Outline for Today

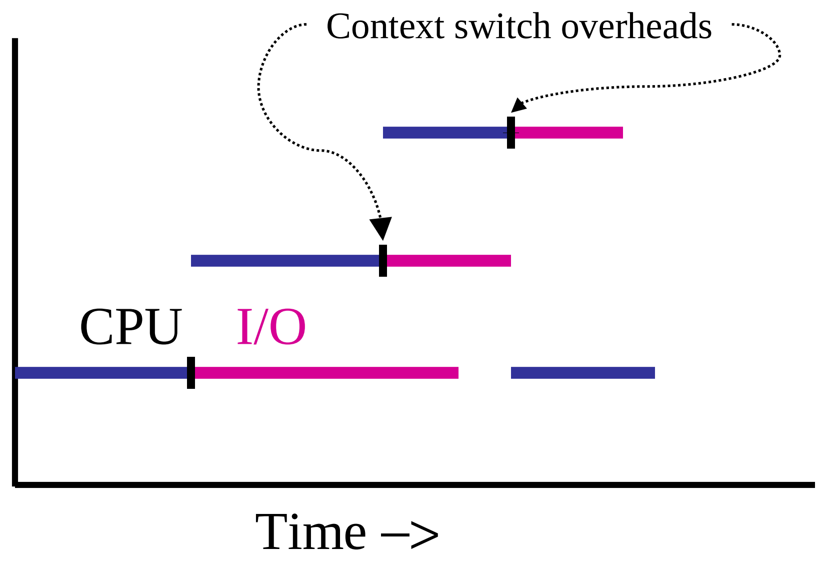
- Objectives:
  - – Scheduling (continued).
  - – System Calls and Interrupts.
- Announcements

Scheduler Policy Goals & Metrics of Success

- *Response time* or latency (to minimize the average time between arrival to completion of requests)
  - How long does it take to do what I asked? (***R***) Arrival –> done.
- *Throughput* (to maximize productivity)
  - How many operations complete per unit of time? (***X***)
- *Utilization* (to maximize use of some device)
  - What percentage of time does the CPU (and each device) spend doing useful work? (***U***)
    time-in-use / elapsed time
- *Fairness*
  - What does this mean? Divide the pie evenly? Guarantee low variance in response times? Freedom from starvation?
  - Proportional sharing of resources
- *Meet deadlines and guarantee jitter-free periodic tasks*
  - real time systems (e.g. process control, continuous media)

Multiprogramming and Utilization

- Early motivation: *Overlap* of computation and I/O
- Determine *mix* and *multiprogramming level* with the goal of “covering” the idle times caused by waiting on I/O.



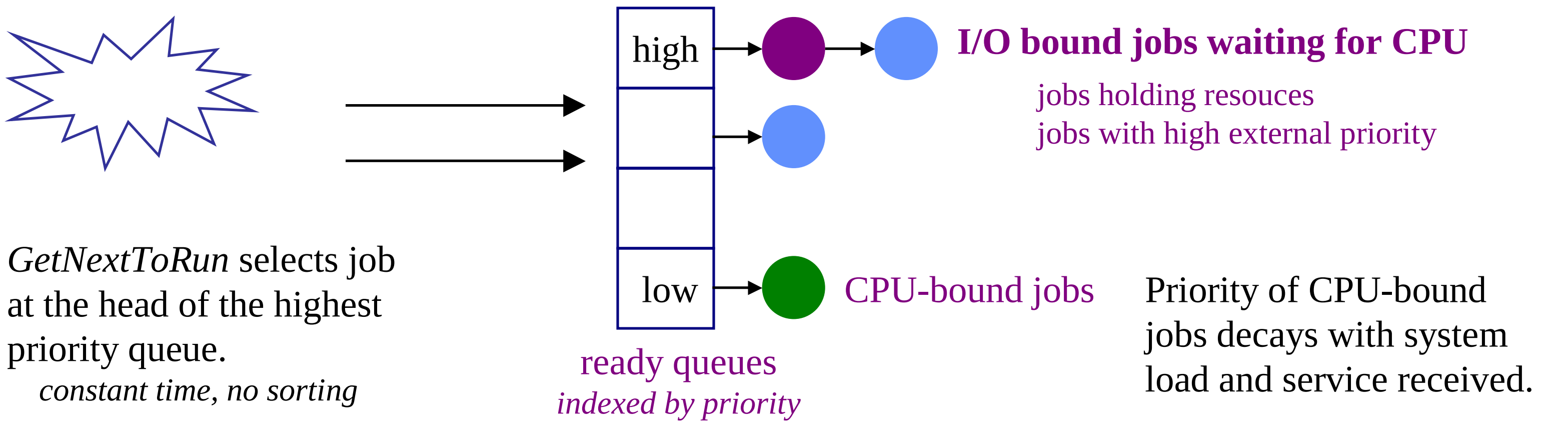
Gantt Chart

Classic Scheduling Algorithms

- SJF - Shortest Job First (provably optimal in minimizing average response time, assuming we know service times in advance)
- FIFO, FCFS
- Round Robin
- Multilevel Feedback Queuing
- Priority Scheduling (using priority queue data structure)

Multilevel Feedback Queue

- Many systems (e.g., Unix variants) use a *multilevel feedback queue*.
  - *multilevel*. Separate queue for each of *N* priority levels.
  - *feedback*. Factor previous behavior into new job priority.


high
low
I/O bound jobs waiting for CPU
jobs holding resouces
jobs with high external priority
CPU-bound jobs
Priority of CPU-bound jobs decays with system load and service received.
GetNextToRun selects job at the head of the highest priority queue.
constant time, no sorting
ready queues
indexed by priority

