OS Interaction and Processes

Kai Shen

1

Multiprogramming

- So far we looked at how machine codes run on hardware and how compilers generate machine codes from high-level programs
- Fine if your program uses the machine exclusively.
  - But not efficient
- Multiprogramming: multiple programs (possibly belong-to/created-by different users)
  - Efficient sharing of resources (CPU, I/O, networking, …)

2

Operating Systems

⇨ Protection in a multiprogramming context
- One program shouldn't steal or alter another's data
- Stop or clean up a program when it does something wrong

⇨ Resource management in a multiprogramming context
- One program should not be able to monopolize shared resources (CPU, I/O, networking, …)

- Thus operating systems!

3

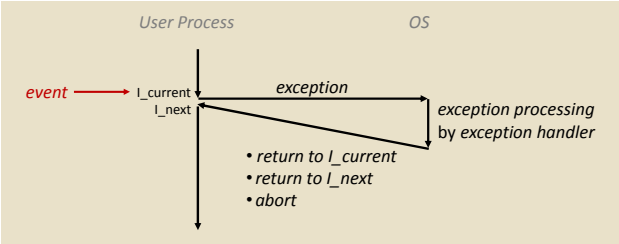
Operating System Interaction

- Perform some special/priviledged functions that a program can't handle on its own (for protection and resource management). Examples?
  - Open a file on disk
  - Process data from a network adapter
  - Access an invalid memory address
  - ⇨ Exceptions
- Coordinate between multiple concurrent program runs
  - Process management and context switches

4

Exceptions

- An *exception* is a transfer of control to the OS in response to some *triggering event*



5

Exception Table IA32 (Excerpt)

Exception Number	Description	Exception Class
0	Divide error	Fault
13	General protection fault	Fault
14	Page fault	Fault
18	Machine check	Abort
32-127	OS-defined	Interrupt or trap
128 (0x80)	System call	Trap
129-255	OS-defined	Interrupt or trap

6

Synchronous Exceptions

Caused by events from the current program execution:

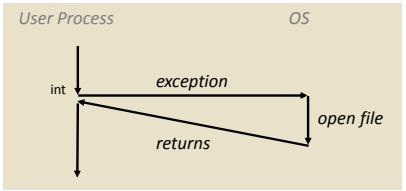
- ***Traps (system calls)***
  - Intentional
  - Returns control to "next" instruction
- ***Faults (software errors)***
  - Unintentional
  - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
  - Either re-executes faulting ("current") instruction or aborts
- ***Aborts (hardware errors)***
  - unintentional and unrecoverable
  - Examples: parity error, machine check
  - Aborts current program

7

Trap Example: Opening File

- User calls: `open(filename, options)`
- Function `open` executes system call instruction `int`

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                   int    $0x80
. . .
```



- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor

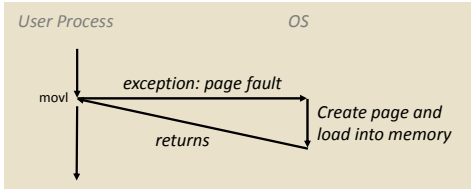
8

Fault Example: Page Fault

- User writes to memory location
- That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```



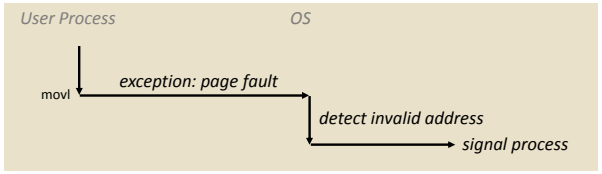
- Page fault handler must load page into physical memory
- Returns to faulting instruction
- Successful on second try

9

Fault Example: Invalid Memory Reference

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```



- Page handler detects invalid address
- Sends SIGSEGV signal to user process
- User process exits with "segmentation fault"

10

Asynchronous Exceptions (Interrupts)

- Caused by external events irrelevant to current program execution
  - Arrival of a packet from a network
  - Arrival of data from a disk
- OS handlers run in a foreign program context
  - Problem for resource accounting

11

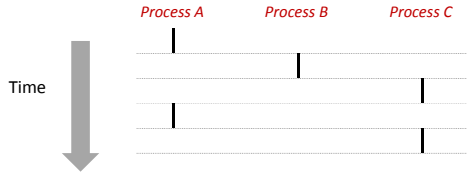
Processes

- Return to multiprogramming
- Definition: A *process* is an instance of a running program.
  - Not the same as "program" or "processor"
- Process provides each program with two key abstractions:
  - Logical control flow: each program seems to have exclusive use of the CPU
  - Private virtual address space: each program seems to have exclusive use of main memory
- How are these illusions maintained?
  - Process executions interleaved (multitasking) or run on separate CPUs
  - Address spaces managed by virtual memory system

12

Concurrent Processes

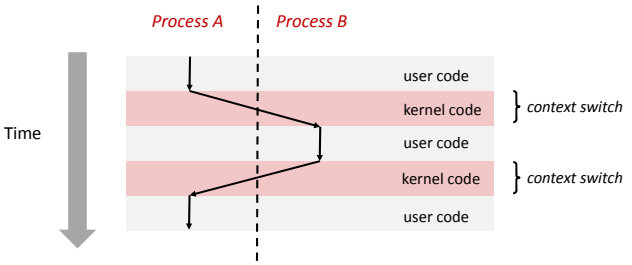
- Two processes *run concurrently* (*are concurrent)* if their flows overlap in time
- Otherwise, they are *sequential*
- Examples (running on single CPU core):
  - Concurrent: A & B, A & C
  - Sequential: B & C



13

Context Switching

- Control flow passes from one process to another via a *context switch*
  - Managed by the OS (kernel code)



14

`fork`: Creating New Processes

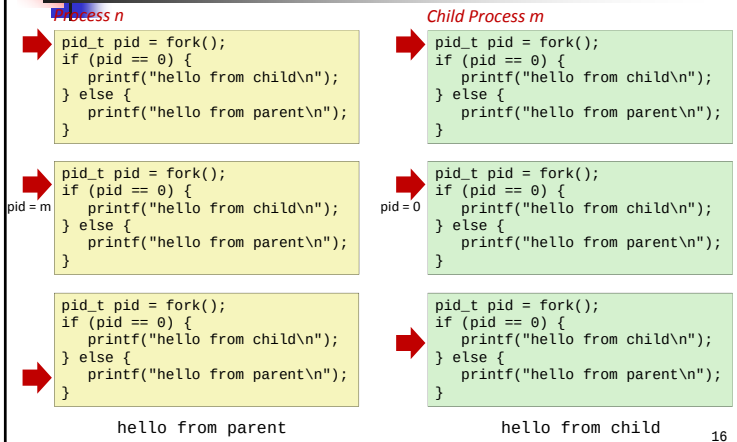
- **`int fork(void)`**
  - creates a new process (child process) that is identical to the calling process (parent process)
  - returns 0 to the child process
  - returns child's **`pid`** to the parent process

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

- Fork is interesting (and often confusing) because it is called *once* but returns *twice*

15

Understanding fork



16

Fork Example #1

- Parent and child both run same code
  - Distinguish parent from child by return value from **`fork`**
- Start with same state (memory space), but each has a separate, private copy after the fork
- Relative ordering of their subsequent execution undefined

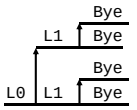
```
void fork_example()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
}
```

17

Fork Example #2

- Both parent and child can continue forking

