VIII. Exceptional Control Flow

- While power is on, the **Program counter** assumes a sequence of values

  $a_0, a_1, \ldots, a_{n-1}$ where a_k is the address of some corresponding instruction I_k

- ***Control transfer*** -- each transition from a_k to a_{k+1}
- ***Control flow*** -- sequence of such control transfers
  - "smooth" sequence when each I_k and I_{k+1} are adjacent in memory
  - abrupt changes caused by **jumps, calls, returns**
    - necessary mechanisms that allow programs to react to changes in **internal program state** representated by program variables

- **Systems must** also be able to **react to changes** in system state that are **not captured by internal program variables** and are **not necessarily related to the execution of the program**
  - Examples
    - hardware timer goes off at regular intervals
    - packets arrive at the network adpater and must be stored in memory
    - prgorams request data from a disk and sleep until they are notified tha the data are ready
    - parent processes that create child processes must be notified when their children terminate

- Modern systems react to abovementioned situations by making abrupt changes (**exceptional control flow (ECF)**) in control flow
  - ECF occurs at all levels of a computer system
    - **Hardware** level -- events detected by the hardware trigger abrupt control transfers to **exception handlers**
    - **Operating systems** level -- kernel transfers control from one user process to another via **context switches**
    - **Application** level -- process can send a **signal** to another process that abruptly transfers control to a **signal handler**
    - **Individual program** -- sidestepping usual stack discipline and making **nonlocal jumps** to arbitrary locations in other functions

- Importance of **ECF** (Understanding ECF will help you...)

1. understand important systems concepts
    - OS uses **ECF** to implement I/O, processes, and virtual memory
2. understand how applications interact with the operating system
    - applications request services from the OS by using a form of ECF known as a **trap** or **system call**
        - writing data to a disk, reading data from a network, creating a new process, terminating the current process
3. write interesting new application programs
    - Unix shells, Web servers
4. understand concurrency
    - exception handler interrupts the execution of an application program
    - processes and threads whose execution overlap in time
    - signal handler that interrupts the execution of an application program
5. understand how software exceptions work
    - software exceptions allow the program to make ***nonlocal*** jumps in response to error conditions

§8.1 Exceptions

- **Exceptions** -- a form of exceptional control flow that are implemented partly by the hardware and partly by the operating system
    - abrupt change in the control flow **in response to** some **change in the processor's state**

- Suppose the processor is executing some current instruction I_{curr} when a significant change (**event**) in the processor's **state** occurs
    - state -- encoded in various bits and signals inside the processor
    - **event** might be directly related to the execution of the current instruction
- When the processor **detects** that the **event has occurred**, it makes an **indirect procedure call (exception)**, through a jump table called an **exception table** to an operating subroutine (**exception handler**)
    - when the exception handler finishes processing, **three** possibilities (depends on the type of event that caused the exception)
        1. handler returns control to the current instruction I_{curr} , the instruction that was executing when the event occurred
        2. handler returns control to I_{next} , the instruction that would've executed next

had the exception not occurred
3. handler aborts the interrupted program

¶8.1.1 Exception Handling

- Each type of possible exception in a system is assigned a **unique nonnegative integer <u>exception number</u>**
    - assigned by the designers of the **processor**
        - divide by zero, page faults, memory access violations, break points, arithmetic overflows
    - assigned by the desginers of the **operating system kernel** (memory-resident part of the OS)
        - system calls, signals from external I/O devices

- At system boot time (reset or powered on), the **OS** allocates and initializes an **<u>exception table</u>**
    - entry k contains the **address of the handler for exception k**
- At run time, the **processor** detects that an **event has occurred** and determines the correspnding **exception number k**
    - then triggers the exception by making an **indirect procedure call,** through entry k of the exception talbe, to the **corresponding handler**
    - exception number = index into the exception table
        - starting address contained in a special CPU register (***exception table base register***)

- Some remarkable properties of **exceptions**
    - return address is **either the current instruction or the next instruction**
    - processor **pushes some additional processor state onto the stack** that will be necessary to **restart the interrupted program** when the handler returns
    - when control is being transferred **from a user progrm to the kernel**, all of the items are **pushed onto the kernel's stack**
    - **exception handlers** run in **kernel mode** (complete access to all system resources)
    - after the handler has processed the event, it **optionally** returns to the interrupted program by executing **"return from interrupt"** instruction which
        1. pops the appropriate stack back into the processor's control and data registers

2. restores the state to **user mode** (if the exception interrupted a user program)
3. returns control to the interrupted program

¶8.1.2 Classes of Exceptions

- Exceptions can be divided into four classes
    1. **Interrupts**
    2. **Traps**
    3. **Faults**
    4. **Aborts**

```
Class       Cause                           Async/Sync   Return behavior
Interrupt   Signal from I/O device          Async        Always to next instruction
Trap        Intentional exception           Sync         Always to next instruction
Fault       Potentially recoverable error   Sync         Might reurun to current instr
Abort       Nonrecoverable error            Sync         Never returns
```

Interrupts

- Occur ***asynchronously*** as a result of **signals** from I/O devices that are external to the processor
    - **not caused by** the execution of any particular instruction
    - handled by ***interrupt handlers***

- Brief process for an interrupt
    - I/O devices(network adapters, disk controllers, timer chips...) **trigger interrupts by signaling a pin** on the processor chip and **placing on to the system bus the exception number** that identifies the device that caused the interrupt
    - After the current instruction finishes executing, the **processor notices** that the interrupt pin has gone high, **reads the exception number** from the system bus, then **calls the appropriate interrupt handler**
    - When the handler returns, it **returns control to the next instruction**