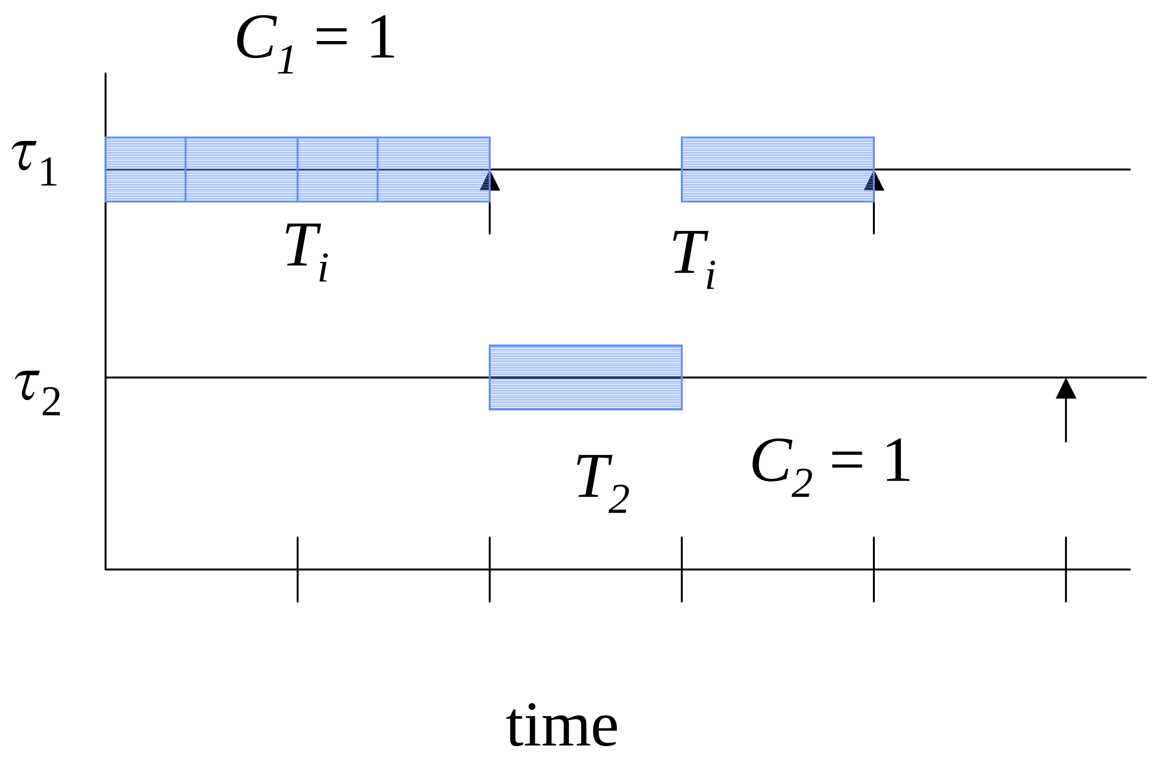
Real Time Schedulers

- Real-time schedulers must support regular, periodic execution of tasks (e.g., continuous media).
  - *CPU Reservations*
    - "I need to execute for *X* out of every *Y* units."
    - Scheduler exercises *admission control* at reservation time: application must handle failure of a reservation request.
  - *Proportional Share*
    - "I need *1/n* of resources"
  - *Time Constraints*
    - "Run this before my *deadline* at time *T*."

Assumptions

- Tasks are periodic with constant interval between requests, T_i (request rate $1/T_i$)
- Each task must be completed before the next request for it occurs
- Tasks are independent
- Run-time for each task is constant (max), C_i
- Any non-periodic tasks are special

Task Model



Definitions

- Deadline is time of next request
- Overflow at time *t* if *t* is deadline of unfulfilled request
- Feasible schedule - for a given set of tasks, a scheduling algorithm produces a schedule so no overflow ever occurs.
- Critical instant for a task - time at which a request will have largest response time.
  - Occurs when task is requested simultaneously with all tasks of higher priority

Rate Monotonic

- Assign priorities to tasks according to their request rates, independent of run times
- Optimal in the sense that no other fixed priority assignment rule can schedule a task set which can not be scheduled by rate monotonic.
- If feasible (fixed) priority assignment exists for some task set, rate monotonic is feasible for that task set.

Earliest Deadline First

- Dynamic algorithm
- Priorities are assigned to tasks according to the deadlines of their current request
- With EDF there is no idle time prior to an overflow
- For a given set of *m* tasks, EDF is feasible iff

$$C_1/T_1 + C_2/T_2 + \ldots + C_m/T_m \leq 1$$

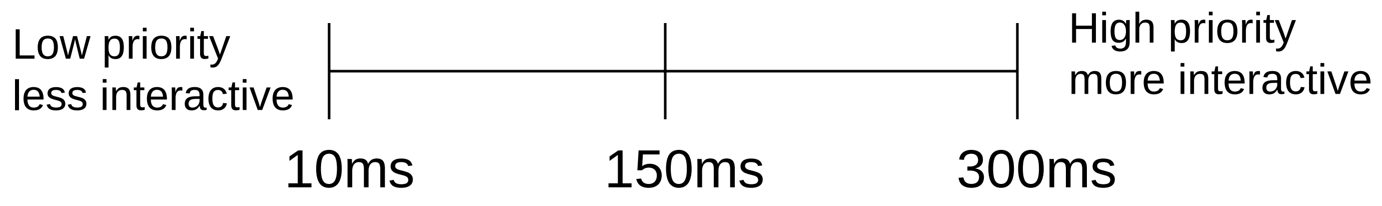
- If a set of tasks can be scheduled by any algorithm, it can be scheduled by EDF

