System Calls, Kernel Mode, and Process Implementation

CS 256/456
Dept. of Computer Science, University of Rochester

Recap of the Last Class

- Processes
  - Process concept
  - Operations on processes
- Signals
  - User-level events
- Pipes
  - Inter-process communication

Signals

- A signal is a small message that notifies a process that an event of some type has occurred in the system.
  - Kernel abstraction for exceptions and interrupts.
  - Sent from the kernel (sometimes at the request of another process) to a process.
  - Different signals are identified by small integer ID’s
  - The only information in a signal is its ID and the fact that it arrived.

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt from keyboard (ctl-c)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`
- Different values for `handler`:
  - `SIG_IGN`: ignore signals of type `signum`
  - `SIG_DFL`: revert to the default action on receipt of signals of type `signum`
- Otherwise, `handler` is the address of a `signal handler`
  - Called when process receives signal of type `signum`
  - Referred to as "installing" the handler.
  - Executing the handler is called "catching" or "handling" the signal.
- When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal.

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb? \n");
    sleep(2);
    printf("Well...\n");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler); /* installs ctl-c handler */
    while(1) {
    }
}
```

A Program That Reacts to Externally Generated Events (ctrl-c)

```
#include <stdlib.h>
#include <stdio.h>
#include <signal.h>

void handler(int sig) {
    printf("You think hitting ctrl-c will stop the bomb? \n");
    sleep(2);
    printf("Well...\n");
    fflush(stdout);
    sleep(1);
    printf("OK\n");
    exit(0);
}

main() {
    signal(SIGINT, handler);/* installs ctl-c handler */
    while(1) {
    }
}
```

A Program That Reacts to Internally Generated Events

```
#include <stdio.h>
#include <signal.h>

int beeps = 0;
/* SIGALRM handler */
void handler(int sig) {
    printf("BEEP\n");
    fflush(stdout);
    if (++beeps < 5)
        alarm(1);
    else {
        printf("BOOM!\n");
        exit(0);
    }
}

main() {
    signal(SIGALRM, handler);
    alarm(1); /* send SIGALRM in 1 second */
    while (1) {
        /* handler returns here */
    }
}
```

Non-queuing Nature of Signals

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal
```
int count = 0;
void child_handler(int sig) {
    int child_status;
    pid_t pid = wait(&child_status);
    count--;
    printf("Received signal %d from process %d\n", sig, pid);
}

void fork14() {
    pid_t pid[N];
    int i, child_status;
    count = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Exit */
            exit(0);
        }
    while (count > 0)
        pause(); /* Suspend until signal occurs */
}
```

linux> a.out
BEEP
BEEP
BEEP
BEEP
BEEP
BOOM!

bass>
Unix Pipes

- A unidirectional data channel that can be used for interprocess communication
- Treated as a special type of file, accessed using read() and write()
- Cannot be accessed from outside the process that created it unless inherited (by a child)
- Pipe ceases to exist once closed or when process terminates
- System calls
  - pipe (int fd[])
  - dup2

Today

- System calls and the interrupt interface
- Process implementation
- Context switches and the scheduling process
- Inter-process communication
  - Shared memory
  - Thread
    - Thread concept
    - Multithreading models
    - Types of threads

System Protection

- User programs (programs not belonging to the OS) are generally not trusted
  - A user program may use an unfair amount of resource
  - A user program may maliciously cause other programs or the OS to fail
- Need protection against untrusted user programs; the system must differentiate between at least two modes of operations
  1. User mode - execution of user programs
     - untrusted
     - not allowed to have complete/direct access to hardware resources
  2. Kernel mode (also system mode or monitor mode) - execution of the operating system
     - trusted
     - allowed to have complete/direct access to hardware resources
- Hardware support is needed for such protection

Unix Startup: Step 1

1. Pushing reset button loads the PC with the address of a small bootstrap program.
2. Bootstrap program loads the boot block (disk block 0).
3. Boot block program loads kernel binary (e.g., /boot/vmlinux)
4. Boot block program passes control to kernel.
5. Kernel handcrafts the data structures for process 0.
Unix Startup: Step 2

init [1]

/etc/inittab

init forks and execs daemons per /etc/inittab, and forks and execs a getty program for the console

Daemons e.g. ftpd, httpd

getty

Unix Startup: Step 3

init [1]

login

The getty process execs a login program

Unix Startup: Step 4

init [1]

login reads login and passwd. if OK, it execs a shell. if not OK, it execs another getty

tcsh

Transition between User/Kernel Mode

• When does the machine run in kernel mode?
  - after machine boot
  - interrupt handler
  - system call
  - exception

Bootstrap

Interrupt/syscall/exception

Kernel

User

To user mode
I/O Device Controllers

- I/O devices have both mechanical component & electronic component
- The electronic component is the device controller
  - It contains control logic, command registers, status registers, and on-board buffer space

I/O Ports & Memory-Mapped I/O

I/O methods:
- Separate I/O and memory space; special I/O commands (IN/OUT)
- Memory-mapped I/O

Issues with them:
- Convenience/efficiency when using high-level language
- Protection mechanisms
- Data caching

Direct Memory Access (DMA)

- Are the addresses CPU sends to the DMA controller virtual or physical addresses?
- Can the disk controller directly read data into the main memory (bypassing the controller buffer)?

The Device-Controller-Software Relationship

- Software Program to manage device controller
- System software (part of OS)

Device controller:
- Contains control logic, command registers, status registers, and on-board buffer space
- Firmware/hardware
I/O Operations

- How is I/O done?
  - I/O devices are much slower than CPU

- Synchronous (polling)
  - After I/O starts, busy-wait while polling the device status register until it shows the operation completes

- Asynchronous (interrupt-driven)
  - After I/O starts, control returns to the user program without waiting for I/O completion
  - Device controller later informs CPU that it has finished its operation by causing an interrupt
  - When an interrupt occurs, current execution is put on hold; the CPU jumps to a service routine called an "interrupt handler"

Memory Protection

- Goal of memory protection?
  - A user program can't use arbitrary amount of memory
  - A user program can't access data belonging to the operating system or other user programs

- How to achieve memory protection?
  - Indirect memory access
    - Memory access with a virtual address which needs to be translated into physical address
  - Add two registers that determine the range of legal addresses a program may access:
    - Base register - holds the smallest legal physical memory address
    - Limit register - contains the size of the range
    - Memory outside the defined range is protected

Hardware Address Protection

- Address of each memory address is checked against "base" and "base+limit"
- Trap to the OS kernel if it falls outside of the range (an exception)

Protection of I/O Devices

- User programs are not allowed to directly access I/O devices
  - Special I/O instructions can only be used in kernel mode
  - Controller registers can only be accessed in kernel mode

- So device drivers, I/O interrupt handlers must run in kernel mode

- User programs perform I/O through requesting the OS (using system calls)
System Call Using the **Trap** Instruction

