# CS 3204
# Operating Systems

Lecture 16

Godmar Back

## Announcements

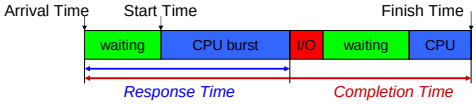
- Project 2 due March 20
- Additional office hours
  - Jai: Friday from 11am – 1pm and Monday from 12 – 2.
  - Xiaomo: Thursday (today) 1:30 -3:30pm and Tuesday 1:00 - 3:00pm
- Midterm March 29



## Scheduling

## CPU Scheduling Terminology

- A job (sometimes called a task, or a job instance)
  - Activity that's scheduled: process or part of a process
- Arrival time: time when job arrives
- Start time: time when job actually starts
- Finish time: time when job is done
- Completion time (aka Turn-around time)
  - Finish time – Arrival time
- Response time
  - Time when user sees response – Arrival time
- Execution time (aka cost): time a job needs to execute





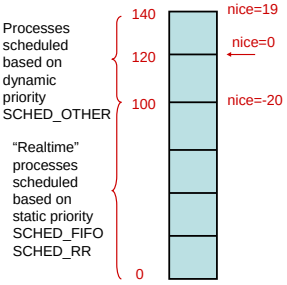

## Basic Scheduling: Summary

- FCFS: simple
  - unfair to short jobs & poor I/O performance (convoy effect)
- RR: helps short jobs
  - loses when jobs are equal length
- SPN: optimal average waiting time
  - which, if ignoring blocking time, leads to optimal average completion time
  - unfair to long jobs
  - requires knowing (or guessing) the future
- MLFQS: approximates SPN without knowing execution time
  - Can still be unfair to long jobs



## Case Study: Linux Scheduler

- Variant of MLFQS
- 140 priorities
  - 0-99 "realtime"
  - 100-140 nonrealtime
- Dynamic priority computed from static priority (nice) plus "interactivity bonus"



## Linux Scheduler (2)

- Instead of recomputation loop, recompute priority at end of each timeslice
  - dyn_prio = nice + interactivity bonus (-5…5)
- Interactivity bonus depends on sleep_avg
  - measures time a process was blocked
- 2 priority arrays (“active” & “expired”) in each runqueue (Linux calls ready queues “runqueue”)



## Linux Scheduler (3)

```
struct prio_array {
        unsigned int nr_active;
        unsigned long bitmap[BITMAP_SIZE];
        struct list_head queue[MAX_PRIO];
};
typedef struct prio_array prio_array_t;

/* find the highest-priority ready thread */
 idx = sched_find_first_bit(array->bitmap);
 queue = array->queue + idx;
 next = list_entry(queue->next, task_t, run_list);
```

```
/* Per CPU runqueue */
struct runqueue {
 prio_array_t *active;
 prio_array_t *expired;
 prio_array_t arrays[2];
 …
}
```

- Finds highest-priority ready thread quickly
- Switching active & expired arrays at end of epoch is simple pointer swap (“O(1)” claim)



## Linux Timeslice Computation

- Linux scales *static* priority to timeslice
  - Nice [ -20 … 0 … 19 ] maps to [800ms … 100 ms … 5ms]
- Various tweaks:
  - “interactive processes” are reinserted into active array even after timeslice expires
    - Unless processes in expired array are starving
  - processes with long timeslices are round-robin’d with other of equal priority at sub-timeslice granularity



## Linux SMP Load Balancing

- Runqueue is per CPU
- Periodically, lengths of runqueues on different CPU is compared
  - Processes are migrated to balance load
- Migrating requires locks on both runqueues

```
static void double_rq_lock(
    runqueue_t *rq1,
    runqueue_t *rq2)
{
   if (rq1 == rq2) {
     spin_lock(&rq1->lock);
   } else {
      if (rq1 < rq2) {
        spin_lock(&rq1->lock);
        spin_lock(&rq2->lock);
      } else {
        spin_lock(&rq2->lock);
        spin_lock(&rq1->lock);
      }
   }
}
```



