Processes

The fundamental activity of an operating system is the *creation*, *management*, and *termination* of processes.

What is a process? Naively:

- a program under execution,
- the "animated" existence of a program,
- an identifiable entity executed on a processor by the operating system.

More particularly, we consider how the operating system itself views a process:

- as an executable instance of a program,
- as the associated data operated upon by the program (variables, temporary results, external (file) storage, ...), and
- as the program's *execution context*.

It is a clear requirement of modern operating systems that they enable many processes to execute efficiently, by maximising their use of the processor, by supporting inter-process communication, and by maintaining reasonable response time.

This is an ongoing challenge: as hardware improves, it is "consumed" by larger, "hungrier" pieces of interlinked software.

Process States

We can view the process from two points of view: that of the processor, and that of the process itself.

- The processor's view is simple: the processor's only role is to execute machine instructions from main memory. Over time, the processor continually executes the sequence of instructions indicated by the program counter (PC).

  The processor is unaware that different sequences of instructions have a logical existence, that different sequences under execution are referred to as processes or tasks.

- From the process's point of view, it is either being executed by the processor, or it is waiting to be executed (for simplicity here, we consider that a terminated process no longer exists).

We've already identified two possible *process states* that a process may be in at any one time:**Running** and **Ready**.

Question: *Can a process determine in what state it is?*

Process Transitions

The operating system's role is to manage the execution of existing and newly created processes by moving them between the two states until they finish.

So we have a simple model consisting of two recurring steps:

1. Newly created processes are created and marked as **Ready**, and are *queued* to run.
2. There is only ever a single process in the **Running** state. It will either:
   - complete its execution and terminate (*exit*), or
   - be *suspended* (by itself or by the operating system), be marked as **Ready**, and be again *queued* to run.

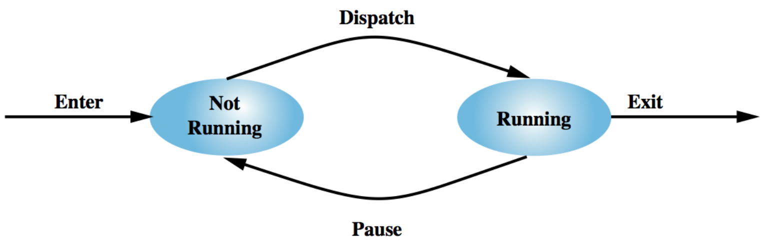
   One of the other **Ready** processes is then commenced (or *resumed*).

Here the operating system has the role of a *dispatcher* - dispatching work for the processor according to some defined policy addressing a combination of fairness, priority, apparent "interactivity", ...

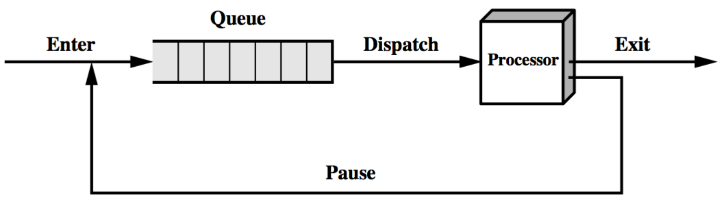
For simplicity (of both understanding and implementation) modern operating systems support the idle process which is always ready to run, and never terminates.

The Simple 2-state Process Model

As we generally have more than two processes available, the **Ready** state is implemented as a queue of available processes:



(a) State transition diagram



(b) Queuing diagram

When scheduling is discussed, we will introduce process priorities when deciding which **Ready** process should be the next to execute.

Process Creation

In supporting the **creation** of a new process, the operating system must allocate resources both for the process and the operating system itself.

The process (program under execution) will require a portion of the available memory to contain its (typically, read-only) instructions and initial data requirements. As the process executes, it will demand additional memory for its execution stack and its heap.

The operating system (as dispatcher) will need to maintain some internal control structures to support the migration of the process between states.

Where do new processes come from?

