CSE 384: Lecture 20 (3/30/20) Spring 2020, Mike C.

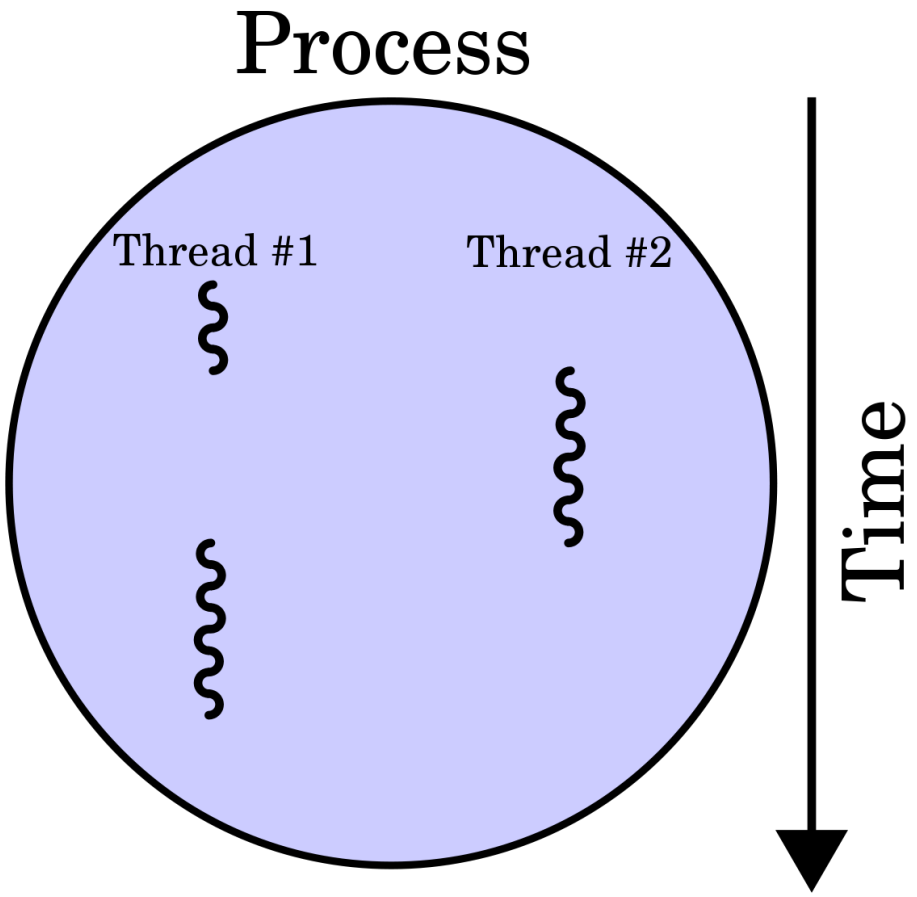
Some slides from Dr. Fawcett's "Threads and Locks" Presentation)

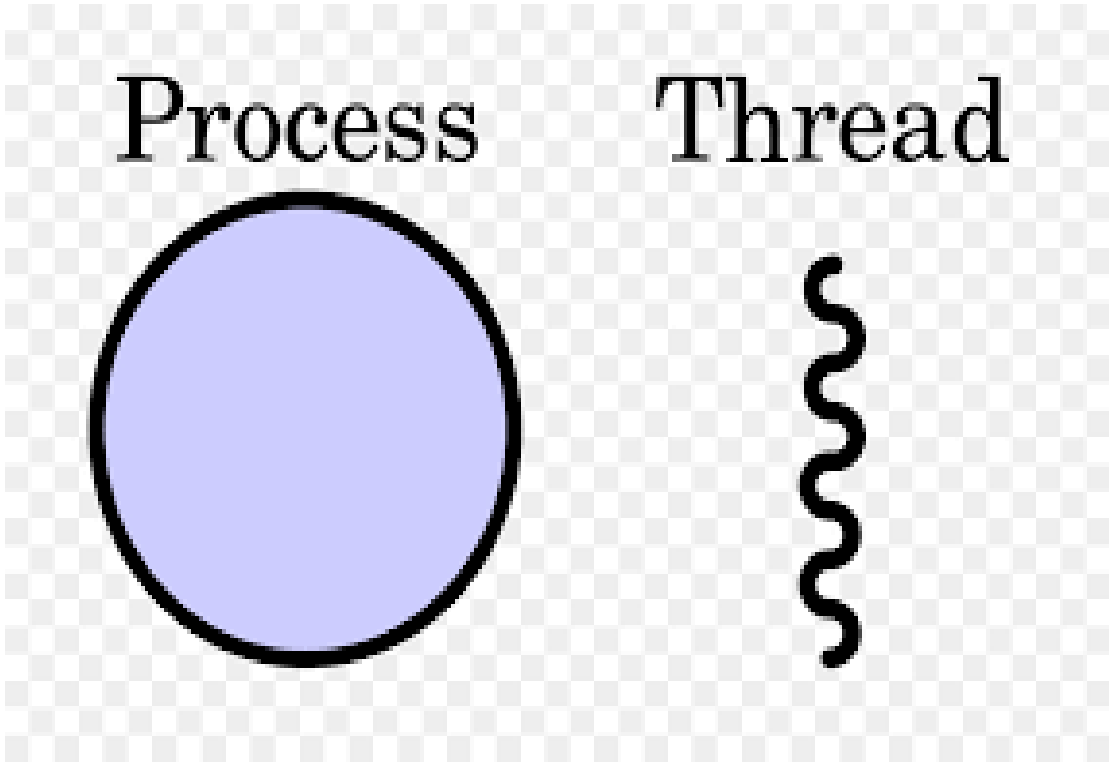
https://ecs.syr.edu/faculty/fawcett/handouts/CoreTechnologies/ThreadsAndSynchronization/presentations/ThreadsWinAndCpp11.pdf

