Chapter 2: Operating Systems Overview

2.1 Introduction

Q.2.1 What is an Operating System?

-
-
-

What are the overall goals of an Operating System?

-
-
-

Q.2.2 What is the kernel?

What services does the kernel provide?

-
-
-
-
-
-
-

How the kernel provides some of these services is the topic of this course

Q.2.3 What language is used to write an OS?

Q.2.4 Definitions

- turnaround time:
- multiprogramming:
- time slice:
- context/process switch:
- parallel processing versus concurrent processing:

2.2 Evolution of Operating Systems

Early Systems (serial processing systems)

- *no operating system*
- large single-process machines
- input devices:
- output devices:

Batch Systems

- operating system called a monitor
- **Q.2.5** The main feature of a batch system is ...
- operator sorts submitted jobs by job type
  batch similar jobs together → less set-up time for monitor
- monitor does automatic job sequencing
- this early OS included protection of system:
  -
  -
  -
- **Q.2.6** What was turnaround time for a job submitted to a batch system?
- problem 1:
- problem 2:

Time Sharing systems

- **Q.2.7** The main features of a time share system are ...

- time sharing systems became common in the early 1970's

Multiprogrammed Batch Systems vs. Time Sharing systems

Modern operating systems

- Consider System Types:
  - Mainframe Computers
  - Personal Computers (one CPU)
  - Parallel Systems (multi-processor systems)
    - SMP (symmetric multiprocessors) = tightly coupled systems
    - MultiCore and ManyCore systems
    - clusters = loosely coupled systems
  - Real-time Systems
    - when there exists rigid time requirements on the operation
    - embedded systems often run real-time operating systems
  - Handheld Systems
    - PDAs, Pocket-PCs, cell phones, etc.
    - often have special-purpose embedded operating system

- **Q.2.8** Summary of System Types:
  -
  -
  -

- **Q.2.9** Consider Challenges for OS Developers
  - CPUs
  - Network
  - Storage

think about how complex the OS must be for today's systems

Q.2.10 How do we tame complexity?

Virtual Machines

- **Q.2.11** Definition:
-
- **Q.2.12** Benefits:
  -
  -
  -
- **Q.2.13** Cons:
  -
  -
- **Q.2.14** Possible implementations:
  1.
  2.

2.3 Topics of an Undergraduate OS Course

1. Processes

- **Q.2.15** Definition of a process ...

- process components:
  -
  -
  -

- Consider an example of how processes may be implemented
  - suppose block of contiguous memory allocated to each process
  - each process recorded in a process list, maintained by the OS
  - **Q.2.16** Definitions:
    * process index =
    * program counter =
    * base register =
    * limit register =

- Activities associated with processes:
  - process creation and deletion (fork and kill)
  - CPU scheduling of processes
  - process communication (pipes)
  - process synchronization

- Multithreading
  - a multithreaded process allows concurrent threads to execute in ONE process
  - **Q.2.17** Definitions:
    * a thread is ...
    * a process is ...

2. Memory (Storage) Management

- OS must manage the cache, main memory and secondary storage units responsibilities include:
  - prevent processes from interfering with each other's memory
  - make allocation of storage transparent
  - handle shared memory

- Virtual memory (VM) allows ...

- VM Details:
  - A process consists of multiple pages
    VM allows individual pages to be swapped in and out during execution
  - VM hides the features of the real memory system from the users
    conceals the fact that memory is limited
  - virtual address spaces can be much larger than real address space
  - VM overallocates memory to increase the degree of multiprogramming
  - **Q.2.18** VM addressing:
    * virtual or logical address =
    * real or physical address =

- Activities associated with memory management:
  - tracking which parts of memory are being used and by which processes
  - allocating memory when space becomes available
  - dynamic mapping between virtual and real addresses

- Activities associated with file management:
  - create/modify/delete files/directories
  - mapping of files onto secondary storage

3. Information Protection and Security

As the Internet grows, there is an increase in concern for the protection of information
Q.2.19 Much of work in this area falls into four categories:

