OS Structure,
Processes & Process Management

Recap

- OS functions
  - Coordinator
    - Protection
    - Communication
    - Resource management
  - Service provider
    - File system, device handler, …
- Questions:
  - How can the OS perform these functions?
  - How is an OS invoked?
  - What is the structure of the OS?

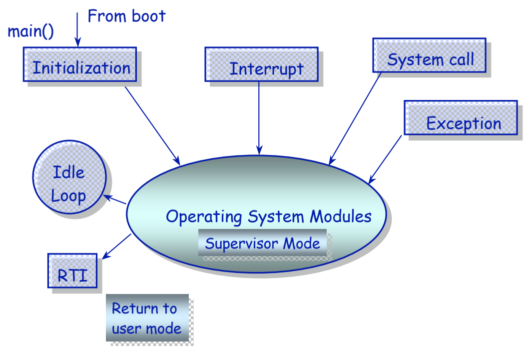
An Operating System in Action

- CPU loads boot program from ROM (e.g. BIOS in PC's)
- Boot program:
  - Examines/checks machine configuration (number of CPU's, how much memory, number & type of hardware devices, etc.)
  - Builds a configuration structure describing the hardware
  - Loads the operating system, and gives it the configuration structure
- Operating system initialization:
  - Initialize kernel data structures
  - Initialize the state of all hardware devices
  - Creates a number of processes to start operation (e.g. getty in UNIX, the Windowing system in NT, e.g.)

O.S. in Action (Cont'd)

- After basic processes have started, the OS runs user programs, if available, otherwise enters the *idle loop*
- In the idle loop:
  - OS executes an infinite loop (UNIX)
  - OS performs some system management & profiling
  - OS halts the processor and enter in low-power mode (notebooks)
- OS wakes up on:
  - Interrupts from hardware devices
  - Exceptions from user programs
  - System calls from user programs
- Two modes of execution
  - User mode: Restricted execution mode (applications)
  - Supervisor mode: Unrestricted access to everything (OS)

Control Flow in an OS



On Interrupts

- Hardware calls the operating system at a pre-specified location
- Operating system saves state of the user program
- Operating system identifies the device and cause of interrupt
- Responds to the interrupt
- Operating system restores state of the user program (if applicable) or some other user program
- Execute an RTI instruction to return to the user program
- User program continues exactly at the same point it was interrupted.

Key Fact: None of this is visible to the user program

On Exceptions

- Hardware calls the operating system at a pre-specified location
- Operating system identifies the cause of the exception (e.g. divide by 0)
- If user program has exception handling specified, then OS adjust the user program state so that it calls its handler
- Execute an RTI instruction to return to the user program
- If user program did not have a specified handler, then OS kills it and runs some other user program, as available

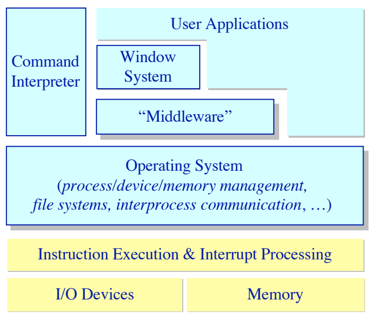
Key Fact: Effects of exceptions are visible to user programs and cause abnormal execution flow

On System Calls

- User program executes a trap instruction (system call)
- Hardware calls the operating system at a pre-specified location
- Operating system identifies the required service and parameters (e.g. open(filename, O_RDONLY))
- Operating system executes the required service
- Operating system sets a register to contain the result of call
- Execute an RTI instruction to return to the user program
- User program receives the result and continues

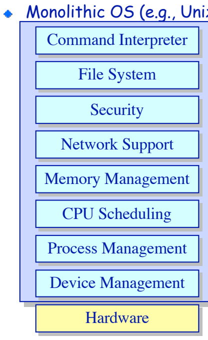
Key Fact: To the user program, it appears as a function call executed under program control

Operating System Today
High-level software architecture

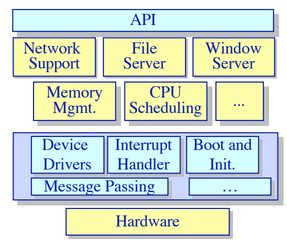


Operating System Structures

- Monolithic OS (e.g., Unix)



- Micro-kernel OS (e.g., Mach, Exokernel, …)



Summary

- An OS is just a program:
  - It has a main() function, which gets called only once (during boot)
  - Like any program, it consumes resources (such as memory), can do silly things (like generating an exception), etc.
- But it is a very strange program:
  - It is "entered" from different locations in response to external events
  - It does not have a single thread of control, it can be invoked simultaneously by two different events (e.g. system call & an interrupt)
  - It is not supposed to terminate
  - It can execute any instruction in the machine

Processes and Process Management
What is a Program? How to run a Program?

- A program consists of code and data
- On running a program, the loader:
  - reads and interprets the executable file
  - sets up the process's memory to contain the code & data from executable
  - pushes "argc", "argv" on the stack
  - sets the CPU registers properly & calls "__start()"
- Program starts running at _start()

```
_start(args) {
    ret = main(args);
    exit(ret)
}
```

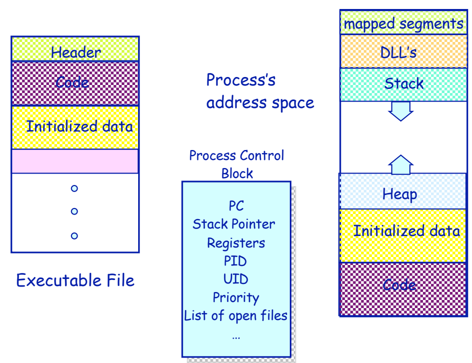
we say "process" is now running, and no longer think of "program"

- When main() returns, OS calls "exit()" which destroys the process and returns all resources

So, What is a Process?

