PRINCIPLES OF COMPUTER SYSTEMS DESIGN

CSE130

Winter 2020

Processes & Threads I

Notices

- **Administration 1 & 2** due 23:59 **Wednesday January 15**
- **Lab 1** due 23:59 **Sunday January 19** **CHANGED!**
- TA Office Hours:
  - Late afternoon / Early Evening in E2-380
  - See class Drupal site for details

Today's Lecture

- Process Overview
- Context Switching
- Process Lifecycle
- Process Control Block
- Process Management

- Lab 1 Secret Sauce

Runtime Context

- A fundamental responsibility of an Operating System is managing the **runtime context**:
  - In batching systems, jobs within a batch shared the same runtime context
  - In early multiprogrammed systems, in-memory jobs shared the same runtime context
  - The machine itself could be said to represent a single runtime context
- In more modern systems (from timesharing onwards) the process encapsulates its own runtime context

Introducing the Process

- A process is the **runtime context** of an executing program - the fundamental unit of work - it consists of:
  - Memory, open files, threads, executable code
  - State (program counter, register values, stack addresses, etc.)
  - Switching between processes is known as ***context switching***
- This information is stored by the OS in a **Process Control Block** (PCB)
- The OS uses PCBs to keep track of and manage all the processes in the system

Processes

- A process is:
  - Heavy weight - lots of runtime state (context)
  - A program may have more than one process
- A computing system is a collection of processes:
  - OS processes executing OS code
  - User processes executing user code
- Processes execute “concurrently” giving:
  - Better utilization of resources
  - Better user productivity

M processes, N processors

- OS typically has many processes running
  - System processes
  - User processes
  - On Linux, Unix, macOS, try:
    - `$ ps aux` (lists processes for all users)
    - `$ top` (live view of process attributes)
- Multicore CPUs mean most machines need to manage **M** processes over **N** CPU cores, and do so *intelligently*


```
Processes: 356 total, 2 running, 354 sleeping, 1769 threads                                09:26:54
Load Avg: 0.94, 0.78, 0.76  CPU usage: 3.62% user, 2.5% sys, 94.31% idle
SharedLibs: 160M resident, 42M data, 16M linkedit.
MemRegions: 95789 total, 2385M resident, 86M private, 801M shared.
PhysMem: 8024M used (2190M wired), 159M unused.
VM: 974G vsize, 637M framework vsize, 9026208(0) swapins, 9545504(0) swapouts.
Networks: packets: 25137087/15G in, 15112567/9133M out. Disks: 9475366/323G read, 7209454/932G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP  PPID  STATE
41153  Preview      11.1 00:40.74 7     4    292+  44M+   2008K  31M    41153 1     sleeping
158    WindowServer 8.3  02:22:12 6     2    682   156M+  1352K- 281M-  158   1     sleeping
48320  top          3.1  00:00.95 1/1   0    21+   5436K  0B     0B     48320 5062  running
1239   Terminal     2.9  55:49.56 13    8    506   65M    7200K  33M    1239  1     sleeping
0      kernel_task  2.1  02:16:31 133/8 0    2     980M-  0B     0B     0     0     running
121    hidd         1.8  30:56.42 6     2    242   2260K  0B     668K   121   1     sleeping
102    launchservic 1.7  01:31.28 6     6    920-  5216K  0B     4616K  102   1     sleeping
29885  X11.bin      1.4  11:58.80 10    2    231   5548K  0B     26M    29884 29884 sleeping
753    Dock         1.4  09:27.68 5     3    389-  10M+   24K    35M    753   1     sleeping
39766- Microsoft Po 1.0  45:50.00 13    4    650+  153M+  4220K  132M   39766 1     sleeping
40871  VBoxSVC      1.0  06:09.41 17    2    331   1684K  0B     8556K  40871 1     sleeping
5477   Dropbox      0.8  36:03.70 202   2    533   59M    0B     125M   5477  1     sleeping
117    loginwindow  0.8  02:11.84 3     2    739+  28M+   4096B  9148K  117   1     sleeping
750    Google Chrom 0.7  04:15:16 44    3    1255  169M   584K   331M   750   1     sleeping
47886  Google Chrom 0.3  00:13.34 13    1    68    12M    0B     9624K  750   750   sleeping
83     HDDFanContro 0.2  45:11.81 2     1    35    1804K  0B     576K   83    1     sleeping
124    notifyd      0.2  01:32.29 3     3    346   1828K  0B     312K   124   1     sleeping
324    distnoted    0.2  00:34.77 9     8    305-  4228K+ 0B     1876K  324   1     sleeping
59     fseventsd    0.1  03:23.14 12    1    299-  2652K  0B     6284K  59    1     sleeping
79     mds          0.1  14:12.71 14    9    639   19M    0B     37M    79    1     sleeping
14698- Microsoft Ex 0.1  15:21.51 13    3    284   14M+   0B     93M-   14698 1     sleeping
14308- Microsoft Wo 0.1  27:46.09 6     3    293   17M+   0B     142M-  14308 1     sleeping
1      launchd      0.1  21:30.15 5     5    2597- 12M+   0B     7048K  1     0     sleeping
40325  cloudd       0.1  00:11.75 9     7    160+  4560K+ 12K    4840K  40325 1     sleeping
96     opendirector 0.1  01:55.16 8     8    1242+ 5548K+ 64K    3364K  96    1     sleeping
40867  VirtualBox   0.0  02:03.46 9     1    227   14M    2828K  25M    40867 1     sleeping
40029  Google Chrom 0.0  01:56.19 24    1    141   79M+   0B     14M    750   750   sleeping
47885  Google Chrom 0.0  00:02.46 15    2    99    13M    0B     6996K  750   750   sleeping
```

