Threads

Threads

- Sometimes a program needs to do multiple tasks concurrently.
- Example: Word processor
  - Tasks include: Display graphics, respond to keystrokes from the user, and perform spelling and grammar checking.

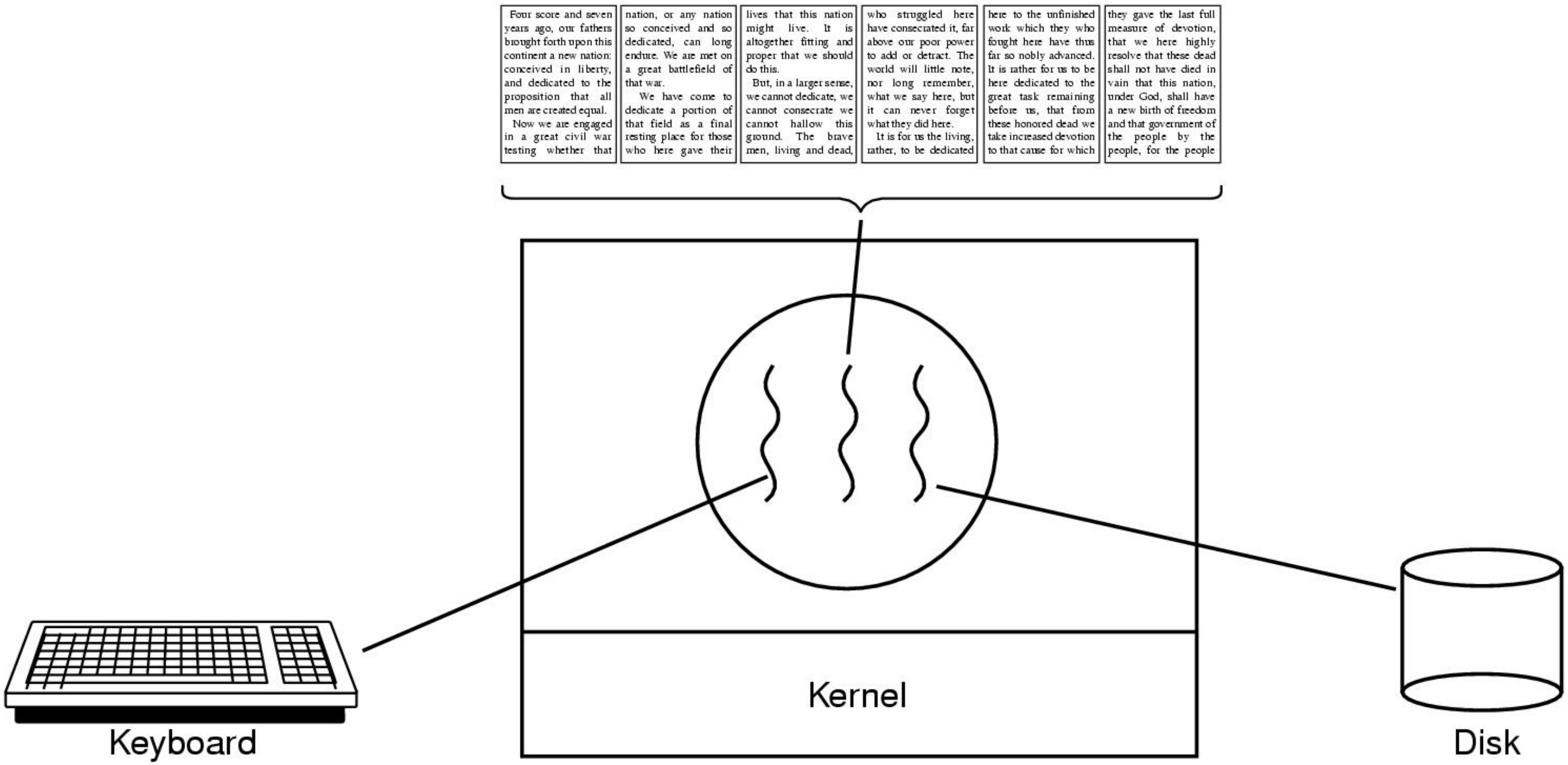
Introduction

- Earlier we discussed the use of forking to create a process
- For example, we could
  - Word processor example: fork a process for each task
  - Web server example: fork a process for each request
- It turns out there is a more efficient alternative (more later)

Threads

- A thread is a basic unit of CPU utilization
- Threads of a process share memory but can execute independently.
- A traditional process can be viewed as a memory address space with a single thread
- A thread is a sequence of instructions that execute sequentially.

Thread Usage - Word Processor



A word processor program with three threads.

Why Not Fork?

- You certainly can fork a new process
- In fact, the first implementation of Apache web servers (Apache 1.0) forked N processes when the web server was started
    - "N" was defined in a configuration file
    - Each child process handled one connection at a time
- Problem: Process creation is time consuming and resource intensive
- Creating threads is not as expensive

Processes vs Threads

- A beta version of Google's browser, Chrome, was released on 2 September 2008.
- Currently browsers allocate a thread for each tab
- Chrome allocates a process for each tab

Chrome Browser

- Chrome browser creates an entirely separate operating system process for every single tab or extra extension you are using.
- By separating out each tab and extension into a separate process, the browser can remain active even if a single tab must close
- Using multiple processes also results in faster surfing speeds as computer memory is allocated only to the currently open tab.
- https://smallbusiness.chron.com/disabling-multiple-processes-google-chrome-33767.html

Chrome Browser (cont.)

- At the core of Chrome browser's architecture is the concept of running each tab, plugin, and extension as a separate process, providing enhanced performance, security, and stability.
- By separating out each tab and extension into a separate process
  - the browser's memory requirement increases, and
  - A higher CPU utilization and power needs.

Question

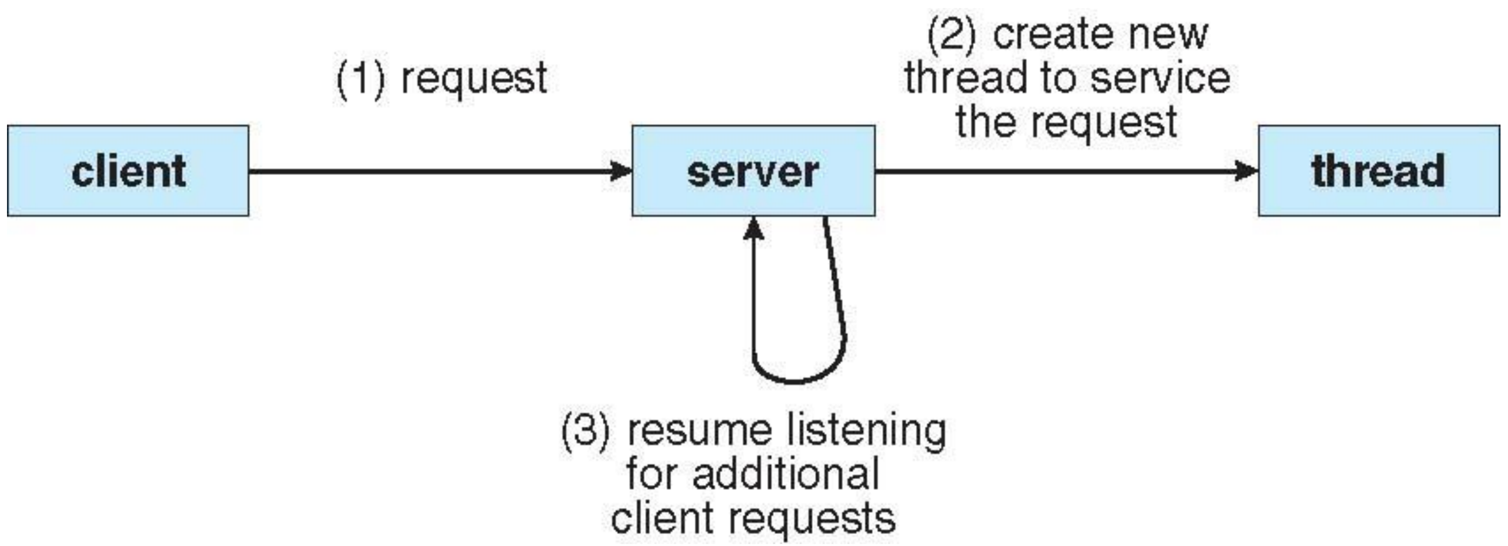
- Consider a multi-threaded web browser with one process
  - What is the disadvantage of this design?

Why Not Fork? (cont.)

- Let's look at web servers
  - This allowed a child process to handle multiple connections at a time
  - Web servers have caches for read pages
  - Forking means that these caches cannot be shared
  - Using threads allows for these caches to be shared

Why Not Fork? (cont.)

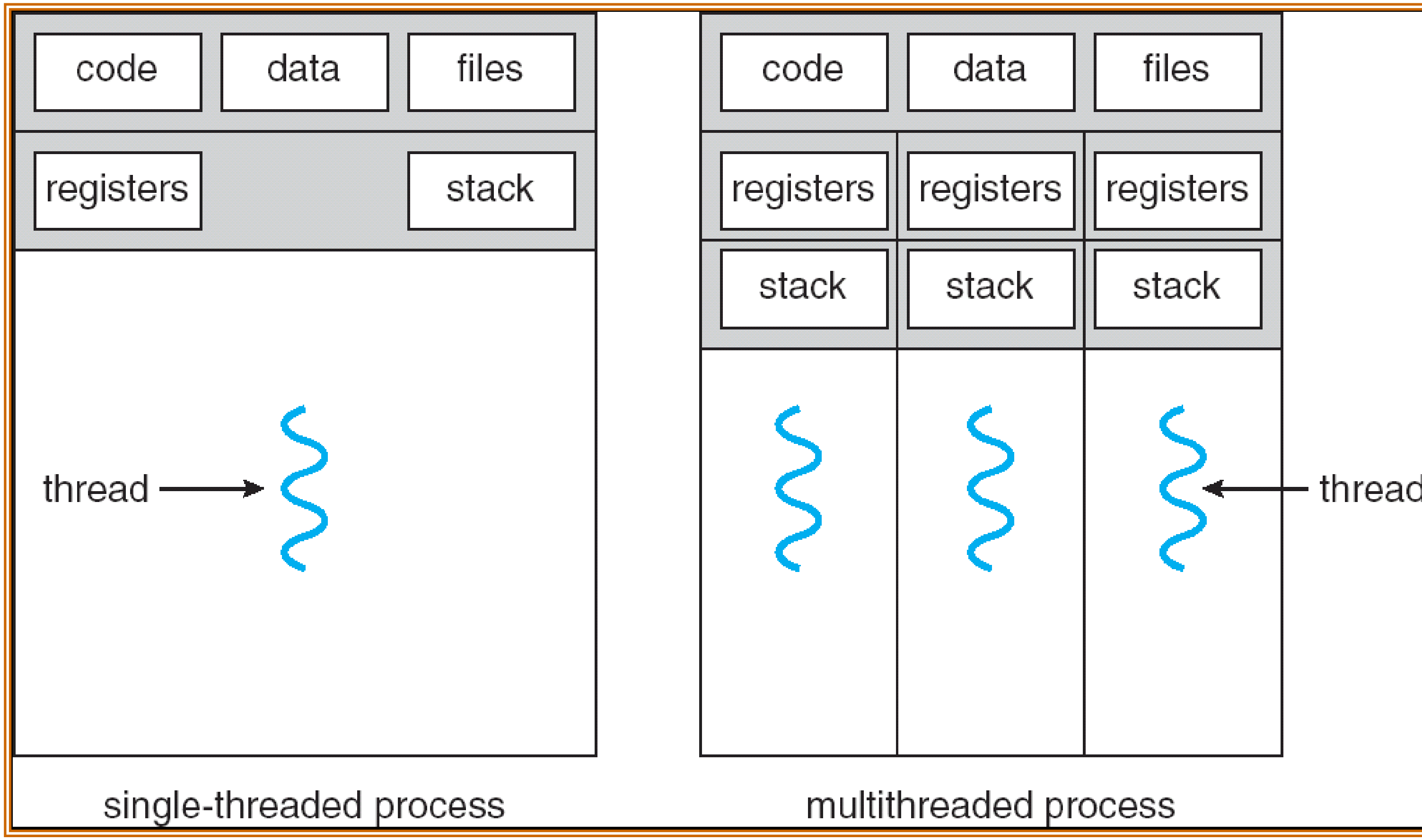
- Example: Web server
  - It is desirable to service requests concurrently



