CS571: Operating Systems

CLASS 2: 29 JAN 2003
16:30 – 19:10

Class 2 Overview

- why an OS?
- tasks & users
- process
- processes & UNIX
What the operating system operates:

- basic hardware configuration:
  - memory
  - cache
  - CPU
  - input device
  - output device

- basic tools:
  - instructions, registers (e.g., PC, PSW)
  - interrupts
  - traps
  - DMA
  - polling

Why Have an OS?

- surely you can operate the hardware in software...why have an OS do what you could do yourself?
Why Have an OS?

- surely you can operate the hardware in software...why have an OS do what you could do yourself?
- you wouldn’t want to do this yourself:
  - way too much software overhead
  - non-portability:
    - brittle to machine configurations
    - can’t port to other platforms

What the OS gives us:

- a set of resources created, maintained and controlled by the OS
- presented as a set of services for use:
  1. program execution
  2. I/O operation
  3. File System
  4. Communication
  5. error detection and recovery
  6. resource allocation
  7. accounting
  8. protection

- invoked via system call (supervisor call)
  - performance of the service requires privilege level higher than that given to user programs
Class 2 Overview

- why an OS?
- tasks & users
- process
- processes & UNIX

A Simple OS: A Day in the Life

- load kernel into memory
  - interrupt and trap handlers
  - initialize vector table(s)
  - other code supporting services to be offered
- begin CLI program (loop):
  repeat {
    read keyboard command
    load invoked program into user memory space
    jump from CLI to user program
    at end of user program, jump back to CLI
  }
  until (the cows come home)
Simple OS:

- runs exactly 1 user program at a time
- runs the program from start to end before user can start another program
Simple OS:

- runs exactly 1 user program at a time
- runs the program from start to end before user can start another program

⇒ single-tasking OS

Single Task

- simple to manage
  - only “1 thing” going on
- easy to do resource handling
  - no other task can ask for resources

- how efficient in use of computer resources?
Single Task

- simple to manage
  - only “1 thing” going on
- easy to do resource handling
  - no other task can ask for resources

- how efficient in use of computer resources?
  - not very

Multi Task

- multi-tasking OS:
  - run one task while another is not able to run
  - create illusion of running ≥ 1 task at same time
Multi Task

- multi-tasking OS:
  - run one task while another is not able to run
  - create illusion of running ≥ 1 task at same time
  - memory management: placement & protection for multiple tasks all resident at same time

- scheduling: when task can have resources, when it can’t (especially CPU)
  - can use hardware to help (e.g., timers)
Multi Task

- **multi-tasking OS:**
  - run one task while another is not able to run
  - create illusion of running ≥ 1 task at same time
  - memory management: placement & protection for multiple tasks all resident at same time
  - scheduling: when task can have resources, when it can’t (especially CPU)
    - can use hardware to help (e.g., timers)
  - communication: tasks may interact cooperatively to achieve an objective

Users?

- how many users can a system have?
  - single user: self-explanatory
 Users?

- how many users can a system have?
  - **single user**: self-explanatory
  - **multi user**: support for different users
- what do you need in order to support multi-users?

- what do you need in order to support multi-users?
  - way to uniquely identify each user
Users?

- what do you need in order to support multi-users?
  - way to uniquely identify each user
  - provide security to protect users from each other
    - e.g., file “ownership”
  - provide ways to let users interact with each other
Multi-User and Multi-Tasking

- are these synonymous?

- NO: can have:
  - single-user multi-tasking
  - multi-user single-tasking
Class 2 Overview

- why an OS?
- tasks & users
- process
- processes & UNIX

The Consumer

- OS provides services for the running of programs, tasks
- we introduce a model for such a ‘consumer’ of these services: the process
The Process

- standard model of user of OS services/resources

- is a program, hence uses:
  - cpu
  - memory
  - possibly, other resources
    - e.g., disk, graphics display, network, keyboard
a process has, at all points in its existence, a status
simplest view:
- running
- not running
Life of a Process

- this is a model that captures states and their transitions:
  - running
  - not running
  - pause
  - dispatch
  - enter
  - exit

