Operating Systems!

Processes to Threads (part 2)

Prof. Travis Peters

Montana State University

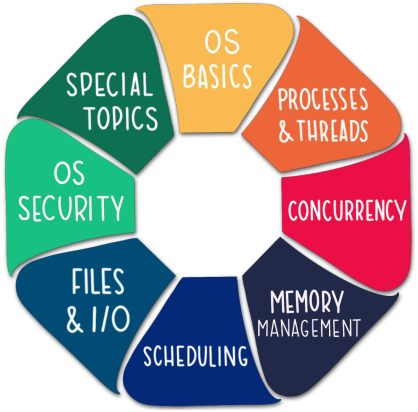
CS 460 - Operating Systems

Fall 2020

https://www.cs.montana.edu/cs460

Some diagrams and notes used in this slide deck have been adapted from Sean Smith's OS courses @ Dartmouth. Thanks, Sean!

Today



Exam Coversheet!

PA2 Starter Files!

- Announcements
  - *Heads up…. Exam 1:* ***Friday [10/02/2020] – due @ 11:59 PM (MST)***
  - *Heads up…. PA2 due* ***Sunday [10/04/2020] @ 11:59 PM (MST)***
  - *Heads up…. Yalnix! Be thinking about teams! ☺*

Talk to Travis if you need help finding team members!

- Learning Objectives
  - Understand threads
    - *more theory (create, terminate, states, etc.)*
    - *more reality (POSIX Threads – "pthreads")*
    - *common issues when working with threads*
      - ➢ *race conditions*
      - ➢ *synchronization strategies*

What To Turn In & Instructions On Submitting your Assignment

Please submit your assignment via your **private** GitHub repository.

Specifically, you should create a `pa1` directory for this assignment where you do all of your work for *this* assignment.

Your submission should contain ***at least*** the following items:

1. The **source code** file(s) you've written for your solution.

   > *Regarding what you have in your repository for the submission: **DO** commit your code and files that are needed to build/run/explain your code to the repository. **DO NOT** commit photos, executables, or other generated files. In general we try not to store generated files (executables, object files, test output, etc.) in a git repository… unless there is a really good reason… and usually there isn't! (We also try not to store large binary assets, such as images or PDFs. We may ask you to do this at times in this course, but these are exceptions, not norms.) A `.gitignore` file is very helpful for telling git to "ignore" specific files or files that match certain patterns. (See this Atlassian tutorial on .gitignore files for more information.)*

2. A `.gitignore` file that specifies files and file types that should ***not*** be tracked by git.
3. A `Makefile` for compiling and testing your program.
   - The `Makefile` ***must*** be written such that we can run `make` (with no arguments) to generate your program.
   - The program your `Makefile` builds ***must*** be called `album`.
   - If the compilation of your program fails (i.e., no executable is built) upon running `make`, ***we will automatically deduct 10 points***.
   - If the program "segfaults" when given an expected configuration of input parameters, ***we will automatically deduct 5 points***.
   - If the compilation has warnings, ***we will automatically deduct 5 points***.

     > *These are serious penalties for issues that can be easily avoided with basic testing!*

4. A `README.md` file (written in Markdown), which provides a summary of the program and tells us anything that isn't obvious from the other files.
5. A `TESTING.md` file (written in Markdown), which provides a summary of how you validated the correctness of your solution. For example, you should document basic configurations (e.g., no photos provided, unsupported file types), etc., and (generally) how your program handles different inputs.

   > *A PHONY Makefile target, or bash script, that invokes your program with various inputs is recommended.*

6. A `lifeline.pdf` "**lifeline diagram**" showing the basic lifeline of the processes involved, and their coordination, for one photo cycle.

   > *I know, I know. I said above not to commit PDFs (in general). I want one here though… 😅 This can be hand-drawn. This can be made with a tool that helps to make diagrams. Whatever tool you prefer is fine. All that matters is that the diagram is (1) a PDF, (2) that the file is named `lifeline.pdf`, and (3) that it depicts the basic lifeline of the processes involved, and their coordination, for one photo cycle.*

7. **Demos.** Submit a link to a video of you demonstrating your solution to this assignment. ***This video should be at most 7 minutes long.***
   - In your demo, *also* include a **GDB demo.** Specifically, part of your demo should include a portion dedicated to you demonstrating how to use `gdb` to answer the question: *"Assuming you have a `photos/` directory in the current directory, when you type `album photos/*.jpg` to the shell, what arguments does the shell give album?"* ***This part of the video should be no more than 2 minutes long.***
     - ***hint:** you might find the `--args` option for gdb helpful when making your gdb demo.*
   - Videos can be recorded and shared using TechSmith, for example. Make sure the video permissions are set to be viewable by anyone with the link. If we cannot view the link when we go to grade your submission you will automatically receive a zero for the relevant demo part(s) of your grade.