Thread State

- Threads share
  - Process address space
    - Text
    - Data (global variables)
    - Heap (dynamically allocated data)
  - OS state
    - Open files, sockets, locks
- Threads have their own CPU context
  - PC, SP, register state, stack
  - errno

Single and Multithreaded Processes



Benefits of Threads

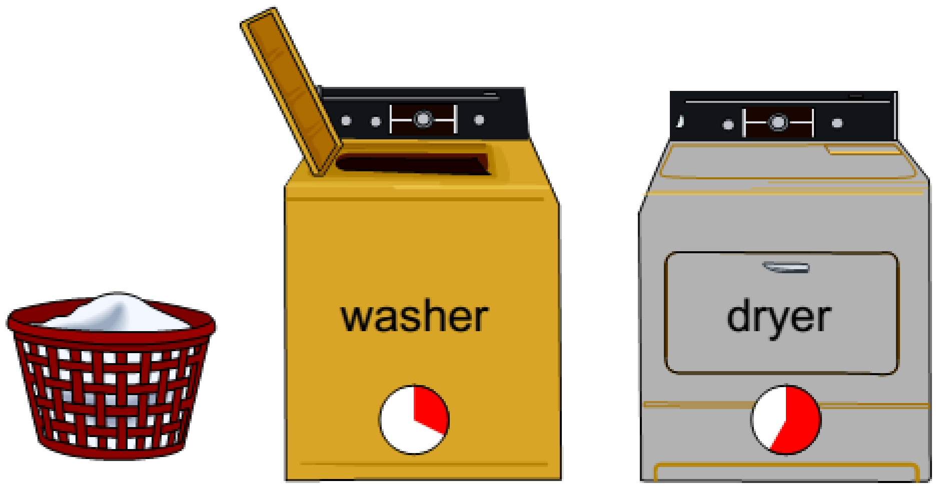
- Responsiveness
  - Overlap computation and blocking due to I/O on a single CPU
- Resource Sharing
  - Example: Word processor
    - The document is shared in memory.
    - Forking would require replication
- Allocating memory and resources for process creation is costly
- Context-switching is faster

Relationship between Latency and Throughput

- Latency and bandwidth only loosely coupled
  - Henry Ford: assembly lines increase bandwidth without reducing latency
- My factory takes 1 day to make a Model-T ford.
  - But I can start building a new car every 10 minutes
  - At 24 hrs/day, I can make 24 * 6 = 144 cars per day
  - A special order for 1 green car, still takes 1 day
  - Throughput is increased, but latency is not.
- Latency reduction is difficult
- Often, one can buy bandwidth
  - E.g., more memory chips, more disks, more computers
  - Big server farms (e.g., google) are high bandwidth

➢ In the laundry room

- ✓ the washing machine takes 20 minutes
- ✓ the dryer takes 40 minutes

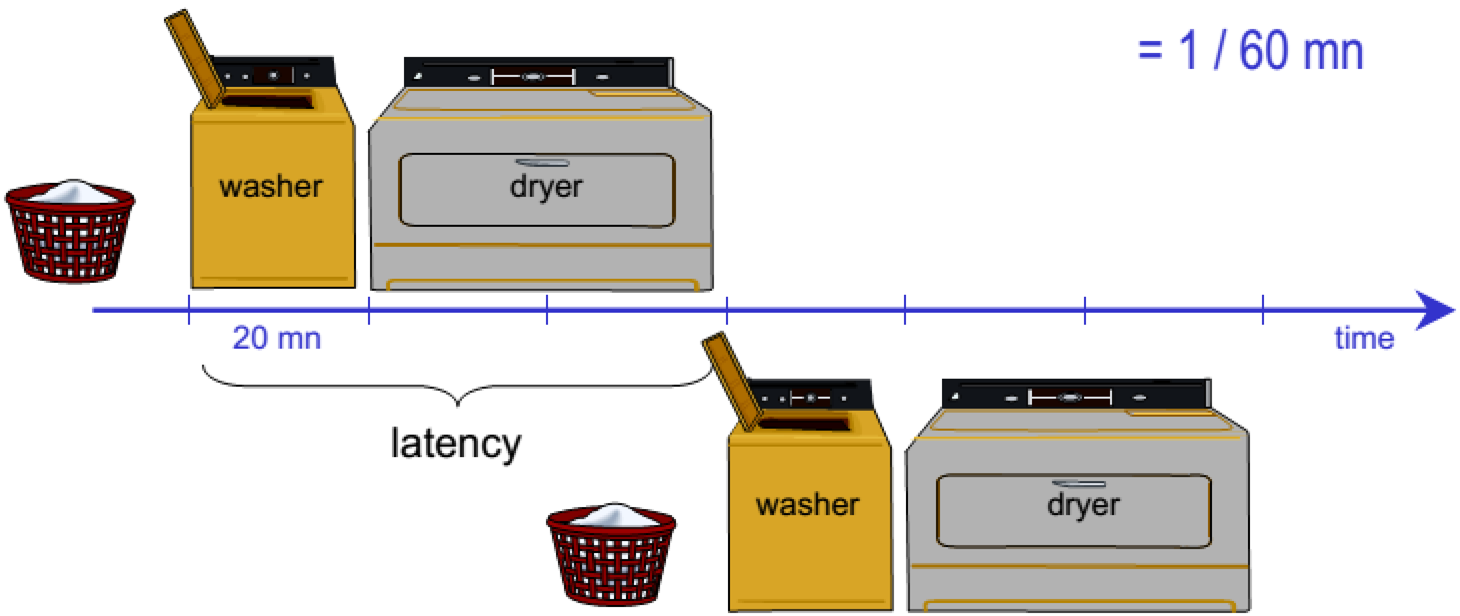


after Gill Pratt (2000) *How Computers Work.* ADUni.org/courses.

➢ 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



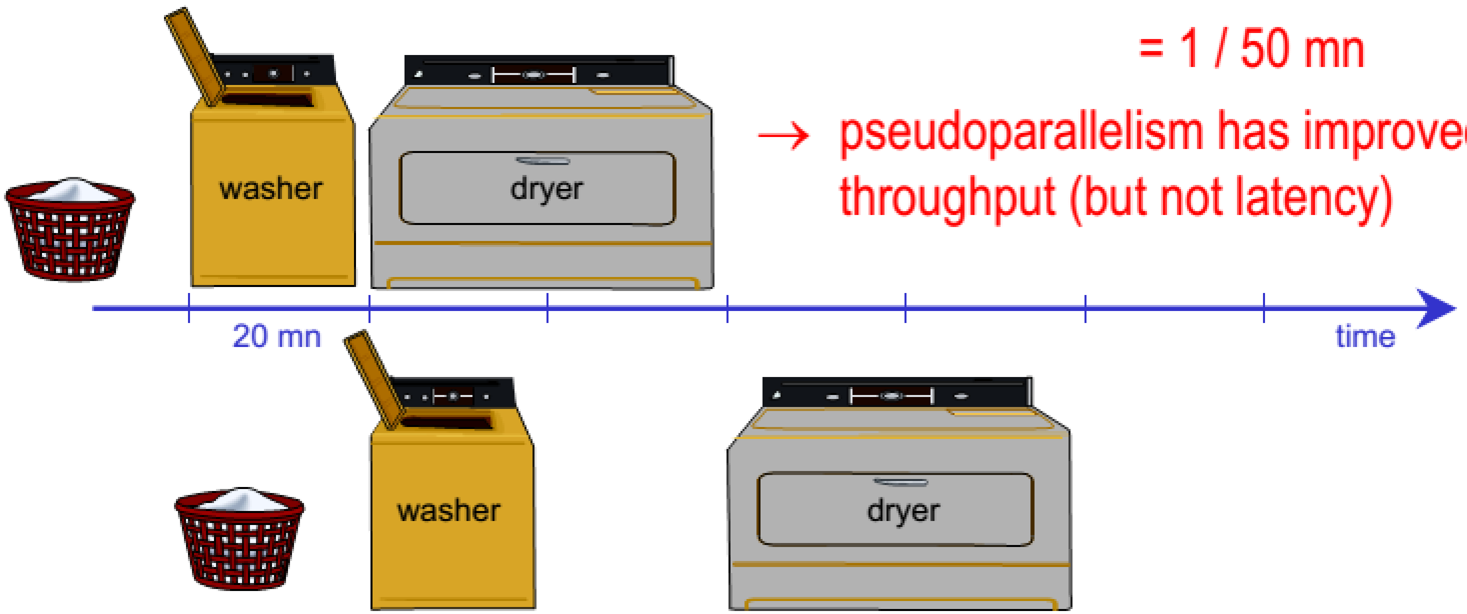
Two loads in a sequence

➢ 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

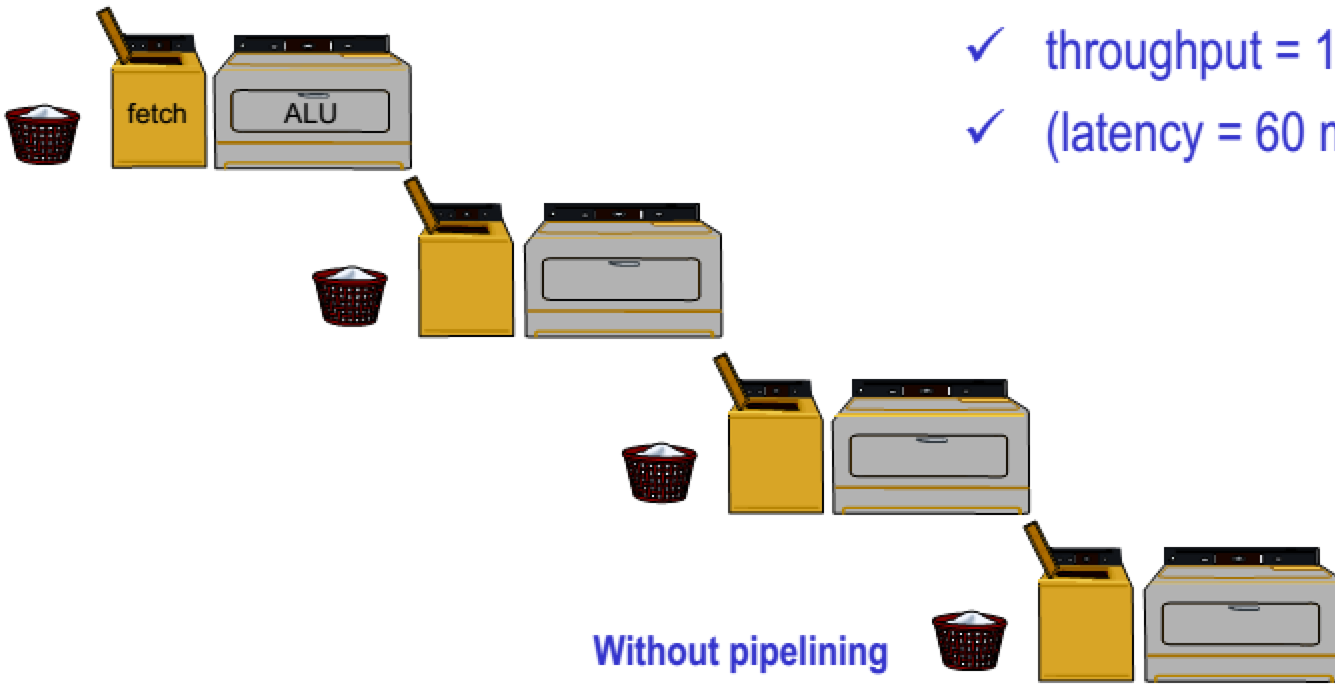
= 1 / 50 mn

→ pseudoparallelism has improved throughput (but not latency)



