Principles of Operating Systems

CS 446/646

2. Processes

René Doursat

Department of Computer Science & Engineering
University of Nevada, Reno

Spring 2006

Principles of Operating Systems

CS 446/646

0. Course Presentation
1. Introduction to Operating Systems
2. **Processes**
3. Memory Management
4. CPU Scheduling
5. Input/Output
6. File System
7. Case Studies

Principles of Operating Systems
CS 446/646

2. Processes

a. Process Description & Control

b. Threads

c. Concurrency

d. Deadlocks

Principles of Operating Systems

CS 446/646

2. Processes

a. **Process Description & Control**
   - ✓ What is a process?
   - ✓ Process states
   - ✓ Process description
   - ✓ Process control

b. **Threads**

c. **Concurrency**

d. **Deadlocks**

2.a Process Description & Control

What is a process?

- A process is the <u>activity</u> of executing a program

Pasta for six

- boil 1 quart salty water
- stir in the pasta
- cook on medium until "al dente"
- serve

thread of execution

Program

Process

2.a Process Description & Control

What is a process?

1. Given that a computer system is organized into
   - ✓ hardware resources (CPU, memory, I/O, timer, disks, etc.)
   - ✓ operating system software
   - ✓ user application software
2. Given the O/S responsibility of executing applications
   - ✓ resources be made available to multiple applications
   - ✓ the CPU, in particular, be switched among multiple applications
   - ✓ the CPU and I/O devices be utilized efficiently

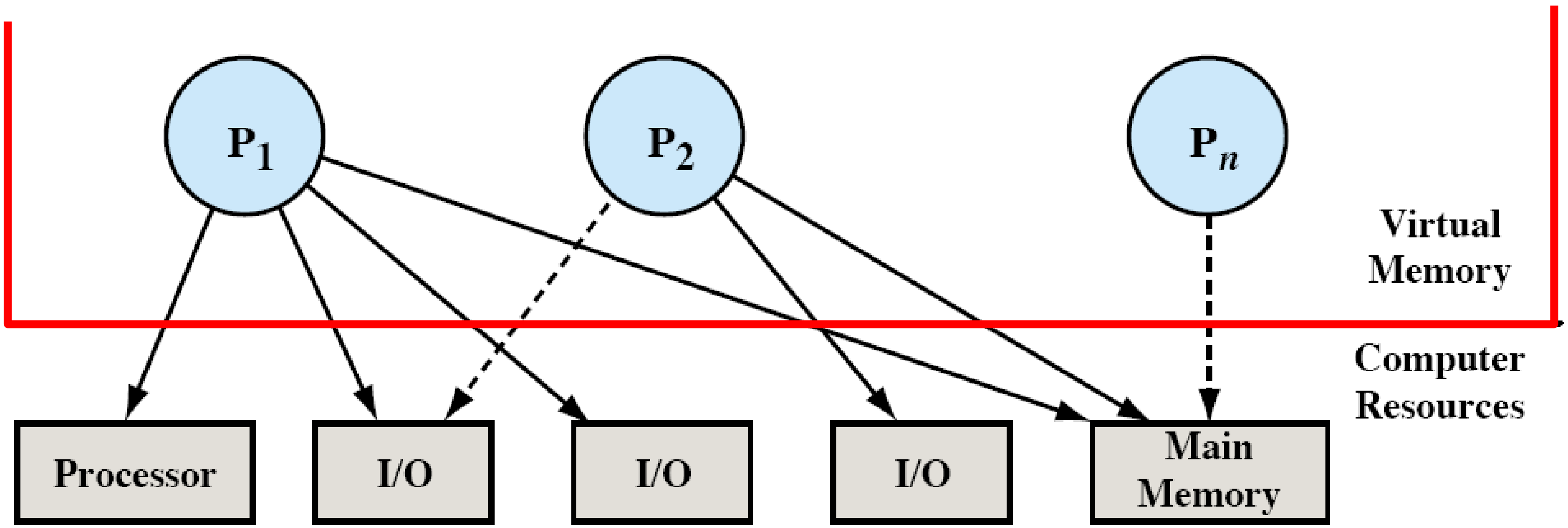
➢ . . . the approach taken by modern O/S is the "process"

- ✓ modern O/S rely on a model in which the execution of an application is abstracted into one or more **processes**

2.a Process Description & Control

What is a process?

- The O/S has to multiplex resources to the processes
  - ✓ a number of processes have been created
  - ✓ each process during the course of its execution needs access to system resources: CPU, main memory, I/O devices



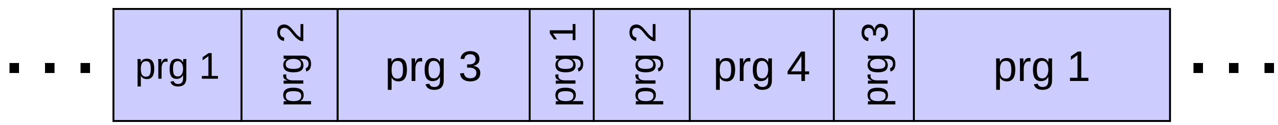
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Resource allocation for processes (one snapshot in time)

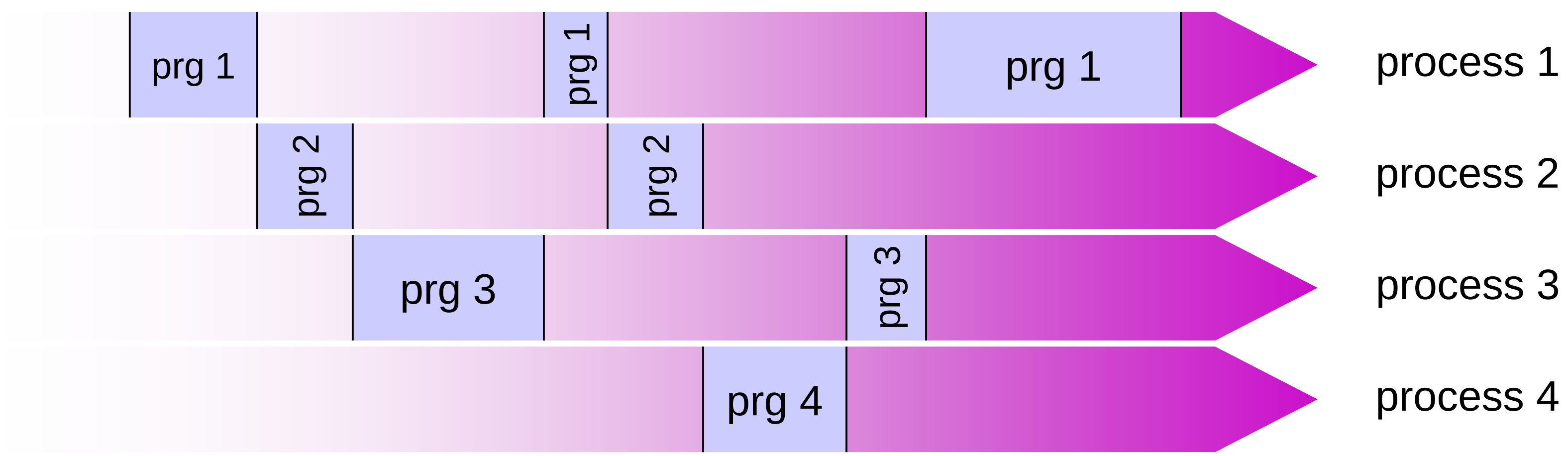
2.a Process Description & Control

What is a process?

- Multitasking can be conveniently described in terms of multiple processes running in (pseudo)parallel



(a) Multitasking from the CPU's viewpoint



(b) Multitasking from the processes' viewpoint = 4 virtual program counters

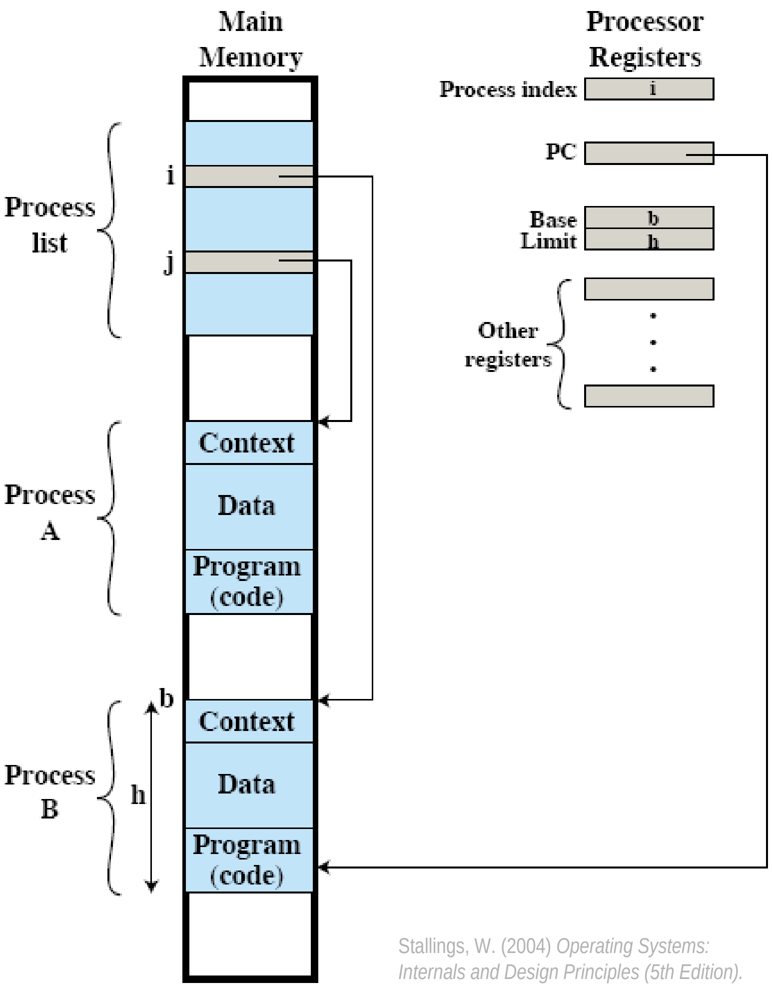
Pseudoparallelism in multitasking

2.a Process Description & Control

What is a process?

- A process image consists of three components

user address space
1. an executable <u>program</u>
2. the associated <u>data</u> needed by the program
3. the execution <u>context</u> of the process, which contains all information the O/S needs to manage the process (ID, state, CPU registers, stack, etc.)



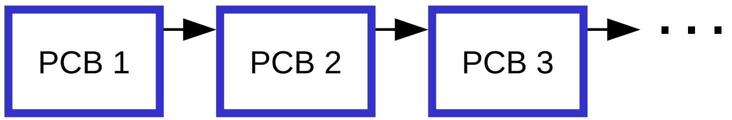
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

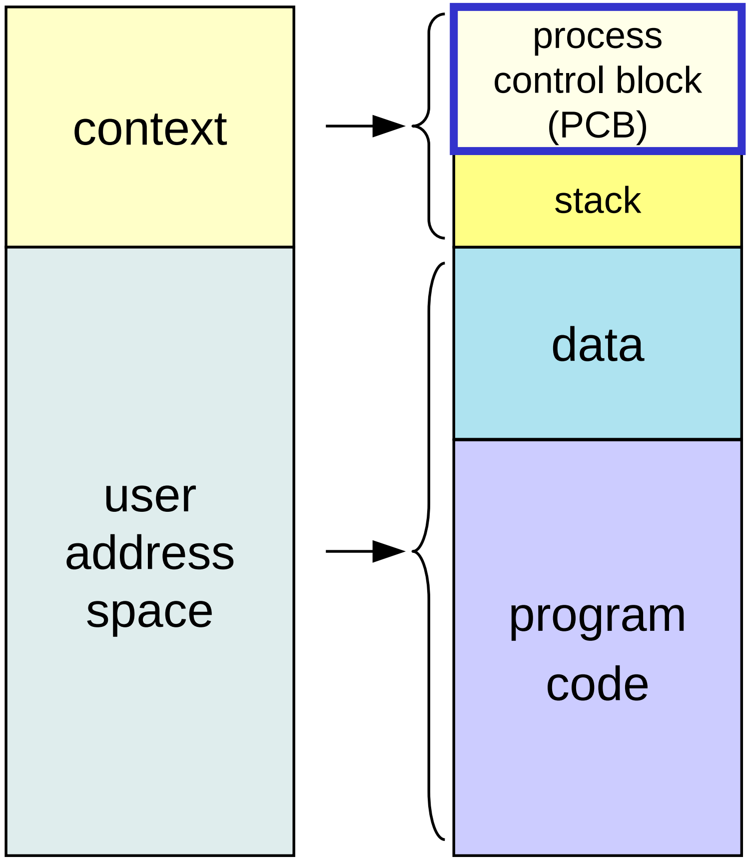
Typical process image implementation

2.a Process Description & Control

What is a process?

- The Process Control Block (PCB)
  - ✓ is included in the context, along with the stack
  - ✓ is a "snapshot" that contains all necessary and sufficient data to restart a process where it left off (ID, state, CPU registers, etc.)
  - ✓ is one entry in the operating system's **process table** (array or linked list)





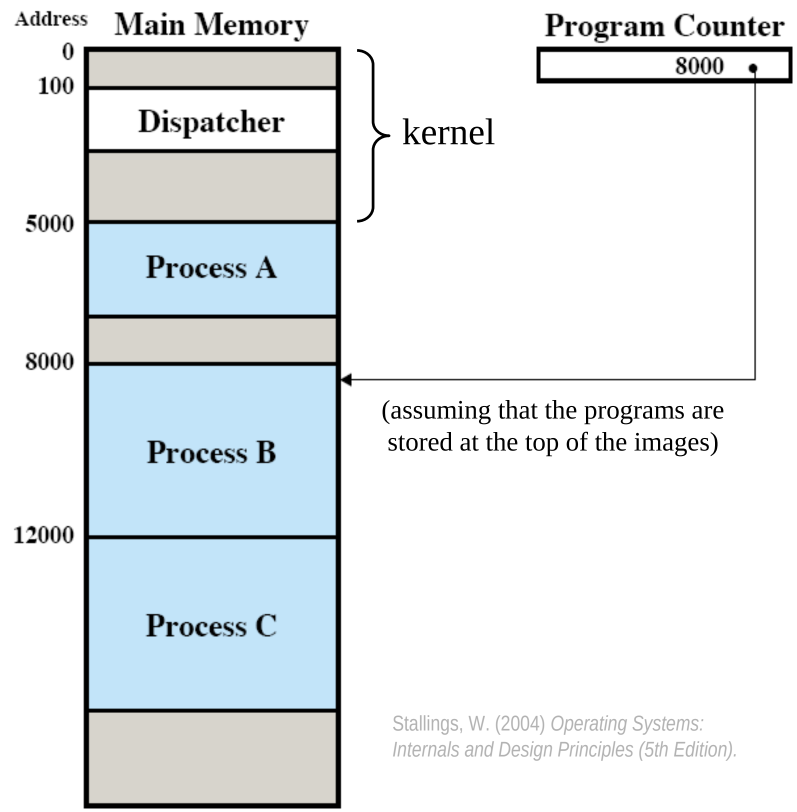
Typical process image implementation

2.a Process Description & Control

What is a process?

- **A dispatcher switches the CPU between processes**
  - ✓ the dispatcher is a routine program in kernel memory space

PC			
1	5000	27	12004
2	5001	28	12005
3	5002	---------------------- Time out	
4	5003	29	100
5	5004	30	101
6	5005	31	102
---------------------- Time out		32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	---------------------- Time out	
16	8003	41	100
------------------ I/O request		42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		---------------------- Time out	



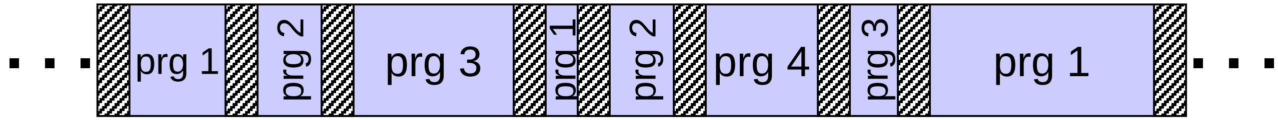
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Dispatching between three processes

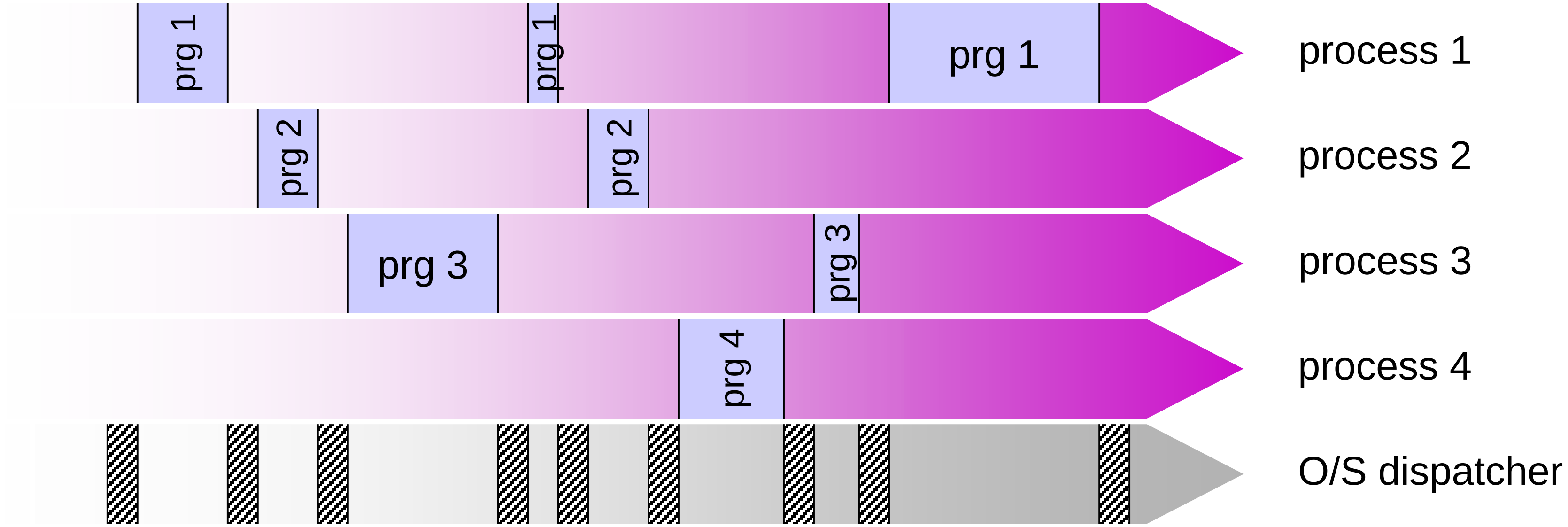
2.a Process Description & Control

What is a process?

