CS 3214: Computer Systems
Lecture 3: Processes

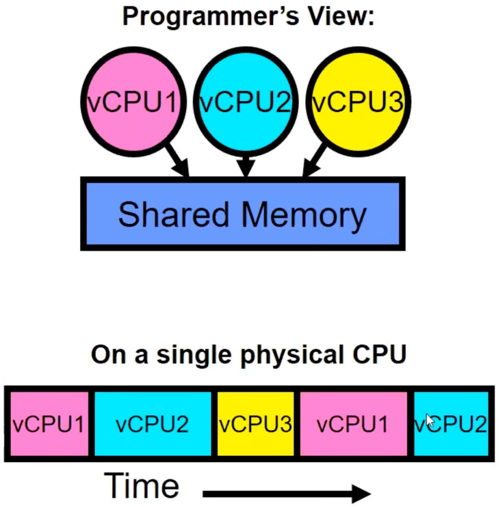
Instructor: Huaicheng Li

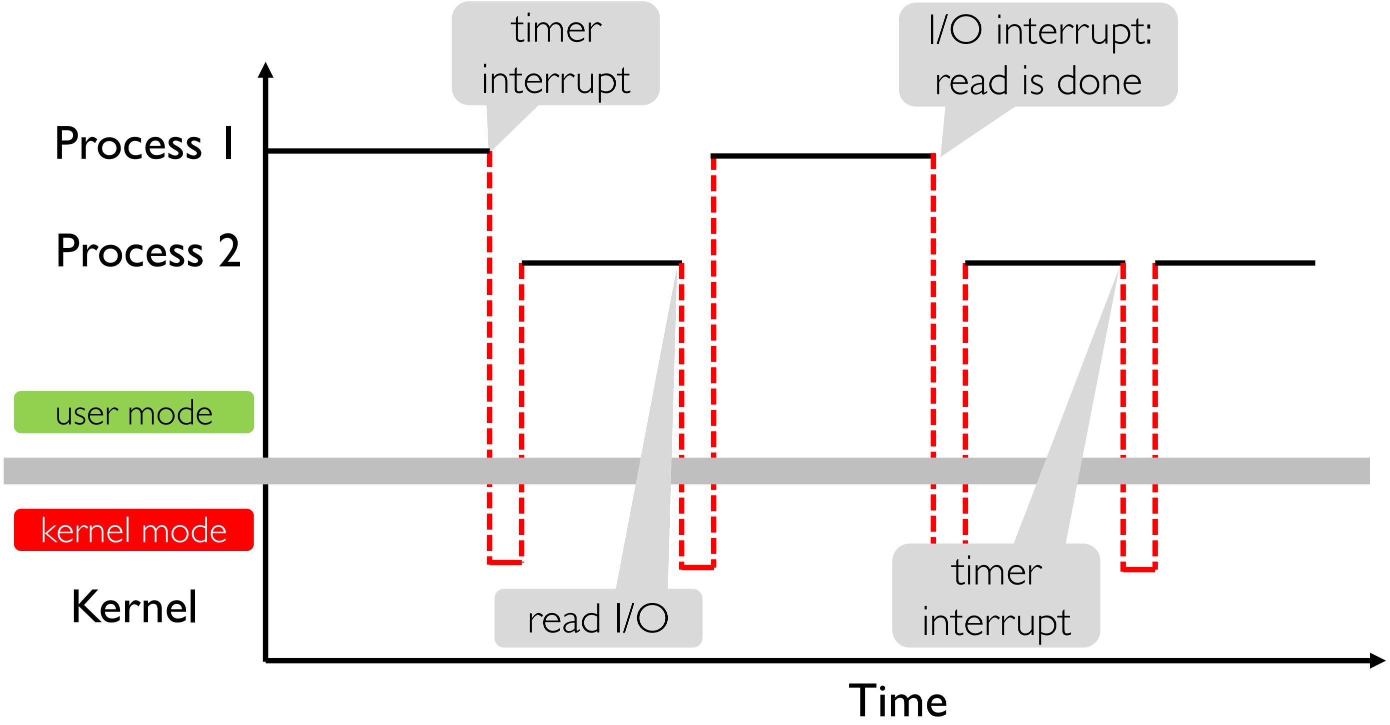
Sept 3 2024

VIRGINIA TECH™

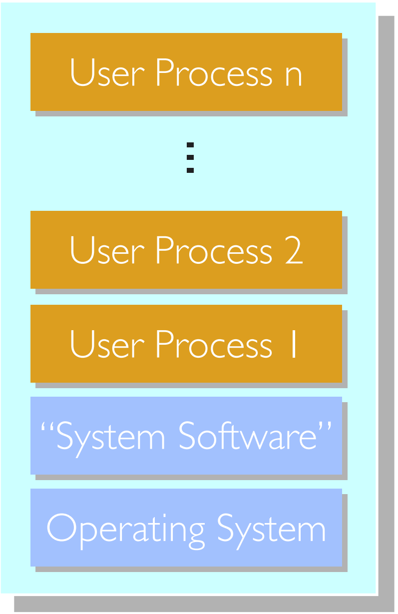
Recap: Process Primer

- Program → Process (an instance of running program)
