COMP310/ECSE427 Study guide

Francis Piché

September 21, 2018

Contents

Part I
Preliminaries

1 Disclaimer

These notes are curated from Professor Muthucumaru Maheswaran COMP310/ECSE427 lectures at McGill University, and *A. Tenenbaum and H. Bos, Modern Operating Systems, 4th Edition, Pearson, 2015*. They are for study purposes only. They are not to be used for monetary gain.

2 About This Guide

I make my notes freely available for other students as a way to stay accountable for taking good notes. If you spot anything incorrect or unclear, don't hesitate to contact me via Facebook or e-mail at `http://francispiche.ca/contact/`.

Part II
Overview of OS Concepts

3 Introduction

3.1 What is an OS?

An operating system is a trusted software which interfaces between the hardware and user applications to provide:

- Security
- Usability
- Efficiency
- Abstractions
- Resource management

They attempt to solve the problem of maximizing utilization, and minimizing idle time, to maximize throughput.

3.2 Design Concerns For Different OS's

For a personal/embedded system: response time should be minimal

For a time-sharing system: there should be *fair* time sharing

In batch-processing systems: goal is to maximize throughput.

3.3 Booting

First, the hardware is powered and the CPU is in "real-mode" which is essentially "trust everything mode". From here, the BIOS are loaded from ROM, and the CPU switches to "protected" or "user" mode. Finally the Kernel finishes initialization and the kernel services (OS) are started.

3.4 Processes

A process is a running program. Each process has an "address space" or "**core image**" in memory which it is allowed to use.

This address space contains the program's:

- executable code
- data (variables etc)
- call stack

The **process table** is an array of structures containing each process in existence. This includes all of the state information about each program, even if it is in a suspended or background state. (Some programs may run periodically or in the background).

A **child process** is a process that was created by another process.

Each person using a system is assigned a **UID**. Each process has the UID of the person who started it. Child processes have the UID of their parent.

Multiple processes allows for better utilization since while one process is idle (for example waiting for I/O) another process can work.

3.5 Memory Management

Processes need to be kept separate from other processes, and memory must also allow more than one process to exist in RAM at the same time.

We might also need to have these processes communicate.

Thanks to virtual memory, the address space can be larger than the actual amount of physical memory addresses. For more on virtual memory, see my COMP273 guide (or this guide in the later sections.) This also allows for better "chunking" of memory to keep processes separate. We can also provide shared memory spaces for inter-process communcation. Virtual memory also allows for processes to not care where in memory they actually are, which is useful since they may be moved around when they get "kicked out" by another process.

3.6 Storage

Need persistent storage, but it's slow. A hard disk is broken up into blocks which contain binary data. So when you use files, for example, they are saved as a series of blocks of binary data on the secondary storage.

3.7 OS and Interrupts

There are two kinds of Interrupts, hardware and software. Hardware is when a device sends a signal to the CPU, for example, a mouse is moved and needs to be processed. Software is when a program (OS or otherwise) throws an exception (trap). This alters the regular flow of the CPU and is therefore an interrupt.

3.8 Dual-Mode OS

Dual mode is the idea of keeping Users (unprivileged) and the kernel (privileged) separate. This provides greater security since this ensures on the trusted OS can make potentially dangerous operations on the core of the computer.

When a program running in user mode sends a system call, the OS then switches to kernel mode to complete the operation, and back when complete.

3.9 Monolith vs Micro-Kernel

There are two main architectures for an OS, the Monolith (one big program that handles everything) and the Micro-kernel. The latter's kernel is just the bare minimum inter-process communication, while the rest is a series of micro-services each capable of executing one OS task.

Part III
Scheduling

4 Processes

A process is an abstraction of a running program. (As stated previously). It is necessary because we need to be able to handle multi-programming. For example, a webserver handling many user requests at the same time.

This allows for better utilizaton (even if only 1 core CPU) since while one program needs to wait (IO or something) another can run. While there is overhead for context-switching, it is still generally faster to do this.

4.1 Process Representation

The process needs to contain:

- Program counter
- Stack
- Register state
- Memory state
- Stack etc.

A process is represented by a process control block. This is a table whose entries (indexed by process id's (PID)), are a sub-table containing (at least):

- CPU State
- Processor ID
- Memory
- Open files
- Status
- Parent and Child process'
- Priority

So to switch context, we would need to save all the things we need, load the new process, run it for a while, and repeat.

4.2 Lifecycle Management

We need to manage how process are handled through their whole lifecycle. This includes their creation, state changes and termination.

Processes are created using a system-call (can't just make processes willy-nilly, need to call the kernel).

There are two approaches to this, either we build the table from scratch, or we clone an existing process. The latter is the UNIX approach and the one we'll focus on. In this, we stop the current process and save it's state. We then make a copy of all the code, data, heap and PCB and give this new process a new PID. Then, we pass it to the dispatcher which will schedule the process.

4.2.1 fork()

This cloning method is done using the `fork()` system call, after which the parent and child run concurrently. Note that the `fork()` call returns the PID of the child process to the parent, and 0 to the child if successful, and -1 otherwise.

