Outline for Today

- Objectives:
  - To review
    - the process and thread abstractions.
    - the mechanisms for implementing processes (threads), including scheduling
  - To detail
    - the Linux design decisions.
- Announcements

OS Abstractions

Abstract machine environment. The OS defines a set of logical resources (objects) and operations on those objects (an interface for the use of those objects).

Hides the physical hardware.

(Traditional) Unix Abstractions

- Processes - thread of control with context
- Files - everything else
  - Regular file – named, linear stream of data bytes
  - Sockets - endpoints of communication, possible between unrelated processes
  - Pipes - unidirectional I/O stream, can be unnamed
  - Devices

Process Abstraction

The Basics of Processes

- Processes are the *OS-provided abstraction* of multiple tasks (including user programs) executing concurrently.
- One instance of a program (which is only a passive set of bits) *executing* (implying an execution context – register state, memory resources, etc.)
- OS schedules processes to share CPU.

Why Use Processes?

- To capture naturally concurrent activities within the structure of the programmed system.
- To gain speedup by overlapping activities or exploiting parallel hardware.
  - From DMA to multiprocessors

Separation of Policy and Mechanism

- “Why and What” vs. “How”
- Objectives and strategies vs. data structures, hardware and software implementation issues.
- Process abstraction vs. Process machinery

Process Abstraction

- Unit of scheduling
- One (or more*) sequential threads of control
  - program counter, register values, call stack
- Unit of resource allocation
  - address space (code and data), open files
  - sometimes called *tasks* or *jobs*
- Operations on processes: fork (clone-style creation), wait (parent on child), exit (self-termination), signal, kill.

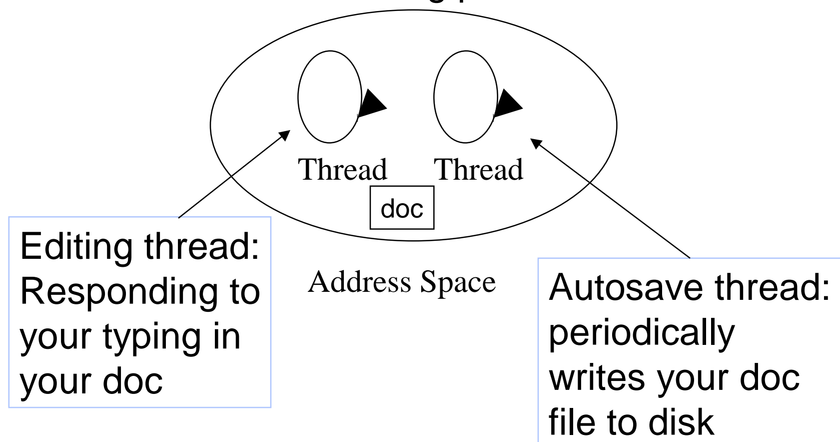
Process-related System Calls in Unix.

Threads and Processes

- Decouple the resource allocation aspect from the control aspect
- Thread abstraction - defines a single sequential instruction stream (PC, stack, register values)
- Task or process - the resource context serving as a “container” for one or more threads (shared address space)
- Kernel-supported threads - unit of scheduling
  (kernel-supported thread operations –> generally slow)

An Example

Doc formatting process



User Level Thread Packages

- To avoid the performance penalty of kernel-supported threads, implement at user level and manage by a run-time system
  - Contained “within” a single kernel entity (process)
  - Invisible to OS (OS schedules their container, not being aware of the threads themselves or their states). Poor scheduling decisions possible.
- User-level thread operations can be 100x faster than kernel thread operations, but need better integration / cooperation with OS.

