CMSC412 Discussion

Wed. Sept 26

Overview

- Some background info
- High level p2 overview
- Misc. notes

Read the Source!
Start Early!

Adding signals (High level)

- Allow processes to "communicate" with one another via signalling
- Allow processes to register behavior upon receiving some signal
- Basically need:
  - A way for a process to "send" something
  - A way for a process to "receive" something
  - A way for processes to register what they want to do
  - A way for a process's execution to change based on whether it received something / what it received
  - A way for a process's execution to resume "normal execution" once the signal has been handled

Some background

- Process flow:
  - Threads get a small quantum to run
  - When time's up / blocks does context switch
    - changes program counter to some instruction
  - More detail in P2 spec

- Process memory
  - Each (user) process has kernel stack / user stack
  - Much more detail in P1 spec

A way for a process to "send" something

- Modify Sys_Kill to send signals
  - Process A can call Sys_Kill at some other process to send a signal
- (Confusing naming: Sys_Kill does not mean "Kill process" anymore)

A way for a process to "receive" something

- Information about what processes have received what signals needs to be stored somewhere.
- This information should be modified via Sys_Kill

A way for processes to register what they want to do

- Sys_Signal
  - Give it a function pointer indicating the user code to execute when it receives some signal
- Similarly, this needs to be stored somewhere as well

Read the Source!
Start Early!

Handle signals

- Sys_RegDeliver (see _Entry.c)
- Check_Pending, Setup_Frame
- Process resumes, if it has a signal (Check_Pending), it will enter Setup_Frame
- High level concept:
  - Either handle signal directly in the kernel (e.g. terminate)
  - Manipulate kernel stack (i.e. the Interrupt_State's program counter) to control what code to execute next (handle signal)
  - Manipulate user stack - to control what code to execute after handling signal (trampoline function)

Recover from signals

- Because you have manipulated user stack...
- Sys_ReturnSignal -> Complete_Handler
- In Complete_Handler:
  - Want to return kernel stack to the way it was before signal handling...

(Actual) Handle signals

- High level concept:
  - Either handle signal directly in the kernel (e.g. terminate)
  - **SAVE current state of kernel stack somewhere**
    - convenient location: on the user stack
  - Manipulate kernel stack (i.e. the Interrupt_State's program counter) to control what code to execute next (handle signal)
  - Manipulate user stack - to control what code to execute after handling signal (trampoline function)

Recover from signals

- Sys_ReturnSignal -> Complete_Handler
- In Complete_Handler:
  - Want to return kernel stack to the way it was before signal handling
  - Take that snapshot of the kernel stack that you saved on user stack and put it back on top of the kernel stack

Misc. notes (more details in spec)

- Implement Sys_WaitNoPid
  - Wait without needing pid

- Various edge cases:
  - e.g. multiple signals
  - e.g. Getting a signal while handling a signal
  - e.g. Invalid input
  - e.g. Is the process going to execute in user space?

- Various changes from p1
  - e.g. Sys_Kill
  - e.g. detached children refcount

Read the Source!
Start Early!