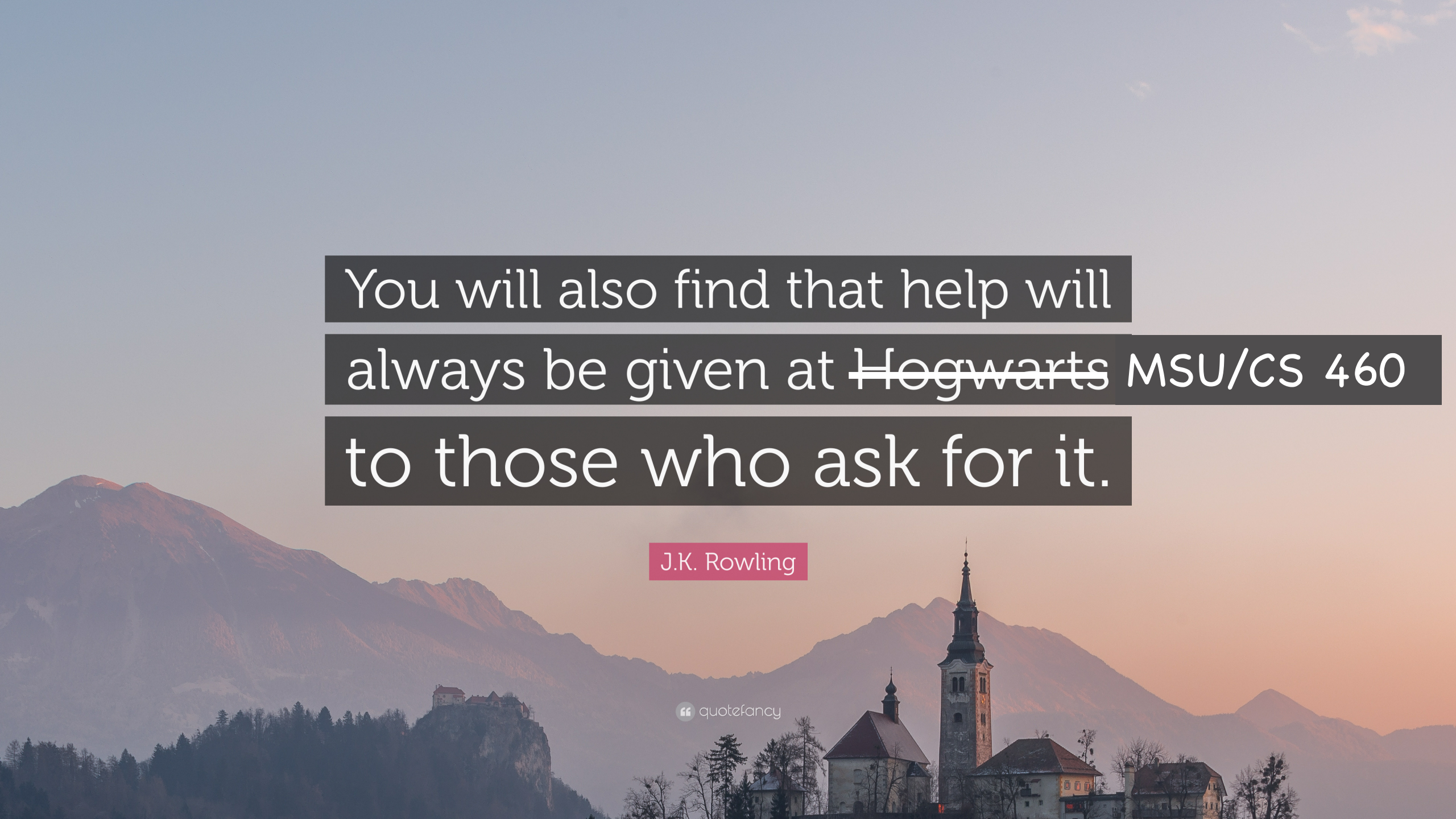
     > *A few recommendations regarding your demos:*
     > - *The rubric calls out specific aspects of your submission, such as files that need to be present, features you need to implement - stuff like that.*
     > - *When creating your demo video(s), **use the rubric as a guide/outline!** In other words, **make sure you highlight the rubric items in your demo video!** Anything you can do to make it clear that you've addressed each specific item in the rubric makes it that much easier for us to give you all the points you've earned.*

IMPORTANT!

ASSIGNMENTS: FOLLOW DIRECTIONS EXACTLY!

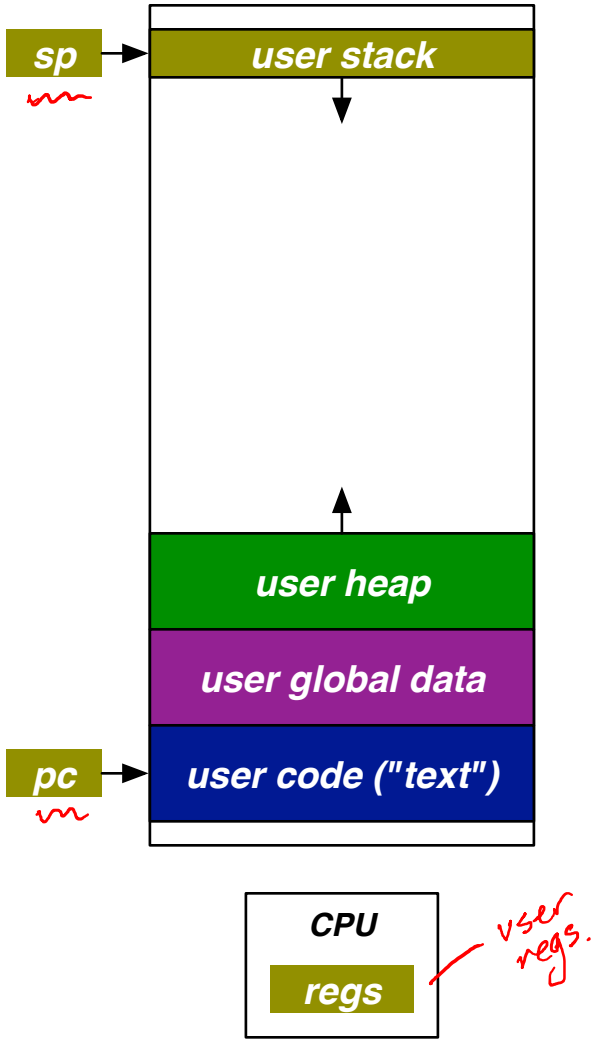
- ✓ missing files?
- ✓ incorrectly named files?
- ✓ ***no executable built?!*** *(wrong name?)*
- ✓ no Makefile / bad Makefile?
- ✓ errors when building?
- ✓ errors with basic inputs?
- ✓ missing basic components of expected output?
- ✓ "junk" in your submission?
- ✓ ….