- availability -
- confidentiality -
- data integrity -
- authenticity -

4. Scheduling and Resource Management

- any resource allocation and scheduling policy must consider three factors:
  - fairness to users
  - efficiency for system
  - differential responsiveness
- **Q.2.20** Schedulers:
  - short-term (or CPU) scheduler:
  - medium-term (or swapper) scheduler:
  - long-term (or job) scheduler:
- I/O queues exist for each I/O device

5. System Structure

- Monolithic systems: no partition of OS into smaller parts
  one HUGE program
  - structure = no structure
    all procedures for OS mixed together
  - problem: as OS grows, complexity of system becomes overwhelming
  - Example: OS/360, version 1
    created by 5,000 programmers over five years
    in 1964: over a million lines of code
  - Example: Multics
    in 1975: over 20 million lines of code
- Layered systems:
  - divide OS functions into a number of layers
    each layer built on top of other layers
  - bottom layer (layer 0) is the hardware
    highest layer (layer N) is the user interface
    a layer communicates with the layer directly above and directly below
  - easier to debug with separate layers
  - allows one layer to change without needing to change other layers

  - **Q.2.21** Problems with layered approach
    - *
    - *
    - *
- Microkernels:
  - moved from horizontal layers to vertical layers
  - define kernel as small as possible
  - implement rest of OS as user processes
  - kernel should be the **ONLY** code that is machine/device dependent
  - client/server communication model (one interface)
  - **Q.2.22** Advantages of a microkernel over layered system are ...
    - *
    - *
    - *
    - *
    - *
    - *
    - *
  - **Q.2.23** Disadvantage of a microkernel:
- Modular systems: needed modular software for growing operating systems
  - Past: original Unix had two modules defined
  - Current: many modules use OO programming techniques
  - **Q.2.24** Advantages of modular design:
    - *
    - *
    - *

2.4 Modern OS Developments

- modular design
- multithreading
- symmetric multiprocessing
- distributed operating systems

2.5 System Calls

Section 1: General Commands

1. file management: touch, rm, cp, chmod, mv, ls, cat, cp
2. file modification: vi, emacs, awk
3. status information: time, du, cal, df
4. PL support: gcc, java
5. communications: mail, rsync, rcp

Section 2: System Calls

- **Q.2.25** Definition of a system call ...

- **Q.2.26** one possible classification of System Calls in Unix:
  1.
  2.
  3.
  4.
  5.

- **Q.2.27** Definition of a system program ...

2.6 Unix Processes

Process creation:

- `cpid = fork();`
  creates copy of the process that executed `fork();`
  - address spaces for the two processes are (almost) identical
  - both processes have identical open files
  - both processes have PC pointing to statement after the `fork()` command
  - fork() example:

```
#include <unistd.h>   /* include file for fork/exec */
main()  {
    int cpid;
    cpid = fork();
    cout << "Hello there ... " << cpid << endl;
}
```

  Q.2.28 Questions:

  - When this code is executed, what is printed?
  - What is the difference between the two processes?

- The *process tree* for this example has a parent process pointing to a child process
- NOTE: communication between these two processes is not available
  must use IPC paradigm or shared memory system calls for communication

Process trees

- fork() fork() example:

```
main()  {
   int cpid1=-111;
   int cpid2=-222;  // LINE A
   cpid1 = fork();  // LINE B
   cpid2 = fork();  // LINE C
   cout << "Hello there ... " << cpid1 << "   " << cpid2 << endl;
}
```

  Q.2.29 Questions:

  - What is the process tree in the double fork example?
  - How are the processes distinguished?

Process execution

- `execv(file, arg_list);`
  `execl, execlp, execle, execv, execvp, execvP`

- process that calls `exec()` is demolished
  `exec()` overlays (most of) address space of caller with executable `file`
- NOTE: open files inherited

Process Synchronization:

- `cpid_done = wait(status);`
  caller sleeps until any child terminates
- `cpid_done = waitpid(pid, status, options);`
  caller sleeps until child with pid terminates
- returns integer which is the pid of terminating child
- if status is not NULL, then stores status of child termination
- Questions (for unix environment):
  - What happens to child if parent is killed?
  - What if child exits before the parent waits?