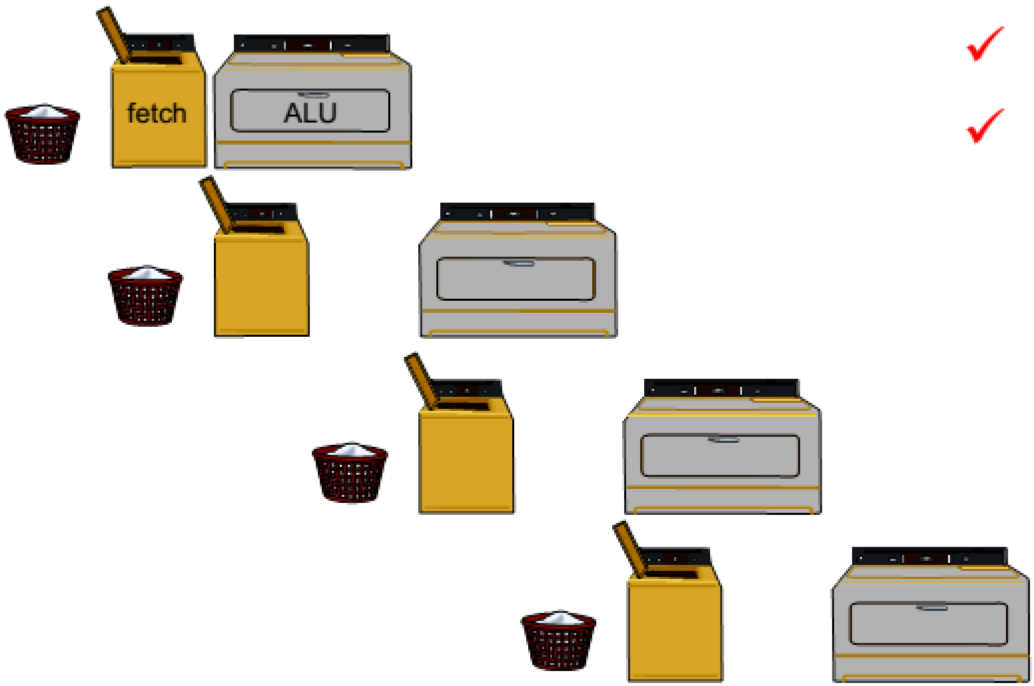
Two loads in parallel

- **This is the principle used in processor <u>pipelining</u>**
  - ✓ 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

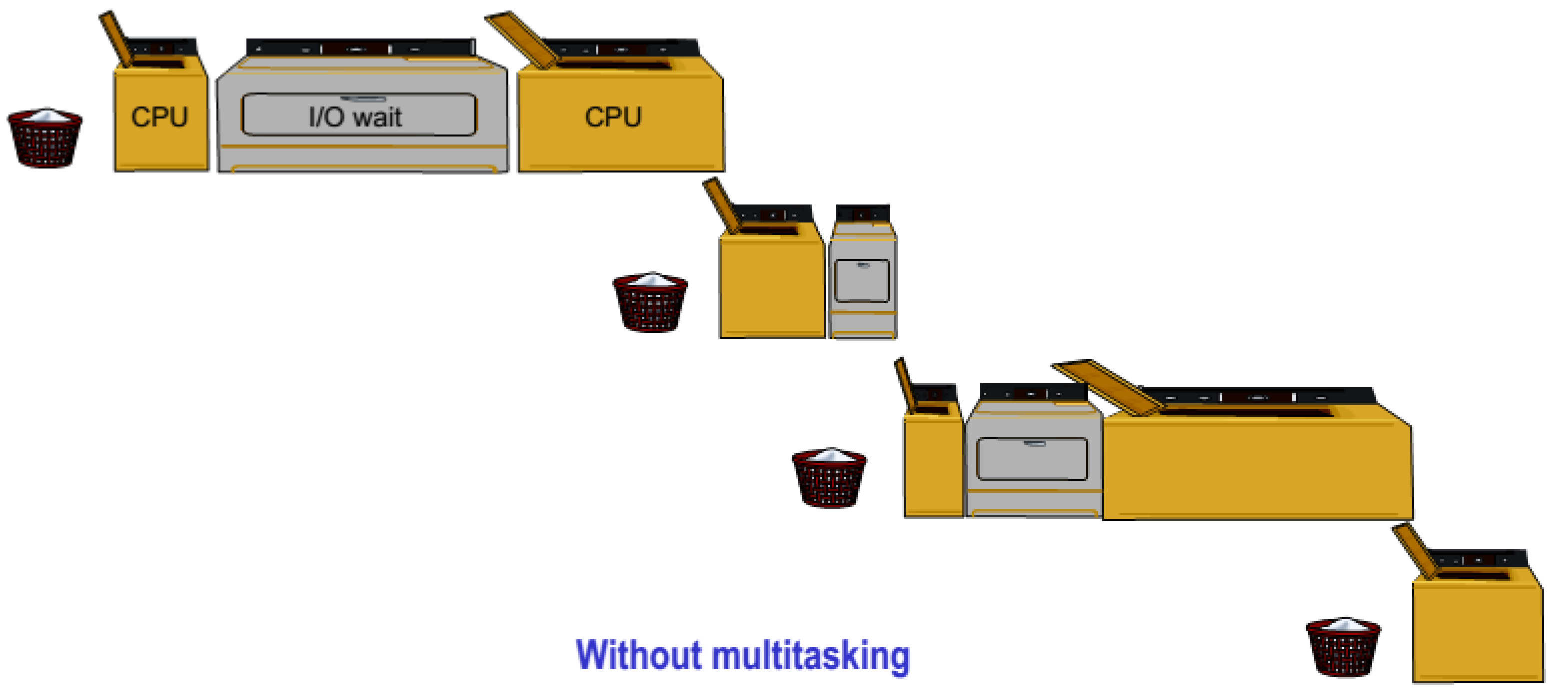
- ➢ **This is the principle used in processor pipelining**
  - ✓ here, washer & dryer are regularly clocked stages
  - ✓ with pipelining: throughput is only 1 over the longest stage



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

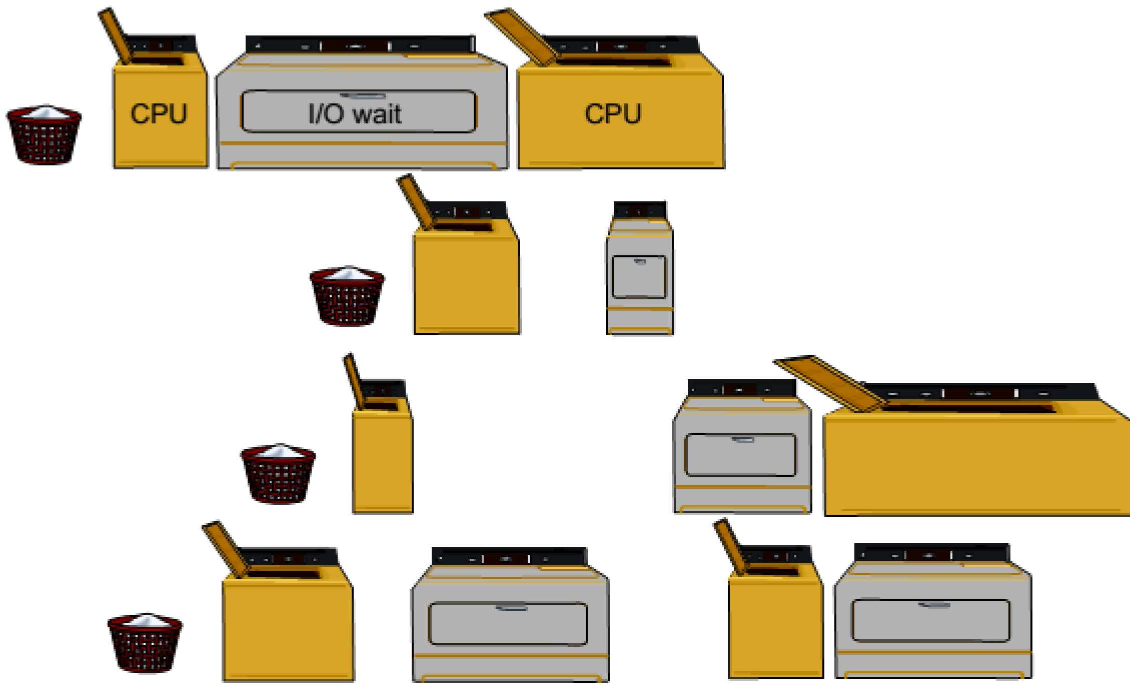
With pipelining

- **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



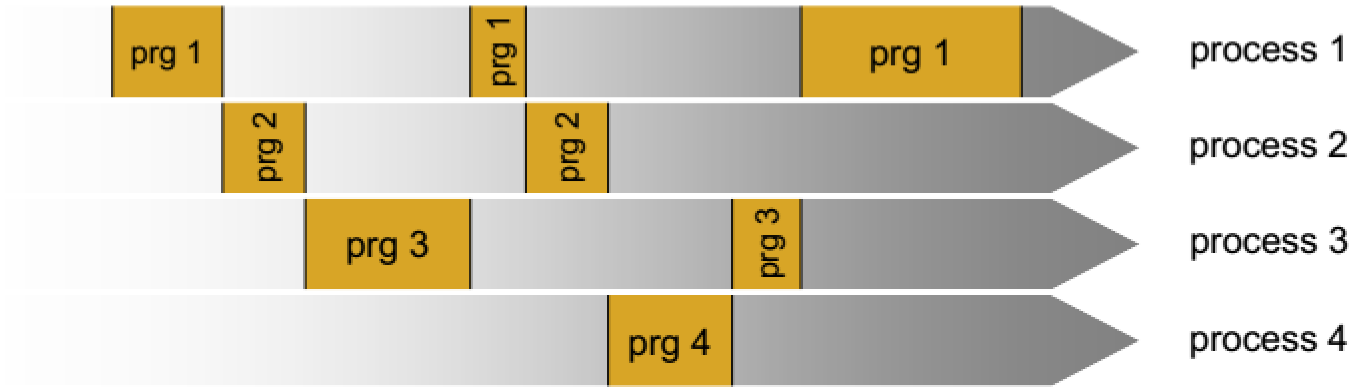
Without multitasking

- **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)