- Context switch vs. dual-mode (user←→kernel) operations

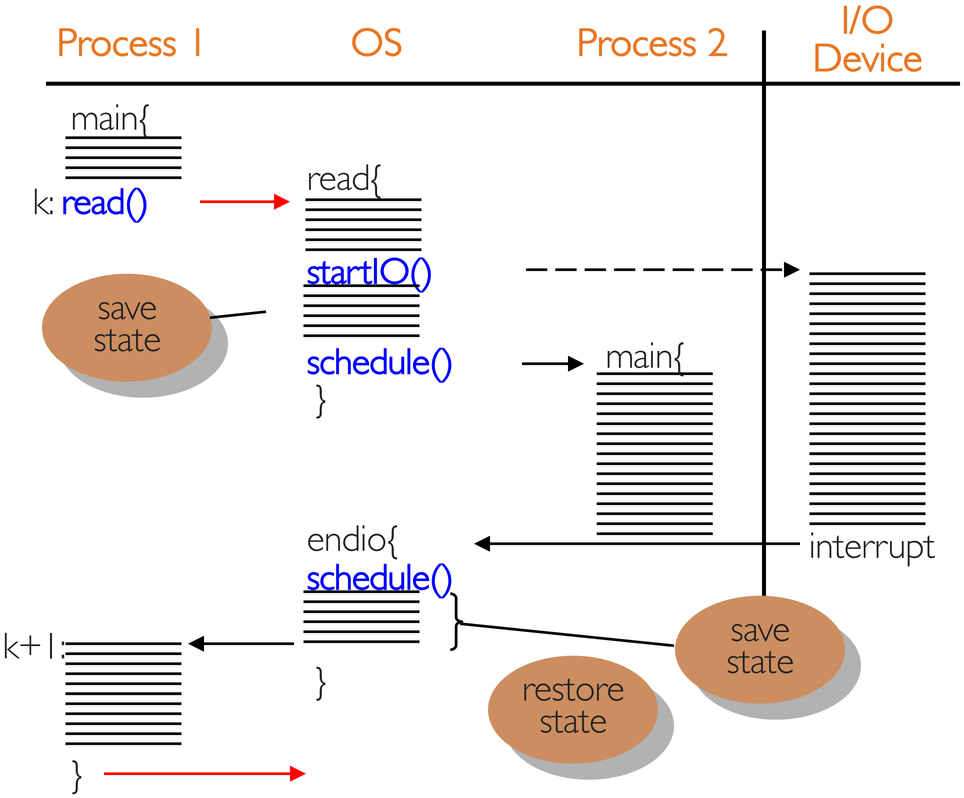




Process Contexts



Memory

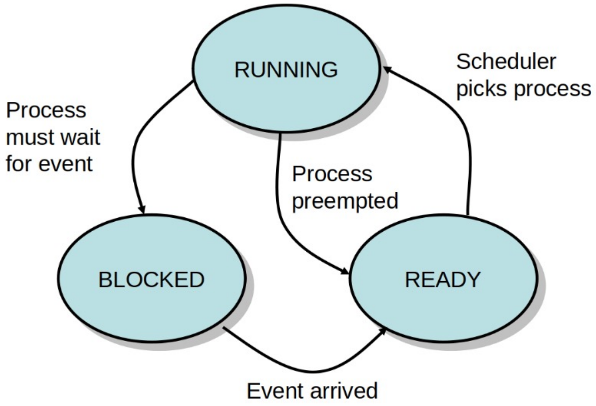


Today's Topics: Process Management

- Process state management
- Process scheduling (not in-depth)
- Process user interface

