Operating Systems (02340123)
Summary - Spring 2025

Razi & Yara

June 29, 2025

Contents

Part I

Lectures & Tutorials

Topic 1: Introduction

Operating System (OS) An Operating System's job is:

- Coordinate the execution of all SW, mainly user apps.
- Provide various common services needed by users & apps.
- An OS of a physical server controls its physical devices, e.g. CPU, memory, disks, etc.
- An OS of a virtual server only *believes* it does. There's another OS underneath, called **hypervisor** which fakes it.

Using an OS allows us to take advantage of ***"virtualization"***:

- **Server Consolidation:** Run multiple servers on one physical server. This is allows for better resource utilization, smaller spaces, and less power consumption.
- **Disentangling SW from HW:** allows for backing up/restoring, live migration, and HW upgrade. This give us the advantage of easier provisioning of new (virtual) servers = "virtual machines", and easier OS-level development and testing.

Most importantly, an OS is **reactive**, "event-driven" system, which means it waits for events to happen and then reacts to them. This is in contrast to typical programs which run from start to end without waiting for external events to occur to invoke them.

	Typical Programs	**OS**
What does it typically do?	Get some input, do some processing, produce output, terminate	Waits & reacts to "events"
Structure	Has a `main` function, which is (more or less) the entry point	No `main`; multiple entry points, one per event
Termination	End of `main`	Power shutdown
Typical goal	$\sim$ Finish as soon as possible	Handle events as quickly as possible $\Rightarrow$ more time for apps to run

Event Synchronousation OS events can be classified into two:

- **Asynchronous interrupts:** keyboard, mouse, network, disk, etc. These are events that can happen at any time and the OS must be ready to handle them.
- **Synchronous:** system calls, divide by zero, page faults, etc. These are events that happen as a result of the program's execution and the OS must handle them immediately.

Multiplexing Multiplexing is the ability of an OS to share a single resource (e.g. CPU, memory, disk) among multiple processes or threads. This allows for better utilization of resources and enables multiple applications to run concurrently. *Multiprogramming means multiplexing the CPU recourse.

Notable services provided by an OS:

1. **Isolation:** Allow multiple processes to coexist using the same resources without stepping on each other's toes. Usually achieved by multiplexing the CPU, memory, and other resource done by the OS. However, some physical resources know how to multiplex themselves, e.g. network cards, sometimes called *"self-virtualizing devices"*.
2. **Abstraction:** Provides convenience & portability by:
   - offering more meaningful, higher-level interfaces
   - hiding HW details, making interaction wiht HW easier.

Topic 2: Processes & Signals

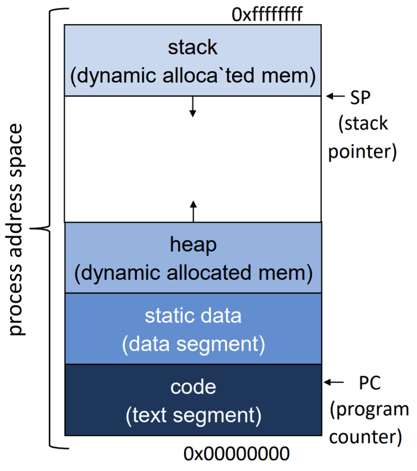
Processes

Each process is an instance of a program in execution, which includes:

- **Program code:** The actual code of the program.
- **Process state:** The current state of the process, including the program counter, registers, and memory management information.
- **Process control block (PCB):** A data structure used by the OS to manage the process, containing information such as process ID, process state, CPU registers, memory management information, and I/O status information.

  A process doesn't have direct access to its PCB, it is managed by the OS (kernel space), i.e. needs privilege level 0 (kernel mode) to access it.

  Each PCB contains: `real_parent`, `parent`, `children`, `siblings`,...
  Each process has a `task_struct current` pointer to its PCB.
- **Process ID (PID):** A unique identifier assigned to each process by the OS. The PID is used by the OS to manage the process and is used in system calls to refer to the process.



Process States A process can be in one of the following states:

- **Running:** The process is currently being executed by the CPU.
- **Ready:** The process is ready to be executed but is waiting for the CPU to become available.

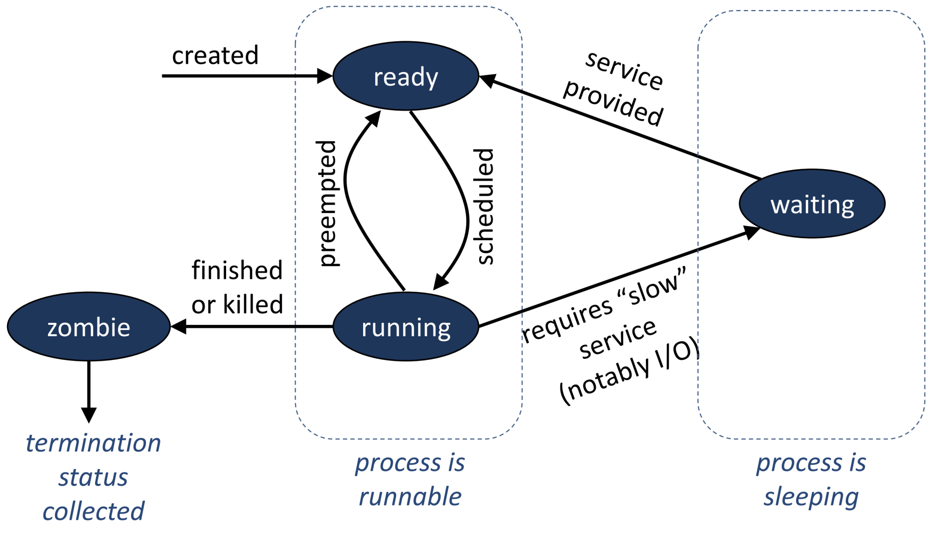


Figure 1: Process States

- **Waiting:** The process is waiting for an event to occur, such as I/O completion or a signal.
- **Zombie:** The process has terminated but its PCB is still in the system, waiting for the parent process to read its exit status. In this state, the process has released almost all of its resources, but the PCB is still in the system.