- implemented something like this:
  - CPU
  - dispatch
  - queue
  - pause

a process runs for a certain time (timeslice) then, if not finished, it is paused to let another process have a turn

- but does a process always use all of its timeslice?
Life of a Process

- but does a process always use all of its timeslice?
  - NO: may block early, e.g., read from keyboard
  - NO: may fail or simply terminate and not use full timeslice
- if blocked, we should ‘boot it out’ early: since it can’t run anymore

Life of a Process

- improved model
  - distinguishes between running, ready, blocked
Life of a Process

3 new states:

1. **new**:
   - OS has done “set up” work to ready a newly arrived process for running but process not yet queued as runnable
   - OS may not admit new processes upon their arrival: may limit number of running processes

2. **exit**:
   - process no longer eligible for running
   - OS reclaims resources, does “clean-up”
   - OS does any accounting updates

3. **blocked**:
   - processes ‘queued’ here to wait for the event on which they are blocked
   - leave this state when event occurs that makes them runnable again
Life of a Process

3. **blocked**: CPU dispatch release

- multiple priority queues possible:
  - a process may change priority during its life
  - processes may belong to different priority classes

- a blocked process, moved to the ‘Q blocked’ is no longer consuming CPU resources
- is it still consuming any other resource?
Life of a Process

- a blocked process, moved to the ‘Q blocked’ is no longer consuming CPU resources
- is it still consuming any other resource?
- YES: memory (at least)
  - e.g., an interactive program waiting for input; user has gone for lunch
- what happens if all processes become blocked waiting?

Life of a Process

- what happens if all processes become blocked waiting? ⇒ CPU becomes idle
- how overcome idleness?
Life of a Process

- what happens if all processes become blocked waiting? $\Rightarrow$ CPU becomes idle
- how overcome idleness?
  - get more processes to run
- but: what if more processes won’t fit into memory?

- but: what if more processes won’t fit into memory?
  - need to move process(es) currently into memory someplace else so as to liberate space
  - can move process' memory image to disk
Life of a Process

- but: what if more processes won’t fit into memory?
  - need to move process(es) currently into memory someplace else so as to liberate space
  - can move process’ memory image to disk
- **swapping:** the moving of an entire process image between disk and memory
  - swap out: write process image out to disk
  - swap in: read process image into memory from disk
- need to adjust our process state model to allow for swapping:

Life of a Process

```
newly created  admit  ready  dispatch  running  release  exit
               |      |       |      |        |      |
               |      |       | time-out |      |
               |      |       |    event wait |
               |      |       |    event occurs |
               |      |       |      | suspend |
               |      |       |      | swap/susp |
  activate   |      |       |      |
  event wait |
  event occurs |
  suspend |
  swap/susp |
```
Life of a Process

- we can swap a blocked process to make room to admit new, runnable processes
- would we ever swap a runnable process?

- YES: e.g., to make room for a higher priority process that needs the memory space

being blocked and being swapped are independent

- so need a further refinement to our process state model:
Life of a Process

Process Attributes

1. identification:
   • PID process identifier (usually an int)
   • PPID parent process PID
   • UID identifier of user to whom process belongs
Process Attributes

2. processor state info:
   - user registers (GPRs,…)
   - control and status registers (PSW,…)
   - stack pointer(s)

Process Attributes

3. process control info:
   - process state (running, waiting)
   - priority (actual; maybe also min/max range)
   - scheduler & timer (time remaining in turn)
   - event id
   - IPC info (status, msg queues, …)
   - MMU info (segment ptrs, page tbl ptrs, …)
   - privilege level
   - resource ownership & use (files, devices, …)
Process Bureaucracy

- how does the OS keep track of all these details for each process?
- in a structure called a process control block (PCB)
  - new PCB allocated to each arriving process
Process Bureaucracy

- kernel supports different process states with something like:

  - running
  - ready
  - blocked

- what advantage does this offer for changing the state of a process?