--> **program continues executing as though the interrupt had never happened**

Traps and System Calls

- ***Traps*** -- ***intentional*** exceptions that occur as a result of executing an instruction
    - **trap handlers** return control to the next instruction
    - provide a procedure-like interface **between user programs and the kernel** --> **system call**

- User programs **request services from the kernel** when...
    1. reading a file (***read***)
    2. creating a new process (***fork***)
    3. loading a new program (**execve**)
    4. terminating the current process (**exit**)

- Processors provide a special **syscall** n instruction that user programs **can execute** when they want to request service n
    - executing a **syscall** causes a **trap** to an **exception handler** thant decodes the argument and **calls the appropriate kernel routine**

- From a programmer's perspective, a system call is identical to a regular function call, **However,** a **system call** runs in **kernal mode** (which allows it to **execute privileged instrucions & access a stack defined in the kernel**)

Faults

- Result from error conditions that a **handler might be able to correct**
- When a fault occurs, the **processor transfers control to the fault handler**
    - If the **handler is able to correct the error condition** , it returns control to the **faulting instruction** (re-execution)
    - If he handler **cannot** correct the error condition, **returns to an abort routine** in the kernel, **terminating the application program** taht caused the fault

- Example) **Page fault exception**
    - when an instruction references a virtual address whose corresponding **page** is not resident in memory and must therefore be retrieved from disk
        - **page fault handler** loads the appropriate page from disk and then returns

control to the instruction that caused the fault

Aborts

- Result from **unrecoverable fatal errors** (typically hardware errors)
- **Abort handlers NEVER** return control to the application program
    - returns control to an **abort** routine that **terminates the application program**

¶8.1.3 Exceptions in Linux / x86-64 Systems

- There are ~256 different exception types
    - 0~31 --> defined by the Intel architects $\Longleftrightarrow$ identical for any x86-64 system
    - 32~255 --> **interrupts & traps** defined by the OS
- Examples of exceptions

Linux / x86-64 Faults and Aborts

- **Divide error** (exception 0)
    - occurs when an application attempts to divide by zero or when the result of a divde instruction is too big for the destination operand
    - Unix does not attempt to recover from divide errors --> **Abort**
    - Linux shells report as " **Floating exceptions** "

- **General protection fault** (exception 13)
    - when a program references an undefined area of virtual memory or because the program attempts to write to a read-only text segment
    - Linux does not attempt to recover from this fault
    - Linux shells report as " **Segmentation faults** "

- **Page fault** (exception 14)
    - handler maps the appropriate page of virtual memory on disk into a page of physical memory, then **restarts the faulting instruction**

- **Machine check** (exception 18)
    - when fatal hardware error is detected during the execution of the faulting instruction
    - **Never** return control to the application program

Linux / x86-64 System Calls

- **Linux** provides **hundreds of system calls** that application programs use when they **want to request services from the kernel**
  - reading a file, writing a file, creating a new process
  - each system call has a **unique integer number** that corresponds to an **offset in a jump table** in the kernel (not the same as the exception table)

- C programs can invoke any system call **directly** by using the ***syscall*** function
- In general, it is better (easier) to use **wrapper functions** (**System-level functions**)
  - package up the arguments, trap to the kernel with the appropriate system call instruction, pass the return status of the system call back to the calling program
  - all arguments to Linux system calls are **passed through general-purpose registers**. By convention,
    - %rax - syscall number
    - %rdi, %rsi, %rdx, %r10, %r8, %r9 -- arguments

    --> On return from the system call, %rcx, %r11 are destroyed, and **%rax contains the return value**

    - negative return value (-4096~-1) **indicates an error**

[Example C code]

```c
int main() {
  write(1, "hello, world\n", 13);
  _exit(0);
}
```

- Arguments to **write** system-level function
  - 1st argument --> sends the output to **stdout**
  - 2nd argument --> sequence of bytes to write
  - 3rd argument --> number of bytes to write

[Implementing C code above directly with **Linux system calls**]

```
.section .data
```

```
string:
  .ascii  "hello, world\n"
string_end:
  .equ len, string_end - string
.section .text
.globl main
main:
  # First, call write(1, "hello, world\n", 13)
  movq $1, %rax       # write is system call 1
  movq $1, %rdi       # Arg1: stdout = descriptor 1
  movq $string, %rsi  # Arg2: hello world string
  movq $len, %rdx     # Arg3: string length
  syscall             # Make the system call

  # Next, call _exit(0)
  movq $60, %rax      # _exit is system call 60
  movq $0, %rdi       # Arg1: exit status is 0
  syscall             # Make the system call
```

§8.2 Processes

- When running a program on a modern system, we are presented with the **illusion** that our program is the **only one currently running** in the system --> provided by **Process**
    - our program seems to have exclusive use of the processor / memory
    - processor appears to execute the instructions without interruption
    - the code & data appear to be the only objects in the system's memory

- **Process** - an instance of a program in execution
    - each program in the system runs in the **context** of some process
        - context consists of the **state** that the program needs to run correctly
            - program's code & data stored in memory + stack + contents of its general purpose registers + program counter + environment variables + set of open file descriptors

- Each time a user **runs a program**, the **shell creates a new process** and then **runs the executable object file** in the context of this new process

- Key abstractions provided by a process
  - **Independent logical control flow** -- providing the illusion that our program has **exclusive use of the processor**
  - **Private address space** -- providing the illusion that our program has **exclusive use of the memory space**

¶8.2.1 Logical Control Flow

- Illusion that the program has exclusive use of the processor (even though many other programs are typically running concurrently on the system)
- **However,** when we use a debugger to **single-step the execution** of our program, we would observe a series of **program counter (PC)** values that **corresponded exclusively** to instructions contained in the program's executable object file or in shared objects linked into the program dynamically at run time

  --> sequence of PC values is known as **logical control flow** **(logical flow)**

