CS 439
Principles of Computer Systems
3: Processes and Scheduling

Where we are

Today:
- What is a process
- How does the kernel handle processes

From Program to Process

On previous slides we would talk about applications
- “Programs that a user want to execute”

What is a program?
- Sequence of instruction
- “Static data”

Side note: Compilation and Linking

Compilation
- Source file -> object file

Linking
- Object files (+ libraries) -> executable (*)

(*) or shared library

#include <stdio.h>
int main(void)
{
    printf("hello class!\n");
}

01010101110101010100...

From allensly.com

(*) or shared library

From allensly.com
ELF Format
- Header
- .text sections
  - Machine code
- .rodata
  - Read-only data
- .data
  - Initialized global vars

Loading
- Runtime memory image
- Created by the loader
  - Copies code and data from storage to memory
  - Jumps into the program code at a well-known entry point
    - C: & _start, defined in crt1.o (or crt0.o)
    - Sets up an initial stack frame
    - Calls main
    - (After main terminates)
    - Returns control to OS

Disclaimer: Simplification (missing VirtMem)

Process
- Process is a program during execution
  - Program = static (executable) file
  - Process = execution state + program
- You can have multiple processes from the same program

Tools to inspect process state:
- ps
- top
- htop

Process Life Cycle
Processes are always either executing, waiting to execute or waiting for an event(s) to occur

Ready to run, Waiting to be scheduled
Actually running
Waiting for event, e.g., IO
Process Control Block (PCB)

PCB (per process) contains info about a process, including its execution state (when not running).

- General Info
  - Process number
  - Process ID
  - Username/ID
  - Open File Info
  - Scheduling Info
- Execution State
  - Process state
  - Program Counter (PC)
  - Stack Pointer (SP)
  - Registers (e.g. GPRs)

Process Manipulation

- Basic process manipulation: creation, program loading, exiting, ...
- Unix
  - Creation and deletion: fork(), exec(), wait(), exit()
  - Process signaling: kill()
  - Process control: ptrace(), nice(), sleep()

fork

- fork()
  - Creates a new process and returns its PID
  - Process receives a copy of the process state
  - Both processes (parent and child) will resume with the next command
  - Child will “see” pid being 0.

```c
int pid = 0;
pid = fork();
if (pid == 0) {
    ChildProcess();
} else {
    ParentProcess();
}
```

wait

- wait(status)
  - Waits for a (any) child to terminate and sets status to the return value.
- waitpid(pid, status, options)
  - returns ECHILD if there are no “unwaited-for” children left
exec

- The `exec()` call allows a process to “load” a different executable and start execution at `_start` (which calls main)

- It allows the caller to specify the number of arguments (argc) and the string argument array (argv)

- If the call is successful
  - it remains the same process ...
  - but it runs a different program
  - code, stack, and heap are overwritten

`man exec`

exit

- Finish execution of the current process

- This system call:
  - takes the “result” of the program as an argument (it saves it in case the parent <shell?> asks how the child exited)
  - closes all open files, connections, etc.
  - deallocates memory
  - deallocates most of the OS structures supporting the process
  - checks if parent is alive:
    - if so, it holds the result value until parent requests it; in this case, process does not really die, but it enters the zombie/defunct state
    - if not, it deallocates all data structures, the process is dead
  - cleans up all waiting zombies

Scheduling

- Long term
  - What to bring into the ready queue

- Medium term
  - Swapping processes out to storage. Later in this course

- Short term
  - Choosing which process to run next, CPU scheduling

Dispatcher

- How does the kernel acquire control from the process
  - Process blocks (IO, event)
  - System call
  - Process is interrupted (if scheduler is preemptive)
    - Interrupt
    - Remember Timesharing Systems...

