# ICS431-OS
# Homework 2

Ahmed Alelg - 201507470

Monday 2nd March, 2020

# Chapter 1

**1.5** How do clustered systems differ from multiprocessor systems? What is required for two machines belonging to a cluster to cooperate to provide a highly available service ?
**Answer:** Clustered systems consist of two or more systems that can have a single or multiprocessor, they also share storage, and are linked via fast LAN cables. To provide a highly available service one machine must be in a hot-standby mode; watching for failures that might occur in any other machine, to take its position.

**1.8** What is the purpose of interrupts? How does an interrupt differ from a trap? Can traps be generated intentionally by a user program? If so, for what purpose?

**Answer:** Interrupts are issued externally either by I/O devices to signal completion or any other state or by system calls to request an I/O operation among other things. While traps or exceptions are issued internally by the processor due to software error; such as division by zero, or accessing unallocated memory, and it can be generated intentionally by a user program. In both situations the control flow is transferred to the kernel mode to execute a specific service routine.

**1.13** Discuss, with examples, how the problem of maintaining coherence of cached data manifests itself in the following processing environments:

**Answer:**
**Single-processor systems:** Since one process runs at a time, the right data is accessed each time hence it is coherent.
**Multiprocessor systems:** If one process updates a variable then it must be immediately updated in all other caches in other CPUs, otherwise coherency will not be achieved; because a variable may reside in two different local caches.
**Distributed systems** Since storage is shared, then if a file is updated in one machine it must be updated in every other machine.

**1.16** Describe some of the challenges of designing operating systems for mobile devices compared with designing operating systems for traditional PCs
**Answer:** Because computing, power, memory and storage are limited in mobile phones, and to run efficient the OS must implement polices that: lower power consumption, terminates unnecessary applications, and wise memory management.

**1.18** Describe some distributed application that would be appropriate for a peer-to-peer system.
**Answer:** File sharing system, on-demand Internet video streaming services.

# Chapter 2

**2.7** What are the two models of interprocess communication? What are the strength and weaknesses of the two approaches?
**Answer:** Either via **shared memory** where the two processes communicate via reading and writing in a certain block of memory. Or via **message passing** where the OS deliver packets of one process to the other.

**2.8** Why is separation of mechanism and policy desirable?
**Answer:** To allow flexibility; different OS types may require different polices to fit the device requirements, which also may change over time. Furthermore, the security of the system will increase, because if one was compromised the other will not be since they are separated.

**2.9** It is sometimes difficult to achieve a layered approach if two components of the OS are dependent on each other. Identify a scenario in which it is unclear how to layer two system components that require tight coupling of their functionalities.

**Answer:** Backing store of virtual memory may needs I/O access provided by the CPU scheduler hence it must be at a lower level. On the other hand, CPU scheduler may needs to swap information in or out of memory requiring Backing store to be at a lower level.

**2.12** What are iOS and Android similar? How are they different?
**Answer:** iOS is derived from OS X, Android is, unlike the latter, open-source OS that is based on Linux kernel. Both OSes provide rich framework for app development, and have stack architecture.

**2.14** The experimental Synthesis operating system has an assembler incorporated in the kernel. To optimize system-call performance, the kernel assembles routines within kernel space to minimize the path that the system call must take through the kernel. This approach is the antithesis of the layered approach, in which the path through the kernel is extended to make building the operating system easier. Discuss the pros and cons of the Synthesis approach to kernel design and system-performance optimization.
**Answer:** Pros: Faster compilation and response time.
Cons: Hard maintenance because coupling is increased. Each architecture may require different compiler. Extra size on the memory.

# Chapter 3

**3.1** Describe the differences among short-term, medium-term, and long-term scheduling.

**Answer:**
short-term: selects and dispatch processes from the ready queue to the CPU

based on a certain policy that tries to achieves OS goals, it is also preemptive; takes the processes with expired time from the cpu to make context switches.

medium-term: balance between concurrent processes by swapping processes between the virtual and main memory

long-term: admits new processes from the job queue to the ready queue, it must balance between I/O and CPU bound processes and concurrency.

