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)

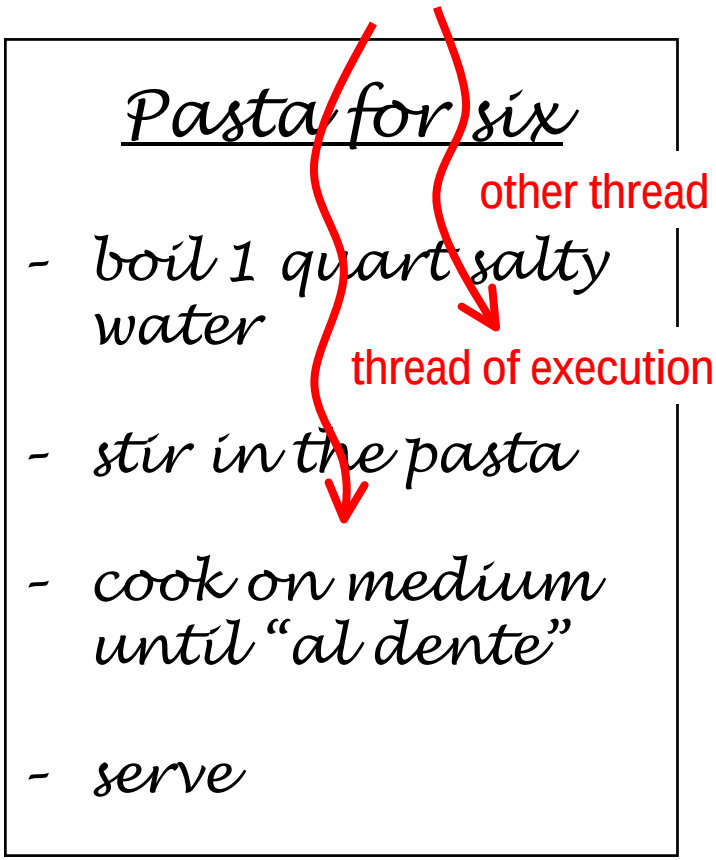
2. Execution & scheduling

- ✓ a process follows an execution path through a program
- ✓ it has an execution state and is scheduled for dispatching