```c
read();

read() {
    trap N_SYS_READ()
}
```

```
sys_read() {
    /* system function */
    ...
    return;
}
```

**CPU Protection**

- **Goal of CPU protection**
  - A user program can't hold the CPU for ever
- **Timer** - interrupts computer after specified period to ensure the OS kernel maintains control
  - Timer is decremented every clock tick
  - When timer reaches the value 0, an interrupt occurs
  - CPU time sharing is implemented in the timer interrupt

**Processes**

- **Def**: A *process* is an instance of a running program.
  - Not the same as "program" or "processor"
- Process provides each program with two key abstractions:
  - Logical control flow
    - Each program seems to have exclusive use of the CPU.
  - Private 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

**Process Management**

- A *process* is a program in execution
  - Unit of work - A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task
  - Protection domain
- **OS responsibilities for process management**:
  - Process creation and deletion
  - Process scheduling, suspension, and resumption
  - Process synchronization, inter-process communication
Process and Its Image

- An operating system executes a variety of programs:
  - A program that browses the Web
  - A program that serves Web requests
- Process - a program in execution.
- A process's state/image in a computer includes:
  - User-mode address space
  - Kernel data structure
  - Registers (including program counter and stack pointer)
- Address space and memory protection
  - Physical memory is divided into user memory and kernel memory
  - Kernel memory can only be accessed when in the kernel mode
  - Each process has its own exclusive address space in the user-mode memory space (sort-of)

Private Address Spaces

- Each process has its own private address space.

User-mode Address Space

- Text: program code, instructions
- Data: initialized global and static variables (those data whose size is known before the execution)
- BSS (block started by symbol): uninitialized global and static variables
- Heap: dynamic memory (those being malloc-ed)
- Stack: local variables and other stuff for function invocations

Process Control Block (PCB)

- OS data structure (in kernel memory) maintaining information associated with each process.
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - Information about open files
  - maybe kernel stack?
Queues for PCBs

- Ready queue - set of all processes ready for execution.
- Device queues - set of processes waiting for an I/O device.
- Process migration between the various queues.

Process State

- As a process executes, it changes state
  - new: The process is being created
  - ready: The process is waiting to be assigned to a process
  - running: Instructions are being executed
  - waiting: The process is waiting for some event to occur
  - terminated: The process has finished execution

Process Creation

- When a process (parent) creates a new process (child)
  - Execution sequence?
  - Address space sharing?
  - Open files inheritance?
  - ...
- UNIX examples
  - fork system call creates new process with a duplicated copy of everything.
  - exec system call used after a fork to replace the process’ memory space with a new program.
  - child and parent compete for CPU like two normal processes.
- Copy-on-write

Unix Process Hierarchy

- [0]
- init [1]
- Daemon e.g. httpd
- Login shell
- Child
- Child
- Child
- Grandchild
- Grandchild
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**.

![Context Switching Diagram](image)

Scheduling: Transferring Context Blocks

**Coroutines**

transfer(other)

- save all callee-saves registers on stack, including ra and fp
- current := sp
- current := other
- sp := *current
- pop all callee-saves registers (including ra, but NOT sp!)
- return (into different coroutine!)

Uniprocessor Scheduling

- Use **Ready List** to reschedule voluntarily (cooperative threading)
  - reschedule:
    - t : cb := dequeue(ready_list)
    - transfer(t)
  - yield:
    - enqueue(ready_list, current)
    - reschedule
  - sleep_on(q):
    - enqueue(q, current)
    - reschedule

Preemption

- Use timer interrupts or signals to trigger involuntary yields
- Protect scheduler data structures by disabling/reenabling prior to/after rescheduling
  - yield:
    - disable_signals
    - enqueue(ready_list, current)
    - reschedule
    - re-enable_signals
Process Termination

- Process executes last statement and gives the control to the OS (exit)
  - Notify parent if it is waiting
  - Deallocate process's resources
- The OS may forcefully terminate a process.
  - Software exceptions
  - Receiving certain signals

CPU Switch From Process to Process

- When can the OS switch the CPU from one process to another?
- Which one to switch to? - scheduling