1/26/18

CS 355
Operating Systems
Processes, Unix Processes and System Calls

Process
• User types command like “run foo” at keyboard
  – I/O device driver for keyboard and screen
• Command is parsed by command shell
• Executable program file “foo” is located on disk
  – file system, I/O device driver for disk
• Contents are loaded into memory and control transferred ==> process comes alive!
  – disk device driver, relocating loader, memory management
• During execution, process may call OS to perform I/O: console, disk, printer, etc.
  – system call interface, I/O device drivers
• When process terminates, memory is reclaimed
  – memory management

What is the Difference btw a Process and a Program?
• A process is a program in execution with associated data and execution context
• Each running instance of a program is a separate process

Keeping Track of Processes
• For each process, OS maintains a data structure, called the process control block (PCB). The PCB provides a way of accessing all information relevant to a process:
  – This data is either contained directly in the PCB, or else the PCB contains pointers to other system tables.
• All current processes (PCBs) are stored in a system table called the process table.
  – Either a linked list or an array, usually a linked list.

Process States
• Possible process states
  – Running: executing
  – Blocked: waiting for I/O
  – Ready: waiting to be scheduled

When are processes created?
• System initialization (many daemons for processing email, printing, web pages etc.)
• Execution of a process creation system call by a running process (fork or CreateProcess)
• User request to create a new process (executing new applications)

List of all active processes: ps (Unix), Ctl-Alt-Del (Windows)
When are processes terminated?

- Normal exit (voluntary)
- Error exit (voluntary)
- Fatal error (involuntary), due to bugs
- Killed by another process (involuntary)

Process Context/Image

- Main memory – logically distinct regions of memory:
  - Text segment: contains executable code (typically read-only)
  - Data/BSS segment: initialized and uninitialized globals, statics and constants
  - Heap: storage area for dynamically allocated data structure, e.g., lists, trees
  - Stack: run-time stack of activation records
- Registers: general registers, PC, SP, PSW, segmentation registers
- Other information:
  - open files table, status of ongoing I/O
  - process status (running, ready, blocked), user id, ...

Memory Layout of a Process

- CPU
- PSW
- Stack Pointer
- Program Counter
- Code/Text
- Data/BSS
- Heap
- Stack

Recall: Where do local variables go? Where do global variables go? How about command line arguments and env vars? How is information exchanged between a program and its subroutines?

Program in Memory

- What is stored in memory?
  - Code
  - Constants
  - Global and static variables
  - Local variables
  - Dynamic memory (malloc)

```c
int SIZE;
char* f(void) {
    char *c;
    SIZE = 10;
    c = malloc(SIZE);
    return c;
}
```

A Typical Stack Frame

- int foo(int arg1, int arg2);
  - Two local vars

```c
int foo(int arg1, int arg2);
```
When does OS get invoked?

- Operating system gets control in two ways
  - A user process calls OS via a system call
  - An interrupt aborts the current user process
- System stack can be used to exchange information between OS and user processes
- Recall: Mode bit in PSW is set only when OS is running

Interrupts

- An interruption of the normal processing of processor.
- Interrupts cause the CPU to suspend its current computation and take up some new task.
- Reasons for interrupts and/or traps:
  - control of asynchronous I/O devices
  - CPU scheduling
  - exceptional conditions (e.g., div. by zero, page fault, illegal instruction) arising during execution
- Interrupts are essentially what drives an OS. We can view the OS as an event-driven system, where an interrupt is an event.
- Bounding interrupt handling latency is important for real-time systems.

Interrupt Handling

- The servicing of an interrupt is known as interrupt handling.
- An integer is associated with each type of interrupt. When an interrupt occurs, the corresponding integer is supplied to the OS usually by the hardware (in a register).
- The OS maintains a table, known as the interrupt vector, that associates each interrupt's id with the starting address of its service routine.

```
<table>
<thead>
<tr>
<th>Interrupt No</th>
<th>Interrupt Handler</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>clock</td>
</tr>
<tr>
<td>1</td>
<td>disk</td>
</tr>
<tr>
<td>2</td>
<td>tty</td>
</tr>
<tr>
<td>3</td>
<td>dev</td>
</tr>
<tr>
<td>4</td>
<td>soft</td>
</tr>
<tr>
<td>5</td>
<td>other</td>
</tr>
</tbody>
</table>
```

Interrupt interrupts!