Linux Scheduling Policy

- Runnable process with highest priority and timeslice remaining runs (SCHED_OTHER policy)
  - Dynamically calculated priority
    - Starts with nice value
    - Bonus or penalty reflecting whether I/O or compute bound by tracking sleep time vs. runnable time: sleep_avg and decremented by timer tick while running

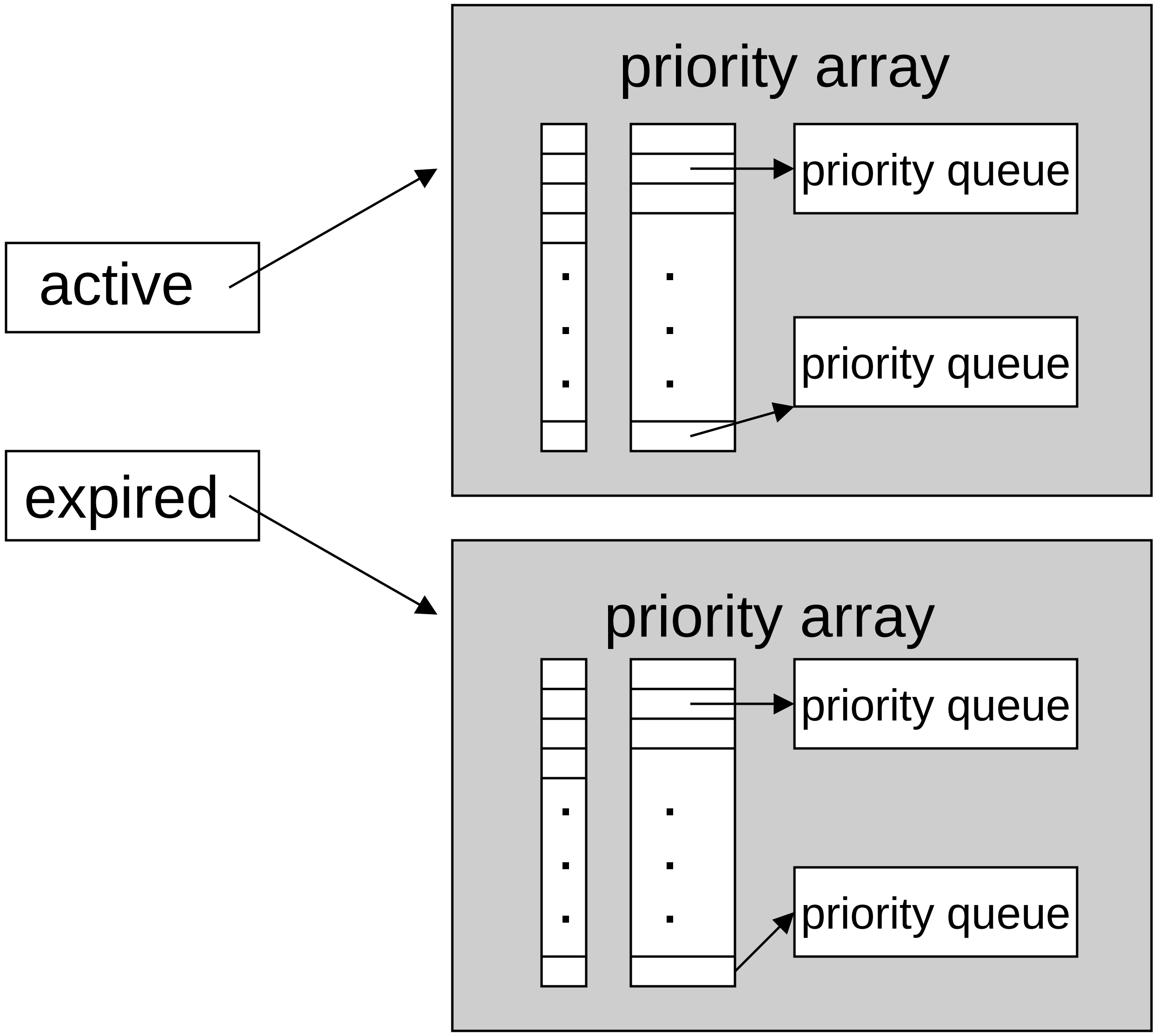
Linux Scheduling Policy

- Dynamically calculated timeslice
  - The higher the dynamic priority, the longer the timeslice:



- Recalculated every round when “expired” and “active” swap
- Exceptions for expired interactive
  - Go back on active unless there are starving expired tasks