- Suppose there is a system that runs three processes, then the single physical control flow of the processor is partitioned into three logical flows (one for each process)
  - processes take turns using the processor
  - each process executes a portion of its flow and then is ***preemtped*** (temporarily suspended) while other processes take their turns
    - can be observed by measuring the elapsed time of each instruction

¶8.2.2 Concurrent Flows

- **Concurrent flow** -- a logical flow whose execution overlaps in time with another flow (two flows are said to **run concurrently**)
- **Multitasking** -- notion of a process taking turns with other processes (sometimes referred to as **time slicing**)
- **Time slice** -- each time period that a process executes a portion of its flow

- If two flows **overlap in time** --> they are **concurrent**

  (independent of the number of processor cores or computers that the flows are running on)

- If two flows are **running concurrently on different processor cores or computers** --> **parallel flows** (running in parallel, or have parallel execution)

¶8.2.3 Private Address Space

- Process provides each program with the **illusion** that **it has <u>exclusive use</u> of the system's <u>address space</u>**
    - provides with its own **private address space**
        - **cannot** in general be **read or written by any other process**

- Each space **has the same general organization**
    - **Bottom portion** of the address space is reserved for the **user program**
        - usual code, data, heap, stack segments
        - code segment always begins at address 0x400000
    - **Top portion** of the address space is reserved for the **kernel** (memory-resident part of the OS)
        - code, data, stack
        - being used when the **kernel executes instructions** on behalf of the process (system calls)

¶8.2.4 User and Kernel Modes

- **<u>Mode bit</u>** in some control register -- characterizes the **privileges** that the process currentlly enjoys
    - Mode bit **set** -- process running in **<u>kernel mode</u>** (**supervisor mode**)
        - **can execute any instruction** in the instruction set and **access any memory location** in the system
    - Mode bit **not set** -- process running in **<u>user mode</u>**
        - **not allowed to** execute ***<u>privileged instructions</u>***
            - halt the processor, change the mode bit, initiate an I/O operation,
            - not allowed to directly reference code or data in the kernel area of the address space

              --> **fatal protection fault**

        --> User programs **MUST** access kernel code and data **indirectly** via the **system call interface**

- Application code is **initially in user mode**
    - To change from **user mode** to **kernel mode** is by **exceptions**
        - **interrupt, fault, trapping system call**
            - When ∃ exception, **Control passes to the exception handler**, the processor changes the mode to **kernel mode**
                - Handler runs in kernel mode and when it **returns to the application code,** the processor changes the mode **back to user mode**

- **/proc** filesystem
    - allows **user mode processes to access the contents of kernel data structures**
    - exports the contents of many kernel data structures as a hierarchy of text files that can be read by user programs

- **/sys** filesystem
    - additional low-level information about system buses and devices

¶8.2.5 Context Switches

- **Context switch** -- **multitasking** using a higher-level form of exceptional control flow

- **Kernel** maintains a **context** for each process
    - **state** that the **kernel needs to restart** a preempted process
        - general-purpose registers + floating-point registers + program counter + user's stack + status registers + kernel's stack + **kernel data structures**
            - **page table** (characterzies the address space), **process table** (contains information about the current process), **file table** (contains information about the files that the process has opened)

- At certain points during the execution of a process, the **kernel can decide to preempt the current process and restart a previously preempted process** (**scheduling**) -- handled by code in the kernel (**scheduler**)
- **Scheduled** -- when the kernel selects a new process to run

  --> **Kernel prempts the current process and transfers control to the new process** using a mechanism called **Context switch**

1. saves the context of the current process
2. restores the saved context of some previously preempted process
3. passes control to this newly restored process

--> can occur while the **kernel is executing a system call** on behalf of the user

- When the system call blocks, because it is **waiting** for some event to occur, then the **kernel can put the current process to sleep and switch to another process**
  - e.g. **read** system call requires a disk access,

    kernel can opt to perform a context switch and **run another process instead of waiting for the data** to arrive from the disk
  - e.g. **sleep** system call

    explicit request to put the calling process to sleep

    (even if a system call does not block, the kernel can decide to perform a context switch rather than return control to the calling process)

- **Context switch** can also occur as a result of an **interrupt**
  - e.g. when the disk sends an interrupt to signal that data have been transferred from disk to memory (return back to original process (prior to **read**)

§8.3 System Call Error Handling

- When Unix system-level functions encounter an error, they typically **return -1** and set global integer variabel **errno** to indicate what went wrong
- Programmers should **always check** for errors

  [Example C code]

```c
if ((pid = fork()) < 0) {
  fprintf(stderr, "fork error: %s\n", strerror(errno));
  exit(0);
}
```

--> **strerror** function returns a text string that describes the error associated with a particular value of **errno**

--> can simplify this code by defining the following ***error-reporting function***:

```c
void unix_error(char *msg) /* Unix-style error */
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(0);
}

/* In some function */
  if ((pid = fork()) < 0)
    unixerror("fork error");
```

- We can simplify our code even further by using **error-handling wrappers**
  - For a given base function **foo**, we define a **wrapper** function **Foo** with identical arguments but with the **first letter** of the name **capitalized**
    - wrapper function..
      - calls the base function
      - checks for errors
      - terminates if there are any problems

[Example C code]

```c
pid_t Fork(void) {
  pid_t pid;

  if ((pid = fork()) < 0)
    unix_error("Fork error");
  return pid;
}

/* In some function */
  pid = Fork()
```