Linux Processes

- Processes and threads are not differentiated – with varying degrees of shared resources
- clone() system call takes flags to determine what resources parent and child processes will share:
  - Open files
  - Signal handlers
  - Address space
  - Same parent

Process-related System Calls

- Simple and powerful primitives for process creation and initialization.
  - Unix ***fork*** creates a *child* process as (initially) a clone of the parent
    [Linux: fork() implemented by clone() system call]
  - parent program runs in child process – maybe just to set it up for ***exec***
  - child can ***exit***, parent can ***wait*** for child to do so.
    [Linux: wait4 system call]

Unix Process Relationships

*The **fork** syscall returns a zero to the child and the child process ID to the parent.*

__Fork__ creates an exact copy of the parent process.

```
int pid;
int status = 0;

if (pid = fork()) {
        /* parent */
        …..
        pid = wait(&status);
} else {
        /* child */
        …..
        exit(status);
}
```

*Parent uses **wait** to sleep until the child exits; wait returns child pid and status.*

__Wait__ variants allow wait on a specific child, or notification of stops and other signals.

*Child process passes status back to parent on **exit**, to report success/failure.*

Child Discipline

- After a *fork*, the parent *program* has complete control over the behavior of its child.
- The child inherits its execution environment from the parent...but the parent program can change it.
- Parent program may cause the child to execute a different program, by calling *exec** in the child context.

Exec, Execve, etc.

- Children should have lives of their own.
- Exec* “boots” the child with a different executable image.
  - parent program makes *exec** syscall (in forked child context) to run a program in a new child process
  - *exec** overlays child process with a new executable image
  - restarts in user mode at predetermined entry point (e.g., *crt0*)
  - no return to parent program (it’s gone)
  - arguments and environment variables passed in memory
  - file descriptors etc. are unchanged

“Join” Scenarios

- Several cases must be considered for join (e.g., *exit/wait*).
  - What if the child exits before the parent joins?
    - “Zombie” process object holds child status and stats.
  - What if the parent continues to run but never joins?
    - How not to fill up memory with zombie processes?
  - What if the parent exits before the child?
    - Orphans become children of **init** (process 1).
  - What if the parent can’t afford to get “stuck” on a join?
    - Unix makes provisions for asynchronous notification.

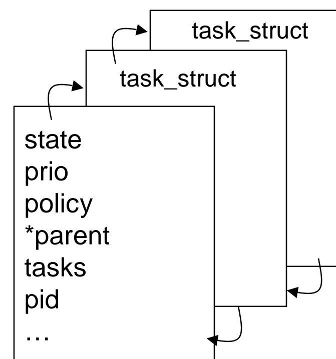
Process Mechanisms

Context Switching

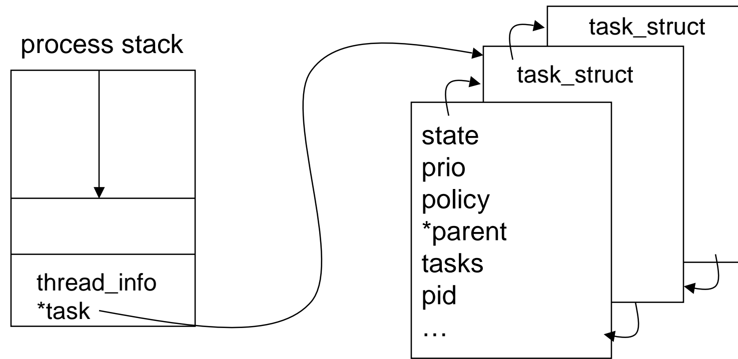
- When a process is running, its program counter, register values, stack pointer, etc. are contained in the hardware registers of the CPU. The process has direct control of the CPU hardware for now.
- When a process is not the one currently running, its current register values are saved in a process descriptor data structure (task_struct)
- Context switching involves moving state between CPU and various processes' descriptors by the OS.