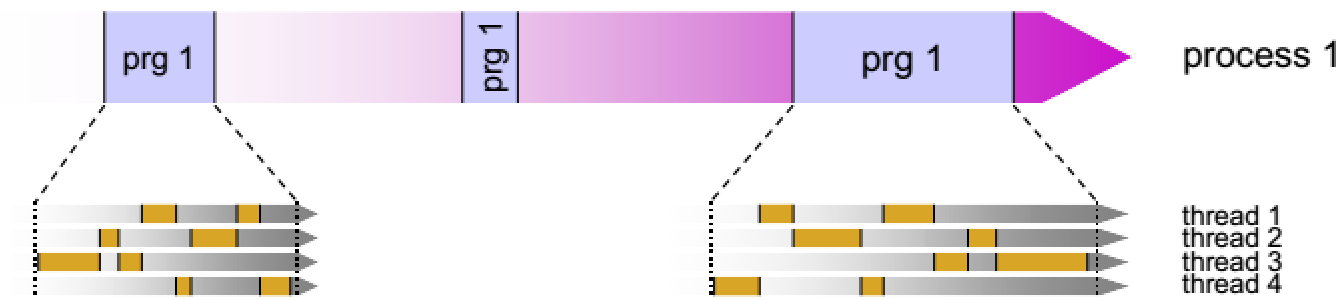
With multitasking

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



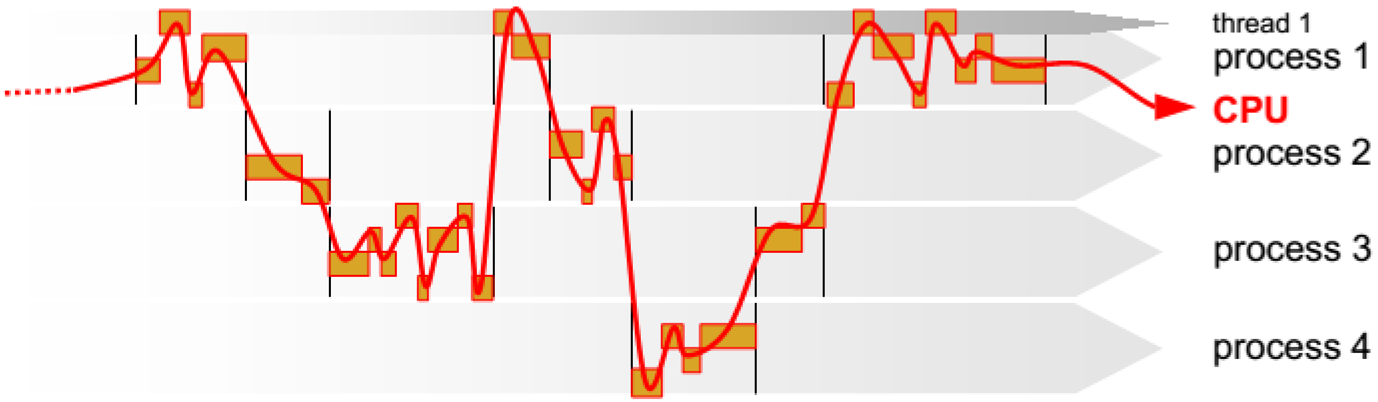
➢ And, naturally, the same idea applies in <u>multithreading</u>

- ✓ 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

- **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

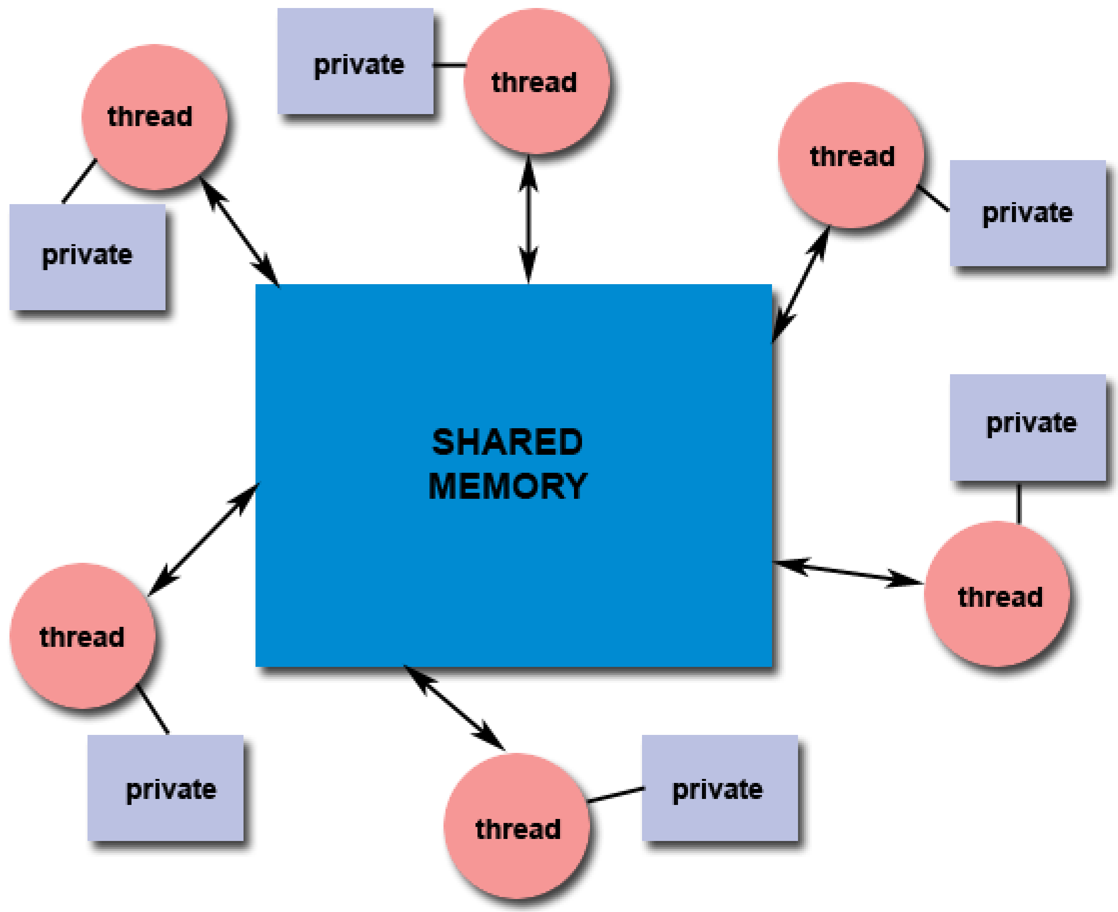


Multithreading

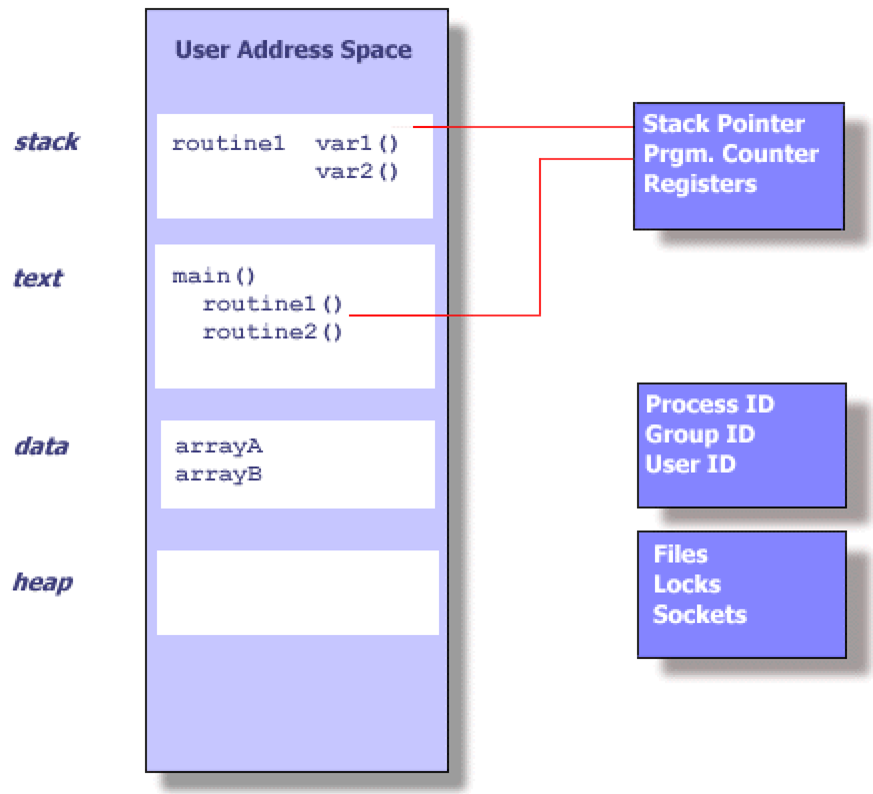
Shared Memory Model

- All threads have access to the same global, shared memory
- Threads also have their own private data
- **Programmers** are responsible for synchronizing access (protecting) shared data.