## Proportional Share Scheduling

- Aka “Fair-Share” Scheduling
- None of algorithms discussed so far give direct way of assigning CPU shares
  - E.g., give 30% of CPU to process A, 70% to process B
- Proportional Share algorithms assign “tickets” or “shares” to processes
  - Process get to use resource in proportion of their shares to total number of shares
- Lottery Scheduling, Stride Scheduling [Waldspurger 1995]



## Lottery Scheduling

- Idea: number tickets between 1…N
  - every process gets $p_i$ tickets according to importance
  - process 1 gets tickets [1… $p_1$-1]
  - process 2 gets tickets [$p_1$… $p_{1+}p_2$-1] and so on.
- Scheduling decision:
  - Hold a lottery and draw ticket, holder gets to run for next timeslice
- Nondeterministic algorithm

## Scheduling Summary

- OS must schedule all resources in a system
  - CPU, Disk, Network, etc.
- CPU Scheduling affects indirectly scheduling of other devices
- Goals:
  - Minimizing latency
  - Maximing throughput
  - Provide fairness
- In Practice: some theory, lots of tweaking



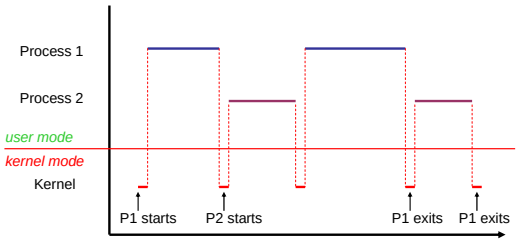


## Goals for Virtual Memory

- Virtualization
  - Maintain illusion that each process has entire memory to itself
  - Allow processes access to more memory than is really in the machine (or: sum of all memory used by all processes > physical memory)
- Protection
  - make sure there's no way for one process to access another process's data

