An Overview of Computer Hardware

Any study of operating systems requires a basic understanding of the basic components forming a computer system.

Although the variety of computer system configurations is forever changing, as (new) component types employ different standards for their interconnection, it is still feasible to develop a simple *computer model*, about which we can discuss components' roles in operating systems.

We traditionally consider four main structural components:

- The processor, or CPU, undertakes arithmetic and logical computation, and directs most input and output services from memory and peripheral devices. There may be more than one processor in a system. Processors execute both the operating system and user/application programs.

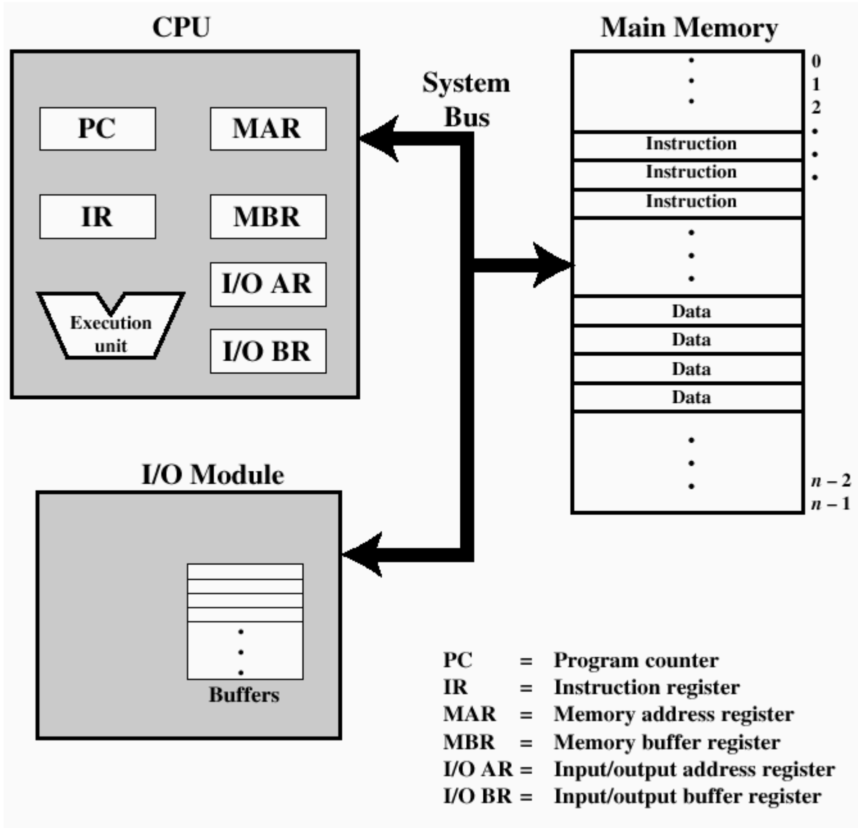
- Main memory, or RAM (random access memory) is used to store both programs and data. Processors will both read and write items of memory at the direction of programs (for data), and as an artifact of running programs (for instructions).

- Input/output modules or devices, and their I/O controllers, move data to and from the computer system, usually to provide longer-term, persistent storage of data (disks, tapes) or to provide representations of data (video, audio).

- A communications bus, or system bus, will connect the processor(s), main memory, and I/O devices together, providing a "highway" on which data may travel between the components. Typically only one component may control the bus at once, and *bus arbitration* decides which that will be.

