Today

- Project 1
- Processes

Project 1

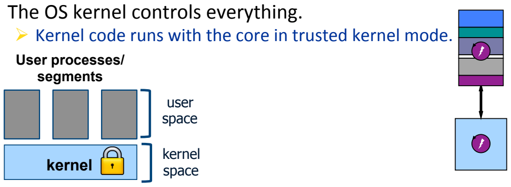
- Submission
  - Just like setup is different because we have a different programming environment, we have different submission
  - Use the instructions on the web page (rather than in the PDF)

Review

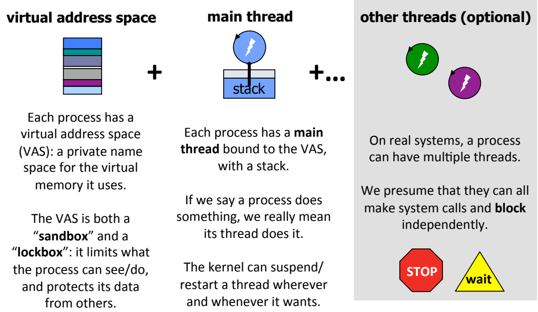
- What information/data is associated with a process?
- How do we create new processes?

The story so far: process and kernel

- A (classical) OS lets us run programs as *processes*.
- A process is a running program instance (with a thread).
  - Program code runs with the CPU core in untrusted user mode.
- Processes are protected/isolated.
  - Virtual address space is a "fenced pasture"
  - Sandbox: can't get out. Lockbox: nobody else can get in.
- The OS kernel controls everything.
  - Kernel code runs with the core in trusted kernel mode.



Processes and Their Threads



Each process has a virtual address space (VAS): a private name space for the virtual memory it uses.

The VAS is both a "**sandbox**" and a "**lockbox**": it limits what the process can see/do, and protects its data from others.

Each process has a **main thread** bound to the VAS, with a stack.

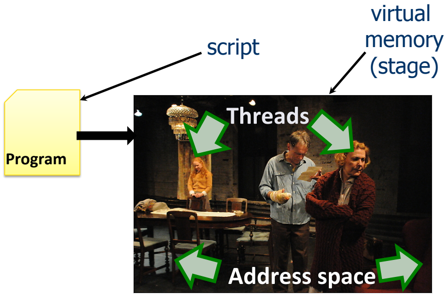
If we say a process does something, we really mean its thread does it.

The kernel can suspend/ restart a thread wherever and whenever it wants.

On real systems, a process can have multiple threads.

We presume that they can all make system calls and **block** independently.

Theater Analogy



Running a program is like performing a play.

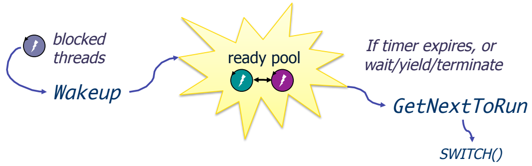
[lpcox]

Foreground and background

- A *multiprogrammed* OS can run many processes concurrently / simultaneously / at the same time.
- When you run a program as a command to the shell (e.g., Terminal), by default the process is *foreground*.
  - The shell calls the OS to create a child process to run the program, passes control of the terminal to the child process, and waits for the process to finish (exit).
- You can run a program in *background* with & syntax.
  - & is an arbitrary syntax used in Unix since the 1960s.
  - The shell creates the child process and starts it running, but keeps control of the terminal to accept another command.
  - & allows you to run multiple concurrent processes from shell

CPU Scheduling 101

- The OS scheduler makes a sequence of "moves"
  - Next move: if a CPU core is idle, pick a ready thread from the ready pool and dispatch it (run it).
  - Scheduler's choice is "nondeterministic"
  - Scheduler and machine determine the interleaving of execution (a schedule).

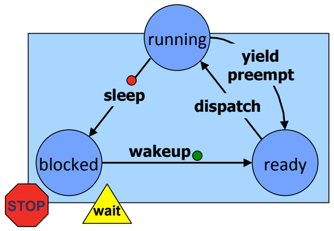


Exceptions: trap, fault, interrupt

	intentional happens every time	**unintentional** contributing factors
synchronous caused by an instruction	**trap: system call** open, close, read, write, fork, exec, exit, wait, kill, etc.	**fault** invalid or protected address or opcode, page fault, overflow, etc.
asynchronous caused by some other event	"software interrupt" software requests an interrupt to be delivered at a later time	**interrupt** caused by an external event: I/O op completed, clock tick, power fail, etc.

