Lecture 4 (Processes & threads)

Process - abstraction of running program

Processes support the ability to have (pseudo) concurrent operation even when there is only one CPU available. In any multiprogramming system, the CPU switches from process to process quickly. At any one instant the CPU is running **only one process.** Illusion of parallelism is called **pseudo-parallelism**

All the runnable software on the computer, sometimes including the OS, is organized into a number of
(**sequential**) **processes.**

A process is an **instance of an executing program** including the current values of the program counter, registers, and variables.

Conceptually, each process has its own **virtual CPU**. In reality, the real CPU switches between different processes. Such a switching is called **multiprogramming.**

There is only **one physical program counter,** so when each process runs, its logical program counter is loaded into the real program counter. When it is finished, the physical program counter is saved in the process' stored logical program counter in memory.

Each of the process has its own flow of control (its own logical program counter) and runs independently of the other ones. All the processes have made progress, but at any given instant **only one process is actually running.**

Program - set of instructions.

Process - activity of some kind. It has program, input, output, state.

If some program runs 2 times, it counts as 2 **distinct processes.**

4 principles that cause a process creation:

1. System initialization (background & foreground processes when booting the PC)
2. Execution of process creation system call by a running a process (to help processes handle their job)

3. User request to create process (in interactive systems)
4. Initiation of batch job (jobs that can run without end user interaction, or can be scheduled to run as resources permit)

Processes that stay in the background to handle some activity such as email are called **daemons**.

Fork - create an **exact clone** of the calling process. Parent and child processes have the same **memory image** (memory image is simply a copy of the process's virtual memory, saved in a file), **environmental strings** and **open files.** Parent and child have their own distinct address spaces. Alternatively, the child may share all of the parent's memory, but in that case the
memory is shared **copy-on-write**, which means that whenever either of the two wants to modify part of the memory, that chunk of memory is explicitly copied first to make sure the modification occurs in a private memory area. NO WRITABLE MEMORY IS SHARED BETWEEN PARENT & CHILD. Child process then executes **execve** or a similar system call to change its memory image and run a new program. Child can manipulate its file descriptors **after the fork but before the execve** in order to accomplish redirection of standard input, standard output, and standard error.

Process termination conditions:

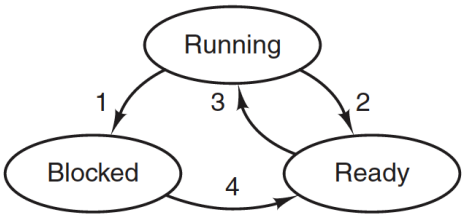
1. Normal exit (voluntary) - work is done
2. Error exit (voluntary) - example: no such file exist. Given incorrect parameters.
3. Fatal error (involuntary) - bug in program (dividing by zero, unallocated memory reference)
4. Killed by another process(involuntary) - if another process execute syscall kill

Parent process and child process are continue to be associated with each other. The child can create its own processes forming **process hierarchy**. Process and all its over descendants form a **process group**.

3 states of process:

1. Running (using a CPU at that instant)
2. Ready (runnable, temporary stops and let another process run)

3. Blocked (unable to run until some external event happens) (happens because its logically cannot continue)



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

Transitions:

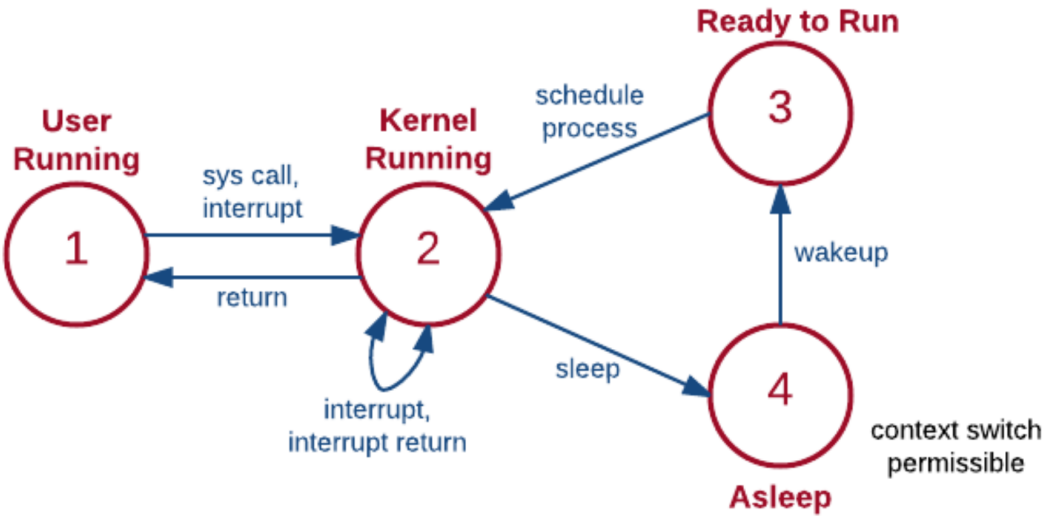
1. **Running → Blocked** (OS discovers that process cannot continue; the process executes a system call such as **pause**; process is blocked automatically in UNIX)
2. **Running → Ready** (scheduler decides to let another processor to have CPU time)
3. **Ready → Running** (scheduler decides to continue this process because all other processor run long enough)
4. **Blocked → Ready** (external event for which process wait happen)

Blocked → Ready → Running (if no other process is running at that moment, transition 3 will be triggered and the process will start running)

The **lowest layer** of a process-structured operating system handles **interrupts** and **scheduling**. Above that layer are **sequential processes**.