Basic Computer Components

Stallings (p11) outlines a traditional computer model, in which the CPU, main memory, and I/O devices are all interconnected by a single systems bus:



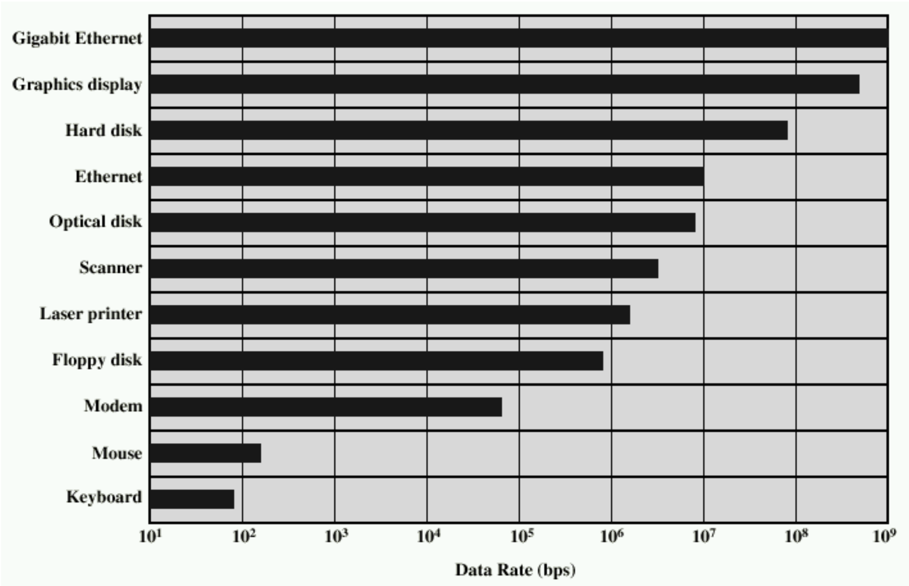
The CPU *fetches* a copy of the contents of uniquely *addressed* memory locations, by identifying the required location in its MAR.

Depending on why the CPU requires the value, it will execute the contents as an *instruction*, or operate on the contents as *data*.

Basic Computer Components, *continued*

Similarly, the CPU will locate data from, or for, distinct locations in the I/O devices using its I/O-AR.

The role of the operating system in managing the flow of data to and from its I/O devices is challenged by the wide variety of devices:



The operating system will attempt to attain *maximum throughput* of its computation and data transfer.

The very challenging area of processor scheduling attempts to keep the (expensive) processor busy at all times, by interleaving computation and communication. While waiting for a slow device to complete its I/O transfer, the processor may be able to undertake other activities.

Processor Registers

As well as special logic to perform arithmetic and logic functions, the processor houses a small number of very fast memory locations, termed *processor registers*.

Data in registers may be read and written very rapidly (with a typical access time of 1-3ns). If the required data is available in registers, rather than main memory, program execution may proceed 10-500× faster.

Registers are generally of two types:

- *user-accessible registers* – are accessible to programs, and may usually be read from and written to under program control. Programs written in an assembly language, either by a human programmer or generated by a compiler for a high-level language, will be able to read and write these registers using specific instructions understood by the processor which usually accept the names of the registers as operands.

  The user-accessible registers are further of two types – *data registers* hold values before the execution of certain instructions, and hold the results after executing certain instructions.

  Address registers hold the address (not contents) of memory locations used in the execution of a program.

  The *memory address register* (MAR) holds the address of memory to be read or written, the *memory buffer register* (MBR) holds the memory's data just read, or just about to be written, *index registers* hold an integer offset from which a sequence of memory references are made, and a *stack pointer* (SP) holds the address of a dedicated portion of memory holding temporary data and other memory addresses.

Processor Registers, *continued*

- *control and status registers* – to execute programs, the processor must itself maintain the contents of a number of special registers.

  The *program counter* (PC) holds the memory address of the next instruction to be executed; the *instruction register* (IR) holds the current instruction being executed,

  Special registers will reflect the *status* of the processor. The *processor status word* (PSW) will typically reflect whether or not the processor may be interrupted by I/O devices, whether privileged instructions may be executed, and use *condition bits* to reflect the status of recently executed operations – did an arithmetic operation *overflow*, did an arithmetic operation perform a *carry*, did we just attempt a division by zero, did the last comparison instruction succeed or fail?