- A dispatcher switches the CPU between processes
  - ✓ the dispatcher is a routine program in kernel memory space



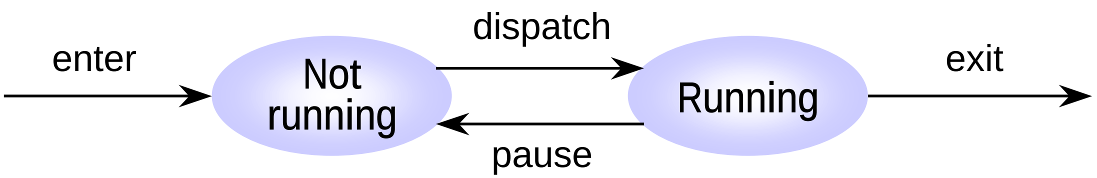
(a) Multitasking from the CPU's viewpoint



2.a Process Description & Control

Process states

- Deep truth: at any time, a given process is either being executed by the CPU or it is not
  - ✓ thus, a process can have two states: running or not running



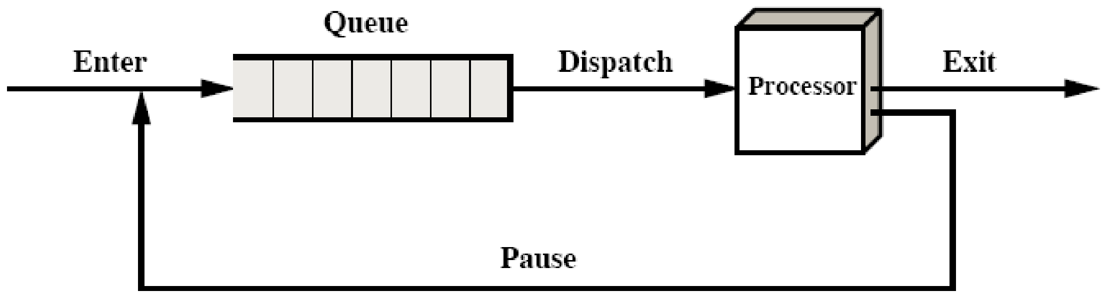
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Transition diagram of a two-state process model

2.a Process Description & Control

Process states

- ➢ How does the O/S keep track of processes and states?
  - ✓ by keeping a queue of pointers to the process control blocks



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

  - ✓ the queue can be implemented as a linked list if each PCB contains a pointer to the next PCB

Queuing diagram of a two-state process model

2.a Process Description & Control

Process states

- Some events that lead to process <u>creation</u> (enter)

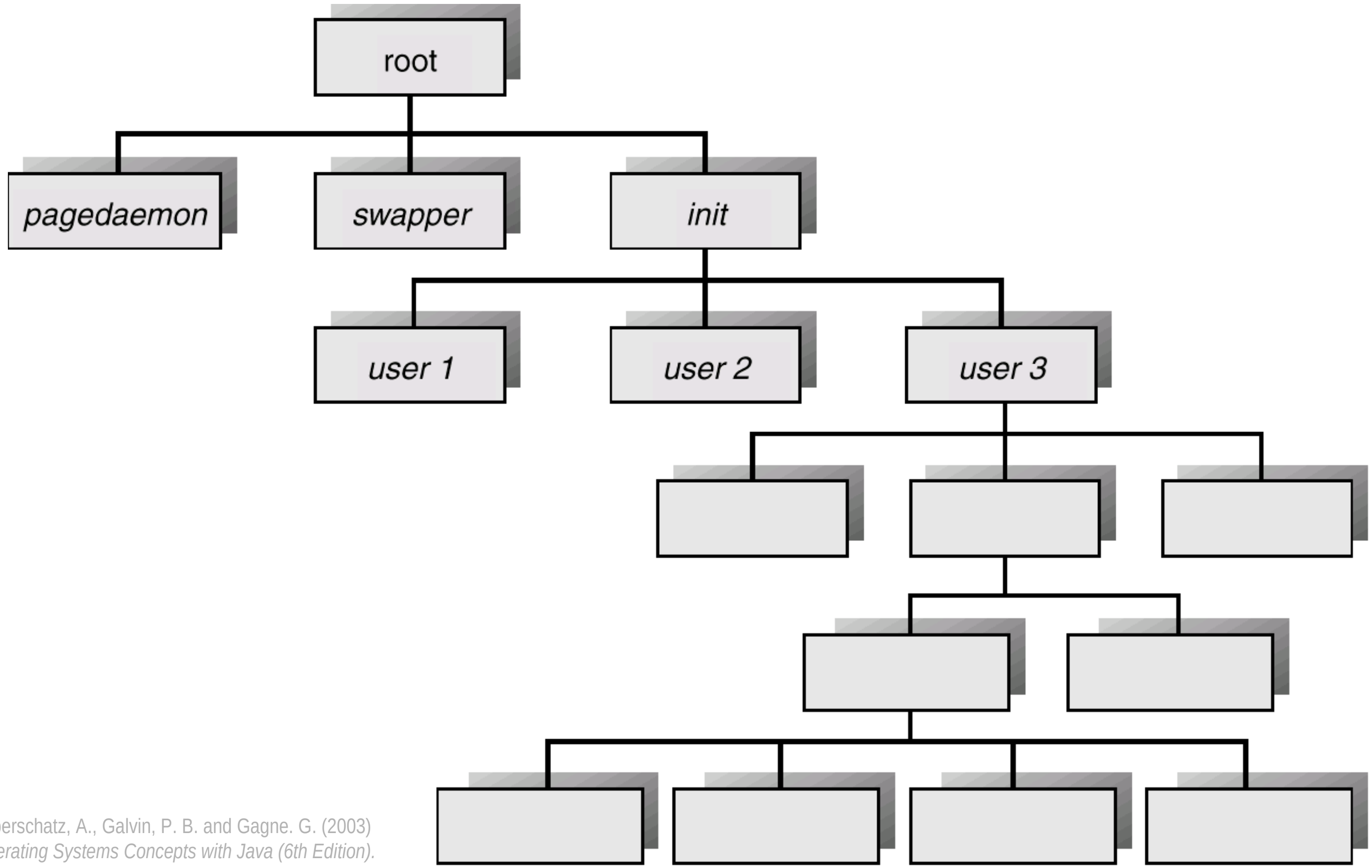
all cases of process spawning

- ✓ the system boots
  - when a system is initialized, several background processes or “daemons” are started (email, logon, etc.)
- ✓ a user requests to run an application
  - by typing a command in the CLI shell or double-clicking in the GUI shell, the user can launch a new process
- ✓ an existing process spawns a child process
  - for example, a server process (print, file) may create a new process for each request it handles
  - the *init* daemon waits for user login and spawns a shell
- ✓ a batch system takes on the next job in line

2.a Process Description & Control

Process states

- Process creation by spawning



Silberschatz, A., Galvin, P. B. and Gagne. G. (2003)
Operating Systems Concepts with Java (6th Edition).

A tree of processes on a typical UNIX system

2.a Process Description & Control

Process states

```
...
int main(...)
{
    ...
    if ((pid = fork()) == 0)                        // create a process
    {
        fprintf(stdout, "Child pid: %i\n", getpid());
        err = execvp(command, arguments);           // execute child
                                                    //   process
        fprintf(stderr, "Child error: %i\n", errno);
        exit(err);
    }
    else if (pid > 0)                               // we are in the
    {                                               //   parent process
        fprintf(stdout, "Parent pid: %i\n", getpid());
        pid2 = waitpid(pid, &status, 0);            // wait for child
        ...                                         //   process
    }
    ...

    return 0;
}
```

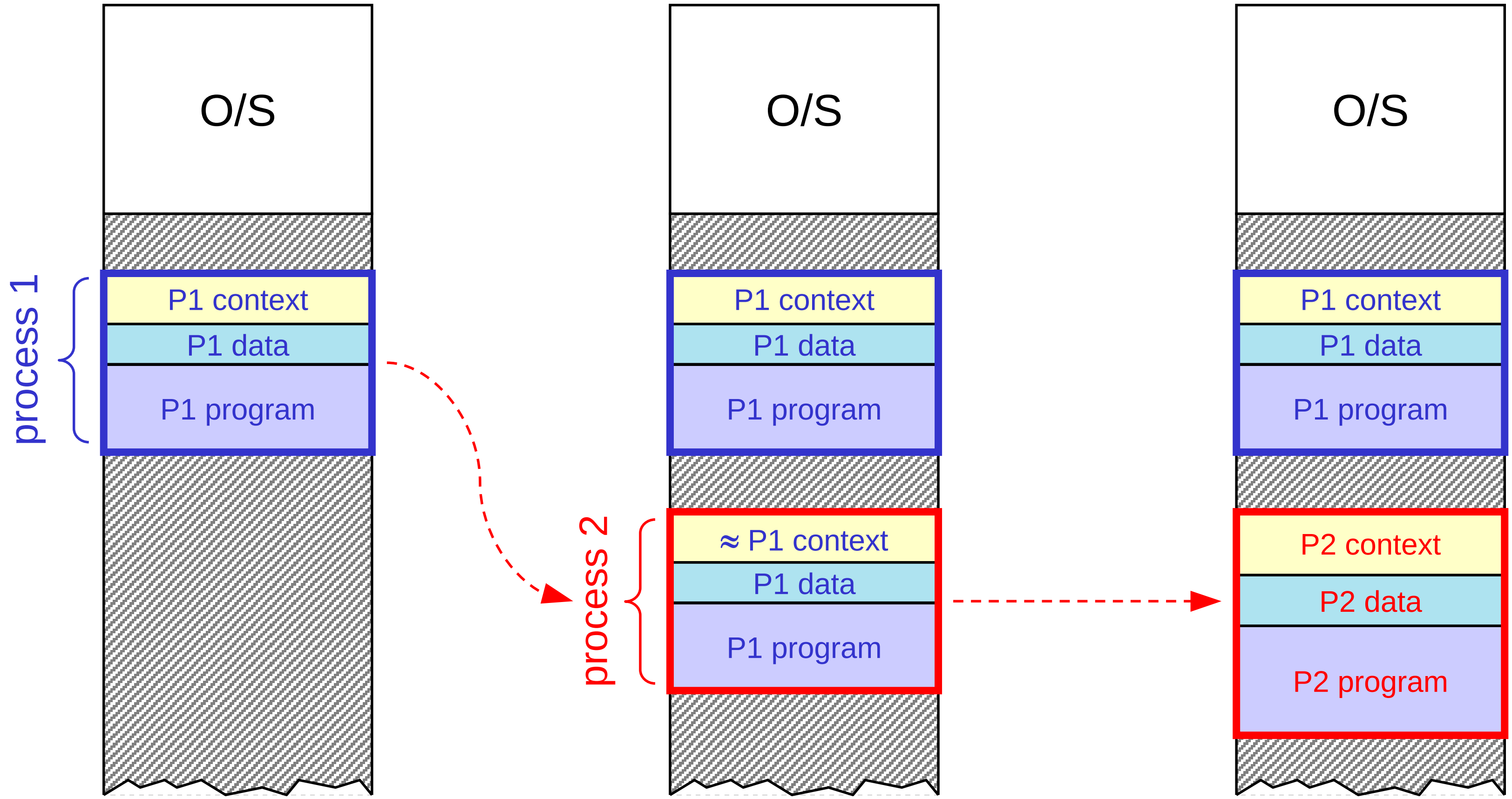
Implementing a shell command interpreter by process spawning

2.a Process Description & Control

Process states

1. Clone child process
   - ✓ `pid = fork()`

2. Replace child's image
   - ✓ `execve(name, ...)`



2.a Process Description & Control

Process states

- Some events that lead to process termination (exit)
  - ✓ regular completion, with or without error code
    - process-triggered: the process voluntarily executes an `exit(err)` system call to indicate to the O/S that it has finished
  - ✓ fatal error (uncatchable or uncaught)
    - O/S-triggered (following system call or preemption):
      - service errors: no memory left for allocation, I/O error, etc.
      - total time limit exceeded
    - hardware interrupt-triggered: arithmetic error, out-of-bounds memory access, etc.
  - ✓ killed by another process via the kernel
    - software interrupt-triggered: the process receives a **`SIGKILL`** signal
    - in some systems the parent takes down its children with it

2.a Process Description & Control

Process states

- Some events that lead to process <u>pause</u> / <u>dispatch</u>
  - ✓ I/O wait
    - O/S-triggered (following system call) ▪ a process invokes an I/O system call that blocks waiting for the I/O device: the O/S puts the process in "Not Running" mode and dispatches another process to the CPU
  - ✓ preemptive timeout
    - hardware interrupt-triggered (timer) ▪ the process receives a timer interrupt and relinquishes control back to the O/S dispatcher: the O/S puts the process in "Not Running" mode and dispatches another process to the CPU
    - ▪ not to be confused with "total time limit exceeded", which leads to process termination

2.a Process Description & Control

Process states

- Steps in making a system call that must wait for I/O

  1. – 3. . . . program prepares stack

  4. . . . program calls **read**

  5. . . . **read** stores **#read** in reg

  6. . . . **read** executes **TRAP**

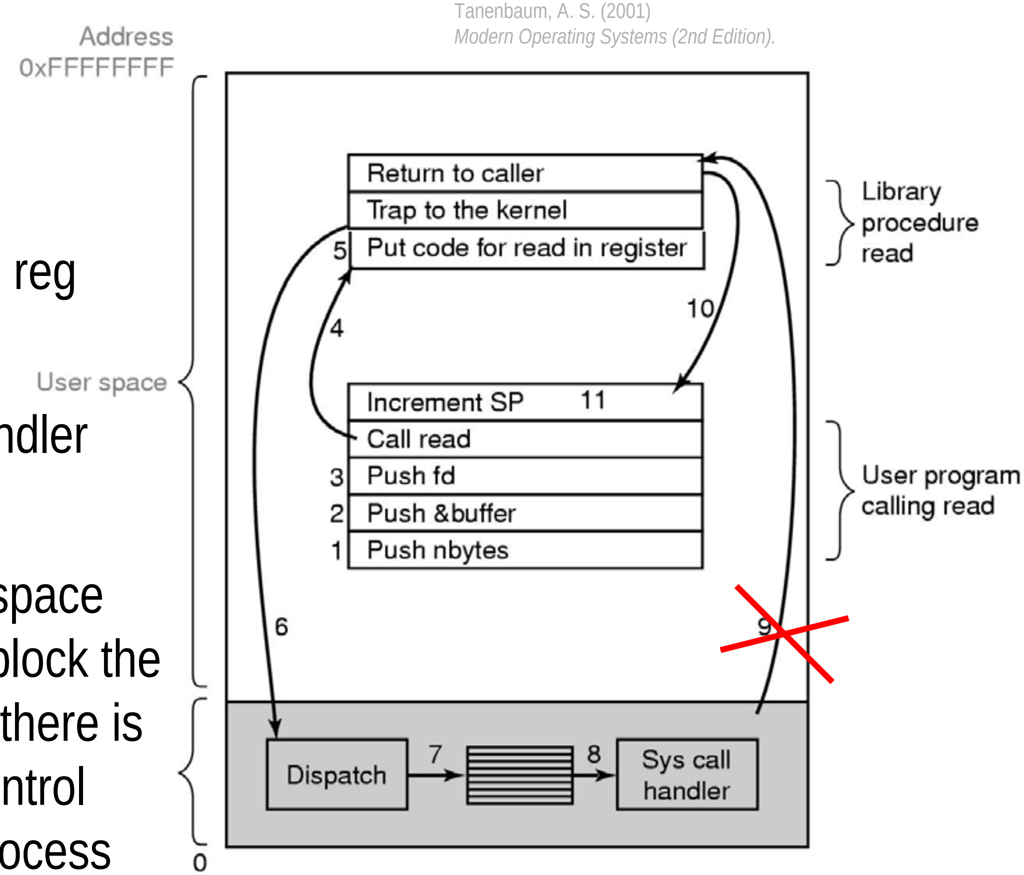
  7. . . . kernel dispatches to call handler

  8. . . . system call handler runs

  9. control does not return to user space right away; the O/S decides to block the caller ("Not Running") because there is no input to read yet; instead, control eventually returns to another process

  → *not just mode switch: full process switch!*

Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

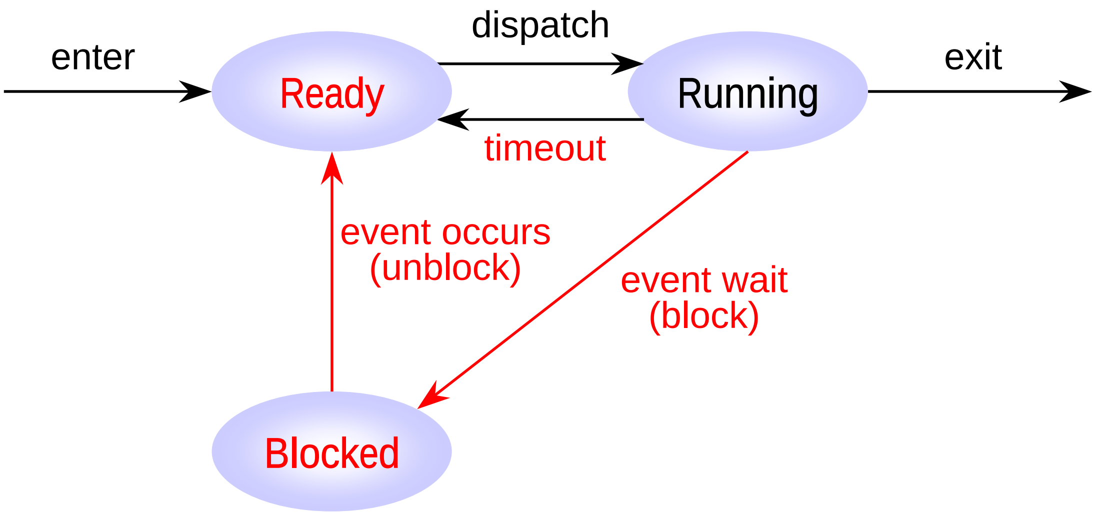


11 steps in making a system call

2.a Process Description & Control

Process states

- Problem with the two-state model
  - ✓ some “Not Running” processes are blocked (waiting for I/O, etc.)
  - ✓ the O/S wastes time scanning the queue for ready processes



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

→ solution: divide “Not Running” into “Ready” and “Blocked”

Transition diagram of a three-state (“Blocked/Ready”) process model

2.a Process Description & Control

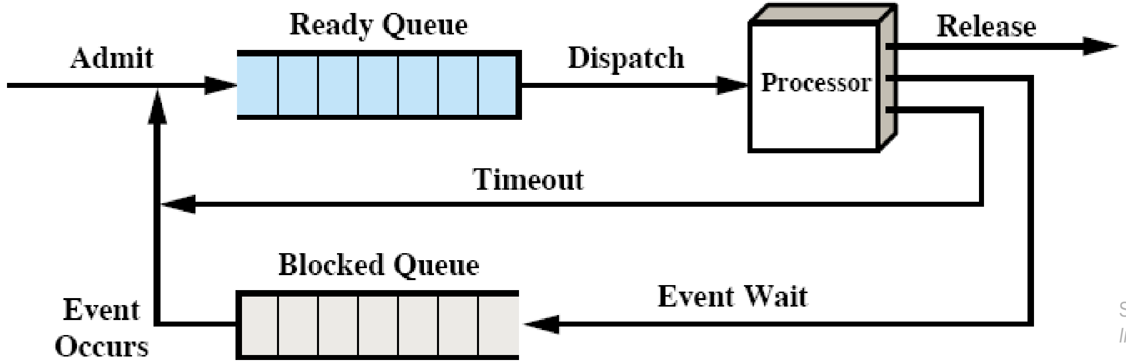
Process states

- Some events that lead to process timeout / dispatch block / unblock
  - ✓ I/O wait
    - O/S-triggered (following system call) ■ a process invokes an I/O system call that blocks waiting for the I/O device: the O/S puts the process in "Blocked" mode and dispatches another process to the CPU
  - ✓ preemptive timeout
    - hardware interrupt-triggered (timer) ■ the process receives a timer interrupt and relinquishes control back to the O/S dispatcher: the O/S puts the process in "Ready" mode and dispatches another process to the CPU
    - ■ not to be confused with "total time limit exceeded", which leads to process termination

2.a Process Description & Control

Process states

- How does the O/S keep track of three process states?
  - ✓ by keeping an extra queue for blocked processes



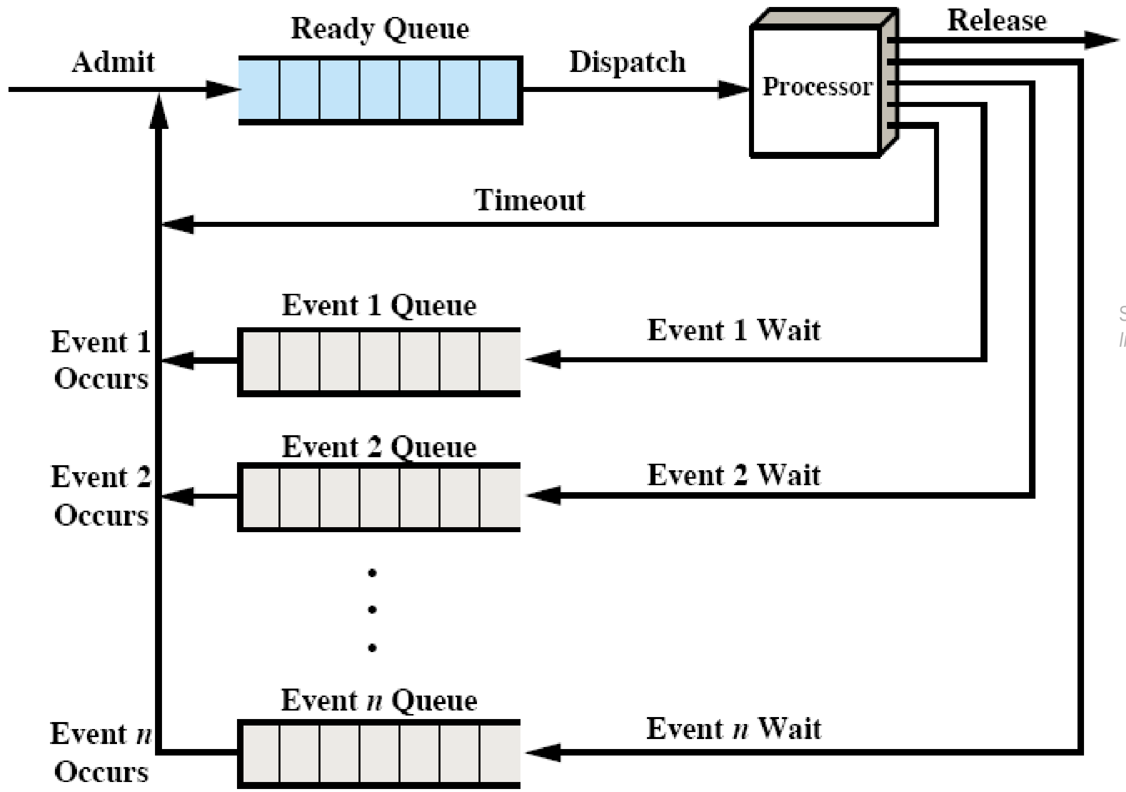
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Queuing diagram of a three-state (“Blocked/Ready”) process model

