CS 33

Architecture and the OS
The Operating System

My Program  Mary's Program  Bob's Program

OS
Processes

• Containers for programs
  – virtual memory
    » address space
  – scheduling
    » one or more threads of control
  – file references
    » open files
  – and lots more!
int main() {
    int i;
    int A[1];

    for (i=0; i++;)
        A[rand()] = i;
}
void runforever()
{
    while(1)
    {
    
int main()
    {
        runforever();
    
}
Architectural Support for the OS

• Not all instructions are created equal ...
  – non-privileged instructions
    » can affect only current program
  – privileged instructions
    » may affect entire system

• Processor mode
  – user mode
    » can execute only non-privileged instructions
  – privileged mode
    » can execute all instructions
Which Instructions Should Be Privileged?

- I/O instructions
- Those that affect how memory is mapped
- Halt instruction
- Some others ...
Who Is Privileged?

- You're not
  - and neither is anyone else
- The operating-system kernel runs in privileged mode
  - nothing else does
  - not even super user on Unix or administrator on Windows
Entering Privileged Mode

- How is OS invoked?
  - very carefully ...
  - strictly in response to interrupts and exceptions
  - (booting is a special case)
Interrupts and Exceptions

- Things don’t always go smoothly ...
  - I/O devices demand attention
  - timers expire
  - programs demand OS services
  - programs demand storage be made accessible
  - programs have problems
- Interrupts
  - demand for attention by external sources
- Exceptions
  - executing program requires attention
Exceptions

- **Traps**
  - “intentional” exceptions
    - execution of special instruction to invoke OS
  - after servicing, execution resumes with next instruction

- **Faults**
  - a problem condition that is normally corrected
  - after servicing, instruction is re-tried

- **Aborts**
  - something went dreadfully wrong ...
  - not possible to re-try instruction, nor to go on to next instruction

These definitions follow those given in “Intel® 64 and IA-32 Architectures Software Developer’s Manual” and are generally accepted even outside of Intel.
Interrupt and Exception Handling

- **Interrupt or exception invokess handler (in OS)**
  - via interrupt and exception vector
    - one entry for each possible interrupt/exception
      - contains
        - address of handler
  - code executed in privileged mode
    - but code is part of the OS

```
intrpt/excp
```

```
handler 0 addr
handler 1 addr
handler 2 addr
...
handler i addr
...
handler n-1 addr
```

```
handler i
```
The reason why there must be a separate stack in privileged mode is that the OS must be guaranteed that when it is executing, it has a valid stack, that the stack pointer must be pointing to a region of memory that can be used as a stack by the OS. Since while the program was running in user mode any value could have been put into the stack-pointer register, when the OS is invoked, it switches to a pre-allocated stack set up just for it.
When a trap or interrupt occurs, the current processor state (registers, including RIP, condition codes, etc.) are saved on the kernel stack. When the system returns back to the interrupted program, this state is restored.
Quiz 1

If an interrupt occurs, which general-purpose registers must be pushed onto the kernel stack?

a) none
b) callee-save registers
c) caller-save registers
d) all
Back to the x86 ...

- It’s complicated
  - more than it should be, but for historical reasons ...
- Not just privileged and non-privileged modes, but four “privilege levels”
  - level 0
    » most privileged, used by OS kernel
  - level 1
    » not normally used
  - level 2
    » not normally used
  - level 3
    » least privileged, used by application code
A Unix process’s address space appears to be three regions of memory: a read-only *text* region (containing executable code); a read-write region consisting of initialized *data* (simply called data), uninitialized data (*BSS* — a directive from an ancient assembler (for the IBM 704 series of computers), standing for Block Started by Symbol and used to reserve space for uninitialized storage), and a *dynamic area*; and a second read-write region containing the process’s user *stack* (a standard Unix process contains only one thread of control).

The first area of read-write storage is often collectively called the data region. Its dynamic portion grows in response to *sbrk* system calls. Most programmers do not use this system call directly, but instead use the *malloc* and *free* library routines, which manage the dynamic area and allocate memory when needed by in turn executing *sbrk* system calls.

The stack region grows implicitly: whenever an attempt is made to reference beyond the current end of stack, the stack is implicitly grown to the new reference. (There are system-wide and per-process limits on the maximum data and stack sizes of processes.)

Note that here we are discussing strictly single-threaded processes. Later we will discuss multi-threaded processes, whose address spaces contain multiple stacks.
Creating Your Own Processes

```c
#include <unistd.h>

int main() {
    pid_t pid;
    if ((pid = fork()) == 0) {
        /* new process starts
         * running here */
    }
    /* old process continues
     * here */
}
```
The only way to create a new process is to use the *fork* system call.
By executing *fork* the parent process creates an almost exact clone of itself that we call the child process. This new process executes the same text as its parent, but contains a copy of the data and a copy of the stack. This copying of the parent to create the child can be very time-consuming if done naively. Some tricks are employed to make it much less so.

