CIS330, Week 9

Processes, Exceptional Control Flow

CSAPPe2, Chapter 8

Plan for Today

- Exceptional Control Flow
  - Exceptions
  - Process context switches
  - Creating and destroying processes

Control Flow

- Computers do Only One Thing
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time.
  - This sequence is the system's physical *control flow* (or *flow of control*).

Time ↓

Physical control flow

<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

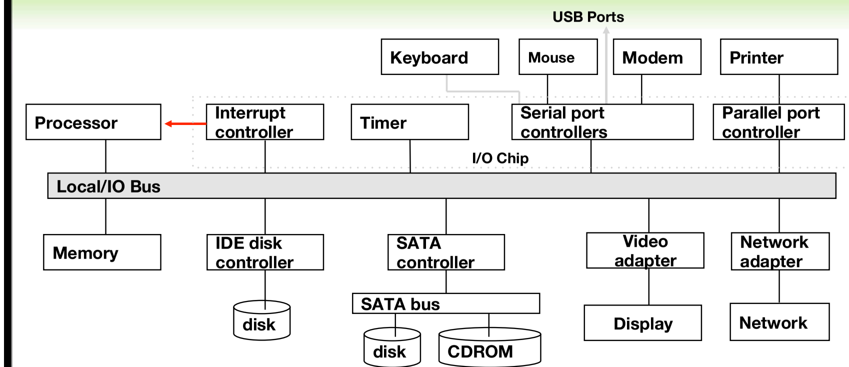
Altering the Control Flow

- Up to Now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return using the stack discipline.
  - Both react to changes in program state.
- Insufficient for a useful system
  - Difficult for the CPU to react to changes in system state.
  - data arrives from a disk or a network adapter.
  - Instruction divides by zero
  - User hits **Ctrl-c** at the keyboard
  - System timer expires
- System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

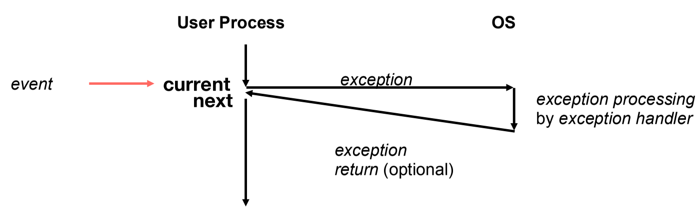
- Mechanisms for exceptional control flow exists at all levels of a computer system.
- Low level Mechanism
  - exceptions
  - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher Level Mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps (`setjmp/longjmp`)
  - Implemented by either:
    - OS software (context switch and signals).
    - C language runtime library: nonlocal jumps.

System context for exceptions

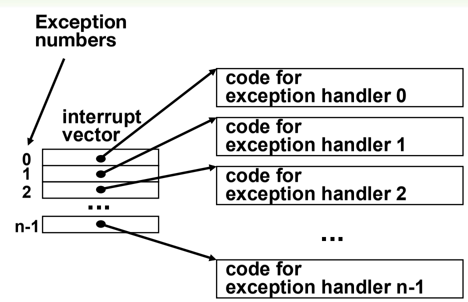


Exceptions

An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)



Interrupt Vectors



- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - handler returns to "next" instruction.
- Examples:
  - I/O interrupts
  - hitting ctl-c at the keyboard
  - arrival of a packet from a network
  - arrival of a data sector from a disk
  - Hard reset interrupt
  - hitting the reset button

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - Traps – intentional, e.g., system calls, breakpoints, special instructions; returns control to "next" instruction.
  - Faults – unintentional but possibly recoverable, e.g., page faults (recoverable), protection faults (unrecoverable), floating-point exceptions; re-executes faulting ("current") instruction or aborts.
  - Aborts: unintentional and unrecoverable, e.g., parity error, machine check; aborts current program

Precise vs. Imprecise Faults

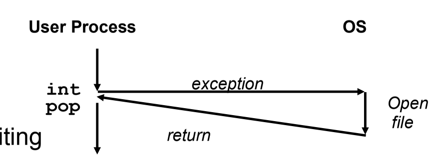
- Precise Faults: the exception handler knows exactly which instruction caused the fault.
  - All prior instructions have completed and no subsequent instructions had any effect.
- Imprecise Faults: the CPU was working on multiple instructions concurrently and an ambiguity may exists as to which instruction caused the Fault.
  - For example, multiple FP instructions were in the pipe and one caused an exception.