2.a Process Description & Control

Process states

- To further reduce scanning, blocked processes can be placed in separate queues depending on the event type



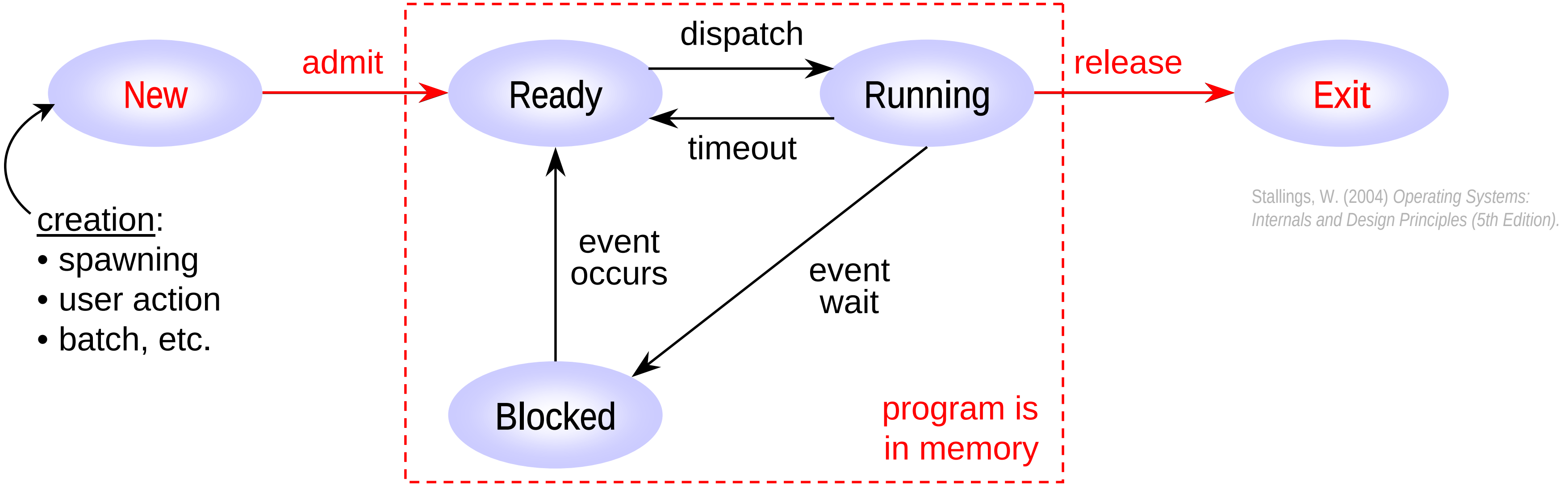
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Queuing diagram of a three-state ("Blocked/Ready") process model with multiple event queues

2.a Process Description & Control

Process states

- How is a process actually created (entered)?
  - ✓ in two steps: first the PCB is created and put in a “New” pool
  - ✓ then, program & data are loaded and the process is “Ready”



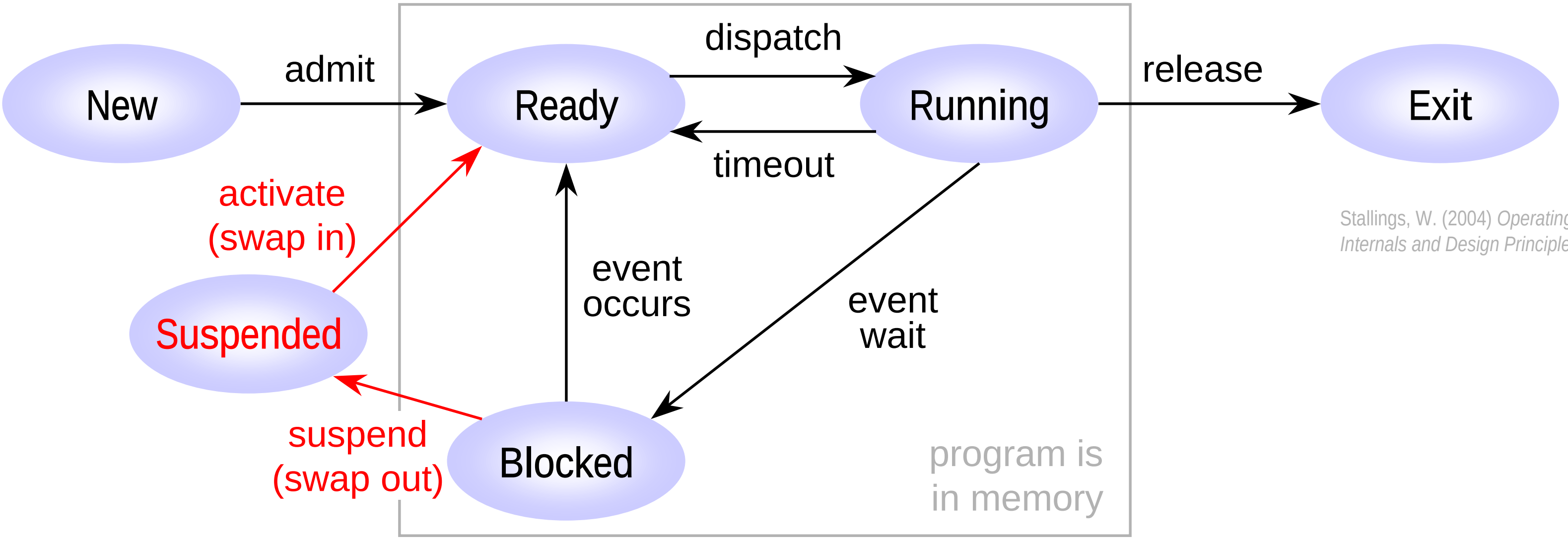
- ✓ conversely with termination: first, program & data are swapped out, while the PCB is retained in an “Exit” pool, then removed

Transition diagram of a five-state (New/Exit) model

2.a Process Description & Control

Process states

- Problems with the "Blocked/Ready" model
  - ✓ blocked processes are taking up memory space
  - ✓ a hungry CPU might soon run out of ready processes in memory



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

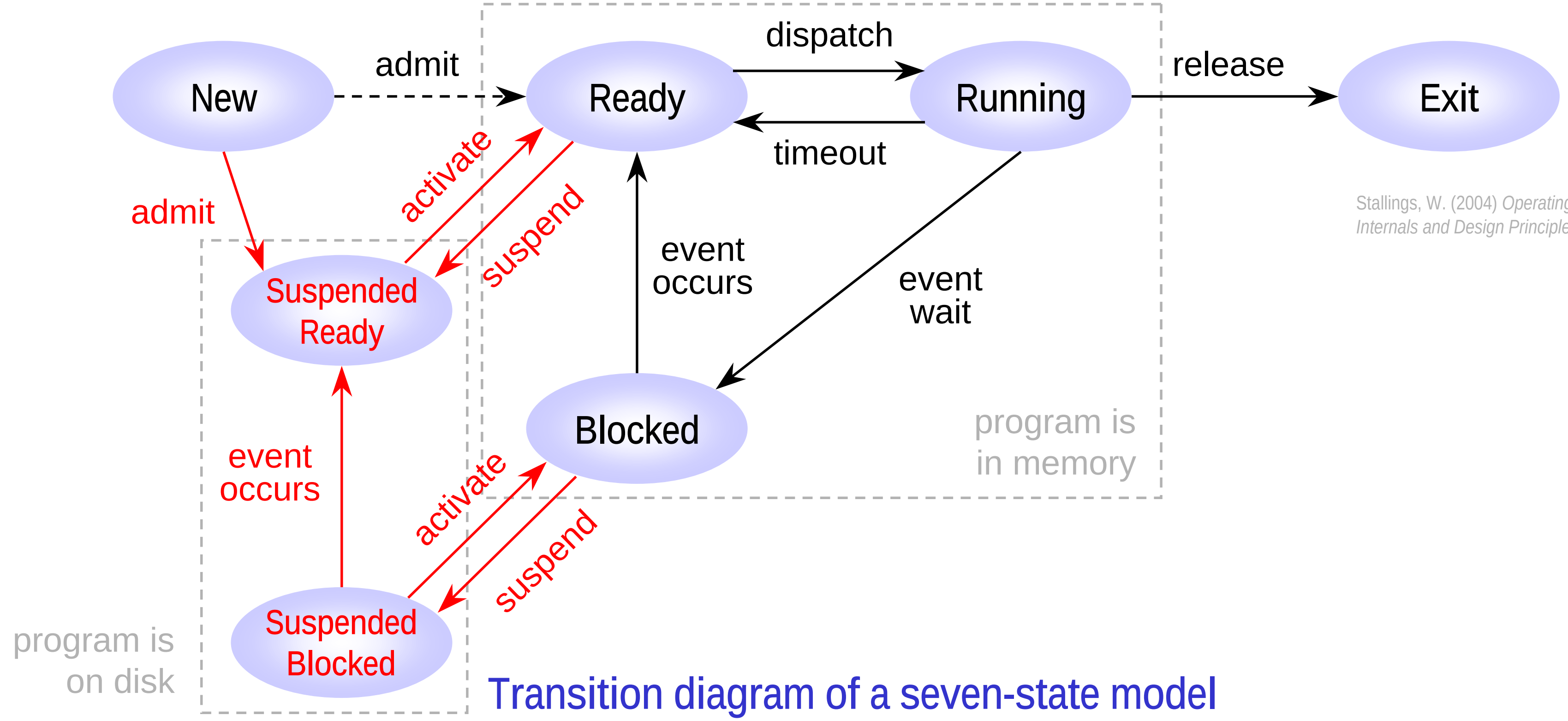
→ solution: swap processes out of memory and put them into a "Suspended" state

Transition diagram of a six-state ("Suspended") model

2.a Process Description & Control

Process states

- Last problem with the "Suspended" model
  - ✓ why swap in a suspended process that was blocked anyway?
  - → solution: add a "Suspended Ready" state



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Transition diagram of a seven-state model

2.a Process Description & Control

Process states

- Two independent concepts × two values each
  - ✓ whether a process is waiting on an event (is "Blocked") or not
  - ✓ whether a process has been swapped out of main memory (is "Suspended") or not
- = Four combined states
  - ✓ "Ready": the process is in memory and available for execution
  - ✓ "Blocked": the process is in main memory awaiting an event
  - ✓ "Suspended Blocked": the process is in secondary memory and awaiting an event
  - ✓ "Suspended Ready": the process is in secondary memory but is available for execution as soon as it is loaded into memory

2.a Process Description & Control

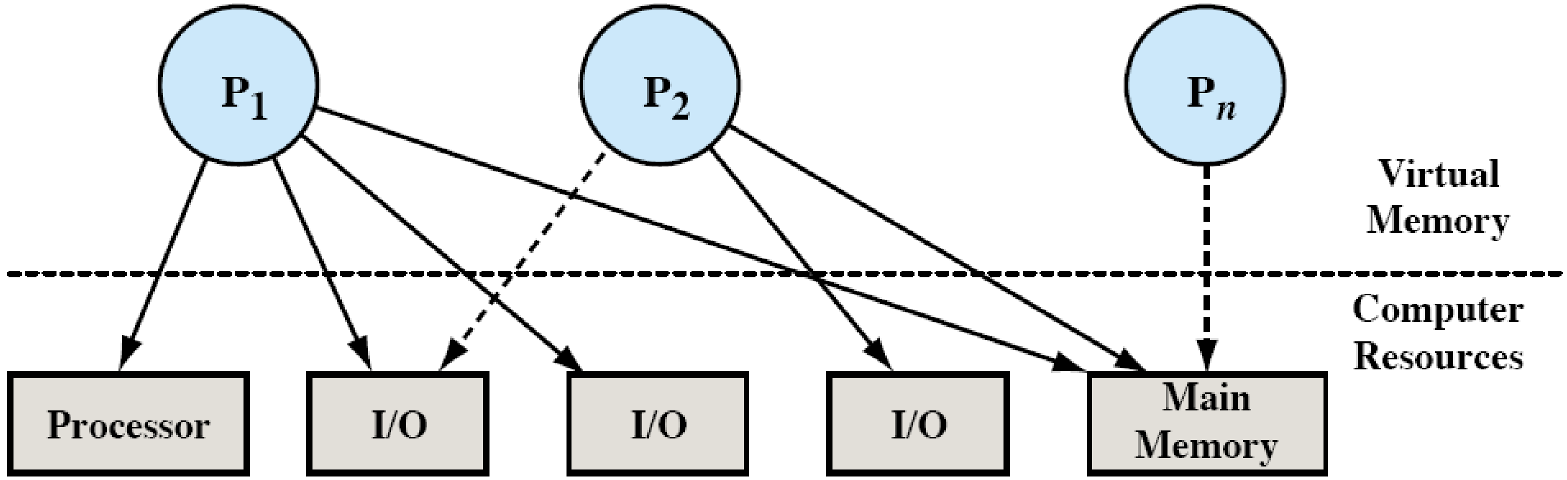
Process states

Note: Release of memory by swapping is not the only motivation for suspending processes. Various background processes may also be turned off and on, depending on CPU load, suspicion of a problem, some periodical timer or by user request.

2.a Process Description & Control

Process description

- **The O/S has to multiplex resources to the processes**
  - ✓ a number of processes have been created
  - ✓ each process during the course of its execution needs access to system resources: CPU, main memory, I/O devices



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Resource allocation for processes (one snapshot in time)

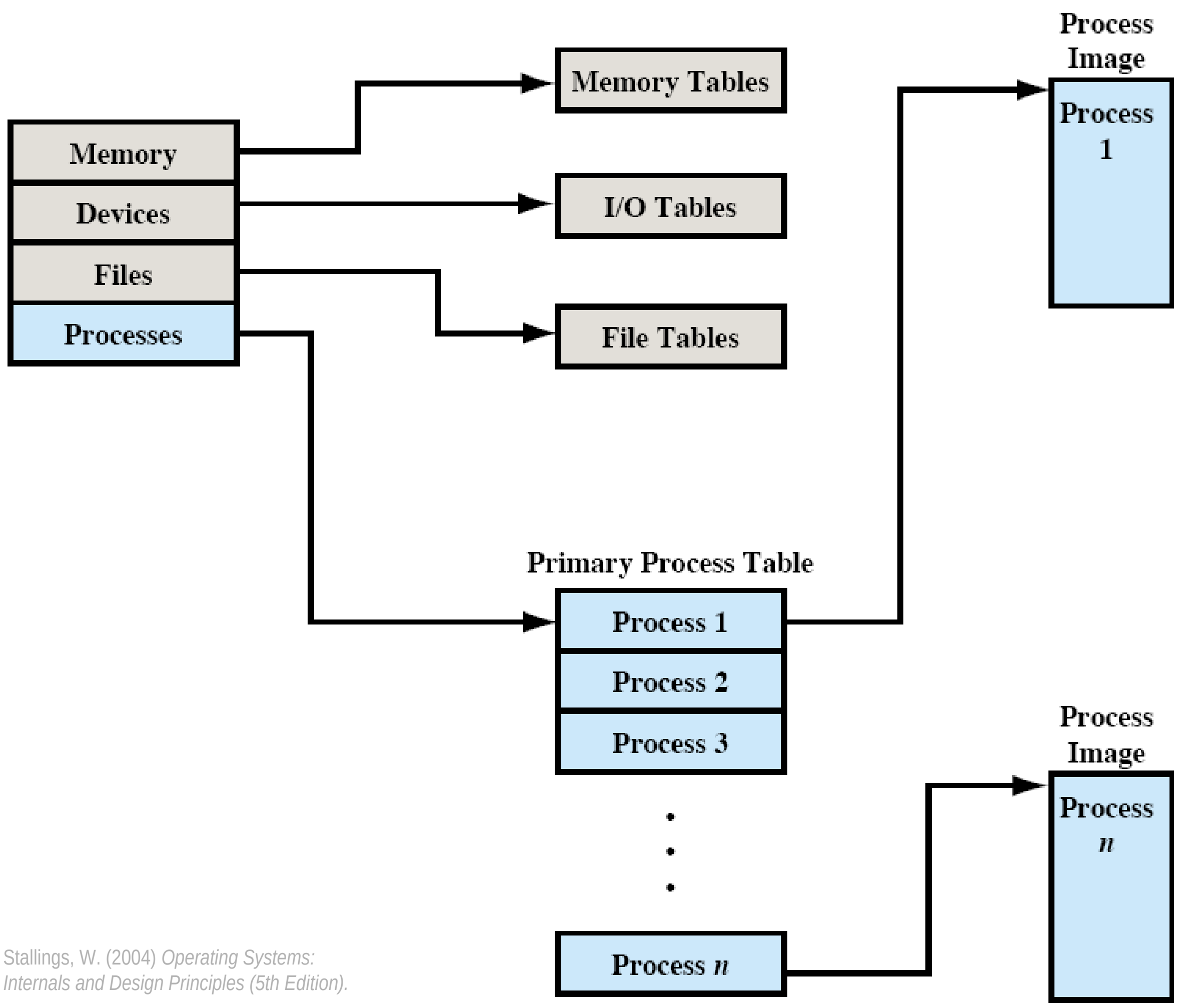
2.a Process Description & Control
Process description

- To do this, the O/S must be a zealous bureaucrat keeping all sorts of tables
  - ✓ **memory tables** – what part of memory is currently reserved for what process
  - ✓ **I/O tables** – what I/O device is currently assigned to what process
  - ✓ **file tables** – what file is currently opened by what process
  - ✓ **process tables** – what are the processes running, blocked, suspended, etc.
- Naturally, these tables are cross-referenced in many ways

Carmen Tomfohrde - *Three-ring binders*

2.a Process Description & Control

Process description



Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

General structure of an operating system's control tables

2.a Process Description & Control

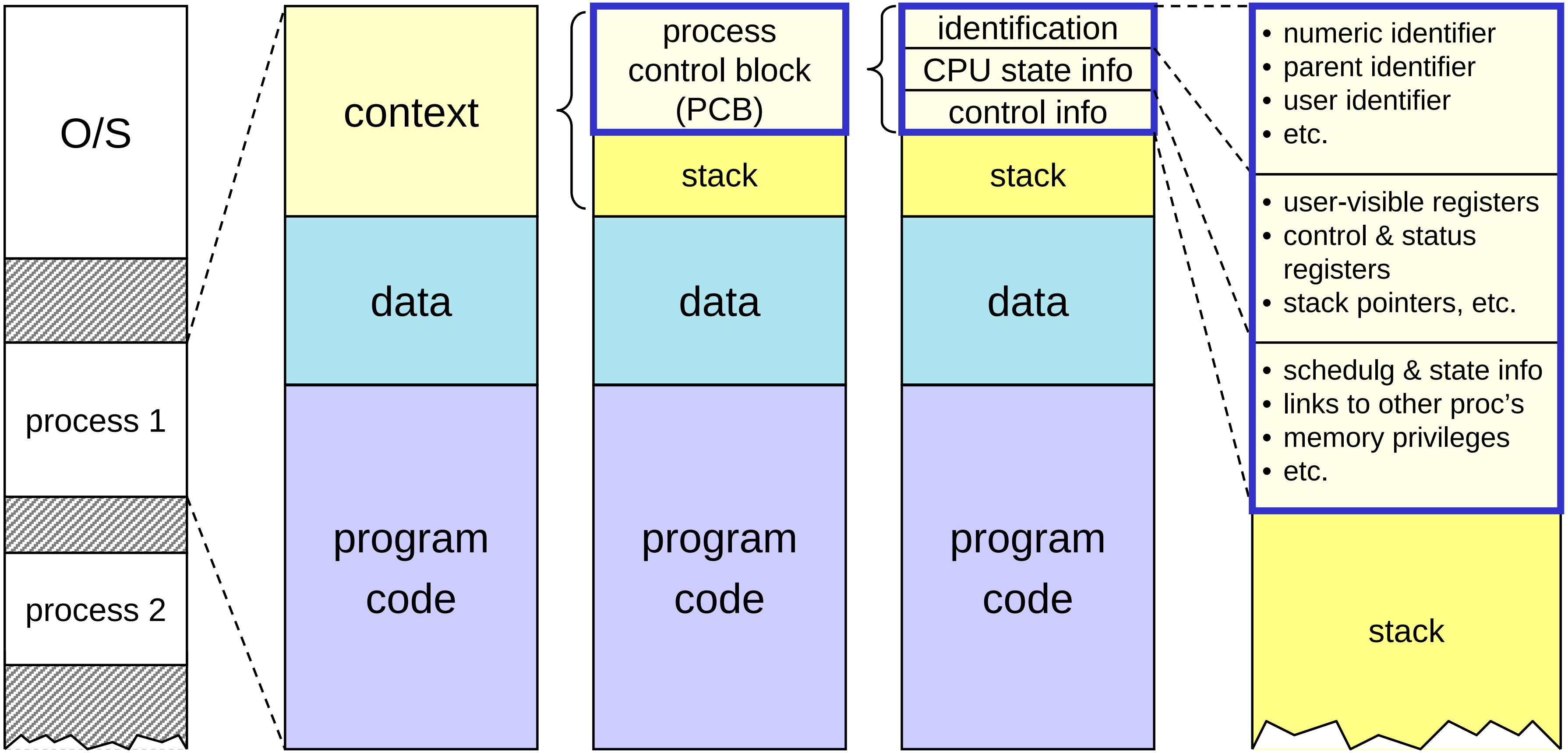
Process description

- In the process table, the O/S keeps one ID structure per process, the *Process Control Block* (PCB), containing:
  - ✓ process identification data
    - numeric identifiers of the process, the parent process, the user, etc.
  - ✓ CPU state information
    - user-visible, control & status registers
    - stack pointers
  - ✓ process control information
    - scheduling: state, priority, awaited event
    - used memory and I/O, opened files, etc.
    - pointer to next PCB

2.a Process Description & Control

Process description

- Example of process and PCB location in memory



Illustrative contents of a process image in (virtual) memory

2.a Process Description & Control

Process description

Note: In reality, depending on the specific O/S:

- PCB, stack, and user address space may be laid out in a different order
- within user space, data and program may be mixed.