- On Intel Pentium, hardware interrupts are numbered 0 to 255
  - 0: divide error
  - 1: debug exception
  - 2: null interrupt
  - 6: invalid opcode
  - 12: stack fault
  - 14: page fault
  - 16: floating-point error
  - 19-31: Intel reserved (not available?)
  - 32-255: maskable interrupts (device controllers, software traps)
- Issue: can CPU be interrupted while an OS interrupt handler is executing?
- Maskable interrupts can be disabled, plus there is a priority among interrupts
- If interrupted-enabled flag is set, race condition may occur when a higher priority interrupt interrupts a lower one in the wrong time

Typical interrupt handling sequence

- Interrupt initiated by I/O device signaling CPU, by exceptional condition arising, through execution of special instruction, etc.
- CPU suspends execution of current instruction stream and saves the state of the interrupted process (on system stack).
- The cause of the interrupt is determined and the interrupt vector is consulted in order to transfer control to the appropriate interrupt handler.
- Interrupt handler performs whatever processing is necessary to deal with the interrupt.
- Previous CPU state is restored (popped) from system stack, and CPU returns control to interrupted task.

Example: Servicing a Timer Interrupt

- Timer device is used in CPU scheduling to make sure control is returned to system every so often (e.g., 1/60 sec.)
- Typically, timer has a single register that can be loaded with an integer indicating a particular time delay (# of ticks).
- Once loaded, timer counts down and when 0 is reached, an interrupt is generated.
- Interrupt handler might do the following:
  - update time-of-day information
  - signal any processes that are "asleep" and awaiting this alarm
  - call the CPU scheduler
- Control returns to user mode, possibly to a different process than the one executing when the interrupt occurred.
Example: Servicing a Disk Interrupt

- When disk controller completes previous transfer, it generates an interrupt.
- Interrupt handler changes the state of a process that was waiting for just-completed transfer from wait-state to ready-state.
- It also examines queue of I/O requests to obtain next request.
- I/O is initiated on next request.
- CPU scheduler called.
- Control returned to user mode.

System Calls

- To obtain “direct access” to operating system services, user programs must make system calls
  - file system
  - I/O routines
  - memory allocate & free routines
- System calls are special, and in fact, are treated like interrupts.
- Programs that make system calls are traditionally called “system programs” and were traditionally implemented in assembly language
  - Win32 API in Windows
- Each instruction set has a special instruction for making system calls:
  - SVC (IBM 360/370)
  - trap (PDP 11)
  - tw (PowerPC) - trap word
  - tcc (Sparc)
  - break (MIPS)

Timeout

- Interrupts, traps, system calls. What’s the difference?

System Calls

- User Mode bit = 0
- OS Mode bit = 1

Application Program

Library Routine

OS Routine

Device Controller

Special Instructions

Software

Hardware

Steps in read() System Call

- User program pushes parameters on to stack 1-3
- User program executes CALL instruction to invoke library routine read in assembly language 4
- Read routine sets up the register for system call number 5
- Read routine executes TRAP instruction to invoke OS 6
- Hardware sets the mode-bit to 1, saves the state of the executing read routine, and transfers control to a fixed location in kernel 6
- Kernel code, using a table look-up based on system call number 7, transfers control (dispatches) to correct system call handler 8
Steps in `read()` System Call (cont)

- OS routine copies parameters from user stack, sets up device driver registers, and executes the system call using privileged instructions.
- OS routine can finish the job and return, or decide to suspend the current user process to avoid waiting.
- Upon return from OS, hardware resets the mode-bit.
- Control transfers to the read library routine and all registers are restored.
- Library routine terminates, transferring control back to original user program.
- User program increments stack pointer to clear the parameters.

Issues

- Can a system call interrupt another system call?
- Can a system call interrupt an interrupt?
- Can an interrupt interrupt a system call?

Creating processes in UNIX

- Important source of information on UNIX is `man`.
- A Unix process is created by another.
  - a newly created process is the “child” of the “parent” process that created it.
  - every process has exactly one parent.
  - a process may create any number of child processes.
- Processes have a unique PID (process ID).
  - index to the PCB in the process table.
- Processes are created in UNIX with the `fork()` system call.

Process Hierarchies

- Processes form a hierarchy.
  - UNIX calls this a process group.
- Signals can be sent to all processes of a group.
- Windows has no concept of process hierarchy.
  - all processes are created equal.

Initialization

- At the root of the family tree of processes is the special process `init`:
  - created as part of the bootstrapping procedure.
  - pid = 1.
  - among other things, `init` spawns a child to listen to each terminal, so that a user may log on.
  - do “`man init`” to learn more about it.
  - all processes are children of `init`.

UNIX Process Control

UNIX provides a number of system calls for process control including:

- `fork` - used to create a new process.
- `exec` - to change the program a process is executing.
- `exit` - used by a process to terminate itself normally.
- `abort` - used by a process to terminate itself abnormally.
- `kill` - used by one process to kill or signal another.
- `wait` - to wait for termination of a child process.
- `sleep` - suspend execution for a specified time interval.
- `getpid` - get process id.
- `getppid` - get parent process id.
The fork System Call

• The fork() system call creates a "clone" of the calling process.