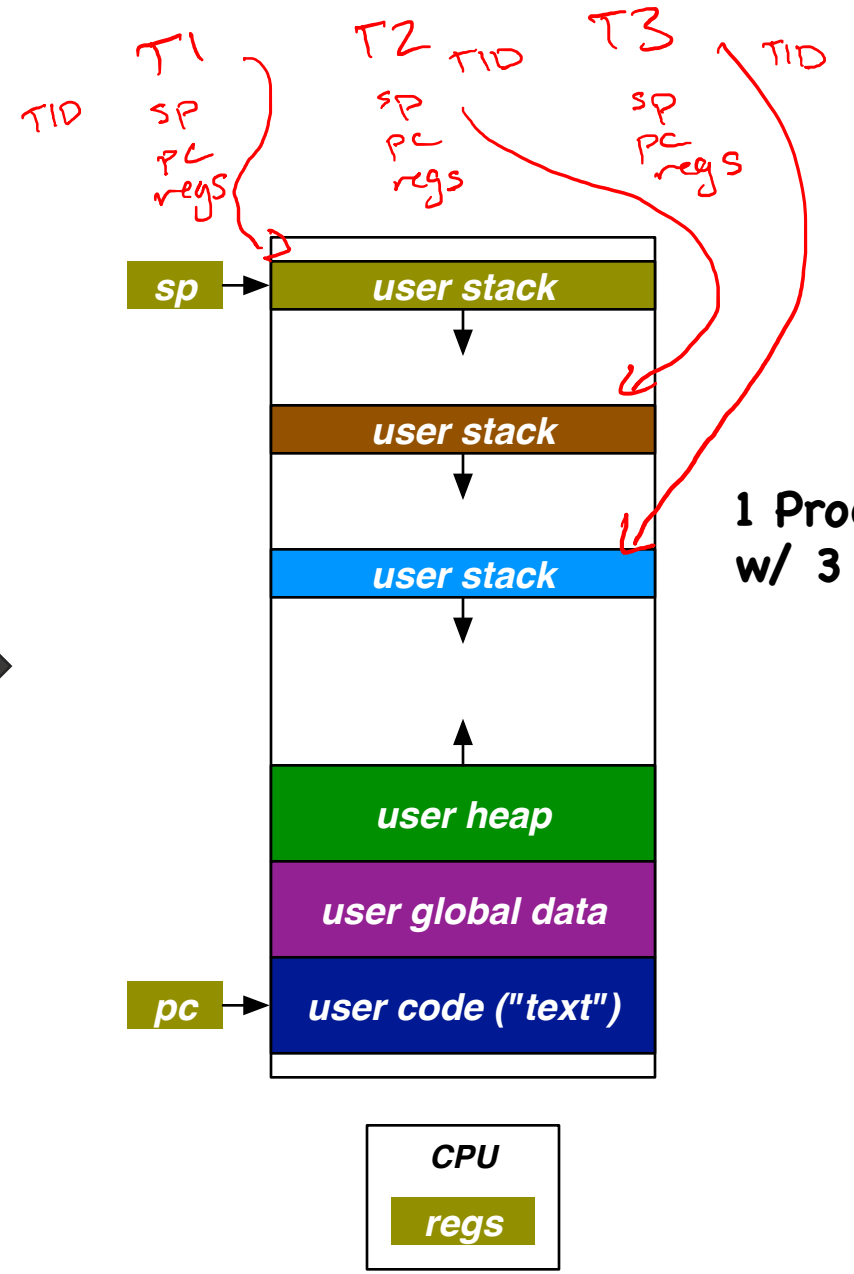
Please: if you aren't comfortable with a concept or tool, come talk to me! ☺


You will also find that help will
always be given at ~~Hogwarts~~ MSU/CS 460
to those who ask for it.
J.K. Rowling
quotefancy

Threads! *(re-orienting)*

1 Process
w/ 1 Thread





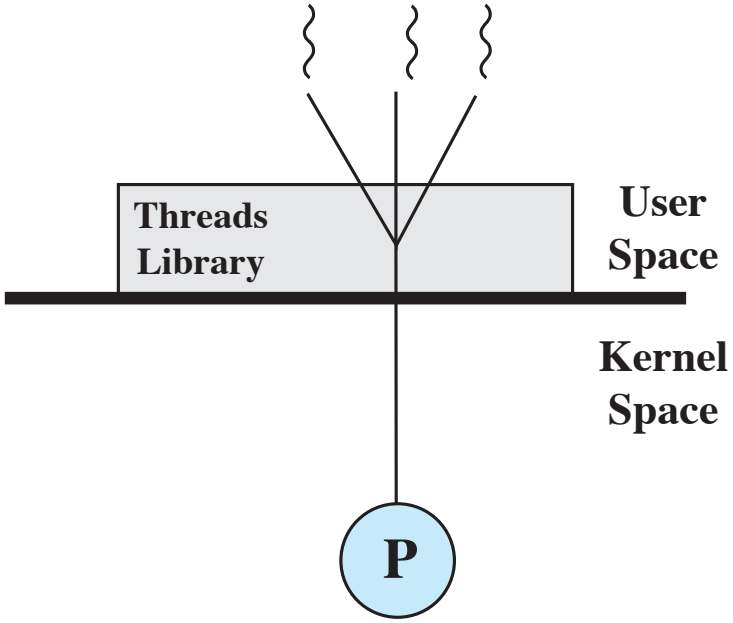
1 Process
w/ 3 Threads!

Threads Below The Surface

How exactly should a system implement threads?

How Do We Implement Threads?

There are a few ways to think about this...



(a) Pure user-level

Another View...