Different types of processors have varying number of registers, For example, some processors may have no data registers at all, some have a few (3→16), or many (32→hundreds).

Processors place constraints on how some registers are used. Some processors expect certain types of data to reside in specific registers. For example, some registers may be expected to hold integer operands for arithmetic instructions, whereas some registers may be reserved for holding float-point data.

As a result of executing a program, processors update many of the address registers independently. For example, instructions to *push* and *pop* data values to and from the stack, automatically modify the contents of the stack pointer.

An *excellent*, albeit expensive, textbook introducing computer system organization in depth is *Computer Organization and Design : The Hardware/Software Interface*, by David A. Patterson and John L. Hennessy. Morgan Kaufmann Publishers, 2nd edition (September 1997).

The Memory Hierarchy

The clear role of memory is to hold instructions and data until they are requested by the processor. While it is easy to make a case for as much memory as possible, having too much can be wasteful (financially) if it is not all required.

We also expect memory to be able to provide the necessary data, as quickly as possible, when called upon. Unfortunately, there is a traditional tradeoff before cost, capacity, and access time:

- the faster the access time, the greater the cost per bit,
- the greater the capacity, the smaller the cost per bit, and
- the greater the capacity, the slower the access time.

The solution taken is not to rely on a single, consistent form of memory, but instead to have a *memory hierarchy*, constrained by requirements and cost.

Memory	Access time	Capacity	Technology	Managed by
registers	1-3ns	1-4KB	custom CMOS	compiler
level-1 cache (on-chip)	2-8ns	8KB-128KB	SRAM	hardware
level-2 cache (off-chip)	5-12ns	0.5MB-8MB	SRAM	hardware
main memory	10-60ns	64MB-2GB	DRAM	operating system
hard disk	3M-10M ns	20-100GB	magnetic	operating system

For example, a contemporary home computer system will include a modest amount of cache memory (128KB), to deliver data as quickly as possible to the processor, a larger main memory (128MB) to store entire programs and less frequently required data, and long term, persistent storage in the form of a hard disk (20GB).

Units of data – bits, bytes, and words

The basic building block is the *bit* (binary digit), which can contain a single piece of binary data (true/false, zero/one, high/low, etc.). Although processors provide instructions to set and compare single bits, it is rarely the most efficient method of manipulating data.

Bits are organized into larger groupings to store values encoded in binary bits. The most basic grouping is the *byte* – the smallest normally *addressable* quantum of main memory (which can be different than the minimum amount of memory fetched at one time).

In modern computers, a byte is almost always an eight bit byte, but history has seen computers with seven, eight, nine, twelve, and sixteen bit bytes.

A *word* is the default data size for a processor. The word size is chosen by the processor's designer and reflects some basic hardware issues (such as the width of internal or external buses). The most common word sizes are 16 and 32; historically words have ranged from 16 to 60 bits.

It is very common speak of a processor's *wordsize*, such as a 32-bit or 64-bit processor. However, different sources will confuse whether this means the size of a single addressable memory location, or the default unit of integer arithmetic.

Some processors require that data be *aligned*, That is, two byte quantities must start on byte addresses that are multiples of two; four byte quantities must start on byte addresses that are multiples of four; etc. Some processors allow data to be unaligned, but this can result in slower execution as the processor may have to align the data itself.

On the interpretation of data

We have seen the computer systems store their data as bits, and group bits together as bytes and words.

However, most importantly, the processor is unable to examine a sequence of bits and uniquely determine what that sequence represents.

A single 32 *bit pattern* could refer to 4 ASCII characters, a 32-bit integer, two 16-bit integers, a floating point value, the address of a memory location, or an instruction to be executed. No *meaning* is stored along with each bit pattern – it is up to the processor to apply some *context* to the sequence to ascribe it some meaning.

For example, a sequence of integers *may* form a sequence of valid processor instructions that could be meaningfully executed; a sequence of processor instructions can always be interpreted as a vector of, say, integers and can thus be added together.