- A process is an abstraction that supports running programs
- A process is the basic unit of execution in an operating system
- Different processes may run several instances of the same program
- At a minimum, process execution requires following resources:
  - Memory to contain the program code and data
  - A set of CPU registers to support execution

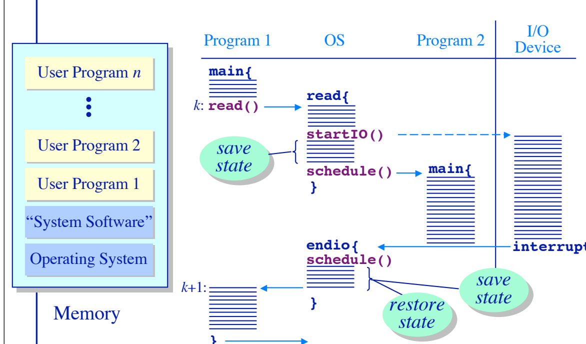
Anatomy of a Process



Process Life Cycle

- Processes are always either *executing*, *waiting to execute* or *waiting for an event* to occur

Process Contexts
Example: Multiprogramming



Process Manipulation

- Basic process manipulation: creation, program loading, exiting, …
- Example: Unix Operating system
  - Creation and deletion: fork(), exec(), wait(), exit()
  - Process signaling: kill()
  - Process control: ptrace(), nice(), sleep()

Process Manipulation in Unix

- The system creates the first process (*sysproc* in Unix)
- The first process creates other processes such that:
  - the creator is called the parent process
  - the created is called the child process
  - the parent/child relationships can be expressed by a process tree
- In Unix, the second process is called *init*
  - it creates all the gettys (login processes) and daemons
  - it should never die
  - it controls the system configuration (num of processes, priorities…)
- Unix system interface includes a call to create processes
  - *fork*()

Unix's fork()

- Creates a child process such that it inherits:
  - identical copy of all parent's variables & memory
  - identical copy of all parent's CPU registers (except one)
- Both parent and child execute at the same point after fork() returns:
  - for the child, fork() returns 0
  - for the parent, fork() returns the process identifier of the child
- Simple implementation of fork():
  - allocate memory for the child process
  - copy parent's memory and CPU registers to child's
  - *Expensive* !!

Can one reduce this overhead without changing semantics?

Unix's fork(): Example Usage

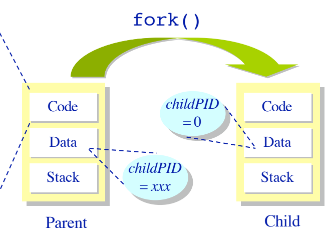
- The execution context for the child process is a *copy* of the parent's context at the time of the call

```
main {
  int childPID;
  S1;

  childPID = fork();

  if(childPID == 0)
    <code for child process>
  else {
     <code for parent process>
     wait();
   }

  S2;
 }
```



Program Loading: exec()

- The exec() call allows a process to "load" a different program and start execution at _start
- It allows a process to specify the number of arguments (argc) and the string argument array (argv)
- If the call is successful
  - it is the same process …
  - but it runs a different program !!
- Two implementation options:
  - overwrite current memory segments with the new values
  - allocate new memory segments, load them with the new values, and deallocate old segments

General Purpose Process Creation

In the parent process:

```
main()
…
int pid = fork();                // create a child
if(pid == 0) {                   // child continues here
    exec("program", argc, argv0, argv1, …);
}
else {                           // parent continues here
…
}
```


Properties of the fork/exec sequence

- In 99% of the time, we call exec() after calling fork()
  - the memory copying during fork() operation is useless
  - the child process will likely close the open files & connections
  - overhead is therefore high
  - might as well combine them in one call (OS/2)
- vfork()
  - a system call that creates a process "without" creating an identical memory image
  - sometimes called lightweight fork()
  - child process is understood to call exec() almost immediately

Orderly Termination: exit()

- After the program finishes execution, it calls *exit*()
- This system call:
  - takes the "result" of the program as an argument
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - If so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - If not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies

The wait() System Call

- A child program returns a value to the parent, so the parent must arrange to receive that value
- The wait() system call serves this purpose
  - it puts the parent to sleep waiting for a child's result
  - when a child calls exit(), the OS unblocks the parent and returns the value passed by exit() as a result of the wait call (along with the pid of the child)
  - if there are no children alive, wait() returns immediately
  - also, if there are zombies waiting for their parents, wait() returns one of the values immediately (and deallocates the zombie)

Process Control

OS must include calls to enable special control of a process:

- Priority manipulation:
  - nice(), which specifies base process priority (initial priority)
  - In UNIX, process priority decays as the process consumes CPU
- Debugging support:
  - ptrace(), allows a process to be put under control of another process
  - The other process can set breakpoints, examine registers, etc.
- Alarms and time:
  - Sleep puts a process on a timer queue waiting for some number of seconds, supporting an alarm functionality

Tying it All Together: The Unix Shell

```
while(! EOF) {
read input
handle regular expressions
int pid = fork();                  // create a child
if(pid == 0) {                     // child continues here
    exec("program", argc, argv0, argv1, …);
}
else {                             // parent continues here
…
}
```

- Translates <CTRL-C> to the kill() system call with SIGKILL
- Translates <CTRL-Z> to the kill() system call with SIGSTOP
- Allows input-output redirections, pipes, and a lot of other stuff that we will see later