For example:

```
main() {
   int i;
   i = 10;
   if (fork() == 0) i+= 20;
   printf(" %d ", i);
}
ouput> 10, 30
```

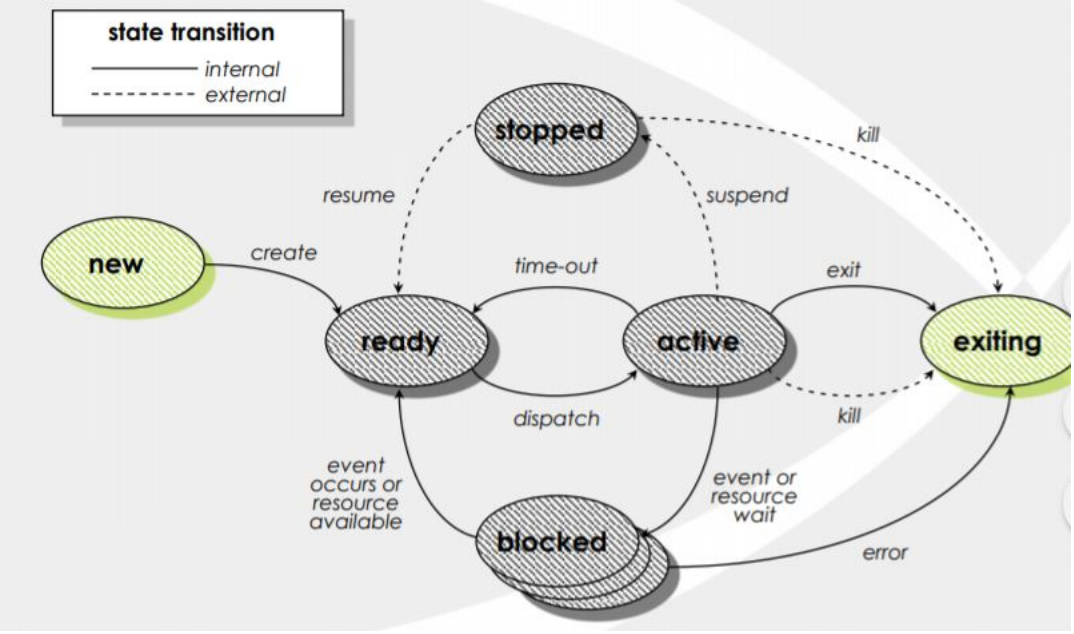
The parent outputs 10 (unchanged) and the child outputs 30 (was affected by the if statement).

4.2.2 State

A process can change state while it is executing. For example if theres no memory or processor available, waiting for some outside event, we finished the task and exit, etc. The possible states are:

- New (Process creation begins)
- Ready (Process has been created and is ready to be loaded and run.)
- Blocked (Waiting for some reason)
- Active (Running)

- Stopped (Suspended by the scheduler)
- Exiting (has been terminated (either normally or by killing))



A process enters the exiting state when either it finishes its task (uses an `exit()` call), caught an exception, or some user decided to kill it.

4.2.3 Tiny Shell

This is Prof Maheswaran's example of a shell.

```
while(1){
   printf("Prompt>")
   getline(line)
   if(strlen(line) > 1){
      if(fork() == 0){
         exec(line)
      }
      wait(child)
   }
}
```

In this, the parent creates a new child process for each command, and waits for the child to complete.

4.3 Dispatcher

There are two ways for the dispatcher to get control of the CPU (since it can only be used by one process at a time, and the dispatcher itself is a process), waiting (trust the other process) or by interrupts. The latter is preferred, since the other process may have an infinite loop,

otherwise be bad.

When an interrupt is sent, the OS saves the state of the active process, and runs the interrupt routine. (This is the process of saving the PC, status, registers, file pointers, memory etc.) Note that while this process occurs, no other interrupts are allowed.

Also note that memory is not always saved to disk. Since we use isolated address spaces for each process, the memory of one process (even if idle) need not be overwritten by another.

5 Process I/O

How do processes deal with I/O?

They have an array of "handles" which are each hooked up to an external device. (Mouse, keyboard, file, etc) These handles allow for easy communication between the process and any external I/O devices.

This table exists in the Kernel memory. (Restricted block of RAM). So the only way for a process to access this table is through a system call.

5.1 File Descriptors

We call the slots of the array `file descriptors`. In code, this looks like:

```
main() {
   char buf[BUFFERSIZE];
   const char* note = "Write failed\n";

   while ((n = read(0, buf, sizeof(buf)) > 0))
      if(write(1, buf, n) != n){
         (void)) write(2, note, strlen(note));
         exit(EXIT_FAILURE);
      }
   return(EXIT_SUCCESS)
}
```

Where we are reading from 0, writing to 1 and 2. We call 0 `standard input`, 1 `standard output`, and 2 `standard error`.

We can then use the `read()` and `write()` methods to specify which file descriptor to use.

If we use `open()` to open a file. This returns a file descriptor of $n > 2$. We can then pass this number to a system call such as `read()` or `write()` to access the file.

It's worth noting that the operations are not being sent/recieved directly to/from disk, since that would be slow. We actually interface with a cached version of the file from kernel memory.

We can do: `close(1)` and `open("file")` to overwrite the 1 to point to the file. (Since open goes to the first available array slot)

Similarly we can do `close(0)` to do input redirection.

We can also implement piping. This is when we have one process' output be the input for another process. We need to pass through the kernel space. We use the `pipe()` system call. The pipe creates two file descriptors, one for each side of the pipe. This works since the pipe (a space in the kernel) is created by a parent process, and since the `fork()` system call clones everything from the parent to the child, the child will have access to the fork.

Then, we rewire the file descriptors such that we set one end of the pipe to 1 (in the parent process), and the output end to 0. (in the child process) This is done using a close, and a `dup()`, which duplicates a file descriptor and puts it in the first open spot.

6 Process Address Space

Part IV
Memory & Virtualization

Part V
Security