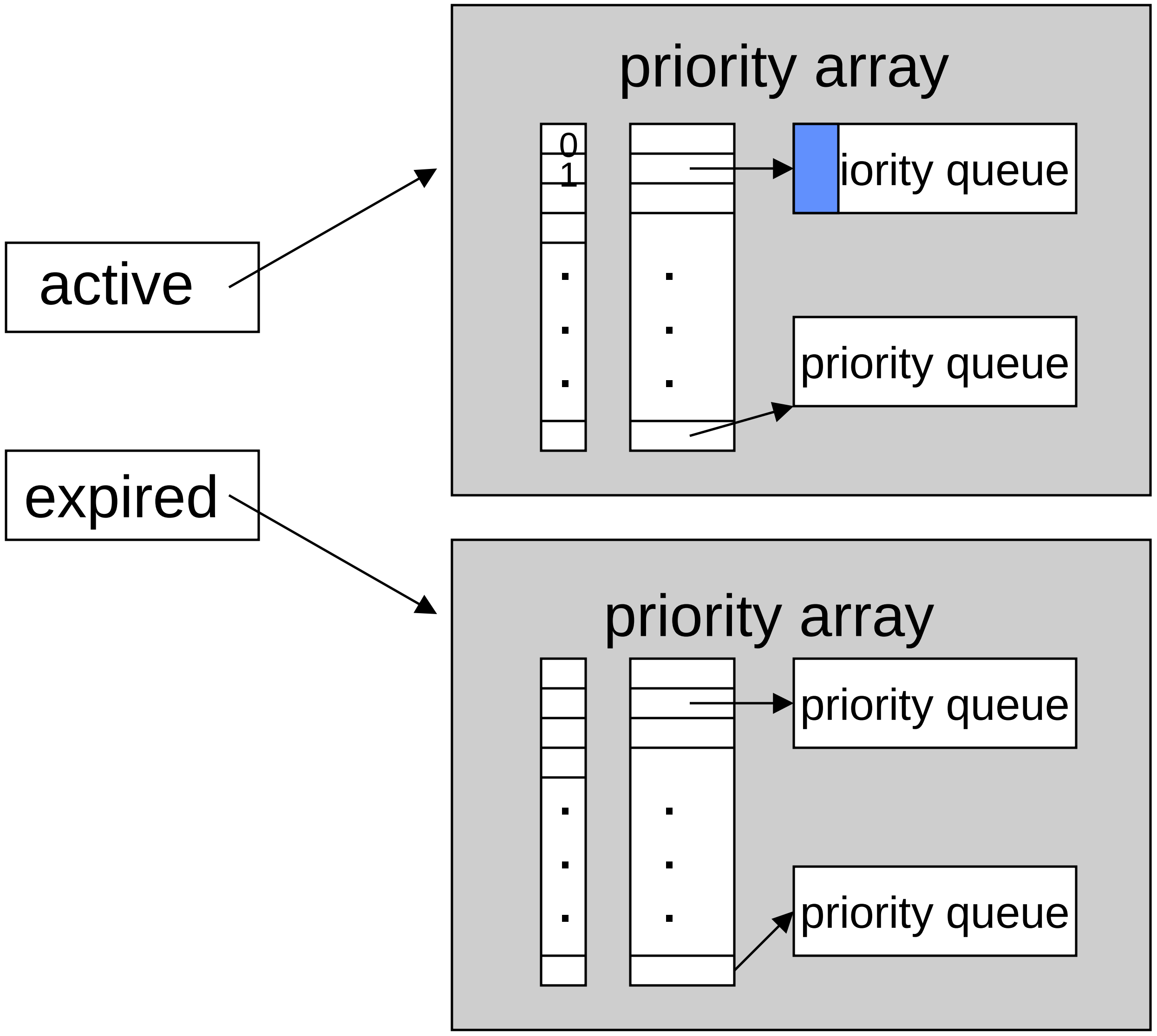
Runqueue for O(1) Scheduler



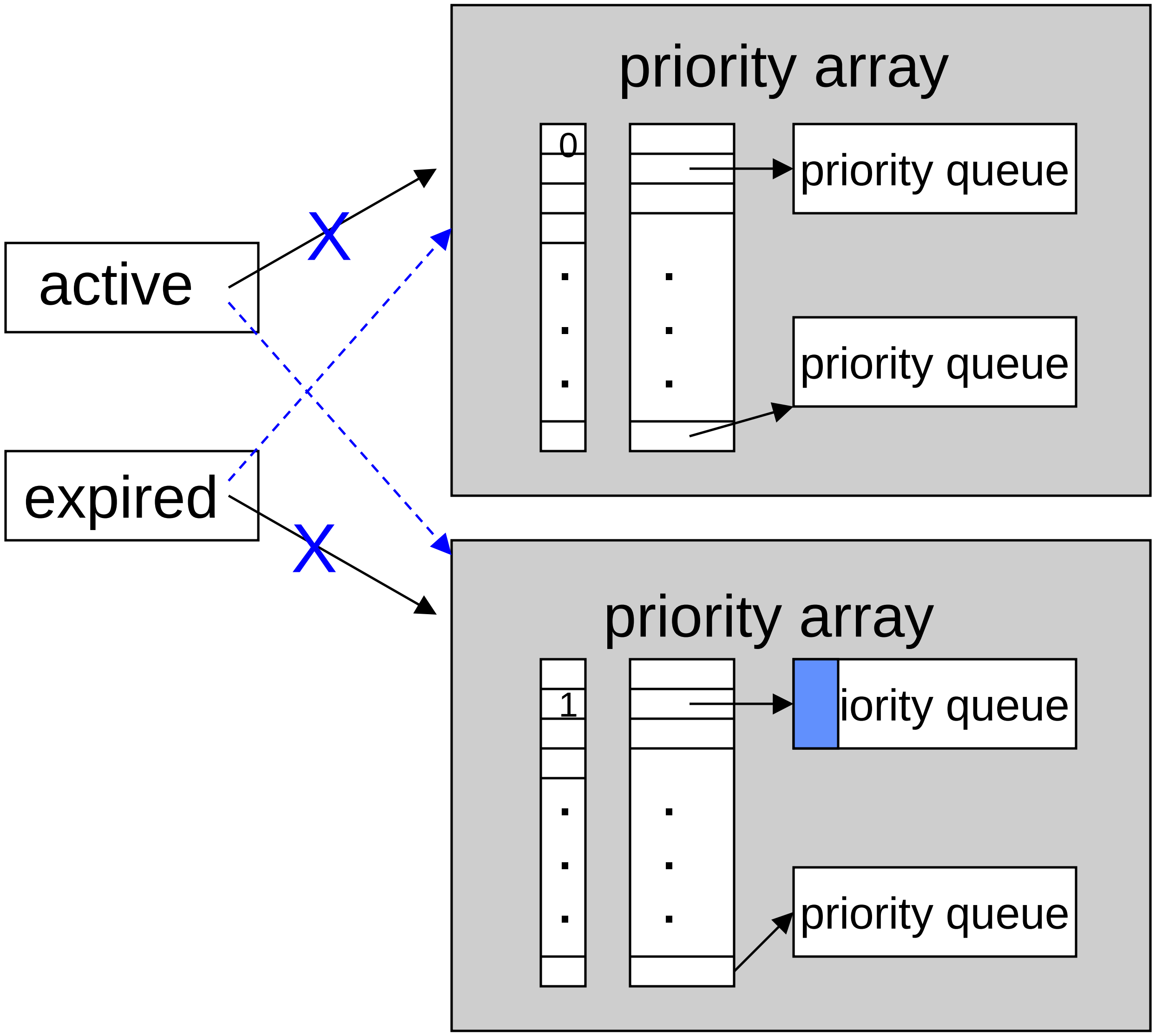
Higher priority
more I/O
300ms

lower priority
more CPU
10ms

Runqueue for O(1) Scheduler



Runqueue for O(1) Scheduler



Linux Real-time

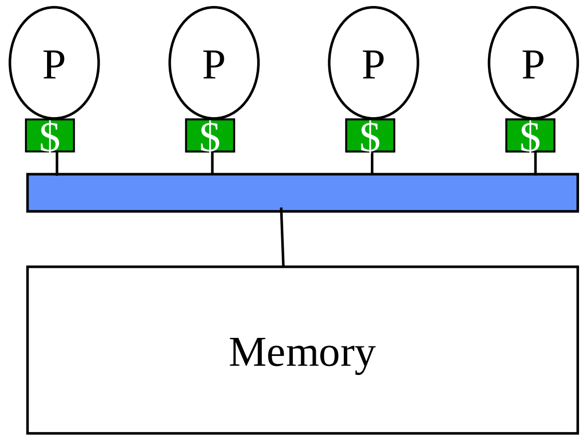
- No guarantees
- SCHED_FIFO
  - Static priority, effectively higher than SCHED_OTHER processes*
  - No timeslice – it runs until it blocks or yields voluntarily
  - RR within same priority level
- SCHED_RR
  - As above but with a timeslice.

* Although their priority number ranges overlap

Support for SMP