Process Control Block

- Information about each process is stored in a PCB
- Used to save and restore state when context switching
- Exact contents are system dependent
- 200+ fields (2,000+ lines of definition) in recent Linux Kernels: `task_struct`

process state	next
	previous
process id	
program counter	
registers	
scheduling info	
memory structure	
open file table	
etc.	

https://raw.githubusercontent.com/torvalds/linux/master/include/linux/sched.h

Likely PCB fields

- **Process ID**: unique number identifying this process (PID)
- **Program Counter**: indicates the next program instruction to execute (PC)
- **Registers**: stack pointer, index registers, and various other system dependent registers
- **Scheduling Info**: priority, scheduling parameters, pointer to scheduling queue
- **Memory Structure**: Page tables, base register, etc.
- **Open File Table**: Set of open files allocated to this process
- **Accounting Information**: CPU time, elapsed time, memory size, page faults, IO blocks, time limits, account numbers

Where is the PCB?

- As it contains critical information about the processes it must be protected from normal user activity
  - In many Operating Systems it is placed at the beginning of the Kernel stack of each process (only accessible during system calls)

Process States & Transitions

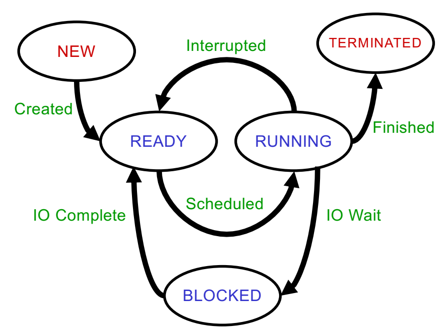
NEW:
The process (P) is being created

READY:
P is waiting to run on the CPU

RUNNING:
P's instructions are being executed

BLOCKED:
P is waiting on IO to complete

TERMINATED:
P's execution is complete



Choosing a Process to Run

- How does the OS decide which process to run next?
- It could...
  - Search a process list, run first ready thread it finds
  - Link together the ready threads into a queue (the ready queue)
    - When CPU becomes available, grab first thread from the ready queue
    - When threads become ready, insert at back of the ready queue
  - Give each thread a priority, organize the queue according to priority
  - Perhaps have multiple queues, one for distinct priority classes
- We'll cover all these in more detail later in the course
- You will tackle this in Lab 2 (woo-hoo!)

Return of Control

- CPU can only do one thing at a time
  - If a process is executing, the process dispatcher can not be
  - So the OS has lost control
  - How does OS regain control of the processor?
- Traps (exception or fault)
  - A system call
  - An error (illegal instruction, addressing violation, divide by 0, etc.)
  - A page fault (memory mistake)
- **Interrupts**
  - Character typed at keyboard
  - Completion of disk operation (controller is ready for more work)
  - Timer - make sure OS eventually gets control (Lab 1 ☺)

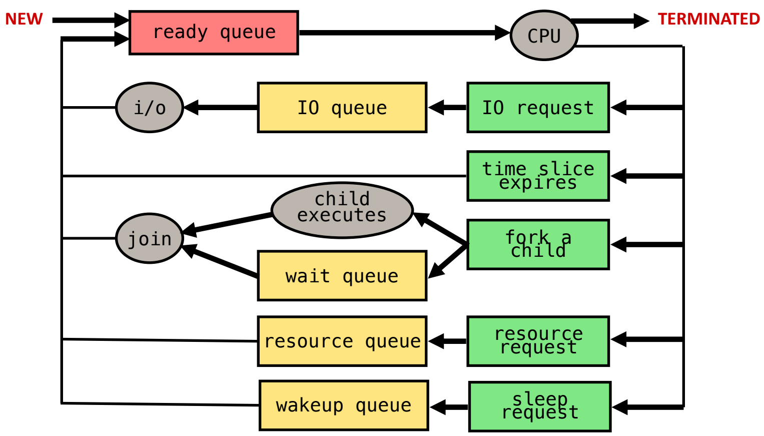
Managing Processes

- OS Process Management consists of:
  - Creating & deleting processes
  - Suspending and resuming execution of processes
  - Allocation of runtime resources
  - Synchronization and communication
- The PCB is maintained by the OS while performing these functions

