CS 134: Operating Systems

Processes

2013-05-17

CS34

CS 134:
Operating Systems
Processes

Overview

Overview
Processes
Processes in Unix
Implementation
States
Threads
Concepts
Uses
Models
Design

Processes & Concurrency

What *is* a process?

What *is* concurrency?

Processes & Concurrency
What *is* a process?
What *is* concurrency?

User’s View of Processes

A fundamental OS abstraction

- But details vary from OS to OS:
  - Batch system—Jobs
  - Time-shared systems—User programs or tasks
- Common idea: Process = “A program in execution”
- Processes have a degree of independence from each other
  - Possibly only allowed communicate through designated mechanisms
  - One errant processes should not affect other unrelated ones

User’s View of Processes
A fundamental OS abstraction
- But details vary from OS to OS:
  - Batch system—Jobs
  - Time-shared systems—User programs or tasks
- Common idea: Process = “A program in execution”
- Processes have a degree of independence from each other
  - Possibly only allowed communicate through designated mechanisms
  - One errant processes should not affect other unrelated ones

Class Exercise

What makes up a process? (“A process has. . . ”)

- In general
- On a typical POSIX system

Class Exercise
What makes up a process? (“A process has. . . ”)
- In general
- On a typical POSIX system

Components of a Process (Unix)

- Execution state
  - Registers
  - Program counter
  - Program status word
  - Stack pointer
- Scheduling information
  - Process state
  - Priority
  - Class, etc.
- Memory
  - Text area
  - Data area
  - Stack area
- Security/Authentication Info
  - User ID
  - Group ID
- I/O State
  - File descriptors
  - Working directory
  - Root directory
- Event Notifications
  - Signals waiting
  - Signal mask
  - Time of next alarm
- Other
  - Process ID
  - Parent process
  - Process group
  - Controlling terminal
  - Start time
  - CPU time
  - Children's CPU time

Components of a Process (Unix)
Execution state Registers Program counter Program status word Stack pointer Scheduling information Process state Priority Class, etc. Memory Text area Data area Stack area Security/Authentication Info User ID Group ID I/O State File descriptors Working directory Root directory Event Notifications Signals waiting Signal mask Time of next alarm Other Process ID Parent process Process group Controlling terminal Start time CPU time Children's CPU time

Processes under UNIX

Processes:

- Create with `fork`
- Exit with `exit`
- Replace “process image” with `execve`

Multiple processes may be active at any one time (compare w/ uniprogrammed system)

Processes under UNIX

Processes:

- Create with fork
- Exit with exit
- Replace “process image” with execve

Multiple processes may be active at any one time (compare w/ uniprogrammed system)

Class Question

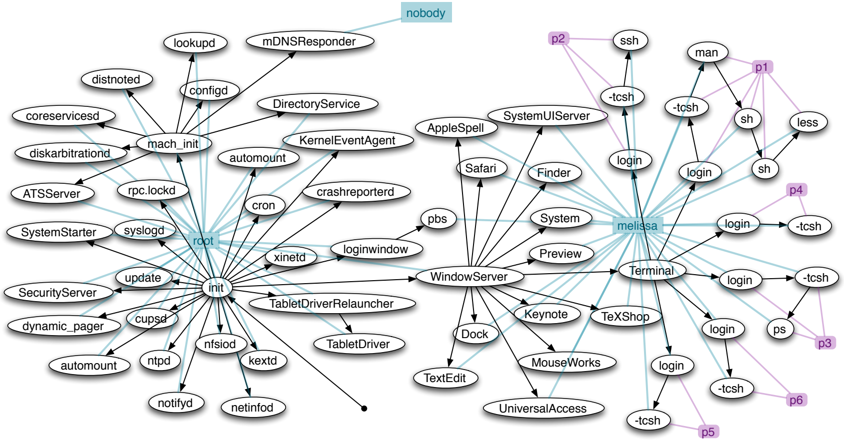
If there's `fork`, should we have `join`?

Class Question

If there's `fork`, should we have `join`?

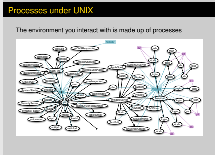
Processes under UNIX

The environment you interact with is made up of processes



2013-05-17

CS34
└Processes
  └Processes in Unix
    └Processes under UNIX



Process Implementation

How does the OS implement the process abstraction?

Process Implementation

How does the OS implement the process abstraction?