Class 2 Overview

- why an OS?
- tasks & users
- process
- processes & UNIX
Processes in UNIX

- each process has a:
  - unique PID (integer)
  - unique UID (integer)
- each process is the ‘offspring’ (child) of a single parent process
- if process has PID j, then its child has PID k, k ≠ j, and j is k’s parent process id, PPID
  - each process (except one) has a PPID

Processes in a UNIX System

- partial list: (use `ps` command)

```
S  UID  PID  PPID  PRI  NI   ADDR     SZ   WCHAN  TTY     TIME  CMD
T  0    0    0    0    0   1428630 0    ?        0:06  sched
S  0    1    0    0    40  20 7014f748 243  ?        1:09  init
S  0    2    0    0    0  0    ?        0:05  pageout
S  0    3    0    1  0    0    1458268 0    419:45  fsflush
S  0  365    1    0    40  20 779f22f0 859  ?        0:01  xfs
S  0  365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
S  0 365    1    0    40  20 7014f060 0    144d958  ?        0:05  pageout
S  0 365    1    0    40  20 7014e978 0    1458268  ?        0:05  pageout
S  0 365    1    0    40  20 70124020 339  ?        0:09  sshd
```
Per Process info in UNIX

- UNIX kernel keeps this kind of information on each process under management:
  1. process status
  2. memory pointers to user area (text, data, stack)
  3. process size
  4. user ids (uid, euid)
  5. process ids (pid, ppid)
  6. event descriptor (events being waited on; valid only when sleeping)
  7. timers (incl. user timers for alarms)
  8. P-link (if runnable, points to next runnable in Q)
  9. mem stat (in real mem; is swapable?)

- additional info is available when process is running, including:
  - signal handler array (for each signal, action is one of ignore, default, or user defined)
  - I/O parms: addresses of user data buffers, etc.
  - user file descriptor table: info on files user has opened (descriptors 0, 1, 2 always open)
  - limit fields: resource limits applicable to this proc
  - permission modes: umask for files created by this process
Memory for UNIX Processes

- every process is allocated memory in which to operate, structured as:

  - **r/w stack**: e.g., automatic variables
    - grows/shrinks
  - **r/w segment**: data accessible by program
  - **ro segment**: executable instruction code

Example (refresher):

```c
float foo(float z)
{
    int i, j;
    float x;
    return(-z);
}
```

- pushes 'y' onto stack,
- pushes return address onto stack,
- puts address of foo into PC
- tells 'foo' that parm z is in stack, of size sizeof(float)
- allocate automatic variables for this context, take space from stack (here, 2 ints and 1 float)
- pop x, j, i from stack,
- insert result into stack,
- pop return address
Process Creation in UNIX

- all (but one) processes come from a parent using: `fork()`:
  - makes an identical copy of the parent process
  - child process gets new PID
  - execution continues in both processes
  - return value:
    - is 0 in child process
    - is PID of child in parent process

- parent may choose:
  - to wait until child finishes before continuing
  - not to wait
- child usually wants to be a different program than the parent, so it replaces the process image it currently contains with a new process image using one of the `exec()` family:
  - replaces current process image with that of a new process image
Process Creation in UNIX

- e.g.,
  ```c
  i = fork();
  if (i == 0) { /* I am child process */
    k = execl(some_other_thing, ...);
  }
  else { /* I am parent */
    k = wait(&j); /* wait for child to finish */
  }
  /* continue here in parent after child exits with status in j */
  ```
Getting a Process’ Attention

- how do you arrange for a program to wait only for a certain time before taking action?

  • need a timer (hardware)
  • timer will generate interrupts at regular intervals
  • OS kernel handles those interrupts
Getting a Process’ Attention

- how do you arrange for a program to wait only for a certain time before taking action?
- how do you catch an illegal memory access and deal with it gracefully before exiting?

