CS 3204
Operating Systems

Lecture 4

Godmar Back

Announcements

- Project 0 is due Feb 1, 11:59pm
- This project is done individually
- Curator instructions will be posted on website

Processes & Threads

Overview

- Definitions
- How does OS execute processes?
  - How do kernel & processes interact
  - How does kernel switch between processes
  - How do interrupts fit in
- What's the difference between threads/processes
- Scheduling of threads
- Synchronization of threads

Process

- These are all possible definitions:
  - A program in execution
  - An instance of a program running on a computer
  - Schedulable entity (*)
  - Unit of resource ownership
  - Unit of protection
  - Execution sequence (*) + current state (*) + set of resources

(*) can be said of threads as well

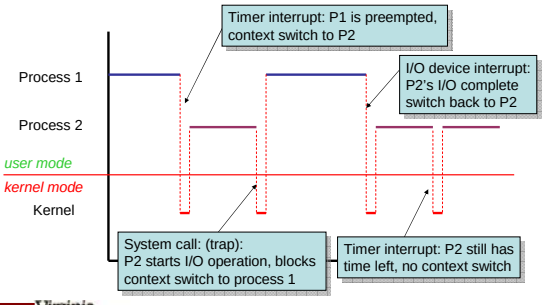
Alternative definition

- Thread:
  - Execution sequence + CPU state (registers + stack)
- Process:
  - (n * Threads + Resources (specifically: accessible heap memory, global variables, file descriptors, etc.)
- In most contemporary OS, n >= 1.
- In Pintos, n=1: a process is a thread – as in traditional Unix.
  - Following discussion applies to both threads & processes.

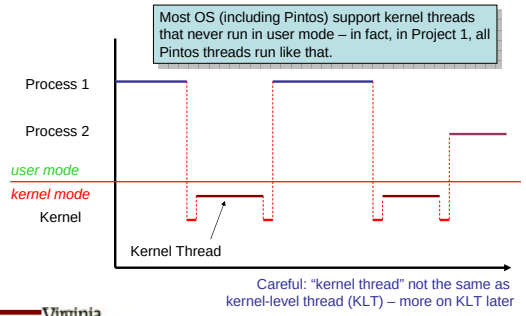
Context Switching

- Multiprogramming: switch to another process if current process is (momentarily) blocked
- Time-sharing: switch to another process periodically to make sure all process make equal progress
  - this switch is called a context switch.
- Must understand how it works
  - how it interacts with user/kernel mode switching
  - how it maintains the illusion of each process having the CPU to itself (process must not notice being switched in and out!)

Context Switching



Aside: Kernel Threads



Mode Switching

- User → Kernel mode
  - For reasons external or internal to CPU
- External (aka hardware) interrupt:
  - timer/clock chip, I/O device, network card, keyboard, mouse
  - asynchronous (with respect to the executing program)
- Internal interrupt (aka software interrupt, trap, or exception)
  - are synchronous
  - System Call (process wants to enter kernel to obtain services) – intended
  - Fault/exception (division by zero, privileged instruction in user mode) – usually unintended
- Kernel → User mode switch on iret instruction

Context vs Mode Switching

- Mode switch guarantees kernel gains control when needed
  - To react to external events
  - To handle error situations
  - Entry into kernel is controlled
- Not all mode switches lead to context switches
  - Kernel code's logic decides when – subject of scheduling
- Mode switch always hardware supported
  - Context switch (typically) not – this means many options for implementing it!

Implementing Processes

- To maintain illusion, must remember a process's information when not currently running
- Process Control Block (PCB)
  - Identifier (*)
  - Value of registers, including stack pointer (*)
  - Information needed by scheduler: process state (whether blocked or not) (*)
  - Resources held by process: file descriptors, memory pages, etc.

(*) applies to TCB (thread control block) as well

PCB vs TCB

- In 1:1 systems (Pintos), TCB==PCB
  - **`struct thread`**
  - add information as projects progresses
- In 1:n syst
  - TCB cont
    schedulin
    process t
  - PCB cont
    about res

```
struct thread
 {
   tid_t tid;                  /* Thread identifier. */
   enum thread_status status;  /* Thread state. */
   char name[16];              /* Name. */
   uint8_t *stack;             /* Saved stack pointer. */
   int priority;               /* Priority. */
   struct list_elem elem;      /* List element. */
   /* others you'll add as needed. */
};
```


Steps in context switch: high-level

- Save the current process's execution state to its PCB
- Update current's PCB as needed
- Choose next process N
- Update N's PCB as needed
- Restore N's PCB execution state
  - May involve reprogramming MMU

Execution State

- Saving/restoring execution state is highly tricky:
  - Must save state without destroying it
- Registers
  - On x86: eax, ebx, ecx, …
- Stack
  - Special area in memory that holds activation records
  - Saving the stack means retaining that area & saving a pointer to it ("stack pointer" = esp)

The Stack, seen from C/C++

```
int a;                          void func(int d)
static int b;                   {
int c = 5;                        static int e;
struct S                          int f;
 {                                struct S w;
   int t;                         int *g = new int[10];
 } s;                            }
```

A.: On stack: d, f, w (including w.t), g
Not on stack: a, b, c, s (including s.t), e, g[0]…g[9]

Switching Procedures

- Inside kernel, context switch is implemented in some procedure (function) called from C code
  - Appears to caller as a procedure call
- Must understand how to switch procedures (call/return)
- Procedure calling conventions
  - Architecture-specific
  - Defined by ABI (application binary interface), implemented by compiler
  - Pintos uses SVR4 ABI

x86 Calling Conventions

- Caller saves caller-saved registers as needed
- Caller pushes arguments, right-to-left on stack via push assembly instruction
- Caller executes CALL instruction: save address of next instruction & jump to callee
- Callee executes:
  - Saves callee-saved registers if they'll be destroyed
  - Puts return value (if any) in eax
- Callee returns: pop return address from stack & jump to it
- Caller resumes: pop arguments off the stack
- Caller restores caller-saved registers, if any