Operating Systems

Definition: *An* operating system *is the software that manages resources in a computer.*

Resources

A *resource* is (usually) hardware that needs to be accessed.

There are rules for accessing resources . . .
. . . rules enforced by the OS.

Rules might . . .
. . . restrict access to sensitive resources . . .
. . . or control access to finite resources.

Typical Resources

- *Terminal I/O.*
- *Filesystem.* (Data stored on disk).
- *Main memory.* (A.k.a. RAM or core).
- *Sensors and actuators.* (In a RTS, for example.)
- *Network services.* (To connect to other computers.)
- *Threads.*
- *CPU time.*

OS description here is for a typical multithreaded of Unix . . .
. . . other modern operating systems work in a similar fashion.

Resources Managed on Behalf Of *Tasks*.

Task requests the resource.

OS determines if task is allowed the resource.

OS determines if task can be given the resource.

E.g., if there is enough available.

If both are true, OS grants the task's request.

To service such requests, the OS must keep track of how much of each resource is available and under what conditions requests can be granted.

Resource Allocation and Usage Example

Consider the following code fragment.

```
  /* Allocate 1000 bytes of storage. */
1: basePointer = malloc( 1000 ) ;
```

The code fragment above is part of some program which is compiled and run on some system.

The program, from when it starts running until it finishes running, is referred to as a *task*.

In line 1 the task requests address space (a resource).

The task asks the OS for 1000 bytes of main-memory address space using the C `malloc` library function.

The OS checks if the task is allowed 1000 more bytes of address space.

The OS also checks if 1000 bytes of address space is available.

If both checks are positive, the space is allocated. (If not, `malloc` returns a null pointer.)


```
  /* Write a 3 into element x. */
2: basePointer[x] = 3;

  /* Open a file for output. */
3: mf = fopen( "myFile.data" , "w" );
```

In line 2 task writes to main memory.

The task will attempt to write to memory address given by `basePointer+x`.

The write makes use of the address-space resource. The system must verify that this resource has been allocated.

If the task has write permission to this address, then the 3 is written.

Otherwise, the OS will terminate execution of the program before the 3 is written.

In line 3 task opens a file for writing, using the filesystem resource.

The OS checks if the task has write permission on the file.

The OS checks if opening the file for writing could be accomplished within the task's resource limits.

If both checks are positive, the OS performs all the actions necessary to open the file.

Description of Major Types of Resources

Terminal I/O.

Provided by the following hardware:

A communication port to which a terminal is connected.

A video display and keyboard connected to the computer.

The OS might have to reserve these devices for the task.

The OS might refuse access to a terminal if some other task had reserved it.

The OS, in most cases, would actually read or write data to these devices.

Sensors and actuators. (In RTS.)

These devices are connected to the computer through an *interface*.

The OS may reserve access to these devices for use by one task.

The OS may also perform the actual reading and writing of the devices.

Filesystem (data stored on disk in an organized form).

Actual hardware usually disk drives.

Disks “know” about tracks and cylinders.

OS organizes (as a librarian) disk into directories and files.

Files are given names, owners, access permissions, etc.

When a task issues a command to read a file the OS:

- finds the disk the file is located on,
- checks access permissions,
- finds the track and cylinder of the needed part of the file,
- sends commands to the disk to retrieve data,
- . . . when the data is available. . .
- reads the data sent by the disk drive, formats it for the task.

This organization makes programmers' lives simpler . . .
. . . and keeps the data relatively secure.

(OSs which do not control access to the filesystem or devices are vulnerable to viruses and worms.)

Main memory (A.k.a. RAM or core).

Whenever the CPU issues an instruction fetch, or does a load or store, a memory address is issued.

The memory address refers to a part of an address space, which is a resource allocated to the task.

Memory is managed by the OS and by special memory-management hardware.

CPU time.

The CPU time resource is time that a task runs on the CPU.

A system can have many tasks, the OS must divide the time between them.

The *scheduler* determines what fraction of CPU time each task gets.

Good CPU time management yields good performance:

- The computer will appear to react quickly.
- Little time to complete a set of jobs.

For a RTS, CPU time management is critical for correct performance.

Network services.

A computer connects to the rest of cyberspace through a *network interface*.

For outgoing data, the OS would format the data into a form suitable for the network interface.

The data would be transfered to the network interface, and then enter the network.

Upon arrival of incoming data, the network interface will alert the computer.