Linux task_struct

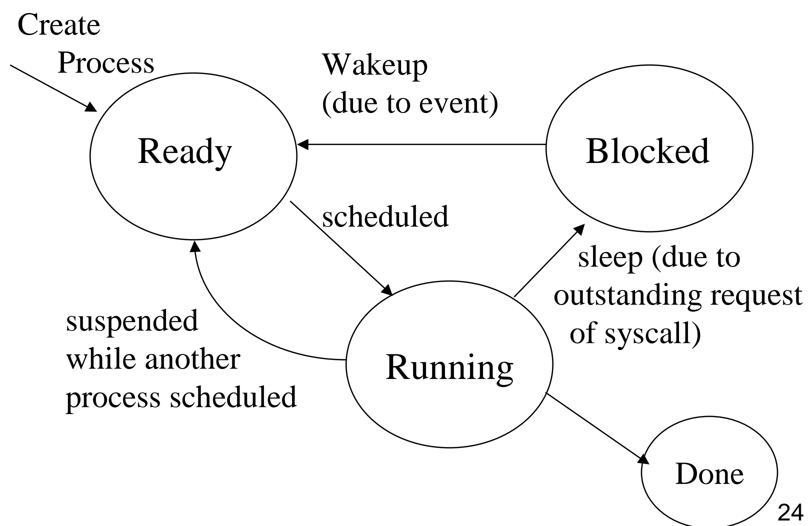
- Process descriptor in kernel memory represents a process (allocated on process creation, deallocated on termination).
  - Linux: task_struct, located via task pointer in thread_info structure on process's kernel state.



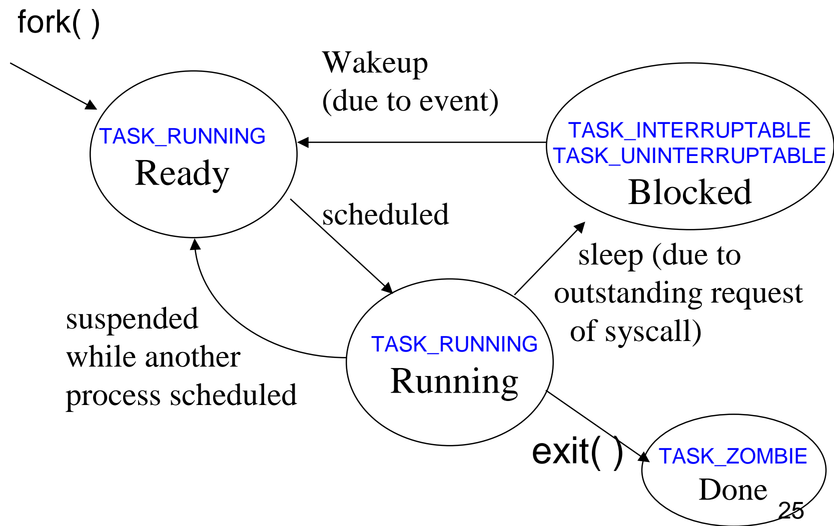
Linux task_struct



Process State Transitions

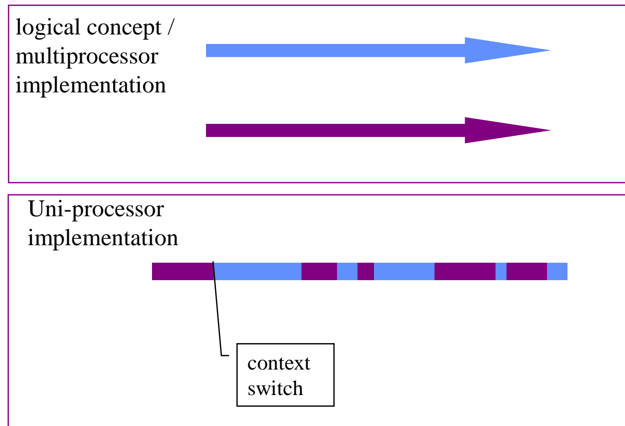


Linux Process States



Scheduling

Interleaved Schedules



Scheduling: Policy and Mechanism

- Scheduling policy answers the question:
  Which process/thread, among all those ready to run, should be given the chance to run next? In what *order* do the processes/threads get to run? For how long?
- Mechanisms are the tools for supporting the process/thread abstractions and affect *how* the scheduling policy can be implemented.
  - How the process or thread is represented to the system - process descriptor.
  - What happens on a context switch.
  - When do we get the chance to make these scheduling decisions (timer interrupts, thread operations that yield or block, user program system calls)

Flavors

- ***Long-term scheduling*** - which jobs get resources (e.g. get allocated memory) and the chance to compete for cycles (to be on the ready queue).
- ***Short-term scheduling or process scheduling*** - which of those gets the next slice of CPU time
- ***Non-preemptive*** - the running process/thread has to explicitly give up control
- ***Preemptive*** - interrupts cause scheduling opportunities to reevaluate who should be running now (is there a more "valuable" ready task?)