Moreover:

- the process image may not be present in physical memory in its entirety
- the portion of process image in memory may not be contiguous, but distributed over disjoint address areas (“pages”).

We will meet the last two concepts again when we study **virtual memory**.

2.a Process Description & Control

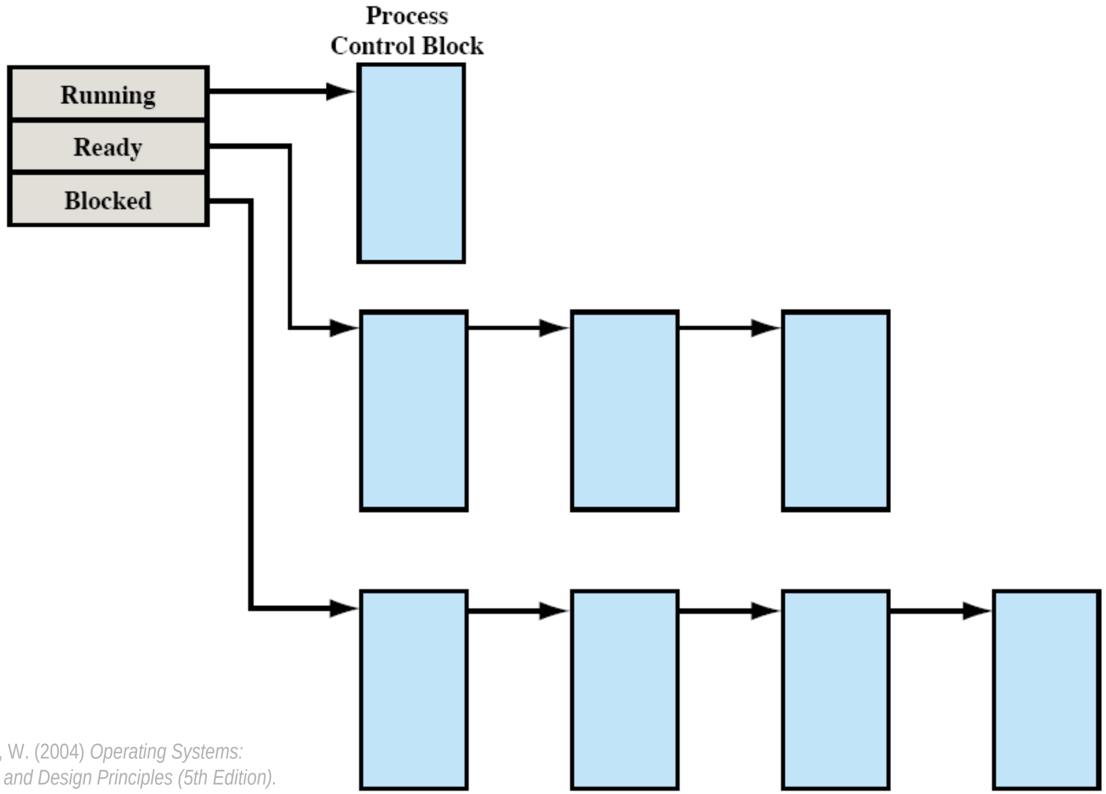
Process description

- The PCB is the most important O/S data structure
  - ✓ the set of PCBs (the process table) practically defines the state of the O/S
  - ✓ PCBs must be read/modified all the time by almost all modules in the O/S: scheduler, resource allocator, interrupt handler, performance monitor, etc.
  - ✓ therefore it is a good design practice to dedicate one low-level handler ("clerk") to the protection of the process table; then, the modules must ask this handler for any read/write access
  - ✓ we have seen this design pattern before: encapsulate a critical resource in a service layer or module for better control and orderly access; this is the whole story of an O/S!

2.a Process Description & Control

Process description

- The process table can be split into per-state queues
  - ✓ PCBs can be linked together if they contain a pointer field



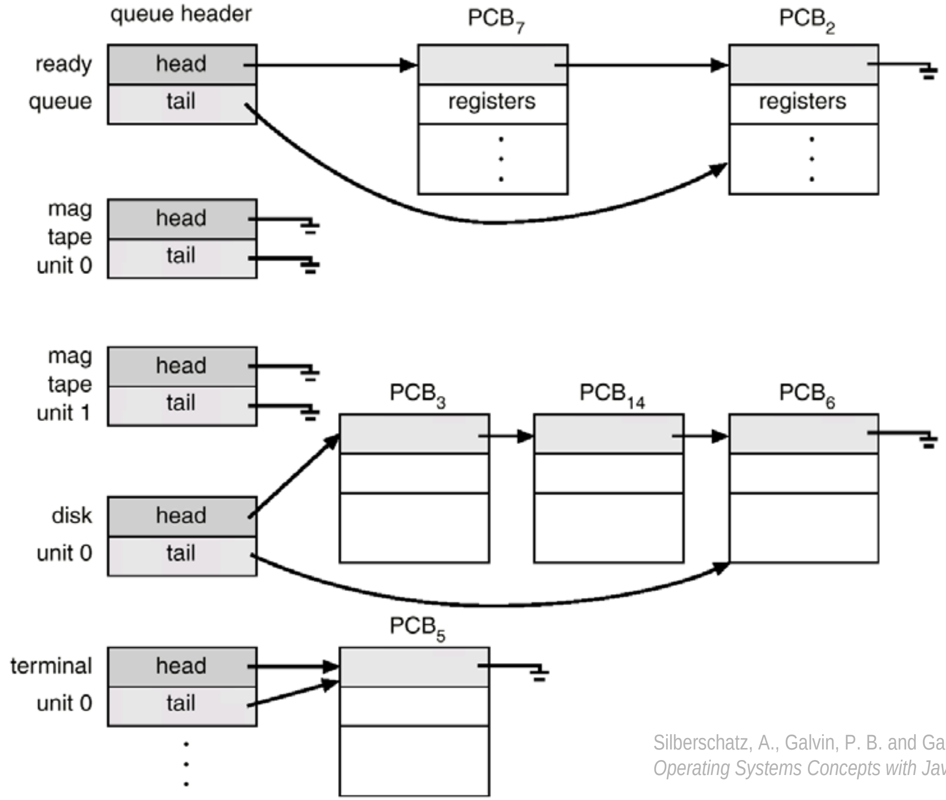
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Structure of process lists or queues

2.a Process Description & Control

Process description

- The blocked processes can themselves be split into device-specific queues



Silberschatz, A., Galvin, P. B. and Gagne. G. (2003)
Operating Systems Concepts with Java (6th Edition).

Various I/O device queues

2.a Process Description & Control

Process description

```
struct task_struct
{
    volatile long state;          /* -1 unrunnable, 0 runnable, >0 stopped */
    unsigned long flags;          /* per process flags, defined below */
    ...
    struct mm_struct *mm;         /* memory */
    ...
    struct task_struct *next_task, *prev_task;   /* linked list */
    ...
    struct linux_binfmt *binfmt;  /* task state */
    int exit_code, exit_signal;
    ...
    pid_t pid;                    /* process ID */
    pid_t pgrp;                   /* process group ID */
    ...
    /*
     * pointers to parent process, youngest child, younger sibling,
     * older sibling, respectively.
     */
    struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
    ...
    struct thread_struct thread;  /* CPU-specific state of this task */
    ...
    struct files_struct *files;   /* open file information */
    ...
}
```

Sample of the PCB data structure `task_struct` in Linux

http://lxr.linux.no

2.a Process Description & Control

Process control

- How is a process created by the O/S, step by step?
  1. a unique identifier is assigned to the new process
     - one new entry is added to the primary process table
  2. memory space is allocated for the process
     - this includes program (with linkages), data, stack and PCB
  3. the PCB is constructed and initialized
     - ID, state = “Ready”, CPU state = empty, resources = none
  4. the PCB is placed in the appropriate queue (linked list)
  5. other O/S modules are notified about the new process
     - create or expand other data structures to accommodate info about the new process

2.a Process Description & Control

Process control

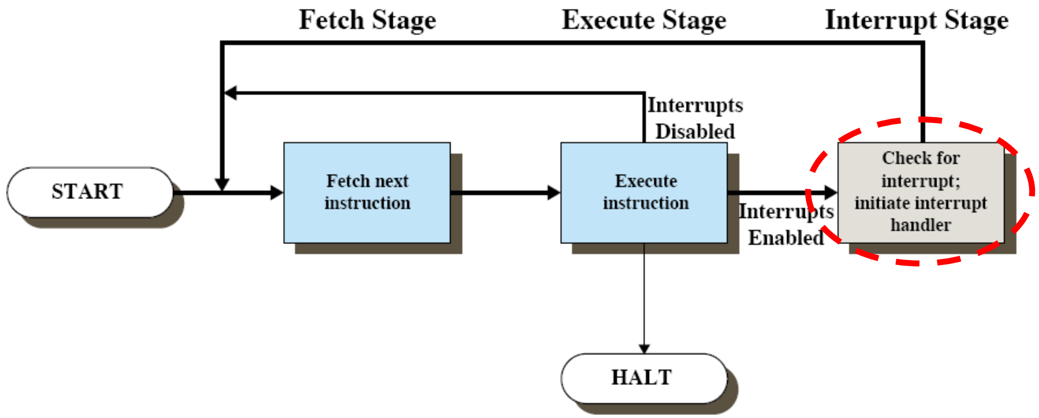
- What events trigger the O/S to switch processes?
  - ✓ **interrupts** — external, asynchronous events, independent of the currently executed process instructions
    - clock interrupt → O/S checks time and may block process
    - I/O interrupt → data has come, O/S may unblock process
    - memory fault → O/S may block process that must wait for a missing page in memory to be swapped in
  - ✓ **exceptions** — internal, synchronous (but involuntary) events caused by instructions → O/S may terminate or recover process
  - ✓ **system calls** — voluntary synchronous events calling a specific O/S service → after service completed, O/S may either resume or block the calling process, depending on I/O, priorities, etc.

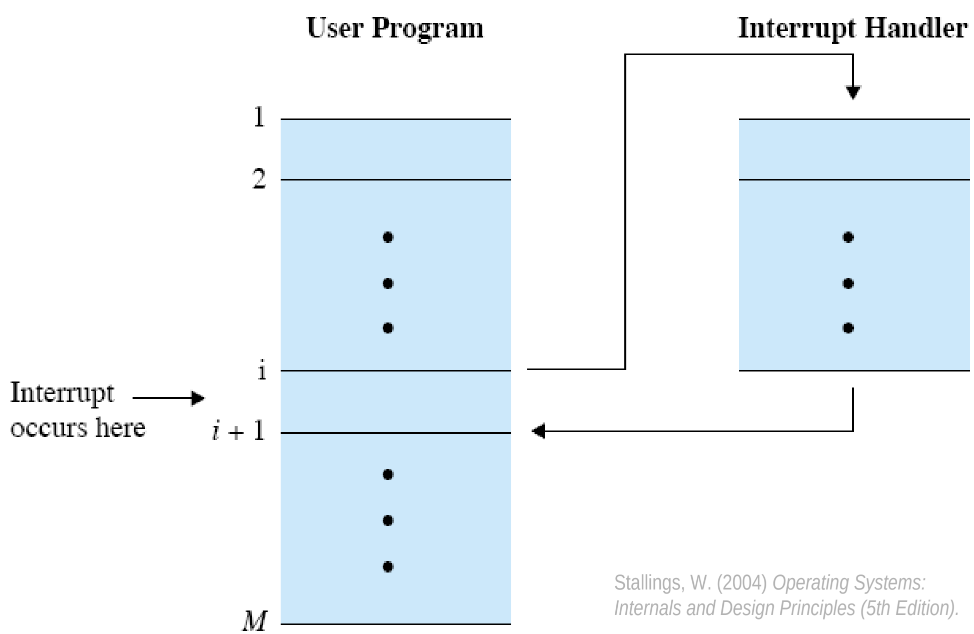
traps (exceptions and system calls)

2.a Process Description & Control

Process control

- **Interrupts or traps**
  - ✓ are caught in a third stage of the fetch/ execute cycle and
  - ✓ transfer control (PC) to an interrupt handler in kernel space,
  - ✓ which branches to O/S routines specific to types of interrupts;
  - ✓ the CPU is eventually returned to this user program . . . or another



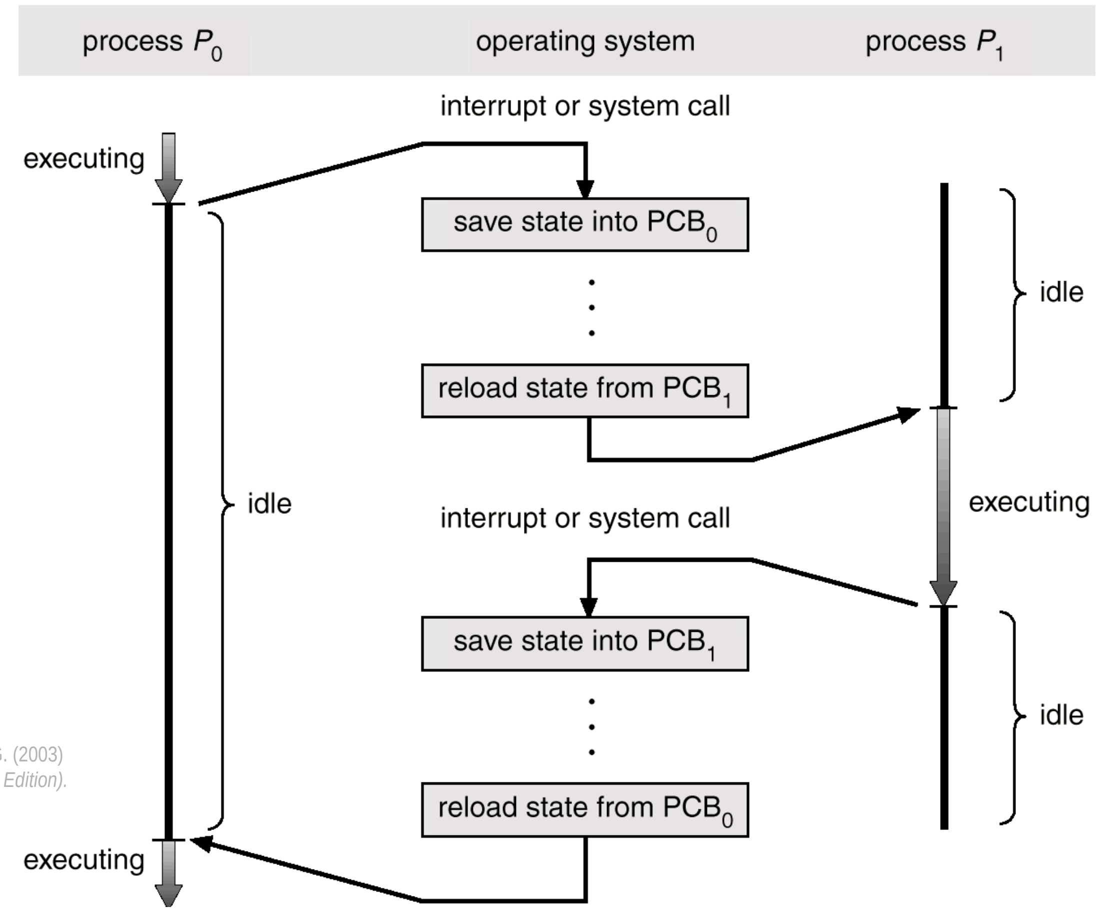


Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

2.a Process Description & Control

Process control

- Process switch



Silberschatz, A., Galvin, P. B. and Gagne. G. (2003)
Operating Systems Concepts with Java (6th Edition).

CPU switch from process to process

2.a Process Description & Control
Process control

- **Mode switching ≠ process switching**
  - ✓ when handling an interrupt, execution is always switched from user mode to kernel mode (“mode switch”)
  - ✓ but this is independent from whether the O/S will return control to the interrupted process or another process (“process switch”)

1. if control (execution) eventually returns to the interrupted process, for example after a nonblocking system call:
   - only the CPU state information (PC, registers, stack info) needed to be saved; this was initiated by the hardware
2. if control eventually passes to another process, for example after a blocking call, interrupt or trap:
   - the whole PCB is saved; this is done by the O/S scheduler

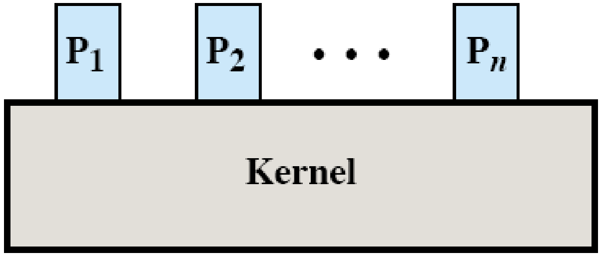
2.a Process Description & Control
Process control

- How does a full process switch happen, step by step?
  1. save CPU context, including PC and registers *(the only step needed in a simple mode switch)*
  2. update process state (to "Ready", "Blocked", etc.) and other related fields of the PCB
  3. move the PCB to the appropriate queue
  4. select another process for execution: this decision is made by the CPU scheduling algorithm of the O/S
  5. update the PCB of the selected process (state = "Running")
  6. update memory management structures
  7. restore CPU context to the values contained in the new PCB

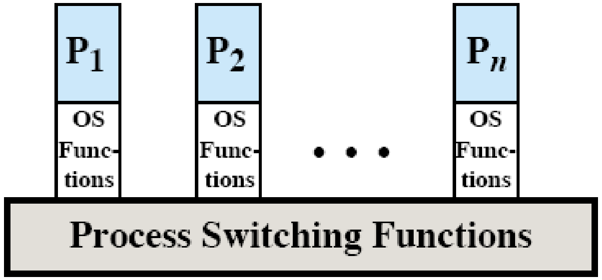
2.a Process Description & Control

Process control

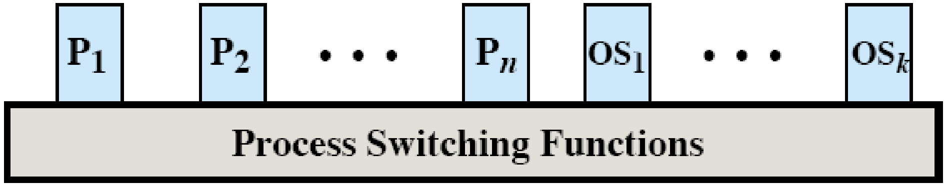
- How is the O/S itself executed? Is it a process, too?