The OS will read the data from the interface and take the appropriate action.

OS Resource-Management Functions

To manage resources, the OS must provide the following:

• *System calls* by which tasks make resource requests.

`Malloc` and `printf` make use of system calls ...
... but are not themselves system calls.

• *Dæmon tasks* and other code needed to manage resources and used for other functions.

Includes tasks that prompt users to log in and dæmons which manage printers and other computer resources.

• *Shells*, programs which provide an interface between the OS and the user.

• *Utility* programs, so that the system administrator and users can query and change the information the OS uses to manage resources.

This includes programs that display a directory listing, increase a user's disk quota, and alter the priorities used by the scheduler.

System Calls

A user program uses *system calls* to make requests of the OS.

A system call, in many operating systems, works something like a subroutine call to the OS.

It is used by library writers and sophisticated user programmers.

(Less-adventurous programmers will call library functions, the library function makes the system call.)

System calls provide a "clean" interface to the operating system:

They have a logical syntax (in a well-designed system).

The syntax of the call is not likely to change even if the implementation of the resource changes. (For example, in newer versions of the computer.)

Typical uses for system calls:

Write a character to a terminal.

Allocate address space.

Spawn a new task.

File I/O.

End execution of the task.

Dæmon Tasks

The purpose of a dæmon task is to manage some hardware.

A dæmon is a task which waits for certain events to occur in the hardware (or elsewhere).

When the events occur, the dæmon will be notified and will take the appropriate action.

A well-known example is the *print dæmon*.

The print dæmon passes data to a printer.

Since printers have finite storage and usually print more slowly than they can accept data, the dæmon cannot transfer all data to be printed at one time.

Instead, the dæmon transfers data in small blocks.

After sending a block to the printer, the dæmon will relinquish control.

When the printer is ready for more data, it will *interrupt* the system, giving control back to the dæmon.

If there is more data, the dæmon will transfer another block.

Systems have many dæmons lurking in the background.

In addition to printing, dæmons also manage network communication, and many other services.

Shells and Utilities

A *shell* is a program that an OS runs to communicate with the user.

The user first encounters the shell after logging in.

The shell prompts the user for input.

Correct input is a command to the shell.

The shell performs the command. Typical commands include:

• Running a task.

• Displaying a directory listing.

• Changing directories.

• Logging out.

After the command completes shell will again prompt for input ...
... unless the command killed the shell's task.

Common Shells

`command.com`, the shell that comes with MS–DOS[1].

`sh`, called the Bourne shell. Used in Unix systems.

`csh`, called the C shell. Also used in Unix systems.

[1] (In the PC world, "shell" is sometimes used only for shells with a menu-driven or graphical interface. The broader definition will be used here.)

Utilities

These programs allow users and operators to change various settings in the operating system and to perform other actions.

Utilities include commands to change file-protection status, create directories, and print files.

The line between OS utility and a regular program is fuzzy.

For example, is a text editor that comes with an OS an OS utility or a bundled program?

Such questions can be ignored for this course.

Tasks, Programs, Processes, and Executables

A *program* is a collection of statements, instructions, etc. intended to be run as a unit.

"Program" can refer to several things . . .
. . . it's important to understand the distinctions:

- A program is in the form of <u>source code</u> . . .
  . . . when in a human-readable language.

  Programmers write source code.

  E.g., source code might be in file `hello.c`[2]

- A program is in the form of <u>object code</u> . . .
  . . . when it consists of *machine instructions*.

  Object code[3] is not a complete program.

- A program is in <u>executable</u> form when . . .
  . . . it contains all machine language instructions needed to run[4].

  E.g., Unix names executables `a.out` by default[5].

- A <u>task</u> (also called a process) is an executable . . .
  . . . which the OS has *loaded* . . .
  . . . and is, has, or will be running.

  Tasks are assigned *process IDs*.

[2] The last few letters of file names of files containing source code usually indicate the type of source code. For example, the `c` in `hello.c`.

[3] The last two letters of file names of files containing object code are usually `.o`. For example, `hello.o`.

[4] Executables are assembled by a linker from object code, libraries, start and stop code, etc.

[5] Most programmers use the program name, rather than `a.out`, for the executable name, for example, `hello`.

Example: consider "Hello, world!" used to introduce C.

