Class 2 Overview

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

What the operating system operates:

- basic hardware configuration: CPU
  - 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?

- 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

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
  - \( \Rightarrow \) 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
- 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?

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?

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

Life of a Process

- a process has, at all points in its existence, a status
- simplest view:
  - running
  - not running
Life of a Process

- this is model captures states and their transitions:
  ```plaintext
  not ready running
  enter dispatch exit
  ```

- implemented something like this:
  ```plaintext
  running
  pause
  CPU
  ```

- 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
  ```plaintext
  not running
  ```

  • distinguishes between running, ready, blocked

Life of a Process

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

  • NO: may block early, e.g., read from keyboard

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:

- a blocked process, moved to the 'Q blocked' is no longer consuming CPU resources
- is it still consuming any other resource?

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

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

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

- 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

• but: what if more processes won’t fit into memory?
  • need to move process(es) currently into memory
    somewhere 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

• 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)

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

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.

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 (including user timers for alarms)
  8. P-link (if runnable, points to next runnable in Q)
  9. Memory status (in real memory; is swapable?)

Memory for UNIX Processes

- Every process is allocated memory in which to operate, structured as:
  - R/W stack: e.g., automatic variables grow/shrinks
  - R/W segment: data accessible by program
  - RO segment: executable instruction code

Processes in a UNIX System

- Partial list (use `ps` command):

```
   S   UID   PID  PPID  C PRI NI     ADDR      SZ    WCHAN  TTY     TIME   CMD
   T     0     0     0  0  0 SY  1428630      0          ?       0:06 sched
   S     0     1     0  0  40 20 7014f748    243 144f8a4        ?       1:08 init
   S     0     2     0  0  40 20 7014f060      0  144f8a4        ?       0:05 pageout
   S     0     3     0  1  0 SY 7014e978      0  1458268        ?       4 19:45 fsflush
   S 19455  11071  1  0  40 20 70124020    339 701db27a       ?       0:09 sshd
   S 19545  5533  5533  0  40 20 7a7ece88    133 7a7ecef4 pts/31   0:00 sh
   S 19545  5534  5533  0  40 20 7a872300    922 7ef603ca ??       0:00 dtterm
   S 19545  5536  5534  0  40 20 76c54728    383 78da0a56 pts/15   0:00 tcsh
   S 19545  6005     1  0  40 20 779f1520    238 779f158c       ?       0:00 Xsession
   S 19545  6060  6005  0  40 20 7a82c790    504 7c397352 pts/31   0:00 sdt_shel
   S 19545  6063  6060  0  40 20 7a893170    336 78d4b8a6 pts/31   0:00 tcsh
   S 19545  6121  6063  0  40 20 7a82e330    964 7fa7e04a pts/31   0:00 dtsessio
   S 19545  6122  6121  0  40 20 7a8587d0   1128 7c3d67b2 ?       2:39 dtwm
   S     0 11611   365  0  40 20 779f37a8    596 7d78cb7a       ?       0:05 sshd
   S 19599 26383 11611  0  40 20 779f30c0    351 7bb8ff06 pts/3    0:00 tcsh
   S 19599 11649 11611  0  40 20 76c42360    312 7f289546        ?       0:00 tcsh
   S 19599 26436 26383  0  40 20 7a7e6350   1109 78fa1d82 pts/3    0:10 pine
```

Per Process info in UNIX

- 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 parameters: 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 process.
  - Permission modes: umask for files created by this process.

Example (refresher):
```
x = foo(y);
float foo(float z)
{
  int i,j;
  float x;
  return(-z);
}
```
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

Process Creation in UNIX

- 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

```c
int 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?

```c
fork()
```

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?

 ⇒ 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:
  \[ \text{sigaction}(\text{int signum}, \\
  \text{const struct sigaction *act}, \\
  \text{struct sigaction *oldact}) \]

  - signal to be caught
  - structure defining new handler
  - structure to hold handler being replaced

  \[ \text{signal}(\text{id}) \]

other uses:

  \[ \text{sigaction}(\text{int signum}, \\
  \text{(const struct sigaction *)0}, \\
  \text{struct sigaction *oldact}) \]

retrieves the current signal action for the designated signal (in *oldact)

other functions can be used to do this, e.g.,

  \[ \text{signal}() \]
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:
  - SIGKILL 9 /* Kill, unblockable (POSIX). */
  - SIGSEGV 11 /* Segmentation violation (ANSI). */
  - SIGALRM 14 /* Alarm clock (POSIX). */
  - SIGCHLD 17 /* Child status has changed (POSIX). */

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

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?

- 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