Critical errors occur when a bit sequence is interpreted in the wrong context. If a processor attempts to execute a meaningless sequence of instructions, a *processor fault* will generally result – LINUX announces this as a "bus error". Similar faults occur when instructions expect data on aligned data boundaries, but are presented with unaligned addresses.

On the interpretation of data, *continued*

As an example of how bytes may be interpreted differently, consider the first few hundred bytes of the disk file */bin/ls*. We know this to be a program, and we expect the operating system to interpret its contents to be a program, and request the processor to execute its contents (a mixture of instructions and data).

However, another program could read the bytes from */bin/ls* and interpret them in other ways, as 32-bit integers:

```
prompt>  od -l /bin/ls
0000000  1179403647       65793           0           0
0000020      196610           1   134518048          52
0000040       45968           0     2097204     2621446
0000060     1441815           6          52   134512692
0000100   134512692         192         192           5
0000120           4           3         244   134512884
```

as octal (8-bit) bytes:

```
prompt>  od -b /bin/ls
0000000 177 105 114 106 001 001 001 000 000 000 000 000 000 000 000 000
0000020 002 000 003 000 001 000 000 000 040 225 004 010 064 000 000 000
0000040 220 263 000 000 000 000 000 000 064 000 040 000 006 000 050 000
0000060 027 000 026 000 006 000 000 000 064 000 000 000 064 200 004 010
0000100 064 200 004 010 300 000 000 000 300 000 000 000 005 000 000 000
0000120 004 000 000 000 003 000 000 000 364 000 000 000 364 200 004 010
```

or as ASCII characters:

```
prompt> od -c /bin/ls
0000000 177   E   L   F 001 001 001  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020 002  \0 003  \0 001  \0  \0  \0     225 004  \b   4  \0  \0  \0
0000040 220 263  \0  \0  \0  \0  \0  \0   4  \0      \0 006  \0   (  \0
0000060 027  \0 026  \0 006  \0  \0  \0   4  \0  \0  \0   4 200 004  \b
0000100   4 200 004  \b 300  \0  \0  \0 300  \0  \0  \0 005  \0  \0  \0
0000120 004  \0  \0  \0 003  \0  \0  \0 364  \0  \0  \0 364 200 004  \b
```

And each interpretation could be correct, depending on context.

Processes and process states

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 maximizing 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, more "hungry" 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, 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, `Running`, or it is ready, waiting to be executed, `Ready`, (for simplicity here, we consider that a terminated process no longer exists).

We've identified two possible *process states* that a process may be in at any one time – `Running` and `Ready` (to execute).

Question: can a process determine in what state it is?

Process Transitions

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

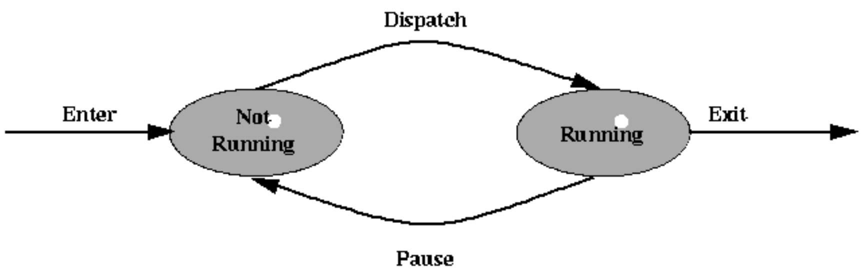
For simplicity (of both understanding and implementation) modern operating systems support the *idle process* which is always ready to run, and never terminates.

Newly created processes are created and marked as "ready to run" and *queued* to run.