Source Code (`hello.c`):

```
#include <stdio.h>

int main(int argv,char **argc)
{
 printf("Hello, world!\n");
 return 0;
}
```

With the following command . . .

```
[omega] % gcc hello.c -c
```

. . . file `hello.o` is generated[6] which contains the object code.

The command . . .

```
[omega] % gcc -o hello hello.o
```

. . . links[7] the object code with libraries and other code . . .
. . . to form the executable, `hello`.

[6] `gcc` is the name of a compiler driver program. The compiler driver calls the preprocessor and then the compiler itself.

[7] Here the compiler driver calls the linker.

The command below creates a task using the executable, `hello`.

```
[omega] % hello

Hello, world!
```

Source, object, and executable files will remain in the filesystem . . .
. . . until deleted.

In contrast, task only existed for a few milliseconds . . .
. . . starting when `hello` was typed at the prompt . . .
. . . and ending moments later.

Another task is created . . .
. . . whenever `hello` is typed.

(It's a distinct task though it does exactly the same thing.)

Range of Tasks

The following are run as tasks:

- Programs written for homework assignments.
- Application programs, such as accounting and payroll software.
- "Productivity" software, such as word processors and spreadsheets.
- Software development tools, such as compilers and linkers.
- Dæmon programs, such as the print dæmon.
- Shells.
- *External* shell commands.

The following are not run as tasks on conventional OSs:

(Instead they run something like a subroutine within another task.)

- System calls.
- *Interrupt handlers.* (These will be covered later.)

A Task's Resource Needs

A task always needs the following resources:

- Address space.
  Executable and data reside in address space.
- A thread.
  The part of the task that actually runs.

  If a task were a business, then a thread would be an employee.
- CPU time.
  (To run the executable.)

Most tasks also need the following resources:

- A terminal to communicate with the user.
- Access to a filesystem to read and write data.

Tasks and Threads

A *thread* is the part of the task that executes instructions.

Each task has one or more threads.

A *context* . . .
. . . is the info. that must be saved when a thread is stopped . . .
. . . and restored when the thread is re-started.

A *context switch* . . .
. . . is the process of removing one thread's context from the CPU . . .
. . . and replacing it with another.

Context information usually includes . . .
. . . the contents of CPU registers . . .
. . . and any other registers used to control the processor and memory.

Context information does <u>not</u> include . . .
. . . memory contents[8], disk contents, etc.

[8] (The memory-management hardware can hold and distinguish the address spaces for multiple tasks, therefore there is no need to remove a task's address space from memory when it temporarily stops running. A conventional CPU can only hold one set of register values, so these have to be saved.)

Threads

In *multithreaded* OSs a single task can have one or more contexts.

In a conventional OS each task has exactly one context.

In multithreaded OSs contexts are referred to as *threads of execution* or simply *threads*.

All threads in a task share the same address space and are usually sharing large amounts of data.

Context switching between two threads in the same task may take much less time than context switching between two threads in different tasks.

After a context switch in a multithreaded system a new thread will be running, and perhaps a new task.

In multithreaded systems, the OS allocates threads to tasks.

In other OSs each task allocated its one-and-only thread.

Context Switching

A single CPU can run only one thread at a time.

The OS switches execution from one thread to another, giving the appearance that many are simultaneously running.

The following occurs during a context switch:

- The context is saved in a data structure that the OS has reserved for the thread.

  These include the general-purpose registers, the stack pointer, and especially the program counter.

- After saving one context, the OS will load another context.

- If the new context is in a different thread the memory management hardware must be switched from one address space to another.

  This is done by writing the memory-management hardware's process-id register.

- The last step is to load the program counter, that is, jump to an instruction in the new thread.

  When the context switch is complete, a different task or thread will be running.

Because of the bookkeeping involved (not just saving the registers) context switching takes a relatively long amount of time.

The CPU-Time Resource

A task is getting CPU time, *running*, whenever the CPU has one of its contexts.

The task will continue to run until:

it requests a resource from the OS,

it attempts to do something it is not allowed to do,

some other event needs to be attended to,

it has exhausted its present CPU time resource allocation.

At this point the OS will take control, perhaps restarting the task at a later time.

The Life of a Task

Here are the steps in the life of a typical task.

- "Pre-natal" step. An executable program is waiting in a file.

- "Birth." A currently running task starts the new task.

  In Unix, this is done in two steps:

  First the currently running task makes a `fork` system call. The OS will make a duplicate of the running task. The two resulting tasks are identical, except for their process IDs.