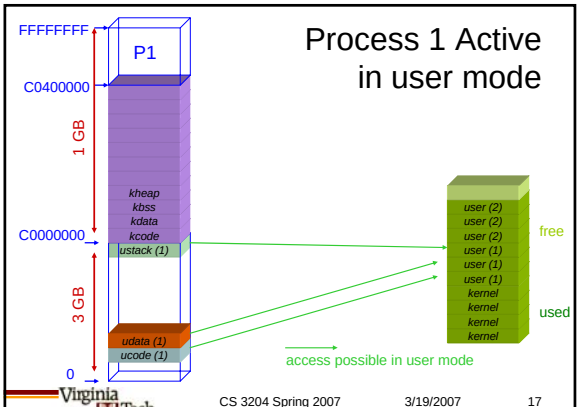


## Context Switching



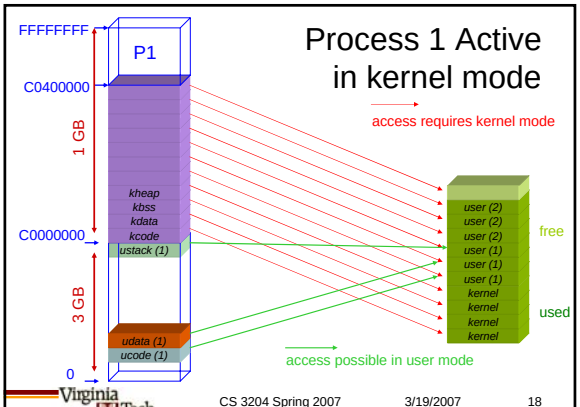


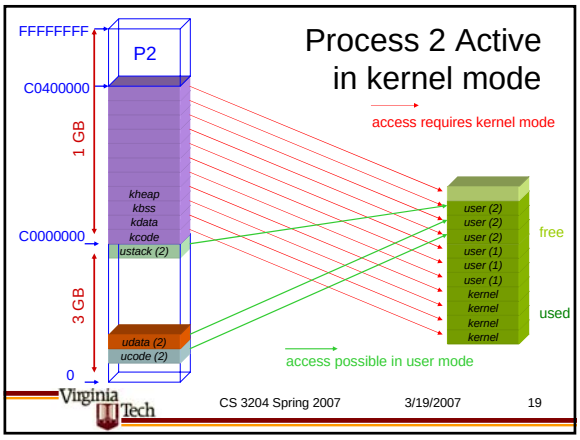

## Process 1 Active in user mode

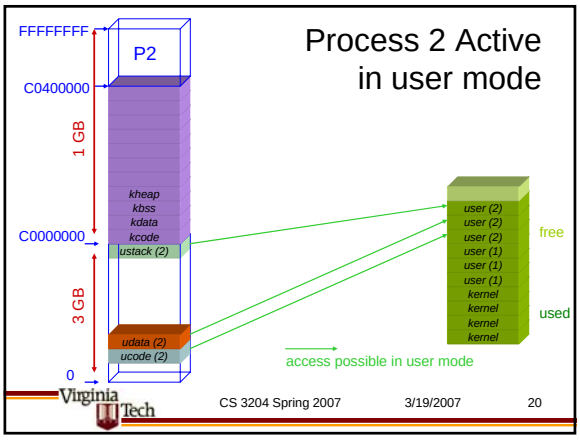





## Process 1 Active in kernel mode






## Page Tables

- How are the arrows in previous pictures represented?
  - Page Table: mathematical function "Trans"

Trans:
{ Process Ids } × { Virtual Addresses } × { user, kernel } × ℘({ read, write, execute })

→ { Physical Addresses } ∪ { INVALID }

- Typically have
  - Trans($p_i$, $v_a$, user, *) = Trans($p_i$, $v_a$, kernel, *)
    OR
    Trans($p_i$, $v_a$, user, *) = INVALID
  - User virtual addresses can be accessed in kernel mode



## Sharing Variations

- We get user-level sharing between processes $p_1$ and $p_2$ if
  - Trans($p_1$, $v_a$, user, *) = Trans($p_2$, $v_a$, user, *)
- Shared physical address doesn't need to be mapped at same virtual address, could be mapped at $v_a$ in $p_1$ and $v_b$ in $p_2$:
  - Trans($p_1$, $v_a$, user, *) = Trans($p_2$, $v_b$, user, *)
- Can also map with different permissions: say $p_1$ can read & write, $p_2$ can only read
  - Trans($p_1$, $v_a$, user, {read, write}) = Trans($p_2$, $v_b$, user, {read})
- In Pintos (and many OS) the kernel virtual address space is shared among all processes & mapped at the same address:
  - Trans($p_i$, $v_a$, kernel, *) = Trans($p_k$, $v_a$, kernel, *) for all processes $p_i$ and $p_k$ and $v_a$ in [0xC0000000, 0xFFFFFFFF]



## Per-Process Page Tables

- Can either keep track of all mappings in a single table, or can split information between tables
  - one for each process
  - mathematically: a projection onto a single process
- For each process $p_i$ define a function *PTrans*$_i$ as
  - PTrans$_i$ ($v_a$, *, *) = Trans($p_i$, $v_a$, user, *)
- Implementation: associate representation of this function with PCB, e.g. per-process hash table
  - Entries are called "page table entries" or PTEs



## Per-Process Page Tables (2)

- Common misconception
  - "User processes use 'user page table' and kernel uses 'kernel page table'" – as if those were two tables
- Not so: mode switch (interrupt, system call) does not change the page table that is used
  - It only activates those entries that require kernel mode within the current process's page table
- Consequence: kernel code also cannot access user addresses that aren't mapped

## Non-Resident Pages

- When implementing virtual memory, some of a process's pages can be swapped out
  - Or may not yet have been faulted in
- Need to record that in page table:

Trans (with paging):
{ Process Ids } × { Virtual Addresses } × { user, kernel } × $\wp$({ read, write, execute })

→ { Physical Addresses } ∪ { INVALID } ∪ { Some Location On Disk }