Recitation 7 - Makefile + PennOS

Agenda

1) **PennOS Intro**
2) **Makefile**

Projects So Far...

Penn Shredder

- Mini program that executes one command
- fork(2), alarm(2), signal(7)
- C programming, processes, and signal handling

Penn Shell

- bash-like shell
- I/O through redirections and pipes
- Job Control (fg/bg)
- Process groups, redirections

You called system calls/user-level functions to control behaviors of processes and simulate a shell.

Now it is your turn to implement these user-level functions and its lower level functionalities

Penn OS

Kernel / Scheduler

- Priority scheduling `(nice)`
- Process States (Running, Blocked, Zombie, etc..)
- Process Control Blocks
- Job Control
- User Contexts `(ucontext)`

File System

- **Single File** of fixed size
- Your own interface for interacting with files and file descriptors
- Simulate UNIX's file system
  - File allocation system
  - Directory
  - Redirections
- Support for UNIX commands
  - `ls, cat, touch, rm`

Penn OS

Penn OS Shell

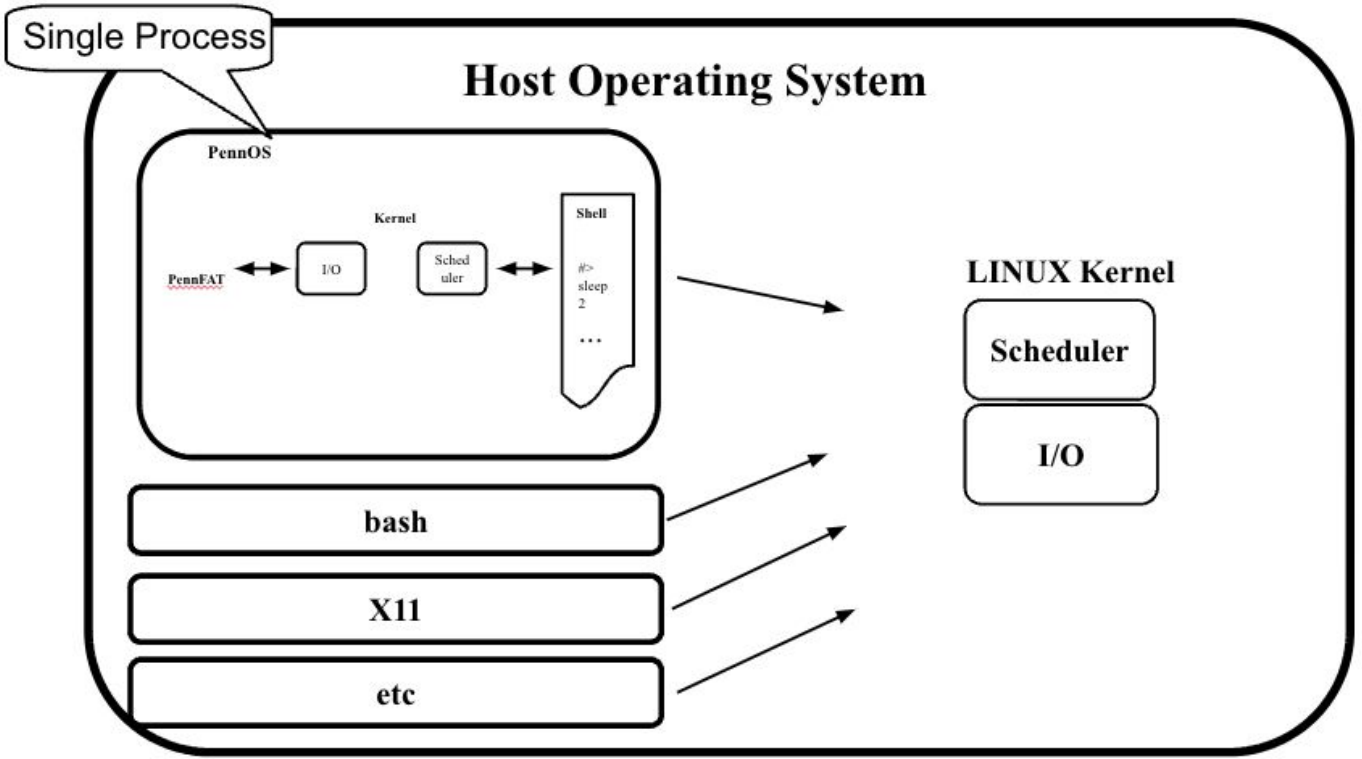
Kernel / Scheduler

- Priority scheduling `(nice)`
- Process States (Running, Blocked, Zombie, etc..)
- Process Control Blocks
- Job Control
- User Contexts `(ucontext)`

