Today: Threads

- How should we schedule threads (or processes) onto the CPU?
- Threads packaged?
- Where should we implement threads? In the kernel? In a user-level?
- What are threads?

Processes communicate either with message passing or shared memory

- Process switch
- The program currently executing on the CPU is changed by performing a context switch
- On a uniprocessor, there is at most one running process at a time.
- A process is either New, Ready, Waiting, Running, or Terminated.
- A process is the unit of execution.
- Processes are represented as Process Control Blocks in the OS
- PCBs contain process state, scheduling, and memory management information, etc

Last Class: Processes
Operating systems can support one or many address spaces, and one or

**Classifying Threaded Systems**

- Simpler than message passing and shared-memory
- No system calls are required to cooperate among threads
- The address space of a process is shared among all its threads

Each process may have multiple threads of control within it:

- **Threads** are bound to a single process.
- **Threads** extract the thread of control information from the process

A thread defines a single sequential execution stream within a process

A process defines the address space, text, resources, etc.

**Processes Versus Threads**
Kernel Threads

Processes

Switching between kernel threads is slightly faster than switching between

It can use the same process scheduling algorithms.

The kernel must manage and schedule threads (as well as processes), but

- The kernel must manage the thread's address space.
- Memory management information does not need to be changed since the thread's
  values of registers, program counter, and stack pointer must be changed.
- Switching between kernel threads of the same process requires a small
  context switch.

Switching between kernel threads is a lightweight process, also known as a

The operating system knows about:

A kernel thread

Example Threaded Program

memory layout:

One possible

library (user code):

forking a thread can be a system call to the kernel, or a procedure call to a thread

consume item mutex

mutex = buffer[out]; out = (out + 1) mod n

while[in = out] do no-op

repeat

producer

end

fork-thread(consumer());
fork-thread(producer());
in = 0; out = 0;

repeat

buffer[in] = value;

while[in = out] do no-op

mutex = producer-item

fork-thread(consumer());
fork-thread(consumer());
in = 0; out = 0;

repeat

buffer[out] = value;

while[in = out] do no-op

mutex = consumer-lock

fork-thread(consumer());
fork-thread(consumer());
in = 0; out = 0;

repeat

buffer[0] = value;

while[in = out] do no-op

mutex = consumer-lock

fork-thread(consumer());
fork-thread(consumer());
in = 0; out = 0;

repeat

buffer[in] = value;

while[in = out] do no-op

mutex = producer-lock
Thread Ready Queue

Kernel Processes

User-Level Thread Scheduler

Current Thread for each Process

Thread Ready

Kernel Processes

User-Level Threads

- The programmer uses a thread library to manage threads (create and delete them, synchronize them, and schedule them).
- The OS only schedules the process, not the threads within the process.
- The OS only knows about the process containing the threads.
- A user-level thread is a thread that the OS does not know about.
slices the OS will dedicate to it.
For kernel threads, the more threads a process creates, the more time
same way as other processes, regardless of the number of user threads.
Since the OS just knows about the process, it schedules the process the
Thread manager.
- Solving this problem requires communication between the kernel and the user-level.
- If a user-level thread is waiting for I/O, the entire process will wait.
- It might run a process that only has idle threads.
It may make poor scheduling decisions.
Since the OS does not know about the existence of the user-level threads.

User-Level Threads: Disadvantages

User-Level Threads are typically much faster than kernel threads
User-level threads do not require system calls to create them or context
switches to move between them.

User-Level Threads: Advantages

There is no context switch involved when switching threads.
Operations

User-level thread operations are orders of magnitude faster than similar kernel threads

---

Operation times in microseconds on a MIPS 3000

<table>
<thead>
<tr>
<th>Operation</th>
<th>Fast Threads</th>
<th>Native Threads</th>
<th>Topaz</th>
<th>Ultima</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signal/Wait</td>
<td>52</td>
<td>229</td>
<td>1846</td>
<td></td>
</tr>
<tr>
<td>Fork</td>
<td>39</td>
<td>328</td>
<td>11,920</td>
<td></td>
</tr>
<tr>
<td>Ultima</td>
<td>73</td>
<td>128</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Fast Threads**: multiple user threads per address space

**Topaz**: multiple kernel threads per address space

**Ultima**: 1 thread per address space

---

More Examples of Kernel and User-Level Threads

---

Example: Kernel and User-Level Threads in Solaris
- Implementation considerations
- Policy Options
- Policy Goals

From the ready que to execute?