Process States

- Running: executing its instructions on CPUs
- Ready: ready to execute but waiting for its turn
- Blocked: stopped due to external events, cannot make use of CPUs even if some are available
- Running → Blocked
  - waiting for: input, exclusion access to a lock, signal, sleep(2s), child process
- Blocked → Ready
  - The waiting is now over!
  - OS adds the process to a ready **queue**
- Ready → Running (CPU scheduling)
  - 1 process per CPU, scheduling policy
- Running → Ready
  - Process de-scheduled (yield or preempted)



Discussion Questions

1. What happens if an n CPU system has exactly n READY processes?
2. What happens if an n CPU system has 0 READY processes?
3. What happens if an n CPU system has k < n READY processes?
4. What happens if an n CPU system has 2n READY processes?
5. What happens if an n CPU system has m >> n READY processes?
6. What is a typical number of BLOCKED/READY/RUNNING processes in a system (e.g., your phone or laptop?)
7. How does the code you write influence the proportion of time your program spends in the READY/RUNNING state?
8. How can the number of processes in the READY/RUNNING state be used to measure CPU demand?
9. Assuming the same functionality is achieved, is it better to write code that causes a process to spend most of its time BLOCKED, or READY?

Answers (permuted order)

- Prefer BLOCKED to READY because it does not consume CPU; use OS facilities to wait for events rather than poll in a loop
- 150 – 500 BLOCKED, and 0 – 2 RUNNING
- Every process takes about twice as long as it normally would
- The load average is a weighted moving average of the size of the ready queue (including RUNNING processes); it says how many CPUs could be kept busy
- System becomes very laggy, processes take much longer than normal
- n – k CPUs are idle, k CPUs run exactly 1 process
- Each CPU runs exactly 1 process
- Performing computation without performing I/O means the process is READY at all times and will be RUNNING if scheduled.
- The system is idle and goes into a low-power mode

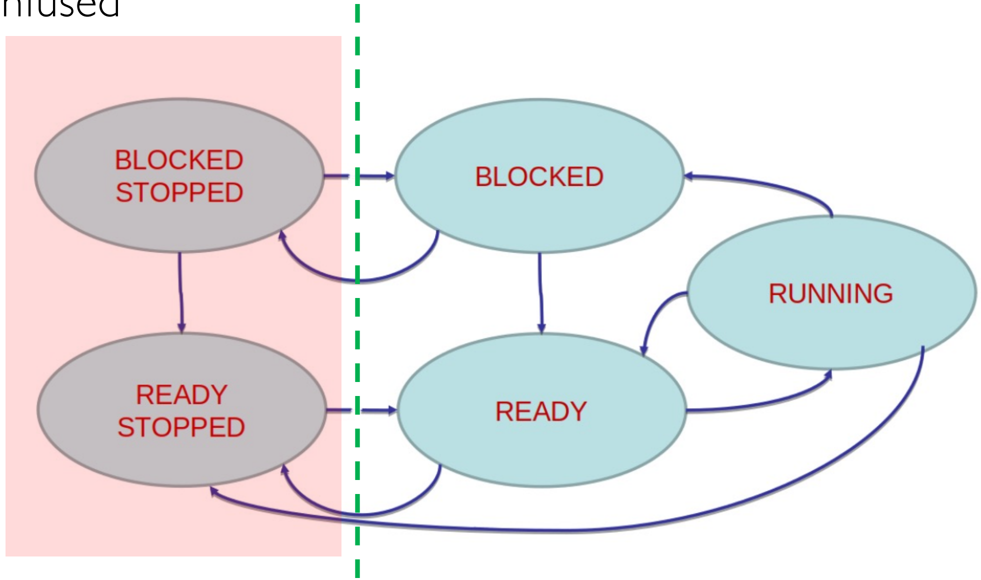
Process States in Linux

- Our model is simplified, real OS often maintain state diagrams with 5-15 states for their threads/tasks
  - Linux uses the following states
  - Command line tool: "ps"

D uninterruptible sleep (usually IO)
I Idle kernel thread
R running or runnable (on run queue)
S interruptible sleep (waiting for an event to complete)
T stopped by job control signal
t stopped by debugger during the tracing
X dead (should never be seen)
Z defunct ("zombie") process, terminated but not reaped by its parent