What is a Thread?

- A thread is a path of execution through a program's code, plus a set of resources (stack, register state, etc) assigned by the operating system.
- A thread lives in one and only one process. A process may have one or more threads.
- Each thread in the process has its own call stack, but shares process code and global data with other threads in the process.
  - Thus local data is unique to each thread
- Pointers are process specific, so threads can share pointers.

Process Versus Thread



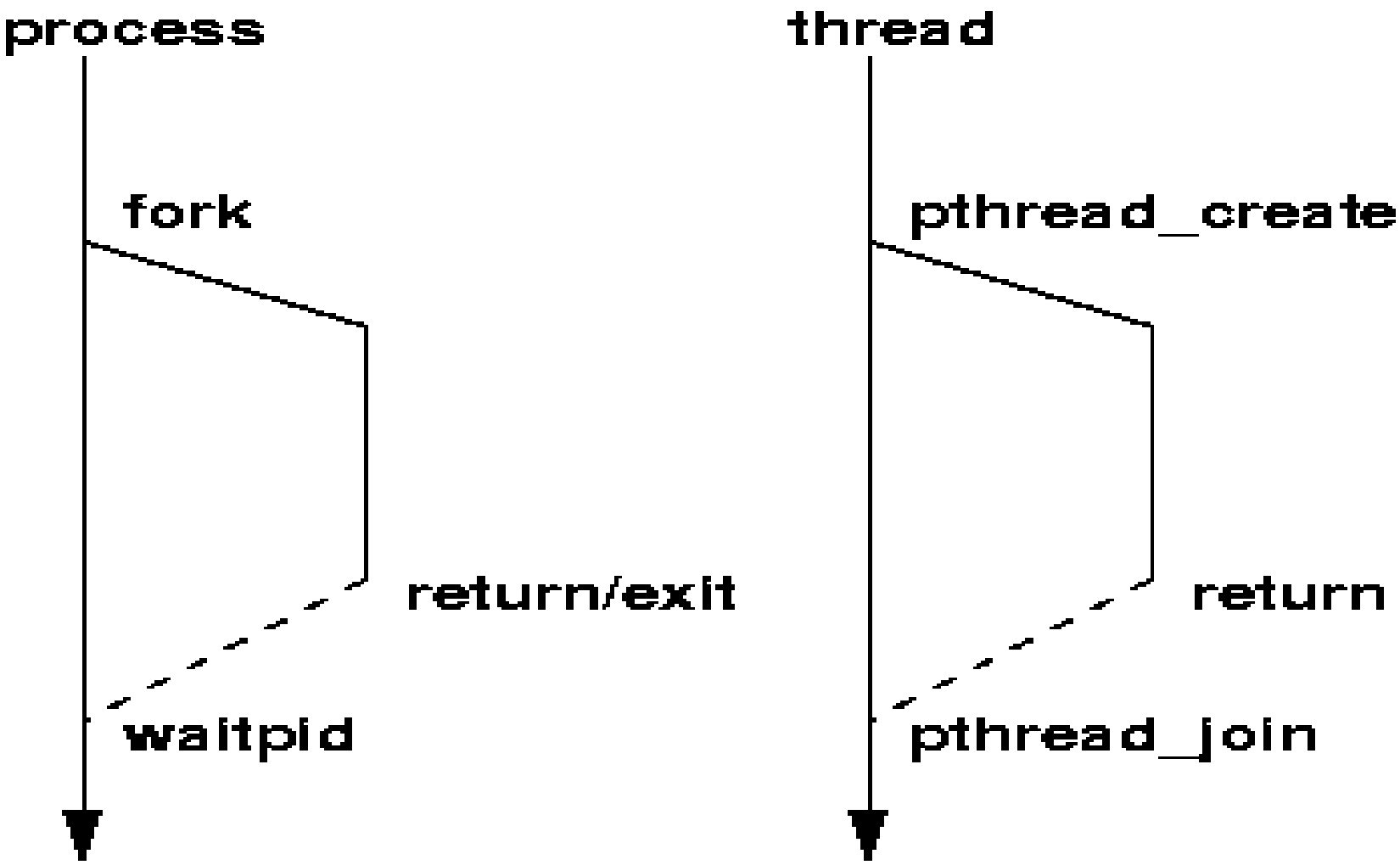


Source: https://simple.wikipedia.org/wiki/Thread_(computer_science)

Starting a Process

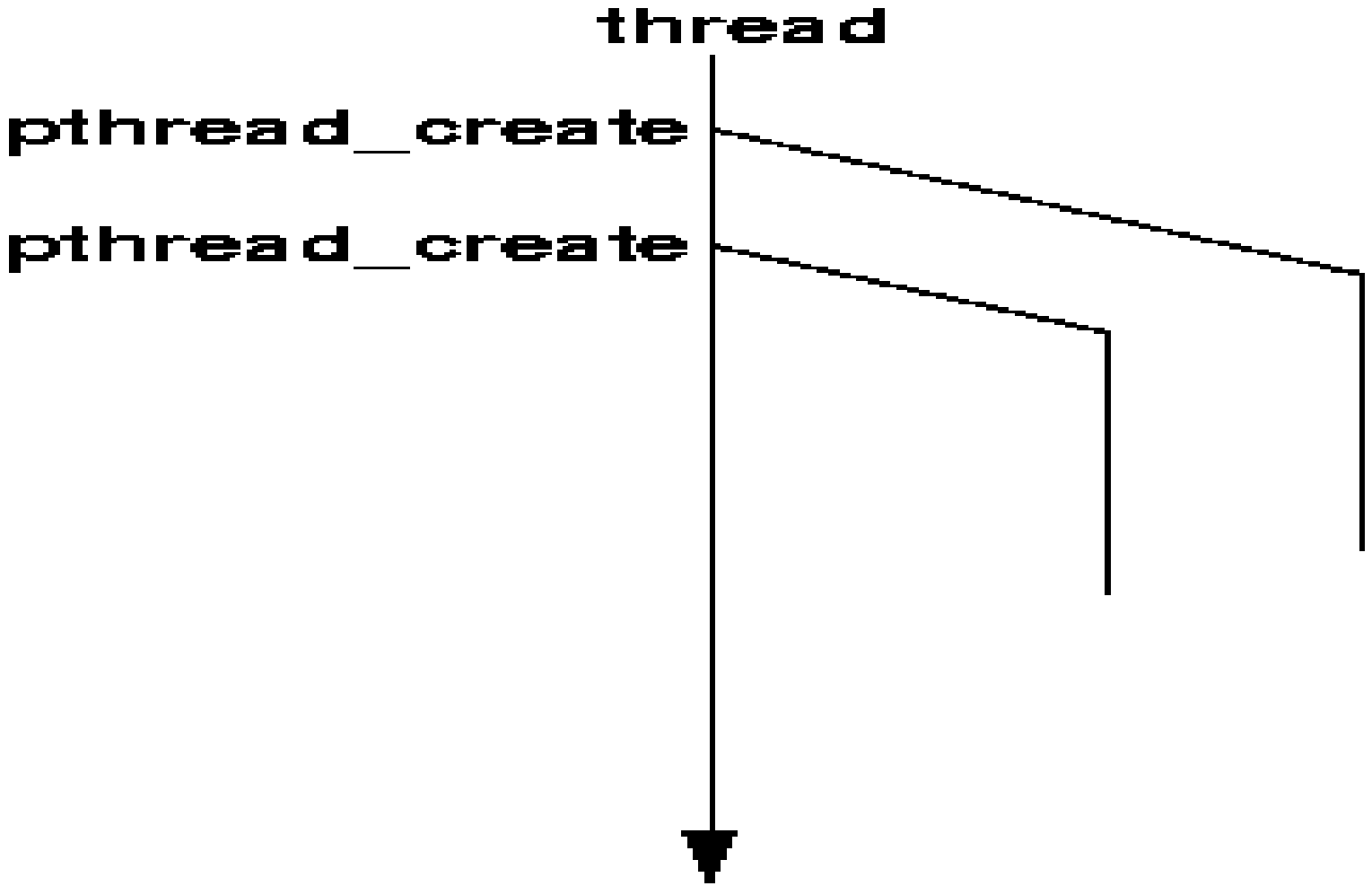
- Every time a process starts, the OS creates a primary thread.
  - The thread begins execution with the application's startup code that initializes libraries and enters main
  - The process continues until main exits and library code calls exit
- You will find demo code for starting a process here:
  - https://mwcorley79.github.io/MikeCorley/lecture19/syscall_code/fork_exec_syscall_examples.tar.gz
    - There is a system call for starting processes and thread in Linux called *Clone().*
    - "man clone" for details.
    - We won't look at Clone() specifically (in the interest of time).
      - *We will briefly cover posix threads (pthreads) and standard C++ 11 Threads (which use pthreads on Linux)*

