the next instruction to execute, and a set of other resources too:

- A program counter
- Contents of registers
- A stack (Local Variables, Temporary Data)
- Data section (Global Variables)
- Heap (memory allocated dynamically at run time with `malloc` or similar functions)

In our first lecture, we took a look at a simple C program that accomplished two tasks -- counting to five, and printing "Hello". Looking at this program as an "operating system" running two "processes", we can find several of these process features. (The "kernel" is in black in this figure; the two processes in orange and purple.)

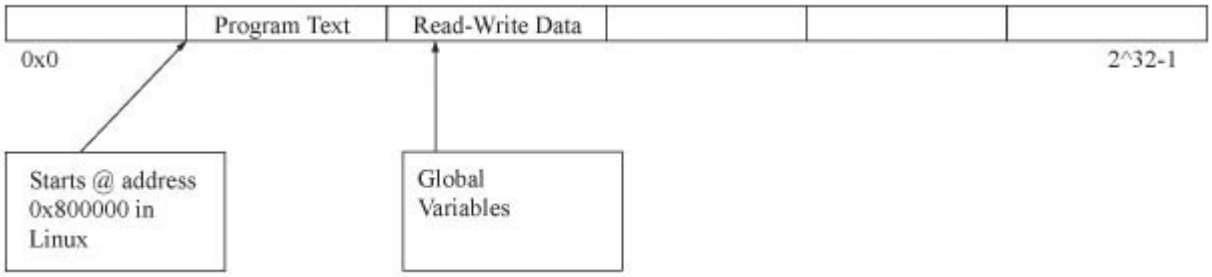
```
int main (int argc,  char *argv[])
{
    int state = 0;                  // Kernel state: Current Process
    int x = 0;                      // 1st process Global Variable
    int printed = 0;                // 2nd process Global Variable
    while (1) {
        if (state % 2 == 0) {
            if (x < 5)              // 1st process code
                  x++;
        } else {
            if (!printed) {         // 2nd process code
                printf("Hello");
                printed = 1;
            }
        }
        state++;                    // Round-robin "process scheduler"
```

```
    }
}
```

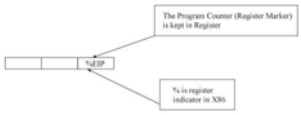
We can also compare a process with a Turing Machine.

Turing Machine	Process State
Tape	Read-Write Data (Global Variables)
Transition Function	Instructions (Program text) which is read only
Current Turing Machine State	Instruction Pointer (IP)

Memory Layout of the Process in x86



Registers



Stack (LastInFirstOut)

We need to keep local variables that are used during the running program somewhere in memory. Where can we keep Local Variables in memory? To place Local Variables where Global Variables are kept is a bad idea. Here is why.

Thought Experiment: Let's treat local variables like global variables. That is, every function will reserve some space in the global variable area (Read-Write Data) for its local variables.

What's wrong? Recursion: Functions that can call themselves! When a function calls itself (see below for an example), the function is effectively running twice, and each copy needs its own local variables. Thus, we need to allocate the function space in the different location. All modern machines do this with a *stack*.

The *stack* is an area of memory reserved for function arguments and local variables. The stack is allocated a function at a time: when we call a function, we *push* more space onto the stack to hold that function's local variables; and when the function returns, we *pop* its local variable space off the stack. The architecture has a special register, called the *stack pointer* (`%esp` on x86), that points to the current stack location. Stacks generally grow from the top down, so when a function is called, the stack pointer is set to a *smaller* value.

Let's take a look at a simple recursive function -- namely, Factorial -- and its pseudo-assembly code.

```
... factorial(5); ...
```

```
// Call 'factorial(5)'
0x02: pushl $5
0x05: call _factorial
0x06: popl %eax              // pop the
argument we pushed
0x09: ... continue ...
```

```
int factorial(int f) {
    if (f == 0)

        return 1;

    else
        return f *
factorial(f - 1);
}
```