- an "under-burdened" operating system may take new process requests from a batch queue,
- a user logging on at a terminal usually creates an interactive control or encapsulating process (shell or command interpreter),
- an existing process may request a new process, and
- the operating system itself may create a process after an indirect request for service (to support networking, printing, ...)

Different operating systems support process creation in different ways.

- by requesting that an existing process be duplicated (using the *fork()* call in Linux and macOS),
- by instantiating a process's image from a named location, typically the program's image from a disk file (using the *spawn()* call in (old)DEC-VMS and the *CreateProcess()* call in Windows).

Process Termination

Stallings summarises typical reasons why a process will**terminate**:

- normal termination,
- execution time-limit exceeded,
- a resource requested is unavailable,
- an arithmetic error (division by zero),
- a memory access violation,
- an invalid request of memory or a held resource,
- an operating system or parent process request, or
- its parent process has terminated.

These and many other events may either terminate the process, or simply return an error indication to the running process. In all cases, the operating system will provide a default action which may or may not be process termination.

It is clear that process termination may be requested (or occur) when a process is either **Running** or **Ready**. The operating system (dispatcher) must handle both cases.

If a process is considered as a (mathematical) function, its return result, considered as a Boolean or integral result, is generally made available to (some) other processes.

Timer Interrupts

Why does a process move from **Running** to **Ready**?

The operating system must meet the two goals of fairness amongst processes and maximal use of resources (here, the processor and, soon, memory).

The first is easily met: enable each process to execute for a predetermined period before moving the **Running** process to the **Ready** queue.

- A *hardware timer* will periodically generate an interrupt (say, every 10 milliseconds). Between the execution of any two instructions, the processor will "look for" interrupts. When the *timer interrupt* occurs, the processor will begin execution of the *interrupt handler*.
- The handler will increment and examine the accumulated time of the currently executing process, and eventually move it from **Running** to **Ready**.
- The maximum time a process is permitted to run before changing state is often termed the *time quantum*.

The Blocking of Processes

The above scenario is simple and fair if all **Ready** processes are always truly ready to execute.

However, the existence of processes which continually execute to the end of their time quanta, often termed *compute-bound* processes, is rare.

More generally, a process will request some input or output (I/O) from a comparatively slow source (such as a disk drive, tape, keyboard, mouse, or clock). Even if the "reply" to the request is available immediately, a synchronous check of this will often exceed the remainder of the process's time quantum. In general the process will have to wait for the request to be satisfied.

The process should no longer be **Running**, but it is not **Ready** either: at least not until its I/O request can be satisfied.

We now introduce a new state for the operating system to support, **Blocked**, to describe processes waiting for I/O requests to be satisfied. A process requesting I/O will, of course, request the operating system to undertake the I/O, but the operating system supports this as three activities:

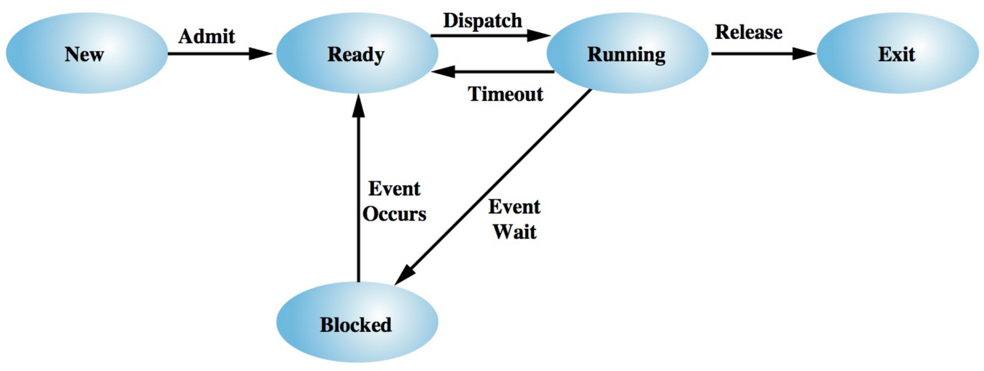
1. requesting I/O to or from the device,
2. moving the process from **Running** to **Blocked**,
3. preparing to accept an interrupt when I/O completes.