**3.3** Construct a process tree to Figure 3.8 to obtain process information for the UNIX and Linux system, use the command `ps -ael`

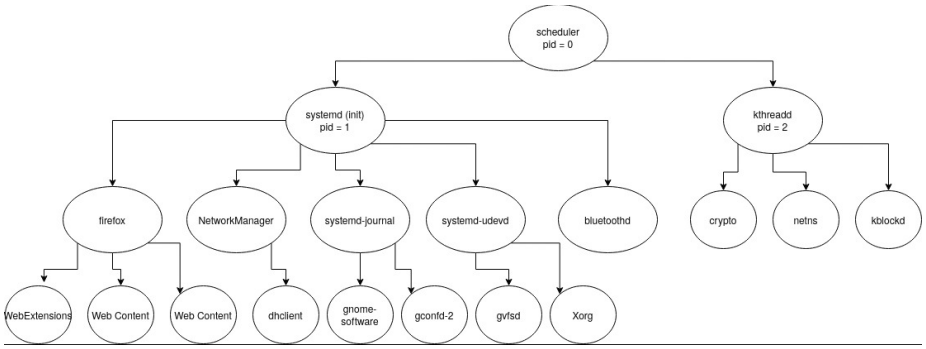


Figure 1: Process tree

**3.5** Including the initial parent process, how many processes are created by the program shown in Figure 3.31

**Answer:** Total = 15 processes as shown in the table:

| i | processes |
|---|---|
| 0 | 1 |
| 1 | $1 + 1 = 2$ |
| 2 | $2 + 2 = 4$ |
| 3 | $4 + 4 = 8$ |

**3.9** Consider the RPC mechanism. Describe the undesirable consequences that could arise from not enforcing either the "at most once" or "exactly once" semantic. Describe possible uses for a mechanism that has neither of these guarantees.

**Answer**

**At most once**: in important but not critical situations such as food delivery request, if multiple duplicates requests were sent then this is undesirable because the customer will pay more for his order and will get extra unwanted food. And

there will be no critical problem if the request hasn't been sent.

**Exactly once**: in important and critical situations such as ambulance request, duplicates will result in sending duplicate ambulances, and if no ambulance was sent then this will result in life threading situation.

Weather retrieval information doesn't require any of these semantics.

**3.10** Using the program shown in Figure 3.34, explain what the output will be at lines X and Y.

**Answer**

Because the child gets its own address space, the modified data will not affect the parent process. The expected output for line X and Y respectively is:

```
CHILD: 0 CHILD: -1 CHILD: -4 CHILD: -9 CHILD: -16
PARENT: 0 PARENT: 1 PARENT: 2 PARENT: 3 PARENT: 4
```

## Practical Questions

Using either a UNIX or a Linux system, write a C program that forks a child process that ultimately becomes a zombie process. This zombie process must remain in the system for at least 10 seconds.

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int main(int argc, char *argv[]) {


pid_t cp = fork();
if (cp == 0){
       sleep(2);
       printf("child exiting now, and turning into zombie\n");
       exit(0);
}

printf("Parent is waiting\n");
sleep(30);
printf("Parent is exiting\n");

}
```

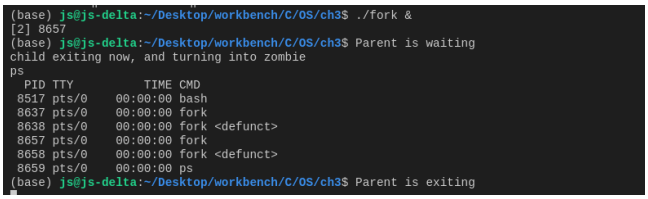


Figure 2: Zombie process

**3.13** Write a PID manager.