H3 §8.4 Process Control

H5 ¶8.4.1 Obtaining Process IDs

- Each process has a **unique positive (nonzero) process ID (PID)**
- **getpid** function returns the **PID** of the calling process
- **getppid** function returns the **PID** of its ***parent***

```c
#include <sys/types.h>
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

--> return an integer value of type **pid_t** (defined in types.h as an **int**)

¶8.4.2 Creating and Terminating Processes

- Think of a **process** as being in one of **three** states
  1. **Running**
     - process is either **executing on the CPU** or **waiting** to be executed and will eventuallly be **scheduled by the kernel**
  2. **Stopped**
     - exeuction of the process is ***suspended*** and **will not be scheduled**
     - as a result of receiving a
       - **SIGSTOP, SIGTSTP, SIGTTIN, SIGTTOU** signal
     - stopped unitl it receives a
       - **SIGCONT** signal, at which point it becomes running again

     --> **signal** - a form of software interrupt
  3. **Terminated**
     - process is **stopped permanently**
     - three reasons to be terminated
       1. receiving a signal whose default action is to termiante the process
       2. returning from the main routine
       3. calling the **exit** function

       ```c
       #include <stdlib.h>
       void exit(int status); // Function does not return
       ```

       --> terminates the process with an **exit status** of **status**

       (other way is to return an integer value from the main routine)

- **Parent process** creates a new running **child process** by calling the **fork** function

```c
#include <sys/types.h>
#include <unistd.h>

pid_t fork(void); // Returns 0 to child, PID to parent, -1 on error
```

- Newly created process is almost, but not quite, **identical to the parent**
- The child gets an **identical (but separate)** copy of the parent's
    - **User-level virtual address space**
        - code
        - data segments
        - heap
        - shared libraries
        - user stack
    - **Open file descriptors**
        - child can read and write any files that were open in the parent when it called **fork**
- **Major difference** between the parent and the newly created child
    - **Different PIDs**

- **Fork** function is quite confusing since it is **called once**, but it **returns <u>twice</u>**
    - once in the **calling process**
        - returns the **PID of the child**
    - once in the newly created **child process**
        - returns a value of **0**

    --> provides an unambiguous way to tell whether the program is executing in the parent or the child

[Example C code]

```c
int main() {
  pid_t pid;
  int x = 1;

```

```c
  pid = Fork();
  if (pid == 0) {     /* Child */
    printf("child : x = %d\n", ++x);
    exit(0);
  }

  /* Parent */
  printf("parent: x = %d\n", --x);
  exit(0);
}
```

--> will return

```
linux> ./fork
parent : x = 0
child : x = 2
```

- Aspects of **fork** functions
  1. **Call once, return twice**
     - called once by the parent, but it returns twice
       - once to the parent and once to the newly created child

  2. **Concurrent execution**
     - parent and child are **separate processes** that run **concurrently**
     - parent process completes its statement first, then followed by the child (with example above)
       - on another system, the reverse might be true

  3. **Duplicate but separate address spaces**
     - when we stop both the parent and the child immediately after the **fork** funnction returned in each process, the address space of each process is **identical**
       - each process has the same user stack, local variable values, heap, global variable values, and code
     - **HOWEVER,** since the parent and the child are **separate processes**,
       - they each have their **own private address spaces**

- any subsequent changes that a parent or child makes to **x** are private and are not reflected in the memory of other process

4. **Shared files**
    - when the parent calls **fork**, the **stdout** file is open and directed to the screen
        - the child inherits this file, and its output is also directed to the screen

- **Nested fork calls**

```c
int main(){
  Fork();
  Fork();
  printf("hello\n");
  exit(0);
}
```

--> prints hello four times

¶8.4.3 Reaping Child Processes

- When a process terminates for any reason, the kernel does not remove it from the system immediately
    - Instead, the process is **kept** aorund in a terminated state **until it is reaped** by its parent
        - When the parent **reaps** the terminated process, at which point it ceases to exist
        - a terminated process that has not yet been reaped is called a **zombie**

- When a parent process terminates, the kernel arranges for the init process to become the adopted parent of any **orphaned children**
    - **init** process has a PID of 1
        - is created by the kerenel during system start-up, **never terminates**, and is the **ancestor of every process**
    - If a parent process terminates without reaping its zombie children, then the kernel arranges for the **init** process to reap them

- Long-running programs (shells, servers) should **always reap their zombie children** (∵

even though zombies are not running, they still consume system memory resources)

- A process waits for its children to terminate or stop by calling the **waitpid** function

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *statusp, int options);
  // Returns PID of child if OK, 0 (if WNOHANG), or -1 on error
```

- By default (options = 0), **waitpid** suspends execution of the calling process until a child process in its ***wait set*** terminates
- If a process in the wait set has **already terminated at the time of the call**, then **waitpid** returns immediately

  --> In either case, **waitpid** retruns the PID of the terminated child that caused **waitpid** to return

  --> at this point, the terminated child has been reaped and the kernel removes all traces of it from the system

Determining the Members of the Wait Set

- Members of the wait set are determined by the **pid** argument
  - If **pid** > 0, then the wait set is the **singleton child process** whose process ID is equal to **pid**
  - If **pid** = -1, then the wait set consists of all of the parent's child processes

Modifying the Default Behavior

- Default behavior can be modified by setting **options** to various combinations of the **WNOHANG, WUNTRACED, WCONTINUED** constants

- **WNOHANG**
  - **return immediately** (with a return value of **0**) if **none** of the child processes in the wait set has **terminated yet**
  - useful when you **want to continue doing useful work while waiting** for a child to

terminate

- **WUNTRACED**
  - suspend execution of the calling process until a process in the wait set becomes **either terminated or stopped**
  - return the **PID** of the **terminated or stopped child** that caused the return (by default, only for the terminated child)
  - useful when you want to **check for both terminated and stopped children**

- **WCONTINUED**
  - suspend exeuction of the calling process until a running process in the wait set is **terminated** or until a stopped process in the wait set **has been resumed** by the receipt of a **SIGCONT** signal