- Every processor has its own private runqueue
- Locking – spinlock protects runqueue
- Load balancing – pulls tasks from busiest runqueue into mine.
- Affinity – cpus_allowed bitmask constrains a process to particular set of processors

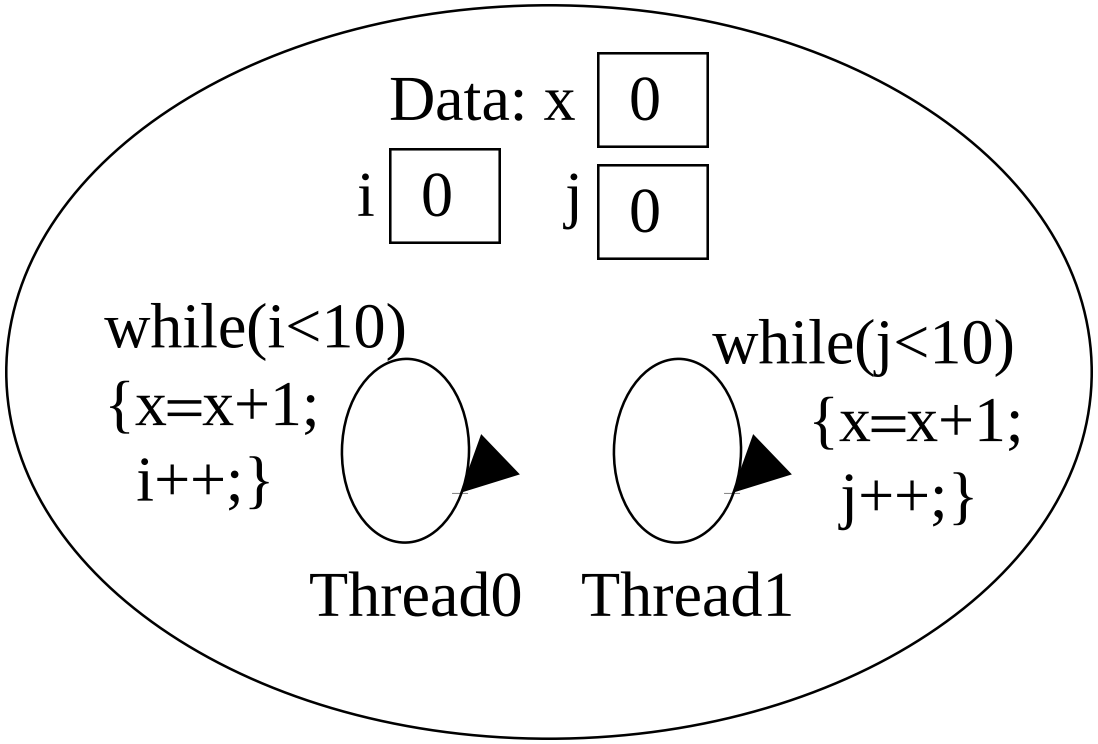
Symmetric mp



- load_balance runs from schedule( ) when runqueue is empty or periodically esp. during idle.
- Prefers to pull processes from expired, not cache-hot, high priority, allowed by affinity

Synchronization

The Trouble with Concurrency in Threads...



What is the value of x when both threads leave this while loop?