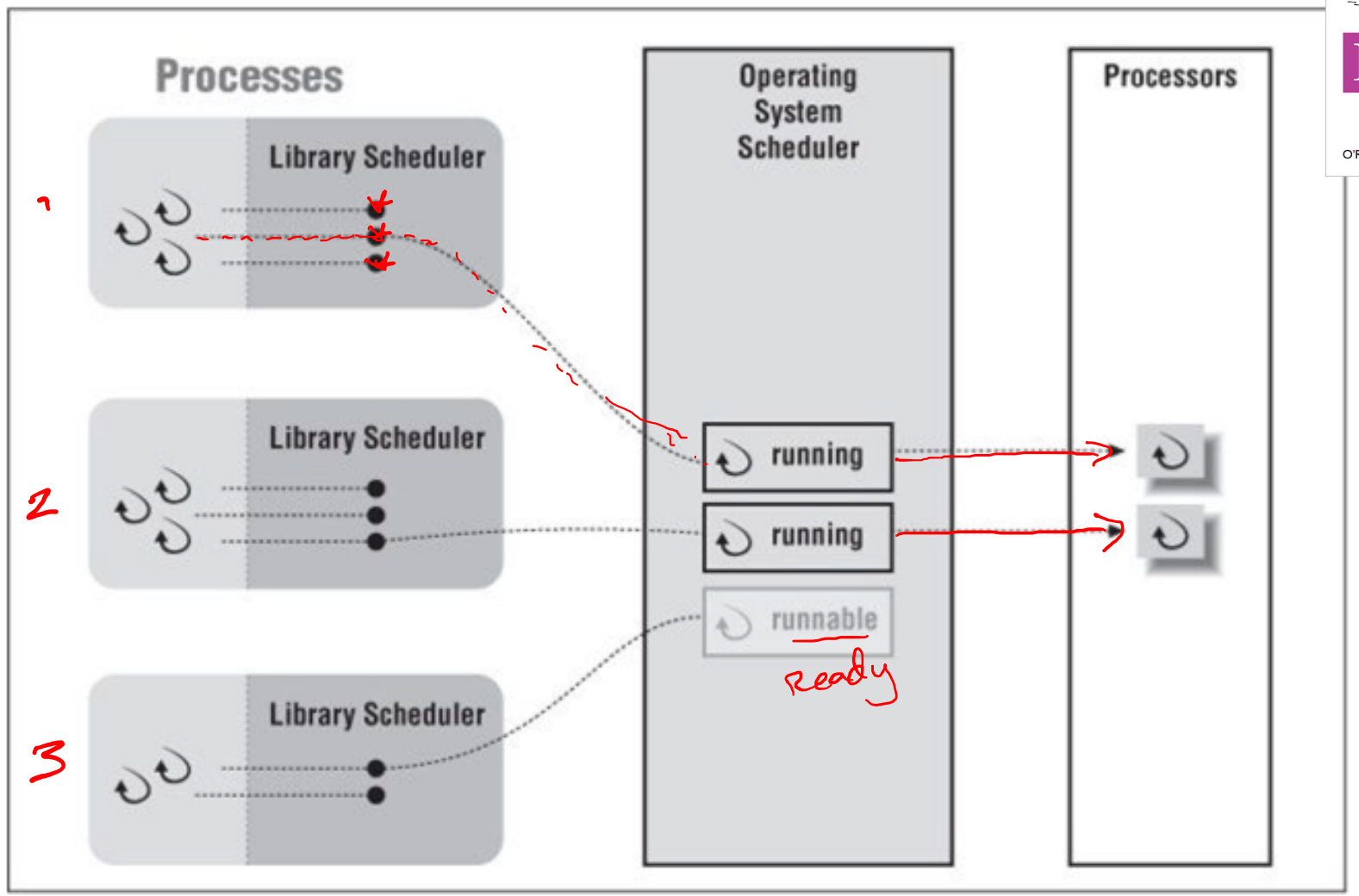
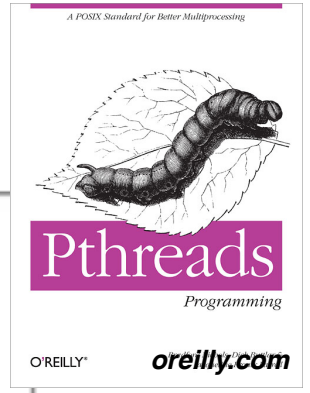
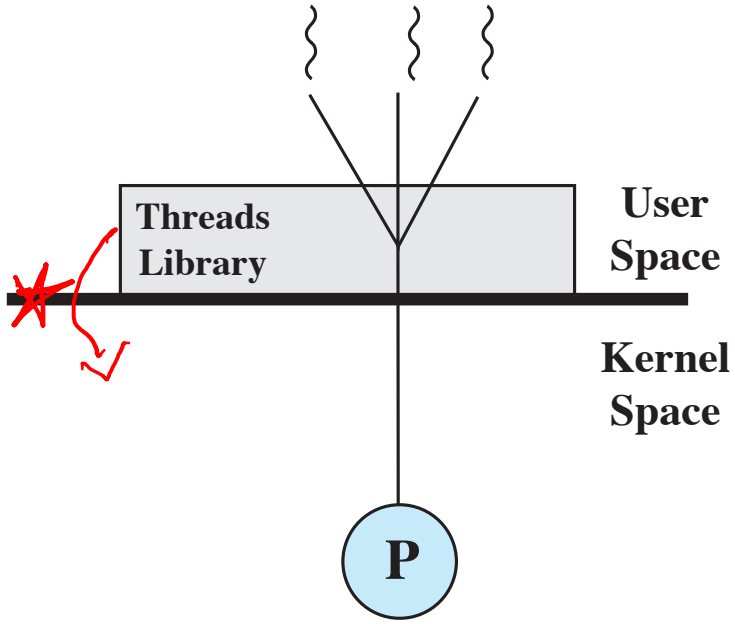


Figure 6-1. User-space thread implementations



How Do We Implement Threads?

There are a few ways to think about this...



(a) Pure user-level

Advantages ☺
- ✓ Doesn't require mode switch
- ✓ Scheduling can be app-specific
- ✓ Portable (no special support from OS needed!)

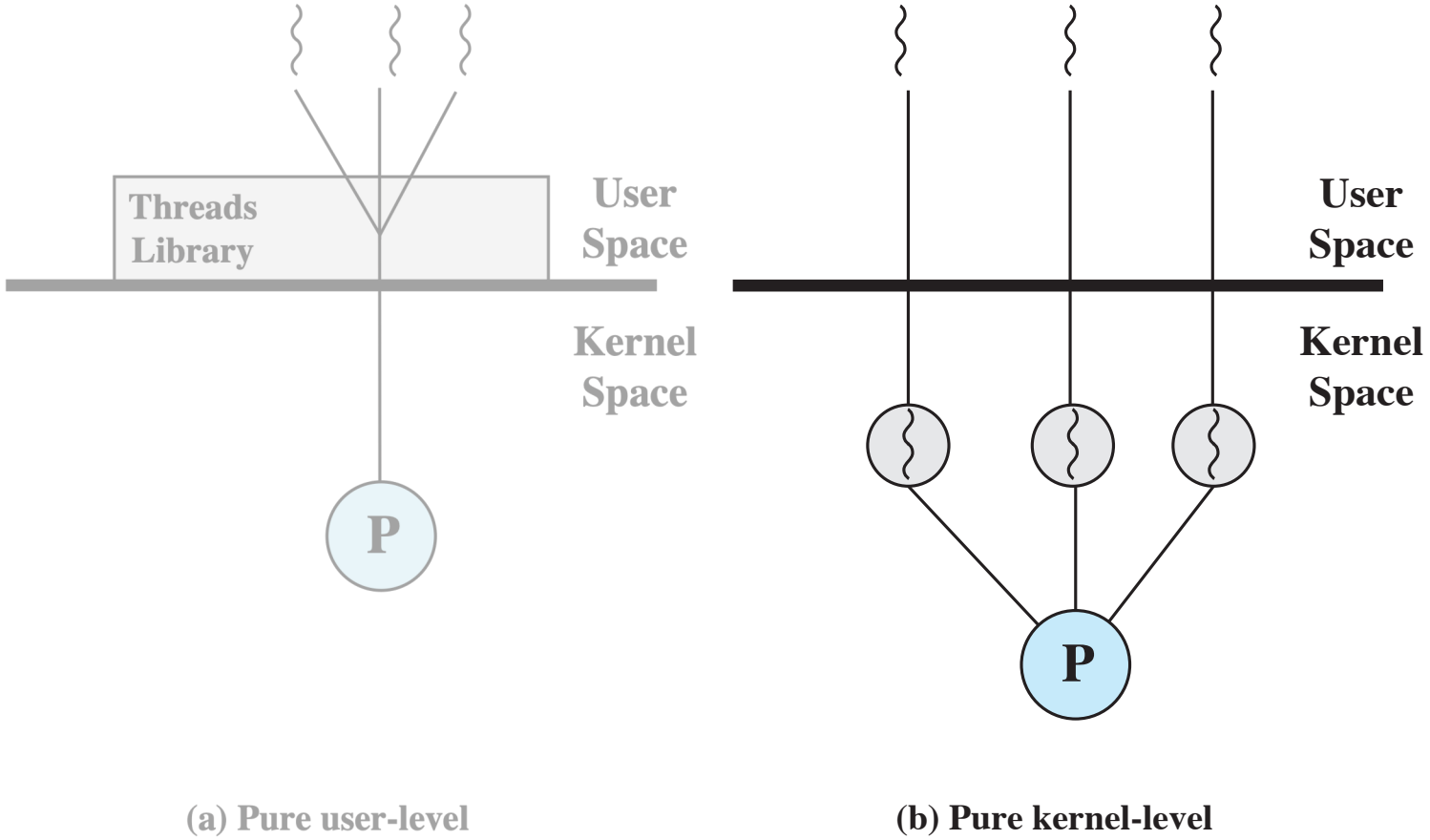
POSIX

Disadvantages ☹
- X Many syscalls are "blocking"
  (if one thread causes process to block, *all threads are blocked*)
- X Can't exploit multiprocessing ☹
  (the *process* only gets 1 processor)

User-level thread Kernel-level thread P Process

How Do We Implement Threads?