As we saw in "ATAM", each process can only access a certain set of utilities and functions, those who require privilege level 3 (user mode). So to access the OS services, a process must use **system calls** which are functions provided by the OS that allow processes to request services from the OS. System calls are typically implemented in the OS kernel and provide a controlled interface for processes to interact with the OS.

Each *syscall*, in case of an error, will change the `errno` variable to indicate the error type. The `errno` variable is a global variable that is set by system calls and some library functions in the event of an error to indicate what went wrong. It is defined in the header file `errno.h`. **Note:** `errno` is not reset to 0 after a successful syscall, so it must be checked immediately after the syscall, and be reset before usage if need be (if there is not any other way to make sure there is an error indeed).
i.e. `errno = <Number of Last Syscall Error>;`

As noted above, each process must be `wait()`ed for by its parent process to be able to release its PCB and resources. This is done by the `wait()` syscall, which suspends the calling process until one of its children terminates. In case the parent process terminates before the child, the child process becomes an orphan process and is adopted by the init process (PID 1), which will then wait for it to terminate and release its resources.

Process Management The OS offers various system calls to manage processes, including: (More details in the functions reference)

- `fork()`: Creates a new process by duplicating the calling process. The new process is called the child process, and the calling process is called the parent process.
- `exec()`: Replaces the current process image with a new process image, effectively running a different program in the same process.
- `wait()`: Suspends the calling process until one of its children terminates.
- `exit()`: Terminates the calling process and releases its resources.
- `getpid()`: Returns the process ID of the calling process.
- `getppid()`: Returns the process ID of the parent process.
- `kill()`: Sends a signal to a process, which can be used to terminate or suspend the process.

Parent Vs. Real Parent Process The real parent process is the one that created the current process using `fork()`, or the one that adopted it in case the real parent terminated before the child.

The parent process is the one *tracing* the current process, e.g. using `ptrace()`. The parent process is the one that will receive signals from the current process, e.g. `SIGCHLD` when the current process terminates.

In most cases, the parent process is the real parent process, but it can be different in some cases, e.g. when a process is being traced by a debugger.

Daemon Processes A daemon process is a background process not controlled by the user. To run a process as a daemon use `nohup <command> &`.

Daemon names usually end with the letter "d", e.g. `sshd` (SSH daemon), `httpd` (HTTP daemon), etc.

Signals

Signals Signals are "notifications" sent to a process to asynchronously notify it that some event has occurred.

* Receiving a signal only occurs then returning from kernel mode, which in turn invokes the corresponding signal handler.

** Default signal handling actions: Either die or ignore

*** In case of several signals from different types, they will be handled by the order of their definition in the signals register.

Each signal has a name, a number, and a default action. All but 3 signals can be blocked, i.e. ignored until the process is ready to handle them. The 3 signals that cannot be blocked are:

- `SIGKILL`: Used to forcefully terminate a process. (Process becomes a zombie)
- `SIGSTOP`: Used to suspend the receiving process. (Make it sleep) The signals is sent when the user presses Ctrl+Z in the terminal. Note: In truth Ctrl+Z sends the `SIGTSTP` signal, however, we don't learn about the differences between the two signals in this course.
- `SIGCONT`: Used to resume a suspended process, usually sent after a `SIGSTOP` signal. The handler for this signal can be customized but it will always resume the process.

`SIGSTOP` and `SIGCONT` are useful for debugging purposes, allowing the user to pause and resume the execution of a process.

Signal Handling A process can define a custom signal handler for a specific signal using the `signal()` or `sigaction()` preferred system calls. To ignore a signal, the process can set its handler to `SIG_IGN`. To restore the default action for a signal, the process can set its handler to `SIG_DFLT`.

Signal Masking A process can block signals using the `sigprocmask()` system call, which allows the process to specify a set of signals to block. This is allows the process to overcome *Race Conditions* resulted from the asynchronous nature of signals.

This is achieved by maintaining a set of currently blocked signals & a set of masked signals which is saved in the PCB.

`Blocked Signals` = mask array.

`Pending Signals` = signals that were sent to the process while it was blocked, and will be handled when the process unblocks them.

Common Signals the following are some of the most common signals:

1. `SIGSEGV, SIGBUS, SIGIILL, SIGFPE`: These are driven by the associated (HW) interrupts - The OS gets the associated interrupt, then the OS interrupt handler sees to it that the misbehaving process gets the associated signal, lastly the signaly handler is invoked.

   - `SIGSEGV`: Segmentation violation (illegal memory reference, e.g., outside an array).
   - `SIGBUS`: Dereference invalid address (null/misaligned, assume it's like SEGV).
   - `SIGILL`: Illegal instruction (trying to invoke privileged instruction).
   - `SIGFPE`: Floating-point exception (despite the name, *all* arithmetic errors, not just floating point. e.g., division by zero).

2. `SIGCHLD`: Parent (not real parent) get it whenver `fork()`ed child terminates or is `SIGSTOP`-ed.

3. `SIGALRM`: Get a signal after some specified time, can be set using the `alarm()` & `setitimer()` system calls.