- **WNOHANG | WUNTRACED**
  - return **immidiately** , with a return value of **0** , if **none** of the children in the wait set **has stopped or terminated** , or with a return value to the **PID** of **one of the stopped or terminated children**

Checking the Exit Status of a Reaped Child

- If the **statusp** argument is non-NULL, the **waitpid** **encodes status information about the child** that caused the return in **status** (value pointed to by **statusp**)
- **wait.h** include file defines several macros for interpreting the **status** argument

- **WIFEXITED** (status) -- returns true if the child terminated normally, via a call to **exit** or a return
- **WEXITSTATUS** (status) -- returns the exit status of a normally terminated child (is only defined if **WIFEXITED** () returned true)
- **WIFSIGNALED** (status) -- returns true if the child process terminated because of a signal that was not caught
- **WTERMSIG** (status) -- returns the number of the signal that caused the child process to terminate (is only defined if **WIFSIGNALED** () returned true
- **WIFSTOPPED** (status) -- returns true if the child that caused the return is currently stopped
- **WSTOPSIG** (status) -- returns the number of the signal that caused the child to stop (is

only defined if **WIFSTOPPED()** returned true)
- **WIFCONTINUED** (status) -- returns true if the child process was restarted by receipt of a **SIGCONT** signal

Error Conditions

- If the calling process has no children, then **waitpid** returns -1 and sets **errno** to **ECHILD**
- If the **waitpid** function was interrupted by a signal, then it returns -1 and sets **errno** to **EINTR**

The wait Function

- **wait** function is a simpler version of **waitpid**

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *statusp);
```

--> calling **wait(&status)** is equivalent to calling **waitpid(-1, &status, 0)**

Examples of Using waitpid

[Example C code]

```
#include "csapp.h"
#define N 2

int main() {
  int status, i;
  pid_t pid;

  /* Parent creates N children */
  for (i = 0; i < N; i++)
    if ((pid = Fork()) == 0)  /* Child */
      exit(100+i);

  /* Parent reaps N children in no particular order */
  while ((pid = waitpid(-1, &status, 0)) > 0) {
```

```
15      if (WIFEXITED(status))
16        printf("child %d terminated normally with exit status=%d\n", pid, WEXITSTATUS(status));
17      else
18        printf("child %d terminated abnormally\n", pid);
19    }
20
21    /* The only normal termination is if there are no more children */
22    if (errno != ECHILD)
23      unix_error("waitpid error");
24
25    exit(0);
26  }
```

- Uses **waitpid** to wait for all of its N children to terminate
- In L10, the parent creates each of the N children, and in L11, each child exits with a unique exit status
- In L14, the parent waits for all of its children to terminate by using **waitpid** as the test condition of a **while** loop
  - since the first arg = -1, the call to **waitpid** blocks until an arbitrary child has terminated
- L15 checks the exit status of the child
  - if the child terminated normally (by calling the **exit** function), then the parent extracts the exit status and prints it on **stdout**
- When all of children have been reaped, the next call to **waitpid** returns **-1**, and sets **errno** to **ECHILD**
  - L22 checks that the **waitpid** function terminated normally, and prints an error message otherwise

--> this program reaps its children in no particular order

==> ***nondeterministic*** behavior (makes reasoning about concurrency so difficult)

--> **never** assume that one outcome will always occur

--> each possible outcome is equally likely

- **However, simple change** can eliminate this nondeterminism in the output order

```
#include "csapp.h"
#define N 2

```

```
4 int main() {
5   int status, i;
6   pid_t pid[N], retpid;
7
8   /* Parent creates N children */
9   for (i = 0; i < N; i++)
10    if ((pid[i] = Fork()) == 0)   /* Child */
11      exit(100+i);
12
13  /* Parent reaps N children in order */
14  i = 0;
15  while ((retpid = waitpid(pid[i++], &status, 0)) > 0) {
16    if (WIFEXITED(status))
17      printf("child %d terminated normally with exit status=%d\n", retpid, WEXITSTATUS(status));
18    else
19      printf("child %d terminated abnormally\n", retpid);
20  }
21
22  /* The only normal termination is if there are no more children */
23  if (errno != ECHILD)
24    unix_error("waitpid error");
25
26  exit(0);
27 }
```

--> In L10, the parent stores the PID's of its children in order and then waits for each child in this same order by calling **waitpid** with the appropriate **PID** in the first argument (rather than -1 for arbitrary)

¶8.4.4. Putting Processes to Sleep

- **sleep** function suspends a process for a specified period of time

```
#include <unistd.h>
unsigned int sleep(unsigned int secs);
                        // Returns seconds left to sleep
```

- returns 0 -- if the requested amount of time has elapsed
- returns number of seconds still left to sleep

        - if **sleep** function returns prematurely because it was interrupted by a ***signal***

- **pause** function puts the calling function to sleep until a signal is received by the process

```c
#include <unistd.h>
int pause(void);                // Always returns -1
```

¶8.4.5 Loading and Running Programs

- **execve** function **loads and runs** a new program in the context of the current process

```c
#include <unistd.h>
int execve(const char *filename, const char *argv[], const char *envp[]);
              // Does not return if OK; returns -1 on error
```

    - loads and runs the executable object file **filename** with the argument list **argv** and the environment variable list **envp**
        - **returns** to the calling program **only if** there is an **error** (not being able to find **filename**)
            - called once but never returns

- **argv** -- points to a null-terminated array of pointers, each of which points to an **argument string** (e.g. ls, -lt, ...)
    - **argv[0]** -- name of the executable object file
- **envp** -- points to a null-terminated array of pointers to **environment variable strings** , each of which is a name-value pair of the form (e.g. PWD=/usr/---, USER = ---)(**name = value**)

- After **execve** loads **filename**, it calls the **start-up** code
    - sets up the stack
    - passes control to the main routine of the new program which has a prototype of the form