Thread states and transitions

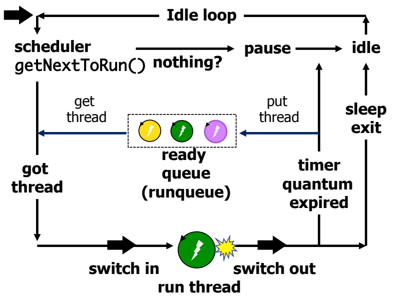
If a thread is in the **ready** state thread, then the system may choose to run it "at any time". The kernel can switch threads whenever it gains control on a core, e.g., by a timer interrupt. If the current thread takes a fault or system call trap, and blocks or exits, then the scheduler switches to another thread. But it could also preempt a **running** thread. From the point of view of the program, dispatch and preemption are **nondeterministic**: we can't know the **schedule** in advance.



These **preempt** and **dispatch** transitions are controlled by the kernel scheduler.

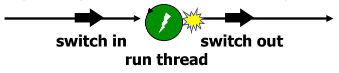
Sleep and **wakeup** transitions are initiated by calls to internal sleep/wakeup APIs by a running thread.

What cores do



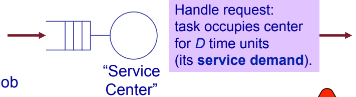
Switching out

- What causes a core to switch out of the current thread?
  - Fault+sleep or fault+kill
  - Trap+sleep or trap+exit
  - Timer interrupt: quantum expired
  - Higher-priority thread becomes ready
  - …?



Note: the thread switch-out cases are **sleep, forced-yield**, and **exit**, all of which occur in kernel mode following a **trap**, **fault**, or **interrupt**. But a trap, fault, or interrupt does not necessarily cause a thread switch!

Understanding performance: queues



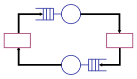
Requests wait here in queue

offered load
request stream @
arrival rate λ
(requests/time)
Request == task == job

"Service Center"
(e.g., a CPU core)

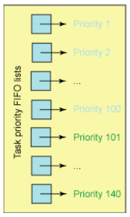
Handle request:
task occupies center
for *D* time units
(its **service demand**).

Performance analysts study behavior of **queuing networks** under various assumptions about the workload and scheduling policies.

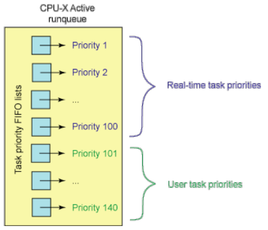


Priority

- Most modern OS schedulers use priority scheduling
  - Each task/thread has a priority value (integer)
  - The scheduler favors higher-priority threads
  - Threads inherit a base priority from the associated process
  - User-settable relative importance within application
  - Internal priority adjustments as an implementation technique within the scheduler.
  - How to set the priority of a thread?
- How many priority levels?
  - 32 (Windows) to 128 (OS X)



Ordering runqueues by priority



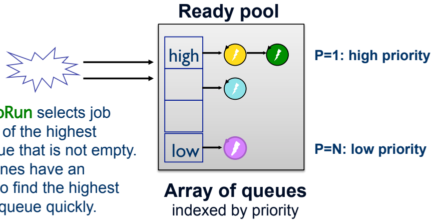
ready queue (runqueue)

In real systems, the simple "cartoon ready queue" may be a **multi-level queue**: an ordered array of queues, one for each priority level.

In a typical OS, each thread has a ***priority***, which may change over time.
When a core is idle, pick a thread with highest priority.
If a higher-priority thread becomes ready, then preempt the thread currently running on the core and switch to the new thread.