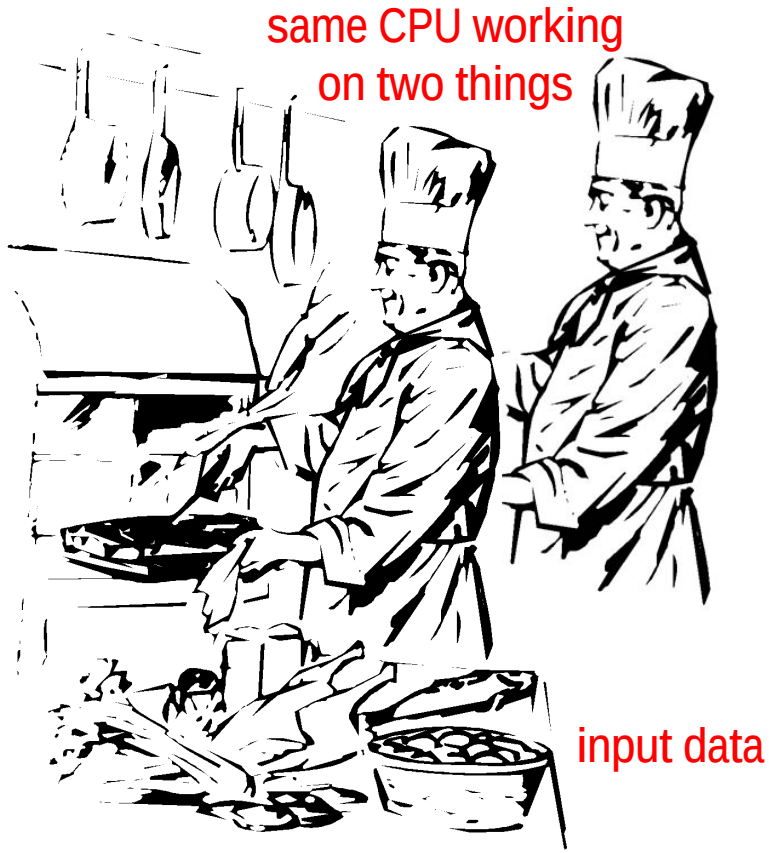
2.b Threads

Separation of resource ownership and execution

- The execution part is a "thread" that can be multiplied



Program

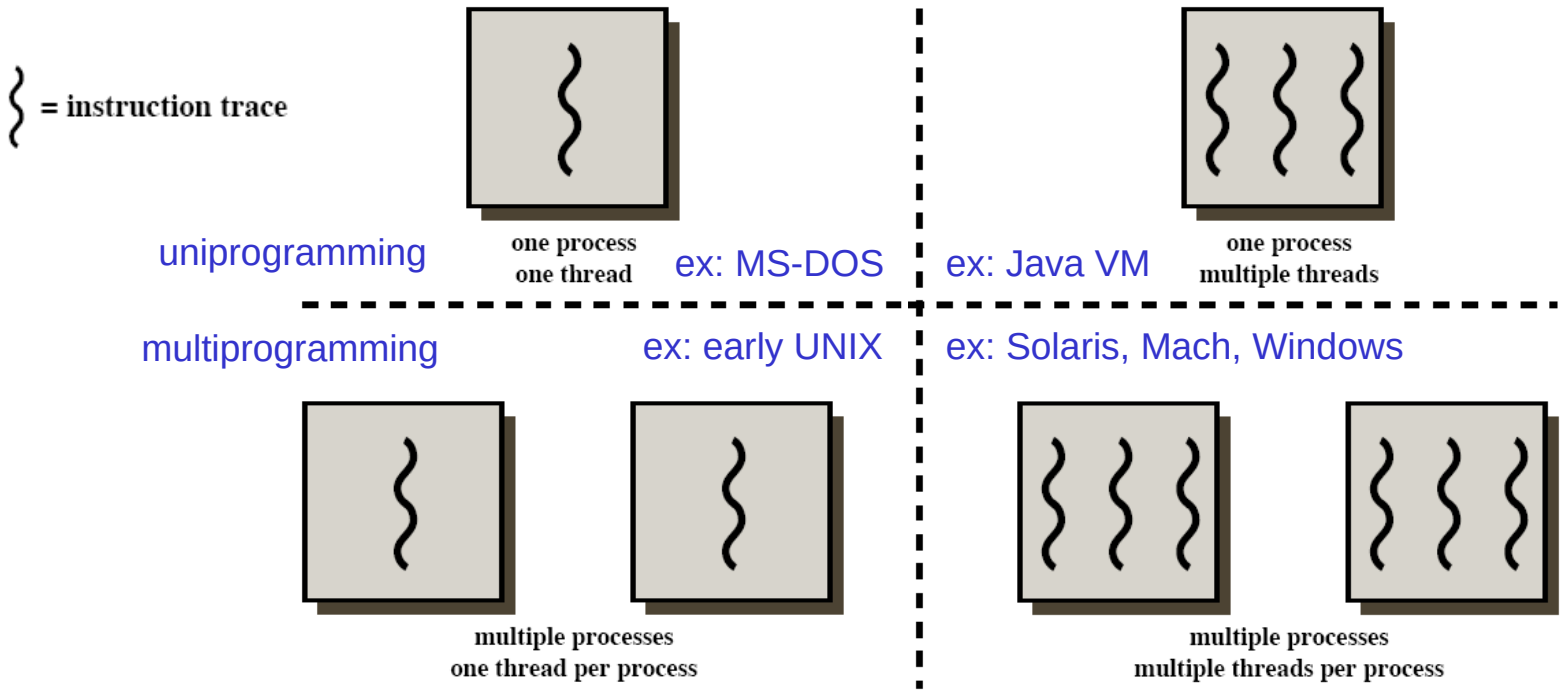


Process

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



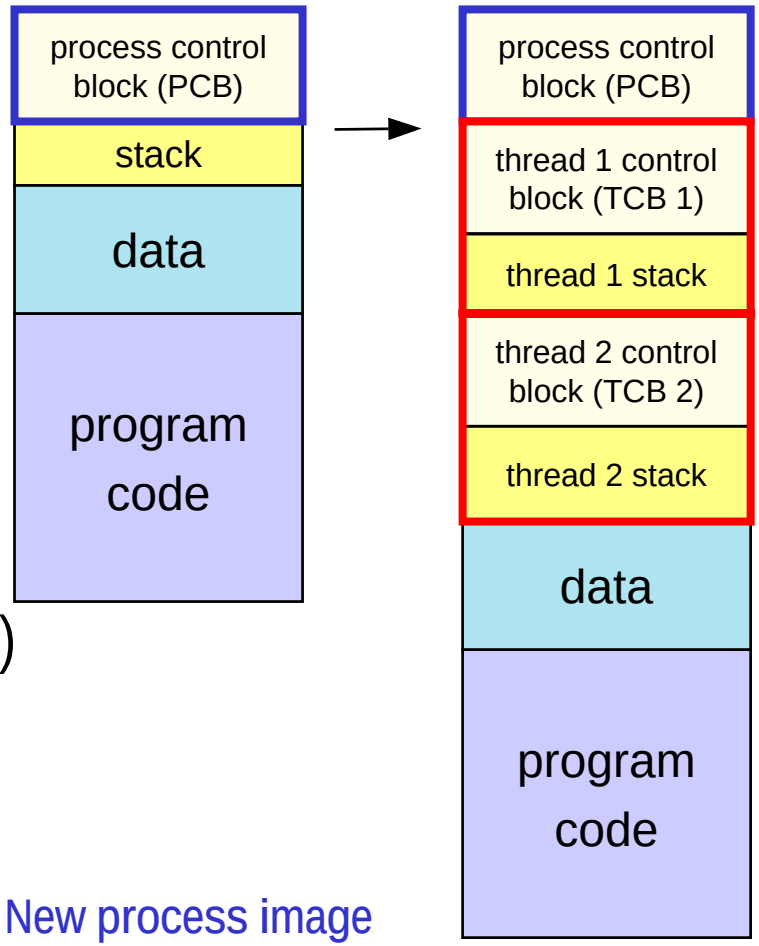
Process-thread relationships

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



2.b Threads

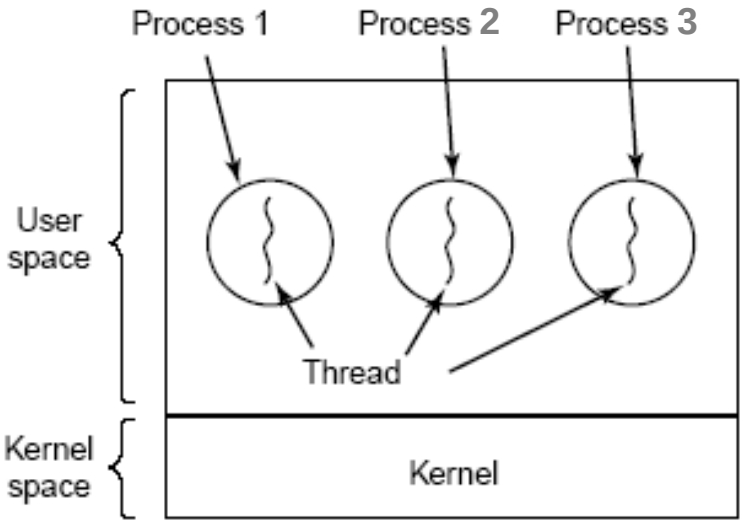
Separation of resource ownership and execution

- 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

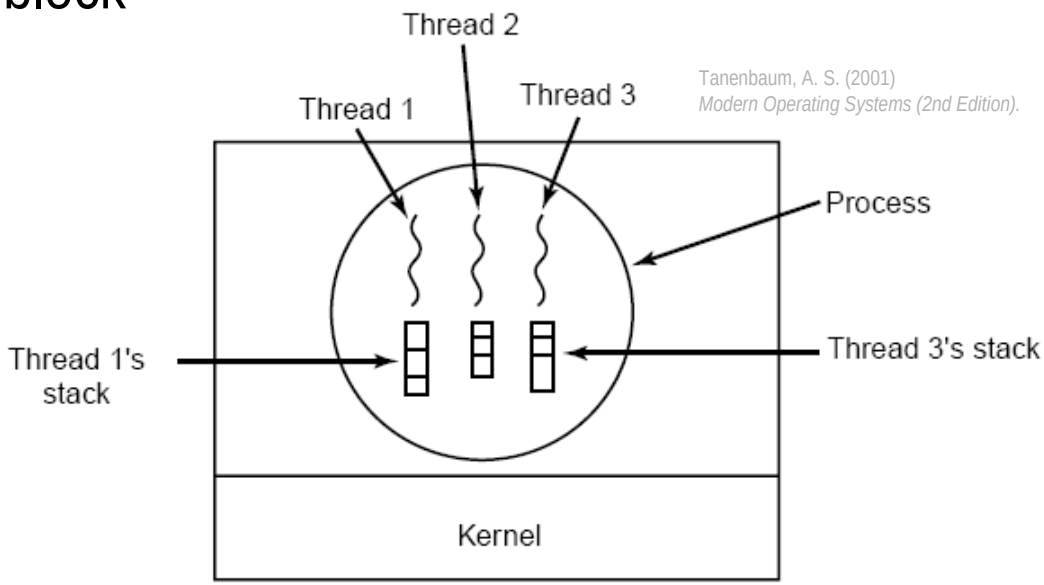
2.b Threads

Separation of resource ownership and execution

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



(a) Three processes with one thread



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

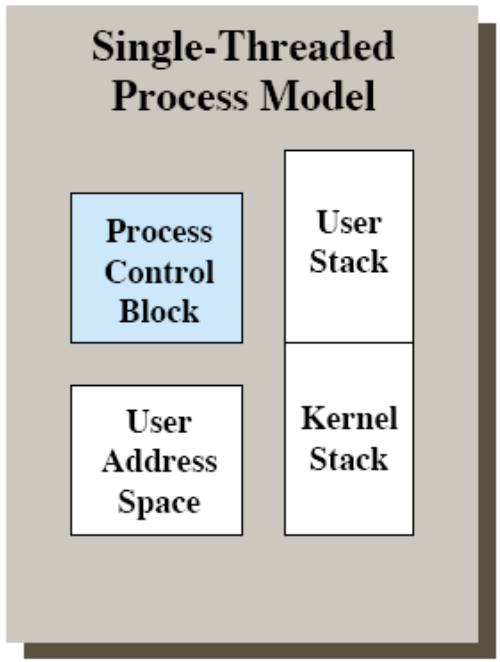
(a) One process with three threads

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

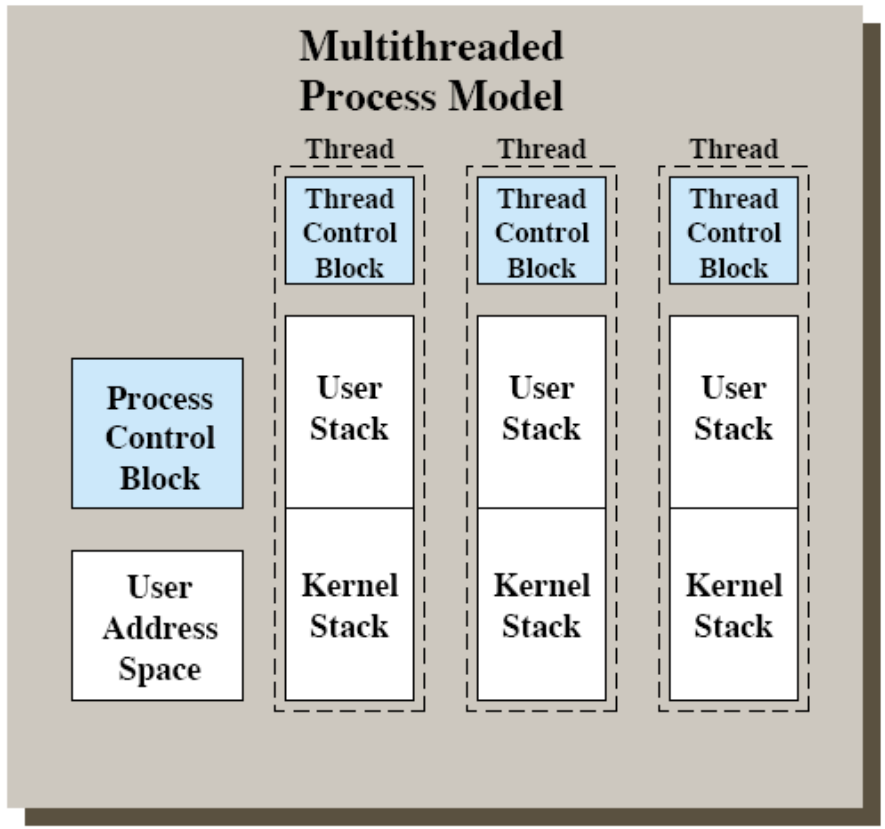
2.b Threads

Separation of resource ownership and execution

- Multithreaded process model (another view)