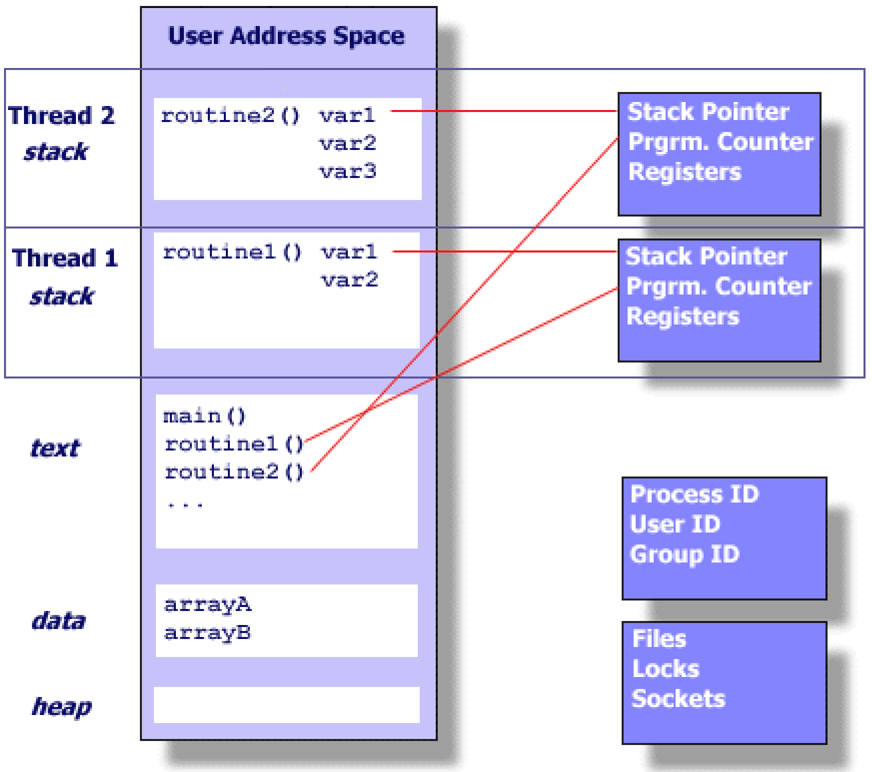
Shared Memory Model



Unix Process

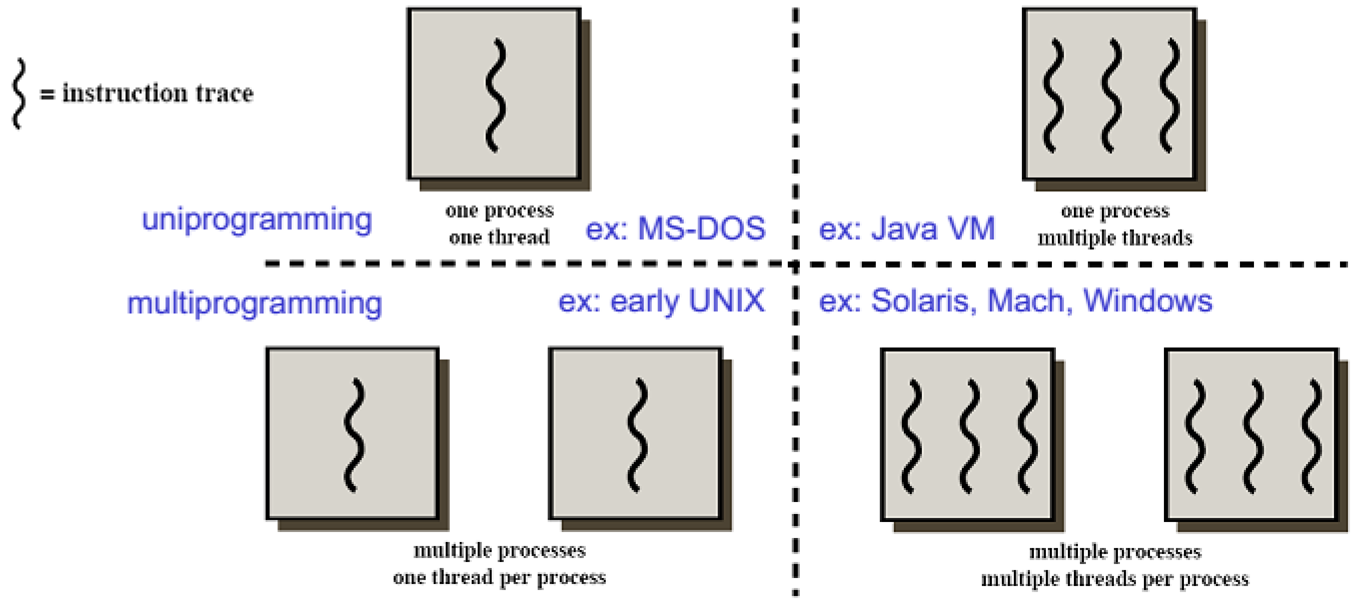


Threads within a Unix Process



➢ Multithreading

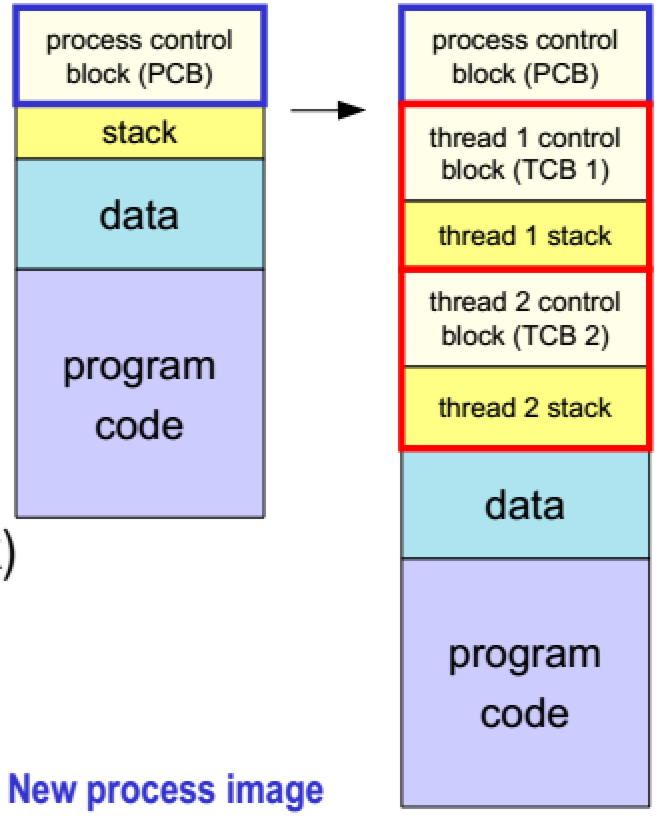
✓ refers to the ability of an operating system to support multiple threads of execution within a single process



Process-thread relationships

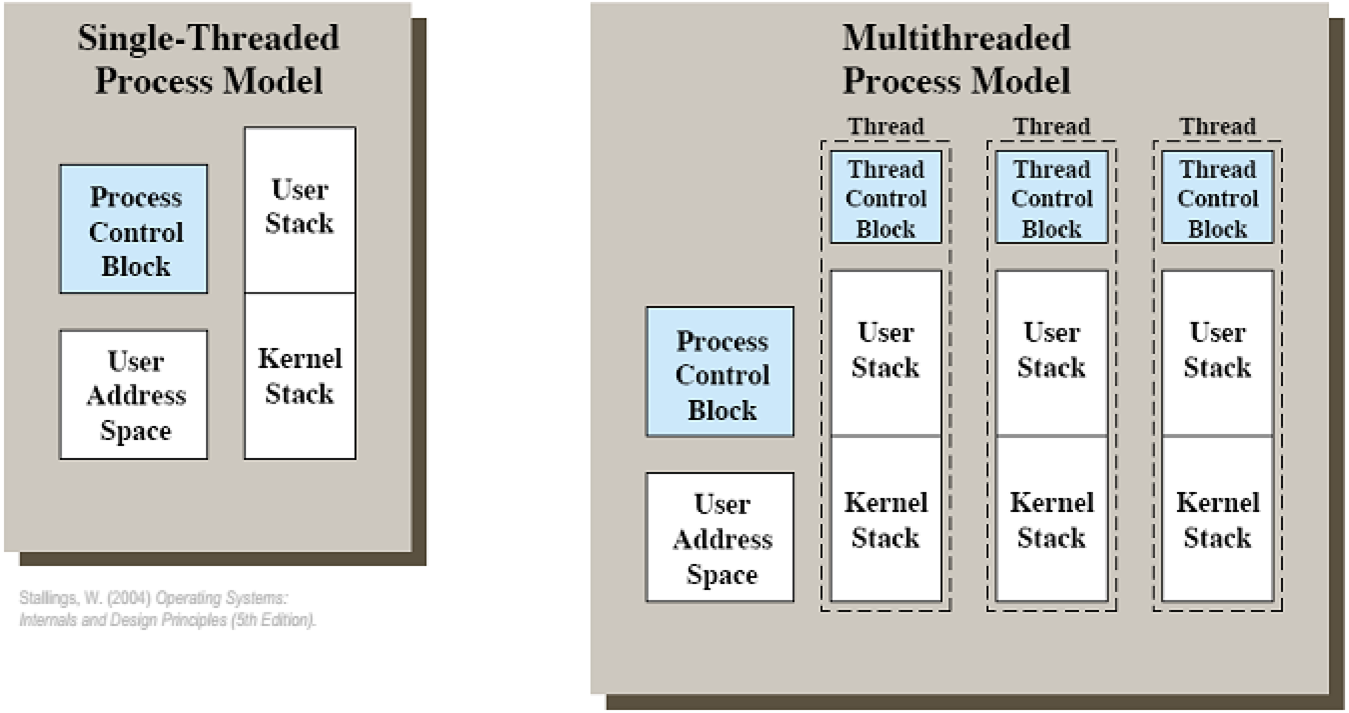
Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

- **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

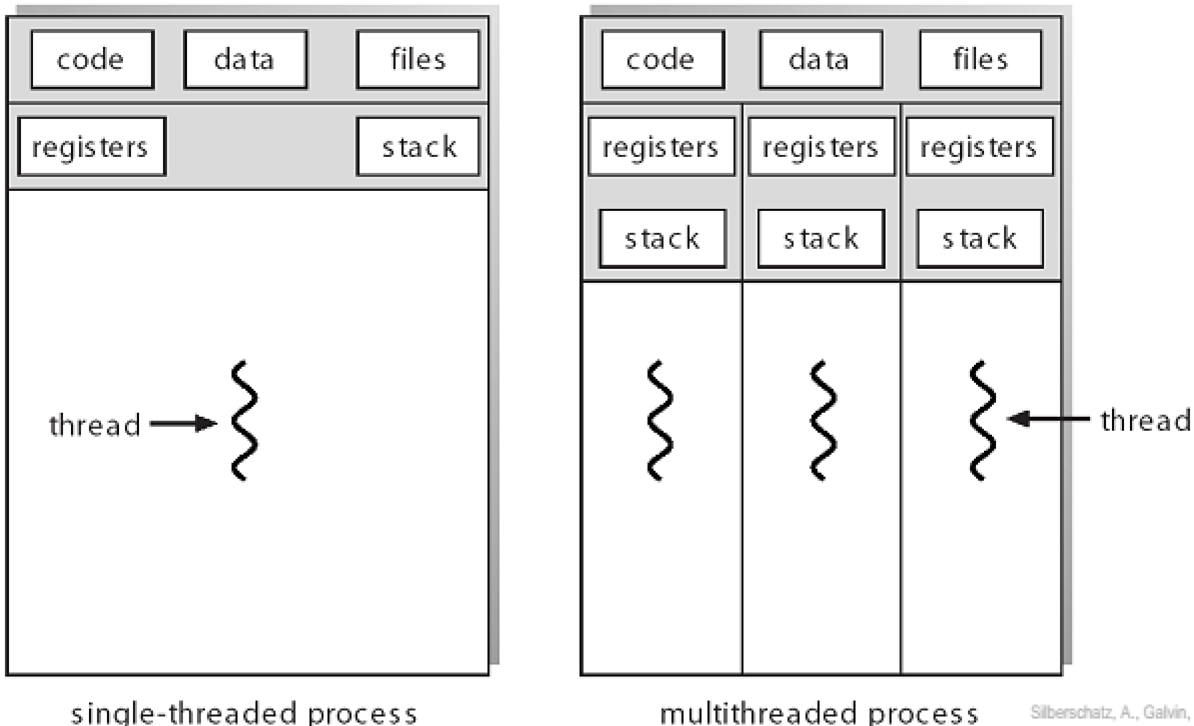
➢ Multithreaded process model (another view)