4. `SIGTRAP`: When debugging/single-stepping a process, the debugger can set a breakpoint in the code, which will cause the process to receive a `SIGTRAP` signal when it reaches that point.

5. `SIGUSR1, SIGUSR2`: User-defined signals, user can decide the meaning of these signals and their handlers.

6. `SIGPIPE`: Write to pipe with no readers.

7. `SIGINT`: Sent when the user presses Ctrl+C in the terminal. The default action is to terminate the process, but it can be customized.

8. `SIGXCPU`: Delivered when a process used up more CPU than its soft-limit allows: soft-/hard limits are set using the `setrlimit()` system call. Soft-limits warn the process its about to exceed the hard-limit, Exceeding the hard-limit will cause `SIGKILL` to be sent to the process.

9. `SIGIO`: Can configure file descriptors such that a signal will be delivered whenever some I/O is ready.

   Typically makes sense when also configuring the file descriptor to be *non-blocking*, e.g., when `read()`ing from a non-blocking file descriptor, the system call immediately returns to user if there's currently nothing to read. In this case, `errno` will be set to `EAGAIN=EWOULDBLOCK`.

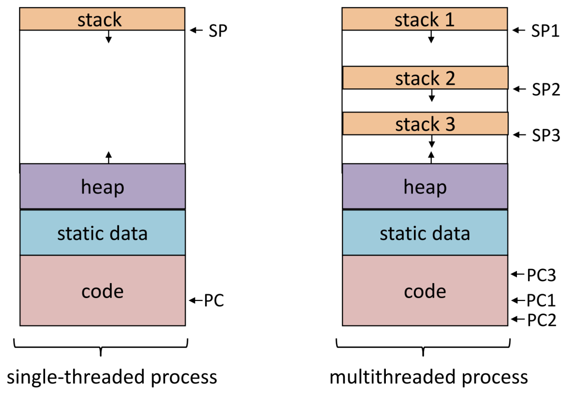
Signals Vs. Interrupts

	interrupts	**signals**
Who triggers them? Who defines their meaning?	Hardware: CPU cores (sync) & other devices (async)	Software (OS), HW is unaware
Who handles them? Who (un)blocks them?	OS	processes
When do they occur?	Both synchronously & asynchronously	Likewise, but, technically, invoked when returning from kernel to user

Topic 3: IPC - Inter-Process Communication

Threads & IPC

Multiprocessing Using several CPU cores (=processors) for running a single job to solve a single "problem".



Threads A process can have multiple threads, which share (nearly) everything, including the process's memory space, file descriptors, heap, static data, code segment and more.

However, each thread has its own stack, registers, and program counter. But they can still access each other's stack since they share the same memory space.

Note that *Global/Static Variables & Dynamically Allocated Memory* are shared between threads since they are stored in the Static Data Segment and the Heap, respectively, which are shared between all threads of a process. However, *Local Variables* are not shared between threads since they are stored in the stack, which is unique to each thread.

	Unique to Process	**Unique to pthread**
Registers (notably PC)	Y	Y
Execution stack	Y	Y
Memory address space	Y	N
Open files	Y	N
Per-open-file position (=offset)	Y	N
Working directory	Y	N
User/group credentials	Y	N
Signal handling	Y	N

OpenMP Open Multi-Processing (OpenMP) consists of a set of compiler *'pragma'* directives that allows the compiler to generate multi-threaded code for parallel execution.

```
#pragma omp parallel for
for (i = 0; i < N; i++) {
        arr[i] = 2*i;
}
```

Pthreads POSIX threads (pthreads) is a standard for multi-threading in C/C++. It provides a set of functions to create and manage threads, as well as to synchronize them. (More details in the functions reference)

File Descriptors A non-negative integer representing an I/O "channel" on some device. File descriptors are saved in the process's `PCB` inside the `FDT` (File Descriptor Table), which is an array of pointers to *file objects*, each representing an open file or device. So in fact, an FD is an index to a kernel array of channels.

Processes don't share `PCB` nor `FDT` but they can share file descriptors, i.e. two processes can have the same file descriptor pointing to the same file object, which allows them to share the same open file or device. Upon forking, the child process inherits a copy of the parent's `FDT`. Note: Threads do share the same `FDT`.

Pipes A pipe is a unidirectional communication channel between two processes, allowing one process to send data to another. Pipes are a pair of two file descriptors `int pipe_fd[2]`. Each integer is a handle to a kernel communication object, which is a buffer that holds the data being sent between the two processes.

- `pipe_fd[0]` = read side of the communication channel.
- `pipe_fd[1]` = write side of the communication channel.
- Everything written via `pipe_fd[1]` can be read via `pipe_fd[0]`.
- Blocking: If the read side is empty, the read operation will block until data is written to the write side. If the write side is full, the write operation will block until space is available.
- `SIGPIPE`: Writing to a pipe whose read end is `close()`d will result in a `SIGPIPE` signal.

Multi-tasking	**Multi-programming**	**Multi-processing**
Having multiple processes *time slice* on the same CPU core.	Having *multiple jobs* in the system (either on the same core or on different cores). i.e. the existence of multiple processes in the system, regardless of whether they are running on the same core or not.	Using *multiple processors (CPU cores) for the same job* in parallel. i.e. creating multiple threads to run on different cores for the same process.

Intro. Context Switching & Caching

Context Switching is the process of saving the state of a currently running process and restoring the state of another process to allow it to run. This is done by the OS kernel and is necessary for multitasking, allowing multiple processes to share the CPU.

- **Context** = the state of a process, including its registers, program counter, stack pointer, and memory management information.
- **Context Switch** = the process of saving the context of the currently running process and restoring the context of another process.