Thread Versus Process



Source: http://www.cs.fsu.edu/~baker/opsys/notes/pthreads.html

Creating a Thread: pthread_create



Scheduling Threads: Context Switching

- Linux is a preemptive multi-tasking system. Each task is scheduled to run for some brief time period before another task is given control of a CPU core.
  - This often time sharing or time slicing
- A thread can be in one of multiple states:
  - Running
  - Blocked or suspended, using virtually no CPU cycles, but consuming some (fairly small) amount of memory
  - Ready to run, using virtually no CPU cycles

Scheduling Activities

- A running task is context switched if:
  - It is blocked waiting for some system event or resource
  - Its time slice expires and is placed back on the queue of ready to run threads
  - It is suspended by putting itself to sleep for some time, e.g., waiting on a timer
  - It is suspended by some other thread
  - It is suspended while the OS takes care of some critical activity
- Blocked threads become ready to run when an event or resource they wait on becomes available, e.g., its handle becomes signaled
- Suspended threads become ready to run when their suspend count is zero

Thread Priority: Scheduling Policy

- SCHED_OTHER (SCHED_NORMAL): Default Linux time-sharing scheduling
  - “nice” value is an attribute that can be used to influence the CPU scheduler to favor or disfavor a process in scheduling decisions. type “man nice”
- Real-time Scheduling
  - SCHED_FIFO: First in-first out scheduling
  - SCHED_FIFO can be used only with static priorities higher than 0,
  - which means that when a SCHED_FIFO thread becomes runnable, it will
  - always immediately preempt any currently running SCHED_OTHER