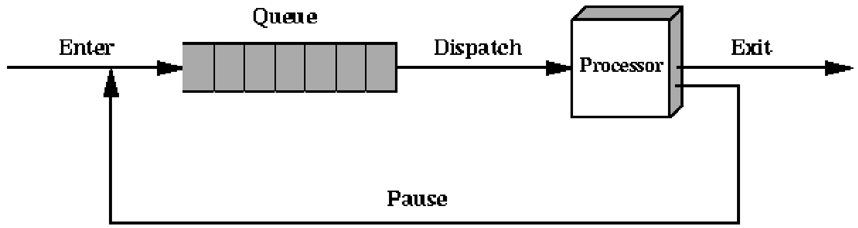
As the single running process terminates, or is interrupted and marked as (only) "ready to run" by the operating system, the next `Ready` process is commenced (or continued).

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

The Simple 2-state Process Model



As we will generally have more than two processes available, the `Ready` state will be implemented as a queue of available processes.



[When scheduling is discussed (lecture topic 5) we'll introduce process priorities when deciding *which* of the `Ready` processes 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 `Running` and `Ready` (until termination).

Where do the 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,
- 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 (ala the `fork()` call in UNIX and Mach),
- by instantiating a process's image from a named location, typically the program's image from a disk file (ala the `spawn()` call in DEC-VMS and the `CreateProcess()` call in Windows-NT).

Process Termination

Stallings (p114) summarizes typical reasons why a process will terminate, including –

- 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 simply `Ready`. The operating system (dispatcher) must handle both states.

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 process move from the `Running` to the `Ready` state?

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 requirement here 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 *timer interrupt handler*.

The timer 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 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-intensive* or *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). The "reply" to the request may be available immediately, but even 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 marked as `Running`, but it is not `Ready` to run either – at least until its I/O request can be satisfied.

The Blocking of Processes, *continued*

We now introduce a new state for the operating system to support, `Blocked`, to describe processes waiting for I/O requests to become available.

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:

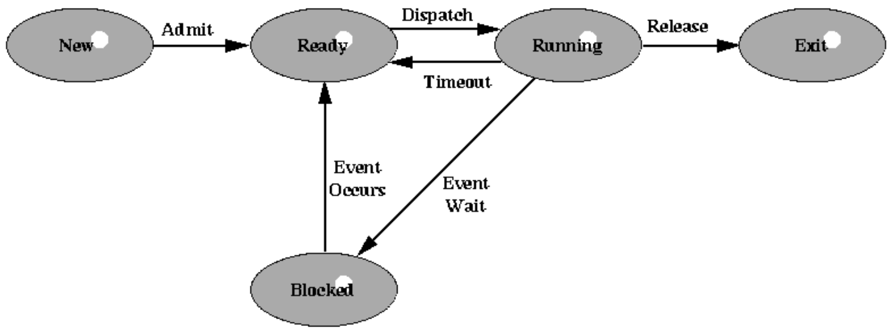
- requesting I/O from the device,
- moving the process from `Running` to `Blocked`,
- preparing to accept an interrupt when I/O completes.

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

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 (p104).

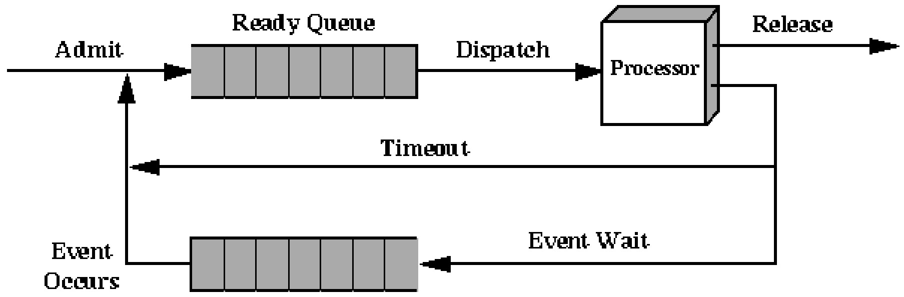


This includes a new state, `New`, for processes newly created but not yet admitted to the `Ready` queue for resourcing reasons.