File System

- **Single File** of fixed size
- Your own interface for interacting with files and file descriptors
- Simulate UNIX's file system
  - File allocation system
  - Directory
  - Redirections
- Support for UNIX commands
  - `ls, cat, touch, rm`

PennOS as a Guest OS



Main Deliverables

- Standalone PennFAT (the file system) - 2 people
  - Commands for creating and removing the binary file acting as the file system
  - Mini shell that takes in other file system-related commands
- Kernel / Scheduler - 2 people
  - Priority Scheduling
  - Process Control Blocks and Process States
  - Logging
- Integration of File System, Scheduler into One Shell - Everyone
  - Parsing
  - Terminal Control
  - Terminal Signaling
  - Synchronous Child Waiting
  - Redirections (Pipeline for extra credit)
  - Error Handling with own p_perror API
  - **SEPARATION OF USER-LEVEL and KERNEL-LEVEL API**

Intro to Makefiles

This project will be submitted on gradescope via git. To submit, place all relevant code and documents in your groups git-repo on the master branch. You must organize your code into directories from the top-level as follows:

- ***`./bin`*** *- all compiled binaries should be placed here*
- ***`./src`*** *- all source code should be placed here*
- ***`./doc`*** *- all documentation should be place here*
- ***`./log`*** *- all PennOS logs should be placed here*

Your code should compile from the top level directory by issuing the ***`make`*** *command. Then, you should go to the gradescope submission and submit your github repo name link. Finally:* ***make sure that you add all of your project partners to the same submission***

~cis3800/23fa/projects/pennos/pennos

What is a Makefile?

- Used to control what gets recompiled
- Only recompiles what needs to be recompiled
- Automate tasks, primarily for building programs
- Describes relationships between files and how to derive target files from source files

Why use Makefiles?

- Avoid redundancy in compilation
- Track dependencies
- Make large projects more manageable

Makefile for PennOS

- `pennfat.c` and `pennos.c` will both contain a main() function
  - Compiler will scream if it sees both during linking

```
all: bin/pennos bin/pennfat

# make pennos binary
bin/pennos: .h and .o, except pennfat.o
     $(CC) ...
# make pennfat binary
bin/pennfat: .h and .o, except pennos.o
     $(CC) ...
```

*Approach 1: use wildcards and filter-out**
Approach 2: manually list files

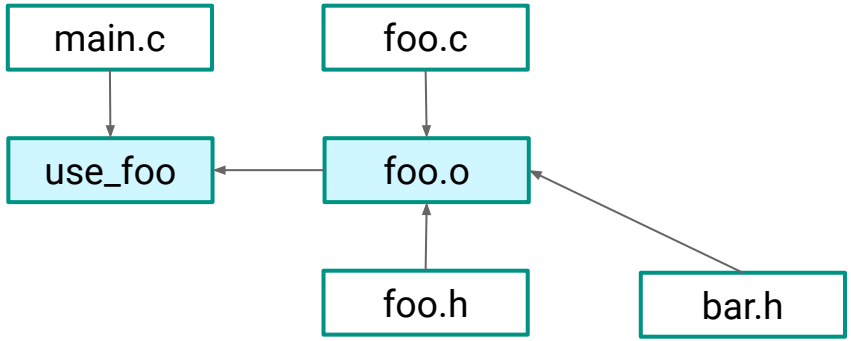
```
├── Makefile
├── bin/
│   ├── pennfat
│   └── pennos
├── doc/
├── log/
├── obj/ (optional)
│   ├── common.o
│   ├── parser.o
│   ├── pennfat.o
│   └── pennos.o
└── src/
    ├── common.c
    ├── common.h
    ├── parser.h
    ├── pennfat.c
    └── pennos.c
```