Process Implementation (cont'd)

The OS needs to maintain a *process image* for each process:

- Process's address space, containing:
  - Program code
  - Program data
  - Processor stack
- Housekeeping information (*PCB*)
  - One of most important is *process state*

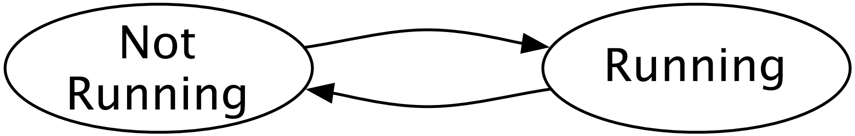
Process Implementation (cont'd)

The OS needs to maintain a *process image* for each process:

- Process's address space, containing:
  - Program code
  - Program data
  - Processor stack
- Housekeeping information (*PCB*)
  - One of most important is *process state*

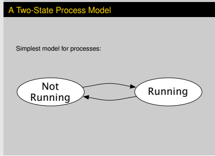
A Two-State Process Model

Simplest model for processes:



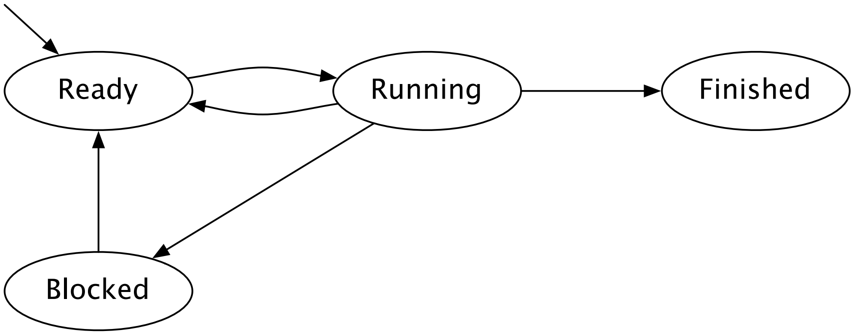
2013-05-17

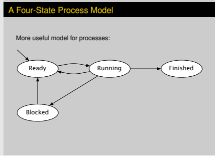
CS34
└─Processes
  └─States
    └─A Two-State Process Model



A Four-State Process Model

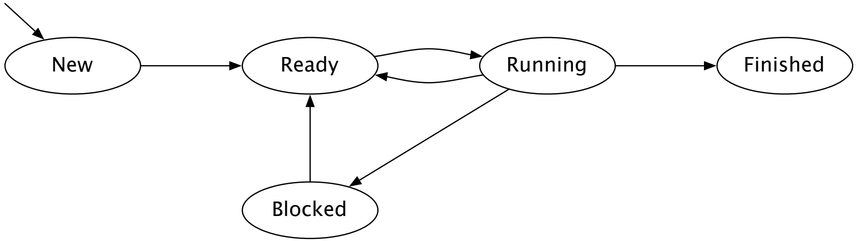
More useful model for processes:





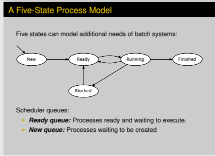
A Five-State Process Model

Five states can model additional needs of batch systems:



Scheduler queues:

- ***Ready queue:*** Processes ready and waiting to execute.
- ***New queue:*** Processes waiting to be created



Generalizing Processes

Simple view of process is

Address space

\+ Thread of execution

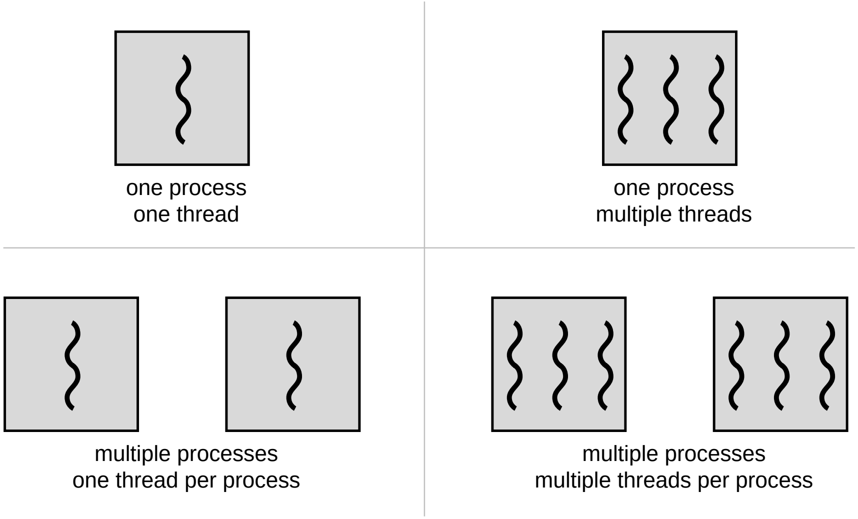
Does the mapping need to be one-to-one?

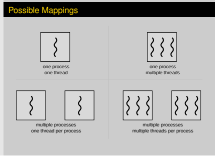
Generalizing Processes

Simple view of process is
Address space
\+ Thread of execution

Does the mapping need to be one-to-one?