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

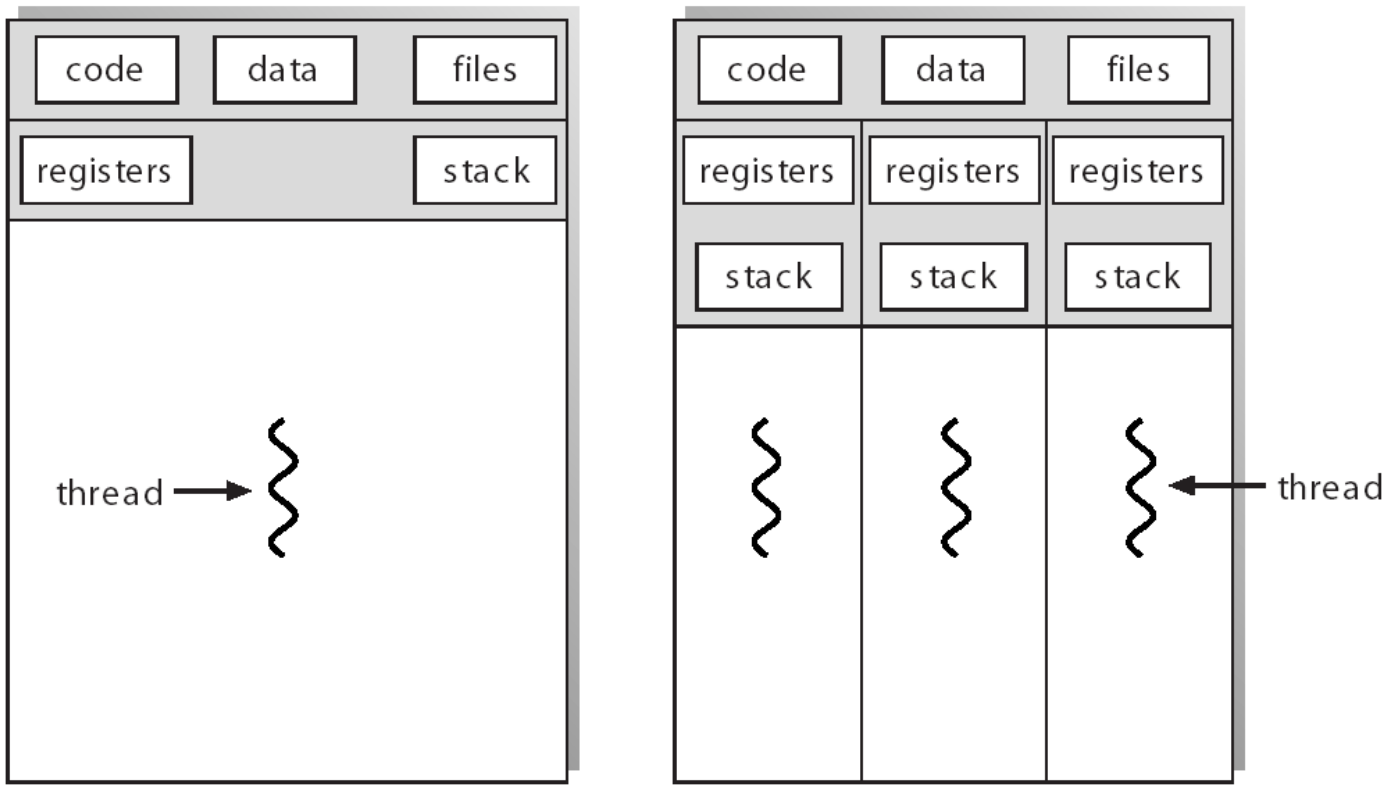


Single-threaded and multithreaded process models (in abstract space)

2.b Threads

Separation of resource ownership and execution

- Multithreaded process model (yet another view)



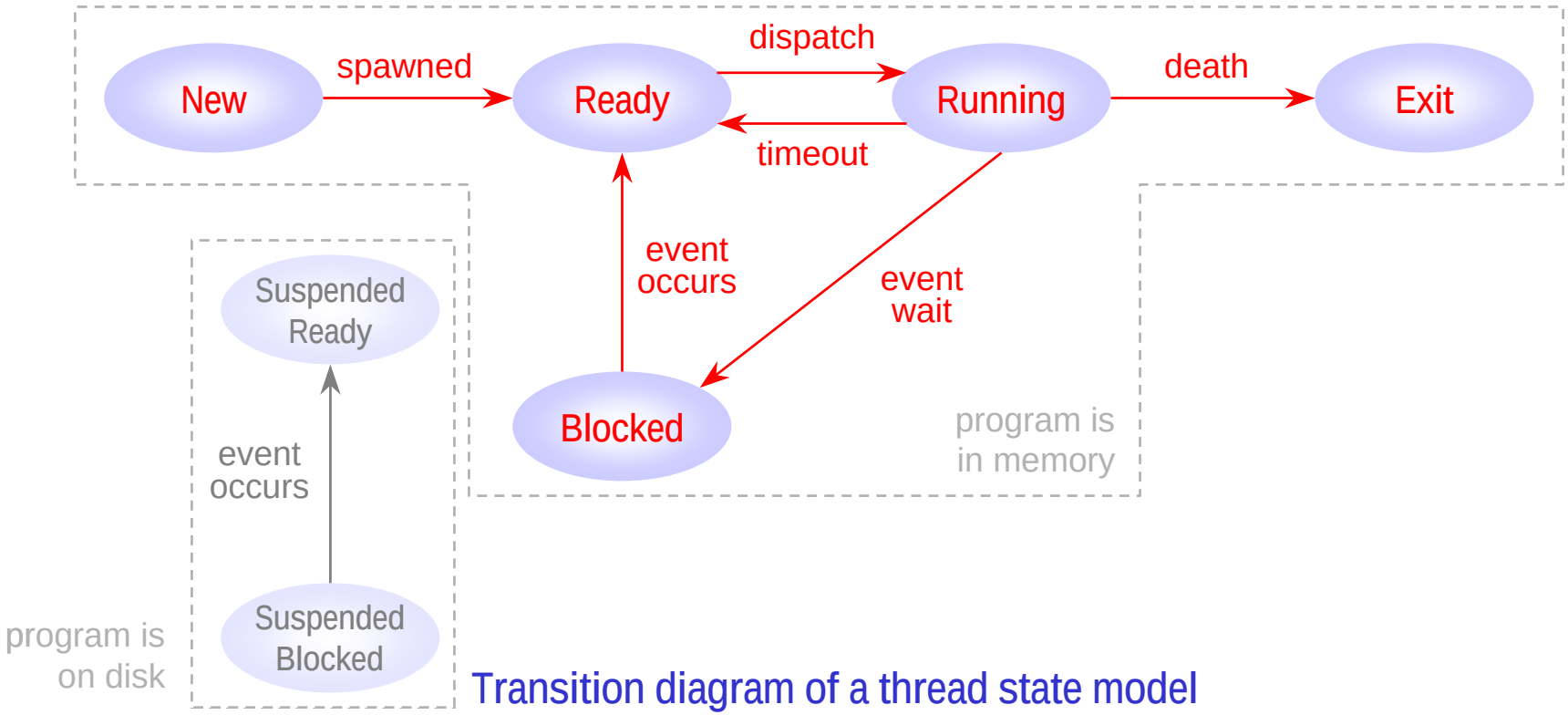
Silberschatz, A., Galvin, P. B. and Gagne. G. (2003) *Operating Systems Concepts with Java (6th Edition).*