Trap Example

- Opening a File
  - User calls **open(filename, options)**

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                   int    $0x80
 804d084:       5b                      pop    %ebx
 . . .
```

- Function open executes system call instruction `int`
- OS must find or create file, get it ready for reading or writing
- Returns integer file descriptor



Fault Example #1

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

Memory Reference

User writes to memory location

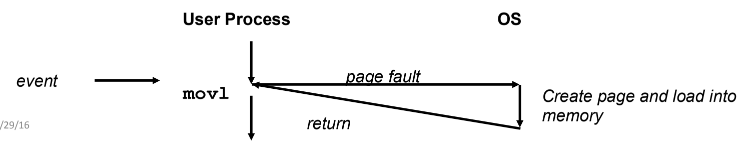
That portion (page) of user's memory is currently on disk

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

Page handler must load page into physical memory

Returns to faulting instruction

Successful on second try



Fault Example #2

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

Memory Reference with TLB miss

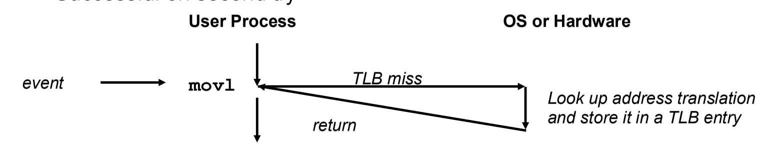
User writes to memory location

That portion (page) of user's memory is currently in physical memory, but the processor has forgotten how to translate the this virtual address to the physical address

TLB must be reloaded with current translation

Returns to faulting instruction

Successful on second try



Fault Example

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

Memory Reference

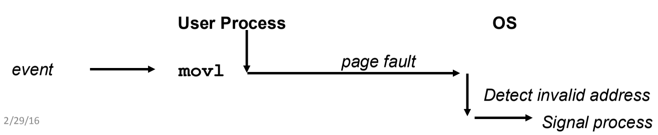
User writes to memory location

Address is not valid

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

Page handler detects invalid address Sends `SIGSEG` signal to user process

User process exits with "segmentation fault"

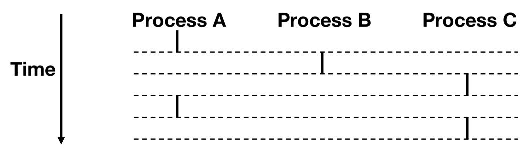


Processes

- Definition: A *process* is an instance of a running program.
  - One of the most profound ideas in computer science
  - 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 address space
  - Each program seems to have exclusive use of main memory
- How are these Illusions maintained?