(a) Separate kernel



(b) O/S functions execute within user processes



(c) O/S functions execute as separate processes

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Relationship between O/S execution and user processes

2.a Process Description & Control

Process control

- Possible designs for the execution of the O/S itself
  - ✓ nonprocess kernel (traditional approach in older O/S)
    - simple mode switch; kernel executes in own region of memory with own stack, outside of any process (i.e., no associated PCB); the only program that is not a "process"
  - ✓ O/S functions execute within each user process (most PCs)
    - the O/S is a collection of routines that can be "attached" to the processes in memory via shared address space
    - only the mode is switched, the current process (which executes user program + kernel program) continues to run
  - ✓ O/S functions execute as full, separate processes (microkernels)
    - modular O/S with clean, minimal interfaces

Principles of Operating Systems
CS 446/646

2. Processes

a. **Process Description & Control**
   - ✓ What is a process?
   - ✓ Process states
   - ✓ Process description
   - ✓ Process control

b. **Threads**

c. **Concurrency**

d. **Deadlocks**

Principles of Operating Systems

CS 446/646

2. Processes

a. Process Description & Control

b. **Threads**

  - ✓ Separation of resource ownership and execution
  - ✓ It's the same old throughput story, again
  - ✓ Practical uses of multithreading
  - ✓ Implementation of threads

c. **Concurrency**

d. **Deadlocks**

2.b Threads

Separation of resource ownership and execution

- In fact, a process embodies two independent concepts
  1. resource ownership
  2. execution & scheduling

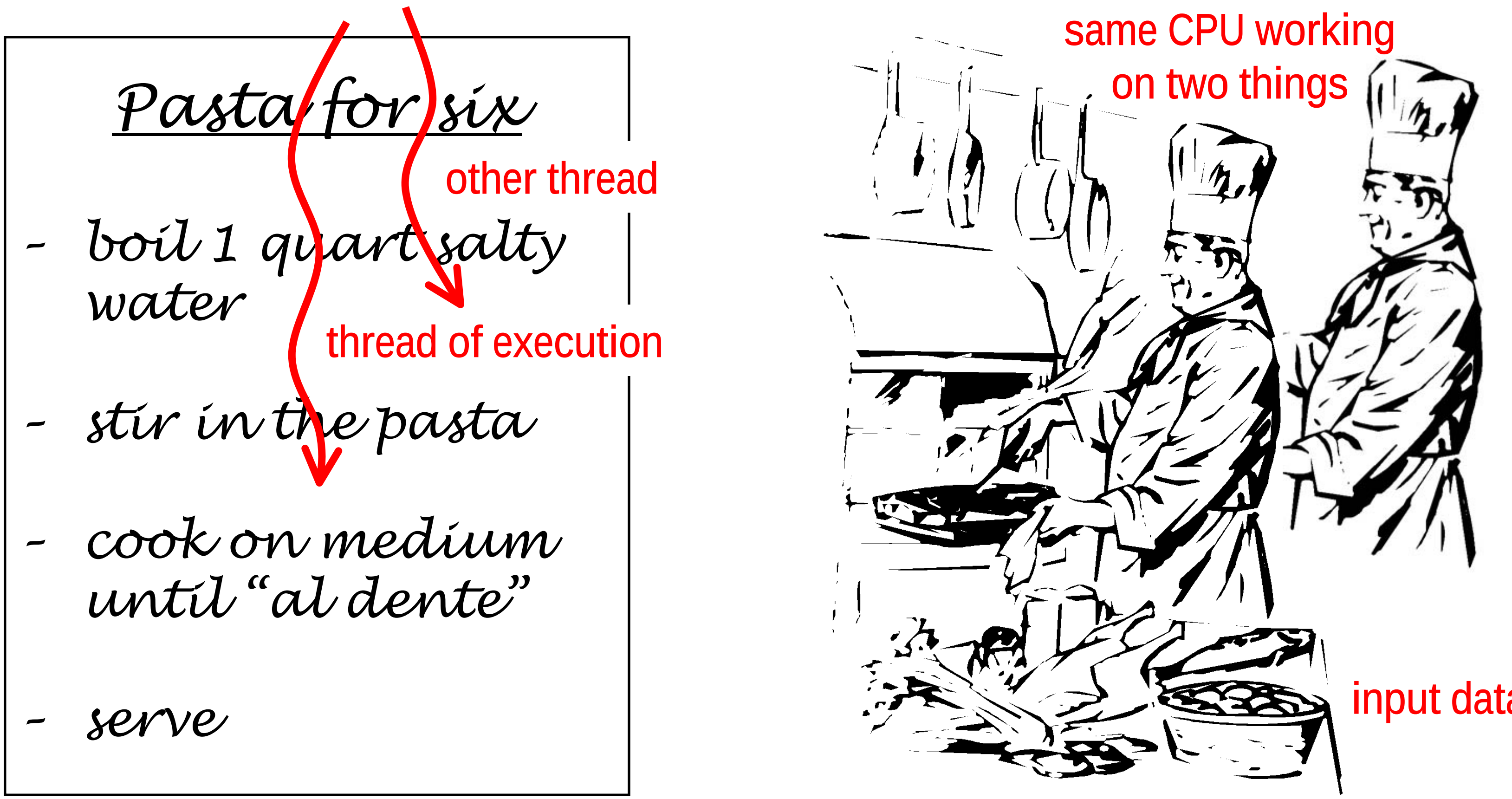
1. Resource ownership
   - ✓ a process is allocated address space to hold the image, and is granted control of I/O devices and files
   - ✓ the O/S prevents interference among processes while they make use of resources (multiplexing)

2. Execution & scheduling
   - ✓ a process follows an execution path through a program
   - ✓ it has an execution state and is scheduled for dispatching

2.b Threads

Separation of resource ownership and execution

- The execution part is a “thread” that can be multiplied



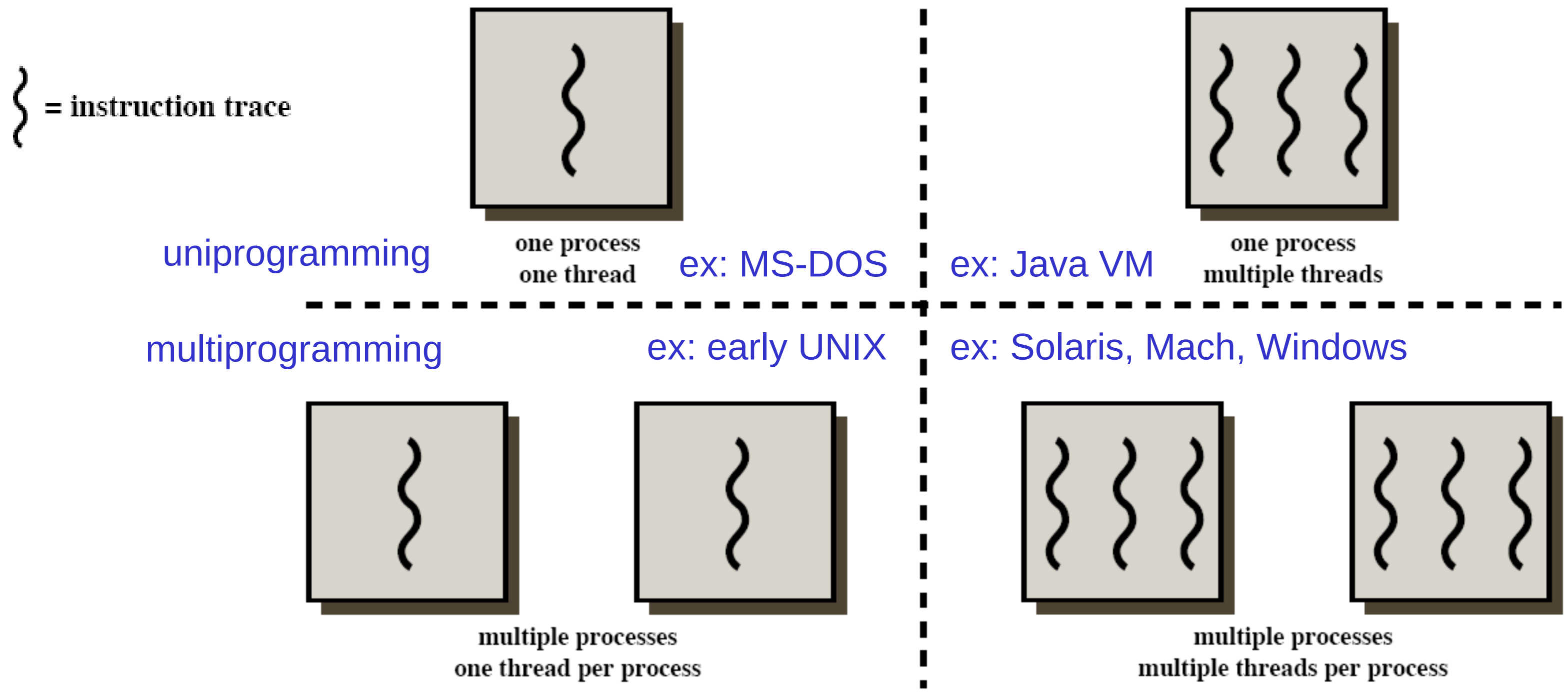
Program

Process

2.b Threads

Separation of resource ownership and execution

- **Multithreading**
  - ✓ refers to the ability of an operating system to support multiple threads of execution within a single process



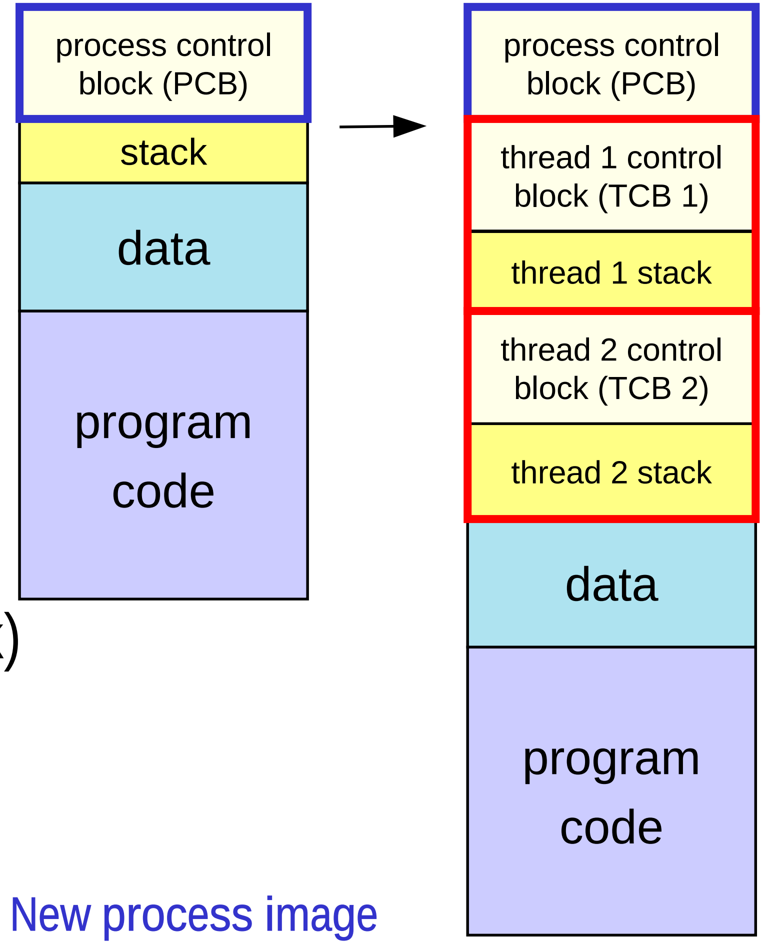
Process-thread relationships

Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

2.b Threads

Separation of resource ownership and execution

- Multithreading requires changes in the process description model
  - ✓ each thread of execution receives its own control block and stack
    - own execution state ("Running", "Blocked", etc.)
    - own copy of CPU registers
    - own execution history (stack)
  - ✓ the process keeps a global control block listing resources currently used



New process image

2.b Threads

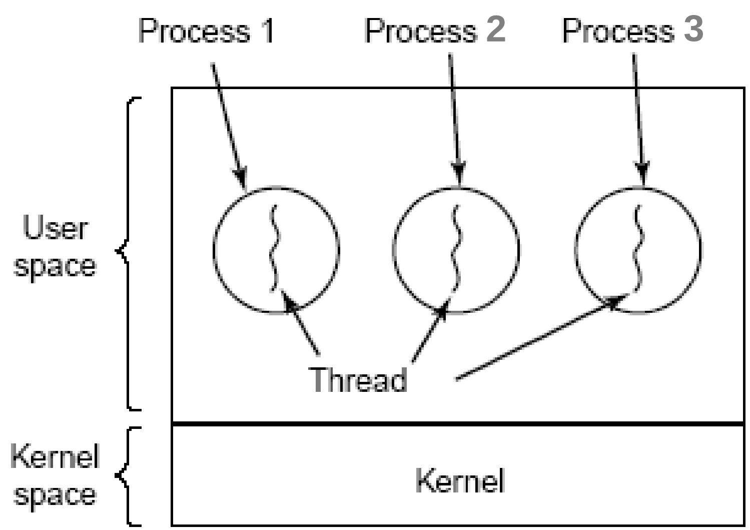
Separation of resource ownership and execution

- Per-process items and per-thread items in the control block structures
  - ✓ process identification data + thread identifiers
    - numeric identifiers of the process, the parent process, the user, etc.
  - ✓ CPU state information
    - user-visible, control & status registers
    - stack pointers
  - ✓ process control information
    - scheduling: state, priority, awaited event
    - used memory and I/O, opened files, etc.
    - pointer to next PCB

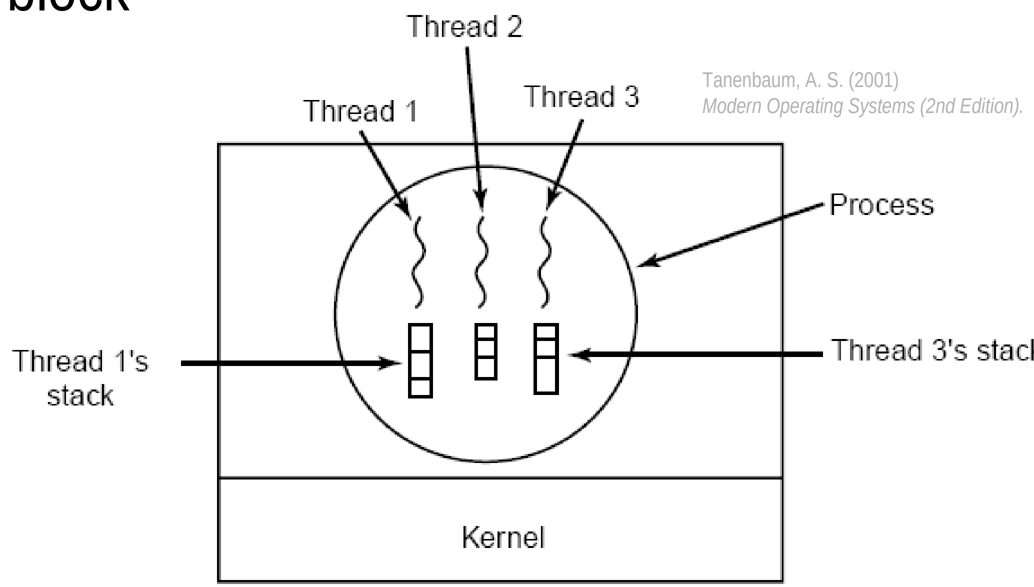
2.b Threads

Separation of resource ownership and execution

- **Multithreaded process model**
  - ✓ all threads share the <u>same address space and resources</u>
  - ✓ spawning a new thread only involves allocating a new stack and a new CPU state block



(a) Three processes with one thread



(a) One process with three threads

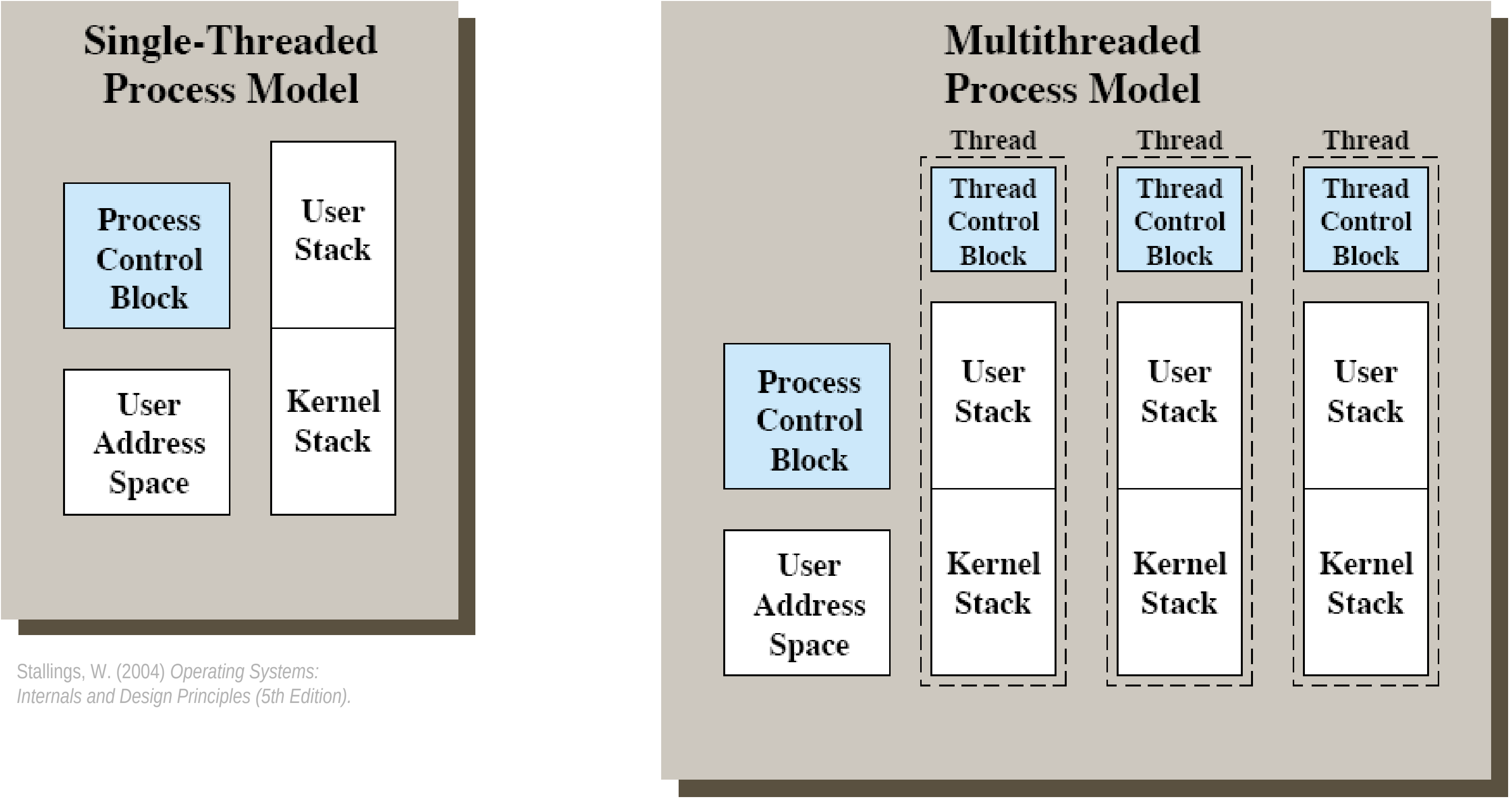
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

Single-threaded and multithreaded process models (in abstract space)

2.b Threads

Separation of resource ownership and execution

- Multithreaded process model (another view)