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

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

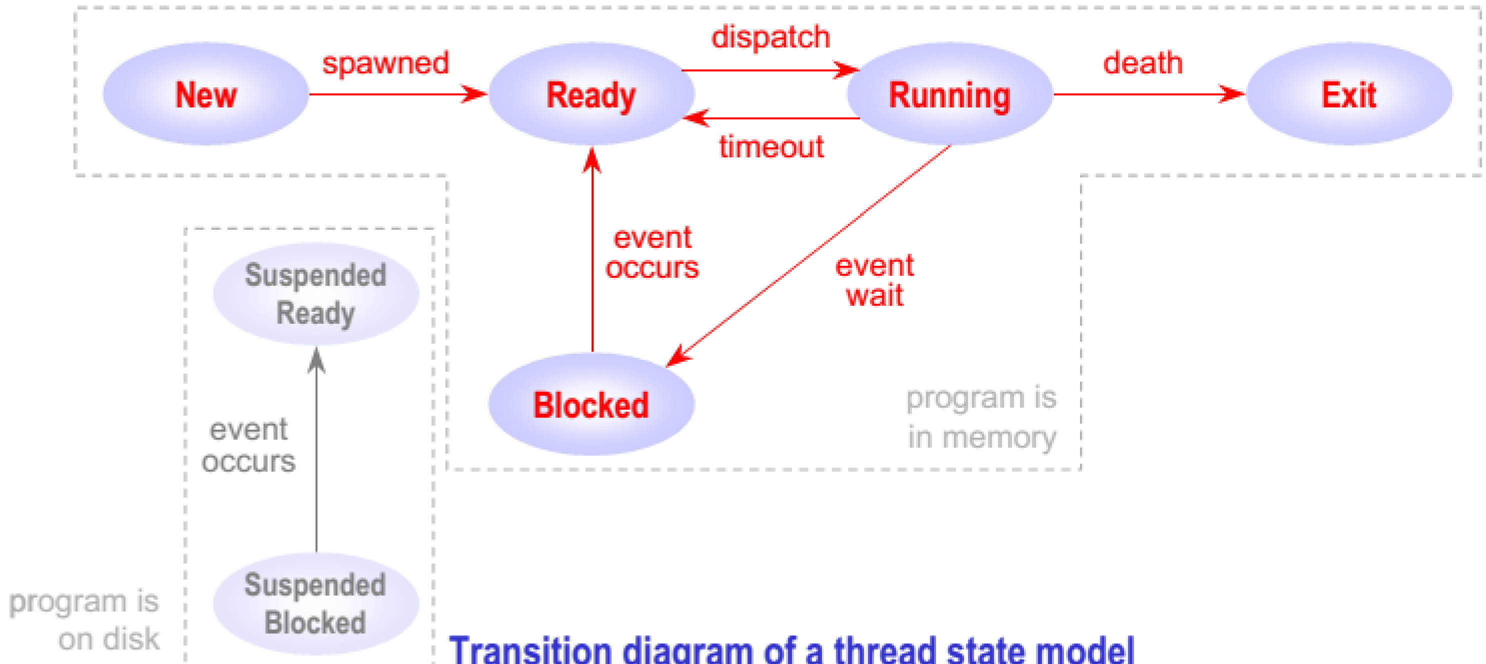
➢ Multithreaded process model (yet another view)



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

➢ Possible thread-level states

- ✓ threads (like processes) can be ready, running or blocked
- ✓ threads can’t be suspended (“swapped out”), only processes can



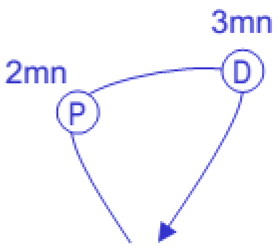
Transition diagram of a thread state model

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.

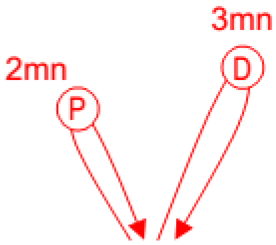
Illustration: two shopping scenarios

- 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

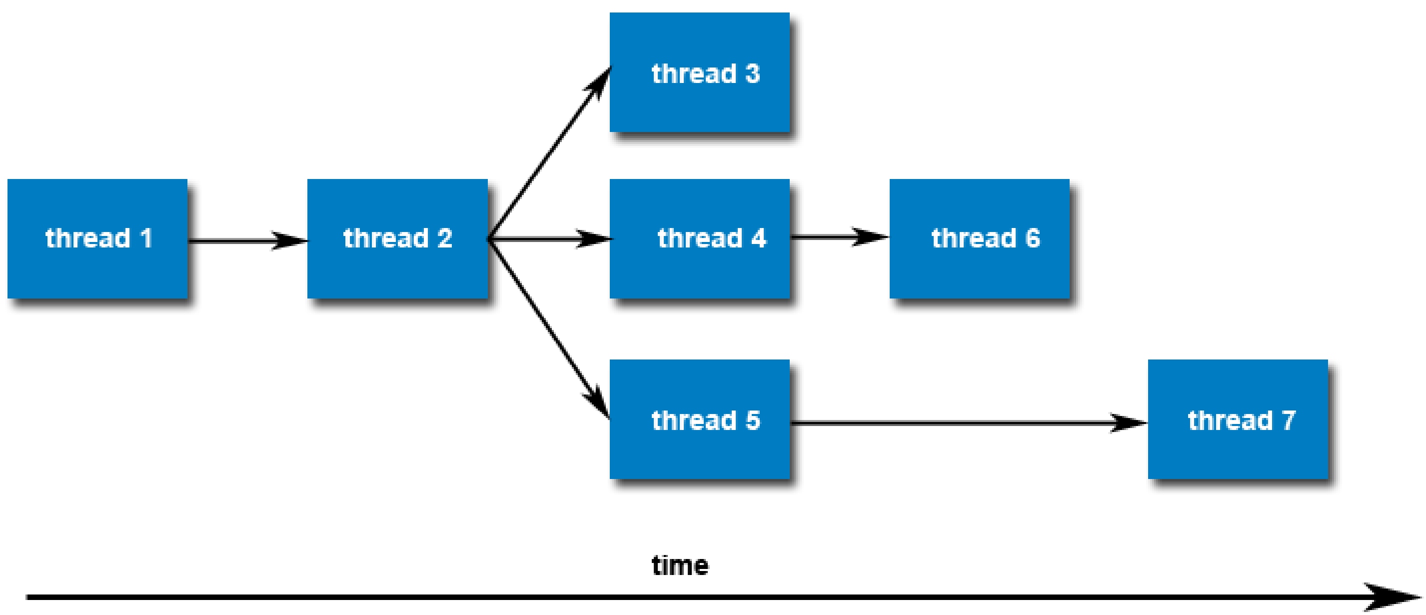
- Multithreaded shopping



  - 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

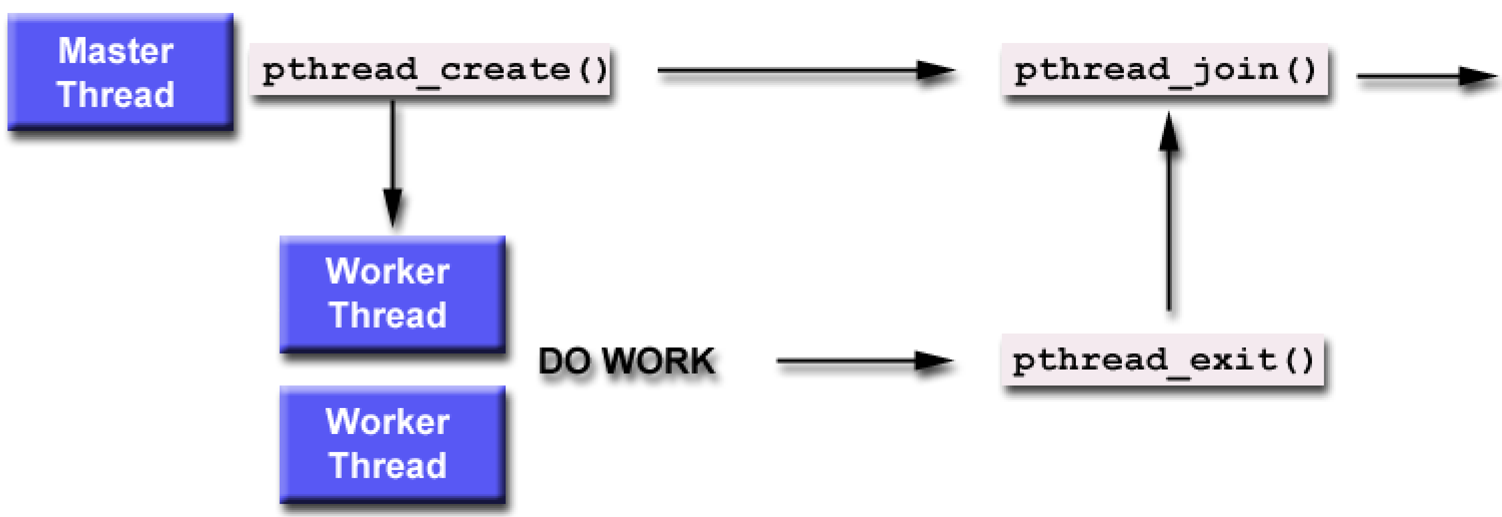
Creating Threads

- Initially *main()* runs a single default tread
- User may create any number of threads anywhere in the the his/her code
- Once created a thread may create other threads. No dependency between threads.

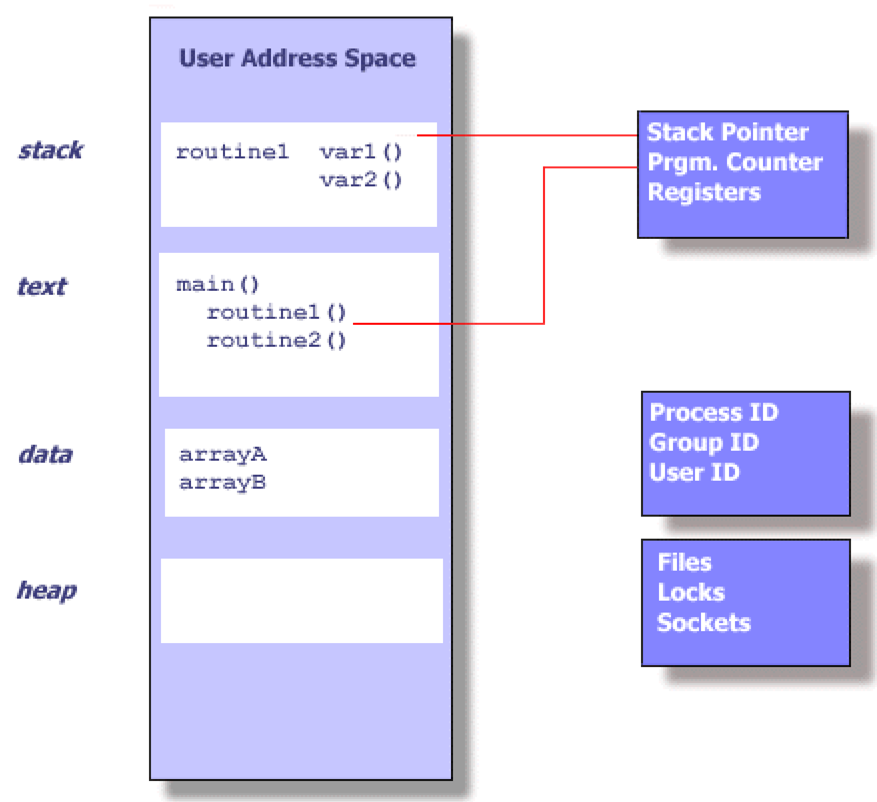


Creating Threads (cont.)

