Process Description and Control

Chapter 3 of [OS4e],
Chapters 4 and 20 of [OSC]:
- Process Concept
- Process States
- Processes in UNIX

Process Concept

Process
- term coined in the 1960’s by the designers of MULTICS
- unit of work (job) in multiprogramming and time-sharing systems
- program in execution (its “animated spirit”)
N.B multiple processes may be associated with same program

Concurrent processes
- protected from each other (restricted access to memory)
- cooperate by reading/writing shared memory
- communicate by OS message-passing service
Process States

How can multiple programs be “in execution” on single processor?
- at most one is actually “running”
- other processes are queued
- running process can be interrupted
- OS selects next process to run (*short-term scheduling*) and dispatches it

(a) State transition diagram

(a) Queuing diagram
Recall that handling an interrupt involves
- *mode switch*, but overhead can be small if control returns to interrupted process

On the other hand, if OS schedules another process in place of interrupted process,
- system information about both processes must be updated
- (on many systems) memory-management hardware must be notified

This process switch (*context switch*) can be performance bottleneck!
Trace

■ sequence of instructions being executed

<table>
<thead>
<tr>
<th>5000</th>
<th>8000</th>
<th>12000</th>
</tr>
</thead>
<tbody>
<tr>
<td>5001</td>
<td>8001</td>
<td>12001</td>
</tr>
<tr>
<td>5002</td>
<td>8002</td>
<td>12002</td>
</tr>
<tr>
<td>5003</td>
<td>8003</td>
<td>12003</td>
</tr>
<tr>
<td>5004</td>
<td>8004</td>
<td>12004</td>
</tr>
<tr>
<td>5005</td>
<td>8005</td>
<td>12005</td>
</tr>
<tr>
<td>5006</td>
<td>8006</td>
<td>12006</td>
</tr>
<tr>
<td>5007</td>
<td>8007</td>
<td>12007</td>
</tr>
<tr>
<td>5008</td>
<td>8008</td>
<td>12008</td>
</tr>
<tr>
<td>5009</td>
<td>8009</td>
<td>12009</td>
</tr>
<tr>
<td>5010</td>
<td>8010</td>
<td>12010</td>
</tr>
<tr>
<td>5011</td>
<td>8011</td>
<td>12011</td>
</tr>
</tbody>
</table>

(a) Trace of Process A (b) Trace of Process B (c) Trace of Process C

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.2 | Traces of Processes of Figure 3.1

Traces of individual processes (and OS) are interleaved

N.B. number of instructions executed by processes and dispatcher are unrealistically low in this example
Not-running =
- ready
- blocked (waiting for completion of I/O)
  e.g. process B blocked after instruction at address 8003 executed

Since only ready process should be dispatched, it makes sense to use several queues
- short-term scheduler only inspects *ready queue*
  (For prioritized scheduling, could have one ready queue per priority level.)
Process created when

- long-term scheduler selects from job queue (in batch-processing system)
- user logs on (in time-sharing systems)
- OS called to invoke system program that can execute concurrently with caller
  e.g., to handle printing of a file
- create-process system call invoked by user program
  e.g., `fork` in UNIX allows parent process to spawn child process; child executes same program, but has different process identifier; child has to make `exec` system call to load and run new program

Creation involves OS

- updating tables in kernel and allocating memory for process image
  - sufficient resources available
Process termination occurs when
- halt instruction executed
- user logs off
- process invokes exit system call (so informing parent)
- unrecoverable error occurs
- another process kills it

Termination involves
- utility programs extracting information
- data structures being deleted
  - in UNIX, process enters zombie state until parent has picked up byte of status information using wait system call
- long-term scheduler possibly being invoked to try to maintain degree of multiprogramming

![Five-State Process Model](image)
Short-term scheduler is invoked frequently
- so must be fast
  e.g., if it takes 10ms to decide to execute a process and that process only runs for 90ms, then what percentage of processor time is being wasted?
  \[
  \frac{10}{90+10} \times 100\% = 10\%
  \]
Long-term scheduler is invoked less frequently
- so should carefully select which job to admit
  i.e., good mix of CPU-bound and I/O-bound jobs

Recall that for processor to execute program, (all or some of) instructions and data must be in main memory

Problem: Running and ready processes may be allocated less memory space than desirable

Solution: middle-term scheduling
- swap process out to disk, e.g., when blocked
- swapped out process is considered suspended
- memory is released for use by another process
Processes in UNIX

**system process**
- program executing in supervisor mode
- for protection, has its own stack
- runs without preemption until it chooses to block
N.B. system *programs* execute in user mode!

**user process**
- program executing in user mode
- system call or interrupt causes user process to temporarily become system process
  - OS is said to execute within user process
system initialization
- kernel is loaded (*booting*)
- (system) process 0 is created and dispatched, it spawns (system) process 1 (*init*) and becomes *swapper* process
- *init* spawns one (user) process per terminal channel to set up the terminal, login a user, and interpret commands
- *init* waits for any child to terminate, then recreates it

N.B. All user processes are descendants of *init*

---

**execution environment (process image)**
- read-only *text segment* (program code)
- *user data segment* (global variables, heap, runtime stack)
  - size may be extended by system call
  - copied to secondary storage when process swaps out
- *system data segment* (saved processor registers, system stack, ...)
  - not addressable in user mode
  - copied to secondary storage when process swaps out
- kernel
  - not addressable in user mode
  - includes *page table* and *text table*

An entry in process table
- is allocated when process is created and freed when process terminates
- records identifier of process, its parent and user associated with it
- maintains process control information, including
  - state, priority, time waiting/executing (to assist scheduling), event waiting on
  - pointers to other entries (so implementing queue), data segments and text table
Each entry in *text table* maintains

- count of number of processes sharing text segment (i.e. executing the same code)
  - When count is zero, entry is freed along with any main and secondary memory holding the segment
- location of text segment in main memory (if loaded)
- location of text segment in secondary memory
  - No need to copy there when process swaps out

![Diagram](image-url)  

*Fig. 1—Process control data structure.*