How should we schedule threads (or processes) onto the CPU?

What threads package?

Where should we implement threads? In the kernel? In a user-level?

Today: Threads

Processes communicate either with message passing or shared memory

context switch

The program currently executing on the CPU is changed by performing a

On a uniprocessor, there is at most one running process at a time.

A process is either New, Ready, Waiting, Running, or Terminated.

- PCBs contain process state, scheduling and memory management information, etc

- Processes are represented as Process Control Blocks in the OS

A process is the unit of execution.

Last Class: Processes
many threads per address space.

Operating systems can support one or many address spaces, and one or

Classifying Threaded Systems

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
process. Switching between kernel threads is slightly faster than switching between

it can be the same process scheduling algorithms.

- The kernel must manage and schedule threads (as well as processes), but
  - the kernel does not need to be changed since the threads share an address space.
  - Memory management information does not need to be changed once the threads
  - The values of registers, program counter, and stack pointer must be changed.

Switching between kernel threads of the same process requires a small
cycle switch.

- A kernel thread, also known as a lightweight process, is a thread that

\[ \text{Kernel Threads} \]

\[ \text{Example Threaded Program} \]

Library (user code):

Forking a thread can be a system call to the kernel or a procedure call to a thread

```
consume_item: index = buffer[in];
while (index < buffer[buffer[in]])
    repeat
        producer;
        repeat
            fork_thread(consumer());
            fork_thread(consumer());
            fork_thread(producer());
            fork_thread(producer());
        end
```
Thread Ready
Queue

Kernel Processes

User-Level Threads

Kernel

User-Level Thread Scheduler

Current Thread for each Process

Thread Ready Queue

Process Ready Queue

Kernel Threads

User-Level Threads

Deleting them, synchronizing them, and scheduling them.

The programmer uses a thread library to manage threads (create and

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.

User-Level Threads
slices the OS will dedicate to it.

- For kernel threads, the more threads a process creates, the more time
  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.
    - If might run a process that only has ID 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

<table>
<thead>
<tr>
<th>User-Level Threads: Disadvantages</th>
</tr>
</thead>
</table>

### User-Level Threads: Advantages

<table>
<thead>
<tr>
<th>User-Level Threads: Advantages</th>
</tr>
</thead>
</table>

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

other threads:

- A thread can voluntarily give up the processor by telling the scheduler it will yield to
  each process might use a different scheduling algorithm for its own threads.
- A user-level code can define a problem-dependent thread scheduling policy.

user-level thread scheduling is more flexible

There is no context switch involved when switching threads.
Kernel
Address Space
Lightweight process

<table>
<thead>
<tr>
<th>Operation Times in Microseconds on a MIPS 3000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>User</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

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

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

**Ultix**: 1 thread per address space

---

### More Examples of Kernel and User-Level Threads

Example: Kernel and User-Level Threads in Solaris

Diagram: Lightweight process connected to threads within the kernel and user-level threads.
- Implementation considerations
- Policy options
- Policy goals

from the ready queue to execute?

Short Term Scheduling: How does (or should) the OS select a process

Primary memory?

multiprogramming, i.e., the number of jobs executing at once in the

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

Process Execution State

CPU activities.

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

Multiprocessor: Running more than one process at a time enables the

only one of the state queues.

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

Waiting

Terminted

Running

New

Scheduler Processes
next I/O Request.

Response time: The time between when a process is ready to run and its

Queue.

Waiting time: The total amount of time that a process is in the Ready

Initialization to termination, including all the waiting time.

Turnaround time: The length of time it takes to run a process from

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

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

- 2. an interrupt occurs or
- 3. a process is created or terminated.
- 1. a process switches from running to waiting.

- The kernel runs the scheduler at least when

Short Term Scheduling
Relax these assumptions.

Assumptions were more realistic, and it is still an open problem how to
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:
1. Minimize waiting time — give each process the same amount of time on the processor.
2. Efficient use of system resources (CPU, I/O devices)
3. Minimize overhead (OS overhead, context switching)
4. Minimize throughput - two components
5. Important than a low variance with a high variance
6. Minimize variance of response time - in interactive systems, predictability may be more
7. Minimize average response time - provide output to the user as quickly as possible and
8. Minimize throughput, response time, as soon as it is received.

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

(throughput, utilization, throughput, ...), 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).

- We will assume a FCFS scheduler that runs when processes are blocked and was doing 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

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lottery Scheduling</td>
<td>Jobs get tickets and scheduler randomly picks winning ticket.</td>
</tr>
<tr>
<td>Multi-Level Feedback Quues</td>
<td>Round Robin on each priority queue.</td>
</tr>
<tr>
<td>SJF: Shortest Job First</td>
<td>Use a time slice and preemption to alternate jobs.</td>
</tr>
<tr>
<td>Round Robin: Use a time slice and preemption to alternate jobs.</td>
<td><strong>FCFS</strong>: First Come, First Served</td>
</tr>
</tbody>
</table>
FCS: Advantages and Disadvantages

Advantages: Simple

Disadvantages:

Advantages: Reduce I/O bound processes to wait for the CPU, leaving the I/O devices idle. Jobs may lead to poor overlap of I/O and CPU since CPU-bound processes will finish before the disk-bound processes.

Disadvantages:

Average wait time is highly variable as short jobs may wait behind long jobs.

FCS Scheduling Policy: Example

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

A requests I/O

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>A</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B</td>
<td>C</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
...throughput...

and goals of operating system (minimize response time, maximize
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
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