Introduction to Processes

CS 537 - Introduction to Operating Systems

Definition

- A process is a program in execution
- It is not the program itself
  - a program is just text
- Only one process can run on a processor at once

Process Description

- A process is completely defined by
  - the CPU registers
    - program counter, stack pointer, control, general purpose, etc.
  - memory regions
    - user and kernel stacks
    - code
    - heap
- To start and stop a program, all of the above must be saved or restored
  - CPU registers must be explicitly saved/restored
  - memory regions are implicitly saved/restored
Memory Regions of a Process

- Every process has 3 main regions
  - text area
    - stores the actual program code
    - static in size (usually)
  - stack area
    - stores local data
      - function parameters, local variables, return address
    - data area (heap)
      - stores program data not on the stack
      - grows dynamically per user requests

Memory Regions of a Process

![Diagram showing process address space with stack region, unused region, data region, and text region]

Note: The stack usually grows down while the data region grows upward — the area in between is free.

User vs. Kernel Stack

- Each process gets its own user stack
  - resides in user space
  - manipulated by the process itself
- In Linux, each process gets its own kernel stack
  - resides in kernel space
  - manipulated by the operating system
  - used by the OS to handle system calls and interrupts that occur while the process is running
User Stack

- Function pointer
- Source code address
- Frame pointer
- Local frame
- Function check
- Return value call first
- Return from local frame
- Local address
- Return from main
- Local stack
- Local stack pointer

Kernel Stack

- Function call register
- Return value call first
- Pointer
- Local address
- Function in kernel
- Return from program
- Pointer mode
- Local address
- User program counter
- User stack pointer

Process Descriptor

- OS data structure that holds all necessary information for a process
  - process state
  - CPU registers
  - memory regions
  - pointers for lists (queues)
  - etc.
### Process Descriptor

<table>
<thead>
<tr>
<th>Pointer</th>
<th>State</th>
</tr>
</thead>
<tbody>
<tr>
<td>process ID number</td>
<td>current state the process is in (i.e. running)</td>
</tr>
<tr>
<td>program counter</td>
<td>Process ID Number</td>
</tr>
<tr>
<td>registers</td>
<td>identifies the current process</td>
</tr>
<tr>
<td>memory regions</td>
<td>Program Counter</td>
</tr>
<tr>
<td>list of open files</td>
<td>needed to restart a process from where it was interrupted</td>
</tr>
</tbody>
</table>

- Pointer
  - used to maintain queues that are linked lists

- State
  - current state the process is in (i.e. running)

- Process ID Number
  - identifies the current process

- Program Counter
  - needed to restart a process from where it was interrupted

### Process Descriptor

- Registers
  - completely define state of process on a CPU

- Memory Limits
  - define the range of legal addresses for a process

- List of Open Files
  - pretty self explanatory
Process States

- 5 generic states for processes
  - new
  - ready
  - running
  - waiting
  - terminated (zombie)
- Many OS’s combine ready and running into runnable state

Process Queues

- Every process belongs to some queue
  - implemented as linked list
  - use the pointer field in the process descriptor
- Ready queue
  - list of jobs that are ready to run
- Waiting queues
  - any job that is not ready to run is waiting on some event
  - I/O, semaphores, communications etc.
  - each of these events gets its own queue
- Queue management and ordering can be important
  - more on this later
Creating Processes

- Parent process creates a child process
  - results in a tree
- Execution options
  - parent and child execute concurrently
  - parent waits for child to terminate
- Address space options
  - child gets its own memory
  - child gets a subset of parent's memory

Creating Processes in Unix

- fork() system call
  - creates exact copy of parent
  - only thing different is return address
    - child gets 0
    - parent gets child ID
  - child may be a heavyweight process
    - has its own address space
    - runs concurrently with parent
  - child may be a lightweight process
    - shares address space with parent (and siblings)
    - still has its own execution context and runs concurrently with parent

Creating Processes in Unix

- exec() system call starts new program
  - needed to get child to do something new
  - remember, child is exact copy of parent
- wait() system call forces parent to suspend until child completes
- exit() system call terminates a process
  - places it into zombie state
Creating Processes in Unix

```c
void main() {
    int fd;
    pid = fork();
    if (pid == 0) {  // child process - start a new program
        execlp("ls", "ls", "-l", NULL);
    } else {  // parent process - wait for child
        wait(NULL);
        exit(0);
    }
}
```

Destroying a Process

- Multiple ways for a process to get destroyed
  - process issues an exit() call
  - parent process issues a kill() call
  - process receives a terminate signal
    - did something illegal
- On death:
  - reclaim all of process's memory regions
  - make process unrunnable
  - put the process in the zombie state
  - However, do not remove its process descriptor from the list of processes

Zombie State

- Why keep process descriptor around?
  - parent may be waiting for child to terminate
    - via the wait() system call
  - parent needs to get the exit code of the child
    - this information is stored in the descriptor
  - if descriptor was destroyed immediately, this information could not be gotten
  - after getting this information, the process descriptor can be removed
    - no more remnants of the process
init Process

- This is one of the first processes spawned by the OS
  - is an ancestor to all other processes
- Runs in the background and does clean-up
  - looks for zombie's whose parents have not issued a wait()
    - removes them from the system
  - looks for processes whose parents have died
    - adopts them as its own