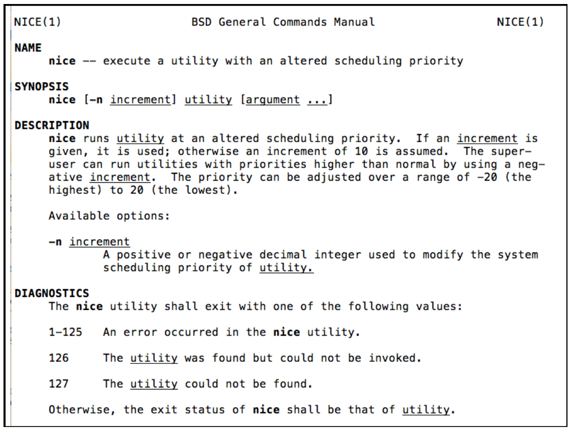
Multi-level queue

Multi-level priority queue structures are commonly used in OSs to represent the run queue == ready pool == ready list.



GetNextToRun selects job at the head of the highest priority queue that is not empty.
Most machines have an instruction to find the highest non-empty queue quickly.

constant time, no sorting

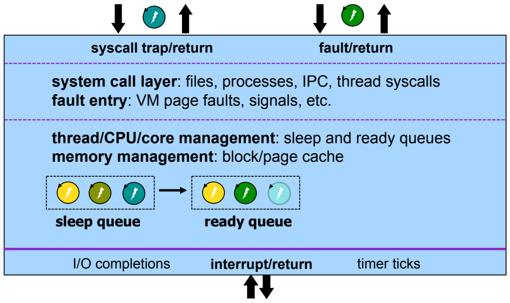


```
NICE(1)                    BSD General Commands Manual                   NICE(1)

NAME
     nice -- execute a utility with an altered scheduling priority

SYNOPSIS
     nice [-n increment] utility [argument ...]

DESCRIPTION
     nice runs utility at an altered scheduling priority.  If an increment is
     given, it is used; otherwise an increment of 10 is assumed.  The super-
     user can run utilities with priorities higher than normal by using a neg-
     ative increment.  The priority can be adjusted over a range of -20 (the
     highest) to 20 (the lowest).

     Available options:

     -n increment
             A positive or negative decimal integer used to modify the system
             scheduling priority of utility.

DIAGNOSTICS
     The nice utility shall exit with one of the following values:

     1-125   An error occurred in the nice utility.

     126     The utility was found but could not be invoked.

     127     The utility could not be found.

     Otherwise, the exit status of nice shall be that of utility.
```

Processor allocation policy

- Key issue: how should an OS allocate its CPU resources among contending demands?
  - Resource allocation policy: how the OS controls use of hardware resources.
- Focus on OS kernel
  - User code can decide how to use the processor time it is given
- Which thread to run on a free core? GetNextThreadToRun
- For how long? How long to let it run before we take the core back and give it to some other thread?
  - timeslice or quantum
- What are the policy goals?

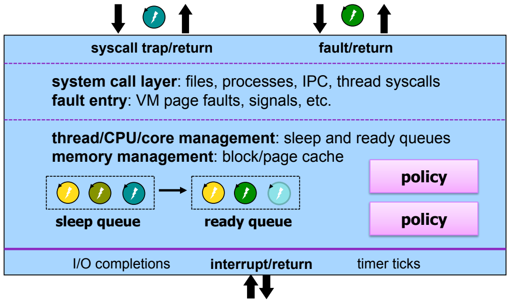
Separation of policy and mechanism



system call layer: files, processes, IPC, thread syscalls
fault entry: VM page faults, signals, etc.

thread/CPU/core management: sleep and ready queues
memory management: block/page cache

Separation of policy and mechanism



system call layer: files, processes, IPC, thread syscalls
fault entry: VM page faults, signals, etc.

thread/CPU/core management: sleep and ready queues
memory management: block/page cache

Scheduler policy goals

What measures help us evaluate
if a schedule or scheduler is "good"?

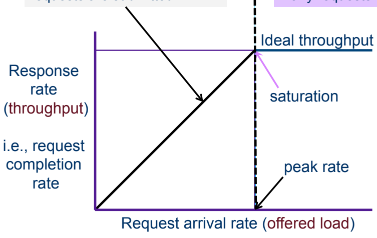
Scheduler policy goals

- Response time or latency, responsiveness
  - How long does it take to complete a task or request? (R)
  - Say a task takes D time units of work (its service demand)
    - But how long does it spend waiting for service?
- Throughput
  - How many tasks/requests complete per unit of time? (X)
  - Utilization: what % of time is each core/device busy? (U)
- Meet deadlines, reduce jitter for periodic tasks
  - e.g., videos and other continuous media

Ideal throughput: cartoon version

throughput == arrival rate
The center is not saturated: it completes requests at the rate requests are submitted.

throughput == peak rate
The center is saturated. It can't go any faster, no matter how many requests are submitted.



