Week02 Process

Sirak Kaewjamnong

517-312 Operating Systems

OS concepts

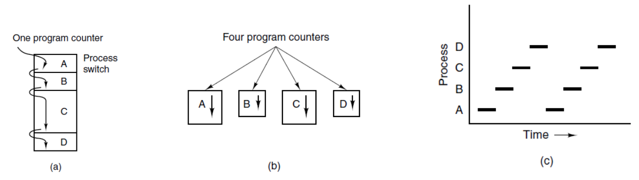
- Process : a program in execution
- Address spaces : memory holds executing programs, to keep them from interfering with others, protection mechanism is needed. The OS creates the abstraction of an address space as a set of process may reference
- Files : function of OS to hide the peculiarities of disks and I/O devices by presenting an abstract model of files.
- Input/Output: physical devices for acquiring input and producing output
- Protection: protecting the system form unwanted intruders
- Shell: the main interface between users and OS

Program and process

- Program
  - Wake up at 8.00 am
  - Take a shower
  - Dressed up
  - Go to room 1240
  - Sleep or study
- Process
  - Action
  - You are the running processes! ! !

What is process?

- Process is an instance of a running program
  - Current values of the program counter
  - Registers
  - variables



Process components

- Process States
- Program Counter
  - the address of the next instruction to be executed for this process
- CPU Registers
  - index registers, stack pointers, general purpose registers;
- CPU Scheduling Information
  - process priority;

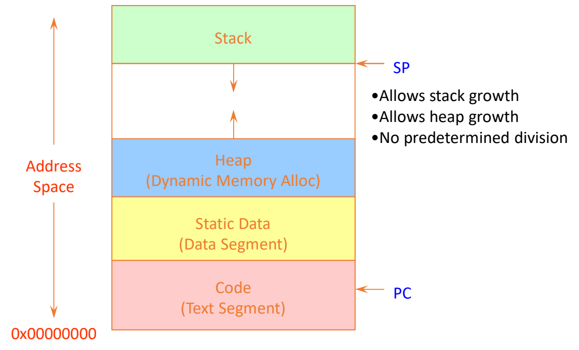
Process components

- Memory Management Information
  - base/limit information eg. virtual->physical mapping
- Accounting Information
  - time limits, process number; owner
- I/O Status Information
  - list of I/O devices allocated to the process;
- An Address Space
  - memory space visible to one process

Implementation of processes

- To implement the process mode, the OS maintains the process table
- It contains important information about process's state:
  - Program counter
  - Stack pointer
  - Memory allocation
  - Status of open files
  - Accounting and scheduling information
  - Every important information that must be saved when process is switched to other states

Process address space



Process creation

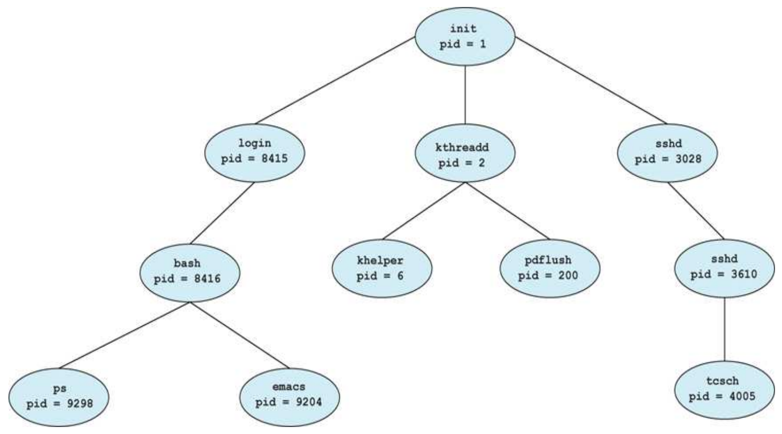
Four principal events that cause processes to be created:

1. System initialization.
2. Execution of a process creation system call by a running process.
3. A user request to create a new process.
4. Initiation of a batch job

Process creation

- A process is created by another process
- Parent is creator, child is created (Unix: ps “PPID” field)
- What creates the first process (Unix: init (PID 1))
- In some systems, the parent defines (or donates) resources and privileges for its children
- Unix: Process User ID is inherited – children of user’s shell execute with user’s privileges
- After creating a child, the parent may either wait for it to finish its task or continue in parallel (or both)