There are a few ways to think about this...



(a) Pure user-level

(b) Pure kernel-level

Another View...

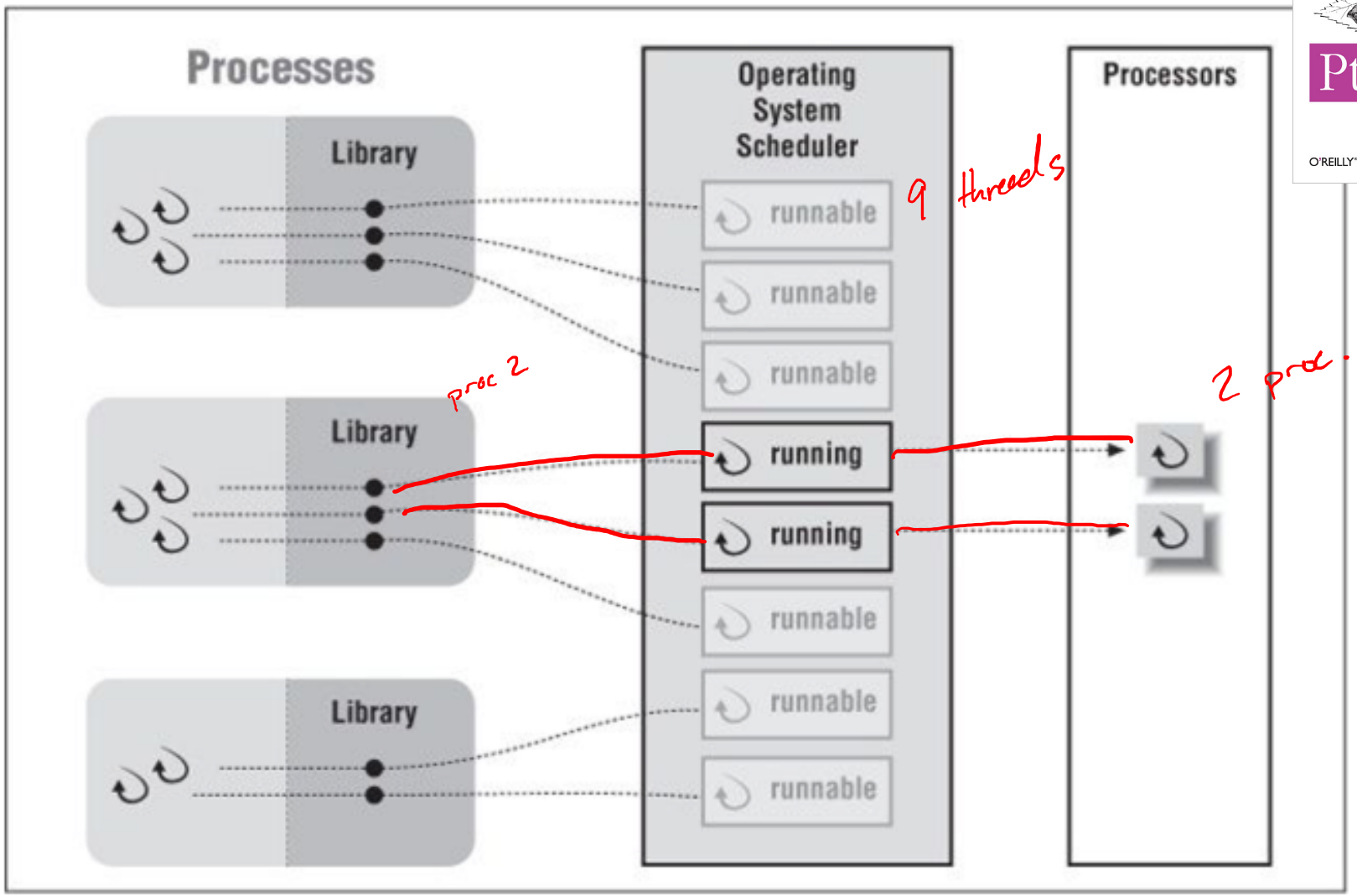
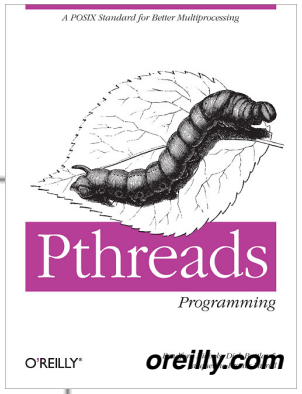


Figure 6-2. Kernel thread–based implementations



Another View...