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

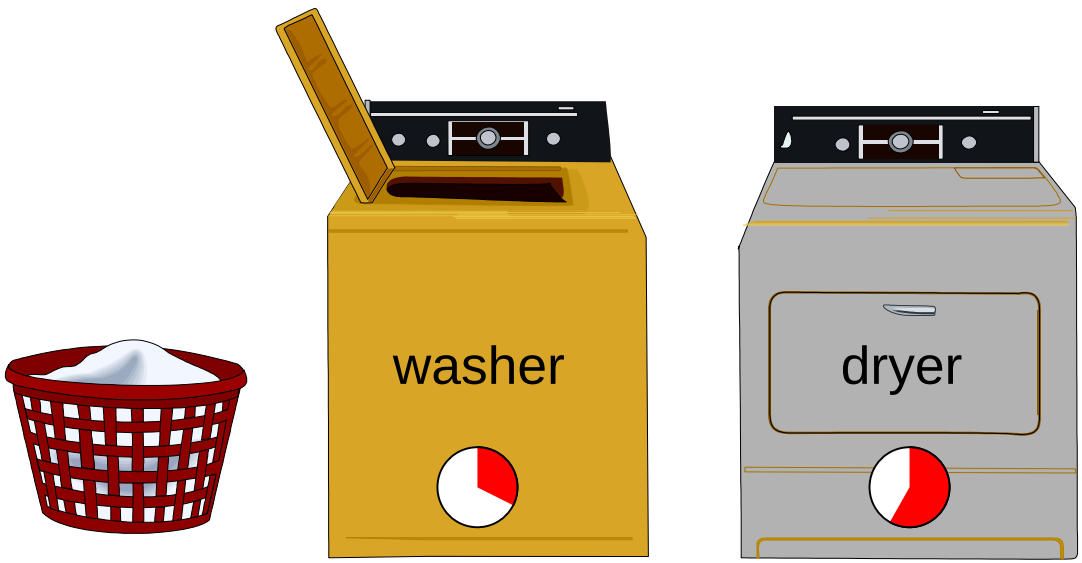


Transition diagram of a thread state model

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

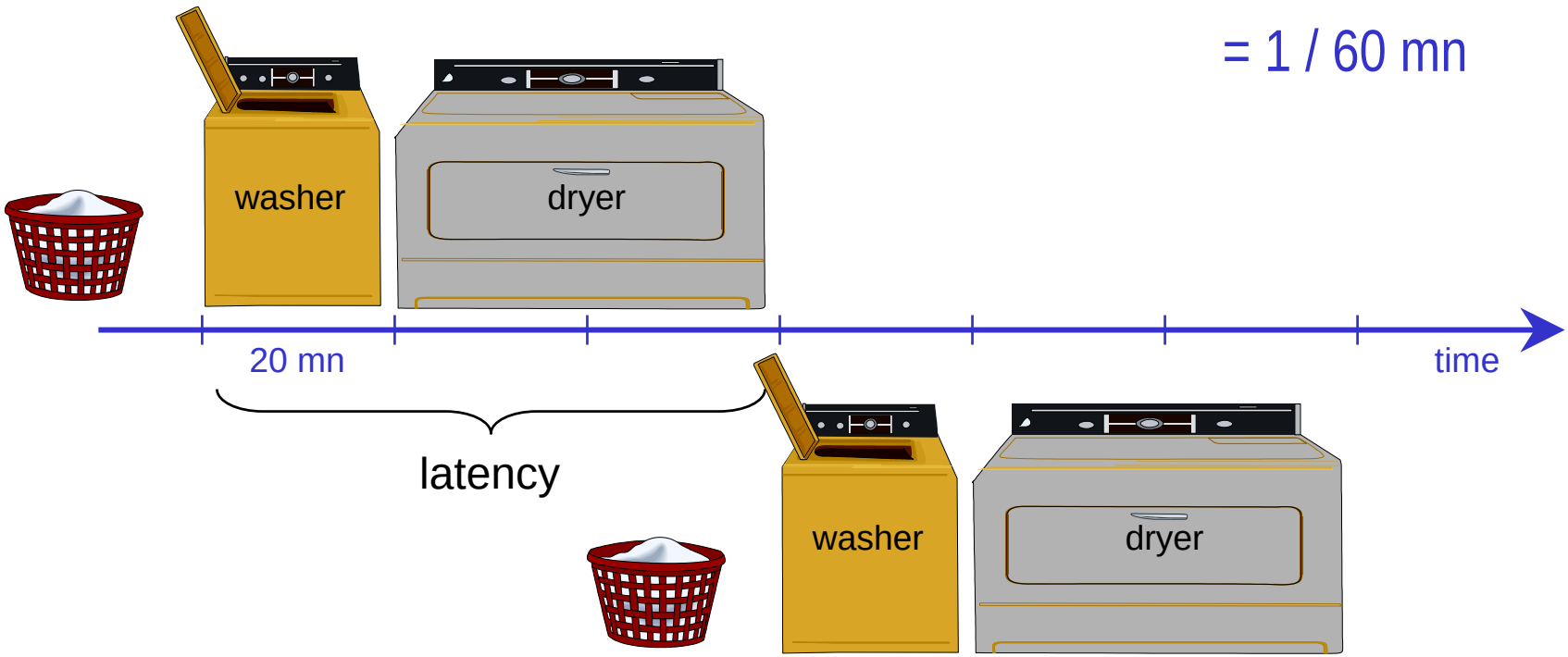


after Gill Pratt (2000) *How Computers Work.* ADUni.org/courses.

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

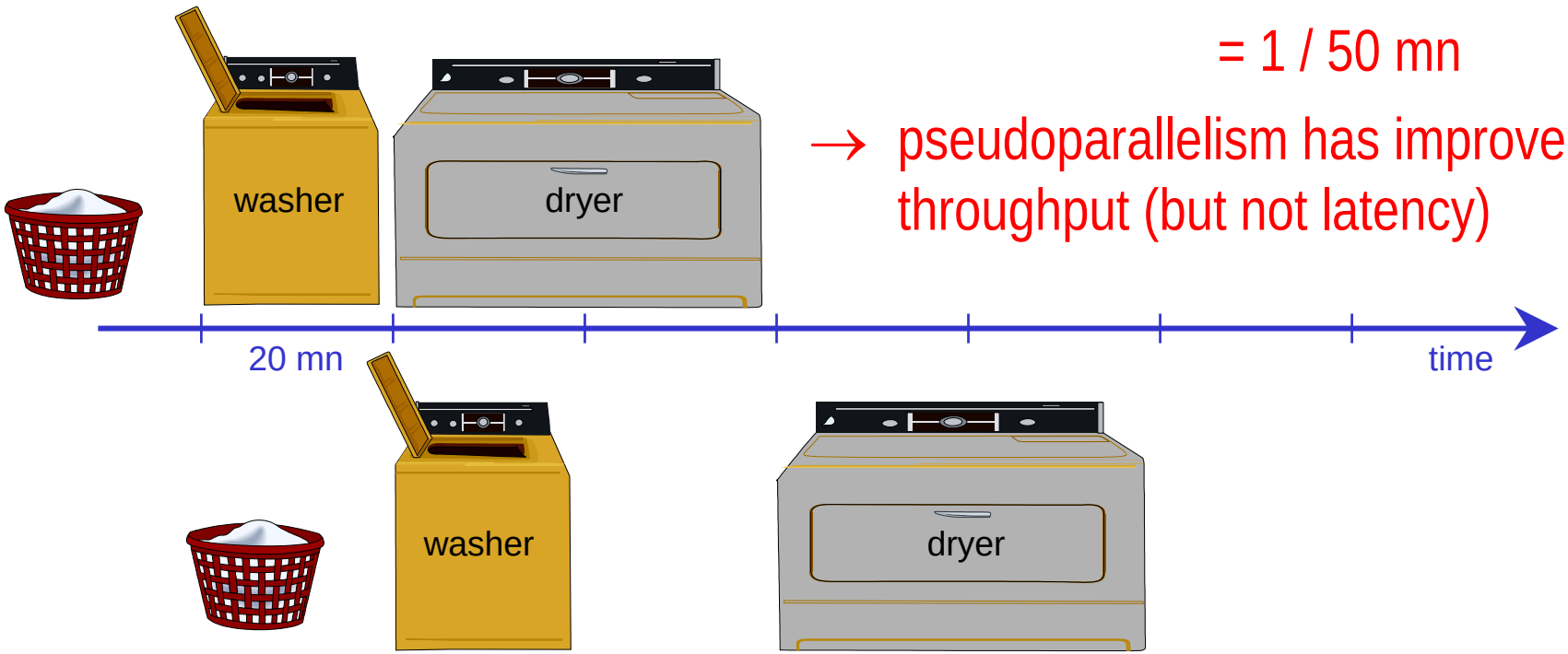


Two loads in a sequence

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

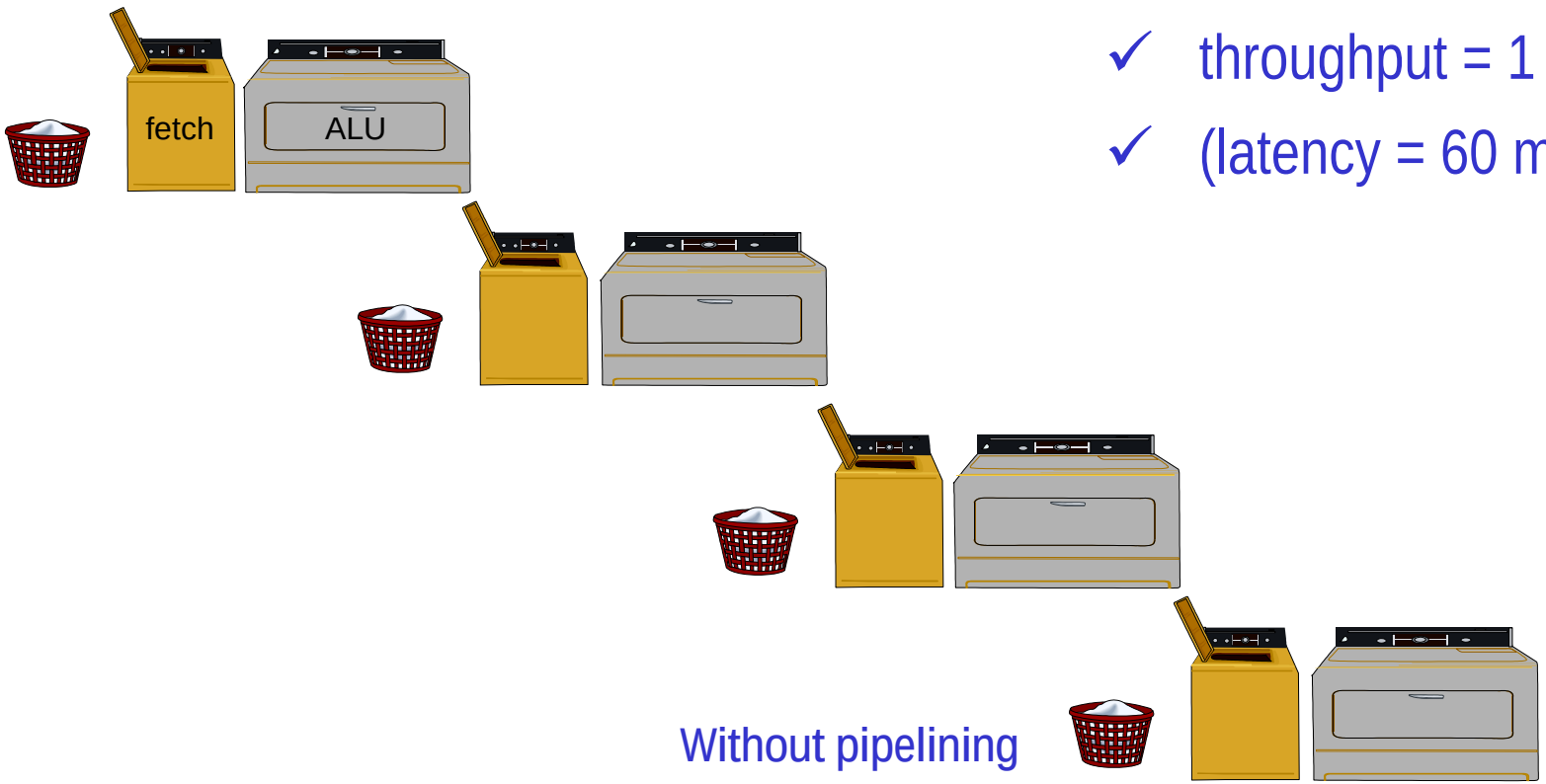
→ pseudoparallelism has improved throughput (but not latency)