  - Process executions interleaved (multitasking)
  - Address spaces managed by virtual memory system

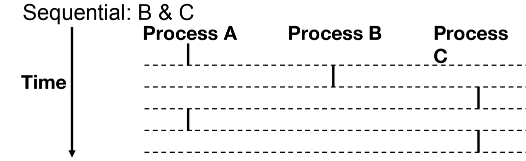
Logical Control Flows

- Each process has its own logical control flow



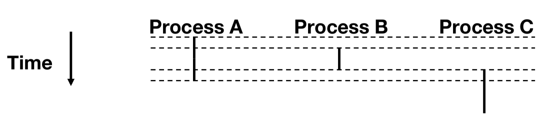
Concurrent Processes

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



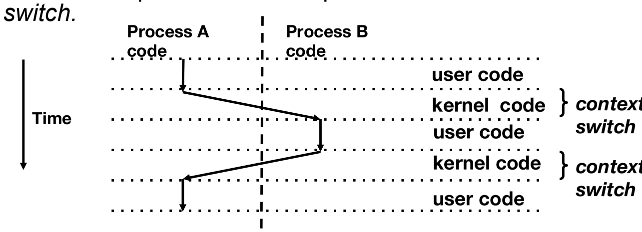
User View of Concurrent Processes

- Control flows for concurrent processes are disjoint in time.
- However, we can think of concurrent processes are running in parallel with each other.



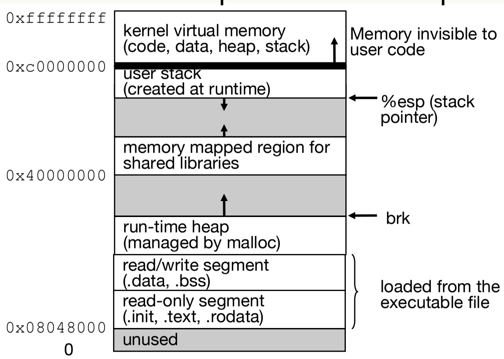
Context Switching

- Processes are managed by a shared chunk of OS code called the *kernel*
  - Important: the kernel is not a separate process, but rather runs as part of some user process
- Control flow passes from one process to another via a *context switch.*



Private Address Spaces

Each process has its own private address space.



execve: Loading and Running Programs

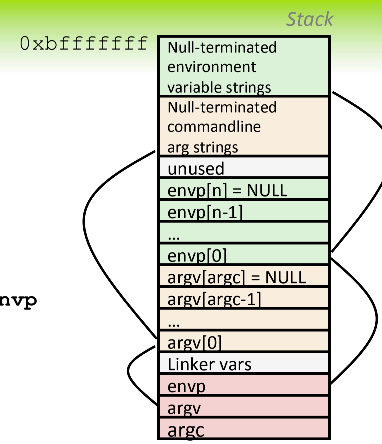
```
int execve(
    char *filename,
    char *argv[],
    char *envp
 )
```

Loads and runs
- Executable **filename**
- With argument list **argv**
- And environment variable **list envp**

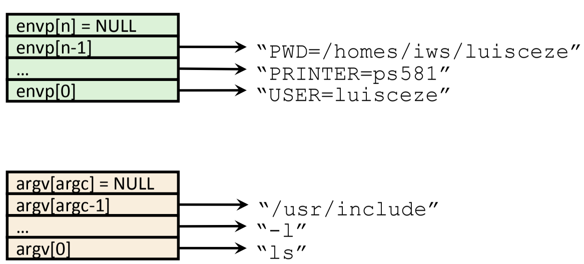
Does not return (unless error)

Overwrites process, keeps pid

Environment variables:
- "name=value" strings



execve: Example



Virtual Machines

- All current general purpose computers support multiple, concurrent *user-level* processes. Is it possible to run multiple kernels on the same machine?
- Yes: Virtual Machines (VM) were supported by IBM mainframes for over 30 years
- Intel's IA32 instruction set architecture is not virtualizable (neither are the Sparc, Mips, and PPC ISAs)
- With a lot of clever hacking, Vmware™ managed to virtualize the IA32 ISA in software
- User Mode Linux

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