Range of Answers

Process 0	Process1
LD x // x currently 0	
	LD x // x currently 0
	Add 1
	ST x // x now 1
	Do 8 more full loops // x = 9
Add 1	
ST x // x now 1, stored over 9	
	LD x // x now 1
Do 9 more full loops // leaving x at 10	
	Add 1
	ST x // x = 2 stored over 10

Nondeterminism

- What unit of work can be performed without interruption? ***Indivisible*** or ***atomic*** operations.
- ***Interleavings*** - possible execution sequences of operations drawn from all threads.
- ***Race condition*** - final results depend on ordering and may not be "correct".

while (i<10) {$\underline{\text{x=x+1}}$; i++;}

load value of x into reg

yield( )

add 1 to reg

yield ( )

store reg value at x

yield ( )

Reasoning about Interleavings

- On a uniprocessor, the possible execution sequences depend on when context switches can occur
  - Voluntary context switch - the process or thread explicitly yields the CPU (blocking on a system call it makes, invoking a Yield operation).
  - Interrupts or exceptions occurring - an asynchronous handler activated that disrupts the execution flow.
  - Preemptive scheduling - a timer interrupt may cause an involuntary context switch at any point in the code.
- On multiprocessors, the ordering of operations on shared memory locations is the important factor.

Critical Sections

- If a sequence of non-atomic operations must be executed *as if* it were atomic in order to be correct, then we need to provide a way to constrain the possible interleavings in this ***critical section*** of our code.
  - Critical sections are code sequences that contribute to “bad” race conditions.
  - Synchronization needed around such critical sections.
- ***Mutual Exclusion*** - goal is to ensure that critical sections execute atomically w.r.t. related critical sections in other threads or processes.
  - How?

The Critical Section Problem

Each process follows this template:

```
while (1)
{ ...other stuff...   //processes in here shouldn't stop
   others
   enter_region( );
   critical section
   exit_region( );
}
```

The problem is to define enter_region and exit_region to ensure mutual exclusion with some degree of fairness.

Implementation Options for Mutual Exclusion

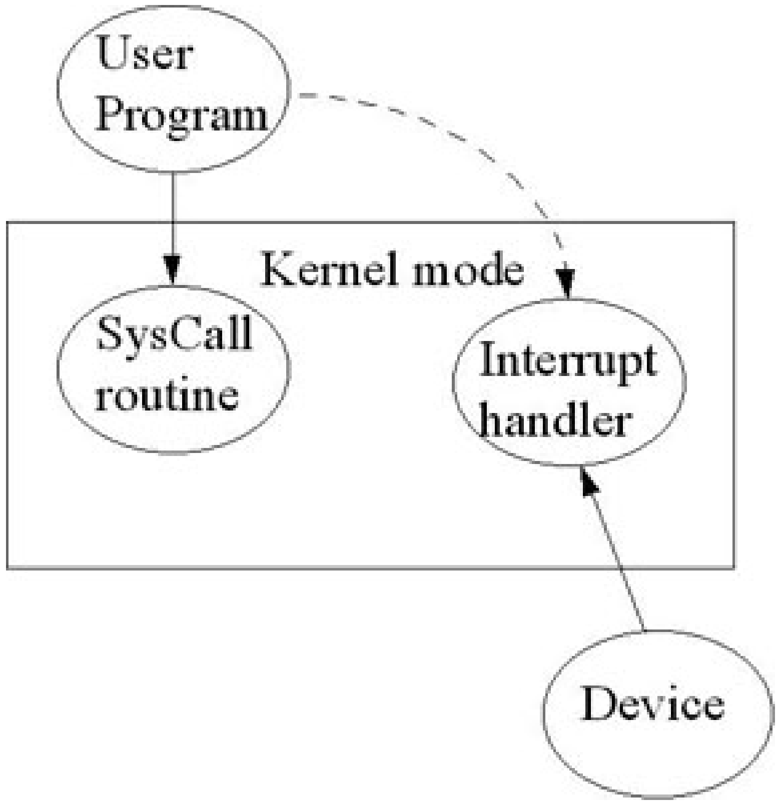
- Disable Interrupts
- Busywaiting solutions - spinlocks
  - execute a tight loop if critical section is busy
  - benefits from specialized atomic (read-mod-write) instructions
- Blocking synchronization
  - sleep (enqueued on wait queue) while C.S. is busy

Synchronization primitives (abstractions, such as locks) which are provided by a system may be implemented with some combination of these techniques.

Kernel Entry

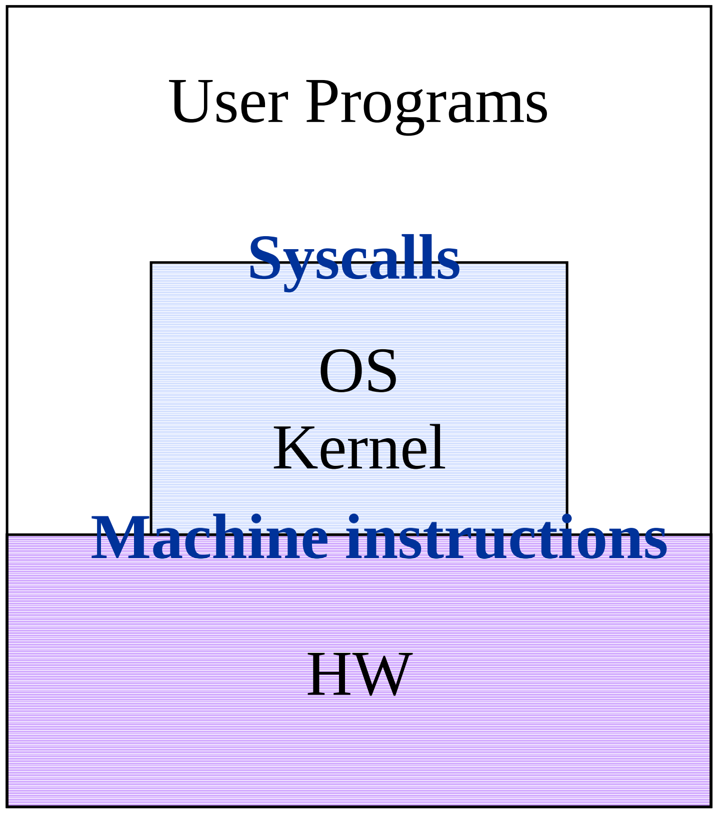
Crossing Protection Boundaries

- For a user to do something "privileged", it must invoke an OS procedure providing that service. How?
- System Calls
  - special trap instruction that causes an exception which vectors to a kernel handler
  - parameters indicate which system routine called