Scheduling Policy (cont.): Round-Robin

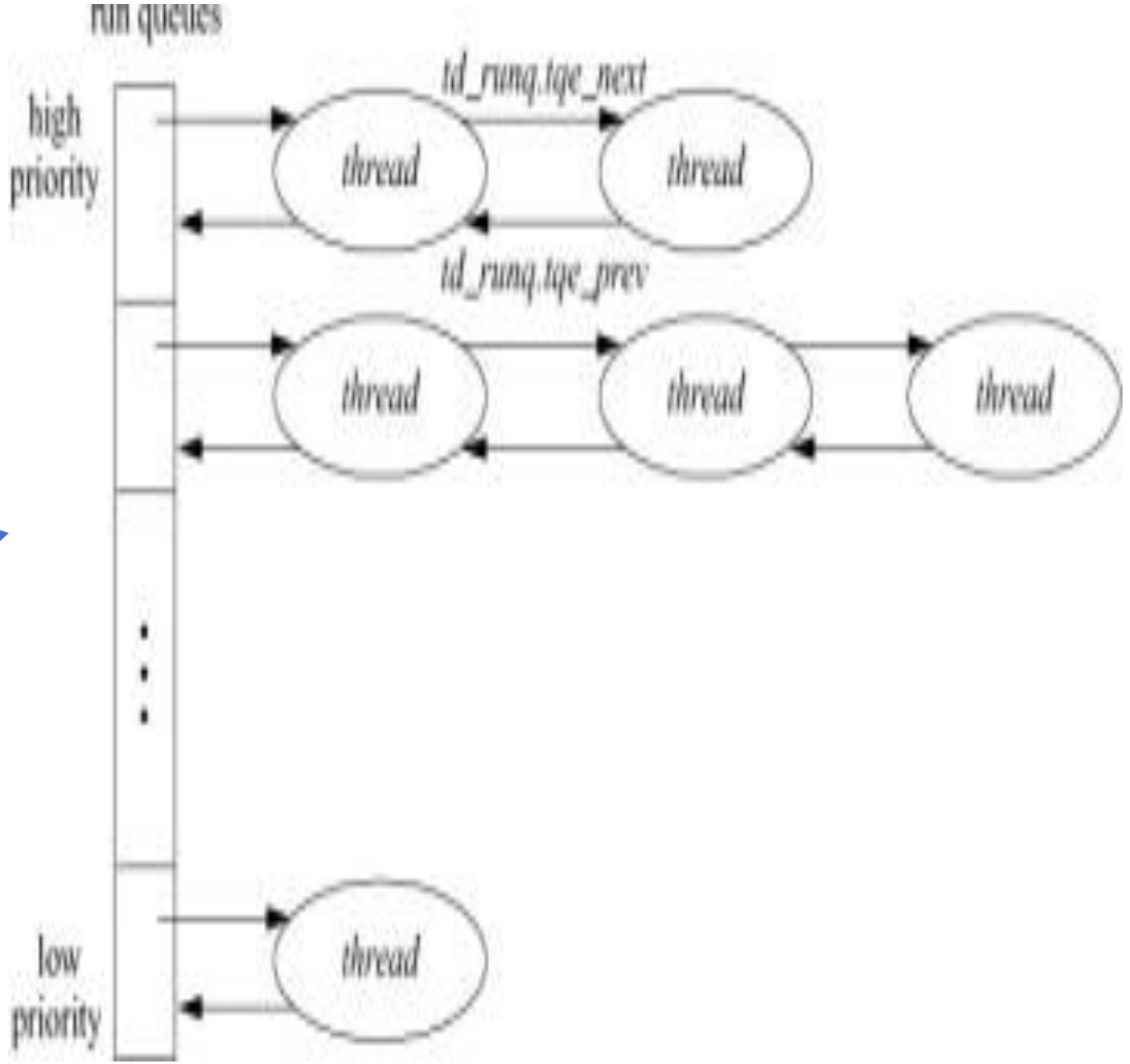
- SCHED_RR: Round-robin scheduling
- SCHED_RR is a simple enhancement of SCHED_FIFO. Everything described
- above for SCHED_FIFO also applies to SCHED_RR, except that each
- thread is allowed to run only for a maximum time quantum. If a
- SCHED_RR thread has been running for a time period equal to or longer
- than the time quantum, it will be put at the end of the list for its
- priority.

Scheduling Policy (cont.)

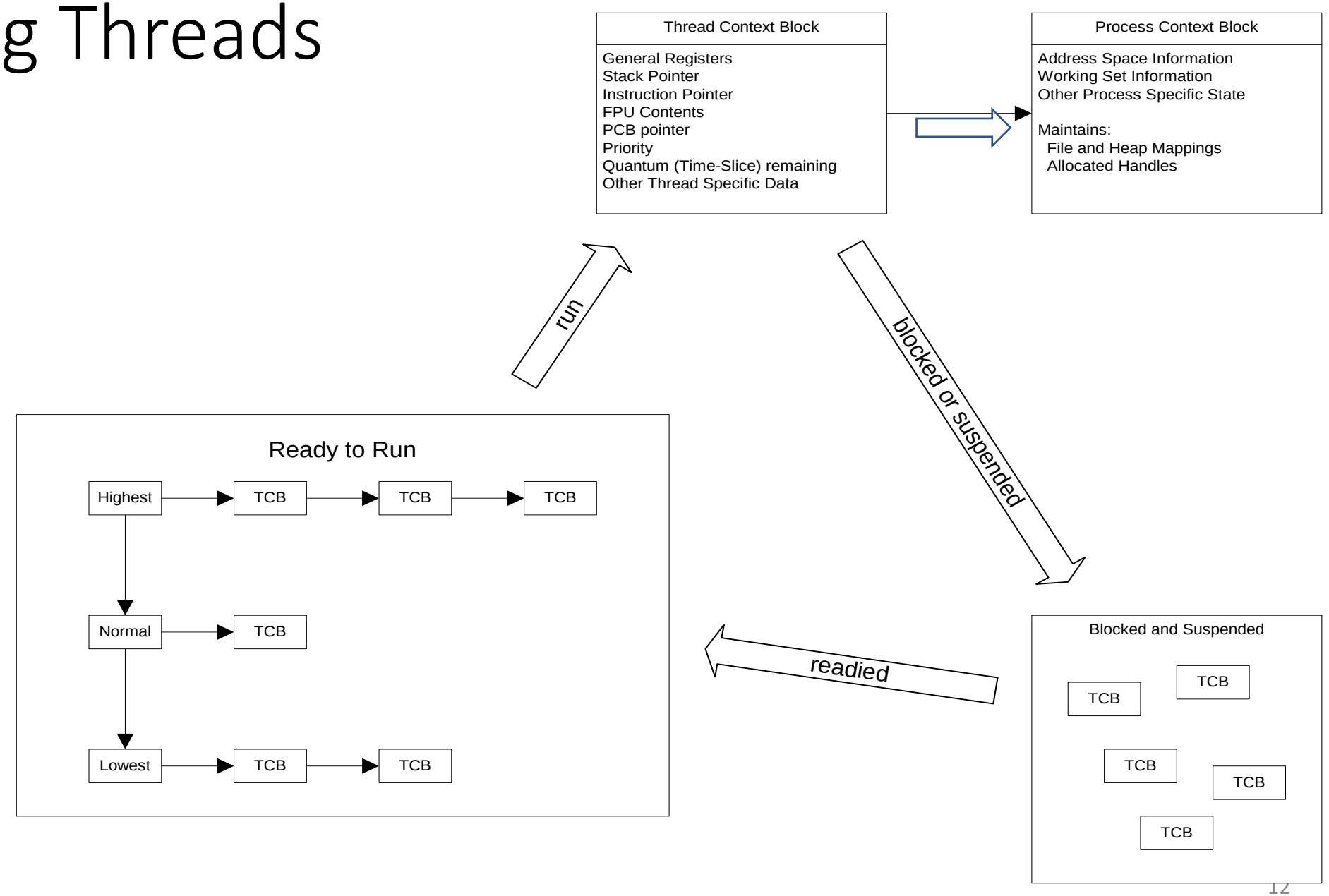
Type “man 7 sched” in Linux for Scheduling details