Typical use:

- primary purpose of forking a child is to execute some other program
  typically use "exec" to overlay new process on top of old process
- typically parent "wait"s until child is finished
  then, after execution of the other process, the parent gains control again
- Example:

```
fork()
if child
   exec()
else
   wait()
```


In fact, a shell (e.g., /bin/sh) basically does:

```
while (not EOF) do
{
    parse_command_line; (get command, arguments, I/O redirection, etc)
    if ((cpid = fork()) == 0) then
    {
        redirect I/O
        exec(cmd, args)
    }
    else
    {
        if (command doesn't end with &), then wait();
    }
}
```

2.7 Unix Pipes for Interprocess Communication (IPC)

Unix IPC: through the filesystem
file descriptors, read()/write(), pipes

What is a Unix pipe?

Pipes:

- pipe allows a one-way communication flow between two processes
- pipe construct shared thru process hierarchy inheritance rules
  common ancestor must establish pipe
- produces a totally reliable *byte stream* between two processes communicating
  FIFO queue of bytes

Definition of a pipe:

```
int fd[2];
int pipe(int *fd);
```

- fd[0]: opened for reading
- fd[1]: opened for writing

To obtain IPC between two processes:

define pipe
fork child
have parent close one end of pipe
have child close the other end of pipe
read/write
close other two pipe ends

Q.2.30 How to do two-way communication?

```
/**********************************/
/*          FORK EXAMPLE            */
/********************************/

#include <iostream>
#include <errno.h>      /* include file for perror */
#include <unistd.h>     /* include file for fork/exec */
#include <sys/types.h>  /* include file for wait */
#include <sys/wait.h>   /* include file for wait */
#include <stdio.h>
using namespace std;

main()
{
  int cpid;
  int cpid2;

  if ((cpid = fork()) == -1) {
    perror(``fork failed'');
    return(-1);
  }

  cout << ``The child is started.'' << endl << endl;

  if (cpid == 0) {       /* This is the child. */

    cout << endl << ``I'm the child and my pid is `` << getpid() << endl;
    cout << ``I'm waiting for the execute sacrifice.'' << endl << endl;

    execl(``/bin/ls'', ``ls'', ``-l'', (char *)0);   /* execute a process */

    perror(``exec failed'');
    return(-1);
  }

  else {                 /* This is the parent. */

    cout << endl << ``I'm the parent. I'll wait on my child.'' << endl;

    cpid2 = wait( (int *) 0);    /* wait for child to finish */

    cout << endl << ``The child with pid = ``
                           << cpid2 << `` is done.'' << endl;
    return(0);
  }
}
```



```
/*********************************/
/*          PIPE EXAMPLE             */
/*******************************/
#include <iostream>
#include <errno.h>  /* include file for perror */
#include <unistd.h> /* include file for pipes */
#include <stdio.h>
#define DATA "There are 10 types of people ... "
using namespace std;

main()
{
  int the_pipe[2];
  int child;

  if (pipe(the_pipe) < 0) {           /* Creates a pipe */
    perror("opening pipe");
    return(-1);
  }
  if ((child = fork()) == -1) {
    perror("fork");
    return(-1);
  }

  if (child == 0) {       /* This is the child. */
    close(the_pipe[0]);   /* It writes msg to its parent */

    cout << "I'm the child; listen to my message!" << endl << endl;

    if (write(the_pipe[1], DATA, sizeof(DATA)) < 0)
      perror("writing message");
    close(the_pipe[1]);
  }
  else {
    char buf[1024];       /* This is the parent. */
    close(the_pipe[1]);   /* It reads the child's message */

    cout << "I'm the parent; I should listen to my child." << endl << endl;

    if (read(the_pipe[0], buf, 1024) < 0)
      perror("reading message");

    cout << "My child says: " << buf << endl;
    close(the_pipe[0]);
  }
  return(0);
}
```