Preemption

- Scheduling policies may be *preemptive* or *non-preemptive*.
    - *Preemptive*: scheduler may unilaterally force a task to relinquish the processor before the task blocks, yields, or completes.
  - *timeslicing* prevents jobs from monopolizing the CPU
    - Scheduler chooses a job and runs it for a *quantum* of CPU time.
    - A job executing longer than its quantum is forced to yield by scheduler code running from the clock interrupt handler.
  - use preemption to honor priorities
    - Preempt a job if a higher priority job enters the *ready* state.

Priority

- Some goals can be met by incorporating a notion of *priority* into a "base" scheduling discipline.
    - Each job in the ready pool has an associated priority value; the scheduler favors jobs with higher priority values.
- *External priority* values:
    - imposed on the system from outside
    - reflect external preferences for particular users or tasks
        - "All jobs are equal, but some jobs are more equal than others."
    - *Example*: Unix **nice** system call to lower priority of a task.
    - *Example*: Urgent tasks in a real-time process control system.
- *Internal priorities*
    - scheduler dynamically calculates and uses for queuing discipline. System adjusts priority values internally as as an *implementation technique* within the scheduler.

Internal Priority

- Drop priority of tasks consuming more than their share
- Boost tasks that already hold resources that are in demand
- Boost tasks that have starved in the recent past
- Adaptive to observed behavior: typically a continuous, dynamic, readjustment in response to observed conditions and events
    - May be visible and controllable to other parts of the system
    - Priority reassigned if I/O bound (large unused portion of quantum) or if CPU bound (nothing left)

Keeping Your Priorities Straight

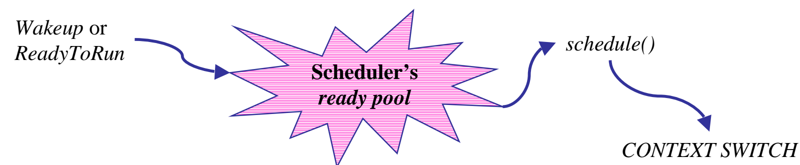
- Priorities must be handled carefully when there are dependencies among tasks with different priorities.
  - A task with priority ***P*** should never impede the progress of a task with priority ***Q*** > ***P***.
    - This is called *priority inversion*, and it is to be avoided.
  - The basic solution is some form of *priority inheritance*.
    - When a task with priority ***Q*** waits on some resource, the holder (with priority ***P)*** temporarily inherits priority ***Q*** if ***Q*** > ***P***.
    - Inheritance may also be needed when tasks coordinate with IPC.
  - Inheritance is useful to meet deadlines and preserve low-jitter execution, as well as to honor priorities.

Pitfalls: Mars Pathfinder Example

- In July 1997, Pathfinder's computer reset itself several times during data collection and transmission from Mars.
  - One of its processes failed to complete by a deadline, triggering the reset.
- Priority Inversion Problem.
  - A low priority process held a mutual exclusion semaphore on a shared data structure, but was preempted to let higher priority processes run.
  - The higher priority process which failed to complete in time was blocked on this semaphore.
  - Meanwhile a bunch of medium priority processes ran, until finally the deadline ran out. The low priority semaphore-holding process never got the chance to run again in that time to get to the point of releasing the semaphore
  - *Priority inheritance* had not been enabled on semaphore.

