CS-202 Week 2

Notes

Lecture 1:

Isolation and Protection

Focus:

- Limited direct execution
- Protecting processes from each other

Protection

- Protect applications from other application's code
  - ensures reliability, security and privacy
- Protect the os from applications
- Protect applications from unfavorable resource utilization
  OS is fully trusted, whilst other applications are untrusted

OS allows processes to run with **restricted rights**
Enforces mechanisms that allow processes to run with **potentially dangerous operations disabled**
OS enforces time-sharing even when the process does not cooperate (example infinite loop)

CPU virtualization: efficient process execution

Goal: give each process the illusion of an exclusive CPU access
Reality: CPU is shared among all processes

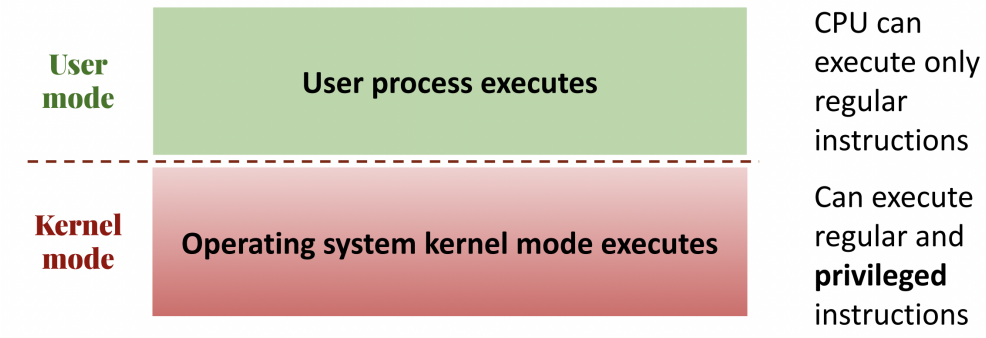
CPU uses time sharing

Limited Direct Execution

Enables programs to execute as fast as possible with some restrictions.

(**Direct execution:** run programs directly on CPU without any restrictions.)

Limited direct execution with dual mode:



Privileged instructions:

Examples:

- Access I/O devices
- Change privilege levels
- Enable or disable interrupts
- Change the MMU register that controls page table (mov %cr3 on x86-64)
- ...

When CPU tries to execute privileged instruction from user mode:

- The instruction does not execute
- The instruction traps (#General protection fault on x86)
- The operating system takes control

System calls

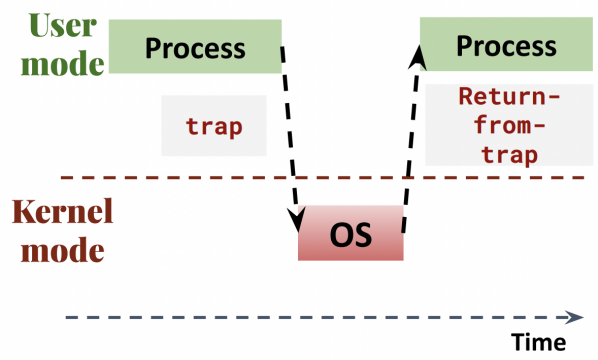
Processes can request OS services through the **system call API**
System calls transfer execution to the OS, meanwhile execution of the process is suspended

Processes execute special **trap instructions** to execute system calls (slide 28):

- CPU jumps into kernel mode and changes privilege level at the same time
- Now privileged operations can be performed

Trap is a signal that instructs OS to perform some functionality immediately

Slide 32 (important)



Traps

Traps also referred to as exceptions.
Exceptions produced by CPU while executing instructions.
Exceptions are synchronous: invoked by CPU only after termination the invocation of an instruction.

OS configures hardware at boot time (slide 36)
Requesting OS services using system call numbers (slide 37)

Interrupts

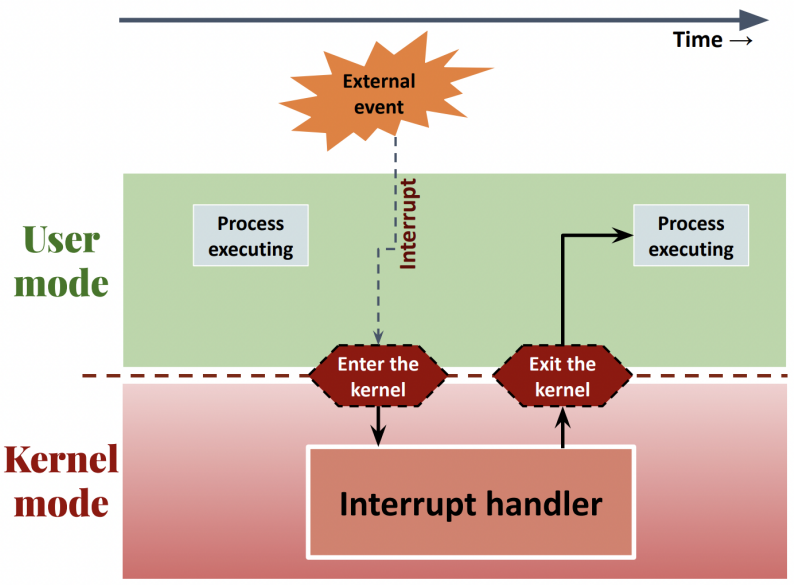
Hardware provides signaling mechanism for the OS to regain control.
Interrupts are asynchronous signals to the CPU that some external event has occurred and requires attention.
Example: Timer interrupt, IO interrupt, inter-processor interrupt