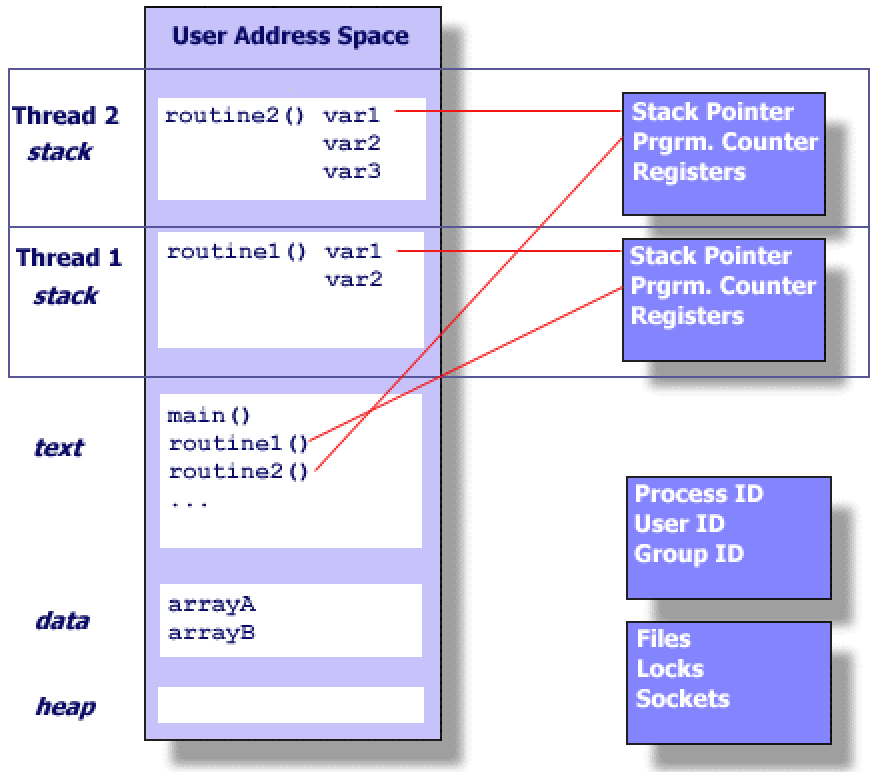
- Joining is one way to obtain synchronization between threads.
- pthread_join() blocks the calling thread until the specified threadid thread terminates.
- A joining thread can match one pthread_join() call



Unix Process



Threads within a Unix Process

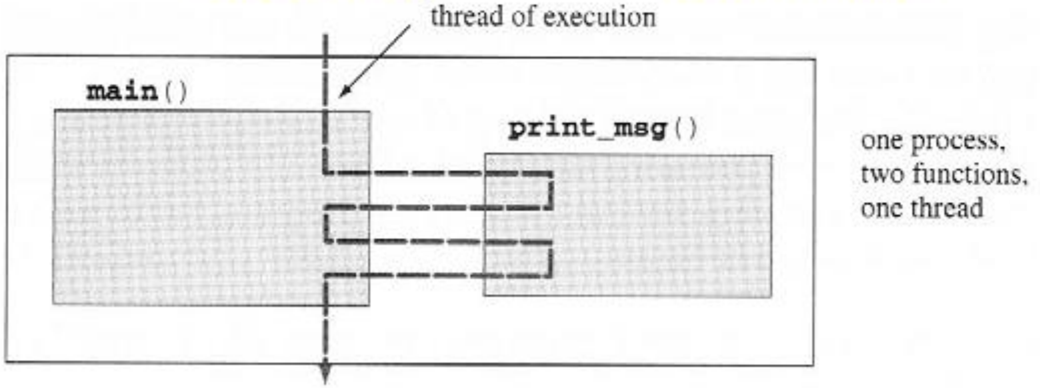


```c
void main(...)
{
    char *produce[] = { "salad", "apples", NULL };
    char *dairy[] = { "milk", "butter", "cheese", NULL };

    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

➢ 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

```
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;
}
```

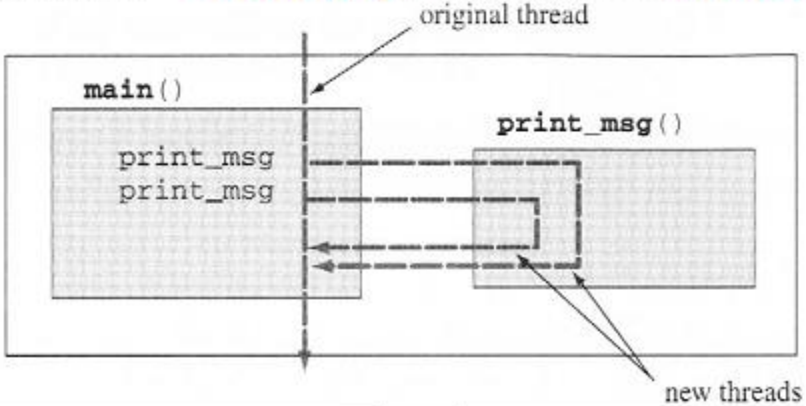
send the kids off!

wait for their return

Multithreaded shopping code

➢ 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

➢ 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`**

- **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

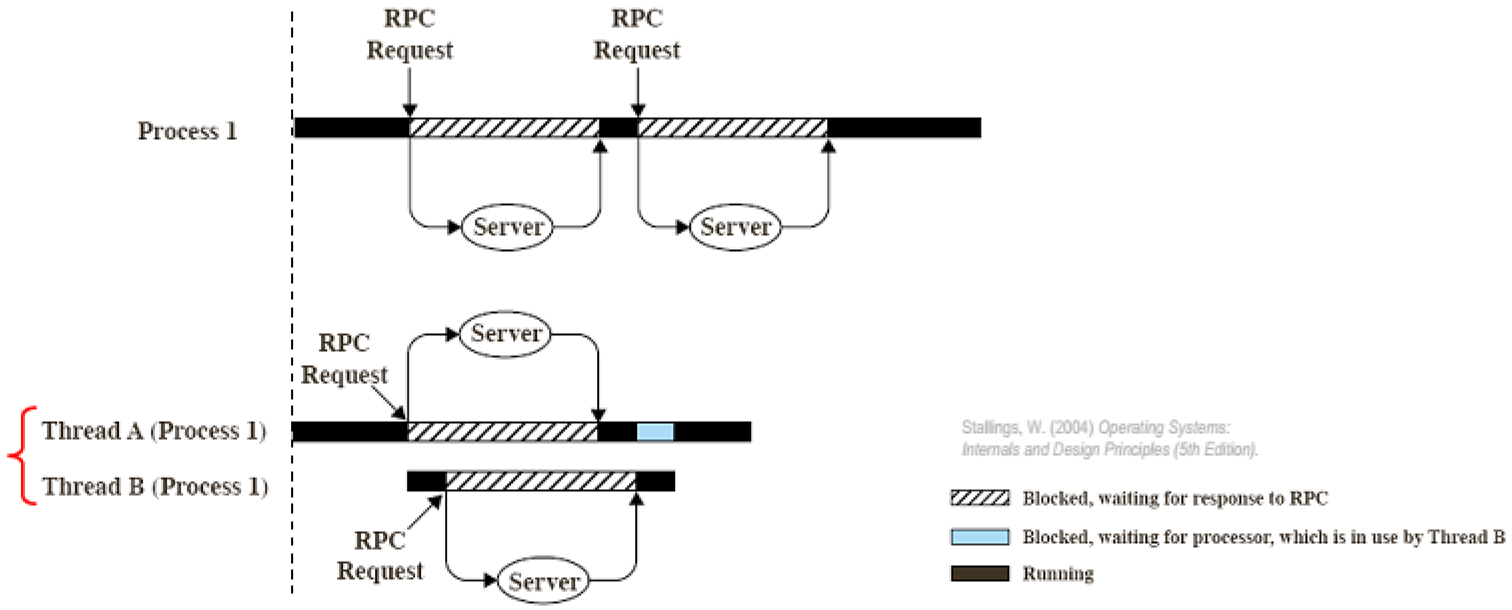
Examples of real-world multithreaded applications

- ✓ 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*

➢ Web client and Remote Procedure Calls (RPCs)

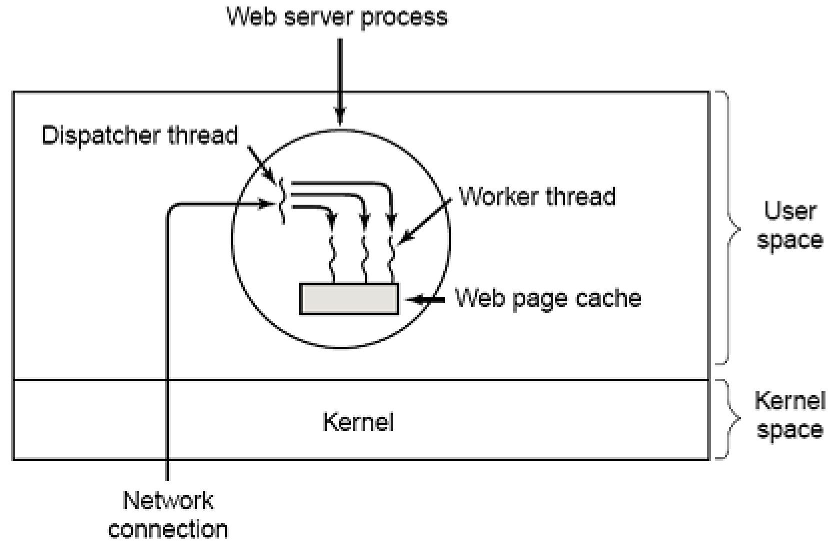
- ✓ the client uses multiple threads to send multiple requests to the same server or different servers, greatly increasing performance



Client RPC using a single thread vs. multiple threads

➢ 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”)

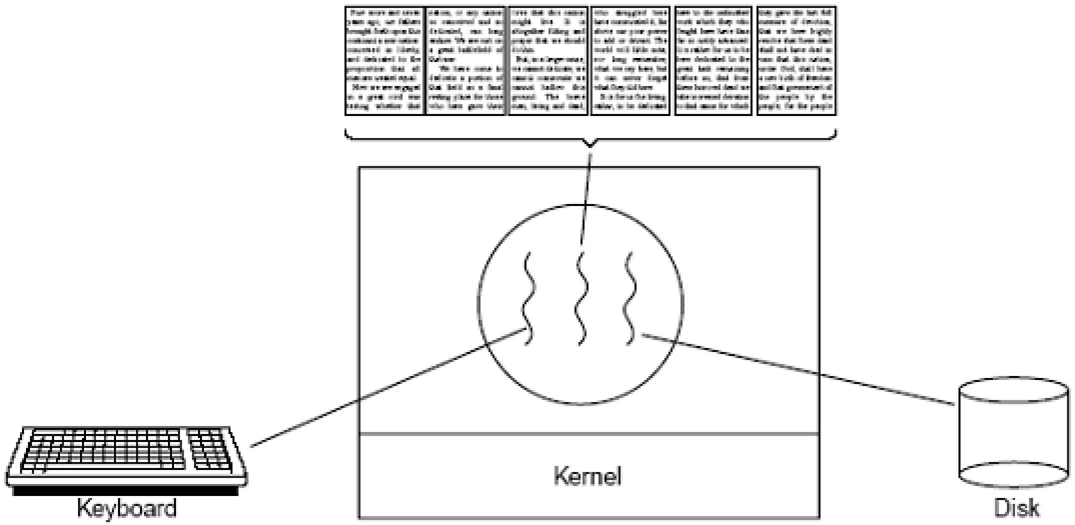


Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

A multithreaded Web server

➢ 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

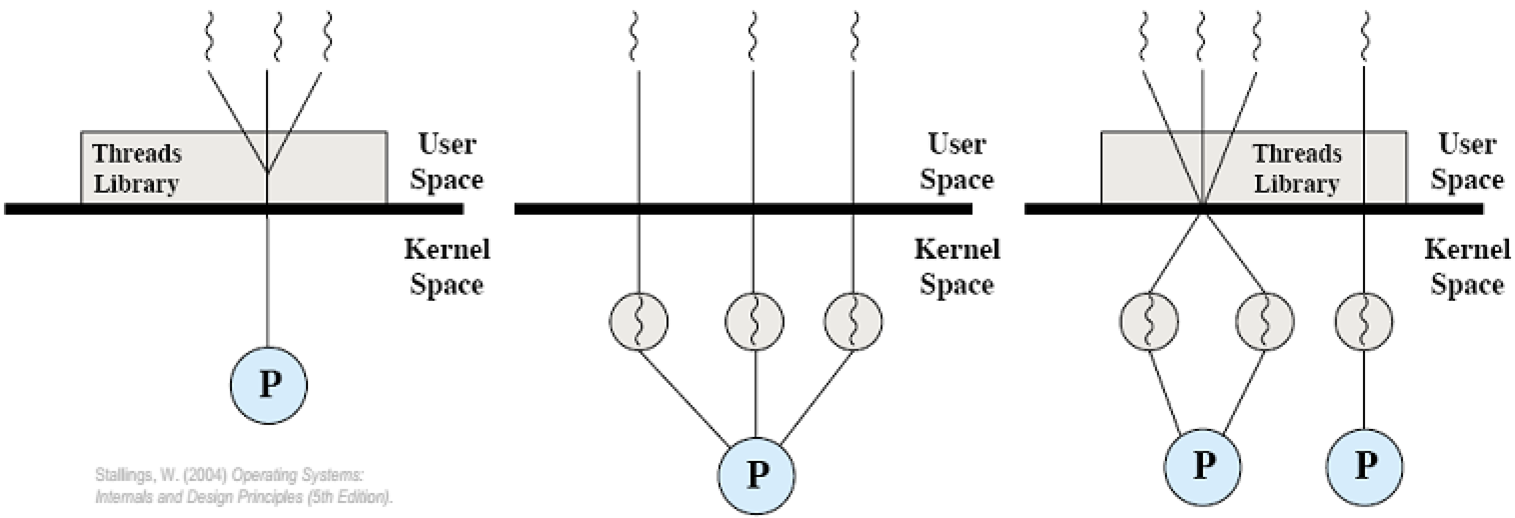


Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition)

A word processor with three threads

- **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)

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

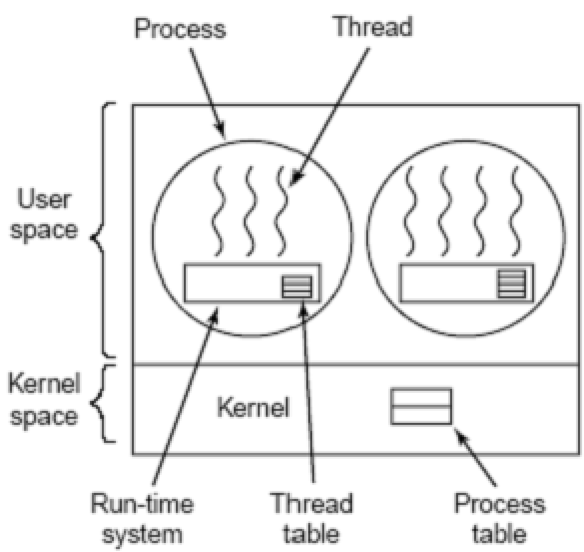


Stallings, W. (2004) *Operating Systems: Internals and Design Principles (5th Edition).*

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

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



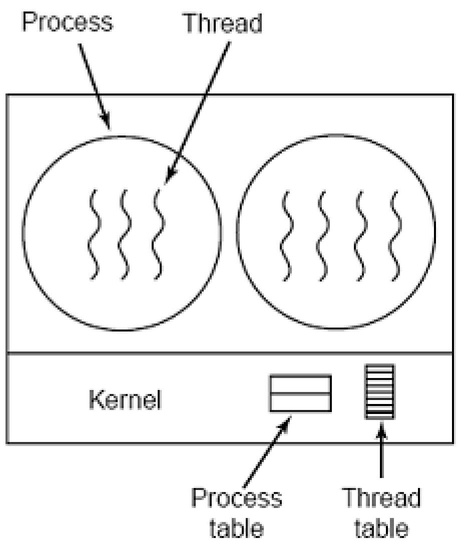
Tanenbaum, A. S. (2001)
Modern Operating Systems (2nd Edition).

A user-level thread package

➢ Kernel-Level Threads (KLTs)

✓ 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

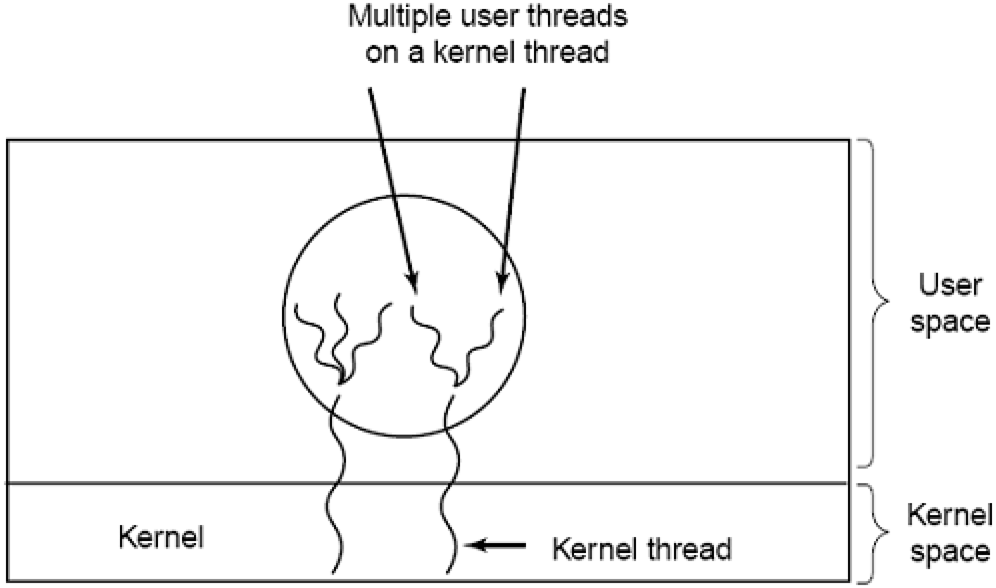


Tanenbaum, A. S. (2001) *Modern Operating Systems (2nd Edition).*

A kernel-level thread package

➢ Hybrid implementation

- ✓ combine both approaches: graft ULTs onto KLTs



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

Multiplexing ULTs onto KLTs

Pthreads: POSIX Threads

- Pthreads is a standard set of C library functions for multithreaded programming
  - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
- Programs must include the file `pthread.h`
- Programs must be linked with the pthread library (`-lpthread`)
  - Done by default by some gcc's (e.g., on Mac OS X)

pthread_create()

- Creates a new thread