Short Term Scheduling: How does the OS select a process

Primary memory?

multiple\programming, i.e., the number of jobs executing at once in the

Long Term Scheduling: How does the OS determine the degree of

**Scheduling Processes**

only one of the state queues.

All of the processes that the OS is currently managing reside in one and

Process Execution State

CPU activities.

OS to increase system utilization and throughput by overlapping I/O and

**Multitasking:** Running more than one process at a time enables the

**Scheduling Processes**
Response Time: The time between when a process is ready to run and its
next I/O request.

Waiting Time: The total amount of time that a process is in the ready
queue.

Turnaround Time: The length of time it takes to run a process from
initialization to termination, including all the waiting time.

Throughput: The number of processes completed in a unit of time.

CPU Utilization: The percentage of time that the CPU is busy.

Criteria for Comparing Scheduling Algorithms:

Preemptive system: the scheduler can interrupt a running process

Events

Non-preemptive system: the scheduler must wait for one of these

1. The kernel runs the scheduler at least when

Short Term Scheduling
Relax these assumptions. Researchers developed these algorithms in the 70’s when these

Processes are independent
•

One thread per process
•

One process per user
•

Simplifying Assumptions

Scheduling Policies:

This might actually increase average response time.

- Minimize waiting time - give each process the same amount of time on the processor.
- Minimize throughput - two components
- 2 different use of system resources (CPU, I/O devices)
  1. Minimize overhead (0S overhead, context switching)
  2. Efficient use of system resources (CPU, I/O devices)

Important than how average with a high variance.

- Minimize variance of response time - in interactive systems, predictability may be more
- Minimize average response time - provide output to the user as quickly as possible and
- Minimize average response time - provide output to the user as quickly as possible and

Instead, choose a scheduling algorithm based on its ability to satisfy a policy

(throughput, utilization, etc...), but this is not generally possible

Ideally, choose a CPU scheduler that optimizes all criteria simultaneously.

Scheduling Policies
blocks (say on an I/O device).

on I/O, but that is non-preemptive; i.e., the job keeps the CPU until it
was done I/O.

In early FCFS schedulers, the job did not relinquish the CPU even when it

The scheduler executes jobs to completion in arrival order.

FCFS: First-Come-First-Served (or FIFO: First-In-First-Out)

**Scheduling Policies**

**Lottery Scheduling:** Jobs get tickets and scheduler randomly picks winning

**Multi-level Feedback Queues:** Round Robin on each priority queue.

**SJF:** Shortest Job First

**Round Robin:** Use a time slice and preemption to alternate jobs.

**FCFS:** First Come, First Served

**Scheduling Algorithms: A Snapshot**
Idle

force I/O bound processes to wait for the CPU, leaving the I/O devices
may lead to poor overlap of I/O and CPU since CPU-bound processes will

Jobs

average wait time is highly variable as short jobs may wait behind long

Disadvantages:

Advantages:

Simple

FCS: Advantages and Disadvantages

If processes arrive 1 time unit apart, what is the average wait time in these three cases?

A requests I/O

0 1 2 3 4 5 7
0

A

A

B

C

A, B, C (A does I/O)

Arrival order: A, B, C (A does I/O)

10
0

A

B

C

A, B, C (no I/O)

Arrival order: A, B, C (no I/O)

10
0

B

C

A

B, C, A (no I/O)

Arrival order: B, C, A (no I/O)

Time

FCS Scheduling Policy: Example
throughput, ...). And goals of operating system (minimizing response time, maximizing decision and should be based on characteristics of processes being run, many scheduling algorithms exist. Selecting an algorithm is a policy used.

- Decisions, resulting in slower process execution than if kernel threads were user-level threads may result in the kernel making poor scheduling decisions.
- Since a context switch is not required, switching between user-level threads is faster than between kernel threads.
- Thread: a single execution stream within a process.

Summary