Thread priority management in an OS scheduler

- Priority queues



Scheduling Threads



Benefits of Using Threads

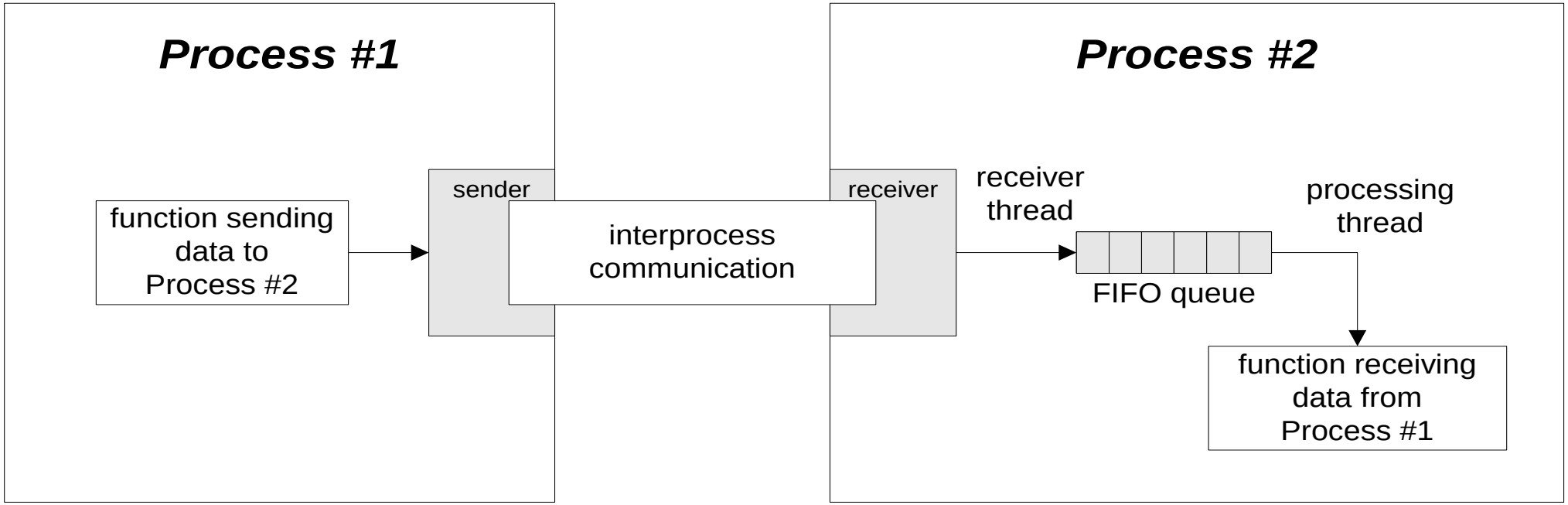
- Keep user interfaces responsive even if required processing takes a long time to complete.
  - Handle background tasks with one or more threads
  - Service the user interface with a dedicated UI thread
- Your program may need to respond to high priority events, so you can assign that event handler to a high priority thread.
- Take advantage of multiple cores available for a computation.
- Avoid low CPU activity when a thread is blocked waiting for response from a slow device or human, allowing other threads to continue.