Process States and Job Control

- Job control: stop/suspend, and resume a process
  - Linux commands: *jobs, bg, fg,*
    - *Ctrl-Z: pause process in time until users tell it to continue*
- Job control and process states
  - Don't be confused



Programmer's View

- Process state transitions are guided by decisions or events outside the programmer's control (user actions, user input, I/O events, inter-process communication, synchronization) and/or decisions made by the OS (scheduling decisions)
- They may occur frequently, and over small time scales
  - e.g., on Linux preemption may occur every 4ms for RUNNING processes
  - when processes interact on shared resources (locks, pipes) they may frequently block/unblock)
- For all practical purposes, these transitions, and the resulting execution order, are unpredictable
- The resulting concurrency requires that programmers not make any assumptions about the order in which processes execute; rather, they must use signaling and synchronization facilities to coordinate any process interactions

CPU Scheduling

- Problem of choosing which process to run next
  - And for how long until the next process runs
- Why bother?
  - Improve performance: amortize context switching costs (fast switching)
  - Improve user experience: e.g., low latency keystrokes (timely)
  - Priorities: favor "important" work over background work (priorities)
  - Fairness
- Linux schedulers (for fun, read more by yourself)
  - CFS (completely fair scheduler)
  - EEVDF (since Linux 6.6, read here, based on a paper in 1995, here)
    - earliest eligible virtual deadline first scheduling

When does Scheduling Happen?

- When a process blocks
- When a device interrupts the CPU to indicate an event occurred (possibly un-blocking a process)
- When a process yields the CPU

- Preemptive scheduling: Setting a timer to interrupt the CPU after some time
  - Places an upper bound on how long a CPU-bound process can run without giving another process a turn
- Non-preemptive scheduling: Processes must explicitly yield the CPU

- OS uses process control blocks (PCBs) to represent a process
- Every resource is represented with a queue
- OS puts PCB on an appropriate queue
  - Ready-to-run queue
  - Blocked for IO queue (queue per device)
  - Zombie queue
- When CPU becomes available, choose from ready to run queue
- When an event occurs, remove waiting process from blocked queue, move to ready queue

Multi Processes in One Application

- e.g., Chrome browser
- Single process cannot overleap CPU and I/O

Progress Management

- OS provide APIs (system calls) to manage process
- Process creation
  - includes ways to set up new process's environment
- Process termination
  - Normal termination (exit(), return from main())
  - Abnormal termination (due to crash or outside intervention, "kill")
  - In either cases, OS cleans up (reclaims all memory, close file-descriptors)
- Process interaction; examples include
  - Waiting for a process to finish (wait())
  - Stopping/continuing a process
- Change a process's scheduling and other attributes
- OS provides facilities to be used by or in coordination with control programs (shell, GUI, task manager): Ctrl-C, Ctrl-Z

Create Processes

- Unix fork()/exec()
  - Child inherits everything, runs same program
  - Only difference is the return value from fork()
    - Child gets 0; parent gets child pid
- A separate exec() system call loads a new program
  - Like getting a brain transplant
- Some programs, like our web server example, fork() clones (without calling exec()).
  - Common case is probably fork+exec

Windows: Create Process

- OS provide APIs (system calls) to manage processes
- Example: CreateProcessA in Windows

```
BOOL CreateProcessA(
  LPCSTR                lpApplicationName,
  LPSTR                 lpCommandLine,
  LPSECURITY_ATTRIBUTES lpProcessAttributes,
  LPSECURITY_ATTRIBUTES lpThreadAttributes,
  BOOL                  bInheritHandles,
  DWORD                 dwCreationFlags,
  LPVOID                lpEnvironment,
  LPCSTR                lpCurrentDirectory,
  LPSTARTUPINFOA        lpStartupInfo,
  LPPROCESS_INFORMATION lpProcessInformation
);
```

- Creates ("spawns") a new process, and instruct it to run a new program with arguments and attributes

Linux/Unix Process Management

- Unix separate process creation from loading a new program
- The fork() system call creates a new process, but does not load a new program
- The newly created process is called a child process (creating process is parent)
  - Processes form a tree-like hierarchy
  - Child processes may inherit parts of their environment from their parents, but are otherwise distinct entities
- The child process then may change/set up the environment and, when ready, load a new program that replaces the current program but retains certain aspects of the environment (exec())
- The parent has the option of waiting for the child process to terminate, which is also called "joining" the child process
  - Parent can also learn how the child process terminated, e.g., the code that the child passed to exit()

exec()

- The exec() call allows a process to "load" a different program and start execution at main (actually _start).