```c
int main(int argc, char *argv[], char *envp[]);

/* Which is equivalent to */
int main(int argc char **argv, char **envp);
```

- Stack organization when **main** begins executing (from bottom to top)
  - argument & environment strings
  - null-terminated array of pointers, each of which points to an environment variable string
    - global variable **environ** points to the first of these pointers, **envp[0]**
  - null-terminated **argv[]**, with each element pointing to an argument string on the stack
  - [TOP] - stack frame for the system start-up function **libc_start_main**

- Three arguments are stored in registers
  1. **argc** (in %rdi) -- gives the number of non-null pointers in the **argv[]** array
  2. **argv** (in %rsi) -- points to the first entry in the **argv[]** array
  3. **envp** (in %rdx) -- points to the first entry in the **envp[]** array

- Several functions for manipulating the environment array

```c
#include <stdlib.h>
char *getenv(const char *name);
    // Returns: pointer to name if it exists, NULL if no match
```

  - **getenv** function searches the environment array for a string **name=value**
    - If found, it returns a pointer to **value**
    - otherwise, returns **NULL**

```c
#include <stdlib.h>
int setenv(const char *name, const char *newvalue, int overwrite);
  // Returns: 0 on success, -1 on error

void unsetenv(const char *name);
  // Returns: nothing
```

- **setenv** replaces **oldvalue** with **newvalue** only if **overwrite** is nonzero
  - If **name** does not exist, then **setenv** adds **name=newvalue** to the array
- **unsetenv** deletes **name=oldvalue** if it exists

§8.5 Signals

- **Linux signal** -- allows processes and the kernel to interrupt other processes
  - small message that **notifies a process** that an event of some type has occurred in the system
  - each signal type corresponds to some kind of system event
    - **low-level hardware exceptions** are processed by the **kernel's exception handlers** and **would not normally be visible to user processes**

      --> **Signals expose** the occurrence of such exceptions to user processes

    e.g. if a process attempts to divide by zero, then the kernel sends it a SIGFPE signal

  - other signals correspond to higher-level software events in the kernel or in other user processes

    e.g. typing Ctrl+C while a process is running in the foreground, then the kernel sends a SIGINT (no. 2) to each process in the foreground process group

¶8.5.1 Signal Terminology

- Transfer of a signal to a destination process occurs in two distinct steps

1. **Sending a signal**
   - kernel **sends** (delivers) a signal to a destination process by **updating some state** in the context of the destination process when...
     1. the kernel has **detected a system event** such as a divide-by-zero error or the termination of a child process

2. a process has invoked the **kill** function to explicitly request the kernel to send a signal to the destination process

2. **Receiving a signal**
    - destination process **receives** a signal **when it is forced by the kernel to react** in some way to the delivery of the signal
    - the process **can** either **ignore** the signal, **terminate** , or **catch** the signal by executing a user-level function called a ***signal handler***

- Pending signal -- a signal that has been sent but not yet received
    - at any point in time, there can be **at most one** pending singal of a particular type
    - if a process has a **pending signal of type** k, then any **subsequent** signals of type k sent to that process are **not queued**

      --> simply discarded

        - a process can selectively **block** the receipt of certain signals
            - when a signal is blocked, it can still be delivered, but the resulting **pending signal will not be received until the process unblocks the signal**
    - is recevied at most once
        - for each process, the kernel maintains the set of pending signals in the **pending bit vector** & set of blocked signals in the **blocked bit vector**

¶8.5.2 Sending Signals

Process Groups

- Every process belongs to exactly one **process group**, identified by a positive integer **process group ID**
    - **getpgrp** function returns the process group ID of the current process

    ```
    #include <unistd.h>
    pid_t getpgrp(void);
    // Returns: process group ID of calling process
    ```

    - By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the **setpgid** fucntion

```c
#include <unistd.h>
int setpgid(pid_t, pid, pid_t pgid);
  // Returns: 0 on success, -1 on error
```

--> changes the process group of process **pid** to **pgid**

- If **pid** is zero, the PID of the current process is used
- If **pgid** is zero, the PID of the process specified by **pid** is used for the process group ID

Sending Signals with the /bin/kill Program

- **/bin/kill** program sends an arbitrary signal to another process

  ```
  linux> /bin/kill -9 15213
  ```

  sends signal 9 (**SIGKILL**) to procss 15213

- **A negative PID** causes the signal to be sent to every process in process group PID

  ```
  linux> /bin/kill -9 -15213
  ```

  sends a SIGKILL signal to every proces in process groupo 15213

Sending Signals from the Keyboard

- Unix shells use the abstraction of a **job** to represent the processes that are created as a result of evaluating a single command line
- There is at most one foreground job and zero or more background jobs
- For example,

  ```
  linux> ls | sort
  ```

  - creates a foreground job consisting of two processes connected by a **Unix pipe**
    - one running the **ls** program
    - other running the **sort** program

--> the shell creates a separate process group for each job

--> typically, the process group ID is taken from one of the parent processes in the job

- Typing Ctrl+C at the keyboard causes the kernel to send a **SIGINT** singal to every process in the foreground process group
  - by default, the result is to terminate the foreground job
- Similarly, Ctrl+Z causes the kernel to send a **SIGTSTP** signal to every process in the foreground process group
  - by default, the result is to stop the foreground job

Sending Signals with the kill Function

- Processes send signals to other processes (including themselves) by calling the **kill** function

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
                    // Returns: 0 if OK, -1 on error
```