More Benefits

- Support access to server resources by multiple concurrent clients.
- For processing with several interacting objects the program may be significantly easier to design by assigning one thread to each object.

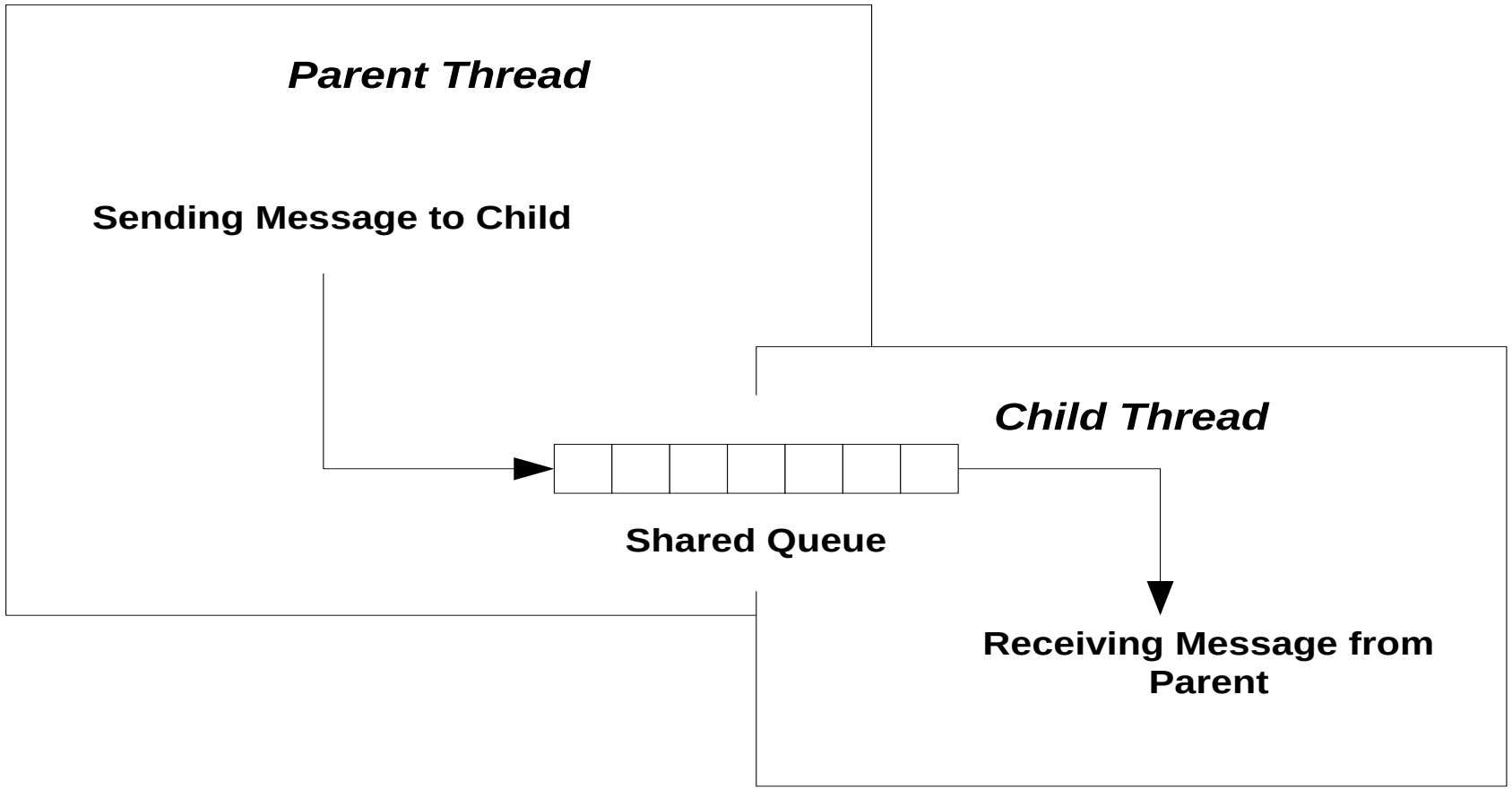
Using Threads to Avoid Blocking

Non-Blocking Communication in Asynchronous System



Shared Resources

- A child thread often needs to communicate with its parent thread. It does this via some shared resource, like a queue.