Context Switch Overhead consists of two components:

- **Direct Overhead:** The measurable time it takes to perform the context switch, which includes saving the current process's state and restoring the next process's state.
- **Indirect Overhead:** The time it takes for the CPU to cache the new process's data, which can be significant if the new process's data is not already in the CPU cache.

Cashing The CPU cache is a small, fast memory that stores frequently accessed data to speed up access times. The Cache allows the CPU access to a fast memory that is closer to the CPU than the main memory (DRAM), and a larger one than the CPU registers. The cache works because of the *Principle of Locality*:

- **Temporal Locality:** If at one point in time a particular memory location is referenced, then it is likely that the same location will be referenced again soon.
- **Spatial Locality:** If a particular memory location is referenced at a particular time, then it is likely that nearby memory locations will be referenced soon.

Copy-on-Write (COW) The `fork()` system call creates a copy of the address space of the parent, but:

- It only creates a *logical* copy.
- There is no physical duplication of the memory pages, until we really need to write to them (from either process). And even then, only the page that is being written to is duplicated. (More details in the virutal memory lectures)

User Level Threads (ULTs) are threads that are managed by the user-level library, rather than the OS kernel. ULTs are not visible to the OS, which means that the OS does not know about them and does not schedule them. This allows for faster context switching between ULTs, but it also means that the OS cannot take advantage of multiple CPU cores to run ULTs in parallel.

The usage of ULTs is mainly for concurrent programming, where we want multiple multiple tasks to progress in parallel without the overhead of kernel-level threads.

Topic 4: Scheduling

Batch (Non-Preemptive) Scheduling

Supercomputers Supercomputers are compromised of multiple nodes, each with multiple CPU cores, and are used for running large-scale computations. Users submit **batch jobs** to the supercomputer with a specified time limit and size. Terminology:

- **Size** = the number of CPU cores to use for the job. Jobs are said to be *wide/big* or *narrow/small*.
- **Runtime** = the time limit for the job to run. Jobs are said to be *short* or *long*.

Metrics for Performance Evaluation When evaluating the performance of a batch scheduling algorithm, we use the following metrics:

- **Average Wait Time:** The wait time of a job is the interval between the time the job is submitted to the time the job starts to run. i.e.

$$\text{waitTime} = \text{startTime} - \text{submitTime}$$

- **Average Response Time:** The response time of a job is the interval between the time the job is submitted to the time the job is terminated. i.e.

$$\text{responseTime} = \text{terminateTime} - \text{submitTime}$$

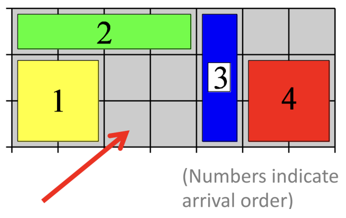
  Note: Average wait time and response time differ only by a constant factor, which is the job's average runtime.

- **The Slowdown / Expansion Factor:** The ratio between a job's response time and its runtime. i.e.

$$\begin{aligned}\text{slowdown} &= \frac{\text{responseTime}}{\text{runtime}} \\ &= \frac{(\text{waitTime} + \text{runtime})}{\text{runtime}} \\ &= 1 + \frac{\text{waitTime}}{\text{runtime}}\end{aligned}$$

- **Utilization:** The percentage of time the resource (CPU) is busy.
- **Throughput:** How much work is done in one time unit.

Batch Scheduling Algorithms



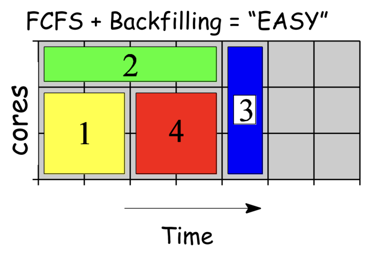
First-Come, First-Served (FCFS) Jobs are scheduled by their arrival time. If there are enough free cores, a newly arriving job starts to run immediately, otherwise it wait, sorted by arrival time, until enough cores are available.

- **Pros:** Simple to implement (FIFO Queue), fair to all jobs.
- **Cons:** Creates *fragmenation*, small/short jobs might wait a long time.

EASY Scheduling (= FCFS + backfilling) Backfilling optimization: A short waiting job can jump over the head of the wait queue (i.e. start at an earlier time) provided that it doesn't delay the job head of the FCFS queue.

The algorithm: whenever a job arrives or terminates, try to start the job head of the FCFS wait queue. Then, iterate over the rest of teh waiting jobs (in FCFS order) and try to backfill them.

- **Pros:** Better utilization (less fragmentation), short jobs have a better chance of running sooner.
- **Cons:** Must know runtimes in advance.



Shortest Job First (SJF) Instead of ordering jobs by their arrival time, we order them by their (typically estimated) runtimes.

- **Pros:** Optimal in terms of performance (min. avg. wait time).
- **Cons:** Unfair as it may cause **starvation** of long jobs. starvation = can theorically wait forever

---

Convoy Effect Slowing down all (possibly short) processes due to currently servicing a very long process. *FCFS* suffers from this effect, as does *EASY* scheduling but to a lesser extent. *SJF* without assumptions also suffers from this effect but to an even lesser extent.

Optimality of SJF As mentioned above, SJF is optimal in terms of performance, in particular, it minimizes the average wait time. **Claim:**
Given:

1. A 1-core system where all jobs are serial.
2. All process arrive together.
3. Their runtimes are known in advance.

Then: *The average wait time of SJF is equal to or less than the average wait time of any other batch scheduling order.*

Fairer Varients of SJF Motivation: disallow job starvation.