• Identical in every respect except
  – the parent process is returned a non-zero value (namely, the pid of the child)
  – the child process is returned zero.
  – The pid returned to the parent can be used by parent in a wait or kill system call.

• What good is this?
  – write code to behave differently if you are child

Example

1. #include <unistd.h>
2. main(){
3.   pid_t pid;
4.   printf("Just one process so far\n\n");
5.   pid = fork();
6.   if (pid == 0) /* code for child */
7.      printf("I'm the child\n\n");
8.   else if (pid > 0) /* code for parent */
9.      printf("The parent, child pid =%d\n",
10.         pid);
11.   else /* error handling */
12.      printf("Fork returned error code\n");
13. }

Example

int main(){
    int x=0;
    fork();
    x++;
    printf("The value of x is %d\n", x);
}
What is the output?

Replacing a Process: exec

• The exec system call replaces a process with a new program
  – it does not create any new process
  – the new program is specified by the name of the file containing the executable and arguments

• The calling process stops running as soon as it calls exec if the executable can be run

• Usually runs after fork()

wait()

• Often a process needs to wait for a child to finish first
  – what the shell usually does when you type something
• wait() blocks until one of the caller's children terminates then returns with status or error
• If status is available for more than one child, the order in which it is reported is unspecified

• pid_t waitpid(pid_t pid, int *
  stat�Loc, int opEnum);
Example

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

int main() {
    int status;
    if (fork() == 0) exit(EXIT_SUCCESS);
    wait (&status);
    pexit(status);
    if (fork() == 0) abort();       /* SIGABRT */
    wait (&status);
    pexit(status);
    if (fork() == 0) status /= 0;   /* SIGFPE */
    wait (&status);
    pexit(status);
}
```

Example

```c
void pexit(int stat) {
    if (WIFEXITED(status))
        printf("Normal termination, exit status = %d\n", WEXITSTATUS(status));
    if (WIFSIGNALED(status))
        printf("Abormal termination, signal number = %d\n", WTERMSIG(status));
    if (WIFSTOPPED(status))
        printf("Stopped, signal number = %d\n", WSTOPSIG(status));
}
```

waitpid()

- `pid_t waitpid(pid_t pid, int *stat_loc, int options);`
- `waitpid()` will wait until a specified (by pid) child terminates
- `waitpid()` also has the option to not block
  - `pid == -1` waits for any child
  - `option == NOHANG` non-blocking
- `option == 0` blocking
- `waitpid(-1,&status,0) == wait(&status)`

How shell executes a command

```plaintext
Parent shell
fork
exec
wait
Child
```

- When you type a command, the shell forks a clone of itself
- The child process makes an exec call, which causes it to stop executing the shell and start executing your command
- The parent process, still running the shell, waits for the child to terminate

exec and Friends

- `exec` does not return, i.e. the program calling `exec` is gone forever!
- `exec` is really a family of system calls, each differ slightly in the way the process arguments are given
  - `execl`, `execlp`, `execlp`, `execv`, `execvp`, `execve`
- The `e`s expect a list of pointers to strings as arguments
  - `const char *exename, const char *arg0, const char *arg1, …, const char *argn`
- The `v`s expect an array of pointers to strings as arguments
- The `p`s will duplicate the shell’s path searching effort
- The `e`s allow an additional parameter specifying the environment variables
- All eventually make a call to `execve` (which is the real system call), the others are C lib front ends

Process Termination

- A self-terminating process sends its parent a SIGCHLD signal and waits for termination status code to be accepted
- A process can be killed by another process using `kill`:
  - `kill(pid, sig)` - sends signal `sig` to process with process-id `pid`. One signal is SIGKILL (terminate the target process immediately).
- When a process terminates, all the resources it owns are reclaimed by the system:
  - PCB reclaimed
  - its memory is deallocated
  - all open files closed and Open File Table reclaimed.
### Orphans and Zombies

- Orphan process – a process whose parent died before it did
  - gets adopted by init
  - killing init is equivalent to killing the entire OS
- Zombie process – a process that is waiting for its parent to accept it's status code
  - shows up with 'Z' in ps

### System Calls and Error Handling

- All system calls return -1 if failed
- **errno** – the global variable that holds the integer error code for the last system call
- **perror(const char *str)** – a library function which describes the last system call error
  - Every process has errno initialized to 0
  - A successful system call never affects errno
  - A failed system call always overwrites errno
  - #include <sys/errno.h>

### Example