- It allows a process to specify the number of arguments (argc) and the string argument array (argv).

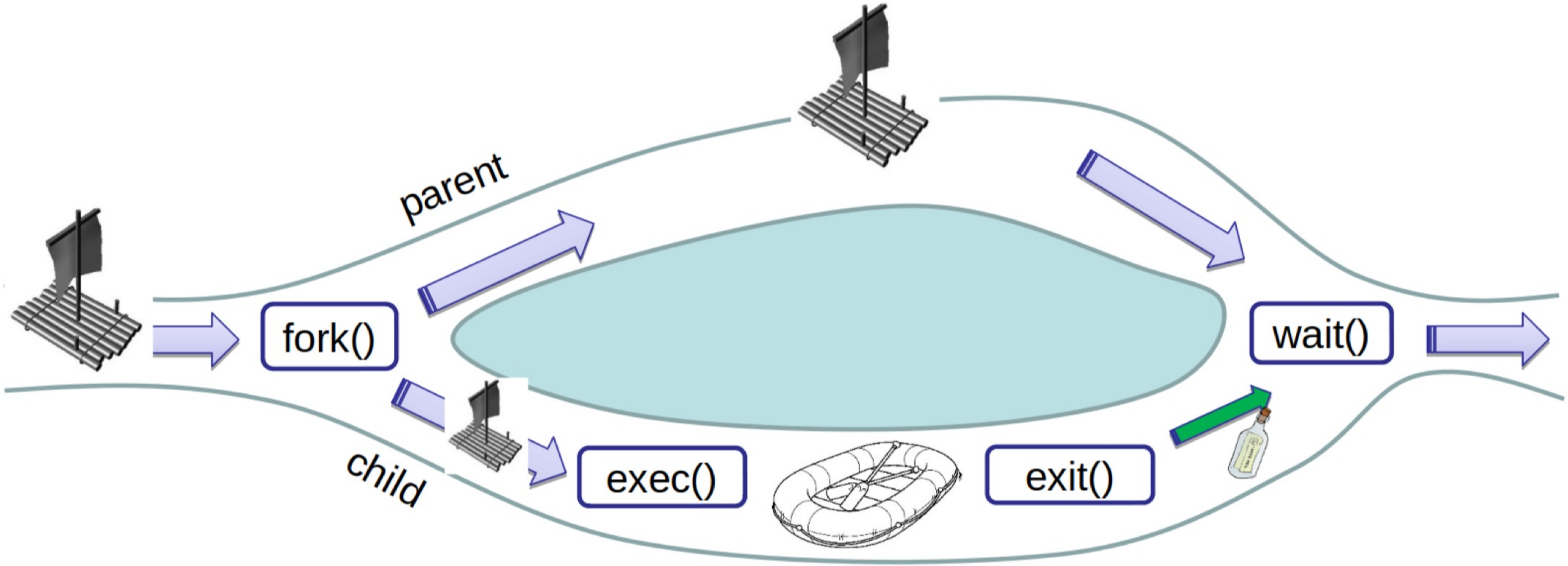
- If the call is successful
  - it is the same process …
  - but it runs a different program !!

- Code, stack & heap is overwritten
  - Sometimes memory mapped files are preserved.

exec() vs fork()

- exec()
  - Keeps process, but discards old program and loads a new program
  - Reinitializes process state (clears heap + stack, starts at new programs's main()); except it retains file descriptors
  - If successful, is called once but does not return
  - includes multiple variants (execvp(), etc)
- fork()
  - Keeps program and process, but also creates a new process
  - New process is a clone of the parent; child state is a (now separate) copy of parent's state, including everything: heap, stack, file descriptors
  - Called once, returns twice (once in parent, once in child)

fork/exec/exit/wait



fork() + exec() Example

In the parent process:

```
main()
…
int r =fork();                        // create a child
if (0 == r) {                         // child continues here
    exec_status = exec("calc", argc, argv0, argv1, …);
    printf("Something is horribly wrong\n");
    exit(exec_status);
} else {                              // parent continues here
   printf("Shall I be mother?");
   …
  child_status = wait(r);
}
```