Process tree



http://www.padakuu.com/article/58-operations-on-process

Process creation: Unix

- In Unix, processes are created using fork()
  int fork()
- fork()
  - Creates and initializes a new PCB
  - Creates a new address space
  - Initializes the address space with a copy of the entire contents of the address space of the parent
  - Initializes the kernel resources to point to the resources used by parent (e.g., open files)
  - Places the PCB on the ready queue
- The child process then executes “exec” system call to change its memory image and run a new program

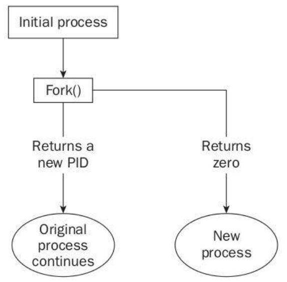
fork() system call

- call p = fork() , a new process (the child process) is created based on the current process. Thus a separated address space is created for the child.
- If fork() system call returns a negative value, it implies that the creation of the process was unsuccessful.
- If a new child process is created, fork() system call returns zero.
- fork() system call returns a positive value and the process ID of the child process to the parent.

http://www.eexploria.com/program-to-implement-system-calls-using-fork-function/

fork() example

- Step 1: Start
- Step 2: Declare pid
- Step 3: Call the fork function for pid. If pid return 0, it is child process
- Step 4: Else print the parent and child by getppid() and getpid().
- Step 6: Stop



fork() example

```
#include<stdio.h>
void main()
{
   int pid;
   pid = fork();
   if(pid == 0){
             printf("I am child pid %d and my parent pid %d\n",getpid(),getppid());
   }else{
      printf("I am parent pid %d and my parent pid %d\n",getpid(),getppid());
   }
   wait(); // Suspends the current process until a child process ends
}
```

OUTPUT

```
I am parent pid = 9701 and my parent pid is 8486
I am child pid = 9702  and my parent pid = 9701
```

exec() system call

- exec system call is used to execute an executable file. The previous executable file in process space is replaced and new file is executed.
- The current process is turned into a new process but the process id PID is not changed
- PID of the process is still but the data, code, stack, heap, etc. of the process are changed and are replaced with those of newly loaded process.
- The new process is executed from the entry point.

exec() system call

- Exec system call is a collection of functions and in C, the standard names for these functions are:

```
int execl(const char* path, const char* arg, …)
int execlp(const char* file, const char* arg, …)
int execle(const char* path, const char* arg, …, char* const envp[])
int execv(const char* path, const char* argv[])
int execvp(const char* file, const char* argv[])
int execvpe(const char* file, const char* argv[], char *const envp[])
```

- path is used to specify the full path name of the file which is to be executes.
- arg is the argument passed. It is actually the name of the file which will be executed in the process. Most of the times the value of arg and path is same.
- const char* arg in functions execl(), execlp() and execle() is considered as arg0, arg1, arg2, …, argn. It is basically a list of pointers to null terminated strings. Here the first argument points to the filename which will be executed as described in point 2.
- envp is an array which contains pointers that point to the environment variables.
- file is used to specify the path name which will identify the path of new process image file

Example fork() and exec() parent.c: the parent program

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>
int main (int argc, char **argv)
{
   int i = 0;
   long sum;
   int pid;
   int status, ret;
   char *myargs [] = { NULL };
   char *myenv [] = { NULL };
   printf ("Parent: Hello, World!\n");

   pid = fork ();
```

```
 if (pid == 0) {

      // I am the child

      execve ("child", myargs, myenv);
   }
   // I am the parent
   printf ("Parent: Waiting for Child to
complete.\n");

   if ((ret = waitpid (pid, &status, 0)) == -1)
       printf ("parent:error\n");

   if (ret == pid)
      printf ("Parent: Child process waited
for.\n");
}
```

Example fork() and exec() child.c: the child program

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

int main (int argc, char **argv)
{
   int i, j;
   long sum;
   // Some arbitrary work done by the child

   printf ("Child: Hello, World!\n");

   printf ("Child: Work completed!\n");
   printf ("Child: Bye now.\n");

   exit (0);
}
```

Example fork() and exec() build and run

```
$ # compile parent
$ gcc parent.c -o parent
$ #compile child
$ gcc child.c -o child
$ # run parent (and child)
$ ./parent
Parent: Hello, World!
Parent: Waiting for Child to complete.
Child: Hello, World!
Child: Work completed!
Child: Bye now.
Parent: Child process waited for.
$
```