This graph shows throughput (e.g., of a server) as a function of offered load. It is idealized: your mileage may vary.

Throughput: reality

Thrashing, also called congestion collapse
Real servers/devices often have some pathological behaviors at saturation. E.g., they abort requests after investing work in them (thrashing), which wastes work, reducing throughput.

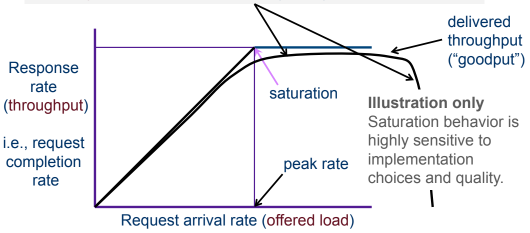
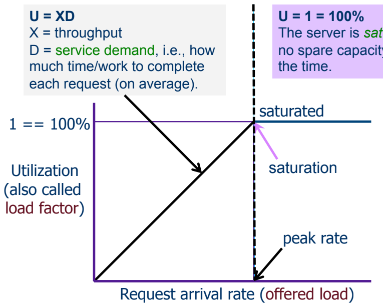


Illustration only
Saturation behavior is highly sensitive to implementation choices and quality.

Utilization

- What is the probability that the center is busy?
  - Answer: some number between 0 and 1.
- What percentage of the time is the center busy?
  - Answer: some number between 0 and 100
- These are interchangeable: called utilization U
- The probability that the service center is idle is 1-U

Utilization: cartoon version



U = XD
X = throughput
D = service demand, i.e., how much time/work to complete each request (on average).

U = 1 = 100%
The server is *saturated*. It has no spare capacity. It is busy all the time.

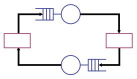
This graph shows utilization (e.g., of a server) as a function of offered load. It is idealized: each request works for D time units on a single service center (e.g., a single CPU core).

The Utilization "Law"

- If the center is not saturated then:
  - U = λD = (arrivals/time) * service demand
- Reminder: that's a rough average estimate for a mix of arrivals with average service demand D.
- If you actually measure utilization at the center, it may vary from this estimate.
  - But not by much.

Understanding utilization and throughput

- Throughput/utilization are "easy" to understand for a single service center that stays busy whenever there is work to do.
- It is more complex for a network of centers/queues that interact, and where each task/job/request uses multiple centers.
- And that's what real computer systems look like.
  - E.g., CPU, disk, network, and mutexes…
  - Other synchronization objects



Is high utilization good or bad?

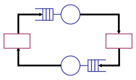
Understanding utilization and throughput

Is high utilization good or bad?

Good. We don't want to pay $$$ for resources and then leave them idle. Especially if there is useful work for them to do!

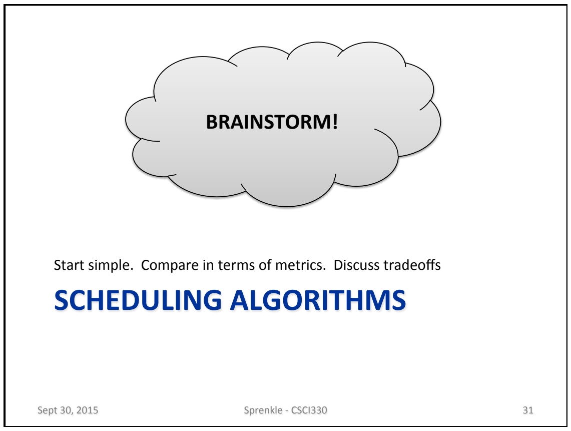
Bad. We want to serve any given workload as efficiently as possible. And we want resources to be ready for use when we need them.