Possible Mappings





Threads

Motivation:

- Traditional processes: Virtual uniprocessor machine
- Multithreaded processes: Virtual multiprocessor machine

Threads

Motivation:
- Traditional processes: Virtual uniprocessor machine
- Multithreaded processes: Virtual multiprocessor machine

Uses of Threads

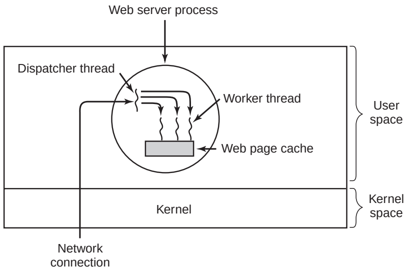
Various reasons why people use threads

- Performing foreground and background work
- Supporting asynchronous processing
- Speeding execution
- Organizing programs

Uses of Threads

Various reasons why people use threads
- Performing foreground and background work
- Supporting asynchronous processing
- Speeding execution
- Organizing programs

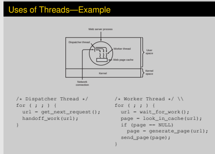
Uses of Threads—Example




```
/* Dispatcher Thread */
for ( ; ; ) {
  url = get_next_request();
  handoff_work(url);
}
```

```
/* Worker Thread */ \\
for ( ; ; ) {
  url = wait_for_work();
  page = look_in_cache(url);
  if (page == NULL)
    page = generate_page(url);
  send_page(page);
}
```


2013-05-17



Class Exercise

Can an application implement threads without built-in thread support in the OS?

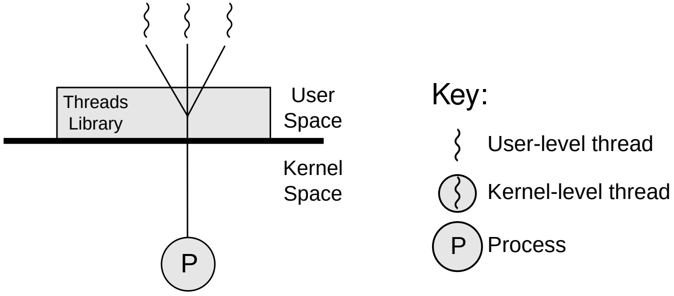
If so, what *does* it need from the from the OS to support threads?

Class Exercise

Can an application implement threads without built-in thread support in the OS?

If so, what *does* it need from the from the OS to support threads?

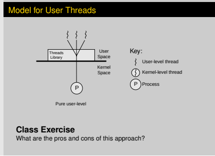
Model for User Threads



Pure user-level

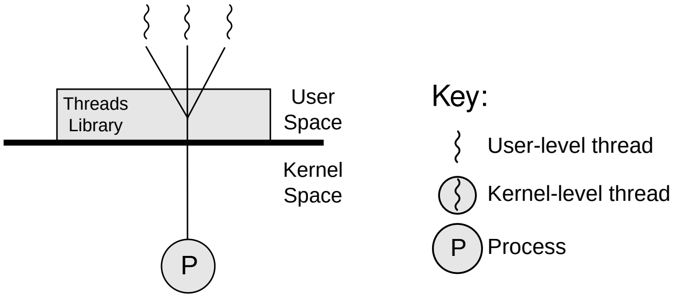
Class Exercise

What are the pros and cons of this approach?



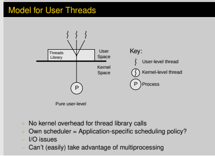
So, maybe we should put the threads in the kernel?

Model for User Threads



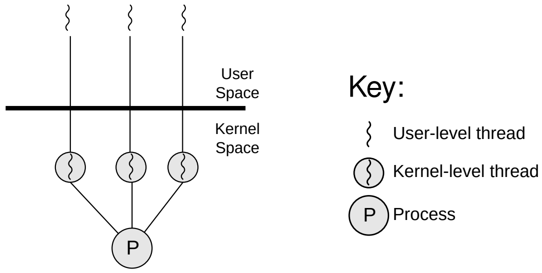
Pure user-level

- \+ No kernel overhead for thread library calls
- \+ Own scheduler = Application-specific scheduling policy?
- – I/O issues
- – Can't (easily) take advantage of multiprocessing



So, maybe we should put the threads in the kernel?

Model for Kernel-Level Threads



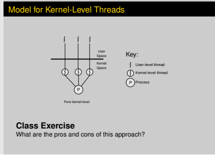
Pure kernel-level

Class Exercise

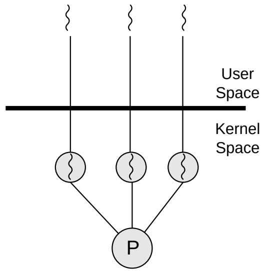
What are the pros and cons of this approach?