  - If **pid** > 0, then the **kill** function sends signal number **sig** to process **pid**
  - If **pid** $= 0$, then **kill** sends signal **sig** to every process in the group of the calling process, including the calling process itself
  - If **pid** < 0, then **kill** sends signal **sig** to every process in process group $|pid|$

[Example C code] -- example of a parent that uses the **kill** function to send a **SIGKILL** signal to its child

```
#include "csapp.h"
int main() {
  pid_t pid;

  /* Child sleeps until SIGKILL signal received, then dies */
  if ((pid = Fork()) == 0) {
    Pause();
    printf("control should never reach here!\n");
```

```c
      exit(0);
    }

    /* Parent sends a SIGKILL signal to a child */
    Kill(pid, SIGKILL);
    exit(0);
}
```

Sending Signals with the alarm Function

- Proess can send **SIGALRM** signal to itself by calling the **alarm** function

  ```c
  #include <unistd.h>
  unsigned int alarm(unsigned int secs);
    // Returns: remaining seconds of previous alarm, or 0 if no previous alarm
  ```

  - arranges for the kernel to send a **SIGALRM** signal to the calling process in **secs** seconds
    - if **secs** = 0, then no new alarm is scheduled
    - in any event, the call to **alarm** cancels any pending alarms and returns the number of seconds remaining until any pending alarm was due to be delivered, or 0 if there were no pending alarms

¶8.5.3 Receiving Signals

- When the kernel switches a process p from kernel mode to user mode (returning from a system call or completing a context switch), the kernel checks the set of **unblocked pending signals** for p
  - If the set is **empty** (usual case)
    - **kernel** passes control to the next instruction (I_{next}) in the logical control flow of p
  - If the set is **nonempty**
    - **kernel** chooses some signal k in the set (typically the smallest) and forces p to **receive** signal k

      --> triggers some **action** by the process

      - once the process completes the action, then control passes back to the next instruction in the logical control flow of p

- **Default action**
    - process terminates
    - process terminates and dumps core
    - process stops (suspends) until restarted by a **SIGCONT** signal
    - process ignores the signal

- A process **can modify the default action** associated with a signal by using the **signal** function
    - exceptions -- **SIGSTOP, SIGKILL** (default actions cannot be changed)

```c
#include <signal.h>
typedef void (*sighandler_t)(int);

sighandler_t signal(int signum, sighandler_t handler);
  // Returns: pointer to previous handler if OK, SIG_ERR on error (does not set errno)
```

- **signal** function can change the action associated with a signal **signum** in one of three ways:
    - if **handler** is **SIG_IGN**, then signals of type **signum** are ignored
    - if **handler** is **SIG_DFL**, then the action for signals of type **signum** reverts to the default action
    - Otherwise, **handler** is the address of a user-defined function, called a **signal handler**, that will be called whenever the process receives a signal of type **signum**
        - changing the default action by passing the address of a handler to the **signal** function is known as **installing the handler**
        - invocation of the handler is called **catching the signal**
        - execution of the handler is referred to as **handling the signal**

- When a process catches a signal of type k, the handler installed for signal k is invoked with a single integer argument set to k
    - allows the same handler function to catch different types of signals
- When the handler executes its **return** statement, control (usually) passes back to the instruction in the control flow where the process was interrupted by the receipt of the signal

[Example C code] -- program that catches the SIGINT signal that is sent whenever the user types Ctrl+C at the keyboard

```c
#include "csapp.h"

void sigint_handler(int sig) /* SIGINT handler */
{
  printf("Caught SIGINT!\n");
  exit(0);
}

int main() {
  /* Install the SIGINT handler */
  if (signal(SIGINT, sigint_handler) == SIG_ERR)
    unix_error("signal error");

  pause(); /* Wait for the receipt of a signal */

  return 0;
}
```

- default action for **SIGINT** is to immediately terminate the process
  - however, in this case, we modify the default behavior to catch the signal, print a message, and then terminate the process
- Signal handlers can be interrupted by other handlers

¶8.5.4 Blocking and Unblocking Signals

- **Implicit blocking mechanism**
  - by default, the kernel blocks any pending signals of the type currently being processed by a handler
- **Explicit blocking mechanism**
  - applications can **explicitly** block and unblock selected signals using the **sigprocmask** function and its helpers

```c
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *oldset);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
    // Returns: 0 if OK, -1 on error
int sigismember(const sigset_t *set, int signum);
    // Returns: 1 if member, 0 if not, -1 on error
```

- **sigprocmask** changes the set of currently blocked signals (the **blocked** bit vector)
  - specific behavior depends on the value of **how**
    - **SIG_BLOCK** -- add the signals in **set** to **blocked** (**blocked = blocked | set**)
    - **SIG_UNBLOCK** -- remove the signals in **set** from **blocked** (**blocked = blocked & ~set**)
    - **SIG_SETMASK** -- **blocked = set**
  - If **oldset** is non-NULL, the previous value of the **blocked** bit vector is stored in **oldset**
- **sigemptyset** -- initializes **set** to the empty set
- **sigfillset** -- adds every signal to **set**
- **sigaddset** -- adds **signum** to set
- **sigdelset** -- deletes **signum** from set
- **sigismember** -- returns 1 if **signum** is a member of set, and 0 if not

[Example C code]

```c
sigset_t mask, prev_mask;

Sigemptyset(&mask);
Sigaddset(&mask, SIGINT);

/* Block SIGINT and save previous blocked set. */
Sigprocmask(SIG_BLCOK, &mask, &prev_mask);

  // Code region that will not be interrupted by SIGINT
/* Restore previous blocked set, unblocking SIGINT */
Sigprocmask(SIG_SETMASK, &prev_mask, NULL);
```

¶8.5.5 Writing Signal Handlers

- What makes signal handling difficult?
    1. Handlers run concurrently with the main program and share the same global variables