• illegal memory access causes hardware trap
• OS kernel handles that trap
Getting a Process’ Attention

- how do you arrange for a program to wait only for a certain time before taking action?
- how do you catch an illegal memory access and deal with it gracefully before exiting?
⇒ need way for OS handling of events to be communicated to our process

UNIX signals

- UNIX provides **signal** mechanism to achieve this
- a signal is ‘delivered’ to a process and:
  - a default handler function is performed, or
  - a user supplied handler function is performed, or
  - the signal is ignored
- note: the signal is a **consequence** of the actual interrupt or trap, not the event itself
Handling signals

- user can:
  - assign a handler function to handle a specific signal
  - designate the action for a specific signal to be 'ignore' (SIG_IGN)
  - reset the action for a specific signal to 'default' (SIG_DFL)

- e.g., use:
  ```c
  sigaction(int signum,
             const struct sigaction *act,
             struct sigaction *oldact)
  ```

Handling signals

- other uses:
  ```c
  sigaction(int signum,
            (const struct sigaction *)(0),
            (struct sigaction *)(0))
  ```
  retrieves the current signal action for the designated signal (in `oldact`)

- other functions can be used to do this, e.g., `signal()`
Limitations

- can user set action to ignore for all signals?

SIGKILL

- can user set action to ignore for all signals?
- NO!! signal 9 cannot be “caught or ignored”
  ⇒ terminate signal
# Signals in Linux

## Some Linux Signals:

<table>
<thead>
<tr>
<th>Signal</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIGHUP</td>
<td>Hangup (POSIX)</td>
</tr>
<tr>
<td>SIGINT</td>
<td>Interrupt (ANSI)</td>
</tr>
<tr>
<td>SIGQUIT</td>
<td>Quit (POSIX)</td>
</tr>
<tr>
<td>SIGILL</td>
<td>Illegal instruction (ANSI)</td>
</tr>
<tr>
<td>SIGTRAP</td>
<td>Trace trap (POSIX)</td>
</tr>
<tr>
<td>SIGABRT</td>
<td>Abort (ANSI)</td>
</tr>
<tr>
<td>SIGIO</td>
<td>IOT trap (4.2 BSD)</td>
</tr>
<tr>
<td>SIGBUS</td>
<td>BUS error (4.2 BSD)</td>
</tr>
<tr>
<td>SIGFPE</td>
<td>Floating-point exception (ANSI)</td>
</tr>
<tr>
<td>SIGKILL</td>
<td>Kill, unblockable (POSIX)</td>
</tr>
<tr>
<td>SIGUSR1</td>
<td>User-defined signal 1 (POSIX)</td>
</tr>
<tr>
<td>SIGSEGV</td>
<td>Segmentation violation (ANSI)</td>
</tr>
<tr>
<td>SIGUSR2</td>
<td>User-defined signal 2 (POSIX)</td>
</tr>
<tr>
<td>SIGPIPE</td>
<td>Broken pipe (POSIX)</td>
</tr>
<tr>
<td>SIGALRM</td>
<td>Alarm clock (POSIX)</td>
</tr>
<tr>
<td>SIGTERM</td>
<td>Termination (ANSI)</td>
</tr>
<tr>
<td>SIGSTKF</td>
<td>Stack fault.</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>Same as SIGCHLD (System V).</td>
</tr>
<tr>
<td>SIGCHLD</td>
<td>Child status has changed (POSIX).</td>
</tr>
</tbody>
</table>

## Default Signal Handler Action

- **default handler action:** terminate process and:
  - dump core: 3,4,5,6,7,8,10,11,12,29
  - discard signal for: 16,19,20,23,28
  - stop process: 17,18,21,22
Signals and Processes

signals arrive at process go 'through' signal mask

1 2 3 4 5 6 7 8 \[\text{\textcolor{red}{\textbullet}}\] 10 11 12 13 ...

- sig1 handler
- sig5 handler
- sig11 handler
- user action
- ignore action
- default action

When Signals Come Knocking…

- when signal arrives at process:
  - current process state saved
  - get new signal mask (duration of handler)
  - invoke signal handler
  - if handler exits normally, process can resume where it left off
  - if want to resume elsewhere, user must arrange to set up return address parameters by hand
Sending Signals

- kernel can send signals to processes

- processes can send signals to
  - themselves
  - other processes
- e.g., `kill(getpid(), SIG_INTR)`
  - can also kill process group rooted at a given PID

- limits on other processes that can be signaled?
Sending Signals

- limits on other processes that can be signaled?
  - can only signal a process having same owner
  - only privileged user can signal any process
Assignment 1

- now available on the course website
- due: 12 Sep 03