CPU Scheduling Policy

- The CPU scheduler makes a sequence of "moves" that determines the *interleaving* of threads.
  - Programs use synchronization to prevent "bad moves".
  - …but otherwise scheduling choices appear (to the program) to be *nondeterministic*.

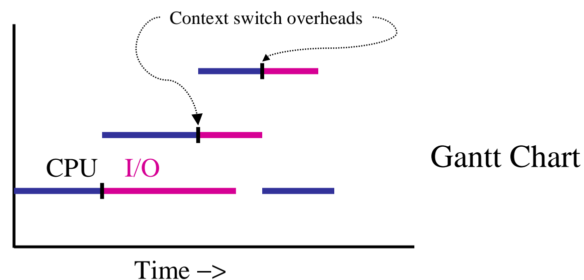


Scheduler Policy Goals & Metrics of Success

- *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.

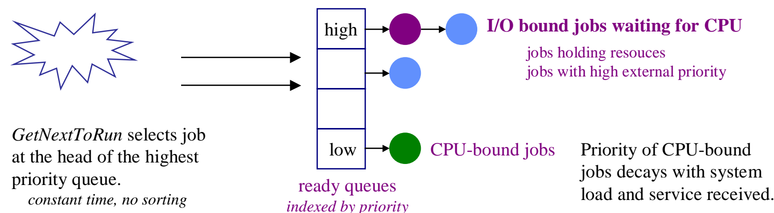


Classic Scheduling Algorithms

- 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.



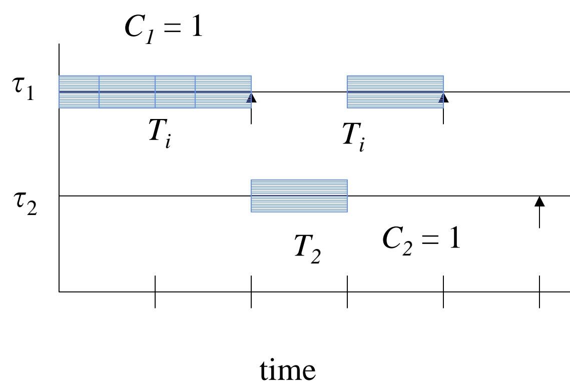
Real Time Schedulers

- 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

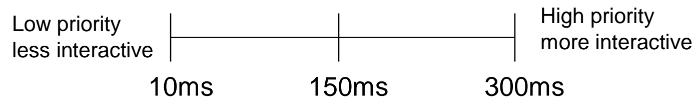
- Dynamic algorithm
- Priorities are assigned to tasks according to the deadlines of their current request
- With EDF there is no idle time prior to an overflow
- For a given set of *m* tasks, EDF is feasible iff
  $C_1/T_1 + C_2/T_2 + \ldots + C_m/T_m \leq 1$
- If a set of tasks can be scheduled by 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

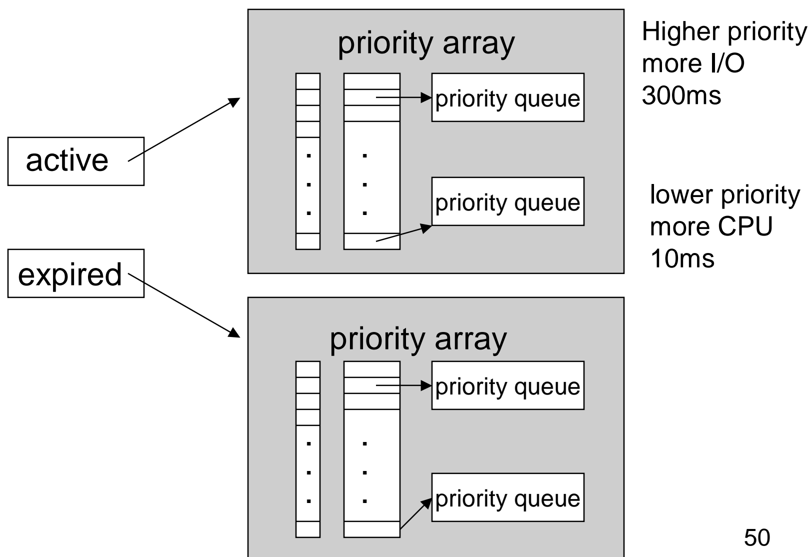
– Dynamically calculated timeslice
  - The higher the dynamic priority, the longer the timeslice:



