Lecture 6:

Processes I: The Basics

www.cl.cam.ac.uk/Teaching/2001/OSFounds/

Today’s Lecture

Today we’ll cover:

- What is a process?
  - Concepts.
  - Process states.
- What OS support do we need?
  - Process control blocks.
  - Context switching.
  - Scheduling.
- The life of a process

Process Concept

- From a user’s point of view, the operating system is there to execute programs:
  - on batch system, refer to jobs
  - on interactive system, refer to processes
    (we’ll use both terms fairly interchangeably)
- Process ≠ Program:
  - a program is static, while a process is dynamic
  - in fact, a process ≜ “a program in execution”
- (Note: “program” here is pretty low level, i.e. native machine code or executable)
- Process includes:
  1. program counter
  2. stack
  3. data section
- Processes execute on virtual processors

Process States

- As a process executes, it changes state:
  - New: the process is being created
  - Running: instructions are being executed
  - Ready: the process is waiting for the CPU (and is prepared to run at any time)
  - Blocked: the process is waiting for some event to occur (and cannot run until it does)
  - Exit: the process has finished execution.
- The operating system is responsible for maintaining the state of each process.
Process Control Block

OS maintains information about every process in a data structure called a **process control block** (PCB):

- Unique process identifier
- Process state (Running, Ready, etc.)
- CPU scheduling & accounting information
- Program counter & CPU registers
- Memory management information
- . . .

**Process Context** = machine environment during the time the process is actively using the CPU, i.e. context includes program counter, general purpose registers, processor status register, . . .

To switch between processes, the OS must:

a) **save** the context of the currently executing process (if any), and

b) **restore** the context of that being resumed.

Time taken depends on h/w support.

Scheduling Queues

- *Job Queue*: batch processes awaiting admission.
- *Ready Queue*: set of all processes residing in main memory, ready and waiting to execute.
- *Wait Queue(s)*: set of processes waiting for an I/O device (or for other processes)
- *Long-term & short-term* schedulers:
  - *Job scheduler* selects which processes should be brought into the ready queue.
  - *CPU scheduler* selects which process should be executed next and allocates CPU.

Process Creation

- Nearly all systems are **hierarchical**: parent processes create children processes.
- **Resource sharing**:
  - parent and children share all resources.
  - children share subset of parent’s resources.
  - parent and child share no resources.
- **Execution**:
  - parent and children execute concurrently.
  - parent waits until children terminate.
- **Address space**:
  - child duplicate of parent.
  - child has a program loaded into it.
- e.g. Unix:
  - `fork()` system call creates a new process
  - all resources shared (child is a clone).
  - `exec1p()` system call used to replace the process’ memory space with a new program.
- NT/2000: `CreateProcess()` system call includes name of program to be executed.
#include <stdio.h>

void main(int argc, char *argv[]) {
    int pid;
    pid = fork();
    if (pid < 0) {
        fprintf(stderr, "Fork failed");
        exit(-1);
    } else if (pid == 0) {
        execlp("/bin/ls", "ls", NULL);
    } else {
        wait(NULL);
        printf("Child complete");
        exit(0);
    }
}

Process Termination

- Process executes last statement and asks the operating system to delete it (exit):
  - return data from child to parent (wait)
  - process' resources are deallocated by the OS.
- Process performs an illegal operation, e.g.
  - makes an attempt to access memory to which it is not authorised,
  - attempts to execute a privileged instruction
- Parent may terminate execution of child processes (abort, kill), e.g. because
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - parent is exiting ("cascading termination")
    (many operating systems do not allow a child to continue if its parent terminates)
- e.g. Unix has wait(), exit() and kill()
- e.g. NT/2000 has ExitProcess() for self and TerminateProcess() for others.

Process Blocking

- In general a process blocks on an event, e.g.
  - an I/O device completes an operation,
  - another process sends a message
- Assume OS provides some kind of general-purpose blocking primitive, e.g. await().
- Need care handling concurrency issues, e.g.
  
  ```c
  if(!no key being pressed) {
    await(keypress);
    printf("Key has been pressed!\n");
  } // handle keyboard input
  ```
  
  What happens if a key is pressed at the first '}'?
  - (This is a big area.)
  - In this course we'll generally assume that problems of this sort do not arise.

Summary

You should now understand:

- What a process is.
- Process states and PCBs.
- Scheduling queues.
- Stages of Process lifecycle.

Next lecture: Processes II: CPU scheduling

Background Reading:

- Silberschatz et al.: Chapter 4