Fork is a very unusual system call: one thread of control flows into it but two threads of control flow out of it, each in a separate address space. From the parent’s point of view, fork does very little: nothing happens to the parent except that fork returns the process ID (PID — an integer) of the new process. The new process starts off life by returning from fork, which it sees as returning a zero.
Quiz 2

The following program

a) runs forever
b) terminates quickly

```c
int flag;
int main() {
    while (flag == 0) {
        if (fork() == 0) {
            flag = 1;
            exit(0);  // causes process to terminate
        }
    }
}
```
The getpid routine returns the caller's process ID.
Putting Programs into Processes

```c
/* prog */
int main() {

fork

exec

if (fork() == 0) {
    execl("prog", 0);
}
```

```c
}
```
Note that the name of the program might be different from the name of the file that contains the program, but usually the name of the program is the last component of the file’s pathname.

Note that a null pointer, termed a *sentinel*, is used to indicate the end of the list of arguments.
Most of the time the purpose of creating a new process is to run a new (i.e., different) program. Once a new process has been created, it can use one of the exec system calls to load a new program image into itself, replacing the prior contents of the process’s address space. Exec is passed the name of a file containing a fully relocated program image (which might require further linking via a runtime linker). The previous text region of the process is replaced with the text of the program image. The data, BSS and dynamic areas of the process are “thrown away” and replaced with the data and BSS of the program image. The contents of the process’s stack are replaced with the arguments that are passed to the main procedure of the program.
A Random Program ...

```c
int main(int argc, char *argv[]) {
    int stop = atoi(argv[1]);
    for (int i = 0; i < stop; i++)
        printf("%d\n", rand());
    return 0;
}
```
Passing It Arguments

• From the shell
  
  $ random 12

• From a C program
  
  if (fork() == 0) {
    char *argv[] = {"random", "12", (void *)0};
    execv("./random", argv);
  }
Quiz 3

```c
if (fork() == 0) {
    char *argv[] = {"random", "12", (void *)0};
    execv("./random", argv);
    printf("random done\n");
}
```

The `printf` statement will be executed only if something goes wrong with `execv`.

a) yes
b) no
Receiving Arguments

```c
int main(int argc, char *argv[]) {
    int stop = atoi(argv[1]);
    for (int i = 0; i < stop; i++)
        printf("%d\n", rand());

    return 0;
}
```

Note that argv[0] is the name by which the program is invoked. argv[1] is the first “real” argument.
Not So Fast ...

- How does the shell invoke your program?

  if (fork() == 0) {
      char *argv = {"random", "12", (void *)0};
      execv("./random", argv);
  }

  /* what does the shell do here?? */
There’s a variant of `waitpid`, called `wait`, that waits for any child of the current process to terminate.
The exit code is used to indicate problems that might have occurred while running a program. The convention is that an exit code of 0 means success; other values indicate some sort of error. Note that if the main function returns, it returns to code that calls exit. The argument passed to exit in this case is the value returned by main.
Shell: To Wait or Not To Wait ...

```
$ who
    if ((pid = fork()) == 0) {
        char *argv[] = {"who", 0};
        execv("who", argv);
    }
    waitpid(pid, &status, 0);
    ...

$ who &
    if ((pid = fork()) == 0) {
        char *argv[] = {"who", 0};
        execv("who", argv);
    }
    ...
```
System Calls

- Sole direct interface between user and kernel
- Implemented as library routines that execute trap instructions to enter kernel
- Errors indicated by returns of −1; error code is in global variable \texttt{errno}

\begin{verbatim}
if (write(fd, buffer, bufsize) == -1) {
    // error!
    printf("error %d\n", errno);
    // see perror
}
\end{verbatim}

System calls, such as \texttt{fork, execv, read, write}, etc., are the only means for application programs to communicate directly with the kernel: they form an API (application program interface) to the kernel. When a program calls such a routine, it is actually placing a call to a subroutine in a system library. The body of this subroutine contains a hardware-specific trap instruction that transfers control and some parameters to the kernel. On return to this library return, the kernel provides an indication of whether or not there was an error and what the error was. The error indication is passed back to the original caller via the functional return value of the library routine. If there was an error, a positive-integer code identifying it is stored in the global variable \texttt{errno}. Rather than simply print this code out, as shown in the slide, one might instead print out an informative error message. This can be done via the \texttt{perror} routine.

The “hardware-specific trap instruction” is (or used to be) the “int” (interrupt) instruction on the x86. However, this instruction is now considered too expensive for such performance-critical operations as system calls. A new facility, known as “sysenter/sysexit” was introduced with the Pentium II processors (in 1997) and has been used by operating systems (including Windows and Linux) ever since. Its description is beyond the scope of this course.
System Calls

Kernel portion of address space

User portion of address space

other stuff

kernel stack

kernel text

write(id, buf, len)

trap into kernel
Each process has its own user address space, but there's a single kernel address space. It contains context information for each user process, including the stacks used by each process when executing system calls.