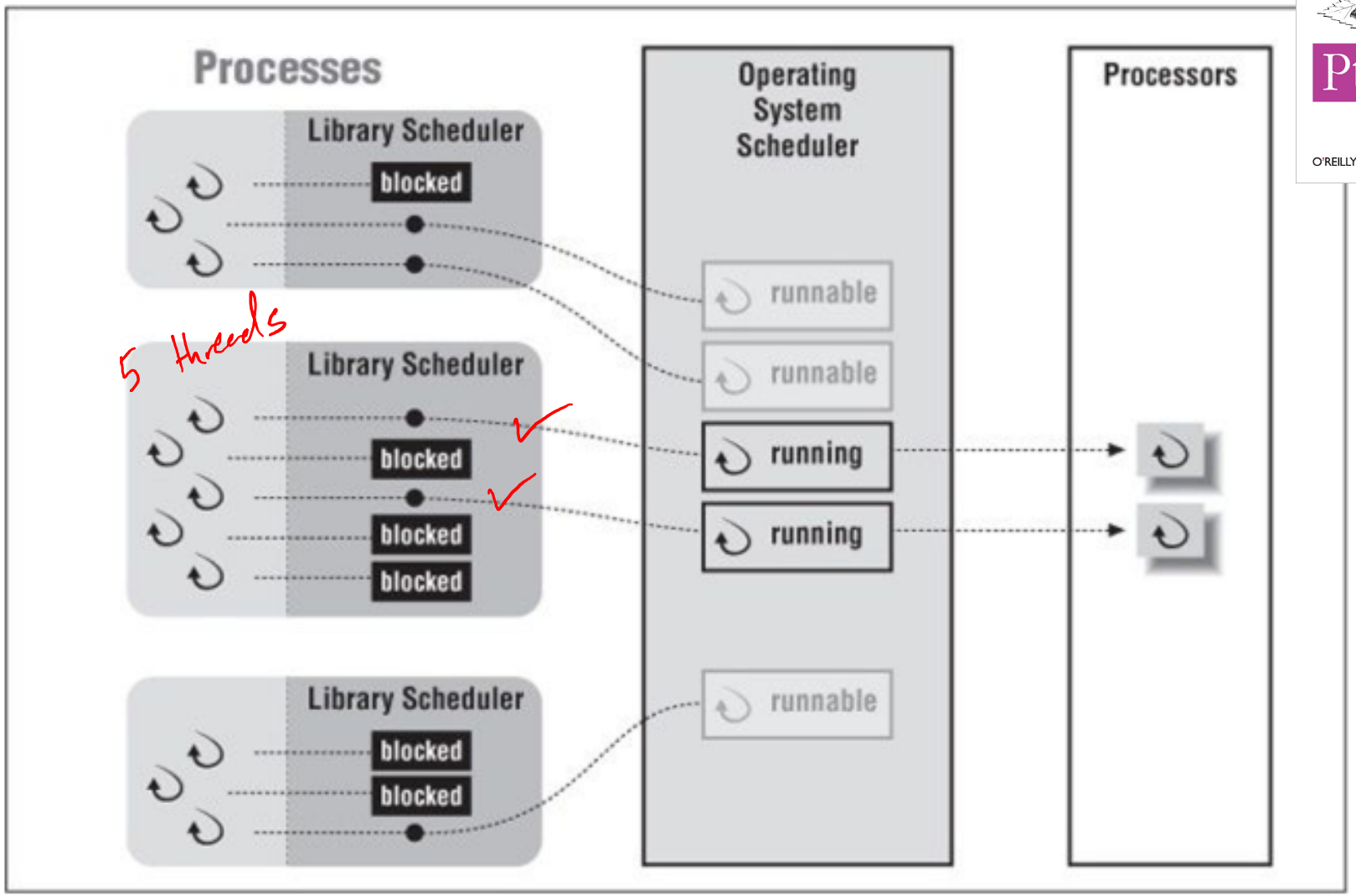
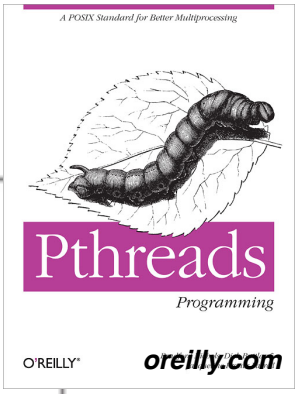
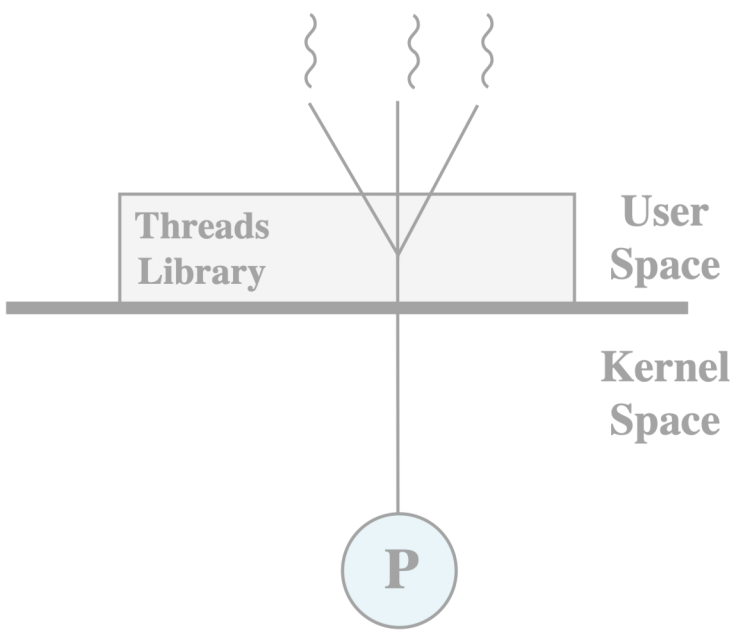


Figure 6-3. Two-level scheduler implementations

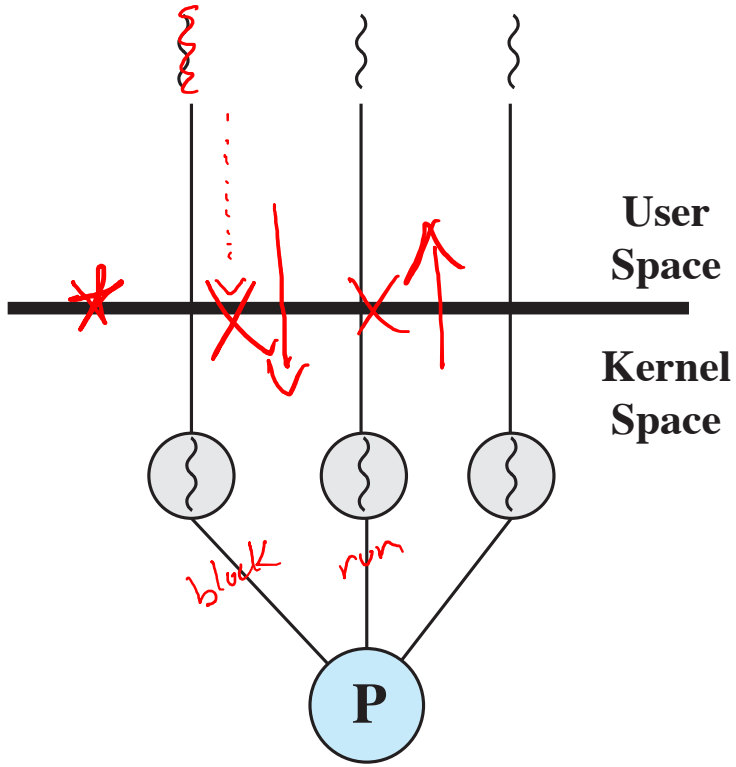


How Do We Implement Threads?