- **Shortest-Job Backfilled First (SJBF):** Exactly like EASY in terms of servicing the head of the wait queue in FCFS order (and not allowing anyone to delay it), but the *backfilling* traversal is done in SJF order, i.e. the next job to be backfilled is the one with the shortest estimated runtime.
- **Largest eXpansion Factor (LXF):** LXF is similar to EASY, but instead of ordering the wait queue in FCFS, it orders jobs based on their current slowdown (greater slowdown = higher priority).

  On every job arrival or termination, the expansion factors are recalculated and the wait queue is resorted, so that the job with the largest expansion factor is always at the head of the queue. Both the head of the queue and the backfilled jobs are serviced in LXF order.

Preemptive Schedulers

Preemption *is the act of suspending one job (process) in favor of another even though it is not finished yet.*

Why do we need preemption? Preemption is necessary for **responsiveness** and when the runtime of jobs vary or unknown in advance.

Quantum *is the maximum amount of time a process is allowed to run before it is preempted.* Quantum is typically milliseconds to 10s of milliseconds, and is *often set per-process*. Usually a CPU-bound process gets long quanta while an I/O-bound process gets short quanta with higher priority.

Performance Metrics for Preemptive Schedulers we use the following:

- **Average Wait Time:** As before, the wait time of a job is the interval between the time the job **is submitted** to the time the job **starts to run**. *Note* that the wait time does not include the preemption wait times (i.e. the time the job is waiting for its turn to run again after being preempted).
- **Response Time (=Turnaround Time):** Like before, the response time is the time from process submission to process completion. *Note* that with preemption we get:

$$\text{responseTime} \geq \text{waitTime} + \text{runtime}$$

  due to the preemption wait times and **Context Switches** cost.
- **Overhead:** How long a context switch takes, and how often context switches happen. (Should be minimized)
- **Utilization & Throughput:** Same as before, but we may want to account for context switch overhead.
- **Makespan Time:** The time it takes for the system to finish all jobs in the system.

Round Robin (RR) Scheduling *Processes are arranged in a cyclic read-queue.* The algorithm works in the following steps:

1. The head process runs until its quantum is exhausted.
2. The head process is then preempted (suspended) and moved to the tail of the queue.
3. The scheduler resumes the next process in the circular list.
4. When we've cycled through all processes in the run-list (and we reach the head process again), we say that the current **"epoch"** is over, and the next epoch begins.

Note: RR Requires a timer interrupt; Typically it's a periodic interrupt (fires every few milliseconds) and upon receiving the interrupt, the OS checks if its time to preempt. **Note 2:** For small enough quantum, it's like everyone of the N processes advances in $1/N$ of the speed of the core called sometimes virtual time. With huge quantum infinity, RR becomes FCFS.

Gang Scheduling Think of it as RR for *parallel* systems. It works as follows:

- Time is divided to slots (seconds or minutes).
- Every job has a **"native"** time slot.
- Algorithm attempts to fill holes in time slots by assigning to them jobs from other native slots (called **"alternative"** slots).
- Algorithm attempts to minimize slot number using **slot unification** when possible.
- Rarely used in practice, if lots of memory is swapped out upon context switch, context switch overhead is too high.

Detailed explanation about alternative slots and slot unification:

Alternative Slots To further enhance resource utilization, the gang scheduling algorithm attempts to fill holes in time slots. A job has a "native" time slot where it is primarily scheduled to run. However, if there are idle processors in other time slots (termed "alternative" slots), the scheduler can assign jobs from their native slots to run in these otherwise unused resources. This allows jobs to get more processing time without interfering with the primary scheduling of other jobs.

Slot Unification This technique aims to minimize the total number of active time slots. If, after some jobs have completed, the remaining jobs in two different time slots can fit into a single slot without conflict, the scheduler can merge them. By unifying slots, the overall cycle time of the matrix is reduced, which can lead to shorter job turnaround times and improved system throughput.

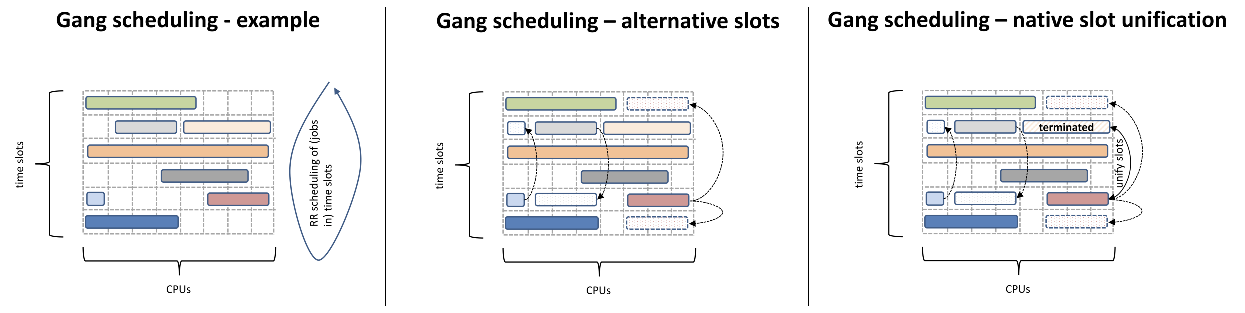

Gang scheduling - example
time slots
CPUs
RR scheduling of (jobs in) time slots
Gang scheduling – alternative slots
time slots
CPUs
Gang scheduling – native slot unification
time slots
terminated
unify slots
CPUs

Figure 2: Gang Scheduling

Batching vs. Preemption **Assume:** (1) A single core system (2) All jobs arrive together (3) Context switch price = 0. **Then:** There exists a non-preemptive algorithm such that:

$$\text{avgResponseTime(non-preemptive)} \leq \text{avgResponseTime(preemptive)}$$

As a result, SJF is also optimal relative to preemptive scheduling (if our assumptions hold).

$$\text{avgResponseTime(SJF)} \leq \text{avgResponseTime(batch or preemptive scheduler)}$$

Connection Between RR & SJF **Assume:** (1) A single core system (2) All jobs arrive together (and only use CPU, no I/O) (3) Quantum is uniform + no context switch overhead.
Then:

$$\text{avgResponse(RR)} \leq \text{avgResponse(SJF)}$$

i.e. RR at best is as good as SJF, and at worst is $2X$ slower.