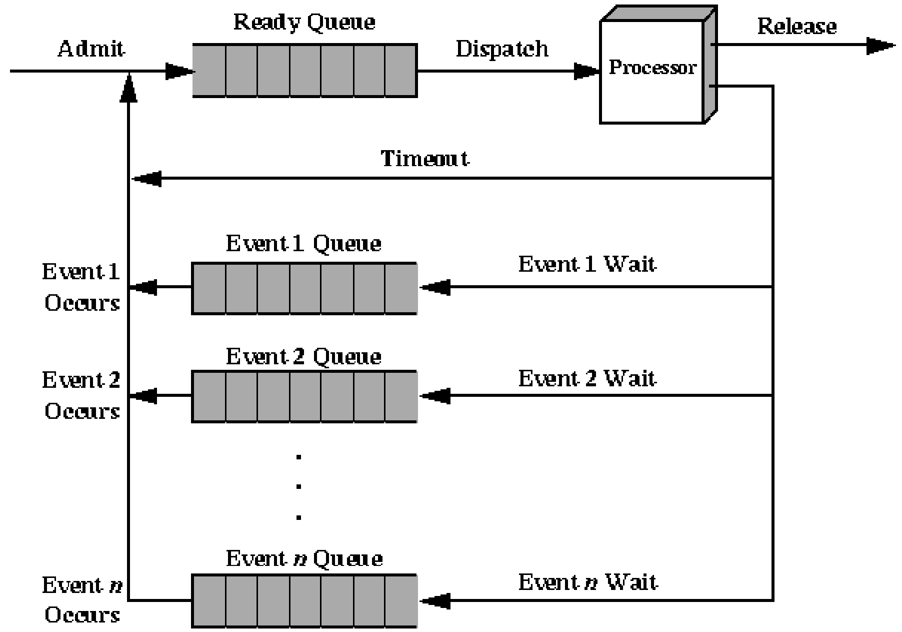
Another new state is `Exit`, for terminated processes whose return result or resources may be required by other processes (for *post-process accounting*, etc).

Supporting Multiple `Blocked` States

The current state model requires the operating system to scan its `Blocked` queue each time an event occurs.



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



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, though its state will also have some physical tag (later).

The possible state transitions are now:

- 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, Blocked→Exit: external process termination requested.

Suspension and Swapping of Processes

Recall that the process is so much faster than I/O. As a consequence, it is quite possible for *all* processes to be blocked on I/O requests, and 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.

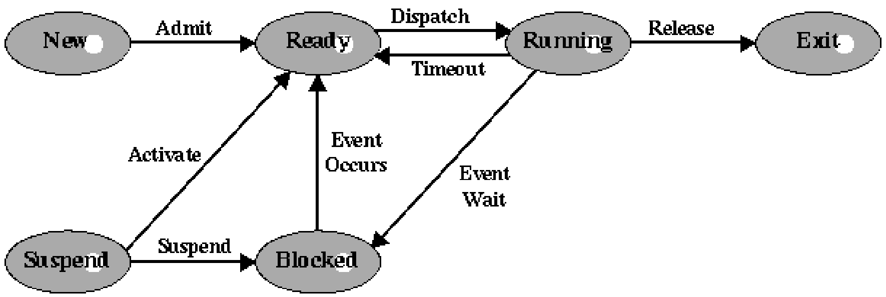
But aside from the expense, providing more memory tends to encourage *larger* process, and not (in general) better support for more processes.

Another solution is *swapping* – moving part, or all, of a process's memory to disk when it is not needed.

Suspension and Swapping of Processes, *cont.*

When none of the processes in main memory is in the `Ready` state, the operating system swaps the memory of some of the `Blocked` processes to disk to recover some memory space. Such processes are moved to a new state – the new `Suspend` state, a queue of processes that have been "kicked" out of main memory.

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 the `Ready` state, or admit a process from the `New` state to the `Ready` state.