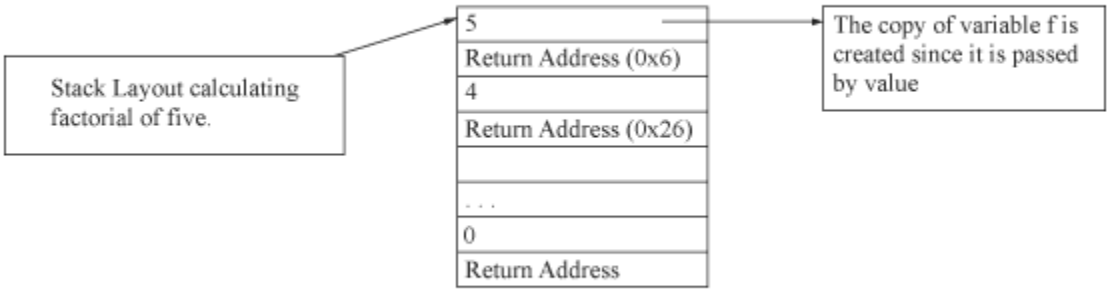
```
_factorial:     // The 'factorial' function
definition
0x10: cmpl 4(%esp), $0       // Is f == 0?
0x12: jne 0x1D               // If not, jump
ahead
0x14: movl $1, %eax          // Return 1
0x17: ret
0x1D: movl 4(%esp), %eax     // %eax = f;
0x20: subl %eax, $1          // %eax--;
0x22: pushl %eax             // push 'f - 1'
as argument
0x24: call _factorial        // call
'factorial' recursively
                             // result is
returned in '%eax'
0x26: mull 4(%esp), %eax     // %eax *= f;
                             // now %eax == f
* factorial(f - 1)
0x29: ret                    // So return!
```

When the function is called:

1. Any function arguments are pushed on the Stack. (In some architectures, the first couple arguments are stored in registers; but on x86, everything goes on the stack.) See the `pushl` instruction at `0x02`.
2. The return address (the next instruction's address from where the function was called) is pushed on the Stack.
3. The processor jumps to the start of the function code. On x86, steps 2 and 3 are combined into one instruction, `call` (see instructions `0x05` and `0x24`). The `call` instruction pushes the return address and jumps to a function atomically.

4. The function pushes space for its local variables onto the stack. (In our example, `factorial` has no local variables, so it doesn't need to do this.)
5. Within the function code, arguments and local variables are referred to using *stack-indirect addressing*. For example, in `factorial`, the address "`4(%esp)`" refers to the function's argument `f`. Why? The last thing pushed onto the stack was the return address (Step 2), so the stack pointer points there. Four bytes above that address -- at `4(%esp)` -- is the first argument.
6. When the function is done, it pops its local variable space off the stack.
7. Then it pops the return address from the stack, and jumps to that address. In x86, the `ret` instruction does this, and the function's return value is stored in the `%eax` register.
8. Finally, the caller pops off any arguments it pushed.

These steps are repeated for reach function call.

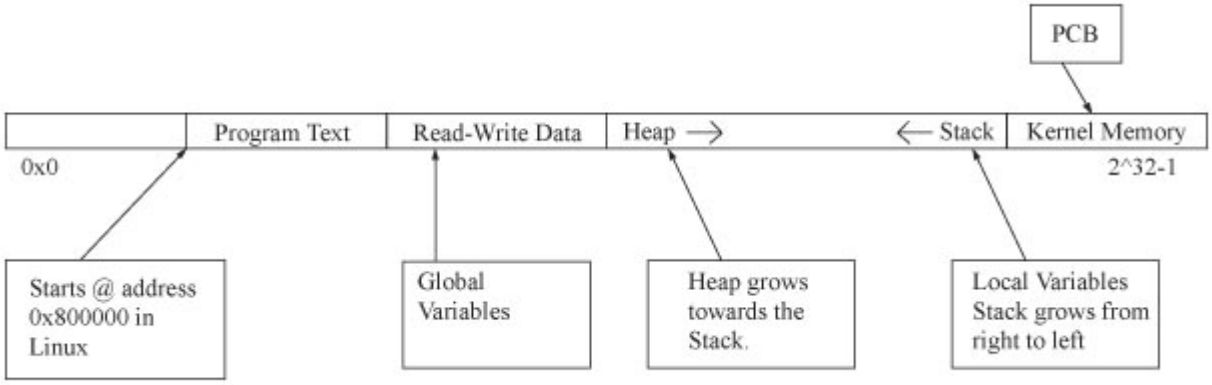


Why do stacks grow downwards? Because it is much more natural and convenient to refer to arguments and local variables with *positive offsets*, such as `4(%esp)`.

The stack assumes that function never returns more than once. It is a feature that almost every programming language has. (There are functions in Scheme that could return more than once.) The reason can be described as the following: When a function returns all the local variables and also the state that the function was in it will be deleted. Therefore that functions state and variables will not exist any more.