       --> can interfere with the main progarm and with other handlers
    2. how/when signals are received is often counterintuitive
    3. different systems can have different signal-handling semantics

Safe Signal Handling

- If a handler and the main program access the same global data structure concurrently, then the results can be unpredictable and often fatal
- To avoid concurrency errors,
    0. Keep handlers as simple as possible
        - make handler set a global flag and return immediately
        - let all processing performed by the main program

    1. Call only async-signal-safe functions in your handlers
        - **async-signal-safe** functions can be **safely called** from a **signal handler** either because
            - it is **reentrant** (e.g. accesses only local vars)
            - it cannot be interrupted by a signal handler
        - The only safe way to generate output from a signal handler is to use **write** function
            - **printf, sprintf** are not safe
        - **_exit** is an async-signal-safe variant of **exit**

    2. Save and restore **errno**
        - many of the Linux async-signal-safe functions set **errno** when they return with an error
            - calling such functions inside a handler might interfere with other parts of the program that rely on **errno**
                - save **errno** to a local variable on entry to the handler and restore it before the handler returns

- only necessary if the handler returns
- not necessary if the handler terminates the process by calling **_exit**

3. Protect accesses to shared global data structures by blocking all signals
    - if a handler shares a global data structure with the main program or with other handlers, handlers and main program should **temporarily block all signals** while accessing that data structure

4. Declare global variables with **volatile**
    - tell the compiler **not to cache a variable** by declaring it with the **volatile** type qualifier

      e.g. volatile int g;

      --> forces the compiler to read the value of g from memory each time it is referenced in the code

5. Declare **flags** with **sig_atomic_t**
    - handler records the receipt of the signal by writing to a global **flag**
    - C provides an integer data type, **sig_atomic_t** -- reads and writes are guaranteed to be **atomic (uninterruptible)** because they can be implemented with a single instruction

      volatile sig_atomic_t flag;

        - can safely read from and write to **sig_atomic_t** variables without temporarily blocking signals

Correct Signal Handling

- Pending signals are not queued
- **pending** bit vector contains exactly one bit for each type of signal

  --> there can be **at most one** pending signal of any particular type

    - if two signals of type k are sent to a destination process while signal k is blocked (because the destination process is currently executing a handler for signal k), then the second signal is **discarded**

[Example C code]

```c
void handler1(int sig) {
  int olderrno = errno;

  if ((waitpid(-1, NULL, 0)) < 0)
    sio_error("waitpid error");
  Sio_puts("Handler reaped child\n");
  Sleep(1);
  errno = olderrno;
}

int main() {
  int i, m;
  char buf[MAXBUF];

  if (signal(SIGCHLD, handler1) == SIG_ERR)
    unix_error("signal error");

  /* Parent creates children */
  for (i = 0; i < 3; i++) {
    if (Fork() == 0) {
      printf("Hello from child %d\n", (int)getpid());
      exit(0);
    }
  }

  /* Parent waits for terminal input and then processes it */
  if ((n = read(STDIN_FILENO, buf, sizeof(buf))) < 0)
    unix_error("read");

  printf("Parent processing input\n");
  while(1)
    ////

  exit(0);
}
```

- This code failed to account for the fact that signals are not queued
- First signal is received and caught by the parent
    - Second signal is delivered while the handler is still processing the first signal

      --> second signal added to the set of pending signals (not received yet since SIGCHLD signals are blocked by the SIGCHLD handler)

        - Third signal arrives while the handler is still processing the first signal

--> Third signal **discarded**

--> Signals cannot be used to count the occurrence of events in other processes

[Example C code w/o forementioned error]

```c
1  void handler2(int sig) {
2    int olderrno = errno;
3
4    while (waitpid(-1, NULL, 0) > 0) {
5      Sio_puts("Handler reaped child\n");
6    }
7    if (errno != ECHILD)
8      Sio_error("waitpid error");
9    Sleep(1);
10   errno = olderrno;
11 }
```

--> When all of the children have been reaped, the next call to waitpid returns -1, and sets errno to ECHILD

--> L7 checks that the waitpid function terminated normally

Portable Signal Handling

- Different systems have different signal-hanling semantics
    - Semantics of the **signal** function varies
        - some older Unix systems restore the action for signal k to its default after signal k has been caught by a handler
            - hanlder must explicitly reinstall itself each time it runs
    - System calls can be interrupted
        - **slow system calls** -- system calls that can potentially block the process for a long period
            - on some older versions of Unix, slow system calls that are interrupted when a handler catches a signal do not resume when the signal handler returns

              --> return immediately to the user with an error condition and **errno** set to **EINTR**
            - programmers must include code that manually restarts interrupted system calls

- To deal with abovementioned issues, the Posix standard defines the **sigaction** function, which allows users to **clearly specify the signal-handling semantics** they want when they install a handler

```
#include <signal.h>
int sigaction(int signum, struct sigaction *act, struct sigaction *oldact);
    // Returns: 0 if OK, -1 on error
```

--> inconvenient since it requires the user to set the entries of a complicated structure

--> define a wrapper function: **Signal** that calls **sigaction**

```
handler_t *Signal (int signum, handler_t *handler) {
  struct sigaction action, old_action;

  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = SA_RESTART;

  if (sigaction(signum, &action, &old_action) < 0)
    unix_error("Signal error");
  return (old_action.sa_handler);
}
```

- **Signal** wrapper installs a signal handler with the following signal handling semantics:
  - only signals of the type currently being processed by the handler are blocked
  - as with all signal implementations, signals are not queued
  - interrupted system calls are automatically restarted whenever possible
  - once the signal handler is installed, it remains instaled until **Signal** is called with a **handler** argument of either **SIG_IGN** or **SIG_DFL**