Shortest Remaining Time First (SRTF) If we remove the assumption that all jobs arrive together, then SJF is not optimal anymore, as it may cause a convoy effect. **SRTF** is a preemptive version of SJF, where the scheduler always runs the job with the shortest remaining time. i.e. when a new job arrives or an old one finishes, the scheduler runs the job with the shortest remaining time, even if it means preempting the currently running job. This is optimal in terms of average wait time, but it can cause starvation for long jobs.

Selfish RR Selfish RR is a variant of RR where only "old enough" processes are running in the run-queue. i.e.

- New processes wait in a FIFO queue, not yet scheduled.
- Older processes scheduled using RR.
- New processes are scheduled when either of the following happens:
  1. The run-queue is empty (i.e. no ready-to-run "old" processes exist).
  2. "Aging" is being applied to new processes (a per-process counter increases over time); When the counter passes a certain threshold, the "new" process becomes "old" and is transferred to the RR queue.
- **Fast Aging** = Algorithm resembles RR, **Slow Aging** = Algorithm resembles FCFS.

General Purpose Schedulers (Priority-Based & Preemptive)

Scheduling using priority Every process is assigned a priority. The priority reflects how "important" it is in that time instance, and it changes dynamically over time. Process with higher priority are favored, as they are scheduled before lower priority processes. The concept of priority can be also applied in batch scheduling. e.g. SJF:priority = runtime (smaller runtime = higher priority), FCFS:priority = arrival time (earlier arrival = higher priority), etc.

Negative Feedback Principle The schedulers of all general-purpose OSes employ a negative feedback policy: Running reduces priority to run more & Not running increases priority to run. As a result, I/O-bound processes (that seldom use the CPU) get higher priority than CPU-bound processes (that use the CPU a lot). This ensures that I/O-bound processes are responsive.

Multi-Level Priority Queue A multi-level priority queue consists of several RR queues, each associated with a priority. processes migrate between the queues so they have a dynamic priority, "important" processes move up (e.g. I/O-bound) and "unimportant" processes move down (e.g. CPU-bound). Priority is greatly affected by the **negative feedback principle**, i.e. by CPU consumption:

- I/O-bound $\Longleftrightarrow$ move up
- CPU-bound $\Longleftrightarrow$ move down

Some schedulers allocate short quanta to higher priority queues, and some don't or even do the opposite.

Linux $\leq$ 2.4 Scheduler

All of the definitions below are relative only to the Linux $\leq$ 2.4 scheduler, which is a preemptive priority-based scheduler that uses a multi-level priority queue. **Note:** A task is either a process or a thread.

Standard POSIX Scheduling Policies POSIX dictates that each task is associated with one of three scheduling policies:

- **"Realtime"** Policies:
  1. `SCHED_RR` (round-robin)
  2. `SCHED_FIFO` (first-in, first-out)
- The default policy:
  3. `SCHED_OTHER` (the default time-sharing policy)

POSIX defines the meaning of `SCHED_OTHER` (aka `SCHED_NORMAL` in Linux) is decided by the OS. Typically employs some multi-level priority queue with the negative feedback loop.

Realtime tasks are always favored by the scheduler. Only an admin can set a task to run with a *realtime* policy, the users can set a policy using the `sched_setscheduler` syscall.

Epoch As mentioned before,every runnable task gets allocated a quantum, which is the CPU time the task is allowed to consume before is it's stopped by the OS. Whenever all quanta of all *runnable* tasks become zero a **new epoch** begins. In this case we allocate additional running time to *all tasks* (runnable or not).

Definitions

- **Task's Priority:** Every task is associated with an integer, the higher the value the higher the priority to run. Every task has a *static* priority and a *dynamic* priority.

- **Static Priority:** A fixed value that indirectly determines the maximal quantum for the task. Fixed unless the user invokes `nice()` or `sched_setscheduler` syscalls.

- **Dynamic Priority:** Is the both the remaining time for the task and its current priority. The dynamic priority decreases over time (while running), if it reaches zero the task is preempted until the next epoch. The dynamic priority is *reset* to the static priority at the beginning of each epoch.

- **HZ:** Linux gets a timer interrupt HZ times per second, i.e. it gets a timer interrupt every $\frac{1}{HZ}$ seconds. The default value of HZ is 100 for x86/Linux2.4.

- **Tick:** A tick can mean either of the following:

  - The time that elapses between two consecutive timer interrupts, i.e. $\frac{1}{HZ}$ seconds.
  - The timer interrupt itself that fires every $\frac{1}{HZ}$ seconds.

  Ticks are used to determine the scheduler timing resolution, the OS measures the passage of time in ticks. The units of the *dynamic priority* are ticks.