  New task makes a `execve` system call, loading and starting the new executable.

- The new task will run for some time.

- Suppose the task needs to wait for user input.

  It will make a system call, requesting input.

  The user may not provide input for some time, so the OS will do a context switch to another task.

  After the user enters some input, the OS will do a context switch back to the task.

- The task computes some more. Suppose it exceeds its CPU time allocation.

  The CPU will *preempt* the task, context switching to a new task.

  Later the task will be resumed.

The Life of a Task, Continued

- The task is resumed, continuing its work.

  ⋮

- Finally, the task finishes: it makes a final system call, `exit`.

  The OS cleans up any "loose ends" the task might have left, for example closing files.

  The OS will then release the task's resources for use by other tasks.

The Kernel

The *kernel*, the core of an OS, is executable code kept in a special area of memory which is accessible only through special CPU instructions.

When a task makes a system call it is executing the special CPU instruction.

The special instruction is usually called a *trap* or a *software interrupt*.

As a result of making a system call, the CPU switches from *user mode* to *privileged mode*.

A second result of a system call is a jump made into the special area of memory, where the kernel starts executing.

In many systems virtual address space is divided into two halves:
one reserved for tasks
the other for the kernel.

System half of every task's address space . . .
. . . mapped to the same physical addresses . . .
. . . saving memory since only one copy needed.

The Kernel vs. Tasks

An OS is able to enforce its control over resources because of special features of the hardware.

For example, the memory-management hardware will halt any task which attempts to access unallocated memory.

So why can't the task tell the memory hardware that it *does* have access to that memory?

If the OS can do it, couldn't the task? Isn't the OS a program, just like the task?

No, it's not.

CPU Modes

The OS is a program made up of instructions, to be sure.

But some of those instructions are *privileged*.

Modern CPUs have a special mode, called *privileged* (a.k.a. system or supervisor) mode.

Privileged instructions can only be executed in privileged mode.

Privileged instructions are used to set the memory-management hardware and to change other sensitive parts of the system.

Privileged mode is turned off by setting a bit in the CPU's *processor status word* (PSW).

While the kernel is running, privileged mode is on.

Before kernel returns to task, privileged mode turned off.

As one might expect, a task cannot change the PSW, so a task cannot change the CPU to privileged mode and loot the system resources.

How the Kernel is Entered

Reasons the kernel gets entered, and what the kernel does:

• A task makes a system call.

The kernel performs the requested action.

• A task attempts to execute an illegal instruction or attempts an illegal memory access.

The kernel may kill the task, allow the task to continue, or call a special *handler* routine provided by the task.

The type of action depends upon what the task attempted.

• An external device signaled the computer for attention, by requesting an *interrupt*.

The kernel will call a special handler routine to attend to the external device.

This will be covered in great detail, later.

• A *timer* (like an alarm clock) has expired, interrupting the CPU.

The action taken by the kernel depends upon why the timer was set.

The timer might have been set:

because an external device would need attention (but could not generate an interrupt),

to update time-related data structures, such as the system clock,

to preempt the running task because its CPU-time allocation is used up.

Example of CPU Activity

13-29

Consider a system in which two programs are running.

One is searching a file for the letter X:

```
/* Program 1 (Will run as task 5371.) */
do { c = fgetc( fileStream ); } while ( c != 'X' );
```

The other is computing π.

```
/* Program 2 (Will run as task 8462.) */
double sum = 0, i = 1, pi;
while( i < 10000000000 ){ sum+=1/i; i+=2; sum-=1/i; i+=2; }
pi = sum * 4.0;
```

The CPU might be doing the following:

User Mode, Task 5371, Prog. 1: Execute code user wrote. (`do`).

User Mode, Task 5371, The `fgetc` library function.

Priv. Mode, Kernel, The `read` system call.

Priv. Mode, Kernel, Context switch to Task 8462.

User Mode, Task 8462, The while loop, written by the user.

Priv. Mode, Kernel, Disk interrupted with requested data. ...
... OS writes data in buffer. Interrupted task resumed.

User Mode, Task 8462, The while loop, written by the user.

Priv. Mode, Kernel, (Entered because of timer.)

Priv. Mode, Kernel, Scheduler, choose new task. Context switch to Task 5371.

User Mode, Task 5371, The `fgetc` library function.

User Mode, Task 5371, While condition comparison.