Heap (Dynamic Memory Allocation)

Memory Layout (4 GB)

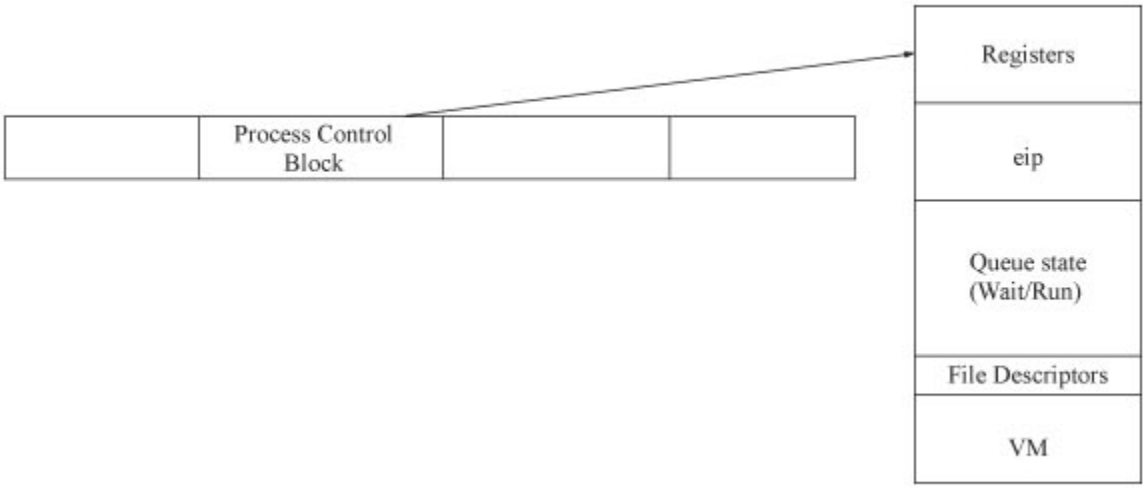


Let's introduce another section of memory which is called heap. In the heap we store dynamically allocated memories. Dynamic memory is being used to limit the process resource usage dynamically based on availability. We use the following commands to allocate and free memory.

- Malloc( ) ' Memory allocation.
- Free( ) ' Free Memory

In edition we also have to store the register state and the program counter. The main reason is that we must be able to run more than one process at a time. Let's introduce the Process Control Block. It is located in the kernel memory and has the following structure

Kernel Memory (Process Control Block)



Scheduling and Processes

What is a scheduler?
Piece of OS that determines: 1) what process is running now? 2) What process will it run next?

RUN QUEUE: Set of processes that are ready to go. "Ready to go" status does not mean that the process is 'useful'. We generally want a small run queue (minimal amount of programs on it).

WAIT QUEUE: A queue holding processes that are waiting for some event to happen. When the event happens, the process will be moved to the run queue so it can run. There are many wait queues, one per event that a process can wait for. For instance, we can have a wait queue for the following: 1) Other processes to finish 2) Input devices 3) Output devices 4) Inter-process communication 5) Timers, and so forth.

Goal of RUN QUEUE and WAIT QUEUE:
Take processes off run queue until they DEFINITELY have work to do with a MINIMUM wait time. This is to maximize efficiency and the available resources.

Blocking vs. Non-Blocking (System Calls)

Blocking: doesn't return until finished (forces process to wait until done). A blocking system call might put a process on a wait queue, if the operation can't complete right away. Non-blocking: returns an error, such as EAGAIN, if the system can't complete the system call right away. This doesn't force the process to wait. Non-blocking system calls never cause a process to be put on a wait queue. This lets the process itself decide how to react if an operation can't complete right away. For instance, the process might go ahead and try to do something else.

Input Devices

Examples of Input Devices: disk, keyboard, joystick... CLOCK: interrupt processor N times a second, kernel takes control.

Timer Wait Queue

There are many ways to implement run queues and wait queues, from simple linked lists to complicated ring data structures and so forth. To get a feeling, let's take a look at how a timer wait queue might be implemented. A process goes onto the timer wait queue to wait until a certain time -- maybe 1 second from now, maybe 20 hours from now. Different processes on the timer wait queue are waiting for different times. How might we implement this?