- **task_struct:** Every task is represented by a *task_struct* object, which contains (among other things):

  1. nice static priority
  2. counter dynamic priority
  3. processor the last CPU core the task ran on
  4. need_resched boolean
  5. mm task's memory address space

- **task's nice** There are two types of nice, user nice and kernel nice. The **kernel's nice** is the *static priority* of the task, which is between 1...40 (higher is better) and the default is 20. The **user's nice** is the parameter passed to the `nice()` syscall, which is between -20...19 (lower is better). Values below 0 require superuser privileges.

$$\text{kernel\_nice} = 20 - \text{user\_nice}$$

- **task's counter** The *dynamic priority* of the task. It is calculated as follows:
  - Upon task creation:

$$\begin{aligned} \text{child.counter} &= \text{parent.counter}/2; \quad \text{(round down)} \\ \text{parent.counter} -&= \text{child.counter}; \quad \text{(round up)} \end{aligned}$$

  - Upon a new epoch:

$$\begin{aligned} \text{task.counter} &= \text{task.counter}/2 + \text{NICE\_TO\_TICKS(task.nice)} \\ &= \text{half of prev dynamic + convert\_to\_ticks(static)} \end{aligned}$$

  - When running: decrement each tick by 1 (task.counter–) until it reaches 0.

  The **NICE_TO_TICKS** function scales 20 (=DEF_PRIORITY) to number of tick compromising **50+ ms**. By default, scales 20 to 5+ ticks:
  `#define NICE_TO_TICKS(kern_nice) ((kern_nice) / 4 + 1)`
  So the quantum range is therefore: (recall that 1 tick = 10 ms)
  - (1/4 + 1=) 1 tick = 10 ms (min.)
  - (20/4 + 1=) 6 ticks = 60 ms (default)
  - (40/4 + 1=) 11 ticks = 110 ms (max.)

- **task's processor** Logical ID of CPU core upon which task has executed most recently, if task is currently running, then this is the core it is running on.

- **task's need_resched** A boolean flag that is checked by kernel just before switching back to user-mode. If set, check if there's a "better" task than the one currently running, and if so, switch to it. Can be thought of as a per-core rather than per-task flag as it is checked only for the currently running task.

- **task's mm** A pointer to the task's memory address space. (More details in the virtual memory lectures)

The scheduler is implemented in the `kernel/sched.c` file, and the task structure is defined in `include/linux/sched.h`. The scheduler is compromised of **4 main functions**:

1. `goodness(task, cpu)` - Given a task and a CPU core, returns how "desirable" it is for that CPU. We compare tasks by this value to determine which task to run next.

2. `schedule()` - Actual implementation of the scheduler. Uses `goodness` to decide which task to run on a given core.

3. `__wake_up_common(wait_queue q)` - Wakes up task(s) when waited-for event has happened. e.g. completion of I/O operation, or a signal being sent to the task.

4. `reschedule_idle(task t)` - Given a task, check whether it can be scheduled on some core. Preferably on an idle core, but if not then by preempting a less "desirable" task on a busy core. Note: used by the `schedule()` & `__wake_up_common()` functions.

Counter Convergence **Claim:** The counter value of an I/O-bound task will quickly converge to 2α geometric series. **Corollary:** By default, an I/O-bound task will have a counter of 12 ticks (=120 ms) as long as it remains I/O-bound and consumes neglligble CPU time.

The Drawback The scheduler is a linear scheduler, i.e. it runs in $O(n)$ time. It was replaced by the "$O(1)$" scheduler which then was replaced by the "*Completely Fair Scheduler*" (CFS) which is $O(\log n)$.

Topic 5: Synchoronization & Deadlocks

Race Condition We say that the program suffers from a *race condition* if the *outcome is nondeterministic and depends on teh timing of uncontrollable, un-synchronized events*

Memory (Cache) Coherency is the mechanism that ensures that any change to a shared piece of data in one cache is eventually propagated to all other caches, preventing processors from working with outdated information.

Cache Consistency it dictates how memory operations (reads and writes) from one processor can be observed by other processors in the system. If store operations are immediately seen by other cores and they can't be reordered with load operations, then consistency holds. (but it's not a necessary condition)

Cache **coherency** relates to a **single** memory location, while cache **consistency** relates to **multiple** memory locations. coherency ensures cores see writes in some order that makes sense, and is uniform across all cores. Consistency is about the order of different read and write operations across multiple memory locations, and how they are observed by different cores.

To enforce ordering for memory consistency, explicit **memory fence** (memory barrier) must be used. The fence makes all store-s that happened before the fence visible to all load-s after the fence, notably, flushes the store buffer to teh memory system.

Atomicy In programming, atomicity means an operation, or a series of operations, appears to happen all at once, without any interruption from other processes. This ensures that either the entire operation completes successfully, or it has no effect at all, preventing partial or incomplete results.

Critical Section A group of operations we need to atomically execute is called a **critical section**. Atomicity will make sure other threads don't see partial results! The critical section can be a uniform code across all threads, or it can be a different.

Mutual Exclusion (mutex) is a property of a critical section that ensures that only one thread can execute it at a time and never simultaneously. Thus a critical section is a "serilaization point".

Locks

Lock A lock is an abstraction that supports two operations: `acquire(lock)` and `release(lock)`. Semantically: It's a memory fence, only one thread at a time can acquire the lock and other simultaneous attempts to acquire are postponed until the lock released. Implementing a lock is nowadays done using hardware support with special instructions that ensure mutual exclusion.