OS configures the **timer interrupt** to regain control
Interrupt is fired every certain milliseconds (every 10 ms in Linux kernel)
With every interrupt, CPU switches from user to kernel mode (context switch) so OS can choose which process to execute

Handling

When there is an interrupt:

- The CPU detects it
- Suspends the user process and switches to kernel mode
- Executes the appropriate interrupt handling code
- Restores the user process



Concurrent interrupts

Hardware provides instructions to temporarily delay the delivery of an interrupt (**disable interrupt**), and enable the again when safe

Interrupts are disabled when an interrupt handler is running

Lecture 2: The os scheduler

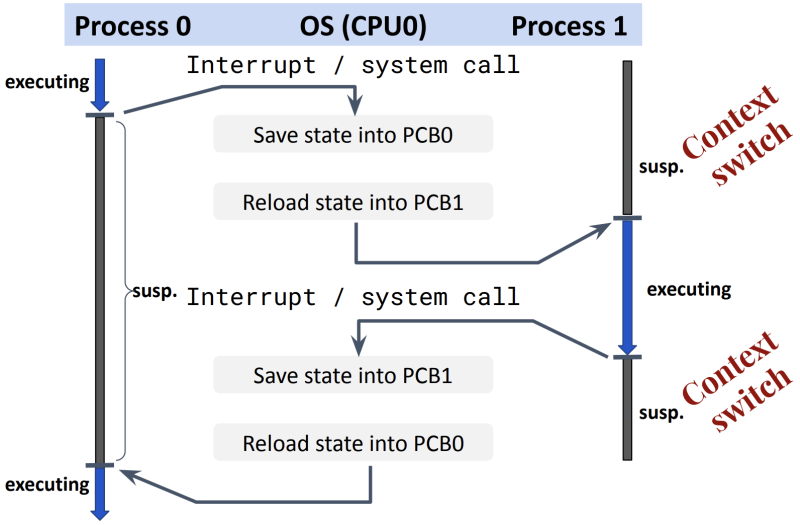
Scheduling mechanisms

Context switching

- OS in kernel mode and cannot return to same process
- OS does not want to run same process
- -> OS performs a **context switch**

A mechanism that allows the OS to store the current process state and switch to another previously stored context

Context of process is represented in the process control block (PCB)
OS maintains PCB for each process



PCB: Process control block

Preemption: OS sets a timer before scheduling a process. Hardware generates interrupt after timer expires

Scheduling metrics

Utilization

What fraction of the time is the CPU executing a job
Goal: Maximize CPU utilization as much as possible

Turnaround time

Total time from a job arrival to job completion
Goal: Minimize the total time as much as possible

Response time

How long long a job takes to be scheduled
Goal: Minimize total time as much as possible

The idle process

Low priority process that is scheduled if no other process is ready.
What process to schedule if all processes are blocked ? The idle process (runs in kernel mode).
Idle mode issues special instructions to save power.

CPU-only Scheduling Policies (not used)

Policies that optimise turnaround time

FIFO (First in first out)

Not a viable strategy. Long jobs delay short jobs (example: slide 29)
Only works is processes have the same amount of execution time.
convoy effect (see slides)

SJF (Shortest job first)

Schedule shortest job first and run to completion.
Reduces average turnaround time.

Long running jobs cannot be interrupted.
Still subject to convoy effect if longer process arrives before others.

STCF (Shortest Time to Completion First)

FIFO and SJF are non-preemptive: only switch to other job when current is finished
STCF extends the SJF by adding preemption
Any time a new job enters system, STCF Scheduler determines which of remaining job has shortest completion time.
STCF then schedules the shortest job first.
Response is not great...

Policies that optimise response time

RR (Round Robin)

Runs jobs for a fixed interval, performing a context switch after said time if other jobs are available
RR policy is fair, evenly divides CPU time among ready processes on a small time scale.
Higher average turnover but lower response time.

CPU & I/O scheduling

IO awareness

Almost ever program performs IO of some kind
IO is usually slow: Read/write requests can take milliseconds to complete
Schedulers need to incorporate CPU and IO by blocking processes.

MLFQ (Multi-level Feedback Queue)

Goal: Supporting general purpose scheduling

First: MLFQ optimises turnaround time:

- important for batch processes

Then: minimizes response time for better interactivity

Insight: Use past behavior as predictor for future behavior

Approach: Multiple levels of round robin

Each level has higher priority and preempts lower level
Process at a higher level will always be scheduled first
High levels have short time slices, lower levels run longer

Rule 1: if priority(A) > priority(B) then A runs
Rule 2: if priority(A) == priority(B) then A & B run in RR
Rule 3: Processes start at top priority
Rule 4: If a process uses its total time slice, the scheduler lowers its priority
Rule 5: Periodically moves all jobs to topmost queue (**priority boosting**)

priority boosting: jobs moved up to highest priority queue

Boosting avoids starvation on long running jobs