- How to switch back to the scheduled process?
  - Dispatcher!
  - Switches context
  - Enter user mode
  - Jump into scheduled process

- How to choose the next process?
  - Scheduler policy
Scheduling Algorithms

- Some notation:
  - Process = Task = Job
  - CPU Utilization
  - Latency
    - Response time: Time between the arrival of a task and the (user-observed) completion of execution
    - Wait time: Time between the arrival of a task and the beginning of execution
  - Throughput
    - Rate of tasks completed per time unit
  - Fairness
    - Equality in the number and timeliness of resources given to a specific task

First-come, First-served

- FIFO Queue
  - Pick job from head of queue, run to completion

- Example:
  - Arrival order: P1, P2
    - avg RT = (1 + 6 + 8) / 3 = 5
  - Arrival order: P2, P1
    - avg RT = (1 + 3 + 8) / 3 = 4

Shortest Job First

- Goal: Minimize response time

- Example:
  - Arrival order: P6, P1, P2
    - avg RT = (1 + 2 + 4 + 9) / 4 = 4
  - Arrival order: P1, P4, P3, P2
    - avg RT = (1 + 2 + 4 + 9) / 4 = 4

- Provably optimal under these conditions, but...
  - We assume that all tasks are submitted at the same time
  - What if P3 arrives after P2 has started and P4 after P3...
Shortest Remaining Time

- Idea: Preempt when a shorter process arrives
  
  Different scenario:
  ```
  P1 arrives at t=0
  P2 arrives at t=1
  P3 arrives at t=2
  P4 arrives at t=5
  ```

- Non-preemptive (SJF):
  ```
  P1  P2  P3  P4  P1
  0   1   2   3   4   5   6   7   8
  avg RT = (1+5+7+2)/4=3.75
  ```

- Preemptive (SRT):
  ```
  P1  P2  P3  P2  P4  P2  P2
  0   1   2   3   4   5   6   7   8
  avg RT = (1+8+2+1)/4=3
  ```

- Priority
  - Alternative to predicting job time
  - Assign a priority to each job
  - Schedule jobs in order of priority
    - If priority = predicted job length, we get a SJF/SRT scheduler
  - Avoiding starvation
    - “Aging”: increase the priority of a job over time

- Problems:
  - Knowing / predicting job length
  - Predict “CPU burst”
    - Common approach: next CPU burst is predicted as exponential average over past CPU bursts.
    - In practice: process is IO bound or CPU bound for a longer periods of time
  - Starvation
    - Long jobs might never execute!
    - This scheduler is not fair!
      - Longest jobs are executed as slowly as possible

- Round Robin
  - Goal: Fix starvation, maximize fairness
  - Our jobs are too well-behaved
    - In practice job length is not always a multiple of the time slice
  - Picking the right quantum size
    - Wait time vs. context switch overhead
No Silver Bullet

- SJF/SRT is optimal for minimizing wait time but it can suffer from starvation
- Priorities can help with starvation but increase the complexity and suffer from different problems
- RR provides good fairness, throughput but suboptimal wait time and context switching overhead can become an issue. It cannot deal with priorities.
- The optimal scheduler depends on the characteristics of the workload
  - Think of interactive processes
  - Is there a better balance?

Multilevel Feedback Queues

- Multiple queues for different types of jobs
  - Queues have different priorities
  - Use Round Robin within queues
  - Scheduler varies the priority based on the observed behavior of a job
    - High priority for interactive (bursty) processes
    - Low priority for jobs using the CPU intensively over longer periods of time
- Different time slice length per priority
  - Usually exponential
  - => Learning, Feedback-driven

Linux: niceness

- Each process has a nice value
  - Default: 0, Range: -20 (highest), 19 (lowest)
- Users can lower the nice value of their process
- Root user can lower and raise the nice value.
- Nice value is used to determine priority of a process
  - Relative to the niceness of the other processes
- Determines the static priority
  - Priority 100 => Niceness = -20 => Q=800 ms
  - Priority 120 => Niceness = 0 => Q=100 ms
  - Priority 139 => Niceness = +20 => Q=5ms
Scheduling on Multiprocessor Systems

- Challenges for a multiprocessor scheduler
  - Balancing CPU Load
    - What is better, all equally utilized or some heavy, some idle?
  - NUMA memory
  - Locality, Cache effects
    - Bringing computation close to data
      - Linux: Scheduler Domains
  - Cost of task migration
  - Scheduler data structures like run queue
- Different people, different answers:
  - Gang scheduling, Space-time multiplexing, Multikernel, ...