Process creation: Windows

- The system call on Windows for creating a process is called, CreateProcess:
- BOOL CreateProcess(char *prog, ...) (simplified)
- CreateProcess
- Creates and initializes a new PCB
- Creates and initializes a new address space
- Loads the program specified by "prog" into the address space
- Places the PCB on the ready queue

Process termination

Typical conditions which terminate a process:

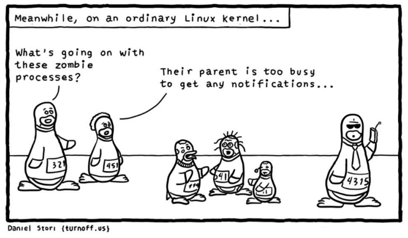
1. Normal exit (voluntary) : the process is finished
2. Error exit (voluntary): the process discovers a fatal error such as request to access an non existing file
3. Fatal error (involuntary): program's bugs such as dividing by zero, illegal instruction
4. Killed by another process (involuntary)

Process termination

- When exit() is called on Unix:
- Threads are terminated (next lec.)
- Open files, network connections are closed
- Address space is de-allocated
- But the PCB still remains in the Process Table
- Only a parent can remove the PCB
- Thus completely terminate the process (called reap)
- Died but not yet reaped process is called a zombie

Zombie process

- Zombie process is a process that has completed execution but still has an entry in the process table
- Zombie process do not use much of system resources, however their process id (PID) remain in the process table
- The systems like Linux have a limited number of process ID, with too many zombies, the available PIDs may run out
- A few zombies are no problem and may have no harm

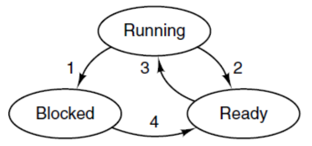

Meanwhile, on an ordinary Linux kernel...
What's going on with these zombie processes?
Their parent is too busy to get any notifications...
321
451
114
41
4315
Daniel Stori {turnoff.us}

http://turnoff.us/geek/zombie-processes/

Process states

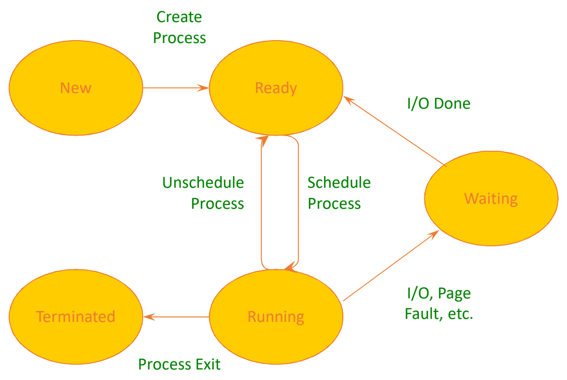
Three states a process may be in:

1. Running (actually using the CPU at that instant).
2. Ready (runnable; temporarily stopped to let another process run).
3. Blocked (unable to run until some external event happens).



1. Process blocks for input
2. Scheduler picks another process
3. Scheduler picks this process
4. Input becomes available

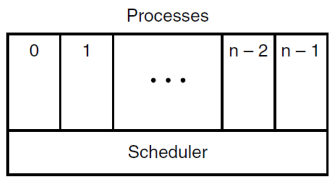
Process state graph



Presentations by Ding Yuan, ECE Dept., University of Toronto

Process states

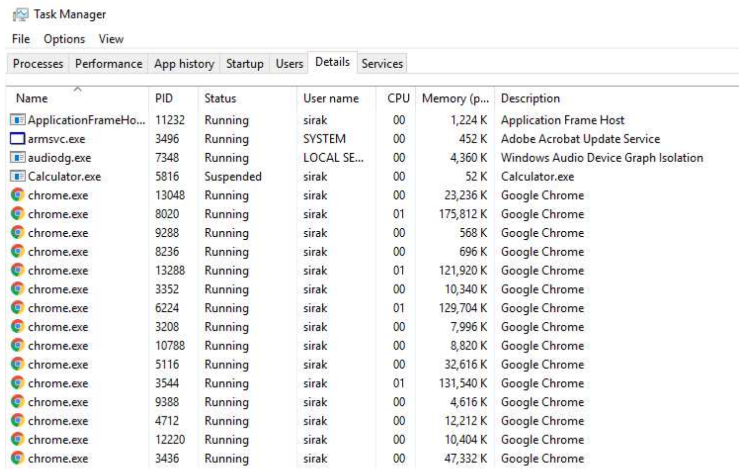
- A variety of processes are running on top of the scheduler
- The scheduler is the lowest level of the OS
- However, few real systems are not structured like this



Questions?

