CMSC 216
Introduction to Computer Systems
Lecture 19
Procedures &
Process Control

Larry Herman & Alan Sussman

Notes

- Project 5 due Thursday
- Exam #2 on Thursday, April 19
  - practice exam available by end of week
- Read Sections 8.2-8.5 on process control
- Don't forget course projects policy
  - have to make a good faith effort on **all** projects before end of semester
  - that means submit a version that works on at least 50% of public tests

Section 3.7, Bryant and O'Hallaron

PROCEDURE IMPLEMENTATION

Register usage conventions

- Notice in the last example we had to reload the value of the parameter i after the recursive function call, because recursive calls to f would destroy the value in %eax
- A convention has developed as to which registers can be used by a procedure without destroying data from the caller
- When P calls Q...
  - Q can do anything it wants with the caller save registers; P is required to store these before the call if there's important data there
  - Q must save the values of the callee save registers, and restore them before returning (if Q uses these registers); P can use these to maintain data across function calls
- For reference (i.e., don't memorize this), and by convention, %eax, %edx, and %ecx are caller save, while %ebx, %esi, and %edi are callee save

Using callee save registers

```
f:      pushl  %ebp
        rrmovl %esp, %ebp

        mrmovl 8(%ebp), %eax
        irmovl $0, %ecx
        addl   %ecx, %eax
        jle    EndIf
        irmovl $-1, %ecx
        addl   %eax, %ecx
        pushl  %ecx
        call   f

EndIf:  mrmovl 8(%ebp), %eax
        wrint  %eax
        rrmovl %ebp, %esp
        popl   %ebp
        ret
```

```
f:      pushl  %ebp
        rrmovl %esp, %ebp
        pushl  %ebx           # store
    callee save
        mrmovl 8(%ebp), %ebx  # holds
    value of i
        irmovl $0, %ecx
        addl   %ecx, %ebx
        jle    EndIf
        irmovl $-1, %ecx
        addl   %ebx, %ecx
        pushl  %ecx
        call   f
        popl   %ecx           # pop arg
    off stack
EndIf:  wrint  %ebx           # no need
    for reload
        popl   %ebx           # restore
    val of reg
        rrmovl %ebp, %esp
        popl   %ebp
        ret
```


Returning values

- In Y86 (and x86), the convention is to store return values in **`%eax`**, and have the caller read this value after the call returns
- This works well enough for integers, but what about something like structs?
  - If the struct has ≤ 3 values, you could use all the caller save registers to pass the value back
  - The struct could be placed in the stack and accessed immediately after the function call

Example of returning values

```
int f(int);

int main() {
  static int n;
  n = f(5);
  printf("%d", n);
  printf("\n");
  return 0;
}

int f(int i) {
  return (i + 1) * 3;
}
```

```
main:  irmovl $0x1000, %esp  # init stack ptr
       irmovl $5, %eax       # load arg to reg
       pushl  %eax           # push arg on stack
       call   f
       rmmovl %eax, n        # store return value
       mrmovl n, %eax        # load value of n
       wrint  %eax
       irmovl $10, %eax      # printf("\n");
       wrch   %eax
       halt                  # return 0;

f:     pushl  %ebp           # save old frame ptr
       rrmovl %esp, %ebp     # set new frame ptr
       mrmovl 8(%ebp), %eax  # load param i
       irmovl $1, %ecx       # m + 1
       addl   %ecx, %eax     # add 1
       irmovl $3, %ecx       # for * 3
       multl  %ecx, %eax     # multiply by 3
       rrmovl %ebp, %esp     # reset stack ptr
       popl   %ebp           # restore frame ptr
       ret

       .align 4
n:     .long 0
```


Sections 8.2-8.5, Bryant and O'Hallaron

PROCESS CONTROL

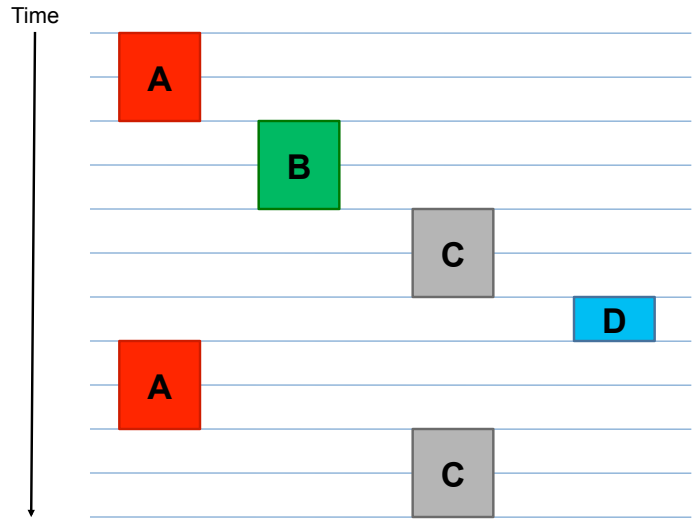
Processes

- Definition: "an instance of a program in execution"
- A process provides a context for the executing program: current memory state, open files, register contents, PC, environment variables
- When you type in the name of a program in the shell, the shell creates a new process in which that program will run