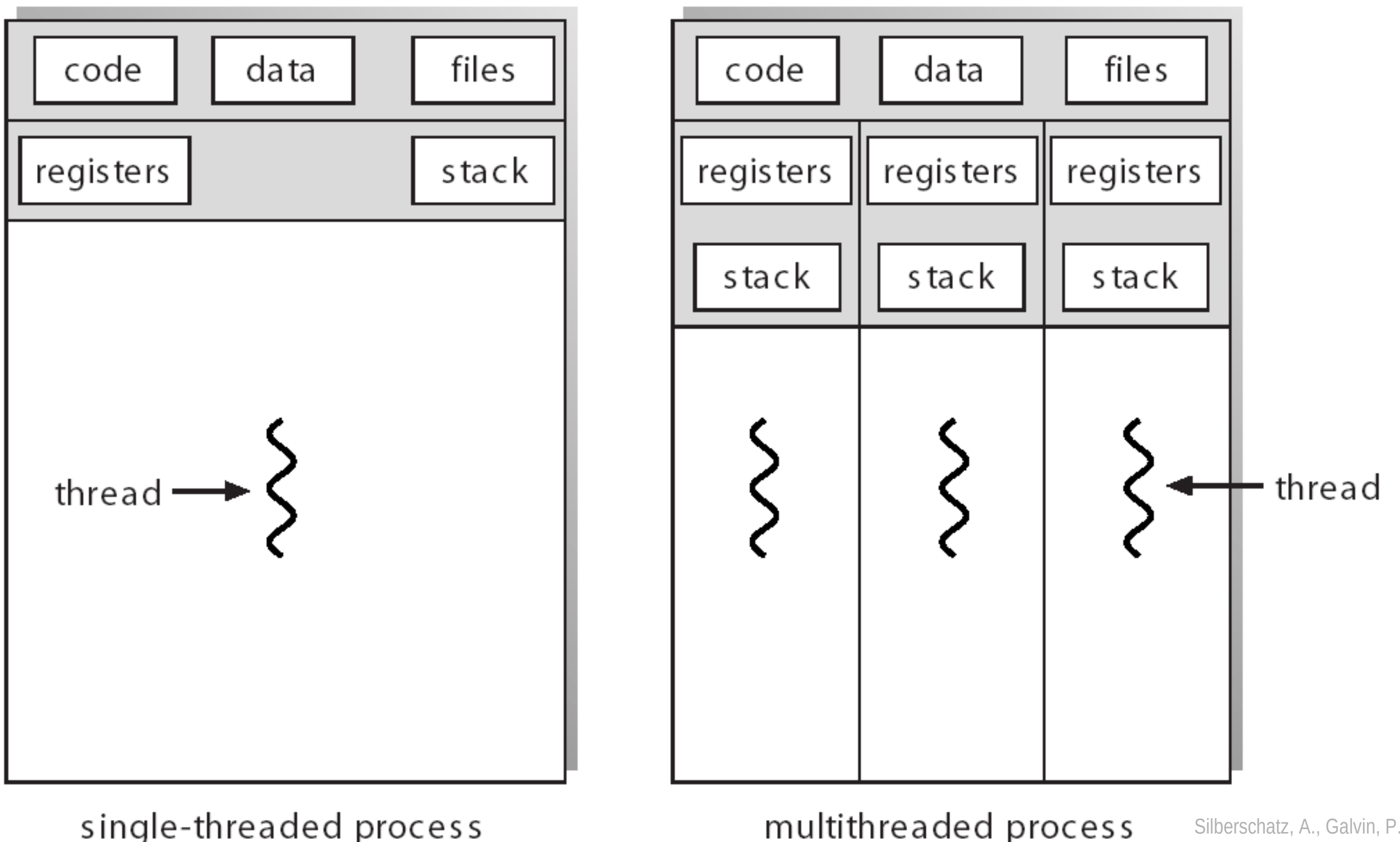
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Single-threaded and multithreaded process models (in abstract space)

2.b Threads

Separation of resource ownership and execution

- Multithreaded process model (yet another view)



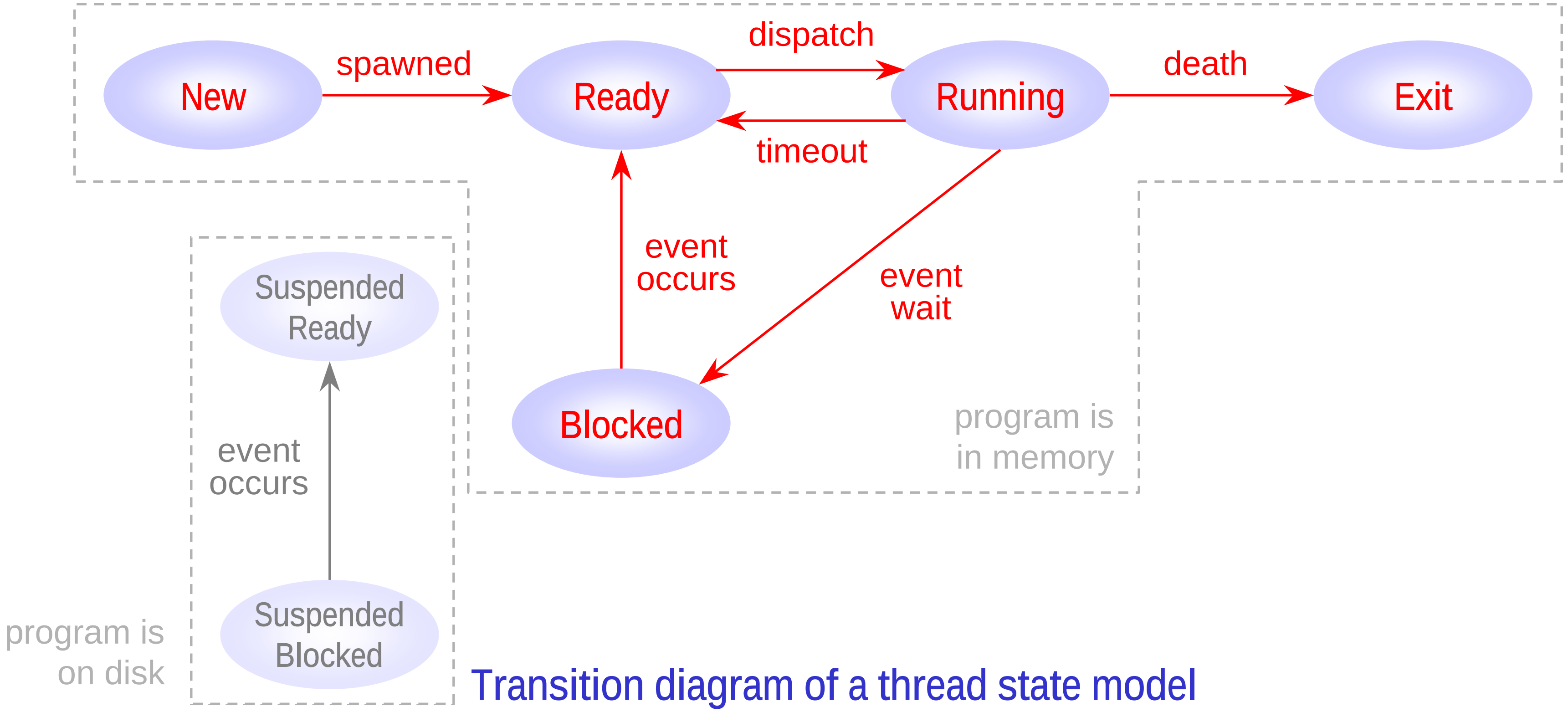
Silberschatz, A., Galvin, P. B. and Gagne. G. (2003)
Operating Systems Concepts with Java (6th Edition).

Single-threaded and multithreaded process models (in abstract space)

2.b Threads

Separation of resource ownership and execution

- **Possible thread-level states**
  - ✓ threads (like processes) can be ready, running or blocked
  - ✓ threads can't be suspended ("swapped out"), only processes can

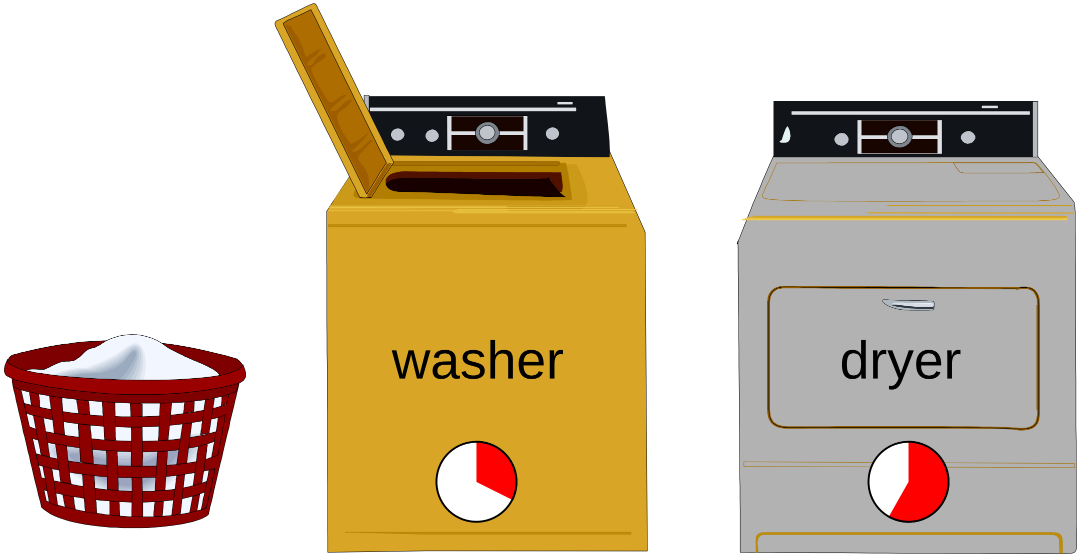


Transition diagram of a thread state model

2.b Threads

It's the same old throughput story, again

- In the laundry room
  - ✓ the washing machine takes 20 minutes
  - ✓ the dryer takes 40 minutes



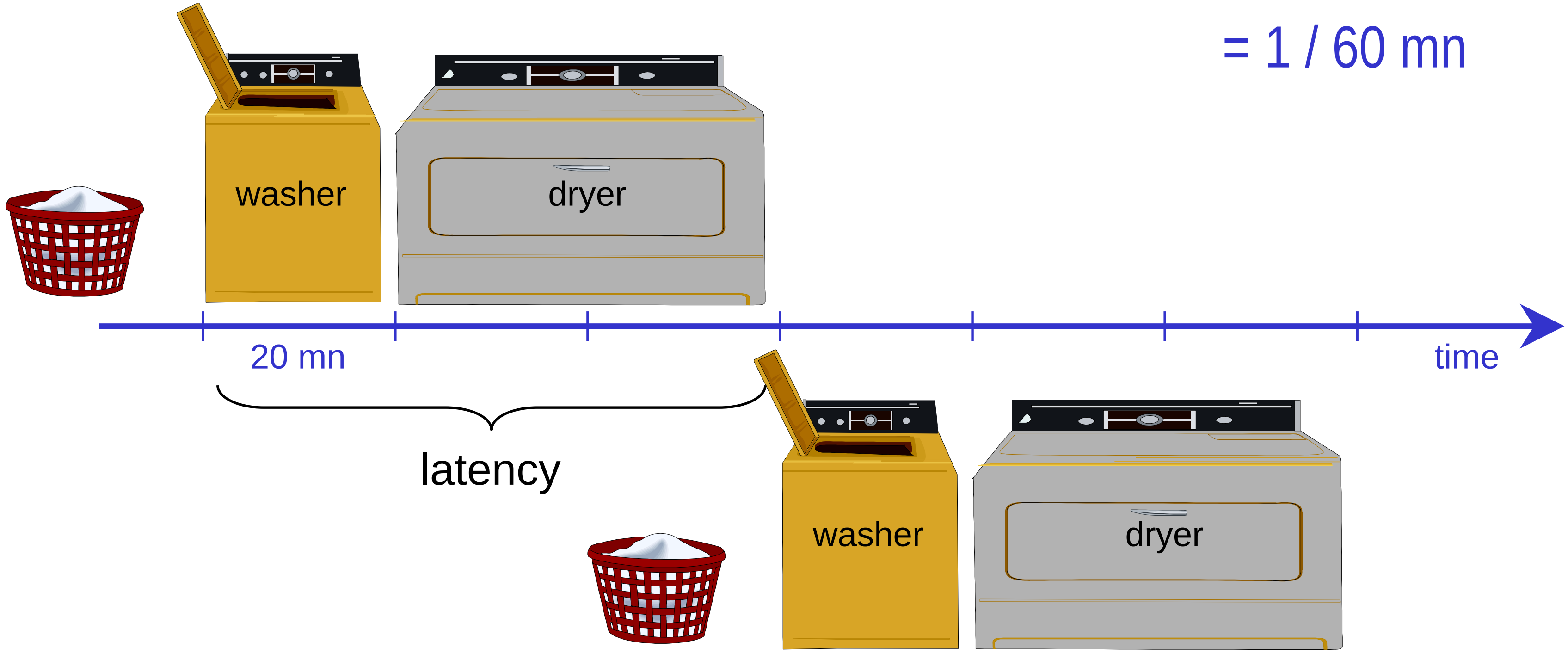
after Gill Pratt (2000) *How Computers Work*. ADUni.org/courses.

2.b Threads

It's the same old throughput story, again

- Doing two loads in a sequence
  - ✓ **latency** = time for one execution to complete = 60 mn
  - ✓ **throughput** = rate of completed executions = 2 / 120 mn

= 1 / 60 mn



Two loads in a sequence

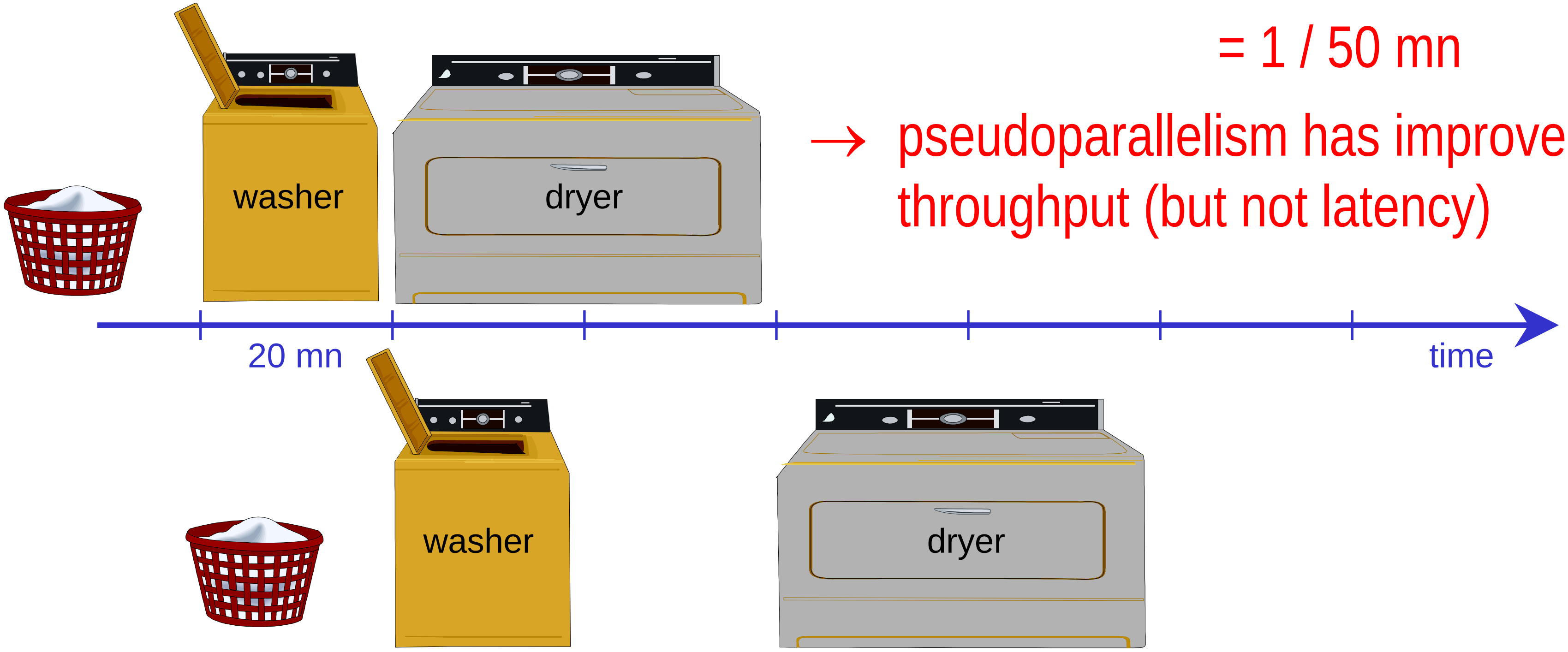
2.b Threads

It's the same old throughput story, again

- Doing two loads in (pseudo)parallel
  - ✓ **latency** = time for one execution to complete = 60 to 80 mn
  - ✓ **throughput** = rate of completed executions = 2 / 100 mn

= 1 / 50 mn

→ pseudoparallelism has improved throughput (but not latency)

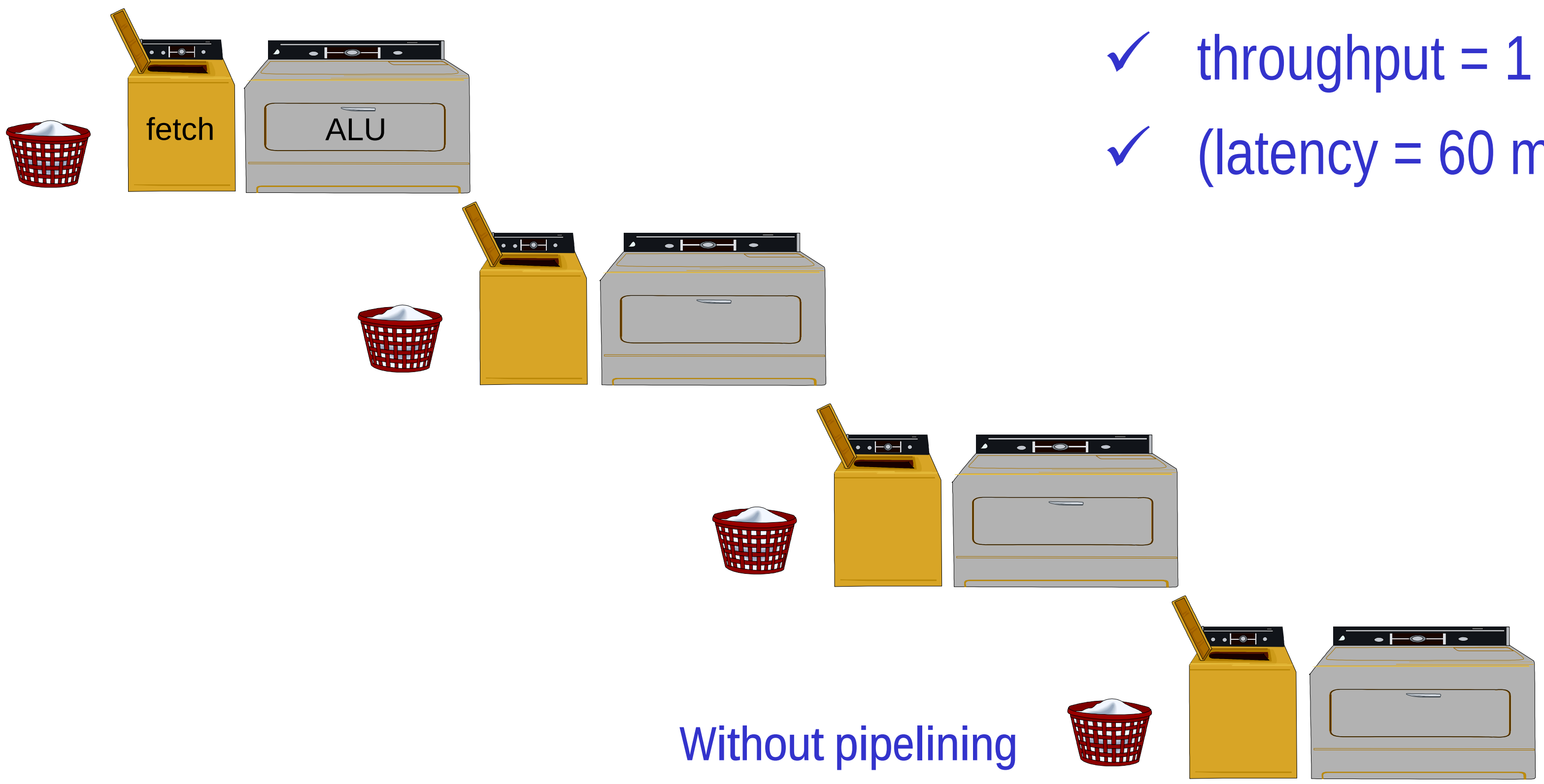


Two loads in parallel

2.b Threads

It's the same old throughput story, again

- This is the principle used in processor pipelining
  - ✓ here, washer & dryer are regularly clocked stages
  - ✓ without pipelining: throughput is 1 over the sum of all stages
    - ✓ throughput = 1 / 60 mn
    - ✓ (latency = 60 mn)



Without pipelining

2.b Threads

It's the same old throughput story, again

- This is the principle used in processor pipelining
  - ✓ here, washer & dryer are regularly clocked stages
  - ✓ with pipelining: throughput is only 1 over the longest stage

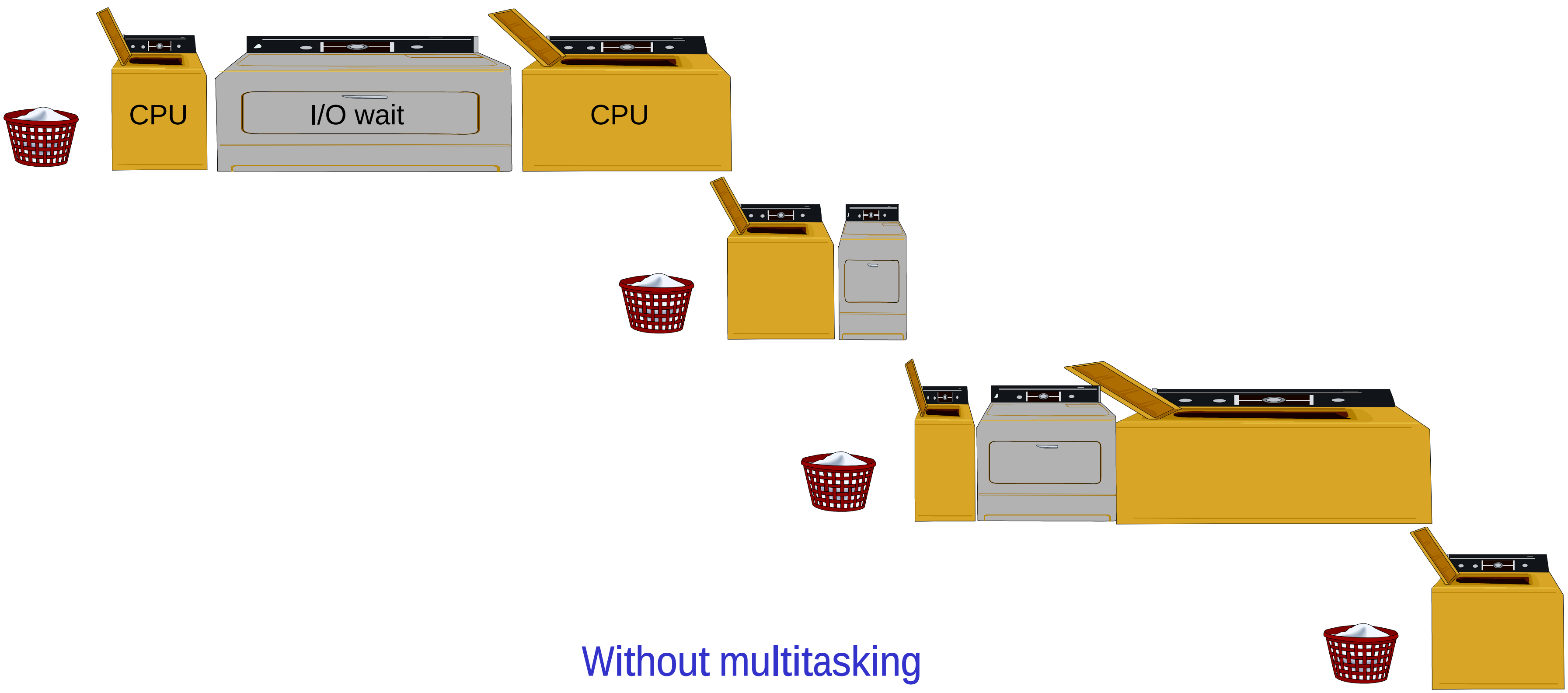
- ✓ throughput = 1 / 40 mn
- ✓ (but latency = 80 mn)

With pipelining

2.b Threads

It's the same old throughput story, again

- This is also the principle used in <u>multitasking</u>
  - ✓ here, the washer is the CPU and the dryer is one I/O device
  - ✓ wash & dry times may vary with loads and repeat in any order

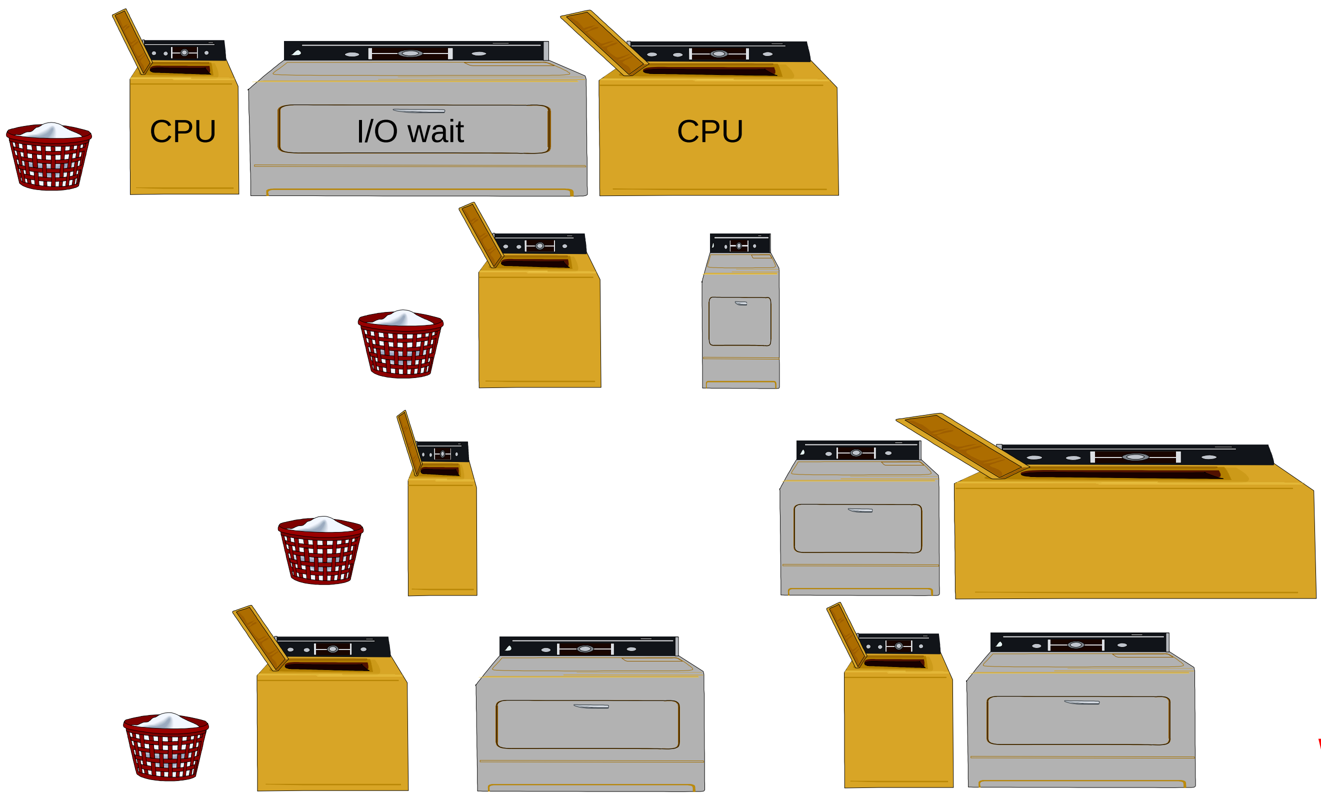


Without multitasking

2.b Threads

It's the same old throughput story, again

- This is also the principle used in <u>multitasking</u>
  - ✓ thanks to multitasking, throughput (CPU utilization) is much higher (but the total time to complete a process is also longer)

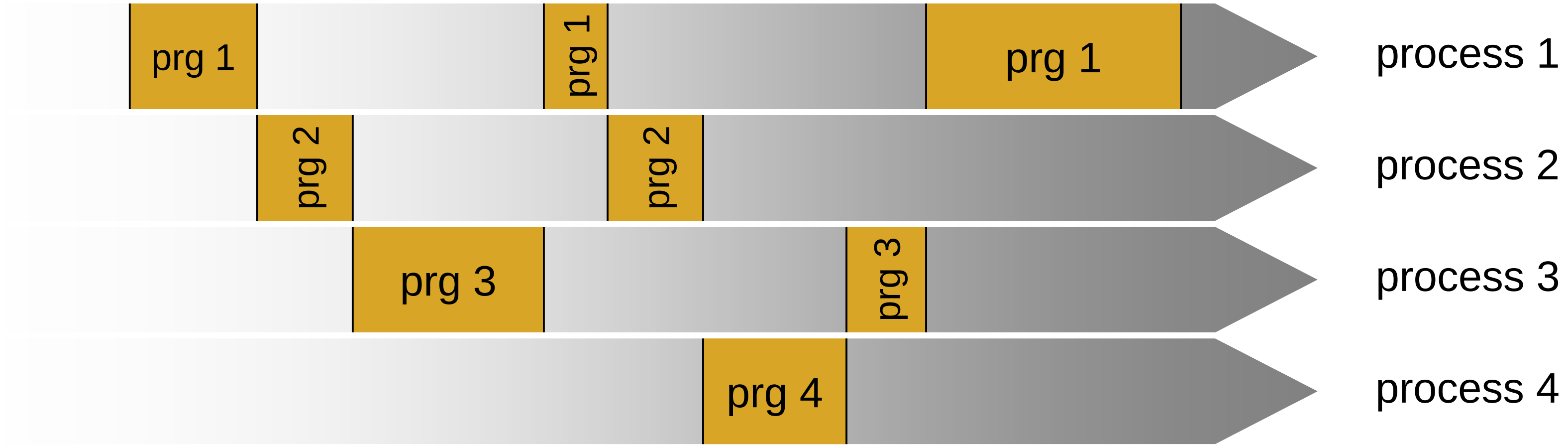


With multitasking

2.b Threads

It's the same old throughput story, again

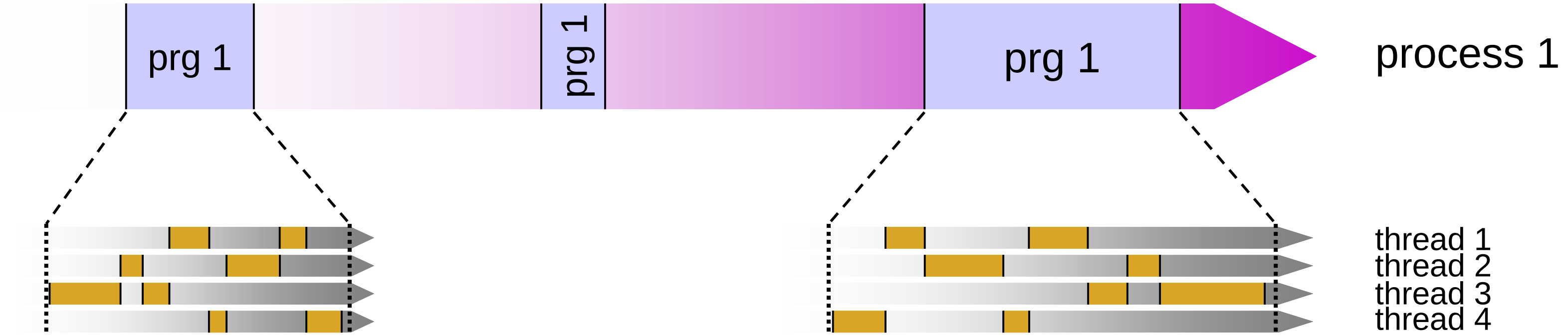
- This is also the principle used in <u>multitasking</u>



2.b Threads

It's the same old throughput story, again

- And, naturally, the same idea applies in <u>multithreading</u>
  - ✓ multithreading is basically the same as multitasking at a finer level of temporal resolution (and within the same address space)
  - ✓ the same illusion of parallelism is achieved at a finer grain

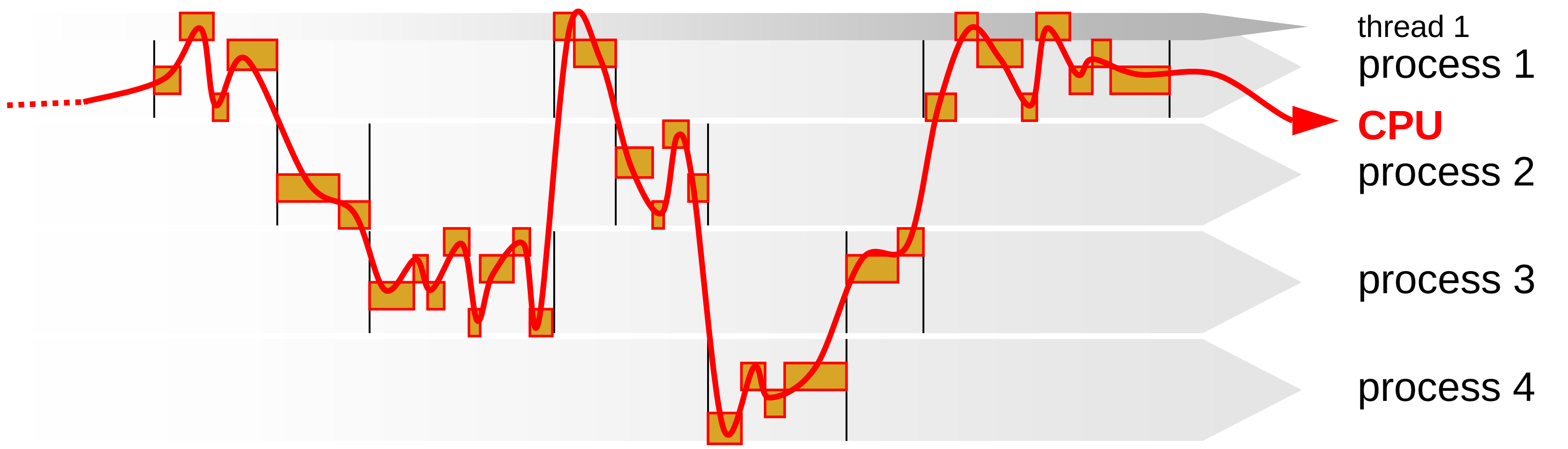


Multithreading

2.b Threads

It's the same old throughput story, again

- **And, naturally, the same idea applies in <u>multithreading</u>**
  - ✓ in a single-processor system, there is still only one CPU (washing machine) going through all the threads of all the processes



Multithreading

2.b Threads

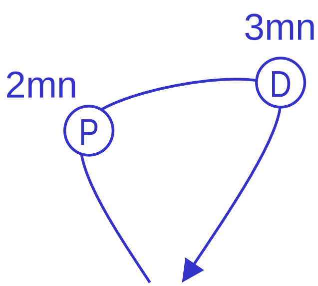
It's the same old throughput story, again

- From processes to threads: a shift of levels
  - ✓ container paradigm
    - there can be multiple processes running in one computer
    - there can be multiple threads running in one process
  - ✓ resource sharing paradigm
    - multiple processes share hardware resources: CPU, physical memory, I/O devices
    - multiple threads share process-owned resources: memory address space, opened files, etc.

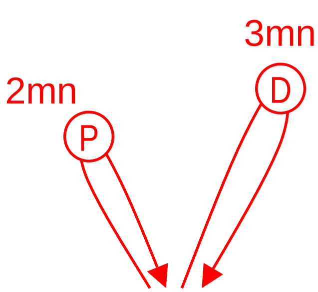
2.b Threads

Practical uses of multithreading

- Illustration: two shopping scenarios
  - ✓ Single-threaded shopping



    - you are in the grocery store
    - first you go to produce and grab salad and apples, then you go to dairy and grab milk, butter and cheese
    - it took you about 1mn x 5 items = 5mn
  - ✓ Multithreaded shopping



    - you take your two kids with you to the grocery store
    - you send them off in two directions with two missions, one toward produce, one toward dairy
    - you wait for their return (at the slot machines) for a maximum duration of about 1mn x 3 items = 3mn

2.b Threads

Practical uses of multithreading

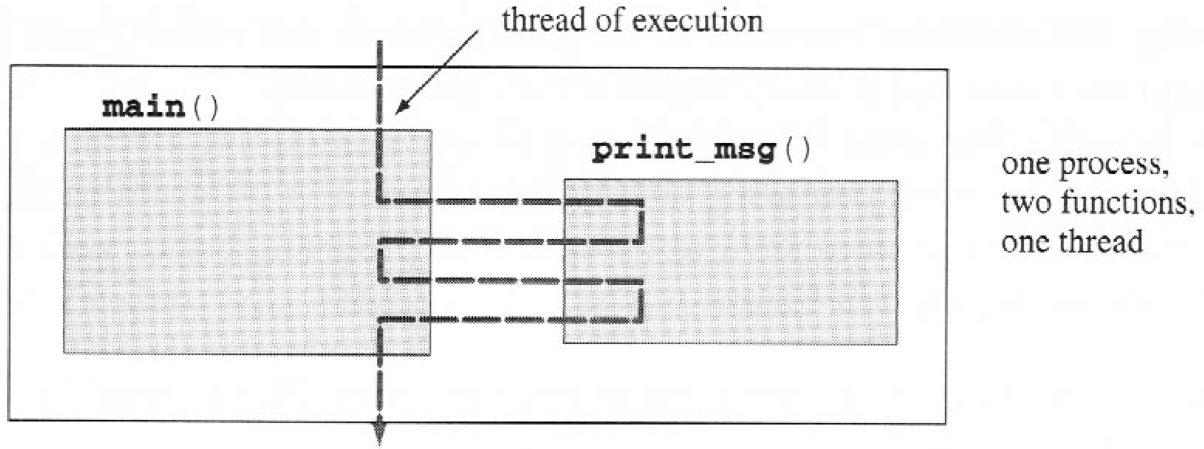
```
void main(...)
{
   char *produce[] = { "salad", "apples", NULL };
   char *dairy[] = { "milk", "butter", "cheese", NULL };

   print_msg(produce);
   print_msg(dairy);
}

void print_msg(char **items)
{
   int i = 0;
   while (items[i] != NULL) {
       printf("grabbing the %s...”, items[i++]);
       fflush(stdout);
       sleep(1);
   }
}
```

Single-threaded shopping code

2.b Threads

Practical uses of multithreading

- Results of single-threaded shopping
  - ✓ total duration ≈ 5 seconds; outcome is deterministic



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

```
> ./single_shopping
grabbing the salad...
grabbing the apples...
grabbing the milk...
grabbing the butter...
grabbing the cheese...
>
```

Single-threaded shopping diagram and output

2.b Threads

Practical uses of multithreading

```
void main(...)
{
    char *produce[] = { "salad", "apples", NULL };
    char *dairy[] = { "milk", "butter", "cheese", NULL };
    void *print_msg(void *);
    pthread_t th1, th2;

    pthread_create(&th1, NULL, print_msg, (void *)produce);
    pthread_create(&th2, NULL, print_msg, (void *)dairy);
    pthread_join(th1, NULL);
    pthread_join(th2, NULL);
}

void *print_msg(void *items)
{
    int i = 0;
    while (items[i] != NULL) {
        printf("grabbing the %s...", (char *)(items[i++]));
        fflush(stdout);
        sleep(1);
    }
    return NULL;
}
```

send the kids off! (pthread_create calls)

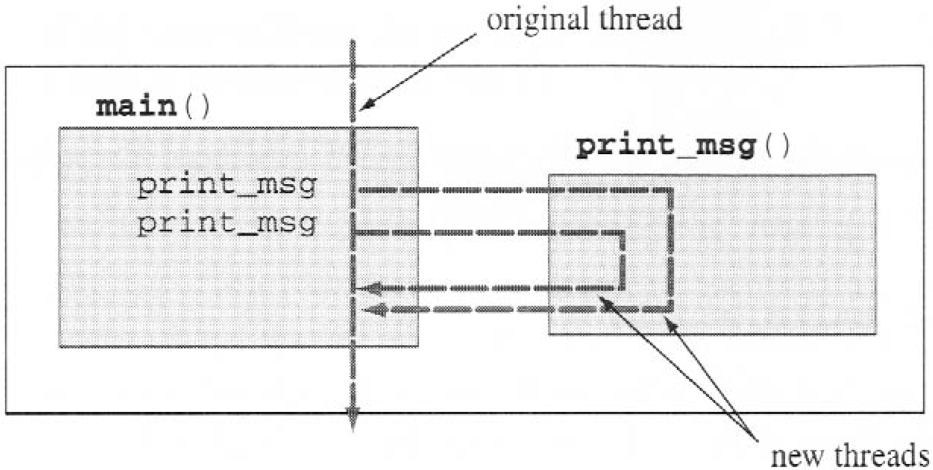
wait for their return (pthread_join calls)

Multithreaded shopping code

2.b Threads

Practical uses of multithreading

- Results of multithreaded shopping
  - ✓ total duration ≈ 3 seconds; outcome is nondeterministic



Molay, B. (2002) *Understanding Unix/Linux Programming (1st Edition).*

```
> ./multi_shopping
grabbing the salad...
grabbing the milk...
grabbing the apples...
grabbing the butter...
grabbing the cheese...
>
```

```
> ./multi_shopping
grabbing the milk...
grabbing the butter...
grabbing the salad...
grabbing the cheese...
grabbing the apples...
>
```

Multithreaded shopping diagram and possible outputs

2.b Threads

Practical uses of multithreading

- System calls for thread creation and termination wait
  - ✓
    ```
    err = pthread_create(pthread_t  *th,
                         pthread_attr_t  *attr,
                         void  *(*func)(void *),
                         void  *arg)
    ```
    creates a new thread of execution and calls **`func(arg)`** within that thread; the new thread can be given specific attributes **`attr`** or default attributes **`NULL`**
  - ✓
    ```
    err = pthread_join(pthread_t  th,
                       void  **retval)
    ```
    blocks the calling thread until the thread specified by **`th`** terminates; the return value from **`th`** can be stored in **`retval`**

2.b Threads

Practical uses of multithreading

- **Benefits of multithreading compared to multitasking**
  - ✓ it takes less time to create a new thread than a new process
  - ✓ it takes less time to terminate a thread than a process
  - ✓ it takes less time to switch between two threads within the same process than between two processes
  - ✓ threads within the same process share memory and files, therefore they can communicate with each other without having to invoke the kernel
  - ✓ for these reasons, threads are sometimes called “lightweight processes”
  - → if an application should be implemented as a set of related executions, it is far more efficient to use threads than processes

2.b Threads

Practical uses of multithreading

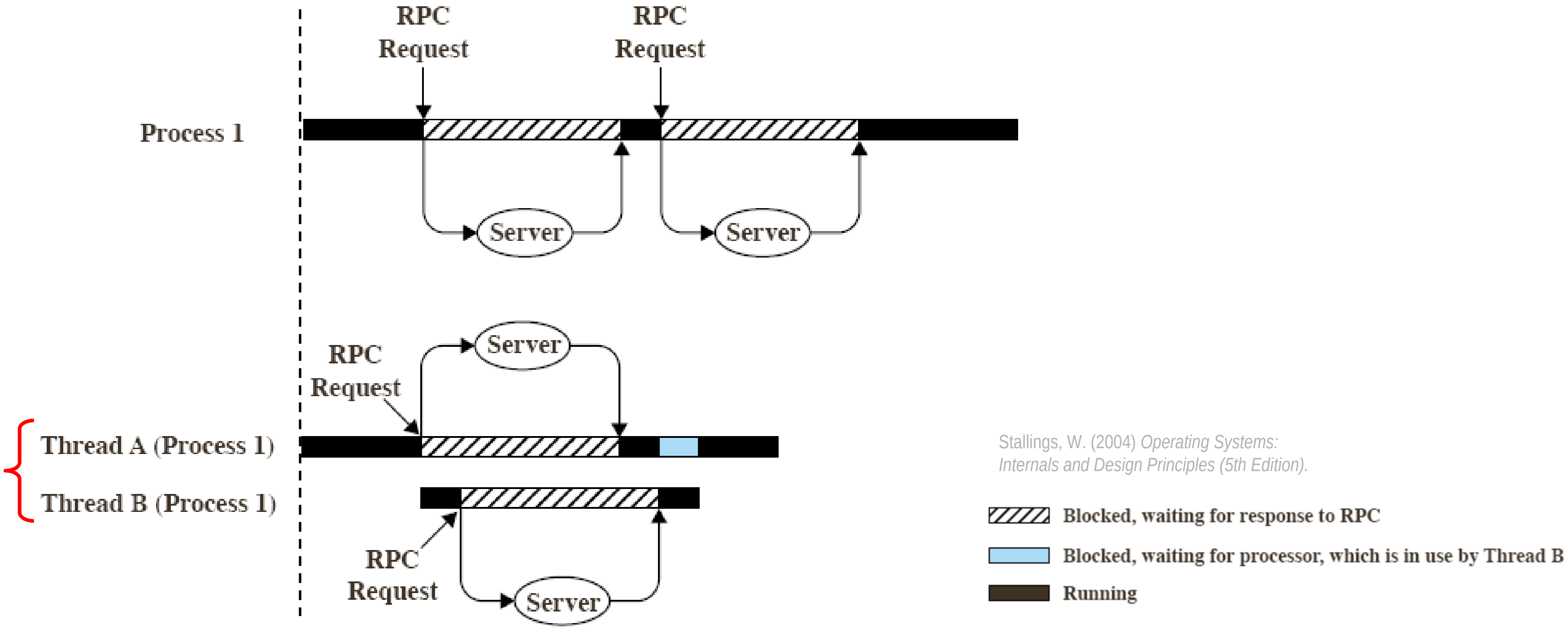
- Examples of real-world multithreaded applications
  - ✓ Web client (browser)
    - must download page components (images, styles, etc.) simultaneously; cannot wait for each image in series
  - ✓ Web server
    - must serve pages to hundreds of Web clients simultaneously; cannot process requests one by one
  - ✓ word processor, spreadsheet
    - provides uninterrupted GUI service to the user while reformatting or saving the document in the background

→ *again, same principles as time-sharing (illusion of interactivity while performing other tasks), this time inside the same process*

2.b Threads

Practical uses of multithreading

- Web client and Remote Procedure Calls (RPCs)
  - ✓ the client uses multiple threads to send multiple requests to the same server or different servers, greatly increasing performance

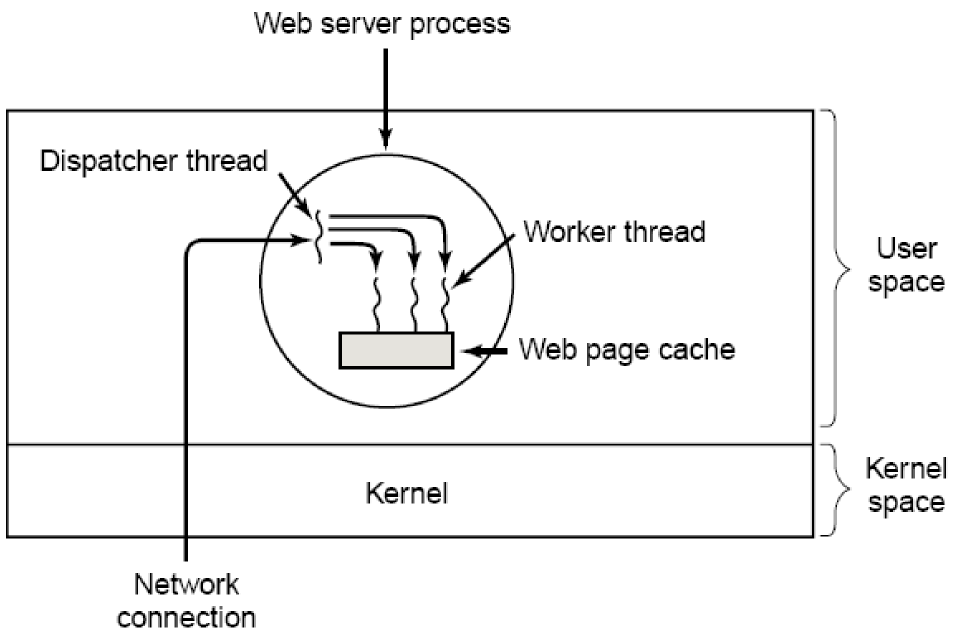


Client RPC using a single thread vs. multiple threads

2.b Threads

Practical uses of multithreading

- **Web server**
  - ✓ as each new request comes in, a “dispatcher thread” spawns a new “worker thread” to read the requested file (worker threads may be discarded or recycled in a “thread pool”)



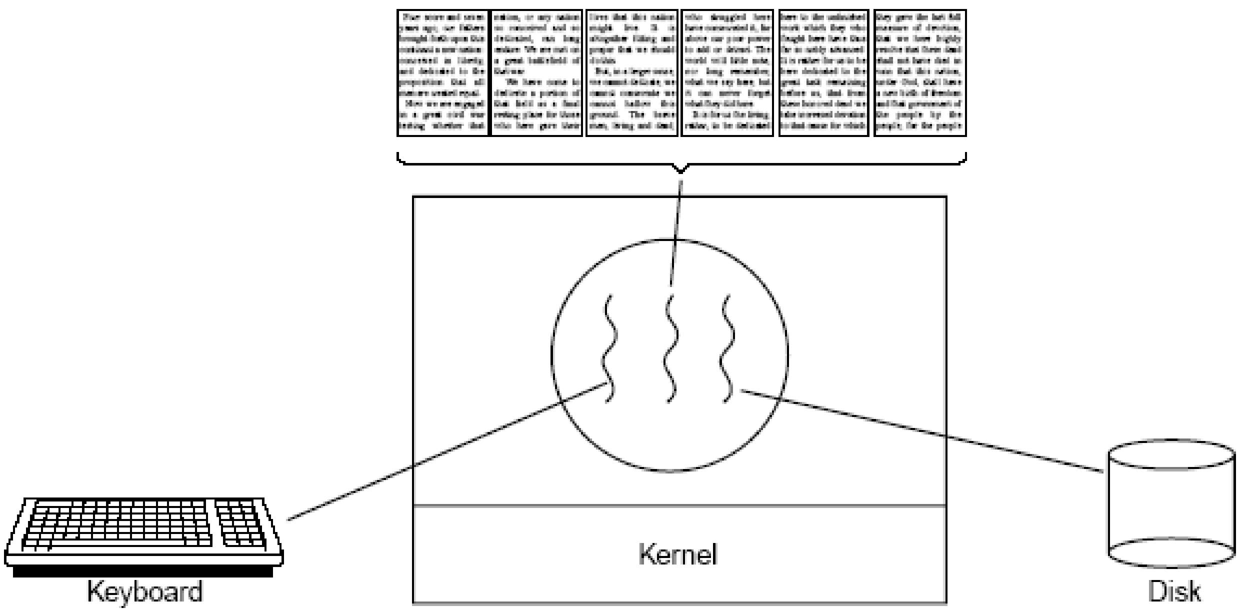
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

A multithreaded Web server

2.b Threads

Practical uses of multithreading

- Word processor
  - ✓ one thread listens continuously to keyboard and mouse events to refresh the GUI; a second thread reformats the document (to prepare page 600); a third thread writes to disk periodically



Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

A word processor with three threads

2.b Threads

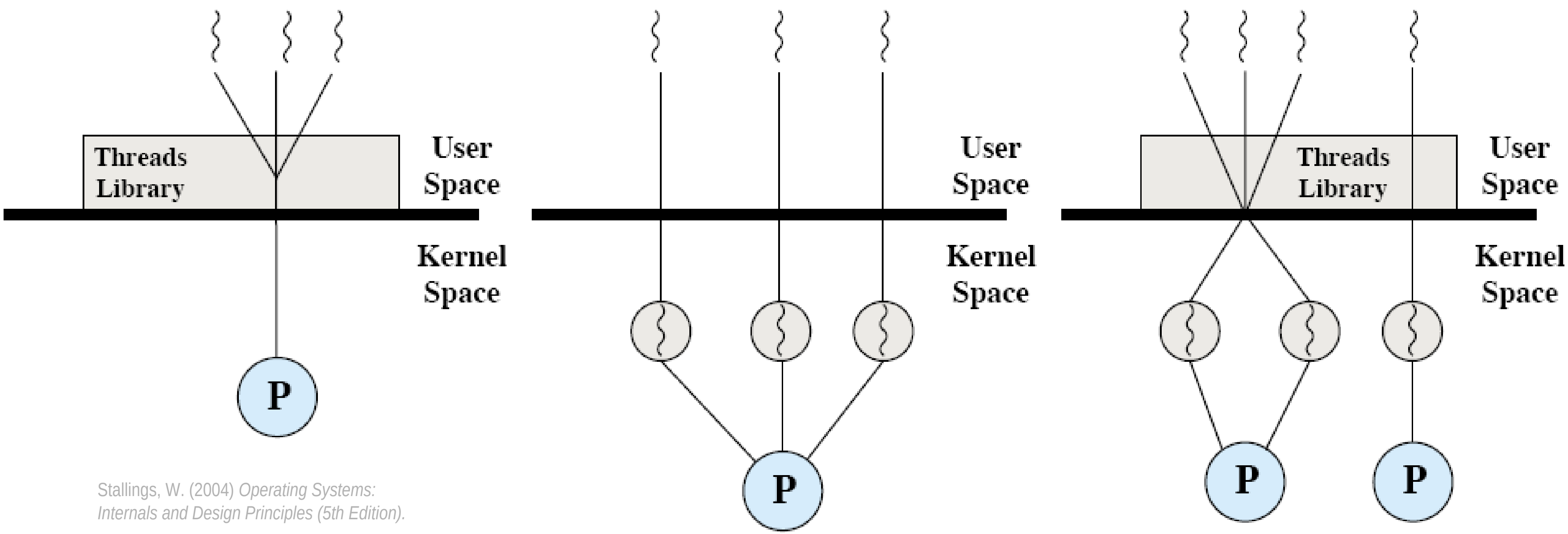
Practical uses of multithreading

- Patterns of multithreading usage across applications
  - ✓ perform foreground and background work in parallel
    - illusion of full-time interactivity toward the user while performing other tasks (same principle as time-sharing)
  - ✓ allow asynchronous processing
    - separate and desynchronize the execution streams of independent tasks that don't need to communicate
    - handle external, surprise events such as client requests
  - ✓ increase speed of execution
    - "stagger" and overlap CPU execution time and I/O wait time (same principle as multiprogramming)

2.b Threads

Implementation of threads

- Two broad categories of thread implementation
  - ✓ User-Level Threads (ULTs)
  - ✓ Kernel-Level Threads (KLTs)



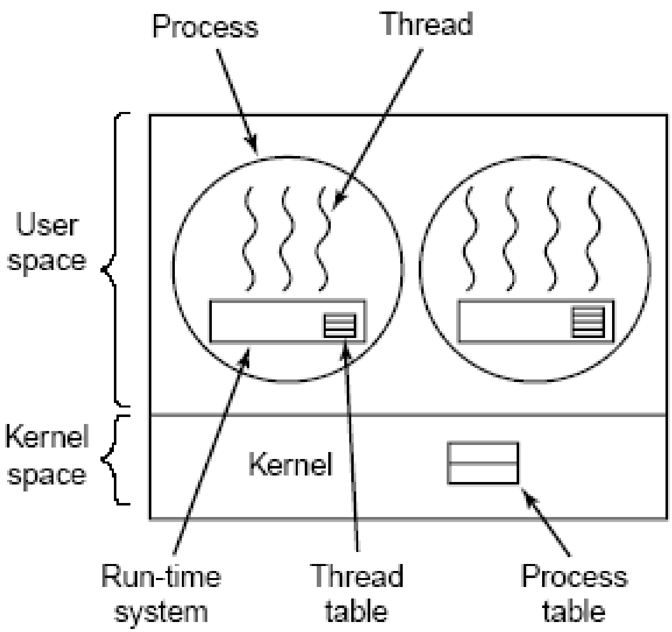
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

Pure user-level (ULT), pure kernel-level (KLT) and combined-level (ULT/KLT) threads

2.b Threads

Implementation of threads

- **User-Level Threads (ULTs)**
  - ✓ the kernel is not aware of the existence of threads, it knows only processes with one thread of execution (one PC)
  - ✓ each user process manages its own private thread table
  - light thread switching: does not need kernel mode privileges
  - cross-platform: ULTs can run on any underlying O/S
  - if a thread blocks, the entire process is blocked, including all other threads in it



Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

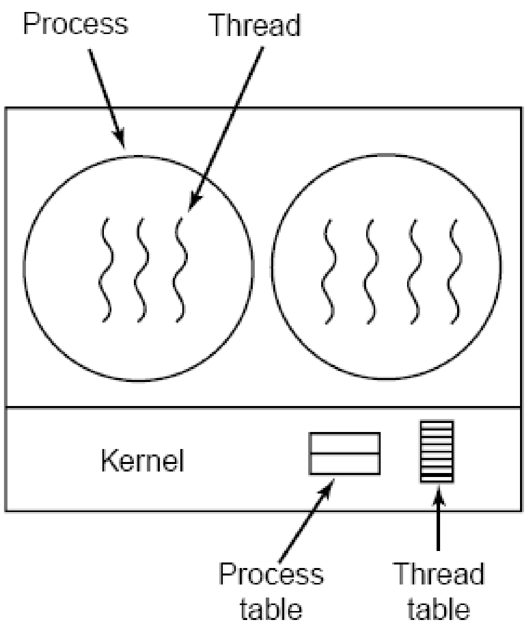
A user-level thread package

2.b Threads

Implementation of threads

- Kernel-Level Threads
  - ✓ the kernel knows about and manages the threads: creating and destroying threads are system calls

  - 👍 fine-grain scheduling, done on a thread basis
  - 👍 if a thread blocks, another one can be scheduled without blocking the whole process
  - 👎 heavy thread switching involving mode switch



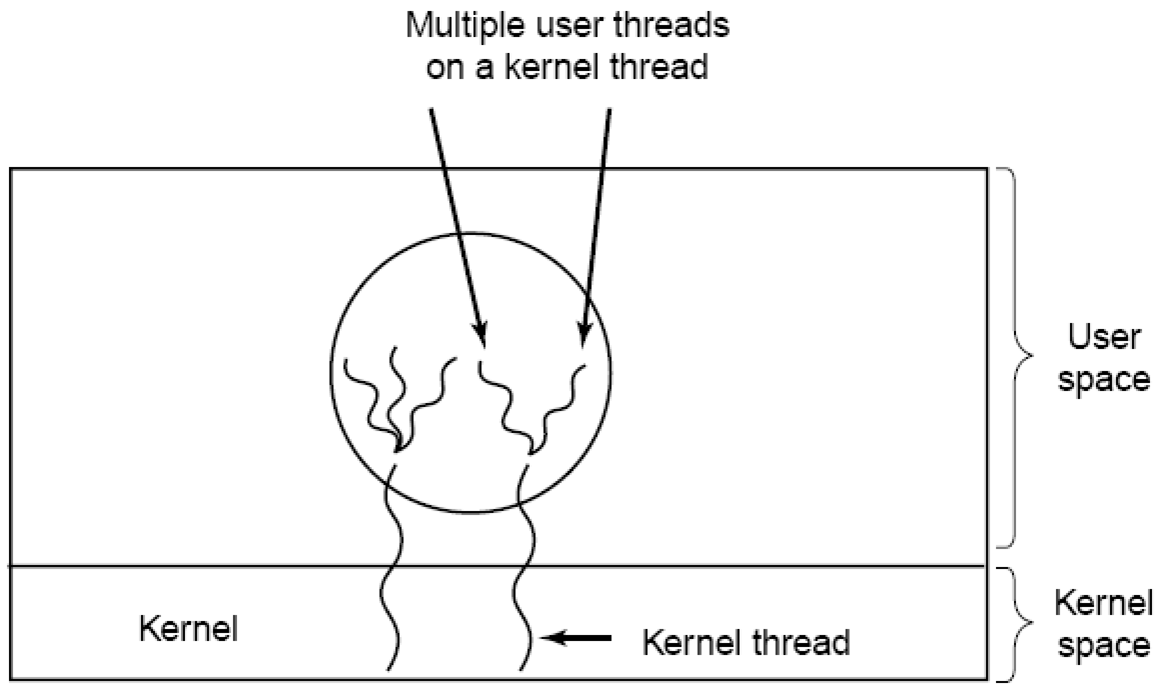
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

A kernel-level thread package

2.b Threads

Implementation of threads

- **Hybrid implementation**
  - ✓ combine both approaches: graft ULTs onto KLTs



Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

Multiplexing ULTs onto KLTs

Principles of Operating Systems

CS 446/646

2. Processes

a. Process Description & Control

b. **Threads**

- ✓ Separation of resource ownership and execution
- ✓ It's the same old throughput story, again
- ✓ Practical uses of multithreading
- ✓ Implementation of threads

c. **Concurrency**

d. **Deadlocks**