Utilization <-> contention



Another goal: fairness

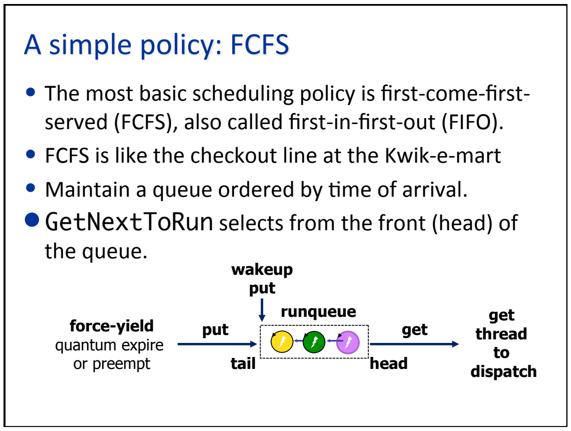
- When resources are shared, fairness is important.
- But what does fairness really mean? What makes an allocation or schedule "fair"?
  - "Divide the pie" evenly? (Or according to weighted shares?)
  - Low variance in allocations or wait times? (Or equal slowdown)
  - e.g., "Jain fairness index"
  - Freedom from starvation? (Or upper bound on wait time)
  - Serve the clients who pay the most? (Market-based)
  - Serve the clients who benefit the most? (Maximize global welfare)
  - Freedom from envy?
- This is a deep and interesting topic. But we skip it.



BRAINSTORM!

Start simple. Compare in terms of metrics. Discuss tradeoffs

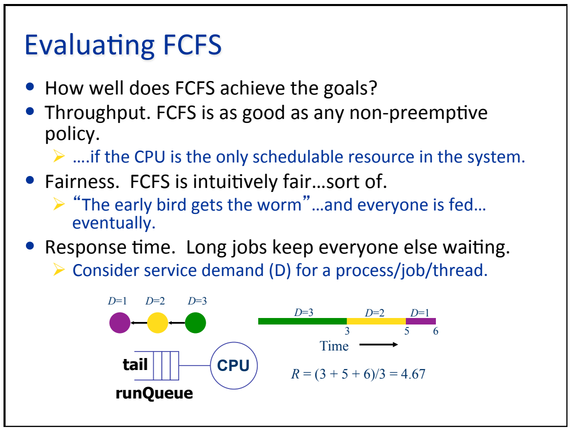
SCHEDULING ALGORITHMS



A simple policy: FCFS

- The most basic scheduling policy is first-come-first-served (FCFS), also called first-in-first-out (FIFO).
- FCFS is like the checkout line at the Kwik-e-mart
- Maintain a queue ordered by time of arrival.
- `GetNextToRun` selects from the front (head) of the queue.

wakeup
put
runqueue
force-yield
quantum expire
or preempt
put
tail
head
get
get
thread
to
dispatch



Evaluating FCFS

- How well does FCFS achieve the goals?
- Throughput. FCFS is as good as any non-preemptive policy.
  - ….if the CPU is the only schedulable resource in the system.
- Fairness. FCFS is intuitively fair…sort of.
  - "The early bird gets the worm"…and everyone is fed… eventually.
- Response time. Long jobs keep everyone else waiting.
  - Consider service demand (D) for a process/job/thread.

D=1 D=2 D=3

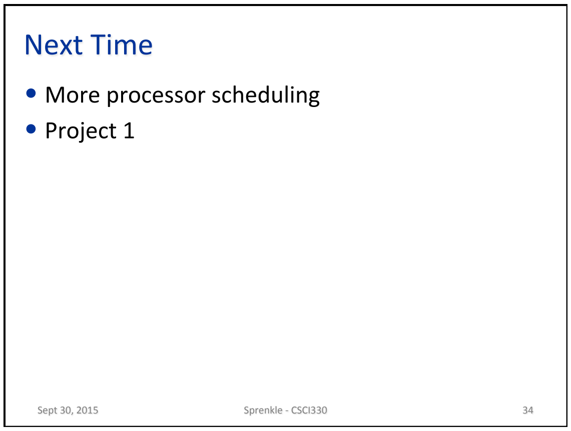
tail
CPU
runQueue

D=3 D=2 D=1

3 5 6

Time

$R = (3 + 5 + 6)/3 = 4.67$



Next Time

- More processor scheduling
- Project 1