Multitasking

- Even though it appears to us that many programs run at the same time, in reality, only one process is able to use a CPU at a time
- The CPU is divided between all the currently running processes by the OS, which forces processes to be suspended
- To each process, it appears as if the process has sole control of the CPU, and logical flow of one process isn't affected by others (except for IPC)

Multitasking example



User and kernel modes

- Some instructions are restricted (or *privileged*), so that only the OS can execute them
  - e.g., halting the CPU, performing I/O, creating processes
- When a process needs to do one of these things, the process must enter kernel mode
- Normally, processes are in user mode

User and kernel modes, cont.

- When in user mode, trying to perform a privileged instruction or access kernel-reserved memory results in a fatal protection fault
- Should a process need to do one of these things, it needs to use the system call interface to access these instructions/areas of memory

Implementation of multitasking

- To move processes in and out of the CPU, the kernel needs to maintain the context (execution state) of each process
- Data structures used to maintain context:
  - page table, stores processes' address spaces
  - process table, stores information about each process
  - file table, lists which files are opened by each process

Context switching

- When a process is to be moved out of the CPU, the context is saved and the context of the incoming process is restored
- This is how the kernel schedules processes in the CPU
- Processes can be switched out whenever the kernel deems it necessary or appropriate
  - when a process is waiting for I/O
  - when a process has been using the CPU for a long time

System calls

- "The system call is the fundamental interface between an application and the Linux kernel." -- the **`syscalls`** manual page
- System calls include functions to do things like
  - open/close/create/delete files
  - read from/write to files
  - create processes
  - read the system clock

Error handling

- When a system call fails, it sets the global integer **`errno`** to a specific number to indicate what went wrong, and often returns -1
- When you call any system call, you need to check to see if it failed
- For example, if the **`fork()`** function fails, it returns -1:

```
if ((pid = fork()) < 0) {
  perror("fork error");
  exit(EX_OSERR);
}
```

- The book defines extra functions (e.g. **`Fork()`**) that call the corresponding existing function but do error handling, as above, to save on code writing

The `err.h` functions

- The header file **`<err.h>`** provides access to prototypes of functions that combine error reporting and exits:
- **`void err(int exit_code, const char *fmt, ...);`**
  - equivalent to:

```
fprintf(stderr, fmt, ...);
fprintf(stderr, ": %s\n", strerror(errno));
exit(exit_code);
```

- **`void errx(int exit_code, const char *fmt, ...);`**
  - equivalent to:

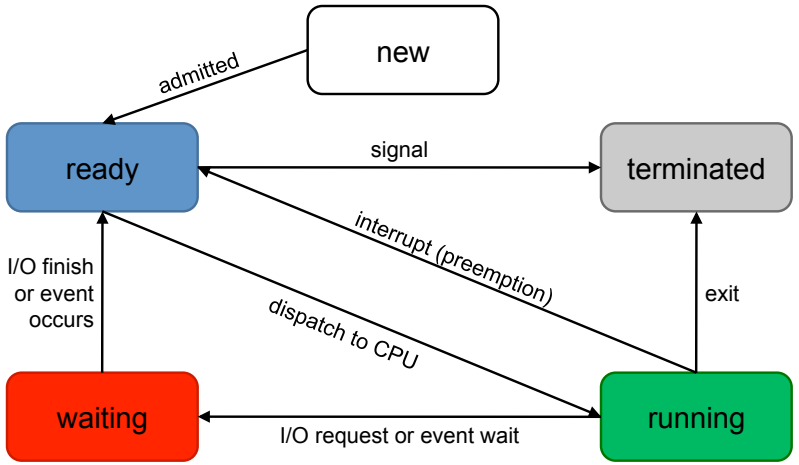
```
fprintf(stderr, fmt, ...);
exit(exit_code);
```

- Non-exiting functions **`warn()`** and **`warnx()`** also exist
- Information on these can be found via "**`man errx`**"
- Can be used to reduce error-checking code while maintaining flexibility in handling

Process life cycle

- Every process goes through several states (the exact type/number vary by OS)
- Typical states:
  - new: process has just been created
  - running: process is executing on the CPU
  - ready: process is waiting to be run
  - waiting: process is waiting for an event (e.g., I/O, signal)
  - terminated: process is finished executing

Process state transitions



Process IDs

- Every active process has a unique process ID, which can be obtained via **`getpid()`**
- The process ID of the current process' parent (the process which created the current process) can be obtained via **`getppid()`**
- Function prototypes:

```
#include <unistd.h>
#include <sys/types.h>

pid_t getpid(void);
pid_t getppid(void);
```