User / Kernel Modes

- Hardware support to differentiate between what we'll allow user code to do by itself (user mode) and what we'll have the OS do (kernel mode).
- Mode indicated by status bit in protected processor register.
- Privileged instructions can only be executed in kernel mode (I/O instructions).
- Protected memory space



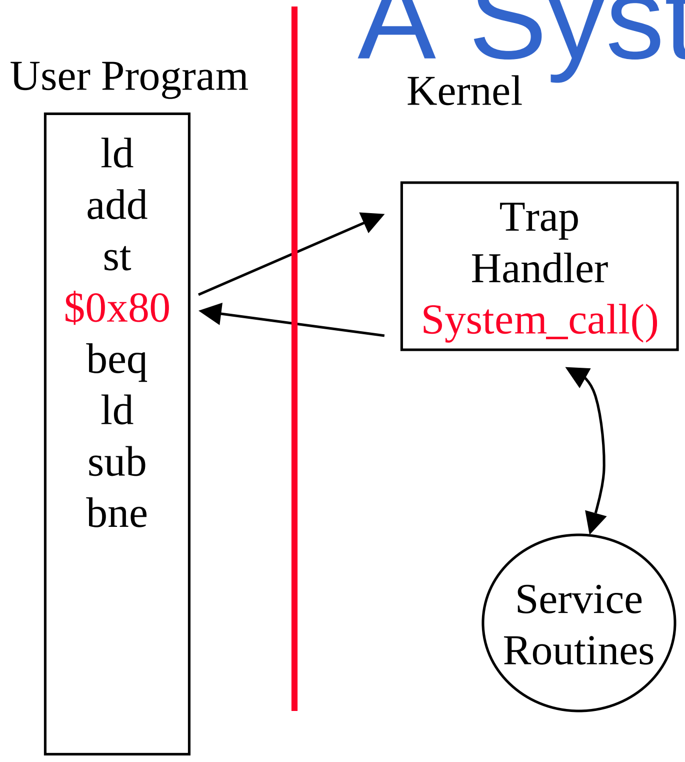
Interrupts and Exceptions

- Unnatural change in control flow
- Interrupt is external event
  - devices: disk, network, keyboard, etc.
  - clock for timeslicing
  - These are useful events, must do something when they occur.
- Exception is potential problem with program
  - segmentation fault
  - bus error
  - divide by 0
  - Don’t want my bug to crash the entire machine
  - page fault (virtual memory…)
- System calls leverage this mechanism

An Execution Context

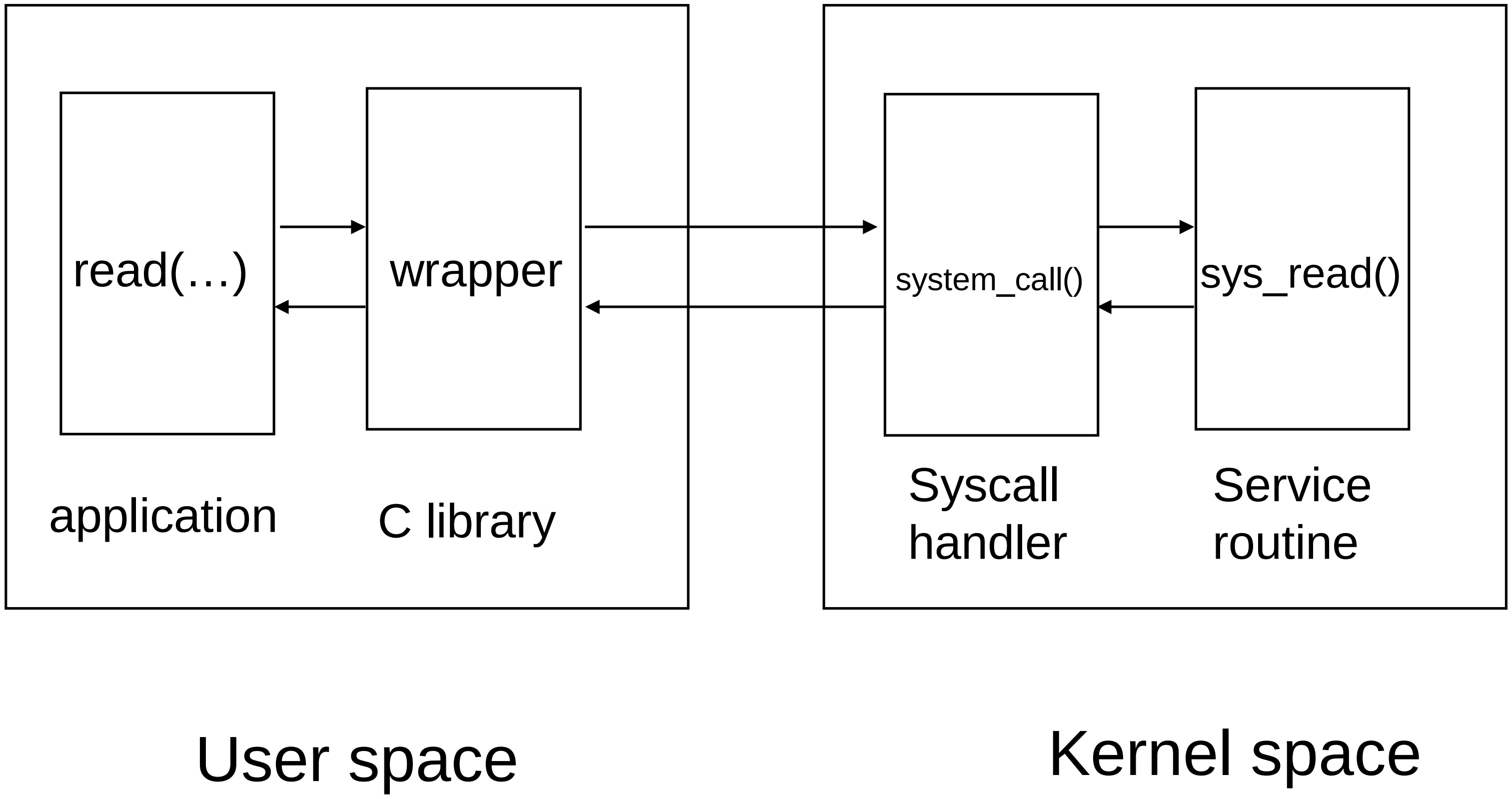
- The state of the CPU associated with a thread of control (process context)
  - general purpose registers (integer and floating point)
  - status registers (e.g., condition codes)
  - program counter, stack pointer
- Kernel executes in process context during system calls
  - Preemptible, kernel capable of sleeping
  - Linux system calls must be reentrant
- Need to be able to switch between contexts
  - better utilization of machine (overlap I/O of one process with computation of another)
  - timeslicing: sharing the machine among many processes