```
#include <stdio.h>
#include <stdlib.h>

#define MIN_PID 300
#define MAX_PID 5000

int allocate_map(void);
int allocate_pid(void);
void release_pid(int pid);

int* pids;

int main(int argc, char *argv[]) {
allocate_map();
int pid1 = allocate_pid();
int pid2 = allocate_pid();
printf("pid1:%d\npid2:%d\n", pid1, pid2);
printf("deleteing pid1\n");
release_pid(pid1);
}

int allocate_map(void) {
pids = (int*) malloc( (MAX_PID-MIN_PID) * sizeof(int));
if (pids != NULL)
return 1; /* unsuccessful */
return 1;
}

int allocate_pid(void) {
int i;
int items = (MAX_PID-MIN_PID);

for(i=0; i< items && pids[i]!=0; i++);
```

```
if(i==items)
return 1; /* full */

pids[i] = 1;
return i + MIN_PID;
}

void release_pid(int pid) {
pids[pid] = 0;
}
```

**3.1** Collect the following basic information about your machine using proc. How many CPU cores does the machine have? How much memory and what fractions of it are free? How many context switches has the system performed since booting-up? How many processes has it forked since booting-up?

**Answer**: The machine have 2 cores, $3.95GB$ memory of which $0.829GB$ is free. The system has preformed $7,202,048$ context switches, and forked $4,046$ processes since its boot up.

```
(base) js@js-delta:~$ sudo cat /proc/cpuinfo
[sudo] password for js:
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 6
model           : 42
model name      : Intel(R) Core(TM) i5-2430M CPU @ 2.40GHz
stepping        : 7
microcode       : 0x2f
cpu MHz         : 1901.911
cache size      : 3072 KB
physical id     : 0
siblings        : 4
core id         : 0
cpu cores       : 2
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
wp              : yes
flags           : fpu vme de pse tsc msr pae mce cx8 apic se
be syscall nx rdtscp lm constant_tsc arch_perfmon pebs bts n
cpl vmx est tm2 ssse3 cx16 xtpr pdcm pcid sse4_1 sse4_2 x2ap
tibp tpr_shadow vnmi flexpriority ept vpid xsaveopt dtherm i
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_st
bogomips        : 4789.33
clflush size    : 64
cache_alignment : 64
address sizes   : 36 bits physical, 48 bits virtual
power management:
```

Figure 3:

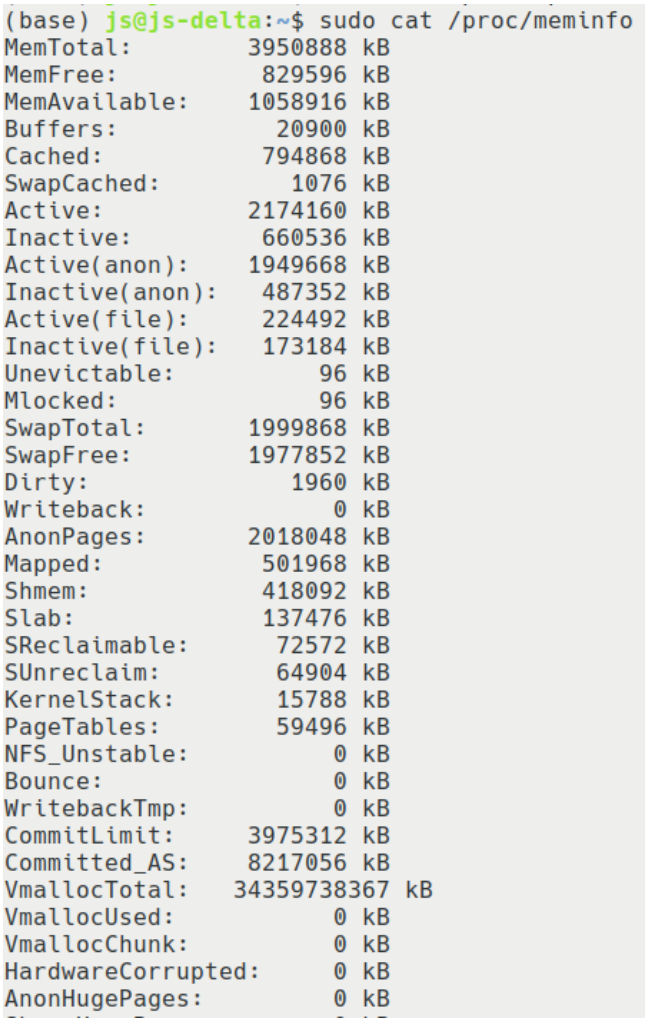
(base) js@js-delta:~$ sudo cat /proc/meminfo
MemTotal: 3950888 kB
MemFree: 829596 kB
MemAvailable: 1058916 kB
Buffers: 20900 kB
Cached: 794868 kB
SwapCached: 1076 kB
Active: 2174160 kB
Inactive: 660536 kB
Active(anon): 1949668 kB
Inactive(anon): 487352 kB
Active(file): 224492 kB
Inactive(file): 173184 kB
Unevictable: 96 kB
Mlocked: 96 kB
SwapTotal: 1999868 kB
SwapFree: 1977852 kB
Dirty: 1960 kB
Writeback: 0 kB
AnonPages: 2018048 kB
Mapped: 501968 kB
Shmem: 418092 kB
Slab: 137476 kB
SReclaimable: 72572 kB
SUnreclaim: 64904 kB
KernelStack: 15788 kB
PageTables: 59496 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 3975312 kB
Committed_AS: 8217056 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 0 kB
VmallocChunk: 0 kB
HardwareCorrupted: 0 kB
AnonHugePages: 0 kB

Figure 4:

```
(base) js@js-delta:~$ cat /proc/stat
cpu  80472 354 15058 379783 2354 0 599 0 (
cpu0 21864 23 3946 91689 1641 0 158 0 0 0
cpu1 18673 39 3455 97661 141 0 16 0 0 0
cpu2 20933 240 4034 94262 273 0 76 0 0 0
cpu3 19001 51 3622 96169 297 0 348 0 0 0
intr 2367563 8 1759 0 0 0 0 0 0 1 115 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
ctxt 7202048
btime 1583134434
processes 4046
procs_running 3
procs_blocked 0
```

Figure 5:

**3.2** Every process consumes some resources (CPU, memory, disk or network bandwidth, and so on). When a process runs as fast as it can, one of these resources is fully utilized, limiting the maximum rate at which the process can make progress. Such a resource is called the bottleneck resource of a process. A process can be bottlenecked by different resources at different points of time, depending on the type of work it is doing. Run each of the four programs (CPU, CPU-print, disk, and disk1) separately, and identify what the bottleneck resource for each is (without reading the code). For example, you may monitor the utilizations of various resources and see which ones are at the peak. Next, read through the code and justify how the bottleneck you identified is consistent with what the code does. For each of the programs, you must write down three things: the bottleneck resource, the reasoning that went into identifying the bottleneck, (e.g., the commands you ran, and the outputs you got), and a justification of the bottleneck from reading the code.

**Answer:**

The bottleneck resource for `cpu.c` is CPU, as shown in the figure below, no read or write activity was detected using `htop` command. Furthermore the source code feature intensive CPU-bound instructions namely infinite multiplication, but doesn't contain any I/O-bound instructions.


Figure 6: cpu.c

The bottleneck resource for `cpu-print.c` is CPU, and involves reading time

from memory and writing to the `stdout` as shown in the figure analysis below conducted by `htop`. The source code feature intensive CPU-bound instructions `count++` as well as I/O bound: `gettimeofday`.

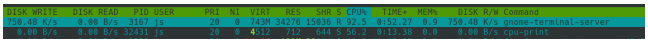


Figure 7: cpu-print.c

The bottleneck resource for `disk.c` is I/O, according to the analysis using `htop`, intensive read and write activity was detected since it was using intensive I/O bound instructions: `fopen` and `fwrite`.


Figure 8: disk.c

The bottleneck resource for `disk1.c` is CPU as well as disk as shown below, according to the analysis using `htop`, read and write activity was high it because features intensive I/O bound instructions such as `fopen` and `fwrite`.

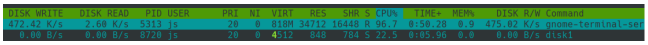


Figure 9: disk1.c

**3.3** Recall that every process runs in one of two modes at any time: user mode and kernel mode. It runs in user mode when it is executing instructions (code from the user). It executes in kernel mode when running code corresponding to system calls etc. Compare (qualitatively) the programs CPU and CPU-print in terms of the amount of time each spends in the user mode and kernel mode, using information from the proc file system. For examples, which programs spend more time in kernel mode than in user mode, and vice versa? Read through their code and justify your observations.

**Answer**

According to the manual of `proc`: the 14th value represents the user mode time measured in clock ticks which is 3535 according to the figure below, while the 15th value 40 contains the time that this process has been scheduled in kernel mode, measured in clock ticks, which is very low. Hence `cpu.c` operates mostly in user mode.

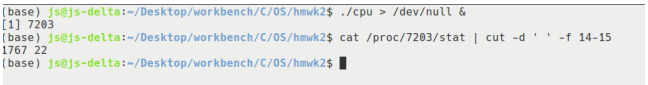


Figure 10: cpu.c

In contrast, `cpu-print.c`, spends more time in the kernel mode 2052 compared to the user mode 878 as shown in the figure:

```
(base) js@js-delta:~$ cat /proc/7579/stat | cut -d ' ' -f 14-15
878 2052
```

Figure 11: cpu-print.c

**3.4.** Recall that a running process can be interrupted for several reasons. When a process must stop running and give up the processor, it's CPU state and registers are stored, and the state of another process is loaded. A process is said to have experienced a context switch when this happens. Context switches are of two types: voluntary and involuntary. A process can voluntarily decide to give up the CPU and wait for some event, e.g., disk I/O. A process can be made to give up its CPU forcibly, e.g., when it has run on a processor for too long, and must give a chance to other processes sharing the CPU. The former is called a voluntary context switch, and the latter is called an involuntary context switch. Compare the programs CPUand disk in terms of the number of voluntary and involuntary context switches. Which program has mostly voluntary context switches, and which has mostly involuntary context switches? Read through their code and justify your observations.

**Answer:** `cpu.c` have a high number of non-voluntary context switching as shown in of them have non-voluntary context switches: 428 as shown below because all it does is computations. However `disk.c` does some reading and writing as discussed before, that is why it have more voluntary context switching 1169 compare to non-voluntary context switching 257 because it requires frequent I/O resources.

```
(base) js@js-delta:~/Desktop/workbench/C/OS/hmwk2$ ./cpu &
[1] 8020
(base) js@js-delta:~/Desktop/workbench/C/OS/hmwk2$ cat /proc/8020/status | grep
"ctxt"
voluntary_ctxt_switches:        0
nonvoluntary_ctxt_switches:     428
(base) js@js-delta:~/Desktop/workbench/C/OS/hmwk2$
```

Figure 12: cpu.c

```
(base) js@js-delta:~/Desktop/workbench/C/OS/hmwk2$ ./disk &
[1] 9543
(base) js@js-delta:~/Desktop/workbench/C/OS/hmwk2$ cat /proc/9543/status | grep
"ctxt"
voluntary_ctxt_switches:        1169
nonvoluntary_ctxt_switches:     257
```

Figure 13: disk.c