There are a few ways to think about this...



(a) Pure user-level



(b) Pure kernel-level

Advantages ☺

- ✓ Blocking syscall? *No problem!* (only that thread is blocked)
- ✓ *Can* exploit multiprocessing!

Disadvantages ☹

- ✗ Lots of mode switching

example:
-> t1 blocks (mode switch)
-> select t2 (mode switch)

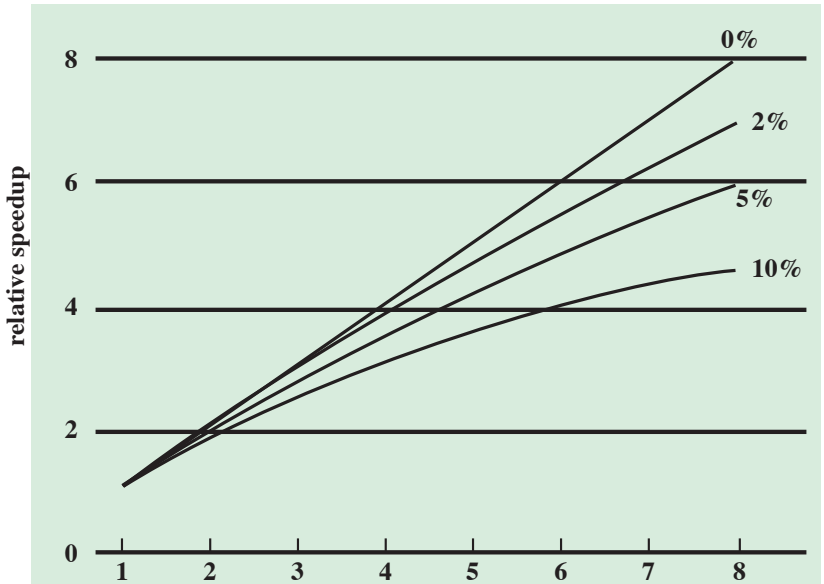
User-level thread Kernel-level thread P Process

Why Use Threads?

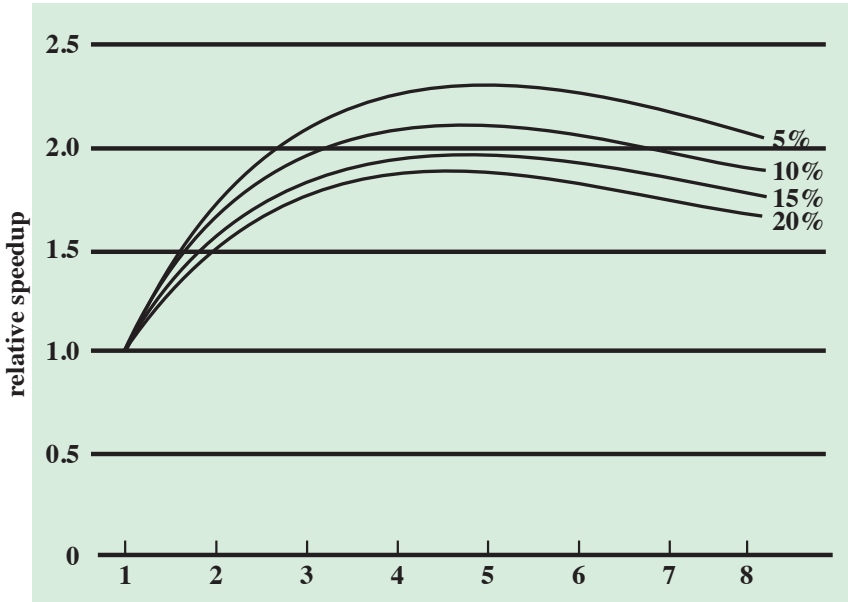
OK, but *how much faster* is it to **create/terminate/switch/communicate... threads?**

Table 4.1 Thread and Process Operation Latencies (μs)

Operation	User-Level Threads	Kernel-Level Threads	Processes
Null Fork	34	948	11,300
Signal Wait	37	441	1,840



(a) Speedup with 0%, 2%, 5%, and 10% sequential portions

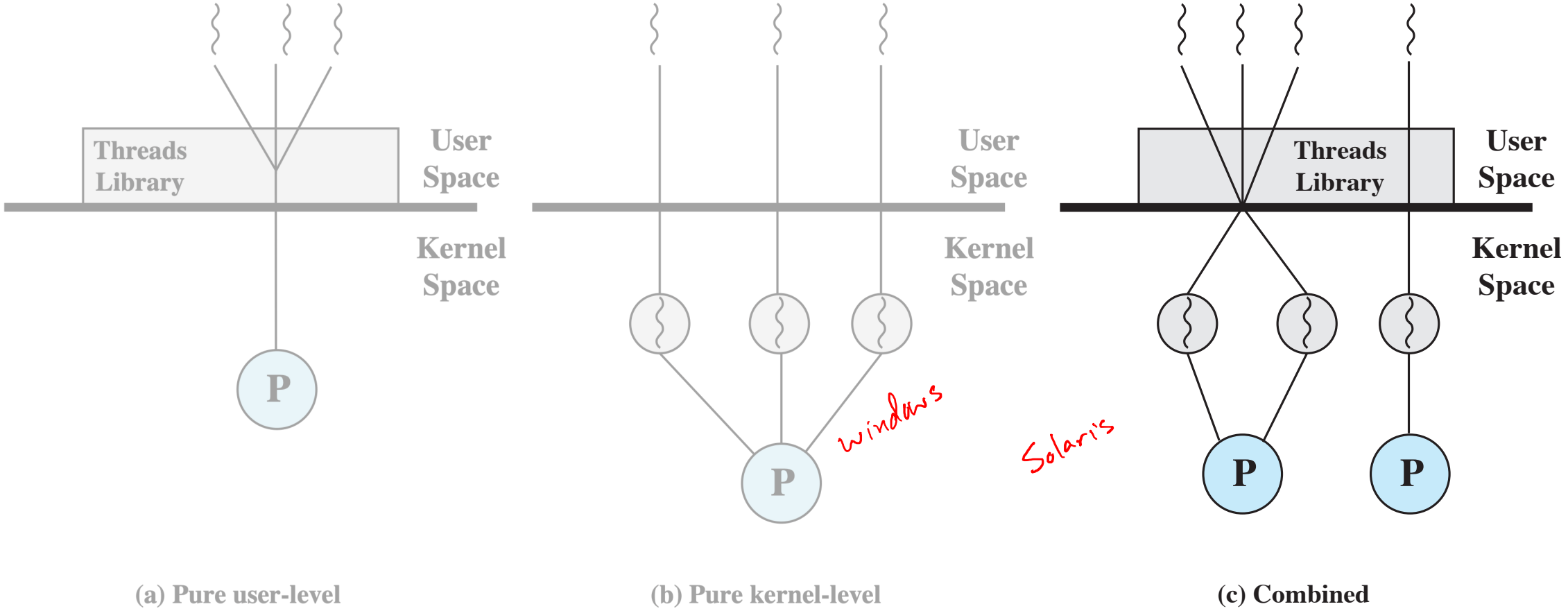


(b) Speedup with overheads

How Do We Implement Threads?