```c
#include <stdio.h>
#include <errno.h>
#include <unistd.h>

int main(int argc, char **argv) {
    int rv; char str[50];
    printf("Before execlp, pid=%d
", getpid());
    rv = execlp("blah", (char *) 0);
    sprintf("%s failed", argv[0]);
    if (rv == -1)
        perror(str);
    return 0;
}
```

### Signals and Signal Handlers

- A Unix system defines a fixed set of signals that can be raised by one process, thereby causing other processes to be interrupted and to (optionally) catch the signal.
- Interrupts also raise signals
- `<sys/signal.h>` lists signal handling functions
- `man 7 signal` gives the list of #define signals
  - man sections: 1 general, 2 sys calls, 3 C lib, 7 conventions, 8 sys admin commands
- When a signal is received, it is passed on to its handler, which can be user created and then registered

### Some System Calls For Process Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pid = fork()</td>
<td>Create a child process identical to the parent</td>
</tr>
<tr>
<td>pid = waitpid(, &amp;status, options)</td>
<td>Wait for a child to terminate</td>
</tr>
<tr>
<td>exit(status)</td>
<td>Terminate process execution and return status</td>
</tr>
</tbody>
</table>

### Some System Calls For File Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fd = open(file, how, ...)</td>
<td>Open a file for reading, writing or both</td>
</tr>
<tr>
<td>s = close(fd)</td>
<td>Close an open file</td>
</tr>
<tr>
<td>n = read(fd, buffer, rbytes)</td>
<td>Read data from a file into a buffer</td>
</tr>
<tr>
<td>n = write(fd, buffer, rbytes)</td>
<td>Write data from a buffer into a file</td>
</tr>
<tr>
<td>s = stat(name, &amp;buf)</td>
<td>Get a file's status information</td>
</tr>
</tbody>
</table>
Some System Calls For Directory Management

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mkdir(name, mode)</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir(name)</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>link(name1, name2)</td>
<td>Create a new entry, named, pointing to name1</td>
</tr>
<tr>
<td>unlink(name)</td>
<td>Remove a directory entry</td>
</tr>
<tr>
<td>mount(special, name, flag)</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>umount(special)</td>
<td>Unmount a file system</td>
</tr>
</tbody>
</table>

Some System Calls For Miscellaneous Tasks

<table>
<thead>
<tr>
<th>Call</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>chdir(name)</td>
<td>Change the working directory</td>
</tr>
<tr>
<td>chmod(name, mode)</td>
<td>Change a file’s protection bits</td>
</tr>
<tr>
<td>kill(pid, signal)</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>elapsed(time, &amp;seconds)</td>
<td>Get the elapsed time since Jan. 1, 1970</td>
</tr>
</tbody>
</table>

Unix and Win32 System Calls

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Win32</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>CreateProcess</td>
<td>Create a new process</td>
</tr>
<tr>
<td>wait</td>
<td>WaitForSingleObject</td>
<td>Wait for a process to end</td>
</tr>
<tr>
<td>exec</td>
<td>CreateProcess</td>
<td>Create process, exec a new process</td>
</tr>
<tr>
<td>exit</td>
<td>Terminate execution</td>
<td></td>
</tr>
<tr>
<td>creat</td>
<td>CreateFile</td>
<td>Create a file or open an existing file</td>
</tr>
<tr>
<td>close</td>
<td>CloseHandle</td>
<td>Close a file</td>
</tr>
<tr>
<td>read</td>
<td>ReadFile</td>
<td>Read data from a file</td>
</tr>
<tr>
<td>write</td>
<td>WriteFile</td>
<td>Write data to a file</td>
</tr>
<tr>
<td>ioctl</td>
<td>GetFileAttributesEx</td>
<td>Get various file attributes</td>
</tr>
<tr>
<td>mkdir</td>
<td>CreateDirectory</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>rmdir</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>unistack</td>
<td>CloseFile</td>
<td>Close the file pointer</td>
</tr>
<tr>
<td>readlink</td>
<td>GetFileAttributesEx</td>
<td>Get various file attributes</td>
</tr>
<tr>
<td>truncate</td>
<td>RemoveDirectory</td>
<td>Remove an empty directory</td>
</tr>
<tr>
<td>chmod</td>
<td>CreateDirectory</td>
<td>Create a new directory</td>
</tr>
<tr>
<td>mount</td>
<td>MountFile</td>
<td>Mount a file system</td>
</tr>
<tr>
<td>umount</td>
<td>UmountFile</td>
<td>Unmount a file system</td>
</tr>
<tr>
<td>setuid</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>setgid</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>setsockopt</td>
<td>SetCurrentDirectory</td>
<td>Change the current working directory</td>
</tr>
<tr>
<td>gettime</td>
<td>GetCurrentDirectory</td>
<td>Get the current time</td>
</tr>
</tbody>
</table>

Devil is in the Details

- Basic principles of system calls, signals and interrupts are not difficult
- Details of syscall traps and interrupts vary from CPU to CPU
  - what is put on the stack and in what order
  - address spaces issues, etc
- That is why OS writers must also be experts of the CPU architecture
- And why you should read manual pages of the system calls in excess