(Very simply) when the I/O completion interrupt occurs, the requesting process is moved from **Blocked** to **Ready**.

A degenerate case of blocking occurs when a process simply wishes to sleep for a given period. We consider such a request as "blocking until a timer interrupt", and have the operating system handle it the same way.

The 5-State Model of Process Execution

At this point, Stallings introduces his 5-state model:



This includes two new states:

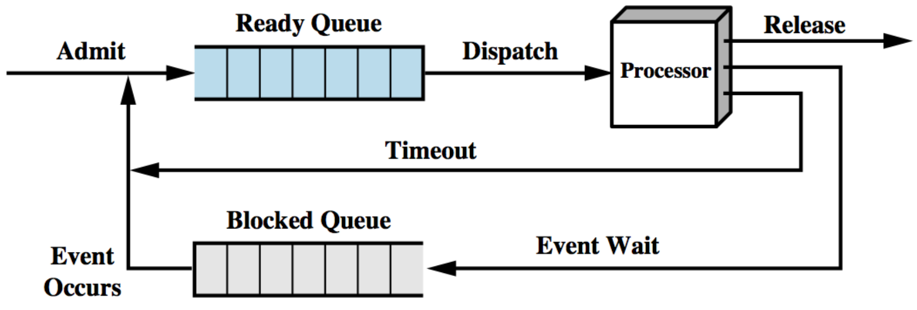
- **New** for newly created processes which haven't yet been admitted to the**Ready** queue for resourcing reasons;
- **Exit** for terminated processes whose return result or resources may be required by other processes, e.g. for post-process accounting.

Each of these states, except for **Running**, is likely to 'hold' more than one process (i.e. there may be more than one process in one of these states).

If a queue of processes is not always ordered in a first-come-first-served (FCFS) manner, then a *priority* or *scheduling mechanism* is necessary.

Supporting Multiple Blocked States

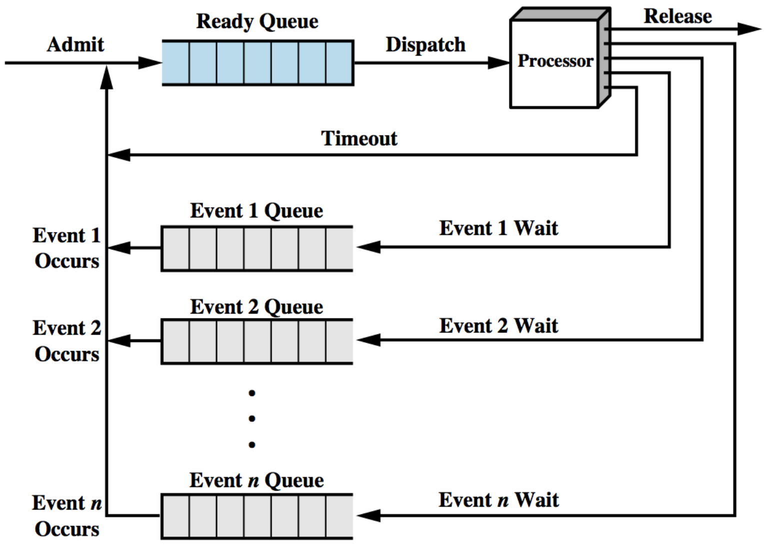
When notification of an I/O or timer completion occurs, the simplest queuing model requires the operating system to scan its **Blocked** queue to determine which process(es) requested, or are interested in, the event:



(a) Single blocked queue

Supporting Multiple Blocked States, *continued*

A better scheme is to maintain a queue for each possible event type. When an event occurs, its (shorter) queue is scanned more quickly:



(b) Multiple blocked queues

A typical example would have one queue for processes blocked on disk-drive-1, another blocked on disk-drive-2, another blocked on the keyboard ......