```
int pthread_create (
            pthread_t *thread,
            pthread_attr_t *attr,
            void * (*start_routine) (void *),
            void *arg);
```

  - Returns 0 to indicate success, otherwise returns error code
  - **thread**: output argument for the id of the new thread
  - **attr**: input argument that specifies the attributes of the thread to be created (NULL = default attributes)
  - **start_routine**: function to use as the start of the new thread
    - must have prototype: void * foo(void*)
  - **arg**: argument to pass to the new thread routine
    - If the thread routine requires multiple arguments, they must be passed bundled up in an array or a structure

pthread_create() example

- Let us say that you want to create a thread to compute the sum of the elements of an array

```
void *do_work(void *arg);
```

- Needs three arguments
  - the array, its size, and where to store the sum
  - we need to bundle these arguments in a structure

```
struct arguments {
        double *array;
        int size;
        double *sum;
}
```

pthread_create() example

```
int main(int argc, char *argv) {
  double array[100];
  double sum;
  pthread_t worker_thread;
  struct arguments *arg;

  arg = (struct arguments *)calloc(1,
                             sizeof(struct arguments));
  arg->array = array;
  arg->size=100;
  arg->sum = &sum;

  if (pthread_create(&worker_thread, NULL,
                     do_work, (void *)arg)) {
    fprintf(stderr,"Error while creating thread\n");
    exit(1);
  }
  ...
}
```

pthread_create() example

```
void *do_work(void *arg) {
 struct arguments *argument;
 int i, size;
 double *array;
 double *sum;

 argument = (struct arguments*)arg;

 size = argument->size;
 array = argument->array;
 sum = argument->sum;

 *sum = 0;
 for (i=0;i<size;i++)
    *sum += array[i];

 return NULL;
}
```

Thread Creation

- Thread identifiers
  - Each thread has a unique identifier (ID), a thread can find out its ID by calling pthread_self().
  - Thread IDs are of type pthread_t which is usually an unsigned int. When debugging it's often useful to do something like this:

```
printf("Thread %u: blah\n",pthread_self());
```

Shared Global Variables

All threads within a process can access global variables.

```
int counter=0;
void *blah(void *arg) {
    counter++;
    printf("Thread %u is number %d\n",
                  pthread_self(),counter);
}
main() {
  int i; pthread_t threadid;
  for (i=0;i<10;i++)
        pthread_create(&threadid,NULL,blah,NULL);
}
```

Problem

- Sharing global variables is dangerous - two threads may attempt to modify the same variable at the same time.
- pthreads includes support for mutual exclusion primitives that can be used to protect against this problem.
- The general idea is to lock something before accessing global variables and to unlock as soon as you are done.
- More on this topic later in the course

User Space Threads

- All code and data structures for the library exist in user space
- Invoking a function in the library results in a local function call in user space and not a system call
- Kernel knows nothing about the threads package

User Space Threads

- This requires that each process has its own private thread table to keep track of the threads in that process.
- The entries of the table contain information for a specific thread
  - Thread's program counter
  - Thread's stack counter
  - Thread's registers
  - Etc
- Switching of threads requires that the values of stack pointer and program counter be changed

Thread Libraries

- Three main libraries in use
  - POSIX PThreads
    - User or kernel level
  - Win32
    - Kernel level library
    - Used in Windows OS
  - Java threads
    - JVM is running on top of a host OS
    - The Java thread API is implemented using a thread library available on the system

```c
#include <pthread.h>
#include <stdio.h>

int sum;
void *runner(void *param);

int main(int argc, char *argv[]){
  pthread_t tid;
  pthread_attr_t attr;

  if( argc != 2 ){
    fprintf(stderr, "usage: a.out <integer
value>\n");
    return( -1 );
  }
  if( atoi(argv[1]) < 0  ){
    fprintf(stderr, "%d must be
 >=0\n", atoi(argv[1]));
    return( -1 );
  }
  printf("here\n");
  pthread_attr_init(&attr);
  pthread_create(&tid, &attr, runner, argv[1]);
  pthread_join(tid, NULL);
  printf("sum = %d\n", sum);
}
```

```c
#include <pthread.h>
#include <stdio.h>



void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  for( i = 1; i <= upper; i++ )
    sum += i;

  pthread_exit(0);
}
```

Summary

- Introduction to the concept of threads
- There will be more discussion throughout the course