A System Call



- Special Instruction to change modes and invoke service
  - read/write I/O device
  - create new process
- Invokes specific kernel routine based on argument
  - Syscall # in eax register
- kernel defined interface
  - Arguments passed in registers
- May return from trap to different process -- schedule()
- instruction to return to user process

Usual Path to Invoking System Call



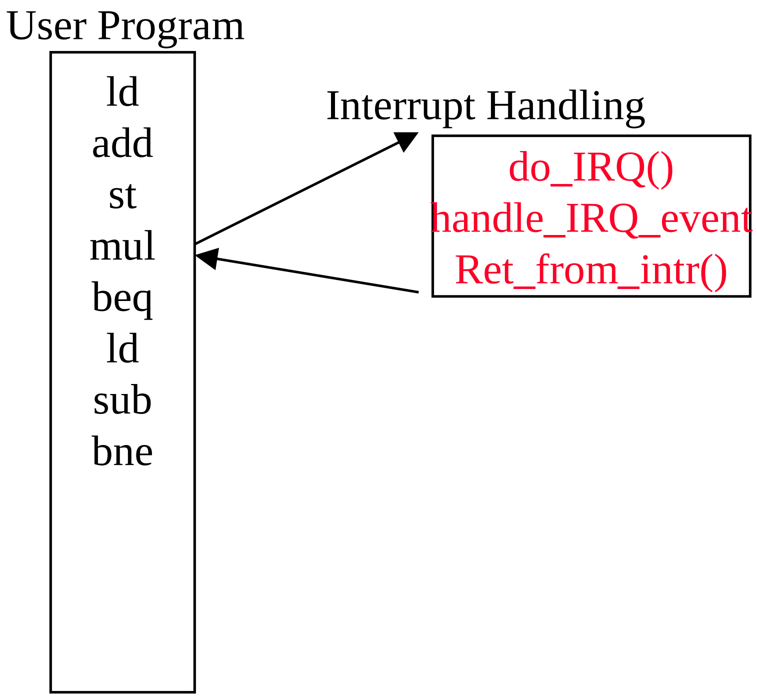
Role of Interrupts in I/O

So, the program needs to access an I/O device…

- Start an I/O operation (special instructions or memory-mapped I/O)
- Device controller performs the operation asynchronously (in parallel with) CPU processing (between controller's buffer & device).
- If DMA, data transferred between controller's buffer and memory without CPU involvement.
- Interrupt signals I/O completion when device is done.

CPU handles Interrupt

- Device raises interrupt line, CPU detects this, CPU stops current operation, disables interrupts, enters kernel mode, saves current program counter, jumps to predefined location, saves other processor state needed to continue at interrupted instruction.
- For each interrupt number, jumps to address of appropriate interrupt service routine. [do_IRQ()]
  - Interrupt context
  - Kernel stack of whatever was interrupted
  - Can not sleep
- Handlers on this line do what needs to be done. [handle_IRQ_event()]
  - Unless SA_INTERRUPT specified when registered, re-enable interrupts during handler execution
  - If line is shared, loop through all handlers
- Restores saved state at interrupted instruction [ret_from_intr()], returns to user mode.



Interrupt Control

- local_irq_disable() and local_irq_enable() -- affecting all interrupts for this processor
- local_irq_save(…) and local_irq_restore(…) – save and disable interrupts on this processor and restore previous interrupt state
- disable_irq(irq), disable_irq_nosynch(irq), enable_irq(irq) – affecting particular interrupt line
- Informational:
  - irqs_disabled() – local interrupts disabled?
  - in_interrupt() – in interrupt context or process context?
  - In_irq() – executing an interrupt handler?

Bottom Half Processing

- Deferring work that is too heavyweight for interrupt handling
- Mechanisms:
  - Softirqs
    “soft interrupts”, statically defined (32 max.) action functions that can run concurrently on SMP
    pending when bit set in 32-bitmask (usually set in associated interrupt handler [raise_softirq()]
    run with interrupts enabled, proper locking required
  - Tasklets
    dynamically created functions that are built upon softirqs,
    two of the same type can not run concurrently
    lists of tasklet_struct hooked to 2 of the softirqs
  - Workqueue
    implemented as kernel-based “worker” threads with process context of their own -- thus allowed to sleep