- What state do you think a process is in most of the time?
- For a single processor machine, how many processes can be in running state?
- Benefit of multi-core?

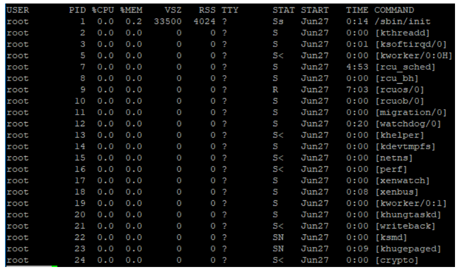
Processes in Windows10



Task Manager
File Options View
Processes Performance App history Startup Users Details Services

Name	PID	Status	User name	CPU	Memory (p...	Description
ApplicationFrameHo...	11232	Running	sirak	00	1,224 K	Application Frame Host
armsvc.exe	3496	Running	SYSTEM	00	452 K	Adobe Acrobat Update Service
audiodg.exe	7348	Running	LOCAL SE...	00	4,360 K	Windows Audio Device Graph Isolation
Calculator.exe	5816	Suspended	sirak	00	52 K	Calculator.exe
chrome.exe	13048	Running	sirak	00	23,236 K	Google Chrome
chrome.exe	8020	Running	sirak	01	175,812 K	Google Chrome
chrome.exe	9288	Running	sirak	00	568 K	Google Chrome
chrome.exe	8236	Running	sirak	00	696 K	Google Chrome
chrome.exe	13288	Running	sirak	01	121,920 K	Google Chrome
chrome.exe	3352	Running	sirak	00	10,340 K	Google Chrome
chrome.exe	6224	Running	sirak	01	129,704 K	Google Chrome
chrome.exe	3208	Running	sirak	00	7,996 K	Google Chrome
chrome.exe	10788	Running	sirak	00	8,820 K	Google Chrome
chrome.exe	5116	Running	sirak	00	32,616 K	Google Chrome
chrome.exe	3544	Running	sirak	01	131,540 K	Google Chrome
chrome.exe	9388	Running	sirak	00	4,616 K	Google Chrome
chrome.exe	4712	Running	sirak	00	12,212 K	Google Chrome
chrome.exe	12220	Running	sirak	00	10,404 K	Google Chrome
chrome.exe	3436	Running	sirak	00	47,332 K	Google Chrome

Processes in Linux



```
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.2  33500  4024 ?        Ss   Jun27   0:14 /sbin/init
root         2  0.0  0.0      0     0 ?        S    Jun27   0:00 [kthreadd]
root         3  0.0  0.0      0     0 ?        S    Jun27   0:01 [ksoftirqd/0]
root         5  0.0  0.0      0     0 ?        S<   Jun27   0:00 [kworker/0:0H]
root         7  0.0  0.0      0     0 ?        S    Jun27   4:53 [rcu_sched]
root         8  0.0  0.0      0     0 ?        S    Jun27   0:00 [rcu_bh]
root         9  0.0  0.0      0     0 ?        R    Jun27   7:03 [rcuos/0]
root        10  0.0  0.0      0     0 ?        S    Jun27   0:00 [rcuob/0]
root        11  0.0  0.0      0     0 ?        S    Jun27   0:00 [migration/0]
root        12  0.0  0.0      0     0 ?        S    Jun27   0:20 [watchdog/0]
root        13  0.0  0.0      0     0 ?        S<   Jun27   0:00 [khelper]
root        14  0.0  0.0      0     0 ?        S    Jun27   0:00 [kdevtmpfs]
root        15  0.0  0.0      0     0 ?        S<   Jun27   0:00 [netns]
root        16  0.0  0.0      0     0 ?        S<   Jun27   0:00 [perf]
root        17  0.0  0.0      0     0 ?        S    Jun27   0:00 [xenwatch]
root        18  0.0  0.0      0     0 ?        S    Jun27   0:08 [xenbus]
root        19  0.0  0.0      0     0 ?        S    Jun27   0:00 [kworker/0:1]
root        20  0.0  0.0      0     0 ?        S    Jun27   0:00 [khungtaskd]
root        21  0.0  0.0      0     0 ?        S<   Jun27   0:00 [writeback]
root        22  0.0  0.0      0     0 ?        SN   Jun27   0:00 [ksmd]
root        23  0.0  0.0      0     0 ?        SN   Jun27   0:09 [khugepaged]
root        24  0.0  0.0      0     0 ?        S<   Jun27   0:00 [crypto]
```