Spinlock A spinlock is a lock that uses busy-waiting to acquire the lock. i.e. it repeatedly checks if the lock is available and only then acquires it. Spinlocks are typically used in low-level code where the overhead of sleeping and waking up is too high, or when the critical section is very short. If interrupt or signal handlers access the same data as the critical section, then the we must disable them, or make sure they acquire the lock too!

```c
struct spinlock {
    uint locked; // 0 = unlocked, 1 = locked
};

// The volatile keyword is crucial here. It tells the compiler that the value at addr
    can be changed by external factors (like another CPU core or hardware) at any time.
    This prevents the compiler from making optimizations like caching the value in a
    register, forcing it to read directly from memory every time.
inline uint xchg(volatile uint *addr, uint newval) {
    // atomic [oldval = *addr, swap(*addr, newval)]
    // xchg is atomic
    // lock adds a memory fence
    uint oldval;
    asm volatile("lock; xchg %0, %1"
                 : "+m" (*addr), "=r"(oldval):
                 "1"(newval):
                 "cc");
    return oldval;
}

void acquire(struct spinlock *lock) {
    disable_interrupts();
    while (xchg(&lock->locked, 1) != 0) {
        // busy-wait until the lock is available
        enable_interrupts();  // if we want to enable
        disable_interrupts(); // interrupts while spinning
    }
}

void release(struct spinlock *lock) {
    xchg(&lock->locked, 0); // set the lock to unlocked
    enable_interrupts();
}
```

Listing 1: Spinlock Implementation

Other Atomic Operations We know of **3** atomic operations that are used to implement spinlocks:

1. **xchg(addr, newval)** - Atomically exchanges the value at `addr` with `newval` and returns the old value.
2. **test_and_set(bool *b)** - Atomically sets `*b` to *true* and returns the old value of `*b`.
3. **cas(int *p, int old_val, int new_val)** - (Compare-and-Swap) Atomically compares the value at `*p` with `old_val`, and if they are equal, sets `*p` to `new_val` and returns true. Otherwise, it returns false and does not change `*p`.

Spinning or Blocking A lock can be implemented using either **spinning** or **blocking**:

- **Spin (busy wait)** - The thread repeatedly checks if the lock is available, consuming CPU cycles while waiting. This is suitable for short critical sections where the wait time is expected to be very short. Note that the user can not block context switches.
- **Block (go to wait/sleep)** - The thread goes to sleep and is woken up when the lock is available. This is suitable for longer critical sections where the wait time can be significant.

Rule of thumb: If the critical section is shorter than the time a context switch takes, then use spinning, otherwise use blocking.
Unlike spinning which can be done by the user, going to sleeping is done by a request to the OS (since it involves changing process states).

Semaphores

The Basics A semaphore is usually implemented as a counter and a queue of waiting threads. A task that announces it's waiting for a resource will get the resource if it is available or will go to sleep otherwise. In case it goes to sleep it will be awakened when the resource becomes available to it.

The fields of the semaphore are as follows:

- Value (integer) (the counter):

  If value is Non-negative, it indicates the number of available resources. If value is negative, it indicates the number of tasks waiting for the resource.

- A queue of waiting task:

  Every task that is waiting for the resource to become available is stored in this queue. When the value is negative, |value| = queue.lenght|/

The functions used to manipulate the semaphore are:

- `wait(sem)`: Decrements the value of the semaphore. If the value after decrementing is negative, the task goes to sleep and is added to the queue of waiting tasks. Otherwise, it continues execution.

- `signal(sem)`: Increments the value of the semaphore. If the value was negative before incrementing, it wakes up one task from the queue of waiting tasks.

Semaphore Implementation The semaphore is implemented in the kernel and not part of the course material. However, the following is a demonstration of the problem known as **lost wakeups**.

```c
struct semaphore_t {
    int value;
    wait_queue_t wait_queue;
    lock_t lock;
};

void wait(semaphore_t *sem) {
    lock(&sem->lock);
    sem->value--;
    if (sem->value < 0) {
        enque(self, &sem->wait_queue);
        unlock(&sem->lock);            // This is the problem!
        block(self);
    } else
        unlock(&sem->lock);
}

void signal(semaphore_t *sem) {
    lock(&sem->lock);
    sem->value++;
    if (sem->value <= 0) {
        p = dequeue(&sem->wait_queue); // Part of the problem too!
        wakeup(p);
    }
}
```

Listing 2: Lost Wakeups Problem

The problem with the above implementation is that we release the the lock before going to sleep but after joining the queue, so if another task calls `signal()` redbefore we go to sleep, it will wake us up (without being asleep) and we will exit the wait queue so *no one will be* *able to wake us up again.* This is known as the **lost wakeups** problem.

Spinlock vs. Semaphore

	spinlock	**semaphore**
wait by	spinning	sleeping
granularity	fine	coarse: might wait for a long time
complexity	lower	greater: typically uses lock
expressiveness	lower	greater: equivalent to lock if maximal value=1 (called "binary semaphore"); otherwise, counting how many resources are available
interface ordering	strict: must acquire before release, must release after acquire	relaxed: may signal(s) without wait(s), may wait(s) without signal(s)
interface dependence	strict: release only invoked by locker	relaxed: signal may be invoked by threads that didn't previously wait and vice versa

Amdahl's Law Amdahl's Law is a formula that gives the maximum improvement to an overall system when only part of the system is improved.

So what is the maximal expected speed when parallelizing?

- Let n be the number of threads.
- Let s be the fraction of the program that is strictly serial ($0 \leq s \leq 1$).
- Let T_n be the time it takes to run the algorithm with n threads.
- Then, optimally:
$$T_n = T_1 \times \left(s + \frac{1-s}{n}\right) \geq T_1 \times sem$$
- The speedup is defined as:
$$speedup = \frac{T_1}{T_n} = \frac{1}{s + \frac{1-s}{n}} \leq \frac{1}{s}$$

Part II

Overall Summary

Part III

Highlights and Notes