# Processes (Part II)

Godmar Back

Virginia Tech

September 6, 2022

## Process States

OS's keep track of the status of each process.

- RUNNING:
  - This process is executing its instructions on a CPU
- READY:
  - This process is ready to execute on a CPU, but currently is not (it is waiting for a CPU to be assigned)
- BLOCKED:
  - This process is not ready to execute on a CPU, because it is waiting for some event
  - it cannot currently make use of a CPU even if one is available

NB: in systems whose kernel supports multi-threading, the states are maintained for each thread separately.

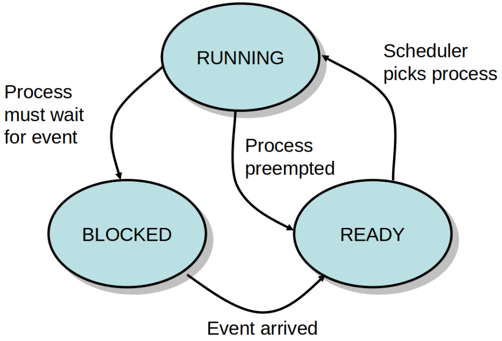

Figure 1: Basic Process State Diagram

## Process State Transitions

- RUNNING → BLOCKED: process cannot continue because it first must wait for something, e.g.
  - for input (keystroke, file from disk, network message, data from Unix pipe)
  - for exclusive access to a resource (acquire a lock)
  - for a signal from another thread/process
  - for time to pass (e.g., `sleep(2)` sys call)
  - for a child process to terminate
- BLOCKED → READY: process becomes ready when that something finally becomes available
  - OS adds process to a ready queue data structure
- READY → RUNNING: process is chosen by the scheduler
  - only 1 process can be chosen per CPU
  - requires scheduling policy if demands exceeds supply
- RUNNING → READY: process is descheduled
  - OS preempted the process to give another READY process a turn
  - or, rarely, process voluntarily yielded the CPU

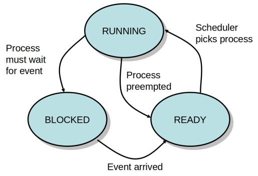

Figure 2: Basic Process State Diagram

## 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 $m \gg n$ 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 (in permuted order)

1. Prefer BLOCKED to READY because it does not consume CPU; use OS facilities to wait for events rather than poll in a loop
2. 150 – 500 BLOCKED, and 0 – 2 RUNNING
3. Every process takes about twice as long as it normally would
4. 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
5. System becomes very laggy, processes take much longer than normal
6. $n - k$ CPUs are idle, $k$ CPUs run exactly 1 process
7. Each CPU runs exactly 1 process
8. Performing computation without performing I/O means the process is READY at all times and will be RUNNING if scheduled.
9. The system is idle and goes into a low-power mode

## Process States in Linux and other OS

- Our model is simplified, real OS often maintain state diagrams with 5-15 states for their threads/tasks
- Case study: Linux uses the following states:

### Linux Process States

```
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
W    paging (not valid since the 2.6.xx kernel)
X    dead (should never be seen)
Z    defunct ("zombie") process, terminated but not reaped by its parent
```

### Thinking Question

Why does Linux not distinguish between RUNNING and READY?

## Process States and Job Control

- Job control: Some systems provide the ability to stop (suspend) a process for some time, and continue it later with all its state intact.
- E.g., in Linux Ctrl-Z
- This mechanism is separate from the state transitions caused by events processes wait for – events can still arrive for stopped processes

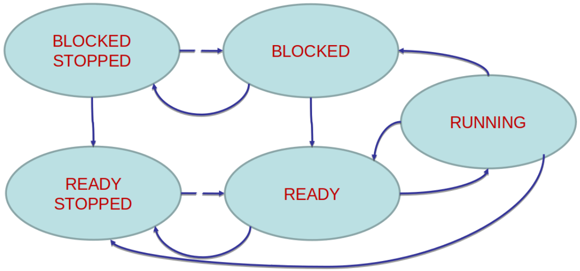

Figure 3: Extended State Diagram including Job Control (conceptually)

## Programmer's View

- Process state transitions are guided by decisions or events outside the programmer's control (user actions, user input, I/O events, interprocess 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

## Mini Glossary

A number of English verbs and gerunds are used with the respect to processes and job control that sometimes have an non-intuitive and/or context-dependent meaning

- *running* can mean
  - (laymen, informal): a running process is one that has been started but hasn't finished.
  - (more precise OS terminology): a process that is currently in the `RUNNING` state, making progress and consuming CPU time in the process
- *stopping* a process in Unix means to momentarily suspend it (independent of whether it's `RUNNING`, `READY`, or `BLOCKED`). Does not terminate the process - the process can be resumed ("continued") later.
- *interrupting* a process (usually with Ctrl-C) typically, by default, terminates (ends) the process (but not always). It does not suspend it. It is not related to (hardware) interrupts.
- *killing* a process means to send a signal to it, which often, but not always, terminates it.

## References