```
if (fork() == 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*

Fork Example #1

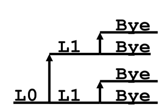
- Parent and child both run same code
  - Distinguish parent from child by return value from `fork`
- Start with same state, but each has private copy
  - Including shared output file descriptor
  - Relative ordering of their print statements undefined

```
void fork1()
{
    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);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```


Fork Example #2

Both parent and child can continue forking

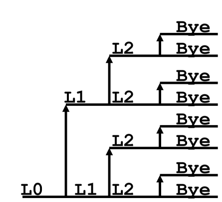
```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #3

Both parent and child can continue forking

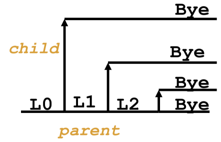
```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



Fork Example #4

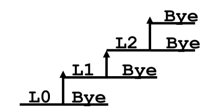
Both parent and child can continue forking

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
        printf("L1\n");
        if (fork() != 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



Fork Example #5

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
        printf("L1\n");
        if (fork() == 0) {
            printf("L2\n");
            fork();
        }
    }
    printf("Bye\n");
}
```



`exit`: Destroying Process

`void exit(int status)`

- exits a process
  - Normally return with status 0
- `atexit()` registers functions to be executed upon exit

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

void fork6() {
   atexit(cleanup);
   fork();
   exit(0);
}
```


Zombies

- Idea
  - When process terminates, still consumes system resources
  - Various tables maintained by OS
  - Called a “zombie”
- 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
  - Only need explicit reaping for long-running processes
  - E.g., shells and servers

Zombie Example

- `ps` shows child process as “defunct”
- Killing parent allows child to be reaped

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
linux> kill 6639
[1]    Terminated
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6642 ttyp9    00:00:00 ps
```

```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Nonterminating Child Example

- Child process still active even though parent has terminated
- Must kill explicitly, or else will keep running indefinitely

```
linux> ./forks 8
Terminating Parent, PID = 6675
Running Child, PID = 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6676 ttyp9    00:00:06 forks
 6677 ttyp9    00:00:00 ps
linux> kill 6676
linux> ps
  PID TTY          TIME CMD
 6585 ttyp9    00:00:00 tcsh
 6678 ttyp9    00:00:00 ps
```

```
void fork8()
{
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```


`wait`: Synchronizing with children

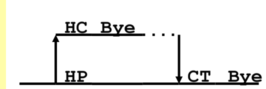
`int wait(int *child_status)`

- 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

`wait`: Synchronizing with children

```
void fork9() {
   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();
}
```



`Wait()` Example

- If multiple children completed, will take in arbitrary order
- Can use macros WIFEXITED and WEXITSTATUS to get information about exit status

```
void fork10() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```


`Waitpid()`

- `waitpid(pid, &status, options)`
  - Can wait for specific process
  - Various options

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                   wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
```


`Wait/Waitpid` Example Outputs

Using `wait` (`fork10`)

```
Child 3565 terminated with exit status 103
Child 3564 terminated with exit status 102
Child 3563 terminated with exit status 101
Child 3562 terminated with exit status 100
Child 3566 terminated with exit status 104
```

Using `waitpid` (`fork11`)

```
Child 3568 terminated with exit status 100
Child 3569 terminated with exit status 101
Child 3570 terminated with exit status 102
Child 3571 terminated with exit status 103
Child 3572 terminated with exit status 104
```


`exec`: Running new programs

`int execl(char *path, char *arg0, char *arg1, …, 0)`

- loads and runs executable at `path` with args `arg0`, `arg1`, …
  - `path` is the complete path of an executable
  - `arg0` becomes the name of the process
    - typically `arg0` is either identical to `path`, or else it contains only the executable filename from `path`
  - “real” arguments to the executable start with `arg1`, etc.
  - list of args is terminated by a `(char *)0` argument
- returns `-1` if error, otherwise doesn’t return!

```
main() {
   if (fork() == 0)
      execl("/usr/bin/cp", "cp", "foo", "bar", 0);
   wait(NULL);
   printf("copy completed\n");
   exit();
}
```

Summary

Exceptions

- Events that require non-standard control flow
- Generated externally (interrupts) or internally (traps and faults)

Processes

- At any given time, system has multiple active processes
- Only one can execute at a time, however,
- Each process appears to have total control of the processor + has a private memory space

Summary (cont'd)

Spawning processes

- Call to `fork`
- One call, two returns

Process completion

- Call `exit`
- One call, no return

Reaping and waiting for Processes

- Call `wait` or `waitpid`

Loading and running Programs

- Call `execl` (or variant)
- One call, (normally) no return