Context Switching

- The OS can only run one process at a time
  - but this isn't very useful
- Simulate multiple processes with time-sharing
  - It gives each process a certain amount of time
  - When the time runs out, context-switch!

Context Switching

- Context switching does a few important things:
  - 1) Save the state of the current thread
  - 2) Load the state of the new thread
  - 3) Start the new thread
  - Look at Switch_To_Thread() in geekos/lowlevel.asm
- What is a state?
  - Kernel → registers, program counter, selectors
  - User → data (user variables, etc)

What To Do

- (1) Handle signals
  - What is a signal?
  - different than context switching
  - signals require jumping to an arbitrary function in user context
  - therefore you must save the old user context before going into the new user context

What To Do

- (2) Add a collection of syscalls to setup and call signals
- P2 is overall not much code, but VERY COMPLICATED → generally all or nothing

Signals

- SIGKILL
  - Produces the same results as Sys_Kill in p1
  - Can't be handled by a process
- SIGUSR1, SIGUSR2
  - User-defined signals
- SIGCHLD
  - You trigger it when a child dies and its parent abandoned it
  - In p2, ALL processes have parents!

System Calls To Implement

- Sys_WaitNoPID(int* status)
  - Looks for a zombie and fills the exit status & reaps
- Sys_Signal(ptr, sig_num)
  - Handle one of the three signals allowed with a user-space function
    - void handler(int sig_num)
  - Store this pointer for lookup later

System Calls To Implement

- Sys_RegDeliver(ptr)
  - Register a trampoline given to you
    - a special function that calls Sys_ReturnSignal at the end of a handler function
- Sys_Kill(pid, sig_num)
  - Send a given signal to a PID. Asynchronous. It's not handled until kernel context switches to the pid.
- Sys_ReturnSignal
  - Signal handling is complete! Restore our old state stored on the user stack.

Example: shell

- Take shell.c
- You need to handle dead processes
  - Sys_Signal(some_cleaning_fn, SIGCHLD)
- Your cleaning function will get called when there is a SIGCHLD
- But it can't do much on its own
  - Must call Sys_WaitNoPID to do the job

How it fits together

- When foo.exe starts:
  - Sys_RegDeliver(ptr) → store given ”trampoline” function (not something you control)
- Sys_Signal(bar, SIGUSR1)
- Someone calls Sys_Kill(your pid, SIGUSR1)
  - Sets a flag so foo.exe knows it has a signal waiting
- Check_Pending_Signal (signal.c) is called to check for flags before context-switching
  - If there's a signal, call Setup_Frame
  - If not, just switch to the thread as normal

How it fits together

- Setup_Frame → save state & modify where in foo.exe we switch to
  - Look up the right handler (bar)
  - Push current kernel interrupt state onto user stack
    - We don't want to lose our registers
  - Push signal # for later
  - Push address of ”trampoline” for later
  - Changes kernel stack so it has
    - Updated user stack ptr → free space
    - Saved program counter = signal handler (bar)
- Now, after context switch, bar gets called!

How it fits together

- Bar finishes. Say it computed the square root of three. (All this coding work for that lousy function?)
- Because of its position in the stack, the trampoline function gets called
  - Anyone know why?
- Trampoline function calls Sys_ReturnSignal
- Sys_ReturnSignal calls Complete_Handler (you write)
  - Restores the kernel stack to where it was before Setup_Frame
- Now the user process can be context switched to without knowing what hit it!

Final Words

- More details in the spec
  - Reading is **MANDATORY**
- Three rules to CS412
  - Start early
  - Start early
  - Start early
  - Also, ask questions, which is easier if you start early
- Any questions?