Then, when an interrupt occurs, it's quick to determine which process(es) should be unblocked, and moved to **Ready**.

The Dispatching Role of Operating Systems

As should now be clear, this view of the operating system as a dispatcher involves the moving of processes from one execution state to another.

The process's state is reflected by where it resides, although its state will also record much other information.

The possible *state transitions* that we have now discussed are:

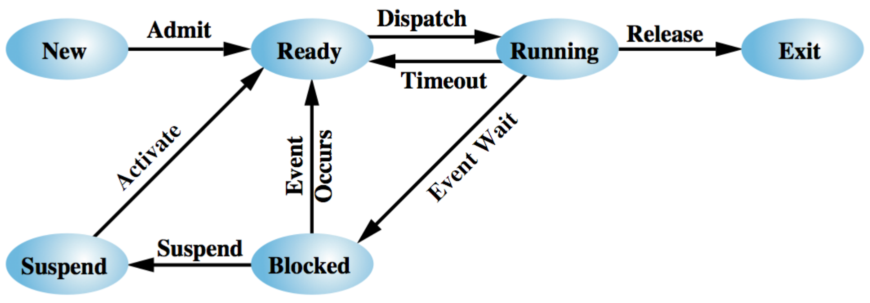
Null → New	a new process is requested.
New → Ready	resources are allocated for the new process.
Ready → Running	a process is given a time quantum.
Running → Ready	a process's execution time quantum expires.
Running → Blocked	a process requests slow I/O.
Blocked → Ready	an I/O interrupt signals that I/O is ready.
Running → Exit	normal or abnormal process termination.
Ready or Blocked → Exit	external process termination requested.

Suspension of Processes

Recall that the processor is much faster than I/O. As a consequence, it is quite possible for all processes to be blocked on I/O requests, when the processor will be idle most of the time while waiting for I/O interrupts.

Question: How to get more executing processes, given that resources such as memory are finite?

To enable more true work to be performed by the processor, we could provide more memory to support the requirements of more processes.



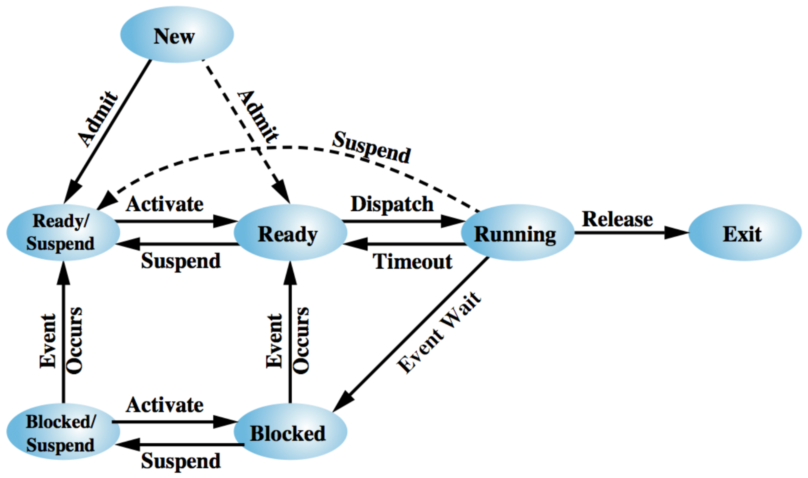
(a) With One Suspend State

But aside from the expense, providing more memory tends to encourage larger processes, not (in general) better support for more processes.

Swapping of Processes

Another solution is *swapping*: moving (part of) a process's memory to disk when it is not needed.

When none of the processes in main memory is **Ready** and more memory is required, the operating system *swaps* the memory of some of the **Blocked** processes to disk to reclaim some memory space. Such processes are moved to a new state: the **Suspend** state, a queue of processes that have been "kicked out" of main memory:



(b) With Two Suspend States

If desperate for even more memory, the operating system can similarly reclaim memory from **Ready** processes.
When memory becomes available, the operating system may now resume execution of a process from **Suspend**, or admit a process from **New** to **Ready**.