Making a Makefile

1. Make a file called `Makefile` (case sensitive)
2. Write triples in this format:

```
target: prerequisites
← Tab →command
← Tab →command
← Tab →command
```

- Colon after `target` is required
- `prereqs` are delimited by spaces
- `command` lines must start with a *TAB*, NOT SPACES
  - Can have multiple commands
  - Can split over multiple lines by ending lines with '\'




```
foo.o: foo.c foo.h bar.h
    clang -g -c foo.c
use_foo: foo.o main.c
    clang -g -o use_foo main.c foo.o
```

Running Makefile

Options:

1) Just "`make`"
   - Runs the first target
2) "make" with a target name
   - Ex: `` `make use_foo` ``

What does `make: 'target' is up to date` mean?

- Target doesn't need to be recompiled

1) Make compares timestamp of latest updates
2) If target is newer then all prereqs, up to date
3) If target is older, it gets recompiled

```
all: use_foo

foo.o: foo.c foo.h bar.h
	clang -g -c foo.c

use_foo: foo.o main.c
	clang -g -o use_foo main.c foo.o
```

Variables

Escaping Special Characters

- Special characters can be escaped with '\'
- $ is a special special, so you escape with '$'

Automatic Variables

- `$@`: Represents the target name
- `$^`: Lists all prerequisites
- `$<`: Represents the first prerequisite

```
# initialize a variable
PROG=penn-shell

# use a variable
PROMPT='"$(PROG)> "'

# escape chars
EXAMPLE_1='"shell\# "' # "shell# "
EXAMPLE_2='"$$ "'      # "$ "
```

Makefile from Project 0/1

CC: Defines the compiler being used (e.g. gcc, clang)

CFLAGS: Compiler flags for the C compiler

- -Wall: Enable all common warnings
- -Werror: Treat warnings as errors
- -g: Add debug information (good for valgrind/gdb)

CPPFLAGS: Flags for the C preprocessor, used with CFLAGs

- -D: Defines a macro
- -DPROMPT: Defines the PROMPT macro
  - Probably want to add for PennFAT/PennOS

clean:

- Often used as a target that removes the output of other targets, but it is not a special word in Make

```
override CPPFLAGS += -DNDEBUG \
-DPROMPT=$(PROMPT)

CC = clang

CFLAGS = -Wall -Werror -g

$(PROG): $(OBJS) $(HEADERS)
	$(CC) -o $@ $(OBJS) parser.o

clean:
	$(RM) $(OBJS) $(PROG)
```

Wildcards, Pattern Rules, Functions

`%`: pattern rule
- Wildcard that matches any non-empty substring for file names

`wildcard`: returns list of filenames matching the pattern
- Ex: `SRCS = $(wildcard src/*.c)`

`filter` / **`filter-out`**: self explanatory
- Ex: `$(filter $(INCLUDE), $(ALL))`
- Ex: `$(filter-out $(EXCLUDE), $(ALL))`

`patsubst`: substitutes text in a list based, on pattern
- Ex: `OBJECTS = $(patsubst src/%.c, obj/%.o, $(SRCS))`

```
# makes any obj/*.o files from corresponding src/*.c
file + headers
obj/%.o: src/%.c $(HEADERS)
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $< -o $@

# gets all sources
ALL_SRCS = $(wildcard src/*.c)

# filter out certain files (foo.c and bar.c)
SRCS = $(filter-out src/foo.c src/bar.c, $(ALL_SRCS))

# get matching object files, from $SRCS
OBJS = $(SRCS:src/%.c=obj/%.o)
```

Bonus Tips

.PHONY: indicate that a target isn't associated with a file

- Good for preventing name conflicts, optimization

`make -B` vs `make -b`

- `-B = -always-make`
  - Consider all targets as out of date, recompile everything
  - Use this one
- `-b = --no-builtin-rules`
  - Disables built in implicit rules of make
  - Not very useful for us

.gitignore

```
# To prevent large commits:
# - ignore binary files
bin/*
# - ignore *.o files
*.o
```

Makefile

```
# run pennos with valgrind
val:
	valgrind --leak-check=full \
	--show-leak-kinds=all \
	--track-origins=yes \
	--verbose bin/pennos
```