Processes & ECF

CS 351: Systems Programming
Michael Saelee <lee@iit.edu>
Agenda

- Definition & OS responsibilities
- Exceptional control flow
  - synch vs. asynch exceptions
  - exception handling procedure
§ Definition & OS responsibilities
process = a program in execution
{ code (program),
global data,
local data (stack),
dynamic data (heap),
PC & other regs }
programs *describe* what we want done, processes *realize* what we want done
... and the operating system runs processes
operating system ˈopəˌreɪtɪŋ ˈsɪstəm
noun
the software that supports a computer's basic functions, such as scheduling tasks, executing applications, and controlling peripherals.
execution = running a process
scheduling = running many processes
peripherals = I/O devices
to do this, the OS is constantly running “in the background”, keeping track of a large amount of process/system metadata
{ code,
global data,
local data (stack),
dynamic data (heap),
PC & other regs,
+ OS-level metadata }
{ code,
global data,
local data (stack),
dynamic data (heap),
PC & other regs,
+ (e.g., pid, owner, memory/CPU usage) }
Logical control flow
Physical flow (1 CPU)
Context switches
context switches are *external* to a process’s *logical* control flow (dictated by user program)—part of *exceptional* control flow
§ Exceptional Control Flow
```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```
logical c.f.

```c
int main() {
    while (1)
    {
        printf("hello world!\n");
    }
    return 0;
}
```
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
Two classes of exceptions:

I. synchronous

II. asynchronous
I. synchronous exceptions are caused by the currently executing instruction
3 subclasses of synchronous exceptions:

1. traps
2. faults
3. aborts
1. traps

traps are *intentionally* triggered by a process
e.g., to invoke a system call
char *str = "hello world";
int len = strlen(str);
write(1, str, len);

mov edx, len
mov ecx, str
mov ebx, 1
mov eax, 4 ; syscall #4
int 0x80  ; trap to OS
return from trap (if it happens) resumes execution at the next logical instruction
2. faults

faults are usually *unintentional*, and may be recoverable or irrecoverable

e.g., segmentation fault, protection fault, page fault, div-by-zero
often, return from fault will result in *retryping* the faulting instruction

— esp. if the handler “fixes” the problem
3. aborts
aborts are *unintentional* and *irrecoverable*

i.e., abort = program/OS termination

e.g., memory ECC error
II. asynchronous exceptions are caused by events external to the current instruction.
```c
int main() {
    while (1) {
        printf("hello world!\n");
    }
    return 0;
}
```

```
hello world!
hello world!
hello world!
hello world!
^C
$
```
hardware initiated asynchronous exceptions are known as *interrupts*
e.g., ctrl-C, ctrl-alt-del, power switch
interrupts are associated with specific processor (hardware) pins

- checked after every CPU cycle

- associated with interrupt handlers
interrupt procedure (typical)

- save context (e.g., user process)
- load OS context
- execute handler
- load context (for …?)
- return
important: after switching context to the OS (for exception handling), there is no guarantee if/when a process will be switched back in!
OS (kernel)
OS (kernel)

trap

handler

P₀ P₁ P₂ P₃ P₄
OS (kernel)
Computer Science

OS (kernel)

P₀  P₁  P₂  P₃  P₄

handler

trap
switching context to the kernel is potentially very expensive

— but the only way to invoke system calls and access I/O
moral (to be reinforced ad nauseum):
use system calls (traps) sparingly!