Processes in Unix, Linux, and Windows

CS-3013 Operating Systems
Hugh C. Lauer

(Slides include materials from Modern Operating Systems, 3rd ed., by Andrew Tanenbaum and from Operating System Concepts, 7th ed., by Silbershatz, Galvin, & Gagne)
Processes in Unix, Linux, and Windows

- In previous topic, we used "process" in a generic way to represent the abstraction of concurrency.
- Unix pre-empted generic term "process" to mean something very specific.
- Linux and Windows adopted Unix definition.
Process in Unix-Linux-Windows comprises

- an *address space* – usually protected and virtual – mapped into memory
- the *code* for the running program
- the *data* for the running program
- an *execution stack* and *stack pointer* (SP); also *heap*
- the *program counter* (PC)
- a set of processor *registers* – general purpose and status
- a set of system *resources*
  - files, network connections, pipes, …
  - privileges, (human) user association, …
- …
Reading Assignment

- Tanenbaum, §2.1
  - Also §10.3.1, 10.3.2
Process Address Space (traditional Unix-Linux-Windows)

0xFFFFFFFF

(Virtual) address space

0x00000000

<table>
<thead>
<tr>
<th>stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dynamically allocated)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>heap</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dynamically allocated)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>static data</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>program code</th>
</tr>
</thead>
<tbody>
<tr>
<td>(text)</td>
</tr>
</tbody>
</table>

PC  SP
Processes in the OS – Representation

- To users (and other processes) process is identified by its Process ID (PID)
- In the OS, processes are represented by entries in a Process Table (PT)
  - PID is index to (or pointer to) a PT entry
  - PT entry = Process Control Block (PCB)
- PCB is a large data structure that contains or points to all info about the process
  - Linux – defined in task_struct (over 70 fields)
    - see include/linux/sched.h
  - Windows XP – defined in EPROCESS – about 60 fields
Processes in the OS – PCB

• Typical PCB contains:
  – execution state
  – PC, SP & processor registers – stored when process is not in *running* state
  – memory management info
  – privileges and owner info
  – scheduling priority
  – resource info
  – accounting info
Process – starting and ending

- Processes are created
  - When the system boots
  - By the actions of another process (more later)
  - By the actions of a user
  - By the actions of a batch manager

- Processes terminate
  - Normally – exit
  - Voluntarily on an error
  - Involuntarily on an error
  - Terminated (killed) by action of
    - a user or
    - another process
Processes – States

- Process has an execution state
  - ready: waiting to be assigned to CPU
  - running: executing on the CPU
  - waiting: waiting for an event, e.g. I/O
Processes – State Queues

- The OS maintains a collection of process state queues
  - typically one queue for each state – e.g., ready, waiting, …
  - each PCB is put onto a queue according to its current state
  - as a process changes state, its PCB is unlinked from one queue, and linked to another
- Process state and the queues change in response to events – interrupts, traps
Processes – Privileges

- Users are given privileges by the system administrator.
- Privileges determine what *rights* a user has for an *object*.
  - Unix/Linux – *Read*|*Write*|*eXecute* by user, group and “other” (i.e., “world”)
  - WinNT – Access Control List
- Processes “inherit” privileges from user
  - or from creating process
Process Creation – Unix & Linux

• Create a new (child) process – `fork();`
  – Allocates new PCB
  – Clones the calling process (almost exactly)
    • Copy of parent process address space
    • Copies resources in kernel (e.g. files)
  – Places new PCB on *Ready queue*
  – Return values from `fork()` call
    • 0 for child
    • child PID for parent
Example of fork()

```c
int main(int argc, char **argv)
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s sees PID of %d\n", name, child_pid);
        return 0;
    } else {
        printf("I am the parent %s. My child is %d\n", name, child_pid);
        return 0;
    }
}
```

% ./forktest
Child of forktest sees PID of 0
I am the parent forktest. My child is 486
Result – Two identical processes

```c
int main(int argc, char **argv)
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s sees PID\n" "%d\n", name, child_pid);
        return 0;
    } else {
        printf("I am the parent %s. " "My child is %d\n", name, child_pid);
        return 0;
    }
}
```

```c
int main(int argc, char **argv)
{
    char *name = argv[0];
    int child_pid = fork();
    if (child_pid == 0) {
        printf("Child of %s sees PID\n" "%d\n", name, child_pid);
        return 0;
    } else {
        printf("I am the parent %s. " "My child is %d\n", name, child_pid);
        return 0;
    }
}
```

One Difference
Starting New Programs

- Unix & Linux:
  - `int exec (char *prog, char **argv)`
  - Check privileges and file type
  - Loads program at path `prog` into address space
    - Replacing previous contents!
    - Execution starts at `main()`
  - Initializes context – e.g. passes arguments
    - `*argv`
  - Place PCB on `ready queue`
  - Preserves, pipes, open files, privileges, etc.
Executing a New Program (Linux-Unix)

• `fork()` followed by `exec()`

• Creates a new process as clone of previous one
  • I.e., same program, but different execution of it

• First thing that clone does is to replace itself with new program
Fork + Exec – shell-like

```c
int main(int argc, char **argv)
{
    char *argvNew[5];
    int pid;
    if ((pid = fork()) < 0) {
        printf("Fork error\n");
        exit(1);
    } else if (pid == 0) { /* child process */
        argvNew[0] = "/bin/ls"; /* i.e., the new program */
        argvNew[1] = "-l";
        if (execve(argvNew[0], argvNew, environ) < 0) {
            printf("Execve error\n");
            exit(1); /* program should not reach this point */
        }
    } else { /* parent */
        wait(pid); /* wait for the child to finish */
    }
}
```
Waiting for a Process

- Multiple variations of *wait* function
  - Including non-blocking *wait* functions
- Waits until child process terminates
  - Acquires termination code from child
  - Child process is destroyed by kernel
- **Zombie:** a process that had never been *waited* for
  - Hence, cannot go away!
  - See Tanenbaum, §10.3.2
Processes – Windows

- Windows NT/XP – combines `fork` & `exec`
  - `CreateProcess(10 arguments)`
  - Not a parent child relationship
  - *Note* – privileges required to create a new process

- See Tanenbaum, §11.4
  - (More in this section than we have discussed so far)
Traditional Unix

- **Processes** are in *separate* address spaces
  - By default, no shared memory
- **Processes** are unit of scheduling
  - A process is *ready*, *waiting*, or *running*
- **Processes** are unit of resource allocation
  - Files, I/O, memory, privileges, ...
- **Processes** are used for (almost) everything!
Windows and Linux

- *Threads* (next topic) are units of scheduling
- Threads are used for everything
Non-Traditional Unix
I.e., iPhone, Android, etc.

- Processes are in *separate* address spaces
  - By default, no shared memory
- **Processes** are unit of scheduling
  - A process is *ready*, *waiting*, or *running*
- Processes are unit of resource allocation
  - Files, I/O, memory, privileges, ...
- Processes are used for (almost) everything!
Reading Assignment

• Tanenbaum, §2.1 & §10.3.1–10.3.2
Questions?