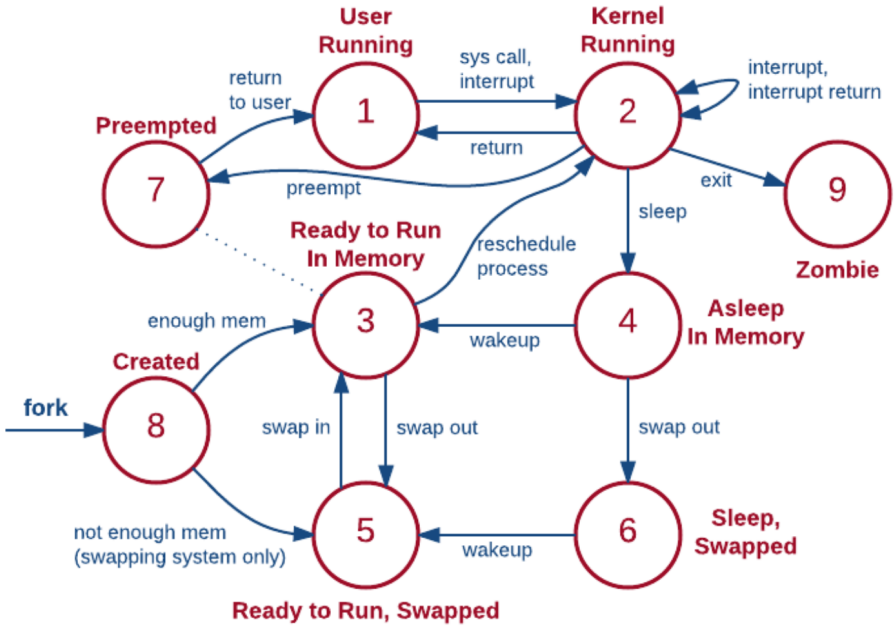
Deeper view of process states:

1. Process is executing in user mode
2. Process is executing in kernel mode
3. Process is not executing but ready
4. Process is sleeping (wait external event)



Complete list:

1. Process is executing in user mode
2. Process is executing in kernel mode
3. Process is not executing but ready
4. Process is sleeping (wait external event)
5. Process is ready but needs to swap into main memory
6. Process is awaiting event and swapped into secondary memory
7. Process is returning from kernel to user and kernel preempts it and does a context switch
   to schedule another process
8. Process is newly created (not ready nor sleeping)
9. Process executed the exit system call and no longer exist but it leaves a record for its parent process to collect (the final state of a process)



Process executing in **kernel** mode can be **preempted** only when it is about to return to user mode

The OS maintains a table (an array of structures), called the **process table**, with one entry per process
(sometimes called a p**rocess control block**)

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		

Figure 2-4. Some of the fields of a typical process-table entry.

Associated with each I/O class is a location called the **interrupt vector**, which contains the address of the interrupt service procedure.

Operating system does when an interrupt occurs:

1. Hardware stacks program counter, etc.
2. Hardware loads new program counter from interrupt vector.
3. Assembly language procedure saves registers.
4. Assembly language procedure sets up new stack.
5. C interrupt service runs (typically reads and buffers input).
6. Scheduler decides which process is to run next.
7. C procedure returns to the assembly code.
8. Assembly language procedure starts up new current process.

CPU utilization = 1 - p^n, (p - fraction of waiting for I/O to complete, n - number of processors at once)

The CPU utilization as a function of n is called the **degree of multiprogramming.**

Multiprogramming - switching from program to program

In traditional OSs each process has an address space and a **single thread of control**

Reasons for having these **miniprocesses**, called **threads:**

1. Run programs in **quasi-parallel** - simultaneous running of multiple threads
2. Faster to create/destroy processes
3. Allows I/O and computing overlap
4. Threads with multiple CPU provide real parallelism

Server construction:

1. Threads (parallelism, blocking syscalls)
2. Single-threaded process (no parallelism, blocking syscalls)
3. Finite-state machine (parallelism, nonblocking syscalls, interrupts)

Classical Thread model:

Process model is based on two independent concepts: **resource grouping** and **execution**

Process has an address space containing program text and data, as well as other resources (open files, child processes, pending alarms, signal handlers, accounting information...)

Thread of execution or just thread has:

- program counter (which instruction execute next)
- registers (hold current work variables)
- stack (contains execution history)

Processes are used to group resources together; threads are the entities scheduled for execution on the CPU.

Threads == **lightweight processes**

Multithreading - allowing multiple threads in the same process. CPU have direct hardware support for multithreading.

Items shared by all threads in the process Private items for each thread

Per process items	**Per thread items**
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

States of thread:

1. Running (thread has a CPU and is active)
2. Blocked (waiting for some event)
3. Ready (thread is scheduled and wait its turn)
4. Terminated

Each thread has its **own stack** that contains frame for each procedure but not yet returned. **Frame** contains the procedure's local variables and return address to

use when procedure call is finished.

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

The code placed around the system call to do the checking is called a **jacket** or **wrapper**.