There are a few ways to think about this...

Advantages ☺
✓ Best of ULTs and KLTs!

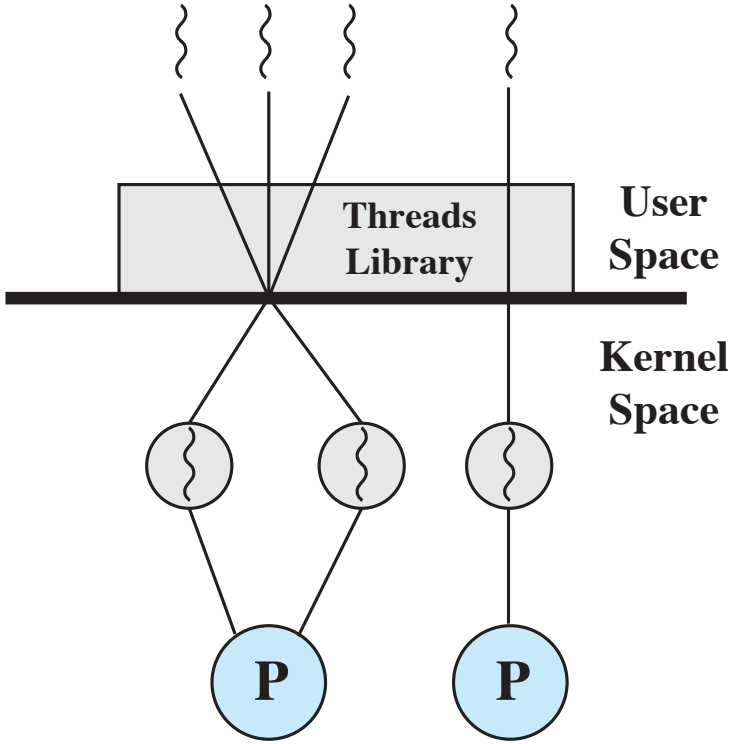


How Do We Implement Threads?

There are a few ways to think about this...

Some people use "**lightweight process**" to refer to particular special variations of threads. My advice:

- be careful when you use the term
- when you hear the term, realize the speaker may be intending to refer to some particular type of beast between "process" and "thread."



(c) Combined

POSIX = Portable Operating System Interface

POSIX Threads (“pthreads”)

Using Threads Above The Surface

- We'll use **pthreads** for thread experimentation

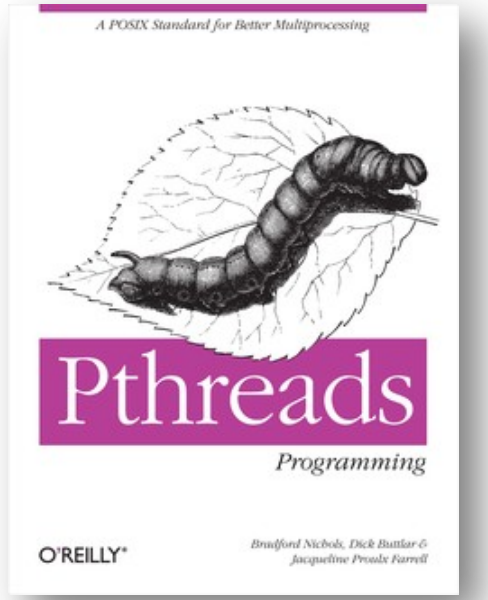
```c
#include <pthread.h>

int main()
{
    // the pthread "type"!
    pthread_t mythread;

    // code using pthreads!
    pthread_create(...);
    pthread_join(...);
    pthread_cancel(...);
    pthread_exit(...);
    pthread_mutex_init(...);
    pthread_mutex_lock(...);
    pthread_mutex_unlock(...);
    pthread_mutex_destroy(...);
    ...
}
```

```
# Debates over the right way to compile...
$ gcc -pthread ...
# vs.
$ gcc ... -lpthread
```

See:
- https://computing.llnl.gov/tutorials/pthreads/#Compiling
- https://stackoverflow.com/a/62561519



*see links from "**reading**" in week 06!*

Spent time at end of class starting to look at demos!

thread-related demos:
- *threads0.c*
- *threads1.c*
- *threads2.c*
- *thread_spy.c*
- *thread_atm.c*

Extras

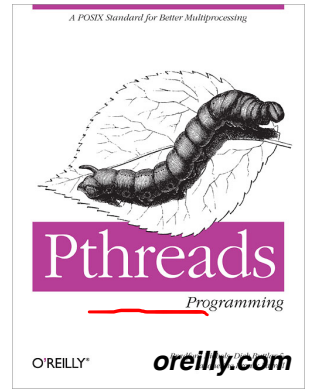
Puzzles...

- **Q:** Why didn't we talk about **IPC** between threads?
- **Q:** Could we have used pipes between threads in the same process? (See **thread_spy.c**)
- **Q:** What do you think happens if one thread in a process issues a blocking system call?
- **Q:** What do you think happens if one thread in a process calls **exec**?
- **Q:** What do you think happens if one thread in a process calls **fork**?
- **Q:** Are pthreads *userlevel* threads or *kernellevel* threads?
- **Q:** What happens if two threads want to use the same variable?
- **Q:** What happens if two threads want to use the same library? (e.g., **asctime.c**)

again, see demos:

- *threads0.c*
- *threads1.c*
- *threads2.c*
- *thread_spy.c*
- *thread_atm.c*

Some implementation details…



Fortunately, the Pthreads standard requires that vendors implement many blocking POSIX.1 calls so that they suspend only the calling thread and not the entire process. (The nonblocking behavior of the I/O calls remains the same.) They include:

fcntl	*open*
pause	*read*
sigsuspend	*sleep*
wait	*waitpid*
write	

Process Exit and Threads

Regardless of whether or not a process contains multiple threads, it can be terminated when:

- Any thread in it makes an *exit* system call.
- The thread running the *main* routine completes its execution.
- A fatal signal is delivered.

When a process exits, all threads in it die immediately, and their resources are released. (If you call *_exit* directly, the system doesn't guarantee the cleanup.)