Life Within the System

- To perform its work, a process consumes a set of runtime resources
- These resources are shared between many processes, available from multiple cores on most modern hardware, and must be intelligently managed by the OS
- **Essentially, process execution can be viewed as moving the process between various resource queues and the CPU**

Process Execution



Multiple Processes

- Advantages:
  - **Information Sharing**
    - concurrent access to shared files and other resources
  - **Performance**
    - while one cooperating process is IO blocked, another can still compute
  - **Modularity**
    - writing smaller programs to do specific tasks is safer and better design
  - **Convenience**
    - users wish to perform a number of different operations "at the same time"

Process Management

- The OS kernel offers various **system calls** to manage multiple processes:
  - create (`fork` and `exec`)
  - coordination (`wait`)
  - termination (`exit`)
  - process sessions and groups
  - communications (IPC and RPC)

Creating a Process

- When a process creates another:
  - the parent continues to execute alongside its child(ren), or..
  - the parent `wait` s (suspends itself) until the child(ren) terminate(s)
- The child may be either:
  - `fork` an exact (except for PID) duplicate of the parent, running the same program, from the same program counter (PC)
  - `exec` a different program altogether

Fork Steps

- A `fork` involves three main steps:
  - Allocating and initializing a new process structure for the child process (including PCB)
    - Enter in to child list of parent
    - Add to parent process group
    - Log Accounting Details
    - Assign PID
    - Etc.
  - Duplicating the entire context of the of the parent, including virtual memory
    - System privileges (of user), open file pointers, scheduling parameters, etc…
  - Scheduling the child process to run
- **Parent is blocked during child process creation**

Fork (UNIX) Example

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

int main(int argc, char* argv[]) {
    int status = 0;
    int wpid;

    int pid = fork();

    switch(pid) {
        case -1: /* error */
          fprintf(stderr, "fork failed\n");
          return -1;
        case 0: /* we are the child */
          printf("Child Running PID %d\n", getpid());
          break;
        default: /* we are the parent */
          printf("Parent Running PID %d\n", getpid());
          while ((wpid = wait(&status)) > 0) {
              printf("Exit status of %d was %d\n", wpid, status);
          }
    }

    return (EXIT_SUCCESS);
}
```


Exec (UNIX)

- In UNIX, `exec` causes the current process to begin execution of a new program, eg:

```
execl("/bin/ls", "/bin/ls", "-l", NULL);
```

- To perform a "child-exec" in UNIX:
  - First the parent forks
  - Then the child calls `exec` to run the new program

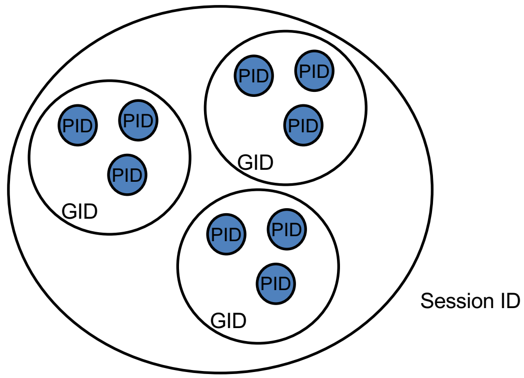
Termination

- When a process finishes execution, it terminates:
  - Voluntarily with the `exit` system call, or
  - Involuntarily as the result of a signal (e.g. `kill`)
- In either case, the exit status is returned to the parent
- A parent which is waiting (via `wait` system call) then resumes execution

Process Groups

- Identifies a set of processes working on one task
  - e.g. a database server
- Collect all their PIDs into a group
  - The set can now be managed as a whole (signals, accounting, filtering etc.)
- Sessions are sets of related groups, say all the groups and processes from a users login shell

Groups of Groups



Lab 1 Secret Sauce

- In the `timer_sleep()` function:
  - Put the current thread to sleep and immediately return
- At regular points in the future:
  - Wake up sleeping threads at or past their wakeup time

Next Lecture

- Threads