2013-05-17

CS34
└Threads
  └Models
    └Model for Kernel-Level Threads



Model for Kernel-Level Threads



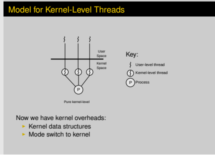
Pure kernel-level

Key:

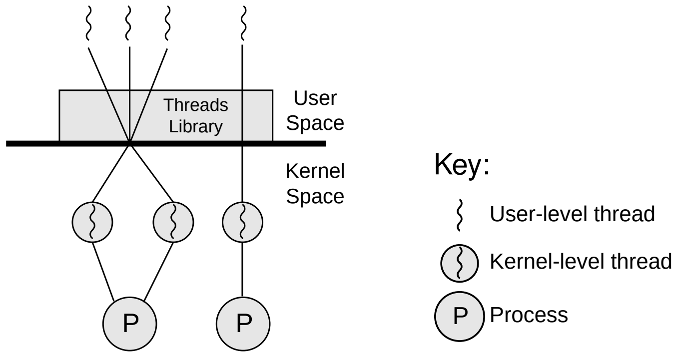
- User-level thread
- Kernel-level thread
- P Process

Now we have kernel overheads:

- Kernel data structures
- Mode switch to kernel



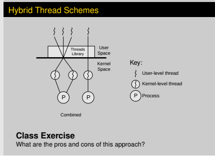
Hybrid Thread Schemes



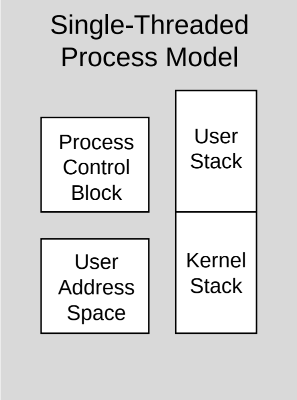
Combined

Class Exercise

What are the pros and cons of this approach?

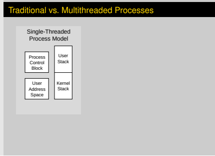


Traditional vs. Multithreaded Processes

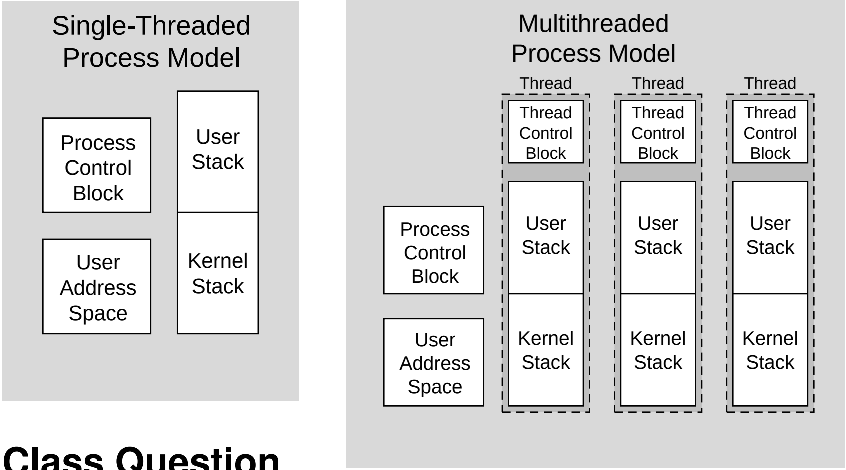


2013-05-17

CS34
└Threads
 └Models
  └Traditional vs. Multithreaded Processes



Traditional vs. Multithreaded Processes

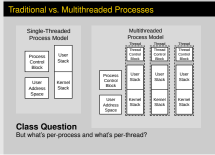


Class Question

But what's per-process and what's per-thread?

2013-05-17

CS34
└Threads
  └Models
    └Traditional vs. Multithreaded Processes



Per-Process vs. Per-Thread—*You* Decide. . .

- Execution state
  - Registers
  - Program counter
  - Program status word
  - Stack pointer
- Scheduling information
  - Process state
  - Priority
  - Class, etc.
- Memory
  - Text area
  - Data area
  - Stack area
- Security/Authentication Info
  - User ID
  - Group ID
- I/O State
  - File descriptors
  - Working directory
  - Root directory
- Event Notifications
  - Signals waiting
  - Signal mask
  - Time of next alarm
- Other
  - Process ID
  - Parent process
  - Process group
  - Controlling terminal
  - Start time
  - CPU time
  - Children's CPU time

2013-05-17

Per-Process vs. Per-Thread—You Decide. . .
Execution state: Registers, Program counter, Program status word, Stack pointer; Scheduling information: Process state, Priority, Class, etc.; Memory: Text area, Data area, Stack area; Security/Authentication Info: User ID, Group ID; I/O State: File descriptors, Working directory, Root directory; Event Notifications: Signals waiting, Signal mask, Time of next alarm; Other: Process ID, Parent process, Process group, Controlling terminal, Start time, CPU time, Children's CPU time