```
void fork_example()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```

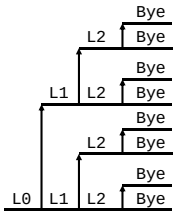


18

Fork Example #3

- Both parent and child can continue forking

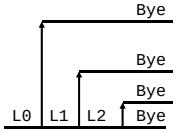
```
void fork_example()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



19

Fork Example #4

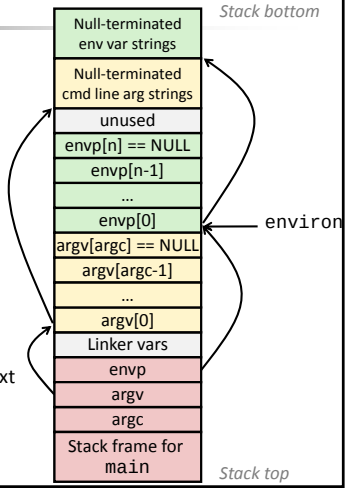
```
void fork_example()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



20

execve: Loading and Running Programs

- `int execve(`
  `char *filename,`
  `char *argv[],`
  `char *envp[])`
- Loads and runs in current process:
  - Executable **`filename`**
  - With argument list **`argv`**
  - And environment variable list **`envp`**
    - "name=value" strings
    - **`getenv`** and **`putenv`**
- Overwrites code, data, and stack
  - keeps pid, open files and signal context
- Does not return (unless error)



exit: Ending a process

- **`void exit(int status)`**
  - exits a process, cleans up system resources
- **`atexit()`** registers your own function to be executed upon exit

```
void cleanup(void) {
   printf("cleaning up\n");
}

void fork_example() {
   atexit(cleanup);
   fork();
   exit(0);
}
```

⇨ Operating system or C library calls?

22

wait: Synchronizing with Children

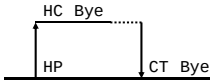
- **`int wait(int *child_status)`**
  - suspends current process until one of its children terminates
  - return value is the **`pid`** of the child process that terminated
  - if **`child_status != NULL`**, then the object it points to will be set to a status indicating why the child process terminated
- Purposes
  - Synchronization with children
  - Know the completion status of children

23

wait: Synchronizing with Children

```
void fork_example() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
   }
   else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
   exit();
}
```



24

`waitpid()`: Waiting for a Specific Process

- **`waitpid(pid, &status, options)`**
  - suspends current process until specific process terminates
  - various options (see textbook)

25

Children Reaping

- When process terminates, it may still need to respond to its parent
  - still consumes system resources (various tables maintained by OS)
  - Called a "zombie": living corpse, half alive and half dead
- Reaping
  - Performed by parent on terminated child
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then child will be reaped by **`init`** process
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

26

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

27