- D Uninterruptible sleep (usually IO)
- R Running or runnable (on run queue)
- S Interruptible sleep (waiting for an event to complete)
- T Stopped, either by a job control signal or because it is being traced.
- X dead (should never be seen)
- Z Defunct ("zombie") process, terminated but not reaped by its parent.
- < high-priority (not nice to other users)
- N low-priority (nice to other users)
- L has pages locked into memory (for real-time and custom IO)
- s is a session leader
- l is multi-threaded (using CLONE_THREAD, like NPTL pthreads do)
- + is in the foreground process group

https://unix.stackexchange.com/questions/18474/what-does-this-process-stat-indicates

Process data structures

How does the OS represent a process in the kernel?

- At any time, there are many processes in the system, each in its particular state
- The OS data structure representing each process is called the Process Control Block (PCB)
- The PCB contains all of the info about a process
- The PCB also is where the OS keeps all of a process' hardware execution state (PC, SP, regs, etc.) when the process is not running
- This state is everything that is needed to restore the hardware to the same state it was in when the process was switched out of the hardware

PCB data structure

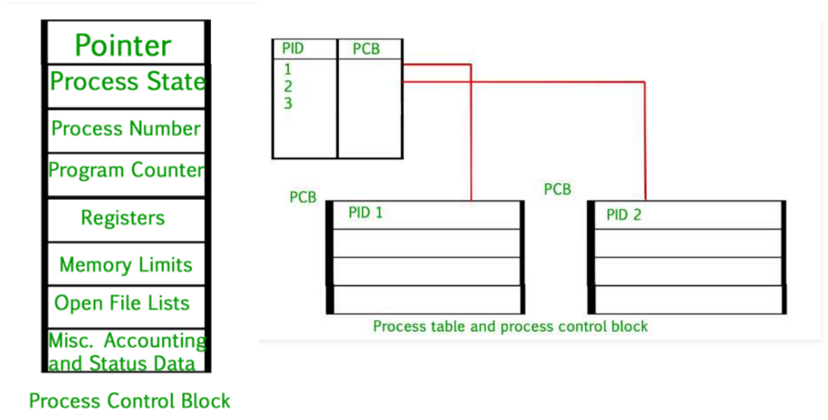
The PCB contains a huge amount of information in one large structure

- Process ID (PID)
- Execution state
- Hardware state: PC, SP, regs
- Memory management
- Scheduling
- Pointers for state queues
- Etc.

Some of the fields of a typical process table

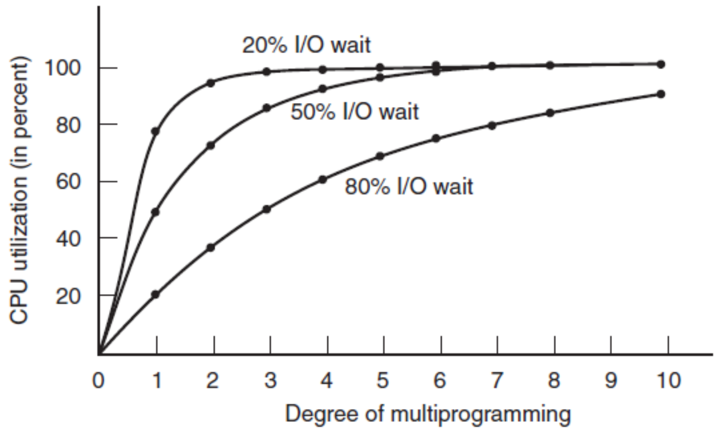
Process management	Memory management	File management
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

PCB and Process Table



https://www.geeksforgeeks.org/operating-system-process-table-process-control-block-pcb/

Modeling multiprogramming



References

- Andrew S. Tanenbaum, Herbert Bos, Modern Operating 4th edition, Pearson, 2015
- Presentations by Ding Yuan, ECE Dept., University of Toronto
- http://www.eexploria.com/program-to-implement-system-calls-using-fork-function/
- https://unix.stackexchange.com/questions/18474/what-does-this-process-stat-indicates
- https://www.howtogeek.com/119815/htg-explains-what-is-a-zombie-process-on-linux/
- http://turnoff.us/geek/zombie-processes/
- https://www.geeksforgeeks.org/operating-system-process-table-process-control-block-pcb/
- https://www.softprayog.in/programming/creating-processes-with-fork-and-exec-in-linux