- Every process on the timer queue could store the number of clock ticks remaining until the wakeup time. Then, on every clock tick, the kernel would decrement this number for every process on the timer queue, and wake up any process with number <= 0. What's wrong with this method? This would take too long because we would have to modify every process on the queue, for every clock tick.
- How could we make this faster? Well, we could store an additional variable, `mindelay`: the minimum number of clock ticks remaining, for any process on the wait queue. Then on most clock ticks, we just decrement `mindelay`; we only need to walk the list when

`mindelay <= 0`. This is an improvement, but it still requires that we walk the whole list from time to time.
- Instead of this, we could store each process's *absolute* wake time. Then we could keep the timer queue sorted in increasing order of wake time. When a clock tick happens, we walk the timer queue, starting with the lowest wake time. If a process's wake time is <= the current time, we wake up the process and keep going. But if the wake time is > the current time, we can *stop walking the list*: since the queue is kept sorted, no one further down the list needs to wake up. Thus, we avoid ever walking the whole list on timer interrupt. There's a tradeoff here, of course, since we *do* need to keep the list in sorted order. Real operating systems use interesting data structures to get the best of both worlds.

Context Switch

Switching the CPU to another process requires saving the state of the old process and loading the saved state of the new process. When a context switch occurs, the Kernel saves the context of the old process in its Process Control Block and loads the saved context of the new process scheduled to run. Context Switch time is pure overhead, because the system does no useful work while switching. Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of any special instructions (such as a single instruction to load or store all the registers). Typical speeds are less than 10 milliseconds, but tens or hundreds of times more expensive than simple function calls.

Requirements:

- Context Switch requires setting up the processor's state to setup the environment for the destination process
- Context Switch requires saving the source process state which includes: Registers, IP, Virtual Memory State, Privilege Levels

The cheapest Context Switch is 100 times more expensive than a function call. Because of huge extra overhead Context Switching has become such a performance bottleneck that programmers are using alternative structures (threads) to speed it up and possibly avoid it, whenever possible.

Creating Processes

Fork System Call

A process may be create several new processes, via a create-process system call, during the course of execution. The creating process is called a Parent process, and new processes are called the Children of that process. Each of these new processes may in turn create other processes forming a tree of processes. In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of resources of the parent process.
When a process creates a new process, two possibilities exist in terms of execution.

- A new process can be created by fork() system call and identified by its process identifier. Every created process has a unique identifier. The created process (Child Process) consists of a copy of the address space of the original process. - The returned code for the fork() is zero for the Child process, whereas the nonzero process identifier of the child is returned to the parent. Both the parent and the child will continue their execution at the same instruction pointer.
- Bootstrapping Process by exec() System call: This system call is used after fork() system call by one of the two processes to replace the processes memory space with a new program. The exec() system call loads a binary file into memory (destroying the memory image of the program containing the exec () system call) and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways

How To Use Fork?

```
pid_t p = fork();
if (p < 0)
    printf("Error\n");
else if (p == 0)
    printf("Child\n");
else /* p > 0 */
    printf("Parent of %d\n", p);
```

One interesting way to think about `fork` and `exec` is to think about how a process can find a *consistent state* to start running. The operating system can't just randomly copy a process, then jump wildly into its program code, and expect everything to work!! The `fork()` system call shows one way to start the new process from a consistent state: namely, start it from the copy of an *existing* consistent state. The `exec()` call shows the other: namely, start it from the single entry point defined by the compiler.

Wait System Call

A parent process can waits for a its child process to finish or exit.
System Call is:

```
 waitpid (pid_t pid,int *status,.........);
```

pid =>the process id of the child that the parent is waiting for.
status => variable that will notify the parent whether the child exited successfully of not.

If the parent terminates, however, all its children have assigned a new parent, the init process. Thus, the children still have a parent to collect their status and execution statistics.

Exit system call

Via the exit() system call, A process terminates its execution and asks the operating system to delete all the resources that it was using. Such as: Virtual Memory, Open Files and Input Output Buffers. However, not all resources are released. In particular, the Process Control Block hangs around to store the process's exit status, until the parent calls `waitpid` appropriately. (This is

what a "zombie" process is: a process that has died, but whose parent hasn't `waited` for it yet.) Processes can also cause each other to exit with signals (see the next lecture). Usually such system calls have special access checks -- for example, a process owned by user U can only kill other processes owned by user U. Otherwise, users could arbitrarily kill each other's jobs.

Note that a parent needs to know the identities of its children in order to terminate their processes. In addition to above, some older systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either successful or unsuccessful), then all its children must also be terminated. This phenomenon is called Cascading termination, which is initiated by the operating system.