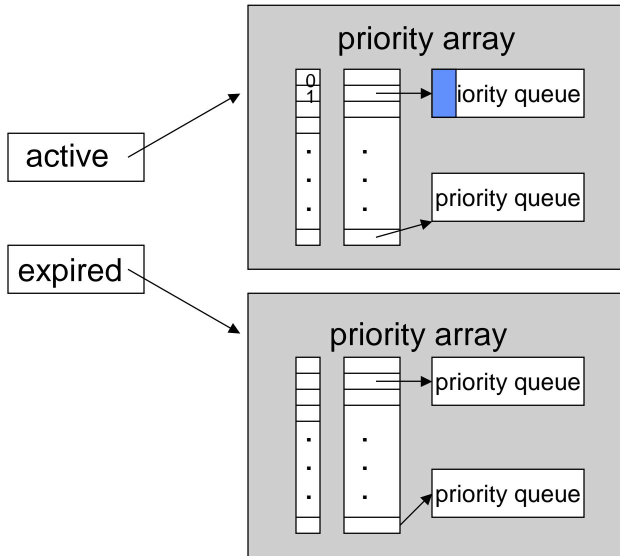
– Recalculated every round when "expired" and "active" swap

– Exceptions for expired interactive
  - Go back on active unless there are starving expired tasks

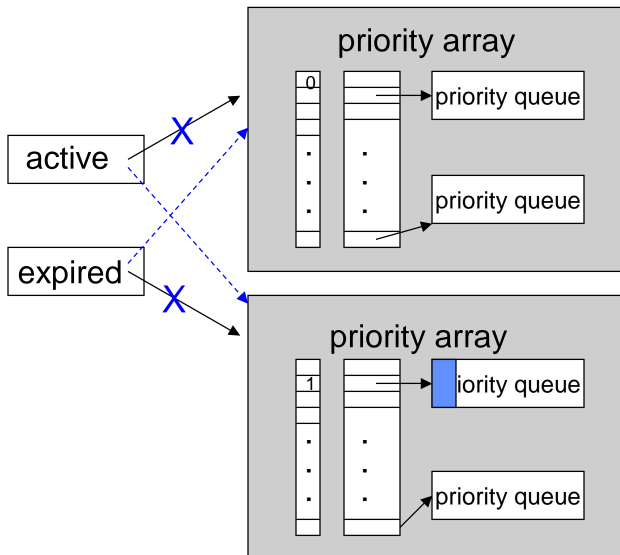
Runqueue for O(1) Scheduler



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

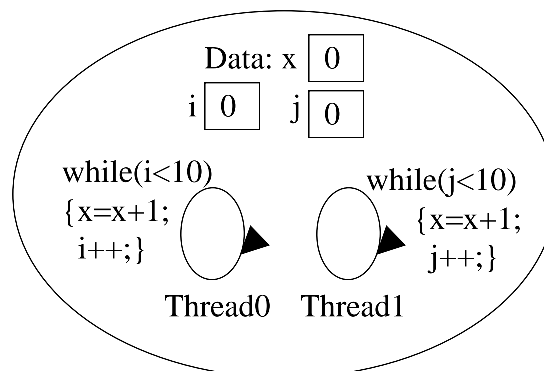
Symmetric mp

P P P P
$ $ $ $
Memory

- 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

LD x // x currently 0

Process1

LD x // x currently 0
Add 1
ST x // x now 1
Do 8 more full loops // x = 9

Process 0

Add 1
ST x // x now 1, stored over 9

Process1

LD x // x now 1

Process 0

Do 9 more full loops // leaving x at 10

Process1

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) {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.