Potential Problems with Threads

- Conflicting access to shared memory
  - One thread begins an operation on shared memory, is suspended and leaves the memory region incompletely transformed.
  - A second thread is activated and accesses the shared memory in the incomplete state, causing errors in its operation and potentially errors in the operation of the suspended thread when it resumes.
- Race conditions occur when:
  - Correct operation depends on the order of completion of two or more independent activities.
  - The order of completion is not deterministic due to use of threads.

More Problems with Threads

- Starvation
  - A high priority thread dominates CPU resources, preventing lower priority threads from running often enough or at all.
- Priority Inversion
  - A low priority task holds a resource needed by a higher priority task, blocking it from running.
- Deadlock
  - Two or more tasks each own resources needed by the other preventing either one from running so neither ever completes and never releases its resources.

Synchronization

- A program may need multiple threads to share some data.
- If access is not controlled to be sequential, then shared data may become corrupted.
  - One thread accesses the data, begins to modify the data, and then is put to sleep because its time slice has expired. The problem arises when the data is in an incomplete state of modification.
  - Another thread awakes and accesses the data, that is only partially modified. The result is very likely to be corrupt data.
- The process of making access serial is called serialization or synchronization.
  - This is often accomplished using Locking privates such as Mutex

Synchronization

- Programs data shared by multiple threads is often referred to as critical sections.
  - Critical sections of code need to be guarded/protected (serialized) in order to prevent errors
  - There a number of techniques and synchronization primitives that can used for various use cases. In the interest of time, we consider one of the common: Mutex:
    - a ***mu****tual* ***ex****clusion object (mutex) a*llows multiple threads to share the same resource, by serializing access to the resource:

```
pthread_mutex_t M;
pthread_mutex_lock (&M);
... critical section ...
pthread_mutex_unlock (&M);
```

Synchronization

- a ***mu****tual* ***ex****clusion object (mutex) a*llows multiple threads to share the same resource, by serializing access to the resource:

  pthread_mutex_t M;

  pthread_mutex_lock (&M);

  ... critical section ...

  pthread_mutex_unlock (&M);

- A thread acquires (locks) the mutex causing all other threads to block, until mutex is released.
  - A thread acquires a mutex must guarantee that it will release the mutex.
    - Failure to release a mutex will result in deadlock condition

Creating Threads using pthreads

```
include <stdio.h>
#include <stdlib.h>
#include <unistd.h>  //Header file for sleep(). man 3 sleep for details.
#include <pthread.h>

// A normal C function that is executed as a thread
// when its name is specified in pthread_create()
void *myThreadFun(void *vargp)
{
    sleep(1);
    printf("Printing GeeksQuiz from Thread \n");
    return NULL;
}

int main()
{
    pthread_t thread_id;
    printf("Before Thread\n");
    pthread_create(&thread_id, NULL, myThreadFun, NULL);
    pthread_join(thread_id, NULL);
    printf("After Thread\n");
    exit(0);
}
```

Source: https://www.geeksforgeeks.org/multithreading-c-2/

Creating Threads using C++11

- `std::thread t(f)`
  - f is a callable object (e.g., function pointer, functor, or lambda that takes no arguments and has void return type)

- `std::thread t(g,a1, a2, …)`
  - g is a callable object that takes arguments a1, a2, … and has void return type

  - C++ 11 Threads are (built on top pthreads in Linux) very powerful. Callable objects mean “anything” invocable to used to start a thread.
    - Pthreads are limited to global thread functions:
      - You cannot “directly” start a thread by invoking a method of an object

Creating Threads with C++ 11 (example)

```
#include <iostream>
#include <thread>
 // This function will be called from a thread
void call_from_thread()
{
    std::cout << "Hello, World" << std::endl;
 }
int main()
{
   // create a thread
   std::thread t1(call_from_thread);
   // Join the thread with the main thread
   t1.join();
   return 0;
 }
```

C++11 Thread Functions

- t.get_id() returns thread id – unique among all running threads
- t.join() blocks until thread t completes, e.g., exits its thread function
- t.detach() disassociates thread object t from underlying OS thread.
- All theads must be joined or detached (not both) before thread object goes out of scope.
- std::this_thread::sleep_for(chrono::duration);

Examples?

https://mwcorley79.github.io/MikeCorley/lecture20/code/thread_demos.tar.gz