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)

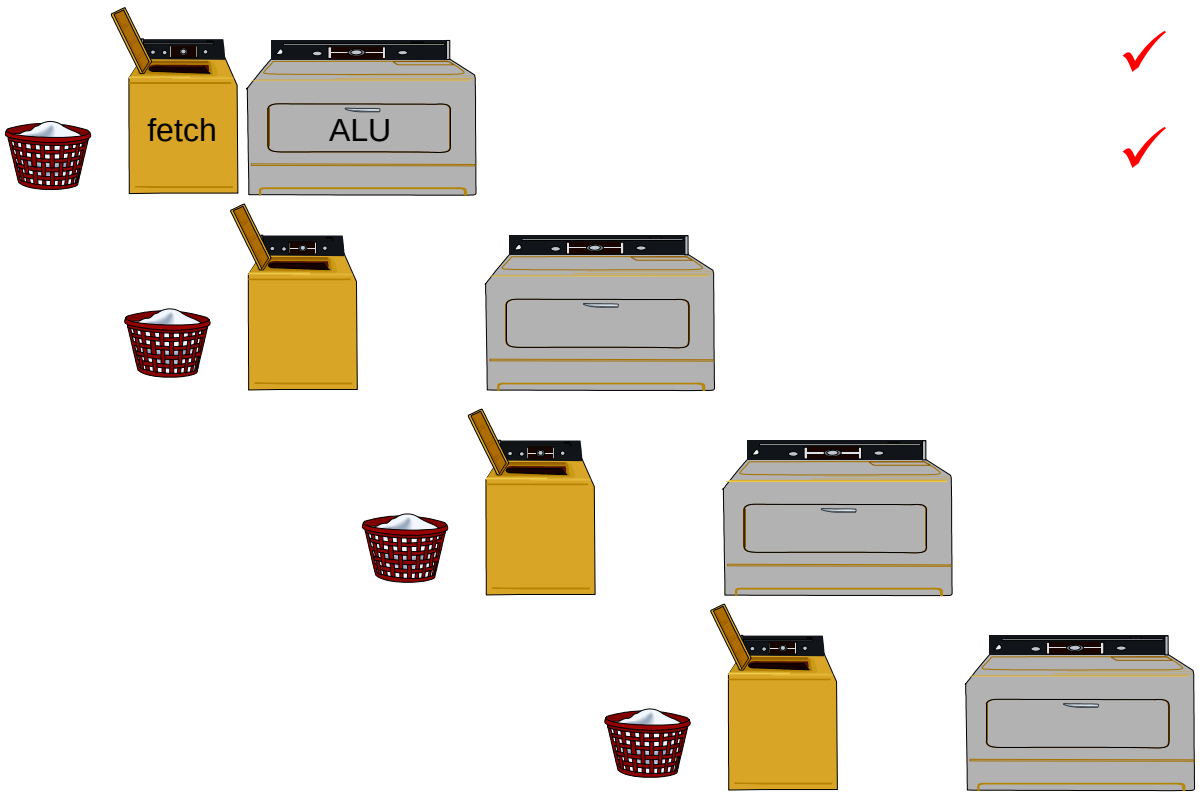


Without pipelining

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)

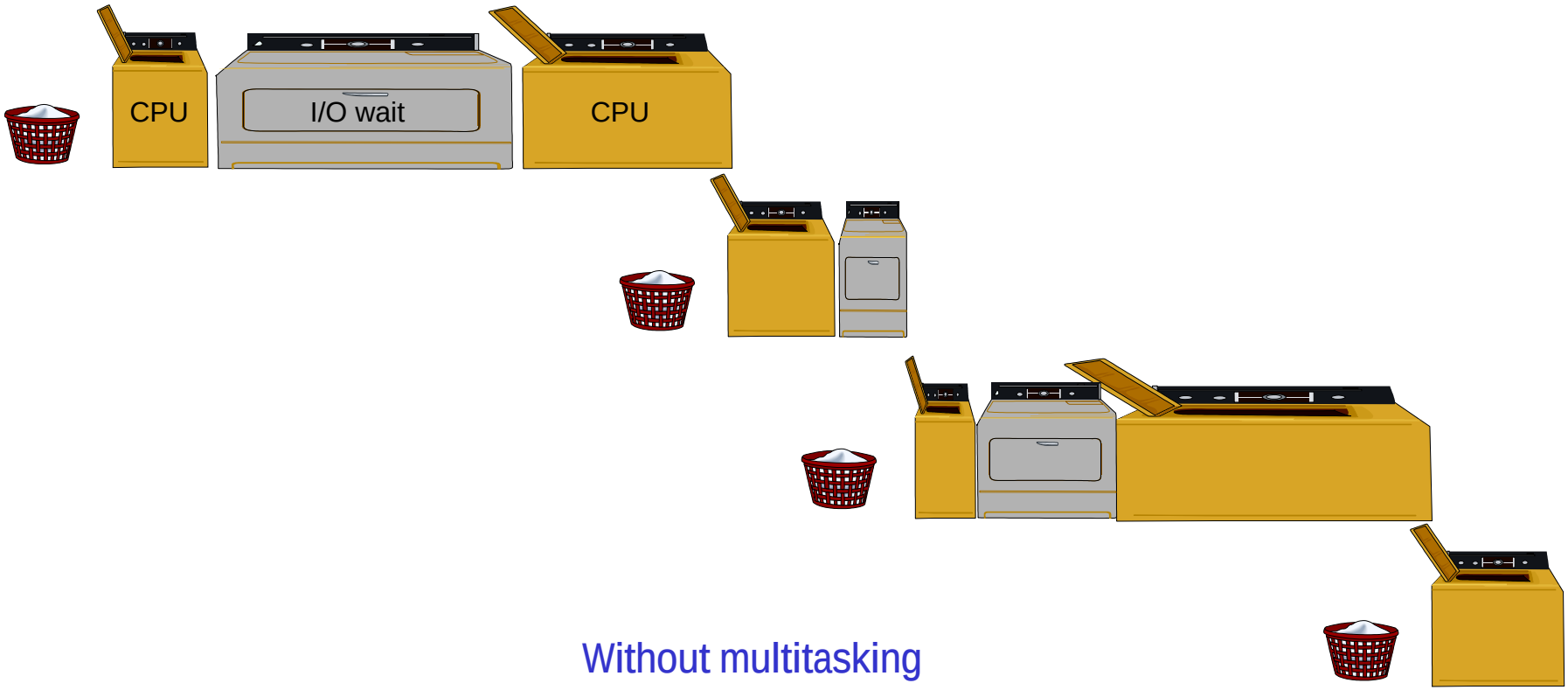


With pipelining

2.b Threads

It's the same old throughput story, again

- This is also the principle used in multitasking
  - ✓ 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

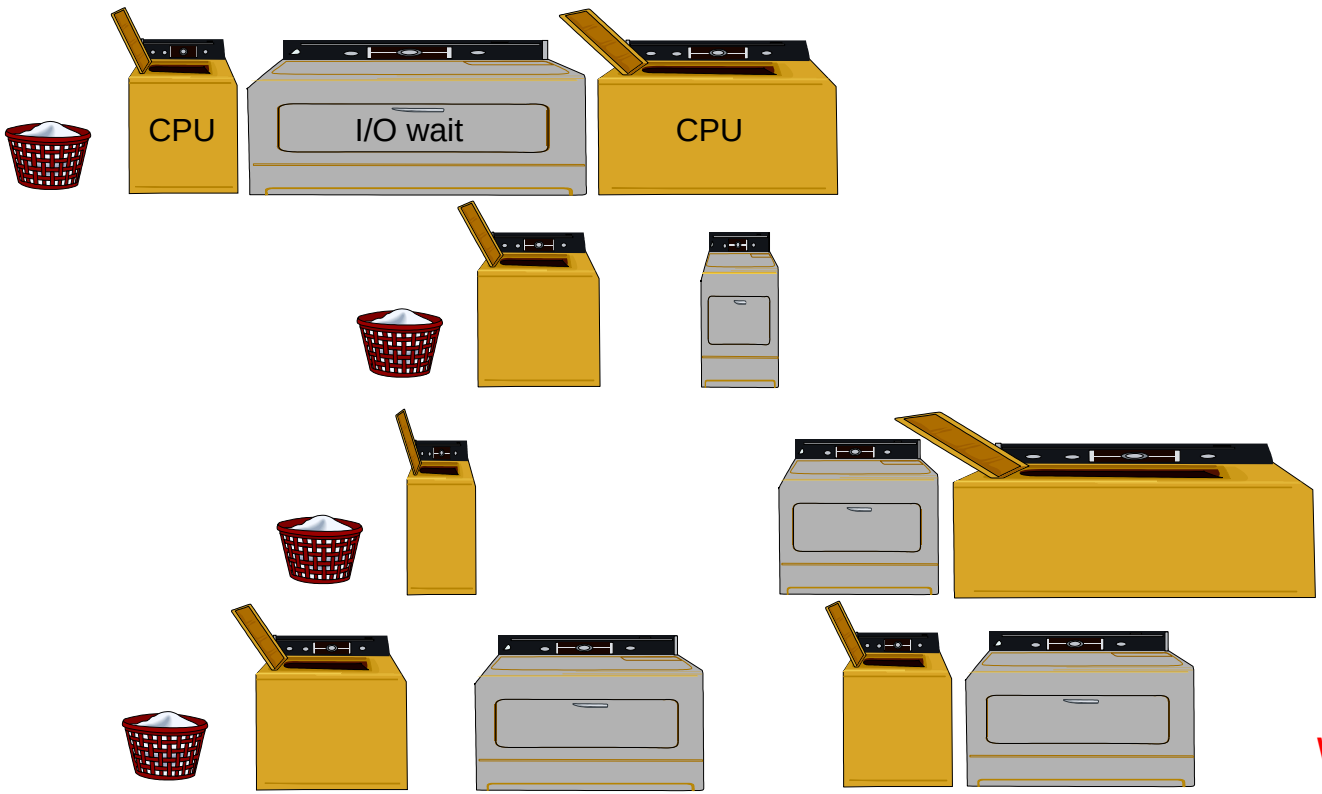


Without multitasking

2.b Threads

It's the same old throughput story, again

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

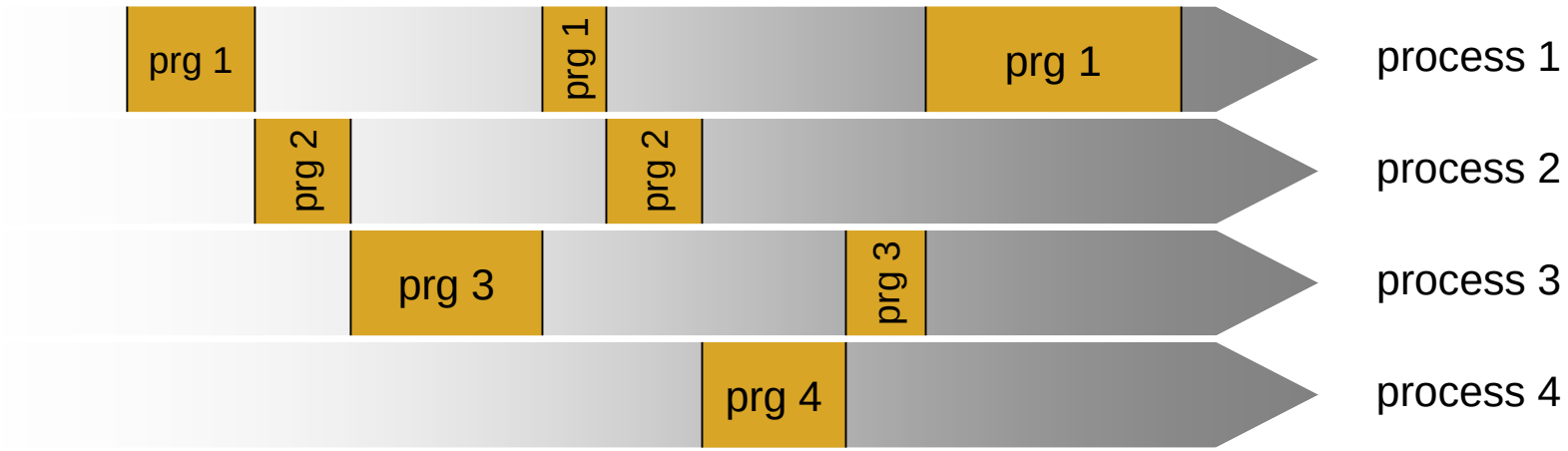


With multitasking

2.b Threads

It's the same old throughput story, again

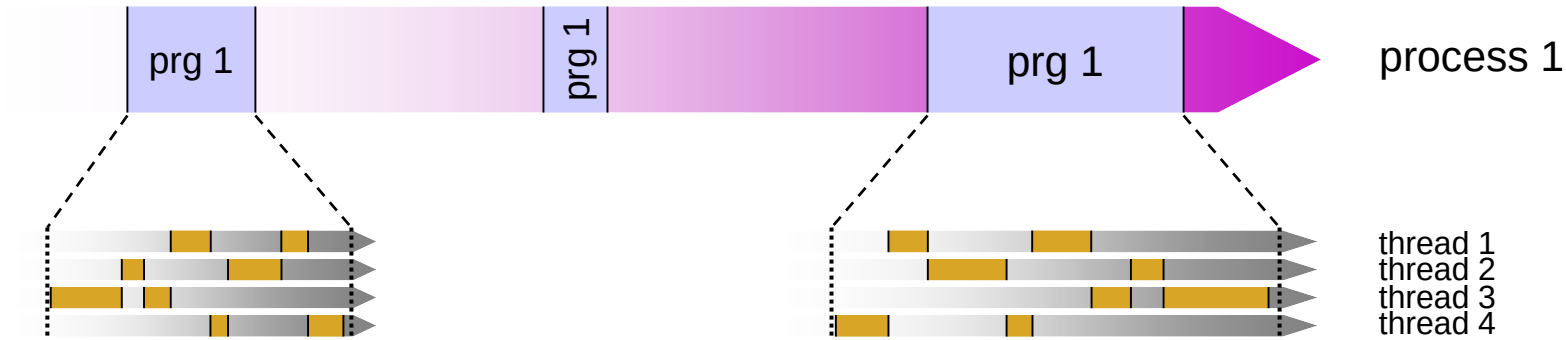
- This is also the principle used in multitasking



2.b Threads

It's the same old throughput story, again

- And, naturally, the same idea applies in multithreading
  - ✓ 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

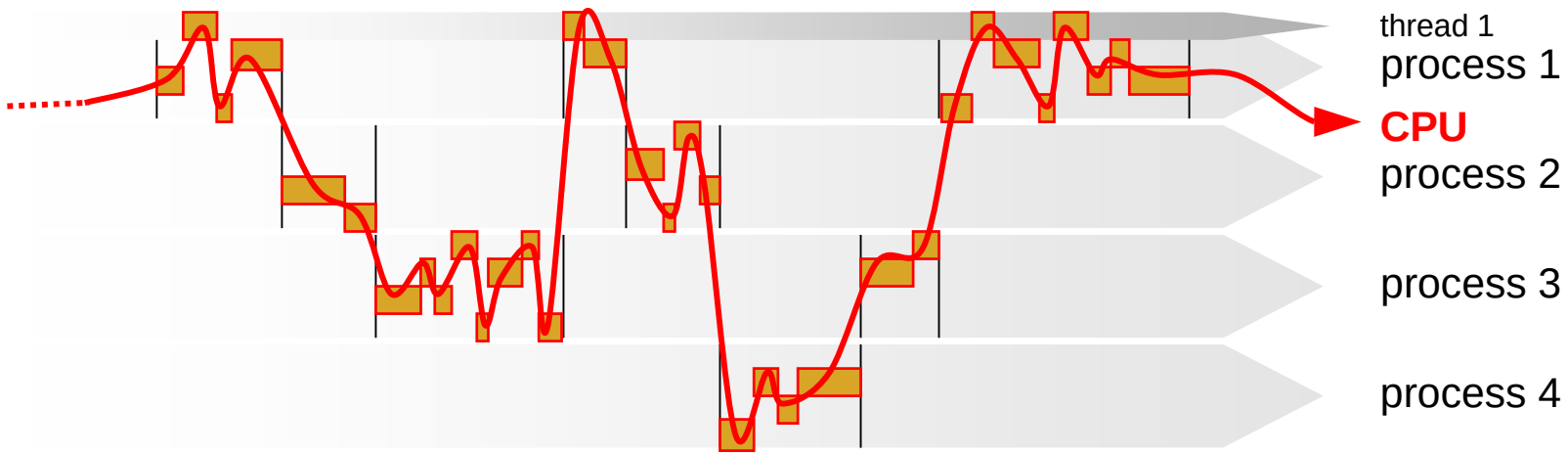


Multithreading

2.b Threads

It's the same old throughput story, again

- And, naturally, the same idea applies in multithreading
  - ✓ 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

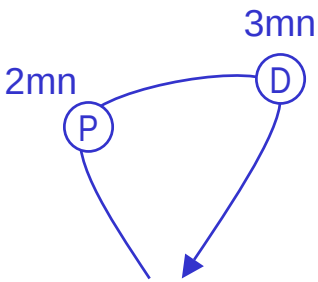
It's the same old throughput story, again

- 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.

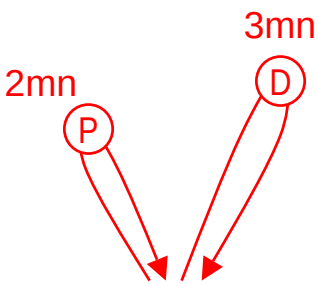
2.b Threads

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

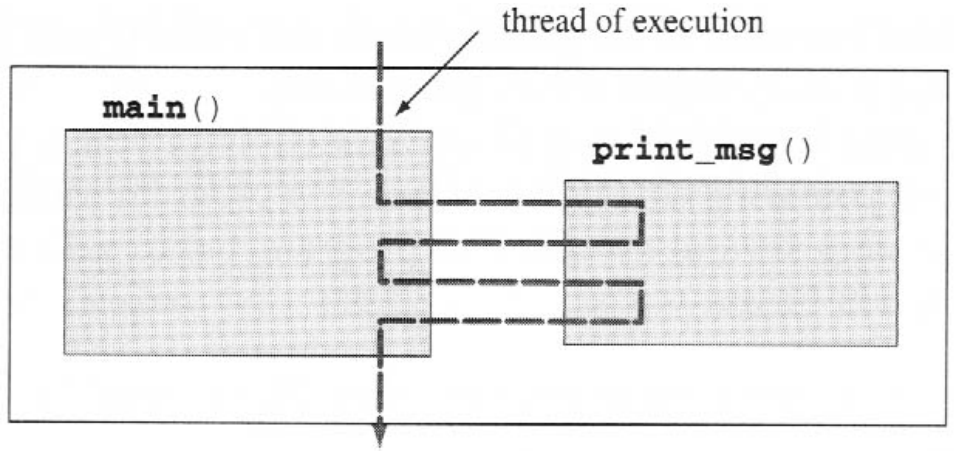
```
void main(...)
{
   char *produce[] = { "salad", "apples", NULL };
   char *dairy[] = { "milk", "butter", "cheese", NULL };

   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!

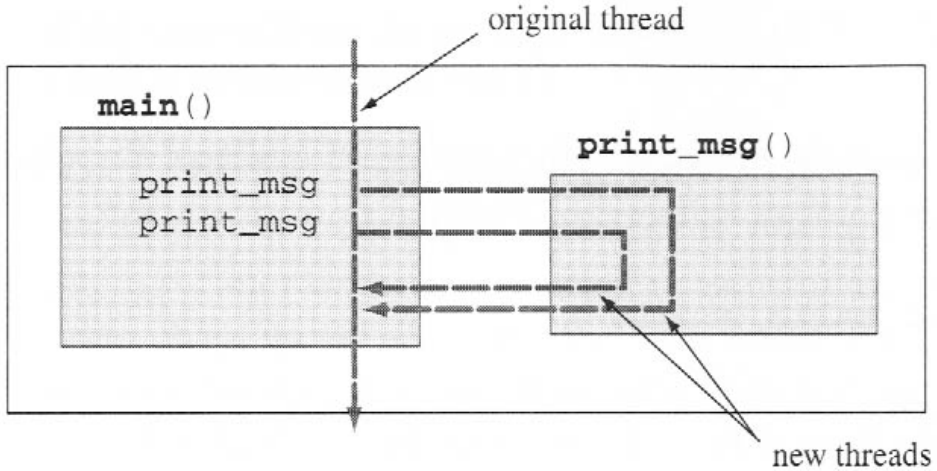
wait for their return

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

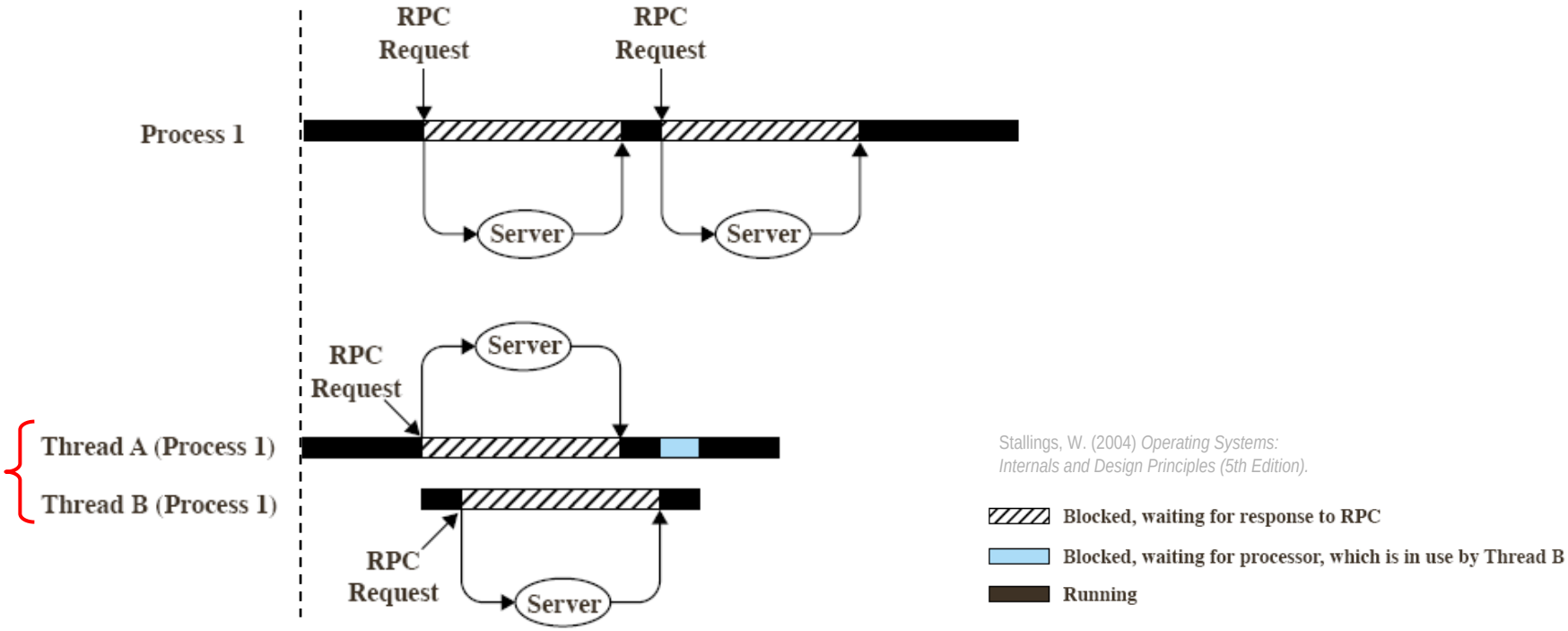
Practical uses of multithreading

- 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

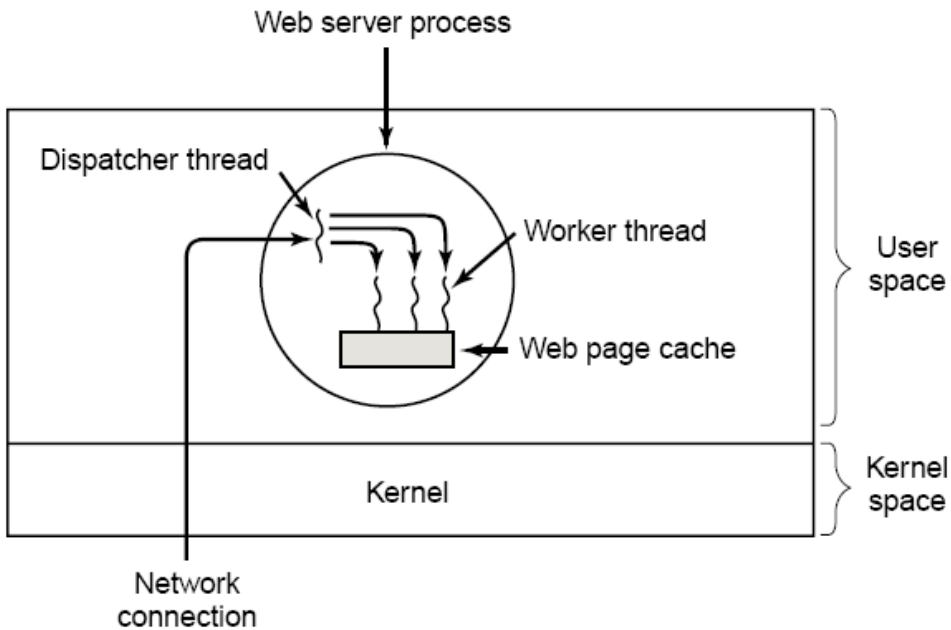


Client RPC using a single thread vs. multiple threads

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”)



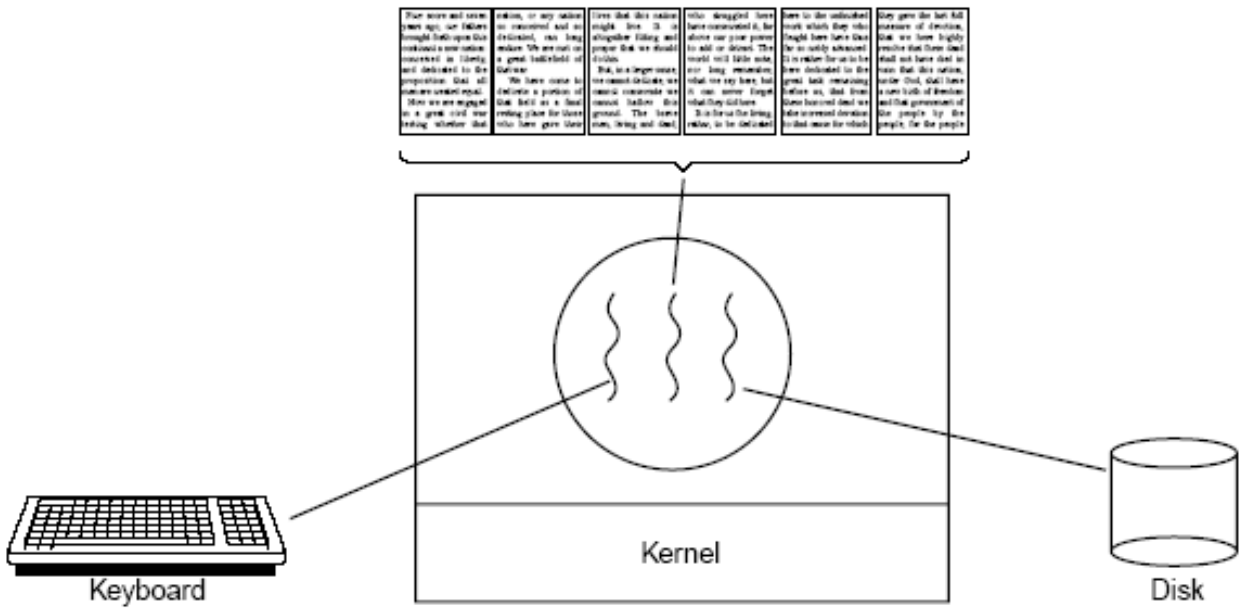
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

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

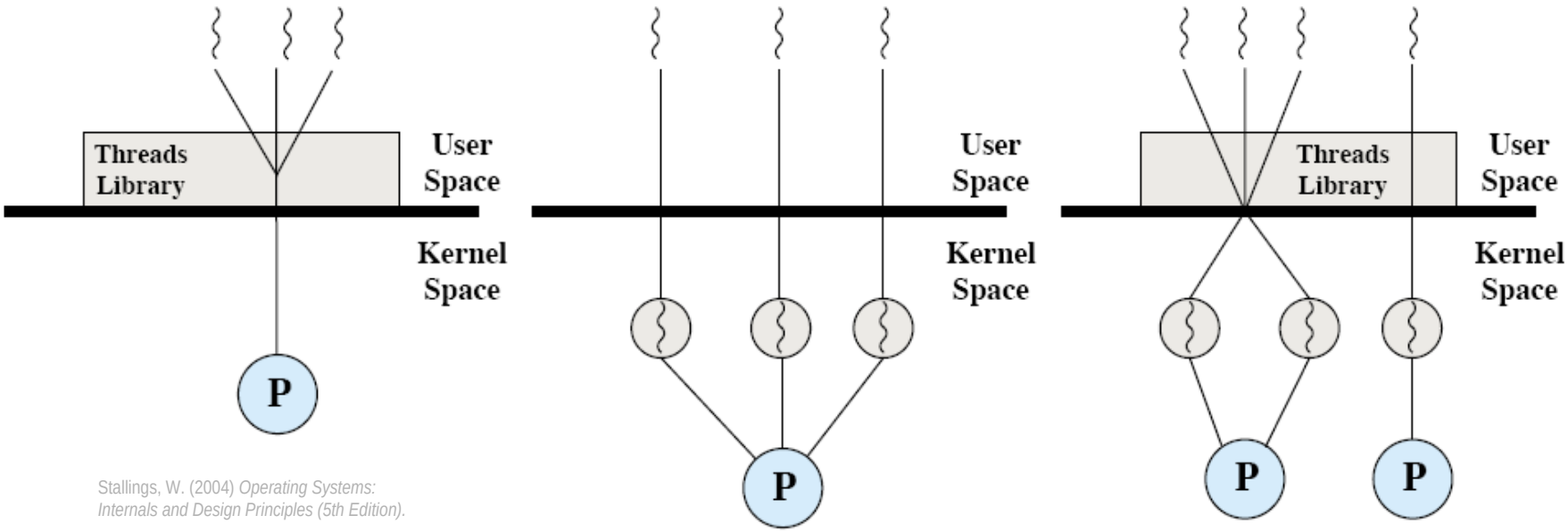
Practical uses of multithreading

- 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).*

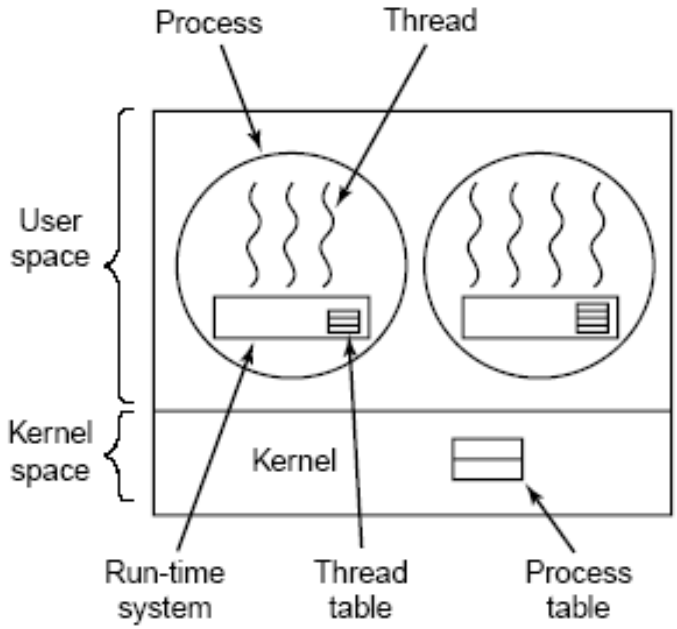
Pure user-level (ULT), pure kernel-level (KLT) and combined-level (ULT/KLT) threads

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

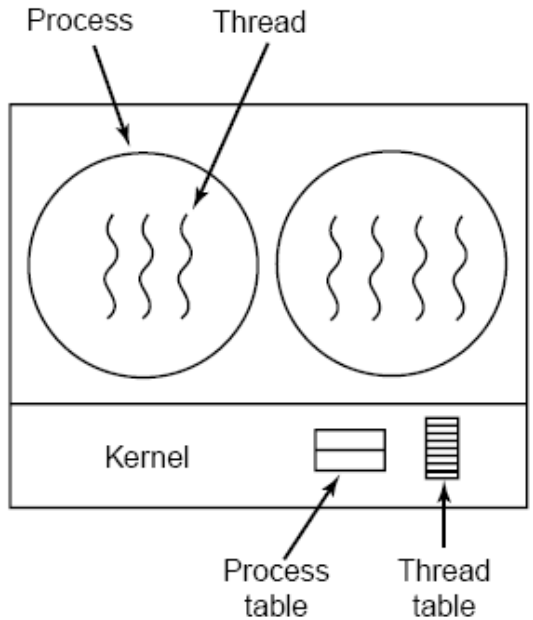
2.b Threads

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



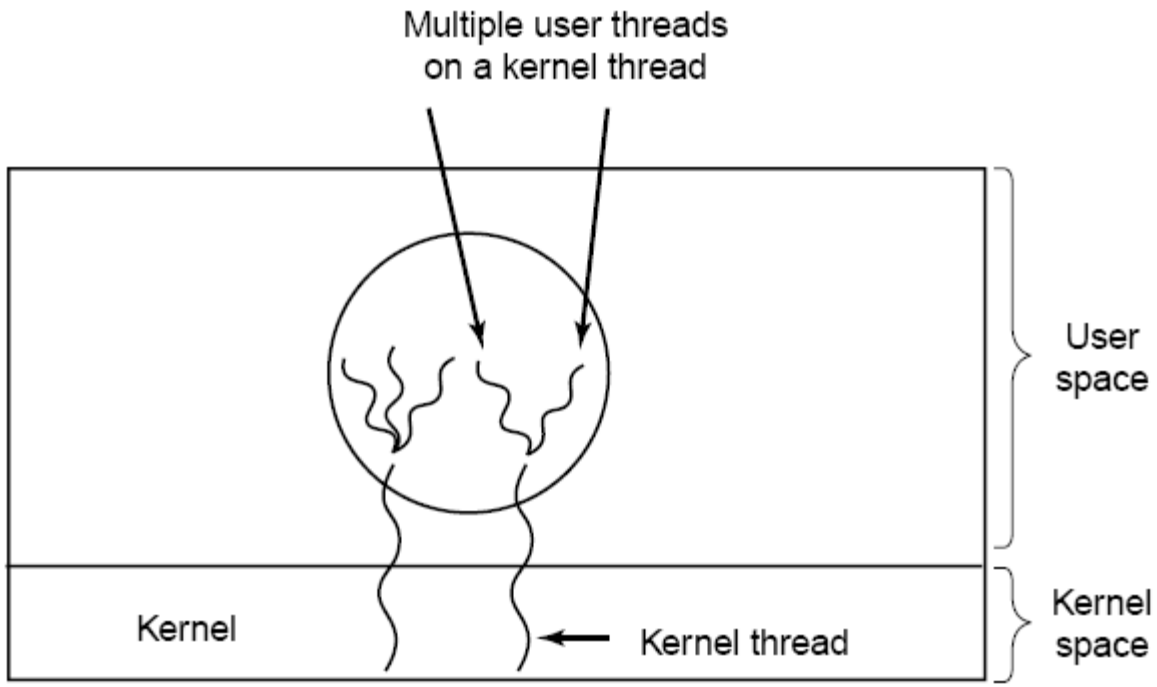
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

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