

Last Time: Caches

- Caches contain sets, each set contains blocks (lines), each line contain bytes
 - Blocks are read/written as a unit
 - Program addresses are broken into tag, set index, and block offset
 - N-way cache has N blocks per set, and also N places where a given byte might be cached
- Policy issues
 - Replacement
 - Write through vs. write back
 - Write allocate vs. write around
- Cache-friendly code
 - Exploits both temporal and spacial locality
 - Tricky when there are multiple nested loops
 - Blocking can help

1

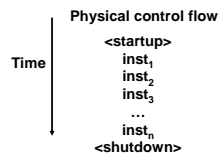
Today: Exceptional Control Flow Part I

- Topics
 - Exceptions
 - Process context switches
 - Creating and destroying processes
 - Lots of overlap with OS (5460)

2

Control Flow

- (Uniprocessor) 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*).



3

Altering the Control Flow

- Until now: two mechanisms for changing control flow:
 - Jumps and branches
 - Call and return using the stack discipline.
 - Both react to changes in program state.
 - » both are **user** level mechanisms
- Insufficient for a useful system
 - CPU must react to changes in system state.
 - » data arrives from a disk or a network adapter.
 - » instruction divides by zero or references illegal memory
 - » user hits control-C at the keyboard or kills window
 - » Time slice expires
- System needs mechanisms for “exceptional control flow”

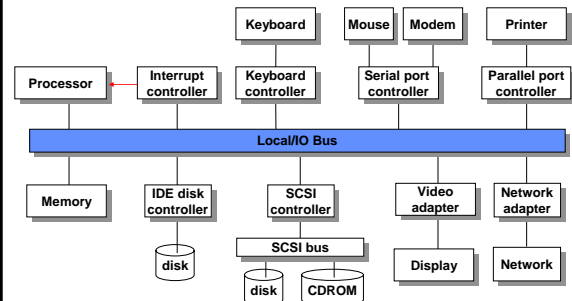
4

Exceptional Control Flow

- Mechanisms for exceptional control flow exists at multiple levels of a computer system.
- Low level Mechanisms
 - Exceptions and Interrupts
 - Implemented by combination of hardware and OS software
 - Can be synchronous or asynchronous
- Higher Level Mechanisms
 - Process or thread context switches
 - Signals (next lecture)
 - Nonlocal jumps (setjmp / longjmp)
 - Language-level exceptions
 - Implemented by either:
 - » OS software (context switch and signals).
 - » C / C++ / Java language runtime library: nonlocal jumps and language-level exceptions

5

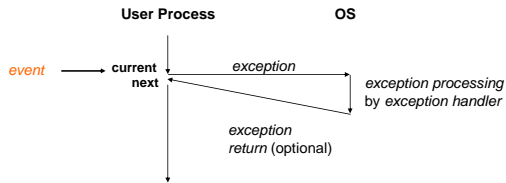
System context for interrupts



6

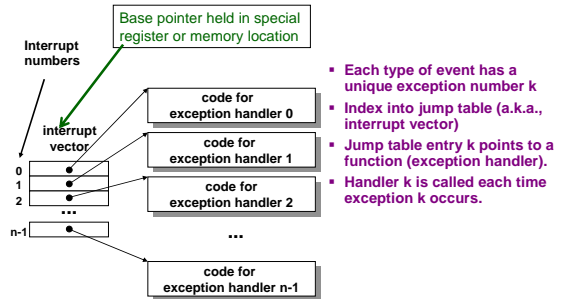
Interrupts and exceptions

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



7

Interrupt / Exception Vectors



8

Interrupt / exception handlers

- Treated a lot like function calls
 - return address is pushed onto the stack, then jump to new address
- Differences
 - push additional state information on the stack
 - needed to be able to restart the current program
 - on the x86: EFLAGS → the condition codes
 - handler may run in either the user or the kernel context
 - depends on the exception type
 - if kernel space then information is pushed onto the kernel stack
 - more context save information is required → context switch
 - kernel runs in *privileged* mode
 - Return address might not be next instruction

9

Returning from an Interrupt / Exception

- Varies depending on the exception type
- there is always some active instruction
 - call it → *current*
 - and then there's the *next* one
- 3 options
 - return to next instruction
 - works for device interrupts
 - Return to someplace completely different, but then later on return to the next instruction
 - return and re-execute the current instruction
 - When instruction did not complete successfully
 - e.g. memory reference that results in a TLB or page miss
 - abort
 - Normal return impossible for some reason
 - E.g. memory parity error

10

Asynchronous exceptions (interrupts)

- Caused by events (changes in state) external to the processor
 - Indicated by setting the processor's interrupt pin
 - interrupt type discrimination varies w/ architecture
 - Multiple interrupt pins
 - external device registers which must be queried
 - handler returns to "next" instruction
- Examples:
 - Arrival of a packet from a network
 - Network transmit queue has become empty
 - Arrival of a data sector from a disk
 - Soft reset interrupt
 - hitting *ctl-alt-delete* on a PC

11

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
 - Traps
 - Intentional
 - Examples: system calls, breakpoint traps, special instructions
 - Returns control to next instruction
 - Faults
 - Unintentional but possibly recoverable
 - Examples: divide-by-zero, page fault, protection fault, unaligned memory access, ...
 - Either re-executes faulting ("current") instruction or aborts
 - Aborts
 - unintentional and unrecoverable
 - Examples: parity error, machine check
 - Aborts current program

12

Who needs standards?

- Beware the terminology
 - every vendor has their own descriptive style
 - meanings vary for:
 - » fault, trap, exception, interrupt, etc.
- Beware the specifics
 - the hardware defines some exceptions
 - the OS defines others
 - There is not really any standard model for interrupt handling across processors and operating systems
 - » Sometimes hardware pushes entire machine state, sometimes just the PC
 - This lecture focuses on UNIX-ish ways of doing things
 - Windows is pretty analogous but a lot more complicated

13

Exceptions on x86 Processors

- Defines 256 exception types
 - Hardware specifies 0c31
 - » 0 – divide by 0 error
 - » 13 – the infamous general protection fault → segmentation fault
 - usually caused when a program accesses an illegal memory addr.
 - e.g. writes to .rodata segment
 - » 14 – page fault
 - » 18 – machine check → abort
 - OS specifies 32-255
 - » 32-127 – OS-defined exceptions
 - » 128 (0x80) – system call (Linux-specific)
 - » 129-255 – OS-defined exceptions

14

System Calls

- Mechanism varies with system
 - but every general-purpose architecture has a set of instructions which cause a trap
- IA32
 - provides the INT n instruction
 - » n can be any 8 bit unsigned value
 - » historically system calls are provided via exception 128
 - e.g. INT 0x80 → system call
 - » INT n is a “software interrupt”

15

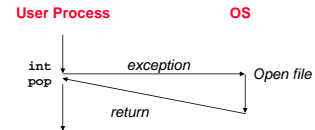
Trap Example

- Opening a File

- User calls open (filename, options)

```
0804d070 <_libc_open>:
. . .
804d080:  b8 05 00 00 00      mov    $0x5, %eax
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



16

Fault Example #1

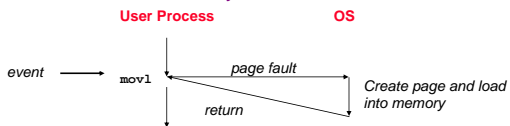
- Memory Reference

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



17

Fault Example #2

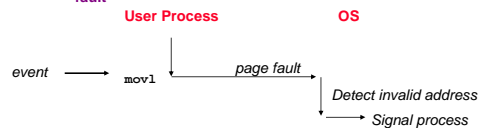
- Memory Reference

- User writes to memory location
- Address is not valid

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

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

- Page handler detects invalid address
- Kernel sends SIGSEGV signal to user process
- User process (usually) exits with “segmentation fault”



18

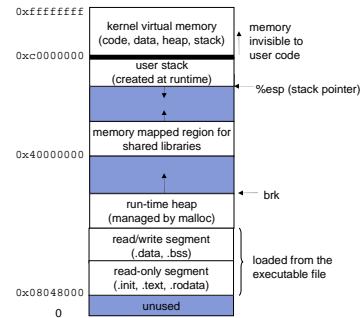
OS Kernel

- Nothing mysterious, it's just another program
 - However, it is NOT a process
- Portion of the OS that is memory resident
 - Brought into memory from disk at boot time
 - » Before any processes exist
 - Usually never paged out
- Part of the virtual address space of every process
- No need to take a context switch to invoke an OS service
 - Efficient!
- At this level, Windows, Linux, Solaris, MacOS X, etc. are quite similar

19

Linux Process Address Space

- Each process has its own private address space.



20

Kernel vs. User mode

- To provide private address spaces
 - something needs to restrict the instructions and data-space that an application can access
 - 2 mechanisms:
 - » Page tables
 - various segments stored in pages
 - Page table holds permissions (more in a few lectures)
 - permissions based on privilege and rwx style access
 - » CPU mode
 - supervisor and user mode supported by all modern processors
 - on the x86 there are 4 levels of privilege, 2 are seldom used
- Kernel provides protection and user services
 - call to kernel is indirect through the system call interface
 - call causes a privilege upgrade for most services

21

Kernel vs. User Mode

1. Program running normally
2. Program executes "INT 80", for example to request a file to be opened
3. Kernel gets control, saves some context, raises privilege level
4. Kernel opens the file
5. Kernel restores context of user mode program (including original privilege level!)
6. User program continues executing

22

System Calls and Error Handling

- Linux
 - 160 system calls
 - » man syscalls → the complete list
- Syscalls from C
 - use the `_syscall` macro
 - » man 2 intro → a description
 - Or roll your own assembly
 - however
 - » no need to directly invoke syscalls for most common calls
 - » C standard library provides a set of wrapper functions
 - package up the arguments, trap to the kernel w/ the desired syscall and pass the return status back
- Error handling & syscall returns
 - 0 → all is well
 - -1 → error
 - » errno contains the type of error
 - » strerror(errno) returns the string which describes the error

23

Wrapper Note

- Historically things were made to be simple and efficient
 - Programs using system call interfaces were assumed to be correct
 - Little or no error checking
 - Syscalls can spuriously fail and need to be retried
- The above represents a tradeoff: make life easy for OS implementors and hard for regular programmers!
 - UNIX philosophy: simple and wrong is better than complex and correct
- Wrapper functions
 - Hide complexity
 - Perform retries automatically
 - Provide error checking

24

Processes

- **Def: A process is an instance of a running program.**
 - Not the same as "program" or "processor"
 - One of the most fundamental ideas in computer science.
- **Process provides each program with three 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
 - Virtualized devices
 - » Sockets, file descriptors, etc.
- **How are these Illusions maintained?**
 - Process executions interleaved (multitasking)
 - Address spaces managed by virtual memory system
 - Devices multiplexed by OS

25

Processes

- **OS might provide only a fixed set of processes**
 - OS might not provide processes at all
 - E.g. embedded systems
- **Key OS challenge: Defining a good "process model"**
 - What is shared between processes and what isn't?
 - How are processes created and terminated?
 - » Can process be killed at an arbitrary time?
 - How are conflicting resource demands resolved?
 - Do processes contain threads?
 - Etc...

26

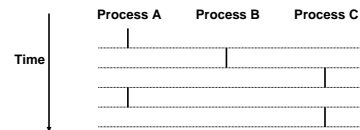
Process Context

- **State information that is required for correct execution**
 - contents of memory
 - » code
 - » data
 - » stack
 - general purpose register contents
 - special purpose registers
 - environment variables
 - » `> printenv`
 - `SHELL=/bin/bash`
 - `USER=regehr`
 - `CWD=/home/regehr/4400/lectures-s04`
 - ...
 - set of open file descriptors and sockets
 - » working set of the process includes the open files

27

Logical Control Flows

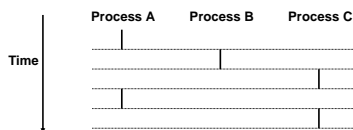
Each process has its own logical control flow



28

Concurrent Processes

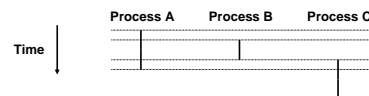
- **Two processes *run concurrently (are concurrent)* if their flows overlap in time.**
- **Otherwise, they are *sequential*.**
- **Examples:**
 - Concurrent: A & B, A & C
 - Sequential: B & C



29

User View of Concurrent Processes

- **Can think of concurrent processes as running in parallel with each other.**

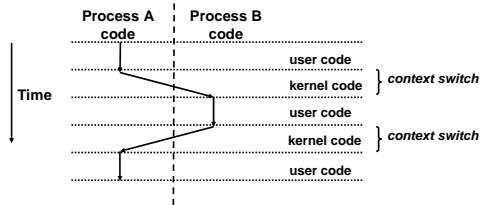


- **However, in reality control flows for concurrent processes are physically disjoint in time**
 - Again, assuming a uniprocessor
- **At the OS level, a context switch is required to change the currently running process**

30

Context Switching

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



31

Managing Context Switches

- Scheduler – a core part of the kernel
 - keeps the list of processes and decides which one goes next
- When does the scheduler get called?
 - Time slice expires (every 30ms, in many cases)
 - Process goes to sleep
 - Disk read, network read, etc.
 - Process wakes up
 - Disk read completes, network read completes, etc.
 - Process is created or terminates
- What does the scheduler do?
 - Decides whether a context switch should happen
 - If so, picks a new process to run and then:
 - saves context of previous process
 - restores new process context and restarts it

32

fork: Creating new processes

- `int fork(void)`
 - creates a new process (child process) that is identical to the calling process (parent process)
 - exact context copy at the time of the call
 - but in a separate virtual address space
 - 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

33

Fork Subtleties

- Call once return twice
 - child and parent process both get a return
 - but initially (point of call) are EXACTLY the same
- Parent and child run concurrently
- Duplicate but separate address spaces
 - initially the same
 - subsequent modifications are private and not seen by the other
 - unless explicitly communicated (details next lecture)
 - shared files
 - since open file descriptors are inherited, concurrent file access is now possible
 - BEWARE concurrent writers problem
 - synchronization & inter-process communication are necessary

34

Fork Example #1

- Key Points
 - 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

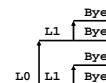
```
void fork1()
{
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n", getpid(), x);
}
```

35

Fork Example #2

- Key Points
 - Both parent and child can run fork multiple times

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



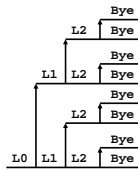
36

Fork Example #3

- **Key Points**

- Both parent and child can fork multiple times

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



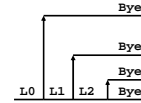
37

Fork Example #4

- **Key Points**

- Both parent and child can run fork multiple times

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



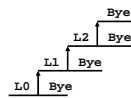
38

Fork Example #5

- **Key Points**

- Both parent and child can run fork multiple times

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

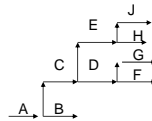


39

Stdout Serializer Example

- **Concurrent processes share a single stdout stream**

- HINT: this is ALWAYS an exam question
- What you see is a topological sort of the process graph



- Could see
 - › ABCDEFGHJ | ACDB... | ACEBD... | ACEJBD ...| etc.
- Could NOT see
 - › ADCB... | ABCDHE... | ACDFFE...| etc.

40

Destroying a Process

- **void exit(int status)**

- exits the current 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);
}
```

41

Zombies

- **Idea**

- When process terminates, still consumes system resources
 - › Various tables maintained by OS
- Called a "zombie"
 - › Not quite alive, not quite 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 – the ancestor of every UNIX process
- Only need explicit reaping for long-running processes
 - › E.g., shells and servers

42

Zombie Example

```
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 */
        }
    }
}
```

- ps shows child process as "defunct"
- Killing parent allows child to be reaped

43

Nonterminating Child Example

```
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
[1] Terminated
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);
        }
}
```

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

44

wait: Synchronizing with children

- `int wait(int *child_status)`
 - suspends current process until one of its children terminates
 - › note this is a wait for one rather than a wait for all
 - › if you want to wait for all there is a different mechanism
 - return value is the pid of the child process that terminated
 - › returns immediately if child process has already terminated
 - › and frees up the resources used by the child zombie
 - if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated

45

For Linux (from man wait)

If status is not NULL

wait or waitpid store status information in the location pointed to by status.

- This status can be evaluated with the following macros (these macros take the stat buffer (an int) as an argument -- not a pointer to the buffer!)
- **WIFEXITED(status)**
 - is non-zero if the child exited normally.
- **WEXITSTATUS(status)**
 - evaluates to the least significant eight bits of the return code of the child which terminated, which may have been set as the argument to a call to `exit()` or as the argument for a return statement in the main program. This macro can only be evaluated if `WIFEXITED` returned non-zero.

46

More

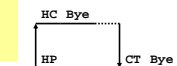
- **WIFSIGNALED(status)**
 - returns true if the child process exited because of a signal which was not caught.
- **WTERMSIG(status)**
 - returns the number of the signal that caused the child process to terminate. This macro can only be evaluated if `WIFSIGNALED` returned non-zero.
- **WIFSTOPPED(status)**
 - returns true if the child process which caused the return is currently stopped; this is only possible if the call was done using `WUNTRACED`.
- **WSTOPSIG(status)**
 - returns the number of the signal which caused the child to stop. This macro can only be evaluated if `WIFSTOPPED` returned non-zero.

47

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();
}
```



48

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);
    }
}
```

49

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);
    }
}
```

50

waitpid details

- #include
 - <sys/types.h> and <sys/wait.h>
 - template
 - » pid_t waitpid(pid_t pid, int *status, int options)
 - The value of pid can be one of:
 - » <-1 which means to wait for any child process whose process group ID is equal to the absolute value of pid.
 - » -1 wait for any child process (same as wait)
 - » 0 wait for any child process whose process group ID is equal to that of the calling process.
 - » > 0 wait for the child whose process ID is equal to the value of pid.

51

waitpid (cont'd)

- options
 - OR of zero or more of the following constants:
 - » WNOHANG - return immediately if no child has exited.
 - » WUNTRACE - return for children which are stopped, and whose status has not been reported.

52

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
```

53

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();
}
```

54

exec family

- A collection of execve front ends
 - see man `exec1` for details
 - » SYNOPSIS
 - » `#include <unistd.h>`
 - » `extern char **environ;`
 - » `int execl(const char *path, const char *arg, ...);`
 - » `int execlp(const char *file, const char *arg, ...);`
 - » `int execlx(const char *path, const char *arg, ..., char * const envp[]);`
 - » `int execv(const char *path, char *const argv[]);`
 - » `int execvp(const char *file, char *const argv[]);`
- `execve`
 - see man `execve` for details

55

Summarizing

- Exceptions
 - Events that require nonstandard 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, though
 - Each process appears to have total control of processor + private memory space

56

Summarizing (cont.)

- Spawning Processes
 - Call to `fork`
 - » One call, two returns
- Terminating Processes
 - Call `exit`
 - » One call, no return
- Reaping Processes
 - Call `wait` or `waitpid`
- Replacing Program